contains q changes as a result of the ith insertion. Let P_i denote this probability (where the probability is taken over random insertion orders, irrespective of the choice of q). Since q could fall through up to three levels in the search tree as a result of each the insertion, the expected length of q's search path in the final structure is at most

$$\sum_{i=1}^{n} 3P_i.$$

We will show that $P_i \leq 4/i$. From this it will follow that the expected path length is at most

$$\sum_{i=1}^{n} 3\frac{4}{i} = 12\sum_{i=1}^{n} \frac{1}{i},$$

which is roughly $12 \ln n = O(\log n)$ by the Harmonic series.

To show that $P_i \leq 4/i$, we apply a backwards analysis. In particular, consider the trapezoid that contains q *after* the ith insertion. Recall from last time that this trapezoid is dependent on at most four segments, which define the top and bottom edges, and the left and right sides of the trapezoid. Since each segment is equally likely to be the last segment to have been added, the probability that the last insertion caused q to belong to a new trapezoid is at most $4/i$. This completes the proof.

Guarantees on Search Time: One shortcoming with this analysis is that even though the search time is provably small in the expected case for a given query point, it might still be the case that once the data structure has been constructed there is a single very long path in the search structure, and the user repeatedly performs queries along this path. Hence, the analysis provides no guarantees on the running time of all queries.

Although we will not prove it, the book presents a stronger result, namely that the length of the maximum search path is also $O(\log n)$ with high probability. In particular, they prove the following.

Lemma: Given a set of n non-crossing line segments in the plane, and a parameter $\lambda > 0$, the probability that the total depth of the randomized search structure exceeds $3\lambda \ln(n + 1)$, is at most $2/(n + 1)^{\lambda \ln 1.25 - 3}$.

For example, for $\lambda = 20$, the probability that the search path exceeds $60 \ln(n + 1)$ is at most $2/(n + 1)^{1.5}$. (The constant factors here are rather weak, but a more careful analysis leads to a better bound.)

Nonetheless, this itself is enough to lead to variant of the algorithm for which $O(\log n)$ time is guaranteed. Rather than just running the algorithm once and taking what it gives, instead keep running it and checking the structure's depth. As soon as the depth is at most $c \log n$ for some suitably chosen c, then stop here. Depending on c and n, the above lemma indicates how long you may need to expect to repeat this process until the final structure has the desired depth. For sufficiently large c, the probability of finding a tree of the desired depth will be bounded away from 0 by some constant factor, and therefore after a constant number of trials (depending on this probability) you will eventually succeed in finding a point location structure of the desired depth. A similar argument can be applied to the space bounds.

Theorem: Given a set of n non-crossing line segments in the plane, in expected $O(n \log n)$ time, it is possible to construct a point location data structure of (worst case) size $O(n)$ that can answer point location queries in (worst case) time $O(\log n)$.

Lecture 16: Voronoi Diagrams and Fortune's Algorithm

Reading: Chapter 7 in the 4M's.

Euclidean Geometry: We now will make a subtle but important shift. Up to now, virtually everything that we have done has not needed the notion of angles, lengths, or distances (except for our work on circles). All geometric tests were made on the basis of orientation tests, a purely affine construct. But there are important geometric algorithms that depend on nonaffine quantities such as distances and angles. Let us begin by defining the *Euclidean length* of a vector $v = (v_x, v_y)$ in the plane to be $|v| = \sqrt{v_x^2 + v_y^2}$. In general, in dimension d it is

$$|v| = \sqrt{v_1^2 + \ldots + v_d^2}.$$

The distance between two points p and q, denoted $\mathrm{dist}(p, q)$ or $|pq|$, is defined to be $|p - q|$.

