CMSC 754: Lecture 4
Line Segment Intersection

Reading: Chapter 2 in the 4M's.

Geometric intersections: One of the most basic problems in computational geometry is that of computing intersections. It has numerous applications.

- In solid modeling complex shapes are constructed by applying various boolean operations (intersection, union, and difference) to simple primitive shapes. The process is called *constructive solid geometry* (CSG). Computing intersections of model surfaces is an essential part of the process.
- In robotics and motion planning it is important to know when two objects intersect for *collision detection* and *collision avoidance*.
- In geographic information systems it is often useful to *overlay* two subdivisions (e.g. a road network and county boundaries to determine where road maintenance responsibilities lie). Since these networks are formed from collections of line segments, this generates a problem of determining intersections of line segments.
- In computer graphics, *ray shooting* is a classical method for rendering scenes. The computationally most intensive part of ray shooting is determining the intersection of the ray with other objects.

In this lecture, we will focus the basic primitive of computing line segment intersections in the plane.

Line-Segment intersection: We are given a set $S = \{s_1, \ldots, s_n\}$ of n line segments in the plane. Let's assume that each is represented by its two endpoints $s_i = \overline{p_i q_i}$. We will consider the problem of reporting pairs of line segments that intersect (see Fig. 1(a)).

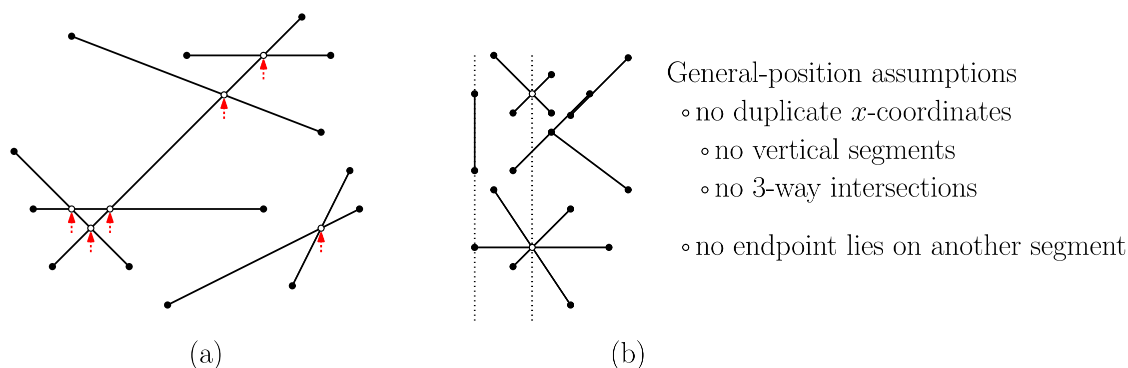

Fig. 1: Line segment intersection.

To simplify the presentation, we will make the usual *general-position assumptions*. In particular, we will rule out the following (see Fig. 1(b)):

- No duplicate x-coordinates among endpoints or intersection points (which implies no vertical line segments or multiple intersections)
- No segment endpoint lies on another segment

All these special cases are relatively easy to cope with, but the algorithms are more complex because they need to deal with various special cases. One common exception to the above is to allow multiple segments to share a common endpoint, since this occurs when dealing with polygons and spatial subdivision.

Output Sensitivity: Observe that n line segments can intersect in as few as zero and as many as $\binom{n}{2} = O(n^2)$ different intersection points. We could settle for an $O(n^2)$ time algorithm, claiming that it is worst-case asymptotically optimal, but it would not be very useful in practice, since in many instances of intersection problems intersections may be rare. Therefore, we will focus on *output sensitive algorithms*, meaning that the running time depends not only on the number of segments, but also the number of intersecting pairs.

Given a set S of n line segments, let $m = m(S)$ denote the number of intersections. We will express the running time of our algorithm in terms of both n and m. As usual, we will assume that the line segments are in general position.

Utility Data Structures: Our algorithm will make use of two standard dynamic data structures (see CLRS for details).

