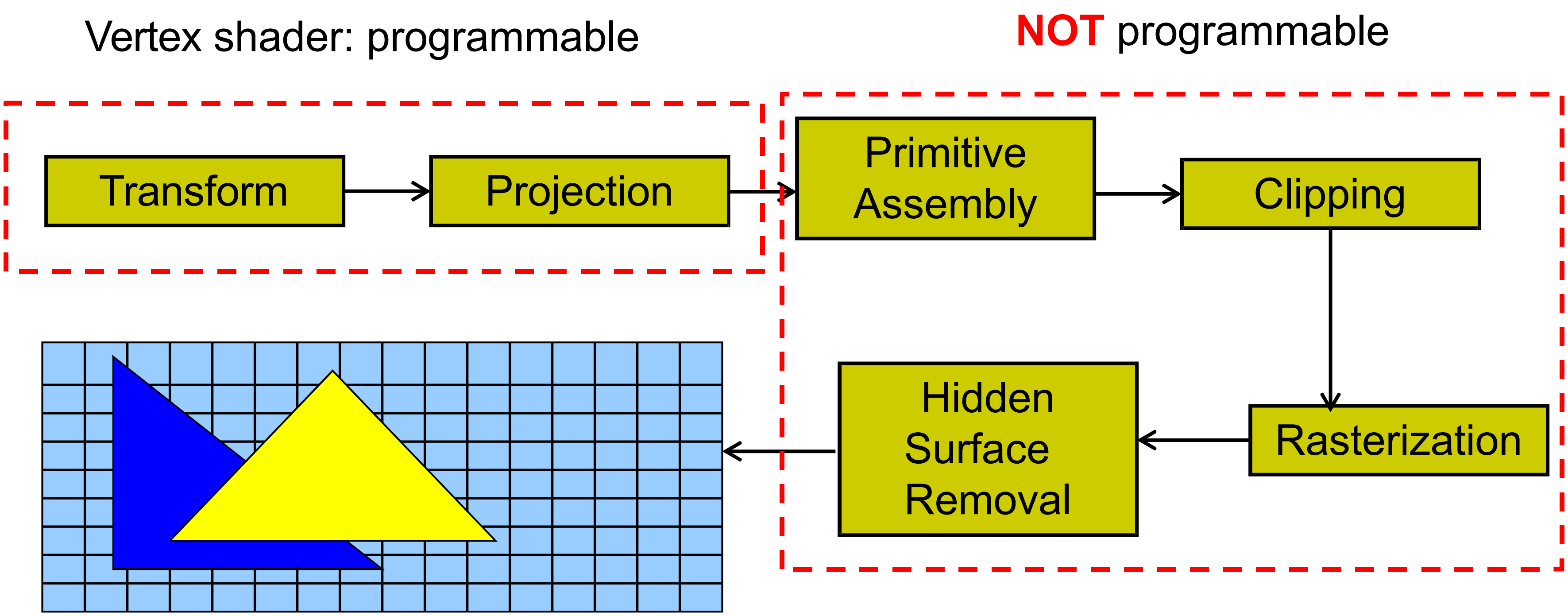
Clipping

Amartya Kundu Durjoy
Lecturer, CSE, UGV

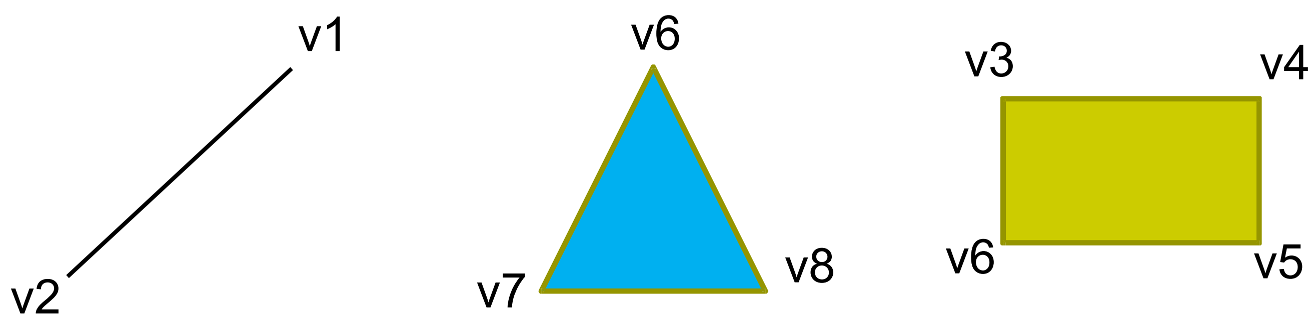
OpenGL Stages

- After projection, several stages before objects drawn to screen
- These stages are non-programmable



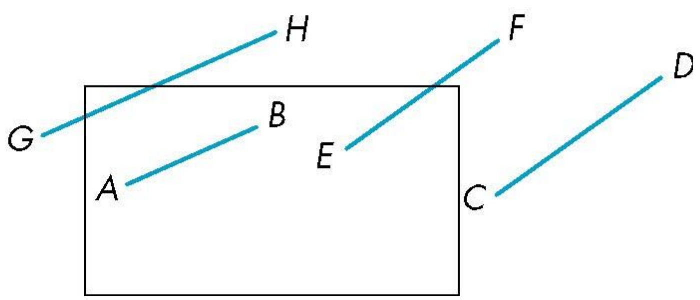
Primitive Assembly

- Up till now: Transformations and projections applied to vertices individually
- **Primitive assembly:** After transforms, projections, individual vertices grouped back into primitives
- **E.g. v6, v7 and v8** grouped back into triangle

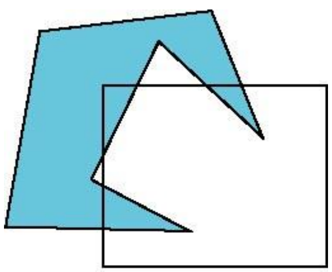


Clipping

- After primitive assembly, subsequent operations are per-primitive
- **Clipping:** Remove primitives (lines, polygons, text, curves) outside view frustum (canonical view volume)



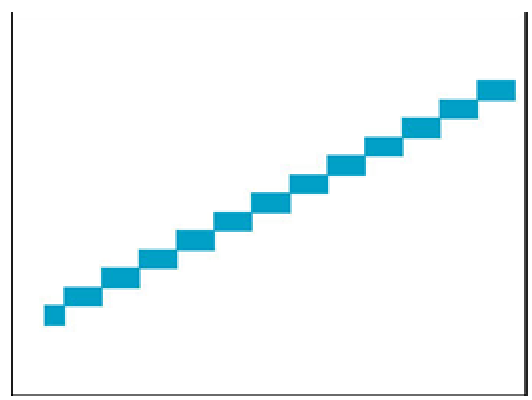
Clipping lines



Clipping polygons

Rasterization

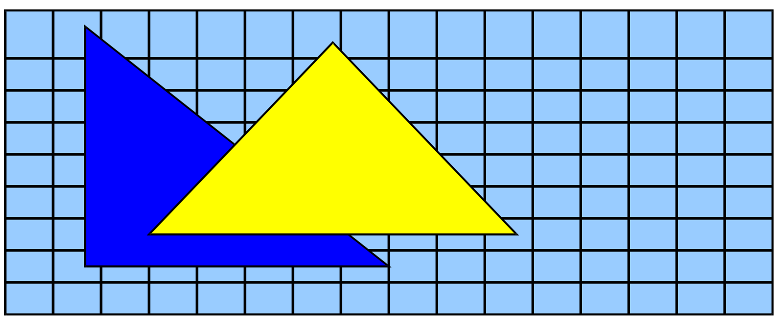
- Determine which pixels that primitives map to
  - Fragment generation
  - Rasterization or scan conversion



