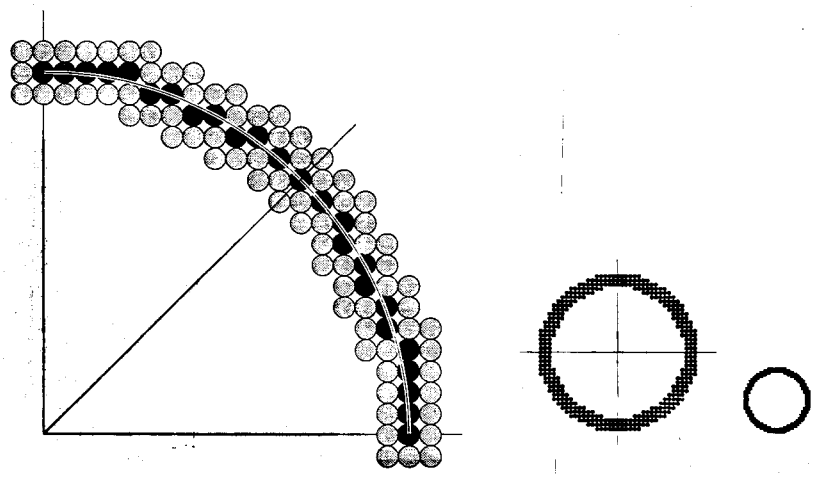

Figure 3.24 Thick circle drawn by tracing a rectangular pen.

Other methods for displaying thick primitives, such as filling areas between inner and outer boundaries constructed at a distance $t/2$ on either side of a single pixel trajectory, are discussed in Chapters 3 and 19 of [FOLE90].

3.8 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 when 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, 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.

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 ith 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 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.

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.

3.9 CLIPPING LINES

This section treats analytical clipping of lines against rectangles; 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 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 9), 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 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

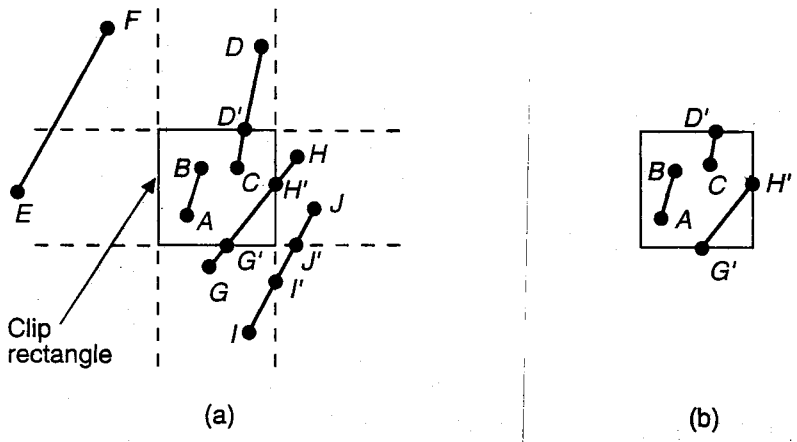

Figure 3.25 Cases for clipping lines.

are considered inside and hence are displayed. Figure 3.25 shows several examples of clipped lines.

3.9.1 Clipping Endpoints

Before we discuss clipping lines, let us 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.

3.9.2 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. 3.25), 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. 3.25, 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.

3.9.3 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. 3.25 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 7.11.2).

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. 3.26). 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:

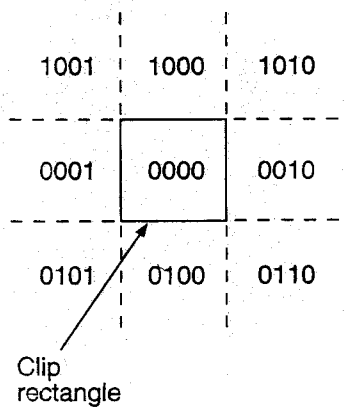

Figure 3.26
Region outcodes.

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 right of right edge	$x > x_{max}$
Fourth bit	outside halfplane of left edge, to 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. 3.25, 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. 3.27. Point *A* has outcode 0000, and point *D* has outcode 1001. The line can be neither trivially accepted nor 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.

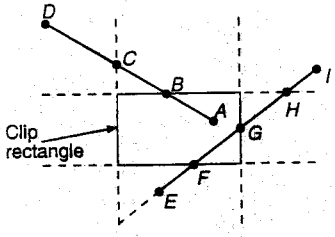

Figure 3.27
Illustration of Cohen–Sutherland line clipping.

In the code of Prog. 3.7, we use a C structure with fields as members to represent the outcode, because this representation is more natural than is an array of four integers. We use an external function to calculate the outcode for modularity; to improve performance, we would, of course, put this code in line.

Program 3.7

Cohen–Sutherland line-clipping algorithm.

```
typedef struct {
   unsigned all;
   unsigned left:4;
   unsigned right:4;
   unsigned bottom:4;
   unsigned top:4;
} outcode;

void  CohenSutherlandLineClipAndDraw(float x0, float y0, float x1, float y1,
            float xmin, float xmax, float ymin, float 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) */
{
   boolean   accept, done;
   outcode outcode0, outcode1, outcodeOut;
   float  x, y;

   accept = FALSE;
   done = FALSE;
   outcode0 = CompOutCode(x0, y0, xmin, xmax, ymin, ymax);
   outcode1 = CompOutCode(x1, y1, xmin, xmax, ymin, ymax);
   do {
```

```
        if (outcode0.all == 0 && outcode1.all == 0) {
            accept = TRUE;
            done = TRUE;
        } else if ((outcode0.all & outcode1.all) != 0)
            done = TRUE;        /* Logical intersection is true, so trivial reject and exit */
        else {
            if (outcode0.all != 0)
                outcodeOut = outcode0;
            else
                outcodeOut = outcode1;
            if (outcodeOut.top) {                   /* Divide line at top of clip rectangle */
                x = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);
                y = ymax;
            } else if (outcodeOut.bottom) {    /* Divide line at bottom of clip rectangle */
                x = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);
                y = ymin;
            } else if (outcodeOut.right) {  /* Divide line at right edge of clip rectangle */
                y = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);
                x = xmax;
            } else if (outcodeOut.left) {        /* Divide line at left edge of clip rectangle */
                y = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);
                x = xmin;
            }
            if (outcodeOut.all == outcode0.all) {
                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);
    if (accept)
        MidpointLineReal(x0, y0, x1, y1, value);    /* Version for float coordinates */
}

outcode CompOutCode (float x, float y,
            float xmin, float xmax, float ymin, float ymax)
{
    outcode code;
    code.top = 0, code.bottom = 0, code.right = 0, code.left = 0, code.all = 0;
    if (y > ymax) {
        code.top = 8;
        code.all += code.top;
    } else if (y < ymin) {
        code.bottom = 4;
        code.all += code.bottom;
    }
```

```
    if (x > xmax) {
        code.right = 2;
        code.all += code.right;
    } else if (x < xmin) {
        code.left = 1;
        code.all += code.left;
    }
    return code;
}
```

We can improve the efficiency of the algorithm slightly by not recalculating slopes (see Exercise 3.22). 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. 3.27). The Nicholl, Lee, and Nicholl [NICH87] algorithm, by contrast, avoids calculating external intersections by subdividing the plane into many more regions; it is discussed in Chapter 19 of [FOLE90]. An advantage of the much simpler Cohen–Sutherland algorithm is that its extension to a 3D orthographic view volume is straightforward, as can be seen in Section 6.6.3.

