3.2 SCAN CONVERTING LINES

A scan-conversion algorithm for lines computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line imposed on a 2D raster grid. In principle, we would like the sequence of pixels to lie as close to the ideal line as possible and to be as straight as possible. Consider a 1-pixel-thick approximation to an ideal line; what properties should it have? For lines with slopes between −1 and 1 inclusive, exactly 1 pixel should be illuminated in each column; for lines with slopes outside this range, exactly 1 pixel should be illuminated in each row. All lines should be drawn with constant brightness, independent of length and orientation, and as rapidly as possible. There should also be provisions for drawing lines that are more than 1 pixel wide, centered on the ideal line, that are affected by line-style and pen-style attributes, and that create other effects needed for high-quality illustrations. For example, the shape of the endpoint regions should be under programmer control to allow beveled, rounded, and mitered corners. We would even like to be able to minimize the jaggies due to the discrete approximation of the ideal line by using antialiasing techniques exploiting the ability to set the intensity of individual pixels on n-bits-per-pixel displays.

For now, we consider only "optimal," 1-pixel-thick lines that have exactly 1 bilevel pixel in each column (or row for steep lines). Later in the chapter, we consider thick primitives and deal with styles.

To visualize the geometry, we recall that SRGP represents a pixel as a circular dot centered at that pixel's (x, y) location on the integer grid. This representation is a convenient approximation to the more or less circular cross-section of the CRT's electron beam, but the exact spacing between the beam spots on an actual display can vary greatly among systems. In some systems, adjacent spots overlap; in others, there may be space between adjacent vertical pixels; in most systems, the spacing is tighter in the horizontal than in the vertical direction. Another variation in coordinate-system representation arises in systems, such as the Macintosh, that treat pixels as being centered in the rectangular box between adjacent grid lines instead of on the grid lines themselves. In this scheme, rectangles are defined to be all pixels interior to the mathematical rectangle defined by two corner points. This definition allows zero-width (null) canvases: The rectangle from (x, y) to (x, y) contains no pixels, unlike the SRGP canvas, which has a single pixel at that point. For now, we continue to represent pixels as disjoint circles centered on a uniform grid, although we shall make some minor changes when we discuss antialiasing.

Figure 3.4 shows a highly magnified view of a 1-pixel-thick line and of the ideal line that it approximates. The intensified pixels are shown as filled circles and the nonintensified

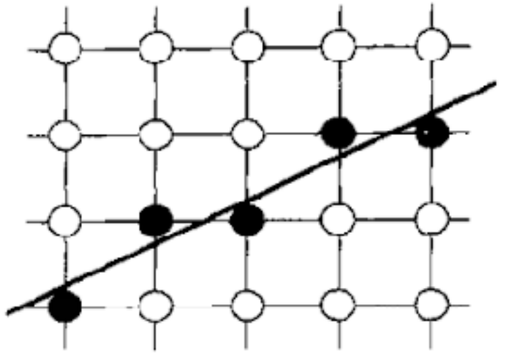


Fig. 3.4 A scan-converted line showing intensified pixels as black circles.

pixels are shown as unfilled circles. On an actual screen, the diameter of the roughly circular pixel is larger than the interpixel spacing, so our symbolic representation exaggerates the discreteness of the pixels.

Since SRGP primitives are defined on an integer grid, the endpoints of a line have integer coordinates. In fact, if we first clip the line to the clip rectangle, a line intersecting a clip edge may actually have an endpoint with a noninteger coordinate value. The same is true when we use a floating-point raster graphics package. (We discuss these noninteger intersections in Section 3.2.3.) Assume that our line has slope $|m| \leq 1$; lines at other slopes can be handled by suitable changes in the development that follows. Also, the most common lines—those that are horizontal, are vertical, or have a slope of ± 1—can be handled as trivial special cases because these lines pass through only pixel centers (see Exercise 3.1).

3.2.1 The Basic Incremental Algorithm

The simplest strategy for scan conversion of lines is to compute the slope m as $\Delta y/\Delta x$, to increment x by 1 starting with the leftmost point, to calculate $y_i = mx_i + B$ for each x_i, and to intensify the pixel at $(x_i, \text{Round}(y_i))$, where $\text{Round}(y_i) = \text{Floor}(0.5 + y_i)$. This computation selects the closest pixel—that is, the pixel whose distance to the true line is smallest.[1] This brute-force strategy is inefficient, however, because each iteration requires a floating-point (or binary fraction) multiply, addition, and invocation of Floor. We can eliminate the multiplication by noting that

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x,$$

and, if $\Delta x = 1$, then $y_{i+1} = y_i + m$.

Thus, a unit change in x changes y by m, which is the slope of the line. For all points (x_i, y_i) on the line, we know that, if $x_{i+1} = x_i + 1$, then $y_{i+1} = y_i + m$; that is, the values of x and y are defined in terms of their previous values (see Fig. 3.5). This is what defines an

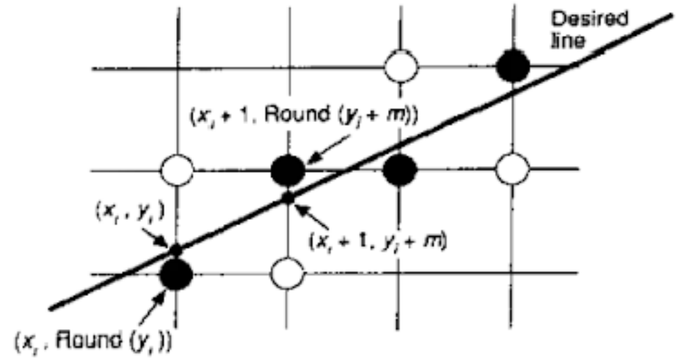


Fig. 3.5 Incremental calculation of (x_i, y_i).

incremental algorithm: At each step, we make incremental calculations based on the preceding step.

We initialize the incremental calculation with (x_0, y_0), the integer coordinates of an endpoint. Note that this incremental technique avoids the need to deal with the y intercept, B, explicitly. If $|m| > 1$, a step in x creates a step in y that is greater than 1. Thus, we must reverse the roles of x and y by assigning a unit step to y and incrementing x by $\Delta x = \Delta y/m = 1/m$. Line, the procedure in Fig. 3.6, implements this technique. The start point must be the left endpoint. Also, it is limited to the case $-1 \leq m \leq 1$, but other slopes may be accommodated by symmetry. The checking for the special cases of horizontal, vertical, or diagonal lines is omitted.

WritePixel, used by Line, is a low-level procedure provided by the device-level software; it places a value into a canvas for a pixel whose coordinates are given as the first two arguments.[2] We assume here that we scan convert only in replace mode; for SRGP's other write modes, we must use a low-level ReadPixel procedure to read the pixel at the destination location, logically combine that pixel with the source pixel, and then write the result into the destination pixel with WritePixel.

This algorithm is often referred to as a *digital differential analyzer (DDA)* algorithm. The DDA is a mechanical device that solves differential equations by numerical methods: It traces out successive (x, y) values by simultaneously incrementing x and y by small steps proportional to the first derivative of x and y. In our case, the x increment is 1, and the y increment is $dy/dx = m$. Since real variables have limited precision, summing an inexact m repetitively introduces cumulative error buildup and eventually a drift away from a true Round(y_i); for most (short) lines, this will not present a problem.

3.2.2 Midpoint Line Algorithm

The drawbacks of procedure Line are that rounding y to an integer takes time, and that the variables y and m must be real or fractional binary because the slope is a fraction. Bresenham developed a classic algorithm [BRES65] that is attractive because it uses only


```
void Line (                                  /* Assumes -1 ≤ m ≤ 1, x0 < x1 */
      int x0, int y0,                        /* Left endpoint */
      int x1, int y1,                        /* Right endpoint */
      int value)                             /* Value to place in line's pixels */
{
   int x;                                    /* x runs from x0 to x1 in unit increments. */

   double dy = y1 - y0;
   double dx = x1 - x0;
   double m = dy / dx;
   double y = y0;
   for (x = x0; x <= x1; x++) {
      WritePixel (x, Round (y), value);      /* Set pixel to value */
      y += m;                                /* Step y by slope m */
   }
} /* Line */
```

Fig. 3.6 The incremental line scan-conversion algorithm.

integer arithmetic, thus avoiding the Round function, and allows the calculation for (x_{i+1}, y_{i+1}) to be performed incrementally—that is, by using the calculation already done at (x_i, y_i). A floating-point version of this algorithm can be applied to lines with arbitrary real-valued endpoint coordinates. Furthermore, Bresenham's incremental technique may be applied to the integer computation of circles as well, although it does not generalize easily to arbitrary conics. We therefore use a slightly different formulation, the *midpoint technique*, first published by Pitteway [PITT67] and adapted by Van Aken [VANA84] and other researchers. For lines and integer circles, the midpoint formulation, as Van Aken shows [VANA85], reduces to the Bresenham formulation and therefore generates the same pixels. Bresenham showed that his line and integer circle algorithms provide the best-fit approximations to true lines and circles by minimizing the error (distance) to the true primitive [BRES77]. Kappel discusses the effects of various error criteria in [KAPP85].

We assume that the line's slope is between 0 and 1. Other slopes can be handled by suitable reflections about the principal axes. We call the lower-left endpoint (x_0, y_0) and the upper-right endpoint (x_1, y_1).

