A 2D GPU MESH GENERATOR

QI MENG
M.Sc., Shandong University, China

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

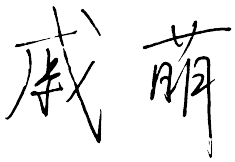
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Qi Meng
November 2014

Acknowledgments

I would like to express my gratitude to all those who helped me during the writing of this thesis.

My deepest gratitude goes first and foremost to Dr. Tan Tiow-Seng, my supervisor, who has offered me valuable suggestions in the academic studies. He has spent much time discussing the topic with me and provided me with inspiring advice. Without his illuminating instruction, insightful criticism and expert guidance, the completion of this thesis would not have been possible.

I am very grateful to Cao Thanh-Tung and Gao Mingcen for selflessly sharing their knowledge and time, for hundreds deep discussions we have had about every topic, for sitting besides me to help me out of the endless code optimizations.

I would like to thank Prof. Low Kok-Lim, Prof. Alan Cheng Ho-Lun, and Prof Huang Zhiyong for their precious comments and suggestions on my research during the weekly meeting of G^3 Lab.

Not forgetting to thank all my fellow lab-mates who offered me great company and assistance in many ways. Thank you, my friends Hua Binh-Son, Cao Thanh-Tung, Gao Mingcen, Guo Jiayan, Li Ruoru, Yan Ke, Liu Linlin, Tang Ke, Su Jun, Alvin Chia, Lai Kuan, Poonna Yospanya, Kang Juan, Sang Ngoc Le, Li Yunzhen, Du Cheng, you have enriched my life in NUS, making it more enjoyable and fun.

Last but not least, my gratitude would go to my beloved family for their loving considerations and great confidence in me, helping me out of difficulties and supporting me without any complaint.

Abstract

Meshes composed of triangles are used in various applications such as computer graphics, interpolation, finite element method, and terrain databases. There are several successful triangle mesh generators which can generate Delaunay triangulation, constrained Delaunay triangulation, conforming Delaunay triangulation and quality triangulation on the CPU. However, there is no similar generator for the graphics processing unit (GPU) architecture existing at this moment.

The GPU has been used not only for graphics processing tasks but also general computations in many disciplines including computational geometry due to its enormous parallel computing power. In computational geometry, early works on the GPU include computing the digital Voronoi diagram and Delaunay triangulation. There has been no prior approach to generate other triangle mesh such as constrained Delaunay triangulation, conforming Delaunay triangulation and quality triangulation efficiently using the parallel computing power of the GPU. The GPU is massively multithreaded, with hundreds of processors, in order to fully utilize the GPU hardware, a parallel algorithm usually needs to have regularized work and localized data access. However it is even not clear how to achieve these criteria while adapting the traditional and usually complex parallel techniques, such as divide-and-conquer to the mesh generation problem. So it is not clear how to efficiently apply traditional parallel algorithms directly on the GPU.

In this thesis, we focus on designing mesh generating algorithms in 2D space on the GPU. Two algorithms termed as GPU-QM and GPU-CDT are proposed in the thesis, which can improve the quality of the Delaunay triangulation for a point set, and compute constrained Delaunay triangulation for a set of points and constraints, respectively. Both of these two algorithms are the first GPU algorithms proposed so far. According to our experiments for both synthetic and real-world data, our GPU algorithms are numerically robust and run faster than the fastest sequential algorithm. Comparing to the fastest sequential implementation, the GPU-QM gains up to 5.5 times speedup; the GPU-CDT gains up to two orders of magnitude speedup. Furthermore, we obtain the first GPU mesh generator by integrating the GPU-QM, GPU-CDT algorithms with an existing work GPU-DT, which can compute Delaunay triangulation for a point set using the GPU. Our mesh generator can compute digital Voronoi diagrams, Delaunay

triangulation, constrained Delaunay triangulation, conforming Delaunay triangulation and high-quality triangle meshes in 2D space on the GPU. In order to handle numerical error and degenerate input, we implement exact predicates and simulation of simplicity method on the GPU based on the sequential implementations. So our generator can handle exact arithmetic and is numerically robust.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Meshes composed of triangles are used in various applications such as computer graphics, interpolation, finite element method, and terrain databases. Although there are several successful triangle mesh generators on the CPU, such as *Triangle*, CGAL and so on, which can generate Delaunay triangulation (DT), constrained Delaunay triangulation (CDT), conforming Delaunay triangulation and quality triangulation, there is no such a generator as far as we know for the graphic processing unit (GPU) architecture. Recently GPU with its enormous parallel computing power has been used widely in many disciplines for general purpose computation. Since GPU uses a massively parallel architecture with hundreds to thousands of processing elements to execute thousands to millions of threads simultaneously, common issues in parallel programming such as cooperation among threads, conflicting data access, and racing conditions become more serious problems. In order to fully utilize the GPU hardware, a parallel algorithm usually needs to have regularized work and localized data access. It is hard and even not clear how to achieve those criteria while adapting the traditional and usually complex parallel techniques, such as divide-and-conquer to the mesh generating problem. In the thesis, we commit to design 2D CDT and quality mesh generating algorithms which are suitable for parallel computation, especially for the GPU architecture. Furthermore, we obtain a 2D mesh generator on the GPU after integrating an existing DT computation algorithm. According to our experiment results for both synthetic and real-world data, our mesh generator is numerically robust and runs much faster than existing sequential algorithms. Comparing the fast sequential implementation, our quality mesh algorithm runs up to 5.5 times faster, while our CDT algorithm runs up to two orders of magnitude faster.

Usually, a 2D mesh generator should have the ability to compute the following triangle mesh:

- DT
- CDT
- Conforming triangulation
- Quality triangulation

For the DT computation, there are several GPU algorithms and implementations, such as [RTCS08, QCT12, QCT13, CNGT14]. Usually, the generation of conforming Delaunay triangulation is combined with quality mesh generation by forcing all edges in the mesh being Delaunay edges. So we do not discuss conforming Delaunay triangulation explicitly here.

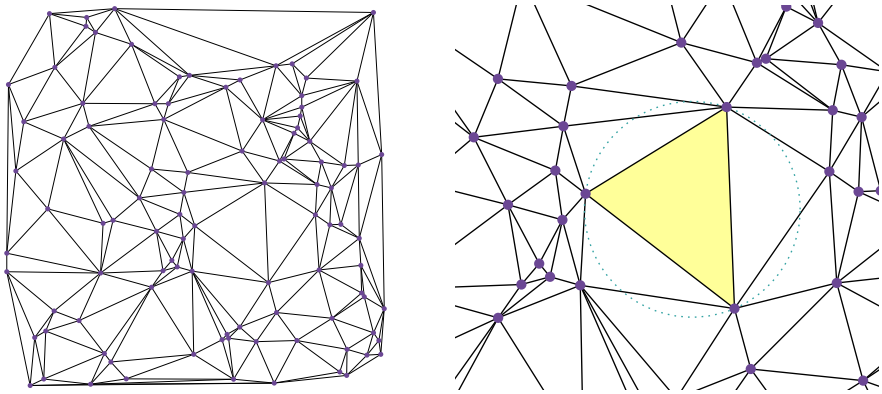
In the thesis, we managed to address the other two mesh generation algorithms listed above, i.e., CDT and quality triangulation. In our CDT generation algorithm, the algorithm can handle any *planar straight line graph* (PSLG) input. While in our quality mesh generation algorithm, the input is a segment-bounded DT of a set of points, and there is no small angle between any two adjacent segments. Setting restrictions on the input of the quality mesh generation algorithm, is because in both theory and practice, no algorithm or implementation can guarantee to terminate for all input data with good quality. Notice that even for the sequential algorithm, it is still a big challenge to handle small angles which are from the input. All existing algorithms are likely to be foiled by input data which have small features, small angles or complicated topologies. Before quality triangulation algorithm can be applied to the GPU scenario, thoroughly preparation on knowledge of both CPU and GPU are needed. Hence, the quality mesh generation algorithm for any PSLG is left to the future discussion. In the thesis we only discuss the quality mesh generation algorithm for a segment-bounded DT of a set of points, and no angle between any two input segments is less than 60°.

In the following sections, background knowledge related to mesh generator and GPU architecture are introduced. At the end of this chapter, contributions of the thesis will be highlighted.

1.1 DT and CDT

DT is one of the most important geometric structures in computational geometry and is named after Boris Delaunay for his work on this topic from 1934. A DT for a set of points S in a plane is a triangulation such that no point in S is inside the circumcircle of any triangle in the triangulation (see Figure 1.1). Such a special property of the Delaunay triangulation is also called *empty circle property*. The DT maximizes the minimum angle of all the angles of the triangles in the triangulation. In other words, among all triangulations of a given set of points, the DT has the largest minimum angle.

Due to its nice property of avoiding long, skinny triangles, the DT has many practical applications in different fields. For example, in geographical information system (GIS), one way to model the terrain is to interpolate the data points based on the DT [Gol94]. In path planning, the DT can be used to compute the Euclidean minimum spanning tree of a set of points, because the latter is always a subgraph of the former [PS85]. The DT is also often

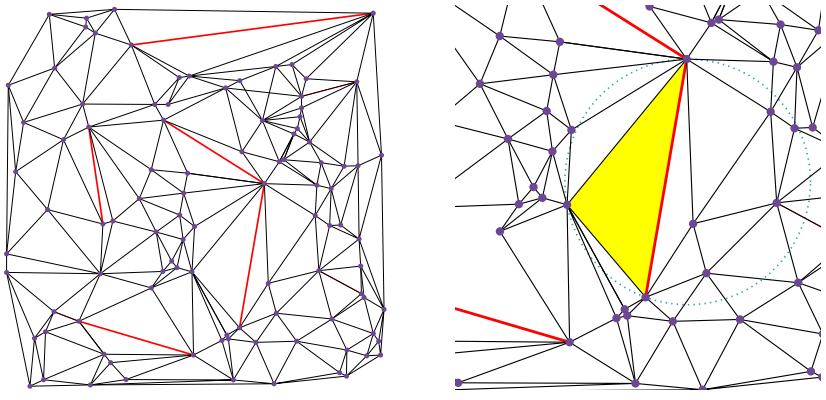


(a) (b)

Figure 1.1: (a) An example of DT of a set of points. (b) No point is inside the circumcircle of any triangle in the triangulation.

used to build quality meshes for the finite element analysis [HDSB01].

CDT is a direct extension of the DT where some edges in the output are enforced beforehand [Che89a]; these edges are referred to as *constraints*. Given a set S of n *points* (or *sites*) in the 2D plane and a set of non-crossing constraints, the CDT is a triangulation of S having all the constraints included, while being as close to the DT of S as possible (see Figure 1.2). So the CDT inherits the DT's optimality: among all possible triangulations of a set of points that include all the constraints, the CDT maximizes the minimum angle. Similar to the empty circle property of the DT, CDT must fulfill the *weaker constrained empty circle property*. To state this property, it is convenient to think of constrained edges as the wall which blocking the view. For a CDT, no visible point is inside the circumcircle of any triangle in the triangulation.



(a) (b)

Figure 1.2: (a) An example of CDT of a set of points and constraints. (b) No visible point is inside the circumcircle of any triangle in the triangulation.

Constraints occur naturally in many applications. For example, in path planning, they

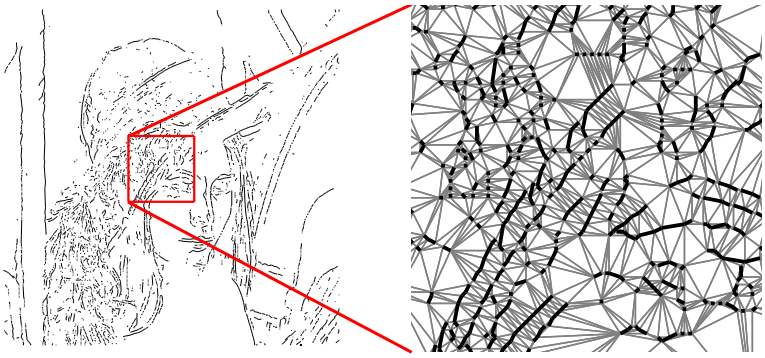


Figure 1.3: An edge map of an image and its CDT.

are obstacles; in GIS, they are boundaries between cities; in surface reconstruction, they are contours in the slices of the body's skull; and in modeling, they are characteristic curves [Boi88, Tre95, Kal10]. In short, the CDT complements the DT and is a very useful structure in many fields. Figure 1.3 shows an example of how to apply CDT in the image vectorization (details can be found in Section 4.5.3). This figure shows an edge map of a raster image (depending on the resolution of the image, the edge map might consist of hundreds of thousands of line segments or constraints), and the CDT result for it.

1.2 Delaunay Refinement

Unstructured meshes such as DT and CDT are important geometric structures in computational geometry. Due to their nice property of avoiding long, skinny triangles, they have many practical applications in different fields. However, in the real-world applications such as interpolation, the finite element method, and the finite volume method, the triangle meshes produced by the mesh generator should satisfy guaranteed bounds on angles, edge lengths, the number of triangles, and the grading of triangles from small to large size. Neither DT nor CDT can satisfy such requirements in theory and practice.

For example, most mesh generation algorithms take a PSLG as their input. A PSLG is a set of vertices and segments (constraints) shown in Figure 1.5a. Although the DT can maximize the minimum angle of all the angles of the triangles in the triangulation, there are some skinny triangles in the mesh. Furthermore, the DT of the vertices may not respect the domain that a user wishes to triangulate. On the other hand, the CDT, as an extension of the DT, can conform to the domain's boundary, but still cannot eliminate skinny triangles in the mesh.

Conformity and skinny triangle deletion, both of these problems can be solved by inserting additional points (or Steiner points, named after Jakob Steiner). The main difficulty is

where to place the additional points. Usually people use the technique called *Delaunay refinement* to generate triangle meshes. The main problem of Delaunay refinement is to find a triangulation that can fit arbitrarily complicated domains, and contain only triangles of appropriate size and shape.

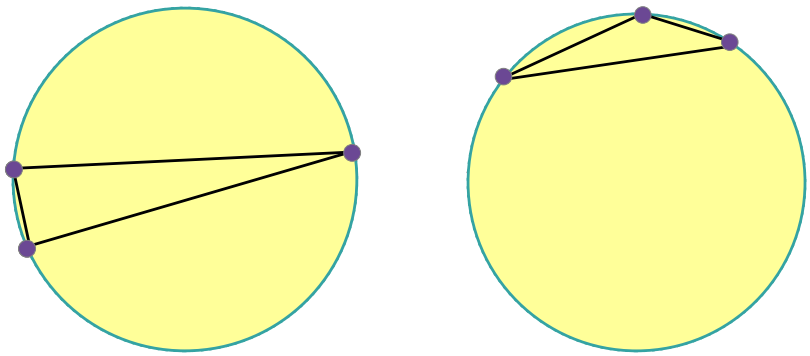


Figure 1.4: Two kinds of skinny triangles whose circumradii are much larger than their shortest edges. (a) Needle, whose longest edge is much longer than its shortest edge. (b) Cap, whose maximum angle is close to 180°.

Generally, a good Delaunay refinement algorithm should satisfy three goals listed in the following:

1. The union of the triangles is the domain of interest, and the triangulation should respect the segments (constraints).

2. All triangles should be relatively "round" shape: there are no too small or large angles. Triangles with too small or large angles are called skinny triangles (low quality or bad triangles; see Figure 1.4). Large angle can lead to large error in the gradients of the interpolated surface, and large discretization error in finite element method. Small angle can lead the coupled systems of algebraic equations such that the finite element method yields to be ill-conditioned.

3. Algorithm can offer as much control as possible over the sizes of triangles, which affects the speed and accuracy. Usually, small triangles offer more accuracy than larger ones, but the computation time is proportional to the number of triangles. Hence, different triangle sizes entails trading off speed and accuracy. For a given mesh, it is not difficult to refine it to generate another mesh which contains more triangles. But the reverse process is relatively complicated [MTT97].

Most mesh generation algorithms take a PSLG as their input. During the Delaunay refinement processing, a segment usually is divided into smaller edges. The domain of interest or the *triangulation domain* is the region that is needed to be triangulated. Such a domain should be *segment-bounded*, meaning that the segments cover the boundary of the triangulation domain from its complement, the *exterior domain*. A triangulation domain can be

either convex or concave, and it may contain holes, but the holes should be bounded by segments.

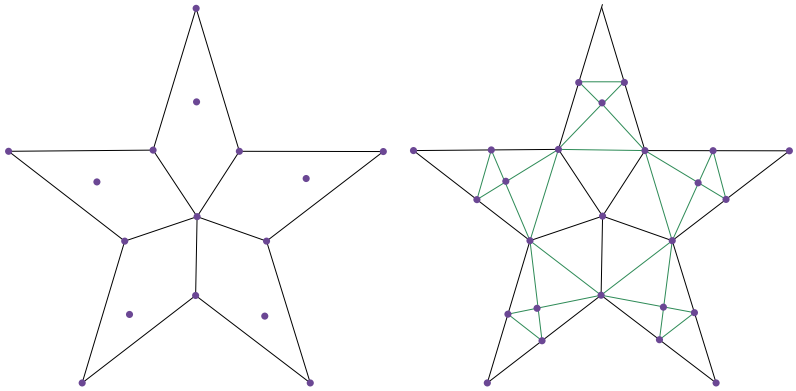


Figure 1.5: (a) A PSLG, and (b) a mesh generated by Ruppert's Delaunay refinement algorithm.

The main process of Delaunay refinement algorithms is to insert carefully placed Steiner points until the mesh meets constraints on triangle quality and size while maintaining a DT or CDT. There are several advantages of the Delaunay refinement algorithm. First, it maintains the DT, which maximizes the minimum angle among all possible triangulations of a point set, and is optimal among all possible triangulations of a point set. Second, inserting a vertex to a DT is a local operation, which is inexpensive except in unusual cases.

In practical applications, such as the finite element method, there are several measures in use for the quality of a triangle. Usually, a good quality mesh means all triangles are non-obtuse, or all with bounded aspect ratio. Aspect ratio of a triangle is the length of the longest edge divided by the length of the shortest altitude. A fairly general measure of triangle is the minimum angle α, since this gives a bound of $\pi - 2\alpha$ on maximum angle and guarantees an aspect ratio between $|\frac{1}{\sin\alpha}|$ and $|\frac{2}{\sin\alpha}|$. The most natural and elegant measure for analyzing Delaunay refinement algorithms is the *circumradius-to-shortest edge ratio* (r/l) of a triangle [MTTW95]. The *circumcenter* and *circumradius* of a triangle are the center and radius of its circumcircle, respectively. Actually, a triangle's circumradius-to-shortest edge ratio r/l is related to its smallest angle θ_{min} by the formula $r/l = 1/(2\sin\theta_{min})$. The smaller a triangle's ratio, the larger its smallest angle, i.e., given an upper bound B on the circumradius-to-shortest edge ratio of all triangles in a mesh, there is no angle smaller than $\arcsin\frac{1}{2B}$ (and vice versa). Clearly, people want to make B as small as possible. For example, if $B = \sqrt{2}$ is employed, all angles are bounded between 20.7° and 138.6°.

Because the quality of the mesh depends solely on how the points which defined the DT are distributed, so the central question of any Delaunay refinement algorithm is how to choose the next Steiner point. The common answer to this question is to insert the next point as

far from other points as possible. If a new point inserted is too close to another point, the resulting small edge will engender thin triangles. For a bad triangle t, a graceful way (can date back at least to mid-1980s [Fre87]) to eliminate it is to insert a Steiner point at its circumcenter and use Lawson's algorithm [Law77] or the Bowyer-Watson algorithm [Bow81] to maintain the triangulation being a DT. After the circumcenter's insertion, the bad triangle cannot survive, because its circumcircle is no longer empty. In the literature, there are other different kinds of techniques on how to choose the next Steiner point. All of them will be introduced in Section 3.1.

Another big challenge to the Delaunay refinement algorithm is to ensure termination of the algorithm while still obtaining good quality mesh. Due to small angles from the input, the Delaunay refinement algorithm cannot always terminate. Small angle inherent in the input geometry cannot be removed, and it is not possible to triangulate a domain without creating any new small angles as pointed by Shewchuk in [She00]: for any angle bound $\theta > 0$, there exists a PSLG X such that it is not possible to triangulate X without creating a new corner (not present in X) whose angle is smaller than θ. This statement imposes a fundamental limitation on any triangle mesh generation algorithm. If no input angle is smaller than 60°, Ruppert's algorithm [Rup95] can guarantee to terminate, and have no angle smaller than k (which is no larger than $\arcsin \frac{1}{2\sqrt{2}} \approx 20.7°$, and all angles are bounded between 20.7° and 138.6°). Similarly, for the same inputs as mentioned above, Chew's second Delaunay refinement algorithm [Che89b] can also terminate, and all angles are between 30° and 120°. Actually, most of existing algorithms work well in practice for many applications, but all of them would fail on some difficult triangulation domains. Therefore, in the thesis, we only focus on Delaunay refinement algorithm for DT mesh of a point set in which the mesh is segment-bounded.

1.3 GPU in Computational Geometry

Driven by the demand for realtime, high-definition 3D graphics, the programmable graphic processor unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 1.6. The discrepancy in floating-point capability between the CPU and the GPU is that more transistors are devoted to data processing rather than data caching and flow control on the GPU, as schematically illustrated by Figure 1.7.

Researchers and developers are becoming more and more interested in doing general purpose computation on GPUs, where the GPUs are used not only for graphics processing tasks but also general computations in many disciplines. These range from numerical analysis and physical simulation to data mining and computational geometry.

The GPU is well-suited in addressing problems that can be expressed as data-parallel com-

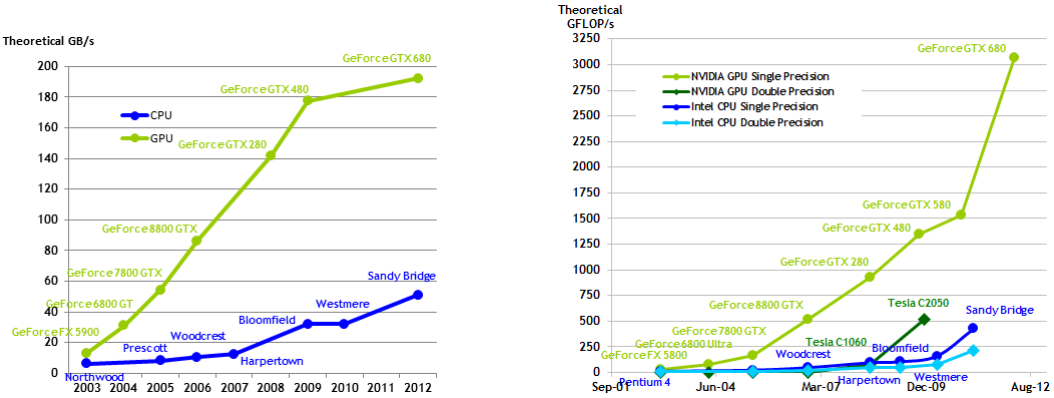


Figure 1.6: (a) Computational power and (b) memory bandwidth of the CPU and GPU (NVIDIA 2012 [NVI13]).

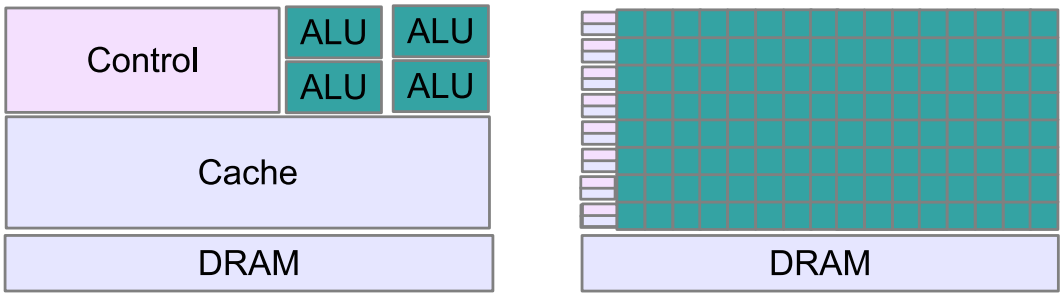


Figure 1.7: The GPU devotes more transistors to data processing.

putations, i.e., the same computation is performed on many data elements in parallel, especially when a high amount of arithmetic computation is involved. In addition, such a trend is accelerated with the recent introduction of CUDA, a general purpose parallel computing architecture, by NVIDIA where developers can now access with ease the full power of GPUs for more complex tasks.

In the field of computational geometry, GPU has been employed to solve some problems. Early works include computing the digital Voronoi diagram (VD) [HKL+99, FG06, CTMT10], a structure that is closely related to the DT. These works also mentioned the possibility of obtaining the latter from the former straightforwardly. However, we notice that the Voronoi diagram in a digital space (of a texture) is not exactly the dual of the DT in a continuous space, and only until recently, Rong et al. [RTCS08] present a serious attempt to derive the DT from the digital VD. Their algorithm, however, is hybrid, since parallel computation is only used in the first part while leaving the rest to a sequential algorithm. After that Qi et al. [QCT12, QCT13] propose complete GPU algorithm for DT computation. Recently Cao et al. [CNGT14] propose a GPU algorithm, which can construct DT in both 2D and 3D

space. In the thesis, we call the algorithm which can generate the DT for a set of points in 2D space as GPU-DT.

As for the CDT and quality mesh problems, there is no efficient GPU solution as far as we know. This is partly because such problems do not present themselves readily to parallel computation. A parallel algorithm, in order to fully utilize the GPU hardware, usually needs to have regularized work and localized data access. It is not clear how to achieve those criteria while adapting the traditional and usually complex parallel techniques, such as divide-and-conquer, to both these problems.

1.4 Objectives and Contributions

Motivated by the rapid increase of the performance of GPU, the flexibility of the parallel programming model CUDA and the observation that there is no known algorithm and implementation of a 2D mesh generator on the GPU, the objective of this work is to develop a 2D mesh generator on the GPU. The major contributions of this work are listed in the following:

1. Algorithm GPU-QM: A new and efficient approach for computing 2D quality triangle mesh on the GPU. It is the first GPU solution for this problem. According to our experiment results, this algorithm can handle both synthetic and real-world data very well. When compared to *Triangle*, our algorithm can generate meshes with similar quality as the meshes generated by *Triangle*, and runs up to 5.5 times faster. Furthermore, this algorithm can offer both termination and quality guarantees.

2. Algorithm GPU-CDT: The first GPU solution to compute the 2D CDT of a PSLG consisting of points and edges. Our implementation runs up to two orders of magnitude faster than the best sequential implementations on the CPU. This result is reflected in our experiment with both randomly generated PSLGs and real-world GIS data having millions of points and edges. Furthermore, we can prove that the algorithm is guaranteed to terminate for any given PSLG, and the total number of flips performed by the algorithm is $\Theta(n^2)$, where n is the number of input points.

3. A software combining GPU-DT, GPU-QM, GPU-CDT together. It is the first GPU mesh generator so far. Similar to *Triangle* software, this software can generate exact DT, CDT, conforming Delaunay triangulations, digital Voronoi diagrams, and high-quality triangle meshes. But our algorithms are parallel which is implemented using the CUDA programming model on nVidia GPUs. The experiment results of the program show that it is numerically robust and runs much faster than any existing CPU programs such as *Triangle* and CGAL. The codes for all the algorithms have also been made freely available to the public.

4. Several key GPU techniques, such as handling exact arithmetic and robustness, handling cooperation among threads, dealing with racing conditions using multiple iterations. The GPU techniques have been carefully documented in the thesis, and provide valuable references for future research.

1.5 Outline

The rest of the thesis is organized as follows. Chapter 2 introduces some basic definitions, data structures, and GPU programming considerations. The algorithms GPU-QM and GPU-CDT would be introduced in Chapter 3 and Chapter 4, respectively. Finally, Chapter 5 concludes the thesis.

Chapter 2

Preliminaries

Before introducing our algorithms in the following chapters, in this chapter we introduce some basic definitions and properties about them in Section 2.1. Section 2.2 describes the common data structure used in our mesh generator. Since all the algorithms proposed in the thesis are based on GPU, some related programming considerations are discussed in Section 2.3. At the end, Section 2.4 presents the experimental environment used in the thesis.

2.1 Terminology and Definition

Definition 2.1.1 (Voronoi diagram). *Let S be a set of n sites in the Euclidean space of $\Re^2$. For each site p of S, the Voronoi region $\mathcal{R}(p)$ of p is the set of points that are closer to p than to other sites of S. The Voronoi diagram $\mathcal{V}(S)$ is the space partition induced by Voronoi regions. The elements of S are also called sites of this Voronoi diagram. The line segments shared by the boundaries of two Voronoi regions are called Voronoi edges, and the points shared by the boundaries of three or more Voronoi regions are called Voronoi vertices.*

In the discrete space, instead of a continuous plane, we only consider the set of all integer

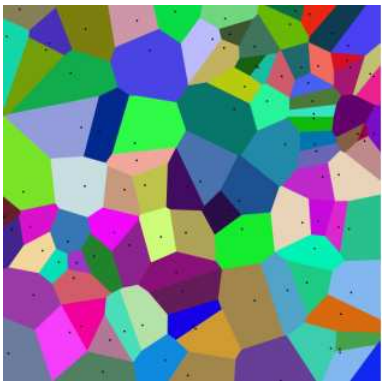


Figure 2.1: Digital Voronoi diagram.

grid points. If a grid point p lies inside the Voronoi region of the site x_i, and we say that p is colored by x_i. In case p is equal distance from x_i and x_j and $i < j$, we color p by x_i. The set of colored grid points form the discrete Voronoi diagram $\mathcal{D}(\mathcal{S})$ of S (see Figure 2.1). We call this procedure Euclidean coloring.

