CONSTRUCTING REGULAR TRIANGULATION VIA LOCAL TRANSFORMATIONS: THEORETICAL AND PRACTICAL ADVANCES

MINGCEN GAO

NATIONAL UNIVERSITY OF SINGAPORE

2015

CONSTRUCTING REGULAR TRIANGULATION VIA LOCAL TRANSFORMATIONS: THEORETICAL AND PRACTICAL ADVANCES

MINGCEN GAO

B.Eng., Northeastern University (China), 2009

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2015

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.

Mingcen Gao
February 10, 2015

Acknowledgment

I sincerely thank my supervisor Tan Tiow Seng, who guided me to computational geometry. Tiow Seng's critical thinking, patient encouragement, and broad research network are invaluable to me. His hard working inspires me to learn hard and improve my skill continuously. Also I really appreciate the freedom of doing the research that I am interested in during these years.

I would like to thank Low Kok-Lim for his courses on GPU programming and on computer graphics. I learned a lot from his class and laid a good foundation in GPU programming.

I want to thank Cheng Ho-lun for his computational geometry course, where I found the interest of research on computational geometry and determined to learn more in this area.

I would like to thank Huang Zhiyong for the opportunity of working with him as a teaching assistant. I learned to use various 3D modelling tools during that time.

I would like to thank Cheng Siu-Wing for inviting me to do an internship in Hong Kong University of Science and Technology. It was a great pleasure to discuss with Siu-Wing about the local transformation problems in computation geometry.

I would like to thank Herbert Edelsbrunner for the inspiring discussion on various geometric problems and the valuable suggestions on this thesis. I am grateful to Herbert for giving me the chance to visit the Institute of Science and Technology Austria.

I would like to thank Leong Hon Wai for his valuable comments on my thesis.

I want to thank my colleagues in G3 lab. I am grateful to Cao Thanh-Tung for the continuous help and collaboration. His optimism encouraged me a lot to overcome the research difficulties during my Ph.D. I like to thank Ashwin Nanjappa and Qi Meng for their discussion and moral support to reduce my stress of being a graduate student.

February 10, 2015

To Mom, Dad and Wei

Contents

Summary

This thesis studies transformation between geometric structures via geometric operations that act on some simplices of a geometric structure. Such a transformation is termed local transformation as an operation utilizes only information local to some neighboring simplices but nothing about other attributes or configurations global in nature to the structure. One famous algorithm of local transformation is Lawson's flip algorithm to construct the Delaunay triangulation from an arbitrary 2D triangulation. Local transformation is simple to be implemented in practice and has been shown to be powerful and efficient to transform among various important fundamental geometric structures. Such a transformation is also useful to repair geometric structures due to small adjustment to their simplices. For today's many-core architecture such as that of the GPU, local transformation is particularly attractive if it can be executed in parallel to gain good speedup at a low cost.

Besides Lawson's flip algorithm, this thesis investigates Shewchuk's star splaying algorithm, and presents the flip-flop algorithm and the twist algorithm in relation to constructing convex hull and regular triangulation, two structures that are geometrically related to Delaunay triangulation.

The thesis starts with a simple and yet non-trivial problem: 2D convex hull. We design a parallel version of the famous Graham's scan, re-casting its scanning stage as a series of flip operations from a star-shaped polygon to the resulting convex hull. In order to avoid the numerical inexactness in constructing star-shaped polygon of the original Graham's scan, we construct the upper and the lower half of the convex hull separately from two star-shaped chains. Such a novel approach is easily realized with the GPU to gain more than 40 times speedup over well-known convex hull implementations.

Next, the thesis examines local transformation to compute Delaunay triangulation, and its generalization of regular triangulation. It is known that one can incrementally construct such triangulation using flips by adding one vertex at a time. That approach, however, strictly alternates between one operation of inserting a vertex and a series of flips to regular triangulation. This dependency of operations limits its use in parallel computation. We discover a novel flip algorithm called flip-flop without such deficiency. This algorithm allows non-restrictive insertion of many vertices into an arbitrary 2D triangulation before transforming it to the regular triangulation. This can also be used to construct the convex hull from a 3D star-shaped polyhedron. Such an approach implemented on the GPU has up to 50 times speedup over existing CPU algorithms.

The thesis continues with examining star splaying, another local transformation that is used to repair convex hull. We propose the gHull algorithm that uses star

splaying to construct the 3D convex hull of a point set, and evaluate it against the flip-flop algorithm. It is noted that star splaying is more efficient when its input is close to the convex hull. Along this line, we exploit the relation between Voronoi diagram and convex hull to build an approximation from the restricted digital Voronoi diagram before employing star splaying. As a result, gHull runs very well on the GPU with up to 30 times faster than the best CPU implementation. However, gHull still does not perform as well as flip-flop especially for points in non-uniform distributions.

Understanding from the above studies, we note that star-shaped polytope plays an important role as the only legitimate input for flipping to 2D and 3D convex hull. The thesis then investigates the possibility of obtaining a star-shape polytope from an arbitrary, possibly self-intersecting, one through local transformation. Towards this end, in the 2D setting, we design a novel twist algorithm to transform an arbitrary polygon to a star-shaped one w.r.t. any chosen point inside its convex hull. In the 3D setting, we discover that a series of flip and twist operations can reach the goal for the special case of a polyhedron that has an extreme vertex connecting to all the other vertices. As for general polyhedrons, the problem remains open.

Excited by the discovery of flip-flop, one wonders what other compromises to the traditional flip algorithm, which is hill-climbing in nature, are possible to compute regular triangulation and convex hull in higher dimensions. The thesis tackles this question by investigating the local transformation from one regular triangulation to another of the same point set in arbitrary dimensions. We discover interesting ways to characterize flipping when the triangulation is cast into a time line. These might be useful to solve the open problem of finding a flip algorithm, free of restrictively sequential execution order, to derive from one regular triangulation to another.

List of Tables

List of Figures

CHAPTER 1

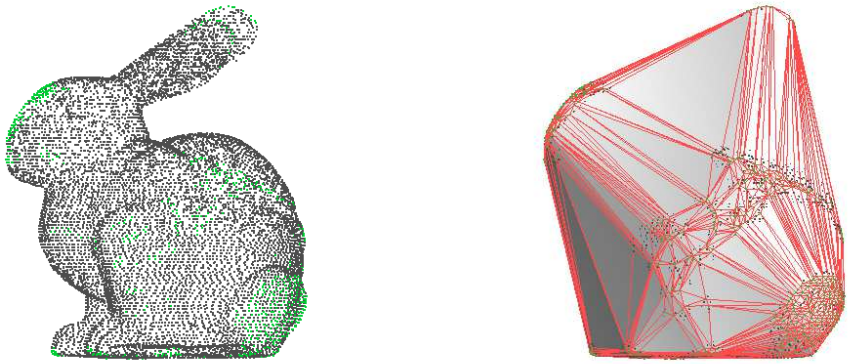
Introduction

Convex hull and regular triangulation are related basic geometric structures. Their usefulness has been shown in many industrial and scientific applications. In the past four decades, many algorithms have been developed to compute them efficiently. These algorithms can be categorized into two classes based on their input being either a point set or an existing geometric structure. This thesis focuses on the latter class, and more specifically on computing convex hull and regular triangulation using local transformation.

Local transformation is a procedure of performing a series of local operations on simplices of a given geometric structure. Each operation works on a few neighboring simplices and uses only information about these simplices and possibly their neighborhood, without considering any global configuration of the structure. One well-known such operation is the flip operation used in Lawson's flip algorithm [Law72] to compute Delaunay triangulation, a specialized regular triangulation. This thesis studies existing local transformations for their nice properties, and then goes beyond them in searching for new ones.

By nature, local transformation employs simple operations and is more likely amendable to parallel computation. It thus has great potential to be mapped onto massively-multithreaded architecture. We are therefore particularly interested in developing algorithms of local transformation for the GPU, the low cost hardware with the advances for its computational capability in solving large scale problems, to gain orders of magnitude speedup in comparison to traditional CPU implementations.

This chapter reviews convex hull, regular triangulation, and local transformation in detail and particularly in relation to GPU computation. It ends with a summary of the main results in this thesis.



(a) (b)

Figure 1.1: (a) The point set of the Stanford bunny. (b) The convex hull.

1.1 Convex Hull and Its Algorithms

The convex hull of a set of n points is the smallest convex region that encloses all the points of the set. Figure 1.1 shows the point set of the Stanford bunny [Sta12] and its convex hull. It is a computationally efficient structure because of its convexity, while providing an approximation of the shape of the point set. Therefore, it is widely used as a bounding volume to check intersection or collision between objects in scientific visualization and computer games [LZB08, MY06], as well as to approximate robots and obstacles for the purpose of path planning [OII03, Str04]. In addition, convex hull is also employed in visual pattern matching [HH06], biology and genetics [WLYZ+09], and astronomy [FGE01, ACHS05].

Algorithms for 2D convex hull have been well studied. Graham [Gra72] presents the first algorithm with worst-case time complexity of $\mathrm{O}(n \log n)$, termed *Graham's scan*. This algorithm performs a scan in linear time on a star-shaped polygon constructed by sorting. Jarvis [Jar73] designs *Jarvis' march* algorithm, which incrementally finds all the edges of the convex hull. The worst-case time complexity of Jarvis' march is $\mathrm{O}(nh)$, where h is the number of the vertices on the convex hull. Divide-and-conquer is also adopted to design convex hull algorithms in [PH77] and [KS86]. While the algorithm in [PH77] directly applies divide-and-conquer, that in [KS86] uses the idea of "marriage-before-conquest" to obtain an optimal worst-case time complexity of $\mathrm{O}(n \log h)$. Eddy [Edd77] presents the first version of *Quickhull*, and Wenger [Wen95] improves it by utilizing randomization. Both of these algorithms have the worst-case time complexity of $\mathrm{O}(n^2)$, and the latter one has the expected running time of $\mathrm{O}(n \log h)$.

The ideas of some 2D convex hull algorithms mentioned above are also adapted to three and higher dimensions. Chand and Kapur [CK70] design the *gift-wrapping* algorithm using the same idea of Jarvis' march. The algorithm in [PH77] can be used to compute 3D convex hull. Edelsbrunner and Shi [ES91] present a 3D algorithm with the worst-case time complexity of $\mathrm{O}(n \log^2 h)$, which can be seen as a 3D version of the "marriage-before-conquest" algorithm in [KS86]. Barber et al. [BDH96] extend the randomized Quickhull to high dimensions. On the other hand, Graham's scan can be neither directly used nor extended to compute convex hull in three and higher dimensions. Chan [Cha96] invents a novel algorithm with the optimal worst-case time complexity of $\mathrm{O}(n \log h)$ by combining Jarvis' march and some optimal 3D convex hull algorithms.

Parallel algorithms of convex hull have also been extensively studied. For example, Miller and Stout [MS88] and Amato and Preparata [AP93] describe parallel algorithms with the time complexity of $\mathrm{O}(\log n)$ for n points using $\mathrm{O}(n)$ 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, the algorithms designed by Dehne et al. [DDD+95] and Gupta and Sen [GS03] may be applicable. These algorithms, however, do not have known implementations that can demonstrate their usefulness in practice.

Recently several convex hull algorithms for the GPU have been developed. Algorithms simulating 2D Quickhull are proposed in [SRKN11] and [JD11]. Tzeng and Owen [TO12] further extend them to three and higher dimensions. However, the output of their algorithm, *CudaQuickHull*, does not have connectivity information, and often contains non-extreme vertices; see Figure 1.2. Stein et al. [SGES12] propose to compute the convex hull in $\mathbb{R}^3$ by iteratively inserting points and flipping all concave edges. This algorithm does not always work, because it prohibits the flipping of concave edges if that causes self-intersection (as indicated in their algorithm) and therefore the final result might still contain concave edges. Tang et al. [TyZTM12] present a hybrid CPU-GPU algorithm. In this algorithm, points are excluded as long as it is proved to be interior during the growing of a polytope by inserting input points on the GPU. Then, points surviving that process are passed to the CPU and a CPU-based algorithm (e.g. Quickhull) is used to compute the convex hull.

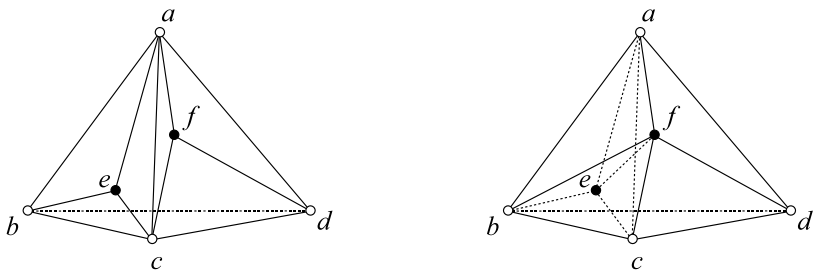


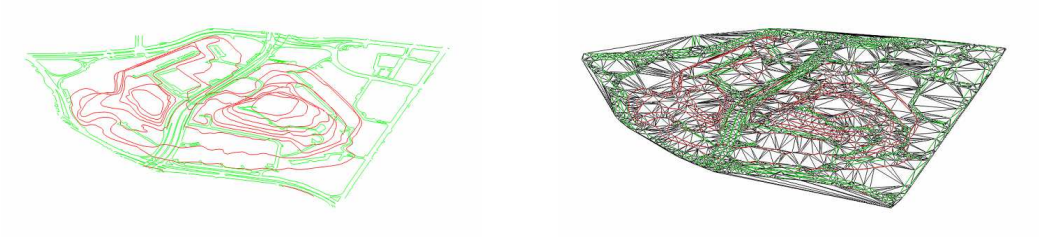
Figure 1.2: An example where CudaQuickHull outputs a wrong result. In (a), after creating the initial tetrahedron $abcd$, CudaQuickHull flags e with $\triangle abc$ and f with $\triangle acd$, and outputs both as extreme vertices. In the correct result in (b), e is not an extreme vertex since it lies inside tetrahedron $fabc$.

1.2 Regular Triangulation and Its Algorithms

Regular triangulation is a geometric structure related to convex hull. Given a set S of points with weights, we obtain S' in one higher dimension by lifting S; the regular triangulation of S corresponds to the lower part of the convex hull of S'. Regular triangulation is an important tool for solving other geometric problems such as constrained Delaunay triangulation [MK04] and mesh refinement [CDE+99]. It is also used in behavior simulation of granular materials [Fer01], path planning [ACK01] and channel detection of protein molecules [Zem09].

Delaunay triangulation is a special regular triangulation. The Delaunay triangulation of a point set S is the regular triangulation of a weighted point set whose points come from S and have weight 0. Delaunay triangulation is also a dual structure of Voronoi diagram. It partitions the convex hull of S, and is a crucial tool for finite element analysis [HDSB01]. More importantly, Delaunay triangulation has many distinctive and favorable properties compared with other ways of partitioning [For97]. Therefore, it has applications in many other areas such as terrain rendering [Gol94, KKS06] and path planning [PS85, LNSV06]. Figure 1.3 illustrates the usage of Delaunay triangulation for constructing terrain from contour and road data.

Many sequential algorithms are developed for computing regular triangulation and Delaunay triangulation, and most of them adopt ideas similar to those of the convex hull algorithms introduced in the previous section. Motivated by the gift-wrapping algorithm, Beyer et al. [BSDMH05] construct the tetrahedrons of the Delaunay/regular triangulation incrementally. Guibas et al. [GS85] and Dwyer [Dwy87] propose algorithms based on the divide-and-conquer approach to compute 2D Delaunay triangulation with optimal worst-case time complexity. Cignoni et al. [Cig98] extend these algorithms to higher dimensions and design the *DeWall* algorithm. Another class of al-



(a) (b)

Figure 1.3: (a) Data of contours (red) and roads (green), and (b) its Delaunay triangulation with constraints.

gorithms uses incremental insertion, i.e. constructs the Delaunay/regular triangulation of $k+1$ points from that of k points by inserting one more point and updating the structure. To update the structure, Bowyer [Bow81] and Watson [Wat81] re-triangulate the affected regions with the newly inserted point (this method is named *Bowyer-Watson's algorithm*), while others use local transformation with flips [Joe91, Raj91, ES92, Joe93]. Besides these approaches, Lawson presents an algorithm that can transform an arbitrary 2D triangulation to its Delaunay triangulation by flips [Law72].

Many parallel algorithms employ the same ideas as the sequential ones do. Teng et al. [TSBP93] and Beyer et al. [BSDMH05] propose parallel algorithms based on incremental construction. These algorithms have a high total work complexity in the worst case. Blelloch et al. [BMHT99] adapt divide-and-conquer in parallel in their algorithm of computing 2D Delaunay triangulation. Batista et al. [BMPS09] present a parallel version of Bowyer-Watson's algorithm for multi-core computers. Qi et al [QCT12] design GPU-DT to compute 2D Delaunay triangulation on the GPU, but their method cannot compute 2D regular triangulation.

1.3 Local Transformation

Local transformation gradually changes a geometric structure by operations that use information local to neighboring simplices. As mentioned, a well-known algorithm of local transformation is Lawson's flip algorithm. Given an arbitrary triangulation in $\mathbb{R}^2$, it iteratively flips away locally non-Delaunay edges till none remains to get the Delaunay triangulation. A nice property of Lawson's flip algorithm is its flexibility on the flip sequence: no particular execution order is needed for flipping among the many non-Delaunay edges available, since any such sequence leads to the Delaunay triangulation.

On the other hand, the situation in higher dimensions is quite different. Flipping can still be used to compute regular triangulation [ES92, Joe93, Raj91], but it requires a very restrictive input triangulation. Specifically, it can only reach the correct output

if the input is created by inserting one new point into regular triangulation.

In another development, Shewchuk creates the splaying operation to compute convex hull and regular triangulation [She05]. Splaying is the insertion of new points into the convex star of a point to enlarge the star. The local transformation of star splaying uses splaying as its operation, in any execution order, to transform the stars of a set of points to the convex hull, and it works in any dimension.

Summing up the above, local transformation may be distinguished by the operation, the type of input geometric structure, and the flexibility on the execution order. It has some exciting computational advantages as compared with other algorithmic paradigms. First, local transformation is usually very simple since it operates on local information only. Second, when the input geometric structure is close to the required output, local transformation is very efficient, making only minor amendment to reach the output. Third, they are particularly amendable to parallel computation, e.g. on the GPU, especially when the execution order is free.

Given the above advantages, local transformation is also often applied to build geometric structures from a point set, by first preprocessing the point set into some initial geometric structure acceptable to the local transformation.

Despite the existing works mentioned above, many interesting yet difficult problems related to local transformation are still unsolved such as flipping for 2D regular triangulation. Besides, local transformation to geometric structures other than regular triangulation and convex hull is also a significant problem with many potential applications.

This thesis presents a novel flip algorithm called flip-flop to compute 2D regular triangulation from an arbitrary triangulation and 3D convex hull from a star-shaped polyhedron. It also presents a novel GPU approach that employs the known star splaying to compute the convex hull of points in $\mathbb{R}^3$, and compares it with flip-flop. Besides, the thesis invents the twist operation to create a star-shaped polygon from an arbitrary one, and to create a star-shaped polyhedron for a particular class of polyhedrons.

All the parallel algorithms proposed in this thesis are presented specifically for the GPU, while they should also work well on any other data-parallel shared-memory architecture. We use some common features on parallel systems such as atomic operations and global synchronization, as well as some standard parallel primitives such as prefix sum, reduction, partition, and also sorting. These primitives are available in several popular GPU libraries [CUD12, Thr14, CUB14], and are free from numerical errors.

1.4 Outline of the Thesis

Chapter 2 reviews some important concepts of convex hull, regular triangulation, and the flip operation. Particularly, extended triangulation as a useful structure in this

thesis is introduced in this chapter.

Chapter 3 formalizes the framework of local transformation with five essential elements: geometric structure, criterion, local operation, local check, and candidate. This interpretation of the framework is illustrated using two existing algorithms of local transformation.

Chapter 4 revisits the famous Graham's scan algorithm to compute 2D convex hull. It reformulates the scanning stage as a flip algorithm that has free execution order, thus making that stage suitable for parallel computation. By doing so, the chapter further develops a parallel version of Graham's scan as well as two other improvements. Our implementation of these algorithms on the GPU reaches an order of magnitude speedup over two well-known CPU implementations of existing convex hull algorithms.

Chapter 5 solves the 2D regular triangulation problem with a novel flip algorithm. It is known that Lawson's flip algorithm is a hill-climbing procedure and can be stuck at a local optimum without reaching the required output. By varying the criterion and the local check in the local transformation framework (Chapter 3) to devise the so-called flip-flop algorithm, we can transform an arbitrary 2D triangulation to its regular triangulation by flips.

Chapter 6 advances flip-flop to solve the 3D convex hull problem. Starting from a star-shaped polyhedron in $\mathbb{R}^3$, flip-flop locally identifies non-extreme vertices of the polyhedron, and flips edges incident to them to reduce their degrees in order to eventually remove them. Our implementation of the algorithm on the GPU reaches more than an order of magnitude speedup over existing CPU solutions.

Chapter 7 investigates the splaying operation as it is known to be powerful for repairing convex hull in arbitrary dimensions. This investigation is to realize it to compute 3D convex hull of a point set, and to compare it with the proposed flip-flop algorithm in Chapter 6. Splaying works best when the input is close to the output; the thesis thus explores a way to first derive a set of convex stars from the closely related structure of digital Voronoi diagram, and then apply star-splaying on these stars to compute the convex hull. The verdict is that flip-flop outperforms splaying in many ways for our GPU implementation.

Chapter 8 studies local transformations to construct star-shaped polytopes that are used in Chapter 4 and Chapter 6. For the 2D setting, a novel operation called twist is used to compute a star-shaped polygon from an arbitrary polygon, possibly with self-intersection. For the 3D setting, the chapter presents a way to work on a special class of input polyhedrons where one extreme vertex connects to all the other vertices. The general problem in three and higher dimensions remains open.

Chapter 9 drills into the possibility of flipping from one regular triangulation to another regular triangulation of the same point set in three and higher dimensions. It discusses a sequential approach where the execution order is restrictive as discovered in prior works. With the introduction of a time line to study local transformation,

interesting properties are observed and proved. These new insights are potentially useful when searching for a flip algorithm with more relaxed execution order. Until then, it remains an open problem whether flipping with no strict execution order can transform one regular triangulation to another.

Chapter 10 concludes the thesis with a summary of the work done, and an outlook of the area.

Experiment Setting

In the experiments mentioned in this thesis, our GPU implementations use the CUDA programming model by NVIDIA, and our CPU implementations use C++. For all these implementations, we adopt the exact arithmetic [She97] and apply the Simulation of Simplicity (SoS) technique [EM90] to guarantee their exactness and robustness.

When comparing with the existing CPU implementations, we do not include the time of transferring data between the CPU and the GPU in our GPU implementations. We will explain the affect of this time to the overall performance in the analysis of time breakdown.

All the experiments are conducted on a PC with an Intel i7 2600K 3.4GHz CPU, 16GB of DDR3 RAM and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory, unless otherwise stated. Visual Studio 2008 and CUDA 4.0 Toolkit are used to compile all the programs, with all optimizations enabled.

CHAPTER 2

Geometric Background

This chapter introduces important geometric concepts related to this thesis. The material is summarized from existing works such as [PS85, Law87, O'R98, Ede01, DLRS10].

Convex hull, a fundamental concept, is the starting point for discussion in almost all the books on computational geometry. It is used to define and develop other geometric concepts such as simplex and simplicial complex, followed by triangulation and polytope. We discuss these concepts in Section 2.1.

Delaunay triangulation is a special triangulation among all possible triangulations of a point set where the circumsphere of each simplex is empty with respect to the point set. Regular triangulation is a generalization of Delaunay triangulation for a set of points with weights. We discuss them in Section 2.2.

Flip as one of the most important local operations is discussed in Section 2.3. It is powerful because one triangulation can be transformed to another completely different one via successive flips. It will be used in a number of algorithms of local transformation in the subsequent chapters.

In this chapter, we assume the points of any given point set in $\mathbb{R}^d$ are in general position, i.e., no $d+1$ points lie in the same $(d-1)$-dimensional hyperplane and no $d+2$ points lie on the same $(d-1)$-dimensional sphere.

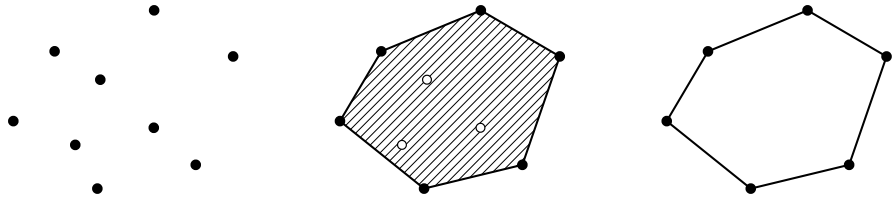


Figure 2.1: (a) A set S of planar points. (b) The convex hull of S. (c) The boundary of the convex hull.

2.1 Convex Hull and Triangulation

A point set is *convex* if the line segment $\overline{pq}$ is completely contained in the set for any two points p and q of the set. Given a point set $S = \{p_i \mid 0 \leq i < n\}$, a *convex combination* of S is a point $x = \sum \lambda_i p_i$ where $\sum \lambda_i = 1$ and $\lambda_i \geq 0$ $(0 \leq i < n)$. The *convex hull* of S, $conv(S)$, is the set of all convex combinations of S. The boundary of $conv(S)$ is denoted as $\mathcal{CH}(S)$. Figure 2.1 shows the convex hull of a set of planar points and its boundary. It is known that $conv(S)$ is the smallest convex set containing all the points of S.

A point p of $conv(S)$ is *extreme* if p is not contained in segment $\overline{uv}$ for any two points u and v of $conv(S)$, $p \notin \{u, v\}$; otherwise p is *non-extreme*. In Figure 2.1(b), the points represented by the solid dots are extreme and all the other points of the convex hull are non-extreme. The convex hull of S can be represented using only its extreme points, i.e. $conv(S) = conv(S')$ where $S' = \{p \mid p \in conv(S)$ and p is extreme$\}$.

A d-dimensional simplex, or simply *d-simplex*, is the convex hull of $d+1$ points in $\mathbb{R}^d$; see Figure 2.2. The d-simplex is the convex hull with the least extreme vertices among all the convex hulls that cannot be embedded in $\mathbb{R}^{d-1}$. Define the (-1)-simplex to be the empty set $\emptyset$. In the d-simplex C constructed by $d+1$ points, each of the $d+1$ points is called a *vertex* of C. A *k-face* of C is the convex hull of some $k+1$ vertices of C and thus is a k-simplex. A $(d-1)$-face of C is also called a *facet* of C.

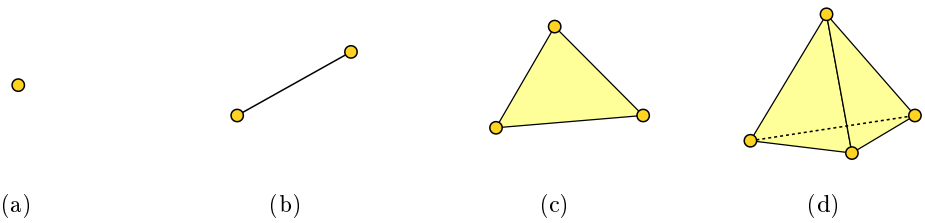


Figure 2.2: (a) A 0-simplex is a vertex. (b) A 1-simplex is an edge. (c) A 2-simplex is a triangle. (d) A 3-simplex is a tetrahedron.

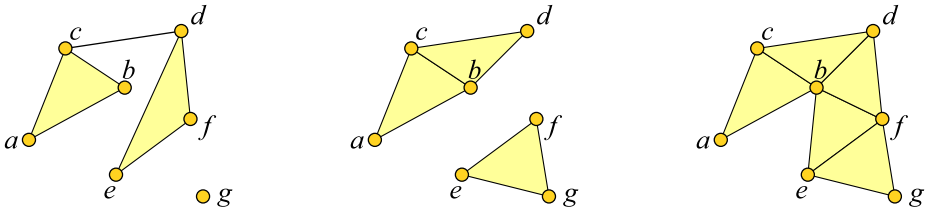


Figure 2.3: (a) A 2-complex that is not pure. (b) and (c) 2-complexes that are pure.

A *simplicial complex* is a set of a finite number of simplices that satisfy the following conditions:

(1) All faces of a simplex are also in the simplicial complex.

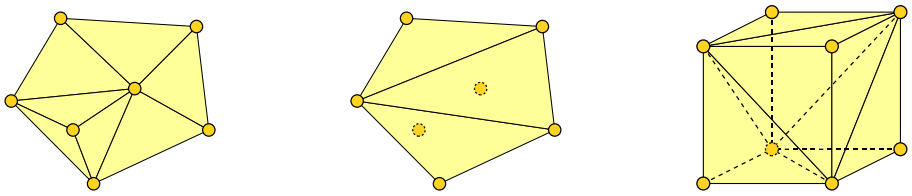
(2) The intersection of any two simplices is a common face of them.

Note that the latter condition implies that any two simplices either are disjoint or share a common face. The *dimension* of a simplicial complex is the largest dimension of its simplices. A simplicial complex with dimension d is called a *d-complex*. A simplex that is not a face of any other simplex is called a *facet* or a *maximal simplex* in a simplicial complex. A simplicial complex is *pure* if all its facets have the same dimension with it; see Figure 2.3. A $(d-1)$-face in a pure d-complex is called a *ridge* of the complex. Ignoring the second condition in the definition of simplicial complex, we obtain *abstract simplicial complex*. Abstract simplicial complex is a combinatorial description of simplicial complex.

In an (abstract) simplicial complex, the *star* of a simplex τ, $star(\tau)$, is the set of all the simplices that contain τ as a face; the *link* of τ, $link(\tau)$, is the set of all the simplices that are the faces of the simplices of $star(\tau)$ and disjoint with τ. Take Figure 2.3(c) as an example, $star(b)$ is $\{b, \overline{ab}, \overline{cb}, \overline{db}, \overline{fb}, \overline{eb}, \triangle abc, \triangle cbd, \triangle dbf, \triangle fbe\}$ and $link(b)$ is $\{\emptyset, a, c, d, f, e, \overline{ac}, \overline{cd}, \overline{df}, \overline{fe}\}$; $star(\overline{bd})$ is $\{\overline{bd}, \triangle cbd, \triangle dbf\}$ and $link(\overline{bd})$ is $\{\emptyset, c, f\}$.

A (convex) polytope is usually defined as a particular convex set in Euclidean space. In this thesis, we mainly focus on the boundary of polytope, which is actually a pure abstract $(d-1)$-complex embedded in $\mathbb{R}^d$ where each ridge is shared by exactly two facets. We use the term *polytope* to refer to this boundary, and allow it to be not convex nor free of self-intersection. A 2D polytope is further called a *polygon*, and a 3D one is called a *polyhedron*. A d-dimensional polytope is homeomorphic to a d-dimensional sphere, and it may have self-intersection. When a polytope has no self-intersection, it is also used to represent the closed region that it bounds.

A polytope $\mathcal{P}$ is *star-shaped* w.r.t. a point s if for any point p on $\mathcal{P}$, $\overline{ps}$ does not intersect with $\mathcal{P}$ except for p. The point s is called a *kernel point* of this star-shaped polytope, and definitely lies inside $\mathcal{P}$. A star-shaped polytope does not have self-intersection by definition.



(a) (b) (c)

Figure 2.4: (a) A 2D triangulation containing all the input points as vertices. (b) A 2D triangulation with some missed input points. (c) A 3D triangulation.

The boundary of the convex hull of S, $\mathcal{CH}(S)$, is usually described as a polytope. $\mathcal{CH}(S)$ is star-shaped w.r.t. any point in the interior of $conv(S)$. $\mathcal{CH}(S)$ can be partitioned into a pure abstract complex where each facet is below the interior of $conv(S)$ and the other where each facet is above. The former is called the *lower hull* of S, while the latter is called the *upper hull* of S. Given a polytope $\mathcal{P}$ whose vertex set is S, we say a face of $\mathcal{P}$ is *extreme* if it is also a face of $\mathcal{CH}(S)$; otherwise it is *non-extreme*.

Given a point set S in $\mathbb{R}^d$, a *triangulation* of S, $\mathcal{T}(S)$, is a simplicial complex that satisfies the following conditions:

(1) Its vertices are in S.

(2) The union of all its d-simplices are $conv(S)$.

In $\mathcal{T}(S)$, a facet of each d-simplex is either in $\mathcal{CH}(S)$ or a common facet of exactly two d-simplices. Thus a triangulation is a pure simplicial complex. Note that some points of S may not be the vertices of $\mathcal{T}(S)$. Figure 2.4 shows some examples of triangulations. Furthermore, we say d-dimensional *abstract triangulation* to refer to an abstract simplicial complex that is homeomorphic to a d-dimensional ball.

Given a set S of points and another point s in $\mathbb{R}^d$, an *extended triangulation* $\mathcal{T}$ of S w.r.t. s is a pure simplicial complex that consists of *real facets* and *virtual facets*. A *real facet* of $\mathcal{T}$ is a d-simplex whose vertices are from S, and the boundary of the union of all the real facets is a d-polytope $\mathcal{P}$ star-shaped w.r.t. s. For each ridge on $\mathcal{P}$, shoot a ray $\overrightarrow{sp}$ for each vertex p of the ridge, and build a cone which is the convex hull of these rays. These cones partition the space outside $\mathcal{P}$ into many regions, each of which corresponds to a ridge on $\mathcal{P}$. By maintaining a *virtual vertex* v, each of these regions is considered as a d-simplex consisting of its corresponding ridge and v, and called a *virtual facet* of $\mathcal{T}$. For example in Figure 2.5, the region $\mathcal{CH}\{\overrightarrow{sp}, \overrightarrow{sq}\} - \triangle spq$ is the virtual triangle $\triangle vpq$ of the given extended triangulation.

The virtual vertex v does not have a real position. For a vertex p on $\mathcal{P}$, v can be imagined as a point that lies on $\overrightarrow{sp}$ and extremely far away from p; see Figure 2.5. By definition, the star of v contains all the virtual facets of $\mathcal{T}$, and the link of v is equal to $\mathcal{P}$. Obviously, $\mathcal{T}$ is a simplicial complex satisfying the following conditions:

(1) Its vertices are in $S \cup \{v\}$.

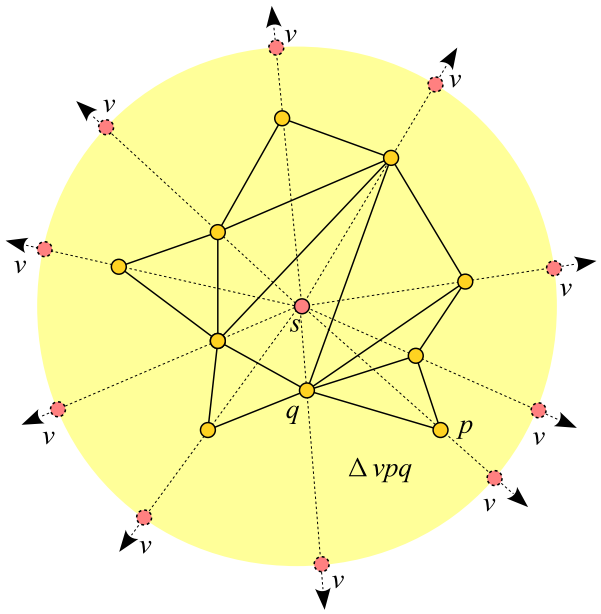


Figure 2.5: An extended triangulation w.r.t. s.

(2) The union of all its d-simplices is $\mathbb{R}^d$.

(3) Each its ridge is shared by exactly two facets.

Note that the concept of extended triangulation is more general than our definition. Formally, an extended triangulation is the triangulation of a compactification of the Euclidean space. In our definition mentioned above, it is actually the triangulation of a compactification to a sphere of the same dimension.

2.2 Delaunay and Regular Triangulation

Let S be a set of n points in $\mathbb{R}^d$. Given two points $p = (a_1, a_2, ..., a_d)$ and $q = (b_1, b_2, ..., b_d)$ of S, their *Euclidean distance* is $|pq| = \sqrt{\sum (a_i - b_i)^2}$. Let C be a d-simplex whose vertices belong to S. The *circumsphere* of C is the sphere passing through all the vertices of C, and the *circumcenter* of C is the center of its circumsphere. Let o be the circumcenter of C and r be the radius of its circumsphere; $|po| = r$ for any vertex $p \in C$. C is *globally Delaunay* (w.r.t. S), or *Delaunay* for short, if $|qo| > r$ for any point $q \in S$ that does not belong to C. Otherwise it is *globally non-Delaunay* (w.r.t. S), or simply *non-Delaunay*. Generally, a simplex is Delaunay if it is a face of some Delaunay d-simplex; otherwise it is non-Delaunay. The *Delaunay triangulation* of S, $\mathcal{DT}(S)$, is the unique triangulation of S in which all the points of S appear as vertices and all the d-simplices are Delaunay. Figure 2.6 shows the Delaunay triangulation of nine points in $\mathbb{R}^2$.

In a triangulation $\mathcal{T}(S)$, let C_1 and C_2 be its two facets that share a common ridge F; let p_1 and p_2 be the link vertices of F that are in C_1 and C_2 respectively. F is *locally*

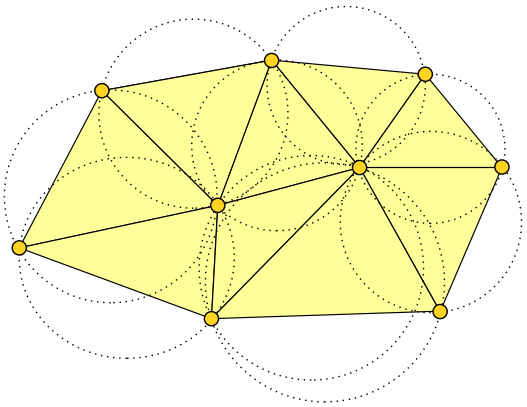


Figure 2.6: The Delaunay triangulation of nine planar points. Each its triangle has a empty circumsphere.

Delaunay if p_1 lies outside the circumsphere of C_2 (or equivalently p_2 lies outside the circumsphere of C_1); otherwise it is *locally non-Delaunay*. The ridges of $\mathcal{T}(S)$ on $\mathcal{CH}(S)$ are considered locally Delaunay. The following well-known theorem, firstly introduced by Boris Delaunay, indicates how to recognize a Delaunay triangulation in a local manner:

Theorem 2.1 ([Law87]). *A triangulation with its vertex set S in $\mathbb{R}^d$ is $\mathcal{DT}(S)$ if and only if all its ridges are locally Delaunay.*

Regular triangulation is the weighted version of Delaunay triangulation. Given a point p, a *weighted point* of p is created by associating p with a real number w_p as its weight. By associating each point $p_i \in S$ with a weight w_i for $0 \leq i < n$, we get a *weighted point set* of S w.r.t. $W = \{w_i \mid 0 \leq i < n\}$. W is called a *weight set* of S. By regarding the weighted point of p_i as the sum of p_i and w_i, we can treat the weighted point set as the direct sum of S and W, and denote it as $S \oplus W$. A triangulation $\mathcal{T}(S \oplus W)$ of $S \oplus W$ is also a triangulation of S. The *power distance* from a point x to a weighted point p is $\pi_p(x) = |xp|^2 - w_p$, where w_p is the weight of p. Two weighted points p and q are *orthogonal* if $|pq|^2 = w_p + w_q$.

Let C be a d-simplex whose vertices belong to $S \oplus W$. The *orthogonal center* of C is the weighted point z that is orthogonal to all the weighted vertices of C. C is *globally regular* (w.r.t. $S \oplus W$), or *regular* for short, if $|qz|^2 > w_q + w_z$ for any weighted point $q \in S \oplus W$ that does not belong to C; otherwise C is *globally non-regular* (w.r.t. $S \oplus W$), or simply non-regular. A simplex from $S \oplus W$ is called *redundant* (w.r.t. $S \oplus W$) if it is not a face of any d-simplex that is regular; otherwise it is *non-redundant* (w.r.t. $S \oplus W$). A point of $S \oplus W$ is called *redundant* (resp., *non-redundant*) if the 0-simplex containing the point is redundant (resp., non-redundant). The *regular triangulation* of $S \oplus W$, denoted as $\mathcal{RT}(S \oplus W)$, is the triangulation of $S \oplus W$ where all the non-redundant points of $S \oplus W$ are contained as vertices and all the facets are regular. When the weights of all the points of $S \oplus W$ are 0, $\mathcal{RT}(S \oplus W) \equiv \mathcal{DT}(S)$.

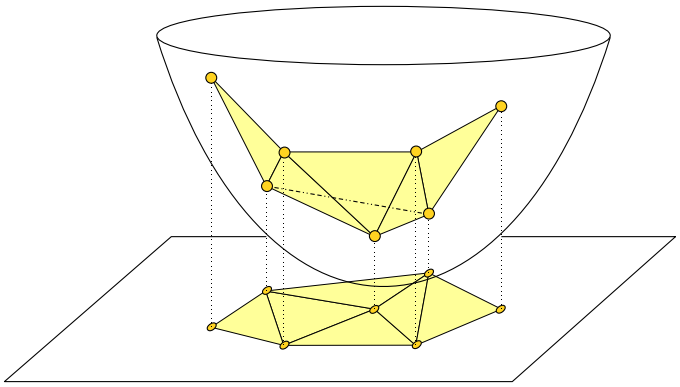


Figure 2.7: A regular triangulation in $\mathbb{R}^2$ and its corresponding lower hull in $\mathbb{R}^3$.

In a triangulation $\mathcal{T}(S \oplus W)$, let C_1 and C_2 be its two facets that share a common ridge F. Let p_i be the link vertex of F in C_i and z_i be the orthogonal center of C_i for $i = \{1, 2\}$. F is *locally regular* if $|p_1 z_2|^2 > w_{p_1} + w_{z_2}$ (or equivalently $|p_2 z_1|^2 > w_{p_2} + w_{z_1}$); otherwise it is *locally non-regular*. The ridges of $\mathcal{T}(S \oplus W)$ on $\mathcal{CH}(S)$ is intrinsically locally regular. We have the following theorem:

Theorem 2.2 ([ES92]). *A triangulation with its weighted vertex set $S \oplus W$ in $\mathbb{R}^d$ is $\mathcal{RT}(S \oplus W)$ if and only if all its ridges are locally regular.*

Regular triangulation in $\mathbb{R}^d$ can be obtained from convex hull in $\mathbb{R}^{d+1}$; see Figure 2.7. Given a weighted point $p = (a_1, a_2, ..., a_d)$ in $\mathbb{R}^d$ whose weight is w_p, define its *lifted point* as $p^+ = (a_1, a_2, ..., a_d, \sum a_i^2 - w_p)$ in $\mathbb{R}^{d+1}$. The $(d+1)$-th coordinate of p^+ is the *height* of p. Call $S^+ = \{p^+ \mid p \in S \oplus W\}$ the *lifted point set* of $S \oplus W$. The projection along the $(d+1)$-th axis of the lower hull of S^+ gives $\mathcal{RT}(S \oplus W)$ [ES92]. Particularly, a regular facet of $\mathcal{RT}(S \oplus W)$ corresponds to a facet of the lower hull of S^+, while the lifted point of a redundant point of $S \oplus W$ is above the lower hull of S^+. Therefore, regular triangulation can be computed using the algorithms of convex hull.

One interesting theoretical problem is checking the *regularity* of a triangulation $\mathcal{T}(S)$, i.e. whether there is a weight set W of S so that $\mathcal{T}(S) \equiv \mathcal{RT}(S \oplus W)$. An existing solution is to use linear programming, where the variables are the weights of the vertices and the constraints are formed from the ridges of $\mathcal{T}(S)$ [MII96].

For an extended triangulation $\mathcal{T}((S \cup \{v\}) \oplus W)$ where v is the virtual vertex, we need to specially define the local regularity of the ridges that are incident to virtual facets. Let s be the kernel point of the extended triangulation. A ridge incident to exactly one virtual facet is always locally regular by definition: one can think that v is infinitely far away so that the circumsphere of any real facet cannot include v. Let G be a ridge incident to two virtual facets C_1 and C_2, and F_i be the ridge of $C_i \cap link(v)$ for $i \in \{0, 1\}$. G is locally regular if F_1 and s lie on the same side of the hyperplane passing through F_2 (equivalently, if F_2 and s lie on the same side of the hyperplane passing through F_1), or otherwise G is locally non-regular. Take the extended triangulation in

Figure 2.5 as an example. The edge $\overline{vp}$ is locally regular, while $\overline{vq}$ is locally non-regular. Note that the local regularity defined here is not related to the weights of the vertices, and thus a locally regular ridge incident to virtual facets is also locally Delaunay. When all the ridges are locally regular, the ridges that are incident to exactly one virtual facet form $\mathcal{CH}(S)$, and the real facets of $\mathcal{T}$ form $\mathcal{RT}(S \oplus W)$.

2.3 Flip on Triangulation

Flip is a local operation that transforms one triangulation to another. Flip algorithm is an algorithm to transform triangulations using flips. Since flipping with degenerate points (i.e. the points may not be in general positions) is much more complicated than that with points in general position, many algorithms do not consider such flipping and leave this issue to implementation. Studies of flip with degenerate points can be found in [Law87, Joe93].

Let S be a set of $d+2$ points in $\mathbb{R}^d$. To triangulate S there are exactly two ways corresponding to the lower and upper hulls of the lifted point set of S. This can be concluded from Radon's theorem:

Theorem 2.3 ([Rad21]). *Let S be a set of $d+2$ points in $\mathbb{R}^d$. Then there exists a partition $T = U \cup V$ so that $conv(U) \cap conv(V) \neq \emptyset$.*

Consider a partition $T = U \cup V$ of S according to Theorem 2.3. Let $\mathcal{T}_U(S)$ be the set of all the d-simplices from S each of which does not contain a point of U. Define $\mathcal{T}_V(S)$ in the same way for V. By Radon's theorem, a d-simplex of $\mathcal{T}_U(S)$ and one of $\mathcal{T}_V(S)$ cannot appear in the same triangulation of S. Thus $\mathcal{T}_U(S)$ and $\mathcal{T}_V(S)$ are the only two triangulations of S. A *flip* on S is to replace one triangulation of S with the other. Based on the distribution of S, the sizes of $\mathcal{T}_U(S)$ and $\mathcal{T}_V(S)$ vary from 1 to $d+1$, and the sum of their sizes is $d+2$. A flip from a triangulation with k facets to one with $d+2-k$ facets is further called a k-$(d+2-k)$ *flip* or a k *to* $(d+2-k)$ *flip*. Specially, a 1-$(d+1)$ flip inserts a vertex into the triangulation, while a $(d+1)$-1 flip removes one. Figure 2.9 shows all the types of flips in 1D, 2D, and 3D triangulation.

In a d-dimensional triangulation $\mathcal{T}$, let F be an internal ridge, i.e. a ridge not on the boundary of $\mathcal{T}$, and S_F be the point set consisting of the vertices of F and the two link vertices of F. We use "flip on F" or "flipping F" to indicate the flip on S_F. If the flip on F is a k-$(d+2-k)$ flip, F is further called a k-$(d+2-k)$ ridge. The flip on F does not exist in $\mathcal{T}$ if neither of the two triangulations of S_F appears in $\mathcal{T}$. In such case F is called *unflippable*; otherwise it is *flippable*. A ridge on the boundary of $\mathcal{T}$ is considered unflippable.

