# Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator

Jonathan Richard Shewchuk
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
jrs@cs.cmu.edu

## 1 Introduction

Triangle is a C program for two-dimensional mesh generation and construction of Delaunay triangulations, constrained Delaunay triangulations, and Voronoï diagrams. Triangle is fast, memory-efficient, and robust; it computes Delaunay triangulations and constrained Delaunay triangulations exactly. Guaranteed-quality meshes (having no small angles) are generated using Ruppert's Delaunay refinement algorithm. Features include user-specified constraints on angles and triangle areas, user-specified holes and concavities, and the economical use of exact arithmetic to improve robustness. Triangle is freely available on the Web at "http://www.cs.cmu.edu/~quake/triangle.html" and from Netlib. This paper discusses many of the key implementation decisions, including the choice of triangulation algorithms and data structures, the steps taken to create and refine a mesh, a number of issues that arise in Ruppert's algorithm, and the use of exact arithmetic.

## 2 Triangulation Algorithms and Data Structures

A triangular mesh generator rests on the efficiency of its triangulation algorithms and data structures, so I discuss these first. I assume the reader is familiar with Delaunay triangulations, constrained Delaunay triangulations, and the incremental insertion algorithms for constructing them. Consult the survey by Bern and Eppstein [2] for an introduction.

There are many Delaunay triangulation algorithms, some of which are surveyed and evaluated by Fortune [7] and Su and Drysdale [18]. Their results indicate a rough parity in speed among the incremental insertion algorithm of Lawson [11], the divide-and-conquer algorithm of Lee and Schachter [12], and the plane-sweep algorithm of Fortune [6]; however, the implementations they study were written by different people. I believe that Triangle is the first instance in which all three algorithms have been implemented with the same data structures and floating-point tests, by one person who gave roughly equal attention to optimizing each. (Some details of how these implementations were optimized appear in Appendix A.)

Table 1 compares the algorithms, including versions that use exact arithmetic (see Section 4) to achieve robustness, and versions that use approximate arithmetic and are hence faster but may fail or produce incorrect output. (The robust and non-robust versions are otherwise identical.) As Su and Drysdale [18] also found, the divide-and-conquer algorithm is fastest, with the sweepline algorithm second. The incremental algorithm performs poorly, spending most of its time in point location. (Su and Drysdale produced a better incremental insertion implementation by using bucketing to perform point location, but it still ranks third. Triangle does not use bucketing because it is easily defeated, as discussed in the appendix.) The agreement between my results and those of Su and Drysdale lends support to their ranking of algorithms.

An important optimization to the divide-and-conquer algorithm, adapted from Dwyer [5], is to partition the vertices with alternating horizontal and vertical cuts (Lee and Schachter's algorithm uses only vertical cuts). Alternating cuts speed the algorithm and, when exact arithmetic is disabled, reduce its likelihood of failure. One million points can be triangulated correctly in a minute on a fast workstation.

All three triangulation algorithms are implemented so as to eliminate duplicate input points; if not eliminated, duplicates can cause catastrophic failures. The sweepline algorithm can easily detect duplicate points as they are removed from the event queue (by comparing each with the previous point removed from the queue), and the incremental insertion algorithm can detect a duplicate point after point location. The divide-and-conquer algorithm begins by sorting the points by their $x$-coordinates, after which duplicates can be detected and removed. This sorting step is a necessary part of the divide-and-conquer algorithm with vertical cuts, but not of the variant with alternating cuts (which must perform a sequence of median-finding operations, alternately by $x$ and

Supported in part by the Natural Sciences and Engineering Research Council of Canada under a 1967 Science and Engineering Scholarship and by the National Science Foundation under Grant ASC-9318163.

| Delaunay triangulation timings (seconds) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of points | 10,000 | | | 100,000 | | | 1,000,000 | | |
| Point distribution / Algorithm | Uniform Random | Boundary of Circle | Tilted Grid | Uniform Random | Boundary of Circle | Tilted Grid | Uniform Random | Boundary of Circle | Tilted Grid |
| Div&Conq, alternating cuts | | | | | | | | | |
| robust | 0.33 | 0.57 | 0.72 | 4.5 | 5.3 | 5.5 | 58 | 61 | 58 |
| non-robust | 0.30 | 0.27 | 0.27 | 4.0 | 4.0 | 3.5 | 53 | 56 | 44 |
| Div&Conq, vertical cuts | | | | | | | | | |
| robust | 0.47 | 1.06 | 0.96 | 6.2 | 9.0 | 7.6 | 79 | 98 | 85 |
| non-robust | 0.36 | 0.17 | failed | 5.0 | 2.1 | 4.2 | 64 | 26 | failed |
| Sweepline | | | | | | | | | |
| non-robust | 0.78 | 0.62 | 0.71 | 10.8 | 8.6 | 10.5 | 147 | 119 | 139 |
| Incremental | | | | | | | | | |
| robust | 1.15 | 3.88 | 2.79 | 24.0 | 112.7 | 101.3 | 545 | 1523 | 2138 |
| non-robust | 0.99 | 2.74 | failed | 21.3 | 94.3 | failed | 486 | 1327 | failed |

Table 1: Timings for triangulation on a DEC 3000/700 with a 225 MHz Alpha processor, not including I/O. Robust and non-robust versions of the Delaunay algorithms triangulated points chosen from one of three different distributions: uniformly distributed random points in a square, random approximately cocircular points, and a tilted 1000 $\times$ 1000 square grid.

$y$-coordinates). Hence, the timings in Table 1 for divide-and-conquer with alternating cuts could be improved slightly if one could guarantee that no duplicate input points would occur; the initial sorting step would be unnecessary.

Should one choose a data structure that uses a record to represent each edge, or one that uses a record to represent each triangle? Triangle was originally written using Guibas and Stolfi's *quad-edge* data structure [10] (without the *Flip* operator), then rewritten using a triangle-based data structure. The quad-edge data structure is popular because it is elegant, because it simultaneously represents a graph and its geometric dual (such as a Delaunay triangulation and the corresponding Voronoï diagram), and because Guibas and Stolfi give detailed pseudocode for implementing the divide-and-conquer and incremental Delaunay algorithms using quad-edges.