Voronoi Diagrams: Voronoi diagrams (like convex hulls) are among the most important structures in computational geometry. A Voronoi diagram records information about what is close to what. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in the plane (or in any dimensional space), which we call *sites*. Define $\mathcal{V}(p_i)$, the *Voronoi cell* for p_i, to be the set of points q in the plane that are closer to p_i than to any other site. That is, the Voronoi cell for p_i is defined to be:

$$\mathcal{V}(p_i) = \{q \mid |p_i q| < |p_j q|, \forall j \neq i\}.$$

Another way to define $\mathcal{V}(p_i)$ is in terms of the intersection of halfplanes. Given two sites p_i and p_j, the set of points that are strictly closer to p_i than to p_j is just the *open* halfplane whose bounding line is the perpendicular bisector between p_i and p_j. Denote this halfplane $h(p_i, p_j)$. It is easy to see that a point q lies in $\mathcal{V}(p_i)$ if and only if q lies within the intersection of $h(p_i, p_j)$ for all $j \neq i$. In other words,

$$\mathcal{V}(p_i) = \cap_{j \neq i} h(p_i, p_j).$$

Since the intersection of halfplanes is a (possibly unbounded) convex polygon, it is easy to see that $\mathcal{V}(p_i)$ is a (possibly unbounded) convex polygon. Finally, define the *Voronoi diagram* of P, denoted $\mathrm{Vor}(P)$ to be what is left of the plane after we remove all the (open) Voronoi cells. It is not hard to prove (see the text) that the Voronoi diagram consists of a collection of line segments, which may be unbounded, either at one end or both. An example is shown in the figure below.

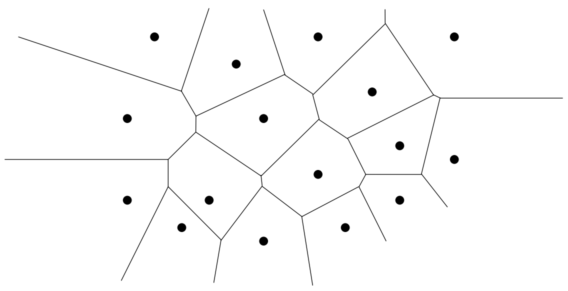

Figure 58: Voronoi diagram

Voronoi diagrams have a number of important applications. These include:

Nearest neighbor queries: One of the most important data structures problems in computational geometry is solving nearest neighbor queries. Given a point set P, and given a query point q, determine the closest point in P to q. This can be answered by first computing a Voronoi diagram and then locating the cell of the diagram that contains q. (We have already discussed point location algorithms.)

Computational morphology: Some of the most important operations in morphology (used very much in computer vision) is that of "growing" and "shrinking" (or "thinning") objects. If we grow a collection of points, by imagining a grass fire starting simultaneously from each point, then the places where the grass fires meet will be along the Voronoi diagram. The *medial axis* of a shape (used in computer vision) is just a Voronoi diagram of its boundary.

Facility location: We want to open a new Blockbuster video. It should be placed as far as possible from any existing video stores. Where should it be placed? It turns out that the vertices of the Voronoi diagram are the points that are locally at maximum distances from any other point in the set.

Neighbors and Interpolation: Given a set of measured height values over some geometric terrain. Each point has (x, y) coordinates and a height value. We would like to interpolate the height value of some query point that is not one of our measured points. To do so, we would like to interpolate its value from neighboring measured points. One way to do this, called *natural neighbor interpolation*, is based on computing the Voronoi neighbors of the query point, assuming that it has one of the original set of measured points.

Properties of the Voronoi diagram: Here are some observations about the structure of Voronoi diagrams in the plane.

Voronoi edges: Each point on an edge of the Voronoi diagram is equidistant from its two nearest neighbors p_i and p_j. Thus, there is a circle centered at such a point such that p_i and p_j lie on this circle, and no other site is interior to the circle.

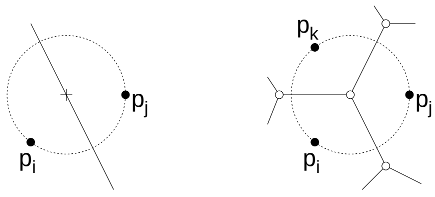

Figure 59: Properties of the Voronoi diagram.

Voronoi vertices: It follows that the vertex at which three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ intersect, called a *Voronoi vertex* is equidistant from all sites. Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior.

