Notes by Franklin Cho

- Divide-and-conquer Delaunay triangulation and its parallel implementation
- Meshing (triangulation) given a boundary
  - Applications to solid modeling and computer graphics

1 Divide-and-Conquer Delaunay Triangulation

In this section, we will examine the divide-and-conquer approach to Delaunay triangulation by Blelloch, Miller and Talmor. This algorithm performs most of the work on split rather than merge, which allows for easy parallelization. For each recursive call, we keep

- B = Delaunay boundary of region
- P = Interior points

Initially, we set $B = \text{Convex\_Hull}(P_{input})$ and $P = P_{input} - B$. We will assume 2-D data set for now. We also assume that the input points are pre-sorted in x and y. Here is the pseudo-code for the Delaunay triangulation algorithm:

Function $DT(P, B)$

if $|P| = 0$ then boundary_solve(B)
else
  x_m = median point along x-axis
  L = split boundary (set of Delaunay edges along $x = x_m$)
  l = points in L
  $P_{left} = \{p \in P : p_x < x_m, p \notin l\}$
  $P_{right} = \{p \in P : p_x > x_m, p \notin l\}$
  $T_{left} = DT(P_{left}, \text{intersect}(B, L))$
  $T_{right} = DT(P_{right}, \text{intersect}(B, \text{reverse}(L))$
  Return T_{left} and T_{right}

In the first line of the function, boundary_solve is called if the region contains no interior points. There are known linear-time algorithm to triangulate a region with no interior points. The difficult part of the algorithm is finding the Delaunay edges along the line $x = x_m$ (Figure 73). As shown in the previous lecture, we can obtain the Delaunay triangulation of a set of points P by projecting the points onto a 3-dimensional parabola $P' = \{(x, y, x^2+y^2) : (x, y) \in P\}$ and then projecting LowerConvexHull(P') back onto the x-y plane. Based on this idea, we find the Delaunay edges along $x = x_m$ by first projecting the points onto the parabola $P' = \{(x - x_m, y, (x - x_m)^2 + y^2) : (x, y) \in P\}$, which is centered on the line $x = x_m$. Notice that the edges of the lower convex hull of P' correspond to the Delaunay edges on the x-y plane (Figure 74). We then form the set P'' by projecting the points in P' onto the plane $x = x_m$, i.e., $P'' = \{(y, (x - x_m)^2 + y^2) : (x, y) \in P\}$ (note we have just dropped the first coordinate from P'. The lower convex hull of P'' (in 2-D) corresponds to the Delaunay edges along $x = x_m$ (Figure 75).

