Flip-Flop: Convex Hull Construction via Star-Shaped Polyhedron in 3D*

Mingcen Gao Thanh-Tung Cao Tiow-Seng Tan
National University of Singapore

Zhiyong Huang
Institute for Infocomm Research, A*STAR

Abstract

Flipping is a local and efficient operation to construct the convex hull in an incremental fashion. However, it is known that the traditional flip algorithm is not able to compute the convex hull when applied to a polyhedron in $\mathbb{R}^3$. Our novel Flip-Flop algorithm is a variant of the flip algorithm. It overcomes the deficiency of the traditional one to always compute the convex hull of a given star-shaped polyhedron with provable correctness. Applying this to construct convex hull of a point set in $\mathbb{R}^3$, we develop ffHull, a flip algorithm that allows nonrestrictive insertion of many vertices before any flipping of edges. This is unlike the well-known incremental fashion of strictly alternating between inserting a single vertex and flipping. The new approach is not only simpler and more efficient for CPU implementation but also maps well to the massively parallel nature of the modern GPU. As shown in our experiments, ffHull running on the CPU is as fast as the best-known convex hull implementation, qHull. As for the GPU, ffHull also outperforms all known prior work. From this, we further obtain the first known solution to computing the 2D regular triangulation on the GPU.

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

Keywords: Lawson's flip, flipping, regular triangulation, Delaunay triangulation, incremental insertion, GPGPU

1 Introduction

Flipping is a local operation to transform one triangulation to another. In particular, any triangulation of a set of points S in $\mathbb{R}^2$ can be transformed into the Delaunay triangulation (DT) of S by Lawson's *flip algorithm* [1977]. This algorithm repeatedly flips any edge in the triangulation that is not locally Delaunay till none exists, after which the result is the DT. The flipping operation can also be generalized to higher dimensions, but unlike the case in $\mathbb{R}^2$, Joe [1989] shows that in $\mathbb{R}^3$ the flip algorithm can be stuck at a local optimum. At that stage, there still are non-locally Delaunay facets but flipping them creates self-intersection. Edelsbrunner and Shah [1992] demonstrate the same situation when flipping on a regular triangulation (a generalization of the DT with weights on vertices) in $\mathbb{R}^2$. Figure 1(a) shows the example by Edelsbrunner and Shah where the weights of the vertices of the small triangle in the center are set so that the dashed edges are not locally regular, but flipping them creates self-intersection. To avoid being stuck at a local optimum, Joe [1991] proposes to insert points one by one followed by flipping to get to the DT after each insertion. That work is later extended to higher dimensions by Rajan [1991] and Edelsbrunner and Shah [1992].

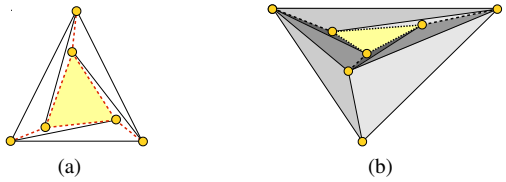

Figure 1: *(a) The stuck configuration when flipping to the 2D regular triangulation, and (b) an analogous configuration in 3D when flipping on a star-shaped polyhedron. The dashed edges need to be flipped, but are all not flippable.*

The flipping operation can also be generalized to flipping on a polyhedron. In fact, by lifting the points in $\mathbb{R}^n$ into $\mathbb{R}^{n+1}$, the problem of constructing the regular triangulation in $\mathbb{R}^n$ becomes the problem of constructing the lower hull in $\mathbb{R}^{n+1}$ [Edelsbrunner and Seidel 1986]. Just for $\mathbb{R}^3$, to transform a polyhedron into the convex hull of its vertices, one repeatedly flips *reflex edges* to increase its volume. If self-intersection needs to be avoided after each flip, it is known that the flip algorithm also does not always terminate at the required convex hull [Aichholzer et al. 2002]. This is true even when the polyhedron is star-shaped; see Figure 1(b). In this example, the upper part of the star-shaped polyhedron is constructed similar to Figure 1(a). The three vertices of the small triangle in the center are pushed down slightly so that all dotted edges are reflex, but are not flippable. On the other hand, if self-intersection is permitted during flipping, it is possible to transform a polyhedron to the convex hull for the restricted case when all vertices are *extreme vertices* that appear on the boundary of the convex hull [Alboul 2003]. In a relevant development, Shewchuk [2005] proposes star splaying, a different kind of local transformation, which is capable of transforming any polyhedron, even with topological error, into the convex hull, but the splaying process is much more complex and less efficient than flipping.

Our work aims to better understand flipping for its simplicity and locality nature having a strong potential for efficient parallel computation. Towards this end, the paper has the following contributions:

- A novel flip algorithm, called Flip-Flop, to provably transform any star-shaped polyhedron to its convex hull.
- An algorithm, called ffHull, to construct the exact 3D convex hull using flipping that works well on both CPUs and GPUs.
- The first known algorithm, called ffRT, to construct the 2D regular triangulation on the GPU.

The Flip-Flop algorithm flips not only reflex edges as in the Lawson's flip algorithm, but also convex ones (hence the "flop" part) when suitable. The salient point of Flip-Flop is its ability to dis-

*The research is supported by the National University of Singapore under grant R-252-000-337-112. Project website: http://www.comp.nus.edu.sg/~tants/flipflop.html. Emails: {mingcen | caothanh | tants}@comp.nus.edu.sg, zyhuang@i2r.a-star.edu.sg.

The 2013 ACM Symposium on Interactive 3D Graphics and Games, 21-23 Mar, Orlando, Florida, USA, pp 45--54.

cover automatically all the *non-extreme vertices* that do not appear on the boundary of the convex hull and to remove them, both using the local flipping operation. Though working for only star-shaped polyhedra, Flip-Flop is used as the key component for ffHull to construct the convex hull of a point set in $\mathbb{R}^3$ and for ffRT to construct the regular triangulation of a point set in $\mathbb{R}^2$, with provable correctness. Our algorithm allows nonrestrictive insertion of many vertices, both extreme and non-extreme, before any flipping of edges. Besides, by enforcing an invariant that the polyhedron is always star-shaped after each flip, which is achieved using only local checks, we ensure no self-intersection checks are needed. These not only maintain the simplicity and locality of the flip algorithm, but also map well to the massively parallel nature of the modern GPUs. Our extensive experiments show that ffHull and ffRT running on the CPU are as fast as the best known implementations for computing convex hull and regular triangulation. As for the GPU, with an implementation in CUDA, ffHull and ffRT demonstrate their suitability for parallel computation by outperforming all known prior work, while remaining robust and exact.

