14. Straight lines and circles

Learning objectives:

- intersection of two line segments
- degenerate configurations
- clipping
- digitized lines and circles
- Bresenham's algorithms
- braiding straight lines

Points are the simplest geometric objects; straight lines and line segments come next. Together, they make up the lion's share of all primitive objects used in two-dimensional geometric computation (e.g. in computer graphics). Using these two primitives only, we can approximate any curve and draw any picture that can be mapped onto a discrete raster. If we do so, most queries about complex figures get reduced to basic queries about points and line segments, such as: is a given point to the left, to the right, or *on* a given line? Do two given line segments intersect? As simple as these questions appear to be, they must be handled efficiently and carefully. Efficiently because these basic primitives of geometric computations are likely to be executed millions of times in a single program run. Carefully because the ubiquitous phenomenon of *degenerate configurations* easily traps the unwary programmer into overflow or meaningless results.

Intersection

The problem of deciding whether two line segments intersect is unexpectedly tricky, as it requires a consideration of three distinct nondegenerate cases, as well as half a dozen degenerate ones. Starting with degenerate objects, we have cases where one or both of the line segments degenerate into points. The code below assumes that line segments of length zero have been eliminated. We must also consider nondegenerate objects in degenerate configurations, as illustrated in Exhibit 14.1. Line segments A and B intersect (strictly). C and D, and E and F, do not intersect; the intersection point of the infinitely extended lines lies on C in the first case, but lies neither on E nor on F in the second case. The next three cases are degenerate: G and H intersect barely (i.e. in an endpoint); I and J overlap (i.e. they intersect in infinitely many points); K and L do not intersect. Careless evaluation of these last two cases is likely to generate overflow.

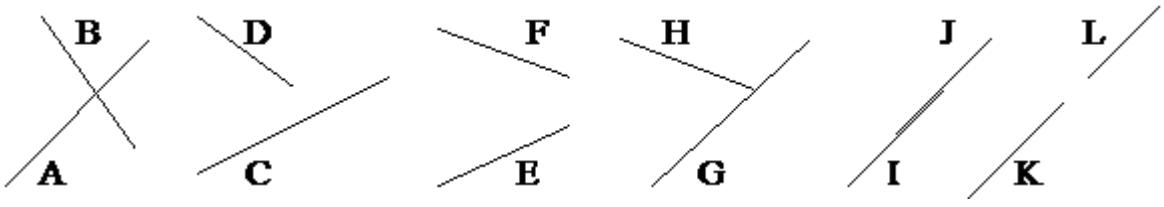

Exhibit 14.1: Cases to be distinguished for the segment intersection problem.

Computing the intersection point of the infinitely extended lines is a naive approach to this decision problem that leads to a three-step process:

1. Check whether the two line segments are parallel (a necessary precaution before attempting to compute the intersection point). If so, we have a degenerate configuration that leads to one of three special cases: not collinear, collinear nonoverlapping, collinear overlapping
2. Compute the intersection point of the extended lines (this step is still subject to numerical problems for lines that are almost parallel).
3. Check whether this intersection point lies on both line segments.

If all we want is a yes/no answer to the intersection question, we can save the effort of computing the intersection point and obtain a simpler and more robust procedure based on the following idea: two line segments intersect strictly iff the two endpoints of each line segment lie on opposite sides of the infinitely extended line of the other segment.

Let L be a line given by the equation $h(x, y) = a \cdot x + b \cdot y + c = 0$, where the coefficients have been normalized such that $a^2 + b^2 = 1$. For a line L given in this Hessean normal form, and for any point $p = (x, y)$, the function h evaluated at p yields the signed distance between p and L: $h(p) > 0$ if p lies on one side of L, $h(p) < 0$ if p lies on the other side, and $h(p) = 0$ if p lies on L. A line segment is usually given by its endpoints (x_1, y_1) and (x_2, y_2), and the Hessean normal form of the infinitely extended line L that passes through (x_1, y_1) and (x_2, y_2) is

$$h(x,y) = \frac{(y_2 - y_1)\cdot(x - x_1) - (x_2 - x_1)\cdot(y - y_1)}{L_{12}} = 0,$$

where

$$L_{12} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} > 0$$

is the length of the line segment, and h(x, y) is the distance of $p = (x, y)$ from L. Two points p and q lie on opposite sides of L iff $h(p) \cdot h(q) < 0$ (Exhibit 14.2). $h(p) = 0$ or $h(q) = 0$ signals a degenerate configuration. Among these, $h(p) = 0$ and $h(q) = 0$ iff the segment (p, q) is collinear with L.

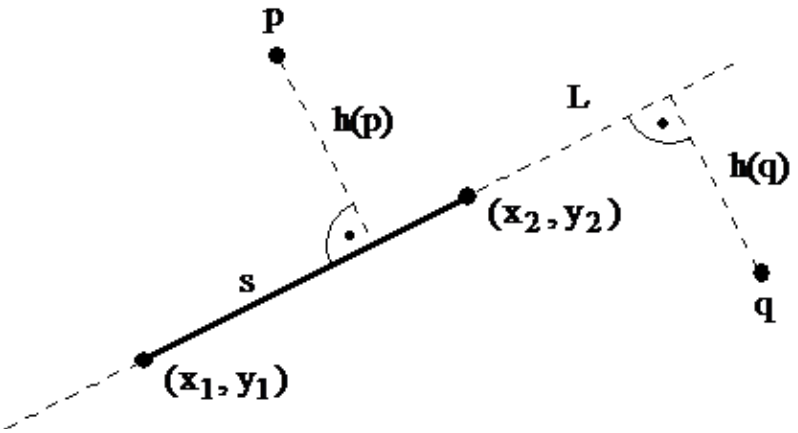

Exhibit 14.2: Segment s, its extended line L, and distance to points p, q as computed by function h.

```
type point   =  record  x, y: real  end;
     segment  =  record  p1, p2: point  end;

function d(s: segment; p: point): real;
{ computes h(p) for the line L determined by s }
var dx, dy, L12: real;
```

```
  begin
    dx := s.p2.x - s.p1.x;  dy := s.p2.y - s.p1.y;
    L12 := sqrt(dx · dx + dy · dy);
    return((dy · (p.x - s.p1.x) - dx · (p.y - s.p1.y)) / L12)
  end;
```

