Rendering Very Large, Very Detailed Terrains

Thomas Lauritsen Steen Lund Nielsen

26th April 2005

Abstract

Usually rendering very large, very detailed terrains has high storage and processing requirements, because huge amounts of data are involved.

In this document a new approach to solve the problem of rendering very larger, very detailed terrains is presented. Our proposal is to split the terrain representation into a low detail terrain, which does not required much storage and at runtime extend this low detail terrain with high amounts of details when needed.

Our algorithm is based on two existing level-of-detail algorithms presented by Ulrich in [20] and De Boer in [3]. We are using the chunked quadtree structure presented by Ulrich, but the simplification process has been replaced by the scheme presented by De Boer. Together these two algorithms constitutes a simple, efficient level-of-detail algorithm and is suitable for runtime addition of details.

The details can be calculated in various ways; we have chosen to use fractals for this purpose. Some techniques to speed up detail calculation is also shown.

Details are added to the terrain at runtime by extending the quadtree with new leaf nodes. The value of the height samples in the new nodes are generated as a combination of the calculated details and a subdivision of the existing height samples. The subdivision scheme used is the one presented by Kobbelt in [11].

Results gathered from our implementation of our method shows that storage and memory requirements are low. They also shows that our level-of-detail algorithm is performing very well and renders high quality images.

As such, the proposed solution works well; by adding details to an otherwise low detail terrain at runtime makes the terrain appear highly detailed, while storage requirements stays low. Given the high performance of the level-of-detail algorithm, our method is capable of rendering very large, very detailed terrains.

Preface

This document presents the results of the midway project conducted by the authors at the Danish Technical University at Informatics and Mathematical Modelling, during the spring semester of 2003. The project has been supervised by Niels Jørgen Christensen.

We would like to thank Niels Jørgen Christensen, Bent Dalgaard Larsen and Andreas Bærentzen for their help, ideas and good discussions about the problems we encountered during the course of this project.

Contents

Chapter 1

Introduction

Terrain rendering is the area of computer graphics that deals with the aspects of visualizing landscapes on computers. As a research field it have existed in many years and is still active today.

The great interest in terrain rendering may be because of the diversity of fields in which it is used. It is used in visualizations, for example by construction engineers or architects when constructing a visual presentation of their designs, or by special effects makers for making virtual environments for cinematic films. It is also used in the field of simulations, e.g. in the flight simulators used to train pilots or in the special area of simulations that is computer games. The rapidly increasing speed of the modern pc enables game developers to use techniques today which only a few years back was reserved for high-end graphics workstations, and among these is advanced terrain rendering methods.

1.1 The Terrain Rendering Problem

The prime problem of terrain rendering is simply: size! When walking in the outdoors one can often see vast areas of land and - very important - features at very different scales, from large mountains many kilometers away, to pebbles and grass at ones feet. When trying to visualize such landscapes on computers in realtime, two questions quickly presents themselves. First, how do we store all this information encoded in such a landscape? Besides larger features such as mountains and valleys there are millions and millions of small bumps and dents, which all are important to our perception of a landscape. Secondly, how do we render all this information in realtime?

Obviously, it is not possible to store everything. But even just storing everything of moderate size quickly builds up in terms of space. For instance, if we chose to measure just the height of a landscape for every 10 centimeters and did this for just one square kilometer of the terrain we would end up with 100 million height measurements! And 10 cm does not really capture much of the finer details of

the ground - and 1 km^2 of land does not make that large a landscape. Trying to capture a large landscape with a decent amount of detail certainly present a storage problem.

Besides storage considerations, the amount of information also presents rendering problems. Using a straight-forward brute force method of "just render everything" simply is not good enough. A simple triangulation of the aforementioned 100 million height measurements would yield 200 million triangles which still is well over the capabilities of modern hardware to render in realtime. And that is still just for 1 km^2 of the landscape.

The canonical solution to the rendering problem is to use level-of-detail rendering. This involves only rendering the parts of a model, in this case the terrain, that is necessary for the given viewpoint. What "necessary" means can vary, but usually it means that the deviation of the rendered image, compared to an image rendered with all details, stays within some defined limit. In practical terms and in the context of terrain rendering this means to render the ground around your feet with high detail while rendering the mountains far away with low detail. This makes sense as the smallest things perceptible by the human eye and, not the least, displayable on a computer screen grows with distance so finer details on far away mountains would not be visible anyway.

A lot of effort has been put into the level-of-detail part of the terrain rendering problem, but not much effort has been focused on the problem of storage. Some algorithms support out-of-core rendering allowing landscapes larger than system memory allows, which does help to some extend, but still enormous amounts of data has to be stored and processed when large, detailed landscapes are rendered.

So, the problem most terrain rendering algorithms are trying to solve is to decrease the amount of data to be rendered by selecting the most important parts of the landscape to be rendered. How the selection is performed differs between algorithms and has evolved alongside the progression of computer hardware. Early on, the actual drawing of polygons was a very expensive action and great care was taken only to select the polygons that carried the most information. Today, thanks to hardware accelerated 3d graphics, drawing a polygon is often faster than to determine if it should be drawn or not. Accordingly, the algorithms have changed from inspecting and selecting individual polygons to grouping polygons in larger groups and doing the selection between these groups instead. This way a larger part of the work can be carried out on the graphics accelerator.

Modern hardware and modern algorithms now have the ability to draw a huge amount of polygons in realtime, but as the polygon count increases so does the storage requirements and because memory and storage capacity of computers has not grown as fast as graphics hardware, storage is really becoming a bottleneck.

1.2 Our Proposal

In this document we will propose a solution to the terrain rendering problem consisting of a new level-of-detail algorithm and an alternative way to significantly reduce to storage requirements.

A key observation is that the precise shape and placement of finer details in a landscape are often not critical to the recognition of a landscape, whereas the shape and placements of larger features such as mountains and valleys are. However, the presence of finer details are significant to our perception of a real landscape.

Based on this observation we propose to split the representation of the terrain in two parts, one containing the larger features of a terrain and one containing the finer details. The larger features will be stored in a conventional manner, but as the shape and placement of the finer details are not critical we propose to generate these at runtime instead of having them stored. We will argue that this significantly reduces the storage requirements enabling larger and more detailed landscapes to be rendered.

1.3 Limitations

We will limit our data representation of landscapes to a regular, square array of evenly spaced height samples, commonly referred to as height fields or height maps. This means the input data essentially is two-dimensional, which makes terrain features such as overhangs or caves impossible. However, it also leads to many optimizations and simplifications and it is a common limitation of terrain rendering algorithms.

Further, we will only handle static data, which means we will not investigate methods for changing the terrain in realtime.

Finally, our target platform is commodity PC hardware and graphics accelerators. We expect realtime performance on newer configurations.

1.4 Structure of this Document

In chapter 2 we will give an overall presentation of the major existing level-of-detail algorithms and evaluate their usefulness in relation to our purpose. Based on this study, we have developed our solution to the terrain rendering problem, and in chapter 3 we will give a thorough description of this solution and in chapter 4 we will describe our implementation of our solution. In chapter 5 we will present results obtained with our implementation and these will, together with our solution in general, be discussed in chapter 6. Finally we will conclude in chapter 7.

Chapter 2

Previous Work

In this chapter we will present and evaluate the major existing level-of-detail algorithms. We will determine how well they fit our purpose, focusing on the algorithms applicability to addition of details to the terrain at runtime, on their performance on modern graphics accelerators and on the general complexity of the algorithms.

About Height Fields

Most terrain level-of-detail algorithms, and all of those presented in this chapter, uses a *height field* as source terrain data. As mentioned, a height field is a 2 dimensional array - or a grid - of height samples. All samples are evenly spaced and the dimensions usually needs to be a power of 2. Height fields are usually also required to be square.

2.1 Lindstrom 1

At SIGGRAPH'96 Lindstrom et al. published *Real-Time, Continues Level of Detail Rendering of Height Fields*[13].

The algorithm is a two step algorithm: The first step is a coarse grained simplification of the terrain and the second step is a fine grained simplification of the terrain.

Coarse Grained Simplification

The coarse grained simplification partitions the terrain into blocks. The vertex dimensions of the blocks, x_{dim} and y_{dim}, are of the form $2^n + 1$ where $n \geq 1$. The blocks overlap each other by exactly one row and column of vertices, such that the vertices along each block edge is shared with its neighboring blocks.

Simplification is done by grouping the blocks in larger blocks of 2×2 original blocks and then removing every other row and column in these larger blocks. The resulting larger blocks have an area equal to the area of their original four blocks but at a lower resolution, since approximately three quarters of the vertices have been removed. This block simplification is done recursively until there is only one block left. The last block then covers the entire terrain at a very low resolution. The blocks are inserted in a quadtree such that the largest, lowest resolution block is at the root and the smallest, highest resolution blocks are at the leaves.

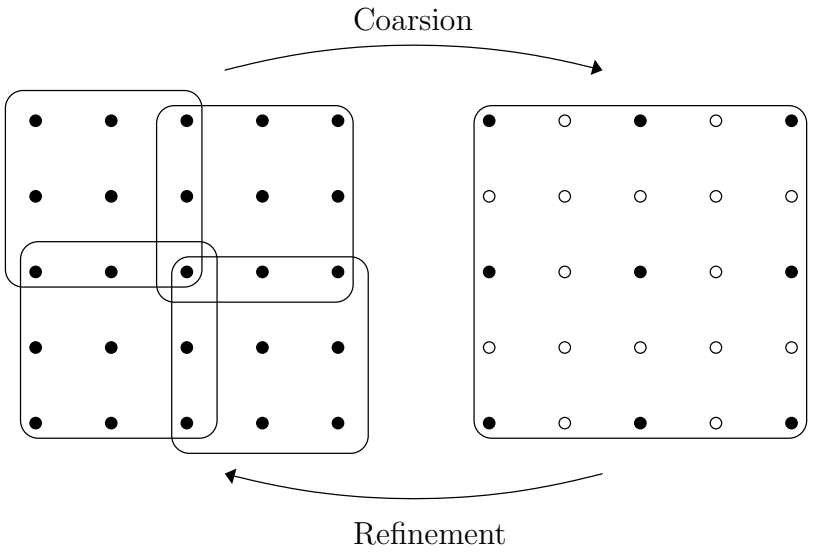

Figure 2.1: 4 blocks of dimensions 3×3 are grouped together. The white vertices are then removed to create a block of a lower resolution.

Fine Grained Simplification

The fine grained simplification is based on *longest edge bisection*. Longest edge bisection divides an isosceles right triangle into two smaller isosceles right triangles by inserting a new vertex on the middle of the hypotenuse and creating an edge from the apex to the new vertex. But as this algorithm is doing simplification the longest edge bisection process is reversed and the vertex at the hypotenuse is removed instead, thereby combining two triangles into one.

In figure 2.2 triangles $\triangle ADB$ and $\triangle BDC$ cannot be combined before the smaller triangles $\triangle AED$, $\triangle DEB$ and $\triangle BFD$, $\triangle DFC$ are combined. Because of cases like this, a hierarchy of vertices is build such that a vertex cannot be removed until all its children in the hierarchy are removed. Because fine grained simplification takes places within each block a vertex hierarchy has to be build for every block.

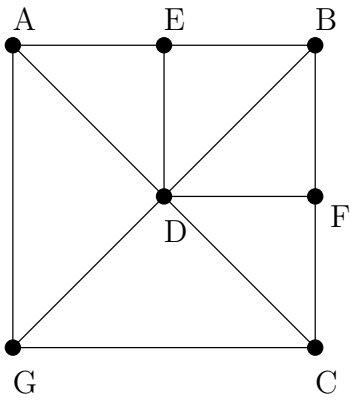

Figure 2.2: Longest edge bisection has been performed on triangles $\triangle ADB$ and $\triangle BDC$ to get the smaller triangles $\triangle AED$, $\triangle DEB$ and $\triangle BFD$, $\triangle DFC$

Level-of-Detail Selection

Whether or not a vertex should be removed from the hierarchy depends on the error introduced into the final rendered image of the terrain when removing this vertex. This error is measured in screen space and compared to a user specified error threshold. If the error is greater than the threshold the vertex should not be removed. The *world space* error δ, introduced by removing a vertex, is expressed as the vertical distance between the vertex and the hypotenuse of the new triangle, as depicted in figure 2.3, where vertex B is assigned a world space error δ_B.

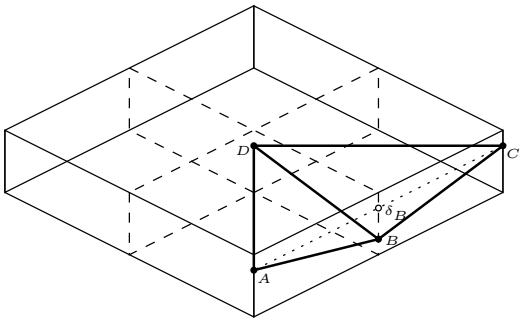

Figure 2.3: The relationship between vertex B and its error δ_B.

The *screen space* error is then determined by projecting δ onto screen space using the distance from the viewpoint to the vertex.

Level-of-detail selection of the coarse grained blocks is done by finding the maximum delta value δ_{max} of each block and using this to determine whether or not a lower resolution block can be selected. For each block δ_{max} is found as the maximum δ of the vertices at the lowest level in the vertex hierarchy of each block.

When level-of-detail selection has been performed, two neighboring blocks may not be at the same resolution and gaps might appear between them since they do not share all edge vertices, this is shown in figure 2.4, where each pair of vertices (A,a), (B,b) and (Cc) share the same position[1], but it is not guaranteed that the vertex d lays on the edge AB and neither is vertex e guaranteed to lay on the edge BC. Because of this gaps might appear in the terrain. Lindstrom shows how this artifact can be avoided by building vertex dependencies between blocks.

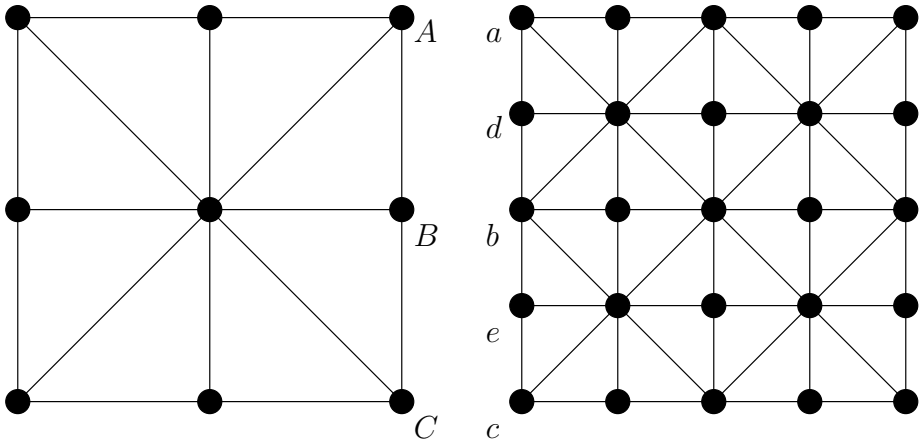

Figure 2.4: Two neighboring blocks at different resolutoin. Gaps might appear in the terrain at vertex d and e.

2.1.1 Evaluation

The coarse grained simplification step in this algorithm is a very simple, yet powerful idea. This would work well on modern graphics hardware.

However, the fine grained simplification step is CPU intensive and would limit utilization of graphics hardware. When the algorithm was developed hardware accelerated 3d graphics was uncommon and it made sense to spend more CPU work on optimizing the geometry to be drawn, but with modern graphics hardware this turns out to be a limitation instead.

The algorithm allows addition of more details at runtime in a simple manner. The nodes at the leaves of the block quadtree could be given four new children each covering an quarter of the terrain covered by their parent. Each new node would be given a vertex density approximately twice as high as their parent, by adding new vertices between each row and column of original vertices. A new vertex hierarchy and the maximum error has to be calculated, but it would be possible to do at runtime.

[1] Though they are drawn apart for clarity

To conclude, the algorithm carries some interesting ideas in the coarse simplification step and is adaptable to runtime detail addition but because of the fine grained simplification step, it is not usable in its current form.

2.2 Lindstrom 2

A more recent paper, *Visualization of Large Terrains Made Easy*[14], published by Lindstrom and Pascucci describes an algorithm that is more hardware friendly than the previous algorithm developed by Lindstrom [13]. The algorithm is also described in [15]. This algorithm is also based on longest edge bisection like the fine grained simplification step in [13], but this time it is used in a recursive top-down refinement of the terrain. The algorithm does not use the block based scheme of [13].

The Directed Acyclic Graph

The terrain data is stored in a *directed acyclic graph* (DAG), where each node has from zero to four children and from zero to two parents. A directed edge in the DAG represent an edge bisection from the apex of a triangle to the hypotenuse. Figure 2.5, shows the first few levels of the DAG.

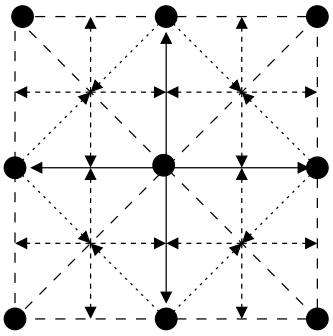

Figure 2.5: The first levels in the DAG. An arrow represents an edge bisection.

Each node in the DAG contains a vertex, a world space error value and a bounding sphere. The bounding sphere of each parent in the DAG is large enough to cover the bounding spheres of all its children. The same goes for the error value of each parent, it is as large or larger than the largest error metric among its children. Two levels of the bounding sphere hierarchy are illustrated in figure 2.6.

The DAG is built bottom-up, because the error values and the bounding spheres needs to be nested. The actual implementation of the DAG can be done in various ways and [15] shows a couple of methods.

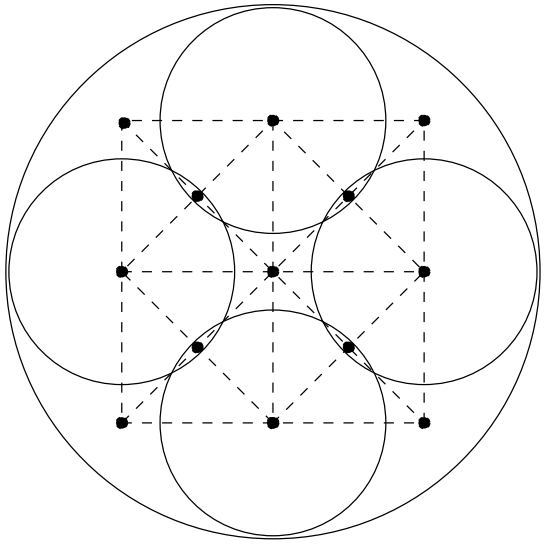

Figure 2.6: A bounding sphere hierarchy.

Level-of-Detail Selection

The detail selection works by traversing the DAG recursively from the vertex at the top of the bounding sphere hierarchy. The world space error of each vertex that is encountered is projected to screen space and compared to a specified error threshold. If the projected error is within the threshold the recursion stops at that vertex.

How the error metric is calculated and projected is not of great importance. A couple of different error metric and projections are suggested in [15].

An interesting property of this algorithm is that because of the DAG and the bounding sphere hierarchy, the terrain can be refined, triangulated and view frustum culled in one pass of the DAG. The triangulation is also interesting because it is possible to represent the entire terrain mesh at any given detail level as one triangle strip, that gets build while traversing the DAG.

It should be noted that this algorithm generates no gaps in the final mesh, so there is no need for special attention to this problem.

The traversal of the DAG is done whenever the viewpoint moves. If the viewpoint is stationary the same triangle strip will just be reused. If the viewpoint is moving the refinement has to be done each frame. If it is not possible to complete the refinement within one frame, the previous triangle strip can be reused until the new triangle strip is ready, but without any guarantee of the screen space error size.

2.2.1 Evaluation

The hardware friendly part of this algorithm lies in the triangle strip generation and possible reuse. This is correctly more hardware friendly than the first algorithm published by Lindstrom et. al. [13], but it is not really efficient on modern graphics hardware.

However, there are some interesting ideas in this algorithm. The fact that the refined terrain mesh can be represented as one triangle strip without cracks is interesting and the fact that the entire terrain data can be refined, triangulated (and stripified) and culled in one pass is also good.

Even though the DAG is build bottom-up, adding details to the terrain at run-time is probably doable, but it depends on how the DAG has been implemented. Basically all that is required is that more vertices are added to the DAG and more spheres added to the bounding sphere hierarchy. In order to not having to rebuild the bounding sphere hierarchy whenever new vertices are added, it is necessary to set the radius of the bounding spheres in the original leaves of the hierarchy to something "greater than zero", to allow for the error of the detail vertices to be added.

In conclusion, this algorithm is usable for our purposes, but as it is not really hardware friendly on modern hardware, it would most likely not perform acceptable.

2.3 ROAM

One of the most popular papers on terrain rendering is *ROAMing Terrain: Real-time Optimally Adapting Meshes* [5] by Duchaineau et al.

Like the two algorithms by Lindstrom, ROAM uses longest edge bisection. But where the algorithms by Lindstrom are based on vertex operations, ROAM is operating on triangles.