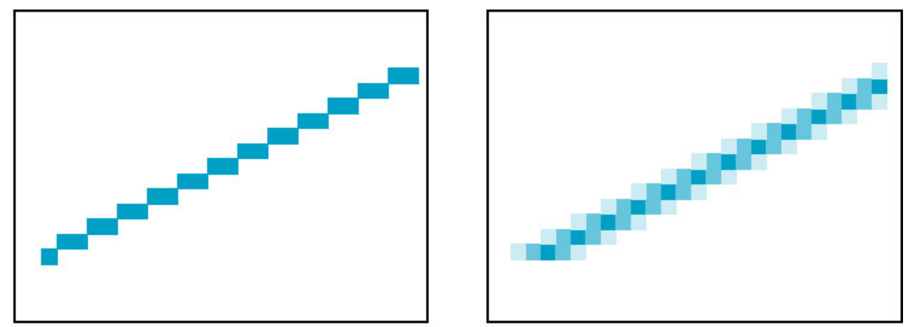
Fragment Processing

- Some tasks deferred until fragment processing

Hidden Surface Removal



Antialiasing



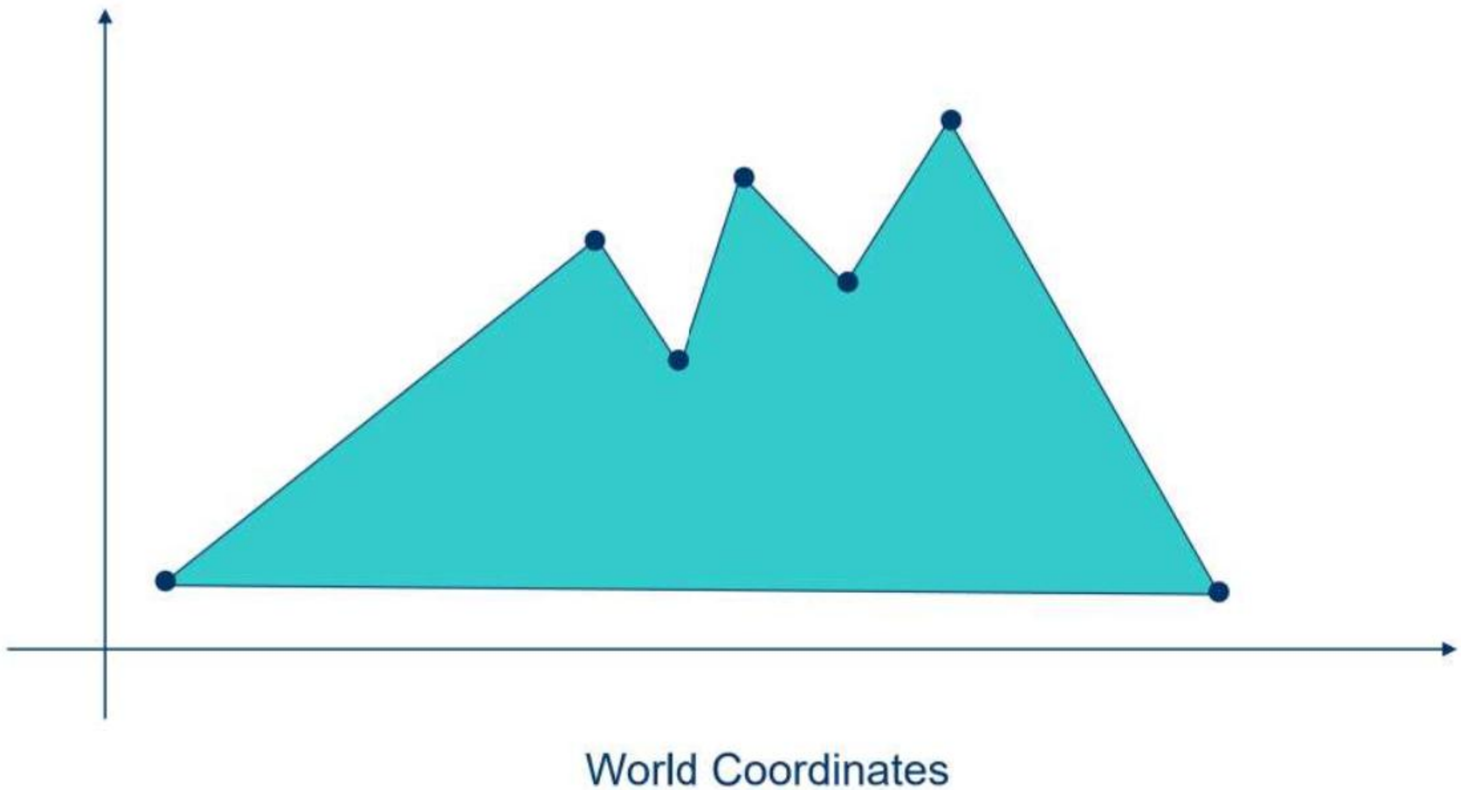
Transformation
Projection

Hidden surface Removal
Antialiasing

Clipping

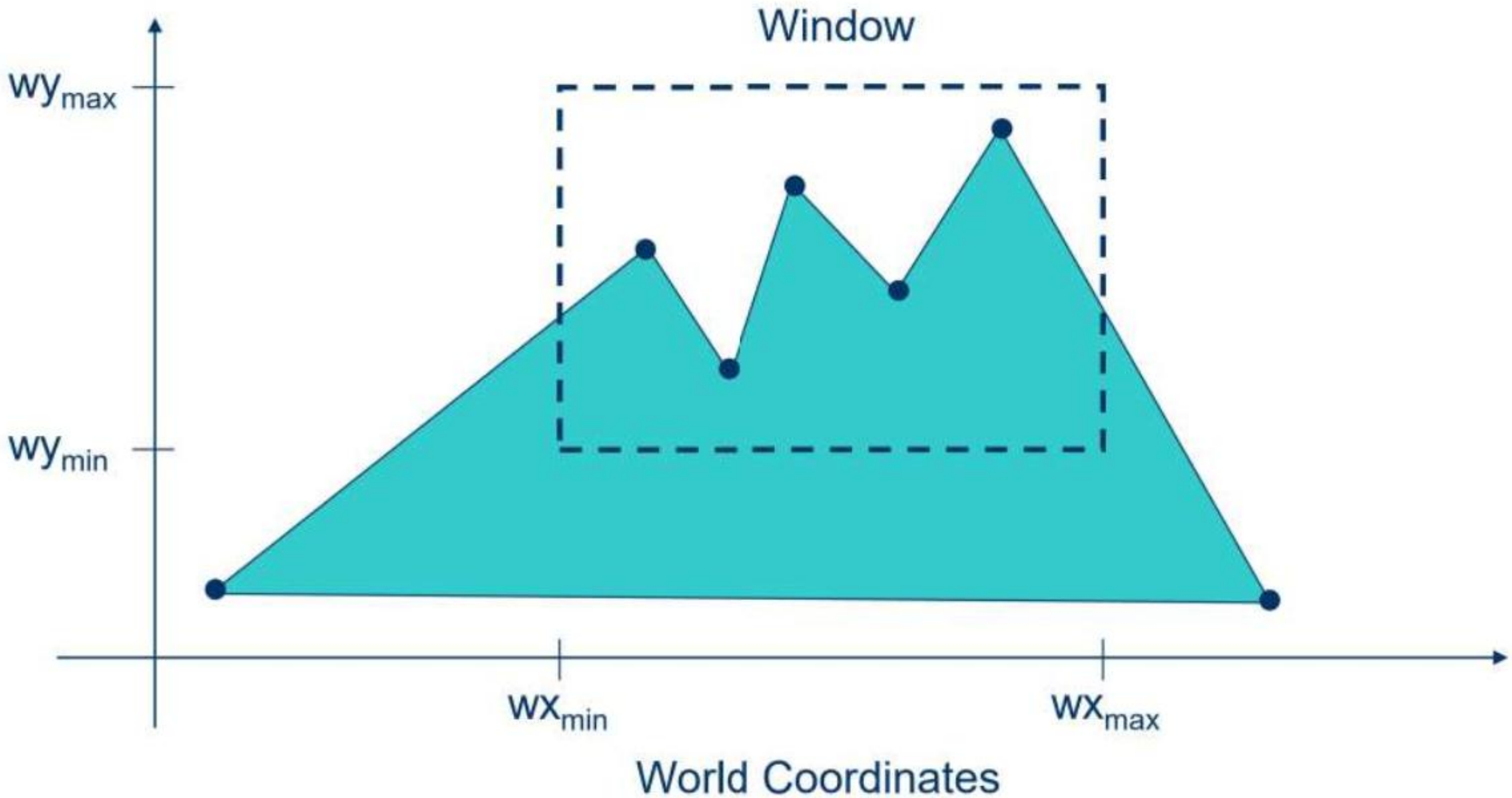
Clipping: Windowing

- A scene is made up of a collection of objects specified in world coordinates



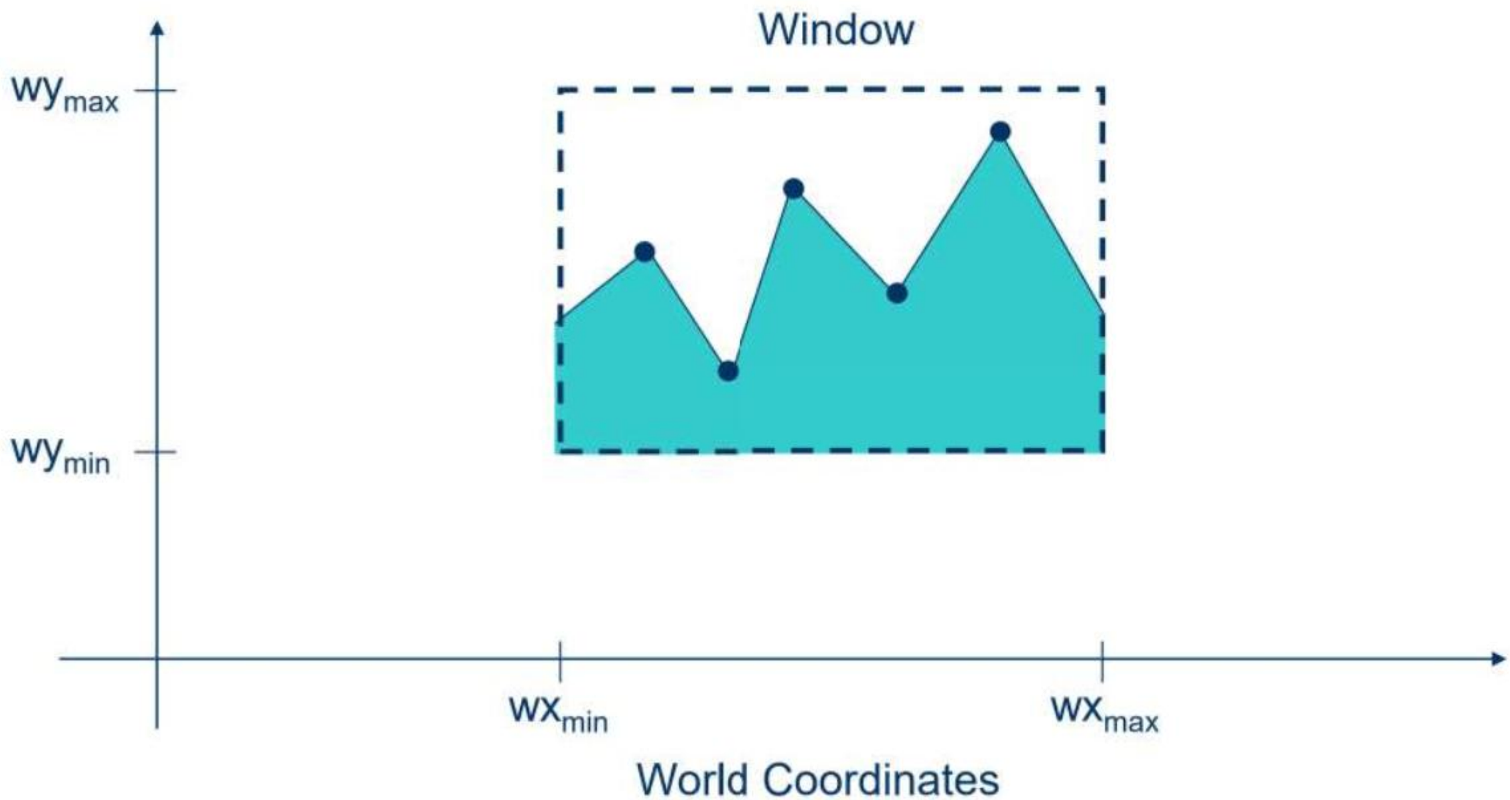
Clipping: Windowing

- When we display a scene only those objects within a particular window are displayed



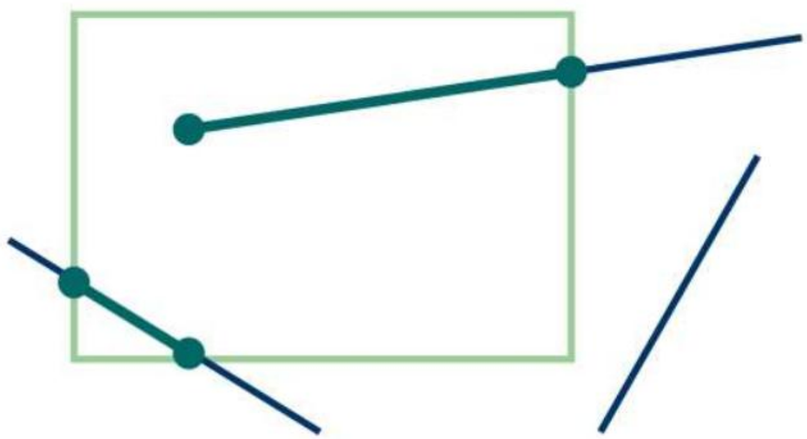
Clipping: Windowing

- Because drawing things to a display takes time we *clip* everything outside the window