3.9.4 A Parametric Line-Clipping Algorithm

The Cohen–Sutherland algorithm is probably still the most commonly used line-clipping algorithm because it has been around the 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

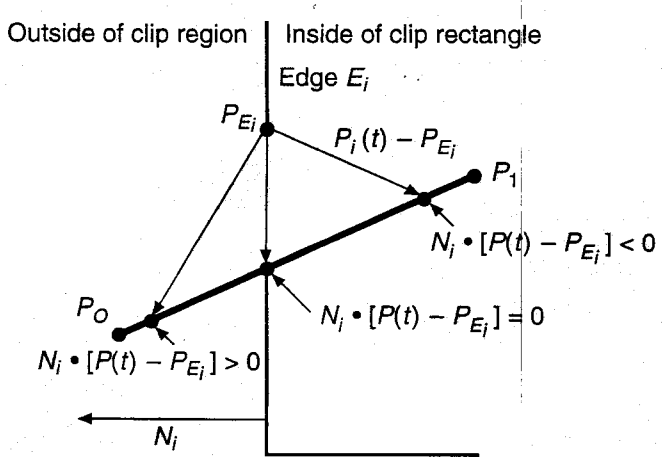

Figure 3.28 Dot products for three points outside, inside, and on the boundary of the clip region.

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 3.28 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)[7], 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

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

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:

[7] 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.

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

First, substitute for $P(t)$:

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

Next, group terms and distribute the dot product:

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

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. 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 simply try sorting the remaining values of t, choosing the intermediate values of t for intersection points, as suggested in Fig. 3.29 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. 3.29 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:

$$N_i \cdot D < 0 \Rightarrow \text{PE (angle greater than 90)},$$

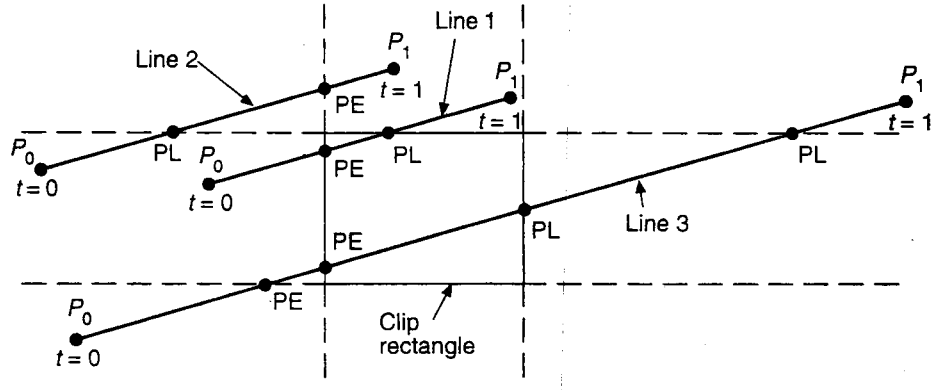

Figure 3.29 Lines lying diagonal to the clip rectangle.

$$N_i \cdot D > 0 \Rightarrow \text{PL (angle less than 90).}$$

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. 3.29 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 Prog. 3.8. The complete version of the code, adapted from [LIAN84], can be found in [FOLE90] as Fig. 3.45.

Program 3.8

Pseudocode for Cyrus–Beck parametric line-clipping algorithm.

```
{
    precalculate Ni and select a PEi for each edge

    for ( each line segment to be clipped ){
        if (P1 = P0)
            line is degenerate so clip as a point;
        else {
            tE = 0;
            tL = 1;
            for ( each candidate intersection with a clip edge ){
                if (Ni · D != 0) {          /* Ignore edges parallel to line */
                    calculate t;
```

```
                    use sign of N_i · D to categorize as PE or PL;
                    if (PE) t_E = max(t_E, t);
                    if (PL) t_L = min(t_L, t);
                }
            }
            if (t_E > t_L)
                return nil;
            else
                return P(t_E) and P(t_L) as true clip intersections;
        }
    }
}
```

In summary, the Cohen–Sutherland algorithm is efficient when outcode testing can be done cheaply (for example, by doing bitwise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. Parametric line clipping wins when many line segments need to be clipped, since the actual calculation of the coordinates of the intersection points is postponed until needed, and testing can be done on parameter values. This parameter calculation is done even for endpoints that would have been trivially accepted in the Cohen–Sutherland strategy, however. The Liang–Barsky algorithm is more efficient than the Cyrus–Beck version because of additional trivial rejection testing that can avoid calculation of all four parameter values for lines that do not intersect the clip rectangle. For lines that cannot be trivially rejected by Cohen–Sutherland because they do not lie in an invisible halfplane, the rejection tests of Liang–Barsky are clearly preferable to the repeated clipping required by Cohen–Sutherland. The Nicholl et al. algorithm [NICH87] is generally preferable to either Cohen–Sutherland or Liang–Barsky but does not generalize to 3D, as does parametric clipping. Speed-ups to Cohen–Sutherland are discussed in [DUVA90].

3.10 CLIPPING CIRCLES

To clip a circle against a rectangle, we can first do a trivial accept/reject test by intersecting the circle's extent (a square of the size of the circle's diameter) with the clip rectangle, using the algorithm in the next section for polygon clipping. If the circle intersects the rectangle, we divide it into quadrants and do the trivial accept or reject test for each. These tests may lead in turn to tests for octants. We can then compute the intersection of the circle and the edge analytically by solving their equations simultaneously, and then scan convert the resulting arcs using the appropriately initialized algorithm with the calculated (and suitably rounded) starting and ending points. If scan conversion is fast, or if the circle is not too large, it is probably more efficient to scissor on a pixel-by-pixel basis, testing each boundary pixel against the rectangle bounds before it is written. An extent check would certainly be useful in any case. If the circle is filled, spans of adjacent interior pixels on each scan line can be filled without bounds checking by clipping each span and then filling its interior pixels.

