

Module - 3 (Clipping and Projections) Window to viewport transformation. Cohen Sutherland Line clipping algorithm. Sutherland Hodgeman Polygon clipping algorithm. Three dimensional viewing pipeline. Projections- Parallel and Perspective projections. Visible surface detection algorithms- Depth buffer algorithm, Scan line algorithm.

## THE VIEWING PIPELINE

- ⇒ **Window**: A world-coordinate area selected for display.  
The window defines what is to be viewed
- ⇒ **Viewport** : An area on a display device to which a window is mapped. The **viewport** defines where it is to be displayed.
- ⇒ Windows and viewports are **rectangles** in standard position, with the rectangle edges parallel to the coordinate axes.
- ⇒ **Viewing transformation**:
  - The mapping transformation is a world-coordinate window-viewport to device coordinates. or the to-windowing transformation.

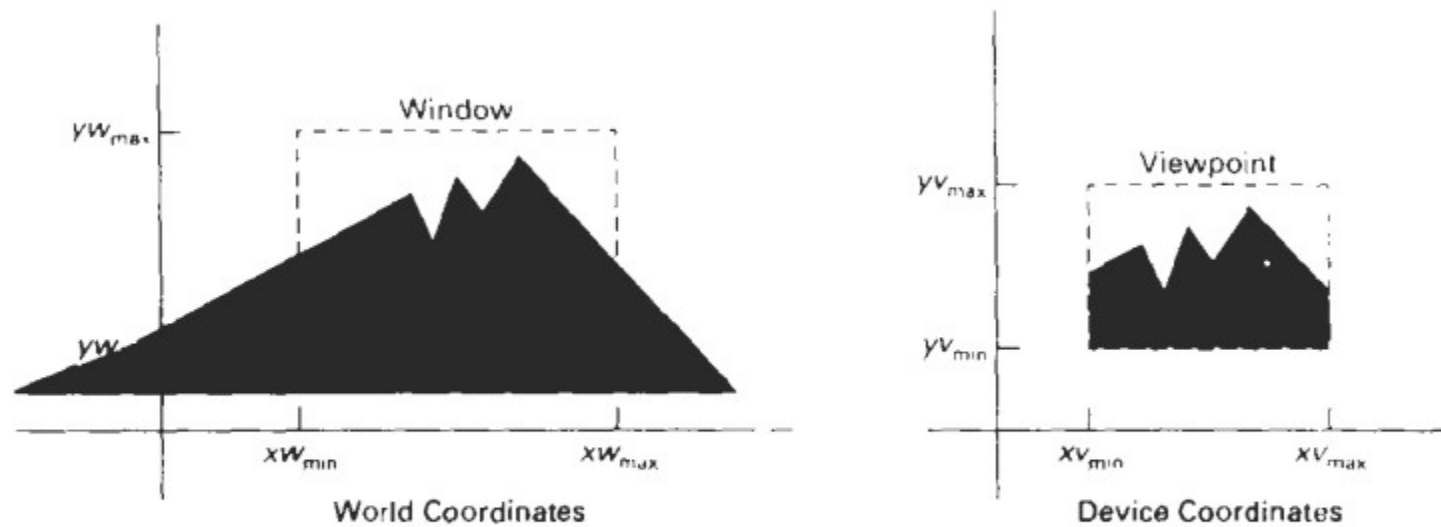
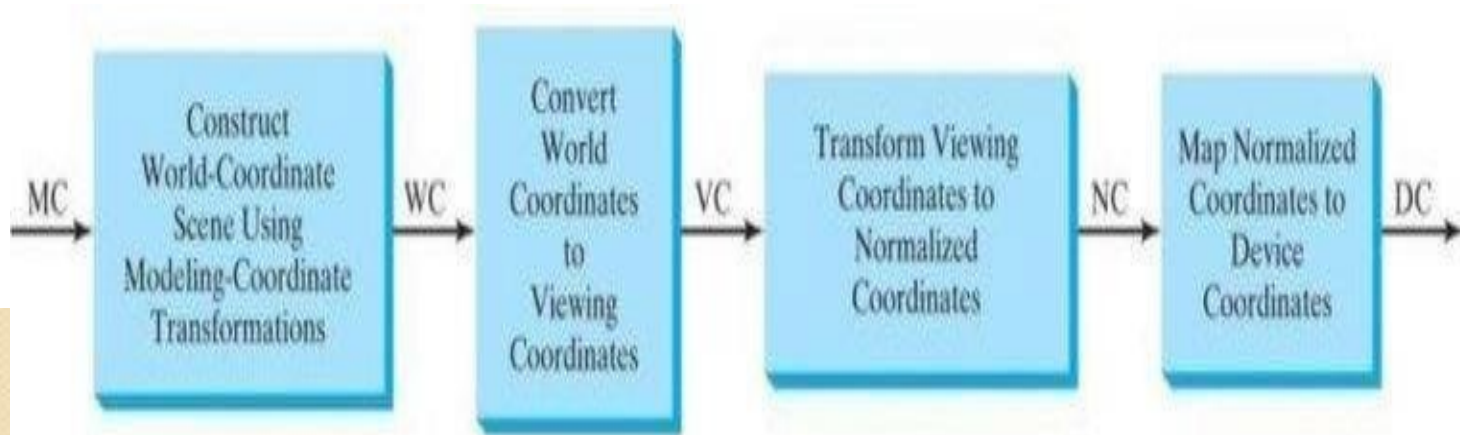


Figure 6-1

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

# Steps in 2D viewing pipeline



The viewing transformation is carried out in several steps.

- ⇒ First, we construct the scene in world coordinates using the output primitives and attributes.
- ⇒ 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 are outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates.

# WINDOW-TO-VIEWPORT COORDINATE TRANSFORMATION

- ⇒ Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates.
- ⇒ Object descriptions are then transferred to normalized device coordinates.
- ⇒ We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates.
- ⇒ If a coordinate position is at the center of the viewing window, for instance, it will be displayed at the center of the viewport.

We obtain the matrix for converting world-coordinate positions to viewing coordinates as a two-step composite transformation: First, we translate the viewing origin to the world origin, then we rotate to align the two coordinate reference frames. The composite two-dimensional transformation to convert world coordinates to viewing coordinates is

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T} \quad (6-1)$$

where  $\mathbf{T}$  is the translation matrix that takes the viewing origin point  $P_0$  to the world origin, and  $\mathbf{R}$  is the rotation matrix that aligns the axes of the two reference frames. Figure 6-4 illustrates the steps in this coordinate transformation.



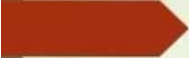
*Figure 6-5*

A point at position  $(xw, yw)$  in a designated window is mapped to viewport coordinates  $(xv, yv)$  so that relative positions in the two areas are the same.

Figure 6-5 illustrates the window-to-viewport mapping. A point at position  $(xw, yw)$  in the window is mapped into position  $(xv, yv)$  in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that

$$\begin{aligned} \frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} &= \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}} \\ \frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} &= \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}} \end{aligned} \quad (6-2)$$





$$\frac{X_w - X_{wmin}}{X_{wmax} - X_{wmin}} = \frac{X_v - X_{vmin}}{X_{vmax} - X_{vmin}} \longrightarrow \textcircled{1}$$

$$\frac{Y_w - Y_{wmin}}{Y_{wmax} - Y_{wmin}} = \frac{Y_v - Y_{vmin}}{Y_{vmax} - Y_{vmin}} \longrightarrow \textcircled{2}$$

From (1)  $X_v = \frac{X_{vmax} - X_{vmin}}{X_{wmax} - X_{wmin}} \times (X_w - X_{wmin}) + X_{vmin}$


From (2)  $Y_v = \frac{Y_{vmax} - Y_{vmin}}{Y_{wmax} - Y_{wmin}} \times (Y_w - Y_{wmin}) + Y_{vmin}$

So, corresponding to a window point  $(X_w, Y_w)$ , the view port point  $(X_v, Y_v)$  is

$$\left\{ \begin{array}{l} \frac{X_{vmax} - X_{vmin}}{X_{wmax} - X_{wmin}} \\ \frac{Y_{vmax} - Y_{vmin}}{Y_{wmax} - Y_{wmin}} \end{array} \right\} \begin{array}{l} * (X_w - X_{wmin}) + X_{vmin} \\ * (Y_w - Y_{wmin}) + Y_{vmin} \end{array} ,$$

↑ window to view port scaling factor of X,  $S_x$

↑ window to view port scaling factor of Y,  $S_y$


$$\text{So, } X_v = S_x ( X_w - X_{wmin} ) + X_{vmin}$$

Scaling factor

Translation factor

$$\& \ Y_v = S_y ( Y_w - Y_{wmin} ) + Y_{vmin}$$

So, window to view port transformation involves  
**SCALING AND TRANSLATION**

$$X_v = S_x ( X_w - X_{wmin} ) + X_{vmin}$$

$$Y_v = S_y ( Y_w - Y_{wmin} ) + Y_{vmin}$$

Equations 6-3 can also be derived with a set of transformations that converts the window area into the viewport area. This conversion is performed with the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of  $(xw_{min}, yw_{min})$  that scales the window area to the size of the viewport.
2. Translate the scaled window area to the position of the viewport.

