Comp 163: Computational Geometry
Tufts University, Spring 2005

Professor Diane Souvaine
Scribe: Ryan Coleman

Point Inclusion and Processing Simple Polygons into Monotone Regions

1 Point Inclusion in a Polygon

[1]

An example of a typical decision problem is the problem of determining *Inclusion in a polygon*. One way is to draw a line from the given query point and compute the intersections of that line with the polygon. Find the number of intersection points on both sides of the query point. If the number is odd on both sides, the point is in the polygon, else it is out. This can clearly be done in $O(n)$ time, where n is the number of vertices in the polygon.

The best algorithm we've discussed for point inclusion in a simple polygon takes $O(n)$ time. A short review of this algorithm follows (leaving out degenerate cases).

1. Begin by drawing a line L through the point to be tested P_0. (Any line passing through the point will do). Also, initialize two counters $Left$ and $Right$ to zero.
2. Assuming you have the list of points of the simple polygon in clockwise order, start at the beginning and walk through the list of points, checking the following:
   (a) Check intersection between the line formed by the current point and the next point with the line L. If the lines intersect, check to which side of P_0 the intersection occurs. Increment the appropriate counter ($Left$ or $Right$). This can be done in constant time.
3. In the end, if both counters $Left$ and $Right$ are odd, the point P_0 is inside the polygon, otherwise it is outside the polygon.

[1]This section adapted from lecture 1, Spring 1990

Since you execute a constant time step for each edge in the list, the algorithm is clearly $O(n)$. Also, the problem has an easy to state lower bound of $\Omega(n)$ since unless you have checked each point, you could be wrong about whether or not the point was in the polygon.

A class of polygons which is easier to handle is the class of **convex polygons**. As the name suggests, these polygons satisfy the property of convexity namely, if $x, y \in P \Rightarrow \lambda x + (1-\lambda)y \in P$ for $\lambda \in [0,1]$.

Point inclusion in a convex polygon takes $O(\log n)$ time.

Given a convex polygon P, we can always find a line which leaves P on one side. A line, which has the above property, and touches P is called a *supporting line* for P. So, a supporting line intersects the boundary but not the interior.

Convex polygons can be characterized by the following properties:

1. Two disjoint convex polygons can always be separated by a line supporting one of them.

2. A convex polygon admits a supporting line at each vertex.

The point inclusion problem becomes rather easy when we are in the realm of convex polygons. Let q be the query point and P be the given convex polygon, where the vertices are given in clockwise order around the boundary. Let y(q) denote the y-coordinate of q. The following steps are required for point inclusion:

```
1. Determine the points of the maximum and minimum y-coordinate.

2. Using binary search, isolate the two edges intersected by the
   line $l$, where $l$ is y = y($q$).

3. See if $q$ lies between these two edges.
```

Using the naive approach, step 1 requires $O(n)$ time, step 2 $O(\log n)$, time and step 3 $O(1)$ time. If step 1 could be performed in lesser time, the overall complexity would be reduced. To reduce that complexity, we use the following property exhibited by the y-coordinates of the points of a convex polygon. If we draw a graph of the index of each point vs. its y-coordinate, the resulting graph has at most one maxima and at most one minima. An example of this is shown in figure 1. The following procedure locates the vertex with the maximum y-coordinate:

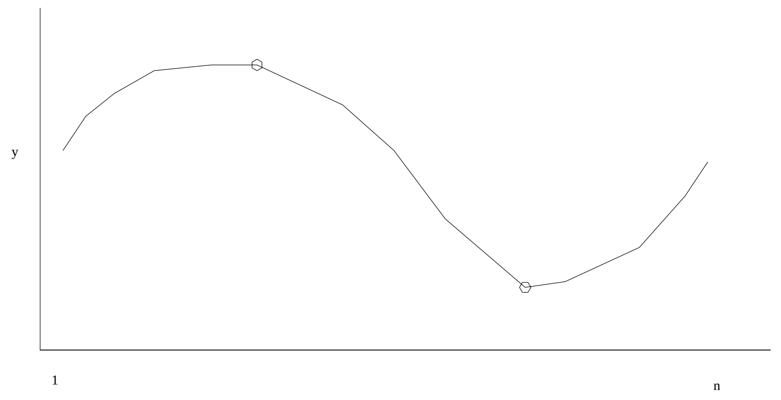

Figure 1: An example graph the y-coordinates of a convex hull

```
    Let y(1),...,y(n) be the y-coordinates.
    Compare y(1), y($\frac{n}{3}$), y($\frac{2n}{3}$) and $y(n)$ and select the largest.
    If the largest is y($\frac{n}{3}$) (resp.  y($\frac{2n}{3}$)), then clearly the maximum
must lie somewhere in the first (resp.  last) $\frac{2}{3}$ entries and vice
versa.
    So we can rule out the last (resp.  first) $\frac{1}{3}$ entries.
    If the largest is y(1) (resp.  y(n)), then the maximum must lie
in the first (resp.  last) $\frac{1}{3}$ entries, and thus $\frac{2}{3}$ entries can be deleted.
```

This algorithm determines the maximum (minimum) entry in $O(\log_{\frac{3}{2}} n)$ time which is $O(\log n)$.

(Fine tuning this algorithm using Fibonnaci numbers improves the constant, but not the asymptotic complexity).

1.1 Relation to Other Problems

The algorithm for point inclusion in convex polygons in $O(\log n)$ time can be generalized to run on any monotone polygon with respect to one dimension. A polygon is monotone in respect to the x-direction if it's intersection with any vertical line is $\emptyset$ or a single connected component. See figure 2 for an example of a monotone polygon and figure 3 for an example of a non-monotone polygon.

Since point inclusion in monotone polygons is $O(\log n)$, if we could spend time breaking a simple polygon into monotone polygons, we could do point

