Efficient terrain triangulation and modification for game applications

Sundar Raman

2008

Sundar, R. (2008). Efficient terrain triangulation and modification for game applications. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47472>

<https://doi.org/10.32657/10356/47472>

Nanyang Technological University

Downloaded on 26 Apr 2023 18:43:23 SGT

2x
748

15mm

EFFICIENT TERRAIN TRIANGULATION AND MODIFICATION FOR GAME APPLICATIONS

SUNDAR RAMAN

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2008

Abstract

The most common representation of a terrain is a heightmap, which needs to be converted into a triangular mesh before rendering. In spite of many existing dynamic real-time algorithms for terrain LOD, an efficient offline terrain mesh representation is not well explored. The fundamental problem of producing an unstructured, yet well-sized, triangular mesh is solved by the properties of Delaunay triangulation, which maximizes the minimum angle for a given point set.

Hence, an intuitive algorithm is proposed, based on constrained conforming Delaunay triangulation, for generating such a terrain. The density of triangulation in different regions of the terrain is determined by its flatness, as seen from a heightmap, and a control map. Additionally, user parameters can be used to vary the level of detail in different regions of the terrain. For further optimizing the generated terrain, novel techniques using constrained data dependent triangulation are proposed.

Tracks and other objects found in a game world can be applied over the terrain using the “stenciling”, “stitching” and “track overlaying” algorithms. Varying levels of detail can be preserved when applying these objects over the terrain, guaranteeing smoothness especially when tracks are laid over the terrain.

I have implemented all algorithms as 3dsMax plugins, and some have been used to generate beautiful looking detailed terrains and tracks in a soon-to-be-released game for the Xbox 360 and PS3 consoles, “Baja: Edge of Control”. I am sure the results speak for the efficiency and usefulness of the proposed methods.

Contents

Figures.....v

Algorithms viii

Acknowledgementsix

1 Introduction.....1

1.1 Motivation1

1.2 Objective & Scope.....3

1.3 Contribution.....5

1.4 Outline of Thesis5

2 Background8

2.1 Types of Meshes.....8

2.2 Desirable properties of Mesh Generators10

2.3 Mesh Generation Methods11

2.4 Convex Hull.....12

2.4.1 Constructing 2D convex hulls13

2.4.2 Andrew’s Monotone Chain algorithm.....14

2.4.3 Constructing 3D convex hulls16

2.4.4 3D Incremental algorithm.....17

2.5 Delaunay Triangulation21

2.5.1 Constructing Delaunay triangulations23

2.5.2 Relationship between convex hulls and Delaunay triangulations26

2.6 Constrained Delaunay Triangulation.....28

3 Related Work31

3.1 Constrained Conforming Delaunay Triangulation31

3.1.1 Chew’s first algorithm.....33

3.1.2 Ruppert’s algorithm.....35

3.1.3	Chew's second algorithm	37
3.2	Data Dependent Triangulation	38
3.2.1	DDT cost functions.....	38
3.2.2	DDT edge swap	41
3.3	Terrain Generation Methods.....	43
3.3.1	Regular Grid vs Triangulated Irregular Network	43
3.3.2	Automatic Extraction.....	45
3.3.3	Radial Sweep algorithm	45
3.3.4	Advancing Front algorithm	46
3.3.5	Greedy insertion algorithms	47
3.3.6	Improved TIN refinement.....	48
3.4	Quad-Edge data structure	49
4	Proposed Algorithms	55
4.1	An "Intuitive" triangulation algorithm	55
4.2	New CCDT algorithm	56
4.3	Proposed Constrained DDT algorithm	60
4.3.1	New DDT cost functions	60
4.3.2	Look-Ahead algorithm	63
4.4	Terrain Simplification algorithm	66
4.4.1	Existing techniques.....	67
4.4.2	Proposed method	68
5	Terrain Generation.....	71
5.1	Heightmap	71
5.2	Control Map.....	72
5.3	User constraints	72
5.4	Terrain Generation algorithm.....	74
5.5	Terrain Optimization algorithms	76
5.6	Results	77
5.6.1	Terrain generation	78
5.6.2	Terrain optimization	80
5.6.3	Performance comparison of different methods	83
5.6.4	Application in retail game	87
6	Terrain Modification	89
6.1	Stenciling.....	89

6.2 Stitching.....92

6.3 Track Overlaying.....94

6.4 Results97

 6.4.1 Stenciling and Stitching.....97

 6.4.2 Track overlaying.....99

 6.4.3 Application in retail game102

7 Conclusion104

References.....106

Publications113

Figures

Figure 1.1: Screenshot from “Super Mario Bros.”, 1985	2
Figure 1.2: Screenshot from “Colin McRae DiRT”, 2007	2
Figure 1.3: Screenshot from “Baja: Edge of Control”, 2008.....	2
Figure 2.1: Structured and unstructured meshes.....	9
Figure 2.2: A set of points and its convex hull	12
Figure 2.3: Monotone chain algorithm illustration.....	15
Figure 2.4: Visibility test	18
Figure 2.5: Horizon as seen from p_r	18
Figure 2.6: Incremental construction: before and after adding p_r	18
Figure 2.7: Splitting facets when p_r lies on the hull	19
Figure 2.8: Bipartite graph showing conflicts	20
Figure 2.9: Determining new conflicts for new face f	20
Figure 2.10: A Delaunay triangulation	22
Figure 2.11: Each edge on the convex hull is Delaunay.....	22
Figure 2.12: Empty circumcircle property of the Delaunay triangulation.....	22
Figure 2.13: After edge swap operation on locally Delaunay edge e	24
Figure 2.14: Lower 3D convex hull projected onto the xy plane	26
Figure 2.15: Bold segments; edge e and triangle t are both constrained Delaunay.....	29
Figure 2.16: CDT construction by segment insertion and retriangulation.....	29
Figure 3.1: Vertex insertion at the circumcenter of a skinny triangle	33
Figure 3.2: A mesh generated by Chew’s first algorithm.....	34
Figure 3.3: Circumcenter of a skinny triangle outside the triangulation	34

Figure 3.4: A mesh generated by Ruppert's algorithm.....	35
Figure 3.5: Splitting encroached segments recursively	36
Figure 3.6: Splitting skinny triangle and maintaining Delaunay property	36
Figure 3.7: Encroached segment split by Chew's second algorithm	37
Figure 3.8: Initial triangulation for computing cost.....	39
Figure 3.9: Edge swap in Lawson's algorithm	41
Figure 3.10: A Regular Grid representation with 4096 vertices.....	43
Figure 3.11: A TIN representation with just 512 vertices	44
Figure 3.12: Mesh produced by automatic extraction	45
Figure 3.13: After the radial sweep and shaping passes	46
Figure 3.14: Terrain generated by advancing front algorithm.....	46
Figure 3.15: Terrains produced by Algorithm 3 and Algorithm 4.....	48
Figure 3.16: Pedrini's approach on the Lake Crater elevation map	49
Figure 3.17: Representation of one quad-edge	50
Figure 3.18: Edge relationships of a quad-edge	51
Figure 3.19: Illustration of a bug in the <i>Locate ()</i> procedure	54
Figure 4.1: Elimination of a skinny triangle – comparison with Chew's algorithm	59
Figure 4.2: Initial triangulation for computing cost.....	61
Figure 4.3: Initial triangulation.....	65
Figure 4.4: Possible Look-Ahead outputs.....	65
Figure 4.5: Basic edge collapse operation	69
Figure 5.1: Heightmap and corresponding triangular mesh	71
Figure 5.2: Control map showing control regions in white	72
Figure 5.3: User parameters for terrain generation algorithm	73
Figure 5.4: User interfaces of terrain generation and optimization plugins	78
Figure 5.5: Inputs heightmap and control map	79
Figure 5.6: Output terrain mesh: high and low resolutions	79
Figure 5.7: Delaunay mesh	80
Figure 5.8: Constrained DDT meshes with 0° and 20° minimum angles.....	80

Figure 5.9: Simplified meshes with inputs from Figure 5.8	81
Figure 5.10: Shaded view of Delaunay mesh	82
Figure 5.11: Shaded view of Constrained DDT mesh with 0° minimum angle	82
Figure 5.12: Shaded view of Constrained DDT mesh with 20° minimum angle	82
Figure 5.13: Error reduction with DDT, 0° minimum angle	83
Figure 5.14: Triangle count reduction with DDT, 0° minimum angle	84
Figure 5.15: Execution times with DDT, 0° minimum angle	84
Figure 5.16: Error reduction with DDT, 20° minimum angle	84
Figure 5.17: Triangle count reduction with DDT, 20° minimum angle	85
Figure 5.18: Execution times with DDT, 20° minimum angle	85
Figure 5.17: A large open world in “Baja: Edge of Control”	87
Figure 5.18: Detailed terrain region with lakes, cliffs and valleys	87
Figure 5.19: Details clearly seen even in far-away regions of the terrain	88
Figure 6.1: Before and after stenciling a plane over a bigger plane	89
Figure 6.2: Before and after stitching a plane over a bigger plane	92
Figure 6.3: User interfaces of terrain modification plugins	98
Figure 6.4: Input terrain and track meshes	98
Figure 6.5: Effect of stenciling and stitching a track over the terrain	98
Figure 6.6: Input terrain with a dense control region	100
Figure 6.7: Detailed track mesh before overlay	100
Figure 6.8: Terrain overlaid with track, wireframe view	100
Figure 6.9: Terrain overlaid with track, shaded edge view	101
Figure 6.10: Terrain overlaid with track, fully shaded view	101
Figure 6.11: Zoomed-in top view, showing perfect preservation of triangles	102
Figure 6.12: A flat looking detailed track over the terrain	103
Figure 6.13: Smooth track region in spite of bumps in the terrain	103

Algorithms

Algorithm 2.1:	3D incremental convex hull	18
Algorithm 2.2:	Optimized visibility testing using conflict graph	20
Algorithm 2.4:	Delaunay triangulation algorithm	28
Algorithm 3.1:	DDT edge swapping with Lawson’s algorithm	42
Algorithm 3.2:	Locate() procedure to search for a vertex	53
Algorithm 4.1:	Intuitive triangulation algorithm	56
Algorithm 4.2:	Proposed CCDT algorithm.....	58
Algorithm 4.3:	Computing cost based on maximum error	62
Algorithm 4.4:	Computing cost based on mean square error	62
Algorithm 4.5:	Constrained DDT with Look-Ahead algorithm	65
Algorithm 4.6:	Terrain simplification algorithm	69
Algorithm 5.1:	Terrain generation overview	74
Algorithm 5.2:	Terrain generation flowchart.....	75
Algorithm 5.3:	Optimization of mean square error	76
Algorithm 5.4:	Optimization of triangle count	77
Algorithm 6.1:	Stenciling algorithm	91
Algorithm 6.2:	Stitching algorithm.....	94
Algorithm 6.3:	Track Overlaying algorithm	97

Acknowledgements

I would like to thank my supervisor, Dr. Zheng Jianmin, Nanyang Technological University, for his technical and moral support during the course of the project. Many difficult concepts were clarified by him, which helped greatly in the implementation of complex algorithms.

Secondly, I would like to thank the staff of TQ Global Pte. Ltd., Singapore, for their continuous support and guidance throughout the duration of my project. Whenever I was stuck at any point, even a short discussion with them would set me back on the right track.

All of the work was done at gameLAB annexe, which is also the company premise of TQ Global Pte. Ltd. I extend my heartfelt gratitude for the excellent work environment and ideal infrastructure I was provided with, which proved conducive for research, implementation and testing.

Finally, I would like to thank the staff from 2XL Games, Inc., USA, for providing overall direction to the project through their vast experience in the PC and console game industry.

Chapter 1

Introduction

This chapter will look at the prime motivation for this project, its objectives and scope, my contributions and finally the outline of the thesis.

1.1 Motivation

“Necessity is the mother of invention” is a well known adage. All of humankind has progressed steadily due to the constant search for something better, which will improve and enhance our quality of life. This need is prevalent in the virtual world of gaming too, and it is evident from the fact that the quality of video games has improved tremendously over the past two decades.

A *video game* is a game that allows interaction with a user interface to generate visual feedback on a video device [1]. A key feature of video games, and hence their appeal, lies in the visual feedback to users. Apart from traditional factors such as *gameplay*, *story*, *input control* etc., which rely on human creativity to a large extent, visual appeal is something which was/is limited primarily by technological advances (or rather, *non-advances*). For instance, in the mid-80s, a best-seller was Super Mario Bros. [2], released by Nintendo in a 320 KB cartridge for the Nintendo Entertainment System (NES) platform [3]. A screenshot is shown in Figure 1.1.

Figure 1.1: Screenshot from “Super Mario Bros.”, 1985

Visual quality in video games has come a long way since then, as can be seen from hundreds of recent games. For example, screenshots from “Colin McRae DIRT” [4] and the soon-to-be-released AAA title “Baja: Edge of Control” [5] clearly illustrate this:

Figure 1.2: Screenshot from “Colin McRae DiRT”, 2007

Figure 1.3: Screenshot from “Baja: Edge of Control”, 2008

Hardware capability has gone up sharply over the past two decades following Moore’s law [6], but user demand is growing even faster. Unlike the older generation, people

today expect everything “bigger, better and cheaper”. It is apparent that this trend will continue to grow in the years to come, and this necessitates even more optimization and judicious use of resources.

One such area which offers game developers scope for optimization is the generation, modification and rendering of large worlds seamlessly and without drop in *frame rate*. However, achieving this is non-trivial, evident from the amount of research in terrains over the past two decades.

Specifically, terrains can be split into two representations; one focused on rendering and the other on generation. Existing algorithms mostly focus on terrain rendering and they use a *regular grid* representation. The other alternative, *triangulated irregular networks*, where the focus is on terrain generation (as opposed to rendering), was quite famous in the 70s and 80s, but its popularity waned due to recent advances in storage and processing capabilities.

In this project I attempt to revive this lost technique, and propose new methods to generate, optimize, modify and efficiently use such a terrain in modern day games to fulfill the initial motivation of rendering large worlds seamlessly.

1.2 Objective & Scope

A *terrain* can be defined as a detailed land surface with irregular undulations. The generation of a terrain is fundamental to many games today. The more realistic a terrain looks, the better the gameplay experience. This is especially true for the racing and simulation game genres, where users interact in vast, open worlds. This is affirmed by the

success of games like Treadmarks [7], Colin McRae Rally [8], Colin McRae DiRT [4] and Motorstorm [9], all widely acclaimed for their impressive looking terrains.

A terrain can be decomposed into two parts: (i) a triangular mesh, and (ii) one or more textures. In using such a terrain in a game, there are two main concerns: (i) the memory needed to store the terrain information for large worlds, and (ii) the time taken to load the terrain volume data into memory during a game's load-time. This project focuses on the triangular mesh part, and algorithms for its efficient generation and modification are developed.

Even within the scope of triangular meshes, there are two limitations: (i) memory usage after loading, and (ii) processing power to modify the loaded terrain (if level-of-detail techniques are used). All existing algorithms make a tradeoff between the two to deliver terrains of acceptable visual quality in games.

To address these issues, there has been a lot of research on terrain generation and rendering in the past 15 years, after the PC became powerful enough to do complex rendering tasks fast enough, at rates of 30 to 60 fps. Yet, there is not a simple, single method which is universally followed today. Most game studios develop their custom tool for terrain generation, which just "does the job", so to speak, without optimizations or improvements.

In this project, I have looked at various existing techniques, their limitation, scope and whether they can be widely deployed with ease. Addressing these issues, I have made variations to known methods, proposed techniques to significantly reduce the overall error and triangle count, and implemented tools for the same.

1.3 Contribution

I have introduced an efficient method for terrain generation, based on the Constrained Conforming Delaunay triangulation, using a heightmap, control map and several constraints, which act as control parameters. The concept of a control map is something new, allowing different regions in the terrain to be generated at varying detail.

Enhancements to the terrain mesh are done via user parameters specifying angle, area and height constraints. The nature of the Delaunay triangulation algorithm further guarantees the maximization of the minimum angle of any triangle, hence producing a well-graded, well-rounded, and more geometrically balanced terrain.

As a post-process to the Delaunay-based terrain, I have proposed a Constrained DDT algorithm to reduce terrain error. It makes use of my two new DDT cost functions and an optimization procedure based on Look-Ahead algorithm. Finally, I have also proposed a Terrain Simplification algorithm to reduce triangle count.

I have proposed three methods for modifying a terrain: Stenciling, Stitching and Track Overlaying, to add objects such as tracks, rivers and guardrails over the terrain. Each has its own characteristics, and Track Overlaying is specifically used to overlay flat-looking tracks over the terrain, without introducing uneven jumps or cracks.

1.4 Outline of Thesis

The central topic of this thesis is terrain generation from a *heightmap* (which is a grayscale image with white pixels indicating the highest points, black ones the lowest and

shades of gray the in-between heights) and its subsequent modification for usability in games. The background and foundation for this, therefore, lies in mesh generation and triangulation. The two major alternatives, and schools of thought for triangulating a set of points are *Delaunay Triangulation* (DT) and *Data Dependent Triangulation* (DDT).

DT and its constrained variant, *Constrained Delaunay Triangulation* (CDT), along with the related topic of convex hulls, are discussed in the section “Background” of Chapter 2. I have dealt with these fundamental topics in detail in an effort to make this thesis self-contained.

There is another variation of DT called *Constrained Conforming Delaunay Triangulation* (CCDT), which uses *Delaunay refinement*. DDT, edge swapping and cost functions play an important role in terrain optimization. These, along with existing terrain generation methods, are discussed in section “Related Work” of Chapter 3. This chapter also contains a brief introduction to the efficient Quad-Edge data structure, which forms the basis of my implementation.

