

B.Comp. Dissertation

Computing 2D Delaunay Triangulation using GPU

By

Cao Thanh Tung

Department of Computer Science

School of Computing

National University of Singapore

2008/2009

B.Comp. Dissertation

Computing 2D Delaunay Triangulation using GPU

By

Cao Thanh Tung

Department of Computer Science

School of Computing

National University of Singapore

2008/2009

Project No: H066860

Advisor: Dr. Tan Tiow Seng

Deliverables:

Report: 1 Volume

Abstract

In recent years, the GPUs have been widely adopted in various general purpose computation fields. In computational geometry field, many papers have proposed methods to utilize the GPUs to solve traditional problems such as Discrete Voronoi Diagram and Delaunay Triangulation. The latest GPU Delaunay Triangulation (GPU-DT) algorithm exploits the GPU to assist in the computation of a Discrete Voronoi Diagram of a set of point S , from which a triangulation $T'(S)$ can be constructed. Due to the discrete nature of the discrete Voronoi diagram, several transformations need to be performed on $T'(S)$ to get the correct Delaunay triangulation $T(S)$. As these transformations are highly complicated, they were originally performed in the CPU. When the number of sites goes up while the resolution of the discrete Voronoi diagram is fixed, the necessary transformation work soon dominates the running time of GPU-DT. In this report, we present a novel approach to transform $T'(S)$ into $T(S)$ in parallel, running completely in the GPU, using CUDA, the latest massively parallel programming model from NVIDIA. Using a well designed multiple passes approach, our algorithm effectively improves the performance of GPU-DT by up to two folds, and up to 180% faster when compared to *Triangle*, the well-known fastest CPU Delaunay Triangulator. We will also present a rigorous proof of the correctness of the construction of a triangulation from the discrete Voronoi Diagram which is used in GPU-DT.

Subject Descriptors:

I.3.1 Hardware Architecture - Graphics processors, Parallel processing

I.3.5 Computational Geometry and Object Modeling - Geometric Algorithm, Languages, and Systems

Keywords:

Delaunay Triangulation, GPGPU, Computational Geometry, Voronoi Diagram

Implementation Software and Hardware:

Microsoft Windows XP Service Pack 3, Microsoft Visual Studio 2005, CUDA 2.0

Intel Core 2 Quad Q6600 2.4Ghz, NVIDIA GeForce GTX280 PCI-X 1024MB

List of Figures

1.1	Computational power and memory bandwidth of the CPU and GPU	2
1.2	The incremental construction in E^2	3
1.3	The incremental insertion in E^2	4
2.1	Disconnected Voronoi region due to sharp corner	7
2.2	Missing triangles due to Voronoi vertices lying outside the texture	9
2.3	Different cases for shifting sites	11
3.1	A difficult shifting case and its solution	14
3.2	Fake boundary to simplify inserting missing site step	17
3.3	Possible conflicts while parallel inserting missing sites and flipping triangles . . .	19
3.4	Flipping two edges that have adjacent pair of edges	26
4.1	Performance comparison between the old CPU and our new GPU implementation	29
4.2	Performance of three fixing steps on different texture size	29
4.3	Comparison between Triangle, old GPU-DT, and our <i>new</i> CUDA implementation	32
4.4	Performance comparison between Triangle and our <i>new</i> CUDA implementation .	33
4.5	Improvement of our <i>new</i> CUDA implementation over <i>Triangle</i>	33
5.1	The bulk of E_i after coloring the first j pixels by flooding.	36
5.2	Illustration for the proof of the Distance Invariant	37
5.3	A seed $s \in S$ enclosed by a monotonic path P_{xx_i} from x till x_i	41
5.4	No seed enclosed by the monotonic path P_{zy}	42
5.5	P_{bd} with db form an enclosed region containing c	43
5.6	A corner of colors a, b, c in counter-clockwise order while a, b, c in clockwise order.	44
5.7	Two corners of the same three colors of a, b, c	45
5.8	Two overlapping triangles	45

Table of Contents

Title	i
Abstract	ii
List of Figures	iii
1 Introduction	1
1.1 Background and Motivation	1
1.2 Related works	3
2 GPU-DT: A 2D GPU Delaunay Triangulator	6
2.1 Background	6
2.2 Discrete Voronoi Diagram	8
2.3 Delaunay Triangulation Construction	10
3 GPU Parallel Construction of Delaunay Triangulation	13
3.1 Challenges and Solution	13
3.1.1 Shifting	14
3.1.2 Inserting Missing Sites	16
3.1.3 Flipping	19
3.2 Implementation details	20
3.2.1 Shifting	21
3.2.2 Inserting Missing Sites	23
3.2.3 Flipping	25
4 Experiment results	28
4.1 CPU vs GPU Implementation Stepwise	28
4.2 Overall Comparison	31
5 Proof of Correctness	34
5.1 Standard Flooding and Ordered Flooding	34
5.1.1 Euclidean Coloring and Standard Flooding	35
5.1.2 Distance Invariant	36
5.1.3 Necks and planarity	40
5.2 Correctness of GPU-DT	41
6 Conclusion and Future Works	47
References	49

Chapter 1

Introduction

1.1 Background and Motivation

In a computer today, the Graphics Processing Unit (GPU) probably has the most powerful computational capability. Researchers and developers are becoming more and more interested in GPGPU - General Purpose computing on the GPU, an area in which the GPU is utilized for many purposes other than graphics processing. There are many reasons behind this trend: a GPU is a low cost hardware with tremendous memory bandwidth and processing power; Graphics hardware is fast and getting faster quickly; and the programmability of the GPU is getting better and better (Owens, Luebke, Govindaraju, Harris, Krger, Lefohn, & Purcell, 2005). Figure 1.1 compares the computational power and memory bandwidth of the GPU and CPU (NVIDIA, 2008). A single GeForce GTX280 can perform more than 900 GFLOPS and has the memory bandwidth of 140GB/s, which is almost ten times faster than the latest CPU. This is because the GPU is designed for highly data-parallel computing intensive tasks. GPU devotes most of its transistors to arithmetic computational purpose. As a result, the GeForce GTX280 with 240 stream processors capable of performing floating point operations has a much higher computational capability compared to the CPU.

Together with the advances in hardware performance, the programmability of the GPU has also significantly improved with the introduction of Fragment Shader, Vertex Shader and Geometry Shader. Recently, NVIDIA has introduced the Tesla unified architecture and CUDA,

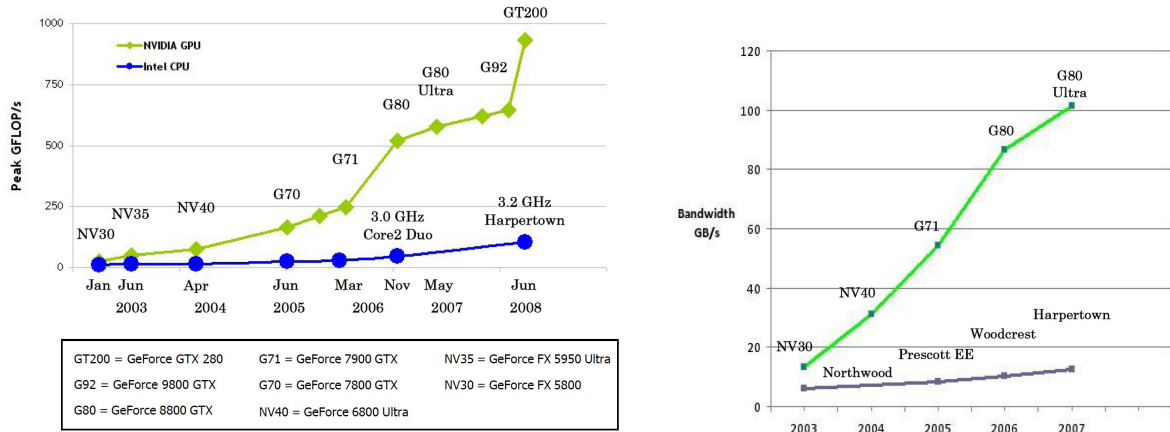


Figure 1.1: Computational power and memory bandwidth of the CPU and GPU

an easy to learn, easy to use scalable parallel programming model (NVIDIA, 2008), letting developers fully utilize the horsepower of the GPUs for many complicated tasks. As such, GPU has been adopted in many general purpose applications ranging from numeric computing operations, physical simulation and game physics to datamining and computational geometry (Owens et al., 2005).

In the field of Computational Geometry, the GPU has been employed to construct Discrete Voronoi Diagram and Delaunay Triangulation of millions of sites. Delaunay triangulation is one of the fundamental problem in computational geometry and is often used to build meshes for the finite element method. Delaunay Triangulation has been thoroughly studied for many years in traditional computational geometry field. The best known Delaunay triangulation implementation on CPU is *Triangle*, a robust and fastest software that won his author the *2003 James Hardy Wilkinson Prize in Numerical Software* (Shewchuk, 1996b). Recently, Rong, Tan, Cao, and Stephanus (2008) proposed GPU-DT, a hybrid method that utilizes both CPU and GPU in the process of constructing 2D exact Delaunay triangulation. GPU-DT is the first algorithm that employ the enormous computing power of the new generation GPU to compute a Discrete Voronoi Diagram of the point set and construct a valid triangulation from that. Later, several transformations are performed on the CPU to convert the constructed triangulation into the correct Delaunay triangulation.

Since GPU-DT still heavily rely on the CPU to perform necessary fixing on the approxi-

mated triangulation, depending on the quality of the approximated triangulation, more work will be performed on the CPU. With the limited size of the discrete Voronoi diagram, when the number of sites increases, the initial triangulation obtained from the Voronoi diagram becomes less accurate, thus the work on the CPU soon becomes the bottleneck of the algorithm. Motivated by the rapid increase of the performance of the GPU and the flexibility of the parallel programming model CUDA, we want to further employ the GPU on the rest of the GPU-DT algorithm, minimizing the workload on the CPU, making GPU-DT the world’s fastest Delaunay Triangulator.

1.2 Related works

There are two commonly used principles for the construction of the Delaunay triangulation: the incremental construction and the incremental insertion. In the incremental construction method, a starting triangle is created and then the triangulation is built by adding new triangles adjacent to the existing triangles (Figure 1.2). This method is often used together with the Divide and Conquer strategy to construct partial Delaunay triangulations and these are then combined together in the merge phase. This method is used in *Triangle* and is considered the fastest algorithm. However, parallel construction of Delaunay triangulation using this strategy

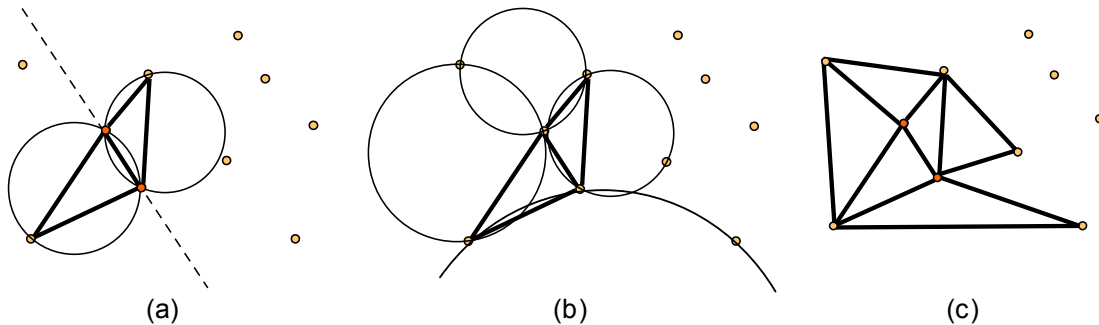


Figure 1.2: The incremental construction in E^2 : (a) initial two triangles, (b) construction of next triangles and (c) partial triangulation (Kohout et al., 2005)

often suffers from two drawbacks. Firstly, the merge phase is usually very complicated, involving not only the building of triangles on the boundary and fixing existing triangles; Secondly, the merge phase usually can only be done by one processor. One example is the algorithm of

Aggarwal et al. (1988). There are some other algorithms that try to reduce the work in the merge phase, but that makes the triangulation phase more complicated and not suitable for implementation on the GPUs.

In the incremental insertion method, we start by constructing a simple triangle mesh that contains all the sites. Then sites are inserted into that triangle mesh by subdividing the triangles containing them, and then in-circle tests are performed recursively on adjacent triangles and flipping are performed if necessary (Figure 1.3). Kohout, Kolingerová, and Žára (2005) give detailed information about several algorithms for parallel construction of the Delaunay triangulation on computers with shared memory and several processors using this method. Those algorithms employ a randomized incremental insertion approach and rely on the fact that when inserted randomly, the chance of dead-lock (e.g. two sites inserting in the same triangle) is small. However, these algorithms are not suitable for the GPU with hundreds of processors and the auxiliary data structures used are hard to be efficiently maintained in the GPU. Recently,

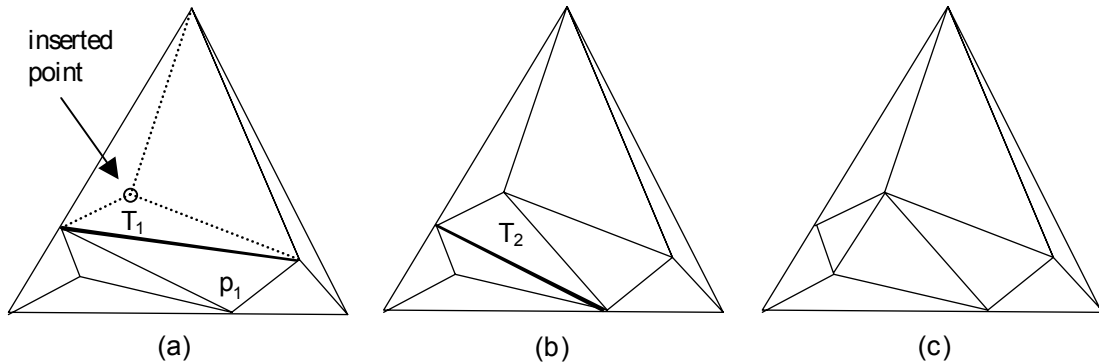


Figure 1.3: The incremental insertion in E^2 . Edges that should be flipped are thick: (a) subdivision, (b) propagation of flips and (c) resulting triangulation (Kohout et al., 2005)

Rong et al. (2008) proposed a method to compute the Delaunay triangulation using the GPU, which is somewhat similar to the incremental insertion approach. First, they construct the discrete Voronoi diagram of the point set in the GPU, from which they construct a valid triangulation. Due to the nature of the discrete Voronoi diagram, sites can go missing and there can be crossing problems when moving the sites back to their continuous coordinates. As such, shifting from discrete to continuous world needs careful adjustment. Having a triangulation

which is a good approximation of the Delaunay triangulation, missing sites are then inserted back, and flipping are then performed to correct all violation of the in-circle criterion. Due to the complexity of performing these adjustments, the authors perform them in the CPU.

