Fortune's Voronoi diagram algorithm extended to convex sites *†

Andrew Farrell

28 October 1994

Declaration

This M.Sc. thesis has been submitted in accordance with the requirements of the University of Dublin. It has not been submitted previously as an exercise for a degree at any other university.

It is entirely my own work, except where due acknowledgement or reference is given

I agree that the Library may lend or copy this thesis upon request.

Andrew Farrell

Signed:

Acknowledgements

I would like to thank the my supervisor Dr. Colm Ó Dúnlaing for helping me throughout the year. His help in fleshing out the final document was invaluable, and he is responsible for helping me to couch some of my proofs in more rigorous language, and was also responsible for the implementation of the Apollonius Problem that we use, and that produced Figures 28 and 29. I would also like to thank Keith Brady, a fellow student, for his invaluable help. He created a postprocessor compatible with the implementation here and responsible for Figures 1, 2, 5 - 13 and 30 - 36. A remark on his work on Voronoi diagrams for line segments was responsible for the formulation of Lemma 4.

*This work was supported in part by the the EEC under the Esprit Basic Research Action 7141 (ALCOM II)

†This work also supported by the European Social Fund

Contents

1 Abstract

This dissertation presents a version of Fortune's sweep-line algorithm based on an observation of Richard Cole's [2]. It describes a generic version of the algorithm to construct the diagram for an arbitrary collection of convex sites, suitably preprocessed. Mathematical arguments are given for the algorithm's correctness. For the case of circular sites, a working implementation is described in detail.

2 Introduction

2.1 Problem Definition

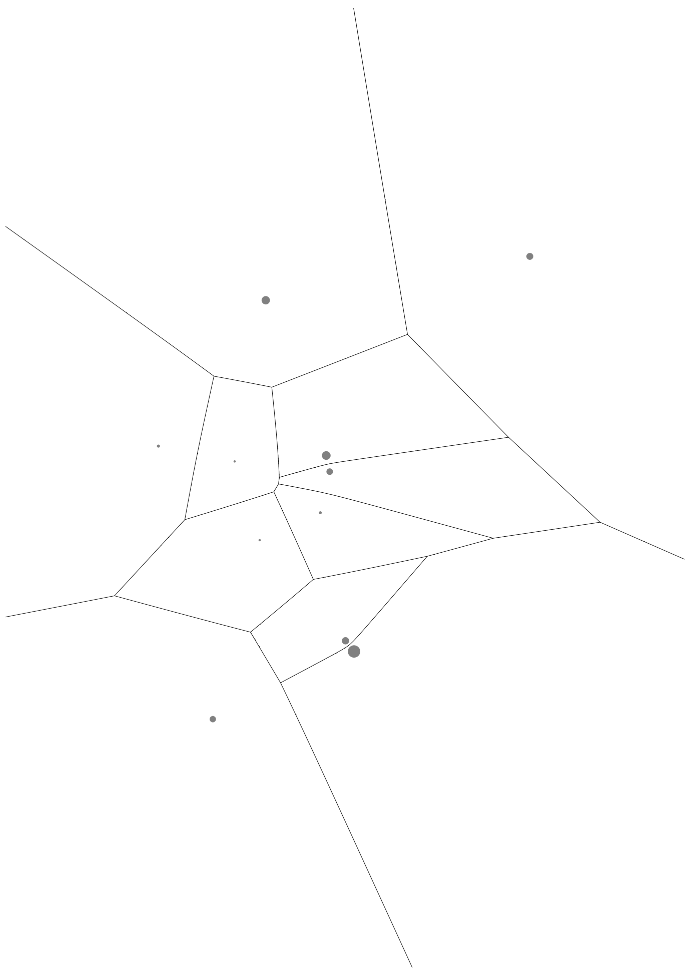
Let S indicate a collection of closed and bounded convex sets (or 'sites') in $\mathcal{R}^2$, under the Euclidean distance function. For any two sets S_1, S_2, the dominance of S_1 over S_2, $Dom(S_1, S_2)$ is the set of all points closer to S_1 than to S_2, where the distance of a point to a convex set is defined as the distance from the point to the nearest point of the set, i.e. the minimum possible distance. It is worth pointing out that under the Euclidean distance function, the nearest point in a convex set to any given point is unique, since if there were two nearest distance points, the point half-way between them would be in the set, and nearer to the given point. The Voronoi region associated with the site S is the intersection of its dominances over all the other sites. The boundary of each region is a collection of separators $Sep(S, S_{n_1})$, $Sep(S, S_{n_2})$ ... $Sep(S, S_{n_m})$, where $Sep(S_1, S_2)$ is the set of all points whose distances from S_1 and S_2 are equal, and less than their distance from any other site. A point where separators join is known as a Voronoi vertex, an alternative definition of which is as the centre of a circle which is tangent to three (or more) sites, but intersects the interior of none. See Figures 1 and 2.

The Voronoi Diagram is an extremely useful geometric construct, the uses of which include collision detection (Robotics), domains of action (Crystallography) and abstraction of geometric data (Image Processing). A comprehensive survey has been carried out by Franz Aurenhammer [1].

The first important algorithm for the calculation of Voronoi diagrams for point sites was due to Shamos & Hoey [11], who proposed a divide-and-conquer algorithm, where the input set was recursively divided into left and right halves, and the resultant diagrams merged back into each other, with an optimum run-time of O(n log n) for n sites.

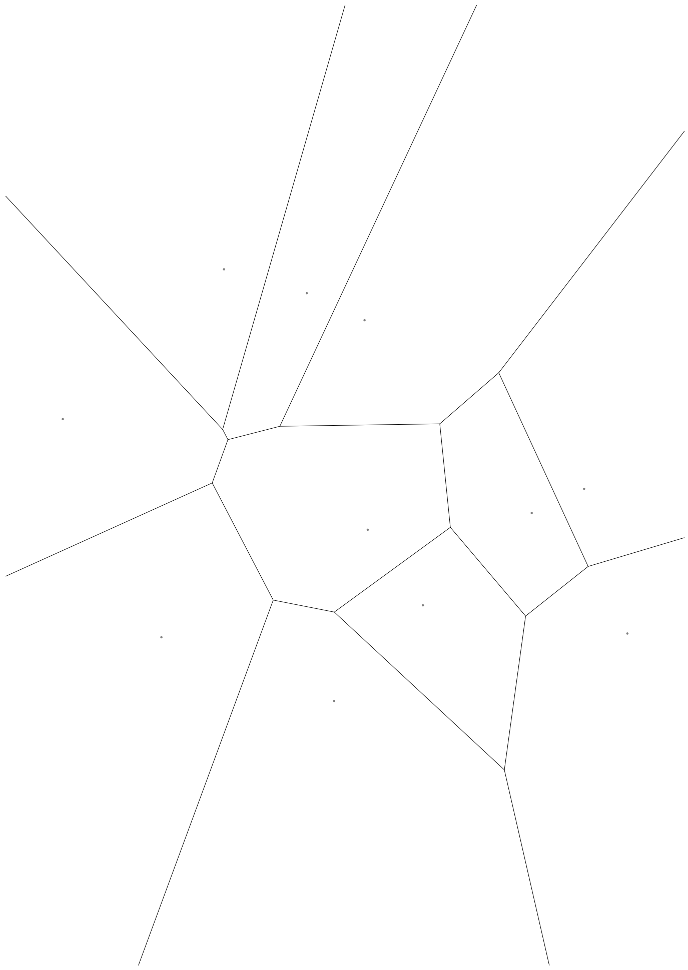
Yap [12] extended this algorithm to handle general 'sites' which are simple curve segments.

An alternative method of calculation was discovered in 1984 by Fortune [4]. Fortune's method involved considering a transform of the Voronoi diagram, obtained by translating every point in the plane upwards by the distance from it to the nearest site. This has the effect of leaving sites unmoved, and translating all Voronoi edges into either a horizontal half-line pointing up from the midpoint of the two sites (if the separator was a vertical line), or a curve, the lowest point of which is its intersection with the higher site. Every region except the lowest (assuming for simplicity that there is a unique lowest site) thus has a lowest point. We then sweep a horizontal line upwards over the sites, and for every site after the lowest, we create the curve associated with the site and the nearest of the already processed sites (i.e. the site



,

Figure 1: A general Voronoi diagram for circles



,

Figure 2: A general Voronoi diagram for points

controlling the region that the new site appears in), and start the relevant curves, one going left, and one right. Apart from the actual sites, the layout of the diagram also changes when two curves collide. This indicates a Voronoi vertex, and the two curves stop at that point, and from above there, a new curve is formed from the sites corresponding to the leftmost and rightmost of the three regions, excluding the middle. After all the sites have been encountered, and all the possible collisions of curves have happened, the modified Voronoi diagram is complete, and after untransformation, describes the actual Voronoi diagram accurately.

Richard Cole [2] made the observation that Fortune's transformation and untransformation were exactly equivalent to sweeping a vertical line, known as the 'reference line', through the sites from left to right, maintaining a construct of all point to the left of the line which were equidistant from the line and their nearest site. This construct is known as the 'beachline', and undergoes exactly equivalent changes to the list of curves in Fortune's paper, with the bonus that vertices can be calculated as intersections of untransformed rather than transformed separators. The beachline was first used by Chee Yap to facilitate parallel construction of the diagram. [5] [7] [8]

As Mentioned above, our version of the sweep-line algorithm was proposed by Colm. Ó Dúnlaing [9], who observed that it generalised naturally to convex sites, and stated without proof properties (dubbed "monotonicity" and "no surprises") leading to a correctness proof for the algorithm.

Our contribution is fourfold. First, we flesh out the details of a generic sweep-line algorithm. Next we supply proofs of several features of the algorithm, including the monotonicity and no surprises properties, under certain non-degeneracy assumptions. Third, we have developed a full working implementation for the case of circular sites. To our knowledge, this is the first implementation for sites which are not points or polygons. Fourth, empirical studies with this implementation suggest that the algorithm remains robust under several kinds of data degeneracy, these examples suggest that the algorithm is unexpectedly robust.

3 Definitions and Outline Description

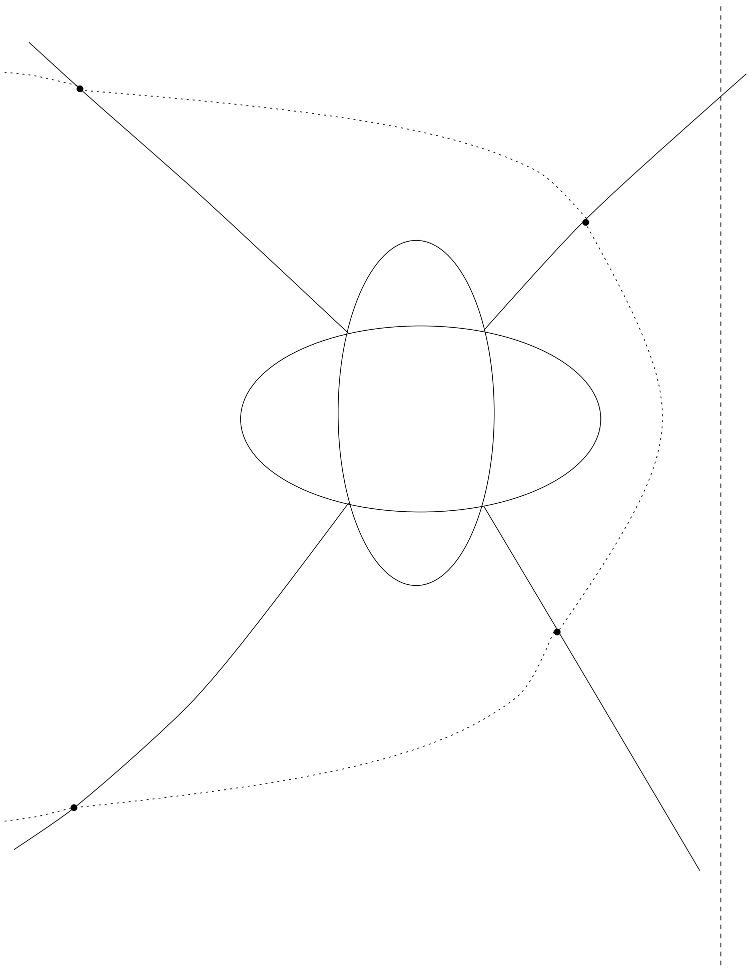
3.1 Definitions

A *site* is any of the convex sets, the separators of which form Voronoi edges.

The *beachline* is the locus of all points left of the reference line which are an equal distance from the line and the nearest site. The beachline is a collection of curves, each of which consists of points for which there is a unique nearest site, joined at points, known as *cusps*, which are an equal distance between the two closest sites and the reference line. A cusp is thus the intersection of the beachline with an edge of the Voronoi diagram.

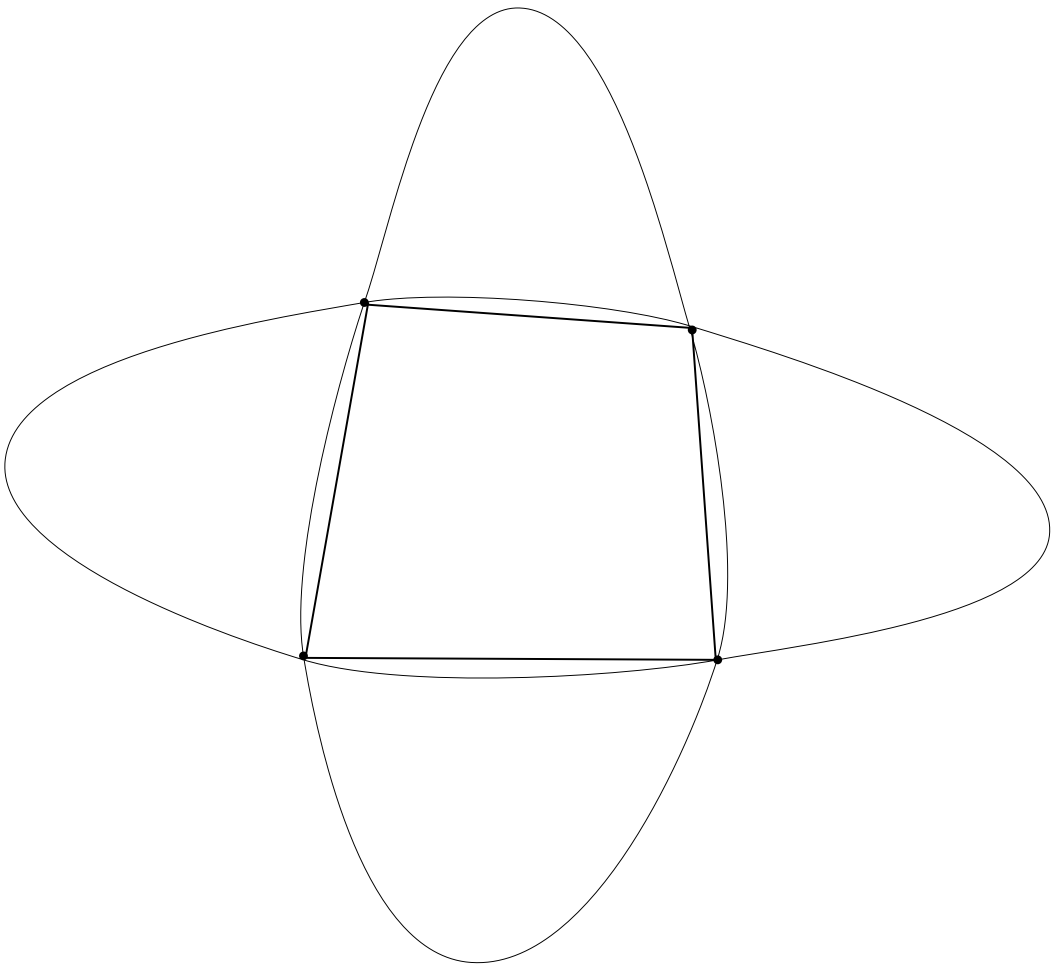
For circles or for disjoint convex sites, this is sufficient to describe a cusp, but generally two sites might have several disjoint Voronoi edges (See Figure 3).

To distinguish them, we divide the exposed site boundaries into disjoint curve segments, calling each one of them a 'part-boundary'. For $2n$ intersections, we divide the total area into $2n+1$ areas by connecting by straight lines each consecutive pair of intersections. See Figure 4. Obviously, no two such lines could cross, since this would mean that one of the two points on the second line would be on the curve between the two consecutive points on the first. Each of



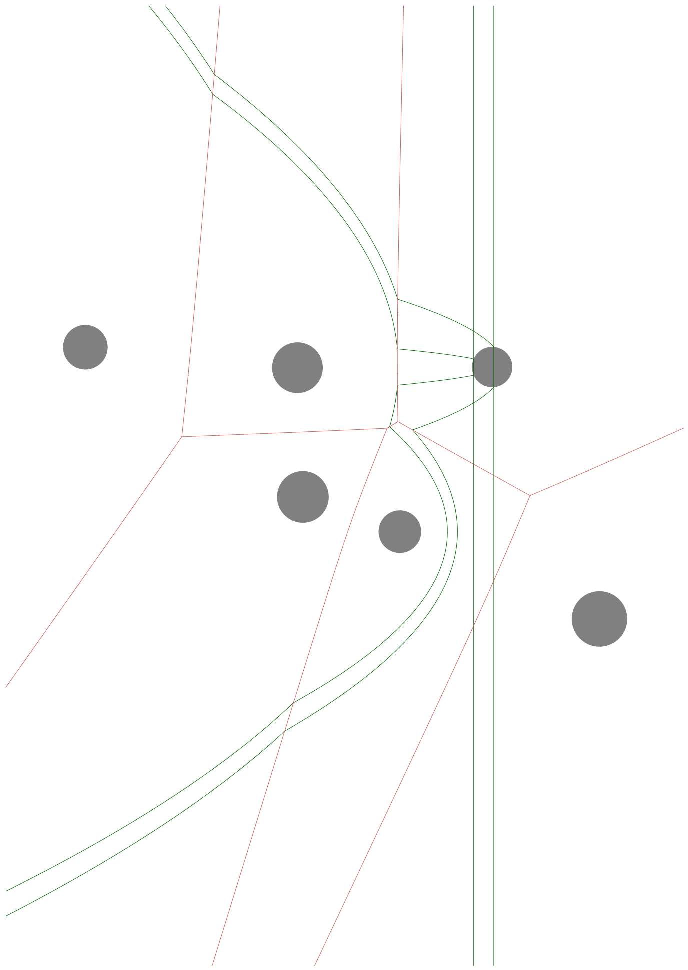
,

Figure 3: Two convex sites can produce more than two cusps



,

Figure 4: Part-sites are separated from the interior of the two sites by a straight line drawn between intersection points.



