FUNDAMENTAL COMPUTATIONAL GEOMETRY ON THE GPU

CAO THANH TUNG
(*B.Comp in Computer Engineering, National University of Singapore, 2009*)

STUDENT ID: HT090409A
SUPERVISOR: DR. TAN TIOW SENG

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

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

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

Cao Thanh Tung
10 June 2014

Acknowledgments

I want to express my gratitude towards all the people that have made this thesis possible. It is thanks to their guidance, collaboration, support and encouragement that my past five-year journey has been fruitful and enjoyable.

A great thanks to my supervisor Dr. Tan Tiow Seng for his continuous, invaluable help and guidance; and also for all those several hours long discussions that form the foundation of my work. He brought me into this field of GPU and parallel programming research since my second year of undergraduate, and he was also the one who reignited my childhood love for geometry. He taught me the integrity of doing research; and his constructive feedback and his perfectionism while preparing manuscripts for conference submission have significantly improved my writing skill.

I would also like to thank Professor Herbert Edelsbrunner for this generous support during my visit to Duke University and my three months internship at the Institute of Science and Technology Austria. I learned a lot from his many in-depth talks during my stay in Austria. His emphasis on intuition and his clarity when illustrating difficult theoretical concepts and proofs have influenced my thinking and research in many ways. I have also received several suggestions from him for the work in this thesis.

I had a pleasure collaborating with several people in the lab, the results of which have been included in this thesis. Thank Tang Ke and Mohamed Anis for helping with my digital Voronoi diagram project. Thank Gao Mingcen for co-authoring with me on the two convex hull works. It really helps to have you around listening to my ideas and sharing your thoughts all the time. I also benefited from doing pair programming with Ashwin Nanjappa for nearly two years. Ashwin is a meticulous developer, and by learning from his programming style, I made far less bugs (and headache) in my subsequent implementations.

Besides those works that are included in this thesis, I am also grateful to have worked together with Qi Meng and Sadegh Nobari on other projects. Qi Meng and I worked together on developing a GPU algorithm for the constrained Delaunay triangulation problem. I also learned about the 2D quality triangulation problem from her. Thank Sadegh for showing me the tenacity in doing research and submitting papers.

During my stay in NUS, I have enjoyed talking with many people. I want to thank Dr. Low Kok Lim and Dr. Cheng Ho-lun for many interesting discussions about computer graphics and computational geometry. Thank my labmates Conrado Ruiz Jr., Kang Juan, Liu Linlin, Le Ngoc Sang, and Poonna Yospanya for making the office much less boring than it would have been. Thank Hua Binh Son for introducing me to the world of photography and old camera lenses.

I am also very fortunate to have met and talked to many people when I visit Duke University and IST Austria. To David L. Millman goes the credit of giving me insights on degree of precision and the possible numerical error in my PBA algorithm. I also thank the Edelsbrunner group in IST Austria for showing me such an intensive research environment and interactions during my stay.

The last, but definitely not the least, thanks I dedicate to my family. Thank mom for letting me leave home at a very early age and for so long. Thank dad for introducing me to the wonderful world of computers and for never saying no to any book that I had ever wanted to read. Thank you, Duong Cam, for being my wife, sharing with me many wonderful moments, and always having confidence in me. Life with you is always so sweet and beautiful.

Abstract

Computational geometry has been an area of study closely linked to computer graphics, computer-aided design, visualization and scientific simulation. Constructing geometric structures such as Voronoi diagram, Delaunay triangulation, convex hull and their variants are among the fundamental problems of computational geometry. Their desirable properties make them useful in many applications such as finite element method, surface reconstruction, collision detection and so on.

From the early days of computational geometry in the 1970s till now, there have been many studies on how to efficiently construct these geometric structures on the CPU. Various algorithmic paradigms to construct them have been designed for both single-core and multi-core systems. Nonetheless, the enormous parallel computation power of the GPU (graphics processing unit) has not been exploited well to solve these problems. One challenge is that constructing these geometric structures requires "global" consideration of all input data. It thus does not map straightforwardly to the GPU architecture that relies on regularized work and localized data to achieve good performance.

In this thesis, we present two approaches for solving these fundamental computational geometry problems on the GPU. In the first approach, we obtain a sketch of the desirable geometric structure in the digital space, followed by deriving an approximation in the continuous space, and finally transforming it into the exact solution. The sketch we use is the digital Voronoi diagram, which we compute using our Parallel Banding Algorithm (PBA) on the GPU. PBA has optimal linear total work, high level of paralelism and excellent memory access pattern. Using this approach, we develop GPU algorithms to construct the 2D and 3D Delaunay triangulation and the 3D convex hull efficiently. Each of these three problems needs a novel approach in order to obtain an approximation from the digital Voronoi diagram and to transform it into the exact solution. In our experiment with synthetic inputs, we obtain more than one order of magnitude speedup when compared to the best available implementations of existing CPU algorithms.

Our second approach combines the incremental insertion technique with local transformations, and in contrast to the first approach, it works completely in the continuous space. Points in the input are inserted in batches, and flipping are

applied in different schedules to get to the final solution. We show that applying this approach to the 2D Delaunay triangulation problem, with the help of several heuristics, yields an even more efficient solution than using the first approach. On the other hand, the 3D convex hull problem needs a novel flipping schedule, while the 3D Delaunay triangulation requires a hybrid approach, with the help of the CPU, to obtain provably correct result. Using this approach, we achieve more than one order of magnitude speedup when compared to existing CPU algorithms, for both synthetic and real-world inputs.

The two algorithmic approaches in this thesis focus on providing a high level of fine-grained parallelism during execution, lacking of which is the main weakness of existing CPU algorithms when adapted to the GPU. In addition, we also discuss some important GPU implementation techniques to achieve high efficiency while remaining robust to numerical error and geometric degeneracy. These techniques mainly focus on reducing thread divergence and random memory access during GPU computation. Overall, this thesis provides a strong foundation for further work on solving computational geometry problems, as well as other problems in general, on the GPU. [1]

[1] The source code of all the implementations in this thesis is fully available at http://www.geomgpu.net

Contents

List of Figures x
List of Algorithms xiii

1 Introduction **1**
1.1 Fundamental computational geometry and applications 2
1.2 Motivations 2
1.3 Contributions 4

2 Background **6**
2.1 Computational geometry 6
2.1.1 Convex hull 6
2.1.2 Voronoi diagram 7
2.1.3 Delaunay triangulation 7
2.1.4 Geometrical relations 9
2.1.5 Flipping 10
2.2 Graphics processing unit 11
2.2.1 Terminology 11
2.2.2 Challenges 12
2.3 Experiment setting 14

3 Related Work **18**
3.1 Digital Voronoi diagram 18
3.1.1 Exact and approximation 18
3.1.2 PRAM algorithms 19
3.1.3 GPU algorithms 20
3.2 Convex hull 21
3.2.1 Sequential and parallel algorithms 21
3.2.2 GPU algorithms 22
3.2.3 Star splaying in $\mathbb{R}^3$ 23
3.3 Delaunay triangulation 24
3.3.1 Sequential algorithms 24
3.3.2 Parallel and streaming algorithms for the CPU 27

4 Digital Space to Continuous Space **29**
4.1 Overview 29
4.2 Digital Voronoi diagram 30

4.2.1 Exact Euclidean distance transform 31
4.2.2 Phase 1: Band sweeping 32
4.2.3 Phase 2: Hierarchical merging 33
4.2.4 Phase 3: Block coloring 34
4.2.5 Complexity analysis 35
4.2.6 3D and higher dimensions 36
4.2.7 Weighted centroidal Voronoi diagram 37
4.2.8 Experiment 39
4.3 Delaunay triangulation in $\mathbb{R}^2$ - The perfect dualization 44
4.3.1 Phase 1a: Digital Voronoi diagram construction 45
4.3.2 Phase 1b: Triangulation construction 47
4.3.3 Phase 2: Shifting 47
4.3.4 Phase 3a: Missing points insertion 50
4.3.5 Phase 3b: Edge flipping 52
4.3.6 Proof of correctness 52
4.3.7 Experiment 53
4.4 Convex hull in $\mathbb{R}^3$ - The digital depth test 57
4.4.1 Phase 1a: Voronoi construction 58
4.4.2 Phase 1b: Star identification 59
4.4.3 Phase 2: Hull approximation 59
4.4.4 Phase 3a: Point addition 60
4.4.5 Phase 3b: Hull completion 60
4.4.6 Proof of correctness 61
4.4.7 Experiment 62
4.5 Delaunay triangulation in $\mathbb{R}^3$ - The difficulties 68
4.5.1 Topological and geometrical difficulties 68
4.5.2 Algorithm using star splaying 70
4.5.3 Experiment 72
4.6 Discussion 74

5 Incremental Insertion with Local Transformation 76
5.1 Overview 76
5.2 3D convex hull revisit 78
5.2.1 Phase 1: Construction 79
5.2.2 Phase 2: Flipping 81
5.2.3 Proof of correctness 82
5.2.4 Experiment 83
5.3 Delaunay triangulation in $\mathbb{R}^2$ and $\mathbb{R}^3$ with adaptive star-splaying 87
5.3.1 Phase 1: Parallel point insertion and flipping 87
5.3.2 Phase 2: Adaptive star splaying 90

5.3.3 Point insertion heuristic 93
5.3.4 Point relocation and the history DAG 94
5.3.5 Compaction versus collection 96
5.3.6 Memory access optimization 96
5.3.7 2D Experiment 99
5.3.8 3D Experiment 102
5.4 Discussion 106

6 Numerical Error and Robustness Issues 108
6.1 Numerical error on digital Voronoi diagram computation 108
6.2 Transforming the point set in DigiDel2D 109
6.3 Exact predicates and symbolic perturbation 110
6.4 Infinity point and the extended triangulation 112

7 Concluding Remarks 113
7.1 Limitations 114
7.1.1 The digital approach 114
7.1.2 The incremental insertion approach 116
7.2 Outlook 117

References 119

List of Figures

1.1 Fundamental geometric structures in computational geometry. 1

2.1 Illustration of one site, its bulk, and its debris. 8
2.2 Stars and links. 8
2.3 The relations between Voronoi diagram, Delaunay triangulation, and convex hull. 9
2.4 The flipping operation in 2D 10
2.5 The flipping operation in 3D 11
2.6 Architecture of the NVIDIA GTX580 GPU [NVIDIA]. 12
2.7 Synthetic point distributions in $\mathbb{R}^2$. 14
2.8 Synthetic point distributions in $\mathbb{R}^3$. 14
2.9 A cropped snapshot of the Delaunay triangulation of one contour map. 15
2.10 Input points from real-world models $\mathbb{R}^3$. 16

3.1 Danielsson's vector template. 19
3.2 Vector templates of some GPU algorithms. 20
3.3 An example in which the algorithm in [TO12] outputs a wrong result. 23
3.4 Star splaying algorithm. 24
3.5 Constructing 2D Delaunay triangulation using divide and conquer. 26

4.1 Using digital space computation to solve computational geometry problems efficiently on the GPU. 30
4.2 Illustration of the three lemmas to compute the exact Euclidean distance transform. 31
4.3 The doubly linked lists embedded on a 2D array. 34
4.4 Centroidal Voronoi diagram. 37
4.5 Illustration of the weighted centroidal Voronoi diagram computation. 38
4.6 Stipple drawing using weighted centroidal Voronoi diagram. 39
4.7 Percentage of running time of the different phases of PBA in 2D with optimized versus unoptimized parameters. 40
4.8 Speedup of PBA using different number of bands for Phase 2. 41
4.9 Performance of PBA in 2D while varying the density of input points. 41
4.10 Running time of different GPU 2D digital Voronoi diagram algorithms, and their speedup over the sequential algorithm. 42
4.11 Running time of different 3D digital Voronoi diagram algorithms, and their speedup over the sequential one. 43

4.12 Percentage of running time of the different phases of PBA in 3D with optimized versus unoptimized parameters. 43
4.13 Duplicate and intersecting triangles when dualizing the digital Voronoi diagram. 45
4.14 Shifting a point may or may not require modifications to the triangulation. 48
4.15 Sample output of DigiDel2D on different point sets. 54
4.16 The running time and speedup of DigiDel2D on uniform and Gaussian point distribution compared to those of Triangle and CGAL. 55
4.17 The running time and speedup of DigiDel2D on some contour datasets. 56
4.18 The running time of different phases of DigiDel2D. 56
4.19 The number of flips performed in Phase 3b of DigiDel2D. 57
4.20 Illustration of Phase 1 of the convex hull algorithm 59
4.21 The digital depth test. 61
4.22 The total running time of DigiHull3D using different grid size on the ball and the thin sphere distribution. 63
4.23 The grid size and the rendering buffer size affect the performance of different phases of DigiHull3D. 64
4.24 The running time and speedup of DigiHull3D over Qhull and CGAL on different test cases. 65
4.25 The speedup of DigiHull3D over Qhull while fixing the total number of points and varying the number of extreme vertices. 66
4.26 The running time of DigiHull3D and its speedup over Qhull and CGAL on different 3D models. 67
4.27 The running time of different phases of DigiHull3D. 68
4.28 The perturbed grid. 69
4.29 The total running time and time breakdown of DigiDel3D on a uniform distribution, using different grid sizes. 73
4.30 The speedup of DigiDel3D compared to CGAL. 73
4.31 The running time comparison between DigiDel3D and CGAL on some real 3D models. 74

5.1 A stuck configuration in 3D when flipping a star-shaped polyhedron. 79
5.2 A result of the Construction phase. 79
5.3 The running time of IncHull3D on different test cases. 83
5.4 The speedup of IncHull3D over Qhull, CGAL, and DigiHull3D. 84
5.5 The speedup of IncHull3D over Qhull when fixing the number of points and varying the number of extreme vertices, compared with that of DigiHull3D. 85
5.6 The running time of IncHull3D and its speedup over Qhull, CGAL and DigiHull3D on different 3D models. 86
5.7 The time spent on different phases of IncHull3D. 86

5.8 At the end of the point insertion and flipping phase of our IncDel3D algorithm, less than 0.05% of the facets, shaded in the figure, are locally non-Delaunay. 90
5.9 Illustration of the adaptive star splaying algorithm in 2D. 91
5.10 Constructing the convex star of s in $\mathbb{R}^2$ lifted to $\mathbb{R}^3$. All vertices shown in the figure are already lifted. 92
5.11 The running time and the number of stars involved of IncDel3D on uniform point distribution when using different point insertion strategy. 94
5.12 A history DAG of flipping in 3D. 95
5.13 The time breakdown of IncDel2D with and without sorting. 97
5.14 The self-sorting data structure for 3D Delaunay triangulation. 98
5.15 The time breakdown of IncDel3D with different data reordering strategies. 98
5.16 Comparing two different strategies for Phase 1 in IncDel2D. 100
5.17 The speedup of IncDel2D compared to Triangle, CGAL and DigiDel2D. 100
5.18 The number of flips performed by IncDel2D versus DigiDel2D. 101
5.19 The running time of IncDel2D and its speedup over Triangle, CGAL, and DigiDel2D on the contour datasets. 102
5.20 The number of flips and the number of failed vertices of IncDel3D using the Insert-Flip strategy compared to the InsertAll-Flip strategy. 103
5.21 The speedup of IncDel3D compared to CGAL and DigiDel3D on different point distributions. 104
5.22 The running time and speedup of IncDel3D on different 3D models. 105
5.23 The time breakdown of IncDel3D with different point distribution. 105
5.24 The percentage of stars involved in Phase 2. 105

6.1 Numerical error while checking if a and c dominates b on the given column. 109

7.1 Two problems associated with the input points being shifted in digital space. 115
7.2 An illustration of a situation where flipping is serialized. 116

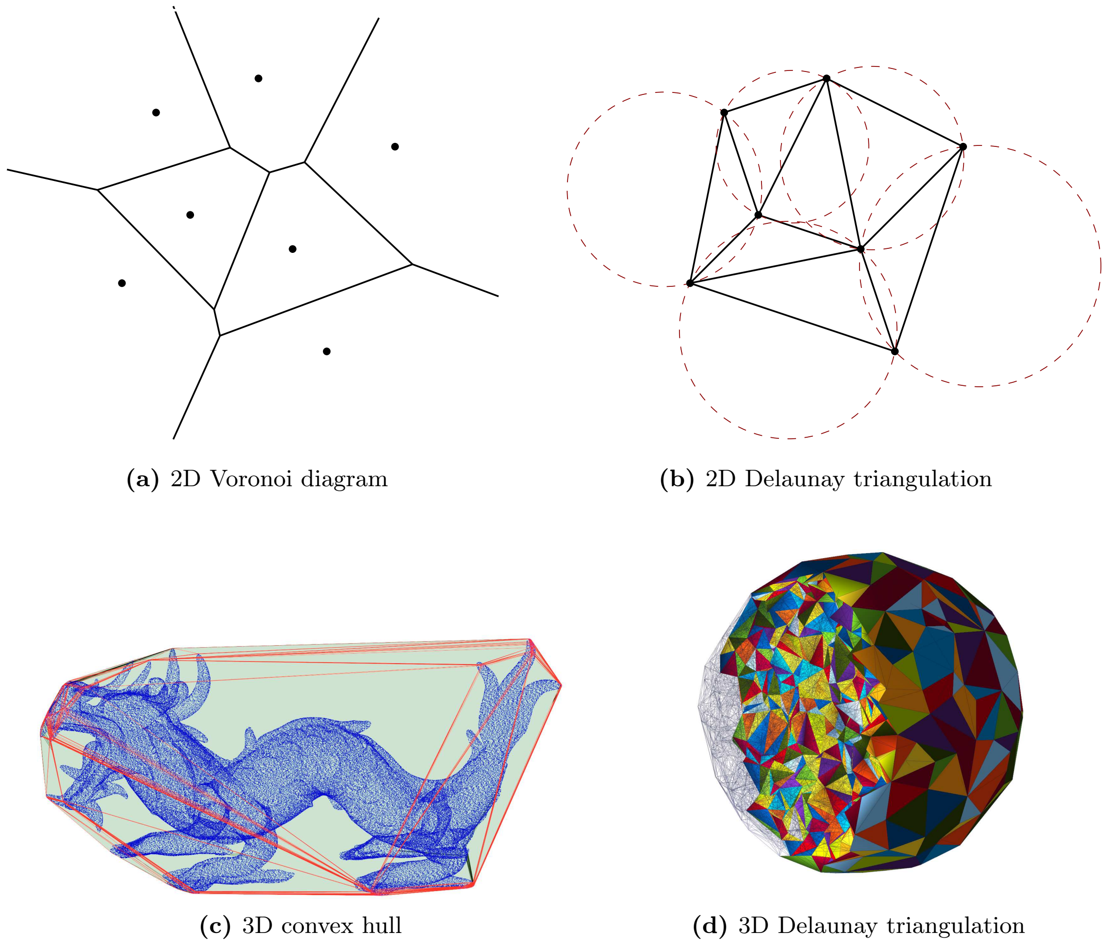
List of Algorithms

4.1 Merging the result of two adjacent bands. 33
4.2 The Standard flooding algorithm. 46
4.3 Recolor the debris in the digital Voronoi diagram. 46
4.4 Shifting points of good cases and recording points of bad cases. 48
4.5 Deleting points of bad cases. 49
4.6 Inserting missing points. 51

5.1 Traditional incremental insertion paradigm. 76
5.2 Parallel incremental insertion with local transformation approach. 77
5.3 Constructing a star-shaped polyhedron. 80
5.4 The Flip-Flop algorithm. 82
5.5 Incremental insertion and flipping to construct the Delaunay triangulation (version 1). 88
5.6 Incremental insertion and flipping to construct the Delaunay triangulation (version 2). 89
5.7 Construct the history DAG from the list of flips. 95

Chapter 1

Introduction



(a) 2D Voronoi diagram

(b) 2D Delaunay triangulation

(c) 3D convex hull

(d) 3D Delaunay triangulation

Figure 1.1: Fundamental geometric structures in computational geometry.

Some fundamental computational geometry problems deal with constructing Voronoi diagram, convex hull and Delaunay triangulation; see Figure 1.1. These structures are widely used in various fields such as computer graphics, computer-aided design, visualization and scientific computation. In this chapter, we give a brief introduction to these fundamental geometric structures, and the motivation as well as the contribution of this thesis.

1.1 Fundamental computational geometry and applications

The *Voronoi diagram* of a point set is a partitioning of the space into cells each associated with an input point. Each point in a cell has the corresponding input point as its closest neighbor. A special type of Voronoi diagram is the (possibly *weighted*) *centroidal Voronoi diagram* in which each site lies exactly at the centroid of its Voronoi cell. These structures have been used in clustering [Aur91] and domain partitioning for various applications such as massively multiplayer online games [Tum04] or peer-to-peer virtual environment [AK12]. Its digital version is closely related to the Euclidean distance transform, a very important structure in the field of image processing and computer vision [Cui99]. The Voronoi diagram is usually obtained by dualizing the Delaunay triangulation, since algorithms to construct the Voronoi diagram usually has lots of numerical error and robustness issues.

The *convex hull* of a point set is the smallest convex set covering the input points. Convex hull is a good form of bounding volume that is useful when checking for intersection or collision between objects [LZB08]. In robotics, it is used to approximate robots and obstacles for the purpose of path planning [MS97]. In general, convex hull is also a useful tool in biology and genetics [WLYZ+09] and object recognition [HH06].

Delaunay triangulation is the dual graph of Voronoi diagram. It is widely used in practice due to many of its desirable properties. For example, in Geographical Information System (GIS), one way to model the terrain is to interpolate the data points based on the Delaunay triangulation [Kre97]. In path planning, the Delaunay triangulation can be used to compute the Euclidean minimum spanning tree of a set of points, because the latter is always a subgraph of the former [PS85]. The Delaunay triangulation is also often used as the starting point to build quality meshes for the finite element method (FEM) [HDSB01]. An essential step in FEM is to discretize the input domain into simple elements such as triangles or tetrahedra, and the numerical error of the whole computation depends on the geometric shapes and the quality of the elements. In $\mathbb{R}^2$, the Delaunay triangulation avoids skinny triangles, while in $\mathbb{R}^3$ it can minimize the containment radius of the tetrahedra. These are invaluable properties for mesh generation.

1.2 Motivations

Given the usefulness of these fundamental geometric structures, many algorithms have been designed to compute them efficiently. Several algorithmic paradigms have been proposed, including incremental construction, divide-and-conquer, plane sweeping, and incremental insertion. Many programs are available to solve computational geometry problems, including *Triangle* [She96a], *CGAL* [CGA] and others [Eri99]. To achieve even higher performance, parallel algorithms are also designed for both distributed and multi-core systems. For

distributed systems, the common approach is to partition the input domain into many small parts, each to be solved independently in a separate computing node, before the results are combined. For multi-core systems, the approach is to start with a coarse structure constructed from a subset of the input points, and then the rest of the points are inserted in parallel to construct the desired structure. With this approach, locking is necessary to guarantee the correctness of the algorithm, and sometimes rolling back is unavoidable.

While systems with multi-core processors are widely available nowadays, they are usually limited to having only 4 to 8 cores. On the other hand, with the development in recent years, the graphics processing unit (GPU) is no longer limited to just for rendering and graphics processing. With the introduction of more flexible programming frameworks such as CUDA [NBGS08] and OpenCL [LKS+10], a growing number of general purpose problems can be solved using the GPU. The GPU provides an enormous computing power, often exceeding that of the CPU. This is achieved by a massively parallel architecture, using hundreds to thousands of processing elements to execute thousands to millions of computing threads simultaneously.

Together with the development of the GPU, there has been a growing interest in GPU solutions for computational geometry problems. Existing algorithms for distributed and multi-core systems do not perform very well on the GPU. First of all, the amount of RAM of a single GPU is about the same as that of a CPU node, so it can only handle a moderate problem size. As such, given the huge number of processing elements on the GPU, the domain partitioning approach for distributed systems generally does not work. The reason is that the number of parts to be partitioned into is too large, leading to parts with very small size. As a result, balancing the load in each processing element and merging the results afterward become prohibitively expensive. Algorithms for multi-core systems are also not applicable, because with the growing number of processing elements, explicit locking becomes very inefficient, if not impossible given the nature of the GPU scheduler. In general, exploiting the enormous parallel computing power of the GPU requires a carefully designed, fine-grained parallel algorithm with regularized work on localized data.

There have been some works that attempt to harness the computing power of the GPU. These include the earlier work of Hoff *et al.* [HKL+99], Rong *et al.* [RT06] and Schneider *et al.* [SKW09] to compute the digital Voronoi diagram; and the more recent works by Stein *et al.* [SGES12] and Tang *et al.* [TZTM12] to compute the 3D convex hull. However, these algorithms are either only able to produce approximate result, not robust enough to handle degeneracy, or hybrid with a major amount of work still being done on the CPU.

The goal of this thesis is to find new algorithmic approaches to use the GPU effectively to solve some major fundamental computational geometry problems. The new approaches should promote fine-grained, wait-free parallel algorithms, and thus are scalable to the increasing number of processing elements on the GPU. These algorithms should also be

provably correct, and are able to handle degeneracy, an inherent problem in computational geometry. More importantly, they should be practically implemented and achieving good speedup compared to the best CPU programs available.

1.3 Contributions

This thesis proposes two algorithmic approaches for designing GPU algorithms to solve some fundamental computational geometry problems.

1. The first approach uses computation in the digital space to approximate result in the continuous space. A general algorithm consists of three phases, from obtaining a sketch, to an approximation, then to the solution. Four major works are presented to demonstrate this approach.

    - We present the *Parallel Banding Algorithm* to compute the exact digital Voronoi diagram on the GPU. The novelty comes from a careful partitioning of the input grid into bands to allow concurrent computation, and an efficient merging of sub-results through clever manipulation of doubly linked lists embedded on a grid. The algorithm outperforms all sequential CPU algorithms in $\mathbb{R}^2$ and $\mathbb{R}^3$, as well as existing GPU-based approximate algorithms. We also show how to obtain the centroidal Voronoi diagram efficiently and accurately; see Section 4.2 and [CTMT10].

    - Using the 2D digital Voronoi diagram as a sketch, we show how to dualize it into a geometrically and topologically valid triangulation, which is an approximation of the 2D Delaunay triangulation. After that, we present a two-step transformation to obtain the desired result. All the steps are done in parallel on the GPU with very high level of parallelism. Our implementation outperforms the best CPU implementations currently available by up to 4 times in speed; see Section 4.3 and [QCT13].

    - We exploit the relation between the 3D Voronoi diagram and the 3D convex hull to compute the latter from the former. More specifically, by computing six slices of the 3D digital Voronoi diagram, all together forming a box enclosing the input point set, we get a good sketch from which we can derive a good approximation of the convex hull. Some extreme points neglected due to the use of digital approximation are added back using a digital depth test followed by a walking approach in the continuous space. The final convex hull is obtained using the star splaying algorithm [She05] on the GPU; see Section 4.4 and [GCN+13].

    - Dualizing the 3D digital Voronoi diagram is significantly more difficult than with the 2D case. We show that it is possible to obtain a geometrically and topologically

valid triangulation, but at a high cost. At the same time, we show that it is also possible to use the star splaying algorithm in a similar way to the convex hull solution, but the efficiency is limited; see Section 4.5.

2. The second approach adapts the traditional incremental insertion technique in a novel, massively parallel manner. In contrast to the computation in the first approach, that in the second approach is done solely in the continuous space. Points in the input are inserted in batches to form an initial structure, and flipping is applied in various schedules in an attempt to obtain the final solution.

    - We revisit the 3D convex hull problem. A novel flipping process called Flip-Flop is proposed to guarantee the algorithm always produces the correct result. With a combination of flipping both reflex edges and convex edges in a clever schedule, we can remove all non-extreme vertices and obtain the convex hull; see Section 5.2 and [GCTH13].

    - We propose a hybrid algorithm to compute the 3D Delaunay triangulation efficiently. Using the GPU, we first insert points in batches, each followed by a series of flipping passes to get closer to the Delaunay triangulation. Although flipping alone cannot always lead us to the correct result, what it achieves is close enough. With the help of a modified star splaying algorithm, applied adaptively on the CPU, we can always get to the correct result. The work done on the CPU is often minimal. Some heuristics are also proposed to further reduce the work of the star splaying step on the CPU, as well as reducing the number of flips performed on the GPU. As such, our hybrid algorithm outperforms all existing sequential CPU algorithms by up to an order of magnitude, in both synthetic as well as real-world inputs. We also adapt the approach to the 2D problem and obtain similar speedup; see Section 5.3 and [CNGT14].

The thesis also includes many implementation details and techniques for efficient implementation of computation geometry algorithms on the GPU. These include optimization techniques to reduce thread divergence and random memory access, which are key factors that affect the performance of GPU code. Furthermore, some techniques are required to guarantee the robustness of the implementation against both numerical error and geometric degeneracy.

Chapter 2

Background

This chapter starts by describing the basic geometric structures, their relations and some of the important properties that are useful for understanding this thesis. The frequently used flipping operation is also described here. For a more complete understanding of these concepts, please refer to the Dutch Book [BCKO08]. We also briefly describe some relevant aspects of the GPU architecture and the important considerations when designing algorithms for the GPU. At the end of the chapter, we summarize the system configuration and input datasets to be used in all the experiments in this thesis.

2.1 Computational geometry

Fundamental computational geometry problems often begin with a given set of points S. We are interested in three main geometric structures: the Voronoi diagram, the convex hull and the Delaunay triangulation. They are all related to one another, as we shall see later. For simplicity, we assume that the input points are in general position, i.e. no three points are collinear, no four points are cocircular in $\mathbb{R}^2$ or coplanar in $\mathbb{R}^3$, no four points are cospherical, and so on. When discussing the implementation details we will show how to deal with such degeneracy in practice.

In the following discussion, let $S = \{s_1, s_2, \dots, s_n\}$ be the set of input points in $\mathbb{R}^d$.

2.1.1 Convex hull

Definition 2.1. The *convex hull* $\mathcal{C}(S)$ of S is the smallest convex set containing S.

For simplicity, we usually refer to only the boundary of the convex hull. In $\mathbb{R}^3$, $\mathcal{C}(S)$ is a convex polyhedron. If points in S are in general positions, then all the facets of $\mathcal{C}(S)$ are triangles. Each point of S on the boundary of $\mathcal{C}(S)$ is called an *extreme vertex*. The boundary of the convex hull can be divided into two parts, the *upper hull* and the *lower hull*. A facet of the convex hull is in the upper hull if the space above it (in a pre-defined direction) is outside the convex hull; otherwise it is in the lower hull.

2.1.2 Voronoi diagram

Definition 2.2. The *Voronoi diagram* $\mathcal{V}(S)$ of S is a tessellation of the space into n cells, one for each input point. A point p lies inside the cell of the input point $s \in S$ if and only if it is as close to s as to other points in S.

The term "close" here usually refers to the Euclidean distance, but can also mean other metrics such as the Manhattan distance or the L_∞ distance. The cell corresponding to the input point s is called the *Voronoi cell* $\mathcal{V}(s)$ of s. Such a cell can either be bounded or unbounded. A Voronoi cell $\mathcal{V}(s)$ is unbounded if and only if s is an extreme vertex.

In $\mathbb{R}^2$, two Voronoi cells intersect at a *Voronoi edge*, and three Voronoi cells intersect at a *Voronoi vertex*. In $\mathbb{R}^3$, two Voronoi cells intersect at a convex polygon, called a *Voronoi face*, while a Voronoi edge or a Voronoi vertex is the intersection of three or four Voronoi cells, respectively. By definition, a Voronoi vertex is of equal distance to the input points corresponding to the Voronoi cells incident to it.

The *digital Voronoi diagram* of a point set S is the digitized version of the Voronoi diagram. We define it over a grid $\mathcal{G}$ of size $M = m^d$, where the input point set S is a subset of the grid points.

Definition 2.3. The *digital Voronoi* cell $\mathcal{V}_D(s)$ of $s \in S$ is the set of all grid points in $\mathcal{G}$ that are closer to s than to any other points in S. The collection of all the digital Voronoi cells of points in S together forms the digital Voronoi diagram of S.

In case there are two input points with equal distance to a grid point, we use their indices to decide. If the grid point p is in $\mathcal{V}_D(s)$, then we say that p is *colored* by s; this is from how we usually visualize the digital Voronoi diagram. From the definition, all grid points in $\mathcal{G}$ are colored.

It is interesting to note that although $\mathcal{V}(s)$ is always connected, $\mathcal{V}_D(s)$ might not be; see Figure 2.1. $\mathcal{V}_D(s)$ has one connected component (called *bulk*) which is path-connected to s, and possibly some *debris* which is disconnected from s. This is simply due to digitization error, and usually is not a significant issue since most applications of the digital Voronoi diagram only use the distance map. Nonetheless, there are some serious topological problems when we dualize the diagram, as we shall see later in Section 4.3.1.

2.1.3 Delaunay triangulation

Definition 2.4. In $\mathbb{R}^2$, a *triangulation* $\mathcal{T}(\mathcal{S})$ is a subdivision of $\mathcal{C}(S)$ into triangles whose vertices are points in S. Two different triangles in $\mathcal{T}(\mathcal{S})$ only meet at a common vertex or edge. The *Delaunay triangulation* $\mathcal{D}(S)$ is a triangulation of S such that the circumcircle of any triangle in $\mathcal{D}(S)$ does not enclose any other points in S.

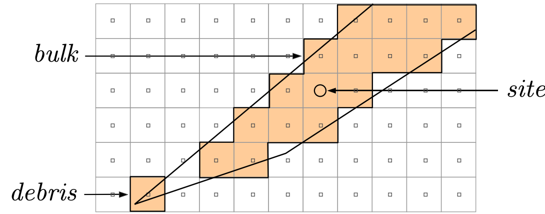
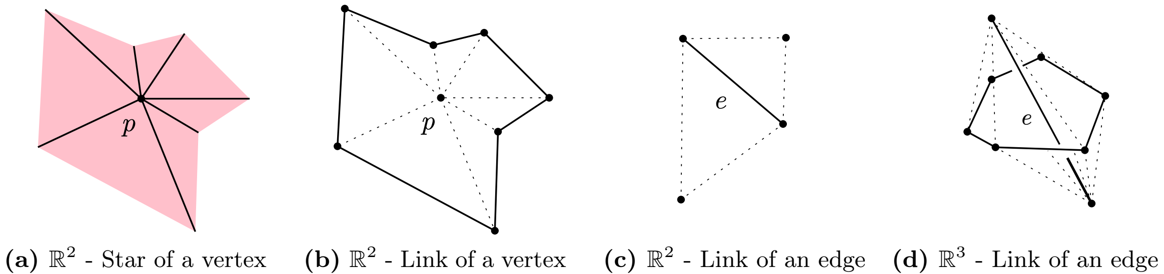


Figure 2.1: Illustration of one site, its bulk, and its debris.

Triangles in $\mathcal{D}(S)$ are said to satisfy the *empty circle* property; see Figure 1.1b. Conversely, any triangle satisfying the empty circle property is said to be a Delaunay triangle. It can be proven that the Delaunay triangulation always exists uniquely for a point set in general position.

In a 2D triangulation, the *star* of a vertex p is the set of all triangles and edges incident to p. The *link* of p is the set of all edges incident to the triangles of the star of p but not containing p. Similarly, the star of an edge is the (up to) two triangles incident to it, and the link of an edge is the (up to) two vertices opposite the edge in these two triangles. Each vertex in a link is called a *link point*. See Figure 2.2 for an illustration. Note that an edge on the boundary of the triangulation (i.e. on the convex hull) has only one triangle incident to it, and thus only one link point.



(a) $\mathbb{R}^2$ - Star of a vertex **(b)** $\mathbb{R}^2$ - Link of a vertex **(c)** $\mathbb{R}^2$ - Link of an edge **(d)** $\mathbb{R}^3$ - Link of an edge

Figure 2.2: Stars and links.

Definition 2.5. Given a triangulation $\mathcal{T}$, an edge $e \in \mathcal{T}$ is said to be *locally Delaunay* if and only if it has only one link point, or each circumcircle of the triangle formed by e and each of its link point does not contain the other link point.

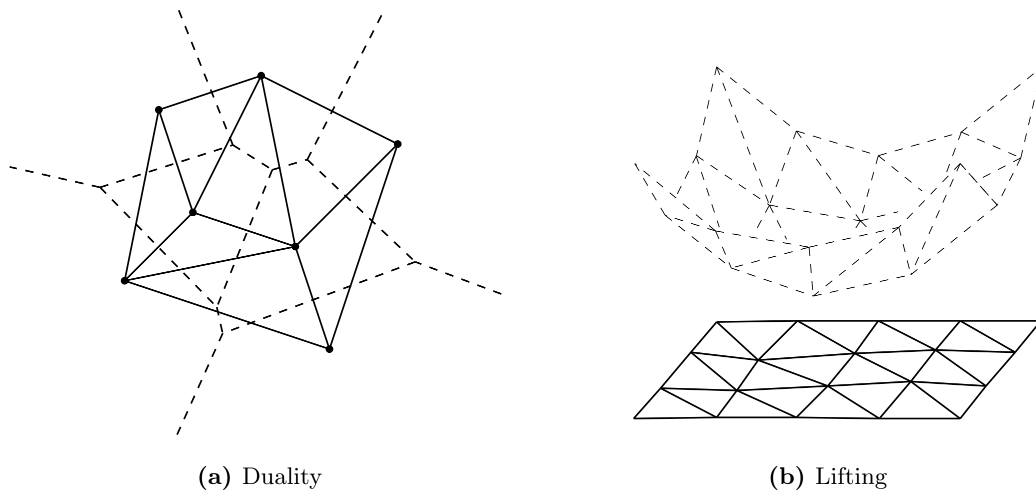
When an edge e has only one link point, it is a boundary edge. It is convenient to imagine that e is incident to a triangle that extends to infinity, and thus has a link point at infinity; therefore e is also locally Delaunay. The locally Delaunay property of an edge is easily verified using an *incircle test*. The following lemma shows the connection between this local property and the global one.

Lemma 2.1 (Delaunay lemma)**.** *If every edge of a triangulation $\mathcal{T}(S)$ is locally Delaunay,*

then $\mathcal{T}(S) \equiv \mathcal{D}(S)$ [Law77].

All the concepts above can be generalized to higher dimensions, such as $\mathbb{R}^3$ in which triangles become tetrahedra, and circumcircles become circumspheres. The link of p in this case is a polyhedron formed by the vertices, edges and facets (i.e. triangles) incident to the tetrahedra of the star of p but not containing p. Similarly, the link of an edge e is a closed chain of vertices and edges from the tetrahedra incident to e, but not intersecting e; see Figure 2.2d.

2.1.4 Geometrical relations



(a) Duality **(b)** Lifting

Figure 2.3: The relations between Voronoi diagram, Delaunay triangulation, and convex hull.

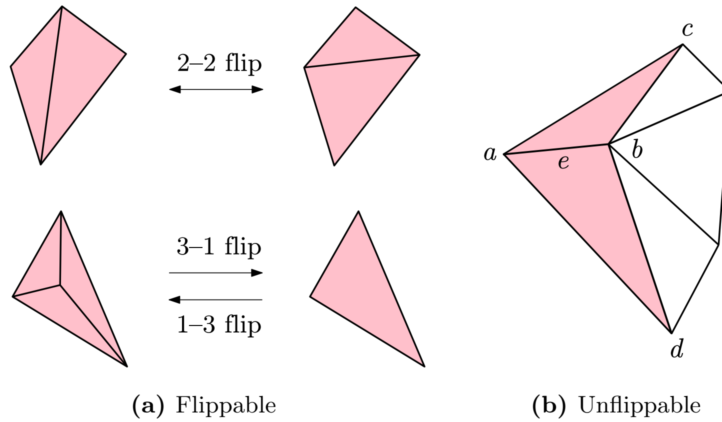
The three fundamental geometric structures discussed in the previous sections are related to one another under two relations: duality and lifting. From one structure we can theoretically derive the others.

The first relation, discovered by Boris Delaunay himself, is between the Voronoi diagram and the Delaunay triangulation. Simply speaking, the Delaunay triangulation is the *dual graph* of the Voronoi diagram. The duality is taken by replacing each Voronoi edge by a straight edge connecting the two corresponding input points, and each Voronoi vertex by a triangle of the three corresponding input points; see Figure 2.3a. The reverse can also be done. In practice, the Voronoi diagram is usually not constructed directly but through the construction and dualization of the Delaunay triangulation.