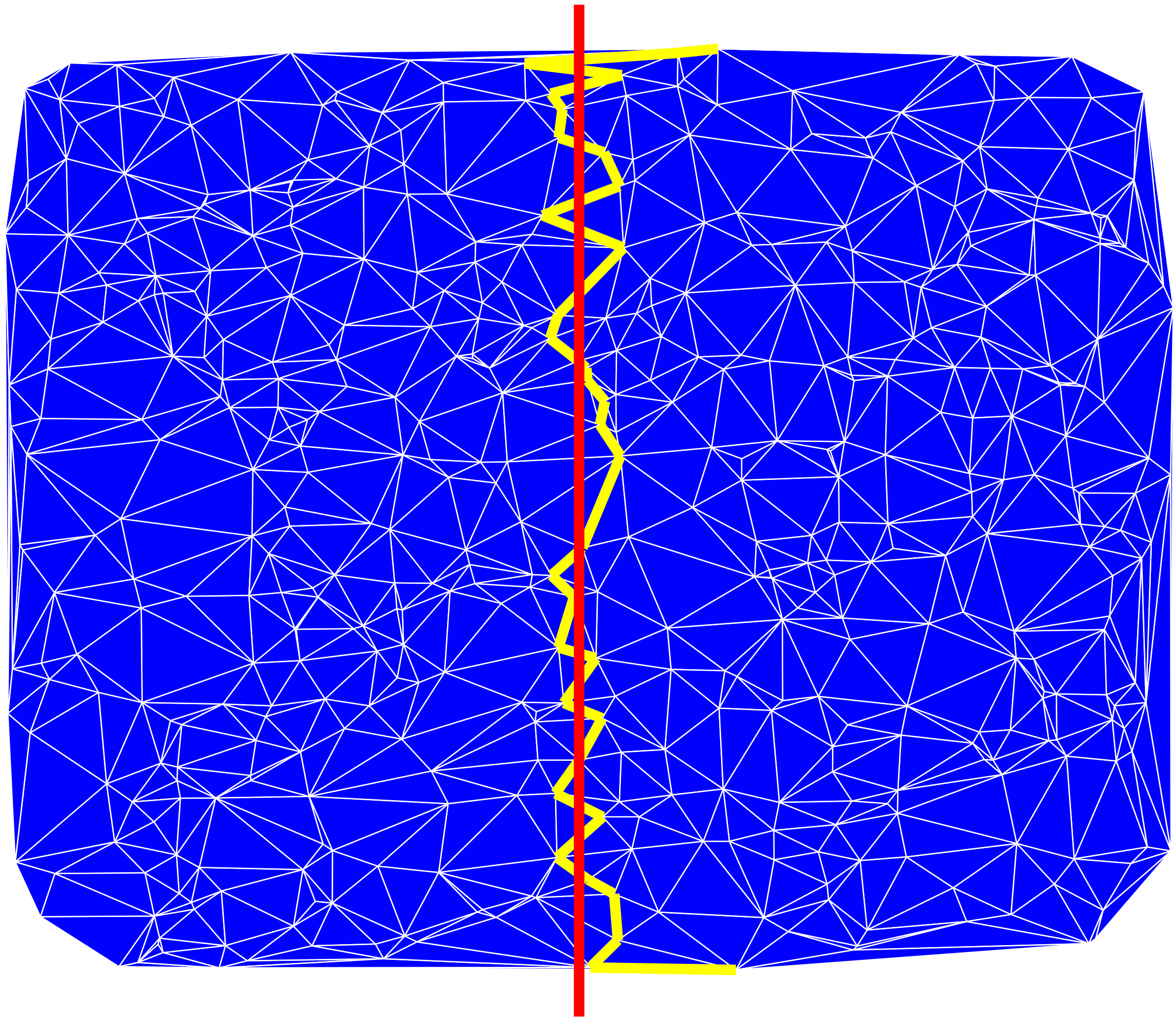


Figure 73: Set of Delaunay edges along $x = x_m$

Let us analyze the asymptotic behavior of the above algorithm. Let n be the total number of interior points ($n = |P|$), let m be the number of points on the border B, and let N be the total number of points ($N = n + m$).

The median x_m can be found in $O(1)$ since we assume that the input points are presorted. The translation and projection trivially takes $O(n)$. The convex hull of C can be found in $O(n)$ since the points are sorted. Since we are dividing the points by their median, we are guaranteeing that each sub-problem has no more than half as many interior points

