A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation

Jim Ruppert*
NASA Ames Research Center
SUBMISSION TO JOURNAL OF ALGORITHMS

February 2, 1994

Abstract

We present a simple new algorithm for triangulating polygons and planar straightline graphs. It provides "shape" and "size" guarantees:

- All triangles have a bounded aspect ratio.
- The number of triangles is within a constant factor of optimal.

Such "quality" triangulations are desirable as meshes for the finite element method, in which the running time generally increases with the number of triangles, and where the convergence and stability may be hurt by very skinny triangles. The technique we use—successive refinement of a Delaunay triangulation—extends a mesh generation technique of Chew by allowing triangles of varying sizes. Compared with previous quadtree-based algorithms for quality mesh generation, the Delaunay refinement approach is much simpler and generally produces meshes with fewer triangles. We also discuss an implementation of the algorithm and evaluate its performance on a variety of inputs.

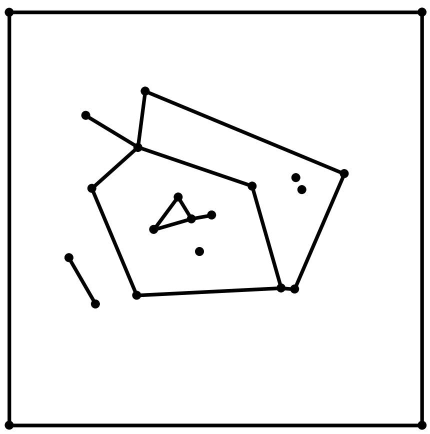
*NASA Ames Research Center, M/S T045-1, Moffett Field, CA 94035-1000. The author is an employee of the Computer Sciences Corporation. Work funded by NASA contract NAS 2-12961. Much of this work was completed while the author was a student in the Computer Science Division at the University of California at Berkeley, supported by funds from NSF PYI Grant CCR-90-58840. A portion of this work was done while the author was at Hewlett-Packard Laboratories, Palo Alto, CA. E-mail: ruppert@nas.nasa.gov.

1 Introduction

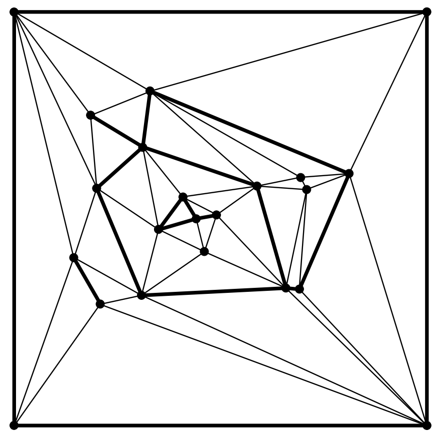
Many applications in computational geometry, graphics, solid modeling, numerical simulation and other areas require complicated geometric objects to be decomposed into simpler pieces for further processing. For instance, in the finite element method, a planar domain is divided into a mesh of elements, typically triangles or quadrilaterals. Differential equations representing some physical property such as heat distribution or airflow are then approximated using functions that are piecewise polynomial within each element. The running times of these algorithms generally depend on the *size* of the decomposition (the number of elements), hence we seek decompositions of small size. Furthermore, in many applications, the numerical stability and convergence are affected by the *shapes* of the elements; excessively "long and skinny" elements can lead to undesirable behavior.

In this paper we focus on the decomposition of 2-dimensional objects such as polygons into triangles. We will refer to this problem both as *mesh generation* and *triangulation*; it is also called *unstructured grid generation.* A triangulation must be a *simplicial complex*, that is, the intersection of any two triangles is either a common edge, a common vertex, or the empty set. *Quality mesh generation* describes techniques that offer a guarantee on some measure of shape, such as all triangles non-obtuse, or all with bounded aspect ratio. The *aspect ratio* of a triangle is the length of the longest edge divided by the length of the shortest altitude. A fairly general measure of triangle shape is the minimum angle α, since this gives a bound of $\pi - 2\alpha$ on maximum angle and guarantees an aspect ratio between $|\frac{1}{\sin\alpha}|$ and $|\frac{2}{\sin\alpha}|$. We allow triangulations to contain *Steiner points*—vertices of the mesh that are not vertices of the input—because in general they are necessary for achieving shape bounds (see Figure 1, for example). A mesh satisfying a certain shape bound is said to be *size-optimal* if the number of triangles is within a constant factor of the minimum number possible in any triangulation of the given input that meets the same shape bound.

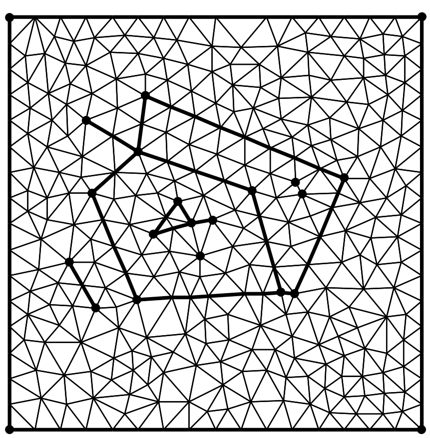
The first algorithm to give a shape guarantee was due to Baker, Grosse and Rafferty [1]. They gave a technique for producing a non-obtuse triangulation of polygons, in which all angles are at most 90°. In addition, the smallest angle is at least 13°. (Of course, this is only possible if all angles in the input are at least 13°.) Together, these bounds guarantee an aspect ratio of at most 4.6. The algorithm places a uniform square grid over the polygon,



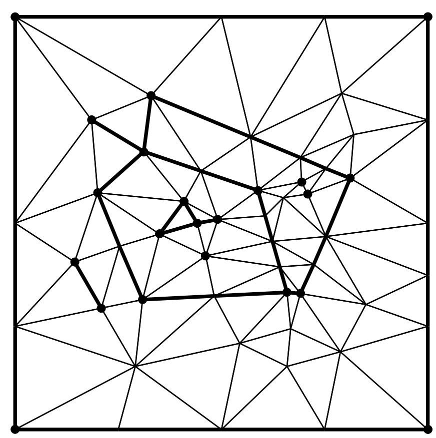
(a) Typical input PSLG and bounding box.



(b) Typical triangulation without added Steiner points. Some small angles are unavoidable.



(c) Uniform mesh with minimum angle 22.5 degrees.



(d) Output of Delaunay refinement algorithm with minimum angle 20 degrees.

Figure 1: Sample input planar straightline graph (PSLG), and several triangulations of it.

with grid spacing determined by the smallest feature present in the polygon. (Roughly speaking, the *smallest feature* is determined either by the pair of closest vertices, or by the closest vertex-edge pair, where the edge does not contain the vertex.) Since the smallest feature determines the mesh density throughout the polygon, the number of triangles can be very large.

Bern, Eppstein and Gilbert gave the first mesh generation algorithm with both shape and size guarantees [3]. They show how to triangulate polygons so that every triangle has aspect ratio at most 5. In addition, their analysis shows that the mesh is size-optimal. One of the key ideas in the algorithm is to replace the uniform grid of [1] with a *quadtree*, which is a recursive subdivision into squares of varying sizes. This yields large triangles in areas of large features. By keeping the quadtree *balanced*, aspect ratios are bounded in the output. Melissaratos and Souvaine give some extensions to the quadtree algorithm [12]. Mitchell and Vavasis show an extension of the quadtree technique to 3D [14]. They give an algorithm that uses *octrees* to produce size-optimal, bounded aspect ratio triangulations of polyhedra.

All the above techniques use grids or quadtrees. A quite different technique for quality mesh generation is *Delaunay refinement*, so-called because a Delaunay triangulation is maintained, and some criterion is used to successively pick new points to add to it. Chew [5] presented a Delaunay refinement algorithm that triangulates a given polygon into a mesh in which all angles are between 30° and 120°. The algorithm produces *uniform* meshes, meaning that all triangles are roughly the same size. The output mesh is size-optimal (to within a constant factor) amongst all uniform meshes. However, there are inputs for which a uniform mesh has many more triangles than are necessary, see Figure 1(c), for instance.

In this paper, we extend Chew's work by giving an algorithm to triangulate planar straightline graphs (PSLGs) such that all triangles in the output have angles between α and $\pi - 2\alpha$. Here α is a parameter that can be chosen between 0° and 20°. The triangles will vary in size, and the mesh will be size optimal to within a constant factor (that depends on α). PSLGs include polygons, polygons with holes, and complexes (objects made of multiple polygons); dangling edges and isolated vertices are also allowed, as shown in Figure 1(a).

Theoretically speaking, our algorithm essentially matches the PSLG algorithms of [12] and [3](modified as mentioned in [2]), but it is distinguished from them in a number of ways: (1) The Delaunay refinement approach is fundamentally different from the quadtree techniques. (2) It is much simpler. With fewer special case constructions, it is easier to implement. (3) It generally produces fewer triangles in practice. (4) It is "parameterized": the user can ask for the "best" mesh with a given number of triangles. In this way, the algorithm takes advantage of the inherent mesh size/shape tradeoff. (5) The output mesh has no favored orientation. In contrast, grid or quadtree based meshes produce many mesh edges aligned with the coordinate axes. Such alignment may affect subsequent computation. (6) Delaunay refinement can be modified to generate a mesh unique to the input, independent of the orientation of the input. (This assumes careful handling of degeneracies, and elimination of the bounding box, as described in Section 5.)

A few words about the input to the algorithm: The input can be any planar straightline graph (PSLG), with dangling edges and isolated points allowed (see Figure 1(a)). As shown in the figure, the algorithm will triangulate a larger region, out to an enclosing box. To get a triangulation of a particular region, say the interior of a polygon, exterior triangles can be removed. (To maintain the size optimality guarantee in this case, the algorithm must be modified slightly, as discussed in Section 5.)

Though the algorithm is based on the Delaunay triangulation, the *constrained* Delaunay triangulation [11],[4], might be a worthwhile alternative. We discuss this briefly in Section 5, and Chew discusses its use in a related mesh generation algorithm [6].

The remainder of this paper is organized as follows. In the next section we present the algorithm. Then we show that it halts and outputs a valid triangulation satisfying the minimum angle bound. We define the *local feature size* at each point in the input, and bound the output size in terms of it. Then we show that every triangle is within a constant factor of the largest possible at that point, which proves size-optimality. To balance the theoretical results, we then discuss some issues that arise in implementing the algorithm, and describe our own implementation. Through a variety of examples, we evaluate its performance in terms of mesh size and shape.

Large portions of the work reported here have appeared elsewhere in preliminary form [16], [17], [18].

2 The Delaunay Refinement Algorithm

The basic idea of the algorithm is to maintain a triangulation, making local improvements in order to remove the skinny triangles. Each improvement involves adding a new vertex to the triangulation and retriangulating. To pick good locations for these new vertices, we use the following fact of elementary geometry:

Fact 1 If triangle $T = abc$ has $\angle bca = \theta$, and p is the circumcenter of T, then $\angle bpa = 2\theta$. (See Figure 6.)

This fact can be proved by considering the angles of the triangles pab, pbc and pca. The *circumcenter* of a triangle is the center of the unique circle through the three vertices of the triangle. As described below, we will generally be adding vertices that are circumcenters, though when such locations are unsuitable, we will instead place new vertices on the input segments.

The particular triangulation we maintain is a Delaunay triangulation, which has been extensively discussed in the literature (see, e.g., [15] or [9]). We recall the definition: given a finite set of points in the plane, three points contribute a triangle to the Delaunay triangulation if the circumcircle through those points contains no other point in its interior. This definition produces a unique triangulation, assuming the appropriate handling of degeneracies (4 or more co-circular points).

Edges of the input PSLG will be referred to as *segments* to distinguish them from the *edges* of the Delaunay triangulation that is maintained. Also, a *vertex* is a vertex of the input or of the growing Delaunay triangulation, whereas a *point* is any point in the plane. During the course of the algorithm, we will maintain a set V of vertices (initialized to the vertices in the input) and a set S of segments (initially those in the input). Vertices are added to the Delaunay triangulation $\mathcal{DT}(V)$ for two reasons: to improve triangle shape, and to insure that all input segments are present in $\mathcal{DT}(V)$ (as the union of one or more Delaunay edges).

The two basic operations in the algorithm are to *split a segment* by adding a vertex at its midpoint, and to *split a triangle* with a vertex at its circumcenter. In each case, the new vertex is added to V; when a segment is split, it is replaced in S by its two subsegments.

