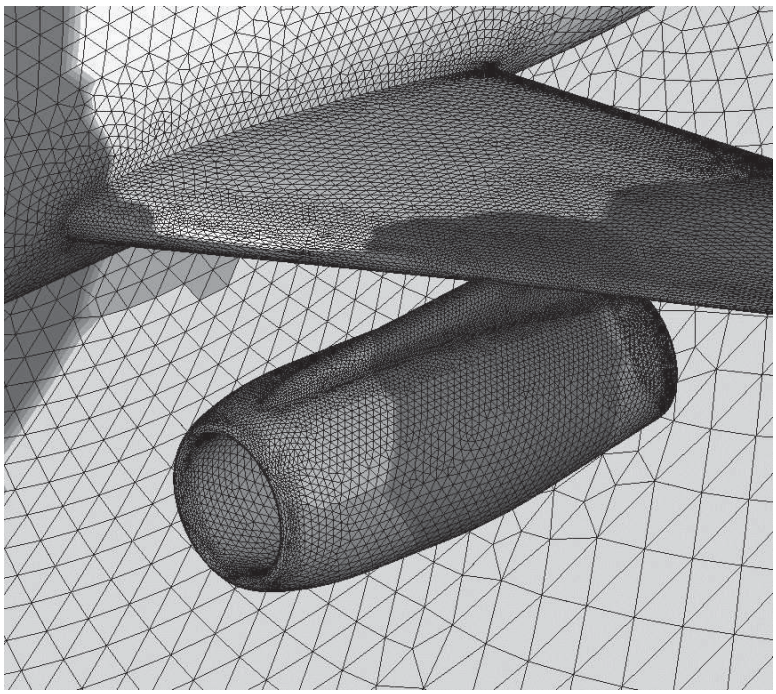
# Chapter 22 Computational Geometry



Domain decomposition for parallel processing of a wing-body-nacelle-pylon configuration. U.S. government photo. NASA.

## Contents

A common problem in robotics is to identify a trajectory from a starting point, $s$, to a target point, $t$, that avoids a certain obstacle. Among the many possible trajectories, suppose we would like to find one that is as short as possible. Let us assume that the environment is mapped out as a two-dimensional map and the obstacle is a connected region, $P$. We can compute a shortest trajectory for a robot to move from $s$ to $t$ while avoiding $P$ with the following strategy (see Figure 22.1):

1. We determine whether the line segment $\ell = \overline{st}$ intersects $P$. If it does not intersect, then $\ell$ is the shortest trajectory avoiding $P$.
2. Otherwise, if $\overline{st}$ intersects $P$, then we compute a geometric object, $H$, known as the ***convex hull***, from the vertices of polygon $P$ plus points $s$ and $t$. This object can be visualized by imagining that there are two rubber bands, connecting $s$ and $t$, with one touching $P$ on the left and one touching $P$ on the right. One of these chains would therefore be traversed by going clockwise from $s$ to $t$ and the other by going counterclockwise from $s$ to $t$.
3. We select and return the shortest of these two polygonal chains with endpoints $s$ and $t$ on $H$.

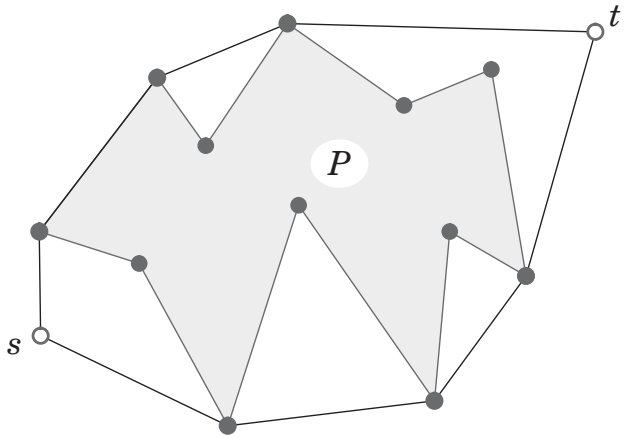



**Figure 22.1:** An example shortest trajectory from a point, $s$, to a point, $t$, avoiding a polygonal region, $P$. The shortest trajectory in this case is the clockwise chain from $s$ to $t$.

Motivated by this, and other problems dealing with geometric objects, such as points, lines, and polygons, this chapter is directed at ***computational geometry***, which studies data structures and algorithms for dealing with geometric data. We include a discussion of the convex hull problem, as well as a general method for designing efficient geometric algorithms, called the ***plane-sweep*** technique. We illustrate the applicability of this technique by showing how it can be used to solve two-dimensional computational geometry problems, by reducing them to a series of one-dimensional problems. Specifically, we apply the plane-sweep technique to the computation of the intersections of a set of orthogonal line segments and to the identification of a closest pair of points from a set of points.

# 22.1 Operations on Geometric Objects

Geometric algorithms take geometric objects of various types as their inputs. The basic geometric objects in the plane are points, lines, segments, and polygons.

There are many ways of representing planar geometric objects. Rather than give separate detailed data structures for representing points, lines, segments, and polygons, however, which would be appropriate for a book on geometric algorithms, we instead assume we have intuitive representations for these objects. For example, a two-dimensional point could be represented by a pair of numbers, a line segment could be represented by a pair of points, a line could be represented by a linear equation, and a polygon could be represented by a circular sequence of points. Still, even with our assumption that geometric objects will have intuitive representations, we nevertheless discuss a few of the algorithmic issues related to working with geometric objects in this section.

## Lines

We can represent a line $l$ as a triple, $(a, b, c)$, such that these values are the coefficients $a$, $b$, and $c$ of the linear equation,

$$ax + by + c = 0,$$

associated with $l$. Alternatively, we may specify instead two different points, $q_1$ and $q_2$, and associate them with the line that goes through both. Given the Cartesian coordinates $(x_1, y_1)$ of $q_1$ and $(x_2, y_2)$ of $q_2$, such that these points have distinct $x$- and $y$-coordinates, the equation of the line $l$ through $q_1$ and $q_2$ is given by

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1},$$

from which we derive the following:

$$\begin{aligned} a &= y_2 - y_1 \\ b &= x_1 - x_2 \\ c &= y_1(x_2 - x_1) - x_1(y_2 - y_1). \end{aligned}$$

## Line Segments

A line segment $s$ is typically represented by the pair $(p, q)$ of points in the plane that form $s$'s endpoints. We may also represent $s$ by giving the line through it, together with a range of $x$- and $y$-coordinates, that restrict this line to the segment $s$. (Why is it insufficient to include just a range of $x$- or $y$-coordinates?)

## Intersection of Two Lines

One of the most important operations that geometric objects should support is the ***intersection*** operation. For example, given two lines $l_1$ and $l_2$, we may want to know whether they ***intersect***, that is, if they have one or more points in common. If $l_1$ and $l_2$ are represented by their equations

$$a_1x + b_1y + c_1 = 0 \quad \text{and} \quad a_2x + b_2y + c_2 = 0,$$

then $l_1$ and $l_2$ intersect if and only if one of the following two conditions is satisfied.

- The determinant
$$\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1b_2 - a_2b_1$$
is nonzero, which implies that the two lines intersect in exactly one point.
- There exists a positive constant $k$ such that: $a_2 = ka_1$, $b_2 = kb_1$, and $c_2 = kc_1$, which implies that the two lines are coincident.

These tests are easily carried out in $O(1)$ time.

## Intersection of Two Line Segments

Testing whether two segments $s_1$ and $s_2$ intersect, that is, if they have one or more points in common, is disappointingly more complicated, however. The reader is encouraged to think about this problem before reading on.

The first approach that might come to mind is to test whether the lines $l_1$ and $l_2$ through $s_1$ and $s_2$, respectively, intersect. We distinguish three cases:

- If $l_1$ and $l_2$ do not intersect, then we know that $s_1$ and $s_2$ also do not intersect and we return a negative answer.
- If $l_1$ and $l_2$ intersect and are coincident, we test whether the ranges of $x$-coordinates of $s_1$ and $s_2$ overlap, and similarly for the $y$-coordinates.
- Otherwise, we compute the intersection point $q$ of $l_1$ and $l_2$. The coordinates of $q$ are given by the solution of the system of two equations in two unknowns that are derived from the equations of lines $l_1$ and $l_2$ (which can be specified as formulas, with a little extra work). Given the intersection point $q$, we can then test whether point $q$ is included both in $s_1$ and $s_2$. This can be done by checking that the $x$-coordinate of $q$ lies in the range of the $x$-coordinates of both $s_1$ and $s_2$, and similarly for the $y$-coordinate of $q$.

This testing method is admittedly somewhat complicated. An alternative approach is based on the concept of ***orientation***, which we discuss next.

## Point Orientation Testing

An important geometric relationship, which arises in many geometric algorithms, and particularly for convex hull construction, is ***orientation***. Given an ordered triplet $(p, q, r)$ of points, we say that $(p, q, r)$ makes a ***left turn*** and is oriented ***counterclockwise*** if the angle that stays on the left-hand side when going from $p$ to $q$ and then to $r$ is less than $\pi$. If the angle on the right-hand side is less than $\pi$ instead, then we say that $(p, q, r)$ makes a ***right turn*** and is oriented ***clockwise***. (See Figure 22.2.) It is possible that the angles of the left- and right-hand sides are both equal to $\pi$, in which case the three points actually do not make a turn, and we say that their orientation is ***collinear***.