# Clipping

- Any procedure that identifies those portions of a picture that are either inside or outside of a specified region of a space is referred to as **clipping**.
- The region against which an object is to be clipped is called a **clip window**. Depending on the application a clip window can be polygon or curved boundaries.
- World- coordinate clipping removes the primitives outside the window from further consideration; thus eliminating the processing necessary to transform these primitives to device space.
- **Clipping type-** point ,line, polygon, curved areas and etc.

## Contd...

- 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 multi-window environment
  - Drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating.

- Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates called **world coordinates clipping**
- Alternatively, the complete world-coordinate picture can be mapped first to device coordinates, or normalized device coordinates, then clipped against the viewport boundaries called **viewport clipping**
- 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 requires the transformation to device coordinates be performed for all objects, including those outside the window area..

Different types of clipping are

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



## POINT CLIPPING

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

$$\begin{aligned}xw_{\min} &\leq x \leq xw_{\max} \\yw_{\min} &\leq y \leq yw_{\max}\end{aligned}\tag{6-5}$$

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.

**Contd...**

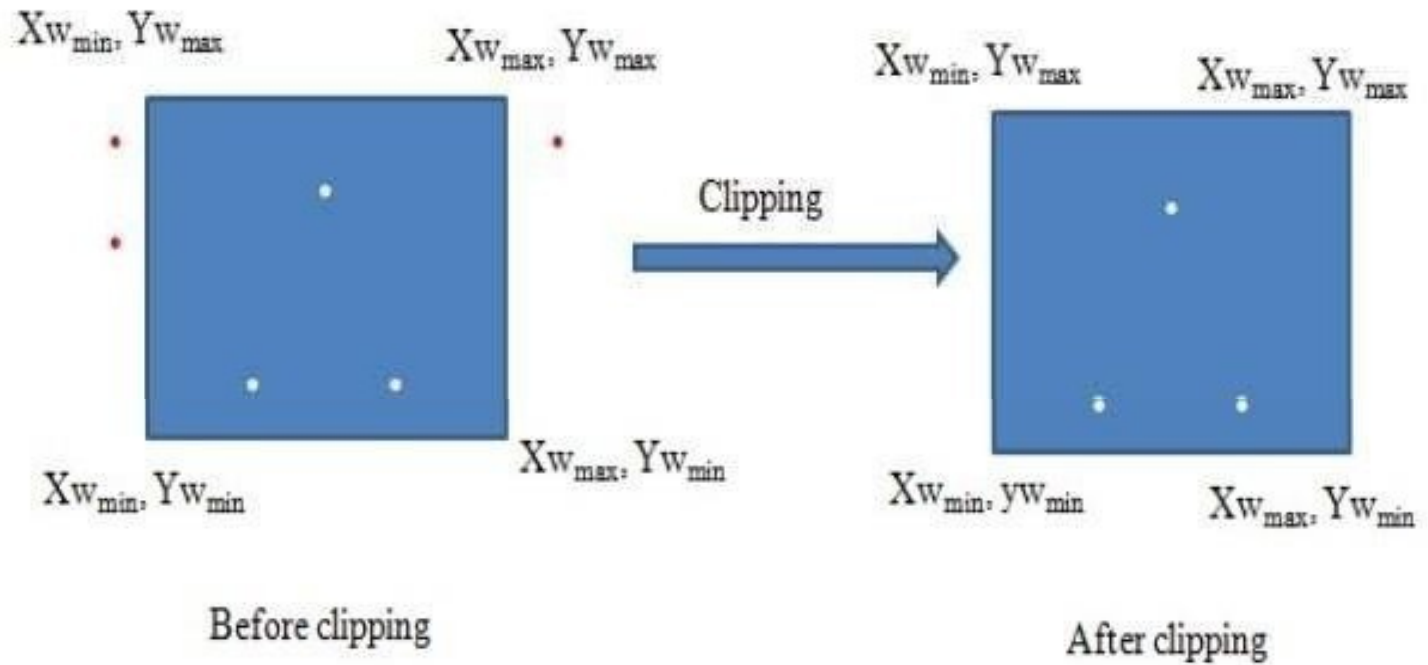


Fig: Point Clipping process

# Line clipping

- The visible segment of a straight line can be determined by **inside – outside test**:
  - A line with both endpoints **inside** clipping boundary, such as the line from  $p_1$  to  $p_2$ , is saved.

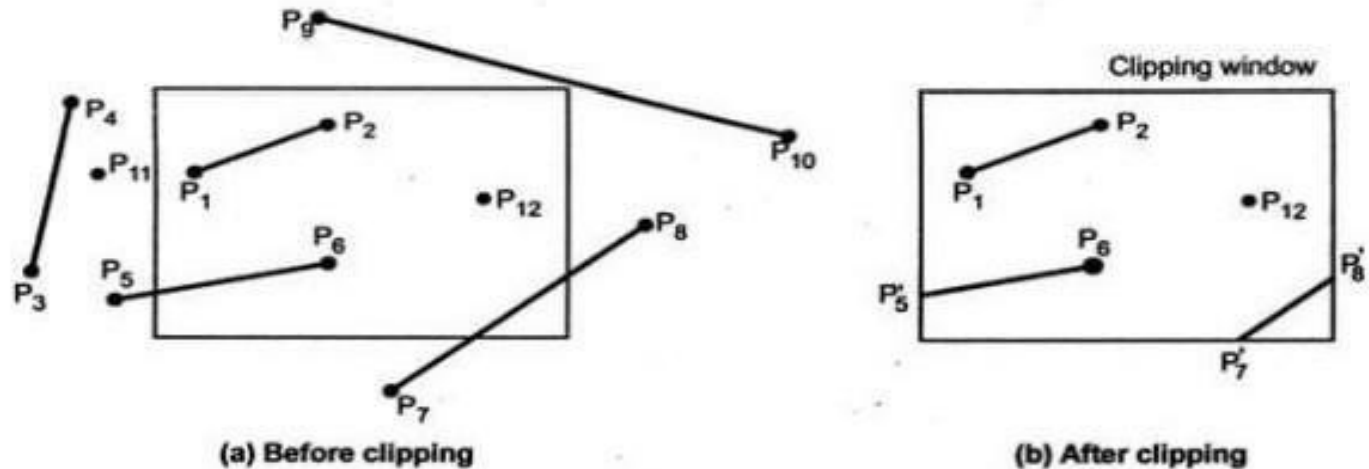


Fig. (e)

## Contd...

- A line with both endpoints **outside** the clip boundary, such as the line from  $p_3$  to  $p_4$ , is not saved.

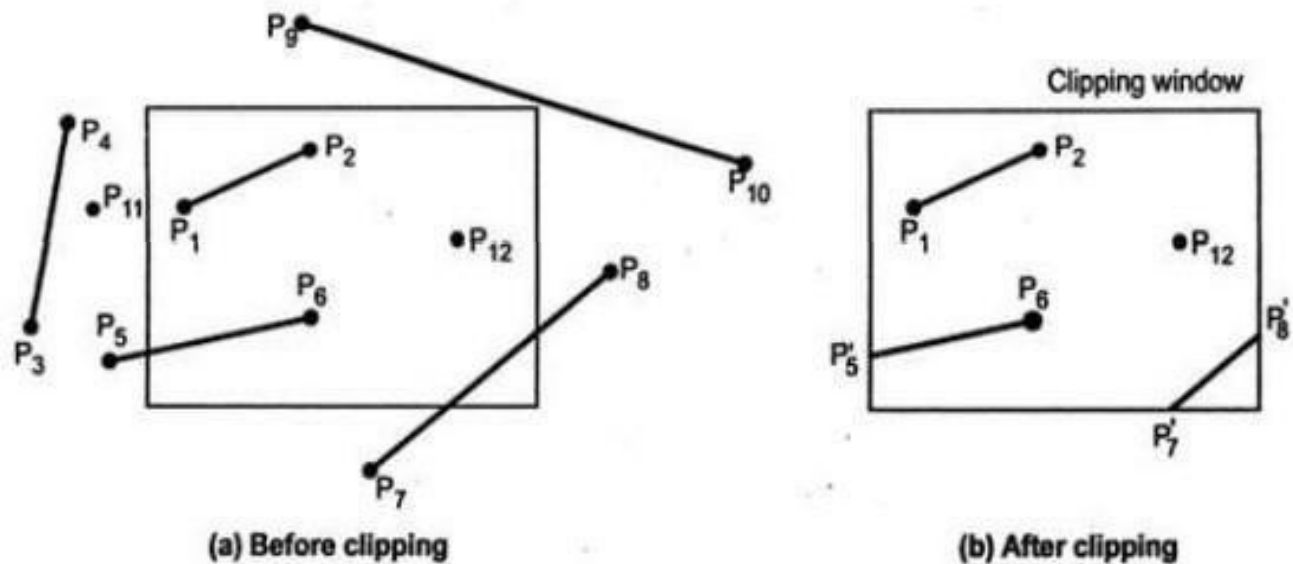


Fig. (e)

## Contd...

- If the line is not completely inside or completely outside (e.g.,  $p_7$  to  $p_8$ ), then perform intersection calculations with one or more clipping boundaries.

