Chapter 6 Viewing Transformation

6.1 Introduction

To display the view of a picture on to the display device, we make use of viewing transformation. Our picture is defined in one coordinate system which has to be mapped on device or screen coordinates. We will make use of various transformations we studied, like translation, scaling and rotation.

6.2 Coordinate Systems

The coordinate system used to address the screen are called **screen coordinates**. This is also called **device coordinates**. Another coordinate system called **world coordinates** system are user defined application specific coordinate system having its own units of measure, axis, origin, etc.

The rectangular region of the world that is visible is called **window**. To relate it with real scenario, window of our house shows only a part of the outside world scene. This small part visible from our window is to be displayed on the device. The rectangle region of screen space that is used to display the window is called **viewport**. The mapping of the world coordinate scene on the device coordinates is called **viewing transformation**. These concepts are better depicted in fig. 6.1.

We can define view port anywhere on the screen. Thus changing the position of viewport, we can view object at different position on the screen. We can also change the

size of the viewport thus changing the size of the object and proportions of the object on the display screen.

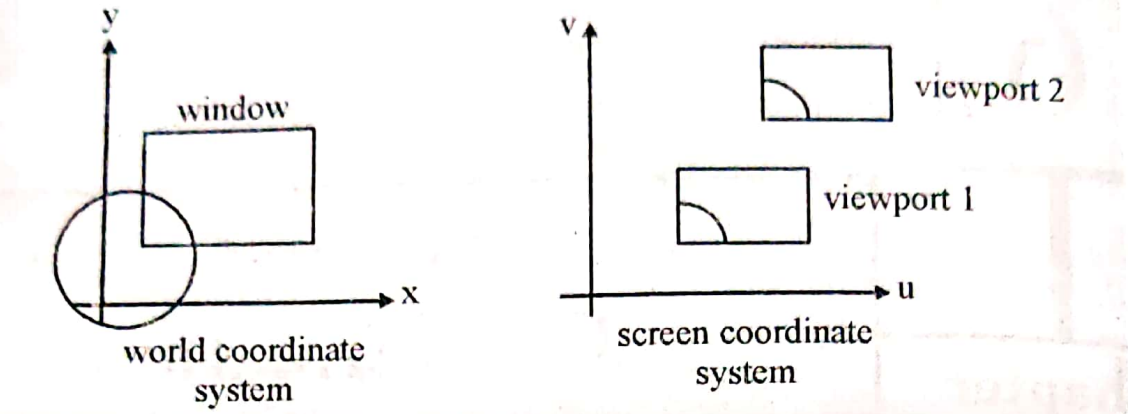


Fig. 6.1

As we can see in fig 6.1, only that portion of the window is displayed which is inside the window. Other part is clipped from the viewport.

6.3 Window to Viewport Transformation

Window and viewport generally have standard rectangle shapes, though they can have any orientation. A real world object which we see through the window is to be displayed on the screen through viewport. This transformation has few basic steps as shown in fig 6.2.

viewing pipeline

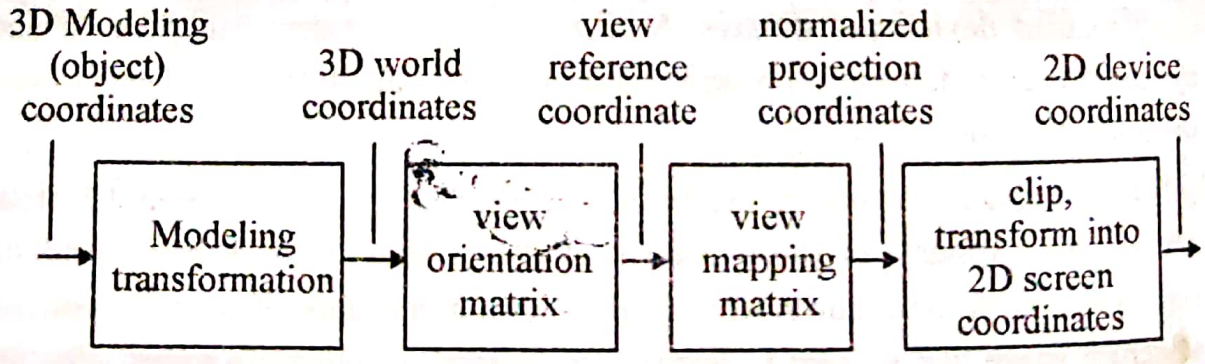


Fig. 6.2

The object is described in object coordinate system, also called modeling coordinate system. Next, to obtain a particular orientation for the window, set up a viewing system in 3D world coordinate plane. Then the world coordinates are transformed in viewing

coordinates. Next we go to normalized projection coordinate system or 3D screen coordinates. Here, normalized means that all coordinate values are either in the internal [0,1] or [-1,1]. Then Mapping from 3D to 2D creates 2D device coordinates, which includes clipping process also. Clipping we would be studying in the next unit.

