Computing Delaunay Refinement Using the GPU*

Zhenghai Chen [1] Meng Qi [2] Tiow-Seng Tan [3]

[1, 3]School of Computing, National University of Singapore
{chenzh | tants}@comp.nus.edu.sg
[2]School of Information Science and Engineering, Shandong Normal University
qimeng0914@gmail.com

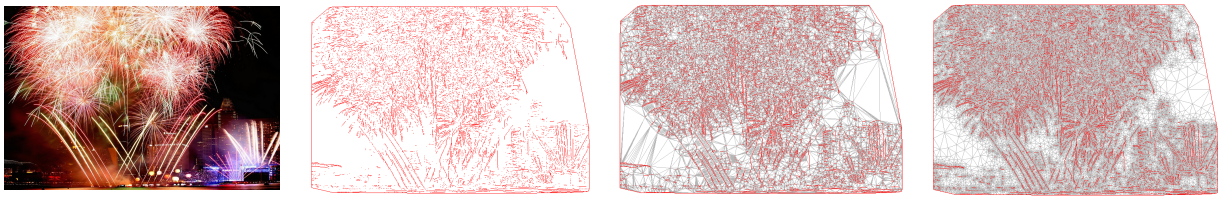


Figure 1: gQM *working on the PSLG obtained from a raster image.*

Abstract

We propose the first working GPU algorithm for the 2D Delaunay refinement problem. Our algorithm adds Steiner points to an input planar straight line graph (PSLG) to generate a constrained Delaunay mesh with triangles having no angle smaller than an input θ. It is shown to run from a few times to an order of magnitude faster than the well-known *Triangle* software, which is the fastest CPU Delaunay mesh generator. Our implementation handles degeneracy and is numerically robust. It is proven to terminate with finite output size for an input PSLG with no angle smaller than 60° and $\theta \leq 20.7^\circ$. In addition, we notice meshes generated by our algorithm are of similar sizes to that by *Triangle*, which has incorporated good consideration in keeping output small in size.

Keywords: GPGPU, Computational Geometry, Mesh Refinement, Finite Element Analysis

Concepts: •**Theory of computation** → **Computational geometry;** •**Computing methodologies** → **Graphics processors;**

1 Introduction

Many engineering and scientific applications, both interactive and non-interactive, such as finite element, interpolation, GIS, path planning, etc. work on meshes whose triangles meeting some quality measures. These measures are such as bound on triangle area, angle and edge length. Our problem of *Delaunay refinement* is as follows: for a given planar straight line graph (PSLG) $\mathcal{G}$ and an angle θ, compute a mesh that is also a constrained Delaunay triangulation (CDT) to cover the domain of $\mathcal{G}$ with triangles having no angle smaller than θ. We want such a *Delaunay refinement algorithm* to run fast and output a mesh having small number of triangles as this mesh is an input to other applications.

Specifically, the input $\mathcal{G}$ contains a point set P and a non-intersecting segment set S with endpoints from P. The output from a Delaunay refinement algorithm is a CDT $\mathcal{T}$ consisting of triangles incorporating all points in P with all segments of S appearing as unions of smaller segments. Points in the mesh but not in P are called *Steiner points*, added by the algorithm to create more triangles to refine the mesh so that all triangles in the mesh have no angle smaller than θ. These Steiner points appear within the convex hull of P, and some subdivide segments of S into smaller ones. We often refer to smaller segments also simply as *segments* just like segments of S. An *input angle* refers to an angle formed by two segments of S at their shared endpoint.

The *Triangle* software developed by Shewchuk [1996] is the fastest CPU Delaunay mesh generator in 2D. However, the increase in the input size of $\mathcal{G}$ can lead *Triangle* to run in hours, which is undesirable to many, especially interactive, applications. There is thus a need to explore what might be possible with the inexpensive, parallel computing power of GPU. At a glance, one may attempt refining a mesh by simply inserting Steiner points concurrently with many GPU threads to gain good speedup. This, however, is problematic as independent, concurrent insertions can lead to redundant ones and also to cast serious doubt on whether the approach can ever terminate in finite time.

This paper proposes a GPU Delaunay refinement algorithm, *gQM* (Section 4), and proves its termination for $\mathcal{G}$ with no input angle smaller than 60° and $\theta \leq 20.7^\circ$ (Section 5). Our implementation of *gQM* is shown to run from a few times to an order of magnitude faster than *Triangle* with output mesh of a similar size to that by *Triangle* (Section 6). We start the discussion with Section 2 on a review of important previous work, and then Section 3 on our considerations to design a good GPU algorithm. Section 7 concludes the paper.

*Website: http://www.comp.nus.edu.sg/~tants/gqm.html

I3D '17, February 25 - 27, 2017, San Francisco, CA, USA
ISBN: 978-1-4503-4886-7/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3023368.3023373

2 Literature Review

For a triangle, let ρ be the ratio of its circumradius to its shortest edge. Then, all angles of this triangle are bounded below by $\arcsin \frac{1}{2\rho}$. So, the Delaunay refinement problem can also be stated as finding a CDT with the ratio ρ of each of its triangles bounded away from an input constant B. In other words, we can use B and θ interchangeably in our discussion. A triangle is said to be *bad* if its $\rho > B$; otherwise *good*. Take for example, if we use $B = \sqrt{2}$, any triangle with an angle below 20.7° is bad.

The central question for the Delaunay refinement algorithm is to decide where should Steiner points be to improve the mesh quality while also guarantee its termination. A reasonable approach is to always insert a Steiner point as far away from other points as possible but still within the convex hull of P. This is to avoid the creation of short edges and thus triangles with large ρ. The two best-known algorithms are Ruppert's algorithm [1995] and Chew's algorithm [1993] in Sections 2.1 and 2.2, respectively. Section 2.3 discusses the mesh generator software, *Triangle* [Shewchuk 1996], and Section 2.4 outlines work in general parallel computation.

2.1 Ruppert's Algorithm

Ruppert's algorithm [1995] has theoretical guarantee for its termination for a PLSG with no input angle is acute and when $B \geq \sqrt{2}$. A point p is *visible* to a segment ab if there is a point $q \in ab$ forming pq that does not intersect any segments of S. The *diametral disk* of a segment is the smallest circular disk that encloses the segment. A segment ab is said to be *encroached by* a point p if p lies in the diametral disk of ab, and p is visible to ab. In such a case, we also say p is an *encroaching point* that *encroaches upon* the segment ab, and ab is an *encroached segment*.

Ruppert's algorithm refines mesh by repeating the following two steps (with the necessary edge flipping to reach a Delaunay triangulation) till there is no more bad triangle:

1. *Split encroached segments*. An encroached segment is split by inserting its midpoint. As a midpoint inserted may also encroach upon some other segments, we need to split these other segments subsequently till no more segments are encroached.
2. *Split bad triangles*. A bad triangle is to be split by inserting its circumcenter. However, if the circumcenter encroaches upon any segment, we reject the insertion and mark all segments it encroaches upon as encroached segments and split them according to the previous step.

