CHAPTER 5

Two-Dimensional Viewing and Clipping

Much like what we see in real life through a small window on the wall or the viewfinder of a camera, a computer-generated image often depicts a partial view of a large scene. Objects are placed into the scene by modeling transformations to a master coordinate system, commonly referred to as the *world coordinate system* (WCS). A rectangular *window* with its edges parallel to the axes of the WCS is used to select the portion of the scene for which an image is to be generated (see Fig. 5-1). Sometimes an additional coordinate system called the *viewing coordinate system* is introduced to simulate the effect of moving and/or tilting the camera.

On the other hand, an image representing a view often becomes part of a larger image, like a photo on an album page, which models a computer monitor's display area. Since album pages vary and monitor sizes differ from one system to another, we want to introduce a device-independent tool to describe the display area. This tool is called the *normalized device coordinate system* (NDCS) in which a unit (1 × 1) square whose lower left corner is at the origin of the coordinate system defines the display area of a virtual display device. A rectangular *viewport* with its edges parallel to the axes of the NDCS is used to specify a sub-region of the display area that embodies the image.

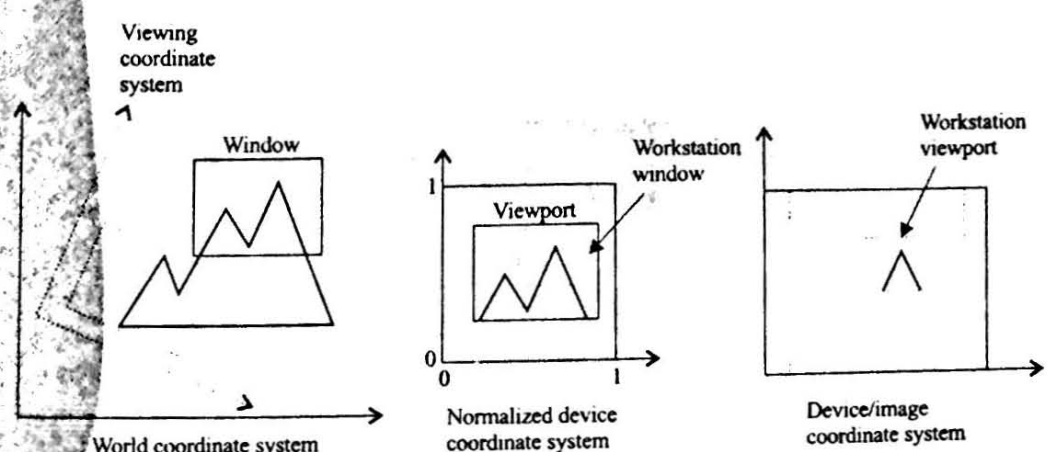


Fig. 5-1 Viewing transformation.

The process that converts object coordinates in WCS to normalized device coordinates is called *window-to-viewport mapping* or *normalization transformation*, which is the subject of Sect. 5.1. The process that maps normalized device coordinates to discrete device/image coordinates is called *workstation transformation*, which is essentially a second window-to-viewport mapping, with a workstation window in the normalized device coordinate system and a workstation viewport in the device coordinate system. Collectively, these two coordinate-mapping operations are referred to as *viewing transformation*.

Workstation transformation is dependent on the resolution of the display device/frame buffer. When the whole display area of the virtual device is mapped to a physical device that does not have a 1/1 aspect ratio, it may be mapped to a square sub-region (see Fig. 5-1) so as to avoid introducing unwanted geometric distortion.

Along with the convenience and flexibility of using a window to specify a localized view comes the need for *clipping*, since objects in the scene may be completely inside the window, completely outside the window, or partially visible through the window (e.g. the mountain-like polygon in Fig. 5-1). The clipping operation eliminates objects or portions of objects that are not visible through the window to ensure the proper construction of the corresponding image.

Note that clipping may occur in the world coordinate or viewing coordinate space, where the window is used to clip the objects; it may also occur in the normalized device coordinate space, where the viewport/workstation window is used to clip. In either case we refer to the window or the viewport/workstation window as the *clipping window*. We discuss point clipping, line clipping, and polygon clipping in Secs. 5.2, 5.3, and 5.4, respectively.

5.1 WINDOW-TO-VIEWPORT MAPPING

A window is specified by four world coordinates: wx_{min}, wx_{max}, wy_{min}, and wy_{max} (see Fig. 5-2). Similarly, a viewport is described by four normalized device coordinates: vx_{min}, vx_{max}, vy_{min}, and vy_{max}. The objective of window-to-viewport mapping is to convert the world coordinates (wx, wy) of an arbitrary point to its corresponding normalized device coordinates (vx, vy). In order to maintain the same relative placement of the point in the viewport as in the window, we require:

$$\frac{wx - wx_{min}}{wx_{max} - wx_{min}} = \frac{vx - vx_{min}}{vx_{max} - vx_{min}} \quad \text{and} \quad \frac{wy - wy_{min}}{wy_{max} - wy_{min}} = \frac{vy - vy_{min}}{vy_{max} - vy_{min}}$$

