CHAPTER SIX

WINDOWING AND CLIPPING

INTRODUCTION

An architect may have a graphics program to draw an entire building but be interested in only the ground floor. A businessman may have a map of sales for the entire nation but be interested in only the northeast and southwest. An integrated circuit designer may have a program for displaying an entire chip but be interested in only a few registers. Often, the computer is used in design applications because it can easily and accurately create, store, and modify very complex drawings. When drawings are too complex, however, they become difficult to read. In such situations it is useful to display only those portions of the drawing that are of immediate interest. This gives the effect of looking at the image through a window. Furthermore, it is desirable to enlarge these portions to take full advantage of the available display surface. The method for selecting and enlarging portions of a drawing is called *windowing*. The technique for not showing that part of the drawing which one is not interested in is called *clipping*. (The term windowing has been used to mean several different things, all related to partitioning of the display. A single consistent definition is used within this text, but other definitions are found in the literature and may be a source of confusion.)

In this chapter we shall consider the ideas of windowing, clipping, and viewing transformation. We will add to our graphics package the routines for setting windows and viewports and for removing lines which lie outside the region we wish to display. The transformation routines of Chapter 4 will guide us in setting up a viewing transformation which will be applied to each display-file instruction as the instruction is created.

THE VIEWING TRANSFORMATION

It is often useful to think of two models of the item we are displaying. There is the object model and there is the image of the object which appears on the display. When we speak of the object, we are actually referring to a model of the object stored in the computer. The object model is said to reside in *object space*. This model represents the object using the physical units of length. In the object space, lengths of the object may be measured in any units from light-years to angstroms. The lengths of the image on the screen, however, must be measured in screen coordinates (we have normalized the screen coordinates so that they range between 0 and 1). (See Figure 6-1.)

We must have some way of converting from the object space units of measure to those of the *image space* (screen space). This can be done by the scaling transformation of Chapter 4. By scaling, we can uniformly reduce the size of the object until its dimensions lie between 0 and 1. Very small objects can be enlarged until their overall dimension is almost 1 unit. The physical dimensions of the object are scaled until they are suitable for display. (See Figure 6-2.)

It may be, however, that the object is too complex to show in its entirety or that we are particularly interested in just a portion of it. We would like to imagine a box about a portion of the object. We would only display what is enclosed in the box. Such a box is called a *window*. (See Figure 6-3.)

It might also happen that we do not wish to use the entire screen for display. We would like to imagine a box on the screen and have the image confined to that box. Such a box in the screen space is called a *viewport*. (See Figure 6-4.)

When the window is changed, we see a different part of the object shown at the same position on the display. (See Figure 6-5.) If we change the viewport, we see the same part of the object drawn at a different place on the display. (See Figure 6-6.)

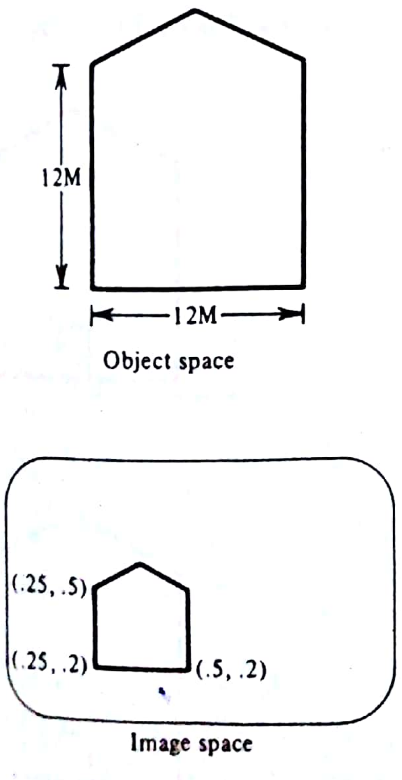

FIGURE 6-1
In the object space, position is measured in physical units, such as meters. In the image space, position is given in normalized screen coordinates.

Scanned by CamScanner

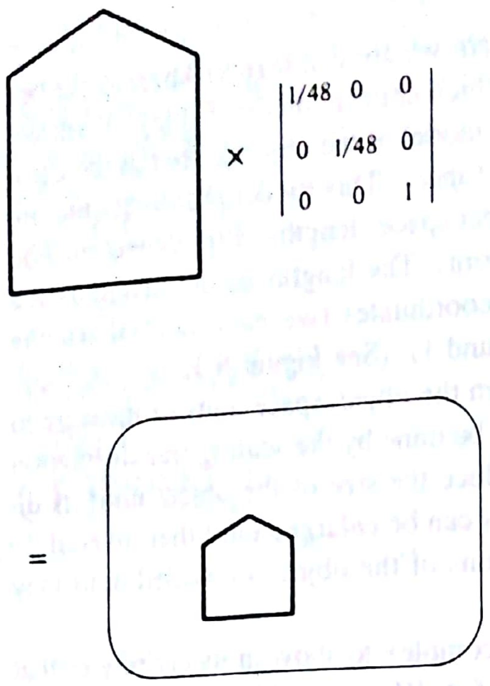
$\times \begin{vmatrix} 1/48 & 0 & 0 \\ 0 & 1/48 & 0 \\ 0 & 0 & 1 \end{vmatrix}$
=

FIGURE 6-2
A scaling transformation will convert object coordinate units to normalized screen coordinates.

In specifying both window and viewport, we have enough information to determine the translation and scaling transformations necessary to map from the object space to the image space. This can be done with the following three steps. First, the object together with its window is translated until the lower-left corner of the window is at the origin. Second, the object and window are scaled until the window has the dimensions of the viewport. In effect, this converts object and window into image and

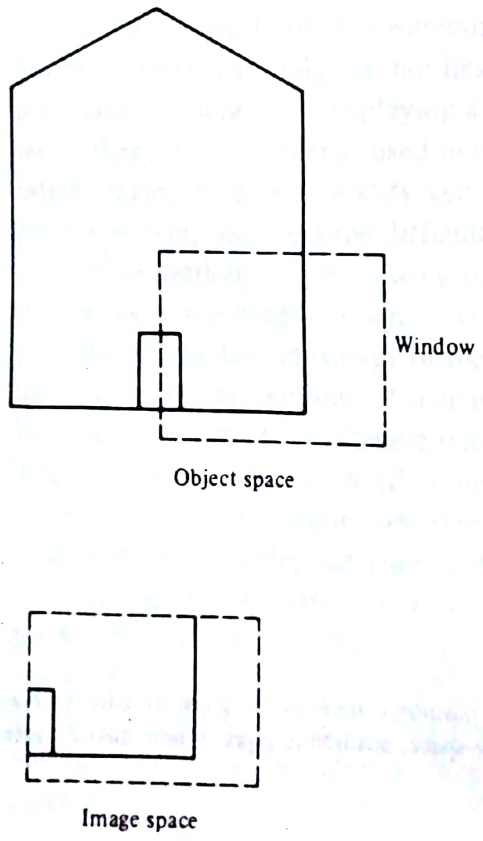

FIGURE 6-3
A window to view only part of an object.

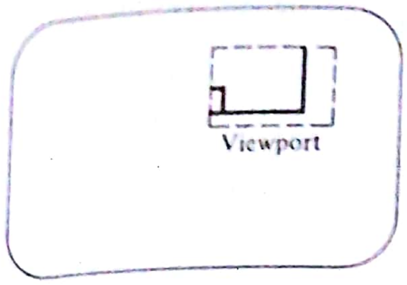

FIGURE 6-4
A viewport to define the part of the screen to be used.

viewport. The final transformation step is another translation to move the viewport to its correct position on the screen. (See Figure 6-7.)

We are really trying to do two things. We are changing the window size to become the size of the viewport (scaling) and we are positioning it at the desired location on the screen (translating). The positioning is just moving the lower-left corner of the window to the viewport's lower-left corner location, but we do this in two steps. We first move the corner to the origin and second move it to the viewport corner location. We take two steps because while it is at the origin, we can perform the necessary scaling without disturbing the corner's position.

The overall transformation which performs these three steps we shall call the *viewing transformation*. It creates a particular view of the object. (See Figure 6-8.)

Let us consider an example of a viewing transformation. If our window has left and right boundaries of 3 and 5 and lower and upper boundaries of 0 and 4, then the first translation matrix would be

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{vmatrix}$$

Suppose the viewport is the upper-right quadrant of the screen with boundaries at 0.5 and 1.0 for both x and y directions. The length of the window is 5 − 3 = 2 in the

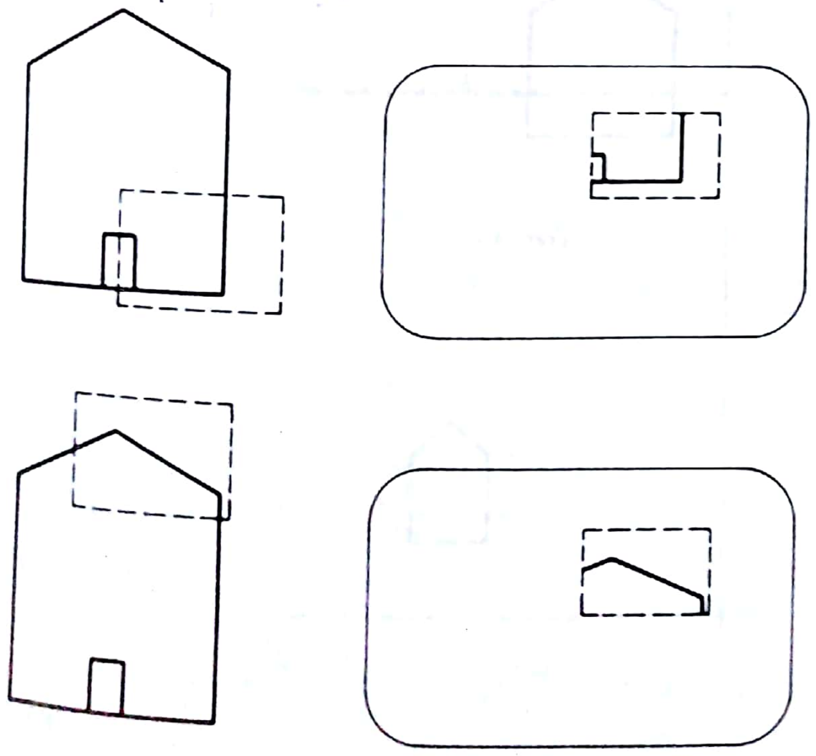

FIGURE 6-5
Different windows, same viewports.