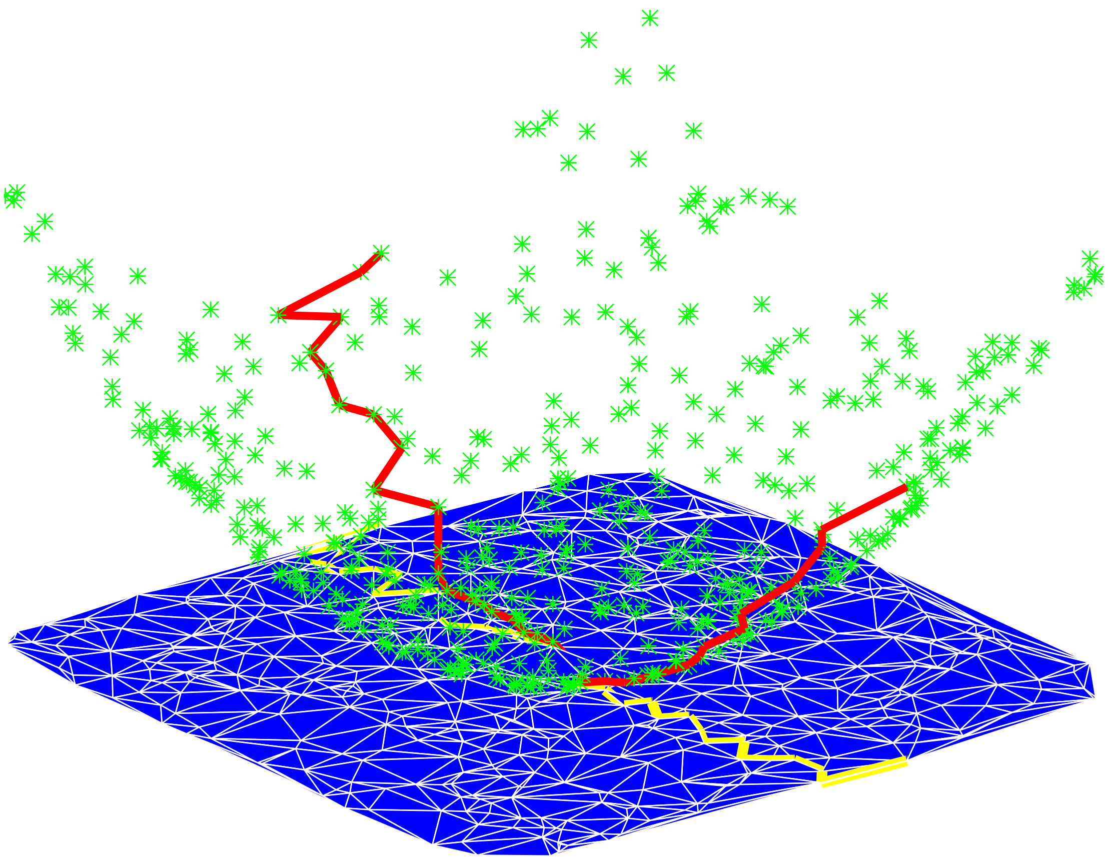


Figure 74: Edges of the Delaunay triangulation on the x-y plane correspond to the edges of the lower convex hull of the points projected onto a paraboloid (P')

