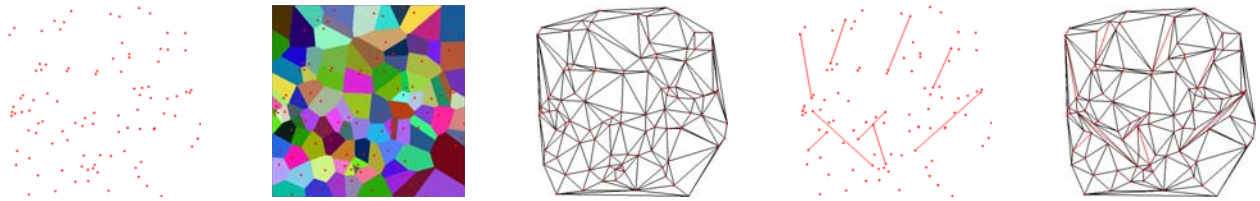
Computing 2D Constrained Delaunay Triangulation Using Graphics Hardware

Meng Qi, Thanh-Tung Cao and Tiow-Seng Tan*
National University of Singapore



(a) point set S (b) Digital VD (c) DT of S (d) PSLG $\mathcal{G}$ (e) CDT of $\mathcal{G}$

Figure 1: *Delaunay triangulation (DT) and constrained Delaunay triangulation (CDT).*

Abstract

This paper presents a novel approach, termed *GPU-CDT*, to compute the *constrained Delaunay triangulation* (CDT) for a *planar straight line graph* (PSLG), consisting of points and edges, using the graphics processing unit (GPU). Although there are many algorithms for constructing the 2D CDT using the CPU, there has been no known prior approach using the parallel computing power of the GPU efficiently. For the special case of the CDT problem with PSLGs consisting of just points, which is the normal Delaunay triangulation problem, a hybrid approach has recently been proposed that uses the GPU together with the CPU to partially speed up the computation. Our GPU-CDT works for such special case too, but the whole computation is fully accelerated by the GPU. Our implementation using the CUDA programming model on nVidia GPUs is numerically robust and runs several times faster than any existing CPU algorithms as well as the prior GPU-CPU hybrid approach. This result is reflected in our experiment with both randomly generated PSLGs and real world GIS data, with millions of points and edges.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—[Geometric algorithms]; I.3.1 [Computer Graphics]: Hardware Architecture—[Graphics processors]

Keywords: GPGPU, Computational Geometry, Voronoi Diagram

1 Introduction

Delaunay triangulation (DT) is one of the most important geometric structures in computational geometry. Due to its nice property of avoiding long, skinny triangles, the DT has many practical applications in different fields. In Geographical Information System (GIS), one way to model terrain at the sampled points is to interpolate the data based on the DT [Gold 1994a; Gold 1994b]. In path planning, the DT can be used to compute the Euclidean minimum spanning tree of a set of points, because every geometric minimum spanning tree must be a subgraph of the DT [Preparata and Shamos 1985]. DT is also often used to build quality meshes for the finite element analysis [Huebner et al. 2001].

Constrained Delaunay triangulation (CDT) is a direct extension of the Delaunay triangulation where some edges in the output are enforced before hand [Chew 1989]; these edges are referred to as *constraints*; see Figure 1. Given a set S of n *points* (or *sites*) in the 2D plane and a set of non-crossing constraints, collectively called a *planar straight line graph* (PSLG), the CDT is a triangulation of S having all the constraints included, while being as close to the DT of S as possible. Constraints occur naturally in many applications. For example, in path planning, they are obstacles; in GIS, boundaries between cities; in surface reconstruction, contours in the slices of the body's skull; in modeling, characteristic curves [Boissonnat 1988; Treinish 1995; Kallmann et al. 2003].

Recently, the Graphics processing unit (GPU) with its enormous parallel computation power has been used for general computation in many disciplines, including computational geometry. Early works include computing the digital Voronoi diagram, a structure that is closely related to the DT [Hoff et al. 1999; Fischer and Gotsman 2006]. These works also mention the possibility of obtaining the latter from the former straightforwardly. However, we note 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. [2008] presented a serious attempt in realizing such approach on GPUs. Their approach, however is a hybrid one, where parallel computation is only used in first part of the computation, leaving the rest to a sequential CPU algorithm. As for the CDT problem, there is no known efficient GPU algorithm as far as we know. In another view, the DT problem, as well as the CDT problem, does not present itself readily to parallel computation. Specifically, it is not clear how to adapt traditional complex parallel algorithms, such as the divide-and-conquer approach, to best use the computing power of the GPU.

Our main contribution here is a novel algorithm, termed *GPU-CDT*, to compute the CDT for a given PSLG, fully parallelized for the GPU. Our experiment shows that our implementation of GPU-CDT using the CUDA programming model [NVI 2010] is robust and efficient. Comparing to *Triangle* [Shewchuk 1996], the fastest sequential mesh generation software, as well as to the hybrid approach of Rong et al. [2008], GPU-CDT runs up to an order of magnitude faster.

The paper is organized as follows. Section 2 introduces some basic definitions and reviews the previous works on both DT and CDT. Section 3 presents our GPU approach to the DT problem, and Section 4 then extends it to handle constraints. Section 5 describes our experimental results, and Section 6 concludes the paper.

*{qimeng | caothanh | tants}@comp.nus.edu.sg

2 Preliminaries

In this section we first introduce some important definitions and properties before stating some related works on computing the DT and CDT. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the Euclidean space $\Re^2$.