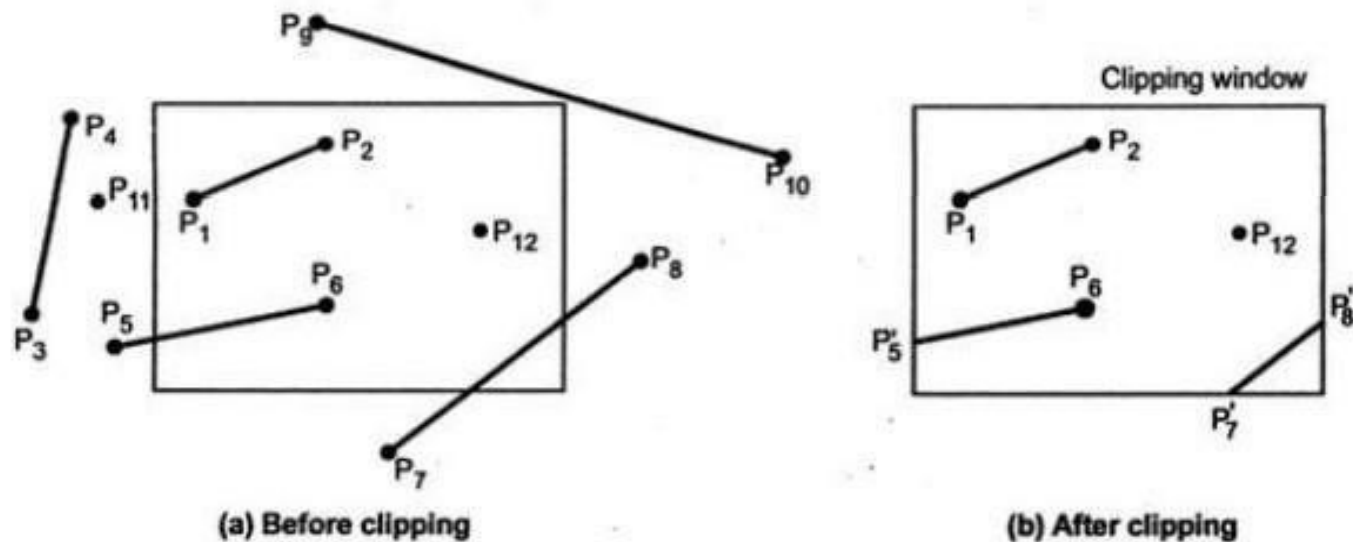


Fig. (e)

- 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 P1 to P2 is saved.

- A line with both endpoints outside any one of the clip boundaries line P3P4, is outside the window.
- 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 can 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

$$x = x_1 + u(x_2 - x_1)$$

$$y = y_1 + u(y_2 - y_1) \text{ where } 0 \leq u \leq 1$$

Could be used to determine values of parameter  $u$  for intersections with 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.

# Line clipping algorithms

⇒ Cohen Sutherland algorithm



# Cohen Sutherland algorithm

- 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 end point 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.
- 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 position

- 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  $x < x_{wmin}$
- Bit 2 is set to 1 if  $x > x_{wmax}$
- Bit 3 is set to 1 if  $y < y_{wmin}$
- Bit 4 is set to 1 if  $y > y_{wmax}$

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  $x - x_{wmin}$

bit 2 is the sign bit of  $x_{wmax} - x$ ;

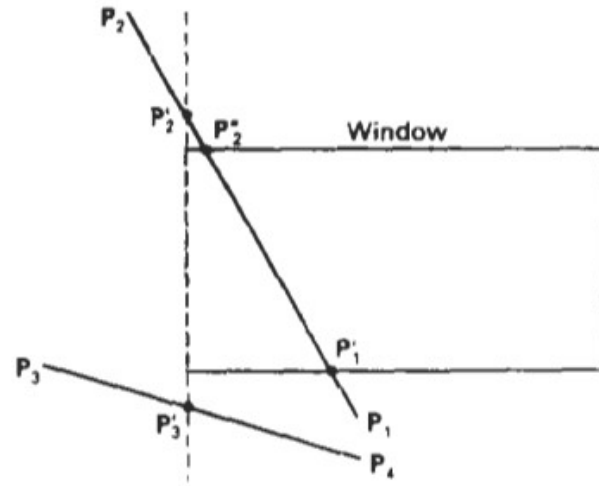
bit 3 is the sign bit of  $y - y_{wmin}$

bit 4 is the sign bit of  $y_{wmax} - y$ .

**Negative sign bit 1**

**Positive sign bit 0**

- After establishing 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 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



- we show how the lines in Fig could be processed. Starting with the bottom endpoint of the line from  $P_1$  to  $P_2$
- 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  $P_2''$

# Cohen-Sutherland Line Clipping

- This algorithm clips the line by performing following steps:
- **Step1: Region code assignment**
  - divide the whole picture region into nine regions by extending the window boundaries as shown in figure below and then assign a 4-bit region code to each region as follows:

## Rules For assigning region code:

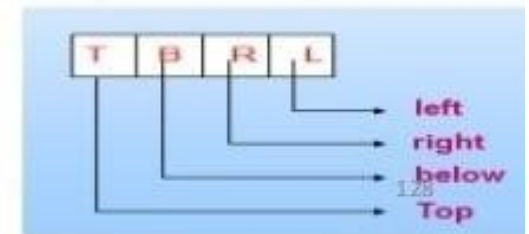
T=1, if the region is above the window,  
= 0, otherwise.

B= 1, if the region is below the window,  
= 0, otherwise.

R=1, if the region is right of window,  
= 0, otherwise.

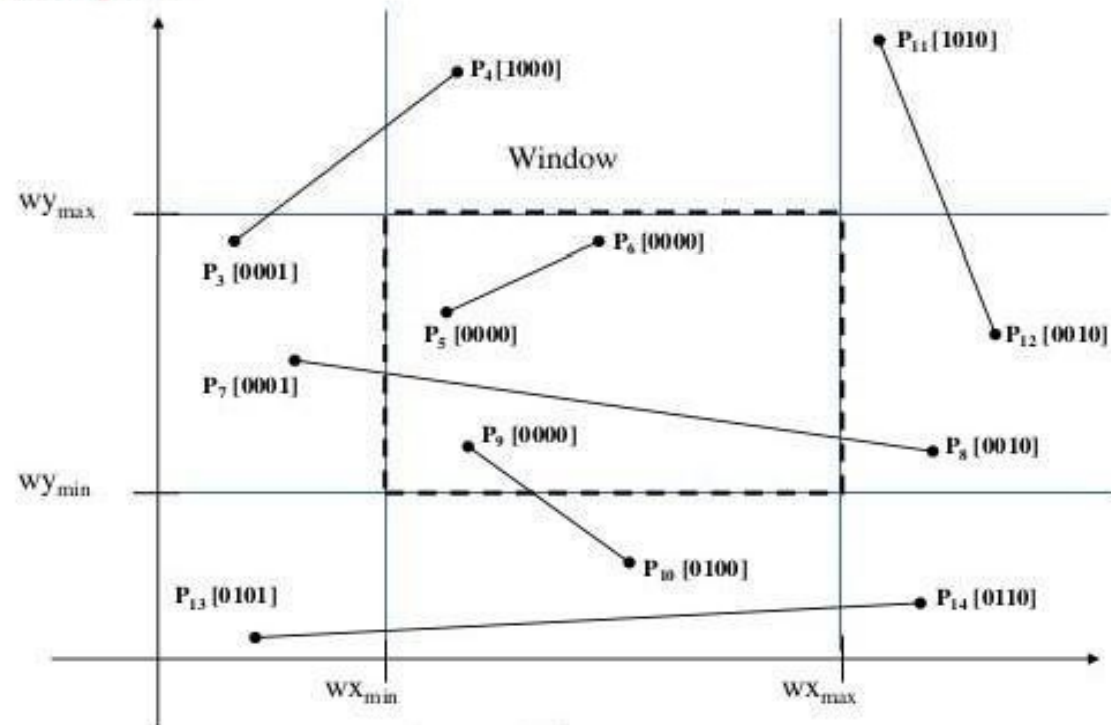
L= 1, if the region is left of window,  
= 0, otherwise.

