Lecture IV
PLANE SWEEP AND REGULARIZATION

In the previous chapter, we considered geometric relationships among points. Such relationships are defined by predicates such as `LeftTurn`(p, q, r). We now consider line segments and points, and their relationships. An obvious predicate here whether two segments intersect. If we consider the relationships between points and segments, we get predicates such as incidence (is a point on a segment?) and sided-ness (is a point to the left of a segment?).

We study some computational problems arising from such relationships in the plane. A fundamental computational technique, the "plane sweep" is introduced. This computational technique is intimately related to a data structure called the **trapezoidal map**. In applications, segments are typically organized to form more complex structures, for example, a subdivision of the plane into polygonal regions. We introduce a hierarchy of problems (collectively called Segment Regularization Problems) which are of practical importance. The plane sweep technique and the trapezoidal map turns out to be the key for a uniform solution framework.

[Move to next lecture:] Computing, manipulating and searching such subdivisions is non-trivial and various data structures have been invented to represent them.

§1. Segment Intersection

We begin with the simplest computational problem for segments. Let S be a set of n closed segments. The **segment intersection problem** is to compute all the pairwise segment intersections of S. There is a trivial $O(n^2)$ algorithm, and in the terms of the standard input-size measure of complexity, this is worst-case optimal. But this is not the end of our story.

Output Sensitivity. We introduce an algorithmic idea that has widely applicable in computaional geometry: **output sensitive algorithms**. Let $I = I(S)$ be the number of pairwise intersections among the segments in S. We call I the **output size** for the instance S. Clearly,

$$0 \leq I(S) \leq \binom{n}{2}.$$

An output sensitive algorithm is one whose complexity depnds on $n = |S|$ as well as on $I = I(S)$. In view of the naive algorithm, we seek a time complexity $T(n)$ of the form

$$T(n) = f(n) + g(n, I)$$

where $f(n) = o(n^2)$ and $g(n, I)$ to be not much worst than n^2. In particular, we will describe an algorithm with complexity $T(n) = O(n + I)\log n$. This algorithm is $\Theta(n^2 \log n)$ in the worst case, but when $I = o(n^2/\log n)$, it is better than the naive one.

Sweepline Algorithm. We will present output-sensitive algorithm from Bentley and Ottmann (1979). It uses a well-known "sweepline" paradigm in geometric algorithms. Imagine a horizontal line H that is sweeping from the top ($y = +\infty$) to bottom ($y = -\infty$). For any real value t, let $H(t)$ denote the horizontal line $\{y = t\}$. We think of t as "time" although you may note that this interpretation is peculiar since t is actually decreasing.

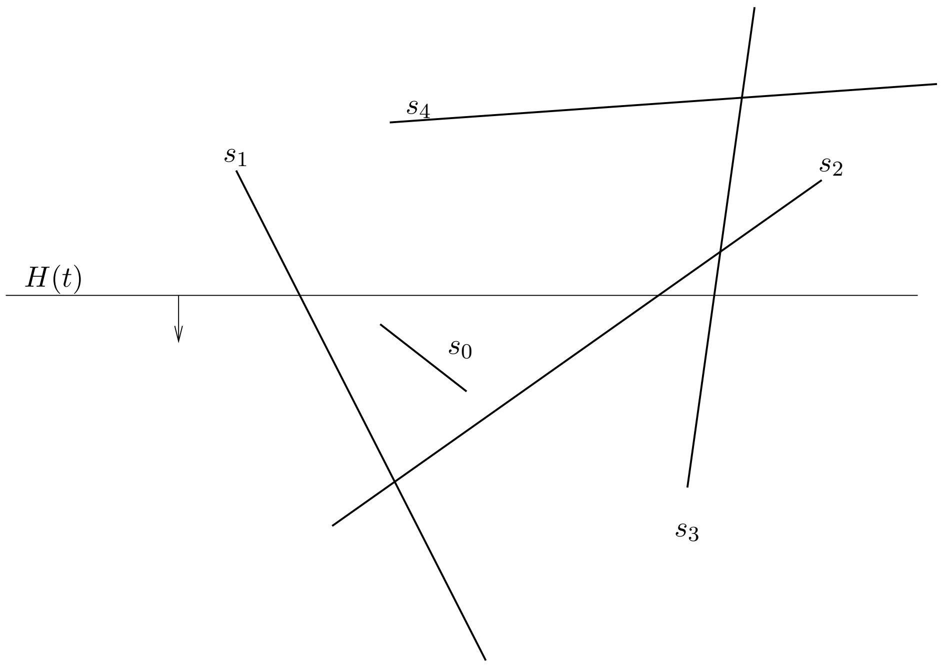

Figure 1: A sweepline H sweeping a set of 4 segments.