Definition 1 (Voronoi diagram) *For each point $p \in S$, the* Voronoi region $\mathcal{R}(p)$ *of p is the set of points in $\Re^2$ that are closer (in Euclidean metric) to p than to any other points in S. The* Voronoi diagram $\mathcal{V}(S)$ *of S is the space partition induced by the Voronoi regions of the points in S. 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*.*

Definition 2 (Digital Voronoi diagram) *In the digital space, we only consider the set of integer grid points. Consider a 2D grid of size $m \times m$. We say that a grid point x is* colored *by the point $p \in S$ if $x \in \mathcal{R}(p)$. In case x is of equal distance from two points p_i and p_j with $i < j$, we color x by p_i. The set of all colored grid points forms the* digital Voronoi diagram $\mathcal{D}(S)$ *of S (see Figure 1(b)). This coloring procedure is referred to as the* Euclidean coloring*.*

Definition 3 (Planar straight line graph) *A* planar straight line graph *(PSLG) $\mathcal{G} = (S, E)$ is a plane graph with vertex set S and edge set E; see Figure 1(d).*

Definition 4 (Delaunay triangulation) *A* triangulation *of S is a PSLG $\mathcal{T} = (S, E)$ such that $|E|$ is maximal. An edge $ab \in E$ is said to satisfy the* empty circle property *with respect to S if there exists a circle passing through a and b such that points in S are not inside the circle. A triangulation $\mathcal{T}$ of S is a* Delaunay triangulation *(DT) if each edge of $\mathcal{T}$ satisfies the empty circle property with respect to S. When points in S are in general positions, i.e. no four or more points are co-circular, the DT of S is unique. Figure 1(c) shows an example of the DT.*

Definition 5 (Constrained Delaunay triangulation) *Two points a and b of a PSLG $\mathcal{G} = (S, E)$ are* visible *from each other if the (open) line segment ab does not intersect any other edge in E. A triangulation $\mathcal{T} = (S, E')$ is a* constrained Delaunay triangulation *(CDT) of $\mathcal{G}$ if $E \subseteq E'$ and each edge $ab \in E'$ is either in E or it satisfies the empty circle property with respect to those points of S visible from both a and b. If $E = \emptyset$, then the CDT is exactly the same as the DT. Figure 1(e) shows an example of the CDT.*

There are many algorithms developed for the CPU to efficiently compute the DT [Aurenhammer 1991; Fortune 1997; Su and Scot Drysdale 1997]. All these algorithms in general follow one of the three well-known algorithmic paradigms: divide-and-conquer [Shamos and Hoey 1975; Dwyer 1987], sweep-line [Fortune 1987] and incremental insertion [Guibas et al. 1992].

The algorithms for constructing the CDT can be grouped into two categories: (a) *Processing points and constraints simultaneously*: Chew [1989] shows that the CDT can be built in optimal $O(n \log n)$ time using a divide and conquer approach to partition the problem into smaller CDT problems within vertical strips. (b) *Processing points and constraints separately*: Since CDT is a generalization of DT with the notion of constraints [Lee and Lin 1986; Kallmann et al. 2003; Bernal 1995; Shewchuk 1996], we can first construct DT of the given point set, then insert each constraint one by one to remove triangles pierced through by the constraint, and re-triangulate regions due to the removal of triangles to include the constraint and to maintain the empty circle property.

3 Computing DT on GPU

The basic idea of our GPU algorithm is to derive from the digital Voronoi diagram $\mathcal{D}(S)$ of the input set of points S an approximation of the DT for transforming into the needed DT. Specifically, the algorithm consists of the following phases:

Phase 1. Digital Voronoi diagram construction Map the input points into a texture and compute the digital Voronoi diagram. If more than one point is mapped to a same pixel, keep just one and treat the others 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.

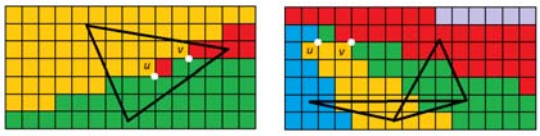
Phase 5. Edge flipping Verify the empty circle property for each edge in the triangulation, and perform *edge flipping* if necessary.

We adopt the triangulation data structure used by Shewchuk [1996] in our computation. A list of triangles is stored in a pre-allocated array of size no more than $2|S|$, each one has the indices of three other triangles edge adjacent to it. Each vertex in S also has a linked list of triangles incident to it.

3.1 Phase 1: Digital Voronoi diagram construction

In this step, we first translate and then scale the point set such that its bounding box fits inside our texture of size $m \times m$. Then we map each input point p to the grid point (or pixel) closest to p in a texture. In case several input points are mapped to a same grid point, only one among them is recorded. Those that are not recorded become *missing points* and will be inserted into the triangulation in a later phase. With the texture containing the points or *seeds*, we use the Parallel banding algorithm (PBA) of Cao et al. [2010a] to compute the digital Voronoi diagram of these seeds. Each Voronoi region of a seed is represented by the color of the seed.

Cao et al. [2010b] shows that by dualizing the output obtained by the Standard flooding, one gets a valid geometrical triangulation. The output of Standard flooding is very close to that of PBA except that each region of a color is connected. We adopt PBA as it is a much faster process than the Standard flooding, but the dual of its output is not a valid geometrical triangulation. In particular, a digital Voronoi region obtained by PBA can be disconnected and its dual thus can have duplicated and intersecting triangles (see Figure 2). Pixels in a disconnected Voronoi region is classified as: *bulk* (pixels) which are path-connected to its seed, and *debris* (pixels) otherwise. We want to get the same output as the Standard flooding to use the theory in [Cao et al. 2010b]. First, we amend the output from PBA by removing all the debris. This is carried out by identifying (in parallel) all pixels whose adjacent pixels closer to their seeds do not have the same color. This process may need a few passes as each pass may remove just some of the debris of a Voronoi region. After that, we can use a flooding algorithm to recolor the debris. The detailed algorithm and its proof of correctness is discussed in Appendix A. The number of debris is usually very small, so the cost of this correction step is insignificant and our approach with PBA is thus a much better one than directly using the Standard flooding.



(a) duplicated triangles (b) intersecting triangles

Figure 2: *Portions of digital Voronoi digrams showing (a) corners u and v that generate the same triangle; and (b) corners u and v generate two intersecting triangles.*

3.2 Phase 2: Triangulation construction

In this phase, we want to dualize the result from the previous phase. A corner shared by up to 4 pixels is incident to 1 to 4 different colors. For a corner with 3 colors, we add one triangle, and for one with 4 colors, we add two (non-intersecting) triangles into the triangulation. This phase can be done in parallel by processing each row of the texture independently. A thread handling one row of the texture first counts the number of triangles to be generated for that row. Then we use the parallel prefix sum primitive to identify the offset in the triangle list from which each thread can start adding triangles.