To optimize the intersection function, we recall the assumption $L_{12} > 0$ and notice that we do not need the actual distance, only its sign. Thus the function d used below avoids computing L_{12}. The function 'intersect' begins by checking whether the two line segments are collinear, and if so, tests them for overlap by intersecting the intervals obtained by projecting the line segments onto the x-axis (or onto the y-axis, if the segments are vertical). Two intervals [a, b] and [c, d] intersect iff min(a, b) ≤ max(c, d) and min(c, d) ≤ max(a, b). This condition could be simplified under the assumption that the representation of segments and intervals is ordered "from left to right" (i.e. for interval [a, b] we have a ≤ b). We do not assume this, as line segments often have a natural direction and cannot be "turned around".

```
  function d(s: segment;  p: point): real;
  begin
    return((s.p2.y - s.p1.y) · (p.x - s.p1.x) - (s.p2.x - s.p1.x) ·
(p.y - s.p1.y))
  end;

  function overlap(a, b, c, d: real): boolean;
  begin  return((min(a, b) ≤ max(c, d)) and (min(c, d) ≤ max(a, b)))
end;

  function intersect(s1, s2: segment): boolean;
  var  d11, d12, d21, d22: real;
  begin
    d11 := d(s1, s2.p1);  d12 := d(s1, s2.p2);
    if  (d11 = 0) and (d12 = 0)  then  { s1 and s2 are collinear }
      if  s1.p1.x = s1.p2.x  then  { vertical }
        return(overlap(s1.p1.y, s1.p2.y, s2.p1.y, s2.p2.y))
      else  { not vertical }
        return(overlap(s1.p1.x, s1.p2.x, s2.p1.x, s2.p2.x))
    else  begin  { s1 and s2 are not collinear }
      d21 := d(s2, s1.p1);  d22 := d(s2, s1.p2);
      return((d11 · d12 ≤ 0) and (d21 · d22 ≤ 0))
    end
  end;
```

In addition to the degeneracy issues we have addressed, there are numerical issues of near-degeneracy that we only mention. The length L_{12} is a *condition number* (i.e. an indicator of the computation's accuracy). As Exhibit 14.3 suggests, it may be numerically impossible to tell on which side of a short line segment L a distant point p lies.

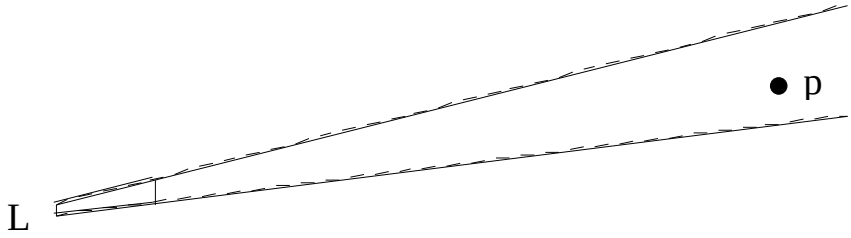

Exhibit 14.3: A point's distance from a segment amplifies the error of the "which side" computation.

Conclusion: A geometric algorithm must check for degenerate configurations explicitly—the code that handles configurations "in general position" will not handle degeneracies.

Clipping

The widespread use of windows on graphic screens makes clipping one of the most frequently executed operations: Given a rectangular window and a configuration in the plane, draw that part of the configuration which lies within the window. Most configurations consist of line segments, so we show how to clip a line segment given by its endpoints (x_1, y_1) and (x_2, y_2) into a window given by its four corners with coordinates {left, right} × {top, bottom}.

The position of a point in relation to the window is described by four boolean variables: ll (to the left of the left border), rr (to the right of the right border), bb (below the lower border), tt (above the upper border):

```
type  wcode = set of (ll, rr, bb, tt);
```

A point inside the window has the code ll = rr = bb = tt = false, abbreviated 0000 (Exhibit 14.4).

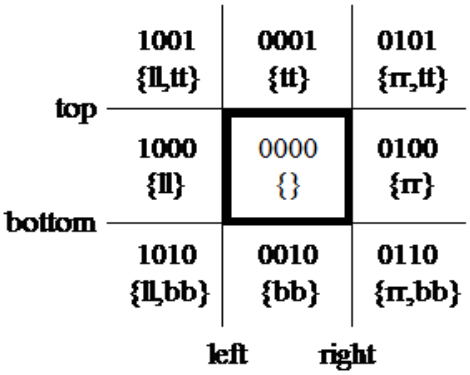

Exhibit 14.4: The clipping window partitions the plane into nine regions.