Definition 2.1.2 (Planar graph). *Let V be a finite set of n vertices in $\Re^2$, and let E be a set of edges determined by the vertices of V. A planar graph is the pair $\mathcal{G} = (V, E)$ that satisfies:*

(i) For each edge $ab \in E$, $ab \bigcap V = \emptyset$, and

(ii) For each edge pair $ab \neq cd$ in E, $ab \bigcap cd = \emptyset$.

Definition 2.1.3 (Planar Straight Line Graph - PSLG). *PSLG is a graph in which the vertices are embedded as points in the Euclidean plane, and the edges are embedded as non-crossing line segments.*

By definition, a PSLG contain both endpoints of every segment and a segment may intersect vertices and other segments only at its endpoints. Figure 2.2 shows an example of PSLG.

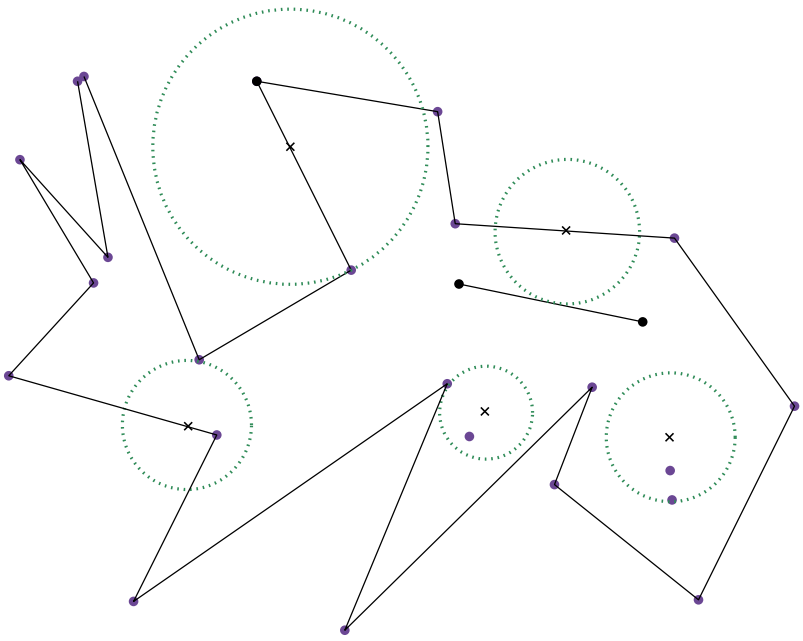


Figure 2.2: A PSLG consists of points and segments. The radius of each disk illustrated here is the local feature size of the point at its center.

Definition 2.1.4 (Local feature size). *Given a PSLG X, the local feature size $lfs(p)$ at any point p is the radius of the smallest disk centered at p that intersects two nonincident vertices or segments of X. Two distinct features, each a vertex or segment, are said to be incident if they intersect.*

Figure 2.2 illustrates the notion of local feature size by giving examples for a variety of points. For each point marked with "cross" in the figure, the corresponding disk is the local

feature size of the point.

Proof. The disk having radius $\text{lfs}(u)$ centered at u intersects two nonincident features of X. The disk having radius $\text{lfs}(u) + |uv|$ centered at v contains the prior disk, and thus also intersects the same two features. Hence, the smallest disk centered at v that intersects two nonincident features of X has radius no larger than $\text{lfs}(u) + |uv|$. □

Definition 2.1.5 (Triangulation). *Triangulation is a PSLG $\mathcal{G} = (V, E)$ such that E is maximal.*

A special case of triangulation is Delaunay triangulation as shown in Figure 2.3.

Definition 2.1.6 (Delaunay Triangulation). *A triangulation $\mathcal{G} = (V, E)$ is Delaunay if all edges $ab \in E$ satisfy the so-called empty circle property (with respect to the set of points V) that is to say, there is a circle that passes through a and b such that the other points of V are exterior to the circle.*

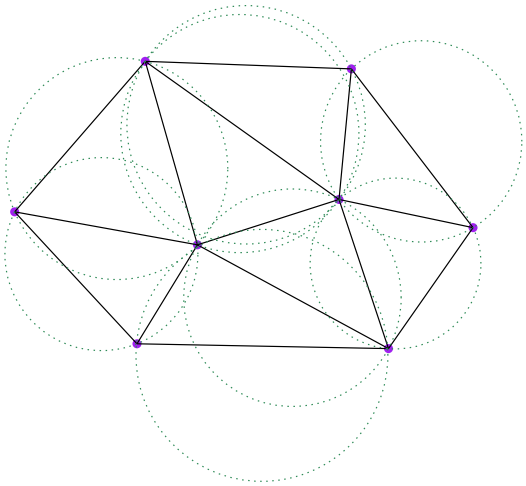


Figure 2.3: An example of DT of a set of points. No visible point is inside the circumcircle of any triangle in the triangulation.

In the non-degenerate case, i.e., no four or more points are co-circular, the Delaunay triangulation for a given set of points is unique. For other degenerate cases, the Delaunay triangulation is not unique, i.e., there are more than one Delaunay triangulation. Then any of them will be acceptable. Among all triangulations of a finite point set $S \subseteq \Re^2$, the Delaunay triangulation maximizes the minimum angle of all the angles of the triangles in the triangulation.

The *star* of a vertex p (St_p) consists of all triangles that contain p. The *link* of p consists of all edges of triangles in the star that are disjoint from p (see Figure 2.4a). Let $p \notin S$ be a point in the interior of the convex hull of S, and assume that $S \bigcup \{p\}$ is non-degenerate. Let D be the DT of S and D_p be the DT of $S \bigcup \{p\}$. The *prestar* of p (Pt_p) consists of all triangles whose circumcircles enclose p (see Figure 2.4b). We can get D_p from D by

substituting the star for the prestar, $D_p = (D - \mathrm{Pt}_p) \bigcup \mathrm{St}_p$. Actually it is the Watson's algorithm [Wat81] for generating DT.

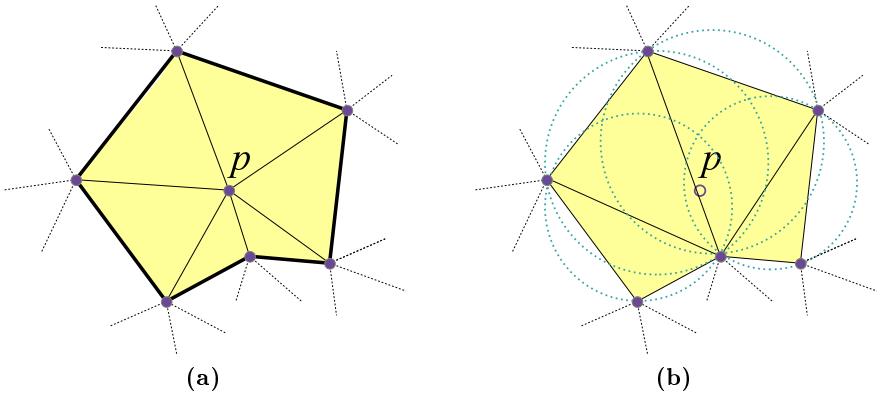


Figure 2.4: (a) Star of p, and the bold edges bound shaded triangles is the link of the star. (b) Prestar of p.

Definition 2.1.7 (Locally Delaunay). *Let S be a point set, T be a triangulation of S. An edge $ab \in T$ is locally Delaunay if*

(i) it belongs to only one traingle and therefore bounds the convex hull of S, or

(ii) it belongs to two triangles, abc and abd, and d lies outside the circumcircle of abc.

Definition 2.1.8 (Delaunay Lemma). *If every edge of T is locally Delaunay then T is the DT of S.*

Edge flip is a locally operation proposed in [Law77]. If ab belongs to two triangles, abc and abd, whose union is a convex quadrangle, then we can flip ab to cd, see Figure 2.5. According to the Delaunay Lemma, we can use edge flips as elementary operations to convert an arbitrary triangulation T to the DT.

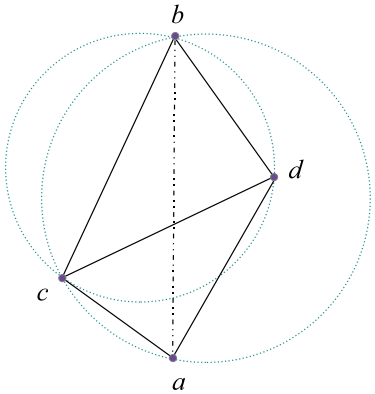


Figure 2.5: Flipping ab to cd. Before flip, ab is not locally Delaunay, and the union of the two traingles abc and abd is convex. After flip, cd is locally Delaunay.

There are many sequential algorithms developed for the CPU to compute the DT [Aur91, For97, SSD97]. All these algorithms in general follow one of the three well-known paradigms: divide-and-conquer [Dwy87], sweep-line [For87] and incremental insertion [GKS92].

a. *Divide-and-Conquer.* Algorithms based on this strategy recursively divide a set of points into two smaller sets, until a set is small enough to compute trivially its DT. Then it merges recursively the results of two small adjacent sets into that of a bigger one, until results of all sets are grouped in the triangulation. Using this approach, the DT can be built in optimal $O(n \log n)$ time [SH75, Dwy87].

b. *Sweep-line.* The Voronoi diagram and DT are dual to each other. Fortune [For87] uses a sweep-line algorithm to construct the Voronoi diagram, from which the DT is obtained. First, the algorithm sorts the input points according to their x-coordinates, then a vertical line, called the *sweep-line*, is swept from left to right. Points behind the sweep-line are already added into the Voronoi diagram, while points ahead of the sweep-line are waiting for processing. As the sweep-line progresses, the Voronoi edges are generated incrementally. The running time of this algorithm is also $O(n \log n)$.

c. *Incremental Insertion.* A natural way to efficiently compute the DT is to repeatedly add points one at a time, re-triangulate the affected parts of the triangulation. To insert a point, we first locate the triangle or edge containing the point. The new point splits the triangle containing itself into three triangles, or the two triangles adjacent to the edge containing itself into four triangles. Subsequently, we perform edge flipping to maintain the triangulation being a DT. Although this incremental insertion approach runs in $O(n^2)$ time in the worst case, the expected time complexity can still be $O(n \log n)$, provided that the points are inserted in a random order [GKS92].

For the GPU algorithm, there are a few recent works, such as [RTCS08, CNGT14]. Since our algorithms proposed in the chapter are GPU algorithms, we can choose any one of them.

Definition 2.1.9 (Visibility). *Two vertices a and b of a graph $\mathcal{G} = (V, E)$ are visible from each other (in $\mathcal{G}$) if the line segment ab does not intersect any of the edges of E.*

Definition 2.1.10 (Constrained Delaunay Triangulation). *Let $\mathcal{G} = (V, E)$ be a planar graph with $E \neq \emptyset$. A triangulation $(V, E \cup E')$ is a constrained Delaunay triangulation of $\mathcal{G}$ if the edges $ab \in E'$ are such that a and b are visible in $\mathcal{G}$, and ab fulfils the empty circle property with respect to the vertices only visible from a and b.*

From here, we know that if $\mathcal{G}$ has no edges then constrained Delaunay triangulation is the same as the Delaunay triangulation. From the definition we can see that the constrained Delaunay triangulation is a triangulation which is as close as possible to the Delaunay triangulation. Figure 2.6 shows an example of constrained Delaunay triangulation.

2.2 Data Structure

Figure 2.7b shows the data structure of the triangulation showed in Figure 2.7a. Throughout the phases of constructing DT, CDT and quality mesh, we need to frequently walk from triangle to triangle, or around the triangle fan of a vertex. As such, we have to maintain two data structures. First, for each triangle we always maintain the link to its three neighbors. The three vertices of a triangle are indexed with 0, 1 and 2. The neighbors of a face are also indexed with 0, 1, 2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index. Second, for each vertex, we maintain a link list of all triangles incident to that vertex.

Any time when the triangulation is changed, we have to keep updating these data structures. Like *Triangle*, all triangles in our algorithm are oriented triangles. Three vertices of an oriented triangle are called origin vertex, destination vertex, apex vertex respectively. An oriented triangle includes a pointer to a triangle and orientation. There are three possible orientations for a triangle abc, that is abc, bca, and cab (all are defined in counterclockwise order). Suppose the orientation of triangle abc is abc, that means a is the apex vertex of triangle, b is the origin vertex of triangle, and c is the destination vertex of triangle (see Figure 2.7c). In practice, we can use an edge to denote the orientation of a triangle, and use the orientated triangle to denote an edge of the triangle. For example, the orientated

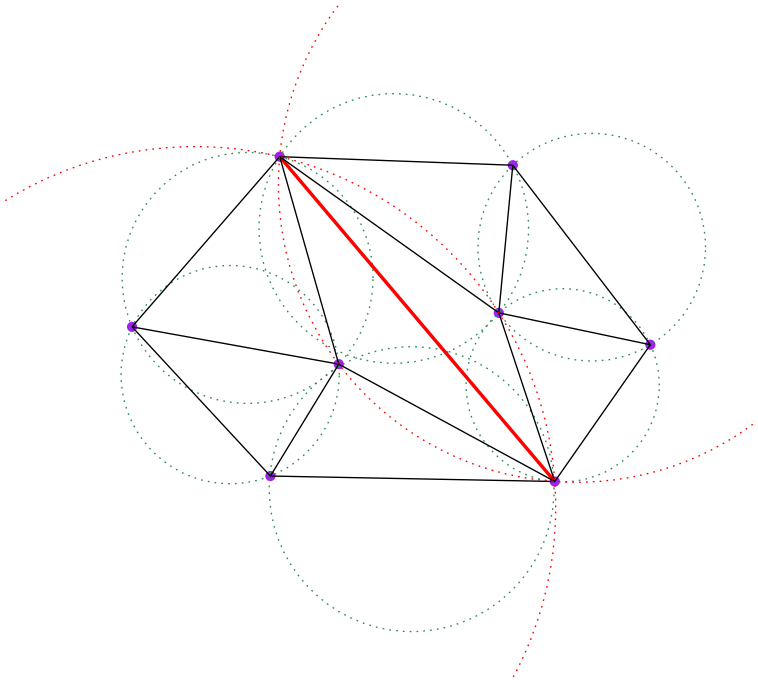


Figure 2.6: An example of CDT of a PSLG which consists of points and one constraint (red line). No visible point is inside the circumcircle of any triangle in the triangulation.

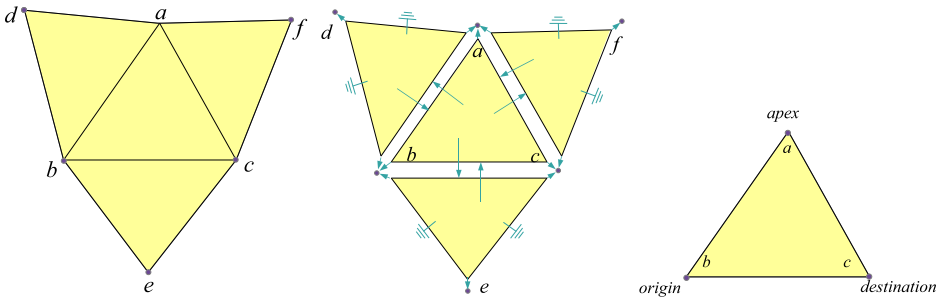


Figure 2.7: (a) A triangulation. (b) Data structure of the triangulation. (c) An orientated triangle.

Operation	Result	Usage
sym(abc)	bad	Find the abutting triangle sharing an edge ab/ba
lnext(abc)	bca	Find the next edge (counterclockwise) of a triangle
lprev(abc)	cab	Find the previous edge (clockwise) of a triangle
onext(abc)	acf	Find the next edge counterclockwise with the same origin
oprev(abc)	adb	Find the next edge clockwise with the same origin
dnext(abc)	dba	Find the next edge counterclockwise with the same destination
dprev(abc)	cbe	Find the next edge clockwise with the same destination
rnext(abc)	fac	Find the next edge (counterclockwise) of the adjacent triangle
rprev(abc)	bec	Find the previous edge (clockwise) of the adjacent triangle
org(abc)	a	Find the origin vertex
dest(abc)	b	Find the destination vertex
apex(abc)	c	Find the apex vertex
bond(abc, bad)	-	Connect two triangles sharing an edge ab/ba

Table 2.1: Primitive operations according to the triangulation in Figure 2.7a.

triangle abc denotes the edge bc of the triangle, and edge bc denotes the orientated triangle abc. Table 2.1 shows some primitive operations defined on an orientated triangle abc.

In the algorithms introduced in the next two chapters, every point, constraint, and triangle has a unique index. For example, there are n points in all, then each point has a index number ranging from 0 to $n-1$.

2.3 GPU Programming Consideration

CUDA is a scalable parallel programming model and a software environment for parallel computing. In the CUDA model, the CPU is called the *host* and is connected to one or more CUDA-capable GPUs called *devices*. A CUDA device is composed of one or more streaming multiprocessors (SM). Each SM is composed of many streaming processors (SP), typically eight SPs per SM. The CUDA programming language is an extension to C and C++ with some extra syntax. Programmer can define parallel functions, called *kernels* executed on the device using CUDA programming language.

A typical cycle of a CUDA program is as follows.

1. Allocate memory of the GPU.

2. Copy the data from the CPU to the GPU.

3. Configure the thread configuration: choose the correct block and grid dimension for the problem.

4. Launch the threads configured.

5. Synchronize the CUDA threads to ensure that the device has completed all its tasks before doing further operations on the GPU memory.

6. Once the threads have completed, data is copied back from the GPU to the CPU.

7. Free the GPU memory.

In all, GPU is massively multithreaded, with hundreds of processors. To effectively utilize the GPU, it is desirable to have tens of thousands of executing threads at any given time. As such, we keep in mind the following two design principles in developing algorithms for the GPU.

First, the GPU architecture is most suitable for data-parallel computations, in which the same computation is performed on multiple pieces of data by many threads. Therefore, we need to make our thread code (or algorithm) as simple (with little complicated control flow) and as uniform (with similar amount of work across various threads) as possible.

Second, with so many threads, cooperation among threads, conflicting data access, and racing conditions are serious problems. To mitigate this, we usually employ some simple checks to break the set of tasks into several groups, within which the tasks can be done

concurrently with no or little conflict. Figure 2.8 shows a possible conflict in parallel processing when inserting points into the triangulation. In this figure, point a is supposed to be inserted on the edge adjacent to triangle A and C; point b is supposed to be inserted on the edge adjacent to triangle A and B; c should be inserted on the edge adjacent to B and C. Obviously, if there is more than one point which need to be inserted into a same triangle, only one point can be processed at a time. But if a point is inserted into an edge, it will change two triangles in the triangulation. So we need to make sure that no other thread is trying to update these triangles. In order to solve the conflict between insertions, we need to break the insertions into several rounds. At each round, we want to insert as many points as possible without any conflict. An array X is needed for the triangles in the triangulation, and every round we do the following two steps:

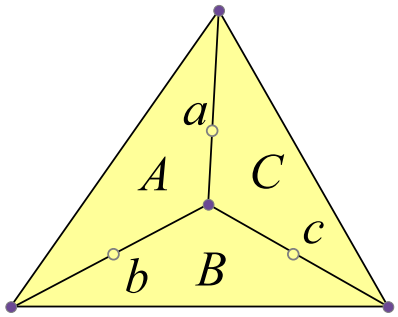


Figure 2.8: Possible conflict in parallel processing when inserting points into the triangulation.

1. Locate the triangle or edge which contains the point p. If the point is contained in a triangle t, mark the triangle with $X[t] = \min(X[t], p)$. If p lies on an edge shared by two triangles t_1 and t_2, we atomically mark both triangles with $X[t_1] = \min(X[t_1], p)$ and $X[t_2] = \min(X[t_2], p)$. For the case shown in Figure 2.8, we mark three triangles as: $X[A] = \min(X[A], a)$, $X[A] = \min(X[A], b)$, $X[B] = \min(X[B], b)$, $X[B] = \min(X[B], c)$, $X[C] = \min(X[C], c)$, $X[C] = \min(X[C], a)$. In CUDA, when several threads try to write to $X[t]$, it is guaranteed one of them will succeed. The marking is done using the *atomic minimum* operation, which is readily available on the GPUs.

2. Using the array X to decide whether to insert p in this round or not. p can be processed in this round if all the marks written by it (either one or two) are not overwritten. For the case shown in Figure 2.8, the point with smallest index among a, b and c will be inserted in this round.

2.4 Experimental Environment

We implement our algorithms by using the CUDA programming model by NVIDIA [NBGS08]. All the experiments are conducted on a PC with an Intel i7 2600K 3.4GHz CPU, 16GB of

DDR3 RAM and an NVIDIA GTX580 Fermi graphics card with 3GB of video memory. CPU and GPU used here were both top-of-the-line at the time of experiment. Visual Studio 2008 and CUDA 5.5 Toolkit are used to compile all the programs, with all optimizations enabled. In the thesis we compare the results from our algorithms with the results from the best-known sequential triangle mesh generators *Triangle* [She96a] and *CGAL* [CGA11].

Triangle A 2D mesh generator, which can generate DT, CDT, conforming Delaunay triangulations, Voronoi diagrams, and high-quality triangle meshes. It has thousands of users, with applications ranging from radiosity rendering and terrain databases to stereo vision and image orientation, as well as dozens of variants of numerical methods.

CGAL The goal of the CGAL Open Source Project is to provide easy access to efficient and reliable geometric algorithms in the form of a C++ library. CGAL is used in various areas needing geometric computation, such as: computer graphics, scientific visualization, computer aided design and modeling, geographic information systems, molecular biology, medical imaging, robotics and motion planning, mesh generation, numerical methods and so on.

According to our experiment results, usually *Triangle* runs faster than CGAL, especially on the Delaunay refinement problem and CDT construction problem. So in the thesis, we always compare our algorithms with *Triangle* unless otherwise stated.

Chapter 3

A GPU Algorithm for Delaunay Refinement

In this chapter, a Delaunay refinement algorithm termed as GPU-QM is proposed, which is the first GPU solution for improving the mesh quality of DT of a set of points so far. According to the experiments on the synthetic and real-world data sets, our GPU algorithm outperforms *Triangle* by 2 to 3 times speedup. The quality of the mesh generated by our algorithm is very similar to the mesh generated by *Triangle*, only less than 1% more Steiner points are inserted in our algorithm. Furthermore, we proposed one variant algorithm called GPU-QM-V based on the GPU-QM algorithm. In this variant algorithm, a new metric for the minimum separation bound between any two Steiner points is used. When using the GPU-QM-V algorithm, we can almost double the performance (4 to 5.5 times speedup) with increasing no more than 4% Steiner points compared to *Triangle*. In addition both GPU-QM and GPU-QM-V algorithms are guaranteed to terminate and numerically robust.

In this chapter, we use B as the upper bound of circumradius-to-shortest edge ratio, and any triangle whose circumradius-to-shortest edge ratio is bigger than B is a bad triangle. In the following sections, we first introduce some related works in Section 3.1. In Section 3.3, an incomplete version of GPU-QM (called basic algorithm) is discussed. In this basic algorithm, we ignore the boundary refining problem, because in order to handle boundary refining, several special mechanisms are needed which would divert attention away from the main framework of the GPU-QM algorithm. In Section 3.4 we add several mechanisms to the incomplete version, such that the boundary edges can be handled. In Section 3.5, GPU-QM-V, the variant algorithm of the GPU-QM is discussed. All implementation details and key GPU techniques used in the algorithms are shown in Section 3.6. In Section 3.7, we compare our results with the fastest sequential implementation on both synthetic and real-world data. Finally, Section 3.8 concludes the chapter.

3.1 Related Works

As mentioned before, the central question of any Delaunay refinement algorithm is where the next point should be inserted. A reasonable answer is that the new point should be inserted as far from other vertices as possible. According to the technique used to choose Steiner

points, there are three different strategies/algorithms, which are termed as *circumcenter-insertion*, *off-center-insertion*, and *sink-insertion*, respectively. These three strategies will be introduced in the following sections. In addition, we will review some parallel Delaunay refinement algorithms in the end.

3.1.1 Circumcenter-insertion

In this kind of the algorithm, the main operation is to insert a point at the circumcenter of a bad triangle (see Figure 3.1), and maintain the DT by using Bowyer-Watson's algorithm to re-triangulate the prestar of the new point, or using Lawson's algorithm to do edge-flip on non-locally Delaunay edges.

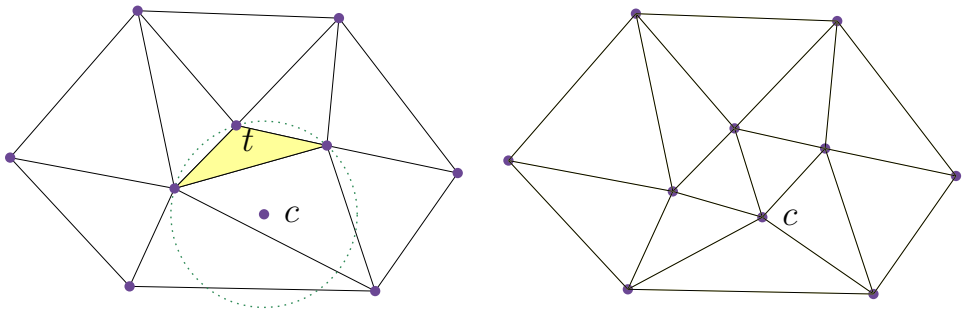


Figure 3.1: Any triangle whose circumradius-to-shortest edge ratio is larger than B is split by inserting its circumcenter. Every new edge has length at least B times that of shortest edge of the bad triangle. Left: before point insertion. Right: after point insertion.

There are many Delaunay refinement algorithms using this strategy to insert Steiner points in the literature. Ruppert's Delaunay refinement algorithm [Rup95] and Chew's second Delaunay refinement algorithm [Che89b] are the best-known algorithms, which perform well in practice and have provable guarantees on both mesh quality and termination. Most of other algorithms ([She00, Ü09, MPW03, EG01]) follow the ideas of these two algorithms. Here we only detail Ruppert's algorithm in the following.

A segment is said to be *encroached* if a point other than its endpoints lies on or inside its diametral circle. Any encroached segment should be split into two segments by inserting a point at its midpoint. For example, in Figure 3.2, point p is inside the diametral circle of segment s, i.e., s is encroached by p. After splitting s at its midpoint with point v_1, the segment s becomes two segments. If the left segment of s is still encroached by p, the left segment should be split again by inserting a point at its midpoint v_2.

Ruppert's algorithm starts from the DT computation for a given PSLG. If there is any encroached segment, split the encroached segment at its midpoint, and update the DT immediately until no segment is encroached. Then we need to check all triangles in the mesh. If there is any bad triangle, split the bad triangle by inserting point at its circumcenter, and update the DT immediately until no triangle is a bad one. Notice that encroached segments

have higher priority than bad triangles, which means if the circumcenter of a bad triangle encroaches one or more segments, the circumcenter should be rejected, and the encroached segments should be split in advance. As for the order in which segments are split, or bad triangles are split, is arbitrary. Ruppert's algorithm uses Lawson's algorithm [Law77] to maintain the Delaunay property of the triangulation.

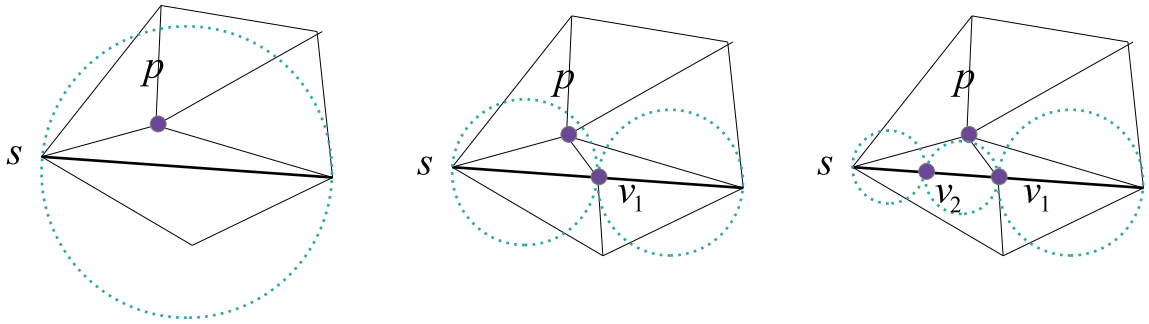


Figure 3.2: Segment is split recursively until no segment is encroached.

For input with no angle is less than 60° and using $B = \sqrt{2}$ as the upper bound of the circumradius-to-shortest edge ratio of the mesh, the triangulation generated by Ruppert's algorithm conforms to the input, and has no angle smaller than k (I.e., no angle is larger than $\arcsin \frac{1}{2\sqrt{2}} \approx 20.7°$, and all angles are bounded between 20.7° and 138.6°).

Notice that, the order of inserting Steiner points is very important. As pointed in [She96a], a heap of bad triangles indexed by their smallest angle confers a 35% reduction in mesh size over a first-in-first-out queue.

3.1.2 Off-center-insertion

Recently, Üngör [Ü09] proposed a new type of Steiner point which he calls *off-center* for the Delaunay refinement algorithm (see Figure 3.3). Given a bad triangle pqr with shortest edge pq and circumcenter c_1, we define the off-center to be the circumcenter c_1 if the circumradius-to-shortest edge ratio of pqc_1 is smaller than or equal to the upper bound B. Otherwise, the off-center is the point c on the bisector of pq (and inside the circumcircle of pqr), which makes the circumradius-to-shortest edge ratio of the triangle pqc exactly B. The main difference of this algorithm from Ruppert's algorithm is that this algorithm always splits a low quality triangle at its off-center other than its circumcenter.

The author proves that the new algorithm has the same quality and size optimality guarantees as the best known Delaunay refinement algorithm. In practice, the new algorithm inserts about 40% fewer Steiner points (hence runs faster) and generates triangulations that have about 30% fewer elements compared with the best previous algorithms.