1001	1000	1010
0001	0000 Window	0010
0101	0100	0110



## Contd...

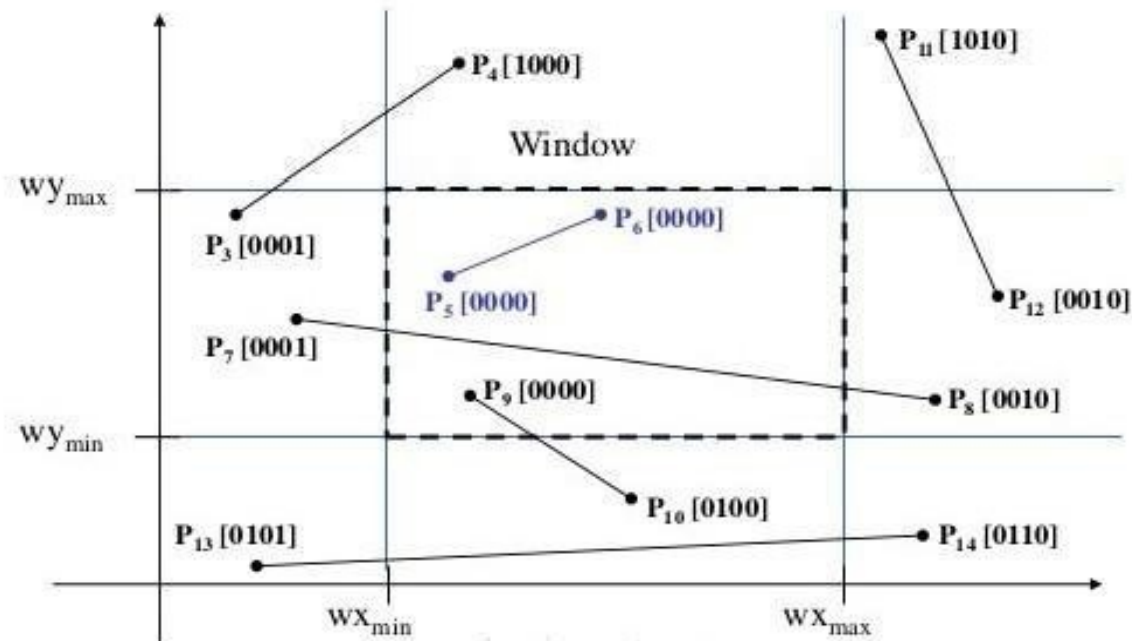
- Every end-point is labelled with the appropriate region code
- For example:





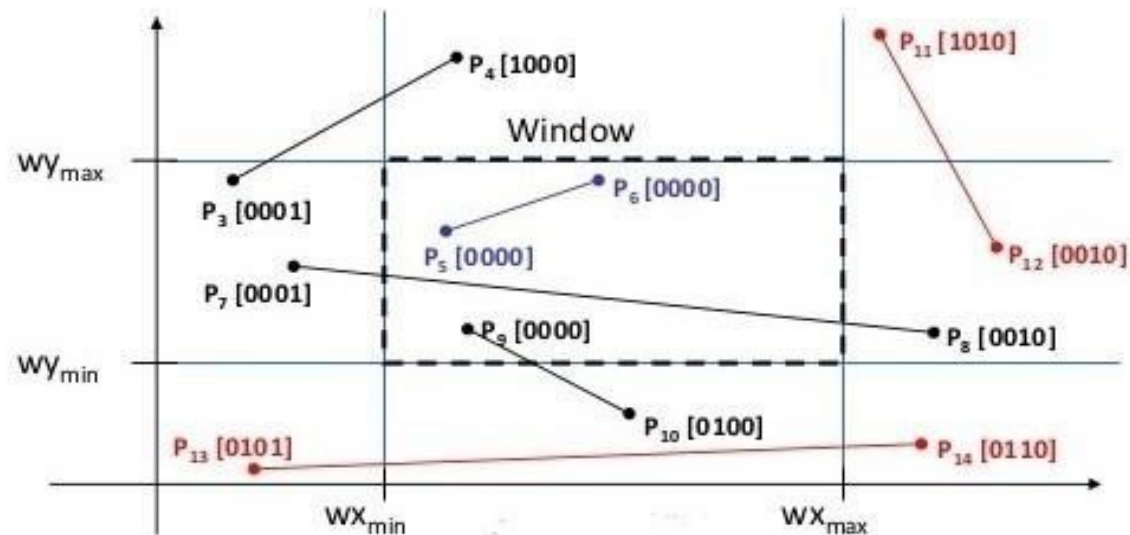
## Contd...

- Step2: Trivial acceptance of line segment
  - Lines completely contained within the window boundaries have region code [0000] for both end-points so are not clipped. i.e., accept these lines trivially.
  - E.g., line  $p_5p_6$



### Contd...

- Step3: Trivial rejection and clipping
  - 3.1 : Any lines that have a **1** in the same bit position in the region-codes for each endpoint are **completely outside** and we reject these lines.
  - The AND operation can efficiently check this: If the logical AND of both region codes result is **not 0000**, the line is **completely outside** the clipping region so clipped.



# AND operation

INPUT		OUTPUT
A	B	$Y = A.B$
0	0	0
0	1	0
1	0	0
1	1	1

## Contd...

3.2: If the logical AND operation results in 0000 then

- a) Choose an endpoint of the line that is outside the window.
- b) Find the intersection point at the window boundary by using the following formula:

Intersection with vertical boundary

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

Where

$$x = xw_{\min} \text{ or } xw_{\max}$$

Intersection with horizontal boundary

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

Where

$$y = yw_{\min} \text{ or } yw_{\max}$$

- c) Replace endpoint with the intersection point and update the region code. Above process is repeated until we find a clipped line either trivially accepted or trivially rejected.

## Algorithm

**Step 1** – Assign a region code for each endpoints.

**Step 2** – If both endpoints have a region code **0000** then accept this line.

**Step 3** – Else, perform the logical **AND** operation for both region codes.

**Step 3.1** – If the result is not **0000**, then reject the line.

**Step 3.2** – Else you need clipping.

**Step 3.2.1** – Choose an endpoint of the line that is outside the window.

**Step 3.2.2** – Find the intersection point at the window boundary using set of equations given below.(based on region code and window boundary ).

**Intersections with a vertical boundary:**

$$y = y_1 + m(x - x_1) \text{ (x is either xmin or xmax)}$$

**Intersections with a horizontal boundary:**

$$x = x_1 + (y - y_1)/m \text{ (y is either ymin or ymax)}$$

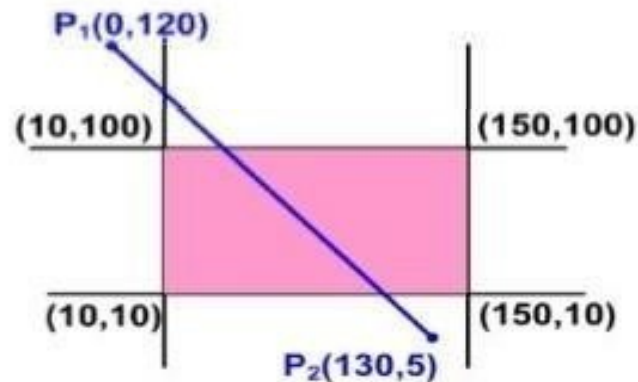
**Step 3.2.3** – Replace endpoint with the intersection point and update the region code.

**Step 4** – Repeat step 1 for other lines.

## Contd..

- **Example:** Use the Cohen-Sutherland line clipping algorithm to clip line defined with end points  $P_1(0,120)$  and  $P_2(130,5)$  against a window defined by the following four points :  $(10,10)$ ,  $(10,100)$ ,  $(150,10)$  and  $(150,100)$

**Obtain the endpoints of line  $P_1P_2$  after cohen-sutherland clipping**





## Contd...

- **Solution:**

1.  $P_1=1001$   $P_2=0100$

2. Both (0000)  $\rightarrow$  NO

3. And Operation

1001

0100

Result  $\rightarrow$  0000

3.1 not(0000)  $\rightarrow$  NO

3.2 0000  $\rightarrow$  yes

3.2.1 choose  $P_1$

3.2.2 Intersection with **LEFT** boundary

$$m = (5-120)/(130-0) = -0.8846$$

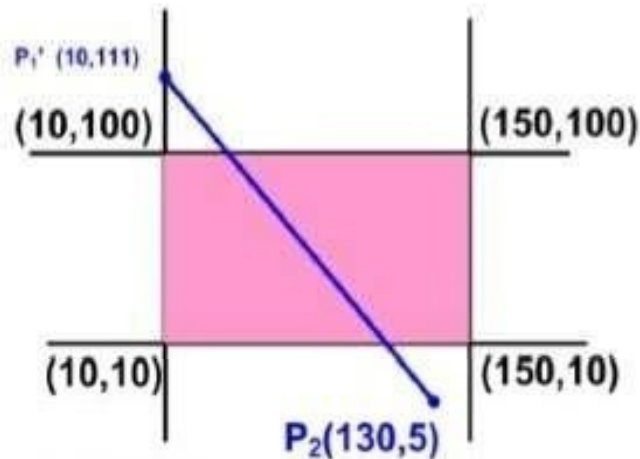
$$y = y_1 + m(x-x_1) \text{ where } x=10$$

