University of Alberta

Library Release Form

Name of Author: Kai Chen

Title of Thesis: Robust Dynamic Constrained Delaunay Triangulation For Pathfinding

Degree: Master of Science

Year this Degree Granted: 2009

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Kai Chen
Apt 901, 9747 - 104 street,
Edmonton, Alberta
Canada, T5K 0Y6

Date: ____________________

Wisdom is knowing what to do next;
virtue is doing it.

University of Alberta

Robust Dynamic Constrained Delaunay Triangulation For Pathfinding

by

Kai Chen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2009

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Robust Dynamic Constrained Delaunay Triangulation For Pathfinding** submitted by Kai Chen in partial fulfillment of the requirements for the degree of **Master of Science**.

____

Dr. Michael Buro

____

Dr. Herbert Yang

____

Dr. Farbod Fahimi

Date: ______________

To my parents

Abstract

Surface triangulation has many applications such as computer graphics, navigation, and AI in video games. For this thesis we have developed a Free Software C++ library for dynamic constrained Delaunay triangulation which handles point and line segment insertions and removals efficiently and robustly. Geometric computations are based on arbitrary-precision arithmetic that avoids problems caused by round-off errors. We describe the theoretical foundations of the underlying algorithms, present performance evaluations, and describe the public interface of the software package.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Michael Buro, for his encouragement, support, patience and guidance. His technical and theoretical insights laid the foundation of my work and guided me throughout my research. I will definitely miss the weekly meetings with him. Throughout my writing period, he provided extensive input to my thesis. I would have been lost without him. I would also like to thank Dr Herb Yang and Dr Farbod Fahimi for reviewing my thesis.

I am grateful to Douglas Demyen for his efforts to explain the pathfinding system in early stage of my research.

I wish to thank the department of Computing Science at University of Alberta for providing a wonderful research environment.

My graduate school years cannot be truly memorable without my friends. Thank you all. Especially, I wish to thank Zhifu Zhang, Yilei Zhang for their advice and encouragement, and Alex Khanh for his timely distractions far away in Toronto.

Lastly, I would like to thank my parents, my girlfriend Lu Zhang, and the rest of my family for their support and understanding.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Previous Work . . . 2
- 1.3 Contributions . . . 3
- 1.4 Thesis Outline . . . 4

2 Properties of Triangulations **5**
- 2.1 Planar Graphs . . . 5
- 2.2 Triangulations . . . 6
  - 2.2.1 Basic Routines for Triangulation . . . 7
- 2.3 Delaunay Triangulations . . . 8
- 2.4 Constrained Delaunay Triangulations . . . 10
  - 2.4.1 Basic Routines for constrained Delaunay Triangulation . . . 12
- 2.5 Dynamic Constrained Delaunay Triangulation . . . 13

3 Computing Dynamic Constrained Delaunay Triangulations **14**
- 3.1 A Survey of Basic Geometric Data Structures . . . 14
  - 3.1.1 Element Lists and Adjacency Lists . . . 14
  - 3.1.2 Winged Edges . . . 15
  - 3.1.3 Half Edges . . . 16
  - 3.1.4 Quad Edge . . . 17
  - 3.1.5 Selected data structure . . . 19
- 3.2 Point Insertion in Delaunay triangulation . . . 21
  - 3.2.1 Point Localization . . . 22
  - 3.2.2 Point Insertion . . . 29
  - 3.2.3 Restoring the Delaunay Property . . . 31
- 3.3 Constrained Edge Insertion . . . 32
- 3.4 Vertex Removal . . . 36
- 3.5 Constrained Polygon Insertion and Removal . . . 37
- 3.6 Conclusion . . . 39

4 Robust Geometric Computations **40**
- 4.1 Fundamental Geometric Functions . . . 41
  - 4.1.1 The Orientation Function . . . 41
  - 4.1.2 The InCircle Function . . . 42
  - 4.1.3 The Intersection Function . . . 42
- 4.2 Floating Point Rounding Errors . . . 44
- 4.3 Exact Geometric Computation . . . 47
  - 4.3.1 Arbitrary-Precision Libraries . . . 47
  - 4.3.2 Interval Arithmetic . . . 48

4.3.3 Extended Algorithm . . . 51
4.3.4 Conclusion . . . 53

5 Experiments **54**
5.1 Experiment 1: An Example of InCircle Computation . . . 55
5.1.1 Experiment Setup . . . 56
5.1.2 Experimental Results . . . 56
5.2 Experiment 2: Point Localization Algorithms . . . 57
5.2.1 Experiment Setup . . . 58
5.2.2 Experimental Results . . . 58
5.3 Experiment 3: Point Insertion Using Different Point Location Algorithm . . . 59
5.3.1 Experiment Setup . . . 59
5.3.2 Experimental Results . . . 60
5.4 Experiment 4: Point Insertion using Exact and Inexact Computation . . . 61
5.4.1 Experiment Setup . . . 61
5.4.2 Experimental Results . . . 62
5.5 Experiment 5: Constraint Insertions . . . 64
5.5.1 Experiment Setup . . . 64
5.5.2 Experimental Results . . . 67
5.6 Experiment 6: Dynamic Updates of Constrained Polygon . . . 68
5.6.1 Experiment setup for constrained polygon insertions . . . 69
5.6.2 Experimental Results . . . 70
5.7 Conclusion . . . 70

6 Conclusions and Future Work **73**

Bibliography **79**

List of Figures

2.1 A triangulation with v=6, k=5, e=10, and f=6 6
2.2 Edge flipping and change of circumcircles 9
2.3 Delaunay triangulation and constrained Delaunay triangulation 11
2.4 Vertex v is not visible from the interior of face $\triangle abc$ 12

3.1 A simple list representation 15
3.2 A Winged Edge representation 15
3.3 FE-HalfEdge and VE-HalfEdge representation 16
3.4 Loop traversal using the Half Edge representation 17
3.5 A complete Quad Edge example 18
3.6 Quad Edge with edge operator Rot,Sym and Next 19
3.7 Our Half Edge data structure 20
3.8 Delaunay triangulation and its dual - the Voronoi Diagram 22
3.9 An example of straight walk 24
3.10 An example of orthogonal walk 24
3.11 An example of oriented walk 25
3.12 Point location loop in oriented walk 26
3.13 Escaping infinite loop with stochastic walk 26
3.14 An example of sector-base point location 28
3.15 Point insertion cases 30
3.16 Constraint insertion 34
3.17 Subregions to be re-triangulated 35
3.18 Case 3 of vertex removal 38

4.1 Incorrect point insertion due to rounding error in geometric function 46
4.2 Calculating intersection coordinates using integer endpoints 52
4.3 The replacement of intersection tests in constraint insertion 53

5.1 Average execution time of Cubic-root sampling, Sector-base and Remembering Stochastic Walk point localization 58
5.2 Initial bounding box setup 59
5.3 Point insertion of 1000 points 60
5.4 Average execution time of point insertion using Cubic-root sampling, Sector-base and Remembering Stochastic Walk point localization 61
5.5 Execution time of point insertions 62
5.6 1000 short axis aligned constraint insertions 63
5.7 200 long axis aligned constraint insertions 63
5.8 50 random constraint insertions 64
5.9 Execution time of short uniform constraint insertions 65
5.10 Execution time of long uniform constraint insertions 66

5.11 Execution time of random constraint insertions 67
5.12 Axis aligned moving constrained polygon 69
5.13 Rotated moving constrained polygon 70
5.14 Execution time of moving rotated square 71
5.15 Execution time of moving axis aligned square 72

Chapter 1

Introduction

1.1 Motivation

Pathfinding is an important research topic for many applications such as routing in networks, motion planning in robotics and computer games. A common approach to pathfinding in computer games is to incorporate a tile grid that overlays the game map and forms a graph of traversable tiles [38]. Best-first search algorithms such as A* are used to find the optimal path between start and goal location through traversable tiles. The tile grid representations are easy to use and the graph of traversable tiles is adapted easily by A*. However, there are several disadvantages of tile grid representations:

- The tile grid may represent a polygonal environment imprecisely as the polygonal obstacles of arbitrary orientation are represented by axis-aligned grid tiles.
- The surface area of the grid tile is uniform. A large number of grid tiles are wasted to represent large open map regions.
- Path smoothing may be required as post-processing of tile paths to achieve straight line movement or smooth cornering. The path smoothing is needed because the optimal tile path is formed by searching eight adjacent tiles in compass direction.
- Grid-based pathfinding usually have large search space with each tile being a graph search node of the search algorithm.

Polygonal mesh representations offer solutions to the problems above:

- Polygonal environments can be represented precisely using polygonal shapes and the boundaries of the polygonal obstacles can be modeled exactly.

- The surface area of the polygonal shapes is not uniform. Large open regions in the environment can be represented by a few polygonal shapes. Therefore, the polygonal representation can significantly reduce the search space for pathfinding.
- Path smoothing is not required for polygonal paths. The actual path naturally goes through vertices of the polygonal path.
- Triangulation-based pathfinding algorithms working on the base triangulation (TA*) and on abstracted graphs (TRA*) as proposed by Demyen and Buro [12] offer significant speed improvements over the grid-based pathfinding algorithms. The search speed is improved because the search space in triangulation-based pathfinding is significantly reduced by the triangulation representation.

1.2 Previous Work

Kallmann et al. proposed the use constrained Delaunay triangulation to model planar environments [23]. The boundaries of obstacles are modeled as constrained edges which are fixed in the triangulation. The environment is then triangulated with unconstrained edges to form a constrained triangulation of the environment. A review of the properties of the constrained Delaunay triangulation is given in Chapter 2. Path planning is done in the triangulated environment [22] with midpoints of triangle edges as the search nodes. Kallmann's DCDT software implements incremental constrained Delaunay algorithms and provides dynamic update operations to the constrained Delaunay triangulation. Kallmann's DCDT implementation is efficient and the computation speed is fast using floating point arithmetic. However, the use of fixed-precision floating point computation may lead to crashes, infinite loops and incorrect outputs. It is also worth noting that the DCDT software package has its own licensing agreement which limits the package to educational, research and non-profit purposes only.

Our original implementation of TA* and TRA* are based on Kallman's DCDT software as the underlying triangulation implementation with additional information (size, degree or linkage of abstractions) attached to different features (vertices edges, faces) of the triangulation. Our interest in developing triangulation software using exact computation comes from the need for a free computational geometry software package with robust implementation of constrained Delaunay triangulation algorithms. The robustness of the software package is important because it is intended to be used in the Open RTS Game Toolkit (ORTS) [6] as part of the triangulation pathfinding engine.

1.3 Contributions

Our goal is to provide a free software package with correct and efficient implementation of constrained Delaunay triangulation algorithms and data structure.

In this thesis, we describe our dynamic constrained Delaunay triangulation software package which uses exact geometric computations. The contributions of this thesis to the field of computational geometry are:

- Most geometric algorithms are not designed for robust implementations. A robust implementation of the geometric algorithms depends on the exact computation of geometric functions [30]. However, exact computations tend to be much slower than inexact computations such as fixed-precision floating point computations. We created a free C++ software package which implements dynamic constrained Delaunay triangulation. The package is fast, robust and versatile. Exact computation is used to compute geometric functions such as the *Orientation*, *InCircle* and *Intersection* tests.

- Geometric algorithms are typically hard to implement because of the degenerate cases and rounding errors. We documented the degenerate cases and the geometric functions affected by numerical errors in various update scenarios. We propose modifications and extensions to the original algorithms to handle these cases.

- Kallmann's DCDT implementation uses the Oriented Walk algorithm for point localization which suffers from degenerate cases during the walking process. Our implementation uses a sector-based point localization algorithm [12]. We proposed a replacement of the Oriented Walk algorithm by the Remembering Stochastic Walk algorithm as the underlying walking algorithm for the sector-based point localization. The Remembering Stochastic Walk algorithm saves on average 1.5 *Orientation* tests per visited triangle by remembering the cross edges between triangles. The savings in computation time can be quite significant because exact computation is used to compute the *Orientation* test. The Remembering Stochastic Walk algorithm naturally handles the degenerate cases in Oriented Walk by randomizing the edge test sequence. Therefore, the implementation is simpler without the need to distinguish different degenerate cases.

- The sector-based point localization algorithm proposed in [12], which is used in the point insertion operation, causes the average performance for n point insertions to be quadratic. This is caused by the need for sector updates after each point insertion

to maintain correct sector-triangle associations. We proposed a new sector update scheme which eliminates the need for liner-time sector updates. Experimental results indicate that the average performance of n point insertions using our Sector-based point localization algorithm is better than point insertions using other on-line point localization algorithms. Our implementation performs better on large sets of input compared to Kallmann's point insertion implementation which uses Oriented Walk algorithm.

- We discuss the numerical rounding errors in fixed-precision floating point arithmetic and review the exact computation of constrained Delaunay triangulations using dynamic filters and arbitrary-precision arithmetic. Extensions to the original algorithm and implementation are proposed to improve the efficiency of exact computations. The computation in *Orientation* test is much simpler than the computation in *Intersection* test. We use the formulation of four orientation tests to test the intersection test between two line segments. The computation time is reduced for some non-intersecting cases where only two orientation tests are needed. With careful engineering of our implementation, we are able to quickly compute and store exact intersection coordinates of crossing line segments with integral endpoint coordinates in the range of 0 to 10^6. The range is sufficient for many applications, such as pathfinding in video games. The replacement of arbitrary-precision arithmetic by fixed-precision arithmetic in the intersection computation speeds up the computation and reduces memory requirements.

1.4 Thesis Outline

Chapter 2 reviews Delaunay and constrained Delaunay triangulation and motivates why they can be constructed by means of edge flipping.

Chapter 3 surveys different algorithms for computing constrained Delaunay triangulations and discusses the incremental algorithms used in our implementation.

Chapter 4 discusses the exact computation of geometric functions by means of combining interval arithmetic and arbitrary-precision arithmetic.

Chapter 5 presents the experiments and compares the performance of update operations in Kallmann's DCDT implementation and in ours.

Chapter 6 concludes the thesis with a summary and a discussion of future work.

Chapter 2

Properties of Triangulations

In this chapter, we first give a review of 2D triangulations. We will show various properties of a point set triangulation discussed in [28]. Secondly, we introduce the Delaunay triangulation which maximizes the minimum interior angle among all possible triangulations of a point set. A sequence of proper edge flips will increase the minimum angle in the triangulation and result in the maximization of the minimum angle in Delaunay triangulation as proved by Herbert [15]. We then describe constrained Delaunay triangulation which relaxes the definition of Delaunay triangulation to accept constrained line segments as inputs which can not be changed in the triangulation [1],[9]. Finally, we discuss a dynamic version of the constrained Delaunay triangulation that can handle mesh changes efficiently.

2.1 Planar Graphs

When we link a set of random points on a piece of paper, we tend to do it without crossing existing edges. This is done to avoid congestion in the drawing. Such non-crossing links are desirable in many applications. Examples are city planning, drainage design and object modeling.

In computational geometry, such drawings without crossing edges are called planar graphs. If we only use straight lines to connect the points, the graph is called planar straight line graph (PSLG). One of the most important properties of planar graphs is Euler's formula:

Theorem 2.1.1 *Let PG(P) be a planar graph with v vertices, e edges and f faces, then:*

$$v = e - f + 2 \tag{2.1}$$

An inductive proof of Theorem 2.1.1 can be found in [4]. We will use Euler's formula to prove the property of triangulations in the next section.

2.2 Triangulations

As explained in [7] and [28], a triangulation T(P) is a maximal embedded planar graph in the plane. The triangulation connects all the points in P with a set of edges E. Edges in E intersect only at common endpoints in P. Different edges contain different sets of endpoints, and there are no intersecting edges in E. Also, any new edge inserted after the triangulation will intersect an existing edge in the planar graph. If not all points in P are collinear, the boundary edges of the triangulation form a convex hull of P (see Figure 2.1). Therefore, a triangulation partitions complicated geometric objects into simple ones, namely triangles. Some important properties of triangulations are stated in the following theorem:

Theorem 2.2.1 *Let P be point set with $|P| = v > 2$ and T(P) be a triangulation of P with k boundary edges. Then, there are exactly e=3v-3-k edges and f=2v-1-k faces in the triangulation.*

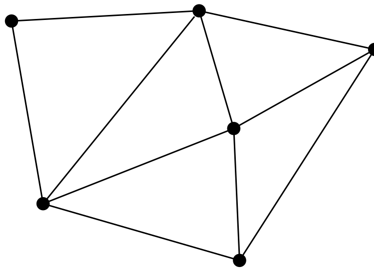


Figure 2.1: A triangulation with v=6, k=5, e=10, and f=6

Proof Let T(P) be a triangulation containing v vertices, e edges and f faces. Then there are m interior triangles and one outer face: $f = m + 1$.

$$f = m + 1 \tag{2.2}$$

Each face has three edges, so the number of double counted edges is

$$
\begin{aligned}
2e &= 3m + k \\
e &= \frac{3m + k}{2}
\end{aligned} \tag{2.3}
$$

Since a triangulation is a planer graph, we can apply Euler's formula for planer graphs [4], which states:

$$v = e - f + 2 \tag{2.4}$$

Substituting (2.2) and (2.3) in (2.4) gives us:

$$v = \frac{3m + k}{2} - (m + 1) + 2 \tag{2.5}$$

$$2v = 3m + k - 2m - 2 + 4 \tag{2.6}$$

Solving for m yields:

$$m = 2v - 2 - k \tag{2.7}$$

Substituting m in(2.2) and (2.3)results in (See Figure 2.1):

$$e = 3v - 3 - k \tag{2.8}$$

$$f = 2v - 1 - k \tag{2.9}$$

2.2.1 Basic Routines for Triangulation

The basic operations for creating a triangulation incrementally are:

- Point localization: The point localization procedure locates the face containing the point to be inserted, or it detects when the points coincides with an existing point or lies on an existing edge.
- Point Insertion: The point insertion procedure inserts a point into the current triangulation, by splitting edges if necessary and adding new edges.

The triangulation of a point set can be constructed using Radial Sweep [31]. The central point of the input is determined and connected to other points in the input. Then, the boundary edges are connected. Finally, a convex hull of the input set is constructed to form

the triangulation of the input point set. Other more efficient algorithms such as Divide and Conquer and Sweep Line can also be used to construct triangulation of a given point set (see Section 3.2). We will discuss the details of point localization operation in Section 3.2.1 and point insertion operation in Section 3.2.2.

2.3 Delaunay Triangulations

A special case of the triangulation for point set P is the Delaunay triangulation DT(P). The difference between a general triangulation and a Delaunay triangulation is that the minimum interior angle of the triangles in DT(P) is maximized among all possible triangulations of P. Intuitively, this increases the "fatness" of the triangles in DT(P), and therefore generates well shaped triangles that tend to be nearly equiangular [36] which covers the area more uniformly, which is beneficial in applications like pathfinding, in which sliver-like triangles can increase the search overhead of A*. Therefore, the well shaped triangles are desirable in our application.

First, we define the minimum angle and empty circumcircle property and prove the connection between these two properties.

Definition 2.3.1 *Let $A(T(P)) = [a_1, a_2...a_n]$ be the interior-angle vector of triangulation $T(P)$ of point set P in ascending order.*

Definition 2.3.2 *Let CC($p_i p_j p_k$) be the circumcircle of triangle $\triangle\ p_i p_j p_k$ in the triangulation. CC($p_i p_j p_k$) is empty if and only if it does not contain any point $p \in P$ in its interior.*

We use Thale's theorem to prove the geometric properties of Delaunay triangulation.

Theorem 2.3.3 *[28] Let $\overline{ab}$ be the line segment intersecting circle C at points a and b. Point c lies inside C, d lies on C and e lies outside C if and only if $\angle acb > \angle adb > \angle aeb$*

Lemma 2.3.4 *Let $\triangle abc$ and $\triangle abd$ be two adjacent triangles sharing the common edge $\overline{ab}$. If either CC(abc) or CC(abd) is non-empty, then the minimum interior angle of the angle vector [$a_1 \ldots a_6$] increases after flipping the common edge.*

Proof We follow the description in [15] to prove the above lemma. Let $\overline{ab}$ be the common edge of two adjacent triangles $\triangle abc$ and $\triangle abd$ with interior angles a_1, a_2, a_3 and a_4, a_5, a_6 (see Figure 2.2 left). Hence, $\triangle abc$ and $\triangle abd$ forms a convex quadrilateral $adbc$ with $\overline{ab}$ being the diagonal of the quadrilateral. Assume point d is inside the CC($\triangle abd$). Flipping

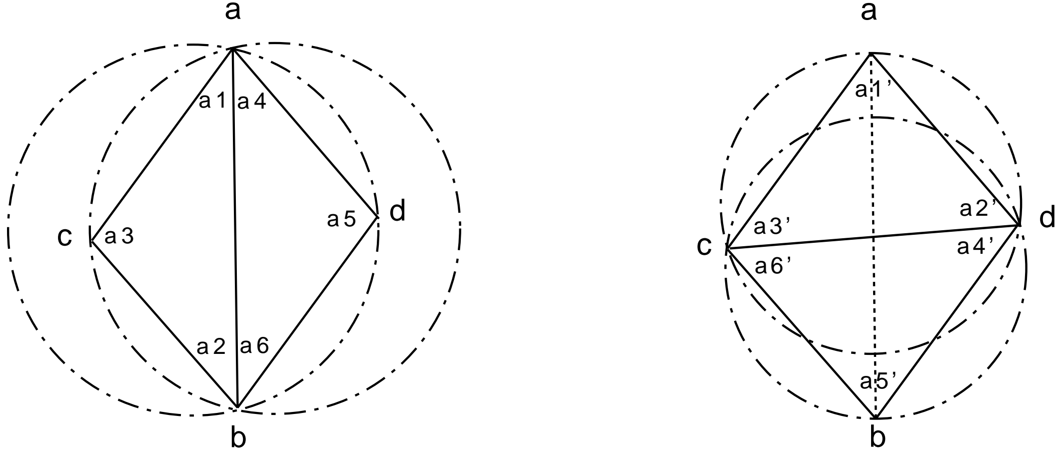


Figure 2.2: Edge flipping and change of circumcircles

the diagonal $\overline{ab}$ to $\overline{cd}$ replaces the existing triangles $\triangle abc$ by $\triangle acd$ with interior angle a'_1, a'_2, a'_3 and replaces $\triangle abd$ by $\triangle cdb$ with interior angle a'_4, a'_5, a'_6 (see Figure 2.2 right). We prove that the minimum angle of the new angle vector will increase after the edge flip when the empty circumcircle property is restored. First the trivial case:

$$
\begin{aligned}
a'_1 &= a_1 + a_4 > a_1 \\
a'_5 &= a_2 + a_6 > a_2
\end{aligned}
$$

And following Thales' theorem, we have:

$$
\begin{aligned}
a'_3 &> a_6 \\
a'_6 &> a_4
\end{aligned}
$$

Finally, the fact that the opposite angles are identical gives us:

$$a'_2 + a_4 = a_2 + a'_6 \Rightarrow a'_2 > a_2 \tag{2.10}$$

$$a'_4 + a_6 = a_1 + a'_3 \Rightarrow a'_4 > a_1 \tag{2.11}$$

Therefore, the minimum angle of the new angle vector $[a'_1 \dots a'_6]$ increases after the edge flip that restores the empty circumcircle property. The minimum angle of the new angle vector stays the same if a, b, c, d are co-circular and increases otherwise. ∎

This shows that the minimum angle increases after the flip. Now we show that after flipping, the circumcircles are empty.

