Multiresolution Triangulations in Terrain Modelling

Jon Mugaas

Cand Scient Thesis

October 2002
Preface

This thesis documents my work on the Cand. Scient. degree in Computer Science at the University of Oslo, faculty of mathematics and natural sciences, department of informatics, direction of Computational Mathematics.

First I would like to thank my supervisor Rune Aasgaard at SINTEF, Applied Mathematics, for guiding me throughout this period. I would also like to thank Thomas Engh Sevaldrud for supervising me the first months, and my internal supervisor Tom Lyche for useful feedback on the writing of this thesis.

Everyone at SINTEF, GIT has contributed to a pleasant working environment. Special thanks goes to Sverre Jonassen for all his help, and for helping me with the installation of uncooperative libraries in linux.

Further I would like to thank Rasmus Bording, and Per Ådne Mugaas for comments. I will especially like to thank Ellen Skjold Kvåle for proofreading and valuable feedback.

Jon Mugaas

October 2002
Contents

1 Introduction 1
  1.1 Thesis Layout .................................. 3

2 Triangulations 4
  2.1 Using Triangulations to Model Terrains .............. 5
  2.2 Delaunay Triangulations .......................... 7
  2.3 Data Dependent Triangulations ........................ 7
  2.4 Memory Requirements .............................. 8
    2.4.1 A Regular Grid Triangulation ..................... 9
  2.5 Multiresolution triangulations ...................... 9
    2.5.1 RIT triangulation ............................. 11

3 Some Common Algorithms Used in Multiresolution Terrain
  Visualization 14
  3.1 Tile Based Multiresolution Terrain .................. 14
    3.1.1 Extending the Tile Based Terrain Model .......... 16
  3.2 Progressive Meshes ................................ 16
    3.2.1 View Dependent Progressive Mesh ................ 16
    3.2.2 Using a View Dependent Progressive Mesh for Terrain
      Visualization ................................... 18
  3.3 Multi-Triangulations ................................ 19
  3.4 Multiresolution Right Angle Isosceles Triangles ...... 21
    3.4.1 Data Structures ................................ 22
    3.4.2 Data Structure Traversal ....................... 25
  3.5 Closing Remarks .................................. 29

4 Memory Handling 32
  4.1 Interleaved Quad Trees ............................ 32
  4.2 Hierarchical Z-order Indexing ........................ 33
    4.2.1 Z-order Indexing ............................... 34
    4.2.2 Binary tree Indexing ......................... 36
4.2.3 Putting it Together ........................................ 36

5 The Test Program .............................................. 39
  5.1 Tile Based RIT Multiresolution Triangulation ............. 40
  5.2 Implementation ........................................... 41
  5.3 The Preprocessor ........................................... 42
     5.3.1 Read Data ........................................... 42
     5.3.2 Compute World Space Errors, and Bounding Spheres .. 42
     5.3.3 Sorting and Storing Vertices .......................... 44
  5.4 The Renderer ............................................... 45
     5.4.1 Triangle Evaluation ................................... 46
     5.4.2 Vertex Buffer ......................................... 54

6 Results .......................................................... 55
  6.1 The Datasets ................................................ 55
  6.2 The Fly-through ........................................... 57
  6.3 The Tests .................................................... 57
     6.3.1 Conclusion ............................................ 59
  6.4 Linear Indexing vs. Hierarchical Z-Indexing ................ 61
     6.4.1 Conclusion ............................................ 61
  6.5 Testing the World Space Error Metrics ...................... 63
     6.5.1 Conclusion ............................................ 63

7 Conclusion ....................................................... 65

A Screen Shots .................................................... 67

B Tables With Results From Fly-Throughs ....................... 72
List of Figures

2.1 a) A set of points. b) A triangulation of the points in A. c) Space expanded to $R^3$ by adding elevation value to each vertex in the triangulation in B. ........................................ 5
2.2 a) Not a Delaunay triangulation, b) A Delaunay triangulation 7
2.3 A triangle strip to the left, and a triangle fan to the right. .. 8
2.4 A Regular mesh to the left, and an arbitrary mesh to the right. 9
2.5 A multi resolution triangulation ................................. 10
2.6 a) A RIT triangulation.
   b) Diamonds
   c)Refinement operator one and two .......................... 12
3.1 a) A tile consists of a triangulations in the center and trans-
sition skirts at the edges. b) Several triangulations and trans-
sition skirts are used for each tile. c) When the right tile is
refined, the right transition skirt on the left tile has to be re-
frined to maintain a valid triangulation. The shaded triangles
are the triangles used in the transition skirts ................... 15
3.2 a) The two geometry operators of a progressive mesh: the
vertex split, and the edge collapse. Both operators depend on
the shaded triangles. b) $v_s$ is stored as the parent of $v_u$ and $v_t$
in a vertex tree. c)The vertex tree. $M_0$ is the base mesh. $M^*$
is the vertex front. ............................................ 17
3.3 The creation of a hierarchical progressive mesh. .......... 18
3.4 A Multi Triangulation. ............................... 19
3.5 Left: RIT base mesh on a regular grid, Right: a selectively
refined RIT triangulation on a regular grid. ..................... 21
3.6 The binary triangle tree .................................. 22
3.7 The vertex tree ......................................... 23
3.8 The triangulation after the first four steps in a triangle tree
traversal, from the root and up. ............................ 25
4.1 Interleaving the coordinates to a z-order index by using a 16-bit lookup table .......................... 34
4.2 The result of z-order indexing. ............................... 34
4.3 The indices are sorted by ordinary z-order indexing. All numbers in this figure are expressed in base four. The different Z-levels are indicated. .............................. 35
4.4 The base indices increasing from left to right sorted in a binary tree ................................. 37
4.5 a) The final index is a combination of two binary trees and a z-order indexing field, b) The resulting indices in a 9x9 grid. c) The data elements are stored in a single array ........................ 37
5.1 The vertex tree expands outside a tile during the preprocessing .............................. 41
5.2 The Preprocessor ........................................ 42
5.3 A class diagram, showing the structure of the renderer .......................... 45
5.4 The functions used to traverse a vertex tree during rendering. .......................... 47
5.5 a) The order of the sub triangles created by vertices inserted at an odd level. Center: The order of the sub triangles created at an even level. Right: The resulting triangle strip after running the methods in 5.4 with a vertex tree. ........... 48
5.6 Pseudocode for the function evalTriangle() ........................................ 49
5.7 Projection of a world space δ-value to a screen space ε .......................... 51
5.8 Pseudocode for view frustum culling of vertices in a vertex tree .......................... 52
6.1 The dataset of the Grand Canyon, with the three base meshes used. The left model uses a 2x1 tiles, the middle 8x4 tiles and the right model uses 32x16 tiles. ............................... 56
6.2 The dataset of south Norway, with the two base meshes used. The left base mesh uses a 4x6 tiles, and the right base mesh uses 16x24 tiles. ............................... 56
6.3 Graphs showing renderer performance with the models from Grand Canyon. Dotted lines represent models using a texture with resolution 8192x4096 pixels. Solid lines represent results from models using a texture with resolution 4096x2048. ......... 58
6.4 Graphs showing rendering performance with the models from the south of Norway. .............................. 60
6.5 These graphs show the difference in performance when using linear indexing and Hierarchical Z indexing. Each graph shows the Hierarchical Z index’ performance in percentage of the linear index’ performance. ............................... 62
6.6 This figure shows the average number of vertices used per frame in a fly-through for two models using two different world space error estimates. These estimates are described in section 5.3.2.

A.1 Screen shots over Grand Canyon. Both screen shots are taken from the same view, using the same geometry.

A.2 Screen shots over Grand Canyon. The top picture uses the same geometry as both the scenes rendered in figure A.1. The bottom picture shows the base mesh from the same view.

A.3 Screen shots using the data model from the south of Norway.

A.4 A screen shot from the Grand Canyon data model.
Chapter 1

Introduction

Within the field of Geographic Information Technology (GIT), it is a common problem to visualize large three-dimensional terrain models. Visualizing such a model without detail reduction can be a computationally intensive task. If one would want an interactive visualization of the terrain model, it is preferable that it is possible to move the viewpoint through the terrain in real time. For a smooth visualization, this requires fast rendering. When visualizing a terrain from a given camera position, the original terrain model may be simplified with no visible difference in the rendered screen. Some parts of the model will be outside the view frustum, and do not contribute to the rendered screen. Other parts of the model are far away from the camera and may be unnecessarily accurate. By reducing the superfluous details in the model from a given view, the number of computations needed to render the view may be reduced considerably, without noticeable effect on the resulting rendered picture. By using efficient algorithms and data structures, interactive real time visualizations of large and detailed terrain models are now possible on a regular home computer.

Real time visualization of terrain models by using a variable level of detail has been an active field of research over the last few years, and several methods of achieving this have been proposed. This thesis focuses on terrain models made with multiresolution triangulations. That is, methods for creating triangulations with a variable level of detail. A more thorough definition of triangulations and multiresolution triangulations is given in chapter 2.

I will present some methods developed to visualize terrains with multiresolution triangulations. With a base in one of them, a terrain visualization
application has been developed. This application is a modification of the algorithm presented in “Visualization of large terrains made easy” by Peter Lindstrom and Valerio Pascucci [2]. Their results show that their method is one of the fastest terrain visualization methods published. Our version of their algorithm is modified to handle several tiles of multiresolution triangulations. Using tiles adds flexibility to the underlying data dimensions and texturing. I have tested how the method performs, and how tiling affects the rendering speed.

The handling of textures may be very important in terrain visualization. If large textures are used, managing of the textures may greatly affect the rendering speed. The terrain rendering application developed uses textures when rendering. However, the emphasis in this thesis has been put on the geometry in multiresolution terrain models, and not the texturing of such models.

All methods used to visualize terrains described in this thesis treat the terrain as a surface over a two-dimensional domain. The surface may be seen as a function $F : \mathbb{R}^2 \rightarrow \mathbb{R}$. In words, a two-dimensional coordinate returns the terrain elevation. This somewhat limits the terrain model. Each coordinate can only return one elevation value, hence vertical mountain walls and overhangs cannot be modeled. If the model samples the terrain regularly in the plane, the steep parts of a terrain will be less accurately modeled than the more flat areas.

Other functions can also be used to model terrains. By using a parametric function $F : \mathbb{R}^2 \rightarrow \mathbb{R}^3$, a more flexible surface is modeled. Overhangs could be modeled, and steep terrains may be modeled more accurately. Other representations could also be used. Seismic data can create data in a three dimensional domain, $F : \mathbb{R}^3 \rightarrow \mathbb{R}$. With this function one would be able to model the ground by for instance using isosurfaces, or voxel graphics. Such methods are often used in the petrol industry to locate oil and gas reservoirs. The functions described in this paragraph are, however, beyond the scope of this thesis.
1.1 Thesis Layout

This thesis is divided into the following chapters:

Chapter 1 This chapter.

Chapter 2 Contains notations and definitions used in the rest of this thesis. An introduction to triangulations is given. A selection of different types of triangulations are also presented.

Chapter 3 Describes some common algorithms used for terrain modeling. The methods presented are a tile based method, Progressive meshes, Multi Triangulation, and several methods which use division of right angle isosceles triangles.

Chapter 4 Describes methods for memory handling in large terrain models when using multiresolution RIT triangulations.

Chapter 5 Describes my implementation of a terrain visualization application.

Chapter 6 Shows the results of the implemented terrain visualization application.

Chapter 7 Conclusion
Chapter 2

Triangulations

In this section I will introduce some of the notation used throughout this thesis. I will also give a definition of a valid triangulation and present some common types of triangulations, and explain their advantages and drawbacks. As the terrain models studied in this thesis all use terrain models described by functions as $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, this section is restricted to triangulations in $\mathbb{R}^2$.

A corner in a triangle is called a vertex. A vertex is denoted as $v$. A triangle $\tau_{i,j,k}$ is described by its three corner vertices $v_i, v_j, v_k$, and it is the convex hull of these three vertices.

A triangulation of a point set $P$, is a set of triangles which uses all of the points in $P$ as vertices. A triangulation is denoted $T$.

Obviously there are several possible triangulations of a set of points. Some of these are unsuitable for any practical purpose, therefore I will now give a definition of a valid triangulation. This definition is similar to the definition given in [12], but I have made some modifications.

**Definition 2.0.1 (Valid triangulation).** $P$ is a set of points in $\mathbb{R}^2$. $\Omega$ is the convex hull of all the points in $P$. A triangulation $T$ of the points in $P$ is valid if the following requirements are met:

1. No triangle $\tau_{i,j,k} \in T$ may be degenerate. That is, the vertices $v_i, v_j, v_k$ defining the triangle $\tau_{i,j,k} \in T$ cannot be colinear.

2. The interior of any two triangles do not intersect, that is, if $\tau_{i,j,k} \in T$
and $\tau_{\alpha,\beta,\gamma} \in T$ with $(i, j, k) \neq (\alpha, \beta, \gamma)$ then

$$\text{Int}(\tau_{i,j,k}) \cap \text{Int}(\tau_{\alpha,\beta,\gamma}) = \emptyset$$

3. The boundary of two triangles can only intersect at a common edge or at a common vertex.

4. The union of all triangles completely fills the domain $\Omega$.

$$\Omega = \bigcup_{\forall \tau_i \in T} \tau_i$$

Using this definition, a valid triangulation of the points $P$ can be seen as a subdivision of the convex hull of $P$. Other definitions do not put the restriction of a convex domain in order for a triangulation to be valid ([17]). I have chosen to do this because it simplifies matters. This definition is similar to what is called a regular valid triangulation in [17].

