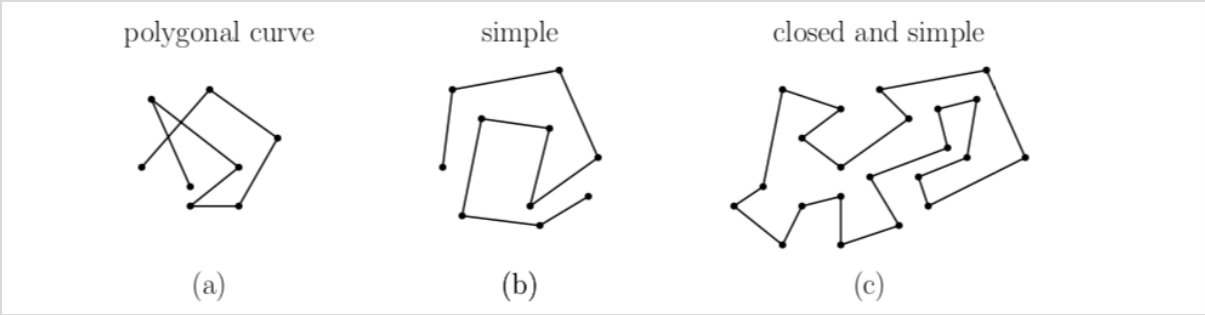
# CS 6301.002.20S Lecture 5—January 27, 2020

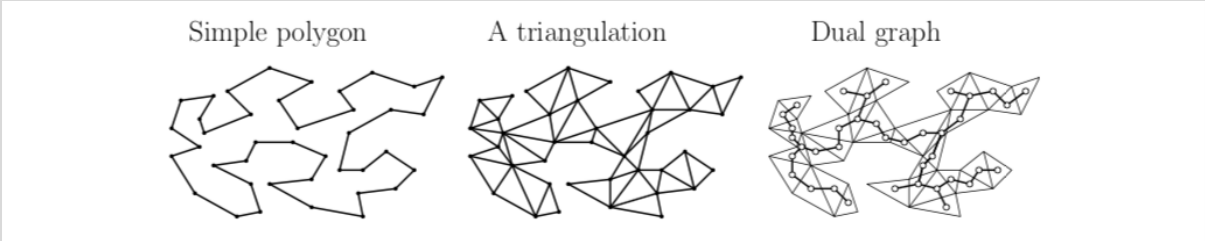
Main topics are #polygon_triangulation.

## Polygon Triangulation

- Today, and likely Tuesday, we'll consider the following problem. First some definitions.
- A *polygonal curve* is a sequence of line segments (edges) where two consecutive line segments share end and start *incident* vertices.
- The first and last vertex may be equal, making the curve *closed*.
- A polygonal curve is *simple* if no two elements intersect unless they are incidental.
- A *simple polygon* is a planar region bounded by a simple polygonal curve.



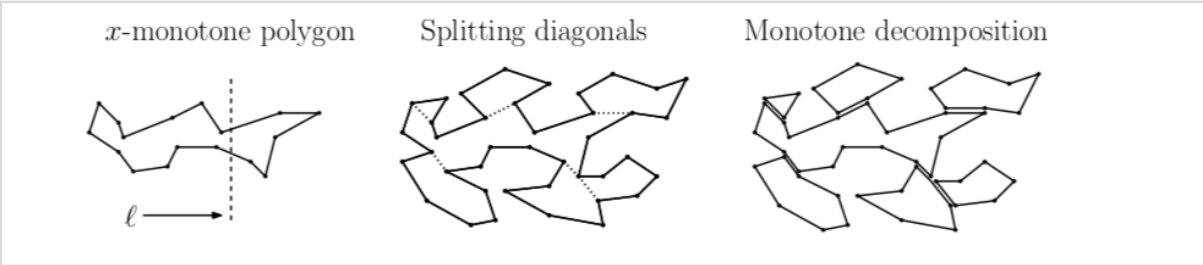

- We want to *triangulate* a polygon of n vertices/edges by subdividing it into triangles using new edges between its vertices.