The second relation is between the Delaunay triangulation and the convex hull. Given a point set S in $\mathbb{R}^2$, we lift each point $p = (x, y)$ to the point $p' = (x, y, x^2 + y^2)$ in $\mathbb{R}^3$, resulting in a new point set S'. The projection of the lower hull of S' back to $\mathbb{R}^2$ is exactly the Delaunay triangulation of S; see Figure 2.3b.

The duality and the lifting relations can also be generalized to $\mathbb{R}^3$ and higher dimensions. Because of the lifting relation, a 2D incircle test can be implemented as a 3D orientation test, while the 3D insphere test is equivalent to the 4D orientation test.

2.1.5 Flipping



(a) Flippable **(b)** Unflippable

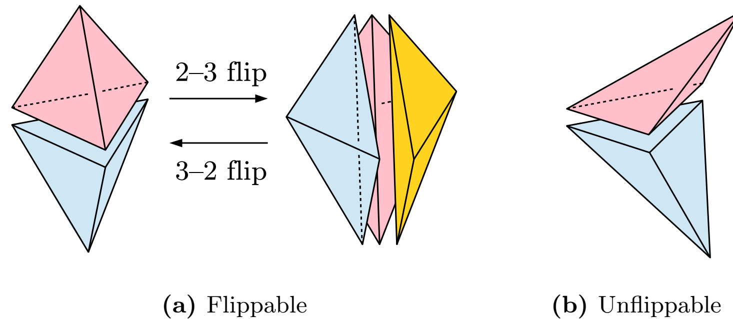
Figure 2.4: The flipping operation in $\mathbb{R}^2$.

In this section we discuss one very important operation, called *flipping*, to locally modify a triangulation or a tetrahedralization. Let us start with a general definition.

Definition 2.6. Given a set S of $d+1$ points in $\mathbb{R}^d$, there exists only two triangulations of S. The *flipping* operation replaces one with the other.

A flip is the smallest topological modification possible to a triangulation. In $\mathbb{R}^2$, a flip either replaces two triangles with another two, or three triangles with one, or the other way around. We call them 2–2 flip and 3–1 (or 1–3) flip respectively; see Figure 2.4a. A 2–2 flip replaces an edge with another edge, so we usually refer to it as an edge flipping operation. In $\mathbb{R}^3$, we have 3–2, 2–3, 4–1 and 1–4 flips; see Figure 2.5a for an illustration of the 2–3 and 3–2 flips.

From the definition, it is clear that a flip can only be performed on $d+1$ points in a triangulation $\mathcal{T}$ if one of the two triangulations of these points completely exists in $\mathcal{T}$. In $\mathbb{R}^2$, consider an edge $e = (a, b)$ and its link $\{c, d\}$. The *induced subcomplex* σ_e of e is the set of all triangles (as well as their edges and vertices) in $\mathcal{T}$ having all vertices in $S_e = \{a, b, c, d\}$. We say that e is *flippable* if and only if σ_e is a triangulation of S_e; in other words, the underlying space of σ_e is the convex hull of S_e. Otherwise, e is *unflippable*; see Figure 2.4b. In $\mathbb{R}^3$, we consider a triangle instead of an edge. A 2–3 unflippable configuration is illustrated in Figure 2.5b.



(a) Flippable (b) Unflippable

Figure 2.5: The flipping operation in $\mathbb{R}^3$.

2.2 Graphics processing unit

Since the introduction of programmable shaders, the GPU has been used for general purpose computation besides its originally designed purpose of rendering graphics. Researchers recast their problems into the graphics pipeline in order to make use of the floating-point performance of the GPU [OLG+07]. Typically, these problems are embarrassingly parallel, and thus few modifications are needed.

In the past six years, starting from the introduction of the CUDA programming framework by NVIDIA [NBGS08], the GPU has undergone several major improvements, from the introduction of atomic operations, double precision floating-point, to the support of caching, call stack and recently dynamic parallelism. Figure 2.6 presents the architecture diagram of the GF100 architecture used in the NVIDIA GTX580 GPU. Other GPU vendors such as AMD and Intel also provide similar features, while supporting the open standard OpenCL [LKS+10]. In this thesis, we use the terms in the CUDA programming framework and NVIDIA's GPU architecture, but most of the discussions are still applicable to OpenCL and AMD GPUs.

2.2.1 Terminology

A CUDA device consists of one or more *stream multiprocessors*, each of which is composed of many *stream processors*; see Figure 2.6. The CUDA programming language is an extension of C++, with some extra syntax elements for GPU code. The GPU code is executed on the GPU as *kernels*. Each kernel is executed across an array of *threads*, which will be scheduled on the stream processors automatically by the GPU.

The threads executing a kernel are organized into *thread blocks* of the same size. The users specify the number of threads per block and the total number of blocks to be executed when launching the kernel. Threads in the same thread block are guaranteed to be executed on the same stream processor, and there is a cheap barrier to synchronize them. On the

Figure 2.6: Architecture of the NVIDIA GTX580 GPU [NVIDIA].

other hand, the only way to synchronize all the threads executing a kernel is to stop the kernel and return to the CPU. This is typically costly and also all the data in the temporary variables and registers are lost. Threads inside the same block also have another mechanism to communicate: using the *shared memory*. Although having a rather small size, the shared memory is faster, by up to two orders of magnitude compared to accessing normal GPU memory (usually referred to as *global memory*). For more details, see the CUDA Programming guide in the CUDA Toolkit.

2.2.2 Challenges

We identify three main challenges when programming for the GPU.

- **Parallelism**. This first and most important challenge, immediately visible from Figure 2.6, is due to the huge number of stream processors on the GPU. An NVIDIA GTX580 has 512 stream processors, and a professional card might have up to several thousands of them. Furthermore, accessing registers and performing mathematical operations all take several cycles, while accessing the GPU memory might take up to hundreds of cycles. This latency can be hidden when there is more than one thread ready to be scheduled for each stream processor. The GPU can switch between different threads with zero latency. As such, for efficiency, it is desirable to have tens of thousands of executing threads at any given time. Developing algorithms with such a level of parallelism is very challenging. Besides, locking is difficult due to the way

threads in the GPU are scheduled, so cooperating among huge number threads becomes even more difficult.

- **Divergence**. The second challenge comes from the architecture of a stream multiprocessor. The stream processors inside a multiprocessor are not independent, but rather grouped into a SIMD group, i.e. they must execute the same instruction in the same cycle. As such, the threads in a thread block are grouped into warps each of 32 threads. Threads in the same warp execute in lockstep. When they need to take different paths, the warp is split into two, each with some threads disabled. These warps are merged again as soon as the divergent path is completed. In the worst case when all 32 threads are on different paths, their execution is effectively serialized. This has a significant effect on the performance, so designing algorithms with less divergent in each kernel is important.

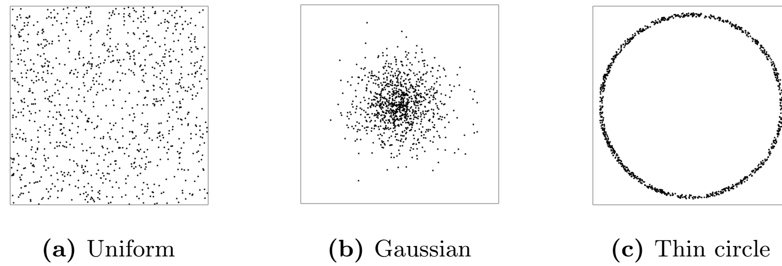
- **Memory**. The third challenge of GPU programming comes from the fact that the memory system is pretty much sequential. In each cycle, if the memory access of some threads in a warp can be combined in a single request (i.e. the access are coherent), then the memory system can serve these threads all together. However, if these accesses cannot be combined, multiple requests are required, and these threads will be served sequentially and thus the performance is reduced. Combining that with the very high latency of the memory and the very tiny amount of cache available (typically less than 100KB per multiprocessor), accessing the memory becomes a serious bottleneck especially for applications with lots of memory access.

The three challenges mentioned above lead to some important design principles when developing algorithms for the GPU. We apply them constantly on all the algorithms present in this thesis.

- First, data-parallel computation, where the same computation is performed by many threads on multiple pieces of data, is preferred. Therefore, we need to make our algorithm as simple and uniform as possible. The work load of each thread should also be similar, since load balancing techniques such as work stealing or work donation might be costly. This is to deal with the parallelism and the divergence challenge.

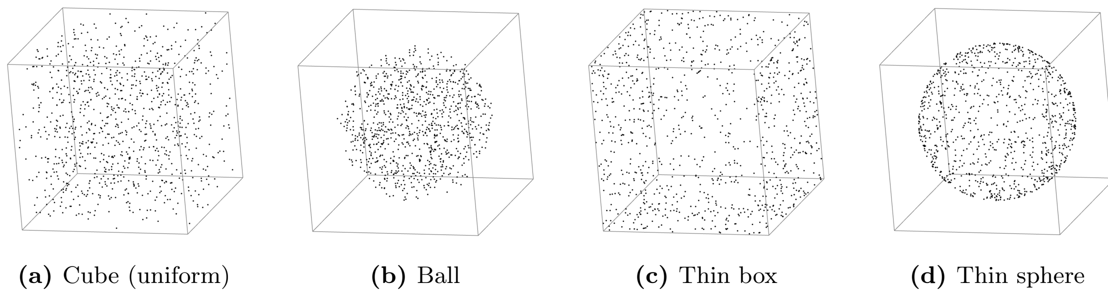
- Second, with so many threads, we usually employ some simple checks to break the set of jobs into several groups, within which the jobs can be done concurrently with no conflicts. That effectively makes the algorithms lock-free, while still allows the algorithm to have very high parallelism.

- Third, our algorithms should strive for locality of threads which access the same data, to improve the utilization of the small cache. Memory accessed by a thread should also be local, not only for better caching efficiency but also for reducing the chance of conflicting with other threads. This is mainly to address the memory challenge, as well as improving the parallelism.



(a) Uniform (b) Gaussian (c) Thin circle

Figure 2.7: Synthetic point distributions in $\mathbb{R}^2$.



(a) Cube (uniform) (b) Ball (c) Thin box (d) Thin sphere

Figure 2.8: Synthetic point distributions in $\mathbb{R}^3$.

2.3 Experiment setting

All the experiments in this thesis are conducted on the same PC unless otherwise stated. The PC has an Intel i7 2600K CPU running at 3.4GHz, with 16GB of DDR3 RAM. The GPU we use is an NVIDIA GTX 580 with 3GB of video memory. Visual Studio 2012 and CUDA 5.0 Toolkit are used to compile all the programs in 64-bit mode, with all optimizations enabled.

The input data for all our problems are a set of point, except the digital Voronoi diagram one where the points are expected to have been labeled directly into the grid. There are three types of data used throughout the experiments.

1. **Synthetic data.** Points are generated randomly in some distributions. For the digital Voronoi diagram problem, points are generated within a grid with certain density. For the 2D Delaunay triangulation problem, we use a uniform and a Gaussian point distribution. Besides, we also generate points uniformly inside a thin circle, i.e. the area between two concentric circles of radius r_1 and r_2 with $r_2 - r_1$ being a very small number; see Figure 2.7. In 3D we additionally have a ball distribution, i.e. points are generated uniformly inside a ball, and a thin sphere distribution similar to the thin circle one. For the 3D convex hull problem, besides these we add a thin box distribution, i.e. we replace spheres with cubes; see Figure 2.8. These distributions allow us to test

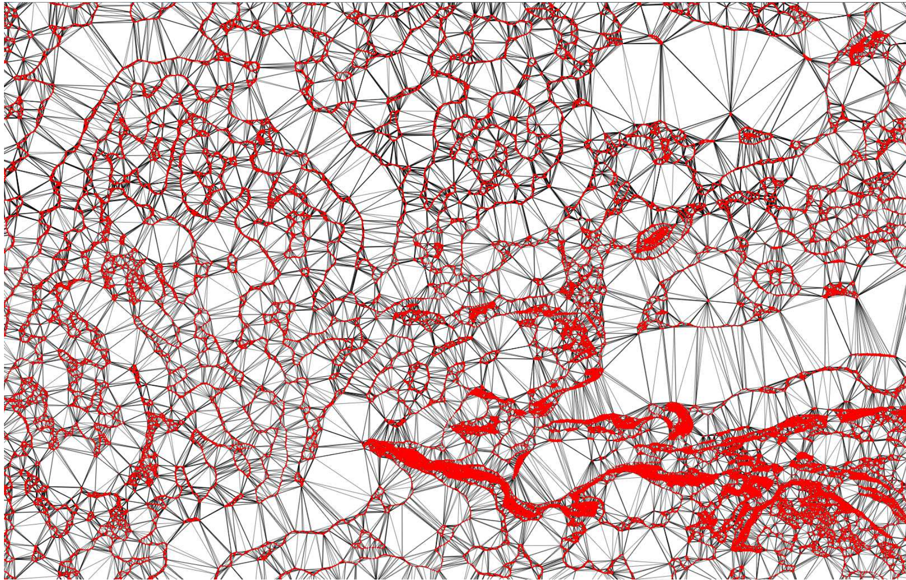
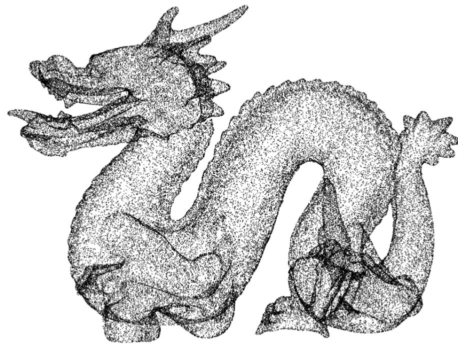


Figure 2.9: A cropped snapshot of the Delaunay triangulation of one contour map.

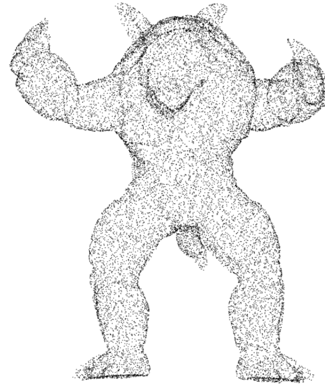
the performance of our algorithms in some controlled, yet representative, situations.

2. **Real-world data.** Here we use point sets from some real-world examples for the testing. In 2D we use the points extracted from the contour maps freely available at https://www.ga.gov.au/. In these datasets, points are distributed non-uniformly along the contour curves, which are mostly nested closed curves, similar to level set curves. See Figure 2.9 for a cropped snapshot of the Delaunay triangulation of one such dataset. In 3D, we use points from several models obtained from objects in the real world, namely Armadillo, Dragon, Happy Buddha, Asian Dragon, Turbine Blade, Angel and Brain; see Figure 2.10. The first four models are scanned surface data obtained from the Stanford 3D Scanning Repository [Sta]. The Turbine Blade and the Angel model are also scanned data from the Georgia Tech Large Geometric Models Archive [Geo]. The Brain model is obtained from the Princeton Suggestive Contour Library [Pri]. Testing on these models demonstrate the expected performance of our algorithms when running on real applications such as FEM or computer games. The points are usually not very nicely distributed, and the amount of degeneracy ranges from moderate to high.

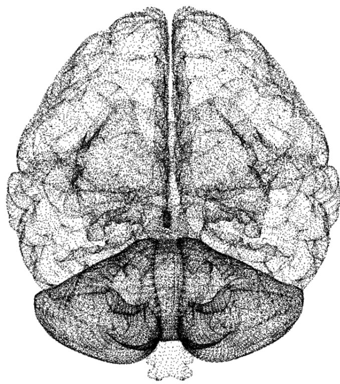
3. **Pathological data.** We also push the limit of the algorithms by testing on some pathological point distributions. In 2D we try points lying exactly on a circle. In 3D, we use points on a sphere, on an ellipsoid, on grid points of a grid, and on two non-intersecting line segments. These cases usually do not happen in practice. It is to show the robustness of our algorithms, as well as to test its efficiency at handling exact computation.



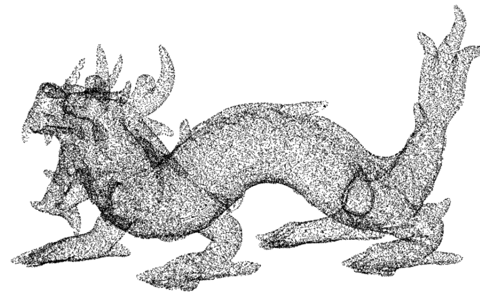
Dragon – 437K points



Armadillo – 172K points



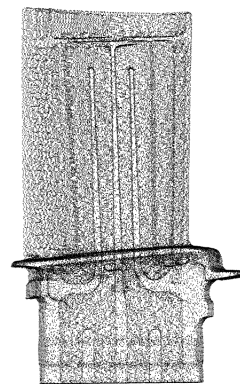
Brain – 294K points



Asian Dragon - 3,609K points

Angel – 237K points

Happy Buddha – 543K points



Blade – 882K points

Figure 2.10: Input points from real-world models $\mathbb{R}^3$.

In our experiment, double precision floating points are used for both input generation and output computation. The total time measured for our implementation always include the time to copy the input from the CPU memory to the GPU memory, as well as the time to copy the result back. In some cases these copying time can be quite a major part of the total time, and we will mention that in the respective experiment sections.

Chapter 3

Related Work

This chapter covers some important previous works on the digital Voronoi diagram, convex hull and Delaunay triangulation problems, including those for multi-core and GPU systems. We cover some traditional approaches for sequential algorithms, and discuss in detail some algorithms that are directly related to our work. At the end of the chapter, we also survey some interesting works that are related to designing algorithms for the GPU.

3.1 Digital Voronoi diagram

Digital Voronoi diagram is actually not a widely known geometric structure in computational geometry. In contrast, it is a well-studied problem in computer vision since it is equivalent to the Euclidean distance transform (EDT) problem, which has many applications such as pattern matching, morphological operations, video stylization, etc. Early works approximate the Euclidean distance using other metrics such as Chamfer distance or chessboard distance for faster computation. However, with recent development in the computation of the EDT, other approximations are no longer necessary. The two recent survey papers [FCTB08, JBS06] compare and contrast many state-of-the-art sequential approaches to solving the problem in 2D and 3D, targeting mainly the *exact* EDT computation. In the following sections, we highlight some of the works in computing the EDT, both exact and approximate. We look into sequential algorithms, parallel (mostly PRAM) algorithms, as well as earlier GPU works. In the discussion, we sometimes use the term *site* to refer to an input point that is associated to a grid point.

3.1.1 Exact and approximation

Both the exact and approximate EDT can be sequentially computed in time linear to the number of grid points $M = m^d$ in d-dimension. Most approximate EDT algorithms are based on Danielsson's vector propagation approach [Dan80]. This approach stores the coordinates of a candidate site for each grid point in the grid. These coordinates are then propagated using a structuring element called *vector template*. Multiple templates are swept in some

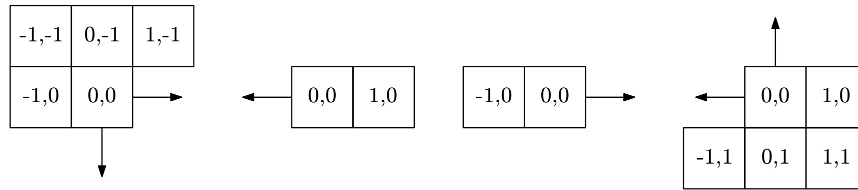


Figure 3.1: Danielsson's vector template.

certain fashions across the image. For example, in 2D the information can be propagated from top to bottom (left to right and then right to left) and then from bottom to top using the template in Figure 3.1. Such an algorithm runs in linear time, and performs very well in practice due to its cache-friendly memory access pattern. It also produces highly accurate EDT with just a small number of grid points possibly having inaccurate nearest site (and thus distance value). As such, this approach is widely used in the computer vision community.

The exact EDT for a binary grid of arbitrary dimensions, on the other hand, can be computed using a dimensionality reduction approach by Maurer *et al.* [MQR03]. For each dimension, the EDT can be computed by using the EDT in the next lower dimension to construct the intersection of the Voronoi cells of the input points with each "row" of the grid. The computation is done using the 3 properties of the digital Voronoi diagram that we discuss in Section 4.2.

It is notable that Maurer *et al.*'s algorithm is inherently parallel. In each dimension, each row of the grid is processed independent from other rows, thus they can be handled in parallel. However, the parallelism is limited and the time complexity is not optimal, especially on low dimensions such as 2D or 3D.

3.1.2 PRAM algorithms

Besides sequential algorithms, a large body of works has been proposed to solve the EDT problem in parallel, targeting the theoretical parallel machine model PRAM, including both the EREW and the CRCW model. Lee *et al.* [LHS03] use dimensionality reduction together with the theorem proven by Kolountzakis and Kutulakos [KK92] to compute the exact EDT in $O(\log^2 M)$ time using $O(M)$ processors. Better still, by redefining the problem of finding the intersection of the Voronoi diagram with each row of the grid as the problem of finding proximate sites (which can be optimally computed in $O(\log m)$ time using $O(\frac{m}{\log m})$ processors [HNO98]), one can compute the exact EDT in $O(\log m)$ time [WHLL01]. Such a result is theoretically optimal. However, all the algorithms mentioned above are developed for the theoretical EREW PRAM model, with no known practical implementation. Better time complexity algorithms for the more powerful CRCW PRAM model are also known;

see [WHLL01]. Our algorithm in Section 4.2 is inspired by Hayashi *et al.* [HNO98], but is much simpler and more practical to implement on the modern graphics hardware.

3.1.3 GPU algorithms

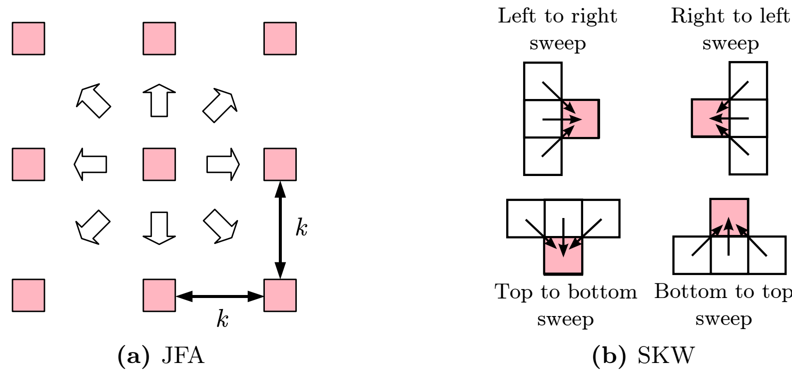


Figure 3.2: Vector templates of some GPU algorithms.

The early attempts to compute the approximate EDT using the graphics hardware include the work of Hoff *et al.* [HKL+99] and Fischer and Gotsman [FG06]. They render a right-angle cone for each site in the image to approximate the distance function, and use the depth-testing feature on the GPU to obtain the distance map. Their method suffers from overdrawing and tessellation error. Sud *et al.* [SGGM06] use a bilinear interpolation equation to compute the distance vector at any point on a polygon using the distance vectors of the polygon vertices. Their method can compute highly accurate distance maps for complex models, but its complexity is dependent on the number of input points. Similar approaches using the graphics pipeline also appear in earlier works [SPG03, SOM04, SGM05].

More recent works use the vector propagation approach to compute the approximate distance transform on the GPU. Rong and Tan [RT06] propose the Jump Flooding Algorithm (JFA) to compute the EDT in $\mathrm{O}(\log m)$ time using $\mathrm{O}(M)$ processors. In 2D, JFA uses the vector template shown in Figure 3.2a. At each pass, each grid point (x, y) propagates its information to eight neighbors at position $(x + i, y + j)$ where $i, j \in \{-k, 0, k\}$. JFA varies the step length k in different passes to propagate information throughout the grid. In the first pass, $k = \frac{m}{2}$, and in each subsequent pass k is halved (assuming that m is a power of 2). JFA uses $\mathrm{O}(\log m)$ passes, thus the EDT can be computed in $\mathrm{O}(\log m)$ time, though with a small rate of error. Although JFA can easily exploit the computing power and memory bandwidth of the GPU, it has a suboptimal total work complexity of $\mathrm{O}(M \log m)$. Besides, the work only provides a little insight into the (expected) low error rate, and not any bound on the absolute distance error. Cuntz and Kolb [CK07] propose a speedup version of JFA by using a hierarchical approach to reduce the total work to $\mathrm{O}(M)$ at the cost of a much high error rate. The higher error rate is because their algorithm relies on down-sampling the input

grid to reduce the total work, while Voronoi diagram is usually very sensitive to any slight change in the position of input points that are close to one another. This thus limits its use in practice.

Schneider *et al.* [SKW09] also modify Danielsson's vector templates slightly; see Figure 3.2b, to allow concurrent propagation for grid points in the same row. Their sweeping algorithm, termed SKW, can be implemented on the GPU with linear total work complexity and the resulting distance map is close to exact. However, SKW has a high time complexity of $O(m)$, and thus usually does not run faster than JFA. This is because it can only perform parallel propagation of grid points in one row at a time (in 2D problem). With a grid size limited by the available memory on the GPU and the need to have tens of thousands of threads or more in order to optimally utilize the processing power available, SKW often under-utilizes the GPU.

3.2 Convex hull

Convex hull is arguably the most fundamental computational geometry problem, with a long history of researches and applications. The concept is so useful and easy to understand that its 2D version is commonly covered in the first undergraduate course in algorithm. In this section, we look at some popular sequential algorithms for convex hull as well as a few recent attempts at constructing this geometric structure on the GPU. We also briefly introduce the star splaying algorithm, an unconventional method to construct or fix the convex hull [She05]. We use and adapt the star splaying algorithm in several places in this thesis. The discussion below focuses on the problem in $\mathbb{R}^3$, but is also mostly applicable to other dimensions.

3.2.1 Sequential and parallel algorithms

Two popular approaches commonly used to construct convex hull are the incremental insertion approach and the divide-and-conquer approach. The incremental insertion approach constructs the convex hull by locating and inserting points incrementally [CS88]. QuickHull [BDH96] is a variant of such approach. In $\mathbb{R}^3$, the algorithm begins with a single tetrahedron or a volume in general, usually formed by four extreme vertices. Input points outside the volume are recursively inserted to grow its size, while points found to be within the volume are discarded from subsequent computation. At each step, the farthest input point from the facets of the volume is chosen to be added. Such a point is an extreme vertex, and this also potentially maximizes the number of input points that can be discarded.

The second approach, divide-and-conquer, is used in the algorithm of Preparata and Hong [PH77]. The input point set is divided into subsets of very small size, such that the convex hull of each subset is easily obtained. Subsequently, a merge procedure for two convex

hulls is recursively applied. The input is divided such that any two sub-results are non-intersecting to simplify the merging procedure. Nevertheless, it is still quite challenging in higher dimensions.

Both the incremental insertion and the divide-and-conquer approach have an $O(n \log n)$ time complexity. In $\mathbb{R}^2$ and $\mathbb{R}^3$, the optimal output-sensitive convex hull algorithm has a time complexity of $\Theta(n \log h)$ where h is the number of extreme vertices [KS86, Cha96]. Empirically, QuickHull is found to have the same output-sensitive time complexity. Because of that and its low overhead in practice, QuickHull has been a popular algorithm adopted by many applications over the years.

Parallel algorithms for convex hull have also been extensively studied in the last few decades. For example, Miller and Stout [MS88] and Amato and Preparata [AP93] propose $O(\log n)$ parallel algorithms using $O(n)$ processors. These algorithms are only of theoretical interest as they have no known efficient implementation. One of the reasons is that these algorithms are complex, making them hard to scale on a fine-grained data-parallel massively-multithreaded architecture. For the current multi-core systems with a small number of independent processors, algorithms designed by Dehne *et al.* [DDD+95] might be more applicable. These algorithms, however, also do not have known implementations to demonstrate their use.

3.2.2 GPU algorithms

Recently, convex hull algorithms designed specifically for the GPU are also studied. Srungarapu *et al.* [SRKN11] and Jurkiewicz and Danilewski [JD10] propose two algorithms similar to QuickHull on the GPU that work for the simple 2D case. Tzeng and Owen [TO12] further extend that approach to $\mathbb{R}^3$ and higher dimensions. However, their algorithm can easily output non-extreme vertices; see Figure 3.3 for a counter-example. Stein *et al.* [SGES12] propose to compute the convex hull in $\mathbb{R}^3$ by iteratively inserting points and flipping. Their claim is that their algorithm flips all reflex edges, thus the result is a convex hull. This, however, is not true as the algorithm prohibits flipping of concave edges if doing so causes self-intersection (as indicated in their paper), thus the final result might still contain reflex edges. The only known approach that can produce the correct convex hull in $\mathbb{R}^3$ with the help of the GPU is that of Tang *et al.* [TZTM12]. In this work, the hull is grown on the GPU by iteratively inserting points into an initial tetrahedron, without any other modification. During the process, any point found to be inside the hull is removed. Then, those points surviving the process are passed back to the CPU memory and a CPU-based program (such as CGAL) is used to compute the convex hull. As pointed out by the authors, if most of the input points are extreme vertices, then their algorithm is even slower than the CPU-based program due to the time wasted on the filtering step on the GPU.

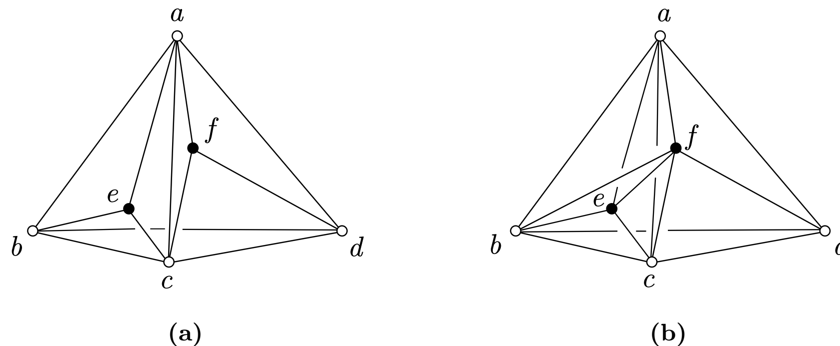


Figure 3.3: An example in which the algorithm in [TO12] outputs a wrong result. In (a), after creating the initial tetrahedron $abcd$, e is flagged with $\triangle abc$ while f is with $\triangle acd$, and both of them are output as extreme vertices. In the correct result in (b), e is not an extreme vertex since it lies inside tetrahedron $fabc$.

3.2.3 Star splaying in $\mathbb{R}^3$

Star splaying [She05] is a very efficient algorithm to repair convex hulls in any dimensions. In this section, we briefly outline the algorithm in $\mathbb{R}^3$.

In $\mathbb{R}^3$, the boundary of a polyhedron is topologically similar to a triangulation in $\mathbb{R}^2$, and the concepts of stars and links also apply. By extending the star of a vertex s to infinity, we get a *cone*; see Figure 3.4a. If the polyhedron is convex, then the cone of each of its vertices is also convex, and at the same time encloses all other vertices of the polyhedron.

The stars of the vertices of a polyhedron are *consistent* with each other. That is, if the star of t contains $\triangle stu$, then the stars of s and u also contain this triangle. Moreover, a set of consistent stars uniquely defines a surface triangulation. However, an arbitrary collection of stars not coming from a polyhedron may not be consistent with each other.

The star splaying algorithm is based on the idea that if the cones of all the vertices are made convex and their corresponding stars are made consistent, then these stars uniquely define the convex hull of the input point set. Starting from a set of stars with their cones being convex, the algorithm repeatedly checks for each triangle stu in the star of t whether this triangle exists in the star of s and u or not. If $\triangle stu$ does not exist in the star of s, then some points (t, u, or both) will be inserted into the star of s in an attempt to *splay* it wider to include $\triangle stu$.

The insertion of a point p (either t or u) into the star of s is done using the traditional beneath-beyond method [Kal81] to guarantee that the cone of s is still convex after the insertion; see Figure 3.4b. Such insertion fails only when the triangle is interior to the cone of s, in which case some vertices on the link of s will be inserted into the star of t to splay

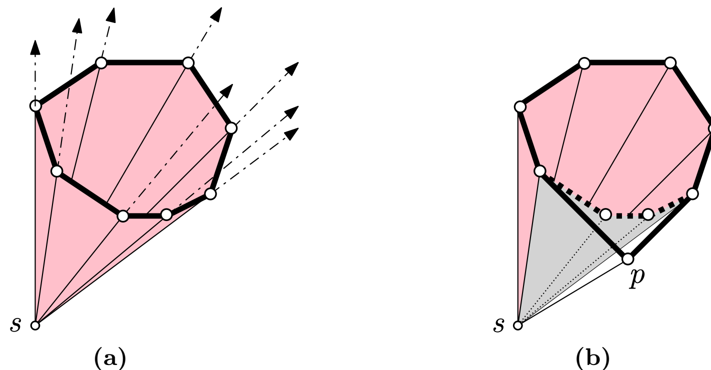


Figure 3.4: Star splaying algorithm. (a) The cone of vertex s with its link in bold. (b) Beneath-beyond insertion of p into the star of s.

that star further to remove $\triangle stu$. If the star of a vertex splays wider than a half space, that vertex is guaranteed not to be an extreme vertex; we call its star a *dead star*.

A nice feature of the star splaying algorithm is that creating stars having convex cones and enforcing their consistencies can both be done independently for each star. This is well suited to the parallel computation model of the GPU since stars can be repeatedly checked and modified in multiple steps without requiring any explicit locking. However, star splaying is not suitable for constructing convex hull from a point set, since its time complexity would be much higher than optimal.

3.3 Delaunay triangulation

The Delaunay triangulation is one of the most useful structures in computational geometry, and thus it received lots of research attention. In this section, we detail some popular approaches to construct the Delaunay triangulation sequentially, followed by some recent work on adopting these approaches to parallel systems, with the focus on those for multi-core ones.

3.3.1 Sequential algorithms

There are four major approaches to construct the Delaunay triangulation of a given point set sequentially: incremental construction, sweep line, divide-and-conquer, and incremental insertion.

Incremental construction

The incremental construction approach is also often referred to as the gift wrapping approach due to the similarity with the corresponding convex hull algorithm. The algorithm was proposed for 2D by McLain [McL76] and generalized to 3D by Cignoni *et al.* [CMS92] in the InCoDe algorithm. In $\mathbb{R}^2$, the Delaunay triangles are incrementally discovered, one at a time. Starting from an arbitrary input point, we find the point nearest to it, and this forms a Delaunay edge. Then, we find another point such that the circumcircle of the triangle formed by these three points is the smallest. This is guaranteed to be a Delaunay triangle. From this, Delaunay triangles are incrementally constructed from the edges that are not yet completed, i.e. those with only one incident triangle. The algorithm is output-sensitive, taking $O(nf)$ time where f is the number of Delaunay triangles. The same method works in higher dimensions as well. Furthermore, some data-structure can be used to reduce the time to search for points to complete a facet; see for example [Dwy91]. Still, this approach is not very efficient due to the mentioned high time complexity.

Sweep line

The sweep line approach is based on the duality between the Voronoi diagram and the Delaunay triangulation. Fortune [For87] uses a sweep line algorithm to construct the Voronoi diagram in $\mathbb{R}^2$, from which the Delaunay triangulation can be obtained. First, the algorithm sorts the input points by their x-coordinates, and then a vertical line, called the *sweep-line*, is swept from left to right. Points behind the sweep-line have already been added into the Voronoi diagram, while points ahead of the sweep-line are waiting for processing. As the sweep-line progresses, the Voronoi edges are generated incrementally. Two events are processed when the sweep-line goes through the space: when an input point is reached, and when the Voronoi vertex is crossed. The running time of this algorithm is $O(n \log n)$.

It is, however, not clear how to generalize this approach to $\mathbb{R}^3$. One of the reasons is that it is much more costly to determine when the sweep-plane passes a Voronoi vertex. As such, this algorithm is not used to construct Delaunay triangulation in dimensions higher than two.

Divide-and-conquer

In $\mathbb{R}^2$, the input point set is repeatedly divided into smaller sets, until a set is small enough that the Delaunay triangulation can trivially be computed. Then the algorithm recursively merges the results of two small adjacent sets into that of a bigger one, until results of all sets are grouped into one triangulation, the Delaunay triangulation. Using this approach, the result can also be computed in optimal $O(n \log n)$ time [SH75, Dwy87].

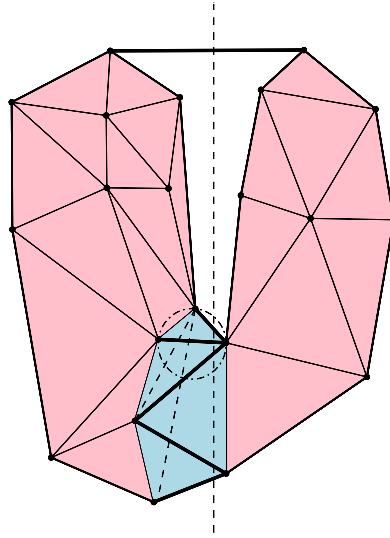


Figure 3.5: Constructing 2D Delaunay triangulation using divide and conquer.

This approach, however, is also difficult to generalize to higher dimensions. One reason is that, as shown in Figure 3.5, the merge phase relies on an explicit ordering of the edges incident to a vertex. Such an ordering, which generalizes to facets incident to a vertex, is not available in $\mathbb{R}^3$ or higher dimensions.

Instead, to use divide-and-conquer in $\mathbb{R}^3$, a *merge-first* approach is needed, as proposed in the DeWall algorithm by Cignoni *et al.* [CMS92]. The idea is to build the Delaunay tetrahedra intersected by the dividing plane first, using the incremental construction approach, before recursively constructing the rest of the Delaunay triangulation on the two sides of the plane. By doing so, the merge phase is avoided.

Incremental insertion

This is arguably the most powerful approach for Delaunay triangulation in particular, and for computational geometry in general. From a general point of view, the approach is to insert input points one by one into the existing structure, and then performing some modification if necessary.

In $\mathbb{R}^2$, three variants are possible. In the first variant, all input points are located and simply inserted by splitting the triangle containing it. Then, a sequence of Delaunay flips are applied on non-Delaunay edges until the Delaunay triangulation is reached [Law77]. This variant has the worst case time complexity of $O(n^2)$. In the second variant, flipping is performed right after each point insertion [GS85]. When points are inserted in a random order, the expected time complexity of $O(n \log n)$ can be achieved. In the third variant, instead of using flipping, all triangles having their circumcircles enclosing the newly inserted point are

removed, with the hole created guaranteed to be star-shaped and can simply be glued with the new point. This is called the Bowyer-Watson algorithm [Bow81, Wat81], and is in a way similar to the beneath-beyond algorithm by Kallay [Kal81].

In $\mathbb{R}^3$, the first variant above is no longer possible, as shown by Joe [Joe89]. Flipping can get stuck, a situation in which no more Delaunay flip is flippable, and yet we still have not reached the Delaunay triangulation. In fact, it is still an open problem whether flipping can transform any 3D triangulation into the Delaunay triangulation. Fortunately, the second and the third variants work without any problem in $\mathbb{R}^3$ or higher dimensions [Joe91].

3.3.2 Parallel and streaming algorithms for the CPU

There are several attempts at parallelizing the construction of Delaunay triangulation on multi-core systems, especially for the 3D case. All of them are based on the incremental insertion approach; see the survey in [KKv05] or the more recent works in [BMPS10, FC12]. Several points are inserted in parallel, followed by either flipping or Bowyer-Watson's algorithm. For correctness, two threads cannot update the same tetrahedron at the same time. Moreover, two parallel insertions also cannot conflict with each other, i.e. the regions affected by them overlap at some tetrahedra. As such, some locking strategies must be applied. When a conflict happens, one of the two insertions must rollback all its work and try again later. When the number of cores is small, which are typically 4 to 8 for current multi-core systems, such algorithm is quite efficient. The implementation in [BMPS10] shows a speedup of 7 times over the sequential CGAL Delaunay triangulator on an 8-core CPU. However, with the huge number of threads needed on the GPU, the conflicts may happen too often for these approaches to be usable, not to mention the complication when locking is involved on the GPU.