Despite the fundamental differences between the data structures, the quad-edge-based and triangle-based implementations of Triangle are both faithful to the Delaunay triangulation algorithms presented by Guibas and Stolfi [10] (I did not implement a quad-edge sweepline algorithm), and hence offer a fair comparison of the data structures. Perhaps the most useful observation of this paper for practitioners is that the divide-and-conquer algorithm, the incremental algorithm, and the Delaunay refinement algorithm for mesh generation were all sped by a factor of two by the triangular data structure. (However, it is worth noting that the code devoted specifically to triangulation is roughly twice as long for the triangular data structure.) A difference so pronounced demands explanation.

First, consider the different storage demands of each data structure, illustrated in Figure 1. Each quad-edge record contains four pointers to neighboring quad-edges, and two pointers to vertices (the endpoints of the edge). Each triangle record contains three pointers to neighboring triangles, and three pointers to vertices. Hence, both structures contain six pointers.[1] A triangulation contains roughly three edges for every two triangles. Hence, the triangular data structure is more space-efficient.

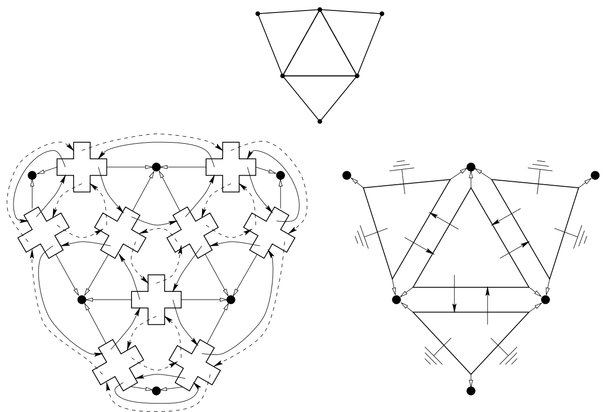

Figure 1: A triangulation (top) and its corresponding representations with quad-edge and triangular data structures. Each quad-edge and each triangle contains six pointers.

It is difficult to ascertain with certainty why the triangular data structure is superior in time as well as space, but one can make educated inferences. When a program makes structural changes to a triangulation, the amount of time used depends in part on the number of pointers that have to be read and written.

[1] Both the quad-edge and triangle data structures must store not only pointers to their neighbors, but also the *orientations* of their neighbors, to make clear how they are connected. For instance, each pointer from a triangle to a neighboring triangle has an associated orientation (a number between zero and two) that indicates which edge of the neighboring triangle is contacted. An important space optimization is to store the orientation of each quad-edge or triangle in the bottom two bits of the corresponding pointer. Thus, each record must be aligned on a four-byte boundary.

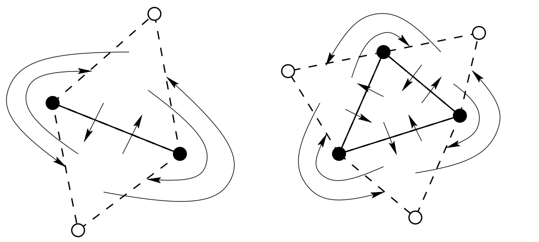

Figure 2: How the triangle-based divide-and-conquer algorithm represents an isolated edge (left) and an isolated triangle (right). Dashed lines represent ghost triangles. White vertices all represent the same "vertex at infinity"; only black vertices have coordinates.

This amount is smaller for the triangular data structure; more of the connectivity information is implicit in each triangle. Caching is improved by the fact that fewer structures are accessed. (For large triangulations, any two adjoining quad-edges or triangles are unlikely to lie in the same cache line.)

Because the triangle-based divide-and-conquer algorithm proved to be fastest, it is worth exploring in some depth. At first glance, the algorithm and data structure seem incompatible. The divide-and-conquer algorithm recursively halves the input vertices until they are partitioned into subsets of two or three vertices each. Each subset is easily triangulated (yielding an edge, two collinear edges, or a triangle), and the triangulations are merged together to form larger ones. If one uses a degenerate triangle to represent an isolated edge, the resulting code is clumsy because of the need to handle special cases. One might partition the input into subsets of three to five vertices, but this does not help if the points in a subset are collinear.

To preserve the elegance of Guibas and Stolfi's presentation of the divide-and-conquer algorithm, each triangulation is surrounded with a layer of "ghost" triangles, one triangle per convex hull edge. The ghost triangles are connected to each other in a ring about a "vertex at infinity" (really just a null pointer). A single edge is represented by two ghost triangles, as illustrated in Figure 2.

Ghost triangles are useful for efficiently traversing the convex hull edges during the merge step. Some are transformed into real triangles during this step; two triangulations are sewn together by fitting their ghost triangles together like the teeth of two gears. (Some edge flips are also needed. See Figure 3.) Each merge step creates only two new triangles; one at the bottom and one at the top of the seam. After all the merge steps are done, the ghost triangles are removed and the triangulation is passed on to the next stage of meshing.

Precisely the same data structure, ghost triangles and all, is used in the sweepline implementation to represent the growing triangulation (which often includes dangling edges). Details are omitted.

Augmentations to the data structure are necessary to support the constrained triangulations needed for mesh generation. Constrained edges are edges that may not be removed in the process of improving the quality of a mesh, and hence may not be flipped during incremental insertion of a vertex. One or more constrained edges collectively represent an input segment. Constrained edges may carry additional information, such as boundary conditions for finite element simulations. (A future version of Triangle may support curved segments this way.) The quad-edge structure supports such constraints easily; each quad-edge is simply annotated to mark the fact that it is constrained, and perhaps annotated with extra information. It is more expensive to represent constraints with the triangular structure; I augment each triangle with three extra pointers (one for each edge), which are usually null but may point to *shell edges*, which represent constrained edges and carry additional information. This eliminates the space advantage of the triangular data structure, but not its time advantage. Triangle uses the longer record only if constraints are needed.