2.1 Using Triangulations to Model Terrains

As a base for a terrain model, a set of positions in $\mathbb{R}^2$ and height values are used. Each position is associated with one, and only one height value. A model of the terrain could be a surface interpolating these points. This may be done by making a triangulation of the positions in $\mathbb{R}^2$. The position decides a vertex’ X-, and Y-value. In addition, the height value in each point is used as a vertex’ Z-coordinate. The terrain surface inside a triangle is the
plane which interpolates the three corner vertices. The whole terrain is then represented by a piecewise linear function. Using this method will create a continuous surface. This means a surface without any cracks or holes.

I here present a theorem stating the continuity properties of this method.

**Theorem 2.1.1.** Given a set of points $P \in \mathbb{R}^2$ and a set of associated height values $H \in \mathbb{R}$. Let $\Omega$ be the convex hull of $P$. A continuous function $S : \Omega \to \mathbb{R}$, which interpolate all the points $v_i = (p_i, h_i)$, where $S(p_i) = h_i$ for $p_i \in P$ and $h_i \in H$, by making a valid triangulation $T$ of the points in $P$, and defining the surface $S$ as:

$$S(x) = \Pi_{i,j,k}(x), x \in \tau_{i,j,k}, \tau_{i,j,k} \in T$$

$\Pi_{i,j,k}(x)$ is the plane which interpolates the three points $v_i, v_j, v_k \in \mathbb{R}^3$

**Proof:** $T$ is a valid triangulation. The interior of the triangles in $T$ do not intersect according to the second valid triangulation criterion. Therefore the triangles will not intersect in $\mathbb{R}^3$. The interior of a triangle is a linear function (bivariate polynomial of first degree) which is continuous. A discontinuity in the surface therefore has to be on the boundary of a triangle between two neighboring triangles. Any two neighboring triangles in the triangulation share exactly two vertices and one edge due to the first and third valid triangulation criterion. As both triangles are planes which intersect both vertices on their common edge, the line connecting the two vertices also has to be in both planes, and there is no discontinuity between two neighboring triangles. The fourth criterion ensures that $\Omega$ is connected by triangles.

Graphics hardware uses triangles as a graphical primitive. This is why using a triangulation is a very efficient way to visualize a surface on a computer.

The Surface function has so far been represented by a piecewise linear function. Each triangle might instead be represented by a higher degree polynomial. By carefully selecting the polynomials inside each triangle, a spline surface of the terrain data may be made. A spline surface is a piecewise polynomial function with continuity requirements on its partial derivatives. This would create a smoother surface than the linear case. However, using such surfaces is beyond the scope of this thesis. Such surfaces, and other spline surfaces are presented in [18] and [19].
2.2 Delaunay Triangulations

Normally there are several triangulations which fulfill all of the valid triangulation criteria, and some of them are often better suited than others. A type of triangulation that is often preferable, and which has some favorable qualities is the Delaunay triangulation.

A Delaunay triangulation is a triangulation which reduces the number of long thin triangles. Long thin triangles may produce unwanted visual effects in visualizations. It may also lead to instabilities in algorithms. A Delaunay triangulation of the points \( P \in \mathbb{R}^2 \) has the property that the circumscribing circle of any triangle \( \tau \) in the triangulation does not contain any other points from \( P \) than the three corner points of \( \tau \); see Figure 2.2. Reasonably fast algorithms exist, to produce a delaunay triangulation. With a point set of \( N \) points, a Delaunay triangulation can be made in the order of \( N\log(N) \) operations. More information about Delaunay Triangulations is to be found in [17].

2.3 Data Dependent Triangulations

A Delaunay triangulation reduces the number of long thin triangles in two-dimensional space. If the space is expanded to three dimensions by adding a height value to each point as described in section 2.1, the desired properties
of Delaunay triangulations are no longer necessarily true. This is solved by using data dependent triangulations. When making a data dependent triangulation, the height value is taken into consideration when the triangulation is created. A data dependent triangulation algorithm reduces the number of long thin triangles in three dimensions. Unfortunately, finding an optimal data dependent triangulation has a much greater complexity than finding a Delaunay triangulation. Iterative methods have been developed which return good results. But making a data dependent triangulation is still much slower than making a Delaunay triangulation.

2.4 Memory Requirements

A triangulation of a set of points where there is no restriction on the position of the points is called an irregular triangulation. When storing such a triangulation each vertex has to be stored with three variables. Two variables places the point in the plane, and a third variable describes the elevation of the point. In addition, information identifying the triangles has to be stored. Topological relations may also be preferable to store. Several structures for representing triangulations are presented in [17]. If no modifications are to be made on the triangulations, the most efficient way to store it is to use a triangle strip or a triangle fan. A triangle strip is a series of neighboring triangles. A triangle strip is stored by using an array of vertices. The first three vertices in the array creates the first triangle. The two last vertices of the first triangle and the next vertex creates the next triangle in the triangle strip. This is repeated through the array. A triangle fan is a group of neigh-
boring triangles sharing a common vertex. The triangle fan is also stored in a vertex array. The first vertex is the common vertex. The consecutive vertices are the other vertices ordered in a clockwise or counter clockwise manner. A triangulation usually have to use several triangle strips and triangle fans in order to cover the whole domain.

2.4.1 A Regular Grid Triangulation

If all vertices in a triangulation are placed on a regular grid in the plane, all vertices may be stored in a two-dimensional array. In this case only the elevation value has to be stored, as its position in the plane is given by its coordinates in the two-dimensional array. In addition the regularity of the grid can make the storing of triangles, and the topological relationship between triangles unnecessary. Thus storing a regular grid triangulation demands less than one third of the storage of a irregular triangulation with the same number of points. On the other hand, as a regular grid triangulation restricts the position of the vertices to a regular grid, more vertices are normally needed to achieve the same accuracy in a regular grid triangulation than for an irregular triangulation.

2.5 Multiresolution triangulations

In this section I will introduce some notation and theory of multiresolution triangulations. A multiresolution triangulation is a system for creating triangulations with variable levels of detail. In order to do this, triangulations made by using a subset of the terrain elevation points are generated.
For smaller objects, the simplest way to achieve this is to generate several models with different detail levels in a preprocess, and then simply select the model which is sufficiently detailed during rendering. If the object is far away from the camera, a coarse model can be used. As the object moves closer to the camera, more detailed models can be used.

For larger objects such as terrains, this solution is not very efficient. Creating and storing several models may be space demanding for large terrain models. In addition, for large models, the camera is often much closer to some parts of the model than other parts, one would therefore want a method which can adaptively refine some parts of the model independently of the other parts of the model.

Several methods have been developed to achieve this in terrain models. Some of these are presented in section 3. In this section I will present a general theory which is common for most of them.

A coarse triangulation using a minimum number of elevation points is used as a start. This triangulation is called the base triangulation. In addition, local refinements are defined. A local refinement replaces a local subset of triangles ($T^-$) with another more detailed triangulation ($T^+$) covering the same area. This area is called the refinement's influence region. $T^+$ is a triangulation which uses more data points than $T^-$, and it is therefore more detailed. The word local means that the refinements influence region is a connected set of triangles. The most detailed triangulation is achieved by using the base mesh as a start, and then performing all the local refinements.
A local refinement may be stored explicitly, or given implicitly by subdivision rules. A refinement may also be a combination of explicitly stored data and certain rules.

In order to decide which refinements are to be performed, an error estimate \((\delta)\) is stored with each local refinement. \(\delta\) reflects the (world space) error in the surface inside the influence region when using \(T^-\). This error value may be presented in several ways, for instance as vector, or the radius of an error sphere. A common error metric used in terrains is a value representing the maximum vertical difference between the finest resolution triangulation and \(T^-\) inside the refinements influence region. By using the \(\delta\)-value, a triangulation satisfying a world space error threshold can be made. This can be done by using the base triangulation as a start, and then perform all refinements which do not satisfy the error threshold.

In order to achieve a view dependent multiresolution model, the \(\delta\)-value can be projected into screen space. The screen space error, \(\epsilon\) reflects the influence of the world space error in screen space for the current view. \(\epsilon\) is then a number for the pixel-error value that a refinement represents in the rendered screen. A view dependent multiresolution triangulation can now be made similarly to the method used earlier, by performing all refinements which do not satisfy a given view space error threshold.

The multiresolution method described in this section is a bit simplified. A local refinement may depend on other local refinements. Often certain triangles, edges, or vertices must be present in the triangulation before a refinement can be performed. If this is the case, special care has to be taken.

Certain models refines the triangulation by dividing the triangles in the triangulation in a regular manner. Such a triangulation is presented in the next section.

### 2.5.1 RIT triangulation

In the field of terrain modeling, a special type of triangulations has become very popular. Throughout this thesis I will call it a Right angle Isosceles Triangles (RIT) triangulation. This section contains definitions and proofs I have made concerning this type of triangulation.
Definition 2.5.1 (RIT triangulation). A Right angle Isosceles Triangle (RIT) triangulation is a triangulation consisting of only right angled isosceles triangles. A valid RIT triangulation is a RIT triangulation which is valid according to the valid triangulation criteria.

Definition 2.5.2 (RIT Diamond). Two triangles in a valid RIT triangulation sharing a common hypotenuse is called a diamond.

A right isosceles triangle may be divided into two smaller triangles by making a new edge from the center of its hypotenuse to the right angle. If this is done, the two new triangles will also be right isosceles triangles. This property can be used to refine a RIT triangulation. Two refinement operators are presented for an RIT triangulation:

Definition 2.5.3 (RIT refinement operator 1). A diamond is refined by inserting a vertex at the center of the two triangles common hypotenuse, and inserting edges from the new vertex to the right angles. This replaces the two neighboring triangles with four smaller right isosceles triangles.

Definition 2.5.4 (RIT refinement operator 2). A triangle with the hypotenuse at the domain boundary is refined by inserting a vertex at the center of the hypotenuse, and connecting it with an edge to the right angle. The original triangle is replaced by two smaller right isosceles triangles.

Using these operators on a RIT triangulation has some favorable properties.

Property 2.5.5. The resulting triangulation after having used refinement operator one or two on a RIT triangulation is also a RIT triangulation.
Proof: When using operator one, only the triangles creating the diamond are affected. Two right angled isosceles triangles are replaced by four triangles which also are right angled isosceles triangles. All other triangles in the triangulation are unaffected by the refinement operator and are therefore still right angled isosceles. A similar proof can be given to refinement operator two.

The consequence of this property is that a RIT triangulation will still be a RIT triangulation after having used the refinement operators an unlimited number of times. This is proven by using property 2.5.5 recursively. If the RIT triangulation is also a valid triangulation according to the valid triangulation criteria (theorem 2.0.1), the following property will hold.

**Property 2.5.6.** If refinement operator one or two is used on a valid RIT triangulation, the resulting triangulation will also be a valid RIT triangulation.

A similar proof of property 2.5.6 can be done for both refinement operators: A connected set of triangles $T_{\text{old}}$ are replaced by another set of triangles $T_{\text{new}}$ covering the same area. Each of the valid triangulation criteria are examined.

1.) The new triangles will not be degenerate because they are sub divisions of right angled isosceles triangles, and are still right right angled and isosceles.

2.) and 3.) The boundary of $T_{\text{old}}$ is the same as the boundary of $T_{\text{new}}$. All other triangles in the triangulation are unaffected. Only the geometry at the interior is changed. The triangles in $T_{\text{new}}$ only intersect at common vertices and common edges, and their interior do not intersect.

4.) $T_{\text{new}}$ covers the same domain as $T_{\text{old}}$, so the new triangulation covers the same domain as the old triangulation.

Property 2.5.6, state that a valid RIT triangulation can be refined an unlimited number of times and still be a valid RIT triangulation. In section 3.4 several algorithms which use RIT triangulations are presented.
Chapter 3

Some Common Algorithms Used in Multiresolution Terrain Visualization

I will in this section give descriptions of some of the most common algorithms used for creating view dependent adaptive terrain models. These methods use terrain height fields as input. A height field is a set of terrain elevation measurements in an $R^2$ domain. All methods are efficient enough for interactive visualizations in real time.

3.1 Tile Based Multiresolution Terrain

Thomas Sevaldrud, with supervisor Morten Dæhlen, developed a method for visualizing large terrains in his Cand.Scient Thesis. I will in this section briefly describe the method. For a more thorough description, the thesis [12] is recommend.

The method divides the terrain regularly into square tiles. For each tile, several triangulations at different detail levels are made. A number indicating the error tolerance in the triangulation is stored with each triangulation. To avoid discontinuities between the tiles a certain has to be made. The triangles with an edge at the tile boundary are replaced by another set of triangles called transition skirts. A transition skirt smoothly joins the triangulations
Figure 3.1: a) A tile consists of a triangulations in the center and transition skirts at the edges. b) Several triangulations and transition skirts are used for each tile. c) When the right tile is refined, the right transition skirt on the left tile has to be refined to maintain a valid triangulation. The shaded triangles are the triangles used in the transition skirts

in two neighboring tiles. Several transition skirts can be made for each tile in order to be able to join it with neighboring tiles at different levels.

The tile triangulations with the corresponding error values, and all the transition skirts are made in advance and stored to disk. A rendering program uses these data to visualize the terrain model. Whenever needed, a continuous triangulation covering the visible terrain is created. This triangulation satisfies a given error threshold. This process can be divided into three parts.

First, the visible tiles are selected. These are the tiles that are partly inside or completely inside the view frustum.

