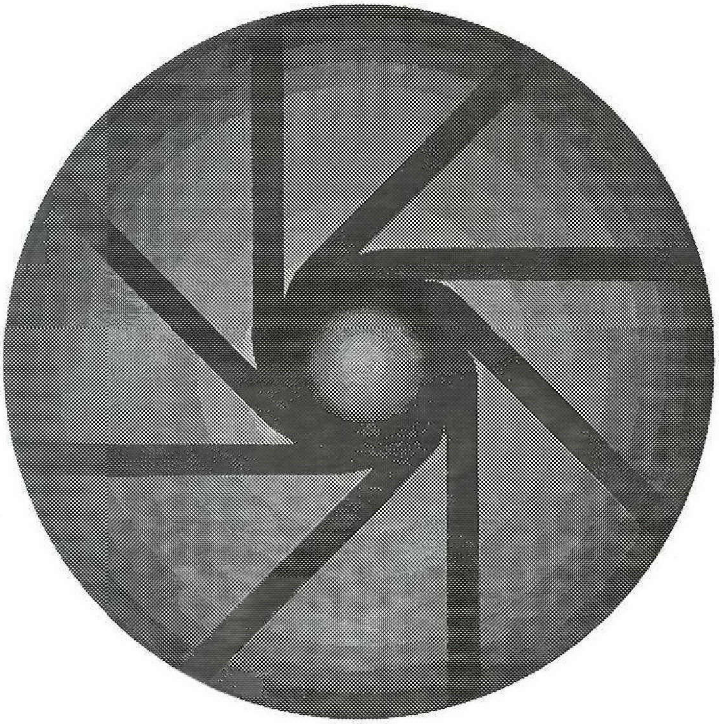
COMPUTATIONAL GEOMETRY AND COMPUTER GRAPHICS IN C++



MICHAEL J. LASZLO

4

Geometric Data Structures

In this chapter we define the classes we will need for working with geometric objects in two and three dimensions. In two dimensions, the operations supported by these classes include splitting a polygon along a chord into two smaller polygons, computing the intersection point of two skew lines, and classifying a point relative to a line. In three dimensions they include classifying a point relative to a plane and finding the intersection of a line and a triangle. This chapter also provides what little linear algebra we will need.

4.1 Vectors

A *coordinate system* provides a frame of reference for specifying positions in the plane. Under the *Cartesian coordinate system*, the plane is endowed with two *coordinate axes* with the same origin (their point of intersection) and same unit length; the axes are perpendicular to each other and oriented as in Figure 4.1a. This establishes a one-to-one correspondence between ordered pairs of numbers (x, y) and points in the plane. The point's first coordinate x indicates its displacement along the horizontal axis, and the point's second coordinate y its displacement along the vertical axis.

An ordered pair (x, y) can also be thought of as a *vector*, as shown in Figure 4.1b. Geometrically, vector (x, y) is a directed line segment beginning at the origin $(0, 0)$ and ending at point (x, y). The origin $(0, 0)$—sometimes denoted $\mathbf{0}$—is called the *zero vector*.

Vector addition and *scalar multiplication* are two fundamental operations for working with vectors (Figure 4.2). Given two vectors $a = (x_a, y_a)$ and $b = (x_b, y_b)$, vector addition

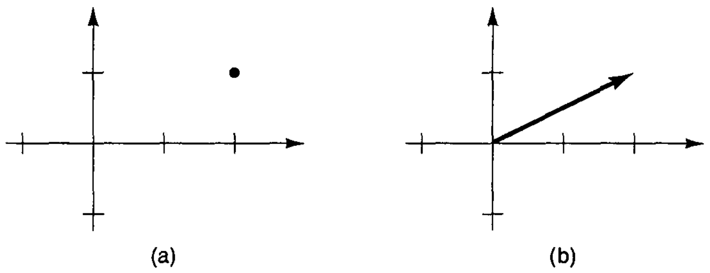


Figure 4.1: Interpreting the ordered pair (2, 1) as (a) a point and (b) a vector.

is defined by $a + b = (x_a + x_b, y_a + y_b)$. Geometrically, vectors a and b determine the parallelogram with vertices $\mathbf{0}$, a, b, and $a + b$.

Scalar multiplication involves the multiplication of a vector by a real number, the *scalar* (Figure 4.2). Given scalar t and vector $b = (x_b, y_b)$, scalar multiplication is defined by $tb = (tx_b, ty_b)$. The operation scales the length of vector b by factor t. The direction of the vector is unchanged if $t > 0$ and reversed if $t < 0$.

Since a vector begins at the origin, it is fully described by the point at which it terminates. Alternatively, a vector can be characterized by its *length* and *direction*. The length of vector $a = (x_a, y_a)$, denoted $\|a\|$, is defined by $\|a\| = \sqrt{x_a^2 + y_a^2}$. This equals the distance between point a and the origin $\mathbf{0}$. A *unit vector* is a vector with length one. Scaling a nonzero vector a by the reciprocal of its length yields a unit vector $\frac{a}{\|a\|}$ with the same direction, an operation known as *normalization*.

The direction of vector a is described by its *polar angle* θ_a, the angle the vector makes with the positive x-axis. Polar angles are measured in counterclockwise rotation starting at the positive x-axis and lie in the range $0 \le \theta_a < 360$ (we will always measure angles in degrees). Figure 4.3 gives some examples.

Vector subtraction is defined in terms of vector addition and scalar multiplication: Given vectors a and b, we have $b - a = b + (-1)a$. In practice, the operation is carried out with coordinate-wise subtraction: $b - a = (x_b - x_a, y_b - y_a)$. Geometrically, the operation identifies the directed line segment $\overrightarrow{ab}$, beginning at point a and ending at point b, with the vector $b - a$ (Figure 4.4).

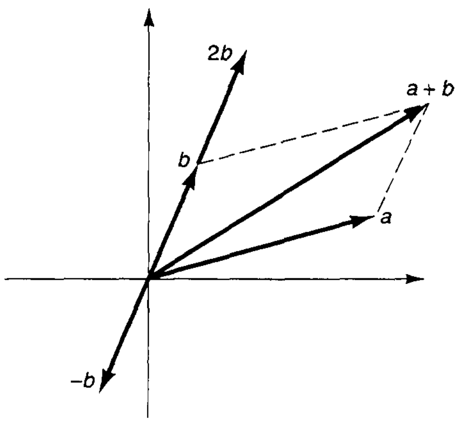


Figure 4.2: Vector addition and scalar multiplication.

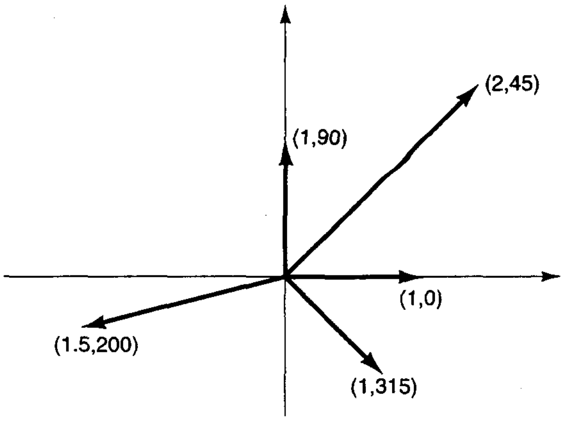


Figure 4.3: Various vectors, given in polar coordinates $a = (\|a\|, \theta_a)$.

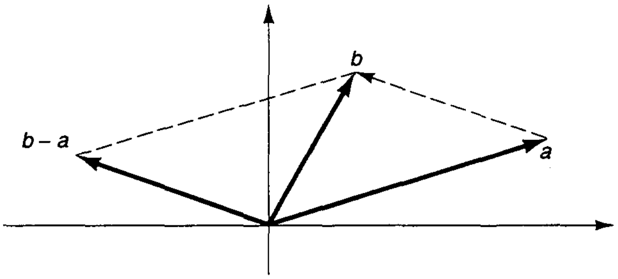


Figure 4.4: Vector subtraction.

A *directed line segment* $\overrightarrow{ab}$ is a vector fixed in the plane. Endpoint a is called the *origin* of $\overrightarrow{ab}$, and endpoint b the *destination*. Two directed line segments $\overrightarrow{ab}$ and $\overrightarrow{cd}$ that have the same length and direction are translates of each other and can be identified with the same canonical directed line segment, the vector $b - a = d - c$. Vector arithmetic provides the machinery for solving problems involving directed line segments that remain unchanged by translation. We illustrate this fact with the following example.

Given three non-collinear points p_0, p_1, p_2, the triangle $\Delta p_0 p_1 p_2$ they determine is *positively oriented* if p_2 lies to the left of $\overrightarrow{p_0 p_1}$, and *negatively oriented* if p_2 lies to the right of $\overrightarrow{p_0 p_1}$ (Figure 4.5). The problem is to describe a procedure for deciding orientation. It is reasonable to solve this problem using vectors since the orientation of a triangle does not change under translation. Letting $a = p_1 - p_0$ and $b = p_2 - p_0$, the problem reduces to one involving the angle θ_{ab} between the vectors, measured counterclockwise starting at vector a. If $0 < \theta_{ab} < 180$, then $\Delta p_0 p_1 p_2$ has positive orientation; otherwise $(180 < \theta_{ab} < 360)$ the triangle has negative orientation.

Vectors a and b assume one of four possible configurations (Figure 4.6). In cases 1 and 3 we have $0 < \theta_{ab} < 180$, and in cases 2 and 4 we have $180 < \theta_{ab} < 360$; in cases 1 and 2 the positive x-axis pierces the angle θ_{ab}, and in cases 3 and 4 it does not. The four

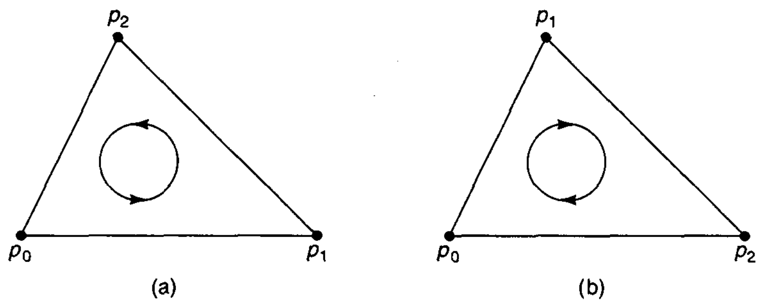


Figure 4.5: Triangle is (a) positively oriented and (b) negatively oriented.

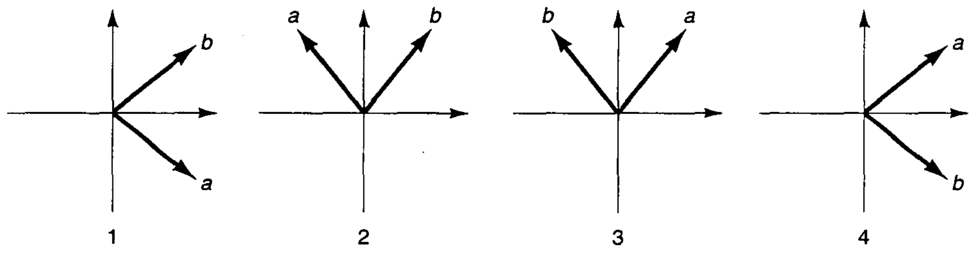


Figure 4.6: The four configurations relevant for deciding a triangle's orientation.

possible configurations correspond to four possible ranges in which the value $Q = \theta_b - \theta_a$ lies:

Case	Range of $Q = \theta_b - \theta_a$	Orientation of $\Delta p_0 p_1 p_2$	sin Q
1	$-360 < Q < -180$	+	+
2	$-180 < Q < 0$	−	−
3	$0 < Q < 180$	+	+
4	$180 < Q < 360$	−	−

To decide the orientation of the triangle, we could compute $Q = \theta_b - \theta_a$ and then answer based on which of the four ranges Q lies in. A better way makes use of the observation that $\sin(Q)$ has the same sign as the triangle's orientation. Since

$$\sin(\theta_b - \theta_a) = \sin\theta_b \, \cos\theta_a - \cos\theta_b \, \sin\theta_a$$

and

$$\cos\theta_a = \frac{x_a}{\|a\|}, \quad \sin\theta_a = \frac{y_a}{\|a\|}, \quad \cos\theta_b = \frac{x_b}{\|b\|}, \quad \sin\theta_b = \frac{y_b}{\|b\|}$$

we have

$$\sin(\theta_b - \theta_a) = \frac{1}{\|a\| \, \|b\|}(x_a y_b - x_b y_a)$$

Because the lengths $\|a\|$ and $\|b\|$ are positive constants, it follows that

$$\text{sign}(\sin(\theta_b - \theta_a)) = \text{sign}(x_a y_b - x_b y_a)$$

Hence $x_a y_b - x_b y_a$ has the same sign as the triangle's orientation. In the next section we will formulate this as a C++ function which reports the orientation of a triangle. It is noteworthy that the expression $x_a y_b - x_b y_a$ has a simple geometric interpretation: It equals the signed area of the parallelogram with vertices $\mathbf{0}$, a, b, and $a + b$.

4.2 Points

4.2.1 The `Point` Class

The class `Point` contains data members `x` and `y` to store a point's coordinates. Its member functions support such operations as classifying this point relative to a given line segment and computing the point's distance from a given line. Additional member functions treat this point as a vector: operator functions for performing vector arithmetic, and functions which return polar angle and length.

```
class Point {
 public:
   double x;
   double y;
   Point(double _x = 0.0, double _y = 0.0);
   Point operator+(Point&);
   Point operator-(Point&);
   friend Point operator*(double, Point&);
   double operator[](int);
   int operator==(Point&);
   int operator!=(Point&);
   int operator<(Point&);
   int operator>(Point&);
   int classify(Point&, Point&);
   int classify(Edge&);
   double polarAngle(void);
   double length(void);
   double distance(Edge&);
};
```

4.2.2 Constructors

The constructor initializes a new point with x and y coordinates:

```
Point::Point(double _x, double _y) :
   x(_x), y(_y)
{
}
```

If arguments are not provided, default arguments initialize the point to (0, 0).

A point can also be initialized with a second point. For example, the declaration `Point p(q)` initializes a new point `p` with the same coordinates as point `q`. In this case, initialization is performed by the default copy constructor (supplied by the C++ compiler), which performs a member-wise copy.