The Binary Triangle Tree

Duchaineau recognizes that a refinement of a terrain using longest edge bisection can be represented with a triangle binary tree (a *bintree*) because each split triangle results in two new triangles. Figure 2.7 shows the first four levels of a triangle bintree.

As seen in figure 2.8, triangle T has three neighbors. When triangle T and its *base neighbor* T_B are both from the same level in the bintree they form a diamond and T (and T_B) can be split.

If T does not form a diamond with its base neighbor it can not immediately be split. If T is required to be split then its base neighbor has to be split first, in order for T to form a diamond with its base neighbor. This forced splitting is

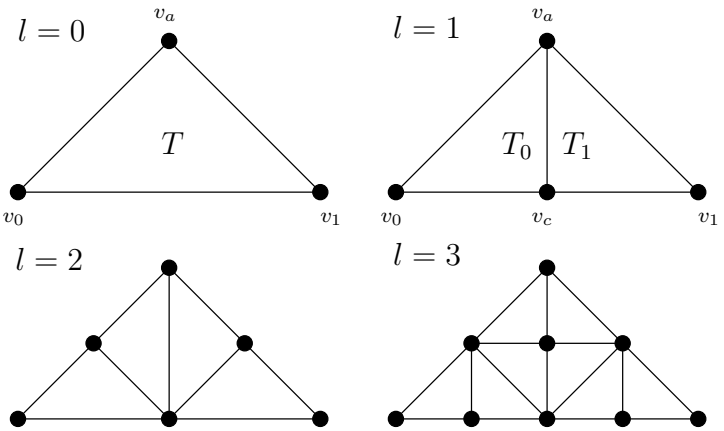

Figure 2.7: Levels $0 - 3$ of a triangle bintree.

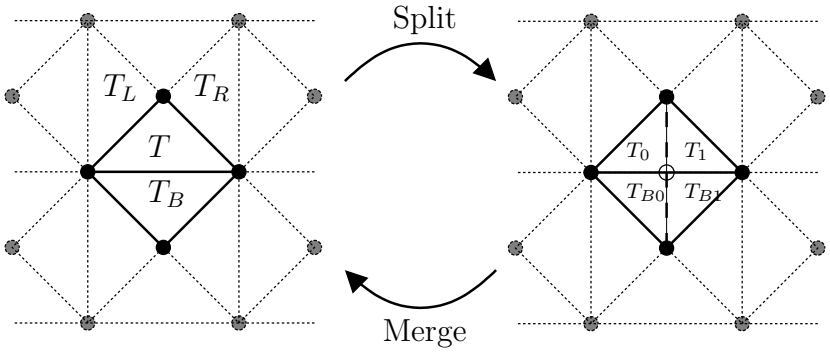

Figure 2.8: This illustrates the neighborhood relationship of triangle T. Also the split and merge operation is illustrated.

done recursively until a diamond is reached which can be split. This situation is illustrated in figure 2.9.

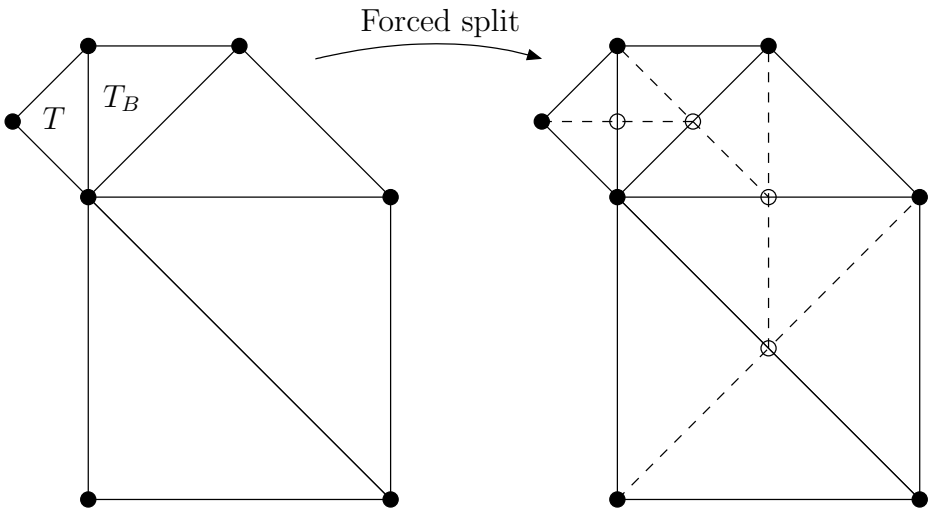

Figure 2.9: Before triangle T can be split, T_B is forced to be split and so does its ancestors.

Level-of-Detail Selection

Level-of-detail selection, or mesh refinement, is done by traversing the bintree in a top-down fashion, splitting diamonds as one descends down the tree. The recursion stops when the screen space error of the resulting mesh is below a specified threshold.

As this top-down refinement is time consuming, utilizing frame-to-frame coherency is suggested. This can be effective if the viewpoint changes slowly and smoothly. The idea is that the triangulation of the terrain does not change significantly from one frame to the next, so the same triangulation can be used in both frames with few modifications. Duchaineau suggests using two priority queues to keep track of which triangles can be merged and which can be split. Each triangle is inserted into one of the queues with an priority approximated by the the world space error projected into screen space.

The error metric is nested and uses a wedge shaped figure called a “wedgie”. A wedgie has basically the same shape as the triangle it surrounds but it also have a thickness. The thickness of these wedgies are nested because a parent has to have a greater error than its children otherwise the top down refinement does not work. In figure 2.10 a simplification of this nesting is shown.

As the error metric is nested the bintree is build bottom-up.

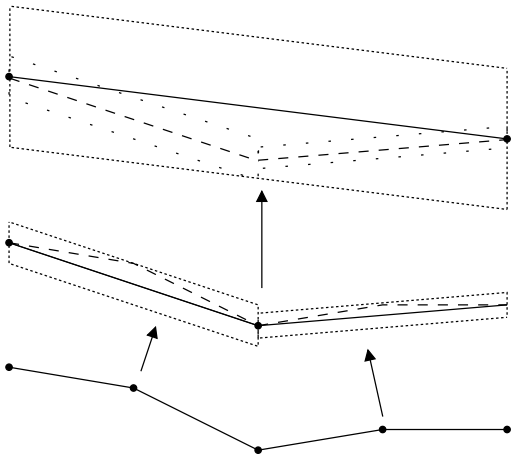

Figure 2.10: The nesting exemplified in 2D.

2.3.1 ROAM 2.0

There is still active development around the ROAM algorithm. A new, improved version, dubbed ROAM 2.0, is in development [4]. ROAM 2.0, among other things, aims at being very hardware friendly on modern hardware by caching and reusing larger parts of the meshes. Also paging and streaming of geometry as well as procedural generation of geometry is among new features. However, as no papers on ROAM 2.0 has been published yet, the algorithm is not available for us to use.

2.3.2 Evaluation

The top-down refinement process of ROAM is a simple and powerful concept, and has been widely used. But for optimal performance the queueing system is needed and this has proven hard to implement. Only by utilizing the queueing system some coherence can be kept in the resulting meshes, which is a requisite for good performance on modern hardware. But even with the queueing system in place some extra work is needed to provide good performance, like the improvements announced for ROAM 2.0.

Adding more vertices to the terrain at runtime can be done by attaching more nodes at the leaves of the bintree. As the error values are nested, they should either be recalculated when new triangles are added, but as this is slow, giving the original leaf nodes an appropriate artificial wedgie thickness would probably be the best solution.

To conclude, the original ROAM algorithm does allow runtime detail additions, but it is simply not likely to perform well enough on modern hardware for our use. The advances in ROAM 2.0 sounds promising, but is still unavailable.

2.4 View-Dependent Progressive Meshes

In *Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering* [10] H. Hoppe shows how a *view dependent progressive mesh* (VDPM) as described in [9] can be adapted to terrain rendering. VDPM is, as the name implies, a view dependent refinement process for a *progressive mesh* (PM).

To understand VDPM an understanding of ordinary - or *view independent* - progressive meshes is necessary. Progressive meshes was introduced by Hoppe in *Progressive Meshes* [8].

The Vertex Hierarchy

Basically any mesh M^n can be represented by a coarse base mesh M^0 and a sequence of refinement transformations known as *vertex splits*. Building a PM is done by simplifying an original mesh using successive *edge collapse* transformations and storing their inverses (i.e. vertex splits) in a vertex hierarchy. The edge collapse/vertex split relation ship is illustrated in figure 2.11 and a vertex hierarchy is shown in figure 2.12.

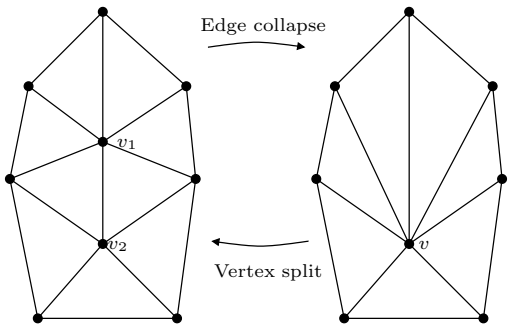

Figure 2.11: Edge v_1 - v_2 is collapsed into vertex v.

Selecting which vertices to collapse depends on an error associated with each edge. When the hierarchy is being built, the edge with the lowest error is chosen to be collapsed first, from the new mesh the next edge with the lowest error is selected for collapsing and so on, until a user specified error threshold is reached, then no more edges are collapsed. When all edge collapses have been performed the vertex split hierarchy can be built, by reversing the edge collapses. Another error is associated with each vertex in order to be able to decide when to do a vertex split.

View-Dependence

The progressive mesh scheme is made view dependent by performing of the vertex splits based on the position and direction of the viewpoint.

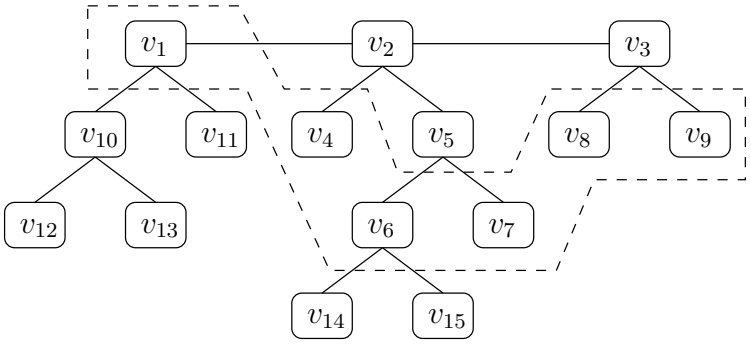

Figure 2.12: A vertex hierarchy. The base mesh M_0 is the vertices at the top of the hierarchy. The vertices enclosed by the dotted line is an example of an *active vertex front*.

Terrain Adaptation

The adaptation of VDPM to terrain rendering is done by simplifying some aspects of the VDPM scheme, as terrain data is essentially 2D, while general VDPM handles 3D data. This is mainly done to save storage. The terrain is also partitioned into blocks and the vertex hierarchy is build from these blocks. The blocks simplified, then stitched together 2×2 and simplified again and this process continues recursively until all blocks have been stitched together and simplified.

Avoiding gaps between blocks is done by not doing edge collapses on edges that are part of the blocks boundaries.

One important difference between the VDPM terrain scheme and most other terrain algorithms is that the source for a VDPM terrain need not be a height field but can be any mesh.

Level-of-Detail Selection

Once the vertex hierarchy has been built detail level selection is done by updating the *active vertex front*. The active vertex front is the vertices in the hierarchy, which is used in the mesh at the current level-of-detail. At first it would be the base mesh. The vertex front is updated by deciding, for each vertex, whether be split into two new vertices, merged with another vertex or left untouched, based upon an error associated with the vertex. And active vertex front is illustrated in the vertex hierarchy of figure 2.12.

The splitting and merging of vertices continues until the entire vertex front has an acceptable screen space error.

For each frame the vertex front from the previous frame is reevaluated, which means it utilizes temporal coherence.

Geometrical Morphing

In order to avoid visual artifacts from vertex splits Hoppe suggest doing geometrical morphing or *geomorphing*. Geomorphing is done by animating the vertex split slowly over time, so that the newly added vertices does not suddenly pop into existence.

2.4.1 Evaluation

The terrain adaptation of view dependent progressive meshes is interesting. VDPM's lesser restrictions on source data could allow terrain features such as caves and overhangs, which most other algorithm does not.

Unfortunately, the algorithm is rather complex and hard to implement. And like other older methods it is not very efficient on modern hardware, as it requires too much CPU work per polygon rendered.

The data structures of this algorithm is not suited for runtime addition of details. If more polygons should be added to the mesh, more vertices should be added at the leaves of the vertex hierarchy. But as the vertex hierarchy is build bottom-up and its particular structure is highly dependent on the information in the leaves of the hierarchy, new leaves cannot be added without recomputing the entire hierarchy. This is not possible in a real-time application.

The fact that runtime detail addition is not easily doable alone makes this algorithm unsuitable for our purposes. Efficiency and complexity is not in the algorithms favor either.

2.5 Geometrical Mipmapping

In 2000 De Boer published *Fast Terrain Rendering Using Geometrical MipMapping*[3], introducing a new level-of-detail algorithm targeted for modern hardware.

Geomipmaps

The algorithm works by partitions the terrain into a number of equal sized square blocks. Within each block the terrain is simplified at runtime in a simple manner much like texture mipmapping, hence the name *geometrical mipmapping* or *geomipmapping* for short. Each block is referred to as a *geomipmap*.

The simplification is simply done by removing each other row and column of vertices in a geomipmap for each detail level until the desired level is reached for the current geomipmap or no more vertices can be removed[2].

[2] All vertices but the four corners are removable.

Error Metric

Each geomipmap level has associated an error in object space. This error value is calculated as the maximum error resulting from the removal of vertices in the simplification step.

When a vertex i is removed an error value δ_i is calculated. It is calculated as the vertical distance from the vertex's position to the simplified mesh. As illustrated in figure 2.13 it can be seen as the vertical distance between the vertex position and the line connecting the vertex's two neighbors.

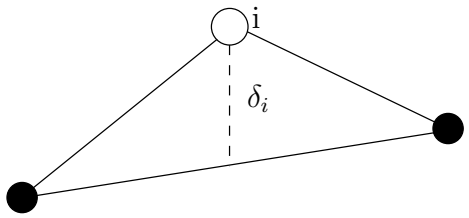

Figure 2.13: The vertex error is the vertical distance between the vertex position and the simplified mesh.

The error value for each geomipmap level is the then $max\{\delta_0, \ldots, \delta_{n-1}\}$, where n is the number of vertices removed in the geomipmap level.

Level-of-Detail Selection

Choosing the appropriate geomipmap levels is done by, for each geomipmap, projecting the object space error of the current level to screen space and comparing it with a user specified error threshold. If the projected screen space error is too large a higher detail geomipmap is chosen. If the projected screen space error is smaller than the threshold it is tested if a lower detail geomipmap level can be used, otherwise the current level is kept.

Avoiding Gaps

Special care has to be taken to avoid gaps in the terrain. Gaps can arise when two neighboring geomipmaps is used with different levels since they will not have same number of vertices at the edge they share.

The suggested solution is to do a special triangulation of the geomipmap with the lowest level, i.e. the geomipmap with the most vertices at the edge. The triangulation is done by skipping vertices at the edge such that there is the same number of vertices at the edges of both geomipmaps. This principle is illustrated in figure 2.14.

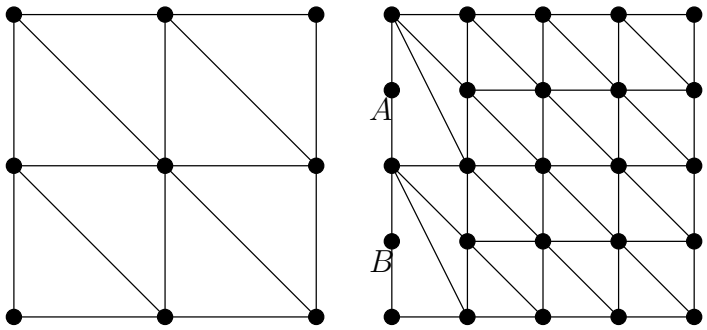

Figure 2.14: Two neighboring geomipmaps at different resolution. Triangulation of the higher resolution geomipmap skips everyother vertex at the edge, in this case vertex A and B.

Optimizations and Extensions

The error projection can be simplified by projecting the error as if the camera is only moving in the horizontal plane. This makes the calculations simpler. However, this only works well if the camera stays near the surface most of the time.

It is suggested that a quadtree could be used to efficiently cull geomipmaps that are not within the view frustum.

De Boer further extends the algorithm by showing how morphing can be used to hide the popping artifacts changes in geomipmap levels otherwise would produce. Morphing is done by displacing the vertices that differs between the two levels of detail that is morphed between.

He also shows how the algorithm can be used for out-of-core rendering of large terrains.

2.5.1 Similar algorithms

Larsen describes a similar algorithm in [12] but he presents another method for eliminating gaps in the terrain and shows that with modern graphics hardware morphing can be performed in hardware by the use of vertex programs.

2.5.2 Evaluation

This algorithm has two main advantages: it is relatively efficient on modern hardware and it is very simple.

It is efficient since detail levels are selected at block-level rather than triangle-level, which means lesser CPU work per drawn triangle. With geomorphs per-

formed in vertex programs even more work is offloaded from the CPU onto the graphics hardware.

But because of gaps the geomipmaps needs to be re-triangulated each time the detail levels are changed, which is not optimal.

The simplicity of this algorithm is partly because of the simplification scheme. While simple, however, it is not the most polygon-efficient simplification scheme and this algorithm thus may require a higher polygon count than other algorithms to achieve a certain error threshold. However, the much improved efficiency more than outweighs this, when compared to algorithms presented in the previous sections.

A more serious drawback of geomipmapping is its lack of scalability. As the terrain size[3] increases the number of geomipmaps grows, and with a high count of geomipmaps even at the lowest detail level, the polygon count may rise to unacceptable levels.

Runtime detail addition is almost trivial, which is the great strength of this algorithm from our point of view. The new detail data can just be added as more detailed levels at the bottom of the geomipmaps, some error values recomputed and then everything just works.

In conclusion, this algorithm does allow us to add details at runtime in a simple way and the algorithm provide acceptable, but not optimal, performance for terrains of limited size. To make it able to perform well with large terrain sizes some modifications needs to be done.

2.6 Chunked Level-of-Detail Control

At Siggraph 2002 Ulrich presented a new level-of-detail algorithm targeted for rendering very large terrains efficiently on modern graphics hardware. The algorithm is presented in *Rendering Massive Terrains using Chunked Level of Detail Control*[20]. The algorithm shares some ideas with geomipmapping, but differs on some significant points.

The Chunked Quadtree

The algorithm is based on a quadtree. Each node of the quadtree covers a part of the terrain in successive levels of detail. The root node has a low resolution mesh that covers the entire terrain. Each child of the root is the assigned a mesh that covers a quarter of the terrain, but at a higher resolution. This works recursively until the full resolution of the terrain data set have been reached.

[3] In this context size is not only determined by the area of the terrain, but also by the height sample density. E.g. lowering the density makes a larger area possible at the same cost.

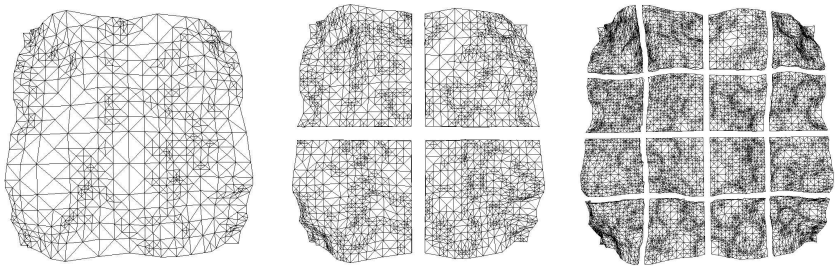

Figure 2.15: The first three levels of an example chunked quadtree. From left to right we have a parent node, its four children and sixteen grandchildren. *(Images courtesy of Thatcher Ulrich).*

In each node in the quadtree the nodes particular area of the terrain is stored in a mesh, which is named a "chunk". Each chunk is static and independent of all other chunks. An example of a chunked quadtree is illustrated in figure 2.15.

Error Metric

Each node also has a maximum geometric error, δ, of its chunk. As in geomipmapping the error metric is the maximum geometric deviation in object space from the original terrain data set. The chunks are constructed such that their error is halved for each level down the tree. For instance if a node have a δ of 16, its children should then have a δ of 8.

It is not stated how the meshes are simplified or how to obtain chunks with the correct δ-relations. All that is told is that it is rather complicated! A free reference implementation with source code is available, however, so it is possible to figure out, if necessary.

Level-of-Detail Selection

Choosing the appropriate level of detail is done recursively starting from the top of the quadtree.