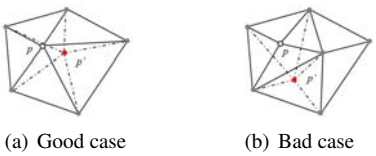
As the digital Voronoi diagram is truncated within the texture, its dual does not always produce a complete triangulation with a convex boundary. We need an additional step of traversing along the boundary of the texture, using the idea similar to the Graham's scan algorithm [1972], to identify triangles whose Voronoi vertices fall outside the texture to add into the end of the triangle list. This additional step is performed in the CPU as it is a simple task and can be done concurrently while the GPU is populating the triangle list.

3.3 Phase 3: Shifting

Points are shifted away from their original positions in Phase 1. We now work on moving them back to their original positions in two stages: first, we scale and translate the computed triangulation to near the original coordinates; second, we then shift the points to their original coordinates. The first stage is done with careful numerical accuracy consideration so that the triangulation remains valid after scaling and translation; see Apendix B. The second stage is described next.

In shifting a point, we assume all its neighboring points remain static. A point to be shifted falls in one of the two categories: *good case* and *bad case*. The former is the case when a point can be shifted to its original position without causing any intersection of triangles (of its fan) with other triangles. Otherwise, it is the latter case; see Figure 3. As expected due to the very small shifting distance, majority of the cases in practice are good cases.

To achieve uniform computation while shifting points in parallel, we handle the two mentioned categories separately. A point p can be shifted as a good case if the orientation of all triangles incident to p remain the same after moving p to its original location. This implies that we cannot shift two points sharing the same edge concurrently. We use multiple passes in which no two adjacent points are shifted in the same pass. This can be ensured by giving priority to points with smaller index; see Algorithm 1. In this algorithm, initially all points are unchecked. To guarantee the robustness of the computation, all the orientation tests are done using Shewchuk's robust orientation predicate [Shewchuk 1996].



(a) Good case (b) Bad case

Figure 3: *Shifting of point from p to p' in (a) good and (b) bad case. Dashed lines indicate the new triangulation after shifting.*

Algorithm 1 Shifting good cases and recording bad cases

```
repeat
  for each unchecked point p_i do in parallel
    if all neighbors p_j of p_i, j < i, has been checked then
      mark p_i as checked
      if shifting p_i is a good case then
        update the coordinate of p_i to its original one
      else
        mark p_i as a bad case
      end if
    end if
  end for
until all points have been checked
```

After all points have been checked and most of them shifted, we have also marked the other points not shifted as bad cases (though they are not necessary so due to possibly new locations of points in its triangle fan). We handle these points by simply deleting them from the triangulation, and mark them as missing points, just as in Phase 1, for later processing. Notice we also need the above-mentioned tie breaker to avoid deleting in parallel two points incident to a same edge. To delete one point, we mark all triangles in its fan as deleted. We then use the ear-cutting method [Highnam 1982] to re-triangulate the resulting star-shape hole. Notice that the number of new triangles is no more than the number of deleted triangles, thus we can use the deleted slots in the triangle list to store new triangles, without any racing memory access during parallel computation.

3.4 Phase 4: Missing points insertion

This phase starts with a triangulation $\mathcal{T}$, and we want to insert into $\mathcal{T}$ in parallel, points of S that are not yet part of $\mathcal{T}$. These points are those missing points identified in Phase 1 and Phase 3. The insertion for each missing point p_i starts by efficiently identifying the triangle(s) in $\mathcal{T}$ that contains p_i, or has an edge passing through p_i. For a missing point obtained from Phase 1, we can start searching from a triangle incident to the point of S mapped to the same grid point as p_i; as for one from Phase 3, we can start searching from a triangle incident to a point p_j of S sharing an edge with p_i before p_i was deleted. During the processing of p_i, if p_j is not yet inserted, we delay the insertion of p_i to a later iteration.

With the triangle(s) containing p_i found, the insertion of p_i splits the single triangle enclosing p_i into three, or the two triangles passing through p_i into four. As the insertion of points is to be carried out in parallel, we need to prevent modification done to some same triangles by the insertions of different points. This can be achieved by first marking the triangles to be modified by p_i with the vertex index i using *atomicMin* operation of CUDA: when a triangle is to be modified by different missing points, only the one with the smallest index should be allowed. The insertion of p_i is carried out only when it managed to retain its marks on the triangle(s) it wants to modified. It is clear that the missing point with smallest index can always be inserted, so the algorithm terminates. This also means that many rounds as shown in the **repeat-until** loop of Algorithm 2

may potentially be needed to complete all the insertions. In practice, the worst does not happen; for the case where many points fall into a same triangle, after the first few insertions, we have missing points quickly distributed across many newly created triangles.

Algorithm 2 Inserting missing points

```
repeat
  for each not yet inserted missing point p_i do in parallel
    mark the triangle(s) containing p_i with i using atomicMin
  end for
  for each not yet inserted missing point p_i do in parallel
    if it successfully marked the triangles for modification then
      insert p_i into the triangulation
    else
      record the triangle found for p_i for its latter point location
    end if
  end for
until all missing points have been inserted
```

Note first that the simple way of marking without using *atomicMin* does not work as there can be a live lock situation when various modifications each needs the locking of two adjacent triangles while these triangles together form a cycle. Note second that there is one caveat in the above algorithm. A point p_i to be inserted can fall outside $\mathcal{T}$, and many new triangles are to be created to include p_i into the new $\mathcal{T}$. We adapt the standard trick of enclosing all points of S by a pre-defined polygonal boundary, that will be removed at the end of the computation.

3.5 Phase 5: Edge flipping

In this phase we verify the empty circle property for each edge $\mathcal{T}$ in parallel. For an edge ab of the triangle abc, we only have to check if the opposite point of the other triangle, say adb, adjacent to ab, if any, is inside the circumcircle of abc. If so, an edge flipping is performed to replace the two triangles abc and adb by triangles acd and cbd. As in the insertion of missing points in parallel, one thread performing an edge flip needs to modify two triangles and can thus conflict with other threads. We use the same multiple passes strategy: each thread first marks the two triangles it needs to modify with its index using the *atomicMin* operation, and only if the thread successfully marks both triangles needed for modification can proceed with the edge flipping. After the completion of this phase, the result is indeed the Delaunay triangulation of the input point set S.