The advantage of this approach is that the triangulation initially constructed is already very close to the correct Delaunay triangulation, and it contains enough triangles so that the chance of conflict while inserting back missing sites in parallel is negligible. As such, in this report, we will propose a method to perform all the necessary adjustment to the initial triangulation in the GPU in parallel using the CUDA parallel programming model. The main contributions of our project are:

- A new and efficient approach for computing exact 2D Delaunay triangulation completely in parallel in the GPU based on GPU-DT (Rong et al., 2008). This work has been released on the web (Cao et al., 2009).
- A more rigorous proof of the correctness of the triangulation constructed from the discrete Voronoi diagram compared to the one provided in (Rong et al., 2008). This work is done in collaboration with Professor Herbert Edelsbrunner.

The rest of the report is organized as follows. Chapter 2 describes in detail the GPU-DT algorithm proposed in (Rong et al., 2008). After that, Chapter 3 discusses the challenges in performing the fixing steps of GPU-DT in parallel in the GPU, our proposed solutions for these problems, and the implementation details. The experiment results with detailed analysis are given in Chapter 4. Chapter 5 will continue the discussion with a rigorous proof of correctness of GPU-DT. Finally, Chapter 6 concludes the report and overviews some possible future works.

Chapter 2

GPU-DT: A 2D GPU Delaunay Triangulator

In this chapter, we will study GPU-DT, a 2D Delaunay triangulation algorithm which utilize both GPU and CPU. We will first start with some background of Voronoi diagram and Delaunay triangulation. After that we will carefully describe the GPU-DT algorithm in Rong et al. (2008). Not only looking at the steps in the algorithm, we will also look at the detail of each step to get the necessary insight before we discuss the challenges in performing them in parallel in the next chapter.

2.1 Background

Let $S = \{x_1, x_2, \dots, x_n\}$ be a set of n distinct sites in the plane. The *Voronoi Diagram* $V(S)$ of S is the subdivision of the plane into n cells, one for each site. A point A lies inside the cell corresponding to a site x if and only if it is closer to x than to any other sites in S . A point B lies on a *Voronoi edge* between two sites x_i and x_j if and only if the largest circle centered at B containing no other sites in S touches exactly two sites x_i and x_j . If such circle touches three sites or more, B is called a *Voronoi vertex*.

The *Delaunay Triangulation* of a set of sites S is a triangulation $T(S)$ such that no sites in S is inside the circumcircle of any triangle in $T(S)$. It has been proven that the Delaunay

triangulation of S is the dual graph of its Voronoi Diagram.

In the discrete version of the Voronoi diagram, instead of a continuous plane, we only consider the set of all integer grid points. A grid point A belongs to the set E_i if A lies inside the Voronoi region of the site x_i , and we say that A is *colored* by x_i . In case A is in equal distance from x_i and x_j and $i < j$, we color A by x_i . The set of colored grid points form the discrete Voronoi diagram $D(S)$ of S . We call this procedure *Euclidean coloring*.

Note that E_i need not be connected. The *bulk* of E_i is the connected component of E_i that contains the seed point, x_i . Other pixels of E_i are its *debris*. Debris exists only inside a sharp corner of the corresponding Voronoi region (Figure 2.1). There are other coloring procedure

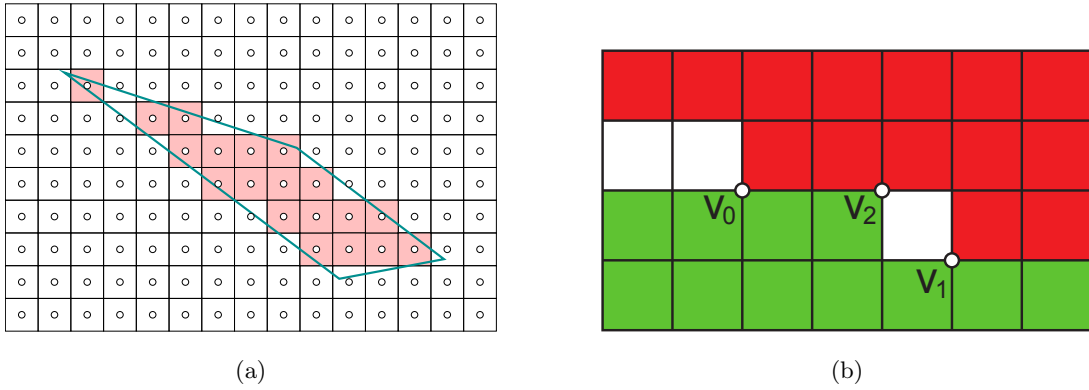


Figure 2.1: A region with a sharp corner. (a) Its corresponding digital region consists of bulk and one debris pixel. (b) Debris pixels can generate duplicate triangle

that can guarantee that the regions stay connected. In order to do so, we only color a pixel A by x_i if by the time we consider A , either $A = x_i$ or A has a neighboring pixel colored by x_i . The *Standard Flooding* algorithm is as follow: We store eligible pixel-color pairs (A, x_i) in a priority queue sorted by $\|A - x_i\|$. At each step, remove the pair with smallest distance, color the pixel A by x_i if A is not yet colored, update the queue, and repeat until the priority queue is empty.

The algorithm described in (Rong et al., 2008) consists of two phases. In the first phase, we compute the discrete Voronoi diagram $D(S)$ of the set of sites S . This will help us construct a valid triangulation $T'(S)$ which is an approximation of the actual Delaunay triangulation $T(S)$. Note that we use the $D(S)$ that is equivalent to what produced by standard flooding, i.e. no

debris pixels.

In $T'(S)$, all sites are in the discrete space. We need to shift back these sites into their original continuous space. Due to the discrete nature of the discrete Voronoi diagram, multiple sites can be mapped into one grid point. We can only keep one of them in the discrete Voronoi diagram. Other duplicated sites are considered missing, and we have to insert them back to our triangulation. In the end, we perform recursive flipping on the triangulation to get the final Delaunay triangulation. During the transformation from $T'(S)$ to $T(S)$, we need to perform counterclockwise tests and in-circle tests. These tests are done robustly using the method proposed by Shewchuk (1996a).

2.2 Discrete Voronoi Diagram

GPU-DT starts by computing the discrete Voronoi diagram of the input point set. The diagram will be computed in a texture, so the first step is to scale the continuous space containing the point set into the same size of the discrete space - the texture, and then render all the sites into the texture. Due to the discrete nature of the texture, multiple sites can be mapped to the same texel. In this case, as discussed above, we only keep one site, and treat others as *missing sites*. They will be inserted back into the triangulation later.

There are various ways to generate the discrete Voronoi diagram. The authors suggested using the Jump flooding algorithm (Rong & Tan, 2006, 2007). This method gives us the best performance, although it does introduce some errors in the diagram. These errors, however, does not change the topology of the diagram, thus do not have any effect on the algorithm (Rong et al., 2008).

The next step would be to construct a triangulation from the discrete Voronoi diagram. First, we need to locate all the Voronoi vertices. A corner shared by four pixels is defined as a Voronoi vertex if these four pixels are of three or four different colors. For each corner surrounded by pixels of three different colors, we add one triangle into our triangulation, and for one that is surrounded by pixels of four colors, we add two triangles into our triangulation.

The result of JFA, however, is equivalent to the Euclidean coloring of the texture, and thus

can contain debris points (Figure 2.1). When this happens, we might be able to identify more than one corner that can generate the same triangle. As such, we need to remove all debris points. For each pixel that is disconnected from the bulk of its site, we color it by the closest sites among those that have colored its neighbors. This process will give us a Voronoi diagram equivalent to the result of the Standard flooding algorithm.

Rong et al. (2008) also notes about the case of missing triangles due to Voronoi vertices lying outside the texture, although the sites themselves are all inside the texture (Figure 2.2). To fix that problem, the authors suggest we traverse along the boundary of the texture with the idea similar to the Graham’s scan algorithm (Graham, 1972), using the counterclockwise test to decide whether to add more triangles into the triangulation. Note that we exclude the

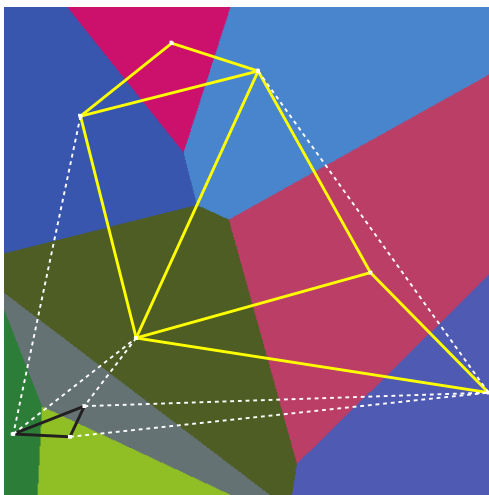


Figure 2.2: Missing triangles due to Voronoi vertices lying outside the texture. (Rong et al., 2008)

case where two diagonally opposite pixels have the same color. Section 5.1.3 will prove that this case will never happen. We will also prove that from the Voronoi diagram computed, we can generate a correct triangulation, i.e. without crossing, overlapping, duplication, and no holes (section 5.2).

To facilitate the next stage of fixing the triangulation, we need to maintain a few important datastructures. First of all, for each triangle, beside their vertices, we also need to know their neighbors. Each triangle will store the index of its three neighboring triangles. With this, we

can *walk* from triangle to triangle easily. Secondly, for each site, we need to maintain a list of triangles that have that site as a vertex. This information together with the neighborhood information let us easily walk around the triangle fan surrounding a site.

2.3 Delaunay Triangulation Construction

After the previous stage, we have obtained a triangulation $T'(S)$ which is very close to a Delaunay triangulation, with just some remaining issues to fix. First of all, sites in $T'(S)$ are in discrete space, they need to be shifted back to the continuous space. Secondly, there are some missing sites that are not recorded in the discrete Voronoi diagram. We need to insert them back to our triangulation. Lastly, we need to perform some edge flipping if necessary to guarantee that the triangulation we obtained is the Delaunay triangulation. In the next three sections, we will discuss in detail the work needed in performing these three steps: Shifting, Inserting missing sites, and Flipping. Note that these steps proposed in (Rong et al., 2008) are to be performed in the CPU. These detail discussions will give the foundation from which we continue to discuss the challenges and our solution to perform these steps in parallel in the GPU.

Shifting

In this step, we need to shift each site x' from the discrete space back to its original position x in the continuous space. After shifting x' back to x , we may have some crossing triangles. Rong et al. (2008) have identified all possible shifting situations (Figure 2.3), and proposed methods to deal with each situation. These situations can be categorized into two cases: good case and bad case. In a good case (Figure 2.3a), there is no crossing after we shift x' to x , and we are still in the triangle fan of x' . In this case, we can simply update the coordinate of x' and do not need to update the triangulation structure. This is the ideal case, and in practice it happens more than 90% of the time. The reason is because the shifting distance from x' to x is small compared to the distance among sites in the discrete space.

In a bad case, shifting x' to x causes crossing triangles. In this case, we need to update

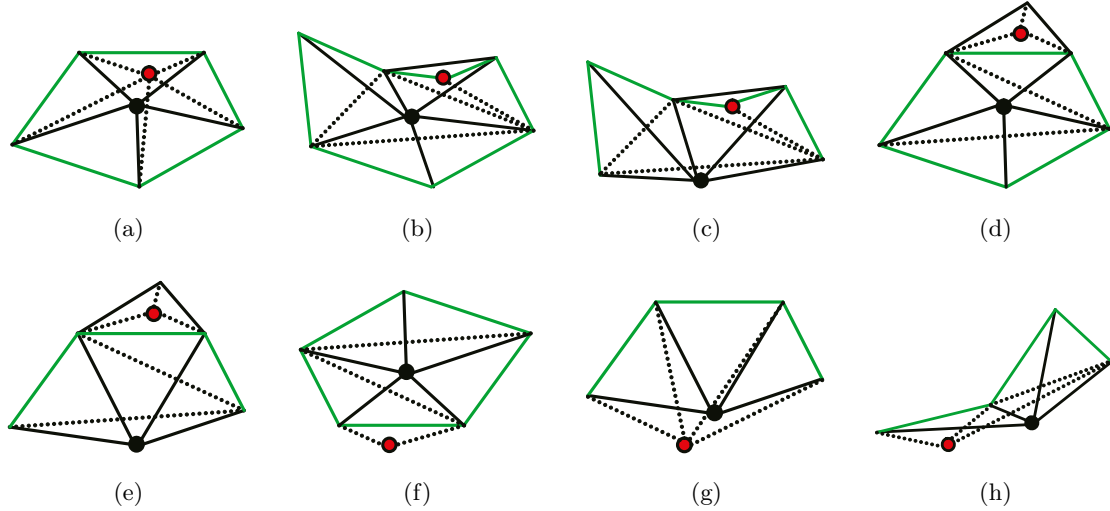


Figure 2.3: Different cases for shifting sites. Solid edges are original edges, dotted edges are new edges.

the triangle mesh to fix these crossing. For each of the situation, we either have to delete some triangles and then subdivide another triangle, or remove x' from the mesh first, triangulate the triangle fan of x' and then insert x into the triangulation. In case x falls outside the triangulation (i.e. outside the convex hull of the triangulation), we will need to add some triangles to fix the convex hull as well. This task is very complicated.

