Window & viewport

window	viewport
(1) A world coordinate area selected for display is called window	(1) An area on a display device to which a window is mapped called a viewport.
(2) window defines what is to be displayed	(2) The viewport defines where it is to ~~d~~ be displayed.
(3) It represents the world coordinate system (Wx, Wy)	(3) It represents device coordinate system (Vx, Vy)

<u>Line Clipping</u> → A line clipping procedure involves several parts like

1) Test line segment to determine whether it is completely inside outside the clip window, & then test it cross one or more clipping boundries and may require calculation of multiple intersection point. To minimize calculation, we devise clipping algorithms that can efficiently identify outside lines and reduce intersection calculations

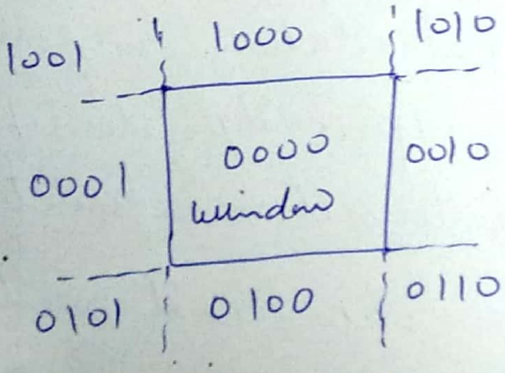
For a line segment with end points (x_1, y_1) and (x_2, y_2) and one or more endpoints outside the clipping rectangle, the parametric representation is

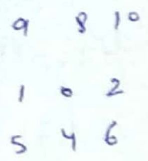
$$x = x_1 + u(x_2 - x_1)$$
$$y = y_1 + u(y_2 - y_1)$$
where $0 \le u \le 1$

the value of u determines the intersection with the clipping boundary.
If the value of u for an intersection with a rectangle boundary edge is outside the range 0 to 1 the line is outside the boundary
if $0 \leq u \leq 1$ line segment intersect the clipping window.

Cohen Sutherland line Clipping Algo:-
This is an efficient line clipping algorithm which speeds up the processing of line segment by performing initial list that reduces the no of intersection calculations. It uses the bit operations for clipping every line end point in a picture is assigned a four digit binary code called region code, that identify the location of the point relative to the boundries of clipping rectangle.

Method:-
Step 1. set the value of each line end points in the form of 4 bit binary code ~~4~~ ~~6~~



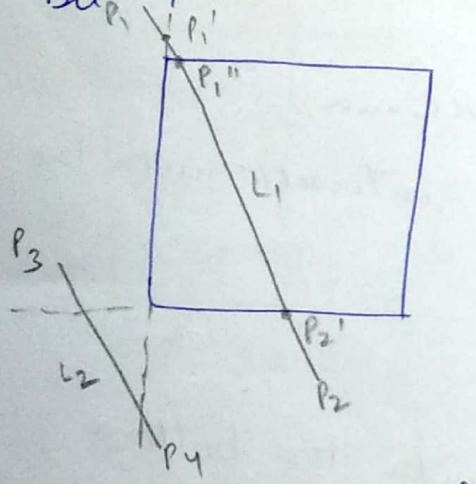


Binary region code assigned to the line end points according to relative position with respect to clipping rectangle.

Bit 1 – left
Bit 2 – Right
Bit 3 – below
Bit 4 – above

A value of 1 in any bit position indicates that the point is in that relative position, otherwise the bit position is set to 0. If the point is within the clip window, the region code is 0000.

Bit 1 is the sign bit of $(x - xw_{min})$ $x < xw_{min}$, set bit "1" otherwise "0"
Bit 2 is the sign bit of $(x - xw_{max})$ $x > xw_{max}$ set bit "1" otherwise "0"
Bit 3 is the sign bit of $(y - yw_{min})$ $y < yw_{min}$ set bit "1" otherwise "0"
Bit 4 is the sign bit of $(y - yw_{max})$ $y > yw_{max}$ set bit "1" otherwise "0"



we have two lines L1 & L2
for line L1 (P_1) = (1001)
line L1 (P_2) = (0100)

For point P_1 of line L_1, it is situated at above and to the left of the window so its Bit 1 and Bit 4 set as 1.

__Step 2__ → After calculating the region code values for all the lines the end points values for lines are checked for trivial acceptance or rejection

① If both end points of the line have region code value "0000" then line is completely inside the window & __accept__ these lines.

(2) If logical AND operation of both the region code of two end points is not "0000" or <> "0000" then line is completely outside the window & it is rejected.

Step 3 If trivially acceptance/rejection failed then line is partially inside and outside so we find out intersection with the boundries of the window

$$\text{slope } m = \frac{y_2 - y_1}{x_2 - x_1}$$

a) If bit 1 is "1" then line intersect with the left boundary

$y_i = y_1 + m(x - x_1)$ where $x = xw_{min}$

(b) If bit 2 is "1" then line intersect with the right boundary

$$y_i = y_1 + m(x - x_1)$$

where $x = xw_{max}$

(c) If bit 3 is "1" line intersect with the bottom boundary

$$x_i = x_1 + (y - y_1)/m$$

where $y = yw_{min}$

d) If bit 4 is "1" line intersect with the upper boundary

$$x_i = x_1 + (y - y_1)/m$$

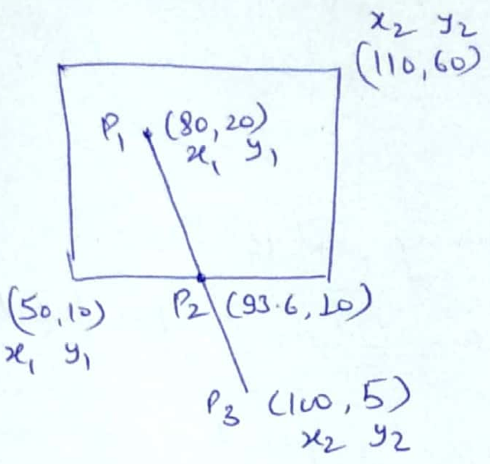
where $y = yw_{max}$

x_i and y_i are the x, y intercept for the line.

Step 4 Iteratively clipped by testing acceptance or rejection Divide the line segment until completely inside or completely outside

Q1 use the cohen sutherland line clipping algorithm ③ to clip line $P_1(80, 20)$ and $P_2(100, 5)$ against a window (50, 10) and (110, 60)

Soln $\Rightarrow$



Region code for P_1 (80, 20)

Bit	Comparision	Result
1	$80 < 50$	0
2	$80 > 110$	0
3	$20 < 10$	0
4	$20 > 60$	0

$P_1 = 0000$ (i.e. it is inside the window)

Region code for P_2 (100, 5)

Bit	Comparision	Result
1	$100 < 50$	0
2	$100 > 110$	0
3	$5 < 10$	~~0~~ 1
4	$5 > 60$	0

$P_2 \rightarrow 0100$ is outside the window

$P_1 = 0000$, $P_2 = 0100$

Slope $m = \dfrac{y_2 - y_1}{x_2 - x_1} \Rightarrow \dfrac{5 - 20}{100 - 80} = \dfrac{-15}{20} = -0.75$

From the outcode (0100) the below bit is 1 so line intersect with bottom boundary

$x = x_1 + (y - y_1)/m$

$x = 100 + (10-5)/-0.75$

$x = 100 - 6.33 = 93.67$ $(y = y_{wmin} = 10)$

so the intersecting point ~~is in~~ bottom is (93.6, 10)

so the new line end points (80, 20) to (93.6, 10)

II Cyrus Back Line Clipping Algorithm – It is more efficient algorithm than Cohen Sutherland Line Clip Algo.

For determining whether a point is inside or or outside a window, this algorithm uses the normal vector to accomplish this.

Here we use i and j as the unit vectors in the x and y directions respectively.