The rest of the paper is organized as follows. Section 2 presents the Flip-Flop algorithm in $\mathbb{R}^3$. Section 3 introduces ffHull and Section 4 extends it to ffRT. Section 5 gives the experiment results of our algorithms on both the CPU and the GPU. Finally, Section 6 concludes the paper with some potential directions for future work. The proof of correctness of the Flip-Flop algorithm and thus of ffHull and ffRT is provided in Appendix A.

2 Flip-Flop Algorithm

We present the Flip-Flop algorithm that takes a star-shaped polyhedron in $\mathbb{R}^3$ as input and produces the convex hull of the points of the polyhedron.

2.1 Definitions

Let S be a set of points in $\mathbb{R}^3$, assuming in general positions. The *convex hull* of S, denoted as $\mathcal{CH}(S)$, is the collection of points where each is a convex combination of the points in S. This collection is also the smallest convex set containing S, and the boundary of $\mathcal{CH}(S)$ is thus a convex polyhedron. Similarly, the convex hull for a set R of rays, denoted as $\mathcal{CH}(R)$, is the collection of points where each is a convex combination of some points on the rays of R. $\mathcal{CH}(R)$ extends to infinity.

Given a point $s \in \mathbb{R}^3$ and three points $a, b, c \in S$, the *cone* of triangle abc w.r.t. s, denoted as $\mathcal{C}_s(\triangle abc)$, is the convex hull of the three rays $\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}\}$. Two cones *overlap* if they contain some common points not on their boundaries. Let $\mathcal{T}$ be a polyhedron with vertices in S and with faces that are triangles. We say $\mathcal{T}$ is a *star-shaped* polyhedron w.r.t. s if and only if s is in the interior of $\mathcal{T}$ and the cones of any two triangles of $\mathcal{T}$ w.r.t. s do not overlap each other. This means the boundary of a star-shaped polyhedron is entirely visible from s. s is called the *kernel* of $\mathcal{T}$.

In $\mathcal{T}$, given an edge $e = \overline{ab}$ with endpoints a and b, e is incident to two triangles $t_1 = abc$ and $t_2 = bad$. $\{c, d\}$ is called the *link* of e. The two sides of a triangle are the two half-spaces defined by the plane containing the triangle. e is a *reflex* edge (w.r.t. s) if c and s lie on different sides of t_2; otherwise it is a *convex* edge. Indeed, since $\mathcal{T}$ is star-shaped, c and s lie on two different sides of t_2 if and only if d and s lie on two different sides of t_1. Convexity and reflexivity are local properties of edges of $\mathcal{T}$. On the other hand, if $\mathcal{T}$ has no reflex edges, then it is the convex hull of its vertices.

Since $\mathcal{T}$ is topologically a 2D triangulation, flips on $\mathcal{T}$ can be classified into *3-1 flips*, *2-2 flips* and *1-3 flips* [Edelsbrunner and Shah 1992]; see Figure 2. A 3-1 flip removes a vertex from $\mathcal{T}$, while a 1-3 flip inserts a new vertex into $\mathcal{T}$. For the discussion of Flip-Flop, we do not concern with 1-3 flips. Given an edge $e = \overline{ab} \in \mathcal{T}$ with $\{c, d\}$ being its link, the *induced subcomplex* of e, denoted as σ_e, is the set of triangles in $\mathcal{T}$ that span $\{a, b, c, d\}$. Clearly, flipping e is a *2-2 flip* if $|\sigma_e| = 2$. We call e in this case a *2-2 edge*. Similarly, e is a *3-1 edge* if $|\sigma_e| = 3$. We say that e is *flippable* w.r.t. s if firstly s is outside the tetrahedron $abcd$, and secondly the union of the cones of the triangles of σ_e is equal to $\mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$. Otherwise, e is *unflippable*. If e is a 3-1 edge, the second condition is always true, while if e is a 2-2 edge, the second condition is true if and only if any vertex among $\{a, b, c, d\}$ is outside the cone of the triangle formed by the other three vertices. If a 2-2 edge is flippable, we also call it *2-2 flippable* for short. Similarly, a 3-1 edge that is flippable is called *3-1 flippable*.

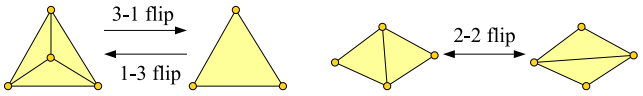

Figure 2: *Flipping on a 3D polyhedron.*

Flipping a flippable edge that is reflex turns it into a convex one, and vice versa. Moreover, the definition ensures that flipping a flippable edge of a star-shaped polyhedron w.r.t. s results in a polyhedron that remains star-shaped w.r.t. s. In another view, no intersections are introduced during such flips, and the check for the flippability of an edge is purely local. As such, flipping can be used as a local operation to transform a star-shaped polyhedron into the convex hull of its vertices.

2.2 Flipping Criteria

Traditionally, we only flip a flippable edge e if it is reflex. This flip increases the volume of $\mathcal{T}$, and we refer to this flipping criterion as the *V-criterion* (V for Volume). Clearly, the convex hull has the maximum volume possible, and thus this flip algorithm is a hill climbing method. It works when all vertices in $\mathcal{T}$ are extreme vertices, but possibly gets stuck at a local optimum when not all vertices are extreme vertices, as shown in Figure 1(b).

We can show that any 3-1 edge that is reflex is always flippable. On the other hand, any 2-2 edge that is reflex is flippable; otherwise it has a non-extreme vertex as its endpoint. If all these non-extreme vertices are removed, we would be able to flip $\mathcal{T}$ to its convex hull. A direct way to remove a vertex is to delete all triangles incident to it and re-triangulate the resulting hole. This is difficult to do while still keeping $\mathcal{T}$ star-shaped. Moreover, it is no longer a local operation when we try to avoid self-intersection. Instead, we observe that to remove a vertex, one can attempt to use 2-2 flips to first reduce its degree to 3; see Figure 3. Then, a 3-1 flip can be applied to complete the vertex removal. A relevant approach has been proposed by Ledoux et al. [2005], but the input needs to be a DT.

Based on the above mentioned observation, we introduce a new criterion for our flip algorithm. We flip a flippable edge as long as

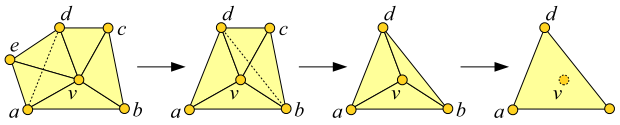

Figure 3: *The star of v as seen from the kernel point. To remove v, we successively flip $\overline{ve}$ and $\overline{vc}$ to reduce the degree of v to 3. Then, v is removed by a 3-1 flip.*

it is incident to a non-extreme vertex. This decreases the degree of a non-extreme vertex in $\mathcal{T}$. However, this might lead to flipping an edge back and forth if multiple vertices are labeled as non-extreme. To avoid this, e is only flipped if that decreases the degree of the vertex with the smallest index among all non-extreme vertices in $\{a, b, c, d\}$. We refer to this as the *D-criterion* (D for Degree).