Inserting Missing Sites

This step handles missing sites resulted due to the mapping of multiple sites into one discrete position. At the beginning of this step, we have a triangulation of a subset of the set of sites S . Those that are missing will be inserted one by one into the triangulation. To insert a site into the triangulation, we first try to locate a triangle that contains this site. There are several cases that need to be considered. If we can find a triangle that contains the missing site inside, we can simply delete this triangle, subdivide it into three triangles, and insert them into the triangulation. In case that the missing site lies on an edge between two triangles, we will have to delete both of these and insert four corresponding triangles. If we cannot find a triangle that contains the missing site, we can safely assume that it lies outside the triangulation, because we can guarantee that there is no hole in the triangulation. In this case, we need to fix the convex

hull.

Flipping

After the inserting missing sites step, we have a full triangulation of S that is close to the correct Delaunay triangulation. The reasons for the inaccuracy are: The previous two steps fix the triangulation without worrying about the in-circle criterion; The triangulation obtained from the discrete Voronoi diagram is not guaranteed to be a Delaunay triangulation. In order to convert our triangulation into a Delaunay triangulation, we traverse all the edges in the triangulation, perform the in-circle test among the two corresponding adjacent triangles, and possibly do an edge-flip. If we perform an edge-flip on an edge, we will recursively check the nearby edges. After all the edges have passed the test, the triangulation is guaranteed to be a Delaunay triangulation. Rong et al. (2008) claimed that this step will not take long as the triangulation constructed from the discrete Voronoi diagram in the first stage should already be very close to the Delaunay triangulation.

Chapter 3

GPU Parallel Construction of Delaunay Triangulation

In this chapter, we will first discuss some challenges that we will encounter when performing Shifting, Inserting missing sites, and Flipping in the GPUs. This includes some robustness issues and synchronization problems. We will also give our solution to efficiently solve these problems, together with detailed discussion on the implementation in CUDA.

3.1 Challenges and Solution

During the process of transforming the triangulation obtained from the discrete Voronoi diagram into the Delaunay triangulation, we need to keep modifying our triangle mesh. On the other hand, in order to fully utilize the GPU, we will need hundred thousands of threads running together. We will need to consider the synchronization problem when two threads trying to modify the triangle mesh, for example the same triangle, at the same time. The more threads we have, the more severe this problem becomes. On the other hand, due to the hardware design of the GPU, if we keep using synchronization primitives such as atomic operations, the thread executions will basically be serialized, which will directly affect the performance. In order to solve this problem, we repeatedly use one simple strategy. We break the set of threads into several groups. Within a group, threads do not conflict with one another. In other words,

threads in the same group can be executed concurrently without doing any synchronization. The question is now how to put threads into groups quickly while keeping the number of groups as small as possible. The strategy will vary depending on the task that we need to do.

A second design principle that we have to keep in mind is that GPU is a SIMD architecture. In order to fully utilize the parallelism of the hardware, we need threads to have as similar execution as possible. Because of that, the GPU is very bad at branching. If two threads run on two different branches in the same processor, their execution will be serialized. As such, we need to make our code as simple and as uniform as possible.

With these two design principles in mind, we will now discuss the challenges in performing each of our three steps in the GPU, and how we solve the problems.

3.1.1 Shifting

Robustness issues

When we shift a site, there are two cases: Good case and Bad case. When we encounter a bad case, the authors suggest that we categorize it further into five different cases in which we can carefully fix the triangle mesh appropriately. One problem with this approach is that it is very difficult to handle cases when there are collinear sites. For example, in Figure 3.1a, shifting G to G' can be very troublesome. We have to make sure not to create triangle CBG' or BAG' . Detecting such situations is already very difficult, not to say performing a correct fix.

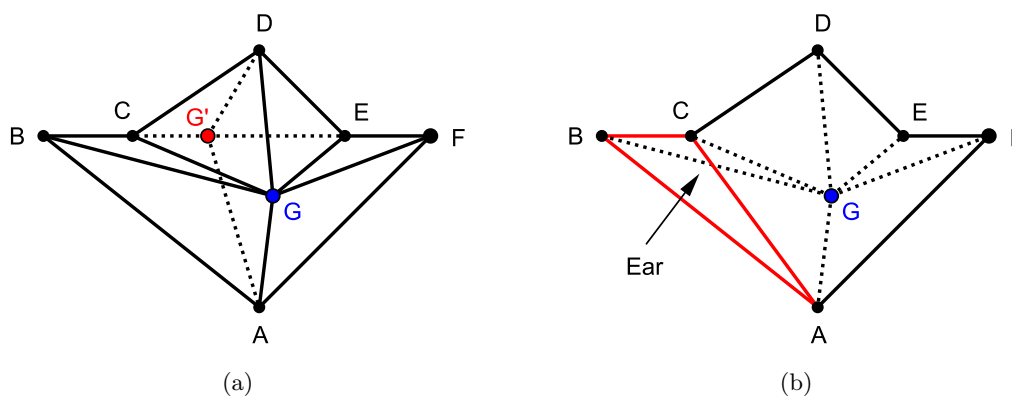


Figure 3.1: Shifting (a) With collinear points (b) Ear cutting

One solution for the above problem is to break it into two simpler problems. We can first

remove G and patch the polygon formed by the triangle fan around G . After that, we can treat this site as a missing site and add it back in the later stage. There are several reasons for this method to be simpler and more robust:

- Patching the polygon is easy. Note that this polygon is a star-shape one, every points can be seen by the original site G . As such, a simple way to triangulate this polygon is to repeatedly remove an ear that does not contain G from the polygon (Figure 3.1b). Doing so we are guaranteed that the result after each step will still be a polygon with similar properties, so we can repeat the process until there are only three or four points left which can be handled easily.
- Inserting a missing site is simple. Now we do not have to worry about shifting a site outside the triangulation, onto the boundary, on an edge, etc. because they will be taken care of automatically by the Inserting missing site step, and we will see later that that step can be very efficiently and robustly performed in parallel.

Parallel processing challenges

To perform this shifting step in parallel, it is easier to first detect all sites that are good cases to shift them first, then handle the bad cases. One difficulty is that whether or not shifting a site is a good case or not depends on the coordinates of the sites on its triangle fan. As such, when we have an edge AB and we want to check if shifting A and B are good case or bad case, we cannot check concurrently, because when we check A , we do not know whether to use the discrete coordinate of B or its original coordinate.

Following our first design principle, we will break the shifting of good cases into multiple rounds. In each round, consider a site A that has not been decided, we check all the the sites on its triangle fan. If these sites are already decided in an earlier round, we can use either their discrete coordinate or continuous coordinate based on their previous decision, i.e. if good case, it should be shifted already, if not, it will be shifted later. However, if a site B on the triangle fan of A is not decided, then only if the index of A is smaller than that of B , we will let A continue, using the discrete coordinate of B , while B will have to wait till the next round and

check again. We will have to repeat this process until all the sites are processed. With this protocol, in each round, we will be able to decide and shift a set of good cases where the sites are not affecting one another.

After the above process, we have been able to shift a huge portion of good cases sites. We are left with those that we considered bad case shifting. As discussed earlier, for each bad case, we will remove the site, patch the polygon formed by the triangle fan of that site, and record the site as a missing site. For two sites that are neighbors (i.e. there is an edge between them), if we process them concurrently, we might run into the case where they want to delete the same triangle at the same time (the triangle that has both of them as vertices). In order to avoid this racing problem without using any explicit lock or synchronization primitive, we employ our first design principle, breaking the process into multiple rounds, each consists of the following steps: Detect the bad case sites that can be processed concurrently using the same protocol that we use when detecting good case shifting sites; Delete all triangles on the triangle fan of each site and patch the hole produced; Mark the site as a missing site.

3.1.2 Inserting Missing Sites

Simplification

We are very concern about the performance of this step. First of all, when the number of sites is huge, there will be a lot of missing sites due to the limited texture size. Secondly, the shifting stage should have introduced a few more sites as missing sites. Soon when the number of sites increase, this step will dominate the running time of GPU-DT.

Fortunately, inserting a site into the triangle mesh is much simpler than shifting a site, because the effect it can cause to the triangulation is very local. There are only several cases that can happen. In the good situation, we can find a triangle which contains the site. This is the simplest case since we only need to triangulate that triangle into three triangles. In case the site we want to insert into lies on an edge between two triangles, we need to subdivide these two triangles into four. The most troublesome case is when our newly inserted site lies outside the triangulation. In this situation, we fail to locate the position of the site, and we

have to patch the convex hull in order to maintain the triangulation. Not only that the process of fixing the convex hull is much more complicated than the work to handle other cases, but it can possibly affect a large region of the triangulation, thus the work to avoid conflicts while we fix the triangulation is much more complicated. Due to our second design principle that we discussed earlier, we want to avoid such situation as far as we can.

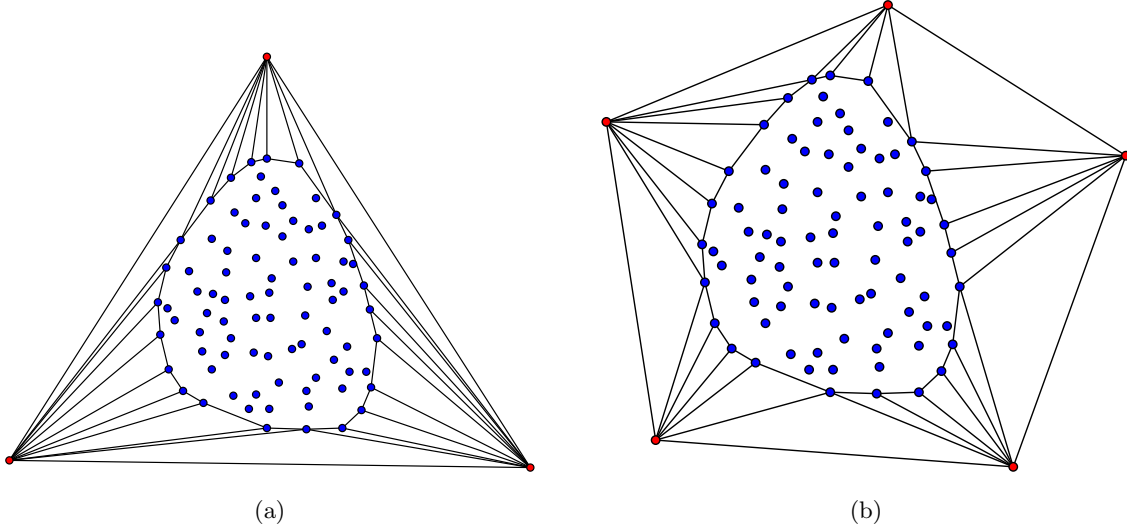


Figure 3.2: Fake boundary (a) A simple 3 new sites (b) More sites to make new triangles bigger

In order to remove such a problem, we want to somehow guarantee that each and every sites we want to insert will lie inside the triangulation. In order to do that, we add a fake boundary that is big enough into the triangulation, to make sure that the boundary of the triangulation covers all sites, including all those missing ones. The idea is that we will insert a few more points far enough onto S such that they form a big boundary containing all the sites in S . We will add these points and new triangles before inserting missing sites, and we can easily remove them after this step. The most trivial way to create the fake boundary is to add three sites very far away such that the triangle formed by these sites covers the whole texture, thus contains all sites. One disadvantage of this method is that when we connect these fake sites to the convex hull of S to form a big triangle mesh, we may end up with a series of very thin triangles, and any shifting of the sites on the convex hull of S will end up being a bad case (Figure 3.2a). The second disadvantage is that let k be the number of sites on the convex hull of S , each of the three fake sites will on average be connected with $\frac{k}{3}$ sites in S , thus making walking around

these sites very slow. To improve this situation, we add \sqrt{k} new sites into S (Figure 3.2b). This not only reduces the number of triangles per site and makes these triangles bigger, but when we want to remove these sites, we can efficiently do that in parallel.

Parallel insertion

For each missing site, we have to locate the triangle that contains it. The insertion of a site can now affect at most two triangles in the triangulation. However, there are still several problems that can happen when we do this in parallel. First of all, when two missing sites try to insert themselves into the same triangle, we cannot let both of them do that concurrently. Secondly, when a missing site is inserted into an edge, it will change two triangles in our triangulation, and we need to make sure that no other thread is trying to update these triangles. In order to avoid all these problems, we employ our first design principle and break the insertion of sites into several rounds. At each round, our goal is to be able to insert as many sites as possible without any conflict. Every round will consist of the following steps:

- Step 1: For each unprocessed missing site x_i , we first try to locate the triangle that it lies in. Note that except the first round, in every subsequent rounds, we can resume the search from the triangle that previously we have found to contain x_i , but because of some conflicts, we have postponed the insertion of x_i until this round. If x_i lies in the triangle a , we write $X[a] = i$. If this site lies on an edge shared between two triangles a and b , we atomically set $X[a] = \min(X[a], i)$ and $X[b] = \min(X[b], i)$.
- Step 2: Using the array X , we will decide whether to insert a missing site in this round or not. For each missing site, it can only be processed in this round if all the tags written by it into X (either one or two tags) are not overwritten.