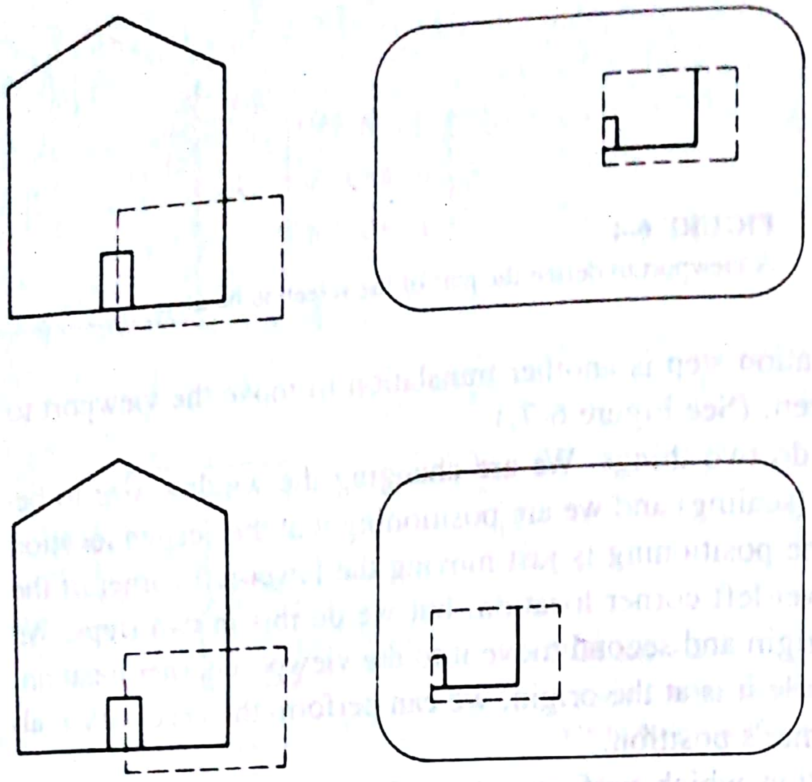

FIGURE 6-6
Same windows, different viewports.

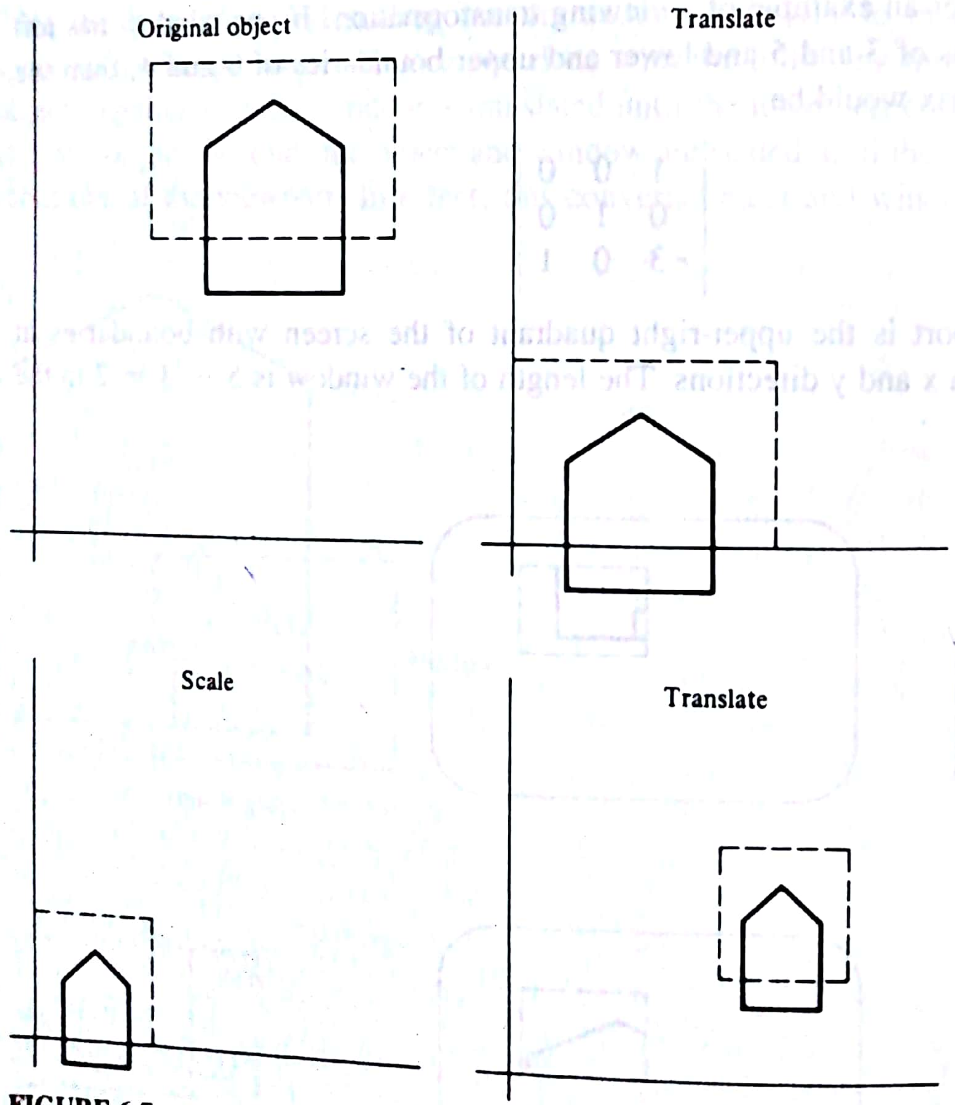

FIGURE 6-7
Steps in the viewing transformation.

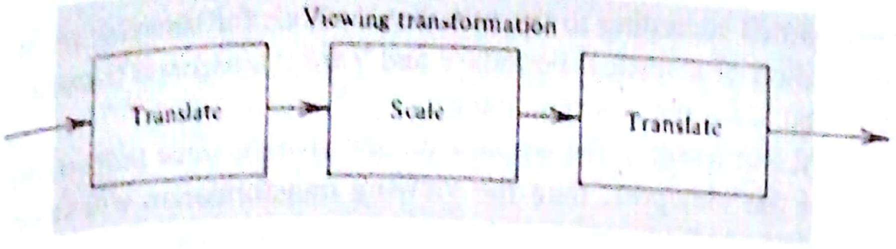

FIGURE 6-8
The viewing transformation.

x direction. The length of the viewport is 1.0 − 0.5 = 0.5, so the x scale factor is 0.5/2 = 0.25. In the y direction, we have the factor 0.5/4 = 0.125, so the scaling transformation matrix will be

$$\begin{vmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Finally, to position the viewport requires a translation of

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{vmatrix}$$

The viewing transformation is then

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{vmatrix} \begin{vmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{vmatrix} = \begin{vmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ -0.25 & 0.5 & 1 \end{vmatrix} \qquad (6.1)$$

In general, the viewing transformation is

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -WXL & -WYL & 1 \end{vmatrix} \begin{vmatrix} \frac{(VXH-VXL)}{(WXH-WXL)} & 0 & 0 \\ 0 & \frac{(VYH-VYL)}{(WYH-WYL)} & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ VXL & VYL & 1 \end{vmatrix}$$

$$= \begin{vmatrix} \frac{(VXH-VXL)}{(WXH-WXL)} & 0 & 0 \\ 0 & \frac{(VYH-VYL)}{(WYH-WYL)} & 0 \\ VXL - WXL\frac{(VXH-VXL)}{(WXH-WXL)} & VYL - WYL\frac{(VYH-VYL)}{(WYH-WYL)} & 1 \end{vmatrix} \qquad (6.2)$$

The variables have been named according to the rule that V stands for viewport and W for window, X for the position of a vertical boundary and Y for a horizontal boundary, H for the high boundary and L for the low boundary.

Note that if the height and width of the window do not have the same proportions as the height and width of the viewport, then the viewing transformation will cause some distortion of the image in order to squeeze the shape selected by the window into the shape presented by the viewport.

VIEWING TRANSFORMATION IMPLEMENTATION

The first step of our viewing transformation is specifying the size of the window. We will confine our window to a rectangular shape parallel with the x and y axes. By doing this, we need only specify the smallest and largest possible x values and the smallest and largest possible y values. Our routine for specifying the size of the window will store these boundary values in global variables so that they will be available when it comes time to perform the transformation. Likewise, we must specify the boundaries of the viewport. The length of the viewport or window in either the x or the y dimension is determined by subtracting the lower boundary from the upper boundary. Note that we cannot have the lower boundary equal to the upper boundary for a window because this would cause us to divide by zero when we try to determine the scaling transformation. Algorithms for setting the viewport and window dimensions are given below.

6.1 Algorithm SET-VIEWPORT(XL, XH, YL, YH) User routine for specifying the viewport

```
Arguments  XL, XH the left and right viewport boundaries
           YL, YH the bottom and top viewport boundaries
Global     VXL-HOLD, VXH-HOLD, VYL-HOLD, VYH-HOLD
           storage for the viewport boundaries
BEGIN
   IF XL ≥ XH OR YL ≥ YH THEN RETURN ERROR 'BAD VIEWPORT';
   VXL-HOLD ← XL;
   VXH-HOLD ← XH;
   VYL-HOLD ← YL;
   VYH-HOLD ← YH;
   RETURN;
END;
```

6.2 Algorithm SET-WINDOW(XL, XH, YL, YH) User routine for specifying the window

```
Arguments  XL, XH the left and right window boundaries
           YL, YH the bottom and top window boundaries
Global     WXL-HOLD, WXH-HOLD, WYL-HOLD, WYH-HOLD
           storage for the window boundaries
BEGIN
   IF XL ≥ XH OR YL ≥ YH THEN RETURN ERROR 'BAD WINDOW';
   WXL-HOLD ← XL;
```

```
    WXH-HOLD ← XH;
    WYL-HOLD ← YL;
    WYH-HOLD ← YH;
    RETURN;
END;
```

In our system we shall not change viewing parameters in the middle of a segment. Each segment is treated as a snapshot of the object. The viewing transformation describes how the camera is positioned. In this model it is reasonable to prohibit movement of the camera while taking the picture. We shall follow this rule by keeping two sets of viewing parameters, one set for the user to change and a second set to actually be used in the windowing and clipping routines. Changing the window in our system becomes a two-step process. First, the user changes his set of window and viewport boundaries. Second, the user's values are copied into the variables actually used by the windowing and clipping routines. By performing this copying as part of the segment-creation process, we ensure that changes in the viewing parameters being used cannot occur in the middle of a segment. (See Figure 6-9.)

We should note that this restriction on changing viewing parameters is just a rule for the particular viewing model we have chosen. There is no fundamental reason why a system could not be written which would allow changing the view at any point; in fact, while the CORE graphic system matches our approach, the GKS system does allow change of the viewing transformation within segments.

We shall write an algorithm for copying the user's specifications into the system's viewing parameters. This routine also calculates the window-to-viewport scale factors.