As in geomipmapping it is assumed that the viewpoint only moves in the horizontal plane. Then for each node visited the chunks object space error δ is projected into a screen space error ρ. ρ is then compared to a user specified threshold τ. If ρ is less than τ the current nodes chunk can be renderer otherwise the children of the current node is examined. This is continued recursively. Pseudocode for this detail selection is illustrated below:

Avoiding Gaps

Because chunks are selected independently there are no guarantees that the edges of two neighboring chunks match up, and this could cause gaps to appear

```
render_lod(node)
    if rho(node, viewpoint) <= tau then
        draw(node.mesh)
    else
        for each child of node
        render_lod(child)
```

Listing 2.1: Detail level selection.

in the terrain. In order to avoid these gaps vertical *skirts* are used. These skirts, as seen in figure 2.16, are hung from the edge of all chunks and are just high enough to cover any holes that might appear because the edges do not match. This causes some inaccuracies in the terrain, but these are much less disturbing than gaps.

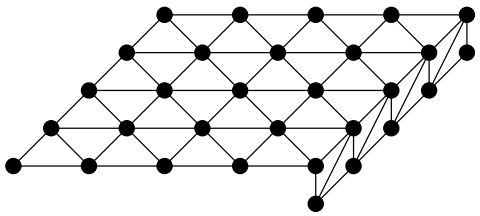

Figure 2.16: Skirts are hung from the edges (here only illustrated on the right edge).

Other Features

Morphing is utilized to hide the popping artifacts changes in detail levels otherwise would produce. The morphing is done in a way similar to how geomipmapping does morphing.

Because of the quadtree structure view frustum culling is extremely simple. As with detail selection this is done recursively, starting from the root of the tree. If a node is visible its children are tested for visibility, but is a node is invisible, all its children are known to be invisible as well.

Ulrich also describes simple out-of-core support. Since all chunks are independent of each other, only the chunks that are visible and at the required levels of details are needed in memory at one time. This allows for huge terrains as long as the maximum visible set fits in main memory.

2.6.1 Evaluation

This algorithm performs excellent on modern hardware. Since polygons are rendered in static chunks, only minimal CPU intervention is needed to draw a

large number of polygons. The static chunks can even reside in memory on the graphics hardware which reduces memory bottlenecks.

The simple out-of-core support is also an interesting feature which enables much larger terrains to be rendered.

The algorithm also scales very well to large terrains, as opposed to geomipmapping. This is due to the use of a quadtree, instead of using fixed size blocks.

Addition of extra detail data at runtime is very difficult, however. The quadtree structure could easily be extended with new detail nodes, but the simplification scheme used is very time consuming[4], and is not possible to perform at runtime.

To conclude, this is the best performing level-of-detail algorithm of the ones presented here, but unfortunately it is not usable for runtime detail addition in its current form.

2.7 Summary

In this chapter we have reviewed six of the most important and popular level-of-detail algorithms. These can clearly be categorized into two different categories: those that were developed with hardware accelerated 3d graphics in mind and those that where not. This is naturally connected to when the algorithms were conceived. The algorithms described in section 2.1, 2.2, 2.3 and 2.4 are the older algorithms targeted unaccelerated graphics. The algorithms in section 2.5 and 2.6 are the new hardware friendly algorithms.

Today, with modern graphics hardware the old algorithms are not very useful in their basic form. They simply do not perform acceptable, which often is caused by too much work being performed by the CPU rather than the GPU. Still the old algorithms carry interesting ideas which could be useful if updated with modern graphics hardware in mind.

The new algorithms of course performs much better on modern hardware, so our work should be based on these. Given our target, the chunked level-of-detail control is the best algorithm seen from a performance point of view, especially when considering support for large terrains. The simplification scheme of this algorithm is unfortunately not usable for our purposes, whereas the scheme used in geomipmapping is very usable.

A combination of the simplification scheme of geomipmapping with the general framework of chunked level-of-detail control, would give us an well performing algorithm, simpler than chunked level-of-detail control, more scalable than geomipmapping and amendable to runtime detail addition. This combination is in fact very similar to the coarse grained simplification step in the algorithm proposed by Lindstrom described in section 2.1!

[4] Judged by the time the free implementation spends on building the chunks.

Chapter 3

Our Method

This chapter will present a detailed description of the terrain rendering method developed during this project. In the first section we will provide an overview of the different parts involved in our method. In the following sections each part will then be described in detail.

3.1 Overview

As discussed in the introduction our technique for rendering very large, very detailed terrains involves augmenting a very large, but coarse height field with fine details at runtime. When adding these fine details to a height field the sample densities quickly increase to the point where a level-of-detail algorithm is a necessity. Our method thus requires a high performance level-of-detail algorithm which is able to render very large height fields and at the same time support addition of details at runtime.

As stated in chapter 2 the algorithm developed by Ulrich in [20] is the currently best performing level-of-detail algorithms. It is able to efficiently store and render very large height fields but it is not capable of adding height field data at runtime. Geomipmapping as presented by De Boer in [3] is also a well performing and very simple algorithm, which does allow runtime addition of height field data, but it does not scale well. These limitations makes both algorithms individually unsuitable for our purpose. Together, however, they have all the properties we need.

Our level-of-detail algorithm is thus a combination of these two algorithms. We combine the scalability and performance of [20] and the simplicity and the ability to add details at runtime of [3]. This combination gives us the sought high performance level-of-detail algorithm capable of rendering very large height fields and use runtime generated data.

On top of this level-of-detail algorithm a system has been developed which is able to generate and add fine details to the terrain at runtime in an memory-

and CPU-friendly manner.

For simplicity, we have imposed the following limitations on the height fields handled by our algorithm:

- The height fields must be square and axis aligned.
- The height fields must have the same amount of samples in each direction and they must be evenly spaced.
- The amount of samples in each direction of the height field must be in the form $n^2 + 1$, where n is an integer value.

These limitations could possibly be removed from our method, but are set for the sake of simplicity and are generally acceptable for most uses.

As the level-of-detail algorithm is the basis of our method we give a detailed presentation of this method first. The following sections will then describe our detail generation system and the combination of this system and our level-of-detail algorithm.

3.2 Level-of-Detail Algorithm

As written, the level-of-detail algorithm developed in this project is a combination of the two currently most popular level-of-detail algorithms, namely *chunked level-of-detail* [20] and *geomipmapping* [3]. Combining these two algorithms gives us an algorithm almost identical to the coarse grained part of the algorithm described by Lindstrom et. al. in [13].

Our algorithm is based on a quadtree structure, it is *chunked* and in structure very similar to [20]. Mesh simplification, however, is done in a mipmapping-manner as employed in [3]. This gives us the performance and scalability of citeTULRICH while keeping most of the simplicity of geomipmapping.

While the algorithm presented here was developed with our detail generation system in mind, it is still perfectly usable as a simple and effective stand alone level-of-detail algorithm.

The algorithm will be presented in full detail in the following sections.

3.2.1 The Quadtree Structure

The core of our level-of-detail algorithm is a quadtree. We define a quadtree as a tree where each node has no or exactly four children. Likewise, each node has exactly one parent except the node in the top of the tree. The node in the top of the tree is called the root and has no parent. Nodes with no children are called leaf nodes and all other nodes are called internal nodes. This is illustrated in figure 3.1.

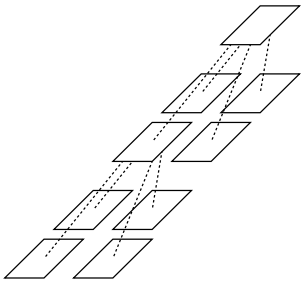

Figure 3.1: A quadtree structure.

The quadtree is build as a 2 dimensional space partitioning structure in a traditional way. This means that each node covers an axis aligned square part of the terrain surface. The root node covers the entire terrain surface, its children each covers one fourth of the terrain surface such that they together covers the entire terrain surface. In the general case each node covers one fourth of the area its parent covers and the four children of a parent together covers exactly the same area as the parent itself. See figure 3.2 for an illustration of how this works.

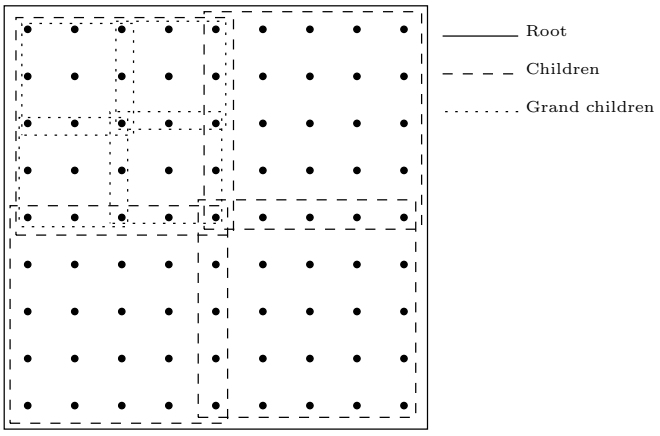

Figure 3.2: Space partitioning using a quadtree structure.

Having a quadtree as a space partitioning structure is in itself not enough for level-of-detail rendering. To use a quadtree in this context, each node in the tree is augmented with a mesh describing the terrain it covers at a certain degree - or *level* - of detail. The root node thus contains a mesh describing the whole terrain, but at a very low level of detail. Its four children contains a mesh describing their fourth of the terrain at a higher level of detail. This continues until the leaf nodes where each node contains a mesh describing their part of the terrain at the highest level of detail.

3.2.2 The Chunked Quadtree

The contents of the meshes in the nodes of the quadtree is a critical parameter of how effective the level-of-detail-rendering will be. The simplest possible mesh is one quadrilateral (or two triangles) covering the area of the node. While this approach gives a very fine control over detail selection, it results in too much CPU work, little GPU[1] utilization and poor polygon throughput.

The reasons for these poor results are due to the way the detail levels are selected and rendered as explained in section 3.2.4. In short the problem is that the CPU must actively select each quadrilateral and either render them individually or better batch them for later rendering, but neither leads to optimal performance.

To maximize polygon throughput many polygons must be drawn with minimal cpu intervention. A simple and effective way to do this is to have the meshes in each quadtree node consist of many polygons, up to several thousands for best utilization of graphics hardware. The detail level restriction of the tree still applies; the root node stores the lowest detail version of the entire terrain and the leaf nodes store the highest detail version of their respective terrain parts. The meshes in the quadtree are called chunks and from that follows the name chunked quadtree. In figure 2.15 a simple illustration of a chunked quadtree is shown.

As each chunk is a complete representation of the terrain it covers (albeit at a given level-of-detail) it is completely independent of other chunks and even the tree itself. This independence is a good property as it simplifies implementation as well as allows for very simple out-of-core support, as described in section 3.2.14.

3.2.3 Mesh Simplification

The chunks in the quadtree are generated through mesh simplification in a bottom up process. In the leaves at the bottom of the tree, the meshes are generated directly from the source height field. The leaves are then gathered in blocks of 2×2, combined and simplified and the resulting meshes are assigned to the respective parents of the leaf nodes. This process continues until all meshes have been combined and simplified into a single mesh at the root node.

The simplification scheme is similar to *geometrical mipmapping*: for each simplification step every other row and column of the mesh are removed. In figure 3.3 the simplification scheme is illustrated.

Obviously this simplification scheme is not optimal. It is easy to contrive examples where it will generate very poor results, but in the general case, at least in domain of terrain meshes, it will produce acceptable results.

The major reason to use this simplification scheme is that it is fast and simple. This is very beneficial when combining the level-of-detail algorithm with the detail generation system as explained in section 3.4.

[1] Graphical Processing Unit

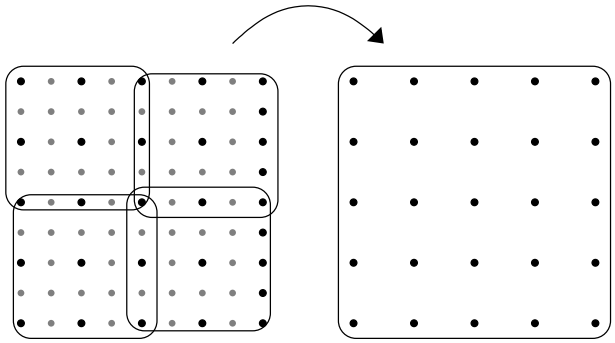

Figure 3.3: Simplification of a height field using the geomipmapping principle.

This simplification scheme gives an added benefit: it keeps the same mesh structure in all meshes. This is a very powerful property. As all meshes are kept square, regular triangulated and of the same size, it allows for simpler implementation and enables some appealing memory and speed optimizations as described in section 3.2.9 and 3.2.10.

3.2.4 Level-of-Detail Selection Using a Quadtree

Selecting the right levels of detail for rendering is simple, when a terrain is organized in a quadtree as described in the previous sections. The function in listing 3.1 shows a simple recursive algorithm that performs this selection.

```
detail_level_select(node)
    if detail level not acceptable and node is internal then
        for each child of node
            detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.1: Detail level selection using a quadtree.

The function does a recursive descent from the given node down the tree in a depth first manner. When a node with the right detail level has been reached the recursion stops and this node is then selected for rendering.

To do detail selection on the entire terrain the function in listing 3.1 is just called with the root node of the tree as argument.

To determine if the detail level of a chunk is acceptable or not an error metric is used in order to measure how large an error a chunk introduces to the rendered image.

3.2.5 Error metric

The error metric we use is the same as used by Lindstrom in the fine grained part of [13], and as used by De Boer in [3], and have been described earlier in section 2.1 and 2.5.

The error metric is used in world space to assign an error to each vertex that is removed from a chunk when simplified. The error δ is the difference in height between the vertex and the line formed by the two neighboring vertices as illustrated in figure 3.4.

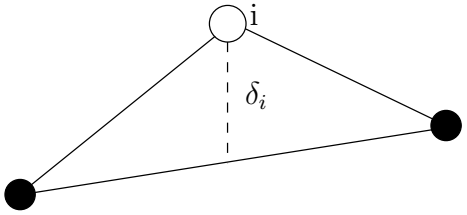

Figure 3.4: The height of δ is the error.

As our simplification is similar to geomipmapping, we employ the same scheme of finding the maximum of the vertex errors and using this as the overall error of the chunk, i.e. this gives us a worst case error. Thus an error δ_m is calculated for each chunk as $max\{\delta_1, \delta_2, \ldots, \delta_n\}$, where n is the number vertices removed when the chunk was simplified.

Intuitively a chunk with a lower detail level should have a larger δ_m than a chunk with a higher detail level. Unfortunately this property does not automatically hold, because simplification is done local to each chunk. To make sure that the error of the chunks grows up through the quadtree, the error assigned to each chunk, δ_c, are nested in a way similar to [5]. The error δ_c is the maximum of the error δ_{c_i} among the i children of the chunk plus the chunks own error δ_m. Thus, the error δ_c is calculated as given in equation 3.1.

$$\delta_c = \begin{cases} 0 & \text{if chunk is a leaf chunk} \\ max\{\delta_{c_0}, \delta_{c_1}, \delta_{c_2}, \delta_{c_3}\} + \delta_m & \text{otherwise} \end{cases} \tag{3.1}$$

By using its error δ_c we can determine if the chunk has an acceptable level-of-detail or if it needs to be replaced by other more detailed chunks[2]. This is done by projecting the the error δ_c to screen space resulting in a screen space error, ϵ. This screen space error is then compared to a user specified error threshold, τ, measured in pixels. If ϵ is greater than τ a higher detail level is needed, otherwise the current chunk is at an acceptable detail level.

Exact projection of δ_c into the screen space error ϵ is described in [13]. In [3] an approximation is made to simplify this calculation of ϵ. We have chosen

[2]It is never necessary to determine if a chunk has too high a detail level, since detail selection is performed top-down through the quadtree.

to use the same simplification, meaning that the calculation of ϵ is performed assuming that the view direction of the viewpoint is always parallel to the horizontal plane[3].

The projection of δ_c into ϵ is show in equation 3.2, where S is the height of the screen in pixels, d is the distance from the viewpoint to the chunk and fov is the field-of-view in radians.

$$\epsilon = \delta_c \frac{S}{2d|tan(\frac{fov}{2})|} \tag{3.2}$$

Instead of performing this projection every time detail selection is needed, [3] shows how the equation can be transformed to calculate a minimum distance d_m to the chunk given the error threshold τ. This way, all that needs to be calculated is the distance d from the viewpoint to the chunk and then comparing d to d_m. If d is less than d_m a higher level-of-detail is needed.

Calculating d_m is done by finding the distance d where ϵ equals τ. Equation 3.2 then becomes equation 3.3.

$$\begin{aligned} \tau &= \delta_c \frac{S}{2d_m|tan(\frac{fov}{2})|} \\ d_m &= \delta_c \frac{S}{2\tau|tan(\frac{fov}{2})|} \end{aligned} \tag{3.3}$$

For each chunk d_m could be calculated and stored, but this would result in a massive recomputation if τ should be changed. Instead we store δ_c with each chunk and precalculate the value $C = \frac{S}{2\tau|tan(\frac{fov}{2})|}$. At detail selection C is then multiplied with δ_c resulting in d_m. The advantage of this approach is that only C has to be recalculated, if τ changes.

Selecting Level-of-Detail

Using this error metric, the level-of-detail selection described in section 3.2.4 becomes as shown in listing 3.2.

3.2.6 Geometry Gaps

When rendering different parts of the terrain with different levels of detail, it cannot be guaranteed that the different meshes will line up perfectly; in fact, it is very likely that they will not. When two neighboring chunks do not line up visually unpleasant holes in the terrain appears. Many algorithms, such as [12] and [3], uses somewhat complicated techniques to "stitch" the different part

[3]This is a reasonable assumption as long as the viewpoint stays near the ground

```
detail_level_select(node)

    d = distance from viewpoint to chunk

    if d < δc(node) · C and node is internal then
        for each child of node
            detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.2: Level-of-detail selection with a quadtree using the error metric. C has been precalculated as described above.

together with extra geometry. The problem with this, besides added algorithmic complexity, is that this stitching has to be done on the cpu and thus limits performance.

A simple, yet effective way to eliminate geometry gaps is to use skirts, as described by Ulrich in [20]. That is, to add a vertical piece of geometry all around each chunk from the edge of the chunk and downwards to some depth. When two chunks with different detail levels meet, the geometry of the surface will still not line up, but the skirts will cover the holes in the terrain, resulting in less visible - in most cases imperceptible - artifacts. See figure 3.5 for an illustration of this concept.

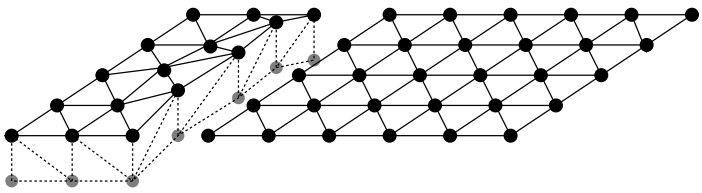

Figure 3.5: Skirst are hung from the edge of the mesh to the left in order to cover the holes that would otherwise appear because the meshes does not match at the edge.

The trick is to get the skirt height just big enough such that no holes will appear, no matter what detail levels of neighboring chunks are selected.

As described in section 3.2.5, the error metric is the height difference between a vertex and the line formed by the two neighboring vertices. That means the error metric can be used to determine the height of the skirts. Because the error is the worst case error and it is nested, so is the height, which means that a node at some detail level is guaranteed to have an error - or height - value associated with it, which is large enough to cover any gaps that might appear. Thus the error δ_c associated with a chunk can also be used as the height of the skirts to the chunk.

This method gives a slight increase in polygon count and fill rate demands. However, compared to the cost of CPU-side stitching, it is virtually free. At the

same time, it is much more simple to implement.

3.2.7 Texturing and Lighting

Level-of-detail algorithms often neglects or simplifies texturing and lighting considerations. But when working with very large terrains texturing and lighting becomes problematic as well.

Texturing

Texturing is often done by using a single large texture and relying on mipmapping for texture level-of-detail. This approach works well if the terrain is limited in size, but with large terrains, even the largest supported texture sizes are too small to provide decent texture resolutions

Our approach is to store one[4] texture in each node in the quadtree. Like the chunks, these textures only covers the part of the terrain the node covers and they all have the same resolution. This gives a simple texture level-of-detail that works the same way as the geometry level-of-detail.

Some care must be taken to avoid texturing artifacts if bilinear filtering is used. When bilinear filtering from a texel at the edge of a texture it will either use the texel itself or the texel on the opposite side of the texture as "neighboring texel" in the filtering, depending on the set texturing state[5]. None of these texels will produce the correct result, as it is actually a texel in the neighboring nodes texture that should be used. This gives "hard edges" in texturing between chunks.

It is naturally impossible to actually use the texel from the texture of the neighboring node, but a way to achieve the same result is to make neighboring textures[6] share the same texel values at the edges. It is then necessary to adjust the texture coordinates, such that the corners of the chunk is exactly in the middle of the texels in the corners of the texture. This way filtering will be correct when two textures with the same detail levels meet.

There will still be hard edges between chunks of different detail levels caused by the same effects, and in some circumstances these artifacts will be visible. We have not yet found a good solution to this problem. However, at decent texture resolutions this artifact is very rarely seen.

Lighting

We have investigated several different possibilities concerning lighting.

[4] We only use one diffuse texture, but our approach is independent of how many and how textures are used.

[5] Depending on the texturing state being either clamp or repeat

[6] Neighboring textures means textures belonging to two neighboring nodes with the same detail level.

The simplest way is to use precalculated static lighting. This could be applied in textures as *light maps* or even simpler just pre-multiplied with and stored in the diffuse texture. This method has the advantages of being simple and fast at runtime and it allows for arbitrary complex lighting calculations as they can be done as a preprocess. The main drawback is that the lighting is static and thus cannot change during runtime.