In CUDA, when several threads try to write to $X[a]$, it is guaranteed that one of the write will succeed. Since the chance for a site to lie on an edge is rather small, using the atomic operation is not very costly. We use the $\min()$ function when we write two tags onto X because we want to avoid possible livelock. In Figure 3.3a, site A will try to set $X[a] = A$ and $X[c] = A$; site B set $X[c] = B$ and $X[b] = B$; site C set $X[a] = C$ and $X[b] = C$. If the result is $X[a] = A$,

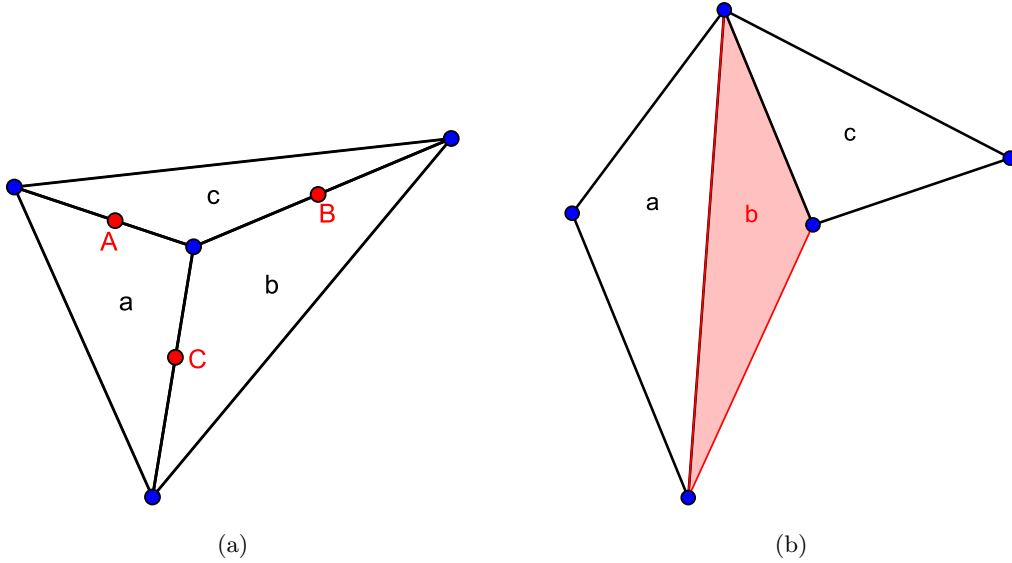


Figure 3.3: Possible conflicts in parallel processing of (a) Inserting missing sites and (b) Flipping triangles

$X[b] = c$ and $X[c] = B$ then none of the three sites can be inserted and we can get into a livelock. Using the $\min()$ function, we are guaranteed that in each round, at least the missing site with the smallest index will be inserted. In practice, when sites are uniformly distributed, they are unlikely to conflict with one another, and thus we need very few rounds. even in the case where a lot of sites fall into one triangle, after the first few insertions, that triangle should be subdivided further, and thus the congestion is reduced.

3.1.3 Flipping

After the inserting missing site step, we need to remove the fake boundary introduced earlier. We can easily perform this step in parallel. Each thread can handle one fake site, removing all triangles connected to it and then fix the corresponding part of the convex hull of S . Since we do not want two threads to try to remove the same triangle, we will also break this process into several rounds and make sure that in each round, no two fake sites that are connected are being deleted. We can employ the same strategy as when we detect good case shifting. Since one fake site is connected to exactly two other fake sites, if we randomize the index of the fake sites, on average we need only two to three rounds. Note that the number of fake sites as well

as fake triangles are quite small (proportional to the number of sites on the convex hull of S), thus this process takes just a small amount of time.

The process of flipping triangles is necessarily done in multiple rounds, because after flipping one edge, another edge might need to be flipped as well. In each round, we will let one thread handle one triangle, checking three neighbors of that triangle to detect possible flipping. Since each flip will affect two triangles, we need to employ similar strategy as inserting missing site stage to avoid possible conflicts (Figure 3.3b).

We note that performing flipping in multiple rounds like this is very expensive. In the shifting stage or inserting missing site stage, the number of threads that need to run in a round decreases rapidly as those that are already able to be executed need not do anything in the later round. Unfortunately it is possible for a triangle to be flipped several time, as long as its neighbors are changed. Fortunately, there is a way to reduce the work as our triangulation converge to the Delaunay triangulation. We notice that if in one round we have already decided that a triangle need not be flipped with any of its three neighbors, then we do not need to process that triangle again unless its neighbors are updated in some later round. We can easily detect the triangles that are flipped in each round, and mark the neighbors of these triangles. In the next round we will only need to reprocess those triangles that are marked. This method can significantly reduce the total work of this step.

3.2 Implementation details

In this section, we will discuss some important implementation detail of our new GPU-DT algorithm. Although most key aspects have already been discussed in the previous section, there are some implementation issues that need to be handled efficiently in order to utilize the power of CUDA and the GPU.

The most important concern is how to update the triangulation. In the GPU, we cannot allocate memory dynamically while threads are running. We always have to preallocate memory and then carefully assign memory regions that will be used by each thread so that they do not overwrite one another. For example, we know that a triangulation of n sites cannot have more

than $2n$ triangles, thus we can preallocate a list of $2n$ items to store the triangle list. However, during the three fixing steps, sometimes we need to delete triangles, and sometimes adding more triangles. We have to carefully maintain the list so that we can utilize the gaps caused by deleted triangles, and not to overflow the triangle list.

A second concern is the datastructure. Throughout the steps, we need to frequently walking from triangle to triangle, or around the triangle fan of a vertex. As such, we have to maintain two datastructures. First, for each triangle we always maintain the link to its three neighbors. Second, for each site, we maintain a link list of all triangles that has that site as a vertex. During the process of deleting and inserting triangles into the mesh, we have to keep updating these datastructure. A better strategy would be to separate this process: We let all threads update the triangle list first, and then we update our two datastructures. We will discuss about these more carefully in the three separate sections below.

3.2.1 Shifting

First step in the shifting stage is to detect good cases. In each round, we let one thread take care of one site. The thread will travel through the triangle fan of the site and use the counterclockwise test to check whether shifting this site will cause any crossing or not. If on our way we find out a neighbor of this site which has a smaller index, we will mark this site as *invalid*, so that we will not process it in this round. Otherwise, in the end we will mark this site as *valid*, and decide whether it is a good case or a bad case. If it is a good case, we will also shift its coordinate.

We use an array $A[]$ to track the processed sites. There is one issue that we have to be careful. If a thread i processes site i and mark it as *processed*, another thread j in the same round might think that that site has already been processed in a previous round, and this can possibly lead to a conflict. That is if i and j are neighbors, and $i < j$, i will assume that j will not be shifted in this round, thus it decides that it is a good case. On the other hand, j seeing i as *processed* can go ahead and shift itself. In order to avoid this problem, in our array $A[]$, we keep the *index* of the round that processes each site, and we can easily detect whether a site

was shifted in an earlier round, or it was just shifted by someone in the same round.

This process will stop when we perform one round of checking, and we cannot process any more site (which mean we have processed all sites). It is possible to count the number of sites processed in each round and add them up until we reach n , but this requires a slow reduction operation. Instead, we use a globally shared flag. At the beginning of each round, the flag is set to 0. Each thread that can shift a site in the round will overwrite this flag with 1. In CUDA if multiple threads try to write into one global variable, exactly one thread will succeed, and no serialization will be performed. Thus, we do not have to worry about a performance hit. In the end of the round, we check this flag and stop if the flag is 0.

The second step in the shifting stage is to shift bad cases. We have already detected the bad cases in the previous step, so in this step we only have to delete them and fix the holes that are left. Since we do not want to delete two neighboring sites at the same time, we will also do this in multiple rounds. We use our familiar strategy to get a set of separating sites to process in one round. What we need to do with each site is to delete the site and all triangles around it, and re-triangulate the hole. Note that for each site, the number of new triangles to be inserted will always be less than the number of triangles deleted, thus we can use the deleted slot to store the new triangles, and mark those unused ones. In order to perform ear-cutting algorithm on a star-shape hole, we need a stack to keep the sites on the polygon. The stack size needed is no more than the number of sites on the polygon, so we can easily preallocate the stack for all threads. Since we never process to neighboring sites, no site can appear in more than one stack, thus the total size of the stacks are safely bounded by n . We repeat the round until all bad case sites are deleted. The steps involved in each round are as follow:

- Detect sites that can be processed in this round. While doing this, we need to travel around the triangle fan of each site, so we will also count the number of vertices around the triangle fan, and compute the stack size needed if we can process this site in this round.
- Having the stack size needed for each thread, we perform a parallel prefix sum computation to compute the offset for each stack in our preallocated array.

- For each sites that can be processed, travel around its triangle fan and mark those triangles as *deleted*. We also push the sites on the polygon onto the stack of the thread.
- Fix the *Vertex array*. Each thread will handle one site. We travel the linked list of triangles that has this site as a vertex, and if a triangle was marked as deleted, we fix the list accordingly. To speed up this step, in the previous step, when deleting triangles, we will also mark those sites that are affected, so that in this step we do not have to go through the list of all sites.
- Patch the holes. Each thread handle one hole, we use the stack created before to perform ear-cutting. Since each site already has a list of triangles that has been deleted by it, it can use these newly available slots to store the new triangles.
- Update the links among triangles. For each newly created triangle, we find its three neighbors by traveling the vertex array of its three vertices. Since the link is bidirectional, we update the the link from the neighboring triangle to these new triangles as well. Note that we do not have to delete the link to the deleted triangles. This is because when we patch the hole, any edge that is adjacent to a deleted triangle would now be adjacent to a new triangle, thus the link will be overwritten.

3.2.2 Inserting Missing Sites

Before we insert missing sites, we need to construct the *fake* boundary as discussed earlier. This can be done as early as when we finish fixing the convex hull of our triangulation. Having the convex hull, we can easily connect them to some *fake* sites to form the fake boundary. We just have to take note not to waste time shifting these sites. As discussed, we will insert \sqrt{k} fake sites where k is the number of sites on the convex hull. This gives the balance between the number of new sites added into the triangulation, and the number of triangles that are adjacent to each new site. The new sites can be positioned evenly around a circle centered at the center of the texture and with big enough radius to cover all sites in S . After that, the new sites will be connected to the convex hull and new triangles will be added to the end of the triangle list.

Since this takes minimal amount of time, it is easier done in the CPU.

Having the fake boundary, inserting sites into the triangle mesh is much easier. One difficulty is that each time we insert one site into the mesh, we either delete one triangle and add three more, or delete two triangles and add four. This means that for each site we need two more slot in the triangle list to store the new triangles. We need to reuse the previously unused deleted slots in the shifting step, or else we might overflow the triangle list. Thus in the beginning of the inserting missing site step, we try to gather all the available slots in the triangle list, either used or deleted slots. To do that, we perform one kernel execution to mark all the unused slots in the triangle list, and then do a stream compaction to have all the index of the unused slots stored in an array.

The insertion of missing sites will then be performed in multiple rounds. Similar to the shifting stage, we will repeat until all sites are processed. Each round will consist of the following steps:

- For each site to be inserted, locate the triangle that contains it. In this step we will also use the strategy discussed in the previous section to detect conflicts.
- For each site that can be inserted in this round, it will need two slots for two new triangles. A parallel prefix sum will help us get the index in the unused triangle list that each thread which will handle a site insertion can use to get two unused slot.
- For each new site, depend on whether it lies inside a triangle or on a triangle edge, delete old triangles and add new triangles into the list. We also mark the sites that are adjacent to a deleted triangle to help speed up the next step.
- Fix the vertex array, similar to the shifting stage.
- Fix the link between triangles, also similar to the shifting stage.

In the end of this stage, we need to remove the fake boundary. Since there are very few sites and triangles to be handled, this step can easily be done.

3.2.3 Flipping

Before we perform this step, there is one important task we need to perform. In the previous steps, we have introduced holes into our triangle list. Since the Flipping step will not delete or insert triangles (or it actually inserts exactly the same amount of triangles that it deletes), we can first pack the triangle list to remove all holes. We can perform a simple stream compaction to pack the triangle list. However, after that the index of the triangles in the list will be changed, and that affects the link between triangles as well as the vertex array. Thus while packing the list, we also note down the new index of each triangle. After that, we can perform one round of fixing the indices of all triangles in the vertex array as well as the link between triangles.

The flipping step will be performed in multiple rounds, and it will end when no more edge can be flipped. In each round, we need to perform the following tasks:

- Detect flippable edge. In this step, each thread process one triangle in the triangle list. Each triangle will check with its three neighboring triangles and perform in-circle test to find flippable pair of triangles. In order to avoid double flipping (two threads flip the same pair of triangles), a triangle will only try to flip with a neighbor if its index is smaller than the neighbor's.
- After using the strategy discussed earlier to avoid possible conflicts, flipping of edges will be performed in parallel. The link between triangles and the vertex array will need to be updated.

As already discussed in the previous section, to speed up, after the first round, later round will only work with triangles that were not flipped in the previous round due to conflicts, and those that are adjacent to a triangle that was flipped in the previous round. There are two other issues that we need to look into. The first one is robustness issue. In order to perform the in-circle test robustly, we need to use the algorithm in (Shewchuk, 1996a). However, that algorithm is too complicated to be performed in the GPU. As a workaround, we use a simple in-circle test for our GPU implementation. In case four sites are almost co-circular, we will note down the triangles, and in the end we will perform recursive flipping on just those triangles on

the CPU using the robust in-circle test.

