CMSC 754: Lecture 5
Polygon Triangulation

Reading: Chapter 3 in the 4M's.

The Polygon Triangulation Problem: Triangulation is the general problem of subdividing a spatial domain into simplices, which in the plane means triangles. We will focus in this lecture on triangulating a *simple polygon* (see Fig. 1). Formal definitions will be given later. (We will assume that the polygon has no holes, but the algorithm that we will present can be generalized to handle such polygons.) Such a subdivision is not necessarily unique, and there may be other criteria to be optimized in computing the triangulation.

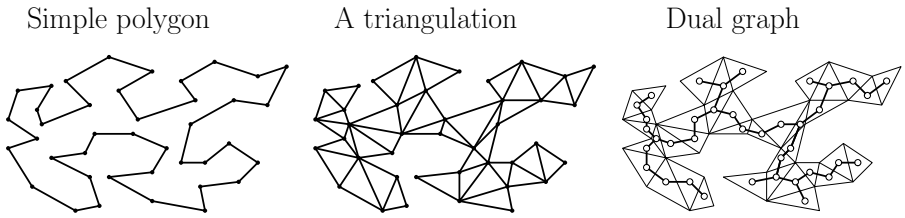

Fig. 1: Polygon triangulation.

Applications: Triangulating simple polygons is important for many reasons. This operation useful, for example, whenever it is needed to decompose a complex shapes a set of disjoint simpler shapes. Note that in some applications it is desirable to produce "fat" (nearly equilateral) triangles, but we will not worry about this issue in this lecture.

A triangulation provides a simple graphical representation of the polygon's interior, which is useful for algorithms that operate on polygons. In particular, consider a graph whose vertices are the triangles of the triangulation and two vertices of this graph are adjacent if the associated triangles are adjacent (see Fig. 1(c)). This is called the *dual graph* of the triangulation. It is easy to show that such a graph is a *free tree*, that is, it is an acylic, connected graph. (If the polygon has holes, then the dual graph will generally have cycles.)

Preliminaries: This simple problem has been the focus of a remarkably large number of papers in computational geometry spanning a number of years. There is a simple naive polynomial-time algorithm for the planar case (as opposed to possibly nonconvex polyhedra in higher dimensions). The idea is based on repeatedly adding "diagonals." We say that two points on the boundary of the polygon are *visible* if the interior of the line segment joining them lies entirely within the interior of the polygon. Define a *diagonal* of the polygon to be the line segment joining any pair of visible vertices.

Observe that the addition of a diagonal splits the polygon into two polygons of smaller size. In particular, if the original polygon has n vertices, the diagonal splits the polygon into two polygons with n_1 and n_2 vertices, respectively, where $n_1, n_2 < n$, and $n_1 + n_2 = n + 2$. Any simple polygon with at least four vertices has at least one diagonal. (This seemingly obvious fact is not that easy to prove. You might try it.) A simple induction argument shows that the final number of diagonals is $n - 3$ and the final number of triangles is $n - 2$.

The naive algorithm operates by repeatedly adding diagonals. Unfortunately, this algorithm is not very efficient (unless the polygon has special properties, for example, convexity) because of the complexity of the visibility test.

There are very simple $O(n \log n)$ algorithms for this problem that have been known for many years. A longstanding open problem was whether there exists an $O(n)$ time algorithm. (Observe that the input polygon is presented as a cyclic list of vertices, and hence the data is in some sense "pre-sorted", which precludes an $\Omega(n \log n)$ lower bound.) The problem of a linear time polygon triangulation was solved by Bernard Chazelle in 1991, but the algorithm (while being a technical tour de force) is so complicated that it is not practical for implementation. Unless other properties of the triangulation are desired, the $O(n \log n)$ algorithm that we will present in this lecture is quite practical and probably preferable in practice to any of the "theoretically" faster algorithms.

A Triangulation in Two Movements: Our approach is based on a two-step process (although with a little cleverness, both steps could be combined into one algorithm).

- First, the simple polygon is decomposed into a collection of simpler polygons, called *monotone polygons*. This step takes $O(n \log n)$ time.
- Second, each of the monotone polygons is triangulated separately, and the result are combined. This step takes $O(n)$ time.