A few notes are in order. First, it is possible for an edge to be flipped several times if its neighbors are changed. Second, in each pass, we only need to check an edge if it was never checked before, or it needed to be flipped but could not in the previous pass due to conflict with other edge flipping, or its neighboring triangles were modified. Third, to perform the in-circle test robustly, we use the robust predicate introduced by Shewchuk [1996]. We perform only the fast check of the predicate in the GPU, while making down the edges with numerically inaccurate in-circle tests (the almost cocircular cases). After all possible flipping are performed and the result is transferred back to the CPU memory, we perform another pass of in-circle test on the very few marked edges in CPU using the robust predicate. After all flipping are done, if any, the result we obtain is the desired Delaunay triangulation.

4 Computing CDT on GPU

To compute the CDT with the GPU, our algorithm processes points and constraints separately 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 Flip edges (non-constraint) if necessary (Phase 5).

Step 1 and Step 3 are as discussed, but Step 2 needs consideration to achieve good parallelism. The naïve approach of having one thread to handle one constraint is not ideal: each constraint can intersect many triangles in $\mathcal{T}$, and each insertion thus has many smaller tasks that can possibly be done in parallel but not explored. Also, the difference in the number of triangles intersected by different constraints results in unbalanced work loads. Worst still, two different threads handling two constraints may intersect some common triangles and the threads cannot proceed independently. Our proposed Step 2 is given in Algorithm 3 with the *outer loop* and the *inner loop*. The idea is to identify constraint-triangle intersections with the outer loop, and uses edge flipping to remove these intersections in the inner loop, all in parallel using multiple passes.

Algorithm 3 Inserting constraints into the triangulation

```
repeat    /* outer loop */
  for each constraint c_i do in parallel
    mark all triangles intersected by c_i with i using atomicMin
  end for
  repeat    /* inner loop : details in Algorithm 4 */
    do edge flipping to remove intersections to constraints
  until no edge that is flippable
until all constraints are inserted
```

4.1 Outer loop: Find constraint-triangle intersections

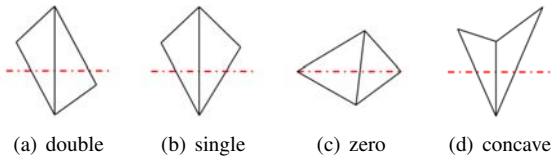
The goal here is to find out, for each triangle in $\mathcal{T}$, the index of a constraint intersecting it, if any. Let $c_i = ab$ be the i^{th} constraint, we go through the triangle fan of a to identify the triangle A intersected by c_i. If c_i is an edge of A, no further processing is needed. Otherwise, from A we start walking along the constraint towards b, visiting all triangles intersected by c_i. For each triangle T found, we record in T the result of the orientation test of each vertex of T against ab, and the index i with *atomicMin* operation. The former information is used to move to the next triangle along the path, while the later's usage is explained in our proof of correctness.

We note that the issue of unbalanced work loads still remains here. However, this is the less time consuming part, while the computation in the more time consuming part, the inner loop, is much more uniform.

4.2 Inner loop: Remove intersections

The inner loop of Algorithm 3 is to perform flipping of edges to reduce eventually the number of intersections between constraints and triangles to zero. A *triangle pair* is a pair of triangles incident to a common edge. See Figure 4. With respect to a constraint, the triangle pair is a *double intersection*, *single intersection*, or *zero intersection*, respectively, if flipping their common edge results in two, one, or zero, respectively, intersections between the new triangle pair and the constraint. A triangle pair is *concave* if flipping is not allowed as its underlying space is a concave quadrilateral.

The difficulty of performing the flipping in parallel is that flipping a triangle pair might not reduce the number of intersections, and restricting to only flipping triangle pairs that are zero and single intersection will reduce the number flippings in parallel. It might also be possible that there are no zero or single intersection that such an approach would be *stuck* with triangle pairs that are double



(a) double (b) single (c) zero (d) concave

Figure 4: *Configurations of triangle pair intersecting constraint (drawn in red dashed line).*

intersection or concave. Our approach overcomes all these by "one-step look-ahead". 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 of A in that chain. See Figure5; the triangle pair (A, C) is *flippable* in one of the following cases:

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

Case 2 (A, C) and (B, A) are both double intersections, and flipping (A, C) would result in B with B's new next triangle forming a single intersection. The later condition is equivalent to having the union of B, A and C as a convex polygon.

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

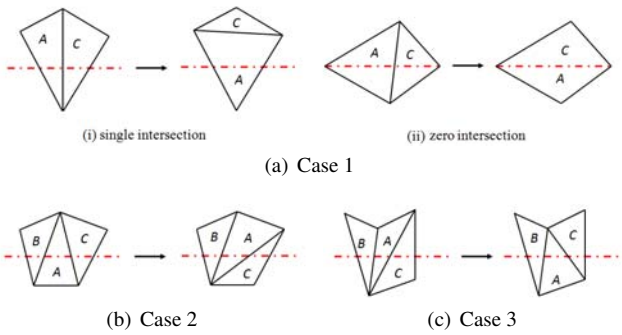
The detail of the inner loop of Algorithm 3 is given in three parallel **for** loop in Algorithm 4.

Algorithm 4 Parallel processing of triangle-constraint intersections

```
repeat
  for each triangle A intersecting a constraint do in parallel
    if C is also marked by the same constraint then
      determine the configuration of (A, C)
    end if
  end for
  for each triangle A intersecting a constraint do in parallel
    if (A, C) is flippable then
      label A, C (and B for Case 2) using atomicMin
    end if
  end for
  for each triangle A intersecting a constraint do in parallel
    if A, C (and B for Case 2) are labeled the same then
      flip (A, C) and update the links between the new
      triangles and their neighbors
    end if
  end for
until no more edge is flippable
```

The labeling with *atomicMin* in the second **for** loop is to prevent conflict in the actual flipping during the third **for** loop. In the labeling, we favor Case 1. Also, when (A, C) is Case 2 or Case 3, B is also labeled in the second **for** loop, and the label is checked in the the third **for** loop so that the "one-step look-ahead" is achieved.

In practice, the **repeat-until** loop of the Algorithm 4 should just repeats the inner loop a few times per each outer loop instead of repeating until *no more edge is flippable*. This is because as the algorithm progresses, there are drastic reduction of the number of flippable cases, and the parallelizm thus reduces. So, by switching to the outer loop after a few (say 5 to 10) rounds of inner loop, the algorithm can discover more flippable cases to improve the parallelizm and performance of the inner loop without compromising on the correctness of the algorithm as proven in the next subsection.