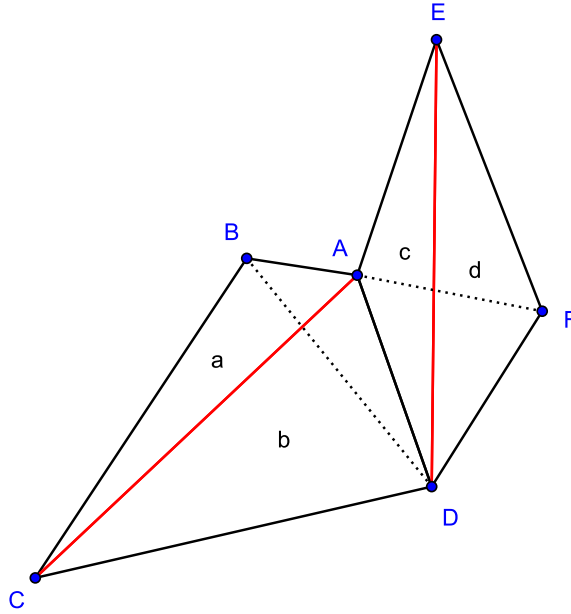


Figure 3.4: Flipping two edges that have adjacent pair of edges

The second issue is how to fix the link between triangles and the vertex array after a flipping round. We ignore the fixing of the vertex array, since we do not need this datastructure anymore. However, that means we cannot fix the triangle neighborhoods by finding neighbors of the newly inserted triangles. Refer to Figure 3.4. Let say edge AC want to flip, and after that triangle a will be $\triangle ABD$ while triangle b will be $\triangle BCD$. If edge ED is not flipping in this round then the problem is simple to solve. Triangle ABD knows that previously the triangle that is adjacent to triangle ACD at edge AD is c , so it can record c as its neighbor. However, it will be more troublesome when edge ED also want to flip in this round, and triangle c becomes $\triangle AFE$ and triangle d becomes $\triangle ADF$. In this case, the neighbor of $\triangle ABD$ on edge AD would be d , not c . To resolve this situation, we break the flipping of triangles into three steps:

- First, each pair of triangles to be flipped by a thread in the current round will update the vertices of the triangles.
- Second, each thread handling two triangles to be flipped will update the link from its four neighboring triangles with the new triangle indices. For example, in Figure 3.4, thread flipping ED will update the neighbor of triangle b on edge AD with the new index d .

Note that this update is done on a temporary array, not the actual datastructure.

- Third, each thread handling two triangles will now try to update their actual links. Triangle b knowing the new neighbor d on edge AD will pass this information to the new triangle a ($\triangle ABD$). Similar thing happens to triangle c and its new neighbor a . If there is no update on the neighbor on edge AD of triangle b (i.e. ED is not flipped in this round), b will pass the old information (triangle c) to the new triangle a .

With the above update strategy, only triangles that are flipping in the current round need to update the link between triangles, and the updating process is simple and does not require any explicit synchronization.

Chapter 4

Experiment results

In this chapter, we will highlight some experiments performed with the newly implemented GPU-DT and compare with the old implementation in (Rong et al., 2008). We will also do some careful analysis to see the effectiveness of our CUDA implementation of Shifting, Inserting missing sites and Flipping. All the tests are performed on a machine with an Intel Core2 Quad Q6600 2.4Ghz and an NVIDIA GeForce GTX280 PCI-X with 1024MB memory. All programs are compiled on Microsoft Visual Studio 2005 and CUDA 2.0 with all optimization options enabled. For each test case, we run on 50 uniformly distributed random input sets and average the running time.

4.1 CPU vs GPU Implementation Stepwise

Figure 4.1 clearly demonstrates the benefit of our CUDA implementation of the Shifting, Inserting missing sites and Flipping steps. The graph shows the comparison between the original CPU and our new GPU implementation using 512^2 texture with number of sites ranging from 10,000 to 1,000,000. We use small textures because on big textures the initial triangulation would be more accurate, thus the time needed for the fixing steps becomes too small. Figure 4.1a shows up to 5x speed up for our new implementation. Note that we have delayed the insertion of the bad case shifting sites until the inserting missing site step. Note also that when the number of sites is more than the number of pixels in the texture, the number of missing sites increases

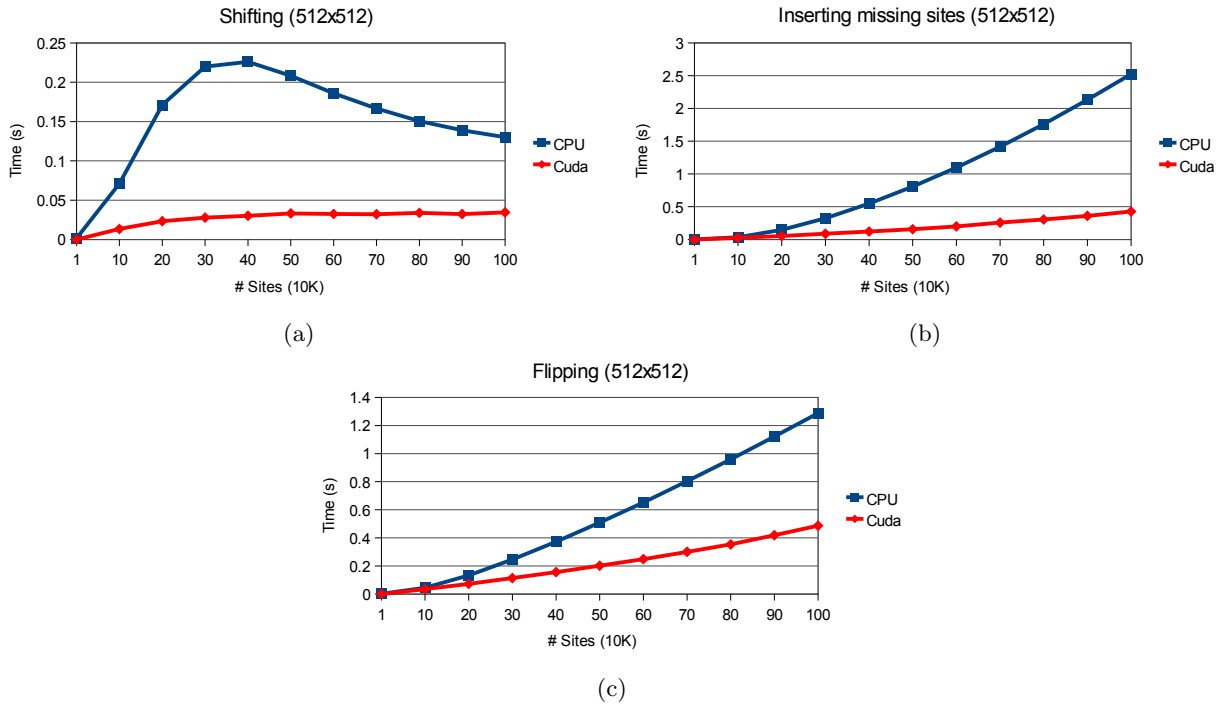


Figure 4.1: Performance comparison between the old CPU and our new GPU implementation

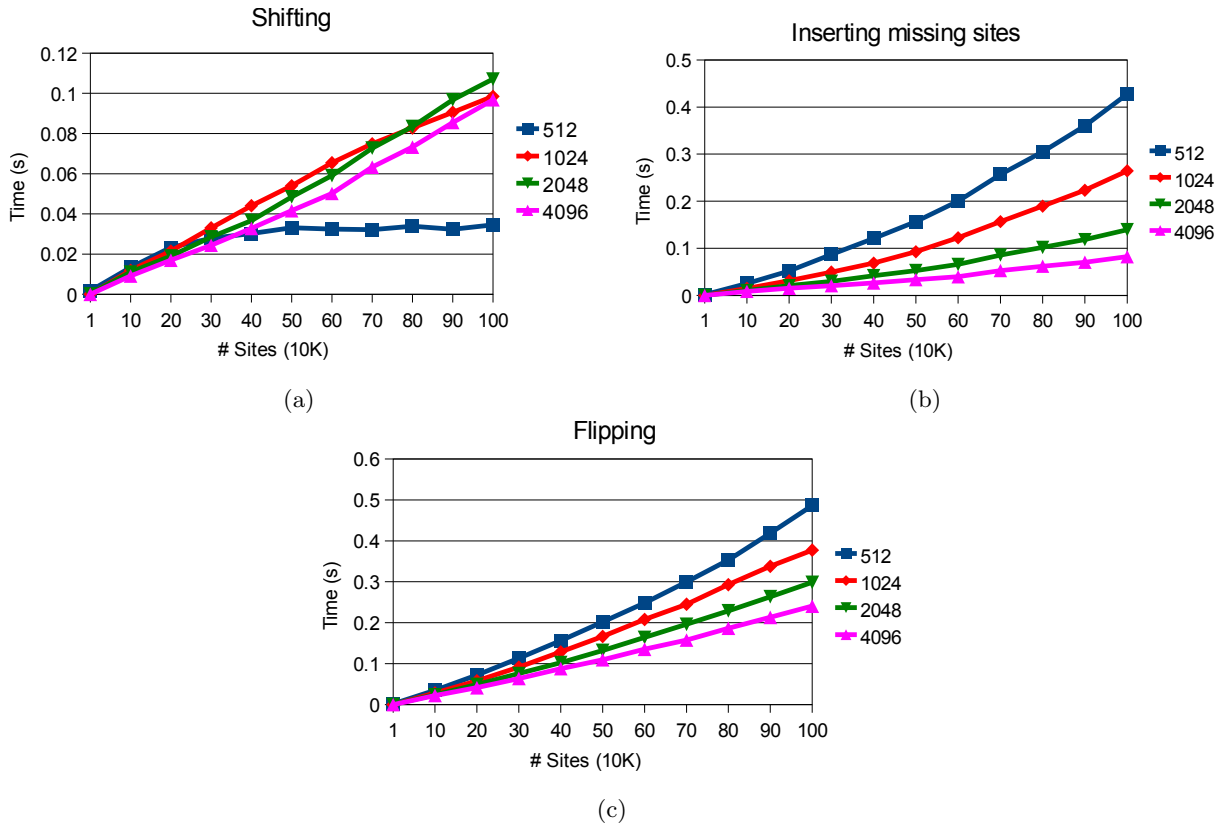


Figure 4.2: Performance of three fixing steps on different texture size

rapidly. Every single pixels in the texture represents a site, so the initial triangulation is very simple. As such, we have more good case shifting and thus the time needed for the shifting step reduces.

Figure 4.1b shows up to 5x speed up for our GPU implementation, with the gradient much smaller than that of the old CPU implementation. This is the result of our use of fake boundary making the inserting missing sites very straightforward. Note that in this step we also have to cover all the bad case shifting sites that were removed in the previous step. The significantly lower growth rate of the running time of this step indicates that our algorithm will work well on bigger input size where there are more missing sites.

Figure 4.1c however only shows a small speed up of up to 3x. One of the reason for the inefficiency of the CUDA implementation of this step is because we need a lot of rounds before our triangulation converges into the Delaunay triangulation. Most of the flippings are done in the first few rounds, while most later rounds can only flip few triangles. This is because when we are near the actual Delaunay triangulation, each time we can only flip a few pair of triangles, and these will cause a few more pairs that need to be flipped. In theory, this behavior should be fine, since the total work needed is still small. However, when we have very few triangles to flip, we can only utilize a few processors in the GPU. One more thing is that in the current CUDA programming model, the overhead of executing a kernel is very high, thus when we run too many rounds, the kernel execution overhead would actually dominate the total running time. This would be improved in the later releases of CUDA.

In Figure 4.2, we try to analyze the performance of our new GPU implementation of three last fixing steps on different texture size. Interestingly enough, the shifting step is almost unaffected by the texture size (Figure 4.2a). Only on the smallest texture 512^2 , from about 300K sites onward, the shifting time is almost constant. The reason is because with such huge number of sites, only about 200K sites can actually be recorded in the texture and processed in this step. In this case, the triangulation we obtained is almost the same, since almost every pixel has one site. On a bigger texture, the running time of this step is about the same, because most of the sites are shifted as good cases anyway, and thus can be shifted at about the same

speed.

Figure 4.2b on the other hand clearly demonstrates the advantage of having big textures. A big texture means that you will have very few missing sites. Also, when we can generate more triangles in earlier steps, we will have less conflict while inserting missing sites, thus it can run faster. Figure 4.2c also show the advantage of using bigger texture. Bigger texture means we can have more accurate discrete Voronoi diagram, thus the initial triangulation would be closer to the Delaunay triangulation, and we need less flipping to transform it to the Delaunay triangulation.

4.2 Overall Comparison

In order to see how our parallel implementation of GPU-DT in CUDA improves the total running time of the algorithm, we compare the total running time of the new implementation with the old implementation using the GPU + CPU hybrid approach. We also compare with *Triangle* (Shewchuk, 1996b), the best known Delaunay triangulation implementation on CPU. Figure 4.3 shows the running time of *Triangle* and GPU-DT on different input size and texture size. Clearly we can see a significant speed up with our new implementation. With small texture such as 512^2 (Figure 4.3a), we can see up to 4x speed up compared to the old GPU-DT. Note that when the texture is bigger (Figure 4.3d), the improvement is much lower. This is because when the texture is small, the two steps of Inserting missing sites and Flipping consume a big portion of the total running time. Our new implementation boosts up these steps by several time, thus the total running time is much lower. On the other hand, when the texture is big, the two steps consume a much smaller amount of time, so in total the running time we saved is small. Note that even on 4096^2 texture, if we keep increasing the number of input sites, we will see huge running time difference between the old and the new GPU-DT implementation. This will be highlighted when we look at the improvement over *Triangle* of our new algorithm. Note also that when the texture is small, previously GPU-DT runs almost 2x slower than *Triangle*. On the other hand, our new GPU-DT runs almost 2x faster than *Triangle*.

