CMSC 754
Computational Geometry[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2002

Lecture 1: Introduction

What is Computational Geometry? Computational geometry is a term claimed by a number of different groups. The term was coined perhaps first by Marvin Minsky in his book "Perceptrons", which was about pattern recognition, and it has also been used often to describe algorithms for manipulating curves and surfaces in solid modeling. Its most widely recognized use, however, is to describe the subfield of algorithm theory that involves the design and analysis of efficient algorithms for problems involving geometric input and output.

The field of computational geometry developed rapidly in the late 70's and through the 80's and 90's, and it still continues to develop. Historically, computational geometry developed as a generalization of the study of algorithms for sorting and searching in 1-dimensional space to problems involving multi-dimensional inputs. It also developed to some extent as a restriction of computational graph theory by considering graphs that arise naturally in geometric settings (e.g., networks of roads or wires in the plane).

Because of its history, the field of computational geometry has focused mostly on problems in 2-dimensional space and to a lesser extent in 3-dimensional space. When problems are considered in multi-dimensional spaces, it is usually assumed that the dimension of the space is a small constant (say, 10 or lower). Because the field was developed by researchers whose training was in discrete algorithms (as opposed to numerical analysis) the field has also focused more on the discrete nature of geometric problems, as opposed to continuous issues. The field primarily deals with straight or flat objects (lines, line segments, polygons, planes, and polyhedra) or simple curved objects such as circles. This is in contrast, say, to fields such as solid modeling, which focus on problems involving more complex curves and surfaces.

A Typical Problem in Computational Geometry: Here is an example of a typical problem, called the *shortest path problem*. Given a set polygonal obstacles in the plane, find the shortest obstacle-avoiding path from some given start point to a given goal point. Although it is possible to reduce this to a shortest path problem on a graph (called the *visibility graph*, which we will discuss later this semester), and then apply a nongeometric algorithm such as Dijkstra's algorithm, it seems that by solving the problem in its geometric domain it should be possible to devise more efficient solutions. This is one of the main reasons for the growth of interest in geometric algorithms.

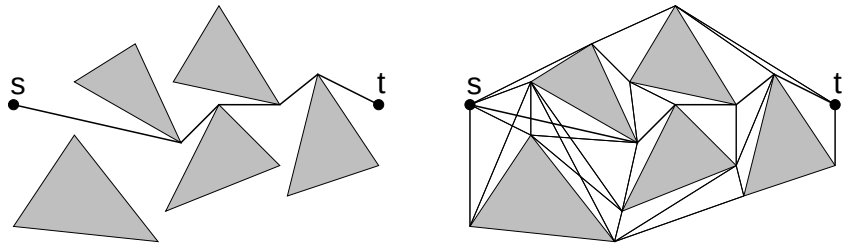

Figure 1: Shortest path problem.

The measure of the quality of an algorithm in computational geometry has traditionally been its *asymptotic worst-case running time*. Thus, an algorithm running in $O(n)$ time is better than one running in $O(n \log n)$ time which is better than one running in $O(n^2)$ time. (This particular problem can be solved in $O(n^2 \log n)$ time by a fairly simple algorithm, in $O(n \log n)$ by a relatively complex algorithm, and it can be approximated quite well by an algorithm whose running time is $O(n \log n)$.) In some cases *average case* running time is considered instead. However, for many types of geometric inputs it is difficult to define input distributions that are both easy to analyze and representative of typical inputs.

There are many fields of computer science that deal with solving problems of a geometric nature. These include computer graphics, computer vision and image processing, robotics, computer-aided design and manufacturing, computational fluid-dynamics, and geographic information systems, to name a few. One of the goals of computational geometry is to provide the basic geometric tools needed from which application areas can then build

their programs. There has been significant progress made towards this goal, but it is still far from being fully realized.

Limitations of Computational Geometry: There are some fairly natural reasons why computational geometry may never fully address the needs of all these applications areas, and these limitations should be understood before undertaking this course. One is the *discrete nature* of computational geometry. In some sense any problem that is solved on digital computers must be expressed in a discrete form, but many applications areas deal with discrete approximations to continuous phenomenon. For example, in image processing the image may be a discretization of a continuous 2-dimensional gray-scale function, and in robotics issues of vibration, oscillation in dynamic control systems are of a continuous nature. Nonetheless, there are many applications in which objects are of a very discrete nature. For example, in geographic information systems, road networks are discretized into collections of line segments.

The other limitation is the fact that computational geometry deals primarily with straight or flat objects. To a large extent, this is a result of the fact that computational geometers were not trained in geometry, but in discrete algorithm design. So they chose problems for which geometry and numerical computation plays a fairly small role. Much of solid modeling, fluid dynamics, and robotics, deals with objects that are modeled with curved surfaces. However, it is possible to approximate curved objects with piecewise planar polygons or polyhedra. This assumption has freed computational geometry to deal with the combinatorial elements of most of the problems, as opposed to dealing with numerical issues. This is one of the things that makes computational geometry fun to study, you do not have to learn a lot of analytic or differential geometry to do it. But, it does limit the applications of computational geometry.

One more limitation is that computational geometry has focused primarily on 2-dimensional problems, and 3-dimensional problems to a limited extent. The nice thing about 2-dimensional problems is that they are easy to visualize and easy to understand. But many of the daunting applications problems reside in 3-dimensional and higher dimensional spaces. Furthermore, issues related to topology are much cleaner in 2- and 3-dimensional spaces than in higher dimensional spaces.