Thus our sole purpose is to find the transformation matrix that maps the window in world coordinates to viewport in screen coordinates.

Window is denoted by (x, y space) with the coordinates x_{min}, y_{min}, x_{max}, y_{max}. Viewport is denoted by (u, v space) with the coordinates u_{min}, v_{min}, u_{max}, v_{max}. The transformation steps for mapping from window to viewport are:

- Translate the window to the origin.
- Scale it to the size of view port,
- translate it to the view port location.

Matrix representation for window to viewport transformation, M_{wv}:

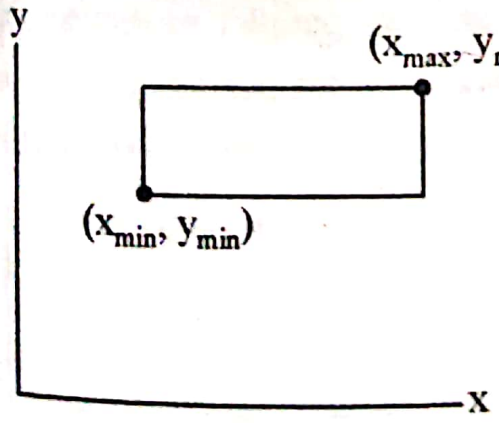
$$M_{wv} = T(u_{min},\ v_{min}) \cdot S(S_x,\ S_y) \cdot T(-x_{min},\ -y_{min})$$

where,
$$S_x = \frac{u_{max} - u_{min}}{x_{max} - x_{min}}; \quad S_y = \frac{v_{max} - v_{min}}{y_{max} - y_{min}}$$

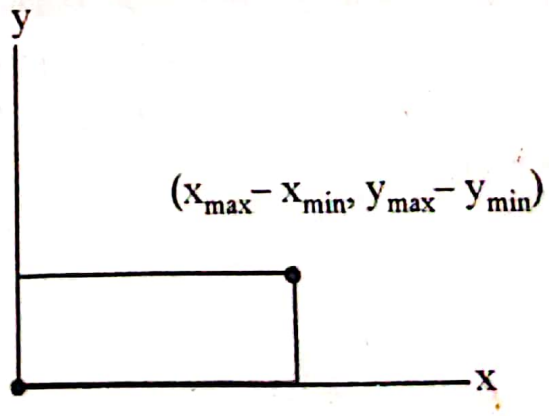
Thus window is first translated to the origin by $(-x_{min}, -y_{min})$, then scaled and then translated by (u_{min}, v_{min}), which is the position of viewport.

Thus,
$$M_{wv} = \begin{bmatrix} S_x & 0 & -x_{min} \cdot S_x + u_{min} \\ 0 & S_y & -y_{min} \cdot S_y + v_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

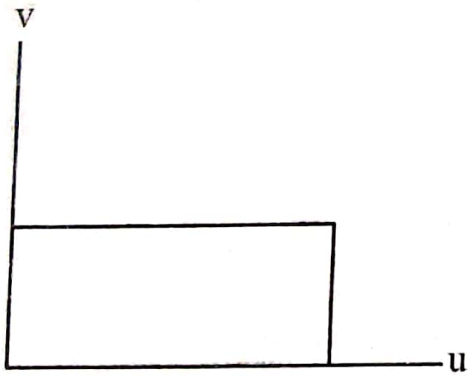
This transformation is graphically shown in fig 6.3



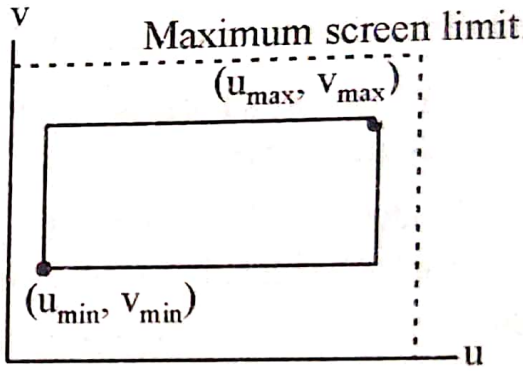
(a) Window in world coordinates



(b) Window translated to origin



(c) Scaling to the size of viewport



(d) Translate to the final viewport position

31 SATURDAY MAY

Matrix Representation of Window to viewport transformation

→ To viewport transformation eqn,

$M_{wv} = T(x_{min}, y_{min}) \cdot S(s_x, s_y) \cdot T(-x_{min}, -y_{min})$

$s_x = \frac{x_{max} - x_{min}}{x_{max} - x_{min}}$, $s_y = \frac{y_{max}}{y_{max}}$

y_{min}

y_{min}

→ Window is first translated to the origin by $(-x_{min}, -y_{min})$ then scaled and then translated by (x_{min}, y_{min}).