The procedure 'classify' determines the position of a point in relation to the window:

```
procedure classify(x, y: real; var c: wcode);
begin
  c := Ø;  { empty set }
  if  x < left  then  c := {ll}  elsif  x > right  then  c := {rr};
  if  y < bottom  then  c := c ∪ {bb}  elsif  y > top  then c := c ∪
{tt}
end;
```

The procedure 'clip' computes the endpoints of the clipped line segment and calls the procedure 'showline' to draw it:

```
procedure clip(x1, y1, x2, y2: real);
```

```
var  c, c1, c2: wcode;  x, y: real;  outside: boolean;
begin  { clip }
   classify(x1, y1, c1);  classify(x2, y2, c2);  outside := false;
   while  (c1 ≠ Ø) or (c2 ≠ Ø)  do
      if  c1 ∩ c2 ≠ Ø  then
         { line segment lies completely outside the window }
         { c1 := Ø;  c2 := Ø;  outside := true }
      else  begin
         c := c1;
         if  c = Ø  then  c := c2;
         if  ll ∈ c  then  { segment intersects left }
            { y := y1 + (y2 - y1) · (left - x1) / (x2 - x1);  x := left }
         elsif  rr ∈ c  then  { segment intersects right }
            { y := y1 + (y2 - y1) · (right - x1) / (x2 - x1); x := right }
         elsif  bb ∈ c  then  { segment intersects bottom }
            { x := x1 + (x2 - x1) · (bottom - y1) / (y2 - y1); y := bottom }
         elsif  tt ∈ c  then  { segment intersects top }
            { x := x1 + (x2 - x1) · (top - y1) / (y2 - y1);  y := top };
         if  c = c1 then { x1 := x;  y1 := y;  classify(x, y, c1) }
              else { x2 := x;  y2 := y;  classify(x, y, c2) }
      end;
   if  not outside  then  showline(x1, y1, x2, y2)
end;  { clip }
```

Drawing digitized lines

A raster graphics screen is an integer grid of pixels, each of which can be turned on or off. Euclidean geometry does not apply directly to such a discretized plane. Any designer using a CAD system will prefer Euclidean geometry to a discrete geometry as a model of the world. The problem of how to approximate the Euclidean plane by an integer grid turns out to be a hard question: How do we map Euclidean geometry onto a digitized space in such a way as to preserve the rich structure of geometry as much as possible? Let's begin with simple instances: How do you map a straight line onto an integer grid, and how do you draw it efficiently? Exhibit 14.5 shows reasonable examples.

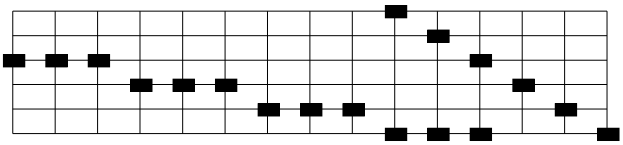

Exhibit 14.5: Digitized lines look like staircases.

Consider the slope $m = (y_2 - y_1) / (x_2 - x_1)$ of a segment with endpoints $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. If $|m| \le 1$ we want one pixel blackened on each x coordinate; if $|m| \ge 1$, one pixel on each y coordinate; these two requirements are consistent for diagonals with $|m| = 1$. Consider the case $|m| \le 1$. A unit step in x takes us from point (x, y) on the line to (x + 1, y + m). So for each x between x_1 and x_2 we paint the pixel (x, y) closest to the mathematical line according to the formula $y = \text{round}(y_1 + m \cdot (x - x_1))$. For the case $|m| > 1$, we reverse the roles of x and y, taking a unit step in y and incrementing x by 1/m. The following procedure draws line segments with $|m| \le 1$ using unit steps in x.

```
procedure line(x1, y1, x2, y2: integer);
var  x, sx: integer;  m: real;
```

```
begin
  PaintPixel(x1, y1);
  if  x1 ≠ x2  then  begin
    x := x1;  sx := sgn(x2 - x1);  m := (y2 - y1) / (x2 - x1);
    while  x ≠ x2  do
      { x := x + sx;  PaintPixel(x, round(y1 + m · (x - x1))) }
  end
end;
```

This straightforward implementation has a number of disadvantages. First, it uses floating-point arithmetic to compute integer coordinates of pixels, a costly process. In addition, rounding errors may prevent the line from being reversible: *reversibility* means that we paint the same pixels, in reverse order, if we call the procedure with the two endpoints interchanged. Reversibility is desirable to avoid the following blemishes: that a line painted twice, from both ends, looks thicker than other lines; worse yet, that painting a line from one end and erasing it from the other leaves spots on the screen. A weaker constraint, which is only concerned with the result and not the process of painting, is easy to achieve but is less useful.

Weak reversibility is most easily achieved by ordering the points p_1 and p_2 lexicographically by x and y coordinates, drawing every line from left to right, and vertical lines from bottom to top. This solution is inadequate for animation, where the direction of drawing is important, and the sequence in which the pixels are painted is determined by the application—drawing the trajectory of a falling apple from the bottom up will not do. Thus interactive graphics needs the stronger constraint.

Efficient algorithms, such as Bresenham's [Bre 65], avoid floating-point arithmetic and expensive multiplications through *incremental* computation: Starting with the current point p_1, a next point is computed as a function of the current point and of the line segment parameters. It turns out that only a few additions, shifts, and comparisons are required. In the following we assume that the slope m of the line satisfies $|m| \le 1$. Let

$$\Delta x = x_2 - x_1, \quad sx = \text{sign}(\Delta x), \qquad \Delta y = y_2 - y_1, \quad sy = \text{sign}(\Delta y).$$

Assume that the pixel (x, y) is the last that has been determined to be the closest to the actual line, and we now want to decide whether the next pixel to be set is (x + sx, y) or (x + sx, y + sy). Exhibit 14.6 depicts the case sx = 1 and sy = 1.