(a) Case 1

(b) Case 2 (c) Case 3

Figure 5: *Flipping consideration of triangle pairs involving A. Constraint is drawn in red dashed line intersecting triangles from left to right, i.e. triangle B comes before A, and C after A.*

4.3 Proof of correctness and complexity analysis

We show here that Algorithm 3 indeed 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 *atomicMin* operation, we ensure that all triangles intersecting c_i are marked with i. It thus suffices to prove the following:

Lemma 1 *The inner loop can always successfully inserts a constraint into the triangulation.*

Proof. Consider the chain of triangles intersecting c_i from a to b. Among these triangles, if there is one or more triangle pairs that are single or zero intersection, then the claim is true as the marking favor each of these cases and flipping is indeed carried out to reduce one intersection with c_i.

Otherwise, we consider the chain of triangles intersecting c_i are only double intersection or concave. We argue in the following that there still exists a triangle pair (A, C) mong them that is flippable, and each flipping is a step closer to removing intersection of triangles with c_i.

If we would remove all triangles intersecting c_i, a polygonal hole is created with vertices $p_1, p_2, \ldots$ as its upper part and $q_1, q_2, \ldots$ as its lower part, excluding a and b; see Figure 6. Any polygon has an ear, so 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 in the chain. We exclude a and b themselves to be q_k. Let A be the previous triangle of C, B be the previous of A. We have $A = q_kp_jp_{j-1}$ since if it would be $q_kp_jq_{k-1}$, then (A, C) should have been a single intersection pair. The triangle pair (A, C) is a double

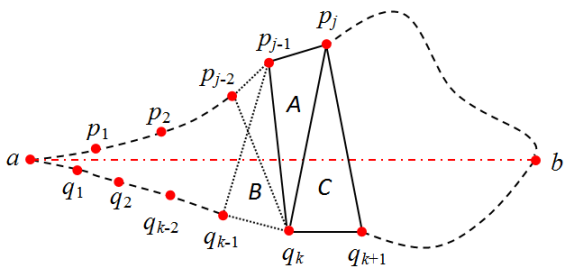


Figure 6: *Consideration when triangle pairs intersecting constraint $c_i = ab$ are either double intersection or concave.*

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 (A, C) is flippable: If $B = q_kp_{j-1}q_{k-1}$ then (B, A) is a double intersection; the union of triangles B, A, C is a convex polygon as needed in Case 2. If $B = q_kp_{j-1}p_{j-2}$ then (B, A) is a concave pair; because $p_{j-2}p_{j-1}q_{k+1}q_k$ is convex by the choice of q_k, triangles B, A, C fulfils Case 3. As long as there are one triangle pair that is flippable, the marking in the second **for** loop will successfully mark one triangle pair for flipping (using the index of A while also favoring flipping of Case 1), and flipping is indeed performed for each pass of the inner loop.

We next show that flipping does not continue forever. Let us assign to each triangle pair the value of 0, 1 and 2 according to its being zero/single intersection, double intersection and concave, respectively. Then, we have a base 3 number, N, to record the configurations of the chain of triangles intersecting c_i. A flipping due to Case 1 deletes a digit in N, Case 2 turns digits 11 into 01, and Case 3 turns digits 21 into 11. In other words, each flipping decreases the value of N. Since N is finite, our algorithm clearly terminates by inserting a constraint as needed by the claim. □

The above concludes that our proposed algorithm computes correctly the CDT of an input PSLG. It also indirectly shows that no flip is *wasteful* with the following bound on the number of flips per constraint:

Lemma 2 *The total number of flips performed by the inner loop to add a constraint is $O(n^2)$ where n is the number of triangles intersecting the constraint.*

Proof. Flipping due to Case 1 cannot be done more than n times since each flip reduces an intersection. Flipping due to Case 2 immediately gives raise to a flipping of Case 1 (with highest priority), and thus cannot be done more than n times too.

Flipping due to Case 3 either eliminates a concave configuration, or pushes the concave configuration towards one end of the constraint. There are initially $O(n)$ concave configurations, and flipping due to Case 1 (or Case 2) can introduce at most two (or one) concave configurations, thus the number of flips in total is $O(n^2)$, as required. □

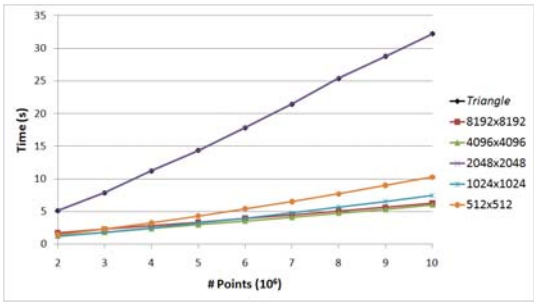
5 Experiment Result

We have developed and tested our algorithm on a PC with an Intel i7 930 2.8Ghz, 6GB DDR3 memory and a single NVIDIA GTX 460 Fermi graphics card with 2GB of video memory. Our program called GPU-CDT is built under Microsoft Visual C++.NET 2008, and complied with all optimization options enabled, while using CUDA 3.1 toolkit [NVI 2010]. The input to the program is a PSLG containing possibly no edges but just points. All input numbers and computations are done in single precision mode, while double precision mode is also supported. To access the efficiency of our program, we compare its performance, on both synthetic and real contour data, with that of the best software available for CPU, called *Triangle* [Shewchuk 1996] in our following discussion. As indicated by Shewchuk, the performance of the divide-and-conquer algorithm of *Triangle* is among the best, and we thus use it to compute DT. To compute CDT, both *Triangle* and GPU-CDT insert constraints one by one based on their results of DT.

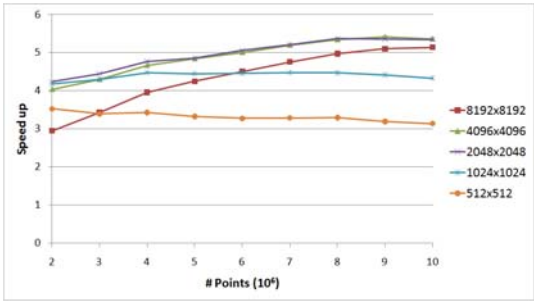
5.1 Synthetic Dataset

To generate synthetic data, we first generate randomly constraints which do not intersect with each other, then generate randomly points which do not lay on any constraints. There are 2 to 10 millions points with up to 1 million constraints in our test cases.