The four inner normal vectors are

Left normal vector $= i$

Right normal vector $= -i$

Bottom normal vector $= j$

Top normal vector $= -j$

Step 1 – The parametric representation of a line is

$P(u) = P_1 + (P_2 - P_1)u$

where $0 \le u \le 1$

$P(u) = [x_1 y_1] + [(x_2 - x_1), (y_2 - y_1)]u$

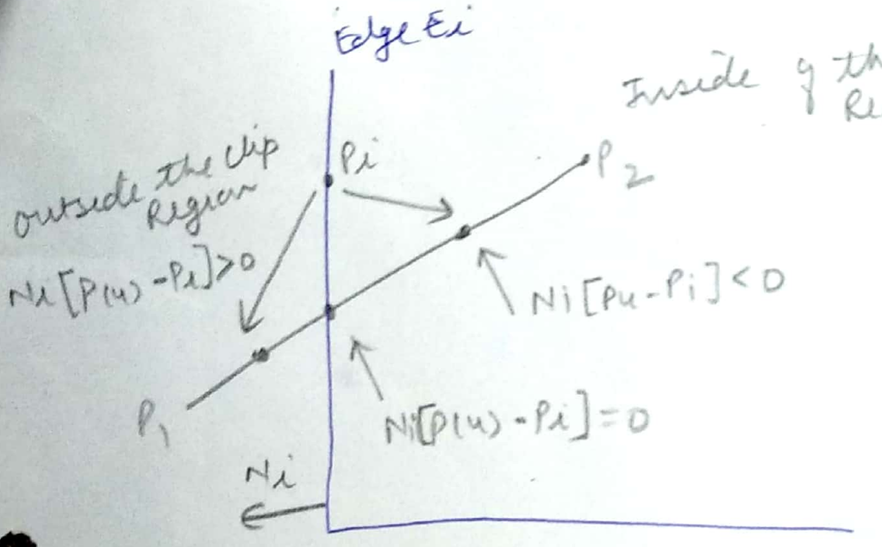
Step 2 – The dot product of inner normal and the vector from any point on the parametric line to any other point on the boundary

$n_i [P(u) - f_i]$ $i = 1, 2, 3$ ---- ---- eq ① is positive, zero or negative for a point on the interior to the region boundary, on the region boundary or exterior to the region

If → $n_i[P(u) - f_i] < 0$ i is negative a point is outside the region boundary

→ $n_i[P(u) - f_i] > 0$ i is positive a point is inside the region boundary

→ $n_i[P(u) - f_i] = 0$ i = zero a point is on the region boundary.



we can write the eq. ① as

$$n_i[P_1 + (P_2 - P_1)u - f_i] = 0$$

which lies on the boundary of the region where vector $P_2 - P_1$ defines the direction of the line

Let $x = P_2 - P_1$

$x =$ direction of the line

$y_i = P_1 - f_i$

y_i = weighting factor

$u(n_i x) + y_i n_i = 0$

Therefore $u = -(y_i n_i) / (n_i x)$ $x <> 0$

$i = 1, 2, 3$

$x . n_i$ can be zero only if $x = 0$ which implies that

$P_2 = P_1$

ie . a point if

$y_i . n_i < 0$ the point is outside

$= 0$ the point is on boundary

> 0 inside

Window is the region of the scene or object to be depticted, defined by a rectangle. A window frame outline's and defines the portion of the scene or object that we are able to see from a particular position at a particular time.
Viewport is the zone of the monitor screen within which we wish to view the selected object. The viewport may be visualized as any rectangular area of monitor screen within which appears the chosen window

We now consider the formal mechanism for displaying views of a picture on an output device. Typically, a graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture. For a two-dimensional picture, a view is selected by specifying a subarea of the total picture area. A user can select a single area for display, or several areas could be selected for simultaneous display or for an animated panning sequence across a scene. The picture parts within the selected areas are then mapped onto specified areas of the device coordinates. When multiple view areas are selected, these areas can be placed in separate display locations, or some areas could be inserted into other, larger display areas. Transformations from world to device coordinates involve translation, rotation, and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected display area.

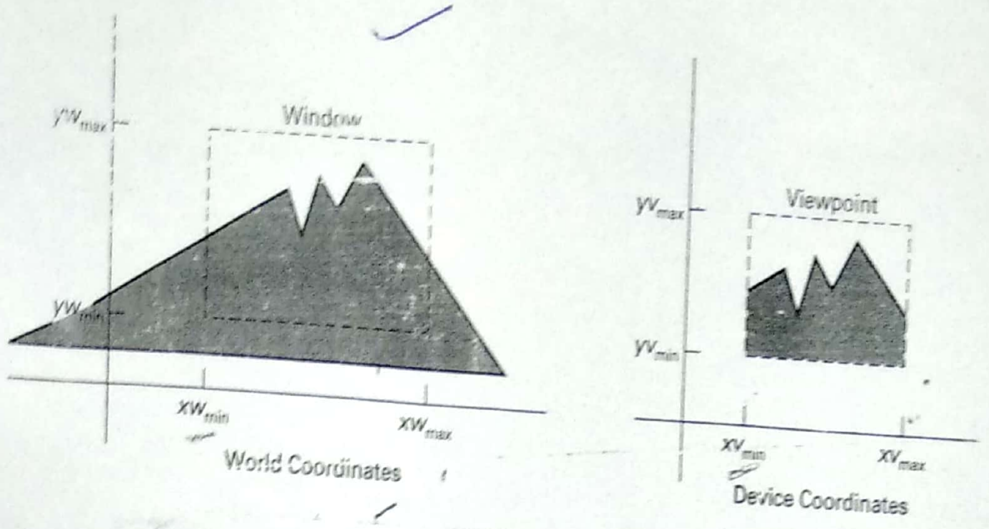
A window is simply the defined view of what a person wishes to represents through Computer graphics.

6-1 THE VIEWING PIPELINE ✓

SR

A world-coordinate area selected for display is called a **window**. An area on a display device to which a window is mapped is called a **viewport**. The window defines *what* is to be viewed; the viewport defines *where* it is to be displayed. Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes. Other window or viewport geometries, such as general polygon shapes and circles, are used in some applications, but these shapes take longer to process. In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a **viewing transformation**. Sometimes the two-dimensional viewing transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation*. But, in general, viewing involves more than just the transformation from the window to the viewport. Figure 6-1 illustrates the mapping of a picture section that falls within a rectangular window onto a designated rectangular viewport.

In computer graphics terminology, the term *window* originally referred to an area of a picture that is selected for viewing, as defined at the beginning of this section. Unfortunately, the same term is now used in window-manager systems to refer to any rectangular screen area that can be moved about, resized, and made active or inactive. In this chapter, we will only use the term window to

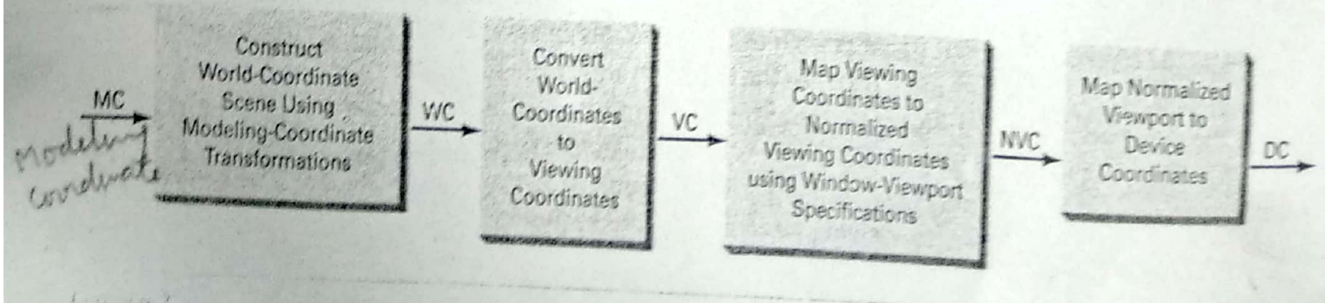


A viewing transformation using standard rectangles for the window and viewport.

refer to an area of a world-coordinate scene that has been selected for display. When we consider graphical user interfaces in Chapter 8, we will discuss screen windows and window-manager systems.

Some graphics packages that provide window and viewport operations allow only standard rectangles, but a more general approach is to allow the rectangular window to have any orientation. In this case, we carry out the viewing transformation in several steps, as indicated in Fig. 6-2. First, we construct the scene in world coordinates using the output primitives and attributes discussed in Chapters 3 and 4. Next, to obtain a particular orientation for the window, we can set up a two-dimensional **viewing-coordinate system** in the world-coordinate plane, and define a window in the viewing-coordinate system. The viewing-coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates. We then define a viewport in normalized coordinates (in the range from 0 to 1) and map the viewing-coordinate description of the scene to normalized coordinates. At the final step, all parts of the picture that lie outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. Figure 6-3 illustrates a rotated viewing-coordinate reference frame and the mapping to normalized coordinates.