As a very practical problem, there are times in which the input point set is too large that the Delaunay triangulation computation cannot be done in the memory of a single machine. For these cases, two solutions have been investigated: streaming and using distributed systems. The streaming approach proposed by Isenburg *et al.* [ILSS06] is based on the concept of spatial finalization. The point stream is spatially partitioned into regions, and finalization tags are added into the stream to indicate when no more points in the stream will fall in the specified regions. After that, a standard incremental insertion algorithm is used. Using these tags, the algorithm can conclude when certain part of the triangulation is finalized, i.e. future insertions cannot modify it any further. These parts can be output and then released from the memory, thus reducing the memory footprint of the program. For distributed systems, a similar domain partitioning approach is also commonly used. In the work of Lo [Lo12], the sub-domains are overlapped such that the Delaunay triangulations computed independently on each of them agree with one another. By doing so, no expensive merging phase is needed. Given a very large input point set, this approach achieves a very attractive

speedup of 10.8 times on a system with 12 processors. Nevertheless, it is not a suitable approach for the GPU since it is not feasible to obtain thousands of sub-domains for the GPU stream processors.

Chapter 4

Digital Space to Continuous Space

4.1 Overview

Modern GPUs require thousands of computation threads to maximize their utilization. In this scenario, the biggest challenge is to orchestrate the work among threads while keeping them fully engaged. This is most effectively done when threads perform regularized work on localized data. For computational geometry problems in continuous space, achieving the above two qualities is not trivial due to the locality of the processing depending heavily on the input data and the topology of the structure being constructed. In this chapter, we propose a novel approach to overcome this difficulty, guiding us to devise efficient GPU algorithms for fundamental computational geometry problems. The approach typically has the following three phases: (see Figure 4.1)

Phase 1: Sketch construction.

This phase is to compute a coarse understanding of the solution, called a sketch, in digital space. The sketch should be computed efficiently with GPU, and it should enable subsequent GPU processing to be of regularized work on preferably localized data.

Phase 2: Approximation construction.

This phase is to compute an approximation of the result in continuous space. This approximation is obtained through modifying in parallel different parts of the sketch. It is an approximation because not all input points are necessarily incorporated, and some parts of the structure might have yet to meet the geometric requirements of the problem.

Phase 3: Solution construction.

This phase is to complete the solution to the given problem. In particular, any input points that are missing in the approximation in the previous phase must now be restored. Further processing is also needed to reach the full geometric requirement of the problem.

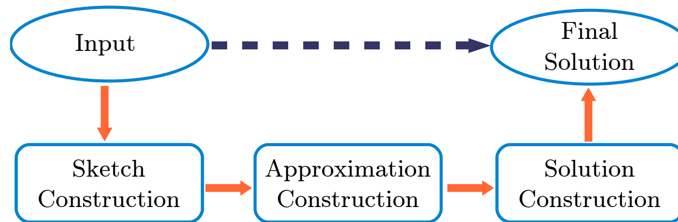


Figure 4.1: Using digital space computation to solve computational geometry problems efficiently on the GPU.

The sketch we use in the following sections is obtained through the digital Voronoi diagram. The reason is that Voronoi diagram has a close connection with both convex hull and Delaunay triangulation, and the digital Voronoi diagram provides a good starting point for solving these problems. This is an unconventional approach, since it is commonly known that the Voronoi diagram is harder to construct than the other two structures, and the dualization also poses topological and geometrical problems. Nevertheless, we show that this approach can actually be done efficiently. In the following sections, we start with developing an efficient algorithm to construct the digital Voronoi diagram on the GPU. Subsequently, we discuss how to apply our proposed approach to construct the 2D Delaunay triangulation (Section 4.3), the 3D convex hull (Section 4.4) and the 3D Delaunay triangulation (Section 4.5) on the GPU. The three proposed algorithms are termed DigiDel2D, DigiHull3D, and DigiDel3D respectively ("digi" for digital).

It is worth noting that not every approximation derived from the digital Voronoi diagram sketch can be transformed into the exact solution efficiently. As a guideline, we should pick an appropriate approximation based on the feasibility of its transformation in Phase 3. This will become more apparent during the subsequent discussions.

4.2 Digital Voronoi diagram

Given a grid $\mathcal{G}$ of size $M = m^d$ and a set S of input points, which are sometimes referred to as sites, with integer coordinates in $\mathcal{G}$, our goal is to compute for each grid point in $\mathcal{G}$ the index of the closest site having smallest Euclidean distance to it. This is almost exactly the same as the Euclidean distance transform problem in which for each grid point we want the coordinates of its closest site. In the following sections, we mainly refer to the problem as the distance transform problem since it is more efficient to store the coordinates instead of storing the index due to faster access. The digital Voronoi diagram can be derived easily from the distance transform result.

4.2.1 Exact Euclidean distance transform

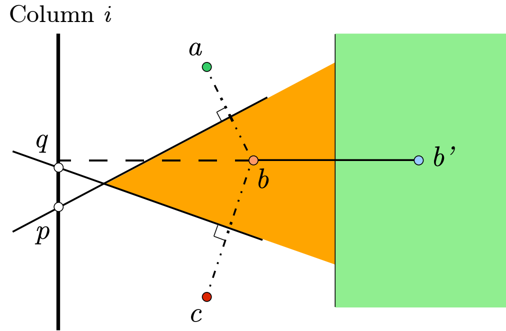


Figure 4.2: Illustration of the three lemmas to compute the exact Euclidean distance transform.

This section reviews the general approach to compute the exact Euclidean distance transform in a dimensionality reduction manner. The initial idea is proposed by Kolountzakis and Kutulakos [KK92], and further extended by Hayashi *et al.* [HNO98], Lee *et al.* [LHS03], and Maurer *et al.* [MQR03]. Our discussion is for the 2D case, i.e. we discuss the computation done in one dimension (for each row) and then in the second dimension (for each column) of the grid. The same idea can easily be extended to higher dimensions by repeating the computation for each additional dimension.

Consider a 2D grid $\mathcal{G}$ of size $M = m^2$. Following the convention in graphics, the upper left corner of the grid has coordinate $(0, 0)$ and lower right corner $(m-1, m-1)$. Write $p(i, j)$ for a pixel p at coordinate (i, j). We assume that the distances from any two input points to a grid point are different, since in case of a tie, we can consider the distance from the input point with smaller coordinate to be smaller. We want to determine the intersection of each column i in $\mathcal{G}$ and the Voronoi diagram of the input points. Let $S_{i,j}$ be the closest site, among all input points on row j, of the grid point (i, j), and let $S_i = \{S_{i,j} \mid S_{i,j} \neq \textsf{null}, j = 0, 1, \ldots, m-1\}$ be the collection of such closest sites for all grid points on column i. Note that $S_{i,j}$ is $\textsf{null}$ when there is no input point on row j. Let P_i be the set of input points whose Voronoi cells intersect column i. These points are termed the *proximate sites* of column i. Among them, each grid point on column i needs to determine one that is its closest site. To help improve the efficiency of this computation, three straightforward lemmas are used; see [MQR03] and Figure 4.2.

Lemma 4.1. *Consider column i and let $b(i_1, j)$ and $b'(i_2, j)$ be two input points in row j. If $|i_1 - i| < |i_2 - i|$, then $\mathcal{V}_D(b')$ cannot intersect column i.* □

This lemma means that for each column i, there can be at most one input point along a row that can potentially be a proximate site; or more specifically, $P_i \subseteq S_i$. As a result, $|P_i| \leq m$.

Lemma 4.2. *Consider column i and let $a(i_1, j_1)$, $b(i_2, j_2)$, $c(i_3, j_3)$ be any three sites with $j_1 < j_2 < j_3$. Let the intersection of the perpendicular bisector of a and b and column i be*

$p(i,u)$, and that of b and c be $q(i,v)$. If $u > v$ then $\mathcal{V}_D(b)$ cannot intersect column i. □

When the mentioned situation happens, we say that a and c *dominate* b on column i. In this case, $b \notin P_i$.

Lemma 4.3. *Let $q(i,v)$ and $p(i,u)$ be two grid points in column i such that $v < u$, and let $a(i_1, j_1)$ and $c(i_3, j_3)$ be the closest sites of q and p respectively. Then we have $j_1 \leq j_3$. Note that $j_1 = j_3$ when $a \equiv c$.* □

This lemma means that the Voronoi cells of the proximate sites of column i appear in exactly the same order as the sites when sorted by their y-coordinates. With the above lemmas, the exact digital Voronoi diagram computation is done by the following three phases:

Phase 1: For each grid point (i,j), compute $S_{i,j}$.

Phase 2: Compute the set P_i for each column i using S_i.

Phase 3: Compute the closest site for each grid point (i,j) using P_i.

In the following sections, we present our Parallel Banding Algorithm (PBA) to perform the above-mentioned three phases on the GPU efficiently.

4.2.2 Phase 1: Band sweeping

In this phase, for each row, we want to compute the 1D distance transform using only those input points in the same row. A trivial approach would be to use a two-pass sweeping (left to right and then right to left sweeping), similar to SKW [SKW09]. This, however, restricts the parallelism to only one thread per row, potentially under-utilizing the GPU. One could also use the 1D JFA [RT06] with better utilization of the GPU at the cost of higher total work. Another possibility would be to use a method similar to the work efficient parallel prefix sum [HSO07]. This approach is too complex as compared to our following simple, yet work and time efficient approach.

Our approach extends the naïve two-pass sweeping approach, with the introduction of bands to effectively increase the level of parallelism. First, we divide $\mathcal{G}$ into k_1 vertical bands of equal size, and use one thread to handle one row of a band, performing the left-right sweeps. Next, for one input point to propagate its information to a different band (on the same row), it has to be the closest site of the first or the last grid point of its band. As such, to combine the result of different bands into that of the whole row, we first propagate the information among the first and the last grid points of all bands using a parallel prefix approach on these $2k_1$ grid points. With this, the first and the last grid point of each band have the correct information. Other grid points inside a band can obtain the correct closest sites by updating,

Algorithm 4.1: Merging P_i^U and P_i^V

```
   Data: Two sets of proximate sites P_i^U and P_i^V.
   Result: The merged result.
1  stack ← P_i^U
2  for c ∈ P_i^V in increasing y-coordinate do
3      while Size(stack) ≥ 2 do
4          b ← Top(stack), a ← SecondTop(stack)
5          if a and c dominate b then Pop(stack)
6          else break
7      end
8      Push(stack,c)
9      if the two sites at the top of stack are from P_i^V then break
10 end
11 Push those sites not yet processed in P_i^V into stack
12 return stack
```

if needed, their current information with that of the first and the last grid point of their band. This can be done in parallel in constant time.

4.2.3 Phase 2: Hierarchical merging

This phase computes the proximate sites P_i for each column i, given S_i. The sequential implementation to determine P_i is to sweep sites in S_i from topmost to bottommost, while maintaining a stack of points that are potentially proximate sites. When a new site $c \in S_i$ is reached, we examine (using Lemma 4.2) whether the site b at the top of the stack is dominated by c and the site a at the second top position in the stack. If so, b is popped out of the stack, and then the examination process is repeated with a taking place of b. Once this is done, c is pushed onto the stack, and the sweeping continues. At the end of the process, the content of the stack is P_i. This approach, however, restricts the level of parallelism to one thread per column.

To increase parallelism, we again employ the idea of banding. First, we divide the grid $\mathcal{G}$ into k_2 horizontal bands of equal size. Let $\mathcal{B} = \{B_1, B_2, \ldots, B_{k_2}\}$ be the set of horizontal bands. For each column of a band $B \in \mathcal{B}$, we use one thread to run the above algorithm to compute the proximate sites. Let P_i^B be the set of proximate sites of column i considering only the sites of S_i within the band B. Then, the challenge is to merge the k_2 resulting sets P_i^B from

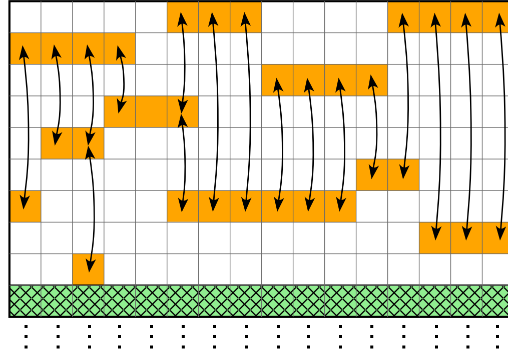


Figure 4.3: The doubly linked lists embedded on a 2D array.

different bands of column i into P_i. To do this, for each column i, we repeatedly merge each two sets of results P_i^U and P_i^V of two consecutive bands U and V (with U above V) into one, forming the result of a bigger band $U \cup V$. The merging is done in a bottom up manner, till there is only one band left. The merging of P_i^U and P_i^V is detailed in Algorithm 4.1. We treat P_i^U as a stack of sorted sites, having the largest y-coordinate one at the top of the stack, and consider each site in P_i^V in increasing y-coordinate. For each site c in P_i^V, we perform the algorithm discussed earlier, repeatedly removing the site at the top of the stack if it is dominated by c and the site at the second top position in the stack.

This algorithm uses two important observations to be efficient. First, refer to line 9. When there are two sites of P_i^V on top of the stack, no further popping is possible and we thus can break the for-loop. This is because sites in P_i^V cannot dominate each other. Second, we can implement the stack, P_i^U and P_i^V as doubly linked lists so that line 11 can be done in constant time. The doubly linked lists are embedded onto a 2D array, termed *proximate array*. Figure 4.3 shows the upper part of the proximate array where the shaded items store potential proximate sites when the processing reaches the checked row during the proximate sites computation for each band. For each site $S_{i,j}$ being considered as a proximate site of column i, we store two pointers on the proximate array at position (i, j): one pointing to the previous proximate site S_{i,j_1} and the other to the next proximate site S_{i,j_2} of column i. These pointers can simply be the indices j_1 and j_2 of the rows correspond to these proximate sites. The first and the last item of each resulting band are used to store the positions of the head and the tail of its doubly linked list.

4.2.4 Phase 3: Block coloring

This phase uses the set P_i of sites, linked as a list in increasing y-coordinate, to compute the closest site for each grid point (i, j) of column i in the order of $j = 0, 1, \ldots, m-1$. At each grid point p, we check the distance of p to the two sites a and b where a is at the front and b

is just after a in the list. If a is closer, then a is the closest site to p, and we use a to *color* p and the process is repeated for the next grid point. If not, we remove a from the list since it can no longer affect any other grid point from p onward (by Lemma 4.3), and use b in place of a as the front of the list to compute the closest site to p.

One might attempt a parallel approach by using a thread to handle a segment of grid points in column i having the same closest site in order to increase the level of parallelism. Such an approach, however, yields a completely non-coherent pattern of write operations as well as non-balanced work load for different threads, and thus does not have good performance in practice.

Instead, we propose to color a block of k_3 consecutive grid points in column i at a time, using k_3 threads. The k_3 threads cooperate through the shared memory. Each thread looks at two sites a and b in the front of the list. In the first case, when its grid point already has a site nearer to a and b, it does nothing. In the second case, when its grid point is nearer to a than to b, the thread sets the closest site to its grid point as a. Otherwise, in the third case, the thread advances the front of the list to b. The process is repeated until no thread advances the front of the list. After finishing a block of k_3 grid points, we move on to the next block of k_3 grid points.

4.2.5 Complexity analysis

In this section, we show that Phase 1 and Phase 2 are work efficient, while Phase 3 is also efficient in most situations.

Lemma 4.4. *Phase 1 takes* $\mathrm{O}(M)$ *total work and* $\mathrm{O}(\log m)$ *time.*

Proof. Choose k_1 to be $\frac{m}{\log m}$. The left-right sweep takes $\mathrm{O}(M)$ total work and $\mathrm{O}(\log m)$ time. The propagation across bands using parallel prefix can be done in $\mathrm{O}(mk_1 \log k_1) \subset \mathrm{O}(M)$ total work in $\mathrm{O}(\log k_1) \subset \mathrm{O}(\log m)$ time. The last update for each grid point within a band can trivially be done in $\mathrm{O}(M)$ total work and $\mathrm{O}(1)$ time, yielding the total work and time complexity as claimed. □

In practice, there is a limit on the number of threads that can run concurrently on the GPU. Thus, a k_1 smaller than that in the proof (as is used in our experiments) might already be optimal. The added advantage of a smaller k_1 is that the work in the propagation across bands is slightly reduced.

Lemma 4.5. *Phase 2 takes* $\mathrm{O}(M)$ *total work.*

Proof. The total work to compute the proximate set for the individual columns and bands is clearly linear. The number of merging operations performed for each column is $(k_2 - 1)$. Consider the ℓ-th merging using Algorithm 4.1, and suppose K_ℓ sites are popped in the

merging. The while-loop in line 3–7 is executed exactly K_ℓ times. Due to the breaking condition in line 9, the for-loop in line 2 can be executed no more than $K_\ell + 2$ times. Line 11 can be done in $\mathrm{O}(1)$ work since we use doubly linked lists. As such, the total work of the merging process for each column is no more than:

$$\sum_\ell K_\ell + \sum_\ell (K_\ell + 2) = \left(2 \sum_\ell K_\ell\right) + 2\,(k_2 - 1).$$

Since we can remove at most a total of $|S_i| \leq m$ sites in all the merging in column i, $\sum_\ell K_\ell \leq m$. Thus, the total work of Phase 2 is $\mathrm{O}\big(M\big)$. □

The above fact means that the total work of Phase 2 is not much affected by the choice of k_2. By taking k_2 all the way up to m, we can have the highest level of parallelism. However, the merging operation still has some overhead, while there is a limit in the level of parallelism of a GPU in practice. By allowing a flexible choice of the number of bands, we can tune the algorithm to work best on different GPUs.

Lemma 4.6. *Phase 3 takes* $\mathrm{O}\big(k_3 M\big)$ *total work in the worst case.*

Proof. The number of attempts needed for each block to confirm its color is the number of Voronoi cells that intersect that block. In the worst case, this number can be k_3, leading to the complexity of $\mathrm{O}\big(k_3 M\big)$. □

Although the presence of k_3 means a super-linear total work for our algorithm, we can maintain optimal total work if we set k_3 to be a small constant. In practice as we observe in our extensive experiments, a small value for k_3 is sufficient to achieve good performance for Phase 3.

4.2.6 3D and higher dimensions

Our algorithm can easily be extended to 3D and higher dimensions. For example, in the 3D case with $M = m^3$ grid points, having done the computation as in the 2D case for each plane where $z = \ell$ for $\ell = 0, 1, 2, \ldots, m-1$, we need to finalize the closest sites for each row of grid points (a, b, ℓ) where a and b are fixed and ℓ ranges from 0 to $(m-1)$. To do so, we apply Phase 2 and Phase 3 on each such row, following the dimensionality reduction approach by Maurer *et al.* [MQR03].

It is also possible to compute the 3D digital Voronoi diagram slice by slice. This is very useful since current GPU memory limits a 3D array to around 512^3. In order to compute the digital Voronoi diagram for a bigger grid, we can perform the computation slice by slice as follows. Let $\mathcal{S}_\ell : z = \ell$ be a slice where ℓ is an integer between 0 and $m-1$. We first compute for each of the m^2 grid points (i, j, ℓ) its closest site among all input points with

i and j as their x- and y-coordinate. This can be done by projecting all the input points onto $\mathcal{S}_\ell$. Then, we can use Phase 2 and Phase 3 of our algorithm once to compute the result along the x-axis and then again along the y-axis to obtain the digital Voronoi diagram on $\mathcal{S}_\ell$. This approach is also very useful for 3D applications that need the result for just one or several slices at any moment.

In general, consider the d dimensional problem where the input size is $M = m^d$. For such a high dimensional problem, our algorithm performs one pass of Phase 1 and $(d-1)$ passes of Phase 2 and Phase 3, each of which takes linear time, thus the total work is only $\mathrm{O}(d^2 M)$. The extra d factor in the total work is the cost of evaluating the distance function in d dimensions.

In contrast, the JFA algorithm [RT06] needs to perform $\log m$ passes; in each, one voxel propagates its information to 3^d other voxels, thus the total work of JFA is $\mathrm{O}(3^d M d \log m)$. Similarly, the SKW algorithm [SKW09] needs to perform $2d$ sweepings; in each, one voxel propagates its information to 3^{d-1} other voxels. As such, the total work of SKW is $\mathrm{O}(d^2 3^{d-1} M)$. Therefore, when we increase d, the running time of JFA and SKW grows much faster than that of our algorithm.

4.2.7 Weighted centroidal Voronoi diagram

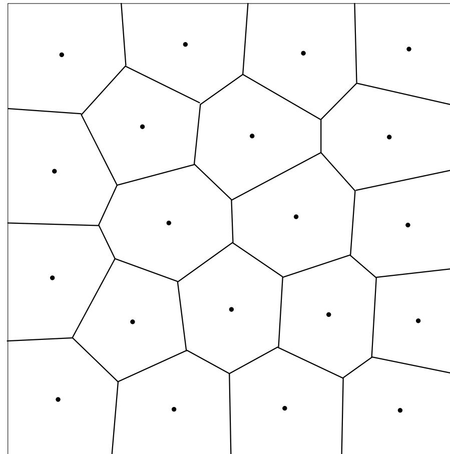


Figure 4.4: Centroidal Voronoi diagram.

In a related problem, the digital Voronoi diagram is used in computing the (*weighted*) *centroidal Voronoi diagram* (CVD), a special Voronoi diagram in which each site lies exactly at the centroid of its Voronoi cell; see Figure 4.4 for an example. The CVD can be generated from a set of input points using Lloyd's iterative algorithm [Llo82]. In each iteration, the algorithm computes the Voronoi diagram, locating the centroid of each Voronoi cell, and then moving each site to the centroid of its Voronoi cell. There were several attempts in computing the centroids of all Voronoi cells using the GPU [VSCG08, Bol09]; however, they

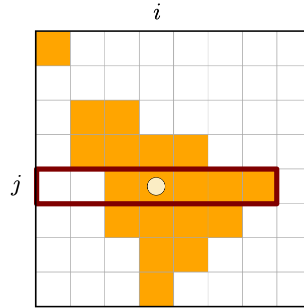


Figure 4.5: Illustration of the weighted centroidal Voronoi diagram computation.

all restrict the processing of each Voronoi cell to a preset area around the corresponding Voronoi site. They thus do not work for non-uniform distribution of input points where each Voronoi cell can possibly spread across the whole grid.

Recall that the weighted centroid of the digital Voronoi cell of $s \in S$ is defined as:

$$C_s = \frac{\sum_{p \in \mathcal{V}_D(s)} p \ w(p)}{\sum_{p \in \mathcal{V}_D(s)} w(p)},$$

where $w(p)$ is the weight of grid point p. We show how to compute the numerator, while the denominator can be computed in a similar way. Figure 4.5 shows one Voronoi cell with the site at position (i, j) and notice that the cell is not necessarily simply connected. A simple strategy is to sum up all the grid points vertically first, and then add up these partial sums to obtain the final result. Note that Lemma 4.3 states that in each column, the set of grid points which are colored by the same site is connected, forming a *chunk*. Thus, we can pre-compute the prefix sum of the weights for each grid point (i, j) along a column as:

$$\mathsf{prefix}[i, j] = \sum_{k=0}^{j-1} p_{i,k} \ w(p_{i,k}),$$

where $p_{i,k}$ is the grid point at (i, k). Then, knowing the starting and ending of a chunk, we can compute the sum of the chunk. The challenge lies in where to store the sum of each chunk for subsequent summing up as the sum of a Voronoi cell. This is discussed in the next two paragraphs.

Lemma 4.1 states that if there is a chunk of grid points in column ℓ belonging to the Voronoi region of site s at (i, j), then s must be the closest site to (ℓ, j) among all sites in row j. As such, we can store the sum of this chunk at position (ℓ, j) in a 2D array, termed *sum array*, as no other sites (in particular those in row j) would possibly need this same storage space. Thus, all the partial sums that belong to site s are stored in row j as in the red region shown in Figure 4.5. The space in the sum array is partitioned among the sites using the result of

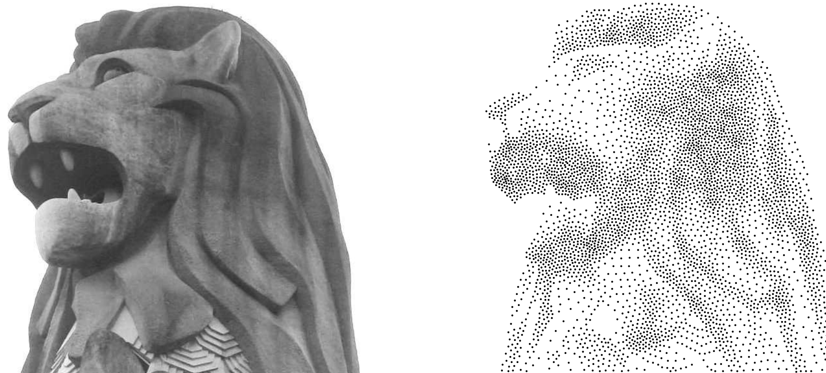


Figure 4.6: The original grayscale image (left) to produce the stipple drawing (right) by using weighted centroidal Voronoi diagram.

Phase 1 of the PBA algorithm. Once we have set up the sum array (as discussed in the next paragraph), a segmented scan of each row of the sum array gives the sum for each Voronoi cell.

To arrive at the partial sums in the sum array using CUDA, we have the following implementation. Each block of threads is used to process a column. For block ℓ processing column ℓ, we use a shared array A_ℓ of m elements. Each thread processing a grid point (ℓ, k) in column ℓ decides whether it is the topmost grid point of a chunk belonging to a site (i, j). If yes, it stores $\mathsf{prefix}[\ell, k]$ in $\mathrm{A}_\ell[j]$. Next, after synchronizing all threads, each thread processing a grid point (ℓ, k) checks whether it is the bottommost grid point of a chunk belonging to a site (i, j). If yes, it gets $\mathsf{prefix}[\ell, k+1]$, subtracts the value stored in $\mathrm{A}_\ell[j]$, and stores the result back to $\mathrm{A}_\ell[j]$. Lastly, again after synchronizing all threads in the block, we write A_ℓ into column ℓ of the sum array to complete the setup of the array. This implementation allows all global memory access to be coherent.

The running time of the algorithm is almost independent of the density of the sites, with the exception that when the number of sites is very small the algorithm runs faster. A direct application of our CVD algorithm, for example, is to create an artistic stipple drawing [Sec02]; see Figure 4.6. For such an application, there can be a large blank area without any sites, thus the Voronoi cells of sites on the boundary of this blank area can be elongated and be a challenging case to other existing works of CVD computation using the GPU [VSCG08, Bol09].

4.2.8 Experiment

In this section, we compare the performance of our PBA algorithm with two other state-of-the-art GPU algorithms, JFA by Rong and Tan [RT06] and SKW by Schiender *et al.* [SKW09].

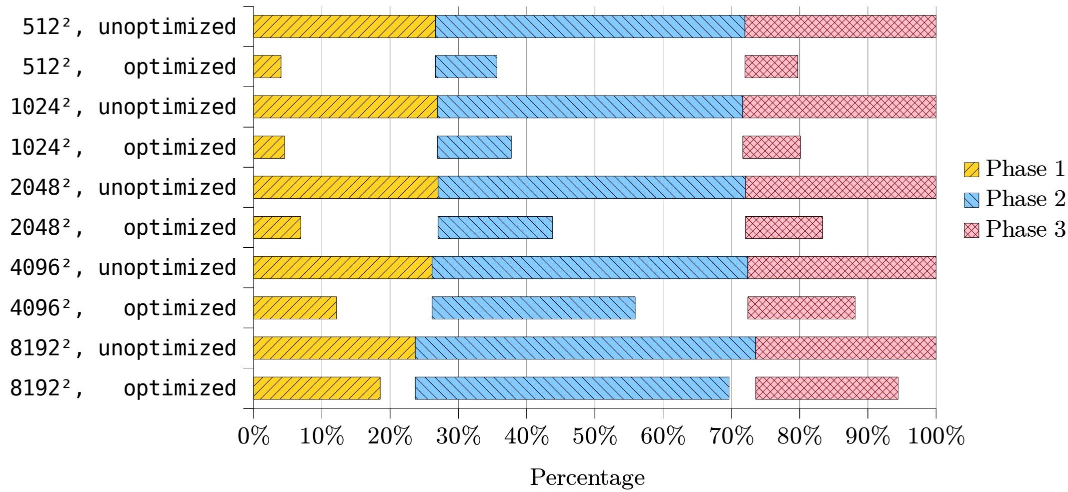


Figure 4.7: Percentage of running time of the different phases of PBA in 2D with optimized versus unoptimized parameters.

We implement both JFA and SKW using CUDA as well. JFA is a simple algorithm and there is no issue for our implementation to achieve the same performance as that in the original paper. On the other hand, there are many implementation choices for SKW. In its original paper [SKW09], the authors use an NVIDIA 8800GTX graphics card in the performance studies. We verify that our implementation of SKW has slightly better performance than in that paper using the same graphics card.

Parameters k_1, k_2 and k_3

The three parameters k_1, k_2 and k_3 are independent from each other, thus we can tune them independently to achieve the best performance on different GPUs. For our NVIDIA GTX580 graphics card, the best values to use are $k_1 = 32$ for all grid sizes; $k_2 = 16$ for grid size 512^2 and 1024^2, and $k_2 = 8$ for other grid sizes; $k_3 = 32, 16, 8, 4, 2$ for grid sizes from 512^2 till 8192^2. Using smaller k_3 for bigger grid sizes is better since the overhead is high when we use a bigger value, while the benefit of having higher parallelism is lesser when the grid gets bigger. Figure 4.7 shows the improvement in running time with the optimized choice of parameters compared to the unoptimized case where $k_1 = k_2 = k_3 = 1$. The total running time of the optimized case is normalized to that of the unoptimized case. This highlights the effect of our banding idea in the three phases of the algorithm. Notice that Phase 2 is the most time-consuming phase and the improvement with the idea of banding is very significant; Figure 4.8 shows the speedup of Phase 2 for different value of k_2. We observe that that the larger the value of k_2, the better the performance of our algorithm, until the overhead of merging dominates. Also, the benefit of our banding idea is particularly clear on small grid sizes.

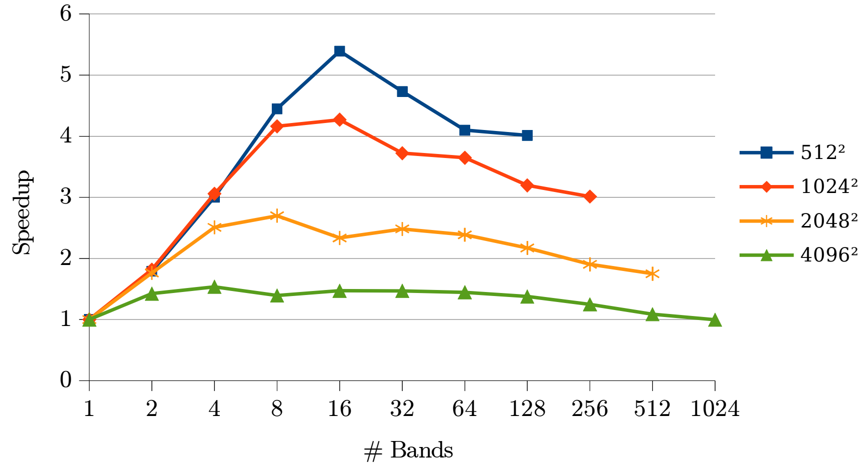


Figure 4.8: Speedup of PBA using different number of bands for Phase 2.

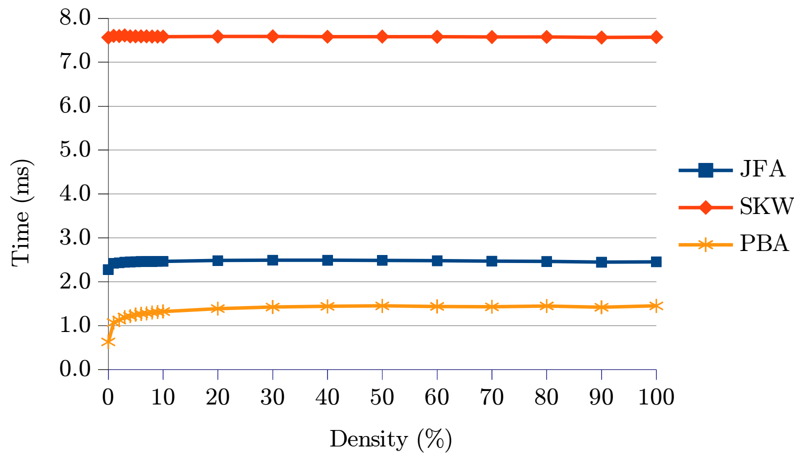


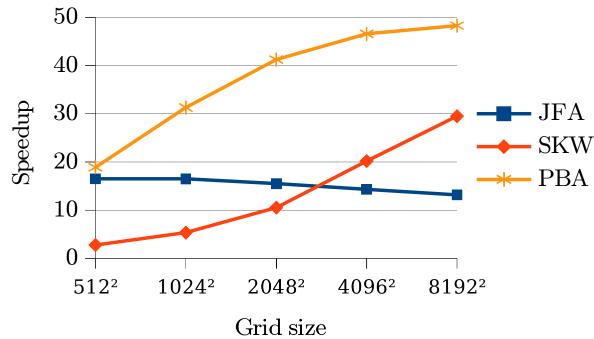
Figure 4.9: Performance of PBA in 2D while varying the density of input points.

Density of input points

The theoretical complexity of all the implemented algorithms is independent of the number of input points on the grid. However, the actual running time can be slightly affected, as shown in Figure 4.9 on a 1024^2 grid. Our algorithm is slightly faster when there are very few input points. This is probably because when the number of input points is so small, Phase 2 (which dominates the computation time) of our algorithm has very few sites to process, and the algorithm thus runs faster. With this understanding, to have a fair comparison to other algorithms, subsequent comparisons are done based on test cases with the density of input points being slightly above 10% and locations chosen randomly.

Grid size	Running time (ms)			
	CPU	JFA	SKW	PBA
512^2	10.1	0.6	3.6	**0.5**
1024^2	40.6	2.5	7.6	**1.3**
2048^2	162.2	10.5	15.4	**3.9**
4096^2	650.5	45.4	32.2	**14.0**
8192^2	2608.4	198.3	88.5	**54.0**

(a)



(b)

Figure 4.10: Running time of different 2D GPU algorithms (a), and their speedup over the sequential algorithm (b).

2D Running Time

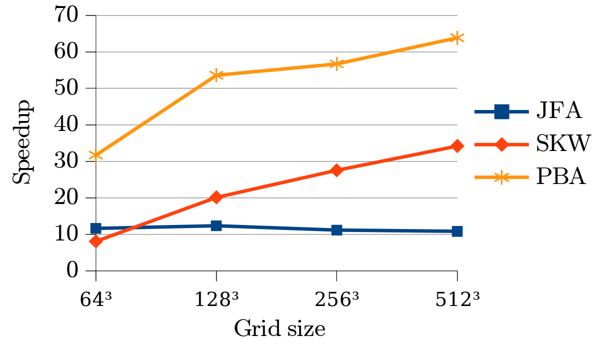
Figure 4.10a presents the running time of the CPU implementation of Maurer *et al.*'s algorithm [MQR03], PBA, as well as the two other algorithms on the GPU, using different grid sizes ranging from 512^2 to 8192^2. PBA performs significantly faster than all other GPU algorithms even though it produces an exact result while others only give an approximation. This is because PBA has a good balance of total work and level of parallelism to utilize the GPU. To appreciate the speedup of PBA better, Figure 4.10b shows the speedup of different GPU algorithms over the CPU one. Our PBA reaches around 20 to 50 times speedup, and is up to 7 times faster than SKW on small grid sizes, though this ratio drops when the grid size increases. This is due to the limited number of processors of the GPU, when the grid is very big, SKW can also have enough level of parallelism to fully utilize the GPU. On the other hand, JFA is reasonably efficient for small grid sizes but performs the worst when the grid size increases. For the largest grid size, PBA outperforms JFA by a factor of 4 times.

3D and Higher Dimensions

Figure 4.11a presents the running time of different algorithms in 3D. Clearly, our new algorithm outperforms all other algorithms for all input sizes. The speedup against the CPU algorithm ranges from 30 to 60 times. PBA is also up to 6 times faster than JFA, and around 2 to 3 times faster than SKW; see Figure 4.11b. Figure 4.12 shows the breakdown in time for each phase of our algorithm, where the optimized cases use $k_1 = 1$, $k_2 = 4$ and $k_3 = 2$. The idea of banding plays a smaller role here in improving the performance of the algorithm as there is already enough level of parallelism (for current GPU) since we now have m^2 rows of computation to be done concurrently in each of the three phases. Looking forward to the hardware that can support many more threads in the near future, our banding idea would

Grid size	Running time (ms)			
	CPU	JFA	SKW	PBA
64^3	22.2	1.9	2.7	**0.7**
128^3	172.8	14.0	8.6	**3.2**
256^3	1395.0	124.9	50.7	**24.6**
512^3	12112.9	1118.9	354.3	**189.7**

(a)



(b)

Figure 4.11: Running time of different 3D algorithms (a), and their speedup over the sequential one (b).

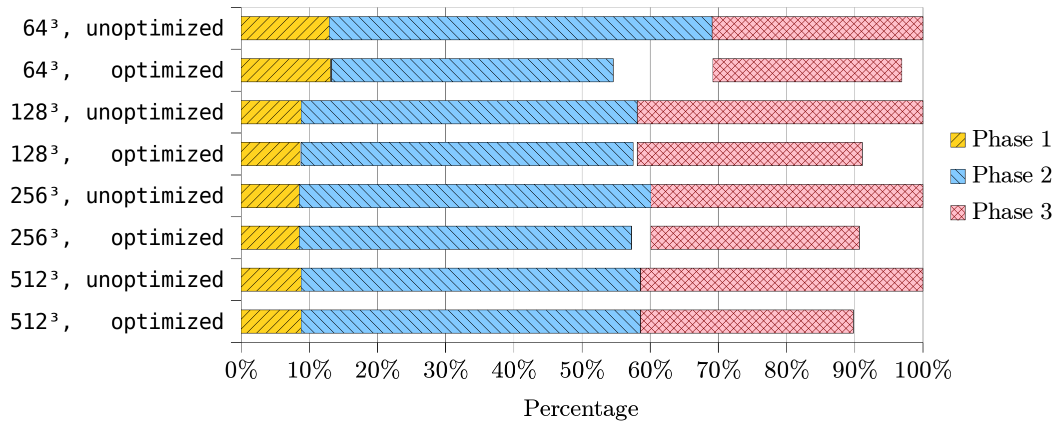


Figure 4.12: Percentage of running time of the different phases of PBA in 3D with optimized versus unoptimized parameters.

remain advantageous. Also, as mentioned before, in case we need to compute the digital Voronoi diagram slice by slice for bigger volumes, the banding idea would be beneficial as the situation is similar to the 2D case.

For the case of $d > 4$, due to the limitation on M, the size m of one dimension becomes very small (assuming a hypercube). One can trivially use Lemma 4.1 to compute the exact digital Voronoi diagram in $O(mM)$ total work. Since m is small, this algorithm achieves performance comparable to, if not better than, that of any above-mentioned algorithms due to its simplicity.

4.3 Delaunay triangulation in $\mathbb{R}^2$ - The perfect dualization

Given that the digital Voronoi diagram can be efficiently computed as discussed in the previous section, we use it as a sketch from which we derive a triangulation close to the Delaunay triangulation[1]. Following the proposed approach in Section 4.1, we have the following algorithm to compute the Delaunay triangulation for an input point set S in $\mathbb{R}^2$:

Phase 1a: Digital Voronoi diagram construction.

Map the input points into a grid and compute the digital Voronoi diagram. If multiple points overlap, then keep one and label others as missing points.

Phase 1b: Triangulation construction.

Construct a triangulation by dualizing all digital Voronoi vertices into triangles. This triangulation is an approximation of the Delaunay triangulation.

Phase 2: Shifting.

Points have been shifted due to the mapping in Phase 1a. Move them back to their original coordinates and modify the triangulation if necessary.

Phase 3a: Missing points insertion.

Insert all missing points into the triangulation.

Phase 3b: Edge flipping.

Verify the empty circle property for each edge in the triangulation, performing edge flipping if necessary.

Phase 1a and Phase 1b of the algorithm is performed in the digital space to construct a sketch of the 2D Delaunay triangulation. Phase 2 transforms this result into a triangulation in the continuous space. Phase 3a and Phase 3b subsequently complete this approximation and transform it to the final result.

We adopt the triangulation data structure used by Shewchuk [She96a] in our algorithm. A list of triangles, referred to as the *triangle list*, is stored in a pre-allocated array of size $2|S|$. Each triangle has the indices of up to three other triangles edge adjacent to it. Each point in S has a linked list of triangles incident to it. These linked lists altogether form a data structure referred to as the *vertex array*. This data structure comes in handy whenever we want to visit all triangles incident to a point.