Lemma 2.3.5 *Let $\triangle$ abc and $\triangle$ abd be two adjacent triangles sharing a common edge $\overline{ab}$. If either CC(abc) or CC(abd) is a non-empty circumcircle, then CC(adc) and CC(cdb) are empty after flipping $\overline{ab}$ to $\overline{cd}$.*

Proof Consider the fact that angle a_2 and a'_2 share the same opposite edge $\overline{ac}$. Following Equation (2.10) and Thales' theorem, point b is outside CC($\triangle adc$). Similarly, a_1 and a'_4 share the same opposite edge $\overline{db}$. Following Equation(2.11) and Thales' theorem, point a is outside CC($\triangle cdb$). Therefore, CC(adc) and CC(cdb) are empty after flipping $\overline{ab}$ to $\overline{cd}$ ∎

We formally define the Delaunay Triangulation as the following:

Definition 2.3.6 *Let T(P) be a triangulation of a set of n>2 points P and CC($p_ip_jp_k$) be the circumcircle of triangle $\triangle p_ip_jp_k$ in T(P). T(P) is a Delaunay triangulation if and only if all CC($p_ip_jp_k$) in T(P) are empty.*

Lemma 2.3.5 suggests that we can construct a Delaunay triangulation of P by flipping the diagonal edges in non-Delaunay triangulation of P until the circumcircles of all triangles are empty. Definition 2.3.6 implies that the edge flipping process for Delaunay triangulation terminates. The following lemma shows the termination of edge flipping process:

Lemma 2.3.7 *[15] The edge flipping process for Delaunay triangulations terminates.*

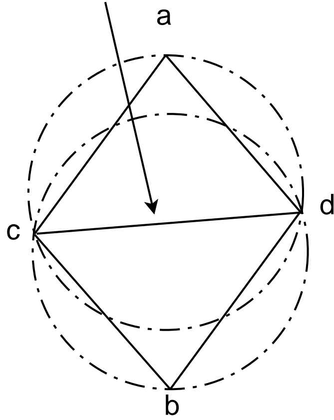
A proof of Lemma 2.3.7 by lifting the triangulation is contained in [15]. We can also show that there are only finitely many triangulations for a given point set and each edge flip switches the triangulation from one to the other while increases the interior-angle vector. Therefore, the edge flipping process terminates.

Moreover, the Delaunay triangulation has the following property:

Theorem 2.3.8 *[28] Any Delaunay triangulation of point set P maximizes the minimum angle over all triangulations of P*

2.4 Constrained Delaunay Triangulations

For some applications, Delaunay triangulation of point sets may not be sufficient to represent the environment precisely. For example, the existing roads and bridges can not be represented by points in the triangulation of terrains. Constrained Delaunay triangulation is a extended version of Delaunay triangulation whose input consists of vertices and non-crossing edges that must be present in the triangulation. Figure 2.3 shows that a line



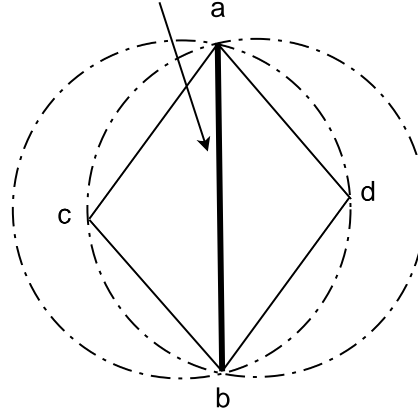


Figure 2.3: Delaunay triangulation and constrained Delaunay triangulation

segment input breaks the empty circumcircle property of the Delaunay triangulation. The input is constrained, so no edge flip can be applied to that line segment. In this section, we discuss Constrainted Delaunay Triangulation (CDT) which allows us to construct triangulations with point and line segment inputs.

First, we have the following definition and notions.

Definition 2.4.1 *Let P be a set of points and E be a set of line segments. And let a and b be two points in the plane. a and b are visible to each other if and only if the segment $\overline{ab}$ does not contain any points from P in its interior and does not cross any line segments in E.*

Furthermore, the weak empty circumcircle property is defined as follows:

Definition 2.4.2 *Let $CC(p_ip_jp_k)$ be the circumcircle of triangle $\triangle\, p_ip_jp_k$ in the constrained Delaunay triangulation. $CC(p_ip_jp_k)$ is empty if and only if it does not contain points from P in its interior that are visible from point p_i, p_j or p_k.*

We try to construct a triangulation similar to the Delaunay triangulation while preserving the weaker circumcircle property. The constrained Delaunay triangulation relaxes the circumcircle property to cope with constrained line segments. Intuitively, we can view the constrained line segments as a set of obstacles, which block the visibility from other vertices in the triangulation (see Figure 2.4).

With the weak empty circle definition above, we can formally define the constrained Delaunay triangulation as the following:

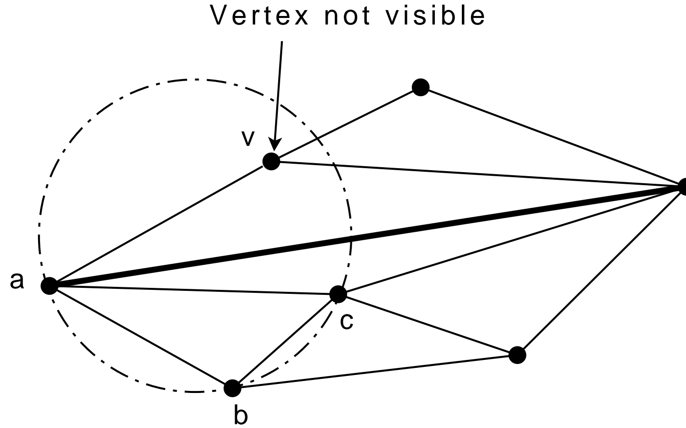


Figure 2.4: Vertex v is not visible from the interior of face $\triangle abc$

Definition 2.4.3 *[8] A constrained Delaunay triangulation CDT(P,E) of point set P and constrained edge set E is a triangulation T of P with E included in T. Each edge e of T is either an edge from E or there exists a circle C with the following properties*

- *the endpoints of e are on the boundary of C, and*
- *if any point from P is in the interior of C then it is not visible from the endpoints of e (i.e. it satisfies the weak empty circle property).*

The constrained Delaunay triangulation is as close to the Delaunay triangulation as possible while keeping the constrained edges. Note that it is possible to have a set of constraints that forces the CDT to become any desired triangulation. Therefore, we can construct the constrained Delaunay triangulation which maximizes the minimum interior angle of the triangles as much as possible by flipping only the unconstrained edges [1].

2.4.1 Basic Routines for constrained Delaunay Triangulation

The input of the constrained Delaunay triangulation consists of a set of points and a set of constrained edges. The points and constrained edges in the triangulation are fixed. Therefore, they are not allowed to be changed. The constrained Delaunay triangulation can be constructed with the following operations:

- Point insertion for constrained Delaunay triangulation.
- Constraint insertion.

The difference between point insertions for Delaunay triangulation and point insertions for constrained Delaunay triangulation is the process of restoring the empty circumcircle

property. The constrained edges are fixed for constrained Delaunay triangulations. Therefore, the edge flipping procedure restores the triangulation to constrained Delaunay triangulation preserving the weak empty circumcircle property. Section 3.3 explains the details of the incremental algorithm for constraint insertions.

2.5 Dynamic Constrained Delaunay Triangulation

In the previous section, we showed that the constrained Delaunay triangulations can be generated by ensuring the weaker circumcircle property. In this section, we explore the requirements for maintaining constrained Delaunay triangulation dynamically.

Geometric algorithms that assume static inputs and use preprocessed data structures are not suitable for applications that require dynamic updates. For example, meshes can change frequently in applications areas such as CAD, robotics and video games. The focus of this work is to create fast terrain modeling software for RTS games [6] which allows us to perform pathfinding on triangulated meshes [12]. For this we need to update the constrained Delaunay triangulation when changes such as constructioned buildings or destroyed bridges destructions occur. As the changes in the terrain are not known beforehand, preprocessing does not help much and efficient local updates on the constrained Delaunay triangulation are required. Dynamic constrained Delaunay triangulation software satisfies the following conditions:

- It provides local updates to the constrained Delaunay triangulation.
- It preserves the weak empty circle property after each local update.

Dynamic constrained Delaunay triangulation algorithms were introduced by Kallmann et al.[23]. The DCDT software package implementing these algorithms uses floating point arithmetic and handles degenerate cases such as overlapping and intersecting constraints.

In the next chapter, we survey various incremental update algorithms and their implementations for efficient insertion and removal of vertices and constraints in constrained Delaunay triangulations.

Chapter 3

Computing Dynamic Constrained Delaunay Triangulations

In this chapter, we discuss the algorithms and their implementation for constructing and updating constrained Delaunay triangulations dynamically. In particular, we describe the underlying geometric data structures, point localization and incremental updates for constrained Delaunay triangulations.

3.1 A Survey of Basic Geometric Data Structures

In computational geometry many ways have been considered to represent polygonal meshes which consist of a vertex set V, an edge set E and the topological representation of a face set F. In this section, we survey several data structures and motivate our choice for mesh representation.

3.1.1 Element Lists and Adjacency Lists

A straightforward mesh representation is to store a list of vertices, a list of edges and a list of faces, see Figure (3.1). This simple data structure provides the geometric information of the mesh structure but it requires additional storage for maintaining topological information in the form of lists that store the adjacency relation between faces, edges and vertices. The main disadvantage of this straightforward approach is the redundant storage and the long query time. Additionally, maintaining the topological information is more complex compared to the other more elegant data structures.

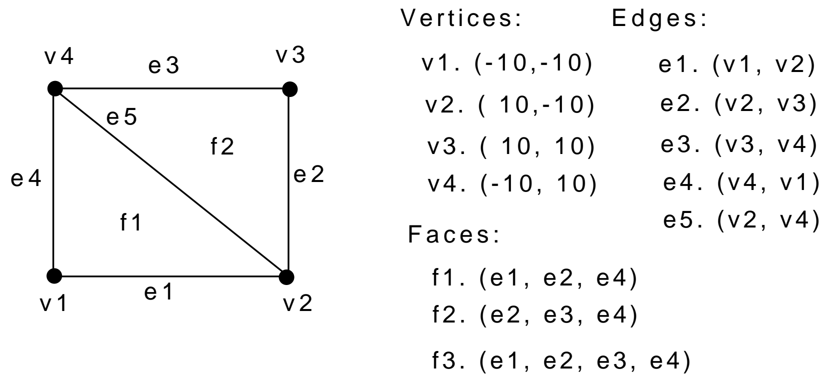


Figure 3.1: A simple list representation

3.1.2 Winged Edges

In 1975, Baumgart introduced the Winged Edge data structure for polygonal models [3]. In order to represent a polygonal mesh using this data structure, each vertex and face stores a reference to its incident edge, and each edge e stores the following references (see Figure 3.2):

- two pointers to its vertices: Dest(e) and Orig(e).
- two pointers to its incident faces : LFace(e) and RFace(e).
- four pointers to its adjacent edges: NextCCW(e), PrevCCW(e), NextCW(e) and PrevCW(e).

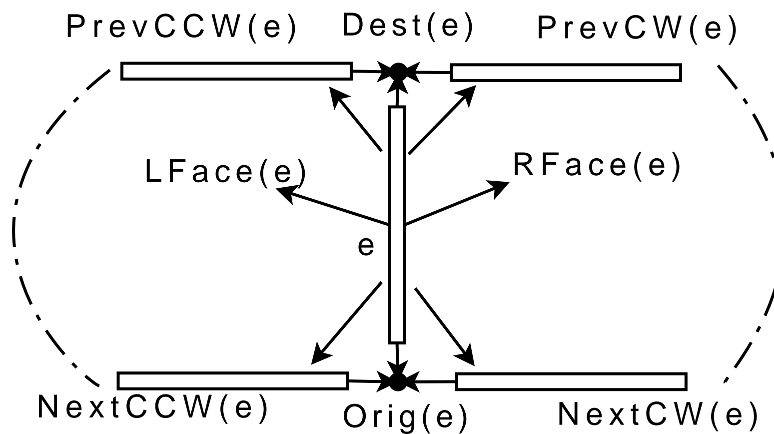


Figure 3.2: A Winged Edge representation

The data structure contains both geometric and topological information of the polygonal mesh. Moreover, it supports fast traversal between vertices, edges and faces, which is essential for many geometry algorithms. However, the Winged Edge representation lacks orientation information for edges. Therefore, traversing from edge to edge requires a case distinction. In the following sections, we will discuss two variations of the Winged Edge data structure that provide orientation information of the edges.

3.1.3 Half Edges

A variation of the Winged Edge data structure splits edges into two half edges of opposite direction and stores mutual references between the opposite half edges [25]. The split of the edge can be either along the faces (FE-HalfEdge) or upon the vertices (VE-HalfEdge), (see Figure 3.3).

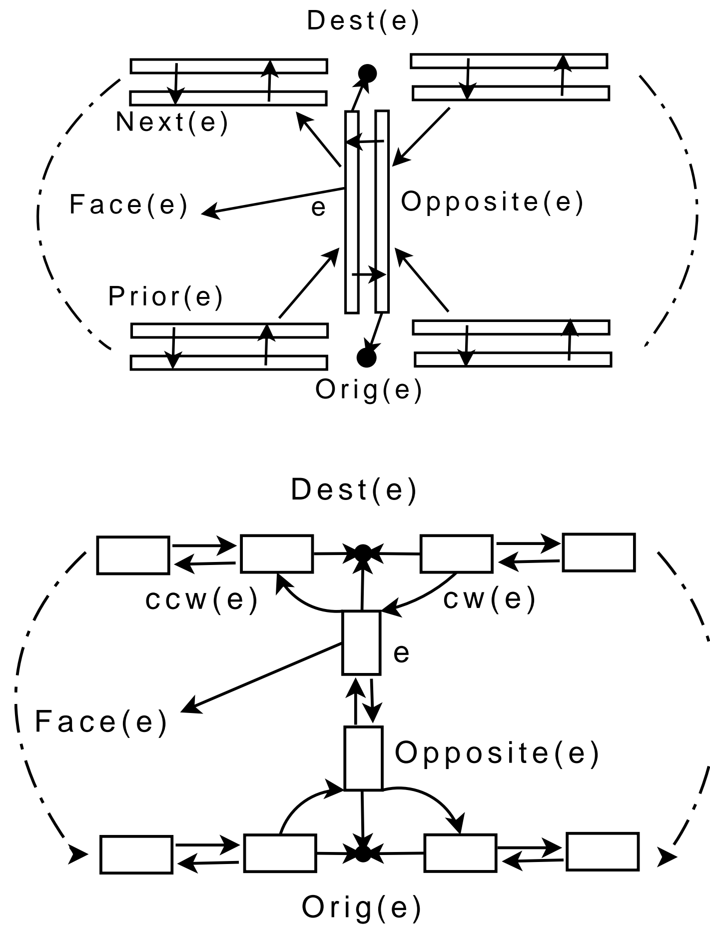


Figure 3.3: FE-HalfEdge and VE-HalfEdge representation

Algorithm 1 vertexLoop(HalfEdge $e1$)

```
1: HalfEdge e2 := e1
2: repeat
3:     e2 := e2.next()
4: until e2 = e1
```

Algorithm 2 faceLoop(HalfEdge $e1$)

```
1: HalfEdge e2 := e1
2: repeat
3:     e2 := e2.next().oppositeEdge()
4: until e2 = e1
```

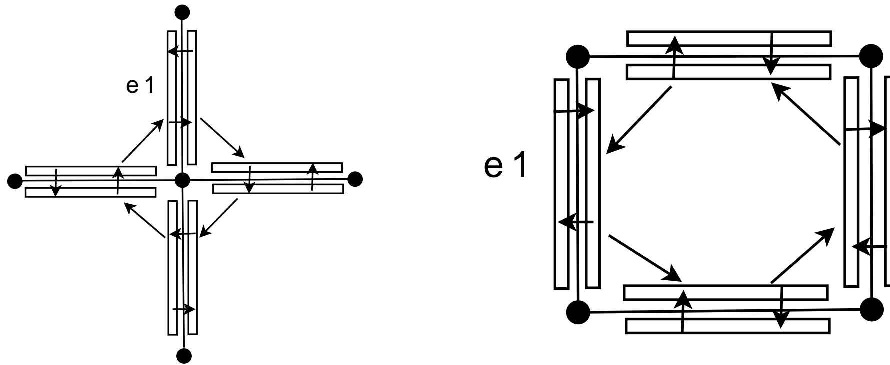


Figure 3.4: Loop traversal using the Half Edge representation

In additional to the geometric and topological information provided in the Winged Edge data structure, the orientation encoding of each edge is also implicitly defined. Therefore, the Half Edge data structure provides a more efficient traversal mechanism to loop around a face or a vertex, see code listing in Algorithm 1, Algorithm 2 and their corresponding figure, Figure 3.4 left, Figure 3.4 right.

The VE-HalfEdge representation, or Split Edge, is the dual form of the FE-HalfEdge representation [25]. A loop around the face in the VE-HalfEdge representation has the dual of a loop around the vertex in the FE-HalfEdge representation.

3.1.4 Quad Edge

The Quad Edge is another variant of the Winged Edge data structure where the edge is split into a quadruplet [25] which is illustracted in Figure 3.5. It was first introduced by L. Guibas and J. Stolfi to provide a uniform view of a subdivision and its geometric dual [19].

For example, the Delaunay Triangulation and its corresponding Voronoi Diagram.

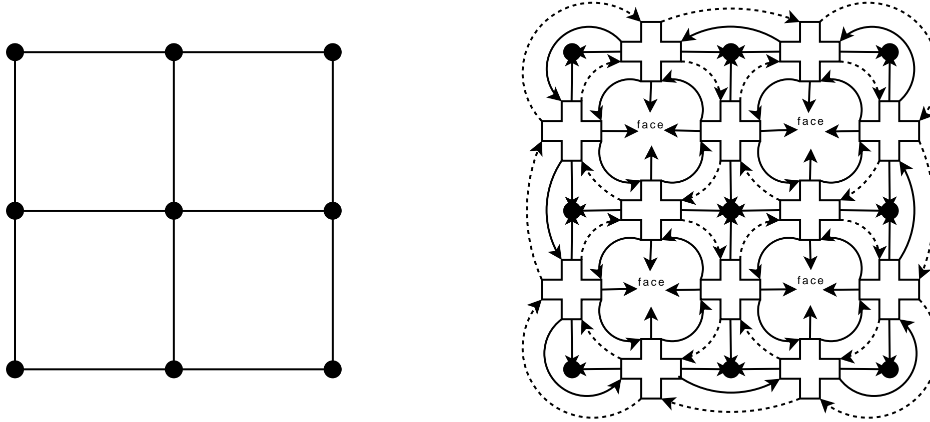


Figure 3.5: A complete Quad Edge example

A Quad Edge cell can be represented with a triple (e, r, f) where e, r and f are defined as follows [25]:

- A vector of four edge records $e[0] \ldots e[3]$ that stores the geometrical information of a vertex or a face and a reference to an adjacent Quad Edge.
- A two bit index r to address the edge record. r is used to indicate the current viewing of one of the edge record $e[0] \ldots e[3]$.
- A one bit counter f to switch views from above to below. We only need f if we want to represent both primal and dual view of a geometric structure.

We can remove component f from the triple (e, r, f) if the Quad Edge data structure represents meshes such as the Delaunay triangulations. Because of the symmetric representation, the Quad Edge data structure is able obtain a particular edge record with the following algebraic edge operators with a calculus modulus 4 for r:

- $\text{Rot}(e, r) = (e, r + 1)$: the operator returns an edge record rotated CCW 90 degrees from edge record e_r.
- $\text{Sym}(e) = (e, r + 2)$: the operator returns an edge record rotated CCW 180 degrees from edge record e_r.
- $\text{Next}(e, r)$=$(e', r + 1)$: the operator returns an edge rotated counterclockwise around a face or a vertex.

4

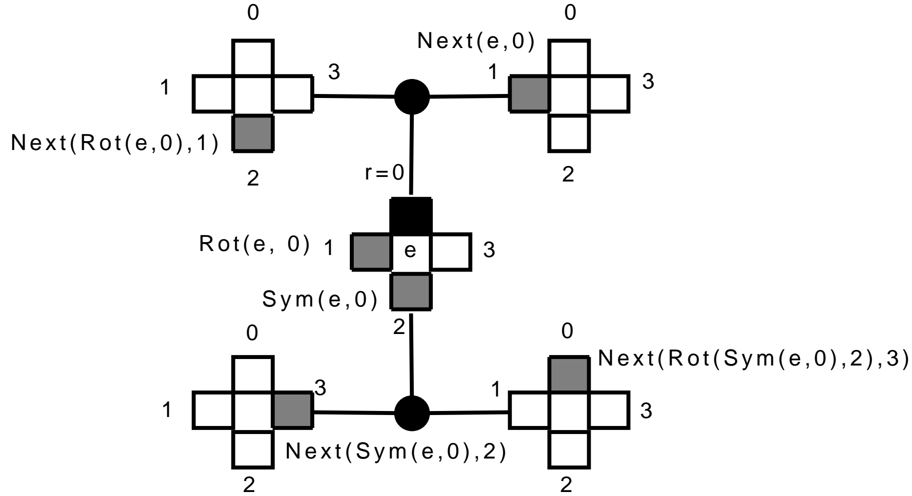


Figure 3.6: Quad Edge with edge operator Rot,Sym and Next

Figure 3.6 illustrates the algebraic edge operators with the starting edge e and the current edge record $e[0]$ marked black. The return edge records from the operators are shown in grey.