Sometimes we just want to know for an input set S of segments Let $S(t) \subseteq S$ denote the set of segments in S that intersects $H(t)$. In figure 1, we see that $S(t) = \{s_1, s_2, s_3\}$. The intersection induces a linear ordering "$\prec_t$" on $S(t)$, where each $s \in S(t)$ is ordered by the x-coordinate of the intersection point $s \cap H(t)$. Thus, in figure 1, we have

$$s_1 \prec_t s_2 \prec_t s_3.$$

As time t increases, the set $S(t)$ or the order $\prec_t$ changes only at the certain **critical times**: these are times that correspond to the y-coordinates of segment endpoints or the intersection of two segments. Let us therefore define a **critical point** of S to be either a segment endpoint or the intersection of two segments of S. Two segments s, s' intersect **properly** if $s \cap s'$ is a single point p; moreover, either p is an endpoint of s and s', or p is in the interior of s and s'. Thus p is either an **endpoint intersection** or an **interior intersection**.

For simplicity, we make several "non-degeneracy" assumptions on S. Degeneracy means the presence of any of the following condition: (1) two critical points of S have the same y-coordinate, (2) A pair of segments intersect improperly, and (3) three segments intersect at a point. Note that we allow two or more segments to share a common endpoint.

Tracking changes in the set $S(t)$. The two endpoints of a segment $[p, q]$ are called its **start end** and **stop end**, based on the order in which the sweepline H encounters them. Let t_0 be a critical time. If t_0 is the y-coordinate of a start end of segment s, then $s \in S(t_0^+) \setminus S(t_0^-)$. If t_0 is the y-coordinate of a stop end of segment s, then $s \in S(t_0^-) \setminus S(t_0^+)$. Note that several changes simultaneous can occur in the transition from $S(t_0^-)$ to $S(t_0^+)$.

Event Queue. In order to update $S(t)$ as t increases, we need a priority queue Q data structure to store the endpoints of segments in S. In general, the items in Q can be viewed as **events**, and we are handling events based on their priorities. If $s = [p, q]$ then intuitively, we want to store p and q as events, using their y-coordinates as priorities. However, we will represent these events as "start(s)" and "stop(s)", respectively. They are called the **start** and **stop events** corresponding to s. If $p.y > q.y$, then the start event will have priority $p.y$, and stop event will have priority $q.y$. Note that Q should be a "max-priority queue" as opposed to a "min-priority queue".

Tracking changes in the ordering $\prec_t$. We will represent the linear ordering $\prec_t$ of the set $S(t)$ by storing $S(t)$ in the leaves of some efficient binary tree data structure T. "Efficient" here could mean any balanced binary tree or splay tree. We may assume that the elements of $S(t)$ are stored only at the leaves of T.

The linear ordering $\prec_t$ of $S(t)$ may change even when $S(t)$ does not change. This happens when $H(t)$ passes through an interior intersection of two segments. If the segments are s, s', then their relative order will be reversed as $H(t)$ passes through $s \cap s'$. Thus we would like to store such proper intersections in Q as well. *There is a problem.* These are the points that our algorithm is supposed to compute in the first place! We are stuck in a circular reasoning if we are not careful. The trick is to only store in Q some of these proper intersections. In particular, for each pair $s, s' \in S(t)$ such that s and s' are adjacent in the linear ordering $<_t$, if s and s' define an interior intersection point p, then we store an event "$swap(s, s')$" in the queue Q with priority equal to $p.x$. It is easy to prove that before t sweeps pass the critical time $p.x$, the pair (s, s') has already been inserted into Q. This is intuitively clear.

Event Handling.

- Insert Event: We need to insert a segment s into the tree T. Suppose s appears between two existing ones s_1, s_2. We are then obligated to check for interior intersections of s with s_1 and s_2. These intersections will be inserted into the queue Q if found.
- Delete Event: After deletion, we have to check if the newly adjacent pair of segments intersect and to insert Q if necessary.
- Swap Event: in this case, we locate in T the two segments s, s' whose intersection cause the swap event. Then we swap them (the rest of T need not change). Again, there may be events to be inserted into Q.

Note that even with our non-degeneracy assumptions, there may be several events with the same priority in Q. But it is easy to see that we do not care about the order in which we handle events with the same priority.

Implementing the search tree T. As in the circular sorting problem in the previous lecture, the "key comparisons" that we make when searching in T are some abstraction of the comparison of numbers. But something interesting is also going on. Assume that the keys stored in T are the segments. When we make a comparison between two segments s and s', we need to specify a third parameter, some time t_0. This comparison reduces to comparing the x-coordinates of $s \cap H(t_0)$ and $s' \cap H(t_0)$. Providing this parameter t_0 is essential, because with some choices of t_0, we might get the wrong comparison result. The search tree T is properly ordered relative to some value t_0. Thus, we need to store t_0 with the tree T. Suppose we are processing an event at time t_1. What should the value of t_0 be as we search in T? In the case of a swap event we can set t_0 to be the average of t_1 and t'_1 where t'_1 is the critical time before t_1. In the case of a start or stop event, the choice $t_0 = t_1$ will be adequate.

Complexity. Assuming that the two data structures are implemented by some kind of balanced or amortized binary trees, we see that each event can be processed in $O(\log n)$ time. Hence the overall time complexity is $O(n + I)\log n)$, as claimed.