Trends in CG in the 80's and 90's: In spite of these limitations, there is still a remarkable array of interesting problems that computational geometry has succeeded in addressing. Throughout the 80's the field developed many techniques for the design of efficient geometric algorithms. These include well-known methods such as divide-and-conquer and dynamic programming, along with a number of newly discovered methods that seem to be particularly well suited to geometric algorithm. These include plane-sweep, randomized incremental constructions, duality-transformations, and fractional cascading.

One of the major focuses of this course will be on understanding technique for designing efficient geometric algorithms. A major part of the assignments in this class will consist of designing and/or analyzing the efficiency of problems of a discrete geometric nature.

However throughout the 80's there a nagging gap was growing between the "theory" and "practice" of designing geometric algorithms. The 80's and early 90's saw many of the open problems of computational geometry solved in the sense that theoretically optimal algorithms were developed for them. However, many of these algorithms were nightmares to implement because of the complexity of the algorithms and the data structures that they required. Furthermore, implementations that did exist were often sensitive to geometric degeneracies that caused them to produce erroneous results or abort. For example, a programmer designing an algorithm that computes the intersections of a set of line segments may not consider the situation when three line segments intersect in a single point. In this rare situation, the data structure being used may be corrupted, and the algorithm aborts.

Much of the recent work in computational geometry has dealt with trying to make the theoretical results of computational geometry accessible to practitioners. This has been done by simplifying existing algorithms, dealing with geometric degeneracies, and producing libraries of geometric procedures. This process is still underway. Whenever possible, we will discuss the simplest known algorithm for solving a problem. Often these algorithms will be randomized algorithms. We will also discuss (hopefully without getting too bogged down in

details) some of the techniques for dealing with degenerate situations in order to produce clean and yet robust geometric software.

Overview of the Semester: Here are some of the topics that we will discuss this semester.

Convex Hulls: Convexity is a very important geometric property. A geometric set is *convex* if for every two points in the set, the line segment joining them is also in the set. One of the first problems identified in the field of computational geometry is that of computing the smallest convex shape, called the *convex hull*, that encloses a set of points.

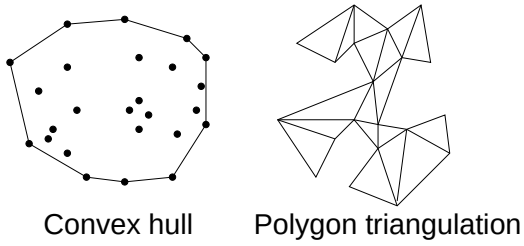

Figure 2: Convex hulls and polygon triangulation.

Intersections: One of the most basic geometric problems is that of determining when two sets of objects intersect one another. Determining whether complex objects intersect often reduces to determining which individual pairs of primitive entities (e.g., line segments) intersect. We will discuss efficient algorithms for computing the intersections of a set of line segments.

Triangulation and Partitioning: Triangulation is a catchword for the more general problem of subdividing a complex domain into a disjoint collection of "simple" objects. The simplest region into which one can decompose a planar object is a triangle (a *tetrahedron* in 3-d and *simplex* in general). We will discuss how to subdivide a polygon into triangles and later in the semester discuss more general subdivisions into trapezoids.

Low-dimensional Linear Programming: Many optimization problems in computational geometry can be stated in the form of a linear programming problem, namely, find the extreme points (e.g. highest or lowest) that satisfies a collection of linear inequalities. Linear programming is an important problem in the combinatorial optimization, and people often need to solve such problems in hundred to perhaps thousand dimensional spaces. However there are many interesting problems (e.g. find the smallest disc enclosing a set of points) that can be posed as low dimensional linear programming problems. In low-dimensional spaces, very simple efficient solutions exist.

Line arrangements and duality: Perhaps one of the most important mathematical structures in computational geometry is that of an arrangement of lines (or generally the arrangement of curves and surfaces). Given n lines in the plane, an arrangement is just the graph formed by considering the intersection points as vertices and line segments joining them as edges. We will show that such a structure can be constructed in $O(n^2)$ time. These reason that this structure is so important is that many problems involving points can be transformed into problems involving lines by a method of duality. For example, suppose that you want to determine whether any three points of a planar point set are collinear. This could be determines in $O(n^3)$ time by brute-force checking of each triple. However, if the points are dualized into lines, then (as we will see later this semester) this reduces to the question of whether there is a vertex of degree greater than 4 in the arrangement.

Voronoi Diagrams and Delaunay Triangulations: Given a set S of points in space, one of the most important problems is the nearest neighbor problem. Given a point that is not in S which point of S is closest to it? One of the techniques used for solving this problem is to subdivide space into regions, according to which point is closest. This gives rise to a geometric partition of space called a *Voronoi diagram*. This geometric

structure arises in many applications of geometry. The dual structure, called a *Delaunay triangulation* also has many interesting properties.

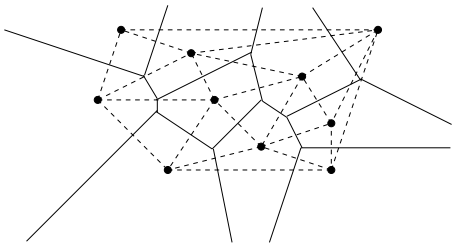

Figure 3: Voronoi diagram and Delaunay triangulation.

Search: Geometric search problems are of the following general form. Given a data set (e.g. points, lines, polygons) which will not change, preprocess this data set into a data structure so that some type of query can be answered as efficiently as possible. For example, a *nearest neighbor* search query is: determine the point of the data set that is closest to a given query point. A *range query* is: determine the set of points (or count the number of points) from the data set that lie within a given region. The region may be a rectangle, disc, or polygonal shape, like a triangle.