Thus

$$\begin{cases} vx = \dfrac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}}(wx - wx_{min}) + vx_{min} \\ vy = \dfrac{vy_{max} - vy_{min}}{wy_{max} - wy_{min}}(wy - wy_{min}) + vy_{min} \end{cases}$$

Since the eight coordinate values that define the window and the viewport are just constants, we can express these two formulas for computing (vx, vy) from (wx, wy) in terms of a translate–scale–translate transformation N

$$\begin{pmatrix} vx \\ vy \\ 1 \end{pmatrix} = N \cdot \begin{pmatrix} wx \\ wy \\ 1 \end{pmatrix}$$

where

$$N = \begin{pmatrix} 1 & 0 & vx_{min} \\ 0 & 1 & vy_{min} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \dfrac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}} & 0 & 0 \\ 0 & \dfrac{vy_{max} - vy_{min}}{wy_{max} - wy_{min}} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & -wx_{min} \\ 0 & 1 & -wy_{min} \\ 0 & 0 & 1 \end{pmatrix}$$

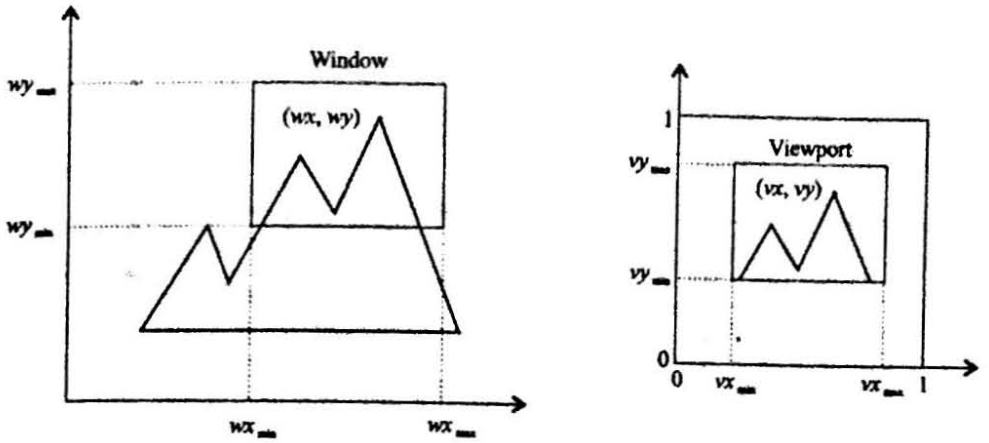


Fig. 5-2 Window-to-viewport mapping.

Note that geometric distortions occur (e.g. squares in the window become rectangles in the viewport) whenever the two scaling constants differ.

5.2 POINT CLIPPING

Point clipping is essentially the evaluation of the following inequalities:

$$x_{\min} \le x \le x_{\max} \qquad \text{and} \qquad y_{\min} \le y \le y_{\max}$$

where $x_{\min}$, $x_{\max}$, $y_{\min}$ and $y_{\max}$ define the clipping window. A point (x, y) is considered inside the window when the inequalities all evaluate to true.

5.3 LINE CLIPPING

Lines that do not intersect the clipping window are either completely inside the window or completely outside the window. On the other hand, a line that intersects the clipping window is divided by the intersection point(s) into segments that are either inside or outside the window. The following algorithms provide efficient ways to decide the spatial relationship between an arbitrary line and the clipping window and to find intersection point(s).

The Cohen–Sutherland Algorithm

In this algorithm we divide the line clipping process into two phases: (1) identify those lines which intersect the clipping window and so need to be clipped and (2) perform the clipping.

All lines fall into one of the following *clipping categories*:

1. *Visible*—both endpoints of the line lie within the window.
2. *Not visible*—the line definitely lies outside the window. This will occur if the line from (x_1, y_1) to (x_2, y_2) satisfies any one of the following four inequalities:

$$x_1, x_2 > x_{\max} \qquad y_1, y_2 > y_{\max}$$
$$x_1, x_2 < x_{\min} \qquad y_1, y_2 < y_{\min}$$

3. *Clipping candidate*—the line is in neither category 1 nor 2.

In Fig. 5-3, line AB is in category 1 (visible); lines CD and EF are in category 2 (not visible); and lines GH, IJ, and KL are in category 3 (clipping candidate).