For a segment s, the circle with s as a diameter is referred to as its *diametral circle*, and we say that a vertex *encroaches upon* segment s if it

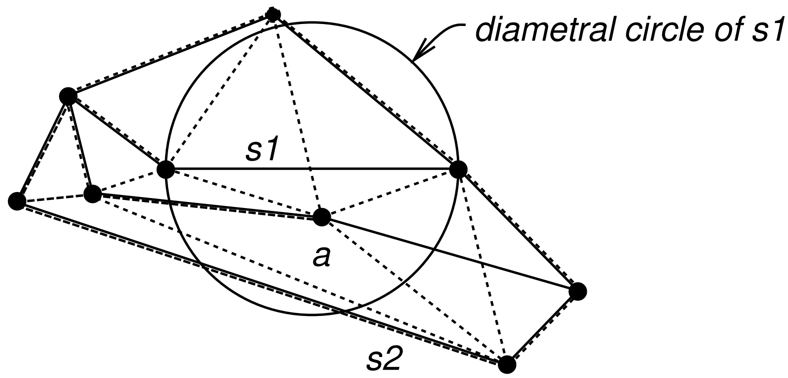


Figure 2: Input PSLG shown in solid lines, Delaunay triangulation of its vertices shown dotted. This is not a valid triangulation of the PSLG because s_1 is not present as a Delaunay edge. Vertex a "encroaches upon" both segments s_1 and s_2.

lies within the diametral circle of s. Figure 2 illustrates this: the vertex a encroaches upon both segments s_1 and s_2 (only s_1's diametral circle is shown). A segment that is encroached upon may or may not be present in the Delaunay triangulation; it is easy to show that any segment not present in the Delaunay triangulation is encroached upon by some vertex.

To simplify the description and analysis of the algorithm, we assume for now that all angles of the input PSLG are at least 90°. In Section 5, this restriction will be removed.

Any triangle with an angle below α is called *skinny*. In essence, the algorithm says to split skinny triangles, unless the triangle's circumcenter would encroach upon some input segment, in which case split the segment instead. Here is the algorithm in detail, including subroutines for the two basic operations:

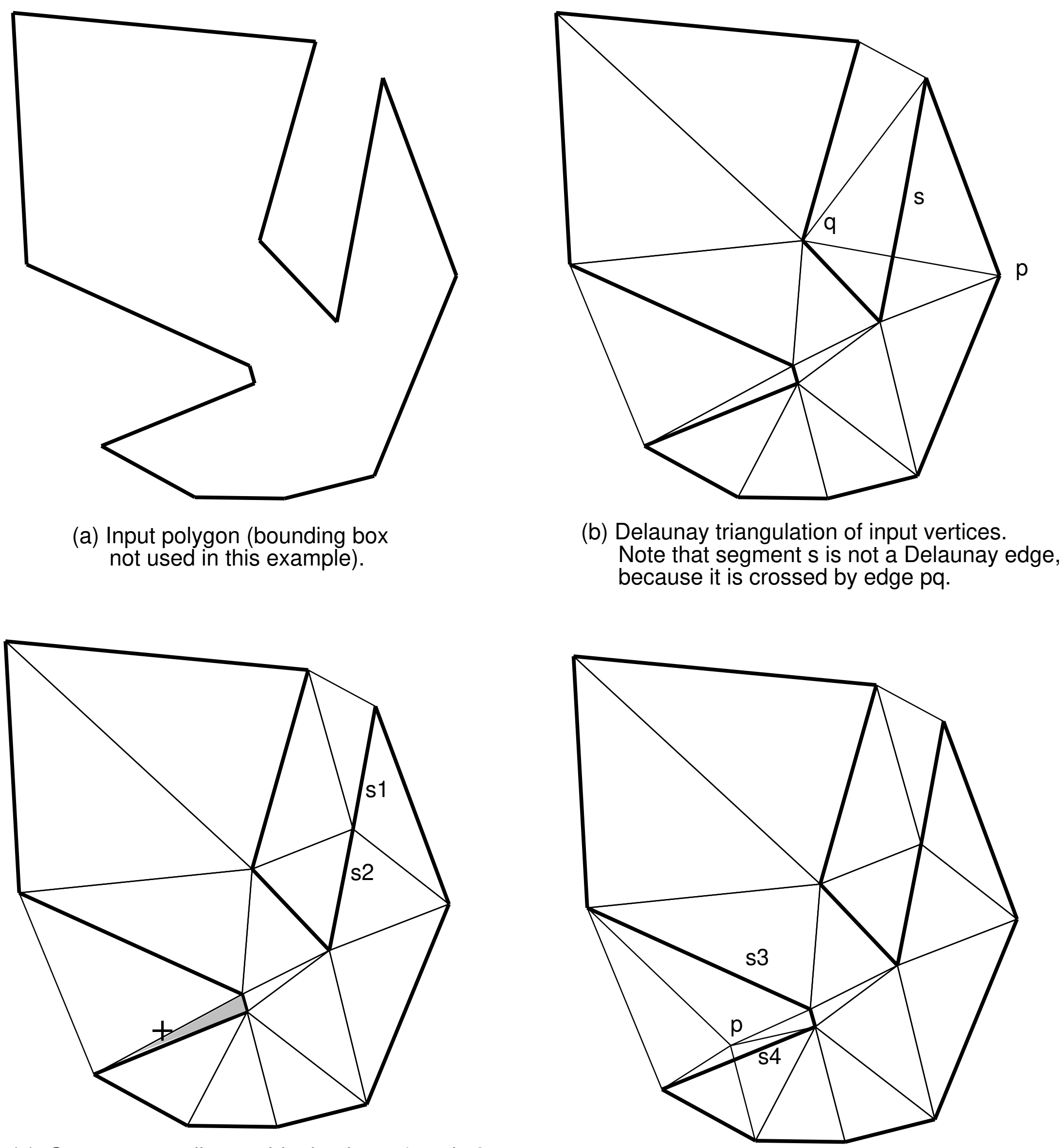
subroutine `SplitTri(triangle` t`)`
    `Add circumcenter of` t `to` V`, updating` $\mathcal{DT}(V)$

subroutine `SplitSeg(segment` s`)`
    `Add midpoint of` s `to` V`, updating` $\mathcal{DT}(V)$
    `Remove` s `from` S`, add its two halves` s_1 `and` s_2 `to` S

Algorithm *DelaunayRefine*

```
INPUT:    planar straightline graph X;
          desired minimum angle bound α
OUTPUT:   triangulation of X, with all angles ≥ α.
Initialize:
   add a bounding square B to X:
      compute extremes of X:  xmin, ymin, xmax, ymax
      let span(X) = max(xmax-xmin, ymax-ymin)
      let B be the square of side 3 × span(X), centered on X
      add the four boundary segments of B to X
   let segment list S = edges of X
   let vertex list V = vertices of X
   compute initial Delaunay triangulation DT(V)
repeat:
   while any segment s is encroached upon:
      SplitSeg(s)
   let t be (any) skinny triangle (min angle < α)
   let p be t's circumcenter
   if p encroaches upon any segments s_1,...,s_k then
      for i = 1 to k:
         SplitSeg(s_i)
   else
      SplitTri(t) (⋆ adds p to V ⋆)
   endif
until no segments encroached upon, and no angles < α
output current Delaunay triangulation DT(V)
```

Figures 3 and 4 show the execution of the algorithm on a simple polygonal example. For clarity, no bounding box is used. In each picture, the input is shown in thick lines, the current Delaunay triangulation is overlayed in thin lines. (The observant reader might notice a slight enhancement in the algorithm used in the example: if a segment s is encroached upon by a vertex on another segment, s does not have to be split as long as it appears in the triangulation, and no skinny triangles are present. For instance, the vertex



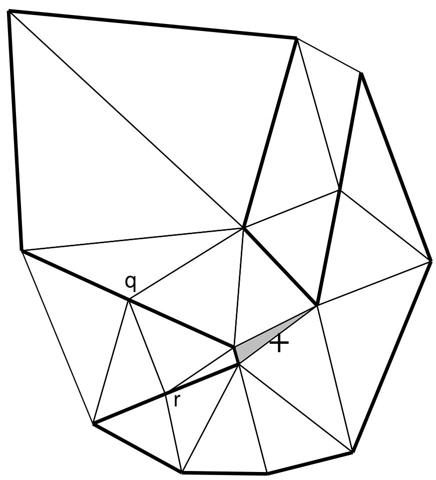
(a) Input polygon (bounding box not used in this example).

(b) Delaunay triangulation of input vertices. Note that segment s is not a Delaunay edge, because it is crossed by edge pq.

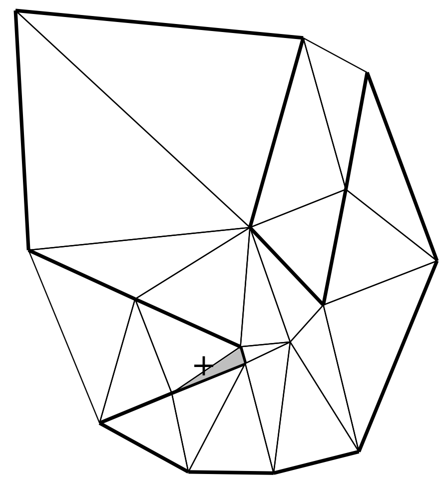
(c) Segment s "split" at midpoint, into s1 and s2. Shaded triangle has smallest angle, 5.9 degrees. Cross indicates its circumcenter.

(d) If circumcenter p were added, it would encroach upon segments s3 and s4.

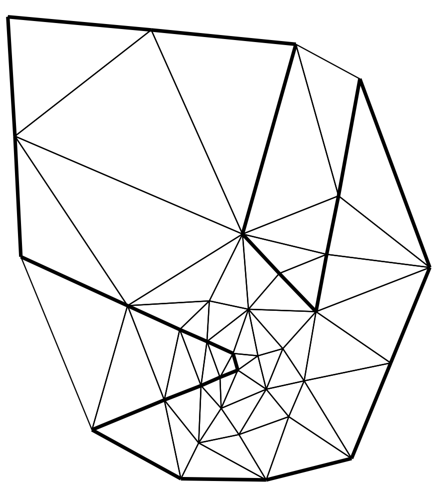
Figure 3: Execution of the algorithm on a simple example. For clarity, bounding box not used.



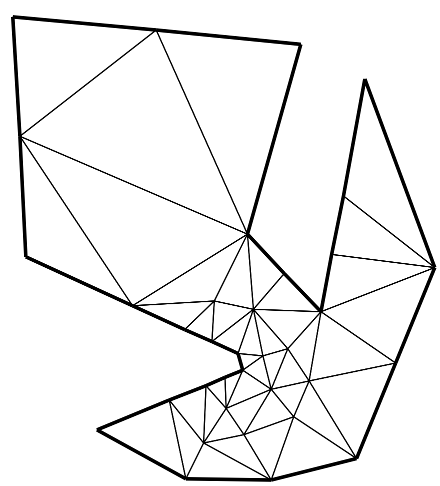
(e) 2 segments were split at q and r.
Shaded triangle now has minimum angle,
9.8 degrees, and will be split.



(f) New minimum angle 11.6 degrees.



(g) Result after allowing execution to
continue until minimum angle is
at least 25 degrees.



(h) Optionally, external triangles can
be removed.

Figure 4: Continuation of example. In this case, minimum angle $\alpha = 25°$.

added between (b) and (c) encroaches upon two segments that are never split.)

In the next section, we show that the algorithm halts for any $\alpha < 20°$. (In practice, larger values can be chosen, up to $\alpha \approx 30°$.) Upon termination, all triangles will have aspect ratios at most $|\frac{2}{\sin\alpha}|$, since all angles smaller than α will have been removed. Furthermore, all input segments will be present in the output (as the union of one or more Delaunay edges), since any segments missing from the Delaunay triangulation are encroached upon, and hence get split until they are present. Note that the algorithm allows skinny triangles to be split in any order. We discuss good orderings and other implementation issues in Section 6.

3 Output Size

In this section we give an upper bound on the number of triangles in the output. The bound depends upon the *local feature size* of the input. At every point in the mesh, the vertex spacing will be close to the local feature size. In the next section, we will show that the local feature size is indeed the desired spacing, since it yields meshes within a constant factor of the optimal size.

Definition 1 *Given a PSLG X, The* **local feature size** *at a point p,* $\mathbf{lfs}_X(p)$*, or simply* $\mathbf{lfs}(p)$*, is the radius of the smallest disk centered at p that intersects 2 non-incident vertices or segments of X.*

Figure 5 illustrates the definition of $\mathit{lfs}(\ )$, the radius of the disk D_i being $\mathit{lfs}(p_i)$. Note D_3 in particular: a smaller disk would intersect 2 segments, but they are incident to each other.