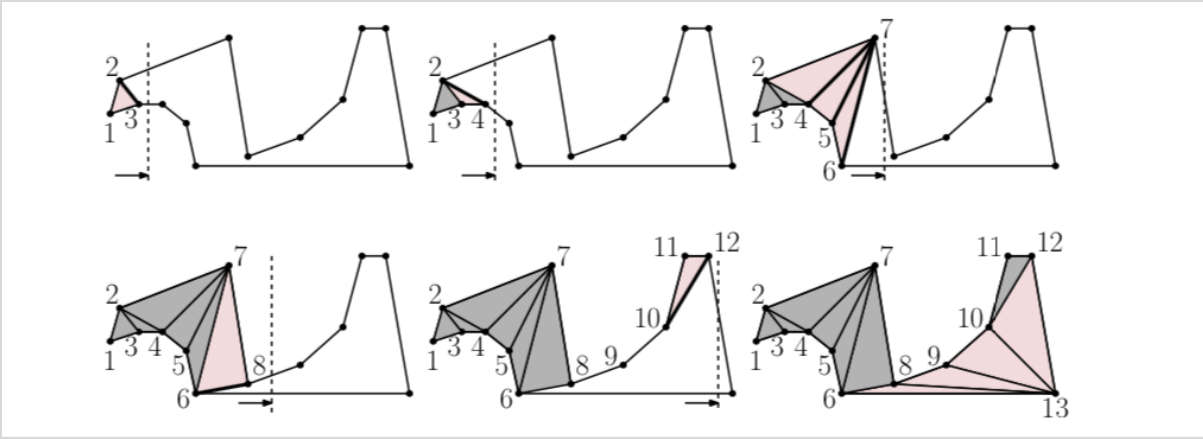

- The main motivation is that triangles are very simple objects, and breaking a region into simple objects is a useful first step in any kind of more complicated work.
- The triangulation also hints at a way to traverse the interior of a simple polygon. There's an object called the *dual graph* where there's one vertex per triangle and edges between adjacent triangles. It turns out this object is always a tree if you're working with the triangulation of a simple polygon.
- You may ask if there even exists a triangulation. Well, yes. Any polygon with at least four vertices has a diagonal between two of them that does not intersect any edge (you can find a proof in the book). You can split the polygon along this diagonal and then recursively triangulate the two halves. A simple inductive proof implies the whole thing has n - 2 triangles.
- But this algorithm is actually pretty slow since finding a diagonal might take awhile. Today, we'll discuss an O(n log n) time algorithm.

## Triangulation Strategy and Monotone Polygons

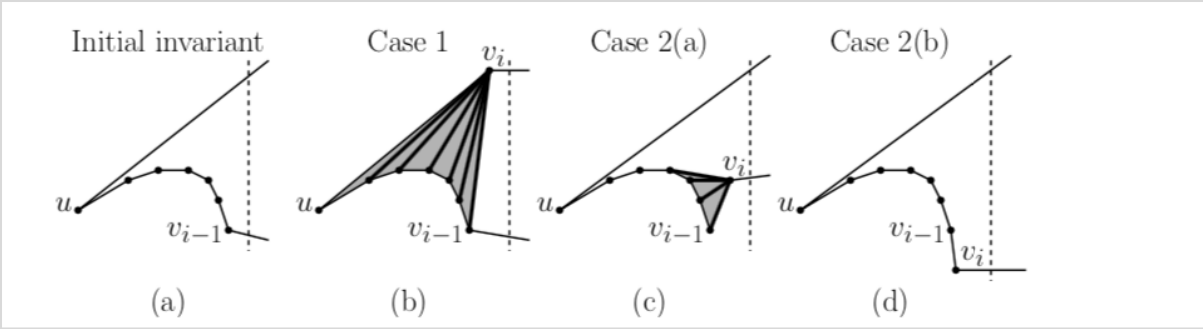
- We're going to do triangulations in two steps.
- First, we'll decompose the given simple polygon into a collection of simpler *monotone* polygons in O(n log n) time [Lee, Preparata '77].
- Then, we'll triangulate each monotone polygon separately and combine the results. This step takes only O(n) time [Garey et al. '78].
- Let's start with the second step, because it is easier and will help motivate why we go through the trouble of doing the first step.
- A polygonal curve C is *monotone* with respect to line ell if each line orthogonal to ell intersects C in one connected component. It is *strictly monotone* if each intersection is one point.
- A simple polygon P is *monotone* with respect to ell if its boundary can be split into two curves, each monotone with respect to ell.




- For lecture, we're going to consider polygons that are monotone with respect to the x-axis. These are called *horizontally monotone* or *x-monotone*.
- Now, suppose we have a horizontally monotone polygon. How are we going to triangulate it?
- Like last week, we'll use a plane sweep algorithm, sweeping a vertical line going left to right. I'll assume no two vertices share an x-coordinate.
- The main idea is that whenever the sweep line hits a vertex, we'll try to triangulate everything to the left of the sweep line that we can, and then split off the triangulated parts. You can think of this splitting off as pulling leaves out of that dual tree I mentioned earlier.
- So let's take at an example and see how this approach might look.