Since the computation of circumcenters may run into numerical problems in practice, using this approach can make the program more robust with improved quality of the mesh. Actu-

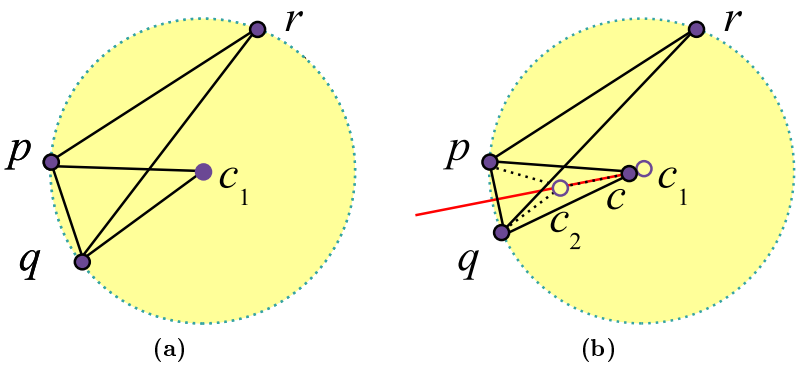


Figure 3.3: The off-center and the circumcenter of triangle pqr is labeled as c and c_1 respectively. (a) If $|c_1p| \leq B|pq|$, then $c = c_1$. (b) Otherwise, $c \neq c_1$ and c is obtained by setting $|c_2p| = B|pq|$, where c_2 is the circumcenter of triangle pqc.

ally, this approach is employed by *Triangle*. Notice that *Triangle* adds a coefficient of 0.95 to the formula of computing the off-centers to improve robust in practice, i.e., the formula of computing c in Figure 3.3 is modified to $|c_2p| = 0.95 \times B \times |pq|$, where $B = \sqrt{2}$.

3.1.3 Sink-insertion

Edelsbrunner and Guoy [EG01] propose sink-insertion as a new technique to improve the mesh quality of DT. In 2D space, a triangle is called a *sink triangle* if its circumcenter is contained in the interior of the triangle. In addition, the circumcenter of a sink triangle is called a *sink*. The basic idea of the algorithm is to eliminate bad triangles by adding sinks as new points to the DT.

Starting from any bad triangle, we need to find a sink to be inserted. For example, for an existing bad triangle t_0, we start a walk along the direction of t_0's circumcenter to the next triangle t_1. Then starting from t_1, do similar walk until we find a sink triangle t_i. In the end, t_i's circumcenter c is the sink to be inserted in the algorithm. Figure 3.4 illustrates how to find a sink from the bad triangle t_0.

The authors claimed that sink-insertion creates about the same mesh quality as circumcenter-insertion algorithm, but it does this in a more economical manner. This is because bad triangles tend to cluster and share sinks. Instead of dealing with a large number of circumcenters, this algorithm only works with a small number of sinks. In addition, the sinks tend to be well separated and thus exhibit fewer dependencies, which is desirable in parallel implementation.

This sink insertion algorithm studies the effect of using sinks instead of circumcenters in the Delaunay refinement algorithm for improving the mesh quality. However, their experiment

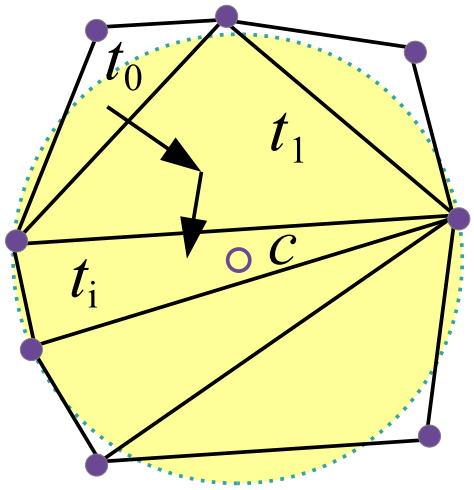


Figure 3.4: Circumcenter c is the sink. The arrow shows how to find the sink from a bad triangle t_0.

results show that the restriction to sinks surprisingly has little effect on the mesh quality.

The authors thought the sink-insertion algorithm should be more economical than the circumcenter-insertion algorithm, because bad triangles tend to cluster and share sinks. Instead of dealing with a large number of circumcenters, they can therefore work with a small number of sinks. However in practice, there are few bad triangles sharing a common sink triangle according to our experiment results on both synthetic and real-world data. Even though some bad triangles cluster at the beginning, after one or two point insertion iterations, few bad triangles may still cluster together. Actually, when the circumcenters are inserted one by one, one circumcenter insertion may remove more than one bad triangle. In other words, circumcenter-insertion may have the same effect as the sink-insertion method. So the sink-insertion method has little effect on reducing the number of Steiner points. Actually, we have tried to implement this method in parallel on the GPU, and the experiment results show that there is little difference between this method and circumcenter-insertion method.

3.1.4 Parallel algorithms

In literature, there are many parallel Delaunay refinement algorithms, such as [STU+02, LC99, LT01, CN99, STU04] and so on. All of these algorithms employ Ruppert's or Chew's second Delaunay refinement algorithm. Although the details of these algorithms are different, most of them employ a simple strategy: at each iteration, they choose a set of independent points to insert into the domain, and then update the DT or CDT. The criteria of choosing independent data set are different for each algorithm, but all of them show that their set of independent points can be constructed efficiently in parallel. In addition,

the number of iterations is a function of L and s, where L is the diameter of the domain, and s is the length of the smallest edge in the output mesh. Each independent set can be handled by Ruppert's or Chew's method. Furthermore, some of these algorithms can be generalized to three-dimension. All these algorithms follow the divide-and-conquer paradigm.

However, a parallel algorithm, in order to fully utilize the GPU hardware, usually needs to have regularized work and localized data access. It is not clear how to achieve these criteria while adapting the divide-and-conquer strategy.

Recently, Nasre et al. [NBP13] claim their GPU algorithm for Delaunay refinement could gain up to 80 times speedup over the sequential implementation. In their algorithm, they tried to insert circumcenters for bad triangles in parallel. Before a particular circumcenter c is inserted, all triangles whose circumcircles enclose c should be found and marked. Then they delete marked triangles, and re-triangulate the deleted region after inserting c. When all circumcenters are inserted at the same time, one triangle can be marked by more than one circumcenter. Under this situation, only one circumcenter can be inserted, other circumcenters should wait. However, we cannot reproduce the same results as they claimed when trying their source code called *LonestarGPU*. We tried to run their code on uniform distributed points as required by their code. However, we found that the output triangulation is not a DT at all.

3.2 Issues for a GPU Delaunay Refinement Algorithm

Generally, there are several issues that need to be considered when designing a GPU algorithm for Delaunay refinement algorithm.

1. How to select Steiner points in parallel. Should we simply select all bad triangles' circumcenters, or select only a particular part of them?

2. How to ensure all Steiner points obtained from the first question are pairwise independent. Or is it too aggressive to limit all Steiner points should be pairwise independent? Is there any other metric to define the minimum separation between any two Steiner points?

3. How to simulate the priority queue used in the sequential implementation on the GPU efficiently.

4. How to handle non-degenerate cases to get robust result.

5. How to handle the boundary refining, and make sure the algorithm can terminate.

Of course the last but not the least issue is how to make the GPU algorithm run fast. In the next section, we will discuss the basic algorithm of GPU-QM, which is a framework of the whole algorithm and can solve all issues mentioned above except for the fifth one. In order to solve the fifth issue, we propose several mechanisms for the basic GPU-QM algorithm,

which will be introduced in Section 3.4. In Section 3.5, we will consider the second issue again to make the GPU-QM algorithm run faster.

3.3 The Basic Algorithm - GPU-QM

In this section, we focus on the basic algorithm/framework of the GPU-QM algorithm. The input is a stable complex of a DT for a set of points S, which by definition is a subcomplex whose circumcenters all lie in the interior of its underlying space. We use $B = 1$ as the upper bound of circumradius-to-shortest edge ratio, so in the output DT, all angles are larger than or equal to 30°. Although there are many algorithms we can use to generate the initial DT of S, considering our algorithm is a GPU algorithm, we choose to use the GPU algorithm mentioned in [QCT12] to generate the DT for S. Mechanisms for refining the boundary is not included in the basic algorithm and will be discussed later in Section 3.4.

3.3.1 Motivation and algorithm overview

Recall the three steps for a sequential Delaunay refinement algorithm. Let us consider the normal case with no degeneracy. For a bad triangle t, we should compute its circumcenter c in the first, and find the container for c (do point location). Then we insert c into the mesh and maintain the mesh to be a DT by using edge-flips. Repeat these steps until no more bad triangle exist. Since we can always simulate the sequential algorithm on the GPU, one naïve method can be designed as follows:

Naïve method 1: Let one thread handle one triangle. For any bad triangle t, compute its circumcenter c, and do point location for c to find its container triangle. Then insert all c into the mesh, using atomic operation to void inserting more than one Steiner point into a same triangle. In the end, do edge-flip in parallel to maintain DT.

When one Steiner point p is inserted, a region near p should be modified, and this region is called *influence region* of p. For a Steiner point p, its influence region is the union of simplices in the closure of its star in the DT mesh after inserting p, or in the closure of its prestar in the DT mesh before inserting p. When many Steiner points are inserted in parallel, their influence regions may overlap. Points whose influence regions are pairwise disjoint are pairwise *independent*.

This method may lead to many more Steiner points comparing to the result from the sequential implementation, and even infinite loop problems. For example, assuming p and q are two Steiner points inserted in the same round for eliminating bad triangle t_p and t_q. If p and q are not independent with each other, their prestars are partially or fully overlapped with each other. In an extreme condition when their prestars are fully overlapped, inserting either p or q is enough to eliminate both t_p and t_q. Therefore, one of p and q is redundant,

termed as *redundant point*. Furthermore, p and q may be too close to each other, such that a short edge pq is generated. The resulting short edge will engender thin triangles. Therefore more Steiner points would be needed in order to eliminate the new thin triangles. If pq is even shorter than the minimum local feature size of the input, the program may not terminate at all.

Naïve method 1 has redundant Steiner points problem as mentioned above, if we can ensure all Steiner points inserted in the same iteration are pairwise independent, then the program can guarantee to terminate. This method is illustrated in the following.

Naïve method 2: Let one thread handle one triangle. For any bad triangle t, compute its circumcenter c, and do point location for c to find its container triangle. Then insert all pairwise independent c into the mesh, and do edge-flip in parallel to maintain DT.

The main problem with this method is that it is hard and time consuming to make sure all Steiner points are independent with each other. In order to find the prestar for a Steiner point p, we need to find all triangles whose circumcircles include p. If one triangle belongs to more than one Steiner point's prestar, only one Steiner point can survive. The work for finding prestar for each Steiner point and avoiding race condition among all prestars by using atomic operation is huge and inefficient on the GPU.

Main idea of our solution: Instead of seeking maximal subset with pairwise independent points, we try to insert all Steiner points in parallel. If some Steiner points are redundant, we delete them later. Different from the sequential method, which inserts Steiner points one by one, and has no point deletion at all if not considering boundary refining (even if take the boundary refining into consideration, only few deletion operations are needed), our GPU algorithm needs to delete quite a number of Steiner points in parallel due to redundant points. Furthermore, since both point insertion and point deletion can be done by using edge-flip operation, we try to combine the operations for detecting and deleting redundant Steiner points with the operations for maintaining DT mesh after point insertion. As mentioned before, the order of circumcenter-insertion is very important. Priority queue can reduce the number of triangles by 35% than the number of triangles when using the first-come-first-split strategy. In order to get similar result as the result generated by the sequential implementation with a priority queue, when two Steiner points are not independent with each other, we always try to insert the one whose priority is higher, and delete the one whose priority is lower.

Our GPU algorithm has three steps shown in the following.

Step 1. Compute candidate Steiner points for bad triangles;

Step 2. Insert Steiner points and update neighboring information;

Step 3. Do edge-flip to maintain DT, while detect and delete redundant Steiner points;

In our algorithm, we need to do these three steps iteratively until no more bad triangles exist in the DT. Usually, the program needs to run dozens of iterations to generate the quality mesh for a given Delaunay mesh. In each iteration, we may need dozens of loops for Step 3 particularly. The details of this algorithm will be discussed in the following section.

One thing that needs to be noticed is that, since the off-center-insertion can reduce the number of Steiner points significantly, we always insert off-center points instead of circumcenters in our algorithm. When the circumcenters are mentioned, that always mean the off-centers actually. In the circumcenter-insertion with priority queue algorithm, the bad triangle with the smallest angle has highest priority to be split, while in the off-center-insertion, the bad triangle with the shortest edge has highest priority to be split. Therefore in our GPU-QM algorithm, when we say a triangle with higher priority means the triangle whose shortest edge is shorter than others'. In practice, many triangles may have the same priority, so we combine two information, the shortest edge and the index of the triangle, to be the priority instead.

3.3.2 Algorithm details

Step 1: Compute circumcenters for bad triangles

In this step, we first go through all triangles to check whether they are good or bad triangles. For each bad triangle, we compute its circumcenter, and do point location for its circumcenter.

There are several problems related to this step.

1. Some bad triangles' circumcenters may lie within a same triangle. In this situation, only one can be inserted, other circumcenters are ignored. Actually, under this situation, insert one circumcenter may eliminate more than one bad triangle. Since we always want to solve the worst triangle firstly, we give the worst triangle highest priority. As mentioned before, we combine two facts, the length of the shortest edge and index of the triangle, to be the priority of a bad triangle, in case more than one triangle has the same length of the shortest edge. In practice, the more cases like this there are, the better performance of the algorithm can achieve. That means instead of dealing with a large number of circumcenters, only working with a small number of circumcenters is more economical.

2. When doing point location for circumcenters, some circumcenters may lie outside of the region of the triangulation. For this situation, we just leave such bad triangles to the end in the basic algorithm discussed in this section. In order to solve those bad triangles, we need to split the boundary edges of the triangulation, which will be discussed in Section 3.4.

3. Circumcenters may lie on the edges of triangles. In order to handle such problem, we use the simulation of simplicity method [EM90] to deal with such degenerate cases.

4. Checking all triangles for all iterations is not necessary. In the first iteration, triangles' status are unknown, we need to check all of them in order to get all bad triangles. However, in the later iterations, especially in the last few iterations, only few triangles are changed, checking all triangles to see whether they are good or bad triangles is a waste. So we use one bit memory for each triangle to record whether it is changed or not during the current iteration. If it is not changed, then in the next iteration, there is no need to check it again.

Step 2: Insert circumcenters and update neighboring information

In this step all candidates of circumcenters selected from the Step 1 are inserted into the DT by using 1-3 flip, termed as *triangle-split* in our algorithm; see Figure 3.5. After doing point insertion, we also need to update neighboring information for all new triangles and their neighbors.

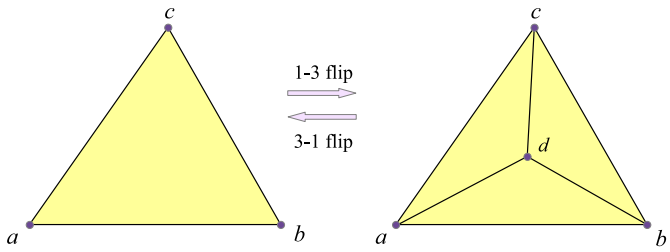


Figure 3.5: 1-3 flip (inserting d into the triangle abc) and 3-1 flip (removing d from the triangulation).

It is possible that some neighboring triangles should be split at the same time. In most of the time, two circumcenters inserted in adjacent triangles cannot be independent with each other. So we add one insertion-filter before we insert circumcenters in parallel. That is for each container triangle t, we check its three neighboring triangles. If any of its neighbors is also a container triangle with a higher priority, t cannot be split in this iteration. Although this filter idea is very simple and easy to implement, it can reduce a lot of redundant Steiner points which is illustrated in Section 3.7.

Step 3: Do edge-flip to maintain DT, while detect and delete redundant Steiner points

As we all know, edge-flip can be used to do 2-2 flip (see Figure 3.6) on non-locally Delaunay edges and change them to be locally Delaunay edges in the mesh. After all non-locally Delaunay edges become locally Delaunay edges, we obtain a DT mesh. If we only want to maintain the triangulation to be a DT after inserting Steiner points, edge-flip is enough. However, as mentioned before, the Steiner points inserted in Step 2 may not be pairwise independent. Some of these Steiner points can be redundant points, which may lead to significant increase of Steiner points in the output and infinite loop for the program. We want to delete redundant Steiner points and maintain the DT property for the triangulation at the same time by using edge-flip.

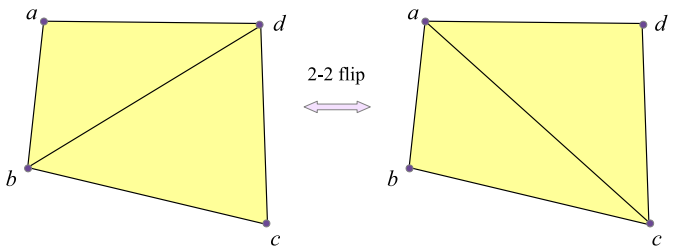


Figure 3.6: 2-2 flip.

In this step, there are two questions that need to be answered. The first question is how to combine the processing for point deletion and maintaining DT mesh. The second question is how to detect and delete redundant points. Let us solve these two questions one by one in the following.

Question 1: How to combine operations for point deletion and DT mesh maintaining?

We use 1-3 flip to insert a Steiner point into its container triangle, similarly, we can use 3-1 flip to remove a point from the triangulation (see Figure 3.5). A recent work called flip-flop [GCTH13] proved that: for a given non-extreme point v, if the degree of v is more than 3, there exists a 2-2 flippable edge incident to v; if the degree of v is 3, any edge incident to it is a 3-1 flippable edge. Figure 3.7 shows an example for deleting a non-extreme point v from the mesh. In this flip-flop algorithm, two major operations are flip and flop. Flip means normal 2-2 edge flip, termed as *Delaunay flip*, which converts a non-locally Delaunay flip into a locally Delaunay flip. Flop means 2-2 edge flip which reduces the degree of the point which should be deleted, and 3-1 flip which removes the point from the mesh. The flips for the flop operations are called *non-Delaunay flips*.

It seems that once we can detect the Steiner points which should be deleted, we can adapt the flip-flop method to use 2-2 and 3-1 flips to maintain the DT mesh and delete Steiner points.

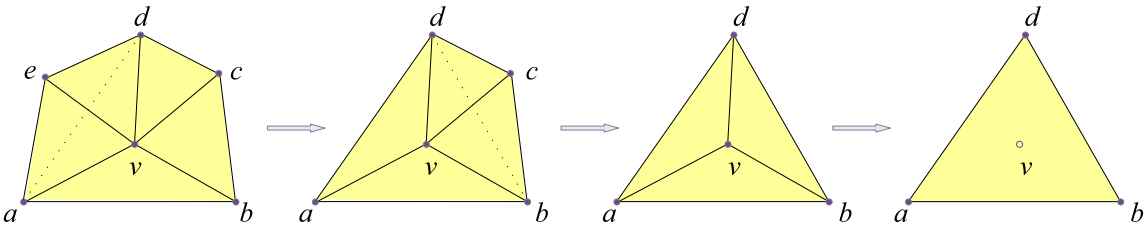


Figure 3.7: Given a non-extreme point v. To remove v, we flip ve and vc to reduce the degree of v to 3. Then, v is removed by a 3-1 flip.

Question 2: How to detect redundant Steiner points?

The points we want to delete are Steiner points which are not independent with other Steiner

points. Given two Steiner points, the Independent Lemma proved in [EG01] provides an easy way to decide whether they are independent with each other or not. Here we only list a part of it used in our algorithm as follows:

Lemma 3.3.1 (Independent Lemma). *If points p and q are independent with each other, the edge pq does not belong to the DT.*

The proof for this lemma is obvious. If p and q are independent implies the prestars of p and q are disjoint, which equals to the stars of them are disjoint. Hence edge pq cannot appear in the DT.

We do Delaunay flips for all non-locally Delaunay edges until we get a DT. In the DT, if any two Steiner points inserted in the current iteration are connected with each other, based on the Independent Lemma, one of them is redundant. Let us mark the one with lower priority as a redundant point. Then do non-Delaunay flips to remove all redundant Steiner points, and do Delaunay flips again to get the DT. Such a loop continues until we get the DT and no two Steiner points inserted in the current iteration connect with each other.

Clearly this solution is not good, because it may waste some flips. Considering a redundant Steiner point p, we first do flips to make the region around p (prestar) to be locally Delaunay, and then do flips to remove it. If we can detect p as a redundant point before making the prestar of it being locally Delaunay, and delete it from the mesh directly, we can save several flips. Motivated by this idea, we derive a new solution for this problem.

Our solution is to do flip-flop directly on the generated mesh of Step 2. Whenever two new Steiner points are going to be connected because of any kinds of flips (Delaunay flip or non-Delaunay flip), we compare the priority of these two points, the one with lower priority is marked as redundant point, and will be deleted in the next loops. Since we want to detect and delete redundant points as soon as possible, we give priority for non-Delaunay flips over Delaunay flips.

Not all triangles need to be checked for every loop. There is no need to check triangles which are not related to any kinds of flips in current loop in future. We use an array termed as *mark* to record such information. Initially, all new triangles generated due to point insertion in Step 2 are marked as should be checked by setting mark$[i] = 0$; other triangles are marked as mark$[i] = -1$, mean no need to be checked in the next loop. In addition we use an enumerated type pStatus $= \{\text{input}, \text{oldSteiner}, \text{newSteiner}, \text{redundant}\}$, to distinguish the points which are input points, Steiner points inserted in previous iterations, Steiner points inserted in the current iteration, and redundant Steiner points which should be deleted, respectively.

The algorithm presented in Algorithm 3.1 shows the first-part of Step 3, i.e., checking triangles to detect redundant points, non-Delaunay and Delaunay flips. One optimization for this part is that a triangle t_i only performs the checking with its neighbor t_j if $i < j$, since each pair of triangles should only be checked at most once in each loop.

Algorithm 3.1: Detect redundant point, non-Delaunay and Delaunay flips

```
1  while exists triangle t with mark[t] = 0 do
2      forall the t with mark[t] = 0 do
3          set minIndex = MAXINT
4          if any of t's vertices p is redundant then
5              set minIndex = the minimum index of p which is redundant
6          end
7          for three neighbors t_i of t do
8              if t's index is smaller than t_i' index, or mark[t_i] = −1 then
9                  if the vertex q which belongs to t_i and does not belong to t is
                   redundant then
10                     minIndex = min(minIndex, q)
11                 end
12                 if minIndex == vertex shared by t and t_i then
13                     mark a non-Delaunay flip for t and t_i
14                     break
15                 end
16                 if no vertex is redundant and t and t_i pass the in-circle test then
17                     if both vertices that are not shared by t and t_i are newSteiner
                       then
18                         mark the vertex with lower priority as redundant
19                         break
20                     else
21                         mark a Delaunay flip for t and t_i
22                         break
23                     end
24                 end
25             end
26         end
27     end
28 end
```

For each triangle t that needs to be checked in this loop, suppose one of its neighboring triangles is t_i. If any of the two vertices shared by t and t_i is a redundant point marked

in previous loops, there should be a non-Delaunay flip between t and t_i (if the point of minIndex's degree is 3, it is a 3-1 flip; if degree is more than 3, it is a 2-2 flip.). Otherwise we do in-circle test on t and t_i. If they pass the in-circle test and the two vertices that are not shared by t and t_i are newSteiner points, that means if we do Delaunay flip on t and t_i, the two newSteiner points will be connected. According to the Lemma 3.3.1, these two newSteiner points are not independent with each other, one of them should be marked as redundant and be deleted in future loops.

Algorithm 3.2: Mark incident triangles for redundant point, and do edge-flip for triangle pairs marked in Algorithm 3.1

```
1  forall the triangle pair marked in Algorithm 3.1 do
2  |  do edge flip
3  |  mark the new triangles t as mark[t] = 0
4  |  update neighboring information
5  end
6  forall the redundant point p do
7  |  for triangle t incident to p do
8  |  |  mark[t] = 0
9  |  end
10 end
```

The rest of the operations of Step 3 are listed in Algorithm 3.2. For a redundant Steiner points, we need to mark all its incident triangles as needed to be checked in the following loops, such that in the following loops we could find non-Delaunay flips for them to reduce the degree of them until they are deleted from the mesh. Then we do edge-flip to all triangle pairs marked in the Algorithm 3.1. One difficulty is how to update the neighboring information for triangles and their neighbors. Two adjacent triangles can participate in two different flips, thus directly updating the adjacent triangles after flipping can cause conflicting memory access. Instead, the updating operation is performed in two steps, and each is done in parallel, with a global synchronization in between. Each triangle has a temporary storage for updating its links. In the first step, each pair of triangles that is just flipped updates the temporary storage of its neighbors. In the second step, each pair of triangles mentioned above inspects its temporary storage and updates its own links. Note that if a neighbor of this pair is not flipped in this iteration, that neighbor is not processed by any thread. Thus, the thread processing this pair needs to update that neighbor's links directly.

Notice that, all new triangles due to the edge-flip should be marked as needed to be checked in the next loop. When performing the in-circle test, we use the exact predicate introduced

by Shewchuk which is adaptive and robust [She96b]. As for the problem of how to adapt this in-circle test in our algorithm on the GPU, we will discuss it in the Section 3.6.3.

3.3.3 Proof of termination

The input of the basic algorithm of GPU-QM is a stable subcomplex of a DT. We use $B = 1$ as the upper bound of circumradius-to-shortest edge ratio, so in the output DT, all angles are larger than or equal to 30°. The proof for the termination is quite obvious.

Claim 3.3.1. *With $B = 1$, the basic algorithm of the GPU-QM is guaranteed to terminate for the given stable subcomplex of a DT.*

Proof. If there is any bad triangle t in the triangulation, and t's circumcenter c lies in the interior of the triangulation, c will be inserted. The only new edges created by the Delaunay insertion of c are edges connected to c. Because c is the circumcenter, and there were no points inside the circumcircle of t before c was inserted, no new edge can be shorter than the circumradius of t due to the choice of B. Since all the Steiner points inserted in the same iteration are independent with other, the distance between any two new points is at least as large as the larger circumradius of two radii. Suppose the length of the shortest edge in the input mesh is ϵ, then no two points in the output mesh are less than 2ϵ apart, since all points lie inside or on the boundary of convex hull of the input points, after finite iterations, we will run out of space to put new point. So the algorithm is guaranteed to terminate as claimed. □

3.3.4 Remaining problems

In this section, the algorithm we discussed is a framework of the GPU-QM algorithm. This basic algorithm inserts circumcenters of bad triangles into the DT mesh to eliminate bad triangles. The algorithm is guaranteed to terminate although there are still some bad triangles surrounding the boundary edges of the triangulation. There are two major problems involved with the basic algorithm of GPU-QM.

1. The input is stable subcomplex of a DT. In practice, most of the DT is not a stable subcomplex, we need to handle boundary refining in order to make our algorithm handle real-world data.

2. How to improve our aggressive redundant Steiner points detection method in Step 3. In our algorithm we want to insert Steiner points that are pairwise independent in parallel. Although the current algorithm satisfies the independent requirement, we may delete some points which are not redundant points wrongly because of the property of flip-flop. In our algorithm, we try to do both non-Delaunay and Delaunay flips at the same time. Two new Steiner points can be connected potentially due to either Delaunay flip

or non-Delaunay flip. If due to a Delaunay flip, based on the Independent Lemma, the two Steiner points must not be independent with each other. Hence, one should be deleted as a redundant point. If the connection due to a non-Delaunay flip, this edge may or may not be the Delaunay edge in the final DT. However, we still mark one point as redundant and delete it in the following loops. This lead to a bad situation: some Steiner points may be inserted several times due to the aggressive deletion method used in Step 3. According to our experiments shown in Section 3.7, 70% points could be inserted more than once. This defect leads to big number of edge-flips. Suppose there are m points which are inserted more than one time, in order to insert them, $3m$ flips are needed, and in order to delete them, $4m$ flips are needed. If m is huge, the time spent on repeatedly inserting and deleting the same points is huge.

In the following sections (Section 3.4 and Section 3.5), we propose some mechanisms to solve these two problems.

3.4 Mechanisms for GPU-QM to Handle Boundary Refining

In the basic GPU-QM algorithm, we neglect the boundary refining problem by restricting the input mesh being a stable subcomplex of a DT, and reject a circumcenter if the circumcenter lies outside or on the boundary of the convex hull of the input DT mesh. The result is there are still a lot of bad triangles around the boundary edges of the convex hull. In this section, we propose several mechanisms for the basic GPU-QM algorithm, to make it can handle boundary refining such that no bad triangles will exist in the final quality mesh.

3.4.1 Motivation and algorithm

In order to handle boundary refining problem, the sequential method splits all encroached segments in the beginning (before doing triangle-split) until no segment is encroached. By doing this all circumcenters of any triangles lie within the region of the triangulation (Lemma 1 in [She01]). If any new point for triangle-split would encroach upon any segment, then it is not inserted; instead, all segments that are encroached upon are split.

In our GPU algorithm we do not eliminate all encroached boundary edges before inserting Steiner points. That is because usually the number of boundary edges is small compared to the number of bad triangles. In order to fully utilize the computation power of GPU, it is better to have tens of thousands of executing threads at any time. In addition, after some Steiner points are inserted, some boundary edges may be encroached upon again, therefore, we do not want to do segment-split, triangle-split, and then segment-split again as a loop. Since both segment-split and triangle-split can be seen as a kind of edge-flip, and the operations for them to update neighboring information are similar, we want to do

segment-split and triangle-split at the same time.

