Triangulation – Contd.

Polygon Triangulation

Efficient triangulation, dividing a polygon into non-intersecting triangles with the polygon's vertices as the triangles' vertices, is $O(n \lg n)$. For a given serial representation of a chain of points in 2D representing the polygon, find the triangulation.

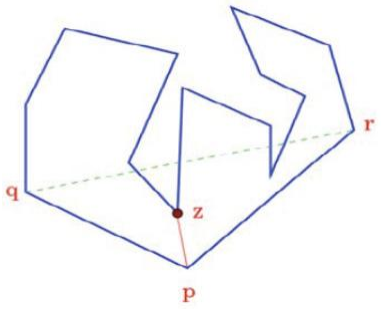
Theorem:

The triangulation of a polygon with n vertices has **exactly** $n-2$ triangles. The number of vertices, edges and faces are all linear.

Proof:

By induction; for $n = 3$ – trivial. Now assume correctness for all $3 < n' < n$, and we will prove correctness for n. To do that we pick a convex corner p of the polygon (any simple polygon **has** to have such corner; not all corners can be concave – can be proved by induction), and let q, r be the adjacent vertices of p.

We connect $q \to r$. If Δpqr is not entirely inside the polygon P, there must be a point z that is a vertex of the polygon and is inside the triangle, and then we would take $\overline{zp}$. Otherwise, we just take $\overline{qr}$. Then that diagonal will have a polygon on each of its sides, and will have n_1, n_2 vertices where $n_1 + n_2 = n + 2$ (as they both share the vertices q, r or z, p). By the induction assumption we get the number of triangles $T(\cdot)$ will be $T(n) = T(n_1) + T(n_2) = n_1 - 2 + n_2 - 2 = n + 2 - 2 - 2 = n - 2$, proving the inductive step.

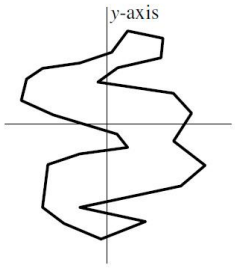


Overview:

We now present an $O(n \lg n)$ incremental desolation algorithm, that breaks the input polygon into smaller convex polygons – where each can be triangulated in linear time. So the steps of the algorithm are:

- Break down the polygon to monotone polygons
- Triangulate each

Monotone polygon:

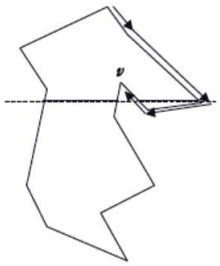


A piecewise linear chain is strictly monotone with respect to a line L, if any perpendicular of L intersects the chain at one point at most. A chain C is monotone with respect to a line L if every line orthogonal to L intersects C in at most one point.

A monotone polygon P w.r.t. a line L if its boundary δP can be split into two chains such that each is monotone with respect to L.

We will discuss y-monotone polygons, that can be broken into two chains to the left and right of the y line.

If any horizontal line that is orthogonal to the line intersects the two chain (boundary of P) at at-most two points.

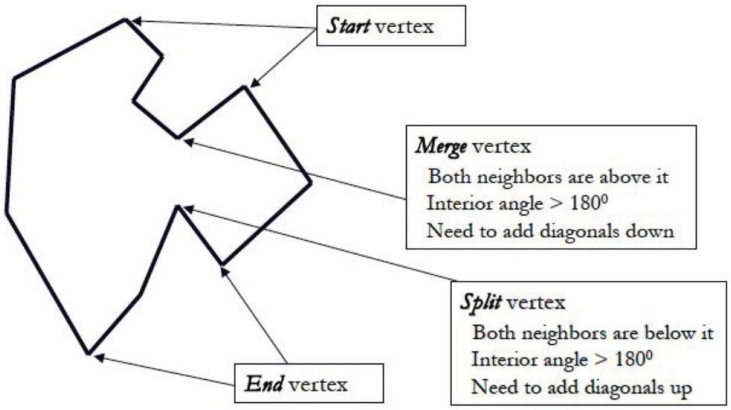


Turn vertices:

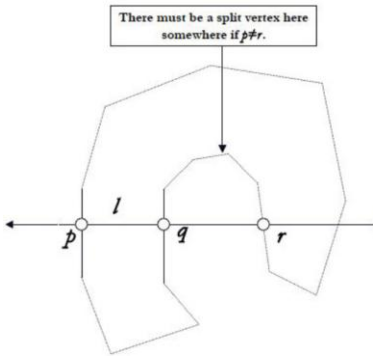
Points that break the monotonicity of polygons: a vertex where the direction of the walk switches from downward to upward (of vice-versa). A partition of P into y-monotone pieces is getting rid of such turns.

