

12.5 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 12.16.)

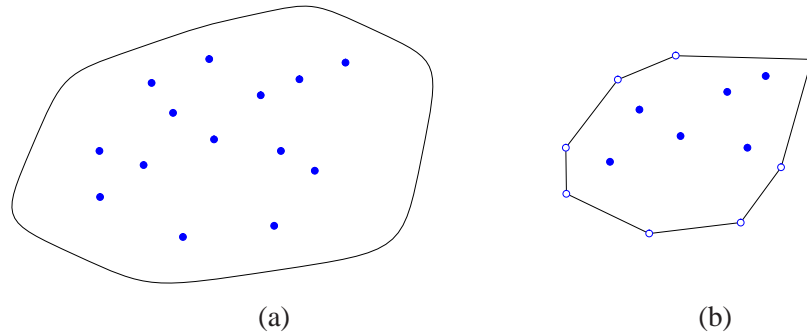


Figure 12.16: The convex hull of a set of points in the plane: (a) an example “rubber band” placed around the points; (b) the convex hull of the points.

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. Before we describe the convex hull and algorithms to compute it in detail, we need to first discuss some representational issues for geometric data objects.

12.5.1 Representations of 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 ADT’s for points, lines, segments, and polygons, which would be appropriate for a book on geometric algorithms, we instead assume we have intuitive representations for these objects. Even so, we briefly mention some of the choices that we can make regarding geometric representations.

We can represent a point in the plane by a pair (x, y) that stores the x and y Cartesian coordinates for that point. While this representation is quite versatile, it is not the only one. There may be some applications where a different representation may be better (such as representing a point as the intersection between two nonparallel lines).

Lines, Segments, and Polygons

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 , 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

$$a = (y_2 - y_1); \quad b = -(x_2 - x_1); \quad c = y_1(x_2 - x_1) - x_1(y_2 - y_1).$$

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?)

We can represent a polygon P by a circular sequence of points, called the *vertices* of P . (See Figure 12.17.) The segments between consecutive vertices of P are called the *edges* of P . Polygon P is said to be *nonintersecting*, or *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 π .

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.

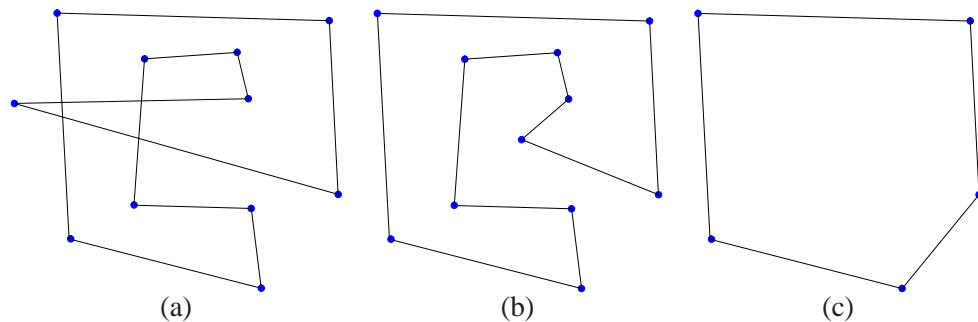


Figure 12.17: Examples of polygons: (a) intersecting, (b) simple, (c) convex.

12.5.2 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 π . If the angle on the right-hand side is less than π instead, then we say that (p, q, r) makes a **right turn** and is oriented **clockwise**. (See Figure 12.18.) It is possible that the angles of the left- and right-hand sides are both equal to π , 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

$$\Delta(p_1, p_2, p_3) = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 - x_2y_1 + x_3y_1 - x_1y_3 + x_2y_3 - x_3y_2. \quad (12.1)$$

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 12.8: *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 12.8; we leave the details as an exercise (R-12.4). In Figure 12.18, we show a triplet (p_1, p_2, p_3) of points such that $x_1 < x_2 < x_3$. 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:

$$\text{Is } \frac{y_3 - y_2}{x_3 - x_2} > \frac{y_2 - y_1}{x_2 - x_1} ? \quad (12.2)$$

By the expansion of $\Delta(p_1, p_2, p_3)$ shown in 12.1, we can verify that inequality 12.2 is equivalent to $\Delta(p_1, p_2, p_3) > 0$.