Consider the line in Fig. 3.7, where the previously selected pixel appears as a black circle and the two pixels from which to choose at the next stage are shown as unfilled circles. Assume that we have just selected the pixel P at (x_P, y_P) and now must choose between the pixel one increment to the right (called the east pixel, E) or the pixel one increment to the right and one increment up (called the northeast pixel, NE). Let Q be the intersection point of the line being scan-converted with the grid line $x = x_P + 1$. In Bresenham's formulation, the difference between the vertical distances from E and NE to Q is computed, and the sign of the difference is used to select the pixel whose distance from Q is smaller as the best approximation to the line. In the midpoint formulation, we observe on which side of the line the midpoint M lies. It is easy to see that, if the midpoint lies above the line, pixel E is closer to the line; if the midpoint lies below the line, pixel NE is closer to the line. The line may pass between E and NE, or both pixels may lie on one side, but in any

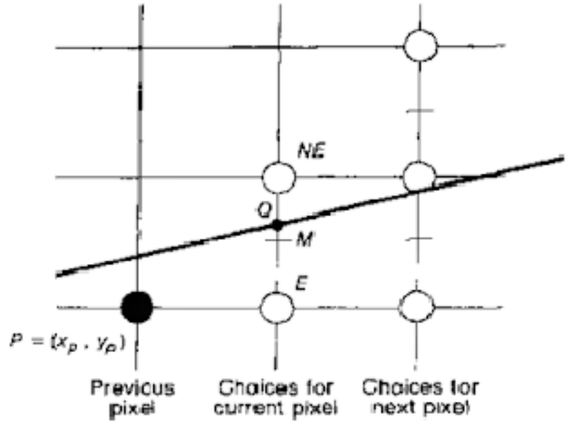


Fig. 3.7 The pixel grid for the midpoint line algorithm, showing the midpoint M, and the E and NE pixels to choose between.

case, the midpoint test chooses the closest pixel. Also, the error—that is, the vertical distance between the chosen pixel and the actual line—is always $\leq 1/2$.

The algorithm chooses NE as the next pixel for the line shown in Fig. 3.7. Now all we need is a way to calculate on which side of the line the midpoint lies. Let's represent the line by an implicit function[3] with coefficients a, b, and c: $F(x, y) = ax + by + c = 0$. (The b coefficient of y is unrelated to the y intercept B in the slope-intercept form.) If $dy = y_1 - y_0$, and $dx = x_1 - x_0$, the slope-intercept form can be written as

$$y = \frac{dy}{dx}x + B;$$

therefore,

$$F(x, y) = dy \cdot x - dx \cdot y + B \cdot dx = 0.$$

Here $a = dy$, $b = -dx$, and $c = B \cdot dx$ in the implicit form.[4]

It can easily be verified that $F(x, y)$ is zero on the line, positive for points below the line, and negative for points above the line. To apply the midpoint criterion, we need only to compute $F(M) = F(x_P + 1, y_P + \frac{1}{2})$ and to test its sign. Because our decision is based on the value of the function at $(x_P + 1, y_P + \frac{1}{2})$, we define a *decision variable* $d = F(x_P + 1, y_P + \frac{1}{2})$. By definition, $d = a(x_P + 1) + b(y_P + \frac{1}{2}) + c$. If $d > 0$, we choose pixel NE; if $d < 0$, we choose E; and if $d = 0$, we can choose either, so we pick E.

Next, we ask what happens to the location of M and therefore to the value of d for the next grid line; both depend, of course, on whether we chose E or NE. If E is chosen, M is

[3] This functional form extends nicely to the implicit formulation of both circles and ellipses.

[4] It is important for the proper functioning of the midpoint algorithm to choose a to be positive; we meet this criterion if dy is positive, since $y_1 > y_0$.

incremented by one step in the x direction. Then,

$$d_{new} = F(x_P + 2, y_P + \tfrac{1}{2}) = a(x_P + 2) + b(y_P + \tfrac{1}{2}) + c,$$

but

$$d_{old} = a(x_P + 1) + b(y_P + \tfrac{1}{2}) + c.$$

Subtracting d_{old} from d_{new} to get the incremental difference, we write $d_{new} = d_{old} + a$.

We call the increment to add after E is chosen Δ_E; $\Delta_E = a = dy$. In other words, we can derive the value of the decision variable at the next step incrementally from the value at the current step without having to compute $F(M)$ directly, by merely adding Δ_E.

If NE is chosen, M is incremented by one step each in both the x and y directions. Then,

$$d_{new} = F(x_P + 2, y_P + \tfrac{3}{2}) = a(x_P + 2) + b(y_P + \tfrac{3}{2}) + c.$$

Subtracting d_{old} from d_{new} to get the incremental difference, we write

$$d_{new} = d_{old} + a + b.$$

We call the increment to add to d after NE is chosen Δ_{NE}; $\Delta_{NE} = a + b = dy - dx$.

Let's summarize the incremental midpoint technique. At each step, the algorithm chooses between 2 pixels based on the sign of the decision variable calculated in the previous iteration; then, it updates the decision variable by adding either Δ_E or Δ_{NE} to the old value, depending on the choice of pixel.

Since the first pixel is simply the first endpoint (x_0, y_0), we can directly calculate the initial value of d for choosing between E and NE. The first midpoint is at $(x_0 + 1, y_0 + \frac{1}{2})$, and

$$\begin{aligned} F(x_0 + 1, y_0 + \tfrac{1}{2}) &= a(x_0 + 1) + b(y_0 + \tfrac{1}{2}) + c \\ &= ax_0 + by_0 + c + a + b/2 \\ &= F(x_0, y_0) + a + b/2. \end{aligned}$$

But (x_0, y_0) is a point on the line and $F(x_0, y_0)$ is therefore 0; hence, d_{start} is just $a + b/2 = dy - dx/2$. Using d_{start}, we choose the second pixel, and so on. To eliminate the fraction in d_{start}, we redefine our original F by multiplying it by 2; $F(x, y) = 2(ax + by + c)$. This multiplies each constant and the decision variable by 2, but does not affect the sign of the decision variable, which is all that matters for the midpoint test.

The arithmetic needed to evaluate d_{new} for any step is simple addition. No time-consuming multiplication is involved. Further, the inner loop is quite simple, as seen in the midpoint algorithm of Fig. 3.8. The first statement in the loop, the test of d, determines the choice of pixel, but we actually increment x and y to that pixel location after updating the decision variable (for compatibility with the circle and ellipse algorithms). Note that this version of the algorithm works for only those lines with slope between 0 and 1; generalizing the algorithm is left as Exercise 3.2. In [SPRO82], Sproull gives an elegant derivation of Bresenham's formulation of this algorithm as a series of program transformations from the original brute-force algorithm. No equivalent of that derivation for circles or ellipses has yet appeared, but the midpoint technique does generalize, as we shall see.

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = 2 * dy - dx,              /* Initial value of d */
    int incrE = 2 * dy;               /* Increment used for move to E */
    int incrNE = 2 * (dy - dx);       /* Increment used for move to NE */
    int x = x0;
    int y = y0;
    WritePixel (x, y, value);         /* The start pixel */

    while (x < x1) {
        if (d <= 0) {                 /* Choose E */
            d += incrE;
            x++;
        } else {                      /* Choose NE */
            d += incrNE;
            x++;
            y++;
        }
        WritePixel (x, y, value);     /* The selected pixel closest to the line */
    }  /* while */