To test the robustness of our new implementation and also to get a full picture of the perfor-

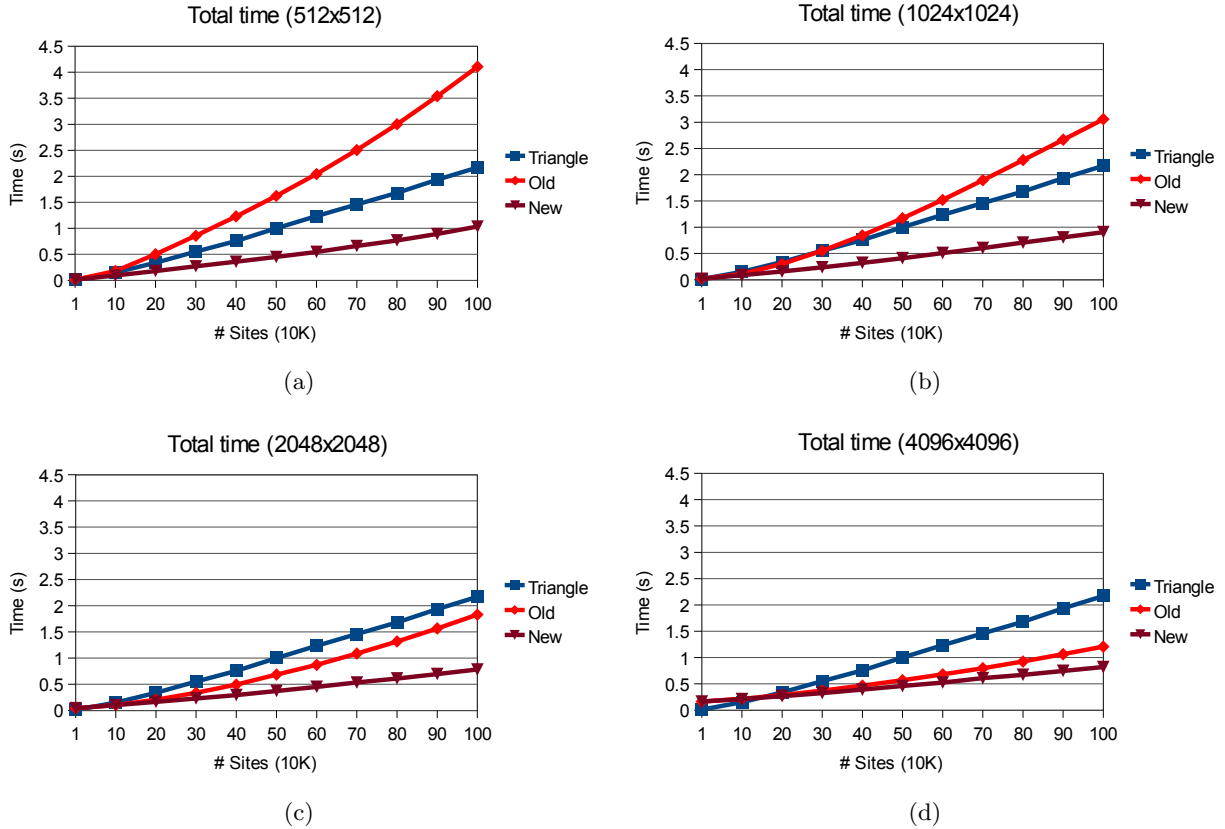


Figure 4.3: Comparison between Triangle, old GPU-DT, and our *new* CUDA implementation

performance of our new GPU-DT implementation compared with *Triangle*, we performed experiments with the number of input sites goes up to 6 millions. Figure 4.4 show the running time of *Triangle* and our new GPU-DT using different texture size. A clear impression is that our GPU-DT runs significantly faster than *Triangle* if we can use a big texture. To see how much faster we are, see Figure 4.5. We compute the improvement of our GPU-DT with *Triangle* by computing the difference in running time divide by the faster running time. Thus, if our running time is 5s and *Triangle*'s running time is 10s, we are 2x faster than triangle, and the improvement is 100% (faster). Overall, our new GPU-DT runs up to 180% faster than *Triangle*. Compared to just about 50% reported in (Rong et al., 2008), we have made a big improvement. Note also that the improvement curve of GPU-DT is now almost always above 0% (except when we run on 512² texture and the number of input is ridiculously big). On the other hand, the old GPU-DT cannot be compared with *Triangle* when such small texture is used.

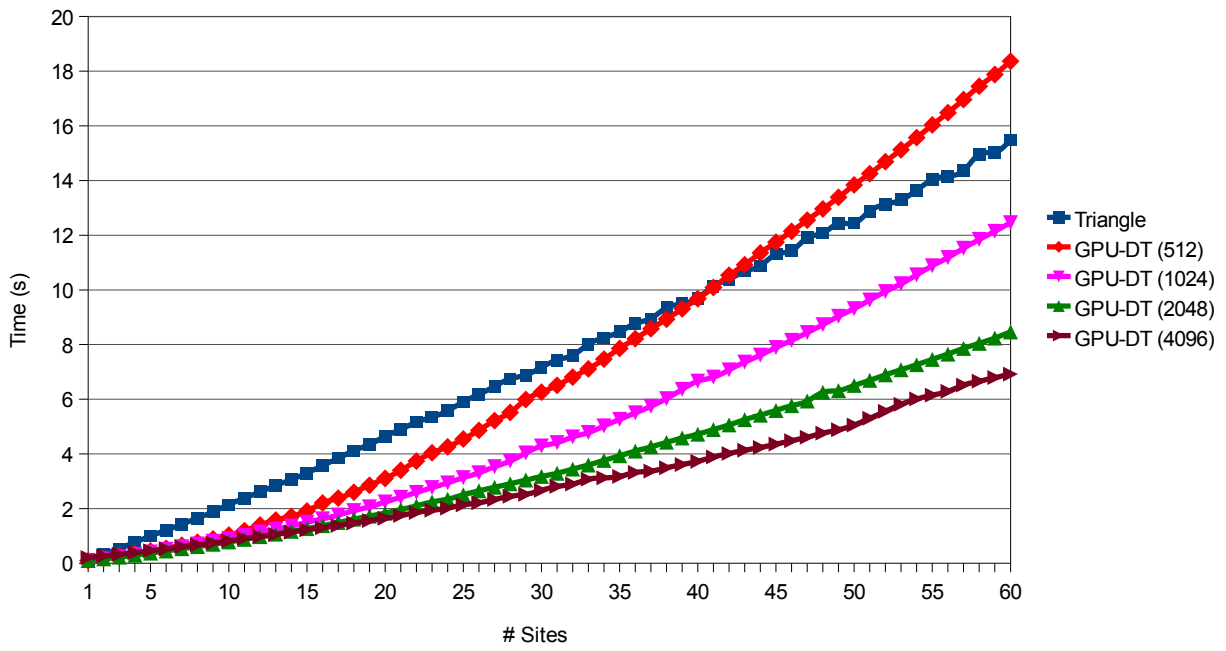


Figure 4.4: Performance comparison between Triangle and our *new* CUDA implementation

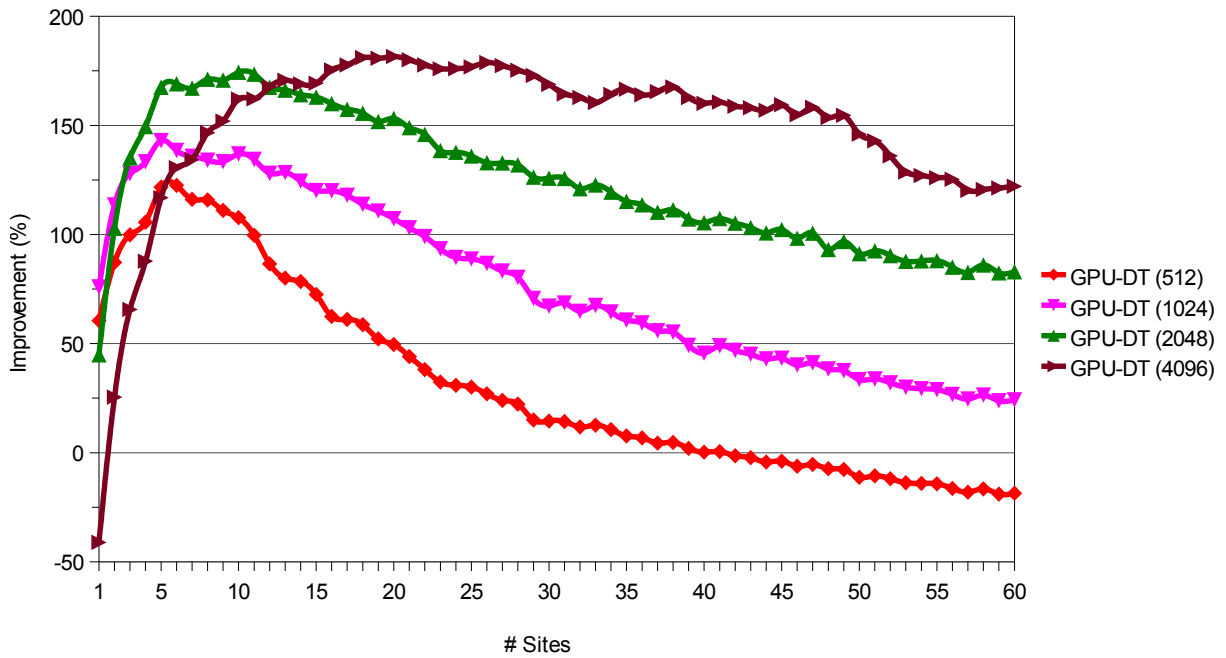


Figure 4.5: Improvement of our *new* CUDA implementation over *Triangle*

Chapter 5

Proof of Correctness

The main concern of the GPU-DT algorithm is the correctness of the triangulation constructed from the discrete Voronoi diagram. The discrete Voronoi diagram used by GPU-DT algorithm is equivalent to that generated by the standard flooding algorithm. In this chapter, we will provide a more rigorous proof showing that the triangulation generated from the discrete Voronoi diagram does not have any crossing, overlapping or hole. This work is done in collaboration with Professor Herbert Edelsbrunner.

5.1 Standard Flooding and Ordered Flooding

We first consider a strengthened version of the standard flooding algorithm, in which a grid point is eligible for coloring only if it is further from the seed point than its neighbors of that color. Induction implies that the pixels are colored in order of distance from the seed point. The algorithm can be reformulated as follows:

Ordered Flooding Algorithm: Sort the entire set of pixel-color pairs by Euclidean distance between pixel and seed point. Scanning the list in order, we let (x, i) be the current pair. If x is yet without color and $x = x_i$ or it has a neighbor pixel with color i then color x with i . Else skip the pair and continue with the next pair.

It is not so easy to see that this algorithm succeeds in coloring all pixels. However, we will prove that ordered flooding is equivalent to standard flooding. It thus follows that also ordered

flooding colors all pixels. This proof was done by Professor Herbert Edelsbrunner. After that, we can use some properties of ordered flooding to show the correctness of GPU-DT.

5.1.1 Euclidean Coloring and Standard Flooding

When we remove a point from S , the Euclidean Voronoi region of a remaining point either stays the same or it grows. The same is true for the regions E_i obtained by Euclidean coloring. Similarly, it is true for the bulk of E_i . However, it is not necessarily true for the regions obtained by flooding. We prove a weaker statement for them, namely that they contain the bulks of Euclidean coloring. It follows that the deletion of a seed point can shrink a flood region only by debris pixels, of which there are generally few.

BULK LEMMA. Each region F_i constructed by the Standard Flooding Algorithm contains the bulk of E_i .

PROOF. We prove that the prefixes of the bulk for E_i obtained by adding the pixels in the order of distance from x_i are connected. It follows that the pixels of the bulk are colored in this same order and with the same color. Let $B_{i,j}$ contain the j pixels of the bulk of E_i closest to the seed point. Thus

$$\{x_i\} = B_{i,1} \subseteq B_{i,2} \subseteq \dots \subseteq B_{i,m} = B_i.$$

Suppose not all of the prefixes are connected and let $B_{i,j} = B_{i,j-1} \cup \{x\}$ be the first that is not connected. Draw the line that passes through x and x_i , as in Figure 5.1. There are neighbors of x on both sides of the line whose distances from x_i are less than $\|x - x_i\|$. On the other hand, x belongs to the bulk of E_i , which is connected, so we can find a path within the bulk that connects x with x_i . Drawing it from pixel center to pixel center with straight edges in between, the path belongs to the Euclidean Voronoi region, V_i , by convexity. Hence, the region surrounded by the path and the straight segment from x to x_i belongs to V_i . This region includes at least one neighbor y of x with $\|y - x_i\| < \|x - x_i\|$. This neighbor precedes x in the ordering of the pixels in the bulk of E_i and thus belongs to $B_{i,j}$, a contradiction to x being separated from $B_{i,j-1}$. □

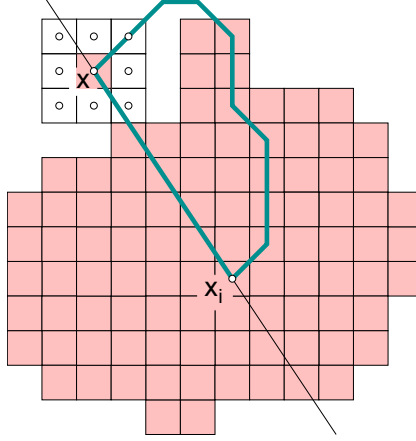


Figure 5.1: The bulk of E_i after coloring the first j pixels by flooding.

5.1.2 Distance Invariant

The Bulk Lemma implies that ordered flooding colors all bulk pixels. We extend this result to all pixels, including debris. We do this by proving that standard flooding colors the pixels of each region in order of distance from the seed point.

