**Fortune Sweep Algorithm**

Now we take a look at a better way to calculate the 2D Voronoi diagram. This algorithm was first introduced by Steven Fortune in [[2]](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/References.htm#fortuneRef) and is often referred to as *Fortune’s Algorithm* as a result. This algorithm has complexity **O(n log n)**, which is optimal for this problem. The following sections introduce the different constructions and terminology that are used by the solution, followed by the algorithm itself.

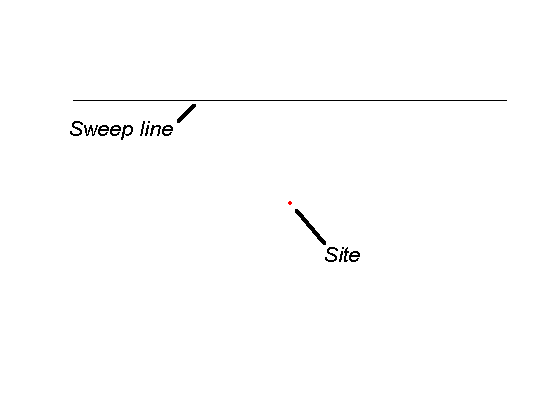
Sections:

**Sweep Line**

A common way to simplify a proximity related problem such as computing the Voronoi diagram is to use a sweep line construction. A sweep line splits the problem domain into two regions, an explored region and an unexplored region. It applies an ordering to the problem because we reason about the explored area based on what we’ve seen so far and ignore the unexplored area. We can get away with not caring about the unexplored area by observing that only points that have reached the sweep line or crossed it can possibly affect the current computation since all other points are too far away. The entire problem area is eventually examined by “sweeping" the line across the set of points from one extreme to another.

Let's look at an example of the sweep line applied to calculating the Voronoi diagram of a single site. Imagine our sweep line, denoted **s**, starts at the top of the image and sweeps down as seen in [Figure 2](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig2). The point set contains only one site **p1**. How can we reason about the area above the sweep line? Before the first and only site is encountered, there is nothing to reason about. Once the sweep line has passed below site **p1**, what area must be in the Voronoi cell of site **p1**? Since there could be a new site lurking just beneath the surface of the sweep line we can only reason that any point closer to **p1** than to the sweep line itself must be in the Voronoi cell of site **p1** at this moment in time. It could be that the Voronoi cell of site **p1** is much larger than what we’re assigning to it at this moment, but the sweep will eventually account for this.

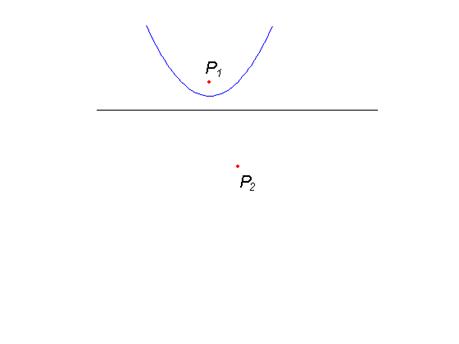
Reasoning in this way divides the area above the sweep line into two disjoint regions separated by a parabola. Points on the parabola are equidistant to **p1** and the sweep line. This dividing line is termed the *beach line* (for reasons that will soon become apparent). The algorithm for computing the Voronoi diagram of a set of points depends entirely on how this beach line changes as the sweep moves through the space. The beach line’s topology changes when a new arc is added or deleted. The following sections look at when these events can occur.



**Figure 2**: Sweep line passing over a single site. We only reason about the area above the sweep line at any point in time. The parabola is degenerate when it first hits the sweep line, creating a vertical line of zero area.

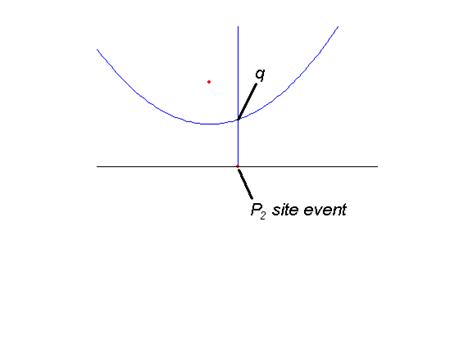
**Site Events**

 So what happens when there is more than one site during the sweep? In [Figure 3](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig3) we can see this situation.