Clipping

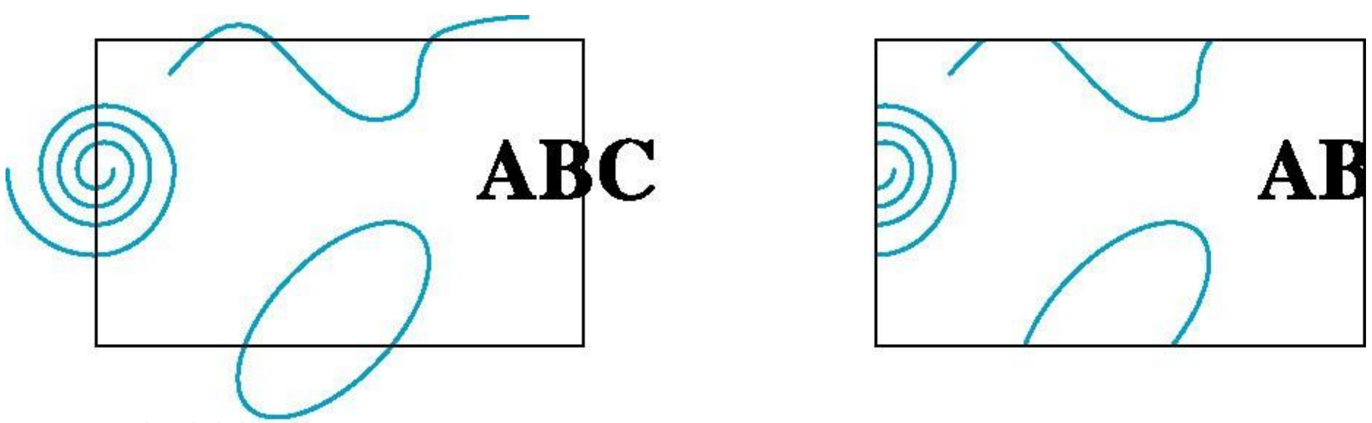
- Analytically calculating the portions of primitives (lines, triangles, polygons) within the clipping window



- Why
  - Bad idea to rasterize outside of framebuffer bounds
  - Also, don't waste time scan converting pixels outside window

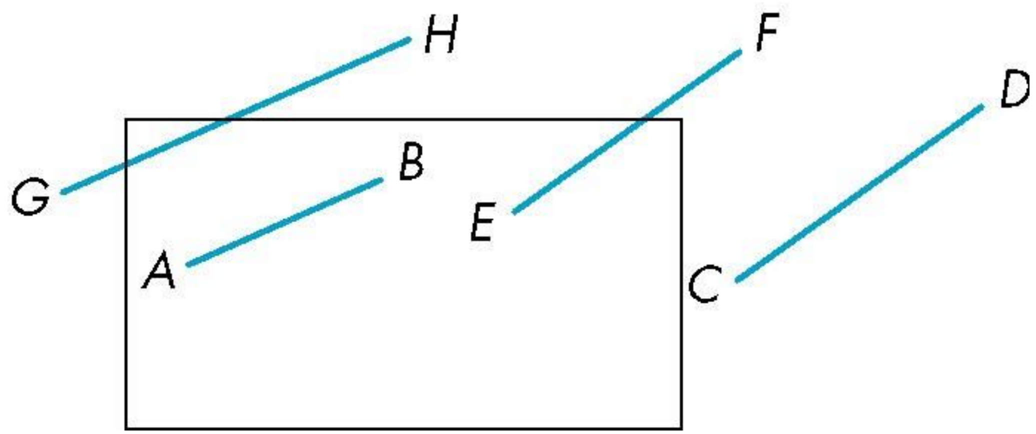
Clipping

- 2D and 3D clipping algorithms
  - 2D against clipping window
  - 3D against clipping volume
- 2D clipping
  - Lines (e.g. dino.dat)
  - Polygons
  - Curves
  - Text



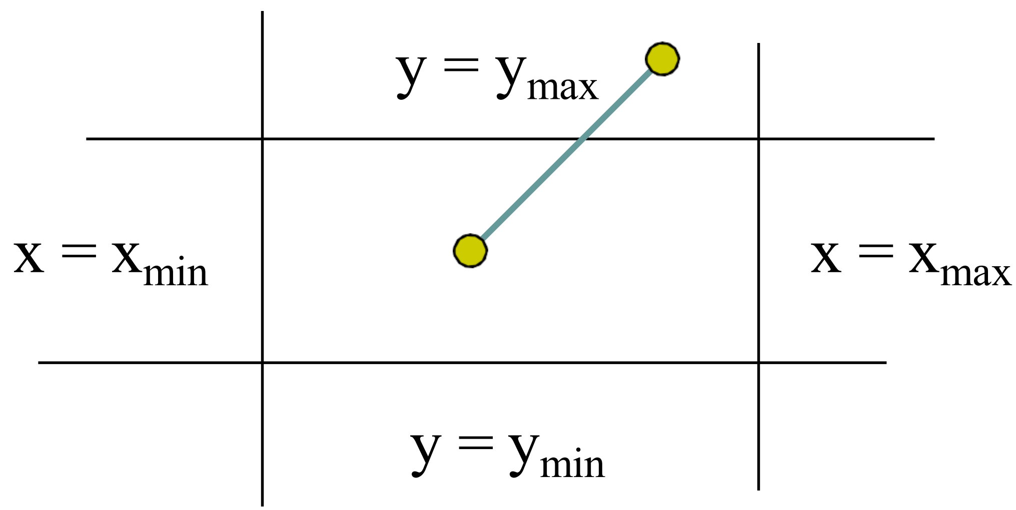
Clipping 2D Line Segments

- **Brute force approach:** compute intersections with all sides of the clipping window
  - Inefficient: one division per intersection