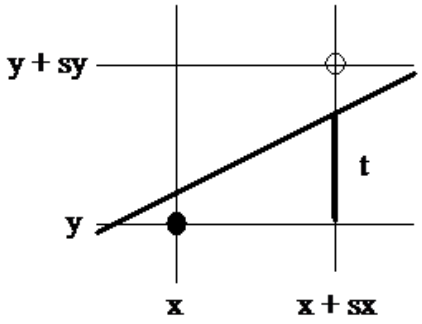

Exhibit 14.6: At the next coordinate x + sx, we identify and paint the pixel closest to the line.

Let t denote the absolute value of the difference between y and the point with abscissa x + sx on the actual line. Then t is given by

$$t = sy \cdot \left(y_1 + \frac{\Delta y}{\Delta x} \cdot (x + sx - x_1) - y \right).$$

The value of t determines the pixel to be drawn:

$$t < \frac{1}{2} \quad \Leftrightarrow \quad 2 \cdot t - 1 < 0: \qquad \mathbf{draw}(x + sx, y).$$

$$t > \frac{1}{2} \quad \Leftrightarrow \quad 2 \cdot t - 1 > 0: \qquad \mathbf{draw}(x + sx, y + sy).$$

As the following example shows, reversibility is not an automatic consequence of the geometric fact that two points determine a unique line, regardless of correct rounding or the order in which the two endpoints are presented. A problem arises when two grid points are equally close to the straight line (Exhibit 14.7).

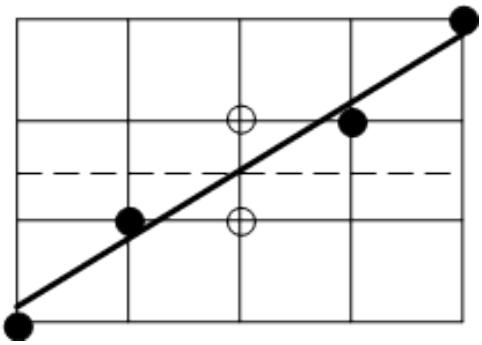

Exhibit 14.7: Breaking the tie among equidistant grid points.

If the tie is not broken in a consistent manner (e.g. by always taking the upper grid point), the resulting algorithm fails to be reversible:

$$t = \frac{1}{2} \quad \Leftrightarrow \quad 2 \cdot t - 1 = 0: \qquad \mathbf{draw} \begin{cases} (x + sx, y) & \text{if} \quad sy = -1, \\ (x + sx, y + sy) & \text{if} \quad sy = 1. \end{cases}$$

All the variables introduced in this problem range over the integers, but the ratio $\frac{(\Delta y)}{(\Delta x)}$ appears to introduce rational expressions. This is easily remedied by multiplying everything with Δx. We define the decision variable d as

$$d = |\Delta x| \cdot (2 \cdot t - 1) = sx \cdot \Delta x \cdot (2 \cdot t - 1). \qquad (*)$$

Let d_i denote the decision variable which determines the pixel $(x^{(i)}, y^{(i)})$ to be drawn in the i-th step. Substituting t and inserting $x = x^{(i-1)}$ and $y = y^{(i-1)}$ in $(*)$ we obtain

$$d_i = sx \cdot sy \cdot (2 \cdot \Delta x \cdot y_1 + 2 \cdot (x^{(i-1)} + sx - x_1) \cdot \Delta y - 2 \cdot \Delta x \cdot y^{(i-1)} - \Delta x \cdot sy)$$

and

$$d_{i+1} = sx \cdot sy \cdot (2 \cdot \mathbf{\Delta x} \cdot y_1 + 2 \cdot (x^{(i)} + sx - x_1) \cdot \Delta y - 2 \cdot \Delta x \cdot y^{(i)} - \Delta x \cdot sy).$$

Subtracting d_i from d_{i+1}, we get

$$d_{i+1} - d_i = sx \cdot sy \cdot (2 \cdot (x^{(i)} - x^{(i-1)}) \cdot \Delta y - 2 \cdot \Delta x \cdot (y^{(i)} - y^{(i-1)})).$$

Since $x^{(i)} - x^{(i-1)} = sx$, we obtain

$$d_{i+1} = d_i + 2 \cdot sy \cdot \Delta y - 2 \cdot sx \cdot \Delta x \cdot sy \cdot (y^{(i)} - y^{(i-1)}).$$

If $d_i < 0$, or $d_i = 0$ and $sy = -1$, then $y^{(i)} = y^{(i-1)}$, and therefore

$d_{i+1} = d_i + 2 \cdot |\Delta y|$.

If $d_i > 0$, or $d_i = 0$ and $sy = 1$, then $y^{(i)} = y^{(i-1)} + sy$, and therefore

$d_{i+1} = d_i + 2 \cdot |\Delta y| - 2 \cdot |\Delta x|$.

This iterative computation of d_{i+1} from the previous d_i lets us select the pixel to be drawn. The initial starting value for d_1 is found by evaluating the formula for d_i, knowing that $(x^{(0)}, y^{(0)}) = (x_1, y_1)$. Then we obtain

$d1 = 2 \cdot |\Delta y| - |\Delta x|$.