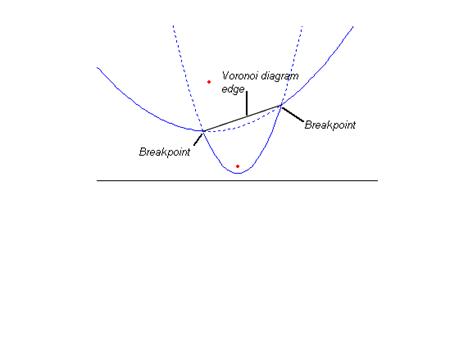
**Figure 3:** Two sites present

 When a site hits the sweep line it is called a *site event*. In [Figure 4](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig4) we can see the site event where **p2** hits the sweep line. Note that it forms a degenerate parabola that intersects **p1**’s parabola at a single point. Let’s call this point of intersection **q**. We know that **q** is equidistant to both **p1** and **p2**, since it is on **p1**’s beach line which is the set of points of equal distance to **p1** and the sweep line. Furthermore, we know that no other site can be closer to **q** otherwise the beach line would not exist at that point (from the way we defined the beach line). Therefore we know that this point must lie on the Voronoi edge that separates the two cells defined by sites **p1** and **p2**.



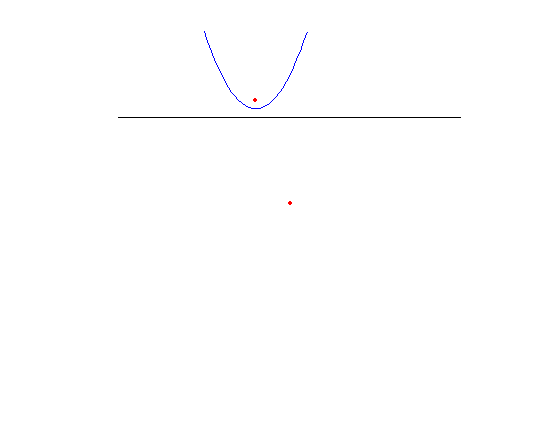
**Figure 4:** **p2**’s site event. Point **q** must be equal distance to both **p2** and **p1**.

As the sweep line continues downwards **p2** begins to define its own parabola that intersects **p1**’s parabola twice as seen in [Figure 5](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig5). These intersections are termed *breakpoints*. If we drew a circle that was centred on each breakpoint we would see that **p1**, **p2**, and the closest point on the sweep line all lie on the circumference of the circle. Since this circle must be empty (if it wasn’t then the breakpoint would not exist at that point) we can see that the breakpoints are defining the edges of the Voronoi diagram as the sweep line moves downwards.



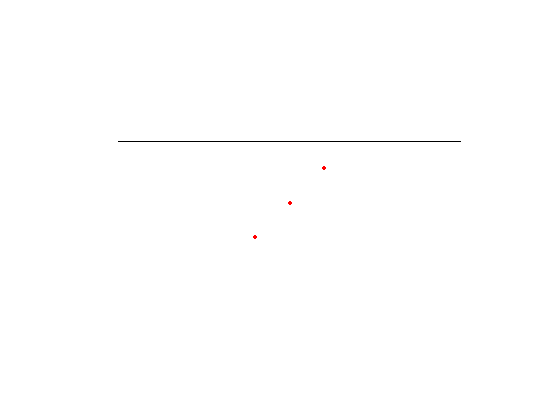
**Figure 5:** The breakpoints are defined by the intersection of the different arcs on the beach line. They also trace out the Voronoi diagram edges.

 The beach line now consists of three arcs - **p2**’s parabola and **p1**’s parabola that is divided into two pieces by **p2**. The beach line is always defined by the parabola that is closest to the sweep line. The appearance of the beach line at this point is where it gets its name – it looks like the arcs created by the meeting of the beach and the ocean. [Figure 6](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig6) shows the progression of the sweep line for two sites.



**Figure 6:** The sweep line progression for two sites

 This process can be extended to more than just two sites, by picturing each new site event creating a new parabolic arc and defining a new Voronoi edge as in [Figure 7](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig7). There is one last issue that remains to be looked at, and that is at points where arcs on the beach line disappear. This is covered in the next section.