2.3 Flip with V- and D-criterion

The Flip-Flop algorithm is designed based on both the V-criterion and the D-criterion. We follow the D-criterion to remove non-extreme vertices while at the same time we use the V-criterion to move closer to the convex hull. In order to apply the D-criterion, we analyze 2-2 unflippable edges to label non-extreme vertices; see Figure 4(a). Algorithm 1 shows the detailed pseudocode of Flip-Flop. The input of the algorithm is a star-shaped polyhedron $\mathcal{T}$ w.r.t. a given point s. The output is $\mathcal{CH}(\mathcal{T})$, i.e. the convex hull of all vertices of $\mathcal{T}$. The algorithm repeatedly checks the edges in $\mathcal{T}$ and flips them if necessary. For efficiency, we use a queue $\mathcal{Q}$ to keep track of edges that need to be checked. An edge that has been checked need not be checked again unless its incident triangles are changed or one of its adjacent vertices is labeled as non-extreme. Each time we flip an edge, we push all the newly created edges into $\mathcal{Q}$. Note that an edge in $\mathcal{Q}$ is attached to its two incident triangles, and thus is considered *outdated* if any one of these triangles are removed from $\mathcal{T}$ due to some flips.

We now discuss Algorithm 1 in details. First of all, all the vertices of $\mathcal{T}$ are labeled as unknown (Line 1) to indicate that we do not know whether they are non-extreme vertices or not. All edges of $\mathcal{T}$ are pushed into $\mathcal{Q}$ for checking (Line 2). The loop in Line 3–20 repeats until no more edges in $\mathcal{T}$ need to be checked, by which $\mathcal{T}$ has no reflex edges and thus the algorithm reaches the required output. Let $e = \overline{ab}$ be an edge popped in one iteration, and let $\{c, d\}$ be its link. If e is outdated, we can just ignore it (Line 5).

First, consider the situation that e is a 3-1 edge (Line 7–10). Either a or b must lie inside the cone of the triangle formed by the other 3 vertices; see Figure 4(b). Without loss of generality, let a be inside $\mathcal{C}_s(\triangle bcd)$. If a is labeled as non-extreme or e is reflex (which also implies that a is non-extreme), we flip e to remove a. In both cases, e is flippable (see Appendix A).

Second, consider the situation that e is a 2-2 edge (Line 11–20). There are two cases, depending on whether the union of the cones of the triangles of σ_e is equal to $\mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$ or not (Line 12). If this condition is true, then e fulfills the second condition of being flippable. We flip e either if e is reflex and no vertices among $\{a, b, c, d\}$ are labeled as non-extreme vertices (Line 13–14), or if s is outside the tetrahedron $abcd$ and a or b is the non-extreme vertex with smallest index among all non-extreme vertices labeled in $\{a, b, c, d\}$ (Line 15-16). This is a combination of the D-criterion and the V-criterion, with priority given to the D-criterion to reduce the degree of non-extreme vertices and eventually remove them. On the other hand, if the condition in Line 12 is false, then e is unflippable. In this case, if e is reflex, without loss of generality we assume a is inside $\mathcal{C}_s(\triangle bcd)$, then a is a non-extreme vertex. We label it in Line 19 so that it is removed in subsequent flips using the D-criterion. Also, all the edges incident to a need to be checked again due to the labeling of a, so we push them into $\mathcal{Q}$.

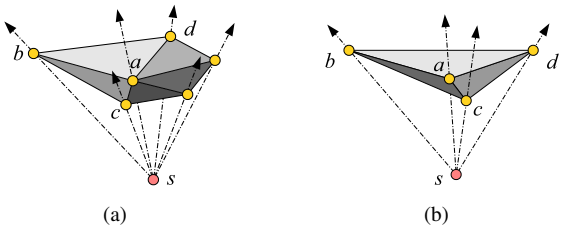

Figure 4: *(a) $\overline{ab}$ is a 2-2 edge that is reflex and unflippable. (b) $\overline{ab}$ is a 3-1 edge that is reflex. In both cases, a is a non-extreme vertex since it lies inside the tetrahedron $sbcd$.*

Algorithm 1 Flip-Flop

Input: a star-shaped polyhedron $\mathcal{T}$ w.r.t. s
Output: $\mathcal{CH}(\mathcal{T})$

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

Note that the algorithm presented above is just one approach of using the V-criterion together with the D-criterion. In fact the flips can vary, and a stack, for example, can be used to replace the queue $\mathcal{Q}$. Another variant, referred to as the *V-biased* Flip-Flop, is to keep performing the flipping according to the V-criterion first, until no more flipping can be done. After that, we can start labeling the non-extreme vertices and use flipping due to the D-criterion to remove them. The process is then repeated until all the non-extreme vertices are identified and removed.

3 Convex Hull Construction

Convex hull is a very fundamental problem in computational geometry, and has been well studied for many years [Preparata and Hong 1977; Clarkson and Shor 1988; Chan 1996]. The most practical algorithm for the CPU is QuickHull [Barber et al. 1996], a variant of the incremental insertion approach where the outermost point is inserted in each round.

There are also several solutions to construct the convex hull with the help of the GPU. For the simple 2D case, efficient GPU algorithms are proposed by Jurkiewicz and Danilewski [2011] and Srungarapu et al. [2011]. For 3D and higher dimensions, Tang et al. [2012] adapt the incremental insertion approach of QuickHull on the GPU to construct a filter to cull away most of the interior points, while sending the rest to a CPU algorithm to construct the convex hull. This approach is not very useful when there are many extreme vertices. Tzeng and Owens [2012] propose CudaQuickHull and Stein et al. [2012] present CudaHull, both of which apply the idea of QuickHull on the GPU. However, their approaches either overlook the difficulty that the points inserted in parallel might

not be extreme vertices, or use the traditional flipping algorithm which cannot always produce the convex hull. More specifically, in the correctness proof for CudaHull, the authors claim that the algorithm flips all *concave edges*, while in fact their algorithm does not flip edges that lead to self-intersections. As a result, in some cases the output polyhedron is not a convex hull because some concave edges remain. Recently, Gao et al. [2012] propose the *gHull* algorithm, using the digital Voronoi diagram to construct an approximation followed by several other fixing steps including the use of star splaying [Shewchuk 2005] on the GPU. Due to the use of an approximation from the digital space, their algorithm cannot handle point sets from a non-uniform distribution very efficiently.