Second, each of the selected tiles are evaluated. For each tile, a triangulation satisfying a view space error threshold function is selected. If fewer transition skirts are made in the preprocess, restrictions to the difference between levels of neighboring tiles are set. If the difference between two neighboring tiles is too great, either the coarser triangulation has to be refined, or the finer tile has to be coarsened, or a combination of both.

Last, the triangles selected at the previous step are joined with transition skirts.

The union of the triangulations selected in the second step and the transition skirts selected in the last step creates a continuous triangulation of the visible domain, and can be rendered to screen.
3.1.1 Extending the Tile Based Terrain Model

By dividing the terrain regularly into tiles, the model will not be very scalable. If the model is viewed at a distance, showing the tiles at even the coarsest level may be too accurate. This has been solved in an extended version of the tile based terrain model. A tile at the coarsest level consists of four triangles. In the extended version, two neighboring triangles can be joined together to create a larger triangle. This operation is not restricted to the tiles, so new triangles may be joined together to cover several tiles. By doing this, the terrain surface may be coarsened even beyond the restriction of the regular grid.

3.2 Progressive Meshes

Hoppe introduces Progressive Meshes (PM) in [4]. This is a method for creating triangulation models with variable level of detail. The two main operators in a progressive mesh are the vertex split and the edge collapse. These operators are the inverse of each other. When an edge collapse is performed, one edge and two triangles are removed from the triangulation, and two vertices are merged into one. A vertex split is the opposite action where a vertex is split into two vertices, and a new edge and two triangles are created. For a vertex split, or an edge collapse to be performed, the triangles which share edges with the triangles which are removed or created must be in the triangulation. These triangles are shaded in figure 3.2a. A progressive mesh consists of a base mesh, which is a coarse approximation of the original mesh, and a sequence of vertex splits. The vertex-splits refines the coarse base mesh into an accurate mesh.

The method described in [4] use a world space error threshold independent of the camera view when deciding the detail level in a triangulation. Later in [6]. Hoppe et.al. extended the theory of progressive meshes to achieve view dependent level of detail.

3.2.1 View Dependent Progressive Mesh

A View Dependent Progressive Mesh (VDPM) is built by coarsening an accurate mesh, and storing the changes. Starting with the finest mesh, the
Figure 3.2: a) The two geometry operators of a progressive mesh: the vertex split, and the edge collapse. Both operators depend on the shaded triangles. b) $v_s$ is stored as the parent of $v_u$ and $v_t$ in a vertex tree. c) The vertex tree. $M_0$ is the base mesh. $M^*$ is the vertex front.

An edge contributing to the least significant error value by using a world space error function, is removed with an edge collapse. This process is repeated until the coarsest mesh is met. For each edge collapse the remaining vertex is stored in a binary tree as the parent of the two vertices defining the collapsed edge (figure 3.2b). The coarsening operation creates a forest of binary vertex trees (figure 3.2c). The roots of these trees are the vertices in the coarsest mesh, or the base mesh. A vertex is called active if it is used in the triangulation. A world space error value is stored in association with each edge collapse during the vertex tree creation. This error is used when the selection of active vertices are done. The set of active vertices are called the vertex front. The vertex front divides the root from the leaves in the vertex binary tree forest. By using the base mesh as the vertex front as a starting point, each vertex in the vertex front is evaluated using a view dependent error function. This function uses the world space error stored with each vertex, and the camera parameters, to approximate the vertex’ error in screen space. If a vertex is too inaccurate and has to be split, it is replaced by its children in the vertex front, and they are further evaluated. The difference between two consecutive rendered frames is often small. To take advantage of this property, the vertex front used in the previous frame may be used as a starting point for evaluating the next frame. In this case each vertex in the vertex front has to be evaluated both for vertex-splits, and edge-collapses.
### 3.2.2 Using a View Dependent Progressive Mesh for Terrain Visualization

In [6], the theory of VDPM is adapted to terrain modeling. This is done by introducing a hierarchical subdivision of large terrain models. In the creation of a VDPM described earlier, it is preferable that the original triangulation may be located in main memory. Terrain models can be very large, and will often exceed the main memory of an ordinary computer. Hoppe solves this by partitioning the original mesh into smaller tiles. Each of these smaller tiles fits into the main memory, and ordinary VDPM preprocessing may be used. Special care has to be taken to prevent discontinuity between neighboring tiles. After each tile has been coarsened to a certain level, two or more neighboring tiles can be combined to larger tiles. The VDPM preprocessing can be used on the combined tiles. These tiles will now fit into main memory as each of the sub-tiles are coarsened versions of the original mesh, and therefore do not contain as much data as before the preprocessing. This process can be repeated recursively until the whole domain is contained in one tile. The result is a hierarchy of view dependent progressive meshes.

View dependent progressive meshes used in terrain rendering is a very efficient method. Implementing it is unfortunately quite complex.

Figure 3.3: The creation of a hierarchical progressive mesh.
3.3 Multi-Triangulations

In [10] theoretical material for a multiresolution method called Multi-Triangulations (MT) is presented. An MT is a collection of refinements stored in a Directed Acyclic Graph (DAG). At the source of the DAG, is the base mesh. And in the drain is the most detailed mesh. All paths through the DAG starts at the source, and ends up in the drain. A refinement replaces a set of triangles $T^-$ with a more detailed set of triangles $T^+$. A refinement is placed in the DAG so that it has an arc from all the refinements which introduce the triangles in $T^-$, and it has arcs pointing to all the refinements which use the triangles introduced in $T^+$. A valid selection of refinements is a selection where all refinements which has an arc pointing to a selected refinement, is also a selected refinement. This is reasonable, as all triangles $T^-$ in a refinement have to be introduced before the refinement itself can be performed.

An implicit and an explicit version have been implemented and are presented in [10]. The explicit version stores all elements in the DAG explicitly. This requires a lot of memory as all triangles in all levels, with dependencies, are stored. This implementation has little practical value as it will be too memory demanding when large terrain data are used.

The implicit version uses a more compact data structure. It stores the base triangulation explicitly, the relationships between refinements are also stored, but each triangle in $T^+$ and $T^-$ are not stored for each refinement. Instead only the points which are inserted in the refinement are stored for each re-
finement, and the rules used in the Delaunay point insertion algorithm are used to make local refinements.
3.4 Multiresolution Right Angle Isosceles Triangles

In the field of multiresolution triangulations for terrain rendering, the use of RIT triangulations has become very popular. I will in this section describe how a multiresolution terrain model can be made by using RIT triangulations and the refinement operators presented in section 2.5.1. This set of multiresolution triangulations was first introduced for terrains in [1], and it has later been used in several published articles ([2], [3], [7], [8], [13]).

A multiresolution triangulation can be made on a quadratic discrete height field with dimension $(2^l+1) \times (2^l+1)$. The coarsest triangulation of this type consists of two right angle isosceles triangles, creating a diamond covering the whole domain. This is the base triangulation. The base triangulation may be refined by using the first refinement operator. Due to the dimension of the underlying grid, the vertex inserted at the center of the hypotenuse corresponds to the point at the center of the height field. The resulting mesh may be further refined by applying the refinement operators where it is needed. The vertices inserted at the hypotenuses will correspond to points in the height field in $2l$ levels of refinement.

Because the base mesh is a valid RIT triangulation according to the valid triangulation criteria, all refinements of it will also be valid RIT triangulations as long as the RIT refinement operators are used. This is confirmed by property 2.5.6.
3.4.1 Data Structures

The grid refinements in this triangulation can advantageously be stored in a directed acyclic graph (DAG). Two alternative data structures are commonly used to describe the relationship between triangles and vertices in a multiresolution RIT model; the binary triangle tree, and the vertex tree.

**Binary Triangle Tree**

The ROAM method, described in [7], uses a tree structure which they call a Binary Triangle Tree (BTT). A node in this tree corresponds to a triangle in the triangulation. Each triangle may be split into two smaller triangles by creating a new edge from the right angle to the center of the hypotenuse. When a triangle is split, the nodes corresponding to the two new triangles are stored as the children of the original triangle node. Each node in this tree has two children and one parent, as each triangle can be divided in two. A triangulation of a square domain uses two trees, one for each triangle in the base mesh. In this tree, the nodes represent triangles in the triangulation. If a triangle node is replaced by its two children in the BTT, this corresponds to a triangle split in the triangulation. Therefore the binary triangle tree naturally corresponds to a triangle split as a refinement operator. To avoid cracks in the triangulation, triangles in the interior of the triangulation must be a part of a diamond before it is split. If this is not the case, the neighboring triangle must be forced to split, so that a diamond is created. This means that a triangle split may lead to a series of triangle splits. For each triangle split, the neighboring triangles have to be evaluated to maintain a valid triangulation, and a continuous surface.
Vertex Tree

In [1] a different data structure is used. This structure corresponds more naturally to the RIT refinement operators presented in section 2.5.1 than the BTT. This structure also use a DAG, but their nodes represent vertices in the triangulation and not triangles as in the BTT. The root node in this tree corresponds to the first vertex inserted at the hypotenuse of the triangles in the base mesh. This vertex divides the two base mesh triangles into four triangles. Each node has two parents and four children, or one parent and two children if they correspond to vertices at the boundary. The parents of a vertex inserted in the center of a diamond, are the two vertices at the right angle of the two triangles defining the diamond. The children of a vertex are the vertices inserted in the diamonds where the vertex is at the right angle of one of the triangles defining the diamond. This is illustrated in figure 3.7 The first four levels of the triangle three are illustrated in figure 3.8

An alternative way of thinking of the vertex tree is as a tree of refinements. Starting with the base mesh, the root of the vertex tree represents the only possible refinement. When a vertex is selected, and a refinement is performed, the children of the vertex node represent further possible refinements. A vertex cannot be selected unless all of its parents are selected. An unselected vertex in the interior with both parents selected in the triangulation is the vertex in the center of a diamond. Selecting the vertex corresponds to performing the first refinement operator. An unselected vertex at the boundary has only one parent. If the vertex’ parent is active, the vertex will be at the center of the hypotenuse of a right angle isosceles triangle, with the parent at the right angle. Selecting it will correspond to performing refinement operator two.

A selection of vertices in this vertex tree, is called a valid selection, if all ancestors of every vertex in the selection is also in the selection.
Definition 3.4.1 (Valid Vertex Selection). Let $V_T$ be the set of vertices defining the triangles in the triangulation $T$. A valid selection of vertices in a vertex tree, is a selection where the following holds for all $v_j \in V_T$:

$$v_j \in V_T \Rightarrow v_{i_1} \in V_T \text{ and } v_{i_2} \in V_T \text{ where } v_{i_1} \text{ and } v_{i_2} \text{ are the parents of } v_j.$$ 

Implicitly, all ancestors of a vertex in $V_T$ must also be in $V_T$.

Let $\Delta_{RIT}$ be the set of all triangulations which can be made by using operator one and two an unlimited number of times on the base mesh. In the vertex tree model, a triangulation in $\Delta_{RIT}$ is implicitly given by a valid selection of vertices in the vertex tree.

Lemma 3.4.2. Any valid vertex selection corresponds to a triangulation in $\Delta_{RIT}$.

Proof: As all parents of all selected vertices are also selected in the triangulation, all selected vertices represent valid refinements. Because the base mesh is a valid RIT triangulation, all valid vertex selections will therefore represent a valid RIT triangulation according to property 2.5.5.

This leads to the following theorem:

Theorem 3.4.3. For each valid vertex selection, there is a corresponding valid triangulation according to the valid triangulation theorem.

Proof: A valid vertex selection corresponds to a triangulation in $\tau_{RIT}$. A triangulation in $\tau_{RIT}$ is a valid triangulation according to the valid Triangulation theorem. This is a property of RIT triangulations. (Property 2.5.6)
3.4.2 Data Structure Traversal

To create a triangulation with a variable level of detail, a subset of the nodes in the BTT, or the vertex tree, has to be selected. Several attempts have been made in order to make an efficient detail selection. The selection should produce the smallest amount of triangles which satisfies a given view space error as efficiently as possible. Here I will describe some of the methods used to make a fast vertex or triangle selection. I will present them in a chronological order.

**Block based and vertex based simplification**

In [1] multiresolution RIT triangulations was introduced for terrain modeling. The algorithm uses the highest resolution grid as a basis. It then simplifies the grid, in a two step process. The first step is a coarse-grained simplification. The entire mesh is evaluated as a quad tree of blocks. A block is a collection of vertices. The vertex decimation starts with the entire finest resolution mesh in one block. If the block is too accurate, the block is coarsened by removing every other vertex of the mesh. If the block is too coarse, the block is refined by substituting it with its four higher resolution children blocks. This process is repeated recursively until all blocks fall within an acceptable error threshold interval. The second step uses the result of the first step as a start. Each triangle in the mesh is now evaluated. If a triangle is too accurate it may be coarsened by joining it with its neighboring triangle. The second step uses a vertex dependency graph similar to the vertex tree described above to maintain a valid vertex selection and avoid discontinuities in the triangulation.
The drawback of this algorithm is that it uses the most detailed triangulation as a base mesh. By doing this, the time it takes to select a set of vertices depends on the amount of data used in the most detailed data set which can be very large. If the coarsest mesh is used as a start, the number of computations used to select a set of triangles will depend on the number of triangles selected for the current view. This is better, as it is, in theory, independent on the size of the most detailed data model.