The *induced-subcomplex* of F, $\mathcal{T}_F$, consists of all the simplices of $\mathcal{T}$ that span the points in S_F. If the underlying space of $\mathcal{T}_F$ is convex (indicating that $\mathcal{T}_F$ is a triangulation of S_F), F is flippable; otherwise it is unflippable. Examples on a 2D triangulation

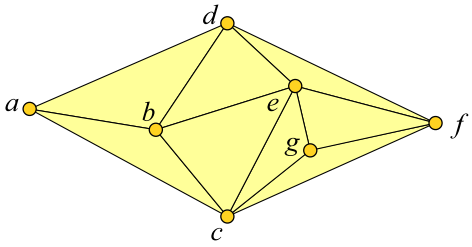
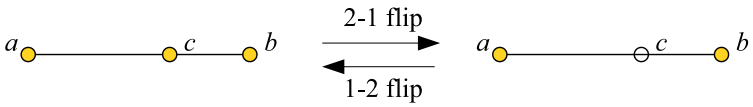


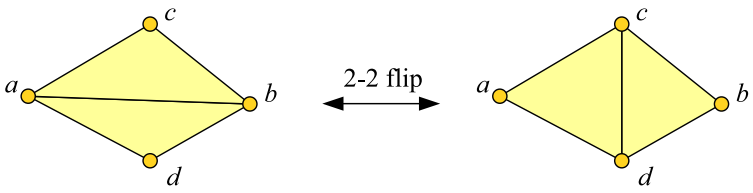
Figure 2.8: The induced-subcomplex of $\overline{ab}$ is $\{\triangle abc, \triangle abd\}$ and that of $\overline{eg}$ is $\{\triangle gec, \triangle gcf, \triangle gfe\}$. Edges $\overline{be}, \overline{bd}, \overline{ec}, \overline{cg}, \overline{eg}$ and $\overline{fg}$ are flippable, while others are unflippable.

are shown in Figure 2.8. Given a flip on a ridge F, the *induced-subcomplex* of the flip is that of F, and thus has a convex underlying space by definition. Note that the above definitions about flip are also applicable to ridges of an extended triangulation.

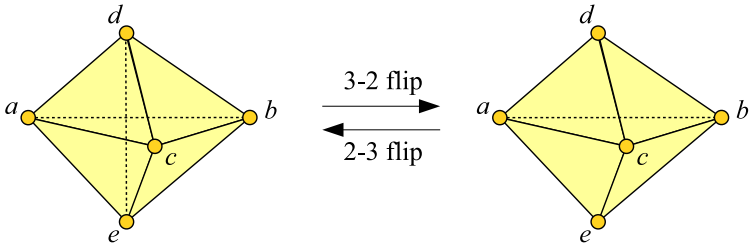
Flipping a flippable ridge results in another triangulation. Based on this we can build the *flip graph* of a set of triangulations, in which a node represents a triangulation and an edge indicates that two triangulations can be transformed to each other by one flip. The flip graph of the regular triangulations of a point set in any dimension is connected, and it corresponds to the *secondary polytope* of the point set [GZK91, BFS90]. The flip graph of all the triangulations of a point set in $\mathbb{R}^2$ is connected [Law72], while for certain point sets in $\mathbb{R}^5$ and $\mathbb{R}^6$ the flip graphs are disconnected [San05, San06]. The situations in $\mathbb{R}^3$ and $\mathbb{R}^4$ are unknown. The connectivity of flip graph is not only of theoretical interest, but also helpful for the development of practical algorithms.



(a) 1D flip



(b) 2D flip



(c) 3D flip

Figure 2.9: Flips in 1D, 2D, and 3D triangulation.

CHAPTER 3

Algorithms of Local Transformation

This chapter revisits some known algorithms to categorize algorithms of local transformation, and presents a framework for better understanding these algorithms and designing new ones.

Local transformation is well-known for its adoption in designing geometric algorithms. Section 3.1 summarizes these algorithms with emphasis on a few important attributes, including the workable dimension in Euclidean space, the type of input geometric structure, the local operation, and the order of executing these operations.

Section 3.2 discusses the essential components in an algorithm of local transformation: geometric structure, local operation, criterion, local check, and candidate. It also elaborates on the common methodology for proving the termination and the correctness of such algorithms, and presents two workflows of local transformation for sequential computation and parallel computation respectively. All these results as a framework of local transformation are widely used in the following chapters.

Following the formalism in Section 3.2 about local transformation, Section 3.3 presents two known examples: Lawson's flip algorithm and Shewchuk's star splaying algorithm.

3.1 Overview

There are several existing algorithms that compute convex hull and regular triangulation using local transformation: the scanning stage of Graham's scan [Gra72], Lawson's flip algorithm [Law72], Rajan's flip algorithm [Raj91], Edelsbrunner-Shah's flip algorithm [ES92] (and similarly Joe's flip algorithm [Joe93]), and star splaying [She05]. For the simplicity of presentation, we use Graham's scan to refer to its scanning stage in this chapter. These algorithms can be categorized by the following aspects.

1. **Workable dimension and input structure**. These two factors decide the usefulness of these algorithms. Graham's scan transforms an arbitrary star-shaped polygon to the convex hull, and Lawson's flip algorithm transforms a 2D triangulation to its Delaunay triangulation; they only work in $\mathbb{R}^2$. The input structure of Rajan's and Edelsbrunner-Shah's flip algorithms is a triangulation created by inserting one point into the regular triangulation of a weighted point set; we denote it as "RT+1" triangulation. Although they work in any dimension, their usefulness is restricted by this special input structure. The best algorithm in terms of workable dimension and input structure is star splaying, which can transform almost any set of stars in any dimension to the convex hull.

2. **Local operation**. It affects the simplicity of these algorithms. The larger the local region affected by an operation is, the more difficult it is to maintain the simplicity of this operation and the algorithms. Graham's scan, Lawson's flip algorithm, and Rajan's and Edelsbrunner-Shah's flip algorithms use flip as their local operation, while star splaying uses splaying, an operation that modifies a whole star. The affected region of a splaying is usually much larger than that of a flip, and this makes star splaying more complicated than the flip algorithms.

3. **Execution order**. An algorithm with *sequential execution order* has to apply local operations in a certain sequence. Such algorithms include Rajan's flip algorithm and Graham's scan. In contrast, Lawson's flip algorithm, Edelsbrunner-Shah's flip algorithm and star splaying have *free execution order*. Algorithms with free execution order offer some degree of freedom to choose the next local operation from many applicable ones, and therefore can be performed in parallel machines like the GPU, as long as local operations do not conflict. An algorithm with sequential execution order may need to spend extra effort to find the next operation to perform, making the algorithm less efficient than one with free execution order. For example, Rajan's flip algorithm needs to additionally maintain a priority queue, and therefore it has higher time complexity than Edelsbrunner-Shah's flip algorithm.

These aspects provide good measures for evaluating algorithms of local transformation. In general, it is desirable yet more difficult to design algorithms with more general inputs, higher workable dimensions, simpler local operations and free execution order. It is almost impossible to design an algorithm while taking good care of

Table 3.1: The workable dimension, input structure, local operation, and execution order of some existing algorithms of local transformation.

Algorithm	Dimension	Input	Operation	Order
Graham's scan	2	star-shaped polygon	2-1 flip	sequential
Lawson's flip	2	triangulation	2-2 flip	free
Rajan's flip	*any*	"RT+1" triangulation	flip	sequential
Edelsbrunner-Shah's flip	*any*	"RT+1" triangulation	flip	free
Star splaying	*any*	a set of stars*	splaying	free

*The star of each vertex v must enclose a vertex that lexicographically precedes v.

all these aspects. One needs to make a balance among them based on requirements from practice. Table 3.1 summarizes the properties of the algorithms discussed above according to these aspects. The types of flips used in Rajan's and Edelsbrunner-Shah's algorithm depend on the particular dimension they work in.

We emphasize the advantages of algorithms of local transformation. First, they are usually simple (to be presented or implemented) because of the simplicity of the local operations they use. For example, one can use one sentence to describe Lawson's flip algorithm: pick a locally non-Delaunay edge to flip until Delaunay triangulation is obtained. Second, with free execution order, they can possibly be adapted to parallel machines for the data parallelism provided by local operations. We show a simple parallel workflow for implementing these algorithms in the next section. Third, they can be very efficient when their inputs are close to the desired results, though their worst-case time complexity may be worse than other algorithms.

3.2 Framework

An algorithm of local transformation consists of the following elements:

Geometric structure. The algorithm works on a geometric structure, continuously updating it until it becomes the desired result. The input and output are two instances of the geometric structure. As the base of all the other elements described below, the geometric structure of the algorithm must be explicitly defined and maintained during the whole procedure in order to guarantee the validity of the other elements. For example, the geometric structure of Graham's scan is star-shaped polygon, which is maintained during the whole execution.

Local operation. The algorithm uses local operation to update its geometric structure. Each local operation changes a local configuration of the geometric structure, so it is designed based on that structure. For example, Lawson's flip algorithm uses flip to transform triangulations, while star splaying uses splaying to update stars. An

algorithm of local transformation may contain one or more types of local operation.

Criterion. The algorithm must apply local operations under some rules in order to compute the desired result. The criterion of the algorithm is a set of rules to guide the application of local operations; it provides a direction for the algorithm to transform the geometric structure. For example, the criterion of Lawson' flip algorithm is to flip all locally non-Delaunay edges. This criterion helps to transform the triangulation closer to its Delaunay triangulation, which has no locally non-Delaunay edges.

Local check. The algorithm checks the validity of a local operation according to two conditions: (1) whether the local operation destroys the geometric structure, and (2) whether it does not satisfy the criterion. Checking for the first condition is usually a local task, while checking for the second one may not be local when the criterion utilizes global information. We call this element local check. Only after a local check can the corresponding local operation be applied. For example, Lawson's flip algorithm flips an edge only after assuring that this edge is locally non-Delaunay and flipping it does not destroy the triangulation.

Candidate. A candidate of the algorithm is a piece of the geometric structure that provides sufficient data for a local operation and its corresponding local check. A candidate is called *valid* if it passes the local check; otherwise it is *invalid*. The local operation on a valid candidate can be applied in the algorithm. For example, an internal edge with its induced subcomplex is a candidate in Lawson's flip algorithm, and is valid if it is flippable and locally non-Delaunay. Note that an algorithm with free execution order usually has multiple valid candidates at any moment, while one with sequential execution order may only have one valid candidate.

To sum up, these elements are strongly related to one another. The geometric structure provides candidates that are used for local operations. The algorithm tests the validity of candidates by local checks based on its criterion. Local operations update the geometric structure, creating new candidates for the further computation.

The correctness of an algorithm of local transformation depends on two arguments: the argument of termination and the argument of final state. The algorithm terminates when there are no more valid candidates. To promise the termination, the algorithm should avoid the infinite re-creation of the same candidates. Particularly, the criterion of the algorithm should provide a direction for local operations so that a candidate is only re-created a constant number of times. When the algorithm terminates, the final state of its geometric structure should be the desired result. We say the algorithm *gets stuck* if it terminates at a wrong result. To promise correct final state, we should design the criterion of the algorithm so that the final state is the desired output if and only if no local operations are further required by the criterion.

A general sequential workflow for implementing algorithm of local transformation is shown in Figure 3.1. A container of candidates is maintained in the workflow. The choice of containers depends on the type of the execution order: if the algorithm allows

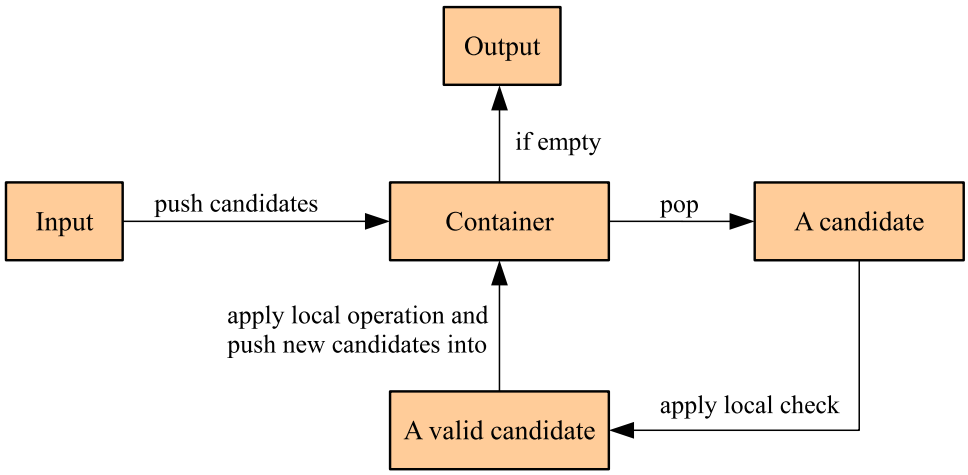


Figure 3.1: A sequential workflow for algorithm of local transformation.

free execution order, a queue or a stack is sufficient; otherwise, the container is usually a priority queue to find the correct order of the candidates to process. Given such an algorithm and its input, we initially extract all the candidates from the input and put them into the container. Each time a candidate is removed from the container and locally checked according to the criterion. If the candidate is invalid, we ignore it and continue with another. Otherwise, we apply local operation on it, creating new candidates and putting them into the container for further consideration. The algorithm repeatedly handles each candidate and terminates after the container is empty.

Figure 3.2 shows a general parallel workflow for implementing algorithm of local transformation that allows free execution order. Given such an algorithm and its input, we organize in an array all the candidates obtained from the input. We run the algorithm by rounds, in each of which a batch of candidates are handled. Particularly, we apply a local check on each candidate in parallel according to the criterion of the algorithm, and collect those valid candidates into another array. Then we apply a local operation on each valid candidate, and collect all the newly created candidates, both also in parallel. The algorithm iteratively handles batches of candidates until no valid candidates exist. Note that some local operations may conflict with one another if the local regions affected by them overlap. In this case we have to add another parallel procedure to avoid this.

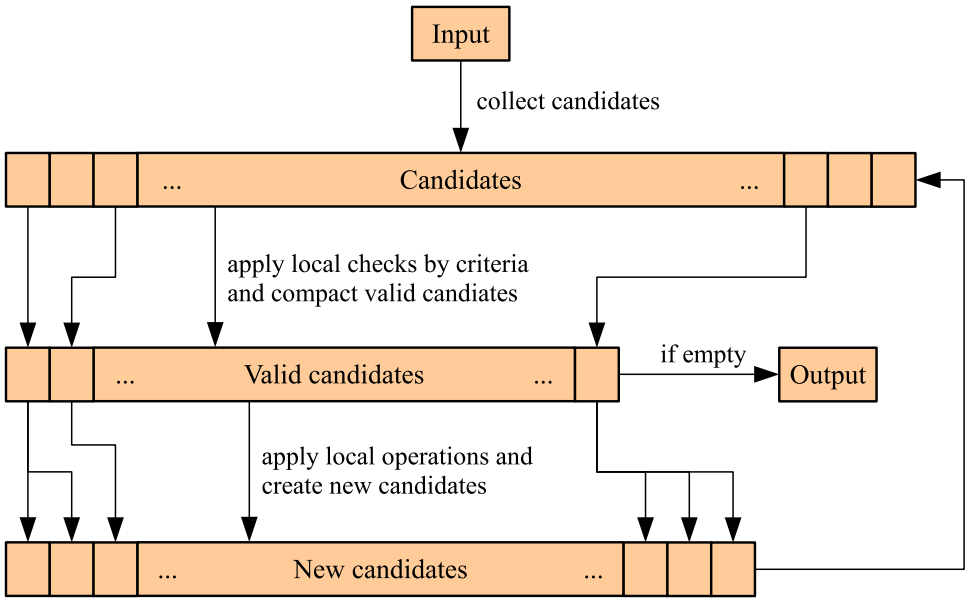


Figure 3.2: A parallel workflow for algorithm of local transformation.

3.3 Examples

3.3.1 Lawson's Flip Algorithm

Lawson's flip algorithm [Law72, Law77] transforms an arbitrary triangulation of a point set S in $\mathbb{R}^2$ into $\mathcal{DT}(S)$ by flips. The **geometric structure** is a 2D triangulation and the **criterion** is to flip all locally non-Delaunay edges. A 3-1 edge is always locally Delaunay; thus the **local operation** is 2-2 flip. As an unflippable 2-2 edge is intrinsically locally Delaunay, the **local check** is only to test whether an edge is locally Delaunay or not. The **candidate** is an edge with its two incident triangles. A 2-2 flip removes a locally non-Delaunay edge and creates a locally Delaunay one. Lawson's flip algorithm iteratively selects a locally non-Delaunay edge to flip until all the edges are locally Delaunay.

We explain Lawson's flip algorithm in the lifted space in order to prove its monotonicity. Let S^+ be the lifted point set of S by the function $(p_1, p_2, ..., p_d) \rightarrow (p_1, p_2, ..., p_d, \sum p_i^2)$. We then obtain the lifted triangulation $\mathcal{T}(S^+)$ by lifting the vertices of $\mathcal{T}(S)$. Setting a fixed virtual point v^+ with a height larger than those of points in S^+, we construct a polyhedron $\mathcal{P}$ using the triangles of $\mathcal{T}(S^+)$ and the triangles formed by v^+ and the boundary edges of $\mathcal{T}(S^+)$. Obviously $\mathcal{P}$ is free of self-intersection and thus its volume is well defined. When a 2-2 flip is applied on $\mathcal{T}(S)$, and thus $\mathcal{T}(S^+)$, in Lawson's flip algorithm, a reflex edge in $\mathcal{P}$ is removed and a convex edge is created. Since no self-intersection is created, the volume of $\mathcal{P}$ increases and thus no edges that are removed can reappear. Therefore, the termination of Lawson's flip algorithm is guaranteed. At the final state, $\mathcal{T}(S)$ does not contain any locally non-Delaunay edge

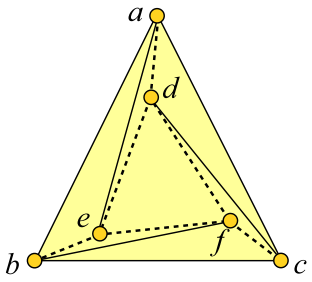


Figure 3.3: The weight of a, b and c is w_1 and the weight of d, e and f is w_2; w_1 is much larger than w_2 so that only the dashed edges are locally non-regular. Unfortunately, all these edges are unflippable.

and thus is $\mathcal{DT}(S)$ by Theorem 2.1. The time complexity of Lawson's flip algorithm is $\mathrm{O}\big(|S|^2\big)$ because a removed edge never reappears in the triangulation.

Having free execution order, Lawson's flip algorithm can be implemented on the GPU following the workflow in Figure 3.2. Specifically, all the edges of its input are organized in an array. In each round a thread assigned to an edge checks whether the edge is locally non-Delaunay. Because the induced sub-complexes of some edges overlap, the flips on them conflict and cannot be applied concurrently. To avoid this, a thread assigned to a locally non-Delaunay edge uses the atomic minimum operation to label the two triangles incident to the edge with its index. After a global synchronization, such a thread flips its edge if the two triangles incident to the edge are still labeled with its index. Since only up to two threads access to the same memory location, the usage of the atomic operation does not affect the efficiency much. This technique is described in [NHS11, QCT12].

One may feel tempted to use Lawson's flip algorithm to transform a triangulation of a weighted point set in $\mathbb{R}^2$ to its regular triangulation, by picking an arbitrary edge that is locally non-regular and flippable to flip until the regular triangulation is obtained. However, this approach can get stuck at a local optimum as shown by the counter-example in Figure 3.3 [ES92]. In the triangulation in this figure, all the locally non-regular edges (shown in dashed) are unflippable.

3.3.2 Star Splaying Algorithm

Star splaying [She05] computes convex hull from a set of convex stars in an arbitrary dimension. Here we only introduce its 3D version. Given the star of a vertex v, call the vertices appearing in the star the *neighbors* of v. The star of v becomes *dead* when a newly inserted point is beyond all its triangles; a star that is not dead is *alive*. In the case when the star of v is dead, v is proved to be non-extreme by the newly inserted point and all the neighbors of v, which altogether compose the *death certificate* of v and its star. The algorithm always ensures each star is convex unless it is dead. The

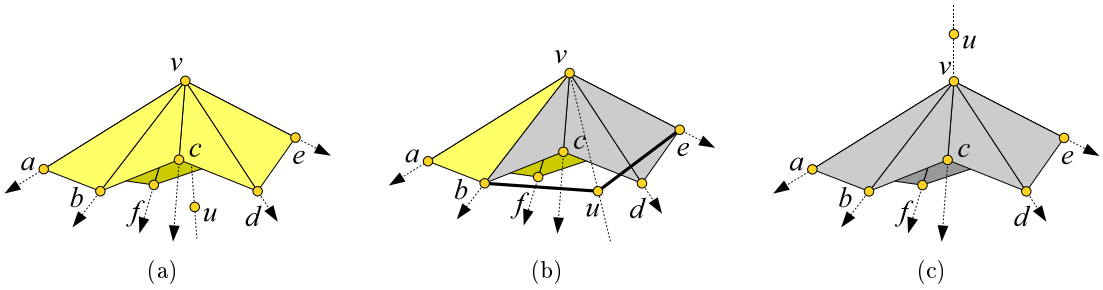


Figure 3.4: The star of v has 6 neighbors and star splaying intends to insert u into it. (a) The insertion fails if u is beneath all the triangles of the stars of v. (b) The insertion is successful if u is beyond some but not all the triangles. Some facets (in gray) are replaced by two new triangles incident to u. (c) The insertion makes a dead star if u is beyond all the triangles.

geometric structure of star splaying is a collection of convex stars and dead stars.

Two stars may not agree with each other. For example, the star of a vertex u contains two triangles $\triangle uvs$ and $\triangle uvt$ but the star of v does not contain both of them. The **local check** of star splaying is to check for *inconsistency*, which is the circumstance that a triangle appears in the stars of some but not all of its vertices. Star splaying uses splaying as its **local operation** to eliminate the inconsistency between stars. Splaying means inserting a point into a convex star. There are three possible cases when a point is inserted into a star. In Case 1, the point is beneath all the triangles of the star, and thus the insertion fails. In Case 2, the point is beyond some but not all the triangles of the star, and thus the insertion succeeds and the star splays (like an umbrella). In Case 3, the point is beyond all the triangles of the star, and thus the star becomes dead. Figure 3.4 illustrates these three cases. The **candidate** of star splaying is a pair of stars, or more accurately, an edge of a star. After a successful insertion, newly created edges need to be checked for inconsistency, and when the star of a vertex becomes dead, all the edges incident to that vertex in other stars also need to be checked.

The **criterion** of star splaying, named *inconsistency enforcement*, instructs how to apply splaying to eliminate inconsistency on an edge. Given an edge $\overline{vu}$, the algorithm either promises that it appears in the stars of both v and u with the same incident triangles, or removes it from both stars. Assuming that $\overline{vu}$ is in the star of v, there exist two cases based on the status of the star of u. In the first case, u has a dead star. The algorithm inserts the death certificate of u into the star of v, which will surely remove $\overline{vu}$ from that star. In the second case, u has a convex star. Insert v into the star of u if it does not have $\overline{vu}$. If the insertion fails, find three vertices in the star of u that prohibits the insertion and insert them into the star of v to remove $\overline{vu}$. Now $\overline{vu}$ exists in the star of u, but both stars disagree on the triangles incident to $\overline{vu}$. Insert the two link vertices of $\overline{vu}$ in one star into the other for both directions. The two stars are then splayed and become agree about $\overline{vu}$.

Star splaying always terminates because the convex stars monotonically splay and thus a point cannot be inserted successfully into the same star more than once. When the algorithm terminates, all the convex stars are consistent with one another. By enforcing a precondition that the input star of each vertex encloses a vertex lexicographically preceding it, the set of stars at the final state of the algorithm provably determines the convex hull.

Star splaying can be implemented in both the sequential and the parallel workflows in Section 3.2. In the sequential workflow, the edges of all stars are pushed into a container, e.g. a queue, and checked for inconsistency. An edge with inconsistency results into multiple point insertions. The newly created edges generated by these insertions are then pushed into the container. In the parallel workflow, the edges of all stars are organized in an array A. In each round a thread assigned to an edge of A checks its inconsistency. The edges with inconsistency are arranged into an array B. One thread assigned to an edge in B creates at most four *virtual insertions* in the format $\langle s, p \rangle$, which means point p should be inserted into star s. All virtual insertions are arranged in an array C. Since insertions for the same star cannot be executed in parallel, C is first sorted by s. Then each set of virtual insertions with the same s is assigned to a thread and processed sequentially. These insertions modify A.

There are two points to note about star splaying. First, star splaying works efficiently when its initial stars are close to the final stars of the convex hull. However, it is unclear how to efficiently construct such initial stars from a given point set. Second, star splaying can be performed in parallel, since inconsistency is locally checked and stars are independently splayed. As described previously, a batch of insertions into the same star handled by a single thread must be done sequentially. As such, threads in the same iteration may have different amount of work, which is not good for parallel machines such as the GPU.

CHAPTER 4

Parallel Flip Algorithms for 2D Convex Hull

This chapter studies the 2D convex hull problem and designs two parallel algorithms to compute 2D convex hull based on our new understanding of the traditional Graham's scan algorithm.

Section 4.1 reinterprets the scanning stage of Graham's scan as a flip algorithm called *flip-pop* that has free execution order. Under this reinterpretation, one can develop a straightforward parallel version of Graham's scan by first sorting all the input points by their angles w.r.t. an extreme point to create a star-shaped polygon and then applying flip-pop. However, the polygon created by the sorting may not be star-shaped in practice due to the numerical inexactness caused by the computation of angles. In Section 4.2, we eliminate this numerical inexactness by constructing an upper and a lower star-shaped chains separately instead of a star-shaped polygon.

Section 4.3 presents an alternative algorithm that applies flip-pop to compute 2D convex hull in parallel. The algorithm computes the two chains via incrementally inserting the points instead of sorting, attempting to remove many non-extreme points during insertion. Section 4.4 presents the experimental results of these two parallel algorithms. Compared with the well-known implementations Qhull and CGAL on the CPU, our proposed algorithms running on the GPU are 40 times faster.

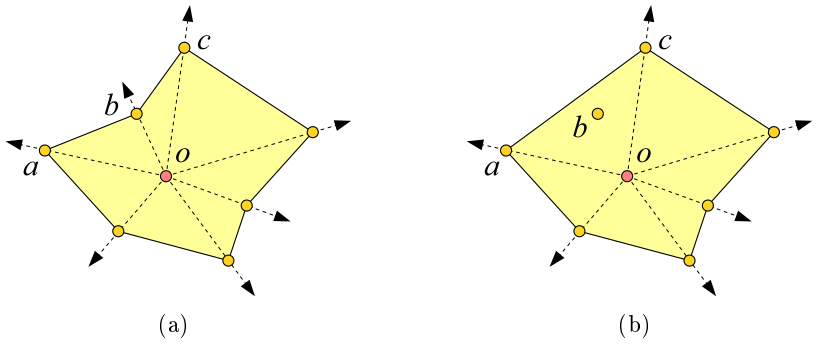


Figure 4.1: (a) A star-shaped polygon w.r.t. a kernel point o. (b) Applying a 2-1 flip on the vertex b removes b and increases the area of the polygon. The polygon remains star-shaped w.r.t. o after the flip.

4.1 Flip-pop

Graham's scan is a competitive algorithm with Quickhull for computing 2D convex hull on the CPU. However, it is hard to be adapted in parallel machines such as the GPU because of its strictly sequential scanning stage. We show that this stage can be re-fitted as a flip algorithm with free order of execution, named *flip-pop*.

We describe the flip-pop algorithm using our framework of local transformation. The **geometric structure** of flip-pop is a star-shaped polygon $\mathcal{P}$. See Figure 4.1(a) for an example of star-shaped polygon. The **local operation** is 2-1 flip, and the **candidate** is a vertex associated with its two incident edges. The **criterion** is designed aiming to increase the area of $\mathcal{P}$ to the maximum: it removes a vertex v by the local operation if v lies inside the triangle formed by the two neighbors of v and the kernel point. Note that the polygon is still star-shaped w.r.t. the same kernel point after removing v; thus the **local check** only tests if a vertex is inside the corresponding triangle. For example, the polygon in Figure 4.1(a) is transformed into the one in Figure 4.1(b) by a 2-1 flip on vertex b as b lies inside $\triangle oac$.

Algorithm 1 and Algorithm 2 show the pseudocode of flip-pop under the sequential and parallel workflows in Section 3.2 respectively. In contrast to Algorithm 1, Algorithm 2 has extra codes for avoiding the conflict of memory read and write due to the parallel execution. Specifically, we partition one round of flipping into two steps, between which a global synchronization is set (Line 9). In the first step (Line 5–8), the thread assigned to a vertex p_i checks whether p_i lies inside the triangle as mentioned in the criterion and, if yes, labels p_i and its two neighboring vertices with the index i. In the second step (Line 10–12), the thread assigned to a vertex p_i removes p_i by 2-1 flip only if p_i and its two neighboring vertices are still labeled with the index i.

The flip-pop algorithm always terminates in $\mathrm{O}(n)$ flips where n is the number of the vertices of $\mathcal{P}$, because any vertex that is removed cannot reappear. When flip-pop

Algorithm 1: Flip-pop under the sequential workflow

input : a star-shaped polygon $\mathcal{P}$ w.r.t. s
output: $\mathcal{CH}(\mathcal{P})$

label all the vertices of $\mathcal{P}$ as unchecked
repeat
  foreach vertex p_i labeled as unchecked **do**
    let $\overrightarrow{p_jp_i}$ and $\overrightarrow{p_ip_k}$ be the two edges incident to p_i
    if $p_i \in \triangle sp_jp_k$ **then**
      apply 2-1 flip on p_i
      label p_j and p_k as unchecked
    else
      label p_i as checked
until all vertices are labeled as checked

Algorithm 2: Flip-pop under the parallel workflow

input : a star-shaped polygon $\mathcal{P}$ w.r.t. s
output: $\mathcal{CH}(\mathcal{P})$

label all the vertices of $\mathcal{P}$ as unchecked
repeat
  foreach vertex p_i labeled as unchecked **do in parallel**
    let $\overrightarrow{p_jp_i}$ and $\overrightarrow{p_ip_k}$ be the two edges incident to p_i
    if $p_i \in \triangle sp_jp_k$ **then**
      use atomic minimum operation to label p_j, p_i and p_k with i
    else
      label p_i as checked
    global synchronization
    if p_j, p_i and p_k are all labeled with i **then**
      apply 2-1 flip on p_i
      label p_j and p_k as unchecked
until all vertices are labeled as checked

terminates, the internal angle of each vertex is smaller than π. Because the polygon is star-shaped throughout the execution and no non-extreme points are removed, the polygon at the final state is the convex hull.

Note that flip-pop is a hill-climbing algorithm from three different points of view. Every flip applied in the algorithm reduces the number of vertices, increases the area of the polygon, and decreases the sum of the inner angles. These multiple measures of monotonicity reflect the simplicity of flip-pop, and this simplicity is mainly because the problem is in $\mathbb{R}^2$. Indeed, flip-pop is difficult to be extended to compute 3D convex hull, because it may get stuck when using these measures to guide flipping.

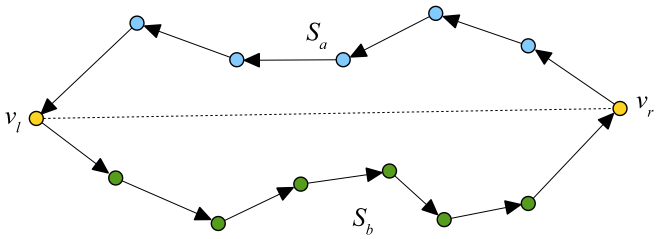


Figure 4.2: An illustration of parallel Graham's scan. The blue nodes are S_a=$\{p \mid p \in S$ and p is above $\overline{v_l v_r}\}$ and the green nodes are S_b=$\{q \mid q \in S$ and q is below $\overline{v_l v_r}\}$, where S is the set of input points and $\{v_l, v_r\}$ are the leftmost and rightmost points of S.

4.2 Parallel Graham's Scan

With flip-pop, Graham's scan can be straightforwardly adapted into a parallel version consisting of a constructing stage to build star shaped polygon and a flipping stage to compute convex hull. In the constructing stage, the bottommost point v of S is selected as the kernel point, and all the points are sorted in ascending order of their polar angles in the polar coordinate system whose origin is v. A star-shaped polygon w.r.t. v is built from this sorted array of points. In the flipping stage, flip-pop is used to transform the star-shaped polygon to the convex hull in parallel.

However, note that the polygon built by sorting may not be star-shaped in practice because of the numerical inexactness created when computing polar angles. If so, flip-pop cannot transform this polygon to the convex hull. To overcome this problem, we use a different polygon that is created from sorting the input points by their original coordinates, and thus avoid the mentioned numerical inexactness. We simply name the proposed algorithm *parallel Graham's scan*.

The key idea of the parallel Graham's scan is to use the infinite point instead of an input point as the kernel point, and build an upper star-shaped chain w.r.t. the kernel point at $(0, -\infty)$ and a lower star-shaped chain w.r.t. the kernel point at $(0, \infty)$. Specifically, we use $\overline{v_l v_r}$ to partition $S \setminus \{v_l, v_r\}$ into S_a=$\{p \mid p \in S$ and p is above $\overline{v_l v_r}\}$ and S_b=$\{q \mid q \in S$ and q is below $\overline{v_l v_r}\}$, where v_l and v_r are the leftmost and rightmost points of S respectively; see Figure 4.2. The upper chain is built via sorting $S_a \cup \{v_l, v_r\}$ by their x-coordinates in descending order, while the lower chain is built via sorting $S_b \cup \{v_l, v_r\}$ by their x-coordinates in ascending order. The two chains thus only intersect at v_l and v_r. By reading the upper chain from v_r to v_l and then the lower chain from v_l to v_r, we obtain a counter-clockwise polygon which is free of self-intersection. This polygon is not necessarily star-shaped but can still be transformed to the convex hull by flip-pop.

Algorithm 3 shows the pseudocode of the parallel Graham's scan. In Line 1, we compute the leftmost and rightmost points of the input point set S using reduction primitives. We then collect $S_a = \{p \mid p \in S$ and p is above $\overline{v_l v_r}\}$ and $S_b = \{q \mid q \in S$

Algorithm 3: Parallel Graham's Scan

input : a set S of points in $\mathbb{R}^2$
output: $\mathcal{CH}(S)$

1 $\{v_l, v_r\} \leftarrow$ the leftmost and rightmost points of S
2 $S_a \leftarrow \{p \mid p \in S$ and p is above $\overline{v_l v_r}\}$
3 $S_b \leftarrow \{q \mid q \in S$ and q is below $\overline{v_l v_r}\}$
4 sort S_a by x-coordinate in descending order
5 sort S_b by x-coordinate in ascending order
6 $\mathcal{P} \leftarrow \langle v_r, S_a, v_l, S_b \rangle$
7 call parallel flip-pop in Algorithm 2

and q is below $\overline{v_l v_r}\}$ by first labeling the points of S according to their relation with $\overline{v_l v_r}$ and then call a partitioning primitive. In Line 4 and Line 5, we sort S_a and S_b by x-coordinates. In Line 6, we create the polygon $\mathcal{P}$ by concatenating v_r, S_a, v_l and S_b. Finally, we apply the parallel flip-pop on $\mathcal{P}$ to compute $\mathcal{CH}(S)$; see Algorithm 2.

The correctness of the parallel Graham's scan depends on two facts. First, the polygon created by sorting is free of intersection. Second, flip-pop (Line 7) transforms it into $\mathcal{CH}(S)$. To prove the second fact, we note that v_l and v_r are extreme points in $\mathcal{CH}(S)$, and thus no flips can be applied on them. Therefore, the flip-pop procedure works separately for the upper and the lower star-shaped chains, and the combined result is $\mathcal{CH}(S)$.

In order to reduce the amount of costly global memory access as well as the number of global synchronization, we try to remove more non-extreme vertices in each iteration by using the shared memory of the GPU. Particularly, each block of threads on the GPU loads a consecutive chunk of vertices plus two more on its left and right into shared memory, and repeatedly applies Algorithm 2 to remove non-extreme vertices in that chunk until no more can be identified. Figure 4.3 shows the difference of executing one iteration with and without using shared memory. Note that this difference will be more apparent when the polygon size and chunk size become larger.

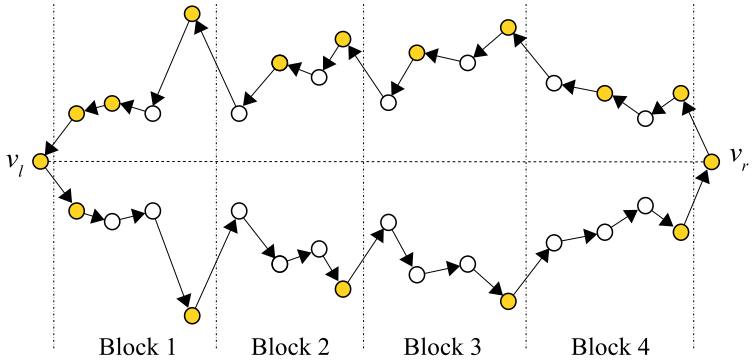


Figure 4.3: The polygon has the same upper and lower chains (only mirrored). We apply one iteration of flip-pop, using shared memory on the lower chain and not on the upper one. Each chunk of vertices on the shared memory has size 4. The result is that 11 non-extreme vertices (white nodes) are removed from the lower chain as compared to only 7 from the upper chain.

4.3 Parallel Graham's Scan via Insertion

The main drawback of the parallel Graham's scan is that it always sorts all the points of S, even if most of them are non-extreme. In this section, we use incremental insertion to build the two star-shaped chains instead of sorting, and then apply flip-pop to transform them to the convex hull of S. This algorithm is named *parallel Graham's scan via insertion*. A purely incremental insertion approach is presented in [SRKN11]. However, their approach may output a wrong result, because it does not always find the correct furthest point for insertion due to numerical error.

Given an oriented edge $\overrightarrow{ab}$ that is non-vertical and a point c in $\mathbb{R}^2$, we say c is *beyond* $\overrightarrow{ab}$ if $\triangle abc$ is clockwise; otherwise c is *beneath* $\overrightarrow{ab}$. When $c.x \in [a.x, b.x]$, we further say c is *right beyond* or *right beneath* $\overrightarrow{ab}$. Given oriented edges $\overrightarrow{v_l v_r}$ and $\overrightarrow{v_r v_l}$ where v_l and v_r are respectively the leftmost and rightmost points of S, each point of $S \setminus \{v_l, v_r\}$ is right beyond one of them. We incrementally insert the points right beyond $\overrightarrow{v_r v_l}$ to form the upper star-shaped chain, while insert the points right beyond $\overrightarrow{v_l v_r}$ to form the lower star-shaped chain. During the insertion, any point that is proved to be non-extreme is excluded from S and will not be considered.

Algorithm 4 shows the pseudocode of the parallel Graham's scan via insertion. In Line 1–6, we do the initialization including computing the leftmost point v_l and the rightmost point v_r, excluding them from S, building $\mathcal{P}$ with these two points, and associating each point of S with one edge of $\mathcal{P}$. We then repeatedly insert the remaining points of S into $\mathcal{P}$. Each iteration consists of an insertion phase and an updating phase. In the insertion phase (Line 8–11), for each edge $\overrightarrow{pq}$ of $\mathcal{P}$, one point is selected from all the points right beyond $\overrightarrow{pq}$ and inserted into $\overrightarrow{pq}$; this point is excluded from S. In the updating phase (Line 12–17), each point originally associated with $\overrightarrow{pq}$ is either associated with a new edge or excluded from S. More specifically, for each point

Algorithm 4: Parallel Graham's Scan via Insertion

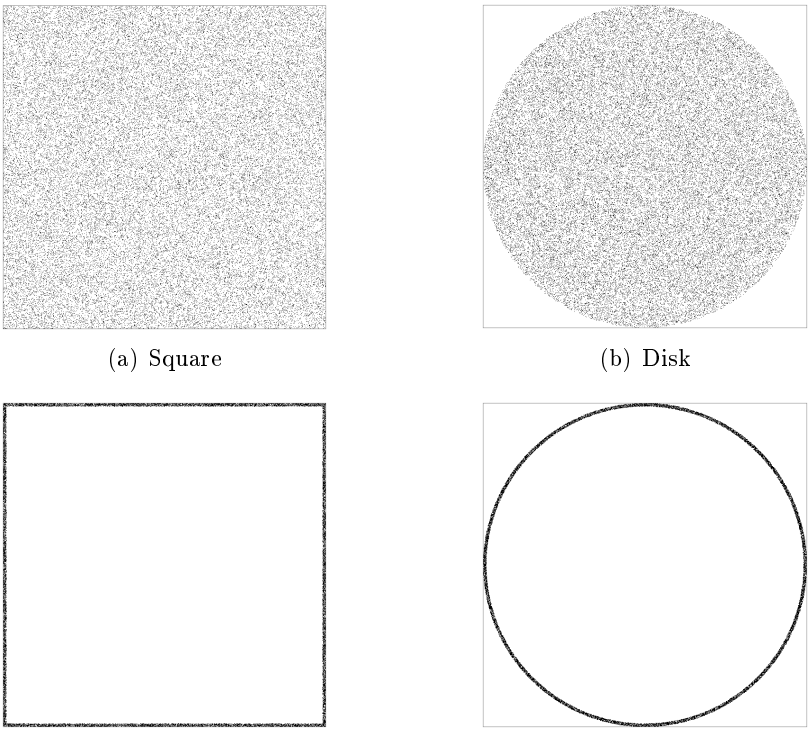
input : a set S of points in $\mathbb{R}^2$
output: $\mathcal{CH}(S)$

1 $\{v_l, v_r\} \leftarrow$ the leftmost and rightmost points of S
2 $S \leftarrow S \setminus \{v_l, v_r\}$
3 $\mathcal{P} \leftarrow \langle v_r, v_l \rangle$
4 **foreach** point p in S **do**
5 **if** p is right beyond $\overrightarrow{v_r v_l}$ **then** associate p with $\overrightarrow{v_r v_l}$
6 **else** associate p with $\overrightarrow{v_l v_r}$
7 **repeat**
8 **foreach** edge $\overrightarrow{pq}$ of $\mathcal{P}$ **do**
9 choose a point r associated with $\overrightarrow{pq}$
10 replace $\overrightarrow{pq}$ with $\overrightarrow{pr}$ and $\overrightarrow{rq}$
11 $S \leftarrow S \setminus \{r\}$
12 **foreach** point u in S **do**
13 let $\overrightarrow{pq}$ be its associated edge, and r be the point inserted into $\overrightarrow{pq}$
14 **if** $(u.x - p.x)(u.x - r.x) \leq 0$ **then** $e \leftarrow \overrightarrow{pr}$
15 **else** $e \leftarrow \overrightarrow{rq}$
16 **if** u is beyond e **then** associate u with e
17 **else** $S \leftarrow S \setminus \{u\}$
18 **until** $S = \emptyset$
19 call parallel flip-pop in Algorithm 2

u, let $\overrightarrow{pq}$ be the edge it is originally associated with, and r be the point inserted into $\overrightarrow{pq}$ in the current insertion phase. Then u must lie in one of the vertical bands defined by $\overrightarrow{pr}$ and $\overrightarrow{qr}$; let e be the edge defining the vertical band containing u (Line 14–15). If u is beneath e, then u is removed because it lies inside the quadrilateral defined by v_l, v_r and e; otherwise it is associated with e. After S becomes empty, we use flip-pop to compute $\mathcal{CH}(S)$ (Line 19).

Note that the polygon created by the constructing stage is free of self-intersection and consists of two star-shaped chains. Furthermore, all the points that are not inserted into the polygon are proved to be non-extreme. Therefore, the parallel Graham's scan via insertion is provably correct.

In our GPU implementation, we intend to insert the furthest one among all the points associated with the same edge, as this can quickly increase the area of $\mathcal{P}$ and remove more non-extreme points. In fact, if we always insert the furthest point, then the resulting polygon of the constructing stage is already the convex hull. However, that requires atomic operation and is also affected by numerical inaccuracy. Instead, we only find an approximately furthest point to insert by simply overwriting the maximum distance in the global memory and picking a point whose distance is equal to the value in the global memory. This approach does not affect the correctness of the algorithm because of the use of flip-pop.



(a) Square (b) Disk

(c) Frame with thickness 0.01 (d) Circle with thickness 0.01

Figure 4.4: Four distributions of tested data for 2D convex hull.

4.4 Experiment

We implement the parallel Graham's scan (Section 4.2) and the parallel Graham's scan via insertion (Section 4.3) on the GPU, and use *Graham-sorting* and *Graham-insertion* to denote these two implementations. The existing implementations used for comparison include Qhull [Qhu12] and CGAL [CGA12]; the former one implements the Quickhull algorithm, while the latter one implements the algorithm in [AT78].

We generate points randomly with coordinates between $[0.0, 1.0]$. Points are distributed uniformly in four distributions: a square, a disk of radius 0.5, a square frame with thickness of 0.01, and a circle with thickness of 0.01; see Figure 4.4.

Running time

Figure 4.5 shows the running time of Graham-sorting and Graham-insertion when they process millions of points on the four distributions. Their running time increases linearly with the number of input points for all the distributions. By comparing their running time, we get similar results for the square, disk and frame distributions: Graham-insertion is much faster than Graham-sorting when the input size is larger than 4×10^6. This is mainly because the number of extreme points is small, so Graham-insertion can quickly exclude many non-extreme points. In contrast, excluding non-extreme

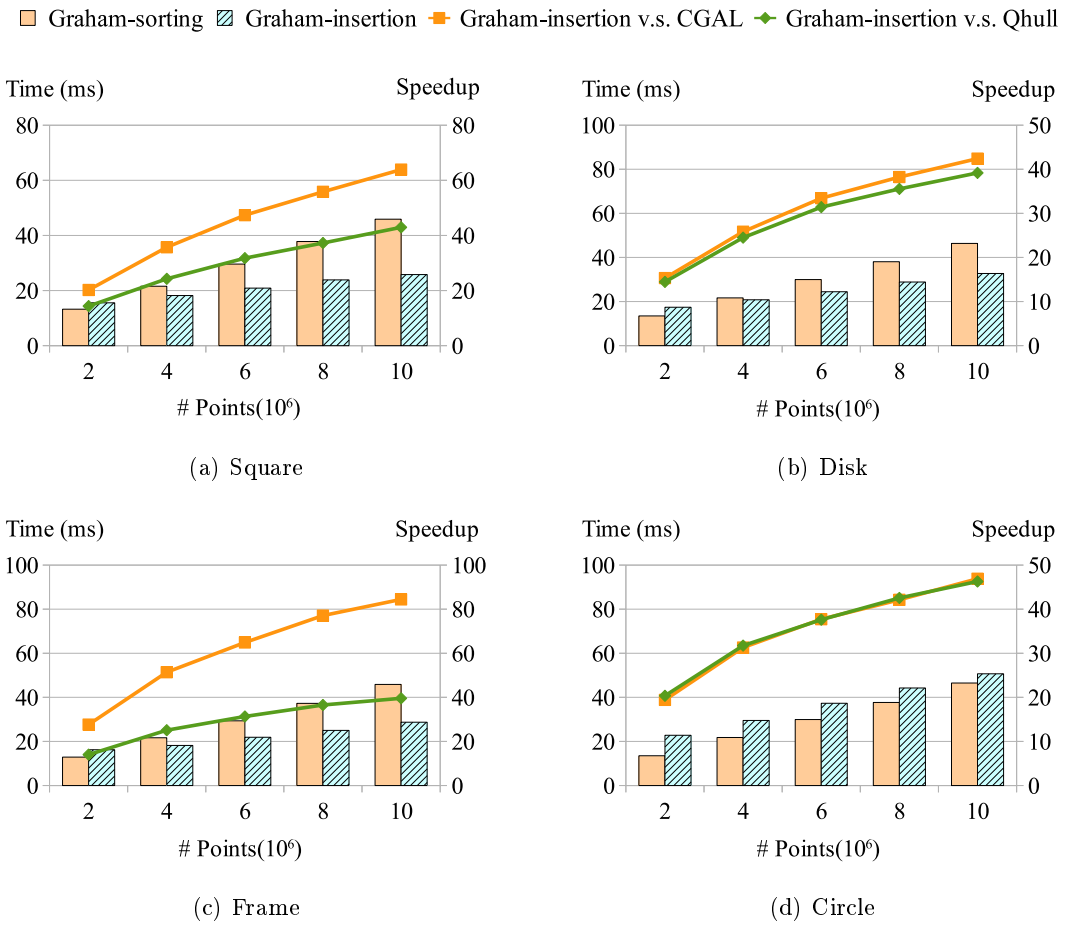


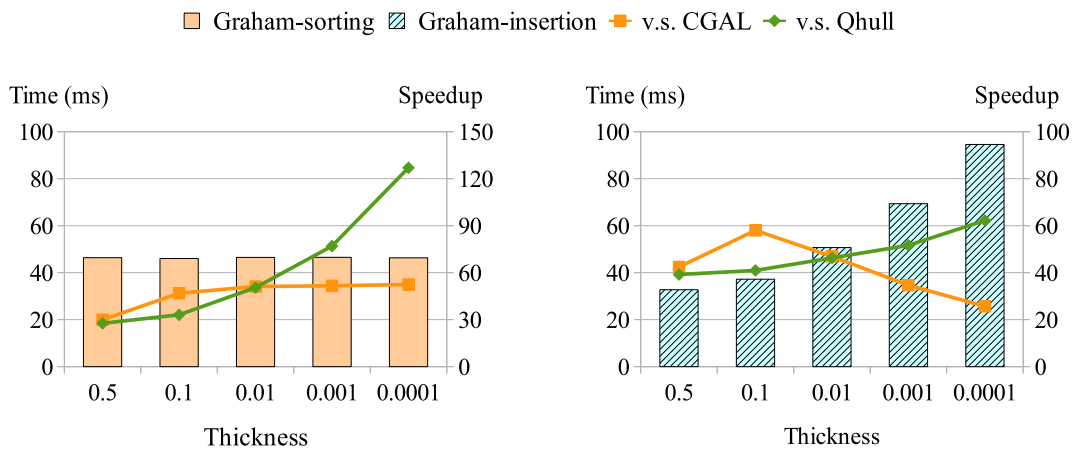
Figure 4.5: Running time and speedups over CGAL and Qhull of Graham-sorting and Graham-insertion for the four distributions of points.

points is not very effective for the circle distribution because of the large number of extreme points. Therefore in this distribution, Graham-sorting runs slightly faster than Graham-insertion for all the input sizes.