}  /* MidpointLine */
```

Fig. 3.8 The midpoint line scan-conversion algorithm.

For a line from point (5, 8) to point (9, 11), the successive values of d are 2, 0, 6, and 4, resulting in the selection of NE, E, NE, and then NE, respectively, as shown in Fig. 3.9. The line appears abnormally jagged because of the enlarged scale of the drawing and the artificially large interpixel spacing used to make the geometry of the algorithm clear. For the same reason, the drawings in the following sections also make the primitives appear blockier than they look on an actual screen.

3.2.3 Additional Issues

Endpoint order. Among the complications to consider is that we must ensure that a line from P_0 to P_1 contains the same set of pixels as the line from P_1 to P_0, so that the appearance of the line is independent of the order of specification of the endpoints. The only place where the choice of pixel is dependent on the direction of the line is where the line passes exactly through the midpoint and the decision variable is zero; going left to right, we chose to pick E for this case. By symmetry, while going from right to left, we would also expect to choose W for $d = 0$, but that would choose a pixel one unit up in y relative to the one chosen for the left-to-right scan. We therefore need to choose SW when $d = 0$ for right-to-left scanning. Similar adjustments need to be made for lines at other slopes.

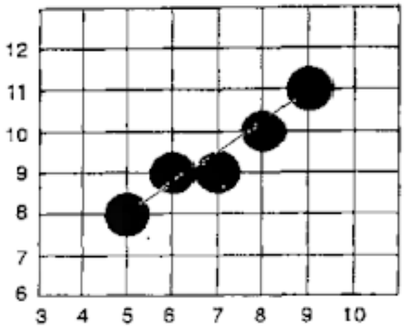


Fig. 3.9 The midpoint line from point (5, 8) to point (9, 11).

The alternative solution of switching a given line's endpoints as needed so that scan conversion always proceeds in the same direction does not work when we use line styles. The line style always "anchors" the specified write mask at the start point, which would be the bottom-left point, independent of line direction. That does not necessarily produce the desired visual effect. In particular, for a dot-dash line pattern of, say, 111100, we would like to have the pattern start at whichever start point is specified, not automatically at the bottom-left point. Also, if the algorithm always put endpoints in a canonical order, the pattern might go left to right for one segment and right to left for the adjoining segment, as a function of the second line's slope; this would create an unexpected discontinuity at the shared vertex, where the pattern should follow seamlessly from one line segment to the next.

Starting at the edge of a clip rectangle. Another issue is that we must modify our algorithm to accept a line that has been analytically clipped by one of the algorithms in Section 3.12. Fig. 3.10(a) shows a line being clipped at the left edge, $x = x_{min}$, of the clip rectangle. The intersection point of the line with the edge has an integer x coordinate but a real y coordinate. The pixel at the left edge, $(x_{min}, \text{Round}(mx_{min} + B))$, is the same pixel that would be drawn at this x value for the unclipped line by the incremental algorithm.[5] Given this initial pixel value, we must next initialize the decision variable at the midpoint between the E and NE positions in the next column over. It is important to realize that this strategy produces the correct sequence of pixels, while clipping the line at the x_{min} boundary and then scan converting the clipped line from $(x_{min}, \text{Round}(mx_{min} + B))$ to (x_1, y_1) using the integer midpoint line algorithm would not—that clipped line has a different slope!

The situation is more complicated if the line intersects a horizontal rather than a vertical edge, as shown in Fig. 3.10 (b). For the type of shallow line shown, there will be multiple pixels lying on the scan line $y = y_{min}$ that correspond to the bottom edge of the clip region. We want to count each of these as inside the clip region, but simply computing the analytical intersection of the line with the $y = y_{min}$ scan line and then rounding the x value of the intersection point would produce pixel A, not the leftmost point of the span of pixels shown, pixel B. From the figure, it is clear that the leftmost pixel of the span, B, is the one

[5] When $mx_{min} + B$ lies exactly halfway between horizontal grid lines, we actually must round down. This is a consequence of choosing pixel E when $d = 0$.

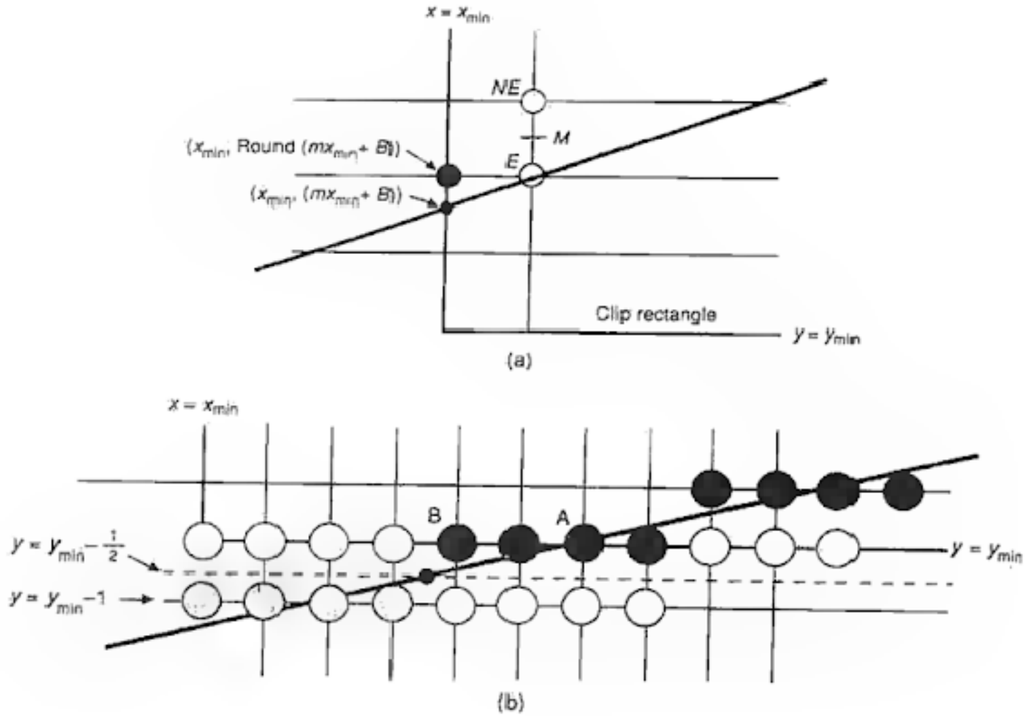


Fig. 3.10 Starting the line at a clip boundary. (a) Intersection with a vertical edge. (b) Intersection with a horizontal edge (grey pixels are on the line but are outside the clip rectangle).

that lies just above and to the right of the place on the grid where the line first crosses above the midpoint $y = y_{min} - \frac{1}{2}$. Therefore, we simply find the intersection of the line with the horizontal line $y = y_{min} - \frac{1}{2}$, and round up the x value; the first pixel, B, is then the one at $(\text{Round}(x_{y_{min}-\frac{1}{2}}), y_{min})$.

Finally, the incremental midpoint algorithm works even if endpoints are specified in a floating-point raster graphics package; the only difference is that the increments are now reals, and the arithmetic is done with reals.

Varying the intensity of a line as a function of slope. Consider the two scan converted lines in Fig. 3.11. Line B, the diagonal line, has a slope of 1 and hence is $\sqrt{2}$ times as long as A, the horizontal line. Yet the same number of pixels (10) is drawn to represent each line. If the intensity of each pixel is I, then the intensity per unit length of line A is I, whereas for line B it is only $I/\sqrt{2}$; this discrepancy is easily detected by the viewer. On a bilevel display, there is no cure for this problem, but on an n-bits-per-pixel system we can compensate by setting the intensity to be a function of the line's slope. Antialiasing, discussed in Section 3.17, achieves an even better result by treating the line as a thin rectangle and computing

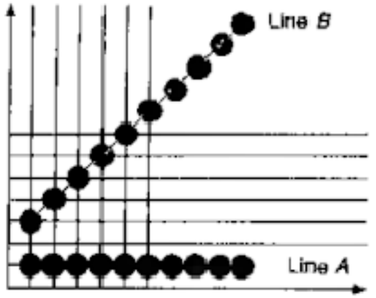


Fig. 3.11 Varying intensity of raster lines as a function of slope.

appropriate intensities for the multiple pixels in each column that lie in or near the rectangle.

Treating the line as a rectangle is also a way to create thick lines. In Section 3.9, we show how to modify the basic scan-conversion algorithms to deal with thick primitives and with primitives whose appearance is affected by line-style and pen-style attributes. Chapter 19 treats several other enhancements of the fundamental algorithms, such as handling endpoint shapes and creating joins between lines with multiple-pixel width.

Outline primitives composed of lines. Knowing how to scan convert lines, how do we scan convert primitives made from lines? Polylines can be scan-converted one line segment at a time. Scan converting rectangles and polygons as area-defining primitives could be done a line segment at a time but that would result in some pixels being drawn that lie outside a primitive's area—see Sections 3.5 and 3.6 for special algorithms to handle this problem. Care must be taken to draw shared vertices of polylines only once, since drawing a vertex twice causes it to change color or to be set to background when writing in xor mode to a screen, or to be written at double intensity on a film recorder. In fact, other pixels may be shared by two line segments that lie close together or cross as well. See Section 19.7 and Exercise 3.8 for a discussion of this, and of the difference between a polyline and a sequence of connected line segments.

3.3 SCAN CONVERTING CIRCLES

Although SRGP does not offer a circle primitive, the implementation will benefit from treating the circular ellipse arc as a special case because of its eight-fold symmetry, both for clipping and for scan conversion. The equation of a circle centered at the origin is $x^2 + y^2 = R^2$. Circles not centered at the origin may be translated to the origin by integer amounts and then scan converted, with pixels written with the appropriate offset. There are several easy but inefficient ways to scan convert a circle. Solving for y in the implicit circle equation, we get the explicit $y = f(x)$ as

$$y = \pm\sqrt{R^2 - x^2}.$$

To draw a quarter circle (the other quarters are drawn by symmetry), we can increment x from 0 to R in unit steps, solving for $+y$ at each step. This approach works, but it is

inefficient because of the multiply and square-root operations. Furthermore, the circle will have large gaps for values of x close to R, because the slope of the circle becomes infinite there (see Fig. 3.12). A similarly inefficient method, which does, however, avoid the large gaps, is to plot $(R\cos\theta, R\sin\theta)$ by stepping θ from 0° to 90°.

3.3.1 Eight-Way Symmetry

We can improve the drawing process of the previous section by taking greater advantage of the symmetry in a circle. Consider first a circle centered at the origin. If the point (x, y) is on the circle, then we can trivially compute seven other points on the circle, as shown in Fig. 3.13. Therefore, we need to compute only one 45° segment to determine the circle completely. For a circle centered at the origin, the eight symmetrical points can be displayed with procedure CirclePoints (the procedure is easily generalized to the case of circles with arbitrary origins):