What about space? The size of T is $O(n)$ and the size of Q is $O(n + I)$. Indeed, in our event handling above, we did not mention the need to delete swap events from Q that are no longer represent adjacent pairs in T. But we can easily perform such deletions to ensure that the size of Q is $O(n)$. Notice that deleted swap events may be re-introduced later. For instance, in figure 1, we see that the swap event (s_1, s_2) will be deleted just as we read the start end of s_0, and again inserted as we sweep through the stop end s_0.

whether S has any interior intersections, $I(S) > 0$ or not. But it is easy to modify the algorithm to solve this "intersection detection problem" in $O(n \log n)$ time.

§2. Cell Complexes and Line Segments

Since we will be discussing several problems with line segments, it is useful to unify some definitions and concepts. Other issues such as degeneracies, which we sidestep above, are also discussed here.

We use the term **segment** to refer to any non-empty connected subset of a line. For the present, the line lies on the plane. The segments may be finite or infinite. Infinite segments are either full lines or half-lines. Thus, the segments may have 0, 1 or 2 finite endpoints. A segment may have only 1 finite endpoint for two different reasons: it may be a half-line, or it may be a degenerate segment that is only a single point. However, for the most part, we will describe our algorithms only for finite segments.

Let S be a set of closed segments. In applications, S has some additional structure. For example, S may represent a simple polygon, or more generally, a political map. Think of a political map of a region $R \subseteq \mathbb{R}^2$ to be a subdivision of R into various "countries". A country might not be connected, but have several connected components – these components will be colored with the same color. Since our algorithms are normally oblivious to political realities (*i.e.*, the colors), we think of the map as a collection of pairwise-disjoint connected sets that "subdivides" R. However, to take care of the points on the boundary of each region, we formalize the concept of a map as follows.

For simplicity, let R be the entire Cartesian plane $\mathbb{R}^2$. We define a **cell complex** of $\mathbb{R}^2$ to be a partition of $\mathbb{R}^2$ into a finite collection K of non-empty connected sets (called **cells** subject to the following conditions:

- Each cell is either a singleton set, an open segment (finite or infinite), or an open subset of the plane. These are known as 0-cells, 1-cells or 2-cells, respectively. We also call the **vertices**, **edges** and **faces** of the cell complex K.
- The endpoint of each edge is a vertex.

The **size** of K is just the number $|K$ of cells. We may also partition K into the vertex set V, the edge set E and the face set F. Thus

$$K = V \uplus E \uplus F.$$

First, let us show a partition K that is not a cell complex. Consider the partition in figure 2, comprising three faces $F = \{f_0, f_1, f_2\}$, two edges $E = \{e_1, e_2\}$ and no vertices $V = \emptyset$. $K = E \cup F$ is not a cell complex since the endpoint of e_2 is not a vertex. Let us note a consequence of our definition (its proof is left as an exercise).

Lemma 1 *The boundary of every face in a cell complex is equal to the union of a set of vertices and edges.*

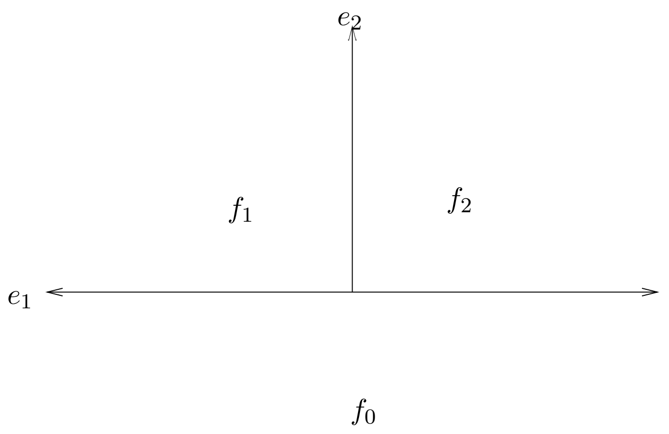

Figure 2: Non-cell complexes

Proof. The boundary $\partial(f)$ of a face f is surely contained in $(\cup V')\cup(\cup E')$ where $V' \subseteq V$ and $E' \subseteq E$. Assume V' and E' are minimum. The lemma can only be violated in the boundary contains a proper subpart of some $e \in E'$. Let $e' = e \setminus \partial(f)$. Since $\partial(f)$ is closed, e' is relatively open. Let v' be a point in $\partial e'$. Take any sufficiently small ball B about v'. Clearly, B intersects some edge $e'' \in E'$, different from e'. If the ball is sufficiently small, it means that v' is an endpoint of e''. But v' is not a vertex since it is a point of e'. Thus K is not a complex. **Q.E.D.**

Our definition is deliberately general, and accepts certain "irregularities": (i) A vertex may be isolated (it is not the endpoint of any edge). (ii) We may have "pinch" vertices. (iii) An edge may have the same face on both sides (such edges may be called "isthmuses". If the isthmus does not connect a hole to an outer boundary of the face, we say the edge is "dangling". (iv) A face may not be simply-connected, *i.e.*, there may be holes in a region.