3.1.5 Selected data structure

Although the Quad Edge data structure provides a more versatile geometric representation with similar storage requirements, it needs modulus operation and vector accesses for basic traversals. For our project, we choose the Half Edge data structure to represent the underlying triangular mesh because the primal triangulation graph is sufficient for our terrain modeling purposes. Furthermore, the Half Edge data structure provides direct access to adjacent vertices, edges and faces for traversal operations which are frequently used by many geometric algorithms. In addition, the interface of Half Edge data structure is easier to understand. Figure(3.7) shows the design of our Half Edge data structure. For our implementation, the internal storage is vectors storing pointers to different features in the triangulation. We extend the Half Edge data structure with the following attributes in different features:

- Vertex: *OutEdge* is a pointer to one of the outgoing edges originated from the vertex.

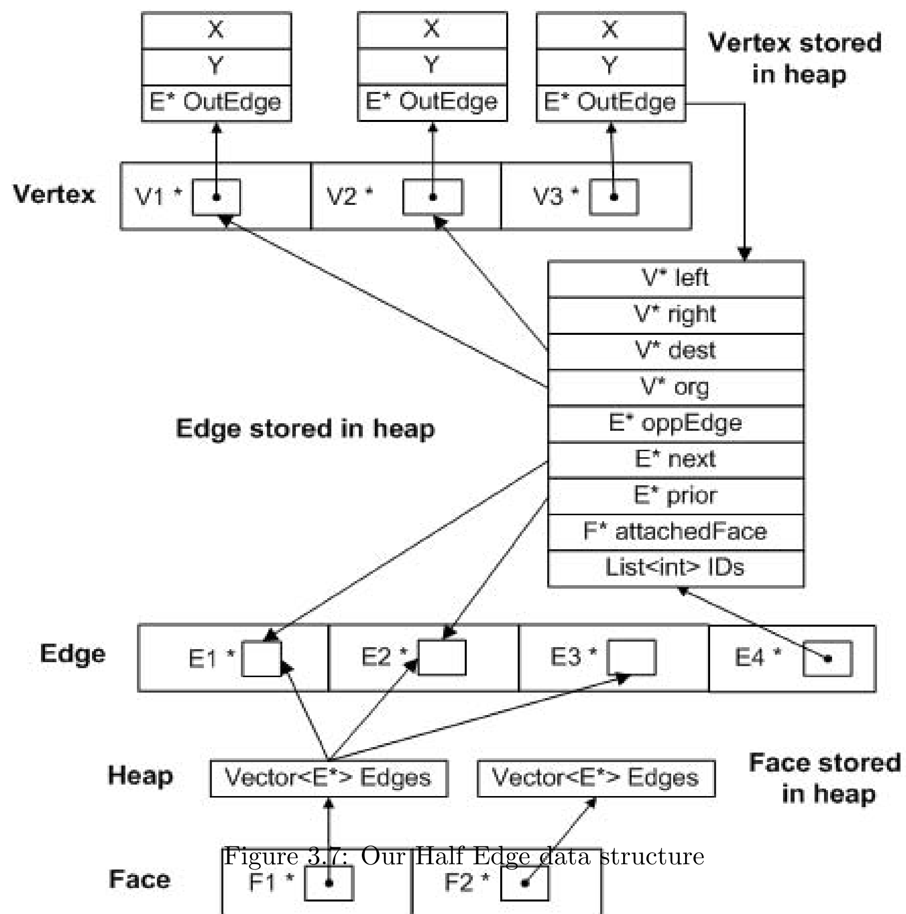


Figure 3.7: Our Half Edge data structure

We uses this attribute as the starting edge for or loop traversals around the vertex.

- Edge: $left$ and $right$ store the farthest constrained endpoints. The integer coordinates of $left$ and $right$ are used to replace the rational coordinates of $dest$ and org as inputs for intersection computation (see Section 4.3.3). The feature also stores a list of IDs for constrained edges. An edge can represent more than one constraint in overlapping cases. The edge is constrained when the ID list is not empty. The ID can also be used to group the constrained edges of different constrained polygons for constrained polygon insertion and removal.

3.2 Point Insertion in Delaunay triangulation

Existing Delaunay triangulation algorithms for point sets can be divided into the following categories:

1. Divide-and-conquers [19]
2. Sweep-line [16]
3. Incremental insertion [19] [28]

The divide-and-conquer algorithm recursively divides the set of input points into smaller and smaller subsets. It stops the division when the subset of points becomes trivial to triangulate into a Delaunay triangulation. It then merges the smaller Delaunay triangulations from bottom up to form one complete Delaunay triangulation. The divide-and-conquer algorithm runs in $O(n \log n)$ time.

The divide-and-conquer and the sweep-line algorithms only work with static inputs. In other words, the set of input points must be known beforehand. In many applications input points are not known in advance and the input changes dynamically. Therefore, incremental algorithms are required. Although the incremental algorithms are not asymptotically optimal in the worst case, they are more suitable for constructing and maintaining a constrained Delaunay triangulation dynamically. Although the incremental algorithm has $O(n^2)$ run time in the worst case, the expected run time can still be $O(n \log n)$ when the input is evenly distributed [20]. Real world application shows that the incremental algorithm is faster on average with randomly distributed inputs.

To insert a point P in the Delaunay triangulation, we need to first locate the triangle F containing the point, insert the point in the triangle by connecting P to the vertices of F and finally restore the Delaunay property use edge flipping. In the following subsections,

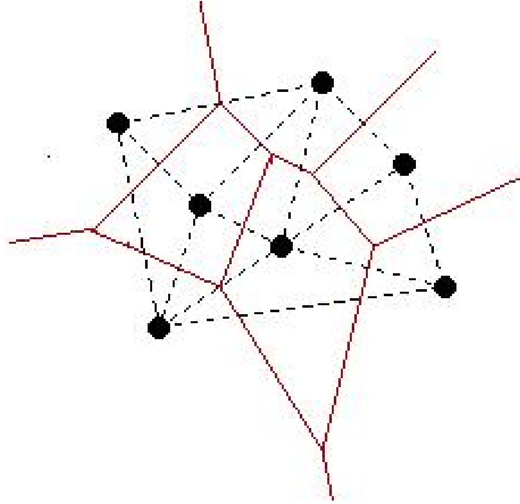


Figure 3.8: Delaunay triangulation and its dual - the Voronoi Diagram

we present the steps of the incremental algorithm and analyze the run time complexity of each step.

3.2.1 Point Localization

Point location is a fundamental problem in computational geometry. It is the starting point of many complex geometric algorithms. An inefficient implementation can affect the overall performance of a geometric algorithm. Various point localization algorithms have been proposed to provide optimal or good average case performance. In this section, we survey the point localization algorithms and discuss the ones suitable for dynamic environments.

First, we define the point localization problem as the following:

Definition 3.2.1 *Given a Delaunay triangulation DT(V,E,F) with a set of vertices V, a set of edges E and a set of face F, and a point p with coordinate x and y in the plane, the goal is to identify a vertex $v \in V$, an edge $e \in E$ or a face $f \in F$ where point p is located.*

A brute force approach is to iterate through all the faces and test if point P is located in the face. Hence, identifying the face point P lies in or the edge or vertex point P lies on has linear time complexity in the number of faces in the mesh

An application using point localization may involve many queries. Several complex methods operating on preprocessed data structures have been proposed to reduce the worst case query time from $O(n)$ to $O(\log n)$. The methods include:

- Vertical slabs [33]
- Triangulation refinement [27]

- Trapezoidal maps [34]

These methods assume a static polygonal mesh and use a preprocessed data structure to speed up queries. However, many applications such as terrain modeling in games have to maintain a dynamic polygonal mesh. The polygonal mesh is called dynamic if both insertions and deletions of the mesh elements are to be executed in real time. Therefore, using a preprocessed data structure is not always ideal for point locating queries because of the cost of maintaining the associated data structure.

A simple walking method for locating query points was first introduced by Green and Sibson [18], based on ideas of Lawson [29]. It can be easily implemented without preprocessing or using any sophisticated data structure. The worst-case running time for a query with the walking point-location method on Delaunay triangulation is $O(n)$ with n vertices[19]. However, the expected running time for a query is $O(\|\overline{PQ}\|\sqrt{n})$ when the vertices are uniformly distributed [13]. $\|\overline{PQ}\|$ denotes the length of the line segment $\overline{PQ}$ where Q is the starting location (explain below) for the walk and P is the query point. Variants of the simple walking algorithm have good average running time. They are simple to implement and more suitable to use for dynamic meshes. The variants of walking algorithms include [13]:

- Straight walk
- Orthogonal walk
- Oriented walk
- Stochastic walk

For the straight walk algorithm, the walking process starts at a random location Q within the mesh. It forms a line segment $\overline{QP}$ between the random location and the query point. The starting location can be a vertex or a point within a random edge or face depending on the implementation. In our example, we choose a random point within a face as the starting location. The walk steps through the faces intersected by the line segment $\overline{QP}$ towards the query point P and the direction of the walk is guided by the intersection test towards P (see Figure 3.9).

The orthogonal walk algorithm partitions the walk into two axis-aligned straight walks. The expensive intersection test used in the walk can be replaced by simple coordinate comparisons while the number of triangles visited of the walk increases (see Figure 3.10). Replacing the intersection test can significantly increase computation speed if exact com-

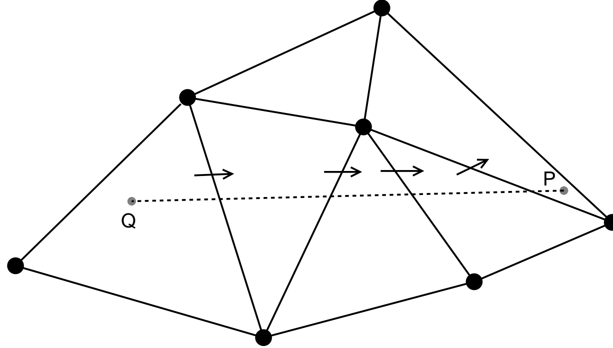


Figure 3.9: An example of straight walk

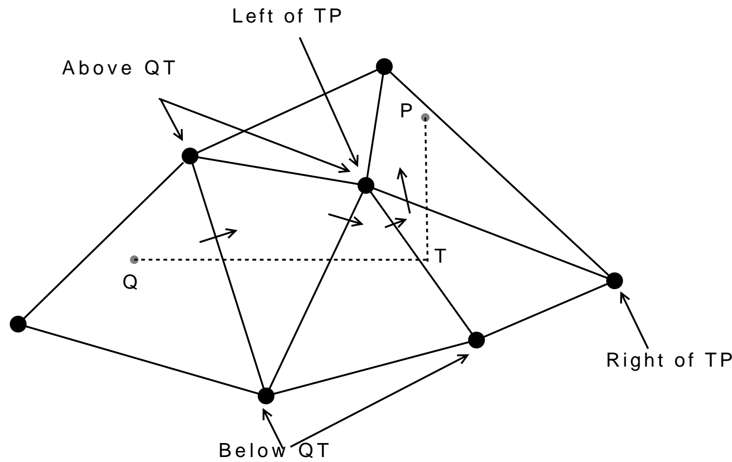


Figure 3.10: An example of orthogonal walk

putation is used for the intersection test. We will discuss the impact of exact computation on the computation speed of geometric functions in the next chapter.

In order to handle the degenerate cases such as $\overline{QP}$ passing through vertices, both the straight walk and orthogonal walk needs to implement specific case handlers, which might require restarting during the walking process. Therefore the implementation is more complex than the proposed straight and orthogonal walk algorithms [13].

Another variant of the straight walk algorithm is the oriented walk. The walk only uses the orientation tests between the current triangle T and query point P to find a crossing edge e separating P and T in a plane. The oriented walk eventually enters the final trian-

gle that encloses the query point and terminates. The oriented walk visits triangles in the vicinity of the line segment $\overline{QP}$ instead of being limited to the ones that intersect $\overline{QP}$ (see Figure(3.11)). It replaces the expensive intersection test with the less expensive orientation test to guide the walk towards query point P.

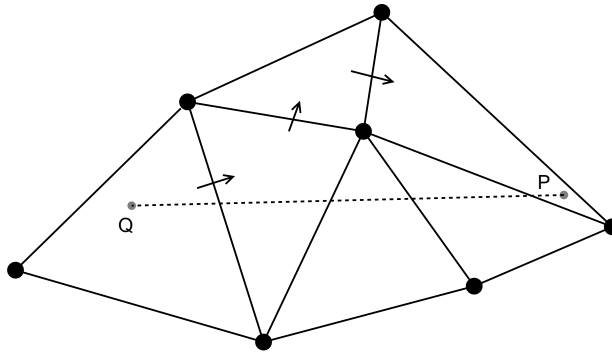


Figure 3.11: An example of oriented walk

The oriented walk algorithm inherently handles the degenerate cases. However, it can enter an infinite loop for non-Delaunay triangulations. Figure(3.12) shows an example of a walking sequence in non-Delaunay triangulation that starts from the edge at step 1 and repeats after crossing the edge at step 6. The loop occurs because the first edge selected for the orientation test is fixed. Therefore, the oriented walk algorithm may enter an infinite loop in a constrained Delaunay triangulation because it contains some constrained edges that are not Delaunay.

The stochastic walk algorithm solves the infinite loop problem by randomizing the test sequence for edges in the triangle. Hence, allowing the walk to eventually escape the loop. The randomness is important when more than one edge can be selected as the crossing edge. (see Figure 3.13). The geometric function used in the stochastic walk is the orientation test. We can identify the crossing edge with 1, 2 or 3 orientation tests within the current triangle. Therefore, the expected number of orientation tests is $\frac{1+2+3}{3} = 2$.

The efficiency of the stochastic walk algorithm can be further improved by remembering the orientation of the crossing edge (Remembering Stochastic Walk [21]). It saves one orientation test per visited triangle . Consider that the walk moves from $\triangle abc$ to $\triangle bcd$ crossing

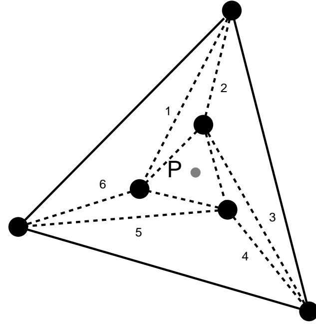


Figure 3.12: Point location loop in oriented walk

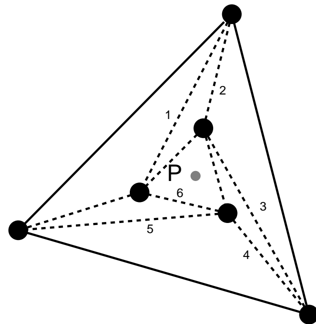


Figure 3.13: Escaping infinite loop with stochastic walk

edge $\overline{bc}$, the orientation of $\overline{bc}$ is known when the walk enters △bcd. It is not necessary to test upon $\overline{bc}$ in △bcd. Therefore one orientation test can be saved by remembering the entering edge. The next triangle can be found with only 1 or 2 orientation tests within the current triangle and the expected number of orientation tests is $\frac{1+2}{2} = 1.5$. The original algorithm does not distinguish query points located on edge or vertex. We modify the proposed algorithm by adding additional tests for the two cases. The pseudocode is presented in Algorithm 3. The worst case performance of the algorithm is $O(2^{\sqrt[3]{n}})$ and terminates with probability 1 [13].

The walk can start from any triangle in the mesh. Finding a good start triangle can

Algorithm 3 rememberingStochasticWalk(Face $currentFace$,HalfEdge $crossEdge$,Point p)

```
1: HalfEdge e1 := select a random Edge from currentFace;
2: e2 := e1;
3: repeat
4:   if e2 ≠ crossEdge then
5:     Vertex v1 := e2.origin;
6:     Vertex v2 := e2.destination;
7:     if CCW(v2, v1, p) then
8:       return locatePoint(nextFace,crossEdge,p)
9:     end if
10:  end if
11:  e2 := e2.next()
12: until e2 = e1
13:
14: {Additional test for query point on vertex or edge}
15: if onVertex(currentFace, x, y) then
16:   return the vertex found
17: else
18:   if onEdge(currentFace, x, y) then
19:     return the edge found
20:   end if
21: end if
22: return currentFace
```

greatly improve the query time of a point localization using a walking method. A simple approach is to use the last visited triangle in the mesh. This triangle is usually adjacent to the site of a previous change in the triangulation. This is suitable for polygonal constraint insertions where the vertices of the polygon are inserted in order. Therefore, each query point is close to the most recently updated triangles in the mesh. Other more general approaches have been proposed with good average query time. In the jump-and-walk algorithm, an extra jumping step was introduced by Devroye et al. to find a good start triangle by sampling random vertices[14]. The jump-and-walk algorithm consists of the following steps:

Algorithm 4 cubicRootSampling(Face $currentFace$,HalfEdge $crossEdge$,Point p)

```
1: Select m vertices from the current list of n vertices in the mesh, m<n.
2: Find the vertex Q closest to the query point P among the m vertices.
3: Perform the straight walk starting from Q to find the triangle enclosing the query point
   P.
```

The expected running time of the jump-and-walk algorithm is O($\sqrt[3]{n}$) when the number of sampled vertices m is $\lceil \sqrt[3]{n} \rceil$ and the query point P is $\frac{2\sqrt{\log n}}{\sqrt[6]{n}}$ away from the boundary ∂C of a bounded convex set C of unit area [14].

A more straightforward sector-based point localization algorithm was introduced in [12]. It overlays a uniform grid to the triangulation (see Figure 3.14) and associates the triangles

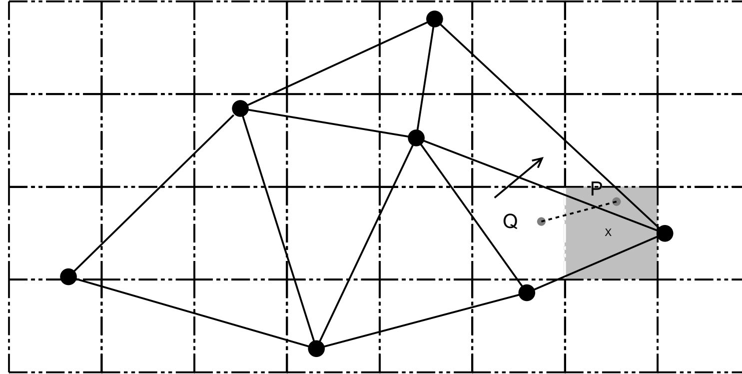


Figure 3.14: An example of sector-base point location

Algorithm 5 sectorBasedLocatePoint(HalfEdge *crossEdge*, Point p)

1: Find sector s_{ij} containing p.
2: Retrieve the corresponding face f in s_{ij}.
3: **if** f is a valid face **then**
4: result := rememberingStochasticWalk(f, *nullCrossEdge*, p);
5: **else**
6: result := rememberingStochasticWalk(updated face in the mesh, *nullCrossEdge*, p);
7: **end if**
8: s_{ij} := result
9: **return** result;

to the sectors where the bounding box of the triangle covers the center of the sectors. The bounding box can be easily calculated by taking the maximum and minimum coordinates of the triangle vertices. The sector location of the query point can be computed and the associated triangle is used as the starting triangle of a Remembering Stochastic Walk. This speeds up the point localization query by jumping to a sector close to the query point and use the associated triangle as the starting location for the stochastic walk algorithm, thereby reducing the number of triangles visited of the stochastic walk. In order to keep

Algorithm 6 sectorUpdates( )

1: **for all** face f in the triangulation **do**
2: Compute the bounding box for f using its maximum and minimum vertex coordinates.
3: **for all** sector s covered by the bounding box **do**
4: Store f in s
5: **end for**
6: **end for**

track of the changes made in the triangulation, there are overheads to update the associating triangles in the sector (see Algorithm 6). The proposed algorithm performs sector updates before each point insertion, which causes quadratic time performance for point insertions. To reduce the overhead, we propose an efficient update method to avoid the invalidation of sectors when the triangulation is changed. The walking process starts from a valid sector triangle and uses the last updated triangle as the starting location when the sector search hits an invalid sector triangle that has been changed or removed in the triangulation. The search result of the Remembering Stochastic walk is stored in the corresponding sector to eliminate the need for linear-time sector updates. The method originated from the observe that the Remembering Stochastic Walk traverses triangles adjacent to the querying sector such that the return value of Remembering Stochastic Walk is located within or adjacent to the querying sector. Therefore, we can store the return value of the current query in the sector, which allows us to dynamically associate the sector to an adjacent triangle without the need to go through all the triangles and update the covering sectors (see Algorithm 5). The algorithm can count the number of triangles visited in Remembering Stochastic Walk and bound the exponential worst case search time by a linear search.

3.2.2 Point Insertion

The implementation presented in this section follows closely the point insertion algorithm described in [28]. In this section, we discuss the details of the point insertion algorithm and describe the important steps to complete the point insertion operation.

After the point localization step, the geometric feature containing the insertion point p_r is found. The geometric feature can be a vertex, an edge or a face in the mesh. If the p_r is on a vertex, we can simply reuse the existing vertex because it is already inserted at the location. The remaining two cases require subdividing the triangle into smaller ones with p_r as one of the vertices.