A more dynamic approach to lighting is to use the built-in lighting features of graphics hardware or by doing lighting calculations in vertex and/or pixel programs. The major benefit of this type of lighting is the ability to change lighting conditions on the fly with no slowdowns, as lighting is always recomputed at each frame. It also has drawbacks, however. Lighting calculations will require normals, which takes up space[7], and is limited in the selection of lighting models. Hardware lights uses a Phong-like local illumination model only, and while vertex and pixel programs are much more flexible still the lighting is generally limited to local models[8]. runtime lighting calculations may also be slower than the simple lighting by texturing as described above.

A third approach, which is a combination of the two previous, is possible: creating diffuse texture maps pre-multiplied with light maps at runtime. The diffuse texture maps can be created by combining an unlit diffuse texture with some lighting information and stored in a new texture. If using the graphics hardwares *render-to-texture* capabilities this can be a quick process. The lighting information can be generated in various ways, e.g. as explained above. This approach has the same speed advantages as the first approach, but is more dynamic. It is not as dynamic as calculating lighting information each frame, but it should be possible to update lighting information with reasonable intervals, depending on the hardware. This makes slowly changing lighting conditions, such as outdoor lighting caused by the sun, possible

3.2.8 Morphing

As described in section 3.2.4, the detail levels are selected on the basis of their error values and the distance towards the viewpoint. As the viewpoint moves towards a chunk, the error introduced by the chunk decreases until the point where the next level of detail will be shown.

When this change in detail level occur an artifact known as *popping* is likely to occur. Popping is when an image (or some part thereof) suddenly changes appearance, and is often very noticeable to the human eye. In this context we can experience two kinds of popping: geometrical popping, when the geometry suddenly changes and textural popping, when the texture (and/or lighting) changes. Both these popping artifacts are caused by the change in level of detail.

[7] However, to save space normals can be encoded in a texture, a *normal map*, where the x, y and z-components of the normal can be saved in the r, g and b-channels of the texture. Further compression may also be possible.

[8] Approximations towards global illumination models can be implemented, such as horizon maps [19], but they are neither simple or free.

Chunked methods are more prone to geometrical popping, because when chunks are replaced many polygons will pop simultaneously making the artifact much more visible.

Geometrical Morphing

To remove geometrical popping we employ the usual remedy: morphing. Instead of performing a discreet switch in detail level we slowly morph from one level into the next. This can either be done over time or based on the error. We suggest the latter, as this only do morphing when the viewpoint is moving which helps hide the swirling artifacts morphing may cause.

Morphing between the meshes of two levels in the quadtree is very easy because of the way we do mesh simplification. Considering a mesh and it children, the children will have twice as many vertices in each direction, but the vertices they have in common have the same positions. Thus it is only the extra vertices of the children we have to morph. If these extra vertices are projected onto the triangles of the parent mesh, the child meshes will appear exactly as the parent mesh. So to morph between these meshes we slowly move the extra vertices from the projected positions to the original positions.

Even though a morph translates each morphing vertex in 3 dimensional space, all vertices are translated the same direction (i.e. upwards) and by knowing this, only one parameter is needed to describe the translation. Thus to support morphing we suggest storing one value with each vertex of each mesh describing the *displacement* or length it needs to move when morphing.

To limit the performance penalty this morphing should be performed on the graphics hardware in a vertex program. Fortunately, this is very simple. Having a displacement value for each vertex and a morph factor f between 0 and 1, the vertex program only needs to do the operation shown in equation 3.4. When the morph factor is one, all vertices are displaced and the mesh appears at full detail but when the morph factor is zero, no vertices are displaced and the mesh appears at one detail level lower than it actually is.

$$vertex.position \leftarrow \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} + f \cdot \begin{bmatrix} 0 \\ 0 \\ displacement \end{bmatrix} \right) \tag{3.4}$$

Calculating the morph factor f is done based on the minimum distance D_m described in section 3.2.5. The morph factor is dependent on the viewpoints position relative to the current chunks d_m and its parents d_{m_p} and is calculated using equation 3.5, where d is the distance between the viewpoint and the current chunk. This calculation of f is identical to the one described by De Boer in [3].

$$f = 1.0 - \frac{d - d_m}{d_{m_p} - d_m} \tag{3.5}$$

If one uses normals for lighting purposes they should be morphed as well to avoid popping artifacts from a sudden change in lighting. We have not investigated this, however.

Textural Morphing

Our texturing scheme leads to textural popping, as the detail levels of the textures are tied to the detail levels of the chunks. Our solution is similar to the one proposed for geometrical popping: morphing. Morphing textures is easier than morphing geometry, as a simple linear interpolation is adequate.

Thus morphing from one textural detail level to the next involves using the two textures in question and blending them using either *multitexturing* or a pixel program. Given two textures and a morph factor f the pixel program should do the simple linear interpolation as given in equation 3.6.

$$pixel.color \leftarrow texture_0 \cdot f + texture_1 \cdot (1 - f) \tag{3.6}$$

Textural morphing as explained above is not constrained to diffuse texturing, but can be applied to light-maps or other similar areas.

3.2.9 Optimizing Memory Consumption

As previously discussed, all meshes in the quadtree have the same structure due to the way the mesh simplification process works.

One simple, yet very important observation to make is that all meshes are actually identical besides origin, scale and the elevation value of each height sample. A "flat" mesh of the same structure could be transformed into any other mesh in the tree by a translation, a scale and by displacing each vertex by adding the appropriate elevation values. These operations are all trivial when using vertex programs.

The operation in equation 3.7 performs this transformation. The vertex position is given as x, y, and the desired height value is given in *elevation*.

$$vertex.position \leftarrow \left(\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \cdot scale + translation + \begin{bmatrix} 0 \\ 0 \\ elevation \end{bmatrix} \right) \tag{3.7}$$

Sharing the same mesh and only storing translation, scale and height values in each chunk results in large memory savings. As vertices usually are represented as at least three 32-bit floating point values a mesh with 64 times 64 vertices consumes 48 kb. Having 256 distinct meshes results in a memory consumption of 12 mb. By utilizing the optimization described in this section and letting the

meshes share the same base mesh, each mesh only consumes 16 Kb for height values. Having 256 meshes memory consumption is then only 4 Mb plus the 48 Kb for the shared mesh. This is a saving of almost 66%.

Further savings can be achieved by compressing the height values. Simple 16 bit or even 8 bit quantization is possible with negligible loss in quality. Using 16 bit height values the 256 meshes from above only consumes 2 Mb of space for height values and the total reduction in memory consumption is around 83%.

3.2.10 Optimizing Polygon Throughput

Most modern graphics hardware are equipped with a *post-T&L vertex cache* [17]. This is a relatively small FIFO cache which stores vertices after being fetched from memory and transformed. When a vertex required for rendering resides in the vertex cache the cost of fetching and transforming it is saved.

As most vertices in our meshes are shared by many polygons[9] utilizing the vertex cache may increase polygon throughput, if memory bandwidth and/or vertex transforms are the limiting factor of the particular graphics hardware.

Our base mesh is a square, regularly triangulated mesh, and as such it can be optimized for nearly perfect vertex cache usage.

The optimal way is to draw the triangles of the mesh in a zigzag fashion, as illustrated in figure 3.6. The width of one zigzag is the half of the vertex cache size, as all vertices will stay exactly long enough in the cache to be reused.

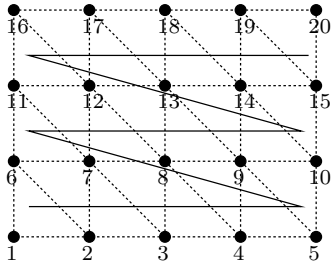

Figure 3.6: By creating a triangle strip in this way, half of the vertices are reused most of time. This examples has been optimized to a vertex cache with 10 entries.

Rendering the meshes in this fashion, each vertex is only fetched and transformed once, except the vertices used in more that one column of zigzags.

The polygon throughput can be further optimized by using triangle strips as rendering primitives. However, when the mesh is optimized for the vertex cache as above, there is no fetching and transformation to be saved by using strips. What is gained, is a reduction in indices needed to render the same amount of triangles. Using strips the amount of indices nearly drops to one third. By

[9] All vertices except those on the edges are shared by 8 polygons each.

including some degenerate triangles[10] it is possible to contain the entire mesh in one strip. See figure 3.7.

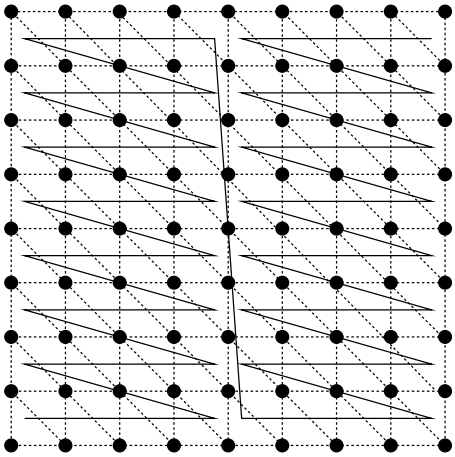

Figure 3.7: Vertex cache optimized stripification of a terrain mesh.

3.2.11 Front-to-Back Rendering

Besides the post-T&L vertex cache most graphics hardware are also equipped with a *depth buffer*[11], that enables it to check whether or not a pixel that is just about to be rendered to the color buffer is closer to the viewpoint than a pixel that might already have been drawn to the same position. If the new pixel is closer to the viewpoint it is rendered to the color buffer otherwise it is discarded.

The depth buffer can be utilized to speed up the rasterizer stage as shown in [1], by doing front-to-back rendering of polygons, this makes it possible for the graphics hardware to discard alot of pixels early in the rasterizer stage, which is faster than rendering the pixels to the color buffer.

We are doing front-to-back rendering by sorting the chunks according to the distance from the viewpoint to the chunks in increasing order. The sorting of chunks is done while descending down the quadtree by sorting the children and the descend down the child that is closest to the viewpoint. A slight change is made to `detail_level_select(node)` as shown in listing 3.3. In this way the chunks that are the closest to the viewpoint is displayed first.

[10] Degenerate triangles is triangles of zero area (which thus produces zero pixels when rendered) to ensure the right winding order of the triangles in the strip.

[11] Also known as a Z-Buffer

```
detail_level_select(node)

    d = distance from viewpoint to node

    if d < δc(node) · C and node is internal then
        sort children according to distance
        for each child of node
            detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.3: Detail level selection combined front-to-back sorting.

3.2.12 View Frustum Culling

Since the quadtree already is organized as a 2 dimensional spatial partitioning structure it supports very simple, yet effective hierarchical view frustum culling: if a node is outside the view-frustum all its children are also outside. This fits well with the detail level selection algorithm in listing 3.1. Adding view frustum culling to this algorithm is simple, as shown in listing 3.4.

```
detail_level_select(node)
    if node does not intersect view frustum then
        return

    d = distance from viewpoint to node

    if d < δc(node) · C and node is internal then
        sort children according to distance
        for each child of node
            detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.4: Detail level selection combined with view frustum culling.

When doing intersection tests between nodes and the view frustum it is advisable just to test a bounding volume of the chunks against the view frustum instead of the triangulated mesh in the chunks. Testing bounding volumes such as spheres or axis aligned boxes is much faster than testing a triangulated mesh and is definitely precise enough for this purpose.

Note: when generating a bounding volume for a node care must be taken that the extent of a node may be lesser than the extents of its children, due to simplification. Thus the bounding volume of a node must be extended such that it will contain all the meshes of all children. Otherwise, the algorithm may cull an invisible parent resulting in removing an otherwise visible child.

The simple way to ensure this property is to make the bounding volume of a node contain the bounding volume of all its children.

3.2.13 Variable Detail Level Quadtree

A requirement for evenly spaced height samples in the source height field is one of the limitations of our method. In some applications, however, this might be wasteful. If the eye point is restricted to some parts of the terrain, only these parts needs high sample density while lower densities is adequate for the rest of the terrain since it is only seen from a certain distance. Significant space savings can be achieved by using this technique.

It is possible to remove this limitation to a certain degree. The height field still must have evenly spaced height samples, but the quadtree build from this height field can have varying maximum detail level in different areas.

This effect is achieved by *pruning* the generated quadtree. That is removing the leaf nodes in the branches of the quadtree that covers the areas of the terrain where lower sample densities are wanted. This pruning continues until the desired sample densities are reached.

We suggest having a weight function which gives the *minimal* weight for an area covered by a node. If this weight is less than some threshold, the node is not necessary. If - and only if - all four children of a node are not necessary they can be removed from the tree[12].

Obviously, it is somewhat inefficient to generate the entire tree and then prune it, so these two processes should be combined for efficiency.

3.2.14 Out-of-Core Support

Even though our method is very memory efficient the amount of memory (ram) will always be a limiting factor on the size of the terrain. The solution is to do out-of-core rendering. Out-of-core literally means using data which is not in core (system) memory. Data may reside on hard disks or on remote servers connected through a network etc. We have only investigated hard drive support.

Because each node in the quadtree is totally independent of all other nodes in the tree, a node is only needed in memory when it is being rendered. This leads to a simple out-of-core support system, in which the detail-level selection simply asks for child nodes to be loaded into memory, when their parent is found to have a too low detail level. The detail selection algorithm from listing 3.4 is then extended as shown in listing 3.5.

This, however, requires some sort of cache in order to keep track of the nodes in memory (the *working set*) and unload unneeded nodes as necessary, or else the above method will eventually run out of memory anyways.

The LRU cache policy[13] works well in this case, since we want to keep the most used nodes in memory while dispose the ones least used.

[12] This is due to the constraint the quadtree imposes: a node *must* have either four or no children.

[13] LRU is an acronym for Least Recently Used and means the cache, when full, will start replacing the items in the cache which have resided in the cache longest without being used.

```
detail_level_select(node)
    if node does not intersect view frustum then
        return

    d = distance from viewpoint to node

    if d < δc(node) · C and node is internal then
        if children not in memory then
            load children into memory

        sort children according to distance

        for each child of node
            detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.5: Detail level selection with simple out-of-core support.

To keep the cache updated, we mark a node as used and put it in front of the cache queue each time we call detail-select-node on it. As detail-select-node calls itself recursively, thus each node it visits as it descents down the tree will be marked as used.

With out-of-core support the factor that limits terrain size is no longer memory but storage, e.g. hard drives. Fortunately as storage is much more affordable, this is not very limiting in practice.

File Structure

A key to efficient hard drive based out-of-core support is a sensible file layout as it speeds up disk access. Keeping related data together physically in the file minimizes the costly seek operations. We store all data related to one node contiguously in the file. We also store siblings contiguously, as they are always loaded at the same time[14]. To keep track of all data in the file, we have placed a directory which contains the start addresses of each node in the file. Finally a header is added which contains the address of the directory, file version number etc. See figure 3.8 for a graphical sketch of the file layout.

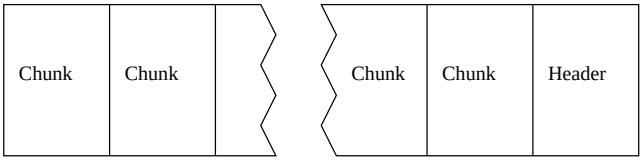

Figure 3.8: The structure of the file.

[14]When a node is found to coarse in the detail selection algorithm, all its children are requested to be loaded. See listings 3.5.

3.2.15 Concurrent Client/Server Design

The algorithm described in this chapter is very suitable in a design similar to a client/server pattern.

The client is the rendering part which asks the server for nodes, when needed during detail selection. The server handles the out-of-core issues such as loading data from disk and caching it. By using asynchronous messaging between the client and the server, the parts could be multi-threaded, allowing time spend waiting for disk i/o to be used for rendering purposes.

This changes the detail selected algorithm slightly, as shown in listing 3.6.

```
detail_level_select(node)

    d = distance from viewpoint to node

    if d < δc(node) · C and node is internal then
        if children not in memory then
            request loading of children into memory
            select node for rendering
        else
            sort children according to distance
            for each child of node
                detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.6: Detail level selection with asynchronous out-of-core support.

Now, the client does not wait until child meshes are loaded into memory, but just selects the parent for rendering as long as the children are not in memory. This allows the rendering to continue smoothly albeit at a lower quality, while the children are being loaded into memory.

3.3 Detail Generation

This section will describe the detail generation system developed in this project.

First we will show how details can be added to an existing height field and how the details can be calculated. Then we will show how detail synthesis can be combined with the level-of-detail algorithm.

3.3.1 Adding Details to Height Fields

Simply speaking adding more details to a height field requires adding more samples to the height field. A simple way to do this would be to quadruple the sample density by adding samples between each row and column, as in figure

3.9. This can be done recursively, thus giving a more and more detailed height field.

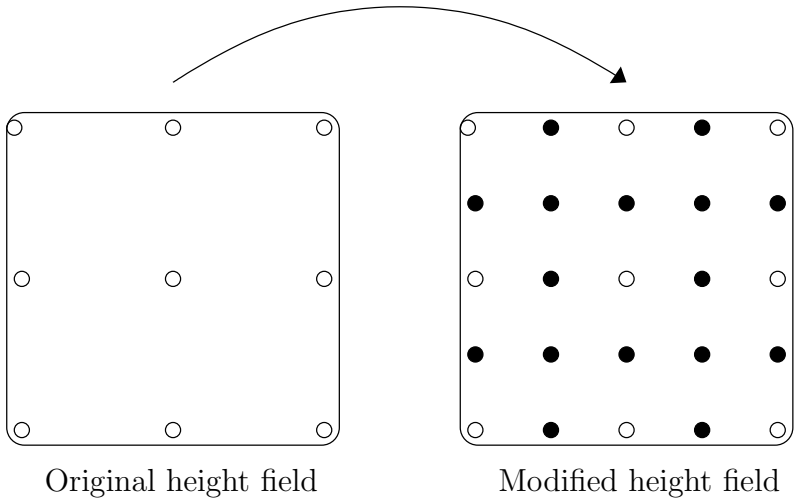

Figure 3.9: One iteration of adding details.

Generating New Samples

Ideally the new samples should add additional details to the terrain. At the same time the new sample values should relate to the existing sample values, so the new values should also be a kind of interpolation of the existing values.

Doing linear interpolation between existing samples places the new samples on straight lines between the original samples, which brings nothing new to the height field. Doing higher order interpolation will smoothen the height field, but still does not bring real details.

To get the wanted details, a higher order interpolation combined with careful displacement of the new samples could be used. The displacements would perform the necessary mutation of the surface which results in extra details. Figure 3.10 shows an example of this.

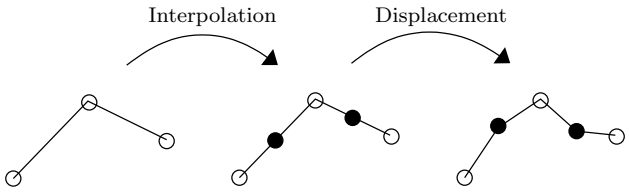

Figure 3.10: Interpolation followed by displacement.

The Displacement Function

To generate the height displacement of the new samples we use a *displacement function*. Because a height field is essentially a 2D grid where each samples has a position in the (x, y)-plane, the displacement function is a 2D function that gives a single displacement value. Furthermore, if $(x_1, y_1) = (x_2, y_2)$ the function must return the same value for both inputs, such that the same area of the height field can be calculated several times with the same result.

Separating Interpolation and Displacement

Unfortunately a problem occurs when recursively interpolating and displacing the same height field. The problem is that the interpolation between samples will also interpolate between the previously added detail samples, such that the new detail samples will in effect sum all previous displacements. This often results in the new samples getting displacements from the original mesh that differs widely from the intended displacements generated from the displacement functions, making it very hard to control the actual displacements.

Because we want to perform detail addition recursively, we found it necessary to do things a little differently, namely to seperate the interpolation and displacement. Thus, we suggest having a *base height field*, on which recursive interpolation can be performed[15] and to have a *displacement height field* that is added to the interpolated samples, before the height field is rendered.

The base height field starts out the same as the height field used for generating the quadtree, but can be extended with more samples by interpolation.

The displacement height field is similar to the base height field, i.e. it is equal in size and number of samples. The difference is that the samples values are obtained from the displacement function instead of from interpolating of the height field.

Creating the Displacement Height Field

Creating the displacement height field is not difficult. We know the (x, y) world space position of each sample in the base height field, so creating the displacement heights for the displacement height field is simply a matter of evaluating the displacement function at the locations of the base height field samples.

3.3.2 Subdivision Surfaces

When extending the base height field, the new sample values are interpolated from the existing sample values.

[15] Recursive interpolation does not yield any problems when displacements are omitted.

If linear interpolation between samples is used a problem arises: Sharp edges stay sharp. This is because linear interpolation only gives C^0 continuity. While this could be acceptable in some cases, we specifically want to remove any sharp edges of the base height field, giving a smooth curving height field onto which we can add details through the displacement height field. Thus a higher order interpolation is needed, as shown in figure 3.11.

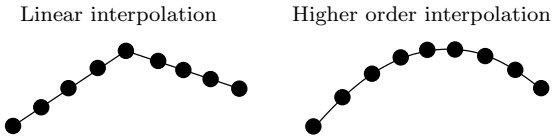

Figure 3.11: Linear interpolation versus higher order interpolation.