For a given input X, $\mathit{lfs}(p)$ is defined for all points p in the plane, and the entire function, which we refer to as $\mathit{lfs}(X)$, is continuous. If $\mathit{lfs}(p)$ is interpreted as an elevation at p, then $\mathit{lfs}(X)$ is a "not-too-steep" surface above the plane. The following lemma shows that it has a Lipschitz condition of 1, i.e. the slope in any direction is at most 1.

Lemma 1 *Given any PSLG X, and any two points p and q in the plane,*

$$\mathit{lfs}(q) \leq \mathit{lfs}(p) + \mathrm{dist}(p, q),$$

where $\mathrm{dist}(p, q)$ *is the Euclidean distance between p and q.*

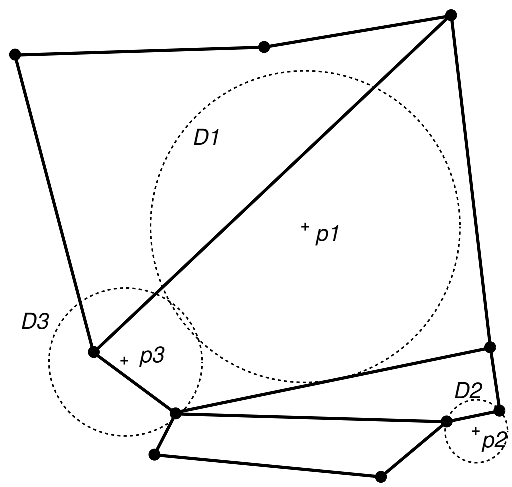


Figure 5: Local feature size at several points. Radius of disk D_i is $lfs(p_i)$.

Proof: The disk D of radius $r = lfs(p)$ centered at p must intersect 2 non-incident portions of X, by definition of $lfs(\ )$. The disk D' of radius $r' = r + \mathrm{dist}(p, q)$ centered at q contains D and hence intersects the same portions of X. So $lfs(q) \leq r'$. Putting this together, we have

$$lfs(q) \leq r' = r + \mathrm{dist}(p, q) = lfs(p) + \mathrm{dist}(p, q).$$

■

The next lemma is the crux of the mesh size analysis. It shows that as each vertex is added, it is at the center of a "vertex-free" circle of radius at least a constant fraction of the local feature size. Thus the density of added vertices is bounded by the geometry of the input. We emphasize that adding vertices does not change the $lfs(\ )$ function, since it is determined by the input.

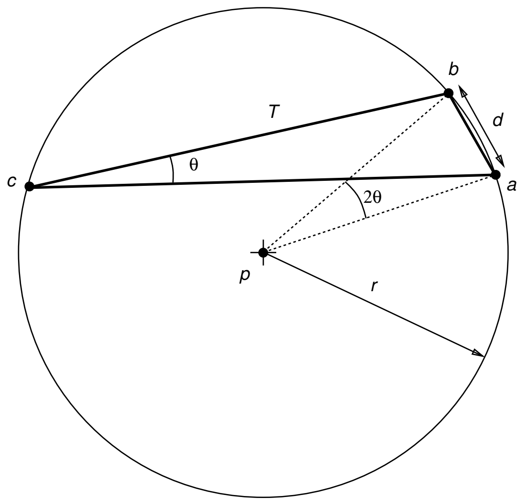


Figure 6: Lemma 2 Case 1: p added as circumcenter of triangle T with small angle $\theta < \alpha$.

Lemma 2 *For fixed constants C_T and C_S, specified below, the following statements hold:*

- *At initialization, for each input vertex p, the distance to its nearest neighbor vertex is at least $lfs(p)$.*
- *When a point p is chosen as the circumcenter of a skinny triangle, the distance to the nearest vertex is at least $\frac{lfs(p)}{C_T}$. (p may be added to the triangulation, or may be rejected because it encroaches upon some segment.)*
- *When a vertex p is added as the midpoint of a split segment, the distance to its nearest neighbor vertex is at least $\frac{lfs(p)}{C_S}$.*

Proof: For any input vertex p, the distance to its nearest neighbor vertex is at least $lfs(p)$, by definition of the $lfs(\ )$ function. This is the base case of the lemma. For vertices added later, we assume the lemma is true for all previous vertices.

Case 1: We first consider the case where p is the circumcenter of a skinny triangle T. Since p is at the center of T's Delaunay circle, its nearest neighbors are the vertices of T (see Figure 6), at a distance of r. Assume the vertices of T are a, b, c, with the smallest angle θ at c. Then the shortest edge of T is from a to b. Call its length d. Without loss of generality, assume a was added after b (or that both were in the input). We will use the fact that a and b are close together to bound $lfs(a)$ in each of several cases, which in turn will bound $lfs(p)$.

> Case 1(a): a was a vertex of the input. Then so was b, so $lfs(a) \leq d$.
>
> Case 1(b): a was added as a circumcenter of some triangle with circumradius $r' \leq d$ (since b was outside that triangle's circumcircle). We can apply this lemma to a, yielding $lfs(a) \leq r'C_T \leq dC_T$.
>
> Case 1(c): a was the midpoint of a segment that was split. Applying this lemma to a now yields $lfs(a) \leq dC_S$, since b was outside a's vertex-free circle.

So we have $lfs(a) \leq dC_S$, assuming we have the condition $\boxed{C_S \geq C_T \geq 1}$, which we will be able to satisfy below. By Fact 1, $\angle apb = 2\theta$, so simple geometry gives $d = 2r \sin\theta$. Lemma 1 gives

$$lfs(p) \leq lfs(a) + r$$

using our bound for $lfs(a)$ we have

$$lfs(p) \leq dC_S + r$$

$$= 2rC_S \sin\theta + r$$

or, since $\theta < \alpha$,

$$r \geq \frac{lfs(p)}{1 + 2C_S \sin\alpha}$$

So we get the desired bound on r as long as we can satisfy the condition $\boxed{C_T \geq 1 + 2C_S \sin\alpha}$.

Case 2: We now consider the case where a vertex p is added to split a segment s. Segment s is split because some vertex or circumcenter a is inside s's diametral circle, which has radius r. (See Figure 7.) We have two cases for a:

Case 2(a): a lies on some segment t, which cannot be incident to s, since we are assuming that all angles in the input PSLG are at least 90°. (Any

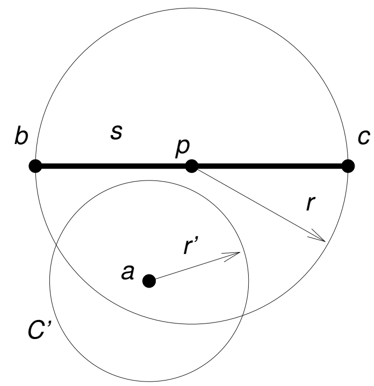


Figure 7: Lemma 2 Case 2: p added to split segment s which is encroached upon by a.

segment incident to s makes a larger angle, and hence would be completely outside the diametral circle.) So there are two non-incident segments, one containing p, the other containing a, within distance r of each other. Thus $\mathit{lfs}(p) \leq r$. Above, we have assumed the condition $C_S \geq 1$, so this case is done.

Case 2(b): a was a circumcenter, proposed for addition to the Delaunay triangulation, but rejected because it lay inside the diametral circle of s. Suppose it was the center of circle C' with radius r'. By applying this lemma to a, we know that $r' \geq \frac{\mathit{lfs}(a)}{C_T}$. Also, b and c, the endpoints of S, must be outside the Delaunay circle C', so $r' \leq \sqrt{2}\, r$. Lemma 1 gives

$$\begin{aligned} \mathit{lfs}(p) &\leq \mathit{lfs}(a) + r \\ &\leq r' C_T + r \\ &\leq \sqrt{2}\, r C_T + r \end{aligned}$$

or

$$r \geq \frac{\mathit{lfs}(p)}{1 + \sqrt{2} C_T}$$

This yields the correct bound on r, provided that $\boxed{C_S \geq 1 + \sqrt{2} C_T}$. ■

It can be checked that the 3 boxed conditions can be simultaneously satisfied for any $\alpha < \arcsin \frac{1}{2\sqrt{2}} \approx 20.7°$. For instance, $C_T = \frac{1+2\sin\alpha}{1-2\sqrt{2}\sin\alpha}$,

$C_S = \frac{1+\sqrt{2}}{1-2\sqrt{2}\sin\alpha}$ will work. For $\alpha = 10°$, we can choose $C_T = 2.8$, and $C_S = 5$.

Since $C_T \leq C_S$, the lemma shows that when a vertex p is added, no other vertex is within distance $\frac{lfs(p)}{C_S}$ of p. The following theorem shows that vertices added later cannot get much closer to p.

Theorem 1 *Given a vertex p of the output mesh, its nearest neighbor vertex q is at a distance at least $\frac{lfs(p)}{C_S+1}$.*

Proof: Lemma 2 handles all but the case when q was added after p, in which case we can apply the lemma to q and get

$$\mathrm{dist}(p,q) \geq \frac{lfs(q)}{C_S}$$

Lemma 1 gives a bound for $lfs(q)$ in terms of $lfs(p)$ and q's distance from p, so

$$\mathrm{dist}(p,q) \geq \frac{lfs(p) - \mathrm{dist}(p,q)}{C_S}$$

rearranging finishes the proof: $\mathrm{dist}(p,q) \geq \frac{lfs(p)}{C_S+1}$ ■

The next theorem uses an area argument to yield a bound on the number of vertices. Intuitively, a region of small local feature size requires small triangles, i.e. the vertex spacing should be proportional to the local feature size. Thus the triangle density in the mesh is proportional to the inverse of the square of the local feature size. So we will "charge" the cost for each vertex to the local feature size around it.

Theorem 2 *The number of vertices in the output mesh is at most*

$$C_1 \int_B \frac{1}{lfs^2(x)} dx,$$

where B is the region enclosed by the bounding square, and C_1 is a constant to be specified.

Proof: The previous theorem says that each vertex p in the mesh is at the center of an open disk of radius $\frac{lfs(p)}{C_S+1}$ that contains no other vertex. Halving the radii gives non-intersecting disks: let D_p be the open disk of radius $r_p = \frac{lfs(p)}{2(C_S+1)}$ centered on p. Since at least one-fourth of each D_p is contained in the bounding square B, we get a lower bound for the integral by summing its value in the disks D_p for every p in the vertex set V:

$$\int_B \frac{1}{lfs^2(x)}dx \geq \frac{1}{4}\sum_{p\in V}\int_{D_p}\frac{1}{lfs^2(x)}dx$$

By Lemma 1, the maximum $lfs(\ )$ attainable in D_p is $lfs(p) + r_p$, which gives a bound for $\int_{D_p}$:

$$\int_{D_p}\frac{1}{lfs^2(x)}dx \geq \text{area}(D_p)\frac{1}{\max_{x\in D_p}\{lfs^2(x)\}}$$

$$\geq \text{area}(D_p)\frac{1}{(lfs(p)+r_p)^2}$$

Using $\text{area}(D_p) = \pi {r_p}^2$, plugging in for r_p, and cancelling yields

$$\int_{D_p}\frac{1}{lfs^2(x)}dx \geq \frac{\pi}{(2C_S+3)^2}$$

Substituting back in for the entire integral,

$$\int_B \frac{1}{lfs^2(x)}dx \geq \frac{1}{4}\sum_{p\in V}\frac{\pi}{(2C_S+3)^2}$$

$$= \frac{\pi}{4(2C_S+3)^2}\sum_{p\in V} 1$$

Since the summation merely counts the number of vertices in the output mesh, the theorem holds if we choose the constant $C_1 \geq \frac{4(2C_S+3)^2}{\pi}$. ■

4 Size-Optimality

Our goal in this section is to show that any triangulation produced by the Delaunay refinement algorithm is size-optimal, meaning that the number of triangles is within a constant factor of the minimum number possible. We first state and prove some properties that any bounded aspect ratio triangulation must have, and then use these properties to show that even the optimal triangulation is not too much better than the output of the Delaunay refinement algorithm. The following properties of bounded aspect ratio triangulations are seemingly obvious, but a number of technical details are required to state and prove them precisely:

- Small input features will be surrounded by proportionally small triangles.
- Nearby triangles have similar sizes.
- The size variation between distant triangles depends on their distance.

The basic idea would be to show that in an optimal mesh, triangle sizes must vary slowly, proportional to the local feature size measure. Since this was the case for Delaunay refinement meshes as well, we could show that they are within a constant factor of optimal. The difficulty in using this approach directly is that triangle size is a step function: size is constant within each triangle, but large discontinuities are possible between triangles, especially near mesh vertices of high degree. To cope with this, we must define precisely what we mean by "triangle size", and show that though it is a step function, it is reasonably well-behaved. With a series of lemmas, we bound the maximum triangle size at an arbitrary point, and show that triangle sizes within a Delaunay refinement mesh are within a constant factor of the largest possible.