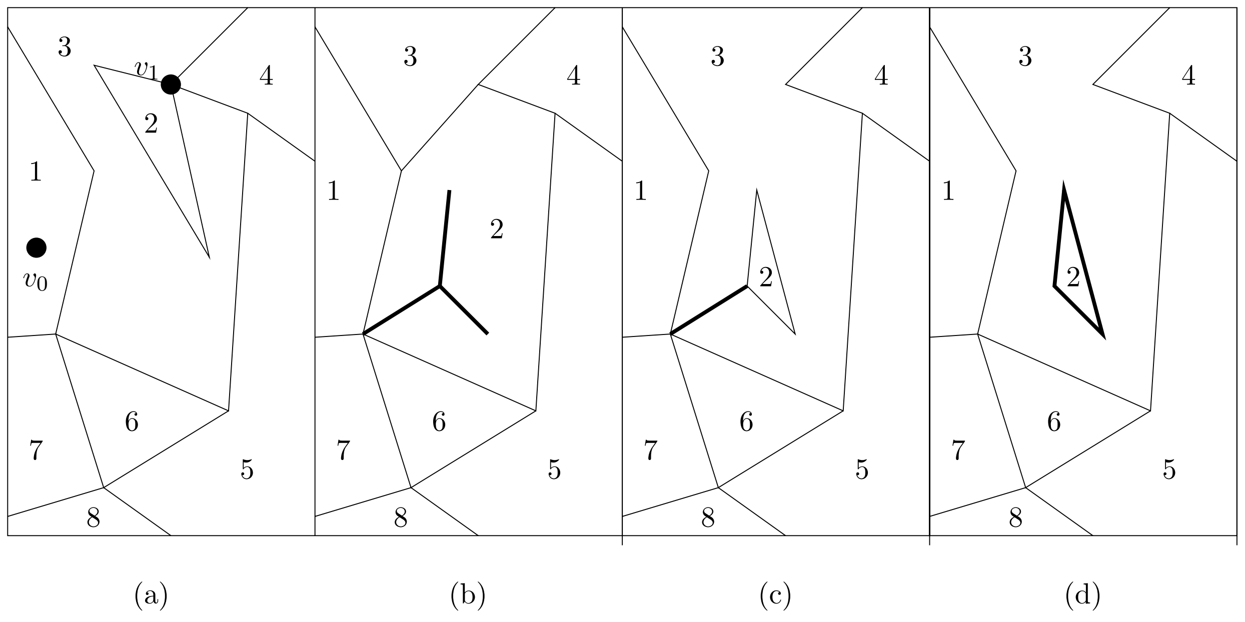

Figure 3: Irregularities: (a) isolated (v_0) and pinched (v_1) vertices (b) isthmus, (c) dangling segment, (d) holes.

These irregularities are illustrated in figure 3. Even if the eventual result of our algorithms do not admit the possibilities, an algorithm may produce them through intermediate construction steps. For instance, if we want to incrementally construct a cell complex, it may be useful to assume that all newly added edges connect two existing vertices. In this case, we would want to allow the introduction of a new isolated vertex, just before we add edges to non-isolate it! Similarly, dangling segments may appear before we finally complete a chain of segments to form a new region. This phenomenon appears in constructive geometry where we admit a very simple set of primitive operations ("Euler operations").

Skeletons and Non-Crossing Segments. A cell complex K is essentially determined by its edges:

Lemma 2 *(a) A cell complex K is determined by its set E of edges and the set of isolated vertices $V' \subseteq V$. (b) Conversely, given any set S of pairwise disjoint point sets, where each set in S is either an open line segment or singletons, we obtain a unique cell complex.*

In general, any set K_1 of pairwise disjoint line segments, where each segment is either an open line segment or a degenerate singleton, is called a **skeleton** (sometimes called 1-skeleton). In particular, restricting a cell complex K to its 0- and 1-cells, we obtain its skeleton K_1. Alternatively, we can think of a skeleton as defined by a set S of closed line segments that is **non-crossing** in the sense that if two segments intersect, then they intersect at a unique point which is a common endpoint of the segments. The endpoints of segments in S are called **vertices** of S.

Our lemma says that K_1 or S defines a cell complex. We take either viewpoint for S, as convenient. However, we will represent them differently. We represent S as some list or set of line segments. We represent K_1 as a list of vertices (with x- and y-coordinates) and list of edges. Each edge is simply a pointer to two vertices. So the main difference is K_1 treat vertices as independent objects, and edges as relations among these vertices, while vertices in S do not have independent representation.

Subdivisions. A cell complex K is called a **(plane) subdivision** if

- Each face f is "regular" in the sense that that f is equal to the interior of its closure $\overline{f}$.
- Each bounded face is simply-connected.

The first property excludes dangling segments or isthmuses. See figure 3. However this does not eliminate "pinch vertices". The second property exclude holes from faces. But note[1] that unbounded faces need not be simply-connected.

Infinite Segments. Most of our algorithms assume the segments are finite. This is not an essential restriction, but simplifies the presentation. Note that every cell complex has at least one infinite face. If all the edges are finite, then there is exactly one infinite face, and this face is not simply-connected.

Let us see how we can bring infinite segments into our framework. An infinite segment has at least one endpoint "at infinity". A **direction** θ is simply an angle in the range $[0, \pi)$. We can identify each direction with a point at infinity.