The dualization of a grid $\mathcal{G}$ colored by the input points in S is as follows. The grid can be interpreted as a set of grid cells of unit size, whose centers being the grid points. The color of each cell is the color of its grid point. As such, a colored grid is a subdivision. In the digital Voronoi diagram, a corner shared by up to four grid cells is incident to one to four

[1] A preliminary study of this approach has appeared in the author's undergraduate thesis.

different colors. If a corner is incident to three or four colors, then it is a *digital Voronoi vertex*, and we can dualize it to form triangles.

4.3.1 Phase 1a: Digital Voronoi diagram construction

In this phase, we first translate and scale the point set such that its bounding box fits inside a 2D grid $\mathcal{G}$ of size m^2. Each point is mapped to the nearest lower left grid point. If several points are mapped to the same grid point, then only one of them is recorded while the rest become *missing points* and will be handled later.

Next, we compute the digital Voronoi diagram of all points on $\mathcal{G}$ on the GPU, using the PBA algorithm in Section 4.2. The problem is that the dual of the digital Voronoi diagram may not be a geometrically valid triangulation, since it can have duplicate and intersecting triangles; see Figure 4.13. This is mainly because each digital Voronoi cell has not only one connected component (called *bulk*) which is path-connected to its site, but also possibly some *debris* which is disconnected from that; see Figure 2.1.

To avoid the debris, we consider another way to color a 2D grid, using Algorithm 4.2, which is also referred to as the *Standard flooding algorithm*. Let Q be a priority queue storing pairs (A, s_i) of grid point $A \in \mathcal{G}$ and input point $s_i \in S$ with color (index) i. Let $\mathsf{N}(A)$ be the set of up to eight grid points neighboring grid point A. The operation $\mathsf{ExtractMin}(\mathsf{Q})$ removes and returns the pair with minimum distance; $\|A - s_i\| \prec \|B - s_j\|$ means $\|A - s_i\| < \|B - s_j\|$, or $\|A - s_i\| = \|B - s_j\|$ with consistent tie breaker (using, for example, the coordinates of those four points).

Intuitively, Algorithm 4.2 grows the regions simultaneously from the input points until they run into each other. In other words, we run multiple versions of breath-first search in parallel, making sure they do not invade each other's territory. It is easy to see that this algorithm succeeds in coloring all grid points, and each region obtained are path-connected.

In [CET14] we prove that by dualizing the output of the Standard flooding algorithm, one gets a valid triangulation of the points. Besides, the result of the standard flooding algorithm

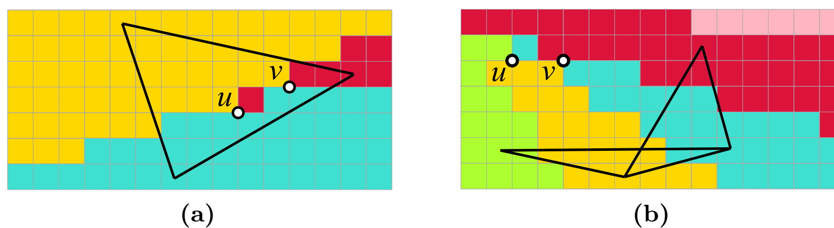


Figure 4.13: (a) Duplicate and (b) intersecting triangles due to the digital Voronoi vertices u and v.

Algorithm 4.2: The Standard flooding algorithm.

```
1  Push(Q, {(A, s_i) | site s_i is at grid point A})
2  while ¬ Empty(Q) do
3  |  (A, s_i) ← ExtractMin(Q)
4  |  if A is not colored then
5  |  |  Color A with s_i
6  |  |  Push(Q, {(B, s_i) | B ∈ N(A))
7  end
```

Algorithm 4.3: Recolor the debris in the digital Voronoi diagram.

```
1  Identify all debris as uncolor
2  Q ← ∅
3  forall the debris A do
4  |  Push(Q, {(A, s_i) | ∃B ∈ N(A) colored by s_i and ‖B − s_i‖ ≺ ‖A − s_i‖})
5  end
6  while ¬ Empty(Q) do
7  |  (A, s_i) ← ExtractMin(Q)
8  |  if A is not colored then
9  |  |  Color A by s_i
10 |  |  Push(Q, {(C, s_i) | C ∈ N(A) and ‖A − s_i‖ ≺ ‖C − s_i‖})
11 end
```

is very close to the digital Voronoi diagram constructed using our PBA algorithm. Indeed, the two results differ only at the debris. In order to obtain the result of the standard flooding algorithm, we amend the digital Voronoi diagram produced by PBA by recoloring the debris of all digital Voronoi cells, which is very few in practice. First of all, we identify and remove the color of the grid points that are debris. Then, we color these grid points using the color of their neighbors, using a priority queue just like the Standard flooding algorithm. The details are given in Algorithm 4.3. This recoloring is done on the CPU and the result is copied to the GPU to complete Phase 1a. The correctness of Algorithm 4.3 is discussed in Section 4.3.6.

4.3.2 Phase 1b: Triangulation construction

In this phase, we dualize the result of the previous phase. For each digital Voronoi vertex incident to three or four colors we add one or two triangles respectively into the triangulation. We assign one thread per grid point to analyze it, and then use parallel prefix sum to calculate the offset in the triangle list for each thread to add its triangles. The number of grid points is usually much larger than the number of threads needed to fully occupy the GPU, so we can use the banding technique similar to Section 4.2 to reduce the number of threads needed and also to reduce the cost of the prefix sum computation.

As the digital Voronoi diagram is truncated within the grid, its dual is not always a complete triangulation with a convex boundary. We fix this by traversing along the boundary of the grid, using the idea similar to Graham's scan algorithm [Gra72] to identify triangles whose Voronoi vertices fall outside the grid and add them into the triangulation. This additional step is performed on the CPU as it is a simple task, and it can be done concurrently while the GPU is populating the triangle list.

In the above computation, we want the generation of each triangle to be independent from that of other triangles in order to achieve good parallelism. Thus, each triangle is generated without linking up with the triangles sharing its edges. Once all the triangles are generated, we construct in parallel the vertex array and use that to identify for each triangle, in parallel, up to three other triangles edge adjacent to it.

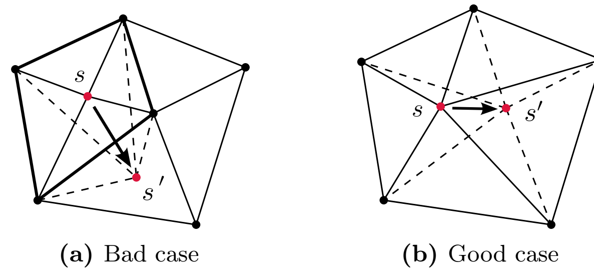
In Phase 2 and Phase 3a of our algorithm, we need to add and delete vertices of the triangulation in parallel. Deleting a vertex on the convex hull or inserting a point outside the convex hull can be quite involved, causing non-uniform parallel computations. We add a dummy infinity point into the triangulation, as described in Section 6.4, to simplify these operations.

At the end of this phase, we obtain a triangulation of the point set in digital space. This is a sketch of the Delaunay triangulation.

4.3.3 Phase 2: Shifting

Recall that during Phase 1a, points are translated, scaled, and then slightly shifted from their original positions. We reverse this process on the constructed triangulation in two steps. First, we reverse the scaling without destroying the validity of the triangulation. Second, we shift the points and fix the triangulation if necessary. The first step, though looks trivial, can result in an invalid triangulation if the possible numerical error during the scaling process is not handled. We discuss about this issue in Section 6.2.

The second step is to shift the points back to their original coordinates. We say two points are neighbors when they are endpoints of an edge in the triangulation. Assuming the neighbors



(a) Bad case (b) Good case

Figure 4.14: Shifting a point s to s' may or may not require modifications to the triangulation.

Algorithm 4.4: Shifting points of good cases and recording points of bad cases.

```
1  set all points as unchecked
2  repeat
3      for each unchecked point s_i do in parallel
4          if all neighbors s_j of s_i, j < i, have been checked then
5              if shifting s_i is a good case then
6                  update s_i to its original coordinate
7              else
8                  mark s_i as a bad case
9              mark s_i as checked
10     end
11 until all points have been checked
```

of a point s are static, shifting s may or may not cause any intersection. We refer to the former as a *bad case* and the latter as a *good case*; see Figure 4.14. The bad case happens when s moves across the boundary formed by its neighbors. After the first step, the distance between each point and its original coordinate is already very small, so we expect majority of the cases in practice are good cases.

To achieve regularized work while shifting points in parallel, we separate the processing into two stages. In the first stage, we only shift points that are good cases. To do so, we perform this stage in multiple iterations. Algorithm 4.4 details our approach. Initially, all the points are marked as unchecked. In each iteration, each thread in charge of an unchecked point s_i first verifies that all its neighbors with indices smaller than i are checked (line 4). We skip the processing of s_i if this condition is not met. Otherwise, if shifting s_i while all its neighbors remain static does not cause any intersection, then we shift it (line 5–6). If the

Algorithm 4.5: Deleting points of bad cases.

```
1  repeat
2      for each point s to be deleted do in parallel
3          if s can be processed in this iteration then
4              Mark s as active
5              Record its degree
6      end
7      Compute parallel prefix sum of the degrees
8      for each point s marked as active do in parallel
9          Mark triangles in s's fan as deleted and store their indices
10     for each point s incident to a deleted triangle do in parallel
11         Fix s's vertex array
12     for each s marked as active do in parallel
13         Triangulate the resulting star-shaped hole and update the vertex array
14     for each new triangle t do in parallel
15         Compute t's 3 neighbors and update the links between t and its neighbors
16 until all bad cases have been deleted
```

shifting creates intersections, then we leave it for the second stage (line 8). After a point is processed, it is marked as checked. Since most points would be processed and marked as checked in the first few iterations, we use compaction after each iteration to skip points that are already marked as checked in subsequent iterations. This helps speed up the later computation.

In the second stage, we delete all points that are bad cases, labelling them as missing points for later processing. Note that we also need the above-mentioned multiple iterations to avoid deleting in parallel two points that are neighbors; see the for-loop at line 2–6 of Algorithm 4.5. Here each parallel for-loop is a CUDA kernel, with a global synchronization at the end. First, for each set of points to be deleted in parallel, we count their degrees. Applying parallel prefix sum on these counts, we get the starting position that each thread might use to store the indices of the triangles to be deleted (line 7). These indices are needed to store the newly created triangles during the re-triangulation. We allocate a piece of memory for each thread to store the indices mentioned above, all of which form a single array. Second, using one thread per point in parallel, we mark all triangles in its fan as deleted and store their indices in the allocated memory (line 8–9). Third, we fix the vertex array in parallel by removing the deleted triangles from it (line 10–11). After that, we use the ear-cutting method [Hig82] to

re-triangulate the resulting star-shaped holes in the triangulation in parallel, while updating the vertex array at the same time (line 12–13). Since the triangles to be created is no more than those deleted, we can use the deleted slots in the triangle list that we recorded earlier to store the new triangles, with no racing memory access during parallel computation. In the end, we update the links between the newly created triangles and those that are edge adjacent to them (line 14–15).

4.3.4 Phase 3a: Missing points insertion

In this phase, we insert the missing points identified in Phase 1a and Phase 2 into the triangulation. The insertion of each missing point s_i starts by locating the triangle, referred to as the *container*, that contains s_i or has an edge passing through s_i. To locate the triangle containing s_i, we could start from a random triangle and walk towards s_i; however, the closer to s_i we start the walking, the faster we locate its container. As such, if s_i is due to Phase 1a, we start walking from a triangle incident to the point of S that was mapped to the same grid point as s_i and was kept in the digital VD computation. If s_i is due to Phase 2, we start walking from a triangle incident to a neighbor s_j of s_i when we delete s_i. Such a neighbor s_j is recorded during Phase 2. Note that if s_j is no longer in the triangulation, then we delay the insertion of s_i to a later iteration, until s_j is inserted. The point location is done in parallel with one thread handling one missing point.

To insert the missing point s_i, we simply split its container into three new triangles. To avoid concurrent modification of the same triangle during the parallel execution, this process is also done in multiple iterations; see Algorithm 4.6. In each iteration, each thread handling a missing point s_i first uses the index i to mark on the container (line 2–5). Next, each thread checks the mark on the triangle and only performs the insertion if its mark is not overwritten by any other threads (line 7). The marking is done using the atomicMinimum operation, which is readily available on the GPU. This avoids any possible live-lock situation, since in each iteration at least the missing point with the smallest index can be inserted.

For each missing point that can be inserted, we mark its container as deleted (line 9). Otherwise, we record its container for point location in later iterations (line 11). After that, we fix the vertex array of points that are incident to some deleted triangles (line 13–14). Then, we generate new triangles, update their neighbors correspondingly, and update the vertex array of points that are incident to some new triangles (line 15–18), similar to the process in Phase 2. One small implementation note is on the insertion of new triangles into the triangle list in parallel. We have to re-use all the deleted slots in the previous phase to make sure that we do not have to resize the triangle list. This can be done by first collecting the list of deleted slots using parallel stream compaction. Each missing point being inserted needs up to two new slots. We can use parallel prefix sum to allocate the available slots in the triangle list to each thread.

Algorithm 4.6: Inserting missing points.

```
1  repeat
2      for each yet-to-be-inserted missing point s_i do in parallel
3          locate the container of s_i
4          mark the triangle containing s_i with i using AtomicMinimum
5      end
6      for each yet-to-be-inserted missing point s_i do in parallel
7          if its container is still marked as i then
8              mark s_i as active
9              mark its container as deleted
10         else
11             record the triangle found for s_i for future point location
12     end
13     for each point s incident to a deleted triangle do in parallel
14         Fix the vertex array of s
15     for each point s marked as active do in parallel
16         Insert s into its container to create new triangles and update the vertex array
17     for each new triangle t do in parallel
18         Find t's neighbors and update the links between them and t
19 until all missing points have been inserted
```

At the end of this phase, we compact the triangle list to remove all slots that are unused. This changes the index of the triangles, so we have to update all the references to them.

The missing point insertion phase might potentially need many iterations to complete all the insertions. This is because missing points might fall into the same triangle; and there could be many of them since we do not explicitly control how many input points can potentially be mapped to a same grid point. The latter problem is the inherent limitation of our digital approach, and we shall see its impact in the experiment with different input datasets. The former problem is not quite apparent in practice, since even though there might be one triangle containing many missing points, after a few iterations, the triangle is subdivided into many more triangles, and the missing points are distributed across them if the order of missing point insertions is random.

4.3.5 Phase 3b: Edge flipping

This phase transforms the current triangulation into the Delaunay triangulation. We verify the empty circle property of each edge in our triangulation in parallel. For an edge ab of $\triangle abc$, we check if the point d of the adjacent $\triangle adb$ is inside the circumcircle of $\triangle abc$. If so, an edge flipping is performed to replace $\triangle abc$ and $\triangle adb$ with $\triangle adc$ and $\triangle cdb$. Note that such an edge is always flippable. This process is done in multiple iterations, and the same strategy as in the previous phase is used to avoid concurrent modification of a triangle by multiple threads. We use one thread to process one triangle, and we mark those that do not need any flipping so we do not need to check them again in the next iterations. That mark needs to be removed when the triangle is later modified.

One difficulty during this phase is the updating of the links between the triangles after each iteration of flipping. Two adjacent triangles can participate in two different flips, thus directly updating the adjacent triangles after flipping can cause conflicting memory access. Instead, the update is performed in two separate GPU kernels. Each triangle has a temporary storage for updating its links. In the first step, each pair of triangles that is just flipped updates the temporary storage of its neighbors. In the second step, each pair of triangles mentioned above inspects its temporary storage and updates its own links. Also, if any neighbor of this pair is not flipped in this iteration, then we directly update this neighbor's links.

One optimization for this phase is that a triangle t_i only performs the in-circle test with its neighbor t_j if $i < j$, since each pair of triangles should only be checked at most once in each iteration.

4.3.6 Proof of correctness

We only give the proof the correctness of Phase 1a, as that of other phases of our algorithm is quite straightforward. We need to prove that using Algorithm 4.3 to recolor the digital Voronoi diagram produces the same output as that of the standard flooding algorithm. The correctness of Phase 1b then follows from our proof in [CET14]. For ease of understanding, we include here some properties of the result obtained by the standard flooding algorithm.

Lemma 4.7 (Ordered Coloring Lemma)**.** *At any iteration of the standard flooding algorithm, for every two input points $s_i, s_j \in S$, we have $\|A - s_i\| \prec \|Y - s_j\|$ for all A colored by s_i and $(Y, s_j) \in \mathcal{Q}$. In other words, flooding colors the grid points in the order of their distance from the site.*

Lemma 4.8 (Monotonic Path Lemma)**.** *For each grid point A in the region colored by s_i in the standard flooding, there is a monotonic path, i.e. a connected sequence of grid points with increasing distance to s_i, from s_i to A and is completely colored by s_i.*

Lemma 4.9 (Bulk Lemma)**.** *The region colored by s_i using standard flooding contains the bulk of s_i in its digital Voronoi diagram.*

Clearly, from the Bulk Lemma, the difference between the standard flooding result and the digital Voronoi diagram is only at the debris. We now show by contradiction that Algorithm 4.3 indeed produces the same output as that of the standard flooding algorithm. Consider the very first instance when our algorithm colors debris A with color r (inside the while-loop) whereas the standard flooding produces grid point A with color $t \neq r$. There are two situations to consider but both lead to a contradiction and thus no such debris exists:

Case 1: $\|A - s_r\| \prec \|A - s_t\|$

From Algorithm 4.3, there exists a neighbor B of A colored by s_r earlier, and $\|B - s_r\| \prec \|A - s_r\|$. It follows that $\|B - s_r\| \prec \|A - s_r\| \prec \|A - s_t\|$. According to our choice of A, the grid point B is also colored by s_r in the standard flooding result. By the Ordered Coloring Lemma, (A, s_r) must have been considered in the standard flooding algorithm before (A, s_t) and A should thus have been colored by s_r, a contradiction.

Case 2: $\|A - s_t\| \prec \|A - s_r\|$

According to the Monotonic Path Lemma, there is a monotonic path from s_t to A. A part of this path has been colored the same (with s_t) in our algorithm when we are about to color A. Let C be the grid point closest to s_t on that path that is not yet colored by our algorithm. With the previous grid point before C having been colored with s_t, (C, s_t) must have been added to Q in our algorithm. Since $\|C - s_t\| \prec \|A - s_t\| \prec \|A - s_r\|$, we must have (C, s_t) inside Q be extracted before (A, s_r), a contradiction.

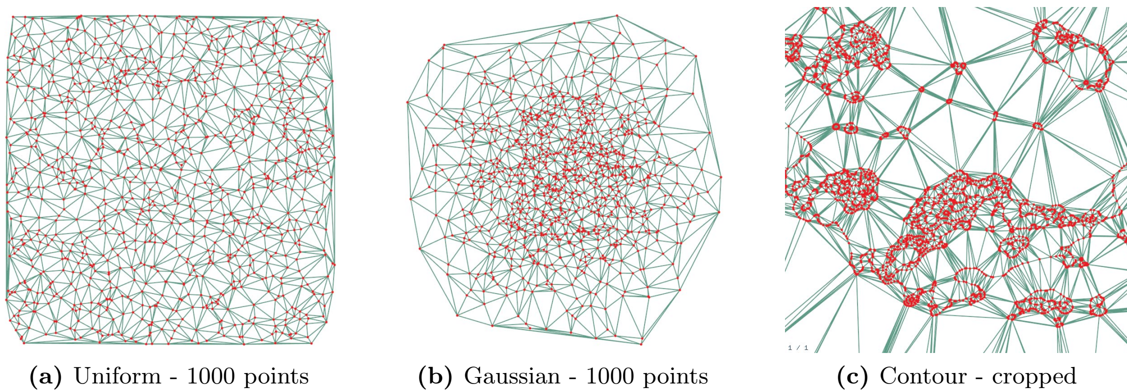
The argument in Case 2 also implies that the algorithm colors all debris. This concludes our claim of correctness of our debris coloring algorithm.

4.3.7 Experiment

To assess the efficiency of our implementation, termed DigiDel2D, we compare its running time, on both synthetic and real-world data, with those of the most popular 2D Delaunay triangulation libraries available, Triangle and CGAL version 4.2. The experiment setting is described in Section 2.3; see Figure 4.15 for some sample results of DigiDel2D. Note that the output of DigiDel2D agrees with that of Triangle and CGAL except when some input points are cocircular, in which case there are more than one possible Delaunay triangulation.

Synthetic data

Figure 4.16a presents the running time of DigiDel2D on point sets of different size from the uniform distribution. The grid sizes used for the digital Voronoi diagram computation



(a) Uniform - 1000 points (b) Gaussian - 1000 points (c) Contour - cropped

Figure 4.15: Sample output of DigiDel2D on different point sets.

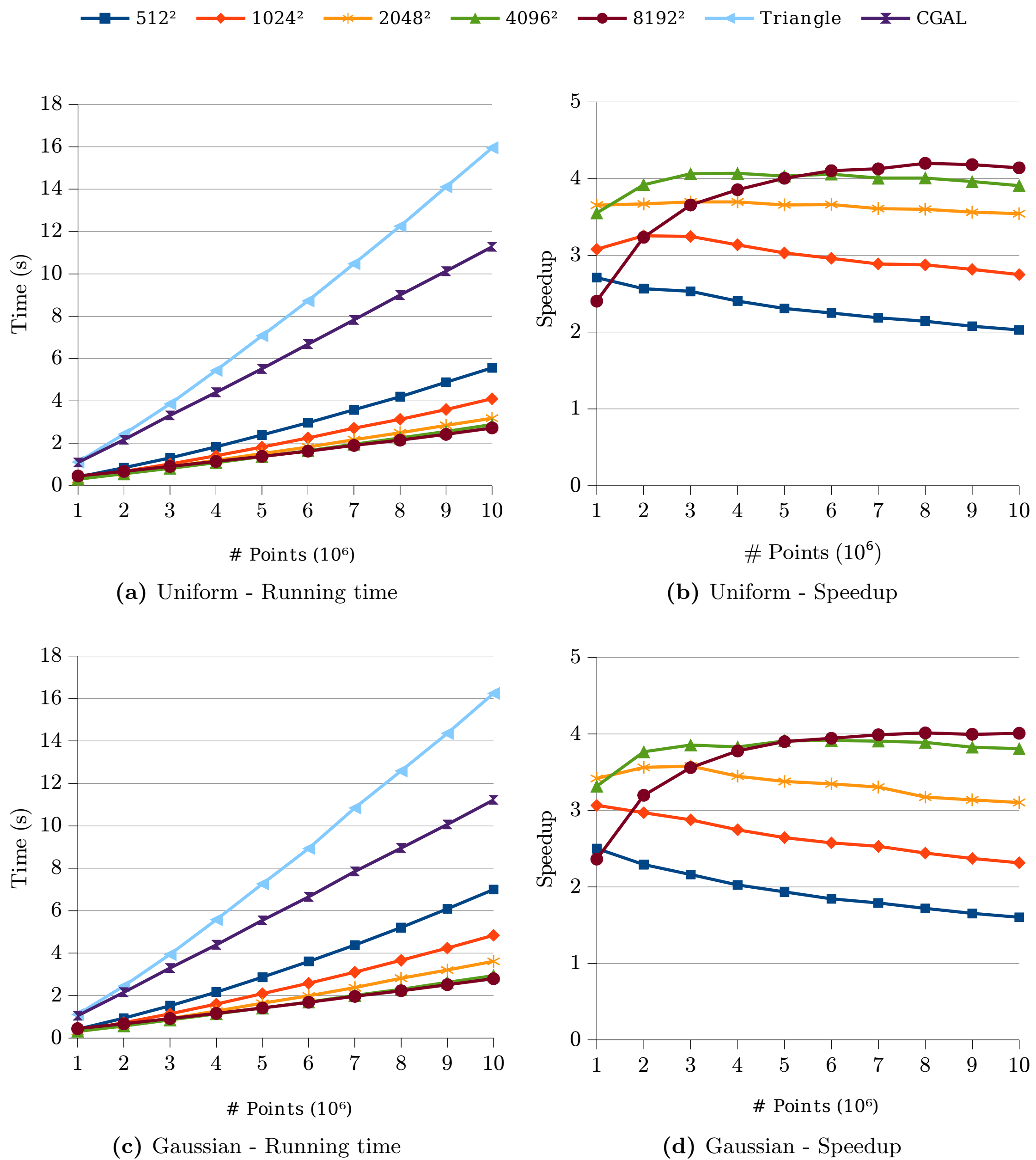
significantly affect the running time of our program, with a larger grid giving faster running time overall. It can also be observed that the running time of both DigiDel2D and the two CPU programs are nearly linear to the number of input points. CGAL achieves better running time than Triangle in all of the tests we run.

Figure 4.16b shows that using grids of size 4096^2, DigiDel2D achieves on average 4 times speedup over CGAL. With an 8192^2 grid size, the running is slower on small input sizes, but quickly catches up and surpasses the speed when using 4096^2 grid size on very large input. On the other hand, with very small grid size such as 512^2, the speedup of DigiDel2D over CGAL drops when the number of input points increases, since such small grid size is too small to provide a good approximation to the Voronoi diagram of the large input point.

The same conclusion is also true when the input points are from a Gaussian distribution; see Figure 4.16c and 4.16d. For small grids, the speedup achieved is slightly lower than with the uniform distribution due to having more points concentrated in the center of the grid and became missing points. The speedup increases quickly when we move to a larger grid, being comparable to the uniform case. This, however, is no longer true on extreme cases such as when points are from the thin circle distribution, or even co-circular, since our uniform digital Voronoi diagram is not a good approximation of the continuous one in this case. Most of the points will be missing after Phase 1a. Nevertheless, our implementation can still handle these cases.

Real-world data

Figure 4.17 provides the running time of DigiDel2D on several contour map datasets of different sizes. Only points on the contours are used as input. In these datasets, points are not uniformly distributed but instead different area has different point density, with points distributing along some curves, leading to Delaunay triangulations with significantly



(a) Uniform - Running time

(b) Uniform - Speedup

(c) Gaussian - Running time

(d) Gaussian - Speedup

Figure 4.16: The running time and speedup of DigiDel2D on uniform and Gaussian point distribution, running with different grid size, compared to those of Triangle and CGAL. The speedup is computed with respect to the running time of CGAL.

Set	# Vertices	*Triangle*	CGAL	DigiDel2D					Speedup over CGAL (max)
				512^2	1024^2	2048^2	4096^2	8192^2	
1	1,177,332	1.37	1.29	0.74	0.64	0.58	0.55	0.72	**2.4**
2	3,180,037	4.19	3.58	2.35	1.80	1.53	1.34	1.39	**2.7**
3	4,461,519	6.34	5.05	3.93	2.83	2.27	1.96	1.93	**2.6**
4	5,721,142	8.46	6.55	5.87	4.05	3.13	2.67	2.50	**2.6**
5	8,569,881	13.60	9.96	10.70	6.86	5.06	4.15	3.68	**2.7**

Figure 4.17: The running time (in seconds) and speedup of DigiDel2D on some contour datasets, running with different grid size, compared to those of Triangle and CGAL.

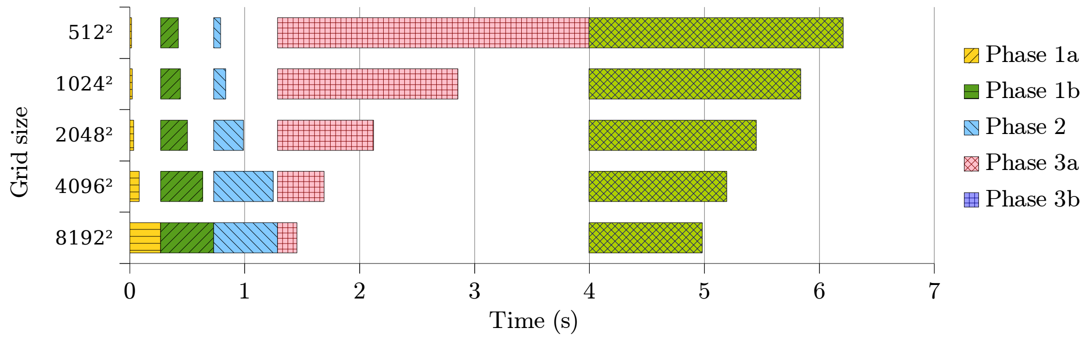


Figure 4.18: The running time of different phases of DigiDel2D.

different vertex degrees. Nevertheless, DigiDel2D can still achieve around 2.5 times speedup over CGAL on all test cases. This speedup, however, is definitely not as good as when points are uniformly distributed, since there can be a lot more missing points.

Time breakdown

We analyze the time spent in different phases of our algorithm in Figure 4.18. Inputs of 10^7 points are used in this experiment. Here a larger grid has some penalty on the digital Voronoi diagram computation time in Phase 1a and the dualization in Phase 1b. Also, more points (that can be mapped onto the grid) are to be shifted in Phase 2. However, it gives a better approximation and less missing points, thus Phase 3a and Phase 3b are significantly faster, contributing to a faster running time overall. In Phase 3b, the number of flips required decreases significantly as the grid size increases, dropping as much as 6 times when moving

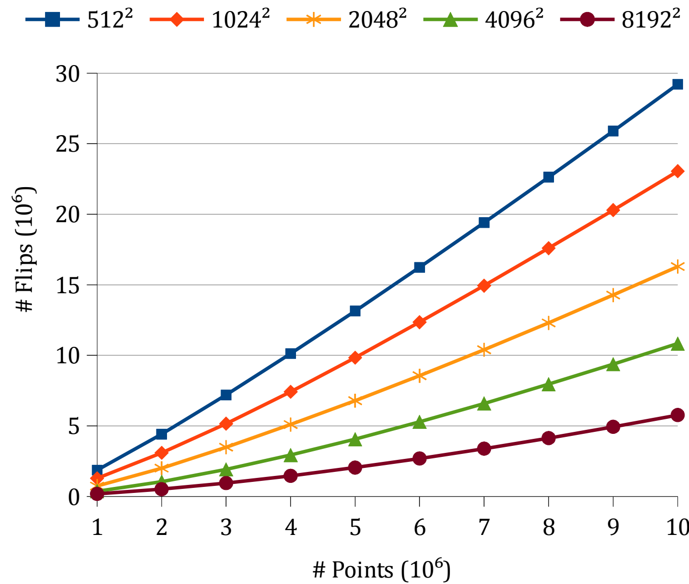


Figure 4.19: The number of flips performed in Phase 3b of DigiDel2D.

from a 512^2 grid to a 8192^2 grid, as show in Figure 4.19. All in all, as a general guideline, given a larger set of input points, a larger grid is preferable.

4.4 Convex hull in $\mathbb{R}^3$ - The digital depth test

In this section, we apply our digital approach to construct the convex hull of a point set $S \in \mathbb{R}^3$ on the GPU. The main idea of our algorithm is similar to that in the previous section, to use the relation between the 3D Voronoi diagram and the convex hull of S. In general, the Voronoi cells of the extreme vertices are unbounded, i.e. they extend to infinity. More specifically, consider a box containing S, the intersection of this box with the 3D Voronoi diagram of S is the dual of the convex hull boundary when the size of the box goes to infinity.

Traditionally, this observation is not computationally useful as the Voronoi diagram (in continuous space) is a structure harder to manage than the convex hull itself. However, we can adapt this to our digital approach. Consider the input point set S enclosed in a large enough grid $\mathcal{G}$, we are interested in six slices of $\mathcal{V}_D(S)$ when intersected with the boundary faces of $\mathcal{G}$. Unfortunately, the proof in Section 4.3.6 and [CET14] cannot be extended to this case, and thus we cannot obtain a proper polyhedron from the dual of these slices. Nevertheless, this dual is still very close to be a polyhedron, and is also close to the convex hull, so we employ the star splaying algorithm as described in Section 3.2.3 to transform it to the convex hull. Our algorithm is as follows:

Phase 1a: Voronoi construction.
Construct six slices $\mathcal{V}_i(S)$ of $\mathcal{V}_D(S)$ when intersected with the boundary face i of $\mathcal{G}$. We denote $S' \subseteq S$ as the set of points with non-empty Voronoi cells in some slices.

Phase 1b: Star identification.
Compute a convex star for each point $s \in S'$ from the neighbourhood information obtained from each $\mathcal{V}_i(S)$.

Phase 2: Hull approximation.
Splay stars of points in S' to arrive at $\mathcal{C}(S')$.

Phase 3a: Point addition.
For each point in $S'' = S \setminus S'$ that is outside $\mathcal{C}(S')$, construct a convex star for it.

Phase 3b: Hull completion.
Apply star splaying once more to compute $\mathcal{C}(S' \cup S'')$, which is also $\mathcal{C}(S)$.

Digital computation is actually applied at two places in this algorithm: during the construction of the sketch in Phase 1a, and during the point addition in Phase 3a. In Phase 3a, a digital depth test is used to identify extreme vertices that were missed in Phase 1a. By using a carefully chosen error bound, we can guarantee the correctness of that digital computation.

4.4.1 Phase 1a: Voronoi construction

We compute six slices of the 3D digital Voronoi diagram of S intersecting with the boundary faces of $\mathcal{G}$ using the method described in Section 4.2.6; see Figure 4.20a for a sample result. Points in S are first shifted to the nearest grid points in $\mathcal{G}$, and then projected onto the corresponding face of the grid, with only the nearest one being kept per grid point. To avoid a huge amount of random memory access in the GPU during this projection, we use a bucket sort technique. For each face of the grid, represented as a 2D grid, we divide it into rectangular regions of small size. Using the GPU radix sort, we can quickly re-arrange the point set such that points projected onto the same region are grouped together. The size of a region is small enough to fit into the on-chip shared memory. We let one block of threads project all points of the same region onto a shared memory array first, using the AtomicMinimum operation to keep the nearest points only, before coherently write out the result to the global memory.

Since the grid $\mathcal{G}$ has finite volume, some bounded Voronoi cells can extend beyond $\mathcal{G}$ and are thus captured on the digital Voronoi diagram slices, although they do not correspond to extreme vertices. To reduce the number of wrongly captured Voronoi cells, we scale the point set to a slightly smaller volume, about 80%, inside $\mathcal{G}$ before the digital Voronoi diagram computation.

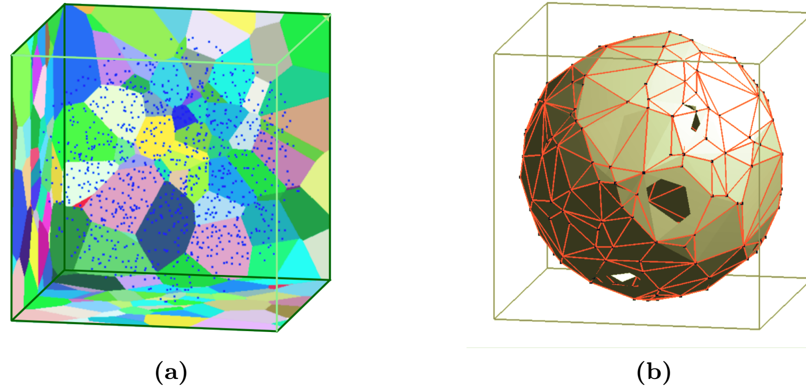


Figure 4.20: (a) The intersection of some faces of $\mathcal{G}$ with the digital Voronoi diagram in Phase 1a. (b) Stars constructed in Phase 1b; they might not be consistent.

4.4.2 Phase 1b: Star identification

The aim here is to quickly derive a sketch of the convex hull from the above digital Voronoi diagram slices $\mathcal{V}_i(S)$. This sketch is obtained as a collection of convex stars of points with non-empty digital Voronoi cells in $\mathcal{V}_i(S)$ for some i. We first dualize $\mathcal{V}_i(S)$ similar to Section 4.3.2, but instead of creating a triangulation, for each vertex we only keep the edges incident to it; those vertices at the other end of these edges are the vertex's neighbors. Then we apply the beneath-beyond algorithm [Kal81], inserting these neighbors one by one to construct its convex star. Note that the stars are not necessarily consistent with one another, as shown in Figure 4.20b. The construction is embarrassingly parallelizable, with one thread handling one star.

4.4.3 Phase 2: Hull approximation

We adapt the star splaying algorithm of Shewchuk [She05] to the GPU. The adaptation allows multiple consistency checking and point insertions to be performed in parallel. We divide the star splaying algorithm into two stages: the *checking stage* and the *inserting stage*, to be alternately performed until all the stars are consistent (i.e. no more edges to check). In the checking stage, we gather all edges that need to be checked and assign each to a thread for consistency checking, as described in Section 3.2.3. If an edge fails the consistency check, then we create up to four insertions for the next stage. In the inserting stage, we gather (with sorting) each set of points to be inserted into the same star and assign it to a thread to perform the insertions. For each new edge created, we mark it for checking later. If an existing edge ab in the star of a is removed, then the edge ba in the star of b (if exists) is marked for checking. Due to the good sketch obtained, we expect the inconsistency to be

minor, as can be seen in Figure 4.20b.

The data-structure for the stars is as follows. To represent a star, we actually store its link, i.e. a set of link points in counter-clockwise order looking from the vertex. Each star is stored in a continuous chunk of memory inside an array called the *star list*. During the splaying, if the size of a star, i.e. the number of link points, increases beyond the size of its memory storage, then we need to resize the whole array.

4.4.4 Phase 3a: Point addition

The aim of this phase is to recover extreme vertices of S that were missed in Phase 1a. Basically, we want to find all the points in S that are outside $\mathcal{C}(S')$. We utilize the graphics rendering pipeline (through OpenGL) to achieve good performance. The idea is to perform two rounds of checking: the first round in the digital space to handle most of the simple cases, before the second, more accurate yet costly checking round in the continuous space. In the first round of checking, we render the triangle faces of $\mathcal{C}(S')$ with the view direction orthogonal to each face of $\mathcal{G}$ in turn. There are six orthogonal directions in total. For each rendering, we use the color and the depth buffer to record the index and the depth value, respectively, of the triangle covering each pixel. Then, we project each point $s \in S \setminus S'$ in the corresponding viewing direction and compare the depth value d_s of s with the depth value d in the depth buffer at the corresponding position. If $d_s - d \leq \tau$ where τ is a constant threshold (which is equal to 1 pixel width; see Section 4.4.6 for the proof of correctness) then s is potentially outside $\mathcal{C}(S')$, and is subject to the second round of checking.

After the first round, most of the points that are clearly inside $\mathcal{C}(S')$ would have been removed. For each point $s \in S$ that is potentially outside, we also record a triangle that covers its projection in one of the viewing direction. Pick a point p inside $\mathcal{C}(S')$ as the viewpoint, using a technique similar to point location, starting from the recorded triangle, we can quickly walk to the triangle t intersected by the ray $\overrightarrow{ps}$ and accurately determine whether s is inside or outside. If s is outside, then we include s in S'' and use three vertices of t to form the initial star of s. This second round of checking can be done in parallel for each point that is potentially outside, using one thread each.

4.4.5 Phase 3b: Hull completion

This phase is exactly the same as Phase 2. We apply the GPU star splaying algorithm to make all the stars after Phase 3a consistent with one another. Thereafter, the set of stars form the convex hull of S.

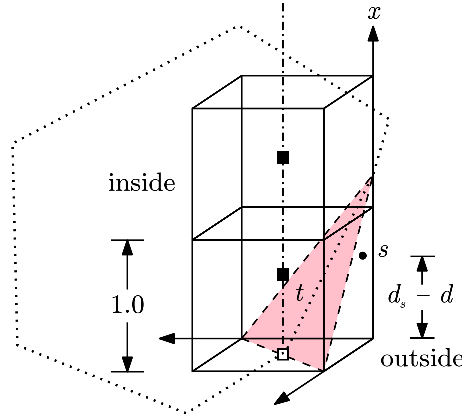


Figure 4.21: The digital depth test of a point s against a triangle t on the boundary of $\mathcal{C}(S')$ when s is outside $\mathcal{C}(S')$.

4.4.6 Proof of correctness

The correctness of our algorithm is straightforward, except for the possible error during Phase 3a where we use the digital depth test to determine if a point is possibly outside $\mathcal{C}(S')$ or not. In this section, we prove that using the error bound $\tau = 1$ as described in Section 4.4.4, the result of Phase 3a is guaranteed to be correct.