Lecture 2: Fixed-Radius Near Neighbors and Geometric Basics

Reading: The material on the Fixed-Radius Near Neighbor problem is taken from the paper: "The complexity of finding fixed-radius near neighbors," by J. L. Bentley, D. F. Stanat, and E. H. Williams, *Information Processing Letters*, 6(6), 1977, 209–212. The material on affine and Euclidean geometry is covered in many textbooks on basic geometry and computer graphics.

Fixed-Radius Near Neighbor Problem: As a warm-up exercise for the course, we begin by considering one of the oldest results in computational geometry. This problem was considered back in the mid 70's, and is a fundamental problem involving a set of points in dimension d. We will consider the problem in the plane, but the generalization to higher dimensions will be straightforward.

We assume that we are given a set P of n points in the plane. As will be our usual practice, we assume that each point p is represented by its (x, y) coordinates, denoted (p_x, p_y). The Euclidean distance between two points p and q, denoted $|pq|$ is

$$|pq| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Given the set P and a distance $r > 0$, our goal is to report all pairs of distinct points $p, q \in P$ such that $|pq| \leq r$. This is called the *fixed-radius near neighbor (reporting) problem*.

Reporting versus Counting: We note that this is a *reporting* problem, which means that our objective is to report all such pairs. This is in contrast to the corresponding *counting* problem, in which the objective is to return a count of the number of pairs satisfying the distance condition.

It is usually easier to solve reporting problems optimally than counting problems. This may seem counterintuitive at first (after all, if you can report, then you can certainly count). The reason is that we know that any algorithm that reports some number k of pairs must take at least $\Omega(k)$ time. Thus if k is large, a reporting algorithm has the luxury of being able to run for a longer time and still claim to be optimal. In contrast, we cannot apply such a lower bound to a counting algorithm.

Site event: Let p_i be the current site. We shoot a vertical ray up to determine the arc that lies immediately above this point in the beach line. Let p_j be the corresponding site. We split this arc, replacing it with the triple of arcs p_j, p_i, p_j which we insert into the beach line. Also we create new (dangling) edge for the Voronoi diagram which lies on the bisector between p_i and p_j. Some old triples that involved p_j may be deleted and some new triples involving p_i will be inserted.

For example, suppose that prior to insertion we had the beach-line sequence

$$\langle p_1, p_2, p_j, p_3, p_4 \rangle.$$

The insertion of p_i splits the arc p_j into two arcs, denoted p'_j and p''_j. Although these are separate arcs, they involve the same site, p_j. The new sequence is

$$\langle p_1, p_2, p'_j, p_i, p''_j, p_3, p_4 \rangle.$$

Any event associated with the old triple p_2, p_j, p_3 will be deleted. We also consider the creation of new events for the triples p_2, p'_j, p_i and p_i, p''_j, p_3. Note that the new triple p'_j, p_i, p''_j cannot generate an event because it only involves two distinct sites.

Vertex event: Let p_i, p_j, and p_k be the three sites that generate this event (from left to right). We delete the arc for p_j from the beach line. We create a new vertex in the Voronoi diagram, and tie the edges for the bisectors (p_i, p_j), (p_j, p_k) to it, and start a new edge for the bisector (p_i, p_k) that starts growing down below. Finally, we delete any events that arose from triples involving this arc of p_j, and generate new events corresponding to consecutive triples involving p_i and p_k (there are two of them).

For example, suppose that prior to insertion we had the beach-line sequence

$$\langle p_1, p_i, p_j, p_k, p_2 \rangle.$$

After the event we have the sequence

$$\langle p_1, p_i, p_k, p_2 \rangle.$$

We remove any events associated with the triples p_1, p_i, p_j and p_j, p_k, p_2. (The event p_i, p_j, p_k has already been removed since we are processing it now.) We also consider the creation of new events for the triples p_1, p_i, p_k and p_i, p_k, p_2.

The analysis follows a typical analysis for plane sweep. Each event involves $O(1)$ processing time plus a constant number accesses to the various data structures. Each of these accesses takes $O(\log n)$ time, and the data structures are all of size $O(n)$. Thus the total time is $O(n \log n)$, and the total space is $O(n)$.

Lecture 17: Delaunay Triangulations

Reading: Chapter 9 in the 4M's.

Delaunay Triangulations: Last time we gave an algorithm for computing Voronoi diagrams. Today we consider the related structure, called a *Delaunay triangulation* (DT). Since the Voronoi diagram is a planar graph, we may naturally ask what is the corresponding dual graph. The vertices for this dual graph can be taken to be the sites themselves. Since (assuming general position) the vertices of the Voronoi diagram are of degree three, it follows that the faces of the dual graph (excluding the exterior face) will be triangles. The resulting dual graph is a triangulation of the sites, the Delaunay triangulation.

Delaunay triangulations have a number of interesting properties, that are consequences of the structure of the Voronoi diagram.

Convex hull: The boundary of the exterior face of the Delaunay triangulation is the boundary of the convex hull of the point set.

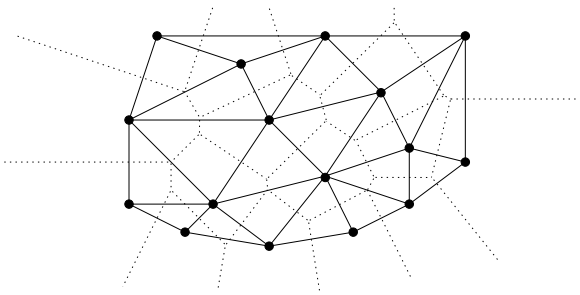

Figure 64: Delaunay triangulation.