```
void CirclePoints (int x, int y, int value)
{
    WritePixel (x, y, value);
    WritePixel (y, x, value);
    WritePixel (y, -x, value);
    WritePixel (x, -y, value);
    WritePixel (-x,  y, value);
    WritePixel (-y, -x, value);
    WritePixel (-y, x, value);
    WritePixel (-x, y, value);
} /* CirclePoints */
```

We do not want to call CirclePoints when $x = y$, because each of four pixels would be set twice; the code is easily modified to handle that boundary condition.

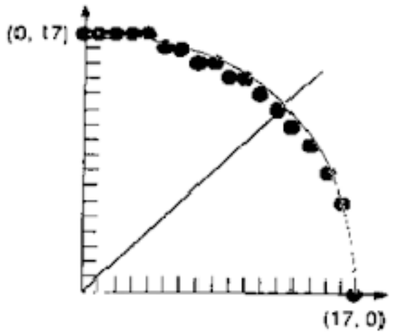


Fig. 3.12 A quarter circle generated with unit steps in x, and with y calculated and then rounded. Unique values of y for each x produce gaps.

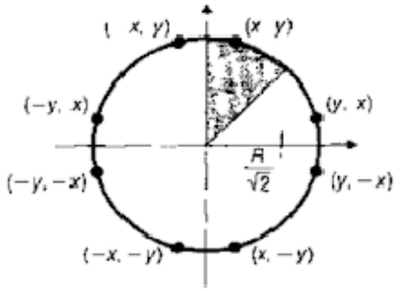


Fig. 3.13 Eight symmetrical points on a circle.

3.3.2 Midpoint Circle Algorithm

Bresenham [BRES77] developed an incremental circle generator that is more efficient than the methods we have discussed. Conceived for use with pen plotters, the algorithm generates all points on a circle centered at the origin by incrementing all the way around the circle. We derive a similar algorithm, again using the midpoint criterion, which, for the case of integer center point and radius, generates the same, optimal set of pixels. Furthermore, the resulting code is essentially the same as that specified in patent 4,371,933 [BRES83].

We consider only 45° of a circle, the second octant from $x = 0$ to $x = y = R/\sqrt{2}$, and use the CirclePoints procedure to display points on the entire circle. As with the midpoint line algorithm, the strategy is to select which of 2 pixels is closer to the circle by evaluating a function at the midpoint between the 2 pixels. In the second octant, if pixel P at (x_p, y_p) has been previously chosen as closest to the circle, the choice of the next pixel is between pixel E and SE (see Fig. 3.14).

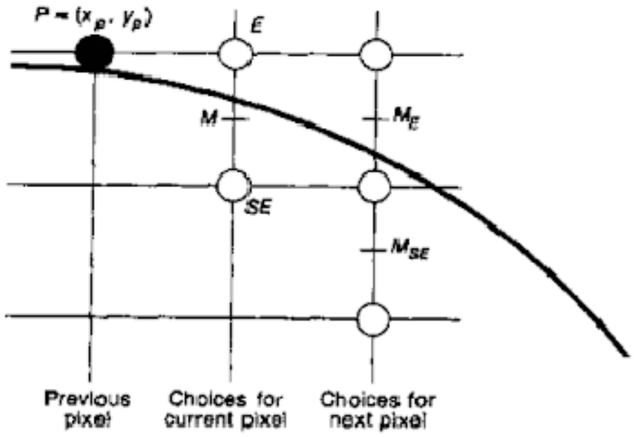


Fig. 3.14 The pixel grid for the midpoint circle algorithm showing M and the pixels E and SE to choose between.

Let $F(x, y) = x^2 + y^2 - R^2$; this function is 0 on the circle, positive outside the circle, and negative inside the circle. It can be shown that if the midpoint between the pixels E and SE is outside the circle, then pixel SE is closer to the circle. On the other hand, if the midpoint is inside the circle, pixel E is closer to the circle.

As for lines, we choose on the basis of the decision variable d, which is the value of the function at the midpoint,

$$d_{old} = F(x_P + 1, y_P - \tfrac{1}{2}) = (x_P + 1)^2 + (y_P - \tfrac{1}{2})^2 - R^2.$$

If $d_{old} < 0$, E is chosen, and the next midpoint will be one increment over in x. Then,

$$d_{new} = F(x_P + 2, y_P - \tfrac{1}{2}) = (x_P + 2)^2 + (y_P - \tfrac{1}{2})^2 - R^2,$$

and $d_{new} = d_{old} + (2x_P + 3)$; therefore, the increment $\Delta_E = 2x_P + 3$.

If $d_{old} \geq 0$, SE is chosen,[6] and the next midpoint will be one increment over in x and one increment down in y. Then

$$d_{new} = F(x_P + 2, y_P - \tfrac{3}{2}) = (x_P + 2)^2 + (y_P - \tfrac{3}{2})^2 - R^2.$$

Since $d_{new} = d_{old} + (2x_P - 2y_P + 5)$, the increment $\Delta_{SE} = 2x_P - 2y_P + 5$.

Recall that, in the linear case, Δ_E and Δ_{NE} were constants; in the quadratic case, however, Δ_E and Δ_{SE} vary at each step and are functions of the particular values of x_P and y_P at the pixel chosen in the previous iteration. Because these functions are expressed in terms of (x_P, y_P), we call P the *point of evaluation*. The Δ functions can be evaluated directly at each step by plugging in the values of x and y for the pixel chosen in the previous iteration. This direct evaluation is not expensive computationally, since the functions are only linear.

In summary, we do the same two steps at each iteration of the algorithm as we did for the line: (1) choose the pixel based on the sign of the variable d computed during the previous iteration, and (2) update the decision variable d with the Δ that corresponds to the choice of pixel. The only difference from the line algorithm is that, in updating d, we evaluate a linear function of the point of evaluation.

All that remains now is to compute the initial condition. By limiting the algorithm to integer radii in the second octant, we know that the starting pixel lies on the circle at $(0, R)$. The next midpoint lies at $(1, R - \frac{1}{2})$, therefore, and $F(1, R - \frac{1}{2}) = 1 + (R^2 - R + \frac{1}{4}) - R^2 = \frac{5}{4} - R$. Now we can implement the algorithm directly, as in Fig. 3.15. Notice how similar in structure this algorithm is to the line algorithm.

The problem with this version is that we are forced to do real arithmetic because of the fractional initialization of d. Although the procedure can be easily modified to handle circles that are not located on integer centers or do not have integer radii, we would like a more efficient, purely integer version. We thus do a simple program transformation to eliminate fractions.

First, we define a new decision variable, h, by $h = d - \frac{1}{4}$, and we substitute $h + \frac{1}{4}$ for d in the code. Now, the initialization is $h = 1 - R$, and the comparison $d < 0$ becomes $h < -\frac{1}{4}$.

[6] Choosing SE when $d = 0$ differs from our choice in the line algorithm and is arbitrary. The reader may wish to simulate the algorithm by hand to see that, for R = 17, 1 pixel is changed by this choice.


```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    double d = 5.0 / 4.0 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)              /* Select E */
            d += 2.0 * x + 3.0;
        else {                  /* Select SE */
            d += 2.0 * (x - y) + 5.0;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.15 The midpoint circle scan-conversion algorithm.

However, since h starts out with an integer value and is incremented by integer values (Δ_E and Δ_{SE}), we can change the comparison to just $h < 0$. We now have an integer algorithm in terms of h; for consistency with the line algorithm, we will substitute d for h throughout. The final, fully integer algorithm is shown in Fig. 3.16.

Figure 3.17 shows the second octant of a circle of radius 17 generated with the algorithm, and the first octant generated by symmetry (compare the results to Fig. 3.12).

Second-order differences. We can improve the performance of the midpoint circle algorithm by using the incremental computation technique even more extensively. We noted that the Δ functions are linear equations, and we computed them directly. Any polynomial can be computed incrementally, however, as we did with the decision variables for both the line and the circle. In effect, we are calculating *first-* and *second-order partial differences*, a useful technique that we encounter again in Chapters 11 and 19. The strategy is to evaluate the function directly at two adjacent points, to calculate the difference (which, for polynomials, is always a polynomial of lower degree), and to apply that difference in each iteration.

If we choose E in the current iteration, the point of evaluation moves from (x_P, y_P) to $(x_P + 1, y_P)$. As we saw, the first-order difference is $\Delta_{E_{old}}$ at $(x_P, y_P) = 2x_P + 3$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_P + 1, y_P) = 2(x_P + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$.

```
void MidpointCircle (int radius, int value)
/* Assumes center of circle is at origin. Integer arithmetic only */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0)          /* Select E */
            d += 2 * x + 3;
        else {              /* Select SE */
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.16 The integer midpoint circle scan-conversion algorithm.

Similarly, $\Delta_{SE_{old}}$ at $(x_p, y_p) = 2x_p - 2y_p + 5$. Therefore,

$$\Delta_{SE_{new}} \text{ at } (x_p + 1, y_p) = 2(x_p + 1) - 2y_p + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 2$.

If we choose SE in the current iteration, the point of evaluation moves from (x_p, y_p) to $(x_p + 1, y_p - 1)$. Therefore,

$$\Delta_{E_{new}} \text{ at } (x_p + 1, y_p - 1) = 2(x_p + 1) + 3,$$

and the second-order difference is $\Delta_{E_{new}} - \Delta_{E_{old}} = 2$. Also,

$$\Delta_{SE_{new}} \text{ at } (x_p + 1, y_p - 1) = 2(x_p + 1) - 2(y_p - 1) + 5,$$

and the second-order difference is $\Delta_{SE_{new}} - \Delta_{SE_{old}} = 4$.

The revised algorithm then consists of the following steps: (1) choose the pixel based on the sign of the variable d computed during the previous iteration; (2) update the decision variable d with either Δ_E or Δ_{SE}, using the value of the corresponding Δ computed during the previous iteration; (3) update the Δs to take into account the move to the new pixel, using the constant differences computed previously; and (4) do the move. Δ_E and Δ_{SE} are initialized using the start pixel $(0, R)$. The revised procedure using this technique is shown in Fig. 3.18.

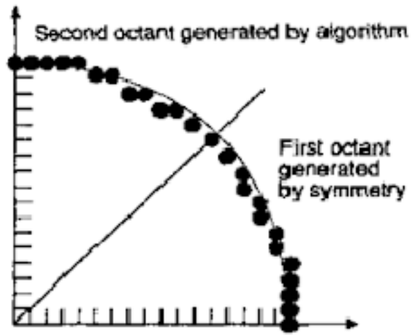


Fig. 3.17 Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;
    int deltaE = 3;
    int deltaSE = -2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {            /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;       /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        CirclePoints (x, y, value);
    } /* while */
} /* MidpointCircle */
```

Fig. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

3.4 SCAN CONVERTING ELLIPSES

Consider the standard ellipse of Fig. 3.19, centered at (0, 0). It is described by the equation

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0,$$

where $2a$ is the length of the major axis along the x axis, and $2b$ is the length of the minor axis along the y axis. The midpoint technique discussed for lines and circles can also be applied to the more general conics. In this chapter, we consider the standard ellipse that is supported by SRGP; in Chapter 19, we deal with ellipses at any angle. Again, to simplify the algorithm, we draw only the arc of the ellipse that lies in the first quadrant, since the other three quadrants can be drawn by symmetry. Note also that standard ellipses centered at integer points other than the origin can be drawn using a simple translation. The algorithm presented here is based on Da Silva's algorithm, which combines the techniques used by Pitteway [PITT67], Van Aken [VANA84] and Kappel [KAPP85] with the use of partial differences [DASI89].

We first divide the quadrant into two regions; the boundary between the two regions is the point at which the curve has a slope of -1 (see Fig. 3.20).