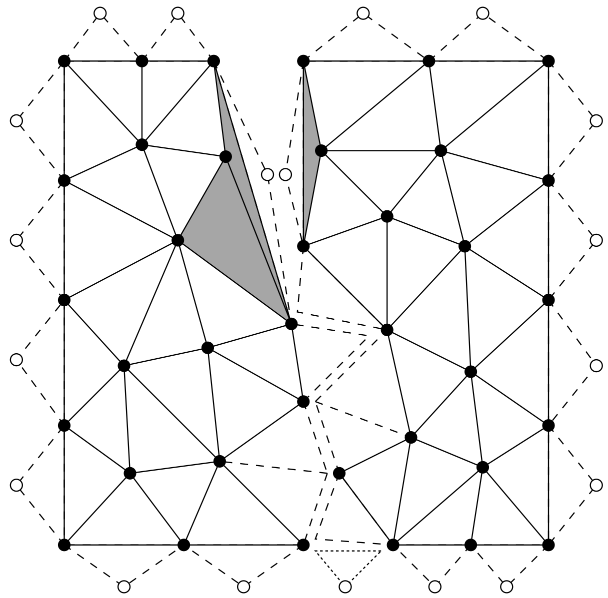

Figure 3: Halfway through a merge step of the divide-and-conquer algorithm. Dashed lines represent ghost triangles and triangles displaced by edge flips. The dotted triangle at bottom center is a newly created ghost triangle. Shaded triangles are non-Delaunay and will be displaced by edge flips.

## 3 Ruppert's Delaunay Refinement Algorithm

Ruppert's algorithm for two-dimensional quality mesh generation [15] is perhaps the first theoretically guaranteed meshing algorithm to be truly satisfactory in practice. It produces meshes with no small angles, using relatively few triangles (though the density of triangles can be increased under user control) and allowing the density of triangles to vary quickly over short distances, as illustrated in Figure 4. (Chew [3] independently developed a similar algo-

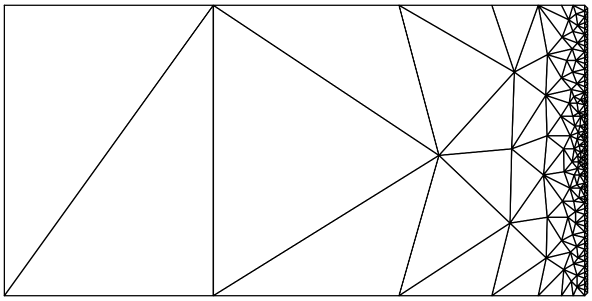

Figure 4: A demonstration of the ability of the Delaunay refinement algorithm to achieve large gradations in triangle size while constraining angles. No angles are smaller than 24°.

rithm.) This section describes Ruppert's Delaunay refinement algorithm as it is implemented in Triangle.

Triangle's input is a *planar straight line graph* (PSLG), defined to be a collection of vertices and segments (where the endpoints of every segment are included in the list of vertices). Figure 5 illustrates a PSLG defining an electric guitar. Although the definition of "PSLG" normally disallows segment intersections (except at segment endpoints), Triangle can detect segment intersections and insert vertices.

The first stage of the algorithm is to find the Delaunay triangulation of the input vertices, as in Figure 6. In general, some of the input segments are missing from the triangulation; the second stage is to insert them. Triangle can force the mesh to conform to the segments in one of two ways, selectable by the user. The first is to insert a new vertex corresponding to the midpoint of any segment that does not appear in the mesh, and use Lawson's incremental insertion algorithm to maintain the Delaunay property. The effect is to split the segment in half, and the two resulting subsegments may appear in the mesh. If not, the procedure is repeated recursively until the original segment is represented by a linear sequence of constrained edges in the mesh.

The second choice is to simply use a constrained Delaunay triangulation (Figure 7). Each segment is inserted by deleting the triangles it overlaps, and retriangulating the regions on each side of the segment. No new vertices are inserted. For reasons explained in Section 3.1, Triangle uses the constrained Delaunay triangulation by default.

The third stage of the algorithm, which diverges from Ruppert [15], is to remove triangles from concavities and holes (Figure 8). A hole is simply a user-specified point in the plane where a "triangle-eating virus" is planted and spread by depth-first search until its advance is halted by segments. (This simple mechanism saves both the user and the implementation from a common outlook wherein one must define oriented curves whose insides are clearly distinguishable from their outsides. Triangle's method makes it easier to treat holes and internal boundaries in a unified manner.[2]) Concavities

[2]I imagine computational geometers replying, "Of course," engineers responding, "Hmm," and solid modeling specialists recoiling in horror.

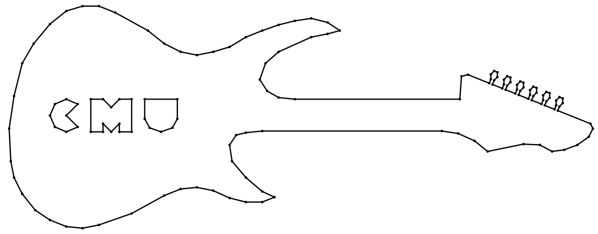



Figure 5: Electric guitar PSLG.

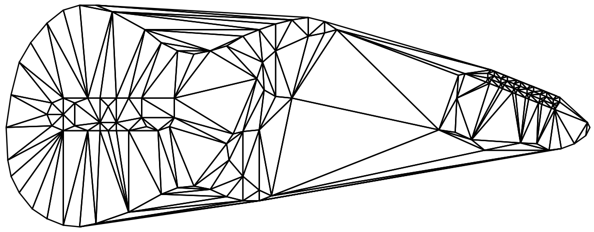

Figure 6: Delaunay triangulation of vertices of PSLG. The triangulation does not conform to all of the input segments.