5.1.1 Varying the number of points



(a) running time



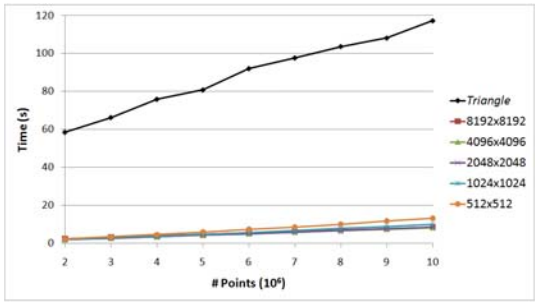
(b) speed up

Figure 7: *Performance comparisons between GPU-CDT and* Triangle *with no constraints and varying the total number of points from 2 millions to 10 millions: running time (top) and speed up of GPU-CDT over* Triangle *(bottom).*

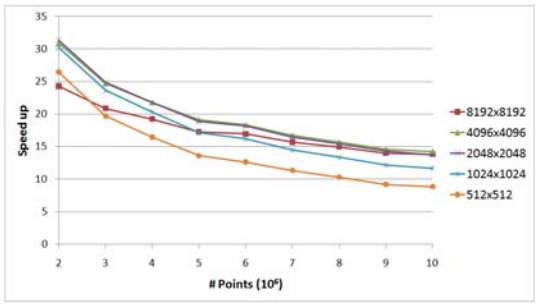
Figure 7 shows the running time and speed up of GPU-CDT compared to that of the *Triangle* with no constraints and varying the number of points in the range of 2 to 10 millions. In general, GPU-CDT can run faster than *Triangle* by a few order of magnitude. We note that GPU-CDT is superior to the prior work reported by Rong et al. [2008] which is better than *Triangle* by no more than 50%. Also, Figure 8 (top) shows the running time of GPU-CDT compared to that of the *Triangle* when fixing the number of constraints to 1 million and varying the number of points in the range of 2 to 10 millions. Figure 8 (bottom) shows the speed up of up to 31 times of GPU-CDT compared to *Triangle*. We notice that in increasing the number of points, the percentage of triangles intersected by constraints decreases (though the total increases), GPU-CDT thus has less flippings to perform in parallel which results in smaller speed up.

5.1.2 Varying the number of constraints

Figure 9 (top) shows the comparison of GPU-CDT with *Triangle* when fixing the number of points at 10 millions while varying the number of constraints from 0 to 1 million. We notice that *Triangle* shows an obvious time increasing when more constraints are added, while GPU-CDT shows a steady and slower growth in computational time. For *Triangle*, it inserts constraints one by one, and thus needs more time when there are more constraints: it takes, for example, 32 seconds to construct a DT but 85 seconds (70% of total time) to insert 1 million constraints into the DT. As for GPU-CDT, the time for generating constrained Delaunay triangulation with 1 million constraints using 8192×8192 texture size is 8.5 seconds with 2.45 seconds (less than 30% of total time) to insert constraints. Figure 9 (bottom) shows the speed up of GPU-CDT over *Triangle*

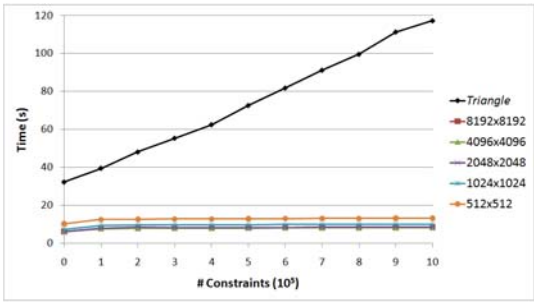


(a) running time

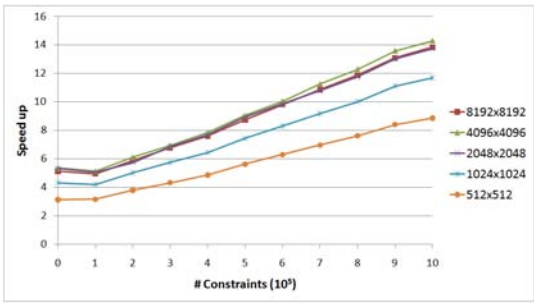


(b) speed up

Figure 8: *Performance comparisons between GPU-CDT and* Triangle *while fixing the number of constraints to 1 million edges and varying the total number of points from 2 millions to 10 millions: running time (top) and speed up of GPU-CDT over* Triangle *(bottom).*



(a) running time



(b) speed up

Figure 9: *Performance comparisons between GPU-CDT and* Triangle *while fixing the number of points at 10 millions and varying the number of constraints from 0 to 1 million: running time (top) and speed up of GPU-CDT over* Triangle *(bottom).*

when varying the number of constraints. For the number of constraints from 0 to 100,000 edges, the speed up decreases due to the overhead of GPU computation in GPU-CDT. However, when the number of constraints increases further, the speed up of GPU-CDT to *Triangle* is obvious due to the increase in the percentage of triangles in DT intersecting constraints and more parallelism can be achieved.

5.1.3 Varying the texture sizes

We test GPU-CDT with different texture sizes varying from 512×512 to 8192×8192 (as in Figure 8 and Figure 9). Basically, when using larger texture size, the total running time is faster and the speed up to *Triangle* is larger. There are two main reasons for this trend. Firstly, the running time for Phase 4 decreases as there are less number of missing points to handle, which also offsets the increase in the running time for Phase 1 and Phase 2. Secondly, with larger texture on a fixed number of points and constraints, the percentage of triangles intersected by constraints decreases, and thus less work to do and thus faster running time.

5.2 Contour Dataset

We downloaded contour maps from https://www.ga.gov.au/ for our experiments. Edges of the contour maps are divided into shorter edges as constraints. These cases have up to 5 millions points and constraints. Figure 10 shows one such example. When there are few points (less than 100,000), GPU-CDT runs slower than *Triangle* due to overhead in working with a large texture of 4096×4096 resolution. For other examples, GPU-CDT runs generally faster than *Triangle*. In these real-world data, all points are connected to at least one constraint. Most of these constraints are very short and do not intersect many common triangles with other constraints. As such, GPU-CDT can handle these test cases very well by inserting all constraints in a few rounds of the outer loop. On the other hand, within each inner loop, the amount of parallelism is low due to fewer triangles intersecting constraints (typically below 35%, well below the norm for synthetic data). On the whole, the average speed up to *Triangle* is up to four times in our tests.