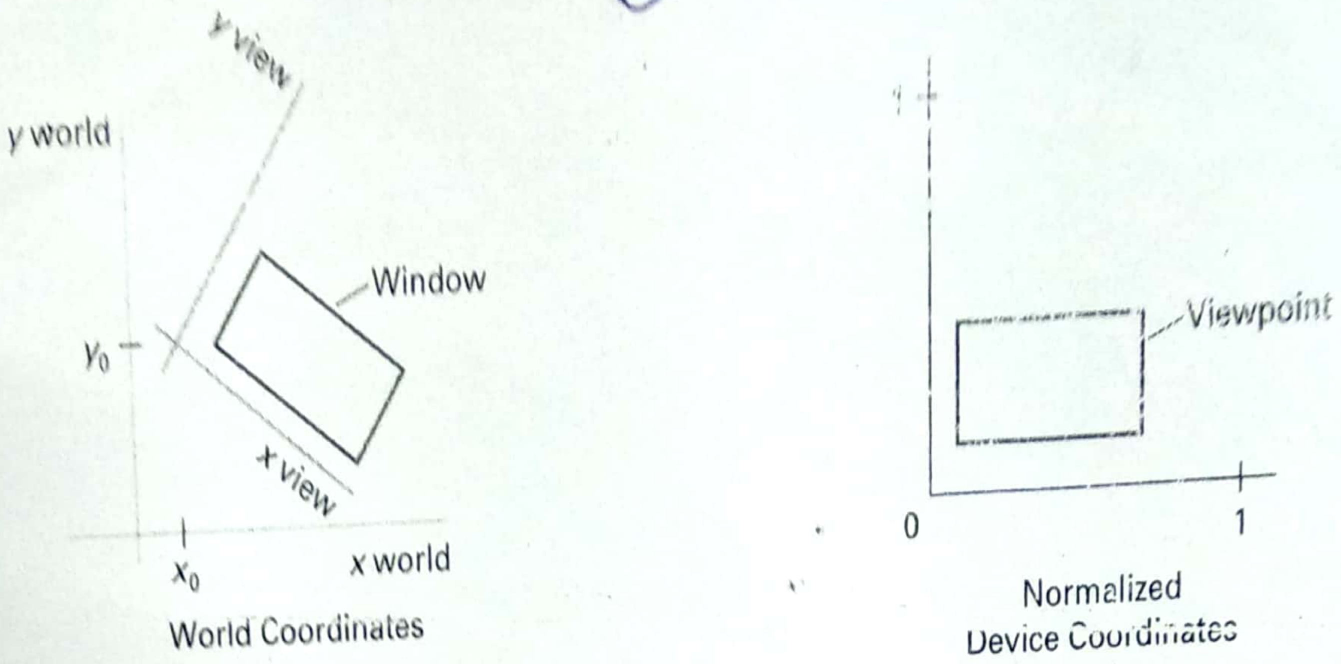
By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized windows on a



MC
Modeling coordinate
Construct World-Coordinate Scene Using Modeling-Coordinate Transformations
WC
Convert World-Coordinates to Viewing Coordinates
VC
Map Viewing Coordinates to Normalized Viewing Coordinates using Window-Viewport Specifications
NVC
Map Normalized Viewport to Device Coordinates
DC

Figure 6-2
The two-dimensional viewing-transformation pipeline.

40



Setting up a rotated world window in viewing coordinates and the corresponding normalized-coordinate viewport.

fixed-size viewport. As the windows are made smaller, we zoom in on some part of a scene to view details that are not shown with larger windows. Similarly, more overview is obtained by zooming out from a section of a scene with successively larger windows. Panning effects are produced by moving a fixed-size window across the various objects in a scene.

Viewports are typically defined within the unit square (normalized coordinates). This provides a means for separating the viewing and other transformations from specific output-device requirements, so that the graphics package is largely device-independent. Once the scene has been transferred to normalized coordinates, the unit square is simply mapped to the display area for the particular output device in use at that time. Different output devices can be used by providing the appropriate device drivers.

When all coordinate transformations are completed, viewport clipping can be performed in normalized coordinates or in device coordinates. This allows us to reduce computations by concatenating the various transformation matrices. Clipping procedures are of fundamental importance in computer graphics. They are used not only in viewing transformations, but also in window-manager systems, in painting and drawing packages to eliminate parts of a picture inside or outside of a designated screen area, and in many other applications.

TWO-DIMENSIONAL VIEWING FUNCTIONS ✓

We define a viewing reference system in a PHIGS application program with the following function:

```
evaluateViewOrientationMatrix (x0, y0, xV, yV,
                               error, viewMatrix)
```

where parameters x0 and y0 are the coordinates of the viewing origin, and parameters xV and yV are the world-coordinate positions for the view up vector. An integer error code is generated if the input parameters are in error; otherwise, the viewMatrix for the world-to-viewing transformation is calculated. Any number of viewing transformation matrices can be defined in an application.

To set up the elements of a window-to-viewport mapping matrix, we invoke the function

```
evaluateViewMappingMatrix (xwmin, xwmax, ywmin, ywmax,
    xvmin, xvmax, yvmin, yvmax, error, viewMappingMatrix)
```

Here, the window limits in viewing coordinates are chosen with parameters xwmin, xwmax, ywmin, and ywmax; and the viewport limits are set with the nor-

6-5 CLIPPING OPERATIONS

Generally, any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a **clipping algorithm,** or simply **clipping**. The region against which an object is to clipped is called a **clip window**.

Applications of clipping include extracting part of a defined scene for viewing; identifying visible surfaces in three-dimensional views; antialiasing line segments or object boundaries; creating objects using solid-modeling procedures; displaying a multiwindow environment; and drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating. Depending on the application, the clip window can be a general polygon or it can even have curved boundaries. We first consider clipping methods using rectangular clip regions, then we discuss methods for other clip-region shapes.

For the viewing transformation, we want to display only those picture parts that are within the window area (assuming that the clipping flags have not been set to *noclip*). Everything outside the window is discarded. Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates. Alternatively, the complete world-coordinate picture can be mapped first to device coordinates, or normalized device coordinates, then clipped against the viewport boundaries. World-coordinate clipping removes those primitives outside the window from further consideration, thus eliminating the processing necessary to transform those primitives to device space. Viewport clipping, on the other hand, can reduce calculations by allowing concatenation of viewing and geometric transformation matrices. But viewport clipping does require that the transformation to device coordinates be performed for all objects, including those outside the window area. On raster systems, clipping algorithms are often combined with scan conversion.

In the following sections, we consider algorithms for clipping the following primitive types

- Point Clipping
- Line Clipping (straight-line segments)
- Area Clipping (polygons)
- Curve Clipping
- Text Clipping

Line and polygon clipping routines are standard components of graphics packages, but many packages accommodate curved objects, particularly spline curves and conics, such as circles and ellipses. Another way to handle curved objects is to approximate them with straight-line segments and apply the line- or polygon-clipping procedure.] SR

6-6 POINT CLIPPING ✓

SR
Assuming that the clip window is a rectangle in standard position, we save a point $\mathbf{P} = (x, y)$ for display if the following inequalities are satisfied:

$$xw_{\min} \le x \le xw_{\max} \qquad (6\text{-}5)$$
$$yw_{\min} \le y \le yw_{\max}$$

where the edges of the clip window $(xw_{\min}, xw_{\max}, yw_{\min}, yw_{\max})$ can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

Although point clipping is applied less often than line or polygon clipping, some applications may require a point-clipping procedure. For example, point clipping can be applied to scenes involving explosions or sea foam that are modeled with particles (points) distributed in some region of the scene.] SR

6-7 LINE CLIPPING ✓

SR
Figure 6-7 illustrates possible relationships between line positions and a standard rectangular clipping region. A line-clipping procedure involves several parts. First, we can test a given line segment to determine whether it lies completely inside the clipping window. If it does not, we try to determine whether it lies completely outside the window. Finally, if we cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries. We process lines through the "inside-outside" tests by checking the line endpoints. A line with both endpoints inside all clipping boundaries, such as the line from $\mathbf{P}_1$ to $\mathbf{P}_2$, is saved. A line with both endpoints outside any one of the clip boundaries (line $\overline{\mathbf{P}_3\mathbf{P}_4}$ in Fig. 6-7) is outside the win-