In the digital depth test in Phase 3a, we use the six faces of $\mathcal{G}$ as six viewing planes. We compare the depth d_s of each point s with the minimum depth value of $\mathcal{C}(S')$ at the corresponding projection of s to quickly exclude points that are inside $\mathcal{C}(S')$. However, since the depth buffer we obtain when rendering $\mathcal{C}(S')$ is of finite resolution, the depth value d of the projection of s is actually the depth value at the center of the pixel containing this projection; see Figure 4.21. Depending on the triangle covering that projection, $(d_s - d)$ can be arbitrarily large. The following lemma shows that as long as we keep every point s that has $(d_s - d) \leq 1$ in one of the projections, we do not miss any point outside $\mathcal{C}(S')$. For simplicity, we assume that the rendering buffer has the same size as that of $\mathcal{G}$, whereas in actual implementation this is not needed.

Lemma 4.10. *Let $s \in S \setminus S'$ be a point outside $\mathcal{C}(S')$. In (at least) one of the six renderings of $\mathcal{C}(S')$ orthogonal to a face of $\mathcal{G}$, we have $(d_s - d) \leq \tau$ where $\tau = 1$ pixel width.*

Proof. The point s is inside a grid cell of $\mathcal{G}$ whose center is the grid point $A = (\bar{x}, \bar{y}, \bar{z})$. The coordinate of s is $(\bar{x} + \delta_x, \bar{y} + \delta_y, \bar{z} + \delta_z)$ where $\delta_x, \delta_y, \delta_z \in [-0.5, 0.5]$. Let t be the triangle covering A in different view directions, and the plane equation of t is $ax + by + cz + K = 0$. Without loss of generality we assume that $a \geq b \geq c$.

Since t appears in the depth buffer and $\mathcal{C}(S')$ is convex, t must be visible from three different viewing directions. This forms a coordinate system in which the plane equation

of t has $a, b, c \geq 0$. In the viewing direction along the positive x-axis, $d_s = \bar{x} + \delta_x$ and d is the depth of t at $(\bar{y}, \bar{z})$. As s is outside $\mathcal{C}(S')$ and thus is in front of the plane of t, $a(\bar{x} + \delta_x) + b(\bar{y} + \delta_y) + c(\bar{z} + \delta_z) + K \leq 0$, and we thus have:

$$\begin{aligned}
d_s - d &= (\bar{x} + \delta_x) - \left(-\frac{b\bar{y} + c\bar{z} + K}{a}\right) \\
&= \frac{a(\bar{x} + \delta_x) + b(\bar{y} + \delta_y - \delta_y) + c(\bar{z} + \delta_z - \delta_z) + K}{a} \\
&= \frac{a(\bar{x} + \delta_x) + b(\bar{y} + \delta_y) + c(\bar{z} + \delta_z) + K}{a} - \frac{b\delta_y}{a} - \frac{c\delta_z}{a} \\
&\leq -\frac{b\delta_y}{a} - \frac{c\delta_z}{a} \\
&\leq \frac{b}{2a} + \frac{c}{2a} \\
&\leq 1
\end{aligned}$$

It is possible that the depth values used in the checking of s in the six viewing directions belong to different triangles. Suppose that the depth value of t is used in one of the directions, then from the above argument, there is one direction in which the depth d of the plane containing t fulfills the inequality $(d_s - d) \leq 1$. Suppose t' is the other triangle that covers s in that direction, then due to the convexity of $\mathcal{C}(S')$, the depth d' of t' must be no smaller than d, and thus $(d_s - d') \leq 1$, as required. □

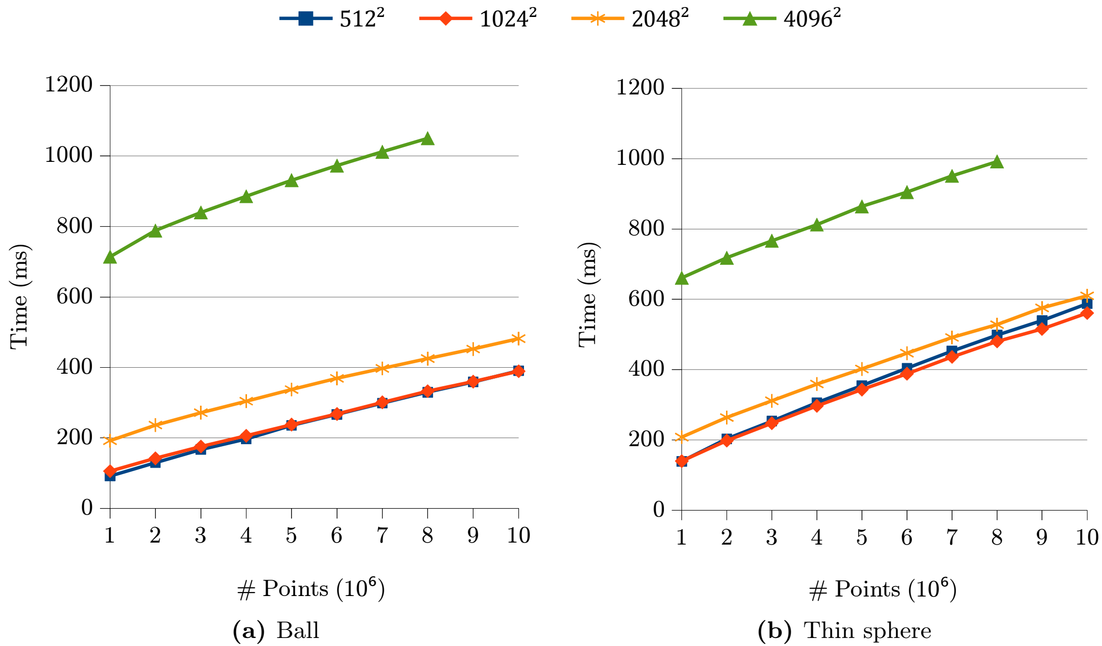
4.4.7 Experiment

We compare the performance of our implementation, called DigiHull3D, with the two fastest sequential implementations of the Quickhull algorithm: Qhull [BDH96] and CGAL [CGA]. Qhull handles round-off error from floating point arithmetic by generating a convex hull with "thick" facets: any exact convex hull must lie between the inner and outer plane of the output facets. On the other hand, CGAL uses exact arithmetic, which is similar to our implementation. In our experiment, we found that CGAL always runs slower than Qhull due to its use of exact arithmetic.

The performance of DigiHull3D can be controlled by two parameters: the grid size (in Phase 1a) and the rendering buffer size (in Phase 3a). As such we first analyze the effect of these parameters before comparing our algorithm with others.

Grid size and rendering buffer size

We run DigiHull3D on two different input point distributions: the ball distribution and the thin sphere distribution, with different grid sizes; see Figure 4.22. 5×10^6 points are used in



(a) Ball

(b) Thin sphere

Figure 4.22: The total running time of DigiHull3D using different grid size on the ball and the thin sphere distribution.

this experiment. Different from the experiment of DigiDel2D, using larger grid size makes the running time much slower, except in the case of the thin sphere distribution where the running time improves a bit when moving from 512^2 grid to 1024^2 grid. There are two reasons for this behavior. First, DigiHull3D needs to compute six digital Voronoi diagram slices, thus the running time of Phase 1a is quite significant, and therefore any increases in the grid size affects the total running time a lot. Second, the number of extreme points is usually small, so the number of Voronoi cells intersecting a boundary face of $\mathcal{G}$ is small, and thus a small grid size still has no problem providing a good approximation. This is further illustrated in Figure 4.23a. Increasing the grid size decreases the running time of star splaying in Phase 3b since fewer points are added in Phase 3a, but the overhead in the digital Voronoi diagram computation in Phase 1a makes the overall running time much slower.

The same behavior is observed for the rendering buffer size in Phase 3a. A larger buffer size makes the second round of checking in the point addition phase faster since the digital depth test in the first round of checking is more accurate. However, the cost of rendering to a larger buffer size outweighs this benefit, making the overall running time of this phase slower; see Figure 4.23b. As such, in the rest of the experiment, we use a grid of size 1024^2 and a rendering buffer size 512^2 for best performance of DigiHull3D.

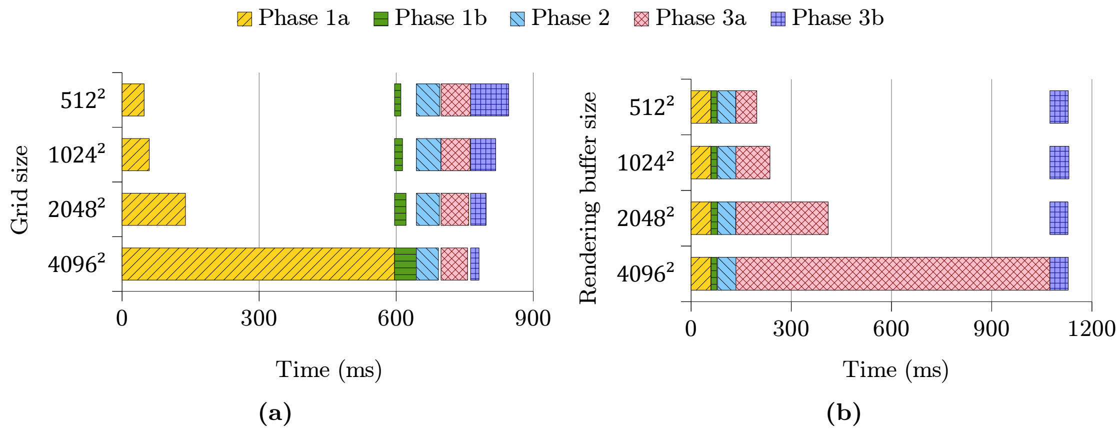


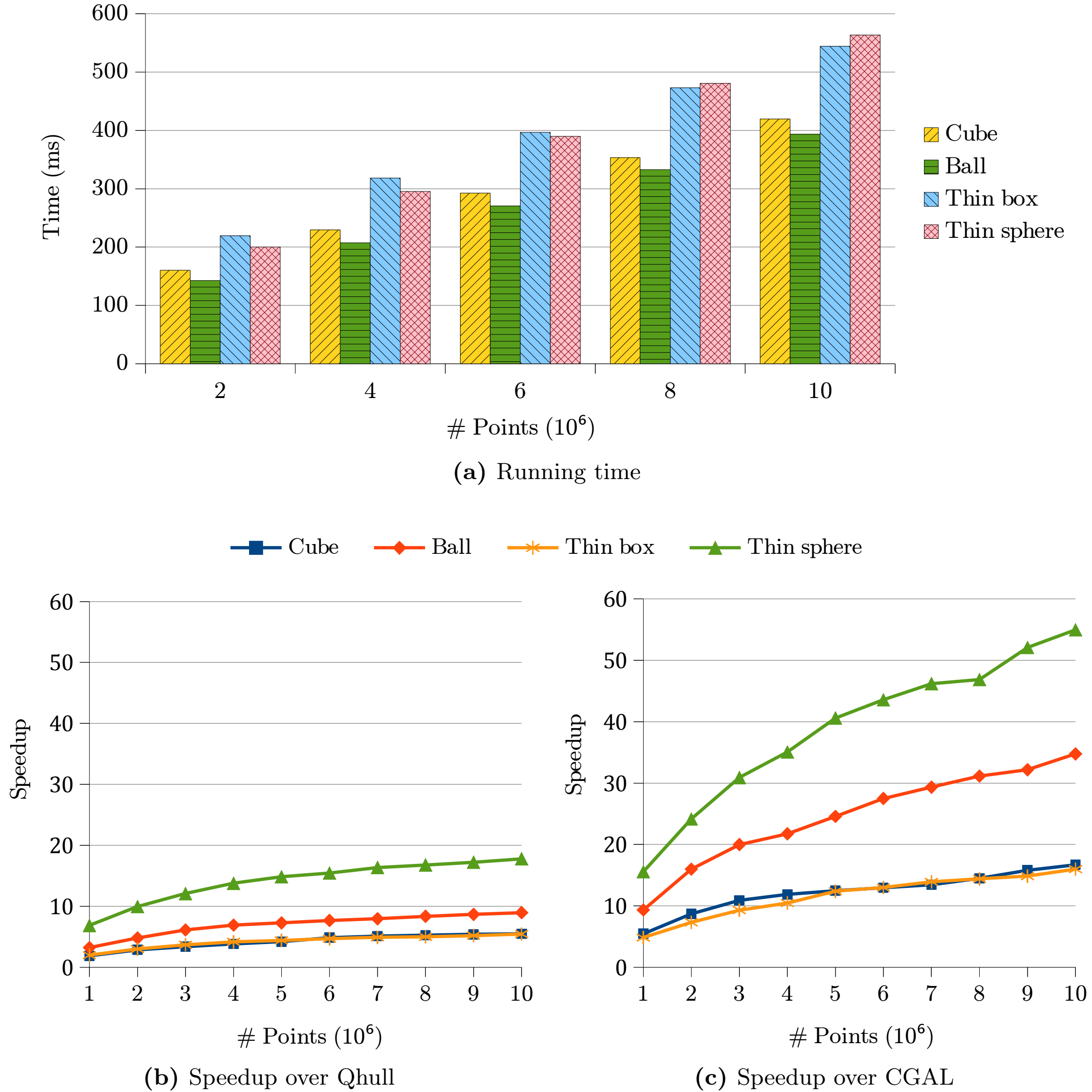
Figure 4.23: The grid size and the rendering buffer size affect the performance of different phases of DigiHull3D.

Synthetic data

We use four point distributions described in Section 2.3 in this experiment: cube, ball, thin box and thin sphere; see Figure 2.8. The cube distribution has very few points on the convex hull, while many points inside can easily be removed by the Quickhull algorithm. The ball distribution is similar, but with a bit more points on the convex hull. The thin box distribution also has very few extreme vertices, but points are distributed close to the convex hull, so it is harder to eliminate them. The thin sphere distribution is a difficult case where many points are on the convex hull while the rest of them are also close to it. These synthetic test cases are highly representative for testing and stressing convex hull algorithms.

We first look at the running time of DigiHull3D while varying the number of points; see Figure 4.24a. Clearly handling the ball and the cube distribution is easier than handling the thin box and the thin sphere distribution, since in the latter two distributions it is more difficult to eliminate the non-extreme vertices. The cube distribution takes a little longer than the ball distribution does since there is degeneracy where coplanar points lie on the boundary.

Figure 4.24b and 4.24c compares the speed of DigiHull3D with that of Qhull and CGAL. In general, DigiHull3D is 3 to 7 times faster than Qhull, and is 7 to 17 times faster than CGAL for the cube and the box distributions. Notably, for the sphere distribution where not only are there many extreme vertices but there are also many points close to the convex hull, DigiHull3D is up to 55 times faster than CGAL and up to 18 times faster than Qhull, even with all the computation being exact. This is mainly because our digital Voronoi diagram gives a very good approximation, and the digital depth test is also very fast in eliminating non-extreme vertices.



(a) Running time

(b) Speedup over Qhull

(c) Speedup over CGAL

Figure 4.24: The running time and speedup of DigiHull3D over Qhull and CGAL on different test cases.

In comparison, the approach of Tang *et al.* [TZTM12] achieves 22 times speedup over CGAL for the ball distribution with 10^7 points, while our speedup is 35 times. Note also that their experiment is done on a similar GPU as ours but with a slower CPU, so CGAL runs slower in their system. According to the authors, their speedup drops when more points are extreme vertices, because there is more work for the CPU processing which uses CGAL. On the other hand, our algorithm achieves an even higher speedup when running on the sphere distribution with many points on or close to the convex hull. Their implementation is not available for us to do a more thorough comparison.

We note that for convex hull computation, a major part of the running time of DigiHull3D is

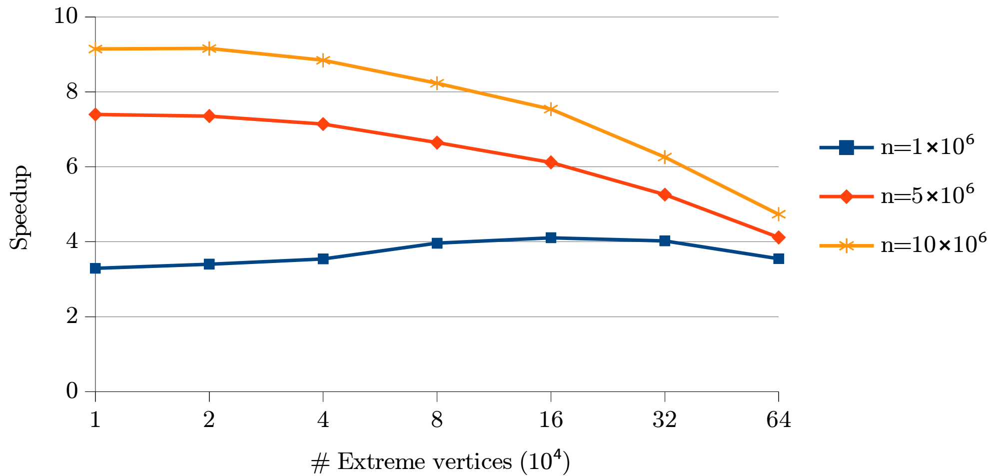


Figure 4.25: The speedup of DigiHull3D over Qhull while fixing the total number of points (n) and varying the number of extreme vertices.

spent on copying the input data from the CPU memory to the GPU memory. This copying sometimes takes more than 40% of the total time. This is because the input point set is very big, while the computation done on the GPU is very fast. For applications that do not require this data transfer (e.g. when data is already in the GPU memory), the performance of DigiHull3D is significantly higher.

Scalability on the number of extreme vertices

In order to investigate the effect of the number of extreme vertices on the performance of the algorithm, we use a slightly modified ball distribution, in which we generate h points on the boundary with the sphere distribution, and $n - h$ points inside with the ball distribution.

Figure 4.25 shows the speedup of DigiHull3D over Qhull when we fix n and vary h multiplicatively from $2^0 \times 10^4$ to $2^6 \times 10^4$. The speedup is higher for larger n, but decreases as h becomes larger. The explanation here is that the digital Voronoi diagram of size 512^2 cannot capture all the extreme vertices. Increasing the grid size might help the approximation, but the overhead is too high. Nevertheless, DigiHull3D is still 3 to 9 times faster than Qhull. A similar behavior can be observed when compared with CGAL, with the speedup being much higher.

Real-world data

We use models of different sizes, ranging from a few hundred thousand to a few million points in this experiment; see Figure 4.26. Points of these models are densely distributed on

Model	# Points	Running time (ms)			Speedup	
		Qhull	CGAL	DigiHull3D	Over Qhull	Over CGAL
Armadillo	172974	42.9	93.1	101.9	**0.4**	**0.9**
Angel	237018	62.3	106.5	95.3	**0.7**	**1.1**
Brain	294012	81.0	142.0	88.3	**0.9**	**1.6**
Dragon	437645	101.0	228.7	107.7	**0.9**	**2.1**
Happy buddha	543652	147.4	352.5	141.4	**1.0**	**2.5**
Blade	882954	186.6	430.9	148.0	**1.3**	**2.9**
Asian dragon	3609600	558.0	994.5	262.2	**2.1**	**3.8**
Thai statue	4999996	714.0	1284.8	407.7	**1.8**	**3.2**

Figure 4.26: The running time of DigiHull3D and its speedup over Qhull and CGAL on different 3D models.

the surface, while their convex hulls have very few vertices (less than 1,000). For models with small number of points our algorithm is slower due to the overhead of the digital computation. Using a smaller grid size and rendering buffer size might help in these cases. Nevertheless, our algorithm still manages to out-perform Qhull and CGAL by up to 2 and 4 times respectively on large models. Note that for computing the convex hull of these real models, a major part of the running time is spent in copying data to and from the GPU memory.

Time breakdown

Figure 4.27 shows the running time of each step of our algorithm on different point distribution with 5×10^6 points. As expected, the behavior differs on different distributions. While the running times of Phase 1a, Phase 1b, and Phase 3a remain almost the same since they are not affected by how the points are distributed, the running times of the other two phases vary significantly. Phase 2 takes more time on the thin box distribution due to many points are wrongly captured and need to be removed by star splaying. On the other hand, Phase 3b takes more time on the sphere distribution due to many points not captured earlier and need to be added later. Phase 1b only takes a small portion of running time for all distributions.

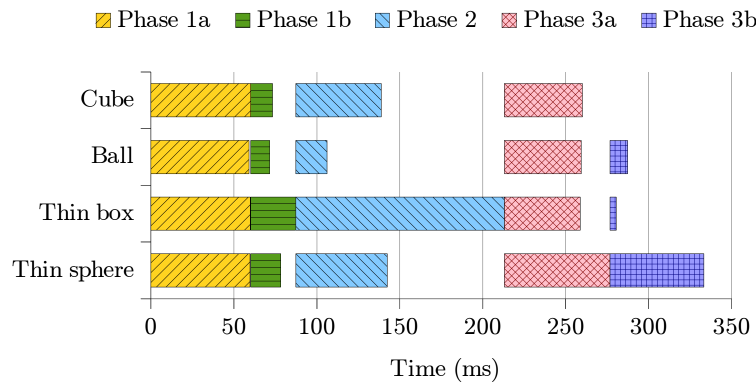


Figure 4.27: The running time of different phases of DigiHull3D.

4.5 Delaunay triangulation in $\mathbb{R}^3$ - The difficulties

Moving forward to adapt the algorithm in Section 4.3 to $\mathbb{R}^3$, there are lots of difficulties. We outline these difficulties in Section 4.5.1, and also show some results that might be of independent interest. In Section 4.5.2 we demonstrate our attempt at adapting the algorithm in Section 4.4 instead to compute the 3D Delaunay triangulation. This algorithm can still robustly construct the 3D Delaunay triangulation on the GPU, but the speedup achieved is limited. Similar to the previous two sections, we map the input point set S into a grid $\mathcal{G}$.

4.5.1 Topological and geometrical difficulties

There are two difficulties when dualizing the 3D digital Voronoi diagrams. The first one is on dualizing the digital Voronoi diagram in $\mathbb{R}^3$, and the second one is on making sure that the dual is a valid triangulation, topologically as well as geometrically.

Dualizing the 3D digital Voronoi diagram

Similar to how we dualize a 2D digital Voronoi diagram, we look at corners shared by multiple grid cells. However, in 3D, since each grid cell is cubical, its corner is shared by up to eight other grid cells with up to eight different colors. It is possible to treat any two colors face-adjacent to each other as connected and dualize that into an edge, but what about those that only share an edge of a grid cell. This problem is similar to triangulating a set of up to eight points. Our aim is to make a consistent interpretation, i.e. a fixed triangulation pattern. Our approach can be visualized as dualizing a *perturbed grid*. The grid cells, which are cubical, are perturbed slightly such that these cells become simple polyhedra; see Figure 4.28. With such a perturbation, only three cells can meet along an edge and only

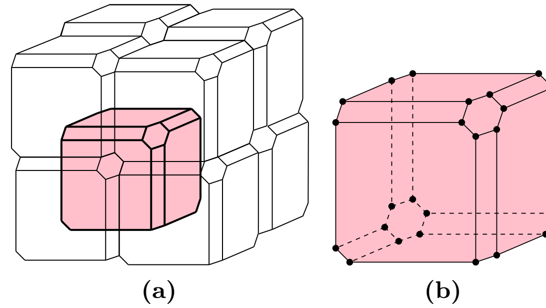


Figure 4.28: (a) A grid of 8 perturbed voxels. (b) A perturbed voxel marked with its 24 corners.

four can meet at a corner to form a digital Voronoi vertex if all four are of different colors. A straightforward implementation of such a perturbation is proposed in [EK12] in which the grid points are moved along the main diagonal: $(i, j, k) \rightarrow (i + \epsilon\Delta, j + \epsilon\Delta, k + \epsilon\Delta)$, where $\epsilon > 0$ is sufficiently small and $\Delta = i + j + k$. We note that this perturbation is merely a convenient method to interpret our grid and does not require any extra computation.

From a digital Voronoi diagram to a valid triangulation

The triangulation dualized directly from the 3D digital Voronoi diagram has similar problems as the one dualized from the 2D digital Voronoi diagram and then more. Not only you can have duplicate or intersecting tetrahedra, but the tetrahedra may form tunnels or voids. The method in Section 4.3.1 and particularly the proof in [CET14] cannot be applied to the 3D case, but we can try to borrow some of the concepts there. A valid triangulation in $\mathbb{R}^3$ has two requirements: it must be homeomorphic to a 3-ball, and there must be no intersections. These are the topological and the geometrical condition of a 3D triangulation, respectively.

Consider a grid $\mathcal{G}$ colored by points in S, each set of connected grid cells with the same color is called a *component*. The set of all components is a subdivision of $\mathcal{G}$. To guarantee that our dualization results in a topologically valid triangulation, following the Nerve theorem [Bor48], the coloring must meet the following conditions:

1. Every component is homeomorphic to a 3-ball.
2. Every intersection of 2 components is homeomorphic to a disk.
3. Every intersection of 3 components is homeomorphic to a line segment.
4. Every intersection of 4 components is homeomorphic to a point.

Furthermore, it is also possible to guarantee that the dualization results in a geometrically valid 3D triangulation, i.e. there is no self-intersections. To do that, we must make sure

that the boundary of the dualized triangulation is a star-shaped polyhedron, and all the tetrahedra obtained are positively orientated, as proven in the following lemma.

Lemma 4.11. *Consider a 3D embedding of a topologically valid triangulation. If the boundary of the embedding is homeomorphic to a 2-sphere and is star-shaped, and all the tetrahedra are positively orientated, then the embedding is a geometrically valid triangulation.*

Proof. Consider the topologically valid triangulation $\mathcal{T}$ in $\mathbb{R}^4$, since the boundary of $\mathcal{T}$ is a star-shaped polyhedron, we can extend all the facets on the boundary to a point k_1 in $\mathbb{R}^4$, forming a 3-sphere $\mathcal{S}_1$. Similarly, we extend the facets of the convex hull of the points in $\mathcal{T}$ to a point k_2 in $\mathbb{R}^4$ to form a 3-sphere $\mathcal{S}_2$, such that $\mathcal{S}_1$ lies completely inside $\mathcal{S}_2$.

We define a continuous map $f : \mathcal{S}_1 \mapsto \mathcal{S}_2$ as follows. Pick a point k inside $\mathcal{S}_1$ as a kernel. The continuous map f maps any point $x \in \mathcal{S}_1$ to a point $y \in \mathcal{S}_2$ where y is the intersection of the ray $\overrightarrow{kx}$ with $\mathcal{S}_2$. Clearly, f maps a point in $\mathcal{T}$ into a point in the convex hull of $\mathcal{T}$, at the same position.

The degree of f is the multiplicity of the image of $\mathcal{S}_2$, i.e. the number of time $\mathcal{S}_1$ maps to $\mathcal{S}_2$, taking into account whether the map preserves the orientation or not. Take any point y in the underlying space of $\mathcal{T}$ (assuming it is not on any triangle, edge or vertex), the degree of f at y is the number of tetrahedra in $\mathcal{T}$ that contain y, counting a tetrahedron positive if it has positive orientation and negative if it has negative orientation. The main point here is that the degree is the same at every point of $\mathcal{S}_2$; see, for example, Chapter 3 of [GP10].

To determine the degree of the map f, we take a point y outside the convex hull of $\mathcal{T}$. Since the boundary of $\mathcal{T}$ is star-shaped, the line ky intersects $\mathcal{S}_1$ at exactly one point, which is on a tetrahedron formed by a facet on the boundary of $\mathcal{T}$ and k_1. As such, the degree of the map f is 1. For any point y' inside the embedding of $\mathcal{T}$, since there is no negatively oriented tetrahedra, y' must be contained by exactly one tetrahedron. This implies that the embedding of $\mathcal{T}$ has no self-intersections. □

It is very costly, and sometimes impossible, to obtain a coloring that satisfies all the conditions mentioned above due to a lot of topological and geometrical constraints. As such, we cannot extend the approach in Section 4.3 to compute the 3D Delaunay triangulation.

4.5.2 Algorithm using star splaying

In this section, we sketch a different algorithm, following an approach similar to that in Section 4.4, to compute the 3D Delaunay triangulation. Our algorithm again makes use of the star splaying algorithm [She05], this time in $\mathbb{R}^4$. The sketch obtained from the digital Voronoi diagram is lifted into $\mathbb{R}^4$ and transformed into a 4D convex hull using the star splaying algorithm. The result projected into $\mathbb{R}^3$ is the 3D Delaunay triangulation. The algorithm consists of the following phases:

Phase 1a: Voronoi Construction.
Construct the digital Voronoi diagram $\mathcal{V}_D(S)$.

Phase 1b: Working Set Identification.
Derive, for each point $s \in S$, a working set of s that includes points of $S \setminus \{s\}$ that are nearby s as indicated in $\mathcal{V}_D(S)$.

Phase 2: Star Creation.
Create a convex star for each point $s \in S$ by inserting points from its working set one by one using the beneath-beyond algorithm.

Phase 3: Star Splaying.
Splay the stars until they are all consistent with one another.

Phase 1a is done similar to Phase 1a in the 2D Delaunay triangulation algorithm. In Phase 1b, we use the perturbed grid interpretation to identify the neighbors of each point $s \in S$. To do so, we first identify the corners incident to four neighbors with different colors. Each of this gives us a tetrahedron on which any two vertices are neighbors. The set of neighbors is the *working set* of s.

Each point needs at least 4 points in its working set to form a star. If any working set is smaller than this number, which sometimes happens, then we augment it the working set of the point's neighbors, if available. We also create the working sets for the missing points from Phase 1a. Each missing point gets the same working set as that of the input point which occupies the grid point that it maps to.

The star creation and the star splaying phase are done similar to Phase 1b of the 3D convex hull algorithm, except it is a 4D star splaying rather than a 3D one. One interesting implementation note is the data-structure used during the star splaying. Each star is allocated a continuous chunk of memory inside a huge array to store its link triangulation. It is very easy for a star to exceed its storage during splaying, and that might happen many times for some very bad stars. Expanding the storage for a star is costly since we have to reallocate the whole star list. Instead, we note that given the good quality of the digital approximation, the size of the star after Phase 2 is mostly correct. We split the storage of a star into two parts, i.e. the star list is stored in two separate arrays. The first storage space of a star is allocated during Phase 2, while the second one can be allocated and resized during Phase 3. This way, the first array is fixed, and contains the majority of information. The second array is dynamic, and can be reallocated frequently, but its size is usually very small so reallocating it is not costly.

4.5.3 Experiment

In this section, we first discuss about the effect of the grid size on the performance of our implementation, termed DigiDel3D, overall as well as of individual phases. After that we compare our performance with that of CGAL, the fastest CPU 3D Delaunay triangulation software available.

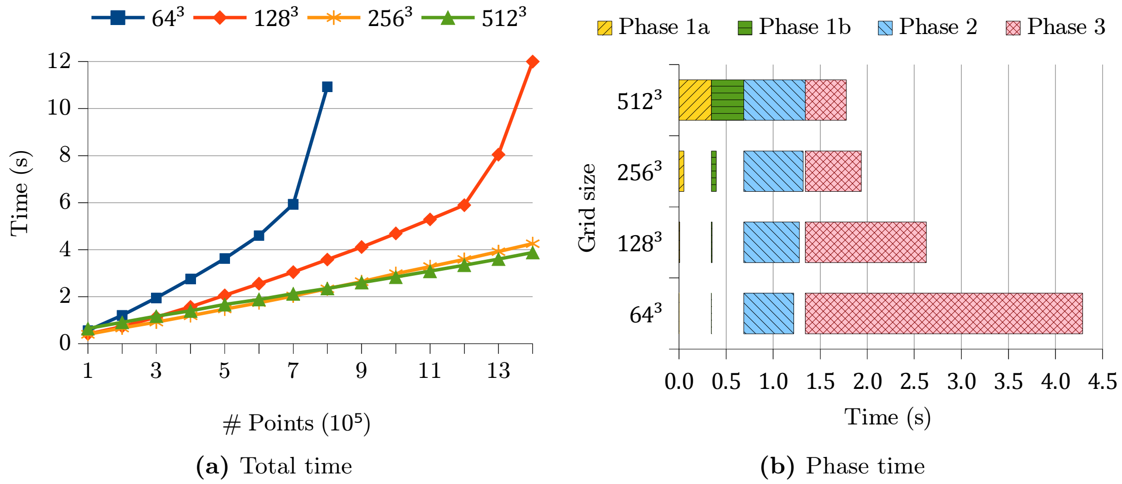
Grid sizes and time breakdown

Similar to the 2D case, the performance of DigiDel3D is strongly dependent on the quality of the approximation derived from the digital computation. Hence, grid size is a very important parameter. Figure 4.29a shows the running time of DigiDel3D using different grid size, on varying number of points ranging from 10^5 to 14×10^5 uniformly distributed. It is clear that using a small grid such as 64^3 is almost impossible for these numbers of input points. The approximation is not good, and many points are missing and thus their stars are only approximated using their neighbor's stars. As such, the star splaying in Phase 3 becomes expensive. Note that star splaying is only efficient when the stars are close to be consistent, as discussed in Shewchuk's paper [She05]. Using a 128^3 grid gives a more reasonable speedup, and the performance is much better when using 256^3 or 512^3 grids.

Figure 4.29b gives a clearer picture of the effect of grid size on the performance of different phases of the algorithm when handling 5×10^5 points uniformly distributed. The performance of Phase 3 is significantly improved, up to 5 times, when moving from the 64^3 grid to the 512^3 grid. The running time of Phase 1a and Phase 1b increases due to having to process a larger grid, but overall both phases take less time than Phase 3. It is only when moving from 256^3 to 512^3 grid that the overhead of handling a larger grid outweighs the improvement in Phase 3. As the number of points increases and hence the work in Phase 3 increases, a larger grid would give even more benefit. Besides, a large grid is also needed to handle points that are non-uniformly distributed.

Synthetic data

We compare the performance of DigiDel3D with CGAL on different point sets from the uniform and the Gaussian distribution, with sizes ranging from 10^5 to 14×10^5; see Figure 4.30. On the uniform distribution, using a 512^3 grid, DigiDel3D manages to outperform CGAL by up to 3.5 times. For point set of size less than 8×10^5, using a 256^3 grid is slightly better, as mentioned earlier. With the Gaussian distribution, clearly we need a 512^3 grid, and even so the speedup of DigiDel3D is still slightly lower. We also observe that the speedup still increases with larger input for the uniform distribution, but with the Gaussian distribution, the speedup starts to drop with the increase in input size. This is because more points are concentrated in the center of the grid, and thus the number of missing points increases very



(a) Total time

(b) Phase time

Figure 4.29: The total running time and time breakdown of DigiDel3D on a uniform distribution, using different grid sizes.

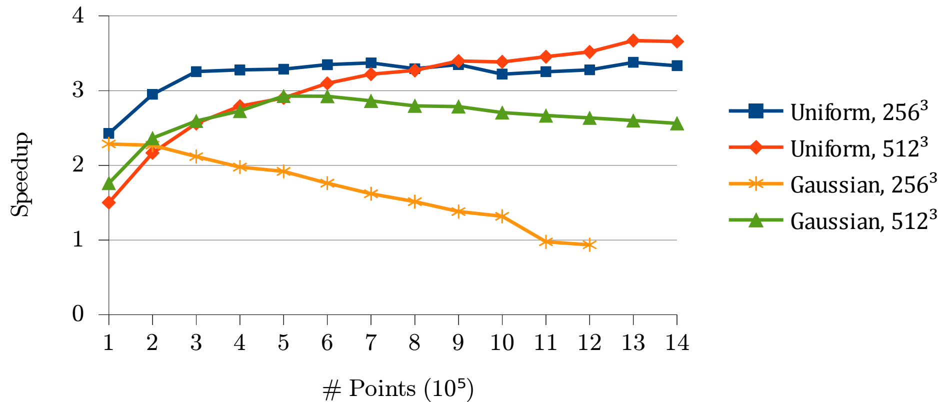


Figure 4.30: The speedup of DigiDel3D compared to CGAL.

fast with the increase in the input size. This being sensitive to input distribution is one of the main disadvantages of using the digital space, and it has now become more obvious in higher dimensions. We have also tested DigiDel3D on a more extreme case, the thin sphere distribution, as described in Section 2.3. DigiDel3D can handle these cases robustly, but the performance is only slightly better than CGAL on small input size, and lower when there are more than 5×10^5 points.

Real-world data

Real-world data poses a greater challenge to DigiDel3D. We run some experiments with the 3D models mentioned in Section 2.3. In these datasets, points are not uniformly distributed,

Model	# Points	Running time (s)			Speedup	
		CGAL	DigiDel3D			
			256^3	512^3	256^3	512^3
Armadillo	172974	1.7	2.2	1.9	**0.7**	**0.9**
Angel	237018	2.4	8.2	5.3	**0.3**	**0.5**
Brain	294012	3.0	3.6	2.7	**0.8**	**1.1**
Dragon	437645	4.7	11.5	6.8	**0.4**	**0.7**
Happy Buddha	543652	5.6	-	11.3	**-**	**0.5**
Blade	882954	9.5	-	31.5	**-**	**0.3**

Figure 4.31: The running time comparison between DigiDel3D and CGAL on some real 3D models.

and there are also a lot of degeneracies. Figure 4.31 shows the running time of both CGAL and DigiDel3D on these models. Using 256^3 grid size, DigiDel3D runs out of memory when processing some large test cases, since the star splaying uses more memory due to the approximation being of low quality. Using 512^3 grid, out of the 6 models tested, only in the Brain model does DigiDel3D manage to outperform CGAL. Nonetheless, our implementation can handle these test cases robustly.

4.6 Discussion

In this chapter, we have introduced a novel approach to solve several fundamental computational geometry problems. Taking advantage of the connection between the fundamental geometric structures and the Voronoi diagram to derive an approximation in digital space, we develop efficient algorithms to construct the convex hull and the Delaunay triangulation in $\mathbb{R}^2$ and $\mathbb{R}^3$ robustly on the GPU, with provable correctness. There are two main advantages of our approach. First, a major part of the computation is done through the construction of the digital Voronoi diagram, which can be performed very quickly and efficiently on the GPU. Our digital Voronoi diagram algorithm is work efficient, with very high parallelism and also has good memory access pattern, thus it can exploit the enormous computation power of the GPU. Second, the approximation from the digital space computation also acts as a very good bootstrapping phase, providing enough input for the data-parallel processing in subsequent phases.

These are not to say that this approach comes with no disadvantage. The most significant shortcoming is also from using the digital computation to approximate continuous structures. As shown in some of the experiment, the approximation is affected by the point distribution in the input. This is not very apparent in 2D, but starts to show up in 3D, especially in the 3D Delaunay triangulation computation. The second shortcoming is from the attempt to dualize the digital Voronoi diagram. There are many possible topological difficulties, which we solve nicely for the 2D Delaunay triangulation case, but not completely for the 3D case and higher dimensions. More research is needed in this area to push the applicability of this approach further.

The work in this chapter has received helps and collaboration from several people. Tang Ke ran some testing on the earlier version of the PBA algorithm, including several approximate ones, and provided valuable feedback. Mohamed Anis provided the preliminary implementation of Schneider *et al.*'s algorithm on CUDA. The DigiDel2D work was developed from an earlier collaboration with Rong Guodong and Stephanus. The convex hull algorithm described in Section 4.4 was done in collaboration with Gao Mingcen, and the 4D star splaying in Section 4.5 was implemented on the GPU together with Ashwin Nanjappa. The validity of the triangulation dualized from the digital Voronoi diagram in both $\mathbb{R}^2$ and $\mathbb{R}^3$ was proven with the help of Professor Herbert Edelsbrunner.

Chapter 5

Incremental Insertion with Local Transformation

5.1 Overview

Algorithm 5.1: Traditional incremental insertion paradigm.

```
  Data: A point set S.
  Result: The resulting structure T
1 T ← a simple initial structure
2 Q ← S \ {p | p ∈T}
3 while ¬ Empty(Q) do
4 |   p ← ExtractMin(Q)
5 |   locate a position of p in T
6 |   insert p into T
7 |   apply a series of local transformation around p
8 end
9 return T
```

In this chapter, we aim to address the most prominent shortcoming of the digital approach proposed in the previous chapter, which is the inability to handle inputs with points not "nicely" distributed. We propose a new approach to construct the convex hull and the Delaunay triangulation, with the goal of achieving good speedup even on real-world datasets. Particularly, we focus on the algorithm to construct the Delaunay triangulation in $\mathbb{R}^3$, since the digital approach has a lot of difficulty handling this problem.