The analysis in this section is similar to that given by Mitchell and Vavasis for their 3D algorithm [14]. A basic notion in their proof is that of a "characteristic length function", which defines the "triangle size" at every point within the triangulation:

Definition 2 *If a point p is contained in a triangulation $\mathcal{T}$ of input PSLG X, then we say the* **edge length** *at p, $el_{\mathcal{T},X}(p)$, or simply $el(p)$, is the length of the longest edge among all triangles of $\mathcal{T}$ containing p.*

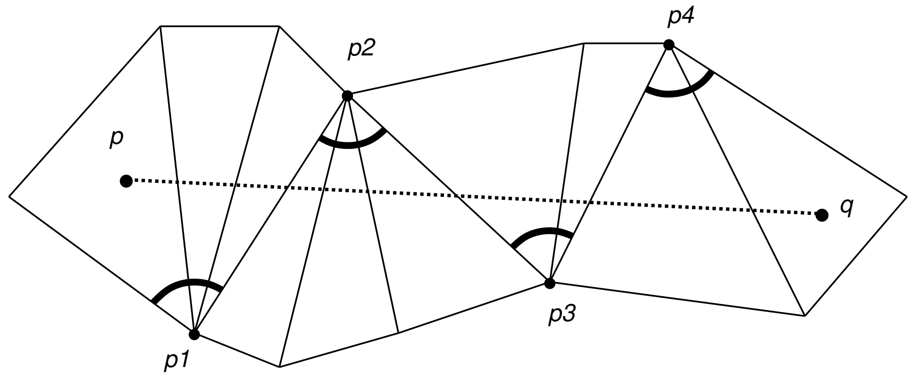


Figure 8: Triangles crossed by the segment from p to q are divided into "fans".

We shall now prove some properties about this function within bounded aspect ratio triangulations. For the remainder of this section, we assume that all triangles have a minimum angle bound of α, which guarantees all aspect ratios are at most $A = \frac{2}{\sin\alpha}$. By considering the shared edge between two triangles, we have the following:

Fact 2 If p and q lie in the interiors of distinct triangles T_p and T_q, which share an edge, then $\frac{el(q)}{el(p)} \leq A$.

Repeated use of this fact gives the following lemma about points in arbitrary triangles:

Lemma 3 *If p and q lie in the interiors of triangles T_p and T_q, respectively, then*

$$el(q) \leq C_2 \cdot el(p) + C_3 \cdot dist(p, q)$$

where C_2 and C_3 are constants to be specified.

Proof: Consider the sequence of triangles crossed by the line segment from p to q, as shown in Figure 8. (Here we are assuming that the triangulation fills a convex region, so that the segment stays within the triangulation.) Any vertex on the segment is treated as though it were to the "right" of the directed segment from p to q. Label any vertices shared by more than two consecutive triangles $p_1, p_2, \cdots$. The "zigzag" edges connecting successive p_i's

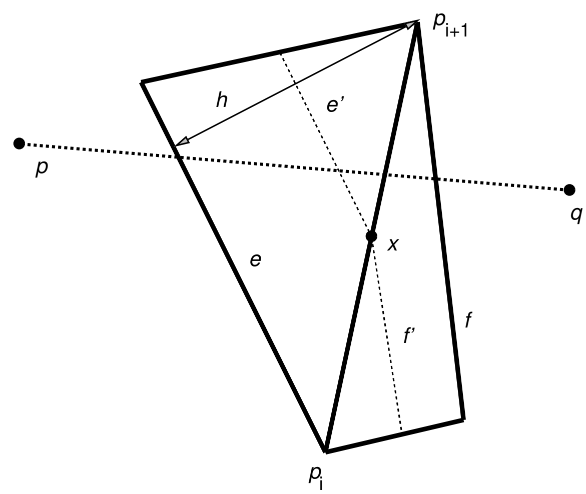


Figure 9: Since line segment pq crosses edges e and f, it must cross e' or f'.

divide the triangles into "fans" around each p_i, indicated by the bold arcs in Figure 8. Since at most $\lfloor \frac{2\pi}{\alpha} \rfloor$ triangles fit around a vertex, each fan contains at most $K = \lfloor \frac{\pi}{\alpha} \rfloor$ triangles, except the first and last, which may contain $K + 1$. We consider two different cases, depending upon the number k of triangles between p and q, including T_p and T_q.

<u>Case 1:</u> $k \leq K + 3$: Using k applications of Fact 2, we see that the lemma holds as long as $C_2 \geq A^{K+3}$ and $C_3 \geq 0$.

<u>Case 2:</u> $k > K + 3$: Since zigzag edges are separated by at most K triangles, there exists some zigzag edge $p_i p_{i+1}$ that is flanked by two triangles, neither of which contains p or q. Consider the closest such edge to q. In Figure 8, this is the edge $p_3 p_4$. Figure 9 shows edge $p_i p_{i+1}$ and its two flanking triangles. We now show that the length of the segment pq is at least half of an altitude of one of these triangles. Let e and f be the two outer edges crossed by pq, as shown. Let x be the midpoint of $p_i p_{i+1}$, and construct e' and f' through x and parallel to e and f, as shown. Since pq crosses e and f, it must cross either e' or f'. If it crosses e', then it is longer than half the altitude h from edge e to vertex p_{i+1}, i.e. $\mathrm{dist}(p, q) \geq \frac{h}{2}$. By the definition of aspect ratio, the longest edge of the triangle containing e' has length at most $A \cdot h$. Thus for any point p' within that triangle we can use $K + 1$ applications of Fact 2 to show that $el(q) \leq A^{K+1} el(p')$. It then follows that

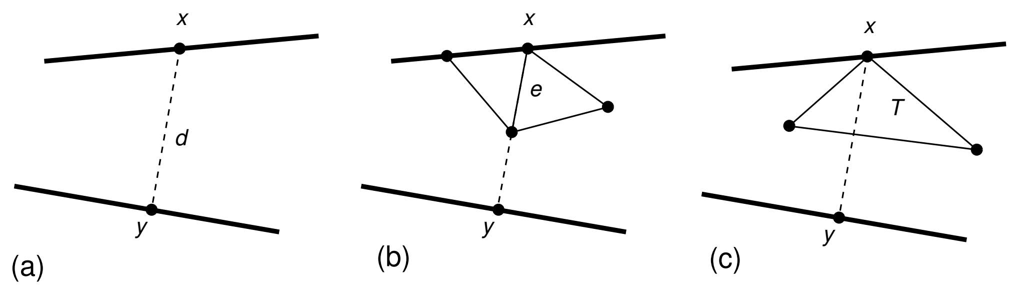


Figure 10: Triangle size along xy if x is a mesh vertex.

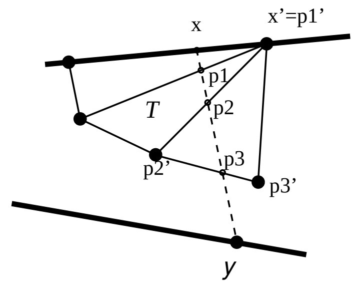


Figure 11: Triangle size along xy if x is not a mesh vertex.

$el(q) \leq A^{K+2}h \leq 2A^{K+2}\text{dist}(p, q)$. The lemma holds for $C_3 = 2A^{K+2}$, since $el(p) > 0$, $C_2 > 0$. The case where f' is crossed by pq is handled similarly, within the same bound. The choice of $C_2 \geq A^{K+3}$, $C_3 \geq 2A^{K+2}$ satisfies all the conditions. ■

Lemma 3 gives a bound on how fast edge lengths can change in a bounded aspect ratio triangulation. The following lemma shows that there must be small triangles near small input features.

Lemma 4 *Let x and y be points (not necessarily endpoints) of non-incident input segments. Let d be the distance between x and y. Then, in any triangulation with aspect ratios bounded by A, there is a point p on the line segment connecting x and y with $el(p) \leq 2d \cdot A$.*

Proof: See Figure 10(a).

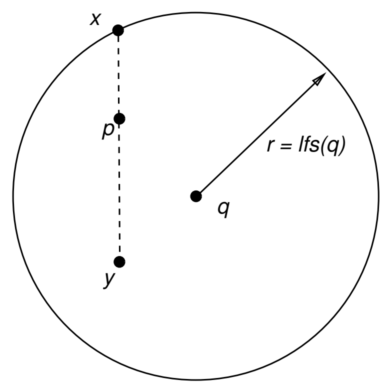


Figure 12: Points x and y on input segments determine edge length $el(p)$ at some point p along xy, and local feature size $lfs(q)$.

Case 1: The easy case is when x or y is a vertex of the triangulation. Without loss of generality, suppose x is a vertex. If there is a triangulation edge at x along the line segment xy (see Figure 10(b)), then that edge has length at most d. For any point p in the interior of the edge, $el(p) \leq d \cdot A$. If the line segment xy is in the interior of some triangle T near x (see Figure 10(c)), then the minimum altitude of T is no longer than T's intersection with xy, so for any point p on xy interior to T, $el(p) \leq d \cdot A$.

Case 2: This case, where x is in the interior of an edge of the triangulation, is illustrated in Figure 11. Let x' be the endpoint nearest x, and consider all triangles incident to x' that intersect the line segment xy. Let $p_1, p_2, \ldots$ be the intersections of xy with the edges of these triangles, as shown. Finally, on the edge containing p_i, label the endpoint closest to p_i as p_i'. Let j be the smallest index such that $p_j' \neq x'$. Such a j exists, because eventually xy will reach an edge not incident to x', for instance the edge containing y. By an argument similar to that used in Case 2 of the previous lemma, we see that the minimum altitude of T, the triangle containing p_{j-1} and p_j is at most $2d$. Then for any point p in the interior of the line segment $p_{j-1}p_j$, $el(p) \leq 2d \cdot A$.
■

Next, we use the preceding lemmas to relate the edge length function $el(\ )$ to our local feature size measure $lfs(\ )$. Recall that we are assuming $\mathcal{T}$ is any triangulation in which all angles are at least α.

Lemma 5 *At any point q in the interior of a triangle of $\mathcal{T}$, $el(q) \leq C_4 lfs(q)$, where C_4 is a constant to be specified.*

Proof: By definition, $lfs(q)$ is the radius r, determined by two points x and y on non-incident segments of the input (see Figure 12). From Lemma 4, there must be some point p along xy with $el(p) \leq 2 \cdot \mathrm{dist}(x, y) \cdot A$. Since we always have $\mathrm{dist}(x, y) \leq 2r$, $el(p) \leq 4r \cdot A$. Using $\mathrm{dist}(p, q) \leq r$ and Lemma 3,

$$
\begin{aligned}
el(q) &\leq el(p) + C_2 \cdot el(p) + C_3 \cdot \mathrm{dist}(p, q) \\
&\leq (C_2 + 1) \cdot el(p) + C_3 \cdot r \\
&\leq (C_2 + 1) \cdot 4r \cdot A + C_3 \cdot r \\
&\leq [(C_2 + 1) \cdot 4A + C_3] \cdot r \\
&\leq [(C_2 + 1) \cdot 4A + C_3] \cdot lfs(q)
\end{aligned}
$$

Choosing $C_4 \geq (C_2 + 1) \cdot 4A + C_3$ concludes the proof. ∎

We can now state and prove the major result of this section: that the mesh output by the Delaunay refinement algorithm is size-optimal to within a constant factor. First we recall the situation: the input is a planar straight-line graph X with all angles at least $90°$, $\alpha \leq 20°$ is the minimum angle bound for the output, which guarantees all triangles have aspect ratio at most $\frac{2}{\sin \alpha}$. The algorithm triangulates the region inside $B(X)$, a larger bounding box of X, and the optimality is with respect to any triangulation of $B(X)$ with minimum angle bound α. (The $90°$ input restriction, and the requirement that the mesh triangulates $B(X)$, will be removed in the next section.)

