Notes by Aaron Brown

Outline

- Introduction to Triangulation
- Basic Geometric Operations
  1. Degeneracies and Precision Issues
  2. Line Side Test
  3. In-Circle Test
- Convex Hulls
  1. Dual Problem: Convex Polytopes
  2. Algorithms
     - Giftwrapping
     - Graham Scan
     - Mergehull
     - Quickhull
- Voronoi Diagrams
- Delaunay Triangulations (*dual of Voronoi diagrams*)
  1. Properties
  2. Algorithms
     - Naive Algorithm
     - Solution via Reduction to 3D Convex Hull
     - Direct Algorithms

The following three lectures cover triangulation, which is one of the many problems in the larger field of "computational geometry". Problems in computational geometry have found many applications in industry. We will also cover convex-hulls and Voronoi-diagrams because they are closely related to triangulation.

1 Introduction to Triangulation

Triangulation is a process that takes a region of space and divides it into subregions. The space may be of any dimension (although we will focus primarily on the two-dimensional problem here), and the subregions typically are shaped as triangles (2D), tetrahedrons (3D), etc. The goal of triangulation is to produce a subdivision that obeys certain "nice" properties, for example one in which the output triangles are as close to equilateral as possible.

Triangulation is used in many real-world applications. One such application is *finite element simulation*, in which a partial differential equation (PDE) is simulated over a continuous surface or space by first using a triangulation to break the space into small, discrete elements, then simulating the PDE discretely over the elements. Finite element simulations are common in fluid flow applications and similar problems; an example that illustrates the role of the triangulation can be seen at:

`http://www.cs.cmu.edu/~scandal/applets/airflow.html`

Other important applications of triangulation include:

- *Surface approximation*: a continuous surface is approximated by a mesh of triangles. This is used in graphics applications (e.g., face representation), compression of surface representations, interpolation of surfaces, and construction applications (*e.g.*, optimizing dirt movement to contour a region for new construction).

- *Nearest neighbor structure identification*: the triangulation includes data identifying the nearest neighbors of each point in the triangulated data set. This is potentially useful for clustering applications, etc.

In general, triangulation operates on several different types of input. The simplest case (and the one that we focus on in these notes) is when the input is just a set of points which define the vertices of the output triangulation. Another possibility is that the input data is a boundary defining a region of space. In this case, the triangulation must generate the interior points needed to complete the triangulation. Finally, the input can be specified as a density function $\varphi(x, y, z)$, which specifies the density of triangles across the region to be triangulated. This last format is especially useful in simulation, as it allows the person developing the model to focus triangle detail (and thus simulation accuracy) on critical portions of the model.

2 Basic Geometrical Operations

2.1 Degeneracies and Numerical Precision

Before getting into the triangulation algorithms themselves, we first need to consider two important issues that arise when these algorithms are implemented in the real world: handling of Degeneracies and numerical precision.

Degeneracies occur (in two dimensions) when three or more points are co-linear, or when four or more points lie on the same circle. Special care is often needed in triangulation algorithms to make them robust with respect to degeneracies.

Another related problem that crops up in the real world is the issue of numerical precision. Since (in general) computers operate with only a fixed amount of floating-point precision errors are introduced by most arithmetic operations. These errors can lead to inconsistent answers, especially when points are almost degenerate. For example if three points a, b and c are almost on a line the answer to whether a is above $b - c$ could differ depending on how you calculate it. There are three potential solutions to the numerical precision problem (and the related degeneracy problem):

1. Assume that the problem doesn't exist. Many existing implementations do this, but it is a bad idea.

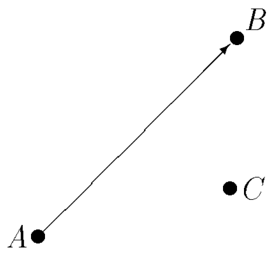
2. Design the algorithm to be robust with respect to degeneracies and to lack of precision (e.g., by using exact or adaptive arithmetic). This is the best solution.

3. Randomly perturb the input data to avoid degeneracies. This may work in most cases, but the possibility of degeneracies or precision problems still remains.

Luckily, most triangulation algorithms only use two basic operations which are sensitive to degeneracies and numerical precision: the *line side test* and the *in-circle test*. These operations encapsulate all of the needed coordinate comparisons, and thus they are the only routines that need be implemented in infinite-precision or adaptive-precision arithmetic. The rest of the triangulation algorithms are primarily combinatorics and graph manipulations.