4.2.3 Vector Arithmetic

Vector addition and vector subtraction are invoked by the operators + and -:

```
Point Point::operator+(Point &p)
{
   return Point(x + p.x, y + p.y);
}

Point Point::operator-(Point &p)
{
   return Point(x - p.x, y - p.y);
}
```

The scalar multiplication operator is made a friend of class `Point`, rather than a member of the class, because its first operand is not of type `Point`. The operator is defined as follows:

```
Point operator*(double s, Point &p)
{
   return Point(s * p.x, s * p.y);
}
```

The `operator[]` member returns this point's x-coordinate if called with *coordinate index* 0, or its y-coordinate if called with 1:

```
double Point::operator[](int i)
{
   return (i == 0) ? x : y;
}
```

4.2.4 Relational Operators

The relational operators == and != are used to determine whether two points are equivalent:

```
int Point::operator==(Point &p)
{
   return (x == p.x) && (y == p.y);
}

int Point::operator!=(Point &p)
{
   return !(*this == p);
}
```

Operators < and > implement the *lexicographic* order relation in which point a is less than point b if either (1) $a.x < b.x$ or (2) $a.x = b.x$ and $a.y < b.y$. Given two points, we first compare their x-coordinates; if their x-coordinates are equal, we then compare their y-coordinates. This is sometimes called the *dictionary order relation* because the same rule orders two-letter words in a dictionary.

```
int Point::operator<(Point &p)
{
   return ((x < p.x) || ((x == p.x) && (y < p.y)));
}

int Point::operator>(Point &p)
{
   return ((x > p.x) || ((x == p.x) && (y > p.y)));
}
```

Infinitely many other orderings of the points in the plane are possible. Nonetheless, it is convenient to use operators < and > to establish a canonical ordering since we will often be storing points in dictionaries, and these operators can be used to help define the necessary comparison functions.

Before turning to the remaining member functions of class `Point`, let us consider the following simple example, which illustrates the use of `Point` objects. The function `orientation` returns 1 if the three points it is handed are positively oriented, -1 if they are negatively oriented, or 0 if they are collinear. The function implements the method explained at the end of the previous section.

```
int orientation(Point &p0, Point &p1, Point &p2)
{
   Point a = p1 - p0;
   Point b = p2 - p0;
   double sa = a.x * b.y - b.x * a.y;
   if (sa > 0.0)
      return 1;
   if (sa < 0.0)
      return -1;
   return 0;
}
```

4.2.5 Point-Line Classification

One important operation is that of classifying a point relative to a directed line segment. The operation reports whether the point lies to the left or right of the directed line segment; and if neither, whether the point lies beyond the directed line segment's destination or behind its origin; and if neither of these, whether it coincides with the origin, coincides with the destination, or lies between them. The directed line segment effectively partitions the plane into seven non-overlapping regions, and the operation reports in which region the point lies (Figure 4.7).

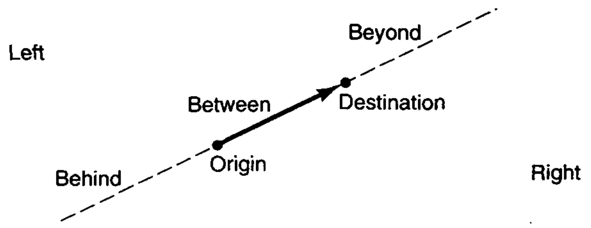


Figure 4.7: Partition of the plane into seven regions by a directed line segment.

Member function `classify` is used to classify this point relative to the directed line segment $\overrightarrow{p_0 p_1}$ from `p0` to `p1`. It returns an enumeration value indicating the point's classification:

```
enum { LEFT, RIGHT, BEYOND, BEHIND, BETWEEN, ORIGIN, DESTINATION };

int Point::classify(Point &p0, Point &p1)
{
   Point p2 = *this;
   Point a = p1 - p0;
   Point b = p2 - p0;
   double sa = a.x * b.y - b.x * a.y;
   if (sa > 0.0)
      return LEFT;
   if (sa < 0.0)
      return RIGHT;
   if ((a.x * b.x < 0.0) || (a.y * b.y < 0.0))
      return BEHIND;
   if (a.length() < b.length())
      return BEYOND;
   if (p0 == p2)
      return ORIGIN;
   if (p1 == p2)
      return DESTINATION ;
   return BETWEEN;
}
```

The orientation of points `p0`, `p1`, and `p2` is first used to decide whether point `p2` lies to the left of, to the right of, or collinear with $\overrightarrow{p_0 p_1}$. In the last case, additional calculations are needed. If vectors `a=p1-p0` and `b=p2-p0` point in opposite directions, then point `p2` lies behind directed segment $\overrightarrow{p_0 p_1}$. If vector `a` is shorter than vector `b`, then `p2` lies beyond $\overrightarrow{p_0 p_1}$. Otherwise `p2` is compared to `p0` and `p1` to decide whether it coincides with one of these two endpoints or lies between them.

A second version of member function `classify`, which is passed an edge rather than a pair of points, is provided for convenience:

```
int Point::classify(Edge &e)
{
   return classify(e.org, e.dest);
}
```

Point-line classification will be used frequently throughout this book. In some applications a more coarse classification suffices (such as deciding whether a point lies to the left of a given directed line segment). Other applications will make full use of this classification scheme.

4.2.6 Polar Coordinates

The *polar coordinate system* provides a second frame of reference for fixing positions in the plane. Originating from the origin **0** is a *polar axis*, a rightward-pointing horizontal ray as in Figure 4.8. A point a is represented by the pair (r_a, θ_a). Regarding point a as a vector originating at the origin, r_a is its length and θ_a its polar angle (the angle that a makes with the polar axis, measured in counterclockwise rotation).

The correspondence between pairs (r_a, θ_a) and points is not one to one; many pairs can represent the same point. The pair $(0, \theta)$ corresponds to the origin for *every* value of θ. Moreover, $(r, \theta + 360k)$ corresponds to the same point as k ranges over the integers.

Points can be represented in Cartesian coordinates or in polar coordinates, and it is sometimes necessary to switch from one coordinate system to the other. As evident in Figure 4.8, the two equations

$$x = r\cos\theta, \qquad y = r\sin\theta$$

transform a point from polar coordinates (r, θ) into Cartesian coordinates (x, y).

To transform back, the distance coordinate r is given by

$$r = \sqrt{x^2 + y^2}$$

To express polar angle θ as a function of x and y, observe that the relation $\tan\theta = \frac{y}{x}$ holds, from which it follows that

$$\theta = \tan^{-1}\frac{y}{x}, \quad x \neq 0 \qquad [4.1]$$

To use Equation 4.1 in function `polarAngle`, it is necessary to distinguish between the quadrants of the plane and to handle the case in which x equals zero:

```
double Point::polarAngle(void)
{
   if ((x == 0.0) && (y == 0.0))
      return -1.0;
   if (x == 0.0)
      return ((y > 0.0) ? 90 : 270);
```

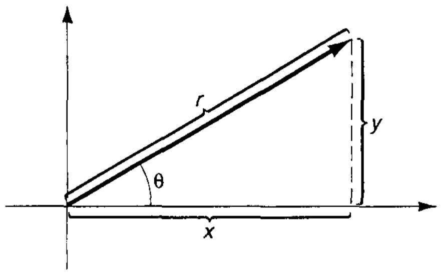


Figure 4.8: Point p is described by polar coordinates (r, θ) and Cartesian coordinates (x, y).

```
   double theta = atan(y / x);        // in radians
   theta *= 360 / (2 * 3.1415926);    // convert to degrees
   if (x > 0.0)    // quadrants 1 and 4
      return ((y >= 0.0) ? theta : 360 + theta);
   else    // quadrants 2 and 3
      return (180 + theta);
}
```

Note that function `polarAngle` returns -1.0 if this vector is the zero vector (it returns a nonnegative value otherwise). This will be used later to simplify the definition of comparison functions based on polar angle.

Member function `length` returns the length of this vector:

```
double Point::length(void)
{
   return sqrt(x*x + y*y);
}
```

Member function `distance` returns the signed distance from this point to an edge. We will define the function in subsection 4.5.3.

4.3 Polygons

Polygons are fascinating—surprisingly so, given how simple they are in concept. In this section we present basic definitions and concepts for talking about polygons and tools for handling them.

4.3.1 What Are Polygons?

A polygon is a closed curve in the plane composed of straight line segments. The segments are called the *edges* or *sides* of the polygon, and the endpoints where two segments meet are called its *vertices*. The number of vertices (or, equivalently, sides) that a polygon possesses is its *size*. For brevity, we will often use *n-gon* to mean a polygon of size n, and $|P|$ to denote the size of some polygon P.

A polygon is *simple* if it does not cross itself. A simple polygon encloses a connected region of the plane, referred to as its *interior*. The unbounded region surrounding a simple polygon forms its *exterior*, and the set of points lying on the polygon itself forms its *boundary*. In this book we will take *polygon* to mean *simple filled polygon*: the union of the boundary and interior of a simple polygon. To say, for instance, that a point lies in a polygon means that the point belongs either to the (simple) polygon's boundary or interior.

Vertices are ordered cyclically around a polygon boundary. Two vertices that are the endpoints of a common edge are *neighbors* and are said to be *adjacent* to one another. A vertex's clockwise neighbor is called its *successor*, and its counterclockwise neighbor its *predecessor*. A *vertex chain*, or simply *chain*, is a section of a polygon boundary. Polygon *traversal* involves moving along a chain from vertex to adjacent vertex, in either clockwise or counterclockwise rotation. Traversal often proceeds full circle around the entire polygon boundary, such as when it is necessary to visit every vertex.

The vertices of a polygon are classified as convex or reflex. A vertex is *convex* if the interior angle at the vertex—through the polygon interior—measures less than or equal to 180 degrees. A vertex is *reflex* otherwise (its interior angle measures greater than 180 degrees).

A line segment between any two nonadjacent vertices is called a *diagonal*. A diagonal is called a *chord* or *internal diagonal* if it lies in the polygon, not crossing the polygon's exterior. Adding a chord to a polygon splits it into smaller subpolygons. Figure 4.9 illustrates some of the notions we have covered relating to polygons.

It is sometimes convenient to regard a point or a line segment as a *degenerate polygon*. A 1-gon consists of a single vertex and a single zero-length edge that connects the vertex to itself. A 2-gon consists of two vertices and two coincident edges that connect the two vertices. Among other benefits, the use of degenerate polygons often simplifies polygon construction: Starting with a 1-gon, we insert a second vertex to form a 2-gon, followed by additional vertices to form conventional polygons of size 3 or greater. By regarding points and line segments as polygons, the initial stages of the process are no different in kind from later stages: Every stage involves the manipulation of polygons.

4.3.2 Convex Polygons

A region in the plane is *convex* if for any two points in the region, the line segment between the two points lies in the region. In Figure 4.9, polygon (a) is convex whereas polygon (b) is not (since the line segment $\overline{pq}$ leaves the polygon). Note that the *boundary* of a convex polygon is *not* convex, but the interior of a convex polygon is.

Convexity has a number of properties that make convex polygons easier to work with than arbitrary polygons. For example, every diagonal of a convex polygon is a chord. In addition, every vertex of a convex polygon is convex. (In a nonconvex polygon, at least one vertex is reflex.) From this it follows that a clockwise traversal of a convex polygon either continues straight or turns right at every vertex.

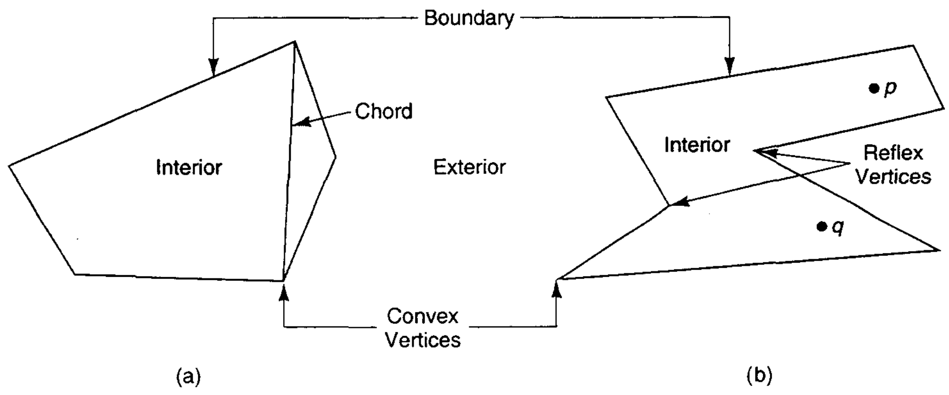


Figure 4.9: Basic concepts involving polygons.

Another property is that the intersection $A \cap B$ of any two convex regions A and B is convex. (To see why, suppose that p and q are any two points in $A \cap B$. Since both p and q lie in A and A is convex, line segment $\overline{pq}$ lies in A. Similarly, $\overline{pq}$ lies in B. Hence $\overline{pq}$ lies in $A \cap B$, so $A \cap B$ must be convex.) It follows that the intersection of two convex polygons is convex—in fact, a (possibly degenerate) convex polygon. Moreover, since a line is convex, the intersection of a line and a convex polygon must be convex: a line segment, or a single point if the line merely grazes the polygon at some vertex. For these and other properties, we will often work with convex polygons in this book.

4.3.3 The `Vertex` Class

We will represent a polygon by its cycle of vertices, stored in a circular doubly linked list. Each node corresponds to a vertex and links to its two neighbors. By following links we can traverse the polygon boundary in either sense of rotation, and by inserting and removing nodes—and updating links generally—we can create and dynamically modify the polygon.

The classes `Vertex` and `Polygon` support this scheme. The polygon is stored in a circular doubly linked list of `Vertex` objects. Since a vertex of a polygon behaves both like a point in the plane and like a node in a linked list, class `Vertex` is derived from both class `Point` and class `Node`. The `Polygon` class contains a data member which points to some vertex of the linked list representing the polygon. Class `Polygon` serves as the public interface for polygons.

Class `Vertex` inherits data members `_next` and `_prev` from base class `Node`, and `x` and `y` from base class `Point`. By convention, `_next` points to this vertex's successor (its clockwise neighbor), and `_prev` to this vertex's predecessor (its counterclockwise neighbor).

```
class Vertex : public Node, public Point {
 public:
   Vertex(double x, double y);
   Vertex(Point&);
   Vertex *cw(void);
   Vertex *ccw(void);
   Vertex *neighbor(int rotation);
   Point point(void);
   Vertex *insert(Vertex*);
   Vertex *remove(void);
   void splice(Vertex*);
   Vertex *split(Vertex*);
   friend class Polygon;
};
```

A `Vertex` object can be initialized from a point or from x- and y-coordinates:

```
Vertex::Vertex(double x, double y) :
   Point(x,y)
{
}
```

```
Vertex::Vertex(Point &p) :
   Point(p)
{
}
```

Member functions cw and ccw yield this vertex's successor and predecessor, respectively:

```
Vertex *Vertex::cw(void)
{
   return (Vertex*)_next;
}

Vertex *Vertex::ccw(void)
{
   return (Vertex*)_prev;
}
```

Member function neighbor returns whichever neighbor is specified by parameter rotation, one of the enumeration values CLOCKWISE or COUNTER_CLOCKWISE:

```
Vertex *Vertex::neighbor(int rotation)
{
   return ((rotation == CLOCKWISE) ? cw() : ccw());
}
```

Member function point returns the point in the plane where this vertex lies:

```
Point Vertex::point(void)
{
   return *((Point*)this);
}
```

Member functions insert, remove, and splice correspond to their counterparts defined in base class Node:

```
Vertex *Vertex::insert(Vertex *v)
{
   return (Vertex *)(Node::insert(v));
}

Vertex *Vertex::remove(void)
{
   return (Vertex *)(Node::remove());
}

void Vertex::splice(Vertex *b)
{
   Node::splice(b);
}
```

Note that remove and insert cast their return values to type pointer-to-Vertex before returning. Explicit type coersion is needed here because C++ will not automatically convert a pointer to the base class to point to a derived class object. The reason is that the C++ compiler cannot be sure that there is a derived class object present to be pointed to, since the base class object need not be part of a derived class object. (C++ *will*, on the other hand, automatically convert a pointer to the derived class to point to a base class object since every derived class object includes within itself a base class object.)