On the road to efficient terrain generation, I have proposed several new algorithms which can be used in a general context, not just specific to terrains. These are catalogued in Chapter 4, “Proposed Algorithms”.

The core terrain generation and optimization procedures, with implementation details and results, are discussed in Chapter 5, “Terrain Generation”.

I have proposed three algorithms for terrain modification: (i) *stenciling*, (ii) *stitching*, and (iii) *track overlaying*. These algorithms, along with implementation and results, are looked at in Chapter 6, “Terrain Modification”.

I consider Chapters 4-6 to be the most important, as my core contribution towards terrain generation, optimization and modification, along with the underlying algorithms which forms their basis, are discussed.

Finally, in Chapter 7, the thesis concludes with a summary of the results and pointers to future work, which includes possible enhancements to the proposed algorithms.

Chapter 2

Background

In this chapter, I briefly look at the types of meshes, the general problem of arbitrary mesh generation and the desirable properties of a mesh generator, followed by some of the existing mesh generation methods.

In order to get the “complete picture” with regard to triangulation, I next discuss the construction of two and three dimensional convex hulls, followed by Delaunay triangulation, its relationship to convex hulls, and its constrained variation, the CDT.

I have purposely left the discussion of CCDT, Delaunay refinement, DDT and existing terrain generation methods for the next chapter “Related Work”, as they are more closely related to my proposed algorithms.

2.1 Types of Meshes

A mesh is a discretization of a geometric domain into simple shapes called elements. Meshes can (usually) be categorized as structured or unstructured. Figure 2.1 illustrates an example of each. Structured meshes exhibit a uniform topological structure that unstructured meshes lack. By definition, in a structured mesh, the *valence*, or the number

of neighboring node, of any non-boundary node is 6, whereas this is not always true for unstructured meshes. In simpler terms, for a structured mesh, the indices of neighbors of

Figure 2.1: Structured (left) and unstructured (right) meshes

any node can be calculated by simple addition, whereas an unstructured mesh necessitates the storage of each node's neighbors. From Figure 2.1, it can be seen that the structured mesh has the same topology as a square grid of triangles, although it is deformed enough that one might fail to notice its structure. Some advantages of structured meshes are simplicity, availability of code, and suitability for multi-grid and finite difference methods. On the other hand, unstructured meshes conform to the domain more easily and allow element sizes to vary more dramatically.

The generation of both types of meshes, given the approximate shape of the object and some input parameters such as minimum and maximum bounds, number of nodes etc., can be surprisingly difficult, each posing challenges of their own. Only unstructured meshes are considered, and furthermore, only simplicial meshes composed of triangles. Meshes with quadrilateral, pentagonal, or other non-simplicial elements are passed over, although they constitute an interesting field of study in their own right. The first reason for this is that my primary interest is in terrain modeling, and a triangular mesh is usually more than sufficient. The second reason is that I need a well-graded terrain with variations in density of triangulation, thereby reducing the number of triangles used to

represent flat or near-planar areas. This would be impossible if a structured mesh were used.

2.2 Desirable properties of Mesh Generators

A useful mesh satisfies constraints that seem almost contradictory. A mesh must conform to the object being modeled, and must ideally meet the constraints on both the size and shapes of its elements.

The first goal of mesh generation is to offer as much control as possible over the sizes of the elements in the mesh. Ideally, this should include the ability to grade from small to large elements over a relatively short distance. Small, densely packed elements offer more accuracy than larger, sparsely packed elements; but the computation time is proportional to the number of elements generated. If elements of uniform size are used throughout the mesh, as in a structured mesh, a small enough size must be chosen to guarantee sufficient accuracy in the most demanding portions of the mesh, thereby possibly incurring excessive computational overhead. To avoid this pitfall, the mesh generator should offer rapid gradation from small to large sizes. With regard to terrain meshes, small or densely packed elements are needed in mountainous regions with sharp features, but not in relatively flat regions. In order to provide a smooth gradation between these two regions, an unstructured mesh is most suitable.

The second goal of mesh generation, and the most difficult, is that the elements should be relatively “round” in shape, because elements with large or small angles can degrade the quality of the mesh. Elements with large angles cause a large *discretization error*. In finite element methods, Babuska and Aziz [10] show that if mesh angles approach 180, convergence to the exact solution may fail to occur. A lower bound on the smallest angle

of a triangulation implicitly bounds the largest angle. If no angle is smaller than θ , no angle is larger than $180 - 2\theta$. Hence, many mesh generation algorithms, including the constrained conforming Delaunay triangulation algorithm I use, take the approach of attempting to bound the smallest angle.

2.3 Mesh Generation Methods

Detailed surveys of mesh generation algorithms are discussed by Thompson and Weatherill [11], and Bern and Eppstein [12]. The most popular approaches to mesh generation can be divided into three classes: Delaunay triangulation methods, advancing front methods and methods based on grids, quadrees or octrees. I focus on algorithms which output Delaunay triangulations.

It is not easy to trace who first used Delaunay triangulations for mesh generation, and also difficult to tell where the suggestion arose to use the triangulation to guide vertex creation. These ideas have been intensively studied in the engineering community since the mid-1980s, and began to attract interest from the computational geometry community in the early 1990s.

Many of the earliest papers suggest performing vertex placement as a separate step, typically using structured grid techniques, prior to Delaunay triangulation. For instance, Cavendish, Field and Frey [13] generate grids of vertices from cross-sections of a three-dimensional object, then form their Delaunay tetrahedralization. The idea of using the triangulation itself as a guide for vertex placement followed quickly; for instance, Frey [14] removes poor quality elements from a triangulation by inserting new vertices at their *circumcenters* – the center of their circumcircles – while maintaining the Delaunay

property of the triangulation. I will show in Chapter 3 how this idea has born fruit into fantastic algorithms for mesh refinement.

2.4 Convex Hull

Given a set $S = \{p_0, p_1, \dots, p_{n-1}\}$ of n points, the *convex combination* of the points in S is a point defined by

$$p = \sum_{i=0}^{n-1} \alpha_i p_i$$

where $\sum_{i=0}^{n-1} \alpha_i = 1 \quad \forall \alpha_i \geq 0$.

Then, the *convex hull* of S , $CH(S)$, is the set of all convex combinations of points in S . This definition is valid for arbitrary d -dimensional space, but I am interested only in 2D and 3D convex hulls.

Figure 2.2: A set of points (left) and its convex hull (right)

A *polygon* is defined as a cyclic sequence $P = (p_0, p_1, \dots, p_{n-1}, p_n = p_0)$ of n points in a plane, such that each consecutive pair determines an edge $\overline{p_i p_{i+1}} \quad \forall i \in (0, n-1)$. A polygon is *simple* if no two of its non-consecutive edges intersect. A simple polygon is *convex* iff for any two points $p, q \in P$, the line segment $\overline{pq}$ lies entirely within the interior of P .

In simpler terms, the convex hull of a 2D point set is the smallest convex polygon which includes all the points, as shown in Figure 2.2. Similarly, the convex hull of a 3D point

set is the smallest convex polyhedron which includes all the points, where a *convex polyhedron* is the 3-dimensional counterpart of a convex polygon.

2.4.1 Constructing 2D convex hulls

Computing the convex hull is a fundamental and classic problem in *computational geometry* [15] that has many interesting applications. Many seemingly unrelated problems in computational geometry can be solved using a convex hull algorithm as a sub-algorithm. I also use it for my terrain generation and modification, though it is not apparent right now. Thus, developing good convex hull algorithms is essential to the design of efficient geometric algorithms for other problems.

Several convex hull algorithms exist. They include *Graham's scan* algorithm [16], Eddy's *Quickhull* algorithm [17], Preparata's *Divide-and-Conquer* algorithm [18], Andrew's *Monotone Chain* algorithm [19] and Kallay's *Incremental* algorithm [20].

Among these, the best algorithms like Graham's scan, divide-and-conquer and monotone chain have a complexity of $O(n \log n)$. Graham's scan can be ruled out because it involves sorting points by angle computation, which in itself is CPU intensive. Divide-and-conquer is widely used, but its implementation is non-trivial as the merging part has to be done carefully. The basic incremental algorithm is $O(n^2)$, but intelligent data structures can be used to make it $O(n \log n)$. I will show how this can be done in Section 2.4.4. Then there is Kirkpatrick's "ultimate" (so called by its author!) *Marriage-before-Conquest* algorithm [21], with a complexity of just $O(n \log h)$, h being the number of vertices in the output hull ($h \leq n$).

An algorithm A is said to be *space-optimal* if the total memory space for storing A , the input data of size n , and the temporary space for execution of A is no more than $n + c$, where c is a constant. Even though marriage-before-conquest is *time-optimal*, it is far from space-optimal as there is large storage overhead, and the implementation is also complicated. That leaves me with the monotone chain algorithm, which I have implemented.

2.4.2 Andrew's Monotone Chain algorithm

Andrew's algorithm computes the upper and lower hulls of a monotone chain of points, and so I refer to it as the Monotone Chain algorithm.

First sort the point set $S = \{P_0, P_1, \dots, P_{n-1}\}$ of n points by increasing x and then y coordinate values. Let the minimum and maximum x -coordinates be x_{min} and x_{max} . Then, $P_0.x = x_{min}$, but there may be other points with this minimum x -coordinate. Let P_{-} be the point in S with $P.x = x_{min}$ first and then min y among all such points. Let P_{+} be the point with $P.x = x_{min}$ first and then max y second. Note that $P_{-} = P_{+}$ when there is a unique x -minimum point. Similarly define P_{+-} and P_{++} as the points with $P.x = x_{max}$ first, and then y min or max second. Again, $P_{+-} = P_{++}$ when there is a unique x -maximum point. Next, join the lower two points, P_{-} and P_{+-} to define a lower line L_{min} . Also join the upper two points, P_{+} and P_{++} to define an upper line L_{max} . These points and lines are shown in Figure 2.3.

Figure 2.3: Monotone chain algorithm illustration

The algorithm now proceeds to construct a lower convex vertex chain Ω_{min} below L_{min} and joining the two lower points P_{--} and P_{++} ; and also an upper convex vertex chain Ω_{max} above L_{max} and joining the two upper points P_{++} and P_{--} . Then the convex hull W of S is constructed by joining Ω_{min} and Ω_{max} together.

For the lower chain, start with P_{--} on the stack. Then process the points of S in sequence. Only consider points strictly below the lower line L_{min} . Suppose that at any stage, the points on the stack are the convex hull of points below L_{min} that have already been processed. Now consider the next point P_k that is below L_{min} . If the stack contains only the one point P_{--} then put P_k onto the stack and proceed to the next stage. Otherwise, determine whether P_k is strictly left of the line between the top two points on the stack. If it is, put P_k onto the stack and proceed. If it is not, pop the top point off the stack, and test P_k against the stack again. Continue until P_k gets pushed onto the stack. After this stage, the stack again contains the vertices of the lower hull for the points already considered. After all points have been processed, push P_{++} onto the stack to complete the lower convex chain.

The upper convex chain Ω_{max} is constructed in an analogous manner, but processes S in decreasing order $\{P_{n-1}, P_{n-2}, \dots, P_0\}$, starts at P_{++} , and considers only points above L_{max} .

Heapsort sorts the elements *in-place* in $O(n \log n)$ time. It can be implemented using constant bounded working memory space. It is thus *time-space* optimal. *Quicksort* is a generic name for a family of algorithms based on partition-exchange sort, which works well for uniformly distributed input. It is widely used though it is not time-optimal.

Thus, if the sorting by x and then y is implemented with heapsort, it will take $O(n \log n)$ time. The actual calculation of the lower and upper hulls each take $O(n)$ time and the merging can be done in $O(1)$ time. Therefore, the overall complexity of Andrew's algorithm is $\text{Max}(O(n \log n), O(n), O(1)) = O(n \log n)$.

2.4.3 Constructing 3D convex hulls

Before thinking of algorithms for constructing the 3D convex hull, it is wise to check if any of the 2D algorithms can be extended, since some of them are already time-optimal.

I start with the already implemented monotone chain algorithm. In order to extend it to 3D, I need to consider 9 points $P_{---}, P_{-++}, P_{-+-}$ etc., and partition 3-space into three areas, with L_{min} and L_{max} being planes rather than lines. It is not easy to figure out which three points to use for constructing each plane, and the merging of the lower and upper hulls is not as trivial as in 2D. So it is safe to skip the monotone chain algorithm.

Out of the other four algorithms mentioned in Section 2.4.1, only quickhull, divide-and-conquer and incremental algorithms can be extended easily. Of these, the easiest to implement is the incremental algorithm, which has a basic complexity of $O(n^2)$, but it can be optimized to $O(n \log n)$ with randomized incremental construction using Clarkson-Shor's *conflict graph* [22].

2.4.4 3D Incremental algorithm

The basic idea of the *Incremental algorithm* is as follows. First take a subset of the input so that the problem is easily solved. Then, add the remaining elements one by one while maintaining the solution at each step. I apply this general idea for convex hull construction as follows.

Let $P = \{p_1, p_2, \dots, p_n\}$ represent the set of n points in 3-space with convex hull $CH(P)$.

Let P_r represent the set of r points $\{p_1, p_2, \dots, p_r\}$ for $r \in (1, n)$. Note that $P_n = P$. A *facet* can be any polygonal face, but here it denotes a triangular face.

```

1:   Choose three non-collinear points  $p_1, p_2, p_3$ 
2:   Choose fourth point  $p_4$ , not coplanar to  $p_1, p_2, p_3$ 
3:   Form tetrahedron  $p_1 p_2 p_3 p_4$ , add its four facets to initialize  $CH(P_4)$ 
4:   For  $r = 5$  to  $n$ 
5:       If  $p_r$  is strictly inside  $CH(P_{r-1})$ 
6:           Ignore  $p_r$ 
7:            $CH(P_r) = CH(P_{r-1})$ 
8:       Else If  $p_r$  lies on the surface of  $CH(P_{r-1})$ 
9:           Find facet(s) (1 or 2)  $F_{old}$ , that  $p_r$  lies on
10:          Add  $p_r$ , re-triangulate  $F_{old}$  to get  $F_{new}$  ( $1 \rightarrow 3; 2 \rightarrow 4$ )
11:           $CH(P_r) = CH(P_{r-1}) - F_{old} + F_{new}$ 
12:       Else If  $p_r$  is outside  $CH(P_{r-1})$ 
13:          Find set of visible facets  $F_v$  from  $p_r$  on  $CH(P_{r-1})$ 

```


14:	Determine horizon of p_r on $CH(P_{r-1})$, based on F_v
15:	Connect each horizon edge to p_r to create set of new facets F_n
16:	$CH(P_r) = CH(P_{r-1}) - F_v + F_n$
17:	End If
18:	End For
19:	Get convex hull $CH(P) = CH(P_n)$

Algorithm 2.1: 3D incremental convex hull

To understand the algorithm better, some illustrations are shown below:

Figure 2.4: Visibility test

Figure 2.5: Horizon as seen from p_r

Figure 2.6: Incremental construction: before (left) and after (right) adding p_r

Steps 5, 8, 12 and 13 of Algorithm 2.1 can be understood by Figure 2.4. I can easily check for orientation of p_r with respect to any facet of $CH(P_{r-1})$. Since all the facets are oriented counter clockwise (CCW) when they are added, p_r is inside the hull if no facet is visible from it. If it lies on the convex hull, it can lie on either one or two facets only. This is because the mesh is triangular and the assumption that $p_i \neq p_j \forall p_i, p_j \in P$.

Figure 2.7: Splitting facets when p_r lies on the hull

When this happens, add p_r to the hull and split the old facets (one facet becomes three, two facets become four), as shown in Figure 2.7.

If p_r lies outside the hull, as shown in Figure 2.6 (left), compute the horizon from p_r as shown in Figure 2.5. Then, p_r is connected to the horizon edges and the new hull is computed, as illustrated in Figure 2.6 (right).

To perform the visibility test, the naïve way would be to check orientation of each existing facet with p_r . This gives rise to a complexity of $O(n^2)$. In order to reduce it to an optimal $O(n \log n)$, I use a conflict graph, which is briefly explained below.

For each facet f maintain $P_{\text{conflict}}(f) \subseteq \{p_{r+1}, \dots, p_n\}$ containing points to be inserted that can see f ; and for each point p_t , where $t > r$, maintain $F_{\text{conflict}}(p_t)$ containing facets of $CH(P_{r-1})$ visible from p_t . This conflict graph G is actually a *bipartite graph*, shown in Figure 2.8.

Figure 2.8: Bipartite graph showing conflicts

- 1: Initialize G with $CH(p_4)$ in linear time
- 2: Walk through $\{p_5, \dots, p_n\}$ to determine which facet each point can see
- 3: **For** $r = 5$ to n
- 4: Discard visible facets from p_r by removing neighbors of p_r in G
- 5: Remove p_r from G
- 6: Determine new conflicts
- 7: **End For**

Algorithm 2.2: Optimized visibility testing using conflict graph

Algorithm 2.2 can easily be incorporated into Algorithm 2.1 since both of them run through points p_5 to p_n after some initialization. The heart of the algorithm is Step 6, which is elaborated below.

Figure 2.9: Determining new conflicts for new face f

In Figure 2.9, if p_t can see new facet f , for $t > r$, then it can see edge e of f . Since e lies on the horizon of p_r , it implies that e was already visible from p_t in $CH(p_{r-1})$.

Further, if p_i can see e , this means it saw either f_1 or f_2 in $CH(P_{r-1})$, which implies p_i was in $P_{\text{conflict}}(f_1)$ or $P_{\text{conflict}}(f_2)$ in $CH(P_{r-1})$. Thus, one can conclude that the new conflict list of f can be found by testing the points in the conflict lists of f_1 and f_2 .