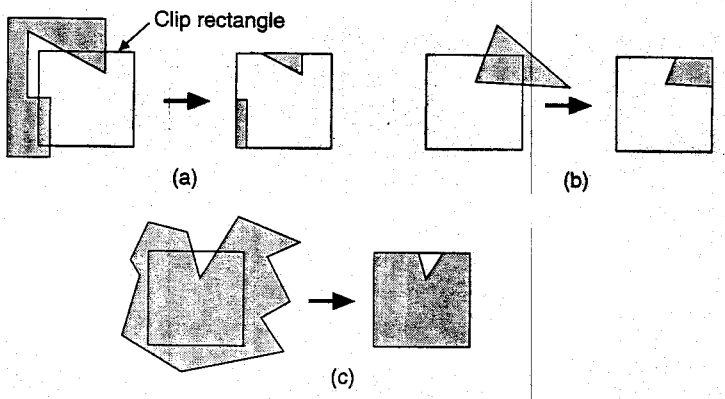

Figure 3.30 Examples of polygon clipping. (a) Multiple components. (b) Simple convex case. (c) Concave case with many exterior edges.

3.11 CLIPPING POLYGONS

An algorithm that clips a polygon must deal with many different cases, as shown in Fig. 3.30. The case in part (a) is particularly noteworthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

3.11.1 The Sutherland–Hodgman Polygon-Clipping Algorithm

Sutherland and Hodgman's polygon-clipping algorithm [SUTH74b] uses a *divide and conquer* strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle (Fig. 3.31), successively clip a polygon against a clip rectangle.

Note the difference between this strategy for a polygon and the Cohen–Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed and clips only when necessary. The actual Sutherland–Hodgman algorithm is in fact more general: A polygon (convex or concave) can be clipped against any convex clipping polygon; in 3D, polygons can be clipped against convex polyhedral volumes defined by planes. The algorithm accepts a series of polygon vertices $v_1, v_2, \ldots, v_n$. In 2D, the vertices define polygon edges from v_i to v_{i+1} and from v_n to v_1. The algorithm clips against a single, infinite clip edge and outputs another

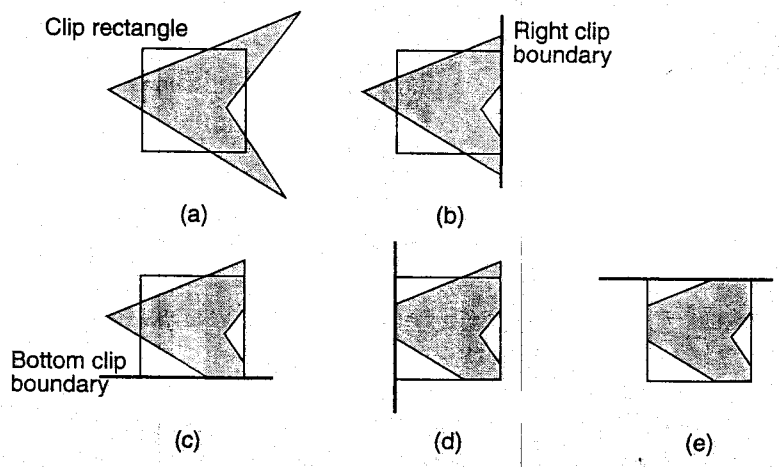

Figure 3.31 Polygon clipping, edge by edge. (a) Before clipping. (b) Clip on right. (c) Clip on bottom. (d) Clip on left. (e) Clip on top; polygon is fully clipped.

series of vertices defining the clipped polygon. In a second pass, the partially clipped polygon is then clipped against the second clip edge, and so on.

The algorithm moves around the polygon from v_n to v_1 and then on back to v_n, at each step examining the relationship between successive vertices and the clip edge. At each step, zero, one, or two vertices are added to the output list of vertices that defines the clipped polygon. Four possible cases must be analyzed, as shown in Fig. 3.32.

Let us consider the polygon edge from vertex s to vertex p in Fig. 3.32. Assume that start point s has been dealt with in the previous iteration. In case 1, when the polygon edge is completely inside the clip boundary, vertex p is added to the output list. In case 2, the intersection point i is output as a vertex because the edge intersects the boundary. In case 3, both vertices are outside the boundary, so there is no output. In case 4, the intersection point i and vertex p are both added to the output list.

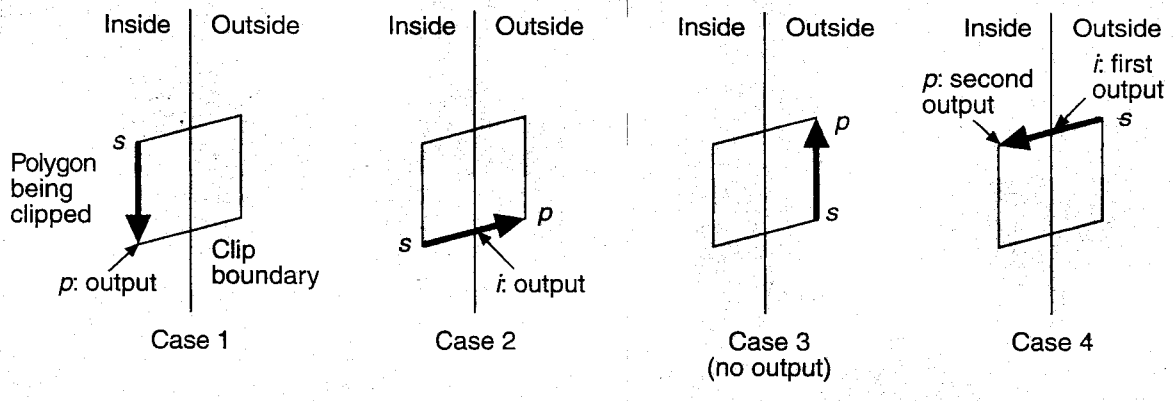

Figure 3.32 Four cases of polygon clipping.

Function SutherlandHodgmanPolygonClip( ) in Prog. 3.9 accepts an array, *inVertexArray,* of vertices and creates another array, *outVertexArray,* of vertices. To keep the code simple, we show no error checking on array bounds, and we use the function Output( ) to place a vertex into *outVertexArray.* The function Intersect( ) calculates the intersection of the polygon edge from vertex *s* to vertex *p* with *clipBoundary*, which is defined by two vertices on the clip polygon's boundary. The function Inside( ) returns TRUE if the vertex is on the inside of the clip boundary, where "inside" is defined as "to the left of the clip boundary when one looks from the first vertex to the second vertex of the clip boundary." This sense corresponds to a counterclockwise enumeration of edges. To calculate whether a point lies outside a clip boundary, we can test the sign of the dot product of the normal to the clip boundary and the polygon edge, as described in Section 3.9.4. (For the simple case of an upright clip rectangle, we need only test the sign of the horizontal or vertical distance to its boundary.)