The last member function, Vertex::split, will be defined shortly.

4.3.4 The Polygon Class

A polygon is represented by a Polygon object. The class contains two data members. The first, _v, points to some vertex of the polygon, the current position of the polygon's window. Most operations on polygons refer either to this window or to the vertex in the window. We will sometimes refer to the vertex in the window as the *current vertex*. The second data member, _size, holds the size of the polygon:

```
class Polygon {
 private:
   Vertex *_v;
   int _size;
   void resize(void);
 public:
   Polygon(void);
   Polygon(Polygon&);
   Polygon(Vertex*);
   ~Polygon(void);
   Vertex *v(void);
   int size(void);
   Point point(void);
   Edge edge(void);
   Vertex *cw(void);
   Vertex *ccw(void);
   Vertex *neighbor(int rotation);
   Vertex *advance(int rotation);
   Vertex *setV(Vertex*);
   Vertex *insert(Point&);
   void remove(void);
   Polygon *split(Vertex*);
};
```

CONSTRUCTORS AND DESTRUCTORS

There are several constructors for class Polygon. The constructor that takes no arguments initializes an empty polygon:

```
Polygon::Polygon(void) :
   _v(NULL), _size(0)
{
}
```

The copy constructor takes some polygon p and initializes a new polygon with p. It performs a deep copy, duplicating the linked list in which p is stored. The new polygon's window is placed over the vertex corresponding to p's current vertex:

```
Polygon::Polygon(Polygon &p)
{
   _size = p._size;
   if (_size == 0)
      _v = NULL;
   else {
      _v = new Vertex(p.point());
      for (int i = 1; i < _size; i++) {
         p.advance(CLOCKWISE);
         _v = _v->insert(new Vertex(p.point()));
      }
      p.advance(CLOCKWISE);
      _v = _v->cw();
   }
}
```

The third constructor initializes a polygon with a circular doubly linked list of vertices:

```
Polygon::Polygon(Vertex *v) :
   _v(v)
{
   resize();
}
```

The constuctor calls private member function resize to update member _size. In general, resize must be called whenever a vertex chain of unknown length is added to or removed from a polygon. Function resize is defined as follows:

```
void Polygon::resize(void)
{
   if (_v == NULL)
      _size = 0;
   else {
      Vertex *v = _v->cw();
      for (_size = 1; v != _v; ++_size, v = v->cw())
         ;
   }
}
```

The destructor ~Polygon deallocates this polygon's vertices before deleting the Polygon object itself:

```
Polygon::~Polygon(void)
{
   if (_v) {
      Vertex *w = _v->cw();
      while (_v != w) {
         delete w->remove();
         w = _v->cw();
      }
   delete _v;
   }
}
```

Access Functions

The next several member functions access data about this polygon. Function v returns this polygon's current vertex, and function size this polygon's size:

```
Vertex *Polygon::v(void)
{
   return _v;
}

int Polygon::size(void)
{
   return _size;
}
```

The pointer returned by member function v can be used as an additional window into the polygon, to supplement the polygon's implicit window. Some applications will require the simultaneous use of several windows into the same polygon—the sole window maintained implicitly by the class does not always suffice.

Member function point returns the point in the plane where the current vertex lies. Member function edge returns the *current edge*. The current edge originates at the current vertex and terminates at the current vertex's successor:

```
Point Polygon::point(void)
{
   return _v->point();
}

Edge Polygon::edge(void)
{
   return Edge(point(), _v->cw()->point());
}
```

We will define the Edge class in the next section.

Member functions cw and ccw return the current vertex's successor and predecessor without moving the window, and neighbor returns the current vertex's successor or predecessor, depending on the argument it is called with (CLOCKWISE or COUNTER_CLOCKWISE):

```
Vertex *Polygon::cw(void)
{
   return _v->cw();
}
```

```
Vertex *Polygon::ccw(void)
{
   return _v->ccw();
}

Vertex *Polygon::neighbor(int rotation)
{
   return _v->neighbor(rotation);
}
```

Update Functions

Member functions `advance` and `setV` move the window over a different vertex; `advance` moves it to the current vertex's successor or predecessor, as specified by the argument:

```
Vertex *Polygon::advance(int rotation)
{
   return _v = _v->neighbor(rotation);
}
```

Member function `setV` moves the window over the vertex `v` supplied as an argument:

```
Vertex *Polygon::setV(Vertex *v)
{
   return _v = v;
}
```

It is the application's responsibility to ensure that `v` is a vertex of *this* polygon.

Member function `insert` inserts a new vertex after the current vertex and then moves the window over the new vertex:

```
Vertex *Polygon::insert(Point &p)
{
   if (_size++ == 0)
      _v = new Vertex(p);
   else
      _v = _v->insert(new Vertex(p));
   return _v;
}
```

Member function `remove` removes the current vertex. The window is moved over the predecessor, or is undefined if the polygon is now empty:

```
void Polygon::remove(void)
{
   Vertex *v = _v;
   _v = (--_size == 0) ? NULL : _v->ccw();
   delete v->remove();
}
```

Splitting Polygons

Polygon splitting involves subdividing a polygon into two smaller subpolygons. The cut is made along some chord. To split along chord $\overrightarrow{ab}$, we first insert a duplicate of vertex *a* after *a* and a duplicate of vertex *b* before *b* (call the duplicates *ap* and *bp*). Then we splice *a* and *bp*. The process is illustrated in Figure 4.10.

Member function `Polygon::split` is defined in terms of `Vertex::split`. The latter function partitions a polygon along the chord connecting this vertex (which plays the role of *a*) to vertex `b`. It returns a pointer to vertex `bp`, the duplicate of `b`:

```
Vertex *Vertex::split(Vertex *b)
{                                   // insert bp before vertex b
   Vertex *bp = b->ccw()->insert(new Vertex(b->point()));
   insert(new Vertex(point()));     // insert ap after this vertex
   splice(bp);
   return bp;
}
```

Function `Polygon::split` splits this polygon along the chord connecting its current vertex to vertex `b`. It returns a pointer to the new polygon, whose window is placed over `bp`, the duplicate of `b`. This polygon's window is not moved:

```
Polygon *Polygon::split(Vertex *b)
{
   Vertex *bp = _v->split(b);
   resize();
   return new Polygon(bp);
}
```

Function `Polygon::split` must be used with some care. If vertex `b` is the successor to the current vertex `_v`, the operation leaves this polygon unchanged. If the cut occurs along a diagonal that is not a chord, one or both of the resulting "polygons" may self-cross. If vertices `b` and `_v` belong to different polygons, the split operation joins the two polygons by two coincident edges that connect the two vertices.

4.3.5 Point Enclosure in a Convex Polygon

In this and the following subsection, we present two simple programs involving polygons. Program `pointInConvexPolygon` is handed a point `s` and convex polygon `p`, and

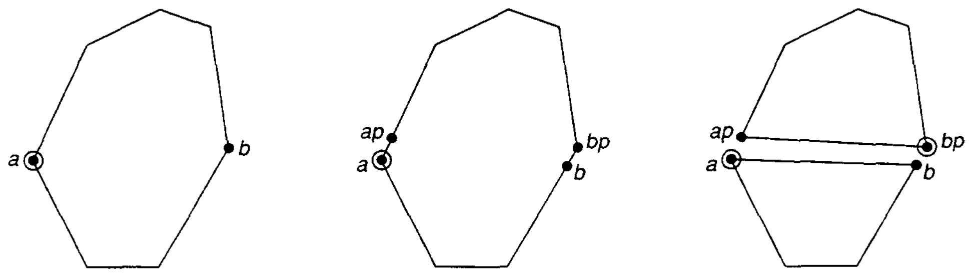


Figure 4.10: Splitting a polygon along chord $\overline{ab}$. The current vertices (in each polygon's window) are circled.

returns TRUE just if the point lies in (the interior or boundary of) polygon p:

```
bool pointInConvexPolygon(Point &s, Polygon &p)
{
   if (p.size() == 1)
      return (s == p.point());
   if (p.size() == 2) {
      int c = s.classify(p.edge());
      return ((c==BETWEEN) || (c==ORIGIN) || (c==DESTINATION));
   }
   Vertex *org = p.v();
   for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE))
      if (s.classify(p.edge()) == LEFT) {
         p.setV(org);
         return FALSE;
      }
   return TRUE;
}
```

The preceding function first handles the special cases in which polygon p is a 1-gon or a 2-gon. In the general case, the algorithm traverses the polygon boundary—moves the window from vertex to adjecent vertex—while comparing point s to each edge in turn. Since p is assumed to be convex, point s lies outside the polygon only if s lies to the left of some edge. Note that the program restores the initial position of p's window upon returning.

4.3.6 Finding the Least Vertex in a Polygon

The following function is passed a polygon p and a comparison function cmp, and then finds the least vertex in p. Here *least vertex* means whichever vertex is less than the others under the linear ordering of points given by cmp. Function leastVertex moves p's window over the least vertex and returns the vertex:

```
Vertex *leastVertex(Polygon &p, int (*cmp)(Point*,Point*))
{
   Vertex *bestV = p.v();
   p.advance(CLOCKWISE);
   for (int i = 1; i < p.size(); p.advance(CLOCKWISE), i++)
      if ((*cmp)(p.v(), bestV) < 0)
         bestV = p.v();
   p.setV(bestV);
   return bestV;
}
```

For instance, to find the leftmost vertex in a polygon, we call leastVertex with the following comparison function:

```
int leftToRightCmp(Point *a, Point *b)
{
   if (*a < *b) return -1;
   if (*a > *b) return 1;
   return 0;
}
```

We use the following comparison function to find the rightmost vertex:

```
int rightToLeftCmp(Point *a, Point *b)
{
   return leftToRightCmp(b, a);
}
```

We will use functions `pointInConvexPolygon` and `leastVertex` often in this book. We will also use the two comparison functions defined here, as well as others to be defined later as the need arises.

4.4 Edges

Most every algorithm we will cover involves lines in one form or another. The *line segment* $\overline{p_0p_1}$ consists of the *endpoints* p_0 and p_1 together with the points that lie between them. When the order of p_0 and p_1 is important, we speak of the *directed line segment* $\overrightarrow{p_0p_1}$. Endpoint p_0 is the *origin* of the directed line segment, and p_1 the *destination*. We will usually refer to a directed line segment as an *edge* when it is the side of some polygon; the edge is directed so that the polygon's interior lies to its right. An *infinite (directed) line* is determined by two points and is directed from the first point to the second. A *ray* is a semi-infinite line starting at the origin and passing through the destination.

4.4.1 The `Edge` Class

The `Edge` class will be used to represent all forms of lines. The class is defined as follows:

```
class Edge {
 public:
   Point org;
   Point dest;
   Edge(Point &_org, Point &_dest);
   Edge(void);
   Edge &rot(void);
   Edge &flip(void);
   Point point(double);
   int intersect(Edge&, double&);
   int cross(Edge&, double&);
   bool isVertical(void);
   double slope(void);
   double y(double);
};
```

An edge's origin and destination endpoints are stored in data members `org` and `dest`, respectively. The `Edge` constructor initializes these data members:

```
Edge::Edge(Point &_org, Point &_dest) :
   org(_org), dest(_dest)
{
}
```

It is also useful to have a constructor for class `Edge` which takes no arguments:

```
Edge::Edge(void) :
   org(Point(0,0)), org(Point(1,0))
{
}
```

4.4.2 Edge Rotations

An *edge rotation* pivots an edge 90 degrees clockwise around its midpoint. Two successive edge rotations are called an *edge flip* since they reverse the direction of an edge. Three successive rotations effectively pivot an edge 90 degrees counterclockwise around its midpoint. Four successive edge rotations leave an edge unchanged. This is illustrated in Figure 4.11.

Figure 4.12 shows how we rotate edge $\overrightarrow{ab}$ into edge $\overrightarrow{cd}$. Where vector $b-a = (x, y)$, the vector n, perpendicular to vector $b-a$, is given by $n = (y, -x)$. The midpoint m between endpoints a and b is given by $m = \frac{1}{2}(a+b)$. Points c and d are then given by $c = m - \frac{1}{2}n$ and $d = m + \frac{1}{2}n$. Rotation is implemented by member function `rot` as follows:

```
Edge &Edge::rot(void)
{
   Point m = 0.5 * (org + dest);
   Point v = dest - org;
   Point n(v.y, -v.x);
   org = m - 0.5 * n;
   dest = m + 0.5 * n;
   return *this;
}
```

Observe that function `rot` is *destructive*: It changes the current edge instead of creating a new edge. The function returns a reference to this edge so calls to `rot` can be readily employed in more complex expressions. This permits, for example, the following concise definition of member function `flip`, for flipping the direction of this edge:

```
Edge &Edge::flip(void)
{
   return rot().rot();
}
```

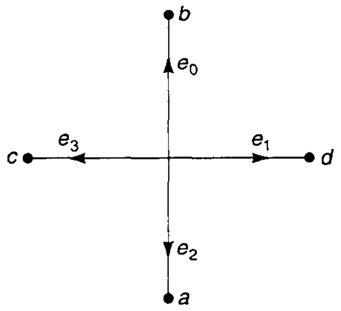


Figure 4.11: Edge e_i is the result of applying i successive edge rotations to edge e_0.

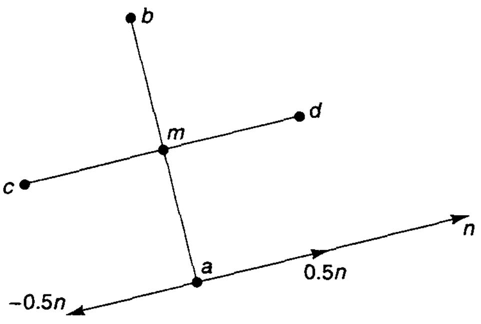


Figure 4.12: Vectors involved in rotating edge $\overline{ab}$. The rotated edge $\overline{cd}$ has endpoints $c = m - \frac{1}{2}n$ and $d = m + \frac{1}{2}n$.

Within the definition of member function `flip`, the first call to `rot` (to the left of the member-access operator) rotates this edge; the second call to `rot` then rotates this edge once again.

4.4.3 Finding the Intersection of Two Lines

The infinite line $\overleftrightarrow{ab}$ through points a and b can be written in parametric form as

$$P(t) = a + t(b - a) \qquad [4.2]$$

where the value of parameter t ranges over the real numbers. (If the value of t is restricted to the range $0 \leq t \leq 1$, Equation 4.2 represents the line segment $\overline{ab}$.) The parametric form of a line establishes a correspondence between the real numbers and the points on the line. Figure 4.13 shows the points on an infinite line corresponding to various values of parameter t.