01 SUNDAY

Mo 7 14 21 28
Tu 1 8 15 22 29
We 2 9 16 23 30
Th 3 10 17 24 31
Fr 4 11 18 25
Sa 5 12 19 26
Su 6 13 20 27

JUL 2014

JUNE MONDAY WK 23 153-212

02

$$M_{wv} = \begin{bmatrix} s_x & 0 & -x_{min} s_x + x_{min} \\ 0 & s_y & -y_{min} s_y + y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

eg: Let us consider an example of a viewing transformation ~~which performs~~ (if our window has left- and right- boundaries of 3 and 5 & lower and upper boundaries of 0 and 4 then the first x'lation would be:-

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

03

154-211 WK 23
TUESDAY JUNE

Suppose the viewport is the upper right quadrant of the screen with boundaries at 0.5 & 1.0 for both x & y directions.

The length of window is 5 − 3 = 2 in the x direction

The length of viewport is

1.0 − 0.5 = 0.5

So the x scale factor is :-

$$\frac{0.5}{2} = 0.25$$

y scale factor : 0.5/4 = 0.125

So, scaling x' formation Matrix :-

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

2014

04

JUNE WEDNESDAY

Finally to position the viewport it requires a x'lation of

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

The viewing x' formation is then :-

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.125 & 0 \\ -0.25 & 0.5 & 1 \end{bmatrix}$$

In general the viewing x' formation matrix is :-

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

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

2014

Scanned with CamScanner

05

156-209 WK 23

THURSDAY | JUNE

MAY 2014

5	12	19	26		Mo
6	13	20	27		Tu
7	14	21	28		We
1	8	15	22	29	Th
2	9	16	23	30	Fr
3	10	17	24	31	Sa
4	11	18	25		Su

(a) $VXL - WXL$ $\quad \frac{(VXH - VXL)}{(WXH - WXL)}$

9

(b) $VYL - WYL$ $\quad \frac{(VYH - VYL)}{(WYH - WYL)}$

10

11

12

Clipping is defined as the process of identifying the objects of the scene that are outside the clipping region, which can be further removed or clipped from the viewing window. The region against which the object is to be clipped is called window. So, the window specifies what is to be shown or displayed. We will assume here that clip window is rectangular in shape, but it can be a polygon shaped also and can have even curved boundaries. Thus, the objects which are outside this rectangular clip window are discarded. Clipping algorithms that perform the process of clipping are of following types :

- Point Clipping
- Line clipping
- Polygon clipping
- Text clipping
- Curve clipping

Point, line and polygon clipping is discussed in this chapter. Other types are not discussed here. For curves, we do approximations with straight line segments and apply same clipping algorithms.

We also assume that maximum and minimum coordinates of clip window are (xw_{max}, yw_{max}) and (xw_{min}, yw_{min}) respectively.

7.2 Point Clipping

Assume a point $P(x, y)$ is to be displayed on the screen. We have to decide if this point lies within the clip window or not. Simple solution to this is to compare the point coordinates with window coordinates.

If $xw_{min} \leq x \leq xw_{max}$,

and $yw_{min} \leq y \leq yw_{max}$

Then point (x, y) lies within the window and can be displayed, otherwise discarded. Point clipping is rarely used as compared to other clipping, but it can be used in situations which involve particle movements such as explosion, dust, etc. Point clipping is shown fig. 7.1.

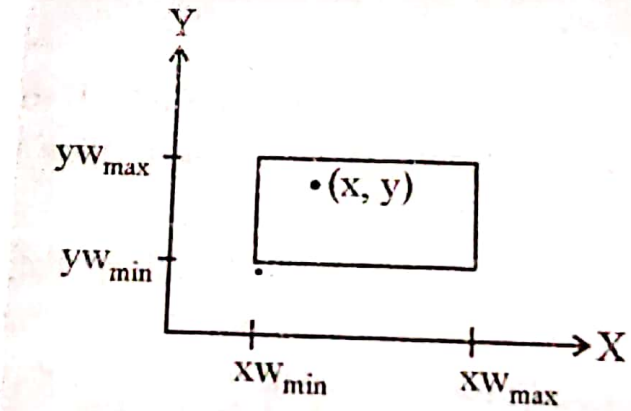


Fig. 7.1 Point clipping

7.3 Line Clipping

A line consists of series of number of points between two end points. We have to consider only the endpoints for clipping purpose, not the points between the endpoints. Observe fig. 7.2 if two endpoints of a line are within the clip window, it is trivially accepted. If one of the line end points is inside, and other end point is outside the clip window, we need to calculate the intersection of line with the edges of the rectangular window.

In fig. 7.2, line $P_1 P_2$ is completely inside the window. Line P_3P_4 has one endpoint outside the window, so we need to calculate the intersection point.