**Priority queues**

ROAM uses the base mesh as a start for the first frame. For the following frames ROAM uses the triangle selection from the previous frame as a start. Two priority queues are used. The split queue holds all triangles in the current triangulation, and is sorted by its view dependent error value, \( \epsilon \). The triangle with the highest priority is split, and its children are inserted in the queue. This is repeated until a certain threshold is met. Each triangle is evaluated for a triangle split, independent of whether it is part of a diamond or not. If a triangle selected to be split is not part of a diamond, forced splits on neighboring triangles is necessary, in order to maintain a continuous surface. A valid triangle selection is created after each refinement. Another queue called the merge queue is used for grid coarsening. When a diamond is split, the resulting four triangles are called a mergeable diamond. All mergeable diamonds in a triangulation are sorted in the merge queue. When the mergeable diamond with the highest priority is too accurate, it is merged into a diamond, and removed from the queue.

One advantage of the ROAM method is that when using the priority queues the triangles which need refinement most is refined first. As a valid triangulation is maintained after each refinement, the detail selection process can be interrupted, and a valid RIT triangulation can be drawn. The priority queue ensures that the refinements which have been performed are the refinements which are needed most.

Unfortunately the ROAM method has some drawbacks too. As the method uses the triangulation created in the previous frame as a start, a lag in the triangulation selection will spread to the next frame, and recursively to the following frames. In addition the sorting of the priority queues can be a time-consuming task.
Nested Error Spheres
In [8] Blow compares the two previous algorithms, and presents a new method for selecting triangles in a RIT triangulation. For each vertex, an isosurface describing the view space error threshold in world space is created. A view space error metric based on distance will create spheres in world space as the isosurface for the error threshold. For the sake of simplicity this error metric is used. A radius describing the vertex’s error sphere is stored with each vertex. The vertex selection process used is simple. If the viewpoint is inside the error sphere of a vertex, the vertex is active in the triangulation. If the viewpoint is outside the error sphere, it is not activated. The error spheres are sorted in a hierarchy, if a sphere is contained in another sphere, it is stored as a child of the sphere. This means that if the viewpoint is outside an error sphere, it will be outside all the sphere’s children. Artificial spheres which are not associated with any vertices can be inserted in the sphere hierarchy to improve nesting. After a vertex selection has been performed, vertex dependencies have to be resolved by using a dependency graph similar to the vertex tree.

Vertex tree traversal using bounding spheres
Lindstrom et al introduces a new vertex selection algorithm in [2]. This method uses a vertex tree as described earlier in this section. Previous selection methods need to resolve dependencies between triangles or vertices during runtime to avoid cracks in the triangulation. This method however, selects a valid vertex selection according to the valid triangulation theorem during the selection process. Therefore no extra effort has to be made after vertex selection in order to avoid cracks in the triangulation.

In addition to height value and possibly x, and y position, two variables are stored with each vertex, a world space error $\delta$, and a bounding sphere radius $r$. The world space error is set so that it decreases as the vertex tree is traversed from the root to the leaves. A vertex bounding sphere $\mathcal{B}_v \subset \mathbb{R}^3$ has the sphere center in the position of $v$, and its radius is set so that it completely contains its children’s bounding spheres, and all of its descendants’ positions.

For all vertex parents $v_p$
$$\delta_{v_p} \geq \delta_{v_c} \text{ and } \mathcal{B}_{v_p} \supseteq \mathcal{B}_{v_c}$$
Where $v_c$ are the children of $v_p$. If $v_c$ is a leaf, $r_{v_c} = 0$ and $\mathcal{B}_{v_c} = $ the point $v_c$

Using the variables stored with each vertex, and the camera parameters, a screen space error can be created.
Let $P(\delta, p)$ be a function that calculates the screen space error from the world space error, $\delta$ placed in position $p$. The only restriction set to $P$ is that $P(\delta_1, p) < P(\delta_2, p)$ when $\delta_1 < \delta_2$.

Let

$$E(\delta, v) = \max_{v_p \in B_v} P(\delta, p)$$

If $P$ is function based only on the distance from the camera to the position $p$, where $P$ increases as the distance decreases, the optimization problem in $E$ reduces to subtracting the distance used in $P$ with the bounding sphere radius, $r_v$. The screen space error $\epsilon_v$ of a vertex $v$ is given by the following formula:

$$\epsilon_v = E(\delta_v, v)$$

By using this screen space error, the following property will hold.

**Property 3.4.4 (Screen Space Error Property).** For all vertex parents, $v_p$

$$\epsilon_{v_p} \geq \epsilon_{v_c}$$

Where $v_c$ can be any of the vertex $v_p$’s children. By using recursion, a vertex will have a screen space error larger than or equal to all of the screen space errors of its descendants.

Proof: Given a vertex $v_p$ and one of it’s children $v_c$ We know that $\delta_{v_p} \geq \delta_{v_c}$ and that $v_c$’s bounding sphere is completely contained in $v_p$’s bounding sphere.

$$\epsilon_{v_c} = E(\delta_{v_c}, v_c) \leq^* E(\delta_{v_c}, v_p) \leq^{**} E(\delta_{v_p}, v_p) = \epsilon_{v_p}$$

*) Because $B_p \supset B_c$

**) Because $\delta_{v_p} \geq \delta_{v_c}$

This is naturally valid for both parents.
When a vertex selection is performed, all vertices with a screen space error larger than a screen space error threshold \( \kappa \) are used in the triangulation. As a child’s screen space error is less than or equal to both of it’s parent’s screen space errors, it will not be activated unless both of it’s parents are activated. Therefore the vertices selected by using this criterion will be a valid vertex selection according to 3.4.1. and a valid triangulation can be made by using these vertices according to theorem 3.4.3.

By using the vertex tree, the vertex selection can be done very quickly. The vertex tree is traversed recursively, starting at the root. If a vertex does not satisfy the screen space error threshold, it is included in the triangulation, and its children are evaluated. The recursion stops when a vertex satisfies the view space error threshold. In that case, all of the vertex’s descendants will also satisfy the error threshold due to screen space property presented above, and they do not have to be evaluated. By using this method, the number of vertex selection computations is proportional to the number of vertices in the triangulation, and independent of the number of vertices in the terrain model.

Lindstrom and Pascucci’s method seems to be one of the most efficient algorithms presented, and this is why we have chosen to examine this method further. In section 4 the memory handling of vertex trees are discussed. Their method is also used as a base for the implementation presented in section 5.

3.5 Closing Remarks

Several methods used for multiresolution triangulations have been presented in this section. These brief summaries are meant to give a general idea of how such algorithms can be implemented. For a more thorough description of each algorithm, the reader is asked to consult to the original articles.

There are several aspects one must be aware of when choosing an algorithm for terrain visualization. I will here present some of them, and compare some of the algorithms.

All methods presented in this section work in two steps. The terrain data is first processed and stored to disk, another program uses the processed data to visualize the terrain. The implementations are often divided in two, as the preprocessing can be time-consuming, and is unnecessary to perform for
each rendering.

The size of the processed data needed to visualize a terrain model varies from method to method. For large terrain models, the terrain data will exceed the amount of internal memory. In such cases the reading of data from the hard drive during run-time will lead to a loss in performance. If compact structures are used, more of the data will fit in main memory and less data has to be read from the hard drive during rendering. Methods for optimizing reading of vertex trees from the hard drive during rendering are presented in section 4, and in [3].

Another aspect one should consider is how the work load should be divided between the Central Processing Unit (CPU), and the Graphics Processing Unit (GPU). For Progressive mesh, and the methods based on the RIT triangulations, calculations have to be done for each vertex used in the triangulation during detail selection. This puts a lot of the work load on the CPU. The resulting triangulation however use the least amount of triangles to fulfill the screen space threshold. This eases the workload of the GPU. The tile based triangulation represents a different strategy. In this method, sets of triangles are evaluated at a time. For a given view, the workload on the CPU is not changed as the view error threshold is changed, because the same tiles are evaluated. The CPU simply chooses different triangulations for the tiles. The resulting triangulation includes more triangles than necessary to satisfy the view error threshold, because the whole tiles must be represented at the same level of detail. By doing this, most of the work load is moved to the GPU. This might be a good strategy, as GPU’s have become very fast over the last years. In addition, when less of the CPU is used for the computer graphics, more of the CPU can be used by other processes.

Some features are more easily implemented in some methods than other. For instance Progressive meshes and some of the RIT methods easily support geomorphing (i.e. [3], [6]). When geomorphing is used, new vertices are gradually moved from the original triangulation to its position. This reduces the amount of sudden changes in the geometry, which is called “popping”. Geomorphing is a feature that is very difficult to implement in the tile based method. The result is that the tile based method will need a very low error threshold, which again will need a large amount of triangles to avoid the “popping" when tiles are coarsened or refined.

Constrained triangulations is a feature that is more easily implemented in the methods which do not use a regular grid of vertices as a base. Constrained
triangulations are used when information from for instance roads or rivers are associated to certain edges and vertices in the triangulation. This sets limitations to how a triangulation can be coarsened and refined. Theoretical material on constrained triangulations is presented in [15], and references therein.
Chapter 4

Memory Handling

When visualizing large terrain models, large data sets are often used. If the dataset in use is larger than the internal memory of the computer, parts of the dataset is stored on the computer’s hard drive. The data located on the hard drive will have a considerably longer access time than the data located in the internal memory, or in the processor cache. When a variable is accessed from the hard drive, the page containing the variable is moved to the computer’s main memory or the cache. If the program then asks for a variable on the same page, it will already be read from the hard drive, so the new variable will be returned very quickly. This is called memory prefetching. Because of this, variables which are likely to be accessed succeedingly should also be stored close to each other in memory. This will hopefully increase the number of cache hits, and decrease the number of page fetches from the hard drive. In the vertex tree mesh described in section 3.4, the data is accessed from the root to the leaves. Following the argumentation done earlier, the array storing these variables should preferably be sorted in this order as well. This will not be the case if the data is stored for example from east to west, and south to north. In this section, I will present methods for achieving this.

4.1 Interleaved Quad Trees

In [2] Lindstrom suggests that the vertices in the vertex tree should be stored as two quad trees. Vertices introduced at an even level should be stored in one quad tree, while the vertices introduced at an odd level should be stored
in a quadtree tilted 45° to the regular grid. The tilted quadtree will expand outside the range of the dataset. The nodes outside the domain will not contain any data, and they are therefore called ghost vertices. By storing the vertices as quadtrees, vertices close to each other in the vertex tree will also be close to each other in memory. This means shorter data access time. The disadvantage by using this method is that as the odd level quad tree grows, the number of unused vertices will increase accordingly. In [2] they estimate that for a large vertex tree, about 66% of the input data will be ghost vertices. This number may be reduced by storing the even level quad tree at the ghost vertices of the tilted odd level quad tree. By doing this the share of ghost vertices is reduced to about 33%. For large data sets, this is still a considerable amount of unused space.

4.2 Hierarchical Z-order Indexing

A method for hierarchical z-indexing of volumes with dimension $2^n$ in each direction is presented in [9]. In [11] a method of storing vertex trees using a quad tree with triples of vertices and two binary trees is presented. In this section I present an indexing method where features from both articles have been combined. The quad tree in [11] is replaced by a hierarchical z index field similar to what is presented in [9]. By doing this, fast conversion from coordinates to an array index is achieved. The vertices are stored in a single array very similar to how they are accessed in a width-first traversal in the vertex tree. This method also has good locality properties in the sense that vertices which are close to each other in each vertex tree level, will be stored close to each other in the array. As opposed to the interleaved quad trees presented in [2], this method has no “ghost vertices”. A similar approach using a H-order space filling curve is presented in [3].

The z-order indexing covers an area of $2^n x 2^n$. In addition, two binary trees are added at the north and east edges to make the grid dimension $2^n + 1 x 2^n + 1$. Both the field covered by the hierarchical z-order indexing, and the two binary trees are stored in a single array. In the following section I will show how to compute an index from an x,y coordinate using this method.
4.2.1 Z-order Indexing

The z-index is very easily computed from the x- and y- coordinates by simply interleaving the bits from each of the coordinates. This may be done by dividing the coordinates into bytes (8-bit). Each byte from each coordinate are combined to a short integer (16 bit), and a lookup table is used to interleave the bits in each byte, for both 16-bit integers. (see figure 4.1)

Afterwards, the index is sorted into its hierarchical level. A grid with dimension $2^l \times 2^l$ has $l$ “Z-levels”. The first level contains one large z. Each z is divided into four smaller z’s in the next level, similar to a quad tree. So the second level has four z’s, the third level has sixteen, and so on. The vertices are sorted by level, and are then stored in z-order for each level. To find the level of the z which contains a given coordinate, the number of zeros at the end of the coordinates’ z-order index in base four is counted. A Z’s level is then the total number of levels minus the number of zeros in the end of the z

---

Figure 4.1: Interleaving the coordinates to a z-order index by using a 16-bit lookup table

Figure 4.2: The result of z order indexing.
Figure 4.3: The indices are sorted by ordinary z-order indexing. All numbers in this figure are expressed in base four. The different Z-levels are indicated.

index expressed in base four. For example in a 8x8 grid there are three levels \(2^3 = 8\). \(X00_4\) will be at the first level, \(XX0_4\) at the second and \(XXX_4\) at the third and last level. This is illustrated in figure 4.3. From this figure another observation can be done. The lower left vertex of a Z is shared by a Z in a lower level. This value is stored in the lowest level. Therefore only the top left, the top right, and the bottom right value are stored. Thus three indices for each Z.

