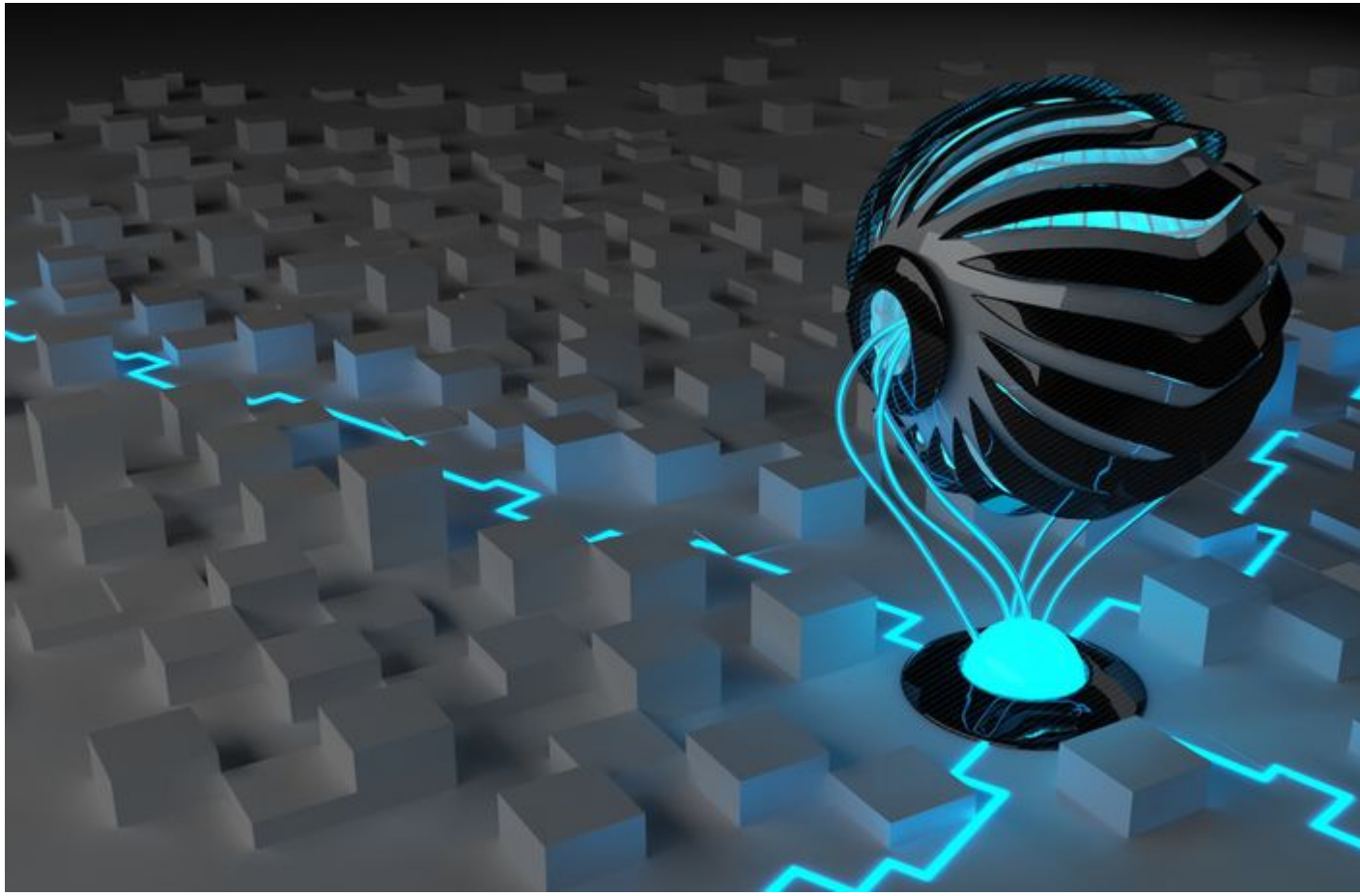
Computer Graphics and Visualization

Chapter 2

Raster Graphics and Algorithms



Prepared by:

Asst. Prof. Sanjivan Satyal

Contents: Raster Graphics and Algorithms [9 hours]

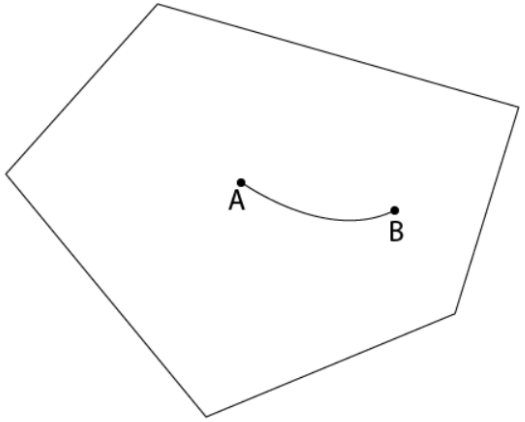
- Rasterizing a Point
- Rasterizing a Straight Line: DDA Line Algorithm, Bresenham's Line Algorithm
- Rasterizing a Circle and an Ellipse: Mid-Point Circle and Ellipse Algorithm
- Scan-Line Polygon Fill Algorithm
- Scan-Line Fill of Curved Boundary Areas
- Boundary-Fill Algorithm
- Flood-Fill Algorithm
- Point Clipping
- Line Clipping
  - Cohen-Sutherland Line Clipping
  - Liang-Barsky Line Clipping
- Polygon Clipping
  - Weiler-Atherton Polygon Clipping
- Text Clipping

Area Filling

Polygon:

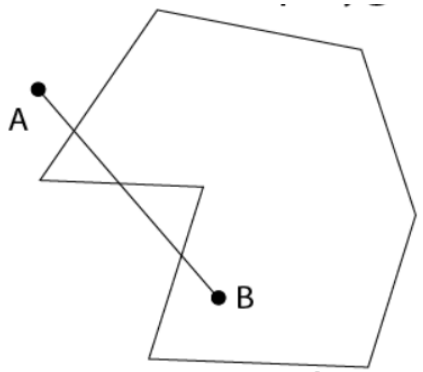
- Polygon is a representation of the surface. It is primitive which is closed in nature. It is formed using a collection of lines. It is also called as many-sided figure. The lines combined to form polygon are called sides or edges. The lines are obtained by combining two vertices.
- Example of Polygon:Triangle, Rectangle, Hexagon, Pentagon
- Types of Polygon
  - Convex
  - Concave

- A polygon is called **convex** of line joining any two interior points of the polygon lies inside the polygon.



Convex polygon

- The **Concave** polygon is a polygon in which the line segment joining any two points within the polygon may not lie completely inside the polygon



Concave polygon

Filled Area Primitives

- Figures on a computer screen can be drawn using polygons. To fill those figures with color, we need to develop some algorithm.
- Polygon are easier to process since they have linear boundaries.
- There are two basic approaches to area filling in raster systems. One way to fill an area is to determine the overlap intervals for scan lines that cresses the area. Another method for area filling is to start from a given interior position and points outward from this until a specified boundary is met

- Filled Area primitives are used to filling solid colors to an area or image or polygon. Filling the polygon means highlighting its pixel with different solid colors. Following are two filled area primitives:
- **Seed Fill Algorithm**
- **Scan Fill Algorithm**

Seed Fill Algorithm:

- In this seed fill algorithm, we select a starting point called seed inside the boundary of the polygon. The seed algorithm can be further classified into two algorithms: **Flood Fill and Boundary filled.**

Seed Fill Algorithm

1. Boundary fill Algorithm
2. Flood Fill Algorithm

Scan fill Algorithm

1. Scan Line Algorithm

Boundary filled Algorithm

- Boundary Fill Algorithm starts at a pixel inside the polygon to be filled and paints the interior proceeding outwards towards the boundary.

- This algorithm works **only if** the color with which the region has to be filled and the color of the boundary of the region are different. If the boundary is of one single color, this approach proceeds outwards pixel by pixel until it hits the boundary of the region.

- **Boundary Fill Algorithm is recursive in nature.** It takes an interior point(x, y), a fill color, and a boundary color as the input.

- The **algorithm** starts by checking the color of (x, y). If it's color is not equal to the fill color and the boundary color, then it is painted with the fill color and the function is called for all the neighbors of (x, y).

- If a point is found to be of fill color or of boundary color, the function does not call its neighbors and returns. This process continues until all points up to the boundary color for the region have been tested.

- The boundary fill algorithm can be implemented by 4-connected pixels or 8-connect pixels.

Flood Fill Algorithm

- Flood fill algorithm is applicable when we want to fill an area that is not defined with in a single-color boundary. If fill area is bounded with different color, we can paint that area by replacing a specified interior color instead of searching of boundary color value.

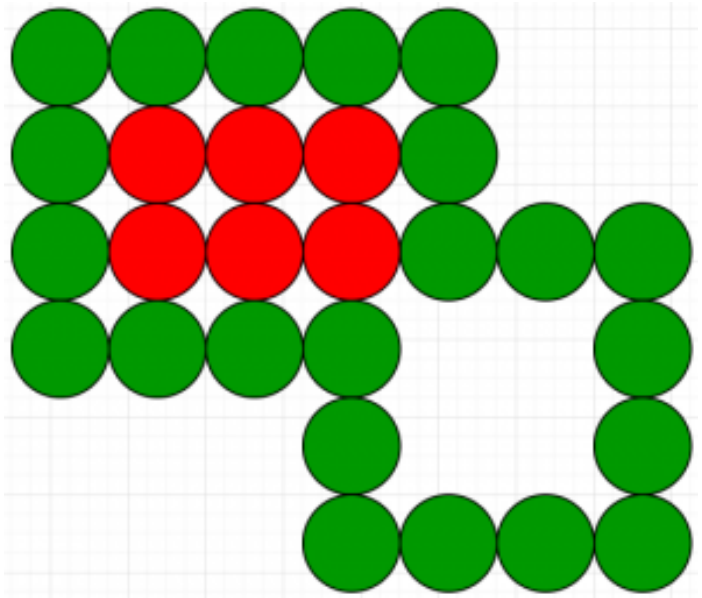
- This approach is called flood fill algorithm.

- We start from a specified interior pixel (x , y) and reassign all pixel values that are currently set to a given interior color with desired fill color.

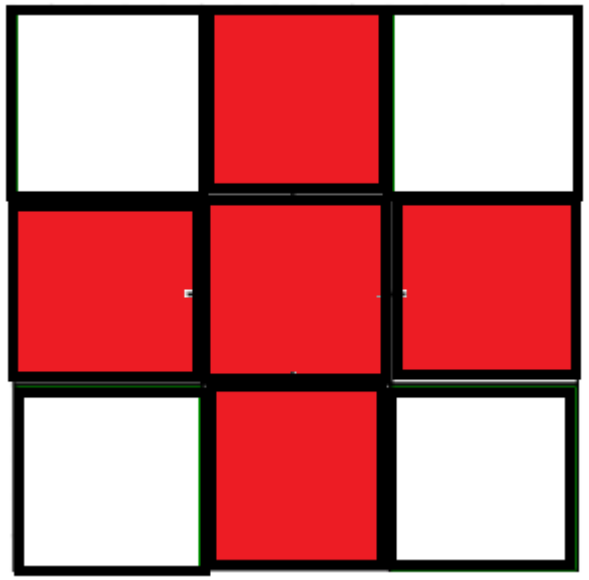
- Once again the algorithm relies on the four-connect or eight connect method of filling color pixels. But instead of looking for the boundary color, it looks for all adjacent pixels that are a part of the interior.