Determining this point is more complex than it was for circles, however. The vector that is perpendicular to the tangent to the curve at point P is called the *gradient*, defined as

$$\text{grad } F(x, y) = \partial F/\partial x\ \mathbf{i} + \partial F/\partial y\ \mathbf{j} = 2b^2x\ \mathbf{i} + 2a^2y\ \mathbf{j}.$$

The boundary between the two regions is the point at which the slope of the curve is -1, and that point occurs when the gradient vector has a slope of 1—that is, when the **i** and **j** components of the gradient are of equal magnitude. The **j** component of the gradient is larger than the **i** component in region 1, and vice versa in region 2. Thus, if at the next midpoint, $a^2(y_P - \frac{1}{2}) \leq b^2(x_P + 1)$, we switch from region 1 to region 2.

As with any midpoint algorithm, we evaluate the function at the midpoint between two pixels and use the sign to determine whether the midpoint lies inside or outside the ellipse and, hence, which pixel lies closer to the ellipse. Therefore, in region 1, if the current pixel is located at (x_P, y_P), then the decision variable for region 1, d_1, is $F(x, y)$ evaluated at $(x_P + 1, y_P - \frac{1}{2})$, the midpoint between E and SE. We now repeat the process we used for deriving the

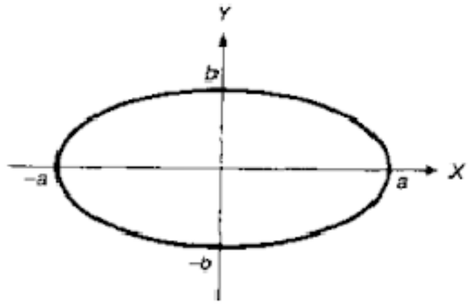


Fig. 3.19 Standard ellipse centered at the origin.

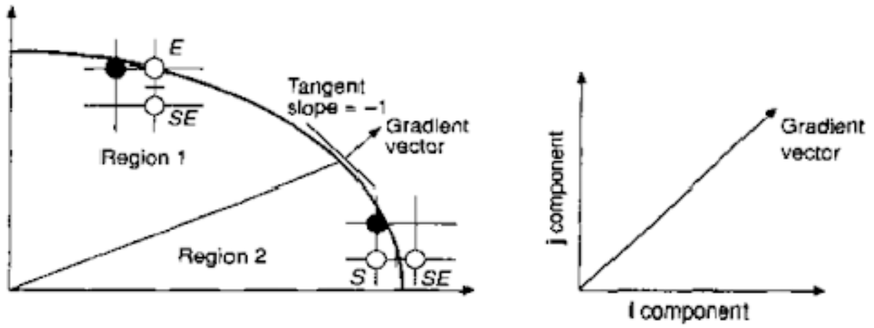


Fig. 3.20 Two regions of the ellipse defined by the 45° tangent.

two Δs for the circle. For a move to E, the next midpoint is one increment over in x. Then,

$$d_{old} = F(x_P + 1, y_P - \tfrac{1}{2}) = b^2(x_P + 1)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2,$$

$$d_{new} = F(x_P + 2, y_P - \tfrac{1}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{1}{2})^2 - a^2b^2.$$

Since $d_{new} = d_{old} + b^2(2x_P + 3)$, the increment $\Delta_E = b^2(2x_P + 3)$.

For a move to SE, the next midpoint is one increment over in x and one increment down in y. Then,

$$d_{new} = F(x_P + 2, y_P - \tfrac{3}{2}) = b^2(x_P + 2)^2 + a^2(y_P - \tfrac{3}{2})^2 - a^2b^2.$$

Since $d_{new} = d_{old} + b^2(2x_P + 3) + a^2(-2y_P + 2)$, the increment $\Delta_{SE} = b^2(2x_P + 3) + a^2(-2y_P + 2)$.

In region 2, if the current pixel is at (x_P, y_P), the decision variable d_2 is $F(x_P + \frac{1}{2}, y_P - 1)$, the midpoint between S and SE. Computations similar to those given for region 1 may be done for region 2.

We must also compute the initial condition. Assuming integer values a and b, the ellipse starts at $(0, b)$, and the first midpoint to be calculated is at $(1, b - \frac{1}{2})$. Then,

$$F(1, b - \tfrac{1}{2}) = b^2 + a^2(b - \tfrac{1}{2})^2 - a^2b^2 = b^2 + a^2(-b + \tfrac{1}{4}).$$

At every iteration in region 1, we must not only test the decision variable d_1 and update the Δ functions, but also see whether we should switch regions by evaluating the gradient at the midpoint between E and SE. When the midpoint crosses over into region 2, we change our choice of the 2 pixels to compare from E and SE to SE and S. At the same time, we have to initialize the decision variable d_2 for region 2 to the midpoint between SE and S. That is, if the last pixel chosen in region 1 is located at (x_P, y_P), then the decision variable d_2 is initialized at $(x_P + \frac{1}{2}, y_P - 1)$. We stop drawing pixels in region 2 when the y value of the pixel is equal to 0.

As with the circle algorithm, we can either calculate the Δ functions directly in each iteration of the loop or compute them with differences. Da Silva shows that computation of second-order partials done for the Δs can, in fact, be used for the gradient as well [DASI89]. He also treats general ellipses that have been rotated and the many tricky

boundary conditions for very thin ellipses. The pseudocode algorithm of Fig. 3.21 uses th simpler direct evaluation rather than the more efficient formulation using second-orde differences; it also skips various tests (see Exercise 3.9). In the case of integer a and b, w can eliminate the fractions via program transformations and use only integer arithmetic.

```
void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
    double d2;

    int x = 0;
    int y = b;
    double d1 = b^2 - (a^2 b) + (0.25 a^2);
    EllipsePoints (x, y, value);              /* The 4-way symmetrical WritePixel */

    /* Test gradient if still in region 1 */
    while ( a^2 (y - 0.5) > b^2 (x + 1) ) {   /* Region 1 */
        if (d1 < 0)                           /* Select E */
            d1 += b^2 (2x + 3);
        else {                                /* Select SE */
            d1 += b^2 (2x + 3) + a^2 (-2y + 2);
            y--;
        }
        x++;
        EllipsePoints (x, y, value);
    }  /* Region 1 */

    d2 = b^2 (x + 0.5)^2 + a^2 (y - 1)^2 - a^2 b^2;
    while (y > 0) {                           /* Region 2 */
        if (d2 < 0) {                         /* Select SE */
            d2 += b^2 (2x + 2) + a^2 (-2y + 3);
            x++;
        } else
            d2 += a^2 (-2y + 3);              /* Select S */
        y--;
        EllipsePoints (x, y, value);
    }  /* Region 2 */
}  /* MidpointEllipse */
```

Fig. 3.21 Pseudocode for midpoint ellipse scan-conversion algorithm.

3.11 CLIPPING IN A RASTER WORLD

As we noted in the introduction to this chapter, it is essential that both clipping and scan conversion be done as rapidly as possible, in order to provide the user with quick updates resulting from changes to the application model. Clipping can be done analytically, on the fly during scan conversion, or as part of a copyPixel with the desired clip rectangle from a canvas storing unclipped primitives to the destination canvas. This third technique would be useful in situations where a large canvas can be generated ahead of time, and the user can then examine pieces of it for a significant period of time by panning the clip rectangle, without updating the contents of the canvas.

Combining clipping and scan conversion, sometimes called *scissoring*, is easy to do for filled or thick primitives as part of span arithmetic: Only the extrema need to be clipped, and no interior pixels need be examined. Scissoring shows yet another advantage of span coherence. Also, if an outline primitive is not much larger than the clip rectangle, not many pixels, relatively speaking, will fall outside the clip region. For such a case, it may well be faster to generate each pixel and to clip it (i.e., to write it conditionally) than to do analytical clipping beforehand. In particular, although the bounds test is in the inner loop, the expensive memory write is avoided for exterior pixels, and both the incremental computation and the testing may run entirely in a fast memory, such as a CPU instruction cache or a display controller's microcode memory.

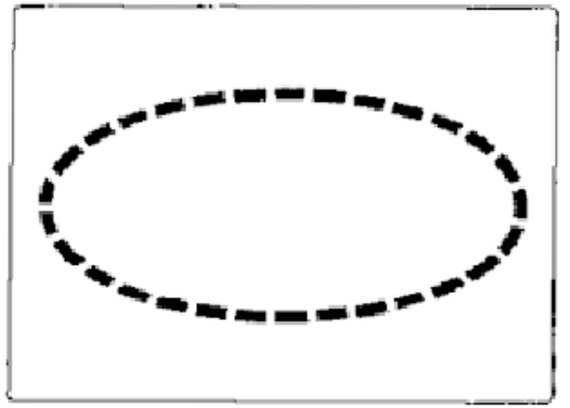


Fig. 3.37 Combining pen pattern and line style.

Other tricks may be useful. For example, one may "home in" on the intersection of a line with a clip edge by doing the standard midpoint scan-conversion algorithm on every *i*th pixel and testing the chosen pixel against the rectangle bounds until the first pixel that lies inside the region is encountered. Then the algorithm has to back up, find the first pixel inside, and to do the normal scan conversion thereafter. The last interior pixel could be similarly determined, or each pixel could be tested as part of the scan-conversion loop and scan conversion stopped the first time the test failed. Testing every eighth pixel works well, since it is a good compromise between having too many tests and too many pixels to back up (see Exercise 3.26).

For graphics packages that operate in floating point, it is best to clip analytically in the floating-point coordinate system and then to scan convert the clipped primitives, being careful to initialize decision variables correctly, as we did for lines in Section 3.2.3. For integer graphics packages such as SRGP, there is a choice between preclipping and then scan converting or doing clipping during scan conversion. Since it is relatively easy to do analytical clipping for lines and polygons, clipping of those primitives is often done before scan conversion, while it is faster to clip other primitives during scan conversion. Also, it is quite common for a floating-point graphics package to do analytical clipping in its coordinate system and then to call lower-level scan-conversion software that actually generates the clipped primitives; this integer graphics software could then do an additional raster clip to rectangular (or even arbitrary) window boundaries. Because analytic clipping of primitives is both useful for integer graphics packages and essential for 2D and 3D floating-point graphics packages, we discuss the basic analytical clipping algorithms in this chapter.