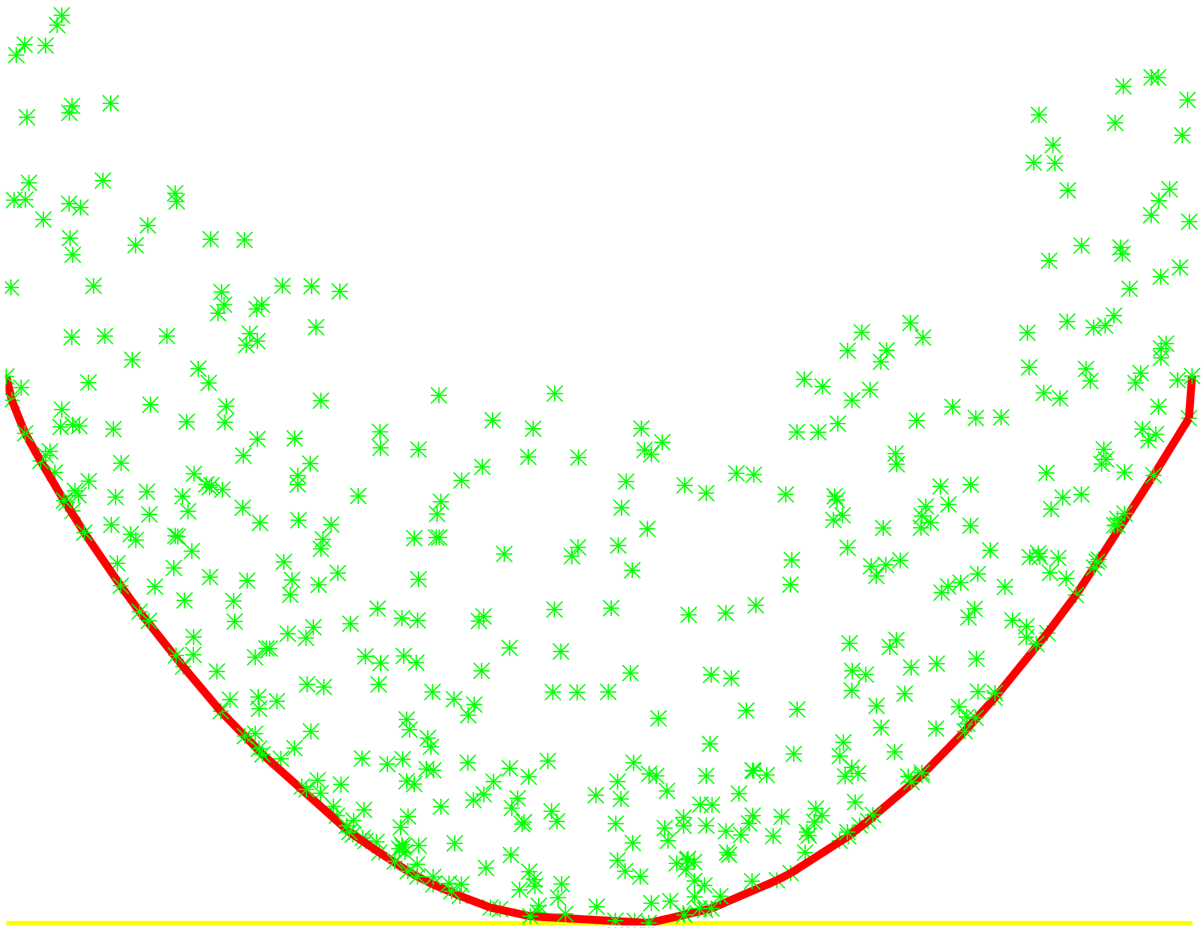


Figure 75: The points in P' projected onto the $x = x_m$ plane (C). The Delaunay edges along $x = x_m$ correspond to the edges on the 2-D lower convex hull of C

as the original problem. Therefore, the running time of the operations on the interior points can be describe by the following recurrence relation:

$$T(n) < 2T(\frac{n}{2}) + O(n)$$

By solving this recurrence relation, we obtain the total running time of $O(n \log n)$ for the operations on interior points.

Now, let us analyze the border-merge operation. For each level of recursion, the border-merge operations take $O(m)$ time. Note that we cannot guarantee that the size of the border in each sub-problem is half the size of the original border. However, we can still place an upper bound on the total amount of time spent on border-merge. Given that there are N total points, the number of edges in the triangulation of the points is bounded by $3N$. Since each boundary edges is counted twice, the absolute upper bound on the number of edges in each level of recursion is $6N$. Therefore, an upper bound on m on each level of recursion is $6N$. Since there are at most $\log N$ levels of recursion, the total time spent merging borders is bounded by $6N \log N = O(N \log N)$.

Counting both operations on points and borders, the total running time of the algorithm is $O(N \log N)$ (since $n < N$). Note that each operation on points and borders can be parallelized easily.

In practice, the following test results were obtained:

- When this algorithm was tested on both uniform and non-uniform inputs, it was found that input sets with uniform and normal distribution bucket well (i.e., it divides nicely when the input set is divided along the median). On the other hand, Kuzmin (zoomed) and line distributions don't bucket well.
- Overall, the running time with non-uniform distributed input set is at most 1.3 times slower than the running time with uniform distributed input. (Previously, the non-uniform inputs took at least 5 times longer).
- This algorithm scales well with problem size, and can solve problems too big for serial machine. The parallel algorithm is only about 2 times slower than a sequential code. (Previously, the parallel algorithm ran 4 times slower than a sequential code).
- This algorithm is a practical parallel 2D Delaunay triangulation algorithm, and it is the first to be competitive with serial code.