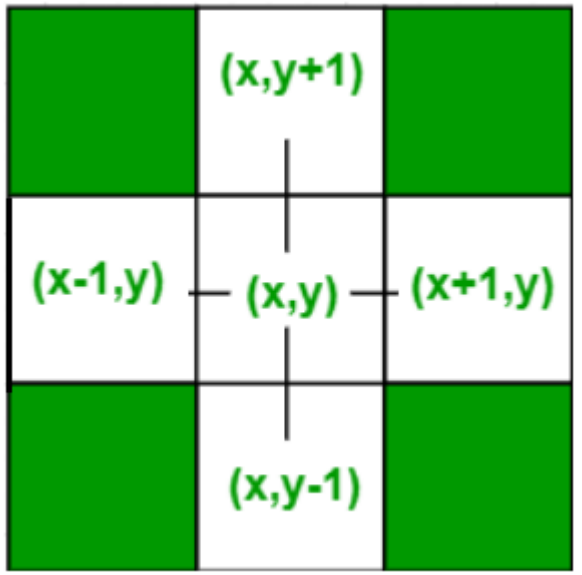
```
BoundaryFill4(x, y, fillColor, boundaryColor)
{
   if (getPixel(x, y) != boundaryColor &&
      getPixel(x, y) != fillColor)
   {
      setPixel(x, y, fillColor);

      BoundaryFill4(x + 1, y, fillColor, boundaryColor);
      BoundaryFill4(x - 1, y, fillColor, boundaryColor);
      BoundaryFill4(x, y + 1, fillColor, boundaryColor);
      BoundaryFill4(x, y - 1, fillColor, boundaryColor);
   }
}
```

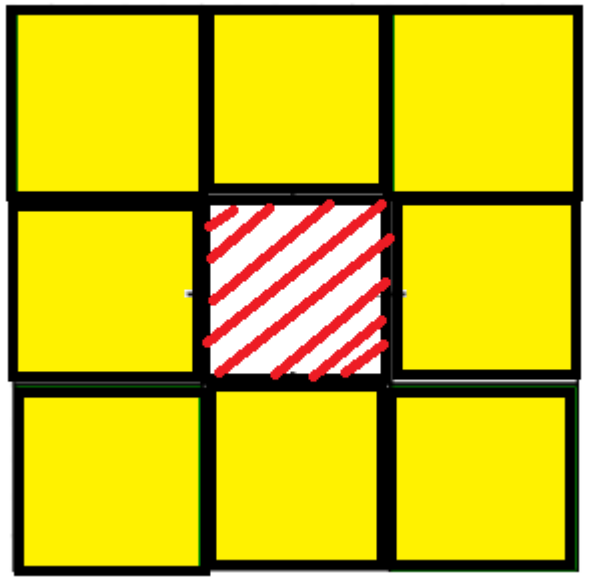


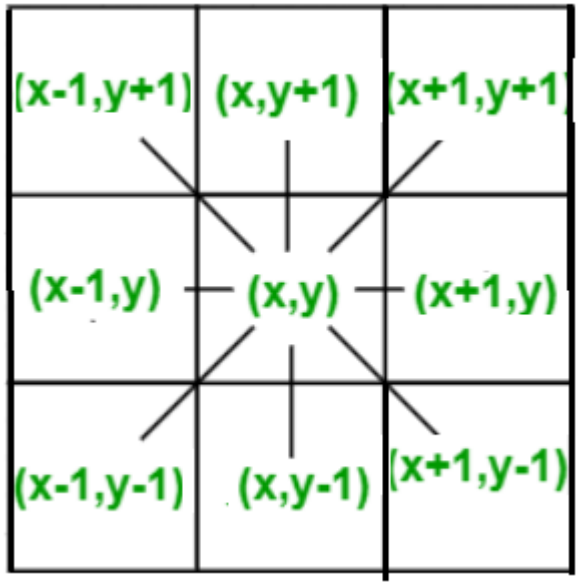
- **4-connected pixels :** After painting a pixel, the function is called for four neighboring points. These are the pixel positions that are right, left, above, and below the current pixel. Areas filled by this method are called 4-connected.

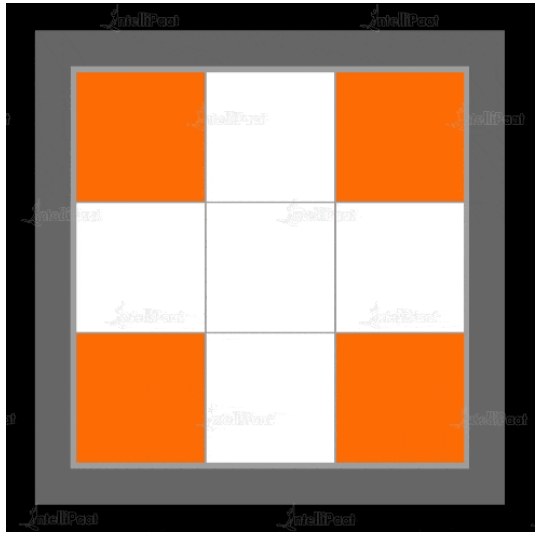


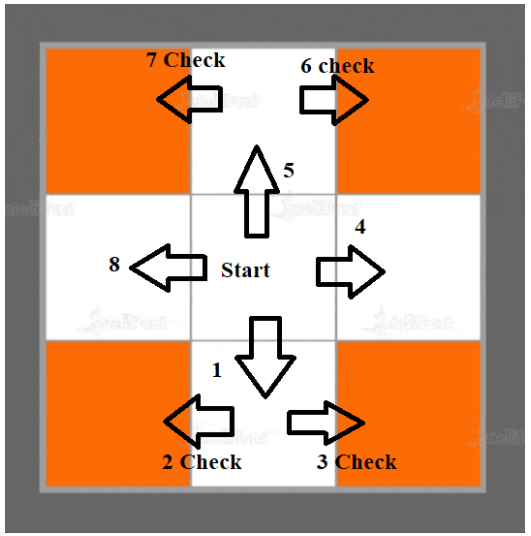


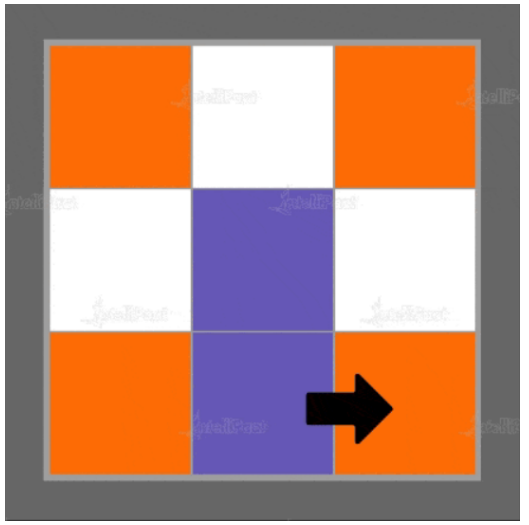
- **8-connected pixels :** More complex figures are filled using this approach. The pixels to be tested are the 8 neighboring pixels, the pixel on the right, left, above, below and the 4 diagonal pixels. Areas filled by this method are called 8-connected

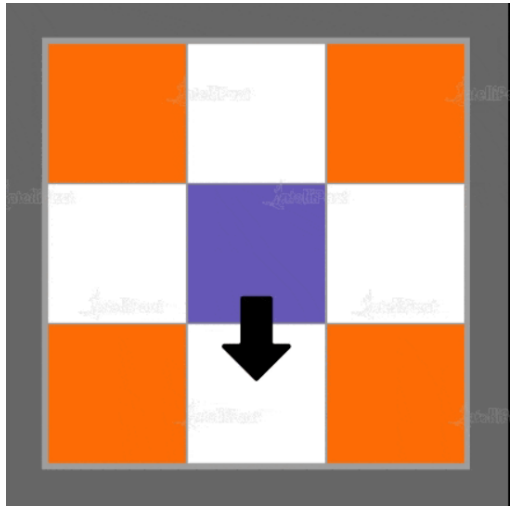


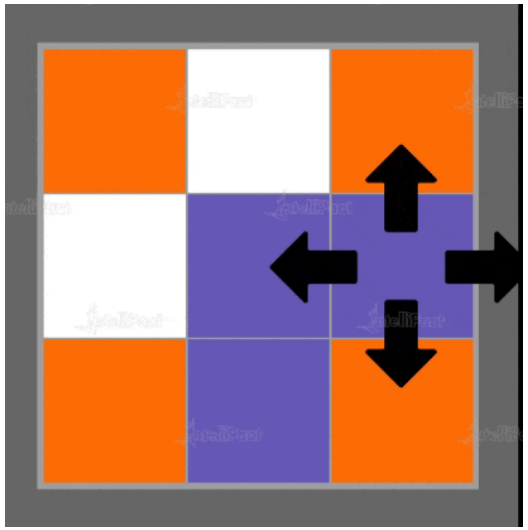


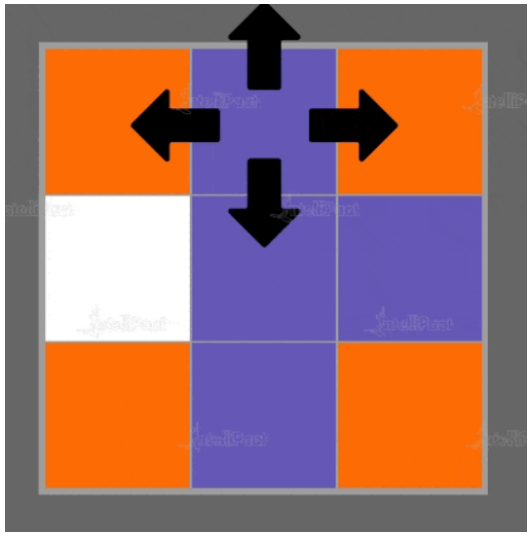


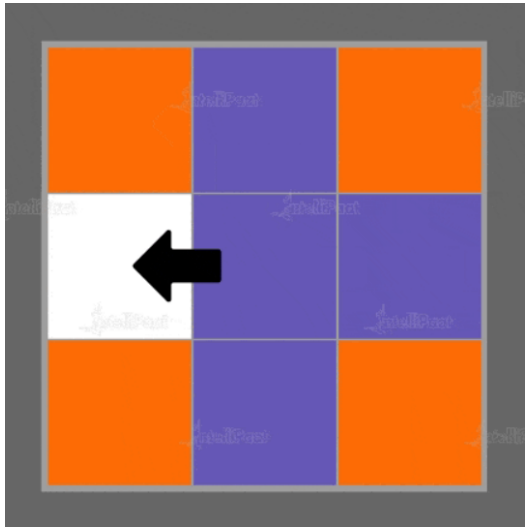

7 Check
6 check
5
4
8
Start
1
2 Check
3 Check

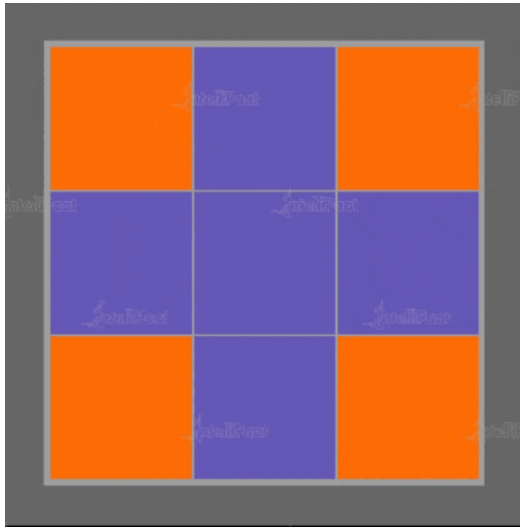












Here are the basic steps of how the algorithm works:

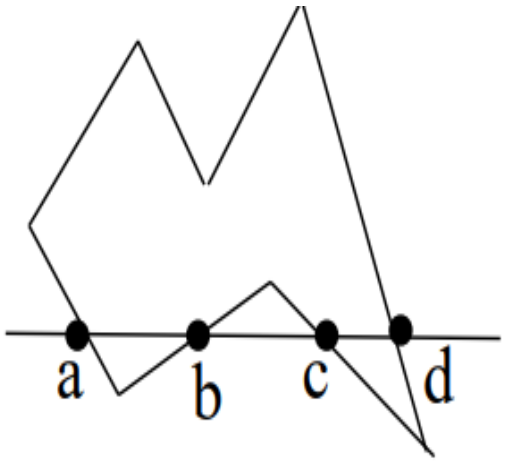
- **Step 1**: Picking Starting Pixels: The algorithm begins by choosing a starting pixel within the image that needs to be filled.
- **Step 2**: Changing the Color of the Starting Pixel: Then, the starting pixel's color is changed to the new desired color.
- **Step 3**: Traversal of Neighboring Pixels: Then, it will examine the neighboring pixels of the starting pixel. For each neighbor, it checks if it's within the bounds of the image and has the same color as the starting pixel's original color.
- **Step 4**: Changing the Color of Neighboring Pixels: If a neighboring pixel meets the criteria (same color as the starting pixel's original color and within image bounds), it's colored with the new color.
- **Step 5**: Repetition of Step 4: This process continues recursively (using a stack or recursive function calls) or iteratively (using a queue or loop) for all eligible neighboring pixels.
- **Step 6:** Final Filling of All Pixels: The algorithm keeps expanding outward and examines the neighboring pixels of the newly filled pixels. This process is continued until all pixels within the enclosed area with the original color have been filled with the new color.

Difference between Boundary and Flood fill Algorithm

Boundary fill Algorithm	Flood fill Algorithm
Area filling is started inside a point within a boundary region and fill the region with in the specified color until it reaches the boundary.	Area filling is started from a point and it replaces the old color with the new color.
It is used in interactive packages where we can specify the region boundary.	It is used when we cannot specify the region boundary.
It is less time consuming.	It consumes more time.
It searches for boundary.	It searches for old color.