Member functions `Edge::intersect` and `Edge::point` are designed to work together to find the intersection point of two infinite lines `e` and `f`. Where `e` and `f` are `Edge` objects, the code fragment

```
double t;
Point p;
if (e.intersect(f, t) == SKEW)
   p = e.point(t);
```

assigns `t` the parametric value (along line `e`) of the point at which lines `e` and `f` intersect, and then sets `p` to this point. Function `intersect` returns the enumeration value `SKEW` if the infinite lines cross at a point, `COLLINEAR` if the lines are collinear, or `PARALLEL` if they are parallel. Function `point` is handed a parametric value `t` and returns the corresponding point. The task is performed by two coordinated functions, rather than by a single function,

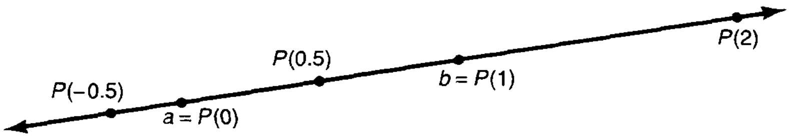


Figure 4.13: Various points on the line through points a and b.

because we are sometimes interested only in the parametric value of an intersection point rather than in the intersection point itself.

The implementation of member function `point` is simple—the parametric value `t` is substituted into the parametric equation for this line:

```
Point Edge::point(double t)
{
   return Point(org + t * (dest - org));
}
```

The implementation of member function `intersect` relies on the notion of the *dot product* $a \cdot b$ of two vectors $a = (x_a, y_a)$ and $b = (x_b, y_b)$, which is defined by $a \cdot b = x_a x_b + y_a y_b$. The dot product has a number of important properties, including the following basic ones:

1. Where a, b, and c are vectors, we have $a \cdot b = b \cdot a$ and
2. $a \cdot (b + c) = a \cdot b + a \cdot c = (b + c) \cdot a$.
3. Where s is a scalar, $(sa) \cdot b = s(a \cdot b)$ and $a \cdot (sb) = s(a \cdot b)$.
4. If a is the zero vector, then $a \cdot a = 0$; otherwise $a \cdot a > 0$.
5. $\|a\|^2 = a \cdot a$.

Using these basic properties, we can show the following property on which our line-intersection technique depends: Two vectors a and b are perpendicular if and only if $a \cdot b = 0$. To see why this is true, observe that a and b are perpendicular if and only if

$$\|a - b\| = \|a + b\|$$

This is illustrated in Figure 4.14a. Squaring both sides yields

$$(a - b) \cdot (a - b) = (a + b) \cdot (a + b)$$

Using the aforementioned properties 1 through 3, this expands to

$$a \cdot a - 2a \cdot b + b \cdot b = a \cdot a + 2a \cdot b + b \cdot b$$

Making cancellations yields

$$4a \cdot b = 0$$

or

$$a \cdot b = 0$$

Hence $a \cdot b = 0$ if and only if vectors a and b are perpendicular.

We can say even more. If the angle between vectors a and b measures less than 90 degrees, then $\|a - b\| < \|a + b\|$ (Figure 4.14b). The same sort of argument can be used to show that this is equivalent to the condition $a \cdot b > 0$. It can be shown similarly that the angle between a and b measures greater than 90 degrees if and only if $a \cdot b < 0$ (Figure 4.14c). These results are summarized by the following theorem:

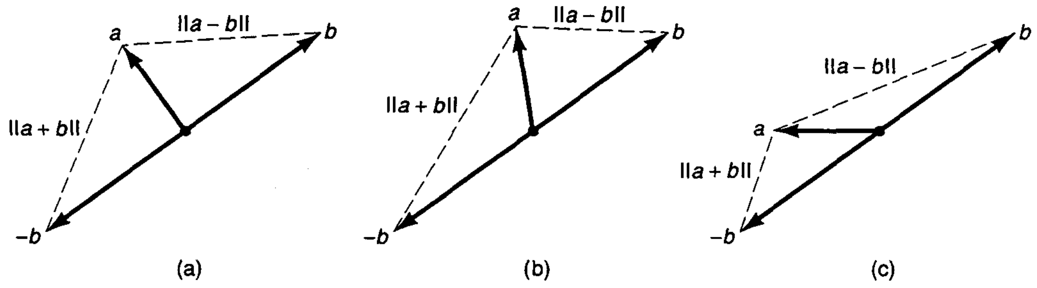


Figure 4.14: The angle between vectors a and b measures (a) 90 degrees if $\|a - b\| = \|a + b\|$, (b) less than 90 if $\|a - b\| < \|a + b\|$, and (c) greater than 90 if $\|a - b\| > \|a + b\|$.

Theorem 2 (Dot Product Theorem) *Let a and b be vectors, and let θ be the angle between them. Then*

$$a \cdot b \left\{ \begin{array}{c} > \\ = \\ < \end{array} \right\} 0 \text{ if and only if } \theta \left\{ \begin{array}{c} < \\ = \\ > \end{array} \right\} 90 \text{ degrees.}$$

The dot product theorem can be used to find the intersection point of two lines $\overleftrightarrow{ab}$ and $\overleftrightarrow{cd}$. Where $\overleftrightarrow{ab}$ is described by $P(t) = a + t(b - a)$, we seek the value of t such that lines $\overleftrightarrow{ab}$ and $\overleftrightarrow{cd}$ cross at point $P(t)$. Since vector $P(t) - c$ is to coincide with line $\overleftrightarrow{cd}$, both $P(t) - c$ and $\overleftrightarrow{cd}$ must be perpendicular to the same vector n. Therefore, using the dot product theorem, we wish to solve for t in the equation

$$n \cdot (P(t) - c) = 0 \qquad [4.3]$$

Since $P(t) = a + t(b - a)$, we can rewrite Equation 4.3 as

$$n \cdot ((a + t(b - a)) - c) = 0$$

Using the basic properties of dot product yields

$$n \cdot (a - c) + n \cdot (t(b - a)) = 0$$

Then distributing out t gives us

$$n \cdot (a - c) + t[n \cdot (b - a)] = 0$$

From this it follows that

$$t = -\frac{n \cdot (a - c)}{n \cdot (b - a)}, \quad n \cdot (b - a) \neq 0 \qquad [4.4]$$

Equation 4.4 holds if and only if infinite lines $\overleftrightarrow{ab}$ and $\overleftrightarrow{cd}$ are skew, implying that they intersect in a single point. If the two lines are parallel or coincident, the fact is indicated by the condition that $n \cdot (b - a) = 0$, since vectors $b - a$ and $d - c$ are then both perpendicular to the same vector n. The following implementation of member function `intersect` results:

```
enum { COLLINEAR, PARALLEL, SKEW, SKEW_CROSS, SKEW_NO_CROSS };

int Edge::intersect(Edge &e, double &t)
{
   Point a = org;
   Point b = dest;
   Point c = e.org;
   Point d = e.dest;
   Point n = Point((d-c).y, (c-d).x);
   double denom = dotProduct(n, b-a);
   if (denom == 0.0) {
      int aclass = org.classify(e);
      if ((aclass==LEFT) || (aclass==RIGHT))
         return PARALLEL;
      else
         return COLLINEAR;
   }
   double num = dotProduct(n, a-c);
   t = -num / denom;
   return SKEW;
}
```

The implementation of function `dotProduct` is straightforward:

```
double dotProduct(Point &p, Point &q)
{
   return (p.x * q.x + p.y * q.y);
}
```

Member function `Edge::cross` returns `SKEW_CROSS` if and only if this line segment intersects line segment `e`. If the line segments do intersect, the parametric value along this line segment corresponding to the point of intersection is returned through reference parameter `t`. Otherwise the function returns `COLLINEAR`, `PARALLEL`, or `SKEW_NO_CROSS`, as appropriate:

```
int Edge::cross(Edge &e, double &t)
{
   double s;
   int crossType = e.intersect(*this, s);
   if ((crossType==COLLINEAR) || (crossType==PARALLEL))
      return crossType;
   if ((s < 0.0) || (s > 1.0))
      return SKEW_NO_CROSS;
   intersect(e, t);
   if ((0.0 <= t) && (t <= 1.0))
      return SKEW_CROSS;
   else
      return SKEW_NO_CROSS;
}
```

4.4.4 Distance from a Point to a Line

The definition of function `Point::distance` illustrates some of the ideas we have just covered. This member function of class `Point` is passed an edge `e`, and it returns the signed distance from this point to edge `e`. Here the *distance* from point `p` to edge `e` equals the minimum distance from `p` to any point along the infinite line determined by `e`. The *signed distance* is positive if `p` lies to the right of `e`, negative if `p` lies to the left of `e`, and zero if `p` is collinear with `e`.

Member function `distance` is defined as follows:

```
double Point::distance(Edge &e)
{
   Edge ab = e;
   ab.flip().rot();  // rotate ab 90 degrees counter-clockwise
   Point n(ab.dest - ab.org);  // n = vector perpendicular to e
   n = (1.0 / n.length()) * n; // normalize n
   Edge f(*this, *this + n);   // f = n, positioned at this point
   double t;
   f.intersect(e, t);          // t = signed distance along f
                               // at which f crosses edge e
   return t;
}
```

The function first obtains the unit-length vector `n`, such that `n` is perpendicular to edge `e` and `n` points to the left of `e`. It then translates `n` such that `n`'s origin coincides with this point, yielding edge `f`. Finally, the function computes the parametric value of edge `f`'s intersection with edge `e`. Since `f` is perpendicular to `e`, is of unit length, and originates at this point, parametric value `t` equals the signed distance from this point to edge `e`.

4.4.5 Additional Utilities

The last three member functions of class `Edge` are provided for convenience. Member function `isVertical` returns `TRUE` only if this edge is vertical:

```
bool Edge::isVertical(void)
{
   return (org.x == dest.x);
}
```

Member function `slope` returns the slope of this edge, or `DBL_MAX` if this edge is vertical:

```
double Edge::slope(void)
{
   if (org.x != dest.x)
      return (dest.y - org.y) / (dest.x - org.x);
   return DBL_MAX;
}
```

Member function y is passed a value x and returns the value y such that (x, y) is a point on this infinite line. The function is defined only if this edge is not vertical:

```
double Edge::y(double x)
{
   return slope() * (x - org.x) + org.y;
}
```

4.5 Geometric Objects in Space

Although we will work mainly in the plane, a few sections of this book will involve geometric objects in three-dimensional space. In this section we will present the classes Point3D, Triangle3D, and Edge3D for manipulating points, triangles, and edges lying in space. The class definitions will be bare bones, providing little more than the functionality we will need. Moreover, for the sake of conciseness, many of the member functions will be defined within the definition of their classes and will be described tersely. This should not hinder clarity since most of the relevant concepts have already been explained in the setting of the two-dimensional plane; new concepts will be discussed in more detail.

4.5.1 Points

Under the Cartesian coordinate system, a point in space is represented by an ordered triple (x, y, z) of real numbers. The Point3D class contains data members x, y, and z to hold a point's coordinates, a constructor, operator functions for the basic vector operations, the operator function [] for coordinate access, a member function for computing dot product, and one for classifying a point relative to a plane:

```
class Point3D {
 public:
   double x;
   double y;
   double z;
   Point3D(double _x, double _y, double _z) :
      x(_x), y(_y), z(_z) {}
   Point3D(void)
      {}
   Point3D operator+(Point3D &p)
      { return Point3D(x + p.x, y + p.y, z + p.z); }
   Point3D operator-(Point3D &p)
      { return Point3D(x - p.x, y - p.y, z - p.z); }
   friend Point3D operator*(double, Point3D &);
   int operator==(Point3D &p)
      { return ((x == p.x) && (y == p.y) && (z == p.z)); }
   int operator!=(Point3D &p)
      { return !(*this == p); }
   double operator[](int i)
      { return ((i == 0) ? x : ((i == 1) ? y : z)); }
   double dotProduct(Point3D &p)
      { return (x*p.x + y*p.y + z*p.z); }
   int classify(Triangle3D &t);
};
```

Scalar multiplication is implemented like this:

```
Point3D operator*(double s, Point3D &p)
{
   return Point3D(s * p.x, s * p.y, s * p.z);
}
```

Member function `classify` reports which side of the plane determined by triangle `t` this point lies in. Its definition will be given in the following subsection.

4.5.2 Triangles

A triangle is determined by its three vertices. For working with triangles in space, it is useful to keep track of each triangle's bounding box and normal vector, as well as its vertices. The *bounding box* of a geometric object is the smallest box that contains the object, where the edges of the box are parallel to the major axes. Figure 4.15 gives some examples.

A vector perpendicular to a given plane P is called a *normal* to P. Given any three non-collinear points p_0, p_1, and p_2 lying in plane P, a normal to P is given by the *cross product* vector $a \times b$, where vectors $a = p_1 - p_0$ and $b = p_2 - p_0$. Letting $a = (x_a, y_a, z_a)$ and $b = (x_b, y_b, z_b)$, the cross product vector is defined by

$$a \times b = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b) \qquad [4.5]$$

The cross product of vectors `a` and `b` is returned by the following function:

```
Point3D crossProduct(Point3D &a, Point3D &b)
{
   return Point3D(a.y * b.z - a.z * b.y,
                  a.z * b.x - a.x * b.z,
                  a.x * b.y - a.y * b.x);
}
```

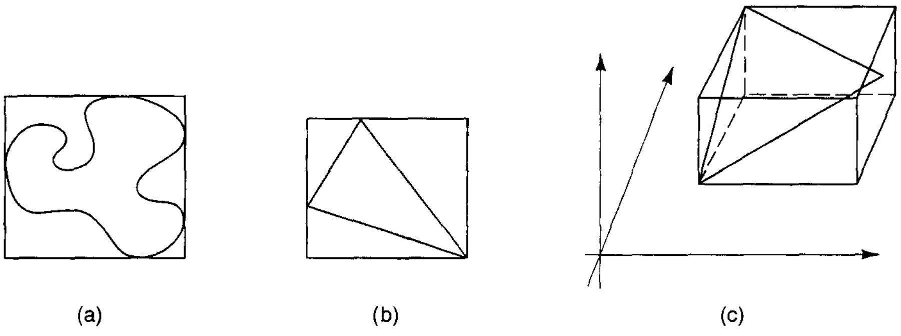


Figure 4.15: Bounding box of (a) a blob in the plane, (b) a triangle in the plane, and (c) a triangle in space.

To show that the cross product vector $a \times b$ is perpendicular to the plane spanned by vectors a and b, we need only show that $a \cdot (a \times b) = 0$ and $b \cdot (a \times b) = 0$. We have

$$\begin{aligned} a \cdot (a \times b) &= (x_a, y_a, z_a) \cdot (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b) \\ &= 0 \end{aligned}$$

since all terms cancel. That $b \cdot (a \times b) = 0$ is shown similarly.