6.3 Algorithm NEW-VIEW-2 Set the clipping and viewing parameters from the current window and viewport specifications

Global	WXL-HOLD, WYL-HOLD, WXH-HOLD, WYH-HOLD the user's window parameters
	VXL-HOLD, VYL-HOLD, VXH-HOLD, VYH-HOLD the user's viewport parameters
	WXL, WYL, WXH, WYH, VXL, VYL, VXH, VYH the current clipping parameters
	WSX, WSY the window-to-viewport scale factors

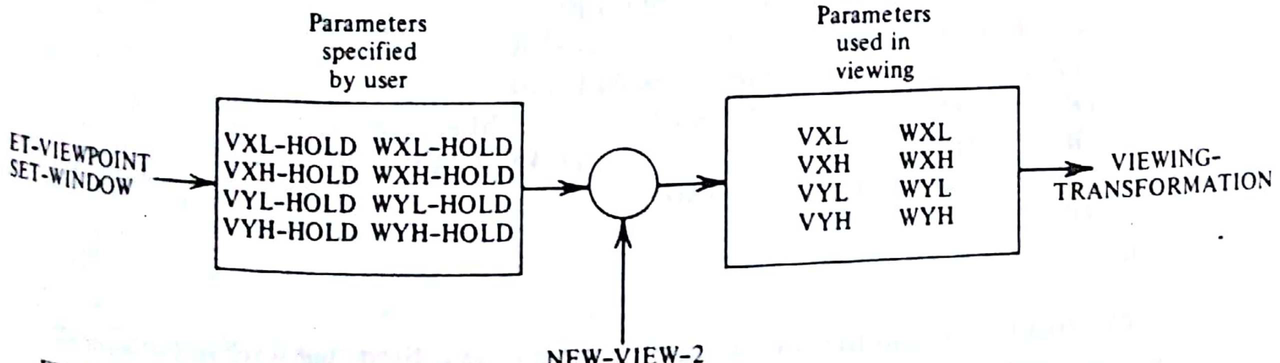

FIGURE 6-9
A NEW-VIEW action is required to make the user's specification that which is used in viewing.

```
BEGIN
    WXL ← WXL-HOLD;
    WYL ← WYL-HOLD;
    WXH ← WXH-HOLD;
    WYH ← WYH-HOLD;
    VXL ← VXL-HOLD;
    VYL ← VYL-HOLD;
    VXH ← VXH-HOLD;
    VYH ← VYH-HOLD;
    WSX ← (VXH − VXL) / (WXH − WXL);
    WSY ← (VYH − VYL) / (WYH − WYL);
    RETURN;
END;
```

We wish any given window setting to apply to an entire display-file segment. We can enforce this restriction by only allowing the above copying of parameters to occur when a segment is created. We shall therefore modify our segment-creation routine to reset the viewing transformation to match the latest user request.

6.4 Algorithm CREATE-SEGMENT(SEGMENT-NAME) (Modification of Algorithm 5.1) User routine to create a named segment

Argument	SEGMENT-NAME the segment name
Global	NOW-OPEN the segment currently open
	FREE the index of the next free display-file cell
	SEGMENT-START, SEGMENT-SIZE, VISIBILITY ANGLE, SCALE-X, SCALE-Y, TRANSLATE-X, TRANSLATE-Y the segment-table arrays
Constant	NUMBER-OF-SEGMENTS size of the segment table

```
BEGIN
    IF NOW-OPEN > 0 THEN RETURN ERROR 'SEGMENT STILL OPEN';
    IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
    THEN
        RETURN ERROR 'INVALID SEGMENT NAME';
    IF SEGMENT-SIZE[SEGMENT-NAME] > 0 THEN
        RETURN ERROR 'SEGMENT ALREADY EXISTS';
    NEW-VIEW-2;
    SEGMENT-START[SEGMENT-NAME] ← FREE;
    SEGMENT-SIZE[SEGMENT-NAME] ← 0;
    VISIBILITY[SEGMENT-NAME] ← VISIBILITY[0];
    ANGLE[SEGMENT-NAME] ← ANGLE[0];
    SCALE-X[SEGMENT-NAME] ← SCALE-X[0];
    SCALE-Y[SEGMENT-NAME] ← SCALE-Y[0];
    TRANSLATE-X[SEGMENT-NAME] ← TRANSLATE-X[0];
    TRANSLATE-Y[SEGMENT-NAME] ← TRANSLATE-Y[0];
    NOW-OPEN ← SEGMENT-NAME;
    RETURN;
END;
```

We wish to perform the following transformations. First, we wish to translate by the lower x and y boundaries of the window. This moves the lower-left corner of the window to the origin. Second, we wish to scale by the size of the viewport divided by the size of the window. This changes the dimensions of the window to those of the viewport. Finally, we wish to translate by the lower x and y boundary values of the viewport. This moves the lower-left corner from the origin to the correct viewport position. We can form each of these transformation matrices as we did in Chapter 4. We can multiply the matrices together to form a single transformation (Equation 6.2) and then apply it to a general point. This yields the following viewing transformation algorithm:

6.5 Algorithm VIEWING-TRANSFORM(OP, X, Y) Viewing transformation of a point

```
Arguments   OP, X, Y the instruction to be transformed
Global      WXL, WYL, WSX, WSY, VXL, VYL window and viewport parameters
Local       X1, Y1 the transformed point
BEGIN
    X1 ← (X − WXL) * WSX + VXL;
    Y1 ← (Y − WYL) * WSY + VYL;
    PUT-POINT(OP, X1, Y1);
    RETURN;
END;
```

Note that the above algorithm not only performs the viewing transformation but also enters the resulting instruction into the display file. The display file will hold the image space model.

CLIPPING

Now that we have seen how our picture may be correctly scaled and positioned, we shall consider how to cut off the lines which are outside the window so that only the lines within the window are displayed. This process is called *clipping*. In clipping we examine each line of the display to determine whether or not it is completely inside the window, lies completely outside the window, or crosses a window boundary. If it is inside, the line is displayed; if it is outside, nothing is drawn. If it crosses the boundary, we must determine the point of intersection and draw only the portion which lies inside. (See Figure 6-10.)

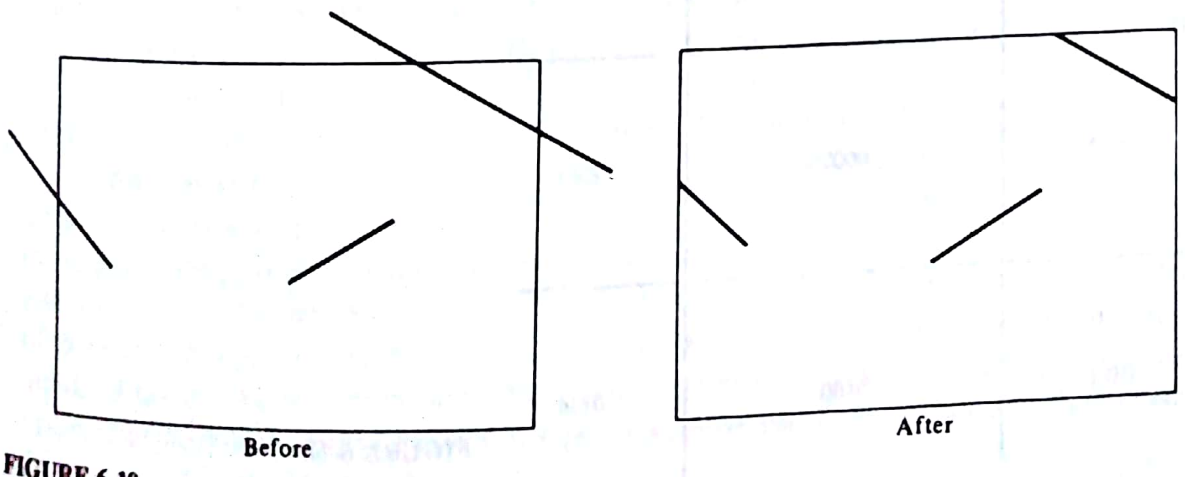

FIGURE 6-10
Clipping.

Different graphic elements may require different clipping techniques. A character, for example, may be either entirely included or omitted depending on whether or not its center lies within the window. This technique will not work for lines, and some methods used for lines will not work for polygons.

THE COHEN-SUTHERLAND OUTCODE ALGORITHM

A popular method for clipping lines is the *Cohen-Sutherland Outcode algorithm*. The algorithm quickly removes lines which lie entirely to one side of the clipping region (both endpoints above, or below, or right, or left). The algorithm makes clever use of bit operations (outcodes) to perform this test efficiently. Segment endpoints are each given 4-bit binary codes. The high-order bit is set to 1 if the point is above the window; the next bit is set to 1 if the point is below the window; the third and fourth bits indicate right and left of the window, respectively. The lines which form the window boundary divide the plane into nine regions with the outcodes shown in Figure 6-11.

If the line is entirely within the window, then both endpoints will have outcode 0000. Segments with this property are accepted (segment ST in Figure 6-12). If the line segment lies entirely on one side of the window (say entirely above it), then both endpoints will have a 1 in the outcode bit position for that side (the first bit will be 1 for both endpoints). We can check to see if the line is entirely on one side of the window by taking the logical AND of the outcodes for the two endpoints. If the result of the AND operation is nonzero, then the line segment may be rejected. Thus one test decides if the line segment is entirely above, or entirely below, or entirely to the right, or entirely to the left of the window. For example, segments AB and CD in Figure 6-12 would be quickly removed.

The difficult cases occur when a line crosses one or more of the lines which contain the clipping boundary (such as segments EF and IJ). For these cases, the point of

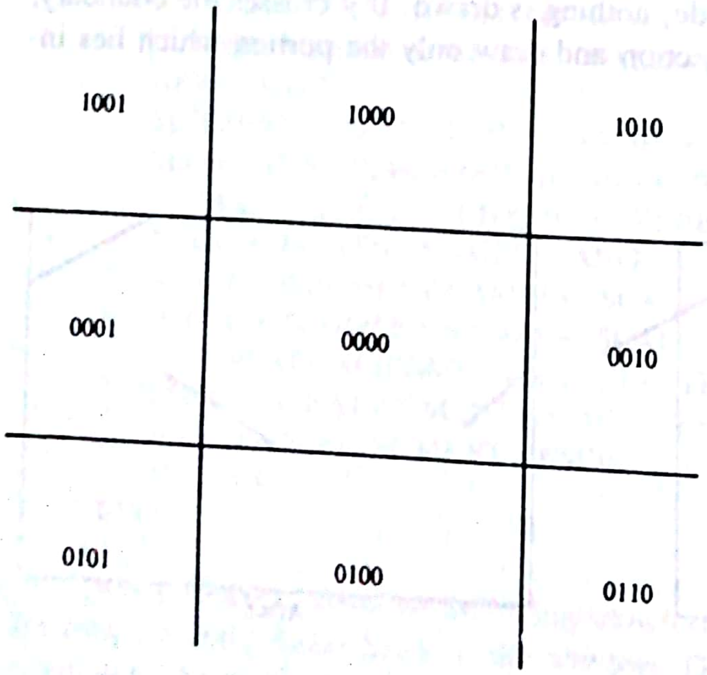

FIGURE 6-11
Outcodes for the plane.