Circumcircle property: The circumcircle of any triangle in the Delaunay triangulation is empty (contains no sites of P).

Empty circle property: Two sites p_i and p_j are connected by an edge in the Delaunay triangulation, if and only if there is an empty circle passing through p_i and p_j. (One direction of the proof is trivial from the circumcircle property. In general, if there is an empty circumcircle passing through p_i and p_j, then the center c of this circle is a point on the edge of the Voronoi diagram between p_i and p_j, because c is equidistant from each of these sites and there is no closer site.)

Closest pair property: The closest pair of sites in P are neighbors in the Delaunay triangulation. (The circle having these two sites as its diameter cannot contain any other sites, and so is an empty circle.)

If the sites are not in general position, in the sense that four or more are cocircular, then the Delaunay triangulation may not be a triangulation at all, but just a planar graph (since the Voronoi vertex that is incident to four or more Voronoi cells will induce a face whose degree is equal to the number of such cells). In this case the more appropriate term would be *Delaunay graph*. However, it is common to either assume the sites are in general position (or to enforce it through some sort of symbolic perturbation) or else to simply triangulate the faces of degree four or more in any arbitrary way. Henceforth we will assume that sites are in general position, so we do not have to deal with these messy situations.

Given a point set P with n sites where there are h sites on the convex hull, it is not hard to prove by Euler's formula that the Delaunay triangulation has $2n-2-h$ triangles, and $3n-3-h$ edges. The ability to determine the number of triangles from n and h only works in the plane. In 3-space, the number of tetrahedra in the Delaunay triangulation can range from $O(n)$ up to $O(n^2)$. In dimension n, the number of simplices (the d-dimensional generalization of a triangle) can range as high as $O(n^{\lceil d/2 \rceil})$.

Minimum Spanning Tree: The Delaunay triangulation possesses some interesting properties that are not directly related to the Voronoi diagram structure. One of these is its relation to the minimum spanning tree. Given a set of n points in the plane, we can think of the points as defining a *Euclidean graph* whose edges are all $\binom{n}{2}$ (undirected) pairs of distinct points, and edge (p_i, p_j) has weight equal to the Euclidean distance from p_i to p_j. A minimum spanning tree is a set of $n-1$ edges that connect the points (into a free tree) such that the total weight of edges is minimized. We could compute the MST using Kruskal's algorithm. Recall that Kruskal's algorithm works by first sorting the edges and inserting them one by one. We could first compute the Euclidean graph, and then pass the result on to Kruskal's algorithm, for a total running time of $O(n^2 \log n)$.

However there is a much faster method based on Delaunay triangulations. First compute the Delaunay triangulation of the point set. We will see later that it can be done in $O(n \log n)$ time. Then compute the MST of the Delaunay triangulation by Kruskal's algorithm and return the result. This leads to a total running time of $O(n \log n)$. The reason that this works is given in the following theorem.

Theorem: The minimum spanning tree of a set of points P (in any dimension) is a subgraph of the Delaunay triangulation.

Proof: Let T be the MST for P, let $w(T)$ denote the total weight of T. Let a and b be any two sites such that ab is an edge of T. Suppose to the contrary that ab is not an edge in the Delaunay triangulation. This implies that there is no empty circle passing through a and b, and in particular, the circle whose diameter is the segment ab contains a site, call it c. (See the figure below.)

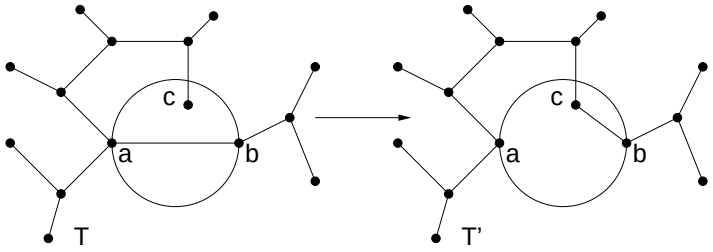

Figure 65: The Delaunay triangulation and MST.

The removal of ab from the MST splits the tree into two subtrees. Assume without loss of generality that c lies in the same subtree as a. Now, remove the edge ab from the MST and add the edge bc in its place. The result will be a spanning tree T' whose weight is

$$w(T') = w(T) + |bc| - |ab| < w(T).$$

The last inequality follows because ab is the diameter of the circle, implying that $|bc| < |ab|$. This contradicts the hypothesis that T is the MST, completing the proof.

By the way, this suggests another interesting question. Among all triangulations, we might ask, does the Delaunay triangulation minimize the total edge length? The answer is no (and there is a simple four-point counterexample). However, this claim was made in a famous paper on Delaunay triangulations, and you may still hear it quoted from time to time. The triangulation that minimizes total edge weight is called the *minimum weight triangulation*. To date, no polynomial time algorithm is known for computing it, and the problem is not known to be NP-complete.

Maximizing Angles and Edge Flipping: Another interesting property of Delaunay triangulations is that among all triangulations, the Delaunay triangulation maximizes the minimum angle. This property is important, because it implies that Delaunay triangulations tend to avoid skinny triangles. This is useful for many applications where triangles are used for the purposes of interpolation.