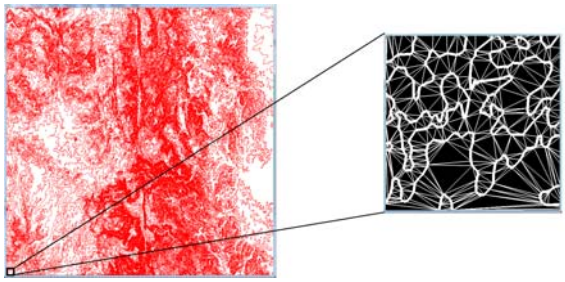


Figure 10: *An example contour dataset with a part of its CDT. Constraints are drawn in thick line segments.*

6 Concluding Remarks

This paper presents a new efficient and robust parallel approach to construct the 2D constrained Delaunay triangulation on the GPU. We have developed and tested the algorithm on both synthetic and contour map data. In both cases, we have shown that our approach runs up to 31 times faster than the best, robust constrained Delaunay triangulation algorithm on the CPU. In another perspective, possibly of independent interest, our contribution here is a framework

of utilizing the GPU in a non-conventional way to capitalize on the computing power of many cores: instead of solving a problem completely in the continuous space, we first transform it to a problem in the digital space, solving the latter to obtain an approximate solution, and then augmenting it into the required solution in the continuous space. With this work, we look forward to developing more efficient computational geometric algorithms running on GPUs in the near future.

References

AURENHAMMER, F. 1991. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys 23*, 3, 345–405.

BERNAL, J. 1995. Inserting line segments into triangulations and tetrahedralizations. Tech. Rep. 5596, National Institute of Standards and Technology.

BOISSONNAT, J.-D. 1988. Shape reconstruction from planar cross sections. *Computer Vision, Graphics, and Image Processing 44*, 1, 1–29.

CAO, T.-T., EDELSBRUNNER, H., AND TAN, T.-S. 2010. Proof of correctness of the digital Delaunay triangulation algorithm. manuscript, available at `http://www.comp.nus.edu.sg/~tants/delaunay2DDownload_files/notes-25-01-2010.pdf`.

CAO, T.-T., TANG, K., MOHAMED, A., AND TAN, T.-S. 2010. Parallel banding algorithm to compute exact distance transform with the GPU. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 83–90.

CHEW, L. P. 1989. Constrained Delaunay triangulations. *Algorithmica 4*, 97–108.

DWYER, R. 1987. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica 2*, 137–151.

FISCHER, I., AND GOTSMAN, C. 2006. Fast approximation of high-order voronoi diagrams and distance transforms on the gpu. *J. Graphics Tools 11*, 4, 39–60.

FORTUNE, S. 1987. A sweepline algorithm for voronoi diagrams. *Algorithmica 2*, 153–174.

FORTUNE, S. 1997. *Handbook of discrete and computational geometry*. CRC Press, Inc., Boca Raton, FL, USA, ch. Voronoi diagrams and Delaunay triangulations, 377–388.

GOLD, C. M. 1994. A review of potential applications of voronoi methods in geomatics. In *Proceedings of the Canadian Conference on GIS*, 1647–1656.

GOLD, C. M. 1994. Review: Spatial tesselations - concepts and applications of Voronoi diagrams. *International Journal of Geographical Information System 8*, 237–238.

GRAHAM, R. L. 1972. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters 1*, 4, 132 – 133.

GUIBAS, L., KNUTH, D., AND SHARIR, M. 1992. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica 7*, 381–413.

HIGHNAM, P. T. 1982. The ears of a polygon. In *Information Processing Letters*, 196–198.

HOFF, III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 277–286.

HUEBNER, K. H., DEWHIRST, D. L., SMITH, D. E., AND BYROM, T. G. 2001. *The Finite Element Method for Engineers*. Wiley, New York, NY, USA.

KALLMANN, M., BIERI, H., AND THALMANN, D. 2003. Fully dynamic constrained Delaunay triangulations. In *Geometric Modelling for Scientific Visualization*, G. Brunnett, B. Hamann, and H. Mueller, Eds. Springer-Verlag.

LEE, D., AND LIN, A. 1986. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry 1*, 201–217.

NVIDIA. 2010. *CUDA C Programming Guide*.

PREPARATA, F. P., AND SHAMOS, M. I. 1985. *Computational geometry: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA.

RONG, G., TAN, T.-S., CAO, T.-T., AND STEPHANUS. 2008. Computing two-dimensional Delaunay triangulation using graphics hardware. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 89–97.

SHAMOS, M. I., AND HOEY, D. 1975. Closest-point problems. *FOCS '75: the 16th Annual Symposium on Foundations of Computer Science*, 151–162.

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.

SU, P., AND SCOT DRYSDALE, R. L. 1997. A comparison of sequential Delaunay triangulation algorithms. *Computational Geometry: Theory and Applications 7* (April), 361–385.

TREINISH, L. A. 1995. Visualization of scattered meteorological data. *IEEE Computer Graphics and Applications 15*, 20–26.

A Our Flooding Algorithm

Phase 1 of our proposed algorithm is to compute a digital Voronoi diagram of a collection of points, or called *seeds*, in a texture. Technically, we want and will prove in the following that the output of this phase is the same as that of the Standard flooding mentioned in [Cao et al. 2010a]. Then, we can apply their result to conclude that Phase 2 of our algorithm indeed produces a triangulation.

In our algorithm, we use the efficient Parallel Banding Algorithm (PBA) [Cao et al. 2010b] running on the GPU to start the coloring, which results in a Euclidean coloring. Each of the Voronoi region resulted has a connected component called *bulk* which is path-connected to its seed, and *debris* (if any) which are disconnected from the seed. Cao et al. show that bulks are subsets of the result of the Standard flooding. So, our challenge is to identify and recolor the debris to be the same as in the Standard flooding. Since there are very few debris in general, the recoloring is done using the CPU and then the result is sent back to the GPU to complete the coloring. Algorithm 5 is our proposed approach. We use Q to mean a priortiy queue, $N(A)$ the set of pixels neigboring pixel A, and p_i a seed with color i. The operation $\min(Q)$ on Q is to remove and return the pair with minimum distance between the pixel center and the seed of the color; $\|A - p_i\| \prec \|B - p_j\|$ means $\|A - p_i\| < \|B - p_j\|$, or $\|A - p_i\| = \|B - p_j\|$ with consistent tie breaker (using, for example, the coordinates of pair of points).