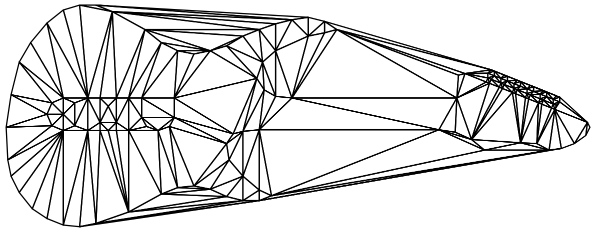

Figure 7: Constrained Delaunay triangulation of PSLG.

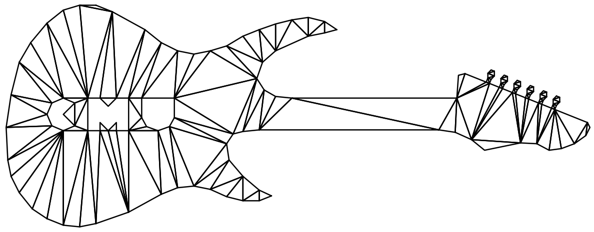

Figure 8: Triangles are removed from concavities and holes.

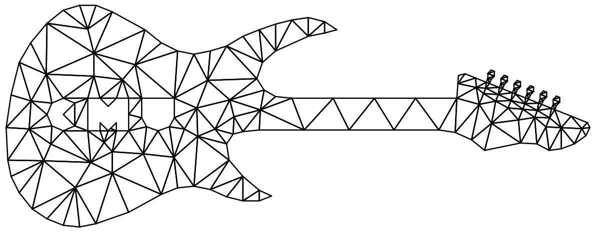

Figure 9: Conforming Delaunay triangulation with $20°$ minimum angle.

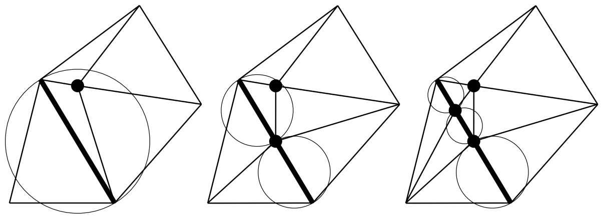

Figure 10: Segments are split recursively (while maintaining the Delaunay property) until no segments are encroached.

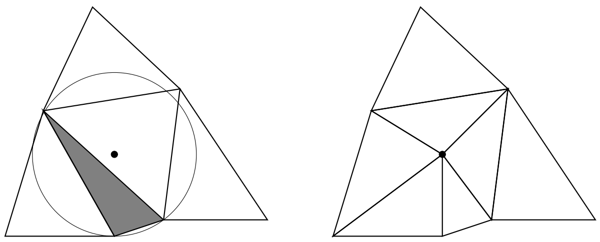

Figure 11: Each bad triangle is split by inserting a vertex at its circumcenter and maintaining the Delaunay property.

are recognized from unconstrained edges on the boundary of the mesh, and the same virus is used to hollow them out.

The fourth stage, and the heart of the algorithm, refines the mesh by inserting additional vertices into the mesh (using Lawson's algorithm to maintain the Delaunay property) until all constraints on minimum angle and maximum triangle area are met (Figure 9). Vertex insertion is governed by two rules.

- The *diametral circle* of a segment is the (unique) smallest circle that contains the segment. A segment is said to be *encroached* if a point lies within its diametral circle. Any encroached segment that arises is immediately split by inserting a vertex at its midpoint. The two resulting subsegments have smaller diametral circles, and may or may not be encroached themselves. See Figure 10.

- The *circumcircle* of a triangle is the unique circle that passes through all three vertices of the triangle. A triangle is said to be *bad* if it has an angle too small or an area too large to satisfy the user's constraints. A bad triangle is split by inserting a vertex at its *circumcenter* (the center of its circumcircle); the Delaunay property guarantees that the triangle is eliminated (see Figure 11). If the new vertex encroaches upon any segment, the vertex is deleted (reversing the insertion process) and all the segments it encroached upon are split.

Encroached segments are given priority over bad triangles. A queue of encroached segments and a queue of bad triangles are initialized at the beginning of the refinement stage and maintained throughout; every vertex insertion may add new members to either queue. The former queue rarely contains more than one segment except at the beginning of the refinement stage, when it may contain many.

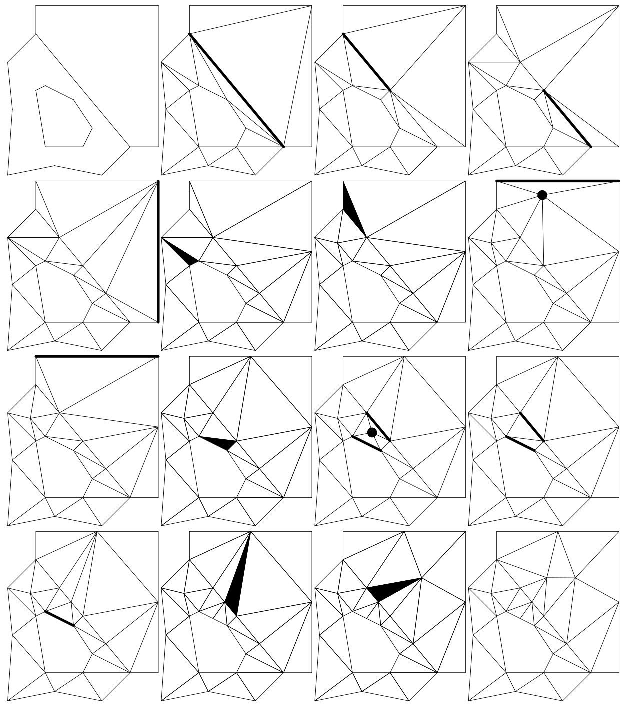

Figure 12: Demonstration of the refinement stage. The first two images are the input PSLG and its constrained Delaunay triangulation. In each image, highlighted segments or triangles are about to be split, and highlighted vertices are about to be deleted. Note that the algorithm easily accommodates internal boundaries and holes.