The Flip-Flop algorithm is a powerful tool to transform any star-shaped polyhedron into its convex hull. We introduce a new algorithm, termed ffHull, that uses Flip-Flop to construct the convex hull of a set of point S in $\mathbb{R}^3$. The main idea is to quickly construct a star-shaped polyhedron $\mathcal{T}$ that is close to $\mathcal{CH}(S)$ by an incremental insertion approach, followed by applying Flip-Flop.

3.1 Algorithm

Algorithm 2 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. $S \leftarrow S \setminus \{a, b, c, d\}$
2: $\mathcal{T} \leftarrow \mathcal{CH}(\{a, b, c, d\})$; $s \leftarrow$ the centroid of $\mathcal{T}$
3: associate each $p \in S$ to $\triangle abc \in \mathcal{T}$ s.t. $p \in \mathcal{C}_s(\triangle abc)$
4: **while** $\neg S.isEmpty()$ **do**
5: **for each** $\triangle abc$ associated by some points **do**
6: let v be the furthest point associated to $\triangle abc$
7: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\triangle vab, \triangle vbc, \triangle vca\} \setminus \{\triangle abc\}$; $S \leftarrow S \setminus \{v\}$
8: **for each** $p \in S$ **do**
9: let p be associated to $\triangle abc$, into which v is just inserted
10: associate p to $t \in \{\triangle vab, \triangle vbc, \triangle vca\}$ s.t. $p \in \mathcal{C}_s(t)$
11: **if** p and s lie on the same side of t **then** $S \leftarrow S \setminus \{p\}$
12: apply Flip-Flop on $\mathcal{T}$

Algorithm 2 shows the pseudocode of ffHull. There are two stages: constructing a star-shaped polyhedron (Line 1–11), and transforming it to the convex hull using Flip-Flop (Line 12). In the first stage, from the initial star-shaped polyhedron $\mathcal{T}$ with 4 extreme vertices (Line 1), we incrementally process input points: an input point is either inserted to become a vertex of the polyhedron or removed if found to be inside the polyhedron constructed so far. Specifically, during the process, we use the centroid s of the initial star-shaped polyhedron as the kernel point (Line 2). For each point p in S, we *associate* p with a unique $\triangle abc$ of $\mathcal{T}$ if p is inside the cone of $\triangle abc$. 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 subsequent loop, $\mathcal{T}$ is grown by inserting the furthest point v associated to each triangle $t \in \mathcal{T}$ into $\mathcal{T}$ (Line 5–7). This is done by replacing t with three new triangles. Each point inserted into $\mathcal{T}$ is removed from S. Each insertion splits a cone $\mathcal{C}_s(t)$ into three new disjoint cones. This guarantees that $\mathcal{T}$ is still star-shaped after the insertion. Line 8–11 update the triangle each point $p \in S$ is associated to, or remove p if it is a non-extreme point. This "growing" process is repeated until S is empty. In the second stage, we simply apply the Flip-Flop algorithm to transform $\mathcal{T}$ to $\mathcal{CH}(S)$.

We have two notes for the above algorithm. Firstly, Flip-Flop actually works for any star-shaped polyhedron. It is thus not necessary to always find the furthest point to insert into $\mathcal{T}$, even though doing so helps to remove non-extreme points in S quickly. 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. Secondly, 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. That is, however, observed to have lower parallelism during flipping in our experiments.

3.2 GPU Implementation

We use CUDA for our implementation of ffHull on the GPU. 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. This is similar to the data structure of triangulation used in CGAL [Boissonnat et al. 2002] and GPU-DT [Qi et al. 2012]. 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 at the same time 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 11 of Algorithm 2, we only need to compute $orient(p, t)$. Similarly, we can check the reflexivity of an edge, without using s. The second technique is to reuse $|orient(p, t)|$, which is the volume of the tetrahedron formed by p and the triangle t, when finding the furthest point to t instead of actually computing the distance.

The details on the implementation of ffHull is as follows. In the first stage, constructing a star-shaped polyhedron, we have four major CUDA kernels. The first kernel (Line 3 of Algorithm 2), with one thread processing one input point p, finds the triangle in the initial triangulation $\mathcal{T}$ 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 processing and 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 5–7), 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 8–11), with one thread processing one point p, updates p's associated triangle if p is still outside $\mathcal{T}$. 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 now associated to. 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. During this stage, the index of the associated triangle of a point is set to -1 if the point is found to be inside the polyhedron, and that point will not be revisited in the subsequent computation.

In the second stage, the Flip-Flop stage, the flipping is done in multiple iterations. In each iteration, we use two kernels, a checking kernel and a flipping kernel, to perform multiple flips in parallel, similar to the technique described in [Qi et al. 2012] and [Navarro et al. 2011]. In the checking kernel, we assign one thread to one triangle to check if one of its edges can be flipped (based on the V- and D- criterion). The difficulty is that the induced sub-complexes of some edges share some triangles, thus the flips are conflicting 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 σ_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 σ_e are still labeled with the index of $\triangle abc$. This guarantees no conflicting flips are performed in the same iteration, and in each iteration at least one flip can be done. Note that only up to three threads write to the same memory location during the labelling, so the use of the atomic operation does not affect the efficiency much.

3.3 Exact Computation and Robustness

The only predicate we use, the 3D orientation predicate, is adapted from the exact predicate of Shewchuk [1997]. Instead of using a multiple-stages adaptive arithmetic, 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 of the determinant being computed or not, we use the forward error analysis approach described by Shewchuk. The error bounds are pre-computed using a single CUDA kernel, and stored for later used by all kernels that need exact computation.

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.

We also use the SoS technique of Edelsbrunner and Mucke [1990] to handle degeneracy. The implementation is as described in their original paper. All the computation of the subdeterminants are done with exact arithmetics. One small note is that although SoS requires computing many different subdeterminants, the CUDA code is written in such a way that the call to exact predicate appears only once, inside a loop with different subdeterminants being passed to it. Consequently, we can force inline the exact predicate function without worrying about the kernel code being too large to compile.

With the use of exact predicates and SoS, we can guarantee the exactness and robustness of our implementation. The only source of inexact computation is when computing the kernel point s. If the initial tetrahedron is almost flat, its centroid computed inexactly can lie outside. 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.

4 Extension to Regular Triangulation

The regular triangulation of a set S of weighted points in $\mathbb{R}^2$ is often computed using the incremental insertion approach by Edelsbrunner and Shah [1992]. On the other hand, the regular triangulation is closely related to the convex hull of S when lifted into $\mathbb{R}^3$. The lift map is $\pi : (x, y) \mapsto (x, y, x^2 + y^2 - w)$ where w is the weight of the point. Let the lifted point set be S'. The lower hull of the convex hull of S' is equivalent to the regular triangulation of S. Existing GPU algorithms to construct convex hull are not suitable for constructing the regular triangulation for several reasons. First, most of the points are expected to appear in the final triangulation, so the filtering method presented by Tang et al. [2012] is not very useful. Second, the range of the z-coordinates is very large compared to that of the x- and y-coordinates, thus using the digital Voronoi diagram as proposed by Gao et al. [2012] is not very suitable.