Ordered Dictionary: Stores a set of objects each associated with a key from an totally ordered domain. Supports the operations *insert*, *delete*, *find*, *predecessor*, *successor*, *get-kth* (which returns the item at some given rank) and *swap* (which swaps two given adjacent entries) all in $O(\log n)$ time and $O(n)$ space, where n is the current number of objects. (This is typically implemented using balanced binary trees, such as red-black trees, or skip lists. Note that because of the predecessor, successor, and swap operations, we cannot used unordered dictionaries like hashing.)

Priority Queue: Stores a set of objects, where each object is associated with a priority value. Supports the operations *insert*, *delete*, and *extract-min* (which accesses and removes the object with the smallest priority value). We also assume that the insert operation returns a special *reference* to the inserted object, which allows us to efficiently delete this item. These operations are supported in $O(\log n)$ time and $O(n)$ space, where n is the current number of objects. (This is typically implemented as a binary heap.)

Plane Sweep: Let us now consider a natural approach for reporting the segment intersections. The method, called *plane sweep*, is a fundamental technique in planar computational geometry. We solve a 2-dimensional problem by simulating the process of sweeping a 1-dimensional line across the plane. The intersections of the sweep line with the segments defines a collection of points along the sweep line.

Although we might visualize the sweeping process as a continuous one, there is a discrete set of *event points* where important things happen. As the line sweeps from left to right, points are inserted, deleted, and may swap order along the sweep line. Thus, we reduce a static 2-dimensional problem to a dynamic 1-dimensional problem.

In any algorithm based on plane sweep there are three basic elements that need to be maintained (see Fig. 2). Let ℓ denote the current sweep line.

(1) the *partial solution* that has already been constructed to the left of ℓ (in our case, the discovered intersection points lying to the left of the sweep line),

(2) the *sweep-line status*, that is, the set of objects intersecting ℓ (in our case, the sorted segments intersecting the sweep line), and

(3) a subset of the *future events* lying to right of ℓ that are scheduled to be processed (in our case, a subset of the intersection points to the right of the sweep line).

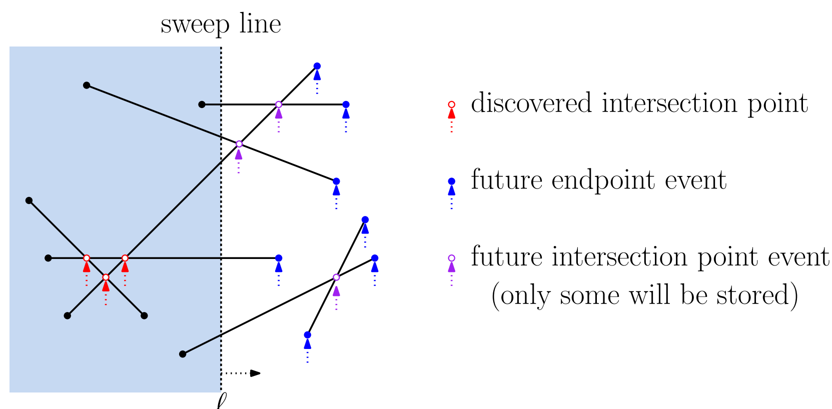

Fig. 2: Plane sweep.

The key to designing an efficient plane-sweep algorithm is determining how to efficiently store and update these three elements as each new event is process. Let's consider each of these elements in greater detail in the context of line-segment intersection.

Sweep line status and above-below comparisons: We will simulate the sweeping of a vertical line ℓ from left to right. The sweep-line status consists of the line segments that intersect the sweep line sorted, say, from top to bottom. In order to maintain this set dynamically, we will store them in an *ordered dictionary*.

But hey! How can we possibly do this efficiently? Every time we move the sweep line even a tiny distance, all the y-coordinates of the intersection points change as well! Clearly, we cannot store the y-coordinates explicitly, for otherwise we would be doomed to $O(n)$ time per event, which would lead to an overall running time that is at least quadratic.