Given a triplet $(p_1, p_2, p_3)$ of three points $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, and $p_3 = (x_3, y_3)$, in the plane, let $\Delta(p_1, p_2, p_3)$ be the determinant defined by

$$\Delta(p_1, p_2, p_3) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 - x_2y_1 + x_3y_1 - x_1y_3 + x_2y_3 - x_3y_2. \quad (22.1)$$

The function $\Delta(p_1, p_2, p_3)$ is often called the "signed area" function, because its absolute value is twice the area of the (possibly degenerate) triangle formed by the points $p_1$, $p_2$, and $p_3$. In addition, we have the following important fact relating this function to orientation testing.

**Theorem 22.1:** *The orientation of a triplet $(p_1, p_2, p_3)$ of points in the plane is counterclockwise, clockwise, or collinear, depending on whether $\Delta(p_1, p_2, p_3)$ is positive, negative, or zero, respectively.*

We sketch the proof of Theorem 22.1; we leave the details as an exercise (R-22.2). In Figure 22.2, we show a triplet $(p_1, p_2, p_3)$ of points with $x_1 < x_2 < x_3$. Clearly, this triplet makes a left turn if the slope of segment $p_2p_3$ is greater than the slope of segment $p_1p_2$. This is expressed by the following question:

$$\text{Is } \frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1} \text{ ?} \quad (22.2)$$

By the expansion of $\Delta(p_1, p_2, p_3)$ shown in 22.1, we can verify that inequality 22.2 is equivalent to $\Delta(p_1, p_2, p_3) > 0$.

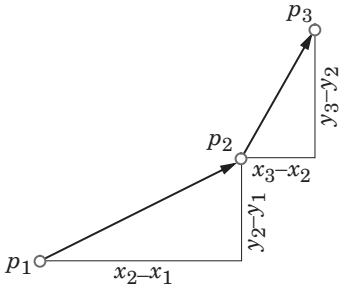



**Figure 22.2:** An example of a left turn. The differences between the coordinates between $p_1$ and $p_2$ and the coordinates of $p_2$ and $p_3$ are also illustrated.

In fact, there is a quick way to show the above equivalence using some facts about determinants. By a standard property of determinants, we can subtract a row from another row without changing the determinant. Hence,

$$\begin{aligned}
\Delta(p_1,p_2,p_3) &= \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \\
&= \begin{vmatrix} x_1 & y_1 & 1 \\ (x_2-x_1) & (y_2-y_1) & 0 \\ (x_3-x_2) & (y_3-y_2) & 0 \end{vmatrix} \\
&= (x_2-x_1)(y_3-y_2)-(y_2-y_1)(x_3-x_2).
\end{aligned}$$

Thus, testing if $\Delta(p_1,p_2,p_3) > 0$ is equivalent to answering question 22.2.

**Example 22.2:** *Using the notion of orientation, let us consider the problem of testing whether two line segments $s_1$ and $s_2$ intersect. Specifically, Let $s_1 = \overline{p_1q_1}$ and $s_2 = \overline{p_2q_2}$ be two segments in the plane. $s_1$ and $s_2$ intersect if and only if* ***one of*** *the following two conditions is verified:*

1. (a) *$(p_1,q_1,p_2)$ and $(p_1,q_1,q_2)$ have different orientations,* ***and***
   (b) *$(p_2,q_2,p_1)$ and $(p_2,q_2,q_1)$ have different orientations.*
2. (a) *$(p_1,q_1,p_2)$, $(p_1,q_1,q_2)$, $(p_2,q_2,p_1)$, and $(p_2,q_2,q_1)$ are all collinear,* ***and***
   (b) *the $x$-projections of $s_1$ and $s_2$ intersect,* ***and***
   (c) *the $y$-projections of $s_1$ and $s_2$ intersect.*

*Condition 1 is illustrated in Figure 22.3. We also show, in Table 22.4, the respective orientation of the triplets $(p_1,q_1,p_2)$, $(p_1,q_1,q_2)$, $(p_2,q_2,p_1)$, and $(p_2,q_2,q_1)$ in each of the four cases for Condition 1. A complete proof is left as an exercise (R-22.3). Note that the conditions also hold if $s_1$ and/or $s_2$ is a degenerate segment with coincident endpoints.*

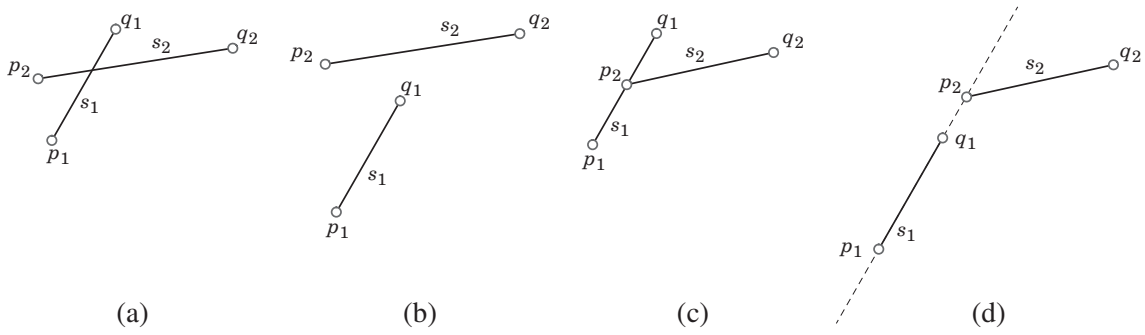



**Figure 22.3:** Examples illustrating four cases of Condition 1 of Example 22.2.

| case | $(p_1, q_1, p_2)$ | $(p_1, q_1, q_2)$ | $(p_2, q_2, p_1)$ | $(p_2, q_2, q_1)$ | intersection? |
|---|---|---|---|---|---|
| (a) | CCW | CW | CW | CCW | yes |
| (b) | CCW | CW | CW | CW | no |
| (c) | COLL | CW | CW | CCW | yes |
| (d) | COLL | CW | CW | CW | no |

**Table 22.4:** The four cases shown in Figure 22.3 for the orientations specified by Condition 1 of Example 22.2, where CCW stands for counterclockwise, CW stands for clockwise, and COLL stands for collinear.

## Polygons

We can represent a polygon $P$ by a cyclical sequence of points, called the ***vertices*** of $P$. (See Figure 22.5.) The segments between consecutive vertices of $P$ are called the ***edges*** of $P$. Polygon $P$ is said to be ***simple***, if intersections between pairs of edges of $P$ happen only at a common endpoint vertex. A polygon is ***convex*** if it is simple and all its internal angles are less than $\pi$.

Our discussion of different ways of representing points, lines, segments, and polygons is not meant to be exhaustive. It is meant simply to indicate the different ways we can implement these geometric objects.

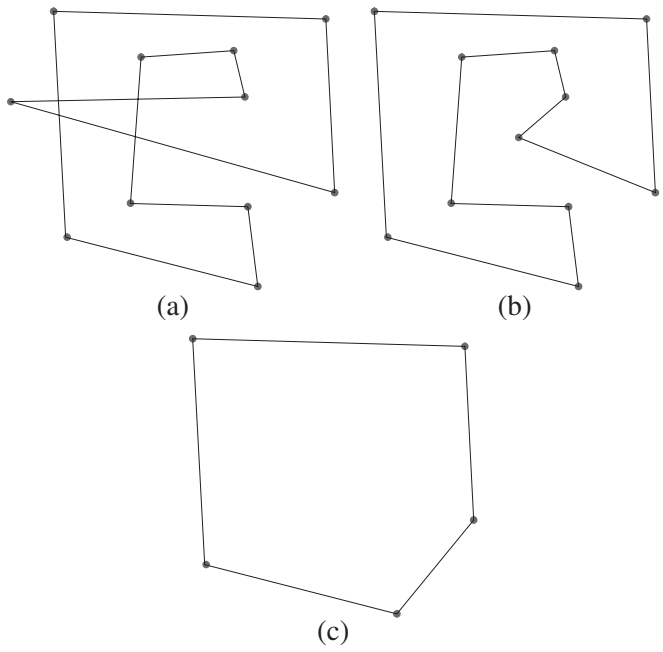
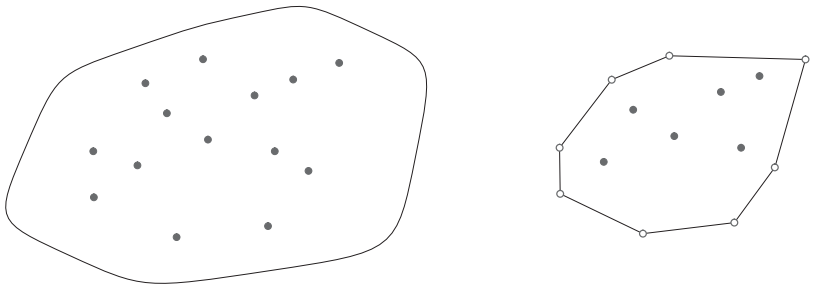



**Figure 22.5:** Examples of polygons: (a) intersecting, (b) simple, (c) convex.

## 22.2 Convex Hulls

One of the most studied geometric problems is that of computing the convex hull of a set of points. Informally speaking, the ***convex hull*** of a set of points in the plane is the shape taken by a rubber band that is placed "around the points" and allowed to shrink to a state of equilibrium. (See Figure 22.6.)



(a) (b)

**Figure 22.6:** The convex hull of a set of points in the plane: (a) an example "rubber band" placed around the points; (b) the convex hull of the points.