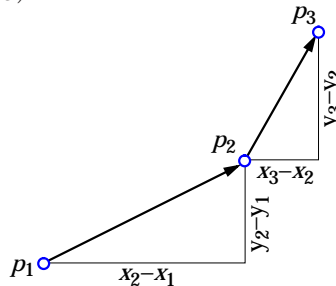


Figure 12.18: An example of a left turn. The differences between the coordinates between p_1 and p_2 and the coordinates of p_2 and p_3 are also illustrated.

Example 12.9: Using the notion of orientation, let us consider the problem of testing whether two 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 12.19. We also show, in Table 12.20, 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-12.5). Note that the conditions also hold if s_1 and/or s_2 is a degenerate segment with coincident endpoints.

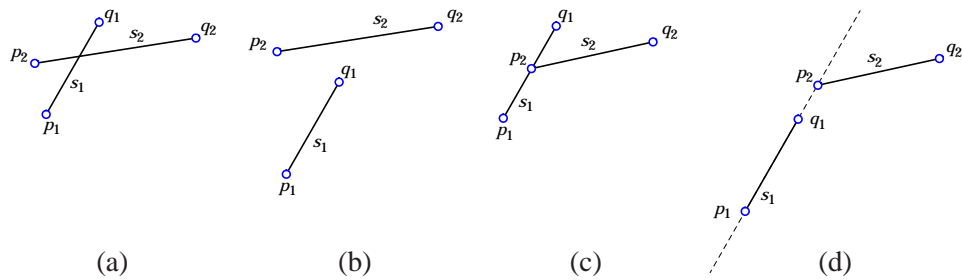


Figure 12.19: Examples illustrating four cases of Condition 1 of Example 12.9.

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 12.20: The four cases shown in Figure 12.19 for the orientations specified by Condition 1 of Example 12.9, where CCW stands for counterclockwise, CW stands for clockwise, and COLL stands for collinear.

12.5.3 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. The following example describes an application of the convex hull problem in a robot motion planning problem.

Example 12.10: *A common problem in robotics is to identify a trajectory from a start point s to a target point t that avoids a certain obstacle. Among the many possible trajectories, we would like to find one that is as short as possible. Let us assume that the obstacle is a polygon P . We can compute a shortest trajectory from s to t that avoids P with the following strategy (see Figure 12.21):*

- We determine if the line segment $\ell = \overline{st}$ intersects P . If it does not intersect, then ℓ is the shortest trajectory avoiding P .
- Otherwise, if \overline{st} intersects P , then we compute the convex hull H of the vertices of polygon P plus points s and t . Note that s and t subdivide the convex hull H into two polygonal chains, one going clockwise from s to t and one going counterclockwise from s to t .
- We select and return the shortest of the two polygonal chains with endpoints s and t on H .

This shortest chain is the shortest path in the plane that avoids the obstacle P .

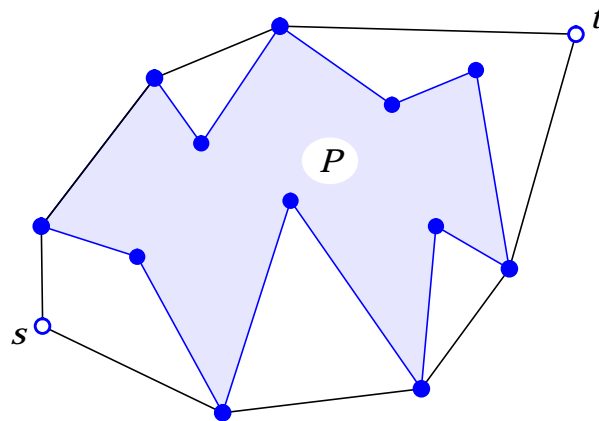


Figure 12.21: An example shortest trajectory from a point s to a point t that avoids a polygonal obstacle P ; the trajectory is a clockwise chain from s to t .

There are a number of applications of the convex hull problem, 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 12.11: *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 12.11 are illustrated in Figure 12.22. A complete proof of them is left as an exercise (R-12.6). As a consequence of Theorem 12.11, 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 x -coordinate
- A point with maximum x -coordinate
- A point with minimum y -coordinate
- A point with maximum y -coordinate.