2.2 Chew's Algorithm

Chew's second algorithm [1993], which we shall refer it simply as Chew's algorithm, also guarantees termination like Ruppert's algorithm but for a PSLG with no input angle smaller than 60° and $B \geq 1$. It produces CDT and not necessarily be a Delaunay triangulation as that by Ruppert's algorithm, and thus has in general smaller output mesh than that by Ruppert's.

We distinguish two types of Steiner points: *midpoint* on a segment (which is a part or whole of a segment of S) and *free point* not on any segment of S. In Chew's algorithm, midpoints once added are permanent to the mesh, but free points which are inserted to destroy bad triangles may be removed subsequently to make room for the insertion of other (permanent) midpoints. This is unlike Ruppert's algorithm which does not remove any Steiner point once inserted.

Chew's algorithm refines the mesh by running the following two steps iteratively till there is no more bad triangle:

1. *Split bad triangles*. For one bad triangle with circumcenter c, we walk from triangle to triangle in the mesh to locate the triangle containing c. In the process, if we encounter an edge of a triangle which is a segment, we stop the locating and go to Step 2 below to handle this segment. If not, we just insert c into the triangle containing c.
2. *Split segments and remove redundant points*. For each segment we encountered in Step 1, we remove free points who lie in the diametral disk of this segment. (These free points are considered as redundant ones for now.) Then we split this segment by inserting its midpoint.

2.3 Delaunay Mesh Generator: *Triangle*

Triangle by Shewchuk [1996] is the best-known CPU software for the Delaunay refinement problem. It actually unifies Ruppert's and Chew's algorithm into one framework of two iterative steps below that one can choose to run it either under Ruppert's or Chew's mode. Note that the unification is not fully as there is still an extra effort under Chew's mode when splitting encroached segments.

1. *Split encroached segments*. An encroached segment is split by inserting its midpoint. Under Chew's mode, all free points in the diametral disk of an encroached segment need to be removed before inserting the midpoint.
2. *Split bad triangles*. Consider the circumcenter of some bad triangle for insertion. If the circumcenter does not encroach upon any segment, insert it into the triangle enclosing the circumcenter; otherwise, we mark all segments it encroaches upon as encroached segments and split them in Step 1.

The above unification is possible by modifying the concept of encroachment in Ruppert's for use in Chew's mode. The *diametral lens* of a segment ab is the intersection of two disks whose centers lie on the bisector of ab, with one center on each side of ab. The intersection part is just big enough to enclose two isosceles triangles whose bases are ab and base angles equal to θ. Under Ruppert's mode, *Triangle* uses diametral disk as before, whereas under Chew's mode, diametral lens. That is, under Chew's mode, a segment ab is said to be encroached by a point p if p lies inside the diametral lens of the segment and p is visible to ab.

2.4 Parallel Algorithms

In parallel computation, there are many Delaunay refinement algorithms. One common strategy is as follows: at each iteration, find a set of independent Steiner points efficiently and then insert each into the domain by either Ruppert's or Chew's algorithm. Spielman et al. [2007] and Hudson et al. [2007] focus on time complexity analysis, while Lohner and Cebral [1999], Lan and Taylor [2001] and Chernikov and Chrisochoides [2005] are not comprehensive in their comparison to other.

A recent work by Nasre et al. [2013] claims that its GPU Delaunay refinement algorithm has high speedup. On the other hand, it does not handle general PSLG with segments. Chrisochoides et al. [2009] propose an algorithm to handle segments but it is designed for a cluster of computers.

3 Design Considerations of GPU Algorithm

We want to achieve a good speedup to produce mesh of a small output size comparable to that by a good sequential one (Section 3.1). In addition, we need to manage the usage of memory (Section 3.2) to avoid data conflict. The following discussion also alludes to

the fact that our proposed algorithm (in Section 4) is not a porting of a sequential one to GPU.

3.1 Speedup and Output Size

Two basic design principles to follow for a good speedup with GPU: regularized work and localized data. The latter is generally taken care of in the implementation to use registers and shared memory as much as possible instead of relying solely on global memory in GPU. As for the former, a straightforward naïve way is to assign one GPU thread to handle one insertion of a Steiner point. This is however problematic as Steiner points are dependent on each other. Thus, two Steiner points that are concurrently inserted may result in them forming a short edge and then later on creating more shorter edges still that the algorithm can no longer guarantee its termination. We discuss in Section 4.3 on how to compute points that are independent and can be inserted in parallel, and Section 4.4 on how to remove points that were inserted inappropriately.

Another regularized work to consider is to have a thread to handle an encroachment. In Ruppert's algorithm, we need to find all the segments that are encroached by a point. In Chew's algorithm, we have to search all points encroaching upon a segment. Both types of encroachment would be inefficient when done naïvely. This is because one segment may encroached by none or many points and likewise one point may encroach upon none or many segments. There are unbalance workload for GPU threads that can be questionable in gaining good speedup. We discuss in Section 4.2 on how to resolve this by inserting Steiner points first, then in Section 4.4 on how to find all encroached segments indirectly via constrained Delaunay property using highly parallel flipping operations.

Output from our algorithm is to be consumed by other applications, we thus want it small in size. Points must thus be added discreetly and with some priority among all threads, and not simply and arbitrarily by each thread to find a possible fix to remove a bad triangle. The latter can end with the undesirable consequence of creating lots of redundancy and short edges that causes the algorithm to run endlessly. In our algorithm, we choose to favor resolving a bad triangle with an edge of shortest length when deciding on a priority among bad triangles.

3.2 Memory

It is possible that multiple GPU threads access the same memory address and thus causes *data conflict*. In our algorithm, we use the commonly known technique of *marking competition* to resolve it. For every memory address that is going to be written, we assign it one marking slot in memory. Before the actual writing to those desired memory addresses, a thread marks its associated marking slots with a unique number using an atomic operation. After all threads finish marking and are synchronized, they check their corresponding marking slots to see if the numbers in the slots equal to their unique numbers. A thread wins its marking competition and thus can proceed to writing if all slots marked by the thread have value equal to its unique number.

In addition, as the size of a mesh increases dramatically during processing, we need to optimize the use of GPU memory. It is inefficient to assign temporal memory space for all elements (triangles, edges and points) in the mesh as not all are active during the different stages of the computation. To optimize, we collect all winners after marking competition using *stream compaction*, and then arrange contiguous memory for them to reduce the required memory space as well as the number of threads to launch for computation.

4 Our Proposed Algorithm: gQM

This section proposes our Delaunay refinement algorithm for GPU, called *gQM*. Its overall structure (Section 4.1) may look like that of *Triangle*, but they differ in three important aspects besides ours being parallel in nature. First, though *gQM* also has two modes, Ruppert's and Chew's, its two modes have the identical structure that we can interchange the use of one from the other by simply switching the definition of encroachment between diametral disk and diametral lens. Second, *gQM* adapts neatly the `FlipFlop` routine [Gao et al. 2013] to discover encroachment and to remove efficiently free points (Section 4.4). Third, *gQM* carefully recognizes and filters away points if inserted would result in their subsequent removal anyway. This can avoid many roll back that can sum to a significant loss in computing effort (Section 4.3). This last issue does not exist for the sequential approach.