With this method for determining new conflicts, the number of facets needed to check for each new point p_r is minimized, thus reducing the overall average complexity from $O(n^2)$ to $O(n \log n)$.

I have not explained what to do when Step 2 of Algorithm 2.1 fails, i.e. when all input points are coplanar. For this purpose, I use a simple triangulation technique (the proposed “intuitive” algorithm) to compute the planar convex hull, which is nothing but a triangulation of the 3D coplanar points, where each facet is added twice, one for each orientation. This is looked at in Chapter 4.

2.5 Delaunay Triangulation

In two dimensions, a *triangulation* of a set V of vertices is a set T of triangles whose vertices collectively form V , whose interiors do not intersect each other, and whose union completely fills the convex hull of V .

The Delaunay triangulation a set of vertices V , $DT(V)$, is a graph defined as follows. Any circle in the plane of the vertices is said to be *empty* if it contains no vertex of V in its interior (vertices are permitted *on* the circle). Let u and v be any two vertices of V . The edge uv is in $DT(V)$ iff there exists an empty circle that passes through u and v . An edge satisfying this property is said to be *Delaunay*. It was introduced by Boris Delaunay [23] in 1934. Figure 2.10 illustrates a sample Delaunay triangulation of a point set.

Figure 2.10: A Delaunay triangulation

The Delaunay triangulation of a vertex set is clearly unique, because by the definition above, there is an unambiguous test for the presence or absence of an edge in the triangulation. Every edge of the convex hull of a vertex set is Delaunay. Figure 2.11 illustrates the reason why. For any convex hull edge e , it is always possible to find an empty circle that contains e by starting with the smallest containing circle of e and “growing” it away from the triangulation.

Figure 2.11: Each edge on the convex hull is Delaunay

Figure 2.12: Empty circumcircle property of the Delaunay triangulation

The *circumcircle* of a triangle is the unique circle that passes through all three of its vertices. A triangle is said to be Delaunay iff its circumcircle is empty. This defining characteristic of Delaunay triangles, illustrated in Figure 2.12, is called the empty

circumcircle property. Due to this property, the following important theorem can be stated:

Theorem 1 *Among all triangulations of a vertex set, the Delaunay triangulation maximizes the minimum angle in the triangulation, minimizes the largest circumcircle, and minimizes the largest min-containment circle, where the min-containment circle of a triangle is the smallest circle that contains it.*

The property of max-min angle optimality was first noted by Lawson [24], and helps to account for the popularity of Delaunay triangulations in mesh generation. Unfortunately, neither this property nor the min-max circumcircle property generalizes to Delaunay triangulations in dimensions higher than two. The property of minimizing the largest min-containment circle was first noted by D'Azevedo and Simpson [25].

Theorem 1 is the primary reason for choosing DT to generate the terrain, since it automatically satisfies the properties of a good mesh generator. For an elegant proof, refer to Shewchuk [26].

2.5.1 Constructing Delaunay triangulations

Three types of algorithms are commonly used for constructing Delaunay triangulations. The simplest is the *Incremental* algorithm, which has the added advantage of being generalized to higher dimensions. There are faster $O(n \log n)$ algorithms based on *Divide-and-Conquer* and *Sweep* techniques. Refer to Su and Drysdale [27, 28] for a comprehensive discussion of these and other 2D Delaunay triangulation algorithms.

An edge is said to be *locally Delaunay* if there exists a circle through its end points which does not include its immediate neighboring vertices, as shown in Figure 2.13 (left). *Edge swap* is a basic operation that can be used to convert a non-locally Delaunay edge into a locally Delaunay edge, and vice versa (Figure 2.13). It was first introduced by Lawson [24] to perform incremental refinement.

Figure 2.13: After edge swap operation (right) on locally Delaunay edge e (left)

Incremental construction takes place by the *swap algorithm*, which maintains the Delaunay property for all edges after each vertex insertion. It was seen earlier that for any Delaunay triangulation, all edges and triangles are Delaunay. The swap algorithm basically ensures that each edge is *locally Delaunay* by performing the necessary edge swaps, and making use of the following theorems:

Theorem 2 *If T is a triangulation and if all the triangles of T are Delaunay, then all the edges of T are Delaunay, and vice versa.*

Theorem 3 *If T is a triangulation whose edges are all locally Delaunay, then every edge of T is also globally Delaunay.*

For simple proofs for the above theorems, refer to Shewchuk [26]. Note that edge swap is not possible when the basic quadrilateral is concave instead of convex. The swap algorithm only performs legitimate edge swaps, and only on non-locally Delaunay edges.

It is guaranteed to terminate after $O(n^2)$ edge swaps, yielding a triangulation whose edges are all Delaunay [24].

The complexity of the basic swap algorithm is $O(n^2)$, where the dominant cost comes from the $O(n)$ time for point location, for searching through all the existing triangles at each step. However, using Clarkson-Shor's *conflict graph* [22] (Section 2.4.4), this can be brought down to $O(\log n)$, thus reducing the overall complexity to $O(n \log n)$.

The first $O(n \log n)$ algorithm for 2D Delaunay triangulation was not an incremental algorithm, but a divide-and-conquer algorithm. Shamos and Hoey [29] developed the algorithm for the construction of a *Voronoi diagram*, which may be easily *dualized* to form a Delaunay triangulation. Implementation-wise, this is an unnecessarily difficult procedure, because forming a Delaunay triangulation directly is much easier, and is in fact the easiest way to construct a Voronoi diagram. Lee and Schachter [30] were the first to publish a divide-and-conquer algorithm that follows this easier path. The algorithm is nonetheless intricate, and Guibas and Stolfi [31] provide an important aid to programmers by filling out many tricky implementation details. Dwyer [32] offers an interesting modification to divide-and-conquer Delaunay triangulation that achieves better asymptotic performance on some vertex sets, and offers improved speed in practice as well. There is also an algorithm for constrained Delaunay triangulations due to Chew [33]. Another well-known 2D Delaunay triangulation algorithm is Fortune's sweepline algorithm [34].

However, there is another beautiful $O(n \log n)$ algorithm, which eliminates all the complexity involved in performing point-inside-triangle and point-inside-circumcircle tests. This is what I have implemented, as shown in the next Section.

2.5.2 Relationship between convex hulls and Delaunay triangulations

In 1986, Edelsbrunner and Seidel [35] discovered a beautiful relationship between Delaunay triangulations and convex hulls in one higher dimension. Their insight was based on Brown [36], who was the first to establish the connection to hulls. Their relation is stated below:

Let $P = \{p_1, p_2, \dots, p_n\}$ represent a set of n points on the plane $z = 0$. Projecting them onto the unit elliptic paraboloid $z = x^2 + y^2$ yields a point set P' such that for each $p_i = (x_i, y_i) \in P$, there exists a $p'_i \in P'$, which is computed as follows:

$$p'_i = (x_i, y_i, x_i^2 + y_i^2) \quad \forall i \in (1, n)$$

By calculating the convex hull of the point set P' and projecting the facets with downward facing normals onto the $z = 0$ plane, one gets the Delaunay triangulation of the original point set P ! This stunning conclusion can be visualized below:

Figure 2.14: Lower 3D convex hull projected onto the xy plane

What Brown claims is that the projected lower 3D convex hull (Figure 2.14) is the Delaunay triangulation of the initial point set. In fact, this is true for any dimension viz. on projecting an r -dimensional convex hull onto the corresponding lower $(r-1)$ -

dimensional space, I would get the Delaunay composition (one cannot call it triangulation anymore!) of the original points in $(r - 1)$ -dimension.

This property is used for Delaunay triangulating a 2D point set as it is elegant, each step isn't computationally expensive, and it can be optimized to $O(n \log n)$ using Clarkson-Shor's conflict graph [22]. This is the reason for discussing 3D convex hull construction and the conflict graph in great detail in Section 2.1.4. The Delaunay triangulation algorithm of a point set, as I have implemented, is formally stated below:

Let P and P' represent the original and upwards-projected point sets respectively; and let $DT(P)$ and $CH(P')$ represent the Delaunay triangulation of P and convex hull of P' respectively.

```

1:   Initialize  $P'$ ,  $DT(P)$  to empty
2:   For each  $p_i = (x_i, y_i) \in P$ 
3:       Add  $p_i' = (x_i, y_i, x_i^2 + y_i^2)$  to  $P'$ 
4:   End For
5:   Compute  $CH(P')$  using the optimized incremental algorithm (Algorithm 2.1)
6:   For each facet  $f \in CH(P')$ 
7:       If  $f$ 's normal faces downwards
8:           For each edge  $e$  of  $f$ 
9:               Set  $z$  value for each vertex in  $e$  to 0
10:              Construct new edge  $e' = e$ 
11:              Add  $e'$  to  $DT(P)$ 
12:           End For

```

13:	End If
14:	End For
15:	$DT(P)$ is the Delaunay triangulation of point set P

Algorithm 2.4: Delaunay triangulation algorithm

2.6 Constrained Delaunay Triangulation

Even though Delaunay triangulation maximizes the minimum angle, the problem of mesh generation is not solved. This is because the usual input for mesh generation is not merely a set of vertices, but a *Planar Straight Line Graph* (PSLG).

A PSLG is a set of vertices and segments that satisfies two constraints. First, for each segment contained in a PSLG, the PSLG must also contain the two vertices that serve as end points for that segment. Second, segments are permitted to intersect only at their endpoints. A set of segments that does not satisfy this condition can be converted into a set of segments that does. I simply perform a segment intersection algorithm [22, 37], add the resulting points of intersection and then divide each original segment into smaller segments.

The *constrained Delaunay triangulation* (CDT) of a PSLG X is similar to the Delaunay triangulation, but every input segment appears as an edge in the CDT. An edge or triangle is said to be *constrained Delaunay* if it satisfies the following two conditions. First, its vertices are visible to each other. Here, visibility is deemed to be obstructed if a segment of X lies between two vertices. Second, there exists a circle that passes through the vertices of the edge or triangle in question, and the circle contains no vertices of X that are visible from the interior of the edge or triangle.

Segments of X are also considered constrained Delaunay. Figure 2.15 shows examples of a constrained Delaunay edge e and constrained Delaunay triangle t .

Figure 2.15: Bold segments; edge e and triangle t are both constrained Delaunay

Input segments are shown in bold. Although there is no empty circle that contains e , the depicted containing circle of e contains no vertices that are visible from the interior of e . There are two vertices inside the circle, but both are hidden behind segments. Hence, e is constrained Delaunay. Similarly, the circumcircle of t contains two vertices, but both are hidden from the interior of t by segments, so t is also constrained Delaunay.

Figure 2.16: CDT construction by segment insertion and retriangulation

To construct the constrained Delaunay triangulation, I start with the Delaunay triangulation of just the vertices of the PSLG (without the segments). Next, each input segment is examined in turn to see if it is missing from the DT. Each missing segment is forced into the triangulation by deleting all the edges it crosses, inserting the new segment, and retriangulating the two resulting polygons (one on each side of the segment), as illustrated in Figure 2.16.

Since the triangulation is convex, any polygon resulting from edge removal has to be simple. So all I need to do is to triangulate the simple polygon by adding diagonals. For a

proof that every simple polygon has a diagonal, refer to Meister [38]. For a proof that every polygon can be triangulated, refer to Bern and Eppstein [12].

If these three steps of edge removal, segment insertion and polygon triangulation are done for each input segment of the PSLG, the final result is the constrained Delaunay triangulation of the PSLG.

Though the idea is simple, there are many ways to implement it, as shown by Chew [33] and Anglada [39]. Shewchuk [40] suggests an $O(n^2)$ gift wrapping algorithm for retriangulating the holes resulting from edge removals and segment insertion, but I prefer to do polygon triangulation for each of the upper and lower polygons using the *ear-cutting algorithm* [41]. The CDT algorithm is the basis for the “stitching” and “track overlaying” algorithms, seen in Chapter 6.

Chapter 3

Related Work

In this chapter, I discuss existing work directly related to my proposed methods and implementation. They fall under two categories: (i) related to triangulation (ii) related to terrain generation. Although the latter is closely related to the former, they are essentially two separate ideas. Hence I have treated them in their own right in this chapter.

First I look at CCDT and its variants using Delaunay refinement, followed by DDT refinement and exiting cost functions. Next I look at the existing terrain generation algorithms, focusing more on triangulated irregular networks rather than regular grids, and the rationale for the same.

Lastly, a very brief introduction to the Quad-Edge data structure is given, without which I would not have been able to implement my algorithms efficiently.

3.1 Constrained Conforming Delaunay Triangulation

The previous chapter showed that the constrained Delaunay triangulation of a PSLG solves the insufficiency of a DT in mesh generation. However, CDT also has limitations, as all the input vertices and/or segment are rarely known prior to triangulation. The real

problem of meshing is to decide where to place the vertices to ensure that the mesh has elements of good quality and proper sizes, as discussed in Section 2.2.

A *conforming Delaunay triangulation* of a PSLG is a true Delaunay triangulation in which each PSLG segment may have been subdivided into several edges by the insertion of additional vertices, called *Steiner points* [42]. Steiner points are necessary to allow the segments to exist in the mesh while maintaining the Delaunay property. They can also be inserted to meet other constraints, such as minimum angle, maximum area etc.

A *constrained conforming Delaunay triangulation* (CCDT) of a PSLG is a constrained Delaunay triangulation that includes Steiner points. As opposed to a conforming Delaunay triangulation, Steiner points are inserted solely to meet constraints on quality and size, since all the triangles of the CDT are already constrained Delaunay. It usually takes fewer vertices to make a good-quality CCDT than a good quality conforming Delaunay triangulation, because the triangles need not be Delaunay, they only need to be constrained Delaunay.

The construction of a CCDT takes place through *Delaunay refinement*, which is the refinement of a Delaunay or constrained Delaunay triangulated mesh by careful insertion of vertices until all the constraints on quality and size are met. The central question in refinement is “where the new vertex should be inserted?”. The most reasonable answer is “as far away as possible from other vertices”. If a new vertex is inserted too close to another vertex, it is likely to create triangles with thin angles, which goes against the property of a good mesh generator (Section 2.2).

Since a Delaunay triangle has no other vertices in its circumcircle, DT is an ideal search structure for finding points that are far from other vertices. Since the circumcenter of a triangle is the point furthest away from its vertices, most algorithms involve vertex insertion at the circumcenter. I briefly discuss three main algorithms due to Chew [43, 44] and Ruppert [45], which make use of the above principle for Delaunay refinement, before looking at the my proposed idea.

3.1.1 Chew's first algorithm

Paul Chew's first Delaunay refinement algorithm [43] produces triangulations of uniform density. The central idea is insertion of vertex at the circumcenter (henceforth called *splitting*) of a "poor" triangle. The notion of a "poor" triangle is that which has circumradius-to-shortest edge ratio (*quality ratio* Q_R) larger than some bound B . In other words, at least one angle of the "poor" triangle is smaller than a *minimum angle* bound of $\arcsin\left(\frac{1}{2B}\right)$. These poor triangles are termed *skinny* triangles.

Figure 3.1: Vertex insertion at the circumcenter of a skinny triangle

Figure 3.1 illustrates insertion of vertex v at the circumcenter of skinny triangle t . Because v is the circumcenter of t , and there were no vertices inside the circumcircle of t before v was inserted, no new edge can be shorter than the circumradius of t . Because t has a circumradius-to-shortest edge ratio larger than B , every new edge has length at least B times that of the shortest edge of t , therefore reducing Q_R for the resulting triangles after retriangulation.

Chew's algorithm employs a bound $B=1$, thus ensuring the output mesh contains no angle smaller than 30° . In other words, if Chew's algorithm is applied to a triangulation whose shortest edge has length $h_{\min}$, then it splits any triangle whose circumradius is greater than $h_{\min}$, hence creating a uniform mesh, as shown in Figure 3.2.

Figure 3.2: A mesh generated by Chew's first algorithm

The input to Chew's first algorithm is a *segment bounded* PSLG, meaning that the region to be refined is always segment bounded. Though the algorithm is simple, it does not handle boundary segments very well, i.e. when the circumcenter lies outside a boundary segment, as shown in Figure 3.3.

Figure 3.3: Circumcenter of a skinny triangle outside the triangulation

The algorithm simply pre-splits the boundary segments uniformly based on $h_{\min}$, thus always giving rise to a uniform mesh. Even if I wanted triangles with variations in sizes, it wouldn't be possible due to this restriction. So I cannot use it directly for terrain generation as it requires a well graded mesh.

3.1.2 Ruppert's algorithm

Jim Ruppert's algorithm for two-dimensional quality mesh generation [45] is the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice. It extends Chew's first algorithm by allowing the density of triangles to vary quickly over short distances, as illustrated in Figure 3.4.

Figure 3.4: A mesh generated by Ruppert's algorithm

Like Chew's first algorithm, Ruppert's algorithm takes a segment bounded PSLG as input. Unlike Chew's algorithm, it may start with either a constrained or unconstrained Delaunay triangulation. Input segments that are missing from the triangulation will be inserted as a natural process of the algorithm, thus eliminating the need for the CDT procedure described in Section 2.6.

Vertex insertion is governed by two rules:

- The diametral circle of a subsegment is the (unique) smallest circle that contains the subsegment. A subsegment is said to be encroached if a vertex lies strictly inside its diametral circle, or if the subsegment does not appear in the triangulation (the latter case generally implies the former, the only exceptions being degenerate examples where several vertices lie precisely on the diametral circle). Any encroached subsegment that arises is immediately bisected by inserting a vertex at its midpoint, as illustrated in Figure 3.5. The two

subsegments that result have smaller diametral circles, and may or may not be encroached themselves.

