7

Search and Intersection

7.1. INTRODUCTION

In this (long) chapter we examine several problems that can be loosely classified as involving search or intersection (or both). This is a vast, well-developed topic, and I will make no attempt at systematic coverage.[1] The chapter starts with two constant-time computations that are generally below the level considered in the computational geometry literature: intersecting two segments (Section 7.2) and intersecting a segment with a triangle (Section 7.3). Implementations are presented for both tasks. Next we employ these algorithms for two more difficult problems: determining whether a point is in a polygon – the "point-in-polygon problem" (Section 7.4), and the "point-in-polyhedron problem" (Section 7.5). The former is a heavily studied problem; the latter has seen less scrutiny. Again implementations are presented for both. We next turn to intersecting two convex polygons (Section 7.6), again with an implementation (the last in the chapter). Intersecting a collection of segments (Section 7.7) leads to intersection of nonconvex polygons (Section 7.8).

The theoretical jewel in this chapter is an algorithm to find extreme points of a polytope in any given query direction (Section 7.10). This leads naturally to planar point location (Section 7.11), which allows us to complete the explanation of the randomized triangulation algorithm from Chapter 2 (Section 2.4.1) with a presentation of a randomized algorithm to construct a search structure for a trapezoid decomposition (Section 7.11.4).

7.2. SEGMENT–SEGMENT INTERSECTION

In Chapter 1 (Section 1.5) we spent some time developing code that detects intersection between two segments for use in triangulation (`Intersect`, Code 1.9), but we never bothered to *compute* the point of intersection. It was not needed in the triangulation algorithm, and it would have forced us to leave the comfortable world of integer coordinates. For many applications, however, the floating-point coordinates of the point of intersection are needed. We will need this to compute the intersections between two polygons in Sections 7.6 and 7.8. Fortunately, it is not too difficult to compute the intersection point (although there are potential pitfalls), and the necessary floating-point calculations are not as problematical here as they sometimes are. In this section we develop code for this task.

[1] See, e.g., de Berg et al. (1997).

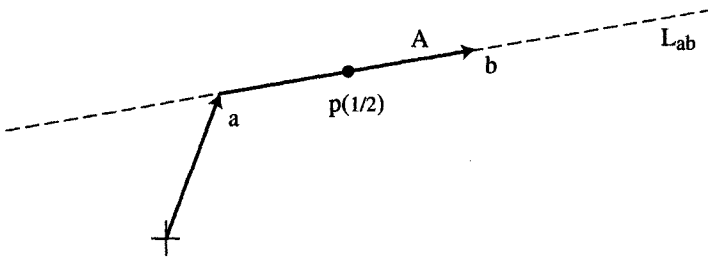

FIGURE 7.1 $p(s) = a + sA$; $p(\frac{1}{2}) = a + \frac{1}{2}A$ is shown.

Although the computation could be simplified a bit by employing the Boolean `Intersect` from Chapter 1, we opt here for an independent calculation. Let the two segments have endpoints a and b and c and d, and let L_{ab} and L_{cd} be the lines containing the two segments. A common method of computing the point of intersection is to solve slope–intercept equations for L_{ab} and L_{cd} simultaneously:[2] two equations in two unknowns (the x and y coordinates of the point of intersection). Instead we will use a parametric representation of the two segments, as the meaning of the variables seems more intuitive. We will see in Section 7.3 that the parametric approach generalizes nicely to more complex intersection computations.

Let $A = b - a$ and $C = d - c$; these vectors point along the segments. Any point on the line L_{ab} can be represented as the vector sum $p(s) = a + sA$, which takes us to a point a on L_{ab}, and then moves some distance along the line by scaling A by s. See Figure 7.1. The variable s is called the *parameter* of this equation. Consider the values obtained for $s = 0$, $s = 1$, and $s = \frac{1}{2}$: $p(0) = a$, $p(1) = a + A = a + b - a = b$, and $p(\frac{1}{2}) = (a + b)/2$. These examples demonstrate that $p(s)$ for $s \in [0, 1]$ represents all the points on the segment ab, with the value of s representing the fraction of the distance between the endpoints; in particular, the extremes of s yield the endpoints.

We can similarly represent the points on the second segment by $q(t) = c + tC$, $t \in [0, 1]$. A point of intersection between the segments is then specified by values of s and t that make $p(s)$ equal to $q(t)$: $a + sA = c + tC$. This vector equation also comprises two equations in two unknowns: the x and y equations, both with s and t as unknowns. With our usual convention of subscripts 0 and 1 indicating x and y coordinates, its solution is

$$s = [a_0(d_1 - c_1) + c_0(a_1 - d_1) + d_0(c_1 - a_1)]/D, \tag{7.1}$$

$$t = [a_0(c_1 - b_1) + b_0(a_1 - c_1) + c_0(b_1 - a_1)]/D, \tag{7.2}$$

$$D = a_0(d_1 - c_1) + b_0(c_1 - d_1) + d_0(b_1 - a_1) + c_0(a_1 - b_1) \tag{7.3}$$

Division by zero is a possibility in these equations. The denominator D happens to be zero iff the two lines are parallel, a claim left to Exercise 7.3.2[1]. Some parallel segments involve intersection, and some do not, as we detailed in Chapter 1 (Section 1.5.4). Temporarily, we will treat parallel segments as nonintersecting. The above equations lead to the rough code shown in Code 7.1. We will first describe this code, then criticize it, and finally revise it.

[2]E.g., see Berger (1986, pp. 332–5).

```
#define  X 0
#define  Y 1
typedef  enum {FALSE, TRUE }bool;
#define DIM 2                          /* Dimension of points */
typedef  int tPointi[DIM];             /* Type integer point */
typedef  double tPointd[DIM];          /* Type double point */

bool  SegSegInt( tPointi a, tPointi b, tPointi c, tPointi d,
                 tPointd p )
{

   double s, t;              /* Parameters of the parametric eqns. */
   double num, denom;        /* Numerator and denominator of eqns. */

   denom = a[X] * ( d[Y] - c[Y] ) +
           b[X] * ( c[Y] - d[Y] ) +
           d[X] * ( b[Y] - a[Y] ) +
           c[X] * ( a[Y] - b[Y] );

   /* If denom is zero, then segments are parallel. */
   if (denom == 0.0)
      return FALSE;

   num = a[X] * ( d[Y] - c[Y] ) +
         c[X] * ( a[Y] - d[Y] ) +
         d[X] * ( c[Y] - a[Y] );
   s = num / denom;

   num = -( a[X] * ( c[Y] - b[Y] ) +
            b[X] * ( a[Y] - c[Y] ) +
            c[X] * ( b[Y] - a[Y] ) );
   t = num / denom;

   p[X] = a[X] + s * ( b[X] - a[X] );
   p[Y] = a[Y] + s * ( b[Y] - a[Y] );

   if       ( (0.0 <= s) && (s <= 1.0) &&
              (0.0 <= t) && (t <= 1.0) )
         return TRUE;
   else  return FALSE;
}
```

Code 7.1 Segment–segment intersection code: rough attempt.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

The code takes the four integer-coordinate endpoints as input and produces two types of output: It returns a Boolean indicating whether or not the segments intersect, and it returns in the point p the double coordinates of the point of intersection; note that p is of type `double tPointd`. The computations of the numerators and denominators parallel Equations (7.1)–(7.3) exactly, and the test for intersection is $0 \leq s \leq 1$ and $0 \leq t \leq 1$.

There are at least three weaknesses to this code:

1. The code does not handle parallel segments. Most applications will need to know whether the segments overlap or not.
2. Many applications need to distinguish proper from improper intersections, just as we did for triangulation in Chapter 1. It would be useful to distinguish these in the output.
3. Although floating-point variables are used, the multiplications are still performed with integer arithmetic before the results are converted to `doubles`. Here is a simple example of how this code can fail due to overflow. Let the four endpoints be

$$a = (-r, -r),$$
$$b = (+r, +r),$$
$$c = (+r, -r),$$
$$d = (-r, +r).$$

The segments form an 'X' shape intersecting at $p = (0, 0)$. Calculation shows that the numerators from Equations (7.1) and (7.2) are both $-4r^2$. For $r = 10^5$, this is -4×10^{10}, which exceeds what can be represented in 32 bits. In this case my machine returns $p = (-267702.8, -267702.8)$ as the point of intersection!

We now address each of these three problems. First, we change the function from `bool` to `char` and have it return a "code" that indicates the type of intersection found. Applications that need to base decisions on whether or not the intersection is proper can use this code. Although the exact codes used should depend on the application, the following capture most needs:

'e': The segments collinearly overlap, sharing a point; 'e' stands for 'edge.'
'v': An endpoint of one segment is on the other segment, but 'e' doesn't hold; 'v' stands for 'vertex.'
'1': The segments intersect properly (i.e., they share a point and neither 'v' nor 'e' holds); '1' stands for `TRUE`.
'0': The segments do not intersect (i.e., they share no points); '0' stands for `FALSE`.

Note that the case where two collinear segments share just one point, an endpoint of each, is classified as 'e' in this scheme, although 'v' might be more appropriate in some contexts.

Second, we increase the range of applicability of the code by forcing the multiplications to floating-point by casting with `(double)`. This leads us to the code shown in Code 7.2. Before moving to the parallel segment case, let us point out a few features

```
char  SegSegInt( tPointi a, tPointi b, tPointi c, tPointi d,
                 tPointd p )
{

   double s, t;          /* The two parameters of the parametric eqns. */
   double num, denom;    /* Numerator and denominator of equations. */
   char code = '?';      /* Return char characterizing intersection. */


  denom = a[X] * (double)( d[Y] - c[Y] ) +
          b[X] * (double)( c[Y] - d[Y] ) +
          d[X] * (double)( b[Y] - a[Y] ) +
          c[X] * (double)( a[Y] - b[Y] );

  /* If denom is zero, then segments are parallel: handle separately. */
  if (denom == 0.0)
     return ParallelInt(a, b, c, d, p);

  num = a[X] * (double)( d[Y] - c[Y] ) +
        c[X] * (double)( a[Y] - d[Y] ) +
        d[X] * (double)( c[Y] - a[Y] );
  if ( (num == 0.0) || (num == denom) ) code = 'v';
  s = num / denom;

  num = -( a[X] * (double)( c[Y] - b[Y] ) +
           b[X] * (double)( a[Y] - c[Y] ) +
           c[X] * (double)( b[Y] - a[Y] ) );
  if ( (num == 0.0) || (num == denom) ) code = 'v';
  t = num / denom;

  if       ( (0.0 < s) && (s < 1.0) &&
             (0.0 < t) && (t < 1.0) )
    code = '1';
  else if ( (0.0 > s) || (s > 1.0) ||
            (0.0 > t) || (t > 1.0) )
    code = '0';

  p[X] = a[X] + s * ( b[X] - a[X] );
  p[Y] = a[Y] + s * ( b[Y] - a[Y] );

  return code;
}
```

Code 7.2 `SegSegInt`.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char  ParallelInt( tPointi a, tPointi b, tPointi c, tPointi d,
                   tPointd p )
{
  if ( !Collinear( a, b, c) )
      return '0';

  if ( Between( a, b, c ) ) {
      Assigndi( p, c ); return 'e';
  }
  if ( Between( a, b, d ) ) {
      Assigndi( p, d ); return 'e';
  }
  if ( Between( c, d, a ) ) {
      Assigndi( p, a ); return 'e';
  }
  if ( Between( c, d, b ) ) {
      Assigndi( p, b ); return 'e';
  }
  return '0';
}

void Assigndi( tPointd p, tPointi a )
{
  int i;
  for ( i = 0; i < DIM; i++ )
      p[i] = a[i];
}

bool    Between( tPointi a, tPointi b, tPointi c )
{
   tPointi      ba, ca;

   /* If ab not vertical, check betweenness on x; else on y. */
   if ( a[X] != b[X] )
     return ((a[X] <= c[X]) && (c[X] <= b[X])) ||
            ((a[X] >= c[X]) && (c[X] >= b[X]));
   else
     return ((a[Y] <= c[Y]) && (c[Y] <= b[Y])) ||
            ((a[Y] >= c[Y]) && (c[Y] >= b[Y]));
}
```

Code 7.3 ParallelInt.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

of this code. Checking the 'v' case is done with num rather than with s and t after division; this skirts possible floating-point inaccuracy in the division. The check for proper intersection is $0 < s < 1$ and $0 < t < 1$; the reverse inequalities yield no intersection.

With the computations forced to `doubles`, the range is greatly extended. I could only make it fail for coordinates each over a billion: $r = 1234567809 \approx 10^9$ in the previous overflow example. It is not surprising that it fails here, as $-4r^2$ is now over 10^{18}, which requires 60 bits, exceeding the accuracy of `double` mantissas on most machines.

Finally we come to parallel segments, handled by a separate procedure `ParallelInt`. Collinear overlap was dealt with in Chapter 1 with the function `Between` (Code 1.8), which is exactly what we need here: The segments overlap iff an endpoint of one lies between the endpoints of the other. There is one small simplification. In the triangulation code, we had `Between` check collinearity, but here we can make one check: If c is not collinear with ab, then the parallel segments ab and cd do not intersect. The straightforward code is shown in Code 7.3. Note that an endpoint is returned as the point of intersection p. It is conceivable that some application might prefer to have the midpoint of overlap returned; in Section 7.6 we will need the entire segment of overlap.

It should be clear that minor modification of this intersection code can find ray–segment, ray–ray, ray–line, or line–ray intersection, by altering the acceptable s and t ranges. For example, accepting any nonnegative s corresponding to a positive stretch of the first segment yields ray–segment intersection.

7.3. SEGMENT–TRIANGLE INTERSECTION

We now turn to the more difficult, but still ultimately straightforward, computation of the point of intersection between a segment and a triangle in three dimensions. We will use this code in Section 7.5 to detect whether a point is in a polyhedron, but it has many other uses. In fact this is one of the most prevalent geometric computations performed today, because it is a key step in "ray tracing" used in computer graphics: finding the intersection between a light ray and a collection of polygons in space.

We will again use a parametric representation to derive the equations. Throughout we will let $T = \triangle abc$ be the triangle and qr the segment, where q is viewed as the originating ("query") endpoint in case qr represents a ray and r is the "ray" endpoint. We will assume throughout that $r \neq q$, so the input segment has nonzero length.

7.3.1. Segment–Plane Intersection

The first step is to determine if qr intersects the plane π containing T. We will pursue this halfway goal throughout this subsection before turing to determining if the point of intersection lies in the triangle.

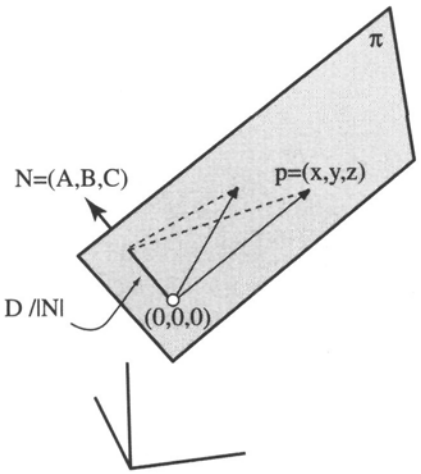

FIGURE 7.2 The dot product of a point on the plane with N is a constant, D when $|N| = 1$.

Recall that just as all points on a line must satisfy a linear equation in x and y, so too must all the points on a plane satisfy an equation

$$\pi : Ax + By + Cz = D. \tag{7.4}$$

We will represent the plane by these four coefficients.[3] It will help to view the first three coefficients as a vector (A, B, C), for then the plane equation can be viewed as a dot product:

$$\pi : (x, y, z) \cdot (A, B, C) = D. \tag{7.5}$$

The geometric meaning of this equation is that every point (x, y, z) on the plane projects to the same length onto (A, B, C). From this and Figure 7.2 it should be clear that $N = (A, B, C)$ is a vector normal (i.e., perpendicular) to the plane. If this vector is unit length,[4] then D is the distance from the origin to π.

Just as in two dimensions, any point $p(t)$ on the segment can be represented by moving out to the q endpoint, and then adding a scaled version of a vector along the segment: $p(t) = q + t(r - q)$. Let us temporarily assume that $q = (0, 0, 0)$ so that $p(t) = tr$; this will make the calculations more transparent. Now we are seeking a value of the parameter t that will stretch r out to the plane. Because every point on the plane must satisfy Equation (7.5), we must have

$$\begin{aligned} p(t) \cdot (A, B, C) &= D, \\ tr \cdot N &= D, \\ t(r \cdot N) &= D, \\ t &= \frac{D}{r \cdot N}. \end{aligned} \tag{7.6}$$

[3] One could "normalize" the equation by dividing by D; then only three coefficients are needed. This, however, requires dealing with $D = 0$ separately.

[4] Unit vectors are said to be "normalized," which can cause confusion with "normal" used in the sense of perpendicular.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

For example, consider the plane containing the three points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. Its plane equation is $x + y + z = 1$, so $N = (1, 1, 1)$. Let $r = (1, 2, 3)$. Equation (7.6) yields

$$t = \frac{1}{(1, 2, 3) \cdot (1, 1, 1)} = \frac{1}{6}.$$

And indeed $tr = \frac{1}{6}(1, 2, 3) = (\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ lies on the plane because $\frac{1}{6} + \frac{1}{3} + \frac{1}{2} = 1$.

Generalizing Equation (7.6) for arbitrary q is not difficult:

$$\begin{aligned} [q + t(r - q)] \cdot N &= D, \\ q \cdot N + t(r - q) \cdot N &= D, \\ t &= \frac{D - q \cdot N}{(r - q) \cdot N}. \end{aligned} \tag{7.7}$$

The reader may wonder why this more complex situation yields an equation in only one unknown, whereas the simpler segment–segment intersection led to two equations in two unknown parameters (Equations (7.1) and (7.2)). The reason is that we have not pinpointed the intersection with respect to the triangle yet; that will involve other unknowns. We could have followed this same strategy in two dimensions, first intersecting a segment with a line, but the situation there was simple enough to permit jumping right to simultaneous determination of the parameters.

One more step remains before we develop code: obtaining the plane coefficients. Usually, and in our case, we start with three points determining a triangle in space, not with the coefficients. The coefficients may be found using the observation above that three of them define a vector normal to the triangle. We can use the cross product from Chapter 1 (Equation (1.1)) to determine N, just as it was used in Chapter 5 to find the lower hull (Section 5.7.4). With N in hand, we can find the fourth coefficient D by substituting any point on the plane, for example, one of the triangle vertices, into Equation (7.5).

Segment–Plane Implementation

We now proceed to code. Looking ahead to how we will employ this code, we use the following simple data structure: Input points will be of type `tPointi`, stored in an array `tPointi Vertices[PMAX]`. Triangles are represented as three integer indices into this array. With some possibility of confusion, we will use type `tPointi` for these three indices. Eventually we will have a collection of triangles, which are stored in an array `tPointi Faces[PMAX]`. Thus `Face[i]` is a particular triangle, and if `T` is a triangle, `Vertices[T[j]]`, $j = 0, 1, 2$ are its three vertices. The data will be read in with straightforward read routines we will not show. See Code 7.4.

We partition the work into two procedures. The first, `PlaneCoeff`, computes N and D as just detailed. We choose to return the coefficients in two pieces, as they are employed in Equation (7.7) separately. For reasons that will become apparent later, we also record and return the index m of the largest component of N. See Code 7.5. The

```
#define X 0
#define Y 1
#define Z 2
#define DIM 3                       /* Dimension of points */
typedef int    tPointi[DIM];        /* Type integer point */
typedef double tPointd[DIM];        /* Type double point */
#define PMAX 10000                  /* Max # of pts */
tPointi Vertices[PMAX];             /* All the points */
tPointi Faces[PMAX];                /* Each triangle face is 3 indices */

main()
{
   int V, F;

   V = ReadVertices();
   F = ReadFaces();
   /* Processing */
}
```

Code 7.4 Type definitions for triangles in space.

```
int     PlaneCoeff( tPointi T, tPointd N, double *D )
{
  int i;
  double t;                 /* Temp storage */
  double biggest = 0.0;     /* Largest component of normal vector. */
  int m = 0;                /* Index of largest component. */

  NormalVec( Vertices[T[0]], Vertices[T[1]], Vertices[T[2]], N );
  *D = Dot( Vertices[T[0]], N );

  /* Find the largest component of N. */
  for ( i = 0; i < DIM; i++ ) {
    t = fabs( N[i] );
    if ( t > biggest ) {
      biggest = t;
      m = i;
    }
  }
  return m;
}
```

Code 7.5 `PlaneCoeff`.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
void    NormalVec( tPointi a, tPointi b, tPointi c, tPointd N )
{
  N[X]=( c[Z] - a[Z] ) * ( b[Y] - a[Y] ) -
       ( b[Z] - a[Z] ) * ( c[Y] - a[Y] );
  N[Y]=( b[Z] - a[Z] ) * ( c[X] - a[X] ) -
       ( b[X] - a[X] ) * ( c[Z] - a[Z] );
  N[Z]=( b[X] - a[X] ) * ( c[Y] - a[Y] ) -
       ( b[Y] - a[Y] ) * ( c[X] - a[X] );
}

double  Dot( tPointi a, tPointd b )
{
  int i;
  double sum = 0.0;

  for( i = 0; i < DIM; i++ )
     sum += a[i] * b[i];
  return sum;
}

void    SubVec( tPointi a, tPointi b, tPointi c )
{
  int i;

  /* a - b => c. */
  for( i = 0; i < DIM; i++ )
     c[i] = a[i] - b[i];
}
```

Code 7.6 Vector utility functions.

code for `NormalVec` (Code 7.6) follows Code 4.12, returning $N = (b - a) \times (c - a)$. Note that we represent N with `double`s even though its coordinates are integers, for the familiar reason: to stave off overflow.

We will follow the convention established in Section 7.2 in having the intersection procedure return a code to classify the intersection:

'p': The segment lies wholly within the plane.
'q': The (first) q endpoint is on the plane (but not 'p').
'r': The (second) r endpoint is on the plane (but not 'p').
'0': The segment lies strictly to one side or the other of the plane.
'1': The segment intersects the plane, and none of {p, q, r} hold.