The direction of the cross product vector is shown in Figure 4.16. When viewed from point $a \times b$ in space, triangle $\Delta 0ab$ is positively oriented. The normal vector having the same length but opposite direction is given by $-a \times b = b \times a$.

Observe that if vectors a and b lie in the xy-plane, then the length of their cross product is $\|a \times b\| = |x_a y_b - y_a x_b|$, the area of the parallelogram with vertices $\mathbf{0}$, a, b, and $a + b$.

Having discussed bounding boxes and normal vectors, we can define the `Triangle3D` class:

```
class Triangle3D {
 private:
   Point3D _v[3];
   Edge3D _boundingBox;
   Point3D _n;
 public:
   int id;
   int mark;
   Triangle3D(Point3D &v0, Point3D &v1, Point3D &v2, int id);
   Triangle3D(void)
      {}
   Point3D operator[](int i)
      { return _v[i]; }
   Edge3D boundingBox()
      { return _boundingBox; }
   Point3D n(void)
      { return _n; }
   double length(void)
      { return sqrt(x*x + y*y + z*z);
};
```

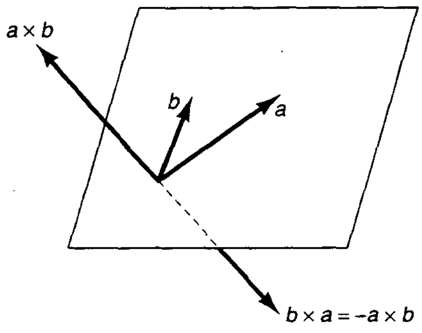


Figure 4.16: The cross product $a \times b$ of vectors a and b.

This triangle's vertices are stored in array _v. Its bounding box is represented by the edge _boundingBox extending from the bounding box's minimum-coordinate corner to its maximum-coordinate corner. The unit normal to the triangle, stored in data member _n, equals the cross product vector (_v[1]-_v[0]) × (_v[2]-_v[0]), divided by its length. Data member id is an identifier for this triangle.

The first constructor Triangle3 makes use of the macro functions max3 and min3 for finding the largest and smallest of three numbers:

```
#define min3(A,B,C)    \\
   ((A)<(B) ? ((A)<(C)?(A):(C)) : ((B)<(C)?(B):(C)))
#define max3(A,B,C)    \\
   ((A)>(B) ? ((A)>(C)?(A):(C)) : ((B)>(C)?(B):(C)))

Triangle3D::Triangle3D(Point3D &v0, Point3D &v1, Point3D &v2,
                       int _id)
{
   id = _id;
   mark = 0;
   _v[0] = v0;
   _v[1] = v1;
   _v[2] = v2;
   _boundingBox.org.x = min3(v0.x, v1.x, v2.x);
   _boundingBox.org.y = min3(v0.y, v1.y, v2.y);
   _boundingBox.org.z = min3(v0.z, v1.z, v2.z);
   _boundingBox.dest.x = max3(v0.x, v1.x, v2.x);
   _boundingBox.dest.y = max3(v0.y, v1.y, v2.y);
   _boundingBox.dest.z = max3(v0.z, v1.z, v2.z);
   _n = crossProduct(v1 - v0, v2 - v0);
   _n = (1.0 / _n.length()) * _n;
}
```

The vertices of a Triangle3D object are accessed through operator [], which is passed the index of the vertex (0, 1, or 2). For instance, where t is a Triangle3D object, t[0] yields t's first vertex. The bounding box and the unit normal vector are accessed through member functions boundingBox and n, respectively. The geometric data members are declared private so the class can ensure self-consistency.

The plane determined by a triangle subdivides space into two half-spaces. The half-space into which the triangle's normal vector points is called the triangle's *positive half-space* since the triangle appears to be positively oriented when viewed from this half-space. The other half-space is called the triangle's *negative half-space*.

With the definition of class Triangle3D in hand, we are in a position to define member function Point3D::classify. Recall that the function reports the half-space—relative to a given triangle p—in which this point lies. The function returns POSITIVE or NEGATIVE if this point lies in p's positive or negative half-space; it returns ON if this point lies on the plane determined by p:

```
#define EPSILON1   1E-12
enum { POSITIVE, NEGATIVE, ON };
```

```
int Point3::classify(Triangle3 &p)
{
   Point3 v = *this - p[0];
   double len = v.length();
   if (len == 0.0)
      return ON;
   v = (1.0 / len) * v;
   double d = v.dotProduct(p.n());
   if (d > EPSILON1)
      return POSITIVE;
   else if (d < -EPSILON1)
      return NEGATIVE;
   else
      return ON;
}
```

Vector v represents a directed line segment which originates at some point on the plane (p[0]) and terminates at the point to be classified (*this). The dot product theorem is used to decide whether the angle between v and the plane's normal vector n is less than, equal to, or greater than 90 degrees.

The function centers the plane of triangle tri within a slab of width 2*EPSILON1. A point which lies within this slab is considered to lie on the plane. This is intended to avoid faulty decisions attributable to round-off, such as when a point on the plane appears to lie off the plane due to limitations of representation.

4.5.3 Edges

The Edge3D class is defined as follows:

```
class Edge3D {
 public:
   Point3D org;
   Point3D dest;
   Edge3D(Point3D &_org, Point3D &_dest) :
      org(_org), dest(_dest) {}
   Edge3D(void)
      {}
   int intersect(Triangle3D &p, double &t);
   Point3D point(double t);
};
```

The first constructor initializes an edge with origin and destination endpoints, which are stored in data members org and dest. Member functions intersect and point play the same role as their counterparts in class Edge. Function intersect finds the parametric value of the infinite line determined by this edge, at the point where the line crosses the plane of triangle p. If the line and plane intersect at a point, the function passes back the parametric value via reference parameter t and returns the enumeration value SKEW; otherwise it returns either PARALLEL or COLLINEAR. Like its counterpart Edge::intersect, member function intersect is implemented using Equation 4.4:

```
int Edge3D::intersect(Triangle3D &p, double &t)
{
   Point3D a = org;
   Point3D b = dest;
   Point3D c = p[0];    // some point on the plane
   Point3 n = p.n();
   double denom = n.dotProduct(b - a);
   if (denom == 0.0) {
      int aclass = org.classify(p);
      if (aclass!=ON)
         return PARALLEL;
      else
         return COLLINEAR;
   }
   double num = n.dotProduct(a - c);
   t = -num / denom;
   return SKEW;
}
```

Member function point returns the point along this line corresponding to parametric value t:

```
Point3D Edge3D::point(double t)
{
   return org + t * (dest - org);
}
```

4.6 Finding the Intersection of a Line and a Triangle

In this section we solve a problem using some of the tools presented in this chapter. Our solution to the problem—that of deciding whether a line pierces a triangle in space—will prove useful later in this book.

A *projection* is a mapping from a higher-dimensional space into a lower-dimensional space. One of its uses is to transform a problem from a higher-dimensional setting to an equivalent problem in a lower-dimensional setting, where there are techniques to solve it. Consider the problem of deciding whether a given infinite line intersects a given triangle p in space. Figure 4.17a depicts one approach to this problem. First compute the point q where the infinite line pierces the plane of triangle p. Then perpendicularly project both p and q into the xy-plane, yielding triangle p' and point q'. The resulting problem in two dimensions—that of deciding whether p' contains q'—is equivalent to the original problem: The answer to the two-dimensional problem is yes if and only if the answer to the original three-dimensional problem is yes. The advantage in applying this transformation is that the two-dimensional problem is easier to solve than the original three-dimensional problem.

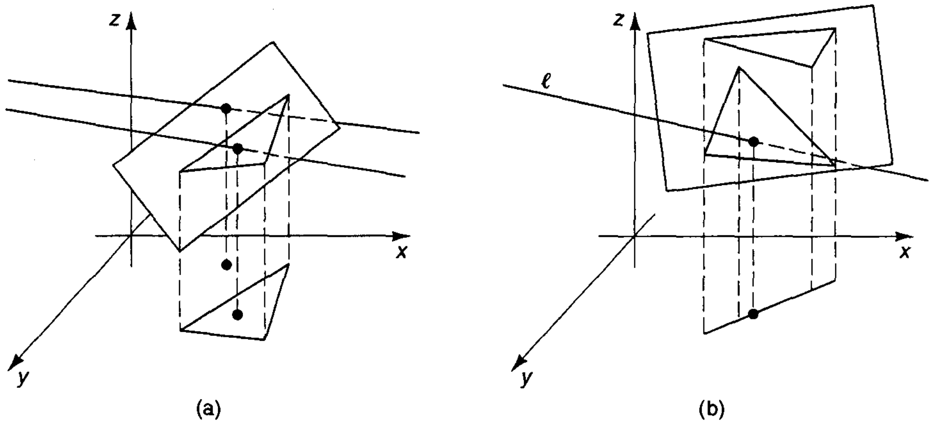


Figure 4.17: (a) Deciding whether a line pierces triangle p. (b) Both triangles project degenerately to the same line segment.

Projection is a many-to-one mapping, and difficulties can arise when too much information is lost. In Figure 4.17b, both triangles project (degenerately) to the same line segment in the xy-plane. It is not hard to see why this two-dimensional problem is not equivalent to the original problem. Given two triangles p_1 and p_2 that project to the same line segment and an infinite line ℓ in space, the two three-dimensional problems that result—one involving ℓ and p_1, the other involving ℓ and p_2—transform to the same two-dimensional problem in the xy-plane. Yet if ℓ pierces (say) triangle p_1 but not triangle p_2, the two-dimensional problem must report a wrong answer in one of the two cases.

To save the algorithm, we test for degeneracy before projecting. A triangle p projects to a line segment in the xy-plane if the triangle's normal vector n is perpendicular to the z-axis. Before projecting, we perform this test; if n is perpendicular to the z-axis, we consider projecting into the yz-plane instead; and if this too would be degenerate, we finally project into the zx-plane. Since vector n cannot be perpendicular to all three axes, at least one of the three projections proves non-degenerate.

The algorithm is implemented by the following function, whose return value—PARALLEL, COLLINEAR, SKEW_CROSS, or SKEW_NO_CROSS—indicates the relationship between infinite line e and triangle p. If the function returns either SKEW_CROSS, indicating that the line pierces the triangle, or SKEW_NO_CROSS, indicating that the line crosses the plane of the triangle without piercing the triangle itself, then the parametric value of the intersection point along e is passed back through reference parameter t:

```
int lineTriangleIntersect(Edge3D &e, Triangle3D &p, double &t)
{
   Point3D q;
   int aclass = e.intersect(p, t);
   if ((aclass==PARALLEL) || (aclass==COLLINEAR))
      return aclass;
   q = e.point(t);
   int h, v;
   if (p.n().dotProduct(Point3D(0,0,1)) != 0.0) {
      h = 0;
      v = 1;
   } else if (p.n().dotProduct(Point3D(1,0,0)) != 0.0) {
      h = 1;
      v = 2;
   } else {
      h = 2;
      v = 0;
   }
   Polygon *pp = project(p, h, v);
   Point qp = Point(q[h], q[v]);
   int answer = pointInConvexPolygon(qp, *pp);
   delete pp;
   return (answer ? SKEW_CROSS : SKEW_NO_CROSS);
}
```

The function call `project(p,h,v)` returns a polygon representing the projection of triangle p into the hv-plane. Arguments h and v are axis indices; for instance, `project(p,0,1)` projects p into the xy-plane. Function `project` assumes that the projection of triangle p is non-degenerate, so its projection is a triangle. The function is defined as follows:

```
Polygon *project(Triangle3D &p, int h, int v)
{
   // project vertices of triangle p
   Point3D a;
   Point pts[3];
   for (int i = 0; i < 3; i++) {
      a = p.v(i);
      pts[i] = Point(a[h], a[v]);
   }
   // insert first two projected vertices into polygon
   Polygon *pp = new Polygon;
   for (i = 0; i < 2; i++)
      pp->insert(pts[i]);
   // insert third projected vertex into polygon
   if (pts[2].classify(pts[0], pts[1]) == LEFT)
      pp->advance(CLOCKWISE);
   pp->insert(pts[2]);
   return pp;
}
```

The only tricky part of function `project` involves insertion of the last of the three projected vertices (`pts[2]`) into the polygon under construction. If the three projected vertices are negatively oriented, then `pts[2]` belongs after `pts[1]`; if positively oriented, `pts[2]` belongs after `pts[0]`. This ensures that the interior of the resulting polygon pp lies to the *right* of each of its edges—that successive calls to `pp->advance(CLOCKWISE)` corresponds to clockwise traversal. Advancing pp's window if the projected vertices are positively oriented does the trick.

4.7 Chapter Notes

Most of the mathematics in this chapter comes from vector algebra, also known as linear algebra. Vectors are the elements of an algebraic structure known as a vector space. Although most aspects of linear algebra admit a geometric interpretation (and our presentation has concentrated on such an interpretation), all the results of linear algebra can be derived using algebra, without appealing to geometry. Introductions to linear algebra are provided by [41, 44].

A number of other books present geometric tools at the level of working code and put them to work in geometric algorithms [3, 20, 61, 66, 73]. Some of the ideas of this chapter can be found in these sources.

4.8 Exercises

1. Show that $x_a y_b - x_b y_a$ equals the signed area of the parallelogram determined by vectors $a = (x_a, y_a)$ and $b = (x_b, y_b)$.
2. Given nonzero vectors a and b, show that $a \cdot b = \|a\| \; \|b\| \; \cos\theta$, where θ is the angle between a and b.
3. Show that $\sin(\alpha - \beta) = \sin\alpha \; \cos\beta \; - \; \cos\alpha \; \sin\beta$.
4. Show that the convex polygon with vertices $v_1, \ldots, v_k$ consists of the set of points of the form $p = \alpha_1 v_1 + \cdots + \alpha_k v_k$, where $\alpha_1 + \cdots + \alpha_k = 1$ and each $\alpha_i \geq 0$. (This expression is known as the *convex combination* of points $v_1, \ldots, v_k$.)
5. Show that the dot product theorem remains valid for vectors in three-dimensional space.
6. Why does the copy constructor `Polygon::Polygon(Polygon&)` perform a deep copy? (Hint: If two polygon objects referred to the same linked list of vertices, what could go wrong?)
7. What are the advantages and disadvantages of representing the various kinds of lines (infinite lines, line segments, rays, etc.) using a single `Edge` class?
8. Write a version of `Polygon::split` that performs error checking.
9. Using the splice operation for circular doubly linked lists, write a (destructive) function `join(Polygon &p, Polygon &q)` which merges polygons `p` and `q` into a single polygon and returns a pointer to the new polygon.
10. Write a function that decides whether a polygon is convex.
11. Devise a data structure for representing a convex n-gon that permits us to decide in $O(\log n)$ time whether a given point belongs to the polygon.
12. Write a function to determine whether a given diagonal of a given polygon is a chord.