Since Graham-insertion performs better in most of the distributions, we use it to compare with CGAL and Qhull, and show the results by the lines in Figure 4.5. The speedup of Graham-insertion over CGAL ranges from 20 to 85 times, and that over Qhull is from 14 to 46 times. The speedup increases as the input size grows. For the disk and circle distributions, CGAL and Qhull have almost the same running time; for the square and frame distributions, Qhull runs much faster than CGAL.

Sensitivity

We test the sensitivity of Graham-sorting and Graham-insertion to the thickness, ranging from 0.5 to 0.0001, of the circle distribution with 10^7 points. The smaller this thickness is, the more difficult it is to remove non-extreme points. The number of extreme points is around 600 for the thickness 0.5, and around 20,000 for the thickness 0.0001.



(a) Graham-sorting

(b) Graham-insertion

Figure 4.6: Running time and speedups over CGAL and Qhull of Graham-sorting and Graham-insertion for a circle distribution of different thicknesses.

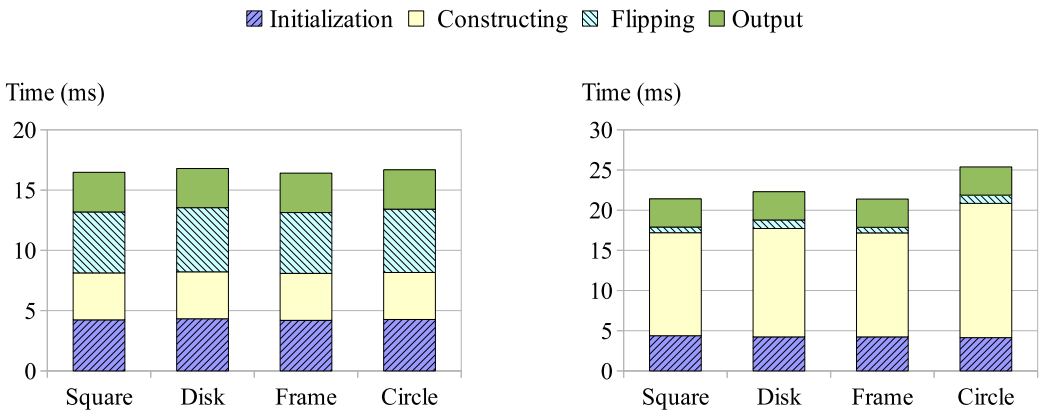
As shown in Figure 4.6(a), the running time of Graham-sorting is almost unchanged when the thickness becomes smaller. The time of the constructing stage of Graham-sorting is mainly decided by the number of input points, which is always 10^7 in this experiment setting. For the flipping stage, decreasing to 600 or 20,000 extreme points from 10^7 input points makes little difference. CGAL is also not sensitive to the thickness: the speedup of Graham-sorting over CGAL keeps at around 50 times when the thickness changes from 0.1 to 0.0001. On the other hand, the speedup of Graham-sorting over Qhull sharply increases and reaches 127 times for the thickness 0.0001. This is because Qhull becomes slower when the input has more extreme points.

The sensitivity of Graham-insertion is illustrated in Figure 4.6(b). When the circle becomes thinner, Graham-insertion needs to perform more insertion rounds in the constructing stage. Meanwhile, the flipping stage is slower as the input star-shaped polygon has more vertices. As such, its speedup over CGAL decreases from 60 to only 25 times. However, its sensitivity is still less than Qhull, since its speedup over Qhull slowly increases as the thickness decreases.

When the thickness of the circle further decreases so that the number of extreme points is around 60,000, Graham-sorting runs 14 times faster than Graham-insertions, and its speedups over CGAL and Qhull are respectively around 52 and 159 times.

Time breakdown

Figure 4.7 shows the time breakdown of Graham-sorting and Graham-insertion for 10^6 points on the four distributions. The initialization and the output stages transfer data between the CPU and the GPU. For both Graham-sorting and Graham-insertion, these two stages take more than 30% of the total running time.



(a) Graham-sorting

(b) Graham-insertion

Figure 4.7: Time breakdown of Graham-sorting and Graham-insertion.

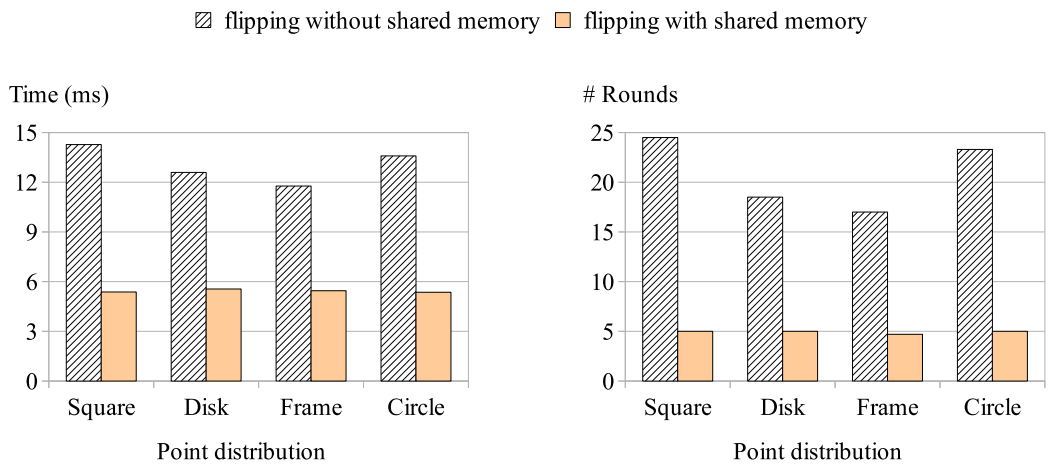
In Graham-sorting, the flipping stage takes around 30% of the total running time for all the distributions. This is reasonable because the constructing stage outputs a polygon containing all the input points, and transforming it to the final convex hull is a substantial work. In contrast, the flipping stage of Graham-insertion only takes a very small part of the total running time. This implies that its constructing stage effectively excludes non-extreme points and outputs a polygon close to convex hull; its flipping stage needs to handle very little work.

Use of shared memory

In the flipping stage of Graham-sorting and Graham-insertion, we employ shared memory to increase the number of non-extreme points removed in each round to decrease the number of flipping rounds. This benefits Graham-sorting more as its flipping stage takes a major percentage of its total running time. Therefore, we run Graham-sorting using 10^6 points on the four distributions, and record the running time of the flipping stage and the number of flipping rounds; see Figure 4.8. By using shared memory, we can reduce the number of flipping rounds by three to five times. However, the running time of the flipping stage is not reduced as much. This is because we have to enforce synchronization in kernels in order to use shared memory, making many threads idle. Consequently, the running time of the flipping stage with shared memory is around 1/3 to 1/2 of that without shared memory.

Limitation

When the distribution of the input point set is not uniform, the constructing stage of Graham-insertion may have many insertion rounds. For example, all the points that are not excluded from S are always associated with the same edge. Then in every



(a) Running time

(b) Flipping rounds

Figure 4.8: Running time and rounds of the flipping stage in Graham-sorting with and without using shared memory.

round only one point is inserted into the polygon. Graham-insertion also becomes not competitive with Graham-sorting when most of the input points are extreme. In contrast, Graham-sorting performs very stable for most of point distributions.

For some star-shaped polygons or chains, the flipping stage, i.e. flip-pop, may be very inefficient. Think about an upper star-shaped chain consisting of two convex sub-chains where all the vertices are non-extreme except for the leftmost and the rightmost ones; however, only the vertex joining the two sub-chains can be locally identified as a non-extreme point. After removing this vertex, one of its two neighbors becomes the new vertex joining two convex sub-chains. In this case, flip-pop can only remove one non-extreme vertex in each iteration.

CHAPTER 5

Flip Algorithm for 2D Regular Triangulation

This chapter goes beyond the traditional way of hill-climbing flipping to arrive at the flip-flop algorithm that solves the 2D regular triangulation problem.

As discussed in Chapter 3, Lawson's flip algorithm is not always able to transform a triangulation of a weighted point set to its regular triangulation, because its greedy nature can lead to a stuck configuration involving redundant vertices. Section 5.1 proposes the *flip-flop* algorithm which can provably identify and remove redundant vertices to reach the solution. When removing redundant vertices, the algorithm may flip, unconventionally, some locally regular edges. Such a flip is distinguished as a "flop", thus the name flip-flop.

The correctness of flip-flop lies in that a locally non-regular and unflippable edge is always incident to a redundant vertex, and thus redundant vertices can be identified locally; see Section 5.2. With this, given a weighted point set, we can first construct an arbitrary triangulation by, for example, incremental insertion and then apply flip-flop to transform it to the regular triangulation. Section 5.3 details such a solution, named ffRT, and Section 5.4 compares the implementations of ffRT on the CPU and the GPU with CGAL. Experimentally, the GPU implementation of ffRT reaches up to 45 times speedup over CGAL.

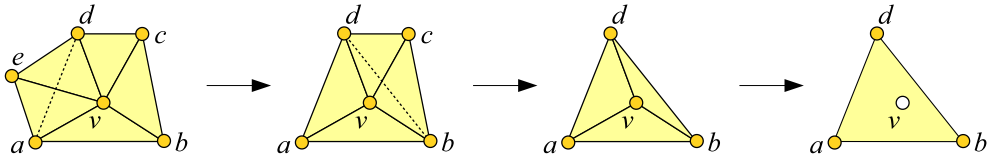


Figure 5.1: To remove v, we successively flip the 2-2 edges $\overline{ve}$ and $\overline{vc}$ to reduce the degree of v to 3. Then, v is removed by a 3-1 flip.

5.1 Flip-flop for 2D Regular Triangulation

Given a point set S in $\mathbb{R}^2$ and a weight set W of S, flip-flop starts with an arbitrary triangulation $\mathcal{T}(S \oplus W)$ that contains all the non-redundant points of S and possibly some redundant ones as its vertices. The **geometric structure** is a 2D triangulation. Besides 2-2 flip as used in Lawson's flip algorithm, the **local operations** include 3-1 flip to remove redundant vertices. When a redundant vertex is identified, we *label* it as *redundant*. Note that labeling is indeed also a local operation; it changes the status of vertices but does not modify the geometric structure. The **candidate** of flip-flop is an edge with its induced-subcomplex, and we commonly refer to an edge as a candidate.

The **criterion** of flip-flop is a combination of two criteria. The first criterion, *V-criterion* (V for Volume), is to flip locally non-regular edges. Using the lifting function $(p_1, p_2) \rightarrow (p_1, p_2, p_1^2 + p_2^2 - w_p)$ where w_p is the weight of p in W, we obtain the lifted point set S^W of S and the lifted triangulation $\mathcal{T}^W(S)$ of $\mathcal{T}(S \oplus W)$ in $\mathbb{R}^3$. The idea in the V-criterion is to lower the 3D surface formed by $\mathcal{T}^W(S)$ by flipping locally non-regular edges. When the lifted triangulation is lowered to the extreme, we get the lower hull of S^W and hence $\mathcal{RT}(S \oplus W)$.

The second criterion, *D-criterion* (D for Degree), aims to reduce the degree of a redundant vertex and finally remove it from $\mathcal{T}(S \oplus W)$. We observe that a series of flips on the edges incident to this vertex can be used to remove it; see Figure 5.1. Given an edge e and its induced-subcomplex $\mathcal{T}_e$, let v be the vertex with the smallest index among all the vertices labeled as redundant in $\mathcal{T}_e$. The D-criterion is to flip e if v is in e; it forbids the flip if v is in $link(e)$; and it gives no decision if v does not exist. The use of the smallest index is to avoid flipping an edge back and forth.

The criterion of flip-flop combines the V- and the D-criterion, giving the D-criterion higher priority: if some vertex in $\mathcal{T}_e$ is labeled as redundant, the D-criterion is used to decide whether to flip e or not; otherwise the V-criterion is used.

The **local check** of flip-flop on an edge e includes one or more of the following tasks: checking the flippability of e to see whether flipping e destroys the geometric structure, a triangulation; checking whether the vertices in $\mathcal{T}_e$ are labeled as redundant for the D-criterion; and checking whether e is locally regular or not for the V-criterion.

Algorithm 5 shows the pseudocode of flip-flop following the sequential workflow of

Algorithm 5: Flip-flop for 2D Regular Triangulation

input : a triangulation $\mathcal{T}$ of a point set S in $\mathbb{R}^2$ and its weight set W
output: $\mathcal{RT}(S \oplus W)$

```
1  label all vertices of T as unknown
2  Q ← { e | e is an edge of T }
3  while Q ≠ ∅ do
4      e ← Q.pop()
5      let e = ab, {c, d} be its link and x ∈ {a, b, c, d} be the redundant vertex with
       smallest index
6      if e is a 3-1 edge then
7          assume a ∈ △bcd w.l.o.g.
8          if a is labeled as redundant or e is locally non-regular then
9              flip e; Q ← {bc, cd, db}
10     else
11         if a ∉ △bcd and b ∉ △acd then
12             if x does not exist and e is locally non-regular then
13                 flip e; Q ← {ac, bc, ad, bd}
14             else if x ∈ {a, b} then
15                 flip e; Q ← {ac, bc, ad, bd}
16         else if e is locally non-regular then
17             assume a ∈ △bcd w.l.o.g.
18             label a as redundant
19             Q ← { e' | e' is an edge of T with a as an endpoint }
```

the local transformation framework in Section 3.2. First of all, all the vertices of $\mathcal{T}$ are labeled as unknown, which means they are not yet labeled as redundant, and all the edges of $\mathcal{T}$ are pushed into $\mathcal{Q}$ for checking. The main loop repeats until no more edges in $\mathcal{T}$ need to be checked, i.e. $\mathcal{Q}$ is empty. Let $e = \overline{ab}$ be an edge popped in one iteration, and let $\{c, d\}$ be its link. We check whether e is a 3-1 or a 2-2 edge. In Case 1, e is a 3-1 edge (Line 6–9); then either a or b must lie inside the triangle formed by the other 3 vertices. Without loss of generality, assume a lies inside $\triangle bcd$. If a is already labeled as redundant or e is locally non-regular (which also implies that a is redundant), we flip e to remove a. In Case 2, e is a 2-2 edge (Line 10–19). If e is flippable, we flip e if either (1) e is locally non-regular and no vertices among $\{a, b, c, d\}$ are labeled as redundant, or (2) one of a and b is labeled as redundant and has the smallest index among all the vertices labeled as redundant in $\{a, b, c, d\}$ (Line 11–15). On the other hand, if e is unflippable and without loss of generality assume that a is inside $\triangle bcd$ (Line 16–19), then it can be proved that a is a redundant vertex; see Lemma 5.1. We hence label a as redundant and push all the edges incident to a into $\mathcal{Q}$.

Note that the algorithm presented above is just one approach of combining the V- and the D-criterion. One variant is to keep flipping according to the V-criterion first,

until no more flips can be done. After that, we can start labeling redundant vertices and use flips due to the D-criterion to remove them. The process is then repeated until all the redundant vertices are identified and removed.

The trivial worst-case time complexity of flip-flop is $O(n^3)$, which comes from the fact that each redundant vertex is identified after $O(n^2)$ flips and then removed by $O(n)$ flips. However, we cannot find any example to prove this complexity is tight. From a theoretical perspective, this bound is unsatisfactory; we should either find a better bound or improve flip-flop to achieve a better worst-case time complexity. In practice, flip-flop works very well as shown in our extensive experiment.

5.2 Proof of Correctness

In this section we prove that flip-flop can transform an arbitrary triangulation $\mathcal{T}(S \oplus W)$ into $\mathcal{RT}(S \oplus W)$. We first show that a redundant vertex can be identified from a 2-2 edge that is locally non-regular and unflippable.

Lemma 5.1. *Any 2-2 edge $e = \overline{ab}$ that is locally non-regular and unflippable is incident to a redundant vertex.*

Proof. Let $\{c, d\}$ be the link of e; See Figure 5.2. Without loss of generality, assume a lies inside $\triangle bcd$. Let a^+, b^+, c^+ and d^+ be the corresponding lifted points in $\mathbb{R}^3$. Because e is locally non-regular, a^+ lies above the plane defined by b^+, c^+ and d^+. Together with the fact that a lies inside $\triangle bcd$, a^+ must be above the lower hull of $\{a^+, b^+, c^+, d^+\}$. Therefore a is a redundant vertex. □

Next, we show that if all the points of S are non-redundant, then $\mathcal{T}(S \oplus W)$ can be transformed into $\mathcal{RT}(S \oplus W)$ by flipping under only the V-criterion. In this case flip-flop is degraded to a hill-climbing flip algorithm similar to Lawson's flip algorithm.

Lemma 5.2. *If all the points of S are non-redundant, any locally non-regular edge of $\mathcal{T}(S \oplus W)$ is a 2-2 flippable edge.*

Proof. By Lemma 5.1 the edge cannot be a 2-2 unflippable edge; otherwise a redundant vertex exists. Similarly it cannot be a 3-1 edge. Therefore, e is a 2-2 flippable edge. □

Theorem 5.3. *Flipping according to the V-criterion transforms $\mathcal{T}(S \oplus W)$ to $\mathcal{RT}(S \oplus W)$ if all the points of S are non-redundant.*

Proof. Recall that $\mathcal{T}^W(S)$ is the lifted triangulation of $\mathcal{T}(S \oplus W)$ using the lifting function. Flipping a locally non-regular edge lowers the surface formed by $\mathcal{T}^W(S)$. Therefore, during a sequence of flips according to the V-criterion, no edge that was removed can appear again. This implies the termination of the algorithm. By Lemma 5.2, at the final state $\mathcal{T}(S \oplus W)$ does not contain any locally non-regular edge, and therefore is $\mathcal{RT}(S \oplus W)$. □

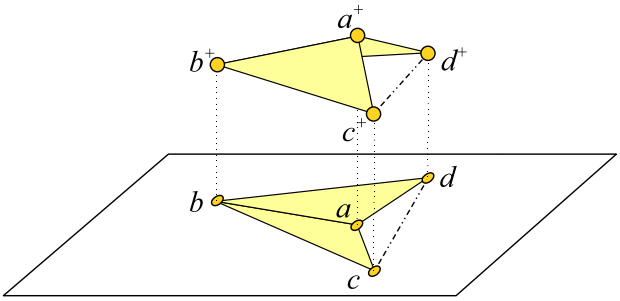


Figure 5.2: $\overline{ab}$ is a 2-2 unflippable edge that is non-regular and a lies inside $\triangle bcd$. Then a is a redundant point.

When $\mathcal{T}(S \oplus W)$ has redundant vertices, flipping under only the V-criterion may get stuck with some redundant vertices remained. In this case the D-criterion must be used to remove redundant vertices. We prove that any vertex not on the boundary is removable by some flips on its incident edges, hence the correctness of flip-flop.

Lemma 5.4. *A vertex v of $\mathcal{T}(S \oplus W)$ not on the boundary can be removed by successively flipping its incident edges.*

Proof. We show that there is always a flippable edge incident to v. Consider three consecutive vertices of a simple polygon; they form an *ear* if the triangle formed by them is inside the polygon. By the *Two Ears Theorem* [Mei75], a simple polygon with more than 3 vertices has at least two non-overlapping ears. Since $link(v)$ is a simple polygon, it has two non-overlapping ears, at most one of which can contain v. Let u and its two neighbors u_- and u_+ be three vertices that form an ear not containing v. Since $link(v)$ is a star-shaped polygon w.r.t. v and v is not in $\triangle u_- u u_+$, each vertex of $\{v, u, u_-, u_+\}$ lies outside the triangle formed by the other three, and thus $\overline{vu}$ is a 2-2 flippable edge. We repeatedly flip 2-2 flippable edges incident to v until its degree decreases to three. Then we can flip any of the remaining edges to remove v. □

Theorem 5.5. *The flip-flop algorithm transforms $\mathcal{T}(S \oplus W)$ to $\mathcal{RT}(S \oplus W)$.*

Proof. By definition, flipping according to the D-criterion decreases the degree of a vertex labeled as redundant without increasing the degree of any other such vertex with smaller index. This implies that the number of flips according to the D-criterion is finite. Between any two such flips, only a finite number of flips according to the V-criterion can be performed as proved in Theorem 5.3. Therefore, flip-flop always terminates.

At the final state, no locally non-regular edges exist in $\mathcal{T}(S \oplus W)$, since such an edge is either directly removed by a flip or used to identify a redundant vertex (Lemma 5.1), which further leads to more flips (Lemma 5.4). By Theorem 2.2, the result is $\mathcal{RT}(S \oplus W)$. □

Theorem 5.5 promises the correctness of flip-flop in sequential execution. When flip-flop is executed in parallel, we use atomic operations to avoid conflicting flips; see Section 4.1. Since the number of redundant vertices is bounded, it always terminates after removing all redundant vertices and flipping all locally non-regular edges. Thus, the proof of Theorem 5.5 applies to the parallel execution as well.

5.3 Compute Regular Triangulation of a Point Set

We develop an algorithm, called *ffRT*, to compute $\mathcal{RT}(S \oplus W)$ from a planar point set S and its weight set W. The algorithm consists of two stages: the constructing stage builds a triangulation that contains all the non-redundant points of S and possibly some redundant ones; and the flipping stage transforms the triangulation into $\mathcal{RT}(S \oplus W)$ using flip-flop. In fact, the constructing stage outputs an extended triangulation, which is further transformed by flip-flop to an extended triangulation containing $\mathcal{RT}(S \oplus W)$.

Figure 5.3(a) and 5.3(b) illustrate the constructing stage of ffRT. We first construct an initial triangle with three points of S and use the center of this triangle as the kernel point s. We maintain an extended triangulation w.r.t. s, and grow it by incrementally inserting the points of S. During this incremental construction, each point not yet inserted is associated with the triangle it lies in; this triangle may be a virtual triangle. A point is excluded from future consideration if it has been inserted into the triangulation or proved to be redundant by the three vertices of the triangle it is associated with. In each iteration, for each triangle, one of the points that are associated with it is selected and inserted into the triangulation by a 1-3 flip, and all the others are re-associated with one of the three new triangles created by the flip. This process results into an extended triangulation of S w.r.t. s that contains all the non-redundant points of S.

In the flipping stage, we apply flip-flop on the extended triangulation with special treatment for the edges of $star(v)$, where v is the virtual vertex. As defined in Sec-

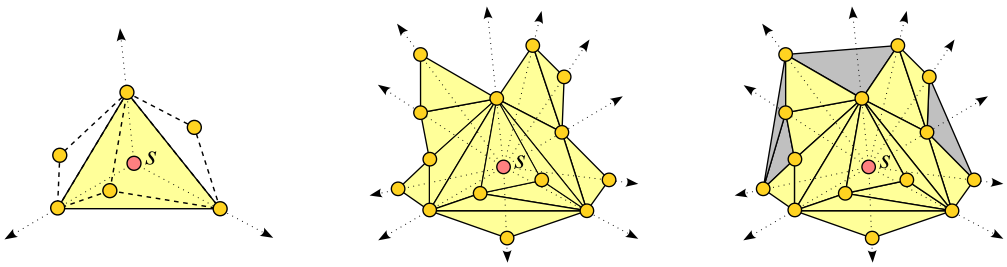
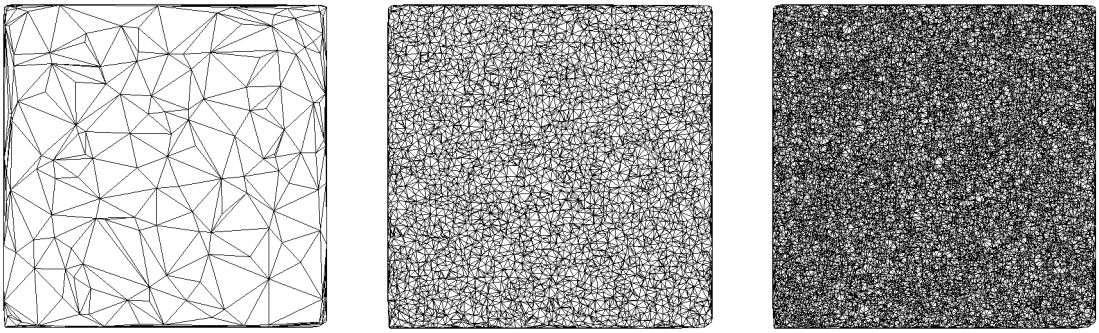


Figure 5.3: (a) The center of a triangle is selected as the kernel point s of the initial extended triangulation. (b) The extended triangulation grows by incrementally inserting points. (c) Flip-flop transforms the link of the virtual vertex to the convex hull of the input points by flipping the locally non-regular edges in the star of the virtual vertex.



(a) (b) (c)

Figure 5.4: Regular triangulations of 20,000 points where the percentage of non-redundant points are (a) %1, (b) %20, and (c) 99%.

tion 2.2, given an edge $\overline{bv}$ incident to $\triangle abv$ and $\triangle bcv$, $\overline{bv}$ is locally non-regular if s and a lie on different sides of $\overline{bc}$. Such an edge is always flippable, and thus will be flipped according to the V-criterion. After flipping $\overline{bv}$, a real triangle $\triangle abc$ is created and $link(v)$ is still star-shaped. When all the edges of $star(v)$ are locally regular, $link(v)$ becomes $\mathcal{CH}(S)$. For example, the gray triangles in Figure 5.3(c) are created by flip-flop.

Note that ffRT can be executed in parallel because the incremental construction of the extended triangulation can easily be adapted to parallel execution, and so is flip-flop since it has free execution order.

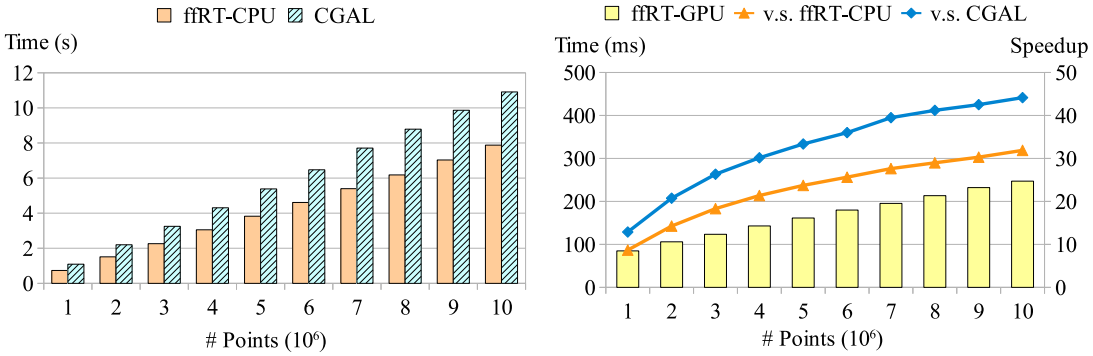
5.4 Experiment

We implement ffRT on both CPU and GPU, and name them *ffRT-CPU* and *ffRT-GPU* respectively. We compare our implementations with CGAL to compute 2D regular triangulation. We randomly generate points in uniform distribution with coordinates between [0.0, 1.0], and test three cases based on the range of weights assigned to the points; see Figure 5.4. In the first case, the weights of the points are randomly chosen from 0.0 to 0.01, so that the number of non-redundant points is approximately 1% of the number of the input points. In the second (resp., third) case, the weight range is $[0.0, 10^{-5}]$ (resp., $[0.0, 2 \times 10^{-7}]$), and approximately 20% (resp., 99%) of the input points are non-redundant.

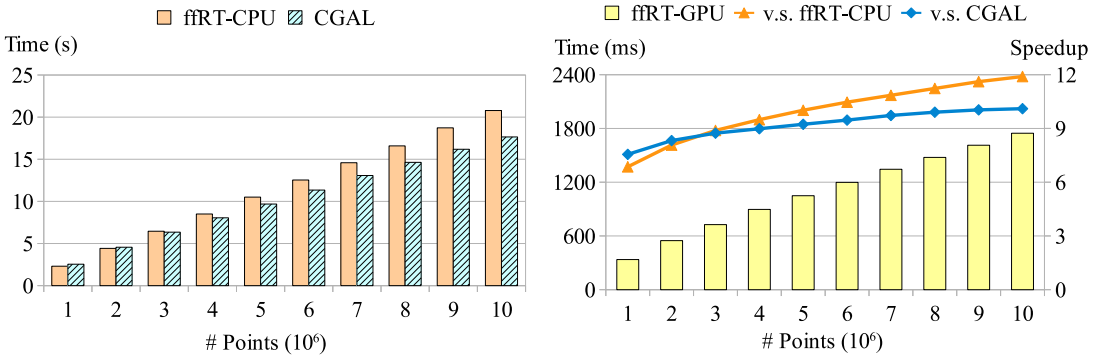
Running time

Figure 5.5 shows the experimental results in the three cases. Our CPU implementation achieves a performance close to CGAL. It runs 30% faster than CGAL in the first case (Figure 5.5(a)), 20% slower in the second case (Figure 5.5(b)), and almost as fast as CGAL in the third case (Figure 5.5(c)). Our GPU implementation achieves up to 45

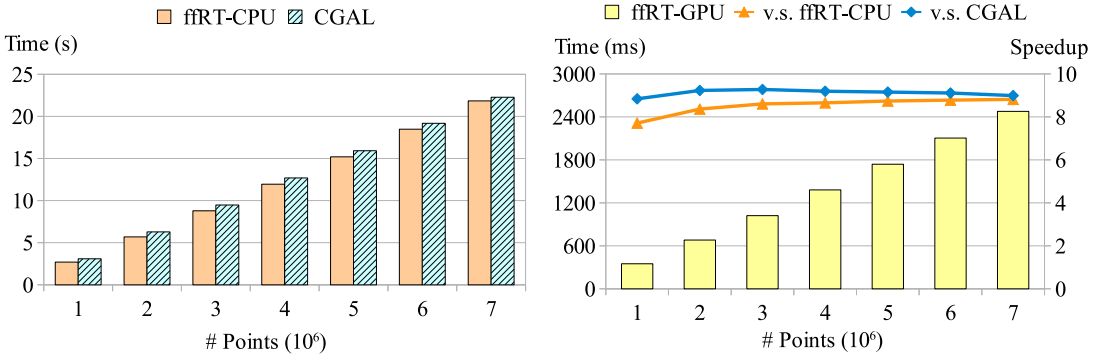
times speedup over CGAL in the first case, but only around 9 times in the other two cases. The main reason is that in the first case, most of redundant points are removed during the constructing stage of ffRT.



(a) Around 1% of the input points are non-redundant.



(b) Around 20% of the input points are non-redundant.



(c) Around 99% of the input points are non-redundant.

Figure 5.5: The running time of ffRT-CPU and CGAL (left), and the running time of ffRT-GPU compared with the two CPU implementations (right).

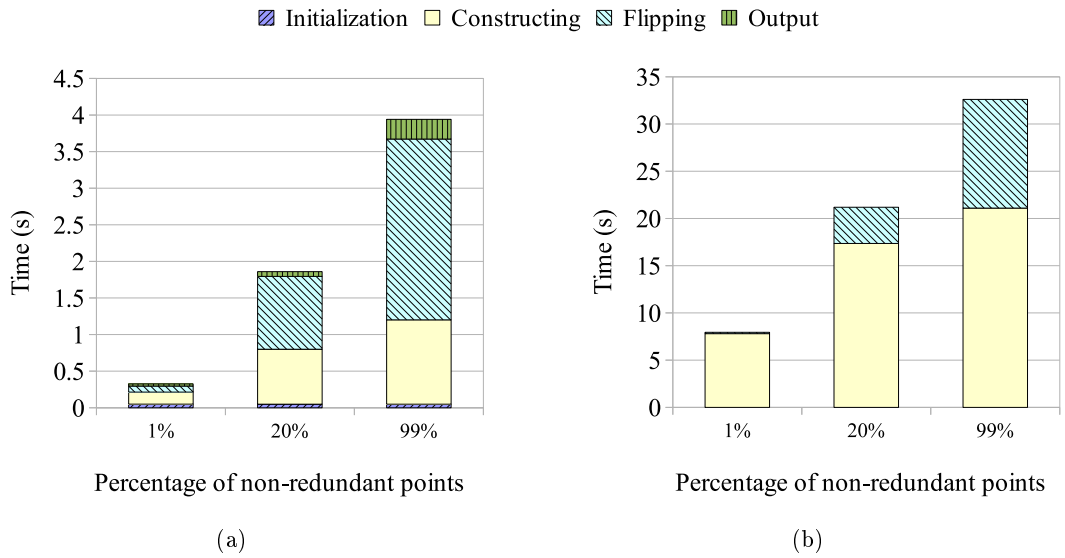


Figure 5.6: Time breakdown of ffRT (a) on the GPU and (b) on the CPU.

Time breakdown

We measure the time breakdown of ffRT for all the three cases with 10^7 points. The total running time of ffRT-GPU can be partitioned based on four stages: initialization (allocating memory and copying data from the CPU to the GPU), constructing, flipping and output (copying data back to the CPU). The running time of ffRT-CPU is partitioned based on only the constructing and the flipping stages.

Figure 5.6(a) shows the time breakdown of ffRT-GPU. As more input points are non-redundant, flipping takes a larger percentage of the total running time. In the first case (around 1% of the input points are non-redundant), flipping is 2 times faster than constructing, while in the third case (around 99% of the input points are non-redundant), flipping becomes 2 times slower than constructing. The main reason is that, when most of the input points are non-redundant, the initial triangulation of the flipping stage is very large and thus the flipping stage has a great amount of work. For all the three cases, the running time of the initialization and the output stages only takes a small proportion of the total time.

Figure 5.6(b) shows the time breakdown of ffRT-CPU. Similar to ffRT-GPU, the running time of the flipping stage takes a larger proportion when more input points are non-redundant. However, the constructing stage is always slower than the flipping stage for all the three cases.

CHAPTER 6

Flip Algorithm for 3D Convex Hull

This chapter advances the flip-flop algorithm in the previous chapter to construct 3D convex hull problem from a star-shaped polyhedron.

For a polyhedron $\mathcal{P}$ with its vertex set S, we want an algorithm to transform it to the convex hull of S by flips. Using the standard hill-climbing approach to monotonically increase the volume of $\mathcal{P}$ has two issues. First, a flip on an edge of $\mathcal{P}$ may result in self-intersection beyond the local region, after which the volume of $\mathcal{P}$ is no longer well defined. Second, if self-intersection is prohibited, the flipping can get stuck with no valid flips (see Figure 6.1). An approach such as the one proposed in [Alb03] only works when all the vertices of $\mathcal{P}$ are extreme.

To address the first issue, we study flips in the context of a star-shaped polyhedron where any self-intersection due to flip can be prevented by a simple local check. Section 6.1 reviews star-shaped polyhedron and its properties. In a star-shaped polyhedron, an unflippable and reflex edge as we define must be incident to a non-extreme vertex. Section 6.2 exploits this to adapt the flip-flop algorithm for 3D convex hull, and Section 6.3 proves its correctness.

With flip-flop, given a point set S, we can compute $\mathcal{CH}(S)$ by first constructing a star-shaped polyhedron of S using, for example, incremental insertion and then applying flip-flop. Section 6.4 details this approach termed ffHull, and Section 6.5 analyzes ffHull experimentally on both CPU and GPU. As shown in the experiment, ffHull on the GPU is up to 50 times faster than Qhull and 170 times faster than CGAL. Also, ffHull on the CPU runs up to 4 times faster than CGAL and as fast as QHull.

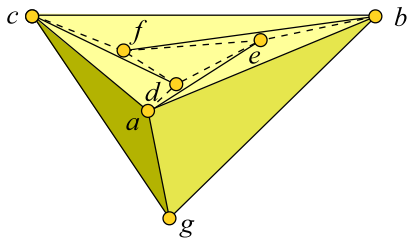


Figure 6.1: This star-shaped polyhedron is constructed as follows. A tetrahedron $gabc$ is initialized where $\triangle abc$ is equilateral and horizontal. Another equilateral and horizontal triangle $\triangle def$ is embedded into $\triangle abc$ and lower than $\triangle abc$. After twisting $\triangle def$ a bit, all dashed edges are reflex but flipping them creates self-intersection.

6.1 Star-Shaped Polyhedron

Let $\mathcal{P}$ be a polyhedron with its vertex set S and s be a point in $\mathbb{R}^3$, and assume that the points of $S \cup \{s\}$ is in general position. Given 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 $\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}\}$, i.e., the collection of points where each is a convex combination of some points on the three rays; see Figure 6.2. The cones of a set T of triangles is the union of the cones of all these triangles: $\mathcal{C}_s(T) = \bigcup_{\triangle \in T} \mathcal{C}_s(\triangle)$. $\mathcal{C}_s(\triangle abc)$ extends to infinity. The convex hull of any two rays of $\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}\}$ is a *facet* of $\mathcal{C}_s(\triangle abc)$, which also extends to infinity. These three facets together with the three rays, s, and the empty set $\emptyset$ form the faces of $\mathcal{C}_s(\triangle abc)$. Two cones *overlap* if their intersection is not a common face of them. The following lemma provides a new way to define star-shaped polyhedron using cones, which is preferable in this thesis.

Lemma 6.1. *A polyhedron $\mathcal{P}$ is star-shaped w.r.t. a point s if and only if $\forall \triangle_1, \triangle_2 \in \mathcal{P}$, $\triangle_1 \neq \triangle_2$, $\mathcal{C}_s(\triangle_1)$ and $\mathcal{C}_s(\triangle_2)$ do not overlap.*

Proof. (only if) Pick a ray that starts at s and is contained in the overlapping region of $\mathcal{C}_s(\triangle_1)$ and $\mathcal{C}_s(\triangle_2)$. The ray intersects with $\mathcal{C}_s(\triangle_1)$ and $\mathcal{C}_s(\triangle_2)$ respectively, and one of the intersected point is in the interior of the line segment whose endpoints are s and the other intersected point. By definition, $\mathcal{P}$ is not star-shaped.

(if) Proved by definition. □

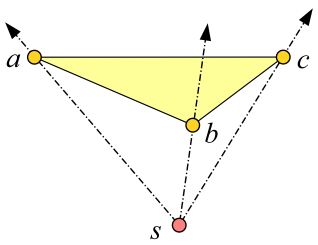


Figure 6.2: The cone of $\triangle abc$ w.r.t. s, $\mathcal{C}_s(\triangle abc)$.

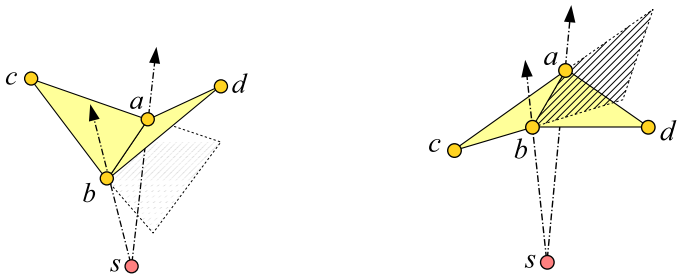


Figure 6.3: (a) $\overline{ab}$ is reflex. (b) $\overline{ab}$ is convex.

From here onward, let $\mathcal{P}$ be a star-shaped polyhedron w.r.t. a kernel point s. $\mathcal{P}$ has the following properties:

- For any triangle t of $\mathcal{P}$, none of the points of S lies inside $\mathcal{C}_s(t)$.
- For any vertex v of $\mathcal{P}$, none of the points of S except v lies inside $\mathcal{C}_s(star(v))$.
- $\mathcal{C}_s(\mathcal{P})$ covers $\mathbb{R}^3$.

Let $e = \overline{ab}$ be an edge of $\mathcal{P}$ with its two incident triangles $\triangle abc$ and $\triangle abd$. The two sides of a triangle are the two half-spaces defined by the plane containing the triangle. The edge e is a *reflex* edge (w.r.t. s) if c and s lie on different sides of $\triangle abd$; otherwise it is a *convex* edge. Besides, since $\mathcal{P}$ is star-shaped, c and s lie on two different sides of $\triangle abd$ if and only if d and s lie on two different sides of $\triangle abc$. Figure 6.3 illustrates these two concepts. Although the convexity and reflexivity of an edge is only a local property of $\mathcal{P}$, we can build the relation between this local property and the global convexity of $\mathcal{P}$ in the following theorem.

Theorem 6.2. *A star-shaped polyhedron $\mathcal{P}$ w.r.t. a point s is convex if and only if all its edges are convex w.r.t. s.*

Proof. (only if) When $\mathcal{P}$ is convex, any triangle defines a plane such that all the vertices of $\mathcal{P}$ and s lie on the same side, indicating that all the three edges of this triangle are convex w.r.t. s.

(if) By contradiction, suppose all the edges of $\mathcal{P}$ are convex while $\mathcal{P}$ is not. There exists a triangle t_0 and a vertex v of $\mathcal{P}$ such that v lies on the different side of t_0 from s. Pick a point p on t_0. The convex hull of $\overrightarrow{sp}$ and $\overrightarrow{sv}$ intersects with a series of edge-adjacent triangles of $\mathcal{P}$ starting at t_0 and ending at a triangle incident to v, as illustrated in Figure 6.4(a). By choosing a proper p, no vertices of $\mathcal{P}$ except v lie on the intersection. Let the sequence of triangles be $\{t_0, t_1, ..., t_k\}$.

Now consider $t_0 = \triangle abc$ and $t_1 = \triangle bcd$ that share a common edge $\overline{bc}$; see Figure 6.4(b). Let H be the plane defined by s, b and c. Since $\mathcal{P}$ is star-shaped w.r.t. s, d and a lie on different sides of H. By assumption $\overline{bc}$ is a convex edge and thus d and s lie on the same side of t_0. On the other hand, v lies on the different side of H from a

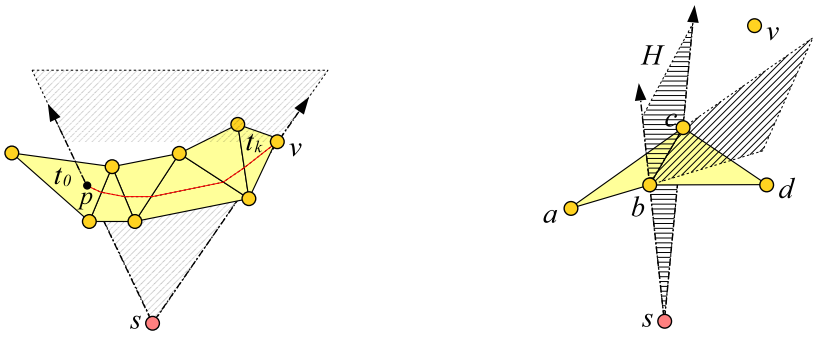


Figure 6.4: (a) The convex hull of $\overrightarrow{sp}$ and $\overrightarrow{sv}$ (shaded region) intersects with a sequence of triangles from t_0 to t_k. The intersection is shown in red segments. (b) H is the plane defined by s, b and c. Both v and d lie on the different side of H with a; d lies on the same side of $\triangle abc$ with s while v lies on the different side of $\triangle abc$ with s. Therefore v and s lie on different sides of $\triangle bcd$.

and on the different side of t_0 from s. We conclude that v lies on the different side of t_1 from s. This argument is repeated with t_1 replacing t_0 and so on. Finally, we conclude that v lies on the different side of t_{k-1} from s, implying that the edge shared by t_{k-1} and t_k is reflex, a contradiction. □

Because of Theorem 6.2, checking whether $\mathcal{P}$ is convex can be done by applying local checks on all the edges of $\mathcal{P}$, similar to Theorem 2.1 and Theorem 2.2. Note that this method only works for star-shaped polyhedron. There exists some polyhedron with self-intersection where all the edges are convex w.r.t. the same point. Figure 6.5 shows a simple 2D illustration.

Let $e = \overline{ab}$ be an edge of $\mathcal{P}$ with two link points c and d, and $\mathcal{T}_e$ be the induced-subcomplex of e. The flippability of e is subject to whether $\mathcal{P}$ is still star-shaped w.r.t. s after flipping e. There are two cases that make $\mathcal{P}$ no longer star-shaped w.r.t. s. The first case is when s lies outside $\mathcal{P}$ after flipping e; this cannot happen if s is outside the tetrahedron $abcd$. The second case is when flipping e creates self-intersection. Similar to the case when flipping on a 2D triangulation, no self-intersection is created by the flip on e if $\mathcal{C}_s(\mathcal{T}_e)$ is convex, i.e., if $\mathcal{C}_s(\mathcal{T}_e) = \mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$. Thus e is *flippable*

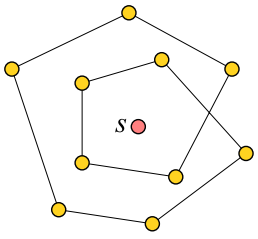


Figure 6.5: All the vertices are "convex", but the polygon is not.

(w.r.t. s) if (1) s is outside the tetrahedron $abcd$, and (2) $\mathcal{C}_s(\mathcal{T}_e)$ is convex; otherwise e is *unflippable*.

Lemma 6.3. *$\mathcal{P}$ is still star-shaped w.r.t. s after flipping a flippable edge of $\mathcal{P}$.*

Proof. Let $e = \overline{ab}$ be a flippable edge of $\mathcal{P}$ with its two link points c and d. The flip on e either adds the tetrahedron $abcd$ into or removes from $\mathcal{P}$. Because s lies outside the tetrahedron $abcd$, s is still inside $\mathcal{P}$ after flipping e. The new triangles created by the flip on e must be contained in $\mathcal{C}_s(\mathcal{T}_e)$ because $\mathcal{C}_s(\mathcal{T}_e)$ is convex. Thus the cone of any newly created triangle does not overlap with those outside $\mathcal{T}_e$.

We now prove that the cones of the newly created triangles do not overlap. If e is a 3-1 edge, only one triangle is created. If e is a 2-2 edge, any point of $\{a, b, c, d\}$ lies outside the cone of the triangle formed by the other three points; thus the cones of the two new triangles do not overlap. In addition, any triangle not in $\mathcal{T}_e$ is not changed by the flip. Therefore after flipping e, the cones of any two triangles of $\mathcal{P}$ do not overlap. By Lemma 6.1, $\mathcal{P}$ is still star-shaped w.r.t. s. □

We finish this section with some notes. First, checking the convexity and flippability of an edge are local operations by definition. Second, flipping a flippable edge that is reflex replaces it with a convex one, and vice versa. Third, by Lemma 6.3, we can always maintain a star-shaped polyhedron w.r.t. a fixed kernel point during flipping.

6.2 Flip-flop for 3D Convex Hull

Flip-flop for 3D convex hull is a variant of the flip-flop for 2D regular triangulation described in Chapter 5 to work on polyhedron. In the remaining of this chapter we use flip-flop to refer to the one for 3D convex hull unless otherwise stated. In this algorithm, the **geometric structure** is a star-shaped polyhedron w.r.t. a fixed kernel point s. The **local operations** consist of 2-2 flip and 3-1 flip, and the **candidate** is an edge associated with its induced-complex. A vertex is labeled as *non-extreme* once it is proved to be a non-extreme point.