The refinement stage is illustrated in Figure 12. Ruppert [15] proves that this procedure halts for an angle constraint of up to 20.7°. In practice, the algorithm generally halts with an angle constraint of 33.8°, but often fails to terminate given an angle constraint of 33.9°. It would be interesting to discover why the cutoff falls there.

### 3.1 Selected Implementation Issues

Triangle removes extraneous triangles from holes and concavities before the refinement stage. This presents no problem for the refinement algorithm; the requirement that no segment be encroached and the Delaunay property together ensure that the circumcenter of every triangle lies within the mesh. (Roundoff error might perturb a circumcenter to just outside the mesh, but it is easy to identify the conflicting edge and treat it as encroached.) An advantage of removing triangles before refinement is that computation is not wasted refining triangles that will eventually be deleted.

A more important advantage is illustrated in Figure 13. If extraneous triangles remain during the refinement stage, overrefinement can occur if very small features outside the object being meshed cause the creation of small triangles inside the mesh. Ruppert suggests solving this problem by using the constrained Delaunay triangulation, and ignoring interactions that take place outside the region being triangulated. Early removal of triangles provides a nearly effortless

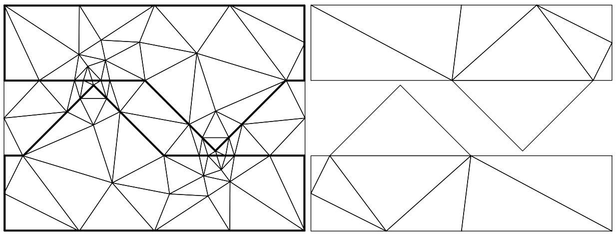

Figure 13: Two variations of the Delaunay refinement algorithm with a 20° minimum angle. Left: Mesh created using segment splitting and late removal of triangles. This illustration includes external triangles, just prior to removal, to show why overrefinement occurs. Right: Mesh created using constrained Delaunay triangulation and early removal of triangles.

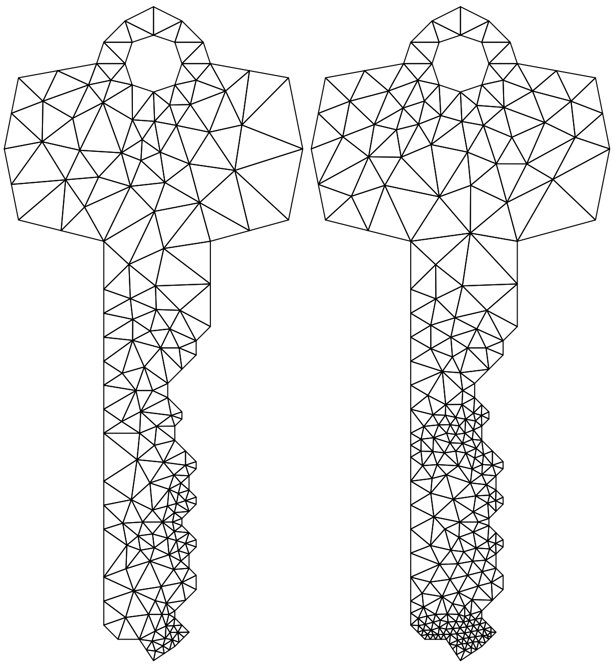

Figure 14: Two meshes with a 33° minimum angle. The left mesh, with 290 triangles, was formed by always splitting the worst existing triangle. The right mesh, with 450 triangles, was formed by using a first-come first-split queue of bad triangles.

way to accomplish this effect. Segments that would normally be considered encroached are ignored (Figure 13, right), because encroached segments are diagnosed by noticing that they occur opposite an obtuse angle in a triangle.

Another determinant of the number of triangles in the final mesh is the order in which bad triangles are split, especially when a strong angle constraint is used. Figure 14 demonstrates how sensitive the refinement algorithm is to the order. For this example with a 33° minimum angle, a heap of bad triangles indexed by their smallest angle confers a 35% reduction in mesh size over a first-in first-out queue. (This difference is typical for large meshes with a strong angle constraint, but thankfully disappears for small meshes and small

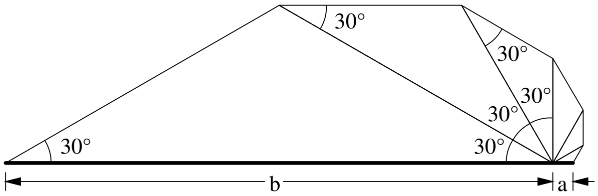


Figure 15: In any triangulation with no angles smaller than $30°$, the ratio $b/a$ cannot exceed $27$.

constraints.) The discrepancy probably occurs because circumcenters of very bad triangles are likely to split more bad triangles than circumcenters of mildly bad triangles. Unfortunately, a heap is slow for large meshes, especially when small area constraints force all of the triangles into the heap. Delaunay refinement usually takes $\mathcal{O}(n)$ time in practice, but use of a heap increases the complexity to $\mathcal{O}(n \log n)$.

Triangle's solution, chosen experimentally, is to use 64 FIFO queues, each representing a different interval of angles. It is counterproductive (in practice) to order well-shaped triangles by their worst angle, so one queue is used for well-shaped but too-large triangles whose angles are all roughly larger than 39°. Triangles with smaller angles are partitioned among the remaining queues. When a bad triangle is chosen for splitting, it is taken from the "worst" nonempty queue. This method yields meshes comparable with those generated using a heap, but is only slightly slower than using a single queue. During the refinement phase, about 21,000 new vertices are generated per second on a DEC 3000/700. These vertices are inserted using the incremental Delaunay algorithm, but are inserted much more quickly than Table 1 would suggest because a triangle's circumcenter can be located quickly by starting the search at the triangle.

### 3.2 A Negative Result on Quality Triangulations

For any angle bound $\theta > 0$, there exists a PSLG $\mathcal{P}$ such that it is not possible to triangulate $\mathcal{P}$ without creating a new corner (not present in $\mathcal{P}$) having angle smaller than $\theta$. Here, I discuss why this is true.