2.2 The Line-side Test

Given a point C and a line AB, the *line-side test* answers the question: "is C above or below AB?"



The line-side test is easily implemented via a matrix determinant operation; if the determinant operation is built with adaptive-precision arithmetic, the line-side test will be stable with respect to numerical precision. Let the line be defined by points $A = (x_a, y_a)$ and $B = (x_b, y_b)$, and let the test point be $C = (x_c, y_c)$. Then let:

$$D = \begin{vmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{vmatrix}$$

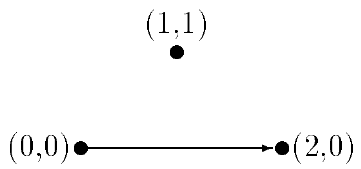


Figure 59: Example of applying the line-side test.

Then the point C is "above" the line $A-B$ iff $D > 0$. Notice as well that the area of triangle ABC is given by $D/2$.

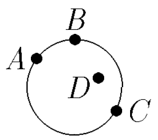
As an example, consider the line AB and the point C with $A = (0,0)$, $B = (2,0)$, and $C = (1,1)$, as illustrated in Figure 59. In this case,

$$D = \begin{vmatrix} 0 & 0 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 1 \end{vmatrix} = 2 > 0.$$

Thus, C is above AB, and the triangle area is $\frac{2}{2} = 1$.

2.3 The In-circle Test

Given three points A, B, C, and D, the *in-circle test* answers the question: "is D in the circle defined by A, B, and C?"



As in the line-side test, the in-circle test can be implemented with a matrix determinant. Again, if the determinant operation uses adaptive-precision arithmetic, the in-circle test should be stable. Notice, however, that A, B, and C only define a unique circle if they are not colinear.

As before, let $A = (x_a, y_a)$, $B = (x_b, y_b)$, $C = (x_c, y_c)$, and $D = (x_d, y_d)$. Then the particular matrix determinant used for the in-circle test is:

$$D = \begin{vmatrix} x_a & y_a & x_a^2 + y_a^2 & 1 \\ x_b & y_b & x_b^2 + y_b^2 & 1 \\ x_c & y_c & x_c^2 + y_c^2 & 1 \\ x_d & y_d & x_d^2 + y_d^2 & 1 \end{vmatrix}$$

Then the point D is in the circle defined by ABC iff $D > 0$.

Notice that this algorithm generalizes to higher dimensions (as does the line-side test). In this case, in d dimensions, $d+1$ points determine the d-dimensional sphere, and D becomes a $(d+2) \times (d+2)$ determinant.

3 Convex Hulls

Given a set of points S, the *convex hull problem* is to find the smallest convex region that contains all points in S. The boundary of the region is called the *convex hull* of S, and is usually specified by a set of points and their neighbor relation.

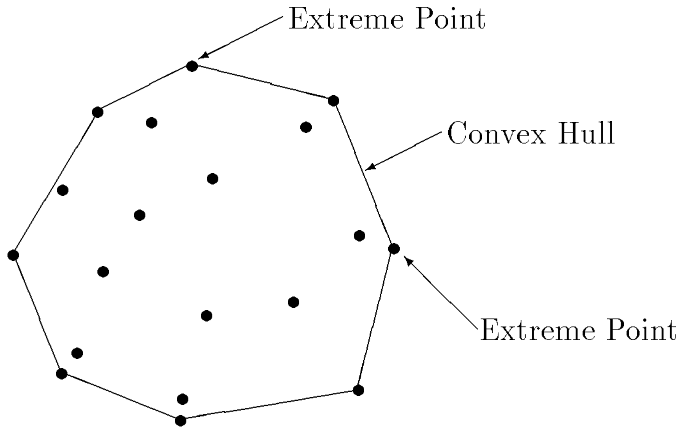


Figure 60: Example of a 2D Convex Hull

Convex hulls are well-defined in any dimension; a 2-dimensional example is given in Figure 60. The 2-D convex hull can be thought of the shape that results when a rubber band is stretched to surround a set of nails, one at each point in S; similarly, the 3-D convex hull is similar to the shape that would be formed if a balloon were stretched to tightly surround the points S in 3-space.