Section 4.1 outlines the structure of *gQM*. Sections 4.2 and 4.3 discuss the two main steps of *gQM*, and Section 4.4 on the `FlipFlop` routine used by them to maintain a CDT and to remove redundant points. In the detailed presentation of the parts of our algorithm, we do not state explicitly initialization for the various data structures as these can easily be deduced.

4.1 Main Structure of *gQM*

Our GPU refinement algorithm, *gQM* is shown as Algorithm 1. It first computes the CDT $\mathcal{T}$ of the input PSLG $\mathcal{G}$ (Line 1) with the GPU algorithm of Qi et al. [2013]. Then, it goes into an iterative process of two steps to refine $\mathcal{T}$: (1) split encroached segments and remove redundant points, and (2) split bad triangles and remove redundant points. The algorithm uses the same data structure for mesh as that in *Triangle* where it can step from one triangle to its adjacent one, and update information of a neighboring triangle in constant time.

The first round of split encroached segments is from Line 2 to 8. This round starts with the CDT of $\mathcal{G}$. In the **for-loop** from Line 2 to 6, segments are possibly encroached by points in P only, since there are no Steiner points yet. Processing the encroachment marker list M_s, we use a stream compaction routine, `CollectList`, to collect active ones into a new list L_s (Line 7). Line 8 is the actual splitting of encroached segments with midpoints. Note that as there are no free points yet before this step, `SplitEncSegs` at Line 8 does only splitting with midpoints without having to remove any free points.

Subsequent iterative process of the mentioned two main steps is the **repeat-loop** from Line 9 to 21. It starts with the **for-loop** (Line 10 to 14) to mark bad triangles into the marker list M_t, and then collect those active ones (bad triangles) into L_t using again `CollectList` (Line 15). If there are bad triangles (i.e. L_t is non-empty), we do splitting of triangles (Line 17) with free points, then collecting again encroached segments (Line 18) passed on by `SplitTriangles` routine (Line 17) to further split these segments with midpoints in `SplitEncSegs` (Line 19).

4.2 SplitEncSegs: Split Encroached Segments

The `SplitEncSegs` routine is shown in Algorithm 2. It is to refine $\mathcal{T}$ till $\mathcal{T}$ has no encroached segments (i.e. $L_s = \emptyset$). We explain it as two parts of `RemoveEncPoints` (Line 2 to 9) and `InsertMidPoints` (Line 10 to 23) within the **while-loop** of Line 1 to 26. This **while-loop** is needed as new Steiner points added to $\mathcal{T}$ in `InsertMidPoints` can result in new encroachment and we thus need to identify (Line 24) and collect (Line 25) them to repeat the process.

Algorithm 1 *gQM*: the Delaunay Refinement Algorithm

Input: PSLG $\mathcal{G}$ with point set P and segment set S, and angle θ

Let M_s be the encroachment marker list for segments
L_s be the encroached segments list
M_t be the marker list for bad triangles
L_t be the bad triangles list

1: Construct the CDT $\mathcal{T}$ of $\mathcal{G}$
2: **for** each segment $s \in S$ **in parallel do**
3: **if** s is encroached by apexes of its incident triangles **then**
4: Mark s as active in M_s
5: **end if**
6: **end for**
7: `CollectList` (M_s, L_s)
8: `SplitEncSegs` $(\mathcal{T}, M_s, L_s)$
9: **repeat**
10: **for** each triangle $t \in \mathcal{T}$ **in parallel do**
11: **if** t is a bad triangle **then**
12: Mark t as active in M_t
13: **end if**
14: **end for**
15: `CollectList` (M_t, L_t)
16: **if** $L_t \neq \emptyset$ **then**
17: `SplitTriangles` $(\mathcal{T}, M_s, L_t)$
18: `CollectList` (M_s, L_s)
19: `SplitEncSegs` $(\mathcal{T}, M_s, L_s)$
20: **end if**
21: **until** L_t is $\emptyset$
22: **return** $\mathcal{T}$

Line 2 to 9: `RemoveEncPoints`

For every encroached segment, we need to remove all its encroaching points before inserting its midpoint. To do this, we adopt an iterative approach in the **while-loop** (Line 3 to 9) to checking apexes of triangles incident to segments for encroachment (Line 4 to 6), and then use `FlipFlop` (Line 7) to maintain $\mathcal{T}$ as a CDT to further discover points that encroach upon segments. When `SplitEncSegs` is invoked for the very first time from Algorithm 1, there are no free points yet and Line 5 does not remove any encroaching apexes.

Two notes are in order. First, our approach is superior to the naïve approach of explicitly finding out those encroaching points with one GPU thread handling one segment in a single go. As mentioned earlier in Section 3, the latter results in unbalance workload for its threads as the numbers of encroaching points vary significantly among segments. In addition, the expensive visibility computation possibly to be used in the latter has been avoided in our approach by the highly parallel operation of edge flipping in `FlipFlop`.

Second, recall Ruppert's algorithm in CPU does not need a process of removal as it simply rejects any insertion of Steiner point that can encroach upon segments. *gQM* running under Ruppert's mode does not know a point is an encroaching point without visibility computation, and thus cannot reject points upfront. It relies on first inserting such a point and then using `FlipFlop` to later discover it is indeed inserted inappropriately and thus scheduled to be removed.

Line 10 to 23: `InsertMidPoints`

`InsertMidPoints` aims to insert midpoints into encroached segments iteratively. It is possible that two encroached segments are in the same triangle but not both can be split in the same round due to data conflict. We use the marking competition mentioned in Section 3.2 to resolve it. The algorithm does the marking with the **for-loop** (Line 12 to 14) using an *atomicMin* (atomic minimum operation). Each triangle records the smallest (index of) s that needs

Algorithm 2 `SplitEncSegs` $(\mathcal{T}, M_s, L_s)$

Input: $\mathcal{T}$ is a CDT
M_s is the encroachment marker list for segments
L_s is the encroached segments list

Let L_d be the deletion segments list
M_r be the integer marker list for all triangles
M_p be the insertion marker list for segments
L_p be the insertion segments list