Degree: If we make the general position assumption that no four sites are cocircular, then the vertices of the Voronoi diagram all have degree three.

Convex hull: A cell of the Voronoi diagram is unbounded if and only if the corresponding site lies on the convex hull. (Observe that a site is on the convex hull if and only if it is the closest point from some point at infinity.) Thus, given a Voronoi diagram, it is easy to extract the convex hull in linear time.

Size: If n denotes the number of sites, then the Voronoi diagram is a planar graph (if we imagine all the unbounded edges as going to a common vertex infinity) with exactly n faces. It follows from Euler's formula that the number of Voronoi vertices is at most $2n - 5$ and the number of edges is at most $3n - 6$. (See the text for details.)

Computing Voronoi Diagrams: There are a number of algorithms for computing Voronoi diagrams. Of course, there is a naive $O(n^2 \log n)$ time algorithm, which operates by computing $\mathcal{V}(p_i)$ by intersecting the $n - 1$ bisector halfplanes $h(p_i, p_j)$, for $j \neq i$. However, there are much more efficient ways, which run in $O(n \log n)$ time. Since the convex hull can be extracted from the Voronoi diagram in $O(n)$ time, it follows that this is asymptotically optimal in the worst-case.

Historically, $O(n^2)$ algorithms for computing Voronoi diagrams were known for many years (based on incremental constructions). When computational geometry came along, a more complex, but asymptotically superior $O(n \log n)$ algorithm was discovered. This algorithm was based on divide-and-conquer. But it was rather complex, and somewhat difficult to understand. Later, Steven Fortune invented a plane sweep algorithm for the problem, which provided a simpler $O(n \log n)$ solution to the problem. It is his algorithm that we will discuss. Somewhat later still, it was discovered that the incremental algorithm is actually quite efficient, if it is run as a randomized incremental algorithm. We will discuss this algorithm later when we talk about the dual structure, called a Delaunay triangulation.

Fortune's Algorithm: Before discussing Fortune's algorithm, it is interesting to consider why this algorithm was not invented much earlier. In fact, it is quite a bit trickier than any plane sweep algorithm we have seen so far. The key to any plane sweep algorithm is the ability to discover all "upcoming" events in an efficient manner. For example, in the line segment intersection algorithm we considered all pairs of line segments that were adjacent in the sweep-line status, and inserted their intersection point in the queue of upcoming events. The problem with the Voronoi diagram is that of predicting when and where the upcoming events will occur. Imagine that you are designing a plane sweep algorithm. Behind the sweep line you have constructed the Voronoi diagram based on the points that have been encountered so far in the sweep. The difficulty is that a site that lies ahead of the sweep line may generate a Voronoi vertex that lies behind the sweep line. How could the sweep algorithm know of the existence of this vertex until it sees the site. But by the time it sees the site, it is too late. It is these *unanticipated events* that make the design of a plane sweep algorithm challenging. (See the figure below.)

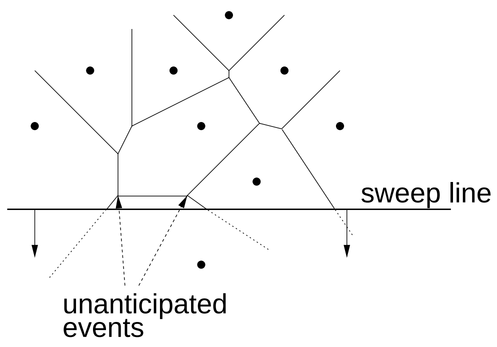

Figure 60: Plane sweep for Voronoi diagrams. Note that the position of the indicated vertices depends on sites that have not yet been encountered by the sweep line, and hence are unknown to the algorithm. (Note that the sweep line moves from top to bottom.)

Fortune made the clever observation of rather than computing the Voronoi diagram through plane sweep in its final form, instead to compute a "distorted" but topologically equivalent version of the diagram. This distorted version of the diagram was based on a transformation that alters the way that distances are measured in the plane. The resulting diagram had the same topological structure as the Voronoi diagram, but its edges were parabolic arcs, rather than straight line segments. Once this distorted diagram was generated, it was an easy matter to "undistort" it to produce the correct Voronoi diagram.