This problem is actually what subdivision surfaces are designed to solve. For our purpose the subdivision surface algorithm described by Kobbelt in [11] fits very well.

We will not present Kobbelt's subdivision algorithm here; it is described very well in the paper by Kobbelt and our use requires no changes to the algorithm. Using this subdivision algorithm results in a base height field with C^1 continuity at the limit[16].

The result of using subdivision surfaces as interpolator of the base height field is that that it becomes very smooth; the sharp edges that might be in the original height field becomes rounded and soft, making the base height field a good base for adding details by displacements.

3.3.3 Detail Synthesis

We have not discussed the actual displacement function yet for a reason: It could be anything. How it should behave depends entirely on what details are wanted, but in general we wish natural and realistic looking details.

Fractals

A lot of research has be done to create synthetic landscapes that looks realistic. One of the most successful attempts at this has been fractals.

Fractals are not really designed to model the complex systems of nature to generate realistic results, rather it is simple mathematical functions that utilizes what is called *self similarity*. When looking at an image of a part of a mountain, a beach head or a branch of a tree, it is often hard to tell the real size of the object. A small part of a mountain looks surprisingly similar to a much larger part of a mountain when scaled, and that is what self similarity is.

[16] That is, if it were subdivided infinitely.

Fractals, then, uses some defined shape and merges several versions of this shape at different scales to generate surprisingly complex and life-like results, among these terrains.

As fractals are simple but powerful tools for generating terrain features, they are perfect as displacement function in our detail synthesis system.

Basis Functions

The shape fractals uses is called a basis function. Again, a basis function can virtually be anything. But often the basis function is just a noise function of some sort. Perlin noise, as presented in [6], fits this purpose well.

Another interesting basis function is bitmaps. Using bitmaps allows artists much greater control over the displacement function and thus the appearance of the final terrain.

From the basis functions a fractal can be created in different ways, as presented in [6]. Common for all methods is that the basis function needs to be sampled several times to get a decent amount of detail in the fractal. The number of times the basis function is sampled is called the *octaves* of the fractal.

Real-time Fractals

Fractals are relatively computational intensive because of the many basis function samples needed to get a reasonable level of detail. This make it difficult to create the displacements height field in real-time, but some tradeoffs between speed and quality can be employed.

One way to improve speed is to do caching of basis functions. This is done by sampling the basis function at discreet intervals and storing the results. Later when the basis function is sampled by the fractal function, the value is found by interpolating the samples of the cached basis function. This concept improves the sampling time of the fractal slightly at the cost of precision of the basis function and thus the quality of the fractal.

Another approach is to cache the actual fractal. Again this is done by sampling and storing. Whenever a height samples is requested from the fractal the result is found by interpolating the cached values. This improves speed greatly, but again it limits precision, and if not careful this becomes visible.

3.4 Combining Level-of-Detail Algorithm and Detail Generation

In this section we will describe how to combine the level-of-detail algorithm presented in section 3.2 with the detail generation system presented in section 3.3.

As already discussed adding more details to a height field is done by adding more samples. As the height field in our level-of-detail algorithm is stored in a chunked quadtree, as described in section 3.2.2, the quadtree must be extended with new chunks in order to add more samples.

As stated previously, our goal is to do this extension at runtime, so a low resolution height field can be used as input, yet it will appear highly detailed when rendered.

Dynamic and Static Nodes

In the next sections we will refer to the nodes in the input quadtree as the *static nodes* and the nodes used to extend the quadtree as the *dynamic nodes*. The only real difference between static and dynamic nodes are their origin: static nodes are read from the input file while the dynamic nodes are generated by interpolation and displacements, as described in section 3.3.

Even if it is possible to have all the static nodes of the quadtree in system memory, it is obvious that if we were to extend the quadtree at all leaf nodes with dynamic nodes, we could quickly run out of system memory, as the number of quadtree nodes grows exponential in relation to the depth of the quadtree. Instead, we suggest only to extend the quadtree in close proximity to the viewpoint. When the viewpoint moves, the previously generated dynamic nodes could then be removed, thus keeping memory usage bounded.

3.4.1 Extending the Quadtree at Runtime

In the level-of-detail algorithm the leaf nodes of the quadtree should have an error value δ_c of zero, such that the recursive refinement of the level-of-detail algorithm stops. This must be changed, such that the static leaf nodes still have some error, such that the refinement continues beyond the static nodes through the dynamic nodes inserted at the static leafs.

Approximating Leaf Node Error

It is hard to get an exact error value at the static leaf nodes as it depends on the subdivision scheme and the displacement function. In theory an infinite amount of subdivisions and displacements should be done and the error of the static leafs could then be calculated as usual. This, however, is not practical, so the error must be approximated.

We approximate this error by stochastic sampling the error of the displacement function. We basically perform a series of random detail additions and uses the maximal error found as the approximated error of the static leafs.

Because the errors in the quadtree are nested, any change in the error at the leafs means that all errors in the tree have to be adjusted. Fortunately this only

needs to be done once, and the estimation and assignment of errors to the leafs could be done as a preprocess.

Approximating Dynamic Node Error

In order for the level-of-detail selection to work beyond the first level of dynamic nodes, these nodes also needs to have an error δ_c which is greater than zero. Finding the exact error of the dynamic nodes is as hard as finding the errors of the static leafs, which makes it impractical.

We have chosen to simply give the dynamic nodes half the error of their parent, which is very crude but works surprisingly well in practice.

3.4.2 Adopted Detail Selection

Since nodes are dynamically added to the quadtree when needed, all nodes in the tree are in principle internal. This changes the level-of-detail selection algorithm slightly, because it is no longer necessary to check if the nodes are internal. The adopted detail selection algorithm is shown in listing 3.7.

```
detail_level_select(node)

    d = distance from viewpoint to node

    if d < δc(node) · C then
        if children not in memory then
            request loading of children into memory
               select node for rendering
        else
            sort children according to distance
            for each child of node
                detail-level-select(child)
    else
        select node to be rendering
```

Listing 3.7: Adopted detail level selection.

3.4.3 Adding Nodes to the Quadtree

When the leaf nodes of the quadtree has an error greater than zero it is possible for the level-of-detail algorithm to determine that a leaf node has too high an error and higher detailed chunks are needed.

Creating the base height field for the dynamic nodes is the reverse of the simplification scheme used by the level-of-detail algorithm. First the parent's base height field is divided into four smaller height fields and the sample density

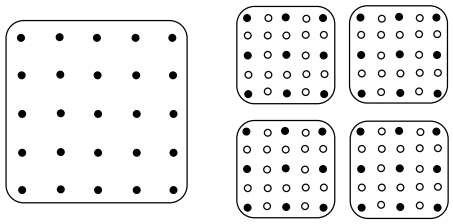

Figure 3.12: Four new chunks are created by copying vertices from the parent (shown in black) and then adding new vertices (shown in white).

is quadrupled by adding new samples that are put in between each row and column in each new height field, as shown in figure 3.12.

The height values of the new samples are calculated using the subdivision scheme described in section 3.3.2. These four new height fields are the base height fields of the four new child nodes.

Four detail height fields similar to the four new base height field are also created and these are filled with height values calculated by the displacement function.

Using these new height fields four new dynamic nodes are created and inserted into the quadtree as children to the parent who requested them.

Borders

As the subdivision is done internally to each chunk and as chunks has no knowledge of their neighbors, a problem arises: When subdividing the chunks at the edges, the subdivision scheme needs samples in the neighboring chunk to guarantee C^1 continuity.

Without these samples, extrapolation is used to estimate the samples, but this results in C^0 continuity as the edges between chunk. This is a visually disturbing artefact and should be avoided.

To avoid this problem we propose to extend the base height field in each chunk with a border of extra samples around the "real" base height field; these border samples are only used during subdivision.

This way, even though the chunks are still independent, the chunks knows which value the samples at the edges of their neighbors holds and subdivision can be performed without visual artifacts.

Creation of Dynamic Nodes

The creation of the dynamic nodes is suitably done in the server-part of the algorithm described in section 3.2.15.

The server then has to determine whether a requested node is available as a static node or if it has to be created as a dynamic node. When dynamic nodes are created they are also cached, just like the static node, to improve performance.

3.4.4 Limiting Dynamic Node Creation

The asynchronous version of the level-of-detail algorithm, as described in 3.2.15 performs best, but as a side effect the requested children are not always immediately available. This can result in an unpleasant scenario: If the viewpoint moves very fast through the terrain, many new dynamic nodes will be requested, but when they become available they are probably not needed anymore because the viewpoint has moved significantly since they were requested.

Thus, many nodes are created but not used, which is poor use of resources. If too many nodes are requested, creating these might take so long to create that creation of important, visible nodes are delayed causing severe visual artifacts.

Also, these unused nodes will pollute the cache which may result in poor performance.

A partial solution is to limit how many requests that are able to be queued. If this is reasonably low, say 4 - 8 nodes, new requests will be denied as long as the old ones are not finished. This will somewhat limit the problem when the camera is moving fast, but it will not eliminate it.

A better solution is to dynamically limit how small details should be requested, depending on the velocity of the viewpoint. When the viewpoint is moving fast small details are not really perceptible, so we can simply omit creating them in the first place. Thus, before allowing a request for new dynamic nodes, the ratio between sample density of the new nodes and the velocity of the viewpoint should be calculated. If it is too low, the request should not be accepted.

This approach effectively solves the problem, as it limits the amount of nodes generated when the viewpoint is moving fast.

3.4.5 Materials

Using the same displacement function over the entire terrain, is a little dull and not very realistic. It would be more interesting if steep mountain sides very given a hard, rock like surface, while small hills were to have a more soft surface.

And why stop there? It would also be nice if the hard rocky mountains were given a rock like texture, while the soft hill were given a grass like texture. The textures that lie in the chunks are probably to low in detail, to give more texture than just a basic color. Higher resolution textures would be appriciated.

Our approach has been to create containers which we call *materials*. Each material has a displacement function and a set of textures to use.

Placing Materials in the Terrain

When different materials have been created they need to placed in the terrain. In [7] Hammes presents an algorithm for identifying the ecosystem of a terrain. This can be used by matching materials to ecosystems and letting Hammes algorithm select which materials to use at different locations in the terrain.

3.5 Summary

In this chapter we have presented our terrain rendering algorithm. It is an algorithm capable of rendering very, large very detailed terrains.

As explained, our algorithm is divided into two parts: The first part is a level-of-detail algorithm that has much in common with the algorithm presented by Ulrich in [20], mixed with some elements of Geometrical Mipmapping as presented in [3]. The second part of our algorithm is a detail generator that adds more details to a height field.

The level-of-detail algorithm is based upon a chunked quadtree, where each node in the quadtree contains a height field, which is a part of the original height field at some resolution. With each node is also associated an error value that indicates how much the height field it contains deviate from the original height field. Based on the viewpoint position and directions, the error values are used in detail level selection to determine which nodes should be rendered.

By giving the leaf nodes of the quadtree an artificial error, the level-of-detail selection is tricked into thinking that there are more nodes and therefore more details in the height field. When the level-of-detail requests nodes that are not in the quadtree the detail synthesizer generates new nodes, using some technique to generate extra detail samples.

The result is an algorithm that is able to render very detailed terrains, even with a low resolution source height field, in real-time on standard hardware.

Chapter 4

Implementation

This chapter presents our implementation of the terrain rendering method developed in this project.

The implementation consists of two programs: the main rendering application and a preprocessing application. The preprocessor reads a terrain heightfield from some source and generates the quadtree data structure and saves it in a data file. The preprocessor is explained in Section 4.1. The render reads the data file generated by the preprocessor, displays the terrain and lets a user control the viewpoint in realtime. The render is presented in section 4.2.

About the code

The programs that constitutes our implementation is written in C++ using Microsoft Visual C++ .NET and are targeted for the Windows operating system. They have been developed on Windows 2000 and Windows XP, but runs on other variations of Windows as well.

We have focused on simplicity and clarity. Thus the implementation may not be the most compact or the fastest possible, but it is hoped that the source code is comprehensible.

Unfortunately, due to time constraints the source code has not been commented. But, as we have tried to write clean and structured code with descriptive naming, we hope it will be readable anyways.

Third Party Libraries

Besides using the two major 3D APIs, OpenGL and Direct3D, we use a few 3rd party libraries. The CG library from NVIDIA is used in conjunction with OpenGL to access the programmable graphics hardware. The libraries libpng [18] and zlib [16] are used for reading images in the png-format. The CGLA and accompanying libraries [2] are used for vector and matrix math and timers.

4.1 Preprocessor

The preprocessor is a stand-alone program for building the quadtree of static nodes from a source height field file.

Source files for the preprocessor are height fields which can be stored either as a bitmap image, the binary terrain format[1] or a Terragen[2] file.

4.1.1 Program Structure

The preprocessor program is build around the class QuadTreeGenerator and some auxiliary classes used for reading input files, writing output and compressing images among other things. The QuadTreeGenerator supplies the core functionality of the simplification of a height field and building of a quadtree.

Main Entry Point

The main entry point, the *main*-function, is located in the file Preprocessor.cpp. This function parses the input from the user and initializes the auxiliary classes, depending on the input, before it initializes the QuadTreeGenerator.

Before the Simplification

A height field is read by the HeightMapGenerator, a texture is read or created if not supplied as input, an optional weight map is read if one is supplied, the ImageCompressor is initialized and the QuadTreeSaver is initialized.

The WeightMap correspond to the weight function described in section 3.2.13.

The ImageCompressor is used to compress the texture images using the graphics hardware.

Finally the QuadTreeSaver writes all the nodes to disk in the order it receives them.

The Simplification

When the QuadTreeGenerator is initialized it starts to create the quadtree. The quadtree is build top-down simply by picking samples out of the original height field and storing them in the smaller height fields that are attached to the nodes.

The building is done recursively until it is detected that the sample density of the nodes equals the samples density of the original height field or until the weight map determines that no more nodes are needed, which ever comes first.

[1] Defined at http://www.vterrain.org/Implementation/BT.html
[2] http://www.planetside.co.uk/terragen

The displacement height field is also created during the recursion, simply by sampling the displacement function at the positions the correspond to the positions of the samples in the base height field. The displacement height field is needed to calculate the correct error of the chunks. It is not saved to disk.

As written in section 3.2.5, the errors of the nodes needs to be nested bottom-up. This is handled by collecting and nesting the errors while the recursion is unwinding.

Before the nodes get stored the base height field is compressed to save space.

Storing the Quadtree

While building the quadtree, each node is given an id, which is an unique identification number. The root node always gets the number 0. Each parent node in the quadtree is told the number of its four children, to be able to reconstruct the quadtree when loaded from disk.

When the QuadTreeSaver receives a node to be stored, it records the nodes id number and the current position in the quadtree storage file in a directory. This directory is stored at the end of the quadtree storage file and is later used by the rendering application to look up the position of nodes in the quadtree storage file.

4.2 The Rendering Application

The rendering application, or in short the render, is the program that implements our terrain rendering method as explained in Section 3.2, 3.3 and 3.4.

4.2.1 Program Structure

This program is structured in a concurrent client/server fashion, like described in section 3.2.15.

Main Entry Point

The main entry point is located in the file TEngine.cpp. It does little more than create the class that represents the entire render engine, namely the class Engine.

Any arguments given at startup are also parsed. These are interpreted either as the name of a data file to use or the names of one or more configuration files to parse. The configuration files are explained in section 4.3.

The Engine Class

The class Engine is a very central class, as it is responsible for overall creation of the entire rendering system. Most functionality is aggregated to other classes, as shown in figure 4.1.

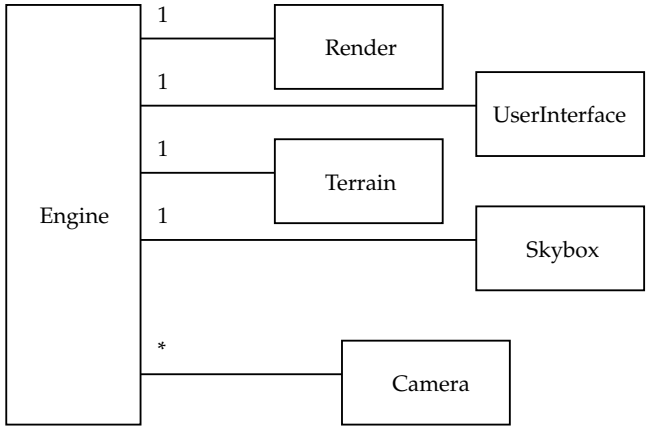

Figure 4.1: The Engine class and its aggregates.

The Render class encapsulates the low-level rendering system, the Terrain class is responsible for the terrain rendering algorithm developed in this project and the Skybox class delivers pretty backgrounds for our terrains. A simple user interface is encapsulated in the class UserInterface which allows for easy textual output to the screen.

Engine itself handles viewpoint movement based on input from the low-level rendering subsystem[3]. The viewpoint itself is encapsulated in the Camera class, which is then supplied to the low-level render who sets the transformation matrices accordingly.

4.2.2 Level-of-Detail System

The class Terrain is the facade of the terrain rendering system. It implements the detail selection algorithm explained in chapter 3. The implementation of this algorithm is combined with rendering, such that selected nodes are immediately scheduled for rendering. The methods `recursive_draw` and `recursive_sorted_draw` implements the detail selection algorithm and drawing. Generating details is controlled by the class Material

The helper method `should_subdivide` determines when chunks should be replaced by their children and the method `calculate_morph_factor` calculates the morph factor for chunks. Both using the error metric described in section 3.2.5.

[3]This is not the most intuitive design, but derives from the usage of GLUT in the OpenGL part of the low-level render, as GLUT is also used to acquire user input.

The Terrain class also owns the quadtree data structure.

4.2.3 The Rendering System

The rendering system is build up around the main render, a set of render states including vertex and pixel programs and a set of *managers*. All of these classes are interfaces and must be implemented with a 3D API[4].

The managers involved are a WindowManager, a MemoryManager and a TextureManager.

The WindowManager handles creation of the output window and user input via keyboard.

Handling of memory for the mesh and the elevation values is done by the MemoryManager. The memory is allocated through the chosen low-level 3D API to ensure that the most suitable form of memory is allocated, but whether that is video memory, system memory or something third is often beyond our control.

Allocation of textures is done by the TextureManager.

Fiannly there is a render-to-texture mechanism, used for creating diffuse textures with light information.

Render States

The different subsystems in the engine is the terrain engine, the skybox, the user interface and the render-to-texture mechanism, each of these requires different textures and hardware states. All of this is wrapped up in the RenderState and all its subclasses TerrainRenderState, SkyboxRenderState, UIRenderState and TextureRenderState. Each of these classes are, like the managers, just interfaces that must be implemented to support a certain 3D API.

The hardware states that these render states handle also include the vertex and pixel programs. Each render state has its own set of vertex and pixel programs specifically developed for the specific task that the render state represents.

The Render

The Render class is the main class of the rendering system, it ties all the managers and render states together and does the actual rendering of meshes.

Preparation For Rendering

Before any rendering takes place some initialization is done. The different managers do different initializations.

[4]This has be done for Direct3D and OpenGL.

The TextureManager allocates as many textures as there can be nodes in the node cache, the MemoryManager allocates memory for all the elevation values and for the template mesh.

The template mesh is, as described in section 3.2.9, just a flat mesh. It consists of a number of vertices and an index list. The index list is optimized for the vertex cache as described in section 3.2.10.

The actual creation and triangulation of the template mesh is done by the HeightMapMesher class.

Rendering

For a subsystem to render its information to the screen it first enables the RenderState associated with the subsystem, then it sends the information to the Render class for rendering.

Depending on the subsystem, the information to be rendered can be a mesh, which is the case for the terrain, the skybox and the render-to-texture subsystems, while the user interface requests to have text rendered to the screen.

Terrain Mesh Rendering

The rendering of the terrain meshes is done using the method presented in section 3.2.9.

Even though each node in the quadtree has its own Mesh class, the actually geometry pointed to by the Mesh class is always the same[5], the difference only lies in the elevation values and textures pointed to by the Mesh.

The scheduling of the mesh for rendering is done by the method `draw_node` in the Terrain class. First the translation and scaling is setup through the render, this is equivalent to *scale* and *translation* of equation 3.7.

Through the TerrainRenderState the morph factor for the node is set and the textures are set.

Then the render is asked to render the mesh. The render binds the mesh to the low-level 3D API, telling it where to find the geometry and the elevation values needed for equation 3.7 and then the rendering is started.

The two equations, equation 3.7 and equation 3.4, that is used for mesh expansion and vertex morphing during rendering, are executed on the graphics hardware and is implemented in TerrainVertexShader.cg.

[5] I.e. the template mesh.

4.2.4 Detail Generation System

The detail generation system is build around the classes QuadNodeGenerator and QuadSynthesizer. The QuadNodeGenerator is used for creating both static and dynamic nodes and the QuadNodeSynthesizer is used to create the base heigt field of the dynamic nodes and to create the displacement height fields for both static and dynamic nodes.