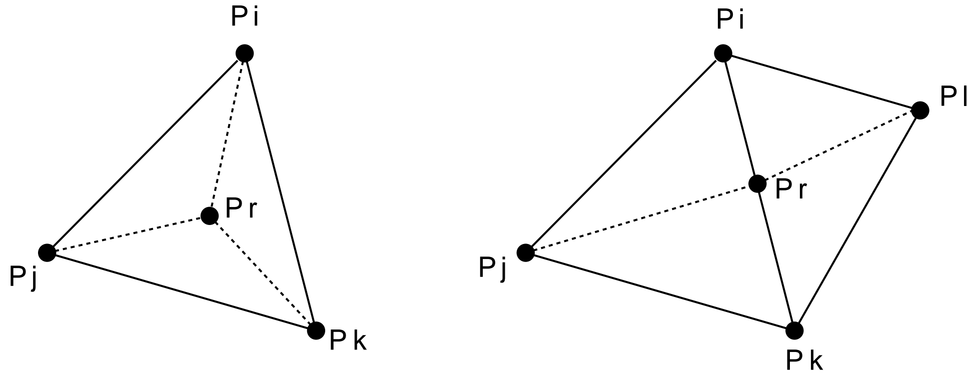


Figure 3.15: Point insertion cases

- If p_r is in a face $\Delta p_i p_j p_k$, we insert the p_r in the triangle and create edges linking p_r to vertices of $\Delta p_i p_j p_k$, see Figure 3.15 left and Algorithm 7.
- If p_r happens to fall on an edge $p_i p_j$ between two adjacent triangles, we split the edge $p_i p_j$ at p_r and create planar edges that link vertices p_l and p_k to vertex p_r, see Figure 3.15 right and Algorithm 8.

Algorithm 7 insertPointInFace(Face $p_i p_k p_j$ Point p_r)

1: create Vertex p_r;
2: connect Vertex p_i and p_r;
3: connect Vertex p_j and p_r;
4: connect Vertex p_k and p_r;

Algorithm 8 insertPointOnEdge(Edge $p_i p_j$ Point p_r)

1: split Edge $p_i p_j$ at Point p_r;
2: connect Vertex p_l and p_r;
3: connect Vertex p_k and p_r;

After the insertion operation, the mesh remains a triangulation with the additional vertex p_r. However, the most recently created triangles might violate the empty circle property and hence the edges incident to the most recently created triangles can become non-Delaunay edges. In the next section, we present the steps to restore the empty circle property using edge flipping and restoring the Delaunay property after a point insertion.

3.2.3 Restoring the Delaunay Property

Every edge is legal before p_r is inserted into the Delaunay triangulation. The edges incident to the changed triangles can be divided into two categories:

- The interior edges. These are the most recently created edges incident to p_r.
- The outer edges. These are the border edges incident to the changed triangles after p_r is inserted.

The interior edges, incident to p_r, are legal edges because we can shrink the originally empty circumcircle C of the triangle to a circumcircle C' of the smaller triangle. Thus, no vertices is in the interior of C'. The outer edges can be illegal and need empty circumcircle checks to ensure it satisfies the Delaunay property. Therefore, we have to check the outer edges and flip the illegal ones to restore the empty circumcircle property (see Algorithm 9 and 10).

Algorithm 9 insertPointInFaceDelaunay(Face $p_i p_k p_j$ Point p_r)

1: create Vertex p_r;
2: connect Vertex p_i and p_r;
3: connect Vertex p_j and p_r;
4: connect Vertex p_k and p_r;
5: {Restore the Delaunay property by recursively flipping the illegal edges. Note that an illegal edge is flipped only once}
6: legalizeEdge($p_i p_j$);
7: legalizeEdge($p_j p_k$);
8: legalizeEdge($p_k p_i$);

Algorithm 10 insertPointOnEdgeDelaunay(Edge $p_i p_j$ Point p_r)

1: split Edge $p_i p_j$ at Point p_r;
2: connect Vertex p_l and p_r;
3: connect Vertex p_k and p_r;
4: {Restore the Delaunay property by recursively flipping the illegal edges. Note that an illegal edge is flipped only once}
5: legalizeEdge($p_i p_l$);
6: legalizeEdge($p_l p_j$);
7: legalizeEdge($p_j p_k$);
8: legalizeEdge($p_k p_i$);

After the edges are flipped, the edges incident to the new triangles may now be illegal and a recursive step is needed for those new outer edges. Algorithm 11 is a recursive implementation of illegal edge flipping procedure.

The point insertion algorithm terminates in the worst case after $O(n)$ steps. [28] [1]. The expected running time, however, is $O(\log n)$ when the points are in general position.

Observe that an edge flip replaces an edge not incident to p_r with one that is incident to p_r. Furthermore, [28] shows that every edge flip creates a legal edge. Therefore, the illegal edges are only flipped once and the legalizeEdge procedure terminates when all the edges are legal.

Algorithm 11 legalizeEdge(Edge $p_i p_j$)

1: {Assume $p_i p_j$ is the diagonal of the quadrilateral $p_r p_i p_l p_j$}
2: **if** $p_i p_j$ is not a boundary edge **then**
3:    **if** CC($\triangle$ $p_i p_j p_r$) contains p_l or CC($\triangle$ $p_i p_j p_l$) contains p_r **then**
4:       flip $p_i p_j$ to $p_r p_l$;
5:       legalizeEdge($p_i p_l$);
6:       legalizeEdge($p_j p_l$);
7:       legalizeEdge($p_i p_r$);
8:       legalizeEdge($p_j p_r$);
9:    **end if**
10: **end if**

Algorithm 12 shows that the point insertion of a Delaunay triangulation consists of one point localization query and the corresponding Delaunay point insertion on the edge or in the face. The constraint insertion operation relies on point insertion procedures in Algorithm 12 to insert the constraint endpoints.

Finally, Algorithm 13 constructs the Delaunay triangulation of point set P. In the next section, we will discuss the constraint insertion algorithm and describe the steps needed to maintain the constrained Delaunay triangulation.

Algorithm 12 insertVertex(int x, int y)

1: Element res := sectorBasedLocatePoint(nullStartFace, Point(x,y));
2: **if** res is $Edge$ **then**
3:    insertPointOnEdgeDelaunay(res,Point(x,y));
4: **else**
5:    insertPointInFaceDelaunay(res,Point(x,y));
6: **end if**

Algorithm 13 constructDelaunayTriangulation(point set P)

1: Initialize the bounding box for P and triangulate it with a diagonal edge.
2: **for all** input points p_i in input set P **do**
3:    insertVertex($p_i.x$, $p_i.y$)
4: **end for**

3.3 Constrained Edge Insertion

In this section, we discuss the implementation of the constrained edge insertion algorithm in the constrained Delaunay triangulation. The implementation follows Anglada's incremental

approach for constraint insertions [1]. We focus on a correct and robust implementation to avoid crashes or incorrect outputs which are not acceptable in many applications. The algorithm proposed by Anglada assumes general position the input data [1]. In our implementation, various modifications to the Anglada's incremental algorithm are introduced to handle the degenerate cases during constraint insertion.

First, we discuss Anglada's incremental algorithm for constraint insertion in constrained Delaunay triangulation. We assume, without losing generality, that all the edges intersected by the inserting constraint are unconstrained edges which can be modified in the triangulation. We will discuss the special case where the inserting constraint intersects with existing constrained edges later. The algorithm inserts a constraint edge with the following steps, see Algorithm 14 Figure(3.16):

Algorithm 14 insertConstraint(Point a, Point b)

1: Insert point a and b in the constrained Delaunay triangulation.
2: Remove unconstrained edges e_i that are intersected by line segment ab.
3: Insert the constraint ab.
4: Retriangulate the polygonal region above and below the constraint ab.

In step 1, we invoke the point insertion algorithm from Algorithm 5. Inserting endpoints a and b in the constrained Delaunay triangulation. The constrained Delaunay property is restored after legalizing the edges with the weak empty circumcircle property.

After the endpoints of $\overline{ab}$ are inserted in the constrained Delaunay triangulation, the algorithm uses a straight-line walk from endpoint a to endpoint b to record the intersecting edges in the triangulation. We need to locate the first triangle t that intersects with ab. This is done by executing the vertex loop around a from Algorithm 1 and testing the intersection between the border edges and line segment ab.

Once the starting triangle is found around endpoint a, the algorithm executes a walk across the shared edge intersecting ab to an adjacent triangle using the intersection test. It keeps walking from triangle to triangle along ab, collecting the intersecting edges $\{e_i\}$ until endpoint b is reached. We can efficiently collect two sets of the vertices $\{pa_i\}$ above and $\{pb_i\}$ below ab in the same pass. The two sets of vertices are needed to re-triangulate the polygonal region in step 4. The intersection test is a geometric function with expensive computation, especially when an arbitrary precision data type is used in the function to ensure the exactness of the geometric computation. We will discuss the modifications to avoid using these expensive geometric functions in the next chapter.

The unconstrained edges do not cross the constrained edges in a planar triangulation, it is safe to remove the unconstrained edges in $\{e_i\}$ from the triangulation which results in an

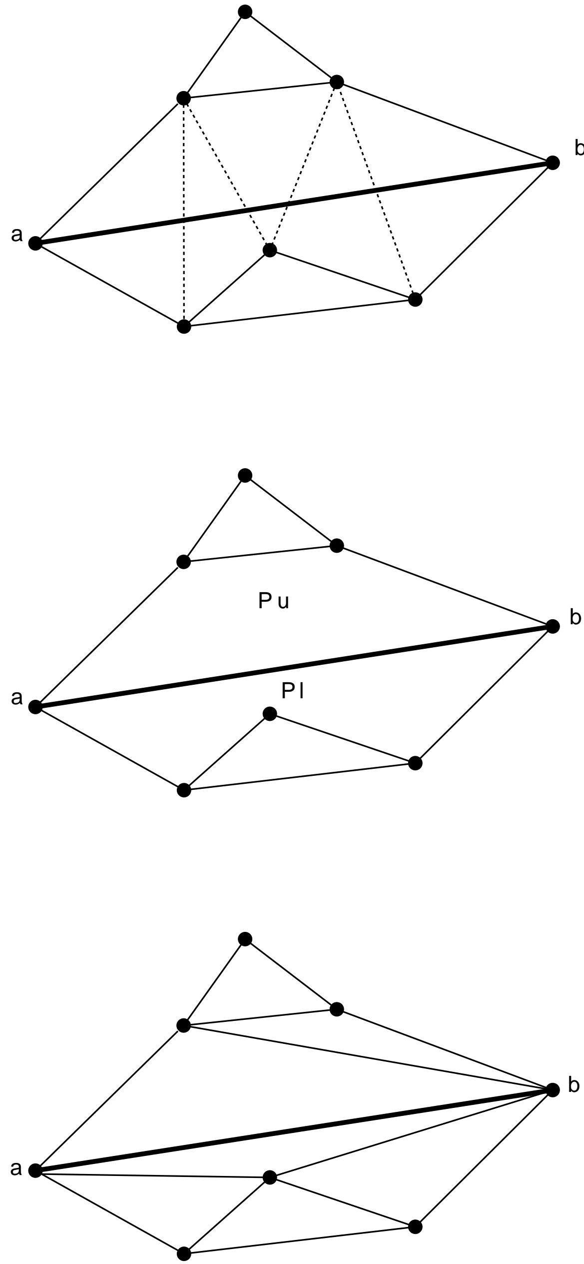


Figure 3.16: Constraint insertion

open regions around the insertion. This completes the operations in step 2.

Step 3 partitions the open polygonal region from step 2 into the upper and lower half separated by ab. A unique constraint ID for ab is added to the ID list of ab making it

constrained. Each constrained edge maintains a list of unique constraint IDs for dynamic updates and for special cases in the constrained Delaunay triangulation such as overlapping. The ID list of a constraint line segment that represents overlapping between two constraints contains two constraint IDs. Then, the upper and lower half of the open region will be re-triangulated in step 4.

The edges in the exterior of the open polygonal region are the same as those in the constrained Delaunay triangulation before the constraint insertion. Therefore, the triangles in the exterior still satisfy the weak empty circumcircle property for constrained Delaunay triangulation. Thus, no changes are needed for the exterior of the open polygonal region. In step 4, the interior of the upper and lower open region are simple polygons. Therefore, a simple polygon triangulation algorithm observing the empty circumcircle property is sufficient to re-triangulate the upper and lower open region. For instance, the re-triangulation process recursively identifies the vertex c $\in \{pa_i\}$ in the upper open region, forming $\triangle abc$ satisfying the empty circumcircle property. The process splits the simple polygon into a triangle and two sub open regions PE and PD to be re-triangulated (see Figure(3.17)). The re-triangulation process terminates when the sub regions are reduced to triangles. The same process can be used to re-triangulate lower open region.

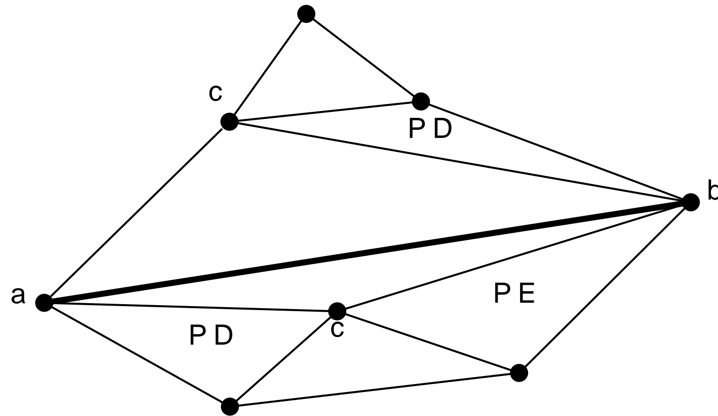


Figure 3.17: Subregions to be re-triangulated

In order to have a robust implementation, we need to pay attention to the special cases that may arise:

- The inserted endpoints a and b are connected by an unconstrained edge. A vertex loop

around a with intersection test will not find the starting triangle and might result in an infinite loop if this special case is not dealt with. For this special case, we can simply identify the unconstrained edge connecting $\overline{ab}$, insert the constraint ID to the constraint ID list of $\overline{ab}$, hence complete the constraint insertion.

- Another similar case occurs when the constrained line segment $\overline{ab}$ to be inserted overlaps an existing edge. For this case, we can also identify the overlapping portion of the edge, insert the constraint ID to the constraint ID list of the overlapping edge and recursively insert the remaining non-overlapping portion.

- A last degenerate case is that the constrained line segment $\overline{ab}$ intersects with existing constraints or vertices. We identify the intersecting constraints or vertices during the straight-line walk in step 1, insert the intersection point on the constrained edges using point insertion. We break up the insertion of $\overline{ab}$ into small constrained intervals between intersection points. We can iteratively insert the individual small constrained intervals using Algorithm 14 because the edges intersected by those constrained intervals are unconstrained edges.

The worst case run time of our implementation is the same as Anglada's incremental algorithm while the robustness of our implementation is greatly improved by carefully handling the special cases. Let e be the number of triangles in constrained Delaunay triangulation intersected by the constrained line segment ab. The dominating step is the re-triangulation of the open polygonal regions. The total number of vertices in the open regions decreases by one after each recursive call for simple polygon triangulation, creating O(e^2) triangulation steps. Therefore, the worst case run time of the incremental constraint insertion is O(e^2).

3.4 Vertex Removal

In order to support constrained Delaunay triangulation dynamically, we need to provide the removal procedures. The vertex removal process follows M. Kallmann's implementation [23].

Vertex removal is important for maintaining a dynamic constrained Delaunay triangulation, it is especially important for constrained polygon removals. In this section, we present the step for vertex removal in constrained Delaunay triangulation. During the vertex removal process, the following case should be identified and handled in order to maintain a fully dynamic triangular mesh:

1. The removing vertex is incident to only unconstrained line segments.

2. The removing vertex is an endpoint of a constrained line segment.

3. The removing vertex is the intersection of two or more constrained line segments.

The first two cases are trivial. For case 1, we first remove the unconstrained edges incident to the vertex, then remove the vertex and re-triangulate the open region. For case 2, we should leave the vertex in the triangulation because it is one of the endpoints of a constrained line segment. Case 3 deals with the steiner vertices after a constraint removal. We remove the steiner vertex if there is only one constrained line segment containing it (see Figure(3.18)). The detail of the implementation is presented in Algorithm 15.

Algorithm 15 removeVertex(Vertex v)

1: loop through the edges incident to v, count and record the constrained ones.
2: **if** constrained edge count = 0 **then**
3: Remove the vertex v and its incident edges.
4: Re-triangulate the open region.
5: **return**
6: **end if**
7: **if** constrained edge count = 2 and the two constraints e_1 e_2 are aligned **then**
8: Let v1 be the endpoint of e_1 opposite of v.
9: Let v2 be the endpoint of e_2 opposite of v.
10: Remove the vertex v and its incident edges.
11: Re-triangulate the open region.
12: Insert constraint e_3 with v1 and v2 as its endpoints.
13: Reset constraint IDs of e_3.
14: **end if**

3.5 Constrained Polygon Insertion and Removal

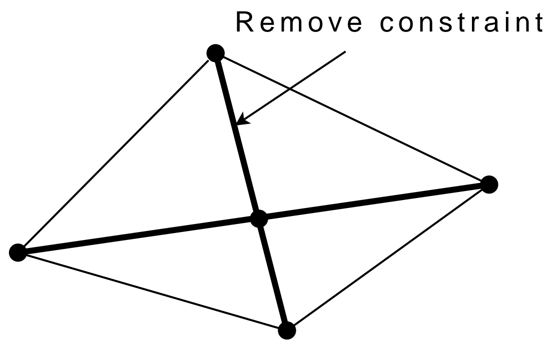
In this section, we describe the process of constrained polygon insertion and removal.

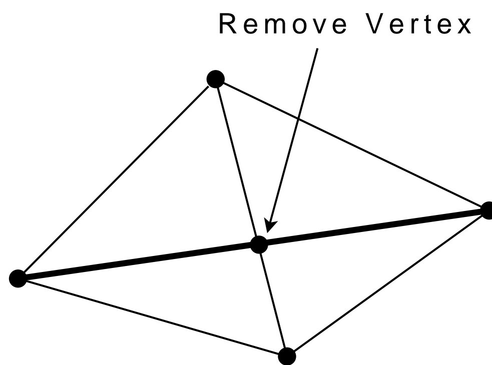
The polygon insertion is done by inserting the edges of a polygon as constraints iteratively and associates the constrained edges with the same ID.

To remove a constrained polygon, we need to identify edges $\{E\}$ and vertices $\{V\}$ associated with the constrained polygon. We then remove constrained edges by removing the constraint ID from the ID list of the corresponding edges. The overlapping portion of a constraint will remain as a constrained edge with one of its IDs removed. On the other hand, a constrained edge becomes unconstrained when the list of IDs is empty and will be removed along with its endpoints. Algorithm 17 shows the constraint removal procedure:

Algorithm 16 removeConstraint(Edge e, int id)

1: remove id from e.IDlist;





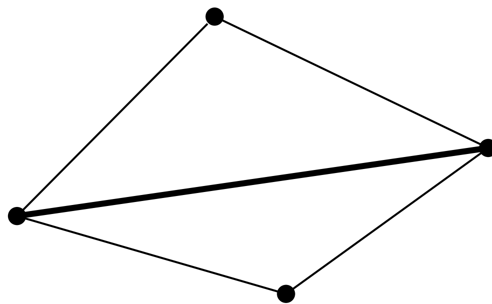


Figure 3.18: Case 3 of vertex removal

Vertices were added during constraint insertions as intersection points with other constraints. Removing vertices associated with the constrained polygon using Algorithm 15 ensures redundant vertices are removed while the necessary intersection points of constraints remain in the triangulation.

To remove a constrained polygon with the *id*, we need to remove *id* from the constrained polygon edges and remove all the intermediate vertices added during constraint insertions. The following procedure summarizes the process:

Algorithm 17 removePolygon(int *id*)

1: identify and collect constrained edges E with *id* from the triangulation.
2: identify and collect incident vertices V for the polygon
3: **for all** constrained edges e_i in E **do**
4: removeConstraint(e_i, *id*)
5: **end for**
6: **for all** constrained vertices v_i in V **do**
7: removeVertex(v_i)
8: **end for**

3.6 Conclusion

In this chapter, we describe the algorithms for dynamic updates in constrained Delaunay triangulation. First, we discuss different geometric data structures. For point insertion, we survey different point localization algorithms and analyze their advantages and disadvantages for dynamic point insertion. We proposed sector-based approach with efficient sector updates for point localization in point insertion operations. We show the degenerate cases for constraint insertions and changes needed to handle them. Finally, we discuss the steps for dynamic updates for constrained polygons.

Chapter 4

Robust Geometric Computations

The correctness of geometric algorithms relies on numerically robust geometric functions to correctly perform decision making on the input data. The geometric functions receive mesh coordinates as inputs, encapsulate geometric decisions and return a set of discrete values for each geometry case. Standard floating point arithmetic is used for many triangulation applications. However, rounding errors in floating point arithmetic can cause incorrect return values. Incorrect results can cause crashes, infinite loops or incorrect mesh outputs. This is a well known issue in computational geometry [26].

Many geometric applications tolerates inexactness in fixed-precision floating point arithmetic because of its speed advantage. However, a theoretically correct geometry algorithm may have inconsistent outputs with a inexact implementation. Therefore, exact computation is an important for robust geometric algorithm implementation.

Triangulation software using fixed-precision floating point arithmetic includes:

- The $Triangle$ package [35] which uses adaptive filters to evaluate $Orientation$ and $InCircle$ tests exactly. However, the $Itersection$ calculation of the new vertices is not exact which cause incorrect outputs or crashes.
- The DCDT package [23] uses ϵ-tolerance in the computation of geometric functions. Therefore, the package does not use exact computation.

Illustrative examples of failures in geometric algorithms with floating point implementations can be found in [26].

Common methods proposed to address this computational inconsistency problem in computational geometry include:

- Exact computations with arbitrary-precision data types.

- Numeric filters with exact error bounds.

- Extended algorithms that tolerate numerical inaccuracies to a certain degree.

In this chapter, we discuss computational inconsistencies caused by floating point arithmetic and address the inconsistency problem in our implementation with exact computation. We also analyze a hybrid method which improves computation speed while maintaining computational correctness using arbitrary-precision arithmetic, dynamic filtering and extended algorithms.

4.1 Fundamental Geometric Functions

In this section, we show that arithmetic accuracy is critical in geometric function evaluation for distinguishing degenerate inputs because only the exact computation of a zero determinant can guarantee the correct decision.

The incremental algorithms for computing a dynamic constrained Delaunay triangulation described in Chapter 3 rely on the following fundamental geometric functions:

- The *Orientation* function that receives the coordinates of three vertices in the mesh and determines whether they are in clockwise, counterclockwise order or collinear. The function is used in point localization algorithm (Section 3.2.1) and constraint insertion (Section 3.3).

- The *InCircle* function that receives the coordinates of four vertices in the mesh and determines whether the fourth vertex is inside of, outside of or on the circumcircle defined by the first three vertices. The function is used in empty circumcircle tests for edge flips (Section 3.2.2).