The key is that we do not need store the actual y-coordinates in the dictionary. Instead, we implement a function that compares two line segments at a given the x-coordinate. This function determines which segment intersects the sweep line above the other. Let's call this a *dynamic comparator*. (A more detailed description is provided at the end of the lecture notes.)

Observe that between consecutive event points (intersection points or segment endpoints) the relative vertical order of segments is constant (see Fig. 3(a)). For each segment, we can compute the associated line equation, and evaluate this function at x_0 to determine which

segment lies on top. The ordered dictionary does not need actual numbers. It just needs a way of comparing objects (see Fig. 3(b)).

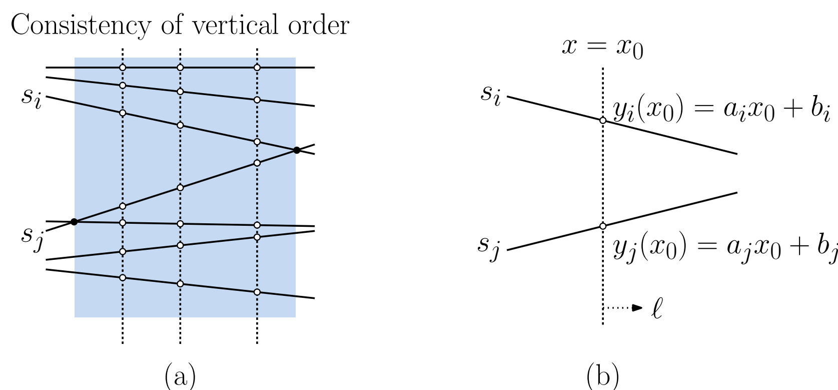

Fig. 3: The dictionary does not need to store absolute y-coordinates, just the ability to make above-below comparisons for any location of the sweep line.

Events and Detecting Intersections: It suffices to process events only when there is a change in the combinatorial structure of the sweep-line status. As mentioned above, these x-coordinates are called *event points*. For our application, we have three types of event points, corresponding to when the sweep line encounters: (1) the left endpoint of a segment, (2) the right endpoint of a segment, and (3) an intersection point between two segments.

Note that endpoint events ((1) and (2)) can be *presorted* before the sweep runs. In contrast, intersection events (3) will be discovered *dynamically* as the sweep executes. It is important that each event be detected before the actual event occurs. Since each pair of segments along the sweep line might intersect, there are $O(n^2)$ potential intersection events to consider, which again would doom us to at least quadratic running time. How can we limit the number of potential intersection points to a manageable number?

Our strategy will be as follows. Whenever two line segments become *adjacent* along the sweep line (one immediately above the other), we will check whether they have an intersection occurring to the right of the sweep line. If so, we will add this new event to a priority queue of future events. This priority queue will be sorted in left-to-right order by x-coordinates. We call this the *adjacent-segment rule.*

A natural question is whether this strategy of scheduling intersections between adjacent pairs is correct. In particular, might it be that two line segments intersect, but just prior to this intersection, they were not adjacent in the sweep-line status? If so, we would miss this event. Happily, this is not the case, but it requires a proof. (If you think it is trivial, note that it would fail to hold if the objects being intersected were general algebraic curves, not line segments.)

Lemma: Consider a set S of line segments in general position, and consider two segments $s_i, s_j \in S$ that intersect in some point p. Then s_i and s_j are adjacent along the sweep line just after the event that immediately precedes p in the sweep.

Proof: By our general position assumptions, no three lines intersect in a common point. Therefore if we consider a placement of the sweep line that is infinitesimally to the left of the intersection point, the line segments s_i and s_j will be adjacent along this sweep line. Consider the event point q with the largest x-coordinate that is strictly less than p_x. Since there are no events between q_x and p_x, there can be no segment intersections within the vertical slab bounded by q on the left and p on the right (the shaded region of Fig. 3), and therefore the order of lines along the sweep line after processing q will be identical the order of the lines along the sweep line just prior p. Therefore, s_i and s_j are adjacent immediately after processing event q and remain so just prior to processing p.