The convex hull corresponds to the intuitive notion of a "boundary" of a set of points and can be used to approximate the shape of a complex object. Indeed, computing the convex hull of a set of points is a fundamental operation in computational geometry.

### Basic Properties of Convex Hulls

We say that a region $R$ is ***convex*** if any time two points $p$ and $q$ are in $R$, the entire line segment $\overline{pq}$ is also in $R$. The ***convex hull*** of a set of points $S$ is the boundary of the smallest convex region that contains all the points of $S$ inside it or on its boundary. The notion of "smallest" refers to either the perimeter or area of the region, both definitions being equivalent. The convex hull of a set of points $S$ in the plane defines a convex polygon, and the points of $S$ on the boundary of the convex hull define the vertices of this polygon.

There are a number of applications of the convex hull problem, in addition to the robot motion planning problem mentioned above, including partitioning problems, shape testing problems, and separation problems. For example, if we wish to determine whether there is a half-plane (that is, a region of the plane on one side of a line) that completely contains a set of points $A$ but completely avoids a set of points $B$, it is enough to compute the convex hulls of $A$ and $B$ and determine whether they intersect each other.

There are many interesting geometric properties associated with convex hulls. The following theorem provides an alternate characterization of the points that are on the convex hull and of those that are not.

**Theorem 22.3:** *Let $S$ be a set of planar points with convex hull $H$. Then*

- *A pair of points $a$ and $b$ of $S$ form an edge of $H$ if and only if all the other points of $S$ are contained on one side of the line through $a$ and $b$.*
- *A point $p$ of $S$ is a vertex of $H$ if and only if there exists a line $l$ through $p$, such that all the other points of $S$ are contained in the same half-plane delimited by $l$ (that is, they are all on the same side of $l$).*
- *A point $p$ of $S$ is not a vertex of $H$ if and only if $p$ is contained in the interior of a triangle formed by three other points of $S$ or in the interior of a segment formed by two other points of $S$.*

The properties expressed by Theorem 22.3 are illustrated in Figure 22.7. A complete proof of them is left as an exercise (R-22.4). As a consequence of Theorem 22.3, we can immediately verify that, in any set $S$ of points in the plane, the following ***critical*** points are always on the boundary of the convex hull of $S$:

- A point with minimum or maximum $x$-coordinate
- A point with minimum or maximum $y$-coordinate.

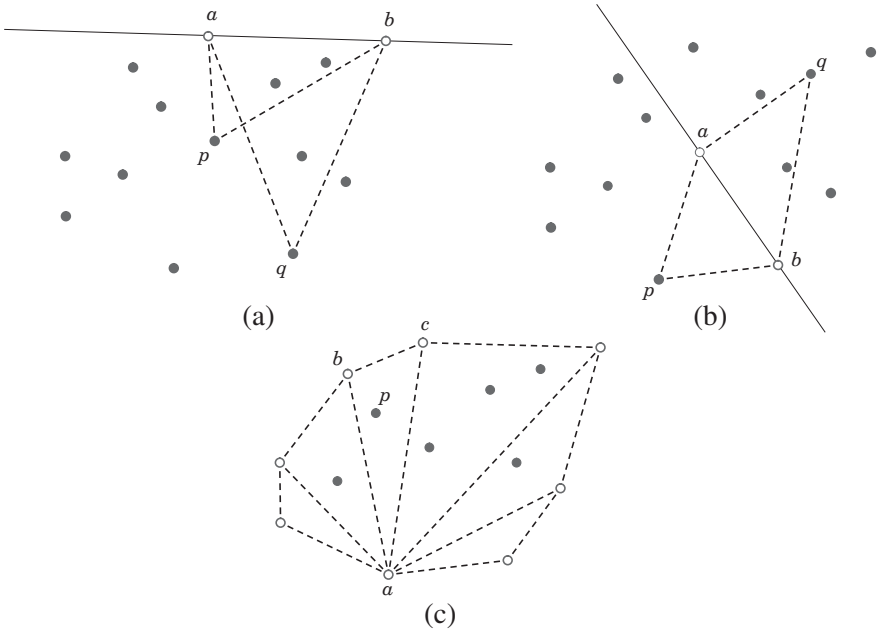



**Figure 22.7:** Illustration of the properties of the convex hull given in Theorem 22.3: (a) points $a$ and $b$ form an edge of the convex hull; (b) points $a$ and $b$ do not form an edge of the convex hull; (c) point $p$ is not on the convex hull.

### 22.2.1 The Gift Wrapping Algorithm

Theorem 22.3 basically states that we can identify a particular point, say one with minimum $y$-coordinate, that provides an initial starting configuration for an algorithm that computes the convex hull. The ***gift wrapping*** algorithm for computing the convex hull of a set of points in the plane is based on just such a starting point and can be intuitively described as follows (see Figure 22.8):

1. View the points as pegs implanted in a level field, and imagine that we tie a rope to the peg corresponding to the point $a$ with minimum $y$-coordinate (and minimum $x$-coordinate if there are ties). Call $a$ the ***anchor point***, and note that $a$ is a vertex of the convex hull.
2. Pull the rope to the right of the anchor point and rotate it counterclockwise until it touches another peg, which is the next vertex of the convex hull.
3. Continue rotating the rope counterclockwise, identifying a new vertex of the convex hull at each step, until the rope gets back to the anchor point.

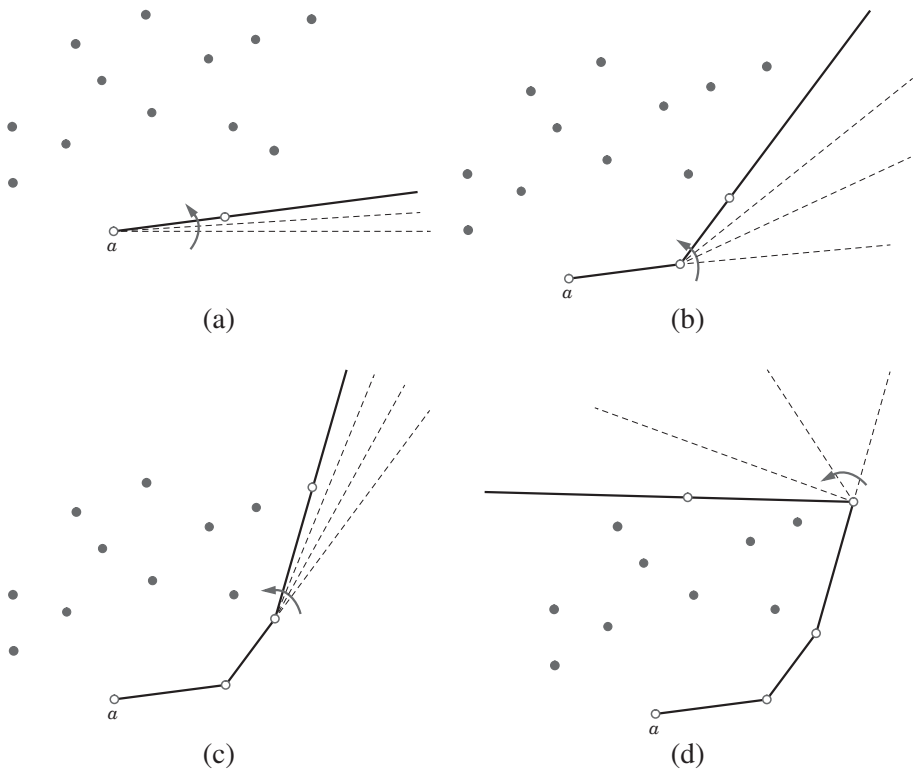



**Figure 22.8:** Initial four wrapping steps of the gift wrapping algorithm.

### Using Orientation Testing for the Wrapping Step

Each time we rotate our "rope" around the current peg until it hits another point, we perform an operation called a ***wrapping*** step. Geometrically, a wrapping step involves starting from a given line $L$ known to be tangent to the convex hull at the current anchor point $a$, and determining the line through $a$ and another point in the set making the smallest angle with $L$. Implementing this wrapping step does not require trigonometric functions and angle calculations, however. Instead, we can use the following theorem, which follows from Theorem 22.3.

**Theorem 22.4:** *Let $S$ be a set of points in the plane, and let $a$ be a vertex of the convex hull, $H$, of $S$. The next vertex of $H$, going counterclockwise from $a$, is the point, $p$, such that triplet $(a, p, q)$ makes a left turn with every other point $q$ of $S$.*

### Details of the Gift Wrapping Algorithm

We give the details in Algorithm 22.9, which starts with the set, $S$, of $n$ points and a point, $a$, in $S$ on the convex hull, $H$, of $S$, which could be, say, a point with minimum $x$-coordinate. Note that finding such a point, $a$, can easily be done in $O(n)$ time.

**Algorithm** GiftWrap$(S, a)$:

***Input:*** A set, $S$, of points in the plane beginning with point $a$, such that $a$ is on the convex hull of $S$
***Output:*** List $H$ of the convex hull vertices of $S$ in counterclockwise order

```
H ← [a]
repeat
    Let b be the last element in H
    Let p be a vertex in S such that p ≠ b
    for each point, q, in S, with q ≠ p and q ≠ b do
        if (b, p, q) forms a right turn then
            p ← q
    Add p to the end of the list H
until p = a
return H
```

**Algorithm 22.9:** The gift wrapping algorithm. Here we make the simplifying assumption that no three of the points in $S$ are collinear.