- The *Intersection* function that receives the coordinates of the endpoints of two intersecting line segments and computes their intersection coordinates. The function is used in constraint insertions(Section 3.3).

4.1.1 The Orientation Function

The *Orientation* function can be implemented in terms of finding the sign of the determinant with vertex coordinates as matrix elements. Let a, b, c be three vertices in the plane with $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$. Then the orientation of c with respect to line segment $\overline{ab}$ can be determined with the following calculation:

$$Orientation(a,b,c) = sign\left(\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}\right) \tag{4.1}$$

The sign of the determinant in Equation (4.1) specifies whether vertex c is to the left of, to the right of, or on the line formed by a and b. In other words, vertices a, b and c are in counterclockwise orientation if the determinant is greater than zero, the vertices are in clockwise orientation if the determinant is less than zero, and the vertices are collinear if the determinant is zero [24].

4.1.2 The InCircle Function

The InCircle function can also be implemented in terms of finding the sign of the determinant with vertex coordinates as matrix elements. Let a, b, c, d be four vertices in the plane with $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$, $d = (d_x, d_y)$. Then the following calculation determines whether d is in the circumcircle defined by the vertices a, b, and c:

$$InCircle(a,b,c,d) = sign\left(\begin{vmatrix} 1 & a_x & a_y & a_x^2 + a_y^2 \\ 1 & b_x & b_y & b_x^2 + b_y^2 \\ 1 & c_x & c_y & c_x^2 + c_y^2 \\ 1 & d_x & d_y & d_x^2 + d_y^2 \end{vmatrix}\right) \tag{4.2}$$

The $InCircle$ computation in Equation (4.2) states whether a vertex d lies inside, outside or on the CC($\triangle abc$) defined by the vertices a, b, and c. Assume that a, b, c are on counterclockwise ordering, the sign of the determinant is less than zero if vertex d is located inside CC($\triangle abc$), the sign is greater than zero if vertex d is located outside the boundary of CC($\triangle abc$), and the sign is zero if d lies on CC($\triangle abc$) [24].

4.1.3 The Intersection Function

Two line segments $\overline{ab}$ and $\overline{cd}$ are intersecting when they cross each other. The intersection can be determined mathematically by analyzing the endpoints of the line segments. The intersection can be a proper intersection where the two line segments share exactly one point and the intersection point lies in the interior of both segments, or an overlapping line segment where two line segments are parallel and share a portion of their interior.

The coordinates of the proper intersection point can be formulated as the determinant computation with coordinates of a, b, c, d in matrix form. Let $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$, $d = (d_x, d_y)$. Then the following calculation computes the intersection coordinates between the lines defined by $\overline{ab}$ and $\overline{cd}$:

$$I_x = \frac{\begin{vmatrix} \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} & \begin{vmatrix} a_x & 1 \\ b_x & 1 \end{vmatrix} \\ \begin{vmatrix} c_x & c_y \\ d_x & d_y \end{vmatrix} & \begin{vmatrix} c_x & 1 \\ d_x & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} a_x & 1 \\ b_x & 1 \end{vmatrix} & \begin{vmatrix} a_y & 1 \\ b_y & 1 \end{vmatrix} \\ \begin{vmatrix} c_x & 1 \\ d_x & 1 \end{vmatrix} & \begin{vmatrix} c_y & 1 \\ d_y & 1 \end{vmatrix} \end{vmatrix}} \tag{4.3}$$

$$I_y = \frac{\begin{vmatrix} \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} & \begin{vmatrix} a_y & 1 \\ b_y & 1 \end{vmatrix} \\ \begin{vmatrix} c_x & c_y \\ d_x & d_y \end{vmatrix} & \begin{vmatrix} c_y & 1 \\ d_y & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} a_x & 1 \\ b_x & 1 \end{vmatrix} & \begin{vmatrix} a_y & 1 \\ b_y & 1 \end{vmatrix} \\ \begin{vmatrix} c_x & 1 \\ d_x & 1 \end{vmatrix} & \begin{vmatrix} c_y & 1 \\ d_y & 1 \end{vmatrix} \end{vmatrix}} \tag{4.4}$$

Equation 4.12 and 4.13 are equivalent to solve for I_x and I_y simultaneously with the following equations:

$$\begin{vmatrix} I_x & I_y & 1 \\ a_x & a_y & 1 \\ b_x & b_y & 1 \end{vmatrix} = 0 \tag{4.5}$$

$$\begin{vmatrix} I_x & I_y & 1 \\ c_x & c_y & 1 \\ d_x & d_y & 1 \end{vmatrix} = 0 \tag{4.6}$$

Therefore, the coordinates of the intersection can be calculated as the following:

$$\beta = (d_y - c_y)(a_x - b_x) - (a_y - b_y)(d_x - c_x) \tag{4.7}$$

$$\alpha_x = (d_y - c_y)(d_x - b_x) - (d_x - c_x)(d_y - b_y) \tag{4.8}$$

$$\alpha_y = (d_y - b_y)(a_x - b_x) - (d_x - b_x)(a_y - b_y) \tag{4.9}$$

$$c_x = \frac{\alpha_x}{\beta} \tag{4.10}$$

$$c_y = \frac{\alpha_y}{\beta} \tag{4.11}$$

$$I_x = b_x + c_x(a_x - b_x) \tag{4.12}$$

$$I_y = b_y + c_y(a_y - b_y) \tag{4.13}$$

The equations apply to lines, so the line segment intersection needs the following degeneracy checks to ensure the intersection is a proper intersection:

- β in Equation 4.7 is zero if the two lines are parallel.
- β in Equation 4.7, α_x in Equation 4.8 and α_y in Equation 4.9 are zero if two lines are collinear.
- c_x and c_y are between 0 and 1 if the intersection point lies in the interior of the corresponding line segments.

The exact computation of the intersection coordinates is essential for correct geometric computations. Some of the algorithms only need to detect the intersection between two line segments. The *Intersection* function can detect whether two line segments intersect by testing the orientation of the endpoints using *Orientations* tests. The two line segments $\overline{ab}$ and $\overline{cd}$ are intersecting if and only if:

- The endpoints a and b are on opposite sides of the line $\overline{cd}$, and
- The endpoints c and d are on opposite sides of the line $\overline{ab}$.

4.2 Floating Point Rounding Errors

In this section, we discuss the computation inaccuracy caused by finite-precision data types and how they affect geometric computations.

Various representations for floating point numbers have been proposed. One of the commonly used ones is to represent a floating point number by a sign s, a significand m , a base b and an exponent e. The significand m is assumed to be normalized such that the leftmost digit of m is nonzero and the radix point is to the right of the leftmost digit. The value can be calculated with the following formula:

$$v = (-1)^s \times m \times 2^e$$

For example, a binary floating point number 1100.1100 can be represented by $s = 0$ $m = 1.1001100$ and $e = 3$ with 8 digit precision using $b = 2$.

The IEEE standard for binary floating point arithmetic (IEEE 754-1985) is the most widely used standard for floating point arithmetic on modern hardware. The standard provides two data types:

- The single-precision ($32bit$) with 1 sign bit, 8-bit exponent and 23-bit significand.

- The double-precision ($64bit$) with 1 sign bit, 11-bit exponent and 52-bit significand.

Because the standard floating point representation uses fixed precision, the floating point numbers can represent only a subset of the rational numbers, and the approximation of decimal values using binary floating point representation can introduce rounding errors. For example, it is not possible to express the decimal number 0.1 accurately because its binary representation is the non-terminating binary floating point number:

$$0.000110011001100110...$$

This number must be rounded to the nearest representable floating point value for hardware storage and computations. In single-precision floating point number with base 2 ($b = 2$) representation and 24 digit precision this is:

$$m = 1.10011001100110011001101, \ e = -4$$

Therefore, the decimal value of the closest representable floating point value is

$$1.60000002384185791015625 \times 2^{-4} = 0.100000001490116119384765625$$

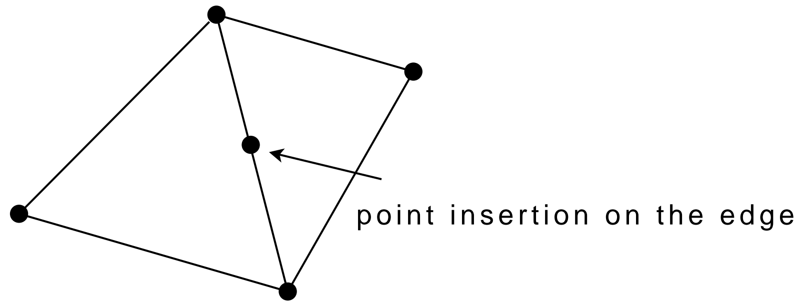
.

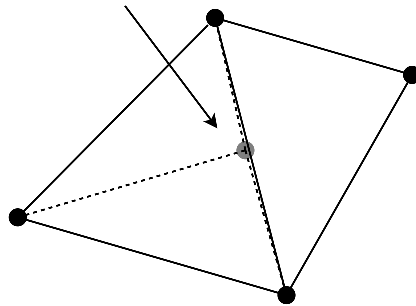
Any finite-precision floating point arithmetic can cause incorrect geometric computations. Also, small round-off errors can accumulate and propagate in sequence of arithmetic operations and introduce big inaccuracy for some computations. For instance, consider the $Orientation(a, b, c)$ computation of the Orientation function from Equation (4.1):

$$Orientation(a, b, c) = sign\left((a_x \times b_y) - (b_x \times a_y) + (b_x \times c_y) - (c_x \times b_y) + (c_x \times a_y) - (a_x \times c_y)\right)$$

For the $Orientation$ and $InCircle$ function, the numerical error introduced by floating point arithmetic affects the final result of the determinant computation. Both the determinants in the collinear and the co-circular case are zero. However, using floating point arithmetic to compute such determinants may produce a non-zero floating point value with the accumulated round-off error.

The return value of the $Intersection$ function is also affected by arithmetic inaccuracy. For example, the computed intersection coordinates may be shifted due to accumulated rounding errors, which causes the algorithm to incorrectly insert the intersection points in a face close to the intersecting constraints (see Figure 4.1).





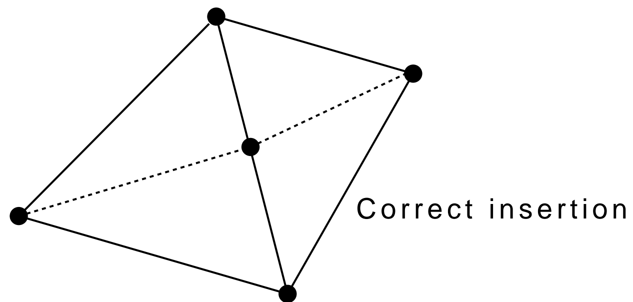


Figure 4.1: Incorrect point insertion due to rounding error in geometric function

Algorithm 18 Equal(double a, double b)

```
1: if (|a − b| ≤ ϵ) then
2:     return true
3: else
4:     return false
5: end if
```

One way to mitigate the effects of numerical inaccuracy in finite-precision floating point arithmetic is to introduce a error tolerance ϵ and to replace the equality tests for two floating point values by:

Implementations of geometric algorithms can tweak the ϵ value and introduce perturbation in the inputs to achieve an acceptable level of robustness. However, such ad-hoc measures still can produce catastrophic failures such as program crashes, useless outputs or infinite loops.

4.3 Exact Geometric Computation

A principled way to resolve the numerical inaccuracy problem is to adopt arbitrary-precision computations. In this section, we discuss the exact computation method used in our implementation and the computational costs associated with this approach. We also discuss the dynamic filtering scheme which reduces the amount of expensive arbitrary-precision operations while still generating correct results. Finally, we introduce various modifications on existing algorithms to further increase the computation speed.

4.3.1 Arbitrary-Precision Libraries

An arbitrary-precision library provides numerical data types with unbound precision. Arbitrary-precision packages usually support the following data types:

- Integers
- Rational numbers
- Floating point numbers

These numerical data types are usually represented by integer vectors which are used by arithmetic operations. The precision of the data types is normally limited only by the amount of available memory in the system. The most commonly used libraries for arbitrary-precision data types and arithmetic are:

- The GNU Multiple Precision Arithmetic Library (GMP) which provides general-purposes arbitrary-precision computations with relatively efficient implementation.

Various optimizations using fast algorithms and assembly code increase the computational speed for arbitrary-precision computations. However, the computation speed of these libraries are still much slower than built-in data types.

- The Library of Efficient Data Types and Algorithms (LEDA) which provides a collection of fixed-precision and arbitrary-precision data types for inexact and exact geometric computations. The speed of arbitrary-precision computation in LEDA is slow compared to the GMP library and most of the implementations of geometric algorithms assume static inputs with little support on dynamic updates.

Our implementation limits the triangulation input to integer coordinates which can be easily adapted because floating point inputs can be multiplied by a large constant factor and processed as integer inputs. The integer coordinate requirement allows us to bound the size of vertex coordinates so we can represent of new intersection point using fixed-precision rational coordinates. The intersection computation can then be formulated as a sequence of integer additions and multiplications, and one division between two integers at the end (see computations Equation 4.7 - 4.13).

Therefore, the coordinates of any vertex in our constrained Delaunay triangulation can be represented exactly by rational numbers, having numerators and denominators in the form of regular integer types. The arbitrary-precision GMP::rational implementation of the geometric predicates then guarantees exact computation without rounding errors or overflows.

The computational cost of arbitrary-precision operation is high compared to the built-in data types due to memory management costs and quadratic complexities for some arithmetic operations. The implementation of geometric algorithms using arbitrary-precision data types like LEDA::rational or LEDA::real is 100 times slower than using the floating point double for large mesh generation. Therefore, the arbitrary-precision arithmetic should be used as a last resort. In the next section, we discuss a fast and efficient dynamic filtering mechanism which can be used to determine the sign of an expression quickly and accurately.

4.3.2 Interval Arithmetic

Computing the determinant using arbitrary-precision arithmetic is expensive. In this section, we discuss dynamic filters as means of restricting the precision needed for arithmetic operations while still computing the exact result. The types of numerical filters include:

- Static filters.

- Semi-static filters.

- Dynamic filters.

The error bounds of static and semi-static filters are determined at compile-time. Static filters are restricted to integral, division-free expressions of small bounded depth with the requirement to preprocess good upper and lower bounds on the input. Semi-static filters solve most of the problems in static filters, but divisions and square roots can only be handled with significantly larger error bounds. On the other hand, The error bounds of dynamic filters are determined at run-time without the restrictions on filter operations and inputs.

Using a dynamic filter can evaluate the sign of a determinate exactly. In [24], Karasick et al. introduce an adaptive filter mechanism to evaluate the sign of a determinant using dynamic filter. The dynamic filter progressively increases its computation accuracy, so the geometric function can evaluated exactly and efficiently for non-degenerate cases where the determinate is non-zero. We proposed a different approach using a combination of dynamic filter and arbitrary-precision arithmetic to compute the exact sign of the determinant in geometric function evaluation.

Interval arithmetic can be used as a dynamic filter for geometric function evaluations. It deals with intervals $[\underline{p}, \overline{p}]$ bounding real number p. For floating point interval arithmetic, $\underline{p}$ is rounded down and $\overline{p}$ is rounded up to the nearest machine-representable number to ensure proper bounding of the interval.

The arbitrary-precision arithmetic is used only in degenerate geometric cases, where the computed interval $[\underline{p}, \overline{p}]$ is inconclusive for the sign of the determinant.

Each input coordinate represented by an rational number $\frac{pn}{pd}$ is first converted into an interval $[Pl, Pu]$ such that $\frac{pn}{pd} \in [Pl, Pu]$. $\frac{pn}{pd}$ is approximated as an floating point interval with the following steps:

- Extract k significant bits from pn and pd to form integer sn and sd. pn and pd are padded with 0 bits if they have less than k significant bits.
- Construct interval $[sn, sn + 1]$ and $[sd, sd + 1]$
- Compute the floating point interval from $[sn, sn + 1] \,/\, [sd, sd + 1]$
- Adjust the floating point ratio according to the bit length difference between pn and pd.

The following example is an approximation with k=24 significant bits:

$$\begin{aligned}\frac{485719}{32431} &\sim \frac{[15543008, 15543009]}{[16604672, 16604673]}\\ &\sim [0.93606227596291713, 0.93606239256035895]\\ &= [14.976996415406674, 14.976998280965743]\end{aligned}$$

Then the exact rational arithmetic operations in the geometric function are replaced by interval operations with basic operations for intervals $[x] = [\underline{x}, \overline{x}]$ and $[y] = [\underline{y}, \overline{y}]$:

$$\begin{aligned}[x] \oplus [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}]\\ [x] \ominus [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}]\\ [x] \otimes [y] &= [\min(\overline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \underline{x} \cdot \underline{y}), \max(\overline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \underline{x} \cdot \underline{y})]\\ [x] \oslash [y] &= [x] \otimes [1/\overline{y}, 1/\underline{y}] \qquad 0 \notin [y]\end{aligned}$$

Our implementation uses the interval arithmetic library from the Boost library for interval computations. The library enables us to overload the computation in geometric functions using interval arithmetic. Therefore, the implementation of geometric functions does not require any change.

Finding the sign of the determinant of a matrix M can be formulated as a function $F(M)$ that maps matrices M to $\{-1, 0, 1\}$. Computing the actual value of the determinant directly using arbitrary-precision arithmetic is expensive. To increase the speed of the computation, we can compute M's interval representation M' and determine the sign of its determinant interval $F'(M') = [d_{min}, d_{max}]$ using interval arithmetic. If $d_{min} > 0$ then $F(M) > 0$ and if $d_{max} < 0$ then $F(M) < 0$. The computation falls back to exact computations when $0 \in [d_{min}, d_{max}]$ For example:

$$\begin{aligned}M &= \begin{pmatrix} \frac{485719}{32431} & \frac{-230337}{5269} \\ \frac{596157}{22949} & \frac{100130}{5141} \end{pmatrix}\\ M' &= \begin{pmatrix} [14.976996415406674, 14.976998280965743] & [-43.715508754033024, -43.715501737434074] \\ [25.977469574393425, 25.977474508693192] & [19.476753645179222, 19.476757014685859] \end{pmatrix}\\ d_{min} > 0 &\Rightarrow F(M) > 0\end{aligned}$$

Dynamic filtering increases the computation speed for frequently used geometric functions while maintaining exactness. The speed improvement is mainly due to the fact that degenerate cases with zero determinants are relatively rare. Therefore, the exact result can be determined by the efficient interval arithmetic.

A further speed improvement can be achieved by caching the computed intervals of the coordinates. It is not necessary to recompute the interval approximations because the coordinates of the vertex do not change.

In this section, we discussed the dynamic filtering technique combined with exact computation for correct geometric function evaluations. The runtime of such a combination is approximately five times slower than the fixed-precision floating point implementation. The fixed-precision floating point implementation, however, can not guarantee correctness. In the next section, we will discuss the memory management cost in the arbitrary-precision library and various ways to reduce it.

4.3.3 Extended Algorithm

After applying various optimizations to the implementation for arbitrary-precision arithmetic using exact data types, tests indicated that memory allocation and deallocation for arbitrary-precision data was the most time consuming part of the execution.

The input data of the triangulated mesh can be easily limited to bounded integer coordinates and it is sufficient to use a fixed-precision rational data type to represent vertex coordinates. We use rational number with 64-bit integer (long long in C++) for numerators and denominators to represent vertex coordinates. The arithmetic precision of our implementation is limited by the intersection coordinate calculation of two intersecting line segments involving rational arithmetic. Rational coordinates arise when computing the intersection coordinates of two line segments with integer inputs. We decrease the data precision required in the $Intersection$ function by allowing only integer coordinates as inputs. The intersection coordinates between two intersecting constraints can be computed using their integer endpoints (see Figure 4.2). The integer inputs for the $Intersection$ computation reduce the rational arithmetic in $Intersection$ calculation to integer arithmetic. Our implementation stores the leftmost and rightmost integer endpoints for each constrained line segment and uses them for intersection computations. Therefore, the intersection between intersecting constraints can be computed with fixed-precision integer arithmetic on relatively large integer grids. With integer coordinates of k-bit length forming integer coordinates range of $[0 \ldots M = 2^k]$, a multiplication of two integer numbers requires twice as many bits and an addition requires an additional bit to represent the computed result. The computation of I_x and I_y in Equation 4.7-Equation 4.12 has an integer range of the following:

$$
\begin{aligned}
|\beta| &\leq 2M^2 &(4.14)\\
|\alpha_x| &\leq 2M^2 &(4.15)\\
|\alpha_y| &\leq 2M^2 &(4.16)\\
|I_x| &= \frac{|b_x\beta + \alpha_x(a_x - b_x)|}{|\beta|} &(4.17)\\
|b_x\beta + \alpha_x(a_x - b_x)| &\leq M \times 2M^2 + 2M^2 \times M = 4M^3 &(4.18)\\
|I_y| &= \frac{|b_y\beta + \alpha_y(a_y - b_y)|}{|d|} &(4.19)\\
|b_y d + \alpha_y(a_y - b_y)| &\leq M \times 2M^2 + 2M^2 \times M = 4M^3 &(4.20)
\end{aligned}
$$

Therefore, the coordinates of integer input for our implementation is limited to $[0 \ldots 2^{20}]$ with 64 bit integers:

$$
\begin{aligned}
4M^3 &\leq 2^{63}\\
M &\leq 2^{20}
\end{aligned}
$$

Our implementation will switch to arbitrary-precision arithmetic for input size greater than 2^{20}.

