A Real-Time 3D Terrain Engine

being a dissertation submitted in partial fulfilment of the
requirements for the degree of Master of Science
at the University of Hull

by

Dimitrios Christopoulos

September 1999

To my parents, for hanging in there
until I finally grew up and for always
being there when I needed them.

Abstract

Terrain visualization is one of the most fascinating and interesting fields in computer graphics. Although its application areas are endless and a significant amount of research has gone into modeling and rendering of terrain, it remains one of the most challenging problem areas.

For realistic representation of terrain, millions of triangles are required. Such detail easily overwhelms even state of the art systems, which can not cope with the huge numbers of polygons needed to represent detailed terrain. Bottlenecks occur primarily in the geometry part of the graphics pipeline.

An even greater challenge is the proper texturing of the terrain. In order to recognize areas of the landscape, unique textures revealing local features (cliffs, fields, houses) are needed. These typically occupy several gigabytes and overload the system memory.

The focus of this project is the development of a terrain engine which addresses these problems, employing specialized optimization methods and algorithms to allow "low-end" computers with 3D capabilities to visualize high complexity landscapes at interactive frame rates.

"The time has come" The Walrus said.
"To talk of many things"
- L.C.Carroll

Acknowledgments

No one lives in a vacuum. Helping and learning from each other is a vital aspect of progress. Looking back at this project I would like to thank and express my gratitude to everyone who has helped me (directly or indirectly) to complete this project.

Many thanks to my project supervisor Mr. Derek Wills for his constant help and support. His enthusiasm and experience were great sources of inspiration and provided the guidance and information needed to complete the project.

I would like to thank also all the MSc staff for their help and most interesting lectures throughout the course. They introduced us to new exciting worlds.

Finally I would like to thank all my classmates and friends in Hull for making this year a good one. My very best regards to everyone of them.
This year will always be remembered as one of the most exciting and interesting years of my life.

Once again,
Thank you all.

1. Introduction

1.1 Introduction to Terrain Visualisation Systems

Computer graphics is concerned to a large extent with the geometric modelling and rendering of realistic scenes. A recurrent problem in generating realistic pictures by computers is to represent natural irregular objects and phenomena without undue time or space overhead (Fournier et al. 1982). Natural objects can only be described inefficiently and with great difficulty using Euclidean geometry (i.e. equations); thus there is a difficulty in modelling them.

One such representation, which can be seen in nature, is that of landscapes. In last few years a considerably effort and research has gone into fast and realistic representation and production of terrain.
Terrain mountain landscapes are of great importance in several areas.

- Geographic Information Systems (GIS)
- Flight and ground vehicle simulators.
- Real time battlefield simulation systems.
- Entertainment
- Tourism and travel planning
- Land planning
- Weather visualisation
- Virtual reality

These are obvious examples where the ability, to render landscapes at fast “real-time” rates, is essential.
However, for terrain of any significant size, rendering the full model is prohibitively expensive. For a realistic representation of a terrain millions of triangles are needed

which are impossible to be rendered in real-time even on high-end systems. This is even more true for low-end systems, which have widespread use and are equipped with 3D graphics capabilities, which can suffer when exposed to from high polygon counts.

Low end systems at the consumer level typically only accelerate the most important, time-critical stages of the rendering pipeline. Therefore they are tuned for the applications they were targeted for. This make them perform best at models containing low number of polygons (like games), relying on high fill rates to refresh the screen. This approach makes models with high numbers of polygons very slow, with most time spent in the polygon set-up stage of the rendering pipeline.

Most current terrain engines try to face these problems and employ specialised optimisations and algorithms to enable low-end consumer machines to visualise high detail landscapes at interactive frame rates.

Terrain remains one of the most challenging types of geometric data because it is not naturally decomposed into parts whose complexity can be adjusted independently making it very difficult to develop a specialised terrain engine on any platform and especially on low end systems.

Figure 1.1 shows the process normally performed when visualising terrain information.

This project will develop a terrain engine for low-end systems which will be able to display detailed landscapes with high triangle counts at real time frame rates. Specialised algorithms and acceleration techniques will be developed and tuned for these systems to produce a highly effective and fast engine.

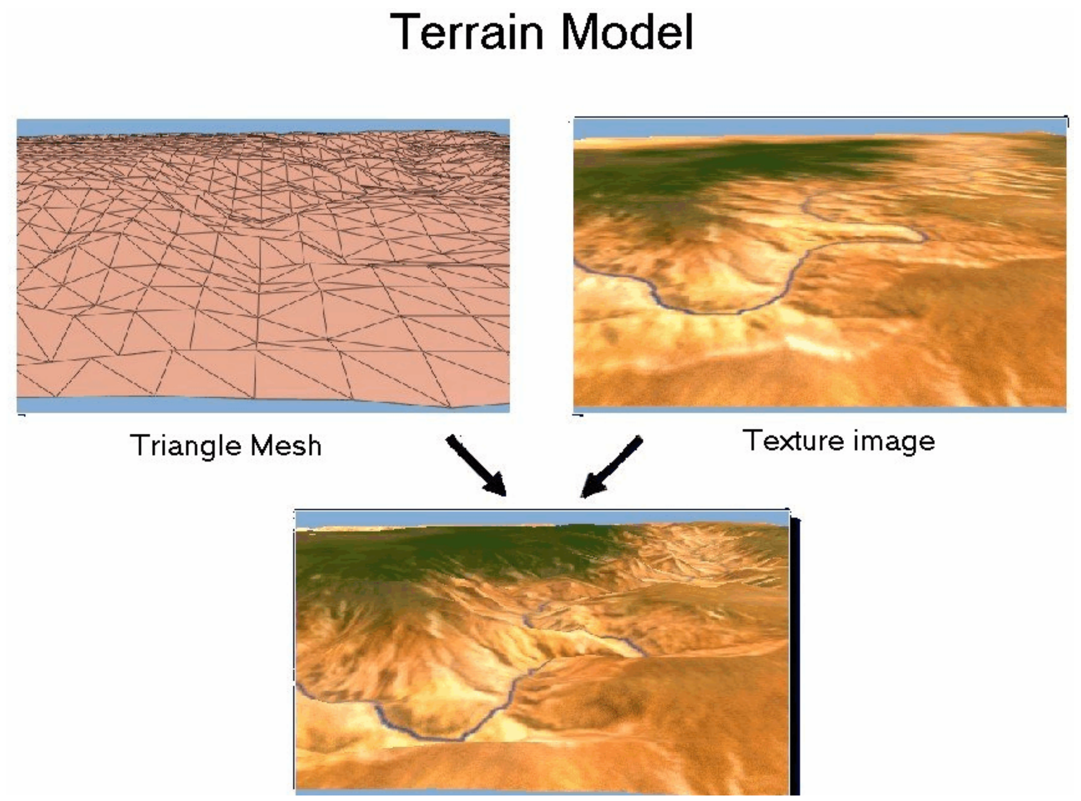


Figure 1.1 Terrain visualisation process. Source (Hoppe 1996)

1.2 Structure of the Document

The remaining document will be organised into the following chapters:

- **Chapter 2. Background**
  The main concepts and ideas behind the project are discussed. Areas such as fractals for terrain, dynamic terrain paging and optimisation techniques and algorithm for accelerated rendering of complex environments shall be covered.

- **Chapter 3. Aims and Objectives**
  The main aims and objectives of the project and the proposed work packages to achieve them are proposed.

- **Chapter 4. Design**
  The design of the software and the algorithms that are finally implemented are examined.

- **Chapter 5. Implementation**
  Technical detail about how the actual algorithms were implemented and problems involved.

- **Chapter 6. Results**
  The results and performance of the terrain engine developed.

- **Chapter 7 .Future Work**
  Areas in which this project could be extended or re-directed.

- **Chapter 8. Conclusion**

Analysis of the different parts comprising the engine and description of their advantages or limitations .

- **Appendices. Colour Plates**

2. Background

This chapter will introduce you to the world of terrain visualization. It explains what a Real-time 3D Terrain Engine is, what the difficulties in building it are, and finally the underlying theory and algorithms which can be used to build it and overcome these difficulties.

2.1 The Problem

As with many things in life almost everything, which is worthy, does not come for free and nothing is ever as simple as it looks in the first place. Although you might have heard these words of wisdom before, the field of terrain visualisation is one example where these apply magnificently. There is an amazingly large amount of ongoing research in this field and the complexity of the methods needed, even in implementing a basic system. Each aspect of terrain visualisation is an open research field on it's own.

The problem is generally speaking, all about resources. Polygons are currently the main modelling method of objects in graphics. All the real world objects are currently transformed into an number of polygons in order to be visualised. The graphics pipeline of the majority of graphics systems is built around this primitive and specialised for them. Therefore the ability of a system to perform at interactive rates (10 fps or more) or even at Real-time (30 fps or more) is dependent on its polygon throughput. Thus the question is "Has the system enough resources (polygon throughput) to handle all these polygons". The answer is frequently negative, even (expensive) state of the art systems are overwhelmed by the amount of data required. This is specially true for low-end 3D systems, which are very common and widespread and have, a limited throughput rate. This makes interactive terrain rendering on such platforms a real challenge, as complex terrain models usually need very high amounts of polygons to produce a realistic representation.

A terrain model usually consists of millions of height samples arranged in a grid, with possibly several hundred thousand polygons visible in a scene. An even larger challenge is texturing the terrain properly. In order to recognise areas of the landscape, unique textures revealing local features (cliffs, fields, houses) are needed. To cover the entire area with textures with a sufficiently high resolution to avoid blurring requires several gigabytes of textures. A terrain engine is in essence a system, which displays views of large data sets at high frame rates using intelligent and specialised rendering algorithms limiting the number of geometric primitives rendered.
It mainly consists of the following components:

- Geometry/Texture disk paging or generation.
- LOD (level of detail) for texture blocks.
- LOD for triangle geometry.
- Culling the view frustum.
- Triangle Stripping.

In the following sections the theory and algorithms behind each component and how it is related to terrain rendering are described.

2.2 Terrain Representation

To represent terrain we need real or synthetic data, which can be processed and displayed. There are two main ways to get hold of such elevation data, either it is already available and stored in a file (Digital Elevation Maps) based upon real landscapes or it is generated using a specialised algorithm as a pre-processing phase or at run-time. Frequently techniques based upon fractals (Mandelbrot 1982) are used.

2.2.1 Digital Elevation Maps

The elevation data is already stored in a file and the elevation values are usually based on real regions on earth or on other planets. Each file provides data for a restricted region and is acquired by measuring the height values for this region. The most common type of elevation data available is gridded. Other elevation data types include point elevations distributed over the area unordered and contours.
Although there are as many data types as there are providers, only two major formats available with height data for free.

- Digital Elevation Model format (DEM)-From the USGS. These range from 1:250,000 scale to 2.5 minute. The lower accuracy is available for most of the United States (USGS 1997).
- Digital Terrain Elevation Data (DTED)-From National Imagery and Mapping Agency. These range from Level 0 which is very coarse and provides wide coverage,

to the occasional 1m data, which is very rare. This data is, in general higher quality, but is really only for military users (USGS 1997).

Advantages

- Based on real world data, it therefore looks very real when rendered.
- The data is available beforehand and only needs to be loaded from file and rendered.
- Changing the file automatically changes the region represented without any other alterations.

Disadvantages

- Large storage requirements if region too big.
- Data in a file covers only a restricted area, restricting the possible movement to other directions. When additional areas are loaded from file this leads to considerable delays.
- Elevation data is not available for all regions and not always to the required accuracy.
- When approaching the ground, the terrain looks unrealistically flat.

2.2.2 Fractal Methods

The use of fractal methods does not rely on acquired elevation data but instead the data is generated using procedural techniques. This can be done either as a pre-processing phase storing the results into a file, or at run time.

Natural objects can be realistically described with fractal geometry methods where procedures rather than equations are used to model objects. In computer graphics fractal methods are used to generate displays of natural objects and visualisations of various mathematical and physical systems. A fractal object has two basic characteristics: infinite detail at every point and self-similarity between the object parts and the overall features of the object (Hearn and Baker 1992). The self-similarity properties of an object can take different forms, depending on the choice of fractal representation. We can describe the amount of variation in the object detail with a number called the fractal dimension.

Fractals can be classified into three general categories:

Self-Similar fractals have parts that are scaled down versions of the entire object. Starting with an initial shape, construct the object subparts by applying a scaling parameter ***S*** to the overall shape. The parameter ***S*** can be the same for every subpart or a random variation in which case we obtain statistically self-similar fractals. These types of fractals are commonly used to model trees (Hearn and Baker 1992).
Self-affine fractals have subparts that have different scaling parameter s_x s_y s_z in different directions. Again we include random variations and obtain statistically self-affine fractals. Terrain, water, and clouds are typically constructed using these models (Hearn and Baker 1992).

Invariant fractal sets formed with non-linear transformations including self-squaring fractals such as the Mandelbrot set (Mandelbrot 1982).

The main methods used for generating terrain using fractal methods can be summarised into the following approaches.

- Fractional Brownian Motion
  - Random Midpoint Displacement or Subdivision: Fast but less accurate.
  - Spectral Synthesis using Fourier transforms: Highly realistic but very slow.
- Physically Based Methods
  - Erosion and Hydrology: Realistic but slow and global.
  - Faulting or Collaging: Artefacts.

The most frequently used method, due to its good speed and acceptable accuracy, is the Random Midpoint Displacement method. Therefore it will be concentrated on this method which is the only one which can be applied easily in real time with very good results.

Highly realistic representations of terrain can be formed using affine fractal methods that model fractional Brownian motion. This is an extension of the standard Brownian motion which is a form of random walk that describes the erratic zig zag movement of particles in a gas or fluid as shown in Figure 2.1.

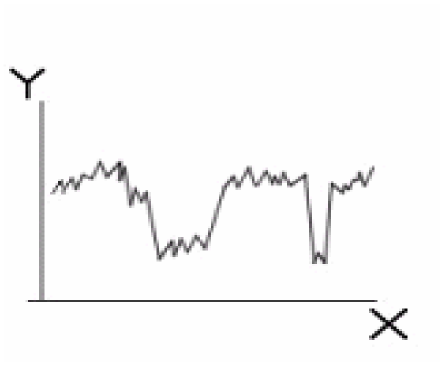


Figure 2.1 Erratic zik zak of particles. Adapted from (Peitgen and Saupe 1988)

Fractional Brownian motion is obtained by adding an additional parameter to the statistical distribution describing Brownian motion. This parameter sets the fractal dimension for the motion path. A fractal curve can be produced having a one-dimensional array with random fractional Brownian elevations, with a two dimensional array and all the elevation connected to from polygons we obtain terrain. Here the fractal dimension specifies the roughness of the terrain.

Fractional Brownian motion (fBm) is computed typically with Fast Fourier transforms, which are slow and unacceptable for real time graphics. Therefore methods have been developed which approximate fBm motion and are consequently faster.

The most popular method of terrain synthesis is recursive subdivision, commonly called “plasma” or random midpoint displacement. The idea behind this method is to take the starting shape, usually a rectangle, and place random heights at the corners. Then the heights at the midpoints of each of the sides and the center are calculated by linear interpolation and a random value gets added or subtracted from that height. This procedure is repeated recursively or in a loop where each time the next level of detail gets computed, adding smaller random values each time, until the map is complete. The random factor that is added to each point is scaled proportional to the length of the sides as shown in Figure 2.2.

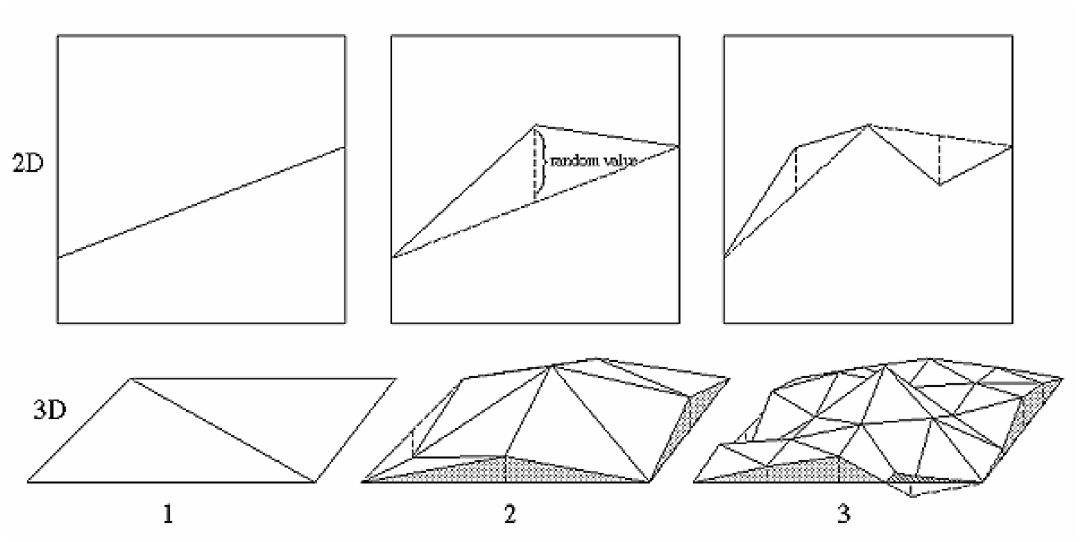


Figure 2.2 Random Midpoint Displacement. Source (Schweighofer et al. 1996)

This method can be made very fast but it suffers from a number of artefacts, usually ugly, due to the poor approximation to fBm of this method (Mandelbrot 1982). These artefacts tend to be most visible when looking down from above, or when looking across the map directly down grid lines or diagonals. There are a few techniques that are used to reduce these artefacts:

- Adding random values to old points at each level:

  Every point at each level gets a random component instead of just the newly generated points. This gets very expensive, so the efficiency of this method is sacrificed.

- Diamond-square method:

  Instead of calculating 4 edge points at the center of each square first calculate the centers of all the squares at that level of detail. Then repeat the calculation using the new centers and the endpoints of each side (the diamonds) to calculate the midpoint of the sides (Figure 2.3) (Fournier et al 1982).

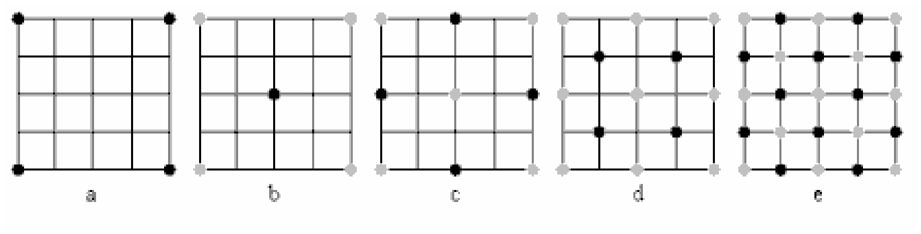


Figure 2.3 The Diamond Square Algorithm. Adapted from (Peitgen and Saupe 1988)

- Offset squares

  Create new points by subdividing each square into four smaller squares. The new points are actually offset from the old points, halfway along the line between corner and the center, instead of at the center and the midpoints of the sides. Each of the four new points is based on a weighted average of the four old points surrounding it, and each point receives a different weighted average. Notice that each point is weighted more heavily towards the old points nearest it [Miller 1986].

- Use a hexagonal grid instead of squares

  This is discussed by Mandelbrot in Peitgen and Saupes “The Fractal Science of Images” (Peitgen and Saupe 1988). It removes the unbroken straight lines that cause the obvious artefacts.

In general fractal methods have the following characteristics.

Advantages

- Minimal storage requirements when executed at run-time due to their procedural nature.
- Not restricted to an area and therefore able to produce infinite terrain.
- Flexible, all characteristics and variables on which the terrain generation is based can be altered to produce suitable terrain. The terrain can be shaped and changed until it has the required appearance.

Disadvantages

- Artefacts can occur depending on which method is chosen.
- More difficult to control detailed appearance because of their random nature.
- Can be computationally expensive depending on which method is chosen.

2.3 LOD for triangle geometry

The most common approach for rendering large scale surfaces is to exploit the traditional 3D graphics pipeline, which is optimised to transform and texture map triangles. The graphics pipeline has two main stages: geometry processing and rasterization.

Typically the rasterization load is relatively constant. In the worst case, the model covers the viewport, and the number of filled pixels is only slightly more than that in the frame buffer. Current graphics systems and even low-end systems have sufficient fill rate to texture map the entire frame buffer at a reasonable resolution (typically 800*600-1024*768) at 30-72 Hz, even with advanced features like trilinear mip-map filtering and detail textures. Instead geometry processing proves to be the bottleneck.

In order to accommodate complex surface models while still maintaining real-time display rates, methods for approximating the polygonal surfaces (Level Of Detail) and using multiresolution models have been proposed. The idea is to render progressively coarser representations of a model as it moves further from the viewer, because there is little point in rendering more triangles than there are pixels. This occurs due to perspective projection. If the geometry is distant to the viewer or forms a certain angle with the view direction, the actual projection of the geometry onto the screen will be small and could therefore be approximated to an accuracy of a few pixels by a much simpler mesh, as demonstrated in Figure 2.4.

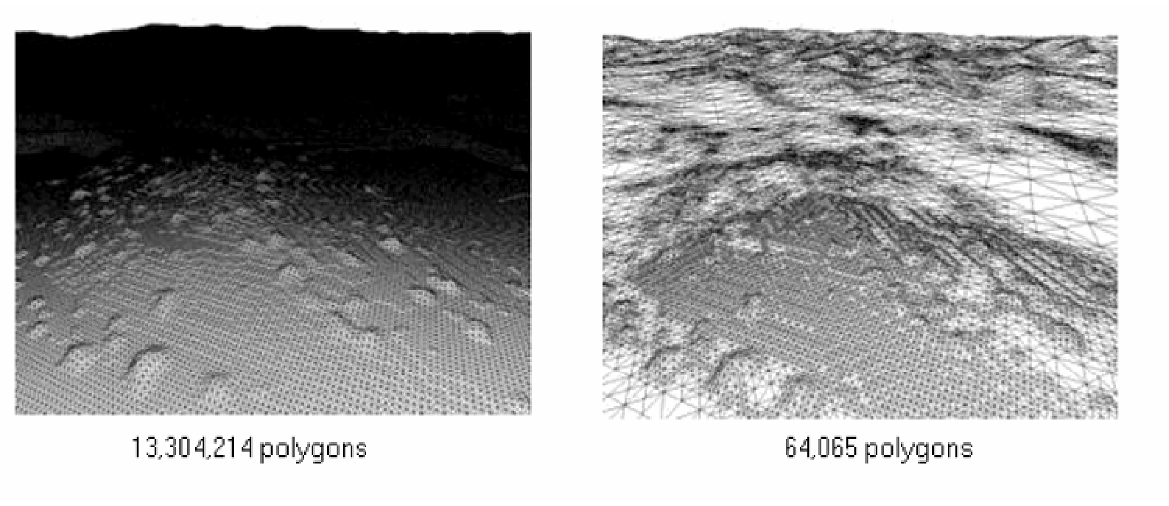


Figure 2.4 LOD applied to a surface mesh. Source (Lindstrom et al. 1996)

2.3.1 Surface simplification and Multiresolution Modelling

As early as 1976, Clark (1976) suggested using simpler versions of the geometry for objects that had lesser visual importance, such as those far away from the viewer. These simplifications are called Levels Of Detail. Simplification algorithms can be used to generate multiple surface models at varying levels of detail, and techniques are employed by the display system to select and render the appropriate level of detail model. The above procedure is called Multiresolution modelling and is based on being able to generate a range of different LOD´s for a model so as to be able to satisfy any request of the system for a particular LOD approximation.

The conditions upon which these simplifications might be used are when the object appears to be small (Funkhouser et al. 1992), when it is moving and when it is in the observer's peripheral vision.

The aim of polygonal simplification, when used for levels of detail generation is to remove primitives from the original mesh in order to produce simpler models, which retain the important visual characteristics of the original object. Ideally, the result should be a whole series of simplifications (as shown in Figure 2.5 taken from (Hoppe 1996)), which can be used in various conditions. The idea, in order to maintain a constant frame rate, is to find a good balance between the richness of the models and the time it takes to display them.

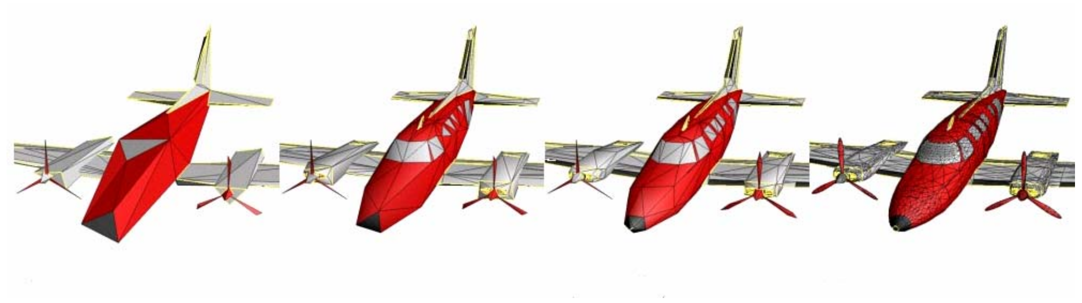


Figure 2.5 Using LOD's for distant objects. Source (Hoppe 1996)

Because the problem of selecting a minimal number of points to achieve certain accuracy has been shown to be NP-hard (Agarwal and Suri 1994) the practical methods for mesh simplification are based on heuristics. Furthermore additional criteria that simplification algorithms consider are:

- **Continuity**: Ensures that when switching from one LOD to another the difference (known as popping effect) will not be noticeable by the observer.
- **Shape Preservation**: Preserves the general shape and characteristics of the object.
- **Approximation Error**: In order to control the simplification, the approximation error should be measured locally (at each primitive). But for the user to be able to specify the simplification, a global bound should be set on the error.
- **Topology Preservation**: A change in the topology of the object must be avoided because it leads to noticeable changes.

Operations performed on the mesh in order to simplify it are typically:

- **Normalisation**: removal of degenerated faces or edges and any primitive defined multiple times.
- **Vertex Simplification**: removal of all points included within a volume. Thus, nearby points and faces are combined.
- **Edge Simplification**: removal of all edges shorter than some threshold.
- **Angle-based Simplification**: removal of edges, which form a closed angle. Conversely, edges, which are aligned, are merged.
- **Face size Simplification**: removal of all faces which have an area smaller than some threshold. Holes might be filled.
- **Face normal Simplification**: merging of all adjacent faces with near parallel normals.