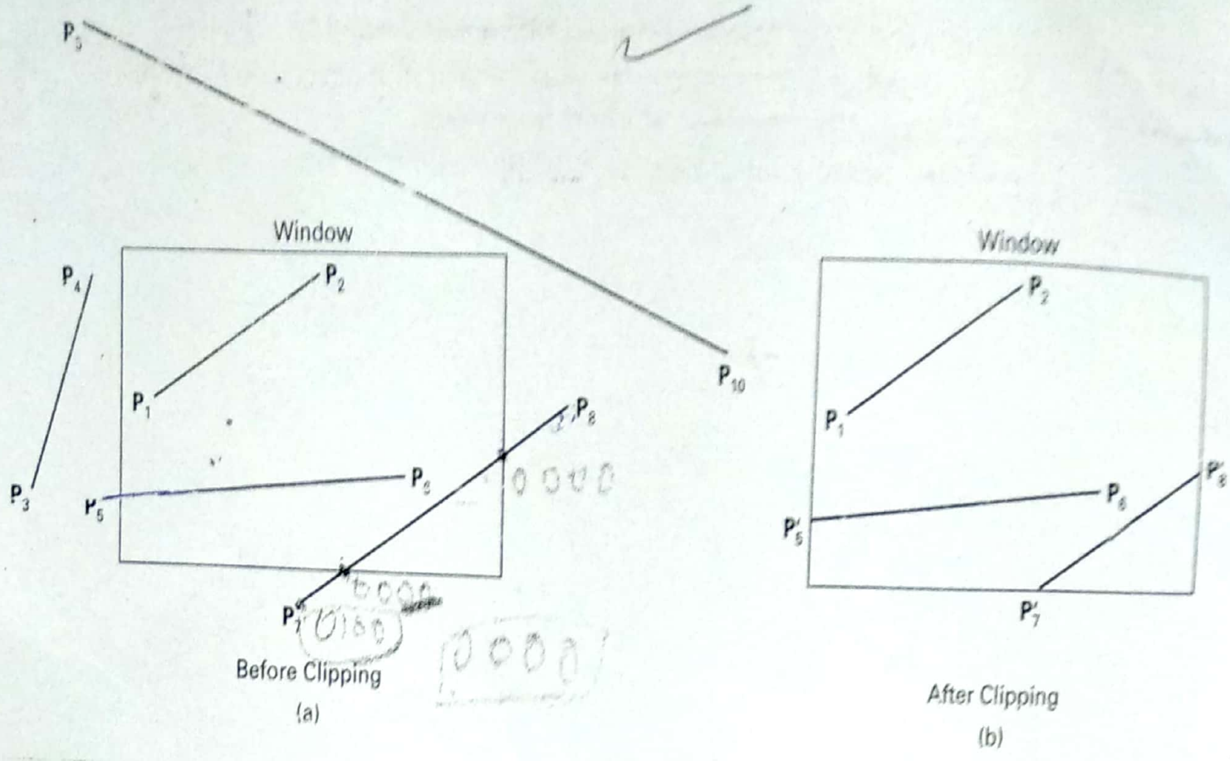


Figure 6-7
Line clipping against a rectangular clip window.

dow. All other lines cross one or more clipping boundaries, and may require calculation of multiple intersection points. To minimize calculations, we try to devise clipping algorithms that can efficiently identify outside lines and reduce intersection calculations.

For a line segment with endpoints (x_1, y_1) and (x_2, y_2) and one or both endpoints outside the clipping rectangle, the parametric representation

$$x = x_1 + u(x_2 - x_1)$$
$$y = y_1 + u(y_2 - y_1), \qquad 0 \le u \le 1 \qquad (6\text{-}6)$$

could be used to determine values of parameter u for intersections with the clipping boundary coordinates. If the value of u for an intersection with a rectangle boundary edge is outside the range 0 to 1, the line does not enter the interior of the window at that boundary. If the value of u is within the range from 0 to 1, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in turn to determine whether any part of the line segment is to be displayed. Line segments that are parallel to window edges can be handled as special cases.

Clipping line segments with these parametric tests requires a good deal of computation, and faster approaches to clipping are possible. A number of efficient line clippers have been developed, and we survey the major algorithms in the next sections. Some algorithms are designed explicitly for two-dimensional pictures and some are easily adapted to three-dimensional applications.

SR

SR Cohen–Sutherland Line Clipping ✓

This is one of the oldest and most popular line-clipping procedures. Generally, the method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated. Every line endpoint in a picture is assigned a four-digit binary code, called a **region code**, that identifies the location of the point relative to the boundaries of the clipping rectangle. Regions are set up in reference to the boundaries as shown in Fig. 6-8. Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the clip window: to the left, right, top, or bottom. By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions can be correlated with the bit positions as

bit 1: left
bit 2: right
bit 3: below
bit 4: above

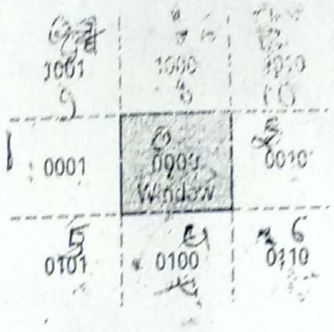


Figure 6-8
Binary region codes assigned to line endpoints according to relative position with respect to the clipping rectangle.

A value of 1 in any bit position indicates that the point is in that relative position; otherwise, the bit position is set to 0. If a point is within the clipping rectangle, the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

Bit values in the region code are determined by comparing endpoint coordinate values (x, y) to the clip boundaries. Bit 1 is set to 1 if $x < xw_{\min}$. The other three bit values can be determined using similar comparisons. For languages in which bit manipulation is possible, region-code bit values can be determined with the following two steps: (1) Calculate differences between endpoint coordinates and clipping boundaries. (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code. Bit 1 is the sign bit of $x - xw_{\min}$; bit 2 is the sign bit of $xw_{\max} - x$; bit 3 is the sign bit of $y - yw_{\min}$; and bit 4 is the sign bit of $yw_{\max} - y$.

Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside. Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially accept these lines. Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we trivially reject these lines. We would discard the line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code. A method that can be used to test lines for total clipping is to perform the logical *and* operation with both region codes. If the result is not 0000, the line is completely outside the clipping region.

Lines that cannot be identified as completely inside or completely outside a clip window by these tests are checked for intersection with the window boundaries. As shown in Fig. 6-9, such lines may or may not cross into the window interior. We begin the clipping process for a line by comparing an outside endpoint to a clipping boundary to determine how much of the line can be discarded. Then the remaining part of the line is checked against the other boundaries, and we continue until either the line is totally discarded or a section is found inside the window. We set up our algorithm to check line endpoints against clipping boundaries in the order left, right, bottom, top.

To illustrate the specific steps in clipping lines against rectangular boundaries using the Cohen-Sutherland algorithm, we show how the lines in Fig. 6-9 could be processed. Starting with the bottom endpoint of the line from P_1 to P_2,

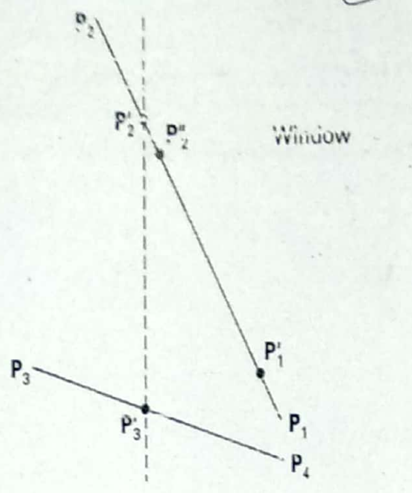


Figure 6-9
Lines extending from one coordinate region to another may pass through the clip window, or they may intersect clipping boundaries without entering the window.

we check P_1 against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle. We then find the intersection point P'_1 with the bottom boundary and discard the line section from P_1 to P'_1. The line now has been reduced to the section from P'_1 to P_2. Since P_2 is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point P'_2 is calculated, but this point is above the window. So the final intersection calculation yields P''_2, and the line from P'_1 to P''_2 is saved. This completes processing for this line, so we save this part and go on to the next line. Point P_3 in the next line is to the left of the clipping rectangle, so we determine the intersection P'_3 and eliminate the line section from P_3 to P'_3. By checking region codes for the line section from P'_3 to P_4, we find that the remainder of the line is below the clip window and can be discarded also.

Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates (x_1, y_1) and (x_2, y_2), the y coordinate of the intersection point with a vertical boundary
① can be obtained with the calculation

$$y = y_1 + m(x - x_1)$$

where the x value is set either to xw_{min} or to xw_{max}, and the slope of the line is calculated as $m = (y_2 - y_1)/(x_2 - x_1)$. Similarly, if we are looking for the intersection
② with a horizontal boundary, the x coordinate can be calculated as

$$x = x_1 + \frac{y - y_1}{m}$$

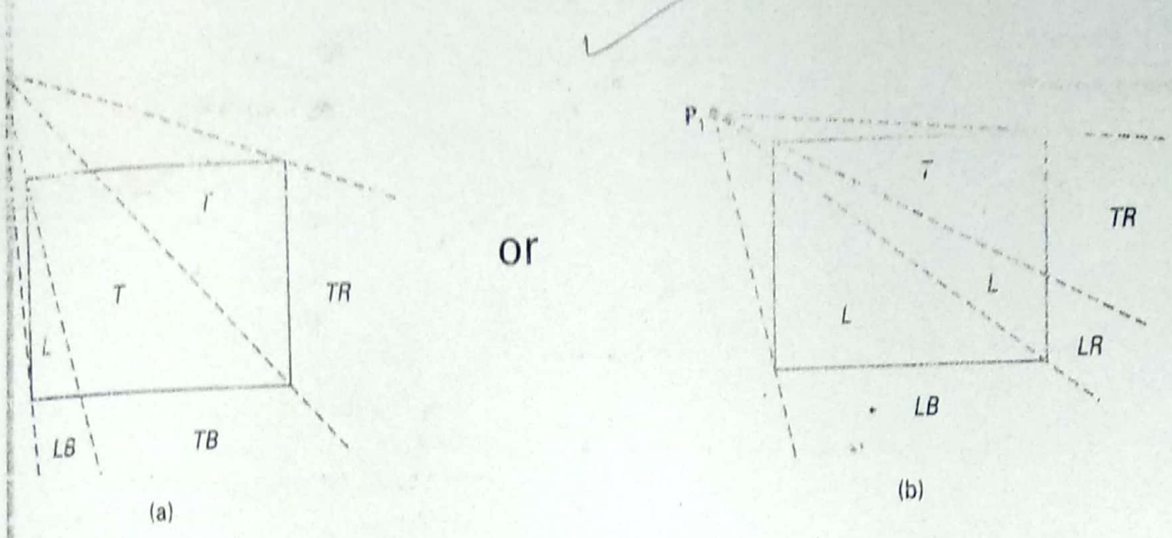
with y set either to yw_{min} or to yw_{max}.