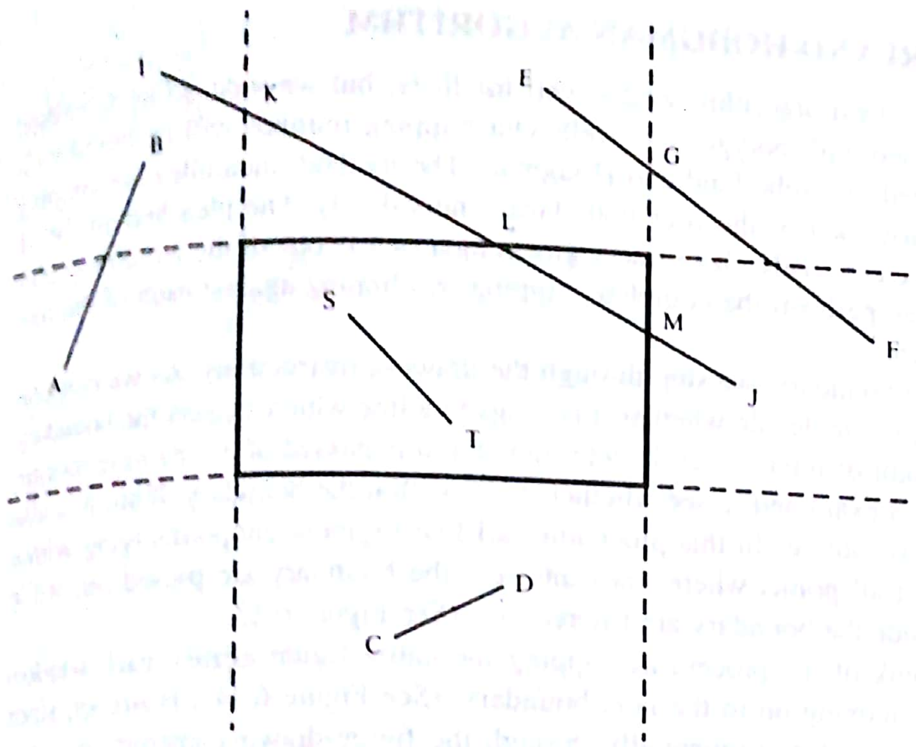

FIGURE 6-12
Testing and dividing line segments.

intersection between the line segment and clipping boundary lines may be used to break up the line segment. The resulting pieces may be tested for acceptance or rejection. Segment EF may be broken into EG and GF, where EG lies above and GF lies to the right, so both would be rejected. Segment IJ might be divided into IK and KJ. IK can be rejected because it lies to the left, but KJ must be further divided. Forming KL and LJ, we see that KL may be rejected as lying above but LJ must still be divided into LM and MJ. LM is contained and accepted, while MJ is to the right and rejected. At worst, the intersections with all four boundary lines will be calculated in order to clip the line.

The following is a brief outline of the algorithm (the details are left as an exercise): First, we compute the outcodes for the two endpoints (p_1 and p_2) of the segment. Next, we enter a loop. Within the loop we check to see if both outcodes are zero; if so, we enter the segment into the display file, exit the loop, and return. If the outcodes are not both zero, then we perform the logical AND function and check for a nonzero result. If this test is nonzero, then we reject the line, exit the loop, and return. If neither of these tests is satisfied, we must subdivide the line segment and repeat the loop. If the outcode for p_1 is zero, exchange the points p_1 and p_2 and also their outcodes. Find a nonzero bit in the outcode of p_1. If it is the high-order bit, then find the intersection of the line with the top boundary of the window. If it is the next bit position, then subdivide along the bottom boundary. The other two bits indicate that the right and left boundaries should be used. Replace the point p_1 with the intersection point and calculate its outcode. Repeat the loop.

THE SUTHERLAND-HODGMAN ALGORITHM

The Cohen-Sutherland algorithm works well for lines, but we would like a method which may be used with polygons as well. Our clipping routines will be based on a method discovered by Sutherland and Hodgman. The method unbundles the clipping test to clip against each of the four boundaries individually. The idea behind the algorithm is that we can easily clip a line segment against any one of the window boundaries. We can then perform the complete clipping by clipping against each of the four boundaries in turn.

To clip at a boundary, we step through the drawing instructions. As we consider each new endpoint, we decide whether it belongs to a line which crosses the boundary. If it does, the point of intersection is determined and is passed on to the next routine. Then each point is examined to see whether it lies within the boundary. If so, it is also passed to the next routine. In this procedure, all line-segment endpoints lying within the boundary and all points where lines intersect the boundary are passed on, while points lying outside the boundary are filtered out. (See Figure 6-13.)

We can think of the process as clipping the entire figure against each window boundary before moving on to the next boundary. (See Figure 6-14.) However, since our clipping process steps sequentially through the figure-drawing instructions, it is possible to begin clipping on a second boundary before the clipping of the entire figure against the first boundary is completed. In fact, each point may be run through all four clipping routines and entered into the display file before the next point is considered.

Algorithms for clipping a figure against each of the four window boundaries are given below. They all follow the same outline. They first check to see if the new point is the first point of a polygon, and if so, they save it. This is used in closing polygons and is discussed below. They examine the new point and last point to see whether the line segment with these endpoints crosses the boundary. The algorithms are called for each new point. We can picture this as pen movements. We start with the pen at some location (the last point, which is stored for each clipping boundary in the arrays XS and YS) and ask to move it to some new position (the new point, X and Y). The clipping routine examines this path to see if it encounters the clipping boundary. If it does,

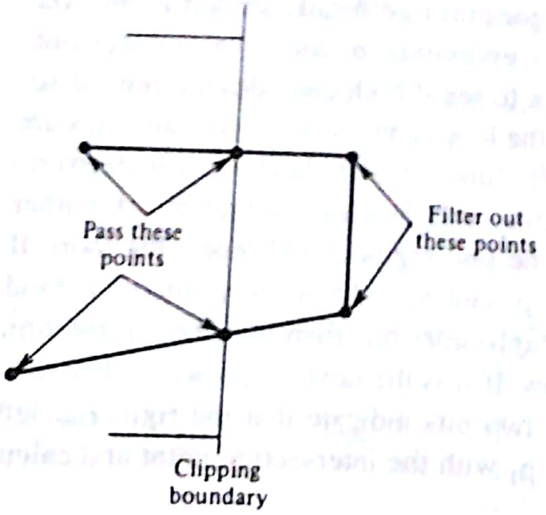

FIGURE 6-13
Clipping against an edge.

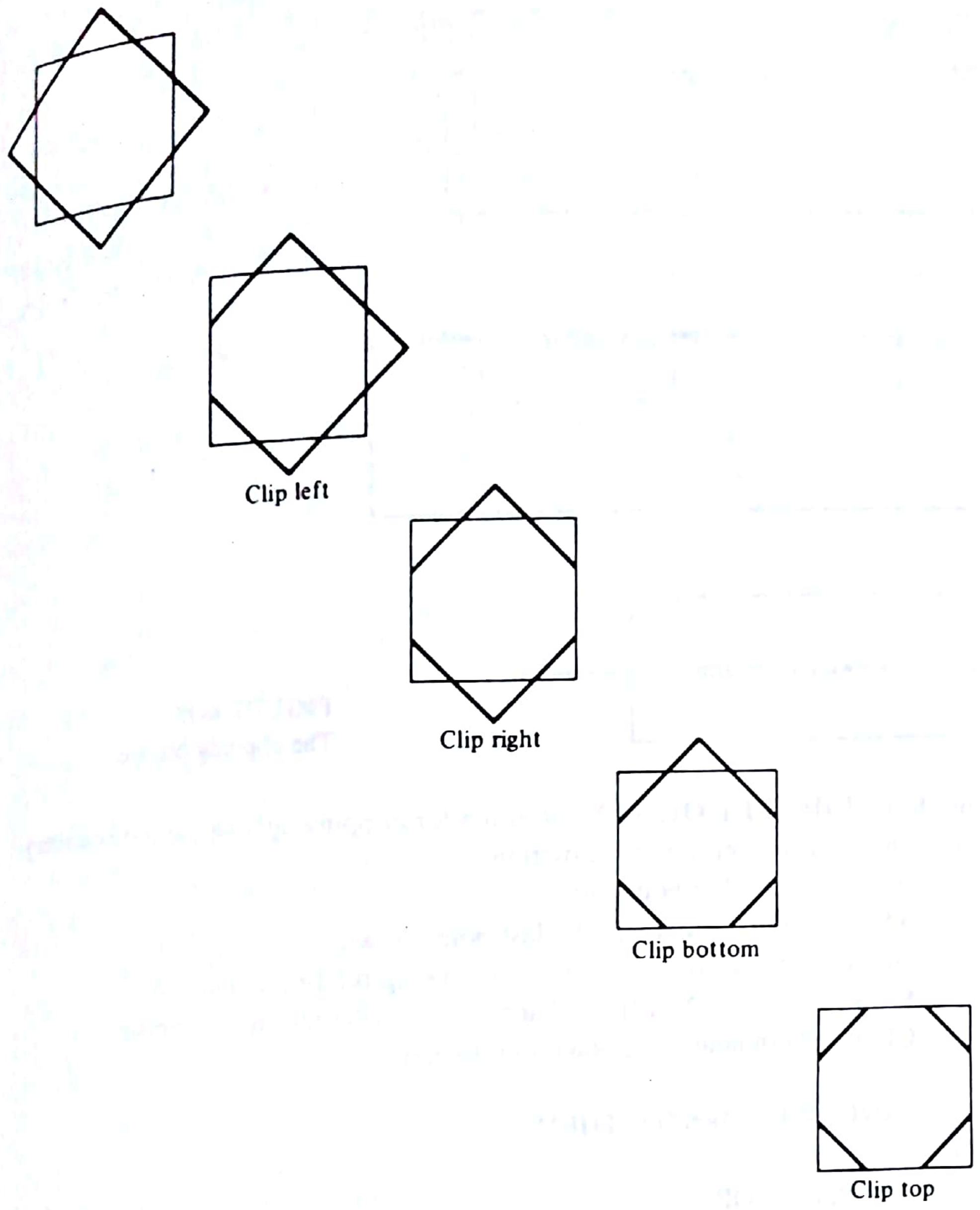

FIGURE 6-14
Clipping against all four window boundaries.

the pen is moved only to the boundary; a new command, corresponding to the clipped point, is entered. If the side is drawn from outside the window to inside the window or the command is for character drawing, then we introduce a MOVE command; otherwise, the command is the same as the original. This means that if our figure passes outside of the window boundary, the pen will move along the boundary to the point where the figure reenters the window region. The algorithms update the last point to be the current point and check the current point to see whether it is inside the window. If so, this instruction is also entered. When we say a command is "entered," we mean that it is passed on to the next routine. For the first three clipping algorithms, the next routine is the algorithm for clipping along the next boundary. The last clipping algorithm actually enters commands into the display file. (See Figure 6-15.)