Theorem 3 *Given $\alpha \leq 20°$, and input X, suppose $\mathcal{T}$ is any triangulation of X with minimum angle bound α. There is a constant C_α such that if $\mathcal{T}$ has N triangles, then the Delaunay refinement triangulation $\mathcal{T}_D$ has $N_D \leq C_\alpha \cdot N$ triangles. Letting $\mathcal{T}$ be the triangulation with fewest possible triangles shows that $\mathcal{T}_D$ is within a factor C_α of optimal.*

Proof: Theorem 2 bounds the number of vertices in the Delaunay refinement triangulation $\mathcal{T}_D$. In any triangulation, the number of triangles is at most twice the number of vertices (true by Euler's relation, see [15], p. 19). Thus $\mathcal{T}_D$ has

$$N_D \leq 2C_1 \int_B \frac{1}{lfs^2(x)} dx$$

triangles. By Lemma 5 this is

$$\leq C_1 \cdot {C_4}^2 \int_B \frac{1}{el^2(x)} dx$$

where the edge-length function $el(\ )$ is with respect to $\mathcal{T}$. (Strictly speaking, Lemma 5 does not apply to edges of the triangulation, but since they have measure 0, they do not contribute to the integral.) We can instead sum the integrals over each triangle $T \in \mathcal{T}$:

$$= C_1 \cdot {C_4}^2 \sum_{T \in \mathcal{T}} \int_T \frac{1}{el^2(x)} dx$$

In each triangle T, $el(\ )$ is constant, just the length of the longest edge. The area of T is at most $\frac{\sqrt{3}}{4} el^2(\ )$, which occurs if T is equilateral. So for T we have

$$\int_T \frac{1}{el^2(x)} dx \leq \frac{\frac{\sqrt{3}}{4} el^2(x)}{el^2(x)} = \frac{\sqrt{3}}{4}$$

Substituting back in,

$$N_D \leq C_1 \cdot {C_4}^2 \, \frac{\sqrt{3}}{4} \sum_{T \in \mathcal{T}} 1$$

We have $\sum_{T \in \mathcal{T}} 1 = N$, since the summation just counts the number of triangles in $\mathcal{T}$. Thus the theorem holds for $C_\alpha = C_1 \cdot {C_4}^2 \, \frac{\sqrt{3}}{4}$. ■

The constant factor C_α depends on the choice of α, but not on X, i.e. the Delaunay refinement algorithm is optimal on every input, not just in the worst case. We discuss C_α more in Section 6.

5 Corner-Lopping and Riemann Sheets

Two issues must be resolved so that the algorithm produces size-optimal bounded aspect ratio triangulations for general 2-dimensional inputs. First,

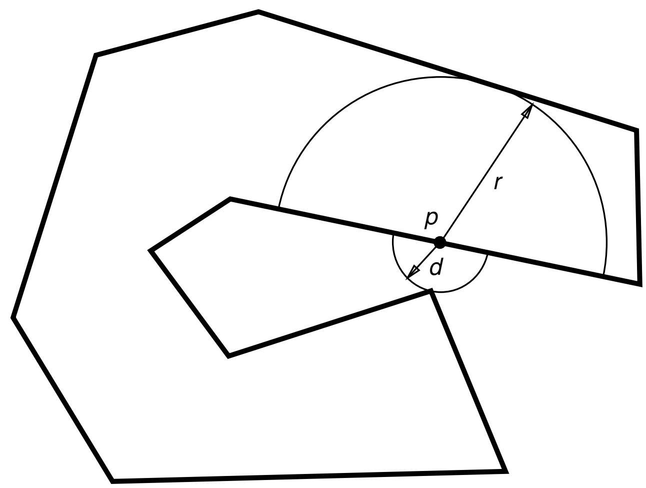


Figure 13: Do the two "arms" of the polygon determine a small feature at p?

we must deal with small input angles reasonably (recall that we made the unreasonable assumption that all angles were at least 90°!). The second issue is subtler, and relates to our definition of local feature size in non-convex polygons: in Figure 13, do the two "arms" of the polygon generate a small feature at p? Our definition says they do, and produces small triangles around p accordingly. This could be suboptimal if only an interior triangulation of the polygon is desired. Fortunately, previous researchers have dealt with both of these concerns, and we can adapt their solutions to our algorithm. These modifications may increase the size of the mesh, but by at most a constant factor.

We handle small angles by "lopping off" the sharp corners during a preprocessing step. For the time being, we continue to assume that all input angles are at least α, the desired minimum output angle. Below, we will mention the case of smaller input angles.

Any input vertex p with a small angle is "shielded" by committing in advance to a specific triangulation around p. Previous approaches surrounded each vertex with a circle [3], or with a cube in 3D [14]. We will sketch how to do this with a circle here. First, for every input vertex p, the local feature size $lfs(p)$ is computed. For an input PSLG of size n, this can easily be done in $O(n^2)$ time by computing all vertex-vertex and vertex-edge pairwise distances. At any vertex p with an angle smaller than 90°, we will intersect

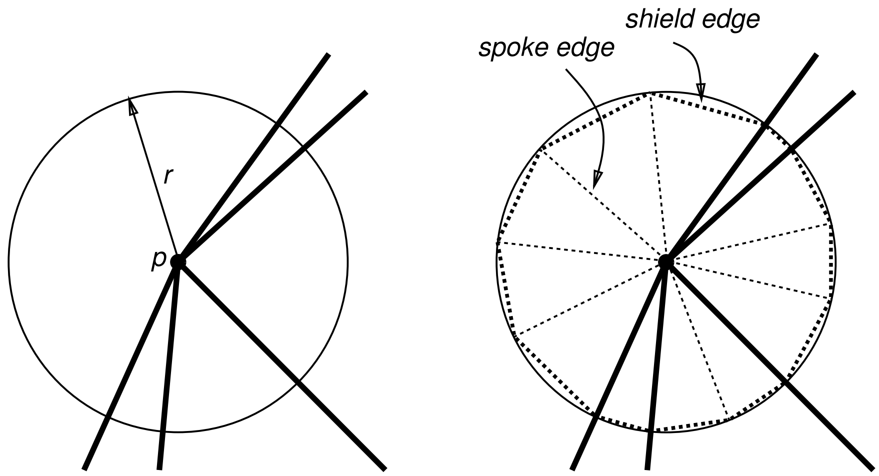


Figure 14: "Lopping off" sharp corners with a shielding circle (input segments in bold).

a circle with the input edges, as shown in Figure 14. The radius of the circle will be $\frac{lfs(p)}{3}$, so that circles around different vertices do not intersect or get too close. Angles at p greater than 2α are divided so as to be between α and 2α, introducing *shield edges* around the circle, and *spoke edges* from p to the circle. These edges, and new vertices, are henceforth considered as part of a modified input PSLG X', and will appear in the output mesh. This reduces the local feature size in X' compared with X, but only by a constant factor that depends on α.

In this modified input, all angles outside the shielding circles will be at least $90°$. Within the shielding circles, our hope is to use the triangles shown in Figure 14 as the output triangles around p. If we disallow the splitting of such triangles, then no vertices will be added within the shielding circle, but still the shield edges may get split, as shown in Figure 15(a). When the algorithm terminates, each shield edge will be split into at most a constant number of pieces, since the local feature size along the edge is proportional to the edge length.

We now have two ways of dealing with split shield edges. Following [14], we place edges between the split vertices and p, as shown in Figure 15(b). (In fact, these will be present as the Delaunay edges.) Since each shield edge is split a constant number of times, the minimum angle in the output mesh

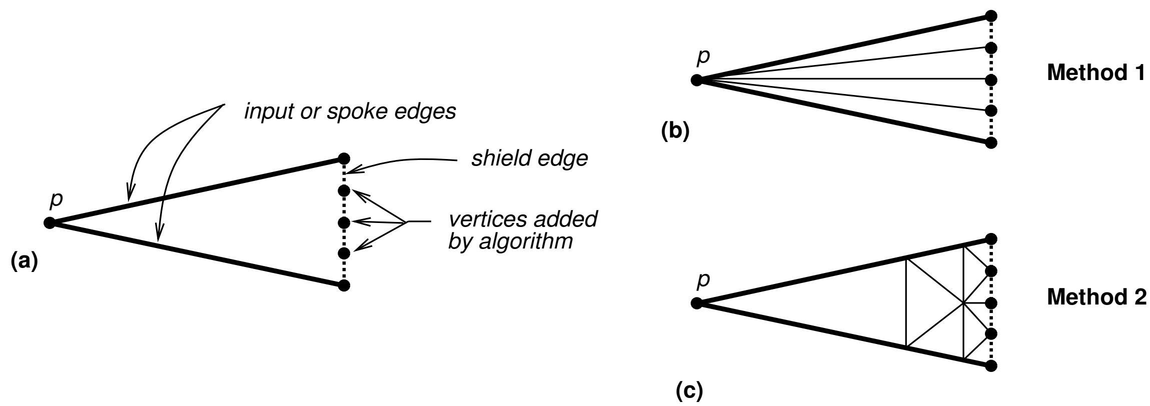


Figure 15: Fixing shield edges that get split.

will be within a constant factor of α, while retaining the constant factor optimality for the mesh size.

Next we sketch a more complicated construction that can avoid splitting the smallest input angle at all. As shown in [3], a construction like that in Figure 15(c) will work for polygon inputs. The idea is that since each shield edge is split at most a constant number of times, we can use a constant number of layers to "merge" the triangles together.

This construction does not work directly for PSLG inputs, since the layers of triangles require vertices to be added to a spoke edge, and the triangulations on either side of the spoke may not agree along the spoke. A construction as shown in Figure 16 will work for $\alpha \leq 10°$. From Section 3, the value of C_S tells us that each shield edge will be split into at most 8 pieces, so at most 3 layers of triangles will be necessary. Roughly speaking, we will "strengthen" our shielding circle by replacing it with 3 layers of triangles, each of a fixed width. The figure shows only the outer layer, of width d, which depends on α. The triangulation within each layer depends on how the shield edge is split. The two most difficult cases are shown in Figure 16. If a shield edge spanning an angle of α is split into 8 pieces, then we use "merging" triangles that are skinny in the radial direction, whereas an unsplit shield edge spanning an angle close to 2α requires a triangle that is skinny in the circumferential direction. Simple calculations show that d can be chosen to balance these two cases so that all angles will be at least α. The other cases and inner layers are similar, and easier to handle. It is

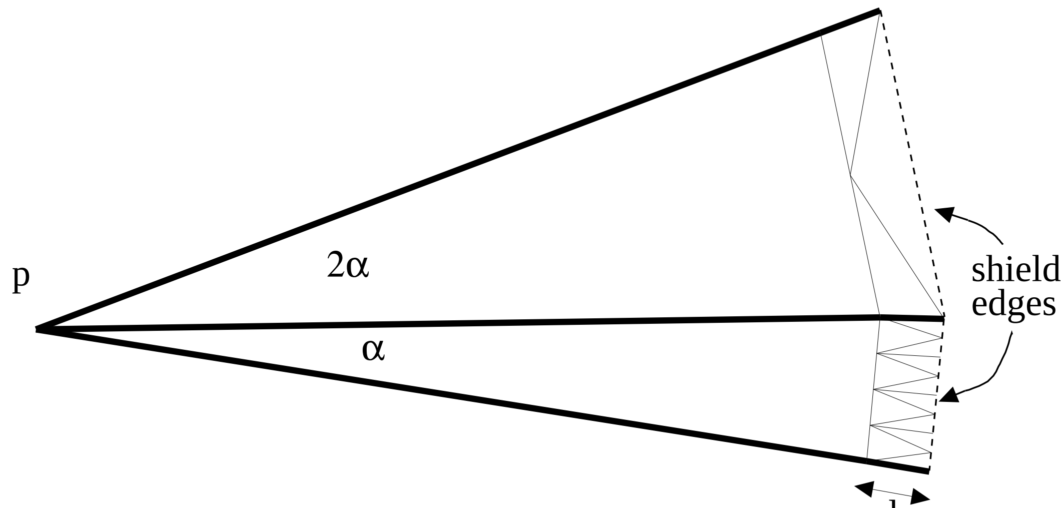


Figure 16: Construction for PSLGs.

not clear how to perform this construction for values of α much larger than $10°$. The problem is that the value of C_S grows, and hence it cannot be guaranteed that the shield edges will be split into a small number of pieces. Another issue is the handling of input angles less than the desired minimum α. For polygons with very small input angles, a construction like that of Figure 15(c) can be used to produce a triangulation in which all angles are greater than α, except for the smaller input angles (which obviously cannot be removed). For PSLGs, it appears quite difficult to handle input angles below α in such a graceful fashion.

In practice, these intricate constructions do not seem to be necessary in order to handle small input angles, as discussed in Section 6.