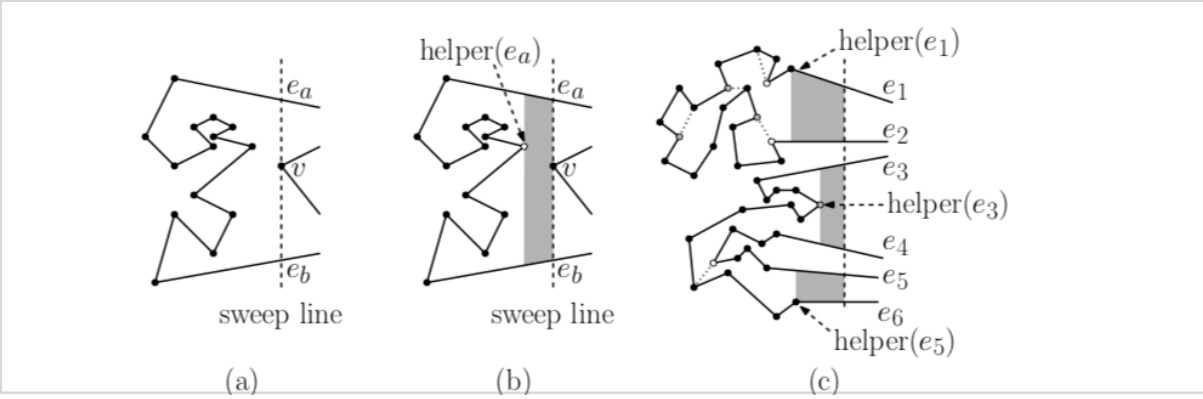

- Essentially, the untriangulated part (that we haven't split off yet) looks like a funnel turned on its side. One side of the funnel is part of a single edge. The other side is what we call a reflex chain.
- A *reflex vertex* is a vertex with interior angle greater than pi. A *reflex chain* is a polygonal chain along the polygon's boundary where internal vertices are reflex.
- We can formally state this observation about funnels as the following invariant:
- Invariant: Suppose the sweep line just processed vertex v. Let u be the leftmost vertex we haven't split off. The part of the polygon not split off and left of the sweep line consists of an upper and lower horizontally monotone chain. One is a reflex chain from v to u. The other is part of an edge from u to the sweep line.
- We'll now discuss the sweep line algorithm that maintains this invariant. The algorithm will store:
  - a stack S storing the vertices on the reflex chain with the rightmost vertex on top and u on the bottom
  - a flag saying whether the reflex chain is on the upper or lower part of the polygon
  - the vertex u farthest left in the untriangulated part of the polygon
- TriangulateMonotone(P):
  - Sort vertices from left to right as v_1, …, v_n by merging the upper and lower chain of P (P is monotone).
  - Push v_1 and v_2 onto stack.
  - For i ← 3 to n - 1
    - If v_i and S[top] lie on different chains
      - Pop all vertices from S
      - Insert diagonals from v_i to each popped vertex except u
      - Push v_{i-1} and v_i onto S (so u ← v_{i-1})
    - Else
      - While |S| ≥ 2 and S[top] is not reflex on chain <S[top - 1], S[top], v_i>.
        - Pop S[top]
        - Add diagonal to new S[top]
      - Push v_i onto S.




- In the first case, edge v_i u exists, so u is visible. The chain is reflex and horizontally monotone so the chain cannot block itself from v_i. Finally, no other part of the polygon can block visibility, because the polygon is horizontally monotone. As we add each diagonal left to right, we split off one triangle and former stack vertex.
- The second case is just like Graham's scan. As long as there's a right-hand turn, we can see S[top - 1] and safely add a diagonal, splitting off S[top]'s new triangle in the process.
- Sorting by merging takes O(n) time. We spend constant time per vertex reached by the sweep line plus diagonal added for O(n) time total.

