Star Splaying: An Algorithm for Repairing Delaunay Triangulations and Convex Hulls

Jonathan Richard Shewchuk
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

Abstract

Star splaying is a general-dimensional algorithm that takes as input a triangulation or an approximation of a convex hull, and produces the Delaunay triangulation, weighted Delaunay triangulation, or convex hull of the vertices in the input. If the input is "nearly Delaunay" or "nearly convex" in a certain sense quantified herein, and it is sparse (i.e. each input vertex adjoins only a constant number of edges), star splaying runs in time linear in the number of vertices. Thus, star splaying can be a fast first step in repairing a high-quality finite element mesh that has lost the Delaunay property after its vertices have moved in response to simulated physical forces. Star splaying is akin to Lawson's edge flip algorithm for converting a triangulation to a Delaunay triangulation, but it works in any dimensionality.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Theory

Keywords: Delaunay triangulation, convex hull, Delaunay repair, star splaying, star flipping, dynamic mesh generation

1 Introduction

Any planar triangulation of a vertex set can be transformed into any other triangulation of the same vertices by a sequence of edge flips [18]. This happy circumstance is famously exploited by Charles Lawson's *flip algorithm* for computing Delaunay triangulations [19]. The flip algorithm is lauded as much for its elegant simplicity as for its usefulness.

Lawson's flip algorithm takes any triangulation of a planar vertex set V and transforms it into the Delaunay triangulation of V by flipping one edge at a time. Decisions about which edges to flip are made from purely local considerations—it is necessary to examine only the two triangles that adjoin an edge to decide whether to flip it. One interpretation of the flip algorithm is as a combinatorial optimization procedure. There is an objective function (described in Section 3) that maps each triangulation to a scalar objective value, and has the property that flipping an edge that is not "locally Delaunay" always increases the triangulation's objective value. The highest-valued triangulation is Delaunay (because only a Delaunay triangulation has every edge locally Delaunay). Lawson's optimization method is pure hill-climbing: it flips an edge only if the new triangulation has a greater objective value than the old one.

The notion of an edge flip generalizes to higher-dimensional operations called *bistellar flips* [7, 20, 23]. The flip algorithm generalizes to any dimensionality [16, 10], but the results do not. In three or more dimensions, the Delaunay triangulation still has the globally maximum objective value, but hill-climbing can get stuck in a local optimum that is not Delaunay [15]. Three-dimensional flipping gets stuck easily in practice. Can a more sophisticated optimization method replace hill-climbing? Not in five- or higher-dimensional space, where some vertex sets have triangulations that cannot be transformed to Delaunay by *any* sequence of bistellar flips [25, 27]. In three- and four-dimensional space, the question is open, important, and a survivor of many mathematicians' attacks: can every triangulation of a vertex set in E^3 or E^4 be transformed to every other triangulation of the same vertex set by bistellar flips? But even if the answer is "yes," bistellar flips are limited as an optimization tool.

Flipping is intrinsically interesting enough to have become an object of extensive mathematical study, even aside from its practical applications. Much investigation focuses on which triangulations can be transformed into each other through flipping [1, 6, 10, 11, 25, 26, 27, 33]. This paper asks a different question: how much more is possible if we bend the rules of flipping?

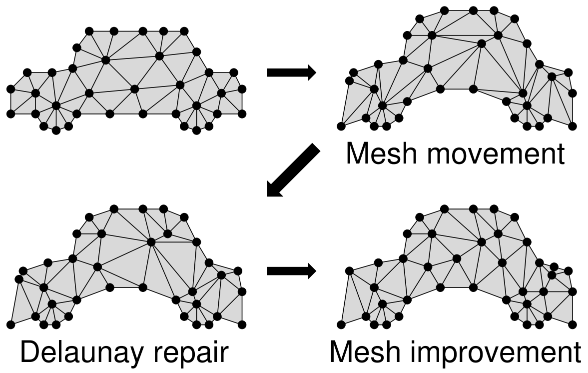


Figure 1. When an initially Delaunay mesh (upper left) moves, the quality of the triangles or tetrahedra may be compromised (upper right). One strategy for restoring their quality is to first restore the Delaunay property to the mesh (lower left), then use guaranteed-quality refinement and coarsening methods to fix any remaining problems (lower right).

Supported in part by the National Science Foundation under Awards ACI-9875170, CCR-0204377, and CCF-0430065, in part by an Alfred P. Sloan Research Fellowship, and in part by a gift from the Okawa Foundation.

SoCG'05, June 6–8, 2005, Pisa, Italy.

The main result of this paper is that a new algorithm described in Section 4, *star splaying*, transforms any triangulation of any dimensionality into a Delaunay or weighted Delaunay triangulation. More importantly, it does so quickly if few changes are needed and the input triangulation is *sparse* (meaning that each vertex adjoins only a constant, or slightly superconstant, number of edges). Alternatively, star splaying can compute the convex hull of a vertex set, and does so with relatively little work if it is given a good, sparse approximation of the convex hull to start with. Star splaying is a hill-climbing optimization algorithm that modifies a geometric structure with local operations, which are performed only if they individually improve an objective function.

Star flipping, described in Section 5, is a recursive variant of star splaying that has no theoretical advantage over plain star splaying, but might be faster in practice because it takes better advantage of the structure of the input triangulation. For general inputs, star splaying and star flipping have poor worst-case asymptotic running times, but for repairing triangulations and hulls that are sparse and only slightly broken, they are provably fast.

1.1 Applications and Motivations

The application that originally motivated the star splaying and star flipping algorithms is the maintenance of moving tetrahedral finite element meshes. (Two other potential applications are discussed in the conclusions.) Suppose the vertices of a Delaunay triangulation move small distances, rendering the triangulation no longer Delaunay. The goal is to restore the Delaunay property, with the vertices fixed in their new positions. Call this the *Delaunay repair* problem. It arises in Lagrangian formulations of finite element methods for modeling a material undergoing large deformations, when the vertices of a finite element mesh move in response to physical forces and the constitutive properties of the material. As Figure 1 illustrates, Delaunay repair can serve as the first step in an algorithm for repairing the quality of the elements in a mesh following mesh movement. There are theoretically guaranteed methods for improving the quality of a mesh through refinement and coarsening, but most of these methods rely on the Delaunay (or weighted Delaunay) property of the restored mesh to guarantee their success [4, 21, 22, 24, 31, 34].

Ideally, an algorithm for Delaunay repair should be fast (with linear running time *and* small constants) if the triangulation is, in some sense, "nearly Delaunay." Guibas and Russel [14] give experimental evidence that the flip algorithm converts three-dimensional triangulations that are arguably "nearly Delaunay" (having been produced by moving the vertices of Delaunay triangulations by small distances) to Delaunay triangulations, up to three times faster than the Delaunay triangulations can be constructed from scratch—so long as flipping does not get stuck. In their experiments, Guibas and Russel found that for small vertex movements, flipping only gets stuck occasionally. A Delaunay repair algorithm can start with flipping, then fall back on star splaying or star flipping to complete the repair when flipping gets stuck. Although star splaying and star flipping might be slower than classic flipping, they guarantee a successful result, and they can be held in reserve until classic flipping has done most of the work.

The central result of this paper is that star splaying and star flipping each restore the Delaunay property in linear time when the input triangulation is sparse and "nearly Delaunay" by one measure, discussed in Section 4.3. High-quality meshes are sparse, and if their vertices have moved small distances, they are likely to be nearly Delaunay by this measure. This result extends to weighted Delaunay (i.e. regular) triangulations and convex hulls. The proof appears in Sections 4.3 and 4.4.

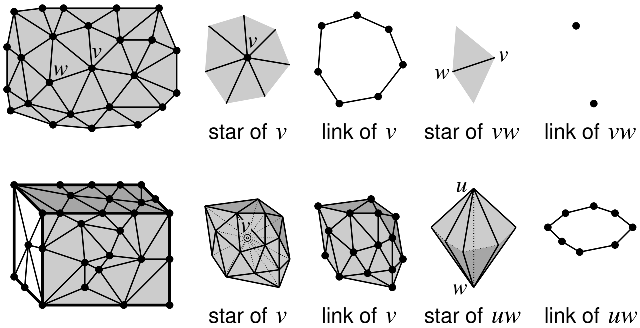


Figure 2. Top: A two-dimensional triangulation, the star and link of a vertex v in that triangulation, and the star and link of an edge vw. The link of v is a one-dimensional triangulation, albeit embedded in E^2. Bottom: A three-dimensional triangulation, and the stars and links of a vertex and an edge inside it. The link of v is a hollow two-dimensional triangulation. The link of uw is a one-dimensional triangulation.