Figure 3.5: Splitting encroached segments recursively

- Each skinny triangle (having Q_R larger than some bound B) is normally split by inserting a vertex at its circumcenter. The Delaunay property guarantees that the triangle is eliminated, as shown in Figure 2.21. However, if the new vertex would encroach upon any subsegment, then it is not inserted; instead, all the subsegments it would encroach upon are split.

Figure 3.6: Splitting skinny triangle and maintaining Delaunay property

Missing segments are forced into the mesh by the same recursive splitting procedure used for encroached subsegments. From the two rules, it is seen that encroached subsegments are given priority over skinny triangles. After all encroached subsegments have been recursively bisected, and no subsegments are encroached, all edges (including subsegments) of the triangulation are Delaunay. A mesh produced by Ruppert's algorithm is truly Delaunay, and not merely constrained Delaunay.

Ruppert's algorithm is better than Chew's first algorithm in handling boundary segments, when the circumcenter of a skinny triangle lies outside the triangulation. In fact,

Ruppert's algorithm doesn't even need to handle this case. If the circumcenter lies outside the triangulation, it will simply lead to subsegment encroachment rather than circumcenter insertion, since they are given priority. The Delaunay property assures that after all subsegment encroachments have been fixed, the circumcenters of all triangles will lie within the triangulation.

3.1.3 Chew's second algorithm

Paul Chew's second Delaunay refinement algorithm [44] is a variant of Ruppert's algorithm that not only adds, but also removes vertices when a subsegment is encroached. The algorithm is briefly explained below:

Chew's second algorithm begins with the constrained Delaunay triangulation of a segment-bounded PSLG, and uses Delaunay refinement with locked subsegments and a quality bound of $B = 1$, but there is no idea of encroached diametral circles. The basic idea of splitting a skinny triangle by inserting a vertex at its circumcenter is retained from his first algorithm. However, it may arise that a skinny triangle t cannot be split because t and its circumcenter c lie on opposite sides of a subsegment s (possibly with c outside the triangulation).

Figure 3.7: Encroached segment split by Chew's second algorithm

Because s is locked, inserting a vertex at c will not remove t from the mesh. Instead, c is rejected for insertion, and all free vertices (but not input vertices or vertices that lie on

segments) that lie in the interior of the diametral circle of s and are visible from the midpoint of s are deleted. Then, a new vertex is inserted at the midpoint of s . The Delaunay property is maintained throughout all deletions and insertions, except that locked subsegments are not swapped. Figure 3.7 illustrates a subsegment split in Chew's algorithm. If several subsegments lie between t and c , only the subsegment visible from the interior of t is split.

3.2 Data Dependent Triangulation

Delaunay triangulation is purely done in 2D. It is a triangulation that maximizes the minimum angle and minimizes the maximum circumcircle of all possible triangulations of the point set (Theorem 1 – Section 2.5). This helps in removing thin sliver triangles, considered a desirable property by mesh generators and engineers.

However, this approach was shown to be not always good by Dyn et al. in their classical paper on Data Dependent Triangulation [46]. DDT, in contrast to Delaunay based methods, uses the height of points in addition to their x and y coordinates to achieve lower approximation errors, but with more slivers.

3.2.1 DDT cost functions

Smoothness of a mesh is quantified by means of a cost c_i associated with each interior edge e_i (boundary edges have cost 0). For a total of n interior edges, Dyn et al. [46] suggested 3 ways of computing the total cost C of the triangulation:

1. L_1 norm:
$$C = \sum_{i=1}^n |c_i|$$
2. L_2 norm:
$$C = \sum_{i=1}^n (c_i)^2$$

3. L_∞ norm: For any two triangulations T_1 and T_2 , define an ordering for their respective edge cost vectors C_1 and C_2 , and chose the one with lexicographically lower c_i .

They also proposed numerous cost functions, and among these, 4 are classified as nearly C^1 . Since they appear best for terrain generation, where C^1 continuity is required for smooth shading, I will go through them with reference to Figure 3.8.

Figure 3.8: Initial triangulation for computing cost

Some terminology: cost of edge $v_2 - v_4$ is computed, which is shared by triangles T_1 and T_3 with normals n_1 and n_3 respectively. The equation of the planes containing these triangles is:

$$z_i = a_i \cdot x + b_i \cdot y + c_i \quad i \in (1,3)$$

The coordinates of vertex v_k are (x_k, y_k, z_k) , for $k \in (1, \dots, 4)$

1. Angle Between Normals (ABN):

This is the cosine of the 3D angle between normals n_1 and n_3 .

$$\text{cost} = n_1 \cdot n_2$$

2. Jump in Normal Derivatives (JND):

$$\text{cost} = |n_x(a_1 - a_3) + n_y(b_1 - b_3)|$$

where (n_x, n_y) is a unit vector in the xy plane orthogonal to the projection of $v_2 - v_4$, onto the xy plane.

3. Deviations from Linear Polynomials (DLP):

This is given by magnitude of h , $\|h\|$, where h is a vector defined as

$$\text{cost} = \|h\|$$

$$\text{where } h = \left(|a_1 x_3 + b_1 y_3 + c_1 - z_3|, |a_3 x_1 + b_3 y_1 + c_3 - z_1| \right)$$

4. Distance from Planes (DP):

This uses the distance measures from v_1 to T_3 and v_3 to T_1 .

$$\text{cost} = \|g\|$$

$$\text{where } g = (\text{dist}(T_1, v_3), \text{dist}(T_3, v_1))$$

$$\text{and } \text{dist}(T_j, v_k) = \frac{|a_j x_k + b_j y_k + c_j - z_k|}{\sqrt{a_j^2 + b_j^2 + 1}} \quad j, k \in (1, 3)$$

In addition, Yu et al. [47] proposed their improved cost function, which I call YU for abbreviation.

5. Improved cost function (YU):

They define 2D angle between normals of the two planes, by interpolating the triangles linearly.

$$\text{cost} = \|p_1\| \|p_3\| - p_1 \cdot p_3$$

$$\text{where } p_i = (a_i, b_i) \Rightarrow \|p_i\| = \sqrt{a_i^2 + b_i^2} \quad i \in (1, 3)$$

In addition to these, I have proposed two new cost functions in Chapter 4, which returns better results.

3.2.2 DDT edge swap

Dyn et al. [46] define an edge e , with cost $C(e)$ and enclosed by quadrilateral Q , to be *locally optimal* if it satisfies one of the two conditions:

- a) Q is not strictly convex
 - b) Q is strictly convex and $C(e) < C(e')$ where e' is obtained by swapping e inside Q
- Q

From the above definition, a *locally optimal triangulation* is defined as one where all its edges are locally optimal. The most straightforward way to generate such triangulation would be to use Lawson's local optimization procedure [24].

Figure 3.9: Edge swap in Lawson's algorithm

The basic edge swap is illustrated in Figure 3.9. Below is my implementation of the original edge swapping algorithm proposed by Dyn et al.:

```

1:   Set input mesh as  $M$ ,  $nFlips = 1$ 
2:   Calculate costs of all edges  $\in M$ 
3:   While  $nFlips > 0$ 
4:       Compute edge list  $E$  with internal edges of  $M$  (not on boundary)
5:       For each edge  $e \in E$ 
6:           If  $e$  is enclosed by a convex quadrilateral

```



```

7:           Initialize  $N$  with 4 neighboring edges of  $e$ 
8:           Set  $C(e)$  as cost of  $e$ ,  $C(N)$  as sum of costs of edges  $\in N$ 
9:           Set  $oldCost = C(e) + C(N)$ 
10:          Swap edge  $e$ 
11:          Recalculate costs  $C(e)$  and  $C(N)$ 
12:          Set  $newCost = C(e) + C(N)$ 
13:          If  $newCost < oldCost$ 
14:              Set  $nFlips = nFlips + 1$ 
15:          Else
16:              Swap back edge  $e$ 
17:              Restore old costs of  $e$  and edges  $\in N$ 
18:          End If
19:      End If
20:  End For
21: End While
22: Return new optimized mesh  $M$ , in which no more edge swaps are possible

```

Algorithm 3.1: DDT edge swapping with Lawson's algorithm

An important point to note in this algorithm: With reference to Figure 3.8, the *total cost* of edge $v_2 - v_4$ (as opposed to its *cost*, which can be calculated using any of the methods in Sections 3.2.1 and 4.3.1) after each edge swap is defined as cost of $v_2 - v_4$ + the cost of its four neighboring edges (dark edges). This is reasonable, since the cost of all the 5 edges together gives a better estimate of how swapping $v_2 - v_4$ will affect the region around it, than its cost alone.

3.3 Terrain Generation Methods

After the extensive discussion on triangulation, I finally come to its application of interest: terrain generation. In this section, I look at some existing algorithms, primarily those using triangulated irregular networks, and the rationale for the same.

3.3.1 Regular Grid vs Triangulated Irregular Network

Terrain meshes can be broadly classified into two representations: *Regular Grid* (RG) and *Triangulated Irregular Network* (TIN). A regular grid is a structure that specifies height values at a regular square tessellation of the domain, as shown in Fig. 3.10.

Figure 3.10: A Regular Grid representation with 4096 vertices

Each vertex in the RG typically represents a pixel in the elevation map (or *heightmap*), and although the mesh approximation is most accurate, too many triangles are required to store even a reasonably sized map. For example, a 4096x4096 elevation map would require 16 million vertices. If each vertex occupies 14 bytes (4 bytes each for x, y, z values, and 2 bytes for the texture coordinate – ignoring vertex normals and support for multiple textures), the mesh alone would consume 224 MB of memory, which is large even by today's standard. Even by generating such a terrain, one cannot accurately represent a 4096x4096 feet world, when vertices would be spaced at least 1 foot apart.

With this method, it is impossible to have vertices close-by, say $\frac{1}{4}$ feet apart, in regions where there are tracks. A TIN, on the other hand, can very well represent a 15000x15000 feet in sufficient detail, with just a 2048x2048 heightmap, as seen in Chapter 5.

A TIN, on the other hand, consists of a set of points not following any pattern, scattered across the input domain in varying density. These points are planar triangulated, and every point becomes a vertex in the triangulation. In this manner, the terrain surface is represented as a set of non-overlapping, contiguous triangular faces, of irregular size and shape. The TIN model for terrains has been used since the seventies [48, 49], and an example is shown in Figure 3.11.

Figure 3.11: A TIN representation with just 512 vertices

The prime motivation for the exclusive interest in TINs is that real-world digital elevation maps are non-stationary, i.e. the roughness of the terrain changes rapidly from one area to another. A regular grid representation must be adjusted to the roughest region, and will look highly redundant in smooth areas. A TIN, however, adapts to these changes accurately, without the need for additional techniques like level of detail (LOD), out-of-core streaming, memory optimization, frame-to-frame coherence etc. These techniques are necessitated due to the fundamental drawback of using an RG, i.e. the inability to load the large data volume of the RG representation into memory all at once. All these techniques in turn give rise to further problems such as T-junctions, cracks, popping

effects etc. while rendering. All papers addressing these issues [50-57] are primarily *terrain rendering* algorithms, and not *terrain generation* algorithms, in the true sense of the phrase. If there is efficient terrain generation, which is the focus of this thesis, it automatically guarantees efficient terrain rendering. Hence, without spending any more time on regular grid methods, I will go to existing TIN techniques.

3.3.2 Automatic Extraction

Proposed by Fowler and Little in 1979 [49], this method identifies “feature points” (points of structural importance, such as peaks, pits and passes) from the input data (dense raster models, similar to heightmaps) and triangulates it to get an initial approximation. Next, support points are inserted progressively until the maximum error is reduced below a specified tolerance. The overall idea is similar to my proposed method, but apart from maximum error, I consider many other parameters which can result in a controlled density terrain, with some DDT based optimizations as well, as seen in Chapter 5.

Figure 3.12: Mesh produced by automatic extraction

3.3.3 Radial Sweep algorithm

The Radial Sweep algorithm [58] is a product of two separate triangulations. First it identifies the vertex closest to the centroid of the input data (which is a set of x, y, z points, but might as well be a heightmap), and radially sweeps edges from it towards other points. This is continued until all possible non-overlapping triangles are formed.

The second stage involves a shaping procedure, where skinny triangles are eliminated by edge swaps on convex quadrilaterals wherever possible, as shown in Figure 3.13.

Figure 3.13: After the radial sweep (left) and shaping (right) passes

New points are inserted simply by connecting them with vertices of the triangle it falls inside. This is a very simple idea, and is good for quick generation, but one cannot control the density or shape of the terrain in any manner.

3.3.4 Advancing Front algorithm

Contrary to popular top-down approaches, which start with a coarse mesh and gradually refine it, Silva and Mitchell [59] use a bottom-up approach, by considering the heightmap as a very high density TIN with 4 points per pixel, and gradually simplifying it to fewer triangles, within a specified error bound.

Figure 3.14: Terrain generated by advancing front algorithm

For simplification, a *greedy cuts* approach is used, whereby maximum vertices are removed with least increase in error. They claim it leads to provable good approximation,

but not optimally. I follow a similar approach for DDT optimization procedure of the Delaunay-based terrain (Chapters 4-5).

3.3.5 Greedy insertion algorithms

Garland and Heckbert [60] proposed four greedy algorithms for fast generation of a TIN terrain from a heightmap. Of these, three are based on Delaunay and one on DDT. Their emphasis is mainly on speed, and not on geometry or topology.

Algorithm 1: Start with two triangles, scan through all points; choose next insertion point as one with maximum error. Do edge swaps after each insertion if necessary.

Algorithm 2: Similar to (1), but choose next insertion point based on Delaunay condition, which makes it an incremental Delaunay algorithm.

Algorithm 3: A speed-optimized form of (2); uses a better data structure, heap for next insertion points, and faster interpolation.

Algorithm 4: Uses same data structures as (3), but performs edge swaps based on DDT criteria (a global cost function which also makes use of height information).

They talk of a hybrid Delaunay-DDT algorithm here, but do not give specific details regarding the combination. I have also proposed a similar hybrid with DDT as a post-processing step with my own cost function which yields lowest error, on top of a custom Delaunay refinement procedure, seen in Chapters 4-5.

Figure 3.15: Terrains produced by Algorithm 3 (left) and Algorithm 4 (right)

Garland's approach looks similar to my proposed algorithm, but there are significant differences:

1. In addition to a single maximum error, I also take in minimum error and minimum area as optional inputs. This gives the user greater control over the output mesh (Chapter 5).
2. I introduce a new concept called control map, by which only specific regions of the terrain can be modified separately, with different densities (Chapter 5).
3. Instead of a simple incremental Delaunay insertion criteria, I use a modified form of Chew's second algorithm [44], which refines the mesh by careful insertion and deletion of vertices, after more considerations (Chapter 4).
4. In the DDT step, there is no notion of edge cost and swapping edges with the highest cost. They simply use "error over a triangle" as a measure (using L_2 or L_∞ norm) to insert points into the mesh. Therefore the output mesh can obviously be optimized further, which is what I attempt to do with my new cost functions, constrained DDT algorithm and the terrain simplification algorithm (Chapter 4).

3.3.6 Improved TIN refinement

Proposed recently by Pedrini [61], it is similar to Garland's approach [60], except that instead of mere greedy insertion, a decimation procedure runs in parallel to refinement.

The next insertion point is chosen as the one with maximum error, and the point to be removed is chosen as the one with minimum error.

Figure 3.16: Pedrini's approach on the Lake Crater elevation map

This simplistic approach, which can be implemented quickly, at best gives local optimization due to its nature: every decimation routine only considers the current triangulation and not the final one. There is no way to guarantee global minimum, even by using a global error function, as the authors have stated. Further, there is no way to perform a controlled mesh refinement, with varying densities in different regions. I have done this with my proposed algorithm in Chapter 5.

3.4 Quad-Edge data structure

The Quad-Edge data structure was invented by Guibas and Stolfi in 1985 [31], and was designed for representing general subdivisions of orientable manifolds. It is similar to Baumgart's Winged-Edge data structure [62], but it can simultaneously represent both a mesh and its *dual*, which makes it an ideal candidate for computing Delaunay-based meshes and their dual Voronoi diagrams simultaneously.

I have used this data structure to do all mesh operations: refining the mesh, maintaining Delaunay condition after each refinement step, performing edge swaps, calculating costs, retaining constrained edges, inserting and deleting vertices, and mesh simplification by

edge collapse etc. It is simple to understand and implement, and provides convenient edge, face and vertex traversal through simple pointer operations. Since speed was one of the considerations, along with the major ones of ideal shape and optimal vertex generation, I chose this for my implementation.

Figure 3.17: Representation of one quad-edge

The basic Quad-Edge structure proposed by its author is shown in Figure 3.17, where each quad consists of four edges, $e[0]$ to $e[3]$. $e[0]$ represents the primal edge and $e[2]$ its symmetric inverse, whereas $e[1]$ represents the dual edge and $e[3]$ its symmetric inverse.