These operations must be used within some kind of control mechanism in order to guide the simplification.

Garland and Heckbert (1994) have done a nice survey on multiresolution algorithms, below are their findings summarised, according to simplification operations performed and their characteristics.

- **Image pyramids**

Image pyramids are the simplest and most common type of multiresolution model used in computer graphics today. They are a successful and fairly simple multiresolution representation of raster images (Rosenfeld 1982).

- **Volume methods**

- Truly multiresolution
- Best on volume data only

Volumetric approaches to multiresolution modelling. More suitable to volume rendering than rendering in polygonal form. (Garland and Heckbert 1994).

- **Vertex decimation**

- Good quality
- Manifolds only
- Preserves topology

Vertex decimation is an iterative surface simplification algorithm. In each step a vertex is removed, all the faces adjacent to that vertex are removed and the model is retriangulated (Garland and Heckbert 1994). Relevant publications are (Schroeder et al. 1992).

- **Vertex clustering**

- Fast & General
- Hard to control
- Poor Quality

Vertex clustering is a simple method. First, the objects are subdivided into smaller boxes which contain the vertices. All the vertices falling within a single cell are cluttered together and replaced by a single representative vertex when a lower Level Of Detail is required (Garland and Heckbert 1994). Relevant publications are (Rossignac and Borrel 1992).

- **Edge Contraction**

- Smooth transitions
- Good quality

An edge contraction (or edge collapse) takes the two endpoints of the target edge, moves them to the same position and deletes one of them. New edges are build and faces that have degenerated into lines or points are removed. Typically this removes two triangular faces per edge contraction. Relevant publications are (Hoppe et al. 1993), (Garland and Heckbert 1997).

- **Simplification envelopes**

- Global Error guarantee
- Oriented manifolds only
- Preserves topology
- Difficult to construct

With this method the global error resulting from the simplification can be controlled. This is accomplished by offsetting the original surface both outwards and inwards. This defines an outer and inner envelope. By generating an approximation that lies between these envelopes, there is always an error guarantee. This naturally preserves the original model topology (Garland and Heckbert 1994). Relevant publications (Cohen et al. 1996).

- **Wavelet surfaces**

- Truly multiresolution
- Smooth manifolds only
- Topology can not change

Requires that the surface be reconstructed using a wavelet representation. This is typically difficult. Eck et al. (1995) developed a method for constructing wavelet representations of arbitrary manifold surfaces. The surface has to be remeshed before wavelets can be constructed. In addition, the topology of the model must remain fixed at all levels of detail. The wavelet representation is also unable to adequately preserve sharp corners and other discontinuities on the surface (Garland and Heckbert 1994).

2.3.2 View dependent or Continuous LOD

Having an approximated terrain model in various resolutions gives us the ability to switch between predefined LOD at run time depending on criteria such as distance, roughness, and screen projection. Although this might be all that is needed for object visualisation for terrain the case is more complicated. Because terrain usually stretches out in the distance different levels of resolution are required in the same representation. Regions near the viewer must always be represented at the highest detail and regions which are distant or have a small screen projection, can be represented in the same frame with lower detail. It is obvious that being able to represent the surface only with one LOD per frame would cause either inaccuracies in near regions if a low LOD was used or redundant polygons for far regions if a high LOD was used.

Terrain representation remains one of the most challenging areas because it is not naturally decomposed into parts whose complexity can be adjusted independently, and because the qualities required by a triangulation is view dependent. Finding such a mesh, and updating it as the viewing parameters change, is referred to as view dependent or continuous LOD control. The challenge is to locally adjust pixel tolerance while maintaining a rendered surface that is both spatially and temporally continuous. To be spatially continuous the mesh should be free of cracks and T-junctions. To be temporally continuous, the rendered mesh should not visibly “pop” from one frame to the next.

Several schemes have been developed to address view dependent LOD control and are summarised into groups in Table 2.1. In essence these algorithms perform the same operations on the mesh and use also the same criteria as their LOD or Multiresolution counterparts. Instead of simplifying the whole terrain at once they refine or combine regions from frame to frame so as to achieve different LOD’s over the same terrain and smooth transitions.

- **Pre-computed**: all possible meshes are available at runtime and combined or switched in or out.
- **Run Time**: mesh is created/edited every frame.

	Regular grid Advantage: less storage, simpler code, faster geometry queries, easily extendable texture mapping.	**Triangulated irregular Network (TIN)** Advantage: Smoother terrain with less error and approximations with less triangles
Pre-computed		▪ (Garland and Heckbert 1995) ▪ (Schroeder and Rossbach 1994) ▪ MultiGen
Run Time	▪ (Lindstrom et al. 1996) ▪ (Duchaineu et al. 1997) ▪ (Miller 1995) ▪ (Falby et al. 1993) ▪ TopoVista	▪ (Hoppe 1997) ▪ (Cohen and Levanoni 1996) ▪ (De Berg and Dobrindt 1998) ▪ (De Floriani and Puppo 1995) ▪ (Xia et al. 1997) ▪ (Willis et al. 1996)

Table 2.1 Comparison of different approaches

Triangulated Irregular Network (TIN)

Much of the previous work on polygonalization of terrain surfaces has concentrated on triangulated irregular networks. A number of different approaches have been developed to produce TIN's using Delaunay and other triangulation's (Garland and Heckbert 1995), (Schroeder and Rossbach 1994). Many representations that have been proposed lend themselves to Real Time LOD.

Willis et al. (1996) describes a hierarchical triangulated irregular network (TIN) data structure with "near/far" annotations for vertex morphing, along with a queue driven top-down refinement procedure for building the triangle mesh for a frame. No advantage is taken from frame to frame coherence.

Xia et al. (1997) and Hoppe (1997) give similar methods for interactive, fine-grained LOD control of general TIN meshes based on view dependent refinement of pre-processed progressive mesh representations (Hoppe 1996). The simplification is done by applying an edge collapse operation, and the result is a simplified mesh and a series of vertex splits which are an inverse of edge collapses and are used to introduce details into the base mesh.

Several methods that use Delaunay triangulations have been proposed by Cohen and Levanoni (1996), De Berg and Dobrindt (1998) and De Floriani and Puppo (1995). In particular by Cohen and Levanoni (1996) support on-line view dependent LOD with temporal coherence, but must resort to “two-stage” geomorphs which will avoid the popping effect. Compared to quad trees and bintrees these methods allow more general distribution of vertices over the domain. However the mesh connectivities are again constrained, in this case by the Delaunay criterion.

In general TIN’s allow variable spacing spacing between vertices of the triangular mesh, approximating a surface at any desired level of accuracy with fewer polygons than other representations. However, the algorithms required to create TIN models are generally computationally expensive.

Regular Grid

Regular grid surface polygonalizations have also been implemented as terrain and general surface approximations.

Miller (1995) uses a quad tree to pre-process a height field defined on a uniform grid. In a pre-processing phase, vertices at each quad tree level are computed using an approximate least squares fit to the level below. For each frame at run time, a priority queue drives quad tree refinement top-down from the root, thus allowing triangle counts to be achieved directly.

Lindstrom et al. (1996) and Duchaineu et al. (1997) who choose triangle bintree meshes for representation, obtain high frame rates for large output meshes using a bottom-up vertex reduction methodology enhanced by an elegant block LOD algorithm. Advantage of frame to frame coherence is taken.

Falby et al. (1993) uses again a quad tree representation for constructing multiple level of detail. Lower LOD’s are produced by discarding every second vertex from the higher LOD, the criterion used as to when to use each level of detail is distance.

Regular grid representations generally produce many more polygons than TIN's for a given level of approximation, but grid representations are typically more compact. They also have the advantage of allowing for easier construction of a multiple level of detail hierarchy. Simply sub sampling grid elevation values produces a coarser level of detail, whereas TIN models generally require complete re-triangulation in order to generate multiple levels of detail.

Of course there are other hybrid methods and representations these techniques Figure 2.6 shows a typical top down view of the various representations for each method .

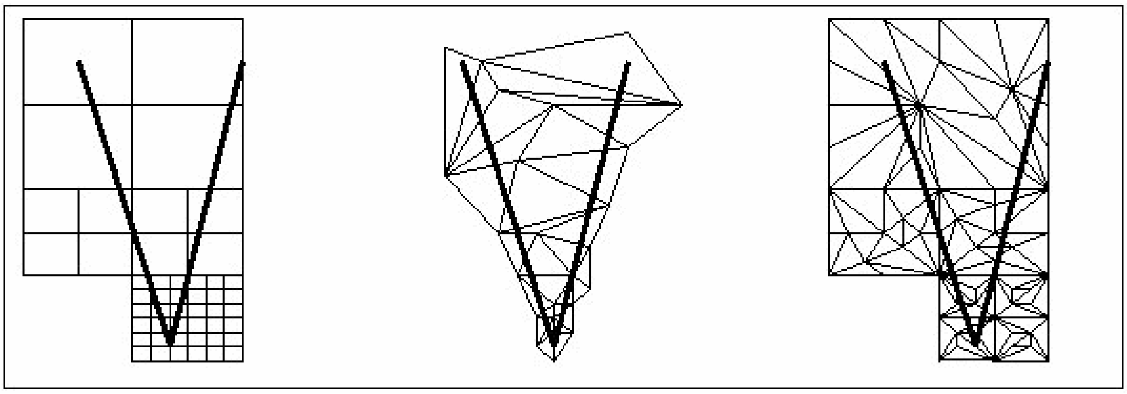


Figure 2.6 Typical Regular Grid, TIN and Hybrid representations.

For typical example frame generated by these approaches is illustrated in Figures 2.7 and 2.8, which are taken from (Hoppe 1997) and (Duchaineu et al. 1997). Both show a birds-eye view of the mesh. Neighbourhoods that are flat, distant or outside the view frustum are triangulated more coarsely than close rough neighbourhoods. Figures 2.7 shows clearly that despite the large LOD approximation in various areas of terrain the actual view remains without loss of quality.

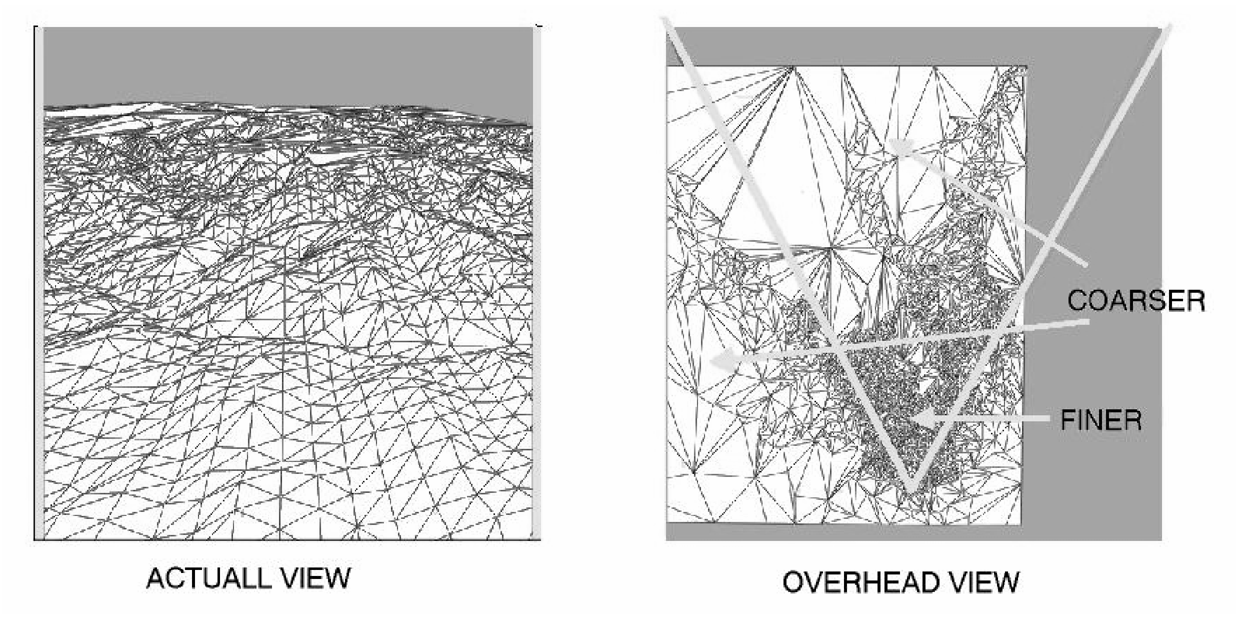


Figure 2.7 View Frustum Culling and LOD applied to terrain mesh. Source (Hope 1997)

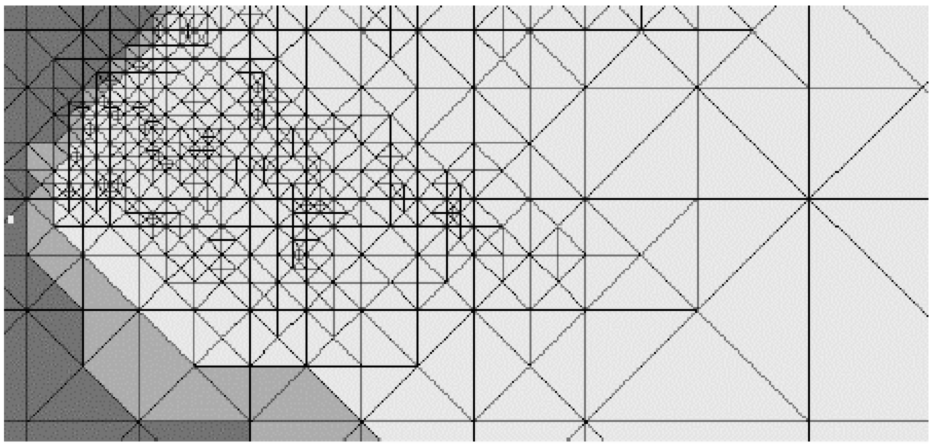


Figure 2.8 Triangulation example frame, with eye looking right. Dark grey region is outside the view frustum, light grey is inside and medium grey overlaps boundary. Source (Duchaineu et al. 1997)

2.4 Texture Mapping

Texture mapping is used in the model to give a more photo-realistic and detailed appearance. Most systems today support hardware accelerated texture mapping. Implementing texture mapping in software is very expensive and is therefore mostly not feasible.

Most hardware implementations of texture mapping require the texture data to be resident in a specially designed memory connected directly to the 3D chipset. This memory is called texture memory and especially in the case of low-end systems is a very sparse resource. Using unique textures (phototexturing) to represent the underlying model requires large amounts of texture data.

2.4.1 Resolution and Colour depth

The main factor determining the amount of data needed is the resolution and the colour depth needed. The resolution should ideally be set to meet the requirements of the application, or as close as is allowed by the system. If the application mostly has high altitude view points, low texture resolutions may be used, as detail will be lost by rendering multiple texels onto the same pixel. If the view point is close to the ground then high resolution textures are necessary to avoid blurring.

Colour depth determines the available colour range, which can be displayed by the textures. The RGBA representation is most common, requiring 3 intensity components (Red, Green, Blue) and one alpha component. Fewer intensities may be used (8 bits yielding 256 intensities) to decrease the memory consumption of each texture. More information about intensity and colour representation can be found in (Foley et al. 1990).

2.4.2 Terrain texturing approaches

In general there are three main approaches to texturing terrain

- Phototexturing
- Detail/Noise textures
- Generic textures

Phototexturing

In this case the model is covered with textures being actual photographs of the object. These photographs are in the case of terrain rendering, high-resolution aerial photos of the area. Phototexturing is only feasible on low-end systems with low resolutions textures, because of rapidly increasing size of the data set with higher resolutions. As an example lets assume the model covers an area of 10x10km, the terrain is represented with 1 texel per square meter and the texture is RGB coded with 8 bits of intensity for each component. Then 10.000x10.000x3= 294 MB would be required.

Detail/Noise Textures

This method can be applied when the size of the phototextures is too large to fit in the memory at once. Use a single texture, essentially noise over the terrain and then colour the vertices of the polygons. Then, when close multipass that texture with your actual texture (using the alpha values at the vertex's to blend in the texture smoothly), over a set number of frames blend the two from 100% vertex/noise to 100% texture. Then use another pass to blend in a smaller, detail noise texture to add noise at the pixel level to avoid the big blur syndrome.

Generic Texturing

If an actual phototextured representation of the terrain is not required or feasible, generic textures may be an appropriate solution. Generic texturing means using a set of different textures for different areas of the terrain model. Thus with 8-10 textures for different terrain types, it should be possible to create a fairly accurate representation of the model. A problem with this kind of texturing is making good transitions from one texture type to another. This can be solved using specialised transition tiles where the two adjacent tile are mixed and blended into each other (Woo et al. 1997) as shown in Figure 2.9.

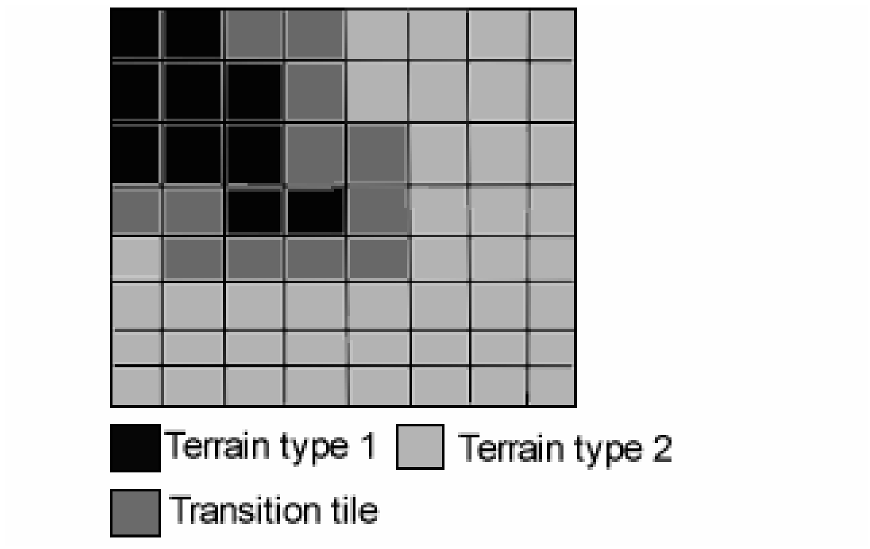


Figure 2.9 Transition tiles between two texture types

2.4.3 Detail Management

Strategies similar to that used for LOD reduction in polygon resolution can be used to determine texture resolution as well. Lindstrom et al. (1995) propose an algorithm where texture resolution is based on distance and viewing angle of the viewer to a texture block of geometry.

Distance Based Texture Resolution

For the distance-based resolution, a series of cut-offs for texture resolution are defined. The resolution is decreased the further away a texture block is. These cut-offs distances should be chosen carefully because the image quality is affected to a greater extend by colour cues than by spatial cues associated with the terrain geometry. When the error introduced is significantly larger than one pixel, the scene appears blurry as each screen pixel does not map to a unique texture pixel (texel).

Viewing Angle Based Texture Resolution

Resolution is also set by inspecting the angle between the normal of the polygon and the view direction. Polygons that are rotated away from the viewer occupie less area on the screen than if they are perpendicular to the line of sight. As the polygon rotates away from the viewer lower texture resolutions may be used to render the polygon. Thus when polygons are viewed from the side, a lower texture resolution can be used.

Mip Mapping

Textures primitives can be viewed at any distance like any other object in the scene. When textured objects move further away from the viewpoint, the texture map must decrease in size along with the projected image of the object. To handle these situations the visualisation API has to filter the texture map down to an appropriate size in order to map it onto the object, without disturbing artefacts. These are not always avoidable, because as the object moves away from the viewpoint the texture must be scaled and filtered and may appear to change abruptly at certain transition points. Furthermore all these operations are too time consuming to be properly performed at run time.
These artefacts and the time consuming operations at run time, can be avoided by specifying a series of pre-filtered texture maps of decreasing resolution, called Mip-Maps (Williams 1983). Mip is short for “multim im pravo”, which means “many things in a small place”. Mipmapping helps minimise the amount of texture data needed to render a scene, since the mipmapping algorithm determines which texture map is to be used on the size (in pixels) of the primitive to be mapped. Figure 2.10 shows a typical Mip-Map pyramid generated for a texture object. Therefore Mip-Maps

- Match the level of detail in the texture maps to the image that is drawn on the screen. For a smaller primitive lower resolution texture is used.
- Avoid shimmering and flashing (aliasing artefacts) in the textures of an object as it moves.

The OpenGL visualisation architecture supports hardware accelerated texturing, multitexturing, and mipmaping.

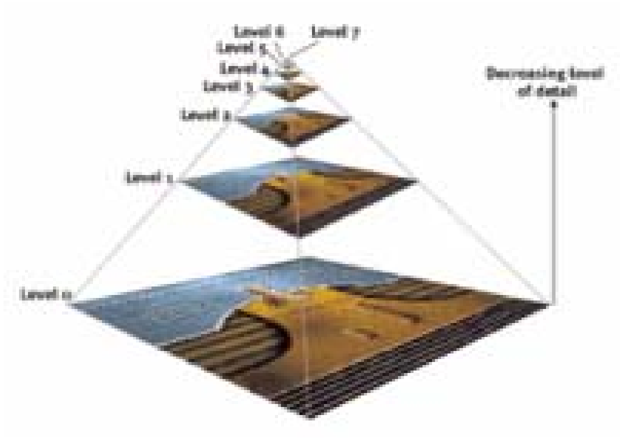


Figure 2.10 A Mip-Map Pyramid. Source (Flavell 1999)

2.5 Culling

LOD management alone is often not enough to achieve the fastest possible rendering of a model. Visibility culling algorithms attempt to avoid drawing objects that are not visible in the image. This approach was first investigated by Clark (1976) who used an object hierarchy to rapidly cull surfaces that lie outside the view frustum. View frustum culling techniques are most effective when only a small part of the scene's geometry is inside the view frustum. Airey et al. (1990) and Teller (1992) described methods for interactive walkthroughs of complex environments that compute the potentially visible set of polygons. Both of these methods require a lengthy pre-processing step for large models. More recently Luebcke and Georges (1995) developed a dynamic version of this algorithm that eliminates the pre-processing. Such methods can be very effective for densely occluded polyhedral environments, such as building interiors, but they are not suited for mostly unoccluded outdoor scenes.

If we could avoid overloading the rendering pipeline with polygons, which would be hidden/obstructed, significant rendering time could be saved. On the other hand if the computation to find out these polygons causes too much overhead the rendering time would not be optimal. Generally we want to keep a balance between computation time and number of rejected polygons.

2.5.1 View-Frustum Culling

Typically the viewing volume of the observer in a scene is called the “View-Frustum”. It consists of six planes (near, far, right, left, top, down) and forms a convex polygon. View-frustum algorithms use spatial data structures and statistical optimisation (bounding volumes) in order to determine quickly if a geometry is inside the frustum. If a node of the structure or a bounding box enclosing polygons overlaps with the view frustum then it is rendered otherwise it is rejected. Figure 2.11 shows an example of such techniques. The scene is cut up into blocks each block, which is not inside the viewing frustum (grey blocks), is not rendered. Assarsson and Moeller (1999) have done extensive research on this topic.

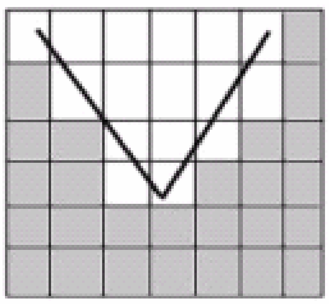


Figure 2.11 View-Frustum culling example

2.5.2 Backface culling

This form of culling is the most widely used and well documented. It based on the fact that in most polygonal objects the viewer never sees the backside of a polygon, so we can safely cut away all polygons, which face away from the viewer. If a polygon faces away is simply determined by the angle, which forms the normal of the polygon with the view direction vector (if less than 90 degrees it is accepted otherwise it is rejected), typically a dot product is performed. If the terrain has mountains this form of culling could also

result in great speed increases, whereas if the terrain is flat almost all polygons can be seen and the overhead compared to the speed increase rises.
Because this operation is done on a per polygon basis it is more expensive and should only be performed after view-frustum culling.

2.5.3 Occlusion Culling

Occlusion culling involves computation of set of polygons that are within the viewing frustum but are not visible from the current viewpoint. This could be because some other geometry (large mountain, or building) is in the line of sight and hides all the otherwise visible area behind it. As in view frustum culling it relies on spatial data structures and statistical optimisation, and can reduce further more reduce the number of polygons send down the pipeline. Hudson et al. (1997) presents some object space techniques on this subject. In general these kinds of algorithms are divided into two partially sub problems.

- Select a set of occluders to use for the given viewpoint. This can be done either at real time, or as a pre-processing task.
- Use them to cull away occluded portions of the model.

Any combinations or primitives can be used as occluders, however usually they are restricted in order to ease the culling to be convex or a union of two convex objects. These algorithms are very difficult and complex to implement compared with the other culling techniques, therefore they are implemented only when it is really necessary for the application or system. Figure 2.11 shows an ideal scene for occlusion culling, the grey areas are culled away by the view-frustum, the light grey are region culled away by the occluder and the white regions are the only ones which will be rendered.

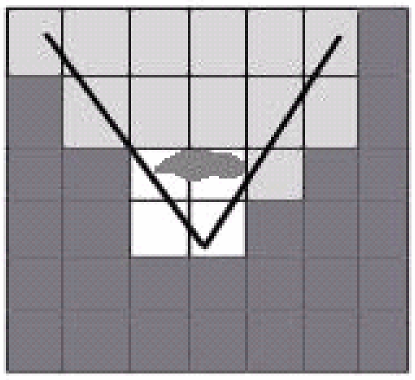


Figure 2.11 Occlusion culling

2.6 Triangle Strips and Spatial Data Structures

After covering the algorithms which accelerate the rendering of the geometry, it is time to review some optimisations, which vary form platform to platform, and the spatial data structures which support these algorithms.

2.6.1 Triangle Strips

Almost all scientific visualisation involving surfaces is currently based on triangles. To speed up the rate at which the current systems can visualise these models is also essential to be careful how they are triangulated. Partitioning polygonal models into triangle strips as shown in Figure 2.13 can significantly reduce rendering times over transmitting triangles individually. This is because although we specify the same number of polygons, using triangle strips a considerably smaller number of triangles/vertices are sent down the pipeline.