Does this algorithm generalize to 3D? Yes in theory, but the algorithm requires 3D convex hull generation as a subroutine, and it is not clear how to parallelize this operation. Also, the border merge is more complicated with 3D borders. Generalizing this algorithm to 3D would be an interesting future work.

2 Incremental Delaunay Triangulation Algorithm

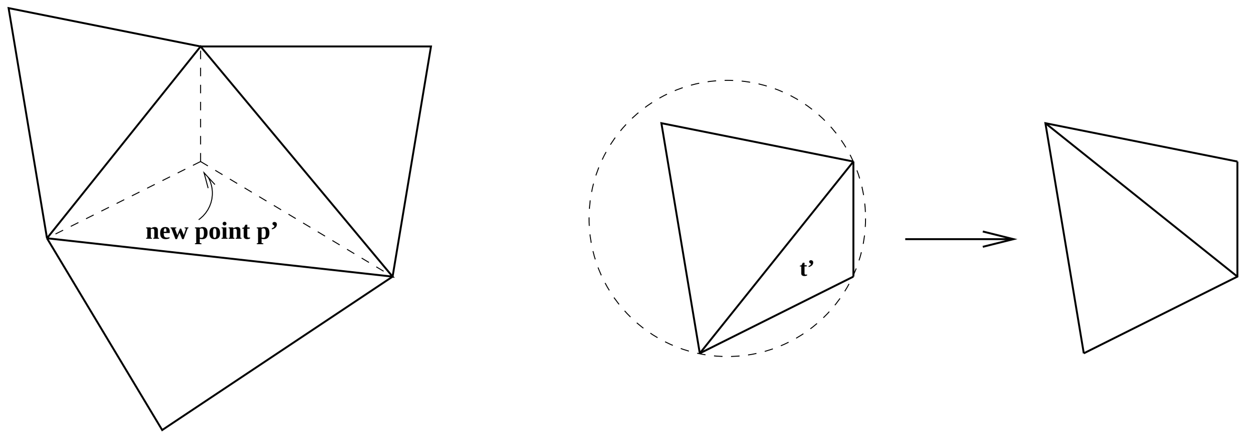
In this section, we will discuss the incremental approach to constructing the Delaunay triangulation. The incremental approach means that the points are added one at a time.

Whenever a point is added, we update the Delaunay triangulation so that at each stage, a complete Delaunay triangulation is generated with the points added so far.

In this algorithm, we assume that the points in P are already Delaunay triangulated. To add a new point p':

1. Locate the triangle t in $DT(P)$ which contains p'
2. Add p' and add the edges from p' to the three corners of t
3. Fix up neighboring regions by "edge flipping"

Step 3 requires more explanation. For each of the newly formed triangle T_{new}, we perform the in-circle-test to see if T_{new} is well-formed: we form the circumcircle of T_{new}, and test if it contains any of the other points. If it does, then T_{new} is not well-formed, and we "flip" the edge (Figure 76). This is repeated for newly created triangles and is discussed in more detail in Lecture 17.



(a) new point p' is added to triangle t, and new edges are formed to the three corners of t

(b) Perform in-circle-test for each sub-triangle, and fix up the neighboring region by "edge flipping"

Figure 76: Step 3 of the Incremental Delaunay triangulation. If the in-circle-test fails, then the edge is "flipped".

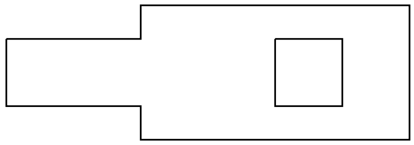
3 Mesh Generation

In this section, we will discuss mesh generation, also called unstructured grid generation, or just triangulation. Given a boundary, we want to triangulate the interior by possibly adding points. We want the final triangulation to have the following properties:

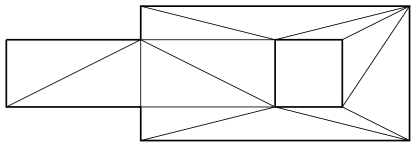
1. Bounded aspect ratio for triangles, i.e. triangles that are not skinny (don't have any small angles).