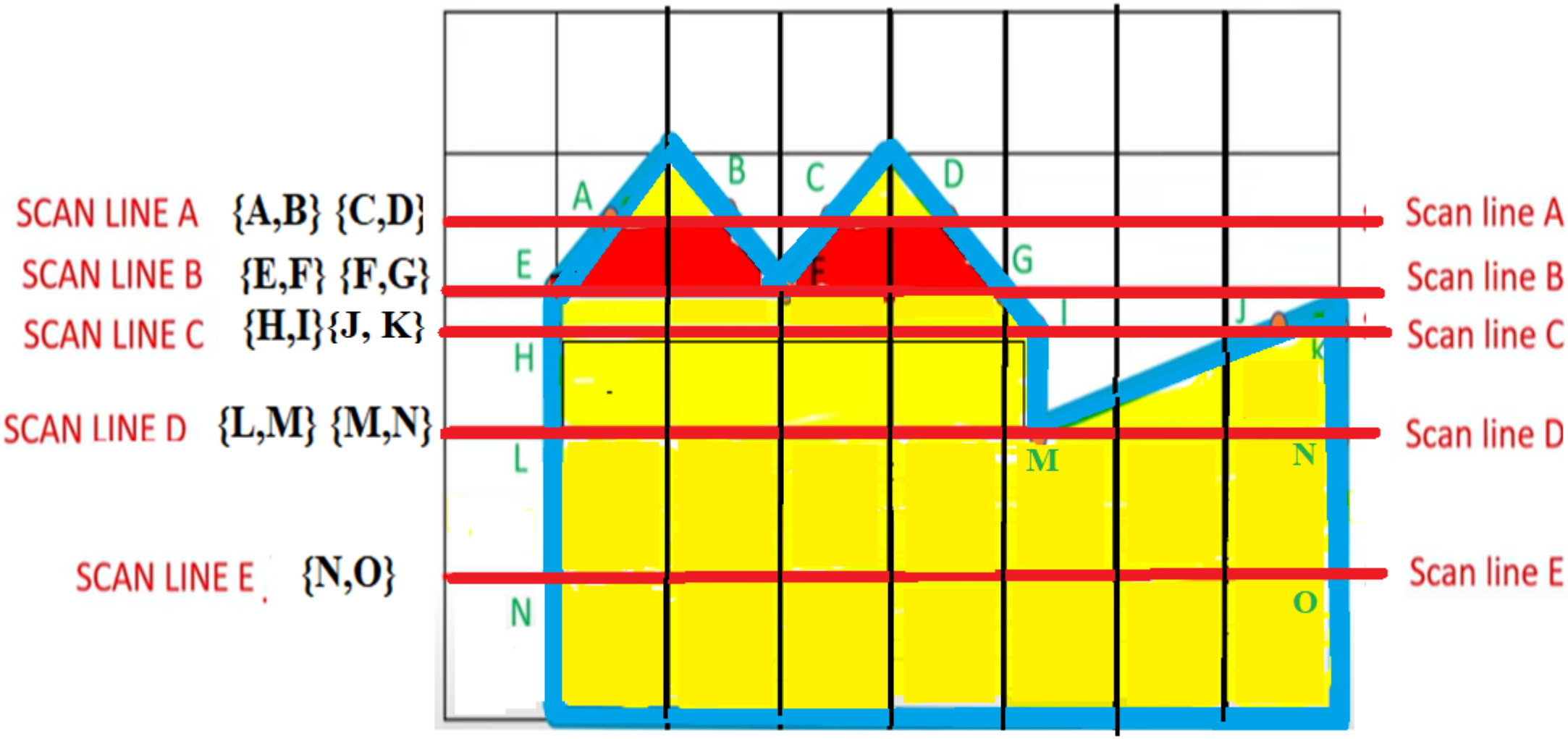
Scan Line polygon fill Algorithm

- Scan-line polygon filling is a basic graphics method in C used to display solid forms on a screen.
- Scanning the image line by line, locating intersections between the scan-line and the polygon's edges, and then filling the pixels between these intersections with the appropriate color is how this approach works.
- It is necessary for producing realistic images in computer graphics programs and games.

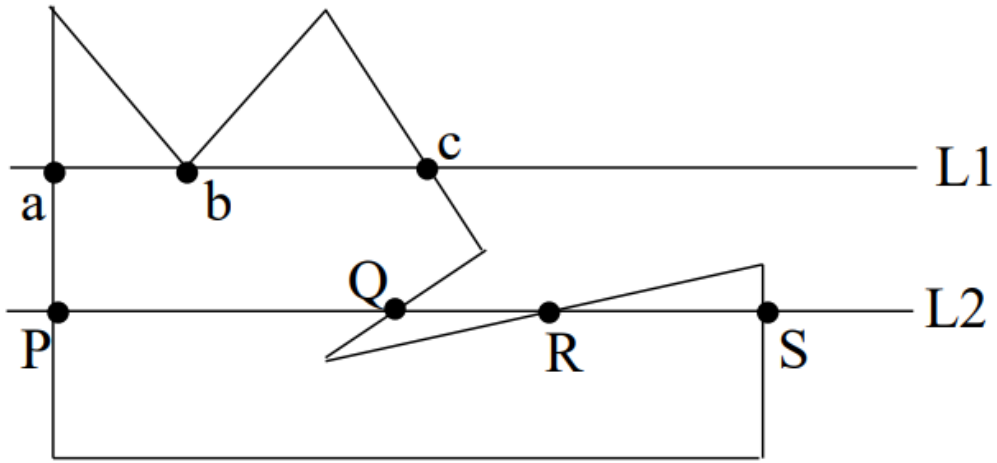


The basic scan-line algorithm is as follows:

- Find the intersections of the scan line with all edges of the polygon
- Sort the intersections points from left to right. i.e. (a, b, c, d)
- Fill in all pixels between pairs of intersections that lie interior to the polygon. i.e. (a, b) & (c, d).


SCAN LINE A {A,B} {C,D}
SCAN LINE B {E,F} {F,G}
SCAN LINE C {H,I}{J, K}
SCAN LINE D {L,M} {M,N}
SCAN LINE E {N,O}
Scan line A
Scan line B
Scan line C
Scan line D
Scan line E
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

Problem:



- For scan line L1 a, b & c are interior point, therefore we take pairwise points (a, b) & (b, c) and fill all the pixel between these points.
- For scan line L2, we should only take pairwise points (P, Q) & (R, S) because (Q, R) is not part of polygon.
- For scan line L1, we took the vertex 'b' twice i.e.(a, b) & (b, c) but for scan line 'L2' we did not take 'Q' twice.

How to determine this?

Solution:

- Make a clockwise or anticlockwise traverse on the edge.
- If 'y' is monotonically increasing or decreasing and direction of 'y' changes, then we have take the vertex twice, otherwise take vertex only once.

Scan-Line Fill of Curved Boundary Areas

It requires more work than polygon filling, since intersection calculation involves nonlinear boundary. For simple curves such as circle or ellipses, performing a scan line fill is straight forward process. We only need to calculate the two scan-line intersection on opposite sides of the curve. Then simply fill the horizontal spans of pixel between the boundary points on opposite side of curve. Symmetries between quadrants are used to reduce the boundary calculation. We can fill generating pixel position along curve boundary using mid-point method

Fig: Interior fill of an elliptical arc

Clipping Operations

- **Clipping**
  - Any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space
- Applied in World Coordinates
- Adapting Primitive Types
  - Point
  - Line
  - Area (or Polygons)
  - Curve

2D Clipping (Application)

- Extracting part of a defined scene for viewing
- Identifying visible surfaces in three-dimensional views
- Creating objects using solid-modeling procedures
- Displaying a multi-window environment
- Anti-aliasing line segments or object boundaries
- Drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating

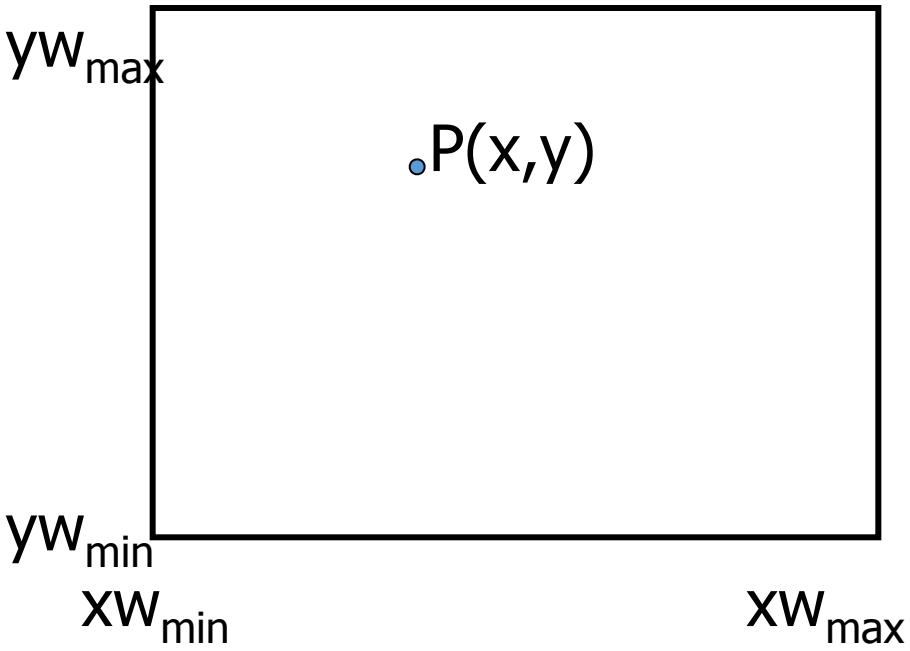
Point Clipping

- Assuming that the clip window is a rectangle in standard position
- For a clipping rectangle in standard position, we save a 2-D point P(x,y) for display if the following inequalities are satisfied:

$$x_{min} \leq x \leq x_{max}$$
$$y_{min} \leq y \leq y_{max}$$

- If any one of these four inequalities is not satisfied, the point is clipped (not saved for display)
- Where $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ define the clipping window can be either the world coordinate window boundaries or viewport boundaries

Point Clipping



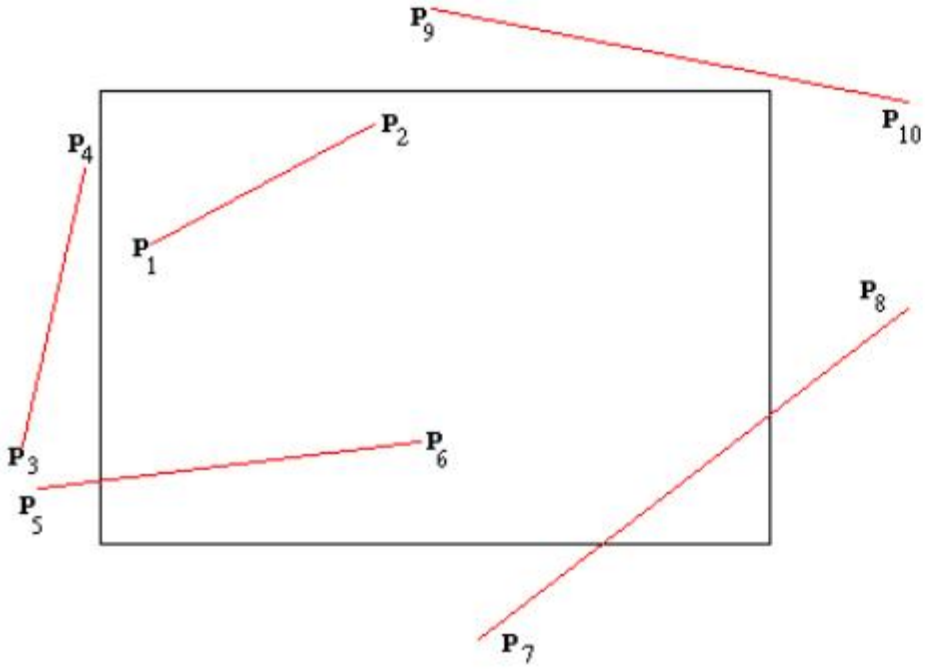
If P(x,y) is inside the window?