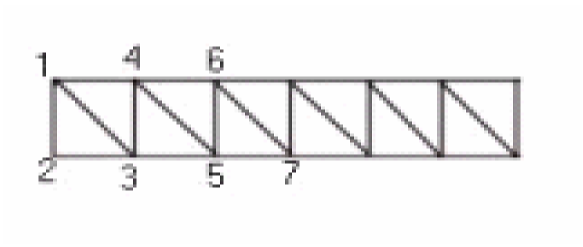


Figure 2.13 Triangle Strips

In a set of triangle strips each triangle shares an edge with its previous one, in this way instead of transforming 3*n (n:number of triangle) vertices, the renderer transforms 2+n vertices. Evans et al. (1996) presents some efficient algorithms on this subject. Most approaches pre-process the object to form triangle strips, in the case of terrain with LOD where vertices get inserted and deleted at run time and the triangulation is not predefined for the whole object, care must be taken when choosing the triangulation algorithm. Again there is a trade off between constructing an optimal triangle strip and the overhead it causes.

2.6.2 Quadtrees

A quadtree is derived by successively subdividing a 2D plane in both dimensions to form quadrants as show in Figure 2.14. When a quadtree is used to represent an area in the plane, each quadrant may be full, partially full, or empty (also called black, gray, white respectively), depending on how much of the quadrant intersects the area. A partially full quadrant is recursively subdivided into subquadrants. Subdivision continues until all quadrants are homogeneous (either full or empty) or until a predetermined cut-off depth is reached. Whenever four sibling quadrants are uniformly full or empty, they are deleted and their partially full parent is replaced with a full or empty node. As we can see in Figure 1, any partially full node at the cutoff depth is classified as full. The successive subdivisions can be represented as a tree with partially full quadrants at the internal nodes and full and empty quadrants at the leaves.

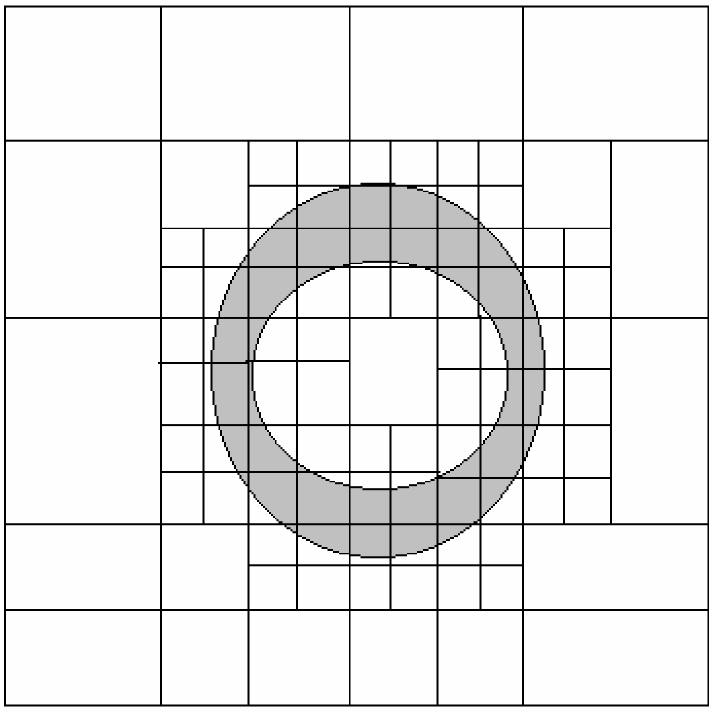


Figure 2.14 Quadtree subdivision

If the criteria for classifying a node as homogeneous are relaxed, allowing nodes that are below or above some threshold to be classified as full or empty, then the representation becomes more compact, but less accurate. The octree is similar to the quadtree, except that its three dimensions are recursively subdivided into octants.

These data structures are not only used to store some geometry efficiently, but they also enable fast execution of operations. These operations include:

- Neighbour finding techniques.
- Set Operations (union, intersection).
- Transformations (translation, scaling, rotation).
- Perimeter computation.
- Component Labeling.

Although Quad and Octrees started as methods for set operations and space saving, they can
be applied efficiently to terrain rendering and storage. The storage process is different in the case of terrain data, which are continuous. In the case of terrain we do not have empty or full nodes, each quadrant represents a portion of the whole terrain and its 4 children occupy the same area as their parent. This is shown in Figure 2.15.

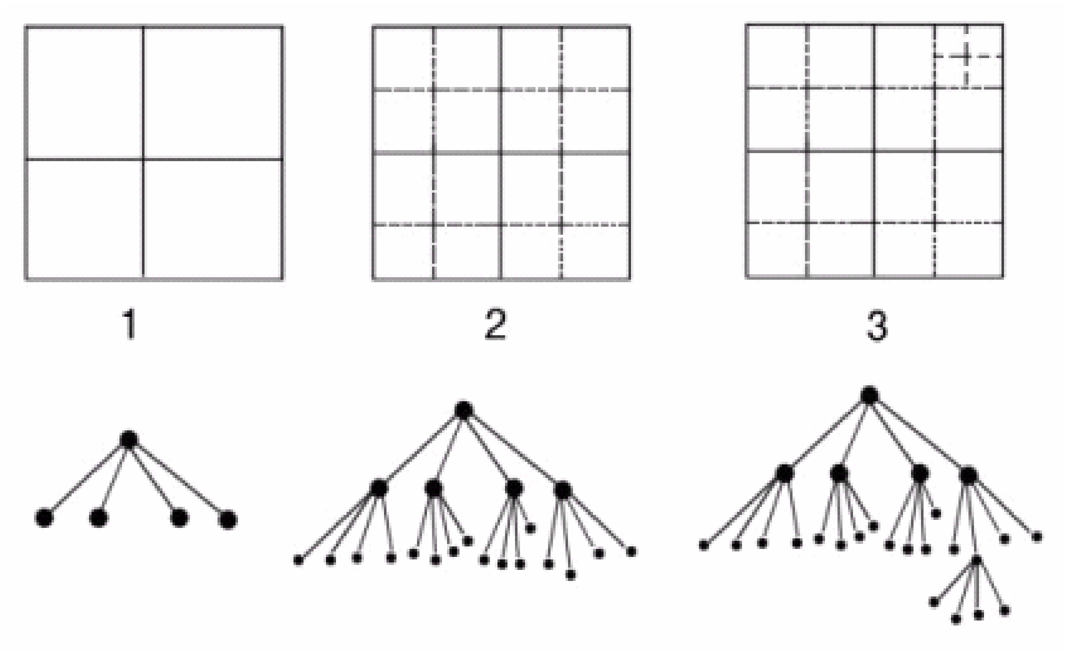


Figure 2.15 Quadtree Data Structure

This representation is not only compact which allows for easy terrain paging/caching but also provides a very convenient way to perform LOD. As can be seen in Figure 2, the quadtree has various levels of nodes before the leaves, which represent the maximum resolution. Each level in the tree structure represents the same area with a different resolution. Thus if the area is distant we can represent it safely using lower resolutions (i.e level 1 in Figure 2), and if it is near, using higher or the maximum resolution (i.e. level 3 in Figure 2).
Problems can arise (cracks) when neighbouring regions are represented with different resolution. This can be overcome either by introducing additional polygons, which will cover these cracks or by the use of a restricted quadtree (Herzen and Barr 1987).

Furthermore because its compactness it is very easy to pre process the whole terrain into regions covered by quadtrees store them on disk and load them into memory once they are needed. Frustum clipping is performed very rapidly with the use of quadtrees. Once the parent is outside the frustum all its children are marked as outside too and thus whole regions can be rejected without any expense.

2.6.2 Octrees

Octrees are extension of quadtrees and are used to store 3D volume data. Each node has eight children and terminal nodes have none. An octree is developed by recursive decomposition of a three-dimensional object, each octant is further subdivided into smaller octants.

An octree can record which subvolumes or voxels are occupied by an object. Therefore it provides an approximation of the object, the accuracy determined by number of recursions done when generating the octree structure for an object.
Octrees can also provide a spatial index to a 3D object. Each node would contain the polygons for that particular area represented by the node. Like quadtrees octrees can be used for LOD computations of 3D objects.

2.6.3 BSP Trees

Binary Space Partitioning Trees recursively divide an object or space using dividing planes (Fuchs et al. 1980). While octrees and quadtrees effectively do this based on a regular pattern, the dividing planes in BSP trees may be arbitrary planes in 3D space. Figure 2.17 shows a BSP tree representation of a room in two dimensions. In this example the dividing planes are perpendicular or parallel, but this does not have to be the case.

BSP trees have application primarily in architectural walkthroughs (Airey et al. 1990). The floors and walls can be represented as plane in the BSP tree, and thus a small Potentially Visible Set (PVS) of polygons can be generated. This partitioning can also be used to divide data so that all of it need not be loaded into memory the same time.

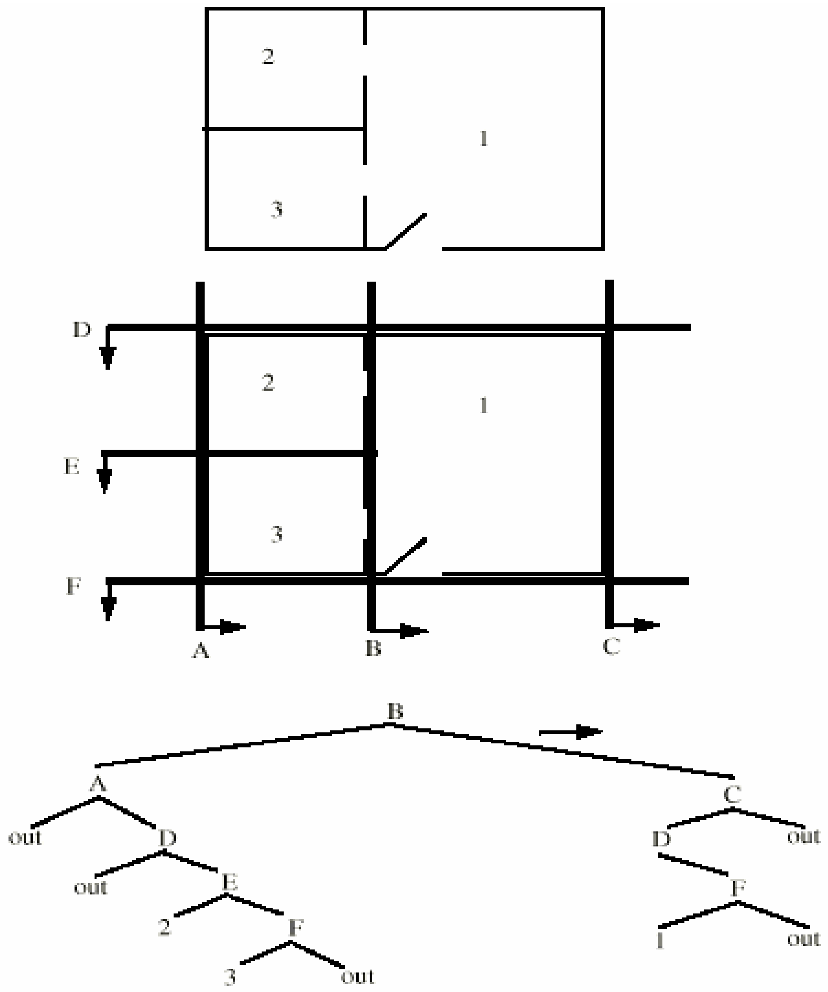


Figure 2.17 BSP tree representation of a room taken from (Fuchs et al. 1990)

3. Aims and Objectives

This chapter briefly outlines the core aims and objectives of the project, and discusses which areas of terrain rendering shall be concentrated on.

3.1 Project Aims

A Greek philosopher said once “Everything moves”. He is right the whole world around us moves and the probability is that we are moving with it. Sometimes though it is good to stop moving and start listening around you, maybe then you will find an aim in what you are doing.

The last years have witnessed a boom in field of 3D graphics, both in software and in hardware. Almost every few months more powerful systems with even more impressive graphics capabilities are made available. Due to this new systems algorithms and methods can be investigated and implemented which weren’t feasible before. One of the primary goals which have become feasible is also the representation of natural objects such as terrain and landscapes in real time. The primary difficulties encountered in such an undertaking are, as explained in the previous chapters, the bandwidth and the resources.

Bandwidth is ability to produce/gather the actual height data. Memory and access times to peripherals limit the size of the actual area which can be displayed and loaded in at a particular time. The user is constrained either by the size of the area, or by large loading times when loading from disk additional data. Therefore intelligent and efficient paging mechanisms are needed.
Procedural modelling and fractal techniques are a solution to this problem and have been applied in the past successfully for the solution of similar problems. The use of such techniques introduces additional problems but the advantages are great. The above facts and because there are less solutions available that use these techniques in the field of

“real time” 3D landscape generation and display (the majority loads up the terrain from file), we will use them as our main method for landscape generation.

Resources are always sparse when it comes to real time display. Therefore appropriate techniques for accelerating rendering have to be investigated and implemented. Level Of Detail techniques for geometry and textures, frustum culling, data structures and triangulation techniques must be implemented efficiently, and will be the main basis of our system. Furthermore the application of these techniques and their combination with procedural (fractal) techniques into one system will be one of the most interesting aspects of this project.

This project will try to continue where others in the past have stopped due to immaturity of the hardware and algorithms they used. The goal is to show that an implementation of real-time systems, which display large 3D landscapes, is feasible and can produce very good results even on low-end 3D systems.

Thus the main aim of the project is:

- Real Time Terrain Rendering on low-end 3D systems

3.2 Project objectives

The overall aim for the project may be broken down into the following objectives:

- Implementation and research of fast fractal terrain generation techniques.
- Implementation of suitable data structure to efficiently store, retrieve and update the terrain in real-time. The structure should be suitable for fractal terrain generation and accommodate for infinite terrain.
- Review suitable rendering acceleration techniques such as LOD and frustum culling.
- Implement and adjust techniques for fractal terrain generation and infinite terrain (most techniques work on a static mesh).

- Implementation of suitable triangulation and system optimisation for fast display of terrain.

Extensions

- Implementation of Environmental aspects and natural phenomena such as clouds, water, atmospheric attenuation and agricultural detail (plants).

3.3 Project Plan

The final outcome of this project will be an interactive display system of terrain. Therefore a computer simulation and walkthroughs of terrain will be presented. Furthermore various data structures and ways of encoding and retrieving terrain information will be tested and analysed. The final structure must meet specific criteria and conditions, which make it suitable for terrain display and encoding. Criteria and techniques will be found to compute and apply LOD to the mesh depending on the view position. Techniques like frustum culling and triangle stripping will be investigated and implemented in time efficient ways. Fractal terrain generation techniques will be researched which produce realistic looking terrain under the given time constraints.

The proposed different stages of the project will be.

1. Background and Basic understanding of fractal terrain generation techniques, and methods for accelerated rendering (LOD, frustum culling, caching, spatial data structures etc.)
2. Implementation of fractal terrain generation technique and visualization of produced terrain without and acceleration methods.
3. Incorporation of methods which speed up rendering dramatically so as to produce interactive simulation. Methods probably employed will be a spatial hierarchy for efficient storage, retrieval and frustum culling taking into account frame to frame coherence (ie. Quadtree), caching and triangle strip generation.
4. Implementation of environmental aspects.
5. Computing speedup and efficiency of applied methods and testing of simulation.
6. Complete the writing of the dissertation.

The system will be programmed using C/C++ programming language. The compiler used will be Visual C++ 5.0, which produces 32 bit applications for the Windows 95/NT operating systems. OpenGL will be used as a graphics library with the glut library providing the user interface routines.

4. Design

The previous chapters have summarised most of the work done in the area of terrain visualisation systems and some of their related subjects. This chapter shall be outlining how the aims and objectives are intended to be achieved. Part 1 of this chapter will cover the desired functionality and requirements of the system. Part 2 will discuss the proposed algorithms that are intended to be implemented.

4.1 Requirements

Before starting analysing and suggesting we have to focus once again on the functionality that our system has to incorporate.

- Terrain Generation

Most terrain engines, because of unavailability of data in memory restrict the movement of the user to certain areas, or have significant loading delays as new geometry is paged in. One objective of this system is to avoid these problems and not restrict the user to an area, but to allow navigation in any direction desired without any restrictions or loading delays.

- Appearance

The terrain should be produced fast and avoid artificial artefacts, it should mimic natural terrain as much as possible. Furthermore natural colouring and texturing should be applied to the geometry rendered so as to provide a natural appearance. Environment features such as clouds, sea, waves, fog should be available for experimentation.

- Rendering

In order to show detailed terrain the system should be able to cope with large amounts of geometry and textures. These should be handled intelligently and reducing their size in order to enable the system to render them in real time. However the reduction in complexity should not be noticeable by the user.

- Feedback and control

Feedback provided by the rendering engine should number of frames per second (fps) displayed, and the mount of geometry (polygons) rendered. There should be options to experiment (shut on or off) with various rendering options such as Level Of Detail or Culling and observe their effect on performance.

4.2 Terrain Generation and Paging

The backbone of every terrain engine is the terrain generation and paging algorithm. As seen from previous chapter the terrain generation is based on fractal algorithms. This approach has the advantage of practically unlimited terrain generation (we can generate as many height values as needed) and minimal storage requirements. The disadvantage is that it is difficult to control.

4.2.1 Fractals

Although there are many fractal algorithms available we have to make a trade off between execution speed and approximation (artefacts).The implementation is based on an algorithm proposed by Fournier et al. (1982) which is one of the fastest available and produces very good results. This algorithm is known as "The Diamond-Square algorithm" based on the order it visits the points when producing the terrain. Every point for which we require is visited only once, whereas other algorithms need multiple passes. Below we will explain the actual algorithm and how it is applied to terrain rendering.

The Diamond-Square algorithm

Although based on a complex theory the basic algorithm as it is applied to fractal terrain generation is simple. Essentially what we do is generate a coarse initial random terrain. Then we will recursively add additional random details that mimic the structure of the whole, but on increasingly smaller scales.

These are the steps we apply to build our fractal terrain:

1. We first assign a random height to the four corner points of a grid (a in Figure 4.1).

2. We then take the average of these four Corners, add a random perturbation and assign this to the midpoint of the grid (b in Figure 4.1). This is called the diamond step because we are creating a diamond pattern on the grid. (At the first iteration the diamonds don't look like diamonds because they are at the edge of the grid; but one look at Figure 4.1 will make this step clearer).

3. We then take each of the diamonds that we have produced, average the four corners, add a random perturbation and assign this to the diamond midpoint (c in Figure 4.1). This is called the square step because we are creating a square on the grid.

4. Next, we reapply the diamond step to each square that we created in the square step, then reapply the square step to each diamond that we created in the diamond step, and so on until our grid is sufficiently dense.

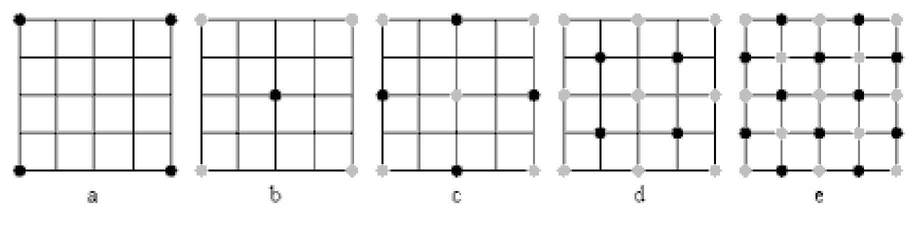


Figure 4.1 Steps of the diamond square algorithm. Adapted from (Peitgen and Saupe 1988)

An obvious question arises: How much do we perturb the grid? We need to know about the roughness coefficient. This is the value, which will determine how much the number range is reduced each time through the loop and therefore will determine the roughness of the resulting fractal. Normally we use as a roughness coefficient a floating point number in the range of 0.0 to 1.0. Lets call it H, $2^{(-H)}$ is therefore a number in the range 1.0 (for small H) to 0.5 (for large H). The random number range can be multiplied by this amount each time through the loop. With H set to 1.0, the random number range will be halved each time through the loop, resulting in a very smooth fractal. With H set to 0.0

the range will not be reduced at all, resulting in something quite jagged. Figure 4.2 shows three ridgelines, each rendered with varying H values.

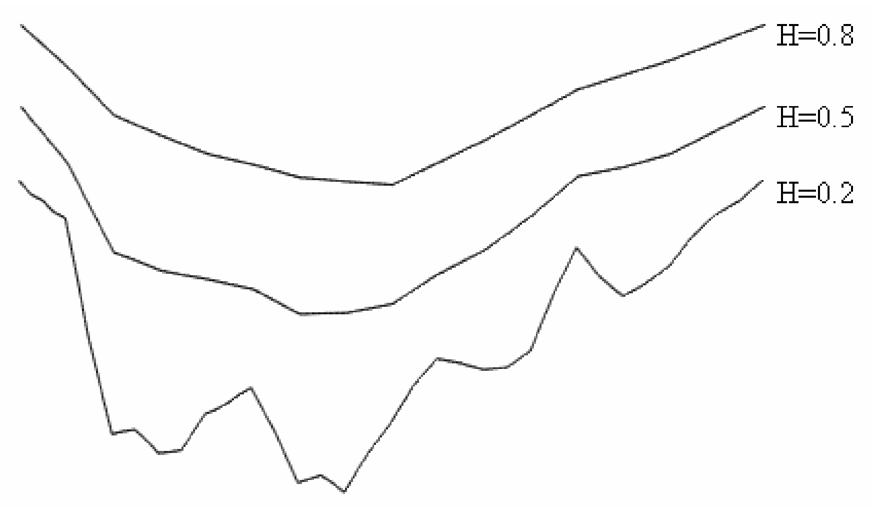


Figure 4.2 Ridgelines with different roughness coefficient values.
Adapted from (Peitgen and Saupe 1988)

Using the above method height values can be generated very fast, and there are enough parameters (roughness coefficient, random number generator) which can be changed to produce different results. The height values are generated in powers of two, as an example after 10 iterations 1024x1024 are generated within milliseconds. Figure 4.3 shows a wireframe terrain generated with the above algorithm (5 iterations, H=0.8) using a initial version of our terrain engine.

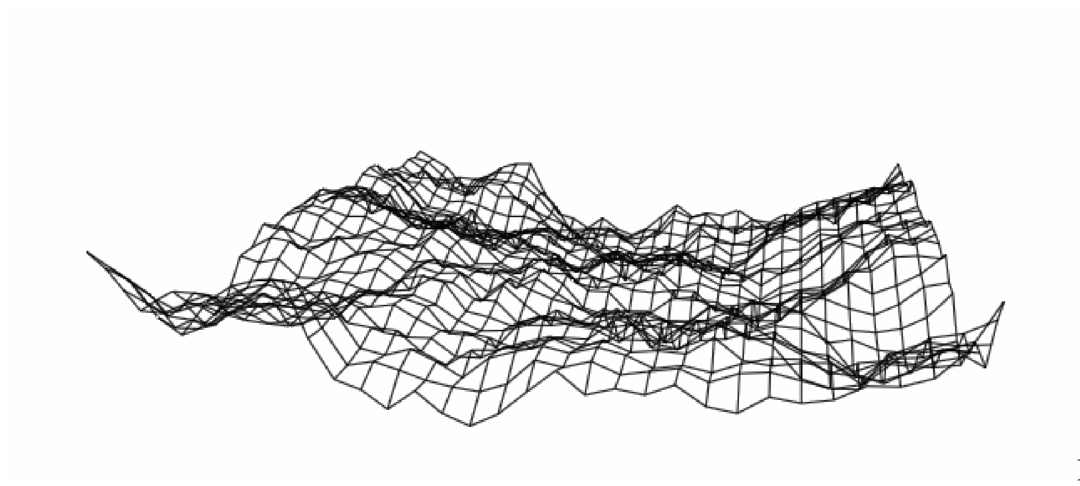


Figure 4.3 Terrain generated with the Diamond Square algorithm.

4.2.2 Dynamic Paging

The amount of data needed to store even modest size terrains is large and can easily overload the memory capacity of a machine. Therefore it is not possible to store all the terrain needed in memory at initialisation. Only a subset can reside in memory at any time and further data has to be swapped in/out on demand. The swapped data can be loaded either from another storage device such as the local hard disk or as in our case either loaded from the cache or be computed on the fly. This process is called commonly "Terrain paging".

In order to accommodate for such dynamic changes a specialised data structure and algorithm is developed which handles and determines when and what blocks should be swapped in or out. The algorithm deployed by the engine handles the terrain information in blocks, the whole terrain is divided into blocks, each block represents a part of the whole terrain it stores all the information needed to render this part of the terrain. All blocks, which are in memory at a current time step, represent the active area for this time step.

When the terrain engine is initialised the user is centred in the middle of the active area displayed and a bounding box is established around the user (centred in the middle of the active area). When the user reaches the bounding box in any direction, blocks in the opposite direction of travel are paged out, new blocks are paged in, in the direction of travel and the bounding box moves. This whole procedure is demonstrated in Figure 4.4 for the case the user travels north. Similar are the cases for travelling south, west or east. In the case of a Northeast movement we would page North and then East, the other combined cases (NW, SW, SE) handled in a similar way. Note that the bounding box moves with active terrain area and is always at its centre.

