UNIVERSITY of DERBY

Derbyshire Business School

A project completed as part of the requirements for the

BSc (Hons) Computer Studies

entitled

Fast Real-Time Terrain Visualisation Algorithms

by

Gerhard Zlabinger

in the years 2003–2004

Abstract

This report is a comprehensive comparison of existing algorithms in the domain of Real-Time Terrain Rendering, focused on technical aspects as well as investigating applicability to military simulations and entertainment products.

This report also documents the implementation of a prototypical terrain rendering engine, capable of rendering terrain at continuous level of detail by using the the QuadTree algorithm described by Stefan Röttger et al in 1998.

Contents

List of Figures

Chapter 1

Introduction

The process of Terrain Visualisation possesses a multitude of applications in industry. Besides the most obvious and profitable field of application — the entertainment industry — the presentation of landscapes forms an integral part of a large number of scientific and military simulations.

The different domains were clearly delimited in terms of research as well as applications in the "early days" of research in the 1970s, a consequence of the tremendous costs for rendering hardware those days.

This has changed significantly as powerful hardware has become cheap in recent years. Constraints in graphics research and development have become blurred or have even vanished completely. A current example for this development is NASA's latest Mars mission, which uses nVidia Graphics hardware to display geographical data collected by the Mars rover Spirit. nVidia — a company that is specializing in graphics hardware for the game market — enables "NASA scientists to interactively plan rover movements using 3D photo realistic views of the surface" on Mars (nVidia 2003). For another example, consider the ROAM algorithm by Mark Duchaineau et al. Although its development was funded jointly by the U.S. Navy and Airforce, the popular computer game TreadMarks

uses an implementation of ROAM to render the environment (Turner 2000).

Processing capacity of modern CPUs is still the restricting factor in application and game development. Both, simulation and game developers have great interest in making their applications as realistic as possible. Thus, the driving thought behind terrain rendering algorithms is to create faster and more efficient graphics algorithms, allowing other costly parts of the simulation components, like artificial intelligence or a physics engine, to consume more processing resources.

A great variety of algorithms has emerged in the last ten years, introducing many interesting and efficient ideas. This project report gives an overview of the most significant algorithms that have been published on this topic.

1.1 Aims and Objectives

- Perform research on existing algorithms/approaches

  The report contains a comprehensive literature review on the topic of terrain rendering algorithms, identifying internationally respected authorities and giving an overview of their work.

- Comparison of different Terrain Visualization Algorithms

  The project includes a critical evaluation of the various algorithms, which points out merits and demerits of different approaches. The results of this analysis are examined for their applicability to military and scientific purposes as well as applications in entertainment industry.

- Gain first-hand experience in engine implementation

  Design and implement a prototypical terrain rendering engine for the Win32 platform using OpenGL.

Chapter 2

Background

2.1 Definition of Terrain Rendering

There is no common definition of terrain rendering. For this report, terrain rendering is considered an integral part of an application concerned with the representation of landscapes on computer screens in real-time. In this respect, a landscape comprehends geographic characteristics of terrain such as hills and valleys, whereas it does not comprise objects (e.g. houses and trees) and also doesn‘t take any environment effects (e.g. cloud shadows) into account.

2.2 Historical Development

Prior to the early 1990s, terrain visualisation was a field of research mostly attended to by the military and institutes that could afford the necessary hardware to provide the processing power needed for rendering large landscapes at real-time. As personal computers began to grow more and more powerful about fifteen years ago, a development

which was accompanied by a respectable progress of graphics hardware capabilities, the entertainment industry also started developing solutions of its own.

Since 1990, a lot of papers have been published presenting many different approaches to the problem of rendering terrain at high detail in real-time.

2.3 Background

Although the methods for rendering terrain in real-time differ from each other in many ways, they share some basic concepts. The most important of those is the underlying data structure that is used to represent the landscape information in memory.

2.3.1 Heightmaps

All algorithms covered in this report project terrain data from a heightmap or heightfield (Watt & Policarpio 2001), (Pollack 2003).

Figure 2.1: A 256x256 heightmap

A heightmap is a regular grid, mostly square or rectangular. It can be thought of as a bitmap, in which x- and y-position in world coordinates are defined by the x- and y-position in the heightmap and the elevation at point(x/y) in the landscape is defined by the intensity[1] value of point(x/y) in the heightmap.

Illustration 2.1 shows a sample heightmap. Black areas have low elevation (valleys), white areas have high elevation (hills).

[1] intensity corresponds to color in a gray scale image

The transformation from map coordinates to world coordinates can mathematically be described as depicted in Figure 2.2.

This way of representation is very efficient, although the fact that the landscape is represented as a two-dimensional surface implies that three dimensional entities (caves, etc...) cannot be described by a heightmap. A particular point(x/y) can only have one unique elevation.

$$x_{world} = x_{map}$$

$$y_{world} = map(x_{map}/y_{map})$$

$$z_{world} = y_{map}$$

Figure 2.2: Heightmap transformation

However, using a regular structure to describe the landscape is superior to conventional methods of representation, i.e. polygonal or mesh representation. For example, it is very simple to find triangle strips in a square, regular grid. Many terrain visualisation algorithms that have emerged, e.g. ROAM or QuadTree algorithms, exploit the fact that the underlying data structure is a regular grid.

In a triangle strip neighbouring triangles share two common vertices. This way, the amount of vertices sent to the rendering hardware can be reduced from $3 * n$ to $2 + n$ (Watt & Policarpio 2001). Benchmark tests carried out by Marshall (2002) and Wright & Sweet (2000) show that an extensive performance gain can be achieved by using triangle strips.

Chapter 3

Classification of Algorithms

3.1 Voxel Graphics based Algorithms

One of the first attempts to create an efficient terrain rendering algorithm for personal use resulted in voxel graphics, discussed by Glassner (1990) and Arvo (1990). This approach casts a number of virtual rays from the view point through a projected screen onto the surface. The color of each pixel is determined by the color of the point where the ray that made his way through that particular pixel hit the surface. This algorithm worked well and produced spectacular results for that time (NovaLogic‘s Comanche: Maximum Overkill, released 1992).

An interesting aspect of voxel algorithms is that level of detail processing, which will be discussed in more detail in section 3.2.3 and section 3.2.6 is automatically performed. This means that objects that are far away from the user‘s view port are rendered with fewer pixels than objects that are near and require more visible detail.

However, due to the development of 3D Graphics Accelerator Hardware which is based on the rendering of triangles and polygons, the voxel graphics algorithm has many drawbacks

compared to approaches that were designed with the architecture of modern rendering devices in mind.

Besides the stated disadvantages on today‘s triangle based rendering hardware, the algorithm is currently under patent by NovaLogic (Abner 2000), preventing any use and improvements to that algorithm in commercial applications. Thus, it is only mentioned for historical reasons and not discussed any further.

3.2 Polygon based Algorithms

3.2.1 Brute Force Rendering

A method to render terrain data on 3D accelerated hardware which might seem obvious at first sight is called Brute Force rendering. This approach relies entirely on the acceleration capabilities of the rendering device. It works by storing the entire height information in one (very possibly vast) vertex buffer and leaving the whole task of optimisation and acceleration to the rendering hardware. There are some graphics hardware manufacturers (e.g. nVidia) who encourage this method of rendering landscapes. Although ensuring high visual detail of the resulting image, this method has several drawbacks in comparison to the algorithms described in the following sections. The most significant is the huge amount of data that has to be transferred to and processed by the graphics device every single frame. The bottleneck of modern graphics hardware not being the rendering speed but the bandwidth of the data transfer, this approach will result in slow performance compared to algorithms that reduce the amount of data to be processed, which are discussed in the next sections.

However, this algorithm might be usable for applications that don‘t need a very high level of detail, since it has very little CPU overhead. A fast brute force implementation

needs to contain very efficient clipping and culling algorithms if large data sets are to be handled (Maréchal 2001).

Modern Graphics Hardware offers a feature that can speed up brute force rendering significantly. Index Buffers (DirectX) or Vertex Arrays (OpenGL) (Microsoft 2001) are arrays of vertex, color and lighting data. Instead of issuing a single draw command for each vertex that is to be drawn, an application can store vertex and color information in arrays and draw them at once (or just a specified range of an array) with only one single function call. According to Marselas (2001), this method has two advantages :

1. it eliminates the function call overhead and

2. it enables graphics hardware to store vertex data in device memory.

Modern CPU architecture makes function calls cheap through prediction algorithms and similar techniques. The following example illustrates that the omitted function call overhead does not lead to a significant performance increase.

Calculations are based on following assumptions: a scene consists of 6500 triangles that have to be rendered every frame. Function calls and corresponding returns take three clock cycles on average on an Intel Pentium 4 processor[1] (Intel 2003), pushing three arguments for glVertex3f() and glColor3f() to the stack takes three more clock cycles, so in total 6 clock cycles are needed for making a function call[2]. The function calls have to be issued inside a control loop iterating over an array of color and vertex information, taking 20 cycles per iteration. A vertex possesses color information as well as texture coordinates. Consequently, three vertices have to be specified for each triangle by issuing three separate function calls (glColor(), glTexCoords() and glVertex()). This has to be done 6500 times.

[1] assuming the jump target resides in the processor cache

[2] this is a simplified calculation but sufficient for the purpose of demonstration

$$6500 * (3 * 3 * 6 + 20) = 481000 cycles$$

Figure 3.1: Function call overhead calculation

The overhead is calculated as shown in figure 3.1, resulting in 481000 CPU cycles for one frame. On a modern Pentium4 with 3.06GHz 481000 cycles take less than one millisecond to complete. This result clearly shows that vertex buffers have lost one of their advantages over the years as processors grew more powerful and function calls got cheaper.

However, the real power of vertex arrays lies within the possibility of the graphics hardware to store vertex data in device memory and therefore enabling vertex optimization as well as reducing bus traffic. The performance increase experienced by the author during an early stage of the engine prototype‘s development was more than 100% after enabling vertex arrays compared to separate rendering function calls for every single vertex in the scene.

This result confirms the observations of Pollack (2003). In the past, algorithms tried to leave as much work load possible to the CPU and to be GPU friendly. Due to advanced graphics hardware acceleration, algorithms work the other way round nowadays.

3.2.2 Culling

“The fastest polygons are those you don‘t draw.” (Microsoft Developer’s Network introduction DirectX, (Microsoft 2001))

Frustum Culling is the process of eliminating triangles from a scene that are not visible to the user from his current view point — that are not inside the view frustum (Picco 2003) as shown in figure 3.2. The “main recursion function is that which selects the faces to draw” as Watt & Policarpo (2003) point out. Therefore, frustum culling is crucial to

the performance of Brute Force algorithms, since all vertices are sent to the rendering hardware.

A very common approach to culling in general terms of 3D computer graphics are Binary Space Partitioning (BSP) Trees (Watt & Policarpio 2001). Such trees are a way of representing all objects in the scene. World space is sub-divided at each tree level as long as there is more than one object inside the node and the node is not empty. If only one object resides within a leaf node it receives a label corresponding with its node. This algorithm allows very fast and efficient culling strategies as well as collision detection mechanisms. There is no need to perform a visibility test on every single object in the scene, instead the visibility test is performed on the sub-divided space derived from the BSP tree. If a tree node is not visible, than all its sub nodes are not visible as well.

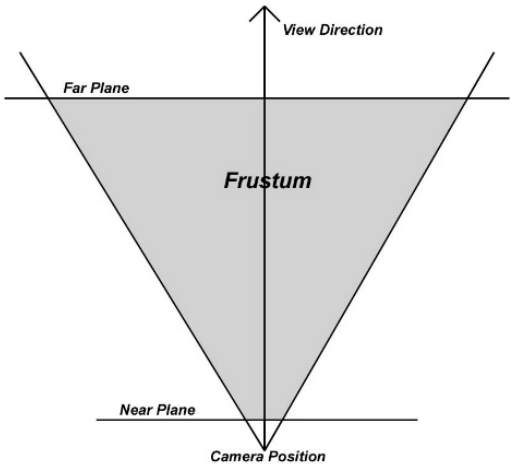

Figure 3.2: A viewing frustum, taken from Picco (2003)

Although BSP trees are most suitable for sub-dividing three dimensional space, variations of it — such as the QuadTree representation discussed later — can be used on two dimensional structures like heightmaps. Most of the culling algorithms discussed in later sections are variations or special cases of BSP trees.

