# Two Dimensional Viewing

# Different Coordinate Systems

- **Modeling Coordinate System**
  - Used to define coordinates that are used to construct the shape of individual parts (objects) of 2D scene.
- **World Coordinate System**
  - Used to organize the individual objects (points, lines, circles etc.) into a scene.
  - A scene is made up of a collection of objects.
  - These objects make up the "scene" or "world" that we want to view, and the coordinates that we use to define the scene are world coordinates.

- **Viewing Coordinate System**
  - Used to define particular view of a 2D scene. Translation, scaling, and rotation of the window will generate a different view of the scene.
  - For a 2D picture, a view is selected by specifying a subarea of the total picture area.
- **Normalized Viewing Coordinates**
  - Viewing coordinates between 0 and 1.
  - Used to make the viewing process independent of the output device. (monitor, mobile, etc.)
- **Device Coordinate or Screen Coordinate System**
  - Used to define coordinates in an output device.
  - Device coordinates are integers within the range (0,0) to $(x_{max}, y_{max})$ for a particular output device.

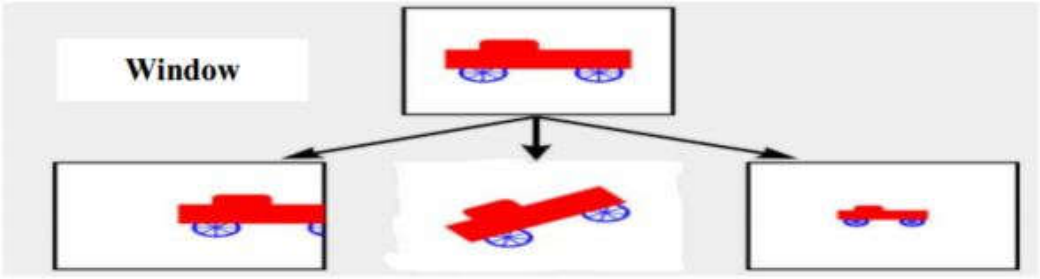



**Fig: several view of the same car**

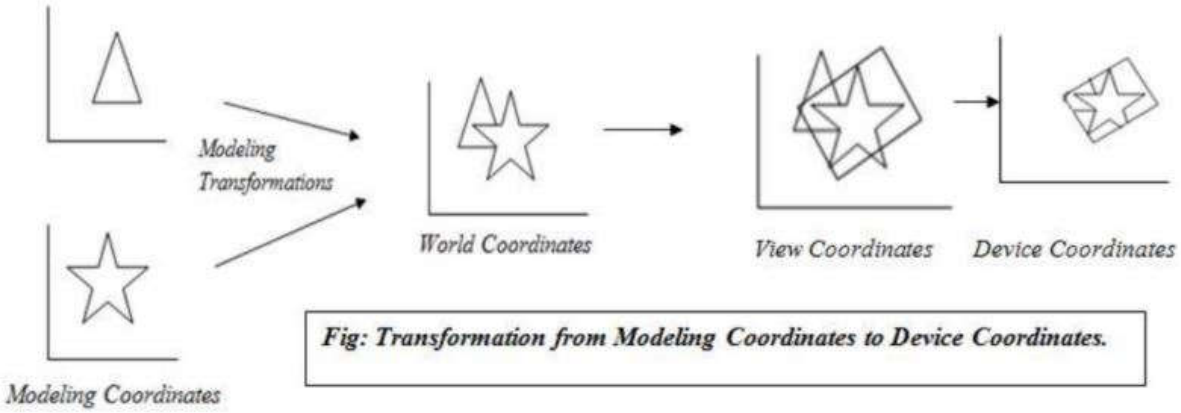
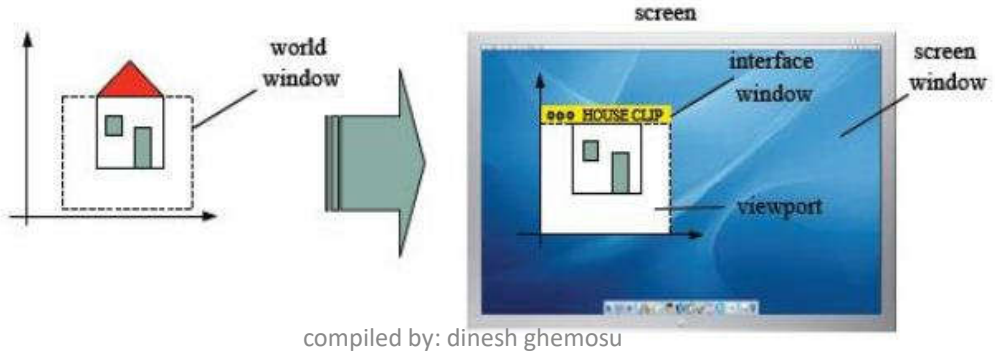
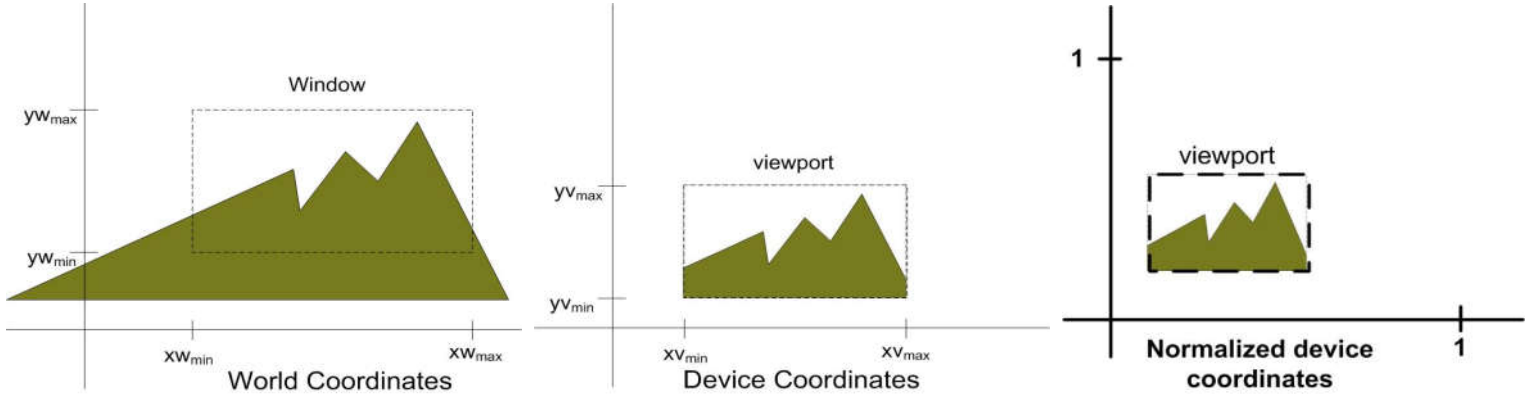