We now discuss how to determine when the code 'p' applies. When the denominator of Equation (7.7) is zero, then qr is parallel to the plane π. This can perhaps best be seen in the simpler version, Equation (7.6), where it is clear that the denominator is zero iff r is orthogonal to N (i.e., if r is parallel to the plane π to which N is orthogonal). It is also clear from that equation that if, in addition, the numerator D is zero, then

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char SegPlaneInt( tPointi T, tPointi q, tPointi r, tPointd p,
                  int *m)
{
  tPointd N; double D;
  tPointi rq;
  double num, denom, t;
  int i;

  *m = PlaneCoeff( T, N, &D );
  num = D - Dot( q, N );
  SubVec( r, q, rq );
  denom = Dot( rq, N );

  if ( denom == 0.0 ) {   /* Segment is parallel to plane. */
     if ( num == 0.0 )    /* q is on plane. */
        return 'p';
     else
        return '0';
  }
  else
     t = num / denom;

  for( i = 0; i < DIM; i++ )
     p[i] = q[i] + t * ( r[i] - q[i] );

  if ( (0.0 < t) && (t < 1.0) )
       return '1';
  else if ( num == 0.0 )        /* t == 0 */
       return 'q';
  else if ( num == denom )      /* t == 1 */
       return 'r';
  else return '0';
}
```

Code 7.7 `SegPlaneInt`.

r lies in the plane. Generalizing to qr, we see in Equation (7.7) that the numerator is zero whenever $q \cdot N = D$, which is precisely the plane equation ((7.5)) with q substituted. So the numerator is zero iff q lies on π. Thus the code 'p' should be returned iff both the numerator and denominator are zero. The codes 'q' and 'r' are determined by $t = 0$ and $t = 1$ respectively, which may be tested on the numerator and denominator to avoid reliance on the floating-point division.

See Code 7.7.

Segment–Triangle Classification

Now that we have the point p of intersection between the segment qr and the plane π containing the triangle T, it only remains to classify the relationship between p and T: Is it inside or out, on the boundary, at a vertex? Although this may seem a simple task,

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

there are some subtleties. We first describe an elegant mathematical approach that we will ultimately choose not to implement.

Barycentric Coordinates. The *barycenter* of an object is its center of gravity.[5] The *barycentric coordinates* of a point p with respect to a triangle $T = \triangle abc$ (in two or three or any dimensions) are the unique real numbers (α, β, γ) that sum to 1 such that

$$\alpha a + \beta b + \gamma c = p. \tag{7.8}$$

From the discussion of convex combinations and affine combinations in Chapter 3 (Section 3.1 and Exercise 3.2.3[4]), it should be clear that Equation (7.8) describes a point on the plane π. The point is in T iff each of the three barycentric coordinates is in $[0, 1]$. The coordinates can be viewed as masses placed at the vertices whose center of gravity is p. For example, let $a = (0, 0)$, $b = (1, 0)$, and $c = (3, 2)$. The barycentric coordinates $(\alpha, \beta, \gamma) = (\frac{1}{2}, 0, \frac{1}{2})$ specify the point $p = (0, 0)/2 + 0(1, 0) + (3, 2)/2 = (\frac{3}{2}, 1)$, the midpoint of the ac edge.

This example illustrates that all the classes we might want to distinguish are encoded in the barycentric coordinates: p is on an edge interior iff exactly one coordinate is zero, p coincides with a vertex iff one coordinate is one, and of course the inside/outside distinction is determined by whether the coordinates are in $[0, 1]$.

The barycentric coordinates can be calculated by solving the Equation (7.8) together with $\alpha + \beta + \gamma = 1$. This gives four equations in three unknowns for three-dimensional triangles. Because the triangle lies in plane, we have redundant information, and the problem can be reduced to solving three equations in three unknowns.

Although it is quite possible to perform this computation, we choose another tack, partly to connect with techniques we used in Chapter 4, and partly because the computation slides into needing considerable precision. Let us make a crude estimate of this precision, assuming no attempt at optimizing. If our input coordinates use L bits of precision, then the normal vector N uses $2L$, and $q \cdot N$ consumes $3L$. Thus the numerator and, similarly, the denominator of Equation (7.7) are each $3L$, so t needs potentially $6L$ bits. Next t is multiplied by $r - q$, raising the count to $7L$ for p. And we have not even started solving the barycentric coordinate equations. We conclude that it will be delicate to classify p based on the floating-point representation of p. Nevertheless, we will in any case need to classify p when it is an endpoint of the query segment (which has precision only L), and we proceed to this task next.

Projection to Two Dimensions. The situation is that we have a point p known to lie on the plane π containing triangle T, and we would like to classify p's relationship to T. Because p lies in π, the problem is fundamentally two dimensional, not three dimensional. However, it would take a bit of work to translate and rotate π so that it coincides with, say, the xy-plane. But two observations allow us to solve the problem

[5] "Bary" means "heavy" in Greek.

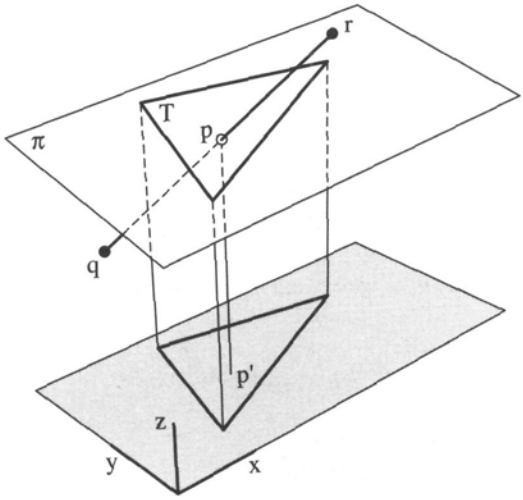

FIGURE 7.3 $p \in T$ iff $p' \in T'$.

in two dimensions without this realignment of the plane. First, p is in T iff it is in a projection of T, say to the xy-plane. This is evident from Figure 7.3. Let p' and T' be the projections of p and T respectively. The complete classification of p with respect to T can be made with these projections: p is in the interior of an edge of T iff p' is in the interior of an edge of T', and so on. But there is a worry: What if π is vertical, when the claim just made fails? This can be avoided by a second observation: Projecting out the coordinate corresponding to the largest component of the vector N normal to π guarantees nondegeneracy. Thus a nearly horizontal plane has a large z component, and projection to the xy-plane is called for. A vertical plane's N will have zero z component and so will be projected to either the xz- or yz-plane, depending on which one is closer to being parallel to π. This is why Code 7.5 computed the index m of the largest component.

We are now prepared to write a procedure `InTri3D` that classifies a point p on a triangle T using the following classification scheme:

'V': p coincides with a Vertex of T.
'E': p is in the relative interior of an Edge of T.
'F': p is in the relative interior of a Face of T.
'0': p does not intersect T.

The top-level code, shown in Code 7.8, does very little: It projects out coordinate m, and passes p' and T' to a procedure `InTri2D` that operates on the two-dimensional projection. Note that we fill up the x and y coordinate slots of `pp` and `Tp` regardless of the coordinate of projection.

Now that the problem is in the xy-plane, it is easy to solve. We can classify p' by computing signed areas as in Chapter 1. The only complication is that we do not know the orientation of the three vertices. But because there are only three, the given order

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char    InTri3D( tPointi T, int m, tPointi p )
{
  int i;              /* Index for X,Y,Z */
  int j;              /* Index for X,Y */
  int k;              /* Index for triangle vertex */
  tPointi pp;         /* projected p */
  tPointi Tp[3];      /* projected T: three new vertices */

  /* Project out coordinate m in both p and the triangular face */
  j = 0;
  for ( i = 0; i < DIM; i++ ) {
    if ( i != m ) {      /* skip largest coordinate */
      pp[j] = p[i];
      for ( k = 0; k < 3; k++ )
        Tp[k][j] = Vertices[T[k]][i];
      j++;
    }
  }
  return( InTri2D( Tp, pp ) );
}
```

Code 7.8 `InTri3D`.

must be either counterclockwise or clockwise. The code must handle both orientations. `InTri2D` first computes the three areas determined by p' with each of the three edges of T'. The classification is based on these areas. See Figure 7.4 (and compare with Figure 1.19). If all three are positive, or all three negative, p' is strictly interior to T'. If two are zero, then p' lies on the lines containing two edges (i.e., at a vertex). If all three are zero, then p' must be collinear with all three edges, which can only happen when T' lies in a line. This case should never occur, so we exit with an error message. That leaves only the case when a single area is zero and the other two are nonzero. Only when the other two have the same sign does p' lie on the interior of an edge of T'. This leads to the code shown in Code 7.9.

Segment in Plane. It should be clear now that the case where the segment qr lies in the plane π can be handled by the same projection method: Project to two dimensions, check if either q' or r' lies in T' (in which case the corresponding endpoint may be returned as p), and if not, check if $q'r'$ intersects each edge of T', using `SegSegInt` (Code 7.2). As we have all these pieces of code assembled, we will not pursue this further, but rather leave the implementation details to Exercise 7.3.2[4].

Classification by Volumes. We are finally prepared to tackle the "usual" case, where qr crosses plane π, and therefore q is on one side and r is on the other. We can classify how qr meets T in a manner similar to how we classified p' in `InTri2D`, except now we compute volumes rather than areas. In particular, we compute the signed volumes of

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

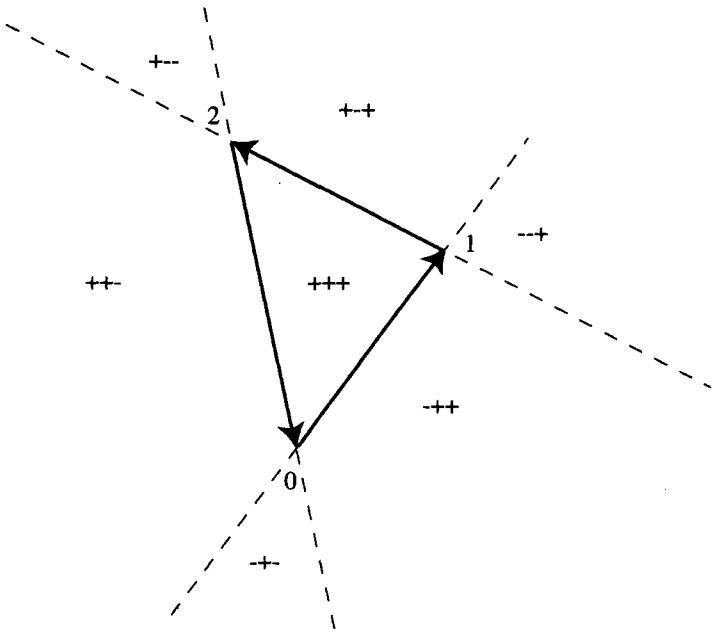

FIGURE 7.4 Assuming the edges of T' are counterclockwise, the sign pattern of the areas determined by p' and each edge are as shown. The boundary line between each + and − has "sign" 0.

the three tetrahedra determined by qr and each edge of T.[6] Let $T = (v_0, v_1, v_2)$. Then the volumes we use are $V_i = \text{Volume}(q, v_i, v_{i+1}, r)$. As with the two-dimensional case, we can only assume the vertices are ordered counterclockwise or clockwise, but this is enough information. We will employ this classification scheme:

'v': The open segment includes a vertex of T.
'e': The open segment includes a point in the relative interior of an edge of T.
'f': The open segment includes a point in the relative interior of a face of T.
'0': The open segment does not intersect triangle T.

If all three V_i are positive, or all three negative, then qr goes through a point strictly interior to T; see Figure 7.5(f). If two of the V_i are of opposite sign, then qr misses T. If one is zero, then qr passes through a point interior to some edge. For example, in Figure 7.5(e), $V_1 = 0$. If two are zero, qr passes through a vertex. In Figure 7.5(v), $V_1 = V_2 = 0$. If all three V_i are zero this implies that qr lies in the plane of T, a situation handled earlier. All these conditions are easily seen to be necessary and sufficient for the corresponding characterization. The straightforward implementation of these rules is embodied in the procedure `SegTriCross`, Code 7.10. `VolumeSign` is the same

[6] An elegant formulation of the same computation can be based on "Plücker coordinates" (Erickson 1997).

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char    InTri2D( tPointi Tp[3], tPointi pp )
{
  int area0, area1, area2;

  area0 = AreaSign( pp, Tp[0], Tp[1] );
  area1 = AreaSign( pp, Tp[1], Tp[2] );
  area2 = AreaSign( pp, Tp[2], Tp[0] );

  if ( ( area0 == 0 ) && ( area1 > 0 ) && ( area2 > 0 )||
       ( area1 == 0 ) && ( area0 > 0 ) && ( area2 > 0 )||
       ( area2 == 0 ) && ( area0 > 0 ) && ( area1 > 0 ))
    return 'E';

  if ( ( area0 == 0 ) && ( area1 < 0 ) && ( area2 > 0 )||
       ( area1 == 0 ) && ( area0 < 0 ) && ( area2 > 0 )||
       ( area2 == 0 ) && ( area0 < 0 ) && ( area1 > 0 ))
    return 'E';

  if ( ( area0 > 0 ) && ( area1 > 0 ) && ( area2 > 0 )||
       ( area0 < 0 ) && ( area1 < 0 ) && ( area2 < 0 ))
    return 'F';

  if ( ( area0 == 0 ) && ( area1 == 0 ) && ( area2 == 0 ) )
    fprintf( stderr, "Error in InTriD\n" ),
      exit (EXIT_FAILURE);

  if ( ( area0 == 0 ) && ( area1 == 0 ) ||
       ( area0 == 0 ) && ( area2 == 0 ) ||
       ( area1 == 0 ) && ( area2 == 0 ) )
    return 'V';

  else
    return '0';
}
```

Code 7.9 `InTri2D`. See Code 4.23 for `AreaSign`.

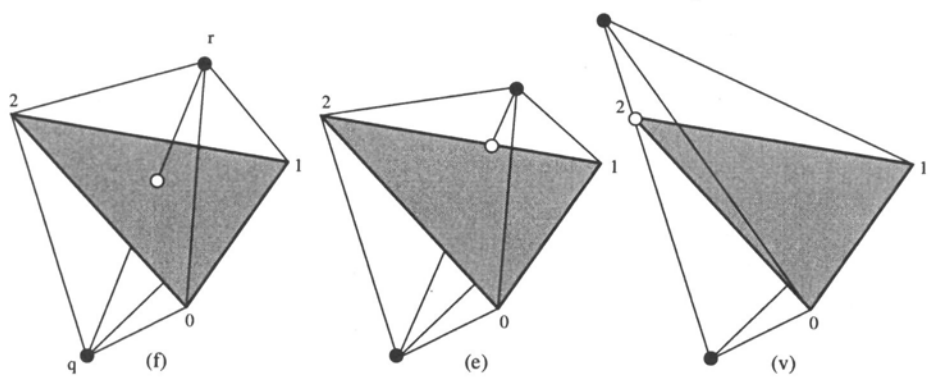

FIGURE 7.5 The segment *qr* intersects *T* in the face (f), on an edge (e), or through a vertex (v).

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char SegTriCross( tPointi T, tPointi q, tPointi r )
{
  int vol0, vol1, vol2;

  vol0 = VolumeSign( q, Vertices[ T[0] ], Vertices[ T[1] ], r );
  vol1 = VolumeSign( q, Vertices[ T[1] ], Vertices[ T[2] ], r );
  vol2 = VolumeSign( q, Vertices[ T[2] ], Vertices[ T[0] ], r );

  /* Same sign: segment intersects interior of triangle. */
  if ( ( ( vol0 > 0 ) && ( vol1 > 0 ) && ( vol2 > 0 ) ) ||
       ( ( vol0 < 0 ) && ( vol1 < 0 ) && ( vol2 < 0 ) ) )
    return 'f';

  /* Opposite sign: no intersection between segment and triangle. */
  if ( ( ( vol0 > 0 ) || ( vol1 > 0 ) || ( vol2 > 0 ) ) &&
       ( ( vol0 < 0 ) || ( vol1 < 0 ) || ( vol2 < 0 ) ) )
    return '0';

  else if ( ( vol0 == 0 ) && ( vol1 == 0 ) && ( vol2 == 0 ) )
    fprintf( stderr, "Error 1 in SegTriCross\n" ),
    exit (EXIT_FAILURE);

  /* Two zeros: segment intersects vertex. */
  else if ( ( ( vol0 == 0 ) && ( vol1 == 0 ) ) ||
            ( ( vol0 == 0 ) && ( vol2 == 0 ) ) ||
            ( ( vol1 == 0 ) && ( vol2 == 0 ) ) )
    return 'v';

  /* One zero: segment intersects edge. */
  else if ( ( vol0 == 0 ) || ( vol1 == 0 ) || ( vol2 == 0 ) )
    return 'e';

  else
    fprintf( stderr, "Error 2 in SegTriCross\n" ),
    exit (EXIT_FAILURE);
}
```

Code 7.10 `SegTriCross`. See Code 4.16 for `VolumeSign`.

as Code 4.16 used in Chapter 4, with accomodation for the slightly different input data structures.

This completes our development of code to intersect a segment with a triangle. The simple top-level procedure is shown in Code 7.11. With `InPlane` unimplemented and simply returning 'p', the code returns a character in {`0`, `p`, `V`, `E`, `F`, `v`, `e`, `f`}, with the following mutually exclusive meanings:

'0': The closed segment does not intersect T.

'p': The segment lies wholly within the plane of T. All the remaining categories assume that 'p' does not hold.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

'V': An endpoint of the segment coincides with a Vertex of T.
'E': An endpoint of the segment is in the relative interior of an Edge of T.
'F': An endpoint of the segment is in the relative interior of a Face of T.
'v': The open segment includes a vertex of T.
'e': The open segment includes a point in the relative interior of an edge of T.
'f': The open segment includes a point in the relative interior of a face of T.

The return codes may be viewed as a refinement on the usual Boolean 0/1, expanding 1 into seven types of degenerate intersection. As mentioned earlier, it is easy to modify this to permit intersection of a ray or line with a triangle, by permitting the range of the parameter t to vary outside of [0, 1]. We will delay illustrating the use of this code until Section 7.5.

```
char    SegTriInt( tPointi T, tPointi q, tPointi r, tPointd p )
{
  int code;
  int m;

  code = SegPlaneInt( T, q, r, p, &m );

  if      ( code == 'q')
      return InTri3D( T, m, q );
  else if ( code == 'r')
      return InTri3D( T, m, r );
  else if ( code == 'p' )
      return InPlane( T, m, q, r, p );
  else
      return SegTriCross( T, q, r );
}
```

Code 7.11 `SegTriInt`.

7.3.2. Exercises

1. *Denominator zero.* Prove that the denominator in the segment–segment intersection equations (Equations (7.1)–(7.3)) is zero iff the segments are parallel.
2. *Ray–segment intersection* [programming]. Modify the `SegSegInt` code to `RaySegInt`, interpreting a as a ray origin and b as a point on the ray, so that it returns a character code indicating the variety of possible intersections between the ray and the segment cd.
3. *Barycentric coordinates.* Let p be a point in the triangle $T = (v_1, v_2, v_3)$ with barycentric coordinates (t_1, t_2, t_3). Join p to the three vertices, partitioning T into three triangles. Call them T_1, T_2, T_3, with T_i not incident to v_i. Prove that the areas of these triangles T_i are proportional to the barycentric coordinates t_i of p (Coxeter 1961, p. 217).
4. *Segment in plane* [programming]. Extend the `SegTriInt` code to handle the case where qr lies entirely in the plane of T, by implementing an appropriate procedure `InPlane`.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

7.4. POINT IN POLYGON

Every time a mouse is clicked inside a shape on a workstation screen, an instance of the point-in-polygon problem is solved: Given a fixed polygon P and a query point q, is $q \in P$? Although the hardware of a particular machine may permit solutions that avoid geometry, we consider the problem here from the computational geometry viewpoint.

If P is convex, the obvious method is to perform a `LeftOn` test (Code 1.6) for each edge of the polygon. Indeed, we used precisely this technique in the two-dimensional incremental hull algorithm in Chapter 3 (Section 3.7). This can be improved to $O(\log n)$, but we leave this to Exercise 7.4.3[1].

The more interesting case is when P is nonconvex. Two rather different methods for solving this problem have become popular: counting ray crossings and computing "winding" numbers.[7] Both are $O(n)$, but one is significantly faster than the other. These algorithms are the topics of the next two subsections.

7.4.1. Winding Number

We start with a mathematically pleasing method that, alas, has been shown to be greatly inferior in practice. It is based on the notion of the "winding number" of a polygon. Imagine you are standing at point q. While watching a point p completely traverse ∂P counterclockwise, pivot so that you always face toward p. If $q \in P$, you would turn a full circle, 2π radians, whereas if $q \notin P$, your total angular turn would be exactly zero (with the usual convention: counterclockwise turns are positive, and clockwise turns negative). This is easy to see if P is convex, and I hope at least intuitively believable when P is arbitrary: After all, you return to your starting orientation, so the total turn must be a whole number of revolutions. See Figure 7.6. We will not pause to prove this claim. The *winding number*[8] of q with respect to P is the number of revolutions ∂P makes about q: the total signed angular turn (call it ω) divided by 2π. We will leave details of the computation to Exercise 7.4.3[8].

Although the winding-number algorithm is appealing, its dependence on floating-point computations, and trigonometric computations in particular, makes it significantly slower (on standard hardware) than the ray-crossing algorithm which we discuss next: An implementation comparison showed it to be more than twenty times slower (Haines 1994)! This incidentally demonstrates the danger of thoughtlessly absorbing constants in the big-O notation.

7.4.2. Ray Crossings

Draw a ray R from q in an arbitrary direction (say, in the $+x$ direction), and count the number of intersections of R with ∂P. The point q is in or out of P if the number of crossings is odd or even, respectively. For example, suppose there are two crossings, as with point q_2 in Figure 7.7, and imagine traveling backwards along R from infinity to

[7] See Haines (1994) for other methods.

[8] See Chinn & Steenrod (1966, pp. 84–6); the winding number is also used in Foley et al. (1990, p. 965).

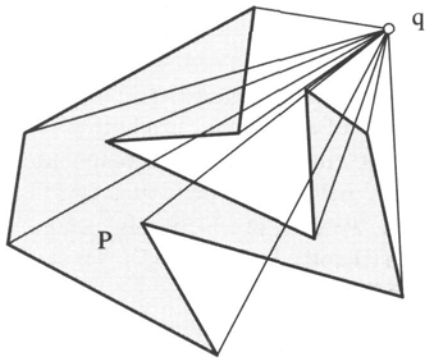

FIGURE 7.6 Exterior points (such as point q) have winding number 0: a total angular turn of 0.

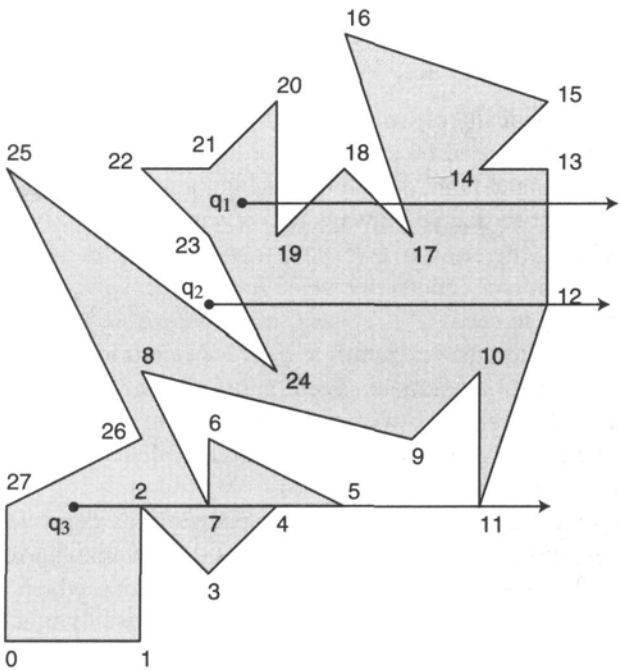

FIGURE 7.7 q_1's ray has five crossings and is inside; q_2's has two crossings and is outside; q_3's rays has five-crossings and is inside.

q_2. The first crossing penetrates to the interior of P; the second moves to the exterior. So $q_2 \notin P$. Similar reasoning shows that q_1 in the figure, whose ray has five crossings, must be inside P.

Despite the simplicity of this idea, implementation is fragile due to the necessity of handling special-case intersections of R with ∂P, as illustrated with point q_3 in Figure 7.7: The ray may hit a vertex or be collinear with an edge. There is also the possibility that q lies directly on ∂P, in which case we would like to conclude that $q \in P$ (because P is closed). Note that even the traditional assumption that no three polygon vertices are collinear will not exclude all these "degenerate" cases.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

Fix R to be horizontal to the right. One method of eliminating most of the difficulties is to require that for an edge e to count as a *crossing* of R, one of e's endpoints must be strictly above R, and the other endpoint on or below. Informally, e is considered to include its lower endpoint but exclude its upper endpoint.[9] Applying this convention for q_3 of the figure, edges (1, 2) and (2, 3) are not crossing (neither edge has an endpoint strictly above), (6, 7) and (7, 8) do count as crossing (v_7 is on or below), (3, 4) and (4, 5) do not cross, and (5, 6), (10, 11), and (11, 12) all cross. The total of five crossings implies that $q_3 \in P$. Note that no edge collinear with the ray counts as crossing, as it has no point strictly above.