The **criterion** of flip-flop is also a combination of the V- and the D-criterion with a slight difference from that for 2D regular triangulation. The V-criterion is to flip reflex edges to increase the volume of the polyhedron; and the D-criterion is to flip the edges incident to a non-extreme vertex in order to remove it from the polyhedron, using the indices of the vertices to avoid flipping back and forth. The D-criterion is prioritized over the V-criterion. The **local check** on an edge e includes one or more tasks as follows: checking the flippability of e, checking the convexity of e, and checking whether the vertices of e and $link(e)$ are labeled as non-extreme.

The pseudocode of flip-flop is shown in Algorithm 6, which is very similar to Algorithm 5. Instead of checking the local regularity of an edge, we here check the convexity

Algorithm 6: Flip-flop for 3D Convex Hull

input : a star-shaped polyhedron $\mathcal{P}$ w.r.t. s with its vertex set S
output: $\mathcal{CH}(S)$

```
1  label all vertices of P as unknown
2  Q ← { e | e is an edge of P }
3  while Q ≠ ∅ do
4      e ← Q.pop()
5      let e = ab, {c, d} be its link and x ∈ {a, b, c, d} be the non-extreme vertex
       with smallest index
6      if e is a 3-1 edge then
7          assume a is inside C_s(△bcd) w.l.o.g.
8          if a is labeled as non-extreme or e is reflex then
9              flip e; Q ← {bc, cd, db}
10     else
11         if a ∉ C_s(△bcd) and b ∉ C_s(△acd) then
12             if x does not exist and e is reflex then
13                 flip e; Q ← {ac, bc, ad, bd}
14             else if x ∈ {a, b} and s ∉ CH({a, b, c, d}) then
15                 flip e; Q ← {ac, bc, ad, bd}
16         else if e is reflex then
17             assume a is inside C_s(△bcd) w.l.o.g.
18             label a as non-extreme
19             Q ← { e' | e' is an edge of P with a as an endpoint }
```

of an edge w.r.t. the kernel point s. The flippability of an edge is checked based on s and the vertices included in the flip. When a 2-2 edge is reflex and unflippable, one of its endpoints can be shown to be non-extreme, and we label it. As an algorithm of local transformation with free execution order, flip-flop for 3D convex hull can be implemented under the sequential and the parallel workflows shown in Section 3.2.

6.3 Proof of Correctness

The main challenge of adapting flip-flop for 3D convex hull is to prove its correctness. This proof is significantly different from the one in Section 5.2.

Recall that $\mathcal{T}_e$ is the induced subcomplex of an edge e. In Line 8–9 of Algorithm 6, when a 3-1 edge is reflex we flip it without checking its flippability, and in Line 16–18, when a 2-2 edge is reflex and unflippable, one of its endpoints is labeled as non-extreme. These are based on the following 2 lemmas.

Lemma 6.4. *Any 3-1 edge of $\mathcal{P}$ that is reflex is flippable.*

Proof. Let $e = \overline{ab}$ be a 3-1 edge of $\mathcal{P}$ that is reflex and $\{c, d\}$ be its link. Without loss of generality, assume $\mathcal{T}_e = \{\triangle abc, \triangle bad, \triangle acd\}$; see Figure 6.6(a). Since e is reflex,

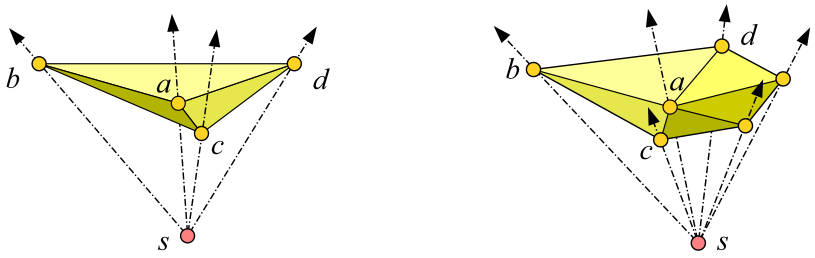


Figure 6.6: (a) $\overline{ab}$ is a 3-1 edge that is reflex. (b) $\overline{ab}$ is a 2-2 edge that is reflex and unflippable. In both cases, a is a non-extreme vertex since it lies inside the tetrahedron $sbcd$.

s and d lie on different sides of $\triangle abc$, so s is outside the tetrahedron $abcd$. As a lies inside $\mathcal{C}_s(\mathcal{T}_e)$, $\mathcal{C}_s(\mathcal{T}_e)$ equals $\mathcal{C}_s(\triangle bcd)$ and thus is convex. Therefore, e is flippable. □

Lemma 6.5. *Any 2-2 unflippable edge of $\mathcal{P}$ that is reflex is incident to a non-extreme vertex.*

Proof. Let $e = \overline{ab}$ be a 2-2 unflippable edge of $\mathcal{P}$ that is reflex and $\{c, d\}$ be the link of e. Since e is reflex, s and d lie on different sides of $\triangle abc$, therefore s is outside the tetrahedron $abcd$. As such, by the definition of the unflippable edge, the union of $\mathcal{C}_s(\triangle abc)$ and $\mathcal{C}_s(\triangle bad)$ must not be equal to $\mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$; see Figure 6.6(b). This implies that either a is inside $\mathcal{C}_s(\triangle bcd)$ or b is inside $\mathcal{C}_s(\triangle acd)$. Assuming the former one without loss of generality, and because e is reflex, a must lie inside the tetrahedron $sbcd$ and thus is non-extreme. □

Similar to Lemma 5.2 and Theorem 5.3, we can prove that any star-shaped polyhedron with all vertices being extreme can be transformed into its convex hull by flip-flop under only the V-criterion. We show that in this case any reflex edge is flippable, which implies that we can flip all the reflex edges of $\mathcal{P}$ to get $\mathcal{CH}(S)$.

Lemma 6.6. *If all the vertices of $\mathcal{P}$ are extreme, any reflex edge of $\mathcal{P}$ is 2-2 flippable.*

Proof. Let e be a reflex edge of $\mathcal{P}$. It cannot be a 3-1 edge; otherwise flipping it removes a non-extreme vertex (Lemma 6.4), a contradiction. Also e cannot be a 2-2 unflippable edge; otherwise one of its endpoints is non-extreme (Lemma 6.5), a contradiction again. Therefore, e is 2-2 flippable. □

Theorem 6.7. *Flipping according to the V-criterion can transform $\mathcal{P}$ to $\mathcal{CH}(S)$ if all the vertices of $\mathcal{P}$ are extreme.*

Proof. Since the volume of $\mathcal{P}$ monotonically increases, an edge removed by a flip cannot reappear, and thus the flipping terminates. By Lemma 6.6, all the reflex edges are

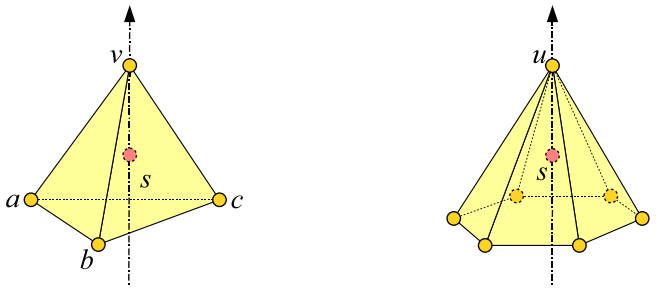


Figure 6.7: Two cases that a vertex is not removable by flipping its incident edges. (a) The kernel point s is inside the tetrahedron $vabc$. Flipping any edge incident to v makes s be outside the polyhedron. (b) The six neighbors of u are coplanar. The kernel point s lies inside the closed region formed by the star of u and the hexagon of the six neighbors. Then flipping any edge incident to u makes the polyhedron be not star-shaped w.r.t. s.

flippable. After the algorithm terminates, $\mathcal{P}$ is a star-shaped polyhedron containing only convex edges; by Theorem 6.2, $\mathcal{P}$ is $\mathcal{CH}(S)$. □

When a vertex of $\mathcal{P}$ is labeled as non-extreme the algorithm flips the edges incident to it according to the D-criterion in order to remove it from $\mathcal{P}$. While any vertex of a 2D triangulation not on the boundary is always removable by flipping its incident edges as shown in Lemma 5.4, a vertex of $\mathcal{P}$ may not be removable if all its incident edges are unflippable. For example, in Figure 6.7(a) all the edges incident to the vertex v are unflippable because flipping any of them makes the kernel point s be outside the polyhedron; in Figure 6.7(b) all the edges incident to the vertex u are unflippable because flipping any of them makes the polyhedron be not star-shaped w.r.t. the kernel point s. We next prove that a non-extreme vertex of $\mathcal{P}$ is always removable by flipping edges incident to it.

We introduce several new concepts for the proofs. Let v, a, b and c be different

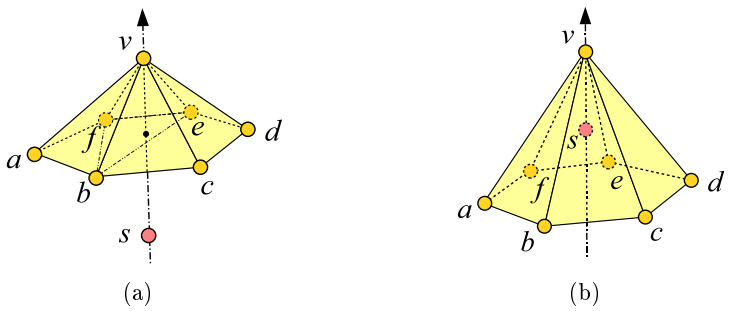


Figure 6.8: (a) v is locally covered by b, e and f on v's link. (b) v is not locally covered because no cone defined by three vertices on v's link contains it.

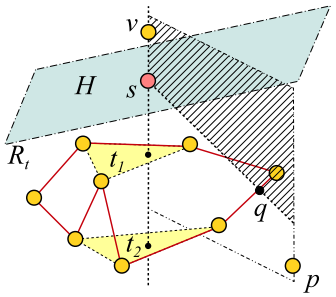


Figure 6.9: If v is not locally covered, the line passing through v and s intersects with $\mathcal{CH}(link(v))$, and v is in fact an extreme vertex. The solid edges form the link of v.

vertices of $\mathcal{P}$. The cone $\mathcal{C}_s(\triangle abc)$ is called a *cover* of v if v lies inside it. It is then called a *minimal cover* of v if no other vertices of $\mathcal{P}$ lie inside it. Note that v might have more than one minimal cover. We say v is *locally covered* if it has a cover formed by three vertices on its link; see Figure 6.8. We show that any non-extreme vertex is locally covered and a locally covered vertex is always incident to a flippable edge, and thus such a vertex can be removed by flipping.

Lemma 6.8. *If v is a non-extreme vertex, then v is locally covered.*

Proof. By contradiction, assume that v is not locally covered, i.e., no cone of three vertices of $link(v)$ is a cover of v. We construct $\mathcal{CH}(link(v))$ and the line vs, as shown in Figure 6.9. Because v is inside $\mathcal{C}_s(star(v))$, the line vs must intersect with $\mathcal{CH}(link(v))$ at two triangles t_1 and t_2. Note that t_1 and t_2 are possibly the same triangle when $link(v)$ has only 3 vertices. Since $\mathcal{C}_s(t_1)$ and $\mathcal{C}_s(t_2)$ are not covers of v, v and s must lie on the same side of both triangles with v being further. Without loss of generality, let the intersection of t_1 and the line vs be nearer to s than that of t_2, and let H be the plane through s and parallel to t_1.

We prove that all vertices of $\mathcal{P}$ lie on the half-space of H not containing v. Let R_t be this half-space; clearly $\mathcal{CH}(link(v))$ lies inside R_t. Let p be an arbitrary vertex in $\mathcal{P}$ other than v and those in $link(v)$. The half-plane defined by vs and containing p must intersect with $link(v)$ at a point q. The point p must lie outside the angle $\widehat{vsq}$; otherwise it falls into $\mathcal{C}_s(star(v))$, contradicting the star-shaped polyhedron. Since q lies in R_t, p also lies in R_t. Thus all vertices of $\mathcal{P}$ other than v lie inside R_t and therefore v is an extreme point, a contradiction. □

Lemma 6.9. *If v is locally covered, it has a minimal cover with vertices on its link.*

Proof. Let $\mathcal{C}_s(\triangle abc)$ be a cover of v such that $\{a, b, c\} \subseteq link(v)$. If there is another vertex $p \in link(v)$ that lies inside $\mathcal{C}_s(\triangle abc)$, without loss of generality we assume that v lies inside $\mathcal{C}_s(\triangle pab)$; see Figure 6.10. In this case, we replace $\triangle abc$ with $\triangle pab$ and repeat the argument. Since the focused cover is shrunk by each replacement, this process can terminate.

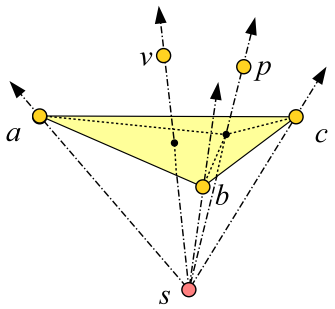


Figure 6.10: The cone of $\triangle abc$ where $\{a, b, c\} \in link(v)$ is a cover of v, and if another $p \in link(v)$ is inside $\mathcal{C}_s(\triangle abc)$, p subdivides the cone into three and v lies inside one of them.

Now let us assume that no other vertices of $link(v)$ lie inside $\mathcal{C}_s(\triangle abc)$. We argue that $\mathcal{C}_s(\triangle abc)$ is completely inside $\mathcal{C}_s(star(v))$. Otherwise, an edge $\overline{pq}$ of $link(v)$ must cut through $\mathcal{C}_s(\triangle abc)$ and thus $\mathcal{C}_s(\triangle vpq)$ overlaps one of the cones of the triangles incident to the edge $\overline{va}$, $\overline{vb}$ and $\overline{vc}$. This violates the fact that $\mathcal{P}$ is star-shaped. Vertices in $\mathcal{P} \setminus link(v)$ cannot lie inside $\mathcal{C}_s(\triangle abc)$. Therefore, $\mathcal{C}_s(\triangle abc)$ is a minimal cover of v. □

Lemma 6.10. *If the degree of a non-extreme vertex v is 3, any edge incident to it is 3-1 flippable.*

Proof. By Lemma 6.8, v is locally covered. Let $\{a, b, c\}$ be the vertices of the link of v. First, s lies outside the tetrahedron $vabc$; otherwise v is not locally covered. Second, similar to the proof of Lemma 6.4, we have $\mathcal{C}_s(\mathcal{T}_e) = \mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sv}\})$ for any edge e incident to v. Therefore any edge incident to v is flippable. □

Lemma 6.11. *If the degree of a non-extreme vertex v is more than 3, there exists a 2-2 flippable edge incident to v.*

Proof. From Lemma 6.8 and Lemma 6.9, let $\mathcal{C}_s(\triangle abc)$ be the minimal cover of v where $\{a, b, c\} \in link(v)$. The three vertices partition the link of v into three chains of vertices: L_{ab}, L_{bc} and L_{ca}, each of which goes between two of these vertices and does not include the third one; see Figure 6.11. Since the degree of v is more than 3, there is at least one chain with more than 2 vertices. Without loss of generality, let L_{ab} be $\langle a, p_1, p_2, \ldots, p_n, b\rangle$ such that $n \geq 1$. We prove that there exists a vertex p_m $(1 \leq m \leq n)$ such that the edge $\overline{vp}_m$ is 2-2 flippable.

Let D_{sva} be the half plane through v, s, a that is defined by the line vs and contains a. Similarly we define D_{svb} and D_{svc}. Let R_{ab} be the region bounded by D_{sva} and D_{svb} and containing the edge $\overline{ab}$. $\mathcal{C}_s(\triangle abc)$ is a cover of v, thus a and b must lie on two different sides of $\triangle svc$. To go from a to b without going into R_{ab}, the chain L_{ab} must intersect with D_{svc} at an edge $e = \overline{pq} \in L_{ab}$, and as a result $\mathcal{C}_s(\triangle vpq)$ overlaps

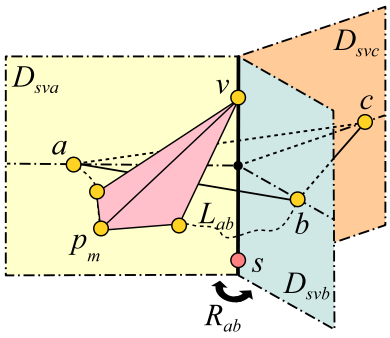


Figure 6.11: If the chain of vertices L_{ab} has more than 2 vertices, there exists a vertex p_m in the chain such that the edge $\overline{vp_m}$ is 2-2 flippable.

one of the cones of the triangles incident to $\overline{vc}$, violating the fact that $\mathcal{P}$ is star-shaped. Therefore, L_{ab} goes through R_{ab}. By the same argument, L_{ab} cannot intersect with D_{sva} or D_{svb} at an edge, therefore L_{ab} lies completely inside R_{ab}.

Now we show how to find the vertex p_m. Since $\mathcal{C}_s(\triangle abc)$ is a minimal cover of v, L_{ab} cannot go through $\mathcal{C}_s(\triangle abc)$; otherwise some cones of the triangles of $\mathcal{P}$ would overlap. Consider the convex hull of $\{\overrightarrow{sv}, \overrightarrow{sa}, \overrightarrow{sp}_1, \ldots, \overrightarrow{sp}_n, \overrightarrow{sb}\}$, there must be a vertex p_m $(1 \leq m \leq n)$ on its boundary. Let p_{m-1} and p_{m+1} be its two neighbors in L_{ab} ($p_{m-1} = a$ if $m = 1$ and $p_{m+1} = b$ if $m = n$). Clearly, p_m lies outside $\mathcal{C}_s(\triangle vp_{m-1}p_{m+1})$. Since p_{m-1}, p_m and p_{m+1} lie inside R_{ab}, v is outside $\mathcal{C}_s(\triangle p_{m-1}p_mp_{m+1})$. Therefore, $\overline{vp}_m$ is a 2-2 edge and the union of $\mathcal{C}_s(\triangle vp_{m-1}p_m)$ and $\mathcal{C}_s(\triangle vp_mp_{m+1})$ is equal to $\mathcal{CH}(\{\overrightarrow{sv}, \overrightarrow{sp}_{m-1}, \overrightarrow{sp}_m, \overrightarrow{sp}_{m+1}\})$. On top of that, since v, p_{m-1}, p_m and p_{m+1} are inside R_{ab}, s is surely outside the tetrahedron $vp_{m-1}p_mp_{m+1}$. As a result, $\overline{vp}_m$ is a 2-2 flippable edge. □

Finally, we can prove the correctness of flip-flop based on the previous lemmas. The proof is very similar to that of Theorem 5.5 and thus omitted here.

Theorem 6.12. *The flip-flop algorithm transforms any star-shaped polyhedron to its convex hull.*

6.4 Compute Convex Hull of a Point Set

We develop *ffHull*, an algorithm utilizing flip-flop to compute the convex hull of a point set S in $\mathbb{R}^3$. Before applying flip-flop, ffHull constructs a star-shaped polyhedron $\mathcal{P}$ using S so that all the points of S are either inserted into $\mathcal{P}$ or proved to be inside $\mathcal{P}$.

Algorithm 7 shows the pseudocode of the sequential version of ffHull. There are two stages: constructing a star-shaped polyhedron $\mathcal{P}$ (Line 1–14), and transforming $\mathcal{P}$ to $\mathcal{CH}(S)$ using flip-flop (Line 15).

In the constructing stage, starting at an initial tetrahedron (Line 1–4), we grow the polyhedron $\mathcal{P}$ star-shaped w.r.t. the centroid s of the initial tetrahedron by incremen-

Algorithm 7: ffHull

input : a set S of points in $\mathbb{R}^3$
output: $\mathcal{CH}(S)$

```
1  let a, b, c, d be any 4 extreme vertices
2  S ← S \ {a, b, c, d}
3  P ← CH({a, b, c, d})
4  s ← the centroid of P
5  associate each p ∈ S to △abc ∈ P s.t. p ∈ C_s(△abc)
6  while S ≠ ∅ do
7      foreach △abc associated by some points do
8          let v be the furthest point associated to △abc
9          P ← P ∪ {△vab, △vbc, △vca} \ {△abc};
10         S ← S \ {v}
11     foreach p ∈ S do
12         let p be associated to △abc, into which v is just inserted
13         associate p to t ∈ {△vab, △vbc, △vca} s.t. p ∈ C_s(t)
14         if p and s lie on the same side of t then S ← S \ {p}
15 apply flip-flop on P
```

tally processing the input points: an input point is either inserted to become a vertex of $\mathcal{P}$ or removed if found to be inside $\mathcal{P}$ constructed so far. For each point p in S, we *associate* p with a unique $\triangle abc$ of $\mathcal{P}$ if p is inside the cone of $\triangle abc$ (Line 5). We remove p if it is inside the tetrahedron $sabc$, i.e., when p and s lie on the same side of $\triangle abc$, since it is a non-extreme point. In the main loop, $\mathcal{P}$ is grown by inserting the furthest point v associated to each triangle $t \in \mathcal{P}$ into $\mathcal{P}$ (Line 7–10); doing so helps to remove non-extreme points in S quickly. This is done by replacing t with three new triangles. Each point inserted into $\mathcal{P}$ is removed from S. Each insertion splits $\mathcal{C}_s(t)$ into three new non-overlapping cones. This guarantees that $\mathcal{P}$ is still star-shaped after the insertion. Line 11–14 update the triangle each point $p \in S$ is associated to, and remove p if it is found to be a non-extreme point. This "growing" process is repeated until S is empty.

In the flipping stage, we simply apply flip-flop to transform $\mathcal{P}$ to $\mathcal{CH}(S)$.

There are two points to note for the above algorithm. First, flip-flop actually works for any star-shaped polyhedron. It is thus not necessary to always find the furthest point to insert into $\mathcal{P}$. In fact, it is costly to find such furthest points because of numerical error. In practice, we often choose the almost furthest points to insert to construct the star-shaped polyhedron. Second, the algorithm presented above is just one approach to use flip-flop to construct the convex hull. Another possibility is to alternate between inserting points and flipping in multiple iterations.

The point insertion process in the constructing stage generates a star-shaped polyhedron from the input points, with no extreme vertices being excluded. By Theo-

rem 6.12, the subsequent flip-flop in the flipping stage correctly computes $\mathcal{CH}(S)$, thus the correctness of ffHull is guaranteed.

6.4.1 GPU Implementation

The polyhedron is represented as an array of triangles, each containing the indices of its three vertices and the indices of the triangles sharing its three edges. Furthermore, some auxiliary arrays are also used for intermediate computation. For example, we need an array to store the index of the furthest point for each triangle and an array to store for each point the index of the triangle it is associated to. The arrays are dynamically expanded rather than pre-allocated since usually only a small number of points appear in the polyhedron.

We use two techniques to simplify (and also optimize) the implementation of ffHull. Let $orient(p, t)$ be the determinant used to determine whether the point p lies beneath or beyond the triangle t. The first technique is to maintain the orientation of each triangle t in the polyhedron so that the kernel point s is beneath t. With this, to perform Line 14 of Algorithm 7, we only need to compute $orient(p, t)$. Similarly, we can check the reflexivity of an edge, without referring to s. The second technique is reusing $|orient(p, t)|$, which is the volume of the tetrahedron formed by p and t, instead of actually computing the distance when finding the furthest point to t.

The details of the implementation are as follows. In the constructing stage we have four major GPU kernels. The first kernel (Line 5 of Algorithm 7), with one thread processing one input point p, finds the triangle in the initial tetrahedron that p is associated to and at the same time participates in the search for the furthest point of each triangle. We use two arrays in the global memory to store for each triangle the furthest point and its distance, with the initial value being 0. The thread associating p to a triangle t uses $orient(p, t)$ to judge whether p is beneath t or not, and marks p as deleted if it is beneath. Otherwise, the distance from p to t (actually $orient(p, t)$ is used) is compared with the currently recorded value and if the new distance is larger, p is recorded as the furthest point of t, and the distance is updated.

The second kernel (Line 7–10), with one thread processing one triangle t, inserts one point associated to t into the polyhedron. A point being inserted into t replaces it with three new triangles. The first one is stored in the original slot, while the other two are appended into the end of the array of triangles. Then, the third kernel, with one thread processing one triangle, updates the full adjacency information of all new triangles and those adjacent to them. A separate kernel is necessary here since updating directly in the second kernel may create memory read and write conflict. The fourth kernel (Line 11–14), with one thread processing one point p, updates p's associated triangle if p is still outside $\mathcal{P}$. For p with t being the previous associated triangle, we first read the slot in the triangle array that previously stored t, which now stores

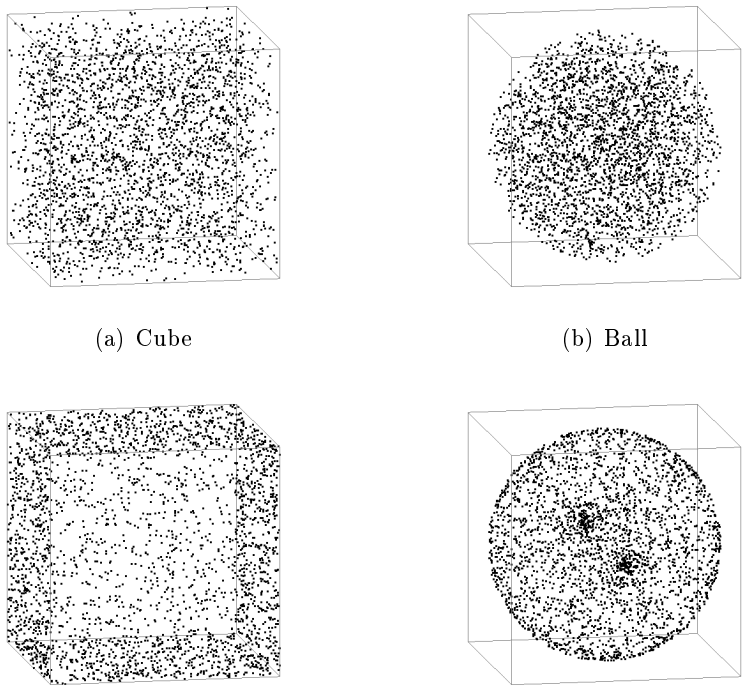
the first new triangle. From its adjacency information, we obtain the other two new triangles. Among these three, we identify the triangle t' that p is to be associated to, and mark p as deleted if necessary. Then the furthest point for t' is updated similar to the first kernel mentioned above. Note that in the first and fourth kernels, since the threads are executed in parallel, we do not always get the furthest point for each triangle. However, this approach is efficient without compromising on the correctness.

The flipping stage is done in multiple iterations. In each iteration, we use two kernels, a checking kernel and a flipping kernel, to perform flip in parallel, similar to the technique described in [QCT12] and [NHS11]. In the checking kernel, we assign one thread to one triangle to check if one of its edges should be flipped based on the criterion. The difficulty is that the induced sub-complexes of some edges share some triangles, thus the flips on these edges conflict and cannot be done in the same iteration. To avoid this, if a thread in charge of $\triangle abc$ wants to flip the edge $e = \overline{ab}$, it uses the atomic minimum operation to label the triangles of $\mathcal{T}_e$ with the index of $\triangle abc$. In the flipping kernel, we also assign one thread to one triangle. The thread in charge of $\triangle abc$ only flips e if the triangles of $\mathcal{T}_e$ are still labeled with the index of $\triangle abc$. This guarantees no conflicting flips are performed concurrently, and in each iteration at least one flip can be done. Since only up to three threads write to the same memory location during the labeling, the use of the atomic operation does not affect the efficiency much.

6.4.2 Exact Computation and Robustness

The only predicate we use, the 3D orientation predicate, is adapted from the exact predicate of Shewchuk [She97]. Together with the SoS technique [EM90], we can guarantee the exactness and robustness of our implementation. The only source of inexact computation is computing the kernel point s. If the initial tetrahedron is almost flat, its centroid computed inexactly can lie outside. In our implementation, by carefully choosing the first four extreme points, we make sure they are far away from each other to avoid the undesirable situation, unless all the input points are almost co-planar. To really fix this problem, we should use the input points as the kernel points. Specifically, we can select an extreme point v_1 as the kernel point, and another extreme point v_2 as the second kernel point. During the insertion, we need to guarantee that $star(v_1)$ is star-shaped w.r.t. v_2 and the polyhedron is star-shaped w.r.t. v_1. This procedure is much more complicated and less efficient than the one described in Algorithm 7.

There are two notes for the GPU implementation. First, instead of using a multiple-stage adaptive arithmetic on the GPU, we only use two stages: a fast computation with all arithmetic operations being done using native floating-point numbers, and an exact computation with all arithmetic operations being fully expanded into arrays of floating-point numbers. To verify whether the fast computation gives the correct sign, we use the forward error analysis approach described by Shewchuk. The error bounds are pre-



(a) Cube (b) Ball

(c) Box with thickness 0.01 (d) Sphere with thickness 0.01

Figure 6.12: Four distributions of tested data for 3D convex hull.

computed and stored for later usage. Second, the exact computation code requires a lot of registers and local memory, so each kernel that needs exact computation is split into two kernels. The first one performs only fast computations, and uses the error bounds to determine whether it requires exact computation or not. In second kernel, only the threads that need exact computation are active. By doing this, the first kernel requires less registers and local memory, and thus can run with higher parallelism. The second kernel, on the other hand, has very little work to do.

6.5 Experiment

We implement ffHull on both CPU and GPU, and denote these two implementations as *ffHull-CPU* and *ffHull-GPU* respectively. We compare their performance with the two fastest sequential implementations of the Quickhull algorithm: Qhull [Qhu12] and CGAL [CGA12]. Qhull handles roundoff errors 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 find that CGAL always runs slower than Qhull due to its use of exact arithmetic.

Points are generated randomly with coordinates between $[0.0, 1.0]$ and distributed

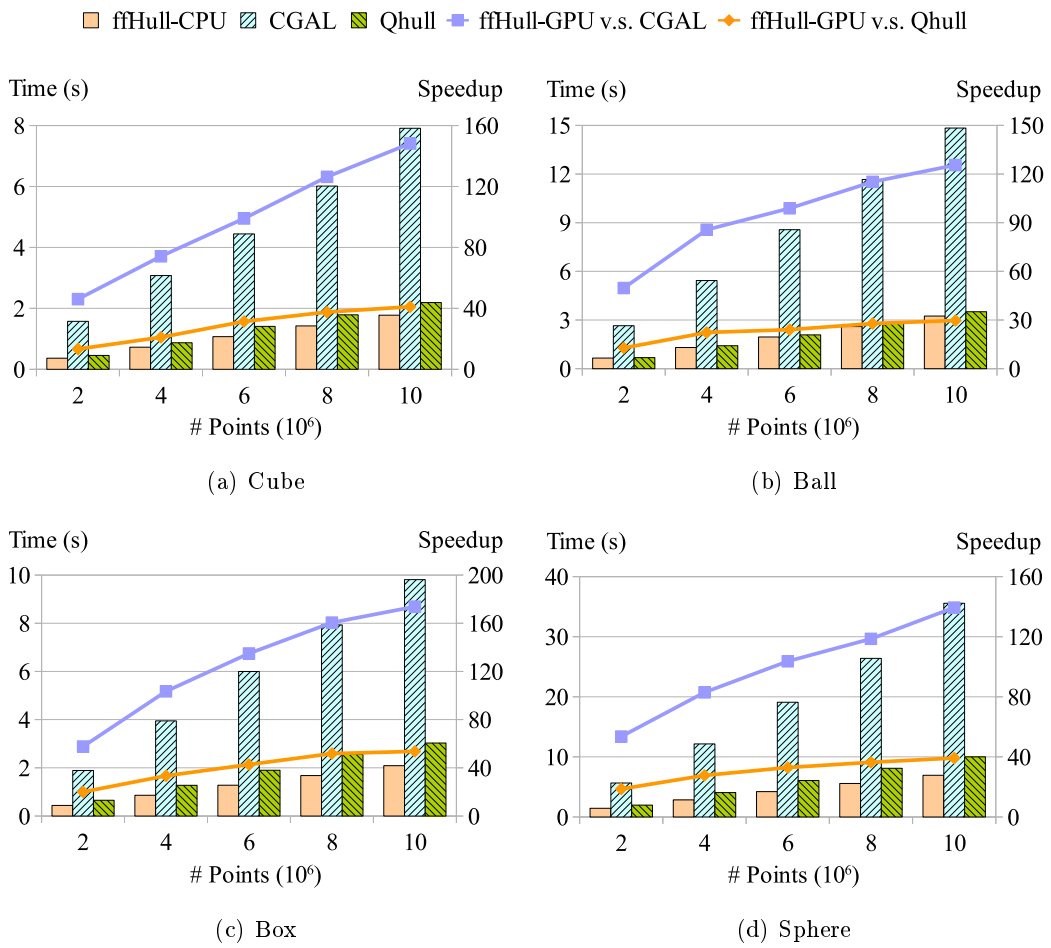


Figure 6.13: Running time of ffHull for the four distributions of points.

uniformly in four distributions: a cube, a ball of radius 0.5, a box with thickness of 0.01, and a sphere with thickness of 0.01; see Figure 6.12. 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 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 sphere is the extreme case where many points are on the convex hull, while the rest of them are also close to it. The size of tested point set is in the range $[10^6, 10^7]$.

Running time

We show the running time of the three CPU implementations in Figure 6.13: ffHull-CPU, CGAL, and Qhull. For all the point distributions, ffHull-CPU runs 3 to 4 times faster than CGAL and as fast as Qhull. Considering ffHull-CPU uses exact computation while Qhull does not, ffHull-CPU has a better performance than Qhull.

Figure 6.13 also shows the speedup of ffHull-GPU over CGAL and Qhull. For the

Table 6.1: Running time of ffHull, CGAL and Qhull on 3D models.

Model	# Points (millions)	Running time (ms)			
		CGAL	Qhull	ffHull-CPU	ffHull-GPU
Asian dragon	3.6	1181	540	997	117
Thai statue	5.0	1538	692	1240	91
Lucy	13.9	4488	1884	3664	192

ball distribution, ffHull-GPU is up to 30 times faster than Qhull and 120 times faster than CGAL. For the cube and the sphere distributions, ffHull-GPU is up to 40 times faster than QHull and 140 times faster than CGAL. Notably, ffHull-GPU performs very well for the box distribution: it runs 50 times faster than Qhull and 170 times faster than CGAL when the number of points are 10^7.

We also test ffHull with models of over a million points from the Stanford 3D scanning repository [Sta12]; see Table 6.1. These models have very few points on the convex hull, and most other points are distributed near the surface with many being co-planar. Since ffHull suffers from a large amount of coplanar points, ffHull-CPU runs approximately 2 times slower than Qhull and only a little faster than CGAL. The speedups of ffHull-GPU over Qhull are from 4.6 to 9.8 times, and those over CGAL are from 10.1 to 23.4 times.

Sensitivity

When the input has more points close to the boundary of the convex hull, ffHull excludes fewer non-extreme points during constructing star-shaped polyhedron and thus spends more time on its both stages. We investigate the sensitivity of ffHull to the thickness of the sphere distribution. Specifically, we test the running time of ffHull using 10^7 points on the sphere distribution with thickness varying from 0.5 to 0.0001.

As shown in Figure 6.14, ffHull-CPU slows down when the sphere becomes thinner, and similar performance can be observed for CGAL. The speedup of ffHull-CPU over CGAL is around 3 to 4 times for all the thicknesses. On the other hand, the rate of slowing down in Qhull is much larger that those in ffHull-CPU and CGAL. Qhull is as fast as ffHull-CPU when the thickness is in [0.5, 0.01], while it runs 3 times slower than ffHull-CPU at thickness 0.0001.

Figure 6.14 also shows the speedup of ffHull-GPU over CGAL and Qhull. As the speedup of ffHull-GPU over CGAL becomes smaller when the thickness varies from 0.01 to 0.0001, we conclude that ffHull-GPU is more sensitive to the thickness than CGAL and ffHull-CPU. This is mainly because ffHull-GPU takes much more time for exact computation when the sphere is thinner. However, the sensitivity of ffHull-GPU is still less than Qhull: the speedup of ffHull-GPU over Qhull increases sharply from 30 to 60 times when the sphere becomes thinner.

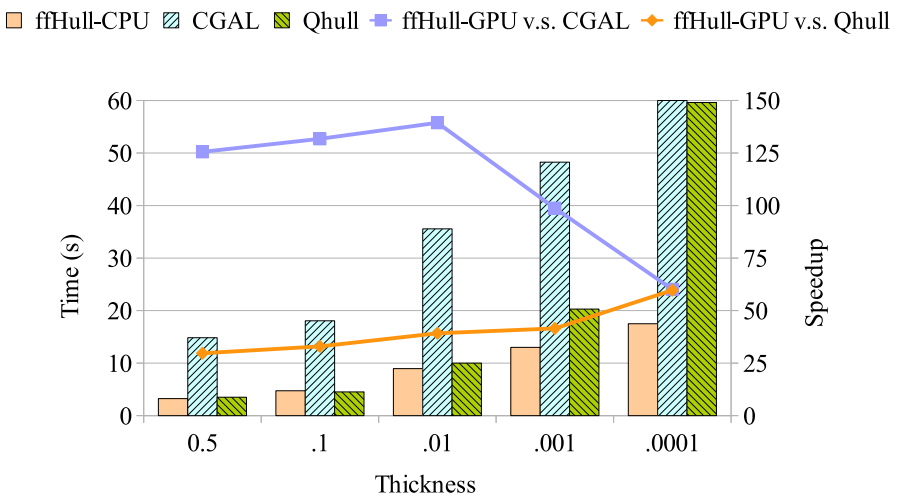


Figure 6.14: Running time of ffHull for a sphere distribution of different thicknesses.

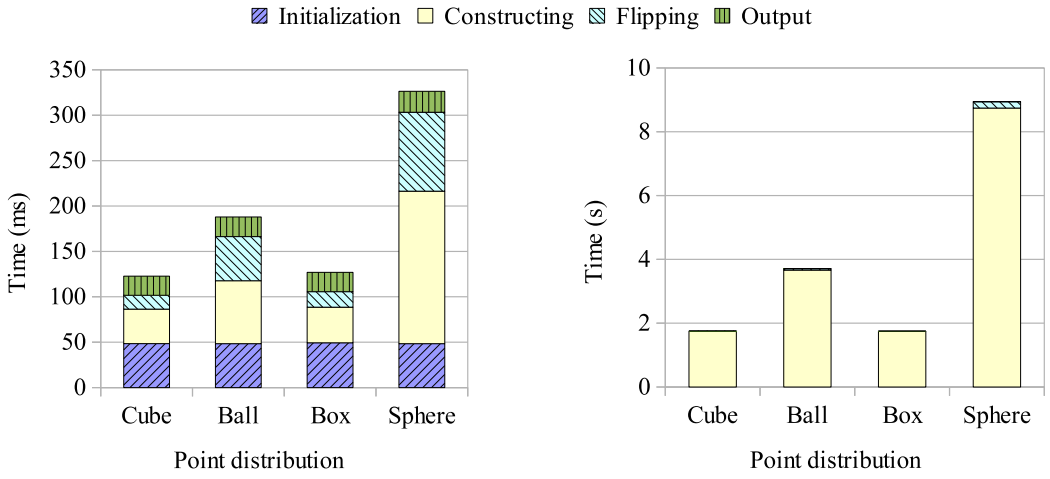
Time breakdown

We first measure the time breakdown of ffHull for the four distributions of 10^7 points. The total running time of ffHull-GPU can be partitioned based on four stages: initialization (allocating memory and copying data from the CPU), constructing, flipping, and output (copying data back to the CPU). The running time of ffHull-CPU can be partitioned in a similar way, yet without the initialization and the output stages.

Figure 6.15(a) shows the time breakdown of ffHull-GPU. The time of constructing takes a major proportion; especially in the sphere distribution, it takes more than 50% of the total running time, as most of the input points cannot be removed during this stage. In contrast, flipping uses much less time, implying that the constructed polyhedron is a proper starting point of flip-flop. The time of the initialization and the output stages is unchanged among different distributions. The time of initialization is quite large, because copying data between CPU and GPU is costly.

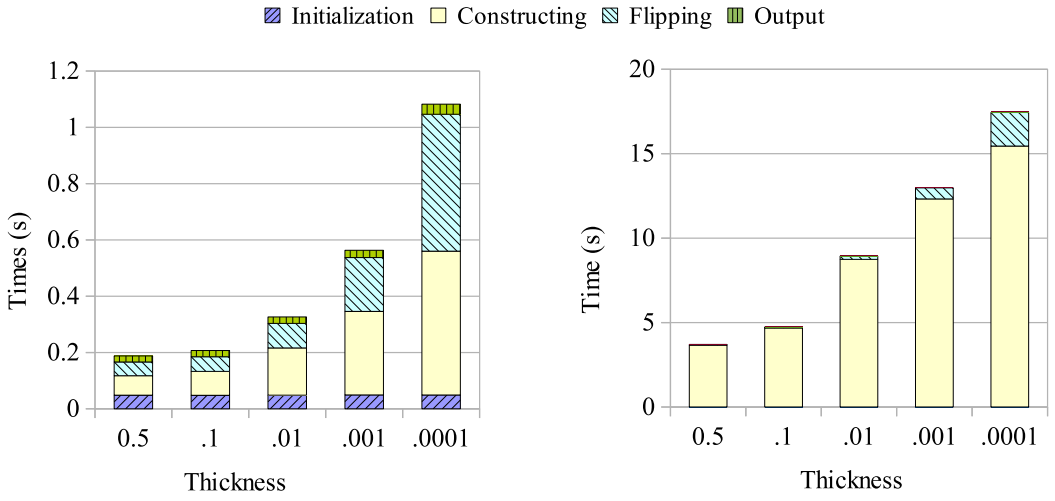
Figure 6.15(b) shows the time breakdown of ffHull-CPU. Unlike ffHull-GPU, flipping of ffHull-CPU takes very few time compared with constructing. The main reason is that ffHull-CPU can always find the furthest point to insert during constructing, and thus provides a much better input for flipping.

We also measure the time breakdown of ffHull with 10^7 points on the sphere distribution of various thicknesses, as shown in Figure 6.16. Point set distributed in the thinner sphere has more extreme points. As more points are on the convex hull, both constructing and flipping take more time, and thus the proportion of the initialization and the output stages in ffHull-GPU decreases. For both ffHull-GPU and ffHull-CPU, flipping takes a larger proportion of the total running time as the sphere becomes thinner. For example, in ffHull-GPU, flipping only takes about 50% as much time as constructing at thickness 0.01, while it takes almost the same time as constructing at



(a) (b)

Figure 6.15: Time breakdown of ffHull for the four distributions of points (a) on the GPU and (b) on the CPU.



(a) (b)

Figure 6.16: Time breakdown of ffHull for a sphere distribution of different thicknesses (a) on the GPU and (b) on the CPU.

thickness 0.0001. This is because the number of points on the convex hull affects the constructing stage at a logarithmic rate (i.e., only affects the number of loops) while it affects the flipping stage at a linear rate in our experiment.

Number of flips

In Figure 6.17, we present the total number of flips performed by ffHull-CPU when running on different distributions with varying number of points. For the cube and the

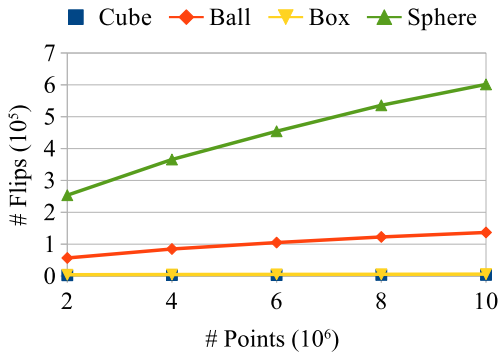


Figure 6.17: Number of flips performed by ffHull on the CPU.

box distributions, due to the small number of points on the convex hull, the numbers of flips needed are very small, as shown by the two overlapping curves near the horizontal axis. On the other hand, for the ball and the sphere distributions, the numbers of flips are nearly linear to the number of points. This result matches with the result of the traditional Lawson's flip algorithm when computing the 2D Delaunay triangulation in practice.

CHAPTER 7

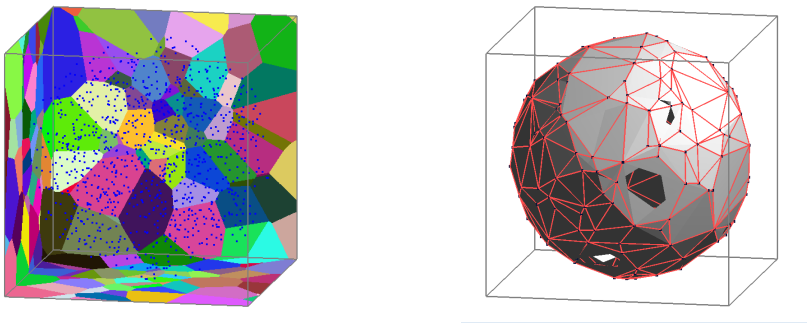
Algorithm using Splaying for 3D Convex Hull

This chapter discusses the use of star splaying to construct convex hull from a point set in $\mathbb{R}^3$, and compares it with the ffHull algorithm developed in the previous chapter.

Star splaying is a powerful algorithm of local transformation compared with flip algorithms because its input, a set of convex stars, is very general and it works in any dimension for computing convex hull. However, to be efficient, it must start from a good approximation of the output. To derive such a good approximation, we exploit the relation between 3D Voronoi diagram and 3D convex hull. In particular, we derive convex stars from the digital restricted Voronoi diagrams on the six sides of a box enclosing all the input points in the digital space. The whole process is termed gHull in Section 7.1.

Section 7.2 further elaborates the implementation details of gHull. Section 7.3 proves the correctness of gHull. Since the performance of gHull is determined by the approximation constructed in the digital space, it is sensitive to the distribution of input points. In particular, we identify three possible problems and propose strategies to remedy them.

Section 7.4 details experiment on gHull, and compares it with the GPU implementation of ffHull and existing CPU implementations, Qhull and CGAL. By design, gHull is meant for GPU implementation. It is up to 30 times faster than Qhull and 100 times faster than CGAL. On the other hand, ffHull on the GPU performs better than gHull for most of input distributions, especially when most of input points are close to the boundary of the convex hull.



(a) (b)

Figure 7.1: (a) Digital restricted Voronoi diagram. (b) Stars constructed from the digital restricted Voronoi diagram; they might not be consistent.

7.1 The gHull Algorithm

The main idea of gHull is to utilize the relation between the 3D Voronoi diagram and the convex hull computed from the same point set S. In particular, only the Voronoi cells of the extreme vertices of S are unbounded, i.e., extend to infinity. Thus, one can first identify these Voronoi cells to derive the extreme vertices of S. Traditionally, this observation is not computationally useful as the Voronoi diagram $\mathcal{V}(S)$ structure is harder to manage than the convex hull, and is just as expensive to compute. But, on the GPU the Parallel Banding Algorithm (PBA) [CTMT10] can compute the digital Voronoi diagram very efficiently and it is a good starting point to derive an approximation of $\mathcal{CH}(S)$.

In gHull, we enclose the input point set S in a box $\mathcal{B}$ that contains integer grid points, each corresponding to one unit cell of $\mathcal{B}$. We use the boundary of $\mathcal{B}$ to capture the unbounded Voronoi cells of S, meaning we only compute six slices of the 3D Voronoi diagram. Theoretically, if $\mathcal{B}$ is large enough, dualizing $\mathcal{V}(S)$ restricted to the faces of $\mathcal{B}$, i.e. the *restricted Voronoi diagram* (Figure 7.1(a)), gives us $\mathcal{CH}(S)$. However, since the Voronoi diagram we compute is in the digital space, and due to the finite size of $\mathcal{B}$, we can only obtain an approximation of the convex hull. We apply star splaying to transform this approximation into the convex hull. Star splaying plays an important role in gHull as a repairing tool, and it is the only known algorithm to compute the convex hull from such an approximation, which may not even be a polyhedron.

The gHull algorithm can be split into five stages:

Stage 1. Voronoi Construction: compute the six 2D slices of the 3D digital Voronoi diagram of S on the boundary of $\mathcal{B}$. Let S' be the set of points whose Voronoi cells appear on these slices.

Our aim is to approximate $\mathcal{V}(S)$ restricted to the six faces of $\mathcal{B}$. We first translate and then scale the input points such that their bounding box fits inside the box $\mathcal{B}$. Then,

we compute the digital Voronoi diagram of S intersecting each side of the boundary of $\mathcal{B}$ on the GPU. To do so, for each side, we project the points onto it, recording one nearest point among those that fall onto the same 2D grid cell. The two coordinates of a point are shifted to the center of the nearest 2D grid cell, while the third coordinate (the distance to the side we are projecting on) is unmodified. We then apply PBA to compute the digital Voronoi diagram.