In Guibas and Stolfi's original representation, each edge e points to 3 things:

- The vertex at its origin ($e \rightarrow \text{Org}$)
- The next counterclockwise (*ccw*) edge bound outward from its origin vertex ($e \rightarrow \text{next}$)
- A data field, which can be used for storing coordinates of the origin vertex

They also define many relationships to neighboring edges, which form the basis for easy navigation and traversal. These are illustrated and explained below:

Figure 3.18: Edge relationships of a quad-edge

Several relationships can be formed elegantly using fast pointer algebra (Figure 3.18):

- The quad edge represented by e is actually $e[0]$
- $e \rightarrow \text{Sym} = e[2]$, $e \rightarrow \text{Rot} = e[1]$, $e \rightarrow \text{InvRot} = e[3]$
- $e \rightarrow \text{Onext} = e \rightarrow \text{next}$
- $e \rightarrow \text{Oprev} = e \rightarrow \text{Rot} \rightarrow \text{Onext} \rightarrow \text{Rot}$
- $e \rightarrow \text{Dnext} = e \rightarrow \text{Sym} \rightarrow \text{Onext} \rightarrow \text{Sym}$
- $e \rightarrow \text{Dprev} = e \rightarrow \text{InvRot} \rightarrow \text{Onext} \rightarrow \text{InvRot}$
- $e \rightarrow \text{Lnext} = e \rightarrow \text{InvRot} \rightarrow \text{Onext} \rightarrow \text{Rot}$
- $e \rightarrow \text{Lprev} = e \rightarrow \text{Onext} \rightarrow \text{Sym}$
- $e \rightarrow \text{Rnext} = e \rightarrow \text{Rot} \rightarrow \text{Onext} \rightarrow \text{InvRot}$
- $e \rightarrow \text{Rprev} = e \rightarrow \text{Sym} \rightarrow \text{Onext}$

However, the above edge representation is still too simple, and has been modified by Heckbert [63] to additionally store face information. He uses additional data structures for face and vertex representations, which further simplifies basic operations and provides means for queries such as finding (i) all the outgoing edges from a certain vertex, (ii) all the boundary edges/vertices of a certain face, (iii) all the neighboring vertices of a given vertex etc. They make life easier for performing operations like vertex insertion and deletion, edge splitting etc.

I have further modified it to store indices for each vertex and face, so that it can be used directly for rendering (using DirectX vertex and index buffers). I have also included costs for each edge and face, as they are used by my constrained DDT and terrain simplification procedures (Chapter 4). My changes are shown as *grayed, bold, italicized* text.

Modified edge structure points to:

- The vertex at its origin ($e \rightarrow \text{Org}$)
- The next ccw edge bound outward from its origin vertex ($e \rightarrow \text{next}$)
- The face it points to ($e \rightarrow \text{face}$), which is NULL for primal edges $e[0]$ and $e[2]$
- *Edge cost ($e \rightarrow \text{cost}$), which is the maximum of the two adjacent face costs.*

The vertex structure points to:

- An outgoing edge ($v \rightarrow \text{edge}$)
- A data field to store its coordinates ($v \rightarrow \text{pos}$)
- *A zero-based unique index ($v \rightarrow \text{id}$)*
- *Additional height information ($v \rightarrow \text{height}$), used only for DDT optimization (not Delaunay refinement)*

The face structure points to:

- An adjacent edge ($f \rightarrow edge$), which is NULL for degenerate faces
- A zero-based unique index ($f \rightarrow id$)
- Face cost ($f \rightarrow cost$), calculated using the L_2 norm as the mean square error of all the pixels lying inside the face

With these modifications to the data structure, I had to change basic topological operations such as *Connect()*, *DeleteEdge()*, *MakeEdge()*, *Swap()* etc. defined by Guibas and Stolfi, to update face and vertex pointers as well.

An important bug in Guibas and Stolfi's *Locate()* procedure [31] was found by Heckbert [64]. It is explained below:

```

PROCEDURE Locate[X] RETURNS [e]:
  e ← some edge;
  DO
    IF X = e.Org OR X = e.Dest THEN RETURN e
    ELSIF RightOf[X, e] THEN e ← e.Sym
    ELSIF NOT RightOf[X, e.Onext] THEN e ← e.Onext
    ELSIF NOT RightOf[X, e.Dprev] THEN e ← e.Dprev
    ELSE RETURN e FI
  OD
END Locate.

```

Algorithm 3.2: Locate() procedure to search for a vertex

The *Locate()* procedure is for locating the face (or its adjacent edge) inside which a given vertex X lies. It starts with any reference edge e in the mesh, and uses the “walking method” of Green and Sibson [65].

Figure 3.19: Illustration of a bug in the *Locate()* procedure

With reference to Figure 3.19, if one starts with edge e_1 , the algorithm loops forever with the sequence of edges “ $e_2 - \dots - e_6 - e_1 - \dots$ ” as shown. Thus Algorithm 3.1 does not always terminate. This procedure is used primarily for point insertion into the Quad-Edge mesh, during Delaunay refinement.

There is no simple way to resolve this using the existing walking method, except to do simple point-inside-triangle tests after traversing all edges once, or after coming back to the same reference edge a few times without success.

So far I have not come across this bug, but I have taken measures for resolution when it happens. My fix is a bit expensive, but it guarantees that there are no infinite loops:

- (i) Perform a simple convex hull test to check if point lies inside the mesh. If it does, continue, else return NULL.
- (ii) On coming back to the same reference edge a few times, or after traversing all the edges without success, perform simple point-inside-triangle tests to check which face the point lies inside.
- (iii) Return the first face the point lies inside. If no such face is found, return NULL.

Chapter 4

Proposed Algorithms

This chapter looks at my proposed algorithms. First is an intuitive triangulation algorithm for a given set of points. This arose more out of need than out of adventure, for finding the 3D convex hull of a set of points. Next is the new CCDT algorithm, which is the basis for my terrain generation algorithm. Lastly, I go through my proposed constrained DDT algorithm and new cost functions, which are used for terrain optimization.

4.1 An “Intuitive” triangulation algorithm

I extend the idea of the 2D incremental convex hull algorithm (Section 2.4.1) to compute the triangulation of a 2D point set or 3D coplanar point set. The algorithm is “intuitive” in the sense that it is possibly the simplest way to triangulate a point set. It does not guarantee any properties like Delaunay triangulation, but it can be easily implemented. I use this to compute the 3D planar convex hull, as a special case, when Step 2 of Algorithm 2.1 fails.

Let $P = \{p_1, p_2, \dots, p_n\}$ represent a set of n points in 2-space (or coplanar 3-space). I need to compute its intuitive triangulation, $IT(P)$. Let P_r represent the set of r points $\{p_1, p_2, \dots, p_r\}$ for $r \in (1, n)$.

```

1:   Choose three non-collinear points  $p_1, p_2, p_3$ 
2:   Construct triangle  $p_1p_2p_3$ , and initialize  $IT(p_3)$ 
3:   For  $r = 4$  to  $n$ 
4:       If  $p_r$  lies on or inside  $IT(p_{r-1})$ 
5:           Find triangle(s) (1 or 2)  $T_{old}$ , that  $p_r$  lies on
6:           Add  $p_r$ , re-triangulate  $T_{old}$  to get  $T_{new}$  ( $1 \rightarrow 3$  or  $2 \rightarrow 4$ )
7:            $IT(p_r) = IT(p_{r-1}) - T_{old} + T_{new}$ 
8:       Else If  $p_r$  lies outside  $IT(p_{r-1})$ 
9:           Find set of edges  $E$  in  $IT(p_{r-1})$  visible from  $p_r$ 
10:          Connect each edge in  $E$  to  $p_r$ , form new triangles  $T_{new}$ 
11:           $IT(p_r) = IT(p_{r-1}) + T_{new}$ 
12:       End If
13:   End For
14:   Get  $IT(P) = IT(p_n)$ 

```

Algorithm 4.1: Intuitive triangulation algorithm

Algorithm 4.1 is almost similar to the 3D convex hull construction (Section 2.4.4), except that now I deal in 2-space and do not discard p_r if it lies inside the intermediate triangulation $IT(p_{r-1})$. It is computationally faster to check for visible edges instead of visible facets when p_r lies outside $IT(p_{r-1})$.

4.2 New CCDT algorithm

I have implemented a variation of Chew's second algorithm (Section 3.1.3), in order to reduce the number of Steiner points inserted to a greater degree. Shewchuk [66]

originally implemented a variation of Chew's second algorithm in his *triangle* program [67, 68], taking into account domains with small angles by using the Miller-Pav-Walkington algorithm [69]. Even though it is theoretically guaranteed to give better results, this is not needed for terrain generation, where the input is a simple PSLG.

Described below is the proposed CCDT algorithm, particularly for terrain generation. Later, if the need arises, I may adopt Shewchuk's approach, consider domains with small angles and use diametral lenses instead of diametral circles. But right now, it is not necessary.

Some terminology: A triangle which is too big (some user constraints are not satisfied) is called a *bad triangle* and a triangle which has one or more angles less than some minimum angle bound is a *skinny triangle*. Steiner points lying on subsegments are *locked* Steiner points (since they cannot be removed once inserted), whereas those inserted at the circumcenter are *unlocked* Steiner points, as they can be removed during further refinement.

1:	Create bounding vertices and segments, if necessary, and modify input PSLG X
2:	Initialize segment list S with segments in modified X
3:	Compute $CDT(X)$, initialize $CCDT(X) = CDT(X)$
4:	For each bad triangle $t \in CCDT(X)$
5:	For each segment $s \in S$
6:	If s is encroached by t
7:	Find all Steiner points P_s inside the diametral circle of s
8:	Remove all unlocked Steiner points $\in P_s$
9:	Split s , add the two subsegments to the segment list


```

10:          Add midpoint of  $s$  as a new locked Steiner point
11:          Retriangulate; update  $CCDT(X)$ 
12:      End If
13:  End For
14:  If no segment  $\in S$  is encroached by  $t$ 
15:      Add circumcenter of  $t$  as a new unlocked Steiner point
16:      Retriangulate; update  $CCDT(X)$ 
17:  End If
18: End For
19: Perform steps 4-18 for each skinny triangle in  $CCDT(X)$ 
20: Return  $CCDT(X)$  as output mesh, which satisfies all user and angle constraints

```

Algorithm 4.2: Proposed CCDT algorithm

There are two main differences between Chew's second algorithm and the proposed algorithm.

The first is that steps 7-11 are performed for all segments s encroached by t , and not merely those visible from t . The second difference is that in step 8, I remove all unlocked Steiner points inside the diametral circle, whereas Chew removes only those *visible* from the midpoint of s (here visibility between two points p_1 and p_2 means a line connecting them does not cut across a segment or subsegment). The rationale behind both these differences is simple: since the initial PSLG is a rectangle, all input segments lie along the boundary of the terrain, and there is no need to perform any sort of visibility test.

The second difference is that Algorithm 4.2 considers both skinny triangles and bad triangles whereas Chew considers only skinny triangles; but this can hardly be called a

difference since this algorithm is implemented with terrain generation in mind (Chapter 5), and eliminating bad triangles is the only way to accommodate user constraints.

These differences can be better understood by an example:

Figure 4.1: Elimination of a skinny triangle – comparison with Chew's algorithm

In Figure 4.1, on the left is a triangulation with skinny triangle T , which needs to be eliminated. The darkened edges are segments which cannot be removed and c is the circumcenter of T . It is clear that T being encroached by both the segments, but first I consider the segment on the right, whose diametral circle is shown by dotted lines.

In one pass of Chew's second algorithm, a vertex is inserted at the midpoint m of the segment, and all unlocked vertices lying inside the diametral circle, and visible from m are removed. This results in a triangulation (top right), where only v_2 is removed and T is eliminated.

In contrast, one pass of my proposed algorithm (Algorithm 4.2, steps 7-11) inserts a vertex at m and removes both v_1 and v_2 , resulting in a triangulation with fewer skinny triangles (bottom right). The reason for this is that I do not perform the visibility test, which only results in increased processing time and more passes to achieve the same results.

4.3 Proposed Constrained DDT algorithm

In order to eliminate the rendering artifacts due to extremely skinny triangles, inevitable in the original DDT edge swap procedure (Section 3.2.2), I have proposed a *Constrained DDT* (CDDT) algorithm (Algorithm 4.5). Akin to the proposed CCDT algorithm (Section 4.2), it takes in a minimum angle constraint, and performs more robust edge swapping using the Look-Ahead algorithm together with my new cost functions.

Before looking at the proposed constrained DDT algorithm (which uses two new cost functions), the proposed CF6 and CF7 cost functions are explained first.

4.3.1 New DDT cost functions

Section 3.2.1 shows 5 cost functions viz. ABN, JND, DLP, DP and YU, as a possible measure of error. Although they associate cost based on distance, normals, angles and other criteria, they did not return good results for terrain optimization, where the goal is to reduce the overall mean square error of a terrain via edge swaps.

For this purpose, I introduce two new cost functions, specifically for reducing terrain error. The results show that they are better than previous cost functions. I use the same figure used in Section 3.2.1 for computing cost.

Figure 4.2: Initial triangulation for computing cost

Some terminology: cost of edge $v_2 - v_4$ is computed, which is shared by triangles T_1 and T_3 . For each pixel $p \in T_1 \cup T_3$, $H(p)$ is the height of p from the input heightmap and $I(p)$ is its interpolated height. The total number of pixels in T_1 and T_3 is denoted by $N(T_1)$ and $N(T_3)$ respectively.

1. Cost based on Maximum Error (CF6):

$$\text{cost} = \max_{p \in T_1 \cup T_3} |H(p) - I(p)|$$

2. Cost based on Mean Square Error (CF7):

$$\text{cost} = \max \left(\frac{\sum_{p \in T_1} |H(p) - I(p)|^2}{N(T_1)}, \frac{\sum_{p \in T_3} |H(p) - I(p)|^2}{N(T_3)} \right)$$

Given below are the algorithmic pseudo-codes for determining CF6 and CF7:

- 1: Initialize cost C of edge $v_2 - v_4$ to zero
- 2: Initialize maximum errors of T_1 and T_3 as m_1 and m_3 , to zero
- 3: **For** each pixel $p \in T_1$
- 4: Calculate height $H(p)$ of p from input heightmap
- 5: Calculate interpolated height $I(p)$ of p from vertices of T_1 (v_1, v_2, v_4)


```

6:      Set  $\varepsilon = |H(p) - I(p)|$ 
7:      Set  $m_1 = \max(\varepsilon, m_1)$ 
8:  End For
9:  Repeat steps 3-8 to calculate  $m_3$  from  $T_3$ 
10: Set  $C = \max(m_1, m_3)$ 
11: Return  $C$  as the cost of edge  $v_2 - v_4$ 

```

Algorithm 4.3: Computing cost based on maximum error

```

1:  Initialize cost  $C$  of edge  $v_2 - v_4$  to zero
2:  Initialize mean square errors of  $T_1$  and  $T_3$  as  $m_1$  and  $m_3$ , to zero
3:  Initialize  $S = 0, count = 0$ 
4:  For each pixel  $p \in T_1$ 
5:      Calculate height  $H(p)$  of  $p$  from input heightmap
6:      Calculate interpolated height  $I(p)$  of  $p$  from vertices of  $T_1$  ( $v_1, v_2, v_4$ )
7:      Set  $\varepsilon = H(p) - I(p)$ 
8:      Set  $S = S + \varepsilon^2$ 
9:      Set  $count = count + 1$ 
10: End For
11: Set  $m_1 = S / count$ 
12: Repeat steps 3-11 to calculate  $m_3$  from  $T_3$ 
13: Set  $m_1$  as cost of face  $T_1$ ,  $m_3$  as cost of face  $T_3$  (for future reuse)
13: Set  $C = \max(m_1, m_3)$ 
14: Return  $C$  as the cost of edge  $v_2 - v_4$ 

```

Algorithm 4.4: Computing cost based on mean square error

4.3.2 Look-Ahead algorithm

Dyn et al. claim that a locally optimal triangulation is obtained using Lawson's algorithm for edge swapping [46]. However, Yu et al. shows that a Look-Ahead approach, which takes into consideration more edges other than just the four neighboring edges, returns better results for image reconstruction [47].

I tested both algorithms, and found out that the Look-Ahead approach is better for denser terrains when used with my proposed cost functions. Overall, it returned better results than any of the existing cost functions used with Lawson's algorithm. The output terrains and a comparison of results are shown in Chapter 5.

Some terminology, followed by my constrained DDT Look-Ahead algorithm:

1. An edge e is *convex* if it lies inside a convex quadrilateral, and is denoted by $CX(e)$.
 $A(e)$ implies that both the triangles adjacent to e satisfy the *minAngle* constraint.
 When e is swapped, it is denoted by e' .
2. The *total cost* of edge $v_2 - v_4$ (as opposed to its *cost*, which can be calculated using any of the methods in Sections 3.2.1 and 4.3.1) after each edge swap is defined as cost of $v_2 - v_4$ + the cost of its four neighboring edges (darkened edges).
3. The "look-ahead" edges of $v_2 - v_4$ in Figure 4.1 are the 8 non-darkened edges.
 Together with $v_2 - v_4$, all the 13 edges are considered, instead of just the 5 darkened ones in Lawson's algorithm.