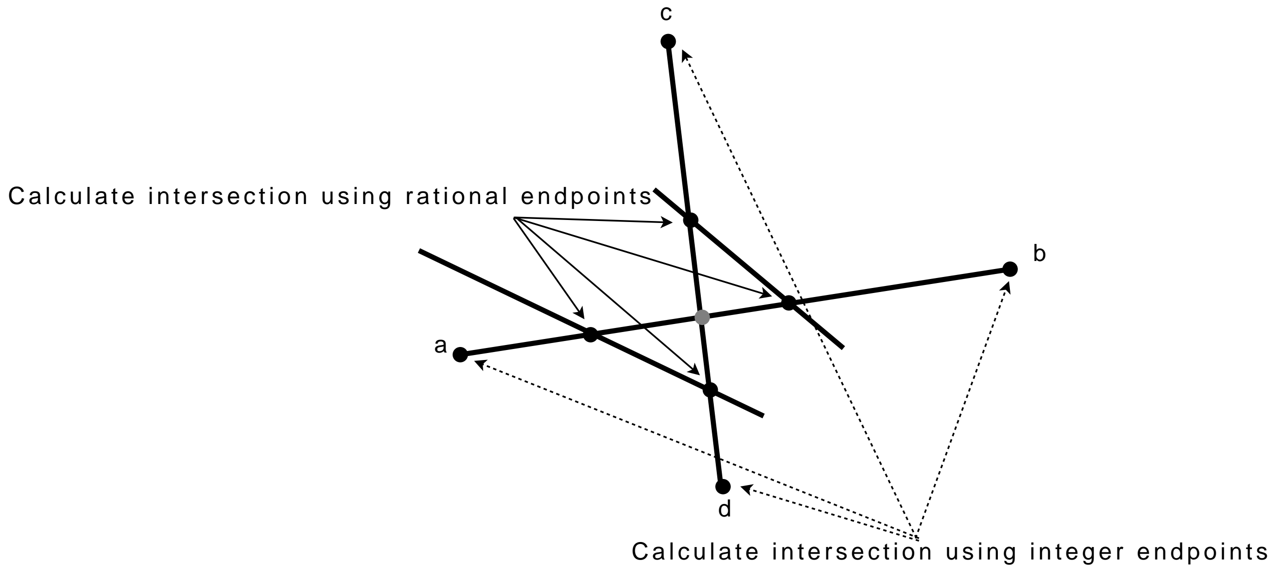


Figure 4.2: Calculating intersection coordinates using integer endpoints

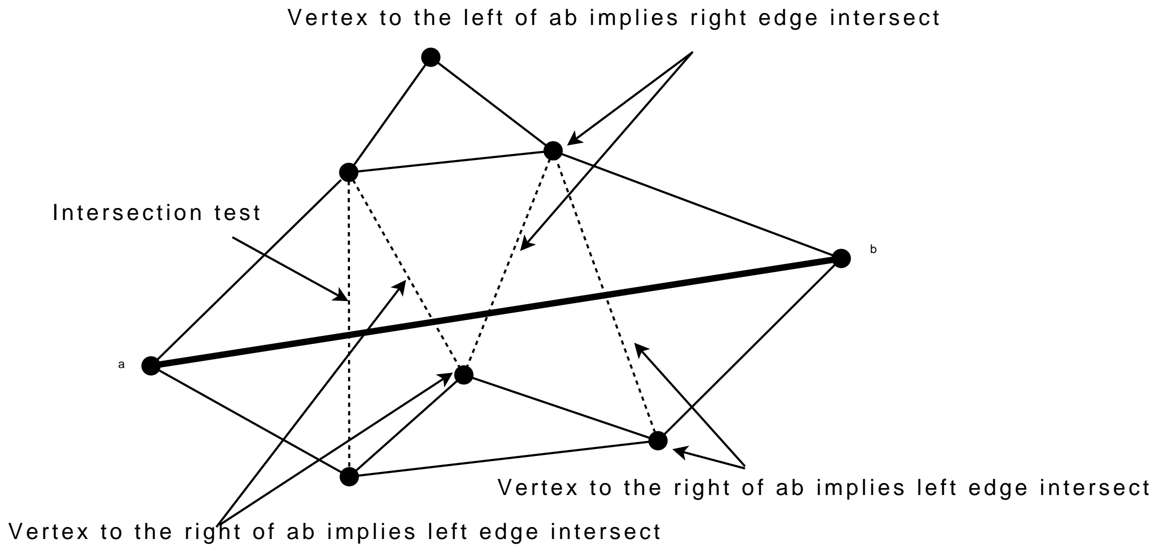


Figure 4.3: The replacement of intersection tests in constraint insertion

The constraint insertion algorithm in Section 3.3 needs to identify the intersection between the inserting constraint and existing line segments in the triangulation. We can implement the intersection test without computing the intersection coordinates. The exact computation of such test is only necessary for locating the start triangle, or to be specific, the first line segment intersected by the inserting constraint. Subsequent line-segment intersection tests are not necessary, and can be replaced by the less expensive orientation tests. The left or right orientation test on the opposite vertex of the current triangle, with respect to the inserting constrained line segment, implies an intersection (see Figure 4.3). The change is effective because the algorithm only traverses adjacent triangles and the computation for non-intersecting line segments requires only two *Orientation* tests. As a result, it avoids the expensive intersecting evaluation by resorting to the simple orientation tests.

4.3.4 Conclusion

In this chapter, we analyze the numerical rounding errors in fixed-precision floating point arithmetic and introduce a combination of dynamic filter and arbitrary-precision arithmetic to compute the geometric functions exactly. Finally, we introduce a method to compute the intersection using integer arithmetic.

Chapter 5

Experiments

This chapter discusses the experiments conducted to evaluate the performance of update operations between Kallmann's inexact implementation and our exact implementation.

Our implementation of the constrained Delaunay triangulation algorithms went through three major revisions. First, the software was implemented assuming integer inputs in general position. The integer arithmetic in this implementation was then relaxed to fixed-precision rational arithmetic for intersection computation. The focuses of the first version of our application were the correct implementation of triangulation algorithms and the detection and handling of degenerate cases such as collinear vertices, cocircular vertices and overlapping line segments. This implementation is limited to small map sizes because of overflow problems in geometric function computation. The second revision of our implementation uses arbitrary-precision arithmetic in the GNU Multi-Precision Library (GMP) [17] to resolve the overflow problems. This version suffers significant loss of computation speed due to high computational cost in arbitrary-precision computation. The execution time of constraint insertion is approximately 50 to 100 times slower than that of Kallmann's floating point implementation in his DCDT package. The third revision of our implementation focuses on reducing the computational cost in arbitrary-precision arithmetic, while maintaining correctness of the geometric functions. We evaluate the slow down factor of the update operations between our exact implementation and Kallmann's inexact implementation. The use of dynamic filter, caching and extended algorithms reduces the slow down factor to 6 for updating constrained polygon in general position.

We show the performance of the update operations implemented using dynamic filter and exact arithmetic. Experimental results indicate that the dynamic filter significantly reduces the number of arbitrary-precision arithmetic using low precision bound interval arithmetic to compute the exact result of the geometric function. The combination of dynamic filtering and exact computation provide us efficient and robust computation of the

geometric functions that are used in constrained Delaunay triangulation algorithms. The software is a single-threaded cross-platform C++ implementation. In all experiments, we use 32-bit Windows XP professional system (service pack 3 build 2600) with Intel Centrino Duo 2GHz CPU and 2.5GB memory (2GB+500MB PC5300). The software has been developed and debugged in Microsoft Visual Studio 2005 Team Suite on 32bit Windows XP professional and ported to 64-bit Fedora10 and compiled using gcc 4.3.0 and -O3 optimization level.

Experiments in this chapter focus on computational efficiency of update operations in constrained Delaunay triangulation with different arithmetic:

- *Double*: the standard double-precision floating point arithmetic [39].
- *Exact*: the arbitrary-precision rational arithmetic using GMP library [17].
- *Interval + Exact*: the combination of double-precision floating point interval arithmetic using Boost Interval Arithmetic Library [5] and the arbitrary-precision rational arithmetic using GMP.

We analyze the performance of Kallmann's DCDT software package as the baseline measurement for fixed-precision floating point implementation. The DCDT software implements the constrained Delaunay triangulation algorithms with dynamic updates using *Double* arithmetic and ϵ-tolerances for degenerate computations in geometric functions[23]. Furthermore, Kallmann's DCDT implementation follows the Topology Oriented implementation paradigm as proposed by Sugihara et al.[37] which minimizes the dependency between the combinatorial and numerical part of an algorithm to achieve certain degree of robustness. Our implementation of the constrained Delaunay triangulation algorithms use fixed-precision rational number for coordinate storage with *Interval + Exact* arithmetic for exact computation.

In the following sections, we present the experiments on different point localization algorithms. We evaluate the point insertion operation using different point localization algorithms and compare the performance of point insertions, constraint insertions and constrained polygon insertions and removals between Kallmann's DCDT implementation and ours.

5.1 Experiment 1: An Example of InCircle Computation

The correctness of geometric functions is important because the geometric algorithms depend on the correct results of the geometric functions. In this section, we evaluate correct-

ness of *InCircle* computation between Kallmann's floating point implementation and our exact implementation. We illustrate through our example that the floating point calculation produce incorrect results which may not be acceptable for some applications. A list of good examples related to numerical rounding errors in floating point arithmetic can be found in [26].

5.1.1 Experiment Setup

For this part of the experiment, the inputs of the *InCircle* function are rational numbers with numerator and denominator in the integer range of $[1 \ldots 100]$. The rational input is converted to its corresponding floating point value for Kallmann's *InCircle* computation where an ϵ-tolerance of 10^{-14} is used for floating point comparison. We evaluate the correctness of InCircle computation using four nearly cocircular points. We generate four cocircular points (p_x, p_y), $(-p_x, p_y)$,$(-p_x, -p_y)$ and $(p_x, -p_y)$ where the numerator and denominator of p_x and p_y are within the integer range of $[1 \ldots 100]$. We perturb the point (p_x, p_y) with a small increment to $(p_x \pm \frac{1}{i}, p_y \pm \frac{1}{i})$. The test is repeated with i ranging from 10 to 10^{17} in increasing magnitude of 10. Each test run is iterated 1000000 times with different nearly cocircular inputs. We measure the result of the InCircle computation using *Double* and *Interval* + *Exact*, and compare them with the result of the InCircle computation using *Exact*.

The result of the *InCircle* computation should correctly classify the perturbed point as inside, outside or cocircular with respect to the circumcircle of the other three fixed points. The small perturbation causes significant rounding error for fixed-precision floating point arithmetics which leads to incorrect results.

5.1.2 Experimental Results

Table 5.1 shows the experiment results of classifying perturbed cocircular points using the InCircle function. It shows that the computation using *Double* mis-classifies perturbed cocircular points when the perturbation is smaller than 10^{-11} while computation using *Interval* + *Exact* correctly classifies all inputs.

The type of misclassification include [26]:

- Sign inversion: the InCircle function mis-classifies inside as outside or vice versa.
- Rounding to zero: the InCircle function mis-classifies inside or outside as cocircular.
- Shifting zero: the InCircle function mis-classifies cocircular as inside or outside.

	Double vs Exact			Interval+Exact vs Exact		
Perturbation	Correct	Incorrect	Failure Rate (%)	Correct	Incorrect	Failure Rate (%)
10^{-1}	1000000	0	0	1000000	0	0
...	1000000	0	0	1000000	0	0
10^{-10}	1000000	0	0	1000000	0	0
10^{-11}	992346	7654	0.8	1000000	0	0
10^{-12}	970117	29883	3.0	1000000	0	0
10^{-13}	919805	80195	8.0	1000000	0	0
10^{-14}	809295	190705	19.0	1000000	0	0
10^{-15}	520306	479694	48.3	1000000	0	0
10^{-16}	60813	939187	94.0	1000000	0	0
10^{-17}	62845	937155	93.7	1000000	0	0

Table 5.1: InCircle classification using *Double* and $Interval + Exact$

The failure rate in *InCircle* computation using *Double* increases significantly as the perturbation value gets smaller. This experiment shows that the evaluation of geometric functions using fixed-precision arithmetic leads to unpredictable results. The correctness of such result can only be verified with an exact computation of the same geometric which guarantees much higher precision. Computation such as *Intersection* required a higher precision for correct results. Computation of *Intersection* using *Double* may lead to worse results.

5.2 Experiment 2: Point Localization Algorithms

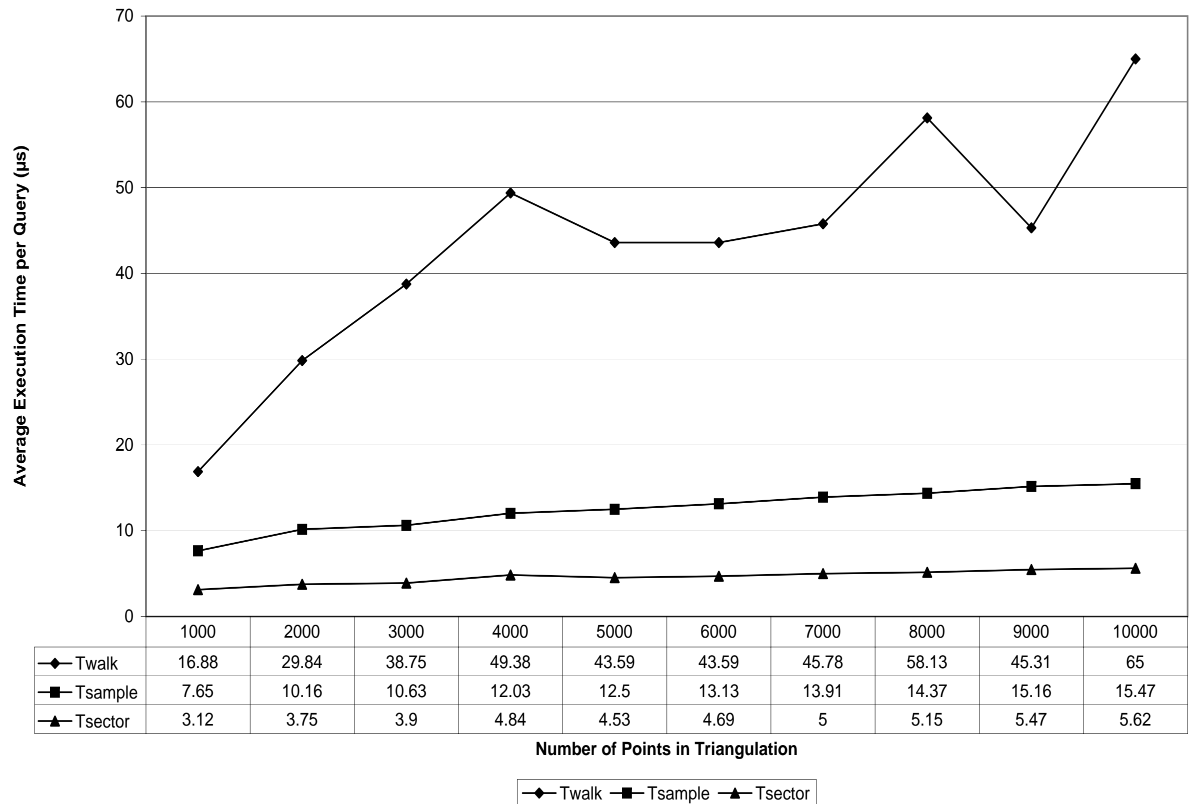
In this section, we evaluate the performance of three point localization algorithm as discussed in Section 3.2.1:

- The Cubic root sampling Jump-and-Walk algorithm as described in Algorithm 4. The algorithm randomly samples $n^{1/3}$ points in the triangulation of n points and starts the walking process with a sampled point closest to the query point.
- The Sector-based Jump-and-Walk algorithm as proposed in Algorithm 5. The algorithm invokes the Remembering Stochastic Walk algorithm after jumping to a starting location using the associated sector triangle. It avoids sector updates by storing the result of the Remembering Stochastic Walk in the corresponding sector. We use the grid size of 16×16 in this experiment.
- Remembering Stochastic Walk algorithm shown in Algorithm 3. This is the underlying walking algorithm for the Jump-and-Walk algorithms above. The algorithm starts the walk from last updated triangle in the triangulation in our experiment.

5.2.1 Experiment Setup

We evaluate the performance of the point localization algorithm by measuring 10000 point localization queries in triangulation of 1000 to 10000 random points. We insert the random points in the triangulation and then measure the execution time of 10000 random point localization queries using Boost::timer. The experiment is repeated with Cubic-root sampling, Sector-base, and Remembering Stochastic Walk point localization algorithm.

5.2.2 Experimental Results



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Twalk	16.88	29.84	38.75	49.38	43.59	43.59	45.78	58.13	45.31	65
Tsample	7.65	10.16	10.63	12.03	12.5	13.13	13.91	14.37	15.16	15.47
Tsector	3.12	3.75	3.9	4.84	4.53	4.69	5	5.15	5.47	5.62

Figure 5.1: Average execution time of Cubic-root sampling, Sector-base and Remembering Stochastic Walk point localization

Figure 5.1 shows that our Sector-based approach is the most efficient one among the point location algorithms. This is due to better starting locations for the walking process and efficient updates of sectors which avoids linear-time sector update procedures. The Sector-based point localization algorithm performs better than the Cubit-root sampling algorithm, which is considered to have $O(N^{1/3})$ average runtime for point localization queries on randomized inputs. Both Jump-and-Walk algorithm improves upon the Remembering Stochastic Walk algorithm, which suffers from bad starting locations due to its random selection of starting locations. It is worth noting that the walking algorithm determines the next triangle to step into by performing *Orientation* tests on the edges of the current triangle. Therefore, traversals on large number of triangles can be costly because of the exact computation in *Orientation* function.

5.3 Experiment 3: Point Insertion Using Different Point Location Algorithm

The running time of incremental point insertion operation is dominated by the time spent in point localization query [23]. Therefore, the performance in the point localization function is reflected in the performance of point insertion operation.

5.3.1 Experiment Setup

In this section, we evaluate the performance of our point insertion operation in our implementation using different point localization algorithms as described in the previous section.

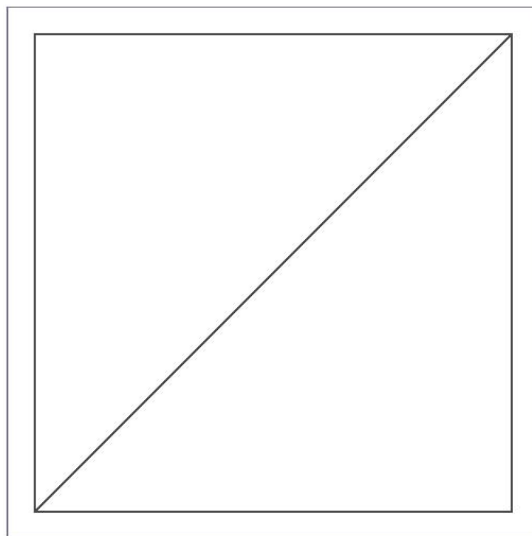


Figure 5.2: Initial bounding box setup

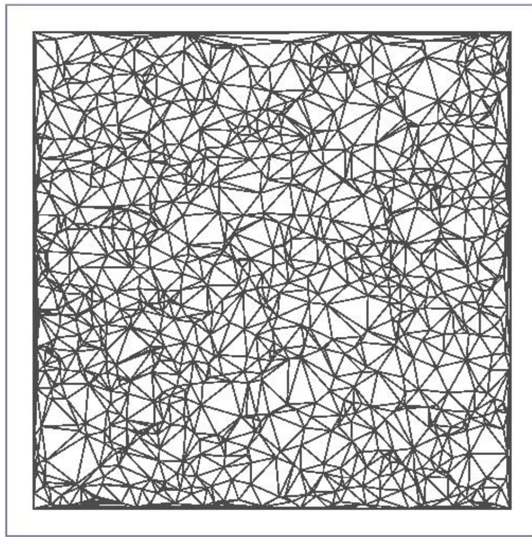


Figure 5.3: Point insertion of 1000 points

We used data inputs that are uniformly distributed over the entire bounding domain. The data points use integer coordinates that are generated inside a 16000×16000 bounding box with coordinate ranging from -8000 to 8000 (see Figure 5.2). The integer coordinates of each insertion are randomly generated using the standard C++ rand() function and scaled to the coordinates range of [-8000,8000] in our experiment. The rand() functions in both implementation are initialized with equal seeds to ensure identical insertion sequences.

The execution time was measured by the Boost::timer for point insertion operations of different input sizes ranging from 1000 to 10000 in steps of 1000 (see example in Figure 5.3). The increasing number of insertions simulates increasing mesh density. Evaluating update operations on dense triangulations allows us to better evaluate the efficiency of update operations under different setups. It also allows us to test the operation with degenerate cases.

5.3.2 Experimental Results

We define $Tstochastic$ as the average execution time of a point insertion using Remembering Stochastic Walk point localization algorithm and define $Tsector$, and $Tsample$ as the average execution time of a point insertion using Sector-based Jump-and-Walk algorithm using 16×16 grid size and Cubit-root sampling Jump-and-Walk algorithm. Figure 5.4 illustrates that $Tsector$ has a much lower costs than $Tsample$ and $Tstochastic$. This is due to the fact that Sector-based Jump-and-Walk algorithm has the best starting location for the walking process, which reduces the number of triangles visited during the walk.