Stage 2. Star Creation: dualize the digital restricted Voronoi diagram to obtain for each point s in S' a set of neighbors, called the *working set* of s and use that to construct a *convex cone*, represented as a *star*, for the point.

We dualize the digital restricted Voronoi diagram obtained in the previous stage to get a set of triangles. The corners of grid cells are grid vertices, each of which is incident to a maximum of 4 different Voronoi cells. Each grid vertex incident to 3 or 4 different Voronoi cells is dualized into one or two non-intersecting triangles respectively. Ideally, dualizing the (non-digital) restricted Voronoi diagram of S on a closed box results in a polyhedron, not necessarily convex, approximating $\mathcal{CH}(S')$. However, in the digital restricted Voronoi diagram, a Voronoi cell can be, for example, disconnected, resulting in the dualized polyhedron having holes or duplicated triangles. Instead of constructing a polyhedron, we only record the information on the adjacencies of the Voronoi cells. For each triangle thus obtained, we add each two vertices of the triangle to the working set of the third one.

For each s in S' in parallel, we create its star (in the continuous space) from its working set such that its cone is convex (Figure 7.1(b)). Each GPU thread handling a point s first creates an initial star from 3 points in the working set, and then incrementally inserts the rest using the beneath-beyond algorithm.

Stage 3. Hull Approximation: apply the star splaying algorithm to obtain the convex hull $\mathcal{CH}(S')$; see Section 3.3.2 for this algorithm.

To perform the star splaying algorithm in parallel while achieving regularized work for different GPU threads, we carry out the inconsistency checking and the insertions of points in two separate steps, alternately performed until all the stars are consistent. Any inconsistency between two alive stars can generate up to 4 insertions, two each from the link of this edge on each star. For any dead vertex, we only maintain 4 neighbors as its death certificate so that inconsistency between an alive star and a dead star only generate 4 insertions. The insertion step is done by first sorting the set of insertions by the indices of the stars they are destined for, and then assigning each thread to perform the insertions into a star independent of others.

Stage 4. Point Addition: collect points of S that lie outside $\mathcal{CH}(S')$ and for each of them construct its star using its nearby vertices of $\mathcal{CH}(S')$.

Due to $\mathcal{CH}(S')$ being an approximation, it may not contain all extreme vertices of S. We use $\mathcal{CH}(S')$ to check the points in S and remove those inside the hull; this is the reason why we construct $\mathcal{CH}(S')$ in Stage 3. The rest of the points can potentially

be extreme vertices. We first perform the checking in the digital space by rendering the triangles of $\mathcal{CH}(S')$ with the viewing direction orthogonal to each side of $\mathcal{B}$ in turn. Then, we use a depth test to eliminate points that clearly lie inside $\mathcal{CH}(S')$. Each GPU thread handling a point s in S projects s onto each side of $\mathcal{B}$ and compares its depth value d_s with the value d on the depth map on that side (with depth value increasing in the viewing direction). If $d_s - d \leq \tau$ where τ is a predefined constant, then s is potentially an extreme vertex. The depth test is done in the digital space, so a conservative threshold (τ equals 1 pixel width) is used to remove non-extreme vertices; see Section 7.3.1 for the proof that this threshold is safe.

To further eliminate non-extreme vertices, we perform another round of checking in continuous space. For each point s that passes the depth test, we also record a triangle A that covers its projection in one of the viewing directions. Notice that A is close to s. Pick an arbitrary point r on $\mathcal{CH}(S')$. The point s is either beyond one of the triangles in the star of r, or the ray $\overrightarrow{rs}$ intersects with $\mathcal{CH}(S')$ at a triangle not in the star of r. Using a walking technique similar to point location, starting from A we can quickly find such a triangle B, and accurately determine whether s is inside or outside $\mathcal{CH}(S')$. If s is outside, we use B to form the initial star of s, otherwise it is eliminated. All this computation can be done on each point independently in parallel. The new stars together with $\mathcal{CH}(S')$ form an approximation of $\mathcal{CH}(S)$.

Stage 5. Hull Completion: perform star splaying again to obtain $\mathcal{CH}(S)$.

7.2 Implementation Details

Due to the nature of the GPU, it is more efficient to allocate memory in large chunks rather than dynamically allocate many small blocks. In our implementation, we use two lists to store the description of the stars and their edges, called the *star list* and the *edge list*, as shown in Figure 7.2. Each star has a contiguous chunk of memory whose size is enough to store its current edges plus a certain amount of free space. Each star records the coordinates of its point, its status (whether it is dead), the number of edges, the size of memory allocated for it, and the starting location of its storage in the edge list. Each edge of a star records the index of the other endpoint, and a flag for checking of consistency. The edge list of a star represents its link vertices in counter-clockwise order.

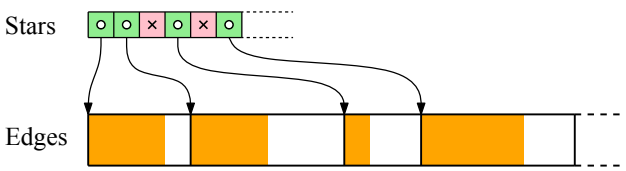


Figure 7.2: Data structures for stars and edges. × indicates a *dead* star.

Algorithm 8: Projecting a chunk of points onto a tile in each block

```
1  declare a k × k real number array A in shared memory for the tile
2  foreach element of A do in parallel
3      initialize the element using maximum real number
4  foreach point of the chuck do in parallel
5      find the pixel and the element of A corresponding to the point
6      compute the distance between the point and the pixel
7      use atomic minimum operation to store the distance into the element
8  foreach element of A do in parallel
9      output the value into global memory
10 foreach point of the chunk do in parallel
11     find the pixel and the element of A corresponding to the point
12     compute the distance between the point and the pixel
13     if the distance is equal to the value of the element then
14         output the index of the point into global memory
```

The difficulty here is that the edge list has a dynamic size as stars are shrinking as well as expanding during star splaying. Any time a star uses up its chunk of allocated storage, we have to expand the edge list. We also use this opportunity to shrink or expand the storage of all the stars to maintain some free space (say 20%) for each star. This helps to reduce the number of times we need to reallocate the edge list. Note that the size of the free space does not affect the performance much, as we observe in our experiment, because the list expansion is relatively cheap compared to the other processing on the GPU. Also, since we start star splaying with a good approximation of the convex hull, the stars typically do not grow drastically.

Voronoi Construction

Before applying the PBA, we need to project the points on the six sides of the box $\mathcal{B}$. This operation entails a lot of random atomic memory accesses to the global memory that are highly inefficient on the GPU. Instead, we perform all the projections in the GPU shared memory to speed up this stage.

$\mathcal{B}$ is partitioned into bricks, each of size $k \times k \times k$. For each point in S, we find the brick that encloses it using a GPU kernel. We accumulate the points that belong to the same brick into a contiguous chunk using a sorting primitive. We identify the starting offset of each such chunk in the sorted list using another kernel.

We use six textures to store the projections on the six sides of $\mathcal{B}$. For each $k \times k$ tile of a texture, use a block of threads to process the points enclosed in the bricks that project onto this tile. Algorithm 8 shows the details of projecting these points onto the tile in each block by one kernel. When many points project onto one pixel, we store the point closest to the tile by using the atomic minimum operation (Line 7). This is

applied on a shared memory array of the block and thus is highly efficient compared to using it on global memory. The constant k can typically be chosen to be 32, so that the $k \times k$ tile can fit in shared memory. The result of these projections is coherently written to global memory to apply PBA and obtain the digital restricted Voronoi diagram.

Star Creation

We construct the working set for each point by scanning the resulting Voronoi diagram textures constructed in the previous stage. For each triangle identified, we generate 6 pairs of its vertices, each pair (a, b) indicating that b is in the working set of a. First, we let one kernel count the number of pairs generated by each grid corner. Next, we pre-allocate an array to store the pairs, and use a prefix sum primitive to compute for each corner the offset in the array to store its pairs. After that, we call another kernel to scan the textures again, generating the working set pairs. Lastly, we sort the list of pairs using a sorting primitive, remove duplicates, and identify the working set for each point as a contiguous chunk of pairs.

Based on the working sets thus constructed, we allocate the storage for the star list and the edge list. A kernel is used to construct an initial star consisting of 3 link points for every point in S'. Each thread constructing an initial star takes 3 points from its working set, checks the 3D orientation, and stores these points in the edge list of that star in counter-clockwise order.

After that, the rest of the working set of each point is inserted into its star in a single kernel. Each thread processes the working set of a point, independent of other points. For each insertion of t into s, we go through the star of s, identifying a (continuous) series of beneath triangles, removing their corresponding edges and inserting t into the edge list of s accordingly.

The beneath-beyond insertion relies heavily on the 3D orientation predicate. It is important that the predicate is computed exactly and co-planar cases are handled correctly. More importantly, the predicate should give the same result when checked from different stars for the star splaying algorithm to converge. In order to achieve this, all our predicates are performed with the Simulation of Simplicity (SoS) technique [EM90] and exact arithmetic [She97].

Hull Approximation

In this stage the star splaying algorithm is adapted for the GPU. The pseudocode of the star splaying implementation on the GPU is outlined in Algorithm 9. In Line 3, we use a compaction primitive on the edge flags to obtain the list of edges to be checked for consistency. Each inconsistent edge can potentially lead to up to four insertions into different stars (see Section 3.3.2). We pre-allocate storage for these possible insertions in Line 4. In Line 5, we use a kernel where each thread processes one edge.

Algorithm 9: Star splaying on the GPU

```
1  flag all edges to be checked for consistency
2  repeat
3  |  collect the edges that are flagged
4  |  allocate space for possible insertions
5  |  check the flagged edges and generate insertions
6  |  sort and compact the list of insertions
7  |  if a star needs more space then
8  |  |  expand the edge list
9  |  perform the insertions to splay stars
10 |  flag edges that need to be checked in the next iteration
11 until there are no more flagged edges
```

The insertions are sorted and compacted in Line 6 and duplicates are removed. Each star then checks if it has enough free space in its edge list and the edge list is expanded if needed (Line 7–8). This expansion is done by computing the required space for each star using a kernel, allocating a new edge list, and then copying the edges over. The insertions (Line 9) are performed similar to those in star creation (Stage 2). In Line 10, we flag all newly created edges. Also, during the insertions, if an edge $\overline{ab}$ in the star of a is deleted, then the edge $\overline{ba}$ in the star of b, if any, needs to be flagged too.

Point Addition

The first round of checking in this stage is carried out in OpenGL, which works seamlessly with other stages done on the GPU. As we keep edges rather than triangles, we first use a kernel to generate the triangles of $\mathcal{CH}(S')$ from the stars. To avoid generating duplicate triangles, each triangle $\triangle abc$ is created from the star of the vertex having the minimum index among the three. Similar to other stages, we first count, then use a prefix sum primitive to compute the offset before actually generating the triangle list.

When a triangle is rendered, we record in the color buffer the index of one of the three vertices so that we can use it as the starting point for our point location in the second round of checking. After the rendering, the depth buffer is processed by a kernel. Each thread processing a point in $S \setminus S'$ checks the depth value to see whether the point can potentially be outside or not. If outside, this point becomes a candidate for the next round of checking.

The second round of checking is explained in Algorithm 10. We use one thread to check one candidate found in the previous round. Let the candidate be s and the corresponding point recorded at the projection of s in the color buffer be c. Also, let r be an arbitrary point in S' where $r \neq c$. In order to determine the triangle B in $\mathcal{CH}(S')$ that is intersected by the ray $\overrightarrow{rs}$, we start walking from c (Line 3–17). Each vertex t on the link of c together with the line rc forms a plane, and we are interested in the

Algorithm 10: Second round of checking in one kernel

```
1  foreach candidate point s do in parallel
2      read the corresponding point c from color buffer
3      repeat
4          p ← ∅, q ← ∅
5          foreach triangle △cvv′ in the star of c do
6              if r is one of v and v′ then continue
7              if s is beyond rcv and beneath rcv′ then
8                  p ← v, q ← v′
9                  break
10         if p ≠ ∅ and q ≠ ∅ then
11             if s is beyond rvv′ then
12                 B ← △cvv′
13             else
14                 c ← v
15         else
16             B ← the first triangle in the star of r
17     until the triangle B is found
18     if s is beyond B then
19         construct initial star of s using B
20     else
21         label s as non-extreme vertex
```

half-plane defined by rc that contains t. The collection of these half-planes partitions the space into several unbounded subspaces around rc; one of these subspaces contains s, which can be identified using 3D orientation checks. This subspace tells us which vertex on the link of c gets us closer to s, until we reach B. Specially, when r is in the star of c, it is possible that none of the subspaces contains s. In this case, s must be beyond one of the two triangles incident to r in the star of c, and we select the first triangle from the star of r as B. After that, using one more 3D orientation check, we can determine accurately if s is outside $\mathcal{CH}(S')$, in which case the three vertices of B form the initial star of s (Line 18–21).

7.3 Digital Approximation Issues

In this section, we discuss some issues of using the digital space. Note that gHull has two places related to the digital space: it computes the digital restricted Voronoi diagram using six textures, and retrieves extreme-points that are previously missed by rendering triangles in the buffers of OpenGL.

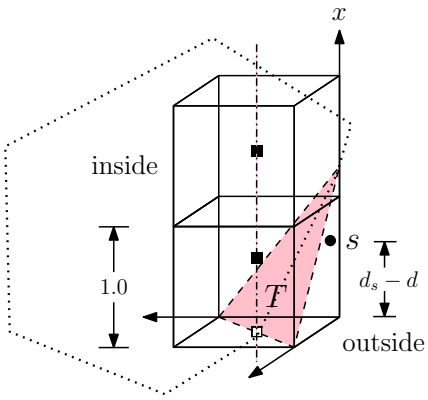


Figure 7.3: The digital depth test of a point s against a triangle T on the boundary of $\mathcal{CH}(S')$ when s is outside $\mathcal{CH}(S')$.

7.3.1 Digital Depth Test

In point addition (Stage 4), we use the six sides of the boundary of $\mathcal{B}$ as the viewing planes. We compare the depth d_s of each point s with the minimum depth value of $\mathcal{CH}(S')$ at the corresponding projection of s to quickly exclude points that are inside $\mathcal{CH}(S')$. However, since the depth buffer we obtain when rendering $\mathcal{CH}(S')$ is of finite resolution, the depth value d of the projection of s is actually the depth value of the center of the cell containing this projection. Depending on the triangle covering that projection, $(d_s - d)$ can be arbitrarily large; see Figure 7.3. The following claim shows that as long as we keep every point s that has $(d_s - d) < 1$ in one of the projections, we do not miss any point outside $\mathcal{CH}(S')$. This tight bound allows us to throw away most of the points that are inside $\mathcal{CH}(S')$.

Lemma 7.1. *Let $s \in S \setminus S'$ be a point outside $\mathcal{CH}(S')$. In (at least) one of the six renderings of $\mathcal{CH}(S')$ orthogonal to a side of $\mathcal{B}$, we have $(d_s - d) \leq \tau$ where $\tau = 1$ pixel width.*

Proof. The point s is inside a unit cell of $\mathcal{B}$ whose center is the grid point $(\bar{x}, \bar{y}, \bar{z})$. The coordinates 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 the cells containing the projections of s in different viewing directions, and the plane equation of T be $ax + by + cz + K = 0$. Without loss of generality we assume that $a \geq b \geq c$.

Since T appears in the depth buffer, and $\mathcal{CH}(S')$ is convex, T must be visible from three different viewing directions. This forms a coordinates 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 $\mathcal{CH}(S')$ 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:

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

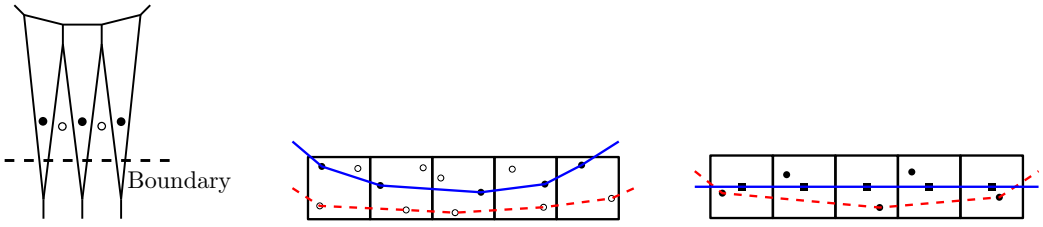
It is possible that the depth values of s used in checking in the six viewing directions belong to different triangles. Suppose that the depth value of triangle T is used in one of the directions, then from the above argument, there is one direction in which the depth d of the plane containing T fulfills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

7.3.2 Convex Hull Approximation

There are three issues related to the use of digital Voronoi diagram that can affect the performance of gHull: *slicing problem*, *under-approximation problem* and *over-approximation problem*; see Figure 7.4. In the experiment, we will show that these problems affect the efficiency of the algorithm in different cases and manners.

Slicing problem. This problem is the result of using a bounded box $\mathcal{B}$ to find the Voronoi cells that are unbounded. As some of the bounded cells can extend beyond $\mathcal{B}$, they are captured although they do not correspond to extreme vertices. Figure 7.4(a) shows a 2D example where among the five cells being captured, only those of the round white points are unbounded. To reduce the number of wrongly captured Voronoi cells, we scale the point set to a slightly smaller volume inside $\mathcal{B}$ when performing Voronoi construction (Stage 1).

Under-approximation problem. When we have multiple points projected to the same pixel, we can only record one point, and thus there are potentially many more points outside $\mathcal{CH}(S')$. See Figure 7.4(b) for a 2D illustration where the round black points are kept, the solid line denotes part of $\mathcal{CH}(S')$ and the dashed line denotes part of $\mathcal{CH}(S)$. The round white points are missing points, many of which are outside $\mathcal{CH}(S')$.



(a) Slicing (b) Under-approximation (c) Over-approximation

Figure 7.4: Three problems associated with the computation in the digital space.

By using an efficient depth test in point addition (Stage 4) of our implementation and accurate location of a nearby triangle for every point outside $\mathcal{CH}(S')$, we are able to construct a good star for that point. This reduces the amount of splaying needed in hull completion (Stage 5).

Over-approximation problem. This problem is caused by the shifting of points in Voronoi construction (Stage 1). In certain cases, for example when points are distributed near the surface of a cube axis-aligned with $\mathcal{B}$, many extreme points are shifted inward, while many non-extreme points are shifted outward and are legitimately captured. This possibly leads to a lot more points captured in Stage 1 and need to be removed in hull approximation (Stage 3). See Figure 7.4(c) for a 2D illustration, where after Stage 1 all the round black points, after shifted to the square black grid points, are captured. In our implementation, for each side of $\mathcal{B}$ we only shift 2 coordinates of the points while keeping the third one untouched. This produces a much better digital restricted Voronoi diagram and thus reduces the effect of this problem.

7.4 Experiment

We compare the GPU implementation of gHull with Qhull and CGAL. We also integrate the experimental results of ffHull shown in the previous chapter.

All the results of gHull are based on the same set of parameters: grid size 1024^3 (i.e. each slice is of size 1024^2), while point set is scaled to 80% of the volume of $\mathcal{B}$. The rendering buffer in point addition (Stage 4) is fixed at 512^2. Using a larger grid size gives a better approximation at the cost of slower Voronoi diagram computation, so it gives little running time improvement. A larger buffer for the depth test is also not desirable, since it incurs extra rendering cost.

Following the experiment in Section 6.5, we generate points randomly with coordinates between $[0.0, 1.0]$ in four distributions: a cube, a ball of radius 0.5, a box with thickness of 0.01, and a sphere with thickness of 0.01; see Figure 6.12.

Running time

Figure 7.5 shows the running time of gHull and ffHull on the GPU. As Qhull is much faster than CGAL, we only show the speedups of gHull and ffHull over Qhull. In general, gHull is 4 to 17 times faster than Qhull (11 to 70 times faster than CGAL) for the cube, ball and box distributions. Notably, for the sphere distribution gHull is up to 30 times faster than Qhull (100 times faster than CGAL), even with all the computation being exact. This is mainly because the digital restricted Voronoi diagram gives a very good approximation. However, ffHull runs much faster than gHull for all the four distributions of points. Notably, ffHull runs around 5 times faster than gHull for the cube and the box distributions where very few points are extreme.

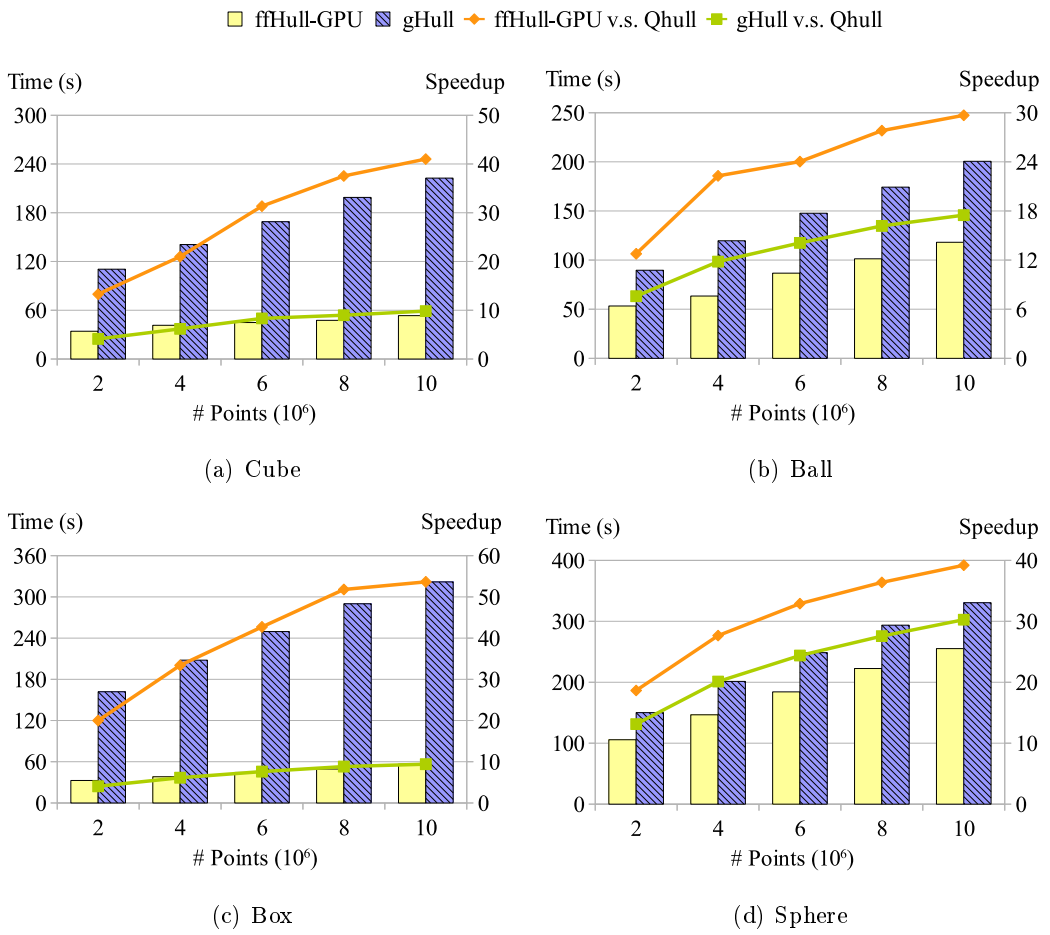


Figure 7.5: Running time of gHull and ffHull on the GPU, and their speedups over Qhull.

Table 7.1 shows the running time of gHull and ffHull for models from the Stanford 3D scanning repository [Sta12]. As we analyze in Section 6.5, most of the points in these models are close to the surface while only very few points are extreme. In this test, the running time of gHull and ffHull on the GPU is very close, mainly because both of them suffer from a large amount of coplanar points. The speedups of gHull over Qhull are from 3.6 to 7 times; the those over CGAL are from 7.9 to 16.9 times.

Table 7.1: Running time of gHull, ffHull, CGAL and Qhull on 3D models.

Model	# Points (millions)	Running time (ms)			
		CGAL	Qhull	gHull	ffHull-GPU
Asian dragon	3.6	1181	540	150	117
Thai statue	5.0	1538	692	168	91
Lucy	13.9	4488	1884	266	192

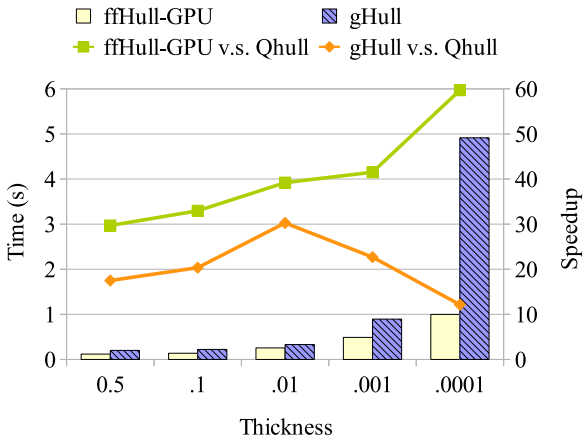


Figure 7.6: gHull v.s. ffHull on the GPU with points on a sphere of different thicknesses.

Sensitivity

The number of points on the convex hull directly affects the running time of both ffHull and gHull. When many points are close to the boundary of the convex hull, gHull will have serious slicing problem and approximation problem. We study the sensitivity of gHull to the thickness of the sphere distribution. As same as the previous section, we test the running time of gHull using 10^7 points on the sphere distribution with thickness varying from 0.5 to 0.0001; see Figure 7.6.

The running time of gHull increases quickly as the sphere gets thinner. It slows down much when the thickness becomes 0.0001, which is only 0.1 pixel width given that we use a 1024^3 grid size. The main reason is that gHull cannot obtain a good approximation by extracting information from the six slices of digital Voronoi diagram for the under-approximation and the over-approximation problems. Its speedup over Qhull initially increases as the Quickhull algorithm becomes less effective in eliminating non-extreme vertices, then decreases but is still more than 10 times faster.

In contrast, the ffHull algorithm is also very efficient when many input points are close to the boundary of the convex hull. When the thickness of the sphere is smaller than 0.001, ffHull performs much better than gHull. The main reason is that ffHull compared with gHull can accurately remove many non-extreme points during its constructing stage. Another reason is rooted in the simplicity of flipping.

Scalability on the number of extreme and non-extreme vertices

In order to investigate the effect of the number of extreme vertices and non-extreme vertices on the performance of gHull, we use a different ball distribution, called controlled ball, in which we first generate h points randomly on a sphere, and then generate $n - h$ points randomly inside a ball of slightly smaller radius. This gives us a point set

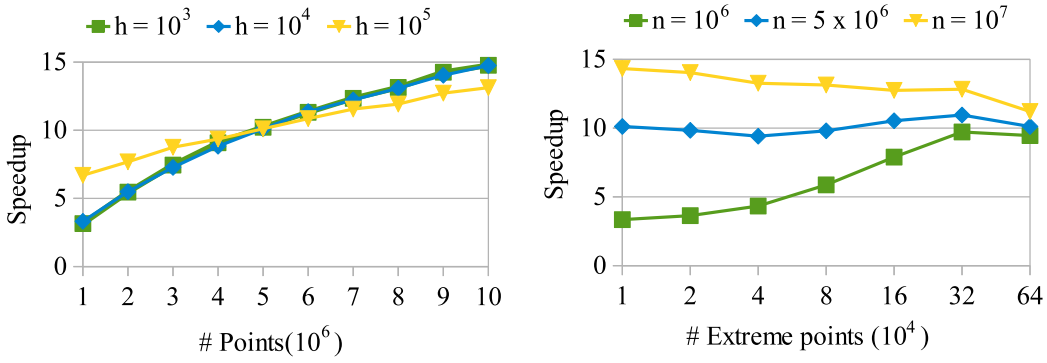


Figure 7.7: Speedup of gHull over Qhull by (a) fixing the number of extreme vertices h while varying the total number of points n, and (b) fixing the total number of points n while varying the number of extreme vertices h.

with n points, out of which h points are extreme vertices.

Figure 7.7(a) shows the speedup of gHull over Qhull when we fix h and vary n in the range of 10^6 to 10^7. As n is larger, the speedup increases from 4 to 15 times. Note that the speedup with $h = 10^3$ and $h = 10^4$ is very close, while is slightly lower when $h = 10^5$. This is the consequence of the under-approximating problem when the texture cannot capture all the extreme vertices due to its limited size.

On the other hand, Figure 7.7(b) shows the speedup of gHull over Qhull when we fix n and vary h multiplicatively from $2^0 \times 10^4$ to $2^6 \times 10^4$. For $n = 10^6$ the speedup increases as h becomes larger, because gHull can quickly capture extreme vertices and remove non-extreme vertices compared with Qhull. Also, speedup is higher for larger n, as reflected earlier in Figure 7.7(a). On the other hand, the speedup slightly decreases for $n = 10^7$ as h becomes larger because of the under-approximating problem. The explanation here is that when n is larger, there are more points near the convex hull boundary, with multiple points falling on the same pixel. As such, many extreme vertices are not captured.

A similar behavior can be observed when gHull is compared with CGAL, with the speedup being 3 to 4 times better.

Time breakdown

Figure 7.8 shows the running time of each stage of gHull on different distributions with 10^7 points. As expected, the behavior differs on different distributions. While the running time of Stage 1 and Stage 4 remains the same since it is not affected by how the points are distributed, the running time of other stages varies significantly. Stage 3 takes more time on the box distribution due to the over-approximation problem, while

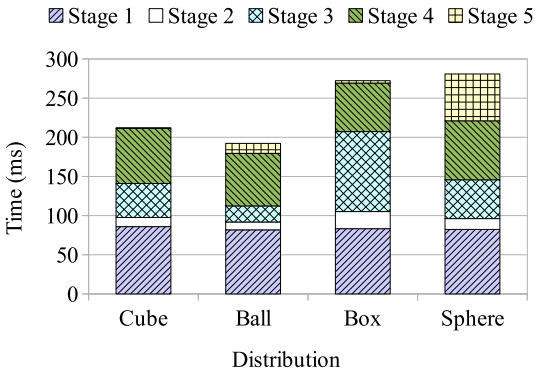


Figure 7.8: Time breakdown of gHull.

Stage 5 takes more time on the sphere distribution due to the under-approximation problem. Stage 2 only takes a small portion of running time for all the distributions. For each distribution, the time of transferring data between the CPU and the GPU takes 24%–33% of the computing time.

Limitation

While the digital space allows us to perform most processing in parallel with regularized work and localized data, its limitation to approximate the computation in the continuous space lies in its uniformity. It is possible to design a test case where points are badly distributed (e.g. points arranged on a thin line convoluted in the space), resulting in a bad digital approximation, and thus lower overall performance. Such a case, however, is not common in practice. The next issue with gHull is with the use of SoS during the star splaying process. Since the digital approximation cannot take into account the perturbation, the resulting approximation might be different from the final result with SoS, especially when there are many perfectly co-planar points on the convex hull. In this case, the algorithm performs a large amount of exact computation for no good purpose. Lastly, our current implementation requires at least 3 times more memory compared to CGAL and Qhull. Part of the reason is because in order to achieve a very high level of parallelism, gHull needs to use some large textures and maintain several auxiliary arrays for parallel primitives. Also, the data structure used during the star splaying process is more costly than the standard triangulation data structure used in other approaches.

CHAPTER 8

Local Transformation to Star-Shaped Polytope

This chapter investigates the local transformation from an arbitrary polytope, possibly with self-intersection, to a star-shaped polytope in $\mathbb{R}^2$ and $\mathbb{R}^3$.

Star-shaped polytope is the only known input acceptable to flip-pop and flip-flop. To extend the use of these algorithms, it is interesting to study local transformation to compute star-shaped polytope from an arbitrary one. On the premise that all simplices of a polytope are oriented in a consistent manner, we can utilize the orientations of simplices to guide the transformation. Section 8.1 reviews the notion of coherently oriented polytope. Section 8.2 introduces a novel twist operation, a local operation on abstract simplicial complex, and explains twist and flip on coherently oriented simplicial complex.

Section 8.3 studies the local transformation to 2D star-shaped polygon. We present a simple twist algorithm to transform an arbitrary polygon to a star-shaped one w.r.t. a given point. The algorithm is robust since it only uses orientation checks as predicates. Section 8.4 attempts to solve the same problem for 3D. We show a provably correct algorithm called *twist-flip* for the special case of polyhedron with one extreme vertex connecting to all the other vertices of the polyhedron. For the general case of arbitrary polyhedron, we present our preliminary findings toward designing a proper algorithm, while leaving the problem itself open.

8.1 Coherent Orientation

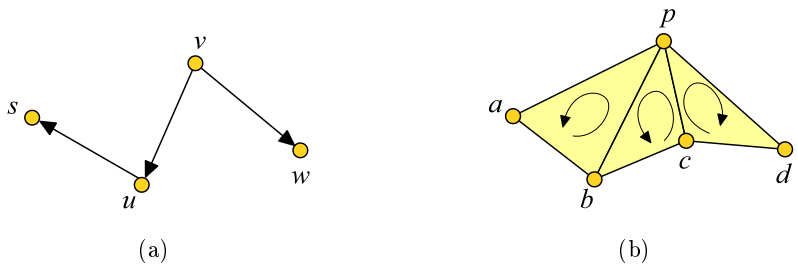
Given a k-simplex C whose vertices are $\{p_0, p_1, ..., p_k\}$, we embed it in $\mathbb{R}^{k+1}$. Let q be an arbitrary point not on the hyperplane defined by C. A *vertex ordering* $\langle p_{a_0}, p_{a_1}, ..., p_{a_k} \rangle$ of C, where $a_i \in \{0, 1, ..., k\}$ for $0 \leq i \leq k$, is a permutation of its vertices. Two vertex orderings $\langle p_{a_0}, p_{a_1}, ..., p_{a_k} \rangle$ and $\langle p_{b_0}, p_{b_1}, ..., p_{b_k} \rangle$ of C are *equivalent* if

$$\begin{vmatrix} p_{a_0}, & 1 \\ p_{a_1}, & 1 \\ ..., & ... \\ p_{a_k}, & 1 \\ q, & 1 \end{vmatrix} = \begin{vmatrix} p_{b_0}, & 1 \\ p_{b_1}, & 1 \\ ..., & ... \\ p_{b_k}, & 1 \\ q, & 1 \end{vmatrix},$$

where $|\cdot|$ denotes the determinant. Simply, two vertex orderings of C are equivalent if they can be transformed to each other by an even number of swap of vertices.

By the definition above, C has exactly two *orientations*: one corresponds to the vertex orderings making positive determinant with q and the other corresponds to those making negative determinant. We usually represent an orientation of C by a corresponding vertex ordering. Take a 2-simplex (triangle) with three vertices $\{a, b, c\}$ for example. One of its orientations is $\langle a, b, c \rangle$ (or equivalently $\langle b, c, a \rangle$ or $\langle c, a, b \rangle$), while the other is $\langle a, c, b \rangle$ (or equivalently $\langle c, b, a \rangle$ or $\langle b, a, c \rangle$). For a k-simplex, each orientation corresponds to $(k+1)!/2$ vertex orderings. In the remaining discussion, any simplex is associated with an orientation.

The orientation of C induces the orientations of its facets. Let $\langle p_{a_0}, p_{a_1}, ..., p_{a_k} \rangle$ be a vertex ordering representing the orientation of C. By removing its first item, we get $\langle p_{a_1}, ..., p_{a_k} \rangle$; this vertex ordering represents the orientation of the facet whose vertices are $\{p_{a_1}, ..., p_{a_k}\}$. To determine the orientation of an arbitrary facet of C, we first find a vertex ordering of C which is equivalent to $\langle p_{a_0}, p_{a_1}, ..., p_{a_k} \rangle$ and whose first vertex is not in the facet, and then remove that vertex from the ordering. For example, if the



(a) (b)

Figure 8.1: (a) $\overrightarrow{vu}$ and $\overrightarrow{us}$ are coherently oriented, while $\overrightarrow{vu}$ and $\overrightarrow{vw}$ are not. (b) $\triangle pab$ and $\triangle pbc$ are coherently oriented, while $\triangle pbc$ and $\triangle pdc$ are not. The arrows indicate orientations.

Figure 8.2: (a) A coherently oriented polygon. (b) A coherently oriented polyhedron. Orientations are indicated by arrows.

orientation of a triangle is $\langle a, b, c\rangle$, the orientations of its three edges are $\langle b, c\rangle$, $\langle c, a\rangle$, and $\langle a, b\rangle$. Recursively, we can determine the orientations of all the faces of C except for its vertices.

Given two different k-simplices ($k > 1$) that share a common facet, they are *coherently oriented* if the orientation of the common facet in the first simplex is not equivalent to that in the second simplex. For the case of 1-simplices (edges), two adjacent edges are *coherently oriented* if their common vertex appears at different positions in their vertex orderings. We use $\overrightarrow{ab}$ to represent an edge whose vertex ordering is $\langle a, b\rangle$; $\overrightarrow{ab} \neq \overrightarrow{ba}$. We use $\triangle abc$ to represent a triangle whose orientation can be represented by $\langle a, b, c\rangle$; $\triangle abc = \triangle bca = \triangle cab \neq \triangle bac = \triangle acb = \triangle cba$. Figure 8.1 shows some examples of coherently oriented simplices in $\mathbb{R}^1$ and $\mathbb{R}^2$.

Recall that a d-dimensional polytope is a pure $(d-1)$-dimensional abstract simplicial complex where each ridge is shared by exactly two facets (see Section 2.1). By this definition, a polytope is a $(d-1)$-manifold without boundary and homeomorphic to a $(d-1)$-sphere topologically. Therefore a polytope is *orientable*, i.e., there is a way to set the orientations of each facet so that every pair of adjacent facets are coherently oriented. Using this way, we say that the polytope is *coherently oriented*. Figure 8.2 shows a polygon and a polyhedron both coherently oriented. In the remaining of this chapter, we assume a polytope is always coherently oriented unless otherwise stated.

8.2 Twist and Flip on Coherently Oriented Triangulation

Given a d-dimensional abstract triangulation (or polytope) $\mathcal{T}$ that is coherently oriented, we represent each of its facets as a vertex ordering, and $\mathcal{T}$ as a collection of vertex orderings. A *swap* on two vertices a and b of $\mathcal{T}$ is to replace a with b and replace b with a in the collection of vertex orderings. A *twist* on an edge $\overline{pq}$ is a swap on p and q. Twist is a local operation because it works on an edge and only changes a local configuration, while swap is not. Figure 8.3 and 8.4 show examples of twisting in polygon and planar triangulation respectively. Note that swap and twist are both combinatorial operations in the collection of vertex orderings representing $\mathcal{T}$, and thus

Figure 8.3: Twisting edge $\overrightarrow{pq}$ of the polygon in (a) leads to the polygon in (b).

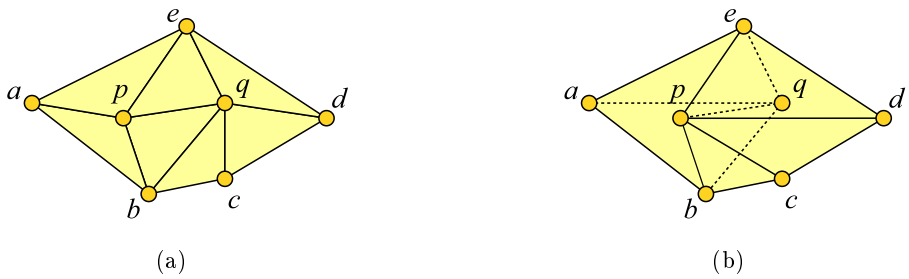


Figure 8.4: Twisting $\overline{pq}$ transforms the triangulation in (a) to the one in (b). The triangulation in (a) is $\{\langle p,a,b\rangle, \langle p,b,q\rangle, \langle p,q,e\rangle, \langle p,e,a\rangle, \langle q,b,c\rangle, \langle q,c,d\rangle, \langle q,d,e\rangle\}$, and the one in (b) is $\{\langle q,a,b\rangle, \langle q,b,p\rangle, \langle q,p,e\rangle, \langle q,e,a\rangle, \langle p,b,c\rangle, \langle p,c,d\rangle, \langle p,d,e\rangle\}$.

do not change the number of the simplices of $\mathcal{T}$. Furthermore, $\mathcal{T}$ is still coherently oriented afterward.

When a flip is applied in $\mathcal{T}$, the orientations of the newly created facets should be assigned so that $\mathcal{T}$ is still coherently oriented. We show how to decide the orientations of these facets from those of the removed ones. Let F be the ridge where the flip is applied on, and $\mathcal{T}_c$ and $\mathcal{T}_r$ be the set of all the facets created and removed by the flip respectively. By definition of flip, $\mathcal{T}_c$ and $\mathcal{T}_r$ have the same boundary. The rule is that, for each ridge on the boundary, its orientation induced from the facets of $\mathcal{T}_c$ should equal to its orientation induced from the facets of $\mathcal{T}_r$. This rule indicates a unique way to assign orientation to the facets of $\mathcal{T}_c$. In Figure 8.5, for example, we flip $\overline{ab}$, transforming $\{\langle a,b,c\rangle, \langle a,d,b\rangle\}$ to $\{\langle a,d,c\rangle, \langle c,d,b\rangle\}$.

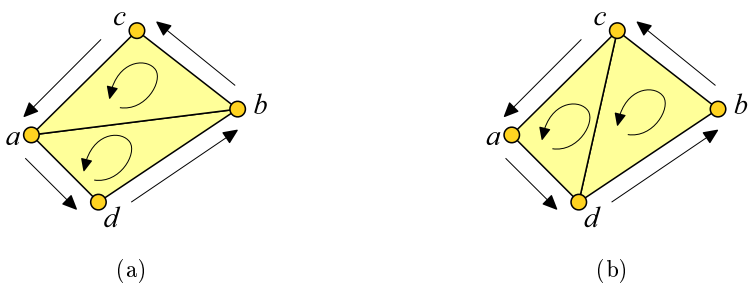


Figure 8.5: Flipping $\overline{ab}$ on a coherently oriented triangulation.

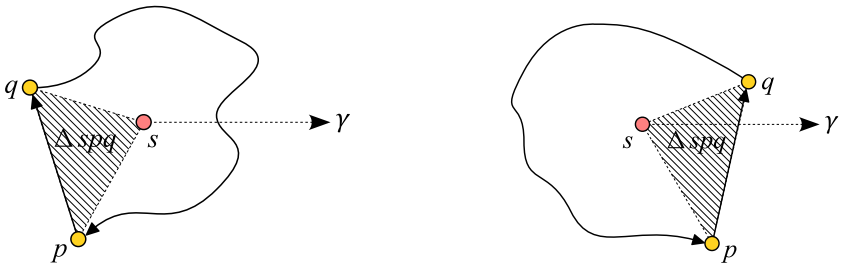


Figure 8.6: The twist algorithm twists the edge $\overrightarrow{pq}$ in two cases: (a) $\overrightarrow{pq}$ does not intersect with γ and $\triangle spq$ is clockwise, and (b) $\overrightarrow{pq}$ intersects with γ and $\triangle spq$ is counter-clockwise, where γ is the ray starting at s with the same direction as positive x-axis.

8.3 Local Transformation to Star-Shaped Polygon

Given a coherently oriented polygon $\mathcal{P}$ with its vertex set S, we design an algorithm that locally transforms $\mathcal{P}$ into a star-shaped polygon of S w.r.t. a given point s inside $\mathcal{CH}(S)$. Since the algorithm applies only twist on the edges of $\mathcal{P}$, we name it *twist algorithm*. In the following discussion, we represent $\mathcal{P}$ as an ordering of its vertices. For example, $\mathcal{P} = \langle p_0, p_1, ...p_{n-1}\rangle$ contains n vertices $\{p_i \mid 0 \leq i \leq n-1\}$, and n oriented edges $\{\overrightarrow{p_i p_{i+1}} \mid 0 \leq i \leq n-1 \text{ and } p_n = p_0\}$. We assume no three points of S lie on the same line.

Let γ be the ray starting at s and having the same direction as positive x-axis. For every vertex p of $\mathcal{P}$, define a function $g(p) \in [0, 2\pi)$ as the angle of rotating γ around s counter-clockwise until γ reaches p. The key idea is to achieve a polygon where all its vertices are sorted in the ascending order of the function g. By the definition of the twist operation, twisting an edge $\overrightarrow{pq}$ swaps the positions of p and q in $\mathcal{P}$. Therefore, we intend to twist $\overrightarrow{pq}$ if $g(p) > g(q)$ unless q is the first vertex of $\mathcal{P}$.

As computing $g(p)$ of a vertex p creates numerical error, we instead use 2D orientation check to compare $g(p)$ and $g(q)$ for two vertices p and q. Specifically, when $\overrightarrow{pq}$ does not intersect with γ, $g(p) > g(q)$ if $\triangle spq$ is clockwise; when $\overrightarrow{pq}$ intersects with γ, $g(p) > g(q)$ if $\triangle spq$ is counter-clockwise. Figure 8.6 illustrates these two cases.

We describe the twist algorithm using the framework shown in Section 3.2. The **geometric structure** is a coherently oriented polygon represented as an ordering of its vertices, and the **local operation** is twist. The **candidate** is an oriented edge of the polygon. The **criterion** of the twist algorithm is to twist an edge $\overrightarrow{pq}$ if (1) q is not the first vertex, and (2) $g(p) > g(q)$. The **local check** is to test the orientation of an edge w.r.t. s and the intersection between the edge and γ.

Algorithm 11 shows the pseudocode of the twist algorithm. The input is $\mathcal{P}$ and an arbitrary point s that lies inside $\mathcal{CH}(S)$. After pushing all the edges of $\mathcal{P}$ into

Algorithm 11: The twist algorithm

input : a polygon $\mathcal{P}$ and a point s inside $\mathcal{CH}(S)$
output: a star-shaped polygon $\mathcal{P}$ w.r.t. s

```
1  s' ← (∞, s.y)
2  Q ← { e | e is an oriented edge of P }
3  while Q ≠ ∅ do
4      pq ← Q.pop()
5      if q is the first vertex of P then continue
6      if (s.y − p.y)(s.y − q.y) < 0 and the orientations of △spq and △s'pq are
       different then
7          if △spq is counter-clockwise then
8              twist pq
9              Q ← {rq, pt}, where rq and pt are the adjacent edges of qp
10     else
11         if △spq is clockwise then
12             twist pq
13             Q ← {rq, pt}, where rq and pt are the adjacent edges of qp
```

a queue $\mathcal{Q}$, the algorithm repeatedly checks the popped edges until $\mathcal{Q}$ is empty. In each iteration (Line 4–13), an edge $\overrightarrow{pq}$ is popped from $\mathcal{Q}$ and checked unless q is the first vertex of $\mathcal{P}$ (which means that p is the last one). In Line 6, we test whether $\overrightarrow{pq}$ intersects with γ. Based on the criterion, we twist $\overrightarrow{pq}$ if $\overrightarrow{pq}$ intersects with γ and $\triangle vpq$ is counter-clockwise (Line 6–9), or if $\overrightarrow{pq}$ does not intersect with γ and $\triangle vpq$ is clockwise (Line 10–13). After twisting $\overrightarrow{pq}$, we push its adjacent edges into $\mathcal{Q}$ for later checks.

Figure 8.7 shows the procedure of transforming a pentagram to a star-shaped polygon w.r.t. its leftmost vertex by the twist algorithm.

Theorem 8.1. *The twist algorithm transforms $\mathcal{P}$ to a star-shaped polygon w.r.t. s, given s as an arbitrary point in $\mathcal{CH}(S)$.*

Proof. By the criterion, the algorithm twists an edge $\overrightarrow{pq}$ if q is not the first vertex and $g(p) > g(q)$. This is equivalent to the bubble sort on the vertex ordering using the function $g(p)$ as the key. The termination of the twist algorithm is promised by that of bubble sort. When the algorithm terminates, the function $g(p)$ defined in $\mathcal{P}$ is monotonically increasing, which implies that $\mathcal{P}$ is star-shaped w.r.t. s. □

The worst-case time complexity of the twist algorithm is $\mathrm{O}(n^2)$ because of bubble sort. The algorithm needs $\Theta(n^2)$ twists if $\mathcal{P}$ is initially star-shaped w.r.t. s, but all its edges are clockwise w.r.t. s. When the input polygon is almost star-shaped w.r.t. a given point, the twist algorithm can be very efficient. An example is the polygon constructed via sorting points by their polar angles in Graham's scan.

