Elementary Geometric Methods

24

Computers are being used more and more to solve large-scale problems that are inherently geometric. Geometric objects such as points, lines and polygons are the basis of a broad variety of important applications and give rise to an interesting set of problems and algorithms.

Geometric algorithms are important in design and analysis systems modeling physical objects ranging from buildings and automobiles to very large-scale integrated circuits. A designer working with a physical object has a geometric intuition that is difficult to support in a computer representation. Many other applications directly involve processing geometric data. For example, a political "gerrymandering" scheme to divide a district up into areas of equal population (and that satisfy other criteria such as putting most of the members of the other party in one area) is a sophisticated geometric algorithm. Other applications abound in mathematics and statistics, fields in which many types of problems can be naturally set in a geometric representation.

Most of the algorithms we've studied have involved text and numbers, which are represented and processed naturally in most programming environments. Indeed, the primitive operations required are implemented in the hardware of most computer systems. We'll see that the situation is different for geometric problems: even the most elementary operations on points and lines can be computationally challenging.

Geometric problems are easy to visualize, but that can be a liability. Many problems that can be solved instantly by a person looking at a piece of paper (example: is a given point inside a given polygon?) require non-trivial computer programs. For more complicated problems, as in many other applications, the method of solution appropriate for computer implementation may well be quite different from the method of solution appropriate for a person.

One might suspect that geometric algorithms would have a long history because of the constructive nature of ancient geometry and because useful applications are so widespread, but actually much of the work in the field has been quite recent. Nonetheless, the work of ancient mathematicians is often useful in the development of algorithms for modem computers. The field of geometric algorithms is interesting to study because of its strong historical context, because new fundamental algorithms are still being developed, and because many important large-scale applications require these algorithms.

Points, Lines, and Polygons

Most of the programs we'll study operate on simple geometric objects defined in a two-dimensional space, though we will consider a few algorithms for higher dimensions. The fundamental object is a *point,* which we consider to be a pair of integers-the "coordinates" of the point in the usual Cartesian system. A *line* is a pair of points, which we assume are connected together by a straight line segment. *A polygon* is a list of points: we assume that successive points are connected by lines and that the first point is connected to the last to make a closed figure.

To work with these geometric objects, we need to decide how to represent them. Usually we use an array representation for polygons, though a linked list or some other representation can be used when appropriate. Most of our programs will use the straightforward representations

```
---------------------------------------------------------------
  type point  =  record x,y: integer end;
         line  =  record p1, p2: point end;
  var polygon:  array [O..Nmax ] of point;
---------------------------------------------------------------
```

Note that points are restricted to have integer coordinates. A real representation could also be used. Using integer coordinates leads to slightly simpler and more efficient algorithms, and is not as severe a restriction as it might seem. As mentioned in Chapter 2, working with integers when possible can be a very significant timesaver in many computing environments, because integer calculations are typically much more efficient than floating-point calculations. Thus, when we can get by with dealing only with integers without introducing much extra complication, we will do so.

More complicated geometric objects will be represented in terms of these basic components. For example, polygons will be represented as arrays of *points.* Note that using arrays of *lines* would result in each point on the polygon being included twice (though that still might be the natural representation for some algorithms). Also, it is

useful in some applications to include extra information associated with each point or line; we can do this by adding an *info* field to the records.
We'll use the sets of points shown in Figure 24.1 to illustrate the operation of several geometric algorithms. The sixteen points on the left are labeled with single letters for reference in explaining the examples, and have the integer coordinates