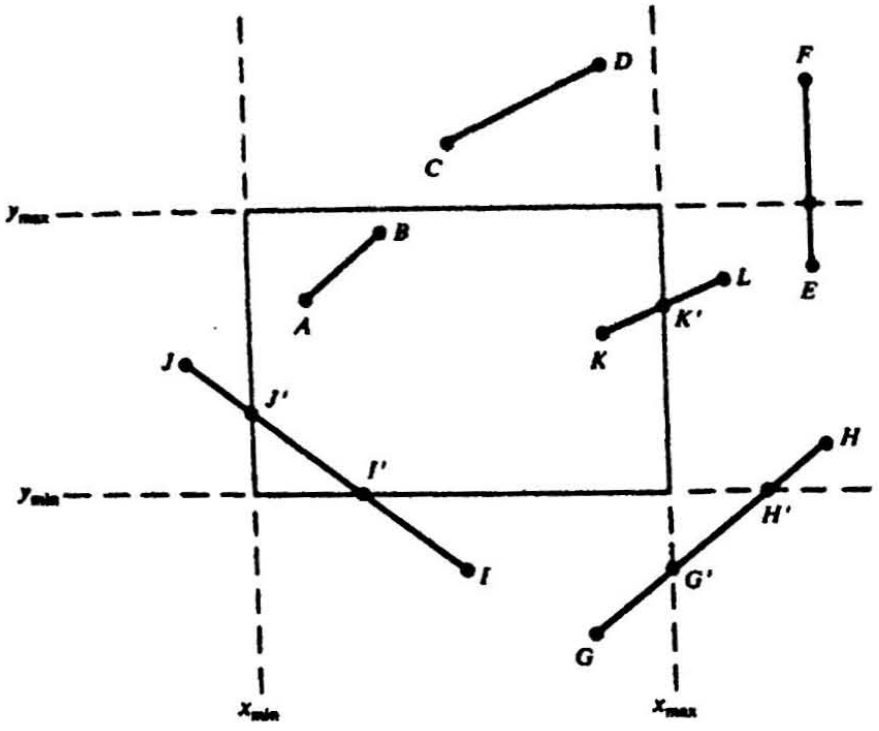
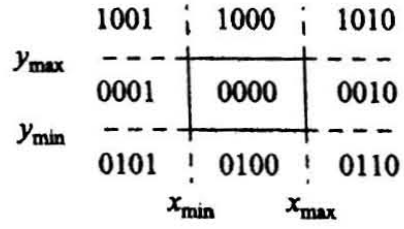


Fig. 5-3

The algorithm employs an efficient procedure for finding the category of a line. It proceeds in two steps:

1. Assign a 4-bit region code to each endpoint of the line. The code is determined according to which of the following nine regions of the plane the endpoint lies in



Starting from the leftmost bit, each bit of the code is set to true (1) or false (0) according to the scheme

$$\text{Bit 1} \equiv \text{endpoint is above the window} = \text{sign}\,(y - y_{max})$$
$$\text{Bit 2} \equiv \text{endpoint is below the window} = \text{sign}\,(y_{min} - y)$$
$$\text{Bit 3} \equiv \text{endpoint is to the right of the window} = \text{sign}\,(x - x_{max})$$
$$\text{Bit 4} \equiv \text{endpoint is to the left of the window} = \text{sign}\,(x_{min} - x)$$

We use the convention that $\text{sign}(a) = 1$ if a is positive, 0 otherwise. Of course, a point with code 0000 is inside the window.

2. The line is visible if both region codes are 0000, and not visible if the bitwise logical AND of the codes is not 0000, and a candidate for clipping if the bitwise logical AND of the region codes is 0000 (see Prob. 5.8).

For a line in category 3 we proceed to find the intersection point of the line with one of the boundaries of the clipping window, or to be exact, with the infinite extension of one of the boundaries (see Fig. 5-4). We choose an endpoint of the line, say (x_1, y_1), that is outside the window, i.e., whose region code is not

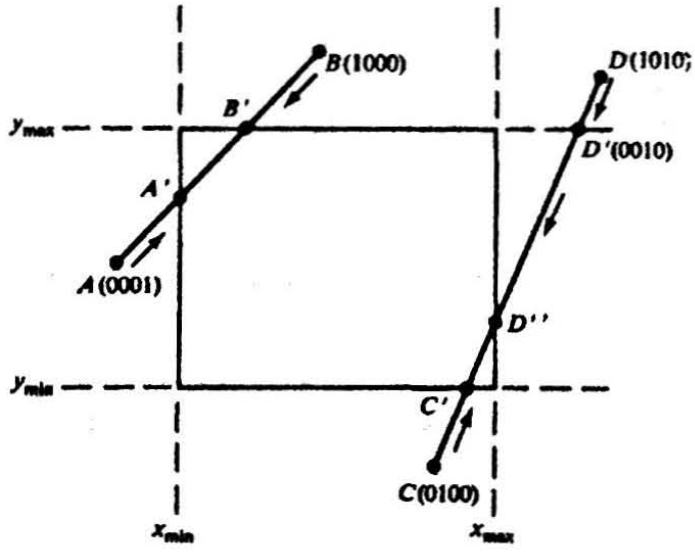


Fig. 5-4

0000. We then select an extended boundary line by observing that those boundary lines that are candidates for intersection are the ones for which the chosen endpoint must be "pushed across" so as to change a "1" in its code to a "0" (see Fig. 5-4). This means:

If bit 1 is 1, intersect with line $y = y_{max}$.
If bit 2 is 1, intersect with line $y = y_{min}$.
If bit 3 is 1, intersect with line $x = x_{max}$.
If bit 4 is 1, intersect with line $x = x_{min}$.