In fact a much stronger statement holds as well. Among all triangulations with the same smallest angle, the Delaunay triangulation maximizes the second smallest angle, and so on. In particular, any triangulation can be associated with a sorted *angle sequence*, that is, the increasing sequence of angles $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ appearing in the triangles of the triangulation. (Note that the length of the sequence will be the same for all triangulations of the same point set, since the number depends only on n and h.)

Theorem: Among all triangulations of a given point set, the Delaunay triangulation has the lexicographically largest angle sequence.

Before getting into the proof, we should recall a few basic facts about angles from basic geometry. First, recall that if we consider the circumcircle of three points, then each angle of the resulting triangle is exactly half the angle of the minor arc subtended by the opposite two points along the circumcircle. It follows as well that if a point is inside this circle then it will subtend a larger angle and a point that is outside will subtend a smaller angle. This in the figure part (a) below, we have $\theta_1 > \theta_2 > \theta_3$.

We will not give a formal proof of the theorem. (One appears in the text.) The main idea is to show that for any triangulation that fails to satisfy the empty circle property, it is possible to perform a local operation, called

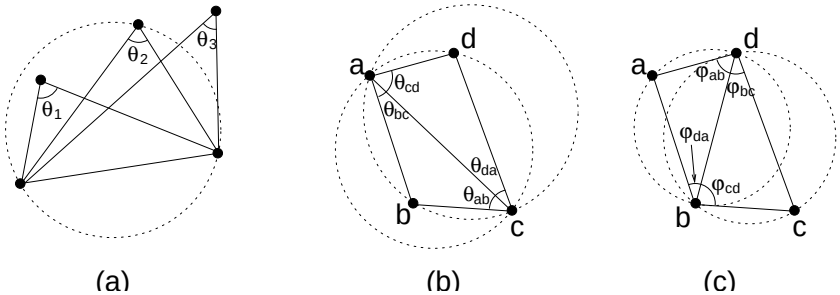

Figure 66: Angles and edge flips.

an *edge flip*, which increases the lexicographical sequence of angles. An edge flip is an important fundamental operation on triangulations in the plane. Given two adjacent triangles $\triangle abc$ and $\triangle cda$, such that their union forms a convex quadrilateral $abcd$, the edge flip operation replaces the diagonal ac with bd. Note that it is only possible when the quadrilateral is convex. Suppose that the initial triangle pair violates the empty circle condition, in that point d lies inside the circumcircle of $\triangle abc$. (Note that this implies that b lies inside the circumcircle of $\triangle cda$.) If we flip the edge it will follow that the two circumcircles of the two resulting triangles, $\triangle abd$ and $\triangle bcd$ are now empty (relative to these four points), and the observation above about circles and angles proves that the minimum angle increases at the same time. In particular, in the figure above, we have

$$\phi_{ab} > \theta_{ab} \qquad \phi_{bc} > \theta_{bc} \qquad \phi_{cd} > \theta_{cd} \qquad \phi_{da} > \theta_{da}.$$

There are two other angles that need to be compared as well (can you spot them?). It is not hard to show that, after swapping, these other two angles cannot be smaller than the minimum of θ_{ab}, θ_{bc}, θ_{cd}, and θ_{da}. (Can you see why?)

Since there are only a finite number of triangulations, this process must eventually terminate with the lexicographically maximum triangulation, and this triangulation must satisfy the empty circle condition, and hence is the Delaunay triangulation.

Lecture 18: Delaunay Triangulations: Incremental Construction

Reading: Chapter 9 in the 4M's.

Constructing the Delaunay Triangulation: We will present a simple randomized $O(n \log n)$ expected time algorithm for constructing Delaunay triangulations for n sites in the plane. The algorithm is remarkably similar in spirit to the randomized algorithm for trapezoidal map algorithm in that not only builds the triangulation but also provides a point-location data structure as well. We will not discuss the point-location data structure in detail, but the details are easy to fill in.

As with any randomized incremental algorithm, the idea is to insert sites in random order, one at a time, and update the triangulation with each new addition. The issues involved with the analysis will be showing that after each insertion the expected number of structural changes in the diagram is $O(1)$. As with other incremental algorithm, we need some way of keeping track of where newly inserted sites are to be placed in the diagram. We will describe a somewhat simpler method than the one we used in the trapezoidal map. Rather than building a data structure, this one simply puts each of the uninserted points into a bucket according to the triangle that contains it in the current triangulation. In this case, we will need to argue that the expected number of times that a site is rebucketed is $O(\log n)$.

Incircle Test: The basic issue in the design of the algorithm is how to update the triangulation when a new site is added. In order to do this, we first investigate the basic properties of a Delaunay triangulation. Recall that a

triangle $\triangle abc$ is in the Delaunay triangulation, if and only if the circumcircle of this triangle contains no other site in its interior. (Recall that we make the general position assumption that no four sites are cocircular.) How do we test whether a site d lies within the interior of the circumcircle of $\triangle abc$? It turns out that this can be reduced to a determinant computation. First off, let us assume that the sequence $\langle abcd \rangle$ defines a counterclockwise convex polygon. (If it does not because d lies inside the triangle $\triangle abc$ then clearly d lies in the circumcircle for this triangle. Otherwise, we can always relabel abc so this is true.) Under this assumption, d lies in the circumcircle determined by the $\triangle abc$ if and only if the following determinant is positive. This is called the *incircle test*. We will assume that this primitive is available to us.

$$\text{inCircle}(a, b, c, d) \;=\; \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} > 0.$$