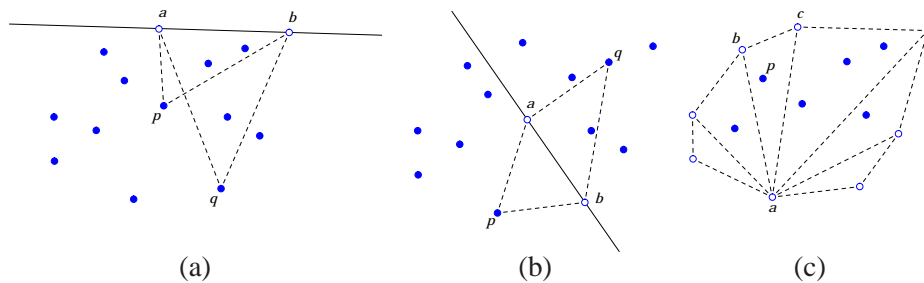


Figure 12.22: Illustration of the properties of the convex hull given in Theorem 12.11: (a) points a and b form an edge of the convex hull; (b) points a and b do not form an edge of the convex hull; (c) point p is not on the convex hull.

12.5.4 The Gift Wrapping Algorithm

Theorem 12.11 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 12.23):

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 corresponds to 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.

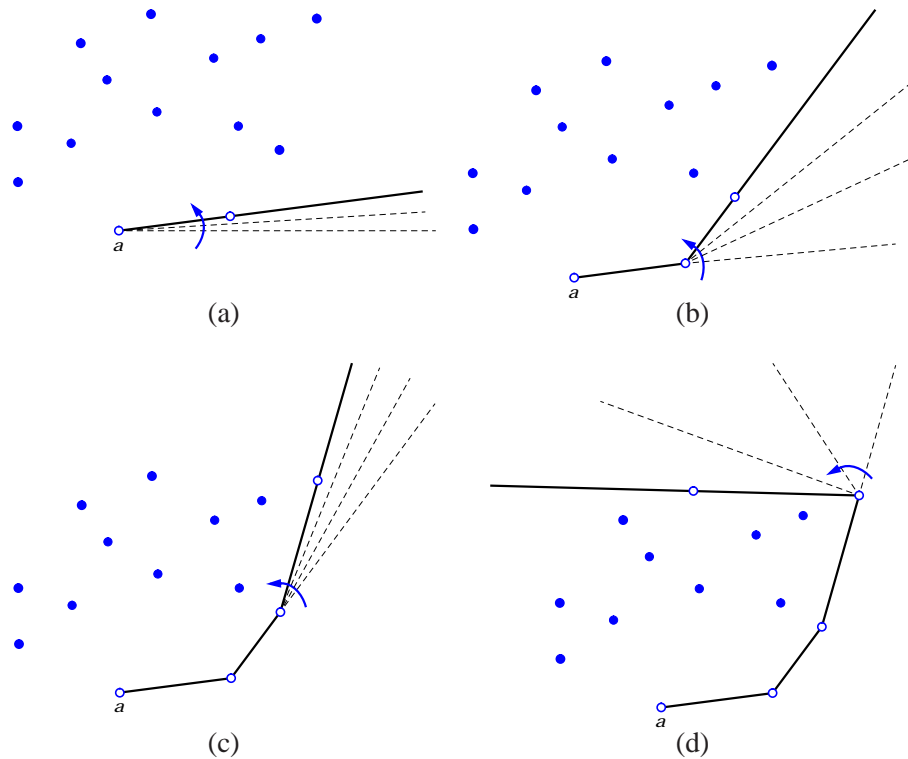


Figure 12.23: Initial four wrapping steps of the gift wrapping algorithm.

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 perform a wrapping step by means of the following theorem, which follows from Theorem 12.11.

Theorem 12.12: *Let S be a set of points in the plane, and let a be a point of S that is 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 .*

Recalling the discussion from Section 2.4.1, let us define a comparator $C(a)$ that uses the orientation of (a, p, q) to compare two points p and q of S . That is, $C(a).isLess(p, q)$ returns true if triplet (a, p, q) makes a left turn. We call the comparator $C(a)$ the **radial** comparator, as it compares points in terms of their radial relationships around the anchor point a . By Theorem 12.12, the vertex following a counterclockwise on the hull is simply the minimum point with respect to the radial comparator $C(a)$.