The second issue is illustrated in Figure 13. We assumed the input was a planar straightline graph, and produced a triangulation that extended out to a larger surrounding box. If only the interior of a polygon is to be triangulated, then we would not consider the clearance between the two "arms" of the polygon in Figure 13 as a small feature. In particular, the local feature size at p should be r, rather than d, as our definition states. Following [14], we modify the definition to use the *geodesic distance* to the 2 nearest non-incident portions of the input. The geodesic distance is measured along the shortest path that stays within the region to be triangulated (e.g. the interior of the polygon). The algorithm is modified to use the *constrained*

Delaunay triangulation (CDT) [11],[4]. The CDT can be computed with the *Riemann sheet* technique of Seidel [20]. This corresponds to Mitchell and Vavasis' use of Riemann volumes for octree mesh generation [14]. Chew has recently described a related mesh generation algorithm that uses the CDT [6].

Using the CDT, one can develop a modified Delaunay refinement algorithm to produce meshes that are unique (for a given α) and independent of the orientation of the input. The CDT allows the algorithm to work without a bounding box; for uniqueness it is also necessary to specify the order in which skinny triangles and encroached edges are split, for instance one could always split the triangle with the largest circumcircle, breaking ties according to vertex indices. In the case of degeneracies (4 or more co-circular points), the Delaunay triangulation is not uniquely defined; this can also be disambiguated with vertex indices.

6 Implementation and Discussion

The pseudocode algorithm given in Section 2, which we call the *basic algorithm*, could be implemented in many ways. In our case, there were two main goals: to allow interactive experimentation with the algorithm, and to produce sample outputs for evaluating its performance. In this section, we discuss some general implementation issues, and describe our own implementation, as well as a variety of possible modifications and enhancements to the basic algorithm. We will also evaluate the algorithm's practical performance with respect to mesh size and shape.

The basic algorithm was quite simple to implement, requiring only a small amount of work beyond the computation of a Delaunay triangulation. The corner-lopping and Riemann sheet modifications mentioned in the previous section were not implemented. Though corner-lopping was required for the theoretical analysis in the case of small input angles, it would require a large implementation effort. Instead, we used a simpler approach that works very well in practice. The implementation runs reliably on many examples, and produces meshes in which the number of triangles seems to be quite reasonable. In practice, the algorithm easily achieves a minimum angle bound of 20°, and can be run longer if desired, though it rarely improves the minimum angle much above 30°. Figure 17 shows an example of its output, given as

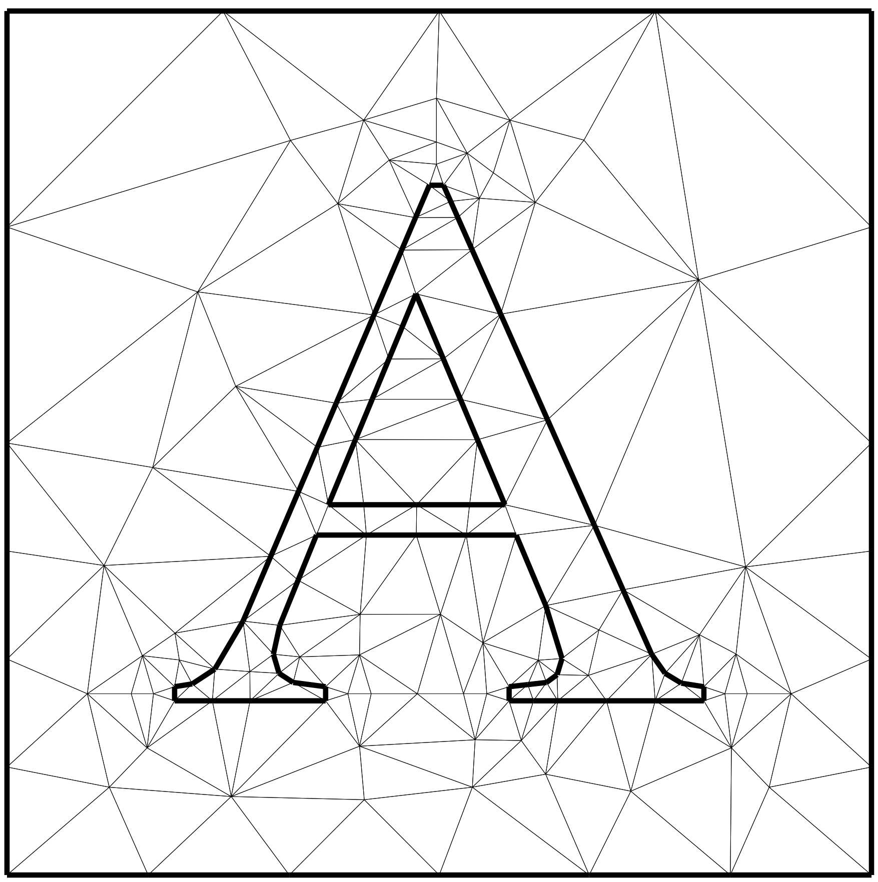


Figure 17: Triangulation produced by the Delaunay refinement algorithm for a sample input, shown in bold. Minimum angle $\approx 20°$.

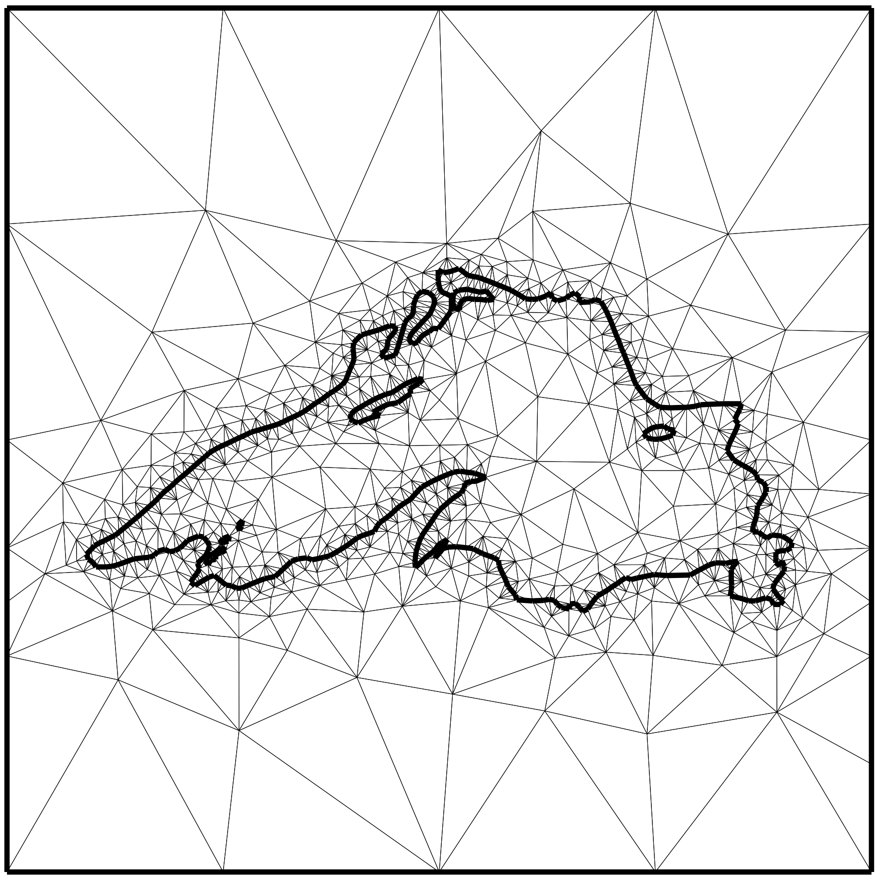


Figure 18: Triangulation of the boundary of Lake Superior. Minimum output angle $\approx 15°$.

input an outline of the letter "A" and a minimum angle bound $\alpha = 20°$. Many of the vertices along the outline were added by the algorithm.

An important choice to be made in any implementation is the Delaunay triangulation algorithm to build upon. There are *incremental* algorithms where points are added one at a time to a growing triangulation, e.g. [10], and *all-at-once* algorithms such as the sweepline algorithm of Fortune [9]. An incremental algorithm seems the logical choice, since our algorithm refines the triangulation by adding points. (Though an all-at-once algorithm could be used for the initial triangulation, and perhaps for splitting segments, since it is possible that many segments may simultaneously be encroached upon.) In our case, efficiency was not an issue, because we merely needed to interactively experiment with the algorithm on relatively small meshes. An implementation of the sweepline algorithm was available, so we have used it throughout, even for incremental addition of a single point to the triangulation. For meshes with several hundred vertices, a full recomputation of the Delaunay triangulation takes less than a second, which is sufficient for interactive use. However, the algorithm would be excessively slow for practical usage on large inputs, so we do not report any timing measurements.

We have not analyzed the asymptotic running time of the Delaunay refinement algorithm in detail. The worst-case running time for incremental Delaunay triangulation is $O(M^2)$, where M is the output size. In practice, such algorithms usually run much faster [10]. Much of the time is typically taken up locating the triangle containing the added point. For non-input vertices, this is simplified in our algorithm by starting at the skinny triangle or encroached upon segment being split.

Figure 18 shows another output of the algorithm, given as input an approximate outline of Lake Superior, including several islands. Since the boundary was represented by roughly equally spaced points, most of the points added by the algorithm were interior points (in the lake, or between the lake and the bounding box). We note that the input contains an angle close to 15°, which was not a problem even though the corner-lopping step was not done.

The basic algorithm of Section 2 says that *any* skinny triangle may be split, though it seems like a good idea to split the triangle with the globally minimum angle. In this way, the algorithm is parameterizable, meaning that it can be halted just as soon as all angles are "large enough". This modification comes at a slight cost, however, since skinny triangles must be maintained in a priority queue to allow the globally minimum angle to be efficiently determined. If we choose to split an arbitrary skinny triangle, then only a list is needed.

The detection of "encroached upon" segments (those containing a point in their diametral circle) can be done efficiently by checking local criteria during each update of the Delaunay triangulation. A segment is encroached upon if either:

1. It is not present as a Delaunay edge (e.g. s_1 in Figure 2), or

2. It is present, but opposite an obtuse angle in a Delaunay triangle (e.g.

s_2 in Figure 2).

Though the basic algorithm specifies a square bounding box 3 times as large as the input, any constant multiple will work. For clarity in our examples, we have used a smaller bounding box. The bounding box has both a theoretical and a practical purpose. Whereas a polygon clearly has an interior, a PSLG input may have dangling edges, and it is not always clear exactly what region is to be triangulated. The convex hull of the PSLG is a logical candidate, but then an input vertex just inside the hull could generate a "small feature" that is not really present in the input. The bounding box gives an unambiguous region to be triangulated, without reducing the local feature size by more than a constant factor. An axis-aligned bounding square also improves the algorithm's robustness, since splitting an edge of the box gives a midpoint which is truly collinear with the endpoints. Otherwise, if roundoff were to occur, then the midpoint could fall inside the edge, causing a very skinny Delaunay triangle to form between the midpoint and the endpoints. The calculation of such a triangle's circumcenter is very ill-conditioned.

Since the base algorithm of Section 2 is so simple, it is easy to experiment with alternative criteria for splitting triangles. For instance, some applications, such as error adaptive solvers, have a maximum desired triangle size. Figure 1(c) shows how to achieve this as well as a minimum angle bound: we change the criterion to split triangles that are skinny **or** large, where *large* means having a circumradius larger than a fixed bound. One can also exclude certain triangles from being split, for instance, in Section 5 triangles were not split if their small angle was part of the input. In Section 2 we also mentioned a situation in which an encroached upon segment need not be split if the encroaching vertex lies on some other input segment. Finally, the user may wish to eliminate large angles, but allow small angles. As discussed in [3], this can generally be done with fewer triangles than in the no-small-angle case. Unfortunately, we cannot take advantage of this fact by modifying the basic algorithm to split triangles with large angles, because every triangle with an angle above $\pi - 2\alpha$ also has an angle below α, and the behavior of the modified algorithm approximates that of the original.

Another variation to the basic algorithm is to split triangles at points other than their circumcenters, and to split segments at points other than their midpoints. For instance, in Figure 6 we saw that the angle opposite ab

doubles if it is moved to the circumcenter. The angle would increase further if p were closer to ab, though possibly at the expense of other triangles. We have not explored this approach, but perhaps it could increase the global minimum angle more rapidly, and reduce the overall number of triangles. Below we discuss how splitting segments at non-midpoints can help in handling small input angles. It would be nice to extend the size-optimality proof to cover these different split points. One would need to show that the proof holds for split points "near" circumcenters and midpoints. We have not done this, but it seems possible, perhaps with weaker optimality constants and minimum angle bounds.

Mesh Size

Next we take several different approaches to evaluating the size of meshes produced by the Delaunay refinement algorithm. We argue that the algorithm performs significantly better than other algorithms with mesh shape guarantees, and somewhat worse than a human might do. We also argue that the analysis of Section 4 gives a gross overestimate of the algorithm's behavior in practice.