,

Figure 5: The ‘beachline’ travels left-to-right across the diagram, tracing out the Voronoi edges as it goes

these lines, then, separates a convex area, the boundary of which intersects the boundary of the union in a segment which is entirely inside only one site, from an inner area, which is entirely inside the intersection of the two sites. We call each of the outer areas *part-sites*. The part-sites include the entire boundary, and are disjoint, except at parts where two part-boundaries meet. For the points/circles implementation, there can be no confusion, as two circles can intersect only twice. For the greater part of this dissertation, we shall use the word ‘site’ as a synonym for ‘part-site’. Where the behaviour differs significantly, we shall use ‘part-site’.

3.2 General Approach

The method we shall employ is that of sweeping a vertical reference line across the sites left-to-right, and maintaining the ‘beachline’. See Figure 5. We shall show that the representation accurately describes the beachline at all times, and that this is sufficient to describe all points on the Voronoi diagram.

3.3 Overview

The method of representing the beachline is as a list of cusps, sorted in ascending order, i.e. increasing y-coordinate. All the places that the reference line is passing through when the beachline changes are stored as 'events' on a queue, and taken off in left-to-right order. We refer to the x-coordinate of the last event taken off as the 'current time', in the sense that it is continually non-decreasing: at any time t, the beachline $A(t)$ is correct for t and all times 'before' the next event.

While the reference line is passing through a site, the beachline dips in towards the site, and passes through exactly the same points as the reference line. The union of line segments where the reference line intersects sites is regarded as part of the beachline, and the endpoints of these line segments are called 'boundary cusps'. All other cusps we call 'normal'.

The places the beachline changes are:

1. The points when the reference line make a left or right tangent with a site: This is where boundary cusps are created and removed respectively.

2. At intersections of boundaries of sites: the boundary cusps change paths, and the point of intersection is one end of an actual Voronoi edge.

3. Vertices, where two normal cusps join together, and a different one starts.

3.4 Calculation

3.4.1 Input and Output

The algorithm takes information provided by a simple preprocessor, and provides information sufficient for any postprocessor to have an accurate and useful model of the Voronoi diagram. There already exists a postprocessor to convert the data into PostScript, a popular language for storing graphical data.

3.4.2 Objects

The beachline is represented as the list of cusps, arranged in ascending order. It is assumed that given two consecutive cusps, the details of the site determining the intervening curve, and the position of the reference line, we can determine the shape of the curve exactly.

Each of the cusps is stored as a record giving its defining sites, a reference to the point the cusp started at, and a marker to the point where it is expected to end, if any. The two sites are respectively designated as the 'lower' or 'higher' site, according to which of the adjacent segments they control. The boundary cusps have either their higher or lower sites marked as 'NULL', depending on whether they are bottom or top boundaries respectively. This perpetuates the invariant that the higher site for any cusp is the lower site for the cusp's successor. Thus, any process that looks at a beachline where the reference line passes through a site will know from this convention that the beachline dips in towards the site, runs through it for as long as the reference line does, and sprouts out the other side. The reference line can pass through two or more sites, without emerging between. In these cases, the first cusp indicates a bottom boundary for the lowest site, and the next indicates an upper boundary

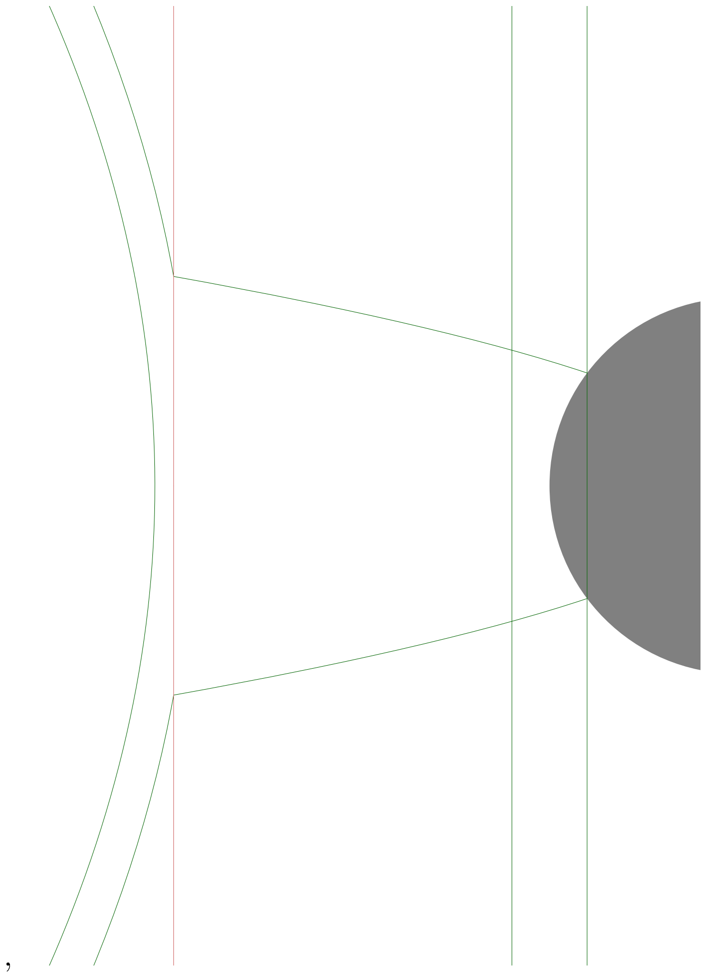


Figure 6: At a ‘start’, we insert into the beachline a normal cusp, a pair of boundary cusps, and a second normal cusp, in that order

for the highest. The details of what goes on the state of the sites between is intentionally ignored. A cusp is often written (lower site, higher site), so that a beachline might be written as: $(a, b)(b, c)(c, d)(d, b)(b, a)$. The boundary cusps trace out site boundaries, and the normal cusps trace out Voronoi edges.

3.4.3 Events

The program only considers the beachline at events when it changes. The effects that the events have on beachline changes are:

1. The ‘start’ events, i.e. left tangents to sites, where the algorithm adds to the beachline two new cusps between the new site and the site controlling the beachline curve immediately left of it, as well as two boundary cusps. See Figure 6.

2. The ‘end’ events, i.e. right tangents to sites, where the beachline deletes two boundary cusps. See Figure 7.

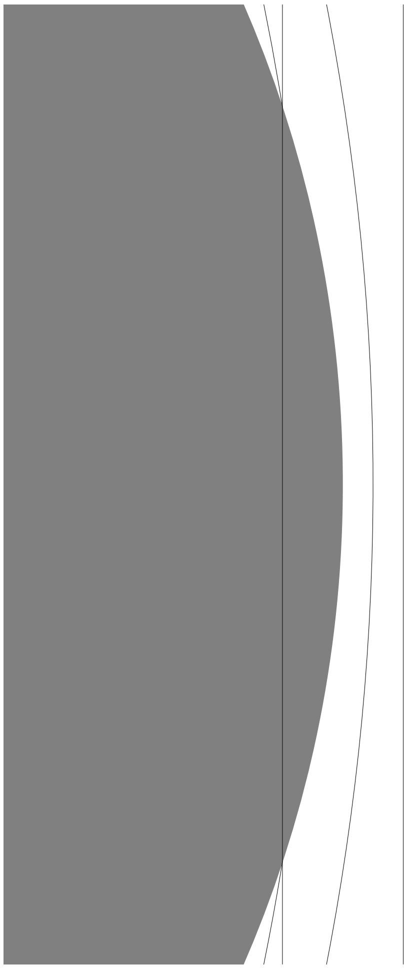
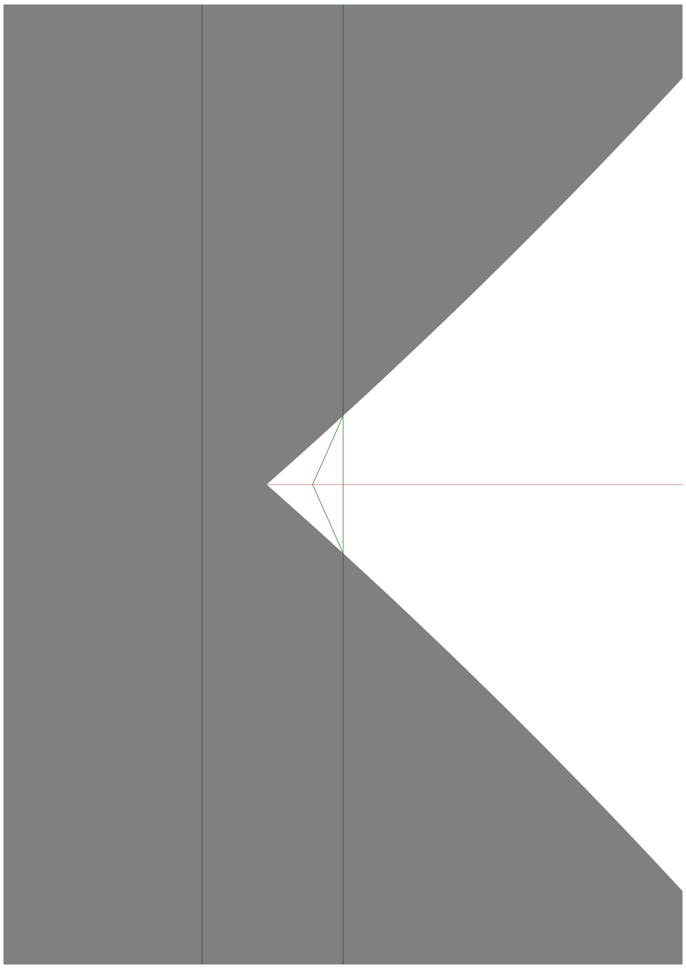
,


Figure 7: At an 'end', we delete a pair of boundary cusps from the beachline



,

Figure 8: At an ‘L’ intersection, we insert into the beachline a lower boundary cusp, a normal cusp, and an upper boundary cusp, in that order

3. Any place where two site boundaries cross. A cusp always comes out of these, for obvious reasons. We separate these into four categories according to how they affect the beachline:

   - ‘L’ intersections: This is where both the boundaries issue right from the intersection. Here we insert into the beachline a top boundary, a normal cusp, and a bottom boundary, in ascending order. See Figure 8.
   - ‘R’ intersections: This is where both the boundaries issue left from the intersection. Here we remove a top boundary, a normal cusp, and a bottom boundary, which should be present in that order. See Figure 9.
   - ‘A’ intersections: This is where both the boundaries are on the bottom half of their respective sites. Here we remove a bottom boundary, and replace it by a normal cusp and a different bottom boundary, in that order. See Figure 10.
   - ‘V’ intersections: This is where both the boundaries are on the top half of their respective sites. Here we remove a top boundary, and replace it by a different top boundary and a normal cusp, in that order. See Figure 11.

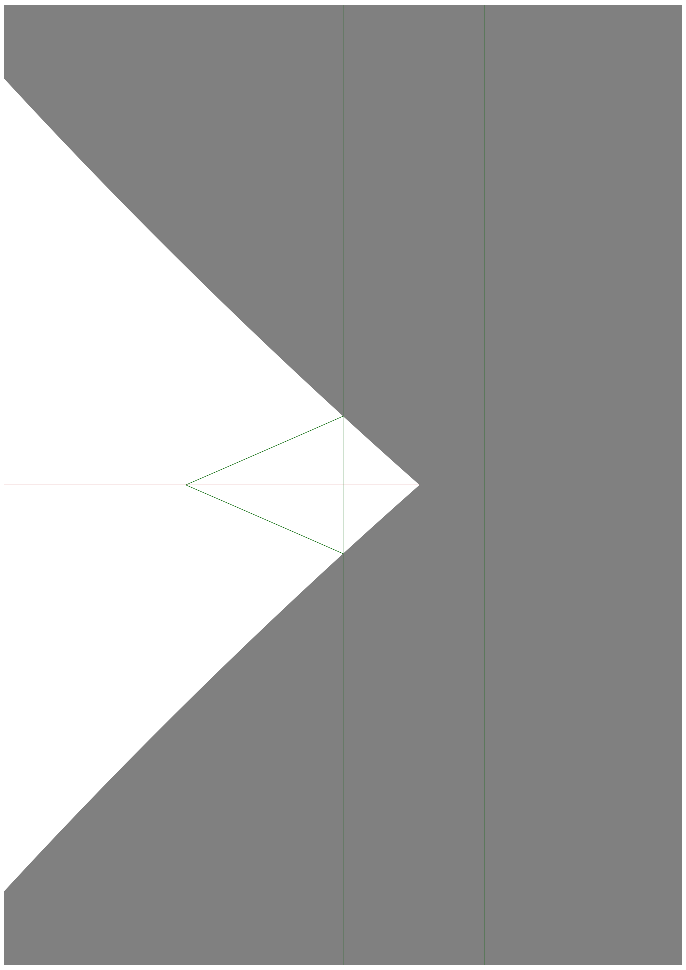

,

Figure 9: At an ‘R’ intersection, we delete from the beachline a lower boundary cusp, a normal cusp, and an upper boundary cusp.

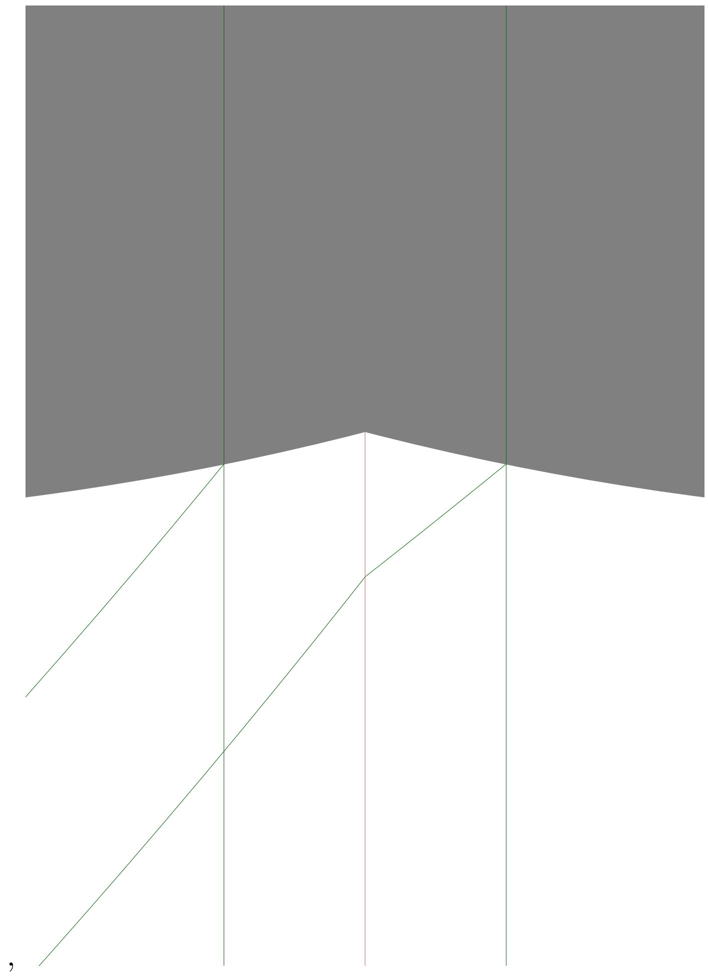
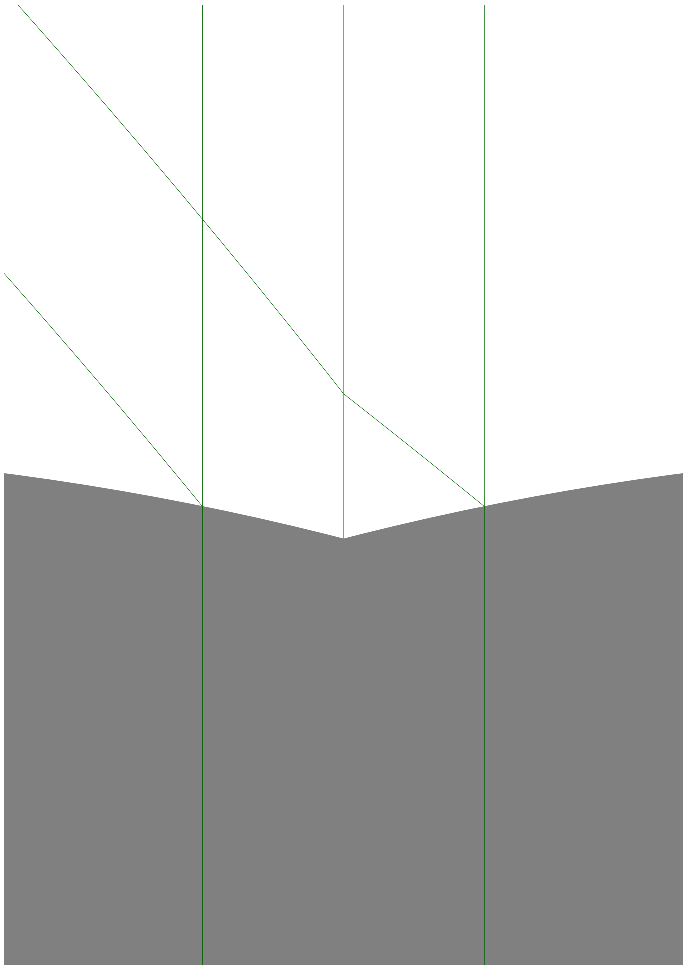


Figure 10: At an ‘A’ intersection, a lower boundary cusp stops and a new boundary cusp is inserted, along with a new normal cusp heading downwards



,

Figure 11: At a ‘V’ intersection, an upper boundary cusp stops and a new boundary cusp is inserted, along with a new normal cusp heading upwards

4. When two cusps join together, forming a vertex: Here we remove the two (necessarily adjacent) cusps from the beachline, and form a new cusp belonging to the lower site of the first (lower) cusp, and the higher site of the second (higher) cusp, and place this cusp in the place vacated by the other two. At the time this happens, the reference line will be tangent on the right to the empty circle touching the three sites (and centred at the vertex).