2 The Ideas of Star Splaying and Star Flipping

Star splaying has two main ideas. First, a triangulation is represented as a collection of *stars*, one for each vertex. The star of a vertex v is the set of simplices in the triangulation that have v for a vertex, as Figure 2 illustrates. Observe that the star of v can contain a simplex without containing all the faces of that simplex. For instance, for any triangle in the star of v, the star contains only two of its edges; and although the star of v may contain many edges, it contains no vertex but v. The *link* of v is the set of simplices that are faces of simplices in v's star, but do not have v for a vertex, as illustrated. Each star is independently represented (at any point in time) by a *link triangulation* data structure, which simultaneously represents both the star and the link of a vertex.

Second, the stars of two different vertices are not required to agree. For example, the star of a vertex w might contain the edge vw and the triangle uvw, whereas the star of v might contain neither of them. In effect, each vertex has its own opinion about what the triangulation currently is, and maintains its star in that imagined triangulation. The differences of opinion among stars create flexibility that allows stuck triangulations to get unstuck.

When a simplex s (an edge, triangle, tetrahedron, etc.) lies in the star of one vertex w of s but not in the star of another vertex v of s, the stars of v and w are *inconsistent* with each other. At this time, the collection of stars does not represent a triangulation. The core of star splaying is a consistency enforcement algorithm that reconciles all the inconsistencies, so the stars represent a convex hull, a Delaunay triangulation, or a weighted Delaunay triangulation in the end. Consistency enforcement causes each star to splay open like an umbrella, hence the name *star splaying*.

The representation may sound wasteful, because information is duplicated among stars. However, there are simple storage optimizations that eliminate most of the redundancy. Star splaying dovetails perfectly with the dictionary-based data structure of Blandford, Blelloch, Cardoze, and Kadow [2] (with minor modifications to support inconsistencies between stars), which represents a triangulation as a list of stars. The Blandford et al. data structure is more compact than traditional triangulation data structures, in part because mutually consistent stars share information to reduce redundancy, thereby improving both memory efficiency and speed. It is also easier to program on top of. The storage space increases with the number of inconsistencies, but usually only a small portion of the triangulation is inconsistent. (Details appear in a full-length version of this paper.)

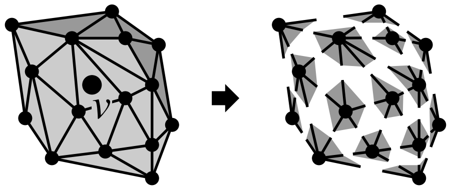


Figure 3. A two-dimensional link triangulation, represented as a collection of two-dimensional stars.

Star flipping is a variant of star splaying that adds two more ideas. First, the representation and the algorithm are recursive on the dimensionality. For example, in a three-dimensional triangulation, the star of a vertex v is represented by v's link, which is a two-dimensional triangulation. This two-dimensional triangulation is represented by a set of two-dimensional stars, as illustrated in Figure 3. These stars are not required to agree with each other either. Each two-dimensional star is represented by a one-dimensional link triangulation (recall Figure 2). The one-dimensional triangulations are called *link rings*, and unlike their higher-dimensional counterparts, they are always internally consistent.

Second, the workhorse of star flipping is the classic flip algorithm, at every level of the recursion. To make a star locally convex, star flipping tries to apply classic flipping within the link triangulation. Only if classic flipping gets stuck before restoring local convexity to a star does star flipping call itself recursively.

Star flipping, described in Section 5, seems likely to run faster than star splaying if the input triangulation is close to Delaunay, because it takes better advantage of the input triangulation.

3 Stars, Rays, and Cones

Star splaying is founded on several observations about the relationships between stars, rays, polyhedral cones, convex hulls, and Delaunay triangulations.

Consider the convex hull H of a set V of vertices in E^{d+1}. (Computing H is a standard way to compute a Delaunay triangulation in E^d; see below.) Suppose that V is *generic*: no $d+2$ points of V lie on a common hyperplane. Then H is a simplicial polytope—every facet of H is a d-simplex. Let ∂H denote the boundary triangulation of H. For consistency, *facets* are d-simplices and *ridges* are $(d-1)$-simplices throughout this paper, whether in E^{d+1} or in E^d.

Imagine wishing to compute not all of H, but just the star of one vertex v of H—specifically, v's star in ∂H, leaving out H proper. See Figure 4(a). Define the set of rays that originate at v and pass through other vertices of V, namely $R = \{\vec{vw} : w \in V\backslash\{v\}\}$. Let H_v be the convex hull of the rays R, illustrated in Figure 4(b). H_v is a polyhedral cone with vertex v and $H \subset H_v$. The star of v wraps around the tip of H_v like a paper shell around an ice cream cone. The star is combinatorially equivalent to the cone's boundary: the face lattice for the proper faces of H_v is isomorphic to the face lattice for the star of v. In the isomorphism, the rays on the boundary of H_v are in one-to-one correspondence with the edges in v's star and the vertices in v's link.

Let h be a hyperplane that separates v from all the other vertices in V, illustrated in Figure 4(c). The cross-section $P = H_v \cap h = H \cap h$ is a d-polytope, namely the convex hull of the intersection points $\{\vec{r} \cap h : r \in R\}$. The face lattice of P's boundary is isomorphic to the face lattice of v's link.

The central observation is that these three problems are essentially equivalent: computing the star or link of v in the boundary of the $(d+1)$-dimensional convex hull H, computing the $(d+1)$-

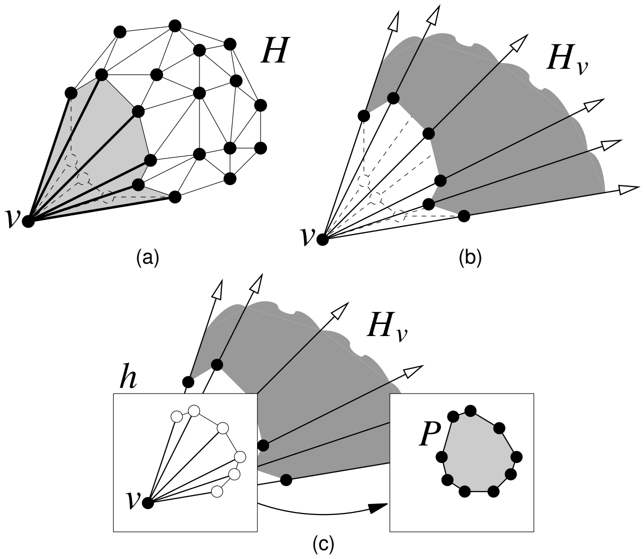


Figure 4. (a) The star of v in a convex polyhedron H. (b) The convex hull H_v of rays, a polyhedral cone whose boundary is combinatorially equivalent to v's star. (c) A cross-section of the cone is the convex hull of the points where the rays intersect the cross-sectional hyperplane.

dimensional convex hull H_v of rays, and computing the d-dimensional convex hull P of points. The wealth of ideas computational geometers employ for the last problem apply immediately to the first problem.

One way to compute H is to use a d-dimensional convex hull algorithm to compute the star of each vertex in V individually. Naively applied, this method is strikingly slow—its *best-case* running time is in $\Theta(n^2)$, where n is the number of vertices in V. But if the candidates for inclusion in the link of each vertex could be pruned to a small number—say, a constant number of vertices in each link—then the method becomes strikingly attractive, as it entails just a linear number of constant-time convex hull computations.

At first glance, one inconvenience of this method appears to be finding a corner-cutting hyperplane h. This step is not only unnecessary; it is unwise, because computing the intersections of the rays in R with h introduces avoidable roundoff errors. Many algorithms for computing convex hulls of point sets in E^d can be adapted to compute convex hulls of rays originating at a common point v in E^{d+1}, simply by replacing the orientation tests on $d+1$ points in E^d with orientation tests on $d+2$ points in E^{d+1} (wherein v is always one of those points).