The Flip-Flop algorithm does not have any of the disadvantages mentioned above. Thus we extend ffHull to ffRT, a new algorithm to compute the regular triangulation of a set of weighted points in $\mathbb{R}^2$. Conceptually, we lift the input point set S into $\mathbb{R}^3$ to get S', simulate ffHull on S' to get $\mathcal{CH}(S')$ and finally return its lower hull as output. The actual implementation of ffRT is adapted from that of ffHull with some modification to improve the efficiency. We initialize the star-shaped polyhedron $\mathcal{T}$ with a tetrahedron containing 3 extreme vertices $\{u, v, w\}$ of S' and a virtual vertex $\hat{p}$ with the z-coordinate being $+\infty$, and thus the kernel point $\hat{s}$ is the centroid of $\{u, v, w\}$ but with the z-coordinate being $+\infty$. This makes all upper hull triangles being incident to $\hat{p}$ and thus can easily be removed before we output the triangulation.

During the growing process, there are two kinds of triangles in $\mathcal{T}$: *virtual triangles* that are incident to $\hat{p}$ and *real triangles* that are not. The cone of a real triangle abc is the space bounded by the three vertical planes going through the edges $\overline{ab}$, $\overline{bc}$, and $\overline{ca}$. The cone of a virtual triangle $ab\hat{p}$ can be seen as the space bounded by the vertical planes going through $a\hat{s}$ and $b\hat{s}$ intersecting with the half space defined by the vertical plane going through $\overline{ab}$ that does not contain $\hat{s}$. Similarly, in the flipping stage there are three kinds of edges: *internal edges* that are incident to two real triangles; *virtual edges* to two virtual triangles; and *silhouette edges* to one real triangle and one virtual triangle. A virtual edge is always flippable, and checking its reflexivity is done using a 2D oriengation check. A silhouette edge is always convex, while an internal edge is treated as a normal edge in ffHull.

5 Experimental Results

Our algorithms are implemented on the GPU using the CUDA programming model by Nvidia, and can easily be ported to OpenCL. 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. All implementations are compiled with all optimizations enabled.

5.1 Convex Hull

We compare the performance of our implementations of the ffHull algorithm on both the CPU and the GPU with the two fastest convex hull implementations on the CPU, qHull and CGAL, as well as with gHull, the recently published GPU algorithm by Gao et al. [2012]. Both CGAL and gHull use exact predicates, similar to our implementations, while qHull produces a convex hull with "thick" facets. According to our experiment results, qHull runs faster than CGAL in most cases, so all the speedup reported in this subsection is with respect to qHull.

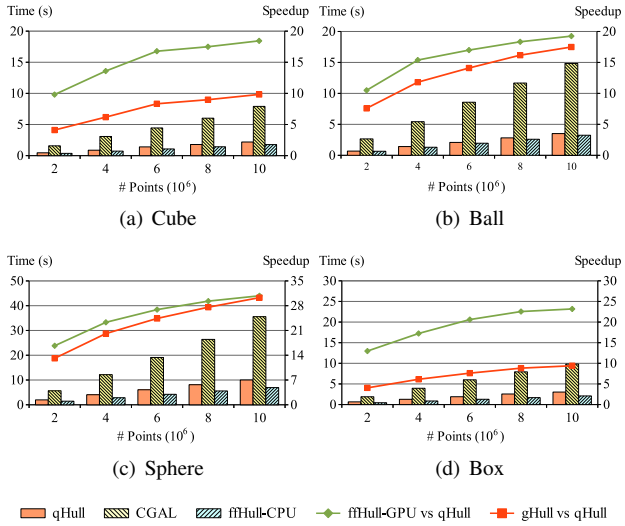

Figure 5: *Running time of ffHull compared to qHull, CGAL (on the CPU) and gHull (on the GPU).*

Running time. First, we show an experiment on some representative point distributions. We generate points randomly with coordinates between [0.0, 1.0]. Following Gao et al.'s work, we use four representative distributions of points to test: a cube, a ball of radius of 0.5, a sphere of thickness of 0.01 and a box with thickness of 0.01. The cube and the box distributions have very few points on the convex hull, while the ball and the sphere distributions have many. The box and the sphere distributions have most points close to the convex hull, making it difficult for the Quickhull algorithm to remove non-extreme points.

Figure 5 shows the running time of the CPU algorithms and the speedup of the GPU algorithms when compared to qHull. Clearly, our CPU implementation of ffHull outperforms CGAL in all the tested distributions. Our CPU version of ffHull is also as fast as qHull, while always producing an exact convex hull rather than just an approximation like qHull does. On the other hand, our GPU implementation of ffHull achieves up to 30x speedup compared to qHull, and is faster than gHull in all the cases. As mentioned by Gao et al. [2012], their algorithm does not perform well in the cube and the box distributions due to the use of approximation in the digital space. Our ffHull is not affected by this, and thus runs much faster than gHull for these cases.

We also use models of over a million points from the Stanford 3D scanning repository to test the ffHull algorithm; see Table 1. These models have very few points on the convex hull, while most other points are distributed near the surface, with many points being coplanar. Once again, our ffHull implementation on the GPU outperforms gHull in all cases. The CPU implementation, though slower than qHull due to a large amount of exact computation required, still outperforms CGAL.

Model	# Points (millions)	Running time (ms)				
		qHull	CGAL	ffHull-CPU	gHull	ffHull-GPU
Asian dragon	3.6	540	1181	997	214	141
Thai statue	5.0	692	1538	1240	180	125
Lucy	13.9	1884	4488	3664	297	263

Table 1: *Running time on different 3D models.*

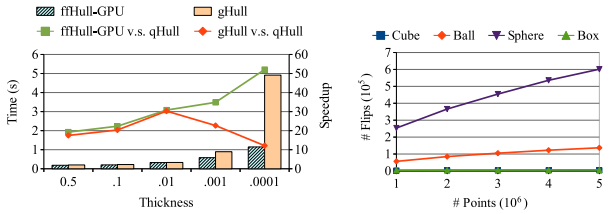

Figure 6: *(a) ffHull vs gHull on the GPU with points on a thin sphere of different thicknesses. (b) The number of flips performed by ffHull on the CPU.*

Sensitivity. The number of points on the convex hull directly affects the running time of all convex hull algorithms. The Flip-Flop algorithm uses only simple and local operations, thus it works very well in parallel, especially when there are many vertices on the input polyhedron. As such, the ffHull algorithm which uses Flip-Flop is also very efficient when many input points are on the convex hull. Figure 6(a) shows our experiment results with the sphere distribution of various thickness, ranging from 0.5 (which is the ball distribution) to 10^{-4}. The total number of points is 10^7. The thinner the sphere is, the more points are on the convex hull, and we see that the speedup of ffHull on the GPU compared to qHull increases sharply from 20x to 50x. In contrast, gHull slows down significantly when the sphere gets thinner due to the use of digital approximation.