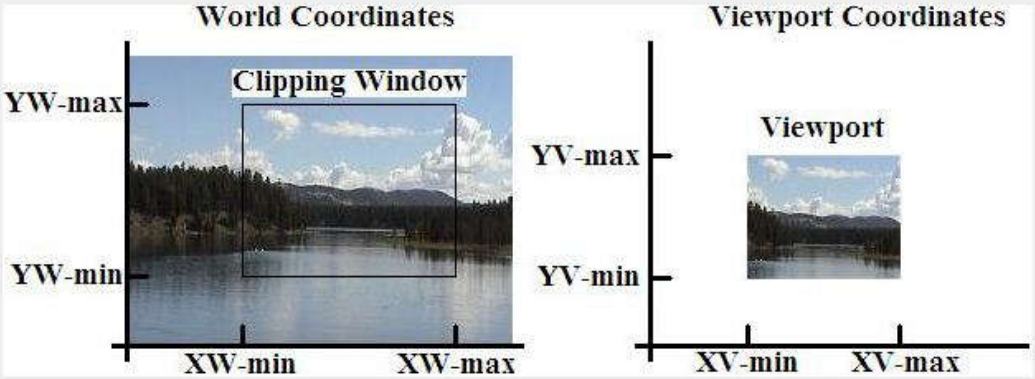

***Fig: Transformation from Modeling Coordinates to Device Coordinates.***

# Viewing Pipeline

- It is a procedures for displaying views of a two-dimensional picture on an output device:
  - Specify which parts of the object to display (clipping window, or world window, or viewing window)
  - Where on the screen to display these parts (viewport).
- **Window** : a world coordinate area selected for display.
- **Viewport** : an area on display to which a window is mapped.
- The window defines *what* is to be viewed; the viewport defines *where* it is to be displayed.
- Windows and viewport are rectangle in standard position, with its edges parallel the coordinates axes.




Window
$yw_{max}$
$yw_{min}$
$xw_{min}$
$xw_{max}$
World Coordinates
viewport
$yv_{max}$
$yv_{min}$
$xv_{min}$
$xv_{max}$
Device Coordinates
1
viewport
Normalized device coordinates
1


World Coordinates
Clipping Window
YW-max
YW-min
XW-min
XW-max
Viewport Coordinates
Viewport
YV-max
YV-min
XV-min
XV-max

- Window and viewport are often rectangular in standard positions, because it simplifies the transformation process and clipping process.
- Other shapes such as polygons, circles take longer time to process

# Applications

- *By changing the positions of the viewports, we can view objects at different positions on the display area of an output device.*
- *Also by varying the size of viewport, we can change size of displayed objects.*
- *Zooming effects* can be obtained by successively mapping different-sized windows on a fixed –sized viewport.
- *Panning effect* are produced by moving a fixed-sized window across the various objects in a scene.

# Types of coordinate system

**World coordinate system(WCS)**

- A right-handed Cartesian coordinate system in which we describe the coordinates of the picture to be displayed.

**Physical Device coordinate system(PDCS)**

- The coordinate system corresponding to device in which we want to display the object.
- Different display device has their own coordinate system.

**Normalized Device Coordinate System (NDCS)**

- The coordinate system in which the display area of the device corresponds to a unit square.

# Viewing Transformation

- Process of mapping 2D world coordinate scene to the device coordinates.
- In particular, objects inside the world or clipping window are mapped to the viewport.
- Also referred to as *window-to-viewport transformation* or the *windowing transformation*

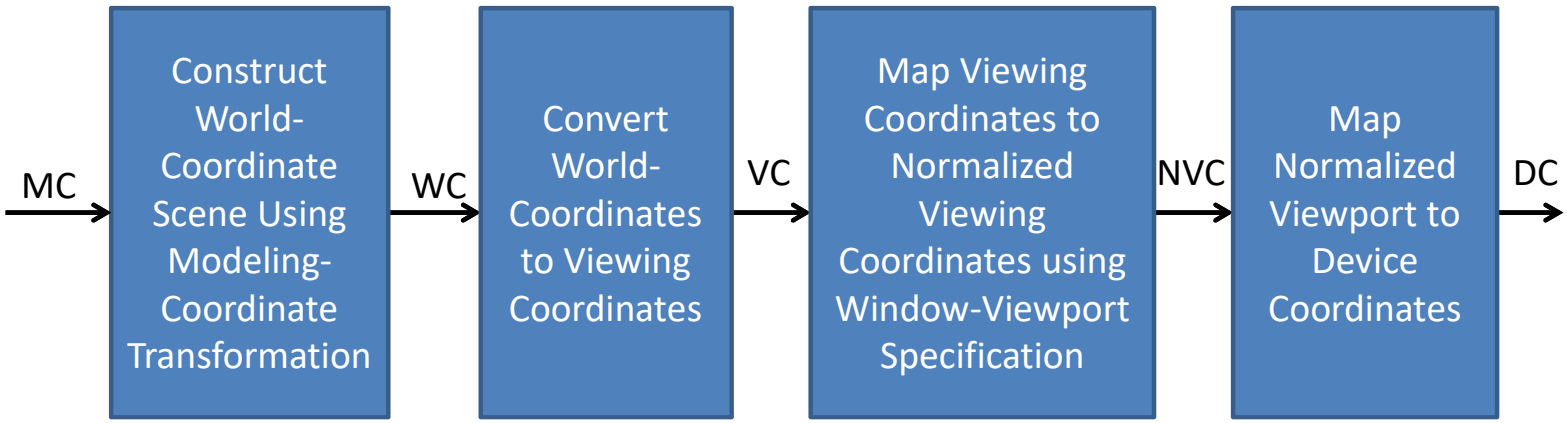



Figure: The Two Dimensional Viewing Transformation Pipeline

# Window to Viewport Coordinate Transformation

- In general a window to viewport transformation involves scaling and translation.

Steps:

1. Translate window to the origin

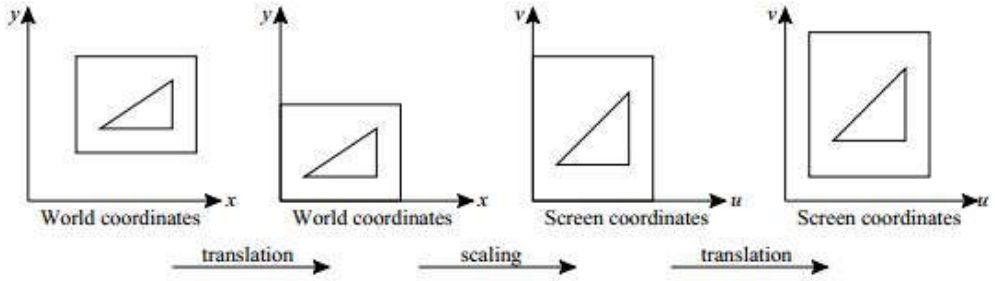
$$(T_x = -X_{wmin},\ T_y = -\ Y_{wmin})$$