The following procedure demonstrates the Cohen-Sutherland line-clipping algorithm. Codes for each endpoint are stored as bytes and processed using bit manipulations.

```
#define ROUND(a)     ((int)(a+0.5))

/* Bit masks encode a point's position relative to the clip edges.  A
   point's status is encoded by OR'ing together appropriate bit masks.
*/
#define LEFT_EDGE    0x1
```

45



gure 6-13
ie two possible sets of clipping regions used in the NLN algorithm when P_1 is above and the left of the clip window.

And an intersection position on the top boundary has $y = y_T$ and $u = (y_T - y_1)/(y_2 - y_1)$, with

$$x = x_1 + \frac{x_2 - x_1}{y_2 - y_1}(y_T - y_1) \tag{6-18}$$

Line Clipping Using Nonrectangular Clip Windows

In some applications, it is often necessary to clip lines against arbitrarily shaped polygons. Algorithms based on parametric line equations, such as the Liang–Barsky method and the earlier Cyrus–Beck approach, can be extended easily to convex polygon windows. We do this by modifying the algorithm to include the parametric equations for the boundaries of the clip region. Preliminary screening of line segments can be accomplished by processing lines against the coordinate extents of the clipping polygon. For concave polygon-clipping regions, we can still apply these parametric clipping procedures if we first split the concave polygon into a set of convex polygons.

Circles or other curved-boundary clipping regions are also possible, but less commonly used. Clipping algorithms for these areas are slower because intersection calculations involve nonlinear curve equations. At the first step, lines can be clipped against the bounding rectangle (coordinate extents) of the curved clipping region. Lines that can be identified as completely outside the bounding rectangle are discarded. To identify inside lines, we can calculate the distance of line endpoints from the circle center. If the square of this distance for both endpoints of a line is less than or equal to the radius squared, we can save the entire line. The remaining lines are then processed through the intersection calculations, which must solve simultaneous circle-line equations.

Splitting Concave Polygons

We can identify a concave polygon by calculating the cross products of successive edge vectors in order around the polygon perimeter. If the z component of

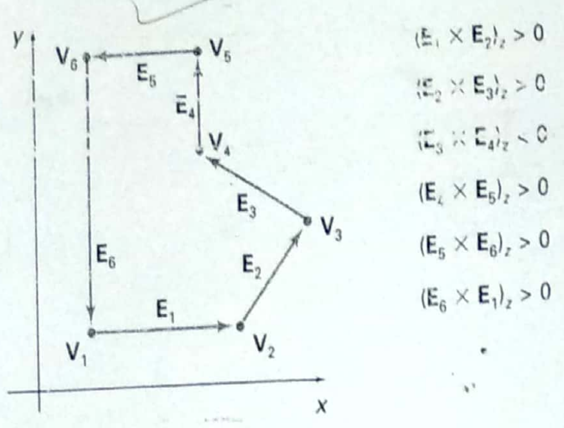


Figure 6-14
Identifying a concave polygon by calculating cross products of successive pairs of edge vectors.

some cross products is positive while others have a negative z component, we have a concave polygon. Otherwise, the polygon is convex. This is assuming that no series of three successive vertices are collinear, in which case the cross product of the two edge vectors for these vertices is zero. If all vertices are collinear, we have a degenerate polygon (a straight line). Figure 6-14 illustrates the edge-vector cross-product method for identifying concave polygons.

A *vector method* for splitting a concave polygon in the xy plane is to calculate the edge-vector cross products in a counterclockwise order and to note the sign of the z component of the cross products. If any z component turns out to be negative (as in Fig. 6-14), the polygon is concave and we can split it along the line of the first edge vector in the cross-product pair. The following example illustrates this method for splitting a concave polygon.

Example 6-2: Vector Method for Splitting Concave Polygons

Figure 6-15 shows a concave polygon with six edges. Edge vectors for this polygon can be expressed as

$$\mathbf{E}_1 = (1, 0, 0), \qquad \mathbf{E}_2 = (1, 1, 0)$$
$$\mathbf{E}_3 = (1, -1, 0), \qquad \mathbf{E}_4 = (0, 2, 0)$$
$$\mathbf{E}_5 = (-3, 0, 0), \qquad \mathbf{E}_6 = (0, -2, 0)$$

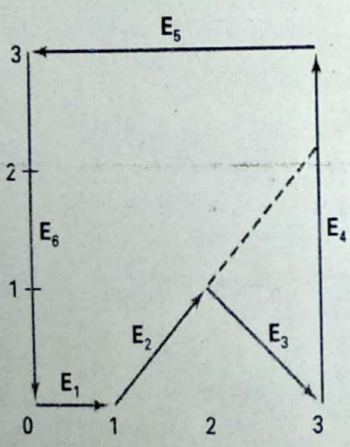


Figure 6-15
Splitting a concave polygon using the vector method.

where the z component is 0, since all edges are in the xy plane. The cross product $\mathbf{E}_i \times \mathbf{E}_j$ for two successive edge vectors is a vector perpendicular to the xy plane with z component equal to $E_{ix}E_{jy} - E_{jx}E_{iy}$.

$$\mathbf{E}_1 \times \mathbf{E}_2 = (0, 0, 1), \qquad \mathbf{E}_2 \times \mathbf{E}_3 = (0, 0, -2)$$
$$\mathbf{E}_3 \times \mathbf{E}_4 = (0, 0, 2), \qquad \mathbf{E}_4 \times \mathbf{E}_5 = (0, 0, 6)$$
$$\mathbf{E}_5 \times \mathbf{E}_6 = (0, 0, 6), \qquad \mathbf{E}_6 \times \mathbf{E}_1 = (0, 0, 2)$$

Since the cross product $\mathbf{E}_2 \times \mathbf{E}_3$ has a negative z component, we split the polygon along the line of vector $\mathbf{E}_2$. The line equation for this edge has a slope of 1 and a y intercept of -1. We then determine the intersection of this line and the other

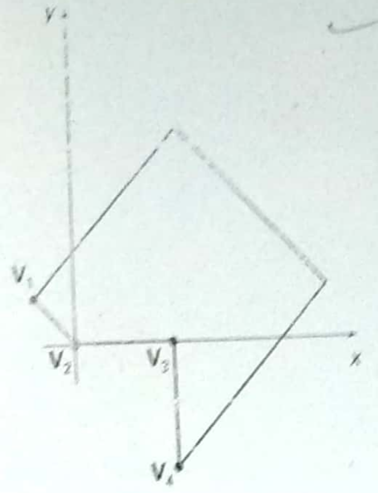


Figure 6-16
Splitting a concave polygon using the rotational method. After rotating V_3 onto the x axis, we find that V_4 is below the x axis. So we split the polygon along the line of $\overline{V_2V_3}$.

polygon edges to split the polygon into two pieces. No other edge cross products are negative, so the two new polygons are both convex.

We can also split a concave polygon using a *rotational method.* Proceeding counterclockwise around the polygon edges, we translate each polygon vertex V_k in turn to the coordinate origin. We then rotate in a clockwise direction so that the next vertex V_{k+1} is on the x axis. If the next vertex, V_{k+2}, is below the x axis, the polygon is concave. We then split the polygon into two new polygons along the x axis and repeat the concave test for each of the two new polygons. Otherwise, we continue to rotate vertices on the x axis and to test for negative y vertex values. Figure 6-16 illustrates the rotational method for splitting a concave polygon.

6-8 POLYGON CLIPPING

To clip polygons, we need to modify the line-clipping procedures discussed in the previous section. A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments (Fig. 6-17), depending on the orientation of the polygon to the clipping window. What we really want to display is a bounded area after clipping, as in Fig. 6-18. For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.

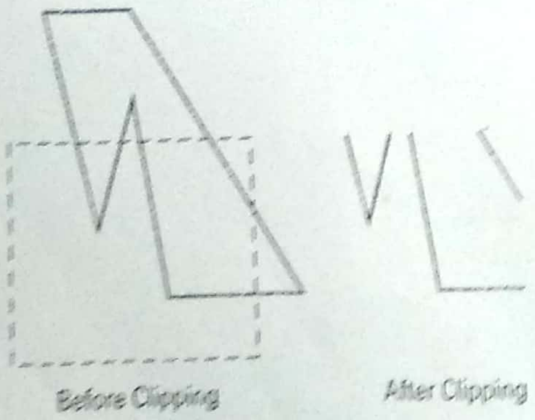


Figure 6-17
Display of a polygon processed by a line-clipping algorithm.