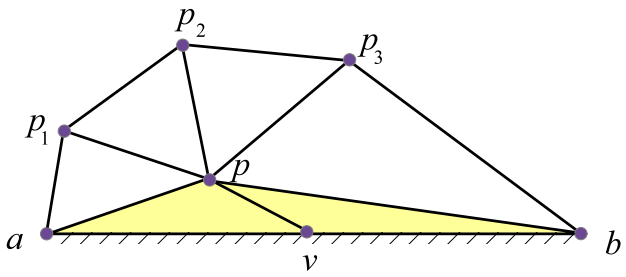


Figure 3.8: A case which generates a shorter edge, and leads to an infinite loop.

One big problem is how to detect all encroached boundaries for each circumcenter, i.e., when c is inserted, it may not aware it encroaches some boundaries that are far away. This unawareness may cause some problems. For example, suppose the midpoint v of an edge ab is inserted into the triangulation in Step 2, at the same time, some circumcenter p_i which encroaches upon ab has been already inserted in some previous iteration. Because of the insertion of v, the insertion of p_i may be redundant. Moreover, the insertion of p_i may generate edges shorter than the existing shortest edge of the mesh, so the algorithm may not terminate.

For example, in Figure 3.8, ab is a boundary edge, v is the midpoint of ab, p is the opposite point of ab which is a circumcenter inserted in some previous iteration. When p was inserted, p did not know the existence of ab. Observe that the triangle abp is a bad triangle, in order to eliminate it, we add v as a Steiner point and insert it in the current iteration. The distance between pv may be even smaller than the minimum shortest edge ϵ of the input DT before v is inserted. Recall that in Step 3, we only mark a Steiner point to be redundant when it is a new Steiner point inserted in the current iteration. So both p and v will remain in the mesh. This short edge pv may lead to further shorter edge to be generated and make the algorithm go into infinite loop.

Our solution is to increase a routine to detect redundant Steiner points before Step 3. For each midpoint inserted in the current iteration like v in Figure 3.8, check the point p which is opposite of ab, and all adjacent points p_i of p. If the checked points are not input points nor midpoints, and they encroach upon ab, mark them to be redundant Steiner points, and delete them in Step 3.

In all, in order to handle boundary refining, we need to do several changes to the basic algorithm of GPU-QM discussed in the previous section.

1. Add a new type of Steiner point - midpoint, and define the container triangle for a midpoint.

For a bad triangle t, and its circumcenter c, we search the container triangle for c by a

walk starting from t (point location) until finding its container or meet a boundary edge. If c lies on the boundary or outside of the triangulation, we treat the last triangle during the point location as the container triangle, and change the c to be the midpoint of the boundary edge.

2. Increase segment-split for Steiner point which is a midpoint.

   Since a Steiner point can be the midpoint of a boundary edge, we need to use 1-2 flip (Figure 3.9) to insert the midpoint. When a midpoint is inserted into the mesh, we split the boundary edge into two boundary edges, and split the triangle into two triangles, this process is termed as *segment-split*. Similarly, we add 2-1 flip in our program to delete a midpoint from the triangulation as well.

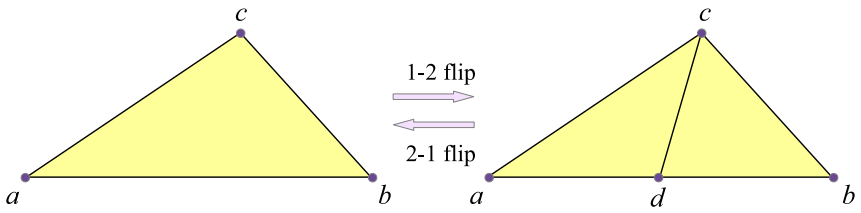


Figure 3.9: 1-2 flip (inserting midpoint d on the edge of ab) and 2-1 flip (removing the midpoint d from the triangulation).

   We give segment-split priority over triangle-split. Consider the following situation: a triangle t is incident to a boundary edge ab. A circumcenter c lies within t, so t can be split into three triangles. At the same time, another circumcenter of some bad triangle (can be t) is outside the boundary edge ab, so t can be split into two triangles as well. If we insist on inserting the circumcenter c, c may encroach the boundary edge and generate a new bad triangle. In order to eliminate the newly generated bad triangle, the midpoint is needed in later iterations. So it is better to insert the midpoint in this iteration, and c may not even be needed in later iterations.

   For any two Steiner points that are both midpoints of some boundaries, in the filter routine before doing point insertion in Step 2, if they are neighbors, we neglect one according to their priority; while in Step 3, even if they are connected by edge-flip, none of them should be marked as redundant. In other words, once a midpoint is inserted into the triangulation, do not delete it any more.

3. Increase a routine of detecting redundant Steiner points before Step 3.

   For each midpoint inserted in the current iteration like v in Figure 3.8, check the point p which is opposite to ab, and all points p_i which are adjacent to p. If the checked points are not input points or midpoints, and they encroach upon ab, mark them to be redundant Steiner points, and delete them in Step 3.

   One problem by doing this is that, this new routine may not discover all Steiner points

that encroach upon the boundary edge. If they are new Steiner points inserted in current iteration, we still have chance to eliminate them in Step 3. If they are old Steiner points inserted before, we cannot eliminate them. But this problem does not affect the termination of the algorithm. If p does not encroach upon ab, the algorithm behaves the same as the one without the new routine. If p encroaches upon ab, it can be discovered by our routine and marked as redundant point. Once p is marked as redundant and deleted in Step 3, the algorithm is guaranteed to terminate.

Actually, if p does not encroach the boundary edge, other points have no chance to encroach the boundary edge at all (see Figure 3.10). That is because before the insertion of v, the triangulation is a DT, since p does not encroach ab, p is outside of the diametral circle of ab. Furthermore, in a DT mesh, all points should be outside the circumcircle of triangle abp. Therefore all points should be also outside the diametral circle of ab, which means no point encroaches ab.

In addition, there are few Steiner points which are not incident to p but still encroach upon the boundary edge in practice. Comparing to the strategy in which all Steiner points that encroach boundary edges should be discovered and deleted, our strategy can save a lot of time and is easy to implement.

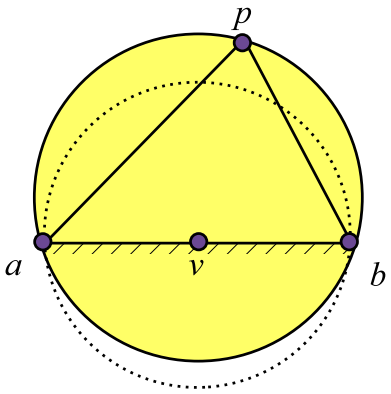


Figure 3.10: An example shows that if p does not encroach boundary edge ab, other points also do not encroach ab.

3.4.2 Proof of termination

Shewchuk claimed that given a DT for a set of points, if there is no encroached boundary edge, all triangles' circumcenters lie inside the triangulation. We present his claim as a lemma in the following, and the proof for it can be found in [She01].

Lemma 3.4.1. *Let T be a segment-bounded DT. (Hence, any edge of T that belongs to only one triangle is a segment). Suppose that T has no encroached segments. Let v be the circumcenter of some triangle t of T. Then v lies in T.*

Similar to Ruppert's and Chew's algorithms, our algorithm can be proved to be guaranteed to terminate if any two incident boundary edges on the convex hull of the input DT are separated by an angle of at least 60°, and a triangle is considered to be bad triangle if its circumradius-to-shortest edge ratio is larger than B, where $B \geq \sqrt{2}$.

In order to prove our algorithm can terminate, we need to prove two things. First, we need to prove the algorithm will terminate if it runs in a sequential manner, i.e., inserting only one Steiner point for each iteration. Second, we need to prove when the algorithm runs in parallel on the GPU, it can also terminate.

Notice that the proof presented below is very similar to the one discussed in [She01]. Here we adapt that proof to prove our algorithm can terminate.

Claim 3.4.1. *Suppose that any two incident boundaries are separated by an angle of at least 60°, and a triangle is considered to be a bad triangle if its circumradius-to-shortest edge ratio is larger than B, where $B \geq \sqrt{2}$. Let lfs_{min} be the shortest distance between two nonincident entities (vertices or segments) of the input DT mesh. If the GPU-QM algorithm runs in a sequential manner (only one Steiner point is inserted in each iteration), it will terminate, with no triangulation edge shorter than lfs_{min}.*

Proof. Assuming that the algorithm will introduce edges shorter than lfs_{min} into the mesh. Let e be the first such edge generated. Obviously, both of the endpoints of e cannot be input vertices, nor lie on nonincident boundary edges. Let v be the most recently inserted endpoint of e.

For each point p inserted into the mesh, the only new edges created because of p are the edges connect to p. Let r_p be the distance from p to its nearest point when p is inserted into the mesh. If p is an input point, r_p is the distance from p to the nearest other input point. By assumption, before v was inserted, no edge shorter than lfs_{min} existed, therefore before inserting v, for any point p in the mesh, $r_p \geq \text{lfs}_{min}$. Considering the following cases:

case 1. If v is the circumcenter of a bad triangle t, and p is the most recently inserted endpoint of the shortest edge of t, then $r_v \geq Br_p \geq \sqrt{2}r_p$.

case 2. If v is the midpoint of an encroached boundary edge l, and p is the opposite vertex of l before v is inserted in Step 2. Before Step 3, we need to check p's status. There are three subcases.

case 2-a. If p is an input point, then $r_v \geq r_p$.

case 2-b. If p is a circumcenter Steiner point and encroaches upon the edge l, p should be marked as a redundant point in Step 2 and be deleted in Step 3. Let p be the circumcenter of the bad triangle t. Since the mesh is DT when inserting p, the circumcircle centered at p contains neither endpoint of l. Hence, we get $r_v \geq r_p/\sqrt{2}$ (see Figure 3.11 for an example where the relation is equality). g is the most recently inserted endpoint of the shortest edge of t, then $r_p \geq Br_g$. So we get $r_v \geq r_p/\sqrt{2} \geq Br_g/\sqrt{2} \geq r_g$.

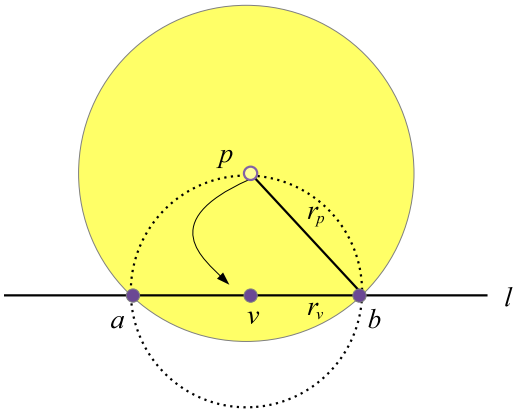


Figure 3.11: When l is encroached upon by p, a circumcenter of some bad triangle, r_v may be a factor of $\sqrt{2}$ smaller than r_p.

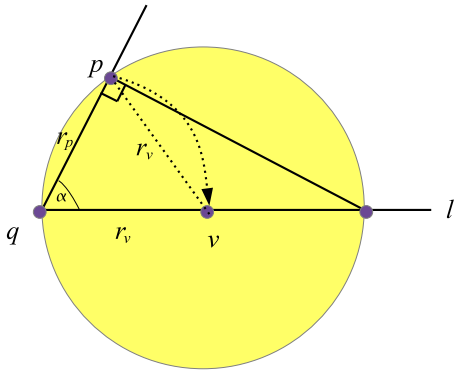


Figure 3.12: v and p lie on incident boundary edges separated by an angle α where $\alpha \geq 60°$.

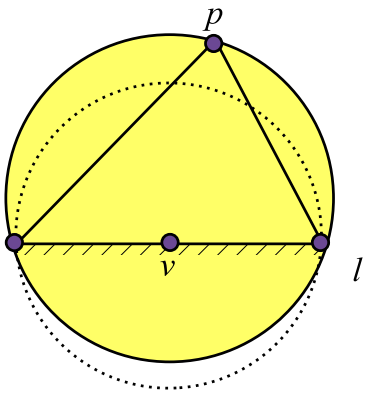


Figure 3.13: l is not encroached by p. Before the insertion of v, the triangulation is a DT mesh, all points are outside the yellow circle, and hence no point would encroach l.

case 2-c. If p is a midpoint, and p, v lie on incident boundary edges separated by an angle α and meet at point q (see Figure 3.12). If p encroaches upon l, p lies on or inside l's diametral circle. Imagining that r_p and α are fixed, then $r_v = |vp|$ is minimized by making l as short as possible, subject to p cannot fall outside l's diametral circle. r_v/r_p is smallest when $|l| = 2r_v$. Based on the basic trigonometry, $|l| \geq r_p/\cos\alpha$, then $r_v \geq r_p/(2\cos\alpha)$. Because $\alpha \geq 60°$, $r_v \geq r_p$.

Notice that the case in which p is a Steiner point, but does not encroach l, does not exist. Recall that v is inserted due to some circumcenter c lying outside the triangulation, i.e., c lies on the opposite side of l with the triangulation. According to Lemma 3.4.1, if c lies outside of the boundary edge l, l must be encroached by some point q. Assume q exists. Since p does not encroach l, p is outside the diametral circle of l. In a DT mesh, all points should be outside the circumcircle of the triangle incident to l. So all points should be also outside the diametral circle of l as shown in Figure 3.13, meaning no point encroaches upon l. This statement contradicts with the assumption that q exists. So if p is a Steiner point, it must be case2-b or case2-c mentioned above.

In all these cases, we have $r_v \geq r_a$ for some point a existing in the mesh before v is inserted. It follows that $r_v \geq \text{lfs}_{min}$, contradicting the assumption that e's length is less than lfs_{min}. It also follows that no edge shorter than lfs_{min} is ever introduced, so the algorithm must terminate. □

Claim 3.4.2. *Suppose that any two incident boundaries are separated by an angle of at least 60°, and a triangle is considered to be a bad triangle if its circumradius-to-shortest edge ratio is larger than B, where $B \geq \sqrt{2}$. Let lfs_{min} be the shortest distance between two nonincident entities (vertices or segments) of the input DT mesh. The GPU-QM algorithm will terminate, with no triangulation edge shorter than lfs_{min}.*

Proof. According to Claim 3.4.1, if the GPU-QM algorithm inserts Steiner point one by one, no edge shorter than lfs_{min} is ever introduced. Here, we want to prove when the algorithm inserts Steiner points in parallel, no edge shorter than lfs_{min} can also be generated.

Considering any two Steiner points inserted in the same iteration p and q.

If both p and q are circumcenters, or one is a circumcenter, the other is a midpoint, they must be independent with each other; otherwise one of them would be marked as a redundant point and be deleted. So edge pq does not exist in the output DT. In addition according to Claim 3.4.1 both p and q will not generate any edge shorter than lfs_{min}.

If both of these two points are midpoints and are connected being edge pq in the output, then this case is the case2-c shown in Claim 3.4.1 actually. Hence, edge pq is not shorter than lfs_{min}.

In all, since no edge shorter than lfs_{min} is ever introduced, the algorithm must terminate. □

3.5 GPU-QM-V: A Variant Algorithm of GPU-QM

In this section, a variant algorithm of GPU-QM termed as GPU-QM-V is proposed. In this algorithm, a new metric for the minimum separation bound between points is used. Experiment results in Section 3.7 show that the GPU-QM-V algorithm can reduce the running time dramatically, and can handle different point distributions very well.

3.5.1 Motivation and algorithm

Now let us review the second question proposed in Section 3.2. According to the Lemma 3.3.1, whenever two Steiner points are connected in DT, they are not independent with each other. However, since we do both Delaunay flips and non-Delaunay flips at the same time in Step 3 of the GPU-QM algorithm, it is very difficult to distinguish whether the two Steiner points are connected because of Delaunay flip or non-Delaunay flip. Hence, the GPU-QM algorithm may delete many Steiner points that are not redundant points at all. Most of non-redundant Steiner points deleted in previous iterations may be inserted back into the triangulation in future iterations. According to our experiment results, more than 70% Steiner points are inserted into the mesh more than one time in the GPU-QM algorithm. In other words, the method for detecting redundant points employed by the GPU-QM algorithm is too aggressive, and wastes a lot of flips and time.

Our solution is to use a new metric for the minimum separation bound between points. When two new Steiner points inserted in parallel in the current iteration which are orthogonal or further than orthogonal, are going to be connected in Step 3, one should be marked as redundant.

The concept of orthogonality is firstly defined in [EG01] and used in their off-line implementation to control the minimum separation bound between sinks. In our research, we first observed and proved the relationship between independent and orthogonality. Then based on the observation, we use orthogonality to insert Steiner points in parallel in the GPU-QM-V algorithm.

Here we introduce the orthogonal concept firstly. For each Steiner point, there is a corresponding *insertion radius* related to it. If the Steiner point is a circumcenter of a bad triangle, then the insertion radius is the circumradius when the Steiner point is inserted; while if the Steiner point is a midpoint of a boundary, then its insertion radius is the radius of the diametral circle of the boundary. We use (x, q) to denote the Steiner point x, and its insertion radius q. Given two Steiner points x and y with their corresponding insertion radius q and r (denoted as (x, q) and (y, r)), x and y are *orthogonal* if $\|x-y\|^2 = q^2+r^2$, are *further than orthogonal* if $\|x-y\|^2 > q^2+r^2$, and are *less than orthogonal* if $\|x-y\|^2 < q^2+r^2$. If two points are orthogonal or further than orthogonal, then the distance between them is at least as large as the larger radius of two radii; see Figure 3.14.

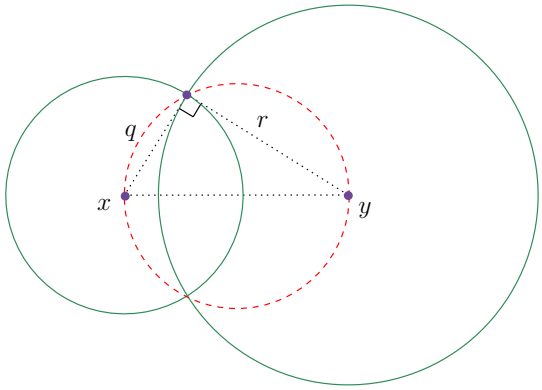


Figure 3.14: (x, q) and (y, r) are orthogonal. The red circle is the diametral circle of xy.

Now let us come back to the GPU-QM-V algorithm. The GPU-QM-V algorithm is almost as the same as the GPU-QM algorithm, except that in Step 3, orthogonality is used as the metric to decide whether a Steiner point is redundant or not. If two new Steiner points inserted in the current iteration are orthogonal or further than orthogonal, keep both of them. Otherwise we mark the one with lower priority as a redundant point and delete it by using edge-flip. The routine for computing the distance between two new Steiner points and judging whether they are orthogonal, further than orthogonal, or less than orthogonal is termed as *orthogonal-test* in the GPU-QM-V algorithm.

Generally speaking, if two points are independent with other, they must also be orthogonal or further orthogonal, but not vice versa. However, we can always prove the following claim.

Claim 3.5.1. *For any two Steiner points x and y inserted in a same iteration with insertion radius q and r, respectively, if x and y are less than orthogonal, they must be dependent with each other.*

Proof. For the given two Steiner points (x, q) and (y, r) which are less than orthogonal, let us assume they are independent with each other, therefore there is no edge xy exists in the DT after inserting both x and y into the triangulation. There are two cases as follows:

1. The stars of x and y have no common points or share only one common point, which is equivalent to saying the insertion circle of the two points are disjoint or intersect with each other on one point. Hence, $\|x - y\|^2 \geq (q + r)^2 > q^2 + r^2$, and x and y are further than orthogonal. This contradicts with the assumption that x and y are less than orthogonal.

2. The stars of x and y have a common edge ab; see Figure 3.15. For example, the insertion circles of x and y intersect on points a' and b'. Because x and y are less than orthogonal, $\angle xa'y \leq 90°$, and $\angle xb'y \leq 90°$. Because a and b are input points or Steiner points inserted in some previous iterations, a and b must lie on or outside of the insertion circle of x and y. So $\angle xay \leq \angle xa'y$, $\angle xby \leq \angle xb'y$, $\angle xay + \angle xby \leq \angle xa'y + \angle xb'y \leq 180°$, and we get

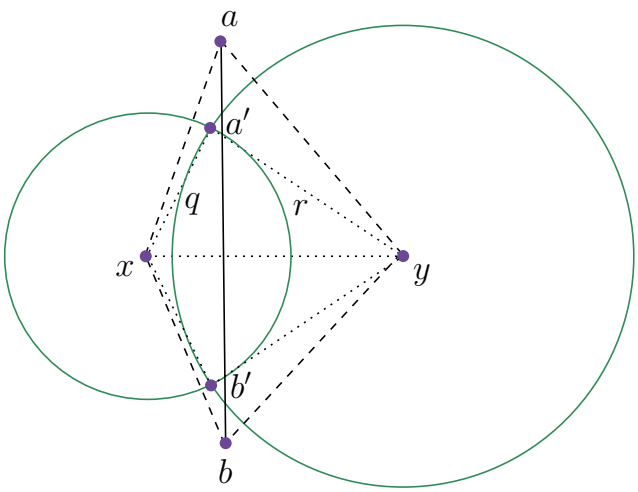


Figure 3.15: (x, q) and (y, r) are less than orthogonal and their insertion circles intersect on points a' and b'. ab is a common edge shared by the star of x and y.

$\angle axb + \angle ayb \geq 180°$. Observe that edge ab is not locally Delaunay at all, which contradicts with the assumption that the mesh is a DT.

These two cases above concludes our proof of Claim 3.5.1. □

In the GPU-QM-V algorithm, if two new Steiner points inserted in the current iteration are going to be connected, we check whether they are less than orthogonal or not. If yes, according to Claim 3.5.1, these two points must be dependent with each other, and the one with lower priority is marked as redundant.

Comparing to the aggressive redundant points detection method used in the GPU-QM algorithm, the method proposed here is less aggressive. When two Steiner points inserted in the same iteration will be connected with each other, we mark the one with lower priority as redundant only when the two points are less than orthogonal, i.e., delete one if two points are surely not dependent with each other. One problem with this new method is we may neglect deleting some redundant Steiner points. Those Steiner points are not independent with other points, but they can pass the orthogonal-test. As a result, as shown in the Section 3.7, the number of Steiner points inserted by the GPU-QM-V algorithm increases slightly compared to the GPU-QM algorithm. But the running time for this variant algorithm decreases significantly and this variant algorithm can handle some special point distributions better than the GPU-QM algorithm.

3.5.2 Proof of termination

The proof of termination for the GPU-QM-V algorithm is very similar to the one for the GPU-QM algorithm. The only difference is that for any two new Steiner points p and q

inserted in the same iteration in parallel, they may not be independent with each other. But the orthogonal-test used in the GPU-QM-V algorithm can guarantee these two points must be orthogonal or further than orthogonal. So the distance between them is at least as large as the larger insertion radius of two radii. Hence, no edge shorter than lfz_{min} is ever introduced, the algorithm is guaranteed to terminate if the input is as the same as the one in the GPU-QM algorithm.

3.6 Implementation Details

In this section, we highlight some implementation techniques used in both GPU-QM and GPU-QM-V algorithms.

3.6.1 Dealing with variable-size arrays

Linked data structures which need dynamic memory allocation is complicated and inefficient to use with our massively parallel algorithm. Instead, we maintain all required information as arrays. For example, the input points are stored as an array of points, and each point is represented by two floating point coordinates. One annoying problem with our algorithms is that we keep adding and deleting points, such that both the sizes of point array and triangle array are changed frequently. Therefore, a solution to handling variable-size arrays is very important. In the following section, we use triangle array as an example.

One simple solution is always adding new triangles at the end of the current array, and mark the slots of deleted triangles as empty (see Figure 3.16). In our algorithms, especially the GPU-QM algorithm, many points would be deleted and inserted again, so the size of the point array and triangle array may increase quickly. Usually, in our algorithms one thread handles one triangle. Large triangle array means large number of threads should be launched. If there are lots of empty slots among the triangle array, many launched threads have no work to do and are wasted. In addition, if we do not re-use the empty slots, there is no way to handle more than 1 million points. After re-using the empty slots, we can run up to 3 million points under our experimental environment mentioned in Section 2.4.

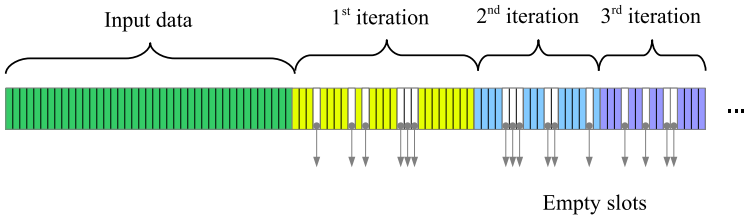


Figure 3.16: Expanding the point list for all iterations.

Another solution is to compact the triangle array immediately after every iteration. After

every iteration, we mark the empty slots in the triangle array, and use parallel stream compaction to compact the array. Notice that we also need to update triangles' neighboring information since the positions of their neighbors are also changed. Usually for a given input data, there are tens of iterations in order to compute its quality mesh on the GPU. It is very costly to compact for every iteration, especially for the iterations near to the end where only few triangles are changed.

In all, we want to re-use the empty slots, but do not want to do parallel stream compaction after every iteration. Our solution here is to maintain an empty list for empty slots in the triangle array (see Figure 3.17). This list records all empty slots' positions in the triangle array which can be re-used, at the same time we record the size of the empty list n_{size}. When new triangles are generated, we compare the number of new triangles with n_{size} to check whether we have enough empty space in the triangle array for the new triangles. If there is enough space, we get the empty slots' positions from the empty list, and put new triangles into the triangle array using the positions obtained. If there is not enough space, we expand the triangle array, and the empty list is expanded at the same time for the new empty slots in the triangle array. After every iteration, instead of doing parallel stream compaction to compact the triangle array as mentioned above, we only collect empty slots from the triangle array. At the end of the algorithms, before we transfer the data back to CPU, we do parallel stream compactions for the triangle array to get rid of the empty slots that still retained.

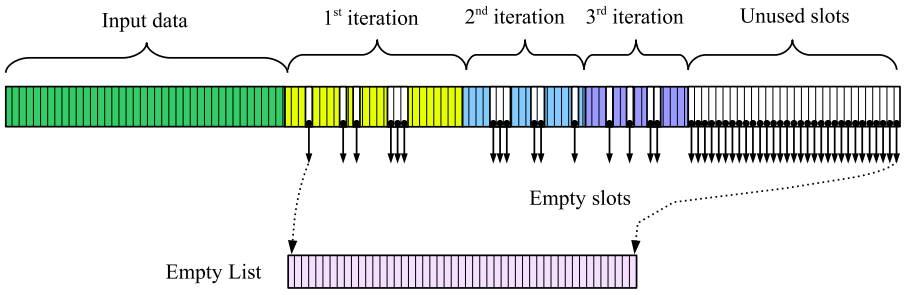


Figure 3.17: Our solution is to re-use empty slots in the point/triangle list.

Furthermore in order to save more time, we prefer not to collect empty slots from the empty list after every iteration. Instead we compare the number of new triangles with the number of available slots in the empty slots. If there is still enough space in the empty list, we use such slots immediately without any collecting operation. Only if there is not enough space in the empty list, we collect empty slots from the triangle array, and consider expanding the triangle list when there is not enough space after the collecting operation.

Usually in the beginning of the algorithms, we have no idea of the final size of the mesh for various of input data, so it is impossible to pre-allocate enough space for points and triangles explicitly in the algorithms. Suppose the initial size of the mesh is n (number of

points), and the number of triangles is $2n$. In our experiments, we prefer to pre-allocate $2n$ memory space for point array, and $4n$ memory space for triangle array. Whenever there is not enough space for new elements, we simply expand the point and triangle arrays by a size of n and $2n$, respectively.

3.6.2 Active threads: compaction or collection

In our algorithms, usually one thread handles one triangle, so we need to launch n (the number of triangles) threads in parallel. However, sometimes, only a few number of threads have work to do, others are idle thus reducing the efficiency of the program due to thread divergence. For example, in Step 3, for a given triangle t, if its mark is zero (i.e., mark$[t] = 0$), it should be checked in the current loop. Only in the first few loops, many triangles need to be checked, while in the loops near the end of Step 3, the number of triangles need to be checked is very small. Let the number of triangles be n, when we launch n threads for the triangle array, only the threads corresponding to the triangles whose mark is zero are active, and are termed as *active threads*. The number of active threads is denoted by m in the following.

Usually, only in the first several loops m is big, while in the loops near the end of Step 3, m is much smaller than n. Instead of launching n threads, we want to launch only active threads. There are two solutions to achieve such a goal.

1. Compact the triangle array after each loop.

   Mark triangles that need to be checked in the next loop, and do stream compaction in parallel at the end of the current loop. But it is very costly especially when m is very small compared to n.

2. Collect active threads that need to be checked in the next loop within current loop.

   Assume that only Delaunay flips are performed in Step 3. Since each flip modifies 2 triangles, we can pre-allocate an array called *activeArray*, with a size of 2 times the number of active threads in current iteration. Each flip records the triangles it modified in the activeArray. At the end of the current loop, we compact the activeArray and use it in the next loop. For this solution, one problem is that it is very hard to predict the correct size of activeArray for the next loop sometimes. For example, in Step 3, we need to detect and delete redundant points as well as maintain Delaunay property for all triangles. For all the redundant points, we need to mark their incident triangles as need to be checked in the next loop. But it is impossible to predict the number of redundant points, and the number of incident triangles for redundant points.

In our implementation, we combine these two solutions. We use an integer *collectSize* as the size of the activeArray and set collectSize $= n \times 0.05$. In the first several loops when

m is bigger than collectSize, we use the first solution, i.e., compact the triangle array to get all triangles which should be checked in the next loop. For the loops near the end of Step 3, when m is smaller than collectSize, the program will employ the second solution automatically. Actually once the program goes into the second solution, the program rarely goes back to the first solution. According to our experience, less than 10% loops go to the first solution, most loops would adopt the second solution automatically.