13. Write a function to determine whether a given `Polygon` object represents an illegal n-gon, one that crosses itself. [The obvious approach, that of comparing all pairs of edges, takes $O(n^2)$ time. Can you think of an algorithm that takes $O(n \log n)$ time?]
14. Devise an algorithm to decide whether a point belongs to an arbitrary (i.e., convex or nonconvex) n-gon that runs in $O(n)$ time.
15. Devise an $O(n \log n)$ time algorithm to decide whether two polygons intersect, where n equals the sum of their sizes.
16. Devise an $O(n)$ time algorithm to decide whether two *convex* polygons intersect, where n equals the sum of their sizes.
17. Write a function to find the intersection point of a line and a triangle in space which does not rely on projection into a plane.
18. Write a function to determine whether two triangles in space intersect.

II

Applications

5

Incremental Insertion

The algorithmic design approach of *incremental insertion* examines the input to a problem one item at a time while maintaining a current solution for those items seen so far. At each increment, the next input item is examined and processed, and the current solution is updated to accommodate the new item. When all the input has been processed, the problem as a whole has been solved.

One reads a mystery novel in much the same manner. The reader maintains a working hypothesis concerning who committed the murder and how and why it took place. Each new clue either confirms the hypothesis or requires that it be revised, or even abandoned and formulated anew. By the book's end when all the clues are in, the reader will have solved the crime, assuming he or she is clever enough and the writer has been fair.

In some cases, the algorithm is capable of maintaining only a current *state* as opposed to a current *solution*, since the portion of the input seen so far is too incomplete to represent a coherent situation. This often happens, for instance, when solving problems involving polygons: If the polygon boundary is processed a vertex at a time, we may not even have our hands on a simple polygon until all the input has been processed. Returning to our mystery novel analogy, we see this is similar to the way the reader's outlook develops even before any murder has taken place—although there is not yet a problem to solve, early clues and insights are organized and readied for use at the first sign of trouble.

The most obvious approach to finding the smallest integer in an array—stepping down the array while keeping track of the smallest integer seen so far—is a computational example of incremental insertion. Insertion involves a conditional assignment to the variable holding the current minimum. At each stage, this variable holds the answer to the problem involving those integers processed so far. Incremental insertion is not usually so simple.

In this chapter we will study a number of (more interesting) algorithms that employ this strategy. The first, insertion sort, is a well-known sorting method most useful for sorting a relatively short list of items. The remaining algorithms solve geometric problems: finding a star-shaped polygon in a finite set of points, finding the convex hull of a set of points, deciding whether a given point lies in a polygon, clipping geometric objects (lines and polygons) to a convex polygon, and triangulating a monotone polygon.

5.1 Insertion Sort

Insertion sort works the way a card player keeps a hand of cards. With the deck face down on the table, the card player draws a number of cards; as the player draws each card, he or she inserts it into the proper position in the hand. When each new card is about to be drawn, the hand is sorted over all the cards that have been drawn so far.

Let us consider how to use insertion sort to arrange array items `a[0]`,...,`a[n-1]` in increasing order. (For brevity, we will refer to this range of items as `a[0..n-1]`.) For each `i` from 1 through $n-1$, at the start of iteration `i` the subarray `a[0..i-1]` is sorted. Our task in iteration i is to sort `a[0..i]` by putting item `a[i]` in its proper position. To do this, we save `a[i]` in some variable `v` and then move items `a[i-1]`, `a[i-2]`,... in turn one position to the right until reaching the first item `a[j-1]` not greater than `v`. Finally, we copy `v` into the "hole" that has been created in position `j`. Figure 5.1 shows how the algorithm sorts a short array of integers.

The algorithm is implemented by function template `insertionSort`, which sorts the array `a[0..n-1]`. Argument `cmp` is a comparison function that returns -1, 0, or 1 if its first argument is less than, equal to, or greater than its second argument:

```
template<class T>
void insertionSort(T a[], int n, int (*cmp)(T,T))
{
   for (int i = 1; i < n; i++) {
      T v = a[i];
      int j = i;
      while ((j > 0) && ((*cmp)(v, a[j-1]) < 0)) {
         a[j] = a[j-1];
         j--;
      }
      a[j] = v;
   }
}
```

In each iteration `i` from 1 to $n-1$, the `while` loop inserts item `a[i]` into the sorted subarray `a[0..i-1]`. The test `j > 0` of the `while` loop ensures that the program does not fall off the left end of array `a` during insertion.

For the sorting programs presented in this book, we will assume that the template type parameter `T` represents a pointer type. Nonetheless, function template `insertionSort` can be used to sort objects of any type that defines both the assignment operator = and a copy

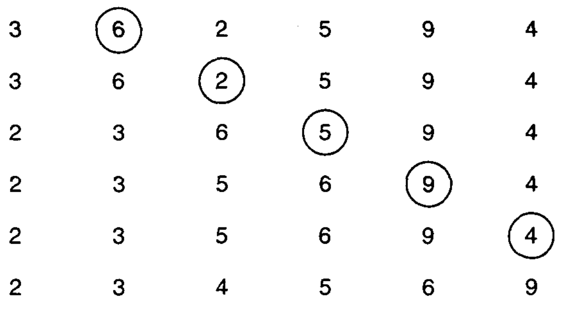


Figure 5.1: Insertion sorting an array of six integers. The next number to be inserted at each step is circled.

constructor. For example, the following code fragment reads 100 strings into array `s` and then sorts them using the standard C++ library function `strcmp` to compare two strings by dictionary order:

```
char buffer[80];
char *s[100];
for (int i = 0; i < 100 ; i++) {
   cin >> buffer;
   s[i] = new char[strlen(buffer)+1];
   strcpy(s[i], buffer);
}
insertionSort(s, 100, strcmp);
```

Note that this code fragment sorts an array of pointer-to-strings (array `s`), rather than the strings themselves. It is often more efficient to sort pointers instead of the objects pointed to. Unless the objects are small (4 bytes or less), a sorting program can move pointers around faster than the objects to which they point.

5.1.1 Analysis

To analyze insertion sort, it suffices to count the number of times the comparison function is called (assuming a comparison takes constant time). The running time of `insertionSort` is $T(n) = \sum_{i=1}^{n} I(i)$, where $I(i)$ time is needed to insert the ith item. Since $I(i)$ costs at most i comparisons, insertion sort requires $T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ comparisons, or about $n^2/2$ comparisons in the worst case. This worst-case behavior in fact occurs whenever the input array is initially sorted in reverse (decreasing) order.

On average, the ith item is compared to about $i/2$ items before its insertion position is found. Thus insertion sort performs about $n^2/4$ comparisons on average, twice as good as the worst case. If the input array is initially almost sorted, the program's expected running time is linear since the ith item is compared to only a constant number of items before reaching its position, on average. Hence insertion sort is a good way to sort an input array known to be almost sorted.

5.2 Finding Star-Shaped Polygons

A finite set of points in the plane can be connected by edges to form a polygon in different ways. Each such polygon is called a *polygonization* of the point set. In this section we devise a method for constructing *star-shaped* polygonizations. More simply, our method "connects the dots" (or points) to form star-shaped polygons.

5.2.1 What Are Star-Shaped Polygons?

Suppose points p and q lie in some polygon. We say that *p sees q* if the line segment $\overline{pq}$ lies in the polygon. Here we are imagining the boundary of polygon P to be composed of opaque walls, and its interior some transparent medium such as air. One point can see another only if no wall stands between them. Seeing is symmetric (if p sees q, then q sees p) but not transitive (if p sees q and q sees point r, it does not follow that p sees r) (Figure 5.2).

The set of those points in a polygon that see every point is called the *kernel* of the polygon. A polygon is said to be *star shaped* (Figure 5.3) if its kernel is nonempty. A polygon is *fan shaped* if its nonempty kernel contains one or more vertices (each such vertex is called an *apex* of the polygon). Every convex polygon is fan shaped since the kernel contains some vertex (in fact, every vertex). Every fan-shaped polygon is star shaped since its kernel is nonempty.

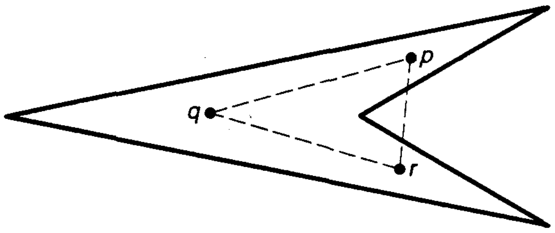


Figure 5.2: Points p and q see one another as do points q and r, yet points p and r do not see each other.

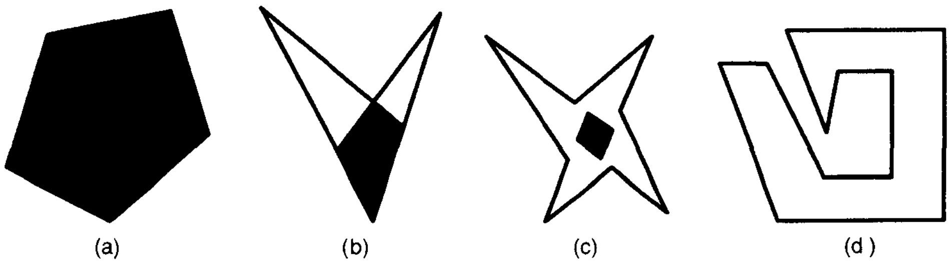


Figure 5.3: Polygons with darkened kernels: (a) a convex polygon; (b) a fan-shaped polygon that is not convex; (c) a star-shaped polygon that is not fan shaped; and (d) a polygon that is not star shaped.

5.2.2 Finding Star-Shaped Polygonizations

Given a set S of points $s_0, s_1, \ldots, s_{n-1}$ in the plane, the problem is to construct a star-shaped polygonization of set S. It is not difficult to see that there may exist more than one such polygon. We will specifically seek one whose kernel contains the first point s_0.

The algorithm works by iteratively constructing a *current polygon* over the points of S. Initially, the current polygon is the 1-gon s_0. In each iteration i from 1 to $n-1$, the next point s_i is inserted into the current polygon. At completion, the current polygon is the star-shaped polygon we seek.

To insert each new point s_i into the current polygon, we perform a clockwise traversal of the current polygon starting from vertex s_0. The traversal proceeds clockwise around the polygon boundary until arriving at the vertex which is to become s_i's successor; s_i is then inserted before this vertex. If the traversal proceeds full circle, returning to s_0, then s_i is inserted before s_0. Here s_0 serves as a *sentinel* which ensures that the traversal does not proceed too far. Figure 5.4 shows snapshots of the algorithm running on a small problem.

Function `starPolygon` is handed an array `s` of `n` points and returns a star-shaped polygon whose kernel contains point `s[0]`:

```
Point originPt;  // global: originPt = s[0]

Polygon *starPolygon(Point s[], int n)
{
   Polygon *p = new Polygon;
   p->insert(s[0]);
```

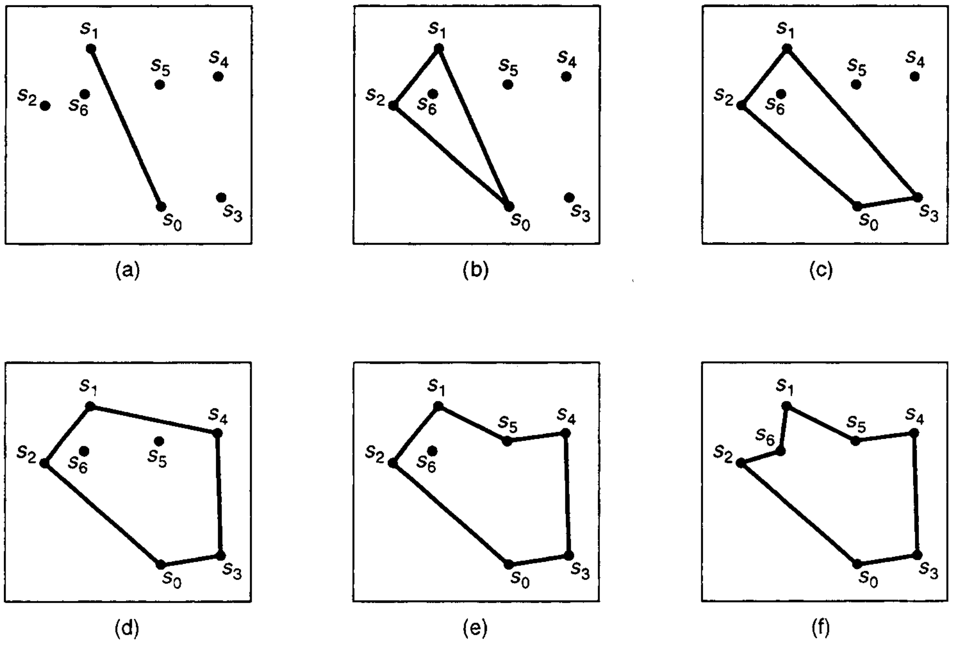


Figure 5.4: Finding a star-shaped polygon in a point set.

```
   Vertex *origin = p->v();
   originPt = origin->point();
   for (int i = 1; i < n; i++) {
      p->setV(origin);
      p->advance(CLOCKWISE);
      while (polarCmp(&s[i], p->v()) < 0)
         p->advance(CLOCKWISE);
      p->advance(COUNTER_CLOCKWISE);
      p->insert(s[i]);
   }
   return p;
}
```

In each iteration i, how do we determine where to insert point s_i along the boundary of the current polygon? We use the fact that the vertices of a star-shaped polygon are ordered radially around each point in its kernel. Since point s_0 is to lie in the kernel, we define a comparison function polarCmp based on the polar coordinates of points relative to point s_0 (i.e., where s_0 is regarded as the origin). Under this relation, point $p = (r_p, \theta_p)$ is considered less than point $q = (r_q, \theta_q)$ if (1) $\theta_p < \theta_q$ or (2) $\theta_p = \theta_q$ and $r_p < r_q$. With respect to this ordering, clockwise traversal of the current polygon proceeds from greater points to lesser points.

Comparison function polarCmp is passed two points p and q and compares them with respect to their radial ordering about point originPt, a global variable. It returns -1, 0, or 1 depending on whether its first argument p is less than, equal to, or greater than its second argument q:

```
int polarCmp(Point *p, Point *q)
{
   Point vp = *p - originPt;
   Point vq = *q - originPt;
   double pPolar = vp.polarAngle();
   double qPolar = vq.polarAngle();
   if (pPolar < qPolar) return -1;
   if (pPolar > qPolar) return 1;
   if (vp.length() < vq.length()) return -1;
   if (vp.length() > vq.length()) return 1;
   return 0;
}
```

Under function polarCmp, originPt is less than every other point in the plane. This is because function Point::polarAngle returns -1.0 if this point equals originPt, and returns a value in the range $[0, 360)$ otherwise. This fact allows point s[0] (=originPt) to serve as a sentinel. Function starPolygon runs in $O(n^2)$ time. Iteration i requires as many as i comparisons, and there are $n - 1$ iterations (the analysis parallels that of insertion sort).

The algorithm for finding star-shaped polygons closely parallels insertion sort. Both algorithms incrementally grow a current solution, represented by an ordering of items, into a complete solution. To insert each new item into the current ordering, both algorithms sequentially traverse the ordering from greatest to least until the item's proper position is reached. Furthermore, both algorithms run in quadratic time in the worst case.