Events of class 1,2 and 3 above are called boundary events.

3.4.4 The preprocessor

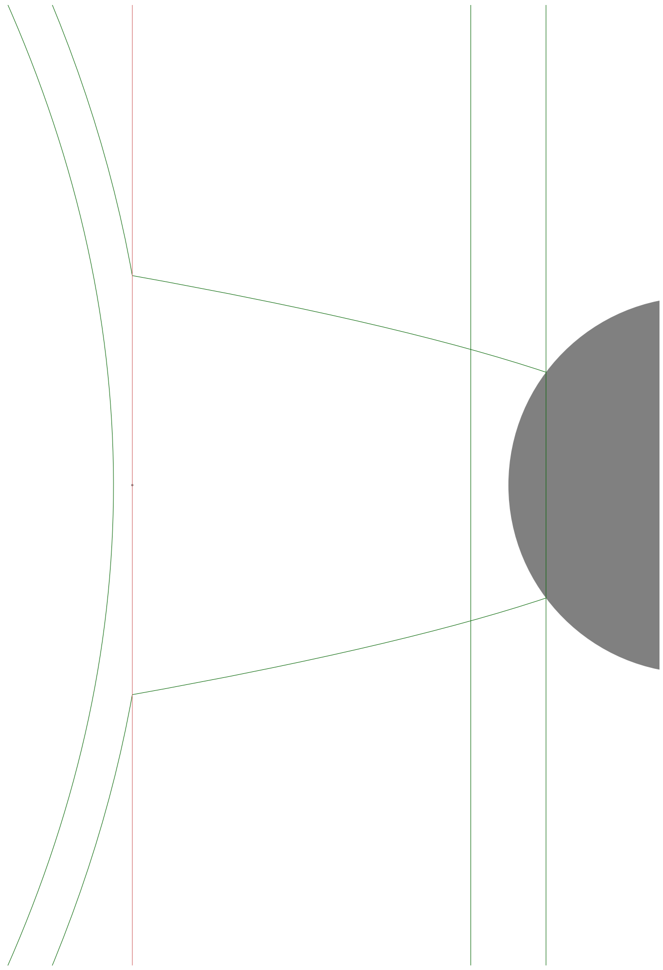
It is assumed that the preprocessor will determine where boundary events occur, remove those that occur inside other sites, and sort them by x-coordinate. Due to the fact that the algorithm has to cater to 'start' and 'end' events, it is necessary to divide the sites into upper and lower halves. Thus, every boundary event has information as to its next upper and lower events imparted to it by the preprocessor. Only 'start' and 'L' events are ever actually supplied both. This information is used to inform future events as to which beachline cusps lead in to them. For single-cusp events such as 'A' or 'V', we specify the only one available. For events with two (or more) incoming cusps, such as 'R' and 'end' (and vertex), we specify the lowest of them, and the algorithm uses this information according to this convention. Also, the preprocessor names which two sites the events are attached to, with one of the entries being null for the single-site events, 'start' and 'end'.

3.4.5 Pseudo-vertices

The exception to the above is the event type 'pseudo-vertex'. When a site is encountered, an edge between the event's point and the nearest previous site starts, in the form of two cusps heading up and down from a point directly to the left of the site (See Figure 12). Since the postprocessor needs to have this point in order to draw the edge, and the only access to the output function is in the recording of events, we find the point, create an event of type 'pseudo-vertex' to represent it, and record it on the output. The 'pseudo-vertex' is not strictly speaking an event, since it is never on the event queue, and its labelling as such is only to preserve the C++ programmer's version of Occam's razor, which states that, where possible, all similar actions (such as output) be implemented as the same function on different varieties of the same objects (different types of events).

3.4.6 The direction of time

The events, with the exception of 'pseudo-vertices', are stored in a queue, indexed by the time at which they affect the beachline. For boundary events, this is their x-coordinate, whereas for vertices, this is the x-coordinate of the vertex plus the distance from the vertex to any one of its closest sites. The boundary events are taken from the preprocessed input, and placed on the queue after all the information about them has been gleaned from the input. They are then taken off in order of increasing x-value, or, in the case of two events with the same x-coordinate, in the order 'start' - 'R' - ('A' or 'V') - 'L' - 'end' - vertex. This ordering is in the



,

Figure 12: At each ‘start’, two cusps start from a pseudo-vertex

order of the manipulations that these events induce on the beachline, and ensures that nothing untoward happens in the case of zero-radius circles, for example.

3.4.7 Overview of the algorithm

The first thing to happen is that the sites are read in, together with indices, and echoed to output in the same fashion. They are indexed by these numbers when they are placed on a list immediately. After this, the boundary events are read in, assumed pre-sorted by x-coordinate, and placed on a list of their own, indexed by their own numbering. The links to sites are made using the indexed site-list, and links to other events are similarly made from their own event-list (remember that all links are leftwards, to sites already read in). After each event has established its links to other sites and events, it is placed on the event queue. After all of them are enqueued, the queue is passed on to the interpretation routine. The beachline is initialised to consist of the two boundary cusps, and a note is made that this was the first site. The remainder of events on the queue are 'processed' until the queue is empty, at which point the beachline is artificially advanced (since all changes have taken place, this just means incrementing the time counter by a percentage). All the cusps at this stage are semi-infinite, so we output the current positions of all the cusp, denoting them by the title EXTENSION, so that a rendering program will have (arbitrary) markers on these cusps.

3.4.8 Event processing

The processing done on each event basically consists of reporting the event to output, and the beachline changes mentioned above. For most boundary events, reporting merely entails taking a unique number from the index counter, and outputting what type of event it is, its physical position, its new index, and which site(s) it is associated with. For 'R' events, as well as vertexes, we also name the normal cusp(s) (one for R, two for vertex) which have ended there, as well as where they started, and what sites were involved. It is assumed that this is sufficient information for the postprocessor to draw the edge.

3.4.9 Physical determination

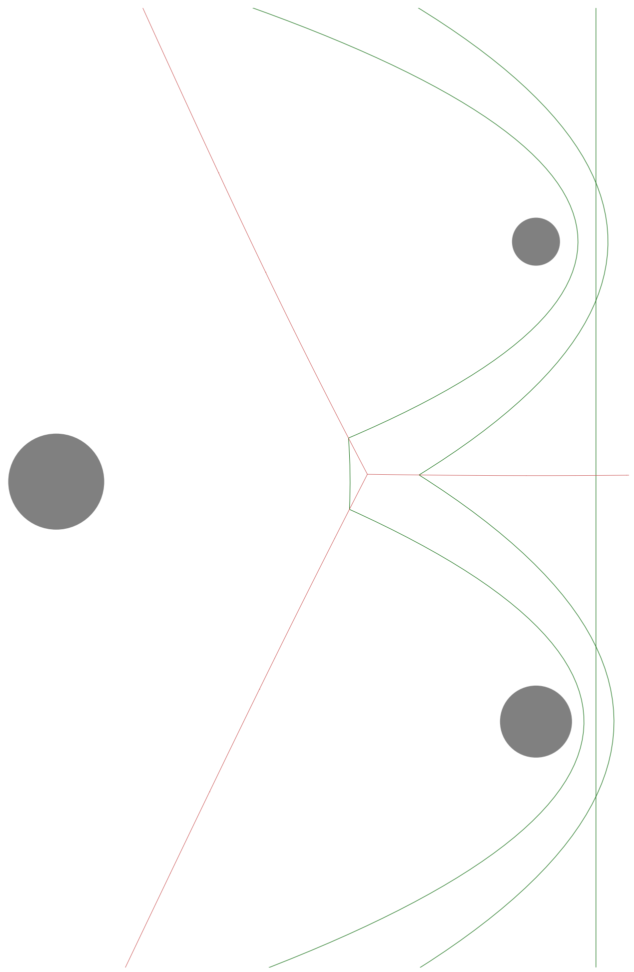
For the beachline manipulations induced by each event to take place, there must be some indication, for each event, as to which cusp(s) in the structure the events apply to. Leaving vertexes to be discussed later, most of the boundary events inherit a cusp started by another event, and are informed of this due to the previous event having a pointer to the next event expected on the upper/lower half of the site. 'R' and 'end' remove cusps, 'A' and 'V' take whatever cusps are given to them, replace them and notify any later events, and only 'start' and 'L' actually create cusps, and find a place in the beachline for them. The method they use for this is not of interest at this point, but eventually the search locates the highest cusp below the given point. In the case of an 'L' event, it is assumed that this is a bottom boundary, and then we insert the three new cusps immediately above it. For a 'start', the cusp we have found is certainly *not* a lower boundary cusp. The upper of the two regions it is defined by will belong to the site nearest to the 'start', and the two new cusps defining the new edge will be between it and the site of the 'start'. In the event of the search returning a NULL cusp as the greatest lower bound cusp (indicating that the 'start' is on the lowest beachline segment, which is unbounded.), then the rôle of the nearest site to the beachline at that point is taken by the first site ever encountered. The processor, after finding the cusp, and so the site, discerns where the cusps will start, creates the 'pseudo-vertex' there, and then inserts the four cusps: the lower of the two normal cusps, the lower boundary cusp, the higher boundary cusp, and the higher of the two normal cusps. It should be noted that if the site a was encountered before the site b, then both cusps (a, b) and (b, a) can be traced back to an origin point to the left of b. Here the cusp (a, b) would have headed down, and the cusp (b, a) headed up. These cusps would have stayed in the same relative position to each other ((a, b) always below (b, a)), since any cross-over of y-coordinate would imply a time at which the y-coordinates were equal, and so the curve between the reference line and the site b would have two intersections with the horizontal line at that coordinate, which is impossible. This result, that for a encountered before b, (a, b) is lower than (b, a), holds true even if b didn't form those two cusps at the start, and is of use in the function to determine the position of a cusp, given the reference line and two sites.

3.4.10 Vertex scheduling

A vertex is scheduled for whenever two cusps will meet. The scheduling function can be called on any particular cusp, and determines whether it will hit its successor. It first checks that the cusp's successor exists, and determines whether the two cusps, if left to themselves, will converge. The main part of this is the actual finding of an empty circle, if any, and differs according to the types of convex site involved. The second part is ensuring that the vertex produced was the product of the cusps (first, second) and (second, third), and not the product of (third, second) and (second, first). This is proven below to be equivalent to checking that the tangent points to the sites first, second and third are in clockwise order around the putative vertex. Assuming that the cusps are due to meet a given time, the scheduler then checks that neither the successor nor the predecessor to the cusp in question are due to expire before the time. If so, nothing is done; The earlier convergence will cause the cusp or its successor to be deleted before converging, and so we wait until it happens, when this cusp will try to be scheduled to converge again. Assuming we get by this, we delete any already scheduled finishes for this cusp and its successor, create a vertex event, (with the knowledge that this cusp will be the lower of the two), and set the cusp's expected finish to be the new vertex. This scheduling routine is called wherever a new cusp is created, or an old cusp is given new neighbours: twice after a 'start', once after an 'end', twice after a vertex, and once after either an 'A' or a 'V'.

3.4.11 Vertex processing

When a vertex is processed, it simply takes its given cusp, removes it and its successor, and replaces them with a new cusp made from the lower site of the first and the higher site of the second (See Figure 13.) It sets the start of the new cusp to be itself (as do all the other events that create cusps:'A', 'V' and 'R'), and, as mentioned above, calls the scheduler on the new cusp and its predecessor.



,

Figure 13: At a vertex, two cusps end and one begins

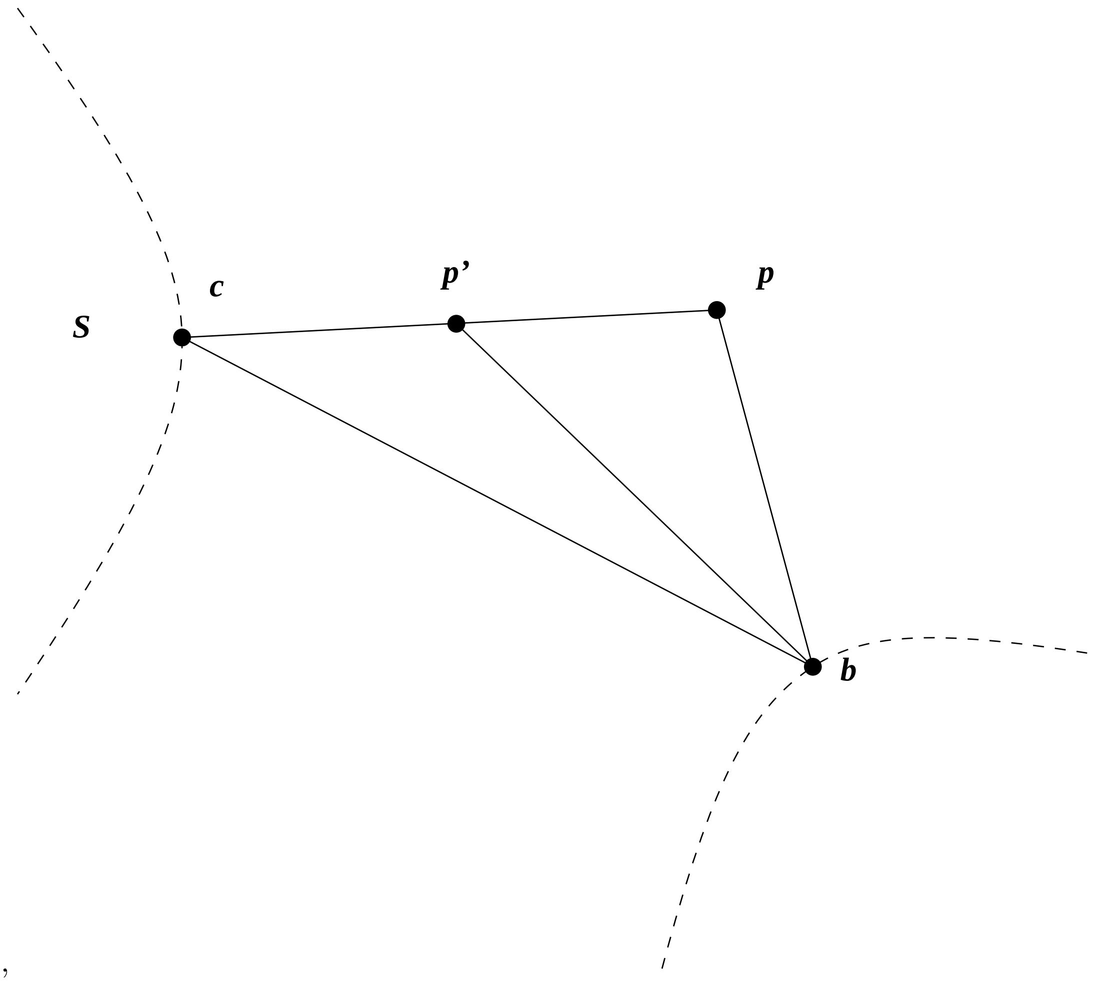


Figure 14: Regions are star-shaped

4 Properties of Diagram and beachline for convex sets

In this section, we give an outline correctness proof for the algorithm. For simplicity, we assume certain non-degeneracy conditions:

a) Site boundaries intersect transversally, or at least if they touch, then the sites are not in a common half-plane, bounded by a line of tangency.

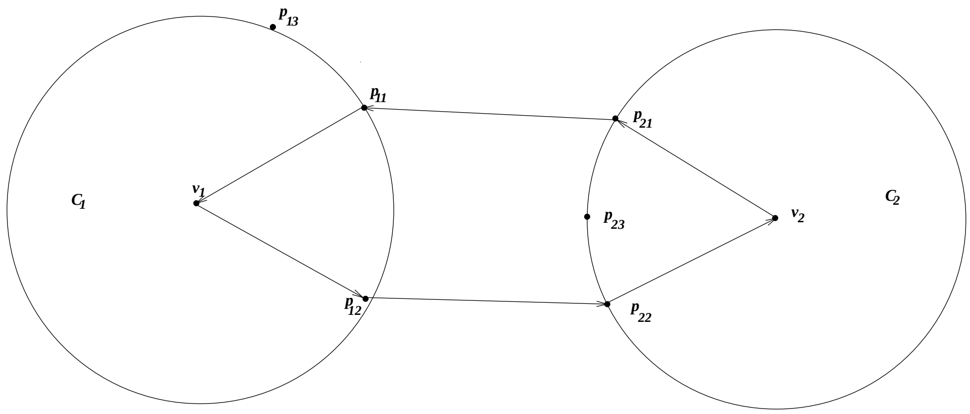
b) Events do not occur simultaneously. This includes the assumptions that sites do not have common left or right tangents and that vertices have exactly three closest sites

Given any point v, its *spokes* are the line segments joining v to the closest points on the closest sites. From the non-degeneracy assumptions, every vertex v has exactly three spokes.