Before revealing what this convention leaves unresolved, we turn to simple code for a function `InPoly0` (Code 7.12) implementing the idea.[10] The code first translates the entire polygon so that q becomes the origin and R coincides with the positive x axis. This step is unnecessary (and wasteful) but makes the code more transparent.[11] In a loop over all edges $e = (i - 1, i)$, it checks whether e "straddles" the x axis according to the definition above. If e straddles, then the x coordinate of the intersection of e with $y = 0$ is computed via a straightforward formula obtained by solving for x in the equation

$$y - y_{i-1} = (x - x_{i-1})(y_i - y_{i-1})/(x_i - x_{i-1}) \tag{7.9}$$

and setting $y = 0$; here (x_{i-1}, y_{i-1}) and (x_i, y_i) are the endpoints of e. Note that `x` is `double` in the code; this dependence on a floating-point calculation can be eliminated (Exercise 7.4.3[7]), but we will leave it to keep attention focused on the algorithm. A crossing is counted whenever the intersection is strictly to the right of the origin. The code returns the character 'i' or 'o' to indicate "in" or "out" respectively.

There is a flaw to this code (aside from the floating-point calculation): Although it returns the correct answer for any point strictly interior to P, it does not handle the points on ∂P consistently. If q_3 is moved horizontally to v_4 in Figure 7.7, `InPoly0` returns `i`, but if q_3 is placed at v_5, it returns `o`. The behavior of this code for points on ∂P is complex, as shown in Figure 7.8. Let us analyze why v_{27} is considered inside. Neither edge (26, 27) nor (27, 0) counts as crossing, because of the strict inequality in the statement `if (x > 0)`. Otherwise v_{27}'s ray has the same five crossing as q_3's, and so $v_{27} \in P$. However, note that v_{22}, a superficially similar vertex, is deemed exterior. How the code treats edges is a bit easier to characterize: Left and horizontal bottom edges are in; right and horizontal top edges are out.

Although this hodgepodge treatment of points on the polygon boundary is dissatisfying from a purist's point of view, for some applications, notably GIS (Geographic Information Systems), this (or similar) behavior is preferred, because it has the property that in a partition of a region into many polygons, every point will be "in" exactly one polygon.[12] This is not obvious, but we'll take it as fact (Exercise 7.4.3[4]). Other

[9] This rule is followed, e.g., in the polygon-filling algorithm of Foley, van Dam, Feiner, Huges & Phillips (1993, Sec. 3.5).

[10] This code is functionally equivalent to many others, e.g., that in FAQ (1997) and Haines (1994).

[11] Note that as written the code will overwrite the polygon coordinates with the shifted coordinates, a side effect rarely desired. Exercise 7.4.3[6] asks for the simple modifications that avoid this.

[12] I owe this point to Haines (1997).

```
char InPoly0( tPointi q, tPolygoni P, int n )
{
  int    i, i1;          /* point index; i1 = i-1 mod n */
  int    d;              /* dimension index */
  double x;              /* x intersection of e with ray */
  int    Rcross = 0;     /* number of right edge/ray crossings */

  /* Shift so that q is the origin. Note this destroys the polygon.
     This is done for pedagogical clarity. */
  for( i = 0; i < n; i++ ) {
     for( d = 0; d < DIM; d++ )
        P[i][d] = P[i][d] - q[d];
  }

  /* For each edge e = (i-1,i), see if crosses ray. */
  for( i = 0; i < n; i++ ) {
     i1 = ( i + n - 1 ) % n;

    /* If e "straddles" the x axis... */
    if( ( ( P[i] [Y] > 0 ) && ( P[i1][Y] <= 0 ) ) ||
        ( ( P[i1][Y] > 0 ) && ( P[i] [Y] <= 0 ) ) ) {

      /* ... compute intersection with x axis. */
      x = (P[i][X] * P[i1][Y] - P[i1][X] * P[i][Y])
          / (double)(P[i1][Y] - P[i][Y]);

      /* e crosses ray if strictly positive intersection. */
      if (x > 0) Rcross++;
    }
  } /* end for */

  /* q inside iff an odd number of crossings. */
  if( (Rcross % 2) == 1 ) return 'i';
  else return 'o';
}
```

Code 7.12 `InPoly0`. Typedefs in Code 7.4 and 1.2.

applications demand a consistent treatment of the boundary points, or a distinction between strictly interior, strictly exterior, and on the boundary.

Although `InPoly0` could be modified with ad hoc tests to check if the query point lies on each edge of the boundary, the characterization of the treatment of edges mentioned above can be exploited to achieve relatively simple code that completely determines the relationship of q with P. The idea is this: If we reflect P in the origin to form a new polygon P', all left and bottom edges become top and right edges respectively, and vice versa. Consequently, points on edges can be distinguished by `InPoly0` giving different

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

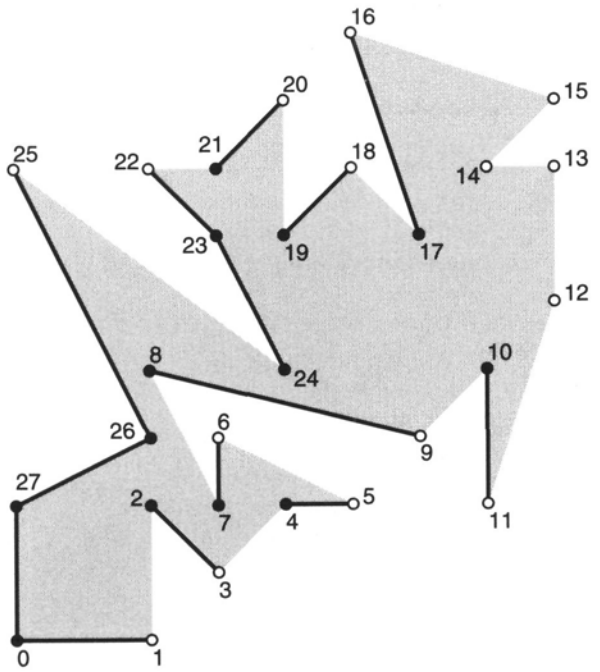

FIGURE 7.8 According to `InPoly0`, dark edges and solid vertices are `i`; other edges and open circles are `o`.

results for the same q in P and in P'. There is no need to actually perform the reflection: Using a leftward ray, with a "straddles" test biased below rather than the above bias used with rightward rays, achieves the same effect.

This method handles points on the relative interior of the edges but does not work for vertices (e.g., both of v_{16}'s rays have zero crossings, due to the `x > 0` test). An explicit easy check for q being a vertex handles this case. This leads to `InPoly1`, Code 7.13, which returns one of four `chars`: {`i`, `o`, `e`, `v`}, representing these mutually exclusive cases:

'i': q is strictly interior.
'o': q is strictly exterior (outside).
'e': q is on an edge, but not an endpoint.
'v': q is a vertex.

A few comments on this code need to be made. The straddles test from `InPoly0`,

```
( ( P[i] [Y] > 0 ) && ( P[i1][Y] <= 0 ) ) ||
( ( P[i1][Y] > 0 ) && ( P[i] [Y] <= 0 ) )
```

has been replaced in `InPoly1` with an assignment of this expression to the Boolean variable `Rstrad`:

```
( P[i][Y] > 0 ) != ( P[i1][Y] > 0 )
```

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char InPoly1( tPointi q, tPolygoni P, int n )
{
  /* Declarations of i, i1, d, x same as in InPoly0; DELETED here. */
  int     Rcross = 0;        /* number of right edge/ray crossings */
  int     Lcross = 0;        /* number of left edge/ray crossings */
  bool    Rstrad, Lstrad;    /* flags indicating the edge strads the x axis. */

  /* Shift so that q is the origin, same as in InPoly0; DELETED here. */

  /* For each edge e = (i-1,i), see if crosses rays. */
  for( i = 0; i < n; i++ ) {
     /* First check if q = (0, 0) is a vertex. */
     if ( P[i][X]==0 && P[i][Y]==0 ) return 'v';
     i1 = ( i + n - 1 ) % n;

     /* Check if e straddles x axis, with bias above/below. */
     Rstrad = ( P[i][Y] > 0 ) != ( P[i1][Y] > 0 );
     Lstrad = ( P[i][Y] < 0 ) != ( P[i1][Y] < 0 );

     if( Rstrad || Lstrad ) {
        /* Compute intersection of e with x axis. */
        x = (P[i][X] * P[i1][Y] - P[i1][X] * P[i][Y])
        / (double)(P[i1][Y] - P[i][Y]);

        if (Rstrad && x > 0) Rcross++;
        if (Lstrad && x < 0) Lcross++;

     } /* end straddle computation */
  } /* end for */

  /* q on an edge if L/Rcross counts are not the same parity. */
  if( ( Rcross % 2 ) != (Lcross % 2 ) ) return 'e';

  /* q inside iff an odd number of crossings. */
  if( (Rcross % 2) == 1 ) return 'i';
  else return 'o';
}
```

Code 7.13 `InPoly1`. (Some portions shared with `InPoly0` are deleted above.)

Although it may not be obvious, these two expressions are logically equivalent: `Rstrad` is `TRUE` iff one endpoint of e is strictly above the x axis and the other is not (i.e., the other is on or below). This more concise form makes it easier to see the proper definition for `Lstrad`: Just reverse the inequalities to bias below. Now the computation of x is needed whenever either of these straddle variables is `TRUE`, which only excludes edges passing through $q = (0, 0)$ (and incidentally protects against division by 0). Finally, the

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

key determination of when to return e reduces to seeing if the two ray-crossing counts have different parity.

7.4.3. Exercises

1. *Point in convex polygon.* Design an algorithm to determine in $O(\log n)$ time if a point is inside a convex polygon.
2. *Worst ray crossings.* Could the ray crossing algorithm be made to run in $O(\log n)$ time for arbitrary query points?
3. *Division vs. multiplication* [programming]. On some machines (e.g., PCs), floating-point division can be as much as twenty times slower than multiplication. Modify the `InPoly1` code to avoid the division in Equation (7.9), and time it on examples to see if there is a significant difference on your machine.
4. *Tessellation by polygons.* Argue that in a partition of a region of the plane into polygons, `InPoly0` classifies a point q "in" at most one polygon.
5. *Speed-up* [programming]. Speed up `InPoly0` (Code 7.12) by avoiding the computation of x whenever the straddling segment is on the negative side of the ray.
6. *Avoid translation* [programming; easy]. Develop a new version of `InPoly1` (Code 7.13) that avoids the unnecessary translation of P and q.
7. *Integer ray crossing* [programming].
   a. Modify `InPoly0` (Code 7.12) to avoid the sole floating-point calculation. Use `AreaSign` (Code 4.23).
   b. Use the `AreaSign` results to decide if q lies on an edge, thereby achieving the functionality of `InPoly1` without shooting rays in both directions.
8. *Winding number* [programming]. Implement the winding number algorithm. The basic routine required is the angle subtended by a polygon edge e_i from the point q. The angle θ_i can be found from the cross product: $v_i \times v_{i+1} = |v_i||v_{i+1}| \sin\theta_i$. Recall that `Area2( q, P[i], P[i+1] )` from Chapter 1 (Code 1.5) is the magnitude of this cross product. The lengths of the vectors must then be divided out, the arcsine computed, and the angles summed over all i.

   Develop code for computing this angle sum. Pay particular attention to the range of angles returned by the `asin` library function, remembering that all counterclockwise turns must be positive angles, and clockwise turns negative angles. Decide what should be done when $|v_i| = 0$ or $|v_{i+1}| = 0$.

7.5. POINT IN POLYHEDRON

Determining whether a point is inside a polyhedron has many applications, including collision detection: Determining if a moving point (e.g., the tip of a tool) has penetrated an object in its environment is an instance. As in two dimensions, the problem is easy if the polyhedron is convex; in fact, the convex hull code in Chapter 4 solves this problem as the first step in `AddOne` (Code 4.15). The nonconvex case admits the same two solutions as in two dimensions: the generalization of the winding number computation and counting ray crossings.

Solid Angles

It is perhaps surprising that the winding number idea works in three dimensions as well as in two. It depends on a notion of signed *solid angle*, a measure of the fraction of a sphere surface consumed by a cone apexed at a point. It is measured in "steradians," which assigns 4π to the full-sphere angle. The solid angle of a tetrahedron with apex q and base T is the surface area of the unit sphere S falling within the tetrahedron when q is placed at the center of S, and the faces incident to q are extended (if necessary) to cut through S. The sign of the angle depends on the orientation of T. If the solid angles formed by q and every face of a polyhedron P are summed, the result is 4π if $q \in P$ and zero of $q \notin P$. This provides an elegant algorithm for point in polyhedron, which, alas, suffers the same pragmatic flaws as its two-dimensional counterpart: It is subject to numerical errors, and it is slow. A timing comparison between the ray-crossing code to be presented below and an implementation of the solid angle approach (Carvalho & Cavalcanti 1995) showed the latter to be twenty-five times slower. Their code is, however, much shorter.

Ray Crossings

The logic behind the ray-crossing algorithm in three dimensions is identical to that for the two-dimensional version: q is inside iff a ray from q to infinity crosses the boundary of P an odd number of times. A ray to infinity can be effectively simulated by a long segment qr, long enough so that r is definitely outside P. As we have worked out segment–triangle intersection in Section 7.3, it would seem easy to count ray crossings. The problematic aspect of this approach is to develop a scheme to count crossings accurately in the presence of the wide variety of possible degeneracies that could occur: qr could lie in a face of P, could hit a vertex, could collinearly overlap with an edge, hit an edge transversely, etc. It seems a proper accounting could be made, but I am not aware of any attempt in this direction. I leave this as an open problem (Exercise 7.5.2[1]).

Here we proceed on the basis of two observations. First, the crossings of a ray without degeneracies, which, for each face f of P, either misses f entirely or passes through a point in its relative interior, are easily counted. Second, "most" rays are nondegenerate in this sense, so a random ray is likely to be nondegenerate. Our plan is then to generate a random ray and check for degeneracies. If there are none, the crossing count answers the query. If a degeneracy is found, the ray is discarded and another random one chosen. Degeneracies can be detected with the `SegTriInt` code developed in Section 7.3 (Code 7.11). This leads to the pseudocode shown in Algorithm 7.1.

We now discuss the generation of the random ray. Let D be the the length of the diagonal of the smallest coordinate-aligned[13] "bounding box" B surrounding P. D is easily computed from the maximum and minimum coordinates of the vertices of P. Let $R = \lceil D \rceil + 1$. If a query point q is outside B, then it is outside P. For any query point inside B, a ray of length R from q must reach outside B (because D is the largest separation between any two points within B) and therefore outside P. We will use this value of R to guarantee that our query ray/segment qr reaches strictly outside P.

Generation of random rays of length R can be viewed as generating random points on the surface of a sphere of radius R. This is a well-studied problem, which we will not explore here.[14] The code `sphere.c` distributed with this book, and used to

[13] A coordinate-aligned object is often called *isothetic*.

[14] See O'Rourke (1997), Shoemake (1992), Arvo (1991), and Knuth (1969, p. 116, 485).

produce the 10,000 points in Figure 4.15, implements one method of generating such points. We will employ that as part of our point-in-polyhedron code without detailing it further.

Algorithm: POINT IN POLYHEDRON
Compute bounding radius R.
loop forever
    r_0 = random ray of length R.
    $r = q + r_0$.
    crossings = 0.
    for each triangle T of polyhedron P do
        `SegTriInt`(T, q, r).
        if degenerate intersection
            then Go back to loop.
            else Increment *crossings* appropriately.
    if *crossings* odd
        then q is inside P.
        else q is outside P.
    Exit.

Algorithm 7.1 Point in Polyhedron.

We make one last point before proceeding to code for the entire algorithm. The for-loop of Algorithm 7.1 calls `SegTriInt` for each face of P. Not only does the ray miss most faces of P, it misses them by a wide margin. This is a situation that calls for a quick miss-test, one that does not do all the (considerable) computation of `SegTriInt`. We will include in our implementation a very simple bounding-box test, as follows: As each face is read in (by `ReadFaces`, a simple routine not shown), a bounding box is computed and stored as two minimum and maximum corner points `tPointi Box[PMAX][2]`. Before committing to the full intersection test with face f, we first see if the query ray qr lies entirely to one side of one of the six faces of the box bounding f with a call to `BoxTest(f,q,r)` (Code 7.14). This returns '0' when nonintersection is guaranteed for this reason, and '?' otherwise. My testing shows that this simple rejection handles more than half of the intersection checks, well worth the slight overhead on the remaining half.

We now present `InPolyhedron`, code for testing if a query point q is in a polyhedron. We design it to return a code as follows:

'V': q coincides with a Vertex of P.
'E': q is in the relative interior of an Edge of P.
'F': q is in the relative interior of a Face of P.
'i' : q is strictly interior to P.
'o' : q is strictly exterior (outside) to P.

The codes {V, E, F} are inherited from the same codes returned by `SegTriInt`: Because we have ensured that r is strictly outside P, if qr has an endpoint on a vertex, edge, or face of P, then it must be the q endpoint. Thus we distinguish the on-boundary cases for

q without further effort. The codes {i, o} are distinguished by the parity of the crossings counter.

```
char BoxTest ( int n, tPointi a, tPointi b )
{
  int i;  /* Coordinate index */
  int w;

  for ( i=0; i < DIM; i++ ) {
      w = Box[ n ][0][i];  /* min: lower left */
      if ( (a[i] < w) && (b[i] < w) ) return '0';
      w = Box[ n ][1][i];  /* max: upper right */
      if ( (a[i] > w) && (b[i] > w) ) return '0';
  }
  return '?';
}
```

Code 7.14 BoxTest.

The overall structure of the main procedure InPolyhedron is shown in Code 7.15. First, query points outside the bounding box for P lead to an immediate return after the InBox(q, bmin, bmax) test; the simple code for InBox is not shown. Then a near-infinite loop adds a random ray to q to get r. (An upper limit is placed on the number of repetitions just as a matter of programming practice.) Next, qr is tested against every face of P, with a for-loop whose body is displayed separately (Code 7.16). If the for-loop runs to completion, then we are certain that the ray is generic, and the parity of crossings determines the result.

The for-loop (Code 7.16) first uses BoxTest hoping for a quick conclusion that the ray misses f, as discussed above. Otherwise SegTriInt is called and its return code used for subsequent decisions. Only if the code is 'f' (qr intersects the face in its relative interior) is the crossings counter incremented. The three codes {p, v, e} all indicate degeneracies: The ray lies in the plane of f or passes through a vertex or edge of f. We do not need further distinctions within the in-plane case 'p': Even if the ray misses f entirely, we are still safe in rejecting this as a degenerate case and awaiting a "better" ray. For all these degeneracies, the for-loop is abandoned, with control returning back to the infinite while-loop for another random ray. The codes {V, E, F} allow immediate exit, as discussed before.

Example: Cube. A simple example is shown in Figure 7.9. Here $q = (5, 5, 5)$ is at the center of a $10 \times 10 \times 10$ cube of 12 triangular faces. With $D = \sqrt{300}$, the ray radius is $R = 19$. The call to RandomRay results (in one particular trial) to $r = (23, 6, 11)$, which is well outside of P. The test against each of the 12 faces leads to 8 decided by BoxTest and 4 calls to SegTriInt, only one of which returns '1'. Thus there is exactly one ray crossing, and q is determined to be inside.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char InPolyhedron( int F, tPointi q,
     tPointi bmin, tPointi bmax, int radius )
{
  tPointi r;  /* Ray endpoint. */
  tPointd p;  /* Intersection point; not used. */
  int f, k = 0, crossings = 0;
  char code = '?';

  /* If query point is outside bounding box, finished. */
  if ( !InBox( q, bmin, bmax ) )
     return 'o';

  LOOP:
  while( k++ < F ) {
     crossings = 0;

     RandomRay( r, radius );
     AddVec( q, r, r );

     for ( f = 0; f < F; f++ ) { /* Begin check each face */
          /* Intersect ray with face f and increment crossings: see BELOW. */
     } /* End check each face */

     /* No degeneracies encountered: ray is generic, so finished. */
     break;

  } /* End while loop */

  /* q strictly interior to polyhedron iff an odd number of crossings. */
  if( ( crossings % 2 ) == 1 )
     return   'i';
  else return 'o';
}
```

Code 7.15 `InPolyhedron`. (`AddVec` is similar to `SubVec` in Code 7.6.)

Example: Nonconvex Polyhedron. A more stringent test is provided by the polyhedron P of $V = 400$ vertices and $F = 796$ (triangle) faces shown in Figure 7.10. Performance of the code was tested by generating random query points within the bounding box of P.[15] Out of 1,000,000 random rays generated, 8,121 (0.8%) were degenerate and caused the while-loop to try again. In 110 cases (0.01%) the loop again failed, and in only one instance of the one million trials did the loop generate three random rays before finding a generic one. Although the polyhedron can hardly be said to be "typical" (whatever that might mean), I do not expect performance to be significantly worse than this 99% "hit rate."