Creating the Dual Height Field

At first it is determined if the requested node is static or dynamic. If the node is static the base heigth field is just read from disk, but if the node is dynamic the base height field is synthesized, by the QuadSynthesizer. The QuadSynthesizer generates the base height field for the new dynamic nodes, based on the parent nodes base height field, using the class KobbeltSubdivider.

Then the displacement height field for the node is build, by sampling the displacement function of the supplied material. The two height fields are put into the class DualHeightMap.

The DualHeightMap has the special functionality, that if a sample value is requested at a position (x, y), it returns the sum of the sample value at (x, y) in the base height field and the sample value at (x, y) in the displacement height field, which is actually what it needed to get the correct elevation values for the mesh.

Creating Nodes

When the DualHeightMap is ready, the QuadNodeGenerator gives it to the HeightMapMesher. The HeightMapMesher retrives a section of the memory allocated by the MemoryManager and copies the elevation values from the DualHeightMap into this memory.

The elevation values are not copied directly into the elevation memory because the skirts, as described in section 3.2.6, has to be taken into consideration and this is done by the HeightMapMesher.

If the elevation values in the DualHeightMap are compressed as they are in static nodes, they are decompressed before being copied. Some hardware will allow us to transfer the compressed values to the graphics hardware and do the decompression in the vertex program, but unfortunately not all.

Finally, a texture with lighting information is created for the node using the render-to-texture mechanism.

4.3 Configuration Files

To be able to reconfigure the terrain rendering engine, other than at compile time, a small configuration scheme was implemented. A configuration file, `tengine.cfg`, is read when the rendering engine is executed. The file has a number of options the user can specify, which are explained in appendix A.

4.4 Usage

Both programs takes a number of inputs which are explained here.

4.4.1 Preprocessing

The preprocessor executable is called `prep`.

Requirements

The preprocessor does compression via the graphics hardware therefore it requires OpenGL and graphics hardware that supports texture compression.

It is know to work on Windows 2000 and Windows XP with either a NVIDIA GeForce3, NVIDIA GeForce4, ATI Radeon 9700 or ATI Radeon 9800.

Arguments

The first argument to `prep` must be a height field in one of the supported height field formats.

After the height field file at number of optional arguments can be given, which are listed below.

-t Followed by a file name will load a texture to use on the terrain. The texture file must be a `png` file. If no texture is given one is generated.

-w Followed by a file name will load a bitmap to use as weight map. The weight map must be a `png` file.

-h Followed by a floating point number specifies the spacing in metres between samples in the height field file. If the height field file already contains a spacing this option has no effect.

-v Followed by a floating point number specifies a vertical multiplier to use on the samples in the height field file. If the height field file already contains a vertical multiplier this option has no effect.

-o Followed by a filename specifies the name of the quadtree file. If this option is not given then the name of the output file will be the same as the input file followed by a `.map`.

4.4.2 Rendering

We have called the executable for the rendering system: `TEngine`.

Requirements

TEngine requires DirectX 9.0 or OpenGL and CG. On the hardware side it requires graphics hardware with programmable vertex and pixel pipelines.

It is known to work on Windows 2000 and Windows XP with either a NVIDIA GeForce3, NVIDIA GeForce4, ATI Radeon 9700 or ATI Radeon 9800.

Arguments

The executable takes an infinite number of arguments of either configuration files or quadtree files. Configuration files are recognized as filenames which end in `.cfg`. Otherwise the argument is thought to be a quadtree file.

If more than one quadtree file arguments is given only the last argument is used as quadtree file.

If more than one configuration file are given they are all parsed. The options in each new configuration file overrides all previous option of same type.

Controls

While the terrain renderer the viewpoint can be controlled by the user.

The mouse controls the viewing direction and the viewpoint is moved via the keyboard using the following keys:

w Moves the viewpoint forward along the viewing direction.

s Moves the viewpoint backward along the viewing direction.

a Moves the viewpoint to the left orthogonal to the viewing direction.

d Moves the viewpoint to the right orthogonal to the viewing direction.

q Moves the viewpoint upwards orthogonal to the viewing direction.

z Moves the viewpoint downwards orthogonal to the viewing direction.

1 Increases the speed of all viewpoint motions, except changes in viewing direction.

2 Decreases the speed of all viewpoint motions, except changes in viewing direction.

l Toggles wireframe rendering on the terrain.

esc Exits the application.

4.5 Shortcomings and Defects

The implementation is stable and a reasonably complete implementation of our method, but a few errors and missing features exists, as listed below:

- Multiple materials are not implemented. This means that the entire terrain will be covered with the same materials.
- The Direct3D version of the render has some minor texturing artifacts, due to missing texture states.
- The cache containing quadtree nodes can be overfilled, if the error threshold is set too small. It will not crash, but keep loading until the cache is full, empty the cache and repeat.
- The texturing has some swirling artifacts, which is due to a bug in the way the materials textures are applied.

Chapter 5

Results

This section will present the major results we achieved with our implementation of the algorithm developed in this project. The results will be presented as images, graphs and data as appropriate.

The presented results will not be analyzed in this chapter; this will be covered in chapter 6.

The collected screen shots will be shown in this chapter, but due to size constraints and printing quality they may be hard to analyze well. Because of that, the original high resolution images are included on the accompanying cd-rom. Also, some of the tests includes viewpoint movement and a movie of this movement as well as other movies have been captured and included on the cd-rom. See appendix B for the full contents of the cd-rom.

5.1 Test Configurations

The results were collected by running a series of tests on our implementation using the computers listed in table 5.1.

	Graphics card	CPU	RAM
PC1	NVIDIA Geforce 4 Ti 4200, 128 mb	AMD Athlon 750 MHz	768 mb
PC2	ATI Radeon 9700 Pro, 128 mb	AMD Athlon XP 1000 MHz	768 mb
PC3	ATI Radeon 9800 Pro, 128 mb	Intel Pentium 4 2.4 GHz	512 mb

Table 5.1: Configurations used to gather results.

5.2 Quality

To estimate the quality of our implementation a series of screen shots were captured.

5.2.1 Level-of-Detail Selection

In figure 5.1 a series of screen shots were taken from the same viewpoint, but with varying error threshold. Screen shots in the left and right column are the same, except on the right the scenes are shown in wireframe.

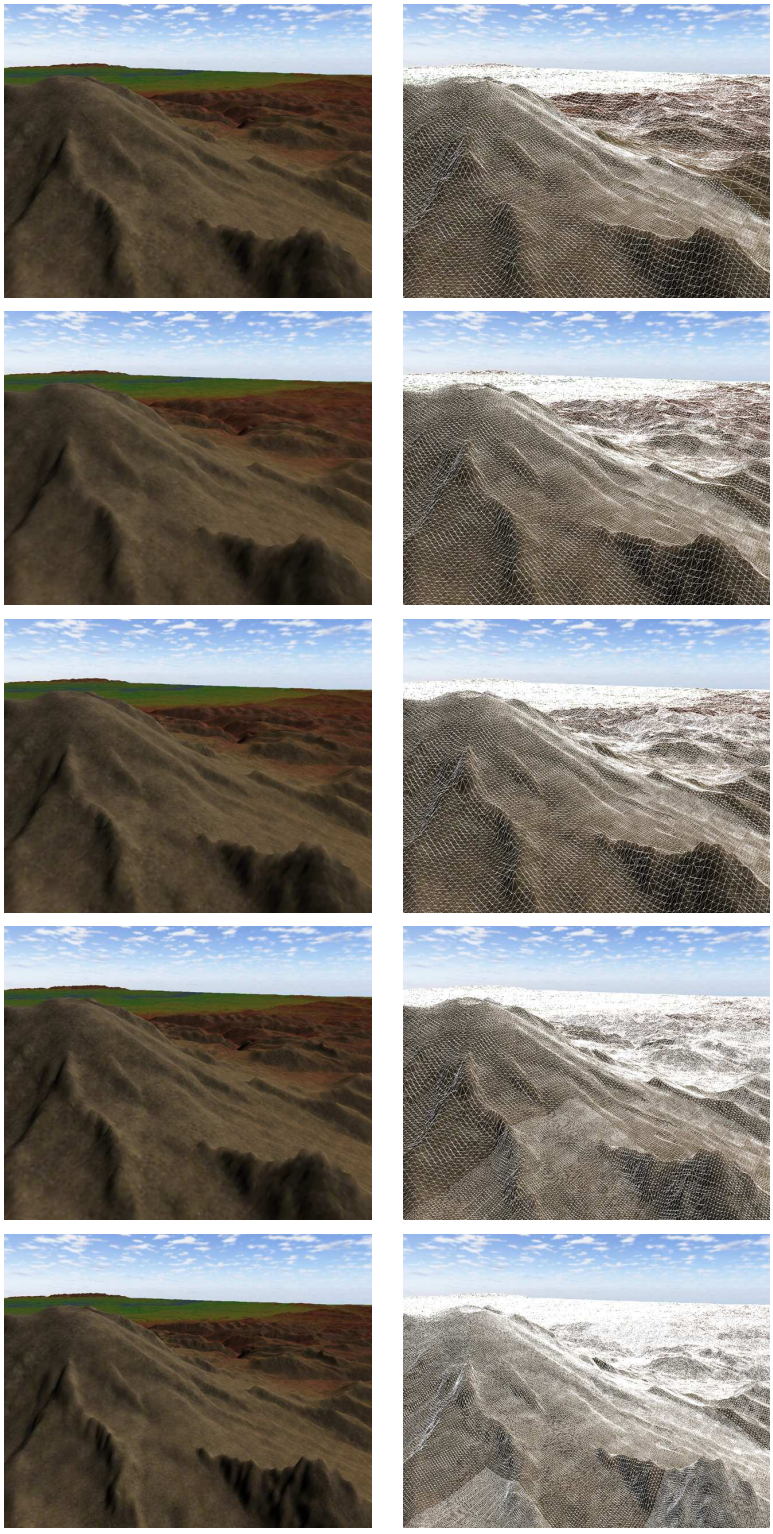

Figure 5.1: Series of screen shots with varying error threshold. From top to bottom the thresholds are 32, 16, 8, 4 and 2.

5.2.2 Subdivision and Detail Addition

To see the effect of the subdivision and detail addition, three screen shots were taken: one without any subdivision and detail addition, one with subdivision and one with both subdivision and detail addition. These shots, both normal and in wireframe, are shown in figure 5.2.

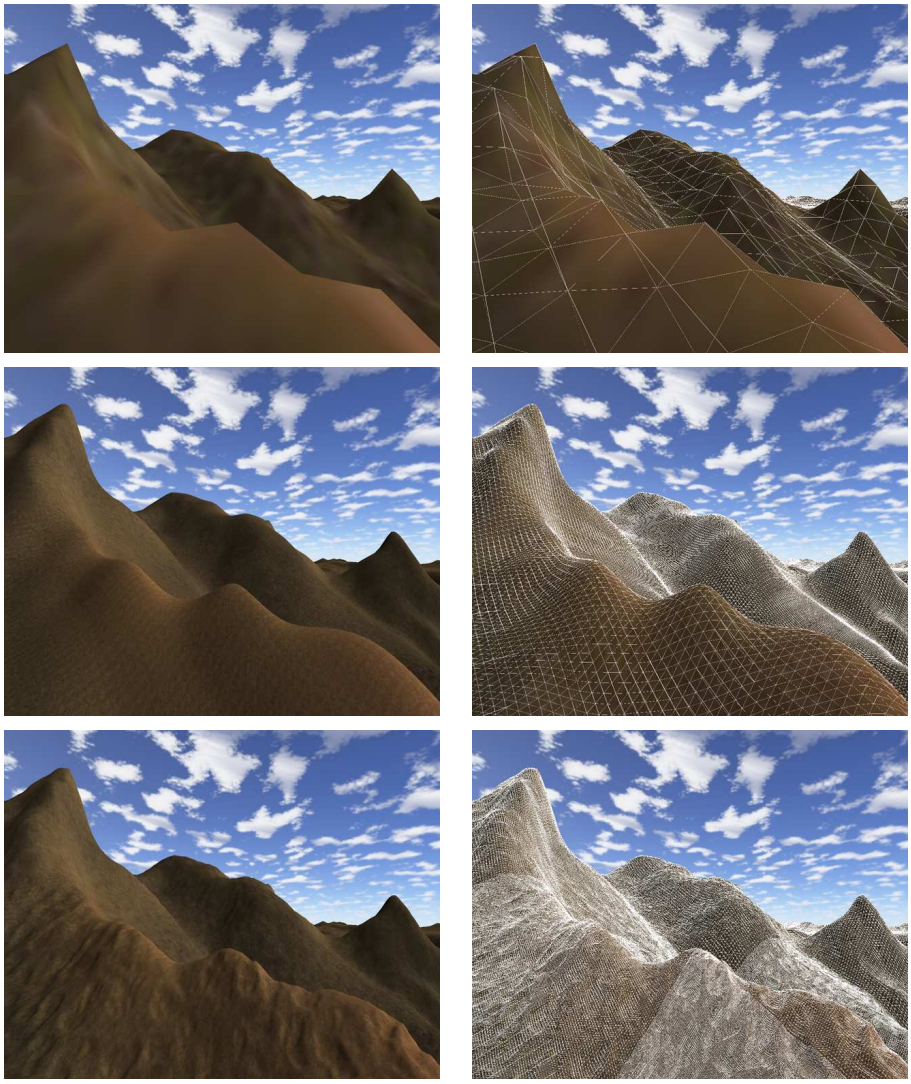

Figure 5.2: Screen shots show the effect of subdivision and detail addition. From top to bottom the screen shots show: no details or subdivision, subdivision only and both details and subdivision.

5.2.3 Materials

To see how the type of details that is applied to the terrain changes the appearance, two screen shots has been taken, as shown in figure 5.3. They share the same viewpoint but have different materials applied to the terrain.

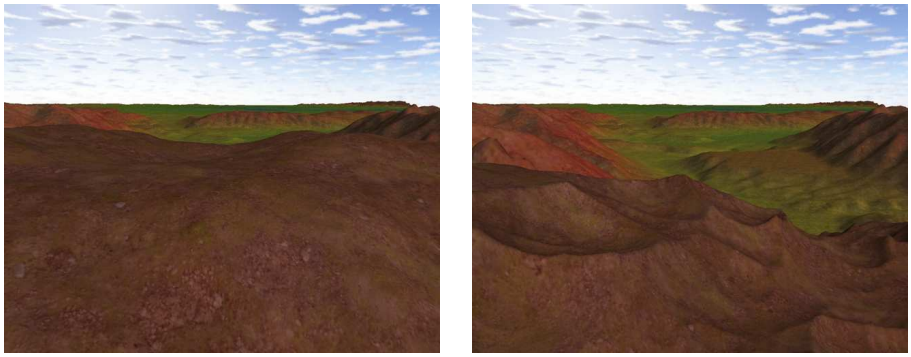

Figure 5.3: Screen shots show how the appearance of the terrain is affected by different materials.

5.2.4 Variable Detail Level

To evaluate the variable detail level compression scheme, two screen shot were taken from the same viewpoint: One with full detail level and one with variable detail level. These shots are shown in figure 5.4. The variable detail level is highest at the center of the terrain (around the viewpoint) and decreases linearly with distance.

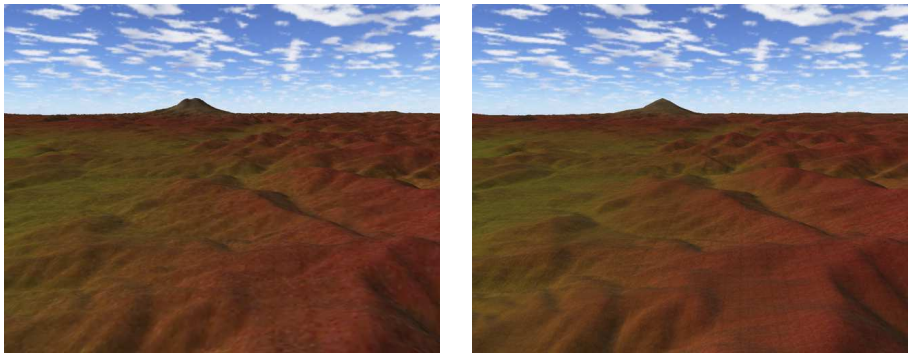

Figure 5.4: Screen shots that show the effect of variable detail level compression. The image on the left is with full detail level, the image on the right is compressed.

5.2.5 Artifacts

The two most visible artifacts of our method is a blocking artifact shown in figure 5.5 and a tiling artefact shown in figure 5.6.

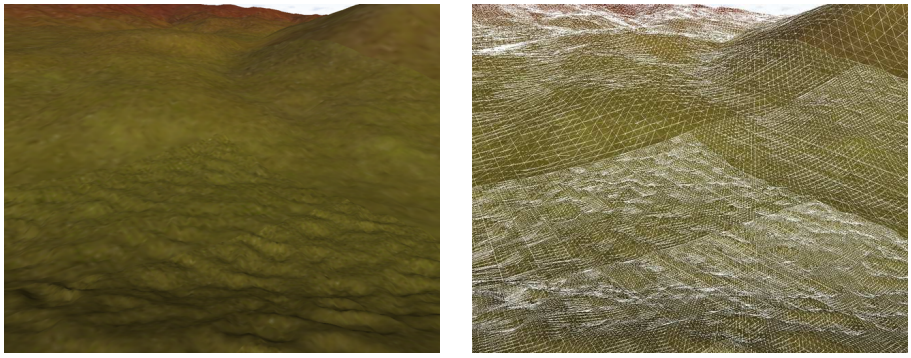

Figure 5.5: Screen shots showing a blocking artifact.

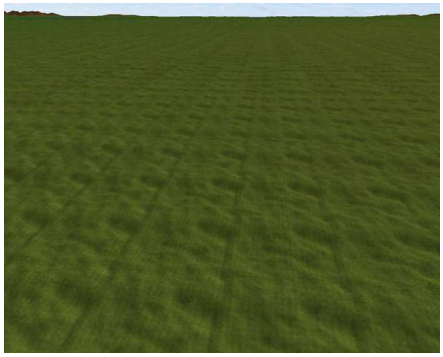

Figure 5.6: Screen shots showing a tiling artifact.

5.3 Performance

To measure the performance of our implementation a series of tests were conducted and performance statistics, either frame rate or polygon rate, were collected.

All tests were performed on all three test configurations. It is then possible to analyze the effects the differences, in the configurations, has on our implementation.

5.3.1 Fill and Transformation Limitation

To test for either fill or transformation limitations two tests were run. The first test rendered a scene with an average number of polygons, but at different display resolutions. This way the impact of increasing the amount of rendered pixel can be seen on the resulting frame rate. Figure 5.7 shows the result of this test.

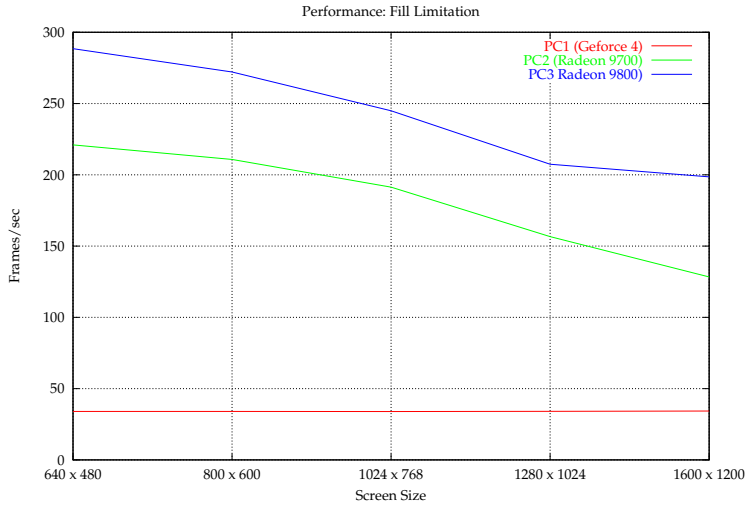

Figure 5.7: Difference in frame rate based on screen resolution.

The second test rendered a scene with an average, fixed display resolution but with varying amount of polygons. Thus, the effect of increasing the amount of polygons rendered can be extracted from the measured frame rate. Figure 5.8 shows the recorded measurements.

5.3.2 Vertex Cache Optimization

To measure the impact of our vertex cache optimization of our meshes, a series of test were run. These test renders a scene with a high polygon count several time, each time giving the mesh optimizer a different target vertex cache size. The measured polygon throughputs, shown in figure 5.9 shows the impacts of the optimizations.

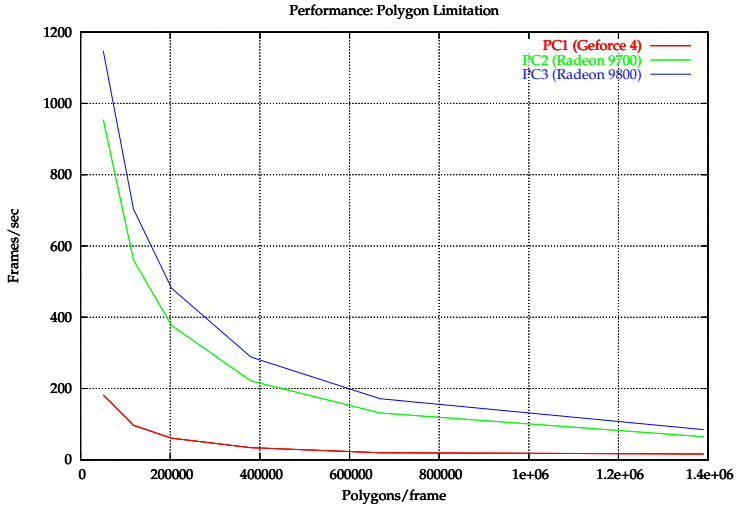

Figure 5.8: Difference in frame rate based on polygons per frames.