We will not prove the correctness of this test, but a simpler assertion, namely that if the above determinant is equal to zero, then the four points are cocircular. The four points are cocircular if there exists a center point $q = (q_x, q_y)$ and a radius r such that

$$(a_x - q_x)^2 + (a_y - q_y)^2 = r^2,$$

and similarly for the other three points. Expanding this and collecting common terms we have

$$(a_x^2 + a_y^2) - 2q_x(a_x) - 2q_y(a_y) + (q_x^2 + q_y^2 - r^2)(1) = 0,$$

and similarly for the other three points, b, c, and d. If we let X_1, X_2, X_3 and X_4 denote the columns of the above matrix we have

$$X_3 - 2q_x X_1 - 2q_y X_2 + (q_x^2 + q_y^2 - r^2)X_4 = 0.$$

Thus, the columns of the above matrix are linearly dependent, implying that their determinant is zero. We will leave the completion of the proof as an exercise. Next time we will show how to use the incircle test to update the triangulation, and present the complete algorithm.

Incremental update: When we add the next site, p_i, the problem is to convert the current Delaunay triangulation into a new Delaunay triangulation containing this site. This will be done by creating a non-Delaunay triangulation containing the new site, and then incrementally "fixing" this triangulation to restore the Delaunay properties. The fundamental changes will be: (1) adding a site to the middle of a triangle, and creating three new edges, and (2) performing an *edge flip*. Both of these operations can be performed in $O(1)$ time, assuming that the triangulation is maintained, say, as a DCEL.

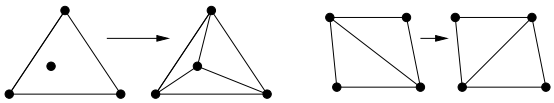

Figure 67: Basic triangulation changes.

The algorithm that we will describe has been known for many years, but was first analyzed by Guibas, Knuth, and Sharir. The algorithm starts within an initial triangulation such that all the points lie in the convex hull. This can be done by enclosing the points in a large triangle. Care must be taken in the construction of this enclosing triangle. It is not sufficient that it simply contain all the points. It should be the case that these points do not lie in the circumcircles of any of the triangles of the final triangulation. Our book suggests computing a triangle that contains all the points, but then fudging with the incircle test so that these points act as if they are invisible.

The sites are added in random order. When a new site p is added, we find the triangle $\triangle abc$ of the current triangulation that contains this site (we will see how later), insert the site in this triangle, and join this site to

the three surrounding vertices. This creates three new triangles, $\triangle pab$, $\triangle pbc$, and $\triangle pca$, each of which may or may not satisfy the empty-circle condition. How do we test this? For each of the triangles that have been added, we check the vertex of the triangle that lies on the opposite side of the edge that does not include p. (If there is no such vertex, because this edge is on the convex hull, then we are done.) If this vertex fails the incircle test, then we swap the edge (creating two new triangles that are adjacent to p). This replaces one triangle that was incident to p with two new triangles. We repeat the same test with these triangles. An example is shown in the figure below.