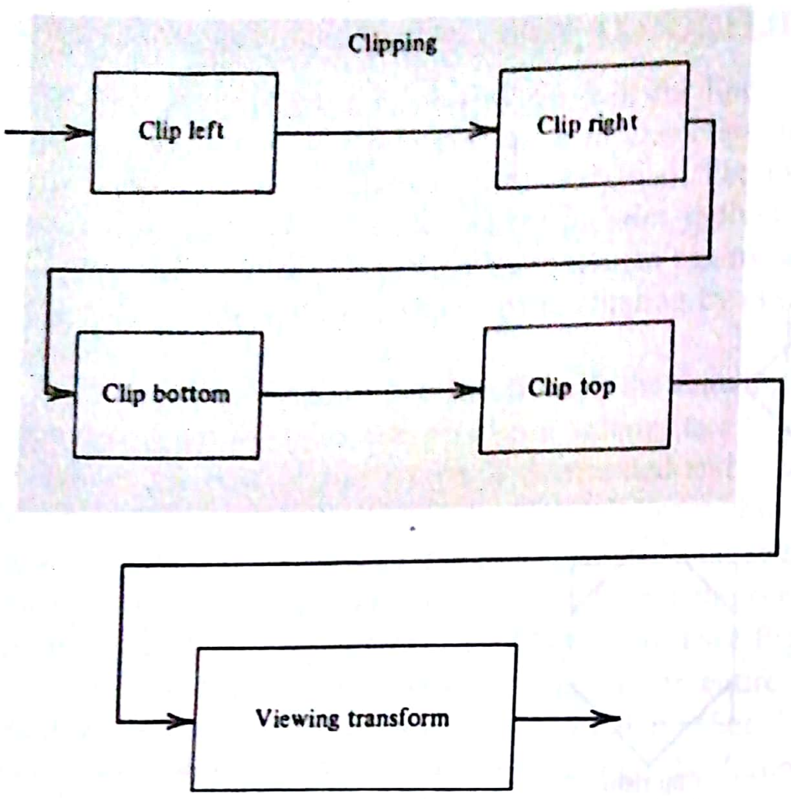

FIGURE 6-15
The clipping process.

```
6.6 Algorithm CLIP-LEFT(OP, X, Y) Routine for clipping against the left boundary
Arguments   OP, X, Y a display-file instruction
Global      WXL window left boundary
            XS, YS arrays containing the last point drawn
            NEEDFIRST array of indicators for saving the first command
            FIRSTOP, FIRSTX, FIRSTY arrays for saving the first command
            CLOSING indicates the stage in polygon
BEGIN
  IF PFLAG AND NEEDFIRST[1] THEN
    BEGIN
      FIRSTOP[1] ← OP;
      FIRSTX[1] ← X;
      FIRSTY[1] ← Y;
      NEEDFIRST[1] ← FALSE;
    END
  Case of drawing from outside in
  ELSE IF X ≥ WXL AND XS[1] < WXL THEN
      CLIP-RIGHT(1, WXL, (Y − YS[1]) * (WXL − X) / (X − XS[1]) + Y)
    Case of drawing from inside out
    ELSE IF X ≤ WXL AND XS[1] > WXL THEN
        IF OP > 0 THEN
          CLIP-RIGHT(OP, WXL, (Y − YS[1]) * (WXL − X) /
          (X − XS[1]) + Y)
        ELSE
          CLIP-RIGHT(1, WXL, (Y − YS[1]) * (WXL − X) /
          (X − XS[1]) + Y);
  Remember point to serve as one of the endpoints of next line segment
  XS[1] ← X;
  YS[1] ← Y;
```

```
    Case of point inside
    IF X ≥ WXL AND CLOSING ≠ 1 THEN CLIP-RIGHT(OP, X, Y);
    RETURN;
END;
```

The calculation which occurs inside the calls to CLIP-RIGHT in the above routine is the determination of the y coordinate of the point where the line intersects the window boundary. The x coordinate of this point is the window boundary position.

6.7 Algorithm CLIP-RIGHT(OP, X, Y) Routine for clipping against the right boundary

Arguments	OP, X, Y a display-file instruction
Global	WXH window right boundary
	XS, YS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
    IF PFLAG AND NEEDFIRST[2] THEN
        BEGIN
            FIRSTOP[2] ← OP;
            FIRSTX[2] ← X;
            FIRSTY[2] ← Y;
            NEEDFIRST[2] ← FALSE;
        END
    ELSE IF X ≤ WXH AND XS[2] > WXH THEN
            CLIP-BOTTOM(1, WXH, (Y − YS[2]) * (WXH − X) / (X − XS[2]) + Y)
        ELSE IF X ≥ WXH AND XS[2] < WXH THEN
                IF OP > 0 THEN
                    CLIP-BOTTOM(OP, WXH, (Y − YS[2]) * (WXH − X) /
                    (X − XS[2]) + Y)
                ELSE
                    CLIP-BOTTOM(1, WXH, (Y − YS[2]) * (WXH − X) /
                    (X − XS[2]) + Y);
    XS[2] ← X;
    YS[2] ← Y;
    IF X ≤ WXH AND CLOSING ≠ 2 THEN CLIP-BOTTOM(OP, X, Y);
    RETURN;
END;
```

6.8 Algorithm CLIP-BOTTOM(OP, X, Y) Routine for clipping against the lower boundary

Arguments	OP, X, Y a display-file instruction
Global	WYL window lower boundary
	XS, YS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
    IF PFLAG AND NEEDFIRST[3] THEN
```

```
        BEGIN
            FIRSTOP[3] ← OP;
            FIRSTX[3] ← X;
            FIRSTY[3] ← Y;
            NEEDFIRST[3] ← FALSE;
        END
    ELSE IF Y ≥ WYL AND YS[3] < WYL THEN
            CLIP-TOP(1, (X − XS[3]) * (WYL − Y) / (Y − YS[3]) + X, WYL)
        ELSE IF Y ≤ WYL AND YS[3] > WYL THEN
                IF OP > 0 THEN
                    CLIP-TOP(OP, (X − XS[3]) * (WYL − Y) /
                    (Y − YS[3]) + X, WYL)
                ELSE
                    CLIP-TOP(1, (X − XS[3]) * (WYL − Y) /
                    (Y − YS[3]) + X, WYL);
    XS[3] ← X;
    YS[3] ← Y;
    IF Y ≥ WYL AND CLOSING ≠ 3 THEN CLIP-TOP(OP, X, Y);
    RETURN;
END;
```

6.9 Algorithm CLIP-TOP(OP, X, Y) Routine for clipping against the upper boundary

Arguments	OP, X, Y a display-file instruction
Global	WYH window upper boundary
	XS, YS arrays containing the last point drawn
	NEEDFIRST array of indicators for saving the first command
	FIRSTOP, FIRSTX, FIRSTY arrays for saving the first command
	CLOSING indicates the stage in polygon

```
BEGIN
    IF PFLAG AND NEEDFIRST[4] THEN
        BEGIN
            FIRSTOP[4] ← OP;
            FIRSTX[4] ← X;
            FIRSTY[4] ← Y;
            NEEDFIRST[4] ← FALSE;
        END
    ELSE IF Y ≤ WYH AND YS[4] > WYH THEN
            SAVE-CLIPPED-POINT(1, (X − XS[4]) * (WYH − Y) / (Y − YS[4]) +
            X, WYH)
        ELSE IF Y ≥ WYH AND YS[4] < WYH THEN
                IF OP > 0 THEN
                    SAVE-CLIPPED-POINT(OP, (X − XS[4]) * (WYH − Y) /
                    (Y − YS[4]) + X, WYH)
                ELSE
                    SAVE-CLIPPED-POINT(1, (X − XS[4]) * (WYH − Y) /
                    (Y − YS[4]) + X, WYH);
    XS[4] ← X;
    YS[4] ← Y;
    IF Y ≤ WYH AND CLOSING ≠ 4 THEN SAVE-CLIPPED-POINT(OP, X, Y);
    RETURN;
END;
```

The SAVE-CLIPPED-POINT routine is used to enter the commands into the display file. It will be described below.

Let's walk through an example to see how these routines operate. Consider the window and sequence of line segments shown in Figure 6-16. Suppose we start with the pen at (2, 2) and attempt to draw the lines to (4,2), (4, 4), (2, 4), and back to (2, 2). The XS and YS array values will be initialized to the current position XS[i] = 2 and YS[i] = 2. The CLIP-LEFT routine is entered with the point X = 4, Y = 2, and the work begins. The CLIP-LEFT routine will compare the segment from (2, 2) to (4, 2) against the window boundary WXL = 1. The segment will not require clipping at this boundary. XS[1] is set to 4 and YS[1] is set again at 2. This point is then passed to the CLIP-RIGHT routine. It compares the segment against the window boundary WXH = 3. The third IF statement in this algorithm discovers that clipping is required and passes the point X = 3, Y = 2 to the CLIP-BOTTOM routine. The XS[2] and YS[2] values are set to 4 and 2, respectively. The CLIP-BOTTOM and CLIP-TOP routines do not have to clip but just pass along the point and remember the (3, 2) position in their XS, YS array elements. The command to draw the line from (2, 2) to (3, 2) is entered into the display file by SAVE-CLIPPED-POINT. The next line segment is seen by CLIP-LEFT as going from (4, 2) to (4, 4). Since this does not cross the left boundary, the point (4, 4) is passed along and remembered. The CLIP-RIGHT routine will also consider the line from (4, 2) to (4, 4). Since both points are outside the right window boundary, this routine will not pass along the point to CLIP-BOTTOM. It will only remember the point (4, 4) as its current pen position. The next point is (2, 4). Again the CLIP-LEFT routine will remember the (2, 4) position in the XS[1], YS[1] array elements and pass the point to clip right. In CLIP-RIGHT the second IF statement will realize that this line crosses from outside the right boundary to inside. It will send a command to MOVE to the point (3, 4) to the CLIP-BOTTOM routine. It remembers the point (2, 4) and finally passes a LINE command to this point. The first of these two calls to CLIP-BOTTOM is for the move from (3, 2) to (3, 4). This does not cross the bottom boundary, so the routine remembers the current position (3, 4) and passes the move command to CLIP-TOP. The CLIP-TOP routine will clip this command at (3, 3) and enter this move into the display file. It will set its current position to (3, 4). Now back to the second call to CLIP-BOTTOM by CLIP-RIGHT. This is a line