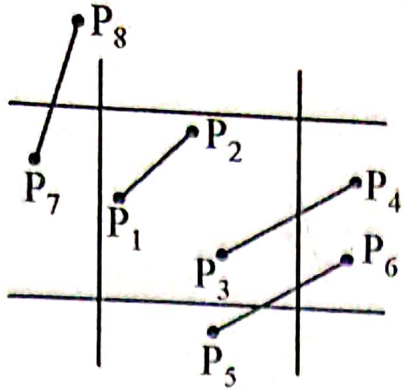


Fig. 7.2

There is a third case, where both endpoints are outside the window. In this case, line may or may not intersect with the window edges. Line P_5P_6 have both endpoints outside the window, but it intersects the window. Line P_7P_8 also has both endpoints outside the window, but it doesn't intersects the window and is completely outside the window. For such lines, we need to do more calculations, to check whether intersection occurs or not and if yes, find out those intersection points.

We follow certain algorithms so that we can identify the lines which are completely outside the window and reduce our calculation overhead. For this purpose we use parametric equation of line segment as

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

where P_0 and P_1 are the two line endpoints and $0 \leq t \leq 1$

Thus, using the value of t, we can determine the position of line. If t is beyond 0 and 1, line does not enter the clipping window and can be completely discarded, else if t is between 0 and 1, line passes through the window and we need to calculate the intersection point.

This method includes lot of computations and we have more efficient algorithms available which we will discuss next.

point.

7.3.2 Cohen-Sutherland Line Clipping Algorithm

This algorithm also reduces calculations, by identifying the lines which can be trivially accepted or rejected. This can be done by comparing the line endpoints with the window coordinates (X_{min}, Y_{min}) and (X_{max}, Y_{max}). For this purpose we assign 4-digit binary code to each endpoint of the line. As shown in Fig. 7.6, we have extended the window to get a plane of nine regions.

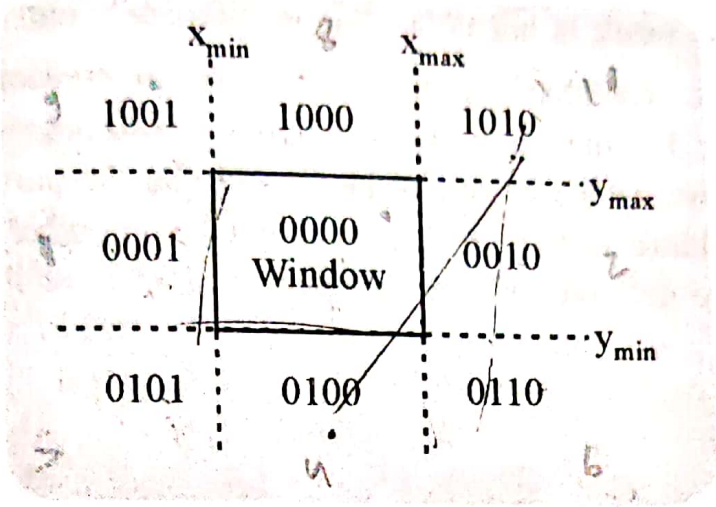


Fig. 7.6

Code for inside the window region is 0000. Each region is represented by 1 bit. First bit from left i.e., MSB is for the region above the top edge. If this bit is 1, it means point is above the top edge or $y > y_{max}$.

Second bit from left is for the region below the bottom edge. If this bit is 1, it means point is below the bottom edge or $y < y_{min}$.

Similarly for third bit, which designates right side portion of the right edge where $x > x_{max}$. And for LSB (fourth bit), point is to the left side of left edge where $x < x_{min}$.

Thus knowing the region code for each of the line endpoints, we can decide if we can trivially accept or reject a line.

Steps for the above processing are :

- Determine the bit values of two endpoints of the line to be clipped. To determine bit value, use the sign bit of the difference of coordinates of clipping boundary and line endpoint.

bit 1 (MSB) : sign $(y_{max} - y)$

bit 2 : sign $(y - y_{min})$

bit 3 : sign $(x_{max} - x)$

bit 4 (LSB) : sign $(x - x_{min})$

Use these endpoint codes to locate the line. Various possibilities are :

- If both endpoint codes are 0000, line is completely inside the box, three is no need to clip. This is simplest case as shown in fig. 7.7 line AB is completely inside the window.
- Any line has 1 in the same bit position of both endpoints, it is guaranteed to lie outside the box completely. For this, we can perform the logical AND operation with both region codes. If result is not 0000, line is completely outside the clipping region.
- If line is neither completely outside nor inside, it may or may not cross into the window interior. For this, at least one endpoint would be outside the window. Starting with this end, we compare with window, and clip the part outside that edge of the window. Remaining part is further compared with other edges or boundaries of rectangle till we get code of 0000 for both endpoints of line.