Assume both edges adjacent to a vertex v go downwards from v, we will cross a diagonal from v to some vertex upwards that partitions the polygon P, such that the polygons created are monotone. What diagonal is added?

Types of vertices:



The start and end vertices are not problematic for the monotonicity of P. Only local minimums / maximums (w.r.t. the y value) are problematic.



Lemma:

A polygon is y-monotone if it has no split or merge vertices.

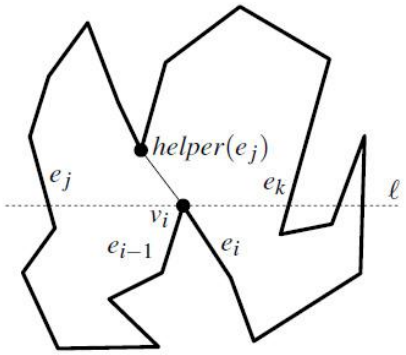
Proof:

Assume P is not y-monotone, so if we draw a line orthogonal to the line, it intersects the line at more than 2 points. We start with such placement of that line l. We walk on the boundary of P from q with the interior to the left of the walk on the chain, and we get to r that must be a different than p (otherwise $p = r$). There will be a local maximum between q and r – that is a split vertex.

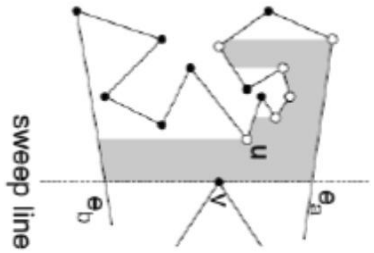
The other way, going down, will find a merge vertex.

In order to get a monotone partition of P we must get rid of all split and merge points.

The algorithm to remove split-points:



Assume v_i is a split point, then when our sweep line l encounters v_i it has two **support edges**, e_j, e_k – the edges l intersects immediately to the left and right of v_i. The best candidate to be the other end of the diagonal from v_i would be the upper end points of e_j or e_k, but that might intersects P. Therefore the chosen point would be $helper(e_j)$ – the vertex between e_j, e_k that has minimal y value. Such vertex must exist, and if the polygon is convex from v_i up, it would be the top lower endpoint between e_j, e_k.



Imagine that there's a light from e_k towards the interior strip (in grey) of the polygon; out of all vertices eliminated by this light, the lowest visible vertex from e_k can be seen by any point on the line l. **Why**?

Assume there is a point on l that cannot see the lowest e_k-visible vertex u, then there must be an edge that "blocks" the view of u. That edge cannot intersect l, since e_j, e_k are the support edges of l. So the lower endpoint of the blocking edge is seen from any point on l and has lower y value than u – but it is also visible by e_k thus it would have been chosen instead of u.

Therefore choosing the lowest vertex between e_j, e_k guarantees it is seen by v_i. This vertex is denoted the **helper** of an edge – the lowest point in the gray area (as in the diagram).

We then cross a diagonal between v_i and the helper of the right (in the book: left) supporting edge, in this case $helper(e_k)$.

Note that helpers are defined only for edges such that the interior of the polygon is to the left of the edge.

Sweep-line Algorithm

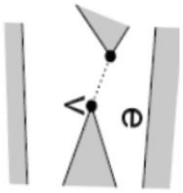
Events:

We start with a static set of events that will not change along the algorithm. The events are simply the endpoints of the edges of the polygon (the vertices), sorted by increasing y value. No priority queue is needed here.

Sweep status:

The list of edges that intersect the sweep line, sorted from right to left. When the line l gets to e_a, $helper(e_a)$ is initialized to the top endpoint of e_a itself. It will change when a new vertex is encountered by l that is seen from e_a and has a lower y-value. The edges that have helpers are kept in a binary search tree, so inserting / deleting / successor / predecessor / find – all take $\lg n$.

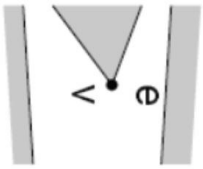
Event types:



Split vertex:

Let v be the current vertex encountered by the sweep line. Checking if v is a split vertex can be done in constant time – checking the P-interior angle of v. We then extract v's support edges e_a, e_b (successor and predecessor - $\lg n$) and we know that at this point:

- The current helper of e_a is to be chosen to cross a diagonal from v to it.
- v now becomes the helper of e_a



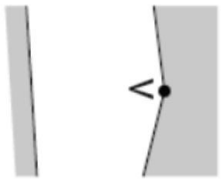
When we get to a merge point:

- We delete the edges adjacent to it from the sweep line, as they stop intersecting it - $O(\lg n)$.
- We update v to be the new helper of the right supporting edge, e_a. Finding that edge (the successor) is $\lg n$, and changing the helper for e_a is constant.