## Analysis of the Gift Wrapping Algorithm

Let us analyze the running time of the gift wrapping algorithm. Let $n$ be the number of points of $S$, and let $h \leq n$ be the number of vertices of the convex hull $H$ of $S$. Let $p_0, \ldots, p_{h-1}$ be the vertices of $H$. Finding the anchor point $a = p_0$ takes $O(n)$ time. Since with each wrapping step of the algorithm we discover a new vertex of the convex hull, the number of wrapping steps is equal to $h$.

Step $i$ is a minimum-finding computation based on finding a point, $p$, such that every other point, $q$, in $S$ forms a left turn when going from $b$ to $p$ to $q$. This step runs in $O(n)$ time, since determining the orientation of a triplet takes $O(1)$ time and we must examine all the points of $S$ to find the smallest with respect to the orientation test. Our description assumed that no three points in $S$ were collinear, but the above algorithm can be extended to handing this case without effecting the asymptotic performance of the algorithm. Thus, we can summarize the performance of the gift wrapping algorithm with the following theorem.

**Theorem 22.5:** *Given a set, $S$, of $n$ points in the plane, the gift wrapping algorithm constructs a list representation of the convex hull of $S$ in $O(nh)$ time, where $h$ is the number of vertices on the boundary of the convex hull.*

## Output Sensitivity

We note that $h$ can be as large as $n$, that is, when all the points in $S$ are vertices on the convex hull; hence, in the worst case, this algorithm runs in $O(n^2)$ time. Still, $h$ can be as small as 3, in which case the gift wrapping algorithm would run in $O(n)$ time. Thus, there is a large difference between the best-case and worst-case performance of the gift wrapping algorithm.

This algorithm is nevertheless reasonably efficient in practice, however, for it can take advantage of the (common) situation when $h$, the number of hull points, is small relative to the number of input points, $n$. That is, this algorithm is said to be an ***output sensitive*** algorithm—an algorithm whose running time depends on the size of the output. Gift wrapping has a running time that varies between linear and quadratic, and is efficient if the convex hull has few vertices. For instance, if we choose $n$ points uniformly and independently at random in the interior of a rectangle, then the expected number of points on their convex hull is $O(\log n)$.

We discuss in the next section an algorithm that is efficient for all hull sizes, although it is slightly more complicated.

### 22.2.2 The Graham Scan Algorithm

A convex hull algorithm that has an efficient running time no matter how many points are on the boundary of the convex is the ***Graham scan*** algorithm. The Graham scan algorithm for computing the convex hull $H$ of a set $P$ of $n$ points in the plane consists of the following three phases:

1. We find a point $a$ of $P$ that is a vertex of $H$ and call it the ***anchor point***. We can, for example, pick as our anchor point $a$ the point in $P$ with minimum $y$-coordinate (and minimum $x$-coordinate if there are ties).
2. We sort the remaining points of $P$ (that is, $P - \{a\}$) radially around $a$, and let $S$ be the resulting sorted list of points. (See Figure 22.10.) In the list $S$, the points of $P$ appear sorted counterclockwise "by angle" with respect to the anchor point $a$, although no explicit computation of angles is performed.

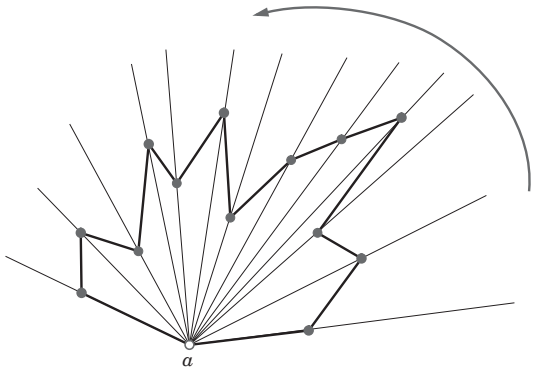



**Figure 22.10:** Sorting around the anchor point in the Graham scan algorithm.

3. After adding the anchor point $a$ at the first and last position of $S$, we ***scan*** through the points in $S$ in (radial) order, maintaining at each step a list $H$ storing a convex chain "surrounding" the points scanned so far. Each time we consider new point $p$, we perform the following test:

   (a) If $p$ forms a left turn with the last two points in $H$, or if $H$ contains fewer than two points, then add $p$ to the end of $H$.

   (b) Otherwise, remove the last point in $H$ and repeat the test for $p$.

   We stop when we return to the anchor point $a$, at which point $H$ stores the vertices of the convex hull of $P$ in counterclockwise order.

The details of the scan phase (Phase 3) are spelled out in Algorithm Scan, described in Algorithm 22.11.

**Algorithm** Scan$(S, a)$:
***Input:*** A list $S$ of points in the plane beginning with point $a$, such that $a$ is on the convex hull of $S$ and the remaining points of $S$ are sorted counterclockwise around $a$
***Output:*** List $S$ with only convex hull vertices remaining

```
S.insertLast(a)        // add a copy of a at the end of S
prev ← S.first()       // so that prev = a initially
curr ← S.after(prev)   // the next point is on the current convex chain
repeat
    next ← S.after(curr)      // advance
    if points (point(prev), point(curr), point(next)) make a left turn then
        prev ← curr
    else
        S.remove(curr)        // point curr is not in the convex hull
        prev ← S.before(prev)
    curr ← S.after(prev)
until curr = S.last()
S.remove(S.last())     // remove the copy of a
```

**Algorithm 22.11:** The scan phase of the Graham scan convex hull algorithm. (See Figure 22.12.) Variables *prev*, *curr*, and *next* are positions (Section 2.2.2) of the list $S$. We assume that an accessor method point$(pos)$ is defined that returns the point stored at position $pos$. We give a simplified description of the algorithm that works only if $S$ has at least three points, and no three points of $S$ are collinear.

## Analysis of the Graham Scan Algorithm

Let us now analyze the running time of the Graham scan algorithm, which is illustrated in Figure 22.12. We denote the number of points in $P$ (and $S$) with $n$. The first phase (finding the anchor point) clearly takes $O(n)$ time. The second phase (sorting the points around the anchor point) takes $O(n \log n)$ time provided we use one of the asymptotically optimal sorting algorithms, such as heap-sort (Section 5.4) or merge-sort (Section 8.1). The analysis of the scan (third) phase is more subtle. To analyze the scan phase of the Graham scan algorithm, let us look more closely at the **repeat** loop of Algorithm 22.11. At each iteration of the loop, either variable *next* advances forward by one position in the list $S$ (successful **if** test), or variable *next* stays at the same position but a point is removed from $S$ (unsuccessful **if** test). Hence, the number of iterations of the **repeat** loop is at most $2n$. Therefore, each statement of algorithm Scan is executed at most $2n$ times. Since each statement requires the execution of $O(1)$ elementary operations in turn, algorithm Scan takes $O(n)$ time. In conclusion, the running time of the Graham scan algorithm is dominated by the second phase, where sorting is performed. Thus, the Graham scan algorithm runs in $O(n \log n)$ time.