Our new approach is motivated by the traditional incremental insertion approach widely used to solve computational geometry problems on the CPU. This traditional approach generally follows the framework in Algorithm 5.1. Here Q is a priority queue to order the insertions in some way. Starting from an initial structure, the algorithm repeatedly inserts more points. Each insertion consists of three basic steps: locate the position, insert, and apply local transformation to restore the properties of the structure.

Algorithm 5.2: Parallel incremental insertion with local transformation approach.

Data: A point set S.
Result: The resulting structure $\mathcal{T}$

```
1  T ← a simple initial structure
2  Q ← S \ {p | p ∈T}
3  while ¬ Empty(Q) do
4  |   evaluate and choose points to be inserted in parallel
5  |   insert the chosen points into T
6  |   remove the inserted points from Q
7  |   update the location of points in Q
8  end
9  repeat
10 |   evaluate and choose the local places to transform in parallel
11 |   apply the local transformations
12 until no more transformation is possible
13 return T
```

Most existing multi-core computational geometry algorithms rely on this traditional approach. The parallelism is achieved by performing multiple insertions (line 4–7) in parallel. This, however, requires locking since two insertions may try to modify the same part of the structure, leading to a race condition. Moreover, to guarantee correctness of the local transformation in line 7, if two insertions affect each other (i.e. their affected regions overlap), then only one can continue while the other needs to roll back and try again later. Such an approach is not applicable to the GPU for many reasons. First, locking is costly if not unfeasible on the GPU. Second, the work of each thread is possibly unbalanced. And most importantly with a huge number of threads on the GPU, the chance that the insertions conflict is very high, leading to a lot of rolling back, wasting the computation.

Our approach is different. We parallelize the steps inside the while-loop separately, making it more suitable for the GPU while being able to be executed in a lock-free manner; see Algorithm 5.2 for an overview. There are two main phases in our approach: the insertion phase and the transformation phase. In the insertion phase, points are quickly inserted in parallel in batches to construct an initial structure. The points are chosen to be inserted in some order to better approximate the final structure. After that, local transformations are applied in parallel, also in multiple batches, until we get the desired result.

There are several challenges when applying this approach. First, the order of point insertion

can affect the performance significantly. And second, it is not always possible to apply the same local transformation algorithm as the sequential algorithm and get the correct result. The details will be clearer in the next sections when we apply this approach to the three problems discussed in the previous chapter. The three algorithms proposed in this chapter are termed IncDel2D, IncHull3D and IncDel3D for the 2D Delaunay triangulation, 3D convex hull, and 3D Delaunay triangulation problem respectively ("Inc" for incremental insertion).

5.2 3D convex hull revisit

The idea of the algorithm in this section is to use parallel insertion to construct a polyhedron from points in S, followed by using flipping to transform it into the convex hull of S. In Section 2.1.5, we have defined the flipping operation on a triangulation. The convex hull is slightly different from the triangulation, so we need to adjust the definition of the flips. The topological structure of a flip remains the same, but the geometrical condition of a flip is redefined. We make sure the flips do not create self-intersection by starting from a star-shaped polyhedron and only allow flips that do not break this star-shaped property.

Definition 5.1. Given a point $s \in \mathbb{R}^3$ and three points $a, b, c \in S$, the *cone* of triangle abc w.r.t. s, denoted as $\mathcal{C}_s(\triangle abc)$ is the convex hull of the three rays $\overrightarrow{sa}$, $\overrightarrow{sb}$, and $\overrightarrow{sc}$.

Two cones *overlap* if they contain some common points not on their boundaries. Let $\mathcal{T}$ be a polyhedron with vertices in S and with faces that are triangles. We say $\mathcal{T}$ is a *star-shaped* polyhedron w.r.t. s if s is in the interior of $\mathcal{T}$ and the cones of any two triangles of $\mathcal{T}$ w.r.t. s do not overlap each other. This means the boundary of a star-shaped polyhedron is entirely visible from s. The point s is called the *kernel* of $\mathcal{T}$.

Let $\mathcal{T}$ be a *star-shaped* polyhedron w.r.t a kernel point s in $\mathbb{R}^3$. Let $e = \overline{ab}$ be an edge in $\mathcal{T}$ with two link points c, d. The edge e is a *reflex edge* (w.r.t. s) if c and s lie on different sides of $\triangle bad$; otherwise it is a *convex edge*. We say that e is *flippable* w.r.t. s if firstly s is outside the tetrahedron $abcd$, and secondly the union of the cones of the triangles of σ_e (the induced subcomplex of e) is equal to $\mathcal{C}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$. Otherwise, e is *unflippable*.

Intuitively, an edge is flippable if after flipping it, the polyhedron is still star-shaped w.r.t. s. This star-shaped condition guarantees that flipping does not create self-intersections in the polyhedron. As mentioned earlier, our convex hull algorithm consists of two phases:

Phase 1: Construction.
Use incremental insertion to construct a star-shaped polyhedron with the points in $S' \subset S$ such that any other point in S is inside the polyhedron.

Phase 2: Flipping.
Use flipping to transform the constructed polyhedron into the convex hull of S.

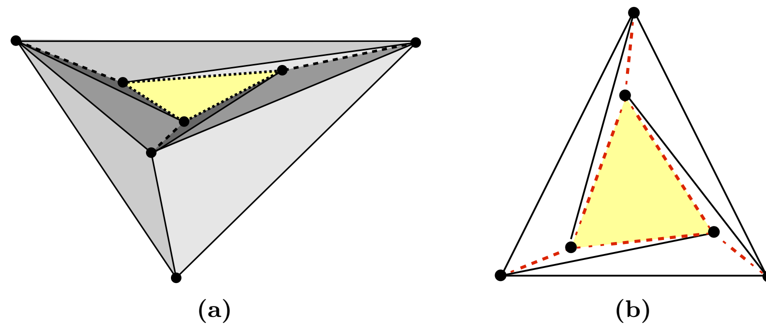


Figure 5.1: (a) A stuck configuration in 3D when flipping a star-shaped polyhedron. The dashed edges are reflex but are all unflippable. (b) The top-down view.

In Phase 2, it is tempting to use an algorithm similar to Phase 3b in Section 4.3.5, flipping in parallel all reflex and flippable edges until no more flips can be done. This, however, does not lead to the convex hull in some cases, such as the one in Figure 5.1. In this example, the three vertices of the small triangle in the center are pushed down slightly so that all dashed edges are reflex but are not flippable.

Instead, we propose a slightly modified flipping algorithm to avoid getting stuck during flipping. The detail is discussed below.

5.2.1 Phase 1: Construction

Algorithm 5.3 constructs a star-shaped polyhedron of S using multiple rounds of point insertion; see Figure 5.2. It follows the structure of the first phase outlined earlier in Algorithm 5.2. First, the polyhedron $\mathcal{T}$ is initialized by four points of S, and the kernel is the centroid of this polyhedron. Each point in S is then assigned to one triangle in $\mathcal{T}$ if it falls inside the corresponding cone (line 4–5). The details of how to choose the four initial points is discussed in Section 6.4.

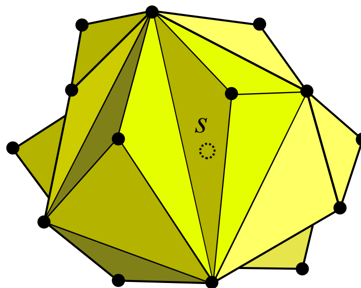


Figure 5.2: A result of the Construction phase.

Algorithm 5.3: Constructing a star-shaped polyhedron.

Data: A point set S in $\mathbb{R}^3$
Result: $\mathcal{C}(S)$

```
1  initialize 𝒯 with 4 points a, b, c, d in S
2  s ← the centroid of 𝒯
3  S ← S \ {a, b, c, d}
4  for each p ∈ S do in parallel
5  |  location[p] ← △t s.t. t ∈ 𝒯 and p ∈ 𝒞_s(t)
6  while ¬Empty(S) do
7  |  for each p ∈ S do in parallel
8  |  |  AtomicMaximum(minDist[location[p]], Distance(p, location[p]))
9  |  for each p ∈ S do in parallel
10 |  |  if minDist[location[p]] = Distance(p, location[p]) then
11 |  |  |  insert[location[p]] ← p
12 |  for each △t ∈ 𝒯 do in parallel
13 |  |  p ← insert[t]
14 |  |  if p ≠ null then
15 |  |  |  split △t into three triangles using p
16 |  for each p ∈ S do in parallel
17 |  |  update location[p] if it was split
18 |  |  if p and s lie on the same side of location[p] then S ← S \ p
19 end
```

The main loop of the algorithm inserts the rest of points of S into $\mathcal{T}$ while removing points that are completely inside $\mathcal{T}$, i.e. not extreme vertices. The process is done in multiple iterations, each one inserts a batch of points, at most one per triangle. First, the GPU kernel in line 7–8 finds for each triangle the point in its cone that is farthest. This is done using the AtomicMaximum function. Then, the GPU kernel in line 9–11 finds the corresponding winner and the kernel in line 12–15 inserts it into the corresponding triangle by splitting that triangle into three. Lastly, the kernel in line 16–18 updates the location of the points that are left in S, while removing those clearly in the interior of $\mathcal{T}$.

The purpose of finding the farthest point for each triangle to insert is motivated by the Quickhull algorithm [BDH96]. It allows us to quickly eliminate most non-extreme vertices.

5.2.2 Phase 2: Flipping

We propose a modified flipping algorithm to avoid getting stuck while flipping. The idea comes from the observation that whenever we have a reflex edge that is unflippable, we can identify a non-extreme vertex.

Lemma 5.1. *Any 2–2 unflippable edge* $e = \overline{ab}$ *that is reflex is incident to a non-extreme vertex. That is, either* a *or* b *is non-extreme* [GCTH13].

On the other hand, if all vertices on the polyhedron are extreme vertices, then flipping always works.

Lemma 5.2. *If all vertices of* $\mathcal{T}$ *are extreme vertices, then any reflex edge* e *of* $\mathcal{T}$ *is 2–2 flippable, and thus* $\mathcal{T}$ *can always be transformed until it has no more reflex edge. After that,* $\mathcal{T}$ *is the convex hull* [GCTH13].

Clearly, since we are constructing the convex hull, any non-extreme vertex should be removed by a 3–1 flip. However, if the degree of that vertex is greater than 3, then we cannot perform a 3–1 flip. Fortunately since we know that this vertex should be removed, we can apply flipping on both the reflex and the convex edges incident to the vertex to decrease its degree until we can perform a 3–1 flip to remove it.

The new flipping algorithm, termed *Flip-Flop* is sketched in Algorithm 5.4. In the description of Algorithm 5.4, we skip the details on how to avoid race condition and how to update the triangulation while flipping, since these have been described in Section 4.3.5. Instead, we focus on the conditions to perform a flip. There are three cases in which we perform the flip of an edge. In line 6, we flip e if it is a 3–1 flip and flipping it removes a non-extreme point (either already labeled or e is reflex). In line 13, we flip e if it is reflex and none of the non-extreme vertices we labeled is involved. In line 14, we flip e if doing so reduces the degree of the non-extreme vertex with smallest index involved in the flip, regardless of whether it is reflex or convex. Note that after each flip, we label all the modified edges to be checked again next round. A vertex is labeled as non-extreme when we can identify it through a reflex and unflippable edge (line 8–10).

The motivation behinds the three flipping cases is clear. For case 1, we can remove a non-extreme vertex, and that moves us closer to the final result so we always do it. For case 2, if the flip does not involve any non-extreme vertex, then we do it when the edge is reflex, similar to the traditional flipping algorithm. Case 3 is a little bit tricky; there are non-extreme vertices involved and thus we only flip if doing so reduces the degree of the non-extreme vertex with smallest index. This is so that the algorithm does not flip an edge back and forth.

Algorithm 5.4: The Flip-Flop algorithm.

```
   Data: A star-shaped polyhedron T
   Result: C(T)
1  label all vertices of T as extreme
2  label all edges as to be checked
3  while there are edges to be check do
4      for each edge e that needs to be checked do in parallel
5          if e is a 3–1 flip then
6              if e is reflex or flipping e removes a non-extreme vertex then flip e
7          else
8              if e is reflex and unflippable then
9                  label a vertex as non-extreme
10                 label all edges on the star of that vertex as to be checked
11             if e is flippable then
12                 x ← the non-extreme vertex with smallest index involved in the flip
13                 if x = null and e is reflex then flip e
14                 if x ≠ null and x ∈ e then flip e
15     end
16 end
```

5.2.3 Proof of correctness

The correctness of our flipping algorithm is guaranteed with the following lemmas.

Lemma 5.3. *If the degree of a non-extreme vertex v is 3, then any edge incident to it is 3–1 flippable* [GCTH13].

Lemma 5.4. *If the degree of a non-extreme vertex v is more than 3, then there exists a 2–2 flippable edge incident to v* [GCTH13].

The Flip-Flop algorithm definitely terminates, since each flip performed either increases the volume of $\mathcal{T}$ or decreases the degree of a non-extreme vertex without increasing the degree of any other non-extreme vertex with smaller index. The algorithm flips all flippable edges, and if there is any unflippable edge, then we can identify a non-extreme vertex; otherwise it contradicts Lemma 5.2. Then, by Lemma 5.3 and Lemma 5.4, we can flip and remove the non-extreme vertices. As such, when the algorithm terminates, the result is the convex hull.

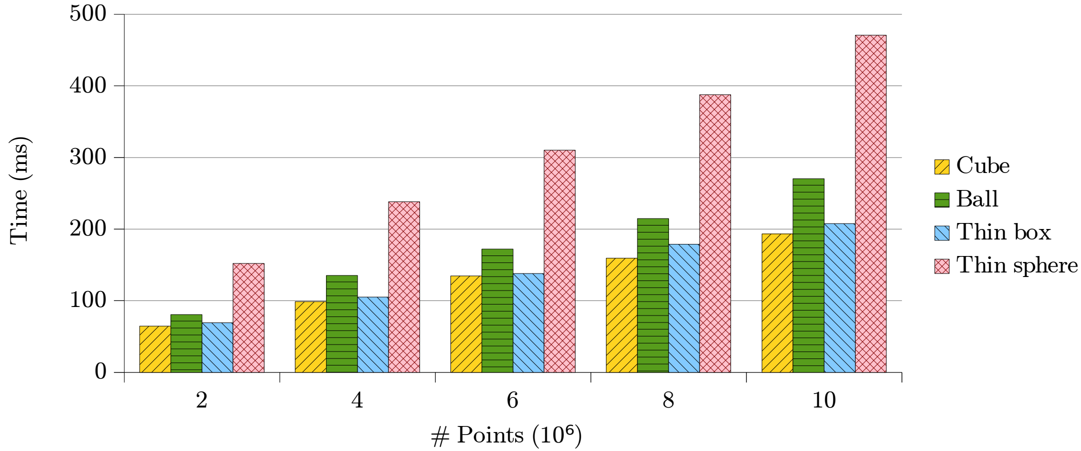


Figure 5.3: The running time of IncHull3D on different test cases.

5.2.4 Experiment

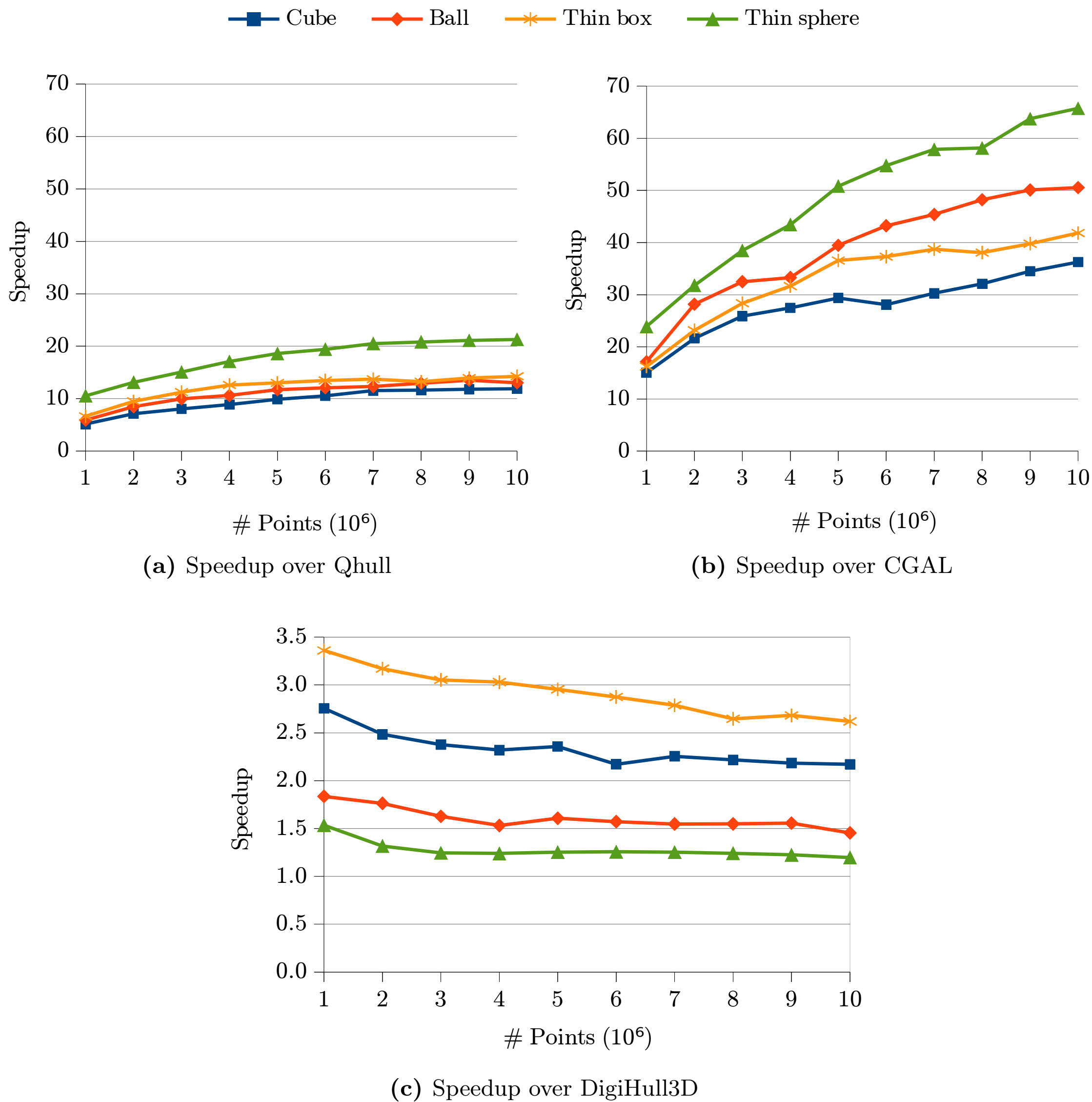
We compare the performance of the implementations of our algorithm, termed IncHull3D, on the GPU with the two fastest convex hull implementations on the CPU, Qhull and CGAL, as well as with DigiHull3D, the algorithm proposed in the previous chapter.

Synthetic data

Similar to Section 4.4, we use four point distributions: cube, ball, thin box and thin sphere as the synthetic data to test our convex hull algorithm. First, we show the running time of IncHull3D on our synthetic data in Figure 5.3. This time, different from DigiHull3D which is more sensitive to how the points are distributed, IncHull3D is only sensitive to the number of extreme vertices. The cube and the thin box distributions both have very few vertices on the convex hull, thus the running time of them is similar. The ball distribution has a few more extreme vertices, so it takes a little longer to process. The thin sphere distribution has the most number of extreme vertices, and thus IncHull3D needs significantly more time to finish the computation.

Figure 5.4 compares the running time of IncHull3D with Qhull, CGAL and DigiHull3D. Our GPU implementation achieves up to 20 times speedup compared to Qhull, and 65 times speedup compared to CGAL. Interestingly, the speedup even increases when there are more extreme vertices. Also, even for small point set and a simple point distribution such as the cube or the ball distribution, IncHull3D still manages to outperform CGAL by more than 15 times.

IncHull3D also performs better than the digital solution DigiHull3D, especially on the cube and the thin box point distributions; see Figure 5.4c. As mentioned in the previous chapter,



(a) Speedup over Qhull

(b) Speedup over CGAL

(c) Speedup over DigiHull3D

Figure 5.4: The speedup of IncHull3D over Qhull, CGAL, and DigiHull3D.

DigiHull3D does not perform well in the cube and the box distributions due to the use of approximation in the digital space. IncHull3D, however, does not have this disadvantage, and thus it is two to three times faster. For the thin sphere distribution, however, IncHull3D is only slightly faster than DigiHull3D, probably due to different implementations.

Scalability on the number of extreme vertices

Similar to the experiment with DigiHull3D, we use the modified ball distribution to test the performance of IncHull3D with varying number of extreme vertices. Figure 5.5 shows the performance comparison against Qhull. We also include the result of DigiHull3D for

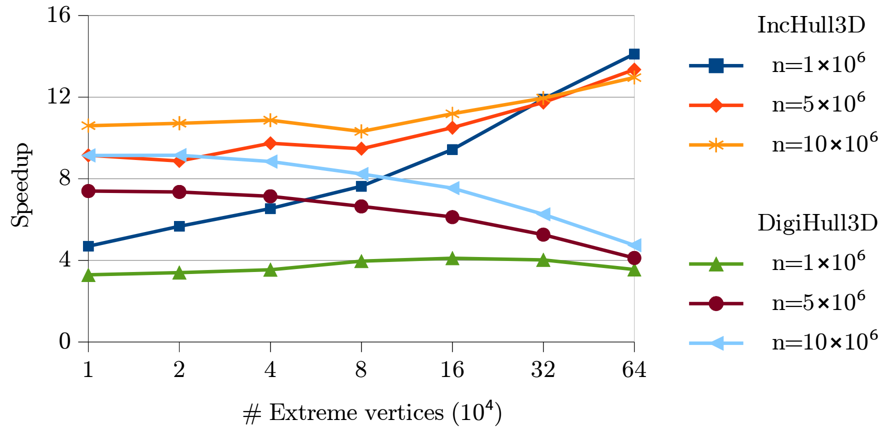


Figure 5.5: The speedup of IncHull3D over Qhull when fixing the number of points and varying the number of extreme vertices, compared with that of DigiHull3D.

comparison. Clearly, with our novel flipping algorithm, IncHull3D handles inputs with many extreme vertices very well. The speedup over Qhull increases sharply when there are more points on the convex hull. This is opposite to DigiHull3D which actually slows down when there are more extreme vertices. As such when the number of extreme vertices is high, IncHull3D easily outperforms DigiHull3D. It is only when the number of extreme vertices is small that DigiHull3D performs close to IncHull3D.

Real-world data

We use the same set of models to test IncHull3D; see Figure 5.6. When the number of input points is small, the cost of copying data to and from the GPU memory dominates the total running time, and thus IncHull3D is only slightly faster than CGAL, while slightly slower than Qhull. This is still better than DigiHull3D since in the latter, there is also the overhead of the digital Voronoi diagram computation. When the number of points is larger, IncHull3D is up to 5 times faster than Qhull, and up to 8 times faster than CGAL. It is also around 2 times faster than DigiHull3D. This shows that IncHull3D is better at handling non-uniform point distribution compared to DigiHull3D, which is our goal when designing the second approach in this chapter.

Time breakdown

In Figure 5.7, we compare the time spent on the two phases of IncHull3D, the Construction phase (Phase 1) and the Flipping phase (Phase 2), on the four synthetic point distribution with 5×10^6 points. As more points are on the convex hull, both stages of the algorithm take more time. Phase 1 needs more time to insert points, since fewer points that are non-extreme

Model	# Points	Running time of IncHull3D (ms)	Speedup over		
			Qhull	CGAL	DigiHull3D
Armadillo	172974	69.9	0.6	1.3	1.5
Angel	237018	80.2	0.8	1.3	1.2
Brain	294012	74.4	1.1	1.9	1.2
Dragon	437645	88.0	1.1	2.6	1.2
Happy buddha	543652	98.6	1.5	3.6	1.4
Blade	882954	88.8	2.1	4.9	1.7
Asian dragon	3609600	162.4	3.4	6.1	1.6
Thai statue	4999996	150.9	4.7	8.5	2.7

Figure 5.6: The running time of IncHull3D and its speedup over Qhull, CGAL and DigiHull3D on different 3D models.

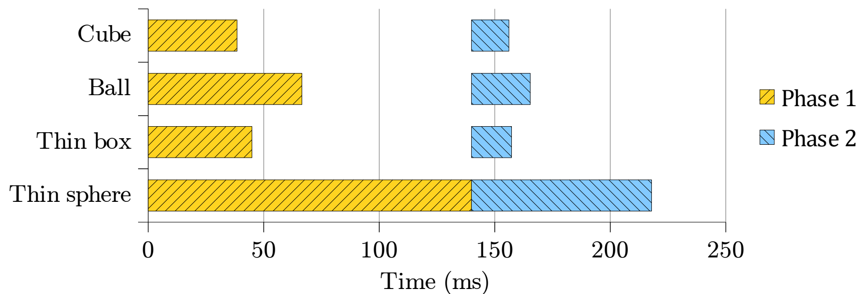


Figure 5.7: The time spent on different phases of IncHull3D.

vertices can be removed so the actual number of points that are inserted is higher. Phase 2 needs more time to flip the constructed polyhedron to the convex hull. The ratio between the times spent in the two phases also changes. When the number of extreme vertices is small, Phase 2 takes much less time than Phase 1, while in the thin sphere distribution when there are more extreme vertices, Phase 2 takes a larger portion of the total time. The trend is that when there are even more extreme vertices, Phase 2 would dominate the running time of IncHull3D. This is because the number of points on the convex hull affects the first phase at a logarithmic rate (i.e. only affects the number of insertion iterations) while it affects the second phase at a linear rate in our experiment.

5.3 Delaunay triangulation in $\mathbb{R}^2$ and $\mathbb{R}^3$ with adaptive star-splaying

Following the proposed approach, we aim to construct the Delaunay triangulation in a similar manner as when we construct the convex hull in the previous section. Parallel point insertion is used to construct an initial triangulation, and parallel flipping is used to transform it into the Delaunay triangulation. This method works correctly in $\mathbb{R}^2$, since any triangulation can be flipped into the Delaunay triangulation, regardless of the flip order. However, in $\mathbb{R}^3$ the flipping can get stuck, as shown by Joe [Joe89]. When there are no more flippable facets, we might still have some non-locally Delaunay ones. The Flip-Flop algorithm does not work in this case, since when constructing the Delaunay triangulation, none of the point should be removed.

We observe that in practice, after the flipping, only a small number of facets might still be non-locally Delaunay. Also, according to the work by Joe [Joe89], these unflippable facets form cycles, called *NLONT-configuration* (non-locally optimal nontransformable). In other words, they cluster together, and that is why they are stuck. We propose to use the star splaying algorithm to transform the obtained triangulation into the Delaunay triangulation. The star splaying algorithm is modified to work adaptively on only those areas with unflippable facets. Overall, our algorithm consists of two phases.

Phase 1: Parallel point insertion and flipping.

Use incremental insertion together with flipping to construct a near-Delaunay triangulation. This phase is performed on the GPU.

Phase 2: Adaptive star splaying.

Apply the star splaying algorithm adaptively to transform the result of Phase 1 into the Delaunay triangulation. This phase is performed on the CPU.

It is possible to perform Phase 2 on the GPU, similar to Section 4.5.2. However, we find that it is easier to do this on the CPU, and actually it allows the work to be smaller since the computation can be done adaptively on only some small areas on the triangulation.

We note that in $\mathbb{R}^2$, only Phase 1 is enough to construct the Delaunay triangulation. In the following discussion, we focus on the problem in $\mathbb{R}^3$. Nevertheless, many techniques discussed below can still be applied on the 2D implementation.

5.3.1 Phase 1: Parallel point insertion and flipping

A simple approach to perform Phase 1 on the GPU is shown in Algorithm 5.5. The algorithm starts by constructing an arbitrary triangulation of the point set S using incremental insertion

Algorithm 5.5: Incremental insertion and flipping to construct the Delaunay triangulation (version 1).

```
   Data: A point set S
   Result: D(S)
1  initialize T with a large enough simplex t
2  for each p ∈ S do in parallel location[p] ← t
3  while ¬Empty(S) do
4  |  for each p ∈ S do in parallel insert[location[p]] ← p
5  |  for each t ∈ T with insert[t] ≠ null do in parallel
6  |  |  split t using insert[t] and remove insert[t] from S
7  |  for each p ∈ S do in parallel update location[p] if it was split
8  end
9  label all facets in T as to be checked
10 while there are facets to be check do
11 |  for each facet f that needs to be checked do in parallel
12 |  |  if f is non-locally Delaunay and flippable then
13 |  |  |  flip f
14 |  |  |  label all updated facets as to be checked
15 end
16 return T
```

(line 3–8). In each iteration, first we pick for each triangle (or tetrahedron) a point inside it to insert, if any (line 4). Then we split the triangle (tetrahedron) and update the neighbors in the next kernel (line 6). Finally, we update the location of the points that are left in S if its old location was split (line 7).

In the second stage, we repeatedly check the facets in $\mathcal{T}$ to find the non-locally Delaunay facets that are flippable, and flip them. This flipping and updating of the neighboring information also requires several kernels to avoid race condition, as described in Section 2.1.5.

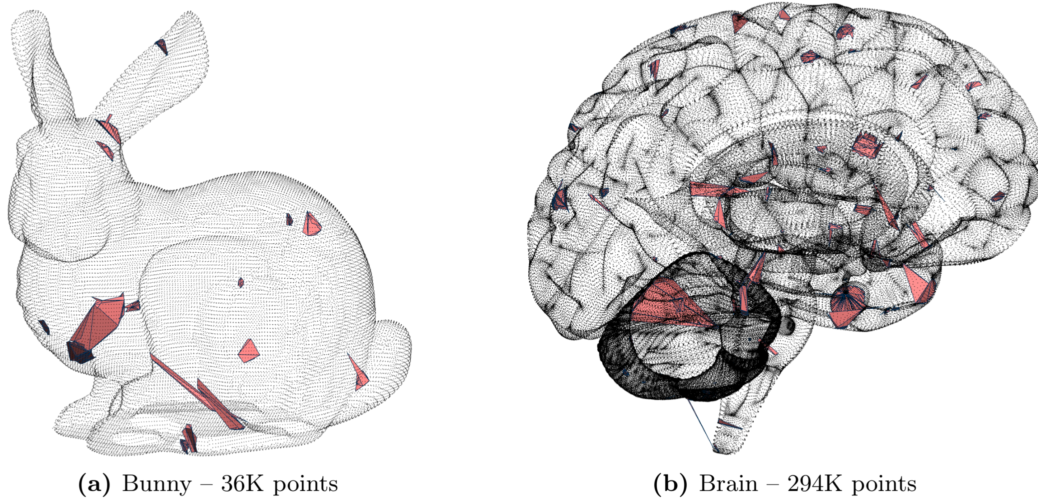
This simple approach, however, leads to many locally non-Delaunay facets in the triangulation in $\mathbb{R}^3$ after flipping, to the extent that it is not practical to be corrected in Phase 2. Instead, we propose to apply flipping after each iteration of point insertion. This has two benefits. First, the earlier we flip, the easier to resolve the locally non-Delaunay facets, and thus fewer unflippable facets remain when we get stuck, as shown later in our experiment. Second, flipping increases the number of tetrahedra significantly before the next iteration of point insertion. If no flipping is performed, then before each iteration of flipping the number of

Algorithm 5.6: Incremental insertion and flipping to construct the Delaunay triangulation (version 2).

```
   Data: A point set S
   Result: D(S)
1  initialize T with a large enough tetrahedron t
2  for each p ∈ S do in parallel location[p] ← t
3  while ¬Empty(S) do
4  |  for each p ∈ S do in parallel insert[location[p]] ← p
5  |  for each t ∈ T with insert[t] ≠ null do in parallel
6  |  |  split t using insert[t] and remove insert[t] from S
7  |  |  label all new facets to be checked
8  |  while there are facets to be checked do
9  |  |  for each facet f that needs to be checked do in parallel
10 |  |  |  if f is locally non-Delaunay and flippable then
11 |  |  |  |  flip f
12 |  |  |  |  label all updated facets to be checked
13 |  end
14 |  update the locations of points in S
15 end
16 return T
```

tetrahedra in $\mathcal{T}$ is $3m$ where m is the number of points inserted so far. By applying flipping after each insertion iteration, the number of tetrahedra in $\mathcal{T}$ approaches the expected value of $6.67m$, the number of tetrahedra in the Delaunay triangulation of a uniformly distributed point set in $\mathbb{R}^3$ [Dwy91]. This means that in the next iteration we can insert more points in parallel, and thus fewer iterations are needed. Besides, if we look at this problem as computing the convex hull in lifted space, then flipping in earlier iterations increases the volume faster, so we expect that less flips are required.

Our new approach is detailed in Algorithm 5.6. In each iteration, we first insert one batch of points into the triangulation (line 4–7). We also label the new facets so that they are checked in the subsequent flipping. The flipping is performed right after a batch of points is inserted (line 8–13). We repeatedly check the facets in $\mathcal{T}$ to find those locally non-Delaunay facets that are flippable, flip them, and update the neighboring information. Finally, we update the locations of the points that are left in S if their old locations were split (line 14), to prepare for the next round of point insertion.



(a) Bunny – 36K points

(b) Brain – 294K points

Figure 5.8: At the end of the point insertion and flipping phase of our IncDel3D algorithm, less than 0.05% of the facets, shaded in the figure, are locally non-Delaunay.

The result of this phase is a triangulation that is very close to the Delaunay triangulation. There are still facets that are not locally Delaunay but are all unflippable. However, using our strategy, this number is usually very small. Figure 5.8 shows the locally non-Delaunay facets at the end of Phase 1 during the Delaunay triangulation construction of two 3D models.

5.3.2 Phase 2: Adaptive star splaying

The star splaying algorithm used in this phase to transform the result of Phase 1 into the Delaunay triangulation is adaptive, and is done sequentially on the CPU. As we show in our experiment, the work needed here is small and thus does not affect the performance of our algorithm.

There are three steps in this phase. Step 1 is to construct the convex stars in $\mathbb{R}^4$. Step 2 is to make the stars consistent by splaying. Step 3 is to convert the stars into a triangulation. These three steps are performed adaptively, with the goal that the work done should be proportional to the actual changes needed.

The three key ideas of our adaptive star splaying algorithm are as follows. First, consider a vertex $s \in \mathcal{T}$ and its star. If s is not incident to any locally non-Delaunay facet, i.e. its star facets are all locally Delaunay, then when lifted to $\mathbb{R}^4$, its star is already convex. Therefore, only stars of the vertices that are incident to some locally non-Delaunay facets are not yet convex and need to be corrected. Other stars can be derived directly from $\mathcal{T}$ without any modification. Second, inside each star that is reconstructed, only the tetrahedra that do not appear in $\mathcal{T}$ need to be checked for consistency, since those that are in $\mathcal{T}$ should already be

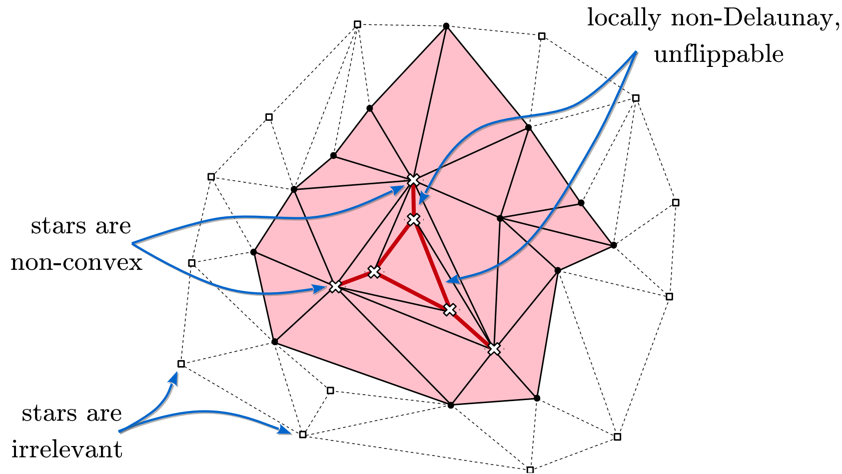


Figure 5.9: 2D illustration of the adaptive star splaying algorithm. Only the shaded region is modified.

consistent. Third, only the stars that are modified need to be converted and patched back into $\mathcal{T}$. These ideas are illustrated in Figure 5.9. Applied correctly, these three ideas help us achieve the goal mentioned earlier.

Building the initial stars

We only rebuild the stars of the vertices that are incident to some locally non-Delaunay facets. These vertices are called *failed vertices*. Since the locally Delaunay checks on all facets have been done on the GPU during the flipping, we use the GPU to collect those failed vertices and pass it to the CPU.

A naïve approach to construct the star of a vertex s is to find all its neighbors in $\mathcal{T}$ and insert them one by one using the beneath-beyond method. This, however, is very costly because a vertex may have many neighbors. Also, it is wasteful since the star of s in $\mathcal{T}$ should already be very close to be convex. Instead, we consider the stars of $\mathcal{T}$ lifted to $\mathbb{R}^4$. Since any vertex s is an extreme vertex, the star of s is contained in a half-space. Thus, there exists a hyperplane $\mathcal{H}$ near s in $\mathbb{R}^4$ that cuts through the star of s. This intersection is a polyhedron $\mathcal{P}$, and the problem of constructing the convex star for s is equivalent to making this polyhedron the convex hull. Figure 5.10 illustrates this in $\mathbb{R}^2$ lifted to $\mathbb{R}^3$. In the following discussion, the term "vertical" means along the lifting direction.

Lemma 5.5. *The polyhedron $\mathcal{P}$ on $\mathcal{H}$ is star-shaped w.r.t the intersection of $\mathcal{H}$ and the vertical line through s.*

Proof. We sweep each tetrahedron t in the star of s in $\mathbb{R}^3$ along the vertical direction, forming

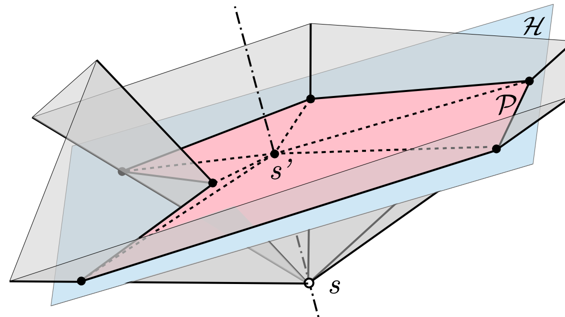


Figure 5.10: Constructing the convex star of s in $\mathbb{R}^2$ lifted to $\mathbb{R}^3$. All vertices shown in the figure are already lifted.

non-overlapping wedges in $\mathbb{R}^4$. Note that this is the same as sweeping the tetrahedra in the lifted star of s. Thus, these wedges intersect $\mathcal{P}$ at some non-overlapping tetrahedra, while the vertical line through s intersects $\mathcal{H}$ at a point s' that is inside $\mathcal{P}$ and is a vertex of all these tetrahedra. The boundary of the polyhedron $\mathcal{P}$ is actually the link of s'. Therefore, $\mathcal{P}$ is star-shaped w.r.t to s' on $\mathcal{H}$. □

This lemma allows us to use the Flip-Flop algorithm in Section 5.2.2 to compute the convex hull of $\mathcal{P}$, which is equivalent to computing the convex star of s. That is much more efficient than constructing the convex stars from scratch. We retrieve the link triangulation of s from $\mathcal{T}$, and apply Flip-Flop to transform it into a convex star in $\mathbb{R}^4$. The hyperplane $\mathcal{H}$ is used for explanation only, and it needs not be explicitly computed. The actual orientation tests can be done directly in $\mathbb{R}^4$ with respect to the point s.

Note that all the Delaunay checks have been done on the GPU in Phase 1. As such, when applying the Flip-Flop algorithm to construct each convex star, we start the checking and flipping from those that were previously marked as locally non-Delaunay facets. That saves a lot of checking cost for the construction.

Adaptive splaying

The splaying step is done as described in Section 3.2.3 and Section 4.5.2. During this step, we may need to access some stars which were not constructed in the previous step. We simply retrieve these stars from $\mathcal{T}$, since they are already convex.

If we check all the tetrahedra created in the previous step for consistency, then we need to pull in the stars of all the vertices incident to the failed vertices. This might turn out to be unnecessary if these stars are still consistent. We observe that if a tetrahedron already exists in $\mathcal{T}$, then it need not be checked since any three of its vertices should have already be on or inside the convex star of the fourth one. Thus, during the flipping process the previous step,

we only label the tetrahedra that are modified, from which we start the consistency check in this step. This reduces the number of checks as well as the number of stars that need to be retrieved from $\mathcal{T}$.