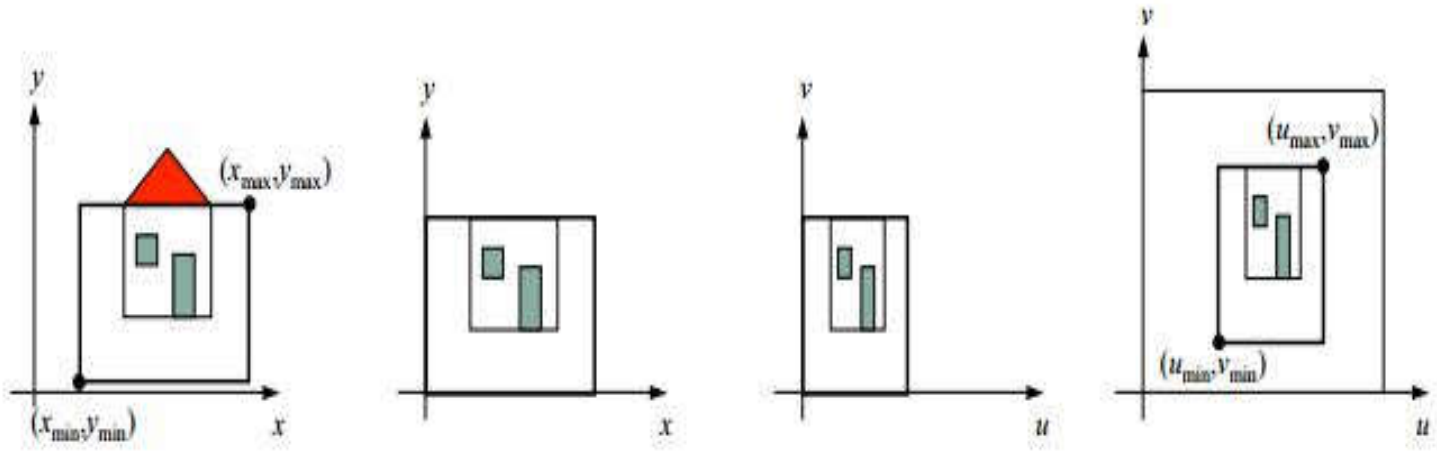
1. Scale the window to size of viewport .

$$(S_x = \frac{X_{vmax} - Xvmin}{X_{wmax} - Xwmin},\ S_y = \frac{X_{vmax} - Xvmin}{X_{wmax} - Xwmin}$$

2. Translate the scaled window to screen position.

$$(T_x = X_{vmin},\ T_y = Y_{vmin})$$

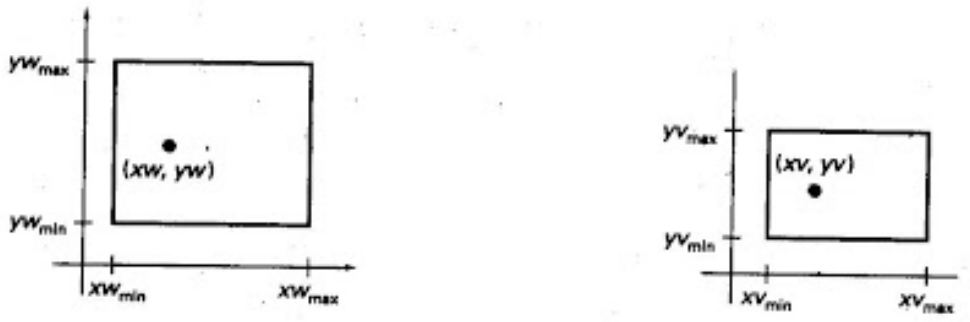



y
$(x_{max}, y_{max})$
$(x_{min}, y_{min})$
x
y
x
v
u
v
$(u_{max}, v_{max})$
$(u_{min}, v_{min})$
u

# Question

- Find the normalization transformation for window to viewport which uses the rectangle whose lower left corner at (2, 2) and upper right corner at (6,10) as a window and the viewport that has lower left corner at (0,0) and upper right corner at (1, 1).

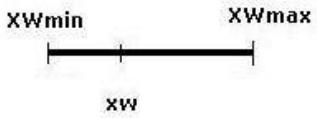
- This mapping or transformation involves developing formulas that start with a point in the world window, say (xw, yw). The formula is used to produce a corresponding point in viewport coordinates, say (xv, yv).

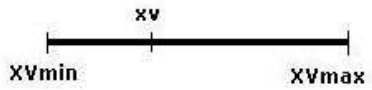


$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

For proportionality in x:



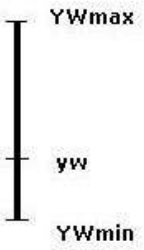



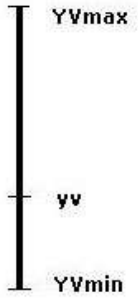

Solving these we get

$$xv = xv_{min} + (xw - xw_{min})sx$$

$$yv = yv_{min} + (yw - yw_{min})sy$$

For proportionality in y:




Where scaling factors are

$$sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

If sx = sy, the proportion is maintained otherwise the scene is stretched

- The position of the viewport can be changed allowing objects to be viewed at different positions on the Interface Window.
- Multiple viewports can also be used to display different sections of a scene at different screen positions.
- Also, by changing the dimensions of the viewport, the size and proportions of the objects being displayed can be manipulated.
- Thus, a zooming affect can be achieved by successively mapping different dimensioned clipping windows on a fixed sized viewport.

# Clipping

- Process of cutting off the line which are outside the window→ **clipping**
- Procedure that identifies those portion of a picture that are either inside or outside of a window → **clipping algorithm**
- Region against which an object is clipped→ **clip window**.

**Types**:

- Point clipping
- Line clipping
- Polygon clipping
- 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:

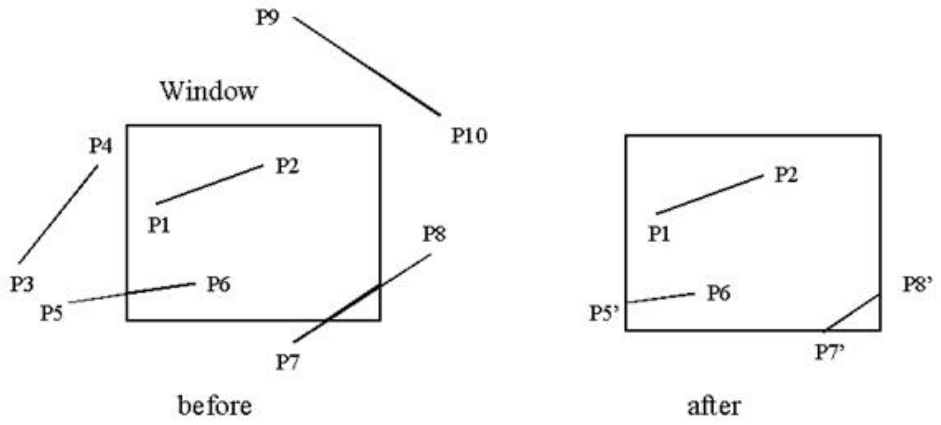
$$xw_{min} \leq x \leq xw_{max}$$

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

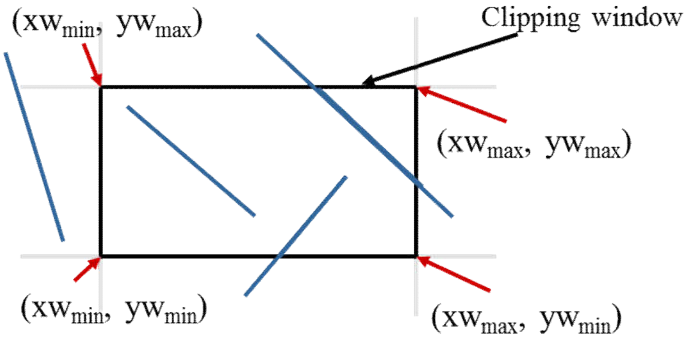
where the edges of the clip window ($xw_{min}$, $xw_{max}$, $yw_{min}$, $yw_{max}$) can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

# Line Clipping

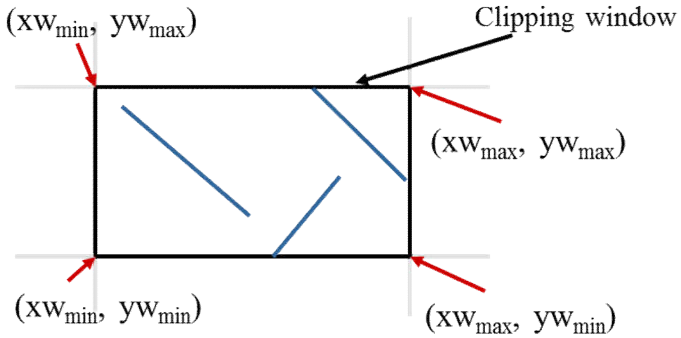
- A line clipping procedures involves several parts.
- We have to test to determine whether the line
  - Completely inside the clipping window, or
  - Completely outside the clipping window, or
  - Partially inside the clipping window.



- A line with both ends points inside the clipping boundaries are saved. (line $P_1P_2$ in above figure)
- A line with both ends points outside the clipping boundaries are not saved. (lines $P_3P_4$ and $P_9P_{10}$)
- A line which cross one or more clipping boundaries(lines $P_5P_6$ and $P_7P_8$ ) require intersection point calculation and the section of line is then clipped.




Before clipping



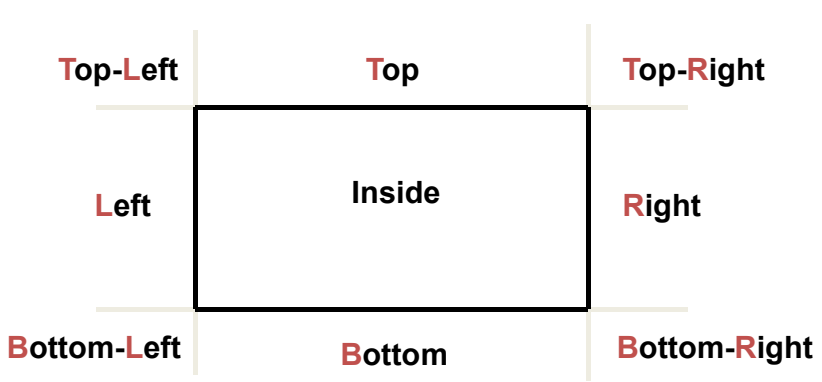

After clipping

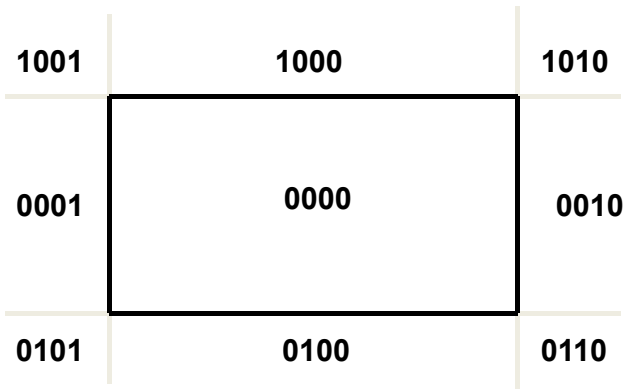
# Line Clipping Algorithm

- Cohen-Sutherland Algorithm
- Liang- Barsky method
- Nicholl – Lee – Nicholl method

# Cohen-Sutherland Algorithm

- The oldest and popular line clipping algorithm.
- In this algorithm, the 2D space is divided into 9 parts and assign a 4 bit code to each region as shown in figure.
- Every line end point in a picture is assigned a four-bit binary code, called **region code or outcodes.** It identifies the point relative to the boundaries of the clipping rectangle region




| Region codes | T | B | R | L |
|---|---|---|---|---|
| Bit position | 4 | 3 | 2 | 1 |

- For any endpoint (*x, y*) of a line, the code's bit are set according to the following conditions:

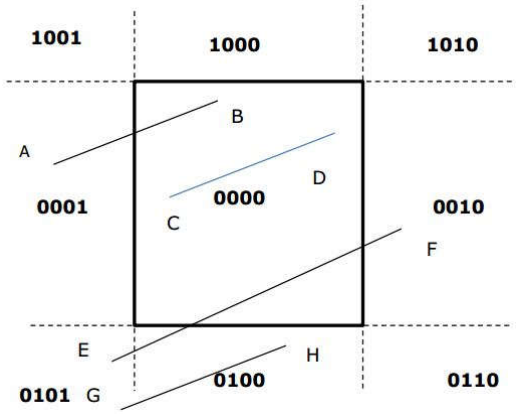
  1st bit set to 1: Point lies to **left** of window $x < xw_{min}$

  2nd bit set to 1: Point lies to **right** of window $x > xw_{max}$

  3rd bit set to 1: Point lies to **bottom** of window $y < yw_{min}$

  4th bit set to 1: Point lies to **top** of window $y > yw_{max}$

- For Example




| Line | Point | Region code |
|---|---|---|
| AB | A | |
| | B | |
| CD | C | |
| | D | |
| EF | E | |
| | F | |
| GH | G | |
| | H | |

# Algorithm

- The line segment's endpoints are tested to see if the line can be trivially accepted or trivially rejected. If the line cannot be trivially accepted or rejected, an intersection of the line with window edge is determined and the trivial reject/accept test if repeated. This process is continued until the line is accepted.