Ruppert's proof that his Delaunay refinement algorithm terminates makes use of the assumption that all interior angles are 90° or larger. This condition is often violated in practice, so he suggests handling small interior angles by surrounding each vertex of an acute angle with a ring of *shield edges*. As the negative result stated above suggests, there are PSLGs for which shield edges fail, and for which no construction can succeed. Fortunately, all such PSLGs I am aware of have an interior angle much smaller than $\theta$, so failure is generally predictable.

The reasoning behind the result is as follows. Suppose a segment in a conforming triangulation has been split into two subsegments of lengths $a$ and $b$, as illustrated in Figure 15. Mitchell [13] proves that if the triangulation has no angles smaller than $\theta$, then the ratio $b/a$ has an upper bound of $(2\cos\theta)^{180°/\theta}$. (This bound is tight if $180°/\theta$ is an integer;

Figure 15 offers an example where the bound is obtained.) Hence any bound on the smallest angle of a triangulation imposes a limit on the gradation of triangle sizes along a segment (or anywhere in the mesh).

A problem can arise if a small angle $\phi$ occurs at the intersection point $o$ of two segments of a PSLG, as illustrated in Figure 16 (top). The small angle cannot be improved, of course, but one does not wish to create any new small angles. Assume that one of the segments is split by a point $p$, which may be present in the input or may be inserted to help achieve the angle constraint elsewhere in the triangulation. The insertion of $p$ forces part of the region between the two segments to be triangulated (Figure 16, center), which can cause a new point $q$ to be inserted on the segment containing $p$. Let $a = |\overline{pq}|$ and $b = |\overline{op}|$ as illustrated. If the angle bound is maintained, the length $a$ cannot be large; the ratio $a/b$ is bounded below

$$\frac{\sin\phi}{\sin\theta}\left(\cos(\theta+\phi)+\frac{\sin(\theta+\phi)}{\tan\theta}\right).$$

If the region above the segments is part of the interior of the PSLG, the fan effect demonstrated in Figure 15 may necessitate the insertion of another vertex $r$ between $o$ and $p$ (Figure 16, bottom); this circumstance is unavoidable if the product of the bounds on $b/a$ and $a/b$ given above is less than one. For an angle constraint of $\theta = 30°$, this condition occurs when $\phi$ is about six tenths of a degree. Unfortunately, the new vertex $r$ creates the same conditions as the vertex $p$, but closer to $o$; the process will cascade, eternally creating smaller and smaller triangles in an attempt to satisfy the angle constraint. No algorithm can produce a finite triangulation of such a PSLG without violating the angle constraint. (It is amusing to consider whether the angle constraint can be met if one is allowed an infinite number of triangles.)

If some PSLGs do not have quality triangulations, what are the implications for shielding? Triangle implements a variant of shielding known as "modified segment splitting using concentric circular shells" (see Ruppert [15] for details), which is generally effective in practice for PSLGs that have small angles greater than 5°, and often for smaller angles. Shielding is useful even though it cannot solve all problems. On the other hand, the Delaunay refinement algorithm does not know to use careful arrangements of triangles as in Figure 15 to manage small input angles, and therefore can fail to terminate even on PSLGs for which a quality triangulation exists. Hence, Triangle prints a warning message when angles smaller than five degrees appear between input segments. The smaller an angle is, and the greater the number of small angles in a PSLG, the less likely Triangle is to terminate. An interesting question for future work is how to determine when and where it is wise to weaken the angle constraint so that termination can be ensured.

This problem presents another motivation for removing triangles from holes and concavities prior to applying the Delaunay refinement algorithm. Holes with small angles might cause the algorithm to fail if triangles are not removed

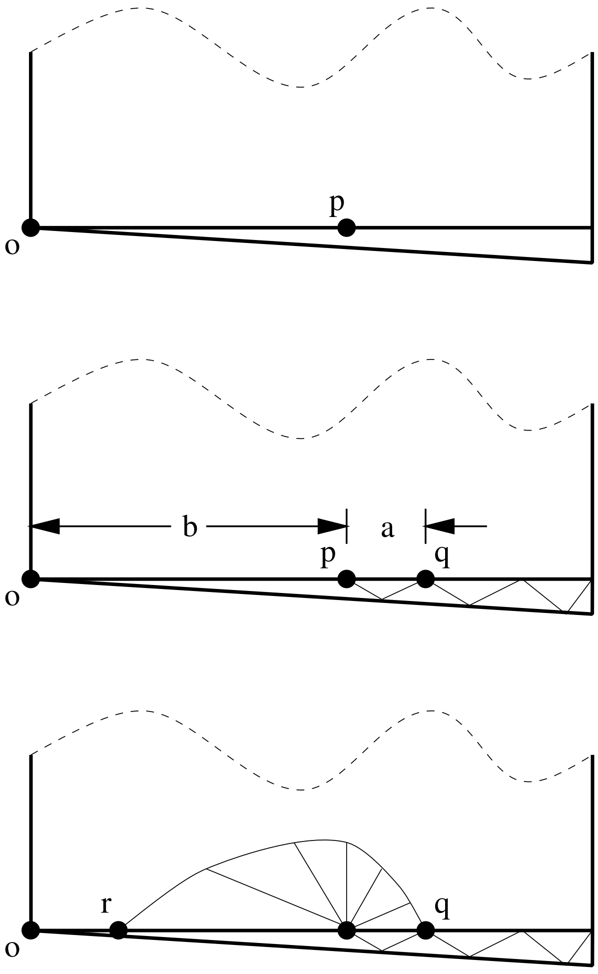


Figure 16: Top: A difficult PSLG with a small interior angle $\phi$. Center: The point $p$ and the angle constraint necessitate the insertion of the point $q$. Bottom: The point $q$ and the angle constraint necessitate the insertion of the point $r$. The process repeats eternally.

until after refinement. Concave objects can be particularly dastardly, because a very small angle may occur between a defining segment of the object and an edge of the convex hull. The user, unaware of the effect of the convex hull edge, would be mystified why the Delaunay refinement algorithm fails to terminate on what appears to be a simple PSLG. (In fact, this is how the issues described in this section first became evident to me.) Early removal of triangles from concavities avoids this problem.