Upon close examination of Figures 17 and 18, one sees many places where moving a vertex could improve triangle shapes, or where a vertex could be removed without decreasing the minimum angle. We might estimate these triangulations to be within a factor of 2–5 times the minimum possible size for the given angle bound. Thus the "true" size-optimality constant for the Delaunay refinement algorithm lies somewhere between 2 and the value of C_α of Section 4. Plugging a minimum angle bound of $\alpha = 20°$ into the inequalities given in Section 4, we get a bound $C_\alpha \approx 1.81 \times 10^{25}$. Though this is the first explicitly stated optimality constant for a bounded aspect ratio triangulation algorithm, the value is clearly meaningless as a practical guarantee. Examination of the analysis shows much slack that might be tightened, for example a constant of A^{2K+6}, with $A \approx 6$, $K \approx 4$, that we suspect can be replaced by 2^K or A^2, but even a reduction of 10 or 15 orders of magnitude would not yield a useful value for C_α. One would really like a stronger proof technique.

We can make a non-rigorous argument about output size using the constant C_S of Section 4. It bounds the density of points along input segments, and its value indicates that at most 5 "layers" of triangles will appear be-

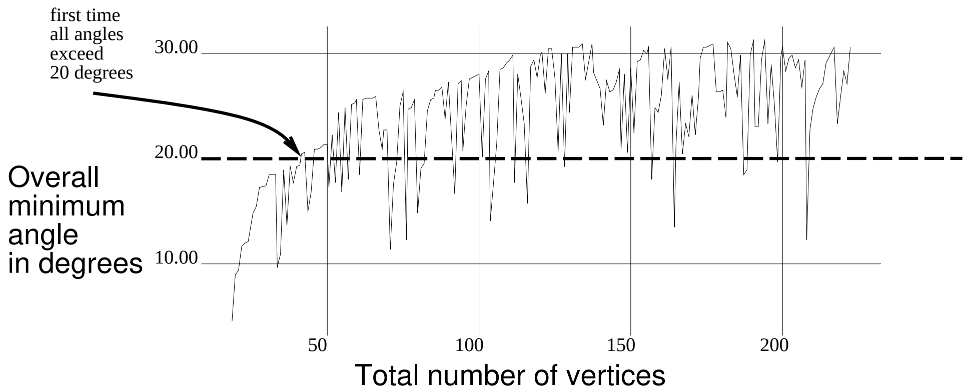


Figure 19: Progress of minimum angle during a typical run.

tween 2 nearby input vertices. In Figure 17, we see that short segments are not broken up at all, and so there is usually only 1 layer. This contrasts with the algorithm in [3], in which each input vertex must be isolated within a 5-by-5 grid of quadtree squares, yielding at least 2–3 layers of triangles between any two vertices. For instance, whereas the Delaunay refinement algorithm would triangulate a square using 2 triangles, the quadtree algorithm would need 18 triangles, or more, depending on the orientation of the square.

Additional evidence concerning the behavior of the Delaunay refinement algorithm comes from Figure 19, which charts the overall minimum angle during a lengthy run on a simple input with about 15 vertices. We see the minimum rise to about 30° and then level off, except for frequent downward spikes when a small angle gets divided in two, then quickly improved. The optimality proof says that eventually, no spike will drop below the dotted line (here, for $\alpha = 20°$), which would be far to the right of the plotted portion of the graph. The arrow points out when the algorithm would actually halt for this case.

The reason for the non-monotone behavior of the minimum angle in Figure 19 is shown in Figure 20. Delaunay triangle T_1 has the minimum angle, and T_2 has a slightly larger angle. The splitting of T_1 causes the small angle in T_2 to be cut, greatly reducing the global minimum.

Next we consider a class of inputs that can be used to benchmark a quality mesh generation algorithm's performance at *grading*, or adapting to very small input features, using as few well-shaped triangles as possible. The

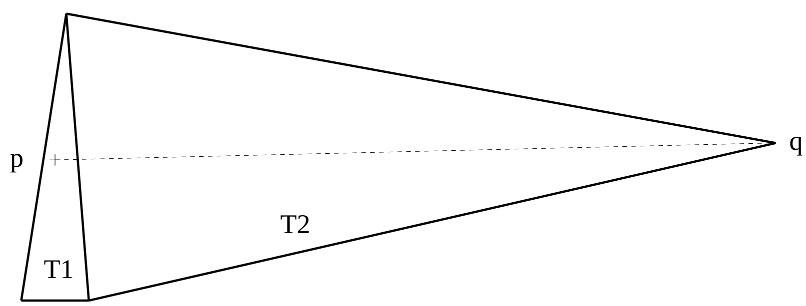


Figure 20: This example shows that splitting a skinny triangle can decrease the global minimum angle. T_1 and T_2 are two triangles in a larger Delaunay triangulation. The cross indicates T_1's circumcenter p, which happens to lie within T_2's circumcircle. If T_1 is split at p, the dotted edge pq may cut the small angle at q in half.

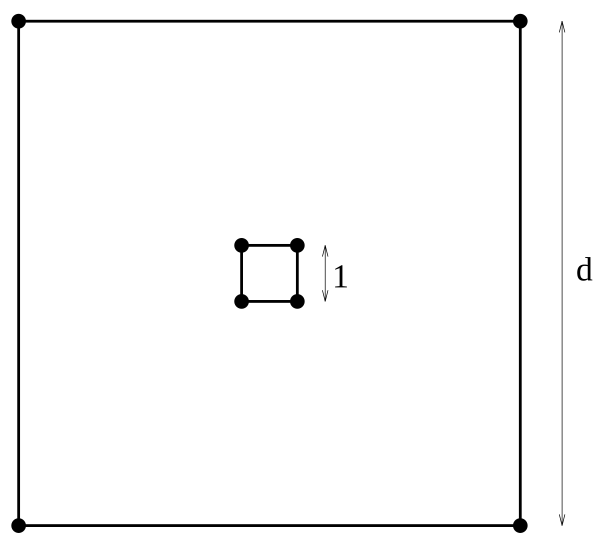


Figure 21: Input for "grading" benchmark.

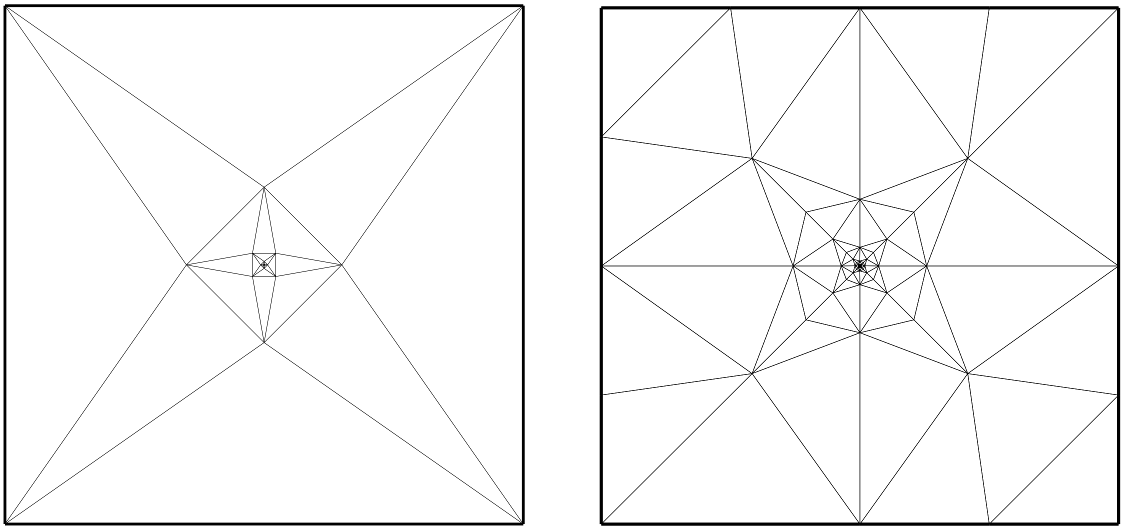


Figure 22: Mesh grading test case with minimum angle 20 degrees. Manual construction on the left, Delaunay refinement output on the right.

input consists of two squares, a unit square centered within a larger square of side d, where d can vary. From Theorem 3 of Section 4, we expect that any size-optimal quality mesh generation algorithm will produce a mesh in which the number of vertices is proportional to the logarithm of d. We are interested in the constants of proportionality, and we will compare the Delaunay refinement algorithm against an "optimal" (or at least very good) mesh created manually.

The manual mesh construction is shown on the left in Figure 22, with the unit square in the center. To each edge of the unit square we attach an isosceles triangle with a single 20° angle opposite the edge of the square, creating a four-pointed star. To this we add four more triangles to fill out a square, rotated 45° from the original. This construction is repeated out to the desired size, though only certain values of d are achievable. For comparison, the output of the Delaunay refinement algorithm is shown on the right for the same test case. Given a minimum angle bound of 20°, meshes produced by the Delaunay refinement algorithm have about 4 times as many vertices as does the manual construction, as shown by the graph in Figure 23. The values for the Delaunay refinement algorithm were computed from four experimental runs, and the performance of the manual case is derived from

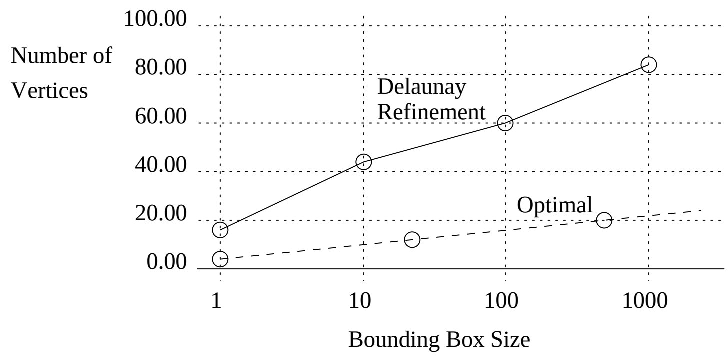


Figure 23: Comparison of mesh sizes for the algorithms shown in Figure 22.

the ratio $\frac{\cos 35°}{\sin 10°} \approx 4.72$ of sizes of successive squares in the construction, since each square adds four vertices.

The two methods may be even a bit closer in performance, since the comparison is performed on those values of d best for the manual construction, and because the Delaunay refinement generally does a bit better than the minimum angle bound (it achieved between 23° and 27° for the four cases in Figure 23). A further question is whether the manual construction is indeed the optimal strategy for this test case. (Asymptotically, it appears that a construction with three-way symmetry allows more rapid grading: start with an equilateral triangle, add an isosceles triangle with a 20° angle opposite each edge, add three more triangles to fill out a larger equilateral triangle, repeat.) If indeed the manual construction is close to optimal, then we have further evidence that there is much slack in the bound for C_α, since that bound essentially depends on the possibility of much more rapid grading than seems possible even in a manual construction.

Small Input Angles

Next we take up the issue of small angles in the input. In Sections 2-4, we assumed that all input angles were at least 90° (and at most 270°). This was necessary in order to prove the termination and size bounds of the algorithm. In Section 5, we showed how this restriction could be removed by using a preprocessing step called "corner-lopping" to isolate small angles. Though the corner-lopping served a theoretical purpose, there are several objections

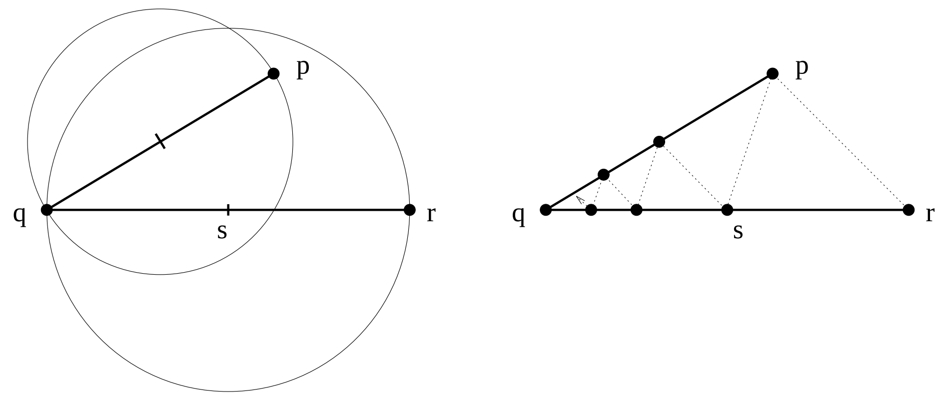


Figure 24: This shows why small input angles can be a problem for the basic algorithm: p encroaches upon qr, which is split at s, which encroaches upon qp, which is split ...

to using it in practice: implementing it would require a large effort, and it might be overkill, producing more triangles than are really necessary. Here we argue that with a simple modification to the basic Delaunay refinement algorithm, small input angles rarely cause problems in practice, and that the few remaining problematic cases can be handled by adding a simplified version of corner-lopping (though we have not been able to prove size-optimality for this modified algorithm).