The arithmetic needed to evaluate these formulas is minimal: addition, subtraction and left shift (multiplication by 2). The following procedure implements this algorithm; it assumes that the slope of the line is between –1 and 1.

```
procedure BresenhamLine(x1, y1, x2, y2: integer);
var  dx, dy, sx, sy, d, x, y: integer;
begin
   dx := |x2 - x1|;  sx := sgn(x2 - x1);
   dy := |y2 - y1|;  sy := sgn(y2 - y1);
   d := 2 · dy - dx;  x := x1;  y := y1;
   PaintPixel(x, y);
   while  x ≠ x2  do  begin
      if  (d > 0) or ((d = 0) and (sy = 1))  then  { y := y + sy;-
2·dx};
      x := x + sx;  d := d + 2 · dy;
      PaintPixel(x, y)
   end
end;
```

The riddle of the braiding straight lines

Two straight lines in a plane intersect in at most one point, right? Important geometric algorithms rest on this well-known theorem of Euclidean geometry and would have to be reexamined if it were untrue. Is this theorem true for *computer lines*, that is, for data objects that represent and approximate straight lines to be processed by a program? Perhaps yes, but mostly no.

Yes. It is possible, of course, to program geometric problems in such a way that every pair of straight lines has at most, or exactly, one intersection point. This is most readily achieved through symbolic computation. For example, if the intersection of L_1 and L_2 is denoted by an expression 'Intersect(L_1, L_2)' that is never evaluated but simply combined with other expressions to represent a geometric construction, we are free to postulate that 'Intersect(L_1, L_2)' is a point.

No. For reasons of efficiency, most computer applications of geometry require the immediate numerical evaluation of every geometric operation. This calculation is done in a discrete, finite number system in which case the theorem need not be true. This fact is most easily seen if we work with a discrete plane of pixels, and we represent a straight line by the set of all pixels touched by an ideal mathematical line. Exhibit 14.8 shows three digitized straight lines in such a square grid model of plane geometry. Two of the lines intersect in a common interval of three pixels, whereas two others have no pixel in common, even though they obviously intersect.

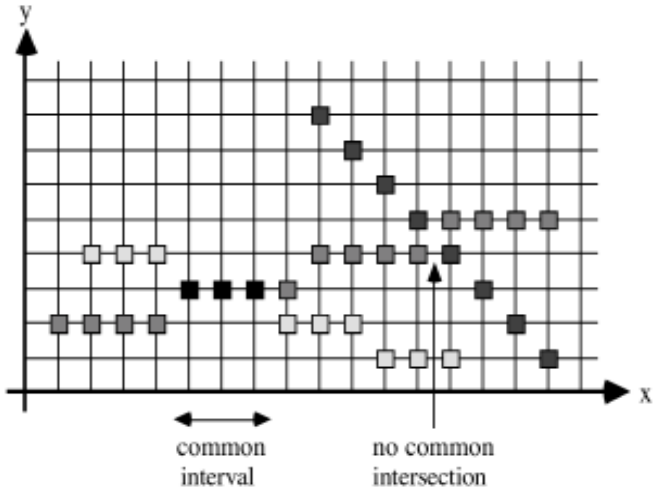

Exhibit 14.8: Two intersecting lines may share none, one, or more pixels.

With floating-point arithmetic the situation is more complicated; but the fact remains that the Euclidean plane is replaced by a discrete set of points embedded in the plane—all those points whose coordinates are representable in the particular number system being used. Experience with numerical computation, and the hazards of rounding errors, suggests that the question "In how many points can two straight lines intersect?" admits the following answers:

- There is no intersection—the mathematically correct intersection cannot be represented in the number system.
- A set of points that lie close to each other: for example, an interval.
- Overflow aborts the calculation before a result is computed, even if the correct result is representable in the number system being used.

Exercise: two lines intersect in how many points?

Construct examples to illustrate these phenomena when using floating-point arithmetic. Choose a suitable system G of floating-point numbers and two distinct straight lines

$a_i \cdot x + b_i \cdot y + c_i = 0$ with $a_i, b_i, c_i \in G$, i=1, 2,

such that, when all operations are performed in G:

(a) There is no point whose coordinates x, y ∈ G satisfy both linear equations.

(b) There are many points whose coordinates x, y ∈ G satisfy both linear equations.

(c) There is exactly one point whose coordinates x, y ∈ G satisfy both linear equations, but the straightforward computation of x and y leads to overflow.

(d) As a consequence of (a) it follows that the definition "two lines intersect they share a common point" is inappropriate for numerical computation. Formulate a numerically meaningful definition of the statement "two line segments intersect".

Exercise (b) may suggest that the points shared by two lines are neighbors. Pictorially, if the slopes of the two lines are almost identical, we expect to see a blurred, elongated intersection. We will show that worse things may happen: two straight lines may intersect in arbitrarily many points, and these points are separated by intervals in which the two lines alternate in lying on top of each other. Computer lines may be braided! To understand this

phenomenon, we need to clarify some concepts: What exactly is a straight line represented on a computer? What is an intersection?

There is no one answer, there are many! Consider the analogy of the mathematical concept of real numbers, defined by axioms. When we approximate real numbers on a computer, we have a choice of many different number systems (e.g. various floating-point number systems, rational arithmetic with variable precision, interval arithmetic). These systems are typically not defined by means of axioms, but rather in terms of concrete representations of the numbers and algorithms for executing the operations on these numbers. Similarly, a *computer line* will be defined in terms of a concrete representation (e.g. two points, a point and a slope, or a linear expression). All we obtain depends on the formulas we use and on the basic arithmetic to operate on these representations. The notion of a straight line can be formalized in many different ways, and although these are likely to be mathematically equivalent, they will lead to data objects with different behavior when evaluated numerically. Performing an operation consists of evaluating a formula. Substituting a formula by a mathematically equivalent one may lead to results that are topologically different, because equivalent formulas may exhibit different sensitivities toward rounding errors.