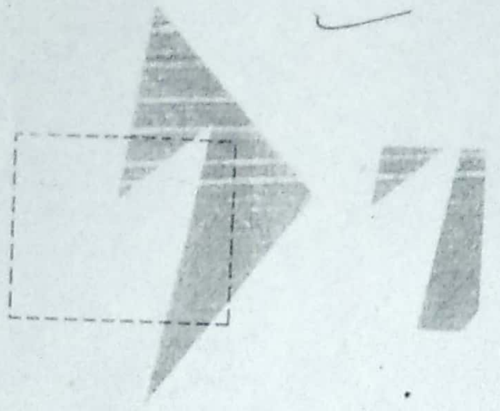


Figure 6-18
Display of a correctly clipped polygon.

Sutherland–Hodgeman Polygon Clipping ✓

We can correctly clip a polygon by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary in turn. Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then be successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper, as in Fig. 6-19. At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.

There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each pair of adjacent polygon vertices is passed to a window boundary clipper, we make the following tests: (1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list. (2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list. (3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list. (4) If both input vertices are outside the window boundary, nothing is added to the output list. These four cases are illustrated in Fig. 6-20 for successive pairs of polygon vertices. Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

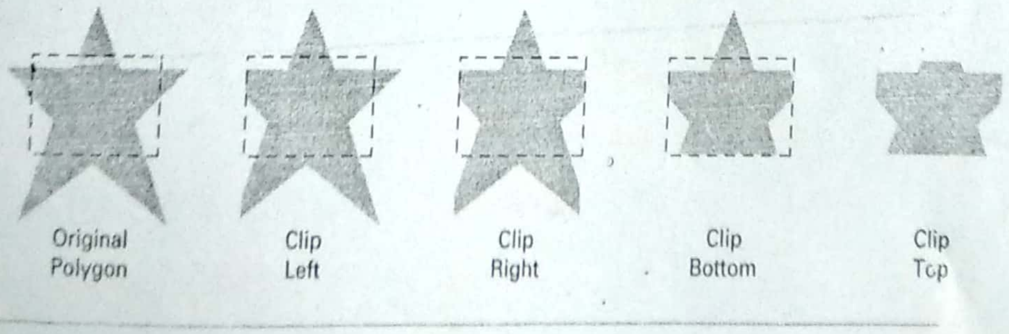


Figure 6-19
Clipping a polygon against successive window boundaries.

50

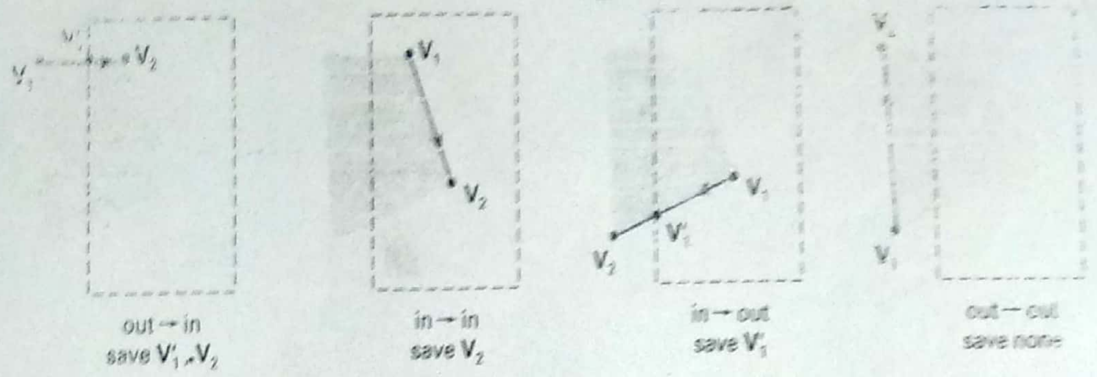


Figure 6-20
Successive processing of pairs of polygon vertices against the left window boundary.

We illustrate this method by processing the area in Fig. 6-21 against the left window boundary. Vertices 1 and 2 are found to be on the outside of the boundary. Moving along to vertex 3, which is inside, we calculate the intersection and save both the intersection point and vertex 3. Vertices 4 and 5 are determined to be inside, and they also are saved. The sixth and final vertex is outside, so we find and save the intersection point. Using the five saved points, we would repeat the process for the next window boundary.

Implementing the algorithm as we have just described requires setting up storage for an output list of vertices as a polygon is clipped against each window boundary. We can eliminate the intermediate output vertex lists by simply clipping individual vertices at each step and passing the clipped vertices on to the next boundary clipper. This can be done with parallel processors or a single processor and a pipeline of clipping routines. A point (either an input vertex or a calculated intersection point) is added to the output vertex list only after it has been been determined to be inside or on a window boundary by all four boundary clippers. Otherwise, the point does not continue in the pipeline. Figure 6-22 shows a polygon and its intersection points with a clip window. In Fig. 6-23, we illustrate the progression of the polygon vertices in Fig. 6-22 through a pipeline of boundary clippers.

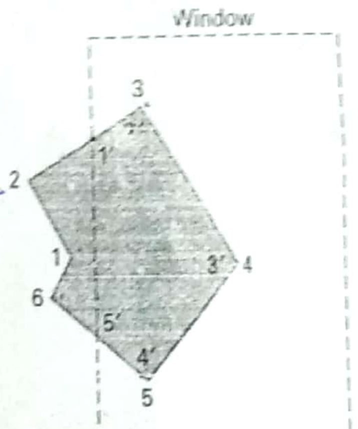


Figure 6-21
Clipping a polygon against the left boundary of a window, starting with vertex 1. Primed numbers are used to label the points in the output vertex list for this window boundary.

The following procedure demonstrates the pipeline clipping approach. An array, s, records the most recent point that was clipped for each clip-window boundary. The main routine passes each vertex *p* to the `clipPoint` routine for clipping against the first window boundary. If the line defined by endpoints *p* and `s[boundary]` crosses this window boundary, the intersection is calculated and passed to the next clipping stage. If *p* is inside the window, it is passed to the next clipping stage. Any point that survives clipping against all window boundaries is then entered into the output array of points. The array `firstPoint` stores for each window boundary the first point clipped against that boundary. After all polygon vertices have been processed, a closing routine clips lines defined by the first and last points clipped against each boundary.

```
typedef enum { Left, Right, Bottom, Top } Edge;
#define N_EDGE 4

int inside (wcPt2 p, Edge b, dcPt wMin, dcPt wMax)
{
  switch (b) {
```

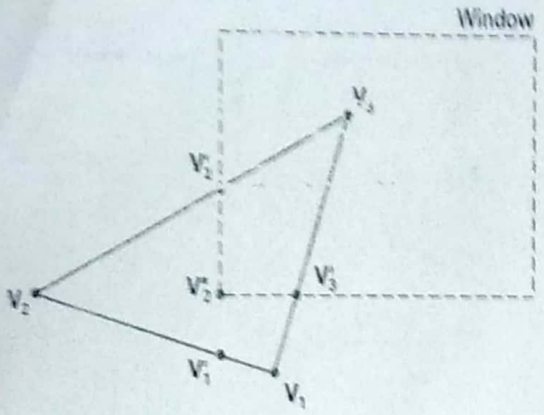


Figure 6-22
A polygon overlapping a rectangular clip window.

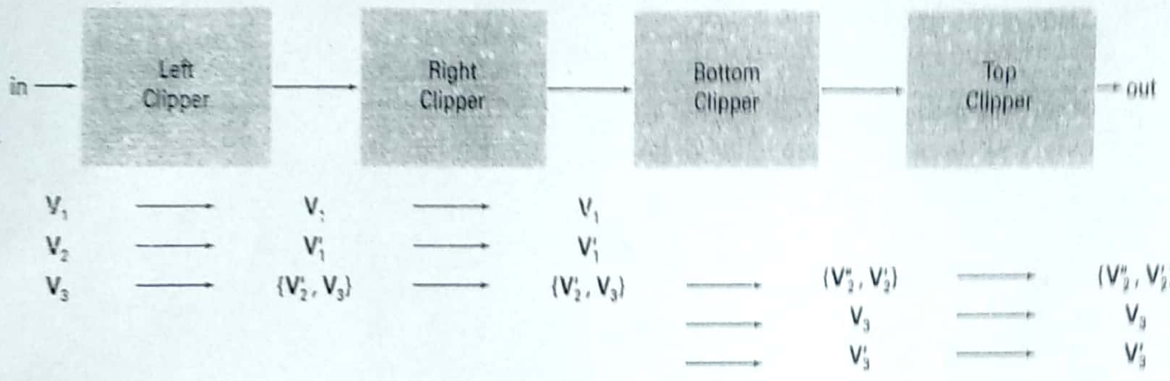

in
Left Clipper
Right Clipper
Bottom Clipper
Top Clipper
out
V_1 V_2 V_3
V_1 V'_1 $\{V'_2, V_3\}$
V_1 V'_1 $\{V'_2, V_3\}$
(V''_2, V'_2) V_3 V'_3
(V''_2, V'_2) V_3 V'_3