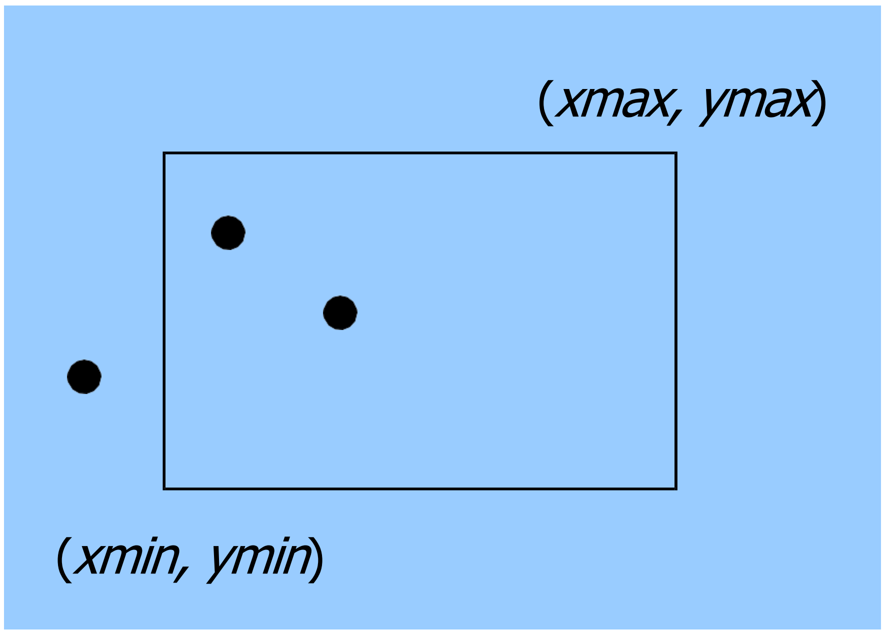


2D Clipping

- **Better Idea:** eliminate as many cases as possible without computing intersections
- **Cohen-Sutherland Clipping algorithm**



Clipping Points

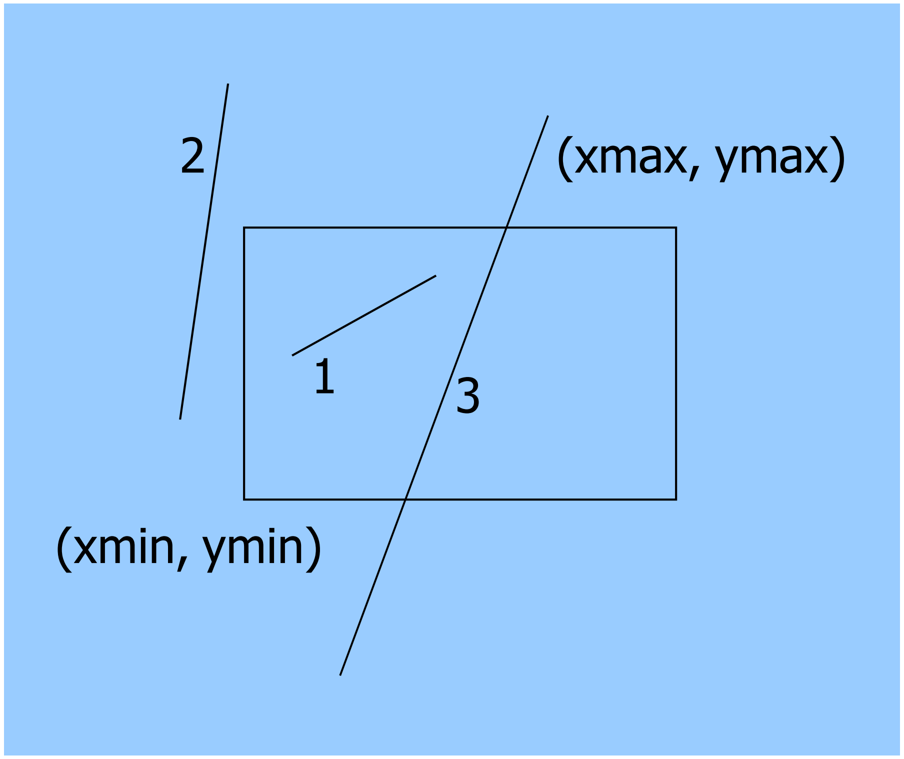


Determine whether a point (x,y) is inside or outside of the world window.

If (xmin <= x <= xmax)
and (ymin <= y <= ymax)

then the point (x,y) is inside else the point is outside

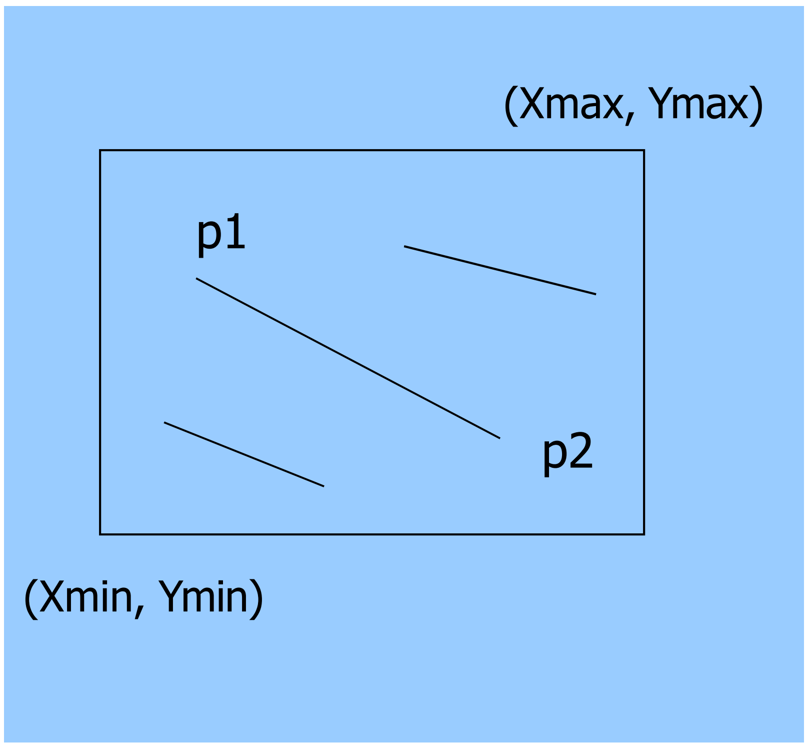
Clipping Lines



3 cases:

- **Case 1:** All of line in
- **Case 2:** All of line out
- **Case 3:** Part in, part out

Clipping Lines: Trivial Accept



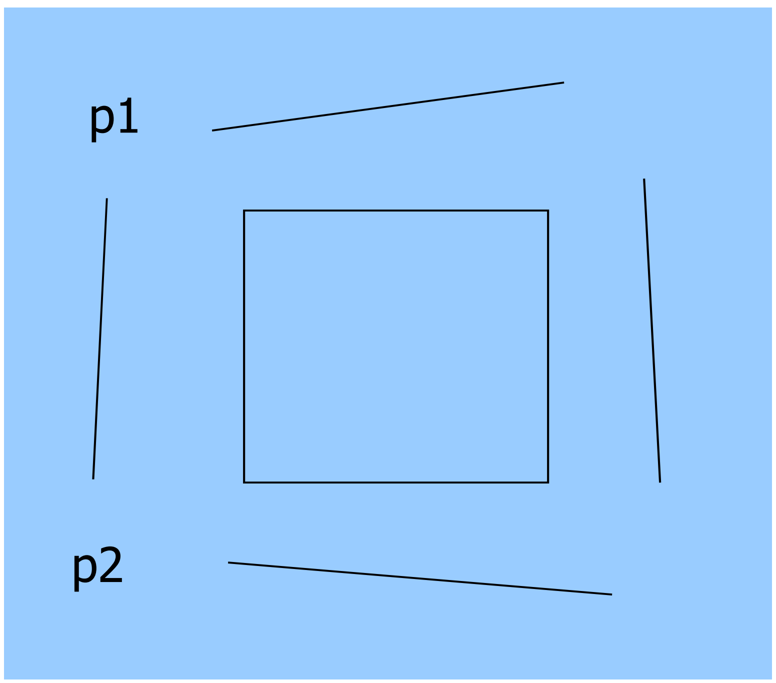
Case 1: All of line in Test line endpoints:

Xmin <= P1.x, P2.x <= Xmax and
Ymin <= P1.y, P2.y <= Ymax

Note: simply comparing the x,y values of endpoints to the x,y values of the rectangle

Result: trivially accepted. Draw the line completely.