Consider a computer that has only integer arithmetic, i.e. we use only the operations +, –, ·, div. Let Z be the set of integers. Two straight lines g_i (i = 1, 2) are given by the following equations:

$$a_i \cdot x + b_i \cdot y + c_i = 0 \quad \text{with} \quad a_i, b_i, c_i \in Z; \quad b_i \neq 0.$$

We consider the problem of whether two given straight lines intersect in a given point x_0. We use the following method: Solve the equations for y [i. e. $y = E_1(x)$ and $y = E_2(x)$] and test whether $E_1(x_0)$ is equal to $E_2(x_0)$.

Is this method suitable? First, we need the following definitions:

$$\mathbf{sign(x)}:=\begin{cases} \mathbf{1} & \mathbf{if\ x>0,} \\ \mathbf{0} & \mathbf{if\ x=0,} \\ \mathbf{-1} & \mathbf{if\ x<0.} \end{cases}$$

$x \in Z$ is a *turn* for the pair (E_1, E_2) iff

$$\text{sign}(E_1(x) - E_2(x)) \neq \text{sign}(E_1(x+1) - E_2(x+1)).$$

An algorithm for the intersection problem is correct iff there are at most two turns.

The intuitive idea behind this definition is the recognition that rounding errors may force us to deal with an intersection interval rather than a single intersection point; but we wish to avoid separate intervals. The definition above partitions the x-axis into at most three disjoint intervals such that in the left interval the first line lies above or below the second line, in the middle interval the lines "intersect", and in the right interval we have the complementary relation of the left one (Exhibit 14.9).

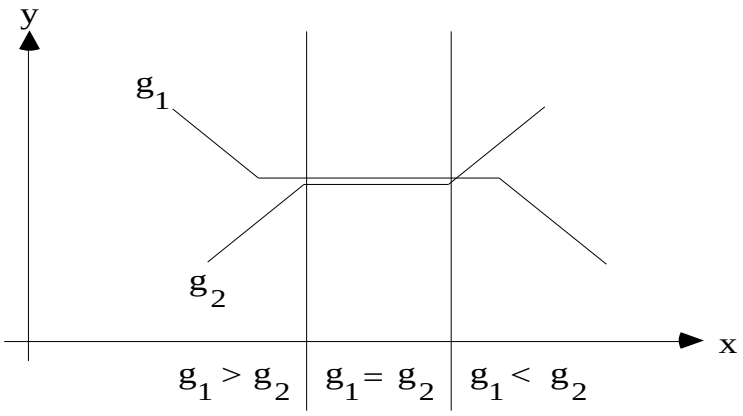

Exhibit 14.9: Desirable consistency condition for intersection of nearly parallel lines.

Consider the straight lines:

$3 \cdot x - 5 \cdot y + 40 = 0$ and $2 \cdot x - 3 \cdot y + 20 = 0$

which lead to the evaluation formulas

$$y = \frac{3 \cdot x + 40}{5} \quad \text{and} \quad y = \frac{2 \cdot x + 20}{3}.$$

Our naive approach compares the expressions

$(3 \cdot x + 40)$ div 5 and $(2 \cdot x + 20)$ div 3.

Using the definitions it is easy to calculate that the turns are

7, 8, 10, 11, 12, 14, 15, 22, 23, 25, 26, 27, 29, 30.

The straight lines have become step functions that intersect many times. They are braided (Exhibit 14.10).

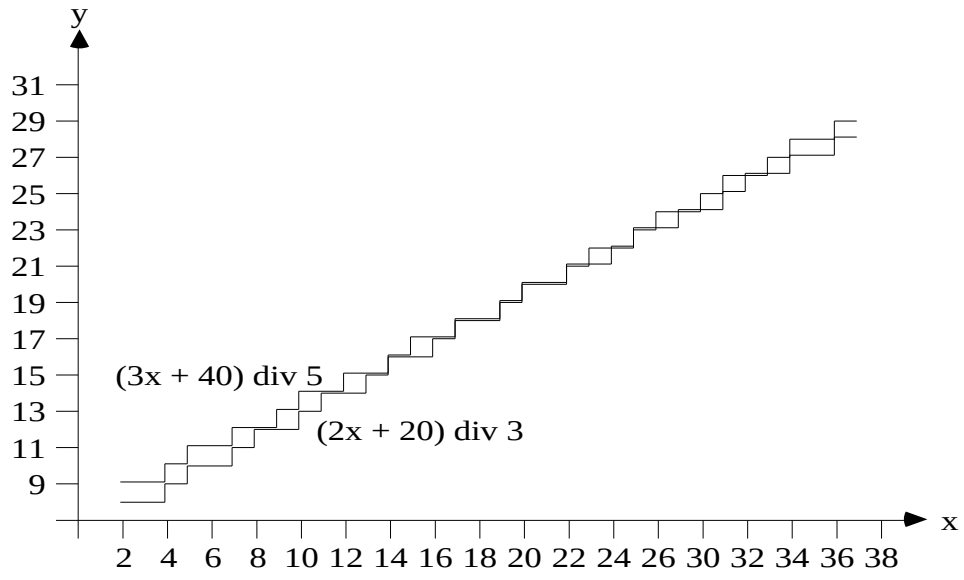

Exhibit 14.10: Braiding straight lines violate the consistency condition of Exhibit 14.9.

Exercise: show that the straight lines

$x - 2 \cdot y = 0$

$k \cdot x - (2 \cdot k + 1) \cdot y = 0$ for any integer $k > 0$

have $2 \cdot k - 1$ turns in the first quadrant.