We also use this technique in other places in our algorithms. For example, in Step 2, we use this technique to mark all new triangles which should to be checked in Step 3. Furthermore, we can use the activeArray obtained from Step 2 as the initial activeArray for Step 3. So in the last several iterations, when only few Steiner points are inserted, we can only launch few threads instead of n threads even in the first several loops of Step 3.

3.6.3 Exact arithmetic and robustness

Our algorithms rely on two *predicates*, the orientation predicate and the in-circle predicate. To deal with numerical error, we adapt and implement the exact predicates on the GPU, based on Shewchuk's sequential implementation [She96b]. Furthermore, we use the simulation of simplicity method [EM90] to deal with degenerate data.

Each predicate consists of two parts: a fast check which uses floating point arithmetic, and an exact check which uses floating point expansion. However, in most cases, the fast check is enough, only few threads go into the exact check. Threads which go into the exact check need more temporary memory and registers, hence the number of threads can be launched at the same time is much smaller compared to the situation in which all threads go into the fast check. In order to fully utilize the computation power of the GPU, it is better to do regularized work for all threads. If some threads go into the fast check, and some threads go into the exact check, the fast checks will be slowed down by the exact checks.

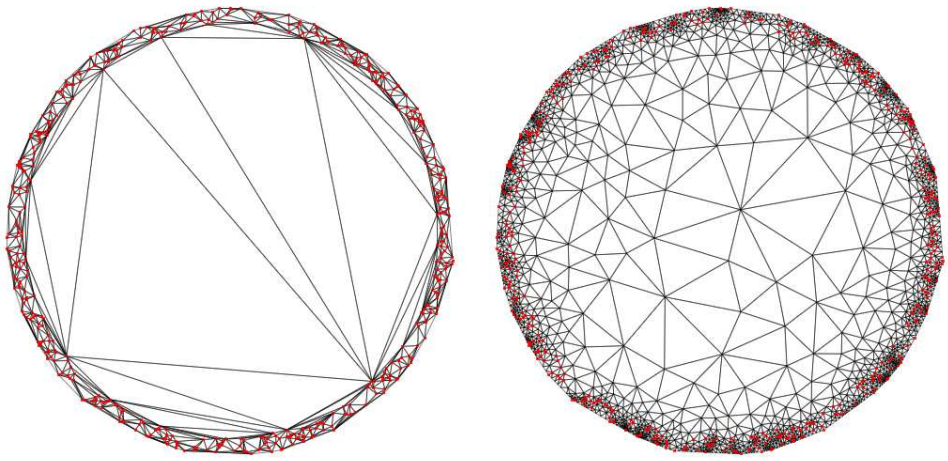
Our solution to handle this unbalanced workload among threads is to split the check kernel into two kernels. One kernel only do fast check for all threads, and the threads that need exact checks are marked at the same time. The second kernel would be launched after the first kernel immediately, and only the marked threads would do the exact checks. According to our experiences and experiments, such a GPU technique can speed up the computing time for predicates by 2 to 3 times. Because the running time for predicates only occupies less than one thousandth of the total running time, there is no explicitly time improvement after using this GPU technique. However, this GPU technique is very useful for other computational geometry problems. For example, the 3D in-circle predicate is very complicated, only several threads can be launched at the same time. It is a big waste to do exact check for every in-circle test actually, most tetrahedral only need fast check. At this situation it is better to use the technique mentioned above.

3.7 Experiment Results

To assess the efficiency of our program, we compare with *Triangle* on the running time, and the quality of the mesh on both synthetic and real-world data. The input to the program is a DT mesh of a set of points. All numbers and computations are done in double precision. The running time includes the time for computation and the time for transferring data from CPU to GPU, and vice versa.

3.7.1 Synthetic dataset

We create synthetic data by generating points randomly in the uniform, Gaussian, and three other distributions which are called disk, circle and grid distribution, respectively. In the disk distribution, all points lie in a disk, which means all points on the boundary of the disk are cocircular. In the circle distribution, points lie in between the boundary of two circles of slightly different radius (see Figure 3.18). In the grid distribution, points are on a grid of size 512×512, which is a degenerate case.



(a) (b)

Figure 3.18: Circle distribution. (a) Input DT mesh. (b) Output quality mesh.

Running time comparison

Figure 3.19 shows the running time and speedup of GPU-QM over *Triangle* on different point distributions, with the input size ranging from 0.5 to 3 millions. Notice that the running time increases linearly with the input size, and the performance for the disk and grid distributions are very similar to the performance for the uniform and Gaussian distributions. This is the result of our careful handling of exact arithmetic on the GPU. Compared to *Triangle*, the speedup is 2 to 3 times on average for all point distributions except the circle distribution. For the circle distribution, usually the speedup is below 2 times. That is because in this

distribution, there is 10 times more insertion iterations compared to other distributions.

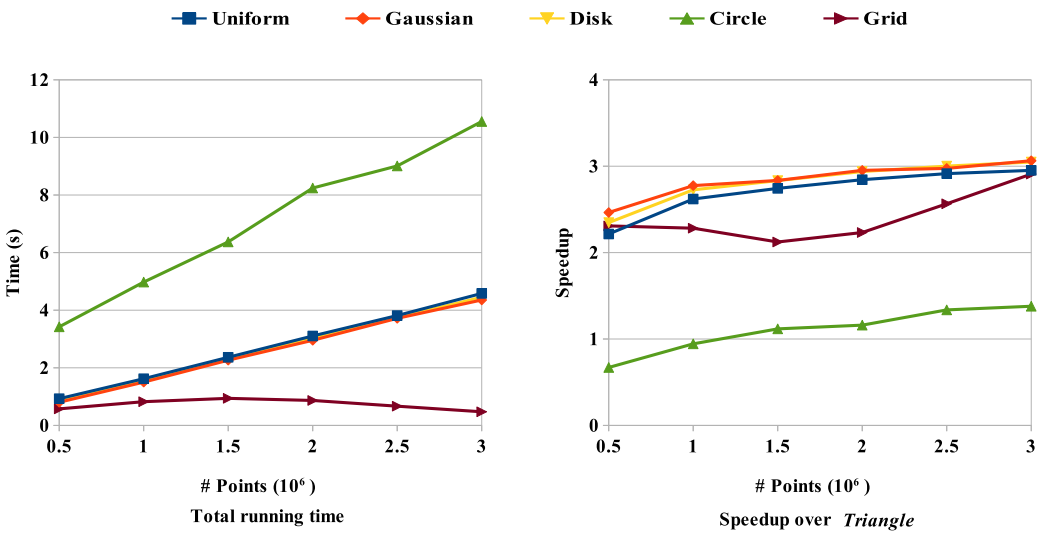


Figure 3.19: Total running time and speedup over *triangle* on different distributions by using GPU-QM algorithm on the GPU.

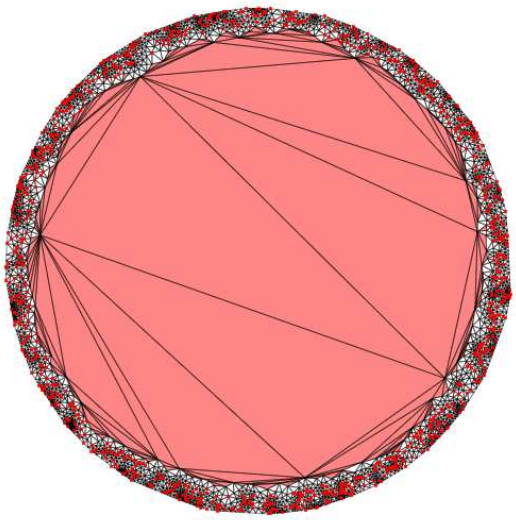


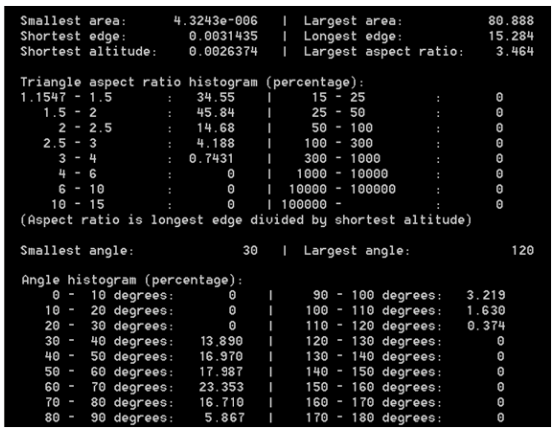
Figure 3.20: Circle distribution: one intermediate result after several iterations. Bad triangles are marked with red color.

Figure 3.20 shows one intermediate result for the circle distribution. Since we always use priority to solve race condition among triangles, a bad triangle whose shortest edge is relatively shorter than neighboring triangles' has priority to be eliminated. The algorithm intends to solve bad triangles lie in a region where the local feature sizes are relative small with priority. As a result several bad triangles whose shortest edges are relatively big in the middle have little chance to be eliminated in the first several iterations. When most of bad triangles near to the boundaries are eliminated, such big bad triangles are still there.

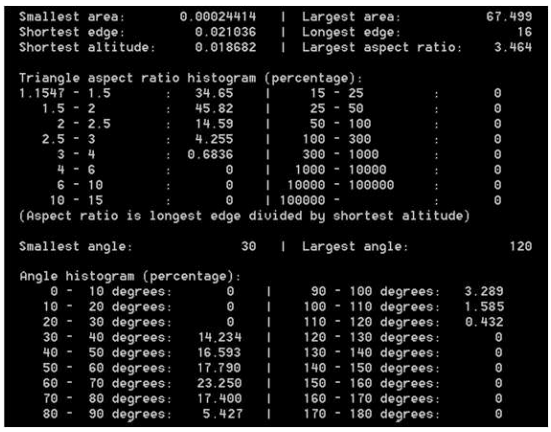
In order to eliminate the bad triangles left in the middle region of the mesh, we continue to insert circumcenters for them. However, these bad triangles are so big such that their circumcircles are very big and may include more than one circumcenter candidate. As a result, only few candidates (sometimes only one) can be inserted in each iteration, other candidates are marked as redundant and be deleted. Therefore there are more iterations for this distribution compared to other point distributions. As shown later, the GPU-QM-V algorithm can handle this problem efficiently.

Mesh quality comparison

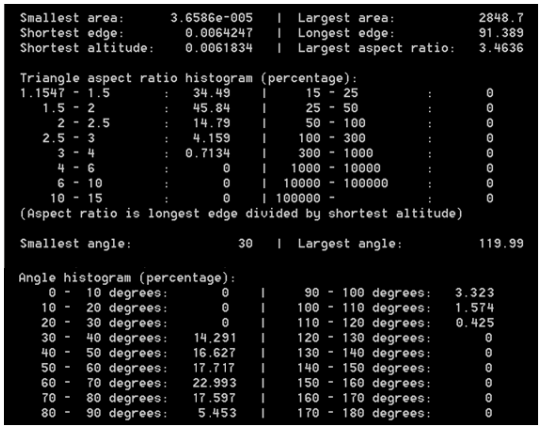
In order to compare the quality of the meshes generated by our algorithm and *Triangle*, we collect several statistics on many measurements, such as the angle histogram, the aspect ratio histogram, the smallest/largest area of triangles, the shortest/longest edge of triangles, and so on. The statistics (see Figure 3.21) show that the meshes generated by our program have almost the same quality as the meshes generated by *Triangle* on the measurements mentioned above. Therefore, we only focus on the number of Steiner points inserted in the program when discussing the quality of the mesh from now on.



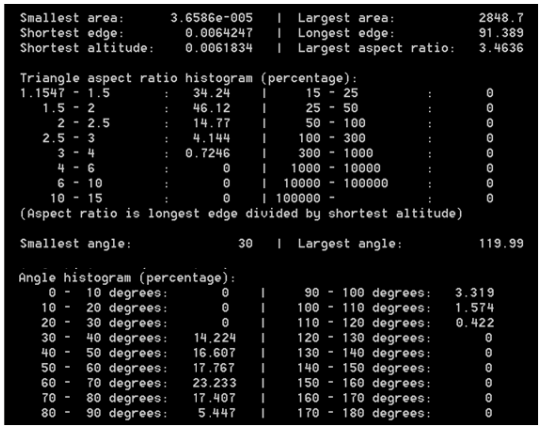
(a)



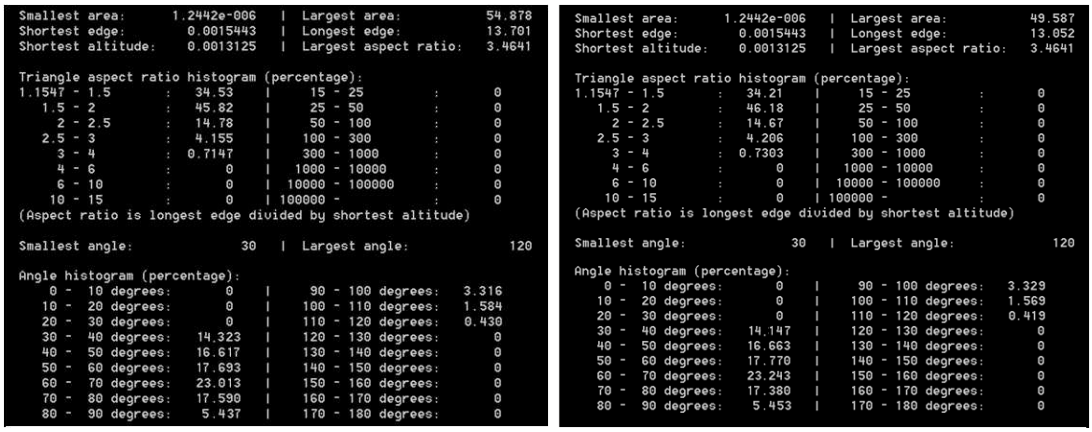
(b)



(c)



(d)



Smallest area: 1.2442e-006 | Largest area: 54.878
Shortest edge: 0.0015443 | Longest edge: 13.701
Shortest altitude: 0.0013125 | Largest aspect ratio: 3.4641

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.53	15 - 25	0
1.5 - 2	45.82	25 - 50	0
2 - 2.5	14.78	50 - 100	0
2.5 - 3	4.155	100 - 300	0
3 - 4	0.7147	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.316
10 - 20 degrees:	0	100 - 110 degrees:	1.584
20 - 30 degrees:	0	110 - 120 degrees:	0.430
30 - 40 degrees:	14.323	120 - 130 degrees:	0
40 - 50 degrees:	16.617	130 - 140 degrees:	0
50 - 60 degrees:	17.693	140 - 150 degrees:	0
60 - 70 degrees:	23.013	150 - 160 degrees:	0
70 - 80 degrees:	17.590	160 - 170 degrees:	0
80 - 90 degrees:	5.437	170 - 180 degrees:	0

(e)

Smallest area: 1.2442e-006 | Largest area: 49.587
Shortest edge: 0.0015443 | Longest edge: 13.052
Shortest altitude: 0.0013125 | Largest aspect ratio: 3.4641

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.21	15 - 25	0
1.5 - 2	46.18	25 - 50	0
2 - 2.5	14.67	50 - 100	0
2.5 - 3	4.206	100 - 300	0
3 - 4	0.7303	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.329
10 - 20 degrees:	0	100 - 110 degrees:	1.569
20 - 30 degrees:	0	110 - 120 degrees:	0.419
30 - 40 degrees:	14.147	120 - 130 degrees:	0
40 - 50 degrees:	16.663	130 - 140 degrees:	0
50 - 60 degrees:	17.770	140 - 150 degrees:	0
60 - 70 degrees:	23.243	150 - 160 degrees:	0
70 - 80 degrees:	17.380	160 - 170 degrees:	0
80 - 90 degrees:	5.453	170 - 180 degrees:	0

(f)

Smallest area: 2.589e-006 | Largest area: 49453
Shortest edge: 0.002038 | Longest edge: 409.5
Shortest altitude: 0.0015447 | Largest aspect ratio: 3.4639

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.68	15 - 25	0
1.5 - 2	45.63	25 - 50	0
2 - 2.5	14.73	50 - 100	0
2.5 - 3	4.221	100 - 300	0
3 - 4	0.7388	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.298
10 - 20 degrees:	0	100 - 110 degrees:	1.592
20 - 30 degrees:	0	110 - 120 degrees:	0.440
30 - 40 degrees:	14.393	120 - 130 degrees:	0
40 - 50 degrees:	16.517	130 - 140 degrees:	0
50 - 60 degrees:	17.633	140 - 150 degrees:	0
60 - 70 degrees:	23.027	150 - 160 degrees:	0
70 - 80 degrees:	17.677	160 - 170 degrees:	0
80 - 90 degrees:	5.393	170 - 180 degrees:	0

(g)

Smallest area: 2.589e-006 | Largest area: 45953
Shortest edge: 0.002038 | Longest edge: 383.17
Shortest altitude: 0.0015447 | Largest aspect ratio: 3.4639

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.3	15 - 25	0
1.5 - 2	46.02	25 - 50	0
2 - 2.5	14.68	50 - 100	0
2.5 - 3	4.236	100 - 300	0
3 - 4	0.7585	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.309
10 - 20 degrees:	0	100 - 110 degrees:	1.581
20 - 30 degrees:	0	110 - 120 degrees:	0.430
30 - 40 degrees:	14.243	120 - 130 degrees:	0
40 - 50 degrees:	16.587	130 - 140 degrees:	0
50 - 60 degrees:	17.717	140 - 150 degrees:	0
60 - 70 degrees:	23.360	150 - 160 degrees:	0
70 - 80 degrees:	17.340	160 - 170 degrees:	0
80 - 90 degrees:	5.433	170 - 180 degrees:	0

(h)

Smallest area: 0.00024414 | Largest area: 67.499
Shortest edge: 0.021036 | Longest edge: 16
Shortest altitude: 0.018682 | Largest aspect ratio: 3.464

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.65	15 - 25	0
1.5 - 2	45.82	25 - 50	0
2 - 2.5	14.59	50 - 100	0
2.5 - 3	4.255	100 - 300	0
3 - 4	0.6836	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.321
10 - 20 degrees:	0	100 - 110 degrees:	1.642
20 - 30 degrees:	0	110 - 120 degrees:	0.383
30 - 40 degrees:	13.935	120 - 130 degrees:	0
40 - 50 degrees:	16.653	130 - 140 degrees:	0
50 - 60 degrees:	17.890	140 - 150 degrees:	0
60 - 70 degrees:	23.153	150 - 160 degrees:	0
70 - 80 degrees:	16.860	160 - 170 degrees:	0
80 - 90 degrees:	5.863	170 - 180 degrees:	0

(i)

Smallest area: 0.00024414 | Largest area: 83.505
Shortest edge: 0.021036 | Longest edge: 16.08
Shortest altitude: 0.018682 | Largest aspect ratio: 3.464

Triangle aspect ratio histogram (percentage):

1.1547 - 1.5	34.33	15 - 25	0
1.5 - 2	46.09	25 - 50	0
2 - 2.5	14.58	50 - 100	0
2.5 - 3	4.301	100 - 300	0
3 - 4	0.7063	300 - 1000	0
4 - 6	0	1000 - 10000	0
6 - 10	0	10000 - 100000	0
10 - 15	0	100000 -	0

(Aspect ratio is longest edge divided by shortest altitude)

Smallest angle: 30 | Largest angle: 120

Angle histogram (percentage):

0 - 10 degrees:	0	90 - 100 degrees:	3.219
10 - 20 degrees:	0	100 - 110 degrees:	1.630
20 - 30 degrees:	0	110 - 120 degrees:	0.374
30 - 40 degrees:	13.890	120 - 130 degrees:	0
40 - 50 degrees:	16.970	130 - 140 degrees:	0
50 - 60 degrees:	17.987	140 - 150 degrees:	0
60 - 70 degrees:	23.353	150 - 160 degrees:	0
70 - 80 degrees:	16.710	160 - 170 degrees:	0
80 - 90 degrees:	5.867	170 - 180 degrees:	0

(j)

Figure 3.21: Mesh quality comparison between *Triangle* (left columns) and GPU-QM (right colums) on different distributions. (a)(b) Uniform distribution. (c)(d) Gaussian distribution. (e)(f) Disk distribution. (g)(h) Circle distribution. (i)(j) Grid distribution.

For the GPU-QM algorithm, experiments show that we only insert less than 1% more Steiner points compared to *Triangle*. Recall that for the sequential algorithm, the implementation with priority queue can reduce 35% Steiner points compared to the implementation without priority queue, which means the order of Steiner points to be inserted is very important. No more than 1% Steiner points compared to *Triangle* with priority queue, is the result of our careful handling of priority among triangles in parallel. In our algorithm we always try to insert the circumcenters for bad triangles whose shortest edges are relatively small in their neighboring space. If we do not set priority for triangles, and randomly choose one triangle between two triangles when they have race relationship, we gain the similar result as the result generated by *Triangle* without priority queue. Figure 3.22 shows the quality comparison among the results of *Triangle* with priority queue, the results of GPU-QM with/without setting priority for triangles in parallel on uniform and grid distributions. Other distributions such as Gaussian, disk and circle distributions have similar behavior as the uniform distribution.

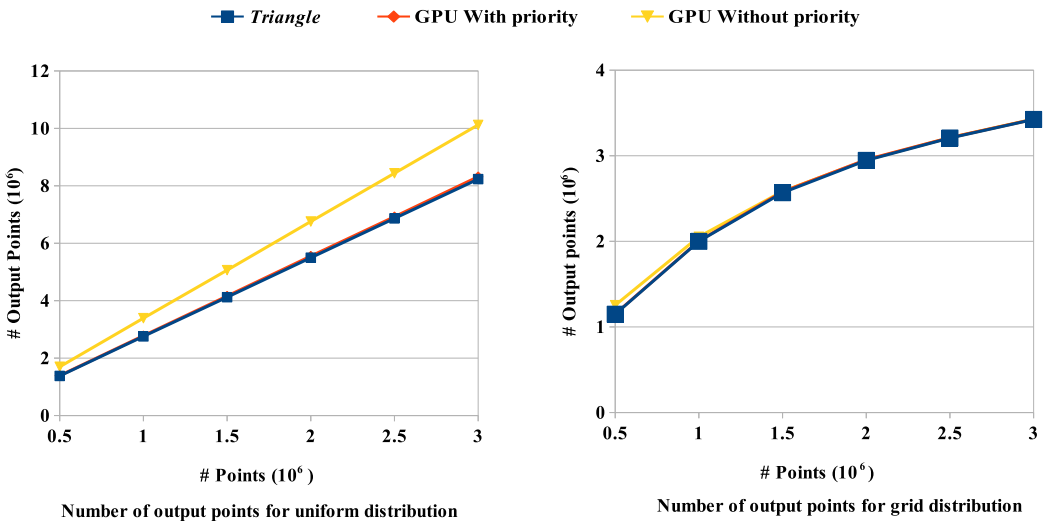


Figure 3.22: Comparison on the number of output points among *Triangle* with priority queue, GPU-QM with priority, and GPU-QM without priority on uniform (left) and grid (right) distributions.

Except for the grid distribution, the other 4 distributions insert similar number of Steiner points for the input DT mesh. For example, for 1 million input points, usually the GPU-QM algorithm would insert 1.7 million Steiner points in order to eliminate all bad triangles. But under the grid distribution, 1 million input points only need 1 million Steiner points, and with the increase of the input size, the ratio of the number of Steiner points to the number of input points has obvious decreasing trend. That is because, in the grid distribution, all points are distributed on a grid of size 512×512. If the grid is full of input points, there are no bad triangles at all. So when the number of input points increases, the grid is occupied more by the input points, the percentage of bad triangles is less, therefore fewer Steiner

points are needed.

Running time of different steps

The GPU-QM algorithm consists of three steps, and these three steps are implemented iteratively until there are no bad triangles existing in the DT mesh. Figure 3.23 shows the running time of different steps for different point distributions for 1 million input points. As shown by the experiments, the time spent on Step 3 occupies more than 80 percent of the total running time. Since the main operation of Step 3 is edge-flip, the number of flips directly contributes to the total running time. As mentioned before, there are two kinds of flips in Step 3. One is the Delaunay flip, which contributes to maintain the Delaunay property for all triangles after point insertion. The other is non-Delaunay flip, which contributes to delete redundant points. So the number of flips depends on the number of Steiner points inserted and the number of Steiner points deleted in the program. In the following part, we will analysis the number of flips in details.

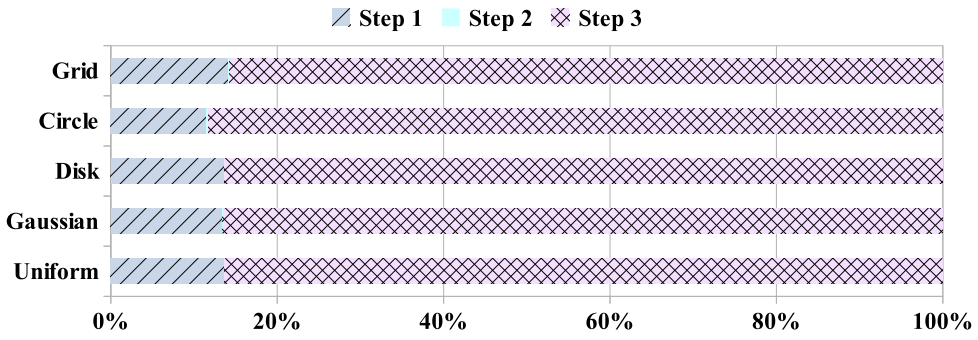


Figure 3.23: The running time of different steps of the GPU-QM algorithm for 1 million points on different point distributions.

Number of flips

As mentioned above, the number of flips depends on the number of Steiner points inserted and the number of Steiner points deleted in the program. We distinguish Steiner points as follows:

Number of Steiner points num_S: the number of Steiner points in the final result.

Number of candidates num_C: the number of Steiner points inserted in total in the program. Many of them can be marked as redundant points and be deleted.

Number of duplicates num_D: the number of Steiner points which are inserted more than one time during the whole process. These points are used to be marked as redundant points and be deleted in some intermediate iterations due to the redundant points detection method used in the algorithm, and are inserted again in later iterations.

The number of Steiner points together with the number of candidates influence the number

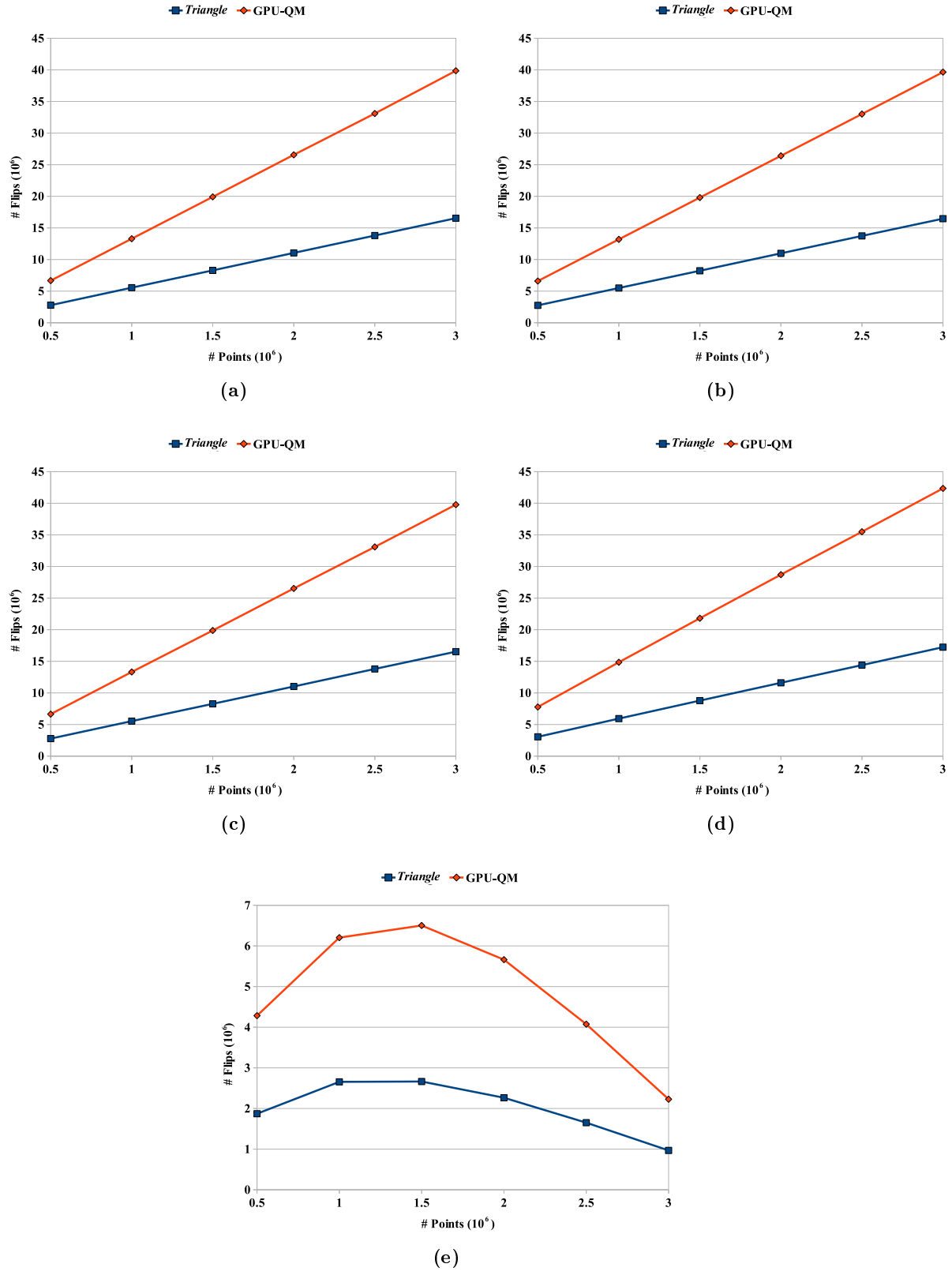


Figure 3.24: Comparison on number of flips between GPU-QM and *Triangle* on different distributions. (a) Uniform. (b)Gaussian. (c) Disk. (d) Circle. (e) Grid.

of flips performed in total. In order to insert num_C candidates, $3 \times num_C$ flips are needed on average, and in order to delete $num_C - num_S$ redundant points, $4 \times (num_C - num_S)$ number of flips are needed. For a given input mesh, in order to reduce the number of flips, i.e., gain minimum running time, we need to reduce the num_C and increase the ratio of num_S/num_C, which is equal to try to reduce the number of the num_D.