Clipping Lines: Trivial Reject



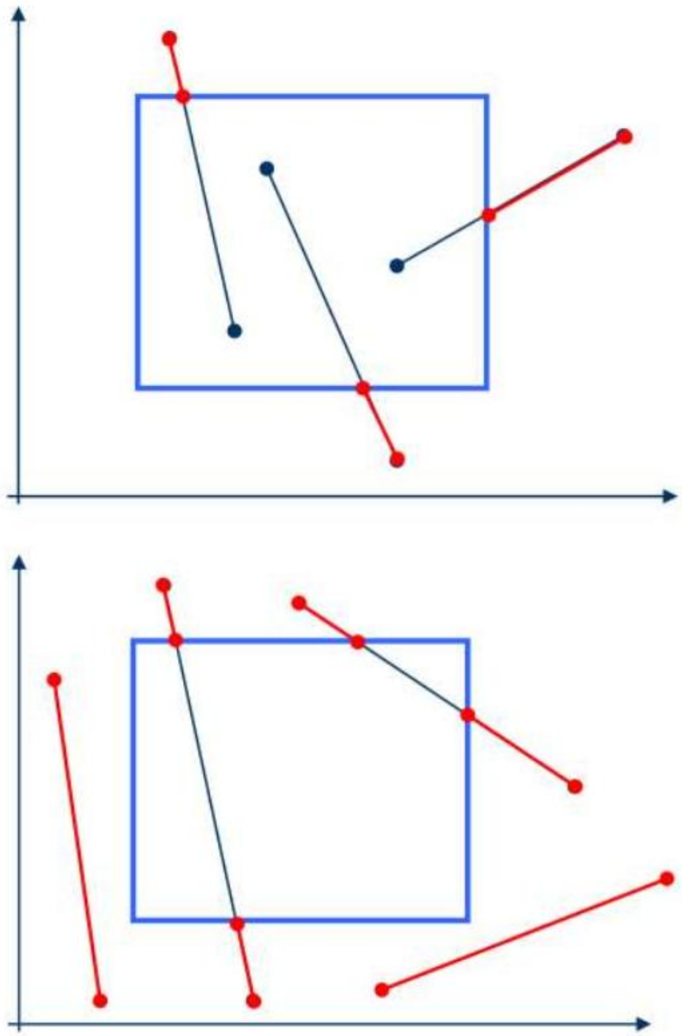
Case 2: All of line out Test line endpoints:

- ***p1.x, p2.x <= Xmin*** **OR**
- ***p1.x, p2.x >= Xmax*** **OR**
- ***p1.y, p2.y <= ymin*** **OR**
- ***p1.y, p2.y >= ymax***

Note: simply comparing the x,y values of endpoints to the x,y values of the rectangle

Result: trivially rejected. Don't draw the line.

Clipping Lines: Non-Trivial Cases



Case 3: Part in, part out

Two variations:

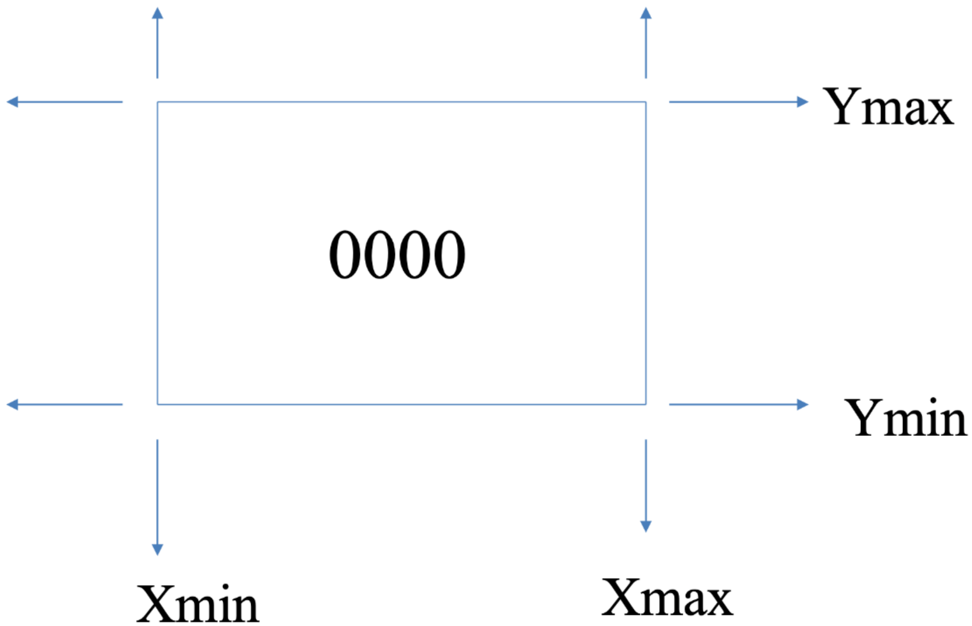
- One point in, other out
- Both points out, but part of the line cuts through the viewport

Cohen-Sutherland Algorithm

- The Cohen-Sutherland Line-Clipping Algorithm performs initial tests on a line to determine whether intersection calculations can be avoided.
  1. First, end-point pairs are checked for Trivial Acceptance.
  2. If the line cannot be trivially accepted, region checks are done for Trivial Rejection.
  3. If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected.
  - These three steps are performed iteratively until what remains can be trivially accepted or rejected.

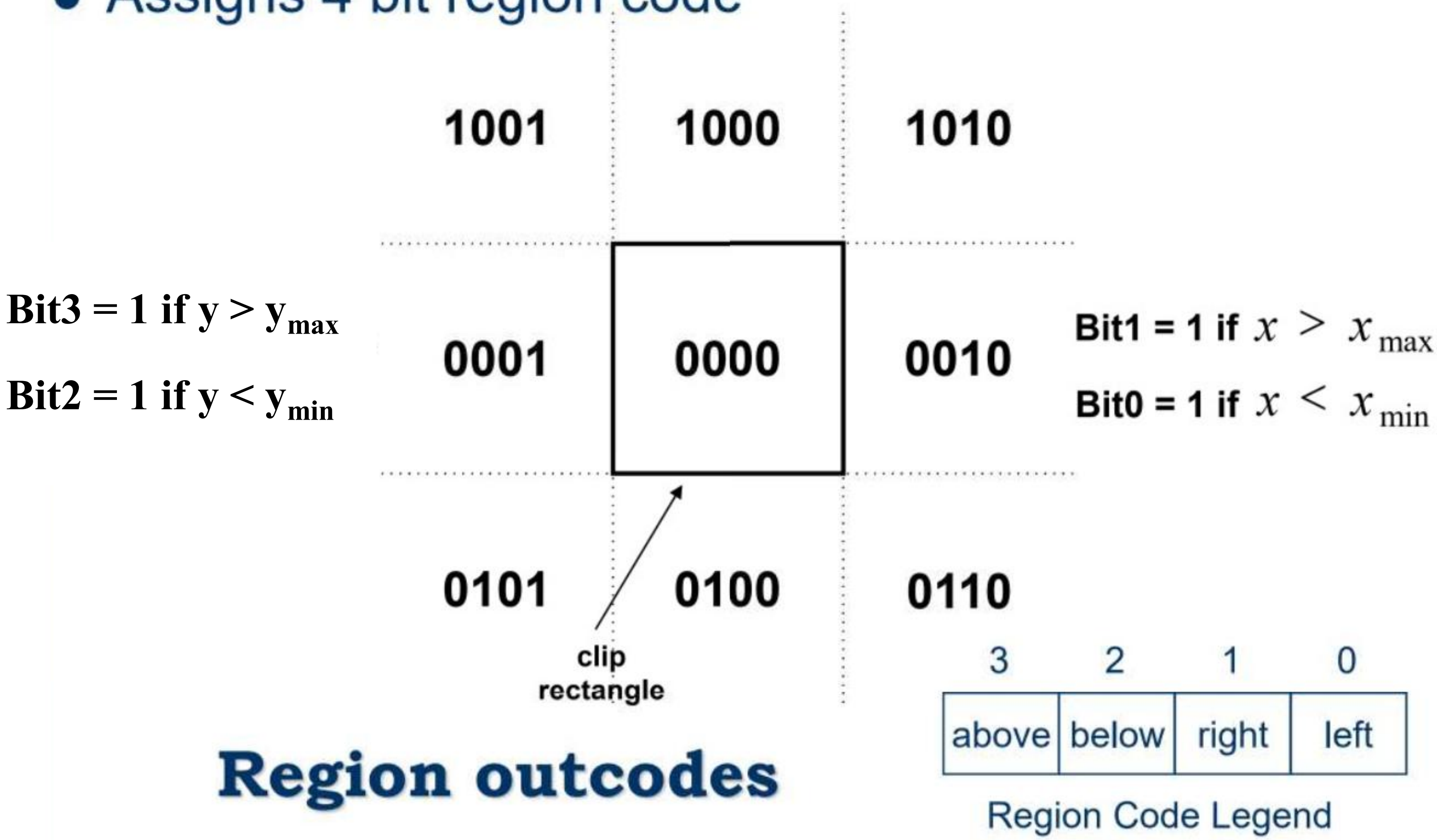
Cohen-Sutherland: World Division

- World Space is divided into regions based on the window boundaries
  - Each region has a unique 4-bit region code
  - Region codes indicate the position of the regions with respect to the window



Cohen-Sutherland Algorithm

- Assigns 4 bit region code



Cohen-Sutherland Algorithm

- A line segment can be trivially accepted (visible) if the outcodes of both the endpoints are zero.
- A line segment can be trivially rejected (not visible) if the logical AND of the outcodes of the endpoints is not zero.
- A line segment is clipping candidate if the logical AND of the outcodes of the endpoints is zero.