3.12 CLIPPING LINES

This section treats analytical clipping of lines against rectangles;[10] algorithms for clipping other primitives are handled in subsequent sections. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that SRGP primitives built out of lines (i.e., polylines, unfilled rectangles, and polygons) can be clipped by repeated application of the line clipper. Furthermore, circles and ellipses may be piecewise-linearly approximated with a sequence of very short lines, so that boundaries can be treated as a single polyline or polygon for both clipping and scan conversion. Conics are represented in some systems as ratios of parametric polynomials (see Chapter 11), a representation that also lends itself readily to an incremental, piecewise linear approximation suitable for a line-clipping algorithm. Clipping a rectangle against a rectangle results in at most a single rectangle. Clipping a convex polygon against a rectangle results in at most a single convex polygon, but clipping a concave polygon may produce more than one concave polygon. Clipping a circle or ellipse against a rectangle results in as many as four arcs.

Lines intersecting a rectangular clip region (or any convex polygon) are always clipped to a single line segment; lines lying on the clip rectangle's border are considered inside and hence are displayed. Figure 3.38 shows several examples of clipped lines.

[10] This chapter does not cover clipping primitives to multiple rectangles (as when windows overlap in a windowing system) or to nonrectangular regions; the latter topic is discussed briefly in Section 19.7.

Fig. 3.38 Cases for clipping lines.

3.12.1 Clipping Endpoints

Before we discuss clipping lines, let's look at the simpler problem of clipping individual points. If the x coordinate boundaries of the clip rectangle are at x_{min} and x_{max}, and the y coordinate boundaries are at y_{min} and y_{max}, then four inequalities must be satisfied for a point at (x, y) to be inside the clip rectangle:

$$x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}.$$

If any of the four inequalities does not hold, the point is outside the clip rectangle.

3.12.2 Clipping Lines by Solving Simultaneous Equations

To clip a line, we need to consider only its endpoints, not its infinitely many interior points. If both endpoints of a line lie inside the clip rectangle (e.g., AB in Fig. 3.38), the entire line lies inside the clip rectangle and can be *trivially accepted*. If one endpoint lies inside and one outside (e.g., CD in the figure), the line intersects the clip rectangle and we must compute the intersection point. If both endpoints are outside the clip rectangle, the line may (or may not) intersect with the clip rectangle (EF, GH, and IJ in the figure), and we need to perform further calculations to determine whether there are any intersections, and if there are, where they occur.

The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those edges; if so, the line cuts the clip rectangle and is partially inside. For each line and clip-rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them. Next, we test whether this intersection point is "interior"—that is, whether it lies within both the clip rectangle edge and the line; if so, there is an intersection with the clip rectangle. In Fig. 3.38, intersection points G' and H' are interior, but I' and J' are not.

When we use this approach, we must solve two simultaneous equations using multiplication and division for each <edge, line> pair. Although the slope-intercept

formula for lines learned in analytic geometry could be used, it describes infinite lines, whereas in graphics and clipping we deal with finite lines (called *line segments* in mathematics). In addition, the slope-intercept formula does not deal with vertical lines—a serious problem, given our upright clip rectangle. A parametric formulation for line segments solves both problems:

$$x = x_0 + t(x_1 - x_0), \; y = y_0 + t(y_1 - y_0).$$

These equations describe (x, y) on the directed line segment from (x_0, y_0) to (x_1, y_1) for the parameter t in the range $[0, 1]$, as simple substitution for t confirms. Two sets of simultaneous equations of this parametric form can be solved for parameters t_{edge} for the edge and t_{line} for the line segment. The values of t_{edge} and t_{line} can then be checked to see whether both lie in $[0, 1]$; if they do, the intersection point lies within both segments and is a true clip-rectangle intersection. Furthermore, the special case of a line parallel to a clip-rectangle edge must also be tested before the simultaneous equations can be solved. Altogether, the brute-force approach involves considerable calculation and testing; it is thus inefficient.

3.12.3 The Cohen–Sutherland Line-Clipping Algorithm

The more efficient Cohen–Sutherland algorithm performs initial tests on a line to determine whether intersection calculations can be avoided. First, endpoint pairs are checked for trivial acceptance. If the line cannot be trivially accepted, *region checks* are done. For instance, two simple comparisons on x show that both endpoints of line *EF* in Fig. 3.38 have an x coordinate less than x_{min} and thus lie in the region to the left of the clip rectangle (i.e., in the outside halfplane defined by the left edge); therefore, line segment *EF* can be *trivially rejected* and needs to be neither clipped nor displayed. Similarly, we can trivially reject lines with both endpoints in regions to the right of x_{max}, below y_{min}, and above y_{max}.

If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected. Thus, a segment is iteratively clipped by testing for trivial acceptance or rejection, and is then subdivided if neither test is successful, until what remains is completely inside the clip rectangle or can be trivially rejected. The algorithm is particularly efficient for two common cases. In the first case of a large clip rectangle enclosing all or most of the display area, most primitives can be trivially accepted. In the second case of a small clip rectangle, almost all primitives can be trivially rejected. This latter case arises in a standard method of doing pick correlation in which a small rectangle surrounding the cursor, called the *pick window*, is used to clip primitives to determine which primitives lie within a small (rectangular) neighborhood of the cursor's *pick point* (see Section 7.12.2).

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions (see Fig. 3.39). Each region is assigned a 4-bit code, determined by where the region lies with respect to the outside halfplanes of the clip-rectangle edges. Each bit in the outcode is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

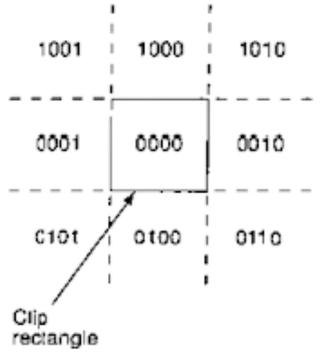


Fig. 3.39 Region outcodes.

First bit	outside halfplane of top edge, above top edge	$y > y_{max}$
Second bit	outside halfplane of bottom edge, below bottom edge	$y < y_{min}$
Third bit	outside halfplane of right edge, to the right of right edge	$x > x_{max}$
Fourth bit	outside halfplane of left edge, to the left of left edge	$x < x_{min}$

Since the region lying above and to the left of the clip rectangle, for example, lies in the outside halfplane of the top and left edges, it is assigned a code of 1001. A particularly efficient way to calculate the outcode derives from the observation that bit 1 is the sign bit of $(y_{max} - y)$; bit 2 is that of $(y - y_{min})$; bit 3 is that of $(x_{max} - x)$; and bit 4 is that of $(x - x_{min})$. Each endpoint of the line segment is then assigned the code of the region in which it lies. We can now use these endpoint codes to determine whether the line segment lies completely inside the clip rectangle or in the outside halfplane of an edge. If both 4-bit codes of the endpoints are zero, then the line lies completely inside the clip rectangle. However, if both endpoints lie in the outside halfplane of a particular edge, as for *EF* in Fig. 3.38, the codes for both endpoints each have the bit set showing that the point lies in the outside halfplane of that edge. For *EF*, the outcodes are 0001 and 1001, respectively, showing with the fourth bit that the line segment lies in the outside halfplane of the left edge. Therefore, if the logical **and** of the codes of the endpoints is not zero, the line can be trivially rejected.

If a line cannot be trivially accepted or rejected, we must subdivide it into two segments such that one or both segments can be discarded. We accomplish this subdivision by using an edge that the line crosses to cut the line into two segments: The section lying in the outside halfplane of the edge is thrown away. We can choose any order in which to test edges, but we must, of course, use the same order each time in the algorithm; we shall use the top-to-bottom, right-to-left order of the outcode. A key property of the outcode is that bits that are set in a nonzero outcode correspond to edges crossed: If one endpoint lies in the outside halfplane of an edge and the line segment fails the trivial-rejection tests, then the other point must lie on the inside halfplane of that edge and the line segment must cross it. Thus, the algorithm always chooses a point that lies outside and then uses an outcode bit that *is* set to determine a clip edge; the edge chosen is the first in the top-to-bottom, right-to-left order—that is, it is the leftmost bit that is set in the outcode.

The algorithm works as follows. We compute the outcodes of both endpoints and check

for trivial acceptance and rejection. If neither test is successful, we find an endpoint that lies outside (at least one will), and then test the outcode to find the edge that is crossed and to determine the corresponding intersection point. We can then clip off the line segment from the outside endpoint to the intersection point by replacing the outside endpoint with the intersection point, and compute the outcode of this new endpoint to prepare for the next iteration.

For example, consider the line segment AD in Fig. 3.40. Point A has outcode 0000 and point D has outcode 1001. The line can be neither trivially accepted or rejected. Therefore, the algorithm chooses D as the outside point, whose outcode shows that the line crosses the top edge and the left edge. By our testing order, we first use the top edge to clip AD to AB, and we compute B's outcode as 0000. In the next iteration, we apply the trivial acceptance/rejection tests to AB, and it is trivially accepted and displayed.

Line EI requires multiple iterations. The first endpoint, E, has an outcode of 0100, so the algorithm chooses it as the outside point and tests the outcode to find that the first edge against which the line is cut is the bottom edge, where EI is clipped to FI. In the second iteration, FI cannot be trivially accepted or rejected. The outcode of the first endpoint, F, is 0000, so the algorithm chooses the outside point I that has outcode 1010. The first edge clipped against is therefore the top edge, yielding FH. H's outcode is determined to be 0010, so the third iteration results in a clip against the right edge to FG. This is trivially accepted in the fourth and final iteration and displayed. A different sequence of clips would have resulted if we had picked I as the initial point: On the basis of its outcode, we would have clipped against the top edge first, then the right edge, and finally the bottom edge.