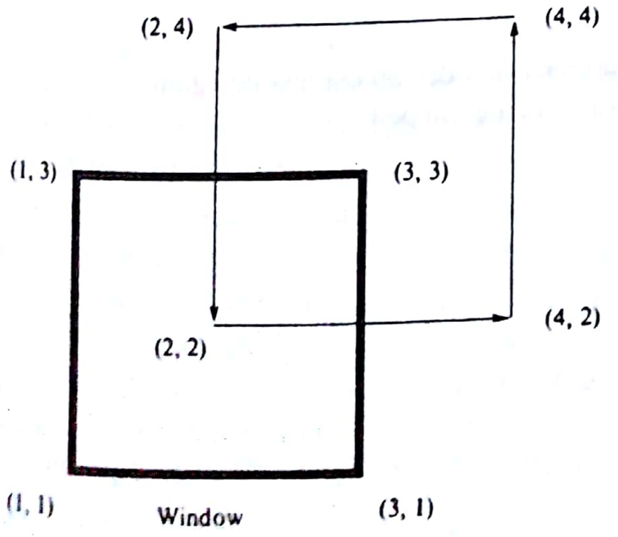

FIGURE 6-16
A clipping example.

command to the point (2, 4). CLIP-BOTTOM will remember this point and pass the command to CLIP-TOP. CLIP-TOP will also remember the point, but will not pass the command any further because the line from (3, 4) to (2, 4) is above the window. Finally, we give the point (2, 2) to the CLIP-LEFT routine. It passes the command to CLIP-RIGHT, which passes it to CLIP-BOTTOM, which passes it in turn to CLIP-TOP. The CLIP-TOP routine sees the segment from (2, 4) to (2, 2). The second IF statement in the routine forwards a command to MOVE to the point (2, 3) to the display file. The final IF statement sends the command to draw a line to the point (2, 2). The net result has been a line from (2, 2) to (3, 2), a move to (3, 3), a move to (2, 3), and a line back to (2, 2).

THE CLIPPING OF POLYGONS

We would like our clipping routine to handle polygons as well as line segments. What will happen if a polygon crosses our window boundary? Our clipping routine will remove some of the polygon's sides, and it will insert a move command instead of a line command along the window boundary. This change in the number of sides in the polygon must be reflected in our initial polygon-drawing operation code. We will consider the move command to be an invisible side since it occupies one instruction and moves the pen just as a line-drawing command. Because of the change in the number of sides, we cannot know what polygon command to enter (if any at all) until the entire polygon has been clipped. We will therefore not enter polygon instructions into the display file immediately. Instead, we shall store them in a temporary area. When all sides have been clipped, we can count how many sides remain, form an appropriate polygon command, and then enter this new command (along with the instructions that were saved for the sides) into the display file. The instructions which survive the clipping routines are therefore treated in two different ways. Instructions which do not belong to a polygon are given a viewing transformation and placed in the display file, while instructions which are part of a polygon are placed in a temporary storage buffer. This decision is made in the algorithm SAVE-CLIPPED-POINT based on a flag PFLAG which indicates polygon processing. The algorithm also keeps track of how many polygon sides have been saved. (See Figure 6-17.)

6.10 Algorithm SAVE-CLIPPED-POINT(OP, X, Y) Saves clipped polygons in the T buffer and sends lines and characters to the display file

Arguments	OP, X, Y a display-file instruction
Global	COUNT-OUT a counter of number of sides on clipped polygon
	PFLAG indicates if a polygon is being clipped

```
BEGIN
    IF PFLAG THEN
        BEGIN
            COUNT-OUT ← COUNT-OUT + 1;
            PUT-IN-T(OP, X, Y, COUNT-OUT);
        END
    ELSE VIEWING-TRANSFORM(OP, X, Y);
    RETURN;
END;
```

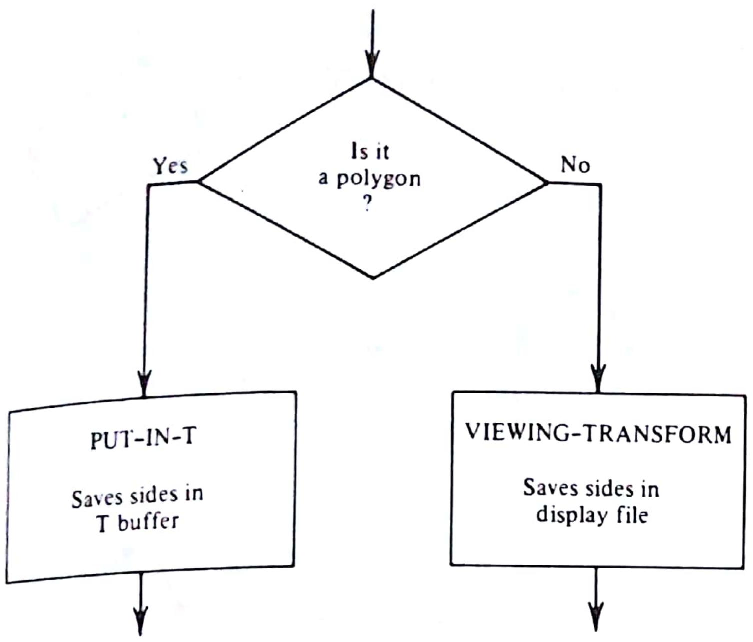

FIGURE 6-17
SAVE-CLIPPED-POINT.

We have called the arrays providing temporary storage for polygons IT, XT, and YT. They must be large enough to hold the maximum number of polygon sides. The above algorithm uses the routine PUT-IN-T to save instructions in these arrays. The algorithm for PUT-IN-T is as follows:

6.11 Algorithm PUT-IN-T(OP, X, Y, INDEX) Save an instruction in the T buffer

```
Arguments   OP, X, Y the instruction to be stored
            INDEX the position at which to store it
Global      IT, XT, YT arrays for temporary storage of polygon sides
BEGIN
   IT[INDEX] ← OP;
   XT[INDEX] ← X;
   YT[INDEX] ← Y;
   RETURN;
END;
```

For polygons, we have an overall clipping routine which counts how many sides of the original polygon have been considered. When all sides are considered, we need to make sure that the polygon is closed. The closing problem is illustrated in Figure 6-18. Clipping against the left boundary results in a starting point above the bottom boundary and an ending point which is below this boundary. Now clipping this sequence of points against the bottom boundary results in a polygon which is not closed because there is no command to move across the boundary, and intersection points are only calculated when the boundary is crossed.

To fix this problem we require each clipping stage to close its version of the polygon. (See Figure 6-19.) To do this each clipping stage stores the first instruction it receives from the polygon. This is done by the first IF statement in each of algorithms 6.6 through 6.9. The NEEDFIRST flag is used to tell if the instruction is the first.

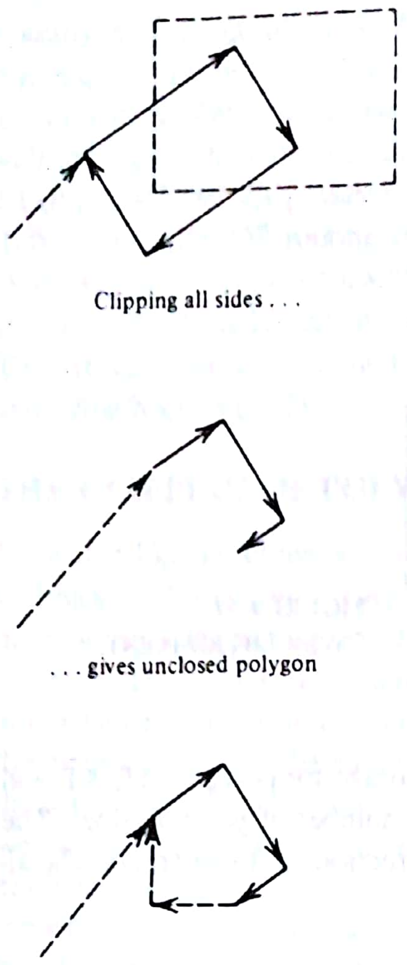

FIGURE 6-18
Closing the polygon.

After all commands have been sent we set the CLOSING variable and output the saved instruction to each clipping stage. This causes each stage to check the edge between the last point of the polygon and the first point for intersection with the clipping boundary. If it does intersect, the intersection point is entered which completes the polygon. For this final check we want to enter the intersection point, but we do not want to enter the first point of the polygon a second time. This is what the CLOSING variable is for. It prevents the reentry of the first point when closing the polygon. After closing the polygon, the final number of sides of the clipped polygon is checked to see whether it is greater than 3. If it is less than 3, then the polygon has collapsed or has been clipped away and no entry at all should be made. If the new polygon has an acceptable number of sides, then we must update the polygon command to reflect this. We must also enter the x and y coordinates of this command so that we begin drawing the polygon at the point where drawing of the sides will terminate. All of this is done by the algorithm CLIP-POLYGON-EDGE.