## 4 Correct Adaptive Tests

The correctness of the incremental and divide-and-conquer algorithms depends on reliable *orientation* and *incircle* tests. The orientation test determines whether a point lies to the left of, to the right of, or on a line; it is used in many (perhaps most) geometric algorithms. The incircle test determines whether a point lies inside, outside, or on a circle. Inexact versions

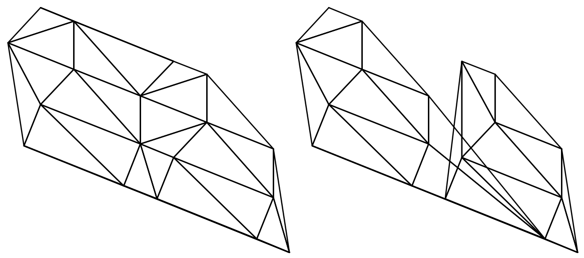

Figure 17: Left: A Delaunay triangulation (two of the guitar's tuning screws). Right: An invalid triangulation created by Triangle with exact arithmetic disabled.

of these tests are vulnerable to roundoff error, and the wrong answers they produce can cause geometric algorithms to hang, crash, or produce incorrect output. Figure 17 demonstrates a real example of the failure of Triangle's divide-and-conquer algorithm.

The easiest solution to many of these robustness problems is to use software implementations of exact arithmetic, albeit often at great expense. It is common to hear reports of implementations being slowed by factors of ten or more as a consequence. The goal of improving the speed of correct geometric calculations has received much recent attention [4, 8, 1], but the most promising proposals take integer or rational inputs, often of limited precision. These methods do not appear to be usable if it is convenient or necessary to use ordinary floating-point inputs.

Triangle includes fast correct implementations of the orientation and incircle tests that take floating-point inputs. They owe their speed to two features. First, they employ new fast algorithms for arbitrary precision arithmetic that have a strong advantage over other software techniques in computations that manipulate values of extended but small precision. Second, they are adaptive; their running time depends on the degree of uncertainty of the result, and is usually small. For instance, the adaptive orientation test is slow only if the points being tested are nearly or exactly collinear.

The orientation and incircle tests both work by computing the sign of a determinant. Fortune and Van Wyk [8] take advantage of the fact that only the sign is needed by using a *floating-point filter*: the determinant is first evaluated approximately, and only if forward error analysis indicates that the sign of the approximate result cannot be trusted does one use an exact test. Triangle's adaptive implementations carry this suggestion to its logical extreme by computing a sequence of successively more accurate approximations to the determinant, stopping only when the accuracy of the sign is assured. To reduce computation time, some of these approximations can reuse previous, less accurate computations. Shewchuk [16] presents details of the arbitrary precision arithmetic algorithms and the adaptivity scheme, and provides empirical evidence that multiple-stage adaptivity can significantly improve on two-stage adaptivity when difficult point sets are triangulated.

Using the adaptive tests, Triangle computes Delaunay triangulations, constrained Delaunay triangulations, and convex hulls exactly, roundoff error notwithstanding. Table 1 shows that the robust tests usually incur only a 10% to 30% overhead, though more time may be needed for points sets with many near-degeneracies. One exception is the divide-and-conquer algorithm with vertical cuts. Because this algorithm repeatedly merges tall, thinly separated triangulations, it performs many orientation tests on nearly-collinear points, and hence the robust version is much slower than the non-robust version. The variant that uses alternating cuts encounters nearly-collinear points less often; hence, its robust version suffers a smaller speed handicap, and its non-robust version is less likely to fail.

Of course, adaptive tests do not solve all robustness problems. Geometric computations that produce new vertices, including circumcenters and segment intersections, could be performed exactly in principle, but the results would have large bit complexity and would be inconvenient to manipulate and expensive to store. Worse, vertices of arbitrarily large bit complexity could eventually be produced in a cascading effect when the Delaunay refinement algorithm inserts circumcenters of triangles whose vertices were themselves circumcenters. Hence, it is infeasible to make the algorithm perfectly robust. Fortunately, the Delaunay refinement algorithm is naturally stable with regard to floating-point roundoff error. Problems arise only when triangles are refined to so small a size that it is no longer possible to construct a circumcenter that is distinct from its triangle's vertices.

I have not produced a robust version of the sweepline algorithm for a somewhat technical reason. The sweepline algorithm maintains a priority queue (normally implemented as a heap) containing two types of events: *site events*, where the sweepline passes over an input point, and *circle events*, where the sweepline reaches the top of a circle defined by three consecutive vertices on the boundary of the triangulation. Unfortunately, the $y$-coordinate of such a circle top is expensive to compute exactly, may be irrational, and has a somewhat complicated exact representation. A robust implementation must keep the events correctly ordered, and hence must replace the simple comparisons normally used to maintain a priority queue with a test that correctly compares two circle tops. Even a fast adaptive version of such a test would be so much slower than simple comparisons that event queue maintenance, which is a dominant cost of the sweepline algorithm, would become prohibitively expensive.

# A Additional Implementation Notes

The sweepline and incremental Delaunay triangulation implementations compared by Su and Drysdale [18] each use some variant of uniform bucketing to locate points. Bucketing yields fast implementations on uniform point sets, but is easily defeated; a small, dense cluster of points in a large, sparsely populated region may all fall into a single bucket. I have not used bucketing in Triangle, preferring algorithms that exhibit good performance with any distribution of input points. As a result, Triangle may be slower than necessary when triangulating uniformly distributed point sets, but will not exhibit asymptotically slower running times on difficult inputs.