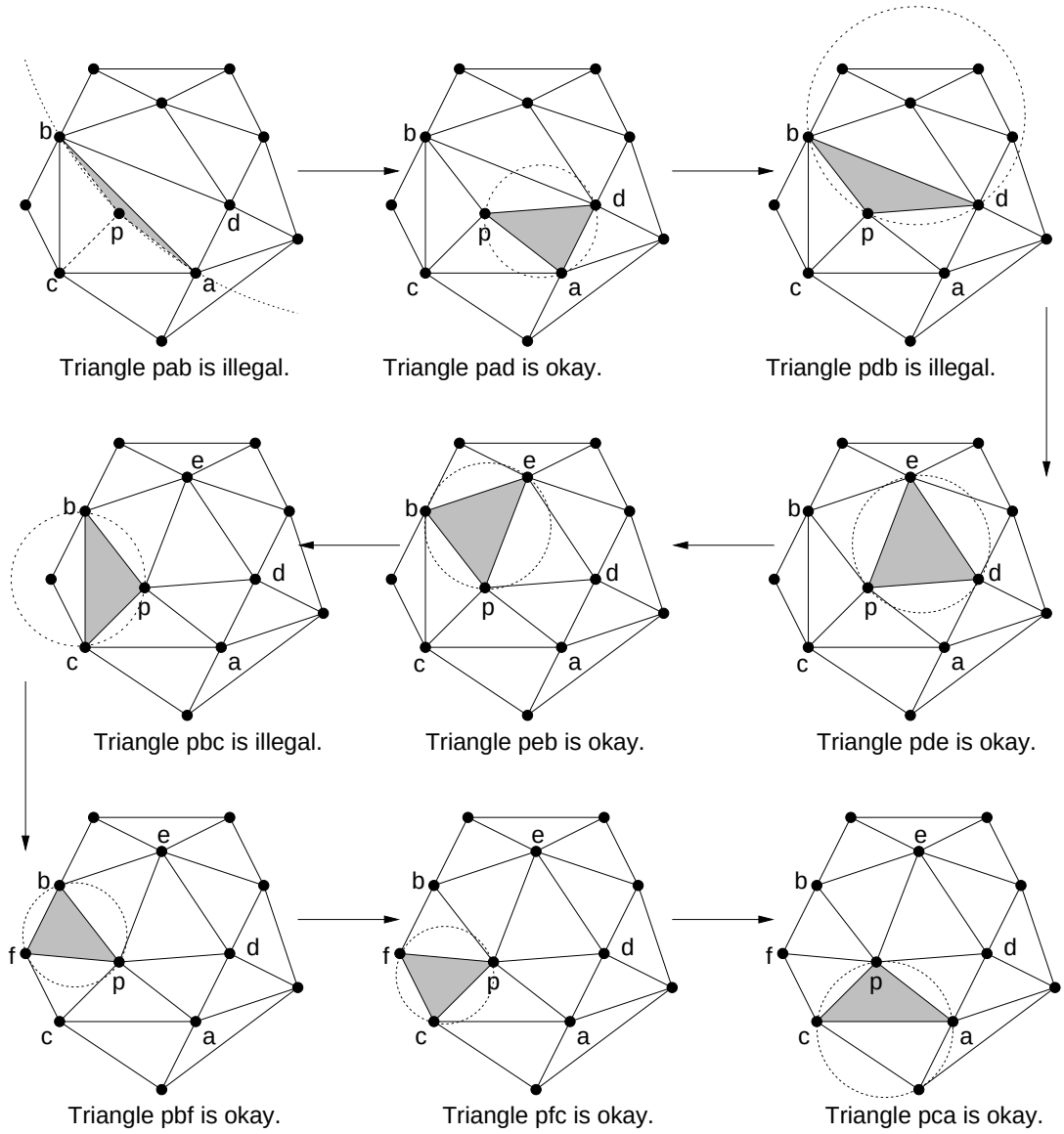

Figure 68: Point insertion.

The code for the incremental algorithm is shown in the figure below. The current triangulation is kept in a global data structure. The edges in the following algorithm are actually pointers to the DCEL.

There is only one major issue in establishing the correctness of the algorithm. When we performed empty-circle tests, we only tested triangles containing the site p, and only sites that lay on the opposite side of an edge of such

```
Insert(p) {
    Find the triangle △abc containing p;
    Insert edges pa, pb, and pc into triangulation;
    SwapTest(ab);              // Fix the surrounding edges
    SwapTest(bc);
    SwapTest(ca);
}

SwapTest(ab) {
    if (ab is an edge on the exterior face) return;
    Let d be the vertex to the right of edge ab;
    if (inCircle(p, a, b, d) {     // d violates the incircle test
        Flip edge ab for pd;
        SwaptTest(ad);             // Fix the new suspect edges
        SwaptTest(db);
    }
}
```

a triangle. We need to establish that these tests are sufficient to guarantee that the final triangulation is indeed Delaunay.

First, we observe that it suffices to consider only triangles that contain p in their circumcircle. The reason is that p is the only newly added site, it is the only site that can cause a violation of the empty-circle property. Clearly the triangle that contained p must be removed, since its circumcircle definitely contains p. Next, we need to argue that it suffices to check only the neighboring triangles after each edge flip. Consider a triangle $\triangle pab$ that contains p and consider the vertex d belonging to the triangle that lies on the opposite side of edge ab. We argue that if d lies outside the circumcircle of pab, then no other point of the point set can lie within this circumcircle.

A complete proof of this takes some effort, but here is a simple justification. What could go wrong? It might be that d lies outside the circumcircle, but there is some other site, say, a vertex e of a triangle adjacent to d, that lies inside the circumcircle. This is illustrated in the following figure. We claim that this cannot happen. It can be shown that if e lies within the circumcircle of $\triangle pab$, then a must lie within the circumcircle of $\triangle bde$. (The argument is a exercise in geometry.) However, this violates the assumption that the initial triangulation (before the insertion of p) was a Delaunay triangulation.

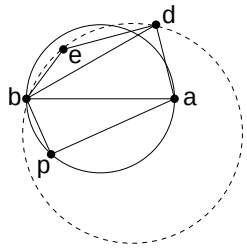

Figure 69: Proof of sufficiency of testing neighboring sites.

As you can see, the algorithm is very simple. The only things that need to be implemented are the DCEL (or other data structure) to store the triangulation, the incircle test, and locating the triangle that contains p. The first two tasks are straightforward. The point location involves a little thought.

Point Location: The point location can be accomplished by one of two means. Our text discusses the idea of building

a history graph point-location data structure, just as we did in the trapezoid map case. A simpler approach is based on the idea of maintaining the uninserted sites in a set of *buckets*. Think of each triangle of the current triangulation as a *bucket* that holds the sites that lie within this triangle and have yet to be inserted. Whenever an edge is flipped, or when a triangle is split into three triangles through point insertion, some old triangles are destroyed and are replaced by a constant number of new triangles. When this happens, we lump together all the sites in the buckets corresponding to the deleted triangles, create new buckets for the newly created triangles, and reassign each site into its new bucket. Since there are a constant number of triangles created, this process requires $O(1)$ time per site that is rebucketed.

Analysis: To analyze the expected running time of algorithm we need to bound two quantities: (1) how many changes are made in the triangulation on average with the addition of each new site, and (2) how much effort is spent in rebucketing sites. As usual, our analysis will be in the worst-case (for any point set) but averaged over all possible insertion orders.

We argue first that the expected number of edge changes with each insertion is $O(1)$ by a simple application of backwards analysis. First observe that (assuming general position) the structure of the Delaunay triangulation is independent of the insertion order of the sites so far. Thus, any of the existing sites is equally likely to have been the last site to be added to the structure. Suppose that some site p was the last to have been added. How much work was needed to insert p? Observe that the initial insertion of p involved the creation of three new edges, all incident to p. Also, whenever an edge swap is performed, a new edge is added to p. These are the only changes that the insertion algorithm can make. Therefore the total number of changes made in the triangulation for the insertion of p is proportional to the degree of p *after* the insertion is complete. Thus the work needed to insert p is proportional to p's degree after the insertion.