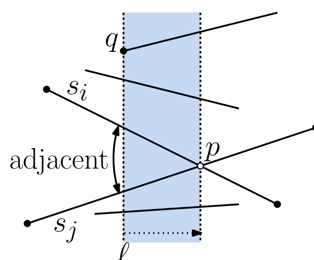

Fig. 4: Correctness of the adjacent-segment rule.

When two formerly adjacent segments cease to be adjacent (e.g., because a new segment is discovered between them), we will delete the event from the queue. While this is not formally necessary, it keeps us from inserting the same event point repeatedly and it guarantees that the total number of events can never exceed $O(n)$.

Data Structures: As mentioned above the segments that intersect the sweep line will be maintained in an ordered dictionary, sorted vertically from top to bottom. The future event points (segment endpoints and impending intersection points) will be stored in a *priority queue*, which will be ordered from left to right by x-coordinates.

Here are the operations assumed to be supported by the *ordered dictionary*, which stores the sweep-line status:

- $r \leftarrow$ insert(s): Insert s (represented symbolically) and return a reference r to its location in the data structure.
- delete(r): Delete the entry associated with reference r.
- $r' \leftarrow$ predecessor(r): Return a reference r' to the segment lying immediately above r (or null if r is the topmost segment).
- $r' \leftarrow$ successor(r): Return a reference r' to the segment lying immediately below r (or null if r is the bottommost segment).
- $r' \leftarrow$ swap(r): Swap r and its immediate successor, returning a reference to r's new location.

Next, here are the operations assumed to be supported by the *priority queue*, which stores the future events sorted by the x-coordinates:

- $r \leftarrow \text{insert}(e, x)$: Insert event e with "priority" x and return a reference r to its location in the data structure.
- $\text{delete}(r)$: Delete the entry associated with reference r.
- $(e, x) \leftarrow \text{extract-min}()$: Extract and return the event from the queue with the smallest priority x.

As mentioned earlier, all of these operations can be performed in $O(\log n')$ and $O(n')$ space, where n' is the current number of entries in the data structure.

The Final Algorithm: All that remains is explaining how to process the events. This is presented in the code block below, and the various cases are illustrated in Fig. 4. (Further details can be found in the 4M's.)

Line Segment Intersection Reporting

(1) Insert all of the endpoints of the line segments of S into the event queue. The initial sweep-line status is empty.

(2) While the event queue is nonempty, extract the next event in the queue. There are three cases, depending on the type of event:

Left endpoint: (see Fig. 5(a))

(a) Insert this line segment s into the sweep-line status, based on the y-coordinate of its left endpoint.

(b) Let s^+ and s^- be the segments immediately above and below s on the sweep line. If there is an event associated with this pair, remove it from the event queue.

(c) Test for an intersection between s and s^+, and if so, add it to the event queue. Do the same for s and s^-.

Right endpoint: (see Fig. 5(b))

(a) Let s^+ and s^- be the segments immediately above and below s on the sweep line.

(b) Delete segment s from the sweep-line status.

(c) Test for an intersection between s^+ and s^- to the right of the sweep line, and if so, add the corresponding event to the event queue.

Intersection: (see Fig. 5(c))

(a) Let s^+ and s^- be the two segments involved (with s^+ above just prior to the intersection). Report this intersection.

(b) Let s^{++} and s^{--} be the segments immediately above and below the intersection. Remove any event involving the pair (s^+, s^{++}) and the pair (s^-, s^{--}).

(c) Swap s^+ and s^- in the sweep-line status (they must be adjacent to each other).

(d) Test for an intersection between s^- and s^{++} to the right of the sweep line, and if so, add it to the event queue. Do the same for s^+ and s^{--}.

