CMSC 754: Lecture 11
Voronoi Diagrams and Fortune's Algorithm

Reading: Chapter 7 in the 4M's. A nice applet illustrating the execution of this algorithm can be found at `http://www.raymondhill.net/voronoi/rhill-voronoi.html`.

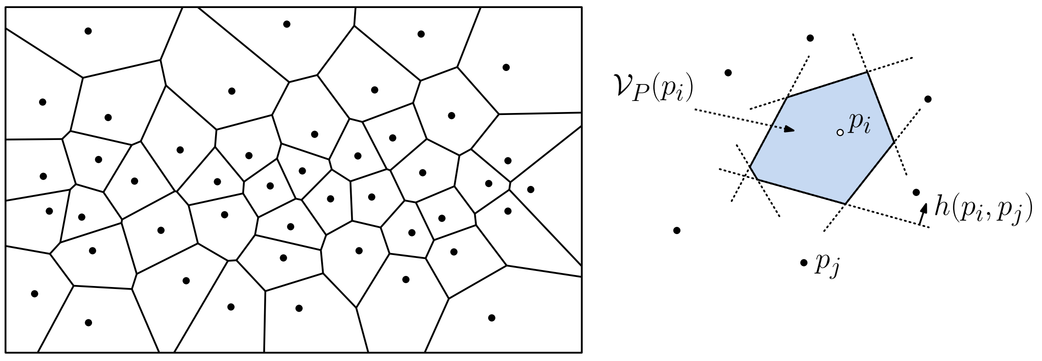
Voronoi Diagrams: Voronoi diagrams are among the most important structures in computational geometry. Throughout, let

$$\|p-q\| \;=\; \left(\sum_{i=1}^{d}(p_i-q_i)^2\right)^{1/2}$$

denote the standard *Euclidean distance* between two points $p, q \in \mathbb{R}^d$. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$, which we call *sites*. Define $\mathcal{V}_P(p_i)$, called the *Voronoi cell*, for p_i, to be the set of points q in space that are closer to p_i than to any other site, that is,

$$\mathcal{V}_P(p_i) \;=\; \{q \in \mathbb{R}^d : \|p_i-q\| < \|p_j-q\|, \forall j \neq i\},$$

When P is clear from context, we will omit it and refer to this simply as $\mathcal{V}(p_i)$. Clearly, the Voronoi cells of two distinct points of P are disjoint. The union of the closure of the Voronoi cells defines a cell complex, which is called the *Voronoi diagram* of P, and is denoted $\mathrm{Vor}(P)$ (see Fig. 1(a)).



(a) (b)

Fig. 1: Voronoi diagram $\mathrm{Vor}(P)$ of a set of points.

The cells of the Voronoi diagram are (possibly unbounded) convex polyhedra. To see this, observe that the set of points that are strictly closer to one site p_i than to another site p_j is equal to the *open halfspace* whose bounding hyperplane is the perpendicular bisector between p_i and p_j. Denote this halfspace $h(p_i, p_j)$. 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,

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

(see Fig. 1(b)). Since the intersection of convex objects is convex, $\mathcal{V}(p_i)$ is a (possibly unbounded) convex polyhedron.

Voronoi diagrams have a huge number of important applications in science and engineering. These include answering nearest neighbor queries, computational morphology and shape analysis, clustering and data mining, facility location, multi-dimensional interpolation.

Nearest neighbor queries: Given a point set P, we wish to preprocess P so that, given a query point q, it is possible to quickly determine the closest point of P to q. This can be answered by first computing a Voronoi diagram and then locating the cell of the diagram that contains q. (In the plane, this can be done by building the trapezoidal map of the edges of the Voronoi diagram. Each trapezoid lies within a single Voronoi cell, and can be labeled with the generating point.)

Computational morphology and shape analysis: A useful structure in shape analysis is called the medial axis. The *medial axis* of a shape (e.g., a simple polygon) is defined to be the union of the center points of all locally maximal disks that are contained within the shape (see Fig. 2). If we generalize the notion of Voronoi diagram to allow sites that are both points and line segments, then the medial axis of a simple polygon can be extracted easily from the Voronoi diagram of these generalized sites.

