

ABSTRACT

Terrain in flight simulators has traditionally been hand-modelled by artists. Ericsson Saab Avionics AB uses a hand-modelled terrain over Gotland for their flight simulator *T³SIM*. Hand-modelled terrain is expensive and has no support for continuous level of detail. This paper presents different methods for automatic terrain generation in real-time based on height data. Several algorithms have been proposed the past few years. However, only two are capable of real-time rendering with current demands on quality and speed. Those are *view-dependent progressive meshes* (VDPM) and *real-time optimally adapting meshes* (ROAM). Although, ROAM restricts the space of possible meshes, its execution time is proportional to the number of triangle changes per frame while the execution time of VDPM is only proportional to the full output mesh size. Some improvements and extensions to ROAM are presented, including a force-merge approach for merging arbitrary triangles. A prototype based on ROAM and the extensions has been implemented, which has shown to reduce the triangle-count compared to traditional methods.

Keywords: terrain, continuous level of detail, multiresolution modelling, surface simplification, triangulation, triangulated irregular network

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Krister Dackland at Umeå University for reading the thesis and giving me valuable comments. I would also like to thank my supervisors Tomas Stenarsson and Torbjörn Söderman at Ericsson Saab Avionics AB for assisting me during my work and for reading the paper. They have also given me valuable comments.

I am grateful to Hugues Hoppe and Mark Duchaineau for answering my questions about view-dependent progressive meshes and real-time optimally adapting meshes, respectively.

CONTENTS

1	INTRODUCTION	1
1.1	BACKGROUND	1
1.2	PROBLEM DESCRIPTION	2
1.3	OUTLINE	3
2	SURVEY OF TERRAIN-RENDERING ALGORITHMS.....	5
2.1	REGULAR GRID METHODS	6
2.2	HIERARCHICAL SUBDIVISION METHODS.....	7
2.3	FEATURE METHODS.....	7
2.4	REFINEMENT METHODS.....	7
2.5	DECIMATION METHODS.....	8
2.6	OPTIMAL METHODS.....	8
3	PROGRESSIVE MESHES.....	9
3.1	DEFINITIONS	9
3.2	PROGRESSIVE MESH REPRESENTATION	9
3.3	PROGRESSIVE MESH CONSTRUCTION	10
3.3.1	<i>Energy Function</i>	10
3.3.2	<i>Merge Tree</i>	11
3.3.3	<i>Quadric Error Metric</i>	11
3.4	SELECTIVE REFINEMENT AND COARSENING	11
3.4.1	<i>Legality of Edge Collapse</i>	12
3.5	VERTEX MORPHING.....	13
3.6	VIEW-DEPENDENT REFINEMENT OF PROGRESSIVE MESHES	14
3.6.1	<i>Overview</i>	14
3.6.2	<i>Selective Refinement Algorithm</i>	15
4	REAL-TIME OPTIMALLY ADAPTING MESHES.....	17
4.1	DEFINITIONS	17
4.2	ROAM REPRESENTATION	17
4.3	VERTEX MORPHING.....	18
4.4	TRIANGULATION ALGORITHMS	19
4.5	QUEUE PRIORITIES.....	21
4.6	PERFORMANCE ENHANCEMENTS	22
4.6.1	<i>View-Frustum Culling</i>	22
4.6.2	<i>Incremental triangle stripping</i>	22
4.6.3	<i>Priority-computation Deferral</i>	22
4.6.4	<i>Progressive Optimisation</i>	22
4.7	COMPARISON TO VDPM.....	23
5	ROAM ALGORITHM AND IMPLEMENTATION ASPECTS	24
5.1	SPLIT ALGORITHM.....	24
5.2	MERGE ALGORITHM	25
5.3	MAIN ALGORITHM.....	27
5.4	IMPLEMENTATION ASPECTS.....	28
5.4.1	<i>Bintree Data Structure</i>	28
5.4.2	<i>Implicit Binary Tree</i>	28
5.4.3	<i>Split and Merge Operations</i>	28
5.4.4	<i>Split and Merge Queues</i>	28
5.4.5	<i>View-Frustum Culling</i>	29
5.4.6	<i>Triangle Count</i>	29
6	RESULTS.....	30
6.1	PERFORMANCE	31
6.1.1	<i>Frame Rates</i>	31

VI Contents

6.1.2	<i>Function Timing</i>	32
7	SUMMARY AND CONCLUSION	33
	REFERENCES	34
	APPENDIX A: COMPUTATIONAL GEOMETRY CONCEPTS	A-1
A.1	QUADTREE	A-1
A.2	KD-TREE	A-1
A.3	RANGE TREE	A-2
A.4	VORONOI DIAGRAM	A-2
A.5	POLYGON TRIANGULATION	A-4
A.5.1	<i>Partitioning a Simple Polygon into Monotone Pieces</i>	A-4
A.5.2	<i>Triangulating a Monotone Polygon</i>	A-4
A.6	DELAUNAY TRIANGULATION	A-5
A.7	DATA-DEPENDENT TRIANGULATION	A-7
	APPENDIX B: USER MANUAL	A-8
	APPENDIX C: GLOSSARY	A-9

1 INTRODUCTION

A *terrain* may be defined as the graph of a continuous function $f: \mathbf{R}^2 \rightarrow \mathbf{R}$ [2]. In computer graphics, these graphs are usually represented by *triangle meshes*, i.e. connected sets of triangles in which no more than two triangles share an edge. It is not uncommon that a triangle mesh consists of several millions of triangles. No graphics workstation of today is capable of rendering such large triangle meshes at interactive frame rates. Therefore, the triangle count of large triangle meshes has to be reduced without deteriorating the visual appearance.

Traditional triangle-reduction methods use a small set of *discrete levels of detail* that each represents the same object at different number of triangles. Simple visualisation systems compute the distance from the observer to an object and choose a level of detail for the entire object based on that distance. Other visualisation systems base the level of detail on the screen-space error of the object. If the distance or error changes beyond a certain limit, the whole object is rendered with another level of detail.

There are two problems with triangle-reduction methods based on discrete levels of detail. First, large objects that may have some regions close to the observer and others more distant should be rendered at different levels of detail for the different regions. Terrain is an example of such an object. The horizon does not need to be rendered with as high detail as nearby parts. Second, changing from one level of detail to another leads to temporal aliasing artefacts known as *popping*. Three methods have been proposed to reduce the popping effects. A large number of levels of detail could be used. This reduces the difference between two consecutive levels of detail, but requires large storage space. Another alternative is to use very complex models with small screen-space errors. Although the difference between two consecutive levels of detail is large, the visible difference is small. However, this consumes unnecessary large amounts of rendering power. The last method *morphs* one level of detail to another, i.e. it either animates the vertex positions or *blends* in the new level of detail through several frames. This is also a computationally expensive task.

Thus, triangle-reduction methods based on discrete level of detail are inadequate for terrain visualisation. Algorithms that support *continuous level of detail* address these problems by computing the appropriate level of detail for every triangle at each frame. Lindstrom et al. [24] give a more precise definition of continuous level of detail. Many of these algorithms exploit frame coherence to minimise the difference in triangulation of the terrain in the two frames.

1.1 Background

Ericsson Saab Avionics AB develops T³SIM (Training and Tactical/Technical Development Simulation System), a software system designed for tactical training in real-time man-in-the-loop air-combat simulations. Ericsson Saab Avionics AB uses the system in their EPSIM facility, shown in Figure 1.1, for prototyping, demonstrations, and evaluations in the development process. It is developed in co-operation with the Swedish Air Force Air Combat Simulation Centre, who uses it for training and tactical development for various missions and threat scenarios that involve intercept elements [8, 9].

T³SIM is intended to perform multi-player air-combat simulations and supports functionality for tactical environment with computer-generated platforms, electronic warfare, combat command, and combat control. The system consists of a core system and five modules: master system control module, pilot station module, exercise observation and evaluation module, combat command and control module, and computer-generated forces module. These modules are operated on SGI computer workstations [9].



Figure 1.1: A view of the EPSIM facility at Ericsson Saab Avionics AB.

Today, T³SIM is primarily intended to perform air-combat simulations beyond visual range for JAS 39 Gripen. This requires focus on the tactical instruments, flight-data display, horizontal-situation display, multi-sensor display, and head-up display. Figure 1.2 shows these instruments in the cockpit of a JAS 39 Gripen aircraft. The flight data display presents basic information such as attitude, speed, and altitude, while the tactical information is presented on the horizontal situation display by symbols representing friends, foes, targets, threats, obstacles, and guiding information, all of which is superimposed on a digital electronic map. The multi-sensor display provides different modes of sensor information, including air-to-air radar modes. Finally, the head-up display provides information for navigation and weapons control. It simulates a holographic diffraction optics combiner that presents the information within the pilot's field of view. Since the tactical instruments are the main information providers, the visual presentation of surrounding environment has been lower prioritised [7, 9].

1.2 Problem description

Future versions of T³SIM will include air-to-air and air-to-land combat simulations within visual range. This increases the requirements of the visual system. The terrain will play a major role in future simulations, since air-combat will take place at lower altitudes. Therefore, the system will require two features not available today:

- Availability to simulate terrain for any region in Sweden and other countries of interest.
- Support of high-quality terrain during low-altitude flights at constant frame rates.

The first requirement enables air-combat training in any region for which height and map data is available, while the second requires terrain rendered with continuous level of detail.

The terrain used today is a hand-made model of Gotland, which is shown in Figure 1.3. Most regions are only modelled as a flat landscape of a coarse triangle mesh textured with flight photographs. While this is acceptable for high-altitude flights, it does not provide the visual quality required for future low-altitude flights.



Figure 1.2: On the left is the flight data display, on the middle the horizontal situation display, on the right the multi-sensor display, and straight ahead is the head-up display.

It has not only been very expensive and time-consuming to build this model; it does also limit the application to Gotland. In addition, T³SIM lacks support of continuous level-of-detail rendering, which restricts the terrain to very flat areas. Rougher terrain would require more and smaller underlying triangles, which are too computationally expensive to render.

A solution to the problems is to develop a program that generates terrain from height and map data. The National Land Survey of Sweden provides height data in square areas of 50×50 kilometres, stored as height samples in a regular grid with 50 metres spacing. The map data is stored in layers, each describing a vegetation type, such as woods and lakes, represented by two-dimensional polygons that encapsulate the vegetation type. Other map information such as roads is stored as lines. Finally, some information is described as points, e.g. houses and lighthouses.

This program could read the height samples and map data and, given a position and view direction of the observer, generate a three-dimensional terrain in real time that the observer would see. This thesis is restricted to terrain rendering based on height data. The purpose is to investigate the algorithms that create a triangle mesh that supports continuous level of detail and build a prototype based on one of these algorithms.

1.3 Outline

Section 2 discusses the requirements of terrain-rendering algorithms and surveys existing algorithms. The desired properties of these algorithms are documented. The algorithms are categorised into six groups, each with its own advantages and drawbacks. Two algorithms are found to be candidates for

further investigation, view-dependent progressive meshes (VDPM) and real-time optimally adapting meshes (ROAM). These are covered in Section 3 and 4.



Figure 1.3: A view of the modelled terrain of Gotland in T³SIM.

Section 3 describes progressive meshes in detail. Progressive meshes have been developed by Hugues Hoppe [17, 18, 19, 20, 21] to produce a triangulated irregular network from a set of height samples. Progressive meshes have been shown to minimise the number of triangles necessary to approximate a height field at a given error bound.

Section 4 discusses real-time optimally adapting meshes. ROAM was developed by Duchaineau et al. [6] in 1997. The triangles are represented hierarchically in a binary tree, which facilitates fast retrieval of individual triangles and view-frustum culling. The neighbours to each triangle are stored in every triangle node in order to avoid cracks by force-splitting triangles. Two optimisation methods are introduced with ROAM, incremental triangle stripping and priority-computation deferral lists.

ROAM has shown to be more suitable for real-time terrain rendering. A prototype has been developed that implements a version of ROAM. Section 5 discusses the implementation. An algorithm has been developed that introduces a new, more general, version of the merge operation. This algorithm is discussed together with other algorithms that control the split and merge process.

Results from the implementation are given in Section 6.

In Appendix A, some fundamental computational geometry definitions and concepts are explained. Several terrain-rendering algorithms use data types based on quadtrees, kd-trees and range trees. These are shortly explained together with a time and space complexity summary. Three common triangulation methods, Delaunay triangulation, data-dependent triangulation, and polygon triangulation are also covered.

Appendix B presents a short user manual for the prototype that has implemented the ROAM algorithm together with the improvements and extensions presented in Section 5.

Appendix C contains a glossary of common concepts within the field of computational geometry and terrain rendering. Only selected words are included.

2 SURVEY OF TERRAIN-RENDERING ALGORITHMS

This Section surveys previous efforts in the area of terrain-rendering algorithms based on regular grids of equally spaced height samples. A terrain-rendering algorithm should maximise the visual quality of the terrain at interactive frame rates. More specifically, the following properties should be included:

- Support of several levels of detail for different regions of the terrain simultaneously.
- Avoidance of cracks and shading discontinuities between regions of different levels of detail.
- Minimisation of object-space or screen-space error bounds.
- Reduction of popping artefacts.
- Maintenance of strict frame rates.
- Minimisation of the number of rendered triangles.
- Compact representation of the triangle mesh with support of fast retrieval of individual triangles.
- Capability of creating long triangle strips or fans.
- Support of view-frustum, backface and occlusion culling.
- Exploitation of frame-to-frame coherence in order to minimise triangulation time.
- Reduction of execution-time overhead.
- Reduction of memory requirements.

The first four properties improve the visual quality while the latter eight improves the performance. Most important is support of dynamic continuous level of detail. The level of detail should be based on both the position of the observer and the structure of the terrain. Distant regions of the terrain will have less impact on the final visual result because of perspective foreshortening. Unnecessary amounts of geometry for distant regions are more of a liability than an asset. Not only do such geometry consume valuable rendering power; it can also produce z-buffer accuracy problems and aliasing artefacts [4]. Thus, distant or flat terrain should be simplified more than nearby or rough terrain.

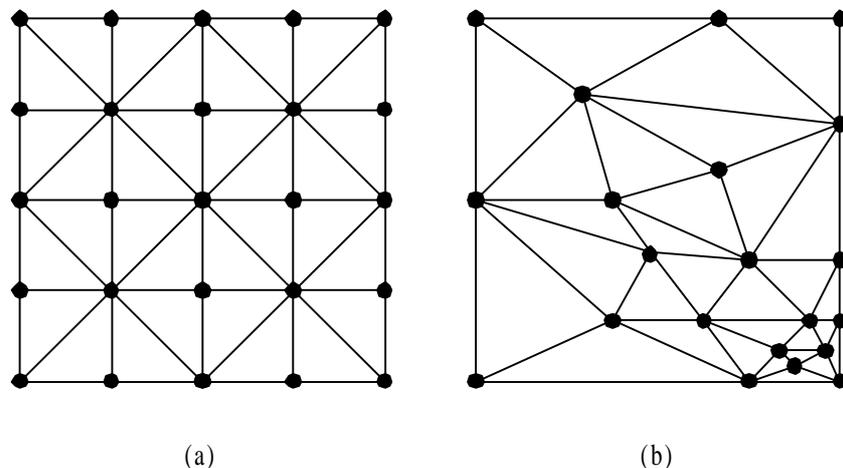


Figure 2.1: a) Top view of a regular triangulation of a height field considering neither view position or terrain structure. b) Top view of a triangulated irregular network considering both view position and terrain structure.

Common methods divide the terrain into square areas known as *tiles*, which are stored at a number of different levels of detail. Nearby tiles are visualised at higher level of detail than distant ones. An example is illustrated in Figure 2.2 (a).

This does however introduce problems at the boundary of the tiles. Not only are the transitions to other levels of detail visible for the observer; they also introduce cracks and shading discontinuities [6, 31]. This can be avoided by ensuring that the projection to the x-y plane of an edge of one triangle contains no vertices of the projection of other triangles. A common method is to sew the tiles as in

Figure 2.2 (b). This solution also reduces the visible differences between different levels of detail. However, the algorithm does not consider the structure of the terrain within each tile. A region within a tile may need more geometry than the rest of the tile. A tile-based algorithm cannot accomplish this.

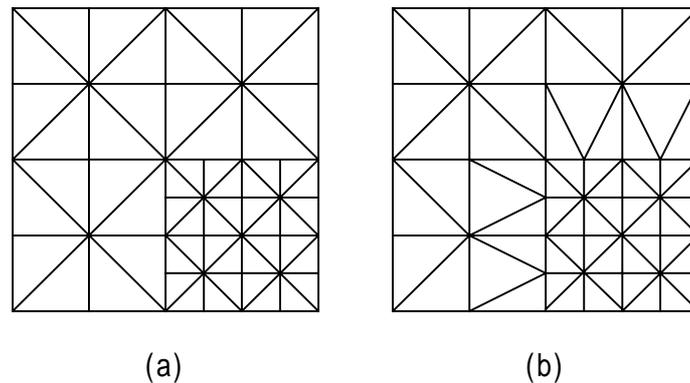


Figure 2.2: a) Subdivision of terrain into tiles. b) The same subdivision, but this time tiles are sewed at the boundary.