Sutherland and Hodgman show how to structure the algorithm so that it is reentrant [SUTH74b]. As soon as a vertex is output, the clipper calls itself with that vertex. Clipping is performed against the next clip boundary, so that no intermediate storage is necessary for the partially clipped polygon: In essence, the polygon is passed through a *pipeline* of clippers. Each step can be implemented as special-purpose hardware with no intervening buffer space. This property (and its generality) makes the algorithm suitable for today's hardware implementations. In the algorithm as it stands, however, new edges may be introduced on the border of the clip rectangle. Consider Fig. 3.30(a)—a new edge is introduced by connecting the left top of the triangle and the left top of the rectangle. A postprocessing phase could eliminate these edges.

Program 3.9

Sutherland-Hodgman polygon-clipping algorithm.

```
typedef struct vertex {
    float  x, y;
} vertex;

typedef vertex edge[2];
typedef vertex vertexArray[MAX];              /* MAX is a declared constant */

void   Intersect(vertex first, vertex second, vertex *clipBoundary,
                 vertex *intersectPt)
{
    if (clipBoundary[0].y == clipBoundary[1].y) {          /* horizontal*/
        intersectPt–>y = clipBoundary[0].y;
        intersectPt–>x = first.x + (clipBoundary[0].y – first.y) *
            (second.x – first.x) / (second.y – first.y);
    } else {                                               /* vertical */
        intersectPt–>x = clipBoundary[0].x;
        intersectPt–>y = first.y + (clipBoundary[0].x – first.x) *
            (second.y – first.y) / (second.x – first.x);
    }
```

```
}

boolean  Inside(vertex testVertex, vertex *clipBoundary)
{
   if (clipBoundary[1].x > clipBoundary[0].x)              /* bottom */
      if (testVertex.y >= clipBoundary[0].y) return TRUE;
   if (clipBoundary[1].x < clipBoundary[0].x)              /* top */
      if (testVertex.y <= clipBoundary[0].y) return TRUE;
   if (clipBoundary[1].y > clipBoundary[0].y)              /* right */
      if (testVertex.x <= clipBoundary[1].x) return TRUE;
   if (clipBoundary[1].y < clipBoundary[0].y)              /* left */
      if (testVertex.x >= clipBoundary[1].x) return TRUE;
   return FALSE;
}

void  Output(vertex newVertex, int *outLength, vertex *outVertexArray)
{
   (*outLength)++;
   outVertexArray[*outLength - 1].x = newVertex.x;
   outVertexArray[*outLength - 1].y = newVertex.y;
}

void  SutherlandHodgmanPolygonClip(vertex *inVertexArray,
      vertex *outVertexArray, int inLength, int *outLength, vertex *clip_boundary)
{
   vertex s, p, i;
   int    j;

   *outLength = 0;
   s = inVertexArray[inLength - 1];      /* Start with the last vertex in inVertexArray */
   for (j = 0; j < inLength; j++) {
      p = inVertexArray[j];    /* Now s and p correspond to the vertices in Fig. 3.33 */
      if (Inside(p, clip_boundary)) {                          /* Cases 1 and 4 */
         if (Inside(s, clip_boundary))
            Output(p, outLength, outVertexArray);              /* Case 1 */
         else {                                                /* Case 4 */
            Intersect(s, p, clip_boundary, &i);
            Output(i, outLength, outVertexArray);
            Output(p, outLength, outVertexArray);
         }
      } else if (Inside(s, clip_boundary)) {                   /* Cases 2 and 3 */
         Intersect(s, p, clip_boundary, &i);                   /* Case 2 */
         Output(i, outLength, outVertexArray);
      }                                                        /* No action for case 3 */
      s = p;                                          /* Advance to next pair of vertices */
   }
}
```

3.12 GENERATING CHARACTERS

3.12.1 Defining and Clipping Characters

There are two basic techniques for defining characters. The most general but most computationally expensive way is to define each character as a curved or polygonal outline and to scan convert it as needed. We first discuss the other, simpler way, in which each character in a given font is specified as a small rectangular bitmap. Generating a character then entails simply using a copyPixel to copy the character's image from an offscreen canvas, called a **font cache**, into the frame buffer at the desired position.

The font cache may actually be in the frame buffer, as follows. In most graphics systems in which the display is refreshed from a private frame buffer, that memory is larger than is strictly required for storing the displayed image. For example, the pixels for a rectangular screen may be stored in a square memory, leaving a rectangular strip of *invisible* screen memory. Alternatively, there may be enough memory for two screens, one of which is being refreshed and one of which is being drawn in, to double-buffer the image. The font cache for the currently displayed font(s) is frequently stored in such invisible screen memory because the display controller's copyPixel works fastest within local image memory. A related use for such invisible memory is for saving screen areas temporarily obscured by popped-up images, such as windows, menus, and forms.

The bitmaps for the font cache are usually created by scanning in enlarged pictures of characters from typesetting fonts in various sizes; a typeface designer can then use a paint program to touch up individual pixels in each character's bitmap as necessary. Alternatively, the type designer may use a paint program to create, from scratch, fonts that are especially designed for screens and low-resolution printers. Since small bitmaps do not scale well, more than one bitmap must be defined for a given character in a given font just to provide various standard sizes. Furthermore, each type face requires its own set of bitmaps; therefore, a distinct font cache is needed for each font loaded by the application.

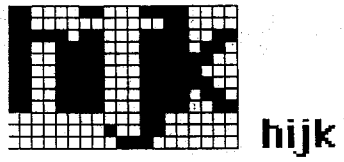

Figure 3.33
Portion of an example of a font cache.

Bitmap characters are clipped automatically by SRGP as part of its implementation of copyPixel. Each character is clipped to the destination rectangle on a pixel-by-pixel basis, a technique that lets us clip a character at any row or column of its bitmap. For systems with slow copyPixel operations, a much faster, although cruder, method is to clip the character or even the entire string on an all-or-nothing basis by doing a trivial accept of the character or string extent. Only if the extent is trivially accepted is the copyPixel applied to the character or string. Even for systems with a fast copyPixel, it is still useful to do trivial accept/reject testing of the string extent as a precursor to clipping individual characters during the copyPixel operation.

SRGP's simple bitmap font-cache technique stores the characters side by side in a canvas that is quite wide but is only as tall as the tallest character; Fig. 3.33 shows a portion of the cache, along with discrete instances of the same characters

at low resolution. Each loaded font is described by a struct (declared in Prog. 3.10) containing a reference to the canvas that stores the characters' images, along with information on the height of the characters and the amount of space to be placed between adjacent characters in a text string. (Some packages store the space between characters as part of a character's width, to allow variable intercharacter spacing.)