Figure 6-23
Processing the vertices of the polygon in Fig. 6-22 through a boundary-clipping pipeline. After all vertices are processed through the pipeline, the vertex list for the clipped polygon is $\{V''_2, V'_2, V_3, V'_3\}$.

```
  case Left:   if (p.x < wMin.x) return (FALSE); break;
  case Right:  if (p.x > wMax.x) return (FALSE); break;
  case Bottom: if (p.y < wMin.y) return (FALSE); break;
  case Top:    if (p.y > wMax.y) return (FALSE); break;
  }
  return (TRUE);
}

int cross (wcPt2 p1, wcPt2 p2, Edge b, dcPt wMin, dcPt wMax)
{
  if (inside (p1, b, wMin, wMax) == inside (p2, b, wMin, wMax))
    return (FALSE);
  else return (TRUE);
}

wcPt2 intersect (wcPt2 p1, wcPt2 p2, Edge b, dcPt wMin, dcPt wMax)
{
  wcPt2 iPt;
  float m;

  if (p1.x != p2.x) m = (p1.y - p2.y) / (p1.x - p2.x);
  switch (b) {
  case Left:
    iPt.x = wMin.x;
    iPt.y = p2.y + (wMin.x - p2.x) * m;
    break;
  case Right:
    iPt.x = wMax.x;
```

51

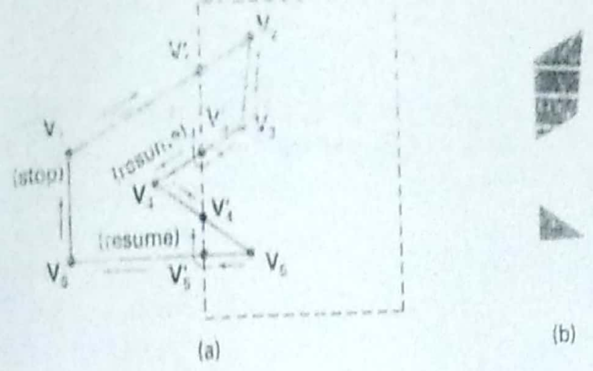


Figure 6-25
Clipping a concave polygon (a) with the Weiler–Atherton algorithm generates the two separate polygon areas in (b).

to-outside pair. For clockwise processing of polygon vertices, we use the following rules:

- For an outside-to-inside pair of vertices, follow the polygon boundary.
- For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction.

In Fig. 6-25, the processing direction in the Weiler–Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.

An improvement on the Weiler–Atherton algorithm is the Weiler algorithm, which applies constructive solid geometry ideas to clip an arbitrary polygon against any polygon-clipping region. Figure 6-26 illustrates the general idea in this approach. For the two polygons in this figure, the correctly clipped polygon is calculated as the intersection of the clipping polygon and the polygon object.]SR

Other Polygon-Clipping Algorithms ✓

Various parametric line-clipping methods have also been adapted to polygon clipping. And they are particularly well suited for clipping against convex polygon-clipping windows. The Liang–Barsky Line Clipper, for example, can be extended to polygon clipping with a general approach similar to that of the Sutherland–Hodgeman method. Parametric line representations are used to process polygon edges in order around the polygon perimeter using region-testing procedures similar to those used in line clipping.

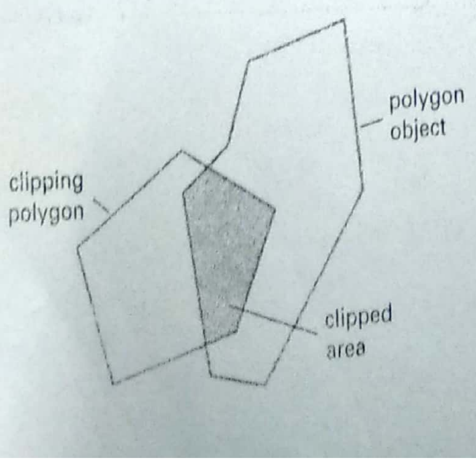
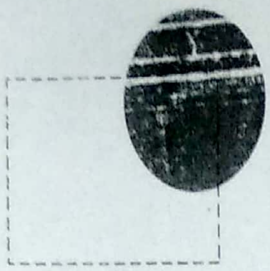


Figure 6-26
Clipping a polygon by determining the intersection of two polygon areas.

CURVE CLIPPING ✓



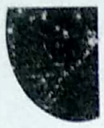


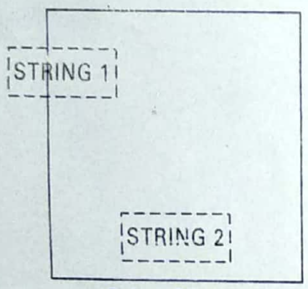
Figure 6-27
Clipping a filled circle.

Areas with curved boundaries can be clipped with methods similar to those discussed in the previous sections. Curve-clipping procedures will involve nonlinear equations, however, and this requires more processing than for objects with linear boundaries.

The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window. If the bounding rectangle for the object is completely inside the window, we save the object. If the rectangle is determined to be completely outside the window, we discard the object. In either case, there is no further computation necessary. But if the bounding rectangle test fails, we can look for other computation-saving approaches. For a circle, we can use the coordinate extents of individual quadrants and then octants for preliminary testing before calculating curve-window intersections. For an ellipse, we can test the coordinate extents of individual quadrants. Figure 6-27 illustrates circle clipping against a rectangular window.

Similar procedures can be applied when clipping a curved object against a general polygon clip region. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region. If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.

6-10 TEXT CLIPPING ✓



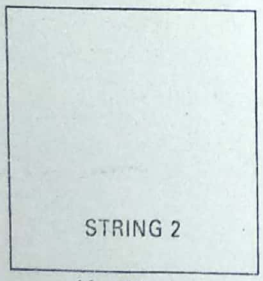


Figure 6-28
Text clipping using a bounding rectangle about the entire string.

There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.

The simplest method for processing character strings relative to a window boundary is to use the *all-or-none string-clipping* strategy shown in Fig. 6-28. If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern. The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping.

An alternative to rejecting an entire character string that overlaps a window boundary is to use the *all-or-none character-clipping* strategy. Here we discard only those characters that are not completely inside the window (Fig. 6-29). In this case, the boundary limits of individual characters are compared to the window. Any character that either overlaps or is outside a window boundary is clipped.

A final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines. If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window (Fig. 6-30). Outline character fonts formed with line segments can be processed in this way using a line-clipping algorithm. Characters defined with bit maps would be clipped by comparing the relative position of the individual pixels in the character grid patterns to the clipping boundaries. JSR

53

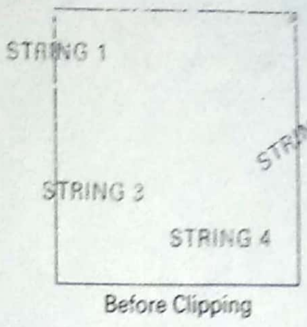
EXTERIOR CLIPPING ✓

So far, we have considered only procedures for clipping a picture to the interior of a region by eliminating everything outside the clipping region. What is saved by these procedures is *inside* the region. In some cases, we want to do the reverse, that is, we want to clip a picture to the exterior of a specified region. The picture parts to be saved are those that are *outside* the region. This is referred to as **exterior clipping**.

A typical example of the application of exterior clipping is in multiple-window systems. To correctly display the screen windows, we often need to apply both internal and external clipping. Figure 6-31 illustrates a multiple-window display. Objects within a window are clipped to the interior of that window. When other higher-priority windows overlap these objects, the objects are also clipped to the exterior of the overlapping windows.

Exterior clipping is used also in other applications that require overlapping pictures. Examples here include the design of page layouts in advertising or publishing applications or for adding labels or design patterns to a picture. The technique can also be used for combining graphs, maps, or schematics. For these applications, we can use exterior clipping to provide a space for an insert into a larger picture.