**Algorithm Steps:**

1. Assign a region code for each endpoints.
2. If both endpoints have a region code 0000 ---→ trivially accept these line.
3. Else, perform the logical AND operation for both region codes.
   - 3.1 **if** the result is **NOT** 0000 → trivially reject the line.
   - 3.2 **else** (i.e. result = 0000, need clipping)
     - 3.2.1. Choose an endpoint of the line that is outside the window.
     - 3.2.2. Find the intersection point at the window boundary (base on region code).
     - 3.2.3. Replace endpoint with the intersection point and update the region code.
     - 3.2.4. Repeat step 2 until we find a clipped line either trivially accepted or trivially rejected.
4. Repeat step 1 for other lines.

**Intersection calculations:**

Intersection with vertical boundary

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

Where

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

Intersection with horizontal boundary

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

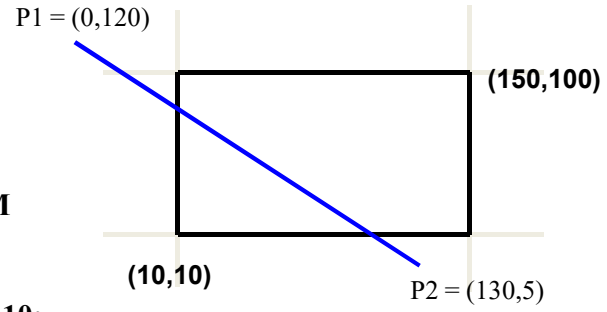
Where

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

# Example

**1. P1=1001, P2=0100**
**2. (both 0000) – No**
**3. AND Operation**
**P1 → 1001**
**P2 → 0100**
**Result 0000**
**3.1 (not 0000) – no**

**3.2 (0000) yes**
**3.2.1 choose P2**
**3.2.2 intersection with BOTTOM boundary**
**m = (5-120)/(130-0) = -0.8846**
**x = x1 + (y –y1)/m where y = 10;**
**x = 130 + (10-5)/ -0.8846 = 124.35 = 124**
**P2' = (124, 10)**
**3.2.3 update region code P2' = 0000**
**3.2. 4 repeat step 2**



# Numerical Examples

- Use the Cohen Sutherland algorithm to clip $P_1(70,20)$ and $P_2(100,10)$ against a window lower left hand corner (50,10) and upper right hand corner (80,40).
- Given a clipping window A(20,20) B(60,20) C(60,40) and D(20,40). Using Cohen Sutherland Algorithm find the visible portion of line segment joining the points P(40,80) and Q(120,30).

# Mid-point Subdivision Algorithm

- It is an extension of Cohen-Sutherland algorithm. It used a recursive process in which we used binary searching.
- This is so because we compute the region codes of each of line segment that has been divided at its mid point.
- The mid point coordinates $P_m(x_m, y_m)$ of a line segment joining $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ are given as:

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

# Mid-point Subdivision Algorithm

For each of the sub-line segment the outcodes (region codes) are determined and again logical AND's are taken:

a. If both are the endpoints of a line has outcodes a 0000, then the line is totally visible.

b. If both the outcodes are not zero then we have to take logical AND of both outcodes and then decide whether that lien is visible or not.

   i. If the logical AND result it non-zero, it means the line is totally invisible.

   ii. If the logical AND result is zero, it means a line may be partially visible.

   - It algorithm is:

     a. If the endpoint is visible then it is the farthest visible point. The process is complete. If not then continue.

     b. If the line is trivially invisible, no output is generated. The process is complete. If not then continue.

     c. Divide the line $\boldsymbol{P_1P_2}$ at its mid-point, $\boldsymbol{P_m}$. Apply the test given above to the two segments $\boldsymbol{P_1P_m}$ and $\boldsymbol{P_mP_2}$. if $\boldsymbol{P_mP_2}$ is rejected as trivially invisible then the mid-point is a **overestimation** of the farthest visible point. Continue with $\boldsymbol{P_1P_m}$, otherwise the mid-point is an **underestimation** of the farthest visible point. Continue with $\boldsymbol{P_2P_m}$. Continue this process till the segment becomes so short that the mid point corresponds to the accuracy of the machine.

# Liang-Barsky Line Clipping

- Developed by Ling Y. and Barsky B. in 1984 for 2D line clipping.
- Can be extended for 3D-clipping.
- They established a geometrical relationship between the window and the parametric line.
- A parametric equation of a line segment is in the form:

  $x = x_1 + u\Delta x$

  $y = y_1 + u\Delta y$

  *where,*

  $\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1$ *and,*

  $\boldsymbol{u = 0 \leq u \leq 1}$

# Liang-Barsky Line Clipping

- For a point (x, y) to be inside the clipping window, we have,

$$xW_{min} \leq x1 + \Delta x.u \leq xWmax$$

$$yW_{min} \leq y1 + \Delta y.u \leq yWmax$$

Rewriting these four inequalities as:

$$p_k u \leq q_k \; where \; k = 1,2,3,4$$

Here parameters p and q are defined as follow:

$p_1 = -\Delta x$ , $q_1 = x_1 - xW_{min}$ (for Left (1st )boundary)

$p_2 = \Delta x$ , $q_2 = xW_{max} - x_1$ (for Right (2nd )boundary)

$p_3 = -\Delta y$ , $q_3 = y_1 - yW_{min}$ (for Bottom (3rd )boundary)

$p_4 = \Delta y$ , $q_4 = yW_{max} - y_1$ (for Top (4th )boundary)

# Liang-Barsky Line Clipping

- Observation of the following facts:

1. If $p_k = 0$, the line is parallel to the corresponding boundary and,
   a. If $q_k < 0$, the line is completely outside the boundary and can be eliminated.
   b. If $q_k > 0$, the line is inside the boundary.
2. If $p_k < 0$, the extended line proceeds from the outside to the inside of the corresponding boundary line.
3. If $p_k > 0$, the extended line proceeds from the inside to the outside of the corresponding boundary line.
4. If $pk \neq 0$, we can calculate the value of $u$ that corresponds to the point where the infinitely extended line intersects the extensions of boundary k as:

$$u = \frac{q_k}{p_k}$$

# Algorithm (Steps):

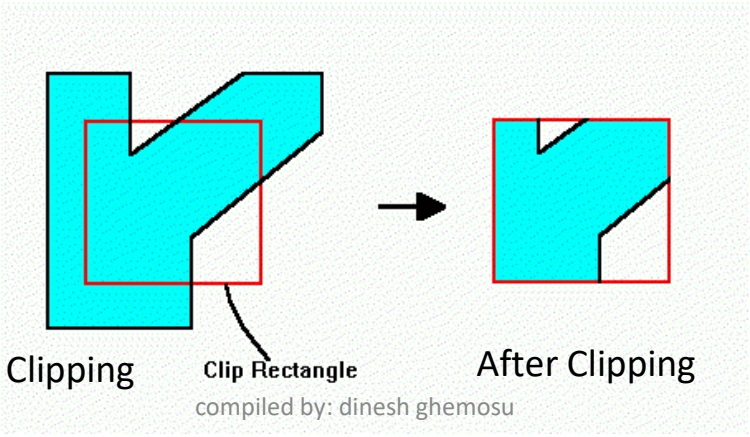
1. The line intersection parameters are initialized to the values as $u_1 = 0$ and $u_2 = 1$.
2. For each clipping boundary, the appropriated values for $p$ and $q$ are calculated and used to find out whether the line can be rejected or intersection parameters are to be adjusted.
3. A value of $r_k$ ( $\frac{p_k}{q_k}$ )is calculated for each of these boundaries and the value of $u_2$ is the minimum of the set consisting of 1 and the calculated $r$-values.
4. If $u_1 > u_2$, the line is completely outside the clip window and it can be rejected
   Else
   The end-points of the clipped line are calculated from the two values of parameter, $u$.
5. When $p < 0$, the parameter $r$ is used to update $u_1$,
   Else
   When p > 0, the parameter r used to update u2.
6. If updation of $u_1$ or $u_2$ results in $u_1 > u_2$, we reject the line.
   Else
   We update the appropriate u paramters only if the new value results in shortening of the boundary.
7. When $p = 0$ and $q < 0$, we discard the line since it is parallel to and outside of this boundary.
8. If the line has not rejected after all the four values of $p$ and $q$ have been tested, then the end-points of the clipped line are determined from values of $u_1$ and $u_2$.

# Polygon Clipping

- Algorithm that will generate one or more close area that are then scan converted for the appropriate area fill. Thus, the output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.

**Algorithm**

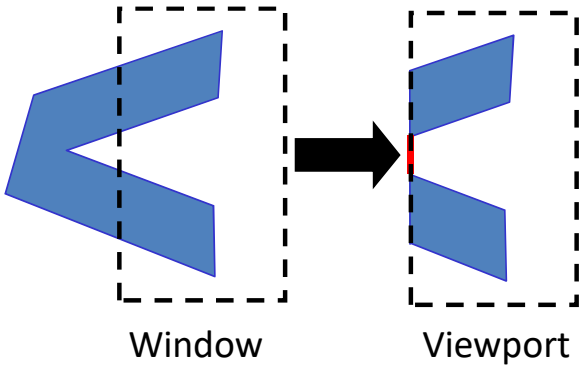
- Sutherland-Hodgeman Polygon Clipping
- Weiler-Atherton Polygon clipping



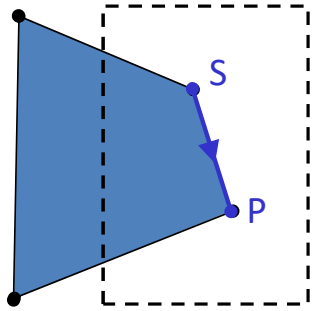
# Sutherland-Hodgeman Polygon Clipping

Introduction

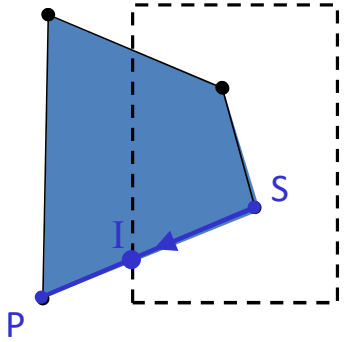
- Uses *divide-and-conquer strategy*.
- A polygon is clipped against each edge of a window in succession.
- At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.



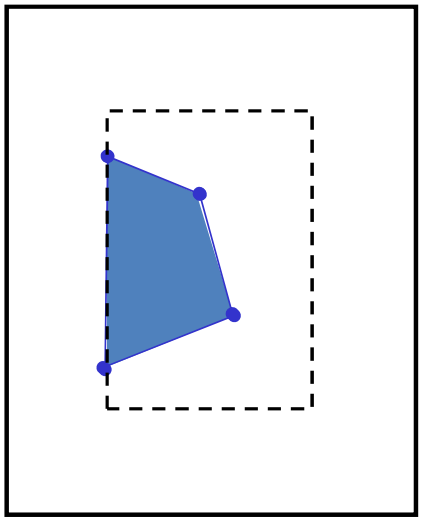
- When we clip a polygon with respect to any particular edge of the window then we need to consider **four different cases**:
- **Case 1**: if the first vertex is outside the window boundary and the second vertex is inside the window then the intersection point of polygon with the boundary edge of window and the vertex which is inside the window is stored in the output vertex list.
- **Case 2**: If both the vertex is inside the window boundary, only the second vertex is added to the output vertex list.
- **Case 3**: if the first vertex is inside window boundary and the second vertex is outside, the edge intersection with the boundary is added to the output vertex list.
- **Case 4**: if both vertices are outside the window boundary, nothing is added to output vertex list.
- Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