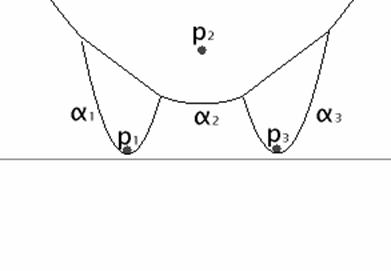
**Figure 7:** The sweep line progression for three sites

**Important:** The most important thing to realize about site events is that a site event is the only way a new arc can appear on the beach line [[1]](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/References.htm#bergRef). When we look at the algorithm that uses the sweep line to calculate the Voronoi diagram this fact is crucial.

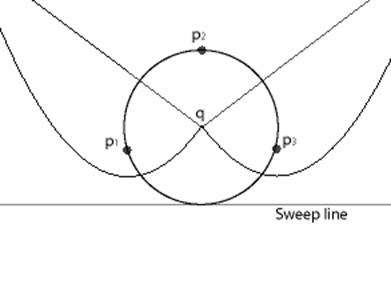
**Circle Events**

You can imagine that as new arcs appear along the beach line created by site events, old arcs that are far away from the sweep line must at some point disappear. When would this happen?

 In [Figure 8](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig8) you can see a situation where this would occur. The beach line originally consists of three arcs, α1, α2 and α3 as generated by three sites **p1**, **p2**, and **p3** respectively. As the sweep line moves downwards α2 is engulfed by its two neighbours. This event is termed a *circle event* and can be seen in [Figure 9](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig9). Let point **q** be the intersection of these three parabolas at that moment. A circle event gets its name because at the moment it occurs you could draw a circle passing through **p1**, **p2**, and **p3** and the closest point on the sweep line and the origin would be centred on point **q**. This circle exists because we know that each site is the same distance away from **q** which is equal to the distance to the sweep line (by the way we constructed the beach line in the first place!). We also know that there can’t be another site inside of this circle since then **q** would be closer to that new site than it is to the sweep line, contradicting the fact that **q** is on the beach line.

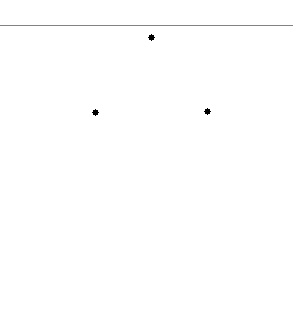


**Figure 8:** Three sites **p1**, **p2**, and **p3** creating three arcs α1, α2 and α3 respectively



**Figure 9:** Circle event where α2 is engulfed by its two neighbours

 You’ve probably already guessed that this will be a vertex in the Voronoi diagram under construction. You could imagine the two converging breakpoints that are defining Voronoi edges meeting at point q, and only one breakpoint leaving – defining a Voronoi vertex of degree 3. See [Figure 10](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig10) for an animation showing this situation. For Voronoi vertices of degree greater than 3 the same progression would happen, except there would be 2 or more arcs disappearing at the exact same point.



**Figure 10**: Creation of a Voronoi vertex of degree 3

**Important:** It turns out that the only way for an arc to disappear from the beach line is through a circle event [[1]](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/References.htm#bergRef).

**Sweep Algorithm**

We are now ready to look at the sweep algorithm for computing the 2D Voronoi diagram of a set of points. The algorithm for this problem does not actually calculate the parabolas that define the beach line as we sweep through the problem space. Doing so would be computationally expensive and as it turns out, unnecessary. Instead, only instances when the beach line changes topology do we need to do calculations. We’ve already seen the two times the beach line can change topology: site events where a new site crosses the sweep line and circle events where an arc on the beach line disappears.

**Data Structures:**

 The sweep line algorithm requires three data structures to keep track of things:

         **A** **priority queue** to keep track of the site events and circle events

        A priority queue is used to tell us when the topology of the beach line could change. The priority of an event is determined by when the sweep line will encounter it. For example, if the sweep line starts at the top of the problem space and moves downwards, the site events would be given higher priority if they have a larger y-coordinate (assuming y increases in the upward direction). Circle events are also given a priority by when the sweep line will encounter them. In our example this would be the y-coordinate of the sweep line as it just grazes the bottom of the circle corresponding to the circle event.

        Site events will store their site. Circle events will store the position they will occur, as well as a pointer to the arc (leaf) in the binary tree that will disappear when this circle event occurs.

        The algorithm progresses by simply popping the highest priority event from the queue and calculating the manner in which the topology of the beach line changes.