From this information one may compute the index of the first vertex in a level. Each Z has three vertices. And the number of Zs in one level is four times the number of Zs in the previous level. The first level has one z. This leads to the following formula:

\[
\text{first index in level} = 3 \times \sum_{i=1}^{\text{level}-1} 4^{i-1} \quad \text{for level}>1
\]

This is a geometric series which can be simplified with the standard formula for a finite geometric series:

\[
3 \times \sum_{i=1}^{\text{level}-1} 4^{i-1} = 3 \times \frac{1 - 4^{\text{level}-1}}{1 - 4} = 4^{\text{level}-1} - 1
\]

The index inside a level will be the z-order index with the zeros at the end shifted away. This is the correct index if each z contains four indices, but as this is not the case and each Z only contains three vertices, this index has to be subtracted by the number of Zs before the current z in this level. This number corresponds to the index inside a level with four indices per z,
divided by four (or the last two bits shifted away). This gives the following formula for the level index:

\[ \text{level index} = (index \gg (zeros)) - (index \gg ((zeros) + 2)) \]

This gives the following formula for computing the final Hierarchical Z-index:

\[ HZIndex = \text{first index in level} + \text{level index} \]

### 4.2.2 Binary tree Indexing

At the north and east side of the z-indexing field, a binary tree has been added. In binary tree indexing, ideas similar to the hierarchical z-indexing have been used. The x coordinate is used as a basis on the north binary tree, and the y-coordinate is used as a basis on the east binary tree. The binary trees has the same number of levels as the z-quad-tree. And similar to the HZ-indexing, the number of zeros at the end in a binary number system decides the hierarchy level. In a binary tree the number of vertices is doubled for each level. The first index of a level is therefore computed by this formula:

\[
\text{first index in level} = \sum_{i=1}^{level-1} 2^{i-1} = \frac{1 - 2^{level-1}}{1 - 2} = 2^{level-1} - 1
\]

The index inside a level is the base index with the last zeros, and the following 1 bit shifted away. That is:

\[ \text{level index} = coord \gg ((zeros) + 1) \]

Similar to the HZ-indexing, the final binary tree index is given by this formula

\[ BinTreeIndex = \text{first index in level} + \text{level index} \]

### 4.2.3 Putting it Together

As mentioned earlier, two binary trees and one hierarchical z-index field are used to store the vertices in the 4-k mesh. In addition the four corner
Figure 4.4: The base indices increasing from left to right sorted in a binary tree

Figure 4.5: a) The final index is a combination of two binary trees and a z-order indexing field, b) The resulting indices in a 9x9 grid. c) The data elements are stored in a single array
vertices are stored separately. In order for the data to be stored similarly to
the vertex tree, all these elements are interleaved in a single array. The four
corners vertices are stored first. Then the elements are sorted level by level,
as described in figure 4.5c. The computation of an index is similar to the one
before.

\[ \text{index} = \text{first index in level} + \text{level index} \]

In this formula, when the “first index in level” is computed, the number of
vertices in all earlier levels of both the hierarchical z index and the two binary
trees have to be counted. The “level index” is computed the same way as
described earlier.

Results after comparing this method with a simpler indexing method in a
terrain visualizing application is presented in section 6.4.
Chapter 5

The Test Program

An application for visualizing terrain data has been implemented. The method used in this program may be seen as an extension to the algorithm described in “Visualization made easy” by Lindstrom and Pascucci. Their article gives a description to an algorithm that is not too difficult to implement, and their algorithm is also one of the most efficient ones algorithms published today.

Their algorithm uses a vertex tree which covers the whole terrain. In this section we will extend their method to use tiles where each tile contains a vertex tree.

There are certain advantages when using tiles.

- The constraints set to the dimensions of the terrain data are less rigid. When only one vertex tree is used on the whole domain, the underlying grid is restricted to \((2^l + 1) \times (2^l + 1)\), By using tiles, terrain data with dimensions of \((w \times 2^l + 1) \times (h \times 2^l + 1)\) where \(w\) is the number of tile colons, and \(h\) are the number of tile rows are allowed.

- Texturing a terrain is made more flexible. If the terrain is large, one would often want the texture to be more detailed than a single texture which can cover the whole terrain domain. If tiling is used, the terrain can be divided into smaller parts which each may use a separate texture.

- View culling is made more efficient. The tiles which are outside the view frustum are not drawn. This reduces both some of the geometry
outside the view, and the size of the textures used.

Using tiles with multi resolution RIT triangulations is not a new idea. In [16] it is described how this may be done using the ROAM algorithm for each tile. Pajarola describe how to use a RIT triangulation with tiles in [13]. Here the dependencies between vertices and between tiles are resolved by using a dependency graph similar to the vertex tree. Both their methods have one disadvantage. To avoid discontinuity between tiles, a lot of extra effort has to be taken. This probably slows down their algorithms. I will describe here how this may be avoided by adapting the algorithm described in [2] to tiles.

5.1 Tile Based RIT Multiresolution Triangulation

The terrain is divided regularly in $w \times h$ square tiles, where $w$ is the number of tile columns and $h$ is the number of tile rows. Each tile contains a base triangulation of two right angle isosceles triangles. A vertex tree is used for each tile, so that each tile contains a regular grid of data points with dimension $2^l + 1$. In order to keep it simple, we use the same dimension on the underlying grid for all tiles. The vertices at the tile borders in the interior of the terrain domain are shared by two tiles, therefore the dimension of the whole domain will be $(w * 2^l + 1) \times (h * 2^l + 1)$. One of the positive features with Lindstrom and Pascucci’s algorithm is that the main effort made to avoid discontinuities is handled during the preprocessing, and not while rendering. The same method may be used after certain modifications to avoid discontinuities between the tiles. The only extra effort needed is simply that one must allow the vertices in a vertex tree to have children outside its tile during the preprocessing. In [2] the vertices at the border in the vertex-tree only have two children. When tiles are used the vertices at the border which are shared by the neighboring tile also have four children, where two are inside the neighboring tile. The vertices at the edge of the whole terrain domain still only have two children.

If this is done, the vertices at the border shared by two tiles will use the same world space errors and bounding spheres for both tiles. This means that if a vertex on the common boundary is selected in one tile during run-time, it will also be selected in the other tile. Due to this, no discontinuities will appear between tiles.
Figure 5.1: The vertex tree expands outside a tile during the preprocessing

The vertex tree graph of a tile is illustrated in figure 5.1.

When the terrain is rendered, the vertex selection algorithm may be used unmodified from [2]. Each of the tiles can be evaluated separately. One single triangle strip may be created for each tile, and no extra effort has to be made in order to avoid discontinuities, as this has already been done during the preprocessing. If two neighboring tiles are drawn, there will not be any discontinuity between them.

5.2 Implementation

I decided to divide the method into two programs, the Preprocessor and Renderer. This was done because the preprocessing may be time-consuming for large datasets, and is unnecessary to perform for each rendering. The Preprocessor divides the terrain into tiles and computes delta and radius values associated with each vertex. It then stores the processed data to the hard drive. The Renderer uses data from the files written by the preprocessor,
and visualizes the terrain.

5.3 The Preprocessor

The Preprocessor may be divided into three parts. This is illustrated in figure 5.2 In this section each part is described.

5.3.1 Read Data

Data is read from various formats. The preprocessor should be capable of reading different data formats, therefore the actual file handling and file interpretation is separated in the DataReader class. A DataReader class has been developed for both the bmp- and the ascii-format, and it can easily be extended to other formats.

5.3.2 Compute World Space Errors, and Bounding Spheres

Once the data has been read, the new variables needed by the algorithm are computed. These variables are the world space error, and the bounding sphere radius.
The world space error, $\delta_v$, of a vertex $v$ is an estimate of the error in the triangulation before $v$ and all its descendants are inserted in the triangulation. The world space error of a parent has to be larger or equal to the world space error of all its children for property 3.4.4 to hold, therefore the vertex tree is traversed setting the delta value recursively, from the root using the following rule:

$$\delta_{vp} = \max \left\{ \delta_{diamond_v}, \delta_{vc}, \text{where } v_c \text{ are all the children of } v_p \right\}$$

$\delta_{diamond_v}$ is an estimate for the error in the triangulation before the vertex $v$ is chosen for insertion. If $v$ is chosen for insertion, both parents are already inserted because of property 3.4.4. Both parents will be vertices at the right angles of two right angled triangles creating a diamond. The vertex which is evaluated for insertion is the vertex at the center of the hypotenuse in the center of the diamond. The error value we are looking for is therefore an estimate of the error in the triangulation by using the diamond. I have implemented two different world space error estimates. One compares the diamond to the finest resolution triangulation. The other computes the change in geometry when inserting the center vertex.

- world space error estimate 1: The diamond, of which the vertex is inserted at the center, is compared to the finest resolution grid. The maximum vertical difference between the finest grid and the two triangles in the diamond is used as the world space error.

- world space error estimate 2: The vertical distance from the vertex to the two triangles creating the diamond is used as the world space error. This is the distance between the center of the hypotenuse shared by the two triangles, and the vertex.

In the same recursive vertex tree traversal, the radius of the bounding sphere associated with the vertex is also computed by this recursive formula:

$$r_i = \left\{ \begin{array}{ll} \text{grid cell size if } v_i \text{ is a leaf} \\
\max_{v_j} ||v_i - v_j|| + r_j, \text{ where } v_j \text{ are all the children of } v_i \end{array} \right.$$  

Using this satisfies the requirement for property 3.4.4, that $B_{vp} \supset B_{vc}$. This implementation does not use radius equal to zero for the vertex leaves because
the same radiuses are used for view culling. By setting the leaf radius equal the grid cell size, a vertex’ radius does not only contain all its descendants’ vertices, but also the triangles where all of its descendants are a part of. This does not affect the validity of property 3.4.4. In this implementation the vertex values are set from the root by calling a function recursively to its children. When tiles are used, a vertex children can be inside the neighboring tiles as described in 5.1. As each vertex in the interior has two parents, special care has to be taken so that each vertex is not evaluated more than once. This problem can be avoided completely by setting the values from the leaves to the root as described in [3] instead.

5.3.3 Sorting and Storing Vertices

The last step is to sort the vertices from a two-dimensional domain into a one dimensional array. As it is desirable to experiment with different indexing methods, this is also separated in the Index Module. This module translates a two-dimensional coordinate to a one dimensional array index. I have implemented an indexing class for linear indexing, as well as the Hierarchical z-order indexing described in 4.2. The linear module simply stores the vertices from east to west, and from south to north inside each tile. The array containing the vertices is stored to disk. Each tile is stored in a separate file, with a header containing some supplemental information.
Figure 5.3: A class diagram, showing the structure of the renderer

5.4 The Renderer

The object structure of the renderer is described in figure 5.3. In this section I will describe the essential algorithms in this program. Most of them are located in the surface object.

For each frame a new triangulation, satisfying an error threshold, is made. Each tile is evaluated. Only the tiles which intersect the view frustum are rendered. For each rendered tile the associated vertex tree is traversed. The vertex tree is traversed recursively from a parent vertex to its children vertices starting at the root vertex. The error metric used ensures that if a vertex is too detailed to be inserted, all of its children will also be too detailed, and the rest of the vertex branch does not need to be further evaluated.

In [2], and [13] a special way of traversing a RIT triangulation is presented for triangle stripping. In [2] a method for traversing a vertex tree and implicitly producing this triangle strip is presented. By using this method, a single triangle strip covering the whole tile is given. A pseudo code for a similar method is given in figure 5.4 This code is slightly modified from the code in [2], but it traverses the vertex tree the same way, and creates the same
triangle strip as in [2], and [13].

I will now explain the code in detail.

tesselate(): The lower left vertex is inserted twice at the beginning of the triangle strip so that the first if test in tStripAppend will not fail. Starting with a diamond creating the coarsest grid, each triangle is evaluated for refinement by calling meshRefine().

meshRefine(triangle*): A triangle is evaluated for refinement by evaluating the vertex which can be inserted at the hypotenuse. For the two triangles creating the coarsest mesh, the root vertex is evaluated. This evaluation is discussed further in section 5.4.1. If a triangle has to be refined, each of the sub triangles are evaluated for further refinement with a call to meshRefine(). The function meshRefine() also appends the vertex at the right angle to the triangle strip with a call to tStripAppend(). This vertex is the vertex which was evaluated in the “parent triangle” at the previous call. The ordering of the calls to a triangle’s sub triangles is essential for the creation of the triangle strip. The triangle children made by inserting a vertex at an even level, (with the vertex root at level 0), are ordered in a counterclockwise manner around the inserted vertex. The sub triangles created by inserting a vertex at an odd level are ordered in a clockwise manner around the inserted vertex. This is illustrated in figure 5.5

tStripAppend(triangle*): This method appends the vertex at the right angle of the triangle in the argument to a triangle strip array. The tests in tStripAppend ensure that the vertex ordering in the triangle strip array is correct, and that no unnecessary vertices are appended.

The triangle strip created in this algorithm traverses the triangulation as shown in figure 5.5.

5.4.1 Triangle Evaluation

In the function meshRefine(), a triangle is evaluated for refinement. This is done by examining the vertex which may be inserted at the center of the hypotenuse of the current triangle.
tesselate()
// Starting with the diamond creating the coarsest mesh,
// each triangle is evaluated for refinement
    TS.append(LowerLeftVertex);
    TS.append(LowerLeftVertex);
    meshRefine(lowerRightTriangle);
    TS.append(UpperRightVertex);
    meshRefine(upperLeftTriangle);
}

meshRefine(triangle* T){
    if (... triangle needs refinement ...){
        meshRefine(T->firstChild());
        tStripAppend(T->rightAngleVertex);
        meshRefine(T->secondChild());
    }
    else{
        tStripAppend(T->rightAngleVertex);
    }
}

tStripAppend(triangle* T){
    if( (... T->rightAngleVertex != TS->lastVertex ...) &&(... T->rightAngleVertex != TS->secondLastVertex...)){
        if(... T->parity == lastVertex->parity ...){
            TS->append(secondLastVertex);
        } else{
            TS->append(T->rightAngleVertex);
        }
    }
}

Figure 5.4: The functions used to traverse a vertex tree during rendering.