Pseudo Code

- ✓ Assign a region code for 2 cut points of a given line
- ✓ If both have region code 0000 then the line is accepted completely
- ✓ Else perform logical AND operation for both region codes
- ✓ If the result is not 0000, the line is outside
- ✓ Else line is partially inside

i. Choose an endpoint of the line that is outside the given rectangle
ii. find intersection point
iii. Replace the endpoint with the intersection point and update the region code
iv. Repeat step 2 until the line is trivially accepted or rejected.

Intersect point

- If bit 3 is 1, intersect with line $y = y_{max}$.
- If bit 2 is 1, intersect with line $y = y_{min}$
- If bit 1 is 1, intersect with line $x = x_{max}$
- If bit 0 is 1, intersect with line $x = x_{min}$

Intersect Point (X_i, Y_i)

$$x_i = x_{\min} \text{ or } x_{\max}$$
$$y_i = y_1 + m(x_i - x_1)$$

If edge line is vertical

or

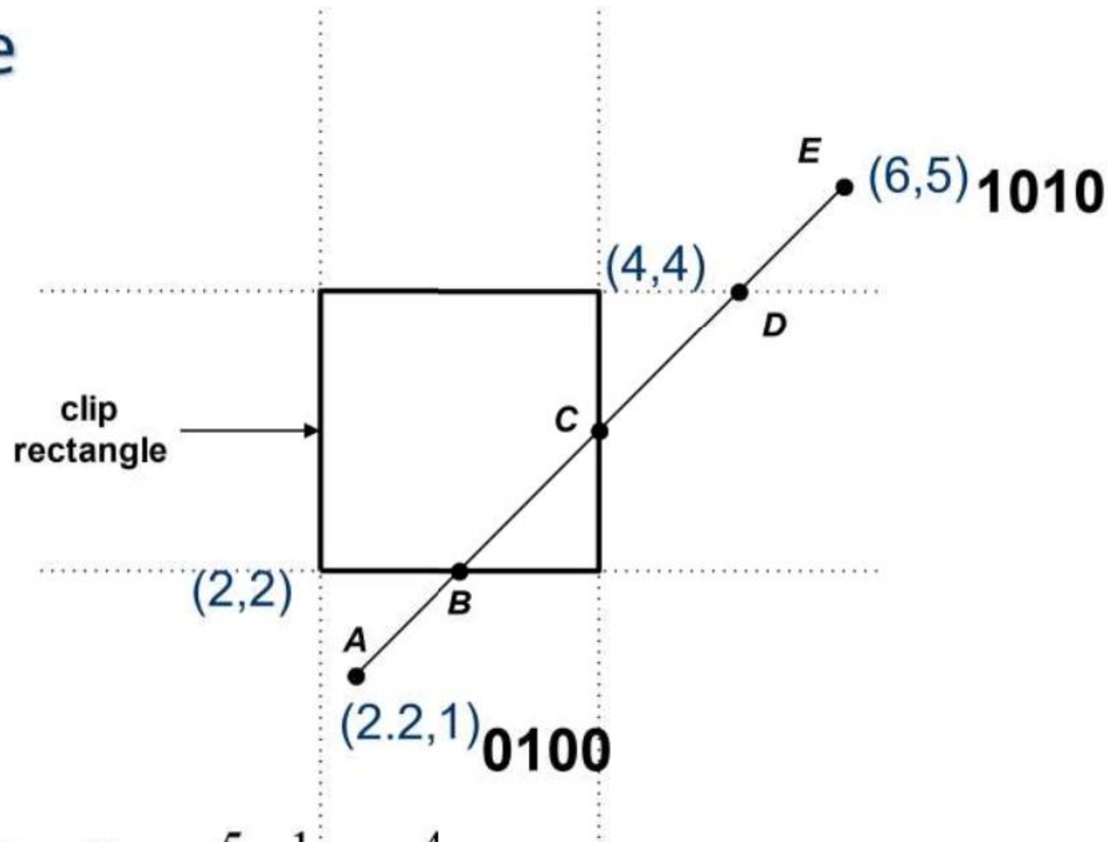
$$x_i = x_1 + (y_i - y_1)/m$$
$$y_i = y_{\min} \text{ or } y_{\max}$$

If edge line is horizontal

$$where, \quad m = (y_2 - y_1)/(x_2 - x_2)$$

Cohen-Sutherland Algorithm

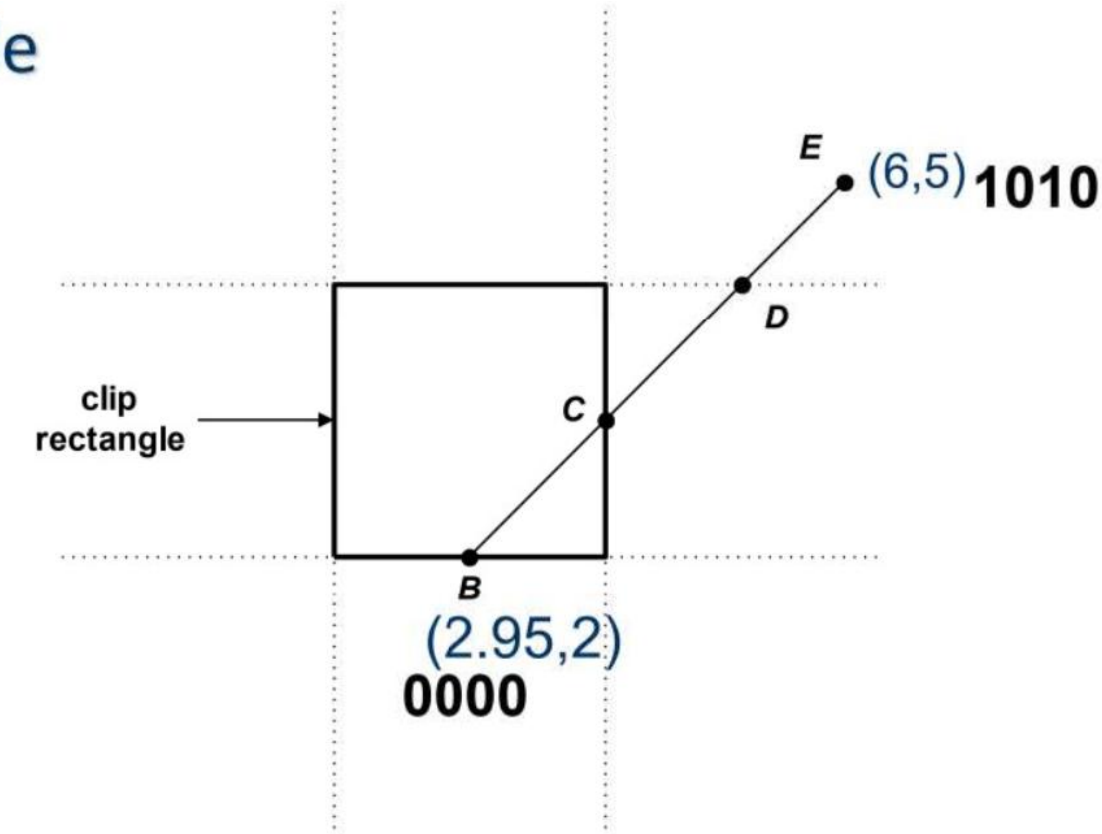
An Example



$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{5-1}{6-2.2} = \frac{4}{3.8} = 1.05$$