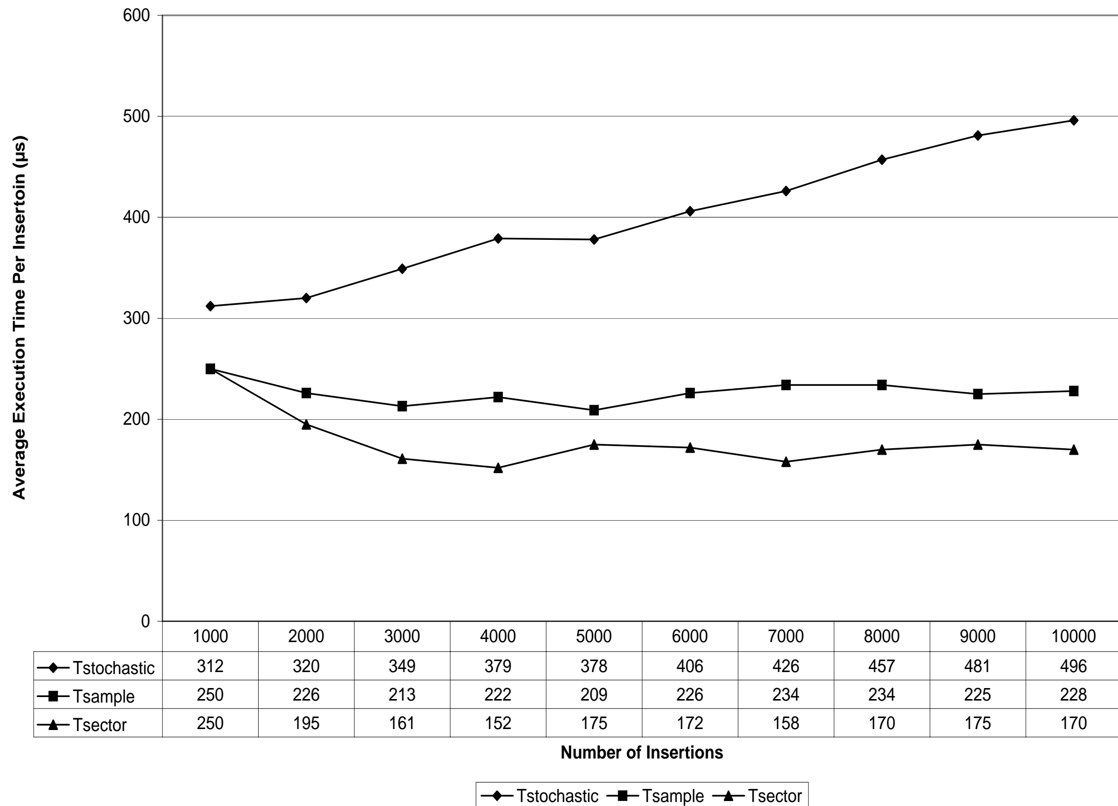


Figure 5.4: Average execution time of point insertion using Cubic-root sampling, Sector-base and Remembering Stochastic Walk point localization

5.4 Experiment 4: Point Insertion using Exact and Inexact Computation

5.4.1 Experiment Setup

The experiments in this section evaluate the computation efficiency of point insertions using $Double$ and $Interval + Exact$ arithmetic. We compare our point insertion implementation with Kallmann's DCDT implementation. The experiment setup is the same as that explained in Section 5.3.1 for both implementations. The point localization algorithm used in DCDT is the Oriented Walk algorithm which is similar to a Stochastic Walk algorithm without randomly testing the triangle edges. The algorithm detects infinite loops and falls

back to linear search for all the triangles to ensure linear time worst case performance. The point localization algorithm in our implementation is the Sector-based point localization algorithm with grid size 16×16 and using the Remembering Stochastic Walk algorithm to perform the walk towards the query point. The number of point insertions in our experiment ranges from 1000 to 10000 in steps of 1000. The execution time is measured using Boost::timer.

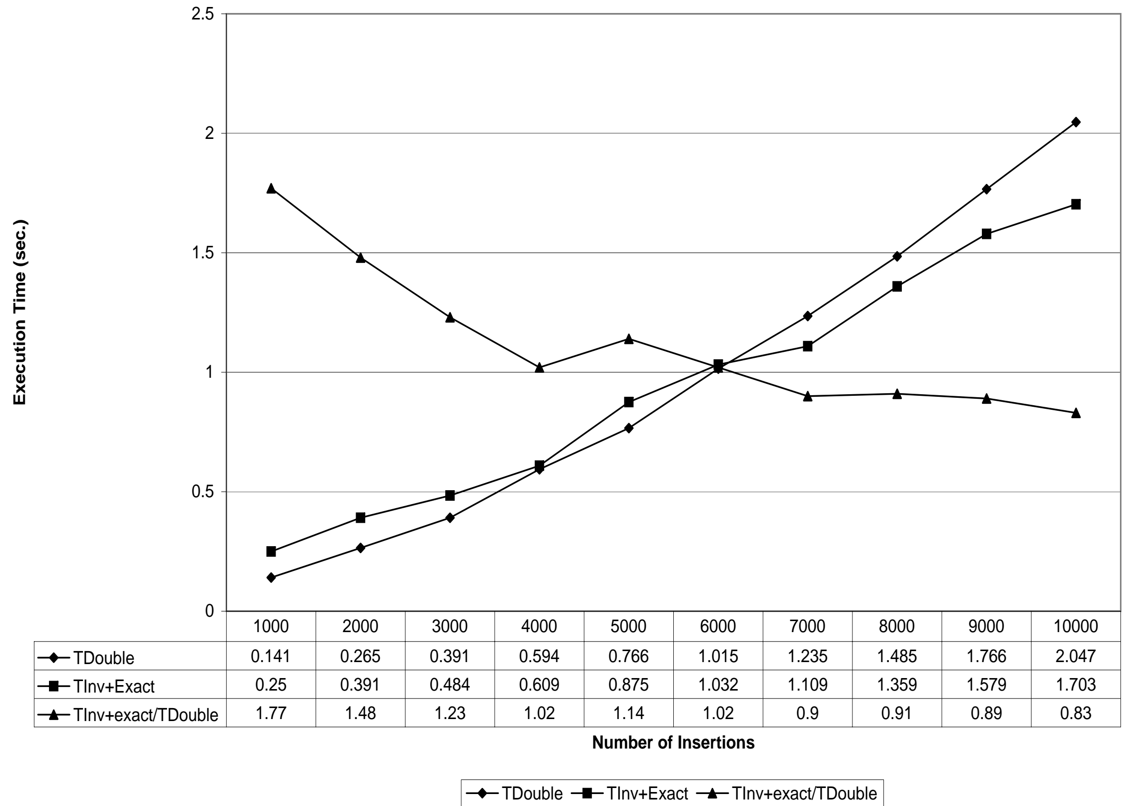


Figure 5.5: Execution time of point insertions

5.4.2 Experimental Results

Figure 5.5 compares the execution time of point insertion between Kallmann's floating point implementation (T_{Double}) and our interval filter and exact implementation ($T_{Interval+Exact}$). The implementation using $Interval + Exact$ performs better than the implementation using

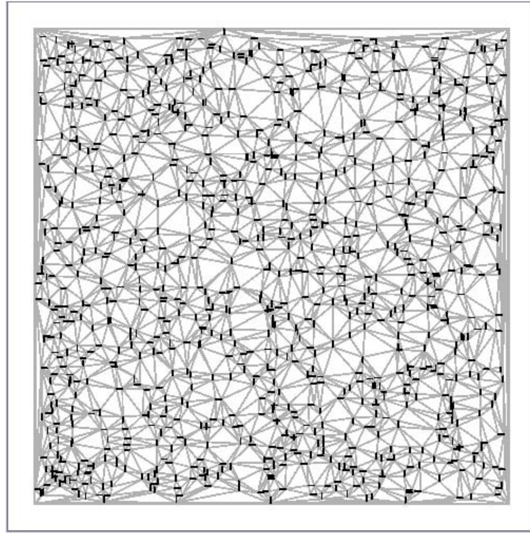


Figure 5.6: 1000 short axis aligned constraint insertions

Double as the density of the triangulation increases. $T_{Interval+Exact}$ performs better because point insertion operation uses the more efficient sector-based point localization algorithm as explained in the previous experiment. The cost saved in efficient point localization algorithm offsets the computational costs for exact computation. Therefore, the overall performance of point insertion increases with integer data inputs using $Interval + Exact$.

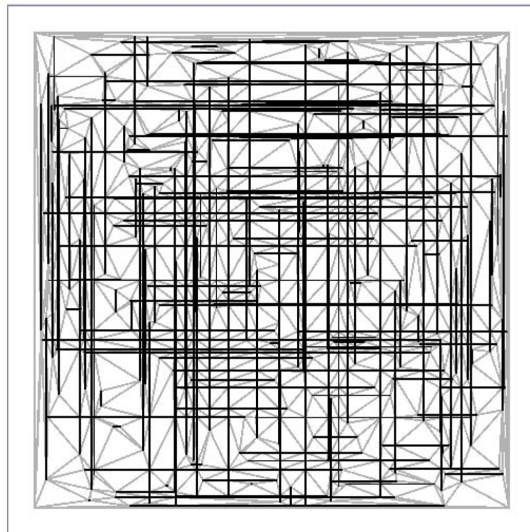


Figure 5.7: 200 long axis aligned constraint insertions

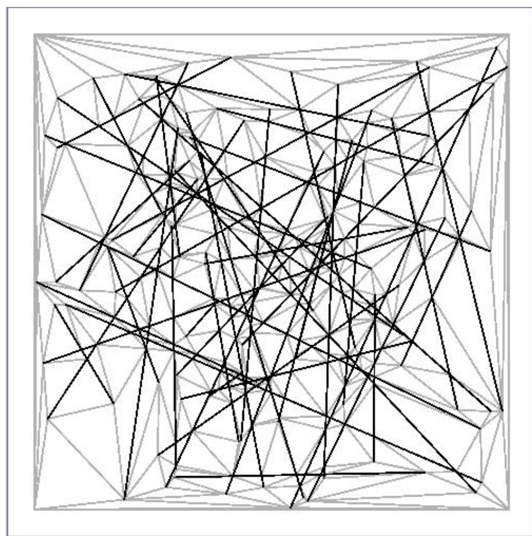


Figure 5.8: 50 random constraint insertions

5.5 Experiment 5: Constraint Insertions

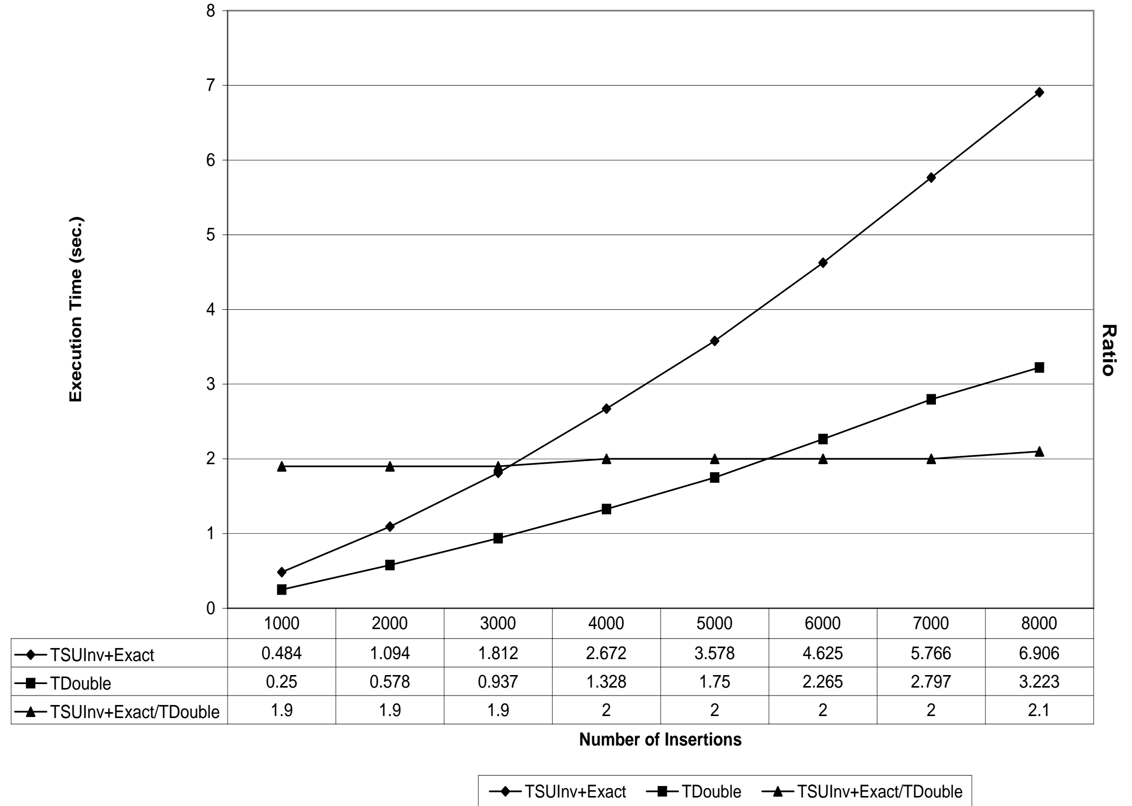
Constraint insertion is the fundamental operation for constrained polygon insertion where the edges of a constrained polygon are inserted as a sequence of constraints. Therefore, the performance of the constraint insertion operation is critical to the overall performance of constrained polygon insertion. In this section, we measure the performance of constraint insertion in different setups.

5.5.1 Experiment Setup

In this section, we measure constraint insertions in three types of setups:

- Setup 1: we measure axis aligned constraint insertions with short, fixed length constrained line segments. The constraints inserted are horizontal or vertical aligned with the bounding box (see Figure 5.6). The short constrained line segments has less intersections with each other, hence the setup allows us to evaluate the computation efficiency of constraint insertions with few intersection computation. In the experiment, the coordinates of one of the constraint endpoints are generated randomly, and the coordinates of the other endpoints are calculated with a random horizontal or vertical shift of length 200. We measure the execution time of constraint insertions with input sizes ranging from 1000 to 8000 in steps of 1000.

- Setup 2: the second experiment uses axis aligned constraint insertions with long, random length constrained line segments. Hence, the constraints intersect only on integer



	1000	2000	3000	4000	5000	6000	7000	8000
TSUInv+Exact	0.484	1.094	1.812	2.672	3.578	4.625	5.766	6.906
TDouble	0.25	0.578	0.937	1.328	1.75	2.265	2.797	3.223
TSUInv+Exact/TDouble	1.9	1.9	1.9	2	2	2	2	2.1

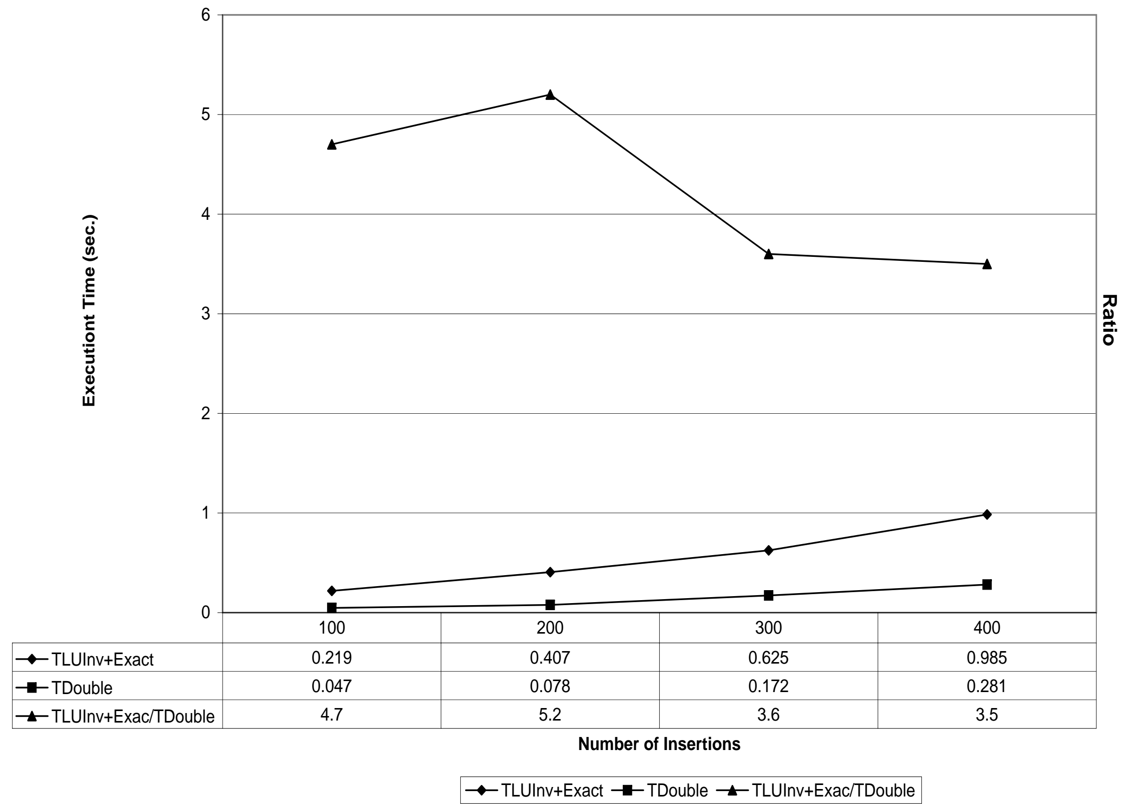
Figure 5.9: Execution time of short uniform constraint insertions

coordinates (see Figure 5.7). For this experiment, coordinate values are randomly generated for the endpoints and an alignment for the x or y coordinates of the endpoints is enforced. This experiment evaluates the performance of constraint insertions with computations in geometric functions being reduced to integer arithmetic. The experiments indicate a reduction to integer arithmetic for intersection calculation increases computation efficiency in our implementation. The execution time of constraint insertions is measured with input sizes ranging from 100 to 400 in steps of 100. We limit the input sizes to below 400 because Kallmann's DCDT implementation crashes with inputs more than 400 due to numerical errors in point localization.

- Setup 3: the setup of the third experiment uses random constraint insertions with

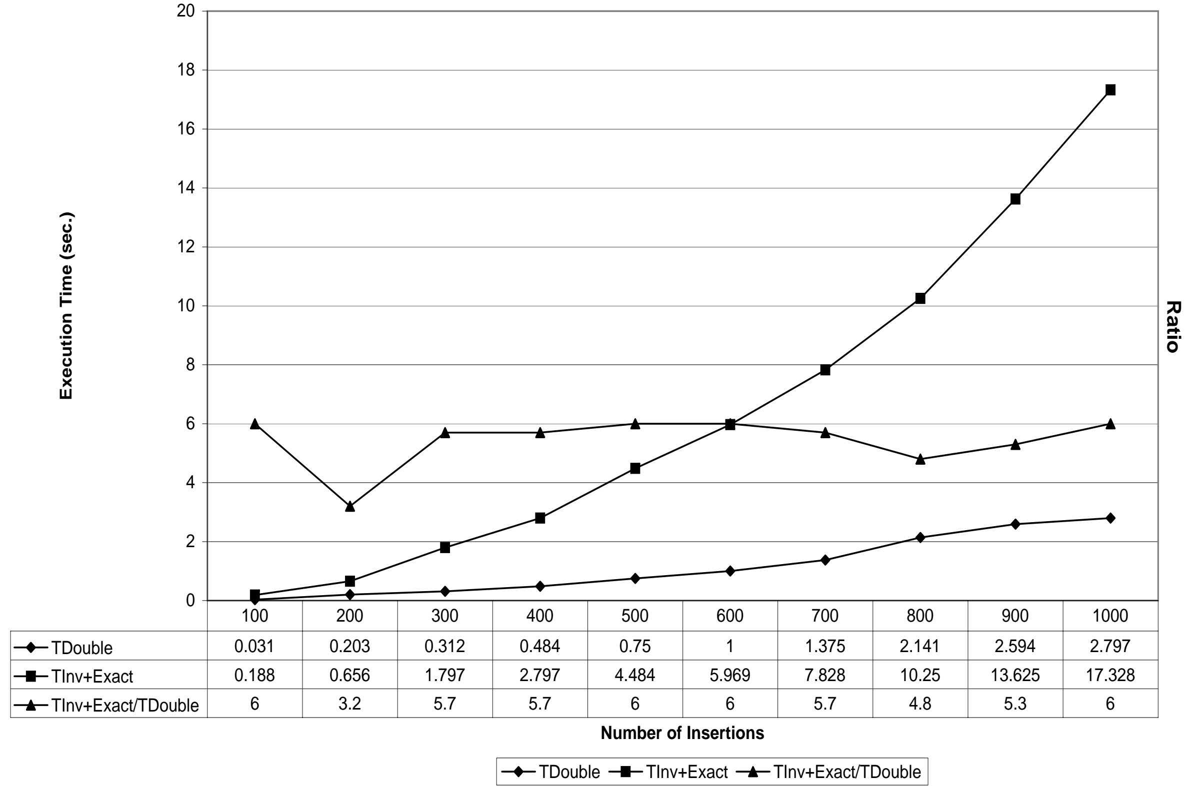
different orientation and length. It simulates a random environment with potential degenerate cases (see Figure 5.8). Both endpoints of the constraint are randomly generated in this setup. We record the execution time of constraint insertions with input sizes ranging from 100 to 1000 in steps of 100

The randomly generated coordinates for the setups above are evenly distributed in the bounding box with its coordinates ranging from -8000 to 8000.



	100	200	300	400
TLUInv+Exact	0.219	0.407	0.625	0.985
TDouble	0.047	0.078	0.172	0.281
TLUInv+Exac/TDouble	4.7	5.2	3.6	3.5

Figure 5.10: Execution time of long uniform constraint insertions



	100	200	300	400	500	600	700	800	900	1000
TDouble	0.031	0.203	0.312	0.484	0.75	1	1.375	2.141	2.594	2.797
TInv+Exact	0.188	0.656	1.797	2.797	4.484	5.969	7.828	10.25	13.625	17.328
TInv+Exact/TDouble	6	3.2	5.7	5.7	6	6	5.7	4.8	5.3	6

Figure 5.11: Execution time of random constraint insertions

5.5.2 Experimental Results

In this section, we show experimental results for the various improvements that are discussed in Section 4.3.2 and Section 4.3.3. The improvements reduce the slow down factor of random constraint insertions between Kallman's DCDT implementation and our implementation to 6.

Figure 5.9 displays the execution time for constraint insertions descibed in Setup 1. The ratio between our exact implementation ($T_{SUInv+Exact}$) and the Kallmann's floating point implementation ($T_{SUDouble}$) stays at 2.

In Figure 5.10, the execution time for constraint insertions with Setup 2 is shown. The execution time for our exact implementation is slower than Kallmann's floating point im-

Number of Insertions	# Exact	# Interval
100	438	17571
200	1504	40284
300	3258	72112
400	5641	119971
500	7557	164523
600	11408	235802
700	15848	310232
800	19309	379709
900	25208	492158
1000	30135	584335

Table 5.2: Arbitrary-precision computation vs interval computation in random constraint insertions

plementation by a factor of 4. However, Kallmann's DCDT software fail to complete the test with some overlapping constraints in this setup because its point localization procedure fails to locate the proper location in degenerate cases due to numerical rounding errors.

Figure 5.11 compares the execution time of random constraint insertion between Kallmann's floating point implementation (T_{Double}) and our interval filter and exact rational implementation ($T_{Inv+Exact}$). The slow down factor of $T_{Inv+Exact}/T_{Double}$ is 6. As discussed in Section 4.3.2, geometric functions using interval arithmetic may have inconclusive results and need to fall back to arbitrary-precision arithmetic. The number of invocations using interval arithmetic and the number of invocations using arbitrary-precision arithmetic in geometric functions are displayed in Table 5.2. The number of expensive exact computation in predicate evaluation is greatly reduced with interval filter while correctness of the outputs are maintained.