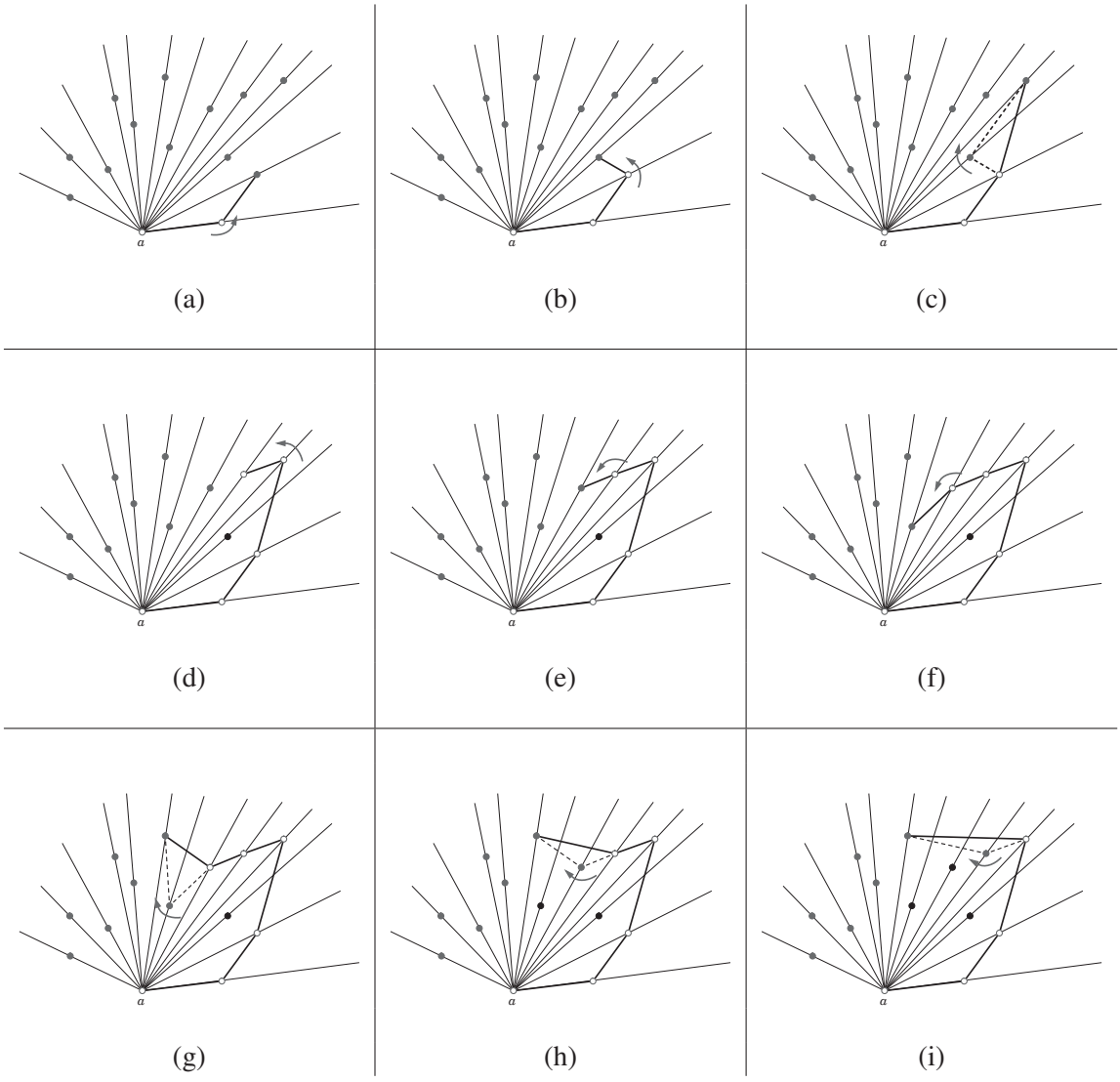



**Figure 22.12:** Third phase of the Graham scan algorithm (see Algorithm 22.11).

## 22.3 Segment Intersection

The next problem we consider is that of finding all the intersecting pairs among a set of $n$ line segments. Of course, we could apply a brute-force algorithm to check every pair of segments to see whether they intersect. Since the number of pairs is $n(n-1)/2$, this algorithm takes $O(n^2)$ time, since we can test any pair for intersection in constant time. If all the pairs intersect, this algorithm is optimal. Still, we would like to have a faster method for the case where the number of intersecting pairs is small or there are no intersections at all. Specifically, if $s$ is the number of intersecting pairs, we would like to have an output sensitive algorithm whose running time depends on both $n$ and $s$. We shall present an algorithm that uses the plane-sweep technique and runs in $O(n \log n + s)$ time for the case when the input set of segments consists of $n$ ***orthogonal segments***, meaning that each segment in the set is either horizontal or vertical.

Before we proceed with our algorithm, we make a slight digression to review a problem discussed previously in this book. This problem is the one-dimensional range-searching problem, in which we wish to dynamically maintain a dictionary of numbers (that is, points on a number line), subject to insertions and deletions and queries of the following form:

findAllInRange$(k_1, k_2)$: Return a list of all the elements in $D$ with key $k$, such that $k_1 \leq k \leq k_2$.

We show in Section 3.2 how we can use any balanced binary search tree to maintain such a dictionary in order to achieve $O(\log n)$ time for insertion and removal, and $O(\log n + s)$ time for answering findAllInRange queries, where $n$ is the number of points in the dictionary and $s$ is the number of returned points in the range.

Let us return to the problem at hand, which is to compute all intersecting pairs of segments from a collection of $n$ horizontal and vertical segments. The main idea of the algorithm for solving this problem is to reduce this two-dimensional problem to a collection of one-dimensional range-searching problems. In so doing, we are going to make use of a technique known as ***plane sweeping***. In using this technique, we solve a static two-dimensional problem by imagining that we sweep the plane with a vertical line, $L$. During the sweep, we maintain various data structures for the objects that interact with $L$, and at various ***events*** we pause the sweep to update and/or query these data structures. In the case of orthogonal segment intersection, for each vertical segment $v$, we consider an event corresponding to a stopping of $L$ at the vertical line $l(v)$ through $v$, and plunge into the "one-dimensional world" of line $l(v)$. (See Figure 22.13.) Only the vertical segment $v$ and the intersections of horizontal segments with $l(v)$ exist in this world. In particular, segment $v$ corresponds to an interval of $l(v)$, a horizontal segment $h$ intersecting $l(v)$ corresponds to a point on $l(v)$, and the horizontal segments crossing $v$ correspond to the points on $l(v)$ contained in the interval.

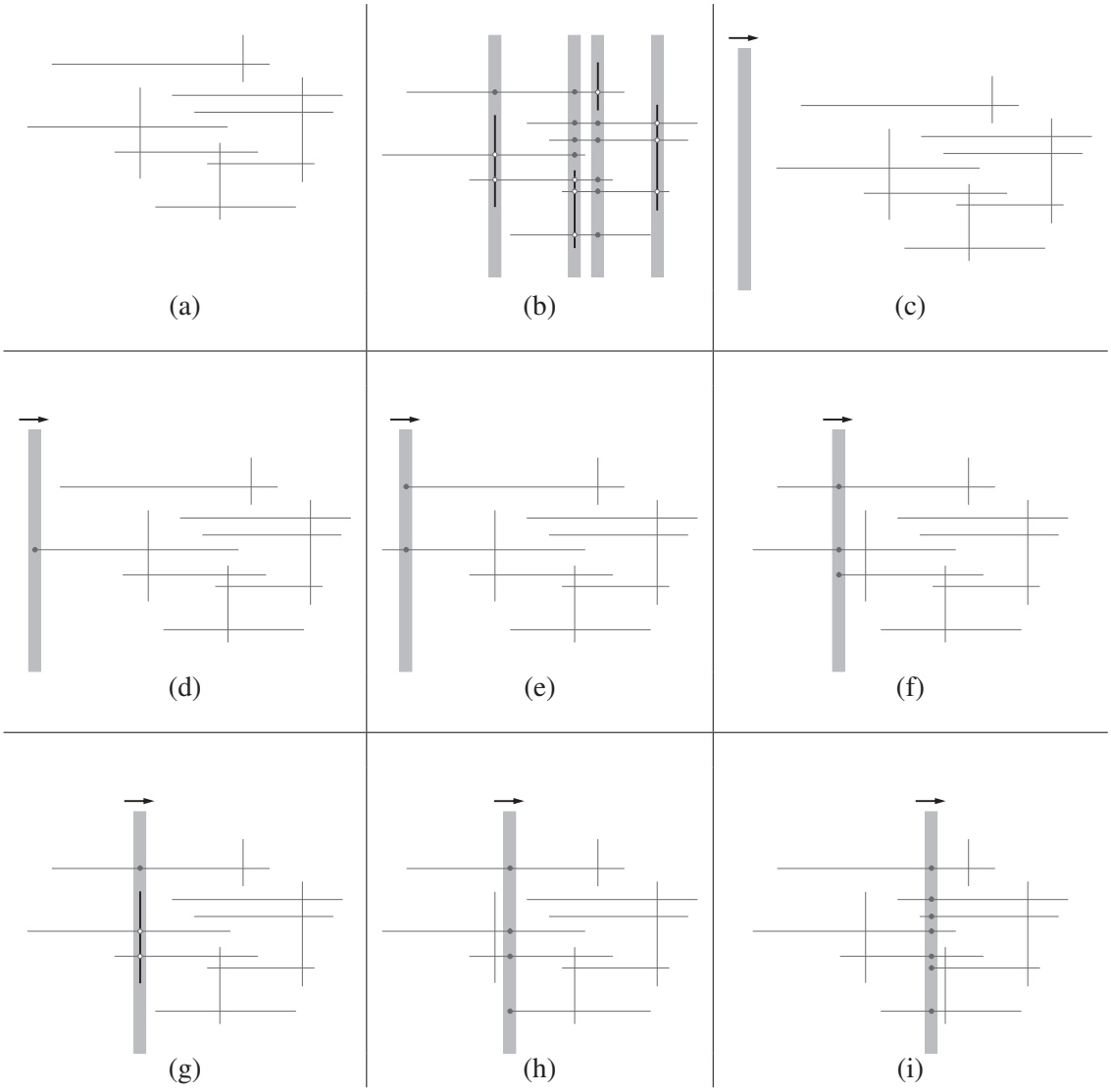



**Figure 22.13:** Plane sweep for orthogonal segment intersection: (a) a set of horizontal and vertical segments; (b) the collection of one-dimensional range-search problems; (c) beginning of the plane sweep (the ordered dictionary $S$ of horizontal segments is empty); (d) the first (left-endpoint) event, causing an insertion into $S$; (e) the second (left-endpoint) event, causing another insertion into $S$; (f) the third (left-endpoint) event, causing yet another insertion into $S$; (g) the first vertical-segment event, causing a range search in $S$ (two intersections reported); (h) the next (left-endpoint) event, causing an insertion into $S$; (i) a left-endpoint event three events later, causing an insertion into $S$.

## The Plane-Sweep Segment Intersection Algorithm

Suppose, then, that we are given a set of $n$ horizontal and vertical segments in the plane. We will determine all pairs of intersecting segments in this set by using the ***plane sweep*** technique and the collective approach suggested by the above idea. This algorithm involves simulating the sweeping of a vertical line, $L$, over the segments, moving from left to right, starting at a location to the left of all the input segments. During the sweep, the set of horizontal segments currently intersected by the sweep line is maintained by means of insertions into and removals from a dictionary ordered by $y$-coordinate. When the sweep encounters a vertical segment $v$, a range query on the dictionary is performed to find the horizontal segments intersecting $v$.