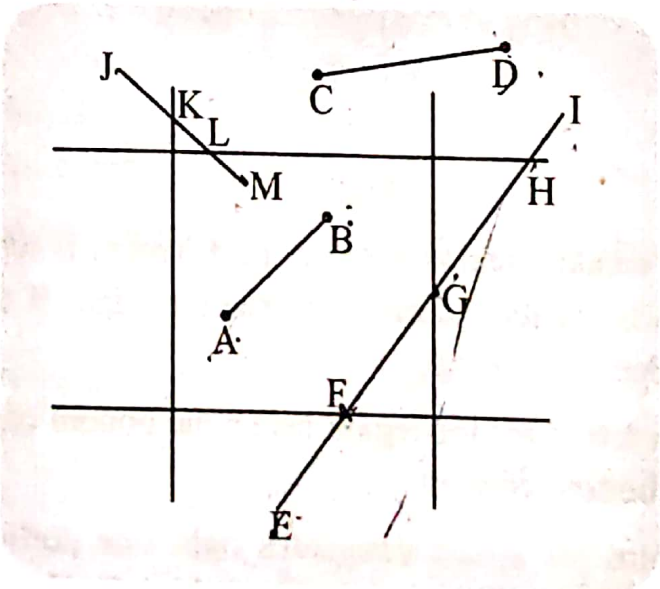


Fig. 7.7

Line AB has region code of 0000 for both endpoints, so it can be trivially accepted.

Line CD has code of 1000 for C and 1010 for D. On performing AND operation we don't get 0000 and it has 1 in MSB of both endpoints, so it can be trivially rejected.

Line EI has code of 0100 for E and 1010 for I, thus can't be trivially accepted or rejected. It may or may not be crossing the window. Starting from point E at the bottom, it is clipped to FI. Outcode of F is 0000 so no further clipping from this side. Now, starting

from I, it is clipped to H (line left is FH). H has out code of 0010, so it is clipped wrt right edge and reduced to FG. G has a code of 0000, so no further clipping is required. Line clipped is left with FG with region code of 0000 for both end points.

Intersection points of clipping boundary with the line can be calculated using slope-intercept form of the line equation. Lets assume, end points of a line are (x_0, y_0) and (x_1, y_1). So, intersection coordinates (x, y) for :

1. Top edge :

equation of line : $y - y_0 = m\ (x - x_0)$

where, $$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Coordinates of point of intersection is (x, y_{max}).

Putting it in above equation,

$$y_{max} - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

or $$(x - x_0) = (y_{max} - y_0)\frac{(x_1 - x_0)}{y_1 - y_0}$$

or $$x = x_0 + (x_1 - x_0) \cdot \left(\frac{y_{max} - y_0}{y_1 - y_0}\right) \qquad \text{.....(7.4)}$$

2. Bottom edge :

Coordinates of point of intersection is (x, y_{min}). Putting it in the equation of line,

$$y_{min} - y_0 = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x - x_0)$$

or $$(x - x_0) = \frac{(y_{min} - y_0)}{(y_1 - y_0)}(x_1 - x_0)$$

or $$x = x_0 + (x_1 - x_0) \cdot \frac{(y_{min} - y_0)}{(y_1 - y_0)} \qquad \text{.....(7.5)}$$

3. Left edge :

Coordinates of point of intersection are (x_{min}, y). Putting in the equation of line.

$$y - y_0 = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x_{min} - x_0)$$

or

$$y = y_0 + (y_1 - y_0) \cdot \frac{(x_{min} - x_0)}{(x_1 - x_0)} \quad \text{.....(7.6)}$$

4. Right edge :

Coordinates of point of intersection are (x_{max}, y). Putting in the equation of line, we get

$$y - y_0 = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x_{max} - x_0)$$

or

$$y = y_0 + (y_1 - y_0) \cdot \frac{(x_{max} - x_0)}{(x_1 - x_0)} \quad \text{.....(7.7)}$$

So, each time we clip a part of line against any edge, we update the corresponding endpoints of the line using above equations.

7.4 Polygon Clipping

Polygon is composed of several lines, thus we can say, polygon clipping involves line clipping. As shown in fig. 7.9, polygon is clipped against the rectangle window and we get a set of lines.