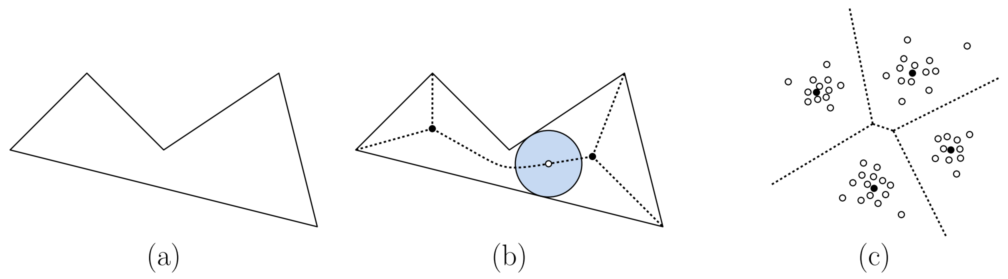


Fig. 2: (a) A simple polygon, (b) its medial axis and a sample maximal disk, and (c) center-based clustering (with cluster centers shown as black points).

Center-based Clustering: Given a set P of points, it is often desirable to represent the union of a significantly smaller set of clusters. In center-based clustering, the clusters are defined by a set C of *cluster centers* (which may or may not be required to be chosen from P). The *cluster* associated with a given center point $q \in C$ is just the subset of points of P that are closer to q than any other center, that is, the subset of P that lies within q's Voronoi cell (see Fig. 2(c)). (How the center points are selected is another question.)

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 properties of the Voronoi diagrams in the plane. These all have natural generalizations to higher dimensions.

Empty circle properties: 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 any such point where p_i and p_j lie on this circle, and no other site is interior to the circle (see Fig. 3(a)).

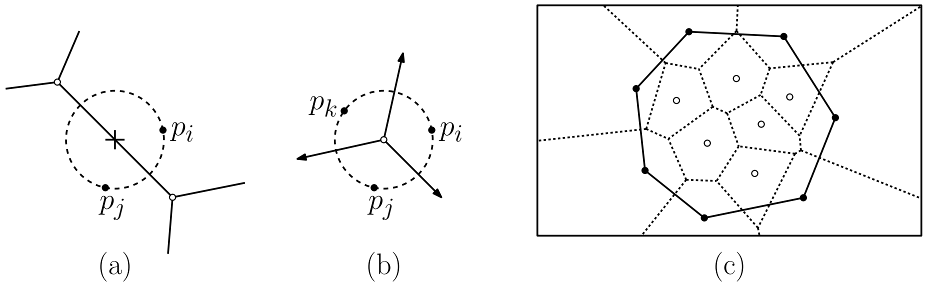


Fig. 3: Properties of the Voronoi diagram.

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 (see Fig. 3(b)). Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior. (In $\mathbb{R}^d$, the vertex is defined by $d+1$ points and the hypersphere centered at the vertex passing through these points is empty.)

Degree: Generally three points in the plane define a unique circle (generally, $d+1$ points in $\mathbb{R}^d$). If we make the general position assumption that no four sites are cocircular, then each vertex of the Voronoi diagram is incident to three edges (generally, $d+1$ facets).

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, namely the point infinitely far along a vector orthogonal to the supporting line through this vertex.) Thus, given a Voronoi diagram, it is easy to extract the vertices of the convex hull in linear time.

Size: Letting n denote the number of sites, the Voronoi diagram with exactly n faces. It follows from *Euler's formula*[1] that the number of Voronoi vertices is roughly $2n$ and the number of edges is roughly $3n$. (See the text for details. In higher dimensions the diagram's combinatorial complexity ranges from $O(n)$ up to $O(n^{\lceil d/2 \rceil})$.)

Computing Voronoi Diagrams: There are a number of algorithms for computing the Voronoi diagram of a set of n sites in the plane. 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

[1]Euler's formula for planar graphs states that a planar graph with v vertices, e edges, and f faces satisfies $v - e + f = 2$. There are n faces, and since each vertex is of degree three, we have $3v = 2e$, from which we infer that $v - (3/2)v + n = 2$, implying that $v = 2n - 4$. A similar argument can be used to bound the number of edges.

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 discovered 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 a variant of this algorithm later when we talk about the dual structure, called the 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.

To see the problem, suppose 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 Fig. 4).

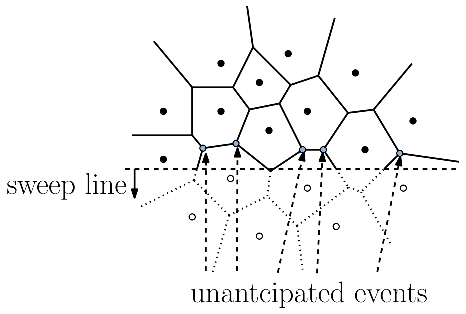


Fig. 4: Plane sweep for Voronoi diagrams. Note that the position of the indicated vertices depends on sites that have not yet been encountered by the sweep line, and hence are unknown to the algorithm. (Note that the sweep line moves from top to bottom.)