$$y = 120 - 0.8846(10-0) = 111.15 \approx 111$$

$$P_1' = (10, 111)$$

3.2.3 Update region code  $P_1' = 1000$  (**TOP**)

3.2.4 Repeat step 2.



## Contd...

1.  $P_1' = 1000$   $P_2 = 0100$

2. Both (0000)  $\rightarrow$  NO

And Operation

1000

0100

Result  $\rightarrow$  0000

3.1 not(0000)  $\rightarrow$  NO

3.2 0000  $\rightarrow$  yes

3.2.1 choose  $P_1'$

3.2.2 Intersection with **TOP** boundary

$$m = (5 - 120) / (130 - 0) = -0.8846$$

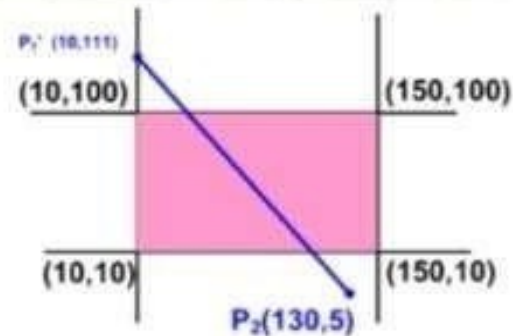
$$x = x_1 + (y - y_1) / m \quad \text{where } y = 100$$

$$x = 10 + (100 - 111) / (-0.8846) = 22.44 \approx 22$$

$$P_1'' = (22, 100)$$

3.2.3 Update region code  $P_1'' = 0000$

3.2.4 Repeat step 2.





## Contd...

1.  $P_1'' = 0000$   $P_2 = 0100$

2. Both (0000)  $\rightarrow$  NO

And Operation

0000

0100

Result  $\rightarrow$  0000

3.1 not(0000)  $\rightarrow$  NO

3.2 0000  $\rightarrow$  yes

3.2.1 choose  $P_2$

3.2.2 Intersection with **BOTTOM** boundary

$$m = (5 - 120) / (130 - 0) = -0.8846$$

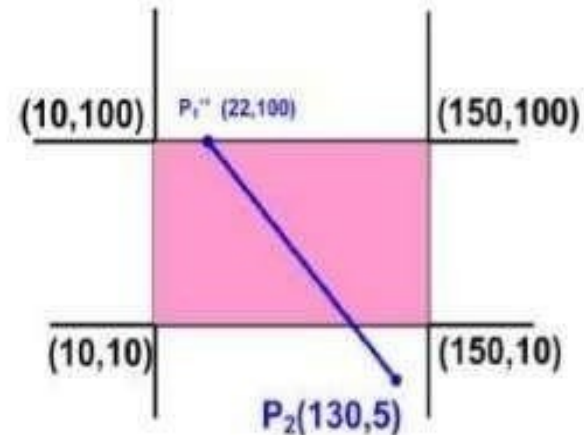
$$x = x_1 + (y - y_1) / m \quad \text{where } y = 10$$

$$x = 130 + (10 - 5) / (-0.8846) = 124.35 \approx 124$$

$$P_2' = (124, 10)$$

3.2.3 Update region code  $P_2' = 0000$

3.2.4 Repeat step 2.



## Contd...

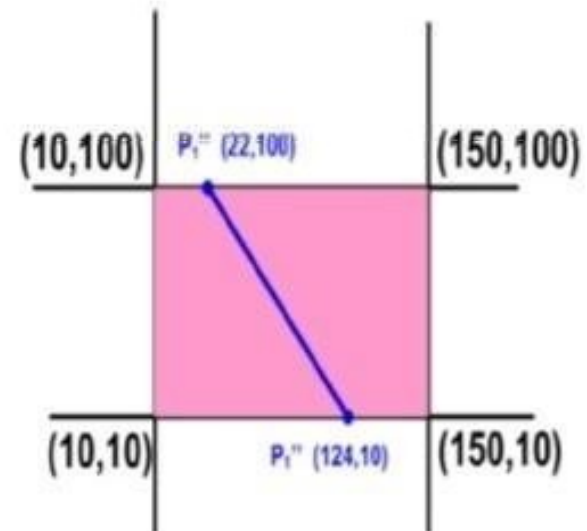
Both (0000) → YES

ACCEPT & DRAW

Thus endpoints after clipping

$P_1'' = (22, 100)$

$P_2' = (124, 10)$



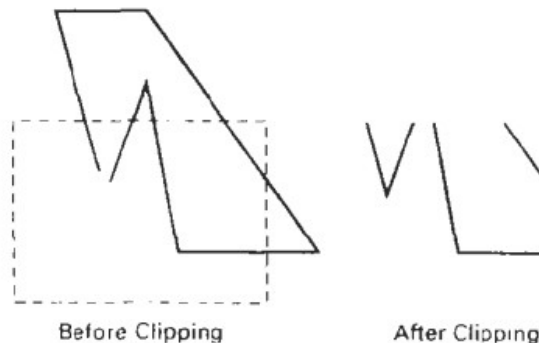
## POLYGON CLIPPING

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.

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.



## 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, 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
- At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper



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

- 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..
- Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

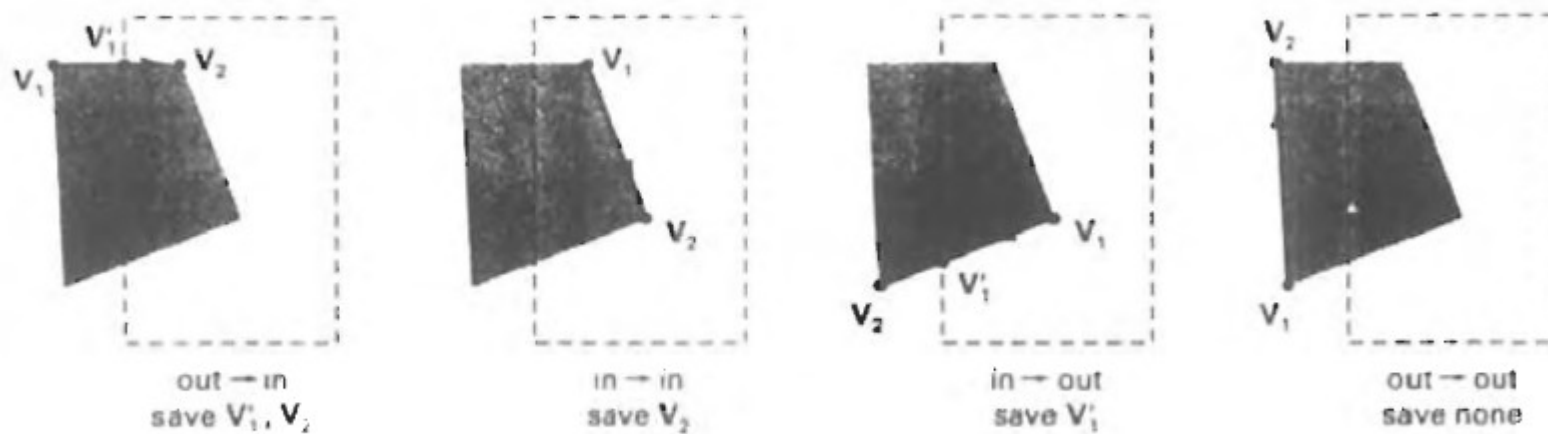
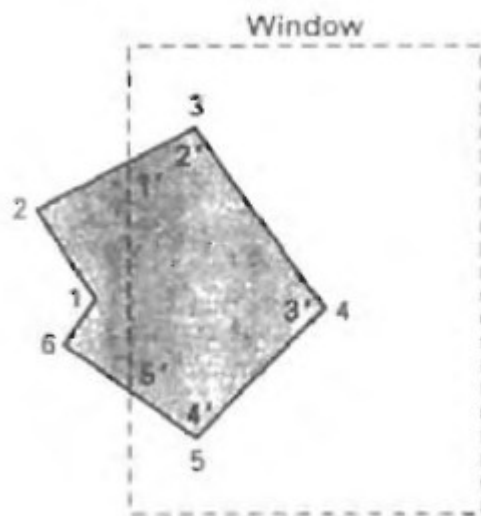


Figure 6-20



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.

# Three Dimensional Viewing Pipeline

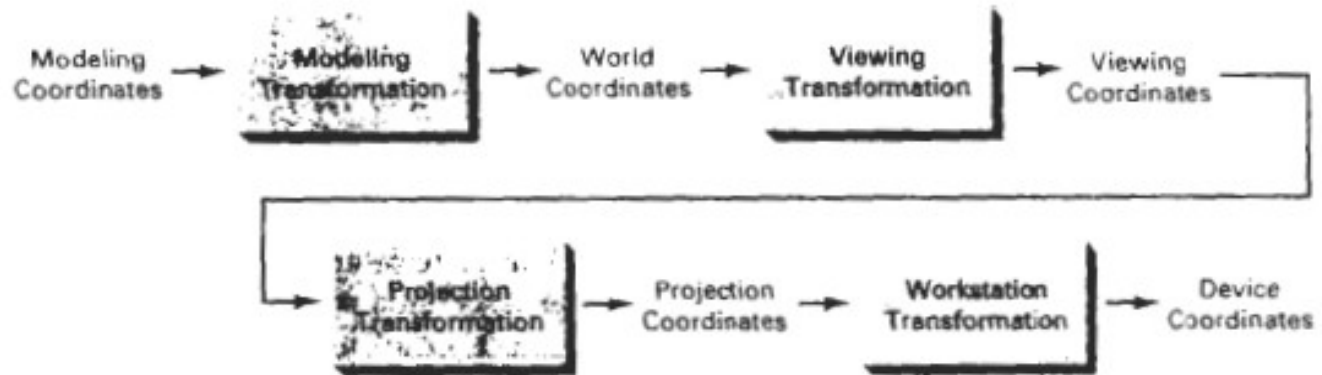


Figure 12-2

General three-dimensional transformation pipeline, from modeling coordinates to final device coordinates.