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

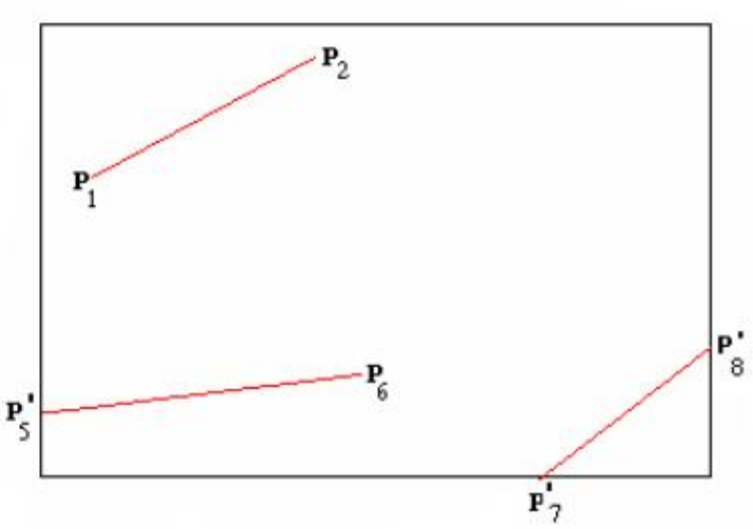
$$yw_{\min} \leq y \leq yw_{\max}$$

Line clipping

Line clipping against a rectangular window



before clipping



after clipping

Line clipping

- **inside – outside test**
- **Completely Inside:** A line with both endpoints **inside** all clipping boundaries, such as the line from $\mathbf{p}_1$ to $\mathbf{p}_2$, is saved
- **Completely Outside:** A line with both endpoints **outside** any one of the clip boundaries, such as the line from $\mathbf{p}_3$ to $\mathbf{p}_4$, is not saved
- If the line is not completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries

Line clipping

- 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

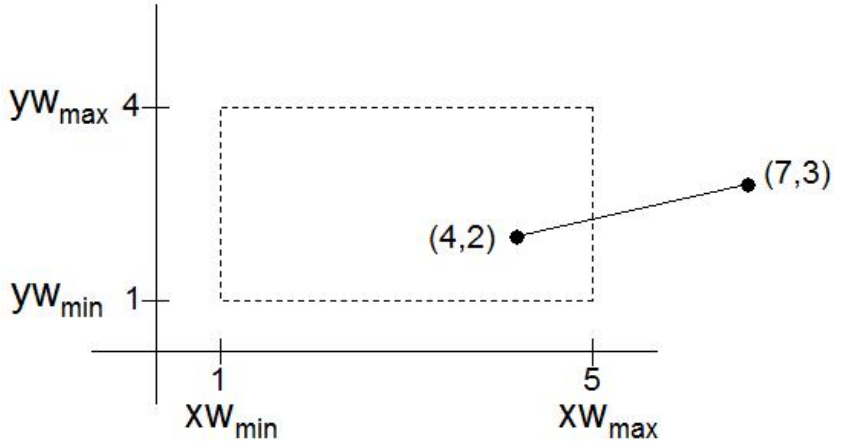
$$x = x_1 + u\,(x_2 - x_1)$$

$$y = y_1 + u\,(y_2 - y_1), \quad 0 \le u \le 1$$

- If the value of **u** for an intersection with a rectangle boundary edge is **outside** the range 0 to 1, the line **does not intersect** with that boundary
- If the value of **u** is **within** the range from 0 to 1, the line segment **intersect** with that boundary

Line clipping

- Solve **u** by substitute **x = xw_{max}** in
- **$x = x_1 + u (x_2 – x_1)$**
- **$5 = 4 + u (7 – 4)$**
- **$u = 1/3$**



- The value of **u** is **within** the range from 0 to 1, the line segment **intersects** with the boundary **xw_{max}**
- Solve **y** by substitute **u** in **$y = y_1 + u (y_2 – y_1)$**
- **$y = 2 + 1/3 (3 – 2) = 2.33$**
- Intersection point at **xw_{max}** is (**5, 2.33**)

Line clipping Algorithms

- Speeds up the processing of the line segment by performing initial tests that reduce the number of intersections that must be calculated
- **Line clipping Algorithm**
  - **Cohen-Sutherland line clipping**
  - **Liang-Barsky line clipping**

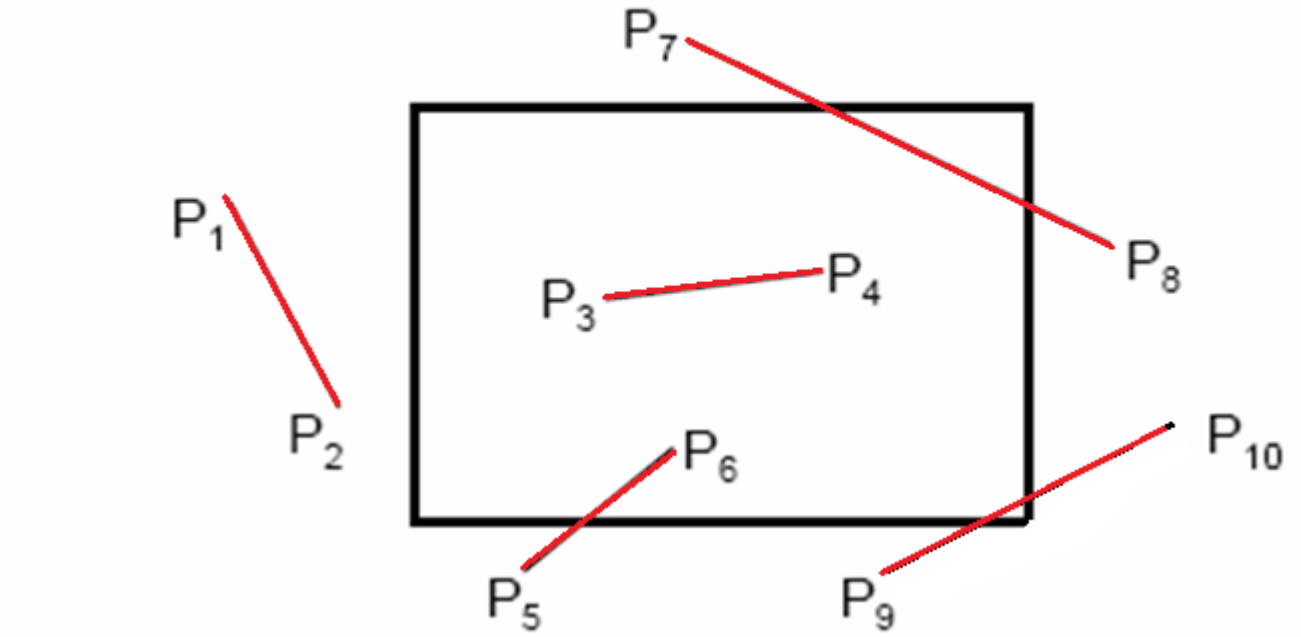
Cohen-Sutherland line clipping

- Divide the line clipping process into two phases:
  - Identify those lines which intersect the clipping window and so need to be clipped.
  - Perform the clipping
- All lines fall into one of the following clipping categories:
  - **Visible**: Both end points of the line lie within the window
  - **Not visible**: The line definitely lies outside the window. This will occur if the line from (x_1,y_1) to (x_2,y_2) satisfies any one of the following inequalities:
  - **Clipping candidate**: the line is in neither category 1 nor 2

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

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

Find the part of a line inside the clip window

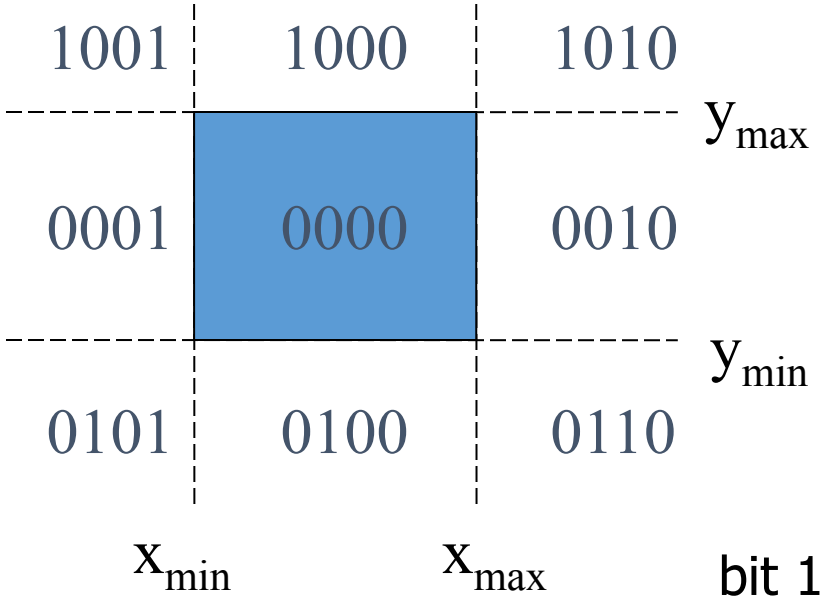


p_3p_4 is in category 1(Visible)

p_1p_2 is in category 2(Not Visible)

$p_5p_6, p_7p_8, p_9p_{10}$ is in category 3(Clipping candidate)

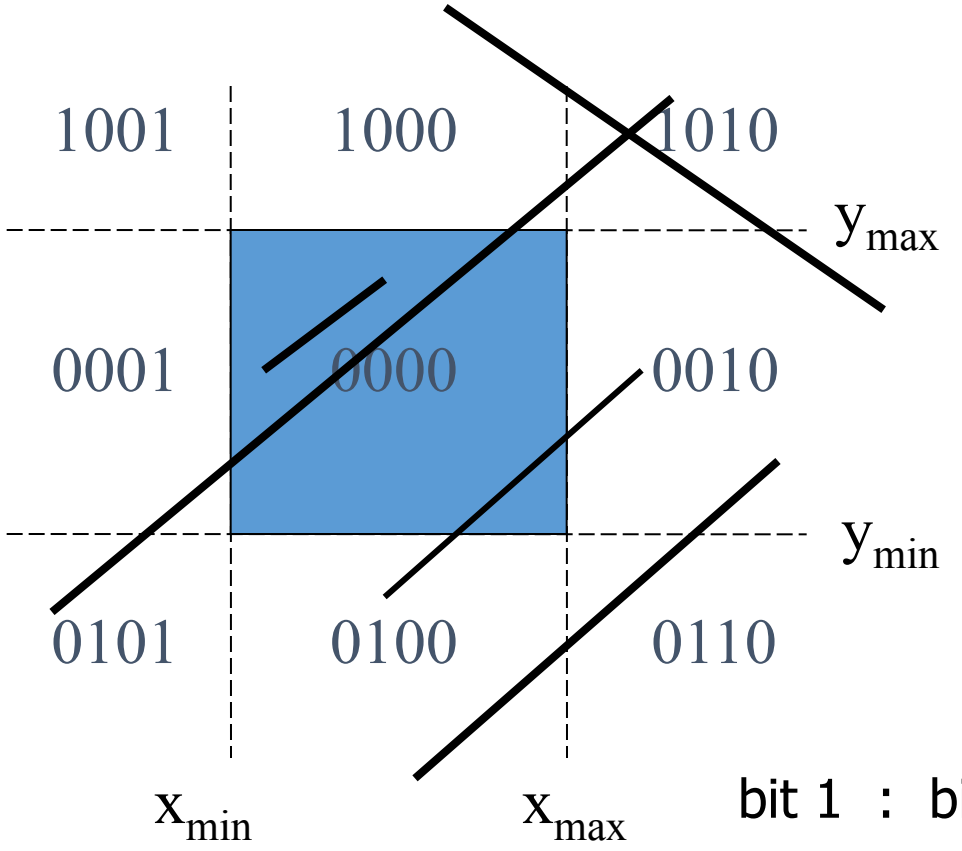
- Assign a four-bit pattern (Region Code) to each endpoint of the given segment. The code is determined according to which of the following nine regions of the plane the endpoint lies in.



bit 1 : bit 2 : bit 3 : bit 4

Top : Bottom : Right : Left:

- Of course, a point with code 0000 is inside the window



bit 1 : bit 2 : bit 3 : bit 4

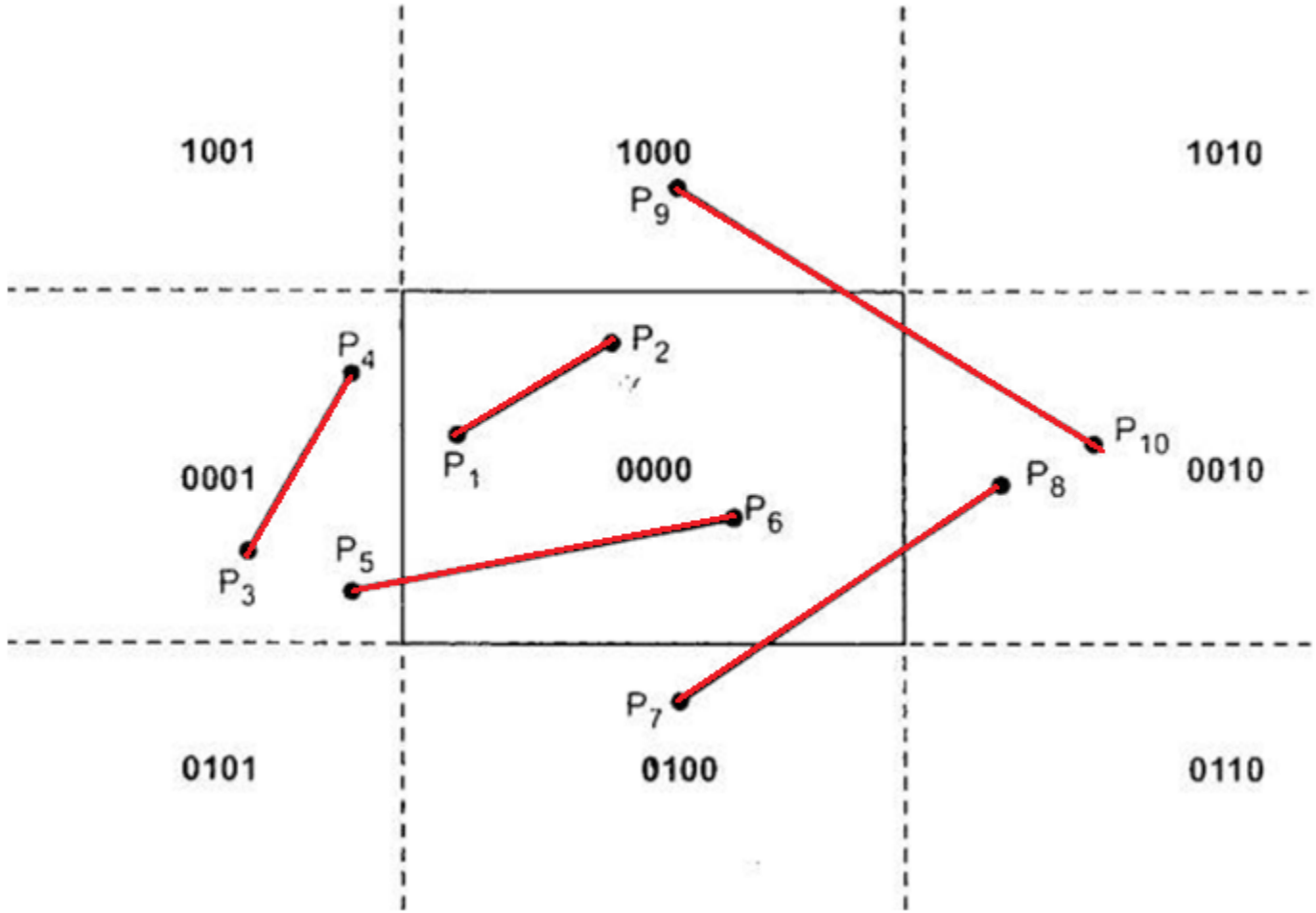
Top : Bottom : Right : Left:

- if both endpoint codes are **0000** ➔ the line segment is visible (inside)

else do the logical **AND** of the two endpoint codes

- **not completely 0000** ➔ the line segment is not visible (outside)
- **completely 0000** ➔ the line segment maybe partially inside (and outside)

- Lines that cannot be identified as being completely inside or completely outside a clipping window are then checked for intersection with the window border lines


1001
1000
1010
P9
P4
P2
P10
P1
0001
0000
0010
P8
P6
P5
P3
P7
0101
0100
0110

Cohen-Sutherland line clipping

Line	End Point Codes		Logical ANDing	Result
P_1 P_2	0000	0000	0000	Completely visible
P_3 P_4	0001	0001	0001	Completely invisible
P_5 P_6	0001	0000	0000	Partially visible
P_7 P_8	0100	0010	0000	Partially visible
P_9 P_{10}	1000	0010	0000	Partially visible

Cohen-Sutherland line clipping

- For a line with endpoint (x_1, y_1) and (x_2, y_2)

$$m = (y_2 - y_1) / (x_2 - x_1)$$

- **Intersection points with the clipping boundary:**
- The intersection with **vertical boundary ($x=x_{min}$** or **$x=x_{max}$)**, the **y** coordinate can be calculated as:
- **Left: $x= x_{min}$**

$$y = y_1 + m(x_{min} - x_1)$$

- **Right:$x = x_{max}$**

$$y = y_1 + m(x_{max} - x_1)$$

Cohen-Sutherland line clipping

- For a line with endpoint (x_1, y_1) and (x_2, y_2)

$$m = (y_2 - y_1) / (x_2 - x_1)$$

- **Intersection points with the clipping boundary:**
- The intersection with a **horizontal boundary** ($y=y_{min}$ or $y=y_{max}$.), the **x** coordinate can be calculated as
- **Top:** $y= y_{max}$

$$x = x_1 + (y_{max} - y_1) / m$$

- **Bottom:** $y = y_{min}$

$$x = x_1 + (y_{min} - y_1) / m$$

Cohen-Sutherland line clipping Algorithm

Algorithm

- Given a line segment with endpoint P1=(x1,y1) and P2=(x2,y2)
- Compute the 4-bit out codes for the two endpoints of the line segment
- If out-code of P1 = out-code of P2=0000 (P1 II P2=0000) then lines lies completely inside the window, so draw the line segments
- else if out-code of P1 = out-code of P2 ≠0000 (P1&&P2≠0000) then line lies completely outside the window, so discard the line segment
- else if out-code of P1 is non zero and out-code of P2 =0000 (P1&&P2=0000) or vise-versa then compute the intersection of the line segment with window boundary and discard the portion of segment that falls completely outside the window. Assign a new four-bit code to the intersection and repeat until either 3 or 4 above are satisfied.

Classwork

- ***Consider a rectangle clipping window with A(20, 20), B(90, 20), C(90, 70) & D(20, 70). Clip the line P1P2 with P1(10, 30) & P2(80, 90) using Cohen-Sutherland line clipping algorithm.***

Classwork

- ***Consider a rectangle clipping window with A(20, 20), B(90, 20), C(90, 70) & D(20, 70). Clip the line P1P2 with P1(10, 30) & P2(80, 90) using Cohen-Sutherland line clipping algorithm.***

Region code for $P1(10, 30)$:

$x - xw_{min} = 10 - 20 = -10$	1	L
$xw_{max} - x = 90 - 10 = 80$	0	R
$y - yw_{min} = 30 - 20 = 10$	0	B
$yw_{max} - y = 70 - 30 = 40$	0	T

Region code for $P2(80, 90)$:

$x - xw_{min} = 80 - 20 = 60$	0	L
$xw_{max} - x = 90 - 80 = 10$	0	R
$y - yw_{min} = 90 - 20 = 70$	0	B
$yw_{max} - y = 70 - 90 = -20$	0	T

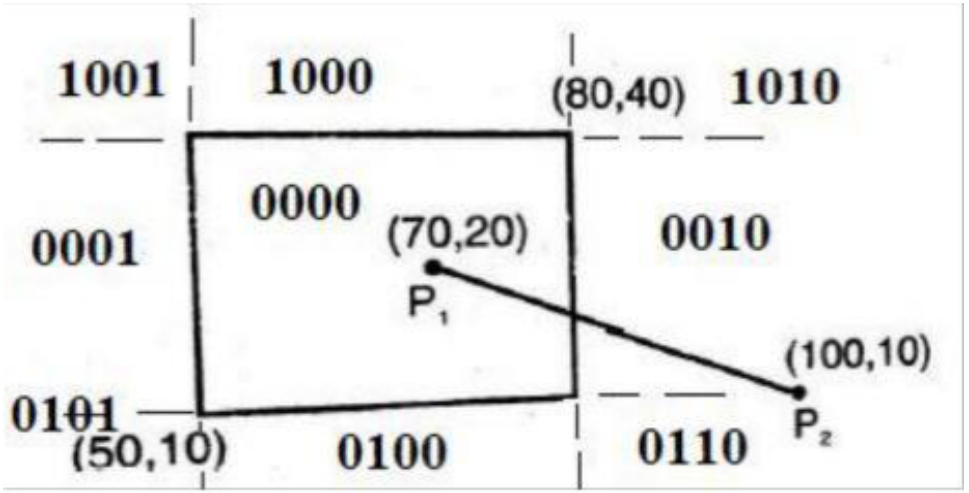
P_1=(20, 38.57) & P_2= (56.67, 70).

Classwork

- **Use the Cohen-Sutherland algorithm to clip the line segment P1(70,20) and P2(100,10) against the window with lower left-hand corner (50,10) and upper right-hand corner (80,40).**

Classwork

- Use the Cohen-Sutherland algorithm to clip the line segment P1(70,20) and P2(100,10) against the window with lower left-hand corner (50,10) and upper right-hand corner (80,40).



P1=0000
P2=0010

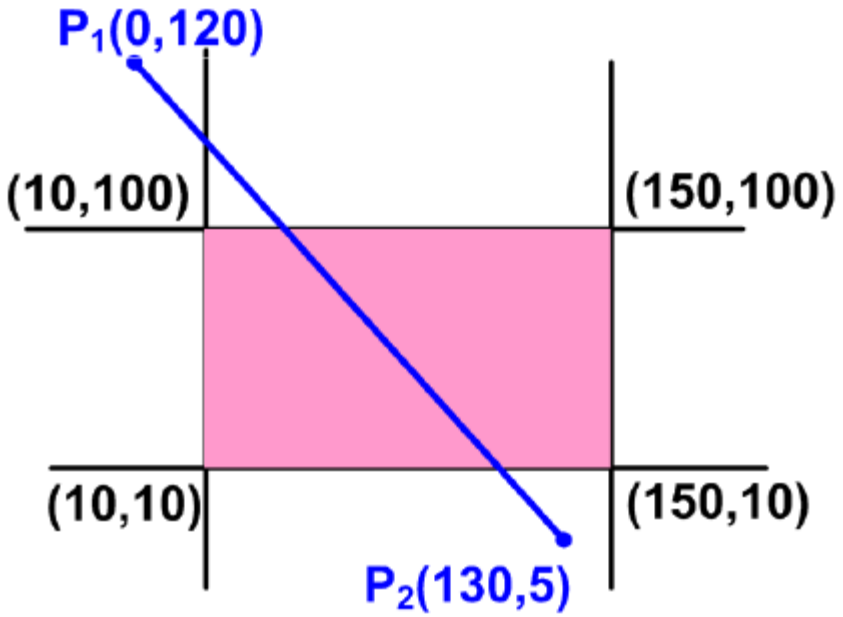
Logical AND of the region codes: 0000
Determine the intersection point:
Since P2 is outside to the right, we need to clip it against the right edge of the window x=80

$$y = y_1 + m(x_{max} - x_1)$$

The intersection point is (80,16.67)

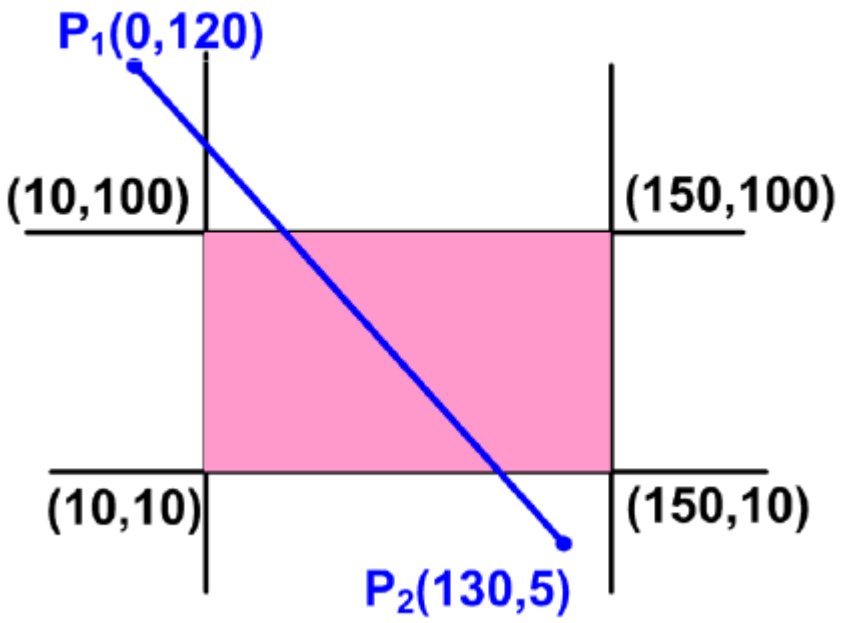
Classwork

- **Clip a line with end point A(5,30), B(20,60) against a clip window with lower most left corner at P1 (10,10) and upper right corner at P2(100, 100)**
- Use the Cohen-Sutherland algorithm to clip the line segment P1, P2