Let $F_{i,j}$ be the set of pixels colored i after j steps of the algorithm. We use standard flooding and thus define $C_{i,j}$ as the set of yet uncolored pixels neighboring at least one pixel in $F_{i,j}$. Furthermore, let C_j be the set of pixel-color pairs (x, i) with $x \in C_{i,j}$ over all i . We let $\text{argmin } C_j$ be the pair in the set that minimizes the Euclidean distance between the pixel and the seed point of the color, breaking ties when we need. The index j counting the steps is implicit in the restatement of the algorithm using the above notation.

Set $j = 0$; $F_i = \emptyset \forall i$; $C = \{(x_i, i) \mid \forall i\}$;

repeat $j = j + 1$; $(x, i) = \text{argmin } C$;

$F_i = F_i \cup \{x\}$; update C

until $C = \emptyset$.

We now have all the notation available to formally state the property maintained throughout the algorithm.

DISTANCE INVARIANT. We have $\|x - x_i\| < \|y - x_i\|$ for all $x \in F_{i,j}$, all $y \in C_{i,j}$, and all i and j .

PROOF. We use induction over j . Let y_0 be the first pixel that violates the claimed inequality and let j_0 be the step this violation arises. A *predecessor* of a colored pixel is a neighboring pixel that received the same color earlier. By assumption, at a step $j < j_0$ all predecessors of a pixel colored i are closer to x_i than this pixel. At step j_0 , y_0 has a neighbor with color i as well, but this neighbor, A , is further from x_i , that is, $\|x_i - A\| > \|x_i - y_0\|$. It follows that A is the only neighbor with color i , else y_0 would have contradicted the inequality before step j_0 or it would have been colored before A .

CASE 1. A is the W-neighbor of y_0 . Without loss of generality, assume that x_i lies in the lower right quadrant of y_0 ; see Figure 5.2a. A has a predecessor that is closer to x_i but not neighboring y_0 . The only possibility is the SW-neighbor B . This further constrains the location of x_i to within a 45° wedge; see Figure 5.2a. The S-neighbor U of A must have

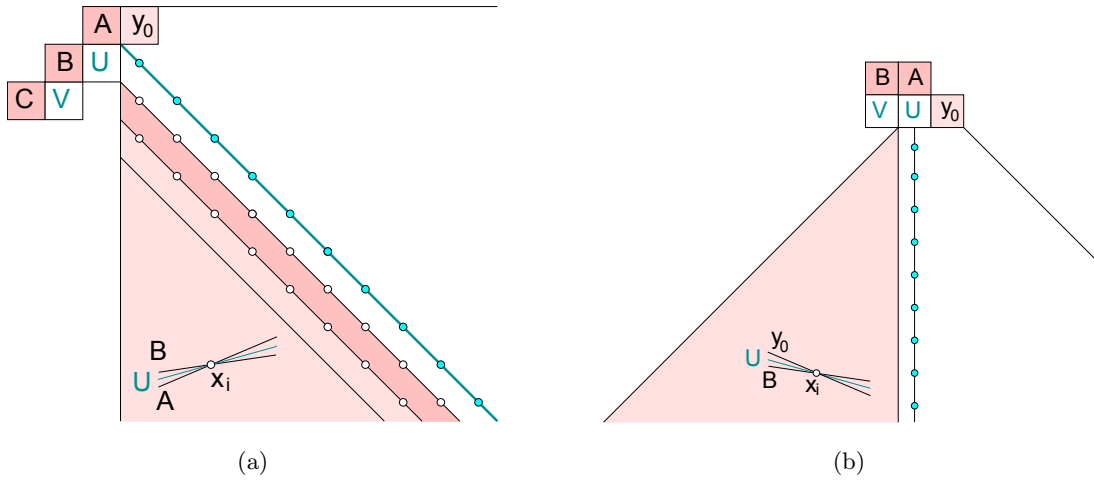


Figure 5.2: Illustration of (a) Case 1 and (b) Case 2, with shaded wedge indicating the possible locations of the seed point, x_i .

been colored before B , else it would have violated the claimed inequality before y_0 did. Hence, $\|x_k - U\| < \|x_i - B\| < \|x_k - B\|$. Similarly, $\|x_k - U\| < \|x_i - A\| < \|x_k - A\|$. Furthermore, we have $\|x_i - A\| < \|x_k - y_0\|$, else y_0 would have been colored before step j_0 . We now have two circle constraints for x_k , namely,

$$\|x_k - A\| > \|x_i - A\|, \quad (5.1)$$

$$\|x_k - B\| > \|x_i - B\|, \quad (5.2)$$

and since U is between A and B as viewed from x_i , this implies $\|x_k - U\| > \|x_i - U\|$. But being closer to x_i than to its own seed is only possible if U has no predecessor in F_i at the time it is colored.

CASE 1.1. The predecessor of B is its S-neighbor, V . Then U must have been colored before V , else color i would have taken precedence over color k . Hence, $\|x_i - U\| < \|x_k - U\| < \|x_i - V\| < \|x_k - V\|$. The half-plane constraint for x_i limits the point x_i within a narrow diagonal strip drawn dark in Figure 5.2a. Similarly, the half-plane constraint for x_k together with $\|x_k - U\| < \|x_k - y_0\|$ obtained earlier constrains x_k to a strip only twice as wide and containing the strip of x_i . In addition to the circle constraint (5.2), we get

$$\|x_k - U\| < \|x_i - V\|.$$

The new constraint can be drawn by translating the circle centered at V to center U . This is a translation along the main diagonal, which leaves only a skinny region of possibilities for x_k . This region contains no integer point other than x_i , hence we arrive at a contradiction.

CASE 1.2. The predecessor of B is its SW-neighbor, C . This further shrinks the wedge of x_i to a diagonal strip twice as wide as the one discussed in Case 1.1. The S-neighbor, V , of B must have been colored before C else it would have contradicted the claimed inequality before y_0 did. But then we can substitute C, V, B for B, U, A and use the same case analysis. We end with a contradiction or repeat the argument again, adding another step to the SW-staircase. The staircase must end too since it leads to x_i .

CASE 2. A is the NW-neighbor of y_0 . Without loss of generality assume that x_i lies in the lower quadrant of y_0 ; see Figure 5.2b. Recall that A is the only neighbor of y_0 colored i and it has a chain of predecessors leading back to x_i . Now we have two possible predecessors of A not adjacent to y_0 , its W- and its SW-neighbor.

CASE 2.1. A has its W-neighbor, B , as predecessor in F_i . This further constrains x_i to a 45° wedge. The S-neighbor, U , of A must have been colored before B , else it would have violated the claimed inequality before y_0 did. Hence, $\|x_k - U\| < \|x_i - B\| < \|x_k - B\|$ and similarly $\|x_k - U\| < \|x_i - A\| < \|x_k - A\|$. We also have $\|x_i - A\| < \|x_k - y_0\|$, else y_0 would have been colored before step j_0 . This gives two circle constraints for x_k , namely

$$\begin{aligned}\|x_k - B\| &> \|x_i - B\|, \\ \|x_k - y_0\| &> \|x_i - y_0\|,\end{aligned}$$

which imply $\|x_k - U\| > \|x_i - U\|$ because U is between B and y_0 if viewed from x_i . But being closer to x_i than to its own seed point, x_k , is only possible if U has no predecessor in F_i at the time it is colored.

CASE 2.1.1. The S-neighbor, V , of B is predecessor of B . Then $\|x_k - U\| < \|x_i - V\| < \|x_k - V\|$. Thus we have $\|x_k - U\|$ smaller than $\|x_k - V\|$ as well as $\|x_k - y_0\|$, which limits x_k to the vertical strip defined by A . The only integer points in this strip lie on the vertical line passing through U . The difference between the squares of the distances from x_k to V and to y_0 on the one hand and to U on the other hand is only one. Both $\|x_i - V\|^2$ and $\|x_i - A\|^2$ are inside this interval of length one, but this is impossible because the difference between the two squares of distances exceeds one. We reached the desired contradiction.

CASE 2.1.2. V is not predecessor of B . Then substitute V for U and repeat the same case analysis. It either ends at a contradiction or repeats again, building another step on the street to the west. But this street must end also because it leads to x_i .

CASE 2.2. A has its SW-neighbor, V , as predecessor. In this case, we may assume that B is not in F_i else we would be in Case 2.1.1. We still have $\|x_k - U\| < \|x_i - V\| < \|x_k - V\|$ and $\|x_k - U\| < \|x_i - A\| < \|x_k - y_0\|$ and hence x_k on the vertical line passing through U . We get the same contradiction as in Case 2.1.1.

□

EQUIVALENCE COROLLARY. The Standard Flooding Algorithm is equivalent to the Ordered Flooding Algorithm, that is, their outputs agree on every input.

Since standard flooding colors all pixels, the equivalence of the two algorithms implies that ordered flooding does too.

5.1.3 Necks and planarity

A *neck* is a pair of diagonally adjacent pixels of the same color whose two common neighbors both have colors that are different from that of the pair. In this section, we prove that it's impossible for our discrete Voronoi diagram to have two diagonally opposite pixels have the same color. This property is useful because it implies that curves drawn within different regions do not cross. We prove this property in the ordered flooding algorithm.

ONE NECK LEMMA. The Ordered Flooding Algorithm produces a coloring in which every square of four pixels has at most one neck.

PROOF. Label the four pixels A , B , U , and V . Assuming two necks, we have the algorithm color the diagonally adjacent pixels A and B with i and the other two diagonally adjacent pixels U and V with $k \neq i$. Without loss of generality, we may assume that A gets colored first and that U gets colored before V . The first assumption implies $\|x_k - U\| < \|x_i - U\|$ and $\|x_k - V\| < \|x_i - V\|$, else U and V would be colored i . The second assumption implies $\|x_k - U\| < \|x_k - V\|$, else we would have a contradiction to ordered flooding. Similarly, $\|x_i - A\| < \|x_i - B\|$ because A gets necessarily colored before B . The assumptions leave three possible sequences, which we discuss in two cases.

CASE 1. U is colored before B . Then $\|x_i - B\| < \|x_k - B\|$, else B would be colored k . It follows that the perpendicular bisector of x_i and x_k separates B on x_i 's side from U and V on x_k 's side. But this implies that A is further from x_i than U is from x_k , a contradiction to ordered flooding.

CASE 2. B is colored second. Ordered flooding implies $\|x_i - A\| < \|x_i - B\| < \|x_k - U\| < \|x_k - V\|$. It follows that x_i is closer to A and B than to U and the same for V . But

there is no integer point with this property, again a contradiction.

□

5.2 Correctness of GPU-DT

The Distance Invariant has one more interesting consequence.

ORDER COROLLARY. The Standard Flooding Algorithm colors the pixels of each F_i in the order of distance from the seed point, x_i .

This implies that we can draw paths from the seed point to every pixel whose Euclidean distance to the seed point increases monotonically along the path.

A *monotonic path* from a seed x till some pixel x_i is a path P_{xx_i} formed by connecting the centers of pixels $x = x_0, x_1, x_2, \dots, x_i$ in the digital Voronoi diagram where each $x_j, j = 1, 2, \dots, i$ has the same color as and is 8-connected to x_{j-1} , and $\|x - x_{j-1}\| < \|x - x_j\|$.

Property 1 Given a seed $x \in S$ and a monotonic path P_{xx_i} from x till some pixel x_i , the (finite) region(s) enclosed by P_{xx_i} and $x_i x$ does not contain any other seed of S .

PROOF. Suppose on the contrary that the finite region R formed by P_{xx_i} and $x_i x$ encloses a

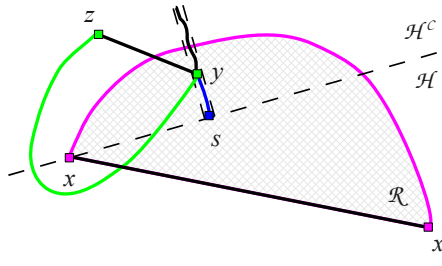


Figure 5.3: A seed $s \in S$ enclosed by a monotonic path P_{xx_i} from x till x_i .

seed $s \in S$. Refer to Figure 5.3. Let the line passing through x and s be ℓ , and let the closed half-space defined by ℓ containing x_i be \mathcal{H} . The complement open half-space of \mathcal{H} is denoted as \mathcal{H}^c . Without loss of generality, we choose s such that $R \cap \mathcal{H}^c$ does not contain any other seed of S . Consider a column of unit width containing s and orthogonal to ℓ . Let the intersection of this column with $R \cap \mathcal{H}^c$ be R_s . By our choice of unit width, R_s contains a continuous path

of pixel centers. R_s cannot be of one color since it has to cross P_{xx_i} . Let y be a pixel in R_s , closest to s but not colored the same as s . y cannot have the same color as x as y is closer to s than to x and it has a predecessor of color s . So, we have y that is colored differently from x and s .

Let y be colored the same as the seed $z \in S$. Again, y is closer to s than to any other seed in \mathcal{H} , so z has to be a seed in \mathcal{H}^c . We can find a monotonic path P_{zy} from z till y . Since P_{zy} cannot cross P_{xx_i} , P_{zy} from z must first enter into \mathcal{H} and then R to reach y . We consider two cases. Case 1, the region R' enclosed by P_{zy} and yz contain some seed, possibly x . In this case, we have reached the same configuration as the given assumption of the claim to be proven but with $\|z - y\| < \|x - x_i\|$. We are done by extremal property on the length $\|x - x_i\|$.