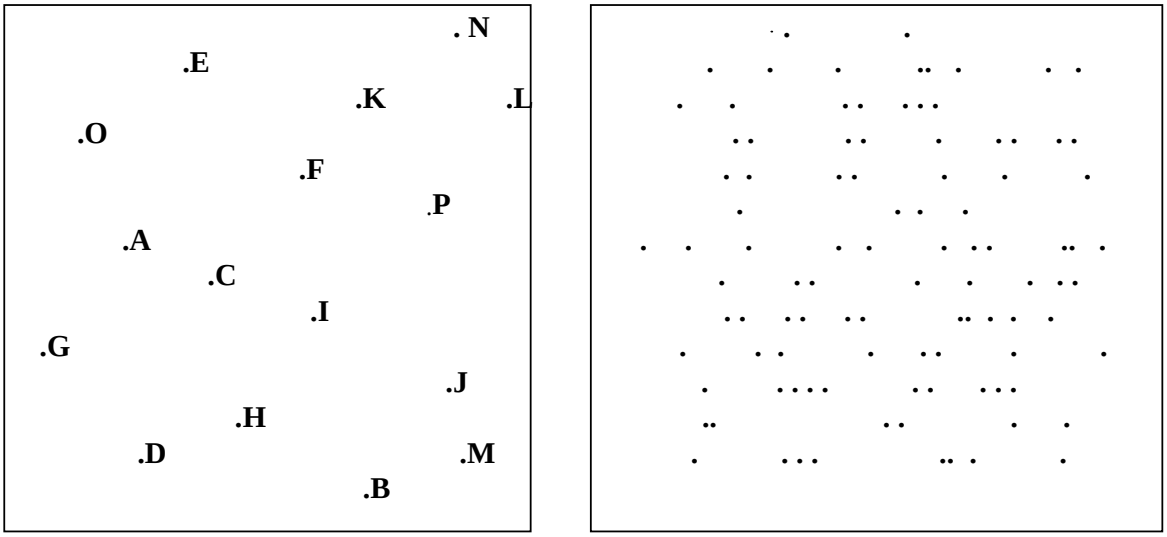

Figure 24.1 Sample point sets for geometric algorithms

shown in Figure 24.2. (The labels we use are assigned in the order in which the points are assumed to appear in the input.) The programs usually have no reason to refer to points "by name"; they are simply stored in an array and are referred to by index. The order in which the points appear in the array may be important in some of the programs: indeed, it is the goal of some geometric algorithms to "sort" the points into some particular order. On the right in Figure 24.1 are 128 points, randomly generated with integer coordinates between 0 and 1000.
A typical program maintains an **array** p [1..N] of points and simply reads in N pairs of integers, assigning the first pair to the x and y coordinates of *p*[1], the second pair to *p* [*2*], etc. When p represents a polygon, it is sometimes convenient to maintain "sentinel" values p[0l=p[N] and p[N+1]=p[1].

Line Segment Intersection

As our first elementary geometric problem, we'll consider determining whether or not two given line segments intersect. Figure 24.3 illustrates some of the situations that can arise. In the first case, the line segments intersect. In the second, the endpoint of one segment is on the other segment: We'll consider this an intersection

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0	P
x	3	11	6	4	5	8	1	7	9	14	10	16	15	13	3	12
y	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

Figure 24.2 Coordinates of points in small sample set (on the left in Figure 24.1).

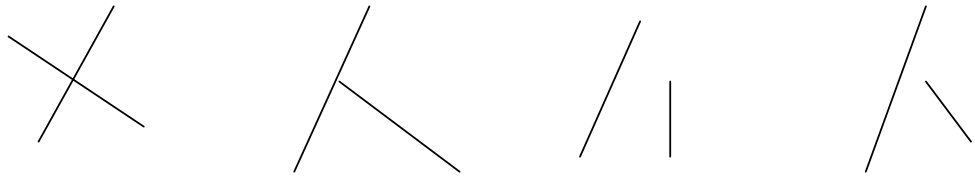

Figure 24.3 Testing whether line segments intersect: four cases

by assuming the segments to be "closed" (endpoints are part of the segments); thus, line segments having a common endpoint intersect. In both the last two cases in Figure 24.3, the segments do not intersect, but the cases differ when we consider the intersection point of the lines defined by the segments. In the fourth case this intersection point falls on one of the segments; in the third it does not. Or, the lines could be parallel (a special case of this that frequently turns up is when one or both of the segments is a single point).

The straightforward way to solve this problem is to find the intersection point of the lines defined by the line segments and then check whether this intersection point falls between the endpoints of both of the segments. Another easy method is based on a tool that we'll find useful later, so we'll consider it in more detail. Given three points, we want to know whether, in traveling from the first to the second to the third, we turn counterclockwise or clockwise. For example, for points A, B, and C in Figure 24.1 the answer is yes, but for points A, B, and D the answer is no. This function is straightforward to compute from the equations for the lines as follows:

```
function ccw (pO,pl.p2: point): integer;
     var dxl , dx2 , dyl dy2: integer;
     begin
    dxl := pl.x - p0.x;   dyl := pl.y – p0.y;
    dx2:= p2.x – p0.x; dy2 := p2.y – p0.y;
    if dxl * dy2 > dyl * dx2  then ccw := l;
    if dxl * dy2 < dyl * dx2  then ccw := -l;
    if dxl * dy2 = dyl * dx2  then
       begin
      if (dxl * dx2 < 0) or (dyl * dy2 < 0) then ccw := -l else
      if (dxl * dxl + dyl * dyl) >= (dx2 * dx2 + dy2 * dy2) then ccw := O  else  ccw := 1;
    end;
  end;
```