As mentioned before, our GPU-QM algorithm only increases no more than 1 percent of Steiner points compared to *Triangle*. However, if comparing the number of flips performed in total, we do more than 2 times of flips than *Triangle* does; see Figure 3.24. According to our statistics, more than 70% Steiner points are duplicate Steiner points, that means most of Steiner points are inserted into the mesh more than one time. In order to reduce the number of candidates and the number of duplicates, we re-design the GPU-QM algorithm with orthogonal-test, which is the GPU-QM-V algorithm. In the next section, we will compare these two algorithms in detail.

Strategies for redundant points detection

In the GPU-QM algorithm, once an edge consisting of two new Steiner points is generated (no matter due to Delaunay flips or non-Delaunay flips), we mark the endpoint with lower priority as the redundant point and delete it in the following edge flips. This strategy may kill many points which are not redundant points at all, due to the fact that after non-Delaunay flips, new Steiner points that are independent with each other may be connected. As a result, more than 70 percent of Steiner points may be inserted more than once in the GPU-QM algorithm, which is a big waste.

In order to solve the defect lies in the GPU-QM algorithm, we use the orthogonal-test as the alternative strategy for redundant points detection in the GPU-QM-V algorithm. According to the experiment results, by using the orthogonal-test, less than 10 percent of Steiner points are inserted into the mesh more than once. Since more than 90% Steiner points are inserted and without being deleted, the num_C decreases a lot (see Figure 3.25), and the $num_C - num_S$ is much smaller than before.

Figure 3.26 shows the comparison on the number of flips when using different detection strategies. In this figure we also include another detection strategy called *no-Deletion*, in which no Steiner points are deleted at all in Step 3. Of course, this strategy may generate shorter edge than the existing edges, so it may not terminate in both theory and practice at all. The reason for including the third strategy in the comparison is that we want to show our GPU-QM-V algorithm can get good balance between the quality of the mesh and the running time of the program with guaranteed termination. Since the number of flips decreases a lot, we gain better performance than the GPU-QM algorithm when comparing with *Triangle*; see Figure 3.27.

Compared to the GPU-QM-V algorithm, the GPU-QM algorithm employs a more aggressive strategy when doing redundant points detection. GPU-QM-V tends to give Steiner points

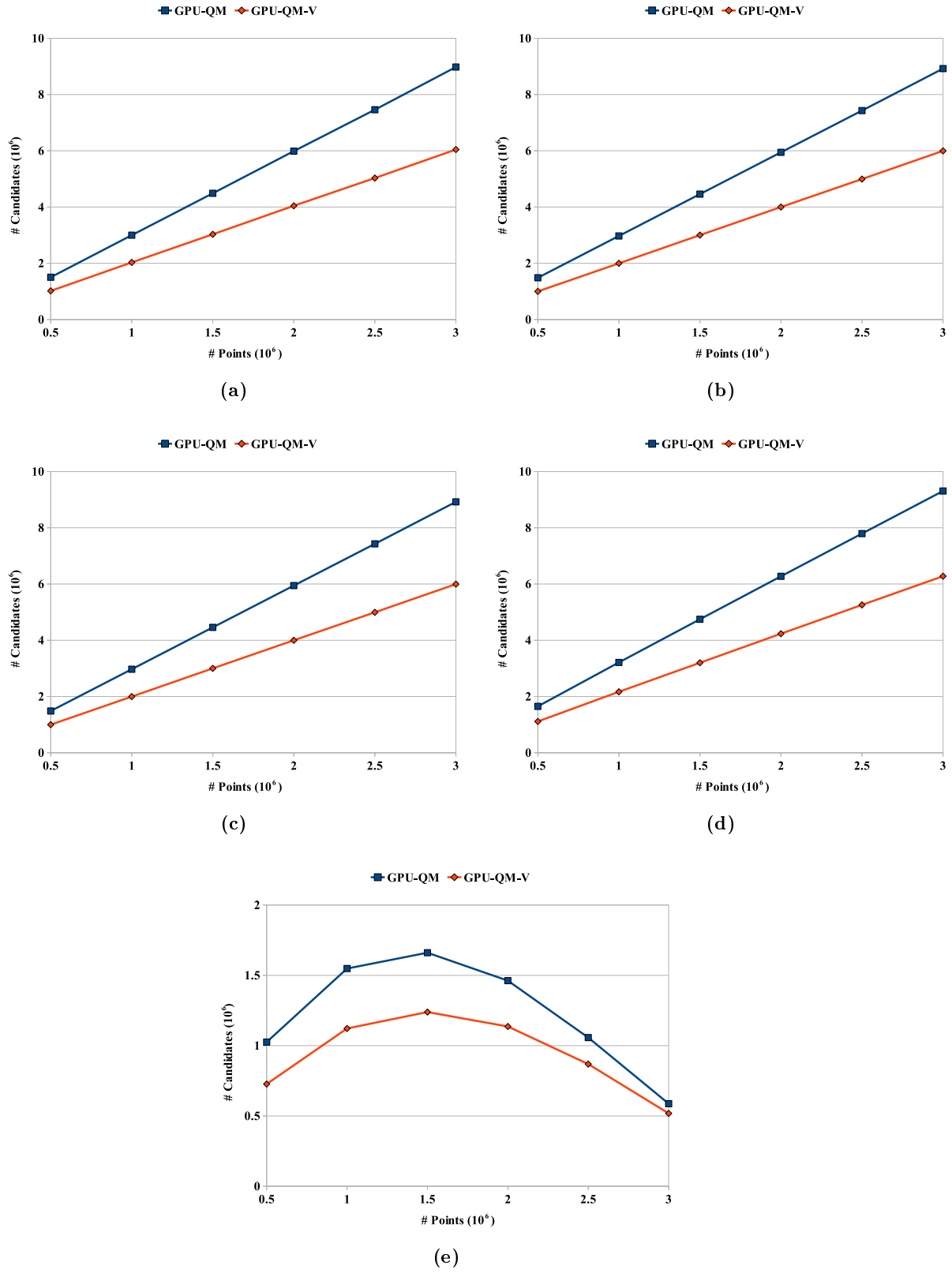


Figure 3.25: Comparison on the number of candidates between GPU-QM and GPU-QM-V on different distributions. (a) Uniform. (b)Gaussian. (c) Disk. (d) Circle. (e) Grid.

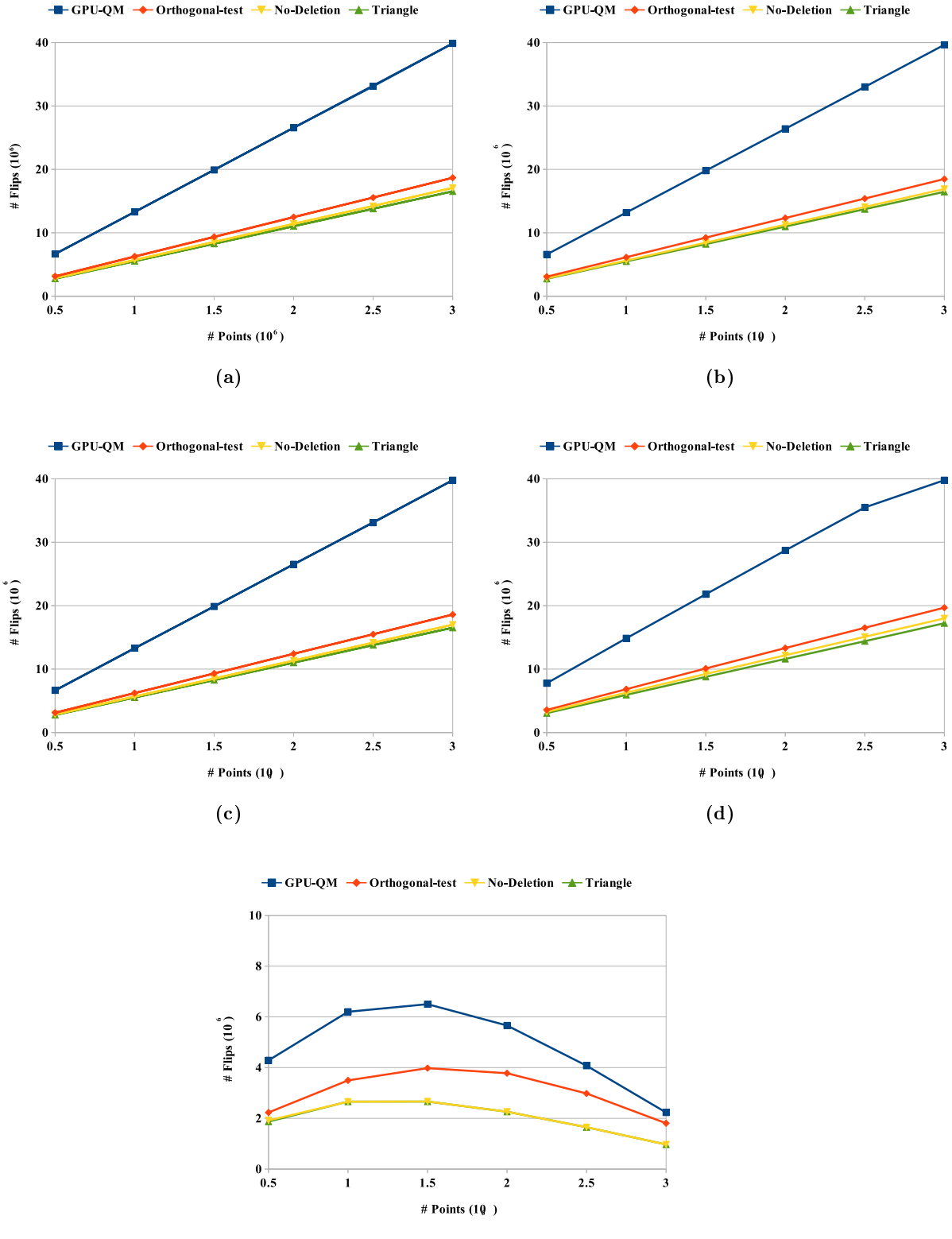


Figure 3.26: Comparison on number of flips for three strategies, GPU-QM, orthogonal-test (i.e., GPU-QM-V), no-Deletion, on different distributions. (a) Uniform. (b)Gaussian. (c) Disk. (d) Circle. (e) Grid.

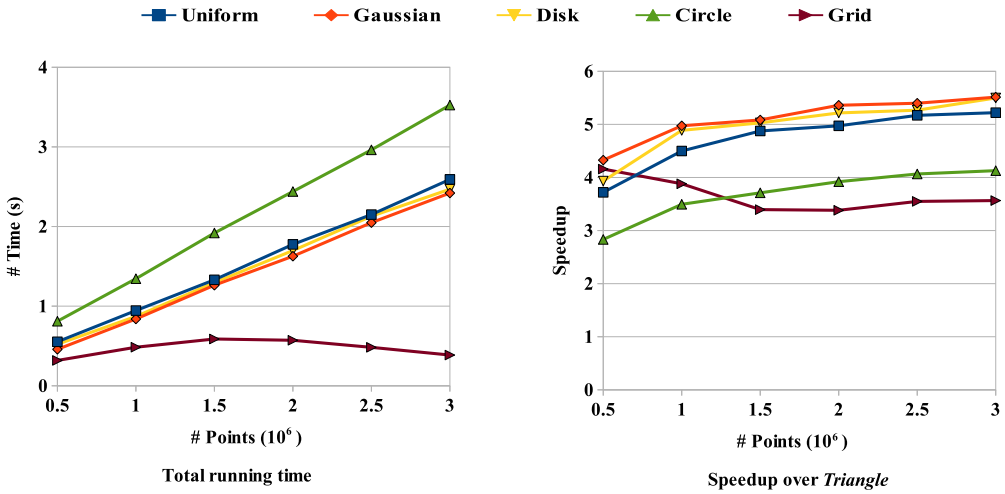


Figure 3.27: The running time and speedup of GPU-QM-V compared to *Triangle* for different point distributions.

that are far away (orthogonal or further than orthogonal) from others an opportunity to stay in the mesh. So it is possible that we do not delete some points that are far away from others but are still not independent with others. As a result, we insert a little bit more Steiner points in the GPU-QM-V algorithm than in the GPU-QM algorithm. According to our statistics, compared to the mesh generated by *Triangle*, GPU-QM inserts less than 1% more Steiner points, while GPU-QM-V inserts less than 4% more Steiner points. As for the speedup over *Triangle*, GPU-QM gains 2 to 3 times speedup (except the circle distribution, which is 0.6 to 1.4 times speedup), while GPU-QM-V gains 3.5 to 5.5 times speedup. Notice that comparing to the GPU-QM algorithm, in the GPU-QM-V algorithm, we insert a little bit more Steiner points, but gain almost doubled speedup.

Although the no-Deletion strategy may gain the best speedup (4 to 6 times) among these three strategies, it may generate 10% more Steiner points, and the most important thing is that it may not terminate in both theory and practice.

Another advantage of GPU-QM-V is that, it can handle the circle distribution very well. The reason is that: although in the last several iterations, bad triangles' circumcircles are large, and they may include several candidates, in the GPU-QM-V algorithm, those candidates are possible far away such that most of them would pass the orthogonal-test, and more candidates are inserted in the same iteration in parallel compared to the number of candidates that are not deleted in the GPU-QM algorithm. From this view, GPU-QM-V has the potential to handle more kinds of point distributions in practice.

Influence of filter

In Step 2, we mentioned that before inserting all candidates, we can use a simple filter to filter out many candidates which intend to be dependent with others. In that filter, for a triangle t which is to be split by inserting a candidate into it, if t's neighbors are also to be split at the same time, we disable one triangle-split, i.e., delete candidates according to priority immediately. This filter is very simple and easy to implement. However, it may influence the number of candidates a lot. Figure 3.28 shows the number of candidates for uniform distributions with and without using the filter. Similar behavior is also observed for other point distributions. Notice that all the results presented prior to this section are already using the insertion filter.

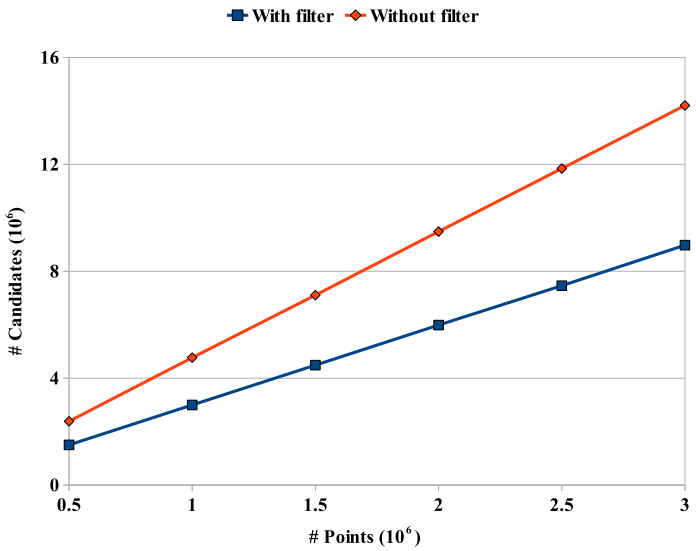


Figure 3.28: Comparison on number of candidates for uniform distribution with and without using the filter.

Reducing the number of candidates means reducing the time for point insertion. Although the time for point insertion (Step 2) is not a time consuming operation (see Figure 3.23), reducing the number of candidates in one iteration may reduce the chances of being dependent with others for each new Steiner point. Therefore the number of Steiner points deleted in Step 3 can also be reduced.

3.7.2 Real-world dataset

To compare our algorithms with *Triangle* on real-world data, we use the contour maps freely available at http://www.ga.gov.au/. Figure 3.29 shows an example of the contour map we used in our experiments. Table 3.1 shows the comparison on running time and the quality of the mesh between our algorithms and *Triangle*. In the table, #I, #O means the number of points in the input and output DT mesh. We gain similar analysis and results on all the measurements we have done for the synthetic data (the running time, the running time

of different steps, the number of flips, the quality of the mesh, and so on). Furthermore, we can apply our algorithms to the image processing problem. Figure 3.30 shows such an example. In addition, we show a zoomed-in image of Figure 3.30 in Figure 3.31, by zooming in the eye portion marked in Figure 3.30.

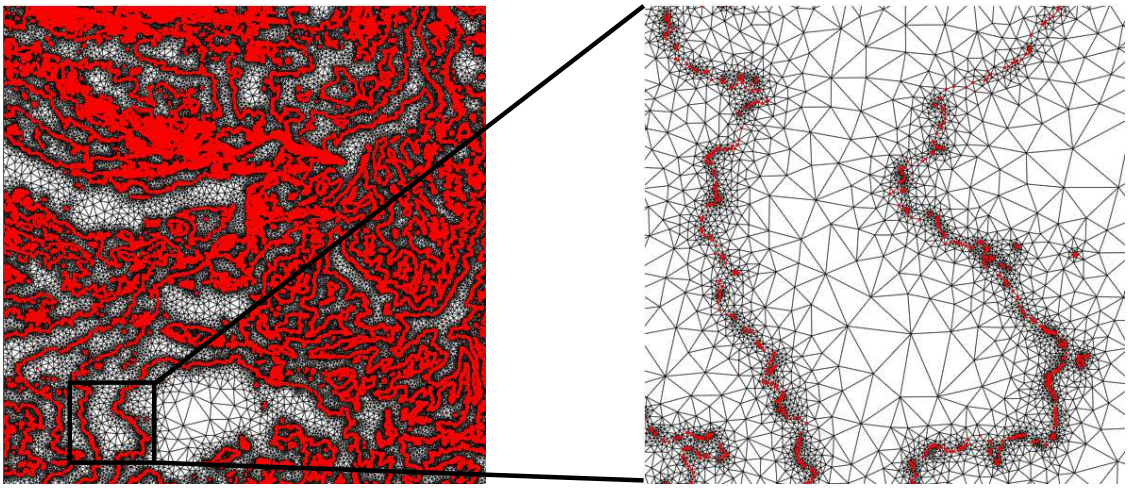


Figure 3.29: An example of contour dataset.

Table 3.1: Statistics for real-world dataset.

Set	#I	*Triangle*		GPU-QM		GPU-QM-V	
		#O	Time (s)	#O	Speedup	#O	Speedup
a	0.5M	2.1M	4.64	2.1M	1.8	2.2M	4.5
b	0.8M	3.3M	7.13	3.3M	2.0	3.5M	4.8
c	1.0M	4.2M	9.56	4.3M	2.1	4.5M	5.0
d	1.2M	5.2M	11.80	5.3M	2.1	5.6M	5.0
e	1.5M	6.4M	14.13	6.4M	2.2	6.8M	5.2

3.8 Summary

In this chapter two algorithms, GPU-QM and GPU-QM-V, are proposed (notice that these two algorithms will be termed as GPU-QM in other places except in this chapter). They are the first GPU algorithms to improve the mesh quality of DT of a set of points. According to our experiments, our algorithms can handle both synthetic and real-world data very well. Furthermore our algorithms are guaranteed to terminate for a given DT mesh if no incident

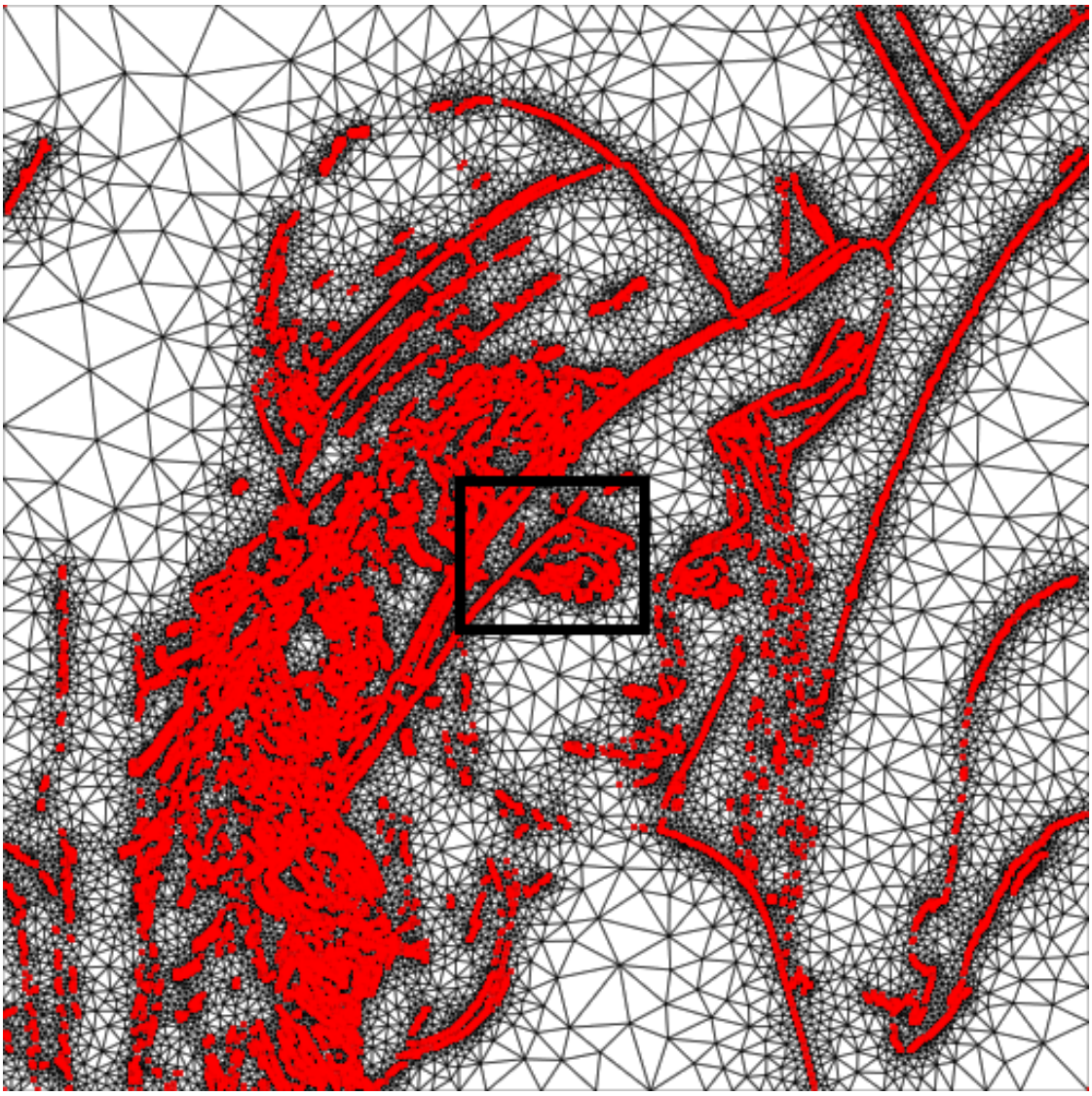


Figure 3.30: A raster image and its quality mesh, in which the red points are the input points abstracted from a raster image.

boundary edges have an angle smaller than 60°. Furthermore, if we enforce all edges in the mesh be Delaunay edges, we can get a conforming Delaunay triangulation.

Although we can handle boundary edges in our algorithms, in some cases we still need to handle constraints inside the convex hull of the DT mesh. Our next goal is to generate quality mesh for a given PSLG. In order to handle constraints for the Delaunay refinement algorithm, we need to generate the CDT first. In the next chapter, we will present our GPU algorithm for the computation of CDT for any given PSLG.

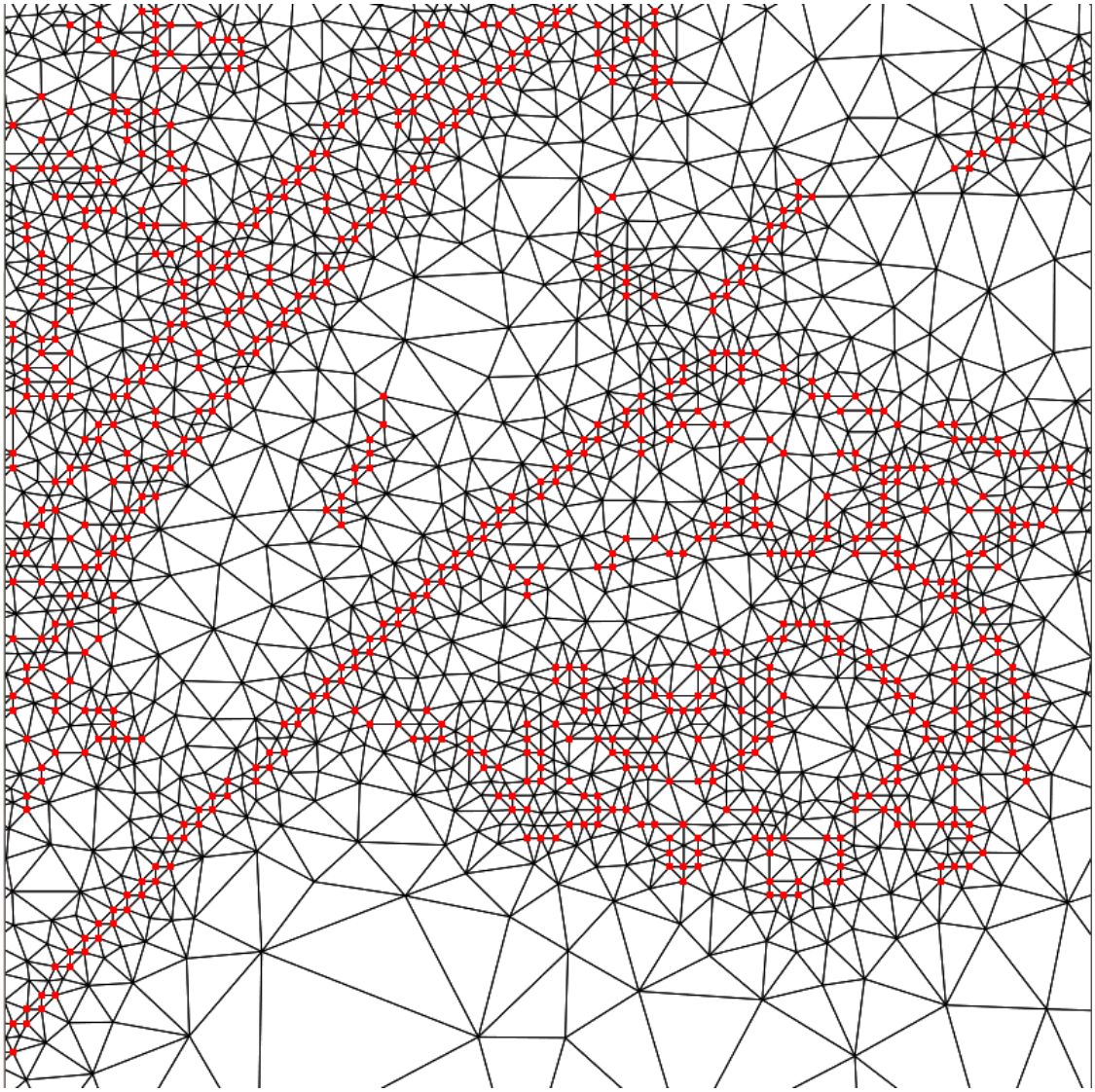


Figure 3.31: Zoom in the portion in black rectangle of Figure 3.30.

Chapter 4

A GPU Algorithm for Constrained Delaunay Triangulation

The constrained Delaunay triangulation (CDT) is a direct extension of the DT where some edges in the output are enforced beforehand; these edges are referred to as *constraints*. Given a PSGL, the CDT is a triangulation of S having all the constraints included, while being as close to the DT of S as possible. For the special case of the CDT problem where the PSLG consists of just points, CDT is the same as the DT. Constraints occur naturally in many applications, such obstacles in path planning, boundaries between cities in GIS, characteristic curves in modeling, and so on.

In this chapter, we propose the first GPU solution (GPU-CDT) to compute the 2D constrained Delaunay triangulation (CDT) of a planar straight line graph (PSLG) consisting of points and edges. There are many existing CPU algorithms to solve the CDT problem in computational geometry, yet there has been no prior approach to solve this problem efficiently using the parallel computing power of the GPU. Our implementation using the CUDA programming model on NVIDIA GPUs is numerically robust, and runs up to two orders of magnitude faster than the best sequential implementations on the CPU. This result is reflected in our experiment with both randomly generated PSLGs and real-world GIS data having millions of points and edges.

This chapter begins with reviewing some sequential algorithms for constructing CDT (Section 4.1). Next the overview and implementation details of the algorithm are presented (Section 4.2 and 4.3). Proof of correction and complexity analysis are provided in Section 4.5. Then the experiment results are provided in Section 4.5. Finally, Section 4.6 concludes the GPU-CDT algorithm.