5.3 Finding Convex Hulls: Insertion Hull

The algorithm we consider in this section—for finding the convex hull of a set of points—is more complicated than both insertion sort and our star-shaped polygonization algorithm. First, finding the proper position of each new item is more involved. Second, it is sometimes necessary to *remove* items from the current solution, so the current solution grows and shrinks as the algorithm proceeds.

5.3.1 What Are Convex Hulls?

Let S be a finite set of points in the plane. The convex hull of set S, denoted $\mathcal{CH}(S)$, equals the intersection of all convex polygons which contain S. Equivalently, $\mathcal{CH}(S)$ is the convex polygon of minimum area which contains all the points of S. Yet another equivalent definition states that $\mathcal{CH}(S)$ equals the union of all triangles determined by points of S.

Imagine the plane to be a sheet of wood with a nail protruding from every point in S. Now stretch a rubber band around all the nails and then release it, allowing it to snap taut against the nails. The taut rubber band conforms to the convex hull boundary. Figure 5.5 gives some examples.

Because they provide a way to approximate a point set or other nonconvex set by a convex region, convex hulls prove useful in a wide range of geometric applications. In pattern recognition, an unknown shape may be represented by its convex hull or by a hierarchy of convex hulls, which is then matched to a database of known shapes. As another example, motion planning, required when a moving robot must negotiate a landscape of obstacles, becomes much easier if the robot is approximated by its convex hull.

A useful scheme for classifying the points of a point set S refers to the convex hull $\mathcal{CH}(S)$. A point is a *boundary point* if it lies in the convex hull boundary, and an *interior point* if it lies in the convex hull interior. Those boundary points which form the

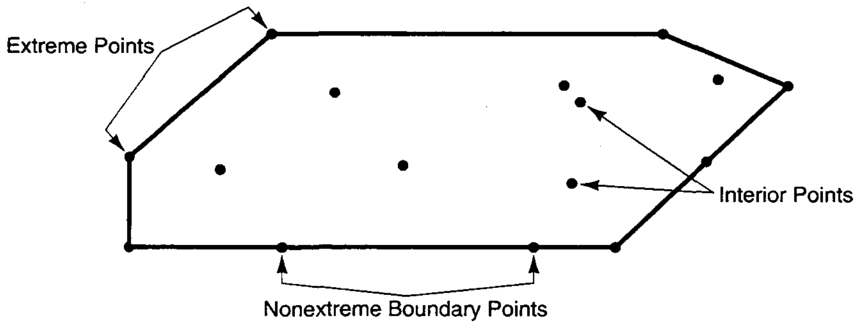


Figure 5.5: A finite set of points and its convex hull.

"corner" vertices of the convex hull are known as *extreme points*. Equivalently, a boundary point is extreme if it does not lie between any two other points of S. Figure 5.5 illustrates these notions. Note that this scheme for classifying points applies even if we are not interested in finding their convex hull per se.

5.3.2 Insertion Hull

Insertion hull, an incremental insertion approach to finding the convex hull of a finite set S of points, inserts a point at a time while maintaining the convex hull of those points inserted so far. We will refer to the convex hull built along the way as the *current hull*. Initially, the current hull consists of a single point of S; at completion, when all points have been inserted, the current hull equals $\mathcal{CH}(S)$ and we are done.

When a new point s is inserted into the current hull, one of two cases occurs. In the first case, s may lie in (the boundary or interior of) the current hull, in which case the current hull does not need to be updated.

In the second case, s lies outside the current hull, requiring that the current hull be modified as in Figure 5.6. Through point s can be drawn two *supporting lines*, each tangent to the current hull. (A line is a *supporting line* of a convex polygon P if the line passes through a vertex of P and the interior of P lies entirely to one side of the line.) The left (right) supporting line $\overrightarrow{s\ell}$ ($\overrightarrow{sr}$) passes through some vertex ℓ (r) of the current hull and lies to the left (right) of the current hull. If you were positioned at point s facing the convex hull, the left supporting line would appear to your left and the right supporting line to your right.

The two supporting vertices ℓ and r split the current hull boundary into two vertex chains: a *near chain* that is nearer point s and a *far chain* that is farther from s. (The near chain lies on the same side of line $\overline{\ell r}$ as s, and the far chain lies on the other side of $\overline{\ell r}$.) To update the current hull, we first find the two vertices ℓ and r which terminate the near and far chains. Then we remove the vertices of the near chain (except for vertices ℓ and r) and insert point s in their place.

The following program `insertionHull` returns the convex hull of the `n` points of array `s`:

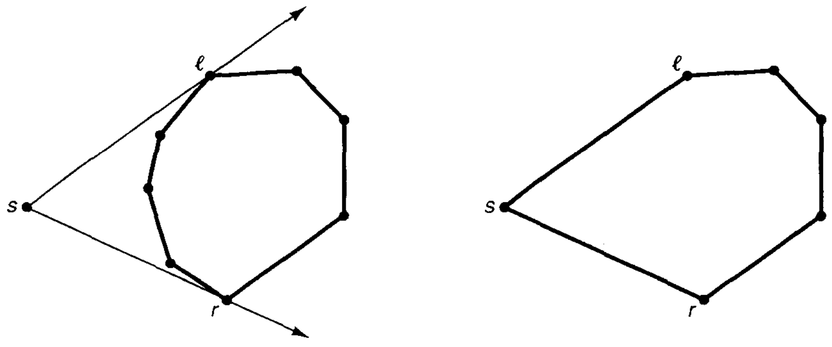


Figure 5.6: Inserting point s into the current hull.

```
Point somePoint;  // global

Polygon *insertionHull(Point s[], int n)
{
   Polygon *p = new Polygon;
   p->insert(s[0]);
   for (int i = 1; i < n; i++) {
      if (pointInConvexPolygon(s[i], *p))
         continue;
      somePoint = s[i];
      leastVertex(*p, closestToPolygonCmp);
      supportingLine(s[i], p, LEFT);
      Vertex *l = p->v();
      supportingLine(s[i], p, RIGHT);
      delete p->split(l);
      p->insert(s[i]);
   }
   return p;
}
```

In iteration `i`, point `s[i]` is inserted into the current hull `p`. The call to function `leastVertex` moves `p`'s window over the vertex that is closest to point `s[i]`. This prepares for the subsequent call to `supportingLine(s[i],p,LEFT)`, which moves the window over the vertex ℓ through which the left supporting line passes. The second call to `supportingLine` then moves the window over vertex r. The `split` operation is used to subdivide polygon `p` along the diagonal $\overline{\ell r}$, thereby separating the near chain from the far chain. The subpolygon consisting of the near chain is returned by `split` and deleted. Finally, point `s[i]` is inserted into polygon `p`, which, after `split` is performed, consists of the far chain.

Let us consider function `supportingLine`. To find the vertex ℓ through which the left supporting line passes, we start at some vertex of the near chain and then traverse clockwise around the current hull until arriving at the first vertex v whose successor is neither to the left of nor beyond directed line segment $\overrightarrow{sv}$. Vertex v is ℓ, the vertex we seek. Note why the process continues if the successor to v (i.e., vertex w) is *beyond* $\overrightarrow{sv}$: v cannot be an extreme point if it lies between s and w, so we must search further.

Function `supportingLine` is called with a polygon `p`, a point `s` outside `p`, and one of the enumeration values `LEFT` or `RIGHT` indicating which vertex (ℓ or r) is being sought. It assumes that the vertex in `p`'s window belongs to the near chain, which is why function `leastVertex` is called first. The function moves polygon `p`'s window over the vertex it finds (ℓ or r):

```
void supportingLine(Point &s, Polygon *p, int side)
{
   int rotation = (side == LEFT) ? CLOCKWISE : COUNTER_CLOCKWISE;
   Vertex *a = p->v();
   Vertex *b = p->neighbor(rotation);
   int c = b->classify(s, *a);
   while ((c == side) || (c == BEYOND) || (c == BETWEEN)) {
      p->advance(rotation);
      a = p->v();
      b = p->neighbor(rotation);
      c = b->classify(s, *a);
   }
}
```

Function `leastVertex`, which was defined in subsection 4.3.6, is used by program `insertionHull` to find the vertex of polygon `p` that is closest to the point stored in global variable `somePoint`. Comparison function `closestToPolygonCmp`, with which `leastVertex` is called, compares two points to decide which is closest to `somePoint`:

```
int closestToPolygonCmp(Point *a, Point *b)
{
   double distA = (somePoint - *a).length();
   double distB = (somePoint - *b).length();
   if (distA < distB) return -1;
   else if (distA > distB) return 1;
   return 0;
}
```

5.3.3 Analysis

As it proceeds, program `insertionHull` may build large current hulls which are disassembled by the time the program finishes. Consider the situation shown in Figure 5.7a, in which all the points except p, q, and r have been inserted. Each of the last three insertions removes a chain of vertices until only a triangular hull remains (Figures 5.7b – d). Clearly, had p, q, and r been inserted *first*, before the other points, the triangular hull would be constructed early and insertion of each remaining point would be faster, involving only the determination that the point lies in the triangle. Thus the order in which points are inserted affects efficiency.

Nonetheless, the cost of building the convex hull is in fact *not* dominated by the operations `insert` and `split` used to assemble and disassemble the current hulls. After all, every point can be inserted at most once and removed at most once. It follows that the total cost for all `insert` and `split` operations over the course of the algorithm is bounded above by $O(n)$.

Likewise, the calls to `supportingLine` are relatively inexpensive: The two calls to `supportingLine` performed in an iteration together take time proportional to the length of the near chain, and this work can be charged to the vertices of the near chain, which are then removed in the same iteration. Since a vertex can be removed at most once, the cost for all calls to `supportingLine` over the course of the algorithm is bounded above by $O(n)$.

It turns out that `insertionHull` spends most of its time executing `pointInConvexPolygon` and `leastVertex`. To process the ith point s_i, the call to each of the two functions takes time proportional to i in the worst case (when the convex

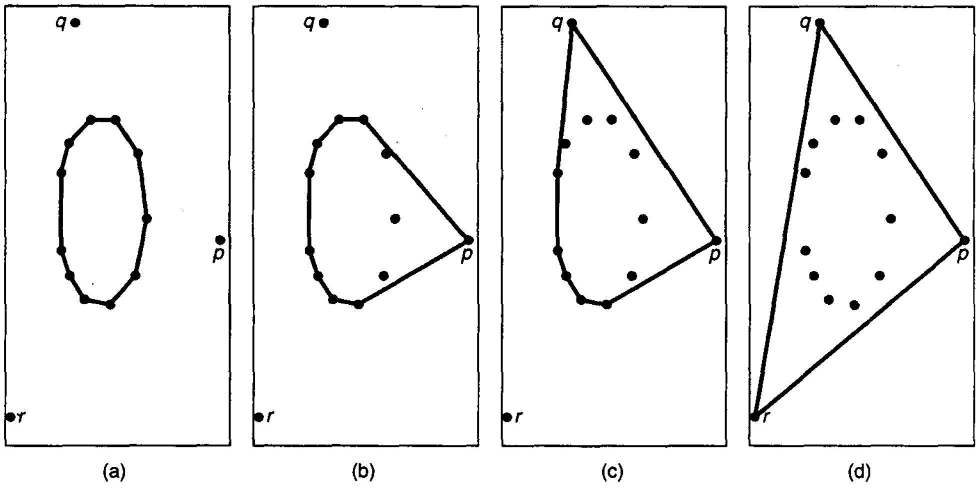


Figure 5.7: Points p, q, and r are inserted last.

hull possesses i vertices). This case occurs *for every* point s_i if they are all extreme points because the current hull will then grow by one vertex per insertion. Hence `insertionHull` runs in $O(n^2)$ time in the worst case.

Later in this book we will cover two algorithms—*Graham scan* and *merge hull*—that compute the convex hull of n points in optimal $O(n \log n)$ time.

5.4 Point Enclosure: The Ray-Shooting Method

In Chapter 4 we devised a simple algorithm for solving the point enclosure problem for convex polygons: The algorithm decides whether a given point a lies inside, outside, or on the boundary of a convex polygon p. It works by testing point a against each edge of p in turn; if point a lies on the wrong side of some edge, the point has been shown to lie outside polygon p; otherwise a has been shown to belong to p. The algorithm takes advantage of the fact that the interior of a convex polygon lies entirely to one side of every edge—thus a point which lies on the wrong side of some edge cannot lie in the polygon interior. The algorithm, however, does not correctly solve the more general point enclosure problem, which allows arbitrary (convex or nonconvex) polygons. In Figure 5.8, for example, the interior of the polygon straddles both sides of the edge labeled e; since point a lies on the "wrong" side of e, the algorithm mistakenly reports that a lies outside polygon.

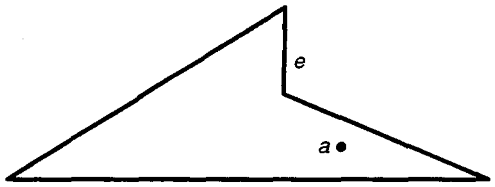


Figure 5.8: How does one determine whether point a lies in the polygon?

The problem of point enclosure relative to a *convex* polygon is like deciding whether an unsorted list of numbers contains only numbers greater than or equal to zero. To solve the problem, we step through the list until reaching some negative number, at which time we report "no"; if none of the numbers turns out to be negative, we report "yes." The answer is yes only if every one of a set of distinct conditions holds true.

Point enclosure relative to an *arbitrary* polygon, on the other hand, is more like the problem of deciding whether the sum of an unsorted list of numbers is greater than or equal to zero. The problem cannot be decided until *all* the numbers have been added together. Adding just some of the numbers, or even all but one of them, cannot solve the problem since the remaining number may change everything. In the same manner, partial examination of a polygon may suggest that it does not contain some distant point, yet it may happen that the last several edges to be examined form a "finger" that protrudes far from the rest of the polygon, capturing the point. Deciding whether a point lies in an arbitrary polygon involves a single condition encompassing the polygon as a whole.

In this section we present the ray-shooting method for solving the point enclosure problem for arbitrary polygons. Imagine doing this to decide point enclosure for point a and polygon p: Starting from some point far from the polygon, move in a straight line toward a. Along the way we cross the polygon boundary zero or more times: the first time crossing into the polygon, the second time crossing back out, the third time crossing back in once again, and so forth, until arriving at a. In general, every odd-numbered crossing carries us into polygon p, and every even-numbered crossing carries us back out of p. If we arrive at a having undergone an odd number of crossings, a lies inside p; and if an even number of crossings, a lies outside p. For example, in Figure 5.9, ray $\overrightarrow{r_a}$ crosses the boundary once; since one is odd, a lies inside the polygon. We can conclude that point b lies outside the polygon since ray $\overrightarrow{r_b}$ crosses the boundary an even number of times (twice).

Transforming this idea into an algorithm turns on two key observations. First, *any* ray that originates at the point a to be classified will do (Figure 5.9). Being free to work with any ray originating at a, we can, for simplicity, work with the *right horizontal ray* $\overrightarrow{r_a}$ originating at a (the unique ray starting at a and directed parallel to the positive x-axis).