Figure 24 shows why small input angles can be a problem. Angle pqr is less than 45°, and p "encroaches" upon qr, since it is within the diametral circle of qr. Hence, segment qr is split, by adding a vertex at its midpoint. This new vertex then encroaches upon segment pq. After splitting pq, we repeat the initial situation, at a smaller scale. As shown on the right, this process of splitting encroached segments can continue indefinitely. (No such examples are known with angles greater than 45°.) This problem is caused by the definition of encroachment, and can usually be avoided in practice by modifying the definition to exclude cases where the encroaching vertex lies on another input segment. However, this does not handle the case of a high-degree vertex, due to the interaction between edges incident to the vertex (this problem was described in a slightly different setting by Saalfield [19]).

In practice, to handle the high-degree case, and cases with very small

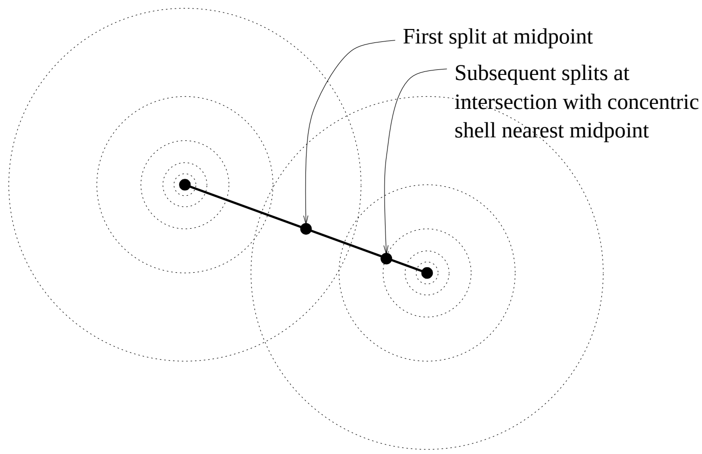


Figure 25: Modified segment splitting using concentric shells.

input angles, we use a method that attempts to simulate corner-lopping. This is a heuristic that uses "concentric circular shells" around each input vertex. The idea is that instead of splitting all segments at their midpoints, which can cause splits to bounce back and forth between segments as in Figure 24, we make the split points "line up" with each other. The approach is illustrated in Figure 25. We imagine that each input vertex is surrounded by concentric circles, each double the radius of the one inside it. The first time a segment is split, the midpoint is used, but from then on, any subsegment with an input vertex as an endpoint is split at the intersection of the segment with the circle nearest the midpoint of the segment.

Specifically, choose D to be a fixed, arbitrary constant (such as 0.01), and let the ith shell have radius $D2^i$, for all integers i. Suppose we must split a subsegment pq, where p is an input vertex. Let the length of pq be $2d$. Then the midpoint of pq has the non-integral shell number $k = \log_2 \frac{d}{D}$. We round off k to the nearest integer k', and place the split point at a distance $d' = D2^{k'}$ from p along pq.

The result of this modified strategy is shown in Figure 26. On the left is an input with many small angles. Given this input, the basic algorithm of Section 2 would loop endlessly, adding split points closer and closer to the vertex at the center of the input. The result of executing the modified algorithm is shown on the right. Note that vertices near the central vertex arrange themselves along the concentric circles, and that the radius of the

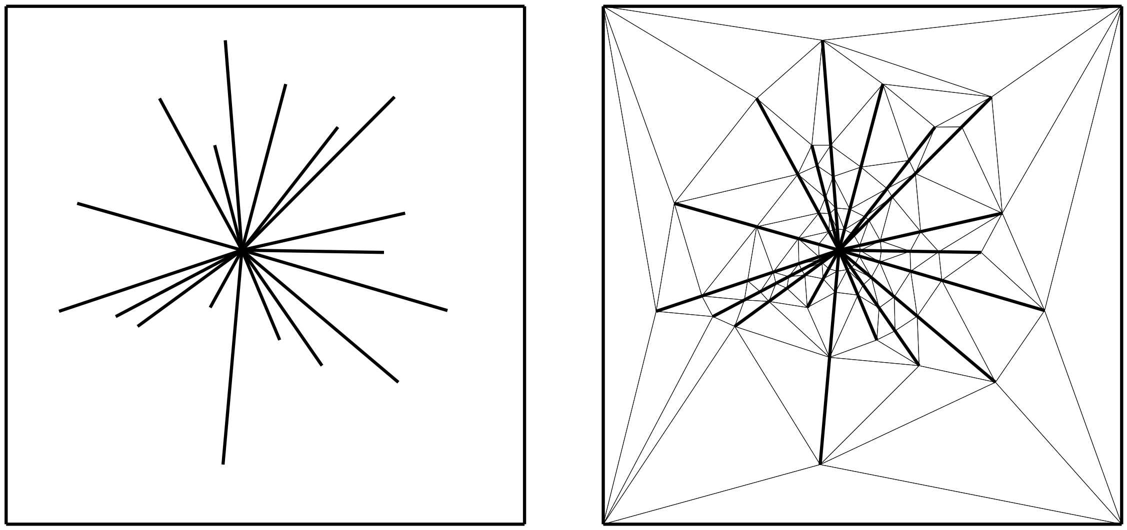


Figure 26: Input with many spokes, and triangulation using "concentric shells" instead of midpoints.

smallest circle is not much smaller than the shortest input edge. The smallest angle in such a mesh is determined by the smallest input angle, but if desired, the algorithm can achieve the usual minimum bound for all non-input angles.

We have not obtained a size-optimality proof for the algorithm under this modified splitting strategy. However, we can show that split points on concentric shells are "close" to the midpoint (between $\frac{1}{2\sqrt{2}}$ and $\frac{\sqrt{2}}{2}$ of the way along the segment). Since the size-optimality proof did not rely on the split points being precisely at the midpoints, it might be extendible to the concentric shells splitting strategy.

7 Generalization to Three Dimensions

It would be very desirable to generalize the Delaunay refinement algorithm to perform 3D tetrahedral meshing of polyhedra and polyhedral complexes. In this regard, we are somewhat pessimistic: the Delaunay refinement algorithm extends fairly readily to 3D, but its bounded aspect ratio guarantee does not. It seems that significant new ideas are necessary in order to get bounded aspect ratio tetrahedra using a Delaunay triangulation based approach.

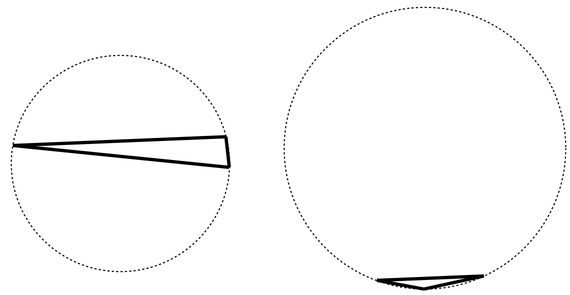


Figure 27: A "sliver" triangle in 2D must have a circumcircle much larger than its shortest edge.

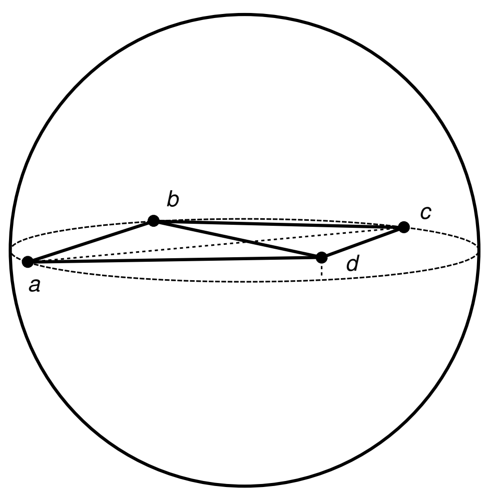


Figure 28: "Sliver" tetrahedron: 4 points spaced around the equator of a sphere, with d raised slightly.

Roughly speaking, the discrepancy between 2D and 3D is the following: in 2D, an "evenly spaced" point set (ie. no large "gaps") will have a Delaunay triangulation with no skinny triangles, but this does not hold true in 3D. Figure 27 shows why: a skinny triangle, or *sliver*, will have a circumcircle much larger than its shortest edge. Such a circumcircle forms a large "gap" not containing any points. In 3D, however, tetrahedra can have roughly equal-length edges, a reasonably-sized circumsphere, and yet be arbitrarily skinny, as shown in Figure 28: four vertices spaced equally around the equator of a sphere, with d raised slightly to a latitude of ϵ above the equator.

These flat sliver tetrahedra appear quite often in 3D Delaunay triangulations. The difficulties of avoiding them or removing them have been discussed in a number of papers, including [8], [13], [7].

8 Conclusion

We have presented a new Delaunay refinement algorithm for bounded aspect ratio triangulation of planar straightline graphs. The algorithm comes with theoretical guarantees on its behavior, yet it is simple enough to be easily implemented, and is likely to find use in practical applications.

There are several directions for further work. Foremost, can the Delaunay refinement algorithm be generalized to work for 3D triangulation of polyhedra? The Delaunay refinement algorithm is well-suited to applications involving adaptive analyses that increase mesh density in regions of large error. For adaptive or dynamic problems, mesh *reduction*, or *coarsening*, is also useful—is there a Delaunay based criterion that indicates good vertices to delete from the mesh? There are several questions regarding the size-optimality constants: Can the analysis be significantly improved? Can tight lower bounds be proved for bounded-aspect ratio triangulation, even for specific inputs?

9 Acknowledgements

I would particularly like to thank Raimund Seidel, for many productive discussions. Helpful suggestions were provided by Balas Natarajan, Marshall Bern, Eric Barszcz, and two anonymous referees. The development of the al-

gorithm was aided by the Voronoi diagram implementation of Steve Fortune (available via *netlib*).

References

[1] B. Baker, E. Grosse, and C.S. Rafferty. Nonobtuse triangulation of polygons. *Disc. and Comput. Geom.*, 3:147–168, 1988.

[2] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.Z. Du and F.K. Hwang, editors, *Computing in Euclidean Geometry*. World Scientific, 1992.

[3] M. Bern, D. Eppstein, and J.R. Gilbert. Provably good mesh generation. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 231–241. IEEE, 1990. To appear in *J. Comp. System Science*.

[4] L.P. Chew. Constrained Delaunay triangulation. *Algorithmica*, 4:97–108, 1989.

[5] L.P. Chew. Guaranteed-quality triangular meshes. Technical report, Cornell University, 1989. No. TR-89-983.

[6] L.P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 274–280. ACM, 1993.

[7] T. Dey, C. Bajaj, and K. Sugihara. On good triangulations in three dimensions. In *Proceedings of the ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, 1991.

[8] D. Field. Implementing Watson's algorithm in three dimensions. In *Proceedings of the Second Annual Symposium on Computational Geometry*, pages 246–259. ACM, 1986.

[9] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

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

[11] D.T. Lee and A. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete Comput. Geom.*, 1:201–217, 1986.

[12] E. Melissaratos and D. Souvaine. Coping with inconsistencies: A new approach to produce quality triangulations of polygonal domains with holes. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 202–211. ACM, 1992.

[13] S. Meshkat, J. Ruppert, and H. Li. Three-dimensional unstructured grid generation based on Delaunay tetrahedrization. In *Proceedings of the 3rd International Conference on Numerical Grid Generation*, pages 841–851, June 1991.

[14] S.A. Mitchell and S.A. Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, pages 212–221. ACM, 1992. Full version in Cornell Tech. Report TR 92-1267, Feb. 1992.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry – an Introduction*. Springer-Verlag, New York, 1985.

[16] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. Technical Report UCB/CSD 92/694, Computer Science Division, University of California, Berkeley, 570 Evans Hall, U.C. Berkeley, CA 94720, June 1992.

[17] J. Ruppert. *Results on Triangulation and High Quality Mesh Generation*. PhD thesis, University of California at Berkeley, 1992.

[18] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proceedings of the Fourth Annual Symposium on Discrete Algorithms*, pages 83–92. ACM-SIAM, January 1993.

[19] A. Saalfield. Delaunay edge refinements. In *Third Canadian Conference on Computational Geometry*, pages 33–36, Vancouver, 1991.

[20] R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. Technical Report 260, Inst. for Information Processing, Graz, Austria, 1988.