47
This is the triangle refinement test used in the function meshRefine() in the program:

\[
\text{if } ((\text{triangle}.\text{level}<\text{maxlevel})\&\&\text{evalTriangle(triangle)})
\]

If the finest detail level has been reached, the triangle cannot be refined. The triangles view space error is computed by using this vertex’ world space error, and the bounding sphere. If the triangle satisfies the screen space error threshold, the triangle will not need to split, and evalTriangle() returns false. If the triangle is too inaccurate and has to be split, evalTriangle() returns true. This function is presented in pseudo code in figure 5.4.1.

This code will now be described in detail. First the tests are described:

**Screen Space error test:** The screen space error is computed by projecting the world space error into screen space. In this implementation a perspective projection is used, as this is the common projection used in computer graphics. To reduce computations, the world space error is projected as if it is parallel to the near cutting plane. Now the problem is reduced to comparing two right angled triangles sharing a common angle. The aspect ratio between two sides in the triangle may be used to compute the view space error in world space coordinates. (See figure 5.7.) D is the distance from the near cutting plane to the point where $\delta$ is projected from.
bool evalTriangle(triangle* T) {
    if(T->centervertex has been evaluated earlier this frame){
        ...use that result by evaluating the vertex status...
    }

    // View culling test
    if(T->centervertex->sphere outside view frustum){
        ...vertex status is set...
        return false;
    }

    // screen space error test
    if (T->centervertex->epsilon < threshold){
        ...vertex status is set...
        return false;
    }

    // If the program has reached this far, the
    // vertex is inside the view frustum, and
    // evaluated to be in the triangulation.
    if((e-threshold)<morphInterval){
        morph vertex
    }
    ...vertex status is set...
    return true;
}

Figure 5.6: Pseudocode for the function evalTriangle()
To convert the view space error in world space coordinates to pixels, \( \hat{\epsilon}_{ws} \) can be multiplied with the pixel scale. The pixel scale is the number of pixels in the height of the rendered picture, divided by the height of the near cutting plane. We assuming that the same pixel scale is used for both the screen width and the screen height.

In order to maintain the property that \( \epsilon_{parent} \geq \epsilon_{child} \), the screen space error of a vertex \( \epsilon_v \) must be set as maximal value of \( \hat{\epsilon} \) when \( \hat{\epsilon} \) is projected from inside the vertex’ bounding sphere.

The distance from the near cutting plane to the vertex has already been computed during view culling described later. From the formula, one can see that \( \hat{\epsilon} \) increases as \( D \) decreases. Therefore \( \hat{\epsilon} \) will be at its maximum when it is projected from the bounding sphere closest to the view cutting plane. So the distance \( D \) which maximizes \( \hat{\epsilon} \) is actually the distance from the near cutting plane to the vertex subtracted with the radius value.

If \( \epsilon \) satisfies the error threshold, the triangle does not have to be refined. If not, the triangle is refined. In addition, if a vertex’ bounding sphere intersects with the near cutting plane, the test is invalid, so the triangle is refined.

When using a perspective projection the denominator may become zero if \( D \) equals -d. This however, is not a problem when view culling is applied, because the vertices where \( D \) equals -d are culled or set active because their sphere partly intersects with the near cutting plane. These situations are therefore not evaluated.

**View frustum Culling:** Tiles outside the view frustum are not drawn. In addition, the parts of the triangulation which are outside the view frustum, in the tiles which intersect the view frustum may be kept at a minimum. The bounding sphere associated with a vertex is set in the preprocess so that it contains all of its’ descendants and all triangles they are part of. By comparing these spheres with the view frustum, the triangulation outside the view frustum may be simplified. This can be done by using the following rules:

- If the vertex’ bounding sphere is outside the viewfrustum, both the vertex itself, and its descendants can be left out.
If the vertex’ bounding sphere is partly inside the viewfrustum, the vertex has to be evaluated further.

- If the vertex’ bounding sphere is completely inside the viewfrustum, both it and all of its descendants are inside the viewfrustum.

For each frame, a function for all six view frustum cutting planes is computed from the projection matrix. These planes are compared with the bounding sphere of the vertex. Pseudo code for view frustum culling is presented in figure 5.4.1:

View culling will leave a valid vertex selection according to 3.4.1. because it will not interfere with property 3.4.4. If a vertex is culled away, all of its descendants shall also be culled away, because a vertex’ bounding sphere completely contains all of its descendants.

This code sets the variable completelyInside to true if the vertex’ bounding sphere is completely inside the viewfrustum. If this is the case, all of the vertex’ children, and all of the triangles sub triangles will be inside the viewfrustum. Therefore the view culling test does not have to be performed on these triangles. This code returns a false value if a vertex is removed from the selection.

Only the vertices whose parents’ bounding spheres are only partly inside the view frustum are evaluated in the view culling test.
// The view culling test in evalTriangle():
if (...parent is completely inside...){
   // no culling evaluation is necessary
   completelyInside = true
}
else{
   completelyInside = true
   for (each view frustum plane){
      float d=distance from point to plane;
      if (d < sphereRadius) completelyInside=false;
      if (d < -sphereRadius) return false;
   }
}
// If the program reaches this far, the vertex shall not be culled,
// and the triangle will be evaluated further.

Figure 5.8: Pseudocode for view frustum culling of vertices in a vertex tree

In order to for a vertex to only be evaluated once per vertex selection, a vertex status array is kept:

**Vertex Status Array:** For each vertex in the vertex buffer, a status integer is kept. When a vertex is evaluated for the first time in a frame, this integer is set using these rules:

- If the vertex is not in the triangulation it is set to the frame number*3
- If the vertex bounding sphere is partly inside the viewfrustum and the vertex is in the triangulation, the integer is set to frame number*3+1
- If the vertex’ bounding sphere is completely inside viewfrustum and the vertex is inside viewfrustum, and the vertex is in the triangulation, the integer is set to frame number*3+2

It is now straightforward to see if a vertex has been evaluated in this frame, and what the result was. As all the vertices at the interior have two parents, they will be evaluated twice, when using the method in figure 5.4, one evaluation from each parent. By using the vertex status array, the vertex evaluations are almost halved. In addition, the view frustum culling is speeded up. By using a running integer instead of a
boolean or enum variable, the status variable does not have to be reset between each frame. Instead the frame number is increased for each frame.

This integer is also used when the vertex buffer is full, and old vertices are removed.

The function evalTriangle() also contains some code for geomorphing.

**Morphing:** A triangle refinement will cause a sudden change in the terrain geometry. If the error threshold is large enough, this may be seen in the rendered picture and the effect is called popping. For larger thresholds, these pops can be quite disturbing. This unwanted effect can be reduced by inserting a vertex more carefully. Instead of placing a new vertex at it’s position, it is first inserted at level with the original surface, causing no visual change in the terrain model. It is then gradually moved to it’s real position. This gives a smoother transition between the geometries. There are two ways of doing this. In [4] and [7] a time based morph is used. When a vertex is inserted, it is moved from the original surface to its position in a given time. Lindstrom uses an error metric driven vertex morph. This is done by introducing a second threshold. When a vertex reaches the error threshold, it is inserted at level with the original surface. When the vertex reaches the second threshold, it will be at its position. When the error is between these two thresholds it is positioned is a linear interpolation between its position and the level of the original surface. The linear interpolation coefficient is given by the vertex’ error value.

The error based morph has been implemented for this application.

### 5.4.2 Vertex Buffer

The renderer uses a vertex buffer. The vertex buffer is an array with x, y, and z coordinates. In addition, similar arrays are kept for the texture coordinates, and for the normal vectors. For each vertex in the model, a vertex buffer index is kept. If a vertex is in the vertex buffer, this variable holds the buffer index of this vertex. If a vertex is not in the vertex buffer, the index is set to -1. By using these arrays, some information is reused from frame to frame. This is an advantage, because most of the vertices in a frame is used in the previous frame. The terrain may be rendered by using vertex arrays.
in OpenGL which is more efficient than rendering in direct mode. When the vertex buffer is full, all vertices which have not been used in this frame are removed from the vertex buffer.
Chapter 6

Results

The author implemented a viewer for a multiresolution terrain model using C++, OpenGL and GLUT, on Linux. To test the program, several fly-throughs have been performed on two datasets. The computer used in all tests is a PIII 1ghz 512 mb ram with a GeForce4 Ti4200 128mb graphics card.

6.1 The Datasets

A dataset of the Grand Canyon is freely available at [20] This dataset contains a height field with dimension 4097x2049 points. The points are sampled regularly with 60 meter between sample points. This is an area of approximately 246x123km and a point set of about 8.4 million points. The dataset also contains a texture of 4096x2048 pixels covering the same area. A new texture of 8192x4096 pixels has also been made to test performance with larger textures.

The second dataset is a height field covering the south of Norway made available to SINTEF from “Statens kartverk” (Norwegian National Survey). The dataset used has a dimension of 4097x6145 datapoints. The points are sampled regularly with 100 meter between sample points. This is an area of 409.7x614.5 km, and a point set of about 25 million points. A one dimensional texture of 1x256 pixels was used when rendering this data. The texture coordinate used is set by the vertex height value.
Figure 6.1: The dataset of the Grand Canyon, with the three base meshes used. The left model uses a 2x1 tiles, the middle 8x4 tiles and the right model uses 32x16 tiles.

Figure 6.2: The dataset of south Norway, with the two base meshes used. The left base mesh uses a 4x6 tiles, and the right base mesh uses 16x24 tiles.
6.2 The Fly-through

In order to test the performance several fly-throughs have been performed. Each fly-through has had some parameters altered. In order to compare the fly-throughs, the camera had to follow the same camera path. To achieve this, a view control module using a spline curve has been developed. The position and orientation of a camera has 9 degrees of freedom. Therefore a 9 dimensional spline curve was used. The curve represents a camera-path which goes smoothly through the landscape. The spline curve is made from a set of control points. This curve is sampled regularly at 840 samples creating 840 camera positions and orientations. Each fly-through renders all 840 positions and computes the following information:

- Time to render all frames.
- Average framerate.
- Average number vertices used in each frame.

This information is presented for all fly-throughs in appendix B.

6.3 The Tests

In order to compare the effect of tiling a domain, several models with different resolution of tiles was made for both the Grand Canyon data set, and the south of Norway data set.

A model was made with the Grand Canyon data set for each of the following tilings. 2x1 tiles with 2049x2049 points, 8x4 tiles with 513x513 points, and 32x16 tiles with 129x129 points. All these models cover the same area. Two series of fly-throughs was performed with each model, one with a texture of 4096x2048 pixels and another with a higher detailed texture of 8192x4096 pixels. Both textures cover the whole domain. The results are presented in figure 6.3.

The solid lines represent fly-throughs which use the texture from the dataset. The dotted lines represent fly-throughs which use the higher resolution texture. One can see from the graph that the model which uses two large tiles
Figure 6.3: Graphs showing renderer performance with the models from Grand Canyon. Dotted lines represent models using a texture with resolution 8192x4096 pixels. Solid lines represent results from models using a texture with resolution 4096x2048.
performs best for large thresholds when the lower resolution texture is used. However, when a higher resolution texture is used, this model’s performance was the weakest. The models tiled at 8x4 performs very well all over. It performs better than the model with 32x16 tiles for error thresholds over about 3 with both textures. For smaller thresholds the 32x16 tile model performs better than both the 8x4 model and the 2x1 model, but with such a small threshold the differences between the models performances are very small.

A fly-through was also made in each of the two models using the dataset from the south of Norway. One model uses 4x6 tiles of 1025x1025 points. The other model uses 16x24 tiles of 257x257 points. The results from the fly-throughs in these models are presented in figure 6.4

This model uses a one dimensional texture of 1x256 pixels. This is a very small texture, and it is not surprising that these fly-throughs give similar results to the fly-throughs in the Grand Canyon dataset with the lowest resolution texture. With a high error threshold the model with the least number of tiles performs the better. However, when the threshold is small, the model with more and smaller tiles is a little faster. When the threshold is small, the difference between the models is very small in this case too.

6.3.1 Conclusion

A single triangle strip is created for each tile. For the models with large tiles, these strips will become very long when the threshold decreases to near one. When several smaller tiles are used, several smaller triangle strips are created and rendered, and this seems to be an advantage at low thresholds. However at such thresholds, the framerates are so low that there are little difference between the models. For large thresholds the ordeal of handling more tiles seem to surpass the advantage of rendering several smaller triangle strips. In addition, when the threshold is large, and several small tiles are used, many of the tiles may be rendered with very little detail, or even as only the two base triangles. These tiles will create triangle strips that are too short to be efficient for rendering.

When a high detailed texture is applied, using large tiles means handling of very large textures. The texture tiles are divided to fit the terrain tiles. When smaller tiles are used, more tiles can be culled, and the tiles drawn fit closer to the view frustum. If one large texture is used over a model with one large
Figure 6.4: Graphs showing rendering performance with the models from the south of Norway.
tile, the entire texture has to be used for each frame even if only a small part of it is visible. If the terrain is divided into smaller tiles, only the textures which are used on the terrain tiles inside the viewfrustum are drawn. By not using more textures than necessary the performance is increased. This is probably why the model with only two large tiles performed badly when the high detailed texture was used. All in all, the models which do not use a high detailed texture perform very similarly for thresholds below 100 frames per second.