Performance

We can now 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, \dots, 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 radial comparator $C(p_{i-1})$, which 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 $C(p_{i-1})$. We conclude that the gift wrapping algorithm runs in time $O(hn)$, which is $O(n^2)$ in the worst case. Indeed, the worst case for the gift wrapping algorithm occurs when $h = n$, that is, when all the points are on the convex hull.

The worst-case running time of the gift wrapping algorithm in terms of n is therefore not very efficient. 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 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. In the next section, we will see an algorithm that is efficient for all hull sizes, although it is slightly more complicated.

12.5.5 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, $P - \{a\}$) using the radial comparator $C(a)$, and let S be the resulting sorted list of points. (See Figure 12.24.) 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 by the comparator.

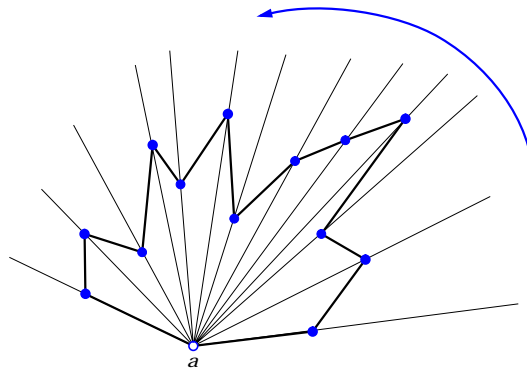


Figure 12.24: Sorting around the anchor point in the Graham scan algorithm.

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 12.25. (See Figure 12.26.)

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 12.25: The scan phase of the Graham scan convex hull algorithm. (See Figure 12.26.) 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.

Performance

Let us now analyze the running time of the Graham scan algorithm. 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 2.4.4) or merge-sort (Section 4.1). 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 12.25. 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.