Our presentation will be different from Fortune's. Rather than distort the diagram, we can think of this algorithm as distorting the sweep line. Actually, we will think of two objects that control the sweeping process. First, there will be a horizontal sweep line, moving from top to bottom. We will also maintain an x-monotonic curve called a *beach line*. (It is so named because it looks like waves rolling up on a beach.) The beach line is a monotone curve formed from pieces of parabolic arcs. As the sweep line moves downward, the beach line follows just behind. The job of the beach line is to prevent us from seeing unanticipated events until the sweep line encounters the corresponding site.

The Beach Line: In order to make these ideas more concrete, recall that the problem with ordinary plane sweep is that sites that lie below the sweep line may affect the diagram that lies above the sweep line. To avoid this problem, we will maintain only the portion of the diagram that cannot be affected by anything that lies below the sweep line. To do this, we will subdivide the halfplane lying above the sweep line into two regions: those points that are closer to some site p above the sweep line than they are to the sweep line itself, and those points that are closer to the sweep line than any site above the sweep line.

What are the geometric properties of the boundary between these two regions? The set of points q that are equidistant from the sweep line to their nearest site above the sweep line is called the *beach line*. Observe that for any point q above the beach line, we know that its closest site cannot be affected by any site that lies below the sweep line. Hence, the portion of the Voronoi diagram that lies above the beach line is "safe" in the sense that we have all the information that we need in order to compute it (without knowing about which sites are still to appear below the sweep line).

What does the beach line look like? Recall from high school geometry that the set of points that are equidistant from a site lying above a horizontal line and the line itself forms a parabola that is open on top (see the figure below, left). With a little analytic geometry, it is easy to show that the parabola becomes "skinnier" as the site becomes closer to the line. In the degenerate case when the line contains the site the parabola degenerates into a vertical ray shooting up from the site. (You should work through the distance equations to see why this is so.)

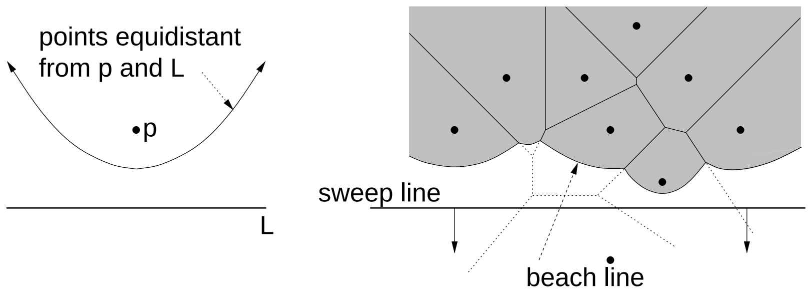

Figure 61: The beach line. Notice that only the portion of the Voronoi diagram that lies above the beach line is computed. The sweep line status maintains the intersection of the Voronoi diagram with the beach line.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site. Note that the parabola of some sites above the beach line will not touch the lower envelope and hence will not contribute to the beach line. Because the parabolas are x-monotone, so is the beach line. Also observe that the vertex where two arcs of the beach line intersect, which we call a *breakpoint*, is a point that is equidistant from two sites and the sweep line, and hence must lie on some Voronoi edge. In particular, if the beach line arcs corresponding to sites p_i and p_j share a common breakpoint on the beach line, then this breakpoint lies on the Voronoi edge between p_i and p_j. From this we have the following important characterization.

Lemma: The beach line is an x-monotone curve made up of parabolic arcs. The breakpoints of the beach line lie on Voronoi edges of the final diagram.

Fortune's algorithm consists of simulating the growth of the beach line as the sweep line moves downward, and in particular tracing the paths of the breakpoints as they travel along the edges of the Voronoi diagram. Of course, as the sweep line moves the parabolas forming the beach line change their shapes continuously. As with all plane-sweep algorithms, we will maintain a sweep-line status and we are interested in simulating the discrete event points where there is a "significant event", that is, any event that changes the topological structure of the Voronoi diagram and the beach line.

