Triangulating a monotone polygon

Suggestion from last class --> connect all "mutually x-adjacent" vertices from top to bottom.

The concept of "mutuality" runs into trouble: recall the example of an x-monotone polygon with x-coordinates doubling each time. Also, using Euclidean distance has its problems.

We concluded that it would be safe to add diagonals between the top and bottom chains, every time we hop from one to the other as we sweep horizontally. This would decompose the x-monotone polygon into what are called "monotone mountains": still monotone, but one of the chains is just a single edge. These are easy to triangulate because they are WEV.

See book by O'Rourke, p.51.
We also went over the proof by induction for triangulating monotone polygons again. Pictures are in notes from last class.
TESTING A POLYGON FOR MONOTONICITY

In a specific direction : easy → it's a local property

call it $\vec{x}$

start at $x_{\text{min}}$

don't go back till you hit $x_{\text{max}}$
Monotonicity in any direction?

Test x-positive directions:

Intersection of angles
Test only reflex vertices: 2 convex are allowed to fail first of all

Suppose a convex fails test (when it shouldn’t)

Somewhere, reflex will also fail

(not extreme) Convex

fail perpendicular test

Extreme convex
So in linear time we can get all the valid angular ranges. Every (reflex) vertex contributes a double wedge = 2 opposite cones on the unit circle. These cones have angle less than 180.

We are looking for the intersection of all of these cones, which all together is at most 2 cones: notice that every time you add one double wedge, you just restrict each cone of the current intersection region to a continuous subset.

It is in fact critical that angles are less than 180 for the above to be true. Thanks to Eliot and company for insisting on this.

So we can update the intersection in $O(1)$ time per vertex. $O(n)$ overall.
DECOMPOSITION OF POLYGON INTO X-MONOTONE PIECES

(partition)

Why? \(\rightarrow\) many reasons: e.g. triangulation

Obviously we've seen how to do this already
\(\leftrightarrow\) we already know how to triangulate
\(\leftrightarrow\) monotone pieces
Clearly no "piece" can contain \( \{ \) or \( \} \) "cusps".

So one method could be to identify these vertices and remove them by adding diagonals. 

\( \Rightarrow \) time: \( O(n) \) per cusp \( \ldots? \) assumes a diagonal will "resolve" a cusp and not create a new one.

So \( O(n^2) \)

Might as well triangulate in \( O(n \log n) \) time meaning... with some other method.

Recall, can always anchor a diagonal on a reflex vertex. Still reflex but not a cusp.
Monotone pieces in $O(n \log n)$ time

→ ties our known triangulation bound

→ why? → each monotone piece can be triangulated in linear time

→ heuristic to partition into larger pieces

Again our plan is to remove cusps
Idea:

- this should not be looking to join to
- this
By adding this orange vertical, we split off vertices that will never belong in the same monotone piece, and will never be mutually involved in "cusp removing"
Adding a vertical through every cusp.

Now we have monotone pieces, but... using Steiner vertices.

Can we find a resolving diagonal for every cusp?
Every “piece” has a left & right vertical wall w/ a cusp vertex on it.

Claim: if \( \Rightarrow \) then both get resolved.

including \( \Rightarrow \) but not
If $\triangleright \triangleright \triangleright$, then $\circ$ can be resolved.

Again the proof is constructive.

Pink vector means sweep to find a vertex to join to the cusp.
Every cusp can find a diagonal in the direction shown by the pink arrows (direction meaning left vs right).

What we haven't shown yet is how to actually get all these verticals through the cusps, efficiently. In fact we can instead form verticals through all vertices.
PARTITION INTO TRAPEZIODS with sweep line

Assume no 2 vertices w/ same x-coord : easy to deal with
(what you're actually storing is the labels of the edges that bound the intersections just mentioned... so when you sweep by a little, these don't change)
While you don’t sweep through any vertices, the only changes come from slopes.

Also, no verticals are added.
When you hit the next vertex, look at edges incident to \( v \).

If they are \( x \)-monotone, just update with new slope. Number of segments on sweep line remains same.