- 1: Set input mesh as M , $nFlips = 1$
- 2: Calculate costs of all edges $\in M$

```

3:   While  $nFlips > 0$ 
4:       Compute edge list  $E$  with internal edges of  $M$  (not on boundary)
5:       For each edge  $e \in E$ , if  $CX(e)$  and  $A(e')$ 
6:           Initialize  $L$  with 8 “look-ahead” edges of  $e$ 
7:           Set  $C(e)$  as total cost of  $e$  ( $e + 4$  neighbors)
8:           Set  $C(L)$  as sum of costs of edges  $\in L$ 
9:           Set  $oldCost = C(e)$ ,  $oldLACost = C(L)$ 
10:          Swap edge  $e$ 
11:          Recalculate cost  $C(e)$ , set it as  $newCost$ 
12:          If  $newCost < oldCost$ 
13:              Set  $nFlips = nFlips + 1$ 
14:          Else
15:              For each edge  $g \in L$ , if  $CX(g)$  and  $A(g')$ 
16:                  Set  $C(g)$  as total cost of  $g$  ( $g + 4$  neighbors)
17:                  Set  $newCost = C(L) - C(g)$ 
18:                  Swap edge  $g$ 
19:                  Recalculate cost  $C(g)$ 
20:                  Set  $newLACost = newCost + C(g)$ 
21:                  If  $newLACost < oldLACost$ 
22:                      Set  $nFlips = nFlips + 1$ 
23:                      Break out of For loop
24:                  Else
25:                      Swap back edge  $g$ 
26:                      Restore old costs of  $g$ , 4 neighbors

```

27:	End If
28:	End For
29:	If any edge $\in L$ was swapped in steps 15-28
30:	Swap back edge e
31:	Restore old costs of e , 4 neighbors
32:	End If
33:	End If
34:	End For
35:	End While
36:	Return new optimized mesh M , in which no more edge swaps are possible

Algorithm 4.5: Constrained DDT with Look-Ahead algorithm

A summary of Algorithm 4.5, followed by an illustration of the Look-Ahead algorithm:

1. For each edge, check if it swappable based on two criteria: it lies inside a convex quadrilateral, and all 6 angles of both adjacent triangles are greater than *minAngle* (the minimum angle constraint) after a swap.
2. If it is swappable, perform the normal edge swap check as per Lawson’s criteria. If it succeeds, swap it. But if it fails, do not immediately reject it. Instead, try to swap it in conjunction with any of its 4 neighboring edges. If the combination of two swaps results in lowered error, accept it; else, reject the edge.

Figure 4.3: Initial triangulation

Figure 4.4: Possible Look-Ahead outputs

Consider an initial triangulation shown in Figure 4.3(a). If sum of the dark edges' cost in 4.3(a) is lesser than the sum of dark edges' cost in 4.3(b), this eliminates 4.3(b) as a possible output. My algorithm then compares the total cost c_1 of all 13 edges in 4.3(a) with the total cost c_2 of all 13 edges in 4.4(a). If $c_2 < c_1$, the triangulation in 4.4(a) replaces the triangulation in 4.3(a). Otherwise, the algorithm compares the cost of c_1 to the cost of all edges in 4.4(b), and then, if needed, to 4.4(c) and 4.4(d).

For brevity, I have only mentioned constrained DDT with the Look-Ahead algorithm. Lawson's optimization procedure (Section 3.2.2) can also be easily incorporated into it. A comparison of pure Delaunay, constrained DDT with Lawson's and constrained DDT with Look-Ahead algorithms is given in Section 5.6, which also compares the performance of the various cost functions. The results show that the combination of constrained DDT with Look-Ahead and my proposed cost function CF7 is best.

4.4 Terrain Simplification algorithm

In the previous section, I showed how to optimize edge costs, and thereby overall error, using edge swaps. However, edge swapping does not reduce the number of vertices/triangles, which is the "actual" optimization from the rendering point of view. In order to do so, I propose a simplification algorithm, which reduces the triangle count given an upper bound for terrain mean square error.

There are popular existing simplification methods, so I will go through them first to show why they are inadequate in this context.

4.4.1 Existing techniques

One of the first mesh simplification techniques was the *Decimation* method proposed by Schroeder et. al. [70]. It removes one vertex at a time, without violating the neighbor's topology, if the resulting surface lies within a user-specified distance of the original geometry. Though intuitive, it works best for generating visually appealing simplified LOD models, and not terrains, since stress is given to appearance rather than error tolerance. My technique is similar to Schroeder's, but with improvements: It removes one vertex at a time, but ensures that each removal results in *overall* lowest terrain mean square error (as opposed to *local* topology preservation based on distance).

The next major simplification technique was Hoppe's *Progressive Meshes* [53], where edge are greedily collapsed based on an energy function, until no more edges can be collapsed. Hoppe's method was dynamic and applicable to all triangular meshes, not just terrains, and the edge collapses were reversible. My method is non-reversible, static, and uses mean square error as the energy function. Obviously, this results in lowered memory usage and faster processing.

Another technique which gained wide-spread popularity was Garland and Heckbert's *Quadric Error Metrics* [71]. The simplification is performed by a sequence of vertex merge operations, where each vertex is associated with a quadric error metric (QEM) and the merge which results in lowest QEM is performed first etc. QEM is based on distance of a vertex to the plane of its surrounding faces. This is again a greedy algorithm, and an improvement over Schroeder's method [70], which we saw earlier. However, it has the same limitation, i.e. using distance as a measure, and optimizing locally instead of globally.

There are other notable dynamic simplification algorithms. For more information, refer to Luebke's excellent survey on simplification methods [72].

4.4.2 Proposed method

Below is my proposed algorithm, followed by a brief explanation:

```

1:   Set input mesh as  $M$  ,  $mse = 0$ 
2:   Calculate heightmap  $H$  from  $M$ 
3:   Initialize  $nPixels$  = number of pixels in  $H$ 
4:   Calculate cost of terrain  $cost\_terrain$  from  $H$ 
4:   Set mean square error  $mse = (cost\_terrain / nPixels)$ 
5:   Compute costs of all edges  $\in M$ 
6:   Set  $e$  = lowest cost fully internal edge of  $M$  (both vertices not on boundary)
7:   While  $e \neq NULL$  And  $mse \leq target\_mse$ 
8:       Set  $v_{ORG}$  = origin vertex of  $e$  ,  $v_{DEST}$  = destination vertex of  $e$ 
9:       Set  $F_{OLD}$  = union of set of faces surrounding  $v_{ORG}$  and  $v_{DEST}$ 
10:      Calculate  $cost\_old$  = sum of costs of faces  $\in F_{OLD}$ 
11:      EdgeCollapse  $e$ 
12:          Split edge  $e$  , set new vertex as  $v_{NEW}$ 
13:          Delete  $v_{ORG}$ 
14:          Delete  $v_{DEST}$ 
15:      End EdgeCollapse
16:      Set  $F_{NEW}$  = set of faces surrounding  $v_{NEW}$ 
17:      Recalculate costs of all edges  $\in F_{NEW}$ 

```



```

18:      Calculate  $cost\_new = \text{sum of costs of faces } \in F_{NEW}$ 
19:      Set  $cost\_change = cost\_new - cost\_old$ 
20:      Set  $mse = mse + (cost\_change / nPixels)$ 
21:      Set  $e = \text{next lowest cost fully internal edge of } M$ 
22:  End While
23:  Undo last edge collapse
24:  Return new optimized mesh  $M$ , in which no more edges can be collapsed

```

Algorithm 4.6: Terrain simplification algorithm

Algorithm 4.6 is basically a greedy edge collapse method, because I always remove the lowest cost edge, thus trying to reduce as many triangles as possible while trying to remain below the given tolerance.

In Algorithm 4.6, the cost of terrain and change in cost after every edge collapse operation is based on *square error*, and not mean square error. Hence I need to divide by $nPixels$ accordingly. The $target_mse$ used for comparison in step 7 can be based on a more coarse mesh with greater error or a user specified value. I have set it to the mean square error of a purely Delaunay-based mesh, before the constrained DDT optimization (using Lawson's or Look-Ahead algorithms). A comparison of terrains and their associated mean square errors, generated with and without optimizations, is shown in Chapter 5.

Figure 4.5: Basic edge collapse operation

I have implemented each edge collapse (steps 8-12 in Algorithm 4.6, Figure 4.2) based on three simple operations:

(i) Splitting the edge (ii) Deleting the origin vertex (iii) Deleting the destination vertex

In this manner, a single edge collapse results in the reduction of 1 vertex, 3 edges and 2 faces. This is confirmed by Euler's formula $V - E + F = 2$.

Chapter 5

Terrain Generation

Using the CCDT algorithm discussed in Section 4.2, I generate a well-graded terrain which is simple, yet captures the required detail at user-specified regions by using a heightmap, a control map and several user defined parameters. Further optimization is done by the new cost functions, constrained DDT algorithm and the terrain simplification algorithm, discussed earlier in Sections 4.3-4.5.

5.1 Heightmap

A *heightmap* is used to store elevation data for the terrain. The heightmap, as defined earlier, is a grayscale image where white pixels indicate the highest points, black ones the lowest, and shades of gray for in-between heights. Each pixel is usually made up of 8 or 16 bits, thereby allowing storage of 2^8 or 2^{16} different height values. Figure 5.1 shows a heightmap and the corresponding triangular mesh representation.

Figure 5.1: Heightmap (left) and corresponding triangular mesh (right)

5.2 Control Map

Besides the heightmap, I introduce something called a *control map*, to indicate the areas in the terrain for which more details need to be added. These areas are called *control regions*. The control map is also a grayscale image, where the control regions, with whiter pixels, are triangulated more densely than the darker regions (see Figure 5.2).

Figure 5.2: Control map showing control regions in white

Using a control map is not only intuitive, but also handy for the artists who want to increase detail only in certain areas of the terrain, like areas which have tracks, or the areas where the camera focuses often. I use the control map for terrain generation, but it can be easily extended for more powerful use in other areas. The actual density of triangulation in the control regions is controlled by user parameters like minimum and maximum height difference error.

5.3 User constraints

Before looking at the algorithm, one needs to understand what the user constraints are, that I have been talking about. They are a set of parameters specifying the amount of usage of the control map in the control regions, and other mesh refinement controllers like minimum area and minimum angle.

To control the results of the generated terrain, I introduce eight parameters: *SizeX* , *SizeY* , *Height* , *MinArea* , *MinError* , *MaxError* , *MinAngle* and *SkipScale* . A screenshot of the user parameters from the 3dsMax plugin interface I implemented for terrain generation is shown in Figure 5.3.

Figure 5.3: User parameters for terrain generation algorithm

SizeX and *SizeY* specify the length and width of the terrain. *Height* is the maximum height of the terrain (all the height values from the heightmap will be mapped from 0 to this value), *MinArea* is the minimum area of any triangle in the projected 2D mesh. The height difference error at any point on the terrain lying inside the control region is guaranteed to be less than or equal to *MinError* , whereas for the other regions, it is guaranteed to be not greater than *MaxError* . In the absence of a control map, *MaxError* is ignored and *MinError* is the upper bound for height difference error.

The algorithm further guarantees that all internal angles of the projected triangles (found by projecting the terrain onto the *xy* plane) are greater than or equal to *MinAngle* . Ruppert [45] shows that the upper limit for *MinAngle* is 20.7° for his Delaunay refinement algorithm (Section 3.1.2), but Shewchuk [67], who has combined Ruppert’s algorithm with Chew’s second algorithm (Section 3.1.3), has argued that in practice, adding new vertices fails only when it exceeds 33.8° . Since my proposed algorithm (Algorithm 4.2) is similar to Shewchuk’s, to be safe, I have used an upper limit of 33°

for *MinAngle* , lest should the algorithm never terminates. *SkipScale* determines the overall accuracy of the algorithm. The next section shows how each of these parameters is used for terrain generation.

These parameters, in addition to the control map, give a lot of flexibility to my algorithm, thus allowing artists to generate terrains having the same height and control maps as inputs, at different resolutions (discrete LOD models).

5.4 Terrain Generation algorithm

The algorithm uses a top-down approach for terrain generation. Initially, I begin with the four corner vertices of the heightmap, gradually refining it through the proposed CCDT algorithm. To put it in a nutshell, the workflow is shown in Algorithm 5.1, since the CCDT algorithm (Algorithm 4.2) is specific to terrain generation and takes care of everything else.

Algorithm 5.1: Terrain generation overview

However, this hardly explains the details. Therefore, Algorithm 5.2 shows a more detailed flowchart, expanding out the CCDT part.

Algorithm 5.2: Terrain generation flowchart

From the four corner vertices, I construct a rectangular PSLG and perform an initial CDT, which simply splits the rectangle into two triangles. Then for each triangle, I check if it satisfies the *MinArea* and *MinAngle* constraints. If it doesn't, it is rejected; else I proceed to do the more computationally expensive height constraint check.

For the height constraint check, I choose 1 to 40 equally distributed pixels inside each triangle, the exact number depending on the *SkipScale* parameter and the size of the current triangle. Next I check if all of them satisfy the height constraint, by comparing them with the heightmap and control map. If the control map is not specified, one can either assume an all-white or all-black one as default (I use all-white). If all the pixels satisfy the constraint, the triangle is accepted; otherwise, it is rejected.

The rejected triangles are then passed back to the CDT algorithm as a new PSLG, to which Steiner points are added and re-triangulated. This check is continued for the split triangles as well, until all the triangles formed satisfy all three constraints.

5.5 Terrain Optimization algorithms

Terrain optimization can be split into two parts:

- (i) Reduction of error for fixed triangle count
- (ii) Reduction of triangle count for fixed error

While the former can be implemented using constrained DDT with either Lawson’s or Look-Ahead algorithms (Section 4.3), the latter makes use of the terrain simplification algorithm (Section 4.4), removing one vertex at a time until the desired error level is reached. Part (i) can use any of the 7 cost functions – 5 existing or 2 proposed by me, whereas Part (ii) uses just mean square error as a measure, since it is fixed.

Algorithm 5.3: Optimization of mean square error

Algorithm 5.4: Optimization of triangle count

The first three steps in both Algorithms 5.3 and 5.4 are the same as in normal terrain generation (Algorithm 5.1). As seen in Chapter 4, constrained DDT optimization reduces mean square error, but not triangle count. On the other hand, terrain simplification performs edge collapses and reduces triangle count, while increasing the error to within a specified upper bound.

The idea of the *control map* introduced in Section 5.2 can be used for optimizing only a part of the terrain (default control map is all-white, covering the whole terrain). This is especially useful for maintaining sharp edges in only certain regions of the terrain, where there are cliffs, ridges, passes etc.

5.6 Results

I implement the terrain generation and optimization algorithms as 3dsMax plugins, shown below:

Figure 5.4: User interfaces of (left) terrain generation and (right) terrain optimization plugins

5.6.1 Terrain generation

All the user parameters shown in Figure 5.4 (left) were already explained in Section 5.2. There is an additional option to use an ‘Initial Edges’ object, whose boundary is retained in the final mesh. If ‘Stencil Object’ is also checked, the final mesh is similar to generating a mesh without checking ‘Stencil Object’ and then applying the stenciling algorithm over it. This is described in Section 6.1.

The algorithm always generates a terrain with such a property (P) that the control regions and those parts where there are a lot of height variations are tessellated more densely, whereas fewer triangles are used for the relatively flat and non-control regions.

Figure 5.5: Inputs heightmap (left) and control map (right)

Figure 5.6: Output terrain mesh: (left) high resolution with 175K triangles
(right) low resolution with just 75K triangles

The terrain in Figure 5.6 (left) was generated with the following parameters: $SizeX = SizeY = 15840$, $Height = 2000$, $MinArea = 10$, $MinError = 5$, $MaxError = 50$, $MinAngle = 0^\circ$ and $SkipScale = 1$. The two other inputs are the heightmap and control map, shown in Figure 5.5.

Furthermore, by modifying the error parameters, a terrain of any resolution can be generated, with reduced number of triangles, while at the same time maintaining the property P . For the terrain in Figure 5.6 (right), $MinError$ and $MaxError$ were set to 10 and 100 respectively, while all the other parameters remained the same. As it can be seen, this terrain looks reasonably detailed in the “non-flat” and control regions, but the other areas are greatly simplified to produce a mesh with $> 55\%$ reduction in the number of

triangles. This gives great flexibility to the artist who can generate multiple resolutions of the same terrain, which can then be used in the game in possibly different rendering contexts.

5.6.2 Terrain optimization

The top views of the purely Delaunay-based terrain, constrained DDT terrain and the simplified terrain are shown in Figures 5.7-5.9. For the constrained DDT and simplified terrains, two versions are shown, using minimum angles of 0° and 20°.

Figure 5.7: Delaunay mesh (1122 vertices, 2218 faces, error = 0.554)

Figure 5.8: Constrained DDT meshes with 0° (left) and 20° (right) minimum angles (vertices, faces, error) = (1122, 2218, 0.374 – left) and (1122, 2218, 0.443 – right)

Figure 5.9: Simplified meshes with inputs from Figure 5.8

(vertices, faces, error) = (591, 1155, 0.553 – left) and (713, 1400, 0.5527 – right)

From Figures 5.7-5.8, it can be seen that when minimum angle is 0° , mean square error is reduced from 0.554 to 0.374, a reduction of 32%. For the 20° case, the reduction is only 20%, from 0.554 to 0.443, but all the rendering artifacts are removed (Figure 5.12).

Using the output meshes in Figures 5.8 as input to the simplification algorithm, I fixed the upper bound of error in order to reduce the triangle count. Comparing Figures 5.7 and 5.9, it can be seen that the triangle count has been reduced from 2218 to 1155 for the 0° case and from 2218 to 1400 for the 20° case. In terms of percentages, they amount to 48% and 37% respectively, a whopping reduction!

0° gives better results in terms of percentage of error reduction (since it swaps maximum number of edges), but introduces rendering artifacts due to very skinny triangles when compared to its Delaunay counterpart. This is shown in Figure 5.11, due to the fact that face normal calculation, which involves cross product of edges, is inaccurate for very skinny triangles (which contain very small edges).

Figure 5.10: Shaded view of Delaunay mesh

Rendering is smooth because Delaunay triangulation maximizes minimum angle

Figure 5.11: Shaded view of Constrained DDT mesh with 0° minimum angle

Rendering artifacts are introduced due to very skinny triangles

Figure 5.12: Shaded view of Constrained DDT mesh with 20° minimum angle

Rendering artifacts are removed completely due to the angle constraint

By using a minimum angle of 20°, these artifacts are removed at the expense of slightly lower error reduction, but the output mesh is visually comparable to the Delaunay mesh, with reduced error. This can be seen by comparing the shaded view of the terrains in Figures 5.10 and 5.12.