4.1 Theorems

Lemma 1 *Each region is star-shaped in that a line from any point in the region to the nearest point on the closest site must stay within the region.*

Proof: (See Figure 14) Let p be a point in a region, and c the point closest to p on a closest



,

Figure 15: The clockwise/anti-clockwise natures of two different vertices for the same three part-sites are converse to each other

site S, p' any point on the open line-segment from c to p. Let b be any point on any site, except c. Then $d(p,b) \geq d(p,c)$. Now $d(p',b) \geq d(p,b) - d(p,p') \geq d(p,c) - d(p,p') = d(p',c)$, with equality only if b is on the line pc, on the same side of p as c is, so $b = c$. which is impossible. Thus, p' is strictly closer to S than to any other site.□

Corollary 1 *Spokes for distinct points p and q to distinct sites S_1 and S_2 repectively cannot intersect on their interiors.*

Proof: Such an intersection point would lie in the interior of two distinct Voronoi regions, which is impossible □

Lemma 2 *Let S_1, S_2 and S_3 be three part-sites, and suppose that there are two vertices v_1 and v_2 equidistant from them. Then the clockwise/anti-clockwise nature of the nearest points on the three sites is opposite for v_1 as it is for v_2.*□

Proof: Let C_i be the tangency circles with centre v_i. Let v_ip_{ij} $(1 \leq j \leq 3)$ be the spokes from v_i $(i = 1,2)$. They are disjoint except at endpoints. Note $p_{1j}p_{2j} \in S_j$ $(1 \leq j \leq 3)$. W.L.O.G. $p_{11}p_{12}p_{13}$ are in clockwise order around C_1. Consider the closed path $v_1p_{12}p_{22}v_2p_{21}p_{11}v_1$ (See Figure 15) containing 6 line segments (though 2 of them may be degenerate). Now, the four spokes are disjoint except perhaps at endpoints, and they are exterior to all sites except at p_{ij}, and the closed line segments $p_{1j}p_{2j}$ are disjoint. Certainly their interiors are disjoint, being in part-sites whose interiors are disjoint, and p_{11} is not interior to $p_{12}p_{22}$, since it is not interior to S_2; $p_{11} \neq p_{12}$ since the spokes from v_1 are disjoint except at v_1, and $p_{11} \neq p_{22}$. For suppose that $p_{11} = p_{22}$, and thus the circles meet. Since S_1 is a convex site tangent to the circle C_2, and p_{11} is an intersection point of the two, it must also be the tangency point, i.e. $p_{11} = p_{21}$. Similarly $p_{12} = p_{22}$. Thus both sites are included in the half plane bounded by the line normal to the spoke vp_{11} and not including C_1, which is an excluded degeneracy. Thus the path forms a Jordan curve. But p_{13} is outside it; therefore so is p_{23}, so p_{21}, p_{22}, p_{23} are in anticlockwise order □

Corollary 2 *For any three part-sites, there are at most two vertices equidistant from those part-sites.*

Note: This result was proven for disjoint convex sites by Leven and Sharir. [6]

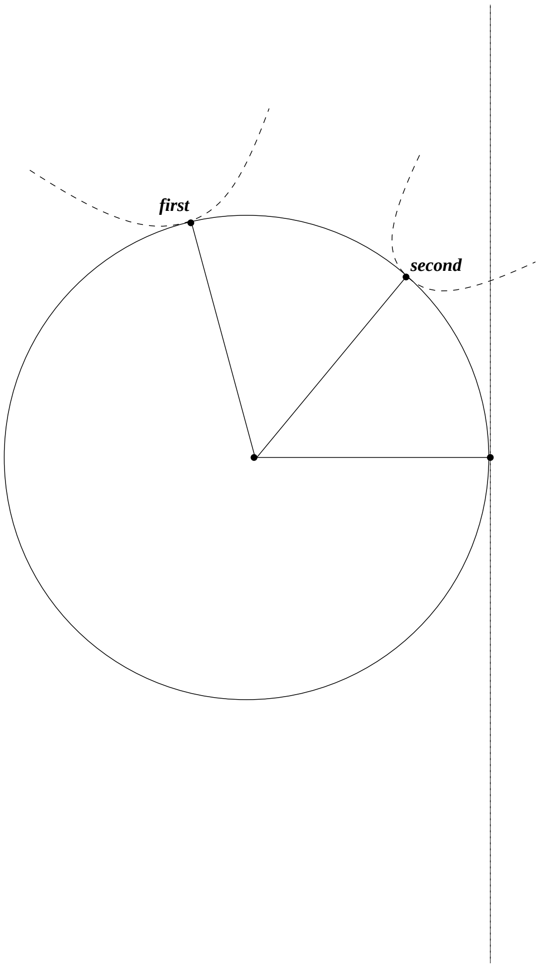
Lemma 3 *At any time t, the actual beachline, $A(t)$, can be uniquely defined by t and the list of (part-) sites controlling each of the segments.*

Proof: Obviously, the bottom-to-top ordering of the sites that control parts of the beachline is unique for any given beachline. For suppose that there are n part-sites. Then each cusp is defined by two part-sites, and each pair of part-sites, together with the reference line, can define a maximum of two cusps, by Corollary 2. Thus at any time there is a finite ($\mathrm{O}(n^2)$) number of cusps on the beachline. Since, by Lemma 7, the point on the beachline with any given y-value is unique, a listing of sites controlling the beachline, listed in order from the site controlling the lowest segment to the one controlling the highest, is unique except where the point on the beachline is closest to more than one site. Since, by Lemma 8, the beachline at these is only made up of two cusps, specifying them by the sites controlling the segments above and below is unambigious.

Given an ordering of sites, we can determine the position of each cusp by calculating the two possible cusps equidistant from the part-sites and the reference lines, and discarding the one for which the 'lower' site controlled the beachline above the cusp. This uniquely determines the position of each cusp, since each three part-sites can only produce two possible equidistant sites (Considering the beachline to be a suitably extendable part-site), by Corollary 2. For any two given cusps, and the site that controls the segment between them, the determination of the curve between them is assumed to be easily calculable, as specified in Section 5 □

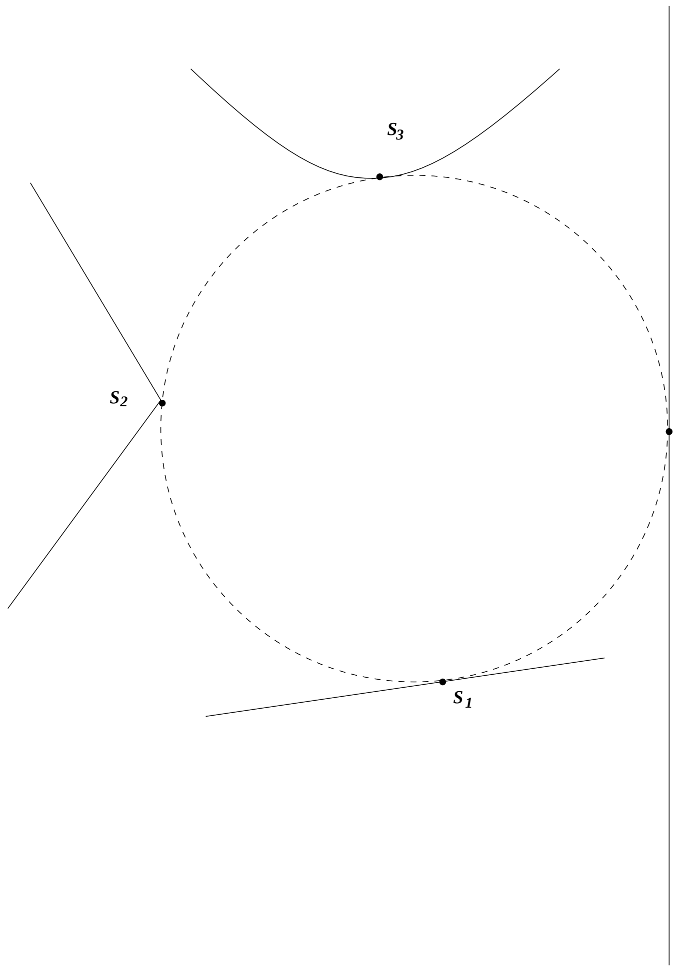
Lemma 4 *Given the cusp (s_1, s_2), and the part-sites s_1 and s_2, then s_1, s_2 and the nearest point on the reference line are in clockwise order around the cusp.*

Proof: Since the position of the cusp is only determined by the sites and the reference line, we can assume without loss of generality the s_1 and s_2 are the only sites. We thus trace the cusp and its dual (s_2, s_1) back to the pseudo-vertex where they start. Relabel the sites first and second, according to the order in which they were encountered. The cusps start directly left of the leftmost point on second, and (first, second) heads down, and (second, first) heads up. It should be noted that the creation of the cusps is the only time that second's tangent point from either cusp is on the same y-coordinate as, and on the right of, the cusp, since thereafter such a point of tangency would be closer than the reference line. Consider the state of the cusp after a small amount of time (i.e. while the cusp is still left of the 'start' point on second, and before any change to the beachline). During this time, the lower cusp must be below its tangent point on second. For suppose that it wasn't, and the tangent was below the cusp and, obviously, not left of the 'start'; then the point where the horizontal line through the cusp met the line joining the tangent and the 'start' would be inside the site (by convexity), above and not right of the tangent, and thus closer to the cusp, by elementary trigonometry. This is a contradiction, so as long as the lower cusp is left of the 'start', it remains lower than the tangent point on second. Thus, after this small amount of time, we have the tangency points on the line (directly right), on first and on second (right, and above). For the tangency points not to be in clockwise order, the tangency point on first would need to be between that of second and the line, which, by taking a time arbitrarily close to the time of the start of the



,

Figure 16: Initially, the tangency points for each cusp are in clockwise order



,

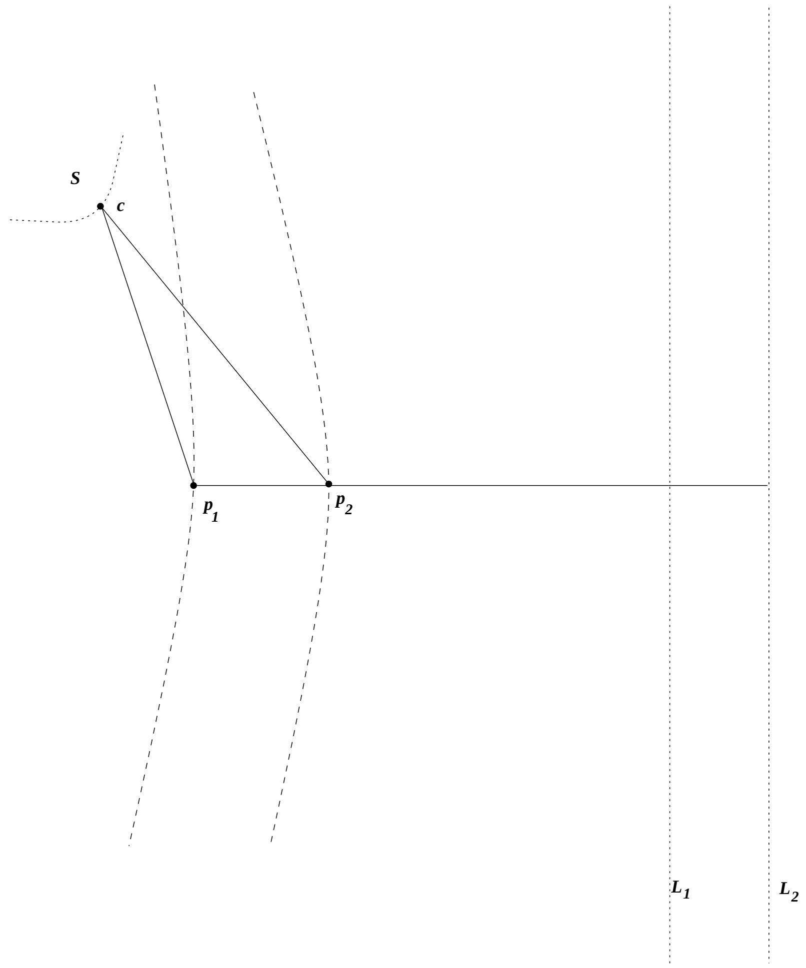
Figure 17: The tangents points for S_1, S_2, S_3 and the reference line lie clockwise around the circle

cusps, we can prove impossible. Thus, the cusp starts out with the points in clockwise order. A similar argument applies to (second, first). See Figure 16.

Due to the continuity of tangent points on convex sites (including the reference line), the clockwise/anti-clockwise nature of the tangency points will only change if one of the tangency points passes between the cusp and another tangency point. In this case, this necessitates that the tangency points become collinear with the cusp, either at different distances (by definition of the cusp, impossible) or at same point (degenerate data), or that the cusp pass left of the line passing through the 'start' event of either second (happens at start, but not elsewhere), or first (by definition, not during lifetime of cusps). Since none of these events happen after the arbitrary time already chosen, the clockwise/anti-clockwise nature of the tangency point remains constant through the lifetime of the cusp. □

Lemma 5 *If the two cusps (s_1, s_2) and (s_2, s_3) converge to a vertex, then the tangency points to the sites s_1, s_2 and s_3 at the time of the vertex event are in clockwise order.*

Proof: (See Figure 17) At the time of the vertex, the two circles of tangency for both cusps become one, centred at the vertex and with radius equal to the distance from the vertex



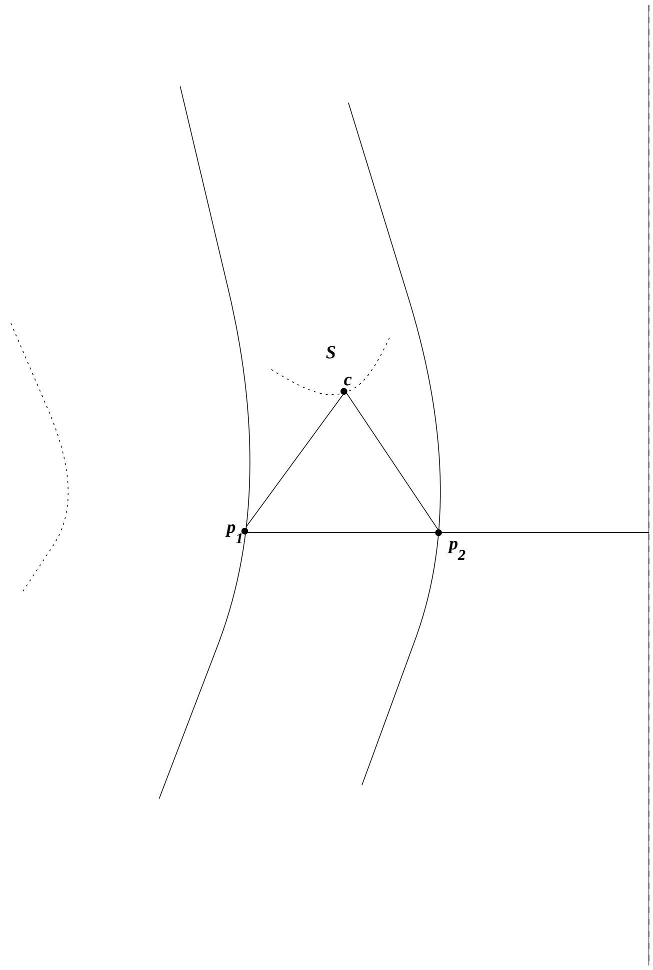
,

Figure 18: As the reference line moves left, so does the point on the beachline for any given y-coordinate

to any of the tangency points. Starting with the tangent point to the reference line and going clockwise, the circle is split into the three sections line-s_1, s_1-s_2, and s_2-line, since (s_1, s_2) is a legitimate cusp. If the tangency point to s_3 was in either of the two section adjacent to s_1, then the tangent points for (line, s_2, s_3) would go anti-clockwise, which is impossible. Thus the order is line-s_1-s_2-s_3-line, and the tangent points go clockwise. $\Box$

Lemma 6 (Monotonicity Lemma) *For any given y_1, the point where the beachline meets the horizontal line $y = y_1$ moves right as the reference line moves right.*

Proof: (See Figure 18) Let $p_1 = (x_1, y_1)$ be the beachline point with y-value y_1 with reference line $x = x_1 + d$. Let S be the closest site to p_1, and let c be the closest point on S to p_1. So $d(c, p_1) = d(p_1, L) = d$. Let $p_2 = (x_2, y_1)$ be the beachline point with y-value y_1 for a reference line to the right of the original, i.e. $x = x_1 + d + \delta$, $\delta > 0$. Suppose $x_2 \leq x_1$, i.e. p_2 is not right of p_1. Then $d(p_2, c) \leq d(p_1, c) + d(p_1, p_2) = d + d(p_1, p_2) = d + x_1 - x_2 < (x_1 + d + \delta) - x_2$. This last term is the distance from p_2 to the second reference line, so a beachline point (p_2) is closer to a point on a site than to its reference line, contrary to definition. Thus $x_2 > x_1$, and p_2 is further right than p_1. $\Box$



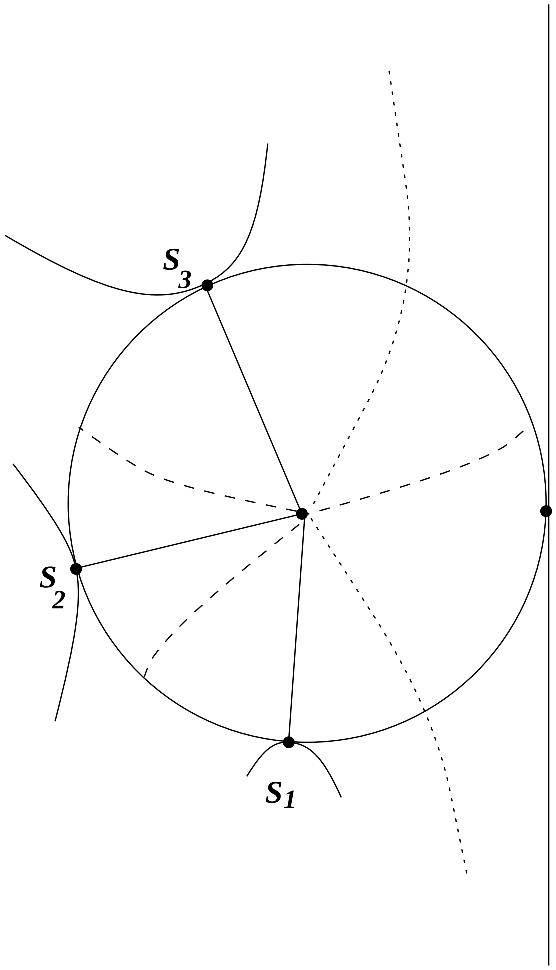
,

Figure 19: Each horizontal line intersects the beachline at most once.