3.1 Dual: Convex Polytopes

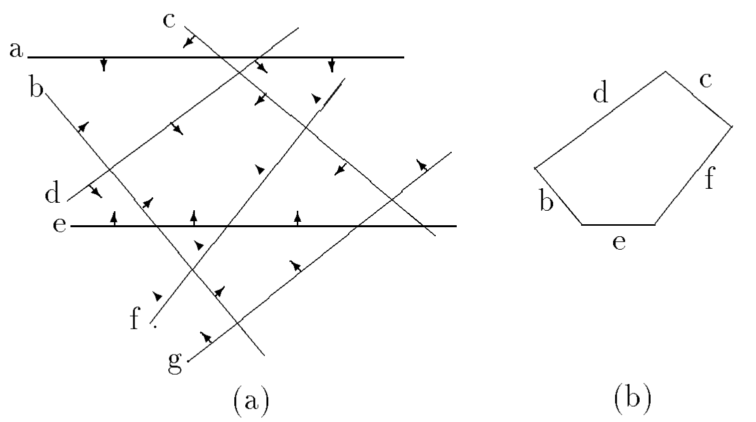


Figure 61: (a) Example 2D Convex Polytope Problem (b) Solution

The convex hull problem has a dual formulation in the problem of *convex polytopes*. The convex polytope problem is to find the region in space that is the intersection of a set of halfspaces (we assume all the halfspaces include the origin). This problem arises in certain linear programming algorithms. Figure 61a illustrates an example of a 2-D convex polytope problem. The solution to a convex polytope problem in general is a list of the "non-redundant" half-spaces (e.g., those lines, planes, or hyperplanes that define the convex region). In the example of Figure 61a, the solution is (b, d, c, f, e), illustrated in Figure 61b.

The convex hull problem and the convex polytope problems are duals, as a problem and solution to one can be mapped into a problem and solution to the other. Thus an algorithm for finding convex hulls can be used to find convex polytopes as well.

The transformation that takes an instance of the convex polytope problem to a convex hull problem is the following, in 2D. Let the ith half-space be defined by the line $\ell_i : a_i x + b_i y = 1$. In higher dimensions, the equation is of the form:

$$\ell_i : \sum_{j=1}^{d} a_{i_j} x_j = 1.$$

Then the dual problem is to find the convex hull of the points p_i, where (in 2D):

$$P_i = \left(\frac{a_i}{a_i^2 + b_i^2}, \frac{b_i}{a_i^2 + b_i^2} \right).$$

In higher dimensions,

$$P_i = \left(\frac{a_{i_1}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \frac{a_{i_2}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \ldots, \frac{a_{i_d}}{a_{i_1}^2 + a_{i_2}^2 + \cdots + a_{i_d}^2}, \right).$$

See Figure 62 for a graphical illustration.

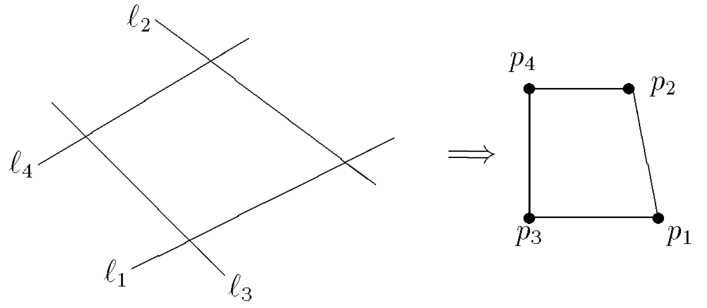


Figure 62: Mapping Convex Polytopes to Convex Hulls

3.2 Algorithms for Finding Convex Hulls

The following is a summary of the most popular algorithms for finding convex hulls. Notice that the algorithms are classified into categories that are similar to those used to classify sorting algorithms. Those marked with a dagger (†) are considered further below. In the time bounds, n is the number of points in the input set S, and h is the number of points in the hull H. The algorithms and bounds we list here are for 2-dimensions although many algorithms generalize to higher dimensions.