Heckbert and Garland [16] have categorised polygonal-surface approximation algorithms based on a regular grid of height samples as follows:

- *Regular grid methods* use a subgrid of equally and periodically spaced height samples from the original grid.
- *Hierarchical subdivision methods* are based on quadtrees, kd-trees, and other hierarchical triangulations. They all use a divide-and-conquer strategy that divides the terrain into smaller regions in a recursive manner to build a tree structure of the regions.
- *Feature methods* create a triangulation based on a subset of the vertices that represents the important features of the terrain.
- *Refinement methods* are based on a coarse approximation, which is refined and re-triangulated during multiple passes.
- *Decimation methods* are based on a triangulation of all vertices, which is simplified and re-triangulated during multiple passes.
- *Optimal methods* are included only for their theoretical properties.

The first two categories generate regular triangulations, while the latter four usually produce triangulated irregular networks. Delaunay and data-dependent triangulation are two common examples of triangulation methods that produce triangulated irregular networks. Delaunay triangulation triangulates a set of two-dimensional points by maximising the minimum angle of all triangles, while data-dependent triangulation methods use the heights of the points to achieve more accurate triangulations, but they introduce more *slivers*, i.e. thin triangles. Both Delaunay and data-dependent triangulation are discussed in more detail in Appendix A.

2.1 Regular Grid Methods

Regular grid subsampling is the simplest surface-simplification algorithm. It samples only the points in every k^{th} row and column. No other point is considered in the approximation. The produced subset is then triangulated to a regular mesh. Heckbert and Garland points out that these methods are simple and fast, but since no consideration is taken to the structure of the terrain or the position of the observer, they produce terrain of low quality.

Regular grid subsampling can be extended to a multiresolution model by hierarchically producing a pyramid. These methods are, according to Heckbert and Garland, the most widely used type of multiresolution terrain model in both the simulation and visualisation community. More sophisticated methods are necessary to produce triangle meshes that meet the high quality standards of today.

2.2 Hierarchical Subdivision Methods

Hierarchical subdivision methods divide the terrain recursively into a tree. They are adaptive, which means that they stop the subdivision according to a set of parameters, such as view-frustum culling and geometric error. In each step of the subdivision process, the region associated with a node is checked against the view frustum. If the whole region is outside, the recursion stops at that branch. Otherwise, it measures the distance from the observer and determines whether the level of detail is high enough. If more details are necessary, the process continues recursively.

As the regular grid methods, hierarchical subdivision methods are simple and fast, but they also facilitate multiresolution-modelling [16].

Several hierarchical subdivision methods exist. Miller [26] bases his algorithm on a quadtree, in which each level is computed by an approximate least-squares fit to the level below. Both a view-independent component and a more accurate view-dependent screen-space error metric are minimised. A specified triangle count is guaranteed, but frame-to-frame coherence is not exploited and T-vertices are allowed.

Lindstrom et al. [24] build pairs of triangles into a hierarchical structure. Each triangle is associated with another triangle to form pairs of triangles, which are simplified to single triangles by merging the individual triangles. This is defined recursively, but sets a major constraint on the height data. The grid must consist of x^2 vertices, where $x = 2^n + 1$ for some nonnegative integer n . A geometric screen-space error metric determines the order of which the triangles are to be merged, but there is no guarantee of error bounds. There is no triangle-count parameter that maintains strict frame rates, nor is vertex morphing included [6, 24].

Duchaineau et al. [6] continue the research of Lindstrom et al. by a method called *real-time optimally adapting meshes*. They use the same space of continuous triangle-bintree meshes, but incorporate two priority queues to drive the triangle split and merge process. They do not split the height data into regular blocks to reduce the problem of cracks between block boundaries as Lindstrom et al. do. Instead, a sequence of forced splits creates a triangle mesh of continuous level of detail. The same geometric screen-space error metric is used, but a guarantee of error bounds is included. In addition, strict frame rates are maintained by setting a maximal number of generated triangles. Vertex morphing is included in the split and merge process. This method is further discussed in Section 4.

2.3 Feature Methods

Some height samples describe the terrain better than others do. Feature methods exploit this fact by determining the n most important vertices, which are known as *features* or *critical points*. A triangle mesh is determined from these vertices, usually by some Delaunay or data-dependent triangulation method. Some methods find important edges known as *break lines* and incorporate them into the triangulation [16]. A common algorithm is constrained Delaunay triangulation. For details, see Appendix A.

Heckbert and Garland conclude that these methods are inferior in comparison to the other methods.

2.4 Refinement Methods

Refinement methods assume that the height samples have already been approximated. During multiple passes, the approximation is refined by inserting one or more vertices. The triangle mesh is re-triangulated in each such pass. Simple rectangular grids usually are initially approximated by two large triangles, while more complex height data need more sophisticated approximation algorithms. The sequence of vertices to be inserted is usually determined by a measure that computes the distance between a vertex and the current triangulation, e.g. the vertical distance. A view-dependent extension computes the projection of the same distance onto the view plane [16].

2.5 Decimation Methods

A decimation method is the opposite of a refinement method. The decimation approach assumes that an initial triangulation of all input data already exists, and iteratively simplifies the triangulation by removing vertices, edges, or triangles [16].

Xia and Varshney [30] have presented a method for performing view-dependent simplification of general triangulated irregular networks. A merge tree is constructed using edge-collapse operations that do not affect neighbour triangles. The simplifications are dependent on viewing direction, lighting, visibility, and include backface-detail reduction based on Gauss-map normal bounds. The algorithm consists of both a preprocessing component and a runtime component, which exploit both screen-space error metrics and frame-to-frame coherence.

Hoppe [17, 18, 19, 20, 21] has based his research on Hoppe et al. [22], who create a decimated triangulated irregular network from three operations, edge collapse, vertex split, and edge swap. Hoppe discovered that the edge collapse was itself enough for simplification, which led to a simplification representation called progressive meshes. Progressive meshes store a fine mesh together with a sequence of edge collapses. As Xia and Varshney, Hoppe reduces backface detail based on nested Gauss-map normal bounds. Some consideration is given to frame-to-frame coherence, but execution times are still proportional to the full output mesh size. Vertex morphing, called geomorphing by Hoppe, is included to reduce popping artefacts. Progressive meshes are further discussed in Section 3.

2.6 Optimal Methods

In addition to the methods described above, it exists a few methods that do not fit into any of the previous categories. Those are optimal methods, which are included only for their theoretical properties rather than practical applicability. The problem of finding an L_∞ -optimal polygonal approximation of a height field is NP-hard, which implicates that any algorithm that finds such an approximation has an exponential time complexity [16].

3 PROGRESSIVE MESHES

Progressive meshes were developed by Hoppe [19] based on mesh optimisation [22], in which the space of triangle meshes is traversed and simplified by applying three operations, edge collapse, edge split, and edge swap. Hoppe discovered that only one of these operations, the edge collapse, is sufficient for simplifying meshes.

Hoppe later extended the algorithm to include view-dependent simplification of triangle meshes, which enabled progressive meshes to be included in real-time applications [20, 21]. A progressive mesh is defined as a sequence of triangle meshes of various levels of detail that are computed based on terrain structure and observer position. The underlying set of vertices is not required to be a regular grid, but this will be assumed when progressive meshes are compared to real-time optimally adapting meshes in Section 4.7.

3.1 Definitions

A *triangle mesh* M is denoted by a tuple (V, F) , where $V \subseteq \mathbf{R}^3$ is a finite set of vertices, and $F \subset V^3$ is a set of ordered triples (v_i, v_j, v_k) that specifies the vertices of the triangles ordered counter-clockwise. Hoppe defines a triangle mesh M as a tuple (K, V) , where K is a *simplicial complex* that represents the connectivity of the vertices, edges, and faces, and $V = \{v_1, v_2, \dots, v_n\} \subseteq \mathbf{R}^3$ is a finite set of vertices. A simplicial complex K consists of a set of vertices $\{v_1, v_2, \dots, v_n\}$ and a set of non-empty subsets of the vertices, which is called the simplices of K . The 0-simplices $\{i\} \in K$ are called vertices, the 1-simplices $\{i, j\} \in K$ are called edges, and the 2-simplices $\{i, j, k\} \in K$ are called triangle faces. For each simplex $s \in K$, $|s|$ denotes the convex hull of its vertices in \mathbf{R}^3 and $|K| = \bigcup_{s \in K} |s|$. $\phi: \mathbf{R}^n \rightarrow \mathbf{R}^3$ is defined as the linear function that maps the i^{th} standard basis vector $e_i \in \mathbf{R}^n$ to $v_i \in \mathbf{R}^3$. The image $\phi(|K|)$ is known as the *geometric realisation* of M .

The *neighbourhood* of a vertex v is defined as the set of triangle faces adjacent to v . It is often called the *star* of the vertex. The *neighbourhood* of an edge $e = \{v_i, v_j\}$ refers to the union of the neighbourhoods of the vertices v_i and v_j . It is often called the *star* of the edge. An edge that has only one adjacent triangle is called a *boundary edge*. A *boundary vertex* is a vertex adjacent to at least one boundary edge. An *interior vertex* is a non-boundary vertex. Finally, the *valence* of a vertex is defined as the number of edges adjacent to that vertex.

Let M^0 be a coarse triangle mesh simplified from a triangle mesh M . A *progressive mesh*, abbreviated PM, of M is a sequence of n vertex splits that refines M^0 into M . A *vertex split* is an elementary mesh operation that adds a vertex and two triangles to the mesh. The number of vertices in M^0 is denoted m_0 . The vertices of mesh M^i is denoted $V^i = \{v_1, v_2, \dots, v_{m_0+i}\}$. The position of a vertex v_j in M^i is labelled v_j^i [19, 21, 23].

3.2 Progressive Mesh Representation

A PM can be illustrated as

$$M^0 \xrightarrow{vsplit_0} M^1 \xrightarrow{vsplit_1} \dots \xrightarrow{vsplit_{n-1}} (M^n = M). \quad (3.1)$$

An operation $vsplit_{i-1}$ defines the vertex split that generates the i^{th} mesh in the PM. A PM is represented by a tuple $(M^0, \{vsplit_0, vsplit_1, \dots, vsplit_{n-1}\})$, of which each vertex-split operation can be parameterised as $vsplit(v_u, v_l, v_r, v_s, v_t, f_l, f_r)$, indicating that a vertex v_u is replaced by two new vertices v_s and v_t . Two triangle faces $f_l = (v_l, v_s, v_t)$ and $f_r = (v_s, v_r, v_t)$ are also inserted into the triangle mesh in the operation. This is illustrated in Figure 3.1. If v_u would be a boundary vertex, then either v_l or v_r are set to invalid and only one face will be introduced into the mesh.

Hoppe [21] parameterises the vertex-split operation as $vsplit(v_s, v_l, v_r, v_t, f_l, f_r)$, which creates one new vertex v_t , and repositions v_s . However, this parameterisation is inadequate, since the new position of v_s is not included.

A PM of M can also be represented as a sequence of n records that applies the inverse operation to vertex split, an *edge collapse*, as illustrated in Figure 3.1. The split of vertex v_u introduces new vertices v_s and v_t , and two new triangle faces $f_l = (v_l, v_s, v_t)$ and $f_r = (v_s, v_r, v_t)$ into the mesh. The edge collapse is the inverse operation, i.e. the vertices v_s and v_t are replaced by a single vertex v_u . The triangle faces f_l and f_r are removed in the process.

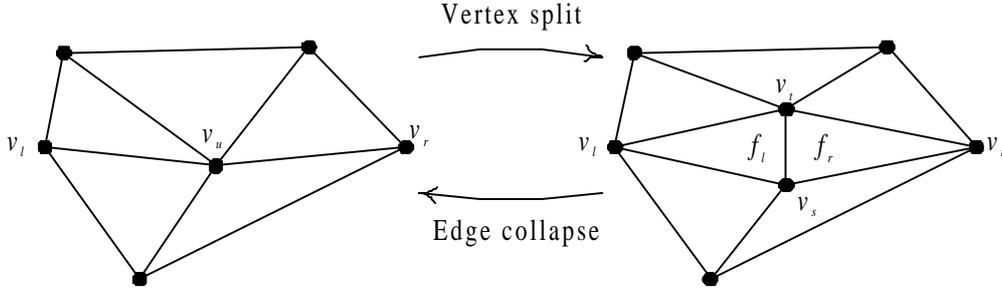


Figure 3.1: The vertex-split and edge-collapse operations.

This sequence becomes

$$(M = M^n) \xrightarrow{ecol_{n-1}} M^{n-1} \xrightarrow{ecol_{n-2}} \dots \xrightarrow{ecol_0} M^0. \quad (3.2)$$

Each edge collapse operation is parameterised as $ecol(v_u, v_l, v_r, v_s, v_t, f_l, f_r)$, which indicates that the vertices v_s and v_t , adjacent to v_l and v_r are collapsed to a vertex v_u that is also adjacent to v_l and v_r . This removes the triangle faces $f_l = (v_l, v_s, v_t)$ and $f_r = (v_s, v_r, v_t)$ from the mesh.

Thus, a PM defines a sequence of meshes M^0, M^1, \dots, M^n from which n view-independent level-of-detail approximations can be retrieved. The order of edge collapse operations determines the quality of the approximating meshes $M^i, i < n$.

3.3 Progressive Mesh Construction

Several different methods determine the order of edge collapses. They have all to compromise between simplicity, speed, and accuracy. To maximise the simplicity and speed, the order of edge collapses could be chosen at random, but this guarantees no accuracy. Since the PM construction is an offline process, the speed is of low priority. Hoppe [19], Xia and Varshney [30], and Garland and Heckbert [13] have presented algorithms that produce progressive meshes of high quality.

3.3.1 Energy Function

To find the right order of edge collapses, Hoppe minimises an energy function $E: \mathbf{M} \rightarrow \mathbf{R}$, defined by

$$E(M) = E_{dist}(M) + E_{rep}(M) + E_{spring}(M), \quad (3.3)$$

where $E_{dist}, E_{rep}, E_{spring}: \mathbf{M} \rightarrow \mathbf{R}$ and \mathbf{M} is the set of all meshes. The *distance energy* term E_{dist} equals the sum of squared distances from the vertices $\{v_1, v_2, \dots, v_n\}$:

$$E_{dist}(M) = \sum_i d^2(v_i, \phi_v(|\mathbf{K}|)). \quad (3.4)$$

The *representation energy* E_{rep} term is defined by

$$E_{rep}(M) = c_{rep} m, \quad (3.5)$$

which is a penalty function that adds a term proportional to the number of vertices m in M . Finally, the *spring energy* term E_{spring} is defined by

$$E_{spring}(M) = \sum_{\{j,k\} \in K} \kappa \|v_j - v_k\|^2. \quad (3.6)$$

The spring energy term can be compared to placing a spring of rest length zero and tension κ on each edge. These three terms together define a function that measures the closeness to fit, penalises meshes with a large number of vertices, and regularises the mesh to guide the optimisation to a desirable local minimum.

The function E has to be minimised in order to find the right order of edge collapses. This is done by minimising E by two nested loops. The outer loop optimises over \mathbf{K} , the connectivity of the mesh, while the inner loop minimises the set $\{E_{dist}(V) + E_{spring}(V) \mid V \in \mathbf{V}\}$.

3.3.2 Merge Tree

Xia and Varshney [30] base their PM sequence in order of edge lengths in a merge tree. At the first level of the tree, as many edges as possible are collapsed as long as the neighbourhoods of the edges do not intersect. At the second level, the remaining edges are collapsed according to the same rule. Within each level, the edge collapses are ordered by increasing edge lengths. The process is repeated until no more edges can be collapsed.

3.3.3 Quadric Error Metric

Garland and Heckbert [13] assign a cost to each potential edge collapse using quadric error metrics. A priority queue that is keyed on cost is built with the minimum cost edge at the top. If an edge is collapsed, a new vertex is introduced. The cost for all edges adjacent to this vertex is recomputed and the priority queue is updated. They associate a symmetric 4×4 matrix Q with each vertex v , which is represented by a vector $[v_x \ v_y \ v_z \ 1]^T$. The error at vertex v is computed by $\varepsilon = v^T Q v$ and represents the squared distances from the vertex to a set of planes in its neighbourhood.

3.4 Selective Refinement and Coarsening

A *selectively refined mesh* M^s is defined as the triangle mesh obtained by applying a subsequence $S \subseteq \{0, 1, \dots, n-1\}$ of the PM vertex-split sequence. This may however introduce inconsistent meshes. In Figure 3.2, two edge collapse operations are performed; one that collapses edge $\{v_i, v_j\}$ and one that collapses edge $\{v_s, v_t\}$, with parameterisations $ecol(v_k, v_l, v_m, v_i, v_j, f_l, f_m)$ and $ecol(v_u, v_j, v_r, v_s, v_t, f_j, f_r)$, respectively.