Time breakdown. We also measure the running time of the two stages of the algorithm on the GPU separately: the time taken to construct a star-shaped polyhedron and the time taken to transform it to the convex hull. The same experiment setting as above is used, with 10^7 points on the sphere distribution of various thickness. As more points are on the convex hull, both stages of the algorithm take more time, but the ratio between the two also changes. At thickness 0.5, the second stage only takes about 45% as much time as the first stage. However, at thickness 10^{-4}, this ratio increases to nearly 100%. This is because the number of points on the convex hull affects the first stage at a logarithmic rate (i.e. only affects the number of loops) while it affects the second stage at a linear rate in our experiment.

Number of flips. In Figure 6(b), we present the total number of flips performed by ffHull on the CPU when running on different distributions with varying number of points. For the cube and the 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 Lawson's traditional flip algorithm when computing the 2D DT in practice.

5.2 Regular Triangulation

We compare the implementation of the ffRT algorithm on both the CPU and the GPU with CGAL. We randomly generate points in uniform distribution with coordinates between [0.0, 1.0]. In Figure 7(a), the weights of the points are randomly chosen from 0.0 to 10^{-5}, so that the number of points in the resulting regular triangulation is approximately 20% of the number of input points. In Figure 7(b) the weight range is $[0.0, 2 \times 10^{-7}]$, and thus approximately 99% of the points appear in the output. In both cases, our CPU implementation achieves a performance close to CGAL, while the GPU implementation gives an 8x speedup on average. Note that

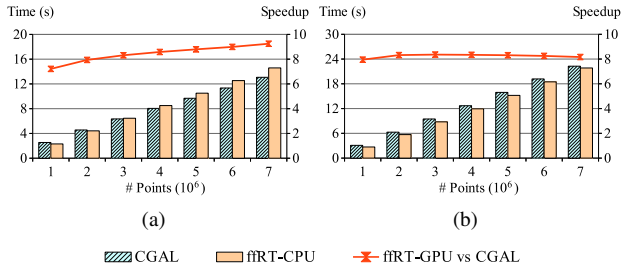

Figure 7: *Performance comparison between CGAL and ffRT on the CPU and the GPU where the number of points in the output is (a) 20% and (b) 99% of the number of input points.*

in both cases, the number of points that can be eliminated is not as high as in the convex hull problem, as such the Flip-Flop algorithm has a lot of work to do. Nevertheless, it can robustly produce the correct result.

6 Concluding Remarks

This paper presents a novel algorithm, Flip-Flop, to construct the convex hull from any 3D star-shaped polyhedron. There are several benefits when using Flip-Flop. First, it is provably correct, and thus can always produce the exact convex hull (or regular triangulation). Second, the algorithm uses only the simple flipping operation, with all computations being purely local. Third, Flip-Flop works with any star-shaped polyhedron, thus during the convex hull (or regular triangulation) construction, many points can be inserted at the same time instead of one at a time in the previous approaches. Last but not least, the flips can be performed in any order as long as they obey the V-criterion and the D-criterion. With all these benefits, it is very suitable for parallel computation. The implementations of ffHull and ffRT, which use Flip-Flop to construct the convex hull in $\mathbb{R}^3$ and the regular triangulation in $\mathbb{R}^2$, perform well on the CPU compared to other popular implementations of existing algorithms. Our implementations on the GPU give significant speedup over the sequential implementations, and also outperform other existing GPU solutions.

References

AICHHOLZER, O., ALBOUL, L., AND HURTADO, F. 2002. On flips in polyhedral surfaces. *Int. J. Foundations of Computer Science 13*, 2, 303–311.

ALBOUL, L. 2003. Optimising triangulated polyhedral surfaces with self-intersections. In *Mathematics of Surfaces*, M. Wilson and R. Martin, Eds., vol. 2768 of *Lecture Notes in Computer Science*. 48–72.

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The Quickhull algorithm for convex hulls. *ACM Trans. Mathematical Software 22*, 4, 469–483.

BOISSONNAT, J.-D., DEVILLERS, O., PION, S., TEILLAUD, M., AND YVINEC, M. 2002. Triangulations in CGAL. *Computational Geometry 22*, 1–3, 5–19.

CHAN, T. M. 1996. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry 16*, 4, 361–368.

CLARKSON, K. L., AND SHOR, P. W. 1988. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *SCG '88: Proc. 4th Symp. Computational Geometry*, ACM, New York, NY, USA, 12–17.

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

EDELSBRUNNER, H., AND SEIDEL, R. 1986. Voronoi diagrams and arrangements. *Discrete and Computational Geometry 1*, 1, 25–44.

EDELSBRUNNER, H., AND SHAH, N. R. 1992. Incremental topological flipping works for regular triangulations. In *SCG '92: Proc. 8th Symp. Computational Geometry*, ACM, New York, NY, USA, 43–52.

GAO, M., CAO, T.-T., NANJAPPA, A., TAN, T.-S., AND HUANG, Z. 2012. A GPU algorithm for convex hull. `http://www.comp.nus.edu.sg/~tants/gHull.html`.

JOE, B. 1989. Three-dimensional triangulations from local transformations. *SIAM J. Scientific and Statistical Computing 10*, 4 (July), 718–741.

JOE, B. 1991. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer Aided Geometric Design 8*, 2 (May), 123–142.

JURKIEWICZ, T., AND DANILEWSKI, P. 2011. Efficient quicksort and 2D convex hull for CUDA, and MSIMD as a realistic model of massively parallel computations. Manuscript.

LAWSON, C. L. 1977. Software for C^1 surface interpolation. In *Mathematical Software III*, Academic Press, New York, J. R. Rice, Ed., 161–194.

LEDOUX, H., GOLD, C. M., AND BACIU, G. 2005. Flipping to robustly delete a vertex in a Delaunay tetrahedralization. In *ICCSA '05: Proc. Int. Conf. Computational Science and its Applications*, Springer-Verlag, Berlin, Heidelberg, 737–747.

NAVARRO, C., HITSCHFELD, N., AND SCHEIHING, E. 2011. A parallel gpu-based algorithm for delaunay edge-flips. In *EuroCG '11: 27th European Workshop on Computational Geometry*, I. M. Hoffmann, Ed., 75–78.

PREPARATA, F. P., AND HONG, S. J. 1977. Convex hulls of finite sets of points in two and three dimensions. *Communication of ACM 20*, 2, 87–93.