The process of determining visibility within the view frustum itself is called occlusion culling. It removes triangles that get overdrawn by triangles nearer to the view point. If the view point is next to a wall, performance could be greatly improved by not considering the polygons that lie beyond that wall. The benefits of occlusion culling strongly depend on how data is represented in memory as well as the data itself. Rendering flat terrain with occlusion culling enabled will not result in any performance gain — quite contrary,

the necessary processing overhead will result in a lower frame rate.

3.2.3 Level of Detail Rendering

Since the bottleneck on modern graphics hardware is not the processing speed but the available bandwidth of the data transfer to the rendering hardware, many algorithms have emerged which provide various techniques to reduce the amount of data that has to be transferred per frame. This techniques are known as Level of Detail rendering methods.

As landscapes tend to be huge, a vast amount of triangles has to be calculated and transferred to the hardware each rendering cycle. Portions of the landscapes may have a very high polygon count to ensure a fine level of detail, but far away from the view point thousands and thousands of polygons may be rendered to only a few pixels on screen (Watt & Policarpio 2001).

The idea of 'level of detail' rendering was first introduced by James H. Clark as early as 1976. The general idea is to reduce the level of detail at which objects are rendered at greater distances, based on calculation of a user specified error threshold. Some algorithms provide this feature by calculating different detailed versions of objects/portions of landscapes off-line and switching between that representations at run-time. The appropriate representation can be chosen according by a function of the distance from the object to the current view point.

However, this approach may result in annoying flickering and jumping on the screen as details pop on and off. King (2001) suggests to take the field of view into account when selecting the correct representation in order to avoid such undesirable artifacts. The author also points out that his algorithm addresses general level of detail issues and that his solution might not be suitable for terrain rendering.

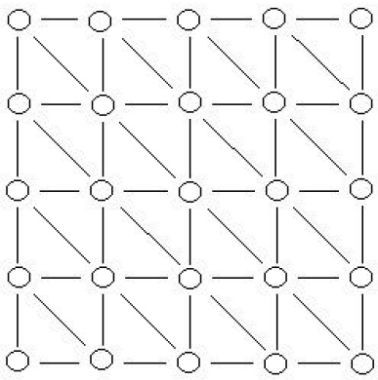

Figure 3.3: A top-level patch, taken from de Boer(2000)

Watt & Policarpo (2003) call the process of calculating different representations of an object off-line "building". They suggest to distinguish between structural faces and detail faces. Structural faces are big, flat surfaces that roughly describe an object, whereas detail faces are such polygons that add visual detail to an object. Structural faces can be used as splitting planes in BSP trees. The authors also discuss some landscape specialisations in level of detail rendering, which get addressed in the following sections.

The algorithms discussed in the following sections all try to simplify the rendered scene by leaving out surfaces that don't add visible detail. Thus, all algorithms can additionally be classified as "vertex removal algorithms", as discussed by Watt & Policarpo (2003).

3.2.4 GeoMipMapping

This approach, proposed by de Boer (2000), works very similar to the texture mip-mapping technique.

Compared to other algorithms that will get discussed in the following sections, terrain simplification using GeoMipMapping is easy to implement, though it will result in lower rendering frame rates than the more complex algorithms.

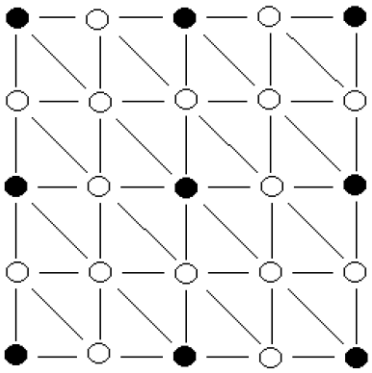

Figure 3.4: A coarse patch, taken from de Boer(2000)

Basically, the heightmap is subdivided into square patches as shown in figure 3.3. A dot represents a point in the heightmap, whereas lines indicate the actual triangles that are sent to the rendering hardware. The figure shows the finest level of detail representation, i.e. the detail level of the heightmap. The vertices that actually get rendered are determined at run-time by using a function of the distance to the view point as detail criteria.

As the viewing distance to the patch grows, the number of vertices used for that particular patch decreases. Figure 3.4 shows a coarser version of the patch. Black dots indicate vertices that are actually used for rendering, white ones are skipped.

The algorithm faces two problems:

1. cracks between patches of different level of detail have to be prevented
2. popping has to be avoided

Both of theses problems can be solved easily. Cracking can be prevented by simply skipping center edge vertices in the finer representation. Popping might occur when the detail level of a patch is changes. A possible solution to this issue is to hide the annoying effect by employing vertex morphing.

3.2.5 Interlocked Tiles

The Interlocked Tiles algorithm is an enhancement to the GeoMipMapping algorithm. It introduces reusable tiles to decrease memory consumption.

Snook (2001) identifies a significant drawback of level of detail algorithms, as "continuous triangulations disrupt speed advantage of hardware transform- and lighting, which relies on static geometry for optimum speed." The algorithm divides the terrain into smaller, reusable tiles. All tiles are square and of the same size and contain the same number of vertices distributed on the same regular grid. The tiles are internally represented as index buffers (DirectX) or element arrays (OpenGL), allowing efficient and fast hardware processing. The vertices to be rendered are chosen at real-time according to a function of distance to the view point. The L1-Norm can be used for this purpose. In order to assure continuous representations, neighbouring tiles have to interlock, i.e. high detail tiles have to link down to low detail tiles. This can easily be achieved by simply skipping center edge vertices.

A large landscape could be represented by only using a few tiles, so this technique is very memory efficient. However, heavy off-line processing is necessary to create patches of terrain that can be used to represent a large map without leaving the impression of repetition. The algorithm is not capable of dealing with dynamic height data, as it is not possible to recalculate the interlocking tiles at run-time.

This method of terrain rendering has been used in the popular game *Outcast* by Ubisoft.

3.2.6 Continuous Level of Detail Rendering

Continuous Level of Detail Rendering (CLOD) algorithms calculate a lower detailed version of an object/landscape patch at runtime, using a function of the object‘s distance to the view point as quality metric. In addition to this, as “the most common drawback of regular grid representations is that the polygonalization is seldom optimal, or even near optimal. Large, flat surfaces may require the same polygon density as small, rough areas do” (Lindstrom, Koller, Ribarsky & Hodges 1996), such algorithms should take local surface curvature into account. The closer the object is to the view point and the more curvature it has, the more details will be rendered for that particular object.

A lot of research has been done on reducing level of detail of polygon meshes (Hoppe 1996), however — since terrain rendering is based on a regular grid (heightmap) — algorithms for continuous level of detail reduction tend to be less complex than general 3D object approximation techniques.

3.2.7 QuadTree Algorithm

A QuadTree is a special cases of a BSP Tree (see 3.2.2). The idea is to sub divide the heightmap into square surfaces and save performance through exploiting characteristics of heightfield terrain representation, in particular the possibility of efficient frustum culling and level of detail rendering.

Ferraris (2001) describes how a QuadTree structure can be used to perform frustum culling. The algorithm presented is basically a Brute Force algorithm with fast frustum culling. No level of detail techniques are employed. However, the QuadTree data structure is suitable for much more sophisticated and powerful rendering mechanisms.

In this section the author discusses and compares two very influential papers which approach CLOD QuadTree algorithms in a different manner. The first is "Real-Time, Continuous Level of Detail Rendering of Height Fields" by Peter Lindstrom et al, proceedings of Siggraph '96. The term "QuadTree" algorithm isn't used in the white paper itself, as this publication represents one of the first terrain rendering algorithms of this type, which hadn't been given a name at the time of publication.

The authors present an algorithm that offers "large reduction in the number of polygons to be rendered" whilst maintaining "smooth, continuous changes between different surface levels of detail" through dynamically generated levels of detail (Lindstrom et al. 1996). The algorithm offers a simple image quality metric, defined as the edge length of a rendered quad measured in screen pixels.

The approximation — or as termed in the paper — simplification of height data is a two step process, consisting of block selection (coarse-grained refinement) and vertex selection (fine grained refinement).

Illustration 3.5 depicts how these two steps are carried out. The algorithm works in a bottom-up manner, calculating the edge length for each block. Blocks are merged and

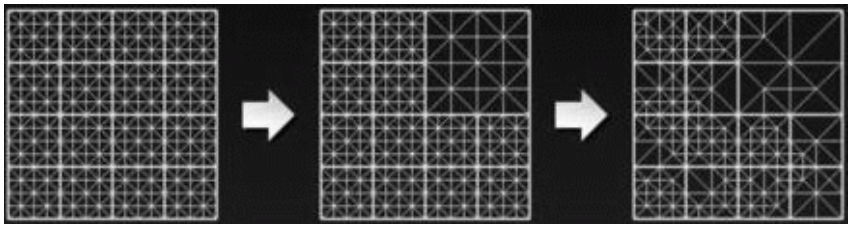

Figure 3.5: Two-step approximation, adapted from Lindstrom (1996)

replaced by their parent blocks if the result is less than the specified error metric (pixels on screen). In the second phase, specified vertices are selected for rendering in a manner that prevents undesirable cracks in the rendered terrain.

Though ensuring a high level of detail at a low number of vertices, the presented algorithm has some significant drawbacks. As the authors point out themselves, their solution is prone to popping, unless mesh morphing is implemented. The more significant demerit is that the algorithm works in a bottom-up manner, thus, every single vertex has to be examined in the refinement process.

3.2.8 Roettger et al

The second issue gets addressed by Röttger, Heidrich, Slusallek & Seidel (1998). The underlying data structure is again a QuadTree. Unlike the algorithm discussed previously, Röttger‘s solution uses a top to bottom approach. Thus, less vertices have to be examined in the approximation process, consuming less processing time.

$$\begin{pmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & 0 & ? & ? & ? & 1 & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & ? & ? & 1 & ? & ? & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & 1 & ? & ? & ? & 1 & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

Figure 3.6: A QuadTree matrix, taken from Röttger(1998)

In the process of refining the terrain, a corresponding QuadTree matrix is produced as shown in Illustration 3.6. The terrain is processed in a top-down manner, where a ‘1’ represents the center of a quad that has to be split into sub quads, and a ‘0’ represents a QuadTree that is sufficiently refined. Elements with a value of ‘?’ don‘t have to be visited during the refinement and rendering processes. Thus — as Röttger et al. (1998) points out — the heightmap size is of no influence to the amount of vertices that are necessary to render a scene, the memory bandwidth needed is directly defined by the desired image quality.

Illustration 3.7 depicts how the resulting QuadTree matrix is subsequently used to split the heightmap into triangle fans for rendering. The heightfield data is recursively evaluated, starting at the top-level quad and refining the four resulting sub nodes until certain quality conditions (that are discussed later) are met.

Cracks between adjacent tree nodes of different levels of detail can be avoided by simply skipping one center edge vertex in the higher detailed node, as long as the level of detail doesn‘t differ by more than one. This condition is automatically maintained by the surface roughness propagation during the preprocessing of the QuadTree, discussed later in this section.

Nodes are recursively split into sub-nodes while the condition $\frac{l}{d} < C$ is met, where l is the node's distance to the view port, d is the edge length of the node and C is the minimum desired resolution. This causes far away quads to be drawn at a coarser level of detail. Note that d directly corresponds to the current level of detail, 1 being the finest resolution.