1: **while** $L_s \neq \emptyset$ **do**
2: Copy L_s to L_d
3: **while** $L_d \neq \emptyset$ **do**
4: **for** each segment $s \in L_d$ **in parallel do**
5: Remove encroaching points that are apexes (and free points) of triangles incident to s
6: **end for**
7: `FlipFlop` $(\mathcal{T}, M_s)$
8: `CollectList` (M_s, L_d)
9: **end while**
10: Copy L_s to L_p
11: **while** $L_p \neq \emptyset$ **do**
12: **for** each segment $s \in L_p$ **in parallel do**
13: Mark $M_r[t_i]$ and $M_r[t_j]$ with s using *atomicMin* where t_i, t_j are triangles incident to s
14: **end for**
15: **for** each segment $s \in L_p$ **in parallel do**
16: **if** s wins the marking in M_r **then** (i.e. $M_r[t_i]$ and $M_r[t_j]$ are both equal to s)
17: Insert the midpoint of s
18: Mark s as inactive in M_p
19: **end if**
20: **end for**
21: `UpdateNeighbors` $(\mathcal{T}, L_p)$
22: `CollectList` (M_p, L_p)
23: **end while**
24: `FlipFlop` $(\mathcal{T}, M_s)$
25: `CollectList` (M_s, L_s)
26: **end while**

it for splitting. Then the actual splitting (Line 17) is carried out with the **for-loop** (Line 15 to 20) for each segment that has successfully be recorded by all its (either one or two) incident triangle(s). Once an encroached segment has successfully be split (Line 17), it is marked as done (Line 18) and need not be considered for the subsequent round.

Splitting results in two or four new triangles. Each of these new triangles has to record its neighboring triangles in the mesh. But, these neighboring triangles may also be newly created in the same round and not available readily for use by other new triangles. To resolve this, we use a two-stage routine, `UpdateNeighbors`, to update neighborhood information (Line 21). First, the two or four triangles resulted by a segment split know the existence of each other, and they can simply update their neighbors in here among themselves. Second, each new triangle t_i resulted from splitting an old triangle t_j has one neighbor outside to be updated. Note that t_i and t_j have one same edge e, and t_j knows its neighbor t_k at e. Then, t_k is either the other needed neighbor of t_i if it is not split, or through the new triangles resulted from t_k, we identify the needed one as the neighbor for t_i.

Line 22 again collects encroached segments needed to be split but not done within this round into L_p for consideration in the next round of the **while-loop** (Line 11 to 23). At the end, this loop completes splitting of all encroached segments found so far.

4.3 SplitTriangles: Split Bad Triangles

The `SplitTriangles` routine is presented in Algorithm 3. We explain it as three parts: `LocateTriContainCirc` (Line 1 to 11) to locate triangles containing circumcenters of bad triangles, `ComputeCavity` (Line 12 to 20) to compute cavities of points with possibility to be inserted, and `InsertFreePoint` (Line 21 to 29) to add free points to split bad triangles (and other good ones).

Line 1 to 11: `LocateTriContainCirc`

Let one GPU thread handle one bad triangle t with circumcenter c to locate the triangle t_c that contains c. Specifically, a thread computes c first (Line 2), and then walks from t in the direction of c, by stepping from one triangle to its adjacent one, until it finds t_c (Line 3). (Note that the position of a circumcenter can be on a segment, in a triangle or outside the mesh. For the former two cases, we process as per normal. As for the last case, we record it as a midpoint on the convex hull edge of the last triangle the thread encounters.) With t_c, the thread does the marking competition with atomicMin to write information about t into $M_r[t_c]$ (Line 4). We use the shortest edge length of t for the atomicMin. For each winner t in the marking competition, its thread records t and the corresponding c and t_c into M_p (Line 7 to 9) for collection by the stream compaction routine, `CollectList` (Line 11). These t_c with c are then be considered for the insertion of a free point at c subsequently.

Algorithm 3 `SplitTriangles`$(\mathcal{T}, M_s, L_t)$

Input: $\mathcal{T}$ is a CDT
M_s is the encroachment marker list for segments
L_t is the bad triangles list

Let M_r be the integer marker list for all triangles
M_p be the point location winner marker list
L_p be the point location winner list
M_c be the cavity winner marker list
L_c be the cavity marking winner list

1: **for** each bad triangle $t \in L_t$ **in parallel do**
2: Compute the circumcenter c of t
3: Walk starting from t to record the triangle t_c containing c
4: Mark $M_r[t_c]$ with the information of t using atomicMin
5: **end for**
6: **for** each bad triangle $t \in L_t$ **in parallel do**
7: **if** t wins the marking in M_r **then**
8: Mark t as active, and store (t, c, t_c) in M_p
9: **end if**
10: **end for**
11: `CollectList`(M_p, L_p)
12: **for** each $(t, c, t_c) \in L_p$ **in parallel do**
13: Mark $M_r[t']$ for each triangles t' in the cavity of c with the shortest edge of t using atomicMin
14: **end for**
15: **for** each $(t, c, t_c) \in L_p$ **in parallel do**
16: **if** t wins the marking in M_r **then**
17: Mark t_c as active, and store (c, t_c) in M_c
18: **end if**
19: **end for**
20: `CollectList`(M_c, L_c)
21: **for** each $(c, t_c) \in L_c$ **in parallel do**
22: **if** c is on a segment s **then**
23: Mark s as encroached in M_s
24: **else**
25: Insert the circumcenter c into $\mathcal{T}$
26: **end if**
27: **end for**
28: `UpdateNeighbors`$(\mathcal{T}, L_p)$
29: `FlipFlop`$(\mathcal{T}, M_s)$

Line 12 to 20: `ComputeCavity`

It is not always the case that each t_c with c as found by `LocateTriContainCirc` (Line 11) can be split by inserting c into $\mathcal{T}$. This is because two such c when inserted concurrently may form a very short edge in $\mathcal{T}$ (so as to maintain $\mathcal{T}$ a CDT) that we no longer can guarantee the termination of the algorithm. So, any one of these two c is actually a redundant one that should not be inserted. We want to first identify as many points that do not affect each other, i.e. independent from each other if inserted concurrently, to then perform the insertion.

Two points p_i and p_j are said to be *Delaunay independent* if they do not form an edge after their concurrent insertions into $\mathcal{T}$ while maintaining $\mathcal{T}$ a CDT. Now the problem becomes how to compute as large a set of points that each pair of points is Delaunay independent. To solve this, we adapt the concept of cavity on Delaunay triangulation from [George and Borouchakin 1998] to CDT. The *cavity* of a point p, not a vertex of $\mathcal{T}$, is the region covered by the union of triangles in $\mathcal{T}$ whose circumcircles enclose p and whose vertices are all visible from p. When two points are Delaunay independent, their cavities do not overlap.

The converse is, however, not true since there can be a case where the cavities of p_i and p_j do not overlap and yet they are not Delaunay independent. In fact, we need to augment to the non-overlapping condition with the condition that an edge shared by the boundaries of two cavities is either a segment or it is a Delaunay edge, then p_i and p_j are indeed Delaunay independent. For our algorithm, we actually use the converse without the augmentation for its simplicity and for there is a good chance that the two points are Delaunay independent still. This caveat is taken care of by `FlipFlop` subsequently to remove those points inserted inappropriately.