[15]The code `cube.c` that produced Figure 4.14 was used to generate the query points.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
for ( f = 0; f < F; f++ ) {   /* Begin check each face */
    if ( BoxTest( f, q, r ) == '0' )
         code = '0';
    else code = SegTriInt( Faces[f], q, r, p );

  /* If ray is degenerate, then goto outer while to generate another. */
  if ( code == 'p' || code == 'v' || code == 'e' ) {
     printf("Degenerate ray\n");
     goto LOOP;
  }

  /* If ray hits face at interior point, increment crossings. */
  else if ( code == 'f' ) {
  crossings++;
  printf( "crossings = %d\n", crossings );
  }

  /* If query endpoint q sits on a V/E/F, return that code. */
  else if ( code == 'V' || code == 'E' || code == 'F' )
     return( code );

  /* If ray misses triangle, do nothing. */
  else if ( code == '0' )
     ;

  else
  fprintf( stderr, "Error, exit(EXIT_FAILURE)\n" ),
  exit (1);

  /* End check each face */
  }
```

Code 7.16 For-loop of `InPolyhedron`.

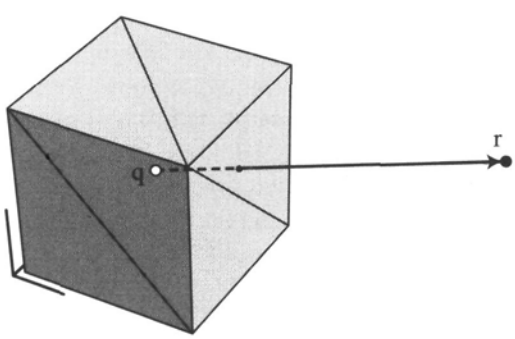

FIGURE 7.9 The ray qr intersects $\triangle(10, 10, 10), (10, 0, 10), (10, 10, 0)$ in its interior.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

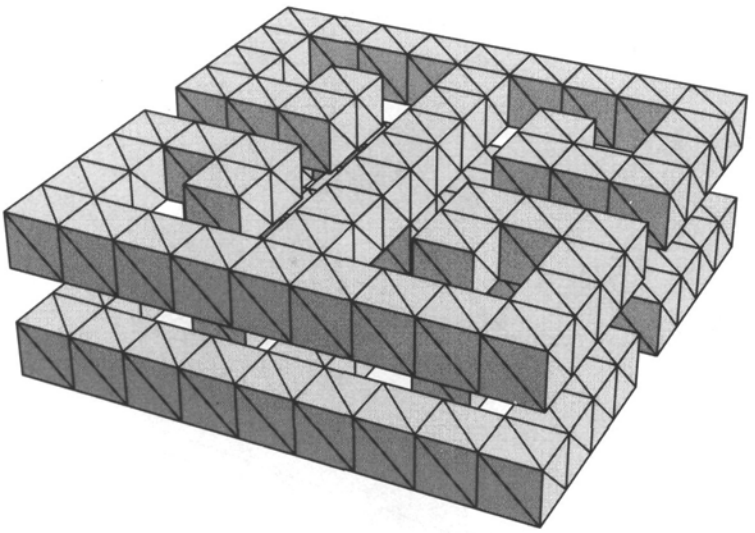

FIGURE 7.10 A polyhedron of $(V, E, F) = (400, 1194, 796)$ vertices, edges, and faces. The top and bottom "layers" are identical, connected by a single cubical channel in the middle layer. The polyhedron is symmetric about all three coordinate axis planes through the center of gravity.

7.5.1. Analysis

Algorithm 7.1 runs in expected time $O(\rho n)$, where ρ is the expected number of iterations before the while-loop finds a generic ray. Although we have just seen that for one combinatorially dense sample polyhedron, $\rho \approx 1.01$, it would be reassuring to prove a theoretical bound. I have not performed an exact analysis (Exercise 7.5.2[2]) but will offer an argument to show that $\rho = 1 + \epsilon$ can be achieved for any $\epsilon > 0$.

We start with two simplifying observations. First, it is easier to analyze random rays whose integer-coordinate tips fall on the surface of a surrounding cube C rather than a surrounding sphere. This is no loss of generality, as we could alter the implementation to follow this less aesthetically pleasing strategy (or choose a sphere large enough to include rays to all the cube surface points). Let each edge of the cube have length L.

Second, we need only concern ourselves with a degeneracy between q and an edge e of P. If q lies in the plane of a face, then there are rays qr that have a q–e degeneracy with edges of the face; and if a ray from q passes through a vertex, it passes through each (closed) incident edge. So let us just concentrate on one edge e of the polyhedron.

If e is close enough to q, then it "projects" to a segment that cuts completely across a face of the bounding cube C, as illustrated in Figure 7.11. In the worst case, the segment renders L integer points on that face of C unusable as ray tips, in the sense that they lead to degenerate rays.[16] If P has E edges, then at most EL points of a face of C can be rendered unusable. In effect, the edges of P produce an arrangement of lines on the

[16]"Renders" is particularly apropos here, because a line is rendered on a raster display by turning on L pixels. See Foley et al. (1990, Sec. 3.2).

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

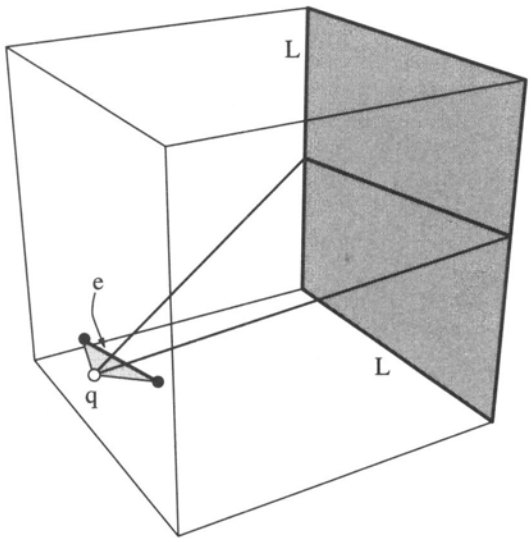

FIGURE 7.11 Edge e "kills" a line of points on the face of the surrounding cube in the sense that any r on that line makes a ray qr degenerate with (pass through) e.

cube face; only rays that miss the arrangement are safely generic. But there are many such, because this cube face contains L^2 integer points. Thus the probability of hitting a degeneracy is at most $EL/L^2 = E/L$. Because E is a constant (1,194 in Figure 7.10), choosing L large enough guarantees any $\epsilon = E/L$ desired.

There is pragmatic pressure in the other direction, however: The larger L is, the smaller the safe range of vertex coordinates before the onset of overflow problems. In practice I have found it best to choose R (which corresponds here to $L/2$) as small as possible, just 1 more than the box diagonal D.

7.5.2. Exercises

1. *Count degenerate crossings* [open]. Work out a scheme that counts ray crossings for any ray, taking into account all possible types of degeneracy. The parity of the count should determine if the point is in or out of the polyhedron. Test by fixing the qr ray in `InPolyhedron` (Code 7.15) to be parallel to the x axis.
2. *Sphere analysis* [open]. Compute a bound on the probability that an integer-coordinate ray tip on the surface of a "digital sphere" will degenerately pass through a vertex, edge, or face of the enclosed polyhedron P. Express your answer as a function of the sphere radius R, the diagonal D of a box surrounding P, and the combinatorial complexity of P (V, E, and F).

7.6. INTERSECTION OF CONVEX POLYGONS

The intersection of two arbitrary polygons of n and m vertices can have quadratic complexity, $\Omega(nm)$: the intersection of the polygons in Figure 7.12 is 25 squares. But the intersection of two convex polygons has only linear complexity, $O(n+m)$. Intersection of convex polygons is a key component of a number of algorithms, including determining whether two sets of points are separable by a line and for solving two-variable

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

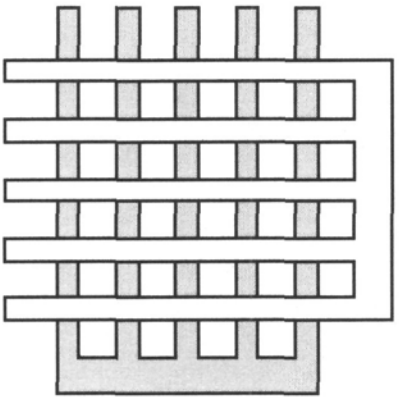

FIGURE 7.12 The intersection of two polygons can have quadratic complexity.

linear programming problems (Shamos 1978). The first linear algorithm was found by Shamos (1978), and since then a variety of different algorithms have been developed, all achieving $O(n + m)$ time complexity. This section describes one that I developed with three undergraduates, an amalgamation of their solutions to a homework assignment (O'Rourke, Chien, Olson & Naddor 1982). I feel it is the simplest algorithm available, but this is hardly an objective opinion.

The basic idea of the algorithm is straightforward, but the translation of the idea into code is somewhat delicate (as is often the case). Assume the boundaries of the two polygons P and Q are oriented counterclockwise as usual, and let A and B be directed edges on each. The algorithm has A and B "chasing" one another, adjusting their speeds so that they meet at every crossing of ∂P and ∂Q. The basic structure is as shown in Algorithm 7.2. A "movie" of the algorithm in action is shown in Figure 7.13.[17] The edges A and B are shown as vectors in the figure. The key clearly lies in the rules for advancing A and B, to which we now turn.

Algorithm: INTERSECTION OF CONVEX POLYGONS
Choose A and B arbitrarily.
repeat
    if A intersects B then
        Check for termination.
        Update an *inside* flag.
    Advance either A or B,
        depending on geometric conditions.
until both A and B cycle their polygons
Handle $P \cap Q = \emptyset$ and $P \subset Q$ and $P \supset Q$ cases.

Algorithm 7.2 Intersection of convex polygons.

[17]This figure was inspired by the animation of this algorithm provided in XYZ GeoBench.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

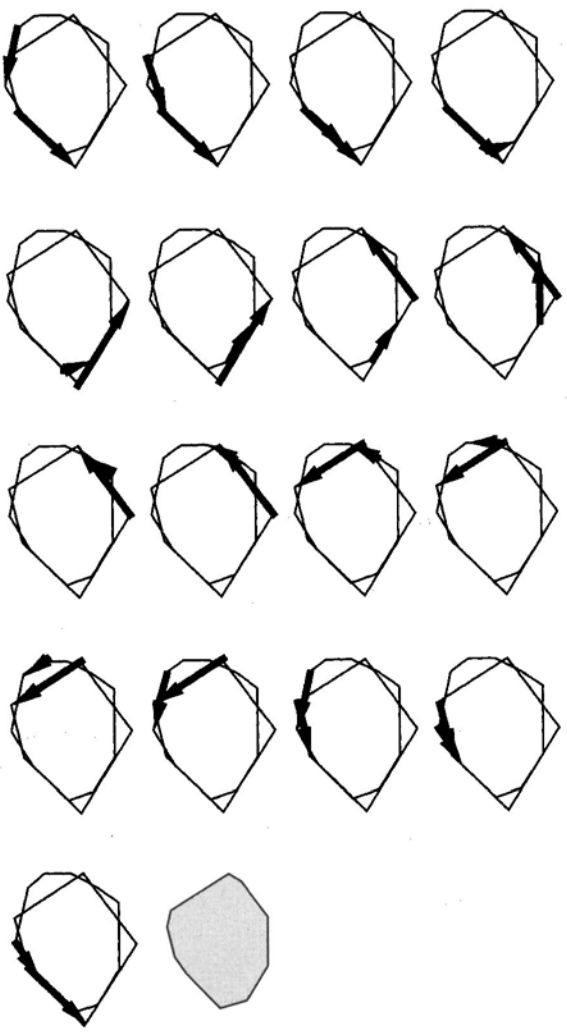

FIGURE 7.13 Snapshots of polygon intersection algorithm, sequenced left to right, top to bottom. This example is explored in more detail in Section 7.6.1.

Let a be the index of the head of A, and b the head of B. If B "aims toward" the line containing A, but does not cross it (as do all the solid vectors in Figure 7.14), then we want to advance B in order to "close in" on a possible intersection with A. This is the essence of the advance rules. The situations in the figure can be captured as follows: Let $H(A)$ be the open halfplane to the left of A. I will use the notation "$A \times B > 0$" to mean that the z coordinate of the cross product is > 0 (recall that this means that the shortest turn of A into B is counterclockwise):

if $A \times B > 0$ and $b \notin H(A)$, or
if $A \times B < 0$ and $b \in H(A)$,
then advance B.

(Let us ignore for the moment collinearities of $P[a]$ with B or $P[b]$ with A.) A similar

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

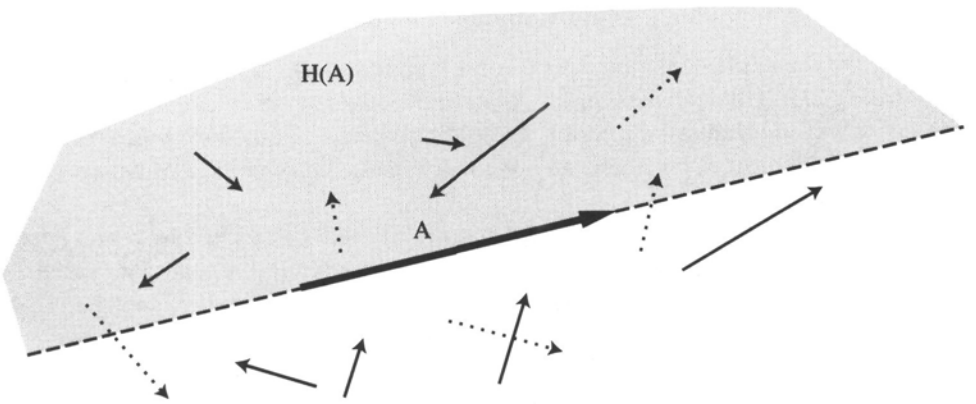

FIGURE 7.14 All the solid B vectors "aim" toward A; none of the dotted vectors do.

rule applies with the roles of A and B reversed (recall that $B \times A = -A \times B$):

$$\begin{aligned} &\text{if } A \times B < 0 \quad \text{and} \quad a \notin H(B), \text{ or} \\ &\text{if } A \times B > 0 \quad \text{and} \quad a \in H(B), \\ &\qquad \text{then advance } A. \end{aligned}$$

If both vectors aim toward each other, either may be advanced. When neither A nor B aim toward the other, we advance whichever is outside the halfplane of the other or either one if they are both outside. It takes some thought to realize that if both $a \in H(B)$ and $b \in H(A)$, then one must aim toward the other; so the above rules cover all cases. The cases may be organized in the following table:

$A \times B$	$a \in H(B)$	$b \in H(A)$	Advance Rule
>0	T	T	A
>0	T	F	A or B
>0	F	T	A
>0	F	F	B
<0	T	T	B
<0	T	F	B
<0	F	T	A or B
<0	F	F	A

These rules are realized by the following condensation, which exploits the freedom in the entries that are arbitrary.

$A \times B$	Halfplane Condition	Advance Rule
>0	$b \in H(A)$	A
>0	$b \notin H(A)$	B
<0	$a \in H(B)$	B
<0	$a \notin H(B)$	A

These are the advance rules we use below.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

7.6.1. Implementation

The core of the implementation is a do-while loop that implements the advance rules in the above table. Although the translation of the table is straightforward, I have not found a concise way to handle all the peripheral issues surrounding this core, which threaten to overwhelm the heart of the algorithm. We will discuss the generic cases before turning to special cases.

The polygon vertices are stored in two arrays P and Q, indexed by a and b, with n and m vertices, respectively. The three main geometric variables on which decisions are based are all computed with `AreaSign` (the sign-version of `Area2`): $A \times B$ is provided by `AreaSign( O, A, B )`, where `O` is the origin; $a \in H(B)$ is `AreaSign( Q[b1], Q[b], P[a] )`, with $b_1 = (b-1) \bmod n$; $b \in H(A)$ is a similar expression.

Code 7.17 shows the local variables, initialization, and overall structure. The central do-while is shown in Code 7.18, for generic cases only (corresponding to the above table). The special case Code 7.21 will be discussed later.

In Code 7.17, the variable `inflag` is an enumerated type that keeps track of which polygon is currently "inside"; it takes on one of the three values `{Pin, Qin, Unknown}`. Before the first intersection, its value is `Unknown`. After a crossing is detected between A and B, `inflag` remembers which one is locally inside just beyond the intersection point. This flag is used to output appropriate polygon vertices as the edge vectors are advanced. If `inflag` remains `Unknown` throughout a cycling of the counters, we know ∂P and ∂Q do not (properly) cross, and either they do not intersect, or they intersect only at a point, or one contains the other.

Termination of the loop is conceptually simple but tricky in practice. One could await the return of the first output point (Exercise 7.6.2[1]), but the version shown bases termination on the edge vector indices: When both a and b have cycled around their polygons, we are finished. In some cases of degenerate intersection, one of the indices does not cycle; so the while-statement also terminates when either has cycled twice after the first intersection, when the counters `aa` and `ba` (the suffix `a` stands for "advances") are reset.

Aside from the initialization details, the basic operations within the loop (Code 7.18) are: Intersect the A and B edge vectors with `SegSegInt`, print the intersection point p and toggle the `inflag` if they do intersect by a call to `InOut`, and finally, advance either a or b according to the advance rules, and perhaps print a vertex, by a call to `Advance`.

An intersection is considered to have occurred when `SegSegInt` returns a code of either '1' or 'v'; the code 'e,' indicating collinear overlap, will not toggle the flag, nor produce any output, except in a special case considered later. `InOut` (Code 7.19) prints the point of intersection and then bases its decision on how to set `inflag` according to which edge vector's head is inside the other vector's half plane. If the situation is not determined, the flag is not toggled.

The `Advance` routine (Code 7.20) advances the counters (`a` by return, `aa` by side effect) and prints out the vertex just passed if it was `inside`. Note that when `inflag`

```
void    ConvexIntersect( tPolygoni P, tPolygoni Q,
                         int n, int m )
                           /* P has n vertices, Q has m vertices. */
{
  int     a, b;            /* indices on P and Q (resp.) */
  int     a1, b1;          /* a-1, b-1 (resp.) */
  tPointi A, B;            /* directed edges on P and Q (resp.) */
  int     cross;           /* sign of z-component of A x B */
  int     bHA, aHB;        /* b in H(A); a in H(b). */
  tPointi Origin = {0,0}; /* (0, 0) */
  tPointd p;               /* double point of intersection */
  tPointd q;               /* second point of intersection (for 'e') */
  tInFlag inflag;          /* {Pin, Qin, Unknown}: which inside */
  int     aa, ba;          /* # advs. on a & b indices (after 1st inter.) */
  bool    FirstPoint;      /* Is this first point? (used to initialize).*/
  tPointd p0;              /* The first point. */
  int     code;            /* SegSegInt return code. */

  /* Initialize variables. */
  a = 0; b = 0; aa = 0; ba = 0;
  inflag = Unknown; FirstPoint = TRUE;

  do {

      /* BODY of do-while: see part (b) below. */

  /* Quit when both adv. indices have cycled, or one has cycled twice. */
  } while ( ((aa < n) || (ba < m)) && (aa < 2*n) && (ba < 2*m) );

  if ( !FirstPoint )   /* If at least one point output, close up. */
      LineTo( p0 );

  /* Deal with remaining special cases: not implemented. */
  if ( inflag == Unknown)
      printf("The boundaries of P and Q do not cross.\n");
}
```

Code 7.17 `ConvexIntersect`, part (a): top-level structure.

is set to, for example, `Pin`, it is not known that a is actually inside; only that just beyond the point of intersection, A is inside (except in a special case). But by the time `Advance` increments a, it is known that a is truly in the intersection.

Before proceeding to a discussion of special cases, we show the output of the code on a relatively generic example, shown in Figure 7.15. The code produces the following

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
do {
   /* Computations of key variables. */
   a1 = (a + n - 1) % n;
   b1 = (b + m - 1) % m;

   SubVec( P[a], P[a1], A );
   SubVec( Q[b], Q[b1], B );

   cross = AreaSign( Origin, A, B );
   aHB   = AreaSign( Q[b1], Q[b], P[a] );
   bHA   = AreaSign( P[a1], P[a], Q[b] );

   /* If A & B intersect, update inflag. */
   code = SegSegInt( P[a1], P[a], Q[b1], Q[b], p );
   if ( code == '1' || code == 'v' ) {
      if ( inflag == Unknown && FirstPoint ) {
         aa = ba = 0;
         FirstPoint = FALSE;
         p0[X] = p[X]; p0[Y] = p[Y];
         MoveTo_d( p0 );
      }
      inflag = InOut( p, inflag, aHB, bHA );
   }
   /*-----Advance rules-----*/
   /* SPECIAL CASES: see part (c) below */
   /* Generic cases. */
   else if ( cross >= 0 ) {
       if ( bHA > 0)
          a = Advance( a, &aa, n, inflag == Pin, P[a] );
       else
          b = Advance( b, &ba, m, inflag == Qin, Q[b] );
   }
   else /* if ( cross < 0 ) */ {
       if ( aHB > 0)
          b = Advance( b, &ba, m, inflag == Qin, Q[b] );
       else
          a = Advance( a, &aa, n, inflag == Pin, P[a] );
   }
} while ( ((aa < n) || (ba < m)) && (aa < 2*n) && (ba < 2*m) );
```

Code 7.18 `ConvexIntersect`, part (b): **do-while** loop. `MoveTo_d` prints a Postscript "moveto" command for `double` coordinates.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
tInFlag InOut( tPointd tp, tInFlag inflag, int aHB, int bHA )
{
  LineTo_d( p );

  /* Update inflag. */
  if       ( aHB > 0)
       return Pin;
  else if ( bHA > 0)
       return Qin;
  else     /* Keep status quo. */
       return inflag;
}
```

Code 7.19 `InOut`. `LineTo_d` prints a Postscript "lineto" command.

```
int    Advance( int a, int *aa, int n, bool inside, tPointi v )
{
  if ( inside )
     LineTo_i( v );
  (*aa)++;
  return (a+1) % n;
}
```

Code 7.20 `Advance`. `LineTo_i` prints a "lineto" command for `int`s.

Postscript output for these polygons:

```
 5.00    8.00  moveto
 5.00    8.00  lineto
13.00    0.00  lineto
19       2     lineto
24      10     lineto
24.00   21.33  lineto
17.80   29.60  lineto
14.67   31.17  lineto
 1.62   23.01  lineto
 0.72   19.12  lineto
 2.50   12.00  lineto
 5.00    8.00  lineto
```

The "movie" in Figure 7.13 shows the progression of A and B for this example. The example is not entirely generic, in that edge (1, 2) of P collinearly overlaps with (4, 0) of Q.

The alternation between integers and reals in the output reflects whether the point is a vertex (and printed in `Advance`) or a computed intersection point (and printed in `InOut`). The repeats of the point (5, 8) at the beginning and end of the list are an artifact of Postscript initialization and closure and could be removed easily. Below we will see more pernicious duplication of points.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

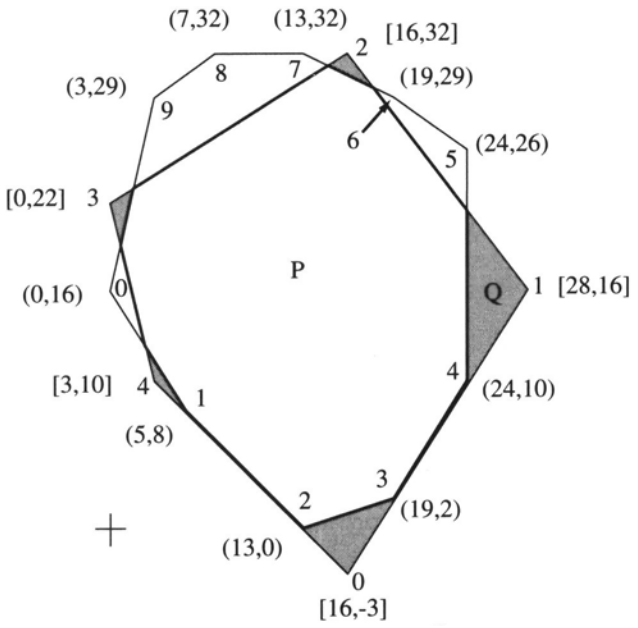

FIGURE 7.15 P is in front; Q is behind. P's vertex coordinates are in parentheses; Q's in brackets.

Special Cases

We finally turn to special cases, of which there seem to be a bewildering variety. See Figure 7.16 for a sampling of test examples.

The special cases all hinge on the special cases of the three geometric variables: $A \times B = 0$; when a lies on the line containing B; and when b lies on the line containing A. All three are indicated by returns of 0 from `AreaSign`. Several combinations are handled appropriately by the "generic cases" advance rules (Code 7.18), but three are isolated prior to that in Code 7.21:

1. If A and B overlap (return code 'e') and are oppositely oriented ($A \cdot B < 0$), then their overlap is the total intersection, for P and Q must lie on opposite sides of the line containing A and B. How the segment of intersection is found will be described in a moment.
2. If A and B are parallel ($A \times B = 0$) and a is strictly right of B and b is strictly right of A, then we may conclude $P \cap Q = \emptyset$.
3. If A and B are collinear (and case 1 above does not hold), then we advance one pointer, but we ensure that no point is output during the advance by arranging for the parameter `inside` to be `FALSE`.

Recall from Section 7.2 that our `SegSegInt` code returned only one point of intersection p even when the return code 'e' indicated overlap. Now that we need the actual overlap, we modify `SegSegInt` to return a second point q, computed in `ParallelInt`, such that pq is the segment of intersection. The modification to `ParallelInt` is shown in Code 7.22. Six possible cases of overlap are methodically identified, and p or q set appropriately: $cd \subseteq ab$; $ab \subseteq cd$; $c \in ab$ (two cases); and $d \in ab$ (two cases).

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

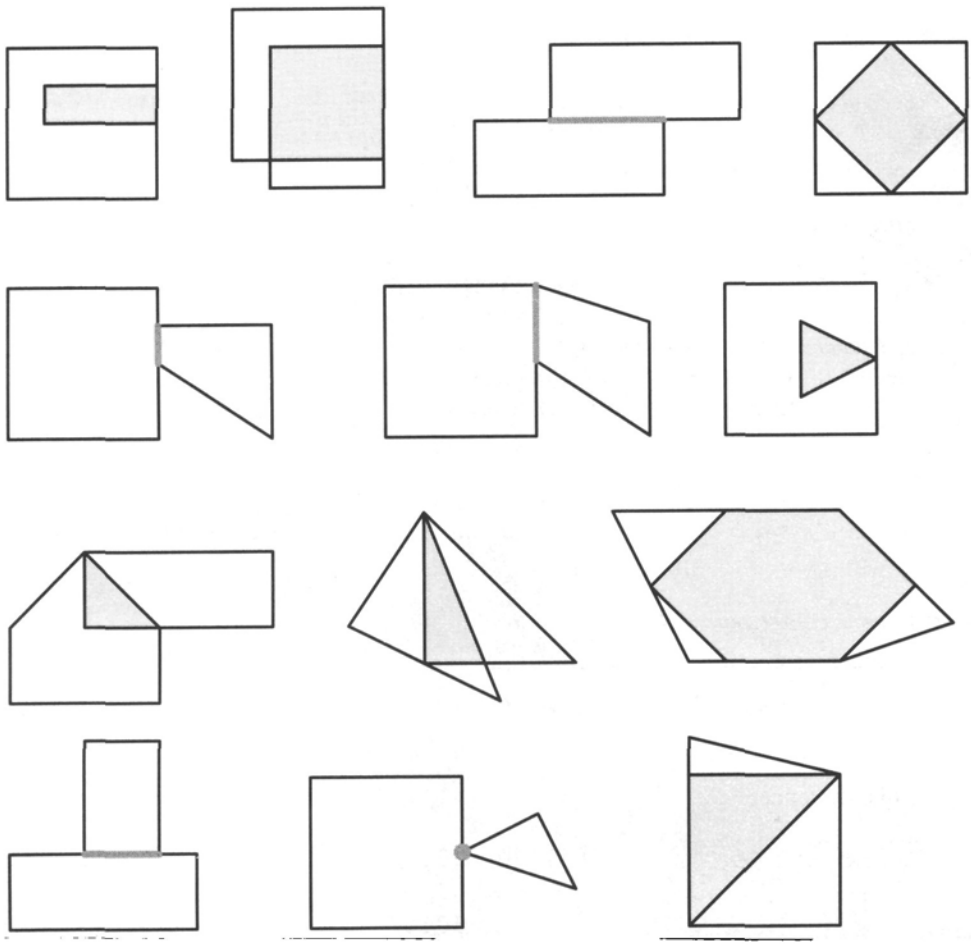

FIGURE 7.16 Output of the `ConvexIntersect` code on a variety of "degenerate" intersections. The intersection is shown shaded in each case.

We conclude with a litany of the weaknesses of the presented code and suggest improvements:

1. When the loop finishes with the `inflag` still `Unknown`, it could be that $P \subset Q$ or $Q \subset P$ or $P \cap Q = \emptyset$, or even that $P \cap Q = v$ where v is a vertex (the latter because `InOut` sometimes maintains the status quo). Handling these cases requires further code. None are all that difficult, but it would be preferable if they could be distinguished "automatically."
2. The loop termination, using two counters, is clumsy. Checking for a repeat of the first point is equally clumsy.
3. Although not evident from the example in Figure 7.15, degeneracies can cause points to be output more than once, sometimes both as vertices and as intersection points. For example, output from two nested squares sharing a corner included

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
/*....Advance rules (continued from part (a)) */

/* Special case: A & B overlap and oppositely oriented. */
if ( ( code == 'e' ) && (Dot( A, B ) < 0) )
      PrintSharedSeg( p, q ), exit(EXIT_SUCCESS);

/* Special case: A & B parallel and separated. */
if ( (cross == 0) && ( aHB < 0) && ( bHA < 0 ) )
      printf("%%P and Q are disjoint.\n"), exit(EXIT_SUCCESS);

/* Special case: A & B collinear. */
else if ( (cross == 0) && ( aHB == 0) && ( bHA == 0 ) ) {
        /* Advance but do not output point. */
        if ( inflag == Pin )
            b = Advance( b, &ba, m, inflag == Qin, Q[b] );
        else
            a = Advance( a, &aa, n, inflag == Pin, P[a] );
  }

/* Generic cases (continued in part (b) above)..... */
```

Code 7.21 `ConvexIntersect`, part (c): special cases.

this sequence:

```
0.00  50.00  lineto
0     50     lineto
0.00  50.00  lineto
```

Such duplicate points could wreak havoc with another program that expects all polygon vertices to be distinct. Although it is not difficult to suppress output of duplicates (Exercise 7.6.2[2]), this raises the integer versus float problem directly, for it will be necessary to decide, for example, if the integer 50 is equal to the floating-point number 50.00 where this latter number is computed with `doubles` in `SegSegInt`. An inexact calculation might result in identical points being considered distinct; use of a "fuzz" factor will inevitably enable acceptance of some distinct points as equal.

4. The biggest weakness is the need to handle many cases specially, often with delicate logic. It would be more satisfying to have the generic code specialize naturally. I leave this as an open problem (Exercise 7.6.2[4]).

7.6.2. Exercises

1. *Loop termination* (Peter Schorn) [programming]. Modify the code so that loop termination depends on cycling past the first output vertex, rather than on the loop counters `aa` and `ba`. Test your code on the two triangles (3, 4), (6, 4), (4, 7) and (4, 7), (2, 5), (6, 2).

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

```
char   ParallelInt( tPointi a, tPointi b, tPointi c, tPointi d,
                    tPointd p, tPointd q )
{
  if ( !Collinear( a, b, c) )
     return '0';

  if ( Between( a, b, c ) && Between( a, b, d ) ) {
     Assigndi( p, c );
     Assigndi( q, d );
     return 'e';
  }
  if ( Between( c, d, a ) && Between( c, d, b ) ) {
     Assigndi( p, a );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, c ) && Between( c, d, b ) ) {
     Assigndi( p, c );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, c ) && Between( c, d, a ) ) {
     Assigndi( p, c );
     Assigndi( q, a );
     return 'e';
  }
  if ( Between( a, b, d ) && Between( c, d, b ) ) {
     Assigndi( p, d );
     Assigndi( q, b );
     return 'e';
  }
  if ( Between( a, b, d ) && Between( c, d, a ) ) {
     Assigndi( p, d );
     Assigndi( q, a );
     return 'e';
  }
  return '0';
}
```

Code 7.22 `ParallelInt`; See Code 7.3 for `Assigndi`.

2. *Duplicate points* [programming]. Modify the code to suppress the output of duplicate points. Do not concern yourself with the float versus integer problem discussed above, but rather just compare `int`s and `double`s with ==, which will force a conversion to `double`s.

   Once you have your code working, try to find a case that will break it. This may be quite difficult, and even impossible on some machines.

3. *Rationals.*
   a. Discuss (but do not implement) how the point of intersection could be represented as an exact rational, a ratio of two integers.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

b. Design a Boolean function that determines if two such rationals are equal. Note that, e.g., $2/6 = 127131/381393$.

c. [programming] Implement the rational equal function.

4. *Clean code* [open]. Design a set of advance rules that handle the special cases more cleanly than does the presented code. Ideally all the cases in Figure 7.16, as well as $P \subset Q$, $Q \subset P$, and $P \cap Q = \emptyset$, would be handled naturally.

7.7. INTERSECTION OF SEGMENTS

Although we have seen that $\Omega(n^2)$ is a lower bound on intersecting two polygons of n edges each, in many applications the worst case is rare. This suggests a goal of developing an output-size sensitive algorithm, one whose complexity depends on k, the size of (the number of vertices in) the output.[18] It turns out that the hard part of this task is a more general problem: finding the intersections among a collection of n segments in the plane. This is more general in that no assumption is made that the segments connect to form polygons. We will now pursue the segment intersection problem, presenting an elegant algorithm due to Bentley & Ottmann (1979), one of the first output-size sensitive algorithms in computational geometry, and return in Section 7.8 to the problem of intersecting polygons.

A brute-force intersection algorithm takes $\Omega(n^2)$ time: Check each segment against every other (using, e.g., `SegSegInt` from Section 7.2). To achieve output sensitivity, we want to compute intersections between only those pairs of segments that actually intersect. This goal sounds circular, but even this formulation carries a hint of a solution, for segments that intersect are close to one another – not throughout their length, but certainly at their point of intersection! If we could somehow "travel down" the lengths of a pair of segments until they become close to one another before deciding to intersect, we could achieve output sensitivity. Plane sweep provides the needed "travel down" mechanism.

Recall from Section 2.4 that a trapezoidalization could be computed efficiently by sweeping a horizontal line L over a collection of polygon edges, and only searching for chord intersections in the local neighborhood of a vertex hit by L. It is just this sort of local focus we need for the segment intersection problem.

Imagine sweeping the line L over a collection of segments $S = \{s_0, s_1, \ldots, s_{n-1}\}$. Let $x = s_i \cap s_j$ be an intersection point between two segments. Just before L reaches x, L pierces both s_i and s_j, and they are adjacent along L: No other segment is between them on L. Thus, at some time prior to every intersection "event" (when L crosses an intersection point), the intersecting segments are adjacent on L. This gives us the sought-for locality: Computing intersections between segments adjacent on L suffices to capture all intersection points. Some of these adjacent segments do not in fact intersect, but we will see that the "wasted effort" is small.

Let us make several simplifying assumptions to keep focused on the main idea: Assume no segment is horizontal and no three segments pass through one point. The plan is to sweep L over the segments, stopping at events of three types, when

[18] See Sections 3.3, 4.6.4, and 6.7.2 for other output-size sensitive algorithms.

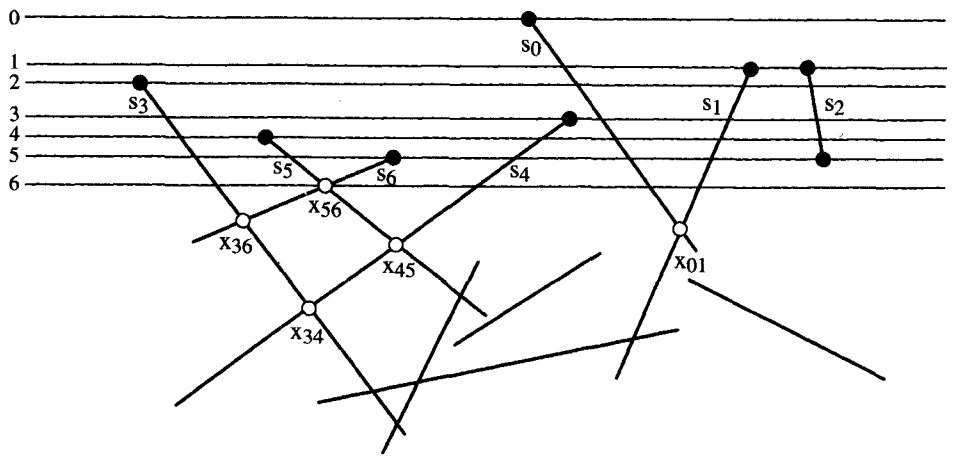

FIGURE 7.17 Bentley–Ottmann sweepline algorithm. Endpoint events are shown as solid circles; intersection points x_{ij} computed by position 6 of L are shown as open circles.

1. the top endpoint of a segment is hit,
2. the bottom endpoint of a segment is passed, or
3. an intersection point between two segments is reached.

All three of these events cause the list $\mathcal{L}$ of segments pierced by L to change: A segment is inserted, deleted, or two adjacent segments switch places, respectively. With each change, intersections between newly adjacent segments must be computed.

Although segments must become adjacent in $\mathcal{L}$ prior to their point of intersection x, it is not guaranteed that x is the next intersection event when it is computed. Rather, the intersection events must be placed in a queue Q sorted by height, along with the segment endpoints. An illustration should make the algorithm clear. Consider the set of segments $S = \{s_0, s_1, \ldots\}$ shown in Figure 7.17. Let a_i be the upper endpoint of segment s_i, and b_i its lower endpoint. Then the event queue is initialized to $Q = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, b_2, \ldots)$, all the segment endpoints sorted top to bottom. When L reaches a_1 and a_2 (position 1), s_0 and s_1 become newly adjacent, and their intersection point x_{01} is added to the queue after b_2. s_1 and s_2 are also newly adjacent but do not intersect. Note that the higher intersection point x_{56} has not yet been constructed. At position 2, L hits a_3; the newly adjacent segments s_3 and s_0 do not intersect. At this point $\mathcal{L} = (s_3, s_0, s_1, s_2)$. At position 3, L hits a_4, and intersection point x_{34} is added to Q at its appropriate location. Note that three intersection events "between" s_3 and s_4 will be encountered before x_{34} is reached. By the time L reaches the first intersection event at position 6, all the endpoints above have been processed, and $\mathcal{L} = (s_3, s_5, s_6, s_4, s_0, s_1)$. This event causes s_5 and s_6 to switch places in $\mathcal{L}$, introducing new adjacencies that result in x_{36} and x_{45} being added to Q. Q now contains all the circled intersection points shown in the figure.

The algorithm needs to maintain two dynamic data structures: one for $\mathcal{L}$ and one for Q. Both must support fast insertions and deletions in order to achieve an overall

low time complexity. We will not pursue the data structure details[19] but only claim that balanced binary trees suffice to permit both $\mathcal{L}$ and Q to be stored in space proportional to their number of elements m, with all needed operations performable in $O(\log m)$ time. We now argue that such structures lead to a time complexity for intersecting n segments of $O((n+k)\log n)$, where k is the number of intersection points between the segments. We will continue to assume that no three segments meet in one point.

The total number of events is $2n + k = O(n + k)$: the $2n$ segment endpoints and the k intersection points. Thus the length of Q is never more than this. Because each event is inserted once and deleted once from Q, the total cost of maintaining Q is $O((n + k)\log(n + k))$. Because $k = O(n^2)$, $O(\log(n + k)) = O(\log n + 2\log n) = O(\log n)$. Thus maintaining Q costs $O((n + k)\log n)$.

The total cost of maintaining $\mathcal{L}$ is $O(n \log n)$: n segments inserted and deleted at $O(\log n)$ each. It only remains to bound the number of intersection computations (each of which can be performed in constant time, by a call to `SegSegInt`, Code 7.2). Recall the earlier worry about "wasted effort." However, the number of intersection calls is at most twice the number of events, because each event results in at most two new segment adjacencies: an inserted segment with its new neighbors, two new neighbors when the segment between is deleted, and new left and right neighbors created by a switch at an intersection event. Thus the total number of intersection calls is $O(n + k)$.

The overall time complexity of the algorithm is therefore $O((n+k)\log n)$, sensitive to the output size k. We have seen that the space requirements are $O(n + k)$ because this is how long Q can grow. It turns out that this can be reduced to $O(n)$ (Exercise 7.8.1[2]). Moreover, both of these desirable complexities can be achieved without any of our simplifying assumptions (Exercise 7.8.1[1]).

These results were achieved by 1981 (Bentley & Ottmann 1979; Brown 1981), but more than a decade of further work was needed to reach an optimal algorithm in both time and space:

Theorem 7.7.1. *The intersection of n segments in the plane may be constructed in* $O(n \log n + k)$ *time* (Chazelle & Edelsbrunner 1992) *and* $O(n)$ *space* (Balaban 1995), *where k is the number of intersection points between the segments.*

Here k is not multiplied by $\log n$ as in the original Bentley–Ottmann algorithm. The practical difference may be slight, but closing the theoretical gap required the development of new techniques.

7.8. INTERSECTION OF NONCONVEX POLYGONS

It is not difficult to alter the Bently–Ottmann sweepline algorithm to compute the intersection of two polygons. Let the two polygons be A and B, with vertices labeled a_i and b_j respectively. The main idea is similar to that used by scan-line algorithms for filling (painting) a polygonal region on a graphics screen[20] and is related to our ray-crossing

[19] See de Berg, van Kreveld, Overmars & Schwarzkopf (1997, Sec. 2.1), Preparata & Shamos (1985, Sec. 7.2.3), or Mehlhorn (1984, Sec. VIII. 4.1).

[20] See, e.g., Foley, van Dam, Feiner, Hughes & Phillips (1993, Sec. 3.5).

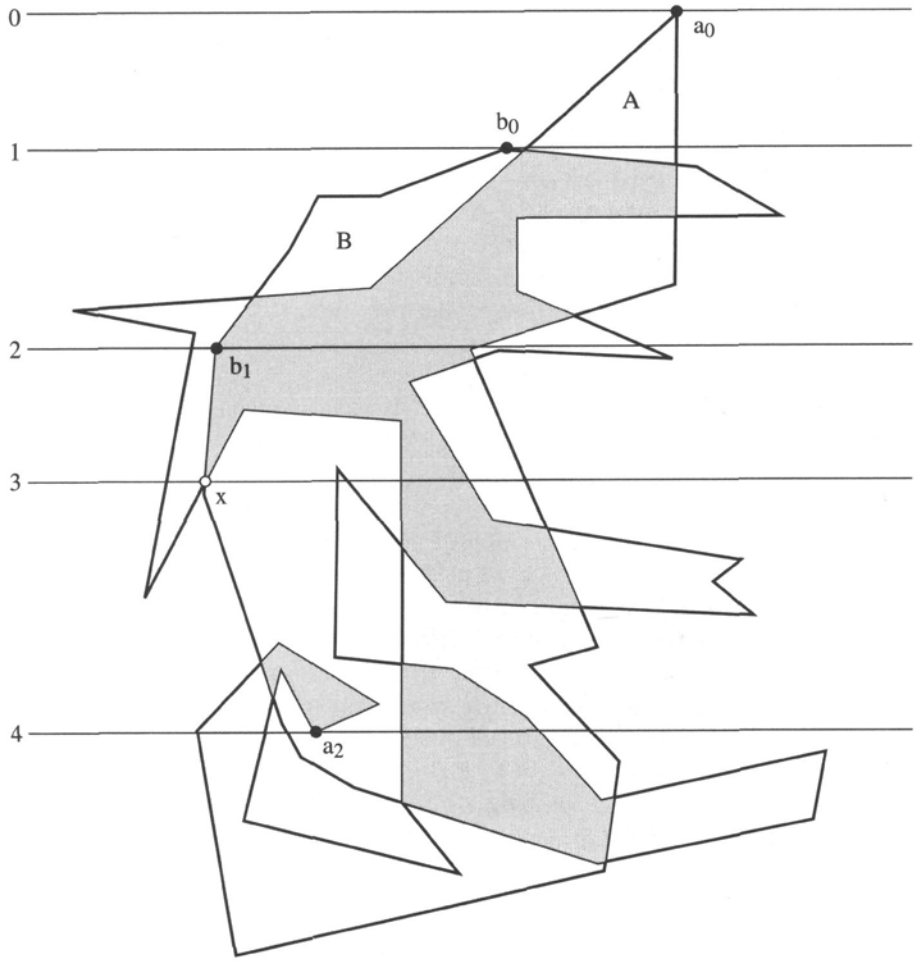

FIGURE 7.18 Intersection of two polygons: $A \cap B$ is shaded darkest.

analysis in Section 7.4. One maintains along the length of the sweep line L a "status" indicator, which has the following value:

$\emptyset$: exterior to both polygons;
A: inside A, but outside B;
B: inside B, but outside A; or
AB: inside both A and B.

The status is recorded for the span between each two adjacent segments pierced by L; clearly it is constant throughout each span.

Consider the example shown in Figure 7.18. When L is at position 2 (event b_1), the left-to-right status list is $(\emptyset, A, AB, B, \emptyset)$. This information can be easily stored in the same data structure representing L. We will not delve into the data structure details, but rather sketch how the status information can be updated in the same sweep that processes the segment intersection events, using the example in Figure 7.18.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

At position 0, when L hits a_0, the fact that both A-edges are below a_0 indicates that we are inserting an A-span. At position 1, a B-span is inserted. Just slightly below b_0, an intersection event opens up an AB-span, easily recognized as such because the intersecting segments each bound A and B from opposite sides, with A and B below. At position 3, intersection event x, the opposite occurs: The intersecting segments each bound A and B above them. Thus an AB-span disappears, replaced by an $\emptyset$-span between the switched segments. At a_2 (position 4), the inverse of the a_0 situation is encountered: The A-edges are above, and an A-span is engulfed by the surrounding B-spans. Although we have not provided precise rules (Exercise 7.8.1[5]), it should be clear that the span status information may be maintained by the sweepline algorithm without altering the asymptotic time or space complexity.

Although this enables us to "paint" the intersection $A \cap B$ on a raster display, there is a further step or two to obtain lists of edges for each "polygonal" piece of $A \cap B$. The reason for the scare quotes around "polygonal" is that the intersection may include pieces that are degerate polygons: segments, points, etc. – what are sometimes collectively called "hair." Whether this is desired as part of the output depends on the application. This issue aside, there is still further work. For example, at position 3 in Figure 7.18, an AB-span disappears at x, but the polygonal piece that disappears locally at x continues on to lower sweepline positions elsewhere. Two AB-spans may merge, revealing that what appeared to be two separate pieces above L are actually joined below.

This aspect of the algorithm may be handled by growing polygonal chain boundaries for the pieces of the intersection as the sweepline progresses and then joining these pieces at certain events. Thus position 3 in the figure is an event that initiates joining a left-bounding AB-chain with a right-bounding AB-chain. Keeping track of the number of "dangling endpoints" of a chain permits detection of when a complete piece of the output has been passed: For example, at position 4 of the figure, a_2 closes up the chain and an entire piece can be printed, whereas at position 3, the chain joined at x remains open at its rightmost piercing with L. Again we will not present details.

Finally, it is easy to see that we could just have easily computed $A \cup B$, or $A \setminus B$, or $B \setminus A$ – the status indicator is all we need to distinguish these. Thus all "Boolean operations" between polygons may be constructed with variants of the Bentley–Ottmann sweepline algorithm, in the same time complexity. These Boolean operations are the heart of many CAD/CAM software systems, which, for example, construct complex parts for numerically controlled machining by subtracting one shape from another, joining shapes, slicing away part of a shape, etc., all of which are Boolean operations.

Theorem 7.8.1. *The intersection, union, or difference of two polygons with a total of n vertices, whose edges intersect in k points, may be constructed in $O(n \log n + k)$ time and $O(n)$ space.*

7.8.1. Exercises

1. *Handling degeneracies.*
   (a) [easy] Show that horizontal segments can be accommodated within the presented algorithm without increasing time or space complexity.

(b) [more difficult] Show that permitting many segments to pass through one intersection point does not lead to greater time or space complexity.