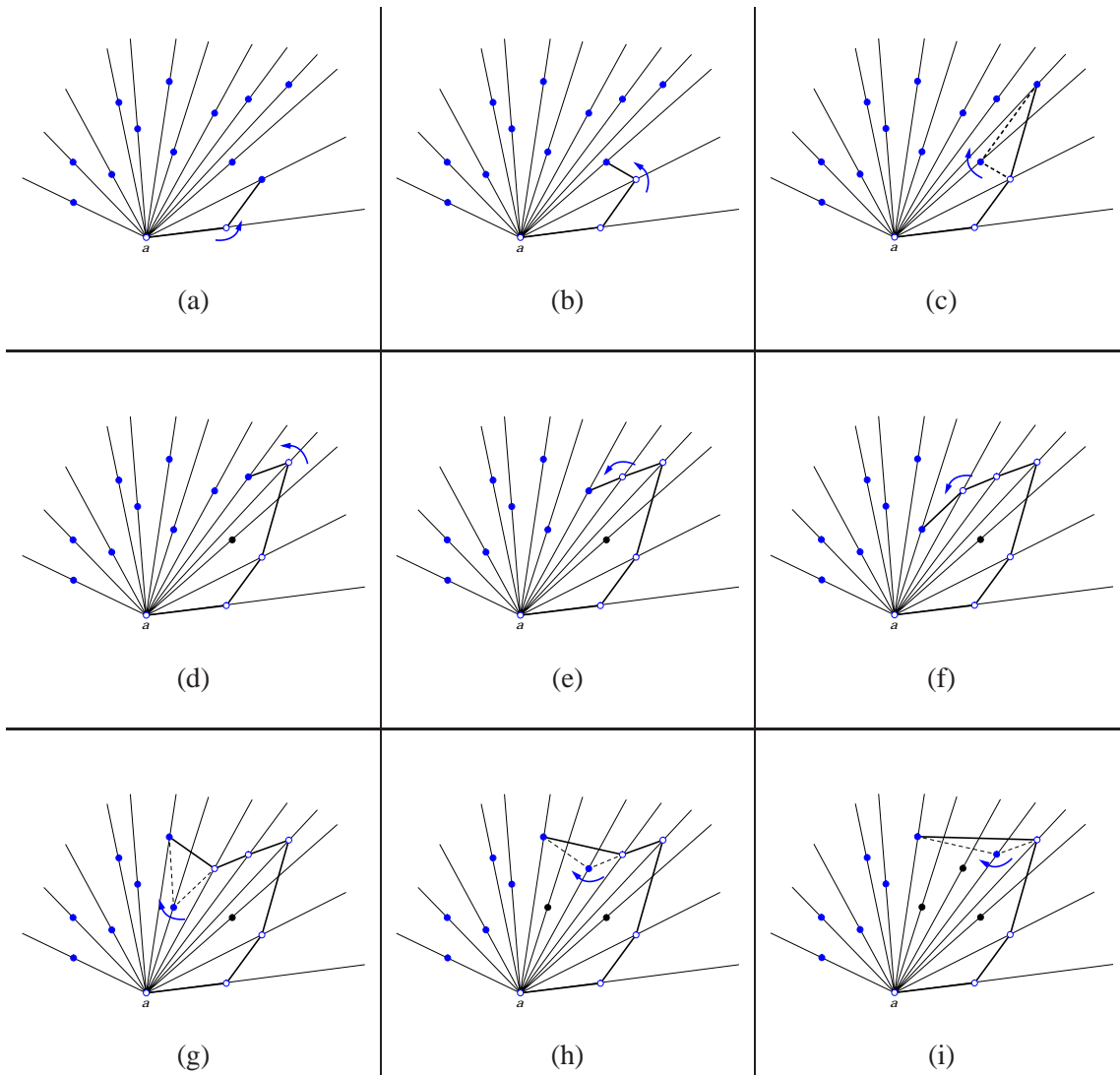


Figure 12.26: Third phase of the Graham scan algorithm (see Algorithm 12.25).

12.6 Java Example: Convex Hull

We described the Graham scan algorithm (Algorithm 12.25) assuming it avoided any *degeneracies*, that is, input configurations that involve annoying special cases (such as coincident or collinear points). When implementing the Graham scan algorithm, however, it is important to handle all possible input configurations. For example, two or more of the points may be coincident, and some triplets of points may be collinear.

In Code Fragments 12.27–12.29, we show a Java implementation of the Graham scan algorithm. The main method is `grahamScan` (Code Fragment 12.27), which uses several auxiliary methods. Because of the degenerate point configurations that may occur, several special situations are handled by method `grahamScan`.

- The input list is first copied into sequence `hull`, which will be returned at the end of the execution (method `copyInputPoints` of Code Fragment 12.28).
- If the input has zero or one point (the output is the same as the input) return.
- If there are two input points, then if the two points are coincident, remove one of them and return.
- The anchor point is computed and removed, together with all the points coincident with it (method `anchorPointSearchAndRemove` of Code Fragment 12.28). If zero or one point is left, reinsert the anchor point and return.
- If none of the above special cases arises, that is, at least two points are left, sort the points counterclockwise around the anchor point with method `sortPoints` (Code Fragment 12.28), which passes a `ConvexHullComparator` (a comparator that allows for a generic sorting algorithm sorting algorithm to sort points counterclockwise radially around a point, as needed in Algorithm 12.25).
- In preparation for the Graham scan, we remove any initial collinear points in the sorted list, except the farthest one from the anchor point (method `removeInitialIntermediatePoints` of Code Fragment 12.29).
- The scan phase of the algorithm is performed calling method `scan` of Code Fragment 12.29.