Lemma 7 *With the exception of the case where the reference line touches a site on the left, there is exactly one beachline point with any given y-coordinate, if there exists a site point left of the reference line.*

Proof: (See Figure 19) Suppose that for the line $x = x_1$ there were two such points, $p_1(x_2, y_1)$ and $p_2(x_3, y_1)$, with p_1 further from the line than p_2 (i.e. $x_2 < x_3$). Let the nearest site to p_2 be S, and the nearest point to p_2 on S be c, so that $d(p_2, c) = x_1 - x_3$. Then $d(p_1, c) \leq d(p_2, c) + d(p_2, p_1) = (x_1 - x_3) + (x_3 - x_2) = x_1 - x_2$. This last is the distance from p_1 to the line and so the further point is nearer or equal to a point on a site (c) than to the line, with equality only happening if p_1, p_2 and c were in a straight line. But then c would be as far to the right of p_2 as p_2 is to the left of the line, i.e. c would be on the line. This is contrary to the condition, so p_1 is strictly nearer to c than to the line, which is impossible. Thus, there is at most one beachline point with the y-coordinate y_1.

It remains to show that there is at least one point on the line $y = y_1$ for the reference line $x = x_1$. Given the existance of one point p on a site S, the intersection (x_2, y_1) of the line $y = y_1$ and the bisector of the points p and (x_1, y_1) will be equidistant from p and the reference line, and thus the minimum distance to S is less that the distance to the reference line. The



,

Figure 20: The separators cannot leave the space bounded by the spokes

point (x_1, y_1), on the other hand, is be default closer to the line than to S, unless the point (x_1, y_1) is contained in S, in which case it is by default on the beachline. So, by continuity, there exists a point on the line $y = y_1$ between (x_1, y_1) and (x_2, y_1) which is equidistant from S and the reference line. Similar points can be found for any other sites, and the point with the largest x-coordinate will be closer to the line than any of the others, by the argument used in the first part. $\square$

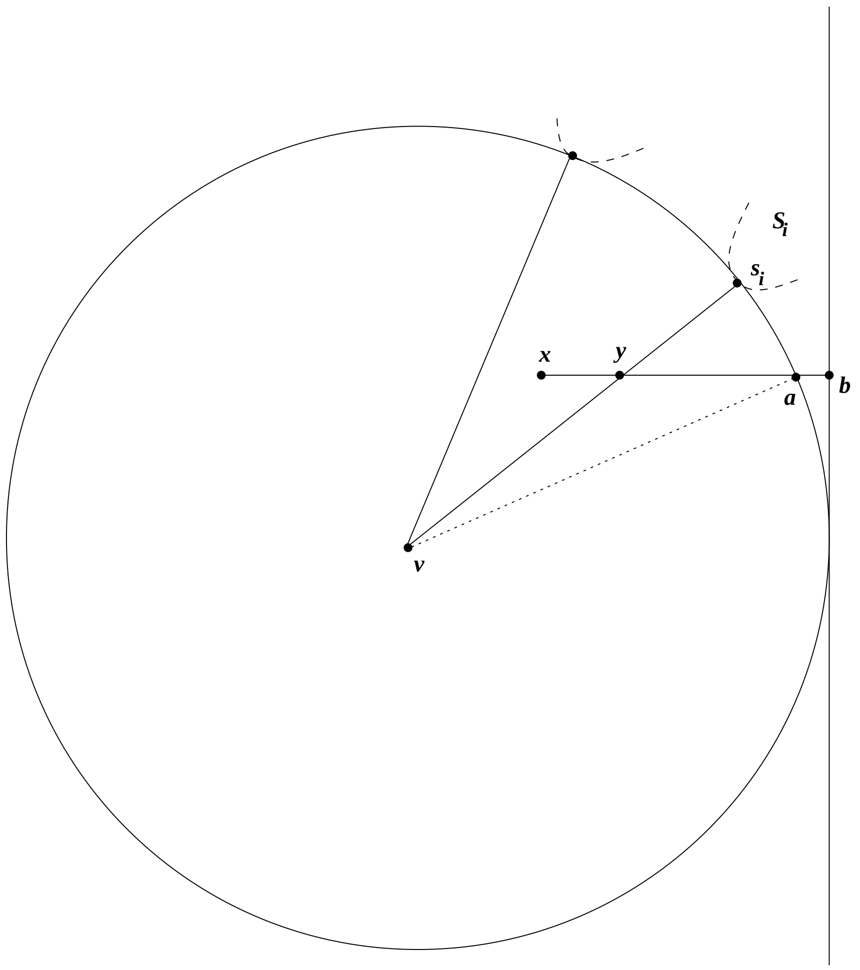
Lemma 8 (The no-surprises lemma) *If three cusps intersect in a vertex then, at the time when the beachline passes through the vertex, the beachline is to the right of two of the separators, and to the left of the other, in the sense that there exists a radius ϵ such that inside an disc of radius ϵ centred at the vertex, one of the separators is entirely contained in the half of the disc which is right of the cusp, and the other two are entirely inside the other half.*

Proof: Consider the tangency circle centred at the vertex v The three sites appear on this circle as tangent points. Given any two of these sites, the separator, while inside the circle, can be divided into the part that strikes the arc between the two sites that doesn't include any other site, and the part that strikes the circle between two other sites. We note that the first part of the separator doesn't pass out of between the area directly between the spokes to the two sites. For suppose that it did, and the separator of S_1 and S_2 passed through the spoke to S_1, at the point x, say. Thus the tangent point to S_2 for this point must lie on the circle with centre x and radius the distance between x and the tangent point to S_1. But this circle is entirely contained in the circle centred at the vertex, mentioned at the start, and there is no point of S_2 inside this circle, except the tangent point, located at the edge. This would mean that the tangent points for S_1 and S_2 were identical, which is degenerate data. Thus the separator remains inside the two straight lines. See Figure 20. Now let the controlling sites be S_1, S_2, S_3, where S_1 is the site belonging to the first tangent point anticlockwise from the intersection of the circle and the reference line, S_2 is the second anticlockwise around the circle, and S_3 is thus the site with tangency point first around the circles clockwise from the right tangent point. Now suppose that the beachline is entirely contained in the secant between the spoke vS_i, and vS_{i+1}, $1 \leq i \leq 2$. Then either v, s_i, or s_{i+1} is the rightmost point of the secant, where s_j is the tangency point of S_j, $1 \leq j \leq 3$. Let x be any point in this secant. Consider the line heading directly right from x and let y, a and b be the points where it hits the boundary of the secant, the boundary of the circle, and the reference line respectively. See Figure 21. Suppose for simplicity that y lies on the line vs_i. Then

$$\begin{aligned} d(x,b) &= d(x,y) + d(y,a) + d(a,b) && (1) \\ &\geq d(x,y) + d(v,a) - d(v,y) + d(a,b) && (2) \\ &= d(x,y) + d(v,s_i) - d(v,y) + d(a,b) && (3) \\ &= d(x,y) + d(s_i,y) + d(a,b) && (4) \\ &\geq d(x,s_i) + d(a,b) && (5) \end{aligned}$$

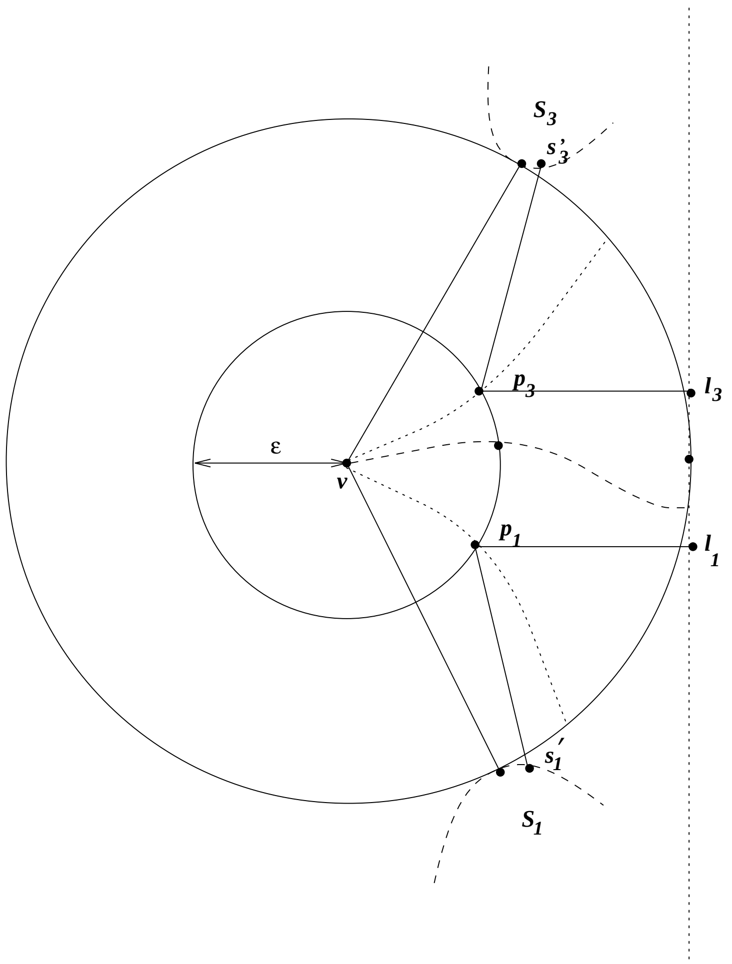
This means that x is closer to s_i than to the nearest point on the line, b, and so obviously not on the beachline. This assumes that the boundary to the right of x is a line, and not part of the circle, but since the curved boundary of the area doesn't include the right tangent, we can always find an ϵ such that the intersection of the circle of radius ϵ around v with the area is entirely to the left of one of the spokes. Thus the beachline doesn't pass though that arc, or thus though the area bounded by the spokes.

Thus both arms of the beachline pass through the secant bounded by the spokes vs_1 and vs_3, and both the separators $Sep(S_1, S_2), Sep(S_2, S_3)$ are thus to the left of the beachline. Take a circle of radius ϵ around v, where ϵ is less than the radius of the tangency circle. Suppose that this circle hits the beachline in two points, p_1 and p_3, closest to S_1 and S_3 respectively. Let the nearest points to these on their respective sites be s'_1 and s'_3 respectively. Let the nearest points on the line be l_1 and l_3 respectively. Since the beachline has exactly one point for every y value, by Lemma 7, and since it passes through p_1, v and p_3 in that order, l_1 must be below the right tangent to the circle, and l_3 above. See Figure 22. Thus p_1 is closer to S_1 than to S_2, and p_2 is closer to S_2 than S_1, so as we move a point p from one to the other along the circle of radius ϵ, staying inside the two spokes throughout, there must, by continuity of tangency points, come a time when the difference of the two distances changes from positive to negative, i.e. where the point is an equal distance from S_1 and S_2. Since the path remains between the two spokes, this point cannot be nearer to any other site, since it would have had to pass through a separator, all of which are strictly outside the spokes. Thus the separator of S_1 and S_2 passes to the right of the beachline. $\square$



,

Figure 21: The horizontal line left from x passes through y, a and b in that order



,

Figure 22: The intersection of the curve with the circle of radius ϵ must lie between the intersections with the beachline.

Lemma 9 *For any point p on any Voronoi edge, the beachline $A(t)$ passes through p for some time t.*

Proof: Let t be the x-coordinate of p, plus the distance from p to the nearest site. By definition of the beachline, p is on $A(t)$ $\square$

Theorem 1 (Correctness Proof) *Let $A(t)$ be the beachline at time t, and $B(t)$ the internal simulation of the beachline, as a list of (part-) sites controlling the beachline curves, in ascending order. If at time t $B(t)$ accurately describes $A(t)$, and $A(t)$ changes once before $t' > t$, then $B(t')$ accurately describes $A(t')$*

Proof: The beachline's representation as a collection of cusps, only changes when a cusp terminates. The creation/termination of cusps at intersection events, and left/right tangents, is obviously handled as listed in Section 3.3, with appropriate changes to the beachline. It is the function of the preprocessor to properly identify these things, and we assume that they happen accordingly. Cusps are also created at pseudo- vertices, when a start event is encountered. At this point, the reference line is passing through the right tangent to a site, and there is a line stretching to the left of point which are an equal distance from the reference line and the new site. This line reaches as far as the pseudo-vertex, previously a beachline point at that y-coordinate, and now the start of two cusps. Assuming that the beachline was correct before this point, we follow this procedure to start the new cusps correctly.

The final case where the nature of the beachline changes is at vertices. Suppose that $A(t)$, the real beachline, agrees with $B(t)$, the computed beachline, at time t, and $t' > t$ is the time of the next event for A, a vertex. We want to show that if this is a vertex event, then it will be scheduled for B, and that B has no event scheduled between t and t'. Assuming that B has no event scheduled between t and t', it follows that $A(t'') \equiv B(t'')$ for all $t < t'' < t'$. From the no-surprises lemma, the vertex at $A(t')$ corresponds to a transition $S_1S_2S_3 \rightarrow S_1S_3$. Since $A(t'') \equiv B(t'')$ for all $t < t'' < t'$, B must have a triple $S_1S_2S_3$, two adjacent cusps. Moreover, when the "most recent" of these two cusps was created, the vertex event at $A(t') = B(t')$ was detected and scheduled because the relevant conditions - the clockwise rule and no earlier event affecting their cusps - applied (the clockwise rule is true by Lemmas 5 and 2 - i.e., because it is a valid cusp, it will be clockwise, and because there is nly one possible clockwise cusp, it will be found by the algorithm).

Thus B gets updated correctly at t. Conversely, we need to show that B does not get modified by an invalid vertex at time $t'' < t'$. Suppose a vertex event at vertex v is scheduled for B at t'', and that for all $t''' < t''$, $B(t''') \equiv A(t''')$. Let $S_1S_2S_3$ be the relevant triple. Since the next event on for A is at t', this cannot be a correct event. Thus some other site s is closer to v, so there exists a small disc around v entirely contained in the Voronoi region owned by s. This implies that the segment $S_1S_2S_3$ is "spurious" being closer to s than to S_2, so it is not on $A(t''')$ for $t''' < t''$ sufficiently close to t''', so it is not on $B(t'')$ for $t''' < t''$ sufficiently close to t''. This is a contradiction, so we see that B does not change incorrectly. $\square$

This last result, together with Lemma 9 proves that the algorithm correctly simulates all changes to the beachline, and so all vertices, sites and intersections of edges with site boundaries will be represented in the output.

5 Required Primitives and run-time analysis

5.1 Algorithm requirements

The preprocessor must take the list of sites, and produce

- a) A list of all right and left tangents to the sites.
- b) A list of points where the boundaries of two sites cross, excluding those inside any other site.

Note that the total number of events thus found need not be linear in the number of sites; In the case of intersecting line segments, for example, m random lines can produce $O(m^2)$ intersections, and so $O(m^2)$ part-sites (See Figure 23). Considering each part-site as a separate site, this falls under non-degenerate data for most algorithms, which will compute the correct Voronoi diagram, together with spurious vertices on the 'interior' part of any site overlap, in $O(m^2 \log(m))$. Given this absolute restriction, we do not feel it misleading to define the variable that the speed of the algorithm will be measured in, n, as the sum of the preprocessed data and the number of sites.

For the algorithm to work, there must also be available implementations of

- c) A method of determining the position of a cusp, given the two sites, and the position of the reference line.
- d) A method of finding the two possible positions of a vertex, given the three part-sites equidistant from the vertex.
- e) A method of determining a point on the beachline, given the position of the reference line, and the y-coordinate of the point.

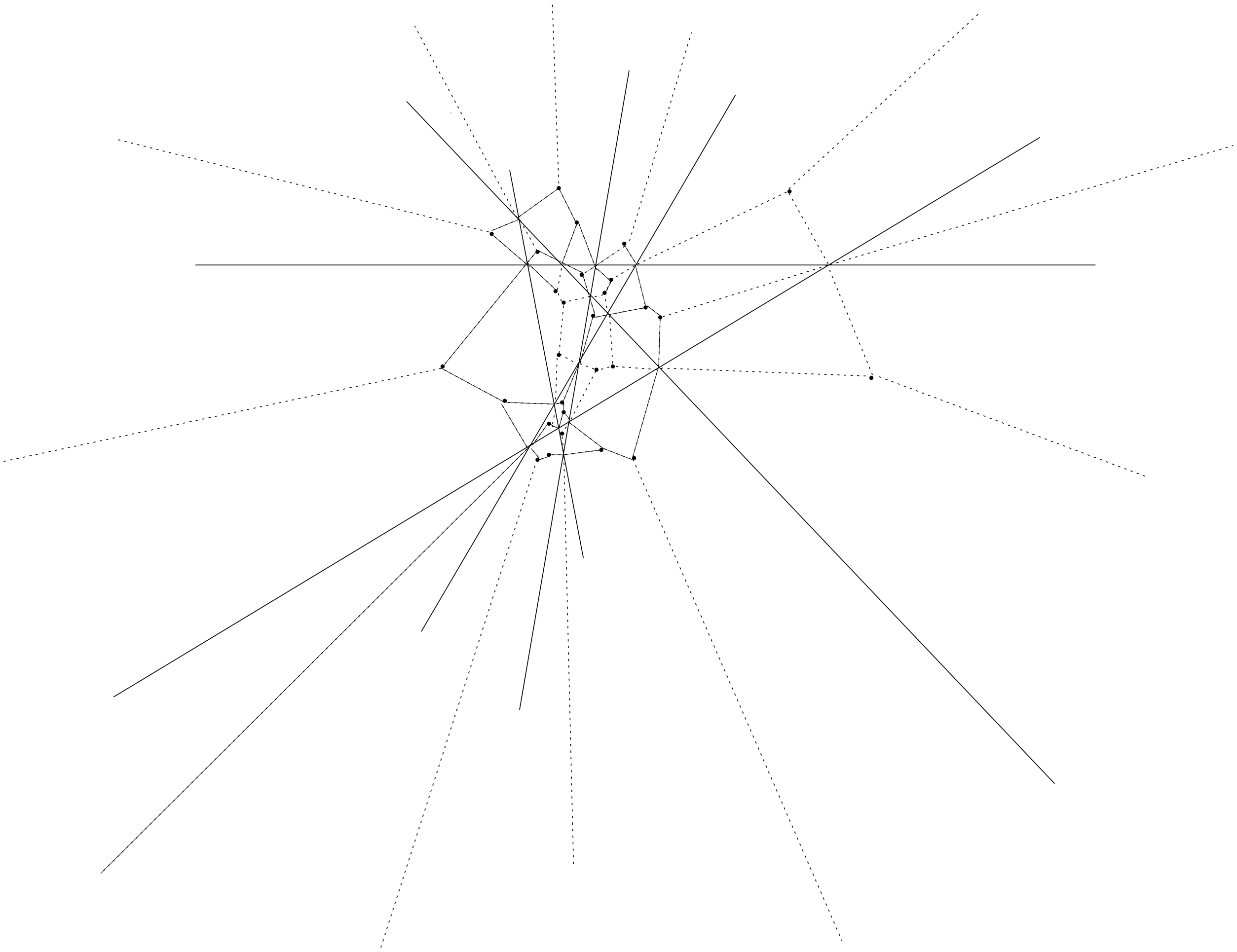
c) and d) must be constant time, and e) $O(\log n)$, where n is the number of cusps on the beachline.