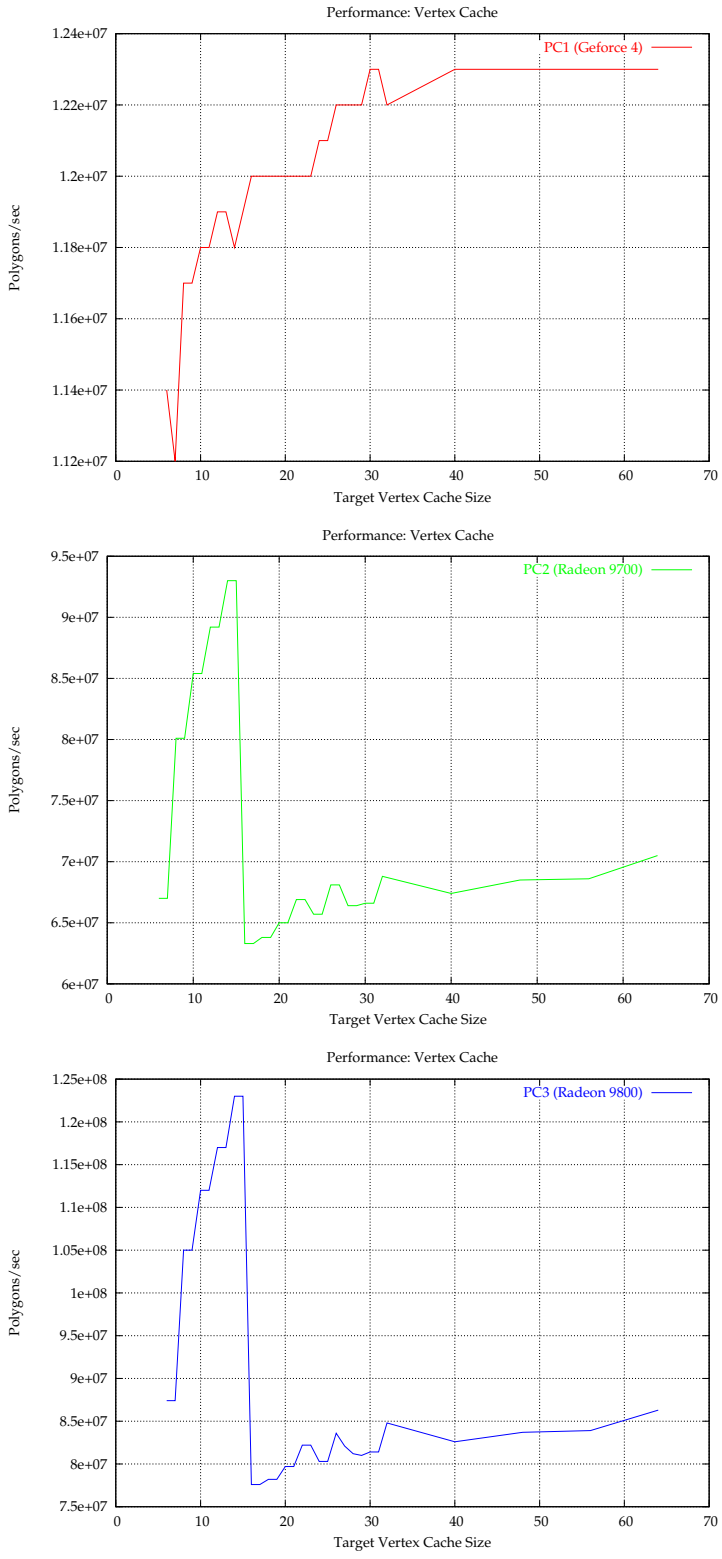

Figure 5.9: Polygon throughput based on target size of vertex cache.

5.3.3 Subdivision and Detail Addition

To measure the performance of our implementation when the viewpoint is moving and the impact of the subdivision and detail addition on this performance, a test were conducted. The test moves the viewpoint through the terrain in a predefined path and records the frame rate during the movement. The test is performed three times, first without subdivision and without detail addition, second with subdivision and finally with both subdivision and detail addition. In figure 5.10 the recorded frame rates is shown.

5.3.4 Performance of Memory APIs

Our implementation is able to use several different memory APIs and two graphics APIs. To measure the performance differences between these APIs - in the context of our implementation - two tests where performed.

Raw Polygon Throughput

To measure the raw polygon throughput differences between these APIs a test were conducted. The test renders the same high polygon count scene as used in the previous test and renders this with the four different API combinations our implementation is capable of. Figure 5.11 shows the measured performance.

Overall Frame Rate

To measure the effect the choice of API has in a more general way, a test has been performed which moves the viewpoint through the terrain while measuring the resulting frame rate. This has been performed for all API combination, as before and the result is shown in figure 5.12.

5.3.5 Chunk Sizes

The impact of the size of the chunks is found by measuring the polygon throughput when rendering a high polygon scene with different chunk sizes. In figure 5.13 the result of this test is shown[1].

5.4 Memory and Storage Consumption

To get an indication of the requirements of our implementation, besides the impact of graphics hardware as tested in the previous section, measurements and estimations of memory and storage consumption have been done.

[1] This test was not conducted on PC3, and PC2 did not support all chunk sizes.

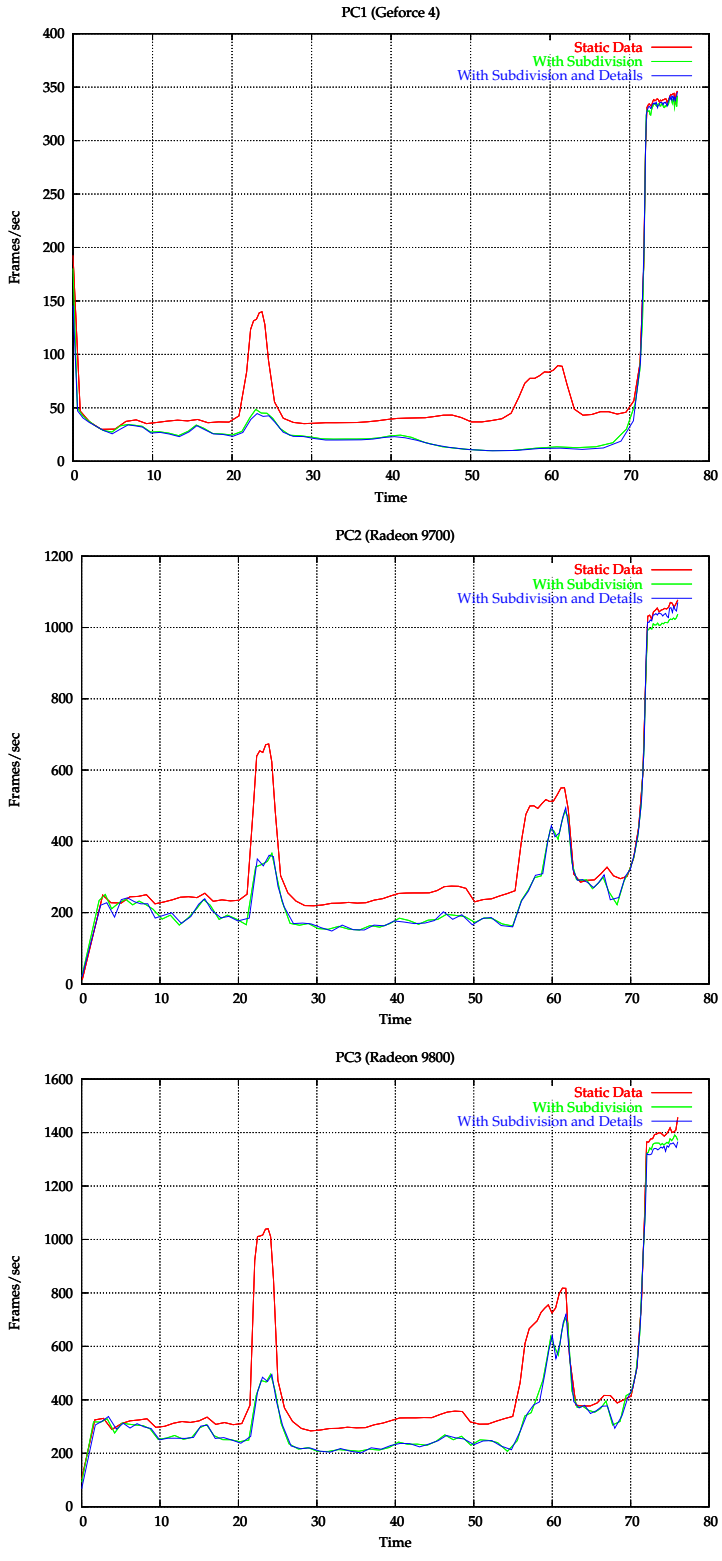

Figure 5.10: Difference in frame rate during camera movement with and without subdivision and detail generation.

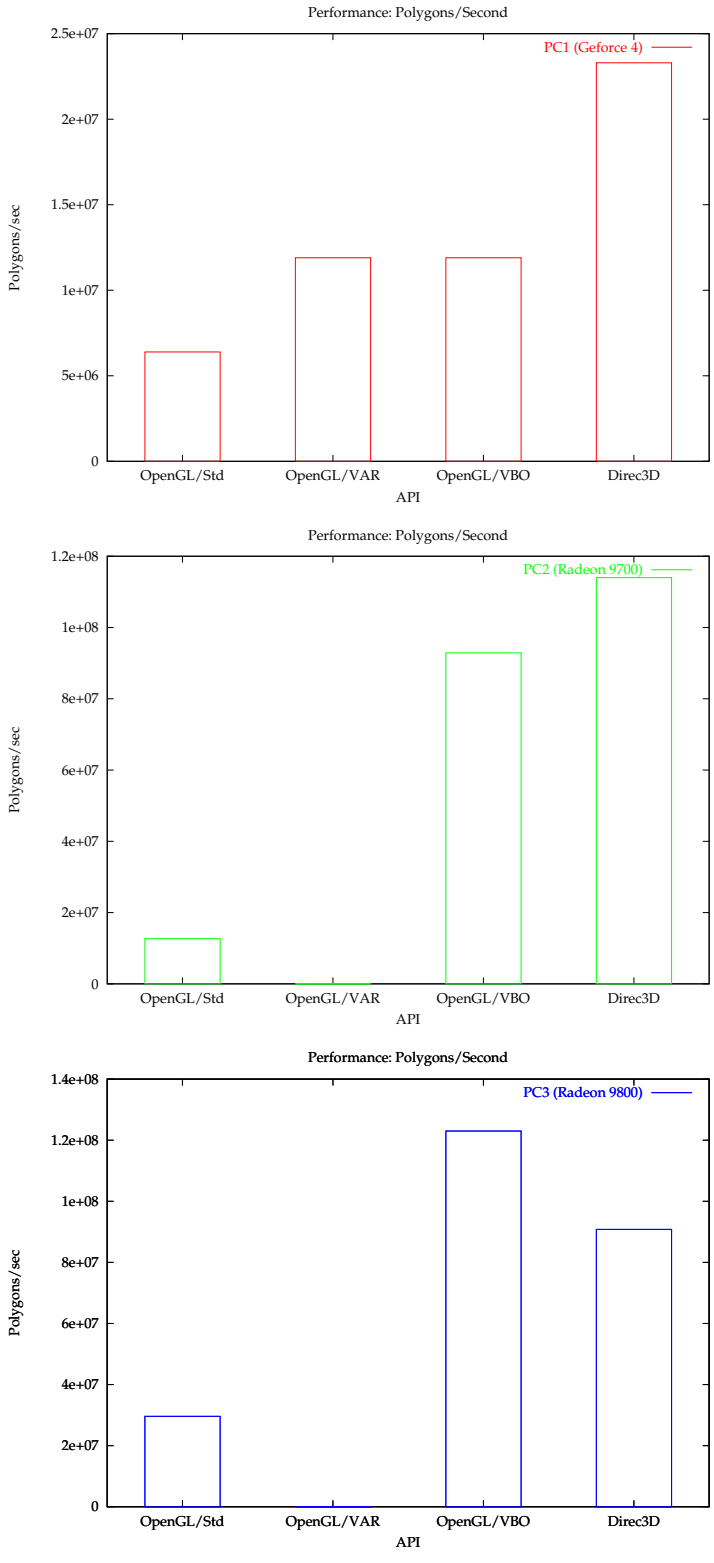

Figure 5.11: Polygon throughput based on memory API.

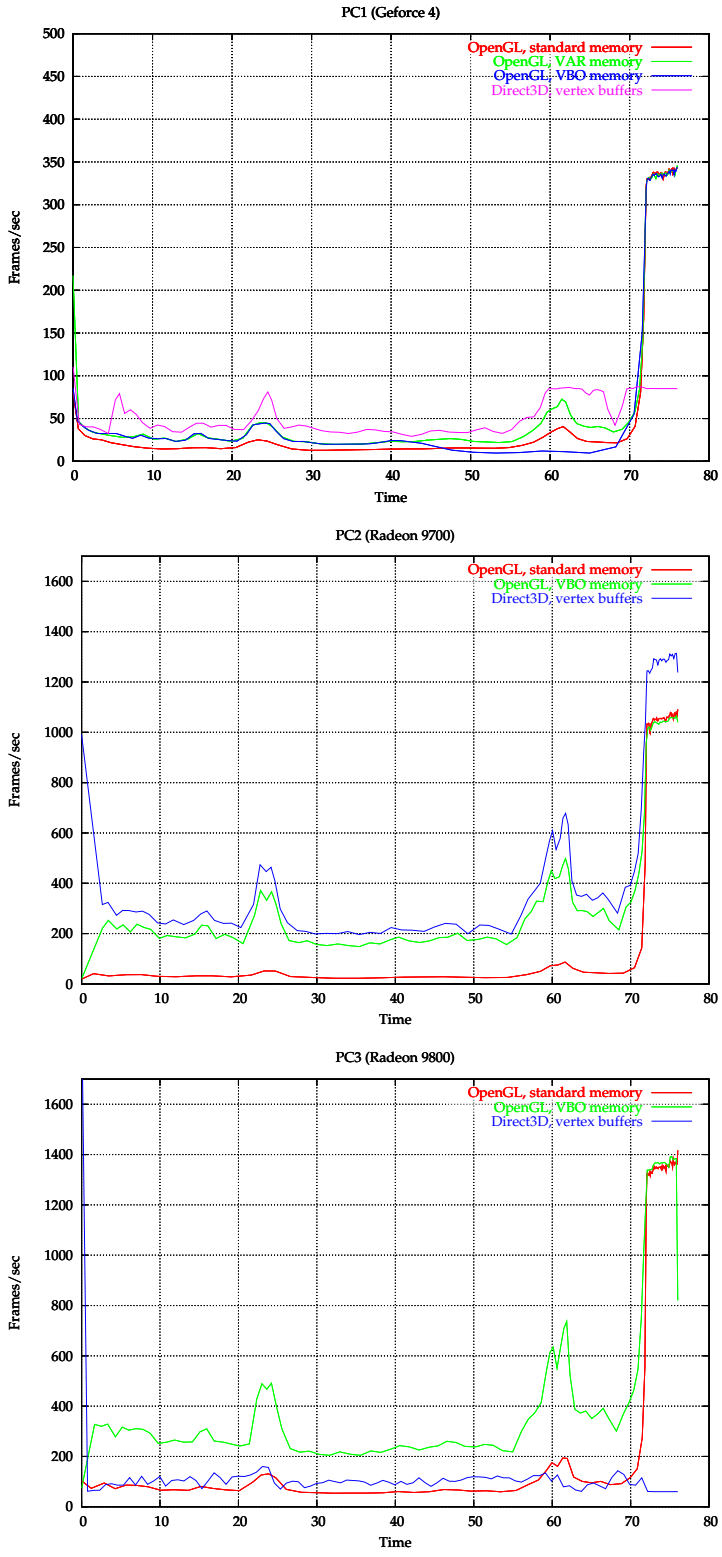

Figure 5.12: Difference in frame rate during camera movement based on API.

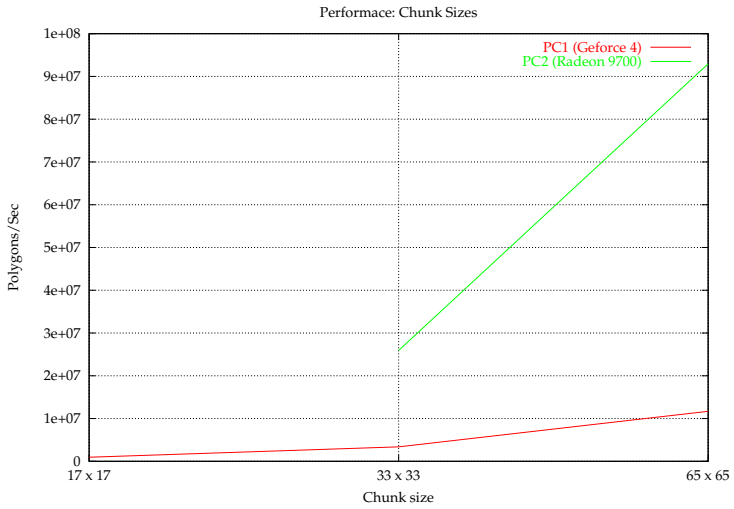

Figure 5.13: Difference in polygon throughput based on chunk size.

5.4.1 System Memory Consumption

Measuring exact system memory consumption is hard. The libraries used, such as OpenGL or the C++ standard libraries, may and will allocate memory without our knowledge. Besides, on a modern paged operating system the amount of memory allocated matters less, as long as the performance is acceptable.

What we can do, however, is determine that the majority of memory allocated directly by our implementation is the memory needed for each quad node. In table 5.2 the estimated memory consumption per quad node is shown. Some overhead may be assumed as well as memory indirectly tied to quad nodes may be required throughout our implementation.

Usage	**Bytes**
Quad node fields	72
Bounding box	40
Texture image fields	9
Texture image pixels	512^2
Height field fields	66
Height field samples	8978^3
Total	**9677**

Table 5.2: Estimated memory consumption of one quad node.

5.4.2 Storage Consumption

To measure hard drive storage consumption our preprocessor tool was used to generate a series of files with varying options. The options available are texture image size, texture image compression and chunk size.

All data is generated using the Puget Sound dataset[21].

Texture Settings

In table 5.3 and illustrated in figure 5.14 is the generated file sizes when varying texture image size and compression, but keeping chunk size constant.

Chunk Size	Texture Size	Texture Compression	File Size
64	16	yes	49.054 kb
64	16	no	53.832 kb
64	32	yes	51.101 kb
64	32	no	70.215 kb
64	64	yes	59.293 kb
64	64	no	135.747 kb
64	128	yes	92.059 kb
64	128	no	397.875 kb
64	256	yes	223.123 kb
64	256	no	1.446.387 kb

Table 5.3: Storage requirement based on texture settings.

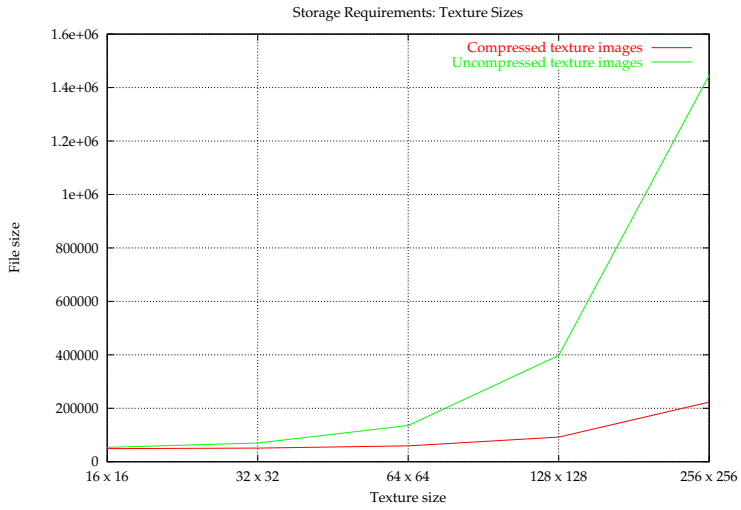

Figure 5.14: Storage requirement based on texture settings.

Chunk Size

When varying chunk size, but keeping texture settings constant, the output listed in table 5.4 is generated.

Chunk Size	Texture Size	Texture Compression	File Size
16	32	yes	141.609 kb
32	32	yes	65.152 kb
64	32	yes	51.101 kb

Table 5.4: Storage requirement based on texture settings.

Variable Detail Level

It is not possible to give exact numbers for how much variable detail level compresses a given terrain, as it solely depends on the weights used. As an example, the size terrain used for quality tests in section 5.2 is shown in table 5.5 along with an comparable uncompressed terrain.

