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).

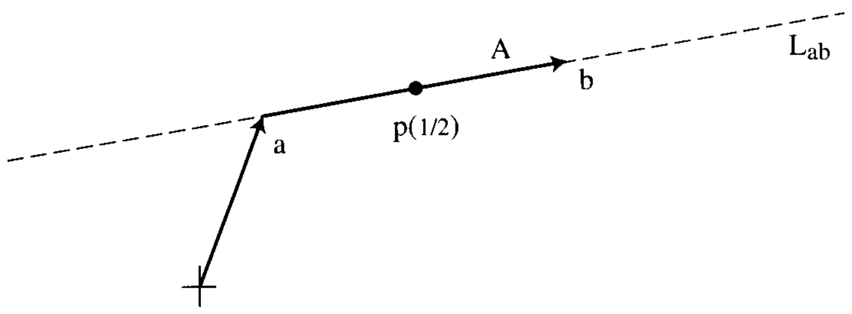

FIGURE 7.1 $p(s) = a + sA$; $p(\frac{1}{2}) = a + \frac{1}{2}A$ is shown.

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.

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.

```
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.

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 $T = \Delta abc$ 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.