The standard incremental insertion method for updating a convex hull is the beneath-beyond method of Kallay [17], which adds one new vertex (or ray) at a time and maintains after each addition the convex hull of the vertices (or rays) processed so far. Let C be the convex hull at a fixed moment in time between vertex insertions. Let f be a facet of C. A point p is said to be *beyond* f if p and C lie on opposite sides of the affine hull of f. The beneath-beyond algorithm adds a vertex w and transforms C into $\mathrm{conv}(C \cup w)$ by finding and deleting every facet of C that w is beyond, then creating new facets that attach w to every ridge of C that adjoins exactly one surviving facet.

This idea works whether C is a convex hull of points or a convex hull of rays with a common origin v. The latter circumstance is illustrated in Figure 5. There is one important algorithmic difference between these two circumstances: a ray can lie beyond *every* facet of the cone C, meaning that the convex hull of the rays is E^{d+1}.

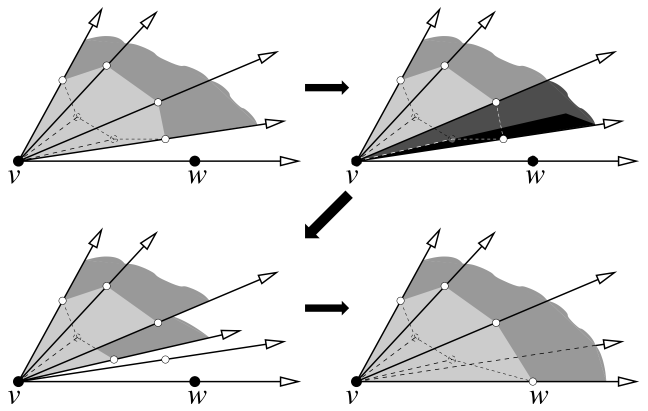


Figure 5. Incremental construction of a convex hull of rays. To add a new ray $\vec{vw}$ (upper left), identify the cone facets visible from w (darkened at upper right), delete them (lower left), and create facets connecting $\vec{vw}$ to the exposed ridges (lower right).

Then v is in the interior of the convex hull, and cannot be a vertex of H.

The beneath-beyond algorithm can use the same topological representation for either a $(d + 1)$-dimensional polyhedral cone C or a d-dimensional polytope P, namely a $(d - 1)$-dimensional triangulation that represents the boundary of P. This triangulation, called v's *link triangulation*, is a topological entity that simultaneously represents v's star and link, the cone C, and the cross-sectional polytope P.

This algorithm generalizes to computing the stars of the edges, 2-faces, and so forth of ∂H. The *star* of a simplex s in a triangulation T is the set of simplices in T that have s for a face, including s itself. Figure 2 shows the stars of edges in two- and three-dimensional triangulations. The *link* of s is the set of simplices that are faces of simplices in v's star, but do not share a vertex with s. Observe that the link of an edge in a three-dimensional triangulation is a one-dimensional triangulation. The link of an i-face of ∂H is combinatorially equivalent to the boundary of the convex hull of some point set in E^{d-i}, and can be computed with the beneath-beyond algorithm. Details appear in the full-length version of this paper.

Clarkson and Shor [5] show that if the beneath-beyond algorithm constructs a convex hull of n vertices in E^d by inserting vertices one by one according to a random permutation (drawn from a uniform distribution) and using a *conflict graph* to perform point location, it runs in $O(n^{\lfloor d/2 \rfloor} + n \log n)$ expected time. Therefore, it computes the star of an i-face of a simplicial $(d + 1)$-dimensional convex hull in $O(n^{\lfloor (d-i)/2 \rfloor} + n \log n)$ expected time.

All these ideas adapt easily if the goal is to compute only the *underside* of H—the facets of H whose outward-directed normals have negative x_{d+1}-coordinates. For example, the beneath-beyond algorithm can incrementally compute just the underside of a cone: simply discard at each step any facet not on the underside of the cone. This makes it possible to compute a Delaunay or weighted Delaunay triangulation without wasting time on the upper boundary of H.

A well-known way to compute the Delaunay triangulation DT(N) of a d-dimensional vertex set N (for "nodes") is to use a $(d+1)$-dimensional convex hull construction algorithm and Seidel's parabolic lifting map [29, 9] (inspired by the numerically less well-behaved spherical lifting map of Brown [3]). Seidel's lifting map lifts each vertex $w \in N$ to a *lifted companion* $w^+ = \langle w, |w|^2 \rangle \in E^{d+1}$, where $|w|^2$ is the inner product of w with itself. The set of lifted vertices is $V = \{w^+ : w \in N\}$, and all the lifted vertices lie on the paraboloid $x_{d+1} = x_1^2 + x_2^2 + \cdots + x_d^2$. DT($N$) is combinatorially equivalent to the underside of the convex hull of V.

For any triangulation T whose vertices are N, define the *lifted triangulation* T^+ to be T with each vertex w replaced by its lifted companion w^+. T^+ is the underside of conv(V) if and only if T is Delaunay. Recall from the introduction that the flip algorithm performs flips that increase a scalar objective function $f(T)$. That objective function is the volume bounded between T^+ below and a fixed arbitrary horizontal hyperplane above. The flip algorithm never actually computes this objective function, but it only flips a ridge of T whose lifted companion on T^+ is not locally convex, which causes a portion of T^+ to move down, thereby increasing the volume above T^+.

If you compute the star of each vertex of H = conv(V) separately, the stars will agree with each other if H is simplicial. If you need only the underside of H, only the underside of H need be simplicial. The algorithms discussed in this paper are only guaranteed to work correctly if the vertex set V is *generic*; but the definition of generic depends on the problem being solved. If the goal is to compute the whole convex hull H, then V is said to be generic if no $d + 2$ vertices in V lie on a common hyperplane. If the goal is to compute the underside of H, then V is said to be generic if no $d + 2$ vertices in V lie on a common *non-vertical* hyperplane.

Unfortunately, both genericity and the use of exact arithmetic in the orientation tests are practical requirements for the success of an implementation of star splaying or star flipping. Genericity can be simulated using the symbolic perturbation techniques of Edelsbrunner and Mücke [8]. If the goal is to compute only the underside of H (i.e. to compute a Delaunay or weighted Delaunay triangulation), then only the x_{d+1}-coordinates of the vertices need to be perturbed, and genericity is almost painless to implement [8, Section 5.4], [32, Section 6].

4 Star Splaying

Star splaying computes a convex hull with the help of the beneath-beyond method for convex cones. Let V be a set of vertices in E^{d+1}, and suppose the goal is to compute its convex hull H = conv(V), or perhaps just the underside of H. Each vertex v in V is assigned a small *starting set* $W_v \subseteq V \setminus \{v\}$ of vertices that are thought likely to be in the link of v in ∂H.

The success and speed of star splaying depend on how well each W_v anticipates which vertices are in v's link in ∂H, and which vertices are not. A precondition that guarantees speed is discussed in Section 4.3, and a precondition that guarantees success is discussed in Section 4.4. There are many ways the starting sets could be assembled. A simple heuristic is to use a spatial data structure like a quadtree or an array of cubical buckets to find some near neighbors for each vertex. (This method might be an easy way to build a parallel Delaunay triangulator for "nicely" distributed vertices.)

It is straightforward to choose starting sets for the Delaunay repair problem. Suppose that T is a simplicial mesh that was produced by moving the vertices of a Delaunay triangulation by small distances. Let V be the set of vertices of the lifted triangulation T^+. For each vertex $v \in V$, choose W_v to be the set of vertices connected to v by an edge of T^+. (But see Section 4.4.)

4.1 The Star Splaying Algorithm

Star splaying begins by computing for each vertex $v \in V$ the convex cone C_v = conv$\{\vec{vw} : w \in W_v\}$, as described in Section 3. If all the stars of the vertices in V are now consistent with each other, star splaying is done and has constructed the convex hull H of V (or

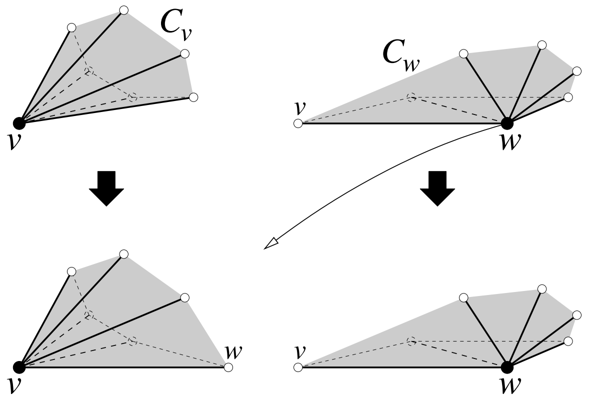


Figure 6. The edge vw is in w's star but not in v's star. The consistency enforcement algorithm inserts w into v's link triangulation by adding the ray $\vec{vw}$ to v's cone C_v.

its underside). This happens if and only if for every vertex v, W_v includes all the neighbors of v in the boundary (or underside) of H.