[1]We could have required unbounded faces to be simply-connected as well. Our choice just simplifies our algorithms.

Degeneracy. Another big practical issue is degenerate inputs. We will try to avoid this issue in this chapter, but it will be treated in depth when we discuss symbolic perturbation methods. Symbolic perturbation can be thought of as infinitesimal perturbations. Many degeneracy problems can be solved by such perturbations. The following is the framework proposed by Yap: assume that an algorithm A can correctly solve all non-degenerate input instances. Such an algorithm is called **generic**. The idea is to construct a "transformation" of A into some A', largely independent of the programming details in A, such that A' will run "correctly" on all inputs. Moreover, A' will be indistinguishable from A on non-degenerate inputs. Suppose for degenerate input I, the output of A' is $A'(I)$. What exactly is $A'(I)$? The answer is *the correct answer for an infinitesimal perturbation of* I. The issue then, is to construct a **postprocessing algorithm** B which takes $A'(I)$ and outputs the correct answer for the actual input I. In specific examples we see that B is quite simple. Moreover, B is "generic" so that it is also independent of the choice of A. This means that once we have constructed B, it will work for any generic A.

There are some problems which seems to be hard to solve under this general framework. In particular, intersection problems. For instance, if s, s' are two segments that "just intersects" (say, an end point of s is in the relative interior of s'). Then a perturbation of s, s' may or may not see this intersection. In some applications, this is immaterial. But suppose we insist on detecting this intersection. One idea we will explore is to perturb s and s' in two opposite directions, ensuring that at least one of them will detect the intersection.

§3. The Trapezoidal Map and its Inverse

The (vertical) **trapezoidal map** for a set S of segments is defined to be two functions, α and β that assign segments (or ∞) $\alpha(p)$ and $\beta(p)$ to each endpoint p of segments in S. See figure 4. The segment $\alpha(p)$ is the first segment that is intersected by a vertical ray that shoots vertically *above* p; $\alpha(p) = \infty$ in case the ray does not intersect any segment of S. The segment $\beta(p)$ is similarly defined for a ray that shoots vertically *below* p. This map can be easily constructed in $O(n \log n)$ time, using a modified Bentley-Ottman algorithm where the sweepline sweeps from left to right.

There is an analogous horizontal trapezoidal map, and the sweepline algorithm for a vertical trapezoidal map could *simultaneously* construct this map (see Exercise). This also shows that, in constructing the vertical trapezoidal map, we could have the sweepline move from top to bottom if we like.

We may define the **inverse trapezoidal map** for S to mean the functions A, B that assigns to each segment $s \in S$ two lists $A(s)$ and $B(s)$. List $A(s)$ (resp., $B(s)$) stores the endpoints of S that are vertically visible (respectively) *above* and *below* s. The points in these lists are sorted by their x-coordinates. Note that p is in $A(s)$ iff $\beta(p) = s$, and p is in $B(s)$ iff $\alpha(p) = s$. These lists can constructed at the same time that we construct the trapezoidal map.

Note from the inverse trapezoidal map, we can construct the trapezoidal map in linear time. Conversely, we can construct the inverse trapezoidal map from the trapezoidal map in linear time, *provided* we also have the endpoints sorted by y-coordinates. See Exercise. If only one of the two maps α and β is needed, we can simplify our procedures accordingly.

Trapezoidal Subdivision. Let S be a set of non-crossing segments. A key advantage of the trapezoidal map is that it leads to a plane subdivision for any such set S. Let $K(S)$ denote this subdivision. The faces of $K(S)$ are "trapezoids" (a convex face with 4 sides, two of which are parallel), hence the name for the corresponding maps. The trapezoids may be unbounded (with one, two or three sides) and the bounded trapezoids degenerate into a triangle. See figure 5.