Although the twist algorithm is described in its sequential version, it can be easily

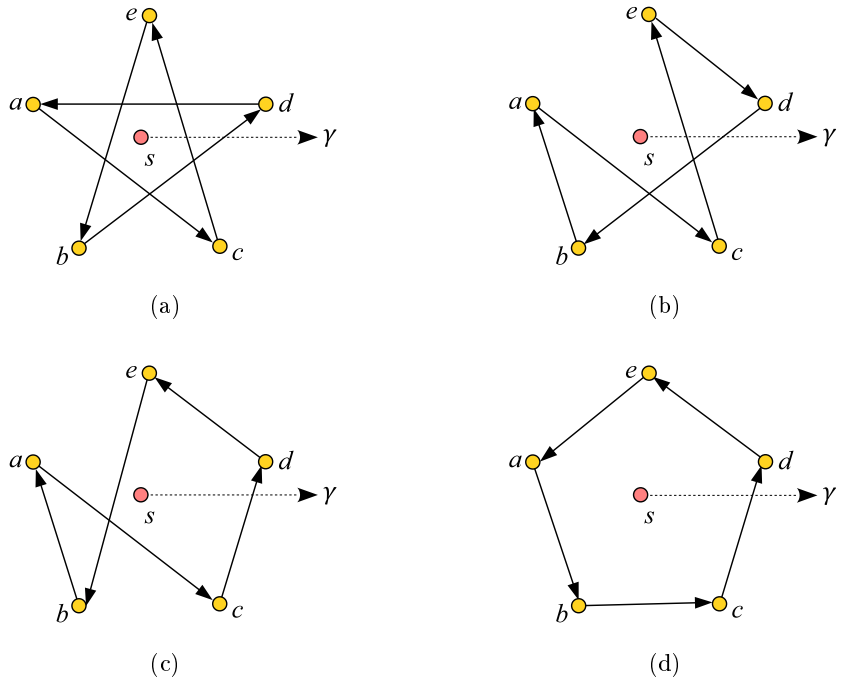


Figure 8.7: The initial polygon in (a) is a pentagram $\langle a, c, e, b, d\rangle$. From (a) to (d) we continuously twist $\overrightarrow{bd}$, $\overrightarrow{ed}$ and $\overrightarrow{ba}$. The final polygon in (d) is a star-shaped polygon $\langle a, b, c, d, e\rangle$ w.r.t. a.

adapted for parallel computation since it has free execution order. Furthermore, we use symbolic perturbation [EM90] to avoid the situation that more than two vertices are collinear. Note that if our outputs are further used to compute convex hull by flip-pop, the same perturbation order must be maintained in order to guarantee the correctness.

8.4 Local Transformation to Star-Shaped Polyhedron

We continue to study the 3D problem: transforming a given polyhedron $\mathcal{P}$ to a star-shaped polyhedron by local operations. Our result for the 2D problem cannot be directly used for the 3D one, since angle for 3D point is not well defined, and even if it can be defined, it can hardly be used for the construction of star-shaped polyhedron. Furthermore, twisting on a polyhedron is a much heavier operation than that on a polygon, as it may affect the stars of many vertices; see Figure 8.4.

To study the problem, we first give a sufficient condition for a coherently oriented k-dimensional polytope to be star-shaped w.r.t. a given point.

Theorem 8.2. *A coherently oriented k-dimensional polytope $\mathcal{P}$ is star-shaped w.r.t. a point s if and only if*

(1) all the facets of $\mathcal{P}$ have the same orientation w.r.t. s, and

(2) there exists a ray γ starting at s and intersecting exactly one facet of $\mathcal{P}$.

Proof. (if) We draw a $(k-1)$-sphere $\mathcal{S}$ using s as the center. For a point x on $\mathcal{P}$, the ray $\overrightarrow{sx}$ intersects with $\mathcal{S}$ at exactly one point y. We map x to y and build a function $f : \mathcal{P} \to \mathcal{S}$. Given an arbitrary point $p \in \mathcal{P}$, for any $x \in \mathcal{P}$ that approaches p, the limit of $f(x)$ exists and is equal to $f(p)$. Therefore, f is continuous. Since $\mathcal{P}$ intersects with γ at exactly one point by Condition (2) and so does $\mathcal{S}$, the degree of f is 1 [Mun84].

By contradiction assume $\mathcal{P}$ is not star-shaped w.r.t. s, and thus there is a ray γ' that starts at s and intersects with $\mathcal{P}$ at m points, $m > 1$. As all the facets of $\mathcal{P}$ have the same orientation w.r.t. s, the existence of γ' indicates that the degree of f is m, a contradiction.

(only if) For every ridge of $\mathcal{P}$, the two facets incident to the ridge are coherently oriented. Hence, when $\mathcal{P}$ is star-shaped w.r.t. s, s must be beyond or beneath both of them, and thus they have the same orientation w.r.t. s. Together with that any two facets of $\mathcal{P}$ is connected, we conclude Condition (1). Condition (2) is naturally proved by the definition of star-shaped polytope. □

Considering the possible difficulty resulted from selecting an arbitrary point as the kernel point, we use the topmost vertex v of $\mathcal{P}$ as the kernel point. When v is $(0,0,\infty)$ and all the other vertices of $\mathcal{P}$ lie in the plane $z = 0$, our problem degenerates into *untangling planar triangulation* problem, which is to transform a 2D abstract triangulation, possibly with self-intersection, to a valid triangulation. This problem is solved by [ASY08] without using local transformation. The main idea of that method is to find the regions affected by the triangles with wrong orientations, and re-triangulate these regions. This algorithm requires that the boundary of the input triangulation is its convex hull. Our proposed problem, on the other hand, is for a more general input geometric structure, and focuses on local transformation.

In the remaining of this chapter, we assume all the vertices of $\mathcal{P}$ are in general position. In Section 8.4.1, we focus on a special type of polyhedron, where the topmost vertex connects to all the other vertices. We design a provably correct twist-flip algorithm to transform it to a star-shaped polyhedron w.r.t. the topmost vertex. In Section 8.4.2, we present our preliminary study on using general polyhedron as input.

8.4.1 Special Case

Let v be the topmost vertex of $\mathcal{P}$. We study a special case where all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v. Despite being restrictive, this case is useful for learning how to design a sufficient condition for $\mathcal{P}$ to be star-shaped w.r.t. v, how to use twist and flip together to fulfill this condition, and how to promise the termination of twisting and flipping. All these techniques are non-trivial and may help to design local transformation for the general case.

We define a few useful concepts to this special case. Shoot a ray from v to the interior of $\mathcal{CH}(link(v))$, and select an arbitrary point $s \neq v$ on the ray. We use s as

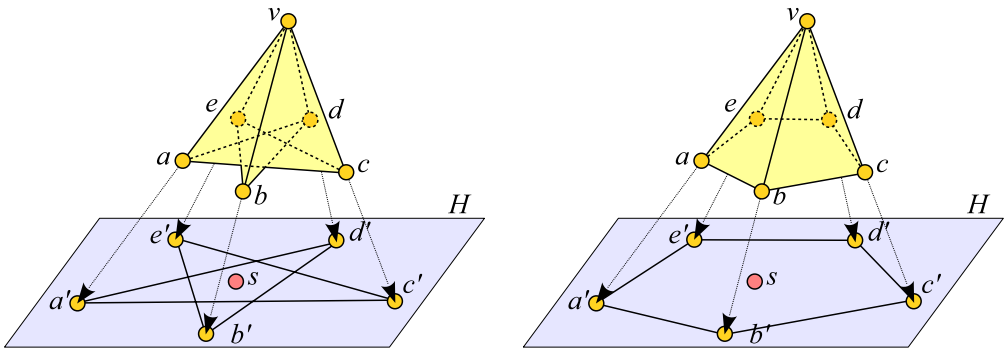


Figure 8.8: (a) Intersecting the extended cone of $star(v)$ and the horizontal hyperplane H through s results into a 2D polygon on H. Build the correspondence between $link(v)$ and the polygon. (b) After transforming the polygon to a star-shaped polygon w.r.t. s, $star(v)$ is also star-shaped w.r.t. s.

the *secondary kernel*; see Figure 8.8(a). Actually s can be the centroid of any three different vertices of $link(v)$. A triangle of $\mathcal{P} \setminus star(v)$ is said to be *positive* if it is counter-clockwise w.r.t. v; otherwise it is *negative*. A triangle of $star(v)$ is *positive* if it is counter-clockwise w.r.t. s; otherwise it is *negative*. An edge shared by a positive triangle and a negative triangle is called a *fold*. Given a triangle t, a point p is *above* t if t is counter-clockwise w.r.t. p; it is *beneath* t otherwise.

Our *twist-flip* algorithm aims to make all the triangles of $\mathcal{P}$ positive, which is sufficient for $\mathcal{P}$ to be star-shaped. The key idea is to first make all the triangles of $star(v)$ positive using twists and then remove all the folds of $\mathcal{P}$ using flips. The algorithm is thus separated into two stages: the *twisting stage* and the *flipping stage*.

The twisting stage

We simulate the twist algorithm on $star(v)$ in order to make it star-shaped w.r.t. s and its triangles positive. Imagine a polyhedral cone created by extending $star(v)$ from v to infinity, and use the horizontal hyperplane H through s to cut this cone. The resulting cross section is a polygon $\mathcal{P}_{2D}$ with s in the interior of its convex hull. Each vertex (resp. edge) of $\mathcal{P}_{2D}$ corresponds to a vertex (resp., an edge) of $link(v)$; see Figure 8.8. Now transform $\mathcal{P}_{2D}$ to a star-shaped polygon w.r.t. s using the twist algorithm, twisting the corresponding edges of $link(v)$ when an edge of $\mathcal{P}_{2D}$ is twisted. At the end, $\mathcal{P}_{2D}$ is star-shaped w.r.t. s, implying that $star(v)$ is also star-shaped w.r.t. s.

Algorithm 12 shows the pseudocode of the twisting stage, which has the same workflow as Algorithm 11. The objects stored in $\mathcal{Q}$ are triangles of $star(v)$, which correspond to edges of $\mathcal{P}_{2D}$. Given a triangle $\triangle vpq$, the edge $\overline{pq}$ needs to be twisted if it needs to be twisted as an edge of $\mathcal{P}_{2D}$ in the twist algorithm; see Line 6–13.

Algorithm 12: The twisting stage of the twist-flip algorithm

input : a polyhedron $\mathcal{P}$ with topmost vertex v and secondary kernel s;
all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v
output: $star(v)$ is star-shaped w.r.t. s and only contains positive triangles

```
1  s' ← (∞, s.y, s.z)
2  Q ← { t | t is an oriented triangle of star(v) }
3  while Q ≠ ∅ do
4      △vpq ← Q.pop()
5      if q is the first vertex of link(v) then continue
6      if the orientations of △vss' w.r.t. p and q are different and the orientations
       of △vpq w.r.t. s and s' are different then
7          if △vpq is counter-clockwise w.r.t. s then
8              twist pq
9              Q ← {△vrq, △vpt}, where △vrq and △vpt are triangles of star(v)
10     else
11         if △vpq is clockwise then
12             twist pq
13             Q ← {△vrq, △vpt}, where △vrq and △vpt are triangles of star(v)
```

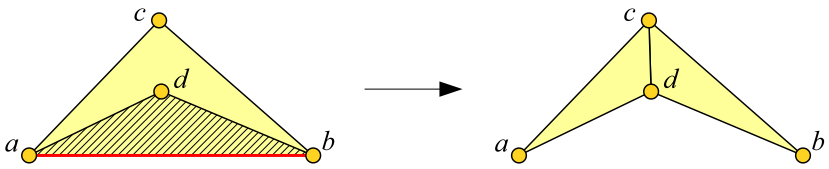
Particularly, Line 6 checks whether $\mathcal{CH}(\overrightarrow{vp}, \overrightarrow{vq})$ intersects with $\overrightarrow{ss'}$, the ray that starts at s and has the same direction as the positive x-axis. Note that a twist modifies not only $star(v)$ but also $\mathcal{P} \setminus star(v)$, and thus $\mathcal{P}$ remains a coherently oriented polyhedron after the twist.

After the twisting stage, for each edge $\overline{pq}$ of $\mathcal{P}_{2D}$, $\triangle spq$ is counter-clockwise. Therefore, all the triangles of $star(v)$ are counter-clockwise w.r.t. s and thus positive. Furthermore, $star(v)$ is star-shaped w.r.t. s because $\mathcal{P}_{2D}$ is star-shaped w.r.t. s. Therefore, we have the following lemma.

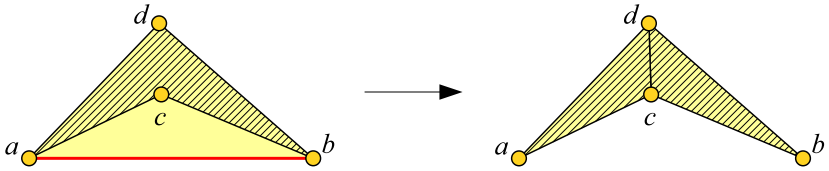
Lemma 8.3. *The twisting stage transforms $\mathcal{P}$ to a polyhedron where $star(v)$ is star-shaped w.r.t. s and all the triangles of $star(v)$ are positive, given v as the topmost vertex of $\mathcal{P}$ and s as a secondary kernel.*

The flipping stage

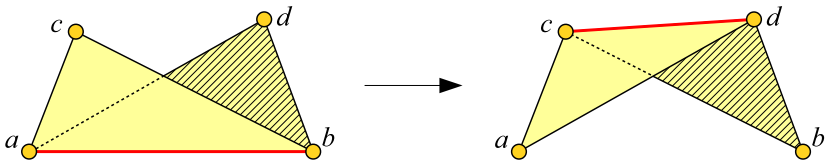
In this stage, we make all the triangles of $\mathcal{P} \setminus star(v)$ positive by flipping the folds of $\mathcal{P}$. A 3-1 flip is not allowed since it removes a vertex from $\mathcal{P}$; a 2-2 flip that creates an existing edge of $\mathcal{P}$ is also not allowed since it breaks the topology of $\mathcal{P}$. Therefore in this context, an edge of $\mathcal{P}$ is *flippable* if it is a 2-2 edge and the new edge created is not already in $\mathcal{P}$; otherwise it is *unflippable*. Right after the twisting stage, every edge of $star(v)$ cannot be a fold as all the triangles of $star(v)$ are already positive, and every edge of $link(v)$ is unflippable since all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v. Therefore, $star(v)$ and $link(v)$ never change during the flipping stage, and thus we only focus on



(a) The flip creates two positive triangles



(b) The flip creates two negative triangles



(c) The flip creates one positive triangle and one negative triangle

Figure 8.9: Three cases of flipping a fold $\overline{ab}$ of the projected triangulation $\mathcal{T}$ of $\mathcal{P}$. In each of these cases, we flip $\{\triangle abc, \triangle bad\}$ to $\{\triangle adc, \triangle bcd\}$. Negative triangles are shaded, and folds are represented by thick line segments in red.

the edges of $\mathcal{P} \setminus (star(v) \cup link(v))$.

To simplify our presentation, we define the *projected triangulation* $\mathcal{T}$ of $\mathcal{P}$ (w.r.t. v and s) in the following way. First, project each vertex p of $\mathcal{P} \setminus \{v\}$ to the intersection of $\overrightarrow{vp}$ and H; see Figure 8.8. This is always possible because v is the topmost vertex of $\mathcal{P}$. $\mathcal{T}$ is an abstract triangulation embedded on H whose vertices are projected from those of $\mathcal{P} \setminus \{v\}$ and whose combinatorial is copied from $\mathcal{P}$. More specifically, the (topological) boundary of $\mathcal{T}$ corresponds to $link(v)$, and each internal edge corresponds to an edge of $\mathcal{P} \setminus (star(v) \cup link(v))$. Each triangle of $\mathcal{P} \setminus star(v)$ and the corresponding one in $\mathcal{T}$ has the same orientation w.r.t. v; we thus say a triangle of $\mathcal{T}$ is positive (resp., negative) if it is counter-clockwise (resp., clockwise) w.r.t. v. In the subsequent discussion, flipping an edge of $\mathcal{T}$ also means flipping the corresponding edge of $\mathcal{P}$.

Consider a fold $\overline{ab}$ of $\mathcal{T}$, and let $\triangle abc$ and $\triangle bad$ be the positive and the negative triangles incident to it. Flipping $\overline{ab}$ removes $\{\triangle abc, \triangle bad\}$ from $\mathcal{T}$, and adds $\{\triangle adc, \triangle bcd\}$. Based on the relative positions of $\{a, b, c, d\}$, the flip is classified to one of the three cases as illustrated in Figure 8.9. In the first case (Figure 8.9(a)), $d \in \triangle abc$, and thus the two new triangles ($\triangle adc$ and $\triangle bcd$) are positive. In the second case (Figure 8.9(b)), $c \in \triangle bad$, and thus the two new triangles are negative. In the third case (Figure 8.9(c)), neither c nor d lies inside the other triangle, and thus the flip creates one positive and one negative triangles. The first case is good because the flip

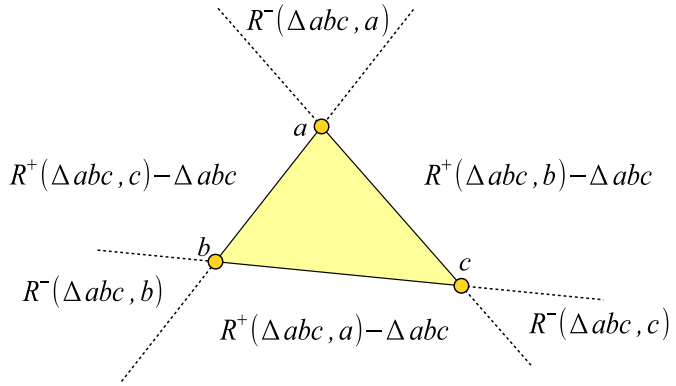


Figure 8.10: $\triangle abc$ has three containing regions and three opposite regions. The intersection of its three containing regions is $\triangle abc$.

removes one fold and reduces the number of negative triangles by one. On the other hand, it is hard to evaluate the latter two cases based on only the numbers of folds and negative triangles. Instead, we will use the angles in the projected triangulation to guide the flipping procedure.

Now consider a triangle $\triangle abc$ of $\mathcal{T}$. The pair of lines passing through $\overline{ab}$ and $\overline{ac}$ partition H into four regions. Among these regions, the one containing $\triangle abc$ is called the *containing region* of $\triangle abc$ at vertex a, and denoted as $R^+(\triangle abc, a)$; the one opposite to $R^+(\triangle abc, a)$ is called the *opposite region* of $\triangle abc$ at vertex a, and denoted as $R^-(\triangle abc, a)$. Using all three pairs of lines passing through the edges of $\triangle abc$, we get in total three containing regions and three opposite regions; see Figure 8.10. These containing regions overlap at $\triangle abc$. As proved later in Lemma 8.5, s can only lie in the opposite regions of negative triangles and in the containing regions of positive triangles. Let $\triangle abc$ be a negative triangle and s lie in $R^-(\triangle abc, a)$. The vertex a and the angle $\angle bac$ are respectively called the *primary vertex* and the *primary angle* of $\triangle abc$.

The key idea of the flipping stage is to monotonically increase the primary angles of the negative triangles. Particularly, we flip a fold if (1) the two created triangles are both positive (Figure 8.9(a)), or (2) exactly one of the two created triangles is negative (Figure 8.9(c)) and the new negative triangle has a larger primary angle than that of the negative triangle being removed. This condition can be further simplified into the following **criterion**: flip a fold $\overline{ab}$ of $\mathcal{T}$ if the primary vertex of its incident negative triangle is neither a nor b. The **local check** is to examine if an edge of $\mathcal{T}$ is a fold and if it contains the primary vertex of the negative triangle incident to it.

In the pseudocode of the flipping stage shown in Algorithm 13, we directly use the polyhedron $\mathcal{P}$ instead of its projected triangulation $\mathcal{T}$ in order to maintain the consistency with Algorithm 12. However, we note that flipping on $\mathcal{P}$ and flipping on $\mathcal{T}$ are equivalent. In Line 5 and Line 6 we test whether an edge $\overline{ab}$ is an internal edge and whether it is a fold respectively. In Line 8, we test if the primary vertex of the negative triangle incident to $\overline{ab}$ is not a vertex of $\overline{ab}$.

Algorithm 13: The flipping stage of the twist-flip algorithm

input : a polyhedron $\mathcal{P}$ with topmost vertex v and secondary kernel s;
all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v;
$star(v)$ is star-shaped w.r.t. s and only contains positive triangles
output: a star-shaped polyhedron $\mathcal{P}$ w.r.t. v

```
1  Q ← {ab | ab is an edge of P \ (star(v) ∪ link(v)) }
2  while Q ≠ ∅ do
3      ab ← Q.pop()
4      let △abc and △bad be the two incident triangles of ab
5      if v ∉ {a, b, c, d} then
6          if △abc and △bad have different orientations w.r.t. v then
7              assume △abc is positive w.l.o.g.
8              if △bas is clockwise w.r.t. v then
9                  flip ab
10                 Q ← {ac, bc, ad, bd}
```

Correctness of the algorithm

Because of Lemma 8.3, we can assume that $star(v)$ is star-shaped w.r.t. s and all the triangles of $star(v)$ are positive, and instead focus on the correctness of the flipping stage. Since all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v, all the vertices of $\mathcal{T}$ are on its boundary. We first prove that s lies in the opposite region of each negative triangle and the containing region of each positive triangle of $\mathcal{T}$.

Lemma 8.4. *An internal edge $\overline{pq}$ of $\mathcal{T}$ partitions the boundary of $\mathcal{T}$ into two chains of edges. Each chain has a vertex of degree 2, and it is not p or q.*

Proof. As all the vertices are on the boundary of $\mathcal{T}$, $\overline{pq}$ partitions $\mathcal{T}$ into two parts, and thus partitions the boundary of $\mathcal{T}$ into two chains. Any vertex on one chain cannot connect to any vertex in the other chain (other than p and q).

Without loss of generality, we focus on one of the two chains, L_{pq}. We prove the claim by induction on k, the number of vertices of L_{pq}, $k \geq 3$. When $k = 3$, the vertex of L_{pq} other than p and q has degree 2, because it can only connect to p and q. Assume the claim is true when $k = m$ for $m \geq 3$. When $k = m + 1$, we pick a vertex $r \in L_{pq} \setminus \{p, q\}$, and the proof is finished if its degree is 2. If the degree of r is larger than 2, it connects with a vertex u on L_{pq} other than its two neighbors. Consequently, r and u cut out a sub-chain of L_{pq}. This sub-chain contains less than $m + 1$ vertices, and thus has a vertex of degree 2. We finish the proof by induction. □

Lemma 8.5. *For a triangle $\triangle abc$ of $\mathcal{T}$, s lies in its opposite regions if and only if it is negative.*

Proof. Recall that $star(v)$ is star-shaped w.r.t. s after the twisting stage, and so is the boundary of $\mathcal{T}$. Let a', b' and c' in $\mathcal{P}$ be the corresponding vertices of a, b and c.

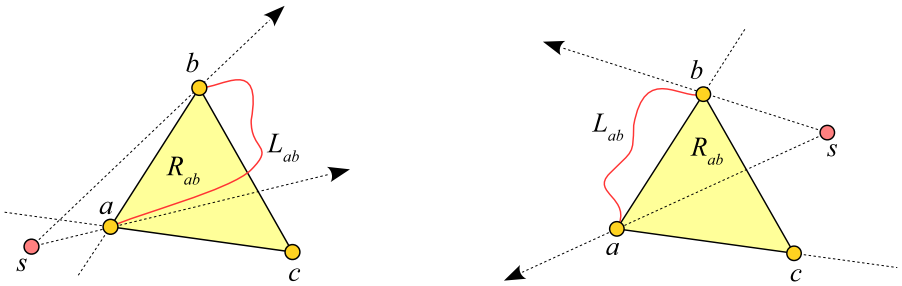


Figure 8.11: $\triangle abc$ is a triangle in the projected triangulation $\mathcal{T}$ of $\mathcal{P}$. R_{ab} is the region bounded by the rays $\overrightarrow{sa}$ and $\overrightarrow{sb}$, and L_{ab} is the chain of edges in the boundary of $\mathcal{T}$ that is completely contained in R_{ab}. (a) When s lies inside $R^-(\triangle abc, a)$, $\triangle abc$ is a negative triangle. (b) When s lies inside $R^+(\triangle abc, a)$, $\triangle abc$ is a positive triangle.

(only if) Without loss of generality, assume that s lies inside $R^-(\triangle abc, a)$. Let R_{ab} be the region bounded by the rays $\overrightarrow{sa}$ and $\overrightarrow{sb}$, and L_{ab} be the chain of edges in the boundary of $\mathcal{T}$ that has a and b as its endpoints and does not contain c; see Figure 8.11(a). L_{ab} is star-shaped w.r.t. s and completely contained in R_{ab}. By Lemma 8.4, L_{ab} has a vertex $r \notin \{a, b\}$ of degree 2, and the corresponding vertex of r in $\mathcal{P}$ has degree 3. We imaginarily apply a 3-1 flip on the corresponding vertex of r in $\mathcal{P}$. This flip removes r from L_{bc}. After the flip, L_{ab} is still star-shaped w.r.t. s, implying that $star(v)$ is still star-shaped w.r.t. s and all the triangles of $star(v)$ are still positive. We repeat this *ear cutting* procedure until $\overline{ab} \in link(v)$ and $\triangle va'b' \in star(v)$. Now $\triangle va'b'$ is counter-clockwise w.r.t. s, and hence clockwise w.r.t. c' as c and s lie on different sides of $\overline{ab}$. Since $\mathcal{P}$ is coherently oriented, $\triangle a'b'c'$ is clockwise w.r.t. v. By definition, $\triangle a'b'c'$ and $\triangle abc$ are negative.

(if) Assume s lies inside a containing region of $\triangle abc$, say $R^+(\triangle abc, a)$; see Figure 8.11(b). Again, we imaginarily apply ear cutting on L_{ab} until $\overline{ab} \in link(v)$ and $\triangle va'b' \in star(v)$. Then $\triangle va'b'$ is counter-clockwise w.r.t. s, and thus is counter-clockwise w.r.t. c'. Therefore $\triangle a'b'c'$ is counter-clockwise w.r.t. v. By definition, $\triangle a'b'c'$ and $\triangle abc$ are positive. □

Corollary 8.6. *For a triangle $\triangle abc$ of $\mathcal{T}$, s lies in its containing regions if and only if it is positive.*

The next lemma shows that we do not need to check the flippability of internal edges of $\mathcal{T}$.

Lemma 8.7. *Any internal edge of $\mathcal{T}$ is flippable.*

Proof. An internal edge $\overline{pq}$ partitions $\mathcal{T}$ into two regions, and thus partitions the boundary of $\mathcal{T}$ into two chains. As the two triangles incident to $\overline{pq}$ belong to different regions, the two link vertices of $\overline{pq}$ are on different chains. As pointed out in the proof

of Lemma 8.4, a vertex on one chain cannot connect to any vertex in the other chain (other than p and q). Therefore the edge connecting the two link vertices of $\overline{pq}$ cannot already exist in $\mathcal{T}$, and thus $\overline{pq}$ is flippable. □

Finally, we show that $\mathcal{P}$ is star-shaped when all its triangles are positive, and the twist-flip algorithm terminates at such a polyhedron.

Lemma 8.8. *$\mathcal{P}$ is star-shaped w.r.t. v if all its triangles are positive and $star(v)$ is star-shaped w.r.t. s.*

Proof. By definition, all the triangles of $star(v)$ are counter-clockwise w.r.t. s, and all the triangles of $\mathcal{P} - star(v)$ are counter-clockwise w.r.t. v.

Let v' be a point that lies in the interior of $\overline{vs}$ and is infinitely close to v. Hence all the triangles of $\mathcal{P} \setminus star(v)$ are counter-clockwise w.r.t. v'. Furthermore, since all the points in the interior of $\overline{vs}$ are above all the triangles of $star(v)$, $star(v)$ is star-shaped w.r.t. v' and all its triangles are counter-clockwise w.r.t. v'. Therefore, all the triangles of $\mathcal{P}$ have the same orientation w.r.t. v', which satisfies the first condition of Theorem 8.2.

Now shoot a ray from v' horizontally. Since v' has a larger z-coordinate than any vertex of $\mathcal{P} \setminus \{v\}$, the ray only intersects with the triangles of $star(v)$. In addition, as $star(v)$ is star-shaped w.r.t. v', the ray can intersect with exactly one triangle. This satisfies the second condition of Theorem 8.2. Therefore, we conclude that $\mathcal{P}$ is star-shaped w.r.t. v' and hence v. □

Theorem 8.9. *The twist-flip algorithm transforms $\mathcal{P}$ to a star-shaped polyhedron w.r.t. its topmost vertex v, under the precondition that all the vertices of $\mathcal{P} \setminus \{v\}$ connect to v in the input.*

Proof. After the twisting stage, $star(v)$ is star-shaped w.r.t. s and all its triangles are positive, i.e. counter-clockwise w.r.t. s (Lemma 8.3). These properties still exist during the flipping stage since we never flip the edges of $star(v) \cup link(v)$.

Termination. Denote the primary angle of a negative triangle $\triangle abc$ of $\mathcal{T}$ as $\theta(\triangle abc)$. We define a global score function of $\mathcal{T}$ as

$$\phi(\mathcal{T}) = \sum(\pi - \theta(t)) \quad \forall t \text{ is a negative triangle of } \mathcal{T}$$

We show that a flip always decreases $\phi(\mathcal{T})$. As illustrated in Figure 8.9, there are three cases of flipping a fold of $\mathcal{T}$. In the first case (Figure 8.9(a)), the vertex d is surely the primary vertex of $\triangle bad$; otherwise assuming a is the primary vertex, s lies inside $R^-(\triangle bad, a)$ and therefore inside $R^-(\triangle abc, a)$, which contradicts Lemma 8.5 since $\triangle abc$ is positive. Hence, we flip all the folds belonging to the first case according to the criterion. Flipping such a fold removes a negative triangle, and thus decreases $\phi(\mathcal{T})$. In the second case (Figure 8.9(b)), the vertex d cannot be the primary vertex of $\triangle bad$; otherwise s lies in $R^-(\triangle bad, d)$ and therefore $R^-(\triangle abc, c)$, which contradicts

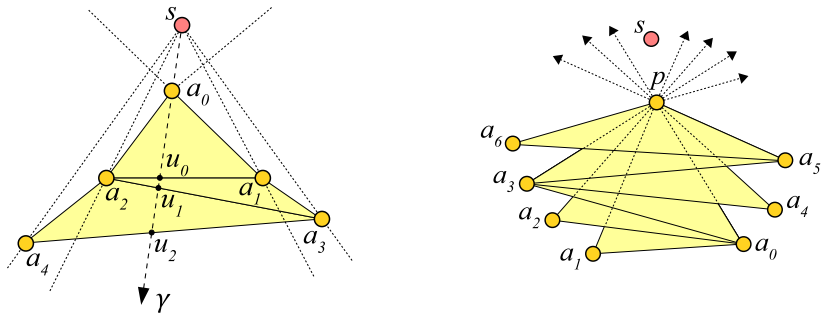


Figure 8.12: (a) When twist-flip terminates, $\mathcal{T}$ has no negative triangulations. Otherwise $\mathcal{T}$ has a flippable fold. (b) In the flipping stage, an initial negative triangle $\triangle pa_0a_1$ generates a series of negative triangles. The vertex p is the common primary vertex of these triangles, and the subscripts of a indicate the appearing order of the vertices in this series.

Lemma 8.5 again. Hence, we never flip the folds belonging to the second case. In the third case (Figure 8.9(c)), we only flip the fold $\overline{ab}$ if d is the primary vertex of $\triangle bad$. This flip removes a negative triangle $\triangle bad$ and creates another negative triangle $\triangle bcd$. Because c is outside $\triangle bad$ and lies on the same side of $\overline{ab}$ with d, $\theta(\triangle bcd) = \angle bdc > \angle adb = \theta(\triangle bad)$. Again, the flip decreases $\phi(\mathcal{T})$.

As $\phi(\mathcal{T})$ is a monotonically decreasing function and the number of vertices is finite, the flipping stage always terminates.

Correctness. By contradiction assume a negative triangle $\triangle a_0a_1a_2$ exists in $\mathcal{T}$ when the algorithm terminates, and its primary vertex is a_0. Let γ be the ray starting at s and passing through a_0, and u_0 be the intersected point of ray γ and $\overline{a_1a_2}$; see Figure 8.12(a).

Now consider $\overline{a_1a_2}$. Let a_i' be the corresponding vertex of a_i in $\mathcal{P}$. Suppose on the contrary that $\overline{a_1a_2}$ is on the boundary of $\mathcal{T}$. Since $\triangle a_0'a_1'a_2'$ is negative and thus clockwise w.r.t. v, $\triangle va_2'a_1'$ must be clockwise w.r.t. a_0' or otherwise $\mathcal{P}$ is not coherently oriented. As s and a_0 lie on the same side of $\overline{a_1a_2}$, $\triangle va_2'a_1'$ is also clockwise w.r.t. s, which contradicts the fact that all the triangles of $star(v)$ are positive.

Therefore $\overline{a_1a_2}$ is an internal edge of $\mathcal{T}$. Furthermore, it cannot be a fold, or otherwise it should be flipped by the criterion of the flipping stage. Hence, the other triangle incident to $\overline{a_1a_2}$, $\triangle a_1a_3a_2$, is also a negative triangle. To promise that s lies inside the opposite regions of $\triangle a_1a_3a_2$ (Lemma 8.5), a_3 must lie outside $\mathcal{CH}(\overrightarrow{sa_1}, \overrightarrow{sa_2})$. Without loss of generality, assume a_1 is the primary vertex of $\triangle a_1a_3a_2$, and thus $\mathcal{CH}(\overrightarrow{sa_2}, \overrightarrow{sa_3})$ contains $\mathcal{CH}(\overrightarrow{sa_1}, \overrightarrow{sa_2})$; see Figure 8.12(a). Then $\overline{a_2a_3}$ intersects with γ at a point u_1, $|su_1| > |su_0|$.

We continue the argument by replacing $\triangle a_0a_1a_2$ with $\triangle a_1a_3a_2$. Again, $\overline{a_2a_3}$ cannot be on the boundary of $\mathcal{T}$, and the next found triangle $\triangle a_2a_3a_4$ must be negative.

Without loss of generality, assume a_2 is the primary vertex of $\triangle a_2a_3a_4$, and thus $\mathcal{CH}(\overrightarrow{sa_3}, \overrightarrow{sa_4})$ contains $\mathcal{CH}(\overrightarrow{sa_2}, \overrightarrow{sa_3})$. Therefore, $\overline{a_3a_4}$ intersects with $\overrightarrow{sa_0}$ at a point u_2 with $|su_2| > |su_1|$. Repeat this argument to find more negative triangles. As $|su_i|$ is monotonically increasing and the number of the triangles is finite, this progress must terminate, and we encounter either an edge on the boundary of $\mathcal{T}$ or a fold. However, as discussed above for $\triangle a_0a_1a_2$, the existence of such an edge either contradicts that $\mathcal{P}$ is coherently oriented, or contradicts the termination of the flipping stage.

Therefore, all the triangles of $\mathcal{T}$ and their corresponding ones in $\mathcal{P}$ are positive. Together with that all the triangles of $star(v)$ are positive, we know that all the triangles of $\mathcal{P}$ are positive. By Lemma 8.8, $\mathcal{P}$ is star-shaped w.r.t. v. □

Time complexity

Let n be the number of the vertices of $\mathcal{P}$. As explained in the twist algorithm in Section 8.3, the twisting stage of the twist-flip algorithm has $\mathrm{O}(n^2)$ worst-case time complexity. Let k be the number of the negative triangles at the beginning of the flipping stage. Every flip either removes a negative triangle (see Figure 8.9(a)), or replaces a negative triangle with one that has a larger primary angle (see Figure 8.9(c)). In the latter case, the removed and the created triangles share one common edge and have the same primary vertex. Therefore, a negative triangle at the beginning generates a series of negative triangles with the same primary vertex, in which the next triangle is created by flipping a fold of the previous one and has a larger primary angle; see Figure 8.12(b). Along the series, each triangle differs from the next one by exactly one vertex, and that vertex cannot appear in any subsequent triangle because of the monotonicity of the primary angles. Thus, the number of negative triangles in each series is $\mathrm{O}(n)$. Therefore the worst-case time complexity of the flipping stage is $\mathrm{O}(kn)$, and the whole algorithm is $\mathrm{O}(n^2)$.

8.4.2 General Case

Lemma 8.5 and Lemma 8.7, which are the basis of the correctness of twist-flip, rely on the precondition that all the vertices of $\mathcal{P}\backslash\{v\}$ connect to v. When the input polyhedron does not satisfy this precondition, twist-flip does not work. A simple way to make twist-flip usable for the general case is to flip the edges of $link(v)$ until v connects to all the other vertices before applying it. However, this approach is cumbersome since it may apply many unnecessary flips in order to connecting v and other vertices.

In this subsection, we show our preliminary study for the general case of transforming an arbitrary polyhedron to a star-shaped one. Assume that we have applied the twisting stage on $\mathcal{P}$, and therefore $star(v)$ is star-shaped w.r.t. s and contains only positive triangles. We introduce the concepts of extended projected triangulation and local winding number, and propose a collection of sufficient conditions to

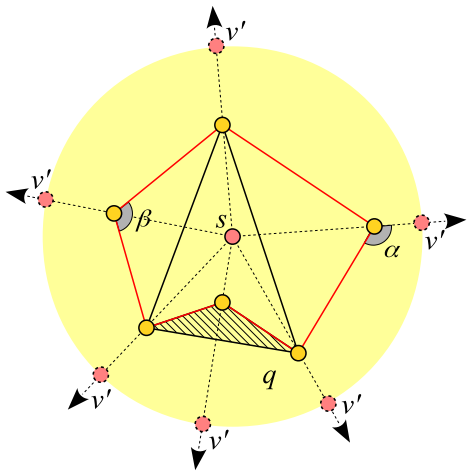


Figure 8.13: The extended projected triangulation $\mathcal{T}'$ of the polytope $\mathcal{P}$ with s being the secondary kernel. The virtual vertex of $\mathcal{T}'$ is v', and $link(v')$ is a star-shaped polygon (in red) w.r.t. s. The only negative triangle of $\mathcal{T}'$ is shaded.

describe star-shaped polyhedron using only local properties (Theorem 8.12). These conditions possibly help to design local transformation to star-shaped polyhedron. In order to prove these, we build some important formulas related to local winding number (Lemma 8.10 and Lemma 8.11).

Recall that the projected triangulation $\mathcal{T}$ of $\mathcal{P}$ is obtained by projecting $\mathcal{P} \setminus star(v)$ on the horizontal hyperplane H that passes through s. The (topological) boundary of $\mathcal{T}$ is a star-shaped polygon w.r.t. s after the twisting stage. Since $\mathcal{T}$ is star-shaped w.r.t. s, we build the extended triangulation $\mathcal{T}'$ using $\mathcal{T}$ and s, and call it *extended projected triangulation* of $\mathcal{P}$; see Figure 8.13. The virtual vertex v' of $\mathcal{T}'$ can be seen as the projection of v, and every triangle of $star(v')$ is positive so that it is coherently oriented with the triangles of $\mathcal{T}$.

Given a vertex p of $\mathcal{T}' \setminus \{v'\}$ and an edge $\overline{ab}$ of $link(p)$, the orientation of $\overline{ab}$ can be derived from that of $\triangle pab$. Therefore $link(p)$ forms a coherently oriented polygon; see Figure 8.14. The *local winding number* of p, denoted as $\lambda(p)$, is the winding number of $link(p)$ w.r.t. p. For example, in Figure 8.14(a), p is a vertex not connecting to v', and its local winding number is 1. In Figure 8.14(b), p connects to v' and its local winding number is 0. A vertex of $\mathcal{T}' \setminus \{v'\}$ is a *positive* (resp., *negative*) vertex if its local winding number is positive (resp., negative). A vertex with local winding number 0 is further called a *0-vertex*.

Let n be the number of the vertices of $\mathcal{T}' \setminus \{v'\}$, and k be the degree of v' ($k \leq n$). The boundary of $\mathcal{T}$ has k vertices. Define $S^+ = \{p \mid p \text{ is a positive vertex of } \mathcal{T}' \setminus \{v'\}\}$ and $S^- = \{p \mid p \text{ is a negative vertex or 0-vertex of } \mathcal{T}' \setminus \{v'\}\}$; $|S^+| + |S^-| = n$. Let t_p and t_n be the numbers of the positive and negative triangles in $\mathcal{T}' \setminus star(v')$. By Euler's Formula we can obtain $t_p + t_n + k = 2(n+1) - 4$ and thus $t_p + t_n = 2n - 2 - k$. In the following two technical lemmas, we build relations between these parameters and the

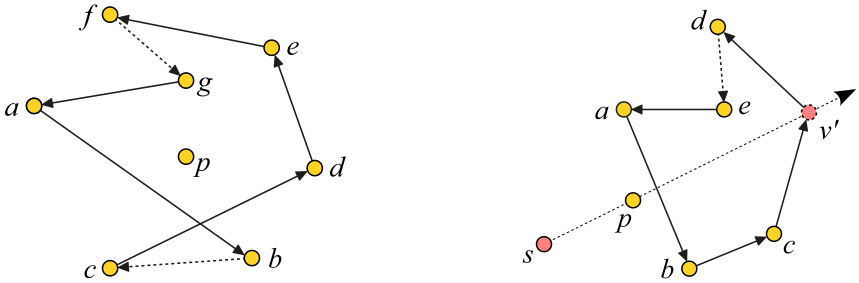


Figure 8.14: In the extended projected triangulation $\mathcal{T}'$ with virtual point v', the link of a vertex p forms a coherently oriented polygon. (a) p does not connect to v'. (b) p connects to v'.

local winding numbers of the vertices of $\mathcal{T}' \setminus \{v'\}$.

Lemma 8.10. $\sum_{p \in S^+ \cup S^-} \lambda(p) = \sum_{p \in S^+} \lambda(p) + \sum_{q \in S^-} \lambda(q) = n - t_n$.

Proof. Given a triangle $\triangle abc$, define its *signed angle* at one of its vertices as the inner angle at the vertex with the sign decided by the orientation of $\triangle abc$: the sign is positive if $\triangle abc$ is positive; otherwise it is negative. The angles incident to the virtual vertex v' are considered to be 0.

Consider the sum of all signed angles in $\mathcal{T}'$ by summing over all triangles. The sum over all real triangles is clearly $t_p\pi - t_n\pi$. On the other hand, since the virtual triangles are all positive, the signed angles that are not incident to v' in the virtual triangles, e.g. the angle α in Figure 8.13, are positive. Hence, the sum of these signed angles is $2k\pi - (k-2)\pi = (k+2)\pi$, where $(k-2)\pi$ is the sum of all the inner angles of the polygon $link(v')$. Therefore the sum of all signed angles is $t_p\pi - t_n\pi + (k+2)\pi = (t_p - t_n + k + 2)\pi = 2(n - t_n)\pi$.

Now consider the sum of all signed angles in $\mathcal{T}'$ by summing over all stars. An angle is incident to one and only one vertex, and the sum of all signed angles incident to a vertex p is $\lambda(p) \times 2\pi$. Therefore the sum of all signed angles is $\sum_{p \in S^+ \cup S^-} 2\lambda(p)\pi$. Equating it with the result of the previous paragraph, we prove the lemma. □

Lemma 8.11.

(1) $\sum_{p \in S^+} \lambda(p) - \sum_{q \in S^-} \lambda(q) = n$ *when* $t_n = 0$;

(2) $\sum_{p \in S^+} \lambda(p) - \sum_{q \in S^-} \lambda(q) < n$ *when* $t_n > 0$.

Proof. (1) Given a vertex $p \in \mathcal{T}' \setminus \{v'\}$, shoot an arbitrary ray from p. In all the triangles of $star(p)$ that intersect with this ray (if any), the number of positive triangles minus the number of negative triangles is equal to $\lambda(p)$. Thus, a 0-vertex or negative vertex

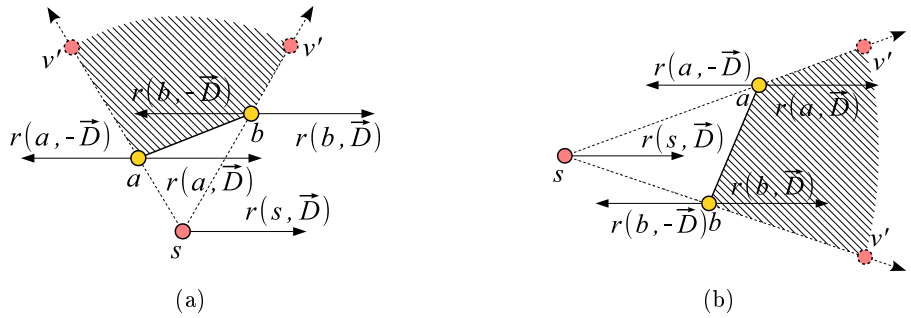


Figure 8.15: $\overrightarrow{D}$ is the positive direction of x-axis, and $\overline{ab}$ is an edge of $link(v')$. (a) $r(s,\overrightarrow{D}) \notin \mathcal{CH}(\overrightarrow{sa},\overrightarrow{sb})$, and $\triangle v'ab \in T^+(b,-\overrightarrow{D})$. (b) $r(s,\overrightarrow{D}) \in \mathcal{CH}(\overrightarrow{sa},\overrightarrow{sb})$, and $\triangle v'ab \in T^+(a,\overrightarrow{D}) \cap T^+(b,\overrightarrow{D})$.

must be incident to some negative triangles, and hence $S^- = \emptyset$ when $t_n = 0$. Also, by Lemma 8.10, $\sum_{p\in S^+} \lambda(p) = n - t_n = n$. Therefore, $\sum_{p\in S^+} \lambda(p) - \sum_{q\in S^-} \lambda(q) = n$.

(2) Choose a fixed direction $\overrightarrow{D}$. Let $r(x,\overrightarrow{D})$ be a ray with direction $\overrightarrow{D}$ from point x. For each vertex p of $\mathcal{T}' \setminus \{v'\}$ shoot two rays $r(p,\overrightarrow{D})$ and $r(p,-\overrightarrow{D})$, where $-\overrightarrow{D}$ is the opposite direction of $\overrightarrow{D}$. Perturb the chosen direction if necessary so that the two rays of any vertex do not pass through another vertex. Let $T^+(p,\overrightarrow{D})$ and $T^+(p,-\overrightarrow{D})$ be the set of the positive triangles of $star(p)$ intersecting with $r(p,\overrightarrow{D})$ and $r(p,-\overrightarrow{D})$ respectively, and $T^-(p,\overrightarrow{D})$ and $T^-(p,-\overrightarrow{D})$ be the set of the negative triangles of $star(p)$ intersecting with $r(p,\overrightarrow{D})$ and $r(p,-\overrightarrow{D})$ respectively.