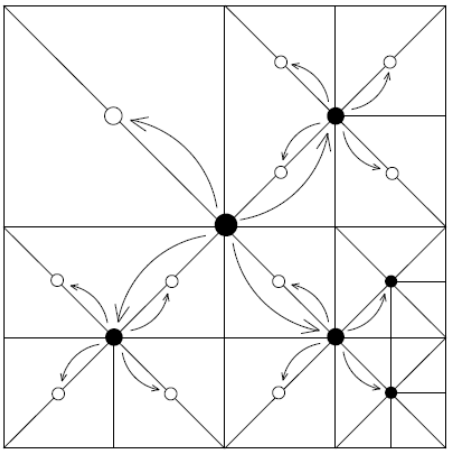

Figure 3.7: QuadTree tesselation, taken from Röttger(1998)

The paper then introduces surface roughness propagation to ensure that rough areas of terrain are rendered at higher level of detail than flat areas. A screen error d2 is calculated for each vertex, representing the difference between the approximated height of a pixel and the real height of that point. The value d2 is defined by the equation $d2 = \frac{l}{d} \max |\, {}^{\mathrm{dh_i}}_{i=1..6} \,|$, dh_1 .. dh_6 being the error at the center of each quad edge and its two diagonals as shown in Illustration 3.8. The calculated values d2 for the finest refinement level are then propagated to the parent nodes.

Based on this error calculation the paper presents an improved version of the original split criteria:

$$\frac{l}{d * C * \max(c * d2, 1)} < 1$$

The constant c specifies the desired local resolution and directly influences the amount of triangles drawn.

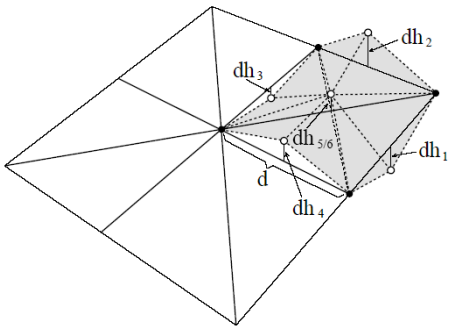

Figure 3.8: d2 error calculation, taken from Röttger(1998)

One final point has to be taken care of. To prevent cracks between quads, the level of detail of adjacent QuadTree nodes must not differ by more than one. This condition can easily by maintained during the pre calculation phase

by ensuring that two neighbouring quads satisfy the condition $\frac{1}{2} < \frac{l_1}{2*l_2} < \frac{C}{2(C-1)}$.

The QuadTree algorithm produces a list of quads that are subsequently sent to the rendering hardware. The most efficient way to do this is to make use of triangle fans. A triangle fan is a number of vertices arranged in a circle around a common center vertex. Thus, only $n + 1$ vertices have to be sent per triangle. This results in nine vertices per quad at maximum instead of twenty-four. If an adjacent quad is of lower detail, the center edge vertex on that edge is simply skipped to avoid cracks.

3.2.9 Real-Time Optimally Adapting Meshes (ROAM)

The ROAM algorithm by Duchaineau, Wolinsk, Sigeti, Millery, Aldrich & Mineev-Weinstein (1997) makes extensive use of triangulation.

As the algorithms discussed in the previous section, it creates a tree. However, the underlying data structure is not a QuadTree, instead, a completely new type of representation is introduced. The root of the tree are two triangles that comprise the whole heightmap — forming what the papers labels a diamond. The leaves contain triangles whose vertices are formed from adjacent vertices in the height field grid — the finest level of detail.

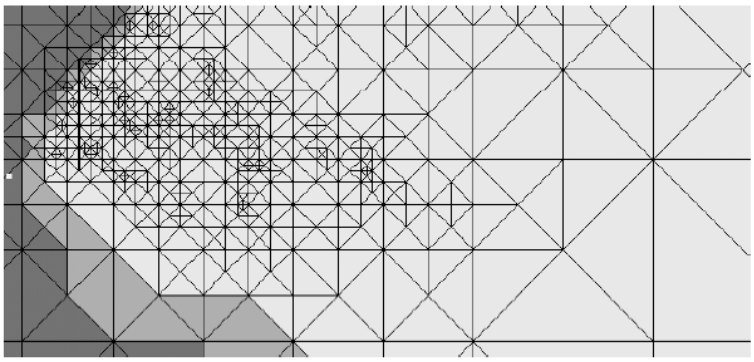

Figure 3.9: ROAM triangulated terrain, taken from Duchaineau et al (1997)

Each triangle in this structure has a base, a right and a left neighbour. These have following properties: Either the neighbours are from the same level, or from the next coarser level (base neighbours), or from the next finer level (left and right).

A split and merge operation defines a transition down or up a level, where splitting adds a vertex and merging removes one. If the base neighbour of a split triangle is from a coarser level, the split has to be done recursively to avoid cracks and gaps in the rendered scene. The papers call this a forced split. A crack occurs when T-junctions are created. Illustration 3.10 depicts how forced splits prevent cracks in terrain.

To ensure temporal continuity, Watt & Policarpio (2001) suggest to animate splits and merges by means of vertex morphing.

Rendering a scene using ROAM takes place in two phases, a pre-processing phase and an on-line processing phase. During the pre-processing phase, a set of view-dependent error bounds is constructed. This phase corresponds to the d2 error calculation of the QuadTree algorithm and works in a similar manner (see section 3.2.8).

The on-line phase significantly differs from previously discussed algorithms. Instead of completely triangulating the scene each frame, a split- and a merge-queue are maintained. These queues update the previous frame's bintree by split and merge operations, stopping when a geometric screen space error criterion is satisfied (view-dependent). In this way, the processing overhead can be greatly reduced for frames that don't differ much from the previous frame.

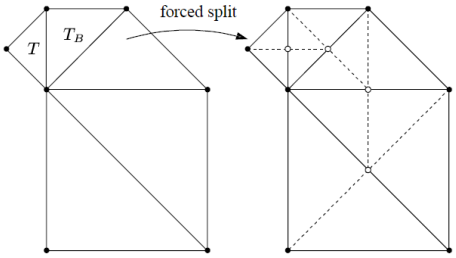

Figure 3.10: A forced split, taken from Duchaineau et al (1997)

It is possible to assign priorities to split and merge operations (high priorities are preferably assigned to operations that prevent screen space errors). This capacity makes ROAM highly scalable, as operations with low priority can be neglected when needed.

	Dynamic Data	Offline Calc.	Overhead	Mem Usage	Complexity
Voxel Graphics	yes	no	medium	low	medium
Brute Force	yes	no	low	low	low
GeoMipMapping	yes	no	medium	low	medium
Interlocked Tiles	no	yes	medium	high	medium
QuadTree	yes	no	medium	medium	high
ROAM	yes	no	high	medium	high

Figure 3.11: Algorithm Comparison

3.3 Conclusion

The algorithms discussed differ fundamentally from each other. Their applicability for a particular purpose is strongly dictated by the nature of the application. On machines with powerful rendering hardware and less powerful processors, the "CPU-friendly" Brute Force algorithm might be the best choice, as it features no processing overhead at all[3].

On the personal computer market, the situation used to be vice versa. Processors are powerful and the rendering hardware presents a bottleneck. In such an environment, the various level of detail rendering algorithm offer a higher frame rate through ellipsis of unnecessary, hardly visible detail. Yet, rapid advance in the graphics hardware market indicates that the situation might change in the near future. This might be the reason why nVidia encourages game developers to use the Brute Force algorithm.

A summarized comparison of the discussed algorithms can be found in figure 3.11.

The Interlocked Tiles algorithm requires a costly precalculation phase, in which the terrain data is examined for reusable tiles. Thus, it is not capable of handling heightmap data of dynamic nature (e.g. bomb detonations resulting in craters in military simulations, etc...). The GeoMipMap, QuadTree and ROAM algorithms are more suitable for such dynamic terrain data, although the latter two algorithms have to recalculate their error-bounds (see section 3.2.8) from scratch. Due to the nature of the used data struc-

[3]except the function call overhead, see section 3.2.1 for a discussion

tures, this can be achieved very efficiently as just portions of the landscape that actually have changed need to be considered.

The algorithm that copes best with dynamic data is the Brute Force method, as no pre-processing whatsoever is necessary. The last statement is only correct if no lighting model is employed. Otherwise, vertex normals have to be recalculated, of course. A very fast and efficient method to do this has been proposed by Shankel (2002).

For purposes which require terrain to be dynamic — as many simulations and games do — the QuadTree and ROAM algorithms are superior to their competitors. However, due to their complexity the required programming effort is tremendous in comparison to a Brute Force implementation. Military Institutions as well as big entertainment corporations can easily afford the necessary programming staff. Yet, if budget is a limiting factor, "easier" algorithms should be considered. The GeoMipMapping algorithm presents a good compromise between complexity and performance.

Chapter 4

Design

The application has been designed using the *Unified Modelling Language 1.5* (2003). Please note that the purpose of this application is to demonstrate engine implementation aspects in general without any specific reference to particular military or entertainment applications. Consequently, general terms ('User') are used instead of application specific terms (i.e. 'Pilot', etc...).

As stated before, a complete solution for an application engine comprises a large set of functionality (objects, sound, etc...) and also include tools for creating landscape data (map editors). The engine implemented as part of this project is a subset of such a complete application engine and is only concerned with the process of terrain rendering.

4.1 Use Case Diagram

The functionality as apparent to the user is illustrated in figure 4.1.

The system engine interacts with one sole actor — the user. That user could be a person playing a computer game as well as a jet pilot being trained for combat. It is possible for

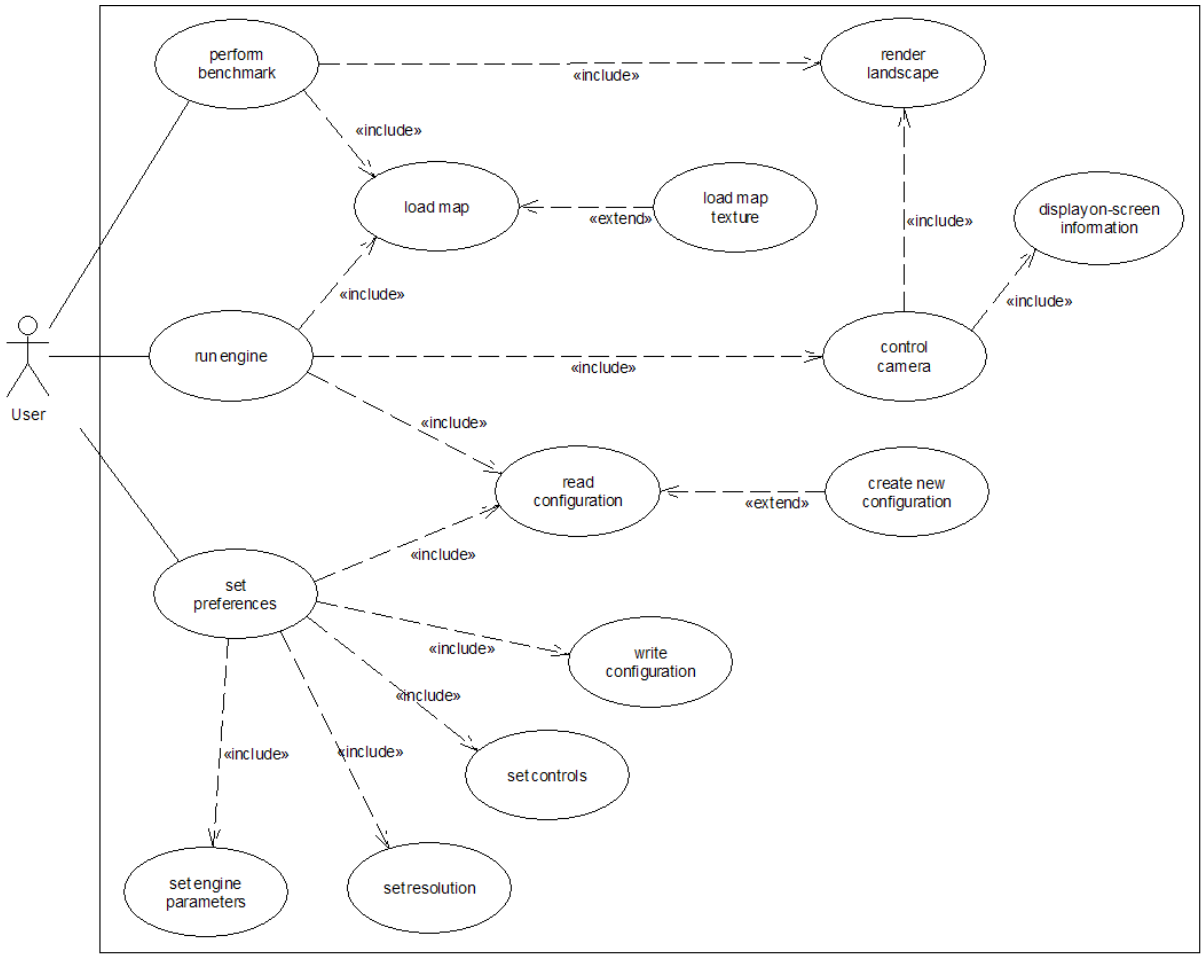

Figure 4.1: Use Case Diagram

the user to specify a set of preferences.