Is braiding due merely to integer arithmetic? Certainly not: rounding errors also occur in floating-point arithmetic, and we can construct even more pathological behavior. As an example, consider a floating-point arithmetic with a two-decimal-digit mantissa. We perform the evaluation operation:

$$y = -\frac{a \cdot x + c}{b}$$

and truncate intermediate results immediately to two decimal places. Consider the straight lines (Exhibit 14.11)

$4.3 \cdot x - 8.3 \cdot y = 0,$

$1.4 \cdot x - 2.7 \cdot y = 0.$

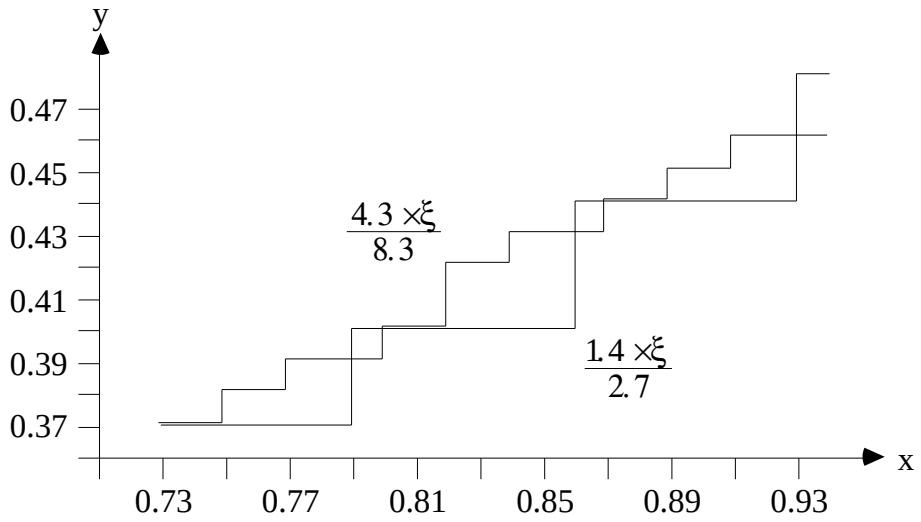

Exhibit 14.11: Example to be verified by manual computation.

These examples were constructed by intersecting straight lines with almost the same slope—a numerically ill-conditioned problem. While working with integer arithmetic, we made the mistake of using the error-prone 'div' operator. The comparison of rational expressions does not require division.

Let $a_1 \cdot x + b_1 \cdot y + c_1 = 0$ and $a_2 \cdot x + b_2 \cdot y + c_2 = 0$ be two straight lines. To find out whether they intersect at x_0, we have to check whether the equality

$$\frac{-c_1 - a_1 \cdot x_0}{b_1} = \frac{-c_2 - a_2 \cdot x_0}{b_2}$$

holds. This is equivalent to $b_2 \cdot c_1 - b_1 \cdot c_2 = x_0 \cdot (a_2 \cdot b_1 - a_1 \cdot b_2)$.

The last formula can be evaluated without error if sufficiently large integer arguments are allowed. Another way to evaluate this formula without error is to limit the size of the operands. For example, if a_i, b_i, c_i, and x_0 are n-digit binary numbers, it suffices to be able to represent 3n-digit binary numbers and to compute with n-digit and 2n-digit binary numbers.

These examples demonstrate that programming even a simple geometric problem can cause unexpected difficulties. Numerical computation forces us to rethink and redefine elementary geometric concepts.

Digitized circles

The concepts, problems and techniques we have discussed in this chapter are not at all restricted to dealing with straight lines—they have their counterparts for any kind of digitized spatial object. Straight lines, defined by linear formulas, are the simplest nontrivial spatial objects and thus best suited to illustrate problems and solutions. In this section we show that the incremental drawing technique generalizes in a straightforward manner to more complex objects such as circles.

The basic parameters that define a circle are the center coordinates (x_c, y_c) and the radius r. To simplify the presentation we first consider a circle with radius r centered around the origin. Such a circle is given by the equation

$$x^2 + y^2 = r^2.$$

Efficient algorithms for drawing circles, such as Bresenham's [Bre 77], avoid floating-point arithmetic and expensive multiplications through *incremental* computation: A new point is computed depending on the current point and on the circle parameters. Bresenham's circle algorithm was conceived for use with pen plotters and therefore generates all points on a circle centered at the origin by incrementing all the way around the circle. We present a modified version of his algorithm which takes advantage of the *eight-way symmetry* of a circle. If (x, y) is a point on the circle, we can easily determine seven other points lying on the circle (Exhibit 14.12). We consider only the 45° segment of the circle shown in the figure by incrementing from x = 0 to x = y = r / , and use eight-way symmetry to display points on the entire circle.

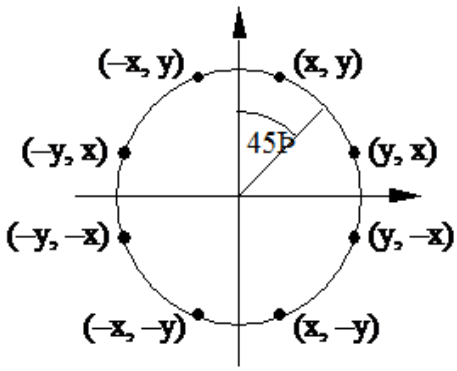

Exhibit 14.12: Eightfold symmetry of the circle.