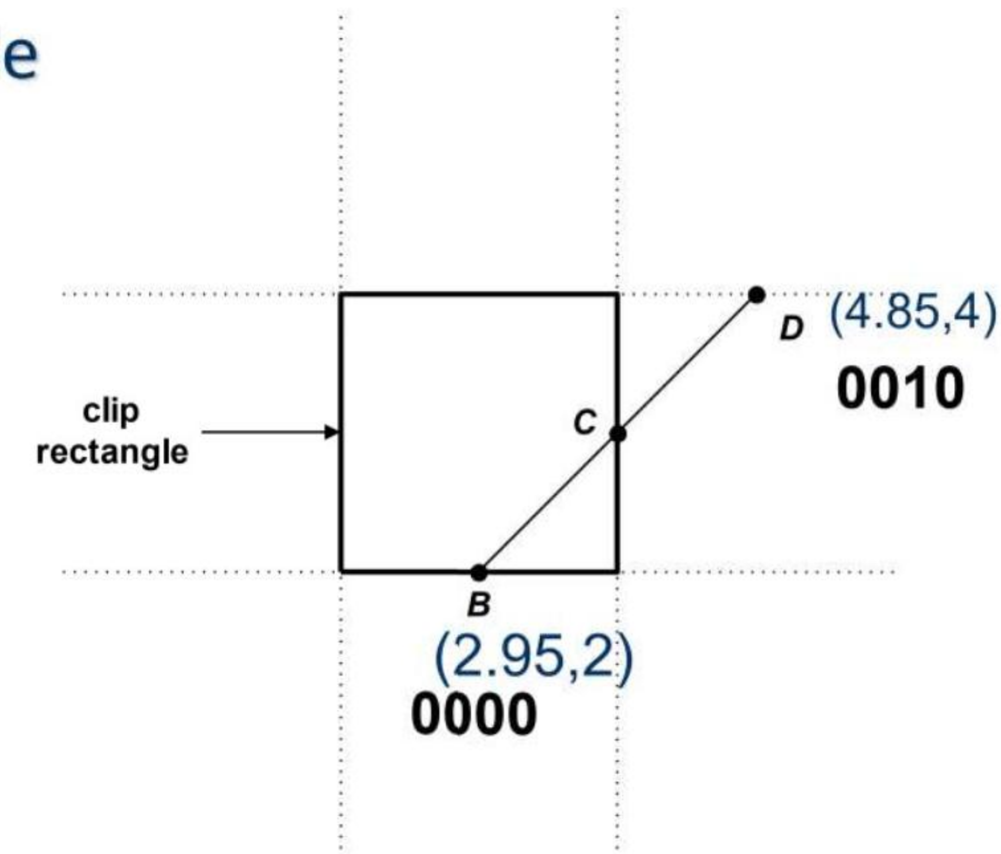
Cohen-Sutherland Algorithm

An Example



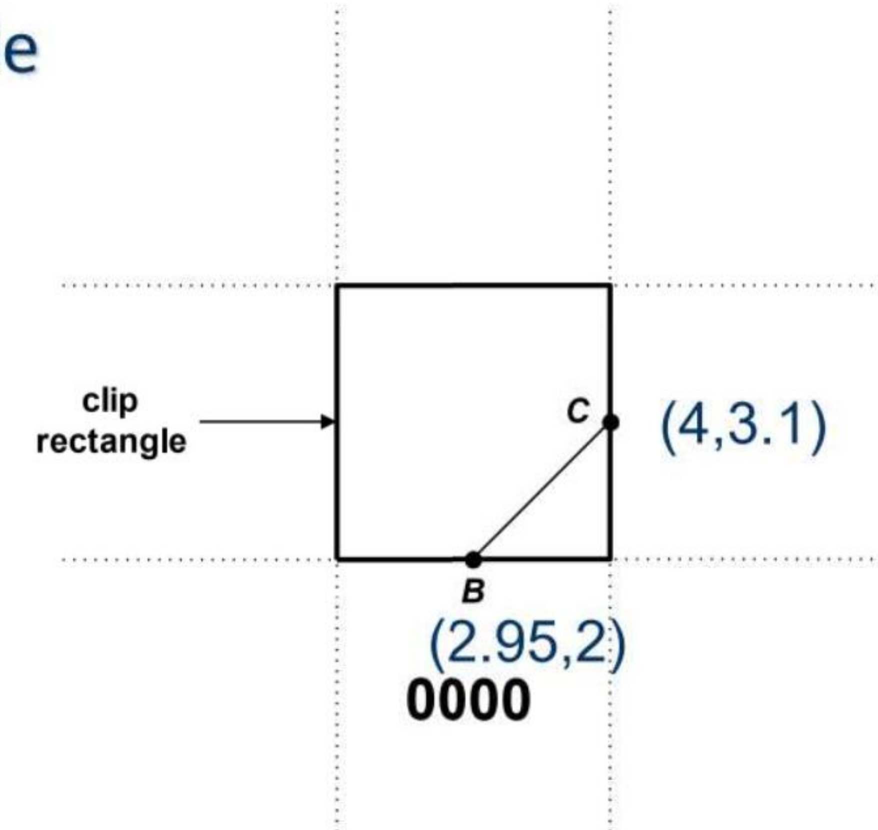
Cohen-Sutherland Algorithm

An Example



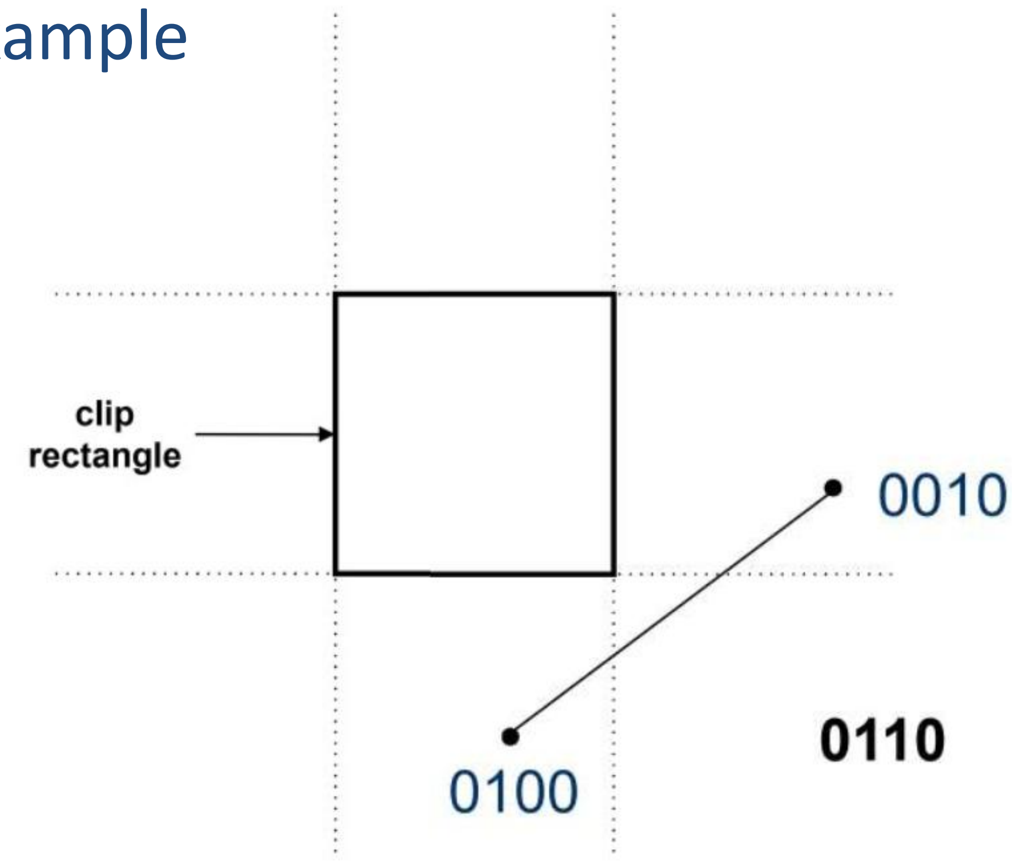
Cohen-Sutherland Algorithm

An Example



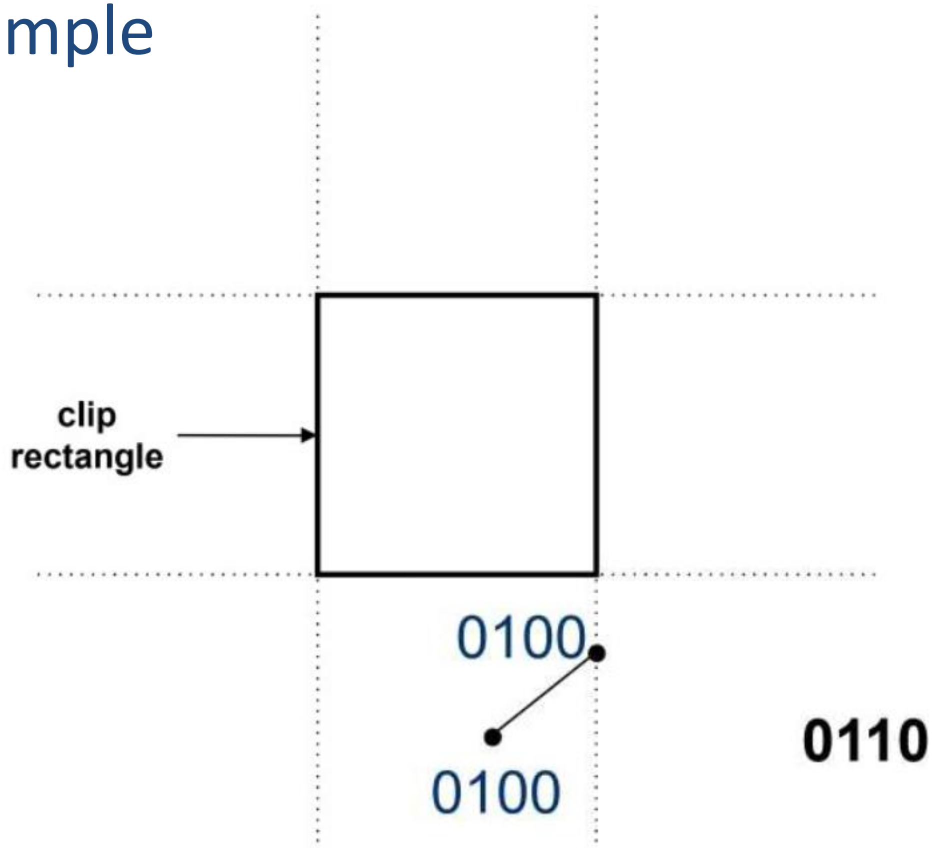
Cohen-Sutherland Algorithm

Another Example



Cohen-Sutherland Algorithm

Another Example



Cohen-Sutherland pseudocode (Hill)

```
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
   do{
        if(trivial accept) return 1; // whole line survives
        if(trivial reject) return 0;  // no portion survives
        // now chop
        if(p1 is outside)
        //  find surviving segment
        {
            if(p1 is to the left) chop against left edge
            else if(p1 is to the right) chop against right edge
            else if(p1 is below) chop against the bottom edge
            else if(p1 is above) chop against the top edge
        }
```