2. As few triangles as possible. This criterion is important for finite element analysis, for example, because the running time is a function of the number of triangles.

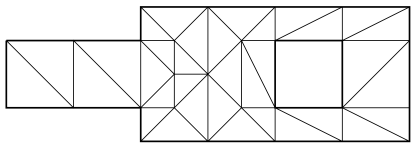
Although, as discussed previously, Delaunay triangulation guarantees the triangulation of a set of points that has the greatest minimum angle (at least in 2-D), as Figure 77 illustrates it is often possible to create triangles with larger minimum angles by adding extra points.



(a) Input Widget



(b) Triangulation without adding points



(c) Triangulation with 6 interior and 10 boundary points added

Figure 77: Extra points can be added to reduce the aspect ratio of the triangles

3.1 Ruppert's Algorithm

Given a set of 2D segments and an angle $\alpha \leq 20°$, Ruppert's algorithm triangulates the input set such that for each triangulated region:

1. no angle is less than α

2. number of triangles is within a constant factor of optimal

Although $\alpha \leq 20°$ is the limit that has been proved, $\alpha \leq 30°$ works well in practice. Ruppert's algorithm works very well in practice, and is used in the "triangle" application.

The basic idea behind Ruppert's algorithm is:

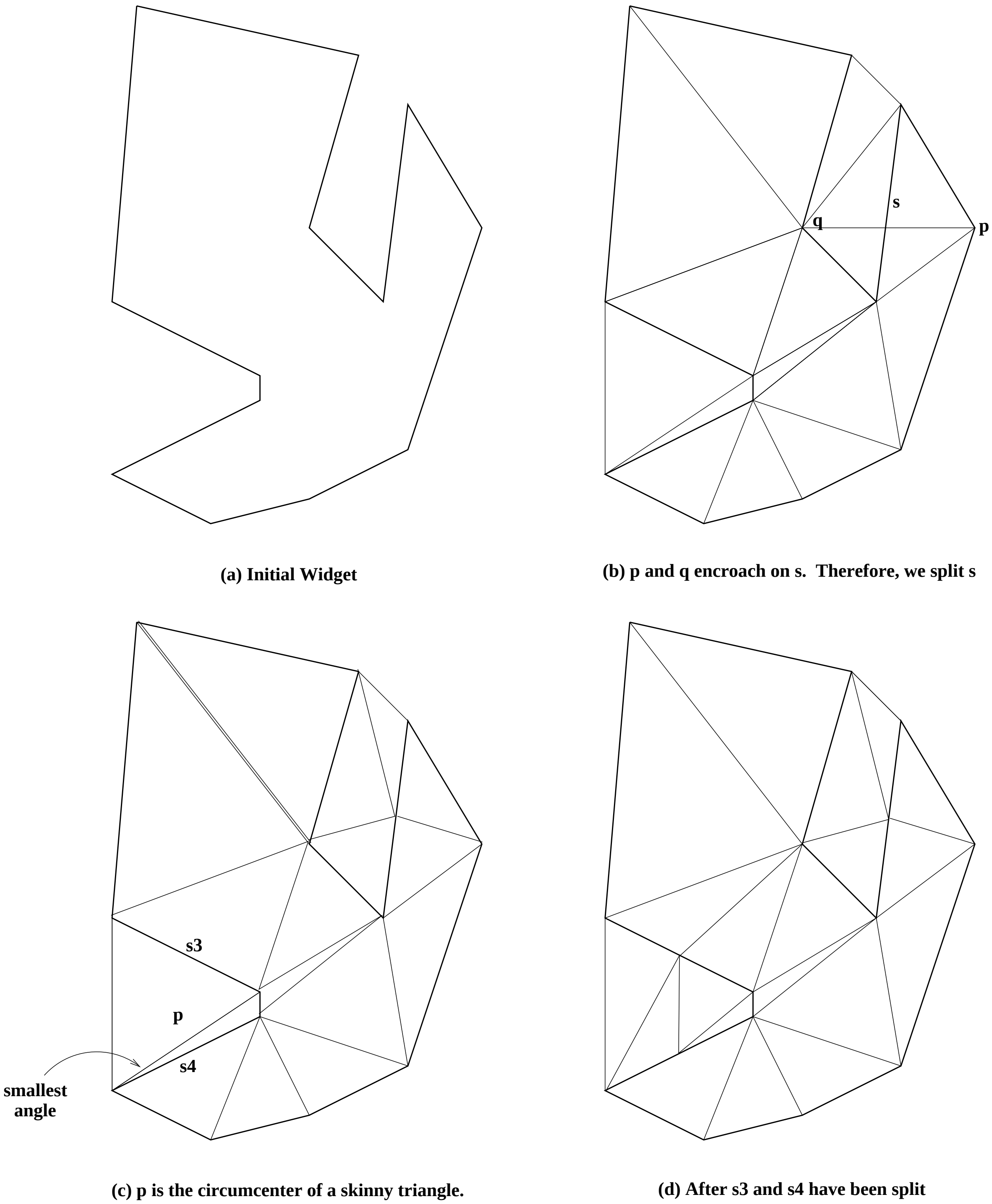
1. Start with Delaunay triangulation of endpoints of segments. As shown previously, Delaunay triangulation would produce the maximum minimum angle.
2. Get rid of skinny triangles by
   - splitting segments in half
   - adding a new point in the center of existing triangle