The triangulation results in a planar subdivision. Such a subdivision could be stored as a planar graph or simply as a set of triangles, but there are representations that are more suited to representing planar subdivisions. One of these is called *double-connect edge list* (or DCEL). This is a linked structure whose individual entities consist of the vertices (0-dimensional elements), edges (1-dimensional elements), triangular faces (2-dimensional elements). Each entity is joined through links to its neighboring elements. For example, each edge stores the two vertices that form its endpoints and the two faces that lie on either side of it.

We refer the reader to Chapter 2 of our text for a more detailed description of the DCEL structure. Henceforth, we will assume that planar subdivisions are stored in a manner than allows local traversals of the structure to be performed $O(1)$ time.

Monotone Polygons: Let's begin with a few definitions. A *polygonal curve* is a collection of line segments, joined end-to-end (see Fig. 2(a)). If the last endpoint is equal to the first endpoint, the polygonal curve is said to be *closed*. The line segments are called *edges*. The endpoints of the edges are called the *vertices* of the polygonal curve. Each edge is *incident* to two vertices (its endpoints), and each vertex is incident (to up) two edges. A polygonal curve is said to be *simple* if no two nonincident elements intersect each other (see Fig. 2(c)). A closed simple polygonal curve decomposes the plane into two parts, its *interior* and *exterior*. Such a polygonal curve is called a *simple polygon* (see Fig. 2(c)). When we say "polygon" we mean simple polygon.

A polygonal curve C is *monotone* with respect to ℓ if each line that is orthogonal to ℓ intersects C in a single connected component. (It may intersect, not at all, at a single point, or along a single line segment.) A polygonal curve C is said to be *strictly monotone* with respect to a given line ℓ, if any line that is orthogonal to ℓ intersects C in at most one point. A simple

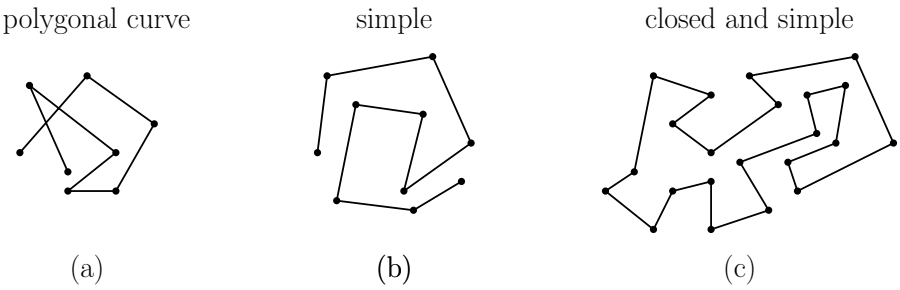

Fig. 2: Polygonal curves and simple polygons.

polygon P is said to be *monotone* with respect to a line ℓ if its boundary, (sometimes denoted bnd(P) or ∂P), can be split into two curves, each of which is monotone with respect to ℓ (see Fig. 3(a)).

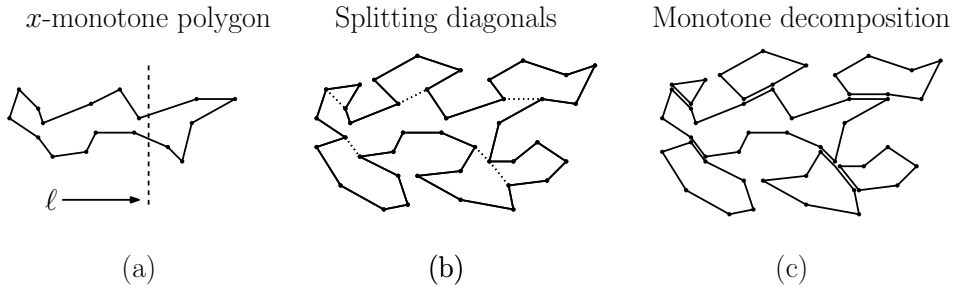

Fig. 3: Monotonicity.

Henceforth, let us consider monotonicity with respect to the x-axis. We will call these polygons *horizontally monotone*. It is easy to test whether a polygon is horizontally monotone. How?

(a) Find the leftmost and rightmost vertices (min and max x-coordinate) in $O(n)$ time.