Program 3.10

Type declarations for the font cache.

```
typedef struct charLocation {
    int leftX, width;                    /*Horizontal location, width of image in font cache*/
} charLocation;

typedef struct fontCacheDescriptor {
    canvasIndexInteger cache;
    int descenderHeight, totalHeight;    /*Height is a constant; width varies*/
    int interCharacterSpacing;           /*Measured in pixels*/
    charLocation locationTable[128];
} fontCacheDescriptor;
```

As described in Section 2.1.5, descender height and total height are constants for a given font—the former is the number of rows of pixels at the bottom of the font cache used only by descenders, and the latter is simply the height of the font-cache canvas. The width of a character, on the other hand, is not considered a constant; thus, a character can occupy the space that suits it, rather than being forced into a fixed-width character box. SRGP puts a fixed amount of space between characters when it draws a text string, the amount being specified as part of each font's descriptor. A word-processing application can display lines of text by using SRGP to display individual words of text, and can right-justify lines by using variable spacing between words and after punctuation to fill out lines so that their rightmost characters are aligned at the right margin. This application involves using the text-extent inquiry facilities to determine where the right edge of each word is, in order to calculate the start of the next word. Needless to say, SRGP's text-handling facilities are really too crude for sophisticated word processing, let alone for typesetting programs, since such applications require far finer control over the spacing of individual letters to deal with such effects as sub- and superscripting, kerning, and printing text that is not horizontally aligned.

3.12.2 Implementing a Text Output Primitive

In the code of Prog. 3.11, we show how SRGP text is implemented internally: Each character in the given string is placed individually, and the space between characters is dictated by the appropriate field in the font descriptor. Note that complexities such as dealing with mixed fonts in a string must be handled by the application program.

Program 3.11

Implementation of character placement for SRGP's text primitive.

```
void SRGP_characterText(point origin, char *stringToPrint,
        fontCacheDescriptor fontInfo)
/* origin is where to place the character in the current canvas*/
{
```

```
    rectangle fontCacheRectangle;
    char charToPrint;
    int i;
    charLocation *fp;

/*Origin specified by the application is for baseline and does not include descender.*/
    origin.y -= fontInfo.descenderHeight;

    for (i = 0; i < strlen(stringToPrint); i++) {
        charToPrint = stringToPrint[i];
        fp = &fontInfo.locationTable[charToPrint];
/*Find the rectangular region within the cache wherein the character lies.*/
        fontCacheRectangle.bottomLeft = SRGP_defPoint(fp->leftX, 0);
        fontCacheRectangle.topRight = SRGP_defPoint(fp->leftX + fp->width - 1,
            fontInfo.totalHeight - 1);
        SRGP_copyPixel(fontInfo.cache, fontCacheRectangle, origin);
/*Update the origin to move past the new character plus intercharacter spacing*/
        origin.x += fp->width + interCharacterSpacing;
    }
}
```

We mentioned that the bitmap technique requires a distinct font cache for each combination of font, size, and face for each different resolution of display or output device supported. A single font in eight different point sizes and four faces (normal, bold, italic, bold italic) thus requires 32 font caches! One way to resolve this storage problem is to represent characters in an abstract, device-independent form using polygonal or curved outlines of their shapes defined with floating-point parameters, and then to transform them appropriately. Polynomial functions called **splines** (see Chapter 9) provide smooth curves with continuous first and higher derivatives and are commonly used to encode text outlines. Although each character definition takes up more space than its representation in a font cache, multiple sizes may be derived from a single stored representation by suitable scaling; also, italics may be quickly approximated by shearing the outline. Another major advantage of storing characters in a completely device-independent form is that the outlines may be arbitrarily translated, rotated, scaled, and clipped (or used as clipping regions themselves).

The storage economy of splined characters is not quite so great as this description suggests. For instance, not all point sizes for a character may be obtained by scaling a single abstract shape, because the shape for an aesthetically pleasing font is typically a function of point size; therefore, each shape suffices for only a limited range of point sizes. Moreover, scan conversion of splined text requires far more processing than does the simple copyPixel implementation, because the device-independent form must be converted to pixel coordinates on the basis of the current size, face, and transformation attributes. Thus, the font-cache technique is still the most common for personal computers and even is used for many workstations. A strategy that offers the best of both methods is to store the fonts in outline form but to convert the ones being used in a given application to their bitmap

equivalents—for example, to build a font cache on the fly. A more detailed discussion of splined text can be found in [FOLE90], Section 19.4.

3.13 SRGP_COPYPIXEL

If only WritePixel and ReadPixel low-level functions are available, the SRGP_copyPixel function can be implemented as a doubly nested for loop for each pixel. For simplicity, assume first that we are working with a bilevel display and do not need to deal with the low-level considerations of writing bits that are not word-aligned. In the inner loop of our simple SRGP_copyPixel, we do a ReadPixel of the source and destination pixels, logically combine them according to the SRGP write mode, and then WritePixel the result. Treating **replace** mode, the most common write mode, as a special case allows a simpler inner loop that does only a ReadPixel/WritePixel of the source into the destination, without having to do a logical operation. The clip rectangle is used during address calculation to restrict the region into which destination pixels are written.

3.14 ANTIALIASING

3.14.1 Increasing Resolution

The primitives drawn so far have a common problem: They have jagged edges. This undesirable effect, known as **the jaggies** or **staircasing**, is the result of an all-or-nothing approach to scan conversion in which each pixel either is replaced with the primitive's color or is left unchanged. Jaggies are an instance of a phenomenon known as **aliasing**. The application of techniques that reduce or eliminate aliasing is referred to as **antialiasing**, and primitives or images produced using these techniques are said to be **antialiased**. [FOLE90], in Chapter 14, discusses basic ideas from signal processing that explain how aliasing got its name, why it occurs, and how to reduce or eliminate it when creating pictures. Here, we content ourselves with a more intuitive explanation of why SRGP's primitives exhibit aliasing, and describe how to modify the line scan-conversion algorithm developed in this chapter to generate antialiased lines.

Consider using the midpoint algorithm to draw a 1-pixel-thick black line, with slope between 0 and 1, on a white background. In each column through which the line passes, the algorithm sets the color of the pixel that is closest to the line. Each time the line moves between columns in which the pixels closest to the line are not in the same row, there is a sharp jag in the line drawn into the canvas, as is clear in Fig. 3.34(a). The same is true for other scan-converted primitives that can assign only one of two intensity values to pixels.