Specifically, during the sweep, we maintain an ordered dictionary, $S$, storing horizontal segments with their keys given by their $y$-coordinates. The sweep pauses at certain ***events*** that trigger the ***actions*** shown in Table 22.14, which are illustrated above in Figure 22.13. We give the details in Algorithm 22.15.

| Event | Action |
|---|---|
| ***left endpoint*** of a horizontal segment $h$ | insert $h$ into dictionary $S$ |
| ***right endpoint*** of a horizontal segment $h$ | remove $h$ from dictionary $S$ |
| ***vertical segment*** $v$ | perform a range search on $S$ with selection range given by the $y$-coordinates of the endpoints of $v$ |

**Table 22.14:** Events triggering actions in the plane-sweep algorithm for orthogonal segment intersection.

**Algorithm** SegmentIntersect($C$):
***Input:*** A collection, $C$, of horizontal and vertical line segments
***Output:*** All the intersecting horizontal-vertical pairs of segments in $C$

Let $S$ be an initially empty dictionary of objects ordered by $y$-coordinates
Let $E$ be a sorted listing of the segments in $C$ by $x$-coordinates
**for** each element, $e$, in $E$ in sorted order **do**
  **if** $e$ is a left endpoint of a horizontal segment, $h = ([x_1, x_2],\ y)$ **then**
    insert $h$ into $S$ with key $y$
  **else if** $e$ is a right endpoint of a horizontal segment, $h = ([x_1, x_2],\ y)$ **then**
    remove $h$ from $S$ using the key $y$
  **else if** $e$ is a vertical segment, $v = (x,\ [y_1, y_2])$ **then**
    Report all segments in $S$ in the range $[y_1, y_2]$ as intersecting $v$

**Algorithm 22.15:** Orthogonal segment intersection reporting algorithm.

### Analysis of the Plane-Sweep Line Intersection Algorithm

To analyze the running time of this plane-sweep algorithm, first note that we must identify all the events and sort them by $x$-coordinate. An event is either an endpoint of a horizontal segment or a vertical segment. Hence, the number of events is at most $2n$. When sorting the events, we compare them by $x$-coordinate, which takes $O(1)$ time. Using one of the asymptotically optimal sorting algorithms, such as heap-sort (Section 5.4) or merge-sort (Section 8.1), we can order the events in $O(n \log n)$ time. The operations performed on the dictionary $S$ are insertions, removals, and range searches. Each time an operation is executed, the size of $S$ is at most $2n$.

We can implement $S$ as a balanced binary search tree (Chapter 4), so that insertions and deletions each take $O(\log n)$ time. As we have reviewed above, range searching in an $n$-element ordered dictionary can be performed in $O(\log n + s)$ time, using $O(n)$ space, where $s$ is the number of items reported. Let us characterize, then, the running time of a range search triggered by a vertical segment $v$ as

$$O(\log n + s(v)),$$

where $s(v)$ is the number of horizontal segments currently in the dictionary $S$ that intersect $v$. Thus, indicating the set of vertical segments with $V$, the running time of the sweep is

$$O\left(2n \log n + \sum_{v \in V} (\log n + s(v))\right).$$

Since the sweep goes through all the segments, the sum of $s(v)$ over all the vertical segments encountered is equal to the total number $s$ of intersecting pairs of segments. Hence, we conclude that the sweep takes time $O(n \log n + s)$.

In summary, the complete segment intersection algorithm, outlined above, consists of the event sorting step followed by the sweep step. Sorting the events takes $O(n \log n)$ time, while sweeping takes $O(n \log n + s)$ time. Thus, the running time of the algorithm is $O(n \log n + s)$. This gives us the following.

**Theorem 22.6:** *Given a collection, $C$, of $n$ horizontal and vertical line segments in the plane, we can report all pairs of intersecting horizontal-vertical pairs of segments in $O(n \log n + s)$ time, where $s$ is the number of pairs.*

Thus, we have an efficient output-sensitive algorithm for orthogonal line segment intersection reporting.

## 22.4 Finding a Closest Pair of Points

Another geometric problem that can be solved using the plane-sweep technique involves the concept of ***proximity***, which is the relationship of ***distance*** that exists between geometric objects. Specifically, we focus on the ***closest pair*** problem, which consists of finding a pair of points $p$ and $q$ that are at a minimum distance from each other in a set of $n$ points. This pair is said to be a closest pair. We will use the Euclidean definition of the distance between two points $a$ and $b$:

$$\mathsf{dist}(a,b) = \sqrt{(x(a)-x(b))^2 + (y(a)-y(b))^2},$$

where $x(p)$ and $y(p)$ respectively denote the $x$- and $y$-coordinates of the point $p$. Applications of the closest pair problem include the verification of mechanical parts and integrated circuits, where it is important that certain separation rules between components be respected.

A straightforward "brute-force" algorithm for solving the closest pair problem is to compute the distance between every pair of points and select a pair with minimum distance. Since the number of pairs is $n(n-1)/2$, this algorithm takes $O(n^2)$ time. We can apply a more clever strategy, however.

It turns out that we can effectively apply the plane-sweep technique to the closest pair problem. We solve the closest pair problem, in this case, by imagining that we sweep the plane by a vertical line from left to right, starting at a position to the left of all $n$ of the input points. As we sweep the line across the plane, we keep track of the closest pair seen so far, and of all those points that are "near" the sweep line. We also keep track of the distance, $d$, between the closest pair seen so far. In particular, as we illustrate in Figure 22.16, while sweeping through the points from left to right, we maintain the following data:

- A closest pair $(a,b)$ among the points encountered, and the distance $d = \mathsf{dist}(a,b)$
- An ordered dictionary $S$ that stores the points lying in a strip of width $d$ to the left of the sweep line and uses the $y$-coordinates of points as keys.

Each input point $p$ corresponds to an event in this plane sweep. When the sweep line encounters a point $p$, we perform the following actions:

1. We update dictionary $S$ by removing the points at horizontal distance greater than $d$ from $p$, that is, each point $r$ such that $x(p) - x(r) > d$.
2. We find the closest point $q$ to the left of $p$ by searching in dictionary $S$ (we will say in a moment how this is done). If $\mathsf{dist}(p,q) < d$, then we update the current closest pair and distance by setting $a \leftarrow p$, $b \leftarrow q$, and $d \leftarrow \mathsf{dist}(p,q)$.
3. We insert $p$ into $S$.

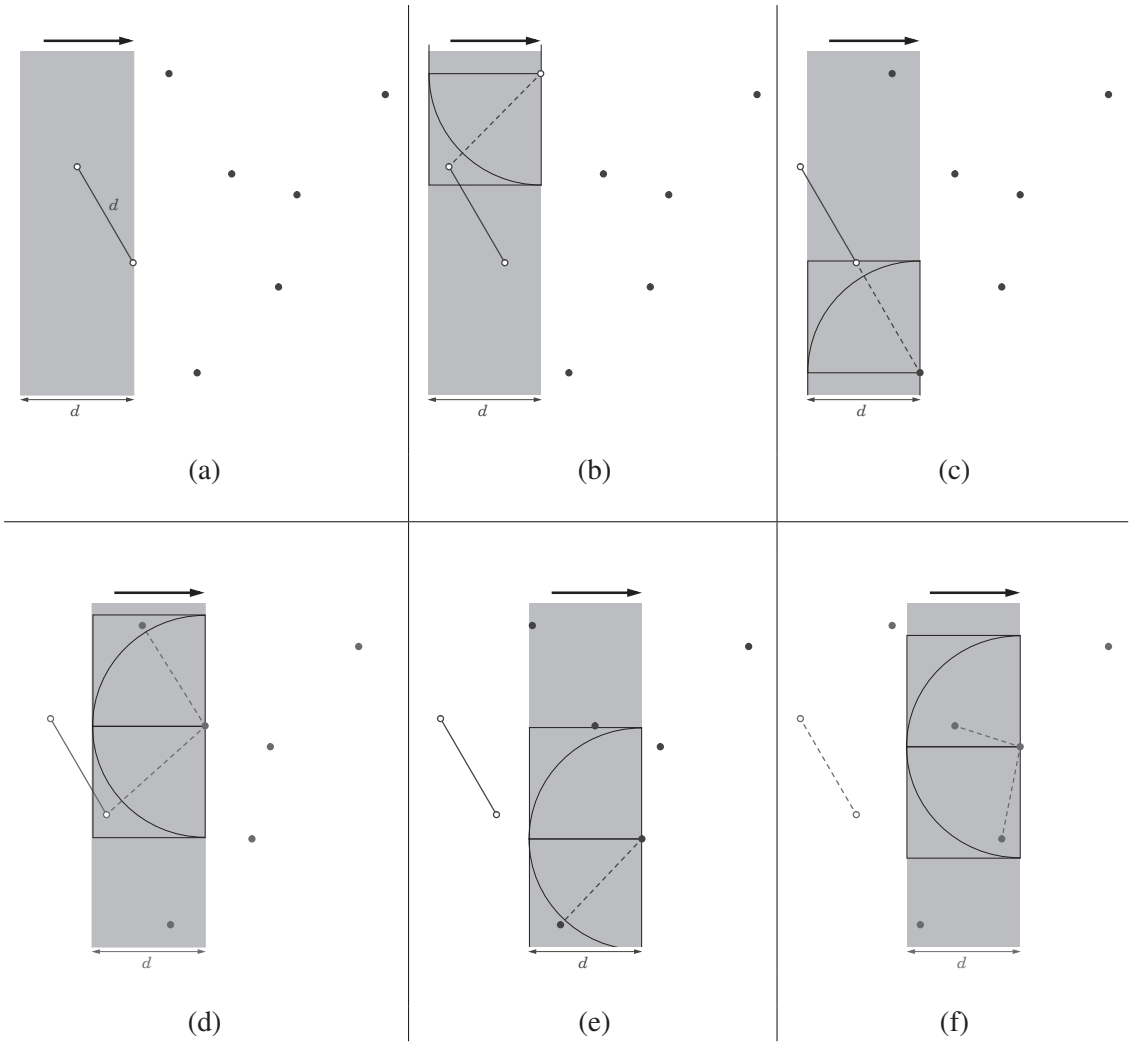



**Figure 22.16:** Plane sweep for the closest pair problem: (a) the first minimum distance $d$ and closest pair (highlighted); (b) the next event (box $B(p, d)$ contains a point, but the half-circle $C(p, d)$ is empty); (c) next event ($C(p, d)$ again is empty, but a point is removed from $S$); (d) next event (with $C(p, d)$ again empty). The dictionary $S$ contains the points in the gray strip of width $d$; (e) a point $p$ is encountered (and a point removed from $S$), with $B(p, d)$ containing 1 point, but $C(p, d)$ containing none; thus, the minimum distance $d$ and closest pair $(a, b)$ stay the same; (f) a point $p$ encountered with $C(p, d)$ containing 2 points (and a point is removed from $S$).