- Basic algorithms
  - Giftwrapping†: $O(nh)$
- Incremental algorithms
  - Graham Scan†: $T(\text{sort}) + O(n)$
  - Unordered Insertion: $T(\text{sort}) + O(n)$
- Divide and Conquer algorithms
  - "Merge-based" D&C algorithms
    * Mergehull†: $T(\text{sort}) + O(n)$
  - "Divide-based" D&C algorithms
    * Quickhull†: $O(n)$ best, $O(n^2)$ worst, $O(n \log n)$ average
    * Kirkpatrick-Seidel: $O(n \log h)$
  - Random Sampling: $O(n \log n)$

The theoretical lower-bound on finding a convex hull is $O(n \log h)$.

3.2.1 Giftwrapping

The idea behind the giftwrapping algorithm (in 2-d) is to move from point to point along the convex hull. To start, an extreme point is chosen as the first point on the hull (this can be done easily by choosing the point in S with largest x-value).

Then, on each step, the point in S with the smallest angle α is chosen to be the next point in the hull, where α is defined as the angle between a ray originating at the current point and continuing along the ray that connects the previous and current points, and a ray connecting the current point and the test point. This is explained much more clearly in a picture in Figure 63.

Each step of the algorithm takes $O(n)$ time, since $|S| = n$ and potentially all points in S must be examined to determine the one with minimum angle. Since the algorithm does h steps (one for each point on the hull), the running time of the algorithm is $O(nh)$. If all the points lie on the hull then the algorithm will take $O(n^2)$ time. Because of this worst-case behavior the algorithm is rarely used in practice, although it is parallelizable and does generalize to 3D.

Note that this algorithm also requires additional primitives besides the line-side and in-circle tests.

3.2.2 Graham Scan

The Graham Scan algorithm constructs the convex hull in two pieces (the top and bottom), then connects them together. For each half, it begins by sorting the points in decreasing order of their x coordinates. Then, starting at the largest (or smallest) x, it scans through the points, building the hull along the way. As it encounters a new point, it connects it to

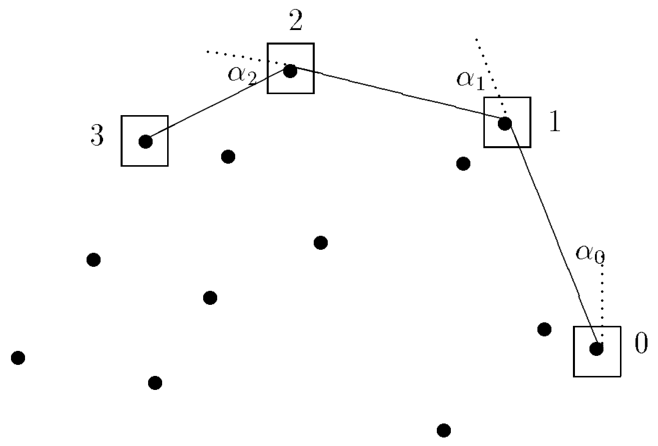


Figure 63: Giftwrapping Algorithm

the previous point, assuming that it will be in the hull. If adding that point causes the hull to be non-convex, the algorithm adds a *bridge* to the appropriate point to ensure that the hull remains convex. This is illustrated in Figure 64; the dashed lines are the bridges added.

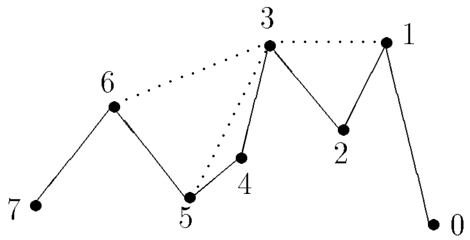


Figure 64: Graham Scan Algorithm

The running time for the Graham Scan algorithm is $T(n) = T(\text{sort}) + O(n)$, since the algorithm must take n steps (one for each point in the scan), and the amortized cost per step can be shown to be $O(1)$. Since $T(\text{sort})$ is $\Omega(n \log n)$ for comparison-based sorts, the Graham Scan algorithm is essentially an $O(n \log n)$ algorithm (unless the data lends itself to a non-comparison-based sorting algorithm).

Finally, note that this algorithm is inherently sequential, and thus does not parallelize well. Even on a sequential machine, it is not the most efficient in practice. Furthermore, it does not generalize to higher dimensions. However, it does have an interesting historical note in that it was developed at Bell Labs in response to a real application, and thus is one of the few hull algorithms that were not theoretically motivated.