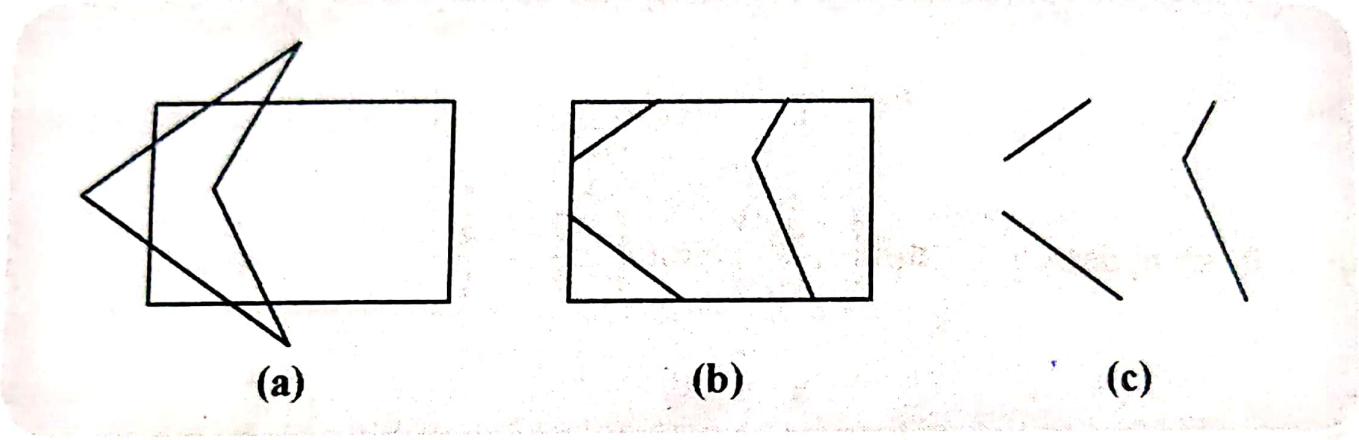


Fig. 7.9

What we wanted was a bounded polygon as the result of clipping, but we have few unconnected lines. Thus we want an algorithm other than line clipping algorithm which is efficient and give the result as required. Sutherland-Hodgman is the algorithm we will discuss.

7.4.1 Sutherland-Hodgman Algorithm

It is divide and conquer approach. We clip the polygon against each edge of the clip rectangle one by one. Each step adds new vertices to the list or removes or retain the existing vertices. Thus there is a change in the existing set of vertices of given polygon.

In fig. 7.9 (a) when polygon is clipped against right edge of the rectangle we have fig. 7.10 (a).

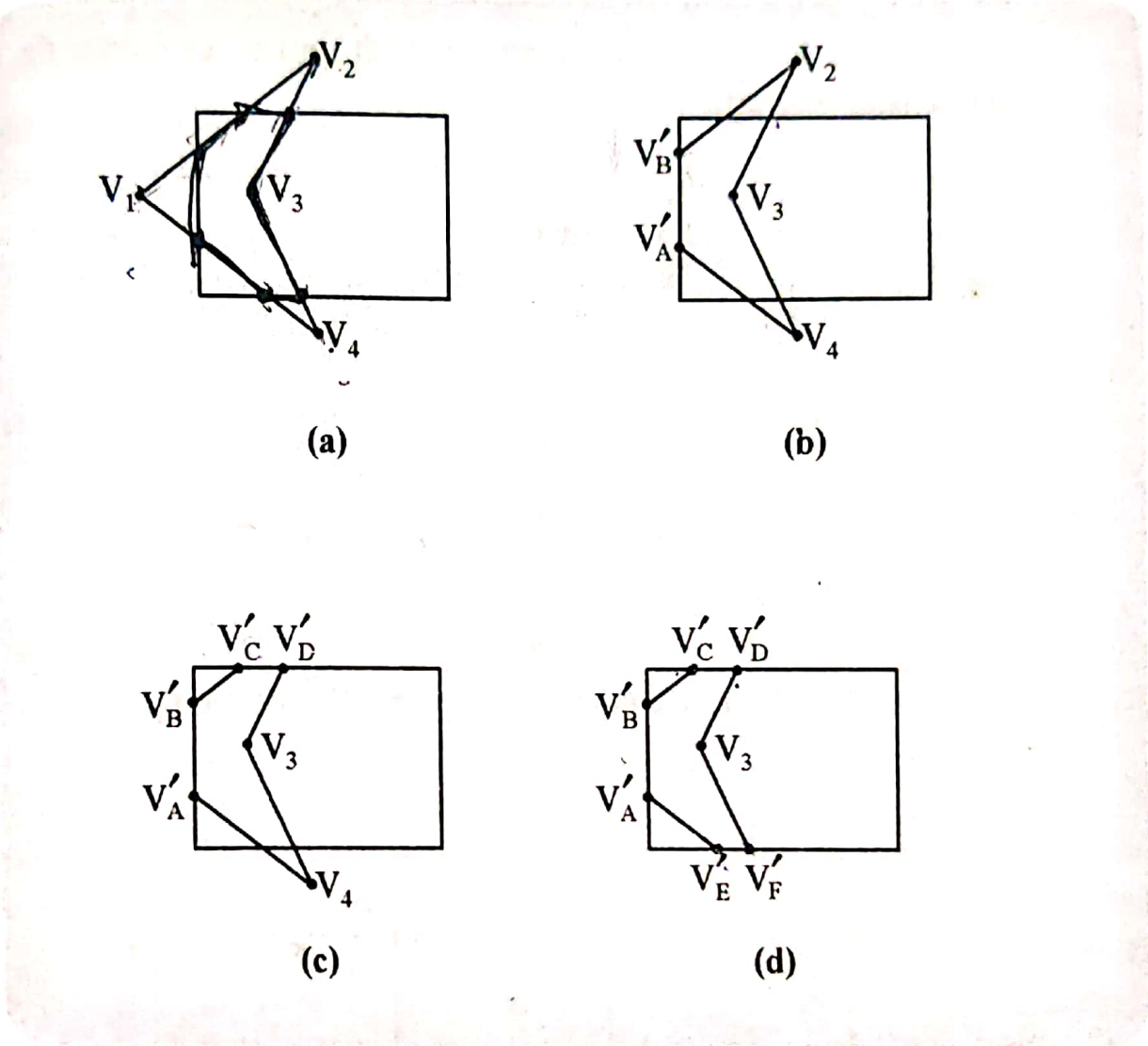
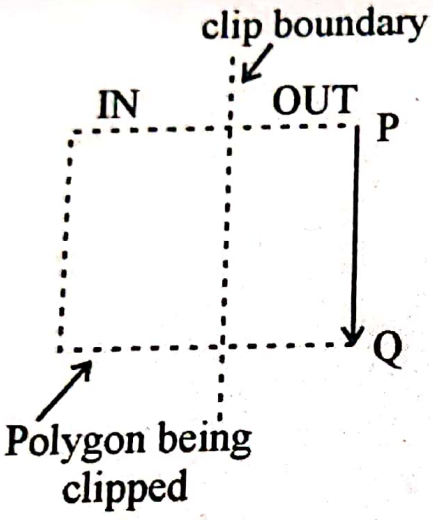