6.4 Linear Indexing vs. Hierarchical Z-Indexing

In section 4.2 a new indexing method is introduced. To test the performance of this indexing method, several fly-throughs were performed on the same dataset with the same base mesh. The results from the fly-throughs are presented in figure 6.5.

From the figures we can see that the HZ-indexing method did not perform as well as we hoped for. For the tiles with the size of 257x257 data points, linear indexing is constantly faster than the HZ-indexing. Both methods perform very similarly for the larger tiles, although the linear indexing seems to be a bit more efficient. If this a continuing tendency, using the HZ indexing might be faster when even larger tiles are used.

6.4.1 Conclusion

The HZ-indexing method optimizes page fetches from parts of the virtual memory located on the hard drive. The computer used for testing has enough internal memory to hold the complete data model of the largest terrain model. The advantage of using a HZ-indexing method is therefore not utilized. If the test was performed on a computer with less memory, or a larger data model was used, the results may have been different.
Figure 6.5: These graphs show the difference in performance when using linear indexing and Hierarchical Z indexing. Each graph shows the Hierarchical Z index’ performance in percentage of the linear index’ performance.
Figure 6.6: This figure shows the average number of vertices used per frame in a fly-through for two models using two different world space error estimates. These estimates are described in section 5.3.2

6.5 Testing the World Space Error Metrics

Two error metrices has been used to compute the world space delta value. Both methods are described in section 5.3.2. In figure 6.6 the results of a series of fly-throughs are presented. Error estimate one produces more vertices than error estimate two for all thresholds. When the rendered pictures of both error metrices are compared, it is very difficult to see any difference between the two error estimates. Both produce a selection of vertices which give a good approximation of the highest resolution triangulation.

6.5.1 Conclusion

Error estimate one compares the diamond, of which the vertex is at the center, with the finest resolution grid. When this estimate is used, the screen space error will be an estimate of how many pixels the rendered terrain deviates from a rendering of the finest resolution grid.

Error estimate two simply compares the difference between two consecutive levels. This is a measure of the size of the “popping” produced when the
vertex is inserted in the triangulation. The screen space error used with this world space error will therefore be an estimate of the “popping” in screen space.

Using error estimate one therefore selects the subset of vertices which produces a rendered screen which is closest to a rendering of the finest resolution mesh. Using error estimate two however, produces the terrain which reduces the amount of “popping” in the rendered screen. It is not surprising that the number of vertices needed to render a terrain which cannot differ more than a certain threshold from a rendering of the highest detailed mesh, needs more vertices than a rendering of a terrain which reduces the amount of popping in the geometry.

One difference between the two, is the time used to compute the two errors. Error estimate one compares each triangle at every level with the finest resolution triangulation. This operation needs computation in the order of $O(N \times l)$ where $N$ is the number of vertices in the dataset, and $l$ is the number of levels in the vertex tree. Error estimate number two however, only needs a constant number of computations for each vertex. Hence, the order of computations needed to compute this error is $O(N)$. From this we can see that error estimate two is $l$ times faster to compute than error estimate one.
Chapter 7

Conclusion

In this section some concluding remarks, and a summary of the sub conclusions in section 6 is presented.
The renderer seems to give fairly good results. When a texture is used there is no visible popping when the threshold is set to less than approximately 2. At this threshold the renderer manages to deliver an average of about 35 frames per second for the Grand Canyon dataset and an average of 23 frames per second for the south of Norway data set. When geo-morphing is applied in addition the threshold can be set as high as 4-6 pixels before the insertion of new vertices is noticeable. At this threshold the renderer delivers from 75 frames to more than 100 frames per second, which is more than necessary. Framerates of over 100 frames per second are more of a theoretical value as few computer screens support refresh rates of more than 100 frames per second. The results for the models with a low detail texture show that when looking at the fly-throughs with framerates up to 100 frames per second, all tilings perform very similarly. When higher resolution textures are used, the effect of tiling seems to be more noticeable. The use of textures has, however, not been in focus in this thesis. Before giving a final conclusion to the effect of tiling on models with higher resolution textures, more optimization for use of textures should be done.

The Hierarchical Z-index did not perform as well as we hoped for, and some reasons for this has been suggested. Under other circumstances the results may have been different.
The two error estimates presented in 5.3.2 gives estimates to two different errors in the triangulation. Error estimate one compares a triangulation with the finest resolution grid. Error estimate two however, gives an estimate of the popping in the geometry when inserting a vertex. The visible difference between the two estimates is very small in the rendered screen, and difficult to compare. When using the same screen space error threshold to select the vertices in a triangulation for both world space error estimates, world space error estimate one produces a few more vertices than world space error estimate two. The biggest difference between the two error estimates is the number of computations used to compute them during the preprocessing. Error estimate two is about \( l \) times faster to compute than error estimate one, where \( l \) is the number of levels in the vertex tree.
Appendix A

Screen Shots

This appendix includes some screen shots from the program described in chapter 5.
Figure A.1: Screen shots over Grand Canyon. Both screen shots are taken from the same view, using the same geometry.
Figure A.2: Screen shots over Grand Canyon. The top picture uses the same geometry as both the scenes rendered in figure A.1. The bottom picture shows the base mesh from the same view.
Figure A.3: Screen shots using the data model from the south of Norway.
Figure A.4: A screen shot from the Grand Canyon data model
Appendix B

Tables With Results From Fly-Throughs

South of Norway 4x6 tiles of 1025x1025 elevation points. Hierarchical Z-indexing:

<table>
<thead>
<tr>
<th>Threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.11</td>
<td>164.384</td>
<td>1280</td>
</tr>
<tr>
<td>7.5</td>
<td>6.53</td>
<td>128.637</td>
<td>1844</td>
</tr>
<tr>
<td>5</td>
<td>10.64</td>
<td>78.9474</td>
<td>3215</td>
</tr>
<tr>
<td>4</td>
<td>14.31</td>
<td>58.7002</td>
<td>4445</td>
</tr>
<tr>
<td>3</td>
<td>21.01</td>
<td>39.981</td>
<td>6809</td>
</tr>
<tr>
<td>2.5</td>
<td>27.34</td>
<td>30.7242</td>
<td>8936</td>
</tr>
<tr>
<td>2</td>
<td>37.11</td>
<td>22.6354</td>
<td>12453</td>
</tr>
<tr>
<td>1.7</td>
<td>46.96</td>
<td>17.8867</td>
<td>15839</td>
</tr>
<tr>
<td>1.5</td>
<td>56.08</td>
<td>14.9786</td>
<td>19023</td>
</tr>
<tr>
<td>1.2</td>
<td>77.51</td>
<td>10.8373</td>
<td>26289</td>
</tr>
<tr>
<td>1</td>
<td>102.29</td>
<td>8.21195</td>
<td>33998</td>
</tr>
<tr>
<td>0.7</td>
<td>194.12</td>
<td>4.32722</td>
<td>55192</td>
</tr>
</tbody>
</table>

South of Norway 4x6 tiles of 1025x1025 elevation points. Linear indexing:
<table>
<thead>
<tr>
<th>Threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.12</td>
<td>164.662</td>
<td>1280</td>
</tr>
<tr>
<td>7.5</td>
<td>6.69</td>
<td>125.561</td>
<td>1844</td>
</tr>
<tr>
<td>5</td>
<td>10.49</td>
<td>80.0763</td>
<td>3215</td>
</tr>
<tr>
<td>4</td>
<td>14.03</td>
<td>59.8717</td>
<td>4445</td>
</tr>
<tr>
<td>3</td>
<td>20.56</td>
<td>40.856</td>
<td>6809</td>
</tr>
<tr>
<td>2.5</td>
<td>26.52</td>
<td>31.6742</td>
<td>8936</td>
</tr>
<tr>
<td>2</td>
<td>36.14</td>
<td>23.2108</td>
<td>12453</td>
</tr>
<tr>
<td>1.7</td>
<td>45.21</td>
<td>18.58</td>
<td>15839</td>
</tr>
<tr>
<td>1.5</td>
<td>53.74</td>
<td>15.6308</td>
<td>19023</td>
</tr>
<tr>
<td>1.2</td>
<td>74.1</td>
<td>11.336</td>
<td>26289</td>
</tr>
<tr>
<td>1</td>
<td>97.81</td>
<td>8.58808</td>
<td>33998</td>
</tr>
<tr>
<td>0.7</td>
<td>185.78</td>
<td>4.52148</td>
<td>55192</td>
</tr>
</tbody>
</table>
South of Norway 16x24 tiles 257x257 elevation points. Hierarchical z-indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.46</td>
<td>153.846</td>
<td>1229</td>
</tr>
<tr>
<td>7.5</td>
<td>7.12</td>
<td>117.978</td>
<td>1805</td>
</tr>
<tr>
<td>5</td>
<td>11.05</td>
<td>76.0181</td>
<td>3210</td>
</tr>
<tr>
<td>4</td>
<td>14.57</td>
<td>57.6527</td>
<td>4467</td>
</tr>
<tr>
<td>3</td>
<td>21.57</td>
<td>38.943</td>
<td>6887</td>
</tr>
<tr>
<td>2.5</td>
<td>27.66</td>
<td>30.3688</td>
<td>9060</td>
</tr>
<tr>
<td>2</td>
<td>37.76</td>
<td>22.2458</td>
<td>12648</td>
</tr>
<tr>
<td>1.7</td>
<td>47.26</td>
<td>17.774</td>
<td>16098</td>
</tr>
<tr>
<td>1.5</td>
<td>56.36</td>
<td>14.9068</td>
<td>19340</td>
</tr>
<tr>
<td>1.2</td>
<td>77.23</td>
<td>10.8766</td>
<td>26723</td>
</tr>
<tr>
<td>1</td>
<td>100.47</td>
<td>8.3607</td>
<td>34542</td>
</tr>
<tr>
<td>0.7</td>
<td>182.19</td>
<td>4.61</td>
<td>56006</td>
</tr>
</tbody>
</table>

South of Norway 16x24 tiles 257x257 elevation points. linear indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.28</td>
<td>159.091</td>
<td>1229</td>
</tr>
<tr>
<td>7.5</td>
<td>6.9</td>
<td>121.739</td>
<td>1805</td>
</tr>
<tr>
<td>5</td>
<td>10.55</td>
<td>79.6208</td>
<td>3210</td>
</tr>
<tr>
<td>4</td>
<td>14.08</td>
<td>59.6591</td>
<td>4467</td>
</tr>
<tr>
<td>3</td>
<td>20.36</td>
<td>41.2574</td>
<td>6887</td>
</tr>
<tr>
<td>2.5</td>
<td>26.38</td>
<td>31.8423</td>
<td>9060</td>
</tr>
<tr>
<td>2</td>
<td>35.7</td>
<td>23.5294</td>
<td>12648</td>
</tr>
<tr>
<td>1.7</td>
<td>44.75</td>
<td>18.771</td>
<td>16098</td>
</tr>
<tr>
<td>1.5</td>
<td>53.37</td>
<td>15.7392</td>
<td>19340</td>
</tr>
<tr>
<td>1.2</td>
<td>72.24</td>
<td>11.6279</td>
<td>26723</td>
</tr>
<tr>
<td>1</td>
<td>94.43</td>
<td>8.89548</td>
<td>34542</td>
</tr>
<tr>
<td>0.7</td>
<td>171.22</td>
<td>4.90597</td>
<td>56006</td>
</tr>
</tbody>
</table>
Grand Canyon 2x1 tiles of 2049x2049 elevation points. Hierarchical z-indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.02</td>
<td>208.955</td>
<td>1045</td>
</tr>
<tr>
<td>7.5</td>
<td>4.8</td>
<td>175</td>
<td>1464</td>
</tr>
<tr>
<td>5</td>
<td>7.66</td>
<td>109.661</td>
<td>2428</td>
</tr>
<tr>
<td>4</td>
<td>9.97</td>
<td>84.2528</td>
<td>3214</td>
</tr>
<tr>
<td>3</td>
<td>14.4</td>
<td>58.3333</td>
<td>4624</td>
</tr>
<tr>
<td>2.5</td>
<td>18.21</td>
<td>46.1285</td>
<td>5818</td>
</tr>
<tr>
<td>2.0</td>
<td>23.46</td>
<td>35.8056</td>
<td>7737</td>
</tr>
<tr>
<td>1.7</td>
<td>28.48</td>
<td>29.4944</td>
<td>9522</td>
</tr>
<tr>
<td>1.5</td>
<td>33.48</td>
<td>25.0896</td>
<td>11166</td>
</tr>
<tr>
<td>1.2</td>
<td>43.52</td>
<td>19.3015</td>
<td>14766</td>
</tr>
<tr>
<td>1</td>
<td>55.78</td>
<td>15.0592</td>
<td>18542</td>
</tr>
<tr>
<td>0.7</td>
<td>88.76</td>
<td>9.4733</td>
<td>28855</td>
</tr>
</tbody>
</table>