Correctness: The correctness of the algorithm essentially follows from our extensive derivation to the algorithm itself. Formally, the correctness proof is based on an induction proof showing that immediately after processing each event: (a) the sweep-line status contains the line segments intersecting the sweep line in sorted order and (b) the event queue contains exactly all the events demanded by the adjacent-segment rule.

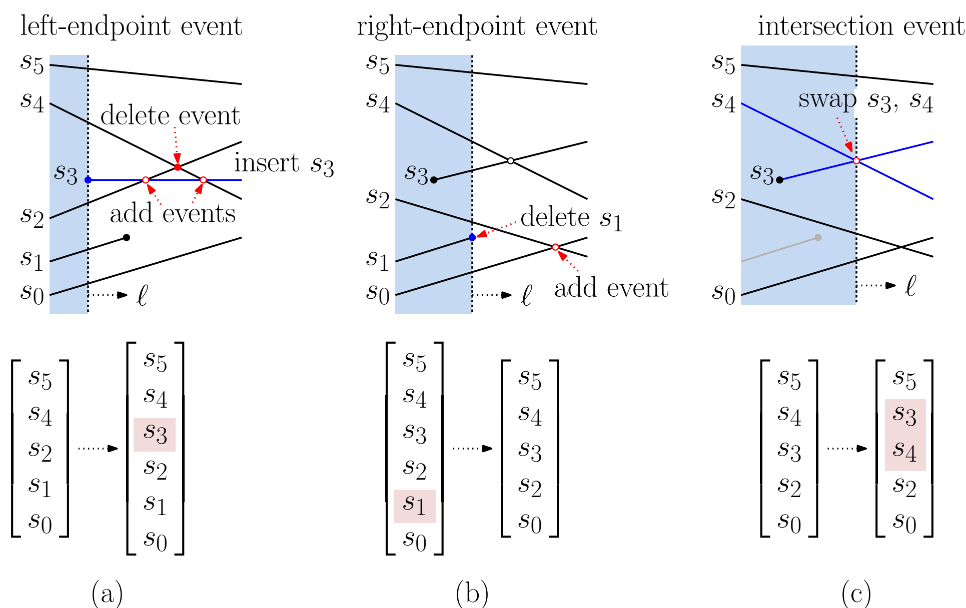

Fig. 5: Plane-sweep algorithm event processing.

Analysis: Altogether, there are $2n+m$ events processed (two endpoints per segment and m intersections). Each event involves a constant amount of work and a constant number of accesses to our data structures. As mentioned above, each access to either of the data structures takes $O(\log n)$ time. Therefore, the total running time is $O((2n+m)\log n) = O(n\log n + m\log n)$. Note that if we output each intersection point without storing it, the total storage requirements never exceed $O(n)$. In summary, we have:

Theorem: Given a set of n line segments S in the plane (subject to our general-position assumptions), the above algorithm correctly reports all the m intersections between these segments in time $O((n+m)\log n)$ time and $O(n)$ space.

Lower Bound: Is this the best possible? No. There is a faster algorithm (which we may discuss later in the semester) that runs in time $O(n\log n + m)$. This latter algorithm is actually optimal. Clearly $\Omega(m)$ time is needed to output the intersections. The lower bound of $\Omega(n\log n)$ results from a reduction from a problem called *element uniqueness*. In this problem, we are give a list of n numbers $X = \langle x_1, \ldots, x_n\rangle$ and we are asked whether there are any duplicates (or all are distinct). Element uniqueness is known to have a lower bound of $\Omega(n\log n)$ in the algebraic decision-tree model of computation. (It can be solved in $O(n)$ time using hashing, but the algebraic decision-tree model does not allow integer division, which is needed by hashing.)

The reduction involves converting each x_i into a line segment s_i that passes through the point $(x_i, 0)$, but otherwise there are no other intersections. (A little cleverness is needed to guarantee that the general-position assumptions are satisfied.) Clearly, two segments s_i and s_j intersect if and only if two elements x_i and x_j of the list are identical. So, determining whether there is even a single intersection requires at least $\Omega(n\log n)$ time.