5.2 Runtime analysis

The event queue is stored in a heap. The Beachline is stored in a Red-Black tree, as in Sedgewick [10].

The algorithm is roughly broken up into two sections:

- The input interpretation. Here we take the sites from input, and the list of pre-calculated intersection points, and create the event queue, This has complexity order $O(n)$ and space $O(n)$, since all the sites and events are stored after processing.
- Event queue processing. For every event on the queue, we remove it from the queue, and process it. The number of events originally on is n. The only others that can be added are vertices. If we increase the complexity of the graph by turning every overlap of sites into a collection of part-sites, we will have a graph with $O(n)$ regions, two vertices



,

Figure 23: The number of vertices for m lines can be $O(m^2)$

per edge, and at least three edges per vertex. Euler's relation gives us $< 2(n)$ vertices and $< 3(n)$ edges, some of which will be inside a site overlap, and thus spurious. Thus the number of iterations is $\mathrm{O}(n)$. Removal from the event queue is $\mathrm{O}(\log(n)$, since the queue is stored as a heap. At any time, the beachline will stored as its (maximum of two) intersections with a number of edges, together with its (unique) intersection with a number of sites, and thus is order $\mathrm{O}(n)$. The processor's actions on each event involve a deletion, an insertion and/or a search of the beachline, each of which are $\mathrm{O}(\log(n)$.

The total time involved is thus $\mathrm{O}((n) \log (n))$.

6 Implementation notes for Circles program

6.1 Input and output descriptions

It is possible, by adding an appropriate line (SHOW_BLN time) in the input, to cause the program to list the exact physical position of the beachline at any given time, in a format such that a post-processor can render it. This passes through into the event queue, and its processing makes no changes to any of the data structures.

The format for input is: (See Figure 24.)

<Number of sites>

CIRCLE < circle centre > <circle radius> <circle index>

...

<event descriptor> <information>

...

In the case of the event descriptor being SHOW_BLN, the only further information the input function looks for is a time. For all other events, the information format is:

<event position> <first site index> <second site index> <previous upper event index> <previous lower event index>

For START and END events, the first site is the site that the events are on, and the second is ignored. For A_CUSP and V_CUSP events, the first site is the controller of the boundary event that is about to be deleted, and the second the controller of the one about to created. For L_CUSP and R_CUSP events, the first site is the controller of the lower of the two boundary cusps, and the second site the controller of the higher.

The previous events are differentiated by whether they lie on the upper or lower halves of the circle. In the case of generalised convex sets, the START and UPPER events will obviously remain, identifiable as the only places that a horizontal line is tangent to the set, and so the division of the object into upper and lower halves is still viable. Obviously, only END and R_CUSP events will ever have two previous events.

The format for output is: (See Figure 25)

{CIRCLE <centre> <radius> <index>}$^+$

{<edge_event_type> <start event> <end event> <controlling sites> |

SHOW_BLN <number of curves on beachline> <time> <index of first site> <position of cusp between second and first> <index of second site> ... <index of last site> |

<other_event_type> <event position> <event index>}$^+$

```
3
CIRCLE 1.000000 0.000000 0.750000 0
CIRCLE 1.000000 1.000000 0.450000 1
CIRCLE 2.000000 2.000000 0.150000 2
LEFT_TANGENT 0.250000 0.000000 0 0 0 0 1
LEFT_TANGENT 0.550000 1.000000 1 1 0 0 2
RIGHT_CUSP 0.683614 0.680000 0 1 1 2 3
LEFT_CUSP 1.316386 0.680000 0 1 0 0 4
RIGHT_TANGENT 1.450000 1.000000 1 1 2 4 5
RIGHT_TANGENT 1.750000 0.000000 0 0 4 1 6
LEFT_TANGENT 1.850000 2.000000 2 2 0 0 7
RIGHT_TANGENT 2.150000 2.000000 2 2 7 7 8
SHOW_BLN 2
0
```

Figure 24: Sample input format

EXTENSION <event on non-infinite end of edge> <position at final time> <events that control the edge>}$^+$

The event types are as listed for input, with the addition of PSEUDO_VERTEX. The edge event types are SEMI_EDGE (starts in a pseudo-vertex) or EDGE (starts in a boundary event, or another vertex).

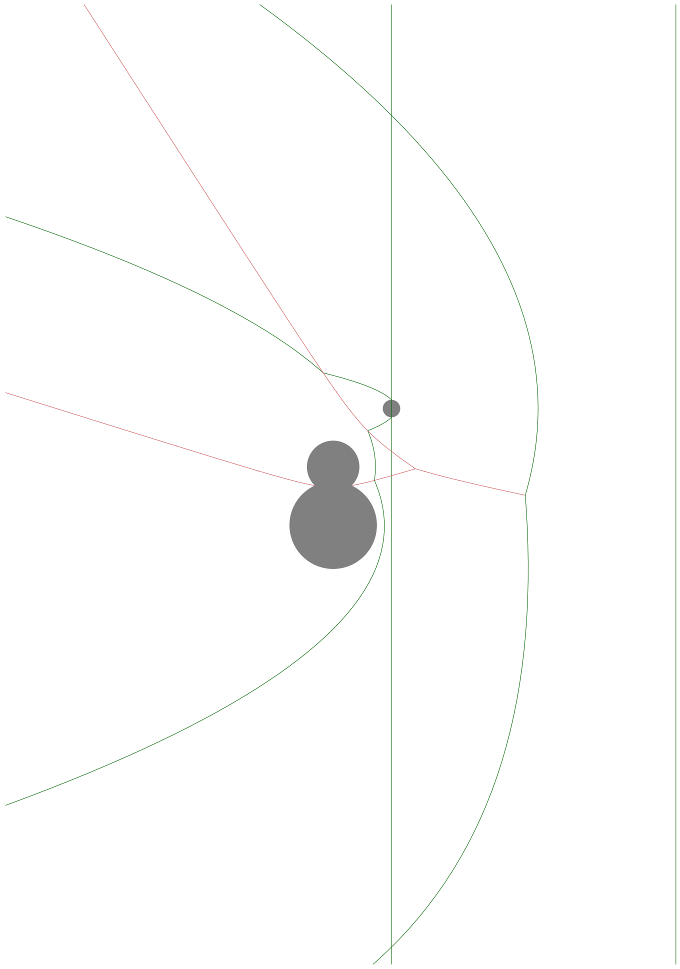
6.2 Apollonius problem

For the case where the sites are circles, the problem of finding the point(s) equidistant from three sites becomes the Problem of Apollonius. In the general case, we use the Solution of Gergonne, as described in Dörrie [3] We first find the power centre of the three circles, and then the three External Similarity Points, where the tangents for any two circles coincide. These three points lie on the same line, called the Similarity axis. We then calculate the pole of each circle in this line, that is, the point where the ratio of the distance from the centre to the radius is the same as the ratio of the radius to the distance from the centre to the line. For each circle, we draw the line from the pole inside it to the power centre, and note the two points of intersection with the circle, one nearer the power centre and one further. The three nearer and three further intersection points each define a circle tangent to the three circles, and thus with centre a potential vertex. We also check that each tangent circle doesn't include any of the actual circles, and vice versa, by checking that line between the centre of the tangent circle and any of the centres of the actual circles passes through the relevant tangent point. See Figure 27. In this implementation, we then check the tangent points on the circles, to determine whether they travel around the centre in clockwise order, and discard any that do not. The implementation uses homogenous coordinates to deal well with the case where two of the circles have the same radius, and their external similarity point is at infinity.

The first special case is where the three radii are identical. Here, we simply take the

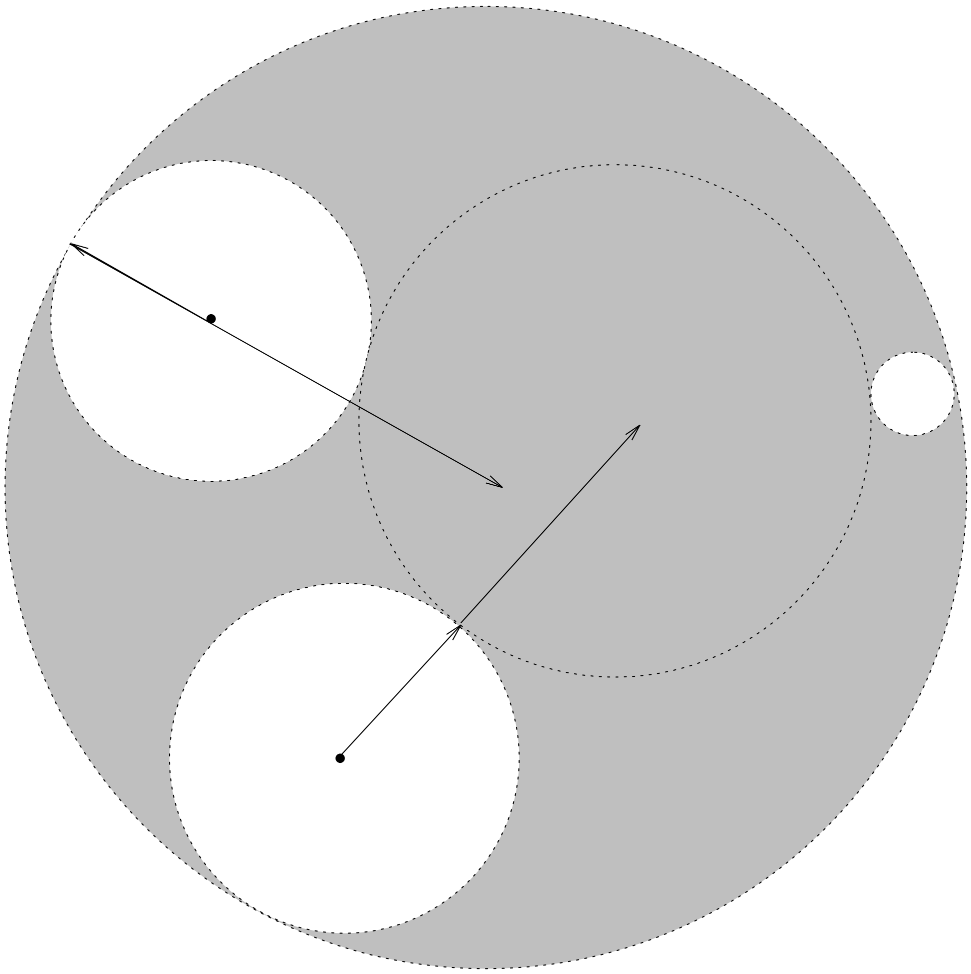
```
1
CIRCLE 1 0  0.75 1
CIRCLE 1 1  0.45 2
CIRCLE 2 2  0.15 3
START 0.25 0  4
START 0.55 1  5
PSEUDO_VERTEX -0.516667 1  6
R 0.683614 0.68  7
SEMI_EDGE 6 7 1 2
L 1.31639 0.68  8
END 1.45 1 9
END 1.75 0 10
START 1.85 2  11
PSEUDO_VERTEX 1.26538 2  12
SHOW_BLN 7 2
1 1.70645 0.768065  2 1.59262 1.61959  3 2 1.85  -1
2 2.15  3 0.829861 2.61118  2 -32.062 10.8986  1
END 2.15 2 13
VERTEX 2.41 0.968108  14
SEMI_EDGE 8 14 1 2
SEMI_EDGE 12 14 2 3
EXTENSION 6 -133.111 42.6761  2 1
EXTENSION 12 -5.16032 11.8613  3 2
EXTENSION 14 4.29332 0.513263  1 3
SHOW_BLN 4 6.8764
1 4.29332 0.513263  3 -5.16032 11.8613  2 -133.111 42.6761  1
```

Figure 25: Sample output format



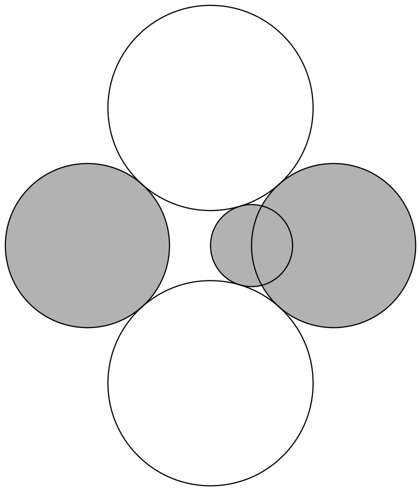
,

Figure 26: Sample Voronoi diagram



,

Figure 27: A change of direction of the line between the centre of the circle, the tangent point, and the centre of the circumcircle indicates that the circumcircle includes the given circle, or vice versa.



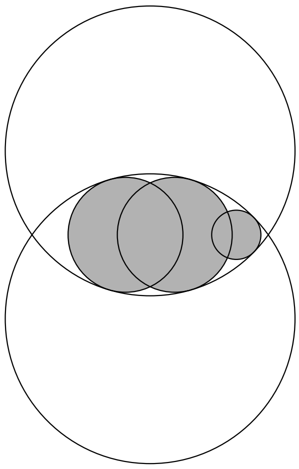
,

Figure 28: In the case where the centres of the circles are collinear, we translate the circles to the x-axis

circumcentre of the the three circles, subtract their radius from the distance between it and any of the centres, and check that the centres are in clockwise order around it.

The second special case occurs when the centre of the three circles are collinear. In this case the power centre of the circles is at infinity, and a new method is needed. We translate and rotate the circles so that the smallest is at the origin, and the vector from the first to the second is pointing along the positive x-axis. We then calculate the circumcentre of the origin, and the two circles centred at the centres of the other two circles, and with radius reduced by the radius of the circles previously at the origin. Due to the transformation of the circles, the clockwise relation simplifies to removing the upper circle, except in the case where the third circle is the smallest. All the circles are then untransformed. See Figures 28 and 29.

A third special case, where one or more of the circles is of zero radius, can be handled within the code. If all three are points, then this is handled naturally by the first special case. The code for the second special case, relying as it does on a conceptual reduction of the smallest circle to a point, also runs perfectly. In the general case, the points used to determine the similarity axis are the external similarity points of the first and second circle, and of the second and third circle. If second is the only point, these will be the same point, and errors



,

Figure 29: Since the circles are symmetric about the (untransformed) x-axis, either both circumcircles enclose the given circles, or neither do.

will naturally occur. If second is the only non-point of the three, then the two similarity points will be the two others, and the similarity axis will be correct. Thus a sufficient safeguard is to ensure that second is not a point. Thus, we check both the first and the third, and if either is a point, then they are 'rotated in' (i.e. $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_1$ would rotate the first circle in). This rotation ensures that the clockwise/anti-clockwise order of tangent points around the potential vertex remains the same.

7 Remarks on performance

7.1 Speed

The implementation, running on a Sun SPARCclassic, processed 500 circles of radius .5-1 inside a 10x10 square in 13 seconds. See Figure 30.

For 1000 circles, radius 0-.25 inside 10x10 square, the time taken was 25 seconds. See Figure 31.

7.2 Degeneracies

By checking for exceptional circumstances at crucial points in the algorithm, this implementation is also exceptionally robust, and can handle data traditionally considered as degenerate. This include:

- Zero - radius circles (points). See Figure 32
- Cocircular points. See Figure 33
- Cocircular circles. See Figure 34
- Collinear points. See Figure 35
- Vertical collinear points (the cusp has an x coordinate of negative infinity). See Figure 36

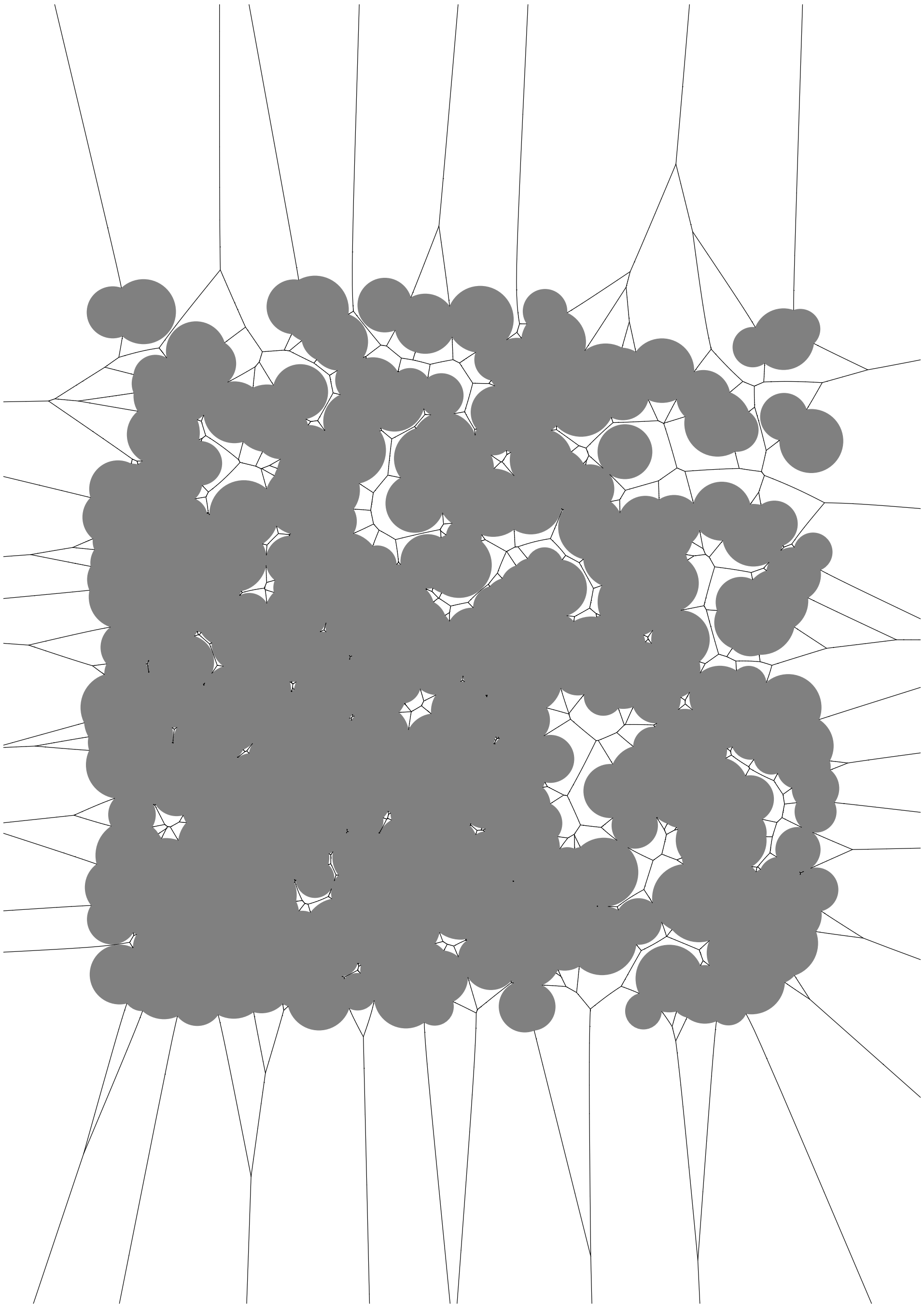
References

[1] F. Aurenhammer (1990). Voronoi diagrams - a survey of a fundamental geometric data structure. Technical report B 90-09, Fachbereich Mathematik, Freie Univrsität Berlin.