        The site events are known beforehand and can be entered into the priority queue during initialization. However, the circle events are detected as the beach line changes its shape, and need to be entered into the queue at that time. More on this process can be found below.

        **An edge vertex list** to keep track of the Voronoi diagram under construction

        When a site event defines a new parabolic arc on the beach line it begins to trace out a new Voronoi edge. Also, when a circle event occurs and two or more edges meet they define a Voronoi vertex at that point, and a new Voronoi edge will leave that vertex defined by the new breakpoint. To maintain these edges and vertices as they are being created an edge vertex list is used.

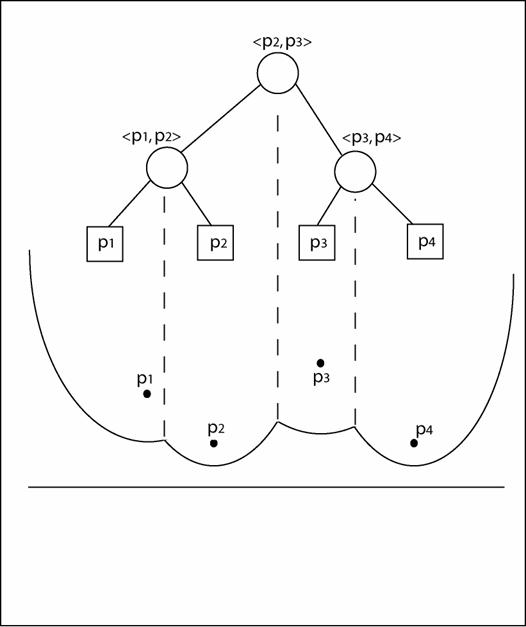
        At the end of the algorithm this list will contain the Voronoi diagram we are trying to calculate.

        **A** **balanced binary search tree** to maintain the topology of the beach line

        A balanced binary search tree is used to maintain the status of the beach line. See [Figure 11](http://cgm.cs.mcgill.ca/%7Emcleish/644/Projects/DerekJohns/Sweep.htm#Fig11) for an example of what the tree would look like in a sample problem. Each leaf of the tree stores a site that defines an arc on the beach line. Each internal node stores the two sites that define the breakpoints on the beach line. Using this representation it becomes easy to determine where a new site event will affect the beach line. In our example, we would simply compare the x-coordinate of the new site to each internal node to find the correct location along the beach line for this new arc. This can be done in O(log*n*) time since we are using a balanced binary search tree.

        When a new arc appears on the beach line it will divide an existing arc into two segments. To make the necessary changes to our tree we delete the leaf corresponding to the original arc that will be divided into two, and replace it with a subtree with three leaves. The middle leaf will store the new site being added, and the other two leaves will contain the original site. The internal nodes need to be updated to reflect the change in breakpoints. The tree can be rebalanced if necessary.

        Finally, each leaf also stores a pointer to a circle event in the priority queue where the arc defined by this site will disappear. If this event does not exist or hasn’t been determined yet this pointer is set to **null**. A circle event is calculated by comparing the breakpoints on either side of the arc. If there are one or no breakpoints, no circle event will occur. If there are two breakpoints, then a test needs to be made to see if these breakpoints will converge. If they will converge, this means that this is a potential circle event and needs to be entered into the priority queue.



**Figure 11**: Binary search tree and its correspondence to the beach line. Internal nodes store the breakpoint information which is the two sites that are creating the breakpoint. External nodes store the sites on the beach line, plus pointers to the circle events in which they will disappear.

 We are now ready to look at the algorithm for this process.

**Algorithm**

**Main**(list of sites)

1. Initialized data structures. Insert site events into the priority queue based on their y-coordinate value.
2. While(Queue not empty)
   1. event = Pop first event from queue
   2. If(event = = Site Event)