Otherwise:

**OR**

Merge

Split
We could maintain a list of intersections, and find the merge or split in $O(n)$ time. But we can also do binary search

$\Rightarrow$ maintain a balanced search tree

Find gap containing $v$. Delete leaf & update other.

Find segment containing $v$. Update leaf & create new.

merge

OR

split
Balanced tree: \( \{ \text{height, insertion, deletion, update} \} \) \( O(\log n) \) time

Of course when we hit a vertex, we create a vertical segment. (notice we will create \( O(n) \) vertical segments)

This is why the slopes are useful: but really just maintain the edge.

\[
\text{compute from previous vertex & slope}
\]

\[
\text{coordinates known}
\]
Result: \( O(\log n) \) per vertex swept
\( O(n \log n) \) to get trapezoidal decomposition

Notice that it is easier to get rid of cusps compared to adding verticals only at cusps.
[Every trapezoid is empty and has a vertex on both verticals]

\( O(n) \) total to triangulate after this.

—Pretty easy algorithm—
Expanded example of triangulating a star with knowledge of the kernel. More details on the algorithm are in notes from previous class.

Recall that we scan first to construct a list of all convex vertices that correspond to convex quadrilaterals: CQ.
Take one CQ and flip it. This affects the rest of the triangulation only.
By flipping, the CQ no longer exists (we "remove" that convex vertex)

... new polygon boundary has the light blue edge instead of the ear we just chopped off.

So we go forward by 1 on the CQ list.
Repeat:
flip
update locally
advance
Notice that so far every convex vertex has qualified as a CQ.
This is the first time that a convex vertex failed to qualify as a CQ. Both of these form non-convex quadrilaterals.
No more CQ's
Must mean we're done.
Triangulation of a star-shaped polygon without knowing the kernel.
- Pick any vertex: 0
- Compute vis.pol of 0
SPECIAL WINDOW REGIONS

Geodesic path \( x \rightarrow y \)

Angular sweep

Pocket

Sweep until you hit a vertex.
Repeat recursively

Until you hit \( v \)
You know what vertex you're starting from, and what vertex you want to end on. If you compute the convex hull of the light blue chain, *ignoring* the part that goes below the dark blue edge, this will contain the geodesic.

You can ignore this part by adding another region test to Melkman's algo.

**Melkman**

\[
\text{time} = O(\text{pocket}) \quad \ldots \text{total} \quad O(n)
\]
So far we have the extended vis.pol of.

NEXT:

If \( \circ \) sees \( \bigcirc \) then form diagonal \( \overline{VY} \).

Of course you do this for all pockets. So in one scan of the vis.pol, you can determine which vertices are seen.
But we've also constructed these other nice regions near each window. Each of them is easy to triangulate.
Recall that we do not know where the red kernel of the star-shaped polygon is. But we will use the fact that it exists to conclude that the purple regions are WEV. That is because the red kernel is in the vis.pol we constructed, meaning it is outside the purple regions.
Another Idea
... that failed

Notice: if you could find a triangle containing a kernel point then you could run Sklansky on the 3 pockets.
Idea:
Find a diagonal that splits the polygon "evenly".
Then run Sklansky on both sides, correctly.
If both terminate, done.
If one fails, repeat Geometric series ... $O(n)$.
> 1 side must terminate correctly.
We need 2 things:
1) Find a balanced cut
2) Be able to recognize when Sklansky fails

I still claim that if Sklansky fails, then you will detect it because you won't "find the convex hull pocket lid".

Which I would say is equivalent to getting some output that doesn't contain the diagonal.

Notice that in this particular example you can immediately see what side the kernel is on, by looking locally at the lower endpoint of the diagonal.
**Balanced Diagonals in Polygonal Triangulations**

...which is not exactly what we needed for the previous idea, but interesting anyway

4. Any diagonal splits a polygon into 2 chains
   How balanced can this split be?

   4. Can you find a diagonal to split $\frac{n}{2}$? $\frac{9n}{10}$ $\frac{n}{10}$?

   6. What if I already give you a triangulation?
      then yes
      which implies yes for the first question
      but the problem is finding it quickly
      without triangulating first.