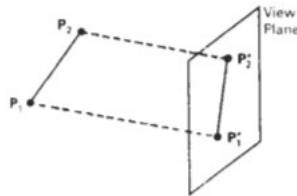


- Figure shows the general processing steps for modeling and converting a world-coordinate description of a scene to device coordinates.
- Once the scene has been modeled, world-coordinate positions are converted to viewing coordinates.
- The viewing-coordinate system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane, which we can think of in analogy with the camera film plane.
- Next, projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane, which will then be mapped to the output device.
- Objects outside the specified viewing limits are clipped from further consideration, and the remaining objects are processed through visible-surface identification and surface-rendering procedures to produce the display within the device viewport.

# PROJECTIONS

- Projection is mapping or transformation from 3D view into 2D view
- ie., world-coordinate descriptions of the objects in a scene are converted to viewing coordinates
- There are 2 types of projection
- **1 parallel projection**

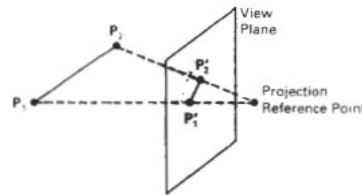
In a parallel projection, coordinate positions are transformed to the viewed plane along parallel lines, as shown in the ,example of Fig.



- A parallel projection preserves relative proportions of objects, and this is the method used in drafting to produce scale drawings of three-dimensional object
- Accurate views of the various sides of an object are obtained with a parallel projection, but this does not give us a realistic representation of the appearance of a three-dimensional object.

- **2 Perspective Projection**

- For a perspective projection the object positions are transformed to the view plane along lines that converge to a point called the projection reference point (or center of projection).
- The projected view of an object is determined calculating the intersection of the projection lines with the view plane.



- A perspective projection, on the other hand, produces realistic views but does not preserve relative proportions.

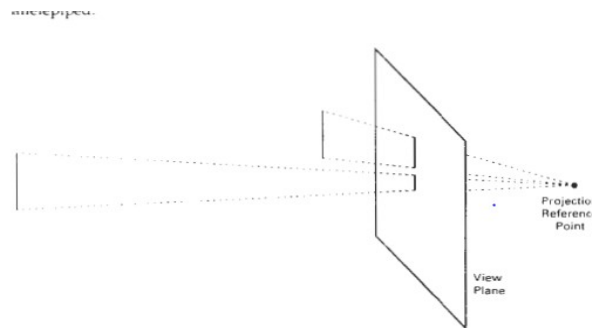


Figure 12-1b  
Perspective projection of equal-sized objects at different distances from the view plane.

## Parallel Projection

- In a parallel projection, coordinate positions are transformed to the viewed plane along parallel lines
- We can specify a parallel projection with a projection vector that defines the direction for the projection lines.
- When the projection is perpendicular to the view plane, we have an orthographic parallel projection.
- Otherwise, we have an oblique parallel projection
- . Figure illustrates the two types of parallel projections.



Figure 12-17  
Orientation of the projection vector  $V_p$  to produce an orthographic projection (a) and an oblique projection (b).

- **1. Orthographic projections**

- When the projection is perpendicular to the view plane, we have an orthographic parallel projection.
- Orthographic projections are most often used to produce the front, side, and top views of an object, as shown in figure Front, side, and rear orthographic projections of an object are called elevators;
- And a top orthographic projection is called a plain view
- Engineering and architectural drawings commonly employ these orthographic projections, because lengths and angles are accurately depicted and can be measured from the drawings.

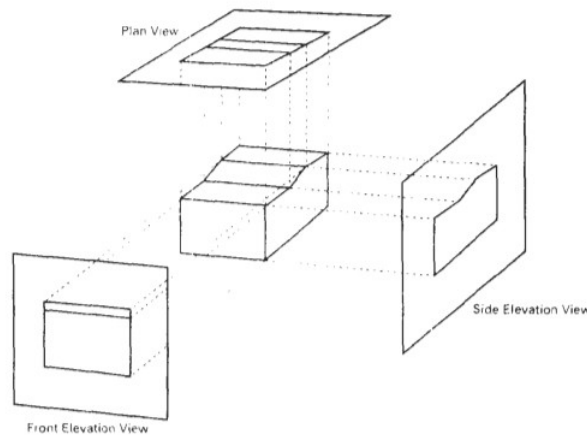
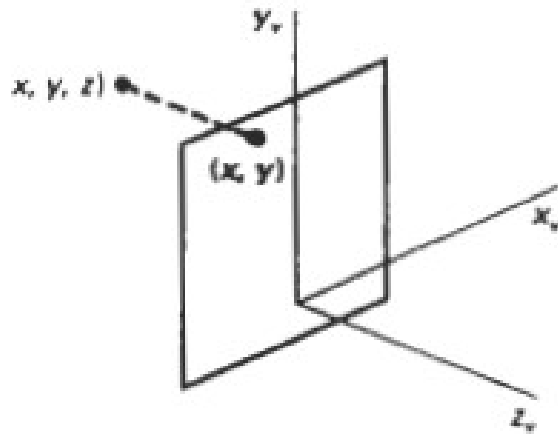


Figure 12-18  
Orthographic projections of an object, displaying plan and elevation views.

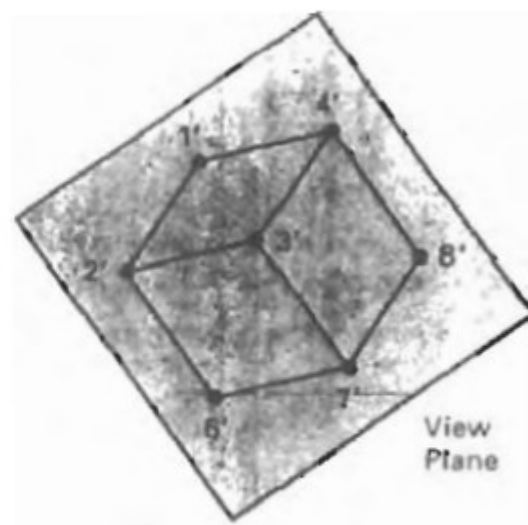
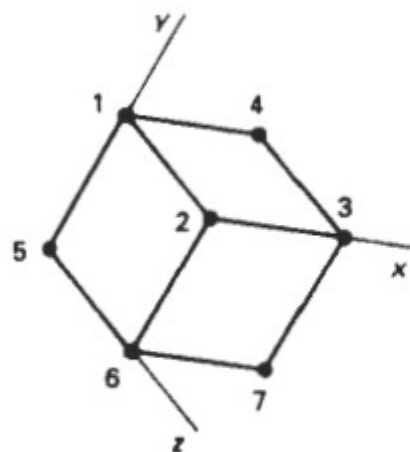
- Transformation equations for an orthographic parallel projection are straight forward.
- If the view plane is placed at position  $z$ , along the  $z$ , axis (Fig. 12-20), then any point  $(x, y, z)$  in viewing coordinates is transformed to projection coordinates as
$$x_p = x, \quad y_p = y$$
- where the original  $z$ -coordinate value is preserved for the depth information needed in depth cueing and visible-surface determination procedures.



*Figure 12-20*  
Orthographic projection of a point  
onto a viewing plane.

## Axonometric orthographic projections.

- In this Projection **it display more than one face of an object**. Such view are called axonometric orthographic projections.
- I.e. at least three faces are shown by manipulating the object (applying basic transformation on the object)
- axonometric orthographic projection are divided into 3 based on a foreshortening factor
  - 1 isometric
  - 2 Dimetric
  - 3 Trimetric
- foreshortening factor: It is the ratio of projected length to true length of a line
- **Trimetric** All the three foreshortening factor (corresponding to each principal axis) different
- **Dimetric** 2 Foreshortening factors are equal
- **Isometric** all the three foreshortening factor are equal