An *inconsistency* is the circumstance that some simplex appears in the star of one of its vertices but not in the stars of all its vertices. A consistency enforcement algorithm fixes the inconsistencies.

The enforcement algorithm fixes inconsistent stars by inserting new vertices into the link triangulations—that is, by adding new rays to the cones using Kallay's beneath-beyond algorithm. Consistency enforcement is guaranteed to terminate, because every enforcement step causes some star to splay wider, like an opening umbrella. The cone corresponding to that star becomes a superset of the old cone, and no cone ever shrinks. The sum of the solid angles of all the cones is an objective function that increases with every enforcement step, and is maximized by the convex hull H.

Consistency enforcement relies on the idea of a *proof* that a simplex is not part of the boundary of a convex hull. For example, in E^3, if a vertex y lies inside a tetrahedron $uvwx$, then $uvwx$ is a proof that y is not on the boundary of the convex hull of any point set that includes $\{u, v, w, x\}$. If the relative interior of an edge uv intersects the relative interior of a triangle wxy, and the two are not coplanar, then uv and wxy each prove that the other is not on the boundary of the convex hull of any point set that includes $\{u, v, w, x, y\}$.

Suppose a simplex s appears in the star of one of its vertices w, but not in the star of another of its vertices v. Because s has vw for an edge, there are only two possibilities, each requiring a different enforcement step.

- **Reconciling an asymmetric edge.** One possibility is that vw does not appear in v's star, though vw appears in w's star. The consistency algorithm inserts w into v's link triangulation—or equivalently, adds the ray $\vec{vw}$ to v's cone C_v, as illustrated in Figure 6.

  If $\vec{vw}$ is inside v's cone, v's link triangulation is unchanged. If this occurs, the star splaying algorithm looks for a proof that the open edge vw is interior to H, and uses that proof to eliminate vw from w's star. Select an arbitrary plane $L \subset E^{d+1}$ that includes $\vec{vw}$, as illustrated in Figure 7. L intersects two facets f and g of v's star (at points other than v), and those facets constitute a proof that the open edge vw is interior to $\mathrm{conv}(f \cup g \cup \{w\}) \subseteq H$. Finding f and g is equivalent to doing point location twice in the $(d-1)$-dimensional link triangulation.

  To ensure that the consistency algorithm makes progress, incrementally insert the vertices of f and g into w's link triangulation (except v, which is already there) as illustrated. The

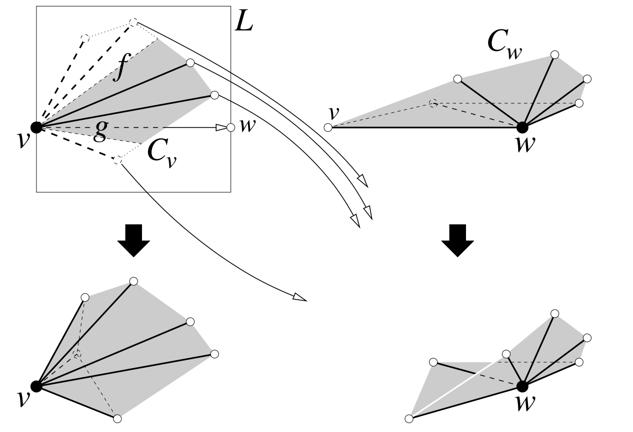


Figure 7. Adding $\vec{vw}$ to v's cone C_v does not change it, because $\vec{vw} \subset C_v$. The consistency enforcement algorithm finds the two facets of C_v that intersect a plane L through $\vec{vw}$, and inserts their vertices into w's link triangulation, eliminating the edge wv.

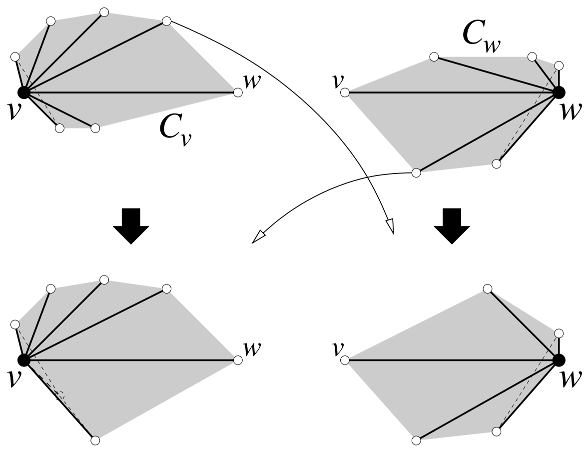


Figure 8. The consistency algorithm makes the stars of v and w agree about which triangles adjoin the edge vw.

  updated cone C_w includes $\mathrm{conv}(f \cup g \cup \{w\})$, so the ray $\vec{wv}$ no longer lies on the boundary of the cone, and v is no longer in w's link triangulation.

  If the goal is to construct only the underside of H, choose L to be vertical. The facet directly below vw suffices to prove that vw is not part of the lower boundary of H. No second facet is needed.

- **Reconciling conflicting edge stars.** The second possibility is that vw appears in the stars of both v and w, but the two stars disagree about what simplices include the edge vw.

  Say that the star of the *ordered edge* vw is the star of vw in v's star, and the star of wv is the star of vw in w's star. In other words, the star of vw reflects what v thinks the star of the edge vw is, whereas the star of wv reflects what w thinks.

  The consistency algorithm finds the vertices that are in the link of vw but not in the link of wv, and incrementally inserts them into w's link triangulation as illustrated in Figure 8. Symmetrically, the vertices that are in the link of wv but not the link of vw are inserted into v's link triangulation.

  After these operations, either the ordered edges vw and wv have the same star, or the edge vw is now absent from the stars of both v and w.

Occasionally, a consistency enforcement step might eliminate the star of v or w (or both) altogether. For example, consider reconciling an asymmetric edge. If w is beyond all the facets of v's cone C_v, then $\mathrm{conv}(C_v \cup \vec{vw}) = E^{d+1}$, which implies that v is in the interior of H. The star splaying algorithm maintains a *death certificate* for v: a proof that v is an interior point. The death certificate comprises the vertices of the two facets of C_v that intersect an arbitrary plane through $\vec{vw}$ (at points other than v), plus w itself. (Imagine Figure 7, upper left, with w moved to the left of the cone's tip. If the goal is to construct only the underside of H, use just the lower facet of C_v that intersects a *vertical* plane through $\vec{vw}$, plus w itself.) The consistency algorithm passes v's death certificate back to w, and also stores the death certificate so it can be used later to reconcile any other vertex that thinks that v is in its link.

The consistency enforcement steps can be applied in any order. Some ways of ordering the steps are more efficient than others; see Sections 4.2 and 4.5 for suggestions.

4.2 Finding Inconsistencies

The consistency enforcement algorithm needs an efficient way to find inconsistencies between stars. It is inexpensive to determine whether an edge appears in the star of only one of its two endpoints. However, checking whether two ordered edges vw and wv have the same star can be expensive, because a single edge might adjoin $\Theta(n^{\lceil d/2\rceil-1})$ facets of the convex hull. It is better to avoid performing this check directly.

One way to drive star splaying efficiently is to maintain a list of facets whose consistency is in doubt. Each record in the list is a triple $\langle f, v, w\rangle$ consisting of a facet f and two of its vertices v and w, such that f was in w's star when the triple was enlisted. After constructing the convex cones of the starting sets, initialize the facet list: for each vertex $w \in V$, for each facet f in w's star, for each vertex $v \neq w$ of f, place $\langle f, v, w\rangle$ in the facet list.