Note that if we could find a $O(n)$-time method to find a balanced diagonal,
that would immediately imply a simple divide-and-conquer $O(n \log n)$ triangulation
algorithm.
Finding a $\frac{n}{4} \vee \frac{3n}{4}$ split given a triangulation

- Arbitrarily split polygon into 4 chains of size $\sim \frac{n}{4}$

- If you find a diagonal between $C_1-C_3$ or $C_2-C_4$ then done.

- Not all diagonals can be within their chain, e.g. $C_1-C_1$ or $C_4-C_4$. Why? Because you would get a quad leftover.

- So, there exists a diagonal from $C_1-C_2$, wlog
Finding a $\frac{5}{4} v \frac{3n}{4}$ split:

- There exists a diagonal from $C_1 - C_2$
- If many, pick the one with endpoints $P_1P_2$ closest to $x_3$ & $x_4$
- Where is the apex of triangle $P_1P_2P_3$?
  → Where isn't it?

- $P_3$ is in $C_3$ or $C_4$

→ So we find a diag $G - C_3$ or $C_2 - C_4$

Done

... after all.
Remember, every polygon triangulation has a dual tree.

\[ \text{max degree 3} \]

It is always possible to make one cut on a degree-3 tree and break it into \( \frac{n}{3} \) or \( \frac{2n}{3} \) parts (roughly).

Why?
It is always possible to make one cut on a degree-3 tree and break it into $\frac{n}{3} \times \frac{2n}{3}$ parts (roughly).

Degree 2 node:

Degree 3 node:

If not balanced then there is a clear direction to make things more balanced.

\[
\begin{align*}
\text{If } \frac{1}{3} \leq A + B \leq \frac{2n}{3} & \rightarrow \text{ cut } B \\
\leq B + C & \leq \leq A + C & \leq \alpha \\
& > \frac{2n}{3} \\
\text{Other wise } A \text{ or } B \text{ or } C & \Rightarrow \text{ So walk that way}
\end{align*}
\]
yes, as pointed out by Max, you go to the heavier subtree.
POINT LOCATION

- We have seen basic examples:
  - Is a given point inside a given polygon [\(\Theta(n)\)]
    - Convex polygon [\(\Theta(n)\)]
    - Star-shaped polygon [\(\mathcal{O}(\log n)\)]

- We can solve those problems faster if the polygon is already known.

  Different scenarios:
  - For one query point
  - For \(k\) query points

2 things to minimize:
- Pre-processing time
- Space (data structure)
Pre-processing
1) store all vertices by x-coord.
Pre-processing
1) store all vertices by x-coord.
2) For every vertical slab:
   store a sorted list of segments that cross the slab (ordered & not intersecting)

Time: \(O(n \log n)\)
Space: \(O(n^2)\)

Query •
Pre-processing
1) store all vertices by x-coord.
2) For every vertical slab:
   store a sorted list of segments that cross the slab
   (ordered & not intersecting)

Time: $O(n \log n)$
Space: $O(n^2)$

Query in $O(\log n)$
1) Binary search on x
2) Binary search in 1 slab
Notice that from one slab to the next, the vertically sorted polygonal segments are almost identical. $(\pm 2)$

So it’s a huge waste of space to store a list of size $O(n)$ in each slab.
A Word About Persistent Data Structures
(just so that you know they exist)

A P.D.S. simply maintains information so that any previous version of the D.S. can be extracted. (without explicitly storing all versions)

It must still support regular operations: ex. insert/delete

For the slab decomposition, think of one binary tree (representing a slab) being modified \( n \) times as it sweeps through polygon

\( \Rightarrow \) Unlike a regular sweep, we care about remembering/retrieving what the tree looked like at any step.
Sarnak & Tarjan  persistent binary search tree for point location via slab decomposition

- $O(\log n)$ query after $n$ operations
- $O(n)$ space .... $O(1)$ per step amortized

This means (?) you could have to waste $O(n)$ time preprocessing between queries, but not many times.

Project ?