(b) These vertices split the polygon's boundary into two curves, an *upper chain* and a *lower chain*. Walk from left to right along each chain, verifying that the x-coordinates are nondecreasing. This takes $O(n)$ time.

(As an exercise, consider the problem of determining whether a polygon is monotone in *any* direction. This can be done in $O(n)$ time.)

Triangulation of Monotone Polygons: We begin by showing how to triangulate a monotone polygon by a simple variation of the plane-sweep method. We will return to the question of how to decompose a polygon into monotone components later.

We begin with the assumption that the vertices of the polygon have been sorted in increasing order of their x-coordinates. (For simplicity we assume no duplicate x-coordinates. Otherwise, break ties between the upper and lower chains arbitrarily, and within a chain break ties so that the chain order is preserved.) Observe that this does not require sorting. We can simply extract the upper and lower chain, and merge them (as done in MergeSort) in $O(n)$

time. Let's make the usual general position assumptions, that no two vertices have the same x-coordinates and no three consecutive vertices are collinear.

We define a *reflex vertex* to be a vertex of the polygon whose interior angle is at least π, and otherwise the vertex is *nonreflex*. We define a *reflex chain* to be a sequence of one or more consecutive reflex vertices along the polygon's boundary.

The idea behind the triangulation algorithm is quite simple: Try to triangulate *everything* you can to the *left* of the current vertex by adding diagonals, and then remove the triangulated region from further consideration. The trickiest aspect of implementing this idea is finding a clean invariant that characterizes the *untriangulated region* that lies to the left of the sweep line.

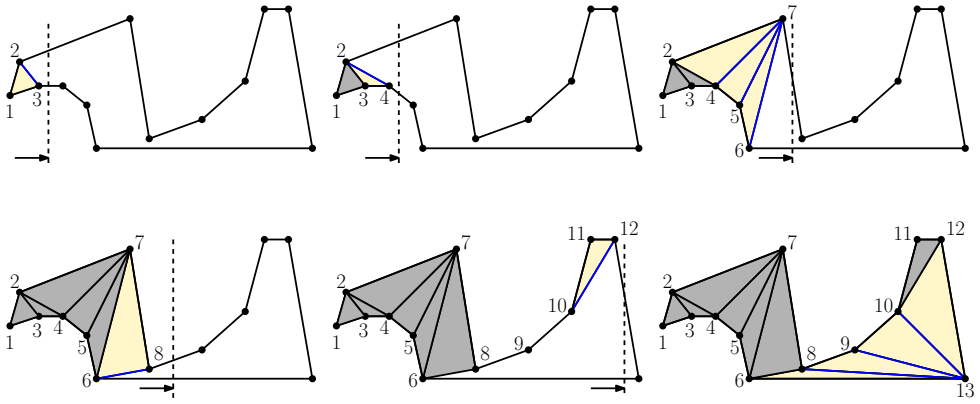

Fig. 4: Triangulating a monotone polygon.

To acquire some intuition, let's consider the example shown in Fig. 4. There is obviously nothing to do until we have at least three vertices. With vertex 3, it is possible to add the diagonal to vertex 2, and so we do this. In adding vertex 4, we can add the diagonal to vertex 2. However, vertices 5 and 6 are not visible to any other nonadjacent vertices so no new diagonals can be added. When we get to vertex 7, it can be connected to 4, 5, and 6. The process continues until reaching the final vertex.

Have we seen enough to conjecture what the untriangulated region to the left of the sweep line looks like? Ideally, this structure will be simple enough to allow us to determine in *constant time* whether it is possible to add another diagonal. And in general we can add each additional diagonal in constant time. Since any triangulation consists of $n-3$ diagonals, the process runs in $O(n)$ total time. This structure is described in the lemma below.

Lemma: (*Main Invariant*) For $i \geq 2$, let v_i be the vertex just processed by the triangulation algorithm. The untriangulated region lying to the left of v_i consists of two x-monotone chains, a lower chain and an upper chain each containing at least one edge. If the chain from v_i to u has two or more edges, then these edges form a reflex chain. The other chain consists of a single edge whose left endpoint is u and whose right endpoint lies to the right of v_i (see Fig. 5(a)).