6.12 Algorithm CLIP-POLYGON-EDGE(OP, X, Y) Close and enter a clipped polygon into the display file
Arguments OP, X, Y a display-file instruction

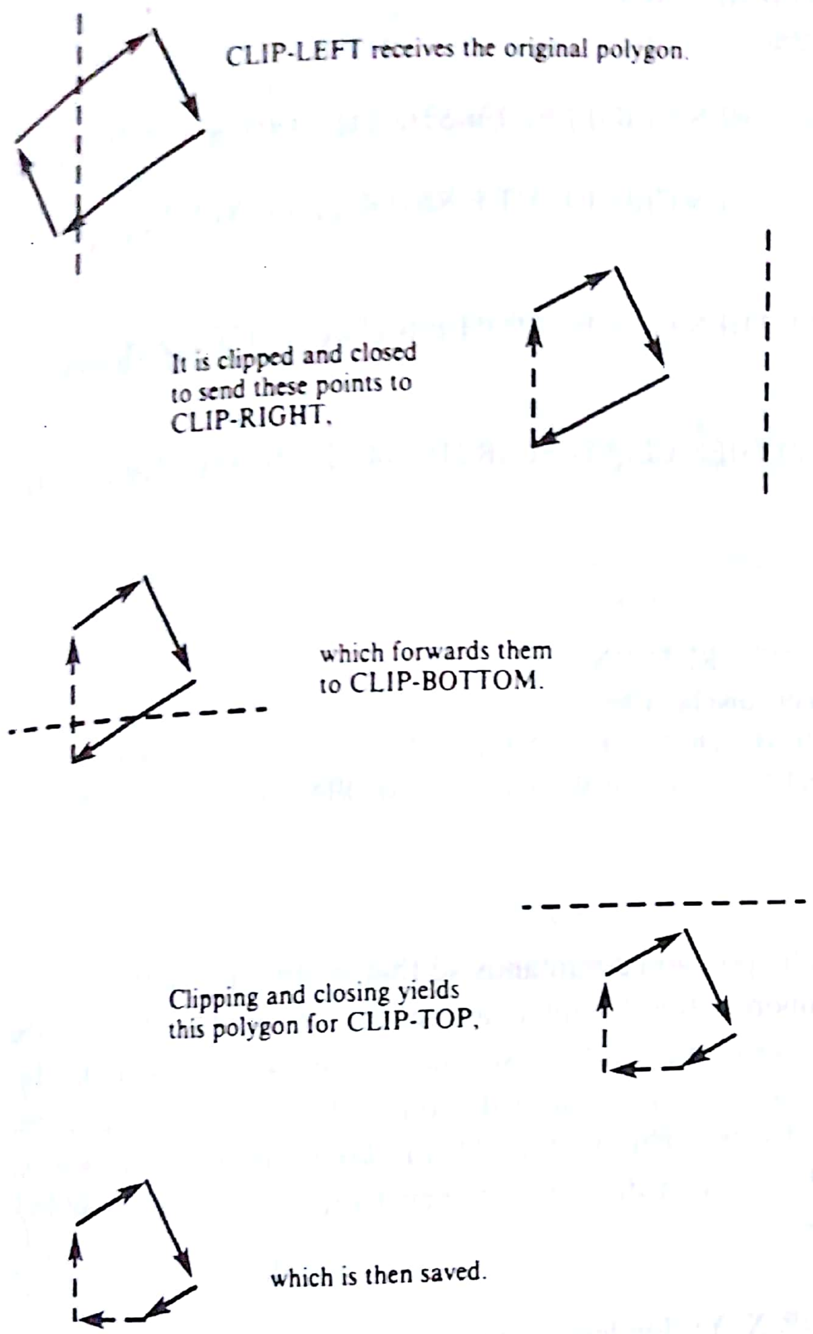
CLIP-LEFT receives the original polygon.
It is clipped and closed to send these points to CLIP-RIGHT,
which forwards them to CLIP-BOTTOM.
Clipping and closing yields this polygon for CLIP-TOP,
which is then saved.

FIGURE 6-19
Each clipping stage closes its polygon.

```
Global   PFLAG indicates that a polygon is being drawn
         COUNT-IN the number of sides remaining to be processed
         COUNT-OUT the number of sides to be entered in the display file
         IT, XT, YT temporary storage arrays for a polygon
         NEEDFIRST array of indicators for saving the first command
         FIRSTOP, FIRSTX, FIRSTY arrays for saving the first command
         CLOSING indicates the stage in polygon
Local    I for stepping through the polygon sides

BEGIN
   COUNT-IN ← COUNT-IN − 1;
   CLIP-LEFT(OP, X, Y);
```

```
        IF COUNT-IN ≠ 0 THEN RETURN;
        close the clipped polygon
        CLOSING ← 1;
        IF NOT NEEDFIRST[1] THEN CLIP-LEFT(FIRSTOP[1], FIRSTX[1], FIRSTY[1]);
        CLOSING ← 2;
        IF NOT NEEDFIRST[2] THEN CLIP-RIGHT(FIRSTOP[2], FIRSTX[2],
        FIRSTY[2]);
        CLOSING ← 3;
        IF NOT NEEDFIRST[3] THEN CLIP-BOTTOM(FIRSTOP[3], FIRSTX[3],
        FIRSTY[3]);
        CLOSING ← 4;
        IF NOT NEEDFIRST[4] THEN CLIP-TOP(FIRSTOP[4], FIRSTX[4], FIRSTY[4]);
        CLOSING ← 0;

        PFLAG ← FALSE;

        IF COUNT-OUT < 3 THEN RETURN;
        enter the polygon into the display file
        VIEWING-TRANSFORM(COUNT-OUT, XT[COUNT-OUT], YT[COUNT-OUT]);
        FOR I = 1 TO COUNT-OUT DO VIEWING-TRANSFORM(IT[I], XT[I], YT[I]);
        RETURN;
    END;
```

We must catch and handle polygon commands so that when a polygon is discovered, it is entered into the temporary file. Counters are set for the number of sides to be expected and the number of sides of the result. Last-point variables for each of the clipping routines are initialized, and a flag is set so that future calls to the clipping routine will be recognized as polygon sides. This is done by the algorithm CLIP. This is the top-level clipping routine. Basically, it decides between handling polygons and handling other graphics primitives.

6.13 Algorithm CLIP(OP, X, Y) Top-level clipping routine

```
Arguments  OP, X, Y the instruction being clipped
Global     PFLAG indicates that a polygon is being processed
           COUNT-IN number of polygon sides still to be input
           COUNT-OUT number of clipped polygon sides stored
           XS, YS arrays for saving the last point drawn
Local      I for initializing the four clipping routines
BEGIN
    IF PFLAG THEN CLIP-POLYGON-EDGE(OP, X, Y)
    ELSE IF OP > 2 THEN
        BEGIN
            PFLAG ← TRUE;
            COUNT-IN ← OP;
            COUNT-OUT ← 0;
            FOR I = 1 TO 4 DO
                BEGIN
                    XS[I] ← X;
                YS[I] ← Y;
            END;
        END
        ELSE CLIP-LEFT(OP, X, Y);
    RETURN;
END;
```

ADDING CLIPPING TO THE SYSTEM

The CLIP algorithm will clip, transform, and save drawing instructions in the display file. We have only to include it as part of the display-file instruction storage process. To do this, we modify our DISPLAY-FILE-ENTER routine. This routine will now get the current object-space pen position and place it on the display file through the clipping routine, transforming it to image space dimensions in the process. (See Figure 6-20.)

6.14 Algorithm DISPLAY-FILE-ENTER(OP) (Modification of algorithm 2.23) Combine operation and position to form an instruction and save it in the display file

```
Argument    OP the operation to be entered
Global      DF-PEN-X, DF-PEN-Y the current pen position
BEGIN
    IF OP < 1 AND OP > −32 THEN PUT-POINT(OP, 0, 0)
    ELSE CLIP(OP, DF-PEN-X, DF-PEN-Y);
    RETURN;
END;
```

We would also like to have an initialization routine which sets the boundaries of the viewport and window to be the same as our normalized screen coordinates, that is, from 0 to 1 in both the x and y directions. This makes the window and viewport transformation transparent for the user who does not wish to use it.

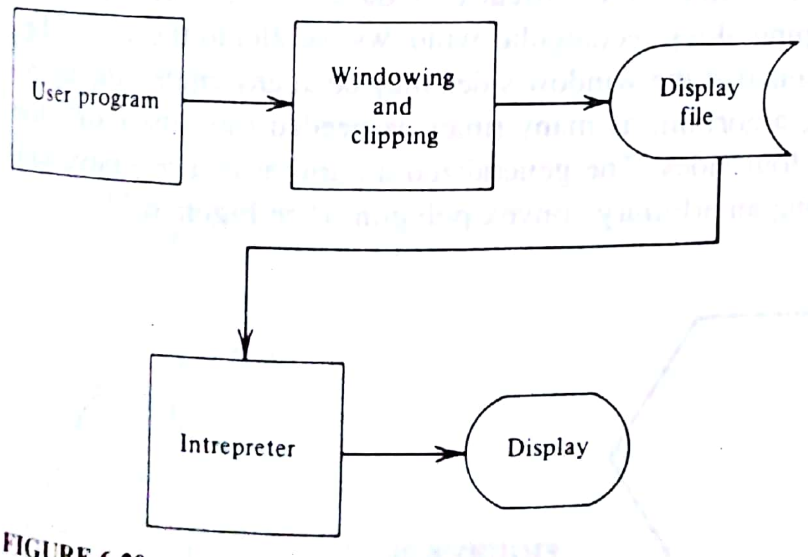

FIGURE 6-20
Adding windowing to the system.

6.15 Algorithm INITIALIZE-6

```
Global      PFLAG polygon processing flag
            XS, YS position of the pens confined by clipping boundaries
            NEEDFIRST array of indicators for saving the first command
            CLOSING indicates the stage in polygon
Local       I for initialization of the four clipping routines
BEGIN
   INITIALIZE-5
   SET-VIEWPORT(0.0, 1.0, 0.0, 1.0);
   SET-WINDOW(0.0, 1.0, 0.0, 1.0);
   NEW-VIEW-2;
   FOR I = 1 to 4 DO
      BEGIN
         NEEDFIRST[I] ← FALSE;
         XS[I] ← 0;
         YS[I] ← 0;
      END;
   CLOSING ← 0;
   PFLAG ← FALSE;
   RETURN;
END;
```

GENERALIZED CLIPPING

We have used four separate clipping routines, one for each boundary. But these routines are almost identical. They differ only in their test for determining whether a point is inside or outside the boundary. It is possible to write these routines in a more general form, so that they will be exactly identical and information about the boundary is passed to the routines through their parameters. In a recursive language this would mean that instead of having four separate routines, only one routine would be needed. This routine would be entered four times (recursively), each time with a different boundary specified by its parameters. Furthermore, the routine can be generalized to clip along any line (not just horizontal and vertical boundaries). This form of the algorithm is not limited to clipping along rectangular windows parallel to the axis. Clipping along arbitrary lines means that the window sides may be at any angle, and by recursively calling the clipping algorithm as many times as needed (not just four), the window can have more than four sides. The generalized algorithm in a recursive language can be used to clip along an arbitrary convex polygon. (See Figure 6-21.)

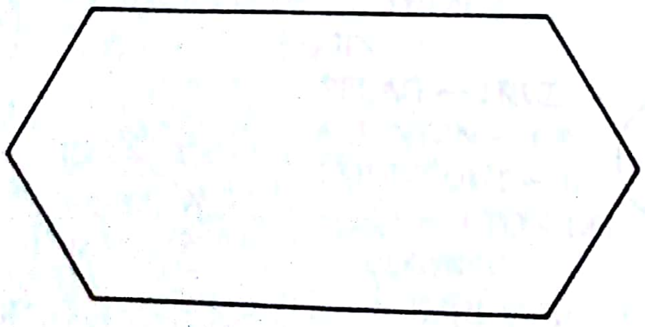

FIGURE 6-21
A window with six clipping boundaries.