Clearly, we can restrict our search of the closest point $q$ to the left of $p$ to the points in dictionary $S$, since all other points will have distance greater than $d$. What we want are those points in $S$ that lie within the half-circle $C(p, d)$ of radius $d$ centered at and to the left of point $p$. (See Figure 22.17.) As a first approximation, we can get the points in the enclosing $d \times 2d$ rectangular box $B(p, d)$ of $C(p, d)$ (Figure 22.17) by performing a range search on $S$ for the points in $S$ with $y$-coordinates in the interval of keys $[y(p) - d, y(p) + d]$. We examine such points, one by one, and find the closest to $p$, denoted $q$. Since the operations performed on dictionary $S$ are range searches, insertions, and removals of points, we implement $S$ by means of an AVL tree or red-black tree.

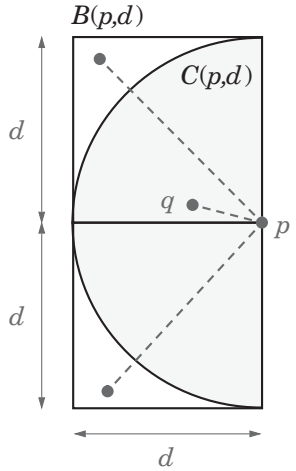



**Figure 22.17:** Box $B(p, d)$ and half-circle $C(p, d)$.

The following intuitive property, whose proof is left as an exercise (R-22.6), is crucial to the analysis of the running time of the algorithm.

**Theorem 22.7:** *A rectangle of width $d$ and height $2d$ can contain at most six points such that any two points are at distance at least $d$.*

Thus, there are at most six points of $S$ that lie in the box $B(p, d)$. So the range-search operation on $S$, to find the points in $B(p, d)$, takes time $O(\log n + 6)$, which is $O(\log n)$. Also, we can find the point in $B(p, d)$ closest to $p$ in $O(1)$ time.

Before we begin the sweep, we sort the points by $x$-coordinate, and store them in an ordered list $X$. The list $X$ is used for two purposes:

- To get the next point to be processed
- To identify the points to be removed from dictionary $S$.

We keep references to two positions in the list $X$, which we denote as firstInStrip and lastInStrip. Position lastInStrip keeps track of the new point to be inserted into $S$, while position firstInStrip keeps track of the left-most point in $S$. By advancing

lastInStrip one step at a time, we find the new point to be processed. By using firstInStrip, we identify the points to be removed from $S$. Namely, while we have

$$x(\mathsf{point}(\mathsf{firstInStrip})) < x(\mathsf{point}(\mathsf{lastInStrip})) - d,$$

we perform operation $\mathsf{remove}(y(\mathsf{point}(\mathsf{firstInStrip})))$ on dictionary $S$ and advance firstInStrip.

Let $n$ be the number of input points. Our analysis of the plane-sweep algorithm for the closest pair problem is based on the following observations:

- The preliminary sorting by $x$-coordinate takes time $O(n \log n)$.
- Each point is inserted once and removed once from dictionary $S$, which has size at most $n$; hence, the total time for inserting and removing elements in $S$ is $O(n \log n)$.
- By Theorem 22.7, each range query in $S$ takes $O(\log n)$ time. We execute such a range query each time we process a new point. Thus, the total time spent for performing range queries is $O(n \log n)$.

We conclude that we can compute a closest pair in a set of $n$ points in time $O(n \log n)$.

## 22.5 Exercises

### Reinforcement

**R-22.1** Verify that the absolute value of the function $\Delta(p_1, p_2, p_3)$ is twice the area of the triangle formed by the points $p_1$, $p_2$, and $p_3$ in the plane.

**R-22.2** Provide a complete proof of Theorem 22.1.

**R-22.3** Provide a complete proof of Example 22.2.

**R-22.4** Provide a complete proof of Theorem 22.3.

**R-22.5** Provide a complete proof of Theorem 22.4.

**R-22.6** Provide a complete proof of Theorem 22.7.

**R-22.7** Give a pseudocode description of the plane-sweep algorithm for finding a closest pair of points among a set of $n$ points in the plane.

**R-22.8** Draw as best you can the convex hull of the following set of points:

$$\{(2,2),(4,4),(6,4),(8,1),(8,7),(9,3),(1,5),(5,4)\}.$$

### Creativity

**C-22.1** Using the orientation test, give a pseudocode description of a method, inTriangle$(p, q, r, s)$, which tests whether a point, $p$, is inside the interior of a triangle $(q, r, s)$, assuming $q$, $r$, and $s$ are listed in counterclockwise order.

**C-22.2** Using the inTriangle$(p, q, r, s)$ method from the previous exercise, and additional orientation tests, design a data structure that can be built for a given $n$-vertex convex polygon, $P$, to determine for any query point, $p$, whether $p$ lies inside $P$. Your data structure should use $O(n)$ space and answer point-in-polygon queries in $O(\log n)$ time.

***Hint:*** Use Figure 22.7c as a guide.

**C-22.3** Let $S$ be a set of $n$ distinct points in the plane. Say that a point $p = (x, y)$ in $S$ is ***max-max*** if there is no other point $q = (x', y')$ in $S$ such that $x < x'$ and $y < y'$. Similarly, say that a point $p = (x, y)$ in $S$ is ***max-min*** if there is no other point $q = (x', y')$ in $S$ such that $x < x'$ and $y' < y$, that $p = (x, y)$ in $S$ is ***min-max*** if there is no other point $q = (x', y')$ in $S$ such that $x' < x$ and $y < y'$, and that $p = (x, y)$ in $S$ is ***min-min*** if there is no other point $q = (x', y')$ in $S$ such that $x' < x$ and $y' < y$. Show that if a point, $p$, is on the convex hull of $S$, then it must also be a max-max, max-min, min-max, or min-min point in $S$.

**C-22.4** Show that the expected number of points on the convex hull of a set of $n$ points chosen uniformly and independently at random in the interior of a rectangle, $R$, is $O(\log n)$.

***Hint:*** Use the fact established in the previous exercise.

**C-22.5** Design an $O(n)$-time algorithm to test whether a given $n$-vertex polygon is convex. You should not assume that $P$ is simple.

**C-22.6** Suppose you are given an $n$-vertex convex polygon, $P$. Describe an $O(n)$-time method for computing the area of $P$.

**C-22.7** Say that a polygon, $P$, is ***monotone*** if any vertical line intersects the boundary of $P$ in at most two points. Given a simple monotone polygon, $P$, with $n$ vertices, show that you can find the convex hull of $P$ in $O(n)$ time.

**C-22.8** Let $C$ be a collection of $n$ horizontal and vertical line segments. Describe an $O(n \log n)$-time algorithm for determining whether the segments in $C$ form a simple polygon.

**C-22.9** Given a set $P$ of $n$ points, design an efficient algorithm for constructing a simple polygon whose vertices are the points of $P$.

**C-22.10** Design an $O(n^2)$-time algorithm for testing whether a polygon with $n$ vertices is simple. Assume that the polygon is given by the list of its vertices.

**C-22.11** Give an example of a set of input points for which the simplified Graham scan algorithm, given in Algorithm 22.11, does not work correctly.

**C-22.12** Let $P$ be a set of $n$ points in the plane. Modify the Graham scan algorithm to compute, for every point $p$ of $P$ that is not a vertex of the convex hull, either a triangle with vertices in $P$, or a segment with endpoints in $P$ that contains $p$.

**C-22.13** Suppose you are given a set $S$ of $n$ line segments in the plane, such that each makes a positive angle with the $x$-axis of either $30^\circ$ or $60^\circ$ (so there are only two possible slopes for the lines in $S$). Describe an efficient algorithm for finding all the pairs of intersecting segments in $S$. What is the running time of your method?

**C-22.14** Suppose we are given an array, $A$, of $n$ nonintersecting segments, $s_1, s_2, \ldots, s_n$, with endpoints on the lines $y = 0$ and $y = 1$, and ordered from left to right. Given a point $q$ with $0 < y(q) < 1$, design an algorithm that in $O(\log n)$ time computes the segment $s_i$ of $A$ immediately to right of $q$, or reports that $q$ is to the right of all the segments.