Algorithm 5 Flooding Algorithm

```
Compute coloring with PBA, and identify all debris as uncolor.
Q = ∅
for each debris A do
  Q = Q ∪ {(A, i) | B ∈ N(A) having been colored i, and
                    ‖B − p_i‖ ≺ ‖A − p_i‖}
while Q ≠ ∅ do
  (A, i) = min(Q)
  if A is not colored then
    Color A with color i
    Q = Q ∪ {(B, i) | B ∈ N(A) and ‖A − p_i‖ ≺ ‖B − p_i‖}
  end if
end while
```

We next show Algorithm 5 indeed produces the same output as that by the Standard flooding. We just need to argue for those pixels that are identified as debris. Consider the very first instance when Algorithm 5 colors a debris A with color r (inside the while-loop) whereas the Standard flooding produced pixel A with color $s \neq r$. There are two situations to consider :

CASE 1: $\|A - p_r\| \prec \|A - p_s\|$. From Algorithm 5, there exists a neighbor B of A colored by r earlier, and $\|B - p_r\| \prec \|A - p_r\|$. It follows that $\|B - p_r\| \prec \|A - p_r\| \prec \|A - p_s\|$. According to our choice of A, pixel B is colored by r in the Standard flooding. By the Ordered Coloring Lemma [Cao et al. 2010b], (A, r) must have been considered in the Standard flooding before (A, s) and A should thus have been colored by r then, a contradiction.

CASE 2: $\|A - p_s\| \prec \|A - p_r\|$. In the result of the Standard flooding, there is a monotonic path from p_s to A. Part of this path has been colored the same (with s) in Algorithm 5 when we are about to color A. Let C be the pixel closest to p_s in that path not yet colored by Algorithm 5. With the previous pixel before C has been colored with s, (C, s) must have been added to Q in Algorithm 5. Since $\|C - p_s\| \leq \|A - p_s\| \prec \|A - p_r\|$, we must have (C, s) inside Q to be extracted before (A, r), a contradiction.

The argument in Case 2 also implies the algorithm colors all pixels. This concludes our claim of correctness of our flooding algorithm.

B Transforming the point set

To work on seeds with floating point coordinates, Phase 1 needs to map them to a $m \times m$ texture. We want precise computation so that the triangulation computed with respect to seeds mapped to the texture remains a triangulation when we do a part of the inverse mapping to the original coordinates of seeds. We discuss below precise computation can be achieved by representing *scale* and *translate* used in the mapping with a certain number of bits.

We just consider the 1D coordinate in x-axis; the discussion can be generalized to 2D with *scale* be the larger one calculated from both dimensions, while *translate* is simply a vector of two components. Let the seeds be such that their minimum and maximum x-coordinates are $x_{\min}$ and $x_{\max}$, respectively. Let x be the original coordinate of a seed. The coordinate of the seed mapped to the texture is thus $\bar{x} = \lfloor (x - \textit{translate})/\textit{scale} \rfloor$ where $\textit{translate} = x_{\min}$ and $\textit{scale} = (x_{\max} - x_{\min})/m$. The computation of a triangulation in Phase 2 is then performed using these integer coordinates.

Then, Phase 3 is to eventually shift all points in the texture back to their positions given in the input. To maintain as many good cases of shifting as possible, we first perform the inverse scaling and shifting for the whole bounding box with all the points. Specifically, we have $x' = (\bar{x} \times \textit{scale} + \textit{translate})$ as our new coordinate of a seed before shifting it (to negate the effect of the truncation to integer coordinate) to the original coordinate x. To ensure we still have a same triangulation with x' in place of $\bar{x}$, we must compute floating point number x' with no rounding error.

Let $(p\text{Max} + 1)$ be the maximum number of bits available for the mantissa in our floating point numbers. Note that the explicit mentioned of "+1" here is a provision for possible overflow of $(\bar{x} \times \textit{scale} + \textit{translate})$. Let the number of bits used to represent the mantissas of the two constants *scale* and *translate* be pS and pT, respectively. Note that $\bar{x}$ is a non-negative integer with maximum value of $(m - 1)$ and thus needs $pM = (\log m)$ bit to represent. We keep $pT = p\text{Max}$.

We are ready to discuss how to set *scale* and *translate* before doing the actual mapping to texture. First, the result of the $(\bar{x} \times \textit{scale})$ is accurately represented using no more than pMax bits, as long as we keep *scale* by doing the necessary round up of its value to use $pS = (p\text{Max} - pM)$ bits. The round up can increase *scale* by just the little bit at the least significant bit of *scale* and thus we still be able to spread out the mapping of seeds on the texture. Second, the addition of $(\bar{x} \times \textit{scale})$ with *translate* can result in rounding error as *translate* can be much smaller or much larger than $(\bar{x} \times \textit{scale})$. Let $\textit{range} = (x_{\max} - x_{\min}) = (m \times \textit{scale})$. We consider two cases to guarantee that the computation of x' is accurate:

CASE 1: $\textit{translate} \leq \textit{range}$. Let 2^t be the largest term in the binary representation of *range*. We reduce *translate* by removing all terms in its binary representation that are smaller than $2^{t-(p\text{Max}-1)}$.

CASE 2: $\textit{translate} > \textit{range}$. Let 2^r be the largest term in the binary representation of $\textit{translate} = x_{\min}$. We round up all terms in the binary representation of *scale* that are smaller than $2^{r-(p\text{Max}-1)+pM}$. Because $\textit{range} = x_{\max} - x_{\min} \geq 2^{r-(p\text{Max}-1)}$, we have *scale* represented by pS bits is larger than $2^{r-(p\text{Max}-1)+pM}$ for any meaningful input and is thus non-zero. Also, the round up does not increase more than double the value of *scale*, and we thus still be able to spread out the mapping of seeds on the texture.