Specifically, the `ComputeCavity` routine finds a good set of (not necessarily) Delaunay independent point set in two stages using marking competition. The first stage (Line 12 to 14) uses one GPU thread to take each circumcenter c of t to mark out, by stepping from the first triangle t_c (containing c) to its neighboring triangles recursively, whose circumcircles enclose c, with the shortest edge length of t using atomicMin. The second stage (Line 15 to 19) uses one GPU thread for each circumcenter c to go through again triangles that belong to the cavity of c to see whether all are marked with the shortest edge length of t. If so, c wins the marking competition and can be a point to be inserted into $\mathcal{T}$ subsequently.

In the implementation, we can choose to stop the recursion at some depth in identifying cavity, and take further risk of not discovering some points that are not Delaunay independent from each other. Should a point subsequently inserted be a redundant one, it will still be recognized and removed from the mesh by `FlipFlop`. In fact, we learn from our experimental results that `ComputeCavity` is relatively inexpensive (even though it uses atomicMin as there are few accesses to same marking locations concurrently) as compared to `FlipFlop` that needs a number of flips and updating of neighborhood information to remove points. So, we can afford from our experience to set the recursion to discover cavity with a deep depth of, say 50.

Line 21 to 29: `InsertFreePoint`

For each point c found by `ComputeCavity` (Line 20), we use a GPU thread to handle it in two different ways. If c lies on a segment, we mark the segment as encroached instead of inserting c (Line 23). Otherwise, c lies on an edge that is not a part of any segment of S, or in some triangle. For the former, we split the edge with c creating new triangles incident to the new smaller edges in $\mathcal{T}$; for the latter,

we split this triangle into three smaller ones with c (Line 25). After inserting the points, we need to update neighborhood information among triangles using `UpdateNeighbors` (Line 28), which has been explained in Section 4.2, and to perform `FlipFlop` (Line 29) to maintain $\mathcal{T}$ as a CDT with no redundant points.

4.4 FlipFlop: Flip to Delaunay; flop to remove point

`FlipFlop`$(\mathcal{T}, M_s)$ maintains the constrained Delaunay property of $\mathcal{T}$ and removes redundant points added into $\mathcal{T}$. It is run as a GPU routine where flipping operations are applied in parallel to the relevant parts of $\mathcal{T}$ till we obtain a CDT that incorporates only the midpoints and necessary free points. Some points are identified as *redundant* because they encroach upon segments, and the others during flipping when we are about to form an edge with two free points inserted in the same iteration. In the latter case, instead of doing flip, the free point with a larger shortest edge is marked as redundant and will be removed by subsequent flips (or rather, flop). The next two paragraphs provide further details.

Moving towards a constrained Delaunay triangulation, we perform in-circle test on two incident triangles. When they fail the test, we use *Delaunay flip*, which is a 2-2 flip that converts two non-locally constrained Delaunay triangles abc and dcb sharing bc (not a segment) to the alternative ones, abd and dca, that are locally constrained Delaunay. Note that each triangle can participate in at most one flipping of its one edge. So, we once again use the now familiar marking competition to mark for each edge its two incident triangles to decide whether the edge can be flipped. (Likewise, when updating neighborhood information of triangles, we need to do it in two stages.)

To remove a free point v, the theory of flip-flop in [Gao et al. 2013] guarantees this can be done through a series of 2-2 flips to reduce the degree of v to 3 and then followed by a 3-1 flip to remove v; see Figure 2. Each of this flip is also termed a *flop* or *non-Delaunay flip* as it is moving away from being a (constrained) Delaunay triangulation.

Two notes are in order. First, for a free point that is removed because it encroached upon a segment, this segment is recorded into M_s for subsequent processing. Second, removing a point actually involves a number of rounds of flipping. Although all are simple operations, their sum can still be significant that we should avoid removing points if ever possible. This is also the reason that we use the less expensive `ComputeCavity` to filter away as many as possible non-Delaunay independent points from being inserted rather than simply inserting all points found in `LocateTriContainCirc` to then removing redundant ones by `FlipFlop`.

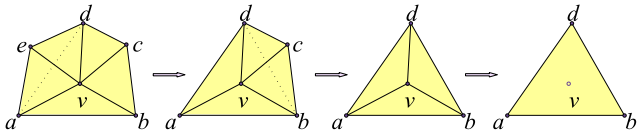


Figure 2: *To remove a free point v, we flip ve and vc to reduce the degree of v to 3. Then, v is removed by a 3-1 flip.*

5 Proof of Termination

A more involved proof as compared to the sequential case is needed to resolve the implication of points inserted in the same iteration by *gQM*. We define *iteration* in the Algorithm 1 as follows. Iteration 0 contains all operations from Line 2 to 8. Then, iteration 1 and so on corresponds to each subsequent refinement loop from Line 10 to 20. At the end of iteration 0, there are only input points and midpoints, and then from iteration 1 onwards, there are also free points inserted to split triangles. We use $B \geq \sqrt{2}$ and require input PSLG $\mathcal{G}$ has no input angles smaller than 60°.

We need some definitions from [Shewchuk 2001]. A *mesh point* is either an input point or a point successfully inserted into $\mathcal{T}$ in an iteration. A *rejected point* is a redundant point that has been removed because it encroached upon some segments. The *insertion radius* of a point v inserted in iteration i is denoted as $r_i(v)$. It is equal to the length of the shortest edge connected to v immediately after v is inserted into the mesh. There are three cases of $r_i(v)$ depending on the type of point v is:

(i) If v is an input point, then $r_0(v)$ is the Euclidean distance between v and the nearest input point visible from v.

(ii) If v is a free point and is inserted because of a bad triangle t, then $r_i(v)$ is the radius of the circumcircle of t. This is because free points inserted in the same iteration cannot be connected to each other as enforced by `FlipFlop`, and all points connected to v immediately after insertion must be points existed before this iteration. As the mesh is a CDT, the radius of the circumcircle of t has the shortest length.

(iii) If v is a midpoint of an encroached segment mn, then there are two situations. If mn is encroached by some mesh points, then $r_i(v)$ is the distance between v and the nearest encroaching point. If there is no encroaching point (so its insertion is due to some rejected point), then $r_i(v)$ is the radius of the diametral circle of mn.

For each point v, we define a point (who is "responsible" for the insertion of v) as its *parent* p as follows:

(i) If v is an input point, p is null.

(ii) If v is a free point at the circumcenter of a bad triangle t, then p is one of the endpoints of the shortest edge of t. If both endpoints of this edge are input points, choose one arbitrarily. If not, pick the one most recently inserted before v. Note that the two endpoints must not both be free points inserted in the same iteration as enforced by `FlipFlop`.

(iii) If v is a midpoint on an encroached segment mn, then p is the nearest encroaching point that is either a mesh point or a rejected point.

For a PSLG $\mathcal{G}$ with point set P and segment set S, the *local feature size* $lfs(v)$ at a point v (not necessarily in P) is the radius of the smallest disk centered at v that intersects (i) two points in P, or (ii) one point u in P and one segment of S not incident to u, or (iii) two non-incident segments of S. We count v as one point intersected if it is a point in P or a point on a segment of S. Let $lfs_{\min}$ be the smallest among all $lfs(v)$ for v in P or on any segment of S.