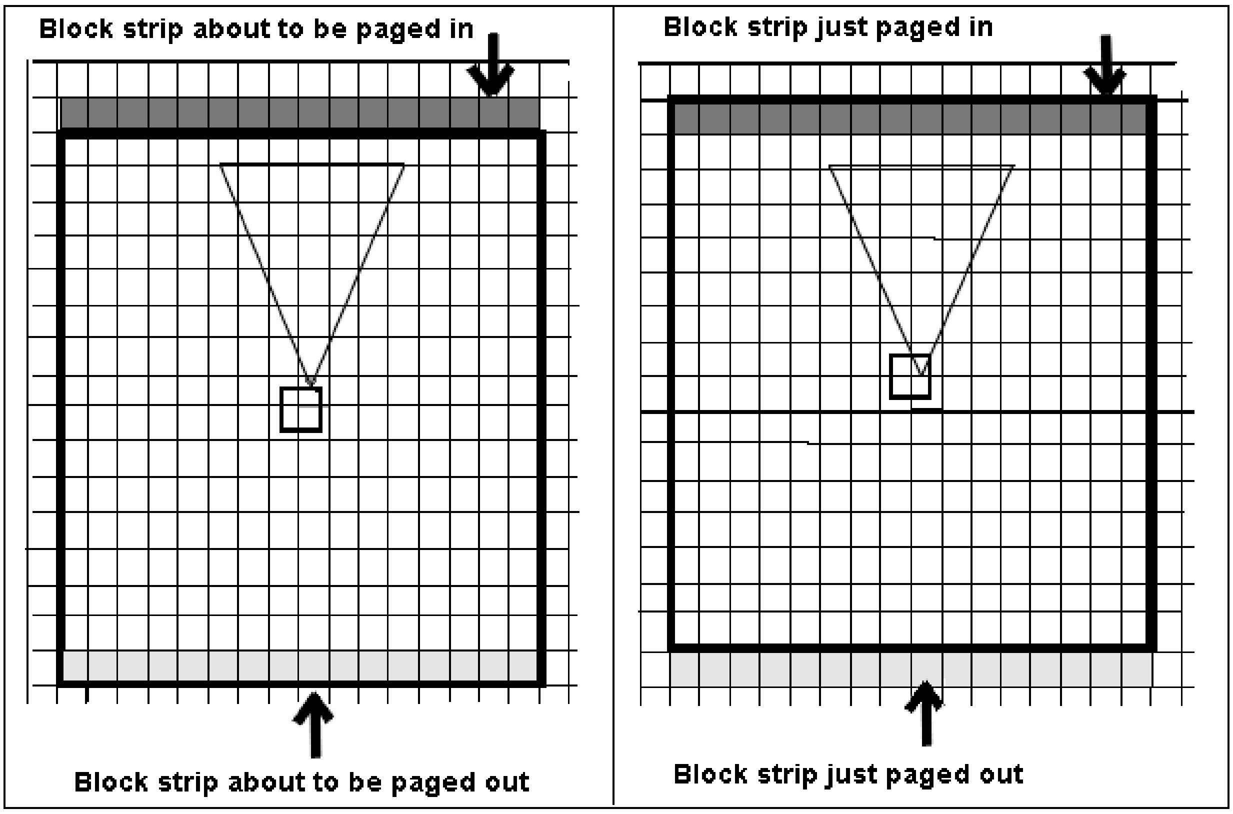

Block strip about to be paged in
Block strip about to be paged out
Block strip just paged in
Block strip just paged out

Figure 4.4 Active area and Terrain paging

Care has to be taken in adjusting the size of the bounding box. If the bounding box were equal to the size of a block, edges of the new and old bounding boxes would coincide. The user could continually move back and forth across the boundary resulting in thrashing. Therefore a slightly bigger bounding box than the current block size (25% bigger) is used, to make sure that the user has travelled a certain amount in the direction of travel before paging takes place (Figure 4.5). The size of the bounding box can be adjusted though, as required by the speed of the user.

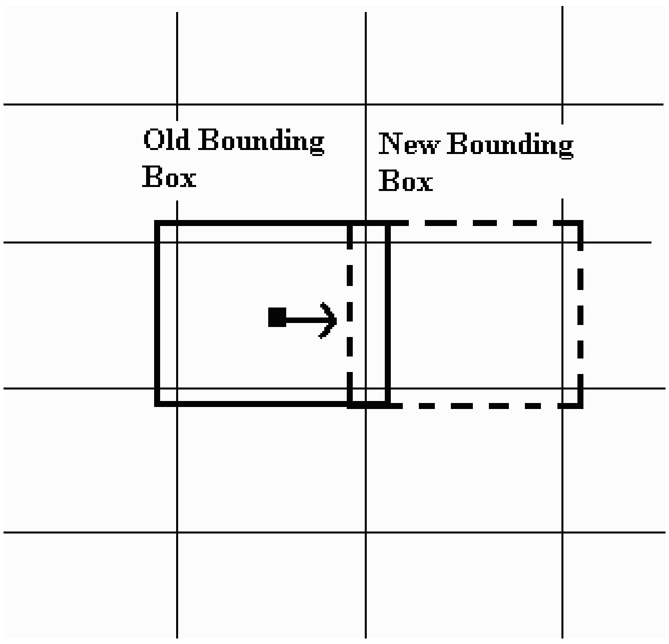


Figure 4.5 Bounding Box for Terrain Paging

4.2.3 Unlimited Terrain

Although the form of terrain paging used is very general and does not depend on any hierarchical data structure implemented, it still has to be combined with the fractal terrain generation algorithm to produce additional terrain information which can be paged in.

The first parameter to be determined is how big are the blocks going to be and how many height values each block will contain. In the engine this is determined by the number of total iterations that are specified by the user to produce terrain using the Diamond Square algorithm. If N iterations were specified, the number of blocks will be the amount of values produced after N-3 iterations, and each block will contain height values equal to the ones produced after N-(N-3)=3 iterations. The amount of 3 iterations is not chosen randomly but tightly coupled with the LOD algorithm explained later. Explained briefly each block is subdivided using 3 iterations, which means 3 Level Of Detail. Figure 4.6 shows how blocks and data would be distributed if 7 iterations were specified. In this example the number of blocks would be produced after 7-3=4 iterations, which means (2^4 =16) 16x16 blocks and each block would contain (2^3+1=9) 9x9 height values.

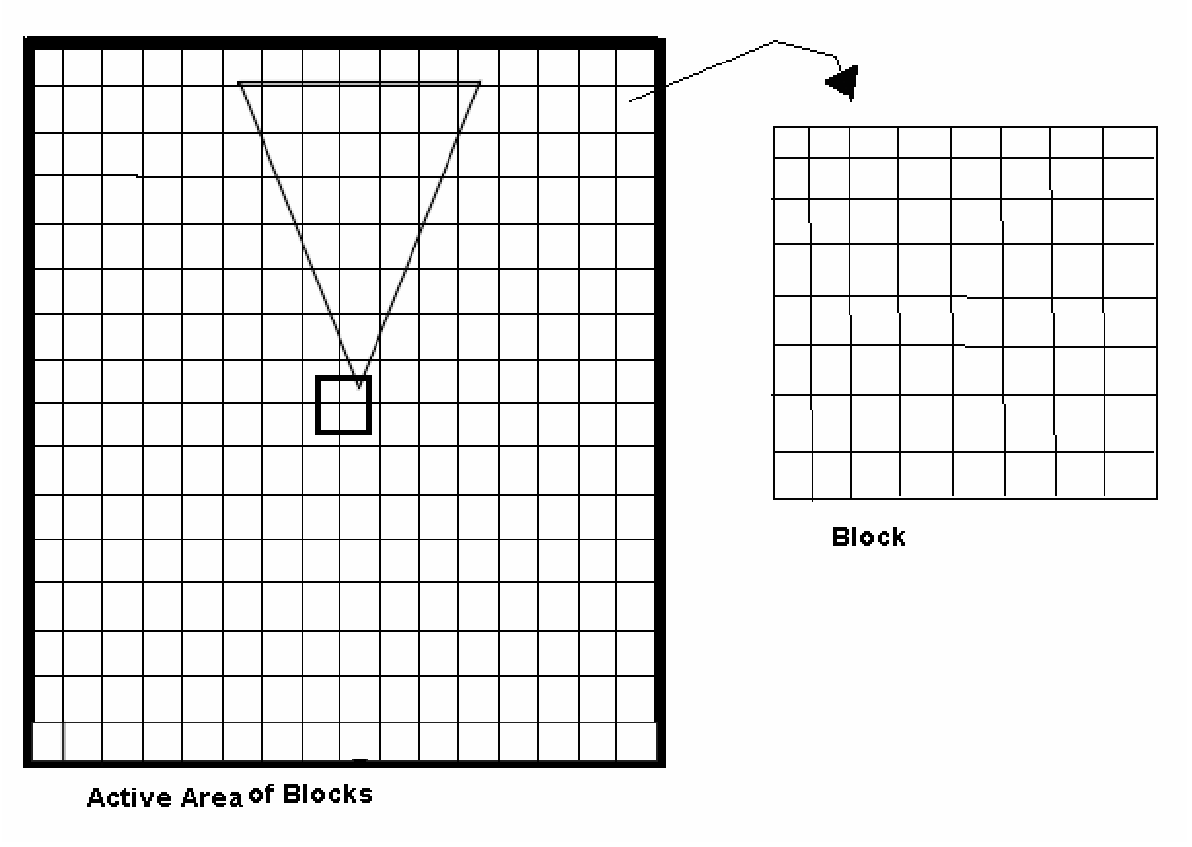


Figure 4.6 Block generation for terrain paging (7 iterations).

Terrain Paging

Now that the terrain is subdivided it can swap in/out blocks according to the terrain-paging algorithm described. For each new block swapped the Diamond Square algorithm is applied to generate new height values. So for each new block its corner height values are computed and the algorithm is applied for 3 iterations. Using this technique the new terrain gets computed in small parts which can be handled easily by the system and doesn't cause noticeable delays. In the case of Figure 4.6 for one swap in any direction, instead of applying the algorithm for terrain generation for 7 iterations to produce $(2^7+1)x(2^7+1)=16661$ height values, only (9x9)x16=1296 height values are needed.

4.2.4 Matching up Blocks

Although this algorithm may seem simple it suffers from a serious flaw. The problem is how to specify the height values for the new corner points. Not just any random value can be used as a corner height because when it comes to apply the terrain generation algorithm the vertices which are common in neighbouring blocks might not match. Figure 4.7 explains this problem in more detail. Suppose a new block was just swapped in, on top of an already existent old block, and new height values are generated for its interior with the Diamond Square algorithm. For demonstration purposes it is assumed that only one iteration per block is needed, which means each block has $2^1+1=3$ height values.

It can be clearly seen that the two blocks share a side and have points in common. For the fractal algorithm to generate new height values inside the new block the initial corner height values for the this new block have to be provided. One solution would be just to copy the common corner height values of the old block to the new one, determine randomly the height values for the other two and apply the algorithm.
This naïve approach would not work because there would be differences in the height values in common interior points (Figure 4.7). These differences are caused because of the random nature of the algorithm used. In essence what we do is generate height values for each block separately, so each common point at a border does not have knowledge of the height values which are next to him on a different block.

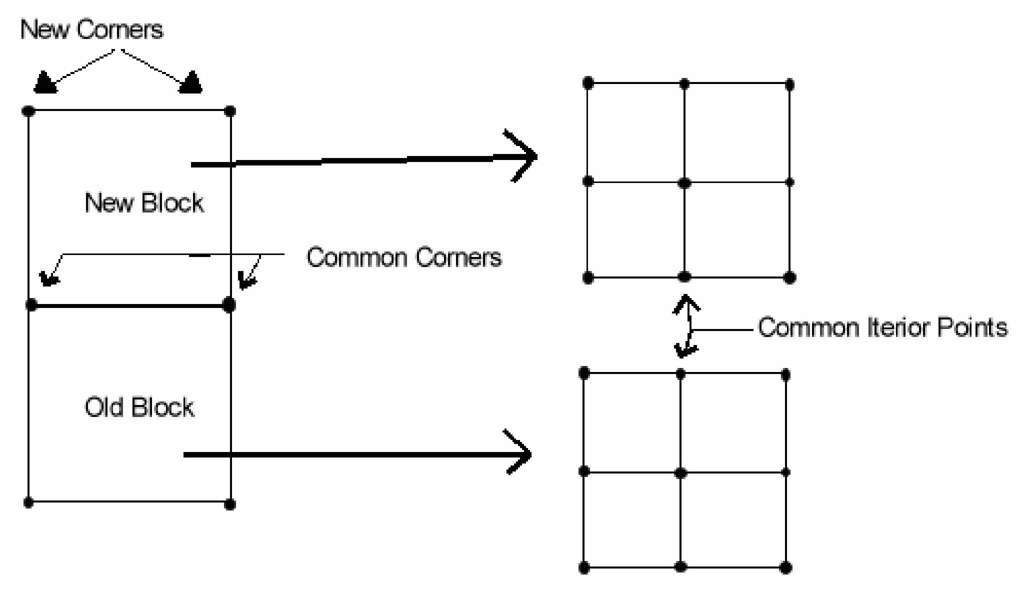


Figure 4.7 Matching up two neighbouring Blocks

Altering the Diamond Square algorithm slightly to work for these cases solves this problem. Before applying the algorithm we copy not only the height values of the common corner points to the newly created block but also all the interior common points. When applying the algorithm onto the block we do not allow these values to change, but only influence the other values which are generated. The positive side-effect of this technique is that although the border points do not change, smooth transitions between tiles are generated because they influence the height generation of other points.

Note that a newly created block can have neighbouring blocks on more then one side (in case of Figure 4.7 there could also be a block to the left or right). All border values inclusive corner values are copied to the new block, any corner point for which a height value was not specified (copied) from the neighbouring points will be given a random value.

4.2.5 Natural appearance

Although the problem of matching up neighbouring blocks is fixed and there are no gaps anymore, there are still some problems to be solved. The actual terrain displayed does not look natural, the new blocks generated would look unnatural rough and steep. The roots of the problem lie again in the random nature of the terrain generation algorithm.

Specifically the problem lies in determining the random values at the corners of the new blocks. Some corner values get determined by copying from neighbouring blocks but the rest of the remaining corner values are determined randomly. This is what causes the problem, and introduces unnatural artefacts such as the terrain getting suddenly rough or sudden changes in altitude.
The specification of the new corners at random causes these artefacts because they are not correlated with the corner points of the blocks around them. Every corner height value of a block does get randomly determined, resulting in sudden changes and uncontrollable behaviour. We must somehow correlate all the corner values of the new blocks generated with corner values of the already existent blocks.

To solve this problem the nature of the fractal algorithm used has to be examined more closely. When applying this algorithm to an area or block all the values, which are determined at the beginning and don't change during the algorithm (as the corner values), influence all the generated values. The newly generated values are interpolated from the already existent ones and a random perturbation is added. This means that when specifying the corner values, more or less, the resulting values that are generated inside are interpolated from these corner values and exhibit a common behaviour. If two of the corner values are low and the other high the resulting terrain will (with random perturbations) have an uphill behaviour from the low points to the higher ones.
In our case we interpolate a large initial area and cut it up into blocks. The points generated including the corners of the blocks are determined by the values of the corner points of the large area. Now if we swap in additional blocks their corner values and

hence also their values inside will have a completely different, random behaviour, and introduce sudden unnatural changes to the terrain.

To avoid these problems we would have to generate each time an area in the direction of movement as big as the initial area interpolated. Having computed all the height values for the whole area we can use these values as corner height values for the new blocks. This technique avoids artefacts because the computed height values are not random chosen anymore, but are interpolated using the Diamond Square algorithm from a much bigger area. The drawback of this technique is that it is very expensive to generate the required height values for large areas every time we swap in/out terrain.
Although we only would have to perform as many iterations needed to compute the corner values of the blocks, which would be N-3 iterations (N: initial iteration specified for whole terrain), these computations would cause a noticeable delay because they have to be performed in every swap. Figure 4.8 shows the difference between the two techniques, (a) shows how the paging technique, which causes artefact problems and (b) the new artefact free technique, which is much more computational expensive.
Lets not forget that our aim was to specify the corner values of the new blocks. After that each new block is matched up with other neighbouring blocks and subdivided (3 iterations). In Figure 4.8 we assume that the terrain is cut into 4x4 blocks (2 iterations). Each block would be normally again subdivided with 3 iterations, which indicates that the initial iterations specified by the user were 2+3=5.

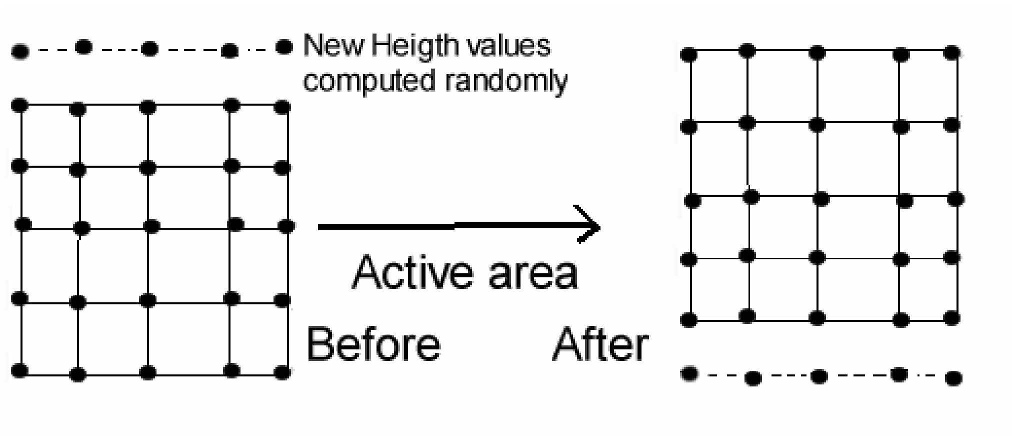


Figure 4.8a The two different techniques for terrain paging using fractals

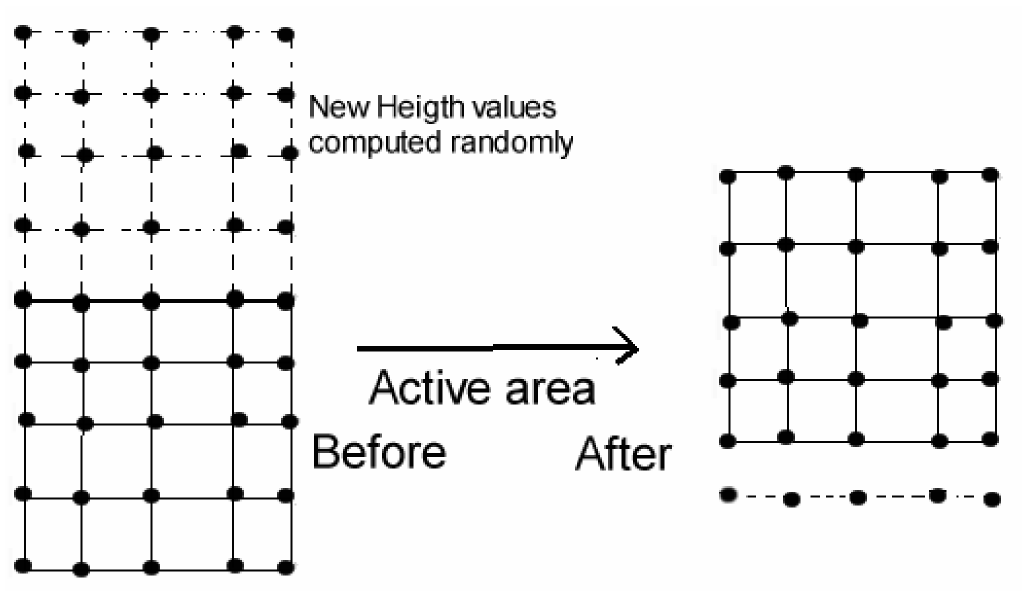


Figure 4.8b The two different techniques for terrain paging using fractals.

4.2.6 Caching

It has already been shown that in order to avoid artefacts we have to perform more computations dividing up the whole area to produce correlated height values.

Although these additional computations are more expensive, they do not have to be repeated for every swap. Because we actually produce much more information than needed at this time, caching gives an essential speed increase. Having produced all these new height values for an area, which is not in the cache, we store them in the cache for retrieval when needed. Until they are replaced by another area in the cache any point within the area will be immediately available. Due to locality of movement the area will only be replaced when the user has left the region and new data has to be computed.

The size of the cache determines how many areas can be held in memory concurrently, ideally this would be a size of 8 areas. This means that it would be possible once a new area is entered, to cache all the surrounding areas and no computation, regarding the corner height values will be needed for movement within that region.

Once again it is very important to understand that only the corner values are computed using this process. Even if an area has to be subdivided, it will be subdivided with N-3 iterations ($2^{(N-3)}+1$ points) in order to form the blocks and not with the full N iterations (2^N+1 points) specified by the user. The missing 3 iterations will be performed for each block separately as it gets swapped in.
In Table 4.1 we describe the whole process of paging fractal terrain as with pseudo code.

```
Perform Diamond-Square Algorithm on initial area (N iterations)
Cut area into block and divide data
While (not finished)
{
  If (user out of bounding box)
  {
        Swap in new blocks

      For each new block do
      {
              If (new corner values needed in cache)
                 {
                     Copy corner values from cache to the blocks
                 }
              else{
                     Perform Diamond-Square for new area (N-3 iterations)
                     Cache computed data
                     Copy corner values from cache to the blocks
                 }

               Match up new block with neighbours
               Perform Diamond-Square algorithm to block (3 iterations)
      }
  }
  Get movement input from user
} //not finished
```

Table 4.1 The paging algorithm for fractal terrain in pseudo code

4.2.7 Controlling Fractal Terrain

Until now the issue about how to combine terrain paging and fractal terrain generation to produce dynamic and unlimited terrain has been addressed. Nothing was said about the other problems that fractals exhibit, their uncontrollable nature. Since the detail is generated on the fly, somehow it has to be made sure that it is always the same. When returning to a place previously visited, you don’t want it to have changed.

This is accomplished by controlling the random number generator. Each random number generator has to be initialised with a seed number, if the same seed number is supplied each time, the same random numbers will be produced. Based upon this fact if we had a method with which we would generate for every point the same random number, we would produce always the same random perturbation values and hence the same terrain.

In order to determine the random perturbation for a point we use its co-ordinates as its seed value. Using this technique we will generate for each point always the same seed value and hence the same random perturbation.

4.3 View Frustum Culling

View frustum cullers (VFC) are typically used in virtual reality software, walkthrough algorithms, scene graph APIs or other 3D graphics applications. In this section an essential and specialised VFC algorithm is developed for the terrain engine.
A frustum consists of six planes, where two are parallel to each other (Figure 4.9).For orthogonal viewing the frustum is a box, and in the case of perspective viewing, the frustum is a truncated pyramid, which is the frustum that we consider in this project.

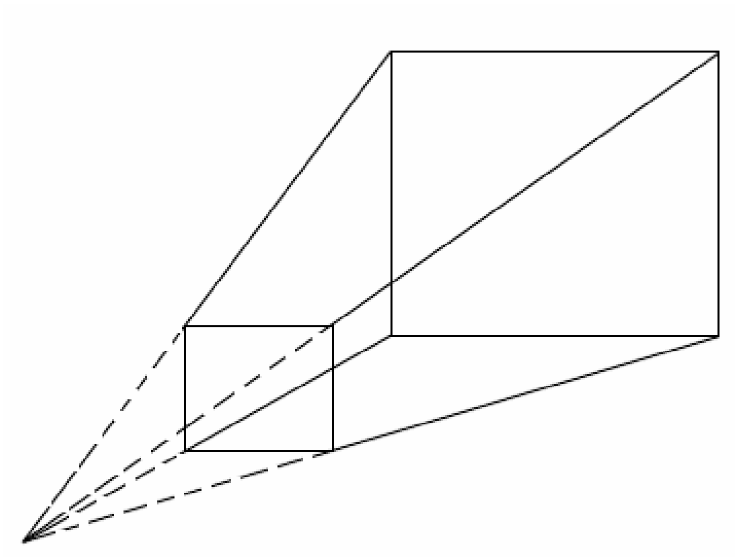


Figure 4.9 Viewing Frustum

In order for a VFC algorithm to perform efficiently a data structure has to be developed which consists usually of Bounding Volumes (BV). Each BV encapsulates a number of primitives that have to be drawn. Often in tree like structures such as scene graphs (a scene graph is a directed acyclic graph, where each node has a BV attached to it), there are also BV which encapsulate other BV and so on. This idea of hierarchical frustum culling was first documented by Clark (1976), and leads to dramatic speed increases as large amounts of BV (i.e. Geometry) are culled from the frustum.

A view frustum culler culls away the Bounding Volumes that lie outside the view frustum, i.e. those objects that are outside the users field of view. In general we have 3 cases, a BV can be completely outside, completely inside or can intersect the frustum (partly inside and partly outside), it depends on the implementation how this last case is handled.

A specialised and highly effective algorithm for a particular case of BV (Bounding Boxes) will be developed. The initial idea is based upon (Greene 1994).

4.3.1 Overlap testing for a Plane and a Bounding Box

Often in culling or clipping to a view frustum it is necessary to “classify” a bounding box with respect to a plane as either lying completely on one side or as intersecting the plane. Normally this involves performing tests on all of the vertices of a cube. Each of these tests requires the evaluation of the plane equation (3 multiplies and 3 adds times 8) which requires 24 multiplies and 24 adds. This method is too expensive and too general, we would like to reduce the amount of operations performed for the special case of rectangular bounding boxes (both arbitrarily oriented OBBs and axis-aligned AABBs).

It seems unnecessary to check all of the points with respect to a plane when the only points needed are the extreme points, when projected onto the axis defined by the normal of the plane. This means finding the farthest vertex in the direction of the normal (p-vertex) and the farthest vertex in the negative direction of the normal (n-vertex) (Figure 4.10). These points can be determined quite quickly.

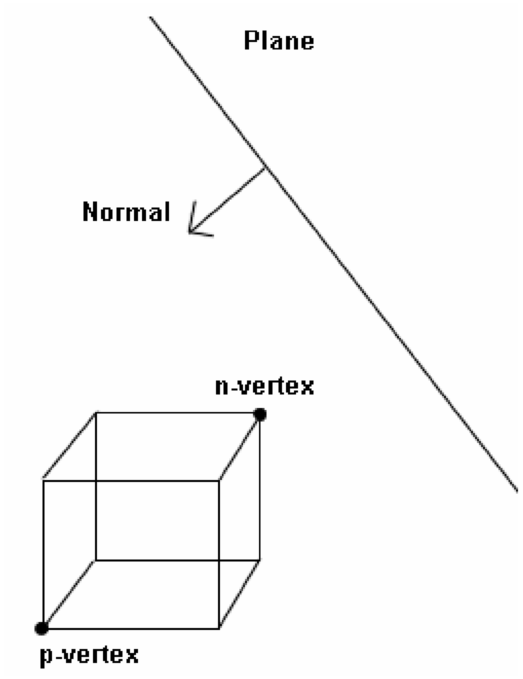


Figure 4.10 Positive far point (p-vertex) and Negative far point (p-vertex).

If the normal of the plane is defined in the box's co-ordinate system we can simply divide the cube in octants and then test in which octant the normal points to, once placed in the middle of the box. If the octant is found, the point which resides in that octant is used.
To transform the plane normal to the box's co-ordinate system, we just have to project the normal onto the normalised axis vectors of the box.
This can be performed by three dot products. Given the box normalised axis vectors Xaxis, Yaxis, Zaxis and the plane normal, we can calculate the normal N′ in box co-ordinates as follows:

$$N' = (\,dot(N,Xaxis),\, dot(N,Yaxis),\, dot(N,Zaxis)\,)$$

This involves the evaluation of (3 dot products, 3 multiplies , 2 adds each times 3) 9 multiplies and 6 adds. The above procedure is performed in the case of oriented bounding boxes, but for axis aligned bounding boxes N′=N and no transformation has to be performed. The transformation done is merely the "identity".