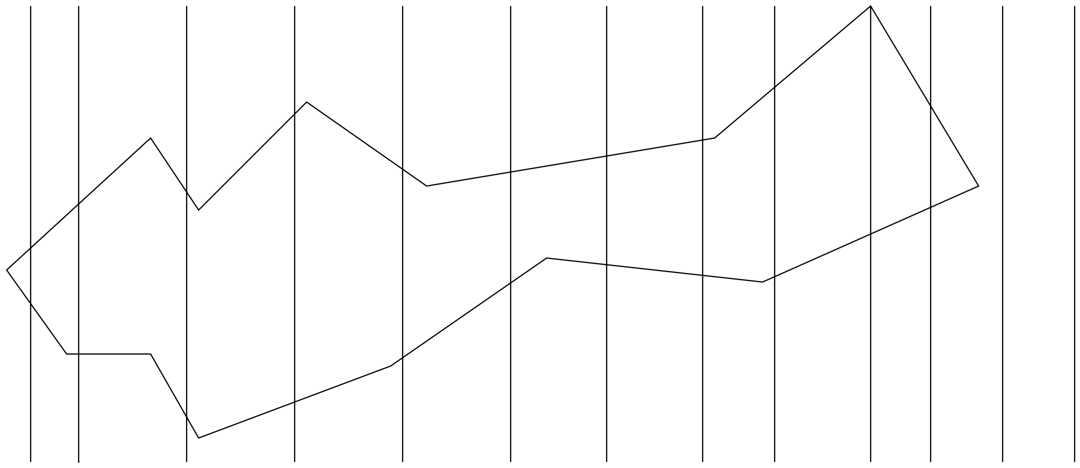

Figure 2: A monotone polygon

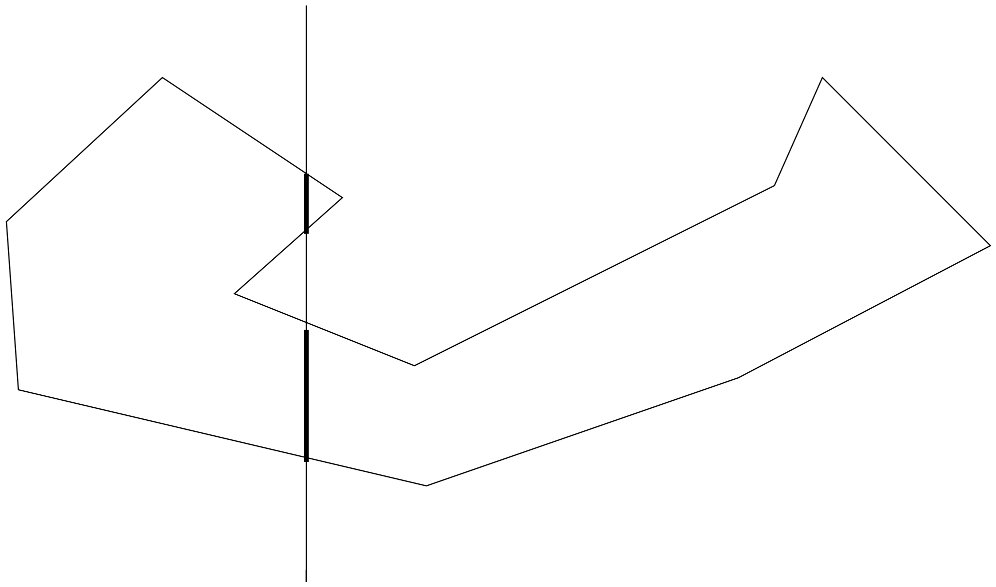

Figure 3: An example of a polygon that is not monotone

inclusion in better than $O(n)$ time, discounting the time spent preprocessing (which we do only do once).

Many problems are easy not only for convex polygons but also for slightly more general classes of polygons. Here are a few important definitions:

A polygonal chain is a sequence of vertices and edges joining adjacent vertices.

A monotone chain is a polygonal chain for which there is a distinguished line l such that if its vertices are projected on l, then the order of the vertices is preserved.

A monotone polygon is a polygon that can be split into two chains which are monotone with respect to a selected direction.

A polygon P is said to be star shaped if there is a point q in its interior such that, for any point $p \in$ P, the line segment joining p and q lies wholly within P.

For a star shaped polygon P, the set of all points q having the above property (the set of points in the interior of P which can see all points of P) is called the kernel of P. The kernel is always a convex polygon and has at most n sides if P has n sides.

It is important to note that inclusion is just as easy in monotone polygons as in convex polygons *if* you know the direction of monotonicity and the same holds for star shaped polygons *if* you know a point in the kernel.

Another way to test for point inclusion in a simple polygon is to use *D*ecomposition. Given a simple polygon, by adding diagonals joining vertices of the polygon which are interior to the polygon, it is possible to decompose the polygon into convex pieces, monotone pieces, or even triangles. To test for point inclusion, we could triangulate the polygon and then determine if the query point lies in one of the triangles interior to the polygon. For a simple polygon, a triangulation gives rise to $O(n)$ triangles (Euler's theorem on planar graphs), and checking within a triangle takes only $O(1)$ time (if a,b,c are the vertices of a triangle in counterclockwise order, check whether q lies to the left of each of $\vec{ab}, \vec{bc}$ and $\vec{ca}$). So, if the triangulation is provided, then inclusion testing can be done in $O(n)$ time.

The problem of triangulating a simple polygon, however, is a classic *Computation Problem* in computational geometry. The best known algorithm for the triangulation of a simple polygon takes $O(n)$ due to an algorithm by Chazelle time and the lower bound is $\Omega(n)$.

2 Processing Simple Polygons into Monotone Regions

2.1 Approach

The approach is this problem is to use a sweep line across the polygon from left to right and update a data structure corresponding to the current vertical components seen as you sweep across. While doing this, divide the simple polygon into monotone regions, putting them into a data structure that allows each region to be analyzed for point inclusion in $O(\log n)$ time and allows each region to know its neighbor regions as well.

The goal of the algorithm is to create a $O(n \log n)$ time algorithm that sweeps the polygon, and at the end have a data structure and algorithm that allows point inclusion in the simple polygon to be decided in $O(\log n)$ time.

2.2 Data Structures

2.2.1 Stopping Points

To simulate a sweep line moving across the plane, a *StoppingPoints* data structure is needed. This holds the points where *STATUS* needs updated. It is keyed on lower x-coordinate (assume general position or break ties based on lower y-coordinate). Each point in *StoppingPoints* needs to have pointers or references to the two edges it is connected to. This could be implemented with pointers into a DLL (doubly-linked list) of vertices in clockwise (or counterclockwise) order.

Some data structure to store the monotone polygons is needed. A *doubly connected edge list* DCEL would work. A DCEL has the following properties:

1. Any vertex has access to one edge that it is on.
2. Any face has access to one edge that surrounds it.
3. Any edge has access to 2 vertices and the face to it's right and the face to it's left.