Fortune's sweepline algorithm uses two nontrivial data structures in addition to the triangulation: a priority queue to store events, and a balanced tree data structure to store the sequence of edges on the boundary of the mesh. Fortune's own implementation, available from Netlib, uses bucketing to perform both these functions; hence, an $\mathcal{O}(n \log n)$ running time is not guaranteed, and Su and Drysdale [18] found that the original implementation exhibits $\mathcal{O}(n^{3/2})$ performance on uniform random point sets. By modifying Fortune's code to use a heap to store events, they obtained $\mathcal{O}(n \log n)$ running time and better performance on large point sets (having more than 50,000 points). However, bucketing outperforms a heap on small point sets.

Triangle's implementation uses a heap as well, and also uses a splay tree [17] to store mesh boundary edges, so that an $\mathcal{O}(n \log n)$ running time is attained, regardless of the distribution of points. Not all boundary edges are stored in the splay tree; when a new edge is created, it is inserted into the tree with probability $0.1$. (The value $0.1$ was chosen empirically to minimize the triangulation time for uniform random point sets.) At any time, the splay tree contains a random sample of roughly one tenth of the boundary edges. When the sweepline sweeps past an input point, the point must be located relative to the boundary edges; this point location involves a search in the splay tree, followed by a search on the boundary of the triangulation itself.

Splay trees adjust themselves so that frequently accessed items are near the top of the tree. Hence, a point set organized so that many new vertices appear at roughly the same location on the boundary of the mesh is likely to be triangulated quickly. This effect partly explains why Triangle's sweepline implementation triangulates points on the boundary of a circle more quickly than the other point sets, even though there are many more boundary edges in the cocircular point set and the splay tree grows to be much larger (containing $\mathcal{O}(n)$ boundary edges instead of $\mathcal{O}(\sqrt{n})$).

Triangle's incremental insertion algorithm for Delaunay triangulation uses the point location method proposed by Mücke, Saias, and Zhu [14]. Their *jump-and-walk* method chooses a random sample of $\mathcal{O}(n^{1/3})$ vertices from the mesh (where $n$ is the number of nodes *currently* in the mesh), determines which of these vertices is closest to the query point, and walks through the mesh from the chosen vertex toward the query point until the triangle containing that point is found. Mücke et al. show that the resulting incremental algorithm takes expected $\mathcal{O}(n^{4/3})$ time on uniform random point sets. Table 1 appears to confirm this analysis. Triangle uses a sample size of $0.45n^{1/3}$; the coefficient was chosen empirically to minimize the triangulation time for uniform random point sets. Triangle also checks the previously inserted point, because in many practical point sets, any two consecutive points have a high likelihood of being near each other.

A more elaborate point location scheme such as that suggested by Guibas, Knuth, and Sharir [9] could be used (along with randomization of the insertion order) to obtain an expected $\mathcal{O}(n \log n)$ triangulation algorithm, but the data structure used for location is likely to take up as much memory as the triangulation itself, and unlikely to surpass the performance of the divide-and-conquer algorithm; hence, I do not intend to pursue it.

Note that all discussion in this paper applies to Triangle version 1.2; earlier versions lack the sweepline algorithm and many optimizations to the other algorithms.

## References


[1] Francis Avnaim, Jean-Daniel Boissonnat, Olivier Devillers, Franco P. Preparata, and Mariette Yvinec. *Evaluating Signs of Determinants Using Single-Precision Arithmetic*. 1995.

[2] Marshall Bern and David Eppstein. *Mesh Generation and Optimal Triangulation*. Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 23–90. World Scientific, Singapore, 1992.

[3] L. Paul Chew. *Guaranteed-Quality Mesh Generation for Curved Surfaces*. Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 274–280. Association for Computing Machinery, May 1993.

[4] Kenneth L. Clarkson. *Safe and Effective Determinant Evaluation*. 33rd Annual Symposium on Foundations of Computer Science, pages 387–395. IEEE Computer Society Press, October 1992.

[5] Rex A. Dwyer. *A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations*. Algorithmica **2**(2):137–151, 1987.

[6] Steven Fortune. *A Sweepline Algorithm for Voronoï Diagrams*. Algorithmica **2**(2):153–174, 1987.

[7] ———. *Voronoï Diagrams and Delaunay Triangulations*. Computing in Euclidean Geometry (Ding-Zhu Du and Frank Hwang, editors), Lecture Notes Series on Computing, volume 1, pages 193–233. World Scientific, Singapore, 1992.

[8] Steven Fortune and Christopher J. Van Wyk. *Efficient Exact Arithmetic for Computational Geometry*. Proceedings of the Ninth Annual Symposium on Computational Geometry, pages 163–172. Association for Computing Machinery, May 1993.

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

[10] Leonidas J. Guibas and Jorge Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoï Diagrams*. ACM Transactions on Graphics **4**(2):74–123, April 1985.

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

[12] D. T. Lee and B. J. Schachter. *Two Algorithms for Constructing a Delaunay Triangulation*. International Journal of Computer and Information Sciences **9**(3):219–242, 1980.

[13] Scott A. Mitchell. *Cardinality Bounds for Triangulations with Bounded Minimum Angle*. Sixth Canadian Conference on Computational Geometry, 1994.

[14] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. *Fast Randomized Point Location Without Preprocessing in Two- and Three-dimensional Delaunay Triangulations*. Proceedings of the Twelfth Annual Symposium on Computational Geometry. Association for Computing Machinery, May 1996.

[15] Jim Ruppert. *A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation*. Journal of Algorithms **18**(3):548–585, May 1995.

[16] Jonathan Richard Shewchuk. *Robust Adaptive Floating-Point Geometric Predicates*. Proceedings of the Twelfth Annual Symposium on Computational Geometry. Association for Computing Machinery, May 1996.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. *Self-Adjusting Binary Search Trees*. Journal of the Association for Computing Machinery **32**(3):652–686, July 1985.

[18] Peter Su and Robert L. Scot Drysdale. *A Comparison of Sequential Delaunay Triangulation Algorithms*. Proceedings of the Eleventh Annual Symposium on Computational Geometry, pages 61–70. Association for Computing Machinery, June 1995.