We will prove the invariant by induction, and in the process we will describe the triangulation

algorithm. As the basis case, consider the case of v_2. Here $u = v_1$, and one chain consists of the single edge v_2v_1 and the other chain consists of the other edge adjacent to v_1. To complete the proof, we will give a case analysis of how to handle the next event, involving v_i, assuming that the invariant holds at v_{i-1}, and see that the invariant is satisfied after each event has been processed. There are the following cases that the algorithm needs to deal with.

Case 1: v_i lies on the opposite chain from v_{i-1}: In this case we add diagonals joining v_i to all the vertices on the reflex chain, from v_{i-1} back to (but not including) u (see Fig. 5(b)). Note that all of these vertices are visible from v_i. Certainly u is visible to v_i. Because the chain is reflex, x-monotone, and lies to the left of v_i it follows that the chain itself cannot block the visibility from v_i to some other vertex on the chain. Finally, the fact that the polygon is x-monotone implies that the unprocessed portion of the polygon (lying to the right of v_i) cannot "sneak back" and block visibility to the chain.

After doing this, we set $u = v_{i-1}$. The invariant holds, and the reflex chain is trivial, consisting of the single edge v_iv_{i-1}.

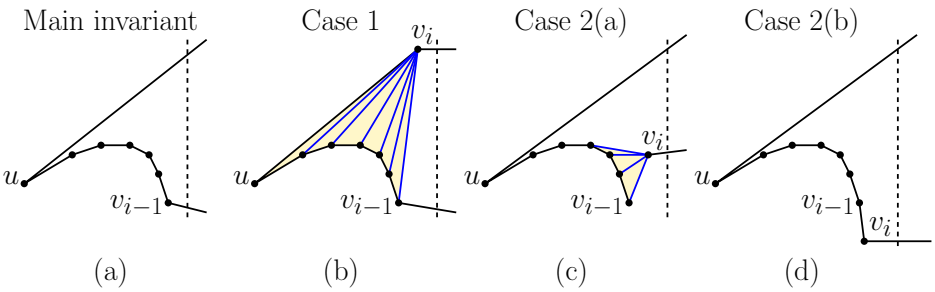

Fig. 5: Triangulation cases.

Case 2: v is on the same chain as v_{i-1}. There are two subcases to be considered:

Case 2(a): The vertex v_{i-1} is a nonreflex vertex: We walk back along the reflex chain adding diagonals joining v_i to prior vertices until we find the last vertex v_j of the chain that is visible to v_i. As can be seen in Fig. 5(c), this will involve connecting v_i to one or more vertices of the chain. Remove these vertices from v_{i-1} back to, but not including v_j from the reflex chain. Add v_i to the end of reflex chain. (You might observe a similarity between this step and the inner loop of Graham's scan.)

Case 2(b): The vertex v_{i-1} is a reflex vertex. In this case v_i cannot see any other vertices of the chain. In this case, we simply add v_i to the end of the existing reflex chain (see Fig. 5(d)).

In either case, when we are done the remaining chain from v_i to u is a reflex chain.

How is this implemented? The vertices on the reflex chain can be stored in a stack. We keep a flag indicating whether the stack is on the upper chain or lower chain, and assume that with each new vertex we know which chain of the polygon it is on. Note that decisions about visibility can be based simply on orientation tests involving v_i and the top two entries on the stack. When we connect v_i by a diagonal, we just pop the stack.

Analysis: We claim that this algorithm runs in $O(n)$ time. As we mentioned earlier, the sorted list of vertices can be constructed in $O(n)$ time through merging. The reflex chain is stored on a stack. In $O(1)$ time per diagonal, we can perform an orientation test to determine whether to add the diagonal and the diagonal can be added in constant time. Since the number of diagonals is $n - 3$, the total time is $O(n)$.

Monotone Subdivision: In order to run the above triangulation algorithm, we first need to subdivide an arbitrary simple polygon P into monotone polygons. This is also done by a plane-sweep approach. We will add a set of nonintersecting diagonals that partition the polygon into monotone pieces (recall Fig. 3).

Observe that the absence of x-monotonicity occurs only at vertices in which the interior angle is greater than 180° and both edges lie either to the left of the vertex or both to the right. We call such a vertex a *scan reflex vertex*. Following our book's notation, we call the first type a *merge vertex* (since as the sweep passes over this vertex the edges seem to be merging) and the latter type a *split vertex*.