The consistency algorithm is a loop that repeats the following operations until the facet list is empty. Remove one triple $\langle f, v, w\rangle$ from the list. If f is in both v's star and w's star, or if f is in neither, discard the triple and move on to the next iteration. Otherwise, perform the appropriate consistency enforcement steps for the edge vw until v and w agree about whether or not f exists. Observe that the test takes $O(1)$ time if no action is taken, which is why facets drive the consistency algorithm.

When the consistency enforcement algorithm inserts a vertex w into the link of a vertex v, new facets appear in v's star. All the new facets have vw for an edge. For each new facet f, for each vertex $u \neq v$ of f (including w), add the triple $\langle f, u, v\rangle$ to the facet list.

If a vertex insertion causes a vertex u to be deleted from v's link, and v is in u's link, then find a facet f in u's star that has v for a vertex, and add the triple $\langle f, v, u\rangle$ to the facet list. If a vertex insertion kills v's star (i.e. v's cone becomes E^{d+1}), do this for every vertex u such that u and v are in each others' links. This ensures that v's death certificate will eventually be sent to u.

When the facet list is empty, star splaying is complete.

One can prove that this procedure catches every inconsistency by verifying that it maintains two invariants. First, if an edge vw appears in the star of w but not in the star of v, then the facet list contains a triple $\langle f, v, w\rangle$ for some facet f in w's star. Second, if an edge vw appears in the stars of both v and w, and a facet f having edge vw appears in the star of w but not in the star of v, then either the facet list contains the triple $\langle f, v, w\rangle$, or it contains a triple $\langle f', w, v\rangle$ where f' is a facet in v's star that has a vertex that is beyond f. (The last circumstance arises if f was once in v's star, but was deleted by a subsequent vertex insertion.)

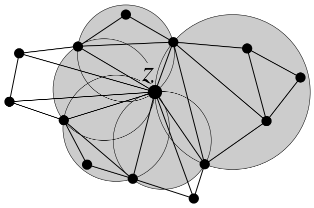


Figure 9. B_z is the union of the shaded discs. η_{z^+} is the number of vertices that are in the interior of B_z (except z) or connected to z by an edge—that is, all the vertices illustrated here except z itself.

The procedure described above suffices to reconcile all inconsistencies, but it can be optimized somewhat. When it is processing a triple $\langle f, v, w\rangle$ for which both v's star and w's star contain a ridge r of f, the reconciliation of the edge stars for vw and wv can be replaced by a simpler and faster step that reconciles two ridge stars—specifically, the star of r in v's star and the star of r in w's star. (The star of a ridge consists of the ridge and its two adjoining facets.)

Another simple improvement is to use three separate facet lists. The second list is processed only while the first list is empty, and the third is processed only while the first two are empty. New triples are placed in the first list, except triples known to represent asymmetric edges, which go into the third list. When the algorithm takes a triple $\langle f, v, w\rangle$ from the first or second list and finds that the vertex w is not in v's link, it places the triple in the third list instead of processing it immediately. When the algorithm takes a triple $\langle f, v, w\rangle$ from the first list and finds that no ridge of f is in v's star, it places the triple in the second list instead of processing it immediately. The idea is that point location in v's link is fastest when the search starts from a known ridge, slower when it starts from a known edge, and slowest when there is no obvious starting point. Many of the inconsistencies in the second and third facet lists might be reconciled indirectly during the processing of other inconsistencies, in which case the cost of point location is lessened.

4.3 The Running Time of Star Splaying

The speed of star splaying depends on how sparse each starting set W_v is, and how well each W_v predicts the edges of the final cone $H_v = \mathrm{conv}\{\vec{vw} : w \in V \backslash \{v\}\}$ for each vertex v. Star splaying begins by creating for each vertex v an initial cone $C_v = \mathrm{conv}\{\vec{vw} : w \in W_v\}$. Subsequently, v's star will splay only for a vertex not in C_v. Thus, for each vertex v, let η_v be the number of vertices in W_v plus the number of vertices of V that are not in the starting cone C_v. The quantities η_v encapsulate both the sparsity and the predictive accuracy of the starting sets, and take part in a bound on the running time of star splaying.

For the Delaunay repair problem, η_v has the following interpretation. Let N be the set of vertices of the starting triangulation T. Let $V = \{w^+ : w \in N\}$ be the vertices of the lifted triangulation T^+. In E^d, let N_z be the set of vertices connected to a vertex $z \in N$ by an edge of T. The starting set for z^+ is $W_{z^+} = \{w^+ : w \in N_z\}$. The boundary of the initial cone C_{z^+} is combinatorially equivalent to the star of z in the Delaunay triangulation $\mathrm{DT}(N_z \cup \{z\})$. Let B_z be the union of the circumscribing balls of the d-simplices that adjoin z in $\mathrm{DT}(N_z \cup \{z\})$, illustrated in Figure 9. A vertex w is in the interior of B_z if and only if w^+ is outside C_{z^+}. Thus, η_{z^+} is the sum of the size of N_z and the number of vertices of N in the interior of B_z.

In high-quality meshes whose smallest solid angles are bounded above some fixed positive constant, the degree of a vertex cannot exceed some constant. If the vertices of T have not moved too far from a high-quality Delaunay configuration, it is likely that every $\eta_v \in O(1)$.

Let n be the number of vertices in V, and let d represent the dimension, where star splaying is computing either a convex hull in E^{d+1}, or a Delaunay or weighted Delaunay triangulation in E^d. Assume that a dictionary-based data structure permits looking up a simplex in $O(1)$ time, and looking up the simplices that adjoin it in time linear in their number. (The full-length version of this paper gives one example of such a data structure.)

When a star splays, the new cone is a superset of the old cone, and a cone never shrinks. The star of a vertex v can splay at most η_v times, because once a cone contains a vertex, introducing the vertex again changes nothing. Although a consistency step sometimes attempts a vertex insertion that has no effect, it performs at most $2d$ of these for every vertex insertion that *does* splay some star. Therefore, star splaying is guaranteed to terminate.

Consider the cost of one splaying operation, which inserts a vertex w into the link triangulation of another vertex v. The costs include point location (finding one facet deleted by w), deleting old facets, and creating new facets. The number of new facets is in $O(\eta_v^{\lceil d/2\rceil-1})$, because the new facets are in one-to-one correspondence with the $(d-2)$-simplices in vw's new link, which is a $(d-2)$-dimensional triangulation. A facet cannot be deleted without being created first, so simply charge the cost of deleting each facet to its creation. (A single vertex insertion can delete up to $\Theta(\eta_v^{\lfloor d/2\rfloor})$ facets, but that cost is amortized over other, less expensive vertex insertions.)

If the order of vertex insertions into a star could be entirely randomized, with each permutation being equally likely, the expected number of new facets per insertion would drop to $O(\eta_v^{\lfloor d/2\rfloor-1})$, or even to $O(1)$ if the vertices are distributed nicely enough that no cone's complexity ever exceeds $O(\eta_v)$ [5].

Unfortunately, although star splaying allows consistency enforcement operations to occur in a random order, ideal randomization of the vertex insertion order is not possible. The vertices in W_v are inserted into v's link before any others. If the starting set W_v is chosen by an adversary, the expected number of new facets per vertex insertion can be in $\Theta(\eta_v^{\lceil d/2\rceil-1})$, even if the vertices of W_v are inserted according to a random permutation, and all the subsequent vertices are inserted according to a random permutation. Nevertheless, incremental insertion frequently realizes the better bound in practice, in which case the speed of point location becomes the performance bottleneck for $d = 3$.

Point location in v's link triangulation by exhaustive search takes $O(\eta_v^{\lfloor d/2\rfloor})$ time. Slow as that may be, if η_v is small for every v—the circumstance star splaying is designed for—then exhaustive search suffices. However, more aggressive approaches are available for links whose sizes exceed some constant. Linear programming can find a facet that w is beyond in $O(\eta_v)$ time, as Seidel [30] points out. For $d = 2$, point location takes $O(\log \eta_v)$ time if each oversized link ring is stored in two balanced search trees (one for the underside and one for the top side).