Patch the triangulation

After the stars are consistent, $\mathcal{T}$ needs to be updated. Consider the set T_p of all tetrahedra in $\mathcal{T}$ we have previously used to build the stars, and let T_n be the set of tetrahedra we can derive from the set of new stars after splaying. During the star construction, we keep a map between the tetrahedra in the stars and the corresponding ones in $\mathcal{T}$. From that, we find the set of newly created tetrahedra $T^+ = T_n \setminus T_p$ and the set of deleted tetrahedra $T^- = T_p \setminus T_n$. We remove the tetrahedra in T^- from $\mathcal{T}$, and add those in T^+ to $\mathcal{T}$.

Next we update the connectivity between the tetrahedra. We do not need to process those in $T_p \setminus T^-$, which are the old tetrahedra that still survive. Instead, for each tetrahedron t in T^+, we set its 4 neighbors and also update the neighbors to point to t using the connectivity from the stars. This way, for the tetrahedra in $T_p \setminus T^-$ that are adjacent to some new tetrahedra, the connectivity is updated correctly. As a result, we only update the portions of $\mathcal{T}$ that are changed. After this step, the resulting triangulation is the Delaunay triangulation.

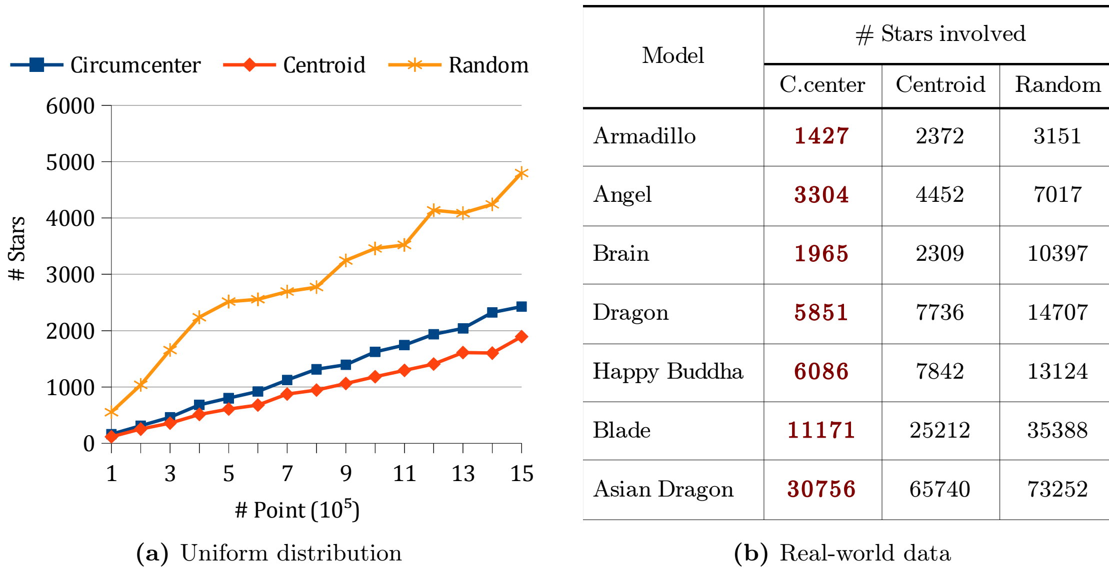
5.3.3 Point insertion heuristic

During the insertion phase, we insert points in batches. Each simplex (triangle or tetrahedron) received at most one point per round to avoid race condition. In each iteration, if there are multiple points in a simplex, then we need to choose which one to insert. In the convex hull construction algorithm in Section 5.2, we pick the point farthest from the triangle so that we can maximize the number of non-extreme points we can discard afterward. For Delaunay triangulation, we propose to insert the point that is nearest to the circumcenter of the simplex.

The motivation of inserting near the circumcenter of a simplex is clear. When lifted to $\mathbb{R}^4$, constructing the Delaunay triangulation is equivalent to constructing the lower hull. Inserting the point nearest to the circumcenter of the simplex is the same as inserting the point farthest to the lifted simplex. In doing so, the volume of the hull grows the most and gets closer to the convex hull, thus the number of flipping in the next phase can be reduced.

Another reason is the point nearest to the circumcenter is also farthest from the vertices of the simplex, i.e. the minimum distance is maximal. As such, the facets created during the insertion are of better quality and thus it is also easier for the flipping later.

To evaluate the efficiency of this heuristic, we analyze the number of stars involved in Phase 2, including those that are created initially and those that are retrieved from the result of



(a) Uniform distribution

Model	# Stars involved		
	C.center	Centroid	Random
Armadillo	**1427**	2372	3151
Angel	**3304**	4452	7017
Brain	**1965**	2309	10397
Dragon	**5851**	7736	14707
Happy Buddha	**6086**	7842	13124
Blade	**11171**	25212	35388
Asian Dragon	**30756**	65740	73252

(b) Real-world data

Figure 5.11: The running time and the number of stars involved of IncDel3D on uniform point distribution when using different point insertion strategy.

Phase 1. We compare the value when using the proposed approach with two other strategies: picking the point near the centroid of the tetrahedron, and picking randomly. Figure 5.11a shows the result on the uniform distribution. Clearly, inserting near the circumcenter is significantly better than choosing a random point to insert, with the number of stars involved in Phase 2 reduces by nearly two times. Interestingly, picking points near the centroid to insert performs slightly better than our choice of inserting points near the circumcenter first. However, the situation becomes clearer when we look at the result on real-world data, as shown in Figure 5.11b. Given the input points from those real 3D models which are usually non-uniformly distributed, our heuristic shows a major advantage. It gives a much better result after Phase 1, and thus in Phase 2 the number of stars involved is significantly reduces, by up to 5 times less than picking points randomly to insert, and up to 2.5 times better than picking points near the centroid.

5.3.4 Point relocation and the history DAG

During Phase 1, as described in Algorithm 5.6, we need to keep updating the locations of points in S after flipping so that the point insertion in the next iteration can be performed. Relocating the remaining points after a new batch of points is inserted is simple, since each existing tetrahedron is either split into four tetrahedra, or is unmodified. However, during flipping, the tetrahedra are also modified. A simple approach is to update after each iteration of flipping. This, however, is not GPU friendly, since all the points need to participate in

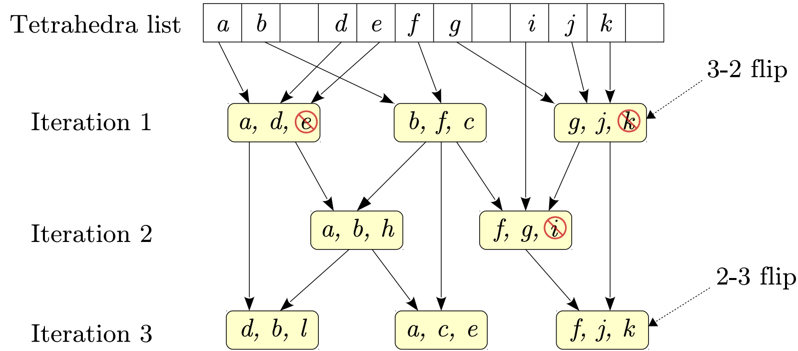


Figure 5.12: A history DAG of flipping in 3D.

Algorithm 5.7: Construct the history DAG from the list of flips.

```
1  let k be the number of flip iterations performed
2  initialize last[t] = null for all tetrahedra t
3  for i = k to 1 do
4  |  for each flip node x performed in iteration i do in parallel
5  |  |  x.n1 = last[x.t1], x.n2 = last[x.t2]
6  |  |  last[x.t1] = last[x.t2] = x
7  |  |  if f is a 2-3 flip then
8  |  |  |  x.n3 = last[x.t3]
9  |  |  else
10 |  |  |  last[x.t3] = x
11 end
```

the relocation step but only few of them are affected by the flips in this iteration. Instead, we record all the flips done in the flipping loop into a directed acyclic graph (DAG), and use this data structure to relocate the points; see Figure 5.12. This history DAG stores the evolution of the triangulation during the flipping. Each node represents a flip, containing the indices of the 5 vertices and the three tetrahedra involved. Note that we reuse the tetrahedra indices, so a 2-3 flip transforms $\{t_1, t_2\}$ to $\{t_1, t_2, t_3\}$, and vice versa. Each node has up to three pointers $\{n_1, n_2, n_3\}$ that point to the nodes corresponding to the future flips that modify the tetrahedra created in this flip.

The history DAG is constructed as follows. During the flipping, we record all the flips as nodes in the DAG, without pointing them to each other. After that, we build the connectivity by processing the iterations of flipping bottom up; see Algorithm 5.7. We use last[t] to

store the last flip node that modifies t. From bottom up, the flips in each iteration are processed in parallel. For each flip f creating tetrahedra t_1, t_2 (and possibly t_3), we update the corresponding node in the DAG. We point that node to the two (or three) nodes that correspond to the future flips that modify its tetrahedra, using the last array (line 5, 8). Then, we update the last array accordingly (line 6, 10). By processing the iterations of flipping from bottom up, setting the pointers are coherent memory writes.

To update the point locations using the history DAG, each thread processing a remaining point s starts from its location t and follows the nodes in the history DAG from last[t] till it reaches a null pointer. At each flip node we use one (or two) orientation tests to determine the new location after that flip. The last tetrahedron recorded is the new location of s.

5.3.5 Compaction versus collection

Consider the flipping loop at line 8–13 of Algorithm 5.6. We assign one thread per tetrahedron, checking its four facets if they are labeled. Any duplicate checks caused by two tetrahedra sharing a facet can be avoided by comparing their indices. We observe that after a few iterations, many tetrahedra have no more facets to be checked, so many threads are idle thus reducing the efficiency due to thread divergence. Two solutions are possible.

The first solution is to compact the list of tetrahedra after each iteration. We label the tetrahedra that need to be checked in this iteration; they are called *active tetrahedra*. We use parallel stream compaction to compact all the active tetrahedra before launching the kernel to check them. This is costly near the end of the flipping where few tetrahedra are involved.

The second solution is to collect the tetrahedra to be checked in the next iteration during the flipping. Each flip modifies at most 3 tetrahedra, so we pre-allocate an array of size 3 times the number of active tetrahedra in this iteration of flipping. Each flip writes down the tetrahedra that it modified into this array. The array is then compacted and used in the next iteration of flipping. This strategy scales well with the number of active tetrahedra in each iteration, but it also shuffles the order of the tetrahedra to be checked, and thus the memory access is not in order.

In our implementation, we combine these two approaches. In the first few iterations when the number of active tetrahedra is still very large, we use the first strategy. When this number drops to below a certain threshold, we switch to the second strategy. This approach can avoid the disadvantage of both strategies and achieve the best performance.

5.3.6 Memory access optimization

Given the enormous parallel computation of the GPU, the performance of our algorithms is usually limited by the memory system. The GPU memory has very high latency, and if the

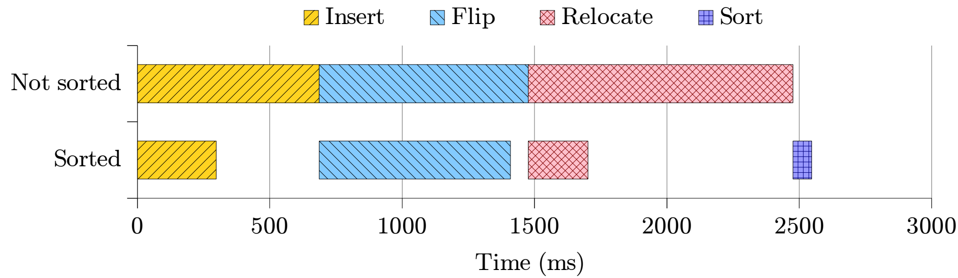


Figure 5.13: The time breakdown of IncDel2D with and without sorting.

accesses are not coherent, then they will be serialized. The latest GPU architecture has an L2 cache, but this cache is very small thus it is important to improve the memory access pattern of our algorithm. There are several approaches to rearrange the data in such a way that the cache is better utilized to reduce the time taken in Phase 1.

First, we observe that during point insertion as well as point relocation, each thread processing a point needs to access the triangle or tetrahedron containing that point, including the coordinates of the vertices, leading to a lot of random memory access. We propose to sort the input points along the Hilbert space filling curve at the beginning of the algorithm. By doing so, points that fall into the same simplex tend to stay near each other in the point list and will be processed by adjacent threads. As such, the accesses by these threads are shared through the cache and the performance is improved. Especially for point relocation, nearby threads tend to travel the same path in the history DAG.

The benefit of this sorting in our 2D implementation is shown in Figure 5.13. In this experiment, 10^7 points uniformly distributed are used as input. As we expect, by sorting the input points, the performance of the point insertion increases by more than 2 times, while that of the point relocation increases nearly 4 times. The extra cost of just a single time sorting, plus updating the point indices back to the original order before output, is insignificant compared to the benefit.

The sorting of input points benefit the point insertion and point relocation tasks, but not the flipping, while the task of flipping is actually the most time consuming one in the 3D Delaunay triangulation problem. During flipping, each thread processing a tetrahedron needs to access its neighboring tetrahedra. It has been shown that by sorting the tetrahedra by its minimum vertex index, accessing neighboring tetrahedra can be faster. However, after each iteration of flipping, the tetrahedra list will be modified and the order may need to be updated. It is not possible to sort the tetrahedra list after each and every iteration of flipping, since that would be too costly. One simple approach is to sort this list only after a new batch of points is inserted, but the cost of sorting is still significant. Besides, during flipping in 3D, additional tetrahedra are added (in the 2–3 flips), thus further scramble the

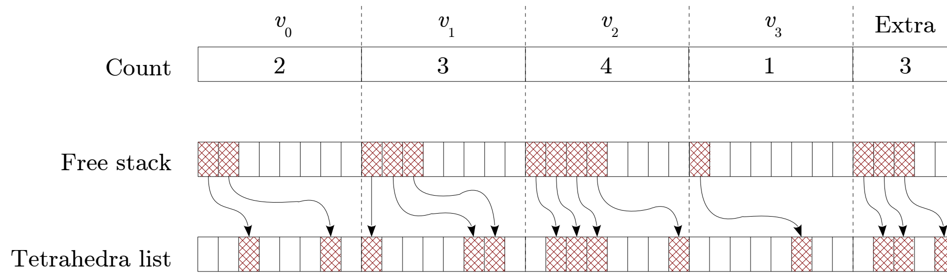


Figure 5.14: The self-sorting data structure for 3D Delaunay triangulation.

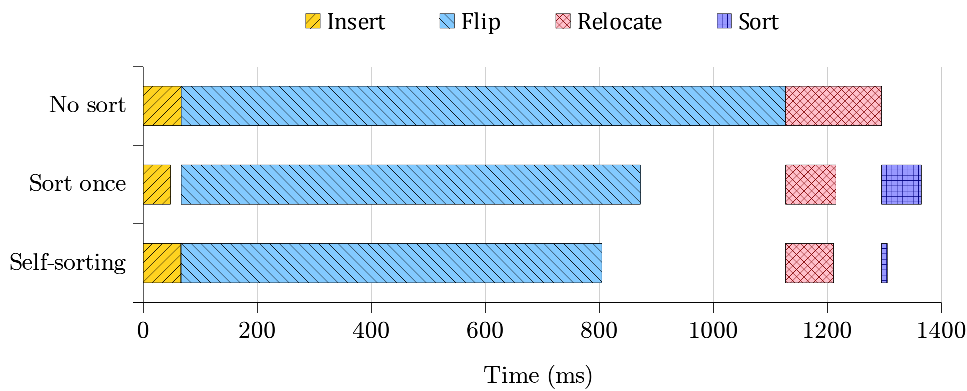


Figure 5.15: The time breakdown of IncDel3D with different data reordering strategies.

sorted order.

We propose a self-sorting data structure to store the tetrahedra such that the sorted order can be maintained during flipping. The idea is to reserve space in the list so that during flipping, new tetrahedra can be inserted at the appropriate places. In the tetrahedra list, each vertex is given a *storage* of k slots. We also keep track of the empty slots in the storage of each vertex; see Figure 5.14. During flipping, for each 3–2 flip, the new empty slot is recorded. For each 2–3 flip, we check the storage of each of the four vertices of the additional tetrahedron to find an available slot. If all the four storages are full, then the tetrahedron is added to the end of the tetrahedra list. The above allocation is done in parallel using atomic operations. Since the number of vertices is large, the contention for these atomic operations is quite small, and thus it does not affect the performance. In our experiment, the average degree of a vertex is around 27, so by choosing $k = 8$, there is a very high chance that the additional tetrahedra are stored at appropriate locations on the list. As a result, the tetrahedra list is always kept partially sorted during flipping.

Figure 5.15 shows the running time of different tasks in our 3D Delaunay triangulation implementation, using different data reordering strategies. These include using the self-sorting data-structure, sorting once after each batch of point insertion, and not sorting at

all. Clearly, just by sorting once, the point insertion and the point relocation tasks already become much faster. The flipping time also reduces by 25%, but at a substantial cost of sorting. The self-sorting data-structure not only reduces the flipping time further, but also removes the sorting cost. As such, in the experiment sections below, we always use this data-structure.

5.3.7 2D Experiment

We apply the proposed algorithm to compute the 2D Delaunay triangulation. Since flipping never gets stuck in $\mathbb{R}^2$, we do not need the adaptive star splaying in Phase 2 of the algorithm. Other than that, all the previously proposed techniques are still helpful. Note that since in $\mathbb{R}^2$ we only need to perform 2–2 flips, there is no need to rearrange the triangles during flipping to improve cache efficiency. Nonetheless, sorting the input points in the beginning is still very helpful. In the following sections, we compare the performance of our implementation, termed IncDel2D, with Triangle, CGAL, as well as the DigiDel2D, the algorithm proposed in the previous chapter.

InsertAll-Flip versus Insert-Flip

First of all, we look at the benefit of applying flipping right after each batch of point insertion. This strategy, termed Insert-Flip, is compared against the simpler strategy of incrementally inserting all the points before doing any flipping, which we call the InsertAll-Flip strategy. Figure 5.16a shows the running time comparison between these two strategies, with input points from a uniform distribution. The Insert-Flip strategy gains about 15% faster. The reason is that the Insert-Flip strategy requires less flips in total to reach the Delaunay triangulation. This is verified in Figure 5.16b where we plot the number of flips used by the two strategies. In general, the Insert-Flip strategy needs more than 30% less flips. Figure 5.16c shows the time spent in inserting points, flipping, and updating point locations, for an input of 10^7 points. Although when mixing point insertions and flipping, the point relocation cost increases quite a bit, the flipping time reduction is much higher, and thus overall the Insert-Flip strategy is still better.

Synthetic data

Figure 5.17a shows the speedup of IncDel2D on the uniform point distribution. In general, IncDel2D outperforms Triangle by around 6 to 10 times, CGAL by 6 to 7 times, and this speedup increases as the number of points increases. IncDel2D is also about 2 times faster than DigiDel2D. Similar behavior is observed with the Gaussian distribution.

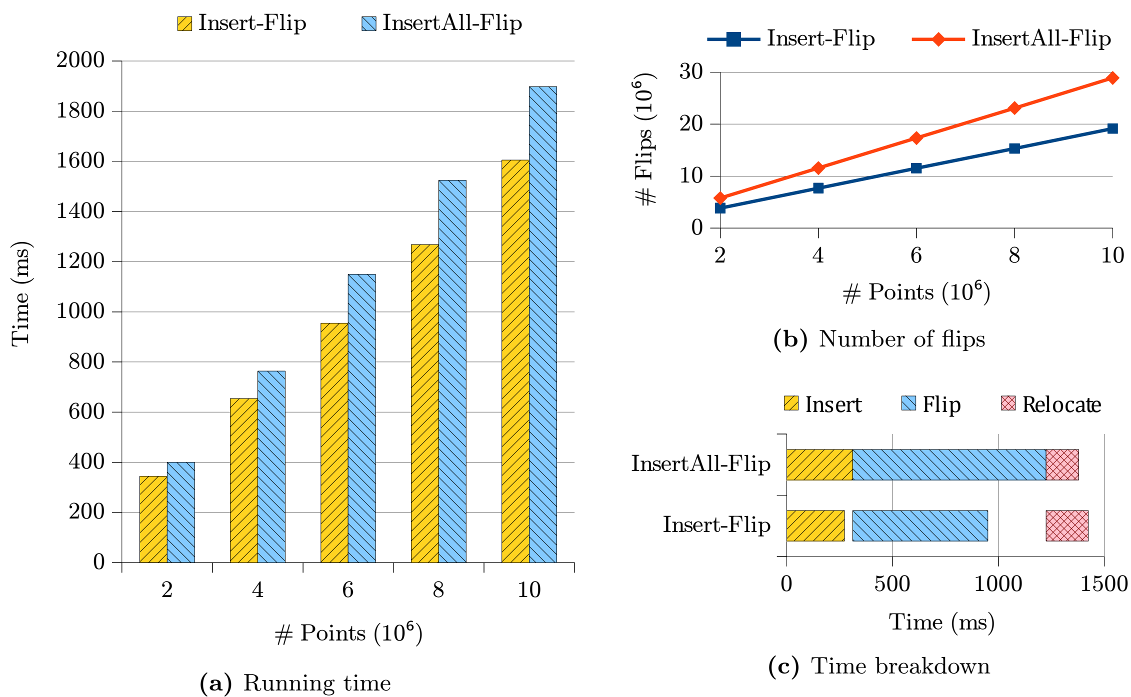


Figure 5.16: Comparing two different strategies for Phase 1 in IncDel2D.

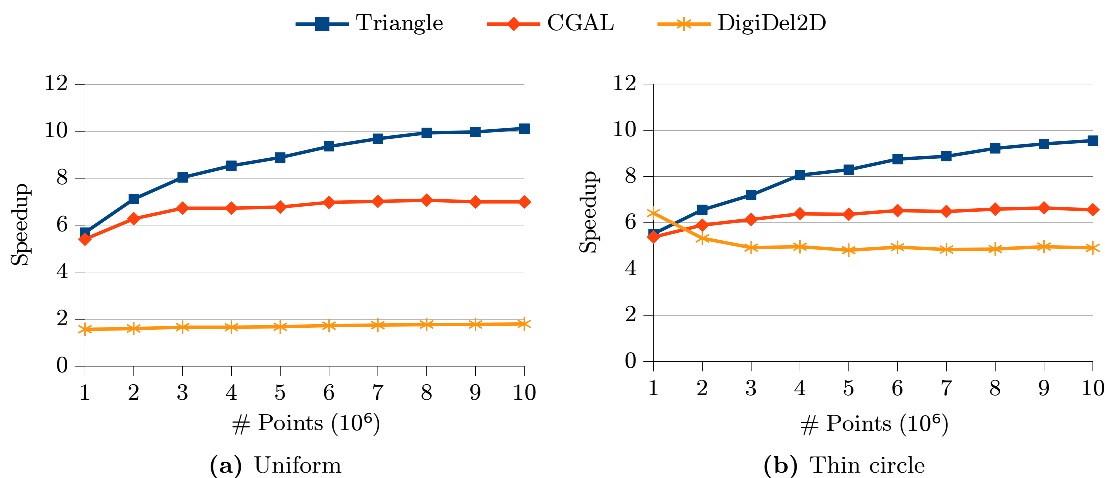
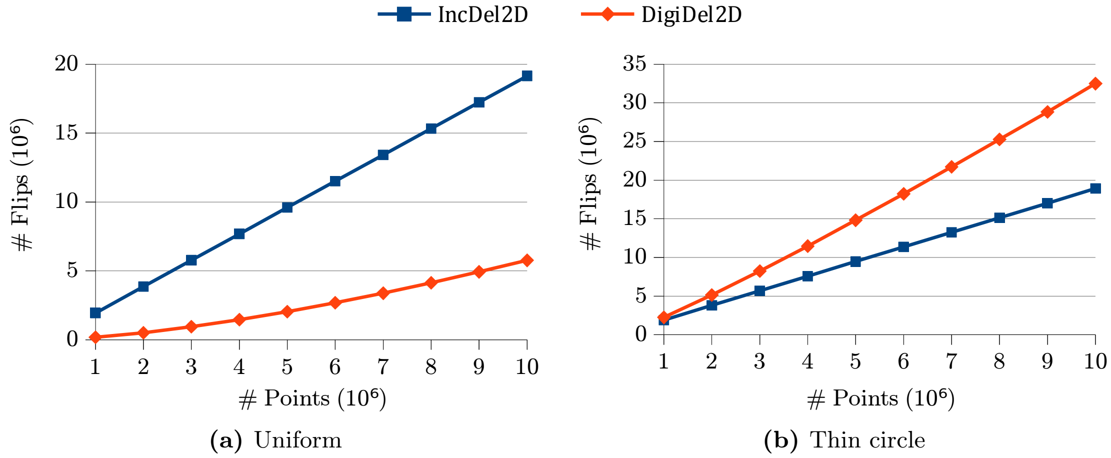


Figure 5.17: The speedup of IncDel2D compared to Triangle, CGAL and DigiDel2D.



(a) Uniform

(b) Thin circle

Figure 5.18: The number of flips performed by IncDel2D versus DigiDel2D.

We also look at the performance on the thin circle distribution in Figure 5.17b, which is where DigiDel2D falls short. For this distribution, it is very difficult for the digital Voronoi diagram to properly approximate the continuous one. IncDel2D achieves almost similar speedup over CGAL and Triangle as in the case of the uniform distribution, while it is about 5 times faster than DigiDel2D, even when the latter uses the 8192^2 grid size.

In Figure 5.18, we compare the number of flips performed by our two GPU algorithms. The design of DigiDel2D is such that the sketch obtained through digital computation is close to the final solution, thus the number of flips needed to transform it into the Delaunay triangulation should be small. Clearly, for the uniform distribution, this is true. IncDel2D requires more than 3 times more flips than DigiDel2D. The reason that DigiDel2D is slower than IncDel2D is possibly due to different implementation and optimizations. However, the scenario is reversed when we look at the result in the thin circle distribution. In this case, since the digital Voronoi diagram can no longer capture the continuous one accurately, the sketch obtained is not very near the Delaunay triangulation, and thus more flips are needed. In fact, DigiDel2D requires 50% more flips than IncDel2D. Due to the benefit of using the Insert-Flip strategy, IncDel2D uses almost the same number of flips in both the uniform and the thin circle distribution.

Real-world data

We present the running time of IncDel2D on the contour datasets in Figure 5.19. As expected from the experiment in the previous section, IncDel2D is only slightly affected by these non-uniformly distributed point sets, with the speedup over Triangle ranges from 6 to 9 times, and that over CGAL is 5.5 to 6.5 times. IncDel2D also outperforms DigiDel2D by 2.5 times, which is as expected since DigiDel2D has more difficulty handling these cases.

Set	# Vertices	Time (ms)	Speedup		
			Triangle	CGAL	DigiDel2D
1	1,177,332	231	**5.9**	**5.6**	**2.4**
2	3,180,037	569	**7.4**	**6.3**	**2.4**
3	4,461,519	802	**7.9**	**6.3**	**2.4**
4	5,721,142	1001	**8.4**	**6.5**	**2.5**
5	8,569,881	1512	**9.0**	**6.6**	**2.4**
6	9,546,638	1702	**9.0**	**6.5**	**2.5**

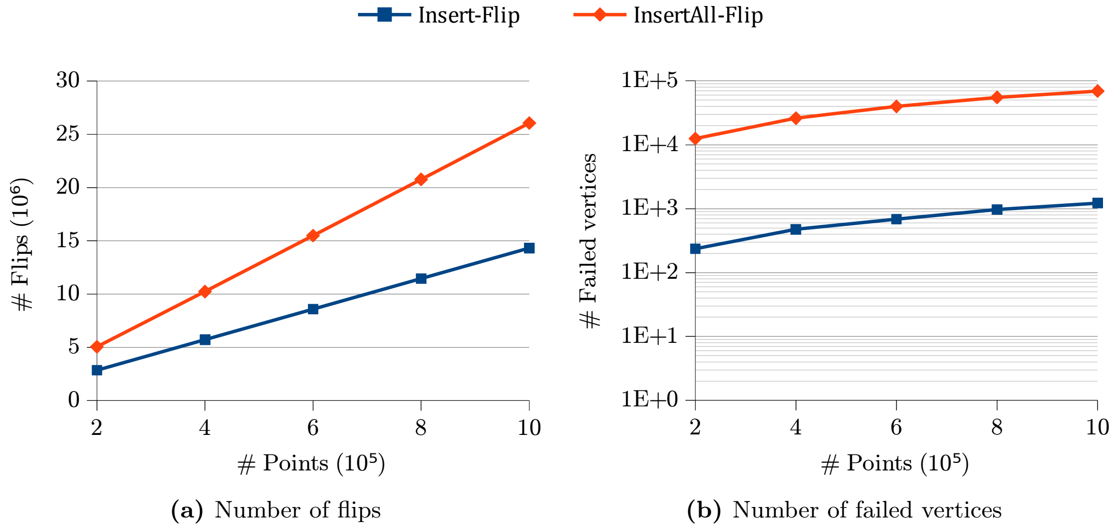
Figure 5.19: The running time of IncDel2D and its speedup over Triangle, CGAL, and DigiDel2D on the contour datasets.

5.3.8 3D Experiment

We compare the performance of our 3D implementation, called IncDel3D, with that of CGAL and DigiDel3D, using all the techniques discussed earlier such as mixing point insertion and flipping, inserting point near to the circumcenter of the tetrahedra, and rearranging the data in memory. We also look at how much more effort is needed in Phase 2 to transform the result after flipping in Phase 1 into the Delaunay triangulation.

InsertAll-Flip versus Insert-Flip

Here first analyze the advantage of using the Insert-Flip strategy versus the InsertAll-Flip strategy. The experiment is done on the uniform distribution. First, similar to the 2D case, this makes the number of flips needed smaller, as can be seen in Figure 5.20a. With the Insert-Flip strategy, the number of flips reduces by about 40%. However, this is not the main advantage of the Insert-Flip strategy in 3D. The main benefit is that it reduces the number of unflippable facets, and thus the number of failed vertices that need to be fixed in Phase 2. As such, in Figure 5.20b, we notice nearly a two order of magnitude reduction in the number of failed vertices when using the Insert-Flip strategy. This is the main reason why performing Phase 2 on the CPU is practical.



(a) Number of flips

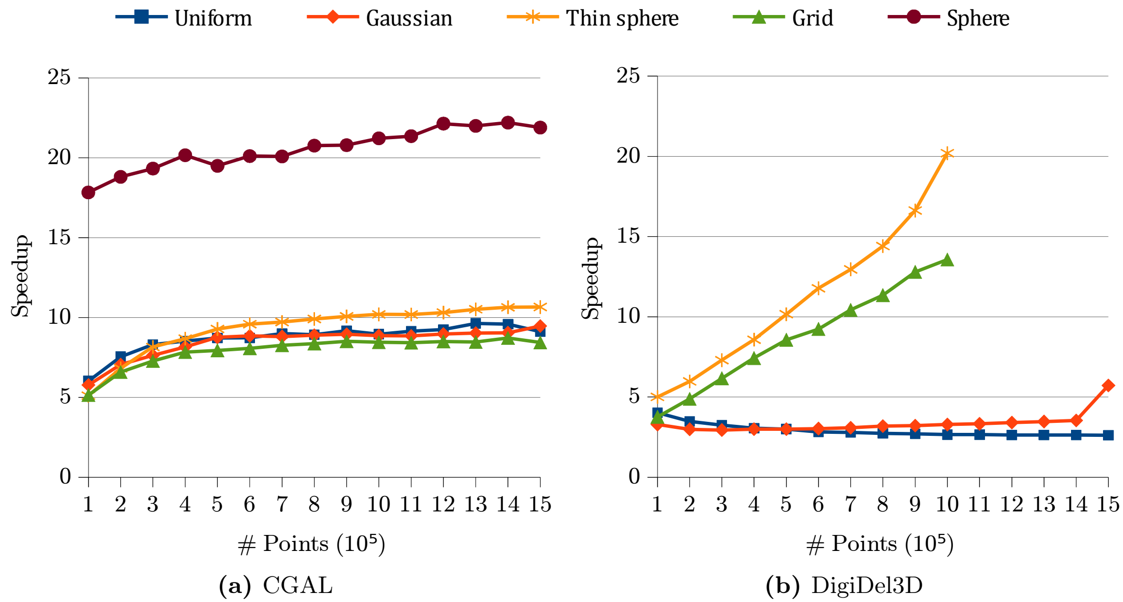
(b) Number of failed vertices

Figure 5.20: The number of flips and the number of failed vertices of IncDel3D using the Insert-Flip strategy compared to the InsertAll-Flip strategy.

Synthetic data

Figure 5.21a shows the speedup of IncDel3D over CGAL on different point distributions, with the input size ranges from 10^5 to 15×10^5. Besides the uniform, Gaussian and thin sphere distribution, we also try two pathological distributions: the grid distribution and the sphere distribution; see Section 2.3 for more details. The speedup starts at 5 to 6 times, and quickly rises to 8 to 10 times when the number of points increase. Particularly, for the sphere distribution, not only can IncDel3D handle them robustly, but the speedup over CGAL is even higher, reaching more than 20 times speedup. This shows that our approach to handle exact computation on the GPU is very efficient. This also means that IncDel3D is not much affected by the distribution of points.

On the other hand, Figure 5.21b shows the comparison between IncDel3D and DigiDel3D, the algorithm using the digital Voronoi diagram introduced in the earlier chapter. Unlike IncDel3D, DigiDel3D is significantly affected by the point distribution. For the uniform and the Gaussian distribution, IncDel3D runs about 3 times faster than DigiDel3D. However, on the thin sphere and the grid distribution, the running time of DigiDel3D increases sharply as the number of points increases, and thus the speedup of IncDel3D also increases. In fact, DigiDel3D cannot handle these two distributions with more than 10^6 points since it runs out of GPU memory. For the thin sphere distribution, this is because the digital computation on the grid of finite resolute cannot capture the Voronoi diagram fully. For the grid distribution, the problem is different. In this pathological case, there are a lot of coplanar points, and thus the star splaying phase requires the Simulation of Simplicity (SoS) technique [EM90] to perturb the points to get a consistent result, while the digital Voronoi diagram computation



(a) CGAL (b) DigiDel3D

Figure 5.21: The speedup of IncDel3D compared to CGAL and DigiDel3D on different point distributions.

is done in the digital space without this perturbation. As for the sphere distribution, the performance of DigiDel3D is much worse than with the thin sphere distribution, and is thus excluded in the chart.

Real-world data

Recall that the aim of the new approach in this chapter is to achieve good speedup even on real-world datasets. Figure 5.22 shows the running time of IncDel3D on some 3D models, as used in the other experiments, and the speedup over CGAL and DigiDel3D. While DigiDel3D has a lot of difficulty handling these non-uniformly distributed point sets, IncDel3D can handle them easily. Without the overhead of computing the digital Voronoi diagram, IncDel3D outperforms CGAL even on small inputs. The speedup ranges from 7 to 10 times, even for models that contain a lot of degeneracies like the Blade model. Compared to DigiDel3D, the new algorithm is 5 to 10 times faster. Moreover, IncDel3D requires less memory than DigiDel3D, so it can handle the Asian Dragon model with more than 3.5×10^6 points on our GPU while DigiDel3D cannot.

Detailed analysis

Figure 5.23 shows the percentage of running time taken by different tasks in our implementation, including Phase 1 and Phase 2, on different point distributions with 10^6 points. These

Model	# Points	IncDel3D time (s)	Speedup	
			CGAL	DigiDel3D
Armadillo	172974	0.3	7.0	5.5
Angel	237018	0.4	7.9	12.1
Brain	294012	0.4	9.4	6.3
Dragon	437645	0.8	8.0	8.6
Happy Buddha	543652	0.9	8.7	12.2
Blade	882954	1.6	7.6	19.2
Asian Dragon	3609600	4.7	10.7	-

Figure 5.22: The running time and speedup of IncDel3D on different 3D models.

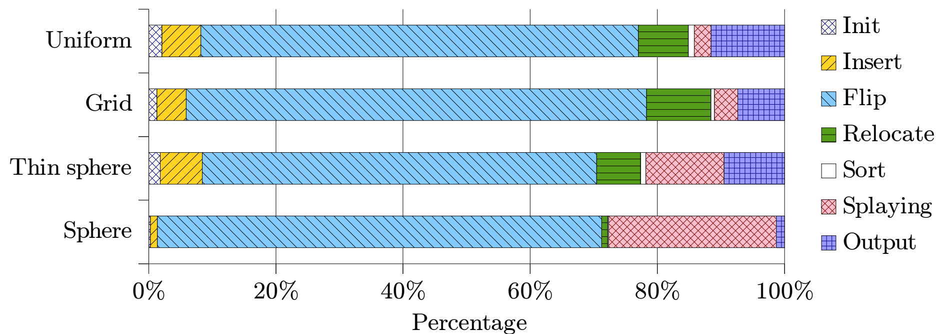


Figure 5.23: The time breakdown of IncDel3D with different point distribution.

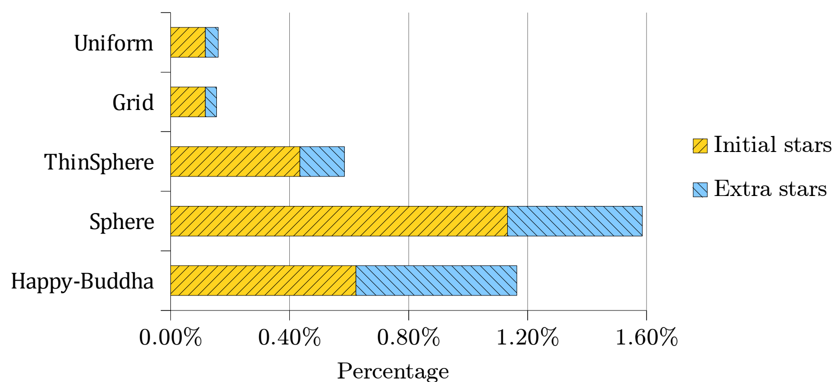


Figure 5.24: The percentage of stars involved in Phase 2.

tasks are initialization, inserting points, flipping, relocating points, sorting, splaying, and outputting the result to CPU memory. As we expect, in the uniform, the Gaussian and the grid distribution, majority of the running time is spent in flipping. Also, by using the history DAG, although the number of flips is so high, the point relocation task still takes quite a moderate amount of time. It would take nearly two times longer if the DAG were not used.

In the thin sphere distribution we start seeing some difference. Being a very non-uniform point distribution, flipping has more difficulty reaching the Delaunay triangulation, and thus more works remain to be done. Therefore, star splaying takes a bigger portion of the running time. Similarly, in the pathological case with points being cospherical, the adaptive star splaying takes nearly 25% of the total running time.

A pathological case for flipping in 3D triangulation would be when points are distributed on two non-parallel non-intersecting curves in 3D. In this case, not only is the number of flips required quadratic, but the number of iterations of flipping is also linear to the number of points, thus IncDel3D becomes much slower. However this case rarely happens in practice.

To further understand the behavior of our algorithm, we look at the number of stars participating in the adaptive star splaying in Phase 2. Figure 5.24 shows the number of stars constructed for the failed vertices, as well as the number of additional stars taken from the triangulation during the splaying. In this experiment we use 5×10^5 points for the synthetic inputs, and the Happy Buddha model, which has approximately the same number of points, as a real-world input for comparison. As can be seen, on the first three distributions, the number of stars initially constructed is very small, only about 600 stars, and less than 200 additional stars are involved in the splaying. On the thin sphere and the sphere distributions, significantly more stars are involved, but still less than 2% of the number of input points in total. The same applies to the real-world test case. Note that the sphere distribution is an extreme case. This shows that the flipping can get quite close to the Delaunay triangulation, and with our adaptive star splaying algorithm, not much more work is needed to obtain the final result.

5.4 Discussion

In this chapter, we have introduced a second approach to solve computational geometry problems on the GPU. Motivated by the traditional incremental insertion algorithm, our approach insert points in batches followed by parallel flipping to transform the structure into the desirable result. This approach is both lock-free and highly scalable, making it very suitable for the GPU.