Variable Detail Level	Chunk Size	Tex. Size	Tex. Comp.	File Size
No	64	32	yes	51.101 kb
Yes	64	32	yes	9.553 kb

Table 5.5: Example storage requirement with and without variable detail level.

Chapter 6

Discussion

In this chapter we will present and discuss what we believe is our methods strengths and weaknesses. We will sum up our results and experiences . Finally we will present our thoughts and ideas for future work based on our solution.

6.1 Analysis of Results

Based on the results presented in the last chapter, we will analyze our implementation of our method.

6.1.1 Quality

The screen shots captured in the previous chapter will be analyzed in this section.

Level-of-Detail Selection

The series of screen shots in figure 5.1 shows the different chunks selected for different error thresholds. As the error is directly linked to the minimum distance from the viewpoint to a chunk, the screen shots in effect shows what happens to the particular part of the terrain when the viewpoint approaches from afar.

To see which chunks are selected, the wireframe shots illustrate this best. At the same time, it is possible to see that between some of the screen shots the selection of chunks does not change, but the image appearance does. This is caused by the morphing, since they do have different error thresholds, but the difference is just not enough to cause other chunks to be selected.

The wireframe shots also shows that our implementation is capable of rendering many polygons, and as presented later in this chapter, quite efficiently.

Subdivision and Detail Addition

As seen from the screen shots in figure 5.2 the subdivision and addition of details has a tremendous effect on the terrain surface.

Without any subdivision and detail addition, the terrain contains very little details and is very edgy (The texture is also very blurred, but that is a side-effect of the way we have tied textures to the chunks).

When subdividing, the terrain surface becomes very smooth – too smooth to be called realistic in most cases, especially when viewed up close. But it forms a great base for adding details, as seen on the figure. With details added, the terrain surface appear highly detailed, although the underlying high field is actually not very detailed.

Materials

The screen shots in figure 5.3 shows that the details added to the terrain has a great affect on the appearance of the terrain surface. It only changes the appearance of those parts of the terrain that is so close, that the details are actually visible.

The two materials used differs only in the displacement function, as the detail texture is the same. Using other textures and other displacement functions can alter the terrain surface in numerous ways.

Variable Detail Level

As seen from the screen shots in figure 5.4 variable detail level does not necessarily change the appearance of the terrain much, even though the storage savings may be significant, as shown in section 5.4.2.

The mountain in the background shows the most significant difference between shots of the compressed and the non-compressed terrain. As the weight function used in the compression is really crude, this could be alleviated by tweaking the weight function and giving the area around the mountain a higher weight.

Artifacts

There are two main artifacts of our method, as seen in figure 5.5 and figure 5.6.

The blocking artifact shown in figure 5.5 results from the chunk based approach itself. It happens when the detail level of neighboring chunks are too far apart, i.e. one is much more detailed than the other. This appears, as seen on the screen shot, as edges or discontinuities through the terrain at the edges of the chunks. In the wireframe shot the difference in detail level of the chunks can be seen.

To alleviate this problem, a lower error threshold can be set, which at least pushes the problems further away from the viewpoint, making it less obvious. A more proper solution may be to use a morph value for each corner of the chunks and interpolate these values over the chunk. Then, one should making sure that the morph values of the neighboring corners of the neighboring chunks have the same morph values and then there will no longer be any discontinuities. This, however, will only work if there is at most one quadtree level between the chunks.

The tiling artifact is caused by the caching of the fractal displacement function. A combination of when the cached area of the function is too small and with some fractal settings, visible detail tiling appears. Possible solutions is to use a larger cache or other fractal settings. A better solution would be to fully implement materials, such that any one material did not cover a large enough area to make tiling visible.

6.1.2 Performance

The performance results gathered in the previous chapter will be analyzed in this section, test by test.

Fill and Transformation Limitation

The graph in figure 5.7 shows the frame rates dependance of screen resolution or fill rate.

Is shows that the frame rate is nearly constant on the NVIDIA Geforce 4 equipped PC1, regardless of screen resolution, indicating that on this platform our application is not fill limited at all.

PC2 and PC3, both equipped with ATI Radeon GPUs, behaves similarly, as they both drops approximately 30 percent in frame rate as screen resolution increases from 640×480 to 1600×1200. This indicates some dependency on fill rate and perhaps slight fill limitation.

Figure 5.8 shows the frame rate dependency of polygons rendered or transformation rate.

All three platforms behave similarly, degrading frame rate as polygon count increases. As polygon count for our terrain scenes lies between 200.000 and 600.000, it is interesting to see that the frame rate drops at least 50 percent from 200.000 to 600.000. This is clearly a must stronger dependency that the fill rate dependency shown earlier, indicating strong transformation limitation.

Thus, for our implementation to perform well, transformation rate is most important. This is not surprising, as we perform little work in the pixel

Vertex Cache Optimization

As seen in figure 5.9 the target size for our vertex cache optimization scheme has an effect on polygon throughput. On the ATI equipped machines, PC2 and PC3, a significant peak appears around a size of 15 entries. At the peak PC3 renders around 123 million polygons per second, while at the maximum target size, it renders around 86 million. Thus using a target cache size of 15 increases polygon throughput around 43 percent compared to using a target size of 65, which is the same as using strips as wide as the chunks.

The latter may be the most intuitive when seeking optimum performance, but this is certainly not the case, as seen.

PC1 does not seem to react the same way to the cache optimization, but just increase throughput as the size - and thus the resulting triangle strip width - increases. The difference between low target sizes and large target sizes is no great for sizes greater than 16. This is a bit odd since, according to the manufacturer, this hardware should be equipped with a vertex cache. It does not not seem to be enabled, however[1].

Subdivision and Detail Addition

The graphs in figure 5.10 shows the frame rate throughout a "tour" through the terrain. The shapes of the graphs are generally the same for all hardware, and the pattern is similar: when subdivision and details are disabled, frame rate is higher. The high peaks are in areas with few polygons, which again indicates that our implementation is transformation limited.

Frame rate is very similar with subdivision only and with subdivision and detail addition, indicating that it is not the CPU work of generating details that takes up time, but perhaps rather the transfer of data to the 3D API.

With subdivision and/or detail addition the peaks are delayed and a little smaller, or in one case not present. This indicates that, when the polygon count drops and the application should get some air, the detail creation thread still has work to do for a little while. This is a sign of CPU limitation and/or bandwidth limitation, in that new information has to be downloaded to the graphics hardware.

Performance Differences in APIs

Figure 5.11 shows the maximum polygon throughput measured for each API. It shows that Direct3D is significantly fastest on the Geforce hardware, slightly fastest on PC2 while significantly slower on PC3. This is a bit odd, since the

[1] During development early tests indicated that the vertex cache was in fact working, but stopped working after a certain driver revision. At the time of writing we have not succeeded in finding an old driver revision that enabled the vertex cache.

close relationship between the GPUs on PC2 and PC3. This is difference is possibly due to differences in the drivers.

The figure also indicates, unsurprisingly, that using standard vertex arrays is the slowest when using OpenGL. On the Geforce using the vertex array range (VAR) and the vertex buffer object (VBO) extension results in equal performance. The Radeon GPUs in PC2 and PC3 does not support the VAR extension

Also notably is that PC2 using Direct3D performs almost as well as PC3 using OpenGL/VBO.

Figure 5.12 shows the frame rate during the path through the terrain. The same differences between APIs are seen, indicating that the polygon throughput dictates the frame rate, again indicating transform limitation. Direct3D perform very poorly on PC3, but best on the other two machines. On PC1 it even has peaks where the OpenGL variants does not. This may indicate that memory transfer is faster in Direct3D, and that bandwidth probably is a limiting factor, as discussed earlier.

Chunk Sizes

As shown in 5.13 the size of the chunks have an large effect on polygon throughput and thus overall performance. The test was not run on PC3 but similar results is expected on that machine.

It shows that the larger the chunk size, the higher polygon throughput. This makes sense, as more polygons are drawn with less CPU intervention.

The size of 65×65[2] is the largest supported, the next possible size of 129×129 has too many polygons to be rendered in one batch in the 3D APIs.

For maximum performance the maximum chunk sizes of 65×65 should be used.

6.1.3 Memory Consumption

As listed in table table 5.2, the total memory consumption per quad node is 9677. This is mainly due to the heightfield samples and slightly to the texture image. The memory consumption per height sample is approximately 2.16 bytes of which the 2 bytes are the height sample itself.

Because of the way we employ out-of-core support, the maximum number of quad nodes in memory is configurable through the cache size. Having a reasonable cache size of 700 entries thus uses approximately 6.45 mb of memory for the quad nodes.

And as our implementation of out-of-core support supports memory mapping, the memory used for the texture images and height samples are in fact memory

[2] With skirts, actually 67×67.

mapped, meaning it may not take up as much physical RAM as calculated. That is completely in the hands of the operating system, however.

Thus our implementation uses very little system memory. However, many textures and many meshes are created through the 3D API which may use system memory to directly store the resources or copies thereof[3]. We have not been able to measure how much memory used by these libraries, unfortunately.

6.1.4 Storage Consumption

The numbers shown in table 5.3 shows that when textures are relatively small, their impact on storage size is negligible. There is virtually no difference between compressed 16×16 texture images and compressed 32×32 texture images and not that large a difference compared to compressed 64×64 texture images. The uncompressed textures quickly takes up space; the difference between uncompressed 16×16 texture images and uncompressed 32×32 texture images is relatively large. For larger texture images the difference is huge.

Thus, texture compression is a must if large textures is wanted, but is also a good choice in the general case since it also minimizes bandwidth requirements when reading the textures from disk and downloading them to the graphics API. The only drawback of compression is a slight loss in texture quality, but this is often covered by the detail textures.

Using chunk sizes less than the maximum 65×65 increases file size somewhat. Combined with the drastically reduces polygon throughput the optimal chunk size is 65×65.

When using a chunk size of 65×65 and compressed texture images of 32×32 the file size of 51.101 kb for the Puget Sound data set, which consists of 4097×4097 samples, gives a storage consumption of 3.1 bytes per height sample. Considering this is including textures and that the height sample itself is 2 bytes, this is a reasonably low number.

The variable detail level scheme makes significant storage savings possible. Obviously, it does lower the detail level in areas of the static height field, but if the viewpoint is kept far from these areas, it is not very visible.

On top of this, the storage requirement is only for the static part of the mesh. Combined with the detail system, the static mesh does not need a very high sample density, yet the terrain will still appear very detailed. This makes room for very large, very detailed terrains with very low storage consumption.

6.1.5 Performance Summary

As analyzed in the previous sections, our implementation performs very well in term of raw polygon throughput. It is, however, still transformation limited. As

[3]Copies may be needed, if the resources resides on the graphics hardware which can be lost.

all our transformation work is done on the GPU, the limiting factor must be the vertex processer on the graphics hardware and/or the bandwidth used by vertex data. Thus, the performance of our level-of-detail algorithm depends mostly on the graphics hardware and thus should scale well with faster hardware.

This is as expected and as planned. By design, we try to do as little CPU work as possible. We also tries to maximize polygon throughput as much as possible by utilizing the vertex cache and by keeping meshes, once created, untouched and preferably in video memory.

Fill rate seems a lesser problem, which is not surprising, since we render front-to-back, which saves fill rate - and since our method in general requires little pixel processing.

The detail generation system seems CPU limited and bandwidth limited, which again is not surprising, since it requires some amount of work to generate detail chunks and some amount bandwidth to download these to the graphics hardware. However, it is not desirable.

In our implementation the detail synthesis is performed in a separate thread, so a multiprocessor system or a processor with hyper-threading may minimize or eliminate this limitation. Unfortunately we were not able to test our implementation on such a system.

At the same time, the detail synthesis could probably be optimized further, but we have not pursued this further.

A small change in our implementation is possible which halves the bandwidth required for transferring meshes. Now, we transfer height values as one floating point value per sample, but we could instead transfer each sample compressed into one short, just as the samples are stored on disk. Unfortunately, this was not possible for us to do in a way compatible with both NVIDIA and ATI hardware. A work-around may be possible, but we did not investigate this further.

6.1.6 Future Work

Even though we believe our method is usable as-is, there is of course always room for improvements.

We think the use of multiple material is important and it needs to be implemented and researched further, to understand its benefits and shortcomings. Doing this would make our method more complete and useful.

As much of the non-GPU part of the work done is detail synthesis, it would probably worth investigating if a more intelligent way to decide when new chunks should be created and which to create, maybe saving some CPU work. Also better and faster ways of creating details may be interesting.

The error estimation for the detail chunks works well in most cases, although it is crude. To fully understand if this is a reasonable choice this has to be

investigated more. The benefit of a better error estimation of the detail chunks would be a better estimation of the resulting screen space error. This could lead to lower polygon requirements as well as a possible reduction of the blocking artifact.

The block-based morphing artifacts could possibly be removed. As mentioned, a simple solution of using morph factors for each corner instead of for each chunk might work, but this has to be investigated further.

Finally, it could be interesting to extend the terrain level-of-detail system with the ability to place objects, such as trees or buildings, in the terrain. Maybe it is possible to combine object level-of-detail with terrain level-of-detail giving a simple, powerful but complete level-of-detail system for outdoor scenes.

Chapter 7

Conclusion

As presented in the previous chapters, we have not invented a new paradigm of terrain rendering. We have rather merged the two currently most popular and highest performing algorithms into what turns out to be an effective combination.

We can state that using the framework of [20] for our level-of-detail algorithm was a good choice. It is simple to understand and implement, yet very efficient. Also, using the simplification scheme of [3] was also a good choice. It is a fast and simple way to simplify terrain meshes, but it does not simplify that well. To achieve the same error as a more sophisticated simplification scheme, more polygons are needed. But, because of the high polygon throughput possible with our method, the higher polygon requirement is not really a problem. Also, when used in conjunction with the quadtree structure, great memory and performance optimizations are possible.

To sum up, the main accomplishments of our method are:

- The algorithm is simple.
- Performance is very good and scales well with the graphics hardware.
- Quality of renderings are high.
- Storage requirements are low.

When extending the level-of-detail algorithm with the detail synthesis system, we get a system that is capable of rendering terrains efficiently and with high image quality. By splitting the terrain representation into a static mesh and runtime generated details we are able to store these terrains with low storage requirements, yet still render them with very high amount of details. Thus we are able to render very large, very detailed terrains!

There method produces a few artifacts, but these can be avoided by a little care, and simple solutions to eliminate these artifacts seems to exist.

Appendix A

Configuration Options

The options in the config file are specified in the file as:

```
option = value
```

All options are listed and explained below.

map_file `[String]` Specifies which map file to use. This is overwritten if another file is specified as parameter to TEngine.

lod_error `[Float]` This specifies the value of τ used in the level-of-detail selection algorithm.

lod_minimum_spacing `[Float]` Sets the minimum spacing between samples. This is useful to limit the addition of details.

dynamic_texture_size `[Integer]` Specifies the width and height of the textures where diffuse lighting is burnt into.

vertex_cache_size `[Integer]` Sets the size of the vertex cache to optimize for.

quad_cache_size `[Integer]` Specifies how many nodes can be stored in the node cache. The higher the better but this is limited by the amount of memory available on the graphics hardware.

cg_profile `[String]` Specifies the CG profile. Valid entries are: "arb", "nv20", "nv30".

reader `[String]` This specifies how the map file is read. Valid entries are: "stream", for ordinary streaming of the file and "memory_mapped" for memory mapping of the file.

render `[String]` Specifies which 3D API is to be used. Valid entries are: "gl" and "dx".

memory `[String]` Selects which memory allocation scheme should be used by the 3D API. If "gl" was selected as render, then valid entries are: "gl_std", for system memory function, "gl_var" for using the NVIDIA VAR extensions and "gl_vbo" for using the ARB VBO extension.

camera_fov `[Float]` The field of view of the camera, specified in degrees.

camera_z_near `[Float]` The distance to the near clipping plane of the camera.

camera_z_far `[Float]` The distance to the far clipping plane of the camera.

detail_create_new_nodes `[Boolean]` Enables or Disables the creation of detail nodes.

detail_subdivision_factor `[Float]` Specifies the ω[1] value used during subdivision.

detail_add_material `[Boolean]` Enables or disables the addition of details to the subdivided terrain.

detail_material_type `[Integer]` Selects which material to use. The current implementation only has to materials: 0 or 1.

camera_path `[String]` A camera path file, used to control the automatic camera movements.

draw_user_interface `[Boolean]` Enables or disables to user interface. Enabling the user interface can hurt performance.

log_file `[String]` Specifies where to store the log file.

demo_mode `[Boolean]` Select if the render move around the terrain, using the specified camera path.

video_mode `[Boolean]` Sets the render in recording mode. The camera moves using the specified camera path and the frames are stored on disk. This mode is only supported with the "dx" render. "demo_mode" overrides this mode.

frames_per_second `[Integer]` Specifies how many frames per second should be rendered during "video_mode".

image_path `[String]` Specifies where to store the images produced during "video_mode".

[1] Described in [11]. 0 for linear interpolation, 1 for regular subdivision.

Appendix B

Contents of the CD-ROM

The contents of the accompanying CD-ROM is listed in this appendix.

B.1 Images

In the folder `images` all the original images gathered in chapter 5 and analyzed in chapter 6 are stored. The images are stored in the subfolders described below.

lod_selection Images for Level-of-Detail Selection, described in section 5.2.1.

details Images for Subdivision and Detail Addition, described in section 5.2.2

materials Images for Materials, described in section 5.2.3

variable_depth Images for Variable Detail Level, described in 5.2.4

artifacts Images for Artifacts, described in 5.2.5

B.2 Videos

The contents of the folder `videos` are a few videos, captured while running he implementation. The videos are in AVI/DIVX-format[1]. The videos are as follows:

puget_sound.avi Movie of tour through the Puget Sound data set used for gathering some of the results in chapter 5.

no_details.avi A tour through a small test terrain, shown without subdivision and without details.

[1] Free codec for this movie format can be downloaded from http://www.divx.com/

only_subdivision.avi The tour through the small test terrain, shown with subdivision but without details.

full_details.avi The tour through the small test terrain, shown with subdivision and with details.

B.3 Demos

The folder `demos` contains executables of our implementation as well as a small set of demos. The batch file `demo.bat` runs all demos.

Remember that DirectX 9.0 needs to be installed in order to run the demo.

B.4 Data Sets

In the folder `data_sets` the source data for the Puget Sound [21] terrain is stored in PNG-format:

ps_height_4k.png The elevation values.

ps_texture_4k.png The texture data.

B.5 Source Code

The source code for our implementation is located in the folder `source`. The main part of our source is located in the `source` folder, and the subfolders contains the following:

CGLA Part of CGLA.

Common Part of CGLA.

Components Part of CGLA.

Graphics Part of CGLA.

libpng Source files for pnglib.

zlib Source files for zlib.

projects Project files for Visual Studio.

postscript Contains all our source as a postscript file, ready to print. Note: this document is 263 pages long.

To open the project in Visual C++, open the solution located in `source`
`projects`
`vc7`
`vc7.sln`.

Bibliography

[1] Tomas Akenine-Möller and Eric Haines. *Real-time Rendering*. A K PETERS, LTD., second edition, 2002.

[2] Andreas Bærentzen. Cgla. World Wide Web. http://www.imm.dtu.dk/ jab/software.html.

[3] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. World Wide Web, October 2000. http://www.flipcode.com/tutorials/geomipmaps.pdf.

[4] Mark Duchaineau. Roam algorithm version 2.0 – work in progress. World Wide Web. http://www.cognigraph.com/ROAM_homepage/ROAM2/.

[5] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[6] David S. Ebert, F. Kenton Musgrave, Darwyn Peachy, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[7] Johan Hammes. Modeling of ecosystems as a data source for real-time terrain rendering. World Wide Web, 200. http://mzone.mweb.co.za/residents/jhammes/nineteen70five/hammes.pdf.

[8] H. Hoppe. Progressive meshes. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 99–108, 1996.

[9] H. Hoppe. View-dependent refinement of progressive meshes. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 189–198, 1997.

[10] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE visualization*, pages 35–42, October 1998.

[11] L. Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In *Computer Graphics Forum (Proc. EUROGRAPHICS '96), 15(3)*, pages 409–420, 1996.

[12] Bent D. Larsen and Niels J. Christensen. Real-time terrain rendering using smooth hardware otimized level of detail. *Journal of WSCG*, 11(1), February 2003.

[13] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, pages 109–117, 1996.

[14] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. *Visualization, 2001 VIS'01. Proceedings*, pages 363–370, 2001.

[15] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *Visualization and Computer Graphics, IEEE Transactions on*, pages 239–254, 2002.

[16] Jean loup Gailly and Mark Adler. zlib. World Wide Web. http://www.gzip.org/zlib/.

[17] Douglas H. Rogers. Vertex cache optimization. World Wide Web, 1999. http://developer.nvidia.com/view.asp?IO=vertex_cache_opt.

[18] Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson, et al. libpng. World Wide Web. http://www.libpng.org.

[19] A. J. Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, /1998.

[20] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. World Wide Web, April 2002. http://www.tulrich.com.

[21] USGS and The University of Washington. Puget sound terrain. World Wide Web. http://www.cc.gatech.edu/projects/large_models/ps.html.