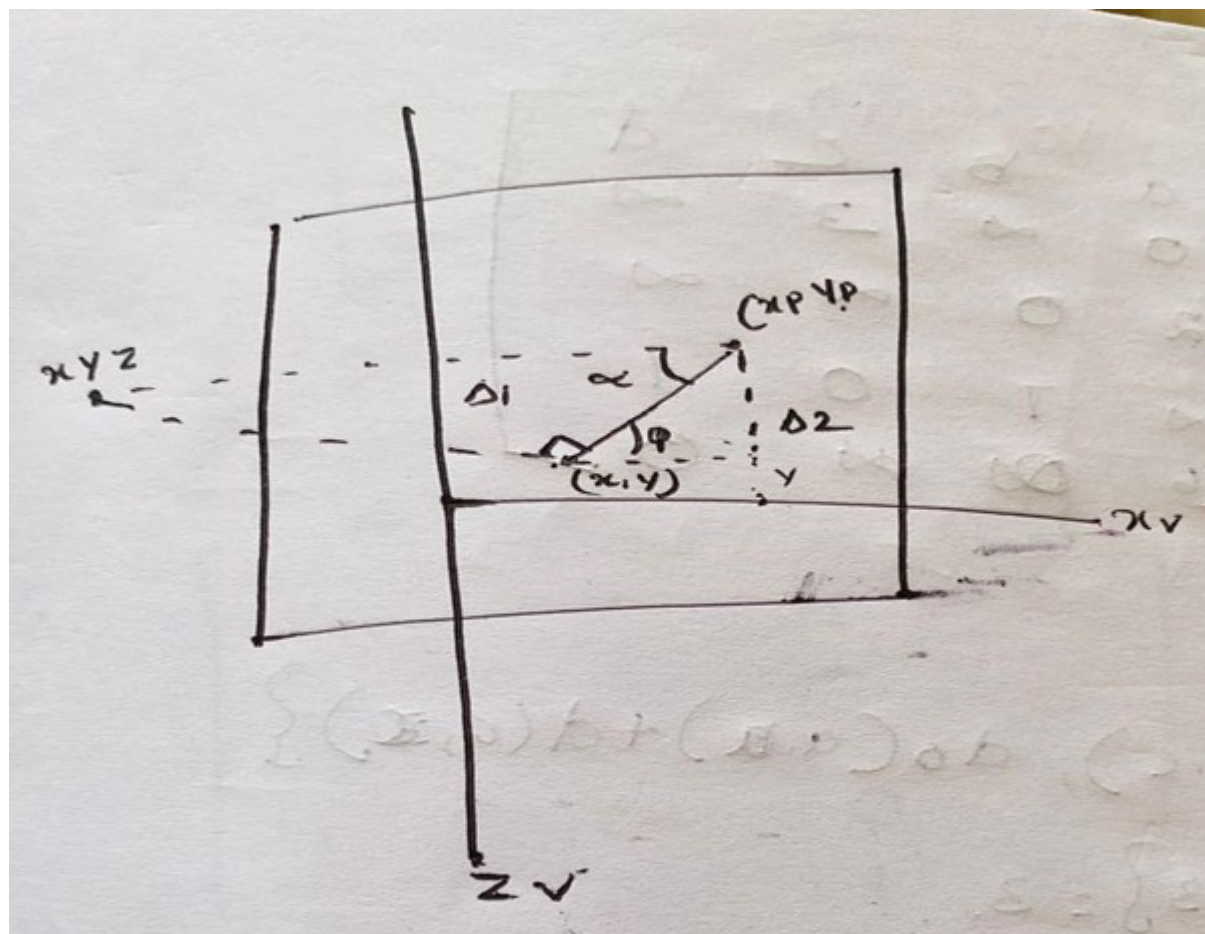
---

**Figure 12-19**  
Isometric projection for a cube.



# An oblique projection

- An oblique projection is obtained by projecting points along parallel lines that are not perpendicular to the projection plane.
- In some applications packages, an oblique projection vector is specified with two angles,  $\alpha$  and  $\phi$ ,
- Point  $(x, y, z)$  is projected to position  $(x_p, y_p)$  on the view plane.
- Orthographic projection coordinates on the plane are  $(x, y)$ .
- The oblique projection line from  $(x, y, z)$  to  $(x_p, y_p)$  makes an angle  $\alpha$  with the line on the projection plane that joins  $(x_p, y_p)$  and  $(x, y)$ .
- This line, of length  $L$  distance between  $(x_p, y_p)$  and  $(x, y)$ .  $L$  makes an angle  $\phi$  with the horizontal direction in the projection plane.



We can express the projection coordinates in terms of  $x$ ,  $y$ ,  $L$ , and  $\phi$  as

$$x_p = x + L \cos \phi$$

$$y_p = y + L \sin \phi$$

$$\tan \alpha = Z/L$$

$$L = \frac{z}{\tan \alpha}$$

$$= zL_1$$

where  $L_1$  is the inverse of  $\tan \alpha$ , which is also the value of  $L$  when  $z = 1$ . We can then write the oblique projection equations as

$$x_p = x + z(L_1 \cos \phi)$$

$$y_p = y + z(L_1 \sin \phi)$$

The transformation matrix for producing any parallel projection onto the  $x_v y_v$  plane can be written as

$$\mathbf{M}_{\text{parallel}} = \begin{bmatrix} 1 & 0 & L_1 \cos \phi & 0 \\ 0 & 1 & L_1 \sin \phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (12-10)$$

Oblique is divided into based on  $\alpha$

1 Cavalier and

2 Cabinet Projection

### **Cavalier Projection**

The value of  $\alpha = 45^\circ$  ie  $\tan \alpha = 1$  ( $\alpha$  is the angle between projection from xyz to xpyp) and line joining (xpyp) and (xy)

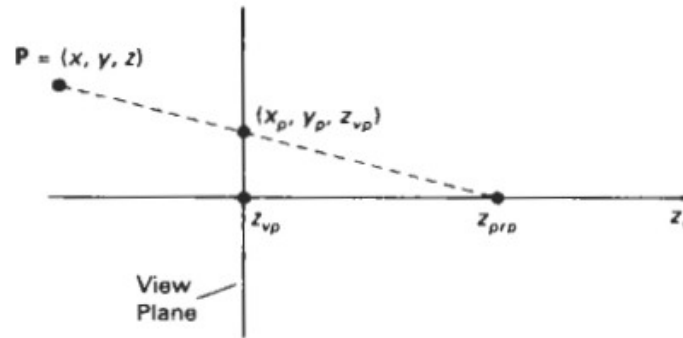
All lines perpendicular to the projection plane are projected with no change in length

### **Cabinet Projection**

When the projection angle  $\alpha$  is chosen so that  $\tan \alpha = 2$ , the resulting view is called a **cabinet** projection. For this angle ( $\approx 63.4^\circ$ ), lines perpendicular to the viewing surface are projected at one-half their length. Cabinet projections appear more realistic than cavalier projections because of this reduction in the length of perpendiculars

# Perspective Projections

To obtained by transforming points along projection lines that meet at the projection reference point.



Suppose we set the projection reference point at position  $z$ , along the  $z$ , axis, and we place the view plane at  $z_{vp}$  as shown in Fig.

We can write equations describing coordinate positions along this perspective projection line in parametric form as

$$x' = x - xu$$

$$y' = y - yu$$

$$z' = z - (z - z_{prp})u$$

Parameter  $u$  takes values from 0 to 1, and coordinate position  $(x', y', z')$  represents any point along the projection line. When  $u = 0$ , we are at position  $\mathbf{P} = (x, y, z)$ . At the other end of the line,  $u = 1$  and we have the projection reference point coordinates  $(0, 0, z_{prp})$ . On the view plane,  $z' = z_{vp}$  and we can solve the  $z'$  equation for parameter  $u$  at this position along the projection line:

$$u = \frac{z_{vp} - z}{z_{prp} - z} \quad (12-12)$$

Substituting this value of  $u$  into the equations for  $x'$  and  $y'$ , we obtain the perspective transformation equations

$$x_p = x \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = x \left( \frac{d_p}{z_{prp} - z} \right)$$

$$y_p = y \left( \frac{z_{prp} - z_{vp}}{z_{prp} - z} \right) = y \left( \frac{d_p}{z_{prp} - z} \right)$$

Where  $d_p = z_{prp} - z_{vp}$  is the distance of the view plane from the projection reference point.

Using a three-dimensional homogeneous-coordinate representation, we can write the perspective-projection transformation 12-13 in matrix form as

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -z_{vp}/d_p & z_{vp}(z_{prp}/d_p) \\ 0 & 0 & -1/d_p & z_{prp}/d_p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (12-14)$$

In this representation, the homogeneous factor is

$$h = \frac{z_{prp} - z}{d_p} \quad (12-15)$$

and the projection coordinates on the view plane are calculated from the homogeneous coordinates as

$$x_p = x_h/h, \quad y_p = y_h/h \quad (12-16)$$

In general, the projection reference point does not have to be along the  $i_z$  axis. We can select any coordinate position ( $x_{rp}$ ,  $y_{rp}$ ,  $z_{rp}$ ) on either side of the view plane for the projection reference point,

# Special cases

If the view plane is taken to be the xy plane, then  $z_{vp} = 0$  and the projection coordinates are

$$x_p = x \left( \frac{z_{prp}}{z_{prp} - z} \right) = x \left( \frac{1}{1 - z/z_{prp}} \right)$$
$$y_p = y \left( \frac{z_{prp}}{z_{prp} - z} \right) = y \left( \frac{1}{1 - z/z_{prp}} \right)$$

If the projection reference point is always taken to be at the viewing-coordinate origin. In this case,  $z_{prp} = 0$  and the projection coordinates on the viewing plane are

$$x_p = x \left( \frac{z_{vp}}{z} \right) = x \left( \frac{1}{z/z_{vp}} \right)$$
$$y_p = y \left( \frac{z_{vp}}{z} \right) = y \left( \frac{1}{z/z_{vp}} \right)$$



# Visible surface detection

- A major consideration in the generation of realistic graphics displays is identifying those parts of a scene that are visible from a chosen viewing position
- There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of application
- Some methods require more memory, some involve more processing time, and some apply only to special types of objects.
- Deciding upon a method for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated.
- Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images.