In the code of Fig. 3.41, we use constant integers and bitwise arithmetic to represent the outcodes, because this representation is more natural than an array with an entry for each outcode. We use an internal procedure to calculate the outcode for modularity; to improve performance, we would, of course, put this code in line.

We can improve the efficiency of the algorithm slightly by not recalculating slopes (see Exercise 3.28). Even with this improvement, however, the algorithm is not the most efficient one. Because testing and clipping are done in a fixed order, the algorithm will sometimes perform needless clipping. Such clipping occurs when the intersection with a rectangle edge is an "external intersection"; that is, when it does not lie on the clip-rectangle boundary (e.g., point H on line EI in Fig. 3.40). The Nicholl, Lee, and

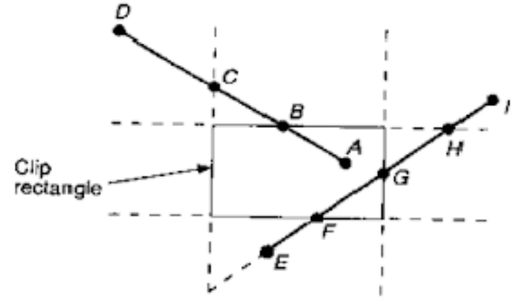


Fig. 3.40 Illustration of Cohen–Sutherland line clipping.


```
typedef unsigned int outcode;
enum {TOP = 0x1, BOTTOM = 0x2, RIGHT = 0x4, LEFT = 0x8};

void CohenSutherlandLineClipAndDraw (
    double x0, double y0, double x1, double y1, double xmin, double xmax,
    double ymin, double ymax, int value)
/* Cohen-Sutherland clipping algorithm for line P0 = (x0, y0) to P1 = (x1, y1) and */
/* clip rectangle with diagonal from (xmin, ymin) to (xmax, ymax) */
{
    /* Outcodes for P0, P1, and whatever point lies outside the clip rectangle */
    outcode outcode0, outcode1, outcodeOut;
    boolean accept = FALSE, done = FALSE;
    outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
    outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
    do {
        if (!(outcode0 | outcode1)) {                    /* Trivial accept and exit */
            accept = TRUE; done = TRUE;
        } else if (outcode0 & outcode1)                  /* Logical and is true, so trivial reject and exit */
            done = TRUE;
        else {
            /* Failed both tests, so calculate the line segment to clip: */
            /* from an outside point to an intersection with clip edge. */
            double x, y;
            /* At least one endpoint is outside the clip rectangle; pick it. */
            outcodeOut = outcode0 ? outcode0 : outcode1;
            /* Now find intersection point. */
            /* use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope) * (y - y0). */
            if (outcodeOut & TOP) {                      /* Divide line at top of clip rect */
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax;
            } else if (outcodeOut & BOTTOM) {            /* Divide line at bottom edge of clip rect */
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin;
            } else if (outcodeOut & RIGHT) {             /* Divide line at right edge of clip rect */
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax;
            } else {                                     /* Divide line at left edge of clip rect */
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin;
            }
            /* Now we move outside point to intersection point to clip, */
            /* and get ready for next pass. */
            if (outcodeOut == outcode0) {
                x0 = x; y0 = y; outcode0 = CompOutCode (x0, y0, xmin, xmax, ymin, ymax);
            } else {
                x1 = x; y1 = y; outcode1 = CompOutCode (x1, y1, xmin, xmax, ymin, ymax);
            }
        }  /* Subdivide */
    } while (done == FALSE);
```

Fig. 3.41 *(Cont.).*

```
    if (accept)
        MidpointLineReal (x0, y0, x1, y1, value);    /* Version for double coordinates */
}   /* CohenSutherlandLineClipAndDraw */

outcode CompOutCode (
    double x, double y, double xmin, double xmax, double ymin, double ymax);
{
    outcode code = 0;
    if (y > ymax)
        code |= TOP;
    else if (y < ymin)
        code |= BOTTOM;
    if (x > xmax)
        code |= RIGHT;
    else if (x < xmin)
        code |= LEFT;
    return code;
}   /* CompOutCode */
```

Fig. 3.41 Cohen–Sutherland line-clipping algorithm.

Nicholl [NICH87] algorithm, by contrast, avoids calculating external intersections by subdividing the plane into many more regions; it is discussed in Chapter 19. An advantage of the much simpler Cohen–Sutherland algorithm is that its extension to a 3D orthographic view volume is straightforward, as seen in Section 6.5.3

3.12.4 A Parametric Line-Clipping Algorithm

The Cohen–Sutherland algorithm is probably still the most commonly used line-clipping algorithm because it has been around longest and has been published widely. In 1978, Cyrus and Beck published an algorithm that takes a fundamentally different and generally more efficient approach to line clipping [CYRU78]. The Cyrus–Beck technique can be used to clip a 2D line against a rectangle or an arbitrary convex polygon in the plane, or a 3D line against an arbitrary convex polyhedron in 3D space. Liang and Barsky later independently developed a more efficient parametric line-clipping algorithm that is especially fast in the special cases of upright 2D and 3D clip regions [LIAN84]. In addition to taking advantage of these simple clip boundaries, they introduced more efficient trivial rejection tests that work for general clip regions. Here we follow the original Cyrus–Beck development to introduce parametric clipping. Since we are concerned only with upright clip rectangles, however, we reduce the Cyrus–Beck formulation to the more efficient Liang–Barsky case at the end of the development.

Recall that the Cohen–Sutherland algorithm, for lines that cannot be trivially accepted or rejected, calculates the (x, y) intersection of a line segment with a clip edge by substituting the known value of x or y for the vertical or horizontal clip edge, respectively. The parametric line algorithm, however, finds the value of the parameter t in the parametric

representation of the line segment for the point at which that segment intersects the infinite line on which the clip edge lies. Because all clip edges are in general intersected by the line, four values of t are calculated. A series of simple comparisons is used to determine which (if any) of the four values of t correspond to actual intersections. Only then are the (x, y) values for one or two actual intersections calculated. In general, this approach saves time over the Cohen–Sutherland intersection-calculation algorithm because it avoids the repetitive looping needed to clip to multiple clip-rectangle edges. Also, calculations in 1D parameter space are simpler than those in 3D coordinate space. Liang and Barsky improve on Cyrus–Beck by examining each t-value as it is generated, to reject some line segments before all four t-values have been computed.

The Cyrus–Beck algorithm is based on the following formulation of the intersection between two lines. Figure 3.42 shows a single edge E_i of the clip rectangle and that edge's outward normal N_i (i.e., outward to the clip rectangle[11]), as well as the line segment from P_0 to P_1 that must be clipped to the edge. Either the edge or the line segment may have to be extended to find the intersection point

As before, this line is represented parametrically as

$$P(t) = P_0 + (P_1 - P_0)t,$$

where $t = 0$ at P_0 and $t = 1$ at P_1. Now, pick an arbitrary point P_{E_i} on edge E_i and consider the three vectors $P(t) - P_{E_i}$ from P_{E_i} to three designated points on the line from P_0 to P_1: the intersection point to be determined, an endpoint of the line on the inside halfplane of the edge, and an endpoint on the line in the outside halfplane of the edge. We can distinguish in which region a point lies by looking at the value of the dot product $N_i \cdot [P(t) - P_{E_i}]$. This value is negative for a point in the inside halfplane, zero for a point on the line containing the edge, and positive for a point that lies in the outside halfplane. The definitions of inside and outside halfplanes of an edge correspond to a counterclockwise enumeration of the edges of the clip region, a convention we shall use throughout this book. Now we can solve for the value of t at the intersection of P_0P_1 with the edge:

$$N_i \cdot [P(t) - P_{E_i}] = 0.$$

First, substitute for $P(t)$:

$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{E_i}] = 0.$$

Next, group terms and distribute the dot product:

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot [P_1 - P_0]t = 0.$$

Let $D = (P_1 - P_0)$ be the vector from P_0 to P_1, and solve for t:

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}, \tag{3.1}$$

Note that this gives a valid value of t only if the denominator of the expression is nonzero.

[11] Cyrus and Beck use inward normals, but we prefer to use outward normals for consistency with plane normals in 3D, which are outward. Our formulation therefore differs only in the testing of a sign.

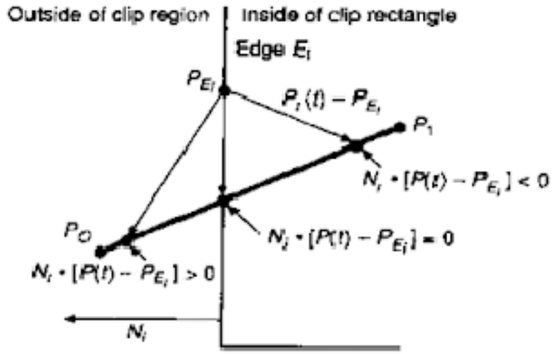


Fig. 3.42 Dot products for three points outside, inside, and on the boundary of the clip region.

For this to be true, the algorithm checks that

$N_i \neq 0$ (that is, the normal should not be 0; this could occur only as a mistake),

$D \neq 0$ (that is, $P_1 \neq P_0$),

$N_i \cdot D \neq 0$ (that is, the edge E_i and the line from P_0 to P_1 are not parallel. If they were parallel, there can be no single intersection for this edge, so the algorithm moves on to the next case.).

Equation (3.1) can be used to find the intersections between P_0P_1 and each edge of the clip rectangle. We do this calculation by determining the normal and an arbitrary P_{E_i}—say, an endpoint of the edge—for each clip edge, then using these values for all line segments. Given the four values of t for a line segment, the next step is to determine which (if any) of the values correspond to internal intersections of the line segment with edges of the clip rectangle. As a first step, any value of t outside the interval [0, 1] can be discarded, since it lies outside P_0P_1. Next, we need to determine whether the intersection lies on the clip boundary.