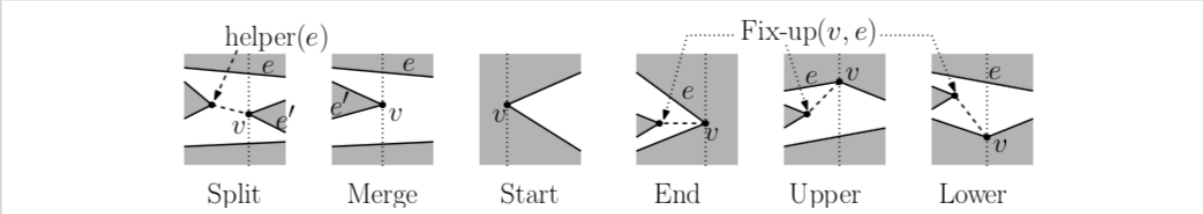
## Monotone Subdivision

- Now we need an algorithm to split an arbitrary simple polygon into monotone polygons. Again, we'll use a plane sweep algorithm, sweeping from left to right.
- The algorithm is largely based on the following observation: A polygon is not horizontally monotone if and only if there is a *scan reflex vertex*, a reflex vertex where both incident edges go left or both incident edges go right.
- The book calls the first kind *merge vertices* and the second kind *split vertices*. The goal is for the sweep line to discover these two kinds of vertices, and add diagonals to them when possible.
- For merge vertices, we'll need to wait for some point in the future to add the diagonal.
- For split vertices, though, we need to add the diagonal immediately to some vertex further left. But how will we know where to add that diagonal?
- Consider a split vertex v. There's an edge e_a immediately above it and an edge e_b immediately below. If we sweep *left* from v, we'll encounter some first vertex below e_a that is visible.
- helper(e_a): Suppose the polygon interior is below e_a. Let e_b be the edge of the polygon just below e_a on the sweep line. helper(e_a) is the rightmost vertex u left of the sweep line such that the vertical segment between e_a and u is entirely in the polygon.
- Again, helper(e_a) is only defined if the polygon interior is below e_a. The helper could be incident to e_a or e_b or neither.




- See how we can take a segment up to e_a and then down-left to helper(e_a)? That whole triangle lies in the polygon, so it's safe to send a diagonal from v back to helper(e_a).
- Our goal will be to add diagonals from each split vertex back to a helper vertex. We'll also keep track of helper merge vertices, since we can easily send diagonals back to them, even from vertices that aren't scan reflex.
- Let's formally define the sweep status. We'll store edges intersected by the sweep line with the polygon interior below them along with their helpers. We'll store them top to bottom in an ordered dictionary again, so we can easily figure out what e_a and helper(e_a) are relevant for each event.
- We'll be trying to help out merge vertices whenever we can, so we'll use a subroutine Fix-up(v, e) where a) e is above v or incident going left and b) the polygon interior is below e: if helper(e) is a merge vertex, add diagonal from v to helper(e).
- We call Fix-up(v, e) whenever processing v will cause e's helper to change or go away entirely.
- Now we just need to deal with lots of cases!
- MakeMonotone(P):
  - Sort vertices from left to right as v_1, …, v_n.
  - For i ← 1 to n:
    - v ← v_i
    - If v is a split vertex
      - Find edge e immediately above v on sweep line.
      - Add diagonal to helper(e).
      - Let e' be the lower edge incident to v.
      - Add e' to sweep line status.
      - Make v the helper of e and e'.
    - If v is a merge vertex
      - Let e' be the lower edge incident to v.
      - Delete e' from status.
      - Fix-up(v, e')
      - Find edge e immediately above v on sweep line.
      - Fix-up(v, e).

    - helper(e) ← v
- If v is a *start vertex* (both incident edges go right and v is not reflex)
    - Let e be the higher incident edge on v.
    - Insert e into  status and set helper(e) ← v.
- If v is an *end vertex* (both incident edges go left and v is not reflex)
    - Let e be the higher incident edge on v.
    - Fix-up(v, e)
    - Delete e from status.
- If v is an *upper-chain vertex* (incident edges go both directions and interior is below v)
    - Let e be incident edge left of v and e' be the incident edge to the right.
    - Fix-up(v, e)
    - Replace e with e' in sweep line status and set helper(e') ← v.
- If v is a *lower-chain vertex*
    - Find edge e immediately above v on sweep line.
    - Fix-up(v, e)
    - helper(e) ← v




- By only going for helpers of edges immediately above each v, we safely add diagonals.
- We add diagonals going left for every split vertex, and we make sure to add a diagonal to helper merge vertices before their edges go away or they are replaced by new helpers. So the resulting polygons are horizontally monotone.
- Finally, each event can be handled in O(log n) time and there are n events, so the whole thing + sorting takes O(n log n) time.