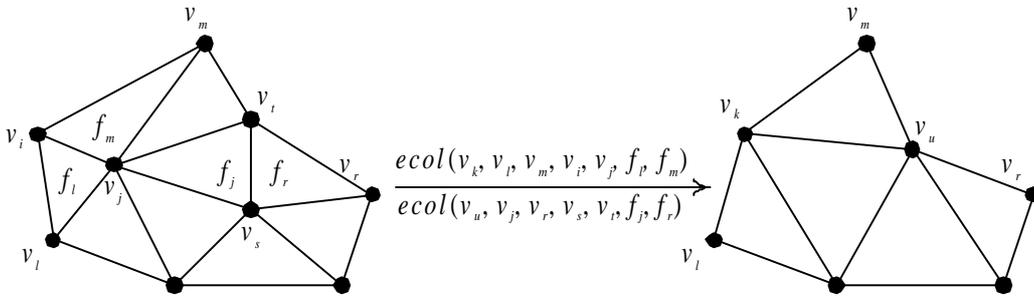


Figure 3.2: Collapsing edges $\{v_i, v_j\}$ and $\{v_s, v_t\}$.

If edge $\{v_i, v_j\}$ is collapsed before edge $\{v_s, v_t\}$, vertex v_j will be replaced by another vertex v_k before edge $\{v_s, v_t\}$ is collapsed, which results in an inconsistent parameterisation, since v_j is included in the parameterisation of the second collapse. However, if the order of collapses is reversed, both edge collapses will become consistent. Thus, the order of edge collapses is important to maintain consistent meshes.

Hoppe [21] introduced new definitions of vertex split and edge collapse that together with a set of legality preconditions are sufficient for consistency. The operations are still the same; only their parameterisations differ. Vertex split is parameterised as $vsplit(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$, and edge collapse is parameterised similarly, $ecol(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$. As shown in Figure 3.3, the vertex split replaces a vertex v_u by two other vertices v_s and v_t . Two new triangle faces are introduced in the operation, $f_l = (v_l, v_s, v_t)$ and $f_r = (v_s, v_r, v_t)$, between two pairs of neighbouring faces (f_{n0}, f_{n1}) and (f_{n2}, f_{n3}). The edge collapse applies the inverse operation, i.e. two vertices v_s and v_t are merged into a single vertex v_u . The two faces f_l and f_r vanish in the process. Meshes with boundary are supported by letting the face neighbours f_{n0} , f_{n1} , f_{n2} , and f_{n3} representing special *nil* values. A vertex split with $f_{n2} = f_{n3} = nil$ creates only a single face f_l .

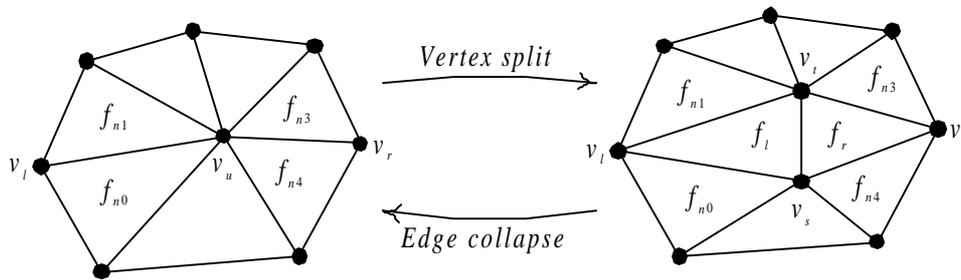


Figure 3.3: The new vertex-split and edge-collapse parameterisations.

3.4.1 Legality of Edge Collapse

There are two necessary preconditions for the new definitions of *vsplit* and *ecol* to be *legal*. A $vsplit(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$ operation is *legal* if v_u is active and all faces f_{n0} , f_{n1} , f_{n2} , and f_{n3} are active, i.e. $v_u, f_{n0}, f_{n1}, f_{n2}, f_{n3}$ are all present in the triangle mesh. An $ecol(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$ operation is *legal* if v_s and v_t are active vertices and the faces adjacent to f_l and f_r are f_{n0}, f_{n1}, f_{n2} , and f_{n3} .

An illegal edge collapse can change the topology of the mesh. If there is a hole in the mesh, as in Figure 3.4 (a), a collapse of the edge $\{v_s, v_t\}$ removes the hole. Collapsing the edge $\{v_s, v_t\}$ in Figure 3.4 (b) creates a mesh that is not 2-manifold and the collapse of the edge $\{v_s, v_t\}$ in Figure 3.4 (c) can invert faces if the resulting vertex is positioned to the left of v_a .

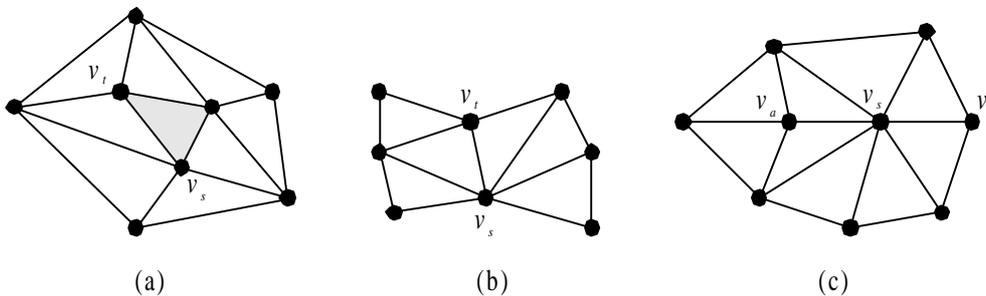


Figure 3.4: a) Collapsing $\{v_s, v_t\}$ removes the hole. b) Collapsing $\{v_s, v_t\}$ results in a mesh that is not 2-manifold. c) Collapsing $\{v_s, v_t\}$ to a vertex that is placed to the left of v_a inverts the faces.

This has resulted in another definition of edge legality. An edge collapse $ecol(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$ is *legal* if the following three conditions are fulfilled [23]:

1. If the resulting vertex v_u is an interior vertex, its valence must be greater than or equal to three. Otherwise, its valence must be greater than or equal to one.
2. If a vertex v_l is adjacent to both v_s and v_t , either the triangle $(v_l, v_t, v_s) \in \mathbf{F}$ or $(v_l, v_s, v_t) \in \mathbf{F}$.
3. If v_s and v_t are both boundary vertices, the edge $\{v_s, v_t\}$ must be boundary.

The problem of mesh inversion can be prevented by three methods. First, based on a method by Lilleskog [23], the dot product of the normals of the triangle before and after the edge collapse can be computed. If the result is negative, the triangle has been flipped, and the edge collapse should be disallowed. Second, this problem can be avoided by constructing a projection plane, on which the neighbourhood of the edge is injectively mapped. The resulting vertex is placed so that inversion is impossible. The third method is the computationally cheapest one, although it restricts the position of the resulting vertex v_u . Place v_u only on a location previously occupied by the collapsed edge.

The introduction of slivers in the terrain can produce aliasing artefacts with texture mapping. Lilleskog [23] and Guézic [14] have developed two methods to avoid them:

1. A vertex of high valence is likely to be adjacent to slivers. These slivers can be avoided by setting an upper limit of vertex valences.
2. Guézic uses a metric called the *compactness* c of a triangle with lengths l_0 , l_1 and l_2 of the sides and area a :

$$c = \frac{4\sqrt{3}a}{l_0^2 + l_1^2 + l_2^2} \quad (3.7)$$

The area can be computed by Heron's formula:

$$a = \sqrt{P(P-l_0)(P-l_1)(P-l_2)}, \text{ where } P = \frac{l_0 + l_1 + l_2}{2} \quad (3.8)$$

Both the valence of a vertex and the compactness of a triangle can be used to restrict edge collapses that would introduce slivers.

3.5 Vertex Morphing

Hoppe has introduced a method called *geomorphing* for morphing vertices between two meshes. A geomorphed mesh is denoted $M^G(\alpha)$, where $0 \leq \alpha \leq 1$, such that $M^G(0)$ looks like M^i and $M^G(1) = M^{i+1}$ and $M^G(\alpha)$ is the linear interpolation between M^i and M^{i+1} . More formally, a geomorph $M^G(\alpha)$ is defined by

$$M^G(\alpha) = (\mathbf{K}^{i+1}, \mathbf{V}_j^G(\alpha)), \quad (3.9)$$

where \mathbf{K}^{i+1} is the simplicial complex for M^{i+1} and $\mathbf{V}_j^G(\alpha)$ is defined by

$$\mathbf{V}_j^G(\alpha) = \begin{cases} \alpha v_j^{i+1} + (1-\alpha)v_{s_i}^i, & j \in \{s_i, m_0 + i + 1\} \\ v_j^i, & j \notin \{s_i, m_0 + i + 1\} \end{cases} \quad (3.10)$$

Thus, $M^G(\alpha)$ has the same connectivity as M^{i+1} , but the positions of the vertices are interpolated from those in M^i .

These ideas can be extended to construct geomorphs between any two meshes. There is a natural correspondence between any two meshes M^c and M^f , with $0 \leq c < f \leq n$. Every vertex in M^f is related to a

unique vertex of M^c by a surjective map A^c , which is obtained by composing the sequence of edge collapse operations. More formally, a vertex v_j in M^f is related to the vertex $v_{A^c(j)}$ in M^c , where

$$A^c(j) = \begin{cases} j, & j \leq m_0 + c \\ A^c(s_{j-m_0-1}), & j > m_0 + c \end{cases} \quad (3.11)$$

Thus, a geomorph can be defined by

$$M^G(\alpha) = (\mathbf{K}^f, \mathbf{V}_j^G(\alpha)), \quad (3.12)$$

where

$$\mathbf{V}_j^G(\alpha) = \alpha v_j^f + (1-\alpha)v_{A^c(j)}^c. \quad (3.13)$$

Thus, smooth transition between two consecutive frames is possible by progressive meshes regardless of the difference in triangulation.

3.6 View-Dependent Refinement of Progressive Meshes

Progressive meshes were originally optimised for view-independent simplification of terrain only, but the algorithm was improved in [21], where Hoppe developed *view-dependent progressive meshes* (VDPM) that incrementally refines the progressive mesh.

3.6.1 Overview

A progressive-mesh sequence can be represented by a forest of binary trees. The roots of the binary trees are the vertices of the base mesh M^0 , and children are created as vertices are split. The original mesh M is constructed from all leaf nodes.

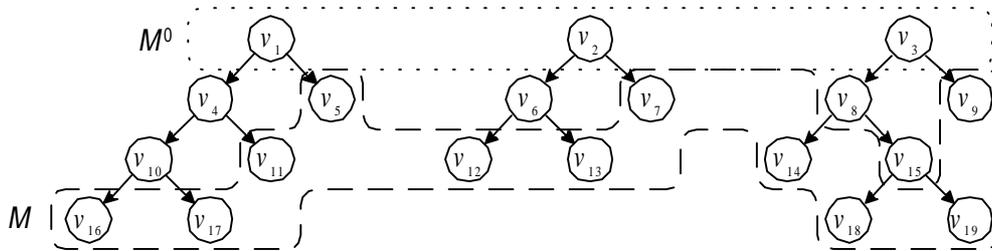


Figure 3.5: A forest of binary trees that represents a VDPM.

A selectively refined mesh M^s is achieved by keeping track of a *vertex front* in the tree. Vertex-split and edge-collapse operations move parts of the vertex front up or down in the hierarchy, as illustrated in Figure 3.6.

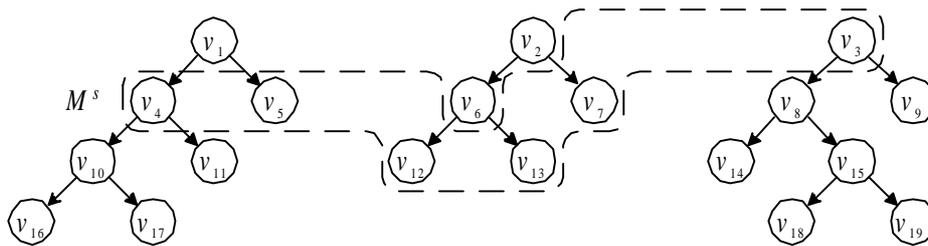


Figure 3.6: A vertex front of a selectively refined mesh M^s .

3.6.2 Selective Refinement Algorithm

There is an algorithm that incrementally adapts a mesh for selective refinement. The vertex front is stored as a list of active vertices together with a list of all active triangles. The vertex list is traversed before each frame is rendered and a decision is made for each vertex whether to leave it as it is, split it, or collapse it.

A query function, *qrefine*, determines whether a vertex v should be split based on the current view parameters. It returns false either if v is outside the view frustum, if it is oriented away from the observer, or if a screen-space geometric error is below a given tolerance. Otherwise, it returns true. Hoppe [21] and Lilleskog [23] discuss these refinement criteria in detail.

Algorithm Adaptive Refinement
Input: • A set of active vertices V in a VDPM.
Output: • A refined triangulation.
<pre> 1. for each $v \in V$ do 2. if v has children and $qrefine(v)$ then 3. force $vsplit(v)$ 4. else 5. if v has a parent and edge collapse is legal for $v.parent$ and not $qrefine(v.parent)$ then 6. $ecol(v.parent)$ 7. end if 8. end if 9. end for </pre>

Algorithm 3.1: Adaptive refinement.

Algorithm 3.1 checks each vertex and splits it if necessary. If *qrefine*(v) evaluates to true, the vertex v should be split. If the preconditions for splitting v are not fulfilled, a sequence of other vertex splits is performed in order for *vsplit*(v) to become legal. This is performed by Algorithm 3.2.

If either v has no children or *qrefine*(v) returns false, v is checked for a parent. If v has a parent and the edge collapse of v and its sibling is legal, the edge collapse is performed if *qrefine* returns false for the parent of v .

Algorithm 3.2 keeps all vertices that have to be split in a stack. The parent of each vertex v in the stack is pushed onto the stack if v is not active. If v is active and *vsplit*(v) is legal, then v is split. This is repeated until the original vertex is split. Hoppe [21] describes both algorithms together with necessary data structures in more detail. Some implementation details are also included.

Algorithm Force vsplit	
Input:	• A vertex v in a VDPM.
Output:	• A refined VDPM, where v is split. Any other vertices that have to be split in order to the split of v is to be legal are also split.
1.	$stack := v$
2.	while $v := stack.pop()$ do
3.	if v has children and $v.f_i \in F$ then
4.	$stack.pop()$
5.	else
6.	if $v \notin V$ then
7.	$stack.push(v.parent)$
8.	else
9.	if $vsplit(v)$ is legal then
10.	$stack.pop()$
11.	$vsplit(v)$
12.	else
13.	for $i := 0$ to 3 do
14.	if $v.f_{ni} \notin F$ then
15.	$stack.push(v.f_{ni}.parent)$
16.	end if
17.	end for
18.	end if
19.	end if
20.	end if
21.	end while

Algorithm 3.2: Force vsplit.

4 REAL-TIME OPTIMALLY ADAPTING MESHES

Real-time optimally adapting meshes, ROAM, were presented by Duchaineau et al. [6] as a further development of previous work by Lindstrom et al. [24]. Duchaineau et al. chose the same space of binary triangle-tree meshes, but used split and merge operations instead of triangle fusions.

4.1 Definitions

This section uses a similar notation as Duchaineau et al. A triangle $T = (v_1, v_2, v_3)$ is defined by the positions of its three vertices v_1, v_2 , and v_3 ordered counter-clockwise. A *triangle bintree* is a binary tree, whose nodes consist of right-isosceles triangles. An edge E with neighbouring vertices v_1 and v_2 is denoted $\{v_1, v_2\}$. A right-isosceles triangle T will be denoted $T = (v_a, v_0, v_1)$, where v_a is the apex vertex, v_0 is the left base vertex, and v_1 is the right base vertex. If $T = (v_a, v_0, v_1)$ is a node in a triangle bintree, the children of T are defined by splitting T along an edge from v_a to the midpoint vertex v_c of v_0 and v_1 . The left child of T is $T_0 = (v_c, v_a, v_0)$ and the right child is $T_1 = (v_c, v_1, v_a)$. Figure 4.1 illustrates a three-level triangle bintree.