To describe the algorithm, we will use the following terminology. A *segment* denotes either the input segment, or part of an input segment. An *edge* denotes a Delaunay edge. A *vertex* denotes endpoint of a segment or an edge (the Delaunay vertices). A point *encroaches* on a segment if it is inside the segment's *diametrical circle*. A diametrical circle of a segment is a circle with the segment as its diameter.

Here is the pseudo-code for Ruppert's Algorithm:

S = input segments
V = endpoints of S
$T = DT(V)$ // Delaunay Triangulation
Repeat
    while any $s \in S$ is encroached upon, split s in half, update T
    let t be a skinny triangle in T (i.e., t contains an angle smaller than α)
    p = circumcenter(t)
    if p encroaches on segment $s_1, s_2, .., s_k$,
        split $s_1, s_2, .., s_k$ in half, update T
    else
        $V = V \cup p$, update $T = DT(V)$
Until no segment is encroached upon and no angle is $< \alpha$
Output T

Figure 78 illustrates Ruppert's algorithm:



(a) Initial Widget

(b) p and q encroach on s. Therefore, we split s

(c) p is the circumcenter of a skinny triangle. p encroaches on s3 and s4, and therefore we must split s3 and s4

(d) After s3 and s4 have been split

Figure 78: Successive steps of Ruppert's algorithm

Let us analyze the running time of this algorithm. If output has n triangles, then the algorithm goes through at most n iterations, and each iteration makes at most n updates. Therefore, the worst-case running time is $O(n^2)$. The algorithm runs in $O(n \log n)$ time in practice. It is currently not known whether Ruppert's algorithm can be parallelized.

The proof of "within constant factor of optimal" clause is based on the concept of "local feature size". Given a point x, the local feature size $\mathrm{lfs}(x, y)$ is defined to be the minimum radius of the circle centered at (x, y) that contains two non-incident segments. $\mathrm{lfs}(x)$ is a continuous function. We will state without proof that:

$$V < C_1 \int_A \frac{1}{(\mathrm{lfs}(x,y))^2} dx dy$$

where V is the number of vertices returned by Ruppert's triangulation, C_1 is a constant and A ranges over the area of the widget. Furthermore it is possible to prove that any triangulation requires at least

$$C_2 \int_A \frac{1}{(\mathrm{lfs}(x,y))^2} dx dy$$

vertices ($C_2 < C_1$). These together show that the number of vertices generated by Ruppert's algorithm is within a constant factor of optimal.