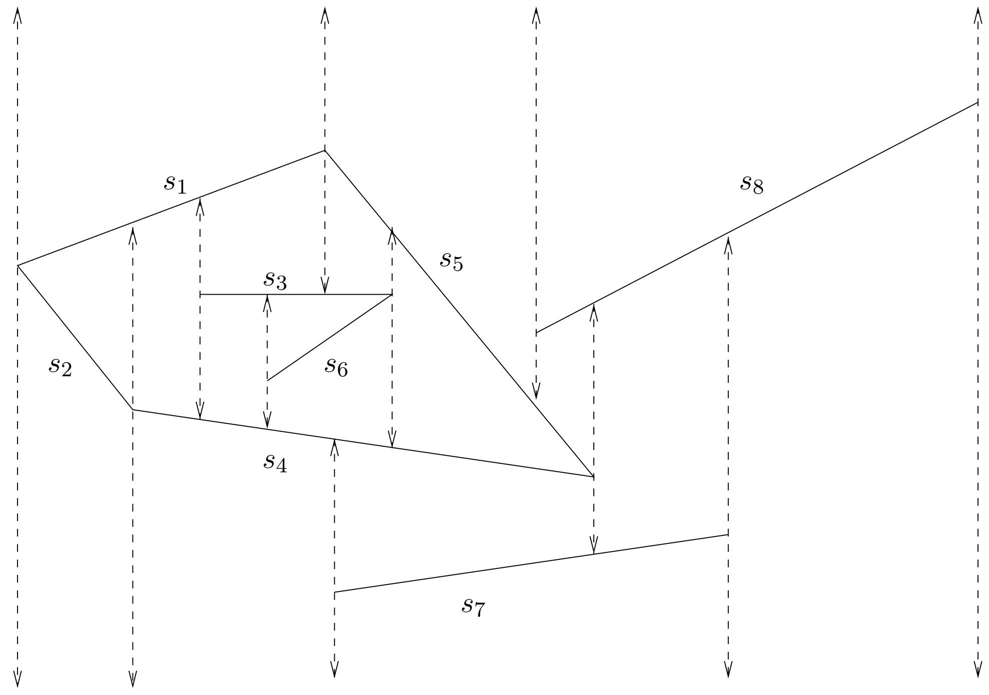

Figure 4: Vertical trapezoidal map

With each bounded trapezoid $\Delta \in K(S)$, we define four bounding objects: two segments of S, $top(\Delta)$ and $bot(\Delta)$, and two endpoints of S, $leftp(\Delta), rightp(\Delta)$. These four objects uniquely determine Δ. If Δ is unbounded, some of these defining objects may be undefined (or taken to be ∞).

Two trapezoids Δ, Δ' are **adjacent** if they share a non-zero length of vertical boundary. Assuming non-degeneracy conditions, each trapezoid is adjacent to at most 4 other trapezoids. Moreover, the four bounding objects are also uniquely defined in this case.

We represent $K(S)$ as a set of trapezoids, each associated with 4 bounding objects, and pointing to its at most 4 adjacent trapezoids.

§4. Regularizations of Non-Crossing Segments

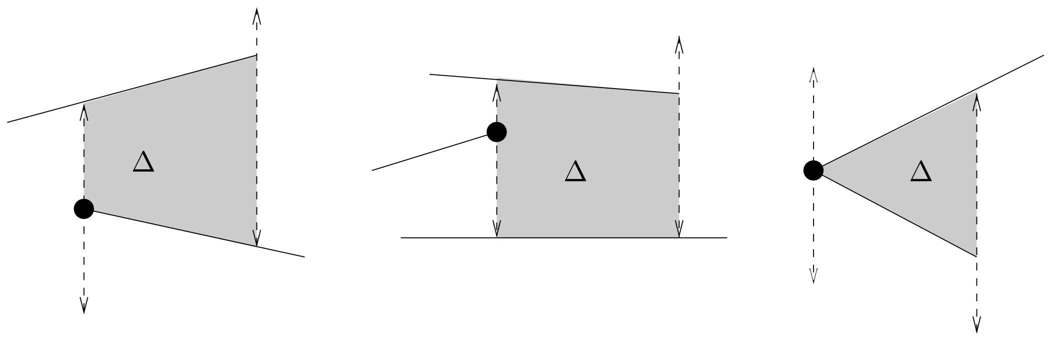

Figure 5: Various $leftp(\Delta)$ configurations

Let S be a finite set of non-crossing segments. The endpoints of segments in S are called its vertices. In this section, we consider some "regularity" properties that S may have.

A set X is θ-monotone in the direction θ if every line perpendicular to the direction θ intersects X in a connected set (possibly empty). If $\theta = 0$ (resp., $\theta = \pi/2$), we say the set is x-monotone (resp., y-monotone), a terminology derived from the directions of the x- and y-axes. A simple polygon is θ-monotone if its interior is θ-monotone. See figure 6 for examples of monotone and non-monotone polygons.

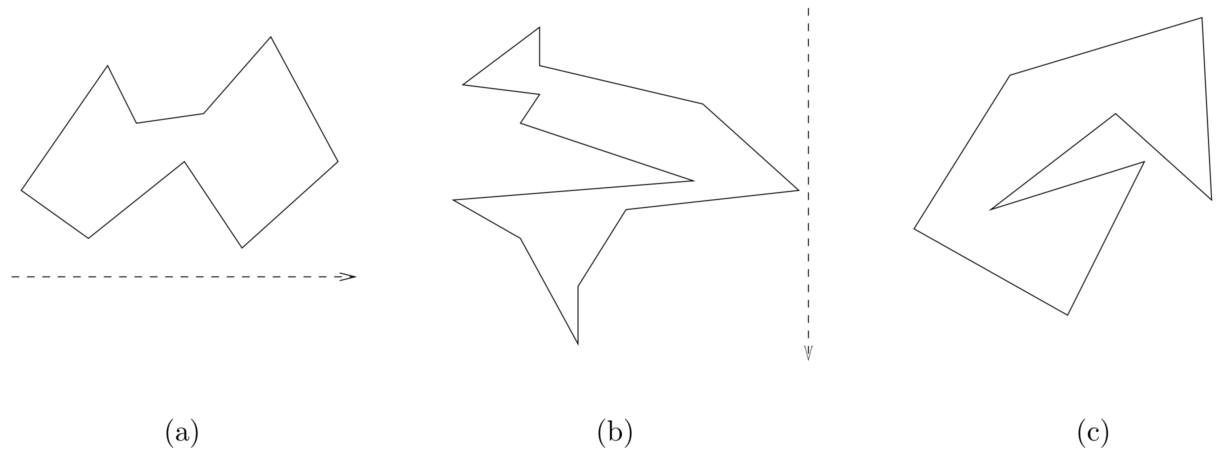

Figure 6: Polygons: (a) x-monotone, (b) y-monotone, (c) non-monotone in any direction.