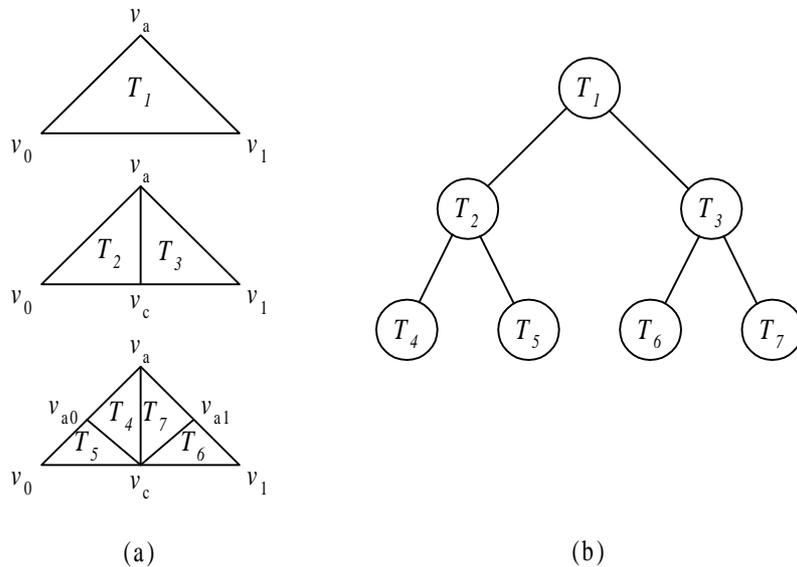


Figure 4.1: The first three levels of a triangle bintree.

4.2 ROAM Representation

The purpose of a triangle bintree is to easily choose the local level of detail of the triangulation. If the terrain is flat or distant, only a few triangles are necessary to approximate a large area, while if the terrain is rough or close, more triangles are required.

The source data is a regular grid of equally spaced height samples. ROAM is limited to only handle data of sizes $(2^n+1) \times (2^n+1)$. However, data structures of other sizes can be divided into several small quadratic blocks. Each block may be represented by two large triangles, who form the root nodes of their respective triangle bintree. The rest of the triangle bintrees are defined by recursively applying the splitting process until a triangle of minimal size is constructed. These triangles form the leaves of the tree.

Each interior triangle T in the triangle bintree has three neighbours. T_B is defined to be the base neighbour that shares its base edge $\{v_0, v_1\}$ with T . Similarly, T_L is defined to be the left neighbour that

shares its right edge $\{v_a, v_0\}$ with T and T_R is defined to be the right neighbour that shares its left edge $\{v_1, v_a\}$ with T . This is illustrated in Figure 4.2.

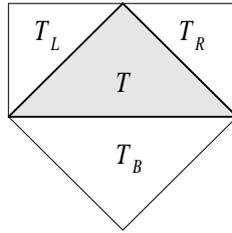


Figure 4.2: The notation for neighbour triangles.

Left and right neighbours to a triangle T in a bintree triangulation are either from the same bintree level l as T or from the next finer level $l+1$. The base neighbour can only be from the same level l or the next coarser level $l-1$.

If both a triangle T and its base neighbour T_B is of the same bintree level, the tuple (T, T_B) is said to be a *diamond*. If T is to be split, T_B has also to be split in order to avoid cracks and shading discontinuities [6, 31]. The split of T_B may also cause other neighbours to be split, resulting in a recursive sequence of splits. The splits in this sequence are known as *forced splits* and are illustrated in Figure 4.3.

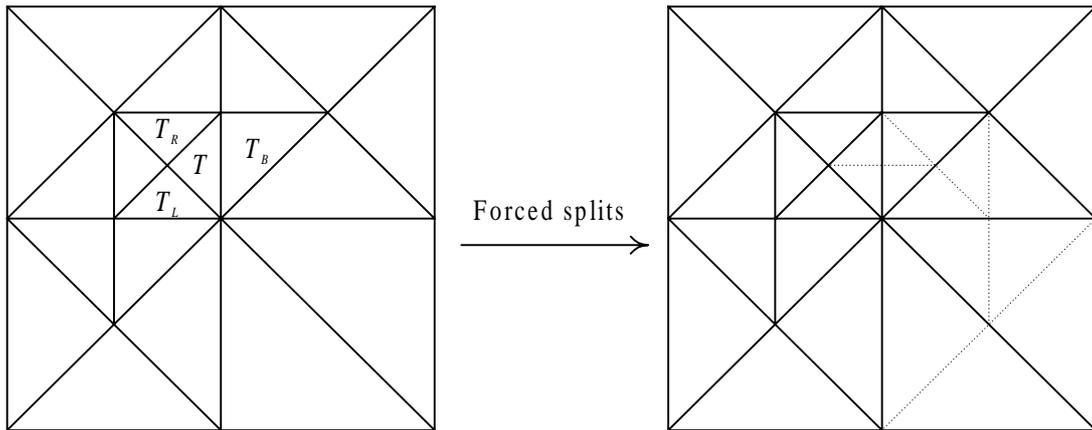


Figure 4.3: The split of T can result in a sequence of forced splits.

If both a triangle T and its base neighbour T_B have been split once, they may be merged. In this case, they are said to form a *mergeable diamond*. However, (T, T_B) does only form a mergeable diamond if none of the children of T or T_B is split.

These two tree operations, *split* and *merge*, are together enough to obtain any other triangulation from a given base triangulation. With the use of force splits, no other efforts are required to avoid cracks or shading discontinuities.

4.3 Vertex Morphing

Vertex morphing provides the possibility of animating the split and merge operations during a set of consecutive frames to avoid popping artefacts. $w(v)$ is defined to be the position (v_x, v_y, v_z) of a vertex v . For a split operation, the unsplit base-edge midpoint

$$w_m = \frac{w(v_0) + w(v_1)}{2} \tag{4.1}$$

is linearly interpolated to the position

$$w_c = w(v_c) \quad (4.2)$$

of the new vertex v_c . Duchaineau et al. linearly interpolate w_m to w_c with intermediate positions

$$w_a(t) = (1-t)w_m + tw_c, \forall t \in [0,1]. \quad (4.3)$$

However, since the frames are visualised at discrete time steps, this may be reformulated. Morphing a vertex through n consecutive frames with start time t_0 becomes

$$w_a(t) = (1-(t-t_0))w_m + (t-t_0)w_c, \forall t \in \{t_0 + \frac{i}{d} \mid i = 0, 1, \dots, n\}, \quad (4.4)$$

where d is the time between two consecutive frames.

4.4 Triangulation Algorithms

Duchaineau et al. have constructed two algorithms that generate optimised triangulations. The first is simple and used in a few rare cases only. It is based on a rough triangulation that is iteratively refined by a sequence of forced splits. A priority queue Q_s , known as a *split queue*, contains monotonic priorities for all triangles to determine the order of these splits. The priority of a triangle T is determined by the error caused by using T instead of the finest triangulation. The error metrics are covered in Section 4.5.

Algorithm Split Queue
Input: <ul style="list-style-type: none"> • A base triangulation T. • An empty priority queue Q_s. • $N_{max} \in \mathbf{N}$, indicating the maximal number of triangles. • $\epsilon_{max} \in \mathbf{R}$, indicating the maximal error.
Output: • An optimal triangulation T .
<ol style="list-style-type: none"> 1. for all triangles $T \in T$ do 2. insert T into Q_s 3. end for 4. while $T < N_{max}$ or $E(T) > \epsilon_{max}$ do 5. identify highest-priority triangle T in Q_s 6. split(T) 7. remove T and other split triangles from Q_s 8. add any new triangles in T to Q_s 9. end while

Algorithm 4.1: Split Queue.

$|T|$ denotes the number of triangles in T and $E(T)$ denotes as the maximal error of all triangles in T .

It can be shown that Algorithm 4.1 produces optimal triangulations at every step. The first two statements in the while loop create the triangulation, while the last two update the split queue.

Since the viewpoint changes between each frame, so should also the triangulation. The split-queue algorithm can be extended to a frame-coherent algorithm that produces an optimal triangulation for a frame f based on the triangulation for the previous frame $f-1$ by applying both split and merge operations.

Assume we are given triangle bintrees T_f for each frame $f \in \mathbf{N}$ and that every triangle in every triangle bintree is given a priority value. The priority of a triangle does not have to equal the priorities of

the same triangle in the other triangle bintrees. The priorities are computed by a view-dependent screen-space geometric error, which is covered in Section 4.5.

A priority split-queue Q_s contains a monotonic priority $p_f(T)$ for each triangle T in the bintree. A second priority queue Q_m contains the priorities for all mergeable diamonds in the current triangulation. The priority for a mergeable diamond (T, T_B) is defined as the maximum priority $m_p = \max\{p_f(T), p_f(T_B)\}$ of the individual triangles. $S_{max}(T)$ denotes the maximum split priority and $M_{min}(T)$ is defined as the minimum merge priority.

Algorithm Merge Queue	
Input:	<ul style="list-style-type: none"> • A base triangulation T. • Empty priority queues Q_s and Q_m. • $N_{max} \in \mathbf{N}$ that indicates the maximal number of triangles. • $\epsilon_{max} \in \mathbf{R}$ indicates the maximal error.
Output:	• Optimal triangulations for all frames.
	<pre> 1. for all frames f do 2. if $f = 0$ then 3. compute priorities for T's triangles and diamonds and insert them into Q_s and Q_m, respectively 4. else 5. let $T = T_{f-1}$ 6. update the priorities for all elements of Q_s and Q_m 7. end if 8. while $T > N_{max}$ or $E(T) > \epsilon_{max}$ or $S_{max}(T) > M_{min}(T)$ do 9. if $T > N_{max}$ or $E(T) < \epsilon_{max}$ then 10. identify the lowest-priority pair (T, T_B) in Q_m 11. merge(T, T_B) 12. remove the merged children from Q_s 13. add the merged parents T and T_B to Q_s 14. remove (T, T_B) from Q_m 15. add all newly mergeable diamonds to Q_m 16. else 17. identify highest-priority T in Q_s 18. split(T) 19. remove T and other split triangles from Q_s 20. add any new triangles in T to Q_s 21. remove any diamonds whose children were split from Q_m 22. add all newly mergeable diamonds to Q_m 23. end if 24. end while 25. set $T_f = T$ 26. end for </pre>

Algorithm 4.2: Merge queue

Algorithm 4.2 produces the same optimal mesh as Algorithm 4.1 by computing the minimal number of split and merge operations necessary to achieve an optimal triangulation. Thus, the algorithm exploits coherence between consecutive frames. It has a time complexity proportional to the minimal number of split and merge operations. However, if all triangles are disjoint in two consecutive frames, the worst-case time complexity is proportionate to the sum of the number of triangles in both triangulations. These cases occur only when there is a large number of triangles between the minimum merge priority and the maximum split priority. Fortunately, these cases are easily discovered and the triangulation is sped up by reinitialising T , Q_s , and Q_m and fall back to algorithm 4.1.

4.5 Queue Priorities

The priority for each triangle in the triangle bintree is determined by an error metric. The error metric used by Duchaineau et al. is a measure of the geometric screen distortion.

Given a triangle T , a *wedgie* is defined as the volume consisting of points $\{(x, y, z) \in \mathbf{R}^3 \mid (x, y) \in P(T) \text{ and } |z - z_T(x, y)| \leq e_T\}$. $P: \mathbf{T} \rightarrow \mathbf{R}^2$ is a function such that $P(T)$ returns the orthogonal projection of T to the xy -plane, $z_T: P(T) \rightarrow \mathbf{R}$ is a function that returns the z -value of a triangle T at position (x, y) , and $e_T \geq 0$ is known as the *wedgie thickness*. A line segment consisting of points $\{(x, y, t) \in \mathbf{R}^3 \mid t - z_T(x, y)| \leq e_T\}$ is called the *thickness segment* for T . A wedgie is illustrated in Figure 4.4 (b).

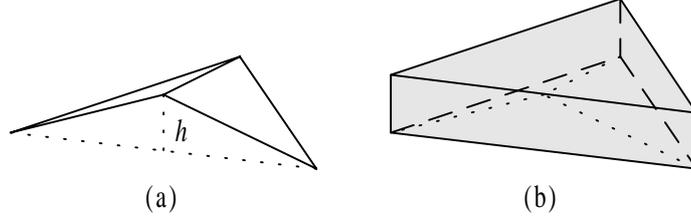


Figure 4.4: a) The height h indicates the object-space error of using a large triangle instead of two smaller ones. b) A wedgie is defined as the grey volume of height h .

The wedgie bounds are built bottom-up, starting with $e_T = 0$ for all the leaves of the triangle bintree. The tightest wedgie bound for a nonleaf triangle T is

$$e_T = \max\{e_{T_0}, e_{T_1}\} + |z(v_c) - z_T(v_c)|, \text{ where } z_T(v_c) = \frac{z(v_0) + z(v_1)}{2}. \quad (4.5)$$

Thus, this formula enables us to assign a priority value for each triangle.

This error metric is view independent, but it can be extended to regard the view position as well. This requires the computation of the geometric screen-space distortion. Let $s(v) \in \mathbf{R}^2$ be the correct screen-space position for a domain point v , and $s_T(v) \in \mathbf{R}^2$ be the approximate position from a triangulation T . Further, let the point-wise geometric distortion at v be defined by $dist(v) = \|s(v) - s_T(v)\|_2$.

The minimal upper bound of the distortion is $\sup\{dist(v) \mid v \in \mathbf{V}\}$, where \mathbf{V} is the set of domain points whose world-space positions $w(v)$ are within the view frustum. This minimal upper bound can occur between two vertices due to the perspective transformation. This is solved by computing an upper bound that is not necessarily minimal.

Let (p, q, r) be the camera-space coordinates of a point $w(v)$. Without loss of generality, assume that the perspective projection is of the form $s = (p/r, q/r)$. The geometric screen-space distortion at v is bounded by projecting the thickness segment at v onto the view plane. Let (a, b, c) be the camera-space vector corresponding to world-space thickness vector $(0, 0, e_T)$. The screen-space distortion at v is bounded by

$$dist_{\max}(v) = \left\| \frac{p+a}{r+c} - \frac{p-a}{r-c}, \frac{q+b}{r+c} - \frac{q-b}{r-c} \right\|_2, \quad (4.6)$$

which can be rewritten as

$$dist_{\max}(v) = \frac{2}{r^2 - c^2} \sqrt{(ar - cp)^2 + (br - cq)^2} \quad (4.7)$$

The minimum of $r^2 - c^2$ and the maximum of $(ar - cp)^2 + (br - cq)^2$ occur at the vertices, although not generally at the same vertex. An upper bound on $dist_{max}(v)$ can thus be obtained by substituting these minimum and maximum values into Equation 4.7.

4.6 Performance Enhancements

Duchaineau et al. have found four optimisation algorithms that increase the frame rate. The first three together decrease the respective computation times for their subtasks by more than a factor of ten, while the fourth ensures strict frame rates.

4.6.1 View-Frustum Culling

The view-frustum culling is based on flags for every triangle. The view frustum can be defined by the intersection of six halfspaces. Four create a pyramid containing the field of view, while the other two are called the near and far clipping planes.

Every triangle is given an *IN* flag for each of the six halfspaces, and three flags named *OUT*, *ALL-IN* or *DONT-KNOW*. *IN* is set when the entire wedgie is inside its corresponding halfspace, *OUT* is set when the entire wedgie is outside at least one halfspace, *ALL-IN* is set when all six *IN* flags are set, and *DONT-KNOW* is set if neither *OUT* or *ALL-IN* is set.

The bintree is traversed recursively to update these flags. Duchaineau et al. describes the purpose of the flags as follows: “If a triangle *T* was labelled *OUT* or *ALL-IN* for the previous frame and these labels are correct for the current frame, the subtree for *T* does not need to be updated and recursion terminates. Otherwise, *T* inherits its *IN* flags from its parent and rechecks its wedgie against the halfspaces not marked *IN*, setting new *IN* flags if appropriate. If the wedgie is entirely outside any of these halfspaces, *T* and all its children are marked *OUT*. If all *IN* flags are set, *T* and all its children are marked *ALL-IN*. Otherwise *T* is marked *DONT-KNOW* and recursion continues to its children.”

4.6.2 Incremental triangle stripping

Rendering performance can be improved by organising triangles into strips. A *triangle strip* is a sequence of three or more vertices, in which every consecutive set of three vertices defines a triangle [11]. Duchaineau et al. have included incremental triangle stripping in their implementation, but the algorithms are not mentioned.

However, they describe the algorithm as a simple, sub-optimal, incremental approach that only considers non-generalised strips. The average strip length has been computed to around four to five triangles. A triangle strip from which a triangle is deleted is either shortened on the end, split in two, or deleted. If a new triangle is to be inserted, it is attached to neighbouring triangle-strip ends, if possible.

4.6.3 Priority-computation Deferral

As the view position changes between two consecutive frames, the split- and merge-queue priorities of all triangles change. Recalculating all priorities is a very time-consuming task. However, it is only necessary to recalculate the priorities when they potentially affect a split or merge operation.

This scheme requires a velocity bound on the observer. A velocity bound determines a time-dependent bound for screen-distortion priorities. The *crossover priority*, i.e. the maximum split-queue priority when the split and merge process has finished, changes very slowly from frame to frame. Test results have shown that these changes are around 1% of the maximum queue priority.