3.2.3 Mergehull

The Mergehull algorithm is a divide-and-conquer algorithm that is similar in structure to merge sort. Mergehull has a trivial divide phase, and does most of its work during the merge phase.

Mergehull's divide stage is simple: the algorithm merely splits the input set S into two subsets S_1, S_2 by one of the coordinates. It then recursively solves S_1 and S_2 (e.g., it computes the convex hull for both S_1 and S_2).

After the recursion returns, the algorithm must merge the two convex hulls for S_1 and S_2 into a single convex hull for S. Conceptually, this involves locating two *bridges* between the hulls—line segments connecting two points in S_1 to two points in S_2. Once the bridges are added to connect the two hulls, the interior segments of the subhulls are discarded. See Figure 65 for an illustration. Finding the bridges and discarding the inside edges requires an Overmars dual binary search, which takes time $O(\log n)$. Note that it is not sufficient just to place the bridges between the minima and maxima of the two sub-hulls.

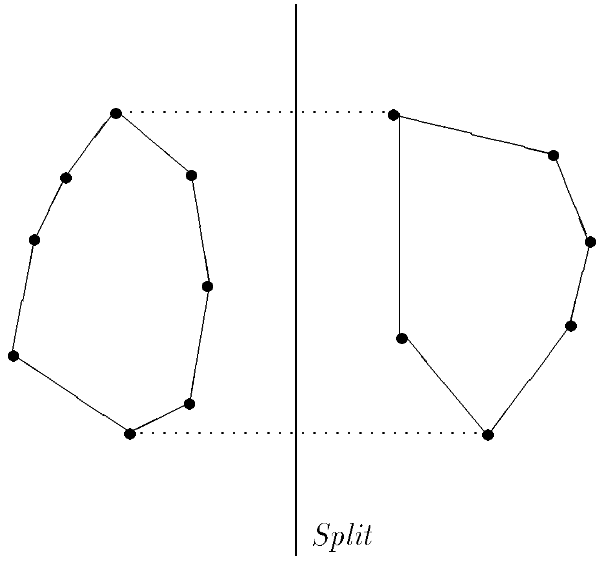


Figure 65: Mergehull Algorithm

If we assume that the points are presorted (to make the subdivision in the divide phase take constant time) and that tree data structures are used throughout, the recurrence for the mergehull algorithm is:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + O(\log n) \\ &= O(n) \end{aligned}$$

When added to the time of the original sort, the total time is $O(n \log n)$.

It is interesting to note that the mergehull algorithm parallelizes, and that the parallel algorithm does well in practice (it obtains a nearly linear speedup with the number of processors). The parallelization is rather simple: the set S of input points is first divided into P regions (rather than in half, as above). Each of the P processors is then assigned one of the regions, for which it sequentially generates a convex hull. Then, the P regions are merged by finding bridges in parallel.

Finally, the mergehull algorithm can be generalized to three or more dimensions. As the dimensionality increases, however, the merge becomes more difficult. For example, in 3D, the bridges between subhulls are no longer line segments, but instead are surfaces that connect the hulls. Finding these surfaces is more expensive: in 3D, the merge takes $O(n)$ time, so the total time for mergehull is $O(n \log n)$, even when the input points are presorted.

3.2.4 Quickhull

The Quickhull algorithm is another variant on the recursive divide-and-conquer method for finding convex hulls. Unlike the mergehull algorithm that we have just seen, quickhull concentrates its work on the divide phase of the algorithm, and uses a trivial merge step. It is thus very similar in structure to the quicksort sorting algorithm, and we will see below that it has similar time complexity.

Quickhull works (in 2D) by locating three points A, B, and C on the convex hull of its input S (with C above both A and B), then recursively refining the hull between AC and CB. Since both halves of the recursion meet at C, which is on the hull, the merge is essentially a no-op. When the algorithm completes, the upper half-hull between A and B will be complete. The algorithm is then repeated for the lower half-hull. Figure 66 details the process graphically.