A **monotone subdivision** is a subdivision in which all bounded faces are θ-monotone for some fixed θ. A **convex subdivision** is a subdivision in which each bounded face is convex. A **(convex) quadrilateral subdivision** is a subdivision in which each bounded face is a convex quadrilateral. A **triangulation** is a subdivision in which each bounded face is triangular. We consider the following hierarchy of "regularity" conditions on S:

- S is non-crossing.
- S defines a subdivision.
- S defines a monotone subdivision.
- S defines a convex subdivision.
- S defines a quadrilateral subdivision.
- S defines a triangulation.

Clearly, each regularity condition in this list implies all the conditions preceding it. By adding sufficiently many new segments that connect vertices of S, we can achieve any of the above regularity conditions. The problem of "subdivisioning", "monotonizing", "convexifying" or "triangulating" a set of non-crossing segments is that of adding such segments to attain the desired regularity condition. We shall also be interested in minimizing the number of added segments. We will see that the trapezoidal map and its inverse will allow us to achieve these regularity objectives in linear time.

In the following, we assume for simplicity that all segments are finite. Hence there is a unique unbounded face. In regularization, we never need to add infinite segments.

Classifying Vertices. We want to classify the vertices of a cell complex K. The most intuitive way to do this is to proceed from the special case to the most general.

First, we classify the vertices of a simple polygon into the following **types**: let v be a vertex and $H(v)$ be the horizontal line through v.

- **start**: the two segments incident to v lies below $H(v)$, and the interior angle at v lies below $H(v)$.
- **stop**: the two segments incident to v lies above $H(v)$. and the interior angle at v lies above $H(v)$.
- **split**: the two segments incident to v lies below $H(v)$, and the exterior angle at v lies below $H(v)$.
- **merge**: the two segments incident to v lies above $H(v)$. and the exterior angle at v lies above $H(v)$.
- **regular**: This has one segment above $H(v)$ and one below $H(v)$.

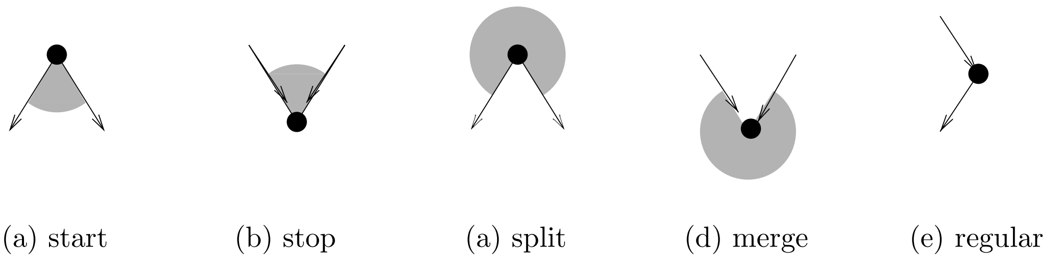

Figure 7: Types of vertices of a simple polygon

For simplicity, we assume that all vertices of our polygons falls under exactly one of the 5 types. This assumption will be well justified because it is one of those situations which can easily be handled by symbolic perturbation techniques. This will be described in a subsequent chapter.

Next consider the vertices of a subdivision K. we classify its vertices as follows: we say a vertex is a **split** or **merge** vertex if it is split or merge vertex for any of the polygonal faces of the subdivision, Otherwise it is classified as **regular**. It should be noted that this classification is unambiguos: a vertex could not be classified as both split and merge.

Finally, consider a cell complex K. First, we find it convenient to split each vertex into two "half-vertices": a **upper half-vertex** is one can only link to other vertices (or half-vertices) with larger y-coordinate. A **lower half-vertex** is similarly defined. Each vertex can be regarded as a pair of these vertices. The **upper degree** of a vertex is the degree of its upper half-vertex, and **lower degree** is similarly defined. We can easily incorporate degeneracies for this discussion, but leave this for an exercise. In a horizontal sweepline sweeping from top to bottom, we assume the upper half-vertex is swept before the lower half-vertex.

Not all the faces of a cell complex K are simple polygons, so the previous scheme does not quite work. Actually, there are only two new kinds of vertices: a vertex is a **slit start** vertex if its upper degree is 0 and its lower degree is 1. It is a **slit stop** vertex if these degrees are 1 and 0 respectively. Together, these are called **slit vertices**.

Lemma 3 *(a) A simple polygon is y-monotone iff it has no split or merge vertices.*
(b) A subdivision is monotone iff it has no split or merge vertices.
(c) A cell complex is a subdivision iff it has no slit vertices.