Fig. 7.10

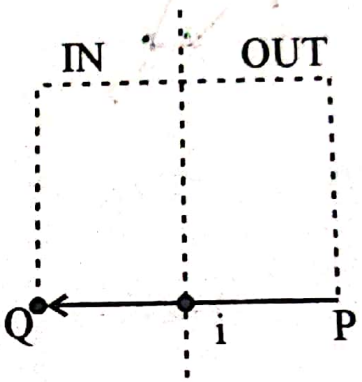
As we can see there is no change in polygon because no edge or vertex of the polygon is beyond right edge of the rectangle. If we clip against left boundary we have fig. 7.10 (b). We now have 5 vertices as compared to 4 vertices initially. If we clip further against top edge we have fig. 7.10 (c). We now have 6 vertices (V_2 is removed and V'_C and V'_D are added). Finally, on clipping against the bottom edge we have fig. 7.10 (d). Final set of vertices contain 7 vertices.

How the processing is done for polygon vertices against the boundary of the clipping rectangle is as follows :

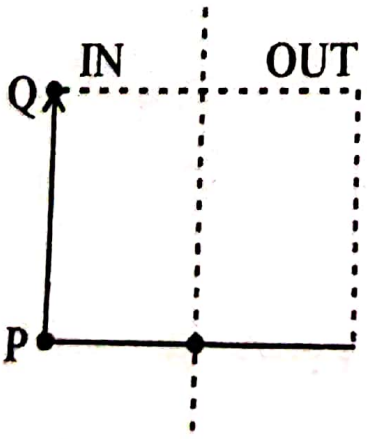
- As we process the vertices of polygon in say, clockwise direction, if we move from outside of the clip rectangle edge to the inside, we add in the vertex list the intersection point and second vertex which is inside.
- If we move from inside to inside, i.e., both vertex of polygon are inside the rectangle, we add in the vertex list only the second vertex of that polygon line.
- If we move from inside to outside, i.e., when first vertex is inside, then we add to the list of vertices only the intersection point.
- Finally, if we move from out to out, i.e., both vertices are outside the boundary, we ignore both the vertices.



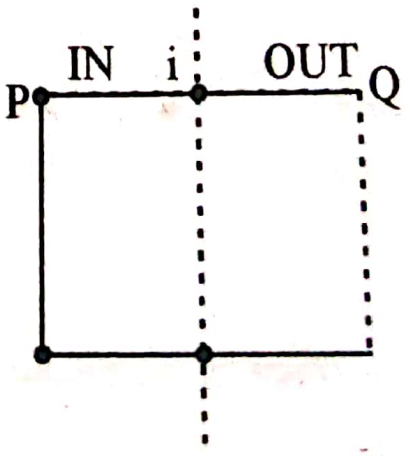
(a) OUTPUT:NULL



(b) OUTPUT:i, Q



(c) OUTPUT:Q



(d) OUTPUT:i

In fig. 7.11 (a), we are clipping polygon against right boundary of the rectangle. So, P and Q are vertices both outside the boundary. So, we ignore both the vertices. Next, processing the edge in fig. 7.11 (b) we move from OUT to IN, So we will add vertex i (*intersection* point) and second vertex Q in the list. In fig. 7.11 (c), we move from IN to IN, so we add only second vertex (Q) in the list.. In fig. 7.11 (d), we move from IN to OUT, thus we add intersection point i in the vertex list.

Thus we have final set of 4 vertices as shown in fig. 7.11 (d). Similarly the polygon can be clipped against left edge, top edge and bottom edge.

We have seen an example where after clipping we have single polygon component. If there are more than one components, then how should we deal with it, so that we have a single, bounded polygon as the final result. Observe in figure 7.12. After clipping we get two polygon components, connected with a single extra line (not bounded).