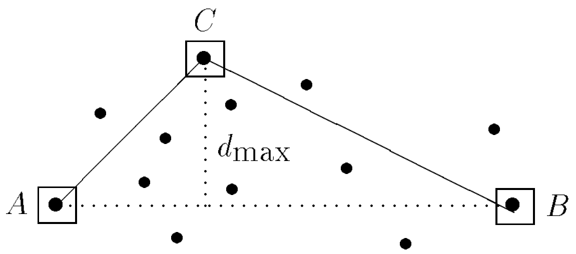


Figure 66: Quickhull Algorithm

The tricky part in Quickhull is implementing the divide stage. A typical algorithm (which works well in the average case but does not guarantee the best time bound) is the following. We start out with two points A, B on the convex hull. These can be found on the first iteration by taking the points in S with minimum and maximum x coordinate. Then, apply the following algorithm:

1. Find the furthest point C from the line joining A and B. To do this, the line-side test (described in Section 2.2) can be used, as it returns the area of the triangle ABC, and

the triangle with largest area is the one that has the furthest point C. Note that, when building the upper half-hull, C should be above AB, whereas C should be above the inverse line BA when building the lower half-hull.

2. Discard all points in triangle ABC, as they clearly will not be on the boundary of the final hull (since ABC is contained within the hull).

3. Recurse on A, C and C, B to fully refine the hull between A and C, and between C and B.

4. Connect the two resulting hulls at C.

5. Repeat for the lower half-hull, and connect it at A and B.

The worst-case running time of the quickhull algorithm is $O(n^2)$, for if the splits (choices of C) are bad, the algorithm may only eliminate one point on each recursion, resulting in a recursion depth of n and a per-recursion cost of n. The input distribution that forces this worst-case time is if all the input points are distributed along a semicircle with exponentially decreasing distance from the left-hand point (see Figure 67).

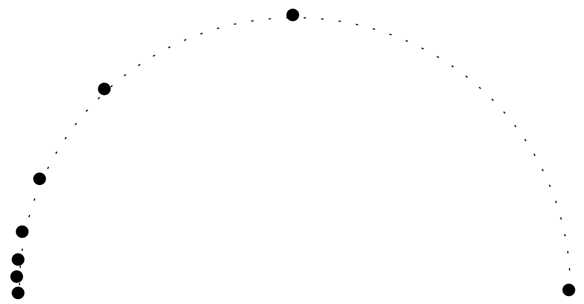


Figure 67: Worst-case Input Distribution for Quickhull

Despite this poor worst-case bound, quickhull runs in $O(n \log n)$ time for most "realistic" point sets. In fact, in practice it can often run as fast as $O(n)$, since the algorithm discards points in step 2, above, reducing the size of the input set for later recursions.

There is also a technique developed by Kirkpatrick and Seidel that locates the ideal "pivot" point C using two-variable linear programming. This variant guarantees the optimal time complexity of $O(n \log h)$. The technique is conceptually similar to the method of using the true sample median to force quicksort to run in $\Theta(n \log n)$. The technique involves finding the bridge segment that connects across the median x coordinate before recursively solving the subhulls themselves. This is the same bridge as found by mergehull, but instead of requiring the solved subhulls for each half it uses linear-programming to find the bridge directly in $O(n)$ time.

Finally, note that quickhull parallelizes easily, and generalizes to three dimensions (in which case the running time is $O(n \log^2 h)$ for an algorithm by Edelsbrunner and Shi).

4 Delaunay Triangulations and Voronoi Diagrams

We will not consider the problem of generating Delaunay triangualations, but we first consider the dual problem of generating Voronoi Diagrams.

4.1 Voronoi Diagrams

A *Voronoi diagram* (or Dirichlet tessellation) is a partition of space into regions $V(p_i)$ based on a set of input points $p_i \in S$ such that:

$$V(p_i) = \{p : |p_i - p| \leq |p_j - p|, \forall\, j \neq i\}.$$

where $|p_i - p|$ indicates the distance from p_i to p (here we assume this distance is Euclidean, but later we will briefly discuss other distance metrics). What this property means is that, for a point p_i, its Voronoi region $V(p_i)$ will contain all points that are closer to p_i than to any other point in S.

Voronoi diagrams are well-defined in any dimension. In two dimensions, the Voronoi regions are convex polygons; in three dimensions, they are convex polyhedra, and so forth. Figure 68 gives an example of a 2D Voronoi diagram.