Proof. (a) See text books.
(b)
(c) **Q.E.D.**

Subdivisioning. We now address the issue of subdivisioning of a non-crossing set S. According to the previous lemma, it amounts to removing all slit vertices. We are further interested in introducing the minimum number of edges to achieve this (otherwise, we could use a triangulation algorithm, say).

We will use a horizontal sweepline H, but with a twist: we first do a downward sweep then an upward sweep. As usual let $H(t)$ denotes its position at $y = t$.

The 2 bounding vertices for each segment are denoted $top(s)$ and $bot(s)$ where we assume that the sweepline reaches $top(s)$ before $bot(s)$. In general, let $H(t)$ intersect the segments $s_1, \ldots, s_k$ from S in this order. This determines a sequence of faces $f_0, f_1, \ldots, f_k$ where $f_0 = f_k$ is the unique unbounded face. Note that the f_i's need not be distinct. In fact, two consecutive regions f_i, f_{i+1} may be identical. The events we consider at $top(s_1, \ldots, s_k)$ where the s_i's are newly encountered segments. We also have $bot(s_1, s_2, \ldots, s_k)$ where $s_1, \ldots, s_k$ are current segments that simultaneously terminate at a common vertex.

Our goal is to introduce regularizing edges. To indicate some of the issues, suppose that at the beginning we have $k = 1$. We have to add at least one edge to connect to $top(s_1)$. But we have no idea whether this new edge is to the left or the right side of s_1.

§5. Map Overlay

Two basic problems in plane subdivisions are: (1) Subdivision Overlay Problem – given two subdivisions S_1, S_2 of the plane, compute their "overlay" subdivision $O(S_1, S_2)$. (2) Point Location Problem – given a subdivision S, preprocess it into some data structure which efficiently supports point location queries. A point location query is specified by a point p, and we have to determine the facet $f \in S$ that contains p.

A map can be viewed as an integration of a variety of map themes: population (towns, urban areas, etc), communications (roads, rail, etc), utility (gas, electricity, etc), vegetation, land use, political boundaries, land marks, etc.

§6. Notes

Lee and Preparata (1977) were the first to use mototone polygon in planar point location. The first algorithm for triangulating a simple polygon is by Gary, Johnson, Preparata and Tarjan (1978), who first monotonizes the polygon and then triangulate each monotone pieces. Hertel and Mehlhorn (1983) gave an algorithm for convexifying a simple polygon that has at most 4 times the minimal number r of convex pieces. Greene (1983) gave an $O(r^2 n^2)$ algorithm to find a minimum convexification; Keil (1985) improved the time to $O(r^2 n \log n)$. The trapezoidal decomposition was introduced by Fournier (1984) and by Chazelle and Incerpi (1984) in the context of triangulating polygons. The major result of this area is Chazelle's proof that triangulation of a simple polygon is linear time. Two results preceding this are Yap (198?) shows that triangulation of a monotone polygon can be achieved quickly in parallel, by two calls to the trapezoidal map. The books of O'Rourke and de Berg et al give account of the problem of triangulating a simple polygon via monotone subdivisions.

Our approach above provides a new framework for looking at several related processes, which we term "regularization". Moreover, the trapezoidal map is shown to be the key for this unified view. We can formulate all the regularization problems in the setting where new vertices "Steiner points" may be introduced. This is especially relevant for the problem of minimum convexification. For instance, Chazelle (1980) shows that minumum Steiner point convexification of a simple polygon can be done in $O(n + r^3)$, where the r now refers to the mimumum number of Steiner point convex pieces.

Exercises

Exercise 6.1: Extend the sweepline segment intersection algorithm by allowing the segments to be doubly-infinite or semi-infinite. ♢

Exercise 6.2: Consider the following variation of segment intersection. We are given two sets R, B of segments. Think of them as the "red" and "blue" segments. The problem is to compute all bichromatic intersections. Of course, we can reduce this to the monochromatic case, and pick out the subset of intersections that is bichromatic, but this would not be output-sensitive. Give an algorithm that is output sensitive. ♢

Exercise 6.3: We can reduce the case of non-simply-connected faces to simply-connected faces by introducing "isthmuses" joining each lake to the outermost boundary (perhaps via other lakes). If there are k lakes, it is sufficient to introduce k edges. In terms of half-edges, both half-edges bound the same region. This can be considered a definition of an isthmus edge. Discuss this extension to our algorithms on plane subdivisions. ♢

Exercise 6.4: If we have the trapezoidal map and the list of endpoints sorted by y-coordinates, we can construct the inverse trapezoidal map in linear time. ♢

Exercise 6.5: Modify the horizontal plane sweep so that we compute both the horizontal and the vertical trapezoidal map. Give an application of having these two maps simultaneously. ♢

Exercise 6.6: Restrict simple subdivisions to have no pinched vertices. Modify the above algorithms accordingly. ♢

Exercise 6.7: If S is a non-crossing set of n finite segments, what is the minimum number of vertices in S? ♢

Exercise 6.8: Give a direct treatment for the degenerate cases in the regularization problems. ♢

End Exercises

END OF LECTURE