For the important case $d = 3$—or any circumstance where the complexity of a star remains linear in η_v after each vertex insertion—there are at least two sublinear-time point location strategies. Raimund Seidel (personal communication) suggests a simple $O(\sqrt{\eta_v})$-time algorithm that is easy to program. It seeks a facet of v's four-dimensional cone C_v that w is beyond. Let q be a point in the interior of C_v (e.g. the centroid of any $d+2$ vertices in v's link). Choose a random sample of $\Theta(\sqrt{\eta_v})$ facets of C_v, and compute for each one a normal vector. Among the chosen facets, find the facet whose normal vector $\vec{\mu}$ maximizes the quotient $\vec{\mu}\cdot(w-v)/\vec{\mu}\cdot(v-q)$. From that facet, do walking point location in C_v: walk through a sequence of adjoining facets for which the quotient is monotonically increasing. The expected number of facets traversed is in $O(\sqrt{\eta_v})$. Either w is beyond the facet of C_v that maximizes the quotient, or w is in C_v. The search can stop early if it finds a facet for which the quotient is positive, indicating that w is beyond that facet.

A second point location method, more cumbersome to implement, is to use a history dag [13, 10]. Point location with a history dag takes expected $O(\log \eta_v)$ time if the complexity of the star never exceeds $O(\eta_v)$ and the vertices are inserted according to a random permutation chosen from a uniform distribution. Though star splaying cannot choose permutations uniformly, there is still a good chance of achieving $O(\log \eta_v)$ performance in practice.

After point location finds one facet to delete, a depth-first search finds the others in $O(1)$ time per deleted facet. Charge this cost to the creation of those facets. The $O(1)$ cost of processing a facet in the facet list can be charged to a facet that was created or deleted when the facet was enlisted. The following theorem sums all the costs.

Theorem 1. *The expected worst-case running time of star splaying is in* $O(\sum_{v\in V}(\eta_v^{\lceil d/2\rceil} + \eta_v \log \eta_v))$ ■

If luck bestows upon us the better performance implied by the randomized bounds, replace the ceiling in Theorem 1 with a floor.

In the very worst case, $\eta_v \in \Theta(n)$ for every v, and the running time is in $O(n^{\lceil d/2\rceil+1} + n^2 \log n)$. By comparison, flipping repairs a Delaunay triangulation in $O(n^{\lfloor d/2\rfloor+1})$ worst-case time—when it works. Star splaying is not an appropriate algorithm to use when the initial approximation to the convex hull has high complexity or is a bad approximation. Computing a convex hull in E^{d+1} from scratch takes $O(n^{\lceil d/2\rceil} + n \log n)$ time [5, 30].

The main result of this paper is that an approximate convex hull or Delaunay triangulation can be repaired in linear time if it is sparse and "nearly convex" or "nearly Delaunay," in the following sense.

Corollary 2. *Star splaying runs in $O(n)$ time when $\eta_v \in O(1)$ for every vertex v.* ■

In this circumstance, point location by exhaustive search suffices.

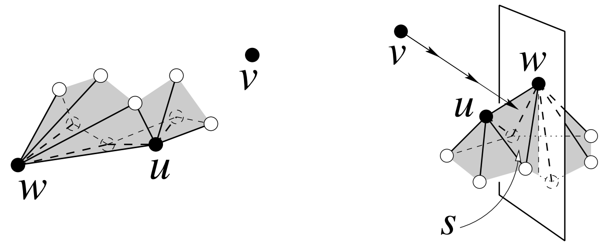
A small number of vertices with slightly superconstant η can be tolerated without sacrificing the linear time bound. For example, if $O(1)$ of the vertices have $\eta \in O(n^{1/(\lfloor d/2\rfloor+1)})$, and the rest have $\eta \in O(1)$, the running time is still linear, even with point location by exhaustive search.

If the flip algorithm runs prior to star splaying, flipping may reduce some or all of the η values; it does not increase any of them. The running time of star splaying is not compromised by flipping first.

4.4 The Correctness of Star Splaying

The star splaying algorithm always terminates, whereupon the stars are all mutually consistent. Does it produce $\mathrm{conv}(V)$?

The answer depends on the starting sets W_v that seed star splaying. If these are poorly chosen, star splaying might construct several triangulated manifolds, each bounding a different convex polytope. These manifolds share no vertices, but might intersect each other. For example, if the vertices are colored red and green, and every vertex has only vertices of its own color in its starting set, the red and green vertices will never find out about each other. If $d = 1$, the "manifolds" might not even be manifolds: a locally convex path can whirl through several self-intersecting turns before returning to its starting point. The following theorem states a precaution that ensures that star splaying produces the convex hull.



(a) (b)

Figure 10. (a) A vertex $w \in \partial H$ that lexicographically precedes v. (b) A vertex w that lexicographically follows v. The vertical plane shows the boundary between points that lexicographically precede w and those that lexicographically follow.

Theorem 3. *Let V be a generic vertex set in E^{d+1}. Suppose that for every vertex $v \in V$ except the lexicographically minimum vertex, v's starting set W_v contains at least one vertex that lexicographically precedes v. Then star splaying constructs the boundary ∂H of the convex hull $H = \mathrm{conv}(V)$.*

Proof. Upon completion, star splaying has computed for each vertex u a cone C_u, which is a convex hull of some rays originating at u and passing through other vertices of V. The "correct" cone H_u is the convex hull of *all* the rays originating at u and passing through other vertices of V (see Section 3). Therefore, $C_u \subseteq H_u$. If $C_u = H_u$ for every $u \in V$, star splaying has computed ∂H.

Suppose for the sake of contradicting the theorem that star splaying fails. Then some vertex's cone does not contain every vertex of V. For the rest of this proof, let v be the lexicographically minimum vertex that is not in every cone.

By the following reasoning, every vertex w that lexicographically precedes v and is on the boundary ∂H has the correct cone; i.e. $C_w = H_w$. Let u be any vertex that shares an edge of ∂H with w—equivalently, any vertex on the boundary of w's correct cone H_w, as illustrated in Figure 10(a). Because w precedes v, w is in every cone, including u's cone C_u. As uw is an edge of the convex hull H, $\vec{uw}$ is on the boundary of C_u, so uw is in u's star. All the stars are consistent, so uw is in w's star. This is true for every neighbor u of w on ∂H, so w's cone is correct, as promised.

By assumption, for every vertex $x \in V$ except the lexicographically minimum vertex, x's starting set W_x contains a vertex that lexicographically precedes x. The convex cone C_x contains this vertex, so unless $C_x = E^{d+1}$ (which has no boundary), some boundary ray of C_x must pass through a vertex (not necessarily the same vertex) that lexicographically precedes x.

It follows that *every* vertex that precedes v has the correct cone. Suppose for the sake of contradiction that this is not true, and let w be the lexicographically minimum vertex whose cone is wrong. Because every vertex on ∂H that precedes v has the correct cone, w is in the interior of H, and its correct cone is $H_w = E^{d+1}$. Because w has the wrong cone, $C_w \neq E^{d+1}$. By the preceding paragraph, some boundary ray of C_w passes through a vertex y that lexicographically precedes w. The edge wy is in w's star. Because the stars are consistent, wy is also in y's star, so the ray $\vec{yw}$ is on the boundary of y's cone. But w is in the interior of every correct cone, so y's cone C_y is wrong. This contradicts the assumption that w is the lexicographically minimum vertex whose cone is wrong. Therefore, every vertex that precedes v has the correct cone.

Recall that v is not in every cone. Let w be the lexicographically minimum vertex whose cone C_w does not contain v. Because w has the wrong cone, w lexicographically follows v. By the second paragraph preceding this one, some boundary ray of C_w passes through a vertex that lexicographically precedes w.

There exists a facet s in w's star and a vertex u of s such that u lexicographically precedes w and v is beyond s, as illustrated in Figure 10(b). To exhibit s, shoot a ray from v toward any point in C_w that lexicographically precedes w, as illustrated, and let f be the facet of C_w that the ray strikes first. Clearly, v is beyond f. Let s be the d-simplex in w's star for which $s \subset f$. At least one vertex of s must lexicographically precede w; call that vertex u. Because the stars are consistent, s is in u's star as well. Because v is beyond s, v is not in u's cone. But this contradicts the assumption that w is the lexicographically minimum vertex whose cone does not contain v. It follows that v is in every cone—contradicting the fact that v is the lexicographically minimum vertex that is not in every cone. Therefore, every vertex is in every cone, and star splaying constructs ∂H. ■

In general, it is easy to make the precondition of Theorem 3 hold—for example, by putting the lexicographically minimum vertex into every starting set that does not already satisfy the precondition. If the starting sets are seeded with a triangulation T of a convex domain, as described in Section 4, then the precondition already holds with no extra effort. Star splaying can succeed if the input is messy—for instance, if the triangulation T is not convex or has "inverted" d-simplices, or if the goal is to convert a dimpled or self-intersecting $(d+1)$-dimensional polytope into a convex hull—but a little effort is necessary to ensure the precondition is met.