To understand how the program works, first suppose that all of the quantities dxl,. *dx2, dyl,* and *dy2* are positive. Then note that the slope of the line connecting *p0* and pl is *dyl/dx1* and the slope of the line connecting p0 and p2 is *dy2ldx2.* Now, if the slope of the second line is greater than the slope of the first, a "left" (counterclockwise) turn is required in the journey from p0 *to* pl *to p2;* if less, a "right" (clockwise) turn is required. Comparing slopes in the program is slightly inconvenient because the lines could be vertical (dxl *or* dx2 could be 0): we multiply by *dxl*dx2* to avoid this. It turns out that the slopes need not be positive for this test to work properly; if, however, the slopes are the same (the three points are collinear), one can envision a variety of ways to define *ccw.* Our choice is to make the function three-valued: rather than **true** and **false** we use 1 and -1, reserving the value 0 for the case where p2 *is on* the line segment *between p0* and *p1***.** If the points are collinear, and p0 is between p2 and pl, we take *ccw* to be -1; if p2 is between p0 and pl, we take *ccw* to be 0; and if pl is between p0 and p2, we take *ccw* to be 1. We'll see that this convention simplifies the coding for functions that use *ccw* in this and the next chapter.

This immediately gives an implementation of the *intersect* function. If both endpoints of each line are on different "sides" (have different ccw values) of the other, then the lines must intersect:

```
function intersect(l1. l2: line): boolean;
  begin
  intersect := ((ccw(l1.pl, ll.p2. 12.pl) * ccw(Il.pl. 1l.p2, 12.p2)) <= 0)
            and ((ccw(12.pl, 12.p2. 1l.pl) * ccw(12.pl, 12.p2, 11.p2)) <= 0);
end;
```

This solution seems to involve a fair amount of computation for such a simple problem. The reader is encouraged to try to find a simpler solution, but should be warned to be sure that the solution works on all cases. For example, if all four points are collinear, there are six different cases (not counting situations where points coincide), only

four of which are intersections. Special cases like this are the bane of geometric algorithms: they cannot be avoided, but we can lessen their impact with primitives like *ccw*.

If many lines are involved, the situation becomes much more complicated. In Chapter 27, we'll see a sophisticated algorithm for determining whether any two of a set of N lines intersect.

Simple Closed Path

To get the flavor of problems dealing with sets of points, let's consider the problem of finding a path through a set of N given points that doesn't intersect itself, visits all the points, and returns to the point at which it started. Such a path is called a *simple closed path.* One can imagine many applications for this: the points might represent homes and the path the route that a mailman might take to get to each of the homes without crossing his path. Or we might simply want a reasonable way to draw the points using a mechanical plotter. This problem is elementary because it asks only for any closed path connecting the points. The problem of finding the best such path, called the *traveling salesman problem,* is much, much more difficult, and we'll look at it in some detail in the last few chapters of this book. In the next chapter, we'll consider a related but much easier problem: finding the shortest path that surrounds a set of *N* given points. In Chapter 31, we'll see how to find the best way to "connect" a set of points.

An easy way to solve the elementary problem at **hand** is the following. Pick one of the points to serve as an "anchor." Then compute the angle made by drawing a line from each of the points in the set to the anchor and then out in the positive horizontal direction (this is part of the polar coordinate of each point with the anchor point as origin). Next, sort the points according to that angle. Finally, connect adjacent points. The result is a simple closed path connecting the points, as shown in Figure 24.4 for the points in Figure 24. 1. In the small set of points, B is used as the anchor: if the points are visited in the order

B M J L N P K F I E C 0 A H G D B

then a simple closed polygon will be traced Out.