For the constrained DDT terrains, I used the Look-Ahead algorithm (Algorithm 4.6) along with my new cost function CF7 (Algorithm 4.4), as this combination generally gives the best results (Figures 5.13-5.16).

5.6.3 Performance comparison of different methods

Given below are the tables comparing different cost functions and algorithms for constrained DDT optimization. Figures 5.13-5.15 show the performance comparison for a minimum angle input of 0° :

INPUT (max error)			Output (RMS error)									
	#verts	#faces	Delaunay	DDT				Look-Ahead				% Reduction (error)
				CF1	CF5	CF6	CF7	CF1	CF5	CF6	CF7	
4	32	51	3.385	3.772	3.655	3.23	3.166	3.25	3.397	3.217	3.24	6.470
3	386	757	1.154	1.272	1.229	1.422	0.95	1.272	1.35	1.645	0.954	17.678
2	1122	2218	0.554	0.504	0.543	0.431	0.466	0.551	0.55	0.449	0.374	32.491
1.2	2692	5357	0.109	0.107	0.11	0.088	0.073	0.106	0.115	0.09	0.071	34.862
1.01	3750	7468	0.068	0.699	0.073	0.058	0.048	0.0698	0.077	0.058	0.046	32.353
0.7	6471	12906	0.037	0.034	0.035	0.031	0.026	0.034	0.036	0.032	0.025	32.432
CF1 = Angle Between Normals												
CF5 = YU cost function												
CF6 = My cost function (based on max error)												
CF7 = My cost function (based on rms error)												

Figure 5.13: Table showing error reduction with DDT, 0° minimum angle

INPUT (max error)	Output (verts, faces, RMS error)								
	Delaunay			DDT Look-Ahead (CF7, 0°)			Simplification algorithm		
	#verts	#faces	error	#verts	#faces	error	#verts	#faces	% Reduction (faces)
4	32	51	3.385	32	51	3.24	28	43	15.686
3	386	757	1.154	386	757	0.954	307	599	20.872
2	1122	2218	0.554	1122	2218	0.374	591	1155	47.926
1.2	2692	5357	0.109	2692	5357	0.071	1885	3743	30.129
1.01	3750	7468	0.068	3750	7468	0.046	2726	5420	27.424
0.7	6471	12906	0.037	6471	12906	0.025	4575	9114	29.382

Figure 5.14: Table showing triangle count reduction with DDT, 0° minimum angle

INPUT (max error)	Time taken (ms)		
	Delaunay	DDT Look-Ahead (CF7, 0°)	Simplification algorithm
4	244	2348	138
3	252	2089	350
2	209	2763	778
1.2	343	4205	18456
1.01	438	8039	68359
0.7	830	11640	425586

Figure 5.15: Table showing execution times with DDT, 0° minimum angle

Figures 5.16-5.18 show the performance comparison for a minimum angle input of 20° :

INPUT (max error)	Output (RMS error)										
	#verts	#faces	Delaunay	DDT Lawson 20°				DDT Look-Ahead 20°			
				CF1	CF5	CF6	CF7	CF1	CF5	CF6	CF7
4	32	51	3.385	3.343	3.348	3.316	3.295	3.334	3.339	3.323	3.294
3	386	757	1.154	1.106	1.138	1.112	1.052	1.1	1.148	1.106	1.051
2	1122	2218	0.554	0.458	0.563	0.456	0.517	0.499	0.482	0.454	0.443
1.2	2692	5357	0.109	0.105	0.107	0.097	0.095	0.104	0.106	0.097	0.093
1.01	3750	7468	0.068	0.067	0.07	0.062	0.059	0.068	0.07	0.061	0.057
0.7	6471	12906	0.037	0.035	0.036	0.035	0.033	0.035	0.037	0.035	0.032
CF1 = Angle Between Normals											
CF5 = YU cost function											
CF6 = My cost function (based on max error)											
CF7 = My cost function (based on rms error)											

Figure 5.16: Table showing error reduction with DDT, 20° minimum angle

INPUT (max error)	Output (verts, faces, RMS error)									
	Delaunay			DDT	Look-Ahead (CF7, 20°)	Simplification algorithm			% Reduction (faces)	
	#verts	#faces	error	#verts	#faces	error	#verts	#faces		error
4	32	51	3.385	32	51	3.294	26	40	3.371	21.569
3	386	757	1.154	386	757	1.051	275	535	1.145	29.326
2	1122	2218	0.554	1122	2218	0.443	714	1400	0.5538	36.880
1.2	2692	5357	0.109	2692	5357	0.093	2099	4171	0.1088	22.139
1.01	3750	7468	0.068	3750	7468	0.057	3012	5992	0.067	19.764
0.7	6471	12906	0.037	6471	12906	0.032	4691	9347	0.03696	27.576

Figure 5.17: Table showing triangle count reduction with DDT, 20° minimum angle

INPUT (max error)	Time taken (ms)		
	Delaunay	DDT Look-Ahead (CF7, 20°)	Simplification algorithm
4	244	254	134
3	252	240	255
2	209	297	537
1.2	343	1048	11935
1.01	438	1731	28032
0.7	830	5376	228176

Figure 5.18: Table showing execution times with DDT, 20° minimum angle

In Figures 5.13 and 5.16, the highlighted field in each row indicates the algorithm + cost function combination with the lowest error. I tested both Lawson’s and Look-Ahead algorithms with all the 7 cost functions, but shown only the 4 best here due to lack of space.

Overall, my proposed cost function based on mean square error, CF7, performs best. In fact, all the previously known cost functions, CF1-CF5 (Section 3.2.1) return poor results, some worse than the input Delaunay, since they were not proposed with terrain optimization in mind. My proposed CF6 cost function does not perform as well as CF7 since it is based on maximum error, and I am comparing mean square errors here, which is undoubtedly a better estimate for terrain error.

I also noticed that as the triangle count increases (terrain becomes denser), Look-Ahead algorithm gives better results than Lawson's algorithm, which validates its authors' claim [47]. This can also be understood by reasoning as well, as checking for more edges around a given edge gives a better estimate when the neighboring edges are closer together. Indeed, this also explains why Lawson's algorithm is better than Look-Ahead in the first two cases (checking more edges is less accurate than checking fewer edges since the edges are not close together).

My simplification algorithm has reduced triangle counts from 15% to 48% (Figures 5.14. 5.17), while still maintaining the error level below that of the Delaunay mesh. The reduction of triangle and vertices count does not mean that the quality of the terrain has reduced. In fact, it is even better than the original terrain (as the error is lower), as it mathematically represents the input heightmap more closely.

Figures 5.15 and 5.18 show that the execution times of DDT are significantly higher than Delaunay. This is because Delaunay algorithm just inserts and removes vertices, whereas DDT optimizes the mesh based on edge costs, which needs to be computed for every triangle on a pixel-by-pixel basis. The 0° test cases are comparatively slower than 20° test cases, since the latter eliminates many faces due to the angle constraint, hence processing fewer edges.

The simplification algorithm greedily removes edges, and each removal re-computes the costs of all the affected surrounding edges. Since this is done for the whole mesh (and not just a part of it, as in dynamic algorithms), the increased processing time is to be expected. Since my algorithms are offline and the focus is on quality more than speed, as per the game company's requirements, this is entirely acceptable.

5.6.4 Application in retail game

My terrain generation plugin (Figure 5.4) has been used to create large, beautiful worlds in an upcoming AAA title “Baja: Edge of Control” [5], for the Xbox 360 and PS3 consoles. Since the game is scheduled to be released only in September 2008, I cannot post images of all the terrains used in it due to copyright issues. A few screenshots which were publicly released are shown below:

Figure 5.17: A large open world in “Baja: Edge of Control”

Figure 5.18: Detailed terrain region with lakes, cliffs and valleys

Figure 5.19: Details clearly seen even in far-away regions of the terrain

Figures 5.17-5.19 only show one world out of the 10 in the game. To make the terrain look beautiful, there are a lot of additional things to be done: Creating multiple textures, programming shaders for their effective blending, generating *normalmaps*, *lightmaps* and *shadowmaps* for accurate rendering, adding terrain objects like shrubs, ecosystem, banners, guardrails, water body etc. for realism, adding support for physics collision detection and response, and many more such things. However, all this is not possible without an efficient underlying terrain mesh, which I have created.

Chapter 6

Terrain Modification

Once the Delaunay triangulated terrain is generated, artists or developers may need to modify it to overlay patches, tracks etc. over it. This chapter describes three methods for such modification. Some terminology: there are two triangular meshes, the destination mesh (D_M) and the source mesh (S_M), and my objective is to overlay S_M over D_M , at any specified position and orientation.

6.1 Stenciling

In *stenciling*, priority is given to the geometry of S_M when overlaying it on D_M . The amount of priority is determined by an error parameter. To illustrate, shown below is a simple example:

Figure 6.1: Before (left) and after (right) stenciling a plane over a bigger plane

In Figure 6.1 (left), S_M is the smaller 2×2 planar grid and D_M is the larger one, with S_M positioned such that it cuts across two edges of D_M . Figure 6.1 (right) shows a single stenciled mesh in which:

- All the edges and vertices of S_M are projected and retained
- No new vertices are introduced
- The edges of D_M which intersect with any edges of S_M are removed
- The region around the stenciled part is retriangulated

It is noteworthy that S_M is projected over D_M “as-is”. However, this has disadvantages when S_M is a track and D_M is a terrain, because height information pertaining to D_M will be lost when vertices of D_M inside the projected region are removed. Therefore, an error parameter is introduced, which controls how much height difference error is allowed over the stenciled region. This allows flexibility to the artists who can make the track as detailed as they want.

The stenciling algorithm is described below:

- 1: Create a new mesh M , initialize it to empty
- 2: Create a new PSLG X , initialize it to D_M
- 3: Generate heightmap H from S_M
- 4: Add all projected S_M vertices lying within D_M , V_S to X
- 5: **If** S_M extends beyond D_M
- 6: Calculate points of intersection V_B of projected S_M edges with bounding edges of D_M
- 7: Add set of vertices V_B to X

```

8:   End If
9:   Add all full projected  $S_M$  edges, excluding those which lie wholly or partially
      outside  $D_M$ , to  $X$ 
10:  Remove all edges of  $D_M$  which intersect with projected  $S_M$  edges, from  $X$ 
11:  Remove all vertices  $D_M$  of lying within projected  $S_M$  region
12:  Triangulate  $X$ 
13:      Do  $CCDT(X)$ , with the following changes (to optimize performance):
14:      Perform area, angle and height constraint checks only for newly formed
          triangles lying within projected  $S_M$  region
15:      Set  $M = CCDT(X)$ 
16:  End Triangulate
17:  For each vertex  $v \in M$ 
18:      If  $v \in V_B$  or  $v \in V_S$ 
19:          Get  $h = \text{height of pixel } (\lfloor v_x \rfloor, \lfloor v_y \rfloor)$  from heightmap  $H$ 
20:          Update  $M$  : set  $v_z = h$ 
21:      End If
22:  End For
23:  Get output stenciled mesh as  $M$ 

```

Algorithm 6.1: Stenciling algorithm

To generate the heightmap in step 3, I scan through all the pixels in each triangle and use interpolation to get their height, from the height information at their vertices. For the line segment intersection tests in steps 6 and 10, I use simple orientation tests to check if they intersect and solve for the point using determinants if they do. In order to check constraints in step 14, I use parameters which can be set from the UI before stenciling, similar to terrain generation.

6.2 Stitching

Stitching is the error-free and more intuitive way of overlaying S_M over D_M . True to the word, I simply “stitch” the two meshes together, while following the height of D_M . To illustrate, the same two input planes as in Figure 6.1 are used.

Figure 6.2: Before (left) and after (right) stitching a plane over a bigger plane

As compared to stenciling, on Figure 6.2 (right) shows a mesh in which:

- All the projected edges of S_M are either retained or split
- All the edges of D_M are also retained or split
- New vertices are created wherever D_M and S_M intersect
- The region around the stitched part is retriangulated

The advantage of stitching is that no “new” vertices are introduced by the stitching algorithm except for the newly created intersection points. This implies that the height difference error is always zero, and a perfect stitched mesh is obtained. It also means that there is no longer any need for user parameters like angle, area and error constraints, since no implicit refinement of the mesh takes place.

The stitching algorithm is outlined below:


```

1:   Create a new mesh  $M$ , initialize it to empty
2:   Create a new PSLG  $X$ , initialize it to empty
3:   Add all  $D_M$  vertices,  $V_D$ , to  $X$ 
4:   Add all non-duplicated projected  $S_M$  vertices lying within  $D_M$ ,  $V_S$ , to  $X$ 
5:   If  $S_M$  extends beyond  $D_M$ 
6:       Calculate points of intersection  $V_B$  of projected  $S_M$  edges with bounding
           edges of  $D_M$ 
7:       Add set of vertices  $V_B$  to  $X$ 
8:   End If
9:   Add new vertices creates due to intersection of every  $S_M$  edge with every  $D_M$ 
       edge,  $V_N$ , to  $X$ 
10:  Add all unaffected edges of  $D_M$  (i.e. not intersecting with any projected  $S_M$ 
       edge) to  $X$ 
11:  Add all unaffected edges of  $S_M$  (i.e. not intersecting with any  $D_M$  edge), lying
       within  $D_M$ , to  $X$ 
12:  Add newly formed edges formed by splitting  $D_M$  edges to  $X$ 
13:  Add newly formed edges formed by splitting projected  $S_M$  edges to  $X$ 
14:  Triangulate  $X$  with the CDT algorithm, set  $M = CDT(X)$ 
15:  For each vertex  $v \in M$ 
16:      If  $v \in V_D$ 
17:          Get height  $h$  of point  $(v_x, v_y)$  directly from  $D_M$ 
18:          Update  $M$ : set  $v_z = h$ 
19:      Else If  $v \in V_S$  or  $v \in V_N$  or  $v \in V_B$ 

```


20:	Calculate height h by point inside triangle tests and interpolation
21:	Update M : set $v_z = h$
22:	End If
21:	End For
22:	Get output stitched mesh as M

Algorithm 6.2: Stitching algorithm

For calculating the height of an arbitrary point on the stitch region in step 20, I simply calculate the equation of the plane containing the triangle and use that to find the height of the desired point. If projected vertex $v(v_x, v_y, v_z)$ lies inside triangle T , with vertices (x_1, y_1, z_1) , (x_2, y_2, z_2) and (x_3, y_3, z_3) , then v_z is calculated by solving the determinant:

$$\begin{vmatrix} v_x - x_1 & v_y - y_1 & v_z - z_1 \\ x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \end{vmatrix} = 0$$

where v_z is the only unknown.

6.3 Track Overlaying

Track Overlaying is a specific algorithm for overlaying tracks onto a terrain, using 3 different methods for height adjustment. This is important because stitching and stenciling, though providing unique triangulations with their own characteristics, do not allow for easy change of height in the overlay region.

This is especially required for tracks, where the track region must have a uniform triangulation which follows the terrain, but at the same time remains reasonably flat so that vehicles can drive over it smoothly without bumps or irregularities.

I have proposed 3 methods for adjusting height in the overlay region:

- (i) Following terrain height
- (ii) Following track height
- (iii) Following blended height

Method (i) is similar to stenciling, without the addition of any new points inside the overlay area. Method (ii) ensures the track region to be smoothest, but at the cost of some bumps on the boundary vertices connecting the track to the terrain. Method (iii) is optimal: it follows the terrain at the edges, track in the middle, and blends between the two in the in-between regions. The blending is done by simple linear interpolation of the two heights, based on distance from the two edges of the track.

The algorithm is formally stated below:

- 1: Create a new mesh M , initialize it to empty
- 2: Create a new PSLG X , initialize it to D_M
- 3: Generate heightmap H from S_M
- 4: Add all projected S_M vertices lying within D_M , V_S to X
- 5: **If** S_M extends beyond D_M
 - 6: Calculate points of intersection V_B of projected S_M edges with bounding edges of D_M
 - 7: Add set of vertices V_B to X
- 8: **End If**
- 9: Add all full projected S_M edges, excluding those which lie wholly or partially outside D_M , to X
- 10: Remove all edges of D_M which intersect with projected S_M edges, from X

```

11:  Remove all vertices  $D_M$  of lying within projected  $S_M$  region
12:  Triangulate  $X$ 
13:      Set  $M = CDT(X)$ 
14:  End Triangulate
15:  If overlay method is (i)
16:      For each vertex  $v \in M$ 
17:          If  $v \in V_S$ 
18:              Get  $h =$  height of pixel  $(\lfloor v_X \rfloor, \lfloor v_Y \rfloor)$  from heightmap  $H$ 
19:              Update  $M$  : set  $v_z = h$ 
20:          End If
21:      End For
22:  Else If overlay method is (ii)
23:      For each vertex  $v \in M$ 
24:          If  $v \in V_S$ 
25:              Get  $h =$  height of  $(v_X, v_Y)$  from  $V_S$ 
26:              Update  $M$  : set  $v_z = h$ 
27:          End If
28:      End For
29:  Else If overlay method is (iii)
30:      For each vertex  $v \in M$ 
31:          If  $v \in V_S$ 
32:              Get  $h_1 =$  height of pixel  $(\lfloor v_X \rfloor, \lfloor v_Y \rfloor)$  from heightmap  $H$ 
33:              Get  $h_2 =$  height of  $(v_X, v_Y)$  from  $V_S$ 
34:              Update  $M$  : set  $v_z$  as linear interpolation of  $h_1$  and  $h_2$ 

```

		based on distance of (v_x, v_y) from track edges
35:		End If
36:		End For
37:	End If	
38:	Get output track overlaid mesh as M	

Algorithm 6.3: Track Overlaying algorithm

6.4 Results

I implemented the terrain modification algorithms also as 3dsMax plugins, whose screenshots are shown in Figure 6.3. Stenciling and Stitching are combined into one plugin (Figure 6.3 left), whereas Track Overlaying was implemented as a separate plugin (Figure 6.3 right) to prevent confusion due to too many options.