Suppose we now use a display device with twice the horizontal and vertical resolution. As shown in Fig. 3.34(b), the line passes through twice as many columns and therefore has twice as many jags, but each jag is half as large in x and

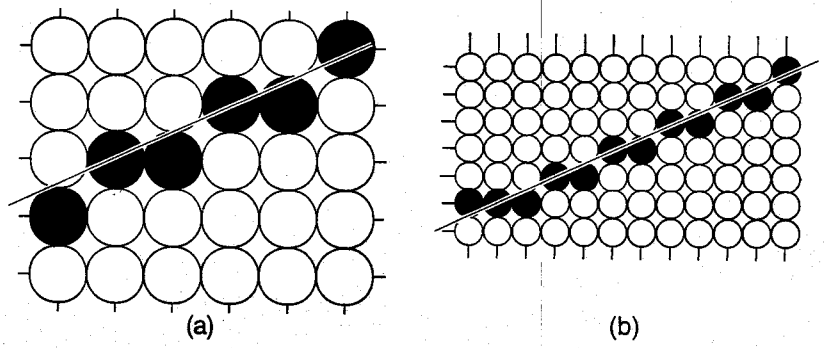

Figure 3.34 (a) Standard midpoint line on a bilevel display. (b) Same line on a display that has twice the linear resolution.

in y. Although the resulting picture looks better, the improvement comes at the price of quadrupling the memory cost, memory bandwidth, and scan-conversion time. Increasing resolution is an expensive solution that only diminishes the problem of jaggies—it does not eliminate the problem. In the following sections, we look at antialiasing techniques that are less costly, yet result in significantly better images.

3.14.2 Unweighted Area Sampling

The first approach to improving picture quality can be developed by recognizing that, although an ideal primitive such as the line has zero width, the primitive we are drawing has nonzero width. A scan-converted primitive occupies a finite area on the screen—even the thinnest horizontal or vertical line on a display surface is 1 pixel thick and lines at other angles have widths that vary over the primitive. Thus, we think of any line as a rectangle of a desired thickness covering a portion of the grid, as shown in Fig. 3.35. It follows that a line should not set the intensity of only a single pixel in a column to black, but rather should contribute some amount of intensity to each pixel in the columns whose area it intersects. (Such varying intensity can be shown on only those displays with multiple bits per pixel, of course.) Then, for 1-pixel-thick lines, only horizontal and vertical lines would affect exactly 1 pixel in their column or row. For lines at other angles, more than 1 pixel would now be set in a column or row, each to an appropriate intensity.

But what is the geometry of a pixel? How large is it? How much intensity should a line contribute to each pixel it intersects? It is computationally simple to assume that the pixels form an array of nonoverlapping square tiles covering the screen, centered on grid points, rather than disjoint circles as earlier in the chapter. [When we refer to a primitive overlapping all or a portion of a pixel, we mean that it covers (part of) the tile; to emphasize this we sometimes refer to the square as the *area represented by the pixel.*] We also assume that a line contributes to each pixel's intensity an amount proportional to the percentage of the pixel's tile it covers. A fully covered pixel on a black-and-white display will be colored black,

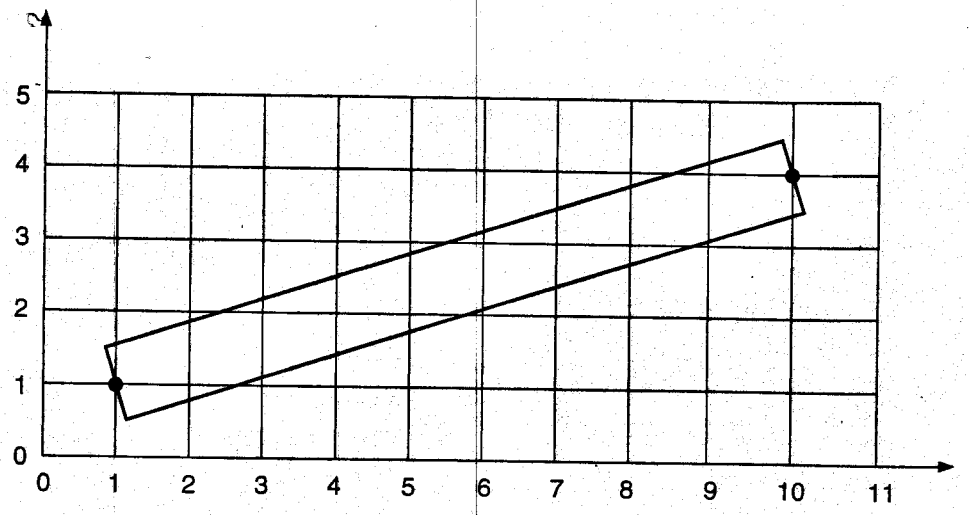

Figure 3.35 Line of nonzero width from point (1,1) to point (10,4).

whereas a partially covered pixel will be colored a gray whose intensity depends on the line's coverage of the pixel. This technique, as applied to the line shown in Fig. 3.35, is shown in Fig. 3.36.

For a black line on a white background, pixel (2, 1) is about 70 percent black, whereas pixel (2, 2) is about 25 percent black. Pixels such as (2, 3) not intersected by the line are completely white. Setting a pixel's intensity in proportion to the amount of its area covered by the primitive softens the harsh, on–off characteristic of the edge of the primitive and yields a more gradual transition between full on and full off. This blurring makes a line look better at a distance, despite the fact

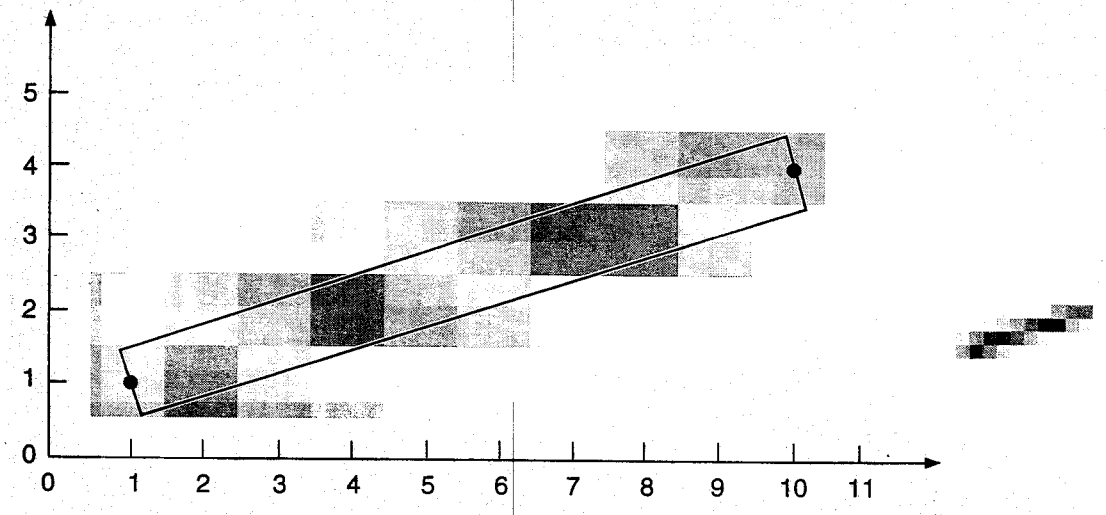

Figure 3.36 Intensity of a pixel is proportional to its area covered by the line.

that it spreads the on–off transition over multiple pixels in a column or row. A rough approximation to the area overlap can be found by dividing the pixel into a finer grid of rectangular subpixels, then counting the number of subpixels inside the line—for example, below the line's top edge or above its bottom edge. (See Exercise 3.25.)