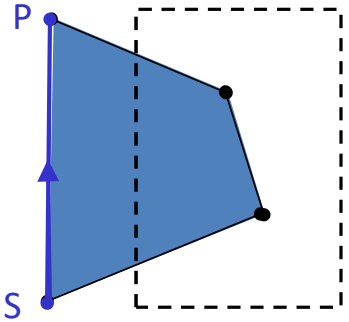



Save Point P

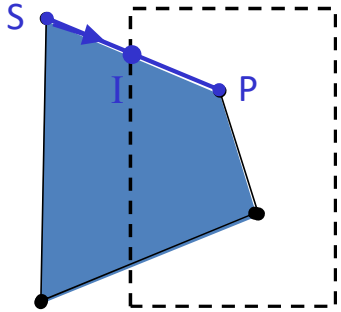



Save Point I

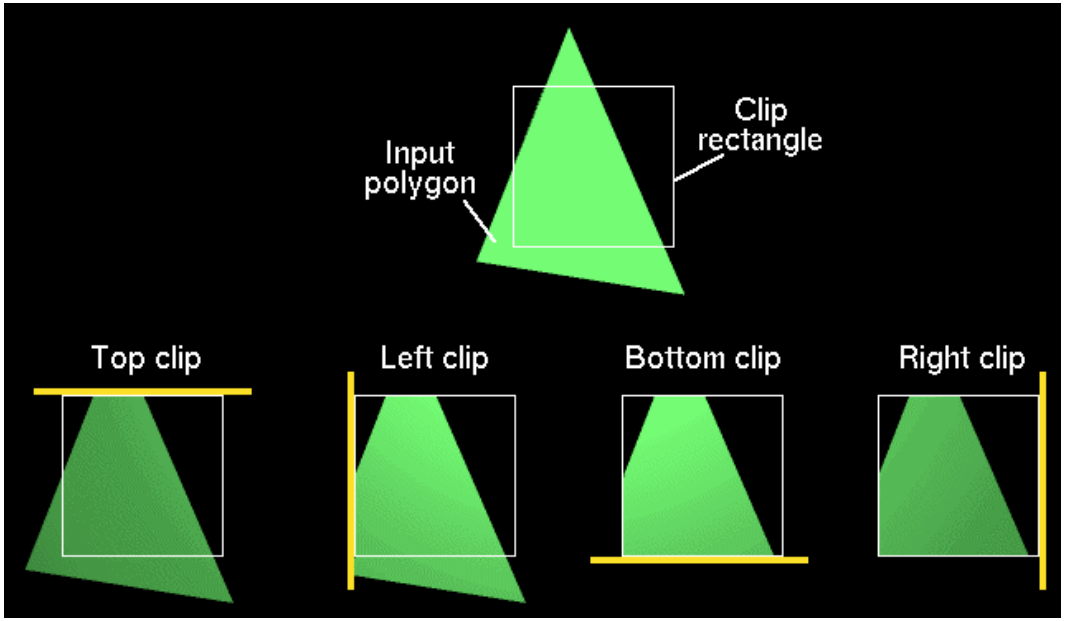





No Points Saved




Save Points I & P


Input
polygon
Clip
rectangle
Top clip
Left clip
Bottom clip
Right clip

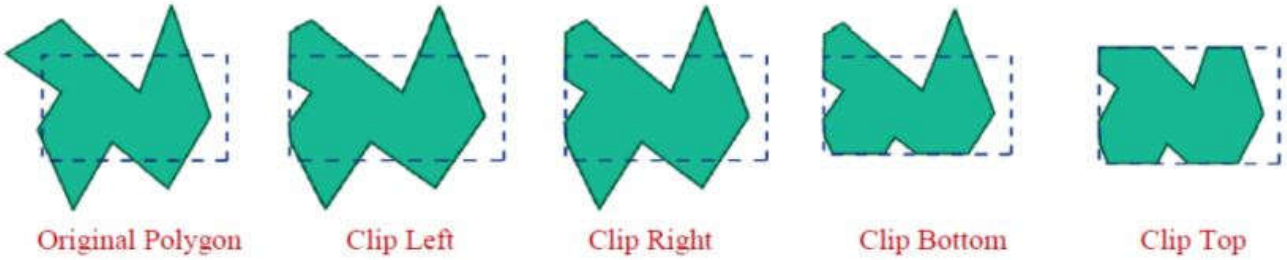



***Fig: Clipping a polygon against successive window boundaries.***

***Algorithm:***

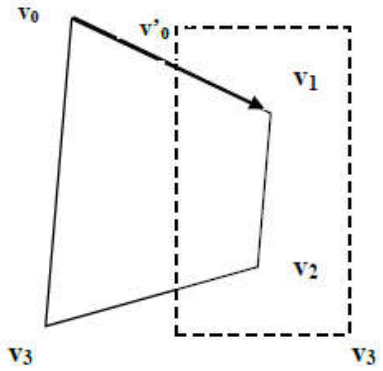
- For each clip window boundary perform the following steps:
  1. Construct an input vertex list ($v_0$, $v_1$… $v_n$) where $v_0 = v_n$.
  2. Create empty output vertex list.
  3. For each pair of adjacent vertices $v_i$ and $v_{i+1}$ perform the following *inside-outside* test:
     a. If out-in ($v_i$ is outside the window boundary and $v_{i+1}$ is inside),
        - *add intersection point $v_i$' and $v_{i+1}$ to the output vertex list.*

     b. If in-in (both $v_i$, $v_{i+1}$ are inside the window boundary),
        - *add $v_{i+1}$ to output vertex list.*

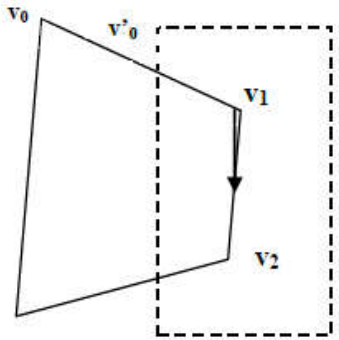
     c. If in-out ($v_i$ is inside the window boundary and $v_{i+1}$ is outside),
        - *add intersection point $v_i$' to output vertex list.*

     d. If out-out (both $v_i$, $v_{i+1}$ are outside the window boundary),
        - *add nothing to output vertex list.*

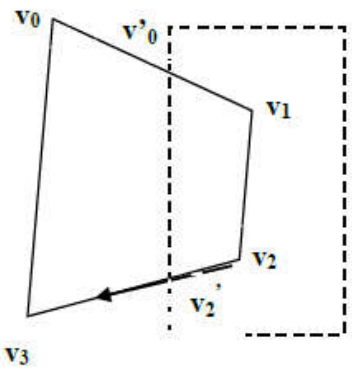
**Example:**



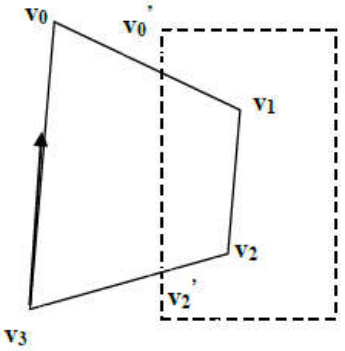

(a) out-in, save $v_0'$, $v_1$



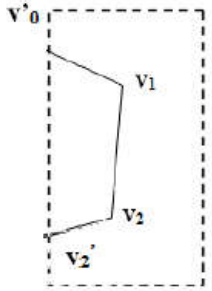

(b) in-in, save $v_2$




(c) in-out, save $v_2'$




(d) out-out, save none




(e) Clipped polygon

**Clipping against left clip boundary:**

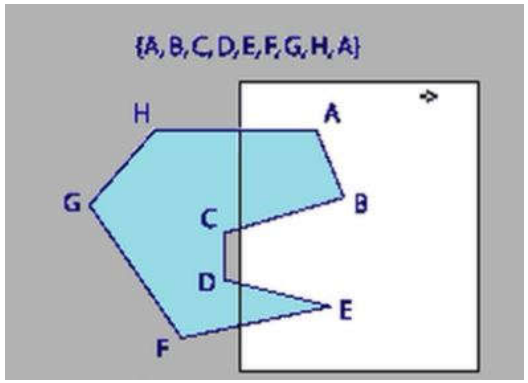
- Input vertex list: $\{v_0, v_1, v_2, v_3\}$
- Output vertex list: { }.
- For vertices $v_0$, $v_1$, *out - in* pair so intersection point $v_0$' and $v_1$ are added to the output vertex list.
- Therefore, output vertex list = $\{v_0', v_1\}$.
- For vertices $v_1$, $v_2$, *in-in* pair so only $v_2$ is added to the output vertex list.
- Now, output vertex list = $\{v_0', v_1, v_2\}$.
- For vertices $v_2$, $v_3$, *in-out* pair so only intersection point $v_2$' is added to the output vertex list.
- Now, output vertex list = $\{ v_0', v_1, v_2, v_2'\}$.
- For vertices $v_3$, $v_0$, *out-out* pair so nothing is added to the output vertex list.
- Finally, output vertex list = $\{ v_0', v_1, v_2, v_2'\}$.
- 
- **Similar, process is repeated for right, bottom, and top boundary.**

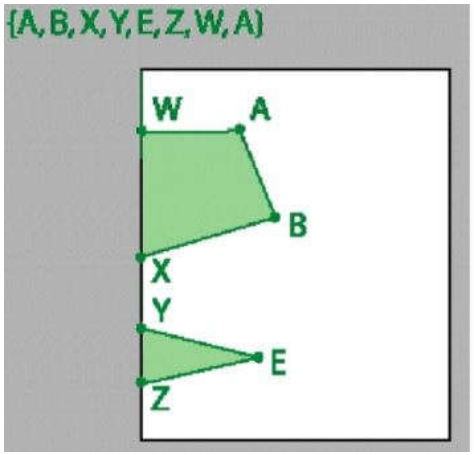
# Can we use line clipping algorithms for polygon clipping?