Having computed the plane’s normal N (now N′ is changed to N) in the box co-ordinate system, we can quickly perform a set of conditional tests to find out in which octant the p and n vertices are. In pseudo code this is

```
If (N.x>0)
{
  If (N.y>0)
    If (N.z>0) {Right Top Front}
    Else {Right Top Back}
  Else
    If (N.z>0) {Right Bottom Front}
    Else {Right Bottom Back}
}
Else
{
  If (N.y>0)
    If (N.z>0) {Left Top Front}
    Else {Left Top Back}
  Else
    If (N.z>0) {Left Bottom Front}
    Else {Left Bottom Back}
}
```

The above code determines the p-vertex. Setting N=-N will find the n-vertex.

The p and n vertices have now been computed with 9 multiplies, 6 adds and 3 comparisons for each point resulting for both points in 18 multiplies, 12 adds, 6 comparisons, and 3 negations to get the –N. That is for the general orientation case, but with axis aligned boxes only 6 comparisons and 3 negations are required. However if advantage of the symmetry that the boxes exhibit is taken into account, the n-vertex can be found as being the opposite corner . In this case only 9 multiplies, 6 adds and 3 comparisons are needed for the general case, and only 3 comparisons if the boxes are axis aligned.

Having found the extreme points the usual test for determining on which side of the plane these points are is applied. The half space determined by the plane in the direction of the normal is referred to as "outside" and the other half space as "inside". If the n-vertex is in the "outside" half space then the whole box is considered to be outside. If the p-vertex is in the "inside" half space then the box must be completely inside. Otherwise the plane intersects the box.

For the above tests in the worst case an additional of 6 multiplies, 6 adds and 2 comparisons are required, resulting in the following overall results.

Previous methods: 24 multiplies, 24 adds and more comparisons than new methods

Arbitrarily oriented (OBB): 15 multiplies, 12 adds, 6 comparisons

Axis Aligned (AABB): 6 multiplies, 6 adds and 5 comparisons

4.3.2 Culling in the Terrain Engine

Having now the basic tool for performing culling against a plane. Modelling the view frustum as a set of 6 planes we are able to implement full view frustum culling. The method, with which the terrain area is divided into blocks, lends itself perfectly to the above method using Axis Aligned Bounding Boxes (AABB).
Each block in our terrain structure has an AABB stored with it. The AABB is determined when the geometry for the block is formed, and thus encloses all the geometry. Therefore performing the above test can quickly and accurately determine if any of the points inside the block is visible.

This VFC algorithm can achieve speed ups of 2-10 times depending on the scene.

4.4 Level Of Detail

Although Frustum culling does a good job and culls away large parts of geometry, there are still enough primitives in the viewing frustum to overload the pipeline completely. To reduce the load even further we have to apply a Level Of Detail algorithm to the scene which is still visible. As described in previous sections LOD algorithms are based on a simple rule, they represent geometry, which is further away and has a small projection on the screen with fewer primitives.

4.4.1 The basic algorithm

In our engine we deploy a simple but highly effective algorithm for surface simplification. We employ a block based simplification scheme and base our LOD criterion on distance.
This means that we represent the geometry in blocks and display these blocks in various Levels Of Detail depending on their distance to the user. If a block is near to the user all the geometry is used to display it and the further the blocks get from the user the less geometry is used to represent the same object.

For the block representations we will use the blocks which we use to do the swap in/out and the view frustum culling. This representation is very convenient because it connects all the main stages into one clear pipeline.

The blocks from the active area, which consists of old blocks and newer ones, which were swapped in/out, are first culled against the frustum and finally drawn with different Level of Detail. The only problem is how exactly are we going to measure the distance from the user and how the blocks are going to be simplified into lower level of detail.

4.4.2 Distance based Simplification

The Level Of Detail, which will be used for a block, is determined by the distance between the user and the centre of a block. Each block will be represented with 3 levels of detail, high, medium and low.

In essence the remaining blocks will be categorised into 3 resolutions and the area split up into 3 sub areas. One area (near to the user) will display the high detail blocks, the middle area will display blocks in medium resolution and the third area (furthest away) will display blocks in low resolution. The areas will build homocentric circles around the user (Figure 4.11).

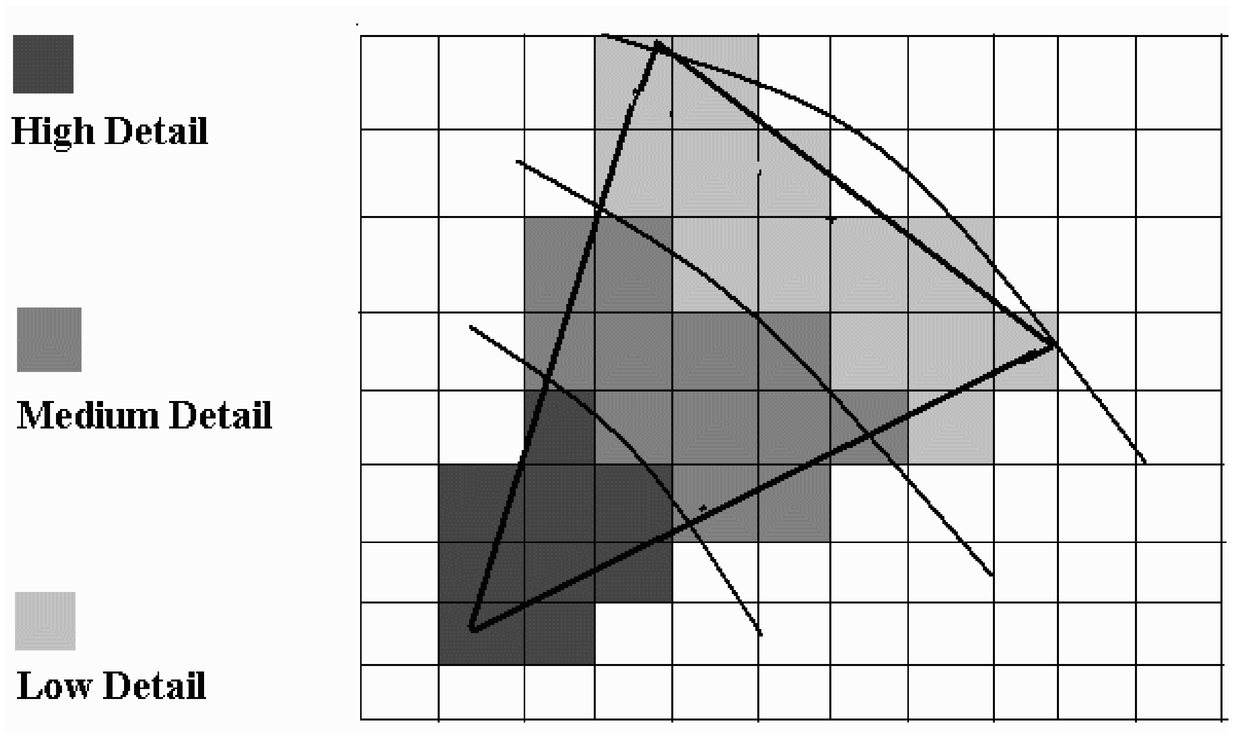


Figure 4.11 Multi-Resolution Rendering of Blocks

The radii of the various resolutions can change and are adjustable to the preference of the user

4.4.3 Block Simplification

After determining at what resolution to draw the blocks, we have to determine how to actually simplify the geometry represented by the block. Again we can couple this demand magnificently and effectively with our fractal terrain generation algorithm.

Section 4.1 described that the fractal algorithm produced fractal terrain by subdividing the initial area according to the iterations specified by the user. Assuming that the user had specified N iterations, we divided the area for paging into as many blocks as were produced with N-3 ($2^{(N-3)}$) iterations, and each block would store the points produced by remaining 3 iterations applied to it.

Observing that each iteration applied onto an area inserts more points (i.e. detail). We can consider that a block after 3 iteration applied to it, is represented with more Level Of Detail than if it would be subdivided with 2 (medium LOD) or even 1 (low LOD) iteration. Therefore we can draw each block with different LOD's just by taking the points produced by the fractal algorithm on different stages of the subdivision process. The more subdivision the more detail. Figure 4.12 shows a block as produced by the fractal algorithm and how an area represented with these blocks can be rendered in various levels of detail.

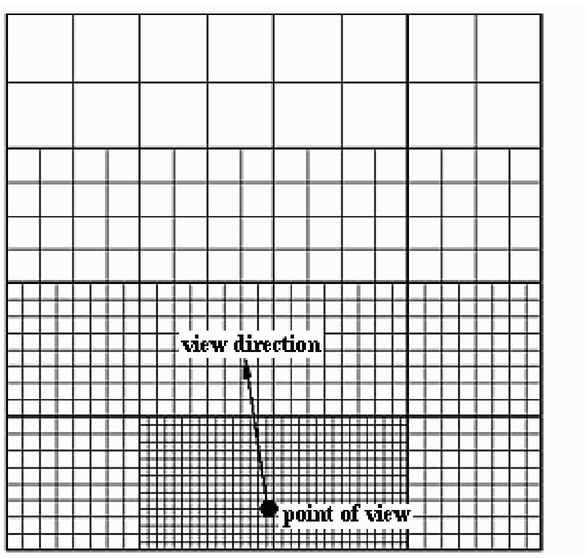


Figure 4.12 Terrain management with multiple levels of detail

With increasing distance to the user, each block is subdivided less often, resulting in a coarser representation of the terrain. Regions further away are smaller on the screen, so the absence of detail goes unnoticed. As a block comes nearer its representation is refined.

The savings from the LOD algorithm are large. A high resolution block (3 iterations) is represented with (2^3+1=9) 9x9=81 points, at medium resolution (2 iterations) it is represented with (2^2+1=5) 5x5=25 points and at low resolution (1 iteration) with just (2^1+1=3) 3x3=9 points. Because these Levels Of Detail are already produced as part of the landscape generation algorithm there is almost no overhead in computing the points for a specific resolution and triangulating the block. The savings in rendering time are vast and the speed up tremendous.

4.4.4 The Gap Problem

One of the problems associated with dynamic changes of polygon resolutions is connecting blocks of different resolutions. Most often, small gaps along the edges will appear, since not all of the points on an edge are shared between the two blocks of different resolution. This can be seen in Figure 4.13 where point D is not common in both resolutions. One solution to this is given in DeHaemer and Zyda (1991), where the vertices of the higher resolution along with the points of the lower resolution are all used to form polygons to cover the gaps (i.e. in Figure 4.13 the points A, B, C, D, E would form 3 triangles). One drawback of this technique is the introduction of additional triangles to cover each gap. The solution used was is to use a number of triangles in the y-z planes (with appropriate lighting and texturing to cover the edges) to fill in the gaps. In Figure 4.13 we would draw the triangle CDE. Chapter 5 will elaborate more thoroughly on this issue. Figure 4.14 illustrates the problem of gaps in the terrain in our terrain engine before applying any solution.

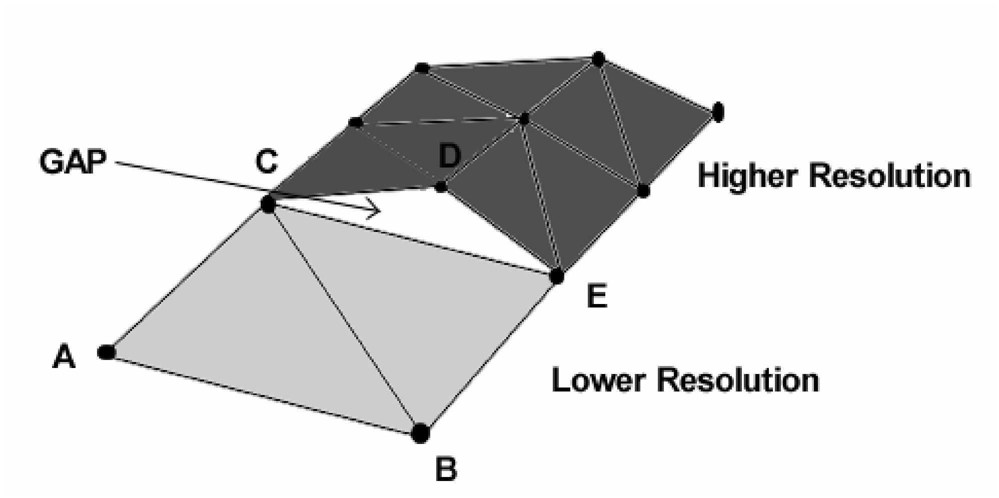


Figure 4.13 Gaps between different resolutions

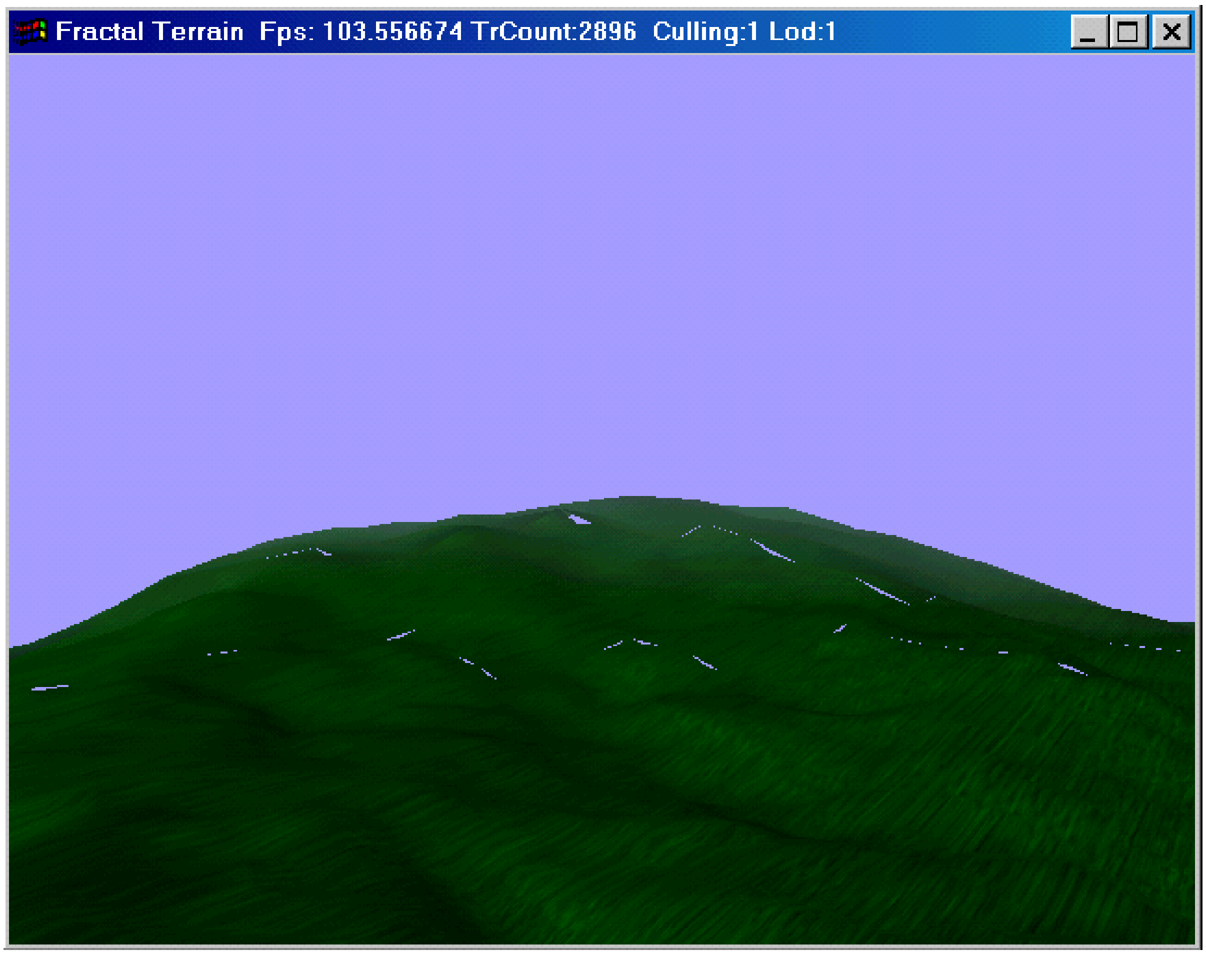

Fractal Terrain Fps: 103.556674 TrCount:2896 Culling:1 Lod:1

Figure 4.14 Gaps generated in the terrain.

5. Implementation

This chapter will discuss how the main algorithms were implemented and any problems and alterations that occurred.

5.1 The Classes

In Figure 5.1 a diagram of the classes developed is shown. Each of these classes encapsulates one or more of the algorithms and functionality described in Chapter 4. Below is a short description of each class.

Tworld	Superclass, handles the main events of the terrain engine such as terrain paging, Level Of Detail and Culling of Blocks.
Tterrain	Encapsulates the fractal terrain generation algorithm, given a area it produces the height, colour and Normals for it.
Tblock	Represents a block of the area, which is handled by the Tworld class. Stores all information needed for this particular area it represents.
Tcache	Caches terrain area previously visited for later use.

Of course there are other friend functions especially in the main function, which handle call-backs (keyboard, mouse, display, timing), load textures, generate random numbers and initialise the engine. Figure 5.1 shows a class diagram of the engine.

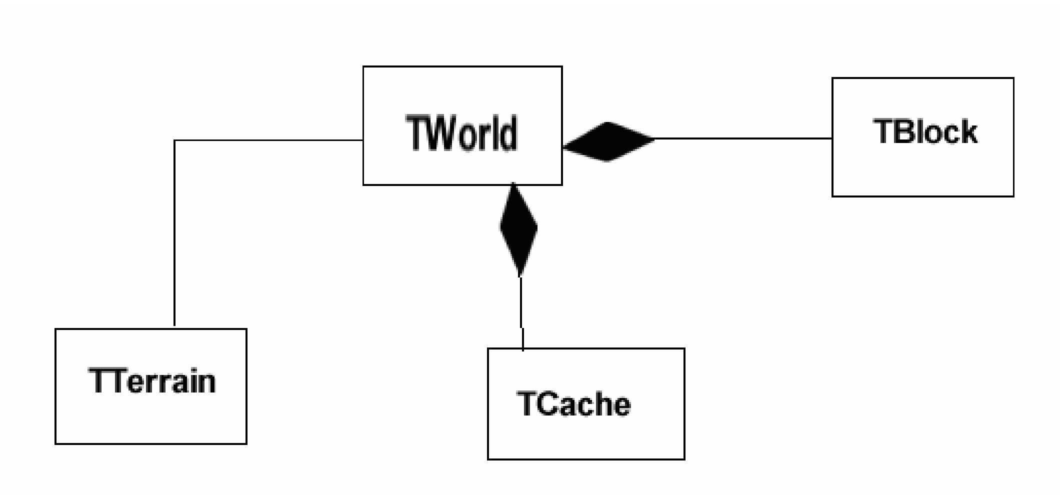


Figure 5.1 Class Diagram

5.1.1 Data Structures

The **TWorld** class handles and implements the algorithm for dynamic paging. The data structure used for was a Heap-Sorted Quadtree. All nodes were stored in an array, in heap sort order. A traditional tree structure with pointers was avoided, because of the delays that would be caused by following the pointers to find the final node.

For a dynamic environment like terrain, a simple data structure is more beneficial. The nodes (blocks) are represented by the TBlock class and were stored in a 1D array. Each cell of the array stores a pointer to a block (TBlock class), using this pointer no copying of blocks is needed when swapping. When a swap occurs only the pointers are copied and not the objects.Memory allocation, which is very time consuming is only done at the beginning. When new blocks have to be swapped in, the old blocks are just swapped out and their memory is used instead of allocating new memory. There are no memory allocations done at run time, nor is any other memory after the initialisation phase.

5.1.2 The Random Number Generator

Usually just generating random fractal terrain is not sufficient. In order for a terrain engine based upon fractal technique to be usable control over terrain generation is required. The random number generator used provides this feature by basing its random values for a point upon its co-ordinates.

Clearly if the same seed is used to initialise the random number generator the same random number will be generated. To determine the seed for a point in our world we construct the seed by its x, z co-ordinates. The random value produced is then used to perturb the existing height value (y co-ordinate).
The function used to determine the seed is given below.

$$Seed = 7*x*z + 5*x + 9*z + Variation;$$

This function gives a very good distribution of the random seed over the world space. A good distribution of the seed values is needed to avoid repeating height values.

5.1.3 Terrain caching

Generating random terrain is not inexpensive in terms of time needed for computation. An efficient caching scheme which caches any previous values generated for a patch would give a defined speed increase. The **TCache** class caches any new height values generated by the **TTerrain** class before they are cut up into blocks (**TBlock** class).
If a specific point is needed when blocks are swapped in (usually new corner values) then it is first looked up in the cache before any computation is done. In most of the cases (except when the user leaves the area completely) any required value has been previously computed along with other surrounding values and is therefore already stored. This

avoids costly re-computations, by avoiding the execution of the fractal terrain generation algorithm.

The cache implemented can keep as much as 8 areas at any one time. Usually this means that all the areas around the user have already been computed and stored in the cache. If the point requested is not in the cache, the whole area in which is resides is computed (using the fractal algorithm) and stored in the cache before the actual point is returned. The Last Recently Used (LRU) order is used for determining which older areas is replaced by a newly computed one. Figure 5.2 shows the concept of the cache.

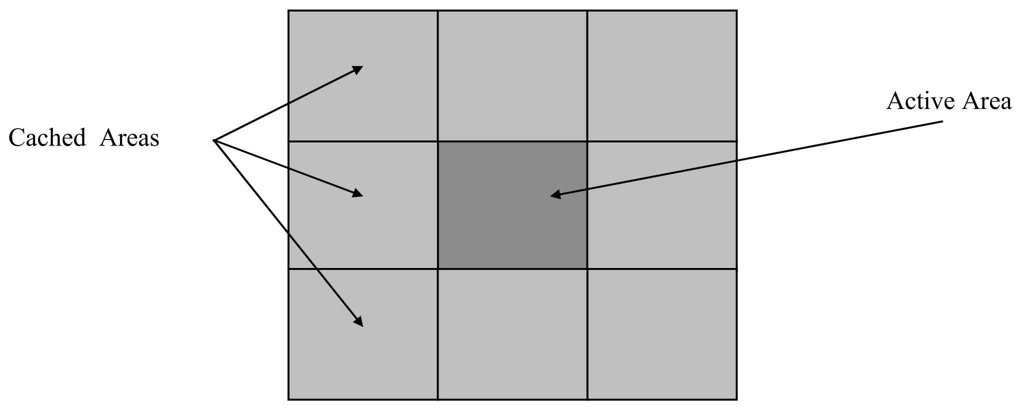


Figure 5.2 The cache scheme

5.2 Level Of Detail

5.2.1 The distance metric

Before rendering each block its Level Of Detail has to be computed. To compute a Level Of Detail for a particular block we have to find it distance from the user. To repeat this procedure for every visible block and for every frame that has to be displayed is too expensive. A method has to be found to avoid re-computation of the LOD values for a block at every frame. A metric is needed for determining when re-computation of the LOD values is required.

Before determining a condition (metric) which will indicates when re-computation is needed, a few assumptions have to be made.

1. At every point in time the terrain is displayed with 3 Levels Of Detail (High, Medium, Low).
2. The distance between the homocentric circles (borders) which determine when a Level Of Detail starts must greater than or equal to the length of the diagonal of the blocks used. Each Level Of Detail must at least as far away from the next one as is the length of a diagonal of a block.

Because the distance computed is always the distance from the user to the centre of a block, the above rules ensure that once a Level Of Detail is computed. It only needs to be re-computed when the user has moved (with respect to its position when the LOD values were computed) more than half of length of the diagonal (of the blocks used). Otherwise the new LOD values will be just the same as the ones initially recomputed. The Level Of Detail algorithm will give a different LOD value for a block, only if the user has moved more than the length of its diagonal with respect to where he was when the block was given a new LOD value.

Looking at how the terrain paging works with the use of a bounding box, a condition, which is based upon the above rules, already exists. If the Level Of Detail is determined when the Bounding Box is established, it has to be changed only when the actual bounding box changes and swapping occurs. Because the bounding box is nearly the size of a block, every move of the user inside it is not going to change the actual distance between the user and the centre of all other blocks (i.e. their LOD). Only when the borders of a bounding box have been crossed and swapping of new blocks is performed, have the new Level Of Detail values to be computed.

Using the above model allows the expensive calculations to be postponed and the user can move happily inside the bounding box until a condition is reached (i.e. the borders of the Bounding Box crossed). Thus the terrain-paging algorithm is connected to the Level Of Detail algorithm resulting in massive time savings, without impairing visual quality or the correctness of the LOD algorithm. Figure 5.3 illustrates this technique for a terrain of 5x5 blocks as active area.

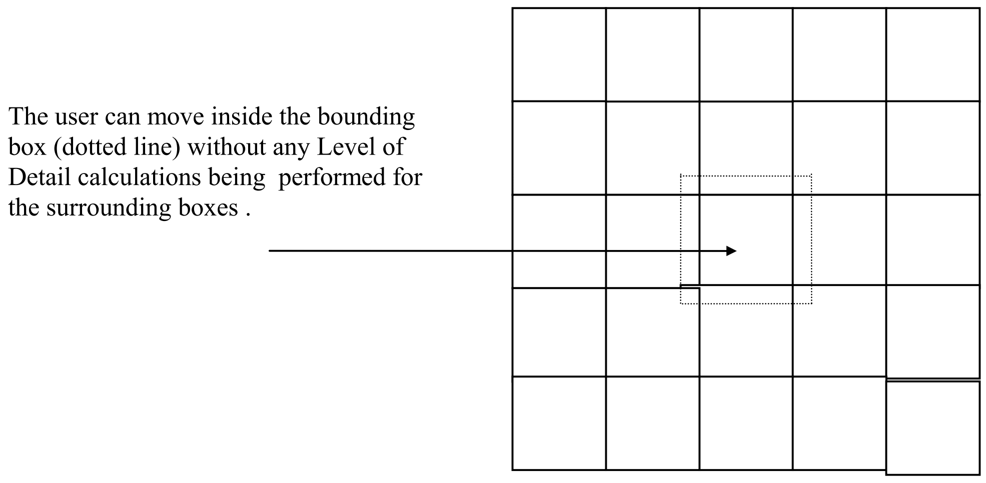

The user can move inside the bounding box (dotted line) without any Level of Detail calculations being performed for the surrounding boxes .