To compute a Delaunay triangulation, star splaying only needs to maintain the underside of each cone. To show that this modified algorithm is correct, replace ∂H in Theorem 3 with the underside of H, and replace each cone with an *extended cone*. The extended cone of a vertex is the set of all points in or above its cone. In other words, if the cone contains a point $(p_1, p_2, \ldots, p_d, p_{d+1})$, the extended cone contains $(p_1, p_2, \ldots, p_d, b)$ for every $b \geq p_{d+1}$. If a vertex proves not to be on the underside of H, its extended cone is E^{d+1}. It is straightforward to verify that Theorem 3 remains correct with these changes.

4.5 Algorithm Optimizations

If flipping precedes star splaying, the flip algorithm maintains a list of ridges that are not locally Delaunay. When flipping gets stuck, that list can identify vertices whose convex cones have already been computed—namely, any vertex that does not adjoin any ridge in the list. Star splaying does not need to waste time computing new cones for vertices whose neighborhoods are already locally Delaunay. If flipping makes good progress before getting stuck, star splaying might only need to operate on a small portion of the triangulation.

Kallay's algorithm inserts a new vertex v by finding one facet that v is beyond, then finding the others by depth-first search. In star splaying, a single d-simplex f might appear in up to $d+1$ different stars. If v is beyond f, then f is not a facet of the convex hull H, and f can be deleted from every star it appears in. The running times of convex hull algorithms are dominated in practice by the costs of numerical tests, like determining whether a vertex is beyond a facet, so it is wise to repeat as few tests as possible.

Thus, Kallay's depth-first search should extend from star to star, and not just take place within a single star. A single vertex insertion by the consistency enforcement algorithm might simultaneously modify many different stars.

Suppose star splaying is implemented this way. If the stars of two vertices v and w are mutually consistent and contain the edge vw, then the two stars will never be inconsistent with each other again. Some data structures, like the Blandford et al. simplex dictionary [2], can exploit this fact by storing only one copy of the link of the edge vw.

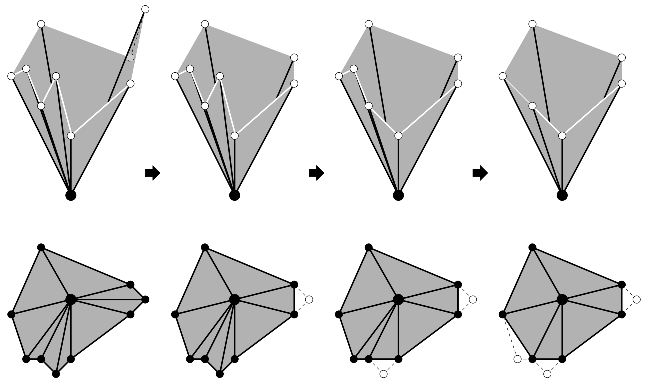


Figure 11. Each vertex in the original triangulation (bottom left) has a starting cone (top left), not necessarily convex, induced by the parabolic lifting map. This illustration shows a sequence of flips in a one-dimensional link ring that make a three-dimensional cone convex. In the flattened star of the vertex (bottom), the flips eliminate edges that are not locally Delaunay.

5 Star Flipping

Star splaying computes a convex cone for a vertex by calling a standard convex hull algorithm. However, in the Delaunay repair problem, every vertex already has a cone, derived from the starting triangulation T and the parabolic lifting map as illustrated in Figure 11. Each lifted star induces a partition of E^{d+1} into two cones, and the upper cone can serve as a *starting cone*, whether it is convex or not. Though the starting cones are generally not all convex, perhaps many of them are "nearly" convex (especially if star splaying was preceded by flipping, applied until it got stuck). It seems a pity to throw this information away.

Consider making each starting cone convex by applying flipping to its link triangulation—and if that gets stuck, by calling star splaying recursively with the dimension d reduced by one. Call this recursive algorithm *star flipping*. Although star flipping is asymptotically slower than star splaying in the worst case, it might run faster in practice on triangulations that are close to Delaunay.

For $d = 2$, the three-dimensional cone of a vertex is represented by a one-dimensional link ring. (Because star flipping computes only the underside of H, the extended cone of a vertex is sometimes represented by an open link chain.) Figure 11 illustrates how flips in a link ring make a cone convex. Each flip replaces two edges of the link ring with one, and eliminates one reflex edge of the cone. Observe that each flip also eliminates one locally non-Delaunay edge from the star of the apex in the flattened space E^2. The algorithm that makes the cone convex is simply Graham's scan [12], which makes one speedy linear-time pass around the link ring.

For $d \geq 3$, each link triangulation is two- or higher-dimensional, and flipping might get stuck. (See Edelsbrunner and Shah [10] for an example of a two-dimensional triangulation that gets stuck during the effort to transform it into a weighted Delaunay triangulation through bistellar flips. The same example can arise in a link triangulation, even if the ultimate goal is to compute an ordinary three-dimensional Delaunay triangulation.) If this happens, the star flipping algorithm recursively invokes itself to make the cone convex. The recursion bottoms out no later than in the one-dimensional link rings, where flipping cannot get stuck.

Suppose that a flip in the link triangulation of a vertex v causes v's cone to include the downward-directed ray (parallel to the x_{d+1}-axis) with endpoint v. Then v is not on the underside of the convex hull H. The vertices that participated in the flip form a death certificate for v. Test for this circumstance after each flip.

Star flipping uses the consistency enforcement algorithm to make stars consistent with each other, but only for stars that are internally consistent and represent convex cones. Once its cone is convex, a star should never have internal inconsistencies again.

To guarantee the correct computation of a convex cone C_v with apex v, star flipping must satisfy the precondition of Theorem 3. This is more difficult than it appears: the precondition no longer refers to the vertices in V; now it refers to their projections—the intersections of the rays $\{\vec{vw} : w \in V \backslash \{v\}\}$ with a cross-sectional hyperplane. The lexicographic ordering of the intersection points is unrelated to the lexicographic ordering of the vertices in V. In general, computing a hyperplane h for which $h \cap C_v$ is bounded is a bit tricky, as it means solving a linear program. Fortunately, for a Delaunay (but not weighted Delaunay) triangulation, there is a simpler answer: choose a hyperplane tangent to the paraboloid $x_{d+1} = x_1^2 + x_2^2 + \cdots + x_d^2$ at v, move it up a bit, and use it to order the intersection points. Alternatively, if $w \in V$ is the vertex nearest v when the vertices are projected to E^d (the space of the triangulation T), then the projection of vw is a Delaunay edge, so $\vec{vw}$ is an extreme ray of C_v. Thus, w can substitute for a lexicographically minimum vertex in every starting set.

The worst-case running time of star flipping is in $O(n^d \log n)$, but this bound probably says nothing about how star flipping might perform in practice. Following the notation of Section 4.3, if $\eta_v \in O(1)$ for every vertex v, then star flipping runs in $O(n)$ time, like star splaying, because every recursive subproblem has $O(1)$ complexity. But the measure η_v does not indicate how close v's starting cone is to being convex, nor whether it will be repaired quickly.

6 Conclusions

An interesting question is whether all point location can be eliminated from the star splaying algorithm. Can consistency enforcement steps be made as local as flipping?

In dynamic mesh generation, it is desirable to change the mesh as little as possible from timestep to timestep, because each change to the mesh necessitates the reinterpolation of physical quantities, with a small concomitant loss of accuracy. An important practical problem is to extend Delaunay repair so it can repair isolated portions of a mesh, leaving untouched most regions where the element quality is still good (even if the elements are not Delaunay).

Star splaying seems promising for two other applications. One is out-of-core and parallel triangulation and meshing. Standard representations of triangulations are difficult to manipulate when they are not stored entirely in one processor's memory, because they are typically composed of large numbers of small data structures connected by interlocking pointers. A Delaunay vertex insertion operation can delete many simplices, and most of the cost of vertex insertion is identifying them. It is risky for two processors to insert vertices simultaneously, because the region they affect might overlap. It is expensive to try to identify and lock numerous tiny data structures.