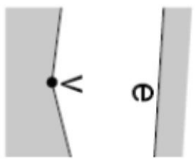
Start vertex:

- Add the two adjacent edges to the sweep line status - $O(\lg n)$.
- Set v as the new vertex of the right edge – constant.

End vertex: the interior angle is < 180 degrees, so we remove both edges from the status - $\log n$. v will not be a helper anymore.



Regular vertex on a right chain: we need to delete the edge we're leaving and add the next, adding a helper for it. Deleting and adding is $\log n$ and making v to be the helper is constant.



Regular vertex on a left chain: other than $\log n$ for deleting previous edge and adding the next, there's finding the successor of v – the right edge e_a – that will take $\lg n$, and making v its new helper – constant.

For the merge points – we do the same upside down, all merge points become split points.

The total complexity: $O(n \lg n)$ – every vertex spends $\lg n$.

Triangulation of monotone polygons:

We can triangulate a monotone polygon by a simple variation of the plane sweep algorithm.

We have both chains to the left and right of the line L, each sorted by its y value. We can merge these two sorted lists in $O(n)$ time.

The idea is to run a sweep line l down through the sorted points, and create triangles as we go. As we go, each triangle we're closing, anything above that triangle is already taken care of.

Lemma:

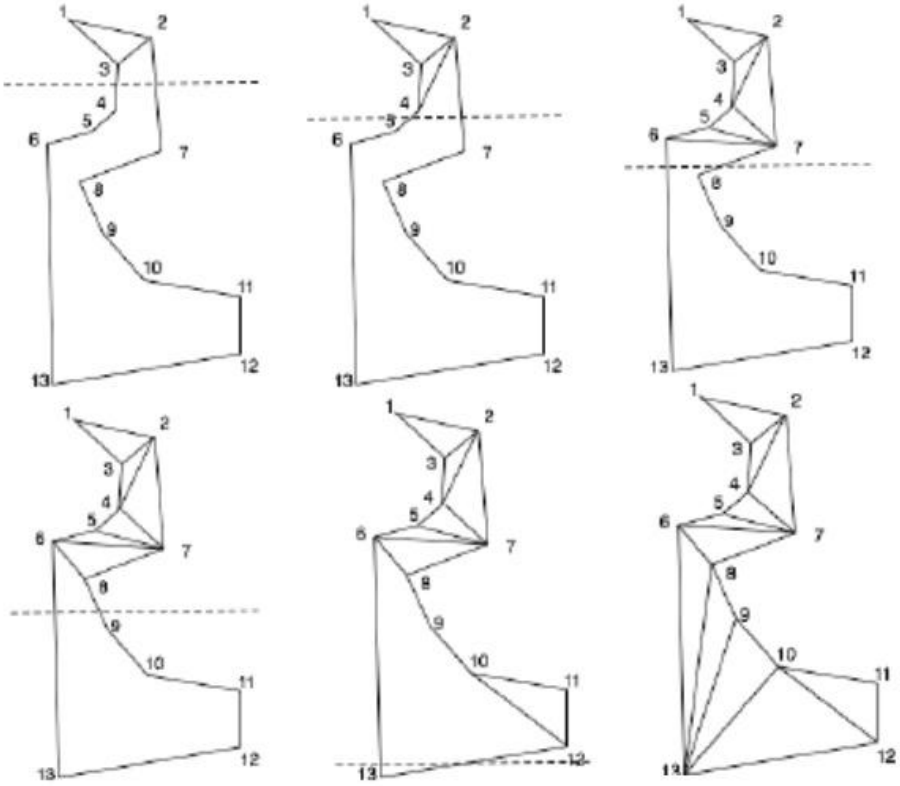
For $i \geq 2$, let v_i be the vertex just processed by the triangulation algorithm, then:

- The un-triangulated region above v_i consists of two y-monotone chains – a left and a right. Let u be the upper vertex that goes from v_i to u, then the chain from v_i to u, that chain is reflex (the interior angles are ≥ 180).
- That was the left chain; the right chain is the single diagonal from u to some point below v_i.

Proof:

Proof by induction: when we get to the first vertex v_i that is not the topmost vertex in the sorted list, it is trivial.

Assume correctness up until v_{i-1}. Cases: Complete explanation from the slides.



Implementation:

The vertices of the reflexed part of the (merged) chain are kept on a stack. As we go along we push any new reflexed vertices to the stack. When we connect v_i to the top of the stack and its follower, we simply pop them.

The algorithm is linear:

- The sorted list can be constructed by merging two chains - $O(n)$.
- For the total diagonals $n-3$ we do constant operations of orientation test to determine whether we add this diagonal (creating a triangle), and this diagonal is added in constant time as well (in a DECL).