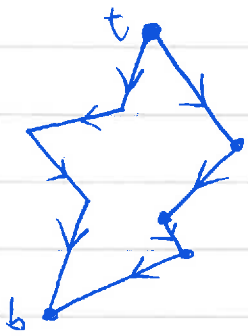
18/10/18 - Lecture 5

Last time,

monotone polygons: both paths from top to bottom are decreasing, w.r.t. lex ordering:

$a > b \iff a_y > b_y$ or $a_y = b_y$ & $a_x < b_x$.

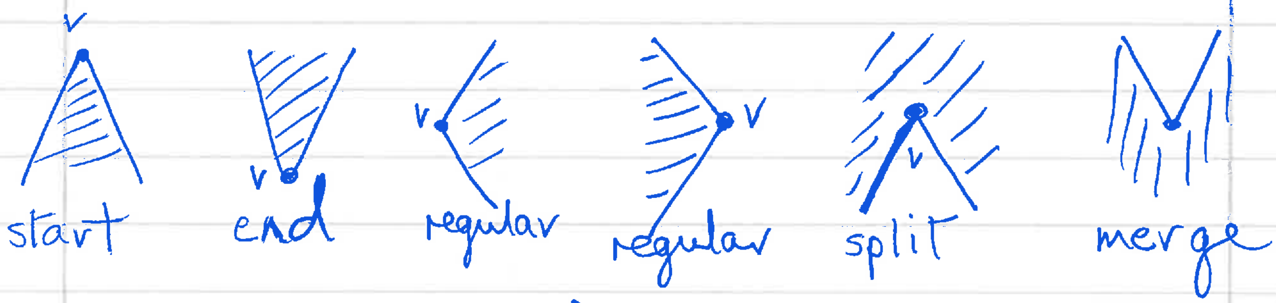


Algorithm for triangulating a simple polygon:

1. Divide it into monotone parts.
2. Triangulate monotone polygon.

- Last week, did 2 - time $O(\log n)$. This week, we do 1 - time $O(n \log n)$. So total time $O(n \log n)$.

Types of vertices & monotony



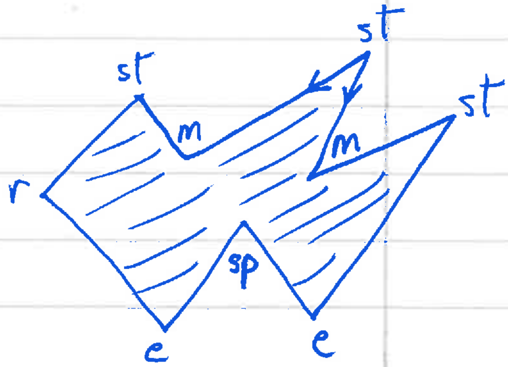
Ie. Start: $v > p, q$ (adj. vertices) & has polygon below.

End: $v < p, q$ & has polygon above.

Reg: $p < v < q$ or $q < v < p$.

Split: $v > p, q$ & has polygon above.

Merge: $v < p, q$ & has polygon below



Theorem) A polygon P is monotone $\iff$ it contains no split/merge vertices.

Proof)

- Suppose P contains a split vertex, so $v > p, q$.
- It can't be start or end, so lies in one path.
- In that path we have (p, v, q) or (q, v, p), neither of which are decreasing.
- Thus P is not monotone. Likewise if P contains a merge vertex it is not monotone.

Conversely, suppose that P is not monotone (suppose wlog left path not decreasing).

- Let p be least vertex on the path such that the path from t (top) to p is decreasing.
- Let q be largest vertex on path such that path from q to b (bottom) is decreasing.

Picture

Since P is not monotone p appears before q on the path.

- If p is above the polygon, it is merge.
- Else p is below the polygon, whence so is q.
- Since q is above its adjacent vertices then q is split.
- Hence P contains either a split or merge vertex.

- Given this result, we can break simple polygons into monotone ones by "removing" split and merge vertices.

Idea: - At merge vertex, draw a line downwards to a vertex.
- At split vertex, draw a line upwards to a vertex.

Naturally, we use a sweep line algorithm from top to bottom.

- Polygon stored in a DCEL.
- In queue we store vertices of polygon.
- In balanced binary tree T we store those edges intersecting the sweep line & having the polygon to the right.

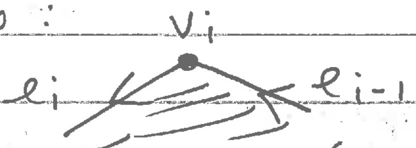
E.g. 

At ℓ, $T = \{e_4, e_2\}$.

- Also, with each edge e in T we store a vertex $p = \text{helper}(e)$ with it:
- helper(e) lies above ℓ,
- horizontal segment between e & helper(e) belongs to P,
- helper(e) is the least vertex (in lex order) with these properties.

- Overview : When sweepline passes a vertex we do some of
  - connect vertex with a helper by a segment in DCEL;
  - add edges & their helpers into T;
  - remove edges & their helpers from T;
  - change helpers of some edges in T.

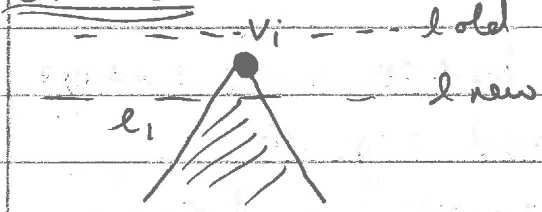
- Also, we use an anticlockwise enumeration of vertices and edges :



beginning from the top. (Calculate using DCEL.)