We could try simply sorting the remaining values of t, choosing the intermediate values of t for intersection points, as suggested in Fig. 3.43 for the case of line 1. But how do we distinguish this case from that of line 2, in which no portion of the line segment lies in the clip rectangle and the intermediate values of t correspond to points not on the clip boundary? Also, which of the four intersections of line 3 are the ones on the clip boundary?

The intersections in Fig. 3.43 are characterized as "potentially entering" (PE) or "potentially leaving" (PL) the clip rectangle, as follows: If moving from P_0 to P_1 causes us to cross a particular edge to enter the edge's inside halfplane, the intersection is PE; if it causes us to leave the edge's inside halfplane, it is PL. Notice that, with this distinction, two interior intersection points of a line intersecting the clip rectangle have opposing labels.

Formally, intersections can be classified as PE or PL on the basis of the angle between P_0P_1 and N_i: If the angle is less than 90°, the intersection is PL; if it is greater than 90°, it is PE. This information is contained in the sign of the dot product of N_i and P_0P_1:

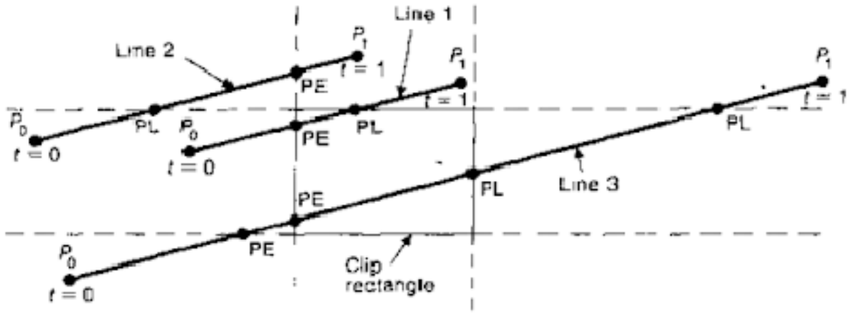


Fig. 3.43 Lines lying diagonal to the clip rectangle

$N_i \cdot D < 0 \Rightarrow$ PE (angle greater than 90),
$N_i \cdot D > 0 \Rightarrow$ PL (angle less than 90).

Notice that $N_i \cdot D$ is merely the denominator of Eq. (3.1), which means that, in the process of calculating t, the intersection can be trivially categorized.

With this categorization, line 3 in Fig. 3.43 suggests the final step in the process. We must choose a (PE, PL) pair that defines the clipped line. The portion of the infinite line through P_0P_1 that is within the clipping region is bounded by the PE intersection with the largest t value, which we call t_E, and the PL intersection with the smallest t value, t_L. The intersecting line segment is then defined by the range (t_E, t_L). But because we are interested in intersecting P_0P_1, not the infinite line, the definition of the range must be further modified so that $t = 0$ is a lower bound for t_E and $t = 1$ is an upper bound for t_L. What if $t_E > t_L$? This is exactly the case for line 2. It means that no portion of P_0P_1 is within the clip rectangle, and the entire line is rejected. Values of t_E and t_L that correspond to actual intersections are used to calculate the corresponding x and y coordinates.

The completed algorithm for upright clip rectangles is pseudocoded in Fig. 3.44. Table 3.1 shows for each edge the values of N_i, a canonical point on the edge, P_{E_i}, the vector $P_0 - P_{E_i}$ and the parameter t. Interestingly enough, because one coordinate of each normal is 0, we do not need to pin down the corresponding coordinate of P_{E_i} (denoted by an indeterminate x or y). Indeed, because the clip edges are horizontal and vertical, many simplifications apply that have natural interpretations. Thus we see from the table that the numerator, the dot product $N_i \cdot (P_0 - P_{E_i})$ determining whether the endpoint P_0 lies inside or outside a specified edge, reduces to the directed horizontal or vertical distance from the point to the edge. This is exactly the same quantity computed for the corresponding component of the Cohen-Sutherland outcode. The denominator dot product $N_i \cdot D$, which determines whether the intersection is potentially entering or leaving, reduces to $\pm dx$ or dy: if dx is positive, the line moves from left to right and is PE for the left edge, PL for the right edge, and so on. Finally, the parameter t, the ratio of numerator and denominator, reduces to the distance to an edge divided by dx or dy, exactly the constant of proportionality we could calculate directly from the parametric line formulation. Note that it is important to preserve the signs of the numerator and denominator instead of cancelling minus signs, because the numerator and denominator as signed distances carry information that is used in the algorithm.

Full Marks: 60 **Answer Any Three** Time: 1:30 Hrs.

1. Let you have to draw a line from P_0(-8, 21) to P_1(-20,30).

(a) Derive initial deviation (d_{init}) and its essential derivatives of the above line using mid-point line-drawing algorithm. 4

(b) Write the algorithm for drawing the above line. 2

(c) Determine the next 8-pixel coordinates of the line starting from P_1 including the values of decision variables in each stage using midpoint line drawing algorithm. 4

2. Suppose you are instructed to draw an ellipse from region-2 to region-1, that is the starting point of the ellipse is (a, 0).

(a) Explain the termination criteria for the above case (i.e., from region-2 to region-1). 4

(b) Determine the first 8-pixel coordinates from (20, 0) of the above ellipse including the values of decision variables in each stage, given that a=20 and b=12. [Given that the center of the ellipse is (-20, 0)] 6

3. (a) Make a list of t for all edges in Cyrus-Beak line clipping algorithm 2

(b) Determine the value of t for each of the following lines for all edges, specify whether they are entering or leaving t. (Given (-120, -100) to (120, 100) are the diagonal corners the clip region). 4

(i) (-150, -200) to (150, 200). (ii) (200, 120) to (-300, -110).

(iii) (-150, 120) to (100, -100). (iv) (-250, 400) to (250, -200).

(c) Make region-outcode for each endpoint for the following 3D lines and determine whether the they are accepted / rejected / partial accepted using Cohen-Sutherland line clipping algorithm. Given that (-120, -100, -80) and (120, 100, 80) are the diagonal corners of the clip region. 4

(i) (-150, -200, -81) to (150, 200, -100). (ii) (200, 120, 85) to (-300, -110, 0).

(iii) (-120, 100, -80) to (100, -100, 80). (iv) (-250, 400, 120) to (250, -200, -60).

4. (a) Let A(0,0), B(6,3), C(6,7), D(3,4) *and* E(0,7) are the corners of a pentagon. Draw (i) Edge Table and (ii) Active Edge Table of $ABCDE$ pentagon for polygon filling using scan-line algorithm. Finally draw the boundary pixels from AET. 2.5+ 2.5+ 1

(b) Let A(10,10), B(90,30), C(70,50), D(100,80), E(60,60), F(30,90) and G(10,70) are the corners of a hexagon. Draw the new polygon using Sutherland-Hodgeman polygon clipping algorithm Given that (0, 0) and (80, 80) are the diagonal corners of the clip region. 4

Previous Year's Incourse Question

Suggestion of Sir for the upcoming incourse

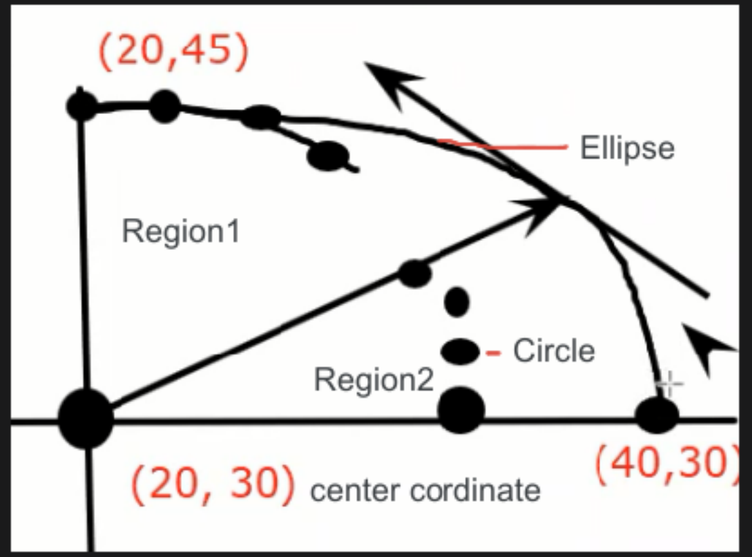
Suggestion from Sir regarding upcoming incourse exam

- Cyrus And Beck Algorithm
  - Derivation of t
  - Derive List of tL or tE for all edges
  - Value given for Ymax, Ymin, Xmax, Xmin, Find t for each boundary. Example:

> *The clipping boundary is(-200,-150) to (200,150). let a line is needed to be clipped for this boundary. Determine tE or tL for each boundary and finally find the value of tEmax and tLmin for following line. Line:(530,-480) to (-300,350)*

- Cohen-Sutherland Algorithm
  - 3D clipping is more important than 2D
  - Clipping Boundary will be given, Determine Outcode0 and Outcode1
  - Defination of Outcode
  - Algorithm is important
- Ellipse
  - Derivation of Regions(Region 1 and 2 derivation is most important)
  - Tremination criteria from one region to another region(region 1 to region 2 important)
    - In the book, clockwise termination is given . Sir, will give counter clockwise termination
  - Example Question:

> *Given center coordinate and (rx ,ry) or(a ,b). find coordinate of next 5 points.*



1

- Circle
  - Same as Ellipse.
  - Example Question:

> *Given center coordinate and (x,ry). find coordinate of next 5 points.(watch the diagram above)*

- Line(Question will be confusing)
  - How to determine Slope?
  - How to make slope independent line using 8 way symmetry?
  - Given the coordinate of starting point and ending point. Derive the line.
  - Determine the coordinate of first 5 pixels of line.

> **(PS): Don't make a mistake in determining zone of the line!**