[2] R. Cole, 1987, Private Communication, quoted by Colm Ó Dúnlaing.

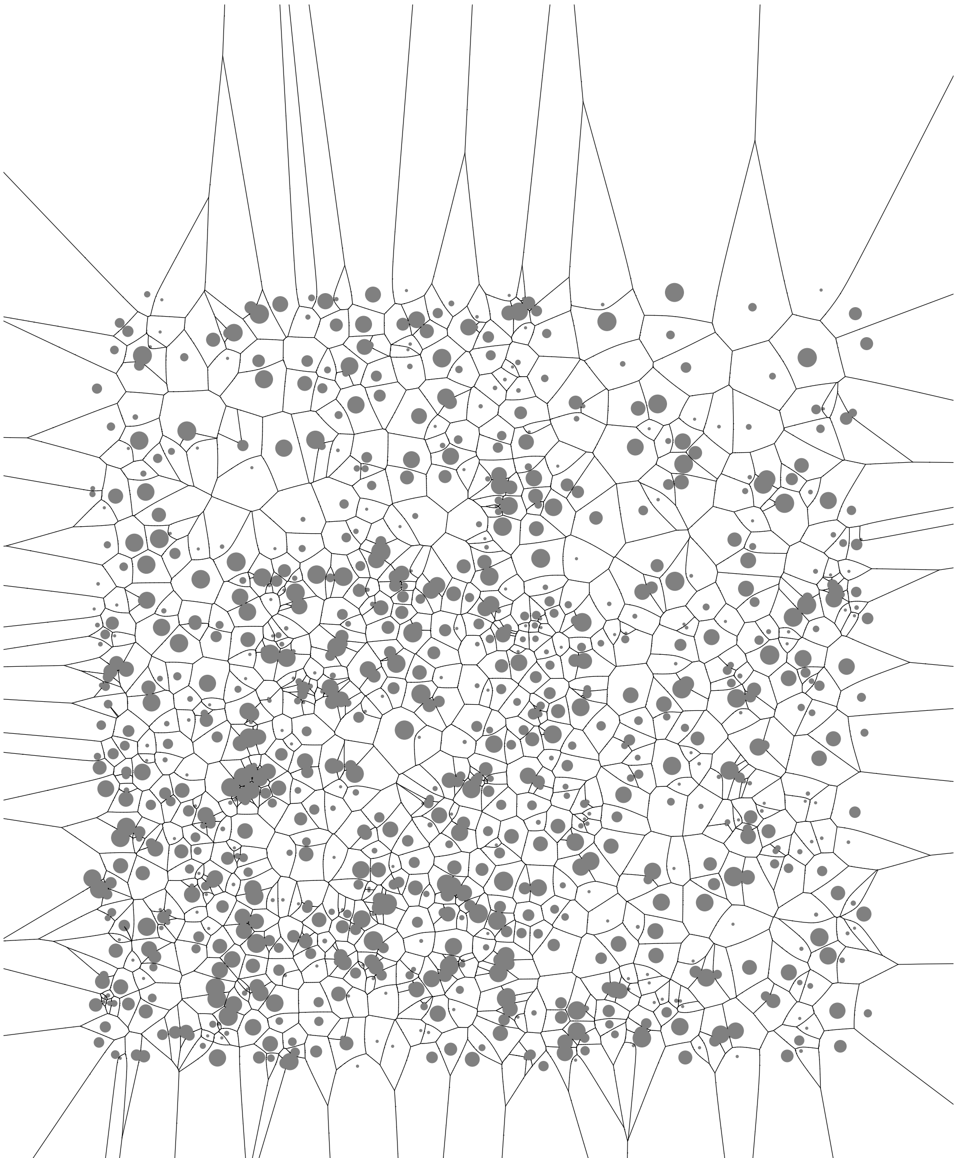
[3] H. Dörrie. *100 great problems of elementary mathematics* (tr. D. Antin). Dover Publications, New York, 1965.

[4] S. Fortune (1987). A sweepline algorithm for Voronoi diagrams. *Algorithmica* 2, 153-174.



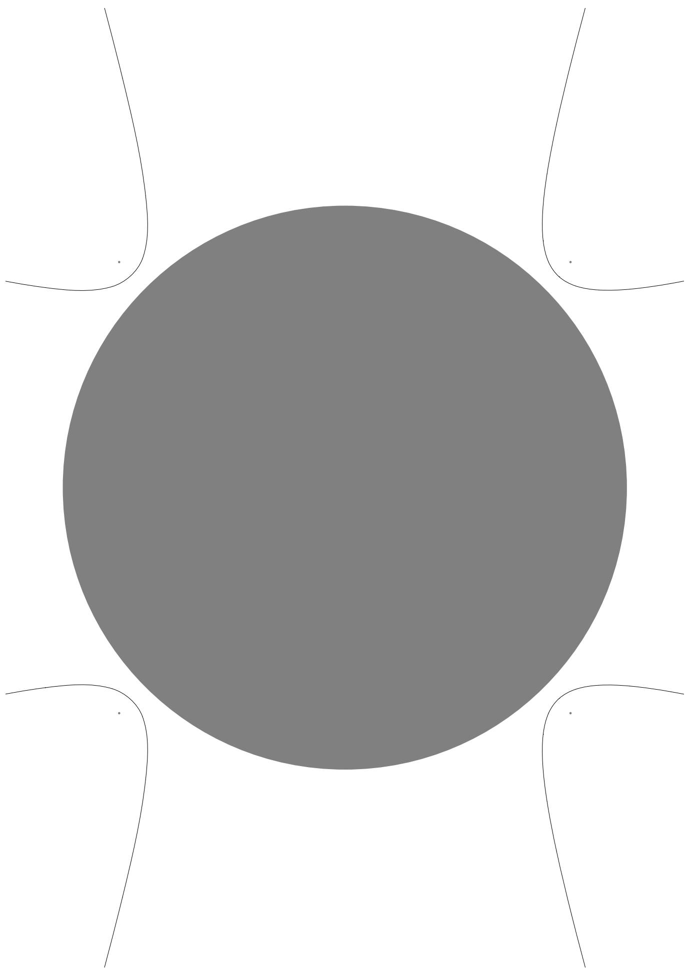
,

Figure 30: 500 circles, of radius .5 - 1, inside a square of side 10



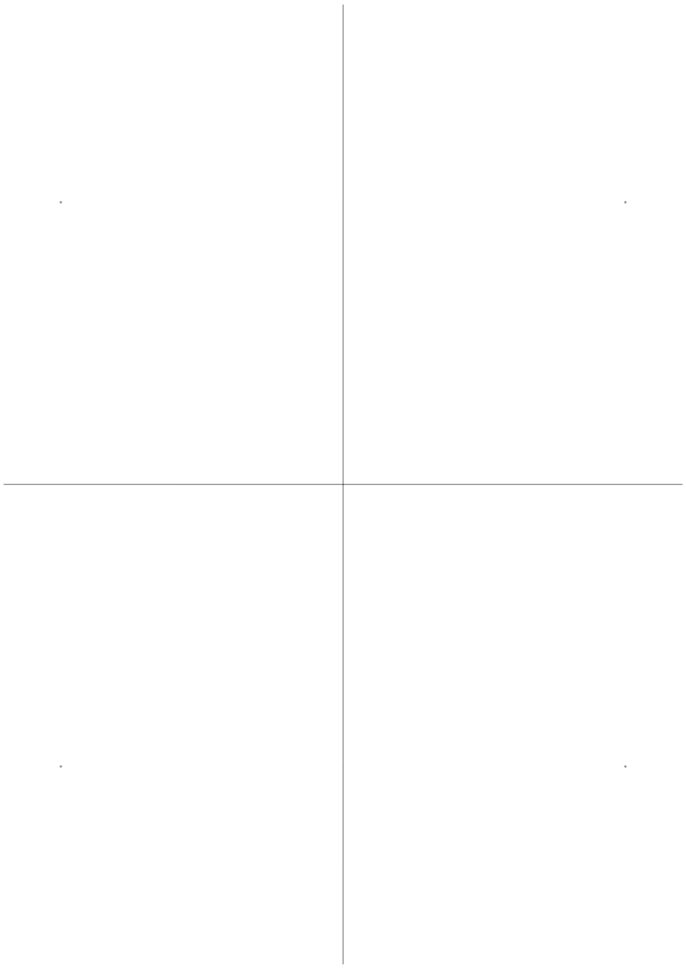
,

Figure 31: 1000 circles, of radius 0 - .25, inside a square of side 10



,

Figure 32: Points and circles



,

Figure 33: 4 co-circular points

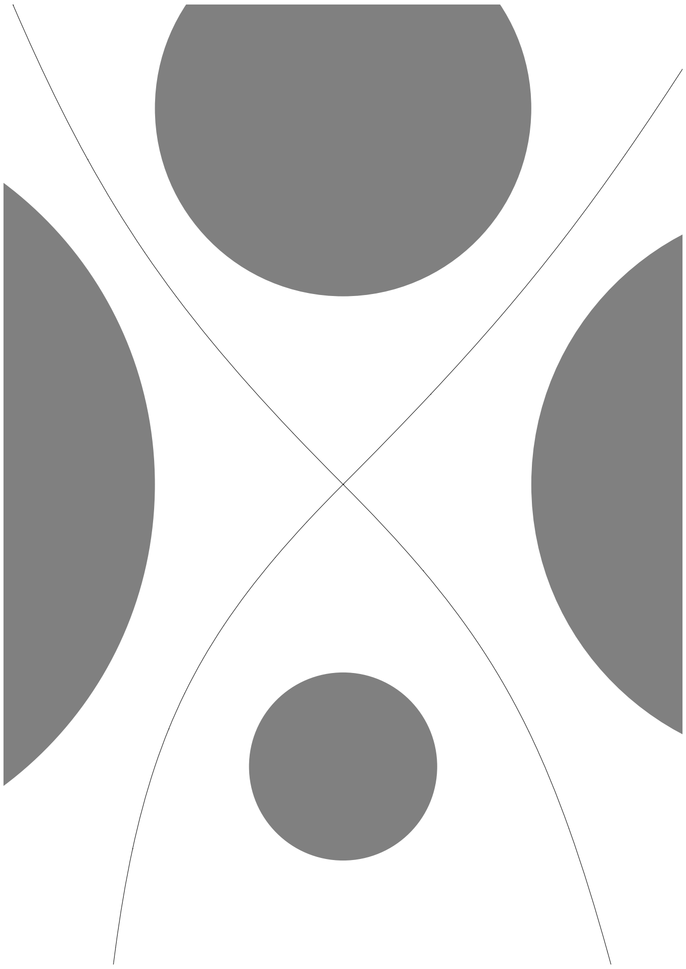


Figure 34: 4 co-circular circles

Figure 35: Three collinear points

,

Figure 36: Three collinear points, vertical

[5] M.T. Goodrich, C. Ó Dúnlaing and C.K. Yap (1989). Constructing the Voronoi Diagram of a set of line segments in parallel. Lecture Notes in Computer Science, Vol. 382 . Springer-Verlag, New York, pp. 12-23.

[6] D. Leven and M. Sharir Intersection and proximity problems and Voronoi diagrams. *Advances in Robotics* Vol 1 (J.T. Schwarz and C.K Yap, Eds.) Lawrence Erlbaum Publishers, Hillsdale, NJ, 1987

[7] C. Ó Dúnlaing, M. Sharir and C. K. Yap (1986). Generalized Voronoi Diagrams for moving a ladder. I: Topological analysis. *Communications on Pure and Applied Mathematics* Vol 39 423-483.

[8] C. Ó Dúnlaing, M. Sharir and C. K. Yap (1987). Generalized Voronoi Diagrams for moving a ladder. II: Efficient construction of the diagram. *Algorithmica* 2, 27-61.

[9] C. Ó Dúnlaing (1991) Fortune's Voronoi sweepline algorithm for convex sites, Proc IMACS'91 Conference, Dublin 133-134.

[10] R. Sedgewick, *Algorithms*(1988) 2nd Edition Addison-Wesley 219-229

[11] M. Shamos and D. Hoey (1975). Closest point problems. *Proc 10th IEEE Symposium on fundamentals of computer science*, 151-162.

[12] C. K. Yap (1987). An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments, *Discrete & Computational Geometry* 2, 365-394.