2. *Reducing the space requirements.* Show that the following strategy of Pach & Sharir (1991) (see also de Berg et al. (1997, p. 29)) reduces the space requirements of the Bentley–Ottmann algorithm to $O(n)$ without increasing its time complexity. Delete an intersection point, such as x_{34} in Figure 7.17, from Q whenever its generating segments cease being adjacent because another segment (s_5 in this example) is encountered between them. The same intersection point is recomputed and reinserted into Q later.
3. *Intersection of segments: implementation* [programming]. Using `SegSegInt` (Code 7.2) as a subroutine, implement the Bentley–Ottmann algorithm with naive data structures.
4. *Degenerate intersections.*
   (a) [easy] Show by example that $A \cap B$ can be a path of segments (i.e., a polygonal chain).
   (b) Prove that no connected component of the intersection of two simple polygons can be topologically equivalent to the union of three segments forming a "Y"-shape.
5. *Span status rules.* Detail the rules for updating the span status information (Section 7.8) for the various events that could occur during a sweep of two polygons.
6. *Polygon simplicity* [easy]. Prove that the Bentley–Ottmann algorithm may be used to detect whether a given list of n points form a simple polygon in time $O(n \log n)$.

7.9. EXTREME POINT OF CONVEX POLYGON

It is frequently necessary to find a boundary point of a convex polygon extreme in a certain direction. For example, the smallest box enclosing a polygon, where the box sides are aligned with the coordinate axes, can be constructed from extreme points in the four compass directions. Although often this computation is performed by a simple $O(n)$ scan of all vertices, it is not surprising that a minor variant of binary search will accomplish the same goal in $O(\log n)$ time. In this section we will sketch such a search algorithm to find a highest point and then generalize to an extreme in a particular, arbitrary direction.

Let the n polygon vertices be $P[0], \ldots, P[n-1]$, labeled counterclockwise. Suppose at some point of the search we know a highest vertex is counterclockwise between indices a and b. We will represent the collection of these indices, our search interval, by $[a, b]$. So if $a < b$, one of $P[a], P[a+1], \ldots, P[b-1], P[b]$ is a highest vertex. For this initial sketch, we will not worry about wraparound through index 0, nor will we be concerned with the possibility that two vertices are equally highest, although both of these issues complicate implementations.

The main idea is to use the directed edges of the polygon to decide how to halve the search interval. Let c be an index strictly between a and b. If the edge A after $P[a]$ points upward, then a is on the right chain of P. If in addition the edge C after $P[c]$ points downward, then c is on the left chain, and we have the situation illustrated in Figure 7.19(a): The highest point is between. In this case we may shorten the original search interval $[a, b]$ to $[a, c]$. A similar shortening occurs if A points downward and C upward ((b) of the figure), or if A and C both point upward ((c) and (d) of the figure), or if A and C both point downward (not shown). This halving process is repeated until the edge after c points down and the edge before it points up,

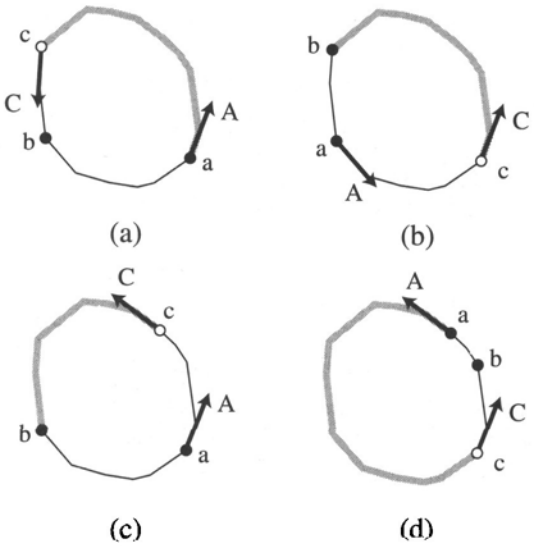

FIGURE 7.19 Four cases for finding a highest point. The $[a, b]$ interval is shortened to the shaded chain in each case.

indicating that c is highest. The pseudocode in Algorithm 7.3 shows the details of the decisions.

Algorithm: HIGHEST POINT OF CONVEX POLYGON
Initialize a and b.
repeat forever
    $c \leftarrow$ index midway from a to b.
    if $P[c]$ is locally highest then return c
    if A points up and C points down
        then $[a, b] \leftarrow [a, c]$
    else if A points down and C points up
        then $[a, b] \leftarrow [c, b]$
    else if A points up and C points up
        if $P[a]$ is above $P[c]$
            then $[a, b] \leftarrow [a, c]$
            else $[a, b] \leftarrow [c, b]$
    else if A points down and C points down
        if $P[a]$ is below $P[c]$
            then $[a, b] \leftarrow [a, c]$
            else $[a, b] \leftarrow [c, b]$

Algorithm 7.3 Highest point of convex polygon.

Three points require further clarification:

1. How is the midway index c computed?
2. How can the loop termination be implemented?
3. How does the possibility that two vertices are equally highest affect the algorithm?

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

Let us tackle the first problem: how to find an index midway between a and b. If $a < b$, then $(a + b)/2$ is midway. Note that if $b = a + 1$, then $(a + b)/2 = a$, due to truncation. If $a \geq b$, the interval $[a, b]$ includes 0, and the formula $(a + b)/2$ no longer works. For example, let $n = 10$, $a = 7$, and $b = 3$. Here $[7, 3] = (7, 8, 9, 0, 1, 2, 3)$, and the midpoint is 0. This can be computed by shifting b by n so that it is again larger than a, and taking the result $\bmod n$: $((a + b + n)/2) \bmod n$. In our example, $((7 + 3 + 10)/2) \bmod 10 = 0$. Note that if $a \geq b$ and $b = (a + 1) \bmod n$, then again the computation yields a, which is the same behavior as when $a < b$: When a and b are adjacent, the midpoint is a.

This gives us a midway function that could be implemented as shown in Code 7.23. Note what this function yields for the midpoint of $[a, a]$, which should represent the entire boundary of P: $(a + n/2) \bmod n$, halfway around from a, exactly as desired.

Loop termination is easy if there is a uniquely highest vertex: Then the vertices adjacent to c are both strictly lower. Capturing the situation where a horizontal edge is

```
int     Midway( int a, int b, int n )
{
  if (a < b) return ( a + b ) / 2;
  else       return ( ( a + b + n ) / 2 ) % n;
}
```

Code 7.23 `Midway`.

highest (or several collinear horizontal edges if the input permits this) is not much more difficult: Neither vertex adjacent to c is higher.

Unfortunately, we cannot be guaranteed that c will ever hit an extreme vertex, due to the truncation in `Midway` when $b = a + 1$. This always truncates clockwise, which can block a last needed counterclockwise step. Termination can be ensured by capturing the $c = a$ case specially. We leave a full implementation to Exercise 7.9.2[2] and turn now to a generalization and an application.

It is easy to alter the algorithm to find an extreme in an arbitrary direction u: Each test that a vector V points downward is replaced by the test $u \cdot V < 0$, each test that $P[a]$ is above $P[c]$ is replaced by the test that $u \cdot (P[a] - P[c]) > 0$, and so on. This permits using the extreme-finding algorithm for more than just bounding box calculations.

7.9.1. Stabbing a Convex Polygon

The problem of finding the intersection of a geometric object with a line is often called the "stabbing" problem. Here we show how the extreme-finding algorithm can be used to stab a convex polygon in $O(\log n)$ time.

Let P be the polygon and L the given line, and let u be a vector orthogonal to L. Find two vertices of P extreme in the $+u$ and $-u$ directions; call these a and b. See Figure 7.20. If both a and b are to one side of L, $L \cap P = \emptyset$. Otherwise a and b split ∂P into two chains whose intersections with L can be found by straightforward binary search: Chain $P[a, b]$ will yield intersection point x in the figure, and $P[b, a]$ will yield y.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

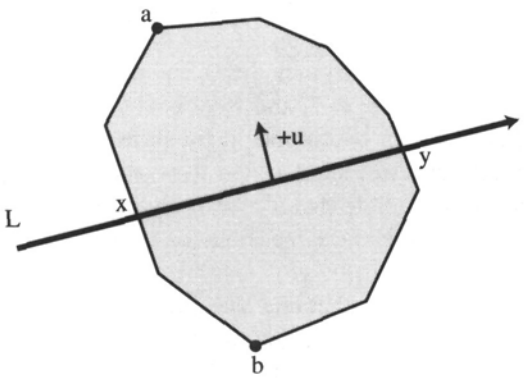

FIGURE 7.20 Stabbing a convex polygon.

7.9.2. Exercises

1. *Collinear points.* Suppose the input polygon contains three or more consecutive collinear vertices. Does this present a problem for Algorithm 7.3?
2. *Implement extremes algorithm* [programming]. Implement Algorithm 7.3, generalized to arbitrary directions u. Test on examples that have an extreme edge.
3. *Line–polygon distance.* Design an algorithm to determine the distance between an arbitrary polygon P of n vertices and a query line L. Define the distance to be

$$\min_{x,y}\{|x - y| \ : \ x \in P \, , \ y \in L\},$$

where x and y are points. Try to achieve $O(\log n)$ per query after preprocessing.

7.10. EXTREMAL POLYTOPE QUERIES

The problem of finding an extreme point of a polytope is much more difficult than the two-dimensional version covered in the previous section. There is no direct counterpart to the one-dimensional search we used on the boundary chain of the convex polygon: The two-dimensional surface of a polytope provides too much freedom in the search direction. Nevertheless, Kirkpatrick (1983) invented a breathtakingly beautiful search structure that permits the problem to be solved in $O(\log n)$ query time, asymptotically the same as in two dimensions (although we will see that the constant of proportionality is larger).

7.10.1. Sketch of Idea

The key idea is to form a sequence of simpler and simpler polytopes nested within the original given polytope P.[21] The innermost polytope is a tetrahedron or triangle, and there are $O(\log n)$ polytopes altogether. Construction of the hierarchy of polytopes can

[21] This sequence is often called the Dobkin–Kirkpatrick hierarchy; see Dobkin & Kirkpatrick (1990).

be done in $O(n)$ time, and storing all of them only uses $O(n)$ space. Once they are constructed, extremal queries can be answered in $O(\log n)$ time. Note that although this matches the time complexity for finding extreme points of convex polygons, the polygons did not require preprocessing (although even to read such a polygon into memory requires $O(n)$ time, which can be considered a crude form of preprocessing).

An extremal query is answered by first finding the extreme for the innermost polytope, and using that to work outwards through the hierarchy toward P. Let the sequence of nested polytopes be $P = P_0, P_1, P_2, \ldots, P_k$, where P_k is the innermost. And let a_i be the extreme point for polytope P_i. We first find the extreme point a_k of P_k by comparing its three or four vertices. Knowing a_k (and some other information) will give us a small set of candidate vertices of P_{k-1} to check for extremality. This yields a_{k-1}, and from that we find a_{k-2}, and so on. It will turn out that the work to move from one polytope to the next in the hierarchy is constant. Because $k = O(\log n)$, the total time to find a_0 is also $O(\log n)$. We now proceed to detail the search structure and the algorithm.

7.10.2. Independent Sets

Recall that the edges and vertices of a polytope form a planar graph (Section 4.1.4); Figure 7.21(a) shows the graph for an icosahedron, Figure 7.22, an example we will use to illustrate ideas throughout this section. Kirkpatrick's key idea depends on the graph theory notion of an "independent set." A set of nodes I of a graph G is called an *independent set* if no two nodes in I are adjacent in G. Thus they are "spread out" in a sense. Such an independent set is marked in Figure 7.21(a). This set of three nodes is in fact a *maximum independent set* for this graph, in that no four nodes form an independent set. It is important for Kirpatrick's scheme that planar graphs have "large" independent sets composed entirely of vertices of "small" degree (i.e., a small number of adjacent nodes); these vague qualifiers will be made precise later.

The construction of P_1, the first polytope nested inside $P = P_0$, proceeds as follows. An independent set of vertices for P_0 is found as in Figure 7.21(a). These vertices, and all their incident edges, are deleted from the graph. The result is shown in Figure 7.21(b). Because the vertices are independent, each deletion produces one new face in the graph. In Figure 7.21(b), each deletion produces a pentagon (which looks like a quadrilateral because two edges are collinear in the drawing). Next, these faces are triangulated; see Fig 7.21(c). In our case we can triangulate them arbitrarily; more on this is discussed in Section 7.10.4. The geometric equivalent to this operation on polytopes is to delete the vertices in the independent set and take the convex hull of the remaining vertices. This produces polytope P_1, which is clearly nested inside P_0, since it is the hull of a subset of P_0's vertices. Figure 7.23 shows P_1 corresponding to the graph in Figure 7.21(c). Note that the pentagons (two of which are visible in the figure) are comprised of three coplanar triangles. In general the vertices adjacent to a deleted independent vertex will not be coplanar; they are in this instance because of the symmetry of the icosahedron. It is the coplanarity and convexity of the face that permitted us to triangulate it arbitrarily. In general we would have to take the hull of the vertices around the boundary of the new face to construct the triangulation.

Now the process is repeated to construct P_2. A set of independent vertices of P_1 are identified, as marked in Figure 7.21(c). These are deleted, producing the graph shown

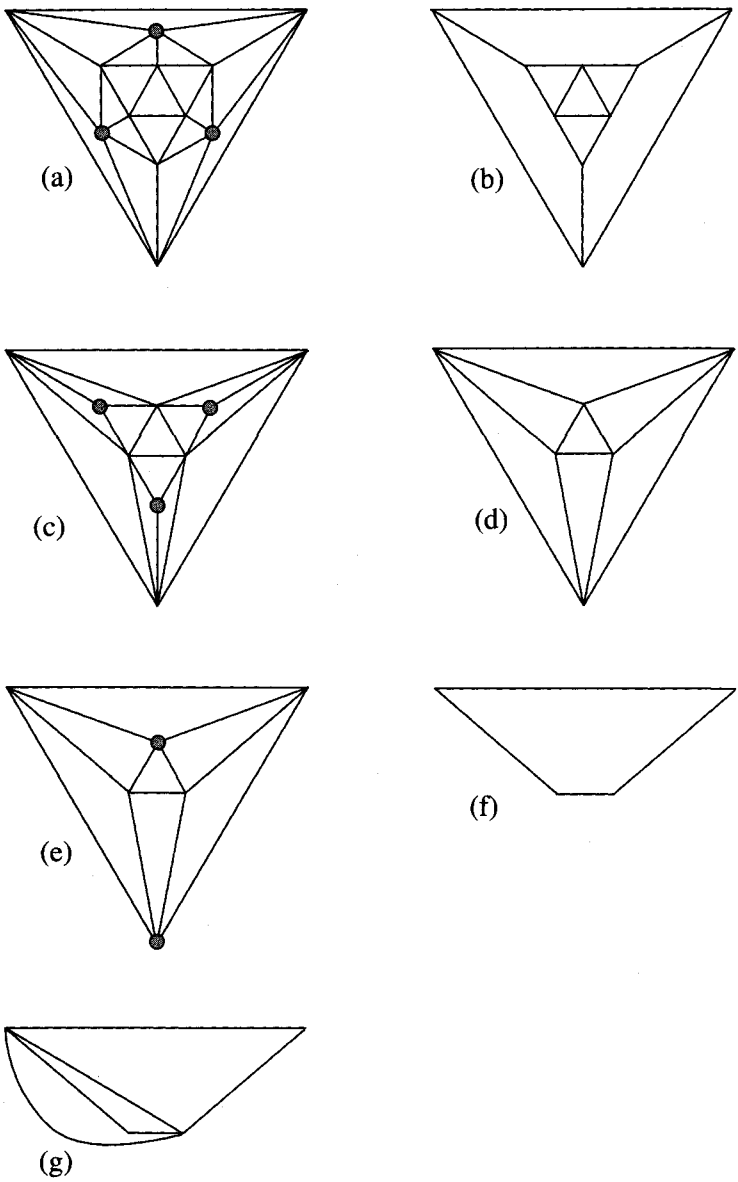

FIGURE 7.21 The graph of the vertices and edges of an icosahedron. Marked nodes form independent sets. (a) Original graph of P_0; (b) after deletion of independent set; (c) after retriangulation: the graph of P_1; (d) after deletion; (e) after retriangulation (same as (d)): the graph of P_2; (f) after deletion; (g) after retriangulation: the graph of P_3.

in Figure 7.21(d). It so happens that, this time, the deletion produces only triangle faces, so no further triangulation is needed. The reader may recognize Figure 7.21(d) as the Schlegel diagram of an octahedron, and indeed the corresponding polytope P_2 is a (nonregular) octahedron, as shown in Figure 7.24.

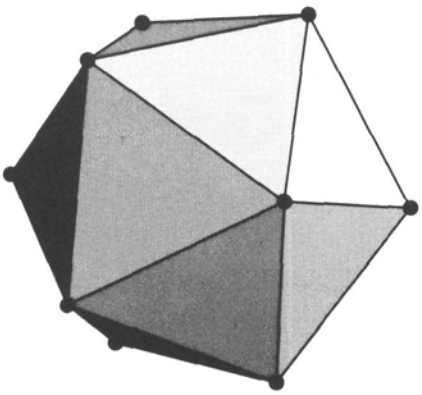

FIGURE 7.22 Icosahedron, P_0 (Figure 7.21(a)).

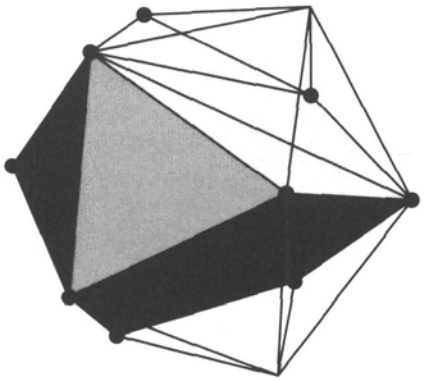

FIGURE 7.23 P_1: 9 vertices, 14 faces (Figure 7.21(c)).

The process is repeated one more time. An independent set of size two is identified in Figure 7.21(e). Deletion produces the graph in Figure 7.21(f). Triangulation of the two quadrilateral faces (one of which is exterior) produces Figure 7.21(g), which is the graph of a tetrahedron. Figure 7.25 displays this tetrahedron, which, again because of the symmetry of the icosahedron, consists of four coplanar points.

7.10.3. Independent Sets: Properties and Algorithm

To achieve a nested polytope hierarchy with the right properties, the independent sets cannot be chosen arbitrarily. Fortunately it is easy to obtain the appropriate properties, as Kirkpatrick showed for arbitrary planar graphs (Kirkpatrick 1983). The arguments are slightly easier for polytope graphs; here I follow the presentation of Edelsbrunner (1987).

In order to achieve only $O(\log n)$ polytopes, it suffices to delete a constant fraction of the vertices at each step. For suppose we can find an independent set of cn vertices on any polytope of n vertices, for $c < 1$. Then at each step, we reduce the vertices by a factor of $(1 - c)$, so after k steps, we will have $n(1 - c)^k$ vertices. This quantity reaches

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

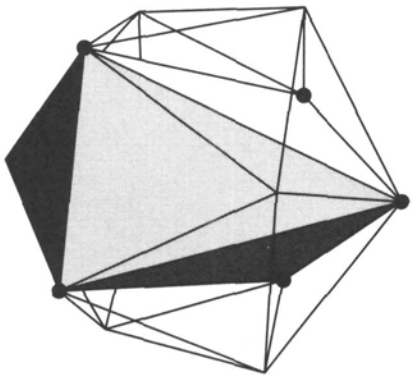

FIGURE 7.24 P_2: an octahedron (Figure 7.21(e)).

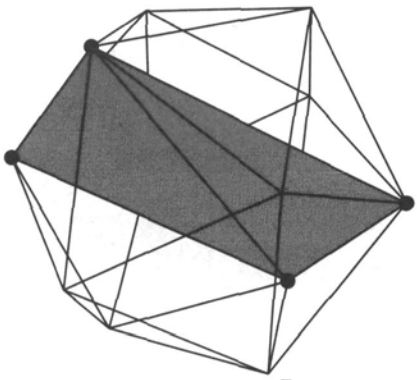

FIGURE 7.25 P_3: a flat tetrahedron (Figure 7.21(g)).

4 when k is a particular value:

$$n(1-c)^k = 4,$$

$$\log n + k\log(1-c) = 2,$$

$$k = \frac{\log n}{-\log(1-c)} - \frac{2}{-\log(1-c)}. \tag{7.10}$$

Since $(1-c) < 1$, $-\log(1-c) > 0$, and the right-hand side of Equation 7.10 is a positive constant times $\log n$, minus another constant; so it is $O(\log n)$. For example, for $n = 2^{20} \approx 10^6$ and $c = 1/10$, $k = 118$.

Thus our goal is to show that every polytope graph has an independent set of size cn for some $c < 1$.

The most natural method of finding an independent set is iterate the following "greedy" procedure: Choose a vertex of lowest degree that is not adjacent to any other vertices previously chosen. The intuition is that low degree vertices "kill" as few other vertex candidates as possible. Although this simple-minded algorithm will not necessarily find a maximum independent set, it turns out to be sufficient for our purposes. We can even loosen it up a bit to choose any vertex whose degree is not too high: This avoids a search for a vertex of lowest degree. In particular, we use Algorithm 7.4. It is clear

that this algorithm produces an independent set and runs in $O(n)$ time on a planar graph of n nodes. What is not so clear is that it produces a "large" independent set. This is established in the following theorem of Edelsbrunner (1987, Theorem 9.8).

Algorithm: INDEPENDENT SET
Input: a graph G.
Output: an independent set I.
$I \leftarrow \emptyset$
Mark all nodes of G of degree ≥ 9.
while some nodes remain unmarked do
    Choose an unmarked node v.
    Mark v and all the neighbors of v.
    $I \leftarrow I \cup \{v\}$.

Algorithm 7.4 Independent set.

Theorem 7.10.1. *An independent set I of a polytope graph G of n vertices produced by Algorithm 7.4 has size at least $n/18$.*

In terms of our previous notation, the theorem claims the constant $c = 1/18$ is achieved by Algorithm 7.4.

Proof. The key to the proof is Euler's formula, $V - E + F = 2$. We established in Chapter 4 (Section 4.1.5, Equation (4.4)) that this formula implies that the number of edges of a polytope graph is bounded above by $3V - 6$: $E \leq 3n - 6$. We now use this to obtain an upper bound on the sum Σ of the degrees of all the nodes of G. This sum double counts every edge of G, since each edge has two endpoints. Thus $\Sigma \leq 6n - 12$.

This bound on the sum of degrees immediately implies that there must be numerous nodes with small degrees. For if all nodes had high degree, the sum of their degrees would exceed this bound. Quantitatively, there must be at least $n/2$ vertices of degree ≤ 8. For suppose the contrary: There are more than $n/2$ nodes of degree ≥ 9. The sum of the degrees of just these nodes is $\geq 9n/2$. The other nodes must each have degree ≥ 3. Let us assume that n is even, to simplify the calculations. The smallest value of Σ would occur when only half the nodes have high degree and the other half have the lowest degree possible. Therefore

$$\Sigma \geq 9n/2 + 3n/2 = 6n. \tag{7.11}$$

This contradicts the upper bound of $6n - 12$ we established above. For n odd, a similar contradiction is obtained (Exercise 7.10.6[2]). Therefore we have established that at least half the nodes of G have degree ≤ 8 and so are candidates for the independent set constructed by Algorithm 7.4. It remains to show that the algorithm selects a "large" number of these candidates.

Every time the algorithm chooses a node v, it marks v and all of v's neighbors. The worst that could happen is that (a) all of these nodes it marks were previously unmarked and (b) v has the highest degree possible, 8. Let m be the number of unmarked nodes

of G of degree ≤ 8. An example may make the relationships clearer. Suppose $m = 90$. A node v is chosen, and in the worst case, 8 unmarked nodes are marked. This reduces m by 9, to 81. Again a node is chosen among these 81, and again in the worst case, m is reduced by 9. It should be clear that at least 1/9-th of m nodes will be added to the independent set I; so with $m = 90$, $|I| \geq 10$.