Triangle-priority recomputations are deferred until the priority bound overlaps the crossover priority. A deferral list is kept for future frames, but only the priorities of the triangles on the current frame have to be recomputed.

4.6.4 Progressive Optimisation

The triangulation optimisation has to be stopped when the allotted frame time is about to expire in order to maintain a strict frame rate. Three algorithms supports progressive optimisation: optimisation

processing, triangle stripping and priority recomputations. The only phase of ROAM not suitable for progressive optimisation is view-frustum culling.

4.7 Comparison to VDPM

ROAM has been compared to VDPM in order to find the best algorithm for the implementation. A thorough comparison would require the implementation of both algorithms and compare them empirically. However, time constraints preclude that option. Therefore, the algorithms have only been compared theoretically. Both algorithms include a preprocessing component and a runtime component. The comparison has only considered the speed of the runtime components, since there are no time constraints on the preprocessing components and memory resources are essentially unlimited at Ericsson Saab Avionics AB.

Both ROAM and VDPM support vertex morphing. A VDPM elementary mesh operation, i.e. vertex split or edge collapse, involves two vertices. Therefore, VDPM is required to morph two vertices in such an operation. On the other hand, a ROAM elementary mesh operation, i.e. a split or merge operation, only involves a single vertex. Thus, vertex morphing is less computationally expensive for ROAM.

A ROAM elementary mesh operation is easily checked for validity. A split operation for a triangle t is valid if t is neither a leaf or previously split. A merge operation for a triangle t is valid if it has previously been split. The positions of the resulting vertices from these operations are given from the height field and require no computations.

A validity test for a VDPM elementary mesh operation requires more computations. A vertex split that is parameterised $vsplit(v_u, v_s, v_t, f_l, f_r, f_{n0}, f_{n1}, f_{n2}, f_{n3})$ is legal if $v_u, f_{n0}, f_{n1}, f_{n2}$, and f_{n3} are active. Thus, one vertex and four triangles have to be checked for presence in the mesh. In addition, the position from the resulting vertex from an edge collapse operation is not given and has to be computed. Additional computations must be made to avoid inverting the triangle when an optimal position is found. Thus, while VDPM optimises the positions of its vertices, ROAM is faster.

ROAM automatically avoids slivers since all triangles are right isosceles. The minimum angle in any triangle is $\pi/4$ radians. VDPM have to make additional tests to avoid slivers. If Guézic's method is used, the compactness of the triangle has to be computed. In addition, finding the compactness requires the computation of the lengths of the three triangle edges and the area of the triangle. Morphing two vertices during a VDPM elementary mesh operation introduces slivers temporarily in the mesh. This cannot be avoided and such an operation may result in aliasing artefacts.

Although both ROAM and VDPM exploits frame coherence, ROAM execution time is proportional to the number of triangle changes per frame, while VDPM execution time is proportional to the full output mesh size.

Both ROAM and VDPM generate triangle strips. Duchaineau et al. claim to achieve 4-5 triangles per strip, while Hoppe has computed an average of 4.2 triangles per strip. The execution times for generating the strips are however not comparable.

Duchaineau et al. conclude that “progressive-mesh preprocessing is organised as a global optimisation process, and thus is too slow to support dynamic terrain”, although the space of triangle meshes that can be produced by ROAM is only a subset of the space of progressive meshes. ROAM produce optimal triangulation within the restricted space of triangulations, but it does not produce optimal triangulation in the space of all possible triangulations. Lilleskog [23] has shown that progressive meshes produce triangle meshes with 50–75% of the triangles produced by Lindstrom et al. [24].

In summary, although ROAM requires more triangles for the same maximal error than VDPM, ROAM is so much faster to generate the triangles that it is preferred for real-time applications. Therefore, the implementation of the prototype is based on ROAM.

5 ROAM ALGORITHM AND IMPLEMENTATION ASPECTS

A prototype based on ROAM has been implemented together with a visualisation engine. Since the split and merge algorithms have not been given explicitly by Duchaineau et al., they were designed before the implementation phase. The major difference between the prototype and the original ROAM algorithm is a new merge operation and a main algorithm that does not need to use any split or merge priority queues. Triangle stripping, progressive optimisation, and priority-computation deferral have not been included because of time constraints.

This chapter describes the algorithms, data structures, and implementation issues for the prototype. In total, four algorithms are presented below. The split algorithm, which splits a triangle into two children and performs all necessary force splits, is based on an algorithm by McNally [25]. The merge algorithm can merge the children of any triangle, even if they also have been split. Finally, a main algorithm that controls the split and merge process is presented.

The data structures for the triangle bintree and its nodes are covered at the end together with some implementation issues.

5.1 Split Algorithm

This algorithm is based on an algorithm by McNally [25], but with two minor changes. The purpose is to split a triangle into two children and perform all necessary force splits.

McNally has not implemented the merge operation and consequently does not exploit frame-to-frame coherency either. Therefore, an extra statement that controls neighbour pointers was added in order to adapt his algorithm into one that can be used together with the merge operation. In addition, McNally's algorithm checks if the left neighbour of a triangle's left neighbour is the triangle itself. This can never happen, and the operation should be removed.

The algorithm is divided into two functions, Split and xSplit. Splitting a triangle T is done by the Split function, which performs any necessary force splits first and then calls xSplit twice, once with T and once with the base neighbour of T .

Algorithm Triangle Split
Input: • An unsplit nonleaf triangle T in a triangle bintree.
Output: • An updated triangle bintree, where T is split and all necessary force splits are performed.
<pre> 1. if $T.BaseNeighbour$ is valid then 2. if $T.BaseNeighbour.BaseNeighbour$ is valid then 3. Split($T.BaseNeighbour$) 4. end if 5. xSplit(T) 6. xSplit($T.BaseNeighbour$) 7. $T.LeftChild.RightNeighbour := T.BaseNeighbour.RightChild$ 8. $T.RightChild.LeftNeighbour := T.BaseNeighbour.LeftChild$ 9. $T.BaseNeighbour.LeftChild.RightNeighbour := T.RightChild$ 10. $T.BaseNeighbour.RightChild.LeftNeighbour := T.LeftChild$ 11. else 12. xSplit(T) 13. set $T.LeftChild.RightNeighbour$ to invalid 14. set $T.RightChild.LeftNeighbour$ to invalid 15. end if </pre>

Algorithm 5.1: Triangle Split.

The xSplit function performs the actual splitting process.

Algorithm xSplit
Input: • An unsplit nonleaf triangle T in a triangle bintree.
Output: • An updated triangle bintree, where T is split and all necessary force splits are performed.
<ol style="list-style-type: none"> 1. $T.LeftChild.LeftNeighbour := T.RightChild$ 2. $T.RightChild.RightNeighbour := T.LeftChild$ 3. $T.LeftChild.BaseNeighbour := T.LeftNeighbour$ 4. $T.RightChild.BaseNeighbour := T.RightNeighbour$ 5. if $T.LeftNeighbour$ is valid then 6. if $T.LeftNeighbour.BaseNeighbour = T$ then 7. $T.LeftNeighbour.BaseNeighbour := T.LeftChild$ 8. $T.LeftNeighbour.Parent.RightNeighbour := T.LeftChild$ 9. else 10. $T.LeftNeighbour.RightNeighbour := T.LeftChild$ 11. end if 12. end if 13. if $T.RightNeighbour$ is valid then 14. if $T.RightNeighbour.BaseNeighbour = T$ then 15. $T.RightNeighbour.BaseNeighbour := T.RightChild$ 16. $T.RightNeighbour.Parent.LeftNeighbour := T.RightChild$ 17. else 18. $T.RightNeighbour.LeftNeighbour := T.RightChild$ 19. end if 20. end if 21. set $T.LeftChild.LeftChild$ to invalid 22. set $T.LeftChild.RightChild$ to invalid 23. set $T.RightChild.LeftChild$ to invalid 24. set $T.RightChild.RightChild$ to invalid

Algorithm 5.2: xSplit.

5.2 Merge Algorithm

The force-split operation was introduced by Duchaineau et al. [6] as a part of the split operation. Recall from Section 4.2 that, if two unsplit triangles T and T_B are both from the same level in a triangle bintree, the pair (T, T_B) is said to be a diamond. To split a triangle T , it has to form a diamond with its base neighbour T_B . If (T, T_B) does not form a diamond, i.e. T and T_B are not both from the same level in the triangle bintree, T_B has to be force-split first.

The merge operation is the opposite of the split operation. If T and its base neighbour T_B are split once, then (T, T_B) is referred to a mergeable diamond. A merge operation is only defined on mergeable diamonds. If a mergeable diamond is merged, the children of T and T_B are removed. The mergeable diamond (T, T_B) is transformed to a simple diamond in the process.

This is an unnecessary complex concept of split and merge operations that is restricted to diamonds and mergeable diamonds. It is possible to redefine the merge operation without any need to consider diamonds or mergeable diamonds. Any previously split triangle may be merged so that all its children are removed. In order to maintain continuity and avoid cracks, all neighbouring triangles have to be merged too. This is a simple recursive process, which is called a force-merge operation. Figure 5.1 shows a triangle on the left that is to be merged. On the right, all its children have been removed and the neighbouring triangles have been forced-merged.

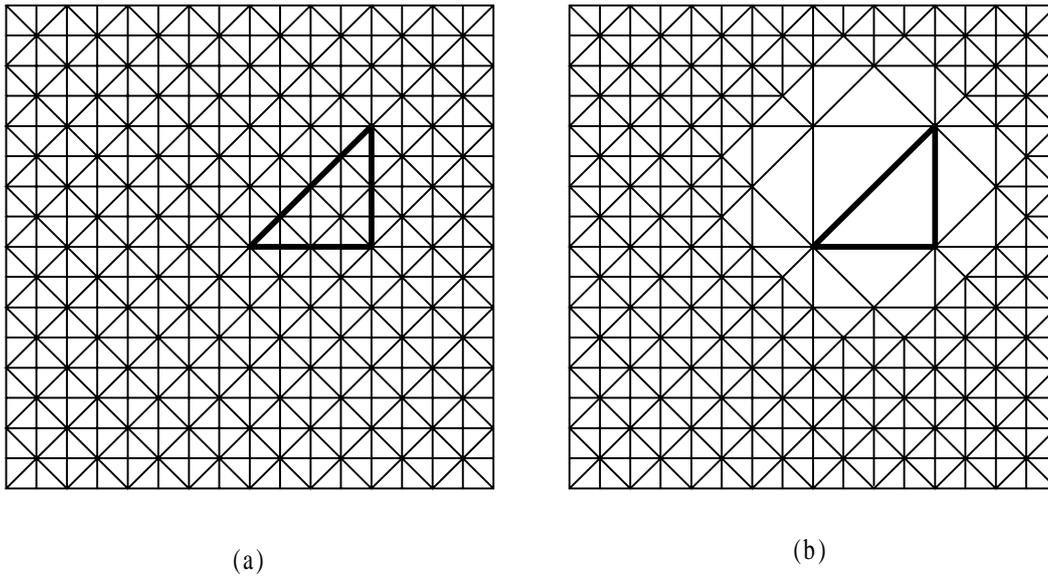


Figure 5.1: The force-merge process that merges the children of an arbitrary triangle.

There are several benefits of the new force-merge operation. First, it is very easy to understand and implement. There is no need to introduce concepts like diamonds or mergeable diamonds. Second, it is more powerful than the old merge operation. Previously, only triangles that were part of a mergeable diamond could be merged. These triangles always had to have two leaf children and a base neighbour with two leaf children. No other triangles could be merged. The new merge operation can merge any nonleaf triangle in the bintree. Third, any implementation of the new merge operation will run faster, since there is no bookkeeping on diamonds and mergeable diamonds. In addition, what previously may have taken several merge operations can now be done in one single operation.

The criticism behind frame-coherent algorithms is that they are very slow if two consecutive frames differ too much. Duchaineau et al. solved this problem by relying on a frame-incoherent algorithm (Algorithm 4.1) in these cases. With the new force-merge algorithm, there is no need to switch to a frame-incoherent algorithm. It will be just as fast as frame-incoherent algorithms in these special cases.

However, the original priority-computation deferral algorithm is not compatible with this merge operation and no alternative has been found. Merging a single triangle involves nothing more than removing its children. The force-merge process then merges the children of the three neighbours of the triangle in a similar way.

Algorithm Triangle Merge
Input: • A split nonleaf triangle T in a triangle bintree.
Output: • An updated triangle bintree, where the children of T are merged and all necessary force merges are performed.
<ol style="list-style-type: none"> 1. set $T.LeftChild$ to invalid 2. set $T.RightChild$ to invalid 3. if $T.LeftNeighbour$ is valid then 4. if $T.LeftNeighbour.RightNeighbour = T.LeftChild$ then 5. $T.LeftNeighbour.RightNeighbour := T$ 6. else 7. Merge($T.LeftNeighbour.Parent$) 8. $T.LeftNeighbour.BaseNeighbour := T$ 9. $T.LeftNeighbour.Parent.RightNeighbour := T$ 10. end if 11. end if

```

12. if  $T.RightNeighbour$  is valid then
13.   if  $T.RightNeighbour.LeftNeighbour = T.RightChild$  then
14.      $T.RightNeighbour.LeftNeighbour := T$ 
15.   else
16.     Merge( $T.RightNeighbour.Parent$ )
17.      $T.RightNeighbour.BaseNeighbour := T$ 
18.      $T.RightNeighbour.Parent.LeftNeighbour := T$ 
19.   end if
20. end if
21. if  $T.BaseNeighbour$  is valid then
22.   if  $T.BaseNeighbour$  is split then
23.     Merge( $T.BaseNeighbour$ )
24.   end if
25. end if

```

Algorithm 5.3: Force merge.

5.3 Main Algorithm

The split and merge algorithms are used by a main algorithm that is called once a frame for each active triangle bintree. During the previous frame, a triangle bintree and the corresponding triangulation were determined. If the view-dependent variances have not changed too rapidly, the triangle bintree of this frame will almost equal the previous frame. Therefore, the main algorithm assumes there is an existing triangle bintree for all frames but the first that only needs to be updated due to the view differences from the last frame.

The main algorithm traverses the tree and examines each triangle to decide whether it has been split or not. If it has not been split, then if the view-dependent variance exceeds a given limit, it is split and the algorithm continues to its children. If it has been split, then if the view-dependent variance is below a given limit, it is merged and the traversing stops at that node.

Algorithm Main	
Input:	• The root triangle T in a triangle bintree.
Output:	• An updated triangle bintree, where all triangles are split, merged, or left as they are.
1.	if not (T is within the view frustum and T was within the view frustum in the previous frame or
2.	T is outside the view frustum and T was outside the view frustum in the previous frame) then
3.	if T is split then
4.	if T should be merged then
5.	Merge(T)
6.	else
7.	Main($T.LeftChild$)
8.	Main($T.RightChild$)
9.	end if
10.	else
11.	if T should be split then
12.	Split(T)
13.	Main($T.LeftChild$)
14.	Main($T.RightChild$)
15.	end if
16.	end if
17.	end if

Algorithm 5.4: Main.

5.4 Implementation Aspects

5.4.1 Bintree Data Structure

The following fields are required for the bintree node data structure:

- Pointer to left neighbour
- Pointer to right neighbour
- Pointer to base neighbour
- Pointer to left child
- Pointer to right child
- Indices to left vertex position
- Indices to right vertex position
- Indices to apex vertex position
- Variance
- View-frustum data
- Priority-computation deferral data

The purpose of the left, right, and base neighbour pointers is to support the force-split and force-merge process. The pointers to the left and right children are necessary for traversing the triangle bintree. The indices to the positions for the left, right, and apex vertices in the height field are required when the triangles are sent for display. The variance field contains a view-independent precomputed variance, on which the screen-space geometric error computation is based. The variance field should only be present in the node if the bintree is implemented as an implicit binary tree since it is very inefficient to recompute the variance during run-time. Otherwise, the variance field should be stored in another tree. The view-frustum data contains the *IN*, *OUT*, *ALL-IN*, and *DONT-KNOW* flags. The priority-computation deferral data contains a nonnegative integer that indicates the number of frames that a priority computation can be deferred.

5.4.2 Implicit Binary Tree

By using an implicit binary tree, the left and right child pointers in the bintree nodes are unnecessary and the pointers to the left, right, and base neighbours are replaced by array indices. An implicit tree is also faster traversed than a dynamic tree since no pointers have to be dereferenced. In addition, no memory has to be allocated or deallocated during dynamic expansion or compression, which speeds up the split and merge process. The drawback is that implicit binary trees require a large amount of preallocated memory but this is no problem for Ericsson Saab Avionics AB. Therefore, the prototype has been developed using implicit binary trees rather than dynamic binary trees.