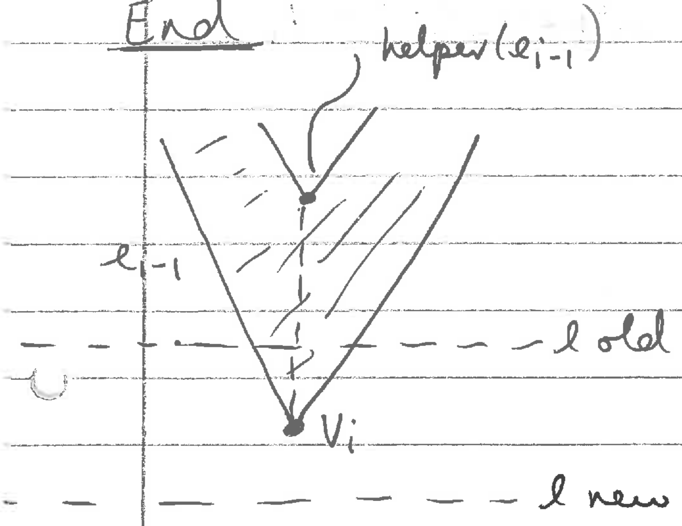
Cases:

Start



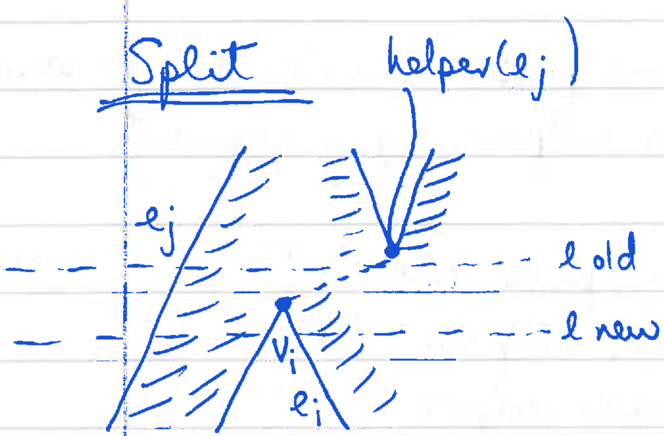
- Add e_i to T
- Set helper(e_i) = V_i

End



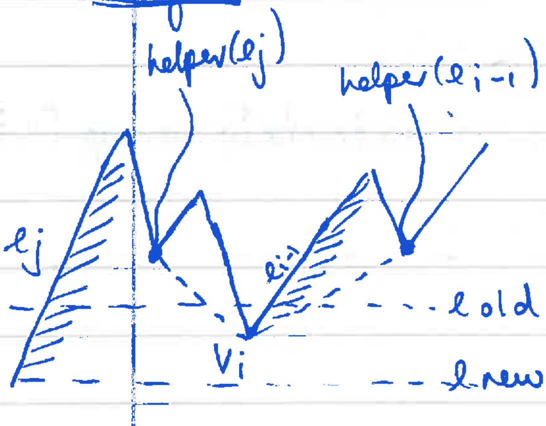
- If helper(e_{i-1}) is merge, add ~~vertex~~ edge from v_i to it.
- Remove e_{i-1} from T.

Split



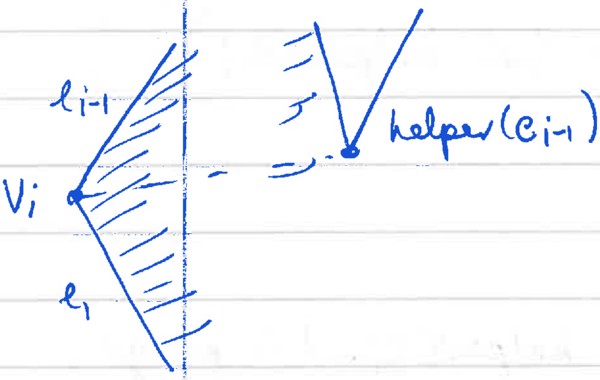
- Search T for closest edge e_j to left of v_i.
- Add edge from v_i to helper(e_j).
- Add e_i to T.
- Set helper(e_j) = v_i, helper(e_i) = v_i.

Merge



- If helper(e_{i-1}) is merge, add edge to v_i.
- Delete e_{i-1} from T.
- If helper(e_j) is merge, add edge to v_i.
- Set helper(e_j) = v_i.

Regular



- If helper(e_{i-1}) is merge add edge to v_i.
- Remove e_{i-1} from T.
- Add e_i to T.
- Set helper(e_i) = v_i.

Why does the algorithm work? (Sketch!)

- Consider split vertex v_i - it is connected to helper(e_j), the lowest vertex between its left & right neighbours e_j & e_k

- Consider merge vertex v_i, its left & right neighbours e_j & e_k. At the vertex v_i, we change helper(e_j) to v_i.

- Then at the max vertex p between e_j & e_k & below the sweep line we add edge to v_i

- In this way both split & merge vertices are removed.

Complexity:

- $O(n \log n)$ to order vertices into Q.
- $O(n)$ to calc. anticlockwise order.
- Each event involves searching in & rebalancing tree - time $O(\log n)$ - plus constant time ops: updating helpers, adding edges to DCEL.
- So time $O(n \log n)$ to handle the n events.
- Therefore complexity $O(n \log n) + O(n) + O(n \log n) = O(n \log n)$.