Thus, by a backwards analysis, the expected time to insert the last point is equal to the average degree of a vertex in the triangulation. (The only exception are the three initial vertices at infinity, which must be the first sites to be inserted.) However, from Euler's formula, we know that the average degree of a vertex in any planar graph is at most 6. (To see this, recall that a planar graph can have at most $3n$ edges, and the sum of vertex degrees is equal to twice the number of edges, which is at most $6n$.) Thus, (irrespective of which insertion this was) the expected number of edge changes for each insertion is $O(1)$.

Next we argue that the expected number of times that a site is rebucketed (as to which triangle it lies in) is $O(\log n)$. Again this is a standard application of backwards analysis. Consider the ith stage of the algorithm (after i sites have been inserted into the triangulation). Consider any one of the remaining $n - i$ sites. We claim that the probability that this site changes triangles is at most $3/i$ (under the assumption that any of the i points could have been the last to be added).

To see this, let q be an uninserted site and let Δ be the triangle containing q after the ith insertion. As observed above, after we insert the ith site all of the newly created triangles are incident to this site. Since Δ is incident to exactly three sites, if any of these three was added last, then Δ would have come into existence after this insertion, implying that q required (at least one) rebucketing. On the other hand, if none of these three was the last to have been added, then the last insertion could not have caused q to be rebucketed. Thus, (ignoring the three initial sites at infinity) the probability that q required rebucketing after the last insertion is exactly $3/i$. Thus, the total number of points that required rebucketings as part of the last insertion is $(n - i)3/i$. To get the total expected number of rebucketings, we sum over all stages, giving

$$\sum_{i=1}^{n} \frac{3}{i}(n-i) \;\leq\; \sum_{i=1}^{n} \frac{3}{i}n \;=\; 3n \sum_{i=1}^{n} \frac{1}{i} \;=\; 3n \ln n + O(1).$$

Thus, the total expected time spent in rebucketing is $O(n \log n)$, as desired.

There is one place in the proof that we were sloppy. (Can you spot it?) We showed that the number of points that required rebucketing is $O(n \log n)$, but notice that when a point is inserted, many rebucketing operations may be needed (one for the initial insertion and one for each additional edge flip). We will not give a careful analysis of the total number of individual rebucketing operations per point, but it is not hard to show that the expected total

number of individual rebucketing operations will not be larger by more than a constant factor. The reason is that (as argued above) each new insertion only results in a constant number of edge flips, and hence the number of individual rebucketings per insertion is also a constant. But a careful proof should consider this. Such a proof is given in our text book.

Lecture 19: Line Arrangements

Reading: Chapter 8 in the 4M's.

Arrangements: So far we have studied a few of the most important structures in computational geometry: convex hulls, Voronoi diagrams and Delaunay triangulations. Perhaps, the next most important structure is that of a *line arrangement*. As with hulls and Voronoi diagrams, it is possible to define arrangements (of $d-1$ dimensional hyperplanes) in any dimension, but we will concentrate on the plane. As with Voronoi diagrams, a line arrangement is a polygonal subdivision of the plane. Unlike most of the structures we have seen up to now, a line arrangement is not defined in terms of a set of points, but rather in terms of a set L of lines. However, line arrangements are used mostly for solving problems on point sets. The connection is that the arrangements are typically constructed in the dual plane. We will begin by defining arrangements, discussing their combinatorial properties and how to construct them, and finally discuss applications of arrangements to other problems in computational geometry.

Before discussing algorithms for computing arrangements and applications, we first provide definitions and some important theorems that will be used in the construction. A finite set L of lines in the plane subdivides the plane. The resulting subdivision is called an *arrangement*, denoted $\mathcal{A}(L)$. Arrangements can be defined for curves as well as lines, and can also be defined for $(d-1)$-dimensional hyperplanes in dimension d. But we will only consider the case of lines in the plane here. In the plane, the arrangement defines a planar graph whose vertices are the points where two or more lines intersect, edges are the intersection free segments (or rays) of the lines, and faces are (possibly unbounded) convex regions containing no line. An example is shown below.

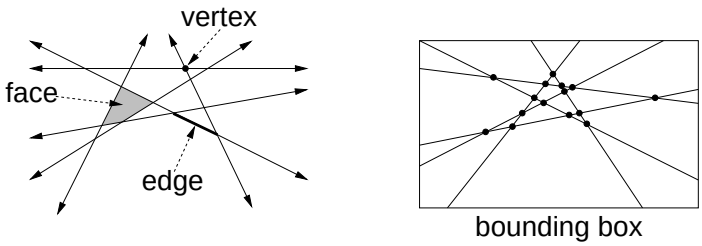

Figure 70: Arrangement of lines.

An arrangement is said to be *simple* if no three lines intersect at a common point. We will make the usual general position assumptions that no three lines intersect in a single point. This assumption is easy to overcome by some sort of symbolic perturbation.

An arrangement is not formally a planar graph, because it has unbounded edges. We can fix this (topologically) by imagining that a vertex is added at infinity, and all the unbounded edges are attached to this vertex. A somewhat more geometric way to fix this is to imagine that there is a bounding box which is large enough to contain all the vertices, and we tie all the unbounded edges off at this box. Rather than computing the coordinates of this huge box (which is possible in $O(n^2)$ time), it is possible to treat the sides of the box as existing at infinity, and handle all comparisons symbolically. For example, the lines that intersect the right side of the "box at infinity" have slopes between $+1$ and -1, and the order in which they intersect this side (from top to bottom) is in decreasing order of slope. (If you don't see this right away, think about it.)