Sweep Line Status: The algorithm maintain the current location (y-coordinate) of the sweep line. It stores, in left-to-right order the set of sites that define the beach line. **Important:** The algorithm never needs to store the parabolic arcs of the beach line. It exists solely for conceptual purposes.

Events: There are two types of events.

Site events: When the sweep line passes over a new site a new arc will be inserted into the beach line.

Vertex events: (What our text calls *circle events*.) When the length of a parabolic arc shrinks to zero, the arc disappears and a new Voronoi vertex will be created at this point.

The algorithm consists of processing these two types of events. As the Voronoi vertices are being discovered by vertex events, it will be an easy matter to update a DCEL for the diagram as we go, and so to link the entire diagram together. Let us consider the two types of events that are encountered.

Site events: A site event is generated whenever the horizontal sweep line passes over a site. As we mentioned before, at the instant that the sweep line touches the point, its associated parabolic arc will degenerate to a vertical ray shooting up from the point to the current beach line. As the sweep line proceeds downwards, this ray will widen into an arc along the beach line. To process a site event we will determine the arc of the sweep line that lies directly above the new site. (Let us make the general position assumption that it does not fall immediately below a vertex of the beach line.) We then split this arc of the beach line in two by inserting a new infinitesimally small arc at this point. As the sweep proceeds, this arc will start to widen, and eventually will join up with other edges in the diagram. (See the figure below.)

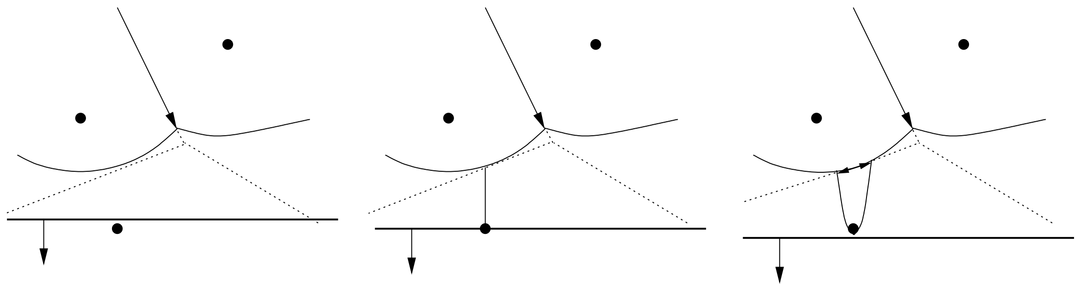

Figure 62: Site events.

It is important to consider whether this is the only way that new arcs can be introduced into the sweep line. In fact it is. We will not prove it, but a careful proof is given in the text. As a consequence of this proof, it follows that the maximum number of arcs on the beach line can be at most $2n - 1$, since each new point can result in creating one new arc, and splitting an existing arc, for a net increase of two arcs per point (except the first).

The nice thing about site events is that they are all known in advance. Thus, after sorting the points by y-coordinate, all these events are known.

Vertex events: In contrast to site events, vertex events are generated dynamically as the algorithm runs. As with the line segment plane sweep algorithm, the important idea is that each such event is generated by objects that are *neighbors* on the beach line. However, unlike the segment intersection where pairs of consecutive segments generated events, here triples of points generate the events.

In particular, consider any three consecutive sites p_i, p_j, and p_k whose arcs appear consecutively on the beach line from left to right. (See the figure below.) Further, suppose that the circumcircle for these three sites lies at least partially below the current sweep line (meaning that the Voronoi vertex has not yet been generated), and that this circumcircle contains no points lying below the sweep line (meaning that no future point will block the creation of the vertex).

Consider the moment at which the sweep line falls to a point where it is tangent to the lowest point of this circle. At this instant the circumcenter of the circle is equidistant from all three sites and from the sweep line. Thus all three parabolic arcs pass through this center point, implying that the contribution of the arc from p_j has disappeared from the beach line. In terms of the Voronoi diagram, the bisectors (p_i, p_j) and (p_j, p_k) have met each other at the Voronoi vertex, and a single bisector (p_i, p_k) remains. (See the figure below.)