Now since we showed above that $m \geq n/2$, it follows that $|I| \geq n/18$. And thus we have established that Algorithm 7.4 always produces an independent set at least 1/18-th the size of the original graph. □

With $c = 1/18$, the number of nested polytopes is (by Equation 7.10) less than $12.13 \log n$. This constant of proportionality leaves much to be desired, but always choosing the unmarked node of smallest degree improves c to 1/7 (Edelsbrunner 1987, Problem 9.9(d)) and the log constant to 4.50.

7.10.4. Construction of Nested Polytope Hierarchy

We now detail the construction of the hierarchy. In the pseudocode shown in Algorithm 7.5, $N(v)$ is the set of neighbors of v: all the vertices adjacent to v.

Algorithm: NESTED POLYTOPE HIERARCHY
Input: a polytope P.
Output: an $O(\log n)$ hierarchy of nested polytopes, $P = P_0, P_1, \ldots, P_k$
$i \leftarrow 0$; $P_0 \leftarrow P$.
while $|P_i| > 4$ do
    Apply Algorithm 7.4 to identify an independent set I of P_i.
    Initialize P_{i+1} to P_i.
    for each vertex $v \in I$ do
        Delete v from P_{i+1}.
        Retriangulate the hole by constructing the hull of $N(v)$.
        Link each new face of P_{i+1} to v.
    Link unchanged faces of P_{i+1} to P_i.

Algorithm 7.5 Nested polytope hierarchy.

Space Requirements

We have already established that the polytope hierarchy has height $O(\log n)$. At first it might seem that the time and space required to construct the hierarchy would be $O(n \log n)$, linear per level, but in fact the total is linear because of the constant fractional reduction between levels of the hierarchy. In particular, with $c = 1/18$, each polytope has at most 17/18-ths as many vertices as its "parent." So the total size is no more than

$$n\,[(17/18) + (17/18)^2 + (17/18)^3 + \cdots].$$

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

Although the sum of powers of $(1 - c)$ has only k terms, it is easier to obtain an upper bound by letting it run to infinity. Then it is the familiar geometric series, with sum

$$\frac{1}{1-(1-c)} = \frac{1}{c} = 18.$$

Therefore the total storage required is at most $18n = O(n)$. And similarly the construction time is $O(n)$, although this needs some argument, not provided here.

Retriangulating Holes

We mentioned earlier that when a vertex v is deleted from P_i, the resulting hole must be triangulated appropriately to produce P_{i+1}. Let $N(v)$ be the neighbors of v. In general they will not be coplanar, and so an arbitrary triangulation will not suffice. We need to compute the convex hull of $N(v)$ and use the "outer faces" of this hull to provide the triangulation. In practice we might recompute the entire hull at each step to construct P_{i+1} from P_i, but this would lead to $O(n \log n)$ time complexity. But observe that $|N(v)| \leq 8$, because v had degree ≤ 8. This means that each hole can be patched with triangles in constant time. And the total number of hole patches necessary for the entire hierarchy construction is no more than a constant times the number of vertices deleted, which is $O(n)$.

Linking Polytopes

It is necessary to connect the polytopes in adjacent positions of the hierarchy with data structure links to aid the search. Because the vertices removed at each step of the hierarchy construction form an independent set, the relationships among vertices, edges, and faces of two adjacent polytopes are unambiguous. We will not go into details here (see Edelsbrunner (1987, pp. 199–200)) but rather will just assume that any reasonable link we need is available.

7.10.5. Extreme Point Algorithm

Now we apply the hierarchy to answer extreme point queries. We will explain the algorithm as if we are seeking the highest point of the polytope, a vertex with largest z coordinate, but the process works for an extreme in any direction u in an analogous fashion. The algorithm was first detailed by Edelsbrunner & Maurer (1985); see also Edelsbrunner (1987, Section 9.5.3).

Let a_i be a highest point of polytope P_i. To keep the presentation simple, we will assume that a_i is unique for each i. The essence of the algorithm is to find the highest point a_k of P_k, the innermost polytope, by brute-force search, and then use a_k to help find a_{k-1}, use this to find a_{k-2}, and so on until a_0 is found, which is the highest point of $P_0 = P$, the original polytope. This process can be viewed as raising a plane π orthogonal to the z axis from a_k, to a_{k-1}, and so on to a_0. Because the polytopes are nested, this plane only moves upwards. An example is shown in Figure 7.26. Here the innermost polytope, a triangle, is not shown.

The key to the algorithm is the relationship between a_{i+1} and a_i. We condense this relationship into two lemmas, Lemmas 7.10.2 and 7.10.3 below. The first is perhaps easiest to see if we imagine π moving downwards, from a_i to a_{i+1}.

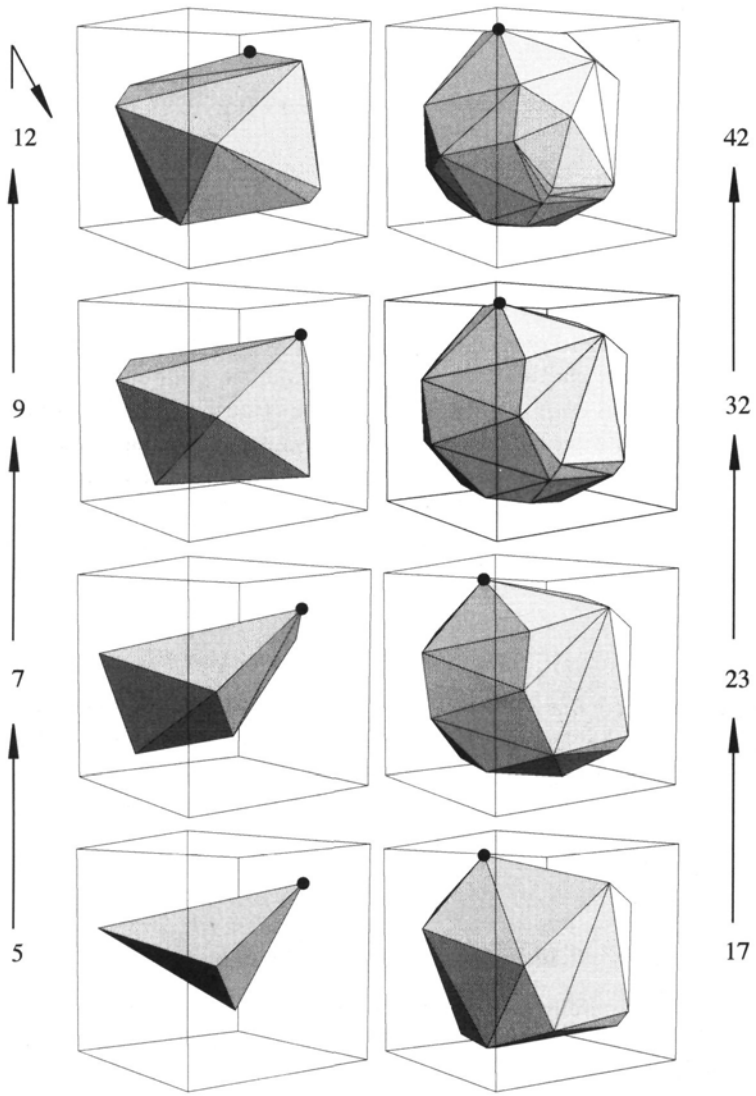

FIGURE 7.26 Polytope hierarchy with highest vertices marked, and number of vertices noted to the side. Box dimensions are ± 100. The highest vertex has z coordinate 63, 63, 63, 77, 92, 92, 94, 94, moving up the hierarchy.

Lemma 7.10.2. *Let a_i and a_{i+1} be uniquely highest vertices of P_i and P_{i+1}. Then either $a_i = a_{i+1}$ or a_{i+1} is the highest among the vertices adjacent to a_i.*

Proof. We consider two cases. First, suppose that a_i is a vertex of both P_i and P_{i+1}. Because $P_i \supset P_{i+1}$, no vertex of P_{i+1} can be higher than the highest of P_i, and therefore the highest vertex a_{i+1} of P_{i+1} must in this case be a_i.

Second, suppose a_i is one of the vertices deleted in the construction of P_{i+1}. Let b_{i+1} be the highest vertex of P_{i+1} among those adjacent to a_i in P_i. The claim of the lemma is that b_{i+1} is the highest vertex of P_{i+1}.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

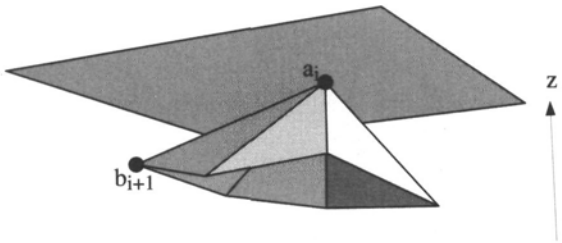

FIGURE 7.27 Highest point a_i of P_i.

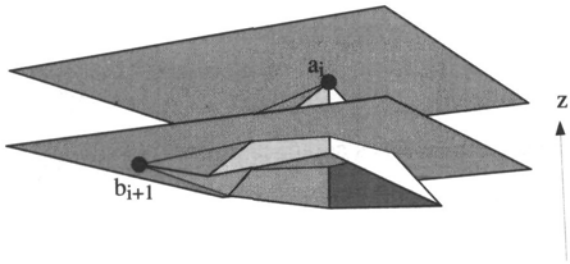

FIGURE 7.28 Highest point $b_{i+1} = a_{i+1}$ of P_{i+1}.

Consider the cone of faces incident to a_i in P_i; see Figure 7.27. Call the infinite extension of this cone C. By the convexity of P_i, $C \supset P_i$, and by nesting, $C \supset P_i \supset P_{i+1}$. Thus $a_{i+1} \in C$. But now no vertex of P_{i+1} can be located in the umbrella-shaped region under a_i, $U = C - P_{i+1}$. So a_{i+1} must lie in the other part of C, $C - U$, which is necessarily below the height of b_{i+1}, as Figure 7.28 makes clear. Therefore $b_{i+1} = a_{i+1}$. □

An immediate corollary of this lemma is that a_i is either identical to a_{i+1}, or is adjacent to it. It might seem this gives us the "hook" we need to move from a_{i+1} to a_i, but in fact this is not enough, because we have no bound on the number of vertices adjacent to a_{i+1}; so if we search them all for a_i the algorithm will work correctly, but it will have time complexity $O(n)$ rather than the $O(\log n)$ we desire. We need a more specific hook from a_{i+1} to a_i.

Extreme Edges

If one projects P_{i+1} onto a plane orthogonal to π, say the xz-plane, then π becomes a line and P_{i+1} becomes a convex polygon P'_{i+1}, as shown in Figure 7.29. Let primes denote objects projected to the xz-plane. Define L_{i+1} and R_{i+1} as the two edges of P_{i+1} that project to the two edges of P'_{i+1} incident to a'_{i+1}, as illustrated.

Now define the "umbrella parents," or just *parents*, of an edge e of P_{i+1} to be the vertices of P_i from which it derives, in the following sense: If e is an edge of P_{i+1} but not of P_i, then it sits "under" some vertex v of P_i whose umbrella of incident faces was deleted to produce P_{i+1}; this v is the (sole) parent of e. (This is most evident in Figure 7.23, where the two diagonals of the upper pentagonal face sit under a vertex of degree five.) If e is an edge of both P_{i+1} and P_i, then its parents are the two vertices of

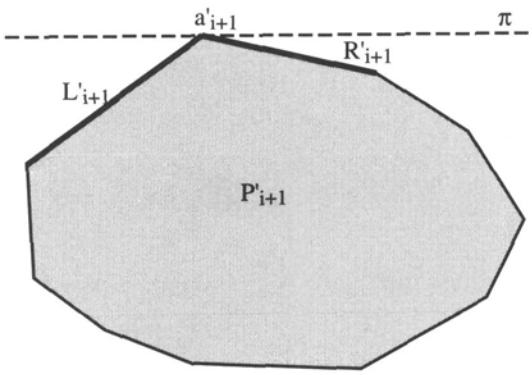

FIGURE 7.29 Definition of extreme edges L_{i+1} and R_{i+1}; The z axis is vertical.

P_i at the tips of the two triangle faces adjacent to e (which may or may not be vertices of P_{i+1}).

The key lemma is:

Lemma 7.10.3. *Let a_i and a_{i+1} be uniquely highest vertices of P_i and P_{i+1}. Then either $a_i = a_{i+1}$ or a_i is the highest among the parents of the extreme edges L_{i+1} and R_{i+1}.*

Before discussing why this might be true, let us explore its consequences. If we have both the extreme vertex a_{i+1} of P_{i+1}, and the extreme edges L_{i+1} and R_{i+1}, we can find the extreme vertex a_i of P_i by examining the (at most) five candidates provided by the lemma. If we can then find the new extreme edges L_i and R_i of P_i in constant time, we have achieved one full step up the hierarchy in constant time, which will result in $O(\log n)$ overall.

How can the extreme edges be computed once a_i is known? There are two cases to consider:

1. $a_i \neq a_{i+1}$. Here, surprisingly, we can use a brute-force search for the extreme edges. The reason is that such a search will have time complexity dependent on the degree of a_i, which, when we first encounter it in the hierarchy of polytopes, is an independent vertex chosen for deletion in the hierarchy construction and therefore has degree ≤ 8.
2. $a_i = a_{i+1}$. Here a brute-force search is not appropriate, because if we move through a series of levels of the hierarchy without the extreme vertex changing, its degree can grow larger and larger by "accretion" of edges: We are only guaranteed a degree ≤ 8 upon first encounter. This accretion is evident in the three innermost polytopes of the hierarchy in Figure 7.26. Fortunately, the new extreme edges are "close" to the old: L_i is either L_{i+1} or is adjacent to a parent of L_{i+1}; and similarly for R_i. I will not justify this claim (Exercise 7.10.6[3]).

So in both cases the new extreme edges can be found in constant time. Now we justify Lemma 7.10.3.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

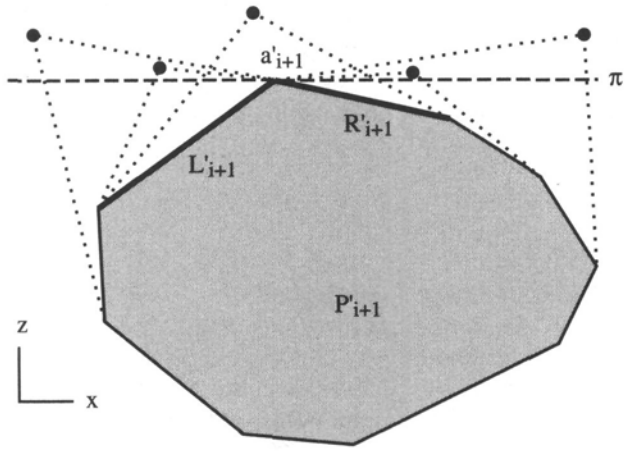

FIGURE 7.30 Potential locations for a_i' and tangents shown dotted.

Proof. Suppose $a_i \neq a_{i+1}$. Then a_i is above π. Of course the projection of P_i onto the xz-plane is a convex polygon P_i' that encloses the projection P_{i+1}'. (Recall that primes indicate projected objects.) If possible locations for a_i' are considered, as in Figure 7.30, it becomes clear that a_i must "sit over" one or both of the extreme edges L_{i+1} and R_{i+1}. This is the intuition behind the lemma.

Why are the dotted connections from a_i' in the figure reasonable possibilities? First, recall that all the vertices adjacent to a_i in P_i are also vertices of P_{i+1}. So the edges in the projection emanate from a_i' and terminate in P_{i+1}' below π. Second, $P_i \supset \text{conv}\{a_i \cup P_{i+1}\}$, so the two tangents through a_i' supporting P_{i+1}' are in P_i'. Third, there can be no edges from a_i' "outside of" these tangents, because such edges could not terminate in P_{i+1}'. Thus the boundary of P_i' includes the a_i' tangents, and Figure 7.30 is an accurate depiction.

Since it is clear that the a_i' tangents must encompass at least one of the extreme edge of P_{i+1}' and that a_i is a parent of this edge, we have established the lemma. □

We can summarize the algorithm in the pseudocode shown in Algorithm 7.6. From Lemmas 7.10.2 and 7.10.3, and from Exercise 7.10.6[3], this algorithm will work correctly, so the only issue remaining is its time complexity. But we have ensured that the work done at each level of the algorithm is constant. This then establishes the query time of the algorithm: $O(\log n)$ levels of the hierarchy are searched, and the work at each level is a constant. Modulo the details we have ignored, we have established the following theorem:

Theorem 7.10.4. *After $O(n)$ time and space preprocessing, polytope extreme-point queries can be answered in $O(\log n)$ time each.*

One important application of this theorem arises in collision detection: detecting whether two convex polyhedra of n and m vertices intersect. Representing each polytope

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

Algorithm: EXTREME POINT OF A POLYTOPE
Input: a polytope P, and a direction vector u.
Output: the vertex a of P extreme in the u direction.
Construct the hierarchy of nested polytopes, $P = P_0, P_1, \ldots, P_k$,
    by running Algorithm 7.5.
$a_k \leftarrow$ the vertex of P_k extreme in the u direction.
Compute L_k and R_k.
for $i = k-1, k-2, \ldots, 1, 0$ do
    $a_i \leftarrow$ the extreme vertex among a_{i+1}
        and the parents of L_{i+1} and R_{i+1}.
    if $a_i \neq a_{i+1}$ then
        for all edges incident to a_i do
            Save extreme edges L_i and R_i.
    else ($a_i = a_{i+1}$) Compute L_i from L_{i+1} etc.

Algorithm 7.6 Extreme point of a polytope.

in a hierarchy, and using Exercise 7.10.6[8], it is possible to compute efficiently the separation between the polytopes at a common level of the hierarchy from the separation between the polytopes at the level below. This leads to an $O(\log n \log m)$ intersection detection algorithm (Dobkin & Kirkpatrick 1983).

7.10.6. Exercises

1. *Innermost polytope* [easy]. Why cannot the innermost polytope of the hierarchy have ≥ 5 vertices?
2. *n odd.* In the proof of Theorem 7.10.1, we only covered the n even case. Follow the argument for n odd, and show the conclusion still holds.
3. $a_i = a_{i+1}$. Argue that if $a_i = a_{i+1}$ in Algorithm 7.6, L_i is either L_{i+1} or it is adjacent to a parent of L_{i+1}. Show how these facts permit L_{i+1} to be found in constant time in this case.
4. *Nested polygon hierarchy.* Develop a method of constructing a hierarchy of $O(\log n)$ convex polygons nested inside a given convex polygon of n vertices. Use this to design an extreme-point algorithm that achieves $O(\log n)$ query time.
5. *The constant c* [easy]. Compute the average constant c for the example in Figure 7.26, and using this, calculate k from Equation 7.10.
6. *Implementation of independent set algorithm* [programming]. Write a program to find an independent set in a given graph using Algorithm 7.4.
7. *Nested polytope implementation* [programming]. Use the convex hull code from Chapter 4 and the independent set code from the previous exercise to find a polytope nested inside the hull of n points. Test it on randomly generated hulls, and compute the average fractional size of the independent sets. Compare this against the $c = 1/18$ constant established in Theorem 7.10.1, and try to explain any difference.
8. *Plane–polyhedron distance.* Design an algorithm to determine the distance between an arbitrary polyhedron P of n vertices and a query plane π. Define the distance to be

$$\min_{x,y}\{|x-y| : x \in P,\ y \in \pi\},$$

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

where x and y are points. Try to achieve $O(\log n)$ per query after preprocessing. Compare with Exercise 7.9.2[3].

9. *Finger probing a polytope* (Skiena 1992). Develop an algorithm for "probing" a polytope P that contains the origin, with a directed line L through the origin. Each probe is to return the first face of P hit by L moving in from infinity. Try to achieve $O(\log n)$ query time, by dualizing P and L with the polar dual discussed in Chapter 6 (Exercise 6.5.3[3]).
10. *Circumscribed hierarchies.*
    a. Define an $O(\log n)$ hierarchy of polygons surrounding a convex polygon, with properties similar to the inscribed hierarchy.
    b. Define an $O(\log n)$ hierarchy of polytopes surrounding a given polytope.
    c. Suggest applications for these circumscribed hierarchies.

7.11. PLANAR POINT LOCATION

7.11.1. Applications

One of the most fundamental of geometric searches involves locating a point in a subdivision of the plane, known as the *planar point location* problem. We have already encountered needs for this search for constructing a trapezoidalization (Section 2.4.1), for searching Voronoi diagrams to find nearest neighbors (Section 5.5.1), and for searching the kth-order Voronoi diagram to find the k-nearest neighbors (Section 6.6).

Another common application of planar point location is determining if a query point is inside a given polytope. Although we already mentioned a method for solving the point-in-polytope problem in Section 7.5, that method is "single-shot" and not efficient for the situation where the polytope is fixed and must be repeatedly queried for different points. The connection between this problem, and planar point location can be seen via the same type of three-to-two dimensions projection used in Section 7.3.1.[22] Suppose the given polytope P sits on the xy-plane, and let P^+ be the set of all faces of P whose outward normal has a nonnegative upward component (i.e., whose z component is ≥ 0). These are the faces visible from $z = +\infty$. Let P^- be the set of all the other faces, whose normals point down. Project P^- onto the $z = 0$ plane, and project P^+ onto the $z = h$ plane, where h is the height of P. This results in two subdivisions on these two planes; call them S^+ and S^-. Now, given any query point q, project it up and down and locate it in both subdivisions. Suppose it projects into face f^+ of S^+. Then this selects out a vertical "prism" as shown in Figure 7.31. It is then easy to decide if q is above or below f in this prism. If it is above f, $q \notin P$. If below, then the process is repeated on the lower subdivision. $q \in P$ iff it is below the face of P provided by the search in S^+ and above the face of P provided by the search in S^-.

7.11.2. Independent Set Algorithm

The reader may have realized already that Kirkpatrick's search structure, presented in Section 7.10, provides a solution to the planar point location problem. Indeed this was his original motivation (Kirkpatrick 1983). The only complication is that a general planar

[22] Suggested in Edelsbrunner (1987, Ex. 11.5).

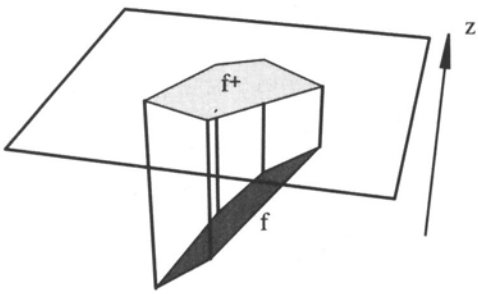

FIGURE 7.31 Face f of the polytope projects up to face f^+ of the upper planar subdivision. Cf. Figure 7.3.

subdivision may have general polygonal faces, which need to be triangulated by a polygon triangulation algorithm. After that step, we can proceed as with the polytope hierarchy, except that each hole produced by a vertex deletion should be retriangulated by a polygon triangulation algorithm. As with the polytope case, however, this retriangulation only takes constant time per hole, since the holes have at most eight vertices.

Theorem 7.11.1. *A polygonal planar subdivision of n vertices can be preprocessed in $O(n)$ time and space so that point location queries can be answered in $O(\log n)$ time.*

7.11.3. Monotone Subdivisions

Although Kirkpatrick's search structure in some sense settles the planar point location problem, it was neither the first algorithm to achieve those bounds nor the most recent. An early algorithm by Dobkin & Lipton (1976) uses quadratic space, but is very simple, and achieves a much better query constant: Queries can be performed with $2 \log n$ comparisons. Lipton & Tarjan (1980) were the first to achieve $O(n)$ preprocessing with $O(\log n)$ query time, but their algorithm is impractically complex. Kirkpatrick's algorithm is elegant and ideal for polytope problems, but its high query constant make it unattractive for general planar subdivision search.