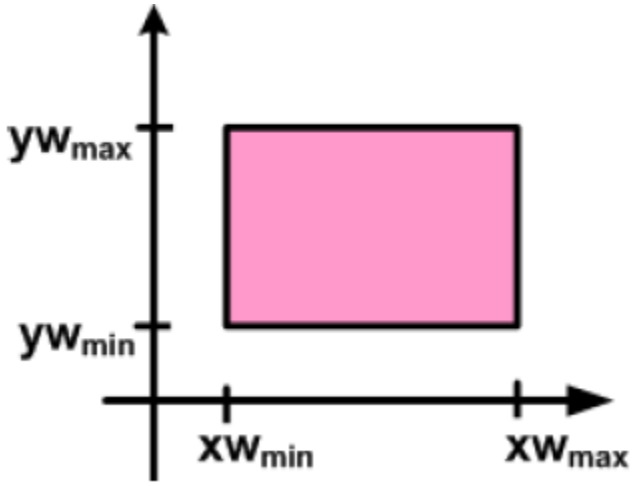


Classwork

- Use the Cohen-Sutherland algorithm to clip the line segment P1, P2
- **P1’’**= (22, 100)
- **P2’**= (124, 10)



Liang-Barsky Line Clipping Algorithm



- For a **line segment** with endpoints (x_1, y_1) and (x_2, y_2), we can describe the line with parametric form:

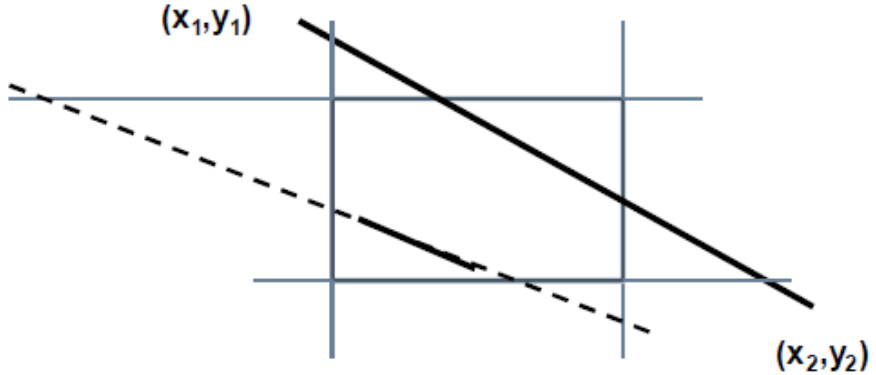
$$x = x_1 + u\,\Delta x$$

$$y = y_1 + u\,\Delta y, \quad 0 \le u \le 1$$

- where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$
- The **point-clipping** condition in the parametric form:

$$xw_{min} \le x_1 + u\,\Delta x \le xw_{max}$$

$$yw_{min} \le y_1 + u\,\Delta y \le yw_{max}$$



Liang-Barsky Line Clipping Algorithm

$$xw_{min} \le x_1 + u\,\Delta x \le xw_{max}$$

$$yw_{min} \le y_1 + u\,\Delta y \le yw_{max}$$

OR

$$u\,\Delta x \ge xw_{min} - x_1 \quad \textbf{(Left boundary)}$$

$$u\,\Delta x \le xw_{max} - x_1 \quad \textbf{(Right boundary)}$$

$$u\,\Delta y \ge yw_{min} - y_1 \quad \textbf{(Bottom boundary)}$$

$$u\,\Delta y \le yw_{max} - y_1 \quad \textbf{(Top boundary)}$$

- Each of these four inequalities can be expressed as:

$$up_k \le q_k,\ k=1, 2, 3, 4$$

Liang-Barsky Line Clipping Algorithm

$$up_k \le q_k,\ k=1, 2, 3, 4$$

$$u\,\Delta x \ge xw_{min} - x_1$$
$$u\,\Delta x \le xw_{max} - x_1$$
$$u\,\Delta y \ge yw_{min} - y_1$$
$$u\,\Delta y \le yw_{max} - y_1$$

- where the parameters **p** and **q** are defined as:

$$p_1 = -\Delta x, \quad q_1 = x_1 - xw_{min} \quad \text{(Left boundary)}$$
$$p_2 = \Delta x, \quad q_2 = xw_{max} - x_1 \quad \text{(Right boundary)}$$
$$p_3 = -\Delta y, \quad q_3 = y_1 - yw_{min} \quad \text{(Bottom boundary)}$$
$$p_4 = \Delta y, \quad q_4 = yw_{max} - y_1 \quad \text{(Top boundary)}$$

Liang-Barsky Line Clipping Algorithm

- If $\mathbf{p_k=0}$, line is **parallel** to window
  - Any lines that is **parallel** to one of the clipping boundaries has $\mathbf{p_k=0}$ for the value of **k** corresponding to that boundary where **k = 1, 2, 3, 4** correspond to **left**, **right**, **bottom** and **top** boundaries respectively
- If $\mathbf{p_k=0}$ and $\mathbf{q_k < 0}$, line is **completely outside** the boundary
  - If, for that value of **k**, we also find $\mathbf{q_k < 0}$, then the line is **completely outside** the boundary and can be eliminated from further consideration
- If $\mathbf{p_k=0}$ $\mathbf{q_k \geq 0}$, then the line is inside the corresponding parallel clipping boundary

Liang-Barsky Line Clipping Algorithm

- $\mathbf{p_k < 0}$
  - The infinite extension of the line proceeds from the **outside to the inside** of the infinite extension of this clipping boundary **(entering)**
- $\mathbf{p_k > 0}$
  - The line proceeds from the **inside to the outside (leaving)**
- For a nonzero value of $\mathbf{p_k}$ ($\mathbf{p_k \neq 0}$), we can calculate the value of **u** that corresponds to the point where the infinitely extended line intersects the extension of boundary **k**

Liang-Barsky Line Clipping Algorithm

Condition	Position of line
$p_k = 0$	Parallel to the clipping boundaries.
$p_k = 0$ & $q_k < 0$	Completely outside the boundary.
$p_k = 0$ & $q_k \geq 0$	Inside the parallel clipping boundary.
$p_k < 0$	Line proceeds from outside to inside.
$p_k > 0$	Line proceeds from inside to outside.

Liang-Barsky Line Clipping Algorithm

- The value of $\mathbf{u_1}$ is determined by looking at the rectangle edges for which the line proceeds from **outside to the inside** ($\mathbf{p_k < 0}$)
  - For these edges, we calculate $\mathbf{r_k = qk / pk}$.
  - The value of $\mathbf{u_1}$ is taken as the **largest** of the set consisting of **0** and the various values of **r**.
  - **p_k < 0, u1= maximum(0, rk) is taken**

Liang-Barsky Line Clipping Algorithm

- The value of $\mathbf{u_2}$ is determined by examining the boundaries for which the line proceeds from **inside to the outside** ($\mathbf{p_k > 0}$).
  - A value of $\mathbf{r_k}$ is calculated for each of these boundaries, and
  - the value of $\mathbf{u_2}$ is the **minimum** of the set consisting of **1** and the calculated **r** values
  - **$p_k > 0$, u2=minimum(1, rk) is taken**

Liang-Barsky Line Clipping Algorithm

- If $\mathbf{u_1 > u_2}$, the line is **completely outside** the clip window and it can be discarded
- Else if $\mathbf{u_2 > u_1}$ the **endpoints of the clipped line** are calculated from the two values of parameter **u**.
  - **If $u_2 < 1$** $\quad x = x_1 + u_2 \Delta x, \quad y = y_1 + u_2 \Delta y$
  - **Otherwise** $\quad x = x_2, \quad y = y_2$
  - **If $u_1 > 0$** $\quad x = x_1 + u_1 \Delta x, \quad y = y_1 + u_1 \Delta y$
  - **Otherwise** $\quad x = x_1, \quad y = y_1$

Liang-Barsky Line Clipping Algorithm

- The Liang-Barsky Line Clipping Algorithm can be summarized as follows:

1. For each line, calculate the parameters **p**, **q** and **r**:

$$r_k = qk / pk.$$

$$p_1 = -\Delta x,\ q_1 = x_1 - xw_{min},\ r_1 = q_1 / p_1$$

$$p_2 = \Delta x,\ q_2 = xw_{max} - x_1\ ,\ r_2 = q_2 / p_2$$

$$p_3 = -\Delta y,\ q_3 = y_1 - yw_{min},\ r_3 = q_3 / p_3$$

$$p_4 = \Delta y,\ q_4 = yw_{max} - y_1,\ r_1 = q_4 / p_4$$

Liang-Barsky Line Clipping Algorithm

2. For each line, calculate values for parameters $\mathbf{u_1}$ and $\mathbf{u_2}$ that define that part of the line that lies within the clip rectangle.

- When,
- $\mathbf{p_k < 0}$**, u1= maximum(0, rk)** is taken.
- $\mathbf{p_k > 0}$**, u2=minimum(1, rk)** is taken.

- 3. **If $\mathbf{u_1 > u_2}$**, the line is **completely outside** the clip window and it can be discarded.
- **4. Otherwise**, the endpoints of the clipped line are calculated from the two values of parameter **u**.

Liang-Barsky Line Clipping Algorithm

The endpoints of the clipped line are calculated from the two values of parameter **u**.

If $u_2 < 1$ $\quad x = x_1 + u_2 \Delta x, \quad y = y_1 + u_2 \Delta y$

Otherwise $\quad x = x_2, \quad y = y_2$

If $u_1 > 0$ $\quad x = x_1 + u_1 \Delta x, \quad y = y_1 + u_1 \Delta y$

Otherwise $\quad x = x_1, \quad y = y_1$

Classwork

Q1. Find the clipping coordinates for a line z_1 z_2 where z_1 =(10,10) and z_2 =(60,30), against window with (x_{wmin} , y_{wmin}) = (15,15), and (x_{wmax}, y_{wmax}) = (25,25).

Classwork

- ***Apply Liang Barsky Line Clipping algorithm to the line with coordinates (30, 60) and(60, 25) against the window*** $(xwmin, ywmin) = (10, 10)$ ***and*** $(xwmax, ywmax) = (50, 50)$.

Q1. Find the clipping coordinates for a line z_1 z_2 where z_1 =(10,10) and z_2 =(60,30), against window with (x_{wmin} , y_{wmin}) = (15,15), and (x_{wmax}, y_{wmax}) = (25,25).

$p_1 = -\Delta x$, $q_1 = x_1 - xw_{min}$, $r_1 = q_1 / p_1$

$p_2 = \Delta x$, $q_2 = xw_{max} - x_1$, $r_2 = q_2 / p_2$

$p_3 = -\Delta y$, $q_3 = y_1 - yw_{min}$, $r_3 = q_3 / p_3$

$p_4 = \Delta y$, $q_4 = yw_{max} - y_1$, $r_1 = q_4 / p_4$

$p_k < 0$, u1= maximum(0, rk)

$p_k > 0$, u2=minimum(1, rk)

If $u_2 < 1$ $x = x_1 + u_2 \Delta x$, $y = y_1 + u_2 \Delta y$

Otherwise $x = x_2$, $y = y_2$

If $u_1 > 0$ $x = x_1 + u_1 \Delta x$, $y = y_1 + u_1 \Delta y$

Otherwise $x = x_1$, $y = y_1$

Q1. Find the clipping coordinates for a line z_1 z_2 where z_1 =(10,10) and z_2 =(60,30), against window with (x_{wmin} , y_{wmin}) = (15,15), and (x_{wmax}, y_{wmax}) = (25,25).

(22.5,15) and (25,16)

Q1. Find the clipping coordinates for a line z_1 z_2 where z_1 =(10,10) and z_2 =(60,30), against window with (x_{wmin}, y_{wmin}) = (15,15), and (x_{wmax}, y_{wmax}) = (25,25).