Consider line CD in Fig. 5-4. If endpoint C is chosen, then the bottom boundary line $y = y_{min}$ is selected for computing intersection. On the other hand, if endpoint D is chosen, then either the top boundary line $y = y_{max}$ or the right boundary line $x = x_{max}$ is used. The coordinates of the intersection point are

$$\begin{cases} x_i = x_{min} \text{ or } x_{max} \\ y_i = y_1 + m(x_i - x_1) \end{cases} \quad \text{if the boundary line is vertical}$$

or

$$\begin{cases} x_i = x_1 + (y_i - y_1)/m \\ y_i = y_{min} \text{ or } y_{max} \end{cases} \quad \text{if the boundary line is horizontal}$$

where $m = (y_2 - y_1)/(x_2 - x_1)$ is the slope of the line.

Now we replace endpoint (x_1, y_1) with the intersection point (x_i, y_i), effectively eliminating the portion of the original line that is on the outside of the selected window boundary. The new endpoint is then assigned an updated region code and the clipped line re-categorized and handled in the same way. This iterative process terminates when we finally reach a clipped line that belongs to either category 1 (visible) or category 2 (not visible).

Midpoint Subdivision

An alternative way to process a line in category 3 is based on binary search. The line is divided at its midpoint into two shorter line segments. The clipping categories of the two new line segments are then determined by their region codes. Each segment in category 3 is divided again into shorter segments and

categorized. This bisection and categorization process continues until each line segment that spans across a window boundary (hence encompasses an intersection point) reaches a threshold for line size and all other segments are either in category 1 (visible) or in category 2 (invisible). The midpoint coordinates (x_m, y_m) of a line joining (x_1, y_1) and (x_2, y_2) are given by

$$x_m = \frac{x_1 + x_2}{2} \qquad y_m = \frac{y_1 + y_2}{2}$$

The example in Fig. 5-5 illustrates how midpoint subdivision is used to zoom in onto the two intersection points I_1 and I_2 with 10 bisections. The process continues until we reach two line segments that are, say, pixel-sized, i.e., mapped to one single pixel each in the image space. If the maximum number of pixels in a line is M, this method will yield a pixel-sized line segment in N subdivisions, where $2^N = M$ or $N = \log_2 M$. For instance, when $M = 1024$ we need at most $N = \log_2 1024 = 10$ subdivisions.

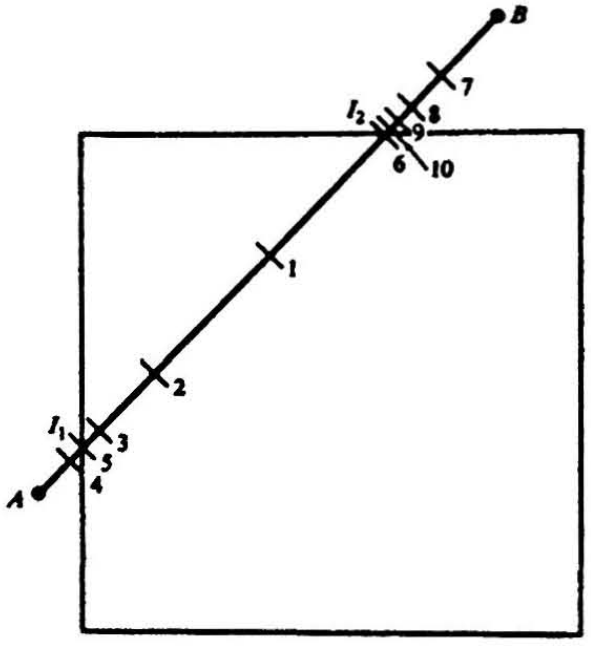


Fig. 5-5

The Liang–Barsky Algorithm

The following parametric equations represent a line from (x_1, y_1) to (x_2, y_2) along with its infinite extension:

$$\begin{cases} x = x_1 + \Delta x \cdot u \\ y = y_1 + \Delta y \cdot u \end{cases}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. The line itself corresponds to $0 \le u \le 1$ (see Fig. 5-6). Notice that when we traverse along the extended line with u increasing from $-\infty$ to $+\infty$, we first move from the outside to the inside of the clipping window's two boundary lines (bottom and left), and then move from the inside to the outside of the other two boundary lines (top and right). If we use u_1 and u_2, where $u_1 \le u_2$, to represent the beginning and end of the visible portion of the line, we have $u_1 = \text{maximum}(0, u_l, u_b)$ and $u_2 = \text{minimum}(1, u_t, u_r)$, where u_l, u_b, u_t, and u_r correspond to the intersection point of the extended line with the window's left, bottom, top, and right boundary, respectively.