4.1 Related Work

4.1.1 Sequential method for computing CDT

In this section, we review some algorithms for constructing CDT in 2D.The input for CDT has both points and constraint. According to the order of handling points and constraint, the algorithms for constructing the CDT can be grouped into two categories:

a. *Processing points and constraints simultaneously.*

Chew shows that the CDT can be built in optimal $O(n \log n)$ time using a divide and conquer technique [Che89a], the same time bound required to build the DT, by using a method similar to that used in [Yap87] for building the Voronoi diagram of a set of simple curved segments. This method assumes that all the vertices and constraints are contained within a given rectangle. Firstly, they sort all the vertices by x-coordinate, and then use this information to divide the rectangle into vertical strips such that every strip only contains one vertex. In each vertical strip, they compute the CDT, adjacent strips are pasted together in pairs to form a bigger strips, and the CDT is built for each newly formed strip until the CDT for the entire rectangle has been built. During the CDT pasting operation, the author uses two tricks to reach the $O(n \log n)$ time bound: one is only keep track cross edges that bound vertex-containing regions, the other one is using infinite vertices so that partial CDTs are linked for efficient access.

Domiter uses sweep-line algorithm to process points and constraints at the same time [Dom04]. Insertion of edge is postponed until it is swept entirely. In this way, the determination of triangles being pierced by the edge is efficient, simple and does not require any additional searching data structure. This work is based on a DT sweep-line algorithm represented in [Zal05].

b. *Processing points and constraints separately.* Since the CDT is a generalization of the DT with the notion of constraints [LL86], we can first construct the DT of the given set of points, then insert constraints one by one into it. Such an insertion can be done either by removing triangles pierced through by each constraint and re-triangulate the region due to the removal of these triangles, or by flipping some edges in a certain order until the constraint appears in the triangulation [Ber95, She96a].

Our approach of computing the CDT on the GPU lies in-between these two categories. We first construct a triangulation of the given set of points, then insert all the constraints using edge flipping, followed by transforming the resulting triangulation into the CDT, also using edge flipping.

4.1.2 GPU-DT algorithm

Since before we insert all constraints, we should use the GPU-DT algorithm to generate a DT for a set of points, the GPU-DT algorithm will be shown briefly in this section.

There are several GPU-DT algorithms we can use as mentioned before. Here we only introduce the one published in [QCT12, QCT13]. This algorithm derives from the digital VD of the input set of points S an approximation of the DT, then transforms it into the needed DT. Specifically, the algorithm consists of the following phases:

Phase 1. Digital Voronoi diagram construction.
Map the input points into a grid and compute its digital VD. If more than one point is mapped to a same grid point, keep just one and treat the other as missing points.

Phase 2. Triangulation construction.
Find all the digital Voronoi vertices to construct triangles for a triangulation. This triangulation is an approximation of the DT.

Phase 3. Shifting.
Points have been moved due to the mapping in Phase 1. Shift points back to their original coordinates and modify the triangulation if necessary.

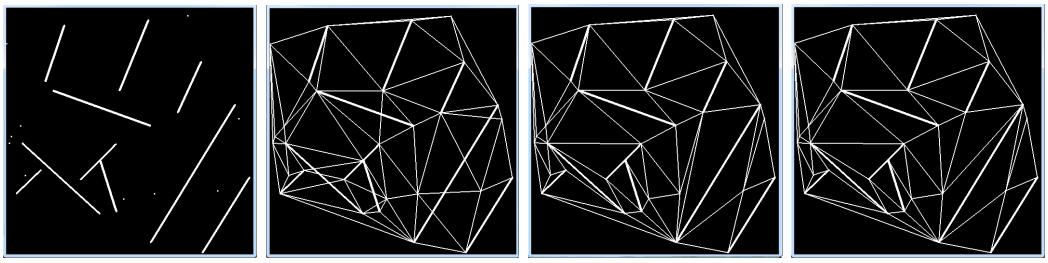
Phase 4. Missing points insertion.
Insert all missing points to be a part of the triangulation.

Phase 5. Edge flipping.
Verify the empty circle property for each edge in the triangulation, and perform edge flipping if necessary.

4.2 Motivation and Algorithm Overview

To introduce constraints into the DT computation, we use the approach of computing a triangulation $\mathcal{T}$ of the set of points first before incorporating the constraints. This is because considering constraints earlier in the digital VD computation makes the dualization much more difficult, and the correctness of the resulting triangulation might not be guaranteed.

The naïve approach of having one parallel thread to handle one constraint, deleting triangles that it pierces through and re-triangulating the created region is not ideal. This is because each constraint can intersect a different number of triangles in $\mathcal{T}$, resulting in unbalanced workloads. Furthermore, two different threads handling two constraints may intersect some common triangles and the threads cannot proceed without some sort of locking, which is very costly on the GPU.



(a) (b) (c) (d)

Figure 4.1: Steps of the GPU-CDT algorithm. (a) Input PSLG; (b) Step 1: Triangulation construction; (c) Step 2: Constraints insertion; (d) Step 3: Edge flipping. (Thick lines are constraints)

To achieve good parallelism, we employ the flipping approach to insert constraints. Multiple pairs of triangles intersected by the same constraint can possibly be flipped in parallel. Also, when two constraints intersect some common triangles, we can still possibly flip some of these common triangles. To regularize work among different threads, this flipping is done before Phase 5 of the DT algorithm in Chapter 4.1.2, so that we can focus on inserting the constraints first, before worrying about the empty circle property. Our algorithm can be summarized as follows:

Step 1. Compute a triangulation $\mathcal{T}$ for all points (Phases 1 to 4);

Step 2. Insert constraints into $\mathcal{T}$ in parallel;

Step 3. Verify the empty circle property for each edge (that is not a constraint), and perform edge flipping if necessary.

Figure 4.1 shows the an example of the 3 steps for a input PSLG. Step 3 is similar to Phase 5 of the DT algorithm, with the slight modification not to flip constraints. Our proposed Step 2, with an *outer loop* and an *inner loop*, is given in Algorithm 4.1. The idea is to identify intersections between constraints and triangles, i.e. constraint-triangle intersections, in the outer loop, and to use edge flipping to remove them in the inner loop, all in parallel using multiple passes.

4.3 Algorithm Details

4.3.1 Outer loop: Find constraint-triangle intersections

For each triangle in the triangulation, we find the index of one constraint intersecting it, if any. Let $c_i = ab$ be the i^{th} constraint in the input, we go through the triangle fan of a

Algorithm 4.1: Inserting constraints into the triangulation

```
1  repeat                                                    /* outer loop */
2      forall the constraint c_i do
3          mark triangles intersecting c_i with i using atomic minimum
4      end
5      repeat                              /* inner loop:  see Algorithm 4.2 */
6          do edge flipping to remove intersections to constraints
7      until no edge is flippable
8  until all constraints are inserted
```

to identify the triangle A intersected by c_i. If c_i is an edge of A, the constraint is already there in the triangulation and no further processing is needed. Otherwise, from A we start walking along the constraint towards b, visiting all triangles intersected by c_i; see Figure 4.2. For each triangle found, we mark it with the index i using the atomic minimum operation. Letting the minimum index remain as the marker is necessary for our proof of correctness. Since we do not modify anything in the triangulation in this step, no locking is needed. The work done in this outer loop achieves coarse-grained parallelism on the GPU, with one thread processing one constraint.

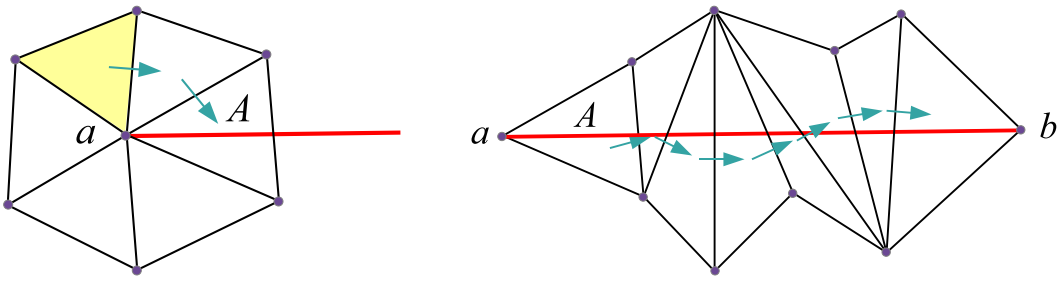


Figure 4.2: (a) Find the first triangle A intersected by the constraint (red line), yellow triangle is the first triangle incident to the a in the vertex array. (b) How to find the other intersected triangles along the constraint (red line).

4.3.2 Inner loop: Remove intersections

The inner loop of Algorithm 4.1 performs edge flipping to reduce the number of constraint-triangle intersections. Here, the parallelism is fine-grained with each thread processing a triangle. Consider a pair of adjacent triangles that are both marked by the same constraint. There are four kinds of configuration for it; see Figure 4.3. A pair is classified as a *zero*

intersection, *single intersection* or *double intersection* configuration if flipping it results in a new pair having zero, one or two intersections with the constraint, respectively. If the flipping is not allowed as its underlying space is a concave quadrilateral, the pair is classified as a *concave* configuration.

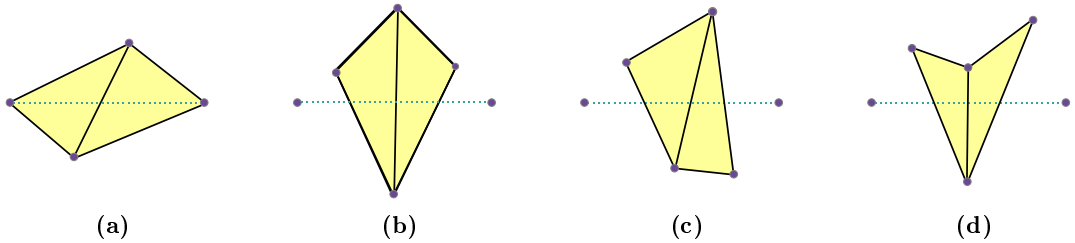


Figure 4.3: Configurations of a triangle pair intersecting a constraint (drawn in dashed line). (a) Zero intersection. (b) Single intersection. (c) Double intersection. (d) Concave.

It might seem good to avoid flipping a double intersection configuration since flipping does not "improve" the situation and thus possibly leading into an infinite loop. However, we note that restricting flipping to only zero and single intersection is not sufficient to get rid of all the constraint-triangle intersections. To overcome this dilemma, we propose a one-step look-ahead strategy. Consider a triangle A in the chain of triangles intersected by a constraint, from one end point to the other, and let B and C be the previous and the next triangle in that chain. The triangle pair (A, C) is *flippable* in one of the following cases: (Figure 4.4)

Case 1. (A, C) is a zero or single intersection.

Case 2. (A, C) and (B, A) are both double intersections, and flipping (A, C) would result in B with its new next triangle forming a single intersection.

Case 3. (A, C) is a double intersection and (B, A) is concave, and flipping (A, C) would result in B with its new next triangle no longer being concave.

Note that Case 2 is equivalent to $(B \cup A \cup C)$ being a convex polygon. We perform the flipping in multiple iterations; see Algorithm 4.2. In each iteration, we first identify the triangle pairs and their configurations (line 2–6). Then for any flippable pair (A, C) as described above, we mark A, C, and possibly the previous triangle of A (which is B in our discussion), with the index of A using the atomic minimum operation (line 7–11). Lastly, we flip (A, C) only if the marks on A and C (and also B for Case 2 and Case 3) remain (line 12–16). This prevents the possible conflicts when updating the triangulation, and allows the one-step look-ahead to be achieved. We also introduce extra weights into the labels used in the marking. Case 1 is given the highest weight, followed by Case 2 and then Case 3. For each step to be done in parallel, we assign one thread to process one triangle. As an optimization, after each iteration, we maintain a compact list of *active* triangles, i.e. those that are still intersected by the constraints. As such, in later iterations, we do not have too

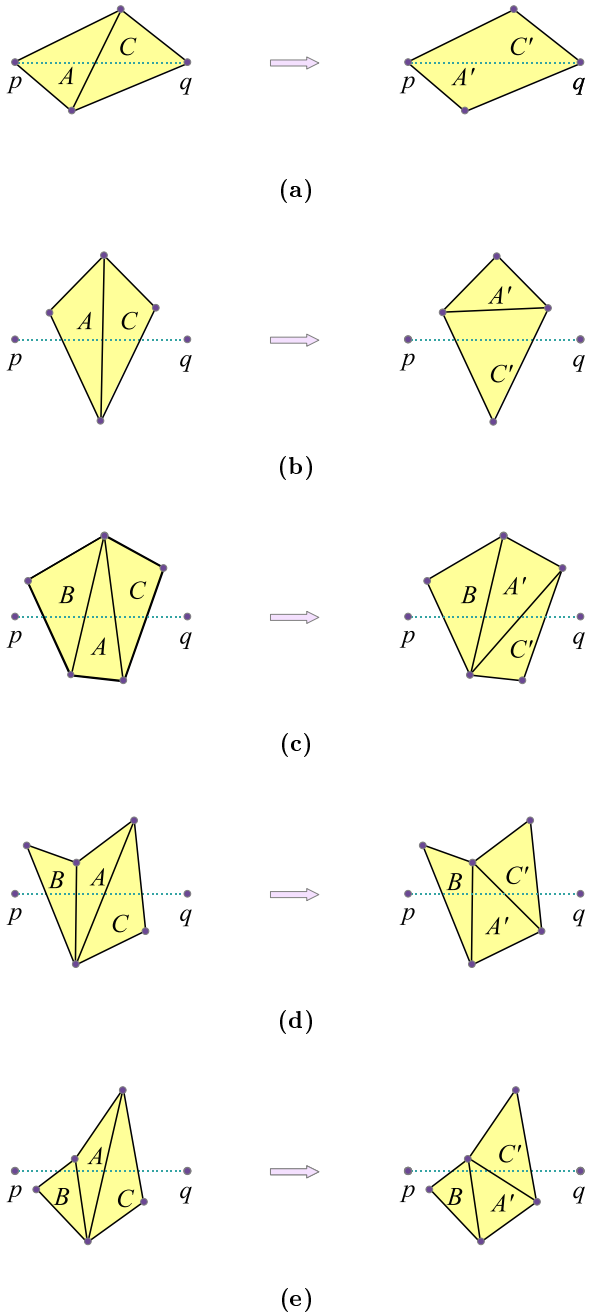


Figure 4.4: Flipping consideration of a triangle pair involving A. The constraint pq intersects the triangles from left to right. (a) Case 1a. (b) Case 1b. (c) Case 2. (d) Case 3a. (e) Case 3b.

many idle threads handling triangles that no longer active.

In practice, the **repeat-until** loop in Algorithm 4.2 is only executed a few times per outer loop iteration instead of repeating until no edge is flippable. The reason is as the algorithm

Algorithm 4.2: Processing of constraint-triangle intersections

```
1  repeat
2      forall the triangle A intersecting a constraint do
3          if C is also marked by the same constraint as A then
4              determine the case of (A, C)
5          end
6      end
7      forall the triangle A intersecting a constraint do
8          if (A, C) is flippable then
9              mark A, C (and B for Case 2 and Case 3) using atomic minimum
10         end
11     end
12     forall the triangle A intersecting a constraint do
13         if A, C (and B for Case 2 and Case 3) retain the same mark then
14             flip (A, C) and update the links between the new triangles and their
               neighbors
15         end
16     end
17 until no edge is flippable
```

progresses, there is a drastic reduction in the number of flippable cases, and thus the parallelism reduces. By switching to the outer loop after a few (say 5 to 10) iterations of the inner loop, the algorithm can discover more flippable cases to improve the parallelism and as a result improving the performance, without compromising on the correctness proven in the next section.

4.4 Proof of Correctness and Complexity Analysis

4.4.1 Proof of correctness

We show that Algorithm 4.1 terminates with all constraints inserted into the triangulation. Consider one iteration of the outer loop, and let $c_i = ab$ be the constraint with the smallest index i that still intersects some triangles in our triangulation. By using the atomic minimum operation, we ensure that all triangles intersecting c_i are marked with i. It thus suffices to

prove the following:

Claim 4.4.1. *The inner loop always successfully inserts one constraint into the triangulation.*

Proof. Consider the chain of triangles intersecting c_i from a to b. Among these triangles, if there are one or more triangle pairs that are single or zero intersection, then the claim is true as the marking favors each of these cases and flipping is indeed carried out, reducing the number of triangles intersecting c_i. Otherwise, consider the chain of triangles having only double intersection or concave configurations. We argue that there exists a triangle pair (A, C) among them that is flippable, and each flip is a step closer to removing the intersections of the triangles with c_i.

If we would remove all triangles intersecting c_i, a polygonal hole is created with points $p_1, p_2, \ldots$ as its upper part and $q_1, q_2, \ldots$ as its lower part, excluding a and b; see Figure 4.5a.

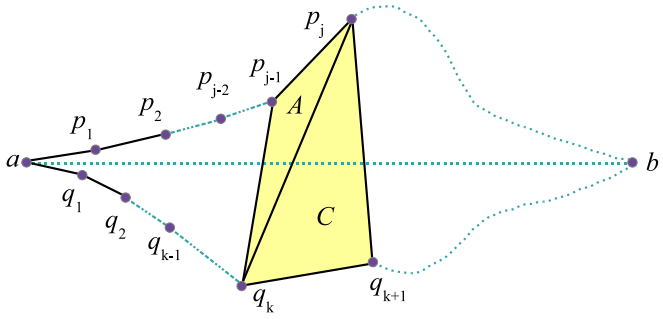
Any polygon has an ear. Let $q_{k-1}q_kq_{k+1}$ be the ear such that the triangle $C = q_kq_{k+1}p_j$ incident to q_kq_{k+1} and intersected by ab is the earliest one in the chain. We exclude a and b themselves to be q_k. Let A be the previous triangle of C and B be the previous of A. We have $A = q_kp_jp_{j-1}$ since if it were $q_kp_jq_{k-1}$, (A, C) would have been a single intersection pair. The triangle pair (A, C) is a double intersection, since the two angles $p_{j-1}p_jq_{k+1}$ and $p_{j-1}q_kq_{k+1}$ are both less than π.

We claim that (A, C) is flippable by considering 3 cases of B as follows. If $B = q_kp_{j-1}p_{j-2}$, (B, A) is a concave pair; see Figure 4.5b. $p_{j-2}p_{j-1}q_{k+1}q_k$ is convex by the choice of q_k, thus triangles B, A, C fulfill Case 3a, and (A, C) is flippable. If $B = q_kp_{j-1}q_{k-1}$ and (B, A) is a double intersection (see Figure 4.5c), the union of triangles B, A, C is a convex polygon as needed in Case 2, so (A, C) is flippable. If $B = q_kp_{j-1}q_{k-1}$ and (B, A) is a concave configuration (see Figure 4.5d), the union of triangles B, A, C fulfill Case 3b, so (A, C) is also flippable. As long as there is one flippable triangle pair, the marking in the inner loop in Algorithm 4.1 will successfully mark one for flipping, and flipping is indeed performed in each pass.

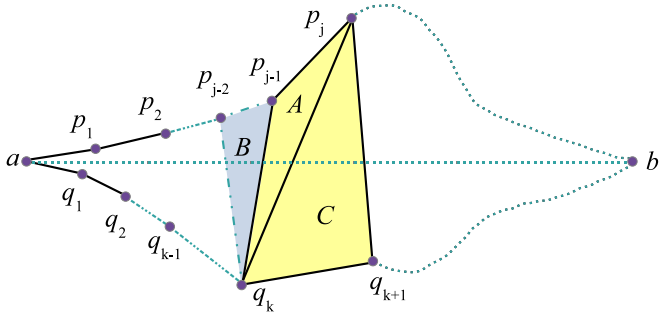
We next show that our inner loop does not go on forever. Let us assign to each pair of triangles a value of 0, 1 or 2. A pair of triangles that is zero or single intersection is assigned value 0; a double intersection, value 1; and a concave, a value 2. As a result, we have a base 3 number, N, to record the cases of the chain of triangles intersecting c_i. A flip due to Case 1 deletes a digit in N; Case 2 turns 11 into 01; and Case 3 turns 21 into 11 (Case 3a) or 01 (Case 3b). In other words, each flip decreases the value of N. Since N is finite, our algorithm clearly terminates, and a constraint is inserted as claimed. □

4.4.2 Complexity analysis

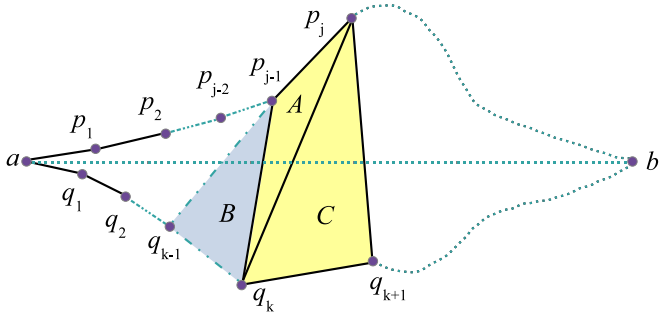
Claim 4.4.1 concludes that our proposed algorithm computes the CDT correctly. In this subsection, we show that no flip is wasteful. We first analyze the number of flips needed to



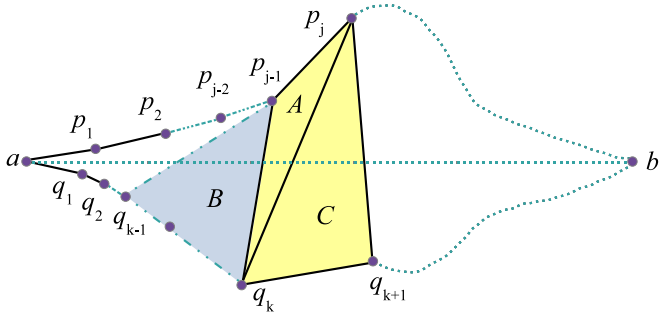
(a)



(b)



(c)



(d)

Figure 4.5: (a) When the triangle pairs intersecting the constraint $c_i = ab$ are only either double intersection or concave, there exists a flippable pair (A, C). (b) B, A and C fulfill Case 3a. (c) B, A and C fulfill Case 2. (d) B, A and C fulfill Case 3b.

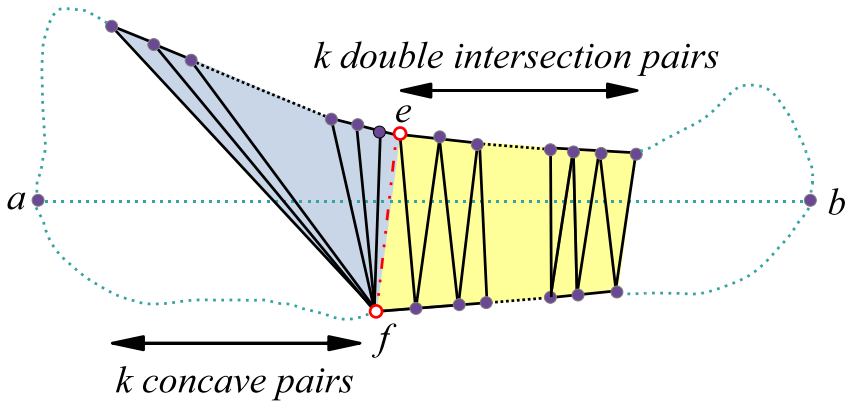
insert one constraint, followed by a bound on the total number of flips needed to insert all the constraints.

Claim 4.4.2. *The total number of flips performed by the inner loop in order to insert one constraint is $O(k^2)$, where k is the number of triangles intersecting the constraint.*

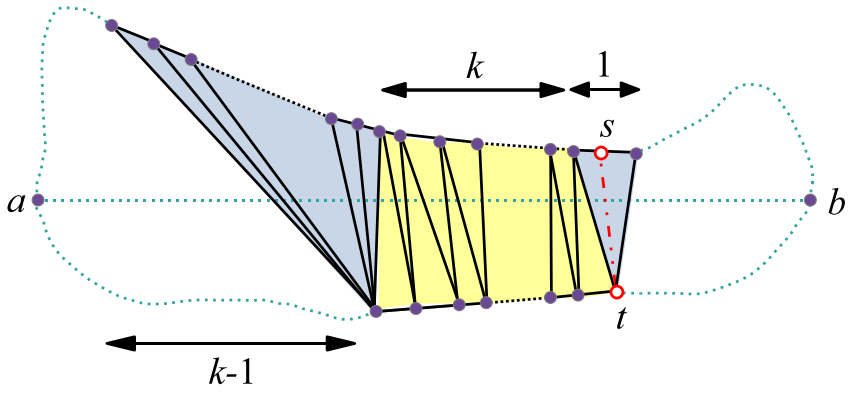
Proof. Flipping due to Case 1 cannot be done more than k times since each flipping removes an intersection. Flipping due to Case 2 immediately gives rise to a flipping of Case 1 (with highest priority), and thus also cannot be done more than k times.

The initial number of concave pairs is bounded by $O(k)$. A flip due to Case 1 (or Case 2) introduces at most two (or one) concave pairs, thus at most $O(k)$ concave pairs can be introduced by these two flipping cases. Flipping due to Case 3 either eliminates a concave pair or pushes it towards one end of the constraint, thus it can be performed no more than $O(k^2)$ times. As a result, the total number of flips is $O(k^2)$. □

Figure 4.6a shows that the worst case of Claim 2 can happen. In this example, among



(a)



(b)

Figure 4.6: A bad case for inserting one constraint.

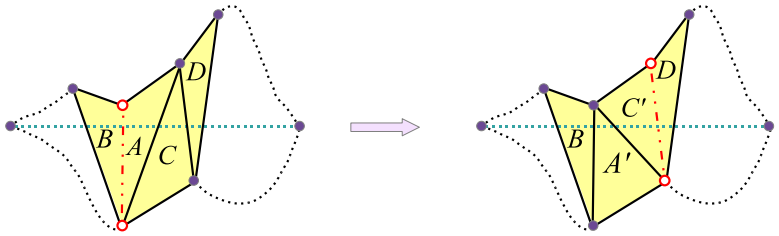


Figure 4.7: Push the concave pair towards the right end of the constraint using a flipping due to Case 3a.

all the triangles intersected by the constraint, there is a particular chain of triangles with k successive concave pairs followed by k double intersection pairs, such that none of the double intersection pairs fulfill Case 2. The only flippable case in this situation is due to Case 3a with two triangles sharing the edge ef. Flipping this pair of triangles and moving on with another $k-1$ flippings due to Case 3a, the algorithm produces Figure 4.6b. The concave pair on the left of the k double intersections has been "removed" and "introduced" in the right at the edge st. In other words, the concave pair is *pushed* towards one end of the constraint. As such, for each concave pair, $O(k)$ flips are needed to remove it, and since there are k concave pairs being removed in parallel, we need $O(k)$ iterations of the inner loop and $O(k^2)$ flips are performed.

One technicality remains in the above construction. We add an extra $O(k)$ concave pairs on the left and $O(k)$ double intersection pairs on the right of the chain of triangles shown in Figure 4.6a. This is to make sure that the triangle pairs incident to a and b can be flipped $O(k)$ times in the $O(k)$ inner loop iterations without affecting the chain of concave and double intersection pairs shown in the figure. The total number of triangles intersecting the constraint is $O(k)$, so our $O(k^2)$ bound on the number of flips needed is tight.

Following the previous claim, in the worst case the total number of flips performed to insert all the constraints may reach $O(n^3)$. However, this bound is not tight. In the next claim, we show that the total work for inserting all constraints is $\Theta(n^2)$.

Claim 4.4.3. *The total number of flips performed by Algorithm 4.1 is $\Theta(n^2)$, where n is the number of input points.*

Proof. For any constraint c_i, if all triangles intersecting it are deleted, a polygonal region is left with s_i points in its upper part and t_i points in the lower part, excluding the endpoints of c_i. The number of triangles intersecting c_i is $k = s_i + t_i$.

According to Claim 2, flipping due to all cases except Case 3a can only be done $O(k)$ times. Flipping due to Case 3a either eliminates a concave pair or pushes it towards the right end of the constraint. When a concave pair is pushed, it moves one step to the right end in both the upper boundary and the lower boundary of the polygonal region (Figure 4.7), thus

each concave pair can only be pushed $\min(s_i, t_i)$ times. As such, the total number of flips to insert c_i is $O(k + k \ \min(s_i, t_i)) \subset O(s_i t_i)$. The total number of flips performed for m constraints is thus $O(\sum_{i=1}^{m} s_i t_i)$.

Each time a constraint c_i is inserted, any edge $p_i q_j$ can never appear later in the triangulation. Not only that, such an edge also cannot be inside the polygonal region of any other constraint, otherwise that region would have intersected c_i. The number of edges that can possibly be formed by the n input points is $O(n^2)$, so $O(\sum_{i=1}^{m} s_i t_i) \in O(n^2)$. This, together with the worst case example shown above, concludes our proof of Claim 3. □

4.5 Experiment Results

To achieve exact and robust result during our computation, we only use orientation and in-circle predicates from the exact predicates of Shewchuk [She96b]. All numbers and computations are done in double precision. The source code is available for download from our project website (`http://www.comp.nus.edu.sg/~tants/cdt.html`).

To assess the efficiency of our GPU-CDT program, we compare its running time, on both synthetic and real-world data, with that of the most popular computational geometry softwares available, *Triangle* and CGAL version 3.9. According to our tests, CGAL runs faster than *Triangle* for the DT computation. However, when constraints are introduced, *Triangle* runs much faster than CGAL. Here, we only show the result of the faster between the two.

4.5.1 Synthetic dataset

To generate synthetic data, we first randomly generate constraints of different lengths that do not intersect each other, then randomly generate points which do not lie on any constraint. Figure 4.8 shows one small synthetic data generated.