**ProcessSite**(event)

* 1. Else

**ProcessCircle**(event)

1. Finish all edges in binary search tree

**ProcessSite**(event)

1. If (binaryTree = = null)
   1. Set event as root
2. Else
   1. Find correct position in binaryTree for this site event. This is done by starting at root and checking breakpoints of internal nodes as you reach them. Go to right or left child in the tree based on whether the current site event’s x-coordinate is greater than or less than the breakpoint (if it is equal to a breakpoint either direction can be chosen). Stop when a leaf node is encountered.
   2. If the arc that this site hits contains a pointer to a circle event in the queue, delete that circle event from the queue (it is a false alarm and will never happen). Also delete circle events involving this arc from the queue. These events can be found by looking at the neighbours of this arc in the binary tree (note that the neighbours will not always exist, and they can be other leaves than just the sibling of this leaf)
   3. Remove the leaf that this site event hits and replace it with a subtree with three leaf nodes. The left and right leaves will contain the original site, and the middle leaf will contain this new site. The two new internal nodes of the tree will represent the two breakpoints that have been created along with the new edge that needs to be created. Create a new edge in the edge vertex list and point each internal node to one side of this new edge.
   4. Check for circle events caused by this new site. Note that we don’t have to check the triple of leaves where the new site is the middle leaf, because the breakpoints can’t converge. Instead, check the triples where this new site is the far left and far right arc. If the breakpoints are converging, calculate the circle event priority and place it in the queue. Make a pointer from the middle leaf (the leaf that will disappear in the circle event) to the event in the queue.

**ProcessCircle**(event)

1. Update breakpoints involving the arc that is disappearing in this event. The edges that these breakpoints point to will be finishing.
2. Check for circle events involving this arc in the immediate left and right arcs of the beach line. If circle events exist at these nodes, delete them.
3. Remove the corresponding arc leaf from the binaryTree. Delete internal parent node of this arc leaf, and promote sibling leaf or subtree to the parent’s position. Update breakpoints in binaryTree to reflect the new breakpoint that has been created. Note that there will be two breakpoints that are disappearing in this event, and one new breakpoint that is being created. This newly created breakpoint needs to point to one side of a new edge created in the edge vertex list. The other side of the edge should be set to the vertex created by this event.
4. Check new triples of arcs created by this rearranging of the binaryTree for circle events. If a circle event is detected put it in the priority queue, and put pointers in the leaf nodes that will disappear in that event.

**Subtle Algorithm Issues**

1. The algorithm starts by initializing the data structures above and populating the priority queue with the site events corresponding to our ***n*** data points. It then proceeds by popping the first event from the priority queue. The tree root is created by the first site event from our queue. If there are two sites with the same priority (same y-value in our example), it means that the sweep line will encounter them at the same time. It turns out that this tie can be handled in any order by the algorithm without any problem except when the tie happens when constructing the tree root. In this case, special code is needed to break the tie and insert one site into the tree first.
2. We need to check the breakpoints of each new triple of leaves created in our binary tree for a potential circle event. Remember that a circle event will occur if the breakpoints are converging. If a circle event is detected we determine its priority and put it in the priority queue. We also store a pointer to that event in the binary tree leaf of the site that will disappear due to this circle event. Just because we have detected a possible circle event does not mean that it will necessarily happen. For instance, an arc triple with converging breakpoints could be separated by the arrival of a new site event that intersects the beach line in-between the two converging breakpoints. In this case the previous circle event that was put in the priority queue will never happen. This detection of a circle event that will not happen is termed a *false alarm*. When this happens the circle event that won’t happen needs to be deleted from the priority queue, which can be done easily with the pointer contained in our binary tree leaf.

**Complexity**

*This algorithm has an* ***O(n log n)*** *running time and uses* ***O(n)*** *amount of storage. These values are derived below.*

Since we are using a balanced binary search tree any primitive operations such as finding the location of a site along the beach line or adding and deleting nodes will take **O(log n)** time. Making changes to the priority queue will also take **O(log n)** time. Any additions to the edge vertex list take a constant amount of time. These three operations can happen for each event a constant number of times, which means that we spend **O(log n)** time per event.

So how many events do we have? Obviously we have at least **n** events since we will have a site event for each point in our data set. How many circle events will we have? Each circle event corresponds to a vertex in the Voronoi diagram. The false circle events are deleted from the priority queue before they are processed, and this deletion is performed while processing other real events in constant time. Therefore false alarms do not contribute to the number of circle events processed. So we can determine the number of circle events by knowing the upper bound on the number of vertices contained in our Voronoi diagram. Using Euler’s formula for a connected planar embedded graph we can calculate this upper bound as ***2n-5***. So this gives us a total of at most ***3n-5*** events, each spending **O(log n)** time doing operations. This gives us an overall running time of **O(n log n)**. The storage for the binary tree, priority queue, and edge vertex list is also a linear function of the number of points being examined. Hence we have **O(n)** storage.