The user can select a desired resolution in which the rendering engine will run, and also whether the application will be rendering to a window or fullscreen. Additionally, a number of algorithm specific settings can be set.

The controls can be freely configured by the user. An individual keyboard mapping can be specified. This does not include the escape key, since it is used for terminating the application.

The bitmap that gets used as heightmap for rendering can be freely selected by the user, as well as an optional texture map.

When the application is started, the user is able to control the viewpoint. Since the application‘s use is not defined, the general term Camera is used for the user's viewpoint and no physical movement model is implemented. For entertainment/simulation purposes, etc... a physics engine would have to be implemented that handles user movement and positions the Camera according to its calculations.

It is possible to perform a benchmark. In this mode, the camera follows a predefined path and the time required is measured by the application.

4.2 Class Diagram

This section describes the classes that are part of the engine system.

The Design Pattern *Strategy* has been used to encapsulate the rendering algorithms. Thus, the actual rendering algorithm is easily interchangeable. This mechanism comes in useful for benchmarking different algorithms. However, due to the time restriction of the final year project, only the QuadTree algorithm is implemented.

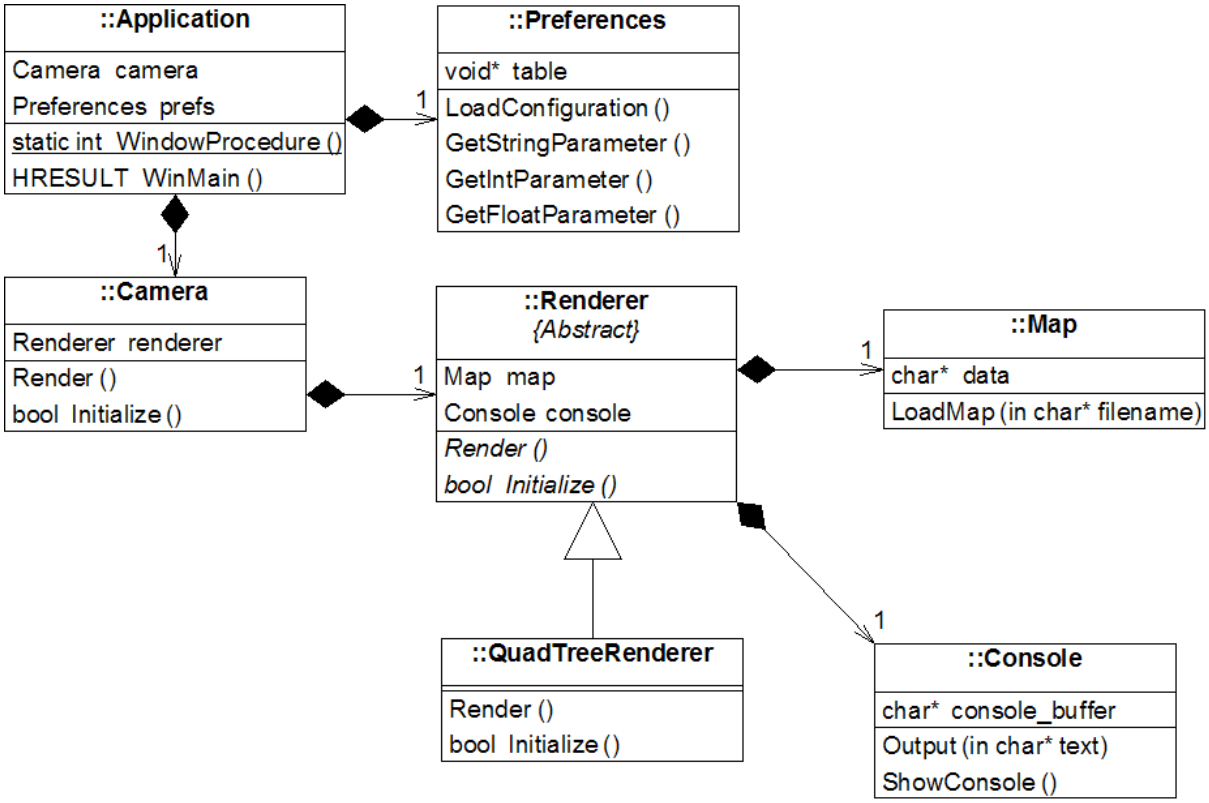

Figure 4.2: Class Case Diagram

A graphical representation of the class design is shown in 4.2.

4.2.1 Application

This class handles operating system related issues. It contains the WinMain() function, which is the startup point of a windows application and creates all other class instances. It comprises the whole operating system interaction, maintains the application message loop and also deals with user input. Contrary to the other classes, the Application class is highly platform dependent and has to be rewritten completely when porting to other platforms.

4.2.2 Renderer

This abstract class acts as interface to the actual rendering class. It defines a set of methods that each implemented rendering class has to support and also defines a way of behavior the rendering classes have to follow (concerning Initialization, Rendering, etc...). This makes it easy to change the actual rendering algorithm, a feature of great value to benchmark applications.

4.2.3 QuadTreeRenderer

This class encapsulates the actual implementation of the QuadTree algorithm by Röttger et al. (1998). The class is derived from *Renderer*.

4.2.4 Map

The class Map represents the heightmap data in memory. It loads the heightmap from a regular, gray scale bitmap file into memory. The heightmap has to be of size $2^n+1 * 2^n+1$.

4.2.5 Camera

The Camera class contains information about the users‘s view point, camera orientation and field of view. Its position and orientation are used as input for the Rendering class. The Camera class doesn't contain any methods concerning movement respecting physical laws. These would have to be implemented by a separate physics engine that sets the Camera‘s position and orientation based on its calculations.

4.2.6 Console

The Console class provides the user with on-screen information and feedback. The current frame rate is displayed in the upper left corner of the screen. The console outputs information messages to the screen and to a text file *log.txt* for reference.

4.3 Changes to the initial Design

Diagrams depicting the engine design as it was initially planned can be found in the appendix (figure A.1 and A.2). At the time the first design was carried out the rendering algorithm had not been decided on. The design was slightly modified in the implementation phase to meet algorithm-dependent requirements:

- a *map segmentation mechanism* was planned, which allows subdivision of large heightmaps into a number of *patches*. Such a mechanism enables algorithms whose performance depends on the heightmap size to render huge landscapes at fast frame rates. However, since the performance of the implemented algorithm is independent of the data set size (as discussed in section 3.2.8, the implementation was not necessary).

- a *camera track recording and playback mechanism* for benchmark purposes was planned as 'nice to have' feature. This could not be accomplished due to lack of time. However, a benchmark mechanism has been implemented, though the camera track is predefined in the engine code.

Chapter 5

Implementation

The prototype is written in C++ for the Microsoft Win32 platform. The non-proprietary API OpenGL is used as hardware abstraction layer. The motivation for not using Direct3D lies in the superior portability of OpenGL to other hard- and software platforms. The source code has been written with portability in mind, reducing Win32 API specific function calls to the bare minimum (DirectInput is used for input, since there is no platform independent method of input). Porting the engine to other platforms (Linux, etc...) is therefore easy, but unfortunately goes beyond the scope of this final year project. However, cross-platform benchmarks of the discussed algorithms are an interesting topic and might be picked upon in *Future Work* (section 6).

Illustration 5.1 shows a screenshot taken of the engine. More screenshots are included in Appendix C.

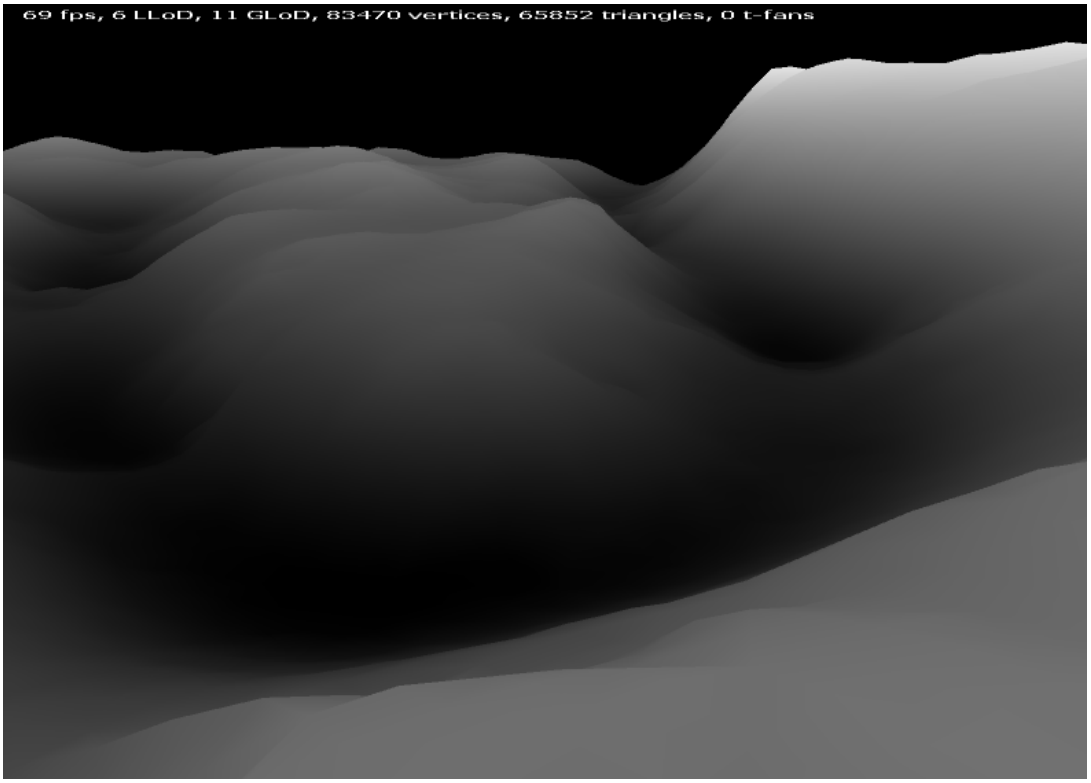

Figure 5.1: A screenshot of the engine

5.1 Details

Information about OpenGL programming was primarily obtained from Shreiner, Woo, Neider & Davis (2001).

The engine uses the QuadTree algorithm to render height field data and is further capable of performing frustum culling.

The pseudo code in figures 5.2 and 5.3 illustrates how the QuadTree algorithm has been implemented. First, the d2 values are calculated in a top-down manner, traversing down the QuadTree. The resulting d2 values are than examined for the crack prevention criteria discussed in section 3.2.8. Therefore, the level of detail of neighbouring quads cannot differ by more than 1. The resulting d2 values represent the error bounds for the current level of detail.

```
if bottom node reached then
  d2_value ← max(interpolation error at the 6 edge centers)
else
  sub_d2_values ← call recursion for sub nodes
  d2_value ← max(sub_d2_values)
end if
return d2_value
```

Figure 5.2: Recursive d2 error calculation

```
K ← globaldetail / (2 * (globaldetail - 1));
if not at bottom then
  d2_values ← recursively prevent cracks for sub nodes
  d2_value ← max(current_d2, neighbouring d2 values * K)
end if
return d2_value
```

Figure 5.3: Recursive crack prevention algorithm

The calculated d2 values are the basis for vertex selection in the main rendering loop. The following pseudo code shows how the actually rendered tree nodes are determined.

```
if inside view frustum then
  if bottom node reached then
    mark node for rendering
  else
    d ← edge length
    l ← distance node to view point
    error ← l / (d * C * max(1, c * d2))
    if error < 1 then
      recursively calculate sub node vertices
    else
      mark node for rendering
    end if
  end if
end if
```

Tree nodes that are marked for rendering after this procedure are then converted to triangle fans. The corresponding vertices are then sent to the rendering hardware, leaving out center edge vertices that would cause cracks with neighbouring quads of lower detail.