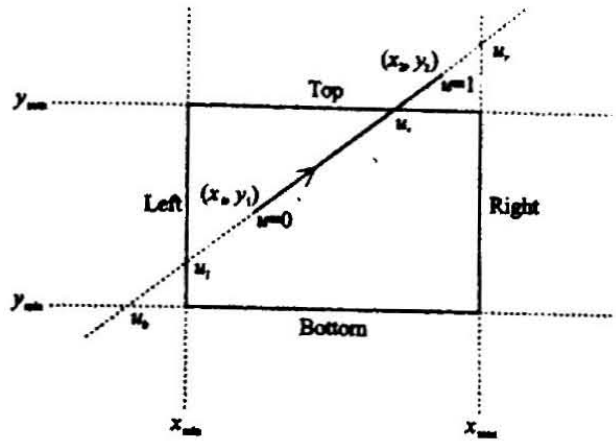


Fig. 5-6

Now consider the tools we need to turn this basic idea into an efficient algorithm. For point (x, y) inside the clipping window, we have

$$x_{\min} \le x_1 + \Delta x \cdot u \le x_{\max}$$
$$y_{\min} \le y_1 + \Delta y \cdot u \le y_{\max}$$

Rewrite the four inequalities as

$$p_k u \le q_k, \qquad k = 1, 2, 3, 4$$

where

$$\begin{aligned} p_1 &= -\Delta x & q_1 &= x_1 - x_{\min} & \text{(left)} \\ p_2 &= \Delta x & q_2 &= x_{\max} - x_1 & \text{(right)} \\ p_3 &= -\Delta y & q_3 &= y_1 - y_{\min} & \text{(bottom)} \\ p_4 &= \Delta y & q_4 &= y_{\max} - y_1 & \text{(top)} \end{aligned}$$

Observe the following facts:

- if $p_k = 0$, the line is parallel to the corresponding boundary and

$$\begin{cases} \text{if } q_k < 0, & \text{the line is completely outside the boundary and can be eliminated} \\ \text{if } q_k \ge 0, & \text{the line is inside the boundary and needs further consideration,} \end{cases}$$

- if $p_k < 0$, the extended line proceeds from the outside to the inside of the corresponding boundary line,
- if $p_k > 0$, the extended line proceeds from the inside to the outside of the corresponding boundary line,
- when $p_k \ne 0$, the value of u that corresponds to the intersection point is q_k/p_k.

The Liang–Barsky algorithm for finding the visible portion of the line, if any, can be stated as a four-step process:

1. If $p_k = 0$ and $q_k < 0$ for any k, eliminate the line and stop. Otherwise proceed to the next step.
2. For all k such that $p_k < 0$, calculate $r_k = q_k/p_k$. Let u_1 be the maximum of the set containing 0 and the calculated r values.
3. For all k such that $p_k > 0$, calculate $r_k = q_k/p_k$. Let u_2 be the minimum of the set containing 1 and the calculated r values.
4. If $u_1 > u_2$, eliminate the line since it is completely outside the clipping window. Otherwise, use u_1 and u_2 to calculate the endpoints of the clipped line.

5.4 POLYGON CLIPPING

In this section we consider the case of using a polygonal clipping window to clip a polygon.

Convex Polygonal Clipping Windows

A polygon is called *convex* if the line joining any two interior points of the polygon lies completely inside the polygon (see Fig. 5-7). A non-convex polygon is said to be *concave*.

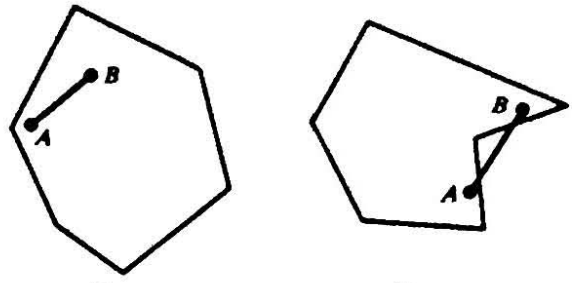
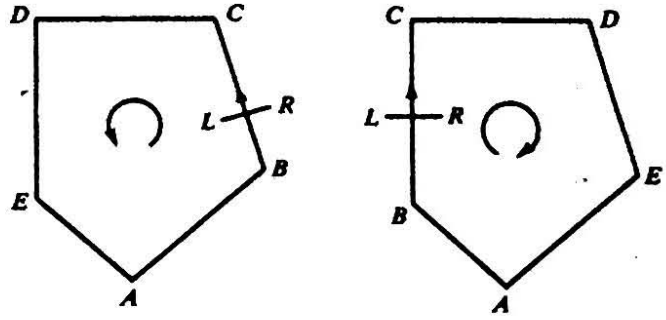


Fig. 5-7

By convention, a polygon with vertices $P_1, \ldots, P_N$ (and edges $P_{i-1}P_i$ and P_NP_1) is said to be *positively oriented* if a tour of the vertices in the given order produces a counterclockwise circuit.

Equivalently, the left hand of a person standing along any directed edge $\overline{P_{i-1}P_i}$ or $\overline{P_NP_1}$ would be pointing inside the polygon [see orientations in Figs. 5-8(*a*) and 5-8(*b*)].



(*a*) Positive orientation. (*b*) Negative orientation.