QI, M., CAO, T.-T., AND TAN, T.-S. 2012. Computing 2D constrained Delaunay triangulation using the GPU. In *I3D '12: Proc. ACM Symp. Interactive 3D Graphics and Games*, ACM, New York, NY, USA, 39–46.

RAJAN, V. T. 1991. Optimality of the Delaunay triangulation in $\mathbb{R}^d$. In *SCG '91: Proc. 7th Symp. Computational Geometry*, ACM, New York, NY, USA, 357–363.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry 18*, 3, 305–363.

SHEWCHUK, J. R. 2005. Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls. In *SCG '05: Proc. 21st ACM Symp. Computational Geometry*, ACM, 237–246.

SRUNGARAPU, S., REDDY, D. P., KOTHAPALLI, K., AND NARAYANAN, P. J. 2011. Fast two dimensional convex hull on the GPU. In *WAINA '11: Proc. IEEE Workshops of International Conference on Advanced Information Networking and*

Applications, IEEE Computer Society, Washington, DC, USA, 7–12.

STEIN, A., GEVA, E., AND EL-SANA, J. 2012. CudaHull: Fast parallel 3D convex hull on the GPU. *Computers & Graphics 36*, 4, 265–271.

TANG, M., ZHAO, J., TONG, R., AND MANOCHA, D. 2012. GPU accelerated convex hull computation. *Computers & Graphics 36*, 5, 498–506.

TZENG, S., AND OWENS, J. D. 2012. Finding convex hulls using Quickhull on the GPU. *CoRR abs/1201.2936*.

A Proof of Correctness

We show that the Flip-Flop algorithm can successfully transform any star-shaped polyhedron in $\mathbb{R}^3$ into its convex hull. It then follows that the ffHull algorithm computes correctly the convex hull of a point set. In the following discussion, let $\mathcal{T}$ be a star-shaped polyhedron in $\mathbb{R}^3$ with respect to a given kernel point s. Let $F(v)$ denote the set of triangles incident to the vertex v of $\mathcal{T}$, and $L(v)$ be the set of vertices and edges in $F(v)$ excluding those incident to v. $F(v)$ and $L(v)$ are called the *fan* and the *link* of v respectively. Given an edge $e = \overline{ab} \in \mathcal{T}$ with $\{c, d\}$ being its link, the *induced subcomplex* of e, denoted as σ_e, is the set of triangles in $\mathcal{T}$ that span $\{a, b, c, d\}$. We use $\mathcal{C}_s(F(v))$ to denote $\bigcup_{t \in F(v)} \mathcal{C}_s(t)$, and similarly for $\mathcal{C}_s(\sigma_e)$.

A.1 Properties of Star-Shaped Polyhedra

Lemma 1. *For any vertex v of $\mathcal{T}$, none of the vertices of $\mathcal{T} \setminus L(v)$ except v lies inside $\mathcal{C}_s(F(v))$. Moreover, v does not lie on the boundary of $\mathcal{C}_s(F(v))$.*

Proof. This follows immediately from the definition of the star-shaped polyhedron. □

Lemma 2. *Any 3-1 edge $e = \overline{ab}$ of $\mathcal{T}$ that is reflex is flippable.*

Proof. Let the link of e be $\{c, d\}$. Without loss of generality, let $\sigma_e = \{\triangle abc, \triangle bad, \triangle acd\}$; see Figure 4(b). Since e is reflex, s and d lie on different sides of $\triangle abc$, so s is outside the tetrahedron $abcd$. On the other hand, by Lemma 1, a lies inside $\mathcal{C}_s(\sigma_e)$, and thus inside $\mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sd}\})$. Therefore, e is flippable. □

Lemma 3. *Any 2-2 unflippable edge $e = \overline{ab}$ that is reflex is incident to a non-extreme vertex.*

Proof. Let the link of e be $\{c, d\}$. 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 4(a). This implies that either a is inside $\mathcal{C}_s(\triangle bcd)$ or b is inside $\mathcal{C}_s(\triangle acd)$, which results in a non-extreme vertex since e is reflex. □

A.2 Star-Shaped Polyhedra with all Extreme Vertices

We prove that any star-shaped polyhedron with all vertices being extreme vertices can be transformed into its convex hull by flippings with the V-criterion. Using the invariant that $\mathcal{T}$ is still star-shaped w.r.t. s after flipping any flippable edge, we show that in this case, any reflex edge is in fact flippable, which implies that we can flip all the reflex edges of $\mathcal{T}$ to get to its convex hull.

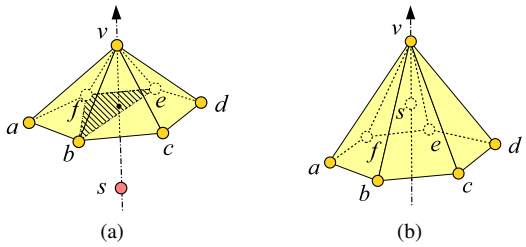

Figure 8: *(a) v is locally covered by b, e and f on v's link. (b) v is not locally covered because no cone defined by three points on v's link contains it.*

Lemma 4. *If all vertices of $\mathcal{T}$ are extreme vertices, any reflex edge e of $\mathcal{T}$ is 2-2 flippable.*

Proof. The edge e cannot be a 3-1 edge; otherwise by the same argument in the proof of Lemma 3, we can show that there is a non-extreme vertex in $\mathcal{T}$. On the other hand, if e is 2-2 unflippable, by Lemma 3 we also have a non-extreme vertex in $\mathcal{T}$. Therefore, e is 2-2 flippable. □

Theorem 1. *Flipping according to the V-criterion can transform $\mathcal{T}$ into its convex hull if all the vertices of $\mathcal{T}$ are extreme vertices.*

Proof. Any reflex edge in $\mathcal{T}$ is flippable and thus is flipped according to the V-criterion. Flipping a reflex edge e of $\mathcal{T}$ increases its volume, and any edge that becomes reflex after the flip is also subsequently flipped. Since $\mathcal{T}$ is free from self-intersection, its volume is bounded, thus the flipping process always terminates with a star-shaped polyhedron with no reflex edges. Since no vertices are removed, the result is the convex hull of the vertices of $\mathcal{T}$. □

A.3 Star-Shaped Polyhedra with Non-Extreme Vertices

We introduce several new concepts for the proofs in this section. Let v, a, b, c be different vertices of $\mathcal{T}$. The cone $\mathcal{C}_s(\triangle abc)$ is called the *cover* of v if v lies inside it. It is then called the *minimal cover* of v if no other vertices of $\mathcal{T}$ 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 8. We show that any non-extreme vertex v is locally covered, and we can always find a flippable edge incident to it. Thus we can use the D-criterion to decrease the degree of v and eventually remove it.