The main difficulty of this approach is that performing local transformation in parallel may not always lead to the final solution. Except for the 2D Delaunay triangulation case, both the 3D convex hull and the 3D Delaunay triangulation have this problem. For the convex

hull case, we introduce a novel flipping strategy that combines both Delaunay flips and non-Delaunay flips to guarantee that we can always transform the initial structure into the convex hull. For the 3D Delaunay triangulation case, although we cannot modify the flipping algorithm to guarantee such result, we introduce an adaptive star splaying algorithm that can take over the result after the flipping is done, and quickly transform it into the Delaunay triangulation. This part of the computation is done on the CPU but it is very efficient, taking an insignificant amount of time in most cases. Also, several heuristics are introduced to reduce the number of flips required, as well as improving the result at the end of the flipping thus reducing the cost of fixing on the CPU even further.

Overall, the three algorithms provided in this chapter is an improvement over those presented in the previous chapter in many cases. The main advantage of the approach in this chapter is that its performance is not too much affected by the input point distribution. While using the digital approximation works very well for points that are uniformly distributed, as in the case of 2D Delaunay triangulation and 3D convex hull, it is no longer as efficient in the non-uniform case, especially in higher dimensions.

The second advantage here is that during the computation, the topology of the structure is always guaranteed to be correct. As such, the data-structure used can be much simpler and also requires less memory. As a result, for most cases we can handle larger input sizes.

Credits are given to several people for collaborating on the work in this chapter. Gao Mingcen contributed a major part on the proof of correctness of the Flip-Flop algorithm in Section 5.2; and Ashwin Nanjappa implemented the preliminary version of the 3D Delaunay triangulation algorithm in Section 5.3.

Chapter 6

Numerical Error and Robustness Issues

In practice, the input point set S might not be of general positions as we assumed in the beginning. There may be points on the same line, plane, circle or sphere. For these *degenerate* inputs, our implementations either explicitly handle it, as has been done in Section 4.3.4, or perturb the input points to avoid it.

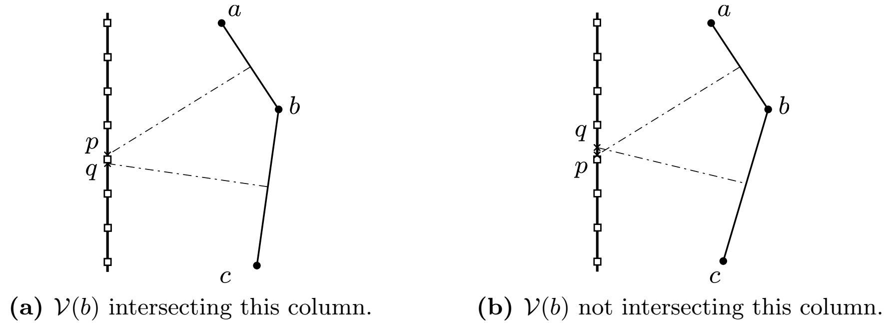
Worse still, the input points may not be exactly collinear for example, but very close to. Numerical error during floating point arithmetic leads to many troublesome problems and inconsistencies while handling degeneracy. In this chapter, we discuss how we handle these issues while keeping our implementation efficient on the GPU. Depending on the situation, certain numerical error might be avoided completely by not using floating point computation, or with some careful considerations. In other cases, we need to perform *exact arithmetic*, i.e. fully expanding the results of floating point computation.

In this chapter, we summarize the techniques that we use to handle numerical error and degeneracy for all the algorithms in this thesis. Besides, for robustness reason as well as for simplicity and thus efficiency of our implementation, we may need to introduce an *infinity point* into the point set. This implementation technique is discussed in the last section of this chapter.

6.1 Numerical error on digital Voronoi diagram computation

During Phase 2 of our PBA algorithm in Section 4.2, we need to use Lemma 4.2 to determine whether the Voronoi cell of a certain point b can appear on the column i or it is completely dominated by the Voronoi cells of a and c on this column. The decision relies on the computation of the two points $p(i, u)$ and $q(i, v)$ which are the intersection of the perpendicular bisector of $\overline{ab}$ and $\overline{bc}$ with column i. This computation can have numerical error due to a division, and as a result b might inaccurately color a grid point while it should not, or it misses a grid point that it should color; see Figure 6.1.

To avoid this numerical error problem, our implementation uses integer division instead of floating point division while calculating u and v. In other words, we perform the checking of



(a) $\mathcal{V}(b)$ intersecting this column. **(b)** $\mathcal{V}(b)$ not intersecting this column.

Figure 6.1: Numerical error while checking if a and c dominates b on the given column. It is difficult to confirm whether $\mathcal{V}(b)$ intersects the column or not.

Lemma 4.2 using $u' = \lfloor u \rfloor$ and $v' = \lfloor v \rfloor$, which can be computed exactly, instead of u and v. Clearly, if $u' < v'$ then $u < v$, and if $u' > v'$ then $u > v$. When $u' = v'$, both intersection points lie in between a pair of consecutive grid points on column i, and thus b cannot color any grid point in this column. In this case we say that b is dominated by a and c in the digital space of column i.

6.2 Transforming the point set in DigiDel2D

In the algorithm in Section 4.3, when working with points having floating point coordinates, Phase 1a needs to map them to an m^2 grid. We want precise computation so that the triangulation computed with respect to points mapped to the grid remains a triangulation when we do a part of the inverse mapping to the original coordinates of the points, as discussion in Section 4.3.3. In this section, we show how precise computation can be achieved by representing the scale and $\mathsf{translation}$ used in the mapping with only certain number of bits.

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

Then, Phase 2 shifts all points in the grid back to their positions given in the input. To maintain as many shifting of good cases as possible, we first perform the inverse scaling and shifting for the whole bounding box with all the points. Specifically, we have $x' = (\bar{x} \times \mathsf{scale} + \mathsf{translation})$ as the new coordinate of the point before shifting it to the original

coordinate x. The subsequent shifting is only to negate the effect of the truncation to integer coordinate. To ensure we still have a valid triangulation with x' in place of $\bar{x}$, we must be able to compute the floating point number x' with no rounding error.

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

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

Case 1: $\mathsf{translation} \leq \mathsf{range}$.

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

Case 2: $\mathsf{translation} > \mathsf{range}$.

We round up scale a little bit as follows. Let 2^r be the largest term in the binary representation of translation. We round up all terms that are smaller than $2^{r-(p\text{Max}-1)+pM}$ in the binary representation of scale. Because $\mathsf{range} = x_{\max} - x_{\min} \geq 2^{r-(p\text{Max}-1)}$, we always have that scale, represented by pS bits, is larger than $2^{r-(p\text{Max}-1)+pM}$ for any meaningful input and is thus non-zero. Also, the rounding up does not increase the value of scale more than twice, and thus we are still able to spread out the mapping of points on the grid.

6.3 Exact predicates and symbolic perturbation

All the algorithms we proposed relies on *predicates* such as orientation test, incircle test and insphere test. These tests are all of high order computations, and thus might have numerical error when using floating point computation. Numerical error is dangerous in that not only does it produce wrong output such as missing some points on the convex hull, but it can also give inconsistent result and thus the star splaying algorithm might fail to converge.

To deal with numerical error, we adopt Shewchuk's implementation of exact predicates

[She96b] on the GPU. Each exact predicate consists of two parts: the fast check and the exact check. In the fast check, computation is done purely using floating point arithmetic, and it uses *arithmetic filtering* to determine if the result is reliable or not. If the absolute value of result is greater than a certain error bound, then it is reliable. Otherwise, the exact check is called. The exact check represents each floating point number as an *expansion*, i.e. an array of floating point numbers in sorted magnitude. Each number is preallocated with the maximum size possibly required to represent it, depending on the computation that leads to it. This way, the computation is guaranteed to be exact.

Most of the threads performing the predicates do not need to go into exact computation. The error bound used in the filter is small enough that exact computation is required only when points are very close to being degenerate. This causes divergence in the thread execution. Besides, each exact computation requires a lot more temporary memory than the fast computation, and hence the number of threads that we can launch to perform exact computation is much smaller. If both the fast and the exact checks are done in the same kernel, then the fast checks are slowed down.

In order to reduce this divergence during predicate computation, we split each kernel that performs some predicates into two separate kernels, to be called one after the other. In the first kernel, all threads only use the fast checks. Threads that need to further use the exact checks will be marked. In the second kernel, only threads that are marked in the previous kernel perform the exact check. This way, the fast checks are done with as many threads as needed, and thus are very efficient.

One issue with this approach is that when launching the kernel to perform the exact check, very few threads actually have to do something. Most of the threads just check to find out they are not marked, and then stop. This significantly wastes the computing power of the GPU. To handle this, it is necessary to obtain a compressed list of threads (or more accurately the list of work items) that need exact checks. Performing compaction all the time is expensive, so we propose to use on-the-fly compaction on the shared memory. Threads in the same block do the compaction on the shared memory, and then one thread performs an AtomicAdd to get the global offset to output the compacted result to the global memory. Since the number of exact checks needed is usually very small, the access to the global memory is reduced.

Even with exact computation, we still have a robustness issue when input points are actually degenerate. For example, when we do point location on a 3D triangulation, if the point lies on a facet, then we still have to decide whether it should belong to one tetrahedron or the other. This decision has to be consistent across the program. The star splaying algorithm also requires consistent predicate results. To handle that, we use the simulation of simplicity method by Edelsbrunner [EM90]. This approach requires us to perform a few more determinant calculations in case of a degenerate point set. That just makes the threads

performing these predicate checks be more divergent. However, with our kernel splitting scheme mentioned above, this does not greatly affect the overall performance of our program.

6.4 Infinity point and the extended triangulation

In all the algorithms discussed in this thesis, one of the most common operations we need to perform is inserting points into an existing triangulation. The insertion is done in parallel, and it is important to make sure that these insertions do not introduce self-intersections in the triangulation. If the points being inserted fall into the existing simplices (triangles or tetrahedra), then the problem is simple since we just need to restrict at most one insertion per simplex. However, if the insertion falls outside the existing triangulation, then the situation is more troublesome. In that case, we use the technique discussed in Section 5.2, introducing a kernel point s and making sure that the boundary of the triangulation is always maintained star-shaped with respect to s. By doing so, the space outside the triangulation is partitioned into a set of none self-intersecting cones formed by the kernel point and the facets on the boundary of the triangulation.

To make the code for handling this boundary situation simpler, we introduce an *infinity point* into the triangulation. The infinity point is considered as a virtual vertex, connected to the boundary of the triangulation to form an *extended triangulation* covering the whole space (either $\mathbb{R}^2$ or $\mathbb{R}^3$). Besides the normal simplices, there is an extra set of infinity simplices adjacent to the boundary of the triangulation, with one vertex being the infinity point. These infinity simplices are treated as normal simplices, except during predicate computations. When an orientation check, incircle check, or insphere check involving the infinity point is performed, the coordinate of the kernel point s, as calculated below, is used instead, and the result is reversed accordingly.

The kernel point s is constructed, in the IncDel3D implementation for example, as follows. We use a parallel scan operation on the GPU to find the two points a and b in S with the minimum and maximum x-coordinates. Then, we use another scan to find the point c that is farthest from the edge ab, and a last scan to find the point d that is farthest from the triangle abc. By doing so, we get an initial tetrahedron $abcd$ that has a large volume in most cases. The point s is calculated to be the centroid of the tetrahedron $abcd$. Note that we also perform an orientation check on $abcd$ and if the predicate result is 0 (i.e. these points are coplanar), the program terminates since the whole input point set is coplanar.

CHAPTER 7

Concluding Remarks

In this thesis, we have proposed two approaches to solve three fundamental computational geometry problems on the GPU. In the first approach, a major part of the computation is performed in the digital space to obtain a good approximation, before a transformation phase is applied to obtain the final result in the continuous space. The digital approximation is derived from the digital Voronoi diagram which can be efficiently constructed using our Parallel Banding Algorithm. The advantage of this approach is that a major part of the computation is done in the digital space with high level of parallelism and a good memory access pattern. This approach works particularly well in 2D when we construct the Delaunay triangulation, since we can provably derive a valid triangulation from the digital computation.

For the higher dimensions problems, namely the 3D convex hull and the 3D Delaunay triangulation, we encounter some issues. The dualization becomes more challenging and the result is in general not topologically correct. We adopt the star splaying algorithm on the GPU to efficiently correct these problems. For the 3D convex hull problem, we also introduce the digital depth test to further perform more work in the digital space while still guaranteeing the correctness of the algorithm in the continuous space. For the 3D Delaunay triangulation problem, we show that it is possible to have a digital structure that can be dualized into a valid 3D triangulation, but obtaining that structure is too costly, even for the GPU. As such, an approach similar to that for the 3D convex hull problem is used. Nevertheless, the approach still allows us to, for the first time, achieve significant speedup when compared to the best sequential CPU implementations currently available, for all the three problems.

Moving on to the second approach, we depart from the digital computation approach and instead try to rely on a combination of the incremental insertion technique and local transformations. Points are inserted in batches with the help of some heuristics to provide a good triangulation before flipping is applied in parallel to transform it to the correct result. In the 2D Delaunay triangulation problem, this approach works very well. In the 3D convex hull problem, we need a slightly modified flipping schedule to guarantee that flipping provably converges to the solution. In the 3D Delaunay triangulation problem, flipping can no longer guarantee us correct result, and actually the existence of a flipping path to transform any

triangulation to the Delaunay one is an open problem. Nevertheless, flipping usually takes us very close to the result, and with our adaptive version of star splaying on the CPU, we can obtain the final result efficiently. We also further provide two heuristics, one on choosing the order of point insertion, and one on alternating between inserting points and flipping, to further reduce the cost of flipping as well as the cost of the subsequent star splaying phase. Overall, this approach allows us to reach a speedup similar to the first approach, if not a bit better. More importantly, this approach is not much dependent on the distribution of the input point set as compared to the first approach, and thus it can handle real-world input data very well.

7.1 Limitations

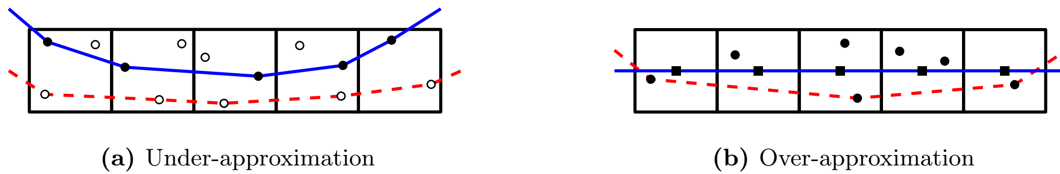
From a high level point of view, there is a fundamental difference between the digital approach and the incremental insertion approach. In the digital approach, through the use of the digital Voronoi diagram to approximate the continuous one, we can capture the geometry of the desired result approximately, while we sometimes fail to capture the topology. As a result, in the 3D convex hull and the 3D Delaunay triangulation problem, it is necessary for us to use the star splaying algorithm to correct the topology of the derived sketch. The only scenario in which the digital computation produces a sketch with correct topology is when constructing the 2D Delaunay triangulation, although the proof for that is rather intricate.

The continuous approach, on the other hand, focuses on maintaining the topology of the structure at all time, while only uses some heuristics to partially capture the geometry. For example, we insert at most one point into a simplex at a time, while flips that cause self-intersections are prohibited. As a result, the implementation of this approach is usually much simpler. However, because the geometry is not well captured, more transformations are needed to get to the final result. In some cases, due to the restriction of the topology, it may not be possible to reach, for example, the 3D Delaunay triangulation. In that case, we still need the star splaying algorithm to handle the few last modifications to the structure under construction. The star splaying algorithm actually disintegrates the existing structure into smaller components (stars) to reduce the topological constraints in order to transform it into the correct result.

In the two following sections, we discuss a few other limitations of the two proposed approach in practice.

7.1.1 The digital approach

Due to the nature of the digital space with finite resolutions, it is difficult to capture all the geometrical information. As a result, there are two cases in which the information in



(a) Under-approximation

(b) Over-approximation

Figure 7.1: Two problems associated with the input points being shifted in digital space.

the continuous space is missed in the digital space. The first case is when multiple input points are mapped to the same grid point. In this case, only one point can be captured in the digital computation, while the rest need to be processed later. As a result, when the input points are cluttered, especially on real-world inputs where points are arranged along some curves or surfaces instead of spreading evenly in the space, many points are missed in the digital computation. The second case is when multiple Voronoi vertices fall into the same grid cell. Again, only one can be captured, while others are either shifted to nearby grid cells or are lost, causing topological problems in the sketch. This is clearly observed when experimenting with the thin circle or the thin sphere point distributions.

Another issue that can affect the quality of the sketch, and thus the performance of our algorithms, is the shifting of input points. When we map the points into the digital space, we shift them slightly. In the convex hull construction described in Section 4.4 for example, this shifting causes two following problems:

Under-approximation problem.

When we have multiple points shifted to the same grid point, we can only record one point, and thus there are potentially many more points outside $\mathcal{C}(S')$. See Figure 7.1a for a 2D illustration where the round black points are kept and the solid line denotes part of $\mathcal{C}(S')$. The round white points are missing points, many of which are outside $\mathcal{C}(S')$. We address this issue by an efficient depth test in Section 4.4.4 and with the walking to locate a nearby triangle for every point outside $\mathcal{C}(S')$, we are able to construct a very good star for that point. This reduces the amount of splaying needed in the next phase.

Over-approximation problem.

In certain cases, for example when points are distributed near the surface of a cube axis-aligned with $\mathcal{G}$, many points that are not extreme vertices are shifted outside and are legitimately captured in the sketch. See Figure 7.1b for a 2D illustration, where after the digital Voronoi diagram computation, all the round black points, after shifted to the square black grid points, are captured. To address this issue, when computing each slice of the digital Voronoi diagram, we only shift two coordinates of the points while keeping the third one untouched (i.e. using floating value). This gives us a much more accurate sketch.

7.1.2 The incremental insertion approach

The incremental insertion with local transformation approach helps avoid many issues associated with the digital computation. However, it still has several limitations that potentially affect the performance of our algorithms.

Firstly, the approach makes heavy use of flipping. Since incremental construction does not provide a good approximation of the desired structure, a lot of flips are needed. Each flip requires memory access to check, perform, and update the neighboring information. These accesses are usually random, and that puts a lot of pressure onto the GPU memory system. Section 5.3.5 has mentioned some techniques to improve the situation, but in general most of our implementations are still memory bound.

Secondly, during the incremental insertion process, at most one point can be inserted into a simplex in each round. In the worst case where after each insertion, all the remaining points fall into the same simplex, we need $O(n)$ rounds of insertion. Fortunately, this is usually not the case in practice, since with our heuristic in choosing the order of points to insert, the remaining points are spread quite evenly among the existing simplices (even on non-uniform point distributions or real-world datasets).

Thirdly, we do not have any clear control over the number of iterations of flipping. Here we are not referring to those pathological cases in which $O(n^2)$ flips are needed. Instead, we are talking about some practical situations in which the flips are serialized. We illustrate this scenario with an example in 2D convex hulls construct, in which each flip is a 2–1 flip to remove a non-extreme vertex; see Figure 7.2. In this example, at any point in time, only one flip is available, and only after that flip is performed does another flip appear. A very similar scenario can happen during 3D convex hull and Delaunay triangulation construction. In that case, the number of iterations of flipping can be very large, with very few flips in each iteration. This leads to a lot of synchronizations between the CPU and the GPU, affecting the performance of the algorithms.

The 3D Delaunay triangulation algorithm has another limitation, which is on the unpredictability of the number of unflippable facets remaining after Phase 1. This number is

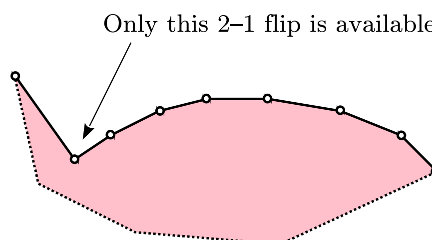


Figure 7.2: An illustration of a situation where flipping is serialized.

affected by the order of points that we insert, and the order of flips performed. In our experiment with synthetic data, this number is usually quite stable, but with real-world data, this number varies significantly. Fortunately our adaptive star splaying algorithm, combined with the use of Flip-Flop in the construction of convex stars, is quite efficient at removing these unflippable facets. It remains a challenging problem to improve the situation, and if possible reaching the Delaunay triangulation through flipping alone.

7.2 Outlook

The last few years have seen many new techniques developed for programming on the GPU. We look into some of the techniques that could change or be used in the algorithms we propose in this thesis.

In the Tao analysis [PNK+11], our algorithms are classified as *morph* algorithms, because the computation repeatedly modifies the connectivity in the underlying structure. A series of papers by the same group of authors [HBP11, NBP13a, NBP13b, NBP13c] presents several techniques for implementing this class of algorithms on the GPU. Many of these techniques have already been independently developed for this thesis. These works also emphasize on the need for better ways to rearrange the data to improve cache efficiency. Wu *et al.* [WZZ+13] prove that the general problem of re-positioning data to minimize non-coalesced memory access on the GPU is NP-complete. They also discuss a few techniques to rearrange data, some of which are also very similar to those we propose.

Besides the memory access problem, the overhead of synchronizing between the GPU and the CPU is also a serious issue. Particularly, our algorithms are all developed in a multi-iterations style. This results in a large number of kernel launches and GPU-CPU communications. The IncDel3D algorithm, for example, requires a few hundred iterations of flipping in total, most of which performing very few actual flips. A recent work by Gupta *et al.* [GSO12] advocates for the use of Persistent Threads, a programming principle in which a kernel is launched with only enough threads so that all of them reside and execute on the GPU at the same time. This allows us to globally synchronize the execution of threads across all blocks, and thus a single kernel launch can perform multiple iterations of computation without the need to stop and exit to the CPU. Such a global synchronization mechanism is also discussed by Xiao and Feng [XF10]. They successfully design a scalable and decentralized global barrier without using atomic operations. The persistent threads style could be useful to reduce the overhead in our implementations.

Recently, Stuart and Owens [SO11] propose several techniques to implement mutex and semaphore efficiently on the GPU. These are two important synchronization primitives that can be used to lock data while processing in parallel. This possibly allows us to use the Bowyer-Watson algorithm on the GPU, similar to the work by Batista *et al.* [BMPS10]. In

our preliminary experiment with the parallel Bowyer-Watson algorithm on the GPU, we find that with an input of one million points, we need more than 400 rounds of insertions, since each round can only insert at most a few thousand points. This is because most of the insertions conflict with each other since the regions affected by each insertion overlap. As a result, even with locking possible and using the persistent threads programming style, it would still be challenging to implement this approach while fully utilize the parallel computing power on the GPU. With the fast growing number of stream processors, such a coarse-grained approach is definitely less favorable compared to the two approaches we present in this thesis.

References

[AK12] Almashor, M. and Khalil, I. Fully peer-to-peer virtual environments with 3D Voronoi diagrams. *Computing*, 94(8–10):679–700, 2012.

[AP93] Amato, N. M. and Preparata, F. P. An NC parallel 3D convex hull algorithm. In *SCG '93: proceedings of the 9th symposium on Computational geometry*, pages 289–297, New York, NY, USA, 1993. ACM.

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

[BCKO08] Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. *Computational geometry: algorithms and applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[BDH96] Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. The Quickhull algorithm for convex hulls. *ACM transactions on Mathematical Software*, 22(4):469–483, 1996. http://www.qhull.org/.

[BMPS10] Batista, V. H., Millman, D. L., Pion, S., and Singler, J. Parallel geometric algorithms for multi-core computers. *Computational geometry*, 43(8):663–677, 2010.

[Bol09] Bollig, E. F. Centroidal Voronoi tesselation of manifolds using the GPU. Master's thesis, Department of Scientific Computing, Florida State University, 2009.

[Bor48] Borsuk, K. On the imbedding of systems of compacta in simplicial complexes. *Fundamenta Mathematicae*, 35(1):217–234, 1948.

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

[CET14] Cao, T.-T., Edelsbrunner, H., and Tan, T.-S. Triangulations from topologically correct digital Voronoi diagrams. *Computational Geometry: Theory and Applications*, 2014. To appear.

[CGA] CGAL, Computational Geometry Algorithms Library, v4.2. http://www.cgal.org.

[Cha96] Chan, T. M. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.

[CK07] Cuntz, N. and Kolb, A. Fast hierarchical 3D distance transforms on the GPU. In *Eurographics*, pages 93–96, 2007.

[CMS92] Cignoni, P., Montani, C., and Scopigno, R. A merge-first divide & conquer algorithm for E^d Delaunay triangulations. Technical report, Istituto CNUCE - C.N.R., Pisa, Italy, 1992.

[CNGT14] Cao, T.-T., Nanjappa, A., Gao, M., and Tan, T.-S. A GPU accelerated algorithm for 3D Delaunay triangulation. In *I3D '14: Proceedings of the 2014 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 47–54, New York, NY, USA, 2014. ACM.

[CS88] Clarkson, K. L. and Shor, P. W. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *SCG '88: Proceedings of the 4th symposium on Computational Geometry*, pages 12–17, New York, NY, USA, 1988. ACM.

[CTMT10] Cao, T.-T., Tang, K., Mohamed, A., and Tan, T.-S. Parallel banding algorithm to compute exact distance transform with the GPU. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90, New York, NY, USA, 2010. ACM.

[Cui99] Cuisenaire, O. *Distance transformations: fast algorithms and applications to medical image processing*. PhD thesis, Universite catholique de Louvain (UCL), Louvain-la-Neuve, Belgium, 1999.

[Dan80] Danielsson, P.-E. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[DDD+95] Dehne, F., Deng, X., Dymond, P., Fabri, A., and Khokhar, A. A. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *SPAA '95: proceedings of the 7th ACM symposium on Parallel Algorithms and Architectures*, pages 27–33, New York, NY, USA, 1995. ACM.

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

[Dwy91] Dwyer, R. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6(1):343–367, 1991.

[EK12] Edelsbrunner, H. and Kerber, M. Dual complexes of cubical subdivisions of R^n. *Discrete & Computational Geometry*, 47(2):393–414, 2012.

[EM90] Edelsbrunner, H. and Mücke, E. P. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[Eri99] Erickson, J. Computational geometry pages, list of software libraries and codes. http://compgeom.cs.uiuc.edu/~jeffe/compgeom/code.html, 1999.

[FC12] Foteinos, P. and Chrisochoides, N. Dynamic parallel 3D Delaunay triangulation. In Quadros, W. R., editor, *Proceedings of the 20th International Meshing Roundtable*, pages 9–26, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[FCTB08] Fabbri, R., Costa, L. D. F., Torelli, J. C., and Bruno, O. M. 2D Euclidean distance transform algorithms: A comparative survey. *ACM computing survey*, 40(1):1–44, 2008.

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

[For87] Fortune, S. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1–4):153–174, 1987.

[GCN+13] Gao, M., Cao, T.-T., Nanjappa, A., Tan, T.-S., and Huang, Z. gHull: a GPU algorithm for 3D convex hull. *ACM transactions on Mathematical Software*, 40(1):3:1–3:19, October 2013.

[GCTH13] Gao, M., Cao, T.-T., Tan, T.-S., and Huang, Z. Flip-flop: convex hull construction via star-shaped polyhedron in 3D. In *I3D '13: proceedings of the 2013 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 45–54, New York, NY, USA, 2013. ACM.

[Geo] The Georgia Tech large geometric models archive. http://www.cc.gatech.edu/projects/large_models/.

[GP10] Guillemin, V. and Pollack, A. *Differential Topology*. American Mathematical Society, Providence, Rhode Island, 2010.

[Gra72] Graham, R. L. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[GS85] Guibas, L. and Stolfi, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[GSO12] Gupta, K., Stuart, J. A., and Owens, J. D. A study of persistent threads style GPU programming for GPGPU workloads. In *InPar '12: Innovative Parallel Computing*, pages 1–14, May 2012.

[HBP11] Hassaan, M. A., Burtscher, M., and Pingali, K. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *PPoPP '11: Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, pages 3–12, New York, NY, USA, 2011. ACM.

[HDSB01] Huebner, K. H., Dewhirst, D. L., Smith, D. E., and Byrom, T. G. *The Finite Element Method for Engineers*. Wiley, New York, NY, USA, 2001.

[HH06] Hahn, H. and Han, Y. Recognition of 3D object using attributed relation graph of silhouette's extended convex hull. In *Advances in Visual Computing*, volume 4292 of *Lecture Notes in Computer Science*, pages 126–135. Springer Berlin Heidelberg, 2006.

[Hig82] Highnam, P. T. The ears of a polygon. *Information Processing Letters*, 15(5):196–198, 1982.

[HKL+99] Hoff, III, K. E., Keyser, J., Lin, M., Manocha, D., and Culver, T. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[HNO98] Hayashi, T., Nakano, K., and Olariu, S. Optimal parallel algorithms for finding proximate points, with applications. *IEEE transactions on Parallel and Distributed Systems*, 9(12):1153–1166, 1998.

[HSO07] Harris, M., Sengupta, S., and Owens, J. D. Parallel prefix sum (scan) with cuda. In Nguyen, H., editor, *GPU Gems 3*, pages 815–876. Addison Wesley, 2007.

[ILSS06] Isenburg, M., Liu, Y., Shewchuk, J., and Snoeyink, J. Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056, July 2006.

[JBS06] Jones, M. W., Baerentzen, J. A., and Sramek, M. 3D distance fields: A survey of techniques and applications. *IEEE transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[JD10] Jurkiewicz, T. and Danilewski, P. Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations. http://www.mpi-inf.mpg.de/~tojot/papers/chull.pdf, October 2010.

[Joe89] Joe, B. Three-dimensional triangulations from local transformations. *SIAM journal on Scientific and Statistical Computing*, 10(4):718, 1989.

[Joe91] Joe, B. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer aided geometric design*, 8(2):123–142, 1991.

[Kal81] Kallay, M. Convex hull algorithms in higher dimensions. Unpublished manuscript, department of mathematics, university of Oklahoma, Norman, Oklahoma, 1981.

[KK92] Kolountzakis, M. N. and Kutulakos, K. N. Fast computation of the Euclidean distance maps for binary images. *Information processing letters*, 43:181–184, 1992.

[KKv05] Kohout, J., Kolingerová, I., and Žára, J. Parallel Delaunay triangulation in E2 and E3 for computers with shared memory. *Parallel computing*, 31(5):491–522, 2005.

[Kre97] Kreveld, M. Algorithms for triangulated terrains. In *SOFSEM'97: Theory and Practice of Informatics*, volume 1338 of *Lecture Notes in Computer Science*, pages 19–36. Springer Berlin Heidelberg, 1997.

[KS86] Kirkpatrick, D. G. and Seidel, R. The ultimate planar convex hull algorithm. *SIAM Journal on Computing*, 15(1):287–299, February 1986.

[Law77] Lawson, C. L. Software for C^1 surface interpolation. In Rice, J. R., editor, *Mathematical Software III*, pages 161–194. Academic Press, 1977.

[LHS03] Lee, Y.-H., Horng, S.-J., and Seitzer, J. Parallel computation of the Euclidean distance transform on a three-dimensional image array. *IEEE transactions on Parallel and Distributed Systems*, 14(3):203–212, 2003.

[LKS+10] Lee, J., Kim, J., Seo, S., Kim, S., Park, J., Kim, H., Dao, T. T., Cho, Y., Seo, S. J., Lee, S. H., Cho, S. M., Song, H. J., Suh, S.-B., and Choi, J.-D. An OpenCL framework for heterogeneous multicores with local memory. In *PACT '10: Proceedings of the 19th international conference on Parallel Architectures and Compilation Techniques*, pages 193–204, New York, NY, USA, 2010. ACM.

[Llo82] Lloyd, S. P. Least squares quantization in PCM. *IEEE transactions on Information Theory*, 28(2):129–137, 1982.

[Lo12] Lo, S. H. Parallel Delaunay triangulation in three dimensions. *Computer Methods in Applied Mechanics and Engineering*, 237-240:88–106, September 2012.

[LZB08] Liu, R., Zhang, H., and Busby, J. Convex hull covering of polygonal scenes for accurate collision detection in games. In *GI '08: Proceedings of Graphics Interface*, pages 203–210, Windsor, Canada, 2008.

[McL76] McLain, D. H. Two dimensional interpolation from random data. *The computer journal*, 19(2):178–181, 1976.

[MQR03] Maurer, Jr., C. R., Qi, R., and Raghavan, V. A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 25(2):265–270, 2003.

[MS88] Miller, R. and Stout, Q. F. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37(12):1605–1618, 1988.

[MS97] Meeran, S. and Share, A. Optimum path planning using convex hull and local search heuristic algorithms. *Mechatronics*, 7(8):737–756, 1997.

[NBGS08] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[NBP13a] Nasre, R., Burtscher, M., and Pingali, K. Atomic-free irregular computations on gpus. In *Proceedings of the 6th workshop on General Purpose Processor using Graphics Processing Units*, pages 96–107, New York, NY, USA, 2013. ACM.

[NBP13b] Nasre, R., Burtscher, M., and Pingali, K. Data-driven versus topology-driven irregular computations on GPUs. In *IPDPS '13: Proceedings of the 27th IEEE international symposium on Parallel and Distributed Processing*, pages 463–474, May 2013.

[NBP13c] Nasre, R., Burtscher, M., and Pingali, K. Morph algorithms on GPUs. In *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 147–156, New York, NY, USA, 2013. ACM.

[OLG+07] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., KrÃijger, J., Lefohn, A. E., and Purcell, T. A survey of general-purpose computation on graphics hardware. *Computer graphics forum*, 26(1):80–113, 2007.

[PH77] Preparata, F. P. and Hong, S. J. Convex hulls of finite sets of points in two and three dimensions. *Communication of ACM*, 20(2):87–93, 1977.

[PNK+11] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. The tao of parallelism in algorithms. In *PLDI '11: Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 12–25, New York, NY, USA, 2011. ACM.

[Pri] The Princeton suggestive contour library. http://gfx.cs.princeton.edu/proj/sugcon/models/.

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

[QCT13] Qi, M., Cao, T.-T., and Tan, T.-S. Computing 2D constrained Delaunay triangulation using the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 19(5):736–748, May 2013.

[RT06] Rong, G. and Tan, T.-S. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *I3D '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Interactive 3D graphics and games*, pages 109–116, New York, NY, USA, 2006. ACM.

[Sec02] Secord, A. Weighted Voronoi stippling. In *NPAR '02: proceedings of the 2nd international symposium on Non-photorealistic Animation and Rendering*, pages 37–43, New York, NY, USA, 2002. ACM.

[SGES12] Stein, A., Geva, E., and El-Sana, J. CudaHull: Fast parallel 3D convex hull on the GPU. *Computers & Graphics*, 36(4):265–271, 2012.

[SGGM06] Sud, A., Govindaraju, N., Gayle, R., and Manocha, D. Interactive 3D distance field computation using linear factorization. In *I3D '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 117–124, New York, NY, USA, 2006. ACM.

[SGM05] Sud, A., Govindaraju, N., and Manocha, D. Interactive computation of discrete generalized Voronoi diagrams using range culling. In *Proceedings of the international symposium on Voronoi diagrams in science and engineering*, 2005.

[SH75] Shamos, M. I. and Hoey, D. Closest-point problems. *FOCS '75: proceedings of the 16th annual symposium on Foundations of Computer Science*, pages 151–162, 1975.

[She96a] Shewchuk, J. R. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Lin, M. and Manocha, D., editors, *Applied Computational Geometry towards geometric engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer Berlin / Heidelberg, 1996.

[She96b] Shewchuk, J. R. Robust adaptive floating-point geometric predicates. In *SoCG '96: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 141–150, New York, NY, USA, 1996. ACM.

[She05] Shewchuk, J. R. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In *SoCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 237–246, New York, NY, USA, 2005. ACM.

[SKW09] Schneider, J., Kraus, M., and Westermann, R. GPU-based real-time discrete Euclidean distance transforms with precise error bounds. In *VISAPP '09: International conference on Computer Vision Theory and Applications*, pages 435–442, 2009.

[SO11] Stuart, J. A. and Owens, J. D. Efficient synchronization primitives for GPUs. *CoRR*, abs/1110.4623, 2011.

[SOM04] Sud, A., Otaduy, M. A., and Manocha, D. DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum*, 23(3):557–566, 2004.

[SPG03] Sigg, C., Peikert, R., and Gross, M. Signed distance transform using graphics hardware. In *VIS'03: Proceedings of the 14th IEEE Visualization 2003*, pages 12–19, Washington, DC, USA, 2003. IEEE Computer Society.

[SRKN11] Srungarapu, S., Reddy, D. P., Kothapalli, K., and Narayanan, P. J. Fast two dimensional convex hull on the GPU. In *Proceedings of the 2011 IEEE workshops of international conference on Advanced Information Networking and Applications*, WAINA '11, pages 7–12, Washington, DC, USA, 2011. IEEE Computer Society.

[Sta] The Stanford 3D scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[TO12] Tzeng, S. and Owens, J. D. Finding convex hulls using Quickhull on the GPU. *CoRR*, abs/1201.2936, 2012.

[Tum04] Tumbde, A. A Voronoi partitioning approach to support massively multiplayer online games. Technical report, The University of Wisconsin, 2004.

[TZTM12] Tang, M., Zhao, J.-Y., Tong, R., and Manocha, D. GPU accelerated convex hull computation. *Computers & Graphics*, 36(5):498–506, 2012.

[VSCG08] Vasconcelos, C. N., Sá, A., Carvalho, P. C., and Gattass, M. Lloyd's algorithm on GPU. In *ISVC '08: proceedings of the 4th international symposium on Advances in Visual Computing*, pages 953–964, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[WHLL01] Wang, Y.-R., Horng, S.-J., Lee, Y.-H., and Lee, P.-Z. Optimal parallel algorithms for the 3D Euclidean distance transform on the CRCW and EREW PRAM models. In *Proceedings of the 19th workshop on Combinatorial Mathematics and Computation Theory*, Taiwan, 2001.

[WLYZ+09] Wang, Y., Ling-Yun, W., Zhang, J.-H., Zhan, Z.-W., Xiang-Sun, Z., and Luonan, C. Evaluating protein similarity from coarse structures. *IEEE/ACM transactions on Computational Biology and Bioinformatics*, 6(4):583–593, 2009.

[WZZ+13] Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., and Shen, X. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 57–68, New York, NY, USA, 2013. ACM.

[XF10] Xiao, S. and Feng, W.-C. Inter-block GPU communication via fast barrier synchronization. In *IPDPS '10: Proceedings of the 24th IEEE international symposium on Parallel and Distributed Processing*, pages 1–12, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

List of publications

1. **Thanh-Tung Cao**, Ke Tang, Mohamed Anis and Tiow-Seng Tan. Parallel Banding Algorithm to compute exact distance transform with the GPU. In *I3D '10: Proceedings of the 2010 ACM symposium on Interactive 3D Graphics and Games*, 83–90, 2010.
2. Mingcen Gao, **Thanh-Tung Cao**, Tiow-Seng Tan, and Zhiyong Huang. gHull – A three dimensional convex hull algorithm for graphics hardware. In *I3D '11: Proceedings of the 2011 ACM symposium on Interactive 3D Graphics and Games*, 204, 2011. Poster.
3. Meng Qi, **Thanh-Tung Cao**, and Tiow-Seng Tan. Computing 2D constrained Delaunay triangulation using the GPU. In *I3D '12: Proceedings of the 2012 ACM symposium on Interactive 3D Graphics and Games*, 39–46, 2012.
4. Meng Qi, **Thanh-Tung Cao**, and Tiow-Seng Tan. Computing 2D constrained Delaunay triangulation using the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 19(5):736–748, 2013.
5. Mingcen Gao, **Thanh-Tung Cao**, and Tiow-Seng Tan. Flip-Flop: Convex hull construction via star-shaped polyhedron in 3D. In *I3D '13: Proceedings of the 2013 ACM symposium on Interactive 3D Graphics and Games*, 45–54, 2013.
6. Mingcen Gao, **Thanh-Tung Cao**, Ashwin Nanjappa and Tiow-Seng Tan. gHull – A GPU algorithm for 3D convex hull. *ACM Transactions on Mathematical Software*, 40(1):3, 2013.
7. **Thanh-Tung Cao**, Ashwin Nanjappa, Mingcen Gao and Tiow-Seng Tan. A GPU accelerated algorithm for 3D Delaunay triangulation. In *I3D '14: Proceedings of the 2014 ACM symposium on Interactive 3D Graphics and Games*, 47–54, 2014.
8. **Thanh-Tung Cao**, Herbert Edelsbrunner, and Tiow-Seng Tan. Triangulations from topologically correct digital Voronoi diagrams. *Computational Geometry: Theory and Applications*, 2014. To appear.