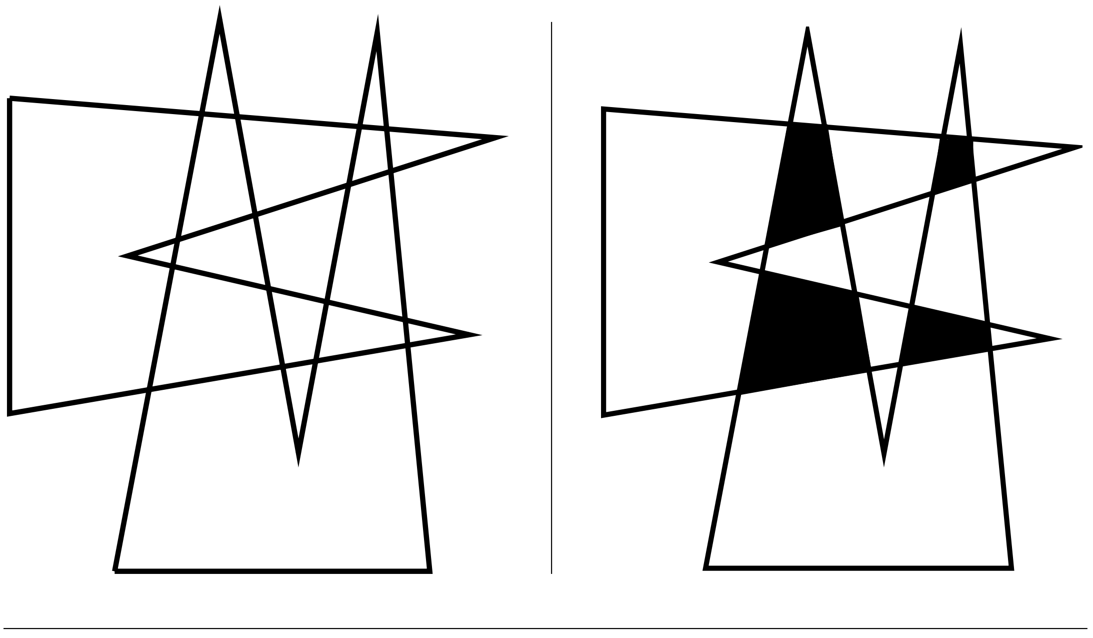

17.8 Intersection Detection

Input description: A set S of lines and line segments $l_1, \ldots, l_n$ or a pair of polygons or polyhedra P_1 and P_2.

Problem description: Which pairs of line segments intersect each other? What is the intersection of P_1 and P_2?

Discussion: Intersection detection is a fundamental geometric primitive with many applications. Picture a virtual-reality simulation of an architectural model for a building. The illusion of reality vanishes the instant the virtual person walks through a virtual wall. To enforce such physical constraints, any such intersection between polyhedral models must be immediately detected and the operator notified or constrained.

Another application of intersection detection is design rule checking for VLSI layouts. A minor design defect resulting in two crossing metal strips could short out the chip, but such errors can be detected before fabrication, using programs to find all intersections between line segments.

Issues arising in intersection detection include:

- *Do you want to compute the intersection or just report it?* – We distinguish between intersection detection and computing the actual intersection.

Detecting that an intersection exists can be substantially easier, and often suffices. For the virtual-reality application, it might not be important to know exactly where we hit the wall—just that we hit it.

- *Are you intersecting lines or line segments?* – The big difference here is that any two lines with different slopes intersect in at exactly one point. All the points of intersections can be found in $O(n^2)$ time by comparing each pair of lines. Constructing the arrangement of the lines provides more information than just the intersection points, and is discussed in Section 17.15 (page 614).

  Finding all the intersections between n line segments is considerably more challenging. Even the basic primitive of testing whether two line segments intersect is not as trivial, as discussed in Section 17.1 (page 564). Of course, we could explicitly test each line segment pair and thus find all intersections in $O(n^2)$ time, but faster algorithms exist when there are few intersection points.

- *How many intersection points do you expect?* – In VLSI design-rule checking, we expect the set of line segments to have few if any intersections. What we seek is an algorithm whose time is *output sensitive*, taking time proportional to the number of intersection points.

  Such output-sensitive algorithms exist for line-segment intersection. The fastest algorithm takes $O(n \lg n + k)$ time, where k is the number of intersections. These algorithms are based on the planar sweepline approach.

- *Can you see point x from point y?* – Visibility queries ask whether vertex x can see vertex y unobstructed in a room full of obstacles. This can be phrased as the following line-segment intersection problem: does the line segment from x to y intersect any obstacle? Such visibility problems arise in robot motion planning (see Section 17.14) and in hidden-surface elimination for computer graphics.