In general, when we implement computational geometry algorithms we must take special care to handle all possible “degenerate” cases.

```

public class ConvexHull {
    private static Sequence hull;
    private static Point2D anchorPoint;
    private static GeomTester2D geomTester = new GeomTester2DImpl();
    // public class method
    public static Sequence grahamScan (Sequence points) {
        Point2D p1, p2;
        copyInputPoints(points); // copy into hull the sequence of input points
        switch (hull.size()) {
            case 0: case 1:
                return hull;
            case 2:
                p1 = (Point2D)hull.first().element();
                p2 = (Point2D)hull.last().element();
                if (geomTester.areEqual(p1,p2))
                    hull.remove(hull.last());
                return hull;
            default: // at least 3 input points
                // compute anchor point and remove it together with coincident points
                anchorPointSearchAndRemove();
                switch (hull.size()) {
                    case 0: case 1:
                        hull.insertFirst(anchorPoint);
                        return hull;
                    default: // at least 2 input points left besides the anchor point
                        sortPoints();// sort the points in hull around the anchor point
                        // remove the (possible) initial collinear points in hull except the
                        // farthest one from the anchor point
                        removeInitialIntermediatePoints();
                        if (hull.size() == 1)
                            hull.insertFirst(anchorPoint);
                        else { // insert the anchor point as first and last element in hull
                            hull.insertFirst(anchorPoint);
                            hull.insertLast(anchorPoint);
                            scan(); // Graham's scan
                            // remove one of the two copies of the anchor point from hull
                            hull.remove(hull.last());
                        }
                }
                return hull;
        }
    }
}

```

Code Fragment 12.27: Method `grahamScan` in the Java implementation of the Graham scan algorithm.

```

private static void copyInputPoints (Sequence points) {
    // copy into hull the sequence of input points
    hull = new NodeSequence();
    Enumeration pe = points.elements();
    while (pe.hasMoreElements()) {
        Point2D p = (Point2D)pe.nextElement();
        hull.insertLast(p);
    }
}

private static void anchorPointSearchAndRemove () {
    // compute the anchor point and remove it from hull together with
    // all the coincident points
    Enumeration pe = hull.positions();
    Position anchor = (Position)pe.nextElement();
    anchorPoint = (Point2D)anchor.element();
    // hull contains at least three elements
    while (pe.hasMoreElements()) {
        Position pos = (Position)pe.nextElement();
        Point2D p = (Point2D)pos.element();
        int aboveBelow = geomTester.aboveBelow(anchorPoint,p);
        int leftRight = geomTester.leftRight(anchorPoint,p);
        if (aboveBelow == GeomTester2D.BELOW ||
            aboveBelow == GeomTester2D.ON &&
            leftRight == GeomTester2D.LEFT) {
            anchor = pos;
            anchorPoint = p;
        }
        else
            if (aboveBelow == GeomTester2D.ON &&
                leftRight == GeomTester2D.ON)
                hull.remove(pos);
    }
    hull.remove(anchor);
}

private static void sortPoints() {
    // sort the points in hull around the anchor point
    SortObject sorter = new ListMergeSort();
    ConvexHullComparator comp = new ConvexHullComparator(anchorPoint,
                                                         geomTester);

    sorter.sort(hull,comp);
}

```

Code Fragment 12.28: Auxiliary methods `copyInputPoints`, `anchorPointSearchAndRemove`, and `sortPoints` called by method `grahamScan` of Code Fragment 12.27.

```

private static void removeInitialIntermediatePoints() {
    // remove the (possible) initial collinear points in hull except the
    // farthest one from the anchor point
    boolean collinear = true;
    while (hull.size() > 1 && collinear) {
        Position pos1 = hull.first();
        Position pos2 = hull.after(pos1);
        Point2D p1 = (Point2D)pos1.element();
        Point2D p2 = (Point2D)pos2.element();
        if (geomTester.leftRightTurn(anchorPoint,p1,p2) ==
            GeomTester2D.COLLINEAR)
            if (geomTester.closest(anchorPoint,p1,p2) == p1)
                hull.remove(pos1);
            else
                hull.remove(pos2);
        else
            collinear = false;
    }
}

private static void scan() {
    // Graham's scan
    Position first = hull.first();
    Position last = hull.last();
    Position prev = hull.first();
    Position curr = hull.after(prev);
    do {
        Position next = hull.after(curr);
        Point2D prevPoint = (Point2D)prev.element();
        Point2D currPoint = (Point2D)curr.element();
        Point2D nextPoint = (Point2D)next.element();
        if (geomTester.leftRightTurn(prevPoint,currPoint,nextPoint) ==
            GeomTester2D.LEFT_TURN)
            prev = curr;
        else {
            hull.remove(curr);
            prev = hull.before(prev);
        }
        curr = hull.after(prev);
    }
    while (curr != last);
}
}

```

Code Fragment 12.29: Auxiliary methods `removeInitialIntermediatePoints` and `scan` called by method `grahamScan` of Code Fragment 12.27.