5.2 Optimisation

The engine has been coded with optimisation for performance in mind. However, due to the time limitations of the final year project, only a subset of the possible code optimisations could be carried out. See *Future Work* in section 6 for additional optimisation tasks which could not be carried out.

1. Division/Multiplications operations are avoided. Fast Bit shift operations are used instead of multiplications whenever possible.

2. Cast operations from float to integer are carried out using a faster code than the implementation created by the Microsoft Visual C++ compiler. See Herf (2000) for details.

3. Recursive function calls are avoided as far as possible. However, recursive function calls are necessary for traversing the QuadTree structure.

4. The engine is capable of using OpenGL extensions. It is possible to use (compiled) vertex buffers and the multi draw extension. Vertex buffers significantly speed up vertex submission, as discussed in section 3.2.1. See Wright & Sweet (2000) for a discussion or *OpenGL Extension Registry* (2004) for details.

5. The rendering algorithm produces triangle fans instead of specifying three vertices for every single triangle. Similar to triangle strips, triangle fans exploit the fact that neighbouring polygons share a certain number of vertices. In a triangle fan, adjacent triangles share a center vertex and the two outer vertices as well. In order to achieve maximum efficiency, triangles have to be arranged in a circle around their common center vertex. Using this way of representation the number of vertices necessary to describe n triangles is reduced to $n + 1$ for $n > 1$.

5.3 Configuration

The engine supports a large number of settings that are freely configurable. At startup the engine reads a configuration file *terrain.ini* which contains the parameters in plain text format. If no configuration file is present, a set of hard coded default values is loaded.

Please consult the user documentation in Appendix B for a description of the various settings.

A graphical frontend to this configuration has been implemented in order to provide a more convenient way of modifying the configuration file. It can be found on the enclosed CD-ROM.

Chapter 6

Evaluation and Future Work

The aims set in the project proposal have been achieved. The final report gives a comprehensive comparison of the most influental ideas in the field of terrain visualisation and a prototypical terrain rendering engine has been implemented.

During the work on the prototype the author gained a lot of experience in the range of computer graphics programming. Therefore, parts of code that caused problems in early development stages could be avoided if the possibility to do the project again existed. An example for this is the heightmap preview feature in the frontend application, which required more time to implement than expected. Documentation in the MSDN is fragmentary when it comes to scaling bitmaps using GDI functions. The time that was required to sort out these problems could have been used to implement some of the features discussed as future work in the following lines. Still, one of the project aims was to gain "First-Hand Experience in Engine Implementation" which has been achieved.

The author focused mainly on technical aspects of development. A technically solid and efficient engine is not sufficient for a commercial product. An application also needs to provide excellent artwork in order to deliver the best viewing experience. As the author‘s art skills are limited, the provided textures and heightmaps are of limited quality as well.

The project specification was set in a technical context, however, a "real" product would require a graphics design team that provides the necessary artwork in order to achieve commercial success.

Other rendering algorithms should be implemented as future work, since algorithms discussed in this report are only the most influential. Numerous other papers that are worth further investigation have been published. For example, it is possible to use triangular irregular networks as data structure for representation of a heightfield. Evans, Kirkpatrick & Townsend (1999) suggests to use isosceles triangles for the network (Right Triangular Irregular Network). This method of data storage offers numerous advantages and is very similar to the ROAM algorithm (see section 3.2.9). Lack of time prevented the author from investigating this source further.

Other rendering algorithms could be implemented. This is easily achievable since the Design Pattern *Strategy* has been employed. Rendering the same heightmap using different rendering algorithms gives a fair comparison in terms of speed and efficiency.

In order to achieve the best result possible, frequently called portions of code (e.g. the rendering loop) should be rewritten in assembler code. Furthermore, performance critical sections should make use of modern processor instruction set enhancements like "MMX", "ISSE", "ISSE2" and "3DNow!". Watt & Policarpio (2001) describe how to implement an efficient SIMD[1] mathmetical engine. This could be used as starting point.

The heightmap and texture data is loaded from windows bitmap files. The application could be enhanced to support more efficient file formats like JPEG, GIF and PNG.

[1]Single Instruction Multiple Data, part of Intel Streaming SIMD Extensions (ISSE)

Bibliography

Abner, W. (2000), 'Novalogic awarded patent for voxel space 2 engine'.
URL: *http://www.cdmag.com/articles/026/082/nova.html [accessed 5-February-2004]*

Arvo, J., ed. (1990), *Graphics Gems II*, Academic Press.

de Boer, W. H. (2000), Fast terrain rendering using geometrical mipmapping.
URL: *http://www.connectii.net/emersion*

Duchaineau, M., Wolinsk, M., Sigeti, D. E., Millery, M. C., Aldrich, C. & Mineev-Weinstein, M. B. (1997), Roaming terrain: Real-time optimally adapting meshes.
URL: *http://www.llnl.gov/graphics/ROAM/ [accessed 28-October-2003]*

Evans, W., Kirkpatrick, D. & Townsend, G. (1999), Right triangular irregular networks.

Ferraris, J. (2001), 'Quadtrees'.
URL: *http://www.gamedev.net/reference/programming/features/quadtrees/ [accessed 20-March-2004]*

Glassner, A., ed. (1990), *Graphics Gems*, Morgan Kauffman.

Herf, M. (2000), 'Know your fpu'.
URL: *http://www.stereopsis.com/FPU.html [accessed 19-January-2004]*

Hoppe, H. (1996), View-dependent refinement of progressive meshes.

Intel (2003), *IA-32 Intel Architecture Software Developer's Manual*, Intel.
URL: *http://www.intel.com/ [accessed 13-February-2004]*

King, Y. (2001), Never let 'em see you pop, *in* M. DeLoura, ed., 'Game Programming Gems', Game Programming Series, Charles River Media, chapter 4.9.

Lindstrom, P., Koller, D., Ribarsky, W. & Hodges, L. F. (1996), Real-time, continuouslevel of detail rendering of heightfields.

Maréchal, S. (2001), 'The second life of brute force terrain mapping'.
URL: *http://www.gamedev.net/reference/programming/features/bruteforce/ [accessed 20-March-2004]*

Marselas, H. (2001), Optimizing vertex submission for opengl, *in* M. DeLoura, ed., 'Game Programming Gems', Game Programming Series, Charles River Media, chapter 4.0.

Marshall, C. S. (2002), Triangle strip creation, optimizations and rendering, *in* T. Dante, ed., 'Game Programming Gems III', Game Programming Series, Charles River Media, chapter 4.5.

Microsoft (2001), *Microsoft Developer Network*, Microsoft.

nVidia (2003), 'Visualising the red planet with nvidia quadro graphics'.
URL: *http://www.nvidia.com/object/mars_rover.html [accessed 02-April-2004]*

OpenGL Extension Registry (2004).
URL: *http://oss.sgi.com/projects/ogl-sample/registry/ [accessed 20-March-2004]*

Picco, D. (2003), 'Frustum culling'.
URL: *http://www.flipcode.com/articles/article_frustumculling-pf.shtml [accessed 18-February-2004]*

Pollack, T. (2003), *Focus On 3D Terrain Programming*, Game Development Series, Premier Press.

Röttger, S., Heidrich, W., Slusallek, P. & Seidel, H.-P. (1998), Real-time generation of continuous levels of detail for height fields, *in* 'WSCG ' 98'.

Shankel, J. (2002), Fast heightfield normal calculation, *in* D. Treglia, ed., 'Game Programming Gems III', Game Programming Series, Charles River Media, chapter 4.2.

Shreiner, D., Woo, M., Neider, J. & Davis, T. (2001), 'Opengl programming guide'.
URL: *http://www.opengl.org/ [accessed 20-April-2004]*

Snook, G. (2001), Simplified terrain using interlocking tiles, *in* M. DeLoura, ed., 'Game Programming Gems II', Game Programming Series, Charles River Media, chapter 4.2.

Turner, B. (2000), 'Real-time dynamic level of detail terrain rendering with roam'.
URL: *http://www.gamasutra.com/features/20000403/turner_pfv.htm [accessed 20-March-2004]*

Unified Modelling Language 1.5 (2003).
URL: *http://www.omg.org/technology/documents/formal/uml.htm [accessed 17-April-2004]*

Watt, A. & Policarpio, F. (2001), *Real-time Rendering and Software Technology*, Vol. 1 of *3D GAMES*, Addison-Wesley.

Watt, A. & Policarpo, F. (2003), *Animation and Advanced Real-time Rendering*, Vol. 2 of *3D GAMES*, Pearson.

Wright, R. S. & Sweet, M. (2000), *OpenGL SuperBible*, Waite Group Press.

Algorithm white papers that have neither an URL nor are published in conference proceedings can be found as PDF in the folder *Papers/* on the accompanying CD-ROM.

Appendix A

Initial Design Diagrams

Figure A.1 and A.2 show the initial engine design as it was proposed in the design phase. The final design is discussed in section 4.