- *Are the intersecting objects convex?* – Better intersection algorithms exist when the line segments form the boundaries of polygons. The critical issue here becomes whether the polygons are convex. Intersecting a convex n-gon with a convex m-gon can be done in $O(n + m)$ time, using the sweepline algorithm discussed next. This is possible because the intersection of two convex polygons must form another convex polygon with at most $n + m$ vertices.

  However, the intersection of two nonconvex polygons is not so well behaved. Consider the intersection of two "combs" generalizing the Picasso-like frontspiece to this section. As illustrated, the intersection of nonconvex polygons may be disconnected and have quadratic size in the worst case.

Intersecting polyhedra is more complicated than polygons, because two polyhedra can intersect even when no edges do. Consider the example of a needle piercing the interior of a face. In general, however, the same issues arise for both polygons and polyhedra.

- *Are you searching for intersections repeatedly with the same basic objects?* – In the walk-through application just described, the room and the objects in it don't change between one scene and the next. Only the person moves, and intersections are rare.

  One common technique is to approximate the objects in the scene by simpler objects that enclose them, such as boxes. Whenever two enclosing boxes intersect, then the underlying objects *might* intersect, and so further work is necessary to decide the issue. However, it is much more efficient to test whether simple boxes intersect than more complicated objects, so we win if collisions are rare. Many variations on this theme are possible, but this idea leads to large performance improvements for complicated environments.

Planar sweep algorithms can be used to efficiently compute the intersections among a set of line segments, or the intersection/union of two polygons. These algorithms keep track of interesting changes as we sweep a vertical line from left to right over the data. At its leftmost position, the line intersects nothing, but as it moves to the right, it encounters a series of events:

- *Insertion* – The leftmost point of a line segment may be encountered, and it is now available to intersect some other line segment.

- *Deletion* – The rightmost point of a line segment is encountered. This means that we have completely swept over the segment, and so it can be deleted from further consideration.

- *Intersection* – If the active line segments that intersect the sweep line are maintained as sorted from top to bottom, the next intersection must occur between neighboring line segments. After this intersection, these two line segments swap their relative order.

Keeping track of what is going on requires two data structures. The future is maintained by an *event queue*, or a priority queue ordered by the x-coordinate of all possible future events of interest: insertion, deletion, and intersection. See Section 12.2 (page 373) for priority queue implementations. The present is represented by the *horizon*—an ordered list of line segments intersecting the current position of the sweepline. The horizon can be maintained using any dictionary data structure, such as a balanced tree.

To adapt this approach to compute the intersection or union of polygons, we modify the processing of the three basic event types. This algorithm can be considerably simplified for pairs of convex polygons, since (1) at most four polygon edges intersect the sweepline, so no horizon data structure is needed, and (2) no event-queue sorting is needed, since we can start from the leftmost vertex of each polygon and proceed to the right by following the polygonal ordering.

Implementations: Both LEDA (see Section 19.1.1 (page 658)) and CGAL (*www.cgal.org*) offers extensive support for line segment and polygonal intersection. In particular, they provide a C++ implementation of the Bentley-Ottmann sweepline algorithm [BO79], finding all k intersection points between n line segments in the plane in $O((n + k) \lg n)$ time.

O'Rourke [O'R01] provides a robust program in C to compute the intersection of two convex polygons. See Section 19.1.10 (page 662).

The University of North Carolina GAMMA group has produced several efficient collision detection libraries, of which *SWIFT++* [EL01] is the most recent member. It can detect intersection, compute approximate/exact distances between objects, and determine object-pair contacts in scenes composed of rigid polyhedral models. See *http://www.cs.unc.edu/∼geom/collide/* for pointers to all of these libraries.

Finding the mutual intersection of a collection of half-spaces is a special case of the convex-hulls, and Qhull [BDH97] is convex hull code of choice for general dimensions. Qhull has been widely used in scientific applications and has a well-maintained homepage at *http://www.qhull.org/*.

Notes: Mount [Mou04] is an excellent survey of algorithms for computing intersections of geometric objects such as line segments, polygons, and polyhedra. Books with chapters discussing such problems include [dBvKOS00, CLRS01, PS85]. Preparata and Shamos [PS85] provide a good exposition on the special case of finding intersections and unions of axis-oriented rectangles—a problem that arises often in VLSI design.

An optimal $O(n \lg n + k)$ algorithm for computing line segment intersections is due to Chazelle and Edelsbrunner [CE92]. Simpler, randomized algorithms achieving the same time bound are thoroughly presented by Mulmuley [Mul94].

Lin and Manocha [LM04] survey techniques and software for collision detection.

Related Problems: Maintaining arrangements (see page 614), motion planning (see page 610).