Lemma 5. *If v is a non-extreme vertex, then v is locally covered.*

Proof. By contradiction, assume that v is not locally covered. This means no cone of three vertices of $L(v)$ is a cover of v. We construct $\mathcal{CH}(L(v))$ and the line vs, as shown in Figure 9(a). By Lemma 1, v is inside $\mathcal{C}_s(F(v))$, so the line vs must intersect $\mathcal{CH}(L(v))$ at two triangles t_1 and t_2. Note that t_1 and t_2 are possibly the same triangle when $L(v)$ 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 $\mathcal{T}$ lie on the half-space of H not containing v. Let R_t be this half-space, clearly $\mathcal{CH}(L(v))$ lies inside R_t. Consider any vertex p in $\mathcal{T}$ other than v and those in $L(v)$. Consider the plane through p, v and s, the half-plane defined by vs and containing p must intersect $L(v)$ at a point q. The point p must lie outside the angle $\widehat{vsq}$; otherwise it falls into $\mathcal{C}_s(F(v))$,

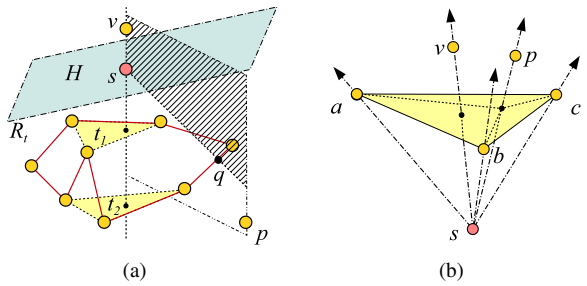

Figure 9: *(a) If v is not locally covered, the line passing through v and s intersects $\mathcal{CH}(L(v))$, and v is in fact an extreme vertex. The solid edges are of the link of v. (b) The cone of $\{a, b, c\} \in L(v)$ is a cover of v, and if another $p \in L(v)$ is inside $\mathcal{C}_s(\triangle abc)$, p subdivides the cone into three and v lies inside one of them.*

contradicting Lemma 1. Since q lies in R_t, p also lies in R_t. Thus all vertices of $\mathcal{T}$ other than v lie inside R_t and therefore v is an extreme point, a contradiction. □

Lemma 6. *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 $\{a, b, c\} \subseteq L(v)$. If there is another vertex $p \in L(v)$ 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 9(b). In this case, we replace $\triangle abc$ with $\triangle pab$ and repeat the argument. Since $\mathcal{C}_s(\triangle pab)$ is inside $\mathcal{C}_s(\triangle abc)$, this process cannot go on forever.

Now let us assume that no other vertex in $L(v)$ lies inside $\mathcal{C}_s(\triangle abc)$. We argue that $\mathcal{C}_s(\triangle abc)$ is completely inside $\mathcal{C}_s(F(v))$. Otherwise, an edge $\overline{pq}$ on $L(v)$ 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 $\mathcal{T}$ is star-shaped.

By Lemma 1, vertices in $\mathcal{T} \setminus L(v)$ cannot lie inside $\mathcal{C}_s(\triangle abc)$. Therefore, $\mathcal{C}_s(\triangle abc)$ is a minimal cover of v. □

Lemma 7. *If the degree of a non-extreme vertex v is 3, any edge incident to it is 3-1 flippable.*

Proof. By Lemma 5, 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 2, we have $\mathcal{C}_s(\sigma_e) = \mathcal{CH}(\{\overrightarrow{sa}, \overrightarrow{sb}, \overrightarrow{sc}, \overrightarrow{sv}\})$, therefore any edge incident to v is flippable. □

Lemma 8. *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 5 and Lemma 6, let $\mathcal{C}_s(\triangle abc)$ be the minimal cover of v where a, b and c are vertices of $L(v)$. 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 10. 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 $(a, p_1, p_2, \ldots, p_n, b)$ 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

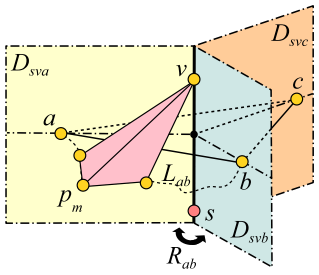

Figure 10: *If the chain of vertices L_{ab} has more than 2 vertices, there exists a vertex p_m such that the edge $\overline{vp}_m$ is 2-2 flippable.*

$\overline{ab}$. $\mathcal{C}_s(\triangle abc)$ 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 one of the cones of the triangles incident to $\overline{vc}$, violating the fact that $\mathcal{T}$ is star-shaped. Therefore, L_{ab} goes through R_{ab}. By the same argument, L_{ab} cannot intersect 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. Let H_{sab} be the plane through s, a and b. 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 $\mathcal{T}$ 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 neighbours in L_{ab} ($p_{m-1} = a$ if $m = 1$ and $p_{m+1} = b$ if $m = n$). Clearly, p_m lies outside $\mathcal{C}_s(\triangle v p_{m-1} p_{m+1})$. Since p_{m-1}, p_m and p_{m+1} lie inside R_{ab}, v is outside $\mathcal{C}_s(\triangle p_{m-1} p_m p_{m+1})$. Therefore, $\overline{vp}_m$ is a 2-2 edge and the union of $\mathcal{C}_s(\triangle v p_{m-1} p_m)$ and $\mathcal{C}_s(\triangle v p_m p_{m+1})$ 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 $v p_{m-1} p_m p_{m+1}$. As a result, $\overline{vp}_m$ is a 2-2 flippable edge. □

Theorem 2. *The Flip-Flop algorithm computes the convex hull of any star-shaped polyhedron.*

Proof. The Flip-Flop algorithm definitely terminates, since each flip performed either increases the volume of $\mathcal{T}$ or decreases the degree of a non-extreme vertex without increasing the degree of any other non-extreme vertex with smaller index. The algorithm flips all flippable edges, and if there is any unflippable edge, by Lemma 3, we can identify a non-extreme vertex, and by Lemma 7 and Lemma 8, we can flip and remove it. As such, when the algorithm terminates, the result is the convex hull. □

Theorem 2 promises the correctness of Flip-Flop in sequential execution. When we execute it in parallel, conflicting flippings may break the validity of the star-shaped polyhedron. By using atomic operations, we avoid this problem. Since the Flip-Flop algorithm follows the V- and D- criterion and the number of non-extreme vertices is limited, the algorithm always terminates after removing all non-extreme vertices and flipping all reflex edges. Thus, the proof of Theorem 2 applies to the parallel execution as well.

Theorem 3. *The ffHull algorithm computes the convex hull of any point set in $\mathbb{R}^3$.*

Proof. The point insertion process in the first phase of ffHull generates a star-shaped polyhedron from the input points, with those certainly non-extreme vertices being excluded. The rest of the proof follows from Theorem 2. □