If *dx* and *dy* are the distances along the *x* and y axes from the anchor point to some other point, then the angle needed in this algorithm is $\tan^{-1}$ dy/dx. Although the arctangent is a built-in function in Pascal (and some other programming environments). it is likely to be slow and leads to at least two annoying extra conditions to compute: whether dx is zero and which quadrant the point is in. Since the angle

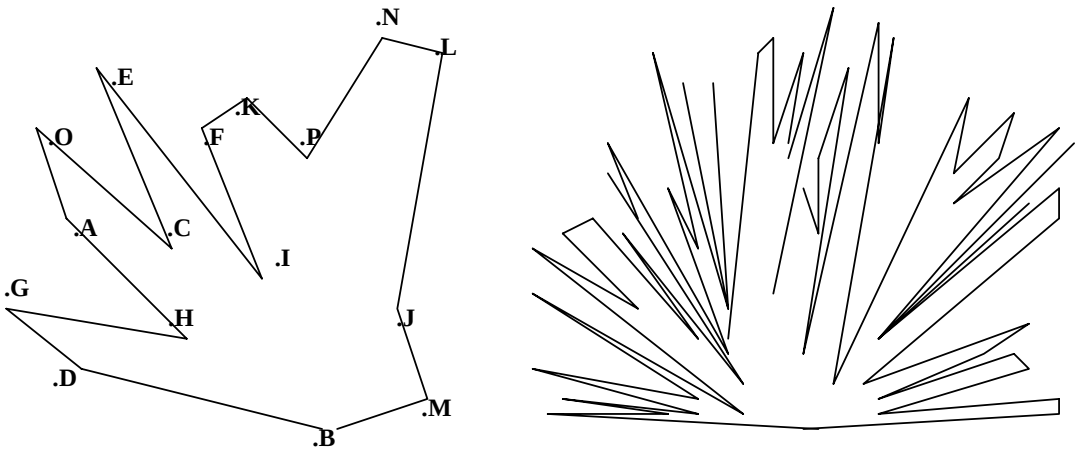

Figure 24.4 Simple closed paths

is used only for the sort in this algorithm, it makes sense to use a function that is much easier to compute but has the same ordering properties as the arctangent (so that when we sort, we get the same result). A good candidate for such a function is simply dy / (dy + dx). Testing for exceptional conditions is still necessary, but simpler. The following program returns a number between 0 and 360 that is *not* the angle made by pl and p2 with the horizontal but which has the same order properties as that angle.

```
-----------------------------------------------------------------------
function theta(pl.  p2: point): real;
    var dx, dy. ax. ay : integer;
            t: real;
begin
dx := p2.x -  pl.x; ax := abs(dx);
dy := p2.y - pl.y; ay := abs(dy);
if (dx = 0) and (dy = 0) then t := 0
   else t := dy/(ax + ay);
if dx < 0 then t := 2 - t
  else if dy < 0 then t := 4 + t;
theta := t * 90.0;
end;
-----------------------------------------------------------------------
```

In some programming environments it may not be worthwhile to use such programs instead of standard trigonometric functions; in others it may lead to significant savings. (In some cases it may be worthwhile to change *theta* to have an integer value, to avoid using real numbers entirely.)

Inclusion in a Polygon

The next problem we'll consider is a natural one: given a point and polygon represented as an array of points, determine whether the point is inside or outside the polygon. A straightforward solution to this problem immediately suggests itself: draw a long line segment from the point in any direction (long enough so that its other endpoint is guaranteed to be outside the polygon) and count the number of lines from the polygon that it crosses. If the number is odd, the point must be inside; if it is even, the point is outside. This is easily seen by tracing what happens as we come in from the endpoint on the outside: after the first line, we are inside, after the second we are outside, etc. If we do this an even number of times, the point at which we end up (the original point) must be outside.

The situation is not quite so simple, however, because some intersections might occur right at the vertices of the input polygon. Figure 24.5 shows some of the situations that must to be handled. The first is a straightforward "outside" case; the second is a straightforward "inside" case; in the third, the test line leaves the