Figure 5.3 Bounding Box and Level Of Detail calculations

5.2.2 Cracks

One of the problems encountered when using multiple levels of detail in the same representation is mentioned in chapter 4.4, these are the gaps (Figure 5.4) which result from adjoining blocks represented at different levels of detail.

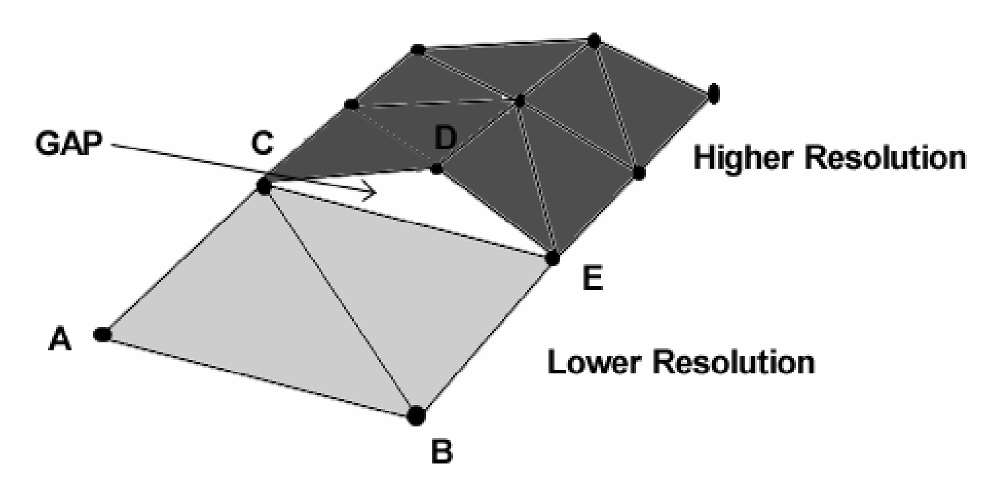


Figure 5.4 Cracks between two levels of detail

The problem is handled in the terrain engine by inserting triangles in the vertical y-z plane which cover the gaps. These triangles are smooth shaded and colour interpolated, using vertex normals which have been averaged with adjoining triangles, resulting in smooth and not abrupt transitions.

Advantages	Easy and fast solution. Usually need only a few small triangles to cover the gaps. Because the difference between two Levels of Detail regarding the triangles is big, the few additional vertically rendered triangles add little cost.
Disadvantages	Generation of artefacts in shading if gaps are big (not normally the case).

The above approach was taken because of the good results it gave optically and because of the small overhead involved. Nevertheless other solutions were also examined and partially implemented. Below is a list of several possible solutions. Although these solutions may not be necessarily worse than the one used, they involved either more computation or more artefacts. For the terrain generation scheme used, the current solution was found to be ideal with respect to cost and the quality provided.

1. Force every vertex on the edge of a high LOD block to be also present on the low LOD block.

Advantages	Good solution, there are never visual anomalies
Disadvantages	More polygons rendered than needed, and Low LOD blocks exhibit “star burst” like artefacts at the sides where joined with higher LOD blocks

2. Build transition tiles from one resolution to the other

Advantages	Low polygon solution. Good visual results.
Disadvantages	Need a lot of memory and management to cover all combinations of high and low detail block arrangements.

3. Force the edges of all low detail blocks to be lower than any vertex along the edges of the high detail blocks. Since the user is always on high detail areas gaps are never seen.

Advantages	Simplicity. Lowest polygon solution
Disadvantages	Not always feasible. Gaps can occur when at high altitudes or rough terrain.

4. Use specialised “continuous LOD” algorithm. These algorithms work with exact error metrics of the parts simplified, smooth edge contractions and vertex moving.

Advantages	Best looking solution. Eliminates pops.
Disadvantages	Very difficult implementation.

5. Clear the screen with a special background colour to make the gaps as imperceptible as possible.

Advantages	Simplicity. No additional polygons needed.
Disadvantages	Gaps are noticeable for some parts of the terrain if it is coloured with different colour values on different parts or drawing any areas with a sea layer in it.

A birds eye view, looking down on to the terrain represented with several Levels Of Detail can be seen in Figure 5.5. The blocks rendered with different Levels Of Detail can clearly be recognised. Figure 5.6 shows the actual terrain view with LOD and Culling on (top) and off (bottom). Note that the visual quality is not impaired and there is a large difference in frames per second and triangle count (on title bar).

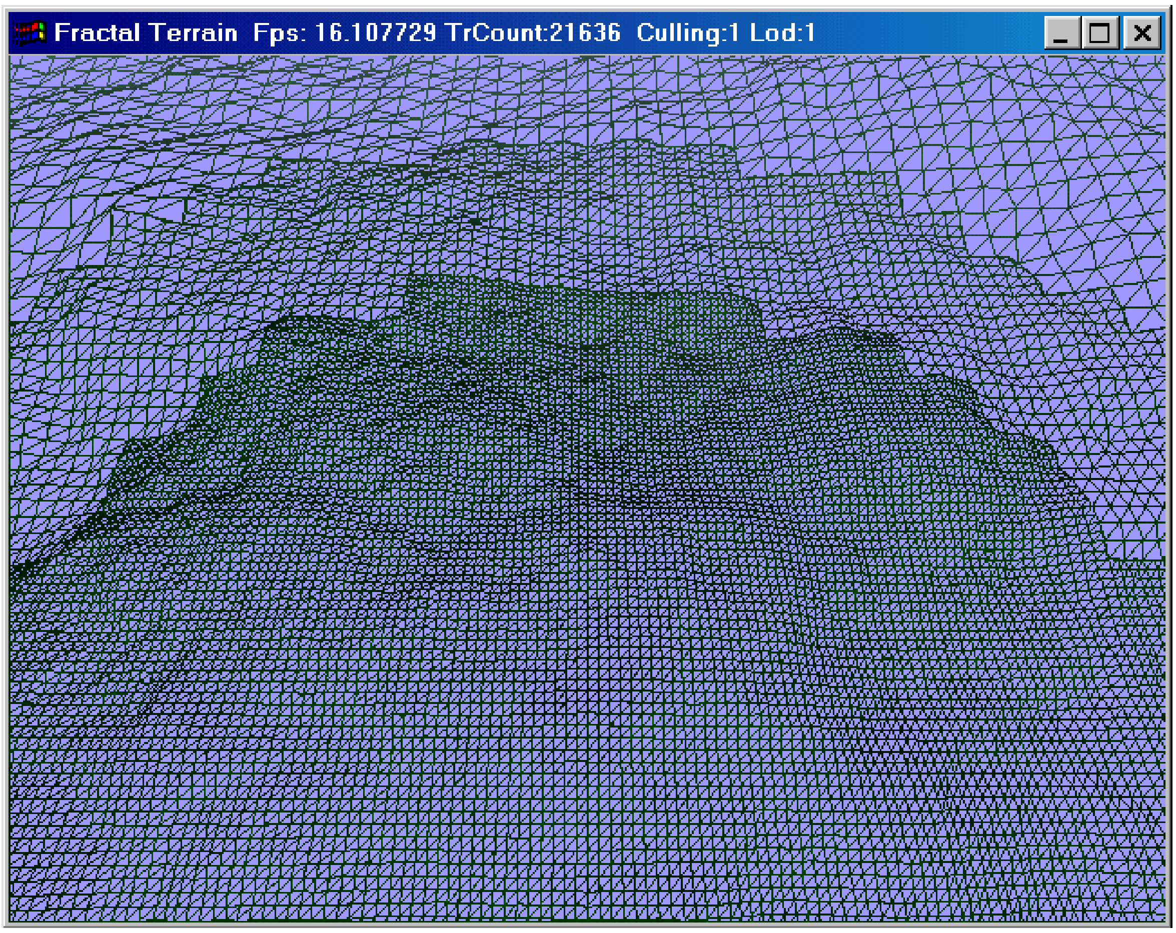
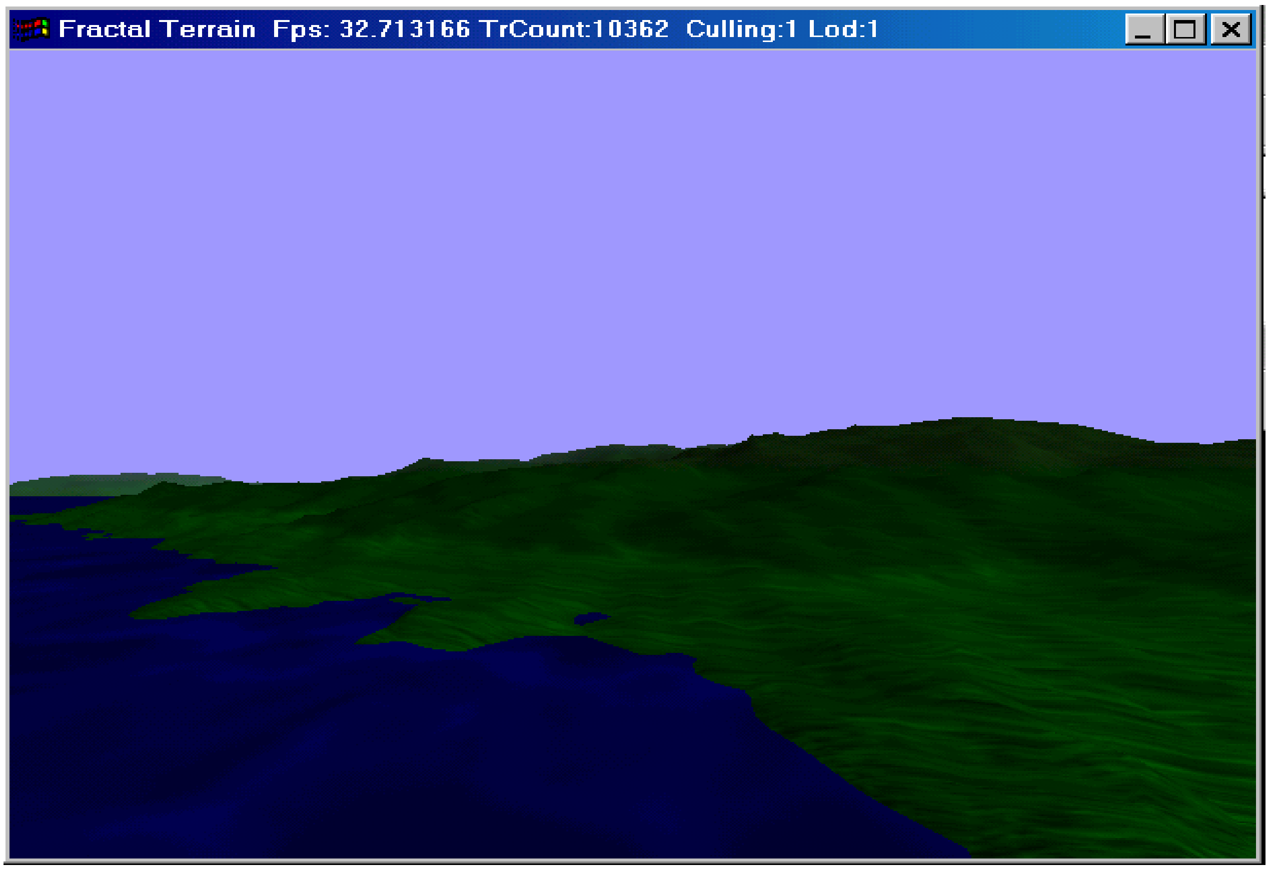

Fractal Terrain Fps: 16.107729 TrCount:21636 Culling:1 Lod:1

Figure 5.5 Birds eye view on different LevelsOf Detail


Fractal Terrain Fps: 32.713166 TrCount:10362 Culling:1 Lod:1

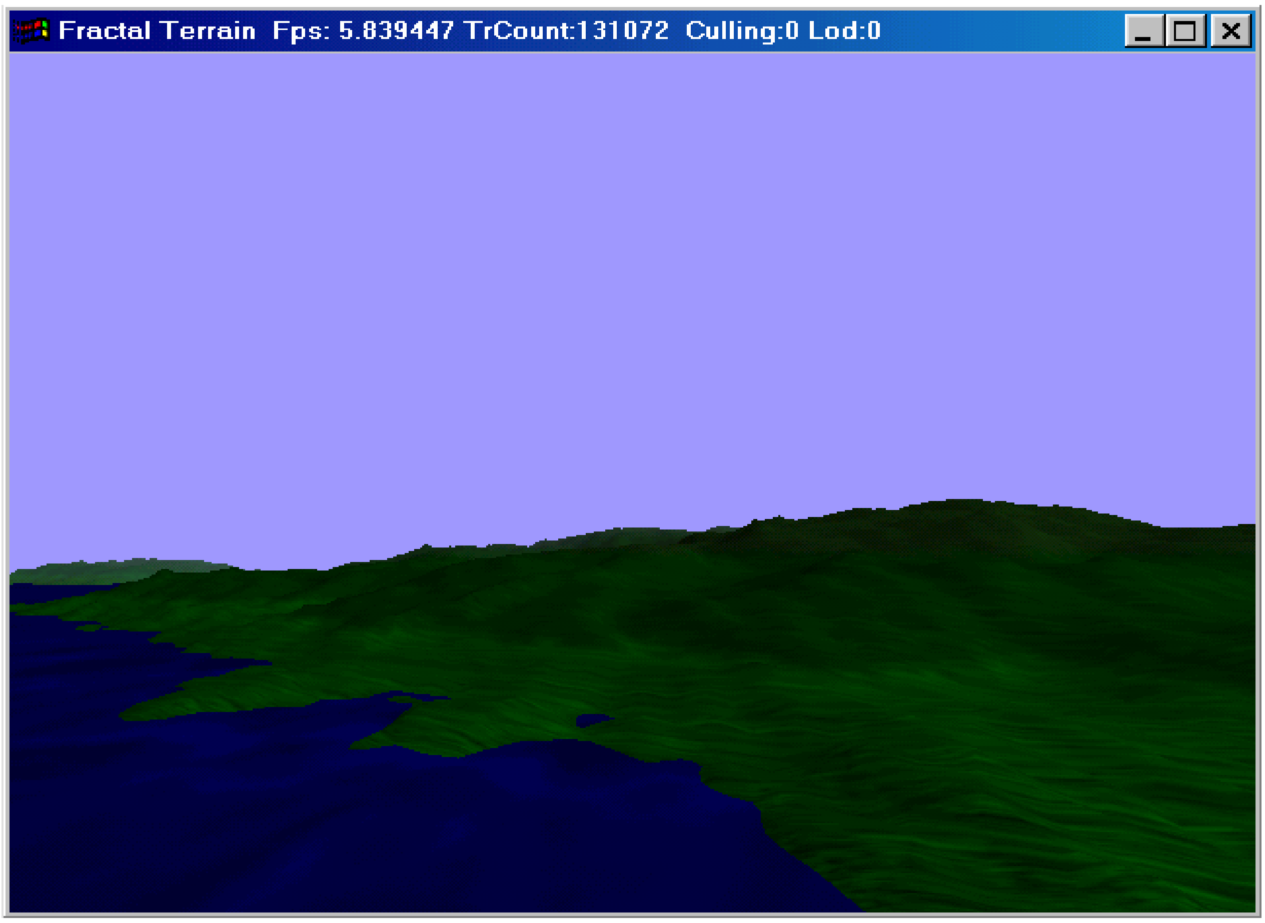

Fractal Terrain Fps: 5.839447 TrCount:131072 Culling:0 Lod:0

Figure 5.6 Actual view with LOD and Culling switched on (top) and off (bottom)

5.3 The Pipeline

The terrain engine can be seen as a simplification pipeline, which resides in front of the actual rendering (OpenGL) pipeline. It produces the geometry, simplifies it using specialised algorithms and finally sends the reduced geometry and instructions on how to render it down the graphics pipeline. Figure 5.7 shows a flow chart of the pipeline.

Note that at the triangulation phase the remaining blocks are triangulated using triangle strips. A greedy insertion (Evans et al. 1996) scheme was used to triangulate each block at various levels of detail. This scheme was implemented easily because of the grid representation used. On average 8-16 triangles are rendered with each strip.

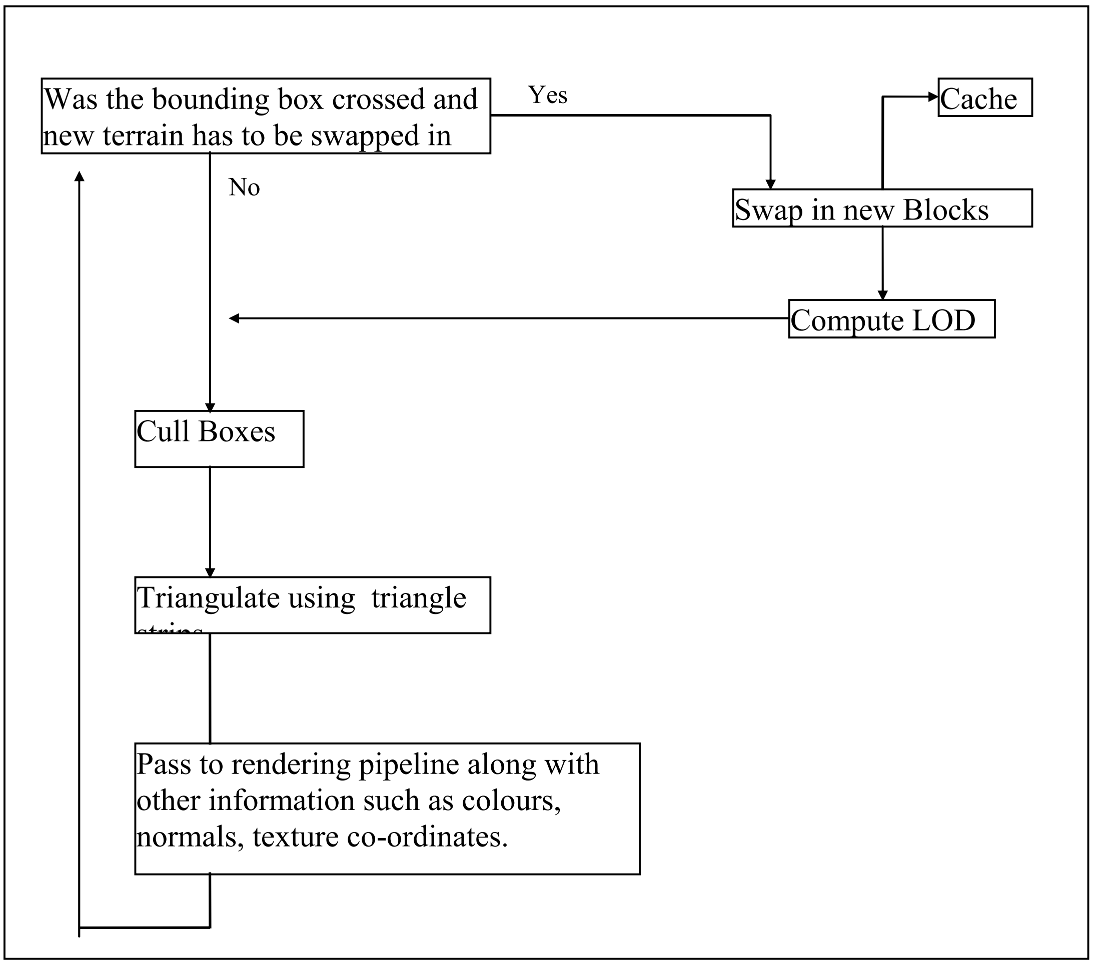

Was the bounding box crossed and new terrain has to be swapped in
Yes
No
Cache
Swap in new Blocks
Compute LOD
Cull Boxes
Triangulate using triangle strips
Pass to rendering pipeline along with other information such as colours, normals, texture co-ordinates.

Figure 5.7 Terrain Engine Pipeline

5.4 Environmental Aspects

Although an important part of any terrain engine is to generate and simplify the actual terrain geometry another important area which should not underestimated is the use of additional techniques, which increase the environmental cues, enhance the features of the terrain and finally add to a better visual quality. These features and techniques are called Environmental Aspects.

5.4.1 Colour Mapping

The interior colour of the triangles drawn is determined by interpolating the colour values at its vertices. Therefore to determine the colour value at the vertices is an important task.

The colour values in the terrain engine for a particular vertex are determined primarily by its height, position relative to the light source and the angle of its normal.
The terrain is divided in to 3 parts lowlands, highlands and mountains. Each of these parts is again divided into 2 parts respectively. Furthermore terrain vertices are additionally tested for their angle between their normal and x-z plane. If this angle is below 28° they are considered cliffs, and below 48° and above 28° they are categorised as slopes.

For each category a different colour value is specified from a default palette, but this can be adjusted by the user. Furthermore, the exact height boundaries where each category starts and ends can also be defined by the user. By default the distribution is as follows.

- Mountain High: Below 100%, white colour.
- Mountain Low: Below 85%, light grey colour.

- Highlands High: Below 70%, brown colour.
- Highlands Low: Below 50%, greenish brown colour.

- Lowlands High: Below 30%, medium green colour.
- Lowlands Low: Below 10%, light green colour.

- Cliffs: Below 28°, dark grey colour.
- Slopes: Between 28° and 48°, greenish grey colour.

The percentages are given for range between Max Height and sea level. See Colour Plate I.1 (Appendix A) for an example of colour mapping as it is used in the engine (default settings).

5.4.2 Texturing

Textures play an important role in the visual quality and natural appearance of terrain. The textures used in the engine are blended (modulated) with the underlying colour values of the triangles to give optimal results.
Sea and clouds are modelled by applying appropriate textures to meshed or flat respectively planes.

Dynamic texturing

Because natural phenomena such as clouds or sea are never static and always moving, a special texture technique is applied. This technique is called "Dynamic Texturing" and is useful for representing dynamic objects. The basic difference is that instead of applying one static texture with the same texture co-ordinates every time, the actual texture co-ordinates get translated over time. This results in the texture appearing as it is moving giving valuable hints to the user about the nature of the object represented (water, clouds).

Alpha blending

Many natural objects such as clouds are partly transparent. Therefore someone can see through the clouds either to the sky or the ground. Traditionally, implementing this transparency effect on a low-end graphics workstation was not feasible. Now, with the advent of accelerated graphics boards, it is possible to model transparent layers in real time. The clouds in the terrain engine are optionally modelled as partially transparent textures. Therefore in addition to moving clouds, there is also the possibility to see through them giving unique visual results.

See Colour Plates I.2, I.3 (Appendix A) for examples on the above techniques.

5.4.3 Waves

Another optical cue, which gives valuable information about the objects and their nature, is animation. Wave animation for the sea-plane is modelled by triangulating the sea plane and distorting each of the vertices by a specified amount. The amount of distortion is computed by mapping a sinusoidal function, which changes phase and values over time, over this 2D mesh. The characteristics of the sinusoidal function (amplitude, phase and frequency) can be controlled so as to model different conditions (still water or storms).

5.4.4 Fog

Rendered images tend to seem unrealistically sharp. While antialiasing makes objects appear more realistic it is still very computationally intensive. By adding fog we get much more realistic effects making objects fade into distance. Fog can be used to model atmospheric effects like haze, mist, smoke or pollution.

Objects that are farther from the viewpoint begin to fade into the fog colour. The terrain engine uses fog to model atmospheric attenuation and hide the far clipping plane.

Underwater environment

Fog is also used for modelling the underwater environment. The terrain engine enables the user to fly freely around and to dive under the sea. The parts of terrain which are under the sea have to be modelled in a different way to give the feeling that an underwater environment has been entered. Greenish-Blue fog with high density is successful used to simulate this effect. Colour Plate I.4 (Appendix A) shows an underwater scene of the engine.

6. Results

This chapter will summarise the results from the algorithms implemented and the performance of the terrain engine. The performance will be measured by the number of triangles actually rendered and the frames per second achieved .

6.1 Performance Results

The underlying philosophy of our approach is to reduce the overall load of the system, by not drawing what is not seen and to draw at different levels of resolution what is inside the view frustum.

Measures were taken from typical data sets produced by 7,8,9 and 10 iterations. The data points were mapped on a 24x24-kilometre area, with intervals depending on the points produced. The pixel resolution used was 1024x768.

The system used for testing the terrain engine was a Pentium II (350 MHz) with 128 Mb RAM system memory and with a 16 Mb 3D graphics accelerator (Riva TNT) card. Although the choice of the system does not affect the final number of triangles which must be rendered, it affects the update rates (frames per second) that can be achieved.

Figures 6.1, 6.2, 6.3, 6.4 show performance data taken from a low altitude flight over a period of 400 seconds for landscapes generated with a differing numbers of iterations. The triangle count and frame rate were sampled every 2 seconds. Each figure shows the number of triangles rendered over time, the average frames per second and the total number of triangles of the original area.

Keeping in mind that current low-end systems are able to display about 30-40 thousand triangles at real-time (>25 fps) or interactive (>10 fps) frame rates, the results show immediately the good performance of the terrain engine.

With 7 iterations (Figure 6.1) the average frame rate was 86 fps and 3292 triangles were rendered with Culling and Level Of Detail enabled. Without any Culling and Level Of Detail techniques applied the frame rate would be 24 fps and 32.768 triangles had to be rendered.

The results for other number of iterations as shown in Table 6.1 illustrate even greater reductions in the numbers of triangles rendered and increases in the frames per second achieved.