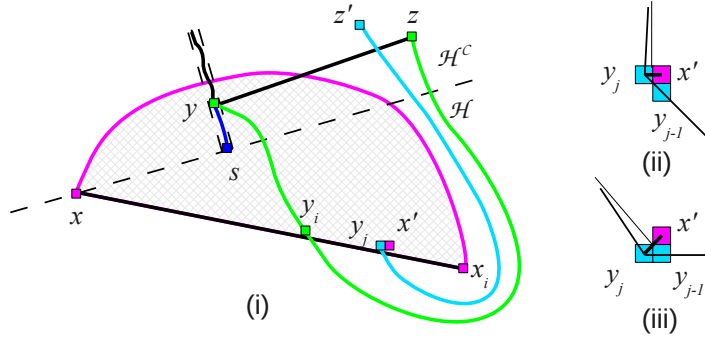


Figure 5.4: No seed enclosed by the monotonic path P_{zy} .

Case 2, R' does not enclose any seed; In this case, R' contains x_i . Consider the set of pixels in R such that any of those has a neighbor not in R and the segment connecting the pixel with such neighbor crosses xx_i . Those pixels form a path P_s from x to x_i , possibly of different colors. Any path that want to cross xx_i to enter R must contain at least one pixel in P_s . Let P_{zy} contains a pixel y_i of P_s . Consider y_j of P_s be farthest from x (possibly the same as y_i), and its successor x' in P_s (Figure 5.4 (i)). Let y_j be colored by a seed z' possibly be the same as z . y_j must be colored before x' or else when we color x' with x we will see one of its uncolored neighbors closer to x than itself, which is impossible. As such, $\|z' - y_j\| < \|x - x'\| < \|x - x_i\|$. If z' is inside the region enclosed by zy and P_{zy} then we can use the same argument as case 1 and we are done. Since z' cannot be in $R \cap \mathcal{H}$, it must either go around y or x_i to reach y_j .

If z' go around y then the region enclosed by $z'y_j$ and $P_{z'y_j}$ contains a seed s and by the same argument as in case 1, we are done. If $P_{z'y_j}$ goes around x_i then it must enclose x' as well. Since y_j is colored before x' , we have $\|z' - y_j\| < \|z' - x'\|$, so z' and x' lies on different sides of the perpendicular bisector of y_jx' . Consider the predecessor y_{j-1} of y_j in $P_{z'y_j}$. For $P_{z'y_j}$ to enclose x' , we need $\angle z'y_jy_{j-1} > \angle z'y_jx'$, thus $\angle z'y_jy_{j-1} > 90^\circ$ (Figure 5.4 (ii), (iii)). This implies that $\|z' - y_{j-1}\| > \|z' - y_j\|$, contradicts to the fact that $P_{z'y_j}$ is a monotonic path. As such, we are again done and the proof is complete. \square

Property 2 Any two edges generated by the algorithm do not intersect.

PROOF. Suppose on the contrary that edge ac intersects bd where $a, b, c, d \in S$ are in clockwise

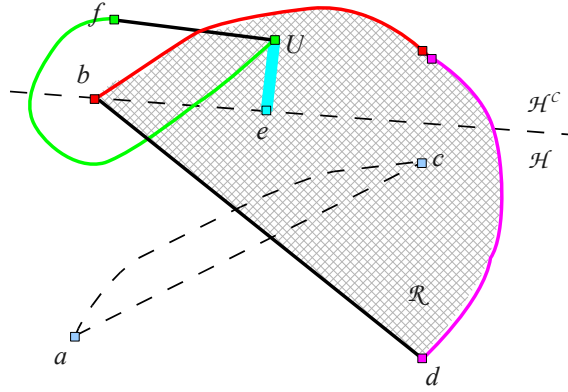


Figure 5.5: ac crosses bd , and we thus have a P_{bd} with db form an enclosed region containing c .

convex positions. Without loss of generality, we show in the following that bd is not generated.

By the existence of ac , we can find a pixel a_i of color a and a pixel c_j of color c such that a_i and c_j are connected. Thus, we have a monotonic path from a till a_i , and a monotonic path from c till c_j . Let us join these two paths together as a path P_{ac} from a to c . Similarly, by the existence of bd , we have a path P_{bd} from b to d . By the One Neck Lemma, these two paths P_{ac} and P_{bd} do not cross. (The case where colors a, b, c, d form a 4-color corner is handled separately and is guaranteed not to create any crossing edges.) Without loss of generality, we assume that P_{ac} crosses bd . As such, P_{bd} cannot cross but to avoid ac . Now P_{bd} together with db form an enclosed region, say R . This region contains the seed c (or similarly, a). Refer to Figure 5.5. From Property 1, such a situation is impossible and we are thus done. \square

Property 3 A corner formed by colors a, b, c in counter-clockwise order also means that seeds a, b, c are in counter-clockwise order.

PROOF. Let the corner be formed by pixels a', b', c' with colors a, b, c , respectively. Suppose on

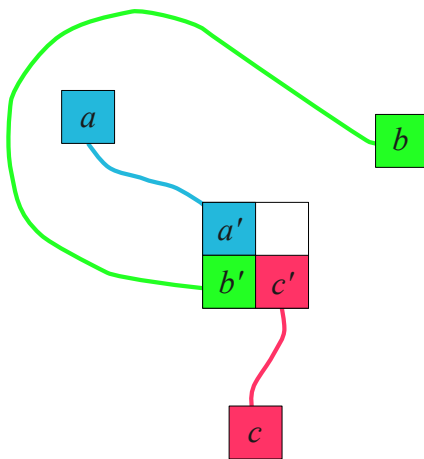


Figure 5.6: A corner of colors a, b, c in counter-clockwise order while a, b, c in clockwise order.

the contrary that a, b, c are in clockwise order. Refer to Figure 5.6. We again use the existence of monotonic paths from seeds here. In other words, we have monotonic paths from a, b, c to a', b', c' , respectively, and path P_{ac} by joining some two monotonic paths a till a' and c till c' . A monotonic path $P_{bb'}$ from b till b' must not cross P_{ac} . As such, $P_{bb'}$ with $b'b$ forms a region enclosing either a or c . Again, with Property 1, such a scenario is impossible and we are done.

□

Property 4 No two triangles generated by the algorithm overlap in area.

PROOF. Because edges of the constructed mesh do not cross, we only need to consider two cases of possible overlapping triangles.

First case, we have two same triangles abc formed by three seeds a, b, c . From the result in the previous section, these two triangles must both be oriented counter-clockwise when a, b, c are oriented counter-clockwise. We must now show that it is impossible to have two different corners generated by a, b, c where these three colors are oriented counter-clockwise. Refer to Figure 5.7. Let one such corner be formed by pixels a', b', c' and the other be a'', b'', c'' . We

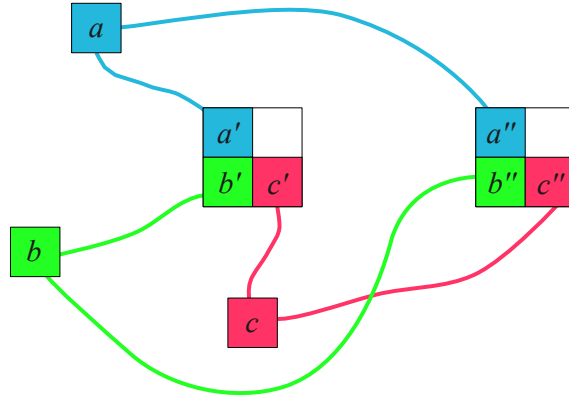


Figure 5.7: Two corners of the same three colors of a, b, c .

have a monotonic path $P_{aa'}$ from a till a' , monotonic path $P_{bb'}$ from b till b' , and monotonic path $P_{cc'}$ from c till c' . Similarly, we must have monotonic paths $P_{aa''}$, $P_{bb''}$, and $P_{cc''}$. Note that $P_{aa'}$ and $P_{aa''}$ cannot be an extension of each other as a corner formed by four pixels of colors a, b, a, c in order does not generate a Voronoi vertex. Likewise, the previous sentence holds for those monotonic paths involving b, b', b'' and c, c', c'' . As specified, $P_{aa'}, P_{bb'}, P_{cc'}$ meet at a corner where color a, b, c are in counter-clockwise order. As such, for $P_{aa''}, P_{bb''}, P_{cc''}$ to meet at a corner where color a, b, c are in counter-clockwise order, there exists an intersection among these six monotonic paths, which is not possible. We are done for this case.

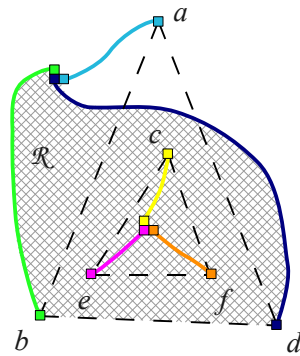


Figure 5.8: Two overlapping triangles abd and cef with possibly $b = e$ and/or $f = d$.

Second case, we have two triangles abd and cef where cef lying within abd (possibly with two identical vertices between them). We just need to consider one configuration and the other cases can be done with the analogous argument. Refer to Figure 5.8. The three monotonic

paths originated from c, e, f are drawn in the figure. Also, the monotonic paths from a, b, d must meet as shown too. Then, the path P_{bd} formed by joining the monotonic paths from b and d , together with c and line segment bd is exactly the same configuration as that in Property 2. With the same argument as in the proof of that property, P_{bd} does not exist, and we are done too. \square

Property 5 *Each edge generated by the algorithm is shared by at most two triangles.*

PROOF. When there are more than two triangles sharing an edge, we have the case of overlapping triangles or crossing edges. This is thus impossible by the above properties, and we are done. \square

Theorem 1 *The proposed algorithm computes the Delaunay triangulation of S .*

PROOF. From the previous properties, we know that the triangulation constructed from our discrete Voronoi diagram has no crossing, overlapping or holes. The Shifting step transforms all the sites back to their original coordinate while maintaining a valid triangulation. Inserting missing sites step adds all missing sites into the triangulation so that we have a complete triangulation of S . Finally, the Flipping step guarantees that our final triangulation is the Delaunay triangulation. \square

Chapter 6

Conclusion and Future Works

In this Honour Year Project, we have discussed the GPU-DT algorithm (Rong et al., 2008) to compute the Delaunay triangulation of a set of points using both the GPU and the CPU. From a discrete Voronoi diagram computed in the GPU using JFA, we construct a triangulation that is the approximation of the Delaunay triangulation. Subsequently, using the CPU, GPU-DT fixes the triangulation by three steps to eventually obtain the correct Delaunay Triangulation. We have revealed many challenges in performing these three fixing steps in parallel in the GPU. While shifting or inserting sites into the triangulation in parallel, it is possible for threads to try to modify the same triangle, thus causing conflicts that can possibly lead to the corruption of the triangle mesh and the failure of the program. We have proposed our solution to efficiently use CUDA to perform these complicated tasks. Our solution is designed based on two principles: Subdivide the work into multiple rounds to avoid implicit synchronization; and simplify the work, possibly breaking it into multiple simpler steps so that it is more suitable to be done in parallel in the GPU. Overall, we have been able to speed up the individual steps several times and in total up to 180% faster than *Triangle*, the best known CPU Delaunay Triangulator. Not only that our improvement makes GPU-DT a lot faster, it also helps GPU-DT running more robust, being able to handle difficult cases that sometime can cause problems to the old GPU-DT implementation.

There are still some weaknesses in our new GPU-DT algorithm. One of which has introduced some performance penalty to our program. Since we want to avoid synchronization, we break

our work into many rounds. The overhead of creating threads and executing kernels at each round seriously affect the performance of our algorithm. This is a limitation of the CUDA programming toolkit, and it will eventually be fixed, potentially improving the performance of our program further. Another limitation of our algorithm is the use of atomic operations in the Inserting missing sites step. Excessive use of atomic operations will reduce the parallelism of our program, thus limit the potential speed up. Our future work would be to design a new strategy to avoid conflict while not relying on atomic operations at all. Also, we will explore the possibility of using OpenCL to replace the CUDA programming paradigm. OpenCL provides a more flexible thread creation and inter-operation, thus can potentially improve the performance of our algorithm.

References

- Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaig, C., & Yap, C. (1988). Parallel computational geometry. In *Algorithmica* 3 (pp. 293–327).
- Cao, T.-T., Rong, G., Stephanus, & Tan, T.-S. (2009). GPU-DT: A 2D Delaunay triangulator using graphics hardware. In <http://www.comp.nus.edu.sg/~tants/delaunay2ddownload.html>.
- Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4), 1972, 132–133.
- Kohout, J., Kolingerová, I., & Žára, J. (2005). Parallel delaunay triangulation in E2 and E3 for computers with shared memory. *Parallel Comput.*, 31(5), 2005, 491–522.
- NVIDIA (2008). CUDA - Compute Unified Device Architecture programming guide 2.0.
- Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., & Purcell, T. J. (2005). A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports* (pp. 21–51), August, 2005.
- Rong, G., & Tan, T.-S. (2006). Jump flooding in GPU with applications to Voronoi diagram and distance transform. *Proceedings of the Symposium on Interactive 3D Graphics and Games* (pp. 109–116), 2006: ACM Press.
- Rong, G., & Tan, T.-S. (2007). Variants of jump flooding algorithm for computing discrete voronoi diagrams. *Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD'07)* (pp. 176–181), 2007.
- Rong, G., Tan, T.-S., Cao, T.-T., & Stephanus (2008). Computing two-dimensional delaunay triangulation using graphics hardware. *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (pp. 89–97), New York, NY, USA, 2008: ACM.
- Shewchuk, J. R. (1996a). Robust Adaptive Floating-Point Geometric Predicates. *Proceedings of the Twelfth Annual Symposium on Computational Geometry* (pp. 141–150), May, 1996: Association for Computing Machinery.
- Shewchuk, J. R. (1996b). Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin, & D. Manocha (Eds.), *Applied computational geometry: Towards geometric engineering*, Vol. 1148 of *Lecture Notes in Computer Science* (pp. 203–222). Springer-Verlag.