Grand Canyon 2x1 tiles of 2049x2049 elevation points. Linear indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.55</td>
<td>184.615</td>
<td>1045</td>
</tr>
<tr>
<td>7.5</td>
<td>4.64</td>
<td>148.936</td>
<td>1464</td>
</tr>
<tr>
<td>5</td>
<td>8.54</td>
<td>98.3607</td>
<td>2428</td>
</tr>
<tr>
<td>4</td>
<td>10.97</td>
<td>76.5725</td>
<td>3214</td>
</tr>
<tr>
<td>3</td>
<td>14.92</td>
<td>56.3003</td>
<td>4624</td>
</tr>
<tr>
<td>2.5</td>
<td>18.7</td>
<td>44.9198</td>
<td>5818</td>
</tr>
<tr>
<td>2.0</td>
<td>24.19</td>
<td>34.7251</td>
<td>7737</td>
</tr>
<tr>
<td>1.7</td>
<td>28.96</td>
<td>29.0055</td>
<td>9522</td>
</tr>
<tr>
<td>1.5</td>
<td>33.74</td>
<td>24.8963</td>
<td>11166</td>
</tr>
<tr>
<td>1.2</td>
<td>43.78</td>
<td>19.1868</td>
<td>14766</td>
</tr>
<tr>
<td>1</td>
<td>55.59</td>
<td>15.1106</td>
<td>18542</td>
</tr>
<tr>
<td>0.7</td>
<td>88.39</td>
<td>9.4733</td>
<td>28855</td>
</tr>
</tbody>
</table>
Grand Canyon 8x4 tiles of 513x513 elevation points. Hierarchical z-indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.99</td>
<td>210.526</td>
<td>1039</td>
</tr>
<tr>
<td>7.5</td>
<td>4.94</td>
<td>170.04</td>
<td>1470</td>
</tr>
<tr>
<td>5</td>
<td>7.58</td>
<td>110.818</td>
<td>2455</td>
</tr>
<tr>
<td>4</td>
<td>10.43</td>
<td>80.5369</td>
<td>3256</td>
</tr>
<tr>
<td>3</td>
<td>14.67</td>
<td>57.2597</td>
<td>4688</td>
</tr>
<tr>
<td>2.5</td>
<td>18.2</td>
<td>46.1538</td>
<td>5898</td>
</tr>
<tr>
<td>2.0</td>
<td>23.52</td>
<td>35.7143</td>
<td>7840</td>
</tr>
<tr>
<td>1.7</td>
<td>28.66</td>
<td>29.3091</td>
<td>9642</td>
</tr>
<tr>
<td>1.5</td>
<td>33.49</td>
<td>25.0821</td>
<td>11301</td>
</tr>
<tr>
<td>1.2</td>
<td>43.34</td>
<td>19.3816</td>
<td>14932</td>
</tr>
<tr>
<td>1</td>
<td>54.63</td>
<td>15.3762</td>
<td>18736</td>
</tr>
<tr>
<td>0.7</td>
<td>83.94</td>
<td>10.0071</td>
<td>29113</td>
</tr>
</tbody>
</table>

Grand Canyon 32x16 tiles of 129x129 elevation points. Hierarchical z-indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.61</td>
<td>182.213</td>
<td>980</td>
</tr>
<tr>
<td>7.5</td>
<td>5.67</td>
<td>148.148</td>
<td>1429</td>
</tr>
<tr>
<td>5</td>
<td>8.17</td>
<td>102.815</td>
<td>2460</td>
</tr>
<tr>
<td>4</td>
<td>10.59</td>
<td>79.3201</td>
<td>3298</td>
</tr>
<tr>
<td>3</td>
<td>14.63</td>
<td>57.3379</td>
<td>4796</td>
</tr>
<tr>
<td>2.5</td>
<td>18.29</td>
<td>45.9267</td>
<td>6055</td>
</tr>
<tr>
<td>2</td>
<td>23.62</td>
<td>35.5631</td>
<td>8067</td>
</tr>
<tr>
<td>1.7</td>
<td>28.25</td>
<td>29.7345</td>
<td>9933</td>
</tr>
<tr>
<td>1.5</td>
<td>32.72</td>
<td>25.6724</td>
<td>11695</td>
</tr>
<tr>
<td>1.2</td>
<td>42.97</td>
<td>19.5485</td>
<td>15385</td>
</tr>
<tr>
<td>1</td>
<td>52.99</td>
<td>15.852</td>
<td>19290</td>
</tr>
<tr>
<td>0.7</td>
<td>82.04</td>
<td>10.2389</td>
<td>29915</td>
</tr>
</tbody>
</table>
Grand Canyon 2x1 tiles of 2049x2049 elevation points. Hierarchical z-indexing. Using a texture of 8192x4096 pixels:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7.28</td>
<td>115.385</td>
<td>1045</td>
</tr>
<tr>
<td>7.5</td>
<td>8.78</td>
<td>95.672</td>
<td>1464</td>
</tr>
<tr>
<td>5</td>
<td>11.39</td>
<td>73.7489</td>
<td>2428</td>
</tr>
<tr>
<td>4</td>
<td>13.73</td>
<td>61.1799</td>
<td>3214</td>
</tr>
<tr>
<td>3</td>
<td>17.82</td>
<td>47.138</td>
<td>4624</td>
</tr>
<tr>
<td>2.5</td>
<td>21.22</td>
<td>39.5853</td>
<td>5818</td>
</tr>
<tr>
<td>2</td>
<td>27.21</td>
<td>30.871</td>
<td>7737</td>
</tr>
<tr>
<td>1.7</td>
<td>32.51</td>
<td>25.8362</td>
<td>9522</td>
</tr>
<tr>
<td>1.5</td>
<td>37.92</td>
<td>22.1519</td>
<td>11166</td>
</tr>
<tr>
<td>1.2</td>
<td>48.37</td>
<td>17.3661</td>
<td>14766</td>
</tr>
<tr>
<td>1</td>
<td>60.89</td>
<td>13.7954</td>
<td>18542</td>
</tr>
<tr>
<td>0.7</td>
<td>98.14</td>
<td>8.5592</td>
<td>28855</td>
</tr>
</tbody>
</table>

Grand Canyon 8x4 tiles of 513x513 elevation points. Hierarchical z-indexing. Using a texture of 8192x4096 pixels:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.85</td>
<td>173.196</td>
<td>1039</td>
</tr>
<tr>
<td>7.5</td>
<td>6.24</td>
<td>134.615</td>
<td>1470</td>
</tr>
<tr>
<td>5</td>
<td>9.05</td>
<td>92.8177</td>
<td>2455</td>
</tr>
<tr>
<td>4</td>
<td>11.51</td>
<td>72.98</td>
<td>3256</td>
</tr>
<tr>
<td>3</td>
<td>15.63</td>
<td>53.7428</td>
<td>4688</td>
</tr>
<tr>
<td>2.5</td>
<td>18.84</td>
<td>44.586</td>
<td>5898</td>
</tr>
<tr>
<td>2</td>
<td>24.9</td>
<td>33.7349</td>
<td>7840</td>
</tr>
<tr>
<td>1.7</td>
<td>30.26</td>
<td>27.7594</td>
<td>9642</td>
</tr>
<tr>
<td>1.5</td>
<td>34.97</td>
<td>24.0206</td>
<td>11301</td>
</tr>
<tr>
<td>1.2</td>
<td>45.62</td>
<td>18.413</td>
<td>14932</td>
</tr>
<tr>
<td>1</td>
<td>56.57</td>
<td>14.8489</td>
<td>18736</td>
</tr>
<tr>
<td>0.7</td>
<td>86.68</td>
<td>9.69082</td>
<td>29113</td>
</tr>
</tbody>
</table>
Grand Canyon 32x16 tiles of 129x129 elevation points. Hierarchical z-indexing. Using textures of 8192x4096 pixels:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5.55</td>
<td>151.351</td>
<td>980</td>
</tr>
<tr>
<td>7.5</td>
<td>6.66</td>
<td>126.126</td>
<td>1429</td>
</tr>
<tr>
<td>5</td>
<td>9.59</td>
<td>87.5912</td>
<td>2460</td>
</tr>
<tr>
<td>4</td>
<td>11.94</td>
<td>70.3518</td>
<td>3298</td>
</tr>
<tr>
<td>3</td>
<td>15.9</td>
<td>52.8302</td>
<td>4796</td>
</tr>
<tr>
<td>2.5</td>
<td>19.5</td>
<td>43.0769</td>
<td>6055</td>
</tr>
<tr>
<td>2.0</td>
<td>25.2</td>
<td>33.3333</td>
<td>8067</td>
</tr>
<tr>
<td>1.7</td>
<td>30.22</td>
<td>27.7962</td>
<td>9933</td>
</tr>
<tr>
<td>1.5</td>
<td>35.19</td>
<td>23.8704</td>
<td>11645</td>
</tr>
<tr>
<td>1.2</td>
<td>45.39</td>
<td>18.5063</td>
<td>15385</td>
</tr>
<tr>
<td>1</td>
<td>55.82</td>
<td>15.0484</td>
<td>19290</td>
</tr>
<tr>
<td>0.7</td>
<td>85.1</td>
<td>9.87074</td>
<td>29915</td>
</tr>
</tbody>
</table>

Grand Canyon 2x1 tiles of 2049x2049 elevation points. Hierarchical z-indexing. Using world space estimate number 2:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.94</td>
<td>170.04</td>
<td>1243</td>
</tr>
<tr>
<td>7.5</td>
<td>6.46</td>
<td>130.031</td>
<td>1747</td>
</tr>
<tr>
<td>5</td>
<td>10.13</td>
<td>82.922</td>
<td>2914</td>
</tr>
<tr>
<td>4</td>
<td>13.13</td>
<td>63.9756</td>
<td>3858</td>
</tr>
<tr>
<td>3</td>
<td>17.92</td>
<td>46.875</td>
<td>5511</td>
</tr>
<tr>
<td>2.5</td>
<td>22.28</td>
<td>32.702</td>
<td>6901</td>
</tr>
<tr>
<td>2</td>
<td>29.01</td>
<td>28.9555</td>
<td>9135</td>
</tr>
<tr>
<td>1.7</td>
<td>35.04</td>
<td>23.9726</td>
<td>11168</td>
</tr>
<tr>
<td>1.5</td>
<td>40.54</td>
<td>20.7203</td>
<td>13011</td>
</tr>
<tr>
<td>1.2</td>
<td>653.54</td>
<td>15.6892</td>
<td>17082</td>
</tr>
<tr>
<td>1</td>
<td>67.56</td>
<td>12.4334</td>
<td>21290</td>
</tr>
<tr>
<td>0.7</td>
<td>107.51</td>
<td>7.81323</td>
<td>32643</td>
</tr>
</tbody>
</table>
South of Norway 1 tile of 4096x4096 elevation points. Linear indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4.11</td>
<td>204.38</td>
<td>947</td>
</tr>
<tr>
<td>7.5</td>
<td>5.07</td>
<td>165.68</td>
<td>1331</td>
</tr>
<tr>
<td>5</td>
<td>8.02</td>
<td>104.738</td>
<td>2272</td>
</tr>
<tr>
<td>4</td>
<td>10.41</td>
<td>80.6915</td>
<td>3120</td>
</tr>
<tr>
<td>3</td>
<td>15.21</td>
<td>55.2268</td>
<td>4749</td>
</tr>
<tr>
<td>2.5</td>
<td>19.35</td>
<td>43.4109</td>
<td>6239</td>
</tr>
<tr>
<td>2.0</td>
<td>26.25</td>
<td>32</td>
<td>8748</td>
</tr>
<tr>
<td>1.7</td>
<td>32.94</td>
<td>25.5009</td>
<td>11214</td>
</tr>
<tr>
<td>1.5</td>
<td>39.18</td>
<td>21.4395</td>
<td>13556</td>
</tr>
<tr>
<td>1.2</td>
<td>54.74</td>
<td>15.3453</td>
<td>18995</td>
</tr>
<tr>
<td>1</td>
<td>73.65</td>
<td>11.4053</td>
<td>24935</td>
</tr>
<tr>
<td>0.8</td>
<td>107.71</td>
<td>7.79872</td>
<td>34606</td>
</tr>
</tbody>
</table>

South of Norway 1 tile of 4096x4096 elevation points. Hierarchical z-indexing:

<table>
<thead>
<tr>
<th>threshold</th>
<th>time</th>
<th>framerate</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.99</td>
<td>210.526</td>
<td>947</td>
</tr>
<tr>
<td>7.5</td>
<td>5.16</td>
<td>162.791</td>
<td>1331</td>
</tr>
<tr>
<td>5</td>
<td>7.99</td>
<td>105.131</td>
<td>2272</td>
</tr>
<tr>
<td>4</td>
<td>10.61</td>
<td>79.1706</td>
<td>3120</td>
</tr>
<tr>
<td>3</td>
<td>15.45</td>
<td>54.3689</td>
<td>4749</td>
</tr>
<tr>
<td>2.5</td>
<td>19.81</td>
<td>42.4028</td>
<td>6239</td>
</tr>
<tr>
<td>2</td>
<td>27.04</td>
<td>31.0651</td>
<td>8748</td>
</tr>
<tr>
<td>1.7</td>
<td>33.98</td>
<td>24.7204</td>
<td>11214</td>
</tr>
<tr>
<td>1.5</td>
<td>41.02</td>
<td>20.4778</td>
<td>13565</td>
</tr>
<tr>
<td>1.2</td>
<td>57.11</td>
<td>14.7085</td>
<td>18995</td>
</tr>
<tr>
<td>1</td>
<td>77.13</td>
<td>10.8907</td>
<td>24935</td>
</tr>
<tr>
<td>0.8</td>
<td>112.85</td>
<td>7.44351</td>
<td>34606</td>
</tr>
</tbody>
</table>
Bibliography