Fact 1. For a point v inserted in iteration i, we have $r_i(v) \geq lfs_{\min}$.

Proof. We use mathematical induction in the iteration number. Let p be the parent of v.

Base case. At iteration $i = 0$, the mesh only contains input points and midpoints. If v is an input point, there is another input point at distance $r_0(v)$ from v. So $r_0(v) \geq lfs(v) \geq lfs_{\min}$. On the other hand, if v is a midpoint on an encroached segment mn, there are two cases:

(i) If p is an input point, or a midpoint on a segment not incident to mn, then by definition, the circle centered at v with

$|vp|$ as radius must intersect p or the segment containing p, so $r_0(v) \geq lfs(v) \geq lfs_{\min}$.

(ii) If v and p are both midpoints and lie on incident segments separated by an angle $60^\circ \leq \alpha < 90^\circ$. To find the lower bound of $r_0(v)$, we imagine that $r_0(p)$ and α are fixed, then $r_0(v) = |vp|$ is minimized by making mn as short as possible. Because p is inside the diametral circle, the minimum of $r_0(v)$ is achieved when $|mn| = 2r_0(v)$; see Figure 3(a). From basic trigonometry, $|mn| = 2r_0(v) \geq r_0(p)/\cos\alpha$, then $r_0(v) \geq r_0(p)/(2\cos\alpha) \geq r_0(p)$ as $\cos\alpha < 1/2$.

Now, for p, we can find its parent u to form a chain of midpoints, $v \to p \to u$. By repeating this argument (for p, and then the parent of p and so on), we arrive eventually at a chain with $v' \to p' \to u'$ such that u' is a midpoint fulfilling case (i) above. So, we have $r_0(p') \geq lfs_{\min}$ by case (i), and we thus also have $r_0(v') \geq r_0(p') \geq lfs_{\min}$ using the resulting inequality in the previous paragraph. Then, tracing the argument along the chains back to our $v \to p \to u$ here, we have $r_0(p) \geq r_0(u) \geq lfs_{\min}$, and thus $r_0(v) \geq r_0(p) \geq lfs_{\min}$.

Inductive step. We assume that the statement holds for $0 \leq i \leq k$, and we want to prove it also holds for $i = k + 1$.

If v is a free point (i.e. circumcenter) inserted due to a bad triangle t, then its parent p is the point inserted in iteration j for $j \leq k$. Hence, the length of the shortest edge of t is at least $r_j(p)$. We have $r_{k+1}(v) \geq B \cdot r_j(p)$ since t is bad, and $r_j(p) \geq lfs_{\min}$ by the induction hypothesis. Thus, we have $r_{k+1}(v) \geq lfs_{\min}$ because $B > 1$.

If v is a midpoint on an encroached segment mn, there are two cases:

(i) If p is a rejected point, which must be rejected in iteration $k + 1$, then p lies inside the diametral circle of mn. Because the mesh is a CDT, the circumcircle of some bad triangle created in earlier iteration with center at p cannot contain the endpoints of mn. Hence, $r_{k+1}(v) \geq r_{k+1}(p)/\sqrt{2}$; see Figure 3(b). The parent of p, denoted by u, must be a point inserted in iteration j for $j \leq k$, so we have $r_{k+1}(p) \geq B \cdot r_j(u)$, and we have $r_j(u) \geq lfs_{\min}$ by the induction hypothesis. Thus, $r_{k+1}(v) \geq r_{k+1}(p)/\sqrt{2} \geq B \cdot r_j(u)/\sqrt{2} \geq r_j(u) \geq lfs_{\min}$ for $B \geq \sqrt{2}$.

(ii) If p is a midpoint (must be inserted in iteration $k + 1$), then we consider two situations. If p lies on a segment that is non-incident to mn, then the circle centered at v with $|vp|$ as radius must intersect p (and the segment incident to p). So $r_{k+1}(v) \geq lfs(v) \geq lfs_{\min}$.

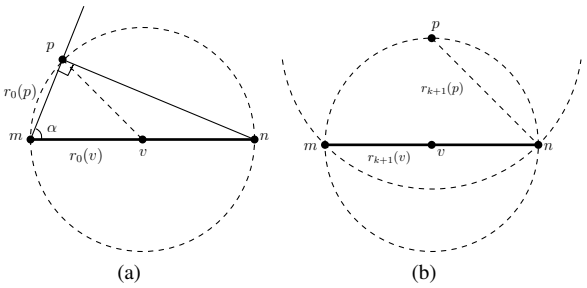


Figure 3: *(a) In iteration 0, if p and v are both midpoints on two incident segments and p is on the diametral circle of mn, $r_0(v)$ reaches its lower bound as shown. (b) In iteration $k + 1$, if v is the midpoint and p is a rejected point, then $r_{k+1}(v) \geq r_{k+1}(p)/\sqrt{2}$.*

If p lies on a segment incident to mn, then just like how we prove for the **Base case (ii)**, we have $r_{k+1}(v) \geq r_{k+1}(p)$. Also, for p, we have its parent u and so on, to form eventually a chain $v' \to p' \to u'$ that u' is either a midpoint lying on a non-incident segment or a rejected point. Then, the argument in the above paragraphs in (i) and (ii) applies. Hence, tracing the argument along the chains back to our $v \to p \to u$, we have $r_{k+1}(v) \geq r_{k+1}(p) \geq r_{k+1}(u) \geq lfs_{\min}$.

So the statement holds for all iterations. □

Fact 2. The GPU Delaunay refinement algorithm *gQM* terminates with no bad triangles in the output mesh.

Proof. By Fact 1, our algorithm never introduces an edge shorter than $lfs_{\min}$ in any iteration. If we remove some redundant points in an iteration, we still insert at least one midpoint. So every iteration inserts at least one point. Because the underlying space of the mesh is finite for the insertion of points, the algorithm must terminate. When it terminates, there is thus no bad triangle in the mesh. □

6 Experimental Results

We implement *gQM* by using the CUDA programming model of NVIDIA [Nickolls et al. 2008]. All experiments are conducted on a PC with an Intel i7-6700 3.4GHz CPU, 16GB of DDR3 RAM and a GTX980 Ti graphics card with 6GB of video memory. All computations in GPU are done with double precision. We compare the results of *gQM* with the best CPU Delaunay mesh generator, *Triangle*. For compiling *Triangle* and *gQM*, we turn on all optimization flags. In measuring computing time, we exclude the time of computing the CDT of the input PSLG. We use simulation of simplicity [Edelsbrunner and Mücke 1990] to deal with degenerate cases, and exact predicate of Shewchuk [1997] to handle robustness.