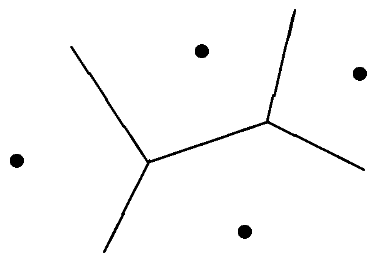


Figure 68: A 2D Voronoi Diagram

Note that there are actually many different ways to represent the diagram. One way is to return for each original point the set of Voronoi neighbors (points that share a Voronoi edge). Another is to return the graph structure formed by the Voronoi edges.

4.2 Delaunay Triangulations

The dual graph of a Voronoi diagram is called a *Delaunay triangulation*. Given a Voronoi diagram, the Delaunay triangulation is formed by connecting the points at the centers of every pair of neighboring Voronoi regions. See Figure 69 for an example. Note that the the Delaunay triangulation will always consist of triangles, since the Delaunay edges join neighboring Voronoi regions, and two neighbor regions of adjacent Voronoi edges will themselves be neighbors. Note that a Delaunay edge is always perpendicular to the Voronoi edge that the two neighbors share, but that the Delaunay edge does not necessarily cross the corresponding Voronoi edge, as illustrated in Figure 70.

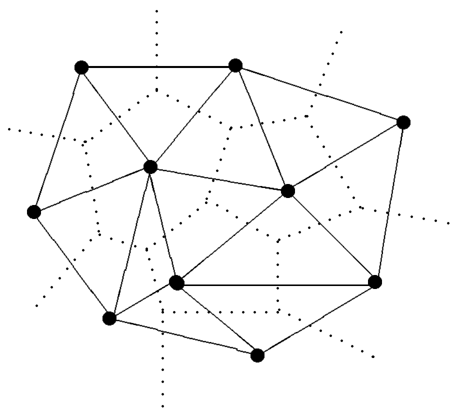


Figure 69: A Delaunay Triangulation (and associated Voronoi diagram)

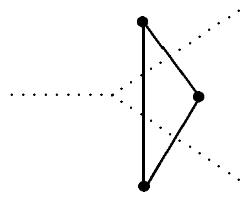


Figure 70: Example of a Delaunay edge that does not cross its Voronoi edge

Finally, note that degeneracies can be encountered in forming a Delaunay triangulation. In particular, if two Voronoi regions intersect only at a corner, there will be more than one possible triangulation (recall that a triangulation contains only non-overlapping triangles). This is illustrated in Figure 71.

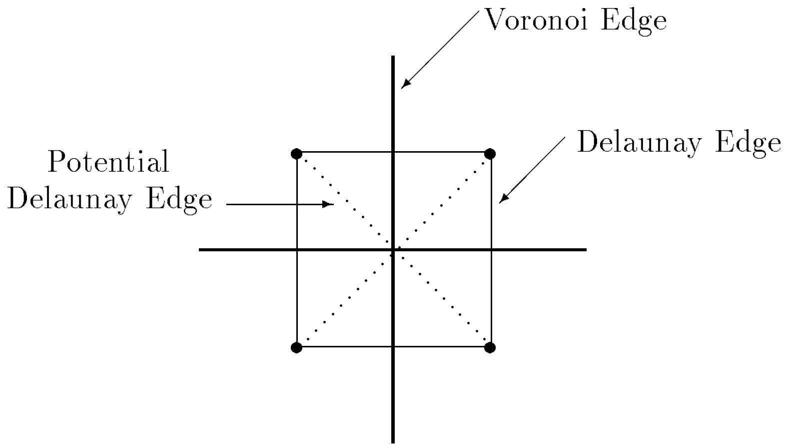


Figure 71: A degenerate case for Delaunay triangulation

4.3 Properties of Delaunay Triangulations

Delaunay triangulations have certain "nice" properties, including:

- The Delaunay triangulation of a non-degenerate set of points is unique. Thus there is only one triangulation that obeys the following set of properties.
- A circle through the three points of a Delaunay triangle contains *no* other points. Also, the intersection of the three corresponding Voronoi edges is at the center of the circle, as shown in Figure 72. This property generalizes to spheres in 3D and so forth.
- The minimum angle across all the angles in all the triangles in a Delaunay triangulation is greater than the minimum angle in any other triangulation of the same points.

These properties tend to result in triangles that are "fat" (closer to equilateral) rather than "skinny". This is a property that is highly desirable in applications, especially finite element simulations where the geometry of the finite elements affects the accuracy and quality of the simulation.