POSITION RELATIVE TO AN ARBITRARY LINE

A line divides a plane into two half planes. Let us consider briefly a half-plane test to determine on which side of a line a point lies. Suppose we have a line specified by the points (x_1, y_1) and (x_2, y_2). Recall from Chapter 1 that if a third point (x, y) is on the line, then

$$(x - x_2)(y_1 - y_2) = (y - y_2)(x_1 - x_2) \tag{6.3}$$

If the left-hand side does not equal the right-hand side, then the point is not on the line. If the left expression is greater than the right

$$(x - x_2)(y_1 - y_2) > (y - y_2)(x_1 - x_2) \tag{6.4}$$

then the point lies on one side; if it is less

$$(x - x_2)(y_1 - y_2) < (y - y_2)(x_1 - x_2) \tag{6.5}$$

then the point lies on the other side. (See Figure 6-22.) The choice of which of the two sides corresponds to the "greater than" case depends upon which of the two line points is named (x_1, y_1).

As an example, consider the line containing points $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (4, 5)$. The point $(x, y) = (3, 4)$ is on the line because

$$(3 - 4)(2 - 5) = (4 - 5)(1 - 4) \tag{6.6}$$

The point (2, 5) is in the half plane above and to the left of the line

$$(2 - 4)(2 - 5) > (5 - 5)(1 - 4) \tag{6.7}$$

For this identification of the points (x_1, y_1) and (x_2, y_2), all points (x, y) which result in the left-hand side greater than the right-hand side are above and left of the line.

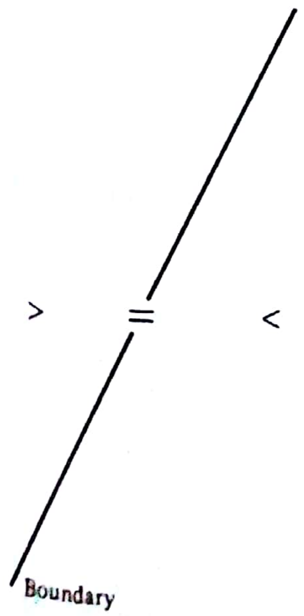

FIGURE 6-22
Deciding on which side of a line a point lies.

The point (4, 3) is below and right. As we can easily verify, it results in the "less than" relation

$$(4 - 4)(2 - 5) < (3 - 5)(1 - 4) \tag{6.8}$$

This test may be used in a clipping algorithm to determine if a point lies inside or outside an arbitrary boundary line.

Other forms of the test are possible. We saw that another form of the line equation is rx + sy + t = 0 (Equation 1.6). Substituting the (x, y) coordinates of a point not on the line into the left-hand side of this expression will also give a positive number for one side of the line and a negative number for the other side. Furthermore, we found in Equation 1.30 that for proper normalization of r, s, and t, the magnitude of this expression is the distance of the point from the line.

MULTIPLE WINDOWING

Some systems allow the use of *multiple windowing*; that is, a first image is created by one or more window transformations on the object. Then, windows are applied to this first image to create a second image. Further windowing transformations may be done until the desired picture is created. Every application of a window transformation allows the user to slice up a portion of the picture and reposition it on the screen. Thus multiple windowing gives the user freedom to rearrange components of the picture. The same effect may be achieved, however, by applying a number of single-window transformations to the object. (See Figure 6-23.)

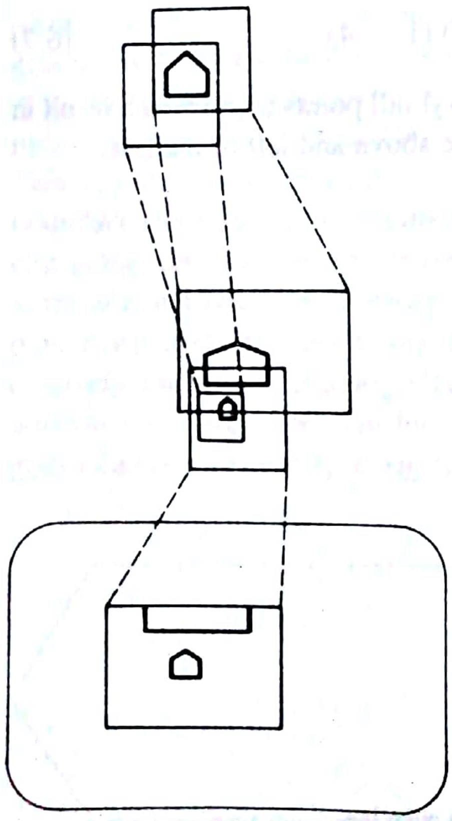

FIGURE 6-23
Multiple windowing.

AN APPLICATION

One important application of computer graphics is the design of integrated circuits. These miniature components may be produced by photographic techniques from large drawings of their circuitry. The drawings describe the areas of the conducting, semiconducting, and insulating materials. Certain patterns or geometries of these materials produce the individual diodes and transistors. A single integrated circuit may have tens of thousands of transistors. Producing a correct drawing for such a complex structure can be quite a task, and computer graphics is an invaluable aid. There is usually a great deal of regularity and repetition within the circuit structure. Our ability to reproduce a pattern by repeated calls upon a single image-generating subroutine is helpful here. Furthermore, the graphics program which draws the circuit may be part of a larger program which provides some checks of the circuit's correctness.

The full drawing of the circuit may be 1 to 2 meters square. If this is reduced to the size of the designer's terminal, the detail will be too fine and too complex to be useful (if it can be displayed at all). What is needed is a clipping window which displays only the portion of the circuit which the designer is currently working on. A call on our SET-WINDOW routine will provide this. If the designer should need to look at two separate portions of the circuit at the same time, the display surface may be separated into two viewports and the portions of the circuit selected by two windows. The designer could specify one window-viewport pair, open a display-file segment, draw the circuit (clipping away all but the portion of interest), close the segment, and then repeat the process for the second portion of the circuit to be displayed. (See Figure 6-24.)

```
SET-WINDOW(20.0, 30.0, 40.0, 50.0);
SET-VIEWPORT(0.2, 0.8, 0.6, 1.0);
CREATE-SEGMENT(1);
DRAW-CIRCUIT;
CLOSE-SEGMENT;
SET-WINDOW(20.0, 30.0, 10.0, 20.0);
SET-VIEWPORT(0.2, 0.8, 0.0, 0.4);
```

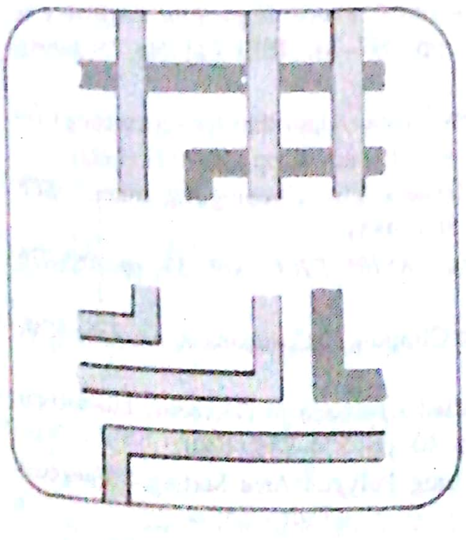

FIGURE 6-24
Use of windows and viewports to examine two portions of an integrated circuit.

```
CREATE-SEGMENT(2);
DRAW-CIRCUIT;
CLOSE-SEGMENT;
```

FURTHER READING

Some other clipping methods for lines may be found in [CYR78], [LIA83], [LIA84], and [ROG85]. The Cohen-Sutherland algorithm is also presented in [FOL82]. An implementation of the Cohen-Sutherland algorithm is given in [WHI86]. The Sutherland-Hodgman algorithm is presented in [SUT74]. The Sutherland-Hodgman algorithm requires a convex clipping region; it also results in a single polygon, even when a division into several independent polygons might seem more natural. An alternative clipping method which overcomes these problems is given by [WEI77]. It is possible to find the intersection between a line segment and a clipping boundary by repeatedly dividing the segment into halves. This is particularly useful on systems which do not support fast division. A clipping method based on this technique is given in [SPR68]. A theoretical discussion and algorithms for the general problem of deciding if a point is inside, outside, or on the boundary of a shape are presented in [LEE77] and [TIL80]. The problem of finding the intersection of two polygons is also considered in [ORO82], [YAM72], and [WEI80]. A formal description of transformations and clipping in a hierarchical picture structure is given in [MAL78].

[CYR78] Cyrus, M., Beck, J., "Generalized Two and Three-Dimensional Clipping," *Computers and Graphics*, vol. 3, no. 1, pp. 23–28 (1978).

[FOL82] Foley, J. D., Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Mass., pp. 146–149 (1982).

[LEE77] Lee, D. T., Preparata, F. P., "Location of a Point in a Planar Subdivision and Its Applications," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 594–606 (1977).

[LIA83] Liang, Y., Barsky, B. A., "An Analysis and Algorithm for Polygon Clipping," *Communications of the ACM*, vol. 26, no. 11, pp. 868–877 (1983).

[LIA84] Liang, Y., Barsky, B. A., "A New Concept and Method for Line Clipping," *ACM Transactions on Graphics*, vol. 3, no. 1, pp. 1–22 (1984).

[MAL78] Mallgren, W. R., Shaw, A. C., "Graphical Transformations and Hierarchic Picture Structures," *Computer Graphics and Image Processing*, vol. 8, no. 2, pp. 237–258 (1978).

[NEW75] Newman, W. M., "Instance Rectangles and Picture Structure," *Proceedings of the Conference on Computer Graphics, Pattern Recognition, & Data Structures*, pp. 297–301, IEEE Cat. No. 75CH0981-1c (1975).

[ORO82] O'Rourke, J., Chen, C., Olson, T., Naddor, D., "A New Linear Algorithm for Intersecting Convex Polygons," *Computer Graphics and Image Processing*, vol. 19, no. 4, pp. 384–391 (1982).

[ROG85] Rogers, D. F., Rybak, L. M., "A Note on an Efficient General Line-Clipping Algorithm," *IEEE Computer Graphics and Applications*, vol. 5, no. 1, pp. 82–86 (1985).

[SPR68] Sproull, R. F., Sutherland, I. E., "A Clipping Divider," *AFIPS FJCC*, vol. 33, pp. 765–776 (1968).

[SUT74] Sutherland, I. E., Hodgman, G. W., "Reentrant Polygon Clipping," *Communications of the ACM*, vol. 17, no. 1, pp. 32–42 (1974).

[TIL80] Tilove, R. B., "Set Membership Classifications: A Unified Approach to Geometric Intersection Problems," *IEEE Transactions on Computers*, vol. C-29, no. 10, pp. 874–883 (1980).

[WEI77] Weiler, K., Atherton, P., "Hidden Surface Removal Using Polygon Area Sorting," *Computer Graphics*, vol. 11, no. 2, pp. 214–222 (1977).