# CISC 829 Computational Geometry

## HW1 due March 20

## WRITTEN EXERCISES:

**Problem 1: Textbook, exercise 1.4, p. 15.**

**Problem 2: Textbook, exercise 2.11, p. 43**

**Problem 3: Simple Polyline Hull Algorithms**

We have seen in class how to compute the convex hull of any 2D point set or polygon with no restrictions. The polygon could have been simple or not, connected or not. It could even have been just a random set of segments or points. The algorithms given, the "Graham Scan" and the "Andrew Chain", computed the hull in O($n$ log $n$) time. In this exercise, you need to develop an algorithm that improves this efficiency to O($n$) linear time for a ***connected simple polyline*** with no abnormal self-intersections.

**Why is it possible?**

Why should there be a faster O($n$) convex hull algorithms for simple polylines and polygons? To understand this, recall that most convex hull algorithms for point sets take O($n$ log $n$) time because initially sort the $n$ points. After that, they generally only require O($n$) time. So, one needs to ask why sorting is needed; that is, what does it accomplish? Consider how other algorithms proceed after sorting is done.

Most 2D convex hull algorithms use a basic incremental strategy. At the $k$-th stage, they have constructed the hull $\mathbf{H}_{k1}$ of the first $k$ points $\{P_0, P_1, ..., P_{k1}\}$, incrementally add the next point $P_k$, and then compute the next hull $\mathbf{H}_k$. How does presorting facilitate this process? The answer is that at each stage, one knows that the next point $P_k$ is exterior to the previous hull $\mathbf{H}_{k1}$, and thus one does not have to test for this. Otherwise, one would have to test $P_k$ against all $k$ edges of $\mathbf{H}_{k-1}$, resulting in O($n^2$) such tests totaled over all stages of the algorithm. Instead, one immediately knows that that $P_k$ is outside $\mathbf{H}_{k-1}$, and can proceed to construct $\mathbf{H}_k$ by extending $\mathbf{H}_{k-1}$. Thus, if presorting in O($n$ log $n$) time has been done, no such tests need to be made, and this results in a faster algorithm.

**Melkman's algorithm**

[Melkman, 1987] devised an ingenious method for organizing and implementing the operations to compute the hull of a simple polyline.

The strategy of the Melkman algorithm is straightforward. It sequentially processes each of the polyline vertices in order. Let the input polyline be given by the ordered vertex set: $\mathbf{V} = \{P_0,$

$P_1$, ..., $P_n$}. At each stage, the algorithm determines and stores (on a double-ended queue) those vertices that form the ordered hull for all polyline vertices considered so far. Then, the next vertex $P_k$ is considered. It satisfies one of two conditions: either (1) it is inside the currently constructed hull, and can be ignored; or (2) it is outside the current hull, and becomes a new hull vertex extending the old hull.

**The double-ended queue (called a "deque")**

A deque has both a top and a bottom. At both ends of the deque, elements can be either added or removed. At the top, we say an element is pushed or popped; while at the bottom, we say an element is inserted or deleted. The deque is given by an ordered list **D** = { $d_{bot}$, ..., $d_{top}$ }where *bot* is the index at the bottom, and *top* is for the top of **D**. The elements $d_i$ are vertices that form a polyline. When $d_{top} = d_{bot}$, then **D** forms a polygon. In the Melkman hull algorithm, after processing vertex $P_k$, the deque $\mathbf{D}_k$ satisfies:

1. The polygon $\mathbf{D}_k$ is the ccw convex hull $\mathbf{H}_k$ of the vertices $\mathbf{D}_k = \{P_0, ..., P_k\}$ already processed.
2. $d_{top} = d_{bot}$ is the most recent vertex processed that was added to $\mathbf{D}_k$.

If $P_k$ is inside $\mathbf{H}_{k-1}$, then $\mathbf{D}_k = \mathbf{D}_{k-1}$, and there is no associated processing. In this case, $P_k$ is inside the subregion of $\mathbf{H}_{k-1}$ bounded by the vertices: $d_{bot}$, $d_{bot+1}$,..., $d_{top-1}$, $d_{top}$

Part 3.1: Prove that if point $P_k$ is inside hull $H_{k-1}$ and $P_{k+1}$ is on hull $H_k$ if and only if edge $P_k$ $P_{k+1}$ crosses over one of the edge segments $d_{bot}d_{bot+1}$ or $d_{top-1}d_{top}$.

Part 3.2: How do you test if $P_k$ crosses the edge segments $d_{bot}d_{bot+1}$ or $d_{top-1}d_{top}$

Part 3.3: When $P_k$ is exterior to $\mathbf{H}_{k-1}$, we must then change $\mathbf{D}_{k-1}$ to produce a new deque $\mathbf{D}_k$ that satisfies the above two conditions. How shall we update $\mathbf{D}_k$?

Part 3.4: Analyze the time complexity of the above algorithm.

Part 3.5: Melkman's algorithm is considered to be the best convex hull algorithm for simple polygons. What distinguishes it from all the rest is that it is actually an on-line algorithm. Explain why this is the case.

## PROGRAMMING EXERCISES:

Part 0. Use your favorite Web search to find some computational geometry applets for convex hull, triangulation, etc. Note the various methods people use for interactive specification of input points and polygons.

Part 1. Adopt an existing Java interface or OpenGL interface to allow addition, deletion, and editing (moving) points on the xy plane. Support clearing (starting over with zero point). Support a restriction (say when pressing the shift key) of the entered point to a purely horizontal displacement, if the true mouse position is inside a 45-degree cone to the left or right of the point at which the shift key was pressed, or purely vertical displacement, if within the complement of this cone.

Part 2. Add a menu button or a key event to instigate a polygon-creation mode, in which points, as they are added, form a linear chain. Add a “close" button or key which creates an edge from the last point entered to the first point entered. Design it so that your point-editing code can be re-used, that is, after the chain is completed you can go back and drag its vertices around.

Part 3. Implement Melkman’s algorithm.

Part 4. Create a short web page that briefly describes your solution in plain language, links to a running applet that demonstrates your running solution, and gives instructions for using the applet. Make sure to tell us about any extra capabilities your solution has. The page should also link to your source code. Email the profs a link to the top-level web page.