6.4.1 Stenciling and Stitching

In Figure 6.3 (left), the three user constraints for minimum area, minimum error and angle are only used for stenciling (when “Stitch Source Object” is unchecked). There is also an option to center S_M over D_M . “Irregular Stitching” is enabled only when “Stitch Source Object” is checked, and allows S_M to extend beyond D_M without getting cut (this means no need for steps 5-8 in Algorithm 6.2, but there may be additional post processing). There is also the option to automatically do mapping of UV co-ordinates, which can otherwise be a hassle for artists.

Figure 6.3: User interfaces of terrain modification plugins

Figure 6.4: Input terrain and track meshes

Figure 6.5: Effect of stenciling (left) and stitching (right) track over terrain

To test stenciling and stitching, I conducted many test cases with terrains and tracks of varying sizes. Figure 6.4 shows the track object positioned over a terrain, at the exact orientation in which it needs to be applied. Figure 6.5 shows the effect of stenciling and stitching of the track over this large terrain. Figure 6.5 (left) shows that in stenciling, as few vertices are added “inside” the projected source mesh, the exact number controlled by the parameters in Figure 6.3. From Figure 6.5 (right), in stitching, all the points of intersection are added, and hence there is no loss of height information in the final mesh. In other words, the track mesh perfectly follows the terrain and its height.

6.4.2 Track overlaying

For track overlaying, there are three choices, described in Section 6.3, and the option to project UVs as well. There is an additional post-processing option of flattening the track (“Flatten Track” button), which makes use of a simple idea: if the range of height values in the overlay regions varies too much within a specified range (hard-coded now), then set them to the same height. This simple idea has scope for improvement, so I am not discussing it further.

To test track overlaying, I created a much more detailed terrain and track than the previous example. The outputs show that the triangulation inside the overlay region is perfectly preserved, with the height adjusted in such a way that the overlay region is very smooth, as shown below in Figures 6.6-6.11:

Figure 6.6: Input terrain with a dense control region

Figure 6.7: Detailed track mesh before overlay

Figure 6.8: Terrain overlaid with track, wireframe view

Figure 6.9: Terrain overlaid with track, shaded edge view

Figure 6.10: Terrain overlaid with track, fully shaded view

Figure 6.11: Zoomed-in top view, showing perfect preservation of triangles

Comparing Figures 6.6 and 6.8, it can be seen that before track overlaying, in spite of more triangles around the track region, the terrain is still not smooth and has many height variations. But in Figure 6.8, all such bumps and undulations are removed and the track region is overlaid smoothly over the terrain. I have used the “follow track” method for this example. From Figure 6.11, it can be clearly seen why this method is superior compared to stenciling or stitching, when it comes to overlaying tracks. Due to the uniform retriangulation, it is easy to project UVs, perform shading, blending etc., in addition to the overall mesh smoothness in the overlay region.

6.4.3 Application in retail game

The application of my generated terrain in “Baja: Edge of Control” [5] was seen in Section 5.6.4. Similarly, the “stitching” and “track overlaying” algorithms are also used in the game to overlay beautiful looking tracks over the terrain. Some samples are shown below:

Figure 6.12: A flat looking detailed track over the terrain

Figure 6.13: Smooth track region in spite of bumps in the terrain

As I mentioned earlier, modifying the mesh to make the track region smooth and even is only one part. The second part is to use shaders to make it look beautiful by applying multiple textures, *detailmaps*, blending with terrain texture etc. My plugin (Figure 6.3) supports automatic UV mapping using DirectX Shader material in 3dsMax, and adjusts vertex colors for smooth blending. However, programming the shaders is beyond the scope of this project, and was done by the game programmers at 2XL, Inc. I have provided them the means to easily use and integrate the modified terrain into the game.

Chapter 7

Conclusion

In this thesis, an efficient method for terrain generation was introduced, based on the Constrained Conforming Delaunay triangulation, using a heightmap, control map and several constraints, which act as control parameters. The concept of a control map was something new, allowing different regions in the terrain to be generated at varying detail.

By using a heightmap together with a control map, generating a terrain using a top-down approach was automated to a large extent. Enhancements to the mesh were done via user parameters specifying angle, area and height constraints. The nature of the Delaunay triangulation algorithm further guaranteed the maximization of the minimum angle of any triangle, hence producing a well-graded, well-rounded, and more geometrically balanced unstructured terrain.

As a post-process to the Delaunay-based terrain, a Constrained DDT algorithm was proposed, in order to reduce terrain error, measured on a pixel-by-pixel basis using the L_2 norm. It makes use of two new DDT cost functions and an optimization procedure based on Look-Ahead algorithm. Finally, a Terrain Simplification algorithm was also proposed, to reduce the triangle count. These optimizations can also be done only on specific parts

of the terrain such as cliffs, valleys and ridges, where sharp features need to be preserved, using a control map.

Three methods were proposed for modifying the terrain: Stenciling, Stitching and Track Overlaying, to add objects such as tracks and guardrails over the terrain. In Stenciling, the geometry of the source mesh was retained as much as possible, using the CCDT algorithm for retriangulation. In Stitching, the source mesh was made to follow the destination mesh perfectly by calculating all points of intersection, using the CDT algorithm for retriangulation. In Track Overlaying, which also uses CDT, users have 3 options to overlay flat-looking tracks over the terrain, without introducing uneven jumps or cracks. The overlay region is smoother than in previous approaches.

As future work, there are some ways for enhancing the Terrain Simplification algorithm (Algorithm 4.7). Although results show that it reduces triangle count greatly while maintaining the same error level as a Delaunay-based mesh, it may not result in the global minimum. This is because it basically follows a greedy approach, which is generally notorious for *not* reaching the global minimum. For this purpose, one can try combining the proposed algorithm with a simulated annealing approach [73].

References

- [1] Wikipedia, "Video game," URL: http://en.wikipedia.org/wiki/Video_game, 2008.
- [2] Nintendo Company Ltd., "Super Mario Bros.," URL: http://en.wikipedia.org/wiki/Super_Mario_Bros., 1985.
- [3] Nintendo Company Ltd., "Nintendo Entertainment System (NES)," URL: http://en.wikipedia.org/wiki/Nintendo_Entertainment_System, 1983.
- [4] The Codemasters Software Company Ltd., "Colin McRae DiRT," URL: <http://www.codemasters.co.uk/dirt/index2.php>, 2007.
- [5] 2XL Games Inc., "Baja: Edge of Control," URL: <http://www.bajagame.com/>, 2008.
- [6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, 1965.
- [7] Longbow Digital Arts Incorporated, "Tread Marks," URL: <http://www.ldagames.com/treadmarks/>, 1999.
- [8] The Codemasters Software Company Ltd., "Colin McRae Rally 04," URL: <http://www.codemasters.com/games/?gameid=1361>, 2004.
- [9] Sony Computer Entertainment Europe, "Motorstorm," URL: <http://www.motorstorm.com/>, 2006.
- [10] I. Babuska and A. K. Aziz, "On the Angle Condition in the Finite Element Method," *SIAM Journal on Numerical Analysis*, vol. 13, pp. 214-226, 1976.
- [11] J. F. Thompson and N. P. Weatherill, "Aspects of numerical grid generation: current science and art," *AIAA Paper*, pp. 93-3539, 1993.

- [12] M. Bern and D. Eppstein, "Mesh generation and optimal triangulation, Computing in Euclidean Geometry, (D.-Z. Du and F. Hwang, eds.)," *World Scientific, Singapore*, 1995.
- [13] C. James, D. A. Cavendish, and H. William, "An Approach to Automatic Three-Dimensional Finite Element Mesh Generation," *International Journal for Numerical Methods in Engineering*, vol. 21, pp. 329-347, February 1985.
- [14] W. H. Frey, "Selective Refinement: a New Strategy For Automatic Node Placement in Graded Triangular Meshes," *International Journal for Numerical Methods in Engineering*, vol. 24, pp. 2183-2200, 1987.
- [15] M. H. Overmars and J. Leeuwen, "Maintenance of Configurations in the Plane," *JCSS*, vol. 23, pp. 166-204, 1981.
- [16] R. Graham, "An efficient algorithm for determining the convex hull of a finite point set," *Information Processing Letters*, vol. 1, pp. 132-133, 1972.
- [17] W. F. Eddy, "A New Convex Hull Algorithm for Planar Sets," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, pp. 398-403, 1977.
- [18] F. P. Preparata and M. I. Shamos, *Computational geometry: an introduction, Chap. 3. Convex Hulls: Basic Algorithms*: Springer-Verlag New York, Inc. New York, NY, USA, 1985.
- [19] A. M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions," *Information Processing Letters*, vol. 9, pp. 216-219, 1979.
- [20] M. Kallay, "The complexity of incremental convex hull algorithms in R^d ," *Information processing letters*, vol. 19, 1984.
- [21] D. G. Kirkpatrick and R. Seidel, "The Ultimate Planar Convex Hull Algorithm?," 1983.
- [22] K. L. Clarkson and P. W. Shor, "Applications of random sampling in computational geometry, II," *Discrete and Computational Geometry*, vol. 4, pp. 387-421, 1989.
- [23] N. Boris, "Delaunay," *Sur la Sphère Vide. Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk*, vol. 7, pp. 793-800, 1934.

- [24] C. L. Lawson, "Software for C 1 surface interpolation," *Mathematical Software*, vol. 3, pp. 161–194, 1977.
- [25] E. F. D'Azevedo and R. B. Simpson, "On Optimal Interpolation Triangle Incidences," *SIAM Journal on Scientific and Statistical Computing*, vol. 10, pp. 1063, 1989.
- [26] J. R. Shewchuk, "Delaunay Refinement Mesh Generation," School of Computer Science, Carnegie Mellon University, 1997.
- [27] P. Su and R. L. Scot Drysdale, "A comparison of sequential Delaunay triangulation algorithms," *Computational Geometry: Theory and Applications*, vol. 7, pp. 361-385, 1997.
- [28] P. Su, "Efficient Parallel Algorithms for Closest Point Problems," *PhD Thesis, Dartmouth College, Hanover, New Hampshire*, May 1994.
- [29] M. I. Shamos and D. Hoey, "Closest-point problems," *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci*, pp. 151-162, 1975.
- [30] D. T. Lee and B. J. Schachter, "Two algorithms for constructing a Delaunay triangulation," *International Journal of Parallel Programming*, vol. 9, pp. 219-242, 1980.
- [31] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi," *ACM Transactions on Graphics (TOG)*, vol. 4, pp. 74-123, 1985.
- [32] R. A. Dwyer, "A faster divide-and-conquer algorithm for constructing delaunay triangulations," *Algorithmica*, vol. 2, pp. 137-151, 1987.
- [33] L. Paul Chew, "Constrained delaunay triangulations," *Algorithmica*, vol. 4, pp. 97-108, 1989.
- [34] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, pp. 153-174, 1987.
- [35] H. Edelsbrunner and R. Seidel, "Voronoi diagrams and arrangements," *Discrete and Computational Geometry*, vol. 1, pp. 25-44, 1986.
- [36] K. Q. Brown, "Voronoi Diagrams from Convex Hulls," *Information Processing Letters*, vol. 9, pp. 223-228, 1979.

- [37] R. Seidel, "Backwards Analysis of Randomized Geometric Algorithms," *International Computer Science Institute*, 1992.
- [38] G. H. Meister, "Polygons Have Ears," *The American Mathematical Monthly*, vol. 82, pp. 648-651, 1975.
- [39] M. V. Anglada, "An improved incremental algorithm for constructing restricted Delaunay triangulations," *Computers & Graphics*, vol. 21, pp. 215-223, 1997.
- [40] J. R. Shewchuk, "Theoretically Guaranteed Mesh Generation," *Fourteenth International Meshing Roundtable*, September 2005.
- [41] H. ElGindy, H. Everett, and G. Toussaint, "Slicing an ear using prune-and-search," *Pattern Recognition Letters*, vol. 14, pp. 719-722, 1993.
- [42] J. Steiner, "Questions proposées. Théorèmes sur l'hexagramum mysticum. Annales de Gergonne, 18 (1827-1828), p. 339-340."
- [43] L. P. Chew, "Guaranteed-Quality Triangular Meshes," *Technical Report TR-89-983, Department of Computer Science, Cornell University*, 1989.
- [44] L. P. Chew, "Guaranteed-quality mesh generation for curved surfaces," *Proceedings of the ninth annual symposium on Computational geometry*, pp. 274-280, 1993.
- [45] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," *Journal of Algorithms*, vol. 18, pp. 548-585, 1995.
- [46] N. Dyn, D. Levin, and S. Rippa, "Data Dependent Triangulations for Piecewise Linear Interpolation," *IMA Journal of Numerical Analysis*, vol. 10, pp. 137-154, 1990.
- [47] X. Yu, B. S. Morse, and T. W. Sederberg, "Image Reconstruction Using Data-Dependent Triangulation," *IEEE Computer Graphics and Applications*, vol. 21, pp. 62-68, 2001.
- [48] T. K. Peucker, R. J. Fowler, J. J. Little, and D. M. Mark, "The triangulated irregular network," *Amer. Soc. Photogrammetry Proc. Digital Terrain Models Symposium*, pp. 516-532, 1978.

- [49] R. J. Fowler and J. J. Little, "Automatic extraction of Irregular Network digital terrain models," *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, pp. 199-207, 1979.
- [50] 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*: ACM Press New York, NY, USA, 1996.
- [51] P. Lindstrom and V. Pascucci, "Visualization of large terrains made easy," *Proceedings of the conference on Visualization'01*, pp. 363-371, 2001.
- [52] M. Duchaineau, M. Wolinsky, D. E. Sigiety, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time Optimally Adapting Meshes," *Visualization'97., Proceedings*, pp. 81-88, 1997.
- [53] H. Hoppe, "Progressive meshes," *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 99-108, 1996.
- [54] H. Hoppe, "View-dependent refinement of progressive meshes," *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 189-198, 1997.
- [55] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," *IEEE Visualization*, pp. 35-42, 1998.
- [56] W. H. de Boer, "Fast Terrain Rendering Using Geometrical MipMapping," E-mersion Project, URL: http://www.flipcode.com/archives/article_geomipmaps.pdf, 2000.
- [57] F. Losasso and H. Hoppe, "Geometry clipmaps: terrain rendering using nested regular grids," *International Conference on Computer Graphics and Interactive Techniques*, pp. 769-776, 2004.
- [58] A. Mirante and N. Weingarten, "The Radial Sweep Algorithm for Constructing Triangulated Irregular Networks," *Computer Graphics and Applications, IEEE*, vol. 2, pp. 11-21, 1982.
- [59] C. T. Silva, J. S. B. Mitchell, and A. E. Kaufman, "Automatic generation of triangular irregular networks using greedycuts," *Visualization, 1995. Visualization'95. Proceedings., IEEE Conference on*, pp. 201-208, 1995.

- [60] M. J. Garland and P. S. Heckbert, *Fast Polygonal Approximation of Terrains and Height Fields*: School of Computer Science, Carnegie Mellon University, 1995.
- [61] H. Pedrini, "An improved refinement and decimation method for adaptive terrain surface approximation," *Proceedings of the 9th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pp. 103–109.
- [62] B. Baumgart, "Winged Edge Polyhedron Representation, AIM-79," *Stanford Univ., Report STAN-CS-320*, 1972.
- [63] P. S. Heckbert, *Graphics Gems IV*: Morgan Kaufmann, 1994.
- [64] P. Heckbert, "Very Brief Note on Point Location in Triangulations," URL: http://www.cs.cmu.edu/~ph/walking_loop.ps, 1994.
- [65] P. J. Green and R. Sibson, "Computing Dirichlet Tessellations in the Plane," *The Computer Journal*, vol. 21, pp. 168–173, 1978.
- [66] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry: Theory and Applications*, vol. 22, pp. 21–74, 2002.
- [67] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148, pp. 203–222, 1996.
- [68] J. R. Shewchuk, "Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator," URL: <http://www.cs.cmu.edu/~quake/triangle.html>, 2005.
- [69] G. L. Miller, S. E. Pav, and N. J. Walkington, "When and why Ruppert's algorithm works," *Proc. 12th Int. Meshing Roundtable*, pp. 91–102, 2003.
- [70] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of triangle meshes," *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, vol. 1992, pp. 65–70, 1992.
- [71] M. Garland and P. S. Heckbert, "Surface simplification using quadric error metrics," *Proceedings of SIGGRAPH*, pp. 209–216, 1997.
- [72] D. P. Luebke, "A Developer's Survey of Polygonal Simplification Algorithms," *IEEE Computer Graphics and Applications*, pp. 24–35, 2001.

- [73] L. L. Schumaker, "Computing optimal triangulations using simulated annealing," *Computer Aided Geometric Design*, vol. 10, pp. 329-345, 1993.

Appendix

Publications

- [1] S. Raman and Z. Jianmin, "Efficient Terrain Triangulation and Modification Algorithms for Game Applications," *International Journal of Computer Games Technology*, vol. 2008, Article ID 316790, 5 pages, March 2008. doi:10.1155/2008/316790
- [2] S. Raman and Z. Jianmin, "Efficient Terrain Triangulation and Merging of World Objects for Game Applications," *Proceedings of Cybergames 2007*, pp. 46-51, September 2007.
- [3] S. Raman, "Efficient Terrain Generation and Modification for Game Applications," *Gamescape 2007 Seminar*, gameLab, NTU, December 2007.