One popular method of performing planar point location depends on *monotone subdivisions*. A subdivision is monotone if every face is monotone, say with respect to the horizontal. A face is *monotone* if it meets every vertical line in a connected set: either a point or a segment (see Section 2.1). Many commonly encountered subdivisions are monotone: triangulations and any convex subdivision such as a Voronoi diagram or a kth-order Voronoi diagram or an arrangement of lines. Those subdivisions that are not monotone can be further partitioned (by, e.g., triangulating each face) to produce a monotone subdivision. The utility of these subdivisions was recognized by Lee & Preparata (1977), and they have been studied intensively ever since.

We will now sketch roughly some of the main ideas behind monotone subdivision search. Define a *separator* in a monotone subdivision as a connected collection of edges of the subdivision that meet every vertical line exactly once. These are monotone chains that separate the subdivision into two parts, above and below.

The main idea is to find a collection of separators that partition the subdivision into "horizontal" strips. Then a double binary search is performed: a vertical search on these

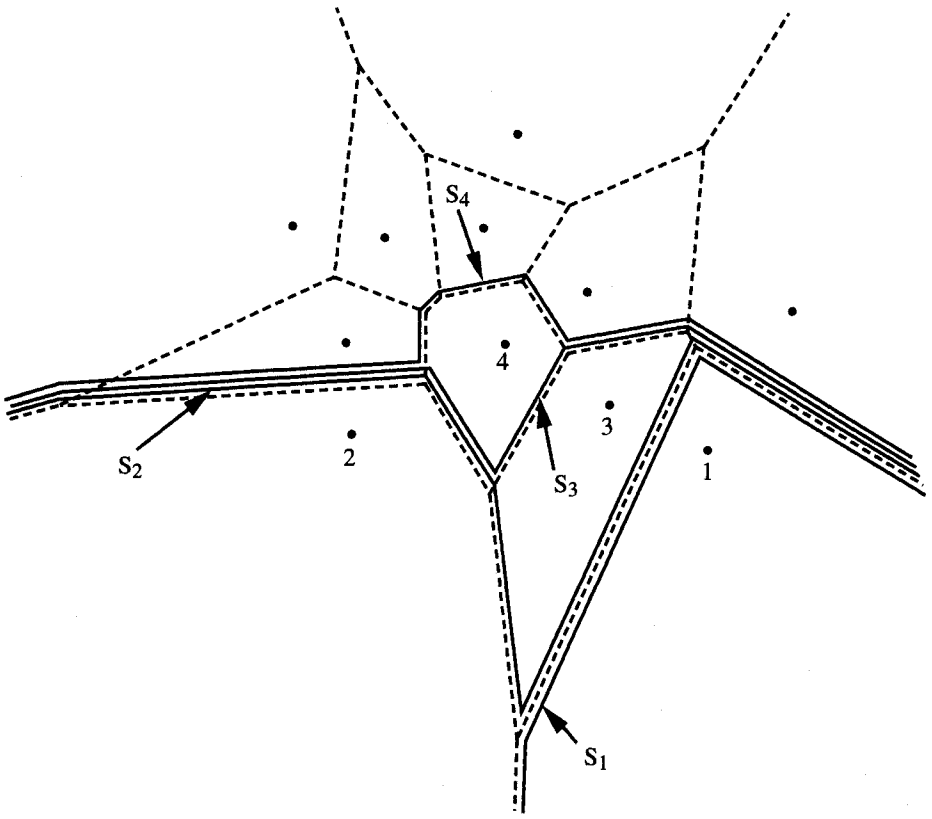

FIGURE 7.32 Separators in a monotone subdivision: $S_1 < S_2 < S_3 < S_4$.

strips to locate the query point between two separators and a horizontal search to locate it within one strip.

An example is shown in Figure 7.32. The subdivision is a Voronoi diagram, which is of course monotone. Four separators are shown. S_1 is the lowest, having only the Voronoi cell C_1 for point 1 below it. S_2 is the next highest, having C_2 and C_1 below it. Note that S_2 is above S_1 throughout their lengths. Similarly S_3 is above C_3, and S_4 is above C_4. This process could be continued, finding a collection of separators $S_1, S_2, \ldots, S_m$ that can be considered sorted vertically, with each pair of adjacent separators having one cell of the subdivision sandwiched between them.

Consider the problem of deciding whether a query point q is above or below some particular separator S_i. This can be accomplished via a horizontal binary search on the x coordinates of the vertices of S_i and the x coordinate of q, because S_i is monotone with respect to the x axis. Once the projection of q on the x axis is located between two endpoints of an edge e of S_i, it can be tested for above or below e. Since any S_i has $O(n)$ edges, the query "Is q above or below S_i?" can be answered in $O(\log n)$ time.

Now this query can be used repeatedly to perform a binary search on the collection of separators. First ask if q is above or below $S_{m/2}$. If it is below, query its relation to $S_{m/4}$; if above, query $S_{3m/4}$; and so on. This binary search will take $O(\log m)$ steps, each of which costs $O(\log n)$. Since $m = O(n)$, the total query time is $O(\log^2 n)$.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

Of course this is asymptotically worse than what is achieved in Theorem 7.11.1. Moreover, it could require quadratic space to store the separators, due to the high degree of shared edges, as is evident in Figure 7.32. However, the algorithm is attractively simple, and it can be improved in both query time and space requirements to achieve the same asymptotic complexities as claimed in Theorem 7.11.1. These improvements are by no means straightforward and awaited the inventions of topological sorting and fractional cascading (Chazelle & Guibas 1986*a*; Chazelle & Guibas 1986*b*) among other ideas. See Edelsbrunner (1987, Chapter 11) for a thorough presentation.

7.11.4. Randomized Trapezoidal Decomposition

Randomized algorithms present a relatively recent, attractive alternative for point location. Here we present one such algorithm due to Seidel (1991), foreshadowed in Chapter 2. In Section 2.4.1, we used trapezoidalization to triangulate a polygon. The same general technique applies to more general objects than polygons. In particular, it works for collections of *noncrossing* segments: no two segments share a point interior to either, but they may share endpoints. Note that the edges of a polygon satisfy this definition. Let $S = \{s_1, \ldots, s_n\}$ be a collection of noncrossing segments. The goal is to extend horizontal chords left and right from each segment endpoint, partitioning the plane into "trapezoids." These are faces of two horizontal sides each, one of which may be degenerate, of zero length. Faces may be unbounded (although sometimes it is convenient to surround S with a large axis-aligned rectangle to ensure that all trapezoids are bounded). See Figure 7.33. There are $O(n)$ trapezoids; Exercise 7.11.5[1] asks for a proof that $3n + 1$ is a tight upper bound.

To simplify the details, we will assume that no two endpoints lie on a horizontal line (see Section 2.2). This limits the neighboring relations:

Lemma 7.11.2. *Each trapezoid has at most two trapezoids neighboring above, and two below, where neighboring trapezoids share a nonzero-length portion of a horizontal chord.*

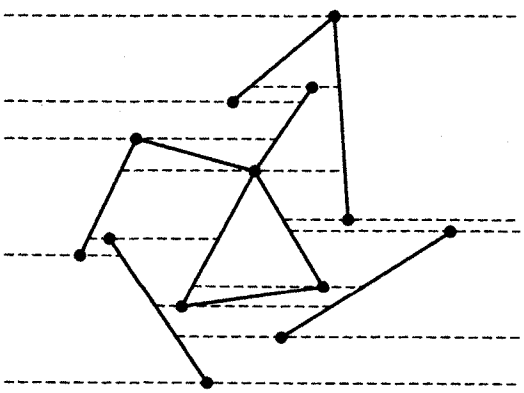

FIGURE 7.33 The trapezoid decomposition induced by $n = 10$ segments.

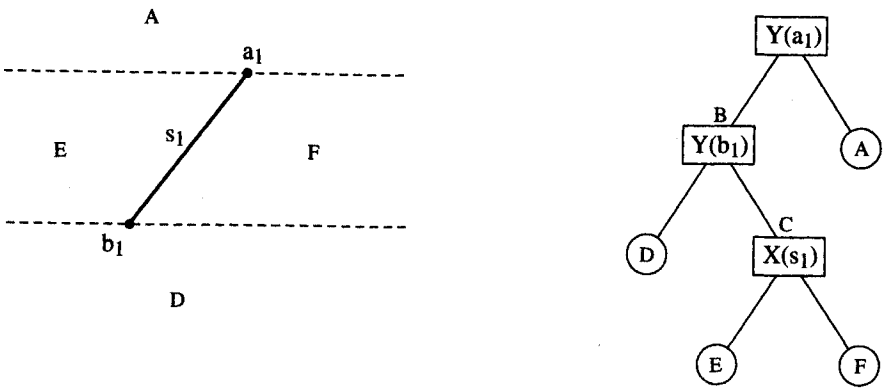

FIGURE 7.34 Search tree after inserting $s_1 = a_1b_1$.

Proof. Suppose a trapezoid had, for example, three neighbors above. The three sections of horizontal chord that form its top side cannot derive from a single endpoint, because each endpoint generates at most two: a left and right chord. Thus this upper side must contain at least two vertices, violating the assumption that no two endpoints lie on a horizontal line. □

This lemma allows us to represent the trapezoid locations with a binary search tree, binary because there are at most two neighbors. This clever search structure was developed in the late 1970s and was explicitly used by Preparata (1981). The version detailed here follows Seidel (1991), with the trapezoid decomposition algorithm growing out of work of Mulmuley (1990).

The search tree has three types of nodes:

1. internal X nodes, which branch left or right of a segment s_i;
2. internal Y nodes, which branch above or below a segment endpoint; and
3. leaf trapezoid nodes.

The search tree is constructed incrementally. Let $s = ab$ be a new segment to be added to an existing structure. The update of the structure can be partitioned into several steps:

1. Add endpoints a and b. For each endpoint, find the trapezoid that contains it by searching in the tree. Split this trapezoid with a Y node.
2. Add segment s.
   (a) "Thread" s through the partition, identifying each trapezoid cut by s.
   (b) On each side of s, merge trapezoids whose left and right bounding segments are the same.
   (c) Create X nodes for all trapezoids separated by s.

We now run through the construction of the search tree for three particular segments $\{s_1, s_2, s_3\}$. Although the details are somewhat tedious, patience will be rewarded by better appreciation of the beauty of the resulting data structure. Throughout the figures (7.34–7.39), below arcs are drawn left of above arcs, and left arcs drawn left of right arcs.

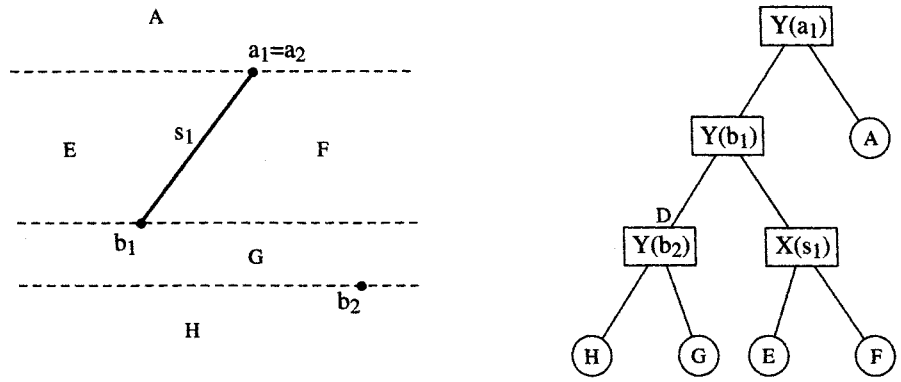

FIGURE 7.35 Search tree after inserting a_2 and b_2.

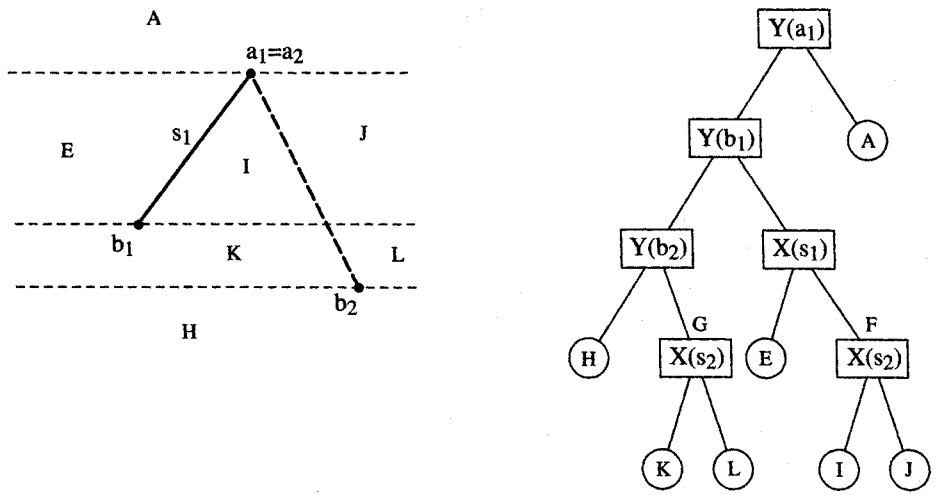

FIGURE 7.36 Search tree after threading s_2.

1. Add a_1. Split the original region (the whole plane) into A and B, above and below respectively. Create a $Y(a_1)$ node as their parent.
2. Add b_1. Locate $b_1 \in B$. Split B into C and D with a $Y(b_1)$ parent node.
3. Thread s_1. Split C into E and F, left and right respectively, with an $X(s_1)$ parent node. The search tree is now as shown in Figure 7.34.
4. Add $a_2 = a_1$. No change to the structure occurs.
5. Add b_2. Locate $b_2 \in D$. Split D into G and H with a $Y(b_2)$ node. See Figure 7.35.
6. Thread s_2. Split F into I and J, and split G into K and L, both with separate $X(s_2)$ parents. See Figure 7.36.
7. Merge along s_2. Merge J and L one region M, because they share the same left and right bounding segments (s_2 and $+\infty$). Rewire tree accordingly. See Figure 7.37.
8. Add a_3. Locate $a_3 \in I$, and split into N and O with a $Y(a_3)$ node.
9. Add b_3. Locate $b_3 \in K$, and split into P and Q with a $Y(b_3)$ node. See Figure 7.38.
10. Thread s_3. Split O into R and S, and split P into T and U, both with $X(s_3)$ parents.
11. Merge along s_3. Merge S and U into one trapezoid V. See Figure 7.39.

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

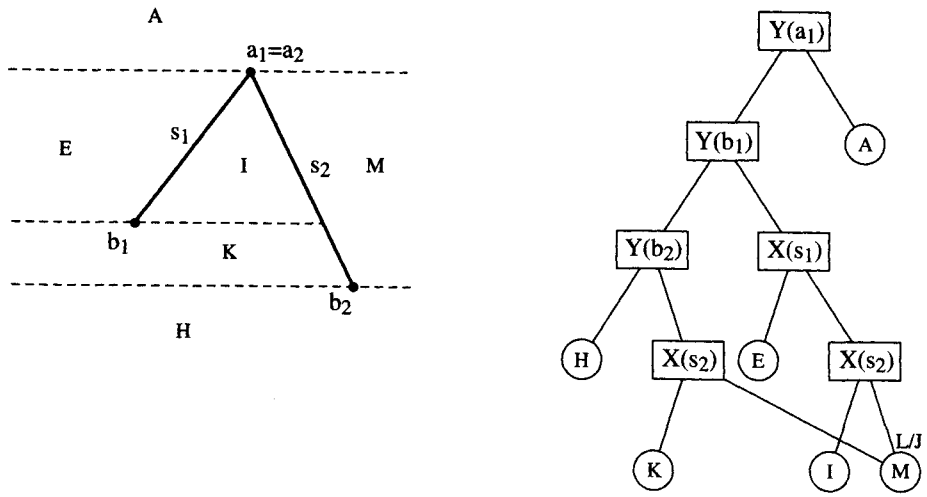

FIGURE 7.37 Search tree after merging regions along s_2.

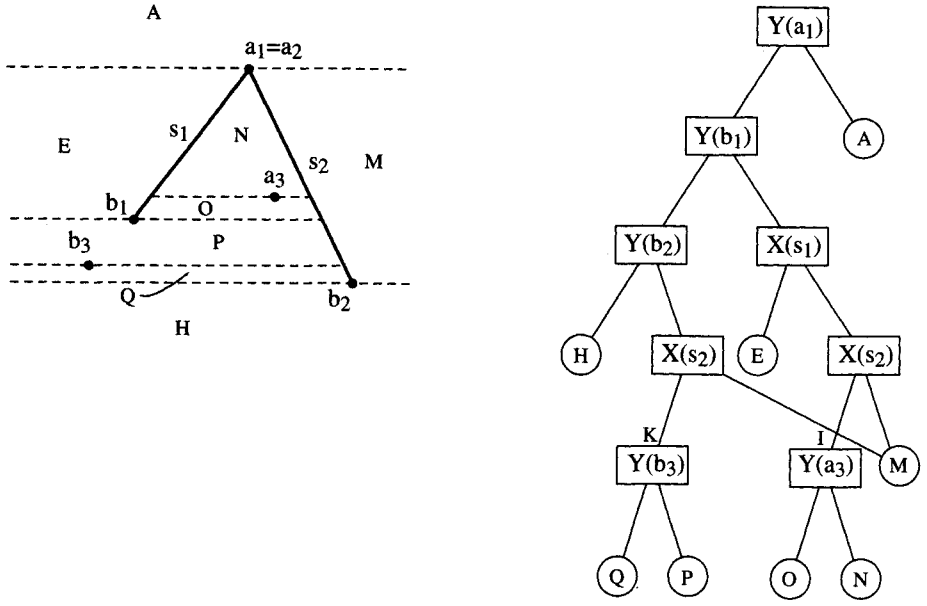

FIGURE 7.38 Search tree after inserting a_3 and b_3.

Let us use the final search tree in Figure 7.39 to locate q in the shaded trapezoid V. Point q is below a_1 and b_1 but above b_2, so from the root the path bends: left, left, right. q is left of s_2, so the left branch is taken at the $X(s_2)$ node. q is above b_3, and finally it is right of s_3. The search path leading to V is highlighted in the figure. Note that not all trapezoids have a unique paths from the root. If q were in V but above b_1, then V would be reached by another route.

It is clear from our description that the search structure obtained is dependent on the order in which the segments are inserted. A "bad" order could result in a thin tree of

https://doi.org/10.1017/CBO9780511804120.008 Published online by Cambridge University Press

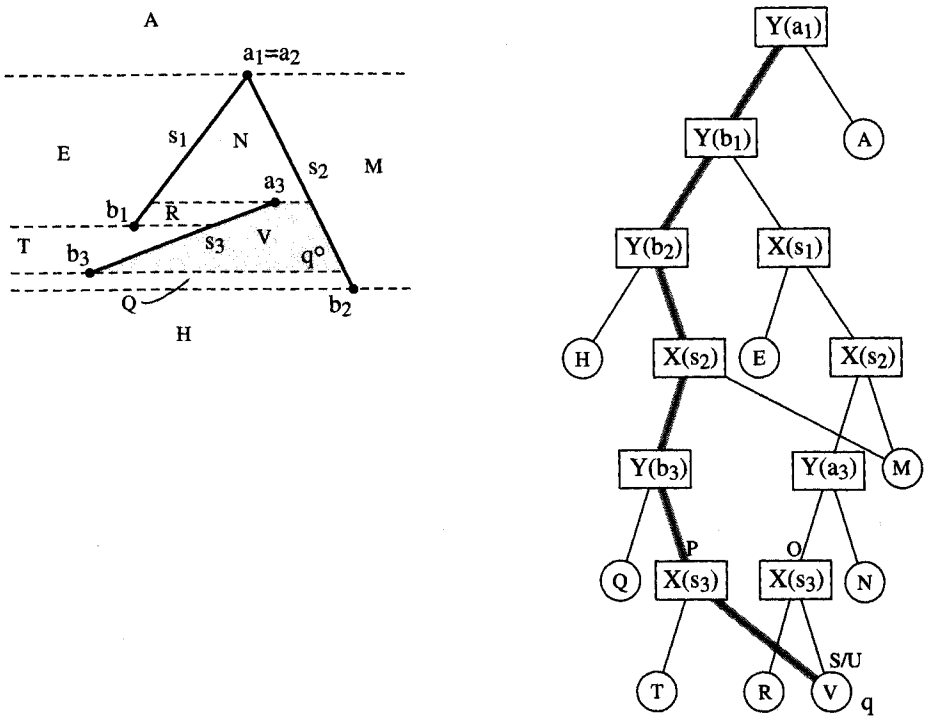

FIGURE 7.39 Search tree after inserting s_3. The search path for $q \in V$ is highlighted.

height $\Omega(n)$; a "good" order will yield a bushy tree of height $O(\log n)$. And the query time is proportional to this height. As mentioned in Section 2.4.1, if the segments are added in random order (i.e., each of the $n!$ orders is equally likely), then it can be proven that the expected height is $O(\log n)$; moreover, the expected time to build the entire structure is $O(n \log n)$. Although it is conceivable that these expectations are weak in the sense that they are broad averages masking nasty performance, in fact it can be further proved that the probability that the tree height significantly exceeds $O(\log n)$ is small.[23] Thus this clean and practical algorithm achieves expected $O(\log n)$ query time with expected $O(n \log n)$ preprocessing.

Planar point location remains an active area of research. Not only does there remain room for improvement on the basic problem discussed here, but two important related problems are very much in flux at this writing: "dynamic" planar point location, where the subdivision changes, for example, by insertions or deletions of Voronoi sites, and point location in subdivisions of three-dimensional and higher spaces.

7.11.5. Exercises

1. *Number of trapezoids.* Prove that the number of trapezoids produced by the trapezoidal decomposition algorithm is at most $3n + 1$ for n segments (de Berg et al. 1997, Lem. 6.2). Include in your count a trapezoid above all segments and one below; or equivalently, surround the segments by a large rectangle and count all trapezoids in the rectangle.

[23] See, e.g., de Berg et al. (1997, Lem. 6.7).

2. *Detection of intersection of convex polygons.* Develop an algorithm for reporting whether or not two convex polygons of n and m vertices intersect. Try to achieve $O(\log(n + m))$ time (Chazelle & Dobkin 1987).
3. *Interval trees.* Preprocess a set of n intervals I (on a line) with integer endpoints so that they can be efficiently queried. Consider three types of queries (the preprocessing need not be the same for all three):
   a. Is x in some interval in I?
   b. Within how many intervals of I does x lie?
   c. Does the interval $[a, b]$ intersect any interval of I?
4. *Length of union of intervals.* Design an algorithm to find the total length covered by the union of n intervals.
5. *Empty circle queries* (Michael Goodrich). Given a set S of n points in the plane, sketch a good method for constructing an efficient data structure to quickly answer empty circle queries. An *empty circle query* for a query point q asks for the largest circle that has q as its center and does not contain any point of S in its interior.
6. *Cops and robbers* (Michael Goodrich). Suppose you are given two sets of n points in the plane, P and R. The points in P represent "police officers" and the points in R represent "robbers." A point q in the plane is *safe* if it is inside the triangle formed by three points in P. A point q in the plane is *robbed* if it is not safe and is inside the triangle formed by three points in R. A point q is *suspect* if it is neither safe nor robbed. Describe an efficient data structure to determine, for any query point q, whether q is safe, robbed, or suspect.