Star splaying allows processors to lock and modify stars atomically. Because stars need not be mutually consistent, a processor only needs to lock two stars at a time. If two processors simultaneously insert two vertices too close together, the resulting inconsistencies between stars will at worst slow down the incremental construction of the triangulation; they are not fatal.

Another intriguing application is constructing and updating three-dimensional constrained Delaunay triangulations (CDTs). Many polyhedra, including a well-known example by Schönhardt [28], cannot be tetrahedralized without adding extra vertices. However, in a three-dimensional triangulation, the link of each vertex is a two-dimensional triangulation, and two-dimensional triangulations can be made to respect straight-line constraints. If the stars of the vertices are not required to be consistent with each other, then each vertex of Schönhardt's polyhedron can have a star that respects the polyhedron in a neighborhood of the vertex. This inconsistent star representation can support vertex insertions; and if subsequent updates (such as vertices inserted into the reflex edges of the polyhedron) make a CDT possible, the update algorithm should recover the CDT, with the stars at last consistent.

There are circumstances wherein several vertices can be simultaneously inserted into a three-dimensional CDT, but it is not possible to insert any one of those vertices alone, because a CDT of the updated domain (after just one vertex insertion) might not exist [33]. The obvious approach is to insert one vertex into the CDT at a time, and perform flipping after each insertion. But if one of the intermediate domains does not have a CDT, experience shows that even after further vertex insertions create a domain that does have a CDT, classic flipping might still get stuck at a triangulation that is not the CDT. In these circumstances, an update algorithm that maintains inconsistent stars can come to the rescue. Enticingly, these ideas may make it possible to define and maintain CDTs of three-dimensional domains with curved boundaries.

It would be valuable to extend star splaying so that it can repair CDTs. Unfortunately, CDT repair seems to be notably harder than Delaunay repair, especially in the case where a mesh has "inverted" elements (i.e. d-simplices having the wrong orientation).

Acknowledgments

My thanks go to François Labelle for debunking a previous failed attempt at these results, and to Raimund Seidel for helpful discussions and for sharing his $O(\sqrt{n})$ point location method.

References

[1] Louis J. Billera, Paul Filliman, and Bernd Sturmfels. *Constructions and Complexity of Secondary Polytopes*. Advances in Mathematics **83**(2):155–179, 1990.

[2] Daniel K. Blandford, Guy E. Blelloch, David E. Cardoze, and Clemens Kadow. *Compact Representations of Simplicial Meshes in Two and Three Dimensions*. Twelfth International Meshing Roundtable, pages 135–146, September 2003.

[3] Kevin Q. Brown. *Voronoi Diagrams from Convex Hulls*. Information Processing Letters **9**:223–228, 1979.

[4] Siu-Wing Cheng, Tamal Krishna Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. *Sliver Exudation*. Journal of the ACM **47**(5):883–904, September 2000.

[5] Kenneth L. Clarkson and Peter W. Shor. *Applications of Random Sampling in Computational Geometry, II*. Discrete & Computational Geometry **4**(1):387–421, 1989.

[6] Jesús A. de Loera, Francisco Santos, and Jorge Urrutia. *The Number of Geometric Bistellar Neighbors of a Triangulation*. Discrete & Computational Geometry **21**(1):131–142, 1999.

[7] Herbert Edelsbrunner. *Geometry and Topology for Mesh Generation*, Cambridge Monographs on Applied and Computational Mathematics, volume 6. Cambridge University Press, New York, 2001.

[8] Herbert Edelsbrunner and Ernst Peter Mücke. *Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms*. ACM Transactions on Graphics **9**(1):66–104, 1990.

[9] Herbert Edelsbrunner and Raimund Seidel. *Voronoi Diagrams and Arrangements*. Discrete & Computational Geometry **1**:25–44, 1986.

[10] Herbert Edelsbrunner and Nimish R. Shah. *Incremental Topological Flipping Works for Regular Triangulations*. Algorithmica **15**(3):223–241, March 1996.

[11] Izrail M. Gel'fand, Mikhail M. Kapranov, and Andrei V. Zelevinsky. *Discriminants of Polynomials in Several Variables and Triangulations of Newton Polyhedra*. Leningrad Mathematical Journal **2**(3):449–505, 1991.

[12] Ronald L. Graham. *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*. Information Processing Letters **1**:132–133, 1972.

[13] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. *Randomized Incremental Construction of Delaunay and Voronoi Diagrams*. Algorithmica **7**(4):381–413, 1992.

[14] Leonidas J. Guibas and Daniel Russel. *An Empirical Comparison of Techniques for Updating Delaunay Triangulations*. Proceedings of the Twentieth Annual Symposium on Computational Geometry, pages 170–179, June 2004.

[15] Barry Joe. *Three-Dimensional Triangulations from Local Transformations*. SIAM Journal on Scientific and Statistical Computing **10**:718–741, 1989.

[16] ———. *Construction of k-Dimensional Delaunay Triangulations Using Local Transformations*. SIAM Journal on Scientific Computing **14**(6):1415–1436, November 1993.

[17] Michael Kallay. *Convex Hull Algorithms in Higher Dimensions*. Unpublished manuscript, Department of Mathematics, University of Oklahoma, Norman, Oklahoma, 1981.

[18] Charles L. Lawson. *Transforming Triangulations*. Discrete Mathematics **3**(4):365–372, 1972.

[19] ———. *Software for C^1 Surface Interpolation*. Mathematical Software III (John R. Rice, editor), pages 161–194. Academic Press, New York, 1977.

[20] ———. *Properties of n-Dimensional Triangulations*. Computer Aided Geometric Design **3**:231–246, 1986.

[21] Xiang-Yang Li and Shang-Hua Teng. *Generating Well-Shaped Delaunay Meshes in 3D*. Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, pages 28–37, January 2001.

[22] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. *Simultaneous Refinement and Coarsening for Adaptive Meshing*. Engineering with Computers **15**(3):280–291, September 1999.

[23] W. B. Raymond Lickorish. *Simplicial Moves on Complexes and Manifolds*. Proceedings of the Kirbyfest (Joel Hass and Martin Scharlemann, editors), Geometry & Topology Monographs, volume 2, pages 299–320. 1999.

[24] Gary L. Miller, Dafna Talmor, and Shang-Hua Teng. *Optimal Good-Aspect-Ratio Coarsening for Unstructured Meshes*. Proceedings of the Eighth Annual Symposium on Discrete Algorithms, pages 538–547, January 1997.

[25] Francisco Santos. *A Point Configuration Whose Space of Triangulations is Disconnected*. Journal of the American Mathematical Society **13**(3):611–637, 2000.

[26] ———. *Triangulations with Very Few Geometric Bistellar Neighbors*. Discrete & Computational Geometry **23**(1):15–33, January 2000.

[27] ———. *Non-Connected Toric Hilbert Schemes*. To appear in Mathematische Annalen, 2005.

[28] E. Schönhardt. *Über die Zerlegung von Dreieckspolyedern in Tetraeder*. Mathematische Annalen **98**:309–312, 1928.

[29] Raimund Seidel. *Voronoi Diagrams in Higher Dimensions*. Diplomarbeit, Institut für Informationsverarbeitung, Technische Universität Graz, 1982.

[30] ———. *Small-Dimensional Linear Programming and Convex Hulls Made Easy*. Discrete & Computational Geometry **6**(5):423–434, 1991.

[31] Jonathan Richard Shewchuk. *Tetrahedral Mesh Generation by Delaunay Refinement*. Proceedings of the Fourteenth Annual Symposium on Computational Geometry, pages 86–95, June 1998.

[32] ———. *Sweep Algorithms for Constructing Higher-Dimensional Constrained Delaunay Triangulations*. Proceedings of the Sixteenth Annual Symposium on Computational Geometry, pages 350–359, June 2000.

[33] ———. *Updating and Constructing Constrained Delaunay and Constrained Regular Triangulations by Flips*. Proceedings of the Nineteenth Annual Symposium on Computational Geometry, pages 181–190, June 2003.

[34] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 1997. Available as Technical Report CMU-CS-97-164.