We apply *gQM* on a few real world datasets such as Figure 1 and some contour maps freely available at http://www.ga.gov.au/. But to better understand the behavior of *gQM* compared to *Triangle*, we need a good spread of different types of data to stress test them. We thus generate synthetic PSLGs with points of the following distributions: uniform, Gaussian, disk (where points lie randomly in a disk) and circle (where points lie randomly in the ring formed by two circles with slightly different radii). Segments are generated randomly after the points are in place. We ensure no two segments intersect or form an input angle less than 5°. See Figure 4 for sample outputs of the four distributions. We note that although our proof of termination in the previous section requires input angles no less than 60°, *gQM* adapts the approach of *Triangle* to still terminate by specially handling triangles in the vicinity of a small input angle. Consequently, the mesh can contain some bad triangles inherited from small input angles.

The number of input points in our PSLGs ranges between 50K and 100K, while the ratio γ of the number of segments to the number of input points ranges from 0.1 to 0.5. The larger input sizes here have stretched the available memory limit of our GPU as the output meshes can have over tens of millions mesh points and triangles. We run *Triangle* and *gQM* under both the Ruppert's and Chew's mode. Besides experimenting with *Triangle* and *gQM* with input $B > \sqrt{2}$ that guarantees termination, we also experiment with $B < \sqrt{2}$. In particular, we present here results with $\theta = 15^\circ$ (where $B > \sqrt{2}$), $\theta = 20^\circ$ (where $B \approx \sqrt{2}$), and $\theta = 25^\circ$ (where $B < \sqrt{2}$). In the following, we summarize from many experimental results to present those important observations. Section 6.1 discusses running time, and Section 6.2 mesh quality.

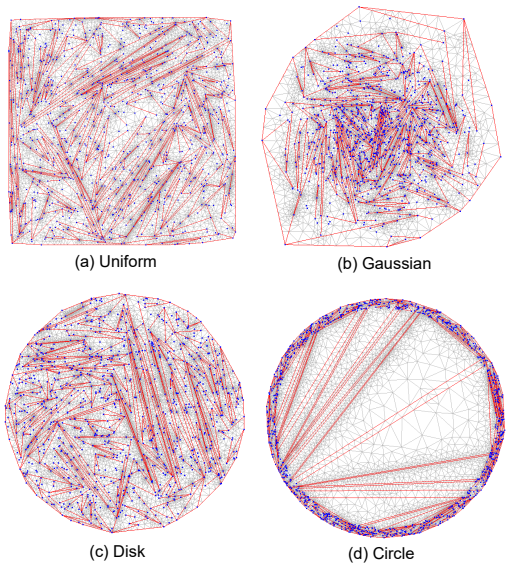


Figure 4: *Sample outputs of* gQM *of the four distributions. Input PSLGs are shown with red segments and blue points.*

6.1 Running time comparison

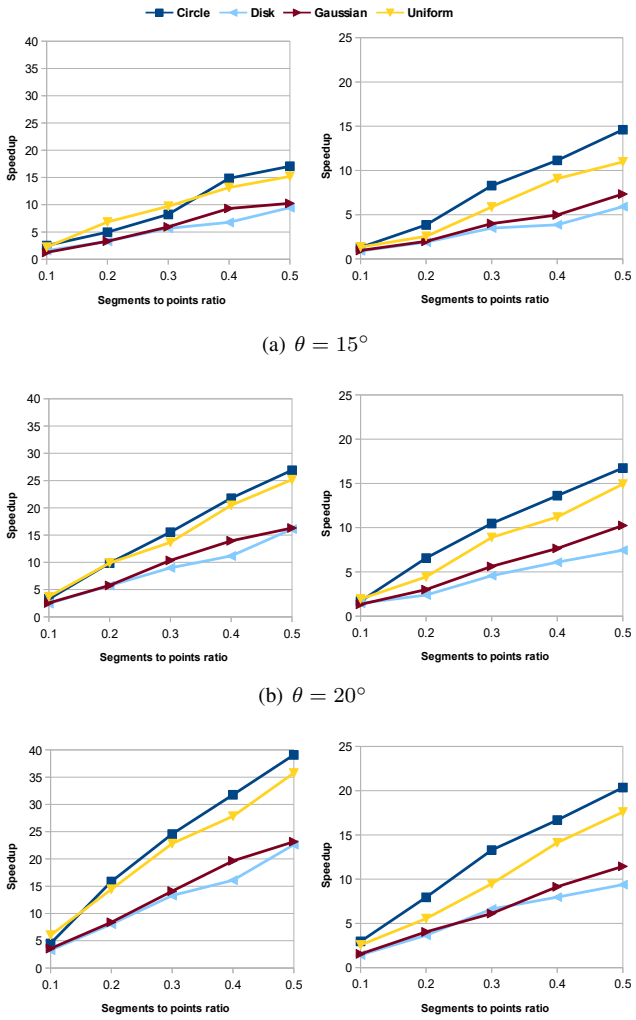
From an input PSLG, we first need to compute its CDT and then perform the refinement. We are interested in mainly the running time for the refinement step done by both *Triangle* and *gQM*. Note that for *gQM*, the running time includes the time for computation as well as the time needed to transfer data between CPU and GPU. Table 1 shows the running time of *Triangle* and *gQM* with 100K input points of the uniform distribution for different values of θ, and for both Ruppert's and Chew's mode. We observe that *Triangle* needs at time close to an hour to complete computation, whereas *gQM* generally can complete within a few minutes.

		Triangle					*gQM*				
θ	γ	0.1	0.2	0.3	0.4	0.5	0.1	0.2	0.3	0.4	0.5
15°	Ruppert	0.4	1.9	4.3	7.8	11.5	0.2	0.3	0.4	0.6	0.8
	Chew	0.4	2.0	4.6	8.4	12.8	0.3	0.8	0.8	0.9	1.2
20°	Ruppert	0.7	3.5	7.5	14.1	20.6	0.2	0.4	0.5	0.7	0.8
	Chew	0.7	3.6	8.5	16.1	24.9	0.4	0.8	1.0	1.4	1.8
25°	Ruppert	1.3	6.7	14.4	27.3	39.8	0.2	0.5	0.6	1.0	1.1
	Chew	1.2	6.2	14.7	28.6	44.1	0.5	1.1	1.5	2.0	2.5

Table 1: *Running time (in minute) by* Triangle *and* gQM *with 100K input points of the uniform distribution.*

The trends of the speedup of *gQM* over *Triangle* for all the distributions with 100K input points and different minimum allowable angles θ are shown in Figure 5. The speedup seems increasing with larger γ for all the distributions. It is also increasing with larger minimum allowable angles θ. This is within expectation as larger γ means more segments available for processing concurrently, and this is advantage to *gQM* running in GPU. We note, under Ruppert's mode, *gQM* reaches more than 10 times speedup for all the distributions with the maximum of 40 on the circle, whereas under Chew's, more than 5 times for all with the maximum of 20 on the circle.

For *gQM*, its Ruppert's mode performs better than Chew's in running time and in its speedup to *Triangle*. One reason we reckon is that *gQM* can remove free points inserted in only the same iteration under Ruppert's mode, but in all previous iterations under Chew's mode. Removal is rather undesirable as computing time was already wasted in the insertion and now again in rolling back.