We call the technique of setting intensity proportional to the amount of area covered **unweighted area sampling**. This technique produces noticeably better results than does setting pixels to full intensity or zero intensity, but there is an even more effective strategy called **weighted area sampling**. To explain the difference between the two forms of area sampling, we note that unweighted area sampling has the following three properties. First, the intensity of a pixel intersected by a line edge decreases as the distance between the pixel center and the edge increases: The farther away a primitive is, the less influence it has on a pixel's intensity. This relation obviously holds because the intensity decreases as the area of overlap decreases, and that area decreases as the line's edge moves away from the pixel's center and toward the boundary of the pixel. When the line covers the pixel completely, the overlap area and therefore the intensity are at a maximum; when the primitive edge is just tangent to the boundary, the area and therefore the intensity are zero.

A second property of unweighted area sampling is that a primitive cannot influence the intensity at a pixel at all if the primitive does not intersect the pixel—that is, if it does not intersect the square tile represented by the pixel. A third property of unweighted area sampling is that equal areas contribute equal intensity, regardless of the distance between the pixel's center and the area; only the total amount of overlapped area matters. Thus, a small area in the corner of the pixel contributes just as much as does an equal-sized area near the pixel's center.

3.14.3 Weighted Area Sampling

In weighted area sampling, we keep unweighted area sampling's first and second properties (intensity decreases with decreased area overlap, and primitives contribute only if they overlap the area represented by the pixel), but we alter the third property. We let equal areas contribute unequally: A small area closer to the pixel center has greater influence than does one at a greater distance. A theoretical basis for this change is given in [FOLE90], Chapter 14, where weighted area sampling is discussed in the context of filtering theory.

To retain the second property, we must make the following change in the geometry of the pixel. In unweighted area sampling, if an edge of a primitive is quite close to the boundary of the square tile we have used to represent a pixel until now, but does not actually intersect this boundary, it will not contribute to the pixel's intensity. In our new approach, the pixel represents a circular area larger than the square tile; the primitive *will* intersect this larger area; hence, it will contribute to the intensity of the pixel. Note that this means that the areas associated with adjacent pixels actually overlap.

To explain the origin of the adjectives *unweighted* and *weighted*, we define a **weighting function** that determines the influence on the intensity of a pixel of a

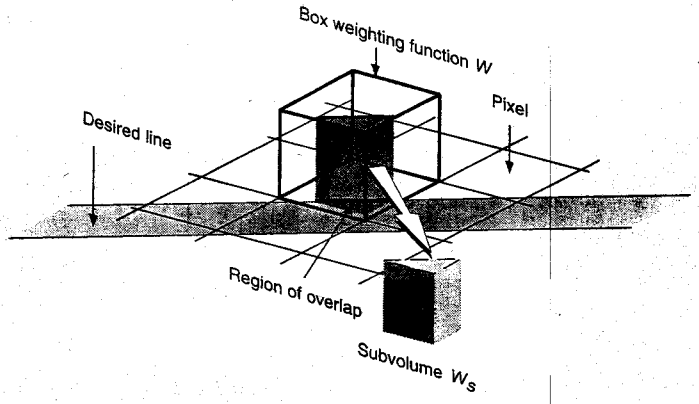

Figure 3.37 Box filter for square pixel.

given small area dA of a primitive, as a function of dA's distance from the center of the pixel. This function is constant for unweighted area sampling, and decreases with increasing distance for weighted area sampling. Think of the weighting function as a function, $W(x, y)$, on the plane, whose height above the (x, y) plane gives the weight for the area dA at (x, y). For unweighted area sampling with the pixels represented as square tiles, the graph of W is a box, as shown in Fig. 3.37.

The figure shows square pixels, with centers indicated by crosses at the intersections of grid lines; the weighting function is shown as a box whose base is that of the current pixel. The intensity contributed by the area of the pixel covered by the primitive is the total of intensity contributions from all small areas in the region of overlap between the primitive and the pixel. The intensity contributed by each small area is proportional to the area multiplied by the weight. Therefore, the total intensity is the integral of the weighting function over the area of overlap. The volume represented by this integral, W_S, is always a fraction between 0 and 1, and the pixel's intensity I is $I_{\max} \cdot W_S$. In Fig. 3.37, W_S is a wedge of the box. The weighting function is also called a **filter function**, and the box is also called a **box filter.** For unweighted area sampling, the height of the box is normalized to 1, so that the box's volume is 1, which causes a thick line covering the entire pixel to have an intensity $I = I_{\max} \cdot 1 = I_{\max}$.

Now let us construct a weighting function for weighted area sampling; it must give less weight to small areas farther away from the pixel center than it does to those closer. Let us pick a weighting function that is the simplest decreasing function of distance; for example, we choose a function that has a maximum at the center of the pixel and decreases linearly with increasing distance from the center. Because of rotational symmetry, the graph of this function forms a circular cone. The circular base of the cone (often called the **support** of the filter) should have a radius larger than you might expect; filtering theory shows that a good choice for the radius is the unit distance of the integer grid. Thus, a primitive fairly far from a pixel's center can still influence that pixel's intensity; also, the supports associated with neighboring pixels overlap, and therefore a single small piece of a primitive

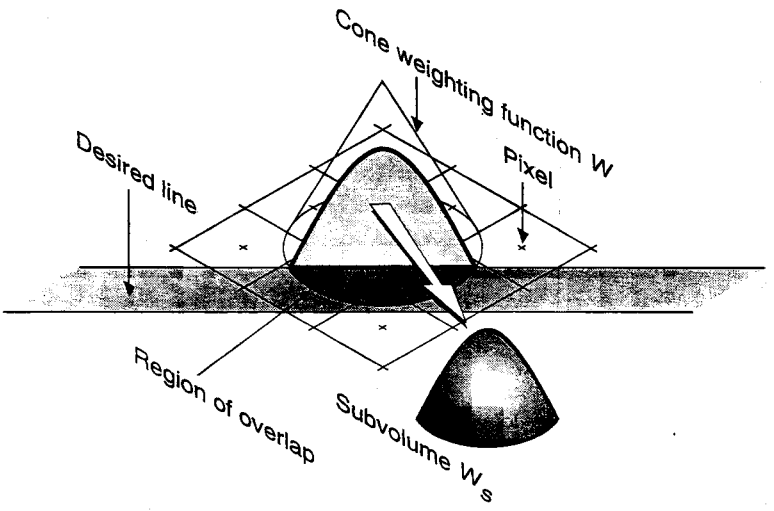

Figure 3.38 Cone filter for circular pixel with diameter of two grid units.

