Algorithm for generation of Voronoi Diagrams

You may use whatever algorithm you like to generate your Voronoi Diagrams, as long as it is *yours* (no using somebody's Voronoi generating package) and runs in at worst O(n^2) time. The algorithm below is the simplest algorithm we could come up with, and it runs in Theta(n^2) (for the truly curious, this bound holds in part because it can be proven that a Voronoi diagram has at most O(n) edges). There are several algorithms which run in O(n log n) time. If you implement one of these you will definitely exceed our expectations (they are all quite tricky!). Information on these can be found on the web (e.g., in an on-line chapter on Voronoi diagrams or from the Voronoi Page), or you can ask us. If you do choose to use an algorithm other than the one outlined below, make sure you carefully document how it works in comments and in your README.

Terminology

We shall refer to the points that define the Voronoi Diagram as *sites*. The sites are the inputs to the algorithm, but are not part of the output. Each site will have a corresponding polygon in the Voronoi Diagram, which we shall call a *cell*. Cells will be made up of *edges*, which meet at *vertices*.

Note: for the numerical calculations we assume that we are dealing with real numbers. When you implement this you should use floating point numbers, but be aware that floating point numbers are *not* the same thing as real numbers. This probably won't cause any problems but sometimes does in computational geometry problems such as this one.

Data Structures

You may use whatever data structures you deem appropriate to implement the four sets below. For each set, we have listed what operations you need to be able to perform.

- A set of sites, *S*. This set must allow enumerate (which may be destructive, because we only do it once).
- A set of edges, *E*. This set must allow enumerate (non-destructive), insert and delete.
- A set of cells, *C*. This set must allow enumerate (non-destructive) and insert.

You will also need data types for the following:

- **Point**; an x-coordinate and a y-coordinate.

- **Edge**; an edge must have two endpoints.
- **Cell**; a cell must have a site and a list of edges.

The Algorithm

Input: int *width*, int *height*, set of sites *S*.

```
[1] Initialize E and C to be empty.

[2] Add three or four "points at infinity" to C, to bound the diagram. For example:

  • Create four cells at (-width, -height), (2*width, -height), (2*width, 2*height), and (-width, 2*height).
  • Add appropriate edges to these cells. For example, the first cell listed above might have edges [(width/2, height/2), (width/2, -10*height)], [(width/2, -10*height), (-10*width, height/2)], and [(-10*width, height/2), (width/2, height/2)].

[3] For each site site in S, do:

    [4] Create new cell cell, with site as its site.

    [5] For each existing cell c in C, do:

        [6] Find the halfway line between site and c�s site (this is the perpendicular bisector of the line segment connecting the two sites). Call this pb.

        [7] Create a data structure X to hold the critical points (see step 11).

        [8] For each edge e of c, do:

            [9] Test the spatial relationship between e and pb.

            [10] If e is on the near side of pb (closer to site than to c�s site), mark it to be deleted (or delete it now provided that doing so will not disrupt your enumeration).

            [11] If e intersects pb, clip e to the far side of pb, and store the point of intersection in X.

        [12] X should now have 0 or 2 points. If it has 2, create a new edge to connect them. Add this edge to c, cell, and E.

        [13] If necessary, delete any edges marked in step 10 from both c and E.
    [14] Add cell to $C$.
```

- Now we have a complete Voronoi Diagram, with a single copy of each edge stored in *E*. We now need to clip each edge to the bounding rectangle, and add new edges along the border to connect the gaps. This can be done as follows:

```
[15] For each side border of the rectangle, do:
    [16] Create a data structure $P$ to hold the critical points (I used a priority queue), and add the endpoints of border to $P$.
    [17] For each edge $e$ in $E$, do (this should look familiar):
        [18] Test the spatial relationship between $e$ and border.
        [19] If $e$ is on the outside of border, mark it to be deleted (or delete it now provided that doing so will not disrupt your enumeration).
        [20] If $e$ intersects border, clip $e$ to the inside of border, and store the point of intersection in $P$.
    [21] Create new edges to connect adjacent points in $P$, and add these edges to $E$.
    [22] If necessary, delete any edges marked in step 19 from $E$.
```

Output: *E*. (You also may find it helpful to output *C*, although some modification is necessary to get *C* to agree with *E*, since the cells in *C* are not clipped to the bounding rectangle.)

The result of this is a set of edges describing the Voronoi diagram. Remember that your output needs to describe the vertices (points), edges (walls), and cells (rooms). Moreover, adjacent rooms should *share* their common wall, not use two separate instantiations of it. You will need to massage this output into that form. But, barring that...

Yay! We�re done! Wasn�t that fun? Keep in mind that you are free to modify this algorithm however you see fit. But just make sure to document your choices!

Testing Spatial Relationships

In the above algorithm we need to be able to determine which side of a line a given point is on. We can do this using *cross products* of the vectors describing those points. I will give a very brief explanation here. For more detail refer to a text on Computational Geometry. A *vector* is essentially like a point; it has an x-coordinate and a y-coordinate. The cross product of two vectors V1 and V2 (with corresponding coordinates (X1, Y1) and (X2, Y2)) is given by V1 x V2 = X1*Y2 - X2*Y1. Note that this is equal to the negative of V2 x V1. If the vectors are collinear (ie they point in the same, or opposite, direction) the cross product will be zero. Now back to the original problem. Suppose we are given a line defined by points A and B, and a point P which we want to test. To do this we calculate the cross product (P - A) x (B - A). If P would be to our *right* as we walk along the vector from A to B, this value will be positive. If P would be to our *left* it will be negative. If P is on the line AB then it will be zero.

Check out Yahoo's Computational Geometry page for tips, tricks, and data structures useful in computational geometry.

THIS IS THE END OF THIS DOCUMENT

I LIED. ACTUALLY, **THIS** IS THE END OF THIS DOCUMENT