8 Appendix: The header file listing

This is a funtional desription of the program, consisting of a header file (the file that keeps a central list of all commands on all objects, so that any of the several modular parts of the program can know which functions it can and cannot use), commented appropriately. The header file is organised by lists of classes of objects, followed by a list of attributes, and private (for internal use of this object type) and public (for use by any other objects) functions. There are several routines which have been declared as functions to save duplication of purpose, or to aid the readability of the program. These are generally local to each modular file, and do not appear in the header file if there is no obvius object that they 'apply' themselves to. We have included those that do not act on specific objects at the end of the header file listing, for completeness.

```
/***********************
        circles.h
        Header file for 2d circle-sites Voronoi diagram program.
        Begun 03 May 1993.
***********************/
#include <fstream.h>
#define TRUE 1
#define FALSE 0
#define BOOLEAN int
#define INFINITY (double) 9999999
typedef enum { START, L, R, A, V, END, VERTEX, PSEUDO_VERTEX, SHOW_BLN }
          EVENT_TYPE;

char * event_type ( EVENT_TYPE t );
    /* This maps event to their corresponding names, for output  purposes */

class point
{
private:
      double my_x, my_y;
public:
      double xcoord(), ycoord();
      point ( double a = (double) -INFINITY , double b  = (double) -INFINITY);
      friend ostream& operator<<( ostream& s, class point & p );
      friend istream& operator>>( istream& s, class point & p );
      /* Standard access, creation, input and output operations*/
};

class circle
{
private:
      point my_centre;
      double my_radius;
public:
      point centre();
      double radius();
      circle ( point centre, double radius= -INFINITY);
      circle ();
      friend ostream& operator<<(ostream& s, class circle & p);
      friend istream& operator>>(istream& s, class circle & p);
      /* Standard access, creation, input and output operations*/
};
```

```
class EVENT
/* An event corresponds to each point where the beachline changes, or a point
that needs to be output for the post-processor */
{
private:
      EVENT_TYPE my_kind;
      /* The event types are: START and END (start and end of circles),
VERTEX,(for Voronoi vertices) L, R, A and V (different types of intersection-
A indicates a change of a lower boundary cusp from one site to another, V
indicates the same for a upper boundary cusp. L indicates the creation of two
boundary cusps, and R indicates the ending of two) PSEUDO_VERTEX (temporary
point created at the insertion of a new site, to indicate where the two new
cusps start from.) and finally SHOW_BLN (to indicate that we should output
the state of the beachline at the given time. This is the only event that
doesn't represent a point */

      int my_index;
      /* Used for output. Whenever any event except a SHOW_BLN is dealt with,
its position is output, and my_index is set to one more than the previous
event. This variable is used by later events to refer back to it in output. */

      class EQ_ENTRY * my_qplace;
        /* A link to the event's place in the event queue. Set when it's
        inserted, used for it's deletion */

        double my_time;
        /* The variable used to index the event in the event queue. For most
events, this is the x-value of the associated point. The exceptions are
VERTEXs, which are indexed by their x-value plus the distance from them to
any of the sites that define them, and SHOW_BLN's which are indexed by their
given time. */

        class blnNODE * my_bln_entry;
        /* Link indicating which beachline entry leads into the current event,
so that the numerical method of determining position, greatest_lb, is used as
little as possible, particularly at events where two cusps come together. For
A and V, this indicates the incoming boundary cusp. For R and END, this
indicates the lower of the incoming boundary cusps, and for VERTEX, this
indicates the lower of the incoming cusps. For the boundary events, this is
set by set_next_upper_event/set_next_lower_event, after the previous boundary
event was handled, due to this information being available from the
preprocessor. For VERTEXs, this is set when the vertex is calculated, by
set_finish on a cusp. The my_bln_entry for the VERTEX is automatically set
to the cusp in question, since the VERTEX would then be the intersection of
the cusp and it's (higher) successor. */
```

```
/* class EVENT continued */
        point my_vertex;
        /* The point associated with the event, if any */

        class SITE *my_first_site;
        class SITE *my_second_site;
       /* For boundary events, indicates the site(s) the event is associated
with. For START and END, my_first_site indicates the relevant site. For A and
V, the convention is that my_first_site is the site that was attached to the
boundary cusp, and my_second_site the site that will be attached to the new
one. For L, my_first_site controls the lower of the new cusps, and
my_second_site the higher. */

    class EVENT *next_upper_event;
    class EVENT *next_lower_event;
    /* For boundary events, the next event on either the upper or lower half
of the relevant circle. Obviously, two are necessary because START engenders
two next events, both of which may have to be informed of the nodes coming
their way. Set from the information given by the preprocessor; It supplies
for each event the previous event on the upper or lower half, and get_input
thus informs the specified event that the current event is next. */

friend class EVENT_QUEUE;

public:
    EVENT_TYPE kind();
    double time();
    /* Standard access functions */

    EVENT (  EVENT_TYPE this_kind, double when =  INFINITY ,
          class SITE * at_first_site = NULL ,
          class SITE * at_second_site = NULL );

      /* Creation routine, for use in get_input, where place in beachline is
not known yet, before events are put on Event_queue */

      EVENT (  EVENT_TYPE this_kind, double when =  INFINITY ,
            class blnNODE *at_bln = NULL );
      /* Creation routine for inside Process, where most created events are to
       be scheduled immediately */
```

```
/* class EVENT continued */

      int index();
      class SITE * first_site();
      class SITE * second_site();

      /* Standard access functions */

      class blnNODE * beachline_entry();
      void record ( ostream& output );
      /* This records the event on the supplied output stream. In the case of
SHOW_BLN, we call the Beachline::Print function, which lists the geometric
state of the beachline. In all the other cases, we increment the
Output_index, a guide to whatever interprets this, and assign my_index of the
event to it, so that future references refer to it consistently. We then
adjust the limits of the view frame, if necessary, to accommodate the event.
We then output the Event type, the position of the event, and it's new index.
In the case where a cusp ends at the event (e.g. VERTEX or R), we announce an
edge, if the previous event on that edge was a vertex, or semi-edge
otherwise, and output the event indices of the previous event, the current
event, and the two sites that determined the ending cusp. This is sufficient
information to draw the edge. */

      void handle ( class EVENT_QUEUE & event_queue, ostream& output );
      /* The main processing function. The behaviour differs for all event
types:
      START: We first record the event, create a pseudo-vertex and record
that. Then we find the highest cusp below the event's y-coordinate, and
find the higher site for the cusp. We then insert the cusp between this
site and the events sites, the two boundary cusps and the cusp between the
events sites and the found site. We then set the start events for the first
and fourth cusps to be the pseudo events and check if we can foresee where
those cusps will end. Finally, we set the next upper and lower events for
the upper and lower boundary cusps respectively

      END: We record the events, delete the associated cusp and its successor,
and check if the previous cusp now has a new intended finish.

      R: Record the event, delete the corresponding cusp and its two
successors.
```

```
/* class EVENT continued */

      VERTEX: If the finish event for the associated node has already
happened, do nothing. Otherwise, we record the event, delete the associated
node and its successor, and create a new node with the appropriate sites.
We then set the start of the new node to this event, and, if necessary, set
the anticipated finishes for the new node and its predecessor.

      SHOW_BLN: Records the event, which will display the beachline.

      A: Records the event, checks that the associated node is a lower
boundary cusp. We then remove the cusp and replace it with a cusp between
the old and new sites, and a boundary cusp on the lower half of the new site.
We check if the cusp below the initial cusp needs to have its projected
finish reset. We set the next lower event for the new lower boundary cusp
if necessary, and set the start of the between-sites cusp to the event.

      V: Records the event, checks that the associated node is a upper
boundary cusp. We then remove the cusp and replace it with a boundary cusp on
the upper half of the new site, and a cusp between the old and new sites.
We check if the cusp between sites needs to have its projected finish reset.
We set the next upper event for the new upper boundary cusp if necessary, and
set the start of the between-sites cusp to the event.

      L: We record the event, and then find the highest cusp below the events
y-coordinate, which we assume to be a lower boundary cusp. We then insert
immediately afterwards a upper boundary cusp for site1, a cusp between site1
and site2, and a lower boundary cusp for site2, where site1 and site2 are,
respectively, the events lower and upper sites. We set the start of the
middle cusp to be this event, and set the next upper/lower events for the
boundary events, as necessary */

      void schedule_me ( class EVENT_QUEUE & event_queue );
      /* Puts event on the event queue */

      void set_vertex (point vertex);
      /* Standard assignment function */
```

```
/* class EVENT continued */

      void delete_me ( class EVENT_QUEUE & event_queue );
      /* Deletes item from the event queue */

    void set_next_upper_event(EVENT*eventptr);
    void set_next_lower_event(EVENT*eventptr);
    void set_bln_entry(blnNODE * nodeptr);
    /* Standard assignment function */

    void pass_to_next_upper(blnNODE *nodeptr);
    void pass_to_next_lower(blnNODE *nodeptr);
    /* If the boundary cusp of nodeptr has either started or changed circles,
by passing through an event, the next upper(lower) event may need to know
which beachline node is incoming. These routines determine if it does, and
if so, calls the appropriate function to set the next upper (lower) event*/

}; // end class EVENT


class EQ_ENTRY
{
private:
      int my_place; class EVENT * event;
      /* Just an indicator of where on the heap the Event queue entry is,
         and what event is carries */
friend class EVENT_QUEUE;
friend int compare(EQ_ENTRY * one,EQ_ENTRY * two);
public:
      EQ_ENTRY ( class EVENT * thisevent );
      /* Standard creation function */
      inline double key ();
      /* Returns the time of the associated event */
};
```

```
class SITE
/* The general sites around which the diagrams will be drawn. */
{
private:
      circle * my_centre;
      /* The actual site */

      int my_index;
      /* An index assigned when they are read in, and repeated when they are
output, so that the output can refer to them coherently */
friend void EVENT::record (ostream& output);
public:
      SITE (circle c,int index)
      circle *centre();
      int index()
     friend ostream& operator<<( ostream& s, class SITE & p );
      /* Standard creation, access and output functions */
};

class EVENT_QUEUE
{
private:
      int count;
      /* Current number of items on the heap */

      int maxcount;
      /* Maximum number of items on the heap */

      class EQ_ENTRY * *item;
      /* The heap */
      void sift_up( int from );
      void sift_down( int from );
      /* Standard heap operations */
public:
      EVENT_QUEUE ( int max = 0 );
      /* Standard creation function */

      void del_item ( class EQ_ENTRY * thisitem );
      /* Standard heap operation to delete a given queue entry */
      class EQ_ENTRY * insert ( class EVENT * thisevent );
      /* Standard heap operation to create a EQ_ENTRY corresponding to
thisevent and insert it into the event queue */
```

```
/* class EVENT_QUEUE continued */

      class EVENT * del_min();
      /* Standard heap operation to delete and return the minimum element */

      int empty();
      /* Returns TRUE if count = 0, FALSE otherwise */

      int size();
      /* Returns count */

      void Print(ostream& output);
};


typedef enum { LOW, EQUAL, HIGH } A_to_B;
      // a 3-way result for comparisons.  To avoid recomputation.


class CUSP
/* A cusp is just a description of the moving point on a Voronoi edge, as
indexed by two sites, and physically determined by their orientation and
the position of the moving reference line. */

{
friend class BEACHLINE;
private:
      class SITE * my_lo_site, * my_hi_site;
public:
      CUSP ( class SITE *bel = NULL, class SITE *abo = NULL );
      class SITE * lo_site();
      class SITE * hi_site();
```

```
/* class CUSP continued */
      /* Standard access and creation operations*/

      point fix_at ( double t );
      /* Given a cusp and a reference time, find the point corresponding
to the cusp at this time. For circles, the case arises where the cusp
is an upper(lower) boundary to a circle, which would happen if the
reference line passed through the circle. These are denoted by the lower
(upper) boundaries being NULL. In this case, x = t, and calculate y
accordingly. The default case is simply the calculation of the intersection
of the beachline segments. In the case of circles with the same radius
(including points), the segments are parabolas with the same directrix;
In the case of different radii, we get the slightly more complex problem of
parabolas with different directrices. Other cases, particularly lines, may
require more complex study. Most cases will have two solutions, representing
the correct cusp (lo,hi) and the bogus cusp (hi,lo). These two cusps can be
traced back (even if they didn't originate from) the point where they were
one, directly to the left of the later of the two (i.e. the site that was
encountered second). Directly after this time, the two cusps were (earlier,
later) followed by (later, earlier). Since the cusps are labeled in
ascending order, this means that (earlier, later) was lower that (later,
earlier). This remains an invariant, since a crossing would indicate a
collapsing of the two cusps into one, and this would mean a vertex with
two of the three sites being identical. This would imply that the Voronoi
region belonging to the earlier site encloses the region belong to the later,
contradicting the star-shaped nature of Voronoi regions. In the case of the
sites having identical minimum x-coordinate, the only available point will be
accurate if the nearest point to it on the 'lower' site is actually lower
than the nearest point on the 'higher'. */

      A_to_B compare ( double b, double t );
      /* Given the current time and a y coordinate, determine whether the cusp
is higher, lower, or on the line y=b */
};
```

```
class blnNODE
{
      // beachline node.  Follows red-black tree organisation.
      /* Basically represents a cusp, with information about where it came
from, where it is, and where it's going */
friend class BEACHLINE;
public:
      class EVENT * start();
      class EVENT * finish();
      class CUSP cusp();
      blnNODE( class SITE * lo_site, class SITE * hi_site );
      /* Standard access and creation operations*/

      void set_start(EVENT * this_event);
      void set_finish(EVENT * this_event);
      /* Standard assignment operations */
private:
      class blnNODE *parent, *left, *right
      BOOLEAN black;
      /* Standard attributes for a node in red-black binary tree */

      class CUSP my_cusp;
      /* The cusp this node represents */
      class EVENT * my_start, * my_finish;
      /* The event at which this cusp started (either a pseudo-vertex or a
vertex) and the event at which it is scheduled to end (if any) */

};
```

```
class BEACHLINE
/* The locus of all points to the left of the current reference line that are
an equal distance from the nearest site to the line. Obviously, this will
maintain a simple shape (e.g., a parabola) within a Voronoi region, and will
change only at cusps, which must lie on the Voronoi edges. It is represented
by a collection of cusps, which are specified as pairs of sites, and are only
given actual positions when the need calls. The beachline is kept up to date
by the accurate scheduling of all events which will change it. The start of a
new site, for example, will create two new cusps. Also two cusps may
disappear into one at a Voronoi vertex, which is scheduled for a time as far
to the right of the vertex as the distance of the vertex from any of its
points. The cusps will come together at this time, and any site inside the
empty circle corresponding to the vertex would have disrupted the cusps, and
canceled the vertex before then. The beachline cusps are stored as nodes in
a red-black binary tree, which are ordered in ascending value of y- coordinate
when listed inorder. */

{
public:
      BEACHLINE();
      /* Standard creation operation */

      void initialise ( class SITE * the_site );
      /* Initialises the beachline with just one site */

      void node_delete ( class blnNODE * node )
      /* Standard binary tree procedure to delete a given node from the
structure */

      class blnNODE * succ ( class blnNODE * node );
      class blnNODE * pred ( class blnNODE * node );
      /* Standard binary tree procedure to find, given any node, its
successor (predecessor) in a tree sorted by inorder (i.e., the node
representing the next highest (lowest) cusp) */

      class blnNODE * first(), * last();
      /* Standard binary tree routines to determine the first and last (i.e.
lowest and highest) nodes in the beachline */

      class blnNODE * insert_after ( class blnNODE * node,
                        class SITE * lo_site, class SITE * hi_site );
      /* Creates a new node with specified low and high sites, and inserts it
immediately after the given node */
```

```
/* class BEACHILINE continued */

      void checkout ( ostream & output = cerr );
      /* Error checking procedure, which determines whether the basic
invariant of red-black trees is intact */

      double x_val ( double y, double t );
      /* This is used after a site is introduced, and (t,y) is the leftmost
point of it. We use greatest_lb to anticipate the new cusps, which currently
have the same position, directly left of (t,y). We thus return the
corresponding x-value. */

      class blnNODE * greatest_lb ( double y, double t );
      /* Given a y-coordinate y and the time t, finds the beachline node
corresponding to the highest cusp below y by descending the tree, turning
left or right according to compare applied to the current nodes cusp */

      double time_to_meet_succ ( class blnNODE * node );
      /* Returns the time that the given node meets its successor in the
beachline, or -INFINITY if it doesn't. We first check that neither the node
nor its successor is NULL, that the node's lower site is not equal to its
successors higher site, and that neither of them are on the boundary of
a site. Any of these also causes us to return -INFINITY. We then extract the
three sites, and call incentre, which returns a circle tangent to the three
sites. After checking that the circle has positive radius (the expected
return value if there exists no such circle, i.e. the cusps do not converge)
we return the x-value of the rightmost point of the circle, at which point
the two cusps will have converged to a Voronoi vertex. */

      void set_finish ( class EVENT_QUEUE & event_queue,
                  class blnNODE * this_node, double from_now, ostream& output);
      /* Checks to see if this_node needs to change it's set_finish variable,
which indicates what vertex, if any, this_node will end at. This routine is
called whenever a new cusp is created. We first find the successor of
this_node, and check that it's neither NULL nor a boundary point. We then
call time_to_meet_succ, to determine if and when the node is scheduled to
hit a vertex. If the time is before the current time from_now, we do nothing,
since the vertex either has already happened, or will never happen, due to a
fourth site intruding. If either this_node's successor or predecessor are
scheduled to hit a vertex before hand, we do nothing, since this would change
the nature of either this_node or this_node's successor, and in either case,
set_finish will be called again on this_node. If we get through this, we
delete (if they exist) the previously expected finishes for this_node and its
```

```
successor, create a new vertex event at the intersection point, and put it on
the event queue, to be processed at the time returned by time_to_meet_succ,
i.e. the x-value of the vertex, plus the distance from the vertex to one of
the tangent points on the three surrounding points. This increment ensures
that the vertex will only be processed if the circle it defines is indeed
empty, and is related to the offset function from Fortune's 1987 paper. */

      void Print ( double t, ostream & output );
      /* Procedure to print out the physical positions of the cusps, together
with the indices of the sites in between, so that a beachline can be drawn,
if required. */
      void Print1 ( ostream & output );
      /* Debugging procedure, which prints out the indices of the lower and
higher sites for each cusp, so that we can tell if one of the basic
invariants (each node's high-site is the same as it's successors low-site)*/

      class SITE * earliest_site();
      /* Standard access operation */

      int size();
      /* Counts the number of nodes on the beachline */

private:
      class blnNODE *root;
      /* The root of the tree */

      int rank;
      /* The rank, a value used for error-checking in red-black trees */

      class SITE *first_site;
      /* The first site inorder */
      void check_node ( ostream & output , class blnNODE * node,
                  int& black_depth );
      void fix_double_red ( class blnNODE * red_child );
      void fix_rank_defect ( class blnNODE * heavy_child );
      void rotate ( class blnNODE * child_node );
      /* Standard red-black tree operations, necessary to maintain a balanced
tree, with optimal insertion-deletion timings */
};
void Process ( EVENT_QUEUE & event_queue, ostream & output,int Site_count);
```

```
int clckws (point a, point b, point c);
/* are the points a, b, c in clockwise order? */

class SITE_LIST
{
public:
class SITE * * item;
/* A collection of pointers to sites */

SITE_LIST(int num=0);
/* Standard creation function */

void addsite(SITE * siteptr, int i);
/* Puts a site siteptr into the space indexed by i */
};
class EVENT_LIST
{
public:
class EVENT * * item;
/* A collection of pointer to events */

EVENT_LIST(int num=0);
/* Standard creation function */

void addevent(EVENT * eventptr,int i);
/* Puts a site siteptr into the space indexed by i */
};
```

```
/*-----------------------------------------------------------------------*/
/* Everything below this line is artificially inserted. These functions
were originally kept in the modular programs in which they were used,
since they do not act on any obvious item, in the sense that handle
obviously acts on an event */
circle * incentre(circle c1, circle c2, circle c3);
/* returns a circle as the incentre of c1,c2 and c3. Automatically
throws out any circle on which the tangent ponts on c1,c2 and c3 are in
anti-clockwise order. */

void Count_sites()
/* Counts the sites, and stores the answer in the global variable
Site_count*/

void get_input()
/* Interprets the input. First, we create the specified sites, and
store them in a temporary list. we then take the list of events, and
for each event, create the data structure necessary, fill in the listed
sites, and put the event on another temporary list. If the event's
description includes link(s) back to another event, we find the indexed
event(s) from the list, and set corresponding link(s) forwards from that.
We then place the event on the global Events-queue, and, if this is the
first event attached to either of its site(s), set the appropriate variable
on the site.*/

main(int argc, char*argv[])
/* Open the specified input and output files, counts the sites, creates
the Event-queue to be four times the number of sites, calls get_input,
and passes the resultant Event_queue to Process. */

double distance ( circle& a, point& b )
/* Returns the distance between a and b */

inline double determinant ( double a, double b, double c, double d )
/* Returns the determinant of the 2*2 matrix
|a b|
|c d|
*/
```

```
point circum_centre ( circle & p , circle  & q , circle  & r )
/* Returns the circumcentre of the three point p,q and r */

int clckws (point a, point b, point c)
/* Returns 1 if the sites go clockwise around any point inside them */

void Adjust_frame ( point & pt )

/* Adjusts the 'world' boundaries to include the point pt, if they didn't
already */

void Process ( EVENT_QUEUE& event_queue, ostream& output,int Site_count)

/* The first processing procedure. We start by taking the first event off the
event queue, and initialising the beachline with its site, and setting the
next upper and lower events of the only two cusps, if necessary. We also
start the worldview limits (North, South, East and West) to its x and y
limits. In the main loop, we take events off the events queue and handle
them, until all are gone. We then list off the cusps where they are, so
the drawing program has sufficient information to draw the semi-infinite
edges. */

int value (EVENT * this_event)
/* Returns a value assigned according to event type, so that events get taken
off in the correct order */

int compare(EQ_ENTRY *one, EQ_ENTRY *two)
/* True if one should be picked before two */
```