Assume that the pixel p = (x, y) is the last that has been determined to be closest to the actual circle, and we now want to decide whether the next pixel to be set is p1 = (x + 1, y) or p_2 = (x + 1, y – 1). Since we restrict ourselves to the 45° circle segment shown above these pixels are the only candidates. Now define

$$d' = (x + 1)^2 + y^2 - r^2$$

$$d'' = (x + 1)^2 + (y - 1)^2 - r^2$$

which are the differences between the squared distances from the center of the circle to p_1 (or p_2) and to the actual circle. If $|d'| \le |d''|$, then p_1 is closer (or equidistant) to the actual circle; if $|d'| > |d''|$, then p_2 is closer. We define the decision variable d as

$$d = d' + d''. \qquad (*)$$

We will show that the rule

If $d \le 0$ then select p_1 else select p_2.

correctly selects the pixel that is closest to the actual circle. Exhibit 14.13 shows a small part of the pixel grid and illustrates the various possible ways [(1) to (5)] how the actual circle may intersect the vertical line at x + 1 in relation to the pixels p_1 and p_2.

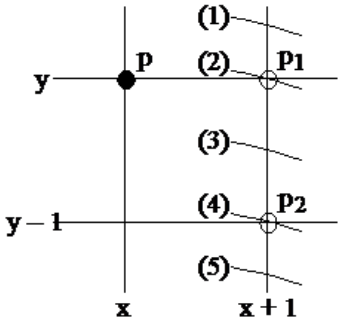

Exhibit 14.13: For a given octant of the circle, if pixel p is lit, only two other pixels p_1 and p_2 need be examined.

In cases (1) and (2) p_2 lies inside, p_1 inside or on the circle, and we therefore obtain $d' \le 0$ and $d'' < 0$. Now $d < 0$, and applying the rule above will lead to the selection of p_1. Since $|d'| \le |d''|$ this selection is correct. In case (3) p_1 lies outside and p_2 inside the circle and we therefore obtain $d' > 0$ and $d'' < 0$. Applying the rule above will lead to the selection of p_1 if $d \le 0$, and p_2 if $d > 0$. This selection is correct since in this case $d \le 0$ is equivalent to $|d'| \le |d''|$. In cases (4) and (5) p_1 lies outside, p_2 outside or on the circle and we therefore obtain $d' > 0$ and $d'' \ge 0$. Now $d > 0$, and applying the rule above will lead to the selection of p_2. Since $|d'| > |d''|$ this selection is correct.

Let d_i denote the decision variable that determines the pixel $(x^{(i)}, y^{(i)})$ to be drawn in the i-th step. Starting with $(x^{(o)}, y^{(o)}) = (0, r)$ we obtain

$$d_1 = 3 - 2 \cdot r.$$

If $d_i \le 0$, then $(x^{(i)}, y^{(i)}) = (x^{(i)} + 1, y^{(i-1)})$, and therefore

$$d_{i+1} = d_i + 4 \cdot x_{i-1} + 6.$$

If $d_i > 0$, then $(x^{(i)}, y^{(i)}) = (x^{(i)} + 1, y^{(i-1)} - 1)$, and therefore

$$d_{i+1} = d_i + 4 \cdot (x_{i-1} - y_{i-1}) + 10.$$

This iterative computation of d_{i+1} from the previous d_i lets us select the correct pixel to be drawn in the (i + 1)-th step. The arithmetic needed to evaluate these formulas is minimal: addition, subtraction, and left shift (multiplication by 4). The following procedure 'BresenhamCircle' which implements this algorithm draws a circle with center (x_c, y_c) and radius r. It uses the procedure 'CirclePoints' to display points on the entire circle. In the cases x = y or r = 1 'CirclePoints' draws each of four pixels twice. This causes no problem on a raster display.

```
procedure BresenhamCircle(xC, yC, r: integer);

  procedure CirclePoints(x, y: integer);
  begin
    PaintPixel(xC + x, yC + y); PaintPixel(xC - x, yC + y);
    PaintPixel(xC + x, yC - y); PaintPixel(xC - x, yC - y);
    PaintPixel(xC + y, yC + x); PaintPixel(xC - y, yC + x);
    PaintPixel(xC + y, yC - x); PaintPixel(xC - y, yC - x)
  end;
```

```
var  x, y, d: integer;
begin
  x := 0;  y := r;  d := 3 - 2 · r;
  while  x < y  do  begin
    CirclePoints(x, y);
    if  d < 0 then d := d + 4 · x + 6
      else { d := d + 4 · (x - y) + 10;  y := y - 1 };
    x := x + 1
  end;
  if  x = y  then  CirclePoints(x, y)
end;    .i).Bresenham's algorithm:circle;
```

Exercises and programming projects

1. Design and implement an efficient geometric primitive which determines whether two aligned rectangles (i.e. rectangles with sides parallel to the coordinate axes) intersect.
2. Design and implement a geometric primitive

   ```
   function inTriangle(t: triangle; p: point): …;
   ```

   which takes a triangle t given by its three vertices and a point p and returns a ternary value: p is inside t, p is on the boundary of t, p is outside t.
3. Use the functions 'intersect' of in "Intersection" and 'inTriangle' above to program a

   ```
   function SegmentIntersectsTriangle(s: segment; t: triangle): …;
   ```

   to check whether segment s and triangle t share common points. 'SegmentIntersectsTriangle' returns a ternary value: yes, degenerate, no. List all distinct cases of degeneracy that may occur, and show how your code handles them.
4. Implement Bresenham's incremental algorithms for drawing digitized straight lines and circles.
5. Two circles (x', y', r') and (x'', y'', r'') are given by the coordinates of their center and their radius. Find effective formulas for deciding the three-way question whether (a) the circles intersect as lines, (b) the circles intersect as disks, or (c) neither. Avoid the square-root operation whenever possible.