may actually contribute to several different pixels (Fig. 3.38). This overlap also ensures that there are no areas of the grid not covered by some pixel, which would be the case if the circular pixels had a radius of only one-half of a grid unit.[8]

As with the box filter, the sum of all intensity contributions for the cone filter is the volume under the cone and above the intersection of the cone's base and the primitive; this volume W_S is a vertical section of the cone, as shown in Fig. 3.38. As with the box filter, the height of the cone is first normalized so that the volume under the entire cone is 1; this allows a pixel whose support is completely covered by a primitive to be displayed at maximum intensity. Although contributions from areas of the primitive far from the pixel's center but still intersecting the support are rather small, a pixel whose center is sufficiently close to a line receives some intensity contribution from that line. Conversely, a pixel that, in the square-geometry model, was entirely covered by a line of unit thickness[9] is not quite as bright as it used to be. The net effect of weighted area sampling is to decrease the contrast between adjacent pixels in order to provide smoother transitions. In particular, with weighted area sampling, a horizontal or vertical line of unit thickness has more than 1 pixel intensified in each column or row, which would not be the case for unweighted area sampling.

The conical filter has two useful properties: rotational symmetry and linear decrease of the function with radial distance. We prefer rotational symmetry

[8] As noted in Section 3.2, pixels displayed on a CRT are roughly circular in cross-section, and adjacent pixels typically overlap; the model of overlapping circles used in weighted area sampling, however, is not directly related to this fact and holds even for display technologies, such as LCD screens or plasma panels, in which the physical pixels are actually nonoverlapping square tiles.

[9] We now say a "a line of unit thickness" rather than "a line 1 pixel thick" to make it clear that the unit of line width is still that of the SRGP grid, whereas the pixel's support has grown to have a two-unit diameter.

because it not only makes area calculations independent of the angle of the line, but also is theoretically optimal. Note, however, that the cone's linear slope (and its radius) are only an approximation to the optimal filter function, although the cone filter is still better than the box filter [FOLE90]. Optimal filters are computationally most expensive, box filters least, and therefore cone filters are a very reasonable compromise between cost and quality. We could integrate the cone filter into our scan-conversion algorithms rather easily. Details of this process can be found in [FOLE90].

3.15 ADVANCED TOPICS

We have only touched on the concepts of clipping and scan conversion in this chapter. In practice, many complex situations arise that require the application of advanced approaches. These are discussed fully in Chapter 19 of [FOLE90], but it will be useful to enumerate some of them here.

Clipping. While the clipping algorithms we have discussed in this chapter will perform correctly in most cases, they will not always do so efficiently. In addition, in some situations they will not be precise or even give the correct answer. Some of the improved algorithms are the Nicholl–Lee–Nicholl approach to 2D clipping, which exhibits a dramatic speedup over both the Liang–Barsky and Cohen–Sutherland algorithms. Also, clipping situations arise that we have not even considered, such as clipping general polygons against other general polygons. For this situation the Weiler polygon algorithm is useful.

Scan-converting primitives. We have only considered scan-conversion of simple primitives—lines, circles, and polygons. Not only are there more accurate and efficient algorithms for these objects, but there are methods for scan-converting more complicated primitives, as well, including ellipses, elliptical arcs, cubic curves, and general conic sections. There are also algorithms for thick primitives, where the boundary is not just a mathematical region, but has arbitrary width. Included in this class of problems is how to attractively and efficiently join thick line segments. Finally, filling self-intersecting polygons, where it is not clear which is inside and which is outside, must often be considered.

Antialiasing. The situations requiring antialiasing are much more numerous than for the straight line cases we have considered. Also, we must have a more complete knowledge of sampling theory in order to antialias correctly. Special algorithms are required for circles, conic sections, general curves, as well as rectangles, polygons, and line ends.

Text. Text is a highly specialized entity, and our earlier techniques are usually not sufficient. In this chapter, we discussed using a font cache to store characters that could then be copied directly to the bitmap, but we also observed certain limitations of this approach: A different cache may be needed for each size of text, and

the intercharacter spacing is fixed. Furthermore, although versions of bold or italic text can be created from this font cache, they are usually unsatisfactory. Even if we have a precise geometric drawing of a character, such as might be provided by a font designer, we cannot scan convert it on a stroke-by-stroke basis. The results will generally be unacceptable. Rather, specialized techniques have been developed to display text, including antialiasing.

Filling algorithms. Sometimes, after drawing a sequence of primitives, we may wish to color them in, or we may wish to color in a region defined by a freehand drawing. For example, it may be easier to make a mosaic pattern by creating a grid of lines and then filling it in with various colors than it is to lay down the colored squares evenly in the first place. Note that, when the first technique is used, no 2D primitives are drawn: We use only the 2D areas that happen to make up the background after the lines are drawn. Thus, determining how large a region to color amounts to detecting when a border is reached. Algorithms to perform this operation are called *fill algorithms*. Among those used are *boundary fill*, *flood fill*, and *tint fill* algorithms. Each has a special purpose and many graphics systems offer them all.

SUMMARY

In this chapter, we have taken our first look at the fundamental clipping and scan-conversion algorithms that are the meat and potatoes of raster graphics packages. We have covered only the basics here; many elaborations and special cases must be considered for robust implementations. You should consult Chapters 14, 17, and 19 of [FOLE90] for a fuller treatment of these issues.

The most important idea of this chapter is that, since speed is essential in interactive raster graphics, incremental scan-conversion algorithms using only integer operations in their inner loops are usually the best. The basic algorithms can be extended to handle thickness, as well as patterns for boundaries or for filling areas. Whereas the basic algorithms that convert single-pixel-wide primitives try to minimize the error between chosen pixels on the Cartesian grid and the ideal primitive defined on the plane, the algorithms for thick primitives can trade off quality and *correctness* for speed. Although much of 2D raster graphics today still operates, even on color displays, with single-bit-per-pixel primitives, we expect that techniques for real-time antialiasing will soon become prevalent.

Exercises

3.1 Implement the special-case code for scan converting horizontal and vertical lines, and lines with slopes of ± 1.

3.2 Modify the midpoint algorithm for scan converting lines (Prog. 3.2) to handle lines at any angle.

3.3 Show why the point-to-line error is always $\leq \frac{1}{2}$ for the midpoint line scan-conversion algorithm.

3.4 Modify the midpoint algorithm for scan converting lines of Exercise 3.2 to handle endpoint order and intersections with clip edges, as discussed in Section 3.2.3.