Another solution is to compromise between the speed of implicit binary trees and the low memory consumption of dynamic binary trees. A tree can be preallocated for all but the last few levels to an implicit binary tree. The last levels of the tree are allocated dynamically as needed. Extra speed is gained if the memory allocation is handled internally by keeping another preallocated array. The memory in this array can then be dynamically used to tree nodes.

5.4.3 Split and Merge Operations

Although both the split and merge algorithms are given in recursive forms, they have been implemented iteratively to avoid function call overheads.

5.4.4 Split and Merge Queues

The split and merge priority queues have not been implemented due to time constraints. However, the improved merge operation reduces the need for such queues. The major benefit of priority queues is progressive optimisation, which is rarely necessary to use.

5.4.5 View-Frustum Culling

The view-frustum culling has been implemented almost as described by Duchaineau et al. The only difference is that no node inherits the parameters of the parent. This has been done because of time constraints but it only reduces the frame rate slightly.

As can be seen in Chapter 6, the overhead of computing the six clipping planes for each frame is compensated by the reduction of triangles.

5.4.6 Triangle Count

A desired number of triangles per frame can be achieved by dynamically changing a priority cut-off value. Normally a triangle is split during frame f only if its view-dependent variance exceeds a given floating-point number ϵ_f . The number of created triangles c is therefore dependent of ϵ_f . If the desired number of triangles d differs from c , a new value of ϵ can be recomputed by the equation

$$\epsilon_f = \frac{c}{d} \epsilon_{f-1} \quad (5.1)$$

If the view position remains unchanged, Equation 5.1 will be computed iteratively and converge to a value that produces the desired triangle count.

6 RESULTS

The prototype generates ROAM terrain at high speeds. In Figure 6.1, the triangles close to the observer are significantly smaller than those triangles that are farther away. The sizes of the triangles' projection to the screen are still the same. The terrain has been generated from a height map of 129×129 height samples. 1200 triangles are being rendered each frame, which have reduced almost all visible popping artefacts. The total number triangles present in the bintree is however much larger. The view-frustum culling removes most of the triangles.

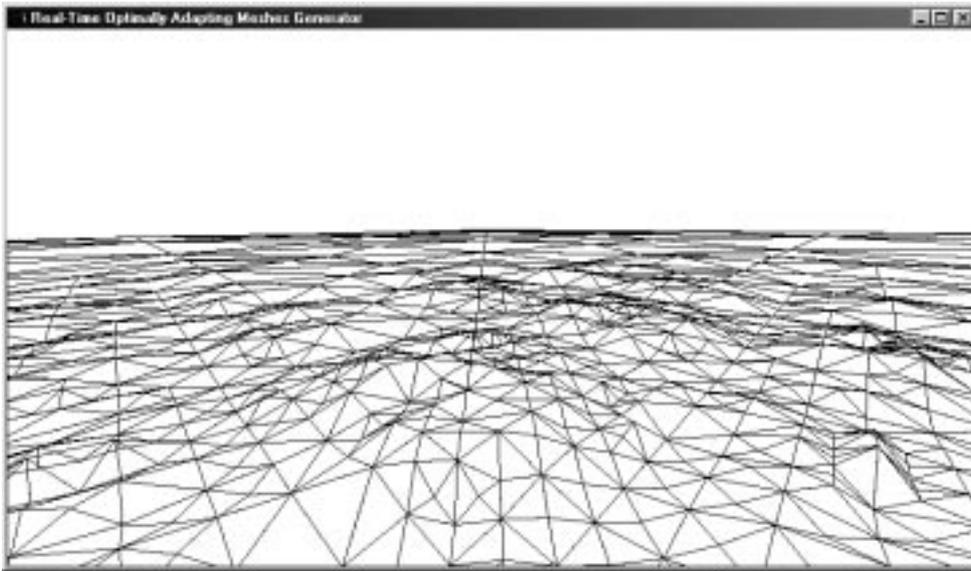


Figure 6.1: ROAM generated terrain.

The characteristics of ROAM are apparent in both Figure 6.1 and Figure 6.2. All triangles are right isosceles and the pattern by which the triangles are split and merged is visible. There are no cracks present and the level of detail is continuous.

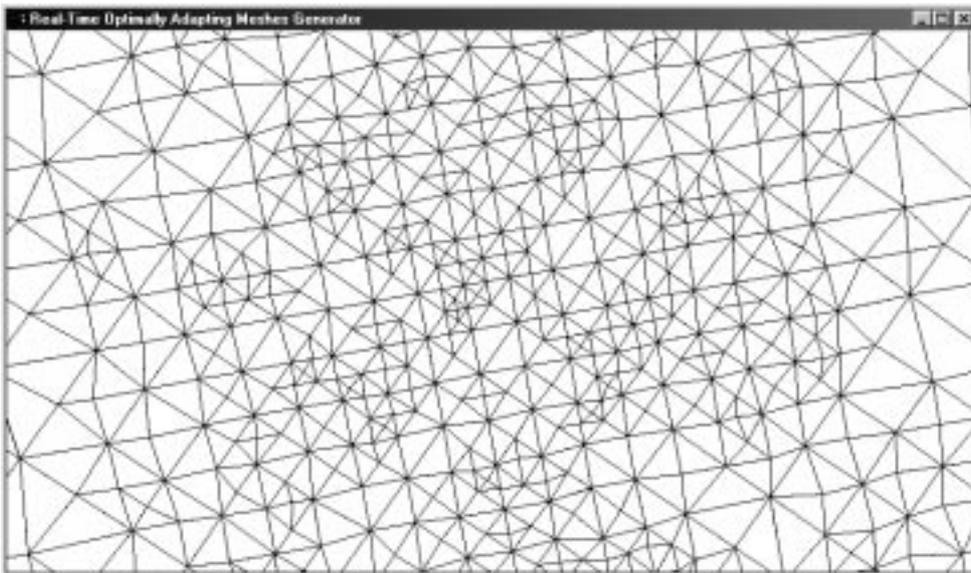


Figure 6.2: Top view of ROAM generated terrain.

6.1 Performance

Performance measurements of the ROAM prototype are made on a Dell Dimension XPS T600 with an Intel 600 MHz Pentium III processor, 256 MB of memory, and an nVidia TNT2 M64 graphics card under Microsoft Windows 98.

The source code has not been optimised. Triangle stripping is not implemented. Instead, vertex arrays are created during every frame, which decreases the frame rate. The view-frustum culling is not identical to the one by Duchaineau et al. The differences are explained in Section 5.4.5.

6.1.1 Frame Rates

Table 6.1 compares the frame rates at different speeds of the observer.

Speed	Frame rate (fps)	Triangles
Standing still	105	1230
Medium speed	97	1190
High speed	91	1160

Table 6.1: Frame rate as function of speed. A height map of 129*129 height samples has been used.

As can be seen in Table 6.1, the frame rate drops at higher speed. This is a result from the exploitation of frame-to-frame coherence. The more triangle changes there are per frame, the more computations are required by ROAM.

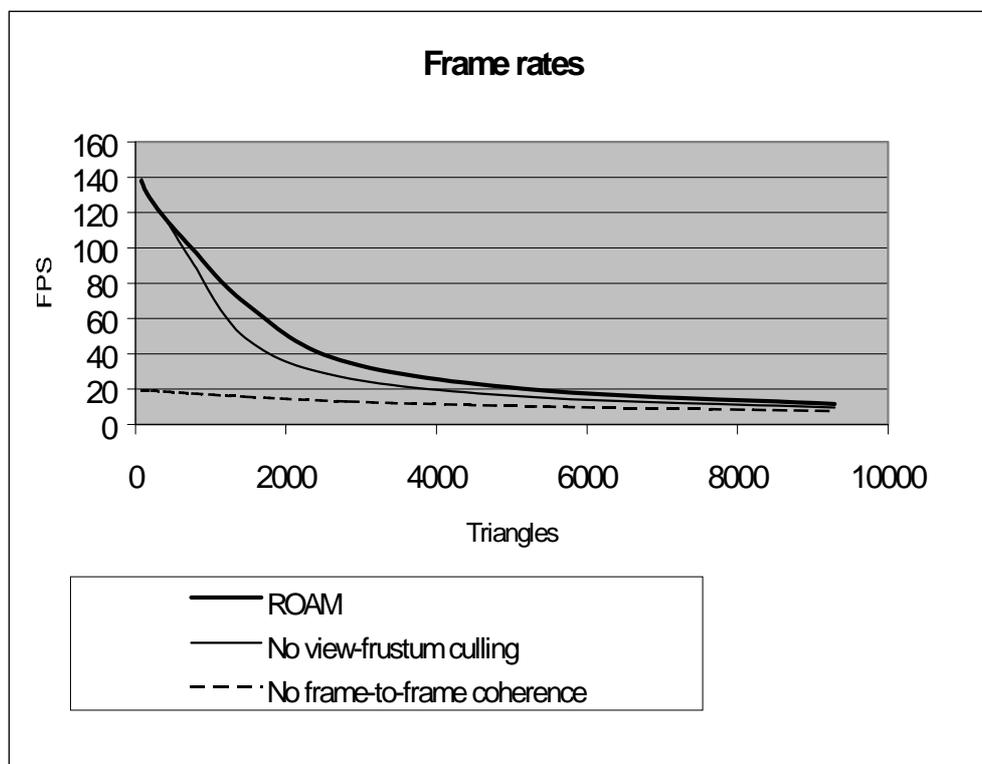


Figure 6.3: Frame rates with and without frame-to-frame coherence and view-frustum culling. ROAM displays the frame rate of the prototype with all optimisation methods included. The second series has turned off the view-frustum culling and the third does not exploit frame-to-frame coherence.

The view-frustum culling process removes all triangles that are not visible to the observer. Time is saved by not including the triangles that are outside the view frustum in the vertex array and sending them to the graphics pipeline. However, including view-frustum culling requires the computation of six clipping planes and tests for all vertices of each triangle against all six clipping planes. Performance results have shown that the computation of the six clipping planes require only 0.1% of the total view-frustum culling process. Figure 6.3 shows the frame rates with and without view-frustum culling.

If frame coherence is not exploited, the only elementary triangle operation that is necessary is the split operation. A frame-incoherent version of ROAM resets the triangle bintree every frame and splits the triangles until the maximum error of all triangles is below a given threshold. Figure 6.3 shows the frame-rate differences between a frame-coherent and a frame-incoherent version.

There is a significant difference in frame rate between a frame-to-frame coherent algorithm and an incoherent algorithm at low triangle counts. The frame-to-frame coherent algorithm is three times as fast as the incoherent algorithm at 2500 triangles per frame. The difference decreases at higher triangle counts. This is simply because the graphics pipeline requires more of the frame time to render all triangles.

View-frustum culling increase the frame rate by 35% at 2500 triangles per frame. The difference decreases at lower triangle counts since the overhead of computing six clipping planes is not compensated by the reduction of triangles. The difference does also decrease at higher triangle counts by the same reason as the difference between a frame-to-frame algorithm and an incoherent algorithm decreases. The graphics pipeline requires most of the frame time to render a large number of triangles.

6.1.2 Function Timing

The run time can be divided into three steps. The first step is performed by the ROAM algorithm, which executes all triangle splits and merges including view-frustum culling and priority computation. The second step generates the vertex arrays, while the third step renders the triangles. Table 6.2 shows the run-time per frame of these three steps.

Functions	Time/frame (ms)
ROAM	11.8
- <i>Triangle split and merges</i>	- 1.4
- <i>View-frustum culling</i>	- 8.5
- <i>Priority computation</i>	- 1.9
Vertex-array generation	0.9
Triangle rendering	2.9
Other	1.3
Total	16.9

Table 6.2: Time spent on the ROAM algorithm, vertex-array generation, and triangle rendering.

The other functions include the I/O and window management in the visualisation engine. Although the view-frustum culling requires 50% of the total frame time, it increases the frame rate as indicated by Figure 6.3. Duchaineau et al. have reached a view-frustum culling implementation that only requires 33% of the ROAM algorithm. The difference between the implementations is the reason behind the results. Optimising the view-frustum culling to a level that is equivalent to Duchaineau. et al. will increase the frame rates by 41%. Including triangle stripping and priority-computation deferral will increase the frame rates further.

7 SUMMARY AND CONCLUSION

Terrain generators have been surveyed and categorised into six groups. Several terrain generation algorithms have been evaluated and two of them, view-dependent progressive meshes (VDPM) and real-time optimally adapting meshes (ROAM), have been discussed further. VDPM generates general triangulated irregular networks, while ROAM generates networks that only consist of right-isosceles triangles. A prototype based on ROAM has been implemented, which shows good results although much optimisation work remains.

Compared to traditional terrain models, real-time terrain generators can provide drastic time savings at virtually no cost. Terrain models will take weeks to produce while a height map can be created in a few days. If real map data is used, time and cost savings are even better. T³SIM will use digital height and map data to produce terrain in real time. Another important benefit is the support of continuous level of detail. Traditional discrete levels of detail suffer from visual artefacts when switching from one level of detail to another. Several methods have been proposed to reduce the artefacts, but none has proven sufficient. Continuous level of detail changes the number of triangles and the sizes of the triangles continuously and removes any visual artefacts. A third advantage of terrain generators is that the terrain can be easily altered in real time.

ROAM has shown to generate a triangle mesh faster than VDPM. ROAM only needs to morph one vertex per triangle while VDPM needs to morph two. The validity of a ROAM elementary mesh operation is faster evaluated. ROAM avoids slivers automatically, while VDPM need extra computations to guarantee their absence. In addition, VDPM always introduce temporary slivers during vertex morphing. However, the largest reason for choosing ROAM instead of VDPM is that its time complexity is proportional to the number of triangle changes while VDPM time complexity is proportional to the full output mesh size.

A new merge operation has proven to simplify the frame-coherent split and merge process. It can merge the children of any nonleaf triangle. Since the original ROAM time complexity is proportional to the number of triangle changes per frame, frames that have to make many splits and merges will need longer frame times. These cases commonly occur when the observer travels fast or makes a quick turn. In future versions of T³SIM, where air-combat simulations within visual range will be included, aircrafts will fly on low altitudes at high velocities. The new merge operation is suitable for these situations, since it decreases the frame times of incoherent frames.

The incremental view-frustum culling increases frame rates, and is therefore an important part of ROAM. It is necessary to include in order to achieve high frame rates.

Hand-modelled terrain is not scalable. There will be a continuous need for increasing the level of detail of the terrain when hardware performance improves. On the other hand, ROAM and all other terrain-rendering algorithms are scalable and will improve the terrain with higher processor speeds.

Thus, real-time terrain generators that support continuous level of detail are superior to traditionally modelled terrain. ROAM has shown to be the fastest of all current terrain-rendering algorithms. T³SIM will therefore gain from using a product that is based on ROAM and map data.

Future work includes implementing the split and merge priority queues for progressive optimisation and exploit triangle stripping instead of vertex arrays. Incorporating priority-computation deferral, backface culling, and occlusion culling would reduce triangle counts further.

REFERENCES

- [1] J. F. Barnes, A data-dependent triangulation for hierarchical rendering, *UC Davis Student Workshop on Computing*, Sept. 1997.
- [2] M. de Berg and K. T. G. Dobrindt, On levels of detail in terrains, In *Proc. 11th Annual ACM Symp. on Computational Geometry*, Jun. 1995.
- [3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*, Springer-Verlag, 1997.
- [4] P. Blekken and T. Lilleskog, *A comparison of different algorithms for terrain rendering*, Spring semester project at CS Dept., Norwegian U. of Science and Technology, 1997.
- [5] B. Chazelle, Triangulating a simple polygon in linear time, In *Proc. 31st Annual IEEE Symp. of Computer Science*, pages 220-230, Oct. 1990.
- [6] M. Duchaineau, M. Wolinsky, D. E. Sigesti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, ROAMing Terrain: Real-time optimally adapting meshes, In *Proc. Visualization '97*, pages 81-88, 1997.
- [7] Ericsson Saab Avionics AB, *Gripen display system*, 1998.
- [8] Ericsson Saab Avionics AB, *T³SIM – training & tactical technical development simulation system*, 1998.
- [9] Ericsson Saab Avionics AB, *T³SIM system overview*, 1999.
- [10] R. E. Fayek, *3D surface modeling using hierarchical topographic triangular meshes*, Ph.D Thesis, U. of Waterloo, 1996.
- [11] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics: principles and practice*, Sec. Ed. in C, Addison-Wesley, 1996.
- [12] M. Garland and P. S. Heckbert, *Fast polygonal approximation of terrains and height fields*, Tech. Rept. CMU-CS-95-181, CS Dept., Carnegie Mellon U., Sept. 1995.
- [13] M. Garland and P. S. Heckbert, Surface simplification using quadric error metrics, In *Proc. SIGGRAPH '97*, pages 209-216, Aug. 1997.
- [14] A. Guézic, Surface simplification with variable tolerance, In *Proc. of the Sec. Int. Symp. on Medical Robotics and Computer Assisted Surgery*, MRCAS '95, Nov. 1995.
- [15] P. S. Heckbert and M. Garland, Multiresolution modeling for fast rendering, In *Proc. Graphics Interface '94*, pages 43-50, Banff, Canada, Canadian Inf. Proc. Soc., May 1994.
- [16] P. S. Heckbert and M. Garland, Survey of polygonal surface simplification algorithms, In *Multiresolution Surface Modeling Course Notes*, ACM SIGGRAPH, 1997.
- [17] H. Hoppe, Efficient implementation of progressive meshes, *Computers and Graphics*, Vol. 22, No. 1, pages 27-36, 1998.
- [18] H. Hoppe, New quadric metric for simplifying meshes with appearance attributes, *IEEE Visualization '99*, Oct. 1999.
- [19] H. Hoppe, Progressive meshes, In *Proc. SIGGRAPH '96*, pages 99-108, Aug. 1996.
- [20] H. Hoppe, Smooth view-dependent level-of-detail control and its application to terrain rendering, *IEEE Visualization '98*, Oct. 1998.
- [21] H. Hoppe, View-dependent refinement of progressive meshes, In *Proc. SIGGRAPH '97*, pages 189-198, Aug. 1997.
- [22] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, Mesh optimization, In *Proc. SIGGRAPH '93*, pages 19-26, 1993.