The Beach Line: The sweeping process will involve sweeping two different object. 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 lags behind the sweep line in such a way that it is unaffected by sites that have yet to be seen. Thus, there are no unanticipated events on the beach line. The sweep-line status will be based on the manner in which the Voronoi edges intersect the beach line, not the actual sweep line.

Let's make these ideas more concrete. We 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 your high-school geometry that the set of points that are equidistant from a point (in this case a site) and a line (in this case the sweep line) is a *parabola* (see Fig. 5(a)). The parabola's shape depends on the distance between p and the line ℓ. As the line moves further away, the parabola becomes "fatter" (see Fig. 5(b)). (In the extreme case when the line contains the site the parabola degenerates into a vertical ray shooting up from the site.)

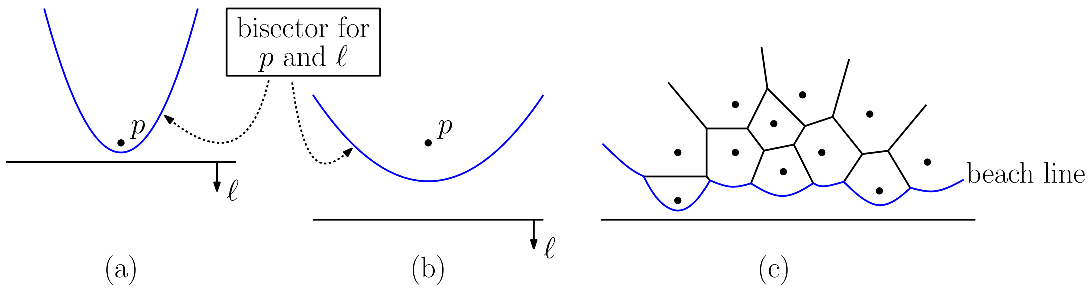


Fig. 5: The beach line. Notice that only the portion of the Voronoi diagram that lies above the beach line is computed. The sweep-line status maintains the intersection of the Voronoi diagram with the beach line.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site (see Fig. 5(c)). Note that the parabola associated with some sites may be *redundant* in the sense that they will not contribute to the beach line. Because the parabolas are x-monotone, so is the beach line. Also observe that the point where two arcs of the beach line intersect, which we call a *breakpoint*, 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 (that is, vertices) 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 or the beach line.

Sweep-Line Status: The algorithm maintains the current location (y-coordinate) of the sweep line. It stores, in left-to-right order the sequence of sites that define the beach line. (We will say more about this later.) **Important:** The algorithm does *not* store the parabolic arcs of the beach line. They are shown solely for conceptual purposes.

Events: There are two types of events:

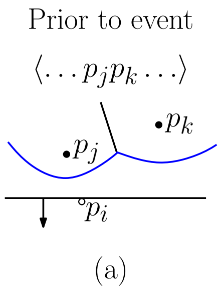
- **Site events:** When the sweep line passes over a new site a new parabolic arc will be inserted into the beach line.
- **Voronoi vertex events:** (What our text calls *circle events.*) When the length of an arc of the beach line shrinks to zero, the arc disappears and a new Voronoi vertex will be created at this point.

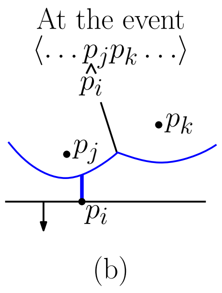
The algorithm consists of processing these two types of events. As the Voronoi vertices are being discovered by Voronoi vertex events, it will be an easy matter to update the diagram as we go (assuming any reasonable representation of this planar cell complex), and so to link the entire diagram together. 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 p_i. 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 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.) Let p_j denote the site generating this arc. We then split this arc in two by inserting a new entry at this point in the sweep-line status. (Initially this corresponds to a infinitesimally small arc along the beach line, but as the sweep line sweeps on, this arc will grow wider. Thus, the entry for $\langle \ldots, p_j, \ldots \rangle$ on the sweep-line status is replaced by the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$ (see Fig. 6).

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, 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). Note that a point may generally contribute more than one arc to the beach line. (As an exercise you might consider what is the maximum number of arcs a single site can contribute.)

The nice thing about site events is that they are all known in advance. Thus, the sites can be presorted by the y-coordinates and inserted as a batch into the event priority queue.




At the event
$\langle \ldots p_j \overset{\wedge}{p_i} p_k \ldots \rangle$
p_j
p_k
p_i
(b)

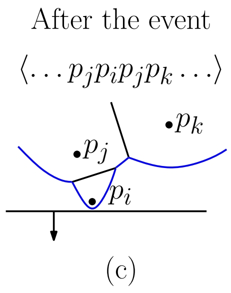
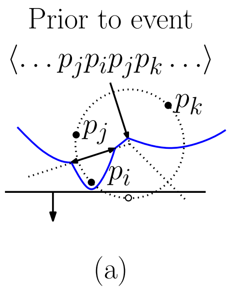


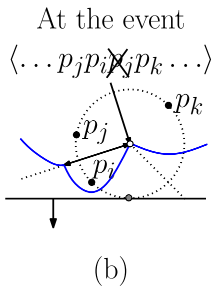
Fig. 6: Site event.