Iterations	**Lod and Culling techniques enabled**	**Average number of triangles rendered**	**Average frames per second achieved**
7	Yes	3,292	86
7	No	32,768	24
8	Yes	11,816	32
8	No	131,072	6
9	Yes	45,000	9.5
9	No	524,288	1.7
10	Yes	169,000	2.5
10	No	2,092,152	

Table 6.1 Comparison of results with and without Culling and Level Of Detail techniques applied

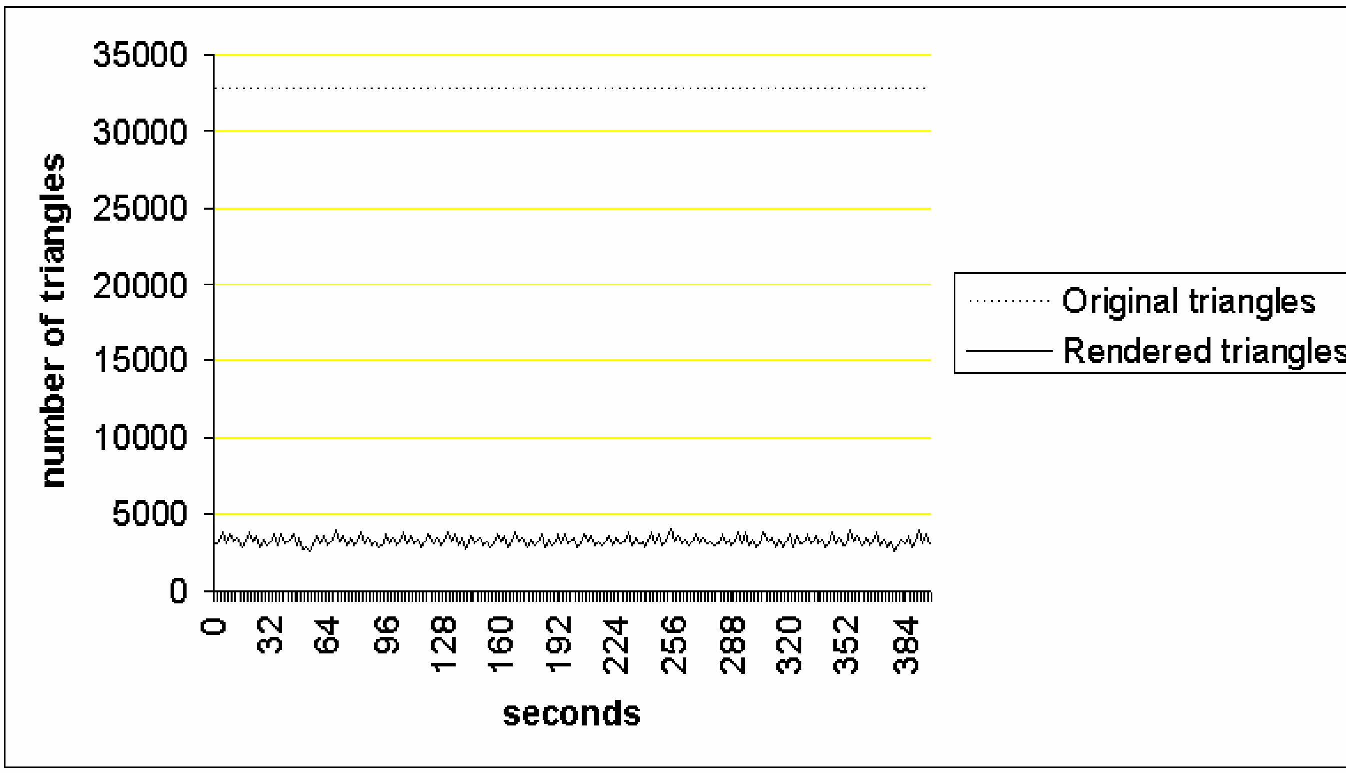


Figure 6.1 Comparison of number of triangles rendered to number of triangles in the original model for 7 iterations.

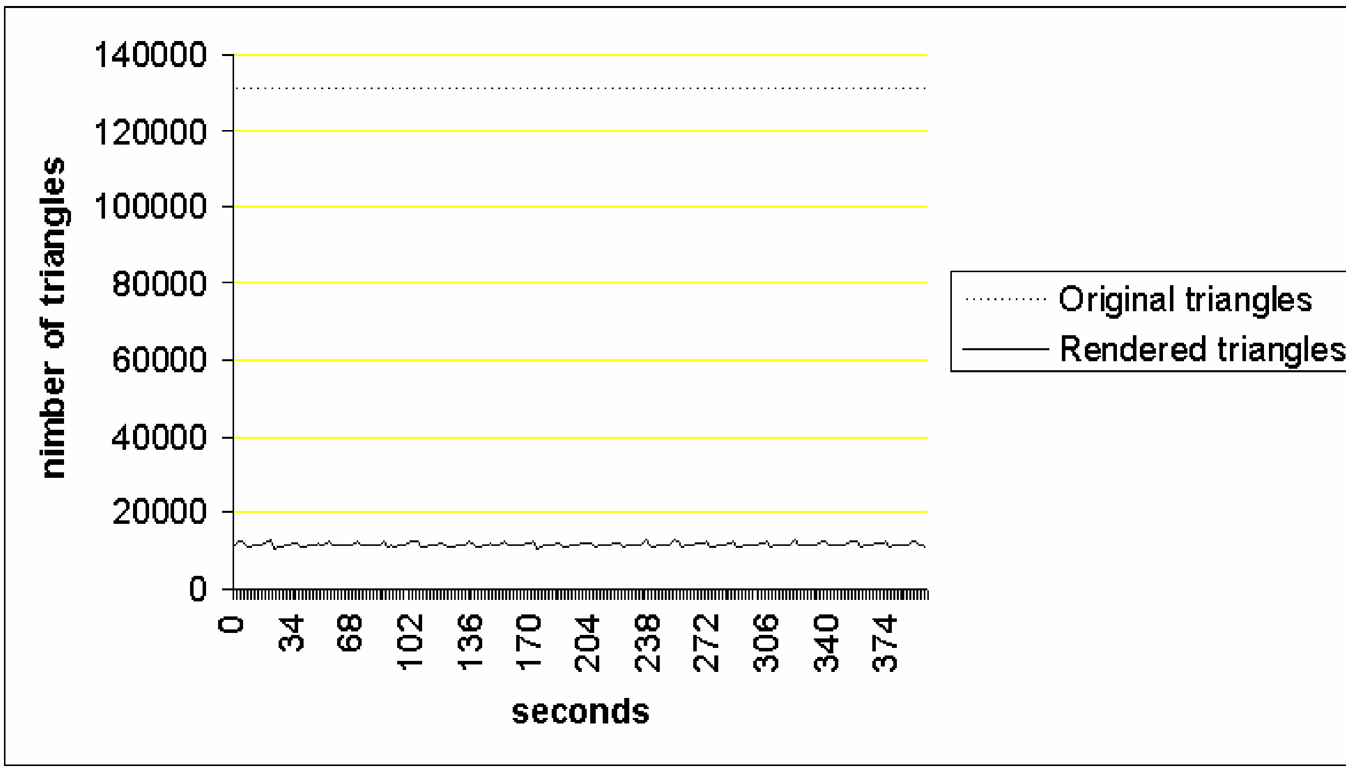


Figure 6.2 Comparison of number of triangles rendered to number of triangles in the original model for 8 iterations.

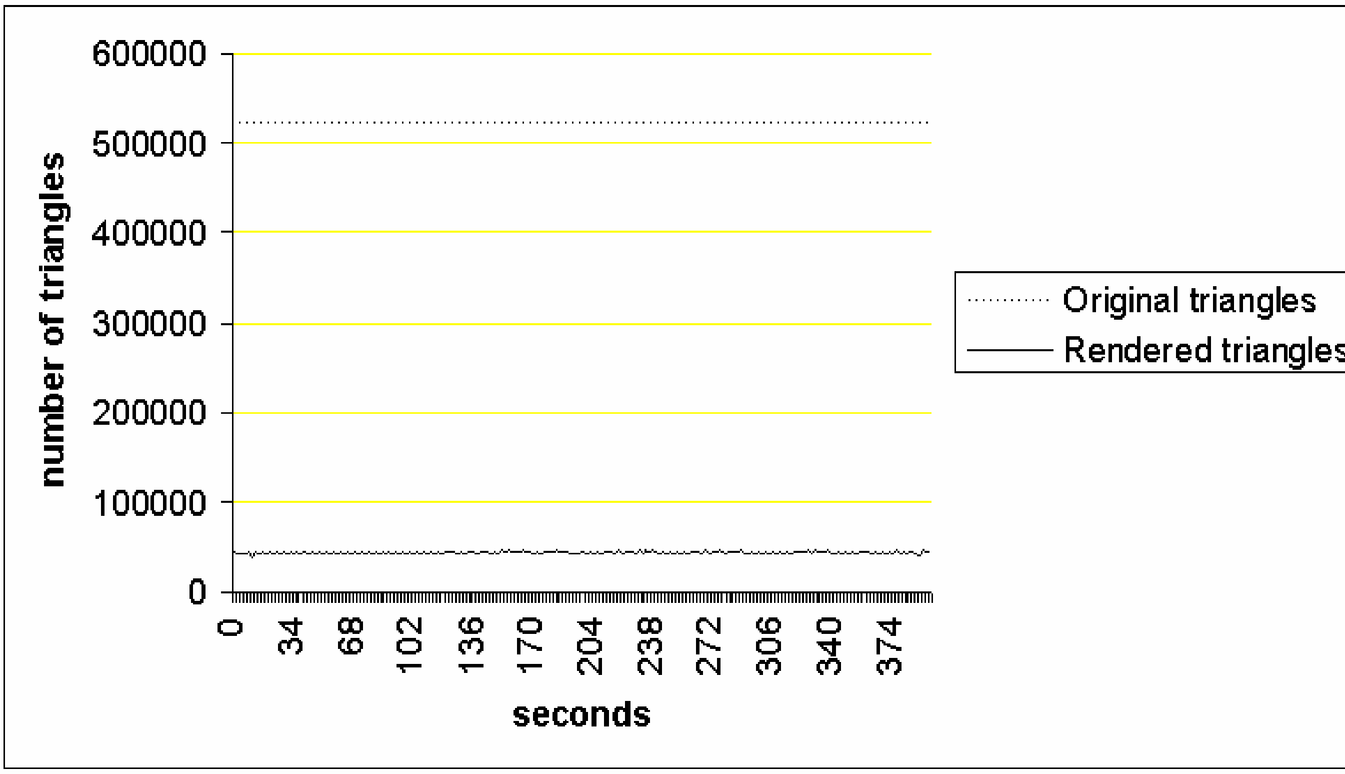


Figure 6.3 Comparison of number of triangles rendered to number of triangles in the original model for 9 iterations.

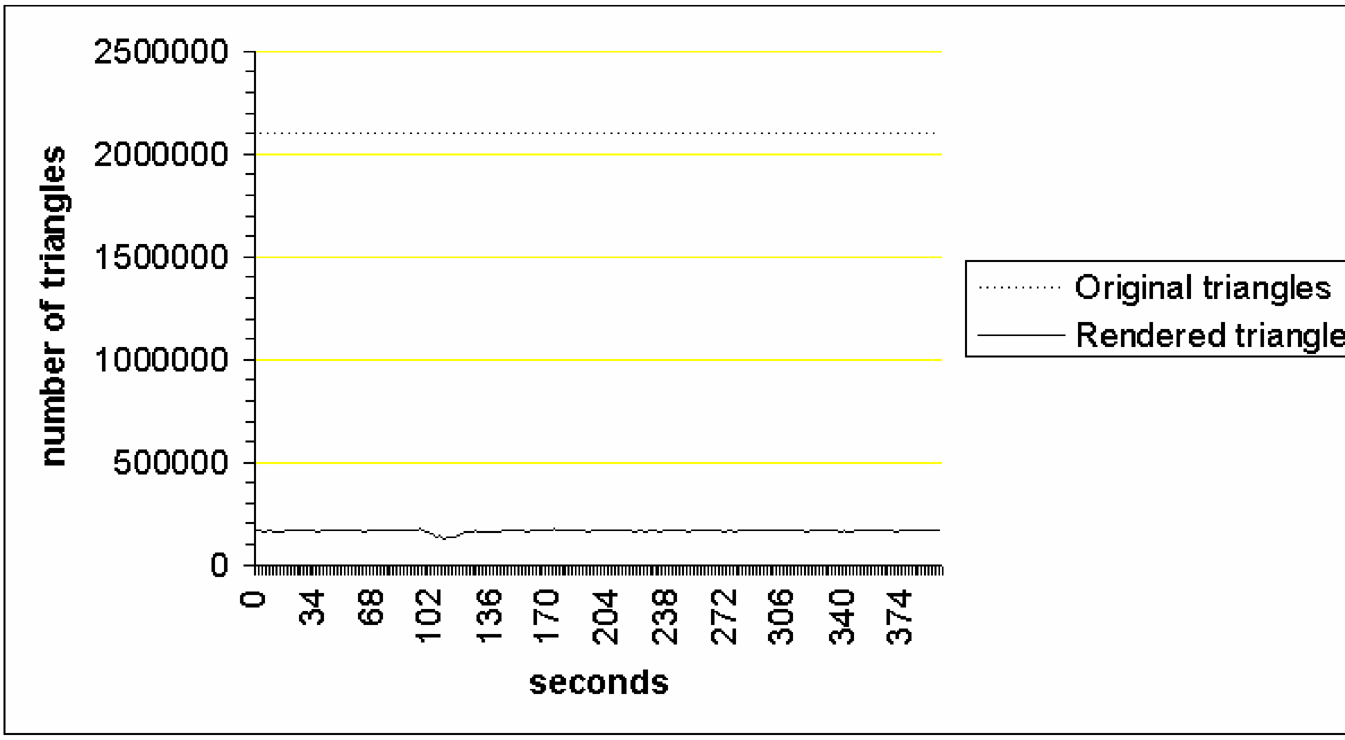


Figure 6.4 Comparison of number of triangles rendered to number of triangles in the original model for 10 iterations.

The techniques used have reduced the number of polygons rendered per frame by an order of magnitude for 7 iterations and almost 2 orders for 8,9 and 10 iterations.
Thus the terrain engine is able to produce and visualise highly detailed landscapes in real-time, not feasible before on low-end systems, without any change on hardware specifications.

Figure 6.5 illustrates the difference with side by side images of the same scene in wireframe. The left image is rendered at full resolution and the right with culling and level of detail algorithms applied. Figure 6.6 shows the same scene textured and with environmental aspects enabled. Colour Plate I (Appendices) presents more screenshots of the terrain engine in action.

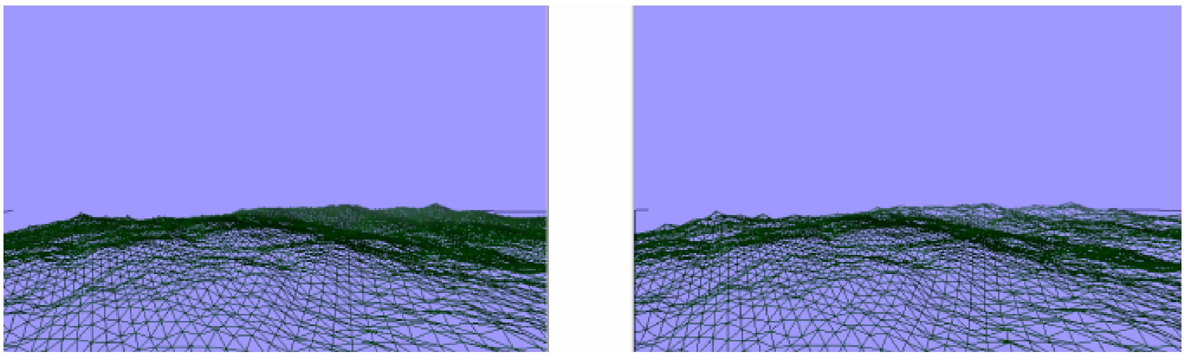


Figure 6.5 The same frame in full resolution (left) and with culling and level of detail (right)

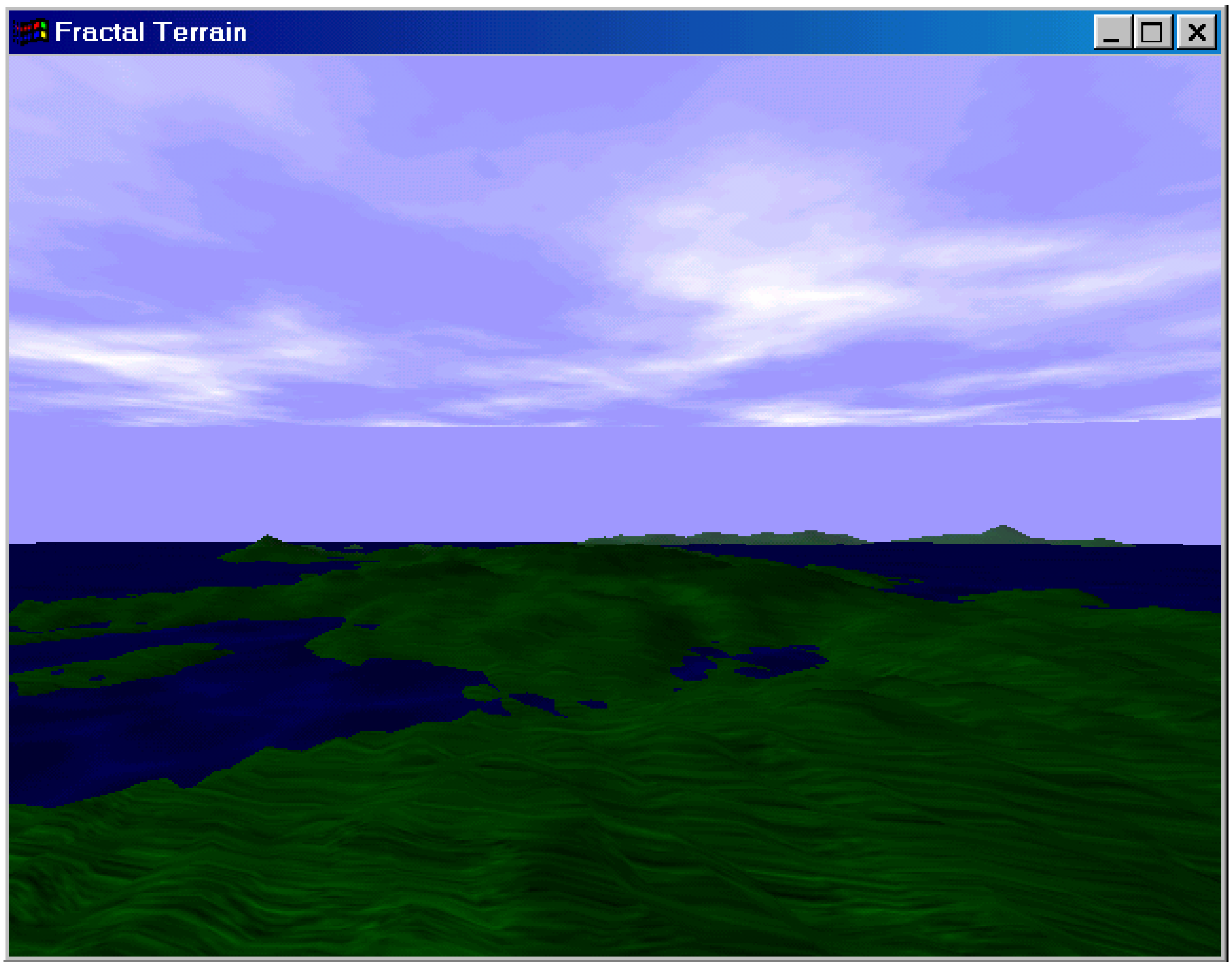


Figure 6.6 The above frame on the right with texturing and environmental aspects.

7. Future Work

The possibilities for extending this project into other areas are practically endless. There is always a better algorithm or some new effect, which can be applied. In this chapter areas for further work and experimentation are highlighted.

7.1 Eliminating popping

One mostly unavoidable artefact which is inherent in many Level Of Detail algorithms is the so called "popping effect". This effect is caused when one area or block, which was previously represented with lower resolution, is now represented with a higher resolution because the user moved towards it. The differences and the sudden change in the representation as new vertices are inserted during this switching between two levels of detail are noticeable to the user as "popping".

One technique to reduce the undesired optical effects caused by popping is to make alpha blended transitions between the low detail block and high detail block over a number of frames. Instead of just switching straight to the other level of detail, the transition is made gradually over some frames blending from 100% low detail block and 0% high detail block to 0% low detail block and 100% high detail block. Using this blending technique the sudden popping will be reduced to an extent where it is not noticeable (Woo et al. 1997).

The other technique which avoids popping completely but is more tedious to implement is called "geomorphing". This technique applies the same principle by creating a gradually transition from one Level Of Detail to another, but rather than use blending the actual vertices are moved from the position where they were originally to the new one. The vertices are thus morphed over different frames.

7.2 Continuous Level Of Detail

The block based LOD algorithms have certain disadvantages, which were discussed, in the previous section. These disadvantages can be completely overcome by employing a different Level Of Detail scheme which is called "continuous level of detail" (Lindstrom et al. 1996). In this technique instead of adaptive subdivision we use edge collapses and vertex moves to produce the new level of detail. These algorithms are far more complex and employ heuristic techniques to determine which operation should be performed. Furthermore it would be more difficult to apply them to the fractal terrain technique used.

The current LOD algorithm is only based on distance and produces non-optimal simplifications. Using schemes which calculate the exact screen space error when a certain resolution is used and also consider the actual area of projection onto the screen, would give more optimal results reducing the triangle count even further.

7.3 Procedural textures

Although procedural textures are still prohibitively expensive for real time use, many natural phenomena could have been modelled using them. Instead of having a standard cloud or sea texture, these textures could be produced by procedural techniques and changed at real time using shuffling or turbulence functions to model dynamic objects.
Also the use of fractal algorithms to produce plants, trees and 3D clouds is well documented and could be used to model objects on the ground..

7.4 More environmental aspects

The methods used to model natural phenomena and environmental aspects proved to be very successful. But there are still many techniques which can be applied such as lightmaps, underwater caustics, rain, thunder, wind etc. to produce more realistic effects. Furthermore the addition of agricultural detail such as plants and trees, or animals such as fishes or birds can produce stunning and lively environments.
Clearly most of these techniques require physics and physically based modelling rather than more actual traditional graphics animation, but it is in this direction in which most aspects concerned modelling virtual environments is likely to move (Sanchez 1999).

7.5 The world is round

The world, which is modelled with our terrain engine, is infinite in all directions and flat. An interesting and quite easily achievable extension would be to alter the engine to accommodate spherical worlds (i.e. planets). With this extension the user could travel around a whole planet and arrive again at the point where he started. Planetary flights and navigation would be a fascinating application area.

The changes which have to be made to the current engine to incorporate this functionality are not difficult to achieve. The fractal terrain algorithm can easily be altered to accommodate spherical terrain generation. The new points have to be generated for a sphere and the size and maximum level of detail for an area has to be dependent on the altitude of flight. On very high altitude (the user is outside the planetary atmosphere), almost all points facing the viewer should be seen at the lowest level of detail. Dixon et al. (1994) has dealt with planetary terrain generation using fractal methods.

8. Conclusion

This chapter discusses the conclusion drawn from this project and the achievements made.

8.1 Achievements

Before drawing any conclusion the final achievements should be assessed. The major achievement of this project was the production of a fast and robust terrain engine, which is able to render at real-time frame rates (>30 frames per seconds), large terrain data sets on low-end systems.

Although this might have been achieved previously the ingenuity of our terrain engine is that it is able to produce practically unlimited terrain data with natural characteristics. With the use of the specialised and controllable fractal terrain generation algorithm, no large data sets have to be stored or read from files. Practically infinite types of terrain can be generated and the user can fly through never before seen “alien” or “earthly” looking landscapes.

All of the overall project aims and objectives were met and almost all of the extensions, with the exception of the construction of plants, were implemented successfully. The extensions of plants into the landscape were left out because they require a new methodology and algorithms (Level Of Detail on objects) which was not feasible to achieve under the current time constraints.

The following sections discuss the success (or inefficiencies) of the particular parts of the engine.

8.2 Fractal terrain generation and paging

The terrain generation and paging element of the terrain engine worked very well. The algorithm for the terrain generation was based upon fractal techniques, which were adapted to produce controllable terrain without obscuring the nature and advantages of the actual algorithm.

An efficient and flexible terrain paging scheme was developed and coupled with the fractal terrain generation algorithm to produce infinite landscapes without any loading of the terrain from data files. With the efficient data structures and caching techniques used, the cost for the generation of the terrain data was minimised to a degree not noticeable by the user and without introducing significant delays. The memory consumption is kept minimal due to procedural techniques and can therefore handle much larger terrain data than normal terrain engines.

The quality of the terrain was very good, and can be adjusted to produce various types of landscapes. By changing the terrain generation factors like roughness, iterations or seed values, a huge gamma of terrain landscapes can be created. These range from rough and strange otherworldly to natural earth like landscapes almost any type can be produced.

The main deficiency of this part of the engine is the weaknesses exhibited by the actual fractal algorithm used. The points are generated as regular square grids, which produce a non-optimal triangle count for a particular scene. The algorithm suffers also from artefacts due to its approximation to fractional Brownian motion. These are sometimes noticeable by the user. However in many cases these artefacts can be hidden successfully using texturing and by adjusting the fractal terrain generation parameters.

8.3 View Frustum Culling

The first optimisation measure implemented was view frustum culling. The technique implemented was based upon hierarchical bounding volumes. The bounding volume was used to cull away large parts of the landscapes, which are not in the viewing frustum.

The technique, which is used to cull away geometry, is tightly coupled with the way the actual terrain data is represented giving optimal results. The blocks, which are used for terrain paging, are also used to perform frustum culling. No additional data structures are needed and because of the arrangement of blocks in heap sorted quad- trees, large blocks of geometry which are behind the user are culled immediately. More expensive and detailed tests are performed only the for blocks which intersect the viewing frustum planes.

The implementation worked well and gave vast speed increases. Although the amount of tests, which have to be done by checking each Bounding Box can be quite expensive, these are reduced by the clever use of the terrain structures to avoid otherwise expensive tests.
Due to inaccuracies when reconstructing the actual frustum, that the OpenGL library uses, some bounding boxes, at the far clipping planes whose edges or sides just protrude the viewing frustum are culled. This effect occurs rarely and is barely noticeable because of the small actual projection of the terrain (due to the distance and size) on the screen.

8.4 Level Of Detail

The Level Of Detail algorithm used discerns itself for its simplicity and large savings in the final polygon count. It is also tightly coupled with the remaining terrain generation and culling algorithms. The simplification performed is block based, each block of terrain is therefore rendered at a particular resolution. The metric upon which the simplification is based is the distance between user and the lower levels of detail that are produced by adaptive subdivision of interior points.

No extra computation is performed for generating these levels of detail because they are already computed as part of the fractal terrain algorithm. The difference between various levels of detail is always in powers of 2 (8,4,2), thus resulting in massive savings.