For a 0-vertex or a negative vertex q, $-\lambda(q) = |T^-(q,\overrightarrow{D})| - |T^+(q,\overrightarrow{D})| = |T^-(q,-\overrightarrow{D})| - |T^+(q,-\overrightarrow{D})|$, indicating $-2\lambda(q) \le |T^-(q,\overrightarrow{D})| + |T^-(q,-\overrightarrow{D})|$. Besides, each negative triangle $\triangle abc$ belongs to exactly one of the set $T^-(x,\mathcal{Y})$ with $x \in \{a,b,c\}$ and $\mathcal{Y} \in \{\overrightarrow{D},-\overrightarrow{D}\}$, and thus all the sets $T^-(\cdot,\cdot)$ are non-overlapping. Consequently,
$$-2\sum_{q\in S^-} \lambda(q) \le \sum_{q\in S^-} (|T^-(q,\overrightarrow{D})| + |T^-(q,-\overrightarrow{D})|) \le t_n. \quad \text{(i)}$$

For a positive vertex p, $\lambda(p) = |T^+(p,\overrightarrow{D})| - |T^-(p,\overrightarrow{D})| = |T^+(p,-\overrightarrow{D})| - |T^-(p,-\overrightarrow{D})|$, indicating $2\lambda(p) \le |T^+(p,\overrightarrow{D})| + |T^+(p,-\overrightarrow{D})|$. Now consider all the sets $T^+(\cdot,\cdot)$. Each positive triangle that is real also appears only once. On the other hand, each virtual triangle $\triangle v'ab$ appears twice if $r(s,\mathcal{Y}) \in \mathcal{CH}(\overrightarrow{sa},\overrightarrow{sb})$ for any direction $\mathcal{Y} \in \{\overrightarrow{D},-\overrightarrow{D}\}$, and appears once otherwise. This is illustrated in Figure 8.15. Nevertheless, since $link(v')$ is star-shaped w.r.t. s, only two of the virtual triangles are counted twice. Hence,
$$2\sum_{p\in S^+} \lambda(p) \le \sum_{p\in S^+} (|T^+(p,\overrightarrow{D})| + |T^+(p,-\overrightarrow{D})|) \le t_p + k + 2. \quad \text{(ii)}$$

From (i) and (ii), $\sum_{p\in S^+} \lambda(q) - \sum_{q\in S^-} \lambda(q) \le (t_n + t_p + k + 2)/2 = n$. When $t_n > 0$, $\mathcal{T}'$ contains negative triangles, and thus has some folds. Let $\overline{ab}$ be a fold of $\mathcal{T}'$ incident to a positive triangle $\triangle abc$ (c can be v') and a negative triangle $\triangle bad$. Choose the direction $\overrightarrow{D}$ so that $r(a,\overrightarrow{D})$ intersects with both triangles. Therefore $|T^-(a,\overrightarrow{D})| > 0$ and $|T^+(a,\overrightarrow{D})| > 0$. Bringing this result back to the previous paragraph, we know that

if a is a positive vertex, $2\lambda(a) < |T^+(a, \overrightarrow{D})| + |T^+(a, -\overrightarrow{D})|$ and hence $2 \sum_{p \in S^+} \lambda(p) < t_p + k + 2$. Similarly if a is a 0-vertex or a negative vertex, $-2\lambda(a) < |T^-(a, \overrightarrow{D})| + |T^-(a, -\overrightarrow{D})|$, and thus $-2 \sum_{q \in S^-} \lambda(q) < t_n$. In either case, $\sum_{p \in S^+} \lambda(p) - \sum_{q \in S^-} \lambda(q) < n$. □

Given a vertex p of $\mathcal{T}' \setminus \{v'\}$, we say p is a *sink* if it lies outside the convex hull of its link; see Figure 8.14(b) as an example. The following lemma provides a collection of sufficient conditions for star-shaped polyhedron.

Theorem 8.12. *Given a polyhedron $\mathcal{P}$ with its topmost vertex v and secondary kernel s, let $\mathcal{T}'$ be the extended projected triangulation of $\mathcal{P}$. If $star(v)$ is star-shaped w.r.t. s and contains only positive triangles, then the following statements are equivalent:*

(1) $\mathcal{P}$ is star-shaped w.r.t. v;
(2) All the triangles of $\mathcal{T}'$ are positive;
(3) All the edges of $\mathcal{T}'$ are not folds;
(4) All the vertices of $\mathcal{T}' \setminus \{v'\}$ are positive;
(5) All the vertices of $\mathcal{T}' \setminus \{v'\}$ are not 0-vertices;
(6) All the vertices of $\mathcal{T}' \setminus \{v'\}$ are not sink.

Proof. It is trivial to see that (1) implies all other statements. We prove that each of the other statements implies (1).

For (2), when all the triangles of $\mathcal{T}'$ are positive, all their corresponding triangles in $\mathcal{P}$ are positive. In addition, $star(v)$ is star-shaped w.r.t. s and all its triangles are also positive. Thus $\mathcal{T}$ is star-shaped w.r.t. v by Lemma 8.8.

For (3), the virtual triangles of $\mathcal{T}'$ are positive, thus when $\mathcal{T}'$ has no folds, all its triangles are positive. By (2), $\mathcal{P}$ is star-shaped w.r.t. v.

For (4), when all the vertices of $\mathcal{T}' \setminus \{v'\}$ are positive, $\sum_{q \in S^-} \lambda(q) = 0$ and $\sum_{p \in S^+} \lambda(p) = n - t_n \leq n$ (Lemma 8.10). Since the local winding number of each vertex is at least 1, $\sum_{p \in S^+} \lambda(p) \geq n$. Thus, $\sum_{p \in S^+} \lambda(p) = n$ and $t_n = 0$. By (2), $\mathcal{P}$ is star-shaped w.r.t. v.

For (5), as $\mathcal{T}'$ has no 0-vertices, the local winding number of each vertex in S^- is at most -1, and thus $-\sum_{q \in S^-} \lambda(q) \geq |S^-|$. Similarly, $\sum_{p \in S^+} \lambda(p) \geq |S^+|$. $\sum_{p \in S^+} \lambda(p) - \sum_{q \in S^-} \lambda(q) \geq |S^+| + |S^-| = n$. By Lemma 8.11, $t_n = 0$. By (2), $\mathcal{P}$ is star-shaped w.r.t. v.

For (6), the sum of all the (absolute) angles in $\mathcal{T}'$ is $t_p\pi + t_n\pi + (k+2)\pi = 2n\pi$; see the proof of Lemma 8.10 for more details. Given a vertex p of $\mathcal{T}' \setminus \{v'\}$, let $\theta(p)$ be the sum of all the angles incident to p. Note that $\theta(p) \geq 2\pi$ if p is not a 0-vertex. Since an angle is incident to exactly one vertex, we conclude $\sum_{p \in \mathcal{T}' \setminus \{v'\}} \theta(p) = 2n\pi$.

Assume by contradiction that $\mathcal{T}'$ has no sinks yet $\mathcal{P}$ is not star-shaped w.r.t. v. By (5), $\mathcal{T}'$ has some 0-vertices. For each 0-vertex q, it is not a sink by assumption. As

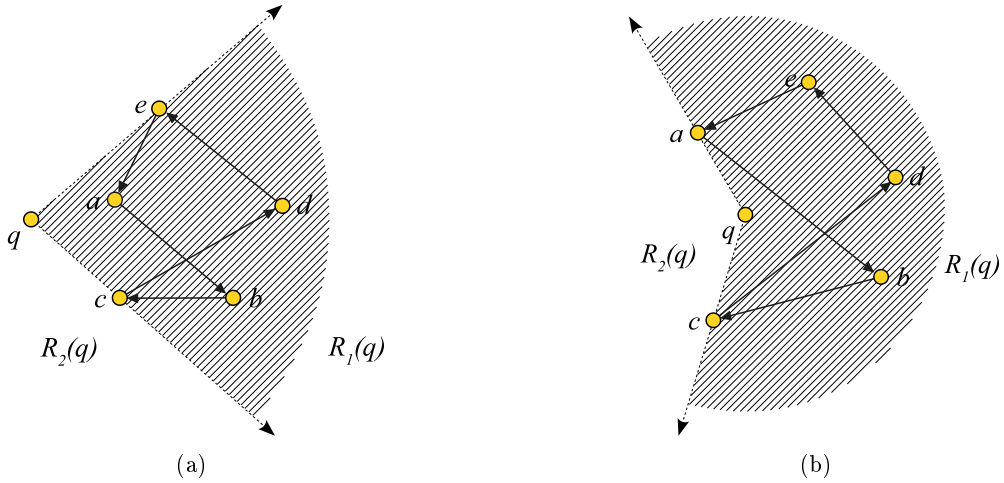


Figure 8.16: For a 0-vertex q, the rays of q intersecting the triangles of $star(q)$ form a pie (shaded regions). (a) When q is a sink, the angle of the pie is smaller than π, and (b) when q is not a sink, the angle is larger than π.

shown in the next paragraph, $\theta(q) > 2\pi$. In addition, $\theta(p) \geq 2\pi$ for each vertex p that is not 0-vertex ($p \neq v'$). Therefore, $\sum_{p \in \mathcal{T}' \setminus \{v'\}} \theta(p) > 2n\pi$, which is a contradiction.

We prove that for any 0-vertex q, $\theta(q) > 2\pi$ if q is not a sink. Partition the fan around q into two sectors: sector $R_1(q)$ that is covered by the triangles in $star(q)$, and sector $R_2(q)$ that is not covered (possibly empty); see Figure 8.16. Note that since q is a 0-vertex, $R_1(q)$ is covered at least twice, i.e., every ray starting at q in the interior of $R_1(q)$ intersects with at least two triangles of $star(q)$. Use $|R_i(q)|$ to represent the angle of the sector; $|R_i(q)| \neq \pi$ by the assumption of general position and $\theta(q) \geq 2|R_1(q)|$. Clearly if q is not a sink, $|R_1(q)| > \pi$ and thus $\theta(q) \geq 2R_1(q) > 2\pi$. □

Each of the statements from (2) to (6) in Theorem 8.12 provides a sufficient condition for $\mathcal{P}$ to be star-shaped. More importantly, these conditions are verifiable using local checks and thus they imply some possible monotonic procedure for the local transformation to star-shaped polyhedron. To sum up this section and end this chapter, we go through each of these statements and discuss their implication.

Statement (2) is about the negative triangles of $\mathcal{T}'$; it suggests a monotonic procedure of reducing the number of the negative triangles. Similarly, Statement (3) implies a procedure of decreasing the number of the folds in $\mathcal{T}'$. However, a flip like the one in Figure 8.9(c) does not reduce the number of the negative triangles or the folds. Statement (4) and Statement (5) focus on the local winding numbers of vertices of $\mathcal{T}'$. The local winding number of a vertex may be changed via flipping a fold. The flip in Figure 8.9(a), for example, increases $\lambda(d)$ by 1, while the one in Figure 8.9(b) decreases $\lambda(c)$ by 1. One may attempt to increase the local winding number of some vertices by flips until all the vertices are positive. However, a flip like the one in Figure 8.9(c) does

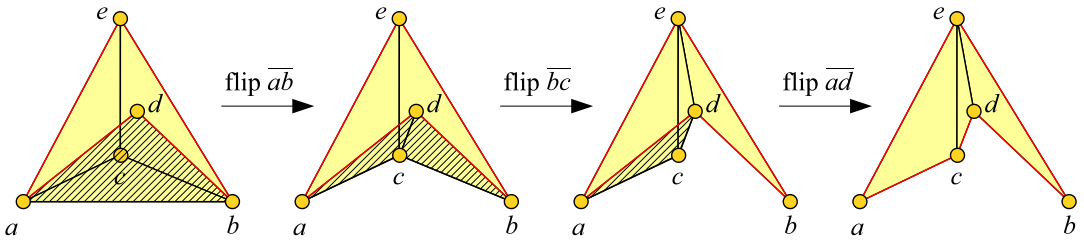


Figure 8.17: $\mathcal{T}$, the projected triangulation of $\mathcal{P}$, is transformed by flips into one that contains no negative triangles. The (topological) boundary of $\mathcal{T}$ are colored in red, and its negative triangles are shaded. Assume that the secondary kernel s is located so that the boundary of $\mathcal{T}$ is star-shaped w.r.t. s during the process.

not change any local winding number.

Figure 8.17 shows a flipping procedure that transforms a projected triangulation containing negative triangles to one that has no negative triangles. The first flip in the procedure increases the numbers of the negative triangles and the folds, and decreases $\lambda(c)$ from 1 to 0. Hence this procedure does not have the monotonicity suggested by Statement (2)–(5).

Statement (6) uses sink to prove star-shaped polyhedron: when $\mathcal{P}$ is not star-shaped, $\mathcal{T}'$ must have a sink. In order to remove a sink p, we should insert more vertices into $star(p)$. This can be done by flipping the edges of $link(p)$, which is actually a reversed action of flipping the edges of $star(p)$ to decrease the degree of p in flip-flop. However, an edge of $link(p)$ is also an edge of the star of another vertex, say q. Flipping the edge removes a vertex from $star(q)$, possibly making q a new sink. Figure 8.18 shows an example for this problem.

To sum up, it is challenging to design an algorithm of local transformation from an arbitrary polyhedron to a star-shaped one. A key step is to find a measure for the monotonicity of the local transformation. As discussed above, such measure can be the local winding number, the sum of degree, the number of sinks, or something else. As a basic rule, a proper measure should effectively reflect the monotonicity of a single flip.

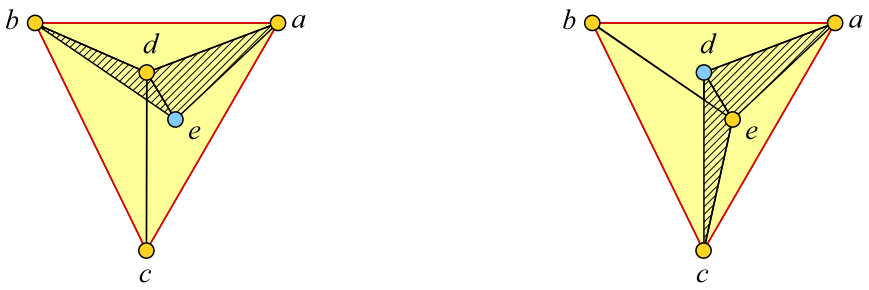


Figure 8.18: The projected triangulation $\mathcal{T}$ of $\mathcal{P}$, whose (topological) boundary is in red and negative triangles are shaded. (a) $\mathcal{T}$ has one sink e. (b) After flipping $\overline{bd}$, e becomes a 0-vertex that is not sink, but d becomes a sink.

CHAPTER 9

Flipping in Higher Dimensions

This chapter studies different ways to locally transform one regular triangulation to another of the same point set in any dimension.

Section 9.1 presents the current state-of-the-art algorithms on transforming from one triangulation to another in high dimensions. These algorithms work for only a few limited class of inputs. There is a counter-example to show that two arbitrary triangulations of a point set are not always reachable from each other through flips, whereas it is always possible if both triangulations are regular. We thus pose the question of designing a local transformation, preferably with free execution order, to reach any regular triangulation from another.

Section 9.2 proposes a moving model for transforming between regular triangulations and presents a sequential algorithm based on the idea of kinetic data structure. Specifically, the algorithm repeatedly updates the regular triangulation when the weights of the vertices continuously change to a given target. This approach is derived straightforwardly from those of [Raj91, Gui98, She03]. This section also discusses the handling of degeneracy, and presents a novel algorithm for finding a perturbation order to realize a strongly regular triangulation.

Section 9.3 further attempts to find a flip algorithm to solve the problem with free execution order. With the geometric structure cast into a time line, we study the properties of ridges, and show how these are affected by flips. Our findings present a new perspective to understand the sequential solution of Section 9.2, and more importantly shed light on the possibility of finding good strategies for flipping between regular triangulations with free execution order.

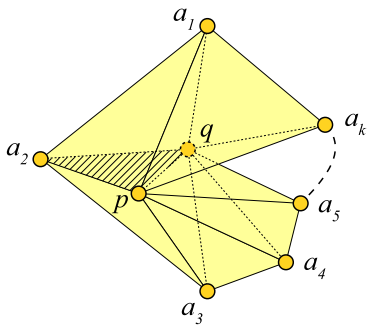


Figure 9.1: Triangle $\triangle pqa_2$ is locally non-Delaunay and unflippable because of reflex edge $\overline{pq}$. Then there must be another locally non-Delaunay triangle in $star(\overline{pq})$.

9.1 Literature Review

A flip algorithm is *monotonic* if it flips only locally non-regular ridges. For example, Lawson's flip algorithm is a monotonic algorithm, while flip-flop is not. The *arbitrarily monotonic flip algorithm* particularly refers to the algorithm that arbitrarily picks a locally non-regular and flippable ridge to flip. Lawson's flip algorithm is the arbitrarily monotonic flip algorithm that computes 2D Delaunay triangulation.

Given a triangulation $\mathcal{T}$ in $\mathbb{R}^d$, let F be a ridge of $\mathcal{T}$ and $\{a, b\}$ be its link vertices. Recall that F is flippable if the induced-subcomplex of F is convex (Section 2.3). Here we introduce another way to check the flippability of F. A facet R of F, which is a $(d-2)$-face of $\mathcal{T}$, is said to be *convex* in F if there exists a hyperplane through R so that F, a, and b lie on the same side of the hyperplane; otherwise it is *reflex* in F. F is flippable if every reflex facet in it has exactly three link vertices [ES92]. In Figure 9.1, $\overline{pq}$ is reflex in $\triangle pqa_2$. As the degree of $\overline{pq}$ is more than 3, $\triangle pqa_2$ is unflippable.

Arbitrarily Monotonic Flip Algorithm Fails in $\mathbb{R}^3$

Joe [Joe89] gives a counter-example to prove that the arbitrarily monotonic flip algorithm may get stuck when computing Delaunay triangulation from an arbitrary 3D triangulation. In his example, all the locally non-Delaunay triangles are unflippable, and thus any monotonic flip algorithm cannot continue.

Joe further explains why the arbitrarily monotonic flip algorithm gets stuck. According to Lemma 9.1, the removal of a locally non-Delaunay and unflippable triangle depends on the removal of another locally non-Delaunay triangle. If that other triangle is also unflippable, its removal depends on the removal of a third one. When these dependencies form a cycle, none of the triangles in this cycle is flippable, and hence the arbitrarily monotonic flip algorithm gets stuck.

Lemma 9.1 ([Joe89]). *Let $\mathcal{T}$ be a triangulation containing tetrahedrons pqa_1a_2 and pqa_2a_3 such that $\triangle pqa_2$ is locally non-Delaunay and unflippable; see Figure 9.1. With-*

out loss of generality, assume $\overline{pq}$ is a reflex ridge in $\triangle pqa_2$. Then there exists another interior face $\triangle pqa_i$, $i \neq 2$, that is locally non-Delaunay.

Lemma 9.1 can be extended on three aspects to arrive at Lemma 9.2. First, we extend Delaunay triangulation to regular triangulation. Second, the conclusion of Lemma 9.1 can be specified into two cases. Third, we claim the lemma in any dimension, as Joe does in [Joe93].

Lemma 9.2. *Let $\mathcal{T}$ be a triangulation in $\mathbb{R}^d$ containing two facets C_1 and C_2 such that the ridge F incident to C_1 and C_2 is locally non-regular and unflippable, and $a_1 \in C_1$ and $a_3 \in C_2$ be the link vertices of F. Without loss of generality, assume that the facet R of F is reflex in F and has more than 3 link vertices, and the link vertex of R in F is a_2. Let $\langle a_1, a_2, a_3, ..., a_k \rangle$ be the ordered link vertices of R for $k \geq 4$. Then one of the following two must be true:*

(1) A ridge containing a_i $(4 \leq i \leq k)$ in $star(R)$ is locally non-regular.

(2) The ridges containing a_1 and a_3 in $star(R)$ are both locally non-regular.

Proof. See Figure 9.1 for the 3D illustration where $R = \overline{pq}$. We lift $\mathcal{T}$ by $v \rightarrow (v, v^2 - w)$, where w is the weight of the vertex v, and prove the lemma in the lifted space. In the following, p, q, a_i and R are also used to represent their counterparts in the lifted space. Let F_i be the ridge consisting R and a_i, and C_i be the facet consisting of R, a_i and a_{i+1}, assuming $a_{k+1} = a_1$ and $F_{k+1} = F_1$. Use H_i to denote the hyperplane passing through C_i.

As $F = F_2$ is locally non-regular, a_1 lies below H_2 in the half space bounded by the vertical plane of F_2 and containing a_1. Now by contradiction assume F_3 is locally regular, meaning that H_2 is below H_3 in the half space bounded by the vertical plane of F_3 and containing a_4. This half space also contains a_1 since R is reflex in F_2, and therefore a_1 is below H_3. Repeat the analogues argument until F_k: either some F_i for $3 \leq i \leq k-1$ is locally non-regular, or a_1 is below H_{k-1}. The latter indicates that F_k is locally non-regular. Therefore, at least one of $\{F_i \mid 3 \leq i \leq k\}$ is locally non-regular. Symmetrically, at lease one of $\{F_i \mid 4 \leq i \leq k+1\}$ is locally non-regular. Summarizing these results, we finish the proof. □

Arbitrarily Monotonic Flip Algorithm Works for "RT+1" Triangulation

Given a point set S in $\mathbb{R}^d$ and its regular triangulation $\mathcal{RT}(S)$, we can insert a point p inside $\mathcal{CH}(S)$ into $\mathcal{RT}(S)$ by simply applying a 1-$(d+1)$ flip. We call the resulting triangulation an "RT+1" triangulation, and similarly we can define "DT+1" triangulation. As shown later, these classes of triangulations are well-known for being workable inputs of the arbitrarily monotonic flip algorithm in any dimension.

Joe proves that the arbitrarily monotonic flip algorithm works for "DT+1" triangulation, first in $\mathbb{R}^3$ [Joe91] and then in any dimensional space [Joe93]. Edelsbrunner and

Shah [ES92] extend these results to regular triangulation. Specifically, they prove that there is always a flippable ridge among those locally non-regular ones during flipping, which can be found as follows. Lift the triangulation by the function $p \rightarrow (p, p^2 - w_p)$, where w_p is the weight of p. Each facet of the lifted triangulation defines a hyperplane, and the facet whose hyperplane has the highest intersection point with the vertical line through v must contain a ridge of $link(v)$. This ridge is locally non-regular and flippable unless the triangulation is regular.

We note without further explanation that the arbitrarily monotonic flip algorithm also works for a "RT+1" extended triangulation.

Arbitrarily Monotonic Flip Algorithm Fails For "RT+2" Triangulation

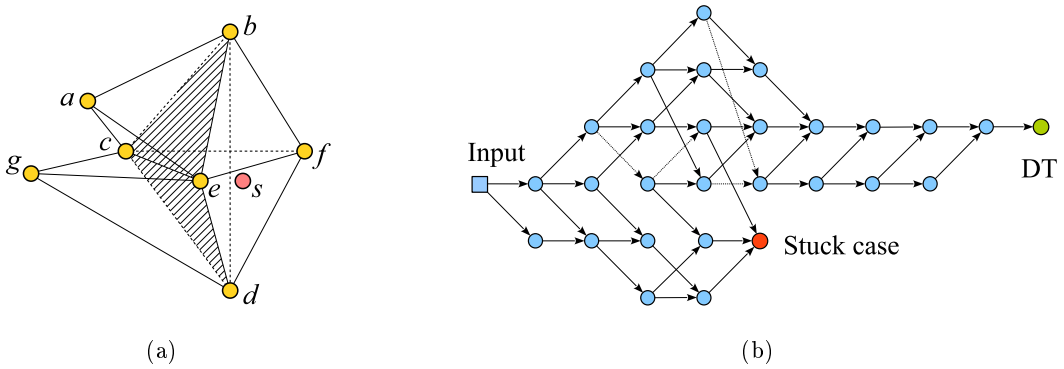
Analogous to "RT+1" triangulation, we define "RT+2" triangulation as a triangulation created by inserting two points into a regular triangulation. We design a counter-example to show that the arbitrarily monotonic flip algorithm does not work for "RT+2" triangulation. This result implies that the workable inputs for the arbitrarily monotonic flip algorithm are very restricted.

We build an extended triangulation $\mathcal{T}$ using seven points $\{a, b, c, d, e, f, g\}$ as its vertices and a point s as its kernel point, with the weights of all these points being 0. Let v be the virtual vertex of $\mathcal{T}$. Table 9.1 lists the coordinates of these points, and Figure 9.2(a) shows the combinatorial of $\mathcal{T}$. Note that the figure does not reflect the real positions of the vertices. The locally non-regular ridges of $\mathcal{T}$ are $\{\triangle bce, \triangle cde, \triangle vbc, \triangle vce\}$, among which $\triangle vbc$ and $\triangle vce$ are flippable. $\mathcal{T}$ can be constructed by inserting a and g into the extended regular triangulation of $\{b, c, d, e, f, v\}$, and thus is an "RT+2" extended triangulation.

Figure 9.2(b) shows the flip graph created by exploring all possible monotonic flip paths. Here a node represents a triangulation and an arrow indicates a flip from one triangulation to another. The flip paths converge to two different terminals. The first terminal, represented by the green node, is the regular triangulation, while the second terminal, represented by the red node, is a triangulation with all the locally non-regular triangles being unflippable. Beginning at $\mathcal{T}$, the arbitrarily monotonic flip algorithm may unfortunately get stuck at the red node.

Table 9.1: The coordinates of the vertices in the "RT+2" triangulation where the arbitrarily monotonic flip algorithm gets stuck.

	a	b	c	d	e	f	g	s
x	0.666	0.849	0.298	0.589	0.848	0.648	0.397	0.597
y	0.795	0.780	0.546	0.327	0.635	0.653	0.654	0.530
z	0.546	0.419	0.067	0.463	0.556	0.139	0.253	0.264



(a) (b)

Figure 9.2: (a) An extended triangulation built by inserting a and g into the extended regular triangulation of $\{b, c, d, e, f\}$ with s as the kernel point. (b) The flip graph of the arbitrarily monotonic flip algorithm on (a). The blue square represents the input triangulation, while the green node represents the regular triangulation.

By replacing the virtual vertex with several real vertices far away from s, we can also build a "RT+2" triangulation as counter-example.

Instant Flip Algorithm for Kinetic Data

For "DT+1" triangulation in d dimension ($d \geq 3$), Rajan [Raj91] designs a monotonic flip algorithm that does not allow arbitrary selection of locally non-Delaunay ridges. Specifically, it first lifts the triangulation by $p \to (p, p^2)$ except for the newly inserted vertex v, which is lifted so that $star(v)$ with only $d + 1$ facets is coplanar. Then it continuously moves the lifted vertex of v to (v, v^2), maintaining the convexity of the lifted triangulation by immediately flipping ridges that become reflex. This approach also works for "RT+1" triangulation.

Rajan's algorithm is in fact an application of *kinetic data structure*. Kinetic data structure tracks the combinatorial changes of the geometric structure during the moving of vertices [Gui98]. By arranging the local updates in the chronological order of "events", it can potentially provide a sequence of local updates to fix the geometric structure while moving. We call a flip algorithm employing this idea an *instant flip algorithm*.

There are also some other instant flip algorithms. Shewchuk [She03] designs one to insert a ridge F into a regular triangulation $\mathcal{T}$, given that $\mathcal{T}$ already contains all the facets of F. Using an instant flip algorithm, Miller and Sheehy [MS13] remove multiple vertices from Delaunay triangulation, and Cheng et al. [CDE+99] eliminate flat tetrahedrons from 3D Delaunay triangulation.

Connectivity of Flip Graph

In a more general context, there are several studies on the flip graph of triangulations and its connectivity. The flip graph of a set of triangulations is a graph where each

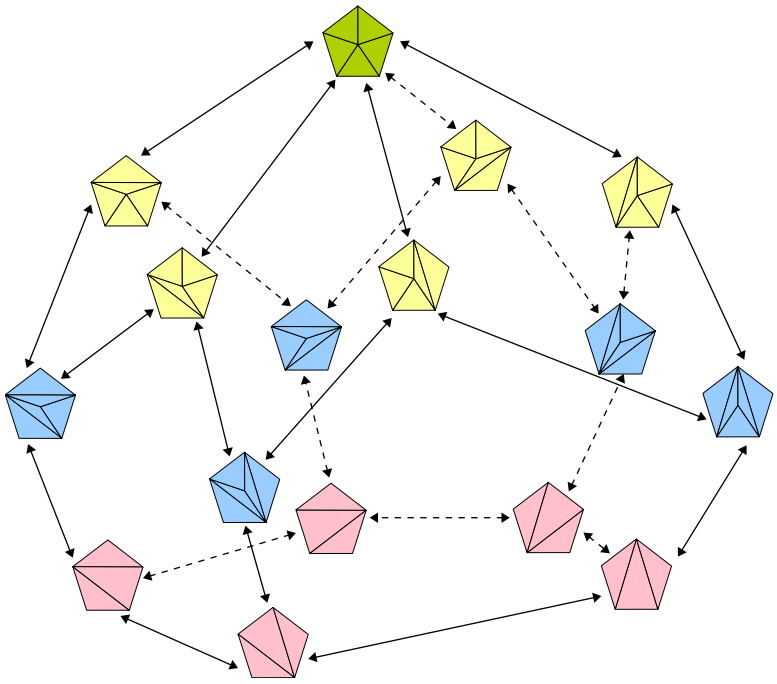


Figure 9.3: The flip graph of the triangulations of six planar points. The six points are arranged such that five form a regular pentagon with the other one locating at the center of the pentagon. The flip graph is combinatorially equivalent to the 1-skeleton of the secondary polytope of the six points.

node represents a triangulation and each edge between two nodes indicates that the two triangulations represented by these nodes differ by exactly one flip; see Figure 9.3 for example. A flip graph is *connected* if each triangulation can be transformed to any other one by flips. The flip graph of all triangulations of a point set S in $\mathbb{R}^2$ is connected, straightforwardly hinted by Lawson's flip algorithm. In contrast, the flip graph of the triangulations of a point set in $\mathbb{R}^d$ ($d \geq 5$) may be disconnected [San00, San05, San06]. It is not known whether such a flip graph in $\mathbb{R}^3$ or $\mathbb{R}^4$ is connected or not.

Gel'fand et al. [GZK91] map each triangulation of n vertices to a point in $\mathbb{R}^n$, whose i-th coordinate is the volume of the star of the i-th vertex, and therefore all the triangulations of a set S of n points are mapped to a set U of points in $\mathbb{R}^n$. The *secondary polytope* of S, $\mathcal{SP}(S)$, is the convex hull of U. The authors prove that an extreme vertex of $\mathcal{SP}(S)$ corresponds to a regular triangulation of S, and an extreme edge of $\mathcal{SP}(S)$ implies that the two corresponding regular triangulations are connected by a flip. Therefore, the 1-skeleton of $\mathcal{SP}(S)$ is the flip graph of all the regular triangulations of S. An important theoretical result naturally obtained is that the flip graph of all the regular triangulations of a point set in any dimension is connected.

Pournin et al. [PL07] identify in the flip graph of the regular triangulations of a point set some particular flip paths that are monotonic w.r.t. the number of vertices.

Based on this, they propose an algorithm to transform one regular triangulation to another, checking whether a flip leads to non-regular triangulation or not in each step. Such checking is computationally inefficient.

Summing up the above results, we know that it is impossible to design a flip algorithm to transform an arbitrary triangulation to its regular triangulation in general dimension because of the disconnectedness of flip graph. The most general yet possibly solvable problem is transforming one regular triangulation to another of the same point set given the weight sets for the two [PL07]. This problem is a generalization of the problem with "RT+1" triangulation [ES92] and the one for deleting multiple vertices [MS13].

In the next section, we use an instant flip algorithm to solve the above mentioned problem, and focus on handling the degeneracy problem.

9.2 Instant Flipping between Regular Triangulations

Let S be a set of n points in $\mathbb{R}^d$ ($d < n$), and W be a weight set of S. With the SoS technique [EM90], we can assume that all the points of S are in general position. Recall that S^W is the lifted point set of $S \oplus W$. Given a triangulation $\mathcal{T}(S)$, the lifted triangulation of $\mathcal{T}(S)$ w.r.t. W, denoted as $\mathcal{T}^W(S)$, is built by replacing all the vertices of $\mathcal{T}(S)$ with their lifted vertices in S^W. By this definition, $\mathcal{RT}^W(S \oplus W)$ is the lower hull of S^W in $\mathbb{R}^{d+1}$.

In a lifted triangulation, let C_1 and C_2 be two facets sharing a common ridge F, and $a \in C_1$ and $b \in C_2$ be the link vertices of F. F is *reflex* if a is beneath the hyperplane through C_2 (equivalently, if b is beneath the hyperplane through C_1). Similarly it is *convex* if a is above the hyperplane. Otherwise it is *flat*. The ridges on the boundary are convex by default.

Our problem is to design a flip algorithm to transform $\mathcal{RT}(S \oplus W)$ to $\mathcal{RT}(S \oplus U)$, where U is another weight set of S. Assume that every vertex of $\mathcal{RT}(S \oplus U)$ belongs to $\mathcal{RT}(S \oplus W)$ so that the less relevant issue of point insertion does not need to be considered.

We build a *moving model* for instant flipping as follows. Maintain a weight set V of S and initialize V with W so that S^V are located in the position of S^W. Move each point of S^V from time 0 to 1 at a constant speed along the axis of the $(d+1)$-th coordinate so that S^V finally arrives at the position of S^U. We represent V as a function of time t: $V(t) = \{v_i(t) \mid v_i(t) = (1-t)w_i + tu_i, 0 \leq i \leq n-1\}$ for $0 \leq t \leq 1$.

In the moving model, the time when a ridge F changes from convex to flat is called the *failure time* of F. The failure time of F is decided by the vertices of $F \cup link(F)$: their coordinates in S and their weights in W and U. Note that a ridge may have no failure time, or its failure time is not in the range (0,1). The former case happens when the ridge is on the boundary or it becomes flat from reflex.

Algorithm 14: The monotonic instant flip algorithm

input : a point set S, its two weight sets W and U, and $\mathcal{RT}(S \oplus W)$
output: $\mathcal{RT}(S \oplus U)$

```
1  T(S) ← RT(S ⊕ W)
2  Q is a priority queue of ridges with failure time as key
3  foreach ridge F in T(S) do
4      t_F ← the failure time of F
5      if t_F ≠ null and t_F ∈ (0,1) then Q ← ⟨F, t_F⟩
6  while Q ≠ ∅ do
7      F ← Q.extractMin()
8      if F ∈ T(S) then
9          flip F
10         foreach boundary facet G of the induced-subcomplex of F do
11             t_G ← the failure time of G
12             if t_G ≠ null and t_G ∈ (0,1) then Q ← ⟨G, t_G⟩
13 RT(S ⊕ U) ← T(S)
```

9.2.1 The Monotonic Instant Flip Algorithm

We show an instant flip algorithm built based on the moving model, named *monotonic instant flip algorithm*. Conceptually, we maintain $\mathcal{RT}^{V(t)}(S \oplus V(t))$, which is $\mathcal{RT}^{W}(S \oplus W)$ when $t = 0$, from time 0 to 1 by instant flipping. At time 1, $\mathcal{RT}^{V(t)}(S \oplus V(t)) = \mathcal{RT}^{U}(S \oplus U)$ gives $\mathcal{RT}(S \oplus U)$.

Algorithm 14 shows the pseudocode of the proposed algorithm. $\mathcal{T}(S)$, initialized as $\mathcal{RT}(S \oplus W)$, is the geometric structure for flipping. A priority queue $\mathcal{Q}$ is maintained so that it always pops the ridge that has minimum failure time. In Line 3–5, all the existing ridges whose failure time is in (0,1) are pushed into $\mathcal{Q}$. The ridge F with the minimum failure time in $\mathcal{Q}$ is extracted from $\mathcal{Q}$ in each iteration, and is flipped if F is still in $\mathcal{T}(S)$ with the same link vertices (Line 7–9). The ridges on the boundary of the induced-subcomplex of F are then pushed into $\mathcal{Q}$ for further consideration (Line 10–12). This process is repeated until $\mathcal{Q}$ is empty.

The proof of correctness of the monotonic instant flip algorithm is straightforwardly inherited from that of kinetic data structure and thus is omitted. Nevertheless, such an algorithm implies the following interesting theorem, according to which the triangulation presented in [Joe89] is a non-regular triangulation.

Theorem 9.3. *A monotonic flip algorithm only gets stuck at non-regular triangulation.*

Proof. Assume this algorithm aims to transform a triangulation of a point set S to $\mathcal{RT}(S \oplus U)$ for some weight set U. By contradiction assume that the algorithm gets stuck at a regular triangulation $\mathcal{T}(S)$ where all the ridges that are locally non-regular w.r.t. U are unflippable. Let W be a weight set so that $\mathcal{T}(S) \equiv \mathcal{RT}(S \oplus W)$. To

transform $\mathcal{RT}(S \oplus W)$ to $\mathcal{RT}(S \oplus U)$, the monotonic instant flip algorithm finds the first ridge to flip. This ridge is locally non-regular and flippable; a contradiction. $\square$

Let $F = \langle a_1, a_2, ..., a_d \rangle$ be a ridge, and $C_1 = \langle a_1, a_2, ..., a_d, a_{d+1} \rangle$ and $C_2 = \langle a_2, a_1, ..., a_d, a_{d+2} \rangle$ be the two facets sharing F. Given a time t, the orientation of C_1 w.r.t. a_{d+2} can be computed by the following $(d+2) \times (d+2)$ determinant:

$$Orient^{d+1}(S_F, t) = \begin{vmatrix} a_1, & {a_1}^2 - w_1 + t(w_1 - u_1), & 1 \\ a_2, & {a_2}^2 - w_2 + t(w_2 - u_2), & 1 \\ ..., & ..., & ... \\ a_d, & {a_d}^2 - w_d + t(w_d - u_d), & 1 \\ a_{d+1}, & {a_{d+1}}^2 - w_{d+1} + t(w_{d+1} - u_{d+1}), & 1 \\ a_{d+2}, & {a_{d+2}}^2 - w_{d+2} + t(w_{d+2} - u_{d+2}), & 1 \end{vmatrix}, \tag{9.1}$$

where $S_F = \langle a_1, a_2, ..., a_d, a_{d+1}, a_{d+2} \rangle$ is the *ordered point set* of F plus its link, and w_i and u_i are the weights of a_i in W and U respectively for $1 \leq i \leq d+2$. Note that ${a_i}^2 - w_i + t(w_i - u_i)$ is the height of a_i at time t in the moving model. At time t, F is convex if $Orient^{d+1}(S_F, t) > 0$, reflex if $Orient^{d+1}(S_F, t) < 0$, and flat otherwise. The failure time of F, if exists, can be computed from $Orient^{d+1}(S_F, t) = 0$.

The following lemma states that a ridge removed by the algorithm never reappears. It relies on the fact that all the vertices move at constant speed in the moving model.

Lemma 9.4. *Any ridge appears at most once in the monotonic instant flip algorithm.*

Proof. Let F be a ridge removed, and $S_F = \langle a_1, a_2, ..., a_d, a_{d+1}, a_{d+2} \rangle$ be the ordered point set of F plus its link. Let $t_F \in (0, 1)$ be the failure time of F and ε be a sufficiently small positive number. By definition, $Orient^{d+1}(S_F, t_F) = 0$, $Orient^{d+1}(S_F, t_F - \varepsilon) > 0$, and $Orient^{d+1}(S_F, t_F + \varepsilon) < 0$. By representing any time $t > t_F$ as $t_F + \lambda\varepsilon$ where $\lambda > 0$, we have

$$\lambda Orient^{d+1}(S_F, t_F - \varepsilon) + Orient^{d+1}(S_F, t_F + \lambda\varepsilon) =$$

$$\begin{vmatrix} a_1, & (\lambda+1)[{a_1}^2 - w_1 + t_F(w_1 - u_1)] + (-\lambda\varepsilon + \lambda\varepsilon)(w_1 - u_1), & 1 \\ a_2, & (\lambda+1)[{a_2}^2 - w_2 + t_F(w_2 - u_2)] + (-\lambda\varepsilon + \lambda\varepsilon)(w_2 - u_2), & 1 \\ ..., & ..., & ... \\ a_d, & (\lambda+1)[{a_d}^2 - w_d + t_F(w_d - u_d)] + (-\lambda\varepsilon + \lambda\varepsilon)(w_d - u_d), & 1 \\ a_{d+1}, & (\lambda+1)[{a_{d+1}}^2 - w_{d+1} + t_F(w_{d+1} - u_{d+1})] + (-\lambda\varepsilon + \lambda\varepsilon)(w_{d+1} - u_{d+1}), & 1 \\ a_{d+2}, & (\lambda+1)[{a_{d+2}}^2 - w_{d+2} + t_F(w_{d+2} - u_{d+2})] + (-\lambda\varepsilon + \lambda\varepsilon)(w_{d+2} - u_{d+2}), & 1 \end{vmatrix}$$

$$= (\lambda+1) Orient^{d+1}(S_F, t_F) = 0.$$

Hence $Orient^{d+1}(S_F, t_F + \lambda\varepsilon) < 0$. Therefore, at any time $t > t_F$, F cannot appear on the lower hull of S_F and thus not on the lower hull of S. □

Indeed, we can further prove that a face appears at most once using a similar proof. Lemma 9.4 implies two properties of the monotonic instant flip algorithm. First, any vertex removed cannot reappear. Second, if a simplex belongs to both $\mathcal{RT}(S \oplus W)$ and $\mathcal{RT}(S \oplus U)$, it is never modified by the algorithm.

Each flip of the monotonic instant flip algorithm either creates or removes one $\lfloor d/2 \rfloor$-simplex [Law87], and there are $n^{\lfloor d/2 \rfloor + 1}$ $\lfloor d/2 \rfloor$-simplices by enumerating all among n points of S. Since any simplex removed cannot reappear in the triangulation, there are at most $n^{\lfloor d/2 \rfloor + 1}$ flips to create $\lfloor d/2 \rfloor$-simplices and at most $n^{\lfloor d/2 \rfloor + 1}$ flips to remove them. Because each flip creates at most d facets, the number of facets created is at most $2dn^{\lfloor d/2 \rfloor + 1}$. The cost of maintaining the priority queue in each flip is therefore $\mathrm{O}\big(\log(2dn^{\lfloor d/2 \rfloor + 1})\big) = \mathrm{O}\big(\lfloor d/2 \rfloor \log n\big)$. Thus the worst-case time complexity of the monotonic instant flip algorithm is $\mathrm{O}\big(\lfloor d/2 \rfloor n^{\lfloor d/2 \rfloor + 1} \log n\big)$.

9.2.2 Computing and Comparing Failure Time

We show how to compute the failure time of a ridge (Line 4 and Line 11 of Algorithm 14). Given a ridge F and the ordered point set $S_F = \langle a_1, a_2, ..., a_d, a_{d+1}, a_{d+2} \rangle$ of F plus its link, define $Orient^d(S_F, a_i)$ as the minor of $Orient^{d+1}(S_F, t)$ by deleting the i-th row and the $(d+1)$-th column:

$$Orient^d(S_F, a_i) = \begin{vmatrix} a_1, & 1 \\ ..., & ... \\ a_{i-1}, & 1 \\ a_{i+1}, & 1 \\ ..., & ... \\ a_{d+2}, & 1 \end{vmatrix}. \tag{9.2}$$

This $(d+1) \times (d+1)$ determinant is indeed that computed during an orientation check on $d+1$ points in $\mathbb{R}^d$ and is not related to the time in the moving model. In addition, it cannot be 0 because of the assumption that the points of S are in general position. Let t_F be the time when F is flat. Then we have

$$\begin{aligned} & Orient^{d+1}(S_F, t_F) && = 0 \\ \Leftrightarrow \quad & \textstyle\sum_{i=1}^{d+2} \{[{a_i}^2 - w_i + t_F(w_i - u_i)] \times Orient^d(S_F, a_i)(-1)^{d+1+i}\} && = 0 \\ \Leftrightarrow \quad & Orient^{d+1}(S_F, 0) + t_F \textstyle\sum_{i=1}^{d+2} [(w_i - u_i) \times Orient^d(S_F, a_i)(-1)^{d+1+i}] && = 0 \\ \Leftrightarrow \quad & Orient^{d+1}(S_F, 0) + t_F[Orient^{d+1}(S_F, 1) - Orient^{d+1}(S_F, 0)] && = 0, \end{aligned}$$

and thus

$$t_F = Orient^{d+1}(S_F, 0)/[Orient^{d+1}(S_F, 0) - Orient^{d+1}(S_F, 1)]. \tag{9.3}$$

When $Orient^{d+1}(S_F, 0) = Orient^{d+1}(S_F, 1)$, the dihedral angle between the hyperplanes passing through the two incident facets of F is unchanged during the movement. When $Orient^{d+1}(S_F, 0) < Orient^{d+1}(S_F, 1)$, F becomes flat from reflex at t_F. In both cases, F has no failure time. When $Orient^{d+1}(S_F, 0) > Orient^{d+1}(S_F, 1)$, t_F is the failure time of F.

Numerical error may happen if we directly store the failure time as real numbers and compare these numbers in the priority queue. Instead, we can compare failure time without explicitly computing its value. Let F and G be two ridges having failure time t_F and t_G, and S_F and S_G be their corresponding ordered point set respectively. By Equation 9.3, the sign of $t_F - t_G$ is equal to that of $Orient^{d+1}(S_F, 1) \times Orient^{d+1}(S_G, 0) - Orient^{d+1}(S_F, 0) \times Orient^{d+1}(S_G, 1)$. As such, the latter equation can be directly evaluated using exact computation when comparison of failure time is needed.

Two ridges may have the same failure time. We avoid this by symbolically perturbing the weights of the input points, similar to the techniques in [She03, DT11]. We fix an order of the points of S called *perturbation order*, and symbolically add a small positive value called *perturbing value* to the weight of each point so that the perturbing value of a point p dominates those of the points behind p in the perturbation order. The perturbing values are so small that they do not affect the comparison of two unequal failure time, while they help to prevent equal failure time.

Let S_F^* be the set of the perturbed points of S_F, and $\epsilon(p)$ be the perturbing value of p. Then we have

$$Orient^{d+1}(S_F^*, t) = \begin{vmatrix} a_1, & {a_1}^2 - w_1 - \epsilon(a_1) + t(w_1 - u_1), & 1 \\ a_2, & {a_2}^2 - w_2 - \epsilon(a_2) + t(w_2 - u_2), & 1 \\ ..., & ..., & 1 \\ a_d, & {a_d}^2 - w_d - \epsilon(a_d) + t(w_d - u_d), & 1 \\ a_{d+1}, & {a_{d+1}}^2 - w_{d+1} - \epsilon(a_{d+1}) + t(w_{d+1} - u_{d+1}), & 1 \\ a_{d+2}, & {a_{d+2}}^2 - w_{d+2} - \epsilon(a_{d+2}) + t(w_{d+2} - u_{d+2}), & 1 \end{vmatrix},$$

and

$$Orient^{d+1}(S_F^*, t) = Orient^{d+1}(S_F, t) - \sum_{i=1}^{d+2} (-1)^{d+i+1} Orient^d(S_F, a_i)\epsilon(a_i) \tag{9.4}$$

Under the perturbation, the sign of $t_F - t_G$ is equal to the sign of

$$Orient^{d+1}(S_F^*, 1) \times Orient^{d+1}(S_G^*, 0) - Orient^{d+1}(S_F^*, 0) \times Orient^{d+1}(S_G^*, 1),$$

which can be decomposed into a main term

$$Orient^{d+1}(S_F, 1) \times Orient^{d+1}(S_G, 0) - Orient^{d+1}(S_F, 0) \times Orient^{d+1}(S_G, 1)$$

and many minor terms, each of which is a multiplication of a perturbing value and its coefficient. The main term decides the sign of $t_F - t_G$ if it is not 0; otherwise the non-zero coefficient of the largest perturbing value does. If all the terms are 0, S_F and S_G have the same points, and F and G are removed by the same flip. Note that the algorithm uses one perturbation order for the whole point set for the sake of consistency and thus the correctness of the algorithm.

When we do not have the assumption of general position in the input point set S, the symbolic perturbation should also be used to prevent degeneracy in $\mathbb{R}^d$. In this case, the coordinates and the weight of each point should be all symbolically perturbed. Then the solution becomes the general symbolic perturbation in $\mathbb{R}^{d+1}$ [EM90].

9.2.3 Finding Perturbation Order

In a special case, the input triangulation $\mathcal{RT}(S \oplus W)$ is a weakly regular triangulation, i.e., $\mathcal{RT}^W(S \oplus W)$ contains some flat ridges. The perturbation order used for comparing failure time must be able to make these flat ridges convex, or otherwise the monotonic instant flip does not have a valid input. In this case, we should specially select a perturbation order for S so that $\mathcal{RT}(S \oplus W)$ is a strongly regular triangulation under it. In this section, we present an algorithm to achieve this target.

Let $\mathcal{T}(S \oplus W)$ be a weakly regular triangulation of $S \oplus W$. Let F be an arbitrary ridge such that $Orient^{d+1}(S_F, 0) = 0$, and S_F be the ordered point set of F plus its link. We intend to find a perturbation order so that $Orient^{d+1}(S_F^*, 0) > 0$, where S_F^* is the perturbed version of S_F. By Equation 9.4, we decompose $Orient^{d+1}(S_F^*, 0)$ into $Orient^{d+1}(S_F, 0)$ and a linear function of the perturbing values of the points in S_F. The key idea is to guarantee that the largest perturbing value in this function has a positive coefficient. With this guarantee, $Orient^{d+1}(S_F^*, 0) > 0$. According to the mentioned decomposition, we define the *positive point set* of F, $S_p(F)$, as the points of S_F whose perturbing values have positive coefficients; and similarly we define the *negative point set of* F, $S_n(F)$.