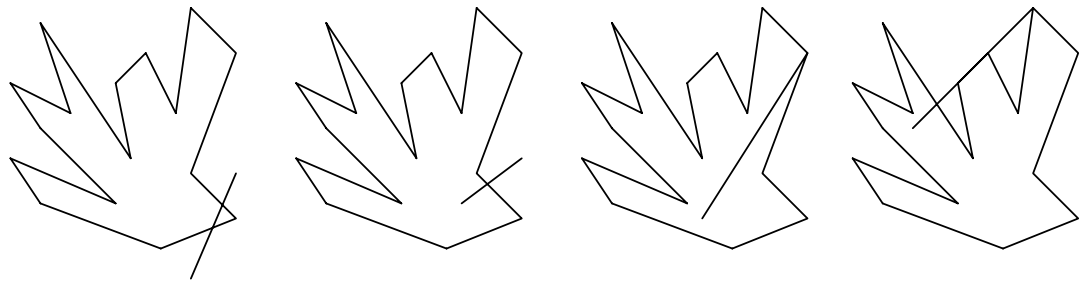

Figure 24.5 Cases to be handled by a point –in-polygon algorithm

polygon at a vertex (after touching two other vertices; and in the fourth, the test line coincides with an edge of the polygon before leaving. In some cases where the test line intersects a vertex it should count as one intersection with the polygon; in other cases it should count as none (or two). The reader may be amused to try to find a simple test to distinguish these cases before reading further.

The need to handle cases where polygon vertices fall on the test lines forces us to do more than just count the line segments in the polygon intersecting the test line. Essentially, we want to travel around the polygon, incrementing an intersection counter whenever we go from one side of the test line to another. One way to implement this is to simply ignore points that fall on the test line, as in the following program:

```
function inside (t: point): boolean;
    var count,  i, j: integer;
          lt. lp: line;
    begin
    count  i := O;  j:=O;

    p [0] := p[N]; p [N+l ] := p [1];
    lt.pl := t; lt.p2 := t;  lt.p2.x:=maxint;
    for i:=l to N do
      begin
      lp.pl := p[i]; lp.p2 := p[i];
      if not intersect(1p, lt) then
          begin
          lp.p2 := p[j];  j := i;
          if intersect(lp, lt) then count := count + 1;
          end;
      end;
    inside := ((count mod 2)=1);
    end;
```

This program uses a horizontal test line for ease of calculation (imagine the diagrams in Figure 24.5 as rotated 45 degrees). The variable j is maintained as the index of the last point on the polygon known not to lie on the test line. The program assumes that p [1] is the point with the smallest x coordinate among all the points with the smallest y coordinate, so that if p [1] is on the test line, then p [0] cannot be. The same polygon can be represented by N different p arrays, but as this illustrates it is sometimes convenient to fix a standard rule for p[1]. (For example, this same rule is useful for p [1] as the "anchor" for the procedure suggested above for computing a simple closed polygon.) If the next point on the polygon that is not on the test line is on the same side of the test line as the jth point, then we need not increment the intersection counter (count); otherwise we have an intersection. The reader may wish to check that this algorithm works properly for the cases in Figure 24.5.

If the polygon has only three or four sides, as is true in many applications, then such a complex program is not called for: a simpler procedure based on calls *to ccw* will be adequate. Another important special case is the *convex polygon, to* be studied in the next chapter, which has the property that no test line can have more than two intersections with the polygon. In this case, a procedure like binary search can be used to determine in O(logN) steps whether or not a point is inside.

Perspective

From the few examples given, it should be clear that it is easy to underestimate the difficulty of solving a particular geometric problem with a computer. There are many other elementary geometric computations that we have not treated at all. For example, a program to compute the area of a polygon makes an interesting exercise. However, the problems we've looked at have provided some basic tools that will be useful in later sections for solving the more difficult problems.

Some of the algorithms we'll study involve building geometric structures from a given set of points. The "simple closed polygon" is an elementary example of this. We will need to decide upon appropriate representations for such structures, develop algorithms to build them, and investigate their use in particular applications. As usual, these considerations are intertwined. For example, the algorithm used in the inside procedure in this chapter depends in an essential way on the representation of the simple closed polygon as an ordered set of points (rather than as an unordered set of lines).

Many of the algorithms we'll study involve *geometric search:* we want to know which points from a given set are close to a given point, or which points fall in a given rectangle, or which points are closest to one another. Many of the algorithms appropriate for such search problems are closely related to the search algorithms studied in Chapters 14-17. The parallels will be quite evident.

Few geometric algorithms have been analyzed to the point that precise statements can be made about their relative performance characteristics. As we've already seen, the running time of a geometric algorithm can depend on many things.

The distribution of the points themselves, the order in which they appear in the input, and whether trigonometric functions are used can all significantly affect the running time of geometric algorithms. As usual in such situations, however, we do have empirical evidence that suggests good algorithms for particular applications. Also, many of the algorithms are derived from complexity studies and are designed for good worst-case performance.