5.6 Experiment 6: Dynamic Updates of Constrained Polygon

Constrained polygons are used to model obstacles in the game maps. Dynamic insertions and removals are needed to model the construction and destruction of obstacles in game maps. Therefore, the performance of dynamic updates of constrained polygons is important for application in games. Insertion of constrained polygon consists of a sequence of constraint insertions for the edges of the polygon. The efficiency of constrained polygon insertion depends on the basic insertion operations such as point localization, point and constraint insertion. Similarly for constrained polygon removals. In this section, we empirically evaluate the performance of dynamic updates of constrained polygons in two simple setups and discuss the effects of degenerate case to the overall performance .

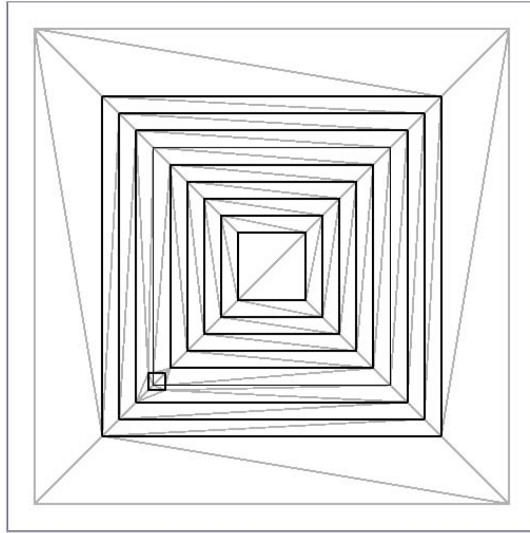


Figure 5.12: Axis aligned moving constrained polygon

5.6.1 Experiment setup for constrained polygon insertions

For tests on constrained polygon insertions and removals, we setup a initial polygonal mesh of 200000 by 200000 bounding domain with coordinates ranging from -100000 to 100000. The box contains squares of shrinking sizes in steps of 10000 for each coordinates. The squares are evenly distributed with horizontal or vertical constrained edges.

A small constrained square is inserted at the lower left corner of the bounding box. The two different setups of constrained square in the experiments are:

- Setup 1: The constrained square is axis aligned with the grid coordinates. The setup introduces intersections of integer coordinates and overlapping constrained edges (see Figure 5.12).
- Setup 2: The constrained square has a rotated angle of 45 degrees (see Figure 5.13). This setup creates intersections of rational coordinates and non-overlapping constrained edges.

The small constrained square is moved along the diagonal of the bounding box with a sequence of dynamic updates of constrained polygon removal, coordinate translations and constrained polygon insertion. The sequence is repeated from 1000 to 8000 iterations in steps of 1000 along the diagonal of bounding box. For both setups, the execution times are recorded using Boost::timer. We evaluate the performance of such dynamic updates between Kallmann's floating point implementation ($T_{PDouble}$,$T_{RDouble}$) and our interval filter and exact rational implementation ($T_{PInterval+Exact}$,$T_{RInterval+Exact}$).

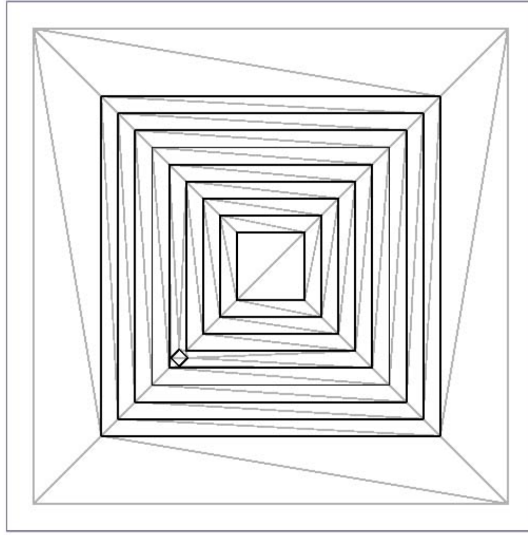


Figure 5.13: Rotated moving constrained polygon

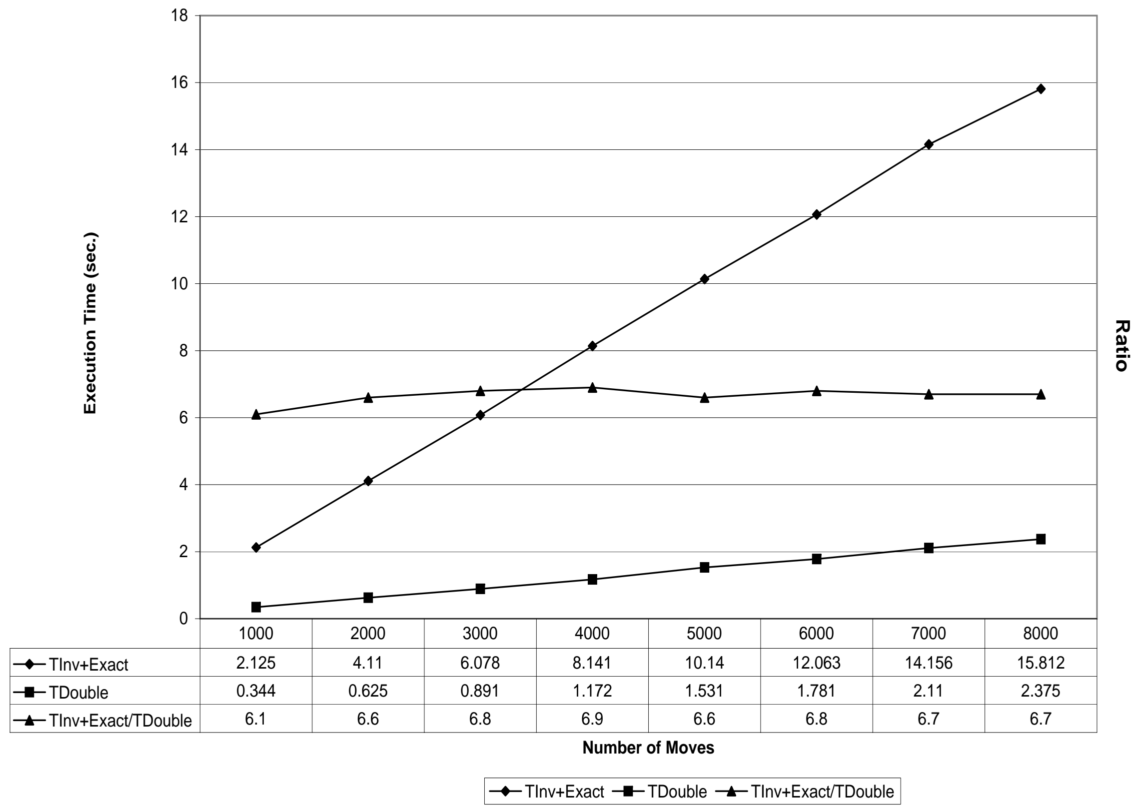
5.6.2 Experimental Results

The $T_{Inv+Exact}/T_{Double}$ ratio is about 7 in Figure 5.14 and 9 in Figure 5.15.

The performance of dynamic updates in Setup 1 is slower because computation using interval arithmetic in the *Orientation* function can not determine the sign of the determinant for degenerate cases such as overlapping. The computation of the geometric function is handled by the slower arbitrary-precision arithmetic.

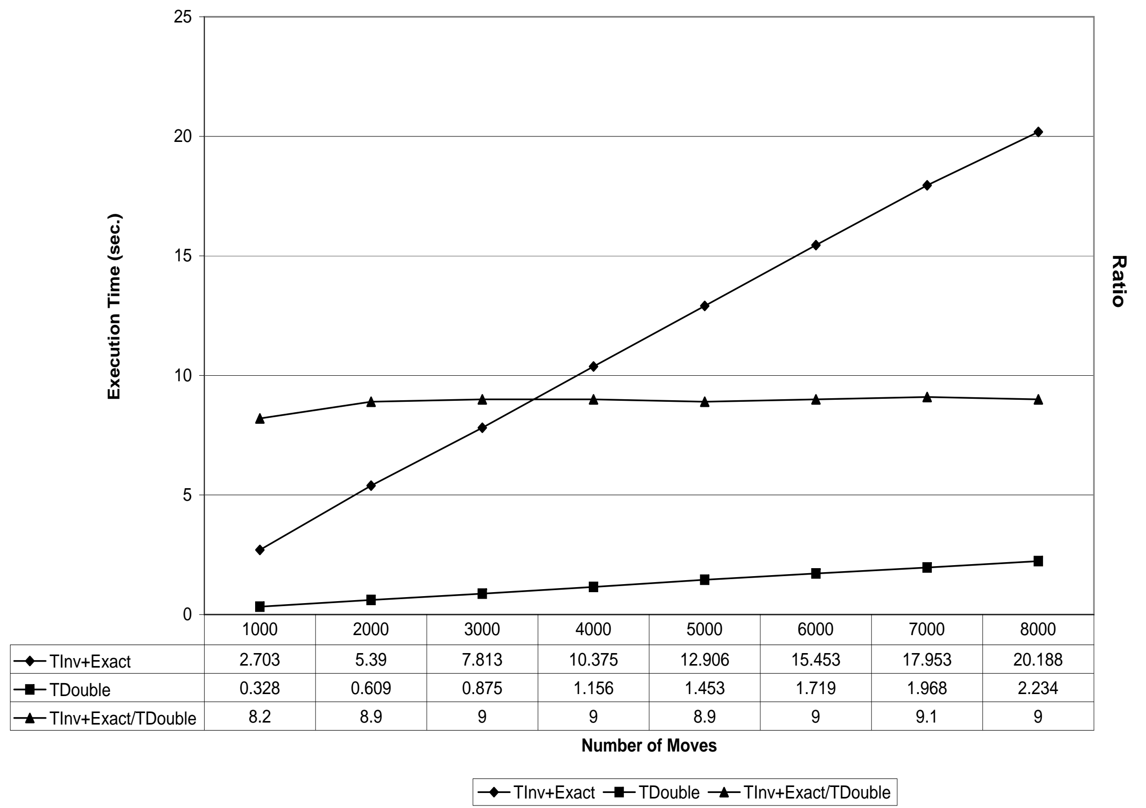
5.7 Conclusion

In this chapter, we first compare the performance of different point localization algorithm and show that our Sector-based point localization algorithm with efficient sector updates has the best overall performance with random inputs. We then evaluate the execution time of point insertion operations using different point localization algorithms. The experimental results show that the sector-based point localization algorithm performs better than point insertion operations using other point localization algorithms. We empirically compares the performance of update operations between our $Interval + Exact$ implementation and Kallmann's *Double* implementation. The experimental results indicate that it is possible to use exact computation for geometric functions with an acceptable performance lost.



	1000	2000	3000	4000	5000	6000	7000	8000
TInv+Exact	2.125	4.11	6.078	8.141	10.14	12.063	14.156	15.812
TDouble	0.344	0.625	0.891	1.172	1.531	1.781	2.11	2.375
TInv+Exact/TDouble	6.1	6.6	6.8	6.9	6.6	6.8	6.7	6.7

Figure 5.14: Execution time of moving rotated square



	1000	2000	3000	4000	5000	6000	7000	8000
TInv+Exact	2.703	5.39	7.813	10.375	12.906	15.453	17.953	20.188
TDouble	0.328	0.609	0.875	1.156	1.453	1.719	1.968	2.234
TInv+Exact/TDouble	8.2	8.9	9	9	8.9	9	9.1	9

Figure 5.15: Execution time of moving axis aligned square

Chapter 6

Conclusions and Future Work

As a part of the thesis research, we developed robust and efficient dynamic constrained Delaunay triangulation software based on exact computation. Robustness and speed efficiency are crucial for the software to be used as part of triangulation pathfinding engine in games such as ORTS. The computational efficiency of update operations has been increased by the following algorithmic improvements:

- The speed of point localization was increased using a sector-based approach on top of the Remembering Stochastic Walk algorithm. The Remembering Stochastic Walk algorithm saves 1.5 *Orientation* tests per visited triangle by remembering the cross edge between triangles. This in turn speeds up point, constrained line segment, and polygon insertion.

- Computing exact intersection coordinates using arbitrary-precision arithmetic is expensive. By limiting the input coordinate range to $[0 \ldots 10^6]$, which is sufficiently large for many application, we are able to compute and store the exact intersection coordinates using fast fixed-precision rational arithmetic. This choice strikes a balance between computation speed and memory requirements and still gives the user the ability to represent large maps.

- Correct geometric algorithms rely on the exact computation of geometric functions such as *Orientation* and *InCircle* test which can be formulated as determining the sign of a determinant. In our software, we address the rounding error problem by evaluating the exact sign of the determinant using interval arithmetic and falling back to arbitrary-precision arithmetic when interval arithmetic is unable to determine the sign of the determinant exactly in geometric function computation.

Experiments for dynamic constrained polygon updates indicate an overall slow down factor of 6 to 9 for our current implementation, compared with error prone software based on fixed-precision floating point. We think that this slow down is acceptable because dynamic updates are still very fast, and the results can be trusted. The software can applied in various areas such as scientific visualization and finite-element mesh generation where correct triangulation is required.

Future Work

There are several future work potentials for dynamic constrained Delaunay triangulation using exact computation:

- Concurrent computation. The computational cost of arbitrary-precision arithmetic is quite high compared to standard floating point arithmetic. Our current implementation is single-threaded. Some operations can be partitioned into a number of similar processes. For example, the re-triangulation of the upper and lower halves of the open region during constraint insertion can be executed concurrently because the triangulation of the two regions does not influence each other. Parallelizing the computation of such operation does not require much synchronization effort and could yield considerable speedups.

- The Delaunay Refinement Algorithm [32] systematically inserts steiner points in the existing constrained Delaunay triangulation forming a conforming Delaunay triangulation with guaranteed bounds on angles (excluding the input angles), edge lengths and the number of triangles [32],[10]. The conforming Delaunay triangulation can be constructed from a constrained Delaunay triangulation with hierarchal data structure storing the changes between them. In conforming Delaunay triangulation, the triangle density is higher around constrained corners of the obstacles. The triangle strip found on the conforming Delaunay triangulation using TA* can be refined back to constrained Delaunay triangulation using the hierarchal data structure and used as guidance for the actual TA* search.

- The integration of our triangulation software and triangulation-based pathfinding into games engine such as ORTS. The software package can be easily integrated into existing game engines as part of the pathfinding engine.

Code Listing

The interface of our implementation is similar to that of Kallmann's DCDT software package. We maintain the same interface to ensure an effortless transaction of pathfinding implementations such as TA* and TRA*.

The random point insertion test:

```
void needlePointTest( int n)
 {
      Vertex* v;
      Vertex::coord_t px,py;
//Set Sector values to -1
      theTriangulator.InitializeSectors();

      while ( n>0 )
      {

           Face* resultFace;
           HalfEdge* resultEdge;
           Vertex* resultVertex;



           DtTriangulator::LocateResult resType;
//Generate integer coordinates
           px = random_coord();
           py = random_coord();

      //Sector-based point location
           resType = theTriangulator.LocatePoint
(NULL, px, py, &resultFace, &resultEdge, &resultVertex);



           if ( resType==DtTriangulator::NotFound )
           {
                printf ( "Face Not Found\n" );
           }
           else if ( resType==DtTriangulator::VertexFound )
           {
```

```
                n--;
            }
            else if ( resType==DtTriangulator::EdgeFound )
            {
                DtMesh::coord_t rpx(px);
                DtMesh::coord_t rpy(py);
                theTriangulator.insertPointOnEdge ( resultEdge, rpx, rpy );
                n--;
            }
            else // a face was found
            {
                DtMesh::coord_t rpx(px);
                DtMesh::coord_t rpy(py);
                theTriangulator.insertPointInFace ( resultFace, rpx, rpy );
                n--;
            }
        }

    }
```

The random constraint insertion test:

```
void constraintNeedlePointTest(int n)
{

    int ax, ay, bx, by;
    while ( n>0 )
    {
        //Generate random integer coordinates
        ax = randomCoord();
        ay = randomCoord();

        bx = randomCoord();
```

```
            by = randomCoord();

            //skip point insertion
            if(ax == bx && ay == by)
            {
                 continue;
            }
  //Insert constraint with endpoint (ax,ay), (bx,by), and ID =n
            CDTMAIN::insertConstraint(ax,ay,bx,by,n);
            n--;
      }
}
```

The constrained polygon test:

```
void constraintPolygonTest()
{
for(int i=1; i<=8000; i++)
       {
            SimplePolygon<Vertex,HalfEdge,Face> pol;

             DCDTMAIN::theTriangulator.getPolygon(9,pol);

//find the centroid of the polygon
             Point c = pol.centroid();

//Compute the translation along the diagonal
             Point p(c.getPosX()+10,c.getPosY()+10);
             std::vector<Point> newPol = pol.translate ( p,c );

  //Remove existing constrained polygon with ID 9
              DCDTMAIN::theTriangulator.removePolygon ( 9 );
//Insert a new constrained polygon with ID 9
              DCDTMAIN::theTriangulator.insertPolygon(newPol,9);

       }
```

```
}
```

Bibliography

[1] M. V. Anglada. An improved incremental algorithm for constructing restricted Delaunay triangulations. In *Computer and Graphics*, volume 21, chapter 2, pages 215–223, 1997.

[2] C. B. Boyer. A History of Mathematics, second edition. *Wiley*, New York, 1968.

[3] B. G. Baumgart. Winged edge polyhedron representation. *Stanford University*, 1972.

[4] J. A. Bondy and U. S. R. Murty. Graph Theory with Applications. *North Holland*, 1997.

[5] H. Bronimann, G. Melquiond, and S. Pion. The Boost Interval Arithmetic Library. In *Proceedings of 5th Conference on Real Numbers and Computers*, pages 65-80, 2003.

[6] M. Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the International Computers and Games Conference*, pages 156–161, 2002.

[7] J. Chen Computational Geometry: methods and algorithm . *Cambridge university press*, 1997.

[8] L. P. Chew. There is a planar graph almost as good as the complete graph. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 169–177, 1986.

[9] L. P. Chew. Constrained Delaunay Triangulations. In *Proceedings of the Annual Symposium on Computational Geometry ACM*, pages 215–222, 1987.

[10] L. P. Chew. Guaranteed-Quality Mesh Generation for Curved Surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 274–280, 1993.1995.

[11] B. Delaunay. Sur la sphère vide. In *Izvestia Akademii Nauk SSSR*, volume 7, pages 793–800. Otdelenie Matematicheskikh i Estestvennykh Nauk, 1934.

[12] D. Demyen and M. Buro. Efficient Triangulation-Based Pathfinding, In *In Proceedings of the Association for the Advancement of Artificial Intelligence Conference* Boston, parges 942C947,2006

[13] O. Devillers, S. Pion, and Mo. Teillaud. Walking in a triangulation, In *Proceedings of the seventeenth annual symposium on computational geometry* , pages 106–114, year 2001.

[14] L. Devroye, E. Mucke, and B. Zhu. A note on point location in Delaunay triangulations of random points, In *Algorithmica*, volume 22, pages 477–482, 1998.

[15] H. Edelsbrunner. Geometry and Topology for Mesh Generation. *Cambridge University Press*, 2001.

[16] S. Fortune. A sweepline algorithm for Voronoi diagrams, In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, volume 4, number 2, pages 313–322, 1986.

[17] T. Granlund, GMP, The GNU Multiple Precision Arithmetic Library, 4.2.1 edition. *http://gmplib.org*, 2007.

[18] P. Green and R. Sibson. Computing Dirichlet Tesselations in the plane, In *The Computer Journal* , volume 21, pages 168–173, 1978.

[19] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi, In *ACM Computing Surveys (CSUR)* , volume 4, number 2, pages 74–123, 1985.

[20] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized Incremental Construction of Delaunay and Voronoi Diagrams. In *ICALP '90: Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 414–431, 1990.

[21] C. Huang and T. Shih. Improvements on Sloan's algorithm for constructing Delaunay triangulations, In *Computers and Geosciences*, volume 24, number 2, pages 193–196, 1998.

[22] M. Kallmann. Path planning in triangulations. In *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 49–54, 2005.

[23] M. Kallmann, H. Bieri, and D. Thalmann. Fully Dynamic Constrained Delaunay Triangulations. In *Geometric Modelling for Scientific Visualization*, pages 241–257, Springer-Verlag, 2003.

[24] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. In *ACM Trans. Graph.*, volume 10, number 1, pages 71–91, 1991.

[25] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proceedings 14th Annual. ACM Symposium*, pages 146–154, 1998.

[26] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry: Theory and Applications*, volume 40, issue 1, pages 61–78, 2008. Springer

[27] D. Kirkpatrick. Optimal Search in Planar Subdivisions, In *SIAM Journal on Computing*, volume 12, issue 1, pages 28–35, 1983.

[28] M. V. Kreveld, M. Overmars, O. Schwarzkopf, and M. de Berg. Computational Geometry: Algorithms and Applications. *Springer-Verlag*, 2000.

[29] C. Lawson. Software for C1 surface interpolation, In *Mathematical Software III* , pages 161–194, 1977.

[30] C. Li, S. Pion, and C. Yap Recent Progress in Exact Geometric Computation In *Journal of Logic and Algebraic Programming*, volume 64, number 1, pages 85–111, 2005.

[31] A. Mirante and N. Weingarten. The Radial Sweep Algorithm for Constructing Triangulated Irregular Networks *Computer Graphics and Applications*, volume 2, issue 3, pages 11–21, 1982.

[32] J. Ruppert. A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation. In*Journal of Algorithms*, volume 18, number 3, pages 548–585, 1995.

[33] N. Sarnak and R. Tarjan. Planar point location using persistent search trees, In *Communications of the ACM*, volume 29, number 7, pages 669–679, 1986.

[34] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons, *Computation Geometry Theory and Application*, volume 1, pages 51–64, 1991.

[35] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Lecture Notes in Computer Science*, volume 1148, pages 203-222, 1996 Springer

[36] R. Sibson. Locally equaiangular triangulations. In *The Computer Journal*, 1978.

[37] K. Sugihara, M. Iri, H. Inagaki, and T. Imai Topology-Oriented Implementation An Approach to Robust Geometric Algorithms In *Algorithmica*, volume 27, number 1, pages 5–20, 2000.

[38] P. Yap. Grid-Based Path-Finding, In *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, pages 44–55, publisher Springer-Verlag ,2002.

[39] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Standard 754*, New York, 1985.