**C-22.15** Describe an $O(n \log n)$-time algorithm for finding the second closest pair of points in a set, $S$, of $n$ points in the plane. That is, you should return the pair, $(p, q)$, in $S$, such that the only pair of points that could be closer to each other than the distance between $p$ and $q$ is the closest pair of points in $S$.

## Applications

**A-22.1** In several computational geometry problems involving distances defined by a set, $S$, of $n$ points in the plane, it is often useful to first know what is the largest distance between a pair of points in $S$, which is known as the ***diameter*** of $S$. Another way to define the diameter of such a set, $S$, is as the largest distance between two parallel lines, $L_1$ and $L_2$, such that all the points of $S$ are either on one of these lines or between them. Suppose, then, that your are given such a set, $S$. Describe an $O(n \log n)$-time algorithm for computing the diameter of $S$.

**A-22.2** Gerrymandering is a process where voting districts are drawn to achieve various political goals, such as maximizing the number of voters from a certain party, rather than to achieve geometric goals, such as having districts drawn to have generally round or square shapes. This process often gives rise to very complicated shapes for voting districts, and it can sometimes be challenging to determine whether a giving person is inside or outside a given district, due to the ways it can wind around. Suppose, then, that you are given a voting district defined by an $n$-vertex simple polygon, $P$. Give an $O(n)$-time algorithm for testing whether a point $q$ is inside or outside of $P$. You may assume that $q$ is not on the boundary of $P$ and that there is no vertex of $P$ with the same $x$-coordinate as $q$.

**A-22.3** Line segments and polygons are used to model geometric objects in computer graphics, video games, and computer-aided design, often in data pipelines that use the output of one program as the input to another. One complication that can arise in such scenarios is that a data format needed for the input to the second system may be missing from the output from the first. For instance, one program may output an object as an unordered set of line segments but the other may require its input to be defined by a polygon. So, suppose you are given a set, $S$, of $n$ line segments, in no particular order. Describe an efficient algorithm for determining whether the segments of $S$ form a polygon, $P$, and if so, give a polygonal representation for $P$. You may allow $P$ to be non-simple (that is, self-intersecting), but $P$ must be a single cyclical chain of distinct vertices connected by the segments in $S$. What is the running time of your algorithm?

**A-22.4** In the ***hidden-line elimination*** problem, we would like to visualize a three-dimensional scene, described by a collection of polygons, from a particular viewing point, $p$, and in a particular direction. This problem is often solved by projecting the edges of the polygons in the scene onto a view plane perpendicular to the viewing direction. Then this set of segments is processed to remove the portions of edges that cannot be seen from $p$ because they are occluded by a polygon in the scene. In order to identify visible and invisible portions of edges, it is helpful to know which pairs of line segments intersect one another. So, suppose you are given a set, $S$, of $n$ line segments in the plane. Describe an algorithm to enumerate all $k$ pairs of intersecting segments in $S$ in $O((n+k)\log n)$ time.

***Hint:*** Use the plane-sweep technique, including segments intersections as events. Note that you cannot know these events in advance, but it is always possible to know the next event to process as you are sweeping.

**A-22.5** Computational ***metrology*** deals with algorithms for measuring things, such as manufactured parts, and one of the standard algorithms is to determine the flatness of an edge of a manufactured part based on a set of points that are sampled along that edge (say, by a laser range-finder or a scanning electron microscope). Since it is extremely rare for such a set of points to be perfectly collinear, we need a rigorous way of defining flatness in this context. One such way is to define the ***flatness*** of a set, $S$, of $n$ two-dimensional points as the smallest distance between two parallel lines, $L_1$ and $L_2$, such that all the points of $S$ are either on one of these lines or between them. This distance is known as the ***width*** of $S$. Describe an $O(n \log n)$-time algorithm for determining the width of such a set, $S$, of $n$ points in the plane.

**A-22.6** Suppose you are hiking in the wild country of some exotic part of the world. Naturally, an important part of your survival gear is a GPS tracking device and a digital map of the region you are hiking in, which includes the pathways of trails, rivers, and creeks, as well as obstacles like cliffs and mountains. These pathways and obstacles are described on your map by a collection, $S$, of $n$ two-dimensional line segments that may intersect only at their endpoints (hence, no pair of segments in $S$ cross). In order to locate your position on this map, however, you need to identify where your current position, $(x, y)$, lies in reference to the segments in $S$. Such a query can be satisfied by imagining that we shoot a vertical ray up or down from the point, $(x, y)$, to the first segment that it hits in $S$, which is known as a ***vertical ray-shooting*** query. In order to facilitate such a query, it is desirable for us to process the segments in $S$ to define a ***trapezoidal decomposition*** of the plane. Such a decomposition is defined by shooting vertical rays up and down from every endpoint of a segment in $S$ until it hits another segment in $S$ or it hits the boundary of the map. In the map defined by $S$ and these vertical rays, every face is either a trapezoid or a triangle. This partitioning allows us to then construct a data structure for answering vertical ray-shooting queries, since such a query can be answered simply by identifying the face in the trapezoidal decomposition that contains the starting point for the ray. Thus, given a set, $S$, of $n$ line segments that may intersect only at their endpoints, describe an $O(n \log n)$-time algorithm for constructing a trapezoidal decomposition of $S$.

**A-22.7** In machine learning applications, we often have some kind of condition defined over a set, $S$, of $n$ points, which we would like to characterize—that is, "learn"—using some simple rule. For instance, these points could correspond to biological attributes of $n$ medical patients and the condition could be whether a given patient tests positive for a given disease or not, and we would like to learn the correlation between these attributes and this disease. Think of the points with positive tests as painted "red," and the tests with negative tests as painted "blue." Suppose that we have a simple two-factor set of attributes; hence, we can view each patient as a two-dimensional point in the plane, which is colored as either red or blue. An ideal characterization, from a machine learning perspective, is if we can separate the points of $S$ by a line, $L$, such that all the points on one side of $L$ are red and all the points on the other side of $L$ are blue. So suppose you are given a set, $S$, of $n$ red and blue points in the plane. Describe an efficient algorithm for determining whether there is a line, $L$, that separates the red and blue points in $S$. What is the running time of your algorithm?

**A-22.8** Imagine a game, Battlestrip, which involves one player laying down a set of horizontal line segments on his computer screen, and another player laying down a set of vertical segments on hers. Then the two computer screens are virtually overlaid on top of one another and the winner is the player with a single segment that intersects the most segments of his or her opponent. Your job is to determine who is the winner. So, suppose you are given a set $S$ of $n$ horizontal and vertical line segments in the plane. Describe an algorithm running in $O(n \log n)$ time for finding the vertical segment, $v$, in $S$ that intersects the maximum number of horizontal segments in $S$, as well as the horizontal segment, $h$, in $S$ that intersects the maximum number of vertical segments in $S$. (Note that there may be many more than $O(n \log n)$ pairs of intersecting horizontal and vertical line segments.)

**A-22.9** Given a set $S$ of points in the plane, define the ***Voronoi diagram*** of $S$ to be the set of regions $V(p)$, called ***Voronoi cells***, defined, for each point $p$ in $S$, as the set of all points $q$ in the plane such that $p$ is a closest neighbor of $q$ in $S$. Such a diagram is useful in a host of different applications, including the construction of data structures to answer nearest-neighbor queries.

a. Show that each cell in a Voronoi diagram is convex.
b. Show that if $p$ and $q$ are a closest pair of points in the set $S$, then the Voronoi cells $V(p)$ and $V(q)$ touch.
c. Show that a point $p$ is on the boundary of the convex hull of the set $S$ if and only if the Voronoi cell $V(p)$ for $p$ is unbounded.

**A-22.10** Given a set $S$ of points in the plane, define the ***Delaunay triangulation*** of $S$ to be the set of all triangles $(p, q, r)$ such that $p$, $q$, and $r$ are in $S$ and the circle defined to have these points on its boundary is empty—it contains no points of $S$ in its interior. Such triangulations have many applications to modeling problems, as Delaunay triangulations tend to avoid "long and skinny" triangles, which are bad for modeling applications.

a. Show that if $p$ and $q$ are a closest pair of points in the set $S$, then $p$ and $q$ are joined by an edge in the Delaunay triangulation.
b. Show that the Voronoi cells $V(p)$ and $V(q)$, as defined in the previous exercise, share an edge in the Voronoi diagram of a point set $S$ if and only if $p$ and $q$ are joined by an edge in the Delaunay triangulation of $S$.

# Chapter Notes

The convex hull algorithm we present in this chapter is a variant of an algorithm given by Graham [91]. The plane-sweep algorithm we present for intersecting orthogonal line segments is due to Bentley and Ottmann [31]. The closest point algorithm we present combines ideas of Bentley [27] and Hinrichs *et al.* [98].

There are several excellent books for computational geometry, including books by Edelsbrunner [62], Mehlhorn [159], O'Rourke [165], Preparata and Shamos [173], Berg *et al.* [32], and handbooks edited by Goodman and O'Rourke [87], and Pach [166]. Other sources for further reading include surveys by Aurenhammer [17], Lee and Preparata [140], and book chapters by Goodrich [88], Lee [139], and Yao [218]. Also, the books by Sedgewick [188, 189] contain several chapters on computational geometry, which have some very nice figures. Indeed, the figures in Sedgewick's books have inspired many of the figures we present in this book.