Sweep-line algorithm: We can now present the algorithm in greater detail. The main structures that we will maintain are the following:

(Partial) Voronoi diagram: The partial Voronoi diagram that has been constructed so far will be stored in a DCEL. There is one technical difficulty caused by the fact that the diagram contains unbounded edges. To handle this we will assume that the entire diagram is to be stored within a large bounding box. (This box should be chosen large enough that all of the Voronoi vertices fit within the box.)

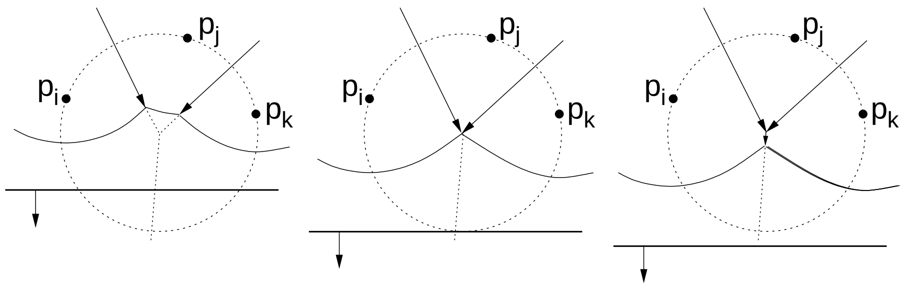

Figure 63: Vertex events.

Beach line: The beach line is represented using a dictionary (e.g. a balanced binary tree or skip list). An important fact of the construction is that *we do not explicitly store the parabolic arcs*. They are just there for the purposes of deriving the algorithm. Instead for each parabolic arc on the current beach line, we store the site that gives rise to this arc. Notice that a site may appear multiple times on the beach line (in fact linearly many times in n). But the total length of the beach line will never exceed $2n - 1$. (You should try to construct an example where a single site contributes multiple arcs to the beach line.)

Between each consecutive pair of sites p_i and p_j, there is a breakpoint. Although the breakpoint moves as a function of the sweep line, observe that it is possible to compute the exact location of the breakpoint as a function of p_i, p_j, and the current y-coordinate of the sweep line. In particular, the breakpoint is the center of a circle that passes through p_i, p_j and is tangent to the sweep line. Thus, as with beach lines, *we do not explicitly store breakpoints*. Rather, we compute them only when we need them.

The important operations that we will have to support on the beach line are

(1) Given a fixed location of the sweep line, determine the arc of the beach line that intersects a given vertical line. This can be done by a binary search on the breakpoints, which are computed "on the fly". (Think about this.)

(2) Compute predecessors and successors on the beach line.

(3) Insert an new arc p_i within a given arc p_j, thus splitting the arc for p_j into two. This creates three arcs, p_j, p_i, and p_j.

(4) Delete an arc from the beach line.

It is not difficult to modify a standard dictionary data structure to perform these operations in $O(\log n)$ time each.

Event queue: The event queue is a priority queue with the ability both to insert and delete new events. Also the event with the largest y-coordinate can be extracted. For each site we store its y-coordinate in the queue.

For each consecutive triple p_i, p_j, p_k on the beach line, we compute the circumcircle of these points. (We'll leave the messy algebraic details as an exercise, but this can be done in $O(1)$ time.) If the lower endpoint of the circle (the minimum y-coordinate on the circle) lies below the sweep line, then we create a vertex event whose y-coordinate is the y-coordinate of the bottom endpoint of the circumcircle. We store this in the priority queue. Each such event in the priority queue has a cross link back to the triple of sites that generated it, and each consecutive triple of sites has a cross link to the event that it generated in the priority queue.

The algorithm proceeds like any plane sweep algorithm. We extract an event, process it, and go on to the next event. Each event may result in a modification of the Voronoi diagram and the beach line, and may result in the creation or deletion of existing events.

Here is how the two types of events are handled:

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.