- [23] T. Lilleskog, *Continuous level of detail*, Master's thesis, CS Dept., Norwegian U. of Science and Technology, Feb. 1998.
- [24] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust and G. A. Turner, Real-time, continuous level of detail rendering of height fields, In *Proc. SIGGRAPH '96*, pages 109-118, Aug. 1996.
- [25] S. McNally, *Binary triangle trees and terrain tessellation*, Longbow Digital Arts, Aug 1999, URL: <http://www.longbowdigitalarts.com/seumas/progbintri.html>
- [26] M. C. Miller, *Multiscale compression of digital terrain data to meet real time rendering rate constraints*, PhD Thesis, U. of California, Davis, 1995,
- [27] F. P. Preparata, M. I. Shamos, *Computational geometry: an introduction*, Springer-Verlag, 1998.
- [28] S. Röttger, W. Heidrich, P. Slusallek and H. Seidel, Real-time generation of continuous levels of detail for height fields, *6th Int. Conf. in Central Europe on Computer Graphics and Visualization '98*, Feb. 1998.
- [29] M. Woo, J. Neider, T. Davis, *OpenGL programming guide*, Third. Ed., Addison-Wesley, 1999.
- [30] J. C. Xia, A. Varshney, Dynamic view-dependent simplification for polygonal models, In *Proc. Visualization '96*, pages 327-334, IEEE Comput. Soc. Press, 1996.
- [31] A. Ögren, Illumination and shading models, In *Proc. USCCS&I'99*, Jun. 1999.

APPENDIX A: COMPUTATIONAL GEOMETRY CONCEPTS

A.1 Quadtree

A *quadtree* is a rooted quaternary tree that hierarchically subdivides a rectangular area containing a set of points. For the purpose of terrain rendering based on a regular square grid of height samples, the rectangular area is restricted to a square and the set of points is restricted to a regular grid of $(n+1)^2$ equally spaced height samples. Usually, $n = 2^k$, for some nonnegative integer k .

The root node corresponds to the whole square, while its four children each correspond to a quadrant of the square. The definition is recursive, i.e. each internal node of the quadtree corresponds to a square region, while the four children each correspond to a quadrant of the parent square. Every internal node is labelled NE, NW, SW, and SE, to indicate the specific square they represent.

The construction of a quadtree takes $O(n^2)$ time and uses $O(n^2)$ space, i.e. it is linear in the number of input points [3, 27].

A quadtree is suitable for simple structure- and view-dependent multiresolution triangulations of terrain. Low-level nodes correspond to a rough triangulation, while high-level nodes correspond to finer triangulations. A quadtree can be constructed off-line, but it is traversed for each frame. The recursion is stopped as soon as a suitable level of detail is found. An example of a quadtree is shown in Figure A.1. Each square is triangulated according to a predetermined scheme.

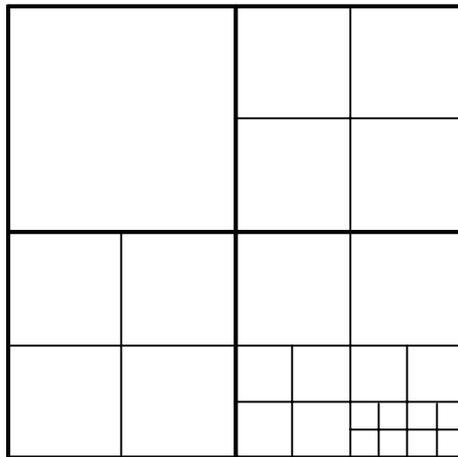


Figure A.1: A quadtree with higher resolution at regions down to the right.

The generalisation of a quadtree into three dimensions is known as an *octree*, of which each node corresponds to a cubic region.

A.2 Kd-tree

Given a finite set of real numbers P , a common problem is to find which elements lie within a specified interval. This is known as an *orthogonal range query*. A straightforward solution is to check each point against the interval, resulting in a linear time algorithm. A faster solution would be to store the numbers in a simple binary search tree, which reduces the time complexity to $O(\log n + k)$, where n is the number of elements in P and k is the number of retrieved points [3].

An extension to this problem is, given a finite set of points P in the plane, to find all points within a specified rectangle. The points can be height samples in a terrain-rendering algorithm or vertices in a triangle mesh.

A-2 Appendix A: Computational Geometry Concepts

A *kd-tree*, known as a *multidimensional binary tree*, is also stored as a binary search tree, but it is interpreted differently. For the application described above, each internal node stores a line splitting the plane into two parts. These lines are orthogonal, but normally only considered vertical and horizontal. The root node contains a vertical line through the median x-coordinate of the points in P , which splits the plane into a left and right region. The left child node contains a horizontal line through the median y-coordinate of the points in the left region, which splits the left region into an upper and lower region. Similarly, the right child node contains a horizontal line through the median y-coordinate of the points in the right region, which splits the right region into an upper and lower region. This process continues splitting the plane with vertical lines at nodes of even depth and horizontal lines at nodes of odd depth [3, 27]. Figure A.2 illustrates an example of plane subdivision and the corresponding kd-tree.

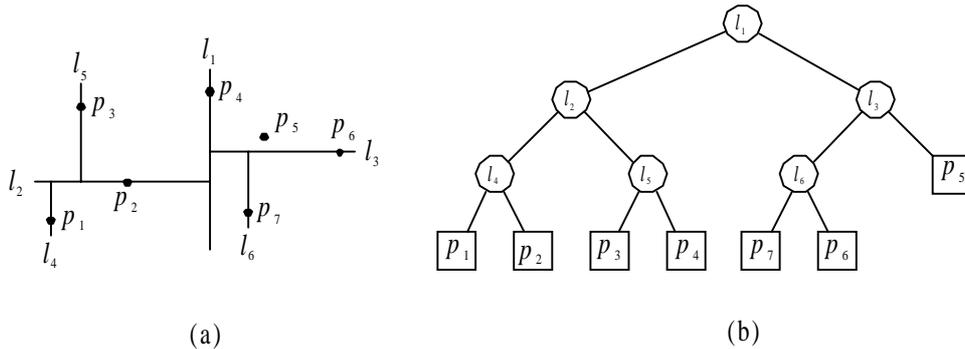


Figure A.2: a) A plane subdivision. b) The corresponding kd-tree.

Originally, k stood for the dimension of the tree, which in this case is two. Nowadays, however, they are called 2-dimensional kd-trees.

The retrieval time complexity of a kd-tree is of $O(\sqrt{n} + k)$, where n is the number of points in the plane and k is the number of retrieved points. It uses $O(n)$ storage and can be constructed in $O(n \log n)$ time. The retrieval time is more important, since it is a real-time process, as opposed to the construction time.

A.3 Range tree

A *range tree* is a multi-level binary tree that improves the retrieval time of kd-trees. A kd-tree alternates the splitting process on x- and y-coordinates. A range tree, on the other hand, controls the splitting process by storing subtrees at each node. The query of finding all points with an x-coordinate within a specified range can be done in time complexity $O(\log n + k)$. Since the range is continuous, the output points belongs to a number of *canonical subsets*, each of which is associated to a node and contains all points in the subtree of that node. For example, the canonical subset of the root node contains all points, while the canonical subset of a leaf contains only the point associated with that leaf [3].

Each node in the tree contains its associated canonical subset stored as a balanced binary search tree on the y-coordinates of the points. The main tree is called the first-level tree, while the canonical-subset trees are known as second-level trees. A range tree can easily be extended to include more dimensions.

Although this requires $O(n \log n)$ storage, it improves retrieval time to $O(\log^2 n + k)$. A technique known as *fractional cascading* can improve retrieval time further to $O(\log n + k)$. Thus, range trees improve performance over kd-trees at the cost of increased storage.

A.4 Voronoi Diagram

The Voronoi diagram can be defined for any number of dimensions, but since terrain-rendering algorithms are based on planar input, this short summary is restricted to the two-dimensional case. The

Voronoi diagram of a finite set of two-dimensional points $\mathbf{P} = \{p_1, p_2, \dots, p_n\} \subseteq \mathfrak{R}^2$ is a subdivision $\mathbf{V} = \{v_1, v_2, \dots, v_n\}$ of \mathfrak{R}^2 with $p \in v_i$ implicates

$$\|p - p_i\|_2 < \|p - p_j\|_2 \quad \forall p_j \in \mathbf{P}, i \neq j, \quad (\text{A.1})$$

i.e. each set $v_i \in \mathbf{V}$ consists of all points that is closer to p_i than any other point in \mathbf{P} . The Voronoi diagram of \mathbf{P} is denoted $\text{Vor}(\mathbf{P})$. Each set v_i in \mathbf{V} is called a *Voronoi cell* or a *Voronoi polygon*, and is denoted $V(p_i)$. The points in \mathbf{P} are called *sites* [3, 27]. A set of sites and their corresponding Voronoi diagram is shown in Figure A.3.

A bisector of two sites $p, q \in \mathbf{P}$ is defined as the perpendicular line of the straight line passing through both p and q . This bisector splits the plane into two halves, of which one contains p and one contains q . The distance from the bisector to p equals the distance to q . Define h to be a function by setting $h(u, v)$ to the open half-plane defined by the bisector of u and v that contains u . This yields $r \in h(u, v)$ if and only if

$$\|r - u\|_2 < \|r - v\|_2. \quad (\text{A.2})$$

Thus

$$V(p_i) = \bigcap_{1 \leq j \leq n, j \neq i} h(p_i, p_j), \quad (\text{A.3})$$

i.e. $V(p_i)$ is the open convex polygonal region defined by the intersection of $n-1$ half-planes as illustrated in Figure A.3. The edges of the polygonal region either are line segments, half lines, or full lines. If the region is bounded, it consists only of line segments, while if it is unbounded, it also consists of at least two half-lines. If all sites in \mathbf{P} are collinear, all edges are full lines.

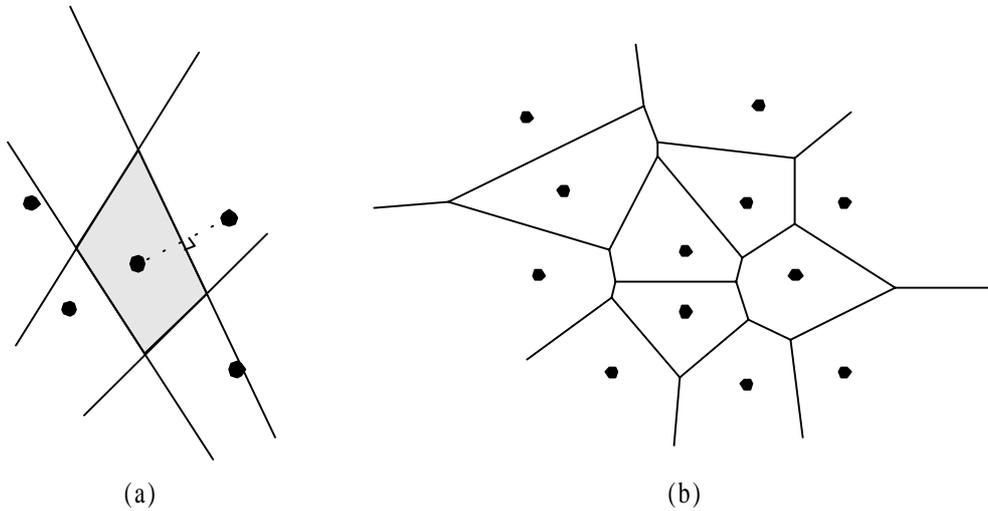


Figure A.3: a) The Voronoi region for a point is the intersection of all halfplanes. b) A Voronoi diagram.

Given a point $q \in \mathfrak{R}^2$, a *largest empty circle* of q with respect to \mathbf{P} is the largest circle centred at q that does not contain any points from \mathbf{P} in its interior. Given this definition, it can be proven that a point q is a vertex of $\text{Vor}(\mathbf{P})$ if and only if the largest empty circle of q with respect to \mathbf{P} contains three or more sites on its boundary. Also, the bisector between two sites u and v in \mathbf{P} contains an edge of $\text{Vor}(\mathbf{P})$ if and only if there is a point $q \in \mathfrak{R}^2$ such that the largest empty circle of q with respect to \mathbf{P} contains both u and v on its boundary but no other sites.

The Voronoi diagram can be computed by a plane sweep algorithm, known as *Fortune's algorithm*, in $O(n \log n)$ time, which has been proven to be optimal [3, 27].

A.5 Polygon Triangulation

The triangulation of a polygon can be computed by a large number of algorithms. In 1990, B. Chazelle developed a linear time algorithm [5]. This section presents a simpler $O(n \log n)$ algorithm that triangulates a simple polygon with n vertices. A *simple polygon* is a polygon that does not intersect itself. It is easily proved that every simple polygon can be triangulated to $n - 2$ triangles [3].

Given a simple polygon P with n vertices, the algorithm creates a triangulation during two steps. The first step partitions the polygon into a set of monotone pieces, while the other triangulates the pieces.

A.5.1 Partitioning a Simple Polygon into Monotone Pieces

A simple polygon is called *monotone* with respect to a line l if for any other line l' , perpendicular to l , the intersection of the polygon with l' is connected, i.e. the intersection is a line, a point, or empty. In particular, a simple polygon is called *y-monotone* if it is monotone with respect to the y-axis.

Denote the y-coordinate of a point p by p_y . Similarly, denote the x-coordinate by p_x . A point p is *below* another point q if $p_y < q_y$ or $p_y = q_y$ and $p_x > q_x$. A point p is *above* a point q if q is below p .

The vertices of a polygon is distinguished into five categories:

1. A vertex v is a *start vertex* if its two neighbours lie below it and the interior angle at v is less than π .
2. A vertex v is a *split vertex* if its two neighbours lie below it and the interior angle at v is greater than π .
3. A vertex v is an *end vertex* if its two neighbours lie above it and the interior angle at v is less than π .
4. A vertex v is a *merge vertex* if its two neighbours lie above it and the interior angle at v is greater than π .
5. A vertex v is a *regular vertex* if it is none of the above, i.e. v has one neighbour lying above and the other below.

The first four types, i.e. start, split, end, and merge vertices, are called *turn vertices*. A polygon with vertices from all five categories is shown in Figure A.4 (a).

De Berg et al. [3] have shown that a polygon is y-monotone if and only if it has neither split nor merge vertices. Thus, all split and merge vertices have to be removed to obtain a partition of monotone polygon pieces.

The split vertices are removed by sweeping from the topmost vertex down to the lowest. If a split vertex v is found, let e_j be the edge immediately to the left of v and e_k be the edge immediately to the right of v . Connect v to the lowest vertex between e_j and e_k , but above v , or, if no such vertex exist, connect v to the upper vertex neighbour of e_j .

The merge vertices are removed in a similar matter. If a merge vertex u is found, let e_j be the edge immediately to the left of u and e_k be the edge immediately to the right of u , as before. Connect u to the highest vertex between e_j and e_k , but below u . If no such vertices exist, connect u to the lower vertex neighbour of e_j .

Removing all split and merge vertices partitions P into a set of monotone polygons. The time complexity of this algorithm is $O(n \log n)$, while the space complexity is linear.

A.5.2 Triangulating a Monotone Polygon

Assume that P is *strictly y-monotone*, i.e. y-monotone without any horizontal edges. Construct two *chains*, one consisting of all the vertices on the left side of P sorted from the top down and the other consisting of all the vertices on the right side, also sorted from the top down. Construct a sequence $u_1, u_2,$

..., u_n consisting of all vertices sorted by decreasing y-coordinate. If two vertices have the same y-coordinate, the leftmost vertex precedes the other. Finally, use a stack S to store the vertices. The algorithm is straightforward:

Push the first two vertices in the sequence onto S . Check each vertex u_i left in the sequence to see if it is on the other chain than the vertex on top of S . If so, pop all vertices from S and insert an edge from u_i to all popped vertices, except the last one. Push back the vertices u_i and u_{i-1} onto the stack.