Procedures for clipping objects to the interior of concave polygon windows can also make use of external clipping. Figure 6-32 shows a line $\overline{\mathbf{P}_1\mathbf{P}_2}$ that is to be clipped to the interior of a concave window with vertices $\mathbf{V}_1\mathbf{V}_2\mathbf{V}_3\mathbf{V}_4\mathbf{V}_5$. Line $\overline{\mathbf{P}_1\mathbf{P}_2}$ can be clipped in two passes: (1) First, $\overline{\mathbf{P}_1\mathbf{P}_2}$ is clipped to the interior of the convex polygon $\mathbf{V}_1\mathbf{V}_2\mathbf{V}_3\mathbf{V}_4$ to yield the clipped segment $\overline{\mathbf{P}'_1\mathbf{P}'_2}$ (Fig. 6-32(b)). (2) Then an external clip of $\overline{\mathbf{P}'_1\mathbf{P}'_2}$ is performed against the convex polygon $\mathbf{V}_1\mathbf{V}_5\mathbf{V}_4$ to yield the final clipped line segment $\overline{\mathbf{P}''_1\mathbf{P}'_2}$.) SR



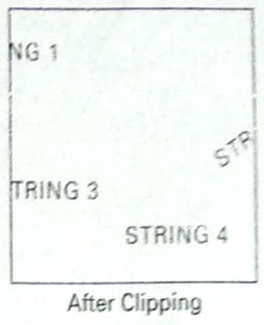
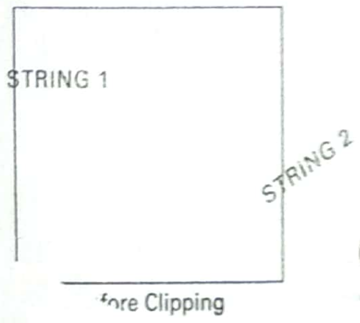


Figure 6-29
Text clipping using a bounding rectangle about individual characters.

SUMMARY

In this chapter, we have seen how we can map a two-dimensional world-coordinate scene to a display device. The viewing-transformation pipelir



246

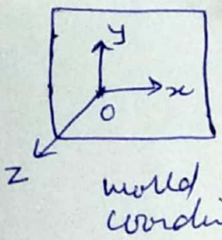
Figure 6-31
A multiple-wi
showing exa
and exterior
Sun Micros

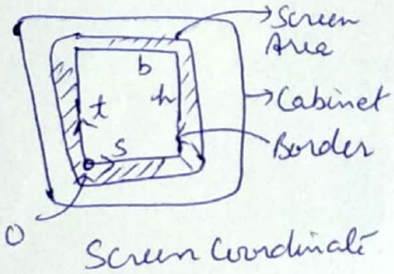
Computer Related coordinate system - Computer graphics uses the ~~Car~~ cartesian coordinate system for most input and for all internal processing and output.

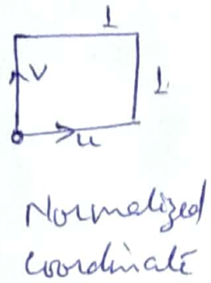
To translate the real world computer graphics we use 4 cartesian system of coordinate

1) World coordinate - x, y
2) Normalized screen coordinate - u, v
3) Screen coordinate - s, t
4) Device coordinate - p, q

1) World coordinate: Almost all work in graphics is based on objects from the real world. The usual right handed Cartesian system of (x, y, z) coordinates, with the z-axis directed towards viewer & is measured in length units. These are World Coordinates







② Screen Coordinate → By screen we mean the area of glass tube that is framed within plastic shell of the VDU. In Computer monitor a narrow border area is left on purpose, to define the rectangular area properly and to accommodate the quantitative nature of the image. The actual display area is a rectangular region, width b and height h, and the intermediate strip between picture & screen ~~ed~~ edges is known as Border.

③ Normalized screen Coordinates:- It is convenient in graphics to represent the location of points as ratio of corresponding overall width (b) and height (h) of display area. A benefit of such normalization (ratioed by maximum value) is that the same values can be used for monitors of different size, aspect ratio and resolution

$$\boxed{\begin{aligned} u &= s/b \\ v &= t/h \end{aligned}}$$

These ratio (u, v) are referred as Normalized screen coordinates

Object space - The window that we see out of from within an area represents our restricted perception of the world, generally defined in a rectangular format. The rectangular piece of the real world is defined in length dimension, preferably with reference to some convenient origin. this corresponds to object space.

Image space - In computer graphics, the term viewport is specifically defined as the area of the computer monitor screen within which anything is displayed. This corresponds to Image space or plotting Area.

Device coordinates - Most computer locate points and draw lines on command from a display controller which assigns intensity & color parameter for a pixel / block of pixels at a specified location of screen, these are called device coordinates

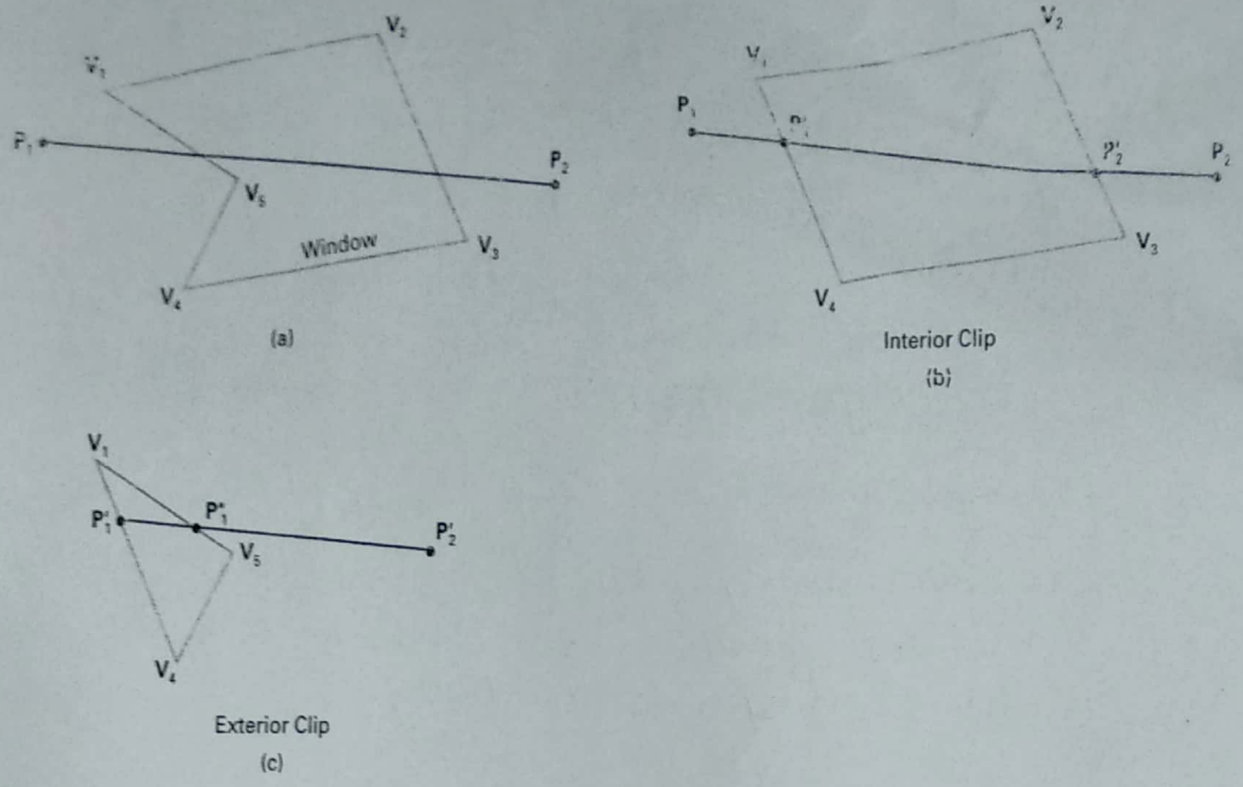


Figure 6-32
Clipping line $\overline{P_1P_2}$ to the interior of a concave polygon with vertices $V_1V_2V_3V_4V_5$ (a), using convex polygons $V_1V_2V_3V_4$ (b) and $V_1V_5V_4$ (c), to produce the clipped line $\overline{P''_1P'_2}$.

cludes constructing the world-coordinate scene using modeling transformation transferring world-coordinates to viewing coordinates, mapping the viewin coordinate descriptions of objects to normalized device coordinates, and final mapping to device coordinates. Normalized coordinates are specified in th range from 0 to 1, and they are used to make viewing packages independent particular output devices.

Viewing coordinates are specified by giving the world-coordinate positic of the viewing origin and the view up vector that defines the direction of th viewing y axis. These parameters are used to construct the viewing transform tion matrix that maps world-coordinate object descriptions to viewing coord nates.

A window is then set up in viewing coordinates, and a viewport is specifie in normalized device coordinates. Typically, the window and viewport are re tangles in standard position (rectangle boundaries are parallel to the coordina axes). The mapping from viewing coordinates to normalized device coordinat is then carried out so that relative positions in the window are maintained in th viewport.

Viewing functions in a graphics programming package are used to crea one or more sets of viewing parameters. One function is typically provided calculate the elements of the matrix for transforming world coordinates to viev ing coordinates. Another function is used to set up the window-to-viewpo transformation matrix, and a third function can be used to specify combinatio of viewing transformations and window mapping in a viewing table. We ca

55