These two approaches are called **object-space methods** and **image-space methods**,

### **object-space methods**

An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible.

eg back face detection and removal

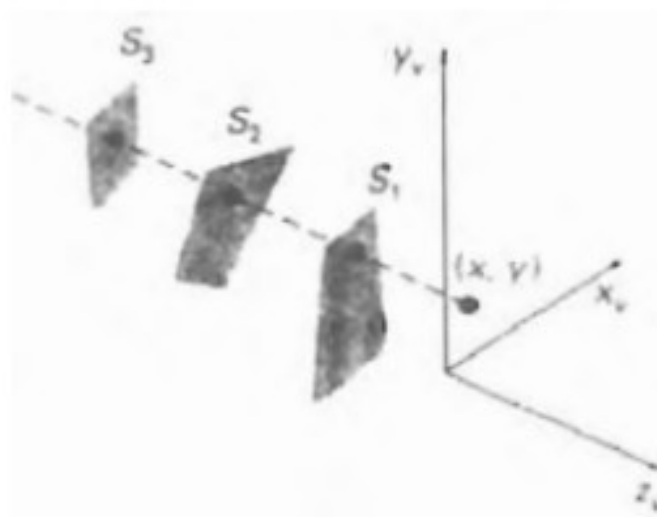
### **image-space methods**

In an image-space algorithm, visibility is decided point by point at each pixel position on the projection plane.

Eg depth buffer Scan line

# DEPTH-BUFFER (Z- buffer Method)

- A commonly used image-space approach to detecting visible surfaces is the depth-buffer method, which compares surface depths at each pixel position on the projection plane.
- This procedure is also referred to as the Z-buffer method, since object depth is usually measured from the view plane along the z axis of a viewing system.
- Each surface of a scene is processed separately, one point at a time across the surface.
- The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement



*Figure 13-4*

At view-plane position  $(x, y)$ , surface  $S_1$  has the smallest depth from the view plane and so is visible at that position.

- With object descriptions converted to projection coordinates, each  $(x, y, z)$  position on a polygon surface corresponds to the orthographic projection point  $(x, y)$  on the view plane.
- Therefore, for each pixel position  $(x, y)$  on the view plane, object depths can be compared by comparing  $z$  values.
- Figure 13-4 shows three surfaces at varying distances along the orthographic projection line from position  $(x, y)$  in a view plane taken as the  $xy$ , plane.
- Surface  $s_1$ , is closest at this position, so its surface intensity value at  $(x, y)$  is saved

We can implement the depth-buffer algorithm in normalized coordinates, so that  $z$  values range from 0 at the back clipping plane to  $z_{\max}$  at the front clipping plane. The value of  $z$ , can be set either to 1 (for a unit cube) or to the largest value that can be stored on the system.

- Two buffer areas are required.
- A **depth buffer** is used to store depth values for each (x, y) position as surfaces are processed,
- and the **refresh buffer** stores the intensity values for each position.
- Initially, all positions in the depth buffer are set to 0 (minimum depth), and the refresh buffer is initialized to the background intensity.
- Each surface listed in the polygon tables is then processed, one scan line at a time, calculating the depth (z value) at each (x, y) pixel position.
- The calculated depth is compared to the value previously stored in the depth buffer at that position.
- If the calculated depth is greater than the value stored in the depth buffer, the new depth value is stored, and the surface intensity at that position is determined and in the same xy location in the refresh buffer

We summarize the steps of a depth-buffer algorithm as follows:

1. Initialize the depth buffer and refresh buffer so that for all buffer positions  $(x, y)$ ,

$$\text{depth}(x, y) = 0, \quad \text{refresh}(x, y) = I_{\text{backgnd}}$$

2. For each position on each polygon surface, compare depth values to previously stored values in the depth buffer to determine visibility.

- Calculate the depth  $z$  for each  $(x, y)$  position on the polygon.
- If  $z > \text{depth}(x, y)$ , then set

$$\text{depth}(x, y) = z, \quad \text{refresh}(x, y) = I_{\text{surf}}(x, y)$$

where  $I_{\text{backgnd}}$  is the value for the background intensity, and  $I_{\text{surf}}(x, y)$  is the projected intensity value for the surface at pixel position  $(x, y)$ . After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the refresh buffer contains the corresponding intensity values for those surfaces.

Depth values for a surface position  $(x, y)$  are calculated from the plane equation for each surface:

$$z = \frac{-Ax - By - D}{C} \quad (13-4)$$

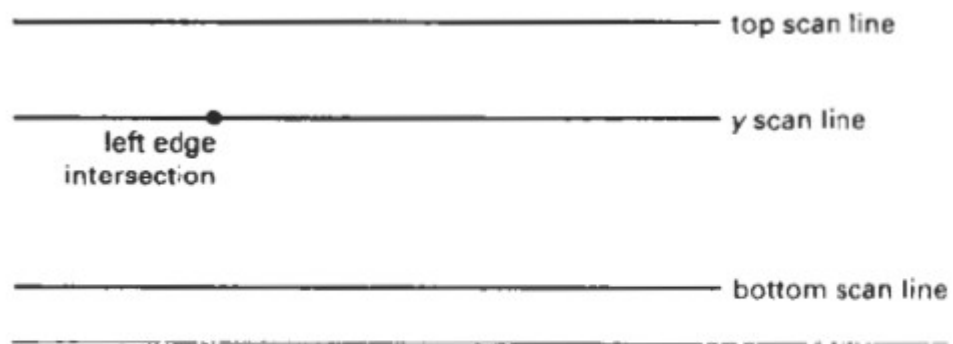
. If the depth of position  $(x, y)$  has been determined to be  $z$ , then the depth  $z'$  of the next position  $(x + 1, y)$  along the scan line is obtained from above equation as

$$z' = \frac{-A(x + 1) - By - D}{C}$$

$$z' = z - \frac{A}{C}$$

- The ratio  $-A/C$  is constant for each surface, so succeeding depth values across a scan line are obtained from preceding values with a single addition.
- On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line (Fig. 13-6). Depth values at each successive position across the scan line are then calculated by Eq. 13-6

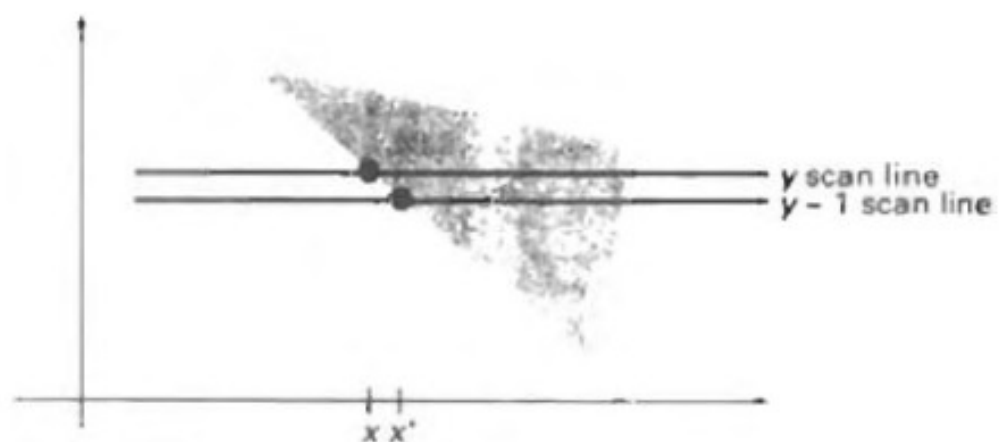




*Figure 13-6*  
Scan lines intersecting a polygon surface.

- We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line, as shown in Fig. 13-6.
- Starting at a top vertex, we can recursively calculate x positions down a left edge of the polygon as  $x' = x - l/m$ , where m is the slope of the edge (Fig. 13-7).
- Depth values down the edge are then obtained recursively as

$$z' = z + \frac{A/m + B}{C}$$



*Figure 13-7*

Intersection positions on successive scan lines along a left polygon edge.

## Homework

- **Example1:** Let ABCD be the rectangular window with A(20, 20), B(90, 120), C(90, 70), and D(20,70). Find the region codes for end points and use Cohen- Sutherland algorithm to clip the line p q with p( 10,30) and q(80, 90).
- **Example2:** Use the Cohen-Sutherland algorithm to clip line defined with end points p1(40,15) and p2(75,45) against a window A(50,10), B(80,10), C(80,40) and D(50,40).
- **Example3:** Use the Cohen-Sutherland algorithm to clip line defined with end points p1(70, 20) and p2(100,10) against a window A(50,10), B(80,10), C(80,40) and D(50,40).

# University Questions

- ⇒ Given a clipping window A(-20,-20), B(40,-20), C(40,30) and D(-20,30). Using Cohen Sutherland line clipping algorithm, find the visible portion of the line segment joining the points P(-30,20) and Q(60,-10). (6)
- ⇒ Explain the Cohen Sutherland line clipping algorithm with suitable examples. (6)
- ⇒ How does Cohen Sutherland algorithm determine whether a line is visible, invisible or a candidate for clipping based on the region codes assigned to the end points of the line? (4)