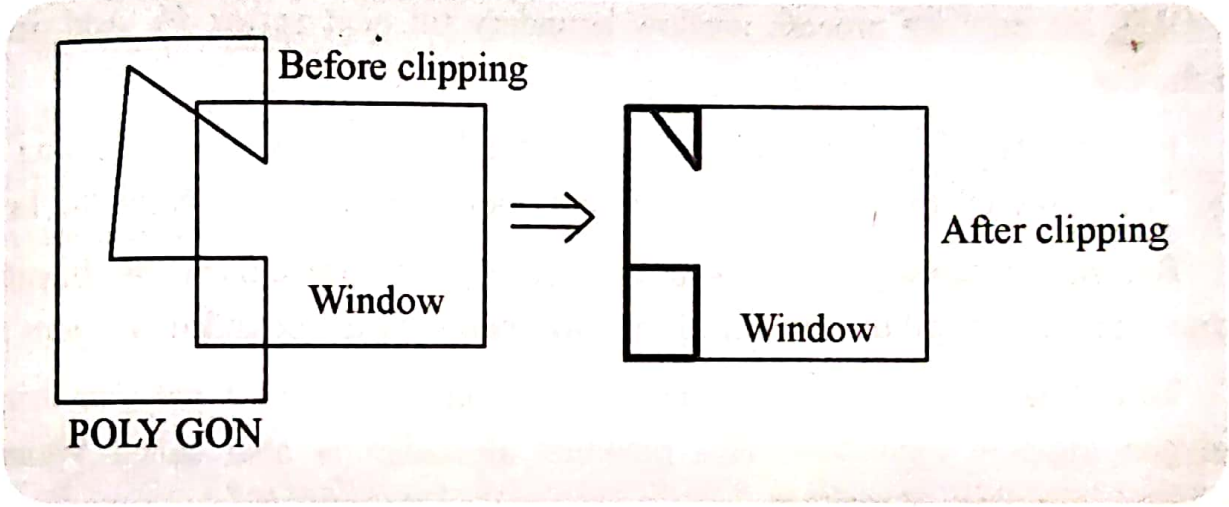


Fig. 7.12

For such cases we modify our polygon clipping algorithm. Basic idea behind it is that while processing vertices of the polygon, we not only follow polygon boundary but also window boundary sometimes. We will process the polygon in clockwise direction. If we are processing from outside window boundary to the inside, follow polygon boundary and if we are processing from inside to outside the boundary, follow window boundary in clockwise direction.

In fig. 7.13, we start processing polygon from vertex V_1 in clock wise direction. Edge V_1V_2 is processed. V_1 and V_2 both are OUT, so we ignore them. Processing V_2 to V_3, we move from OUT to IN, so we process polygon boundary and add intersection point V_2' and V_3 in the vertex list.

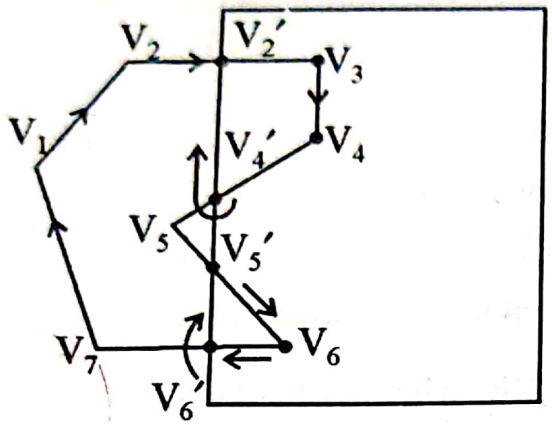


Fig 7.13

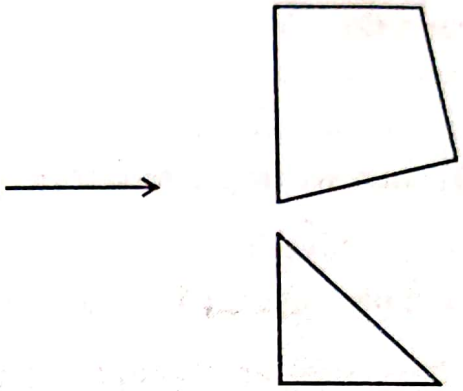


Fig. 7.14

Moving on like this, when we process from vertex V_4 to V_5, we are moving from IN to OUT. So now we process window boundary till next vertex V_2'. Add new vertex V_4' in the list.

Resume from V_5 vertex, add V_5', V_6 in the list. Again while processing from V_6 to V_7, we move IN to OUT, process window boundary and add V_6' in the list.

Resume processing from V_7 to V_1, both vertices are outside the boundary, so we ignore them. After all the processing we have two separate bounded polygons in fig. 7.14.

Thus now we can process and properly clip any sort of polygon using modified polygon clipping algorithm. This modified algorithm is also called **Weiler-Atherton polygon clipping algorithm.**