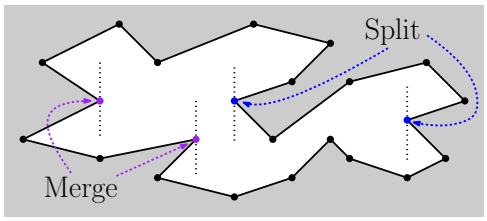

Fig. 6: Merge and split reflex vertices.

Our approach will be to apply a left-to-right plane sweep (see Fig. 7(a)), which will add diagonals to all the split and merge vertices. We add a diagonal to each split vertex as soon as we reach it. We add a diagonal to each merge vertex when we encounter the next visible vertex to its right.

The key is storing enough information in the sweep-line status to allow us to determine where this diagonal will go. When a split vertex v is encountered in the sweep, there will be an edge e_a of the polygon lying above and an edge e_b lying below. We might consider attaching the split vertex to left endpoint of one of these two edges, but it might be that neither endpoint is visible to the split vertex. Instead, we need to maintain a vertex that is visible to any split vertex that may arise between e_a and e_b. To do this, imagine a sweeping a vertical segment between e_a and e_b to the left until it hits a vertex. Called this helper(e_a) (see Fig. 7(b)).

helper(e_a) : Let e_b be the edge of the polygon lying just below e_a on the sweep line. The helper is the rightmost vertically visible vertex on or below e_a on the polygonal chain between e_a and e_b. This vertex may either be on e_a, e_b, or it may lie between them.

Another way to visualize the helper is to imagine sweeping out a trapezoid to the left from the sweep line. The top side of the trapezoid lies on e_a, the bottom side lies on e_b, the right side lies on the sweep line, and the left side is sweeps as far as it can until hitting a vertex (see the shaded regions of Figs. 7(b) and (c)).

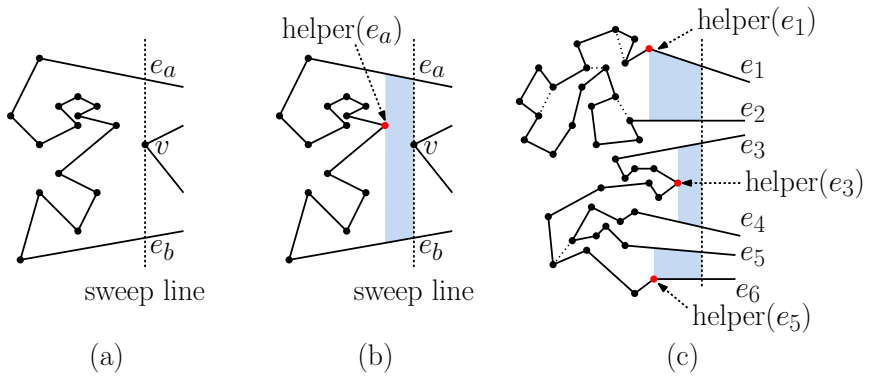

Fig. 7: Split vertices, merge vertices, and helpers.

Observe that helper(e_a) is defined with respect to the current location of the sweep line. As the sweep line moves, its value changes. The helper is defined only for those edges intersected by the sweep line. Our approach will be to join each split vertex to helper(e_a), where e_a is the edge of P immediately above the split vertex. (Note that it is possible that the helper is the left endpoint of e_a.) When we hit a merge vertex, we cannot add a diagonal right away. Instead, our approach is to take note of any time a helper is a merge vertex. The diagonal will be added when the very next visible vertex is processed.

Events: The endpoints of the edges of the polygon. These are sorted by increasing order of x-coordinates. Since no new events are generated, the events may be stored in a simple sorted list (i.e., no priority queue is needed).

Sweep status: The sweep line status consists of the list of edges that intersect the sweep line, sorted from top to bottom. (Our book notes that we actually only need to store edges such that the interior of the polygon lies just below this edge, since these are the only edges that we evaluate helper from.)

These edges are stored in a dictionary (e.g., a balanced binary tree), so that the operations of insert, delete, find, predecessor and successor can be evaluated in $O(\log n)$ time each.