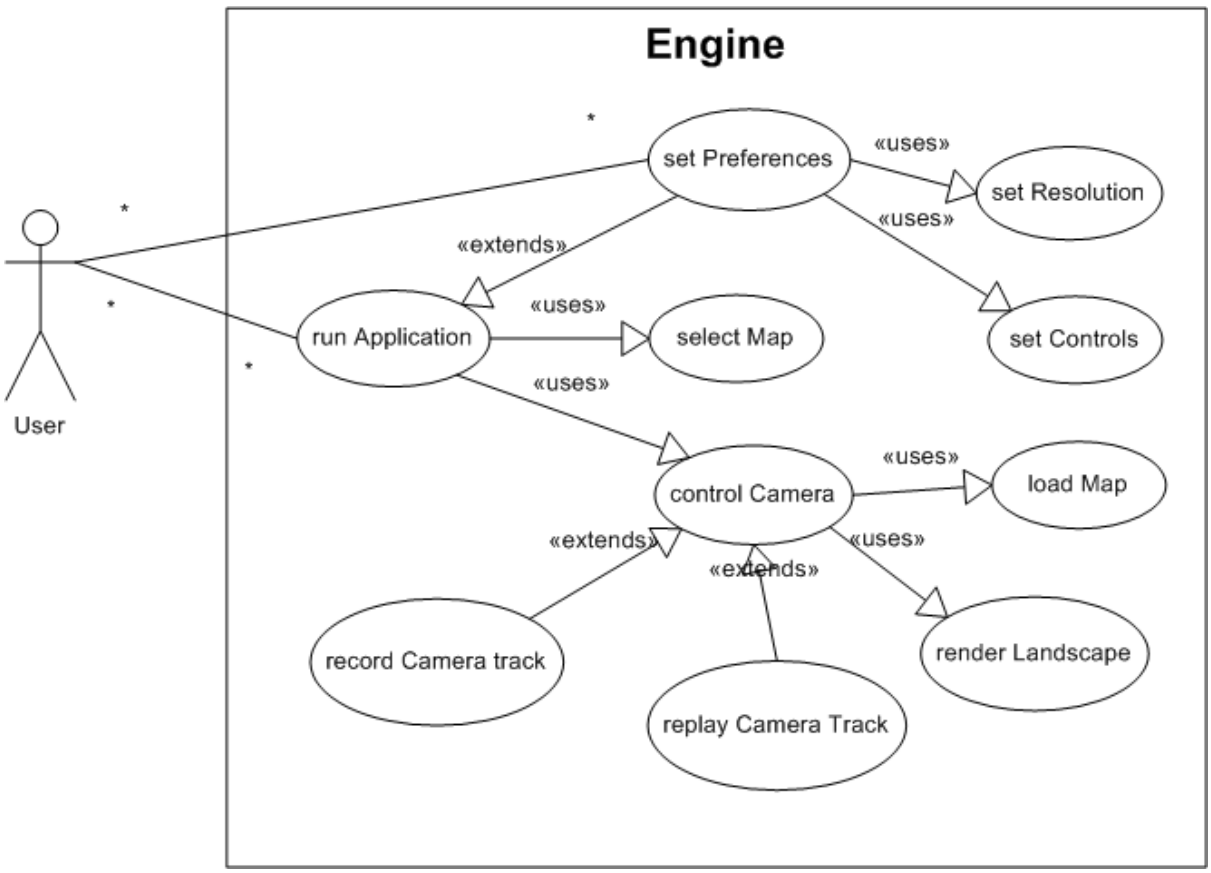

Figure A.1: Initial Use Case Diagram

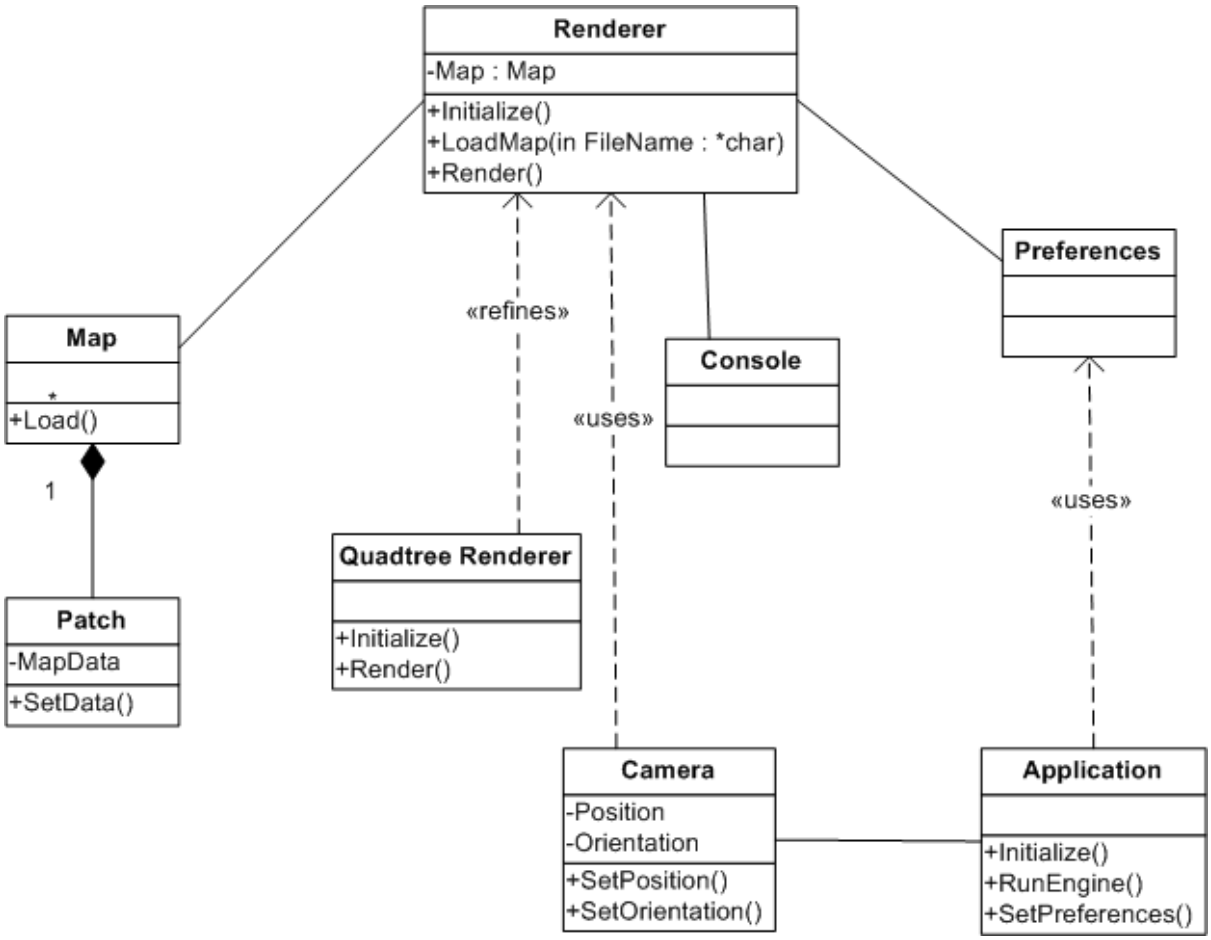

Figure A.2: Initial Class Diagram

Appendix B

User Documentation

This user documentation is also available as a help file in the executable package on the CD-ROM enclosed to this report. To access the documentation, start "Frontend.exe" and press F1.

B.1 Engine Configuration

All settings are stored in the file *terrain.ini*, located in the same directory as *TerrainEngine.exe* resides.

Frontend.exe is a graphical frontend to this configuration file. It has been written in order to provide a more convenient method of altering the engine configuration. However, the settings can easily be edited by hand if the frontend is not available.

The following sections describe the individual parameters based on the frontend application.

B.1.1 'General Settings' Tab

This property page contains all the general settings.

A heightmap can be specified either by typing in the filename or using the file chooser dialog. The picture in the middle shows a preview of the selected file. The heightmap has to be a gray scale bitmap file with dimensions $2^n + 1 * 2^n + 1$.

In *Benchmark Mode*, the user has no control over the camera. It follows a predefined path and terminates, displaying the time required. This is useful for comparing different configurations.

Perform Frustum Culling specifies whether to cull vertices that are not inside the viewing frustum. See section 3.2.2 for details.

Demonstrate Frustum Culling translates the point of view after culling and the model-view transformation have been applied. This feature can be used to illustrate how frustum culling works. See Illustration C.3 for an example.

Run in Fullscreen specifies that the engine should try to run in fullscreen mode. However, if the specified screen resolution can not be set, it will fall back to windowed mode.

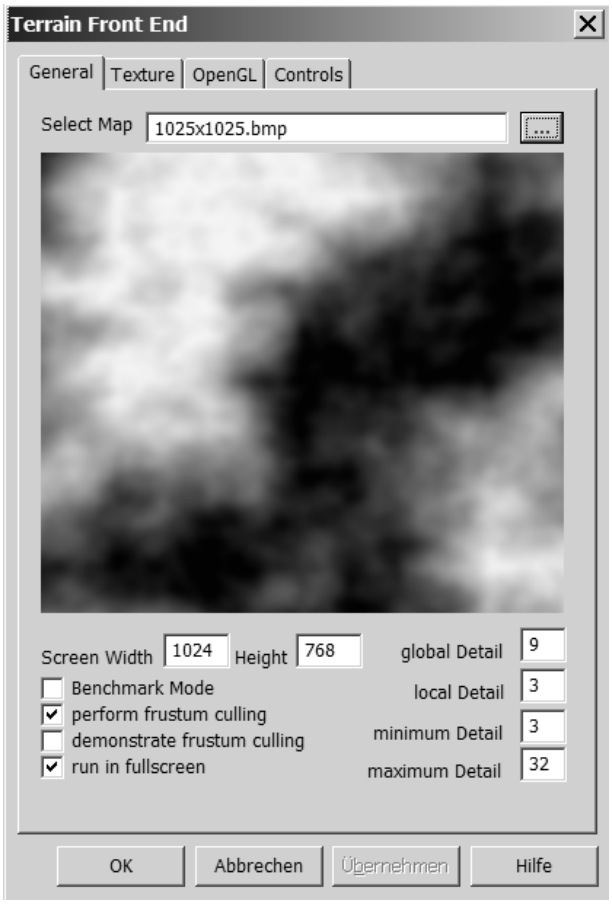

Figure B.1: The General Tab

Global Detail and *local Detail* specify the parameters for the QuadTree rendering algorithm. See 3.2.8 in the project report for a discussion. These two parameters can be altered at run-time using the specified keys. (Default keys are 'I', 'K', 'O' and 'L')

B.1.2 'Texture' Tab

All settings regarding texture mapping are located on this page.

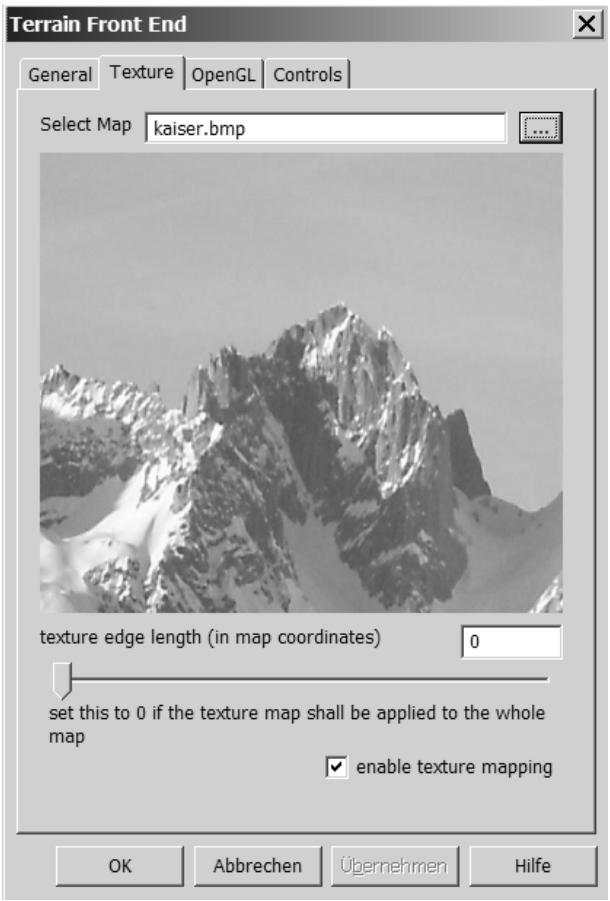

Figure B.2: The Texture Tab

Enable Texture Mapping specifies whether to use textures at all. If disabled, the scene is rendered with gouraud shading by default.

Texture Edge Length controls how the texture is scaled against the map. Setting this parameter to 1 means that the texture is mapped to exactly one square in the heightmap. Setting this parameter to the width of the heightmap means that the texture is applied to the whole heightmap. This behaviour can also be achieved by setting Texture Edge Length to zero.

B.1.3 'OpenGL' Tab

The settings on this page allow modification of OpenGL specific settings.

The *Field Of View* settings determine the projection matrix and view frustum. The projection matrix is specified in degrees in the y direction and an aspect ratio. Setting the aspect ratio to 1 means that the field of view in x direction is the same as in the y direction.

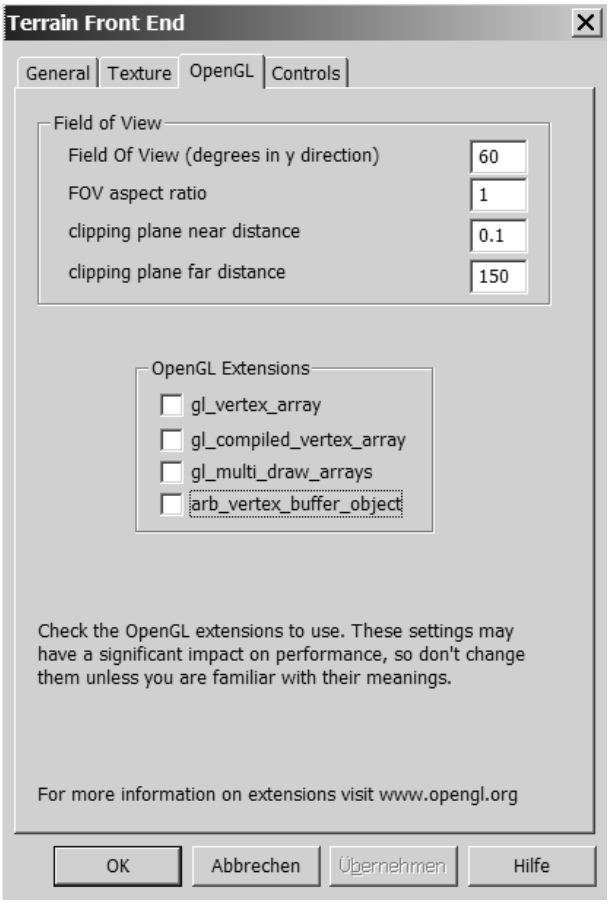

Figure B.3: The OpenGL Tab

The *Near* and *Far Clipping Plane* specify the clipping planes for the viewing frustum. The distance to the near clipping plane has to be greater than zero. The default values are 0.1 and 150.