2.2.2 STATUS

A second data structure, a Balanced Binary Search Tree, that represents the current portion of the polygon the sweep line is currently crossing is needed (the $STATUS$ data structure) maintained using each intersecting the sweep line in its current position.

$STATUS$ is a BBST of edges, sorted based on relative y-position of edges. For our purposes and examples, lower y-coordinate edges will be to the left of edges with higher y-coordinates.

Edges can be stored as 4-tuples $< X_{min}, X_{max}, m, b >$ where the terms are the x-coordinates of the endpoints, the slope of the line and the intercept of the line.

Either you could store edges in the leaf nodes of the tree, and use the internal nodes as decision nodes to reach the edge nodes, or you could leave all the data in the internal nodes. It will also be useful to cyclically link our edges (so that each edge knows the edges above and below it), so it is easier to use the tree with data stored in leaf nodes, though you could also doubly thread the other form of the tree.

Each edge which has the interior of the polygon directly above it needs a pointer to its *helper*. This is defined as the point closest to the sweep line between this interior edge and the edge to it's right in $STATUS$. (This is used later in the section on critical points). It should be noted that the helper of an edge could point to its own vertex.

2.3 Algorithm Sketch

As points are processed (off the *StoppingPoints* heap), 3 kinds of updates to $STATUS$ are possible.

1. Put 2 new edges into $STATUS$. See figure 4.

2. Replace an edge within $STATUS$ (adding old edge into DCEL). See figure 5.

3. Remove 2 edges from $STATUS$ (adding old edges into DCEL) Note: these edges will be adjacent in $STATUS$. See figure 6.

We now need the concept of critical vertices. These are the vertices that will have to be 'fixed' in order to break the polygon into monotone regions. The first or the third kind of update to the BBST can be critical vertices. When the sweep line passes through those points (or, they are popped off the *StoppingPoints* heap), if these points are 'interior' to the polygon, they are critical points. Testing 'interior' is simple, once the node in $STATUS$ is

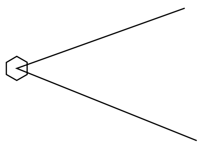
Figure 4: Put 2 new edges in $STATUS$

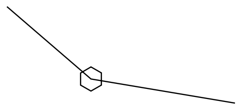
Figure 5: Replace an edge in $STATUS$

found where the edges will be added, count the number of data nodes to the left and to the right of this node. If they are both odd, then the point is a critical point, otherwise it is not a critical point. See figure 7 for examples of critical points.

While updating the helper field of lines in $STATUS$ where the immediate region above them is interior to the polygon, you do two things to fix the critical points, and break the polygon into monotone regions.

Call the critical point in figure 7 on the left a *split* and the one on the right a *merge*. A merge is easier to handle, you simply connect it to the helper of the line to the left in $STATUS$ where the critical point is found.

The method for handling splits is harder, since you can't see anything to the right of the sweep line yet. But, if whenever replacing a helper pointer that was not the endpoint of the line itself, you can connect the previous and new values of helper, you will connect the critical points together.

Both of these connects are not simple procedures, since you have to connect edges, split faces, etc. in the DCEL output.

Algorithm can be extended to simple polygon with holes by adding support for multiple linked lists of points representing boundaries of both the polygon and the holes.

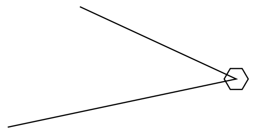
Figure 6: Remove 2 edges from $STATUS$

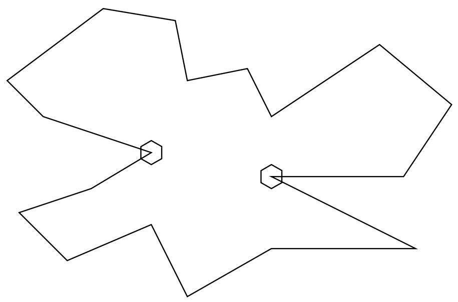

Figure 7: Examples of each kind of critical point

Since for each n points in *StoppingPoints*, $STATUS$ may have to be searched and updated, which takes $O(\log n)$ time, the algorithm takes a total of $O(n \log n)$ time. $STATUS$ never has more than n edges in it, since there are only n edges in the simple polygon, and *StoppingPoints* as well contains only the original vertices of the polygon, so the space complexity is $O(n)$.