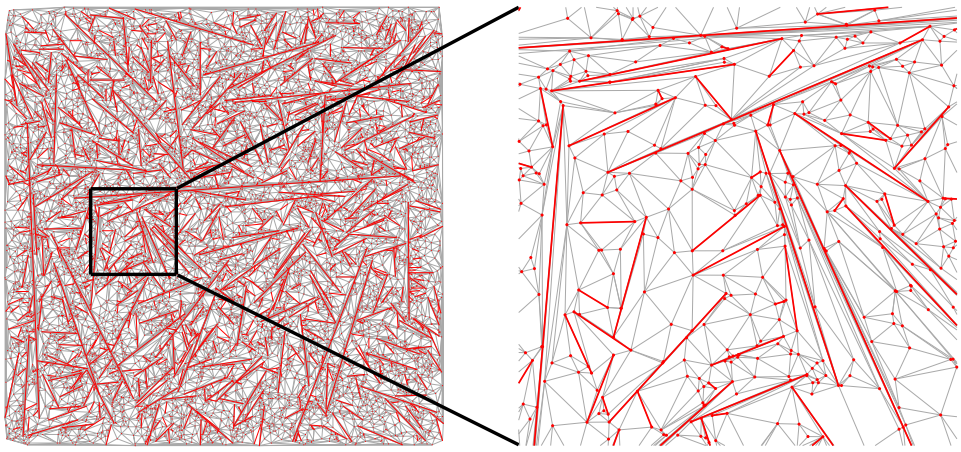


Figure 4.8: A synthetic dataset (left) and its constrained Delaunay triangulation (right).

When constraints are introduced, we observe a substantial speedup, of up to two orders of magnitude, compared to both *Triangle* and CGAL (with CGAL being much slower than *Triangle*). *Triangle* inserts constraints one by one (also using an edge-flip method) on the DT of the set of points. We compare the time for constraints insertion by subtracting the time for the DT computation from the time for the CDT computation on the same set of points.

Running time: Figure 4.9a and 4.9b show the performance comparison of *Triangle* with GPU-CDT on different number of points and constraints, with different grid sizes. Clearly, the more constraints there are, the higher is the speedup we can achieve. This is because only a small part of our algorithm in inserting constraints is done with coarse-grained parallelism, while the majority of the processing is done with fine-grained parallelism. As such, our algorithm scales well with the amount of work available. Note that we achieve

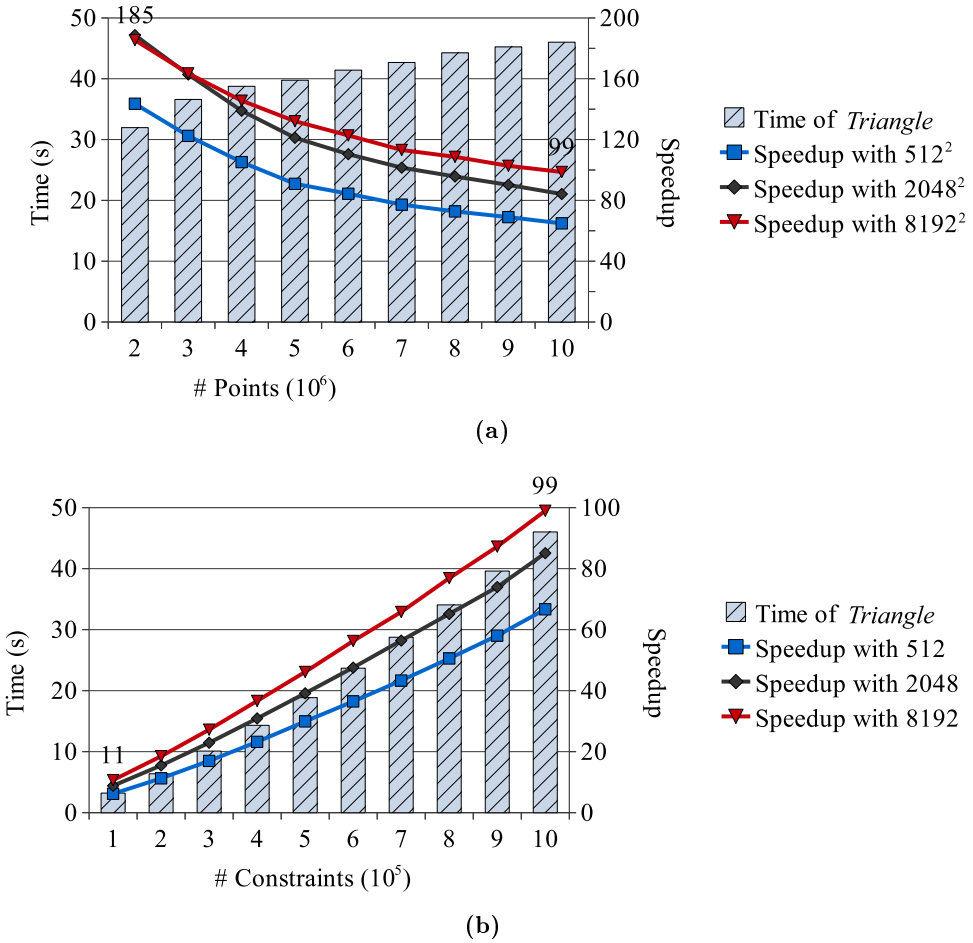


Figure 4.9: Speedup over *Triangle* when computing the CDT, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

better performance for constraints insertion when using bigger grid sizes because the number of constraint-triangle intersections decreases (see Figure 4.10a and 4.10b), possibly due to the fact that the triangulation produced by Step 1 is closer to the DT with grid size getting bigger.

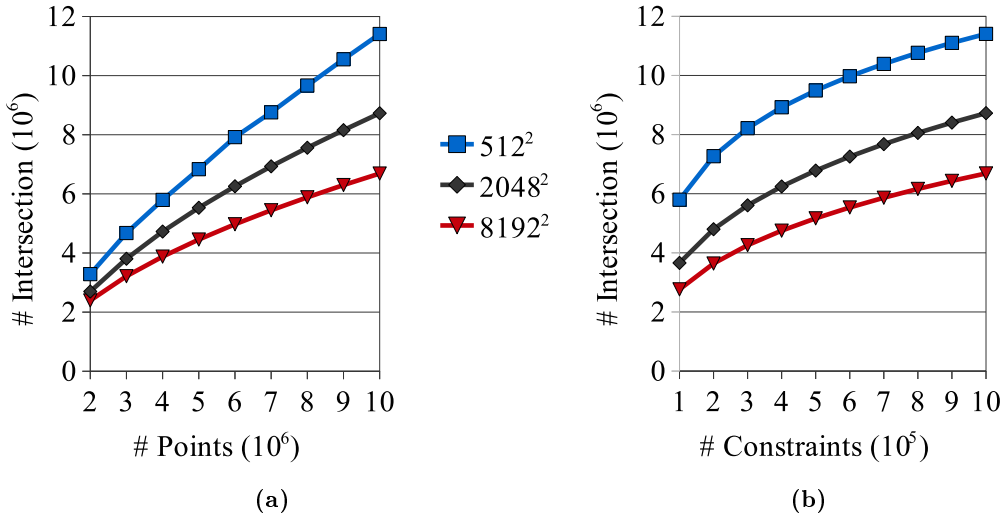


Figure 4.10: Total number of constraint-triangle intersections with different grid sizes, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

Running time of different phases: Figure 4.11a and 4.11b shows the running time of different phases of GPU-CDT using 8192^2 grid size. Similar behavior is also observed for other grid sizes. The time for inserting constraints for our program occupies less than 20% of the total time. On the same datasets, *Triangle* spends most of its time inserting constraints. For example, given 10M points and 1M constraints, *Triangle* spends 62 seconds for constructing the CDT, in which 46 seconds are spent on constraints insertion. In contrast, GPU-CDT spends 3.2 seconds for constructing the CDT, in which only 0.47 seconds are spent on constraints insertion. As such, when comparing the total running time of our program with that of *Triangle*, we achieve significant speedup, ranges from 10 to 49 times.

Number of flippings: We compare with *Triangle* on the number of flippings needed to insert constraints. To get a fair comparison, we follow *Triangle*'s approach by modifying our implementation to insert the constraints after we have computed the DT. The number of flippings for GPU-CDT to insert constraints include both the flippings to make the constraints appear in the triangulation and those to make the triangulation a CDT.

Figure 4.12a and 4.12b shows the comparison with varying number of points and constraints. The number of flippings is plotted against the number of constraint-triangle intersections to highlight that the relationship between the two numbers is linear in practice, much lower than the worst case complexity analyzed in Section 4.4.2. Note that we perform slightly less

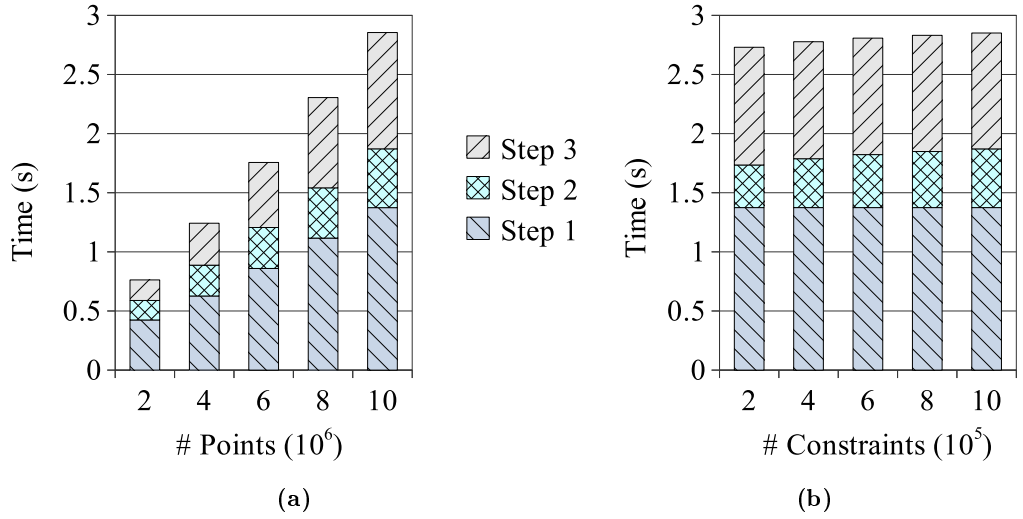


Figure 4.11: Running time for different steps for computing CDT, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

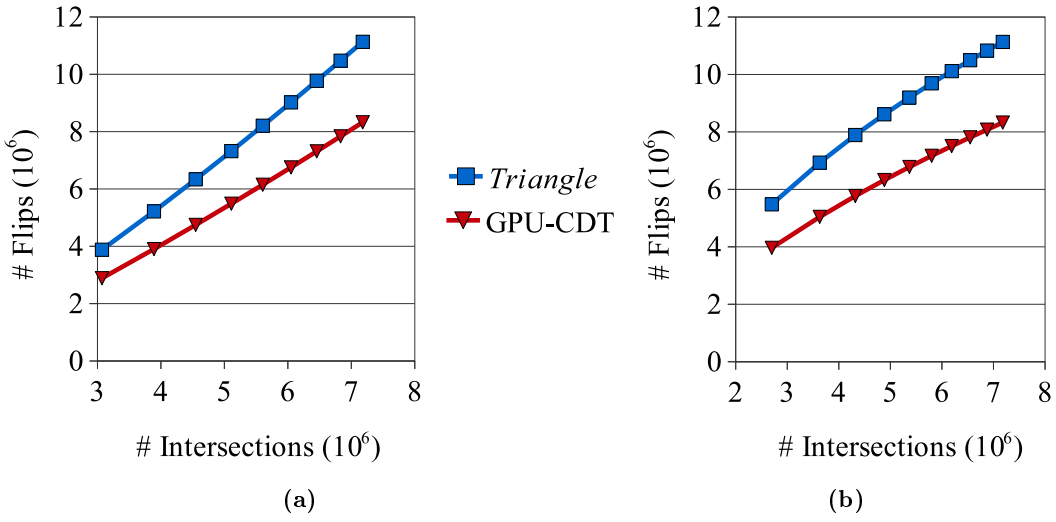


Figure 4.12: Comparison with *Triangle* on the total number of flippings when inserting constraints, (a) with 1M constraints and varying the number of points, and (b) with 10M points and varying the number of constraints.

edge flippings than *Triangle*, possibly due to our algorithm giving extra weights to Case 1 and Case 2 that leads to immediate removal of constraint-triangle intersections.

4.5.2 Real-world dataset

To compare GPU-CDT with *Triangle* for the real-world dataset, we use contour maps which are freely available at https://www.ga.gov.au/. Figure 4.13 shows an example of the contour maps we used for our experiment and its CDT. For such contour maps, the number of constraints is similar to the number of points. Here we use the number of points to denote the data size of the contour map. Table 4.1 shows the running time comparison with *Triangle* for 6 contour map examples.

Case	Data Size	Constraints Insertion (s)		Speedup
		Triangle	GPU-CDT	
a	1.2M	0.665	0.046	14×
b	3.2M	1.982	0.071	28×
c	4.5M	2.526	0.097	26×
d	5.7M	3.181	0.133	24×
e	8.5M	4.755	0.245	19×
f	9.5M	6.036	0.244	24×

Table 4.1: Running time of contour dataset.

GPU-CDT generally runs faster than *Triangle*. In these real-world data, most constraints are very short and do not intersect many triangles (if at all). Figure 4.14 shows the dis-

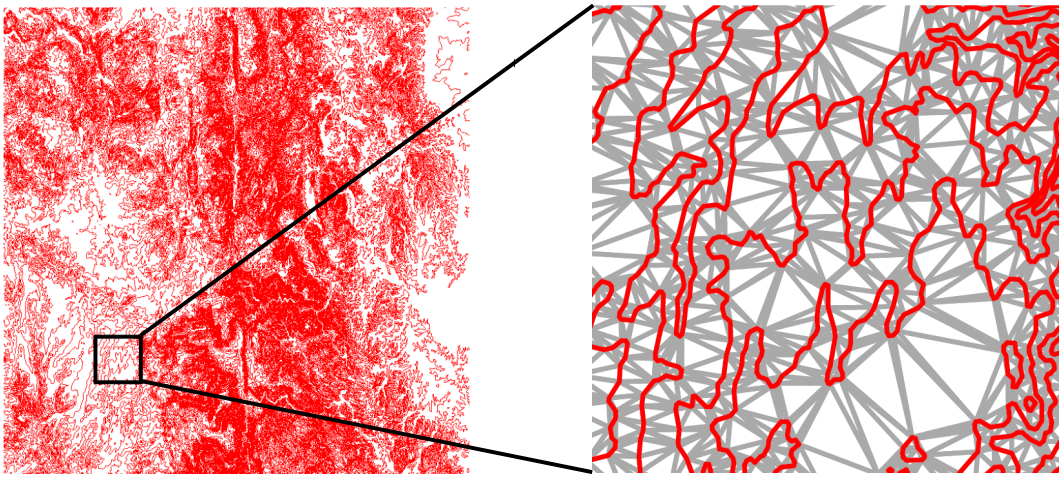


Figure 4.13: A contour map with its CDT

tribution of the number of intersections per constraint collected by our GPU-CDT for the Example f contour dataset (with about 9.5M points and 9.5M constraints) and a representative synthetic dataset of 10M points with 1M constraints. The maximum number of intersections is 51 for the contour dataset as compared to 7073 for the synthetic dataset.

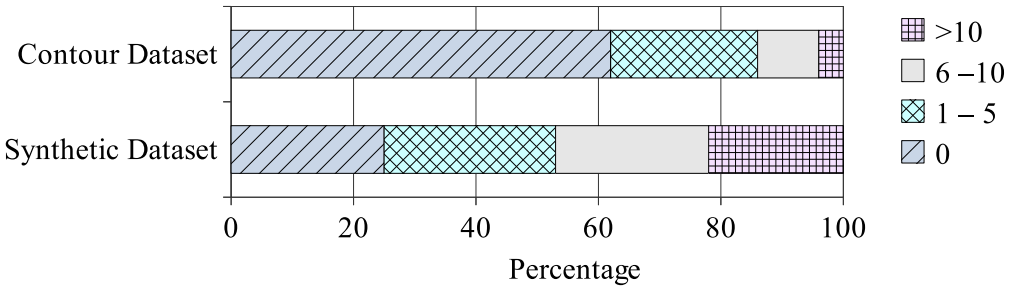


Figure 4.14: The distribution of the number of intersections per constraint.

For both cases, the total number of constraint-triangle intersections is around 6M. *Triangle* inserts constraints much slower when the constraints are long (one constraint intersects many triangles), taking 46 seconds for the synthetic dataset, while only 6 seconds for the contour dataset with mostly short constraints. On the other hand, due to our fine-grained approach, our program can easily process the synthetic dataset consisting of both long and short constraints with similar parallelism as the contour dataset with all very short constraints. Our running time for synthetic and contour dataset are 0.47 and 0.25 seconds respectively, achieving a significant speedup over the sequential approach.

4.5.3 Image vectorization

As mentioned earlier, the CDT can be used in many applications. Prasad and Skourikhine [PS06] present a framework for transforming a raster image into a vector image comprised of polygons that can be subsequently used in image analysis. Their algorithm consists of the following steps. First, edges are recognized using some standard edge detection algorithm. Second, contiguous edge pixel chains are extracted and an edge map consisting of straight line segments is constructed. Third, the CDT to the edge map (which is actually a PSLG) is computed. Finally, adjacent *trixels* (triangles) are merged based on certain grouping filters, and the connected components of the trixel grouping graph yield polygons that represent the image. Figure 4.15 shows a raster image and the result after computing the CDT of the edge map of it.

In practice, depending on the resolution of the image, the edge map might consist of hundreds of thousands of line segments (constraints) of different lengths, thus using our GPU-CDT can help speed up the computation. The constraints here are similar in nature to those in the contour datasets, so the performance of GPU-CDT here is similar to that for

(a) (b)

Figure 4.15: A raster image (top) and the CDT for its edge map (bottom).

contour datasets.

4.6 Summary

This chapter presents a new, efficient and robust parallel approach to construct the 2D constrained Delaunay triangulation on the GPU. Our approach is scalable, capable of maximizing the parallelism on the GPU. That has been shown in our experiment with both synthetic and real-world data. We have shown that our implementation can achieve two orders of magnitude better performance than the best CPU libraries available.

GPU-CDT gives extra weights to Case 1 and Case 2 of the constraint-triangle intersections to remove many constraint-triangle intersections. While *Triangle* uses sequential method to insert constraints. For each constraint to be inserted, the program discovers all intersected triangles, iterates them from left to right, and does edge-flip if necessary to remove inserted triangles. Extra weights technique may reduce the number of flips for inserting a particular constraint. From this view, introducing extra weights for Case 1 and Case 2 for sequential algorithm may also contribute to speed up the sequential algorithm.

Chapter 5

Conclusion

In this thesis, two algorithms termed as GPU-QM and GPU-CDT are proposed, which can improve the quality of the DT for a set of points, and compute CDT for a set of points and constraints, respectively. Both of these two algorithms are the first GPU algorithms proposed so far. According to our experiments for both synthetic and real-world data, our GPU algorithms are numerically robust and run up to two orders of magnitude faster than the fastest sequential algorithm. Furthermore, we obtain the first GPU mesh generator by integrating the GPU-QM, GPU-CDT algorithms with an existing work termed as GPU-DT, which can compute DT for a set of points using the GPU. Our mesh generator can compute digital Voronoi diagrams, exact Delaunay triangulation, constrained Delaunay triangulation, conforming Delaunay triangulation and high-quality triangle meshes in 2D space on the GPU.

5.1 Performance Analysis

Compared to the fastest CPU mesh generator, our GPU mesh generator is numerically robust and run much faster. However, the three main algorithms/functions of the mesh generator have different performance. Their speedups over the best sequential implementations have significant difference. For the GPU-DT algorithm, according to [QCT12, QCT13, CNGT14], it gains up to 7 times speedup. For the GPU-QM algorithm, we get 3 to 5.5 times speedup for different point distributions. As for the GPU-CDT algorithm, it runs up to two orders of magnitude faster than the best sequential implementation. One common reason for these differences on speedup is the different total work performed compared to the sequential implementation. The most time consuming operation of these three algorithms is edge-flip. So the running time depends on the number of flips performed. The GPU-DT algorithm does almost the same number of flips as the CPU mesh generator does. However, the GPU-QM algorithm does 2 times more edge-flips than the sequential method which is because we delete a number of redundant Steiner points to decrease the total number of Steiner points inserted in parallel. Since more edge-flips are needed, the speedup of GPU-QM over sequential implementation is less than GPU-DT's. As for the GPU-CDT al-

gorithm, we do 30% less edge flips than the sequential implementation, which is because we set priority to different kinds of flips and use one-step look-ahead strategy. Since the number of flips decreases, it is possible to gain better performance from GPU-CDT compared to the GPU-DT and GPU-QM algorithms. Noticed that GPU-CDT can gain significantly higher speedup than GPU-DT and GPU-QM, the main reason is the memory access. For all mesh generating algorithms, *Triangle* rearranges the data for points so that the cache is better utilized during point insertion, point location and edge-flip. However when constraints are introduced, rearranging data for constraints is difficult, at that time *Triangle* faces the same memory access problem as our GPU algorithms. Generally speaking, when we have memory access problem for our GPU algorithms, and *Triangle* has not, usually the speedup over *Triangle* is less than 10; when both of our algorithms and *Triangle* have memory access problem, we can expect great speedup over *Triangle* like GPU-CDT. In other words, our GPU mesh generator will benefit a lot from improvements in the data structure and memory access optimization in the future.

5.2 Future Work

Generating quality mesh for a PSLG is so inherently complex that any known algorithm is likely to be foiled by input data having small features, small angles, or complicated topologies. *Triangle* tried to propose some modifications which ensure that the program can always produce a valid mesh such that the quality of the mesh degrades gracefully as the input degrades. But the program can still fail on some difficult cases. Before our GPU mesh generator can generate quality mesh for a PSLG, there are many works that need to be prepared. In the future, we will try to solve this problem for a PSLG without small angles firstly. For a sequential method, algorithm of generating quality mesh for a set of points (including boundary refining) is as same as the the algorithm of generating quality mesh for a PSLG without small input angles. However on the GPU, these two algorithms could differ a lot. The main difficulty for handling constraints inside the triangulation is that it is very hard to be aware of all encroached constraints for a Steiner point. Many edges shorter than existing minimum local feather size may be generated so that the program can fall into an infinite loop. For the boundary constraints, in the GPU-QM algorithm, a simple and easy to implement method is proposed. But for the constraints inside the triangulation, the current method is not valid any more, and a more complicated method would be needed. When the input PSLG has small angles, the Delaunay refinement algorithm becomes more complicated. People would like to triangulate a domain without creating any small angles that aren't already present in the input. Unfortunately, no algorithm can achieve this goal for all triangulation domains [She01]. For this problem, we may adopt similar strategy as the one used in *Triangle*, i.e., try to protect some small angles, and do not split constraints/triangles if any smaller angles/shorter edge would be generated.

5.3 Summary

The algorithms described in this thesis are efficient and numerically robust. Besides two mesh generation algorithms and the mesh generator on the GPU, this thesis also provides useful information for devising other parallel and GPU algorithms to solve other computational geometry problems. In addition our mesh generator is freely available on the internet for download.

References

[Aur91] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[Ber95] Javier Bernal. Inserting line segments into triangulations and tetrahedralizations. Technical Report 5596, National Institute of Standards and Technology, 1995.

[Boi88] Jean-Daniel Boissonnat. Shape reconstruction from planar cross sections. *Computer Vision, Graphics, and Image Processing*, 44(1):1–29, 1988.

[Bow81] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, January 1981.

[CGA11] CGAL. Computational Geometry Algorithms Library. http://www.cgal.org, 2011.

[Che89a] L. Paul Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[Che89b] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report 89–983, Department of Computer Science, Cornell University, 1989.

[CN99] Nikos Chrisochoides and Demian Nave. Simultaneous mesh generation and partitioning for delaunay meshes. In *Proceedings of 8th International Meshing Roundtable*, pages 55–66, 1999.

[CNGT14] Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao, and Tiow-Seng Tan. A gpu accelerated algorithm for 3d delaunay triangulation. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 47–54, New York, NY, USA, 2014. ACM.

[CTMT10] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *I3D '10: Proc. ACM Symp. Interactive 3D Graphics and Games*, pages 83–90, New York, NY, USA, 2010. ACM.

[Dom04] Vid Domiter. Constrained delaunay triangulation using plane subdivision. In *Proceedings of the 8th central European seminar on computer graphics*, pages 105–110, 2004.

[Dwy87] Rex Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[EG01] Herbert Edelsbrunner and Damrong Guoy. Sink-insertion for mesh improvement. In *Proceedings of the seventeenth annual symposium on Computational geometry*, SCG '01, pages 115–123, New York, NY, USA, 2001. ACM.

[EM90] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, January 1990.

[FG06] Ian Fischer and Craig Gotsman. Fast approximation of high-order voronoi diagrams and distance transforms on the GPU. *J. Graphics Tools*, 11(4):39–60, 2006.

[For87] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For97] Steven Fortune. *Handbook of discrete and computational geometry*, chapter Voronoi diagrams and Delaunay triangulations, pages 377–388. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[Fre87] William H. Frey. Selective refinement: A new strategy for automatic node placement in graded triangular meshes. *International Journal for Numerical Methods in Engineering*, 24(11):2183–2200, 1987.

[GCTH13] Mingcen Gao, Thanh-Tung Cao, Tiow-Seng Tan, and Zhiyong Huang. Flip-flop: Convex hull construction via star-shaped polyhedron in 3d. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 45–54, New York, NY, USA, 2013. ACM.

[GKS92] Leonidas Guibas, Donald Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[Gol94] Christopher. M. Gold. A review of potential applications of voronoi methods in geomatics. In *Proceedings of the Canadian Conference on GIS*, pages 1647–1656, 1994.

[HDSB01] Kenneth H. Huebner, Donald L. Dewhirst, Douglas E. Smith, and Ted G. Byrom. *The Finite Element Method for Engineers*. Wiley, New York, NY, USA, 2001.

[HKL+99] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH '99*, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Kal10] Marcelo Kallmann. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 159–168, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[Law77] C. L. Lawson. Mathematical Software III; Software for C1 surface interpolation. In J. R. Rice, editor, *Software for C1 Surface Interpolation*, pages 161–194. Academic Press, New York, 1977.

[LC99] Rainald Lohner and Juan R. Cebral. Parallel advancing front grid generation. In *In International Meshing Roundtable, Sandia National Labs*, pages 67–74, 1999.

[LL86] D. Lee and A. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry*, 1:201–217, 1986.

[LT01] Zhiling Lan and Valerie E. Taylor. Dynamic load balancing for structured adaptive mesh refinement applications. In *Proc. of 30th International Conference on Parallel Processing*, pages 571–579, 2001.

[MPW03] Gary L. Miller, Steven E. Pav, and Noel Walkington. When and why ruppert's algorithm works. In *IMR*, pages 91–102, 2003.

[MTT97] Gary L. Miller, Dafna Talmor, and Shang-Hua Teng. Optimal good-aspect-ratio coarsening for unstructured meshes. In Michael E. Saks, editor, *SODA*, pages 538–547. ACM/SIAM, 1997.

[MTTW95] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 683–692, New York, NY, USA, 1995. ACM.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[NBP13] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on gpus. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM.

[NVI13] NVIDIA. CUDA C Programming Guide, 2013.

[PS85] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[PS06] Lakshman Prasad and Alexei N. Skourikhine. Vectorized image segmentation via trixel agglomeration. *Pattern Recognition*, 39:501–514, April 2006.

[QCT12] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2d constrained delaunay triangulation using the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 39–46, New York, NY, USA, 2012. ACM.

[QCT13] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2d constrained delaunay triangulation using the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 19(5):736–748, 2013.

[RTCS08] Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao, and Stephanus. Computing two-dimensional Delaunay triangulation using graphics hardware. In *I3D '08: Proc. Symp. Interactive 3D Graphics and Games*, pages 89–97, New York, NY, USA, 2008. ACM.

[Rup95] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, May 1995.

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. *FOCS '75*, pages 151–162, 1975.

[She96a] Jonathan Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming Lin and Dinesh Manocha, editors, *Applied Computational Geometry Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer Berlin / Heidelberg, 1996.

[She96b] Jonathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *in Proc. 12th Annu. ACM Sympos. Comput. Geom*, pages 141–150, 1996.

[She00] Jonathan Richard Shewchuk. Mesh generation for domains with small angles. In *Proceedings of the sixteenth annual symposium on Computational geometry*, SCG '00, pages 1–10, New York, NY, USA, 2000. ACM.

[She01] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22:1–3, 2001.

[SSD97] Peter Su and Robert L. Scot Drysdale. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry: Theory and Applications*, 7:361–385, April 1997.

[STU+02] Daniel A. Spielman, Shang-Hua Teng, Alper Üngör, I Shang-hua, and Teng Alper. Parallel delaunay refinement: Algorithms and analyses. In *Proceedings of 11th International Meshing Roundtable*, pages 205–217, 2002.

[STU04] DanielA. Spielman, Shang-hua Teng, and Alper Üngör. Parallel delaunay refinement with off-centers. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 812–819. Springer Berlin Heidelberg, 2004.

[Tre95] Lloyd A. Treinish. Visualization of scattered meteorological data. *IEEE Computer Graphics and Applications*, 15:20–26, 1995.

[Ü09] Alper Üngör. Off-centers: A new type of steiner points for computing size-optimal quality-guaranteed delaunay triangulations. *Comput. Geom. Theory Appl.*, 42(2):109–118, February 2009.

[Wat81] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, January 1981.

[Yap87] Chee Yap. An O(n log n) algorithm for the Voronoi diagram of a simple curve segments. *Discrete and Computational Geometry*, 2:365–393, 1987.

[Zal05] Borut Zalik. An efficient sweep-line delaunay triangulation algorithm. *Computer-Aided Design*, 37(10):1027 – 1038, 2005.