It is possible to specify which *OpenGL Extensions* the rendering function should use. These settings have significant influence on the performance of the engine and should therefore only be modified by experienced users.

The gl_vertex_array extensions enables an application to store vertex data in buffers instead of issuing a separate function call for every single vertex. This extension significantly speeds up data transfer to the hardware, as vertex data can be optimized by the graphics driver. Using the gl_compiled_vertex_array extension enables even more vertex optimizing by pre compiling the vertex data. This can speed up rendering on some cards, but might cause problems or slow down rendering on some devices.

Using the gl_multidraw_arrays extensions multiple objects can be rendered with one single function call.

The arb_vertex_buffer_object allows clients to store vertex data in server-side memory. This can speed up rendering on some cards.

B.1.4 'Controls' Tab

On this property page the controls can be specified.

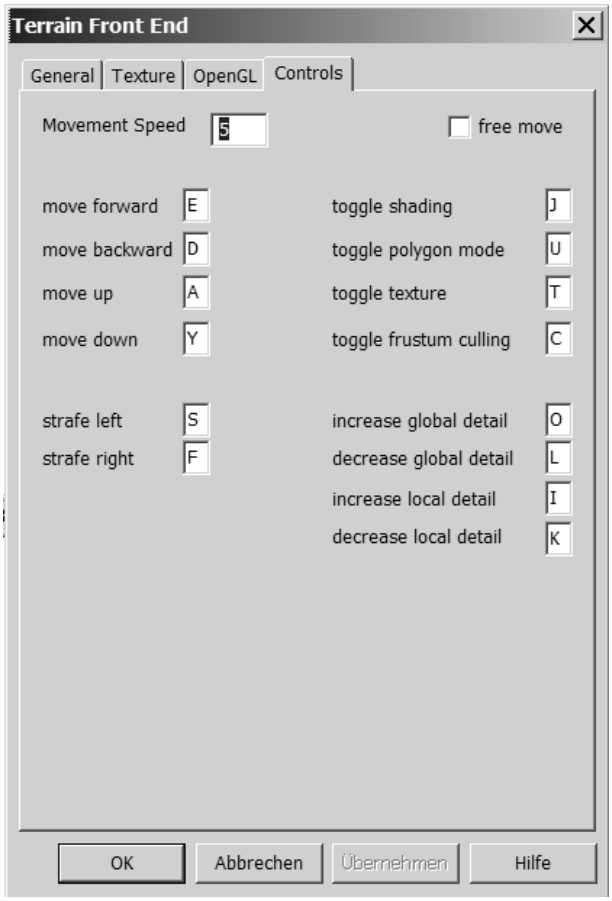

Figure B.4: The Controls Tab

Movement Speed specified how fast the camera moves when a key is pressed. The speed is measured in map units per second.

If *Free Move* is selected, the camera altitude can be freely controlled by the user. The corresponding keys are 'move up' and 'move down'. By default, the camera hovers two map units above the ground.

The keys *increase global detail* and *decrease global detail* recalculate the error bounds for the QuadTree algorithm with a new value for C. This step may take some time on slow machines, during which the engine freezes. Altering the local level of detail doesn't take any time, as there is no need to recalculate the error bounds.

The 'toggle' keys can be used to set different rendering modes. The shade model switch toggles between flat shading and smooth shading, sometimes referred to as 'Gouraud Shading'. The polygon mode switch specifies whether the scene is rendered using filled polygons or as a wire

Figure B.5: On-Screen Information

B.2 Starting the Engine

The engine executable file is named *TerrainEngine.exe*. On start, the program attempts to read the configuration from *terrain.ini*. A default configuration is used if the file doesn't exist. The engine then tries to load the heightmap and the texture map[1] from disk. If an error occurs, an information message is displayed and the engine terminates.

After a successful start the user receives control over the camera. The camera is controlled using the mouse and keyboard, as specified in the configuration file.

Press the escape key to terminate the application.

B.2.1 On Screen Information

During execution, information about the engine is displayed at the top of the screen.

The first number indicates the number of rendered frames per second. *LLoD* is the current local level of detail, *GLoD* the current global level of detail.

Triangles indicates the number of triangles sent to the rendering hardware for each frame. Similarly, *T-Fans* is the number of triangle fans.

[1] if texture mapping is enabled

Appendix C

Screenshots

This section contains screenshots taken of the prototype.