This algorithm proved to be sufficient for terrain visualisation and the error between actual and simplified area was within acceptable limits (barely noticeable).
Although block-based algorithms such as this one produce good results and are simpler to implement they exhibit some disadvantages. The polygons produced from the simplified points for an area are not optimal due to the regular grid representation. Traditional triangulation algorithms such as Delaney triangulation or Triangulated Irregular Models would produce the same representation with fewer polygons.
Popping artefacts can occur when a block is switched from one level of detail to another due to their difference in representation. This effect is even more obvious if the points simplified have large height differences. These popping artefacts, although hidden by texturing, are especially visible with rough terrain or at high altitudes where the user has all the blocks in the frustum.

8.5 Environmental Aspects

The techniques and representations used to simulate natural phenomena and the environment were found adequate and successful. Not only did they enhance the overall visual quality and natural appearance but also contributed to the better immersion of the user to the environment.

Techniques such as dynamic texturing, fog, animation of water waves and alpha blending were employed in order to give realistic results. There are also other more expensive techniques such as light mapping, caustic rendering and physically based modelling which can be employed to give even more realistic results.

The observation was made that some of the techniques used overloaded the rasterisation units of the graphics card and resulted in a drastic reduction of the frames per seconds rendered. Therefore the application of such effects must be undertaken with care and with continuing evaluation of the speed decrease for every feature added.

8.6 Time Management

The overall time scales as described in the Gantt chart were kept. Short delays were caused at the end of the project. These resulted from the large amount of complex algorithms which had to be researched at the beginning. The adaptation of the original fractal algorithm to an infinite area which proved very demanding and finally the development of fast techniques for rendering. Each possible interaction, combination and implication had to be found to get optimal results.

Clearly projects of this size never end, there is always something to do better, but finally a compromise has to be made in order to keep the deadlines.

This project has proven successfully that the time has come where with the use of good algorithms, highly impressive real-time graphics applications (such as terrain rendering) can be produced on low-end systems. Furthermore it shows the importance of intelligent algorithms, efficient coding and optimisation. Skills, which have been lost or forgotten by the high level, approaches to software engineering used today.

9. References

(Agarwal and Suri 1994) Agarwal P.K and Suri S. 1994
Surface approximation and geometric partitions.
Proceedings ACM-SIAM Symposium on Discrete Algorithms,
pp. 24-33

(Airey et al. 1990) Airey J., Rohlf J., Brooks F. 1990
Towards image realism with interactive update rates in complex virtual building environments.
Computer Graphics Vol. 24, Number 2, pp. 41-50, March 1990

(Assarson and Moeller 1999) Assarson V. and Moeller T. 1999
Optimised View Frustum Culling Algorithms.
Chalmers University of Technology
Department of Computer Engineering
Technical report 99-3, March 1999

(Clark 1972) Clark J.H. 1972.
Hierarchical geometric models for visible surface algorithms.
Communications of the ACM, October 1972.
Vol. 19, Number 10, pp. 547-554

(Cohen et al. 1996) Cohen J., Varshney A., Manocha D., Turk G., Weber H., Agarwal P., Brooks F., Wright W. 1996
Simplification Envelopes.
Computer Graphics (SIGGRAPH '96 Proceedings)

(Cohen-Or and Levanoi 1996) Cohen-Or D. and Levanoi Y. 1996
Temporal Continuity of levels of detail in delauney triangulated terrain.
In Proc. Visualisation '96, pp. 37-42.
IEE Comput. Soc. Press, 1996.

(Da Haemer and Zyda 1991) Da Haemer M.K, Zyda M.J. 1991
Simplification of objects renderd by polygonal transformations
Computer and Graphics 15(2), pp. 175-184.

(De Floriani and Puppo 1995) De Floriani L. and Puppo E. 1995

Hierarchical Triangulations for Multiresolution Surface Description
ACM Transactions on Graphics 14(4),
October 1995, pp. 363-411.

(Dixon et al. 1994)	Dixon A.R., Kirby G.H., Willis D.P.M. 1994 Towards Context Dependent Interpolation of Digital Elevation Models. Computer Graphics Forum (EUROGRAPHICS 94) Vol. 13, Number 3, pp.23-32.
(Duchaineau et al. 1997)	Duchaineau M., Wolinsky M., Sigeti D., Miller M., Aldricht C., Mineev-Weinstein M. 1997 ROAMing Terrain: Real Time Optimally Adapting Meshes. URL "www.llni.gov-graphics-ROAM" (13-4-99)
(Eck et al. 1995)	Eck M., DeRose T., Duchamp T., Hoppe H., Lounsbery M., Stuetzle W., 1995 Multiresolution analysis of arbitrary meshes. SIGGRAPH '95.
(Evans et al. 1996)	Evans F., Skiena S., Varshney A. 1996. Optimising triangle strips for fast rendering. In Proc. Visualization 1996, pp319-326. IEEE Computer Soc. Press.
(Falby et al. 1993)	Falby J.S, Zyda M.J, Pratt D.R., Mackey R.L. 1993 NPSNET: Hierarchical Data Structures for Real-Time 3D Visual Simulation. Computer and Graphics 17(1), pp. 65-69, 1993
(Flavell 199)	Andrew Flavell 1999 Run-Time Mip-Map Filtering Gamasutra Magazine http://www.gamasutra.com/features/programming/19981211/mipmap01. htm (16/9/99)
(Foley et al. 1990)	Foley J., Van Dam A., Hughes J., Feiner S. 1990 Computer Graphics: Principles and Practice

Addison Wesley, Reading Mass 1990

(Fournier 1982) Fournier A. 1982.
Computer Rendering of Stochastic Models.
Communications of the ACM, June 1982
Vol.25, Number 6, pp 371-384.

(Fuch et al. 1980) Fuch H., Kedem Z., Naylor B. 1980
On Visible Surface generation by A Priori tree structures
Proc. Of SIGGRAPH, Volume 14(3), pp. 124-133, July 1980

(Funkhouser et al. 1992) Funkhouser T.A., Sequin C.H., Teller S.J. 1992.
Management of large amounts of data in interactive building walkthroughs.
Computer Graphics, October 1992.
Vol. 25 , pp.11-20

(Garland and Heckbert 1995) Garland M. and Heckbert P.S 1995
Fast Polygonal Approximations of Terrain and Height Fields.
MSc Thesis, Carnegie Mellon University, September 1995.

(Garland and Heckbert 1994) Garland M. and Heckbert P.S. 1994.
Multiresolution Modelling for fast rendering.
Proceedings of Graphics Interface
pp. 43-50, May 1994.

(Garland and Heckbert 1997) Garland M. and Heckbert P.S. 1997
Surface simplification using quadric error metrics
SIGGRAPH 1997 Proc, August 1997

(Greene 1994) Greene N. 1994
Graphics Gems IV, Heckbert P.S
Gem 1.7, pp. 74-82
Academic Press Inc., 1994

(Hearn and Baker 1992) Hearn D. and Baker P. 1992.
International Edition Computer Graphics, C Version.

	Prentice Hall International Inc., Second edtion, pp. 362-378
(Hoppe 1996)	Hoppe H. 1996. Progressive Meshes. In SIGGRAPH 96 Proc., pp. 99-108, August 1996.
(Hoppe 1997)	Hoppe H. 1997. View-dependent refinement of progressive meshes. In SIGGRAPH 97 Proc., August 1997.
(Hoppe et al. 1993)	Hoppe H., DeRose T., McDonald J., Stuezle W. 1993 Mesh Optimization. Computer Graphics (SIGGRAPH 93 Proc.) Vol. 27, pp 19-26, August 1993
(Hudson et al. 1997)	Hudson T., Manocha D., Cohen J., Lin M., Hoff K., Zhang H. 1997 Accelerated Occlusion Culling using Shadow Frusta. Department of Computer Science, University of North Carolina, Proceedings of 13'th Symposium on Computational Geometry, June 4-6 1997
(Lindstrom et al. 1995)	Lindstrom P., Koller D., Hodges L., Ribarsky W., Faust N., Turner Gregory 1995 Level of Detail Management for Real-Time Rendering of Photo textured Terrain. Tech report GIT-GVU-95-06, January 1995
(Lindstrom et al. 1996)	Lindstrom P., Koller D., Ribarsky W., Hodges L.F., Faust N., Gregory A. 1996. Real-time continuous level of detail rendering of height fields. In SIGGRAPH 96 Proc., pp109-118, August 1996.
(Luebcke and Georges 1995)	Luebcke D., Georges C. 1995. Portals and Mirrors: Simple fast evaluation of potentially visible sets. In 1995 Symposium on Interactive 3D Graphics, pp. 105-106,April 1995.

(Mandelbrot 1982) Mandelbrot B. 1982.
Technical Correspondence.
Communications of the ACM, August 1982
Vol. 28, Number 8, pp 581-583

(Miller 1986) Miller G. 1986.
The Definition and Rendering of Terrain Maps.
Communications of the ACM, August 1986
Vol. 20, Number 4, pp. 39-48

(Miller 1995) Miller M.C. 1995
Multiscale compression of Digital Terrain Data to meet Real Time Rendering Rate Constraints.
PhD thesis, University of California, Davis.

(Peitgen and Saupe 1988) Peitgen H. and Saupe D. 1988.
The Science of Fractal Images.
Springer Verlag

(Rosenfeld 1982) Rosenfeld A. 1982
Multiresolution Image processing and Analysis.
Springer Berlin 1984.
Leesberg, VA, July 1982

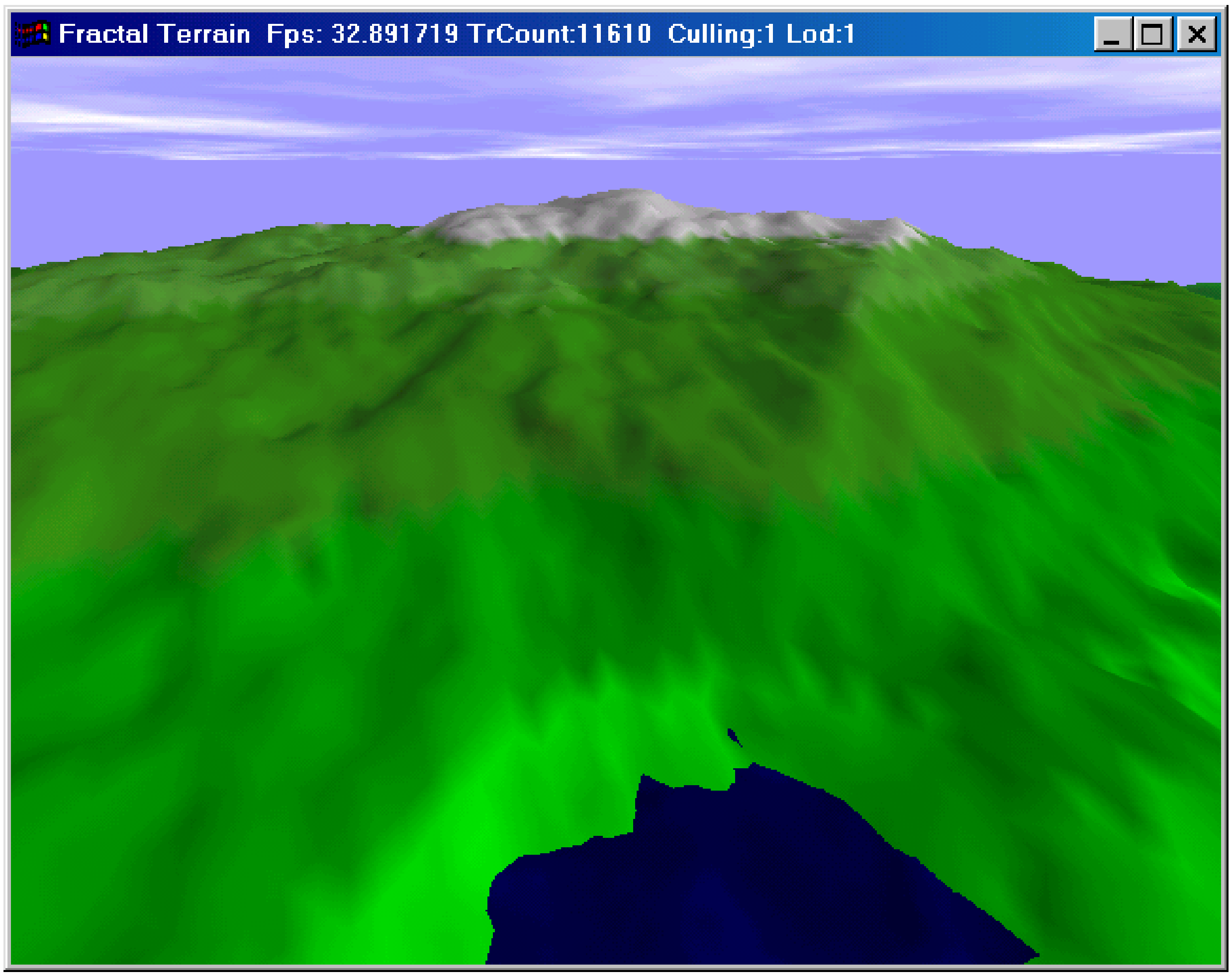
(Rossignac and Borel 1992) Rossignac J. and Borrel P. 1992
Multi-resolution 3D approximations for rendering complex scenes.
Technical report, Yorktown Heights, NY 10598, February 1992.

(Sanchez 1999) Sanchez Crespe D. 1999
SIGGRAPH '99 From a Game Development Perspective
Gamasutra Magazine
http://www.gamasutra.com/features/19990820/siggraph_01.htm
(16/9/99)

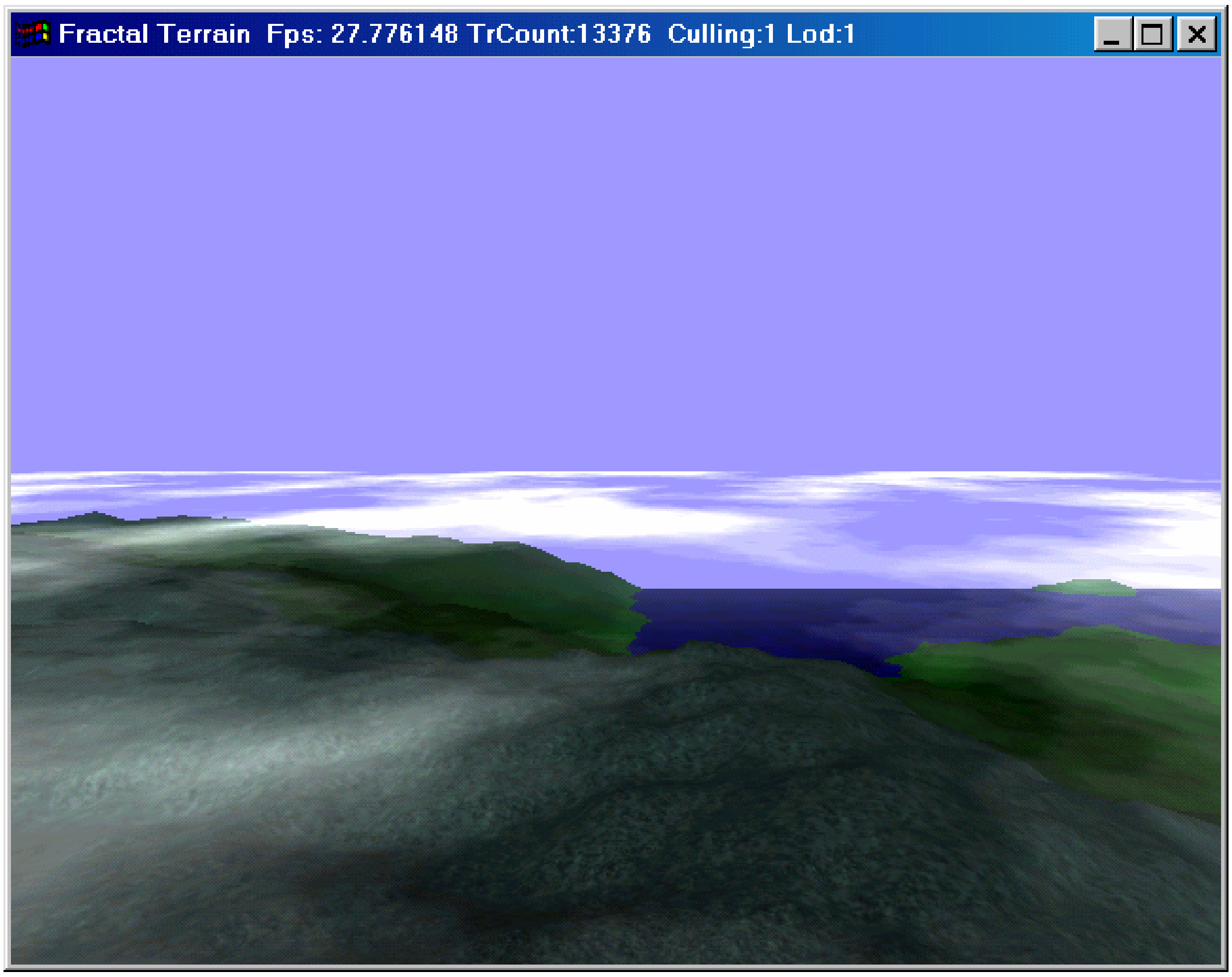
(Schroeder and Rosbach 1994)	Schroeder F. and Rosbach P. 1994 Managing the complexity of Digital Terrain Models Computers and Graphics , 18(6), pp. 775-783, 1994
(Schroeder et al. 1992)	Schroeder W.J, Zarge A.J., Lorensen W.E. 1992 Decimation of triangle meshes Computer Graphics (SIGGRAPH '92 Proceedings) 26(2), pp. 65-70, July 1992
(Schweighofer 1996)	Schweighofer T., Radatz K., Schmalstieg D. 1996 Infinite Virtual Landscapes Undergraduate Student Project Institute of Computer Graphics Vienna University Of Technology
(Teller 1992)	Teller S. 1992. Visibility Computations in Densely Occluded Polyhedral Environments. PhD thesis, Computer Science Division UC Berkeley, California 94720, October 1992. Available as Report No. UCB/CSD-92-708.
(USGS 1997)	Standarts for Digital Models: Part 1- General US. Department of the interion. National Mapping Division http://rmmcweb.cr.usgs.gov/public/nmpstds/demstds.html (16/9/99)
(Williams 1993)	Williams L. 1993 Pyramidal Parametrics. Computer Graphics 17(3), pp. 1-11, July 1983
(Woo et al. 1997)	Woo M., Neider J., Davis T. 1997 OpenGL Programming Guide, Second Edition The Official Guide to Learning OpenGL, Version 1.1 Addison-Wesley Developers Press, 1997

Appendices

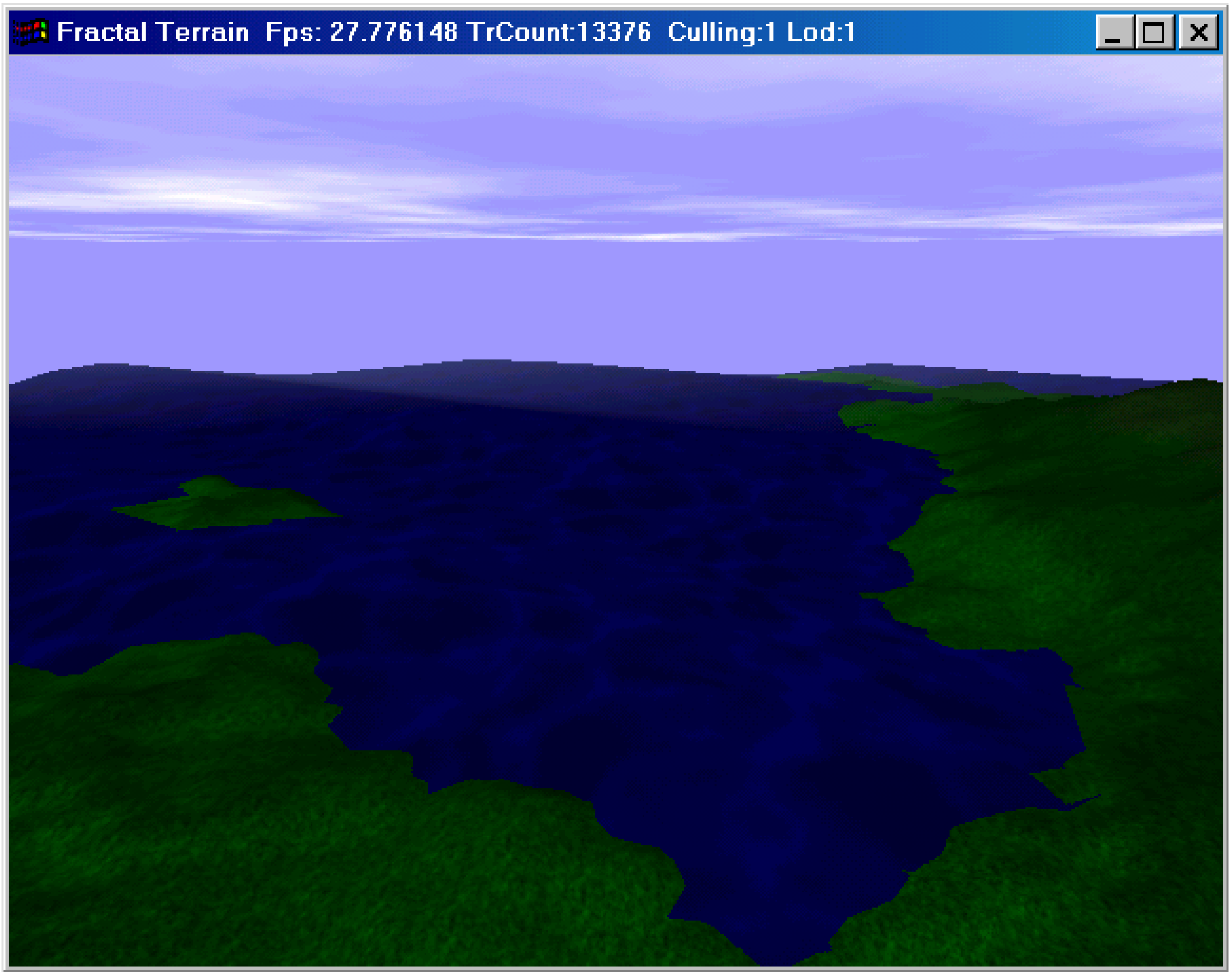
A. Colour Plate I


Fractal Terrain Fps: 32.891719 TrCount:11610 Culling:1 Lod:1

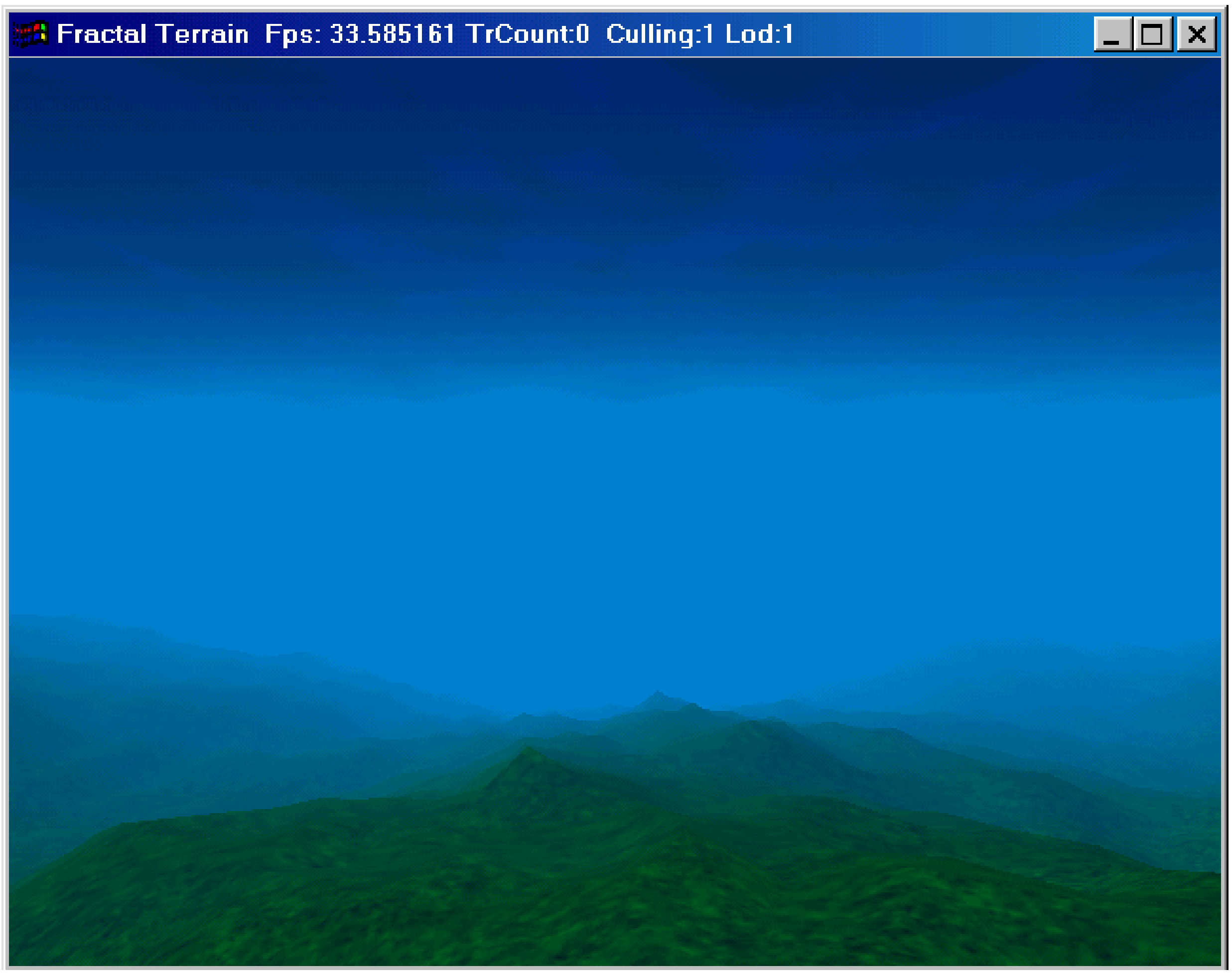
Colour Plate I.1 Colour mapping with texturing disabled, and environmental aspects (clouds, sea).


Fractal Terrain Fps: 27.776148 TrCount:13376 Culling:1 Lod:1

Colour Plate I.2 Screen shot above the clouds with transparent clouds, texture mapping enabled.


Fractal Terrain Fps: 27.776148 TrCount:13376 Culling:1 Lod:1

Colour Plate I.3 Waves using sinusoidal perturbation on textured sea.



Colour Plate I.4 Underwater view.