(a) $\theta = 15^\circ$

(b) $\theta = 20^\circ$

(c) $\theta = 25^\circ$

Figure 5: *The speedup of* gQM *over* Triangle *for 100K input points under Ruppert's (left), and Chew's mode (right) with different minimum allowable angles θ.*

6.2 Mesh quality comparison

Table 2 shows the numbers of points in the output meshes created by *Triangle* and *gQM* for input points of the uniform distribution. To better appreciate this, we have Figure 6 to show the amount of extra output points (in percentage) in *gQM* as compared to *Triangle*. For $\theta \leq 20^\circ$, their output sizes differ by only a small percentage, and thus insignificant. This means *gQM* has a good speedup and a reasonable output size. The reasonable output may be attributed to our way of handling priority in favor of triangles with shorter edges for splitting. On the other hand, when $\theta = 25^\circ$, *gQM* needs significantly more mesh points than *Triangle*, though *gQM* still offers good speedup. We reckon here *gQM* becomes aggressive in creat-

ing short edges in parallel that causes the creation of many more mesh points. On the other hand, we see one encouraging fact in Figure 6: when γ increases, the amount of extra mesh points (in percentage) in *gQM* actually decreases for all the distributions.

		Triangle					*gQM*				
θ	γ	0.1	0.2	0.3	0.4	0.5	0.1	0.2	0.3	0.4	0.5
15°											
Ruppert		1.53	3.62	5.42	7.16	8.76	1.50	3.52	5.27	6.96	8.52
Chew		0.78	1.62	2.34	3.02	3.65	0.78	1.61	2.31	2.98	3.59
20°											
Ruppert		1.77	4.12	6.14	8.09	9.87	1.74	3.98	5.91	7.76	9.47
Chew		1.13	2.43	3.52	4.57	5.53	1.13	2.37	3.41	4.40	5.31
25°											
Ruppert		2.34	5.38	7.95	10.42	12.67	2.52	5.67	8.30	10.83	13.15
Chew		1.73	3.79	5.51	7.16	8.67	1.86	3.99	5.74	7.42	8.94

Table 2: *Output points (in million) created by* Triangle *and* gQM *with 100K input points of the uniform distribution.*

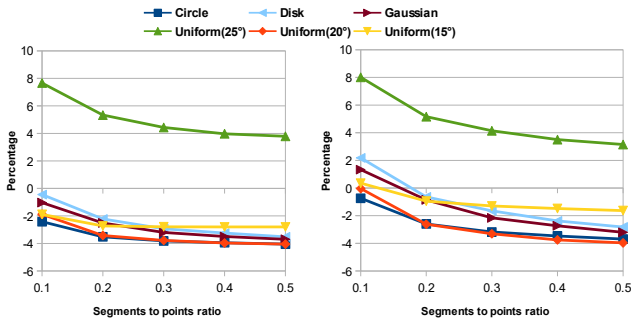


Figure 6: *The amount of extra points (in percentage) in the output mesh of* gQM *compared to* Triangle *with 100K input points. Negative percentage means less instead of extra points. Ruppert's mode is on the left, while Chew's on the right.*

7 Concluding Remarks

This paper proposes *gQM*, the first working Delaunay refinement algorithm on GPU with PSLG as input. It fully unifies Ruppert's and Chew's algorithm. It employs an efficient cavity computation to filter away points that are bound to be removed if inserted, and it incorporates the parallel `FlipFlop` neatly to assist in removing point. *gQM* runs from a few times to an order of magnitude faster than *Triangle*. Moreover, *gQM* is guaranteed to terminate for PSLG with no input angle smaller than 60°.

We list here two possible improvements to our current implementation of *gQM*. Firstly, for $B \geq \sqrt{2}$ which *gQM* can guarantee termination, its output size of mesh is comparable to that by *Triangle*. On the other hand, when $B < \sqrt{2}$ with say $\theta = 25^\circ$ (which *gQM* still can terminate often in practice though not guaranteed), its output size is less desirable than that by *Triangle*. We hope to narrow this gap. Secondly, *gQM* working on real world dataset outputs quality mesh comparable to that by *Triangle*. But, it has relatively smaller speedup compared to working with our synthetic data. We are investigating the possible reasons and ways to further improve the performance of *gQM*.

References

CHERNIKOV, A. N., AND CHRISOCHOIDES, N. P. 2005. Parallel 2D graded guaranteed quality Delaunay mesh refinement. In *Proceedings of the 14th International Meshing Roundtable*, Springer Berlin Heidelberg, 505–517.

CHEW, L. P. 1993. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ninth annual symposium on Computational geometry*, ACM, 274–280.

CHRISOCHOIDES, N., CHERNIKOV, A., FEDOROV, A., KOT, A., LINARDAKIS, L., AND FOTEINOS, P. 2009. *Towards Exascale Parallel Delaunay Mesh Generation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

EDELSBRUNNER, H., AND MÜCKE, E. P. 1990. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph. 9*, 1 (Jan.), 66–104.

GAO, M., CAO, T.-T., TAN, T.-S., AND HUANG, Z. 2013. Flipflop: Convex hull construction via star-shaped polyhedron in 3d. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '13, 45–54.

GEORGE, P.-L., AND BOROUCHAKIN, H. 1998. *Delaunay Triangulation and Meshing: Application to Finite Elements*. Hermes Science Publications.

HUDSON, B., MILLER, G. L., AND PHILLIPS, T. 2007. Sparse parallel Delaunay mesh refinement. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM, New York, NY, USA, SPAA '07, 339–347.

LAN, Z., AND TAYLOR, V. E. 2001. Dynamic load balancing for structured adaptive mesh refinement applications. In *Proc. of 30th International Conference on Parallel Processing*, 571–579.

LOHNER, R., AND CEBRAL, J. R. 1999. Parallel advancing front grid generation. In *International Meshing Roundtable, Sandia National Labs*, 67–74.

NASRE, R., BURTSCHER, M., AND PINGALI, K. 2013. Morph algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, New York, NY, USA, PPoPP '13, 147–156.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar.), 40–53.

QI, M., CAO, T.-T., AND TAN, T.-S. 2013. Computing 2D constrained Delaunay triangulation using the GPU. *IEEE Trans Vis Comput Graph 19*, 5 (May), 736–748.

RUPPERT, J. 1995. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms 18*, 3 (May), 548–585.

SHEWCHUK, J. 1996. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *Applied Computational Geometry Towards Geometric Engineering*, M. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 203–222.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18*, 305–368.

SHEWCHUK, J. R. 2001. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications 22*, 21–74.

SPIELMAN, D. A., TENG, S.-H., AND ÜNGÖR, A. 2007. Parallel Delaunay refinement: Algorithms and analyses. *International Journal of Computational Geometry & Applications 17*, 1, 1–30.