Dynamic Comparator: (Optional) Let's consider how to design a function that compares two line segments relative to the current sweep line. We are given the sweep line $x = x_0$ and two segments s_i and s_j. Assuming each segment is nonvertical and has endpoints $p_i = (p_{i,x}, p_{i,y})$ and $q_i = (q_{i,x}, q_{i,y})$, we can compute the associated line equations $\ell_i : y = a_i x + b_i$ and $\ell_j : y = a_j x + b_j$, by solving the simultaneous equations

$$p_{i,y} = a_i p_{i,x} + b_i \qquad q_{i,y} = a_i q_{i,x} + b_i,$$

which yields

$$a_i = \frac{p_{i,y} - q_{i,y}}{p_{i,x} - q_{i,x}} \qquad b_i = \frac{p_{i,x} q_{i,y} - p_{i,y} q_{i,x}}{p_{i,x} - q_{i,x}}.$$

By our general-position assumption, the segment is nonvertical, and the denominator is nonzero.

Given that the sweep line is at $x = x_0$, we can define our dynamic comparator to be:

$$\texttt{compare}(s_i, s_j; x_0) = \text{sign}((a_j x_0 + b_j) - (a_i x_0 + b_i)),$$

which returns $+1$ if s_j is above s_i, 0 if they coincide, and -1 if s_j is below s_i.

This is the sign of a rationally-valued function, but we can multiply out the denominator to obtain an algebraic function of degree-3 in the segment coordinates. Thus, if the coordinates are expressed as integers, we can determine the sign using at most triple-precision arithmetic.

Computing Intersection Points: (Optional) We have assumed that the primitive of computing the intersection point of two line segments can be performed exactly in $O(1)$ time. Let us see how we might do this. Let $\overline{ab}$ and $\overline{cd}$ be two line segments in the plane, given by their endpoints, for example $a = (a_x, a_y)$. First observe that it is possible to determine *whether* these line segments intersect, simply by applying an appropriate combination of orientation tests. (We will leave this as an exercise.) However, this alone is not sufficient for the plane-sweep algorithm.

One way to determine the point at which the segments intersect is to use a *parametric representation* of the segments. Any point on the line segment $\overline{ab}$ can be written as a convex combination involving a real parameter s:

$$p(s) = (1 - s)a + sb, \qquad \text{for } 0 \le s \le 1,$$

and similarly for $\overline{cd}$ we may introduce a parameter t:

$$q(t) = (1 - t)c + td, \qquad \text{for } 0 \le t \le 1$$

(see Fig. 6).

An intersection occurs if and only if we can find s and t in the desired ranges such that $p(s) = q(t)$. Thus we obtain the two equations:

$$(1 - s)a_x + sb_x = (1 - t)c_x + td_x \qquad \text{and} \qquad (1 - s)a_y + sb_y = (1 - t)c_y + td_y.$$

The coordinates of the points are all known, so it is just a simple exercise in linear algebra to solve for s and t as functions of the coordinates of a, b, c, and d. (A solution may generally

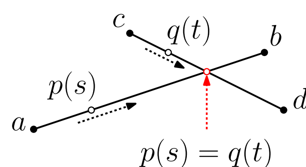

Fig. 6: Plane-sweep algorithm event processing.

not exist, but this means that the segments are parallel. By our assumption that no segments are collinear, this implies that the segments do not intersect.) Once s and t are known, it remains to just check with $0 \leq s, t \leq 1$, to confirm that the intersection point occurs within the line segment (and not in the extended infinite line).

As in our earlier example of determining the order of segments along the sweep line, if all the coordinates are integers, this yields formulas for s and t as rational numbers, and hence the coordinates of the intersection point are themselves rational numbers. If it is needed to perform exact computations on these coordinates, rather than converting them to floating point, it is possible to save the numerator and denominator of each coordinate as a pair of (multiple precision) integers.