If the vertex on top of S is on the same chain as u_i , then pop all vertices in S . Insert edges between u_i and all popped vertices except for the first, as long as they are inside P . Push back the vertex u_i and the last popped vertex onto S .

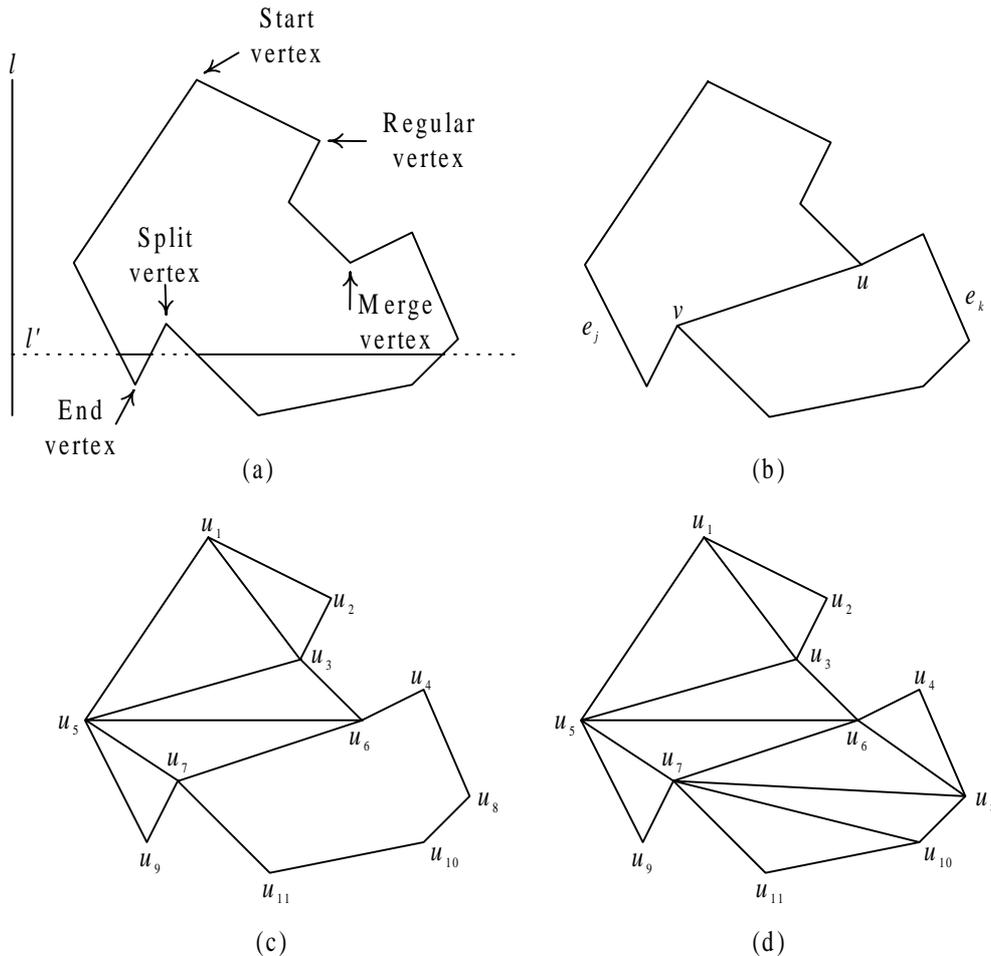


Figure A.4: a) Start, regular, split, merge, and end vertices. b) Partitioning a simple polygon into monotone polygons. c) Triangulating the upper monotone polygon. d) Triangulating both monotone polygons.

The last step is to add edges from u_n to all vertices in the stack, except the first and last one. This produces a triangulation of P in linear time.

Thus, this composite algorithm for triangulating an arbitrary simple polygon takes $O(n \log n)$ time.

A.6 Delaunay Triangulation

The dual graph of the Voronoi diagram is the *Delaunay graph* [27]. Given a set of points P in the plane, the *Delaunay triangulation* creates a triangle mesh from the Delaunay graph that maximises the minimum angle of all triangles. Small angles cause slivers, which in turn can introduce large

approximation errors of terrain and aliasing problems for texture maps. Since the Delaunay triangulation maximises the minimum angle of all triangles, it generally produces triangle meshes of better visual quality than other methods.

Let T be a triangulation of a set of points P in the plane consisting of n triangles. Furthermore, let $A(T)$ denote the *angle-vector* $(\alpha_1, \alpha_2, \dots, \alpha_{3n})$ containing all $3n$ angles of the triangles in T , sorted by increasing value. Let T' be another triangulation of P with angle-vector $A(T') = (\alpha'_1, \alpha'_2, \dots, \alpha'_{3n})$. $A(T)$ is defined to be larger than $A(T')$, denoted $A(T) > A(T')$, if there is an $i \in \mathbf{Z}_{3n}$ such that $\alpha_j = \alpha'_j$ for all $j < i$ and $\alpha_i > \alpha'_i$. Other relational properties follow similarly.

A triangulation is called *angle-optimal* if $A(T) \geq A(T')$ for all triangulations T' of P [3]. A Delaunay triangulation will be shown an angle-optimal triangulation.

Consider a triangulation T of four points p_i, p_j, p_k , and p_l , with triangles (p_i, p_j, p_l) and (p_j, p_k, p_l) and angles v_1, v_2, v_3, v_4, v_5 , and v_6 , as in Figure A.5. Let e be an edge between p_j and p_l . Construct a new triangulation T' by *flipping* e to e' between p_i and p_k , which changes the angles to $v'_1, v'_2, v'_3, v'_4, v'_5$, and v'_6 . The edge e is called *illegal* if

$$\min_{1 \leq i \leq 6} v_i < \min_{1 \leq i \leq 6} v'_i. \tag{A.4}$$

This definition applies to all non-boundary edges. A *legal triangulation* is a triangulation that does not contain any illegal edges.

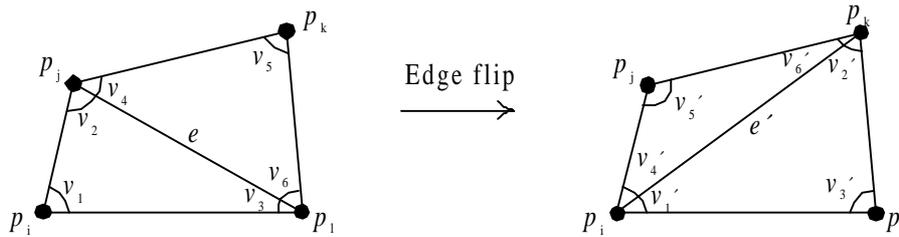


Figure A.5: An edge flip operation.

Given a Voronoi diagram $\text{Vor}(P)$ of P , the *Delaunay graph* of P , denoted $DG(P)$, has a vertex at each point in P and edges between any two vertices whose corresponding Voronoi cells in $\text{Vor}(P)$ are adjacent [27]. Figure A.6 (b) shows a Delaunay triangulation of the point set in Figure A.3 (b).

A finite set of points $P \subseteq \mathfrak{R}^2$ is in *general position* if there is no circle whose boundary contains four points in P . If P is in general position, then all vertices in $\text{Vor}(P)$ is of degree three, and as a consequence, all bounded regions in $DG(P)$ are triangles. In this case, the triangulation of P with edges between every pair of points that are adjacent in $DG(P)$ is known as the *Delaunay triangulation* of P . If P is not in general position, then the Delaunay graph will contain a convex polygon with at least four vertices. The Delaunay triangulation is extended to include triangulations of non-general point sets, where the polygons are triangulated. The triangulation of a polygon was covered Section A.5. Thus, a Delaunay triangulation of a set of points P is unique if and only if P is in general position [3].

There are three important properties of the Delaunay triangulation. Let T be a triangulation of a set of planar points P .

1. T is a Delaunay triangulation if and only if the circumcircle of any triangle in T does not contain a point of P in its interior.
2. T is legal if and only if T is a Delaunay triangulation of P .
3. If T is a Delaunay triangulation of P then it maximises the minimum angle over all triangulations of P .

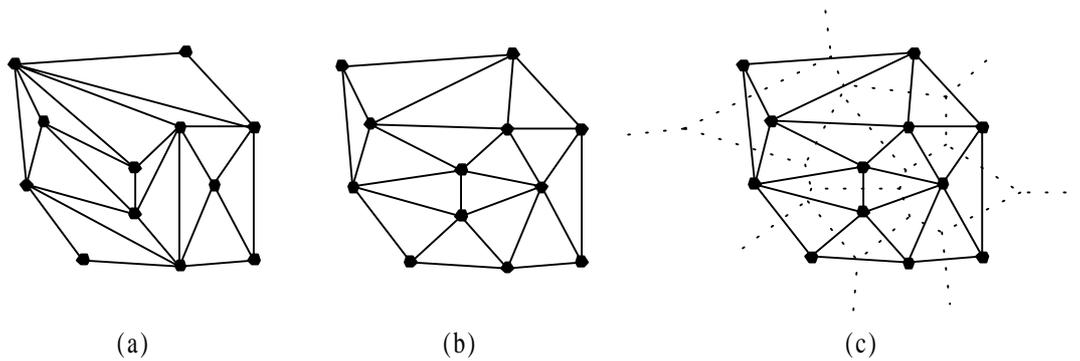


Figure A.6: a) An arbitrary triangulation of the point set in Figure A.3 b). (b) A Delaunay triangulation of the same point set. c) The underlying Voronoi diagram shows that the Delaunay triangulation contains an edge between any two adjacent sites.

There are several methods for computing a Delaunay triangulation of a set of planar points P . Blekken et al. [4] divide the methods into five categories:

- *Two-step algorithms* are based on arbitrary triangulations but rearrange them into Delaunay triangulations.
- *Incremental algorithms* are based on Delaunay triangulation on a subset of P , but refine the triangulation while maintaining the Delaunay property when inserting new points.
- *Divide-and-conquer algorithms* construct the Delaunay triangulation by recursively splitting the point set into two halves, constructing Delaunay triangulations for each half, and merging the halves while maintaining the Delaunay property.
- *Sweep-line algorithms* compute the Voronoi diagram and transform it into a Delaunay triangulation using a sweep line.
- *Three-dimensional algorithms* compute the three-dimensional convex hull of the point set and project the lower portion onto the x-y plane.

There are two extensions to Delaunay triangulation. *Constrained Delaunay triangulation* is based on a set of points P and a set of constrained edges E and is constructed by ensuring that the *circumcircle* of each triangle does not contain any point of P that is *visible* from all three vertices of the triangle. A point p_1 is visible from a point p_2 if the straight line between them does not cross the interior of any of the edges in E . A constrained Delaunay triangulation does not necessarily fulfil the Delaunay property.

Conforming Delaunay triangulation improves the constrained Delaunay triangulation by adding more points into P in order to fulfil the Delaunay property.

A.7 Data-Dependent Triangulation

While Delaunay triangulation uses two-dimensional information only, *data-dependent triangulation* algorithms achieve more accurate approximations of the triangle meshes by considering the topology of the terrain [10]. However, they generally introduce slivers, which Delaunay triangulation avoids by maximising the minimum angle of all triangles. Slivers can introduce aliasing effects in texture maps, which reduce the visual quality of the terrain.

Several papers survey many different data-dependent triangulation algorithms [1, 10, 12]. An incremental Delaunay triangulation can be generalised to a data-dependent triangulation algorithm by iteratively flipping the edges of the Delaunay triangulation. An edge is flipped if the new edge approximates the terrain better than the previous. This introduces illegal edges and destroys the Delaunay property in favour of error reduction. What triangulation algorithm that should be chosen depends on the application and priority of visual quality and approximation error.

APPENDIX B: USER MANUAL

The prototype, roam.exe, can be executed on any IBM compatible PC with an Intel Pentium III processor running Microsoft Windows 95 or 98. The following three files must be copied to C:\Windows\System\; glut.dll, glu.dll, and opengl32.dll.

Both the keyboard and the mouse can control the observer.

Key	Action
0	Return to start position.
1	Accelerate.
2	Turn up.
3	Decelerate.
4	Rotate to the left.
5	Stop.
6	Rotate to the right.
7	Turn left.
8	Turn down.
9	Turn right.
Esc	Exit

Table B.1: The keyboard control keys for the prototype.

By holding down the left or right mouse button and drawing the mouse at one of the four directions controls the observer according to Table B.2:

Mouse button	Mouse direction	Action
Left	Up	Turn down.
	Down	Turn up.
	Left	Rotate to the left.
	Right	Rotate to the right.
Right	Up	Accelerate.
	Down	Decelerate.

Table B.2: The mouse control for the prototype

APPENDIX C: GLOSSARY

Backface culling	The removal of primitives facing away from the observer, i.e. those on the backside of an object.
Data-dependent triangulation	A set triangulation methods that use the heights and other information of a set of vertices to achieve a more accurate triangulation than Delaunay triangulation.
Decimation methods	Triangulation simplification methods that simplify an initial triangulation containing all data points during multiple passes. During each pass, a vertex, edge, or triangle is removed and the mesh is retriangulated.
Delaunay triangulation	A triangulation method that maximises the minimal angle of all triangles. Constrained Delaunay triangulation is an extension to Delaunay triangulation that includes a specified set of edges. Conforming Delaunay triangulation is an extension to constrained Delaunay triangulation, which adds vertices to guarantee the Delaunay property.
Feature	A vertex that contains important information about the terrain, such as peaks, ridges and valleys. Also known as a critical point.
Feature methods	Triangulation simplification methods whose triangulations are based on features only. Constrained Delaunay triangulation is often used if certain edges have to be included.
Height field	A two-dimensional regular grid of equally spaced height samples.
Hierarchical subdivision methods	Triangulation simplification methods that divide the terrain recursively into regions to form a hierarchical tree, in which each node represents a specific region of the terrain. The children of a node together represent the same region as the parent, but at higher level of detail.
Level of detail (LOD)	Discrete level of detail is a method for displaying the same object at different levels of detail from different distances. Continuous level of detail is a method of computing the correct level of detail for each region of an object. Different regions of the object are displayed at different levels of detail simultaneously.
Kd-tree	A binary tree that recursively subdivides a space. Each node of a k -dimensional kd-tree divides a k -dimensional space by a $(k-1)$ -dimensional plane. Two such planes in two nodes of level i and j are orthogonal if $i \neq j$.
Multiresolution modelling	The modelling of an object at different discrete levels of detail.
Occlusion culling	The removal of primitives occluded by other primitives and therefore not visible to the observer.
Optimal methods	Triangulation methods that find the optimal approximation of a grid.
Polygon triangulation	The division of a polygon into a triangle mesh.

Popping	An aliasing artefact appearing when changing from one discrete level of detail to another.
Progressive Meshes (PM)	A continuous level-of-detail method for view-dependent terrain rendering.
Quadtree	A quaternary tree that recursively divides an area to four equal regions at each node.
Range tree	A multi-dimensional binary tree. Each node contains another binary tree.
Real-time Optimally Adapting Meshes (ROAM)	A continuous level-of-detail method for view-dependent terrain rendering.
Refinement methods	Triangulation simplification methods that start with a coarse approximation and refine it during multiple passes until the appropriate amount of triangles is found or until an error goes below a certain limit.
Regular grid methods	Triangulation simplification methods that only use every k^{th} row and column of the height field as vertex set of the triangulation.
Sliver	A triangle with at least one very small angle.
T³SIM	A flight simulator that is developed by Ericsson Saab Avionics AB for tactical training in real-time man-in-the-loop air-combat simulations.
Terrain	The graph of a continuous function $f: \mathfrak{R}^2 \rightarrow \mathfrak{R}$.
Tile	A square area in a terrain that is represented by several levels of detail.
Triangle fan/strip	A set of triangles adjacent so that every new triangle adds one new vertex. For n triangles, only $n+2$ vertices are necessary. Long triangle strips or fans reduce rendering time. The difference between a triangle strip and a triangle fan is the composition of triangles.
Triangle mesh	A mesh consisting only of triangles such that each edge is adjacent to at most two triangles.
Triangulated irregular network (TIN)	A triangle mesh that consists of non-overlapping variable-sized triangles.
View-frustum culling	The removal of all primitives outside the view frustum, i.e. a bounded volume containing the observer's field of view.
Voronoi diagram	Given a set of points \mathbf{P} in the plane, a Voronoi diagram is a subdivision of the plane into regions such that each region is associated to exactly one point in \mathbf{P} and contains all points in the plane that is closer to that point than any other.
Wedgie	A volume that extends vertically above and below a triangle with a thickness defined by children triangles.