4.4 Algorithms for Delaunay Triangulation

The Delaunay triangulation for a set of points can be found using one of several types of algorithms: a naive direct algorithm, by reducing the problem to a convex hull problem and applying a convex hull algorithm, and by using a direct algorithm. We will consider each of these cases in turn.

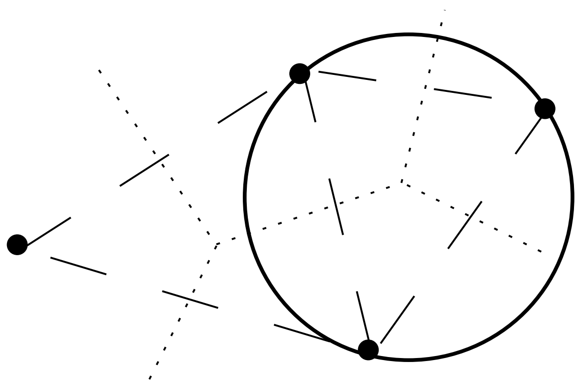


Figure 72: Circle test for Delaunay triangles

4.4.1 Naive Algorithm

A naive way of finding the Delaunay triangulation is the following: for every three-point-subset of the input set, check if any other point is in its circle. Iff not, those three points define a Delaunay triangle.

This algorithm runs in $O(n^4)$ time, since there are n^3 triples of points to check, and each check takes $O(n)$ time (since the in-circle test must be applied n times). Although polynomial-time, this algorithm is quite impractical since, as we will see, there are algorithms that run in $O(n \log n)$ time (and the constant factors in the big-O are quite small).

4.4.2 Solution via Reduction to 3D Convex Hull

A more practical method for finding the Delaunay triangulation of a set of d-dimensional points uses a reduction of the Delaunay triangulation problem to the problem of finding a convex hull in $d + 1$ dimensions. For the typical 2D triangulation case, a 3D convex hull is used.

The reduction is as follows. The original input points P are projected upward onto a three-dimensional parabolic surface centered at the origin. Then, the 3D convex hull of those projected points is found. When the hull is represented as a neighbor graph and projected back down to the two-dimensional plane, the result is the Delaunay triangulation of P. More precisely, given P, we first form:

$$P' = \left\{(x, y, x^2 + y^2) : (x, y) \in P\right\}.$$

Then let C =LowerConvexHull(P'), represented as a neighbor graph. Then, the neighbor graph of C is equivalent to Delaunay(P).

The reason that this works is related to the properties of the Delaunay triangulation, in particular, the property that states that any circle through three points of the input set containing no other points circumscribes a Delaunay triangle through those three points. Recall that any slice of a 3D parabolic surface maps onto a circle in the x-y plane. Also, notice that any face of the 3D convex hull defines a slice through the parabola. Thus the projection of this slice onto the plane is a circle, and the circle will pass through the projection of the three points on the plane. Since the slice is only for a single hull face, there will be no other projected points in that circle, so the projection of the hull face defines a Delaunay triangle. Since this works for all hull faces, the result faces of the convex hull in 3-D are the Delaunay triangles in 2-D (when projected down).

Using this reduction to find the Delaunay triangulation gives an efficient algorithm that runs in $O(n \log n)$. However, there are still better algorithms in practice, such as some of the direct algorithms described in the next section.

4.4.3 Direct Algorithms

The following algorithms can be used to find either Voronoi diagrams or Delaunay triangulations, as the two problems are duals of each other. Those algorithms marked with a dagger (†) are easy to parallelize.

- Intersecting Halfspaces†: $O(n^2 \log n)$

  This algorithm works as follows: for each input point, draw a line to every other point. Generate a halfspace separating the input point from every other point (by bisecting the lines drawn in the first step). Compute the halfspace intersection of these halfspaces (a convex hull problem running in $O(n \log n)$). This gives a Voronoi region. Repeat for all n points.

- Radial Sweep

- Incremental Construction: $O(n \log n)$ (expected time)

- Plane Sweep: $O(n \log n)$

- Divide and Conquer Algorithms

  - Merge-based: $O(n \log n)$
  - Split-based†: $O(n \log n)$

In this class we will only discuss the incremental and the split-based divide-and-conquer algorithms.