Event processing: There are six event types based on a case analysis of the local structure of edges around each vertex. Let v be the current vertex encountered by the sweep (see Fig. 8). Recall that, whenever we see a split vertex, we add a diagonal to the helper of the edge immediately above it. We defer adding diagonals to merge vertices until the next opportunity arises. To help with this, we define a common action called "fix-up." It is given a vertex v and an edge e (either above v or incident to its left). The fix-up function adds a diagonal to helper(e), if helper(e) is a merge vertex.

fix-up(v, e)**:** If helper(e) is a merge vertex, add a diagonal from v to this merge vertex.

Split vertex(v)**:** Search the sweep line status to find the edge e lying immediately above v. Add a diagonal connecting v to helper(e). Add the two edges incident to v into the sweep line status. Let e' be the lower of these two edges. Make v the helper of both e and e'.

Merge vertex(v)**:** Find the two edges incident to this vertex in the sweep line status (they must be adjacent). Let e' be the lower of the two. Delete them both. Let e be the edge lying immediately above v. fix-up(v, e) and fix-up(v, e'). Set the helper of e to v.

Start vertex(v)**:** (Both edges lie to the right of v, but the interior angle is smaller than π.) Insert this vertex's edges into the sweep line status. Set the helper of the upper edge to v.

End vertex(v)**:** (Both edges lie to the left of v, but the interior angle is larger than π.) Let e be the upper of the two edges. fix-up(v, e). Delete both edges from the sweep line status.

Upper-chain vertex(v)**:** (One edge is to the left, and one to the right, and the polygon interior is below.) Let e be the edge just to the left of v. fix-up(v, e). Replace the edge to v's left with the edge to its right in the sweep line status. Make v the helper of the new edge.

Lower-chain vertex(v)**:** (One edge is to the left, and one to the right, and the polygon interior is above.) Let e be the edge immediately above v. fix-up(v, e). Replace the edge to v's left with the edge to its right in the sweep line status. Make v the helper of the new edge.

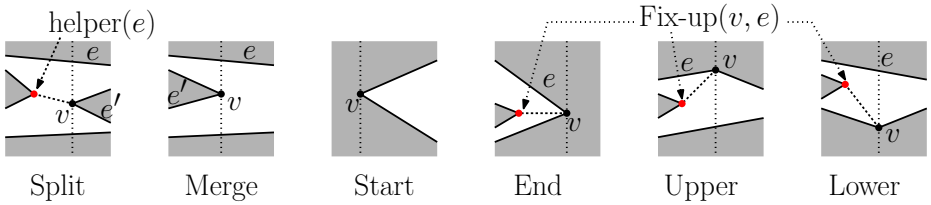

Fig. 8: Plane sweep cases, where v is the vertex being swept. The label e denotes the edge such that helper$(e) \leftarrow v$.

Correctness: Given the number of cases, establishing correctness is a bit of a pain. We will refer you to the 4M's book for a careful proof, but here are main points that need to be established in the proof.

Helpers are correctly updated: Immediately after processing each event, the helpers of all relevant edges have been properly updated.

Merge vertices are correctly resolved: Whenever we encounter a merge vertex, we add a diagonal to resolve this non-monotonicity.

Split vertices are correctly resolved: When a split vertex is visited, it becomes a helper of the edge e immediately above. We will resolve this non-monotonicity when e's helper changes by the invocation of fix-up.

Added diagonals do not intersect each other: Added diagonals lie within a single "helper trapezoid" which has no vertices except on its left and right vertical sides. If both of these vertices are scan reflex vertices (merger on the left and split on the right) we will add a single diagonal to resolve both (see the Split case of Fig. 8).

Analysis: Given a simple polygon with n vertices, there are n events, one for each vertex. Each event involves a constant amount of processing and a constant number of accesses to the sweep-line dictionary. Thus, the time per event is $O(\log n)$, and hence the overall time is $O(n \log n)$. We have the following:

Theorem: Given an n-vertex simple polygon, in $O(n \log n)$ time, the above sweep-line algorithm correctly add diagonals to decompose it into monotone pieces.

By combining this with the $O(n)$ time algorithm for triangulating a monotone polygon, we obtain the following result.

Theorem: Given a simple polygon with n vertices, it is possible to triangulate it in $O(n \log n)$ time.