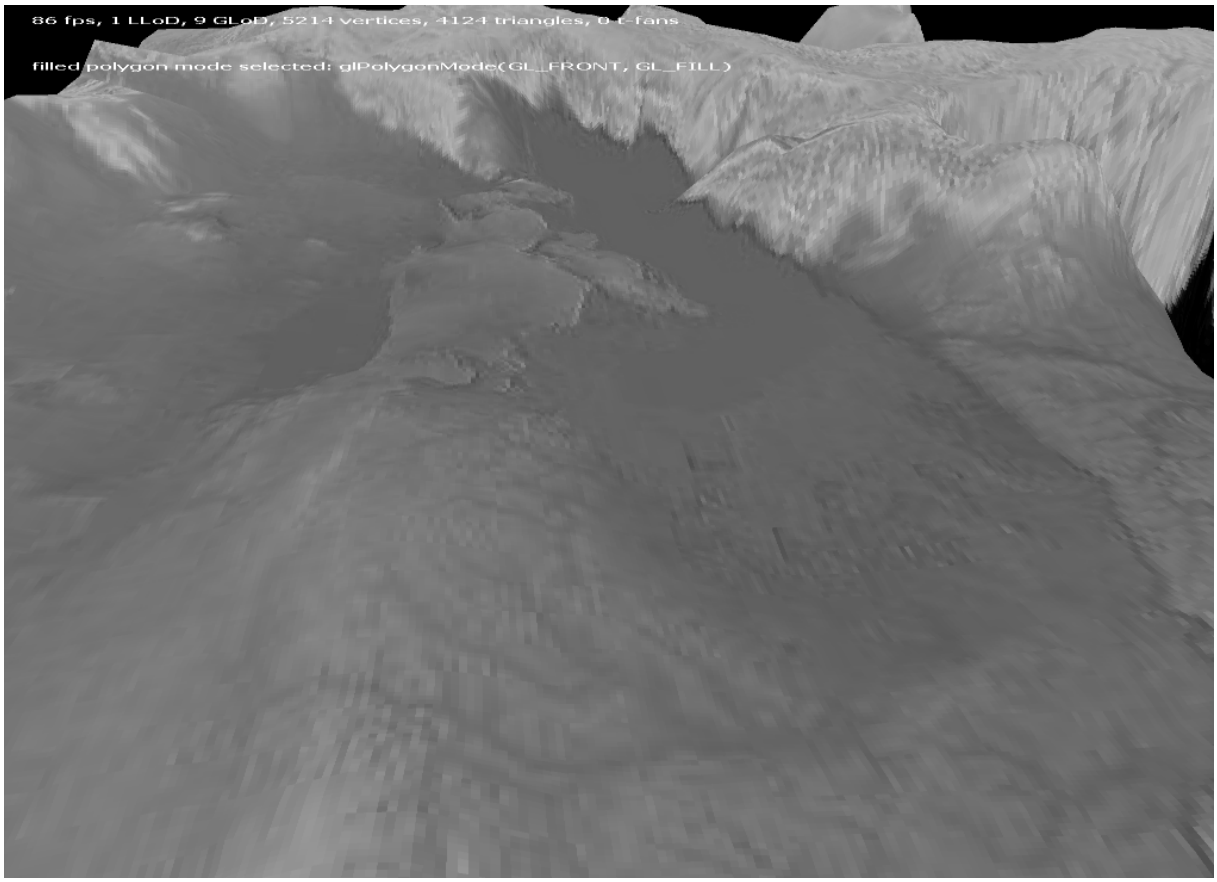

Figure C.1: A textured heightmap

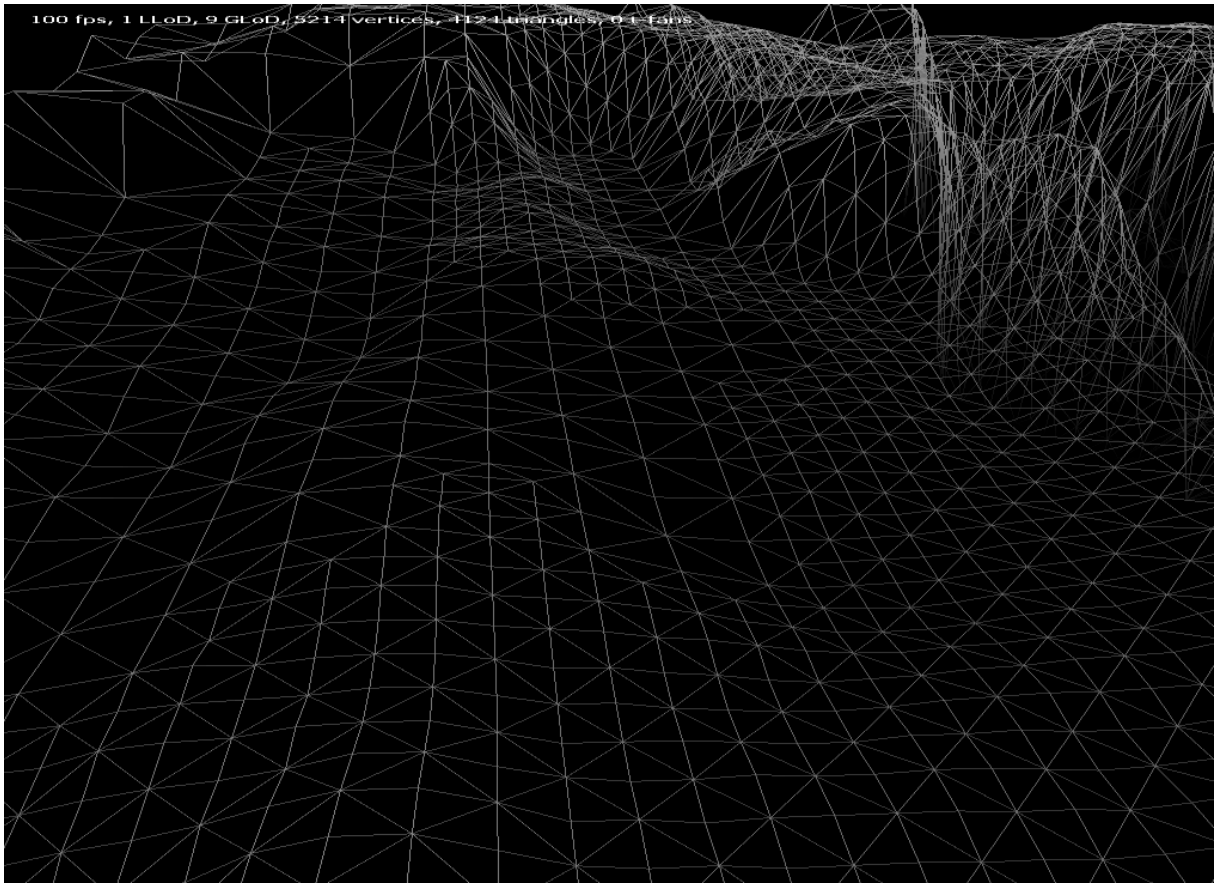

Figure C.2: The same scene as C.1 without texture mapping

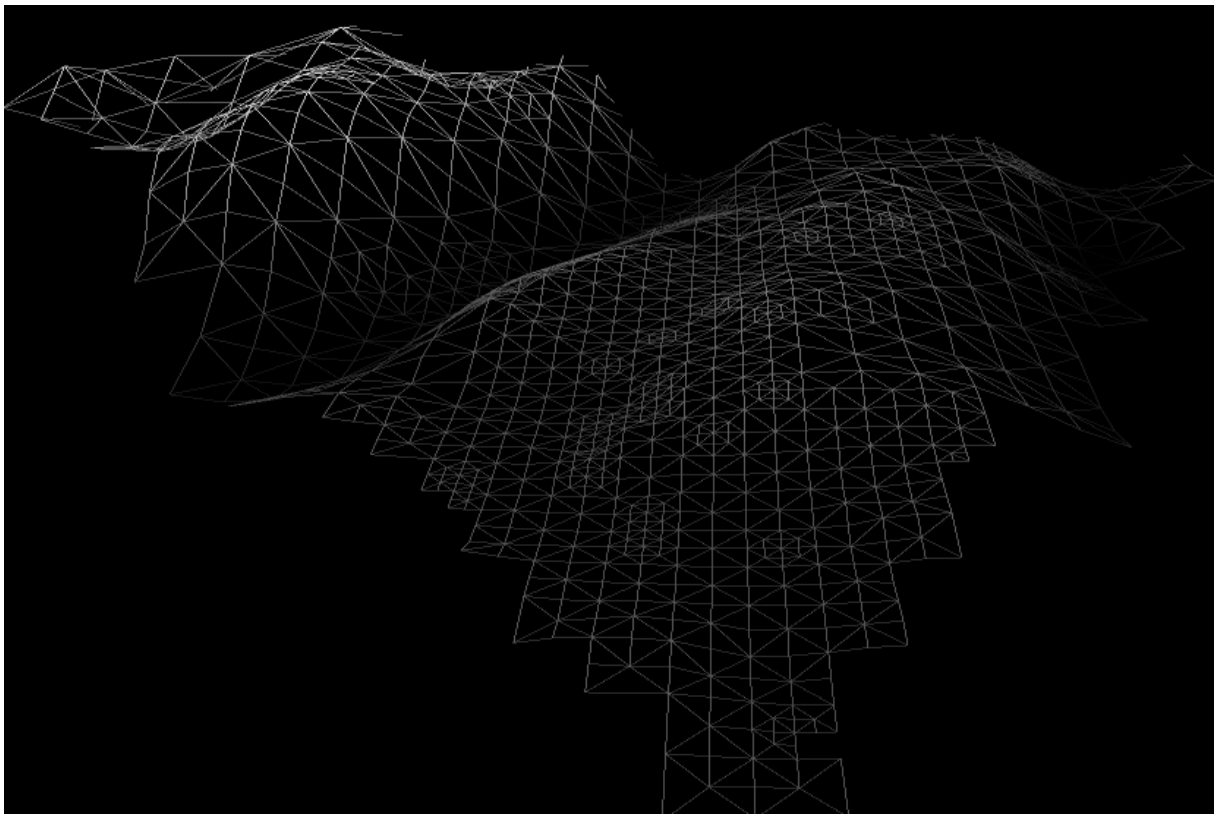

Figure C.3: Frustum culling demonstration

Appendix D

Project Proposal

The next page shows the final year project proposal.

School of Computing & Technology
University of Derby

Computing Scheme - Final Year Project Proposal

Name: ZLABINGER Gerhard
Project Title: Fast Real-Time Terrain Visualisation Algorithms

Main Supervisor:
Second Supervisor:

Does this project meet the BCS Accreditation requirements? Please include any constraints.

Aims and Objectives of Project:

- **Perform research on existing Algorithms and Approaches for rendering terrain in real-time**
- **Examine different algorithms for advantages/disadvantages**
- **Analyse Results for applicability to Simulation/Entertainment/Military Applications**
- **Implement a prototypical Terrain Rendering Engine**

Expected Outcomes or Deliverables:

- **Prototype of a Terrain Rendering Engine**
- **Project Report**

Methodology:

- **Literature Review of published papers on this topic (especially the ROAM-Algorithm by Mark Duchaineau/Lawrence Livermore National Laboratory and the work of Peter Lindstrom/Georgia Institute of Technology)**
- **Study Algorithms presented on various Internet Platforms (e.g. QuadTree-Representation of Terrain, Triangulation, ...)**

Hardware and Software Requirements (these MUST be available before the project starts):

Signature: **Date:**

Supervisor's Signature: **Date:**
(If you sign here, you are AGREEING to manage this student's project)

Appendix E

Progress Reports

The following pages contain the progress reports compiled for the meetings with the supervisor.

Progress Report, 24.10.2003

Tasks Completed

Identify the Internationally recognized Experts

- Mark Duchaineau, Lawrence Livermore National Laboratory
- Peter Lindstrom, Georgia Institute of Technology
- H. Hoppe, Microsoft Research
- S. Röttger

Find Sources on Web, Library...

Papers downloaded from the Web (ACM Digital Library)

- Stefan Röttger, Real-Time Generation of Continuous Levels of Detail for Height Fields
- Mark Duchaineau et al, ROAMing Terrain: Real-time Optimally Adapting Meshes
- Peter Lindstrom et al , Visualization of Large Terrains Made Easy, IEEE Visualization 2001
- PANKAJ K. AGARWAL, Efficient Algorithms for Geometric Optimization, ACM Computing Surveys, Vol. 30, No. 4, December 1998
- Randy K. Scoggins et al, Enabling Level-of-Detail Matching for Exterior Scene Synthesis
- Joshua Levenberg, Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry, IEEE Visualization 2002
- LEIIA DE FLORIANI et al, Hierarchical Triangulation for Multiresolution Surface Description, ACM Transactions on Graphics, Vol. 14, No. 4, October 1995
- Brandon Lloyd et al, Horizon Occlusion Culling for Real-time Rendering of Hierarchical Terrains, IEEE Visualization 2002
- Boris Rabinovich, Visualization of Large Terrains in Resource-Limited Computing Environments, IEEE Visualization 1997
- Renato Pajarola, QuadTIN: Quadtree based Triangulated Irregular Networks, IEEE Visualization 2002
- many more...

Books found in Learning Centre

- Watt A., Policarpo F. (2001). 3D GAMES, Real-time Rendering and Software Technology
- Watt A., Policarpo F. (2003). 3D GAMES, Animation and Advanced Real-time Rendering
- (1990). Graphics Gems
- (1991). Graphics Gems II
- (1992). Graphics Gems III
- (1994). Graphics Gems IV
- (2000). Game Programming Gems
- (2001). Game Programming Gems 2
- (2002). Game Programming Gems 3

Next Tasks:

Evaluate Sources

Problems Encountered:

none

Brief summary of project relevant content:

Watt A., Policarpo F. (2001). 3D GAMES, Real-time Rendering and Software Technology
Gives a overview about existing approaches to real-time terrain visualization, discusses ROAM and work of Peter Lindstrom in a very manner. Contains useful information about 3D application programming in general

Watt A., Policarpo F. (2003). 3D GAMES, Animation and Advanced Real-time Rendering

Contains useful information about 3D Engine programming in general.

(1990). Graphics Gems

(1991). Graphics Gems II

(1992). Graphics Gems III

(1994). Graphics Gems IV

(2000). Game Programming Gems

contains useful information on game/engine design

Yossarian King, 4.9 'Never let 'em see you pop': Discusses means of Level of Detail Rendering, not precisely in context of terrain rendering, but quite useful. Introduces a so called magnification factor to the Level of Detail selection, which is a relation between the size of an object on screen and the actual size of the object

(2001). Game Programming Gems 2
Greg Snook, Simplified Terrain Using Interlocking Tiles: presents an easy to implement alternative to complex level of detail algorithms, such as ROAM, ... uses relatively small patches (tiles) that are connected to each other using link elements

(2002). Game Programming Gems 3
Jason Shankel, Maxis, Fast Heightfield Normal Calculation: presents a method for fast computing face and vertex normals in height maps. This method could be very useful for dynamically changing terrain, since in that case the normals, which are needed for realistic lighting, would have to be recomputed at rendering time.

Progress Report, 7.11.2003

Completed Tasks

Found Additional Sources

- Polack, Trent (2003), *3D Terrain Programming, Focus On,* Premier Press, ISBN 1-59200-028-2

Current Tasks

Evaluate Sources

- started to read/evaluate the sources

Write Literature Review

- started to write literature revuew
- create bibliography
- create glossary

Next Tasks:

Evaluate Sources

Write Literature Review

Problems Encountered:

Postal Delay due to strike

Progress Report, 21.11.2003

Current Tasks

Evaluate Sources

- continued to read/evaluate the sources

Write Literature Review

- continued to write literature revuew
- continued bibliography
- continued glossary

Next Tasks:

Evaluate Sources

Write Literature Review

Write Interim Report

Create UML Design

Next Milestones:

Interim Report due 5th December

Problems Encountered:

none

Progress Report, 12.12.2003

Current Tasks

Evaluate Sources

- continued to read/evaluate the sources

Write Literature Review

- continued to write literature revuew
- continued bibliography
- continued glossary

Create UML Design

- Created Use Case Diagram
- Created Initial UML Class Design

Next Tasks:

Finish evaluating Sources

Finish Literature Review

Problems Encountered:

delay due to assignments in other modules

According to the project plan, I'm about two weeks behind. However, this should not result in a real problem since I've calculated enough buffer time at the end of the project and, moreover, the christmas holidays are a great opportunity to catch up with the original project plan in case of further delays in the future.

Progress Report, 13.02.2004

Progress

I've decided to implement the engine in OpenGL rather than DirectX, as using a non os-dependent hardware abstraction layers allows interesting performance comparison on different platforms (linux, etc...).

In the process of implementing the terrain rendering engine, I often discover new aspects of algorithms that are worth discussing in the literature review. That's mainly the reason the literature review is not finished, yet.

Current Tasks

Write Literature Review

- finalize literature review

Implement Prototyped Terrain Rendering Engine

- Created Application Framework
- Started implementing QuadTree algorithm

Next Tasks:

Finish Literature Review

Design

Implement Engine

Test Engine

Layout

Problems Encountered:

Problem: 64-Bit int bugs in GNU C++ compiler prevent measuring high resolution timer
Solution: switched to Microsoft Visual C++ .NET compiler.

Problem: Documentation of ATI vendor specific OpenGL extensions seems to be incomplete
Possible Solution: Use ARB extensions instead

Progress Report, 27.02.2004

Progress

The last two weeks I've been mainly concerned with implementation of the QuadTree algorithm as well as implementing ***culling***.

The prototyped engine now performs ***surface roughness propagation,*** the implementation of the roettgers quadtree algorithm is now complete.

Next tasks will be to apply texture mapping, allow larger maps by patching and performance tuning by removing math.h calls (sqrt, cos, sin, ...) using lookup tables.

Current Tasks

Write Literature Review

- finalize literature review

Implement Prototyped Terrain Rendering Engine

- Started implementing QuadTree algorithm
- optimize performance
- apply textures

Next Tasks:

Finish Literature Review

Design

Implement Engine

Test Engine

Layout

Problems Encountered:

none

Progress Report, 19.03.2004

Progress

The past weeks I've been optimizing the performance of the engine as well as fixing bugs. There engine now has a 'console', providing on-screen information to the user.

The most significant enhancement is that the map-size isn't static anymore, thus it doesn't have to be known at compile time. The surface map and all internal buffers are now completely dynamic, enabling the user to render any square height map with $width = 2^n + 1$ without re-compiling the engine.

Texture mapping has been applied (though the current texture is always a checked image, as I couldn't find appropriate freeware to create terrain textures that match a given heightmap).

A graphical user interface is almost finished, it allows the user to modify engine settings in a more convenient way than editing the configuration file with a text-editor.

Current Tasks

Write Literature Review

- finalize literature review

Implement Prototyped Terrain Rendering Engine

- finish GUI
- find & eliminate bugs

Next Tasks:

Test

Finish Literature Review

Problems Encountered:

Graphical User Interface: MFC CStatic control seems to either have bugs or be wrong documented. If no solution can be found, workaround is to draw image of heightmap manually using native Win32Api GDI functions.