Fig. 5-8

Let $A(x_1, y_1)$ and $B(x_2, y_2)$ be the endpoints of a directed line segment. A point $P(x, y)$ will be to the *left* of the line segment if the expression $C = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1)$ is positive (see Prob. 5.13). We say that the point is to the *right* of the line segment if this quantity is negative. If a point P is to the right of any one edge of a positively oriented, convex polygon, it is outside the polygon. If it is to the left of *every* edge of the polygon, it is inside the polygon.

This observation forms the basis for clipping any polygon, convex or concave, against a convex polygonal clipping window.

The Sutherland–Hodgman Algorithm

Let $P_1, \ldots, P_N$ be the vertex list of the polygon to be clipped. Let edge E, determined by endpoints A and B, be any edge of the positively oriented, convex clipping polygon. We clip each edge of the polygon in

turn against the edge E of the clipping polygon, forming a new polygon whose vertices are determined as follows.

Consider the edge $\overline{P_{i-1}P_i}$:

1. If both P_{i-1} and P_i are to the left of the edge, vertex P_i is placed on the *vertex output list* of the clipped polygon [Fig. 5-9(*a*)].
2. If both P_{i-1} and P_i are to the right of the edge, nothing is placed on the vertex output list [Fig. 5-9(*b*)].
3. If P_{i-1} is to the left and P_i is to the right of the edge E, the intersection point I of line segment $\overline{P_{i-1}P_i}$ with the extended edge E is calculated and placed on the vertex output list [Fig. 5-9(*c*)].
4. If P_{i-1} is to the right and P_i is to the left of edge E, the intersection point I of the line segment $\overline{P_{i-1}P_i}$ with the extended edge E is calculated. Both I and P_i are placed on the vertex output list [Fig. 5-9(*d*)].

The algorithm proceeds in stages by passing each clipped polygon to the next edge of the window and clipping. See Probs. 5.14 and 5.15.

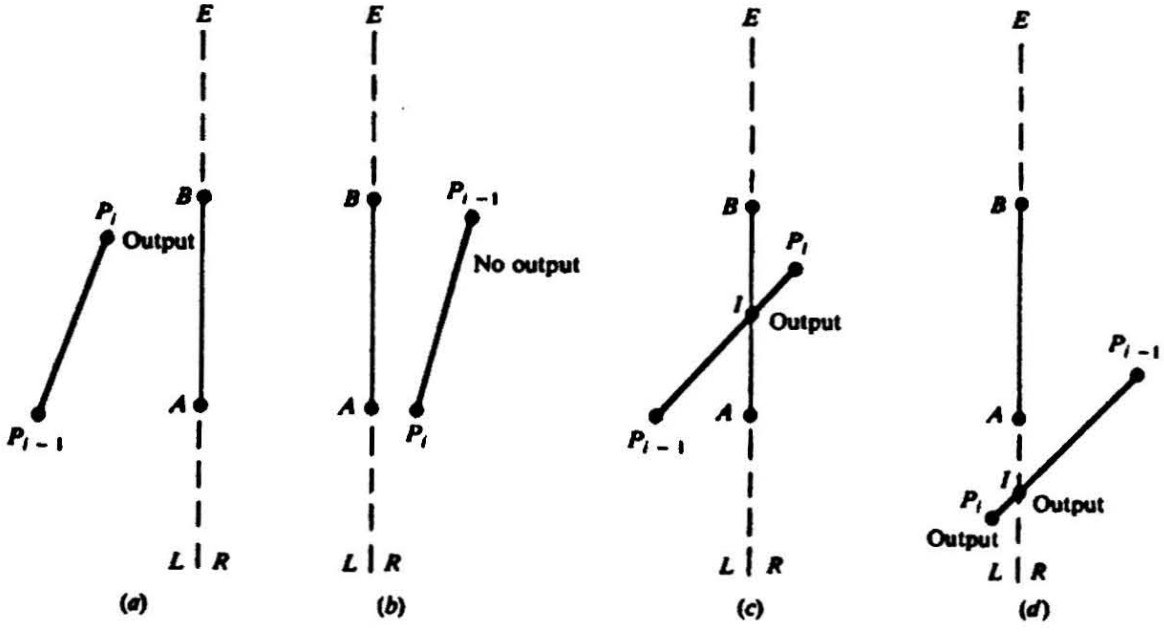


Fig. 5-9

Special attention is necessary in using the Sutherland–Hodgman algorithm in order to avoid unwanted effects. Consider the example in Fig. 5-10(*a*). The correct result should consist of two disconnected parts, a square in the lower left corner of the clipping window and a triangle at the top [see Fig. 5-10(*b*)]. However, the algorithm produces a list of vertices (see Prob. 5.16) that forms a figure with the two parts connected by extra edges [see Fig. 5-10(*c*)]. The fact that these edges are drawn twice in opposite direction can be used to devise a post-processing step to eliminate them.