Solution:

Here, The line coordinates are:

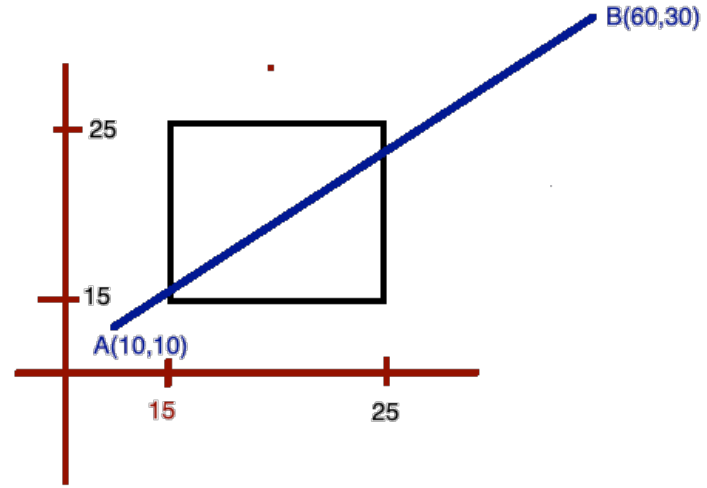
$x_1 = 10$ $\quad$ $y_1 = 10$

$x_2 = 60$ $\quad$ $y_2 = 30$

Now, the window coordinates are:

$x_{wmin} = 15$ $\quad$ $y_{wmin} = 15$

$x_{wmax} = 25$ $\quad$ $y_{wmax} = 25$



1. Calculate the value of parameter p_i and q_i, for i = 1,2,3 &

$p_1 = -\Delta x$ $\quad$ $q_1 = x_1 - x_{wmin}$

$p_1 = -50$ $\quad$ $q_1 = -5$ $\quad$ $\frac{q_1}{p_1} = 0.1$

$p_2 = \Delta x$ $\quad$ $q_2 = x_{wmax} - x_1$

$p_2 = 50$ $\quad$ $q_2 = 15$ $\quad$ $\frac{q_2}{p_2} = 0.3$

$p_3 = -\Delta y$ $\quad$ $q_3 = y_1 - y_{wmin}$

$p_3 = -20$ $\quad$ $q_3 = -5$ $\quad$ $\frac{q_3}{p_3} = 0.25$

$p_4 = \Delta y$ $\quad$ $q_4 = y_{wmax} - y_1$

$p_4 = 20$ $\quad$ $q_4 = 15$ $\quad$ $\frac{q_4}{p_4} = 0.75$

- t_1= max(0, 0.25, 0.1) = 0.25, for the values ($p_i < 0$)
- t_2= min(0.3, 0.75, 1)= 0.3, for the values ($p_i > 0$)

here, $t_1 < t_2$,

Calculate the intersection points:

$xx_1 = x_1 + t_1 * \Delta x$

= 10 + 0.25 * 50

= 22.5

$yy_1 = y_1 + t_1 * \Delta y$

= 10 + 0.25 * 20

= 15

$xx_2 = x_1 + t_2 * \Delta x$

= 10 + 0.3 * 50

= 25

$yy_2 = y_1 + t_2 * \Delta y$

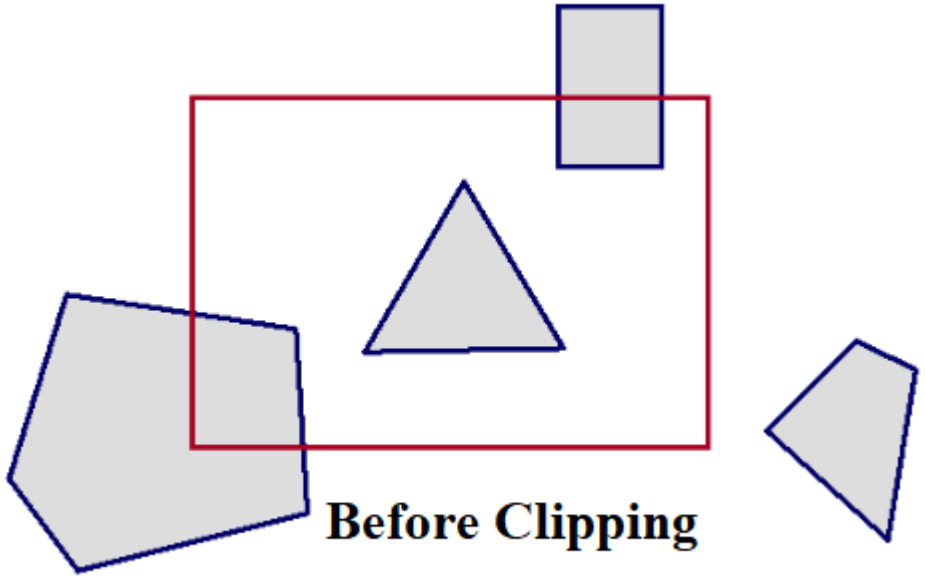
= 10 + 0.3 * 20

= 16

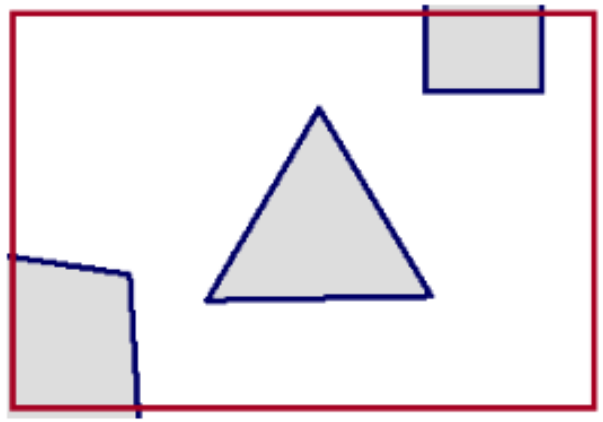
The Clipped Line end points are (22.5,15) & (25, 16)

Polygon Clipping

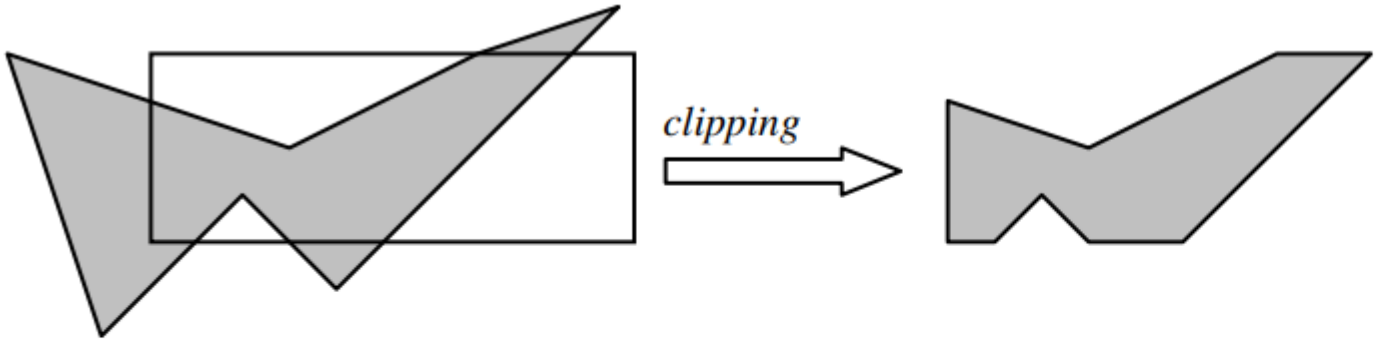
- Polygon clipping is one of those humble tasks computers do all the time. It's a basic operation in creating graphic output of all kinds.
- Polygon clipping is defined by Liang and Barsky (1983) as the process of removing those parts of a polygon that lie outside a clipping window.
- A polygon clipping algorithm receives a polygon and a clipping window as input.
- The algorithm must evaluate each edge of the polygon against each edge of the clipping window, usually a rectangle.
- As a result, new edges may be added, and existing edges may be discarded, retained, or divided

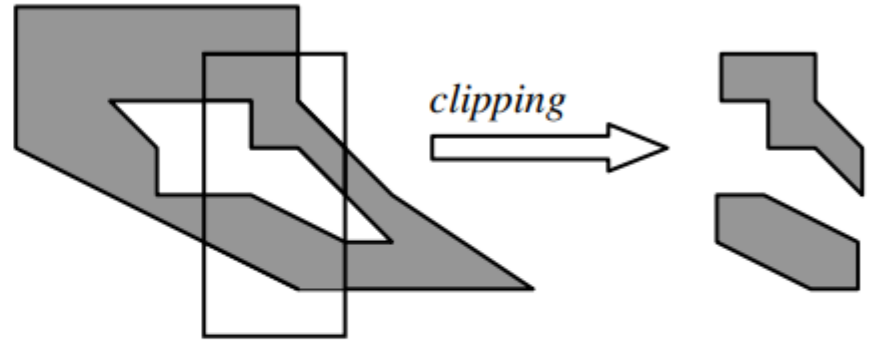


Before Clipping



After Clipping





(a) clipping a polygon that does not have hole;

(b) clipping a polygon that has a hole.

Examples of polygon clipping by a rectangle window:

Weiler-Atherton Polygon Clipping

- When the clipped polygons have two or more separate sections, then it is the concave polygon handled by this algorithm.

- The vertex-processing procedures for window boundaries are modified so that concave polygon is displayed.

- Let the clipping window be initially called Clipping Window (**clip polygon**)and the polygon to be clipped (**subject polygon**).

- We start with an arbitrary vertex of the subject polygon and trace around its border in the clockwise direction until an intersection with the clip polygon is encountered:

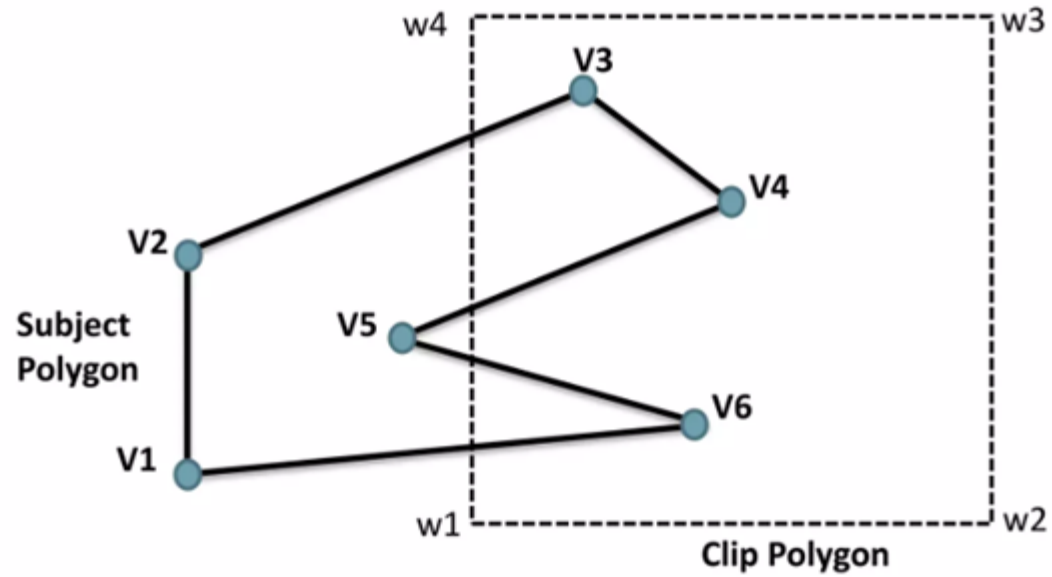
Algorithm

1. Assume the polygon listed in clockwise order.
2. If the edge enter the clip polygon, record the intersection points and continue to trace the subject polygon.
3. If the edge leaves the clip polygon, record the intersection point and make a right follow the clip polygon in same manner. i.e treat clip polygon as subject polygon.

   Continue until vertex reach visited vertex.

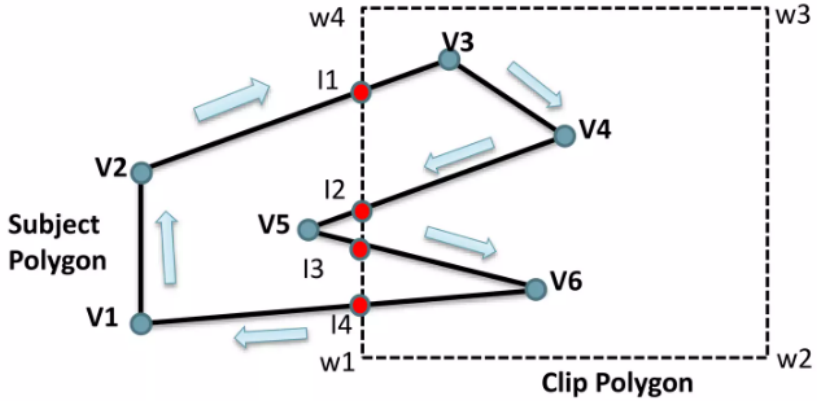
The basis of the clipping logic—the fundamental rule for how it decides what to keep and what to cut—is the direction of intersection (Entering vs. Exiting)

Example: **Clip the Subject Polygon**



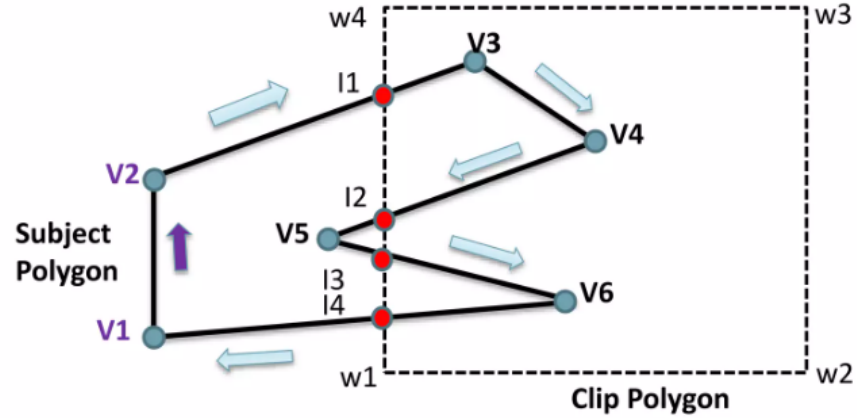
Solution:

1. Clockwise notation in subject polygon.

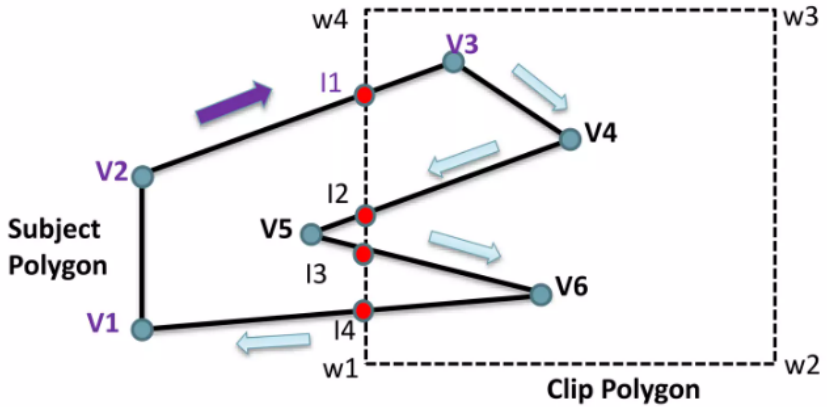


Here, Intersection points are I1, I2, I3 & I4

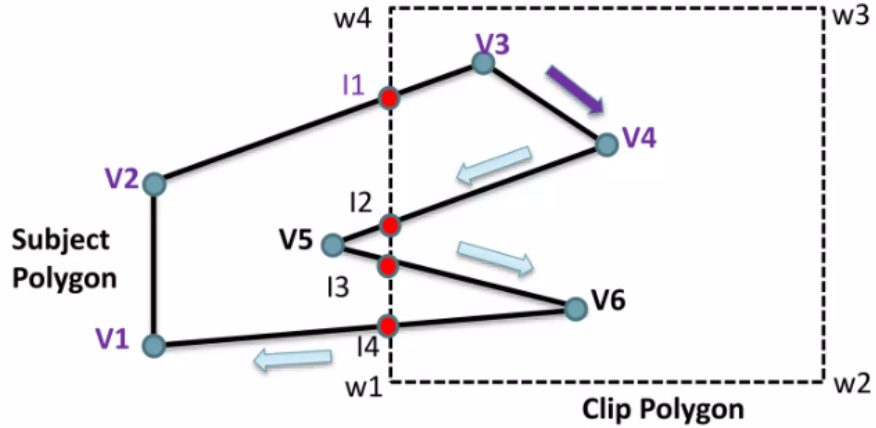
2. Start from V1 vertex to V2 vertex in clockwise direction, both outside vertices then leave.



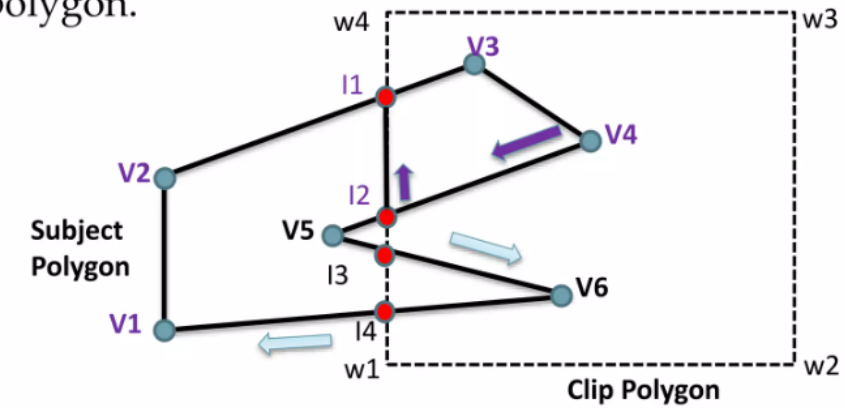
3. From V2 vertex to Vertex V3 in clockwise direction, Here V2 is outside and V3 is inside. so, record Intersection Point I1and continue to subject polygon to V3 Vertex.



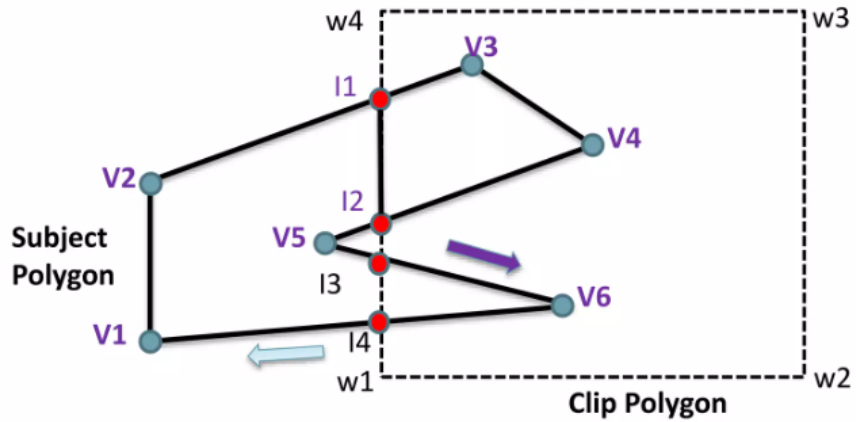
4. From V3 vertex to Vertex V4 in clockwise direction. Here both V2and V3 is inside. so, continue to subject polygon to V4 Vertex.



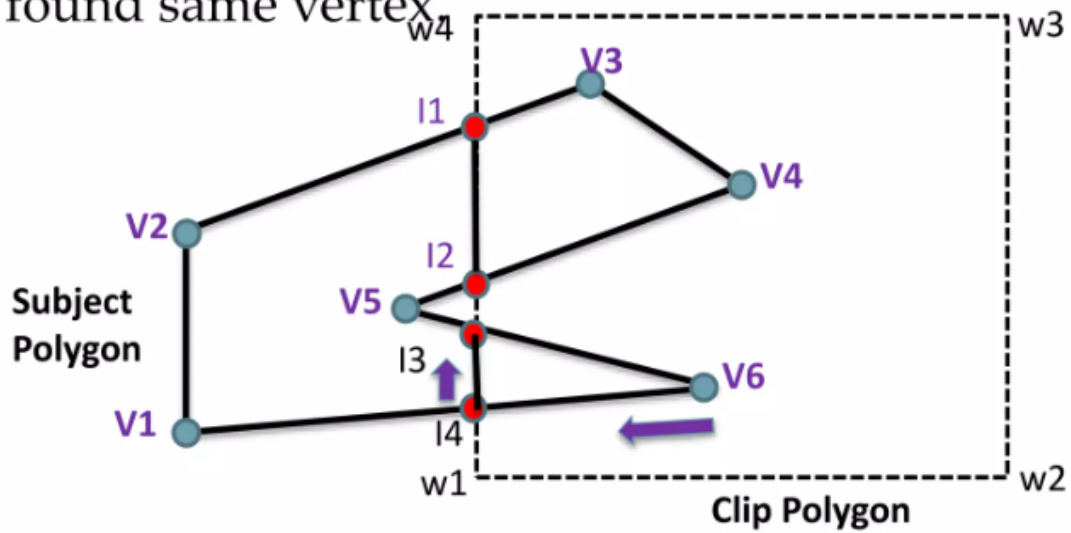
5. From V4 vertex to Vertex V5 in clockwise direction. Here both V4 in inside point and V5 is outside. so, continue to **Clip Polygon** clockwise to intersection point from I2 to I1. Also clip the polygon.



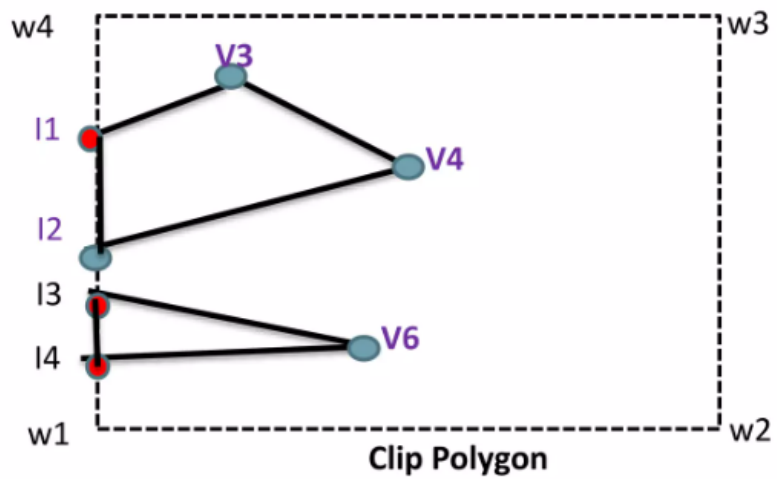
6. From V5 vertex to Vertex V6 in clockwise direction. Here both V5 is outside and V6 is inside. so, continue to subject polygon to V6 Vertex.



7. From V6 vertex to Vertex V1 in clockwise direction. Here both V6 in inside vertex and V1 is outside. so, continue to clip polygon from Intersection point I4 to I3 intersection point. Also clip if found same vertex.



8. Final Clipped part of subject polygon is



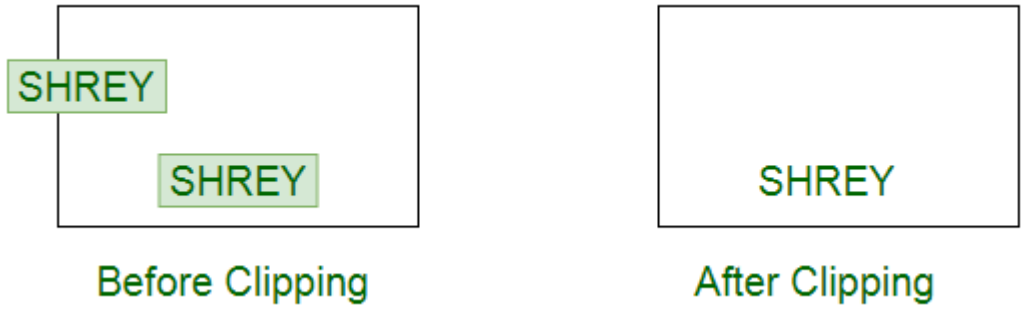
Text Clipping (Assignment)

- **Text Clipping** is a process of clipping the string. In this process, we clip the whole character or only some part of it depending upon the requirement of the application.

1. **All or None String Clipping method**
2. **All or None Character Clipping method**
3. **Text Clipping method**

1.All or None String Clipping method –

- In this method, if the whole string is inside the clip window then we consider it. Otherwise, the string is completely removed.
- Text pattern is considered under a bounding rectangle.
- The boundary positions of the rectangle are then compared to the window boundaries.
- String is rejected if there is any overlapping between the string and the window. This method produces the fastest text clipping.

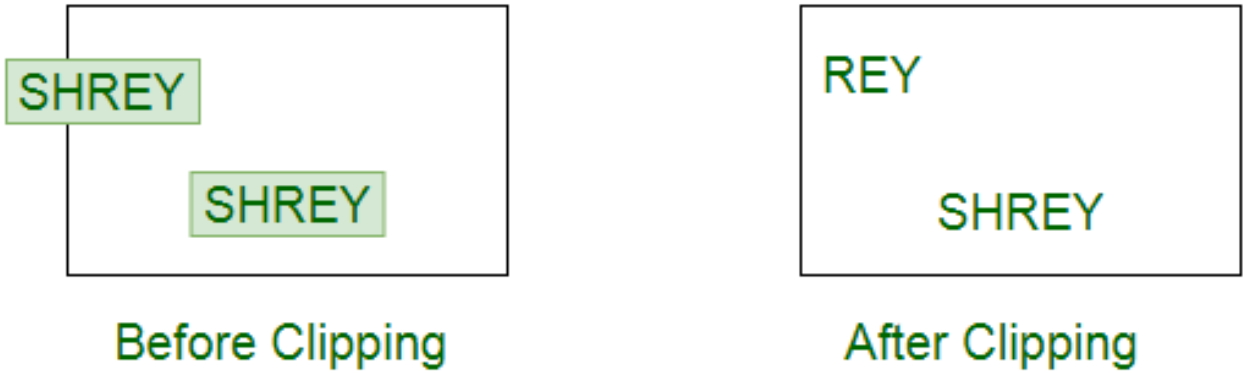


Before Clipping

After Clipping

2. All or None Character Clipping method

- In this method, we keep the characters of the string which lies inside clip window and remove all the characters which lie outside the clip window.
- The boundary limits of individual characters are compared to the window.
- In case of overlapping of character with the clip window, we remove the character.



3. Text Clipping method

- In this method, we keep the characters of the string which lies inside the clip window and remove all the characters which lie outside the clip window.
- If a character overlaps the window boundary then we keep that part of the character which lies inside the window and discard that part which lies outside the clip window.

Before Clipping

After Clipping

Thank you