Algorithm 15 shows the pseudocode to produce such a correct perturbing order $\mathcal{A}$ of the points in S. Assume all the points of S appear in $\mathcal{T}(S \oplus W)$ for simplicity in explanation. For a ridge F with $Orient^{d+1}(S_F, 0) = 0$, we use $perturbed(F)$ to record if it has been successively perturbed, i.e., one point of $S_p(F)$ has been put into $\mathcal{A}$. For a point p, $negDegree(p)$ stores the number of the ridges that have not been perturbed and contain p in their negative point sets, and $posRidges(p)$ is the set of all the ridges that contain p in their positive point sets. These variables are initialized in Line 1–9. The queue $\mathcal{Q}$ stores all the points whose $negDegree$ is 0 (Line 10). In each iteration, a point p is popped from $\mathcal{Q}$ and appended to the end of $\mathcal{A}$ (Line 12–13). For every ridge

Algorithm 15: Finding the perturbation order

input : Weighted point set $S \oplus W$ and a regular triangulation $\mathcal{T}(S \oplus W)$
output: A list $\mathcal{A}$ of S

```
1  A ← ∅
2  foreach point p ∈ S do negDegree(p) ← 0; posRidges(p) ← ∅
3  foreach ridge F ∈ T(S ⊕ W) do
4      if Orient^{d+1}(S_F, 0) = 0 then
5          perturbed(F) ← false
6          foreach p ∈ S_n(F) do negDegree(p) ← negDegree(p) + 1
7          foreach p ∈ S_p(F) do posRidges(p) ← posRidges(p) ∪ F
8      else
9          perturbed(F) ← true
10 Q ← {p | p ∈ S and negDegree(p) = 0}
11 while Q ≠ ∅ do
12     p ← Q.pop()
13     A.append(p)
14     foreach F ∈ posRidges(p) do
15         if perturbed(F) = false then
16             perturbed(F) ← true
17             foreach q ∈ S_n(F) do
18                 negDegree(q) ← negDegree(q) − 1
19                 if negDegree(q) = 0 then Q ← q
```

F in $posRidges(p)$ whose $perturbed(F)$ is false, set $perturbed(F)$ to be true as a point in $S_p(F)$ has been perturbed, decrease $negDegree(q)$ by 1 for each point q in $S_n(F)$, and push q into $\mathcal{Q}$ if $negDegree(q)$ becomes 0 (Line 14–19).

Theorem 9.5. *Algorithm 15 outputs a correct perturbation order if there exists one to realize the input triangulation as a strongly regular triangulation.*

Proof. The algorithm always terminates, because $negDegree(p)$ of a point p is monotonically decreasing and thus p can be pushed into $\mathcal{Q}$ at most once. Let $\mathcal{O}$ be a perturbation order that realizes $\mathcal{T}(S \oplus W)$ as a strongly regular triangulation.

By contradiction assume that when the algorithm terminates, there remain some points whose $negDegree(\cdot)$ are larger than 0. Let p be the point with the highest rank in $\mathcal{O}$ among all the remaining ones. At least one ridge F containing p in $S_n(F)$ is not perturbed; otherwise $negDegree(p) = 0$. All the points in $S_p(F)$ have their $negDegree(\cdot)$ larger than 0, or else F should be perturbed. These points are thus not perturbed, and therefore have lower ranks than p in $\mathcal{O}$ by the choice of p. This indicates that F is locally non-regular by perturbation under $\mathcal{O}$; a contradiction.

Therefore, all the points of S are in $\mathcal{A}$. For a ridge G with $Orient^{d+1}(S_G, 0) > 0$, the perturbation does not affect it. For a ridge F with $Orient^{d+1}(S_F, 0) = 0$, a point

in $S_n(F)$ is appended into $\mathcal{A}$ only after F is perturbed. This indicates that among all the points in S_F, the one with the highest rank in $\mathcal{A}$ must come from $S_p(F)$, and thus $Orient^{d+1}(S_F^*, 0) > 0$. Consequently, all the ridges are locally regular under $\mathcal{A}$. □

Let $|S|$ be the number of points and $|\mathcal{T}|$ be the number of facets of $\mathcal{T}(S \oplus W)$. The number of the ridges of $\mathcal{T}(S \oplus W)$ is thus $\mathrm{O}\big(d|\mathcal{T}|\big)$. Let u be the number of the flat ridges, and D_k be the time complexity of computing a $k \times k$ determinant. The running time of Line 2 and Line 10 is $\mathrm{O}\big(|S|\big)$. In each iteration of the loop in Line 3–9, in order to decide $S_p(F)$ and $S_n(F)$ and compute $Orient^{d+1}(S_F, 0)$, we compute $Orient^d(S_F, a_i)$ for $0 \leq i \leq d+2$ in $\mathrm{O}\big(dD_{d+1}\big)$. Therefore the time complexity of this loop is $\mathrm{O}\big(d^2 D_{d+1}|\mathcal{T}|\big)$. In the loop of Line 11–19, each point is popped from $\mathcal{Q}$ once, and each ridge can exist in the $posRidges(\cdot)$ of at most d points. Thus the number of the times of executing Line 17–19 is $\mathrm{O}\big(du\big)$, and the time complexity of Line 11–19 is thus $\mathrm{O}\big(d^2 u\big)$. Therefore, the worst-case time complexity of Algorithm 15 is $\mathrm{O}\big(d^2(D_{d+1}|\mathcal{T}| + u)\big)$, and is $\mathrm{O}\big(|\mathcal{T}|\big)$ when d is a constant.

9.3 Extended Moving Model

In this section we extend the moving model and further introduce the novel concepts of changing direction and changing time of ridges, aiming to discover new flip algorithms of transforming between regular triangulations. We visualize these properties of the ridges on a time line, and study how they are affected by flips. We finally discuss the implication of this extended moving model.

The extended moving model is built as follows. The lifted point set $S^{V(t)}$ of S moves in parallel to the axis of the $(d+1)$-th coordinate in the time period $(-\infty,+\infty)$ and it overlaps with S^W and S^U at time 0 and 1 respectively; each point of $S^{V(t)}$ moves at a constant speed. Therefore, $V(t) = \{v_i(t) \mid v_i(t) = (1-t)w_i + tu_i, 0 \leq i \leq n-1\}$ for $-\infty < t < +\infty$.

Given a triangulation $\mathcal{T}$ of $S^{V(t)}$, let F be a ridge of $\mathcal{T}$ not on the boundary, and S_F be the ordered point set of F plus its link. During the movement of $S^{V(t)}$ from $t = -\infty$ to $t = +\infty$, F may deform from reflex to convex or reverse. We use *changing direction* of F to indicate this property of F, and denote it as $\sigma(F)$. F has *positive* changing direction, i.e., $\sigma(F) = +$, if it deforms from reflex to convex, and has *negative* changing direction, i.e., $\sigma(F) = -$, if reverse. For simplicity, F is positive (resp., negative) if $\sigma(F) = +$ (reps., $-$). By this definition, a ridge may have a different changing direction and changing time when its link vertices are changed. Therefore in the remaining discussion, we assume two ridges having the same vertices are different if they have different links.

Because $Orient^{d+1}(S_F, t)$, which determines the convexity of F, is a linear function of time t, F cannot deform between reflex and convex more than once, and its changing

Figure 9.4: (a) A time line that is not well-separated and (b) a well-separated time line at $t = 2$. Positive nodes are in black while negative ones are in white.

direction is thus fixed and equals to the sign of $Orient^{d+1}(S_F, 1) - Orient^{d+1}(S_F, 0)$. The *changing time* of F, denoted as $\tau(F)$, is the time when F becomes flat; $Orient^{d+1}(S_F, \tau(F)) = 0$. Especially, for a ridge G that is always convex in $(-\infty, +\infty)$, define $\sigma(G) = +$ and $\tau(G) = -\infty$; for a ridge G' that is always reflex, define $\sigma(G') = -$ and $\tau(G') = -\infty$. We assume that $\mathcal{T}$ does not have a ridge that is always flat in $(-\infty, +\infty)$, since such a ridge can be eliminated by symbolic perturbation (see Section 9.2.2).

The *time line* of the triangulation $\mathcal{T}$ is a time axis spanning $(-\infty, +\infty)$ with embedded nodes, each of which represents a ridge of $\mathcal{T}'$. The node N representing a ridge F is called the *node* of F, and F is called the ridge for N. Define the changing direction and changing time of N as those of F. N is placed at $\tau(N)$ on the time line, and is *larger* (resp., *smaller*) than another node N' if it is on the right (resp., left) of N', i.e. $\tau(N)$ is larger (resp., smaller) than $\tau(N')$.

A time line is *well-separated* at a time t if $\tau(N_+) \leq t \leq \tau(N_-)$ for any positive node N_+ and any negative node N_-. Figure 9.4 shows two examples of time line. Using the symbolic perturbation in Section 9.2.2, we can assume that two nodes cannot have the same changing time unless they are included in the same flip. Note that a time line spans the whole time period $(-\infty, +\infty)$, and thus is a static description of a triangulation or its subset.

Theorem 9.6. *$\mathcal{T}$ is a regular triangulation if its time line is well separated.*

Proof. Let t_0 be a time so that the time line of $\mathcal{T}$ is well separated at t_0. At time t_0, the ridge for each positive node is convex by the definition of changing direction; similarly, the ridge for each negative node is also convex. Therefore in the original space, all these ridges are locally regular. By Theorem 2.2, $\mathcal{T}$ is equal to $\mathcal{RT}(S \oplus V(t_0))$. □

Given $\mathbb{T}$ as the time line of $\mathcal{T}$, a flip on $\mathcal{T}$ not only changes $\mathcal{T}$, but also affects $\mathbb{T}$. In order to explain these changes in $\mathbb{T}$ concisely and clearly, we use a circle to represent $\mathbb{T}$; see Figure 9.5. The time $-\infty$ and $+\infty$ overlap at the so-called *infinity boundary*, and the nodes whose changing time is $-\infty$ are placed in that position. All the nodes are located so that their relative positions are retained.

We first show that how a flip affects the nodes of the ridges that it creates and removes using the following lemma.

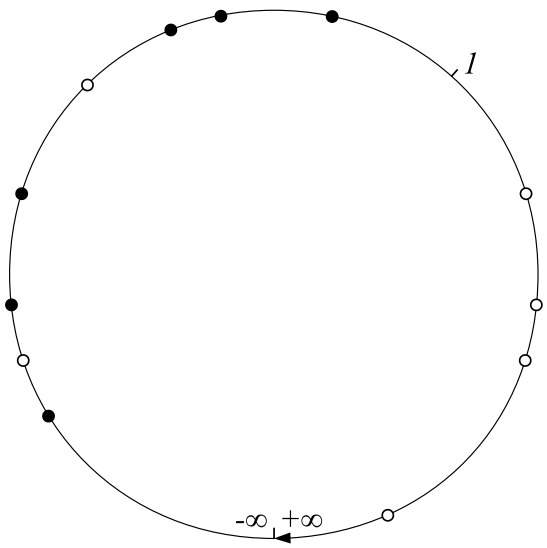


Figure 9.5: A circular time line is created by bending a time line and joining the time $-\infty$ and $+\infty$.

Lemma 9.7. *For a flip on $\mathcal{T}$, all the nodes removed have the same changing direction σ_r and changing time τ_r, and all the nodes created have the same changing direction σ_c and changing time τ_c. Furthermore, $\sigma_r = -\sigma_c$ and $\tau_r = \tau_c$.*

Proof. We only prove the lemma for $\sigma_r = +$. Similar arguments can be used to prove the lemma for $\sigma_r = -$. Let S_F be the set of vertices included in the flip. Clearly, both τ_r and τ_c are the time when the points of S_F are coplanar, and thus $\tau_r = \tau_c$. Since $\sigma_r = +$, all the facets removed by the flip form the upper hull of S_F before τ_r and the lower hull of S_F after τ_r. Conversely, all the facets created by the flip form the lower hull of S_F before τ_c and the upper hull of S_F after τ_c. Therefore $\sigma_r = -\sigma_c$. Note that the lemma is still valid if $\tau_r = \tau_c = -\infty$. □

By Lemma 9.7, all the ridges created by a flip can be represented by the same node, so can all the ridges removed. Furthermore, as the ridges created and removed do not exist in $\mathcal{T}$ at the same time, we use the same node for all these ridges and allow the flip to change the properties of the node. The next lemma states how a flip affects the nodes of the ridges on the boundary of its induced-subcomplex.

Lemma 9.8. *For a flip on $\mathcal{T}$, let N in $\mathbb{T}$ be the node of all the ridges created and removed, and N' be the node of one ridge on the boundary of the induced-subcomplex before and after the flip. Then the flip pulls N' nearer to N if $\sigma(N') = \sigma(N)$ before the flip; otherwise pushes N' further from N. When N' crosses the infinity boundary, its changing direction is inverted.*

Proof. Let F be a ridge removed by the flip and S_F be the set of the vertices included in the flip. Let G and G' be the ridges for N' before and after the flip respectively.

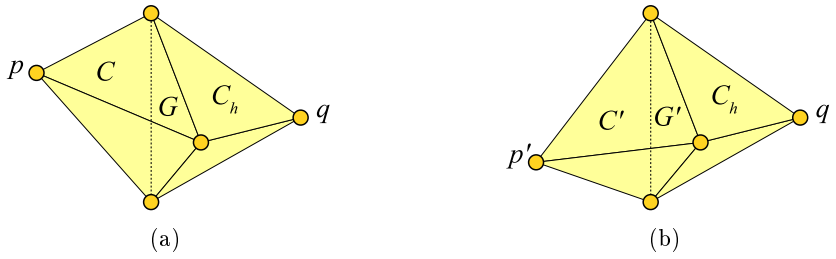


Figure 9.6: For a 3D flip and its related induced-subcomplex, a boundary ridge and its two incident facets (a) before and (b) after the flip.

G and G' have the same vertices, but differ in one link vertex. Let C be the facet of the induced-subcomplex incident to G before the flip, and C' be the one incident to G' after the flip. The vertices $p \in C$ and $p' \in C'$ are respectively link vertices of G and G'. Let C_h be the other facet incident to G and G' (not in the induced-subcomplex), and $q \in C_h$ be a link vertex of G and G'. See Figure 9.6 for an illustration in $\mathbb{R}^3$.

At time $\tau(F)$, C and C' lie on the same hyperplane, and therefore the status of G' (i.e., whether it is convex, flat or reflex) is the same as that of G. At time $\tau(G)$, since C and C_h lie on the same hyperplane, the relation between p' and the hyperplane passing through C_h is equal to that between p' and the hyperplane passing through C; the latter relation reflects whether F appears on the lower hull or the upper hull of S_F. Therefore the status of G' is the same as that of F at time $\tau(G)$. Note that this result is still valid if one of $\tau(F)$ and $\tau(G)$ is $-\infty$. We use these important results to prove the following case analysis.

Without loss of generality, assume that $\sigma(F) = +$. Based on the results in the previous paragraph, we can list all the cases based on $\sigma(G)$ and the relation between $\tau(F)$ and $\tau(G)$:

(1) if $\sigma(G) = +$ and $\tau(G) < \tau(F)$, then $\sigma(G') = +$ and $\tau(G) < \tau(G') < \tau(F)$
(2) if $\sigma(G) = +$ and $\tau(F) < \tau(G)$, then $\sigma(G') = +$ and $\tau(F) < \tau(G') < \tau(G)$
(3) if $\sigma(G) = -$ and $\tau(G) < \tau(F)$, then either
  (a) $\sigma(G') = -$ and $\tau(G') < \tau(G) < \tau(F)$, or
  (b) $\sigma(G') = +$ and $\tau(G) < \tau(F) < \tau(G')$
(4) if $\sigma(G) = -$ and $\tau(F) < \tau(G)$, then either
  (a) $\sigma(G') = -$ and $\tau(F) < \tau(G) < \tau(G')$, or
  (b) $\sigma(G') = +$ and $\tau(G') < \tau(F) < \tau(G)$

In (1), $\sigma(G) = +$ and $\tau(G) < \tau(F)$ indicate that F is reflex at $\tau(G)$ and G is convex at $\tau(F)$. Therefore G' is reflex at $\tau(G)$ and convex at $\tau(F)$, which implies that $\sigma(G') = +$ and $\tau(G) < \tau(G') < \tau(F)$. Similarly in (2), G' is reflex at $\tau(F)$ and convex at $\tau(G)$; therefore $\sigma(G') = +$ and $\tau(F) < \tau(G') < \tau(G)$. In (3), G' is reflex at both $\tau(G)$ and $\tau(F)$, implying two cases: if $\sigma(G') = +$, $\tau(G')$ must be larger than both $\tau(G)$ and $\tau(F)$; otherwise $\tau(G')$ must be smaller than both $\tau(G)$ and $\tau(F)$. In (4), G' is

convex at both $\tau(F)$ and $\tau(G)$. Therefore, $\tau(G')$ must be smaller than $\tau(G)$ and $\tau(F)$ if $\sigma(G') = +$, or be larger than $\tau(G)$ and $\tau(F)$ otherwise.

Note that the above cases include the consideration of time $-\infty$. When $\tau(F)$ be $-\infty$, $\tau(G)$ and $\tau(G')$ cannot be $-\infty$ or smaller because of the symbolic perturbation. Then (1), (3) and (4.a) automatically become invalid while the rests are still true. On the other hand, (2), (4) and (3.a) become invalid when $\tau(G)$ is $-\infty$. □

Note that a flip never changes the ridges outside induced subcomplex. Summarizing this fact together with Lemma 9.7 and Lemma 9.8, we have the following theorem.

Theorem 9.9. *For a flip on $\mathcal{T}$, use a node N in $\mathbb{T}$ for all the removed and created ridges, and use a node N' for one ridge on the boundary of the induced-subcomplex before and after the flip. Then the following statements are true:*

(1) The flip does not change $\tau(N)$, but inverts $\sigma(N)$.

(2) The flip pulls N' nearer to N if $\sigma(N') = \sigma(N)$ before the flip, otherwise pushes N' further from N. When N' crosses the infinity boundary, its changing direction is inverted.

(3) The flip does not affect any ridge outside the induced subcomplex.

We review the monotonic instant flip algorithm based on Theorem 9.9. As discussed in Section 9.2.1, the monotonic instant flip algorithm always flip the ridge that has the minimum failure time. That ridge corresponds to the minimum negative node on the time line, and flipping it keeps the time line being well-separated, as shown in the following lemma.

Lemma 9.10. *If $\mathbb{T}$ is well-separated, the ridge F of $\mathcal{T}$ for the minimum negative node is flippable, and $\mathbb{T}$ is still well-separated after flipping F.*

Proof. Let N_F be the minimum negative node of $\mathbb{T}$ and ϵ be a sufficiently small positive value. At $\tau(F) + \epsilon$, F is reflex, while all the other facets are convex. By Lemma 9.2, F is flippable, or otherwise there must be some other reflex ridges. As indicated in Theorem 9.9, flipping F will remove the negative nodes at $\tau(F)$ from $\mathbb{T}$, and insert some positive nodes at $\tau(F)$ into $\mathbb{T}$.

Let G be a ridge on the boundary of the induced-subcomplex of F, and G' be the corresponding ridge of G after the flip. G and G' have the same vertices, but differ at one link vertex. If $\sigma(G) = -$, then $\tau(G) > \tau(F)$, and therefore $\sigma(G') = -$ and $\tau(F) < \tau(G') < \tau(G)$ after the flip. If $\sigma(G) = +$, then $\tau(G) < \tau(F)$, and thus either (a) $\sigma(G') = +$ and $\tau(G') < \tau(G) < \tau(F)$ or (b) $\sigma(G') = -$ and $\tau(G') > \tau(F)$. Whichever case happens, $\mathbb{T}$ after flipping F is well-separated at $\tau(F) + \epsilon$. □

In the monotonic instant flip algorithm, the time line of the initial triangulation, $\mathcal{RT}(S \oplus W)$, is well-separated at time 0. By flipping the first ridge F_1, the new time line is also well-separated. In the new time line, the changing time of the maximum

positive node is $\tau(F_1)$ while that of the minimum negative node is larger than $\tau(F_1)$. The algorithm repeats this procedure until it obtains a time line where the changing time of the maximum positive node is smaller than 1 and that of the minimum negative node is larger than 1. This final time line is well-separated at time 1, and thus the triangulation is $\mathcal{RT}(S \oplus U)$. By Theorem 9.6, all the triangulations created during the algorithm are regular. As a summary, the **criterion** of the monotonic instant flip algorithm is to flip the minimum negative node on the time line of the triangulation.

The concepts of changing direction and changing time in the extended moving model provide new information for local transformation between regular triangulations. While two locally non-regular ridges have not much differences in the traditional monotonic flip algorithm, they are distinctive when we look at their changing direction and changing time. The time line extracts and presents the changing direction and the changing time of the ridges in a simple 1D structure (i.e., a circle). On the other hand, it does not display the flippability of ridges. However, we can derive the flippability of some ridges from the time line, e.g. in Lemma 9.10. As another example, Lemma 9.11 identifies a flippable ridge locally in a star.

Lemma 9.11. *Given a $(d-2)$-face R of $\mathcal{T}$ and the time line $\mathbb{T}_R$ of $star(R)$, the ridge for the minimum negative node in $\mathbb{T}_R$ is flippable if $\mathbb{T}_R$ is well-separated.*

Proof. This follows directly from Lemma 9.2, similar to the proof of Lemma 9.10. □

To sum up, the concepts presented in this section open a new way to understand and analyze flipping procedure, and we hope they can assist in making a breakthrough on flipping triangulations in higher dimensions.

CHAPTER 10

Conclusions

Coming to the end of the thesis, we hereby summarize our understanding of local transformation till date, and provide insights on a possible future journey on the subject.

10.1 Local Transformation Revisit

Around the topics of local transformation, we study and develop a series of algorithms involving around flipping, splaying and twisting.

On flipping, it turns out that one can relax the hill-climbing approach to develop the powerful flip-flop algorithm to solve the 2D regular triangulation and the 3D convex hull problems. However, flip-flop, as it is, does not move beyond 3D convex hull to higher dimensions.

On splaying, it was known to be very powerful with the ability to fix convex hull in any dimension, but not clear on its use in practice to efficiently compute the convex hull of a point set. The thesis implements a way, through the digital restricted Voronoi diagram of the given point set, to provide a set of convex stars as input to the star-splaying algorithm.

On twisting, it is a new operation introduced in this thesis. This was motivated by the need to obtain a star-shaped polygon from an arbitrary, possibly self-interesting, polygon. This operation is also used in an attempt to transform a polyhedron to a star-shaped one. The latter is solved with the twist-flip algorithm only for the restricted case of input polyhedron with an extreme vertex connecting to all the other vertices. The general problem remains open.

On the whole, the work also recognizes some possible disadvantages of the algorithms of local transformation. The major one is the known worst case time complexities of quadratic for 2D Lawson's flip algorithm and 2D twist algorithm, and cubic for 3D flip-flop algorithm and 3D star-splaying algorithm. Nevertheless, such time complexities are hardly observed or even possible in practice.

The simplicity of algorithms of local transformation remains an important advan-

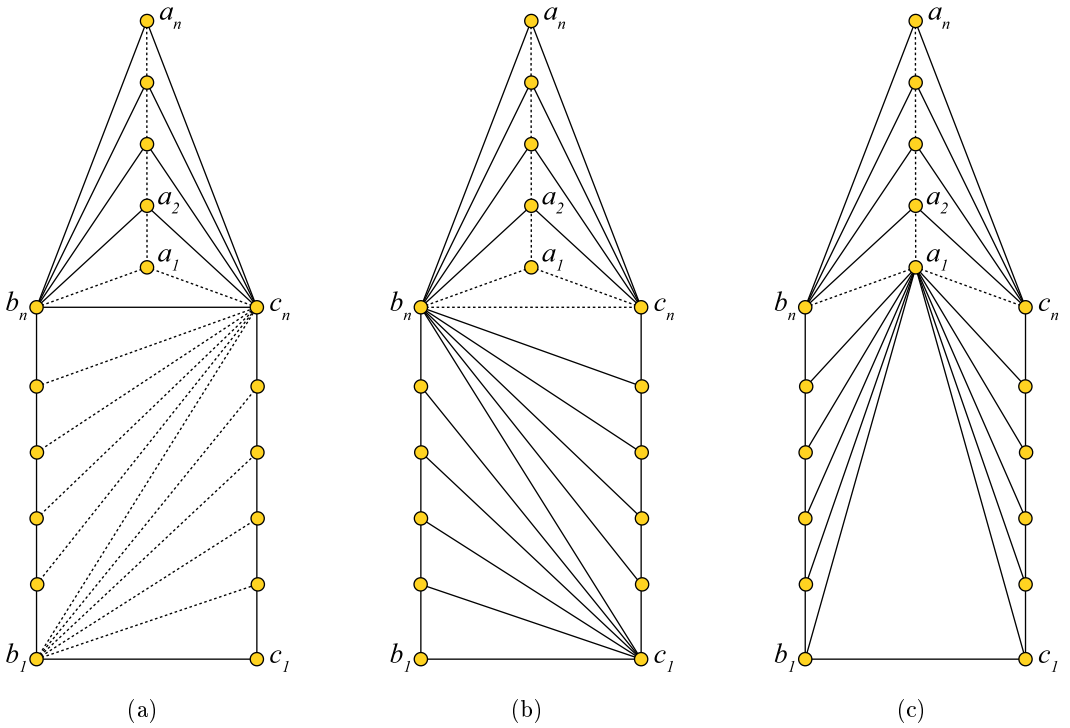


Figure 10.1: A triangulation of $3n$ points from which flipping according to the V- and the D-criterion may contain $\Theta(n^3)$ flips. Locally non-regular edges are drawn as dashed. Flipping by the V-criterion from (a) to (b) may contain $\Theta(n^2)$ flips. Then flip the edge $\overline{b_i c_j}$ in $link(a_1)$ by the V-criterion until (c) is obtained. In (c), a_1 is identified as redundant because of $\overline{a_1 a_2}$. Flipping the edges of $star(v)$ by the D-criterion removes a_1, resulting into a triangulation similar to (a). Repeat this procedure until a_{n-1} is removed. The whole process has $n-1$ rounds; each contains $\Theta(n^2)$ flips.

tage for their potential or already extensive uses in practice. For example, twist-flip, which is the most complicated algorithm among all the proposed ones in this thesis, employs only twist and flip as operations, and uses fairly simple criteria. In addition, such algorithms are powerful as a repairing tool, and can be mapped to parallel machines when they allow free execution order. In short, we demonstrate that these algorithms can be efficient in practice, and in particular when they are implemented on the GPU.

Open problem: What is the time complexity of flip-flop?

The obvious worst-case time complexity of flip-flop is $O(n^3)$, when the triangulation has $O(n)$ redundant vertices and each is discovered after $O(n^2)$ flips. Figure 10.1 shows such a triangulation of $3n$ vertices with n non-extreme ones. The flip-flop algorithm indeed uses $\Theta(n^3)$ flips if it strictly follows the flipping sequence mentioned in the caption of the figure. This, however, does not prove that cubic is a tight bound, as there remains flexibility in flip-flop to operate in different ways. For example, we can

immediately focus on removing a vertex as soon as it is identified as redundant. In this way, the triangulation in Figure 10.1 is transformed to regular triangulation in $O(n^2)$.

10.2 Dimensional Dependency of Geometric Structures

As discussed in Chapter 2, convex hull and regular triangulation are closely related structures. Here we further present another interpretation about their relation that can be expanded to understand the relation between triangulation and star-shaped polytope. From the latter, we can understand why star-shaped polytope is a workable input for our flip algorithms of computing 2D and 3D convex hull.

On one hand, a k-dimensional regular triangulation can be obtained from the $(k+1)$-dimensional convex hull of its lifted vertices. On the other hand, a $(k+1)$-dimensional convex hull is the boundary of a $(k+1)$-dimensional regular triangulation of its vertices. From these relations, $(k+1)$-dimensional convex hull can be seen as a regular triangulation whose dimension is between k and $k+1$. We therefore interpret it as a $k.5$-dimensional regular triangulation.

An analogous argument can be applied on triangulation and star-shaped polytope. On one hand, a k-dimensional extended triangulation can be deduced from a $(k+1)$-dimensional star-shaped polytope, by realizing the virtual point to be a point in the $(k+1)$-th dimension. On the other hand, a $(k+1)$-dimensional star-shaped polytope is the boundary between the real and the virtual facets of a $(k+1)$-dimensional extended triangulation. Hence, a $(k+1)$-dimensional star-shaped polytope can be seen as a $k.5$-dimensional extended triangulation.

Furthermore, as explained next, a $(k+1)$-dimensional star-shaped polytope triangulates a set of $(k+1)$-dimensional points in a k-dimensional manner. Each facet of the polytope, when associated with the kernel point, corresponds to a subspace that is the convex hull of all the rays starting at the kernel point and passing through the vertices of the facet (e.g., the cone defined in Section 6.1). The subspaces of all the facets partition $\mathbb{R}^{k+1}$ without any overlapping, and thus we say the polytope triangulates these points. Yet, the polytope is actually a k-manifold without boundary, and thus we say such a triangulation is done in a k-dimensional manner.

Because of this understanding, we successfully use flip-flop, which originally works for 2D regular triangulation, to transform an arbitrary star-shaped polyhedron to its convex hull. Therefore, flip-flop works for both 2D and 2.5D problems. In Figure 10.2 we list the flip algorithms that compute regular triangulation with free execution order. In some way related to this, we develop twist-flip to transform a polyhedron with one extreme vertex connecting to all other vertices into a star-shaped polyhedron.

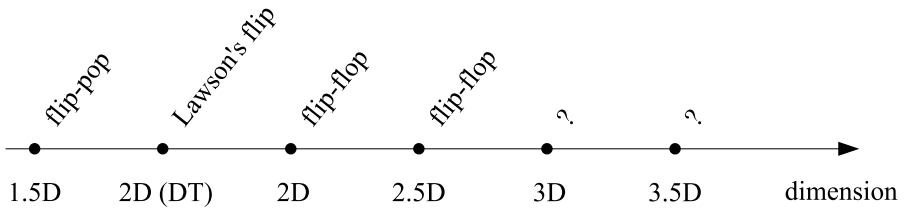


Figure 10.2: Flip algorithms that compute regular triangulation with free execution order, sorted by dimensions.

Open problem: How to use the spirit of flip-flop in 3D regular triangulation?

The major breakthrough of flip-flop compared with the traditional hill-climbing flip algorithm is its ability to identify and remove redundant vertices so as to avoid getting stuck at a local optimum. However, it is difficult to be extended for 3D triangulation, where monotonic flipping gets stuck not only because of redundant vertices but also because of redundant edges. A major difficulty is that a redundant vertex or a redundant edge may not be removable by flipping the triangles of its stars, since all these triangles can be unflippable; see Figure 10.3 as an example for redundant vertex. The keys of solving this problem rely on how to escape from local optimum and how to prevent reappearance of redundant edges.

10.3 Concluding Remarks

Before this research, it was open for the following sample of computational problems:

Question 1: Can one perform flipping to transform an arbitrary 2D triangulation to its regular triangulation?

Question 2: Can one perform flipping to transform an arbitrary 3D triangulation to its regular triangulation?

Question 3: Can one perform flipping to transform between two regular triangula-

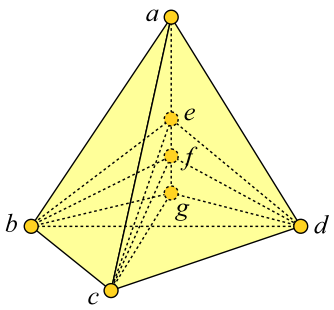


Figure 10.3: This triangulation is created by successively inserting e, f and g into the tetrahedron $abcd$. Set the weight of e to be very small so that e is redundant. All the triangles of $star(e)$ are unflippable.

tions of the same point set in an arbitrary dimension?

Question 4: Can one perform local transformation to compute star-shaped polytope from an arbitrary polytope?

Question 1 is solved with our flip-flop in Chapter 5. The bonus of this is also a solution to the 3D convex hull problem. One remaining open problem here is the time complexity of flip-flop.

On the other hand, Question 2 remains open still, though the success demonstrated by the flip-flop algorithm is an indication that one could possibly look for relaxation of the hill-climbing flip algorithm to solve this question. As mentioned in the previous section, this is really challenging.

As for Question 3, it is affirmative when flips are structured in some sequence, but remains elusive whether a free execution order is possible. Upon reflection, a successful flip algorithm should either use monotonic flips while avoiding getting stuck at some bad configuration, or use non-monotonic flips in escaping from the bad configuration. The algorithm in Chapter 9, which has sequential execution order, belongs to the former, while flip-flop in Chapter 5 and Chapter 6 belongs to the latter. The introduction of time line to the study of changing time and changing direction of ridges in Chapter 9 is one hope to understand this problem.

For Question 4, the thesis positively answers its 2D version and part of its 3D version by devising the twist and the twist-flip algorithms. It remains open how to locally transform a general polyhedron to a star-shaped polyhedron. Such a problem becomes even more difficult if the given kernel point is an arbitrary point in the convex hull of the polyhedron or if the dimension is higher than three.

Augmenting Table 3.1 with new algorithms presented in this thesis and an additional column about output geometric structures, Table 10.1 summarizes the current understanding about algorithms of local transformation. With the local transformation framework dealing with geometric structure, local operation, criterion, and local check, plus consideration of whether local operations need to be ordered in some way, we leave behind more to be understood to expand upon the table!

Table 10.1: Algorithms of local transformation and their features.

Algorithm	Dimension	Input	Output	Operation	Order
Graham's scan	2	star-shaped polygon	convex hull	2-1 flip	sequential
flip-pop	2	star-shaped polygon	convex hull	2-1 flip	free
Lawson's flip algorithm	2	triangulation	Delaunay triangulation	2-2 flip	free
flip-flop	2	triangulation	regular triangulation	2-2/3-1 flip	free
	3	star-shaped polyhedron	convex hull	2-2/3-1 flip	free
Joe's flip algorithm	*any*	"DT+1" triangulation	Delaunay triangulation	flip	free
Rajan's flip algorithm	*any*	"RT+1" triangulation	regular triangulation	flip	sequential
Edelsbrunner-shah's flip algorithm	*any*	"RT+1" triangulation	regular triangulation	flip	free
star splaying	*any*	a set of stars*	convex hull	splaying	free
monotonic instant flip algorithm	*any*	regular triangulation	regular triangulation	flip	sequential
twist algorithm	2	polygon	star-shaped polygon	twist	free
twist-flip	3	special polyhedron†	star-shaped polyhedron	twist, flip	free

*The star of each vertex must contain a lexicographically smaller vertex

†The input polyhedron has an extreme vertex that connects to all the other vertices; the output polyhedron is star-shaped w.r.t. this extreme vertex

Bibliography

[ACHS05] Neal R. Amundson, Alexandre Caboussat, Jiwen He, and John H. Seinfeld. An optimization problem related to the modeling of atmospheric organic aerosols. *Comptes Rendus Mathematique*, 340(10):765 – 768, 2005.

[ACK01] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications*, SMA '01, pages 249–266, New York, NY, USA, 2001. ACM.

[Alb03] Lyuba Alboul. Optimising triangulated polyhedral surfaces with self-intersections. In Michael Wilson and Ralph Martin, editors, *Mathematics of Surfaces*, volume 2768 of *Lecture Notes in Computer Science*, pages 48–72. 2003.

[AP93] Nancy M. Amato and Franco P. Preparata. An NC parallel 3D convex hull algorithm. In *SoCG '93: Proc. 9th Symp. Computational Geometry*, pages 289–297, New York, NY, USA, 1993. ACM.

[ASY08] Pankaj K. Agarwal, Bardia Sadri, and Hai Yu. Untangling triangulations through local explorations. In *Proceedings of the Twenty-fourth Annual Symposium on Computational Geometry*, SCG '08, pages 288–297, New York, NY, USA, 2008. ACM.

[AT78] Selim G. Akl and Godfried T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219 – 222, 1978.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Mathematical Software*, 22(4):469–483, 1996.

[BFS90] Louis J. Billera, Paul Filliman, and Bernd Sturmfels. Constructions and complexity of secondary polytopes. *Advances in Mathematics*, 83(2):155 – 179, 1990.

[BMHT99] G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3-4):243–269, 1999.

[BMPS09] Vicente H.F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. In *Proceedings of the Twenty-fifth Annual Symposium on Computational Geometry*, SCG '09, pages 217–226, New York, NY, USA, 2009. ACM.

[Bow81] A. Bowyer. Computing Dirichlet tessellations. In *Comput. J. 24*, pages 162–166, 1981.

[BSDMH05] Tilo Beyer, Gernot Schaller, Andreas Deutsch, and Michael Meyer-Hermann. Parallel dynamic and kinetic regular triangulation in three dimensions. *Computer Physics Communications*, 172(2):86 – 108, 2005.

[CDE+99] Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver exudation. In *Proceedings of the fifteenth annual symposium on Computational geometry*, SCG '99, pages 1–13, New York, NY, USA, 1999. ACM.

[CGA12] CGAL. CGAL, computational geometry algorithms library, 2012. `http://www.cgal.org`.

[Cha96] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(4):361–368, 1996.

[Cig98] P. Cignoni. DeWall: a fast divide and conquer Delaunay triangulation algorithm in E^d. *Computer-Aided Design*, 30(5):333–341, April 1998.

[CK70] Donald R. Chand and Sham S. Kapur. An algorithm for convex polytopes. *J. ACM*, 17(1):78–86, January 1970.

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

[CUB14] CUB. CUB, CUDA data parallel primitives library, 2014. `http://nvlabs.github.io/cub`.

[CUD12] CUDPP. CUDPP, CUDA data parallel primitives library, 2012. `http://gpgpu.org/developer/cudpp`.

[DDD+95] Frank Dehne, Xiaotie Deng, Patrick Dymond, Andreas Fabri, and Ashfaq A. Khokhar. A randomized parallel 3D convex hull algorithm for coarse grained multicomputers. In *SPAA '95: Proc. 7th ACM Symp. Parallel Algorithms and Architectures*, pages 27–33, New York, NY, USA, 1995. ACM.

[DLRS10] Jesus A. De Loera, Jorg Rambau, and Francisco Santos. *Triangulations: structures for algorithms and applications.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[DT11] Olivier Devillers and Monique Teillaud. Perturbations for Delaunay and weighted Delaunay 3D triangulations. *Comput. Geom. Theory Appl.*, 44(3):160–168, April 2011.

[Dwy87] Rex Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(1):137–151, November 1987.

[Edd77] William F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3(4):398–403, December 1977.

[Ede01] Herbert Edelsbrunner. *Geometry and topology for mesh generation.* Springer-Verlag, 2001.

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

[ES91] Herbert Edelsbrunner and Weiping Shi. An O(nlog2h) time algorithm for the three-dimensional convex hull problem. *SIAM J. Comput.*, 20:259–269, March 1991.

[ES92] Herbert Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *SCG '92: Proc. 8th Symp. Computational Geometry*, pages 43–52, New York, NY, USA, 1992. ACM.

[Fer01] J.A. Ferrez. *Dynamic triangulations for efficient 3D simulation of granular materials.* 2001.

[FGE01] Olac Fuentes, Ravi K. Gulati, and Optica Y Electronica. Prediction of stellar atmospheric parameters from spectra, spectral indices and spectral lines using machine learning. In *Experimental Astronomy 12:1*, pages 21–31, 2001.

[For97] Steven Fortune. *Handbook of discrete and computational geometry.* CRC Press, Inc., Boca Raton, FL, USA, 1997.

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

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. In *Information Processing Letters*, pages 132–133, 1972.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Trans. Graph.*, 4(2):74–123, April 1985.

[GS03] Neelima Gupta and Sandeep Sen. Faster output-sensitive parallel algorithms for 3D convex hulls and vector maxima. *J. Parallel and Distributed Computing*, 63(4):488 – 500, 2003.

[Gui98] Leonidas J. Guibas. Kinetic data structures: a state of the art report. In *Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective: the algorithmic perspective*, WAFR '98, pages 191–209, Natick, MA, USA, 1998. A. K. Peters, Ltd.

[GZK91] I. M. Gel'fand, A. V. Zelevinskii, and M. M. Kapranov. Discriminants of polynomials in several variables and triangulations of Newton polyhedra. *Leningrad Mathematical Journal*, 2(3):449–505, 1991.

[HDSB01] Kenneth H. Huebner, Donald L. Dewhirst, Douglas E. Smith, and Ted G. Byrom. *The finite element method for engineers.* Wiley, New York, NY, USA, 2001.

[HH06] Hernsoo Hahn and Youngjoon Han. 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. 2006.

[Jar73] R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18 – 21, 1973.

[JD11] Tomasz Jurkiewicz and Piotr Danilewski. Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations. November 2011. `http://www.mpi-inf.mpg.de/~tojot/papers/chull.pdf`.

[Joe89] Barry Joe. Three-dimensional triangulations from local transformations. *SIAM J. Scientific and Statistical Computing*, 10(4):718–741, July 1989.

[Joe91] Barry Joe. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer Aided Geometric Design*, 8(2):123–142, May 1991.

[Joe93] Barry Joe. Construction of K-dimensional Delaunay triangulations using local transformations. *SIAM J. Sci. Comput.*, 14(6):1415–1436, November 1993.

[KKS06] Dong-Soo Kang, Yun-Jin Kim, and Byeong-Seok Shin. Efficient large-scale terrain rendering method for real-world game simulation. In *Proceedings of the First international conference on Technologies for E-Learning and Digital Entertainment*, Edutainment'06, pages 597–605, Berlin, Heidelberg, 2006. Springer-Verlag.

[KS86] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15:287–299, February 1986.

[Law72] C. L. Lawson. Transforming triangulations. *Discrete Mathematics*, 3(4):365–372, 1972.

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

[Law87] Charles L. Lawson. Properties of n-dimensional triangulations. *Comput. Aided Geom. Des.*, 3(4):231–246, January 1987.

[LNSV06] Marc Lanctot, Nicolas Ng, Man Sun, and Clark Verbrugge. Path-finding for large scale multiplayer computer games. Technical report, 2006.

[LZB08] Rong Liu, Hao Zhang, and James Busby. Convex hull covering of polygonal scenes for accurate collision detection in games. In *GI '08: Proc. Graphics Interface*, pages 203–210, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[Mei75] G.H. Meisters. Polygons have ears. *The American Mathematical Monthly*, 82(6):648 – 651, 1975.

[MII96] Tomonari Masada, Hiroshi Imai, and Keiko Imai. Enumeration of regular triangulations. In *Proceedings of the twelfth annual symposium on Computational geometry*, SCG '96, pages 224–233, New York, NY, USA, 1996. ACM.

[MK04] Pavel Maur and Ivana Kolingerová. The employment of regular triangulation for constrained Delaunay triangulation. In *ICCSA 2004: International Conference of Computational Science and Its Applications*, volume 3045, pages 198–206. Springer, 2004.

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

[MS13] Gary L. Miller and Donald R. Sheehy. A new approach to output-sensitive Voronoi diagrams and Delaunay triangulations. In *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry*, SoCG '13, pages 281–288, New York, NY, USA, 2013. ACM.

[Mun84] J.R. Munkres. *Elements of algebraic topology*. Advanced book classics. Perseus Books, 1984.

[MY06] Hai Mao and Yee-Hong Yang. Particle-based immiscible fluid-fluid collision. In *GI '06: Proc. Graphics Interface*, pages 49–55, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[NHS11] Cristobal Navarro, Nancy Hitschfeld, and Eliana Scheihing. A parallel GPU-based algorithm for Delaunay edge-flips. In *EuroCG '11: 27th European Workshop on Computational Geometry*, pages 75–78. I. M. Hoffmann, Ed., 2011.

[OII03] K. Okada, M. Inaba, and H. Inoue. Walking navigation system of humanoid robot using stereo vision based floor recognition and path planning with multi-layered body image. In *IROS '03 Int'l Conf. Intelligent Robots and Systems*, pages 2155 – 2160 vol.3. IEEE, 2003.

[O'R98] Joseph O'Rourke. *Computational geometry in C*. Cambridge University Press, New York, NY, USA, 2nd edition, 1998.

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

[PL07] L. Pournin and Th. M. Liebling. Constrained paths in the flip-graph of regular triangulations. *Comput. Geom. Theory Appl.*, 37(2):134–140, July 2007.

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

[QCT12] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. Computing 2D constrained Delaunay triangulation using the GPU. In *I3D '12: Proc. ACM*

SIGGRAPH Symp. Interactive 3D Graphics and Games, pages 39–46, New York, NY, USA, 2012. ACM.

[Qhu12] Qhull. Qhull, computational geometry algorithms library, 2012. `http://www.qhull.org`.

[Rad21] Johann Radon. Mengen konvexer körper, die einen gemeinsamen punkt enthalten. *Mathematische Annalen*, 83(1-2):113–115, 1921.

[Raj91] V. T. Rajan. Optimality of the Delaunay triangulation in $\mathbb{R}^d$. In *SCG '91: Proc. 7th Symp. Computational Geometry*, pages 357–363, New York, NY, USA, 1991. ACM.

[San00] Francisco Santos. A point set whose space of triangulations is disconnected. *Journal of the American Mathematical Society*, 13(3):611–637, 2000.

[San05] Francisco Santos. Non-connected toric hilbert schemes. *Mathematische Annalen*, 332(3):645–665, 2005.

[San06] Francisco Santos. A non-connected graph of triangulations in general position, 2006.

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

[She97] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[She03] Jonathan Richard Shewchuk. Updating and constructing constrained Delaunay and constrained regular triangulations by flips. In *Proceedings of the nineteenth annual symposium on Computational geometry*, SCG '03, pages 181–190, New York, NY, USA, 2003. ACM.

[She05] Jonathan Richard Shewchuk. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In *SCG '05: Proc. 21st ACM Symp. Computational Geometry*, pages 237–246. ACM, 2005.

[SRKN11] Srikanth Srungarapu, Durga Prasad Reddy, Kishore Kothapalli, and P. J. Narayanan. 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.

[Sta12] Stanford. The Stanford 3D scanning repository, 2012. `http://graphics.stanford.edu/data/3Dscanrep/`.

[Str04] Morten Strandberg. Robot path planning: an object-oriented approach. *PhD Thesis, KTH Royal Institute of Technology*, 2004.

[Thr14] Thrust. Thrust, CUDA data parallel primitives library, 2014. `http://http://code.google.com/p/thrust`.

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

[TSBP93] Y. A. Teng, F. Sullivan, I. Beichl, and E. Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 112–121, New York, NY, USA, 1993. ACM.

[TyZTM12] Min Tang, Jie yi Zhao, Ruofeng Tong, and Dinesh Manocha. GPU accelerated convex hull computation. *Computers & Graphics*, 36(5):498 – 506, 2012.

[Wat81] D. F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. In *Comput. J. 24*, pages 167–172, 1981.

[Wen95] R. Wenger. Randomized quick hull. *Algorithmica*, 17, 1995.

[WLYZ+09] Yong Wang, Wu Ling-Yun, Ji-Hong Zhang, Zhong-Wei Zhan, Zhang Xiang-Sun, and Chen Luonan. Evaluating protein similarity from coarse structures. *IEEE/ACM Trans. Computational Biology and Bioinformatics*, 6(4):583–593, 2009.

[Zem09] Michal Zemek. Regular triangulation in 3D and its applications. Technical report, University of West Bohemia, 2009.

List of Publications

[1] Mingcen Gao, Thanh-Tung Cao, Tiow-Seng Tan and Zhiyong Huang, "Flip-flop: convex hull construction via star-shaped polyhedron in 3D", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, (New York, NY), pp.45-54, ACM, 2013.

[2] Mingcen Gao, Thanh-Tung Cao, Ashwin Nanjappa, Tiow-Seng Tan and Zhiyong Huang, "gHull: a GPU algorithm for 3D convex hull", in *ACM Transactions on Mathematical Software*, Volume 40 Issue 1, September 2013.

[3] Thanh-Tung Cao, Ashwin Nanjappa, Mingcen Gao and Tiow-Seng Tan, "A GPU accelerated algorithm for 3D Delaunay triangulation", in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, (New York, NY), ACM, 2014.