The Weiler–Atherton Algorithm

Let the clipping window be initially called the clip polygon, and the polygon to be clipped the subject polygon [see Fig. 5-11(*a*)]. We start with an arbitrary vertex of the subject polygon and trace around its border in the clockwise direction until an intersection with the clip polygon is encountered:

- If the edge enters the clip polygon, record the intersection point and continue to trace the subject polygon.

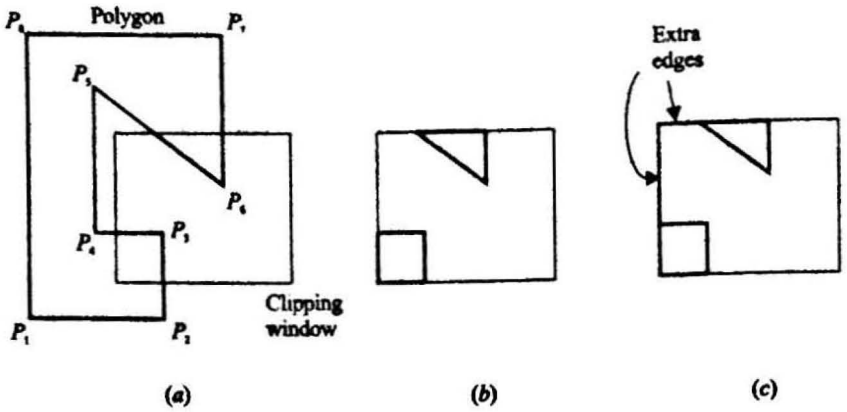


Fig. 5-10

- If the edge leaves the clip polygon, record the intersection point and make a right turn to follow the clip polygon in the same manner (i.e., treat the clip polygon as subject polygon and the subject polygon as clip polygon and proceed as before).

Whenever our path of traversal forms a sub-polygon we output the sub-polygon as part of the overall result. We then continue to trace the rest of the original subject polygon from a recorded intersection point that marks the beginning of a not-yet-traced edge or portion of an edge. The algorithm terminates when the entire border of the original subject polygon has been traced exactly once.

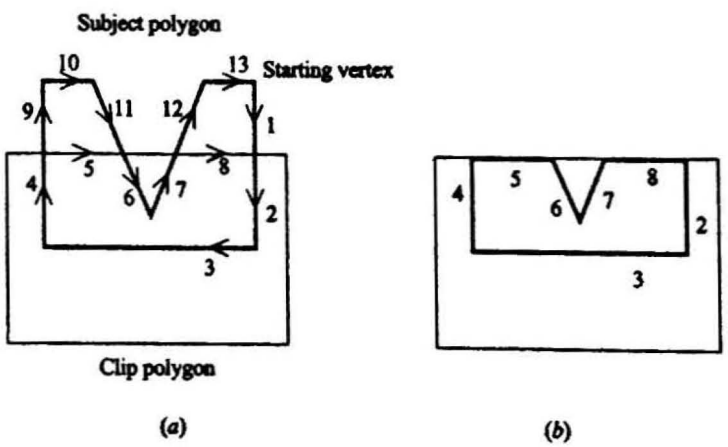


Fig. 5-11

For example, the numbers in Fig. 5-11(*a*) indicate the order in which the edges and portions of edges are traced. We begin at the starting vertex and continue along the same edge (from 1 to 2) of the subject polygon as it enters the clip polygon. As we move along the edge that is leaving the clip polygon we make a right turn (from 4 to 5) onto the clip polygon, which is now considered the subject polygon. Following the same logic leads to the next right turn (from 5 to 6) onto the current clip polygon, which is really the original subject polygon. With the next step done (from 7 to 8) in the same way we have a sub-polygon for output [see Fig. 5-11(*b*)]. We then resume our traversal of the original subject polygon from the recorded intersection point where we first changed our course. Going from 9 to 10 to 11 produces no output. After

skipping the already-traversed 6 and 7, we continue with 12 and 13 and come to an end. The figure in Fig. 5-11(*b*) is the final result.

Applying the Weiler–Atherton algorithm to clip the polygon in Fig. 5-10(*a*) produces correct result [see Fig. 5-12(*a*) and (*b*)].

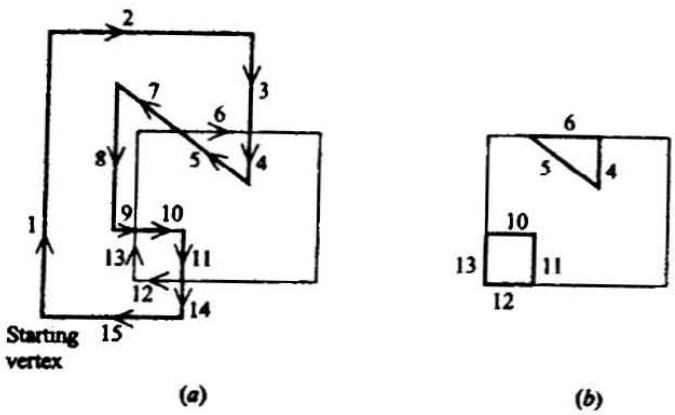


Fig. 5-12

5.5 EXAMPLE: A 2D GRAPHICS PIPELINE

Shared by many graphics systems is the overall system architecture called the graphics pipeline. The operational organization of a 2D graphics pipeline is shown in Fig. 5-13. Although 2D graphics is typically treated as a special case ($z = 0$) of three-dimensional graphics, it demonstrates the common working principle and basic application of these pipelined systems.