The second key observation is that the order of boundary crossings along ray $\overrightarrow{r_a}$ is irrelevant; all that matters is the parity (oddness or evenness) of their total number. Therefore, rather than simulate moving along ray $\overrightarrow{r_a}$, it is enough for the algorithm to

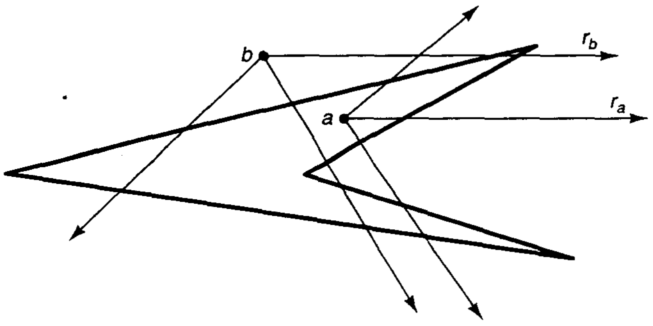


Figure 5.9: Every ray originating at a crosses the boundary an odd number of times, and every ray originating at b crosses an even number of times.

detect all edge crossings *in any order*, updating parity along the way. The easiest way to do this is to traverse the polygon boundary, toggling a parity bit whenever we visit an edge which ray $\overrightarrow{r_a}$ crosses.

Relative to the right horizontal ray $\overrightarrow{r_a}$, we distinguish three types of polygon edges: *touching edges*, which contain point a; *crossing edges*, which do not contain point a but which ray $\overrightarrow{r_a}$ crosses; and *inessential edges*, which ray $\overrightarrow{r_a}$ does not meet at all. For example, in Figure 5.10, edge c is a crossing edge, edge d is a touching edge, and edge e is an inessential edge.

Function `pointInPolygon` solves the point enclosure problem for point `a` and polygon `p`. The algorithm traverses the boundary of the polygon while toggling variable `parity` for each crossing edge it encounters. It returns the enumeration value `INSIDE` if the final value of `parity` is 1 (indicating odd), and `OUTSIDE` if its final value is 0 (indicating even). If a touching edge is discovered, the algorithm immediately returns the enumeration value `BOUNDARY`.

```
enum { INSIDE, OUTSIDE, BOUNDARY };      // point classifications
enum { TOUCHING, CROSSING, INESSENTIAL };// edge classifications

int pointInPolygon(Point &a, Polygon &p)
{
   int parity = 0;
   for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE)) {
      Edge e = p.edge();
      switch (edgeType(a, e)) {
       case TOUCHING:
         return BOUNDARY;
       case CROSSING:
         parity = 1 - parity;
      }
   }
   return (parity ? INSIDE: OUTSIDE);
}
```

Function call `edgeType(a,e)` classifies edge `e` with respect to right horizontal ray $\overrightarrow{r_a}$, returning one of the enumeration values `TOUCHING`, `CROSSING`, or `INESSENTIAL`. Definition of `edgeType` is somewhat tricky because function `pointInPolygon` must correctly handle the special cases that arise when ray $\overrightarrow{r_a}$ pierces vertices. Consider Figure 5.11. In case (a) the parity should be toggled—the ray crosses the boundary only once

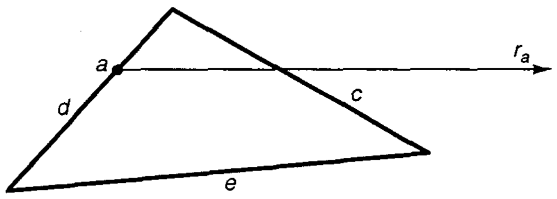


Figure 5.10: Edge c is a crossing edge, edge d a touching edge, and edge e an inessential edge.

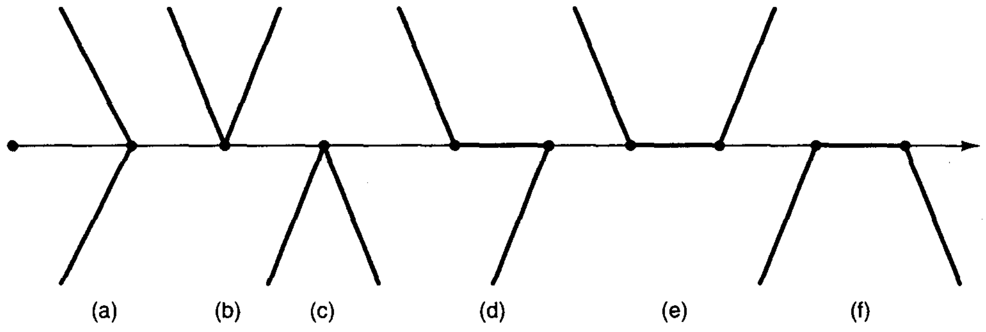


Figure 5.11: Special cases: `parity` is toggled one time in cases (a) and (d), zero times in cases (b) and (e), and two times in cases (c) and (f).

even though, in doing so, it crosses two edges. In cases (b) and (c) the parity should not be changed. This can be achieved by polishing our scheme for classifying edges as follows:

- Edge e is a *touching* edge if e contains point a.
- Edge e is a *crossing* edge if (1) e is not horizontal and (2) ray $\overrightarrow{r_a}$ crosses e at some point other than e's lower endpoint.
- Edge e is an *inessential* edge if e is neither a crossing nor a touching edge.

Referring to Figure 5.11, we see that in case (a), variable `parity` is toggled once; in case (b), `parity` is not changed; and in case (c), `parity` is toggled twice with the net effect of remaining unchanged. Note that horizontal edges not containing point a are considered inessential and so are ignored by function `pointInPolygon`. Therefore, cases (d), (e), and (f) are handled the same as cases (a), (b), and (c), respectively.

Function `edgeType` classifies edge `e` as `CROSSING`, `TOUCHING`, or `INESSENTIAL` with respect to point `a`:

```
int edgeType(Point &a, Edge &e)
{
   Point v = e.org;
   Point w = e.dest;
   switch (a.classify(e)) {
    case LEFT:
      return ((v.y<a.y) && (a.y<=w.x)) ? CROSSING : INESSENTIAL;
    case RIGHT:
      return ((w.y<a.y) && (a.y<=v.y)) ? CROSSING : INESSENTIAL;
    case BETWEEN:
    case ORIGIN:
    case DESTINATION:
      return TOUCHING;
    default:
      return INESSENTIAL;
   }
}
```

Note how function `edgeType` detects crossing edges. If point `a` lies to the left of edge `e`, the edge is a crossing edge only if `v` (=`e.org`) lies below ray $\overrightarrow{r_a}$ and `w` (=`e.dest`) lies on or above the ray. For then the edge cannot be horizontal, and ray $\overrightarrow{r_a}$ must cross the edge at some point other than its lower endpoint. Alternatively, if `a` lies to the right of edge `e`, the roles of `v` and `w` are interchanged.

Program `pointInPolygon` runs in time proportional to the size of the polygon in the worst case (when point `a` does not lie on the polygon boundary).

5.5 Point Enclosure: The Signed Angle Method

Let us consider another approach to the point enclosure problem. This approach requires the notion of a *signed angle*. Given directed segment $\overrightarrow{bc}$ and some point a, suppose that the angle between vectors $\overrightarrow{ab}$ and $\overrightarrow{ac}$ measures θ. The *signed angle* at point a relative to $\overrightarrow{bc}$ then measures θ if c lies to the left of or is collinear with $\overrightarrow{ab}$, and $-\theta$ if c lies to the right of $\overrightarrow{ab}$. Note that the signed angle and the orientation of triangle Δabc have the same sign.

We can extend the definition of signed angle to vertex chains. The signed angle at point a relative to a vertex chain is the sum of the signed angles at a relative to the chain's edges. Figure 5.12 gives some examples.

Function `signedAngle` computes and returns the signed angle at point `a` relative to edge `e`. After first treating the cases in which `a` is collinear with `e`, the function distinguishes between the configurations shown in Figure 4.6:

```
double signedAngle(Point &a, Edge &e)
{
   Point v = e.org - a;
   Point w = e.dest - a;
   double va = v.polarAngle();
   double wa = w.polarAngle();
   if ((va == -1.0) || (wa == -1.0))
      return 180.0;
   double x = wa - va;
   if ((x == 180.0) || (x == -180.0))
```

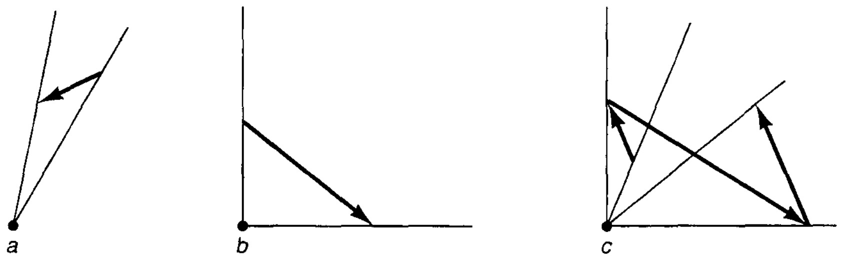


Figure 5.12: (a) The signed angle at point a is 20; (b) the signed angle at b is -90; and (c) the signed angle at c is $20 - 90 + 40 = -30$.

```
      return 180.0;
   else if (x < -180.0)
      return (x + 360.0);
   else if (x > 180.0)
      return (x - 360.0);
   else
      return x;
}
```

Let us consider how signed angles are used to classify a given point a with respect to a given polygon p. Assume that a does not lie in the boundary of p. Let A denote the signed angle at a relative to the boundary of p, where p has clockwise sense of rotation. A is useful for classifying the point: $A = -360$ degrees if a is inside the polygon, and $A = 0$ if a is outside the polygon. It is easy to see why this is true in the case of convex polygons. If a is inside convex polygon p, the boundary of p encircles a a full 360 degrees. Alternatively, if a is outside p, the boundary of p can be split into a near chain and a far chain, relative to point a. (Near and far chains were defined in Section 5.3.) Where A_n denotes the signed angle at p relative to the near chain and A_f the signed angle relative to the far chain, we have $A_n = -A_f$, from which it follows that $A = A_n + A_f = 0$.

That this also holds for *nonconvex* polygons is less obvious. Suppose first that a lies outside the polygon. Imagine casting a ray from point a through every vertex of the polygon, thereby partitioning the polygon into a number of triangles and convex quadrilaterals $p_1, p_2, \ldots, p_k$ (Figure 5.13a). Since a lies outside each p_i and each p_i is convex, the signed angle A_i at a relative to the boundary of p_i is zero (i.e., $A_i = 0$). But $A = A_1 + \cdots + A_k$ since the summation counts every edge of the original polygon p exactly once. Observe that new edges introduced by the rays contribute zero to A. It follows that $A = 0$ if point a is outside the polygon.

Figure 5.13b illustrates why $A = -360$ if a is inside polygon p. As before, imagine partitioning the polygon by rays originating from a, but this time preserve a small convex polygonal neighborhood around a (denoted p_0 in the figure). Since p_0 is convex and contains a, we know that $A_0 = -360$. Furthermore, since a lies outside the remaining

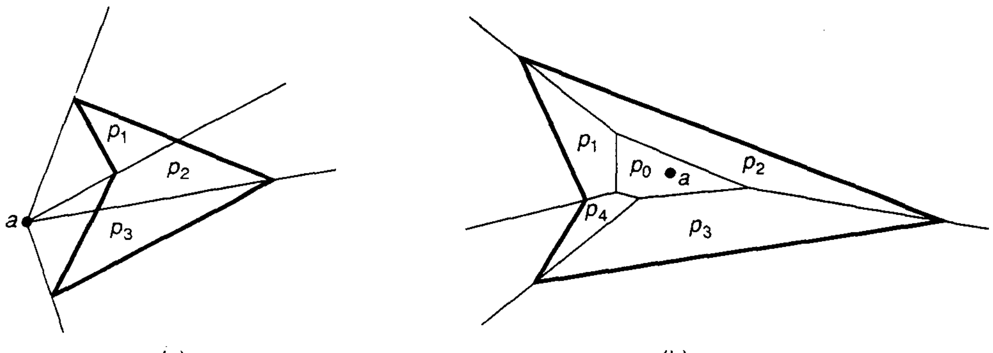


Figure 5.13: (a) Point a is outside the polygon, so $A = A_1 + A_2 + A_3 = 0 + 0 + 0 = 0$; and (b) point b is inside the polygon, so $A = A_0 + A_1 + A_2 + A_3 + A_4 = -360 + 0 + 0 + 0 + 0 = -360$.

(convex) polygons p_i, we have $A_i = 0$ for each $i = 1, 2, \ldots, k$. It follows that $A = A_0 + A_1 + \cdots + A_k = -360 + 0 + \cdots + 0 = -360$.

Function `pointInPolygon2` solves the point enclosure problem for point `a` and polygon `p`. The signed angle at `a` relative to the polygon is accumulated in `total` as each edge is visited in turn. If `a` is found to lie on some edge (the signed angle relative to the edge equals 180), the function immediately returns the enumeration value `BOUNDARY`. Otherwise, when all polygon edges have been processed, it returns `INSIDE` or `OUTSIDE` depending on the final value of `total`.

```
int pointInPolygon2(Point &a, Polygon &p)
{
   double total = 0.0;
   for (int i = 0; i < p.size(); i++, p.advance(CLOCKWISE)) {
      Edge e = p.edge();
      double x = signedAngle(a, e);
      if (x == 180.0)
         return BOUNDARY;
      total += x;
   }
   return ((total < -180.0) ? INSIDE: OUTSIDE);
}
```

Program `pointInPolygon2` runs in time linear in the size of the polygon in the worst case.

5.6 Line Clipping: The Cyrus-Beck Algorithm

The process of discarding that portion of a geometric object that lies outside a given region is called *clipping*. Clipping is used for many purposes in computer graphics. In a windowing system, a window may serve as a small aperture into a panorama that extends far beyond. When drawing into the panorama, it is necessary to clip away those portions of objects that do not fall under the purview of the window. In some text editors, it is necessary to clip characters that do not fit on a line. In three-dimensional graphics, objects in space are clipped to a volume before being projected into the image plane, to avoid wasting time projecting things that will not be seen anyway.

In this section we present the Cyrus-Beck algorithm for clipping a line segment to a convex polygon. In the subsequent section we will cover the Sutherland-Hodgman algorithm for clipping an arbitrary polygon to a convex polygon.

Let s be a line segment and let p be a convex n-gon to which s is to be clipped. Here p is called a *clip polygon* and s the *subject*. We seek $s \cap p$, that portion of s which lies in p.

Let $\overrightarrow{s}$ denote one of the two directed infinite lines determined by s. Suppose we extend each of the n edges of polygon p to infinity, in both directions. Line $\overrightarrow{s}$ crosses all these extended edges of p in no more than n distinct intersection points. (If $\overrightarrow{s}$ is parallel to or collinear with some edge of p, the edge does not contribute an intersection point; moreover, if $\overrightarrow{s}$ passes through a vertex of polygon p, two intersection points coincide.)