Voronoi vertex events: In contrast to site events, Voronoi vertex events are generated dynamically as the algorithm runs. As with the line segment intersection algorithm, the important idea is that each such event is generated by objects that are *adjacent* on the beach line (and thus, can be found efficiently). 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 Fig. 7(a). 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. Thus, the triple of consecutive sites p_i, p_j, p_k on the sweep-line status is replaced with p_i, p_k (see Fig. 7).





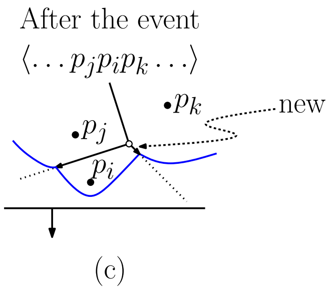


Fig. 7: Voronoi vertex event.

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 any reasonable data structure for storing planar subdivisions, for example, a doubly-connected edge list. There is one technical difficulty caused by the fact that the diagram contains unbounded edges. This can be handled by enclosing everything within a sufficiently large bounding box. (It should be large enough to contain all the Voronoi vertices, but this is not that easy to compute in advance.) An alternative is to create an imaginary Voronoi vertex "at infinity" and connect all the unbounded edges to this imaginary vertex.

Beach line: The beach line consists of the sorted sequence of sites whose arcs form the beach line. It is represented using a dictionary (e.g. a balanced binary tree or skip list). As mentioned above, 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.

The key search operation is that of locating the arc of the beach line that lies directly above a newly discovered site. (As an exercise, before reading the next paragraph you might think about how you would design a binary search to locate this arc, given that you only have the sites, not the actual arcs.)

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.) Once the breakpoint is computed, we can then determine whether a newly added site is to its left or right. Using the sorted ordering of the sites, we use this primitive comparison to drive a binary search for the arc lying above the new site.

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

Search: Given the current y-coordinate of the sweep line and a new site p_i, determine the arc of the beach line lies immediately above p_i. Let p_j denote the site that contributes this arc. Return a reference to this beach line entry.

Insert and split: Insert a new entry for p_i within a given arc p_j of the beach line (thus effectively replacing the single arc $\langle \ldots, p_j, \ldots \rangle$ with the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$. Return a reference to the newly added beach line entry (for future use).

Delete: Given a reference to an entry p_j on the beach line, delete this entry. This replaces a triple $\langle \ldots, p_i, p_j, p_k, \ldots \rangle$ with the pair $\langle \ldots, p_i, p_k, \ldots \rangle$.

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. All operations can be implemented in $O(\log n)$ time assuming that the priority queue is stored as an ordered dictionary.

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 Voronoi 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. The algorithm starts by inserting the topmost vertex into the sweep-line status. 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 in somewhat greater detail.

Site event: Let p_i be the new site (see Fig. 6 above).

(1) Advance the sweep line so that it passes through p_i. Apply the above search operation to determine the beach line arc that lies immediately above p_i. Let p_j be the corresponding site.

(2) Applying the above insert-and-split operation, inserting a new entry for p_i, thus replacing $\langle \ldots, p_j, \ldots \rangle$ with $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$.

(3) Create a new (dangling) edge in the Voronoi diagram, which lies on the bisector between p_i and p_j.

(4) Some old triples that involved p_j may need to be deleted and some new triples involving p_i will be inserted, based on the change of neighbors on the beach line. (The straightforward details are omitted.)
Note that the newly created beach-line triple p_j, p_i, p_j does not generate an event because it only involves two distinct sites.

Voronoi vertex event: Let p_i, p_j, and p_k be the three sites that generated this event, from left to right (see Fig. 7 above).

(1) Delete the entry for p_j from the beach line status. (Thus eliminating its associated arc.)

(2) Create a new vertex in the Voronoi diagram (at the circumcenter of $\{p_i, p_j, p_k\}$) and join the two Voronoi edges for the bisectors (p_i, p_j), (p_j, p_k) to this vertex.

(3) Create a new (dangling) edge for the bisector between p_i and p_k.

(4) Delete any events that arose from triples involving the arc of p_j, and generate new events corresponding to consecutive triples involving p_i and p_k. (There are two of them. The straightforward details are omitted.)

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