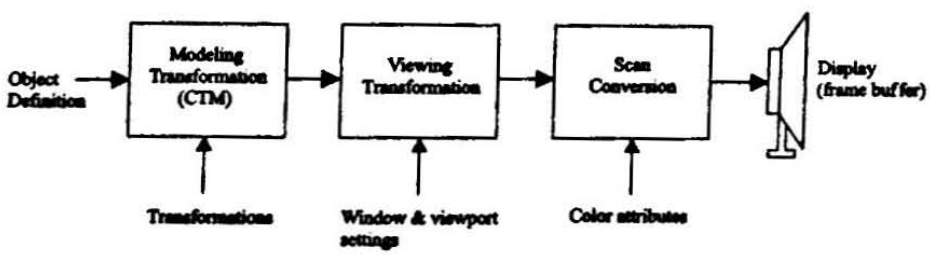


Fig. 5-13 A 2D graphics pipeline.

At the beginning of the pipeline we have object data (e.g., vertex coordinates for lines and polygons that make up individual objects) stored in application-specific data structures. A graphics application uses system subroutines to initialize and to change, among other things, the transformations that are to be applied to the original data, window and viewport settings, and the color attributes of the objects. Whenever a drawing subroutine is called to render a pre-defined object, the graphics system first applies the specified modeling transformation to the original data, then carries out viewing transformation using the current window and viewport settings, and finally performs scan conversion to set the proper pixels in the frame buffer with the specified color attributes.

Suppose that we have an object centered in its own coordinate system [see Fig. 5-14(*a*)], and we are to construct a sequence of images that animates the object rotating around its center and moving along a circular path in a square display area [see Fig. 5-14(*b*)]. We generate each image as follows: first rotate the object around its center by angle α, then translate the rotated object by offset $\cdot$ **I** to position its center on the positive x axis of the WCS, and rotate it with respect to the origin of the WCS by angle β. We control the amount of the first rotation from one image to the next by $\Delta\alpha$, and that of the second rotation by $\Delta\beta$.

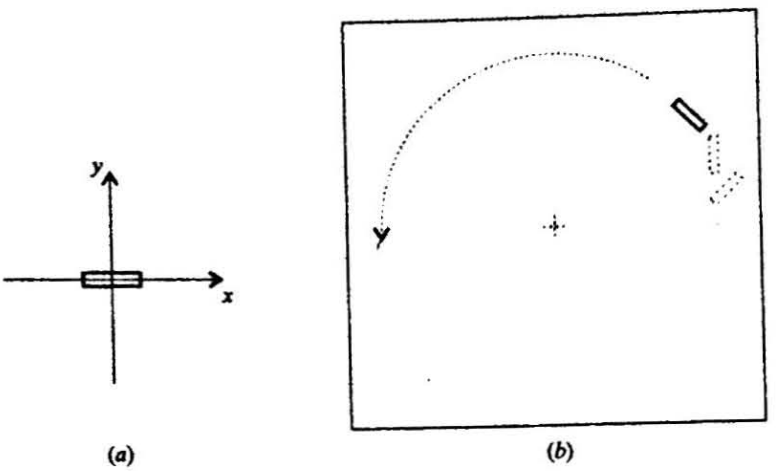


Fig. 5-14

```
window(-winsize/2, winsize/2, -winsize/2, winsize/2);
α = 0;
while (1) {
   setColor(background);
   clear();
   setColor(color);
   pushCTM();
   translate(offset, 0);
   rotate(α);
   drawObject();
   popCTM();
   α = α + Δα;
   rotate(Δβ);
}
```

We first set the window of winsize by winsize to be sufficiently large and centered at the origin of the WCS to cover the entire scene. The system's default viewport coincides with the unit display area in the NDCS. The default workstation window is the same as the viewport and the default workstation viewport corresponds to the whole square display area.

The graphics system maintains a stack of composite transformation matrices. The CTM on top of the stack, called the current CTM, is initially an identity matrix and is automatically used in modeling transformation. Each call to translate, scale, and rotate causes the system to generate a corresponding transformation matrix and to reset the current CTM to take into account the generated matrix. The order of transformation is maintained in such a way that the most recently specified transformation is applied first. When pushCTM() is called, the system makes a copy of the current CTM and pushes it onto the stack (now we have two copies of the current CTM on the stack). When popCTM() is called, the system simply removes the CTM on top of the stack (now we have restored the CTM that was second to the removed CTM to be the current CTM).

Panning and Zooming

Two simple camera effects can be achieved by changing the position or size of the window. When the position of the window is, for example, moved to the left, an object in the scene that is visible through the window would appear moved to the right, much like what we see in the viewfinder when we move or pan a