- NO
- This is because if we use a line clippers to clip a polygon, what we get is the series of unconnected line segments.

# Problem with Sutherland-Hodgman Algorithm

- All convex polygons are correctly clipped by this algorithm but concave polygons may be displayed with some extraneous lines. This occurs when the clipped polygon should have two or more separate sections. But there is only one output vertex list, the last vertex in the list is always joined to the first vertex i.e. we are forming an edge between the last and first vertex. For example:




Note: Edges XY and ZW!

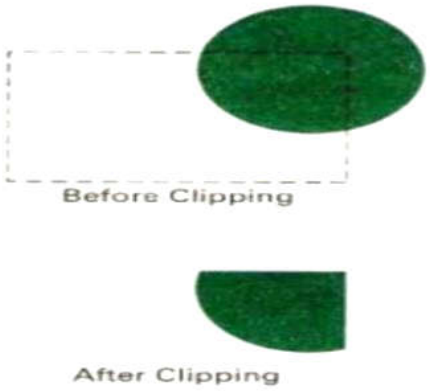
- To overcome this problem:
  - Split the concave polygon into two or more convex polygons and process each convex polygon separately.
  - Use Weiler-Atherton Polygon clipping algorithm.

# Wiler-Atherton Algorithm

- General clipping algorithm for concave polygons with holes
- Produce multiple polygons (with holes)
- Make linked list data structure

# Curve Clipping

- Involves non-linear equations, so requires more processing than for objects with linear boundaries.
- The bounding rectangle (coordinate extent) for a circle or other curved object is used to test for overlap with a rectangular clip window.
- For a circle, we can use the coordinate extents of individual quadrants and then octants before calculating curve-window intersections.
- For an ellipse, we can teste the coordinate extents of individual quadrants.



# Text Clipping

- Two techniques (Method-I) :

1. All or none string clipping
2. All or none character clipping
3. Component character clipping

# ALL or non string clipping

- In this method, we create a rectangular box around the string.
- The boundary position of this rectangle is then compared to the window boundaries and the string is rejected if these two boundaries get overlapped.
- If all of the string is inside the clip window, we keep it else the string is discarded.

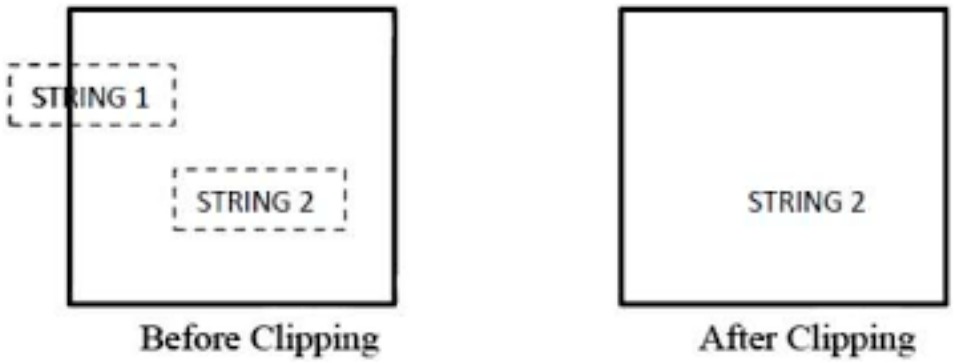



Figure: All or none string clipping

Disadvantages

- Even if a single character of our string is outside the window, we clip the entire text.
- It is faster but an efficient method.

# All or none character clipping

- In this method, we don't reject the entire string if one of its character is outside the clip window.
- The text that is inside the window is displayed fully.

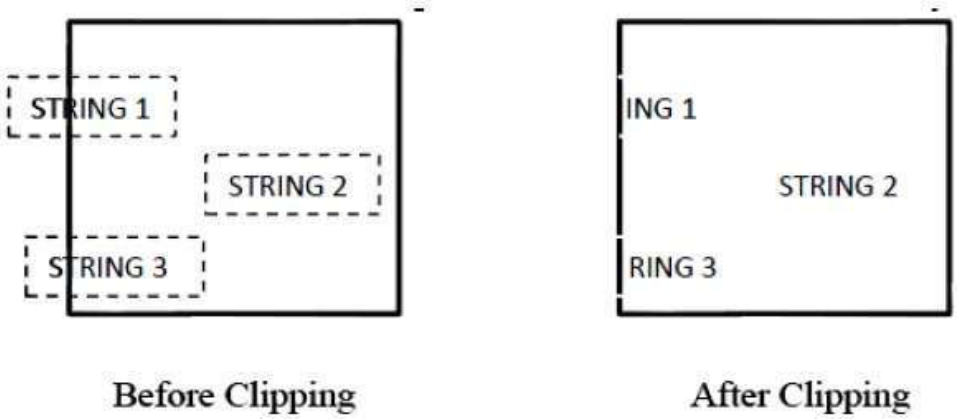



**Figure**: All or none character clipping

# Component Character Clipping

- We can clip the components of individual character also.
- Here, we treat the character as a set of lines or pixels.
- If the individual character overlaps the window boundary then we clip that part of the character which is outside the window.
- Time consuming
- Most accurate method

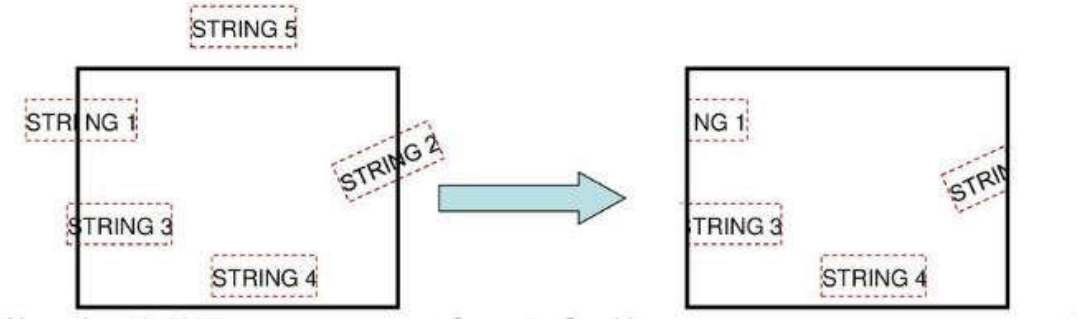



**Figure**: Component character Clipping