Cohen-Sutherland pseudocode (Hill)

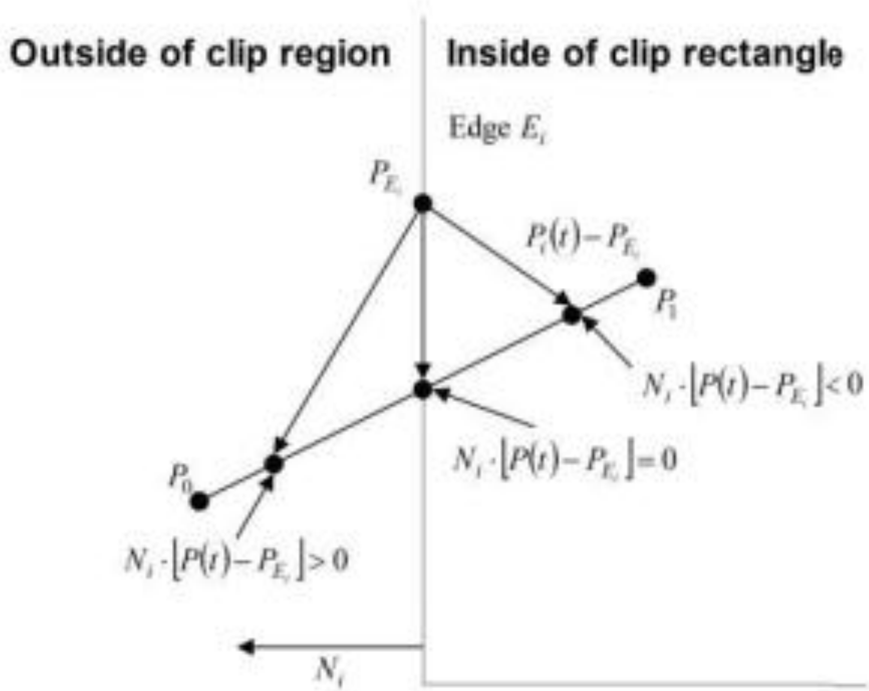
```
        else // p2 is outside
                // find surviving segment
        {
            if(p2 is to the left) chop against left edge
          else if(p2 is to right) chop against right edge
          else if(p2 is below) chop against the bottom edge
          else if(p2 is above) chop against the top edge
        }
    }while(1);
}
```

Parametric Line-Clipping

(1) This fundamentally different (from the Cohen-Sutherland algorithm) and generally more efficient algorithm was originally published by Cyrus and Beck.

(2) Liang and Barsky later independently developed a more efficient algorithm that is especially fast in the special cases of upright 2D and 3D clipping regions. They also introduced more efficient trivial rejection tests for general clip regions.

The Cyrus-Beck Algorithm



$$Line\ P_0P_1 : P(t) = P_0 + (P_1 - P_0)t$$

$$N_i \cdot \left[P(t) - P_{E_i} \right] = 0$$

$$\Rightarrow N_i \cdot \left[P_0 + (P_1 - P_0)t - P_{E_i} \right] = 0$$

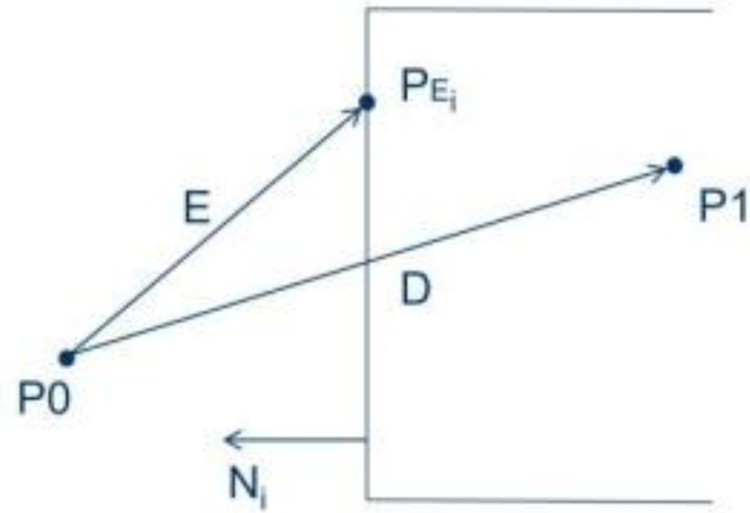
$$\Rightarrow t = -\frac{N_i \cdot \left[P_0 - P_{E_i} \right]}{N_i \cdot (P_1 - P_0)}$$

$$\Rightarrow t = \frac{N_i \cdot \left[P_{E_i} - P_0 \right]}{N_i \cdot D}, \quad D = P_1 - P_0$$

The Cyrus-Beck Algorithm

here

$$N_i \cdot D < 0$$



$$t = \frac{N_i \cdot \left[P_{E_i} - P_0\right]}{N_i \cdot D} = \frac{N_i \cdot E}{N_i \cdot D}$$

t exists when

(1) $N_i \neq 0$

(2) $D \neq 0 \Rightarrow P_0 \neq P_1$

(3) $N_i \cdot D \neq 0$

PE = Potentially Entering

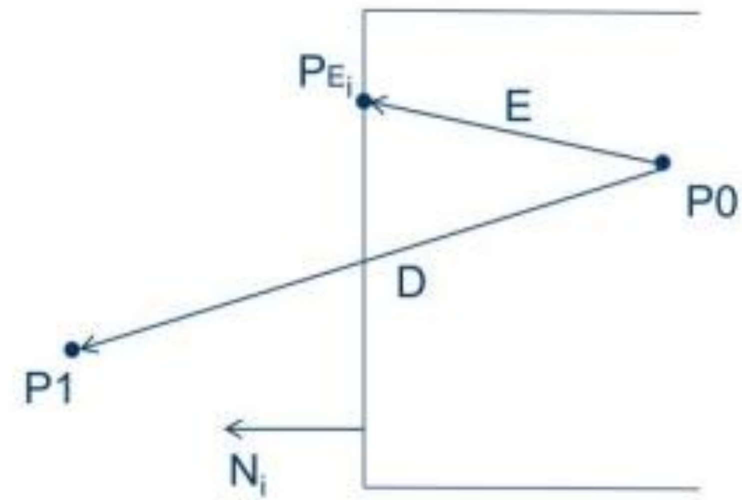
$$N_i \cdot D < 0 \Rightarrow PE$$

$$\Rightarrow Angle > 90°$$

The Cyrus-Beck Algorithm

here

$$N_i \cdot D > 0$$



$$t = \frac{N_i \cdot [P_{E_i} - P_0]}{N_i \cdot D} = \frac{N_i \cdot E}{N_i \cdot D}$$

t exists when

(1) $N_i \neq 0$

(2) $D \neq 0 \Rightarrow P_0 \neq P_1$

(3) $N_i \cdot D \neq 0$

PL = Potentially Leaving

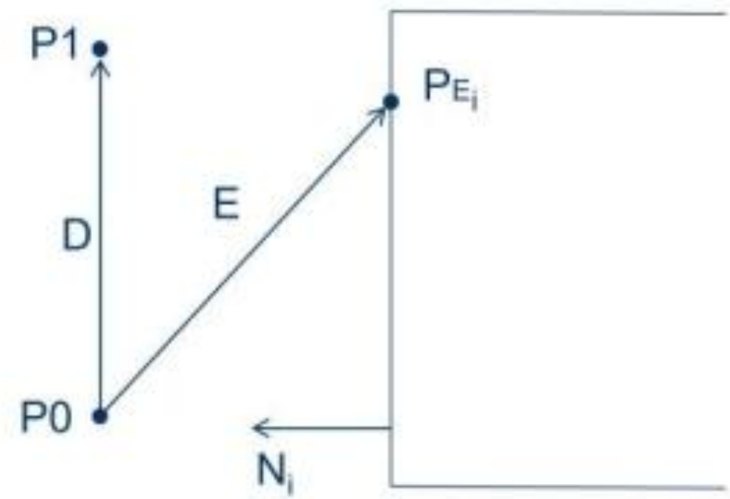
$$N_i \cdot D > 0 \Rightarrow PL$$

$$\Rightarrow Angle < 90°$$

The Cyrus-Beck Algorithm

here

$$N_i \cdot D = 0$$



$$t = \frac{N_i \cdot \left[P_{E_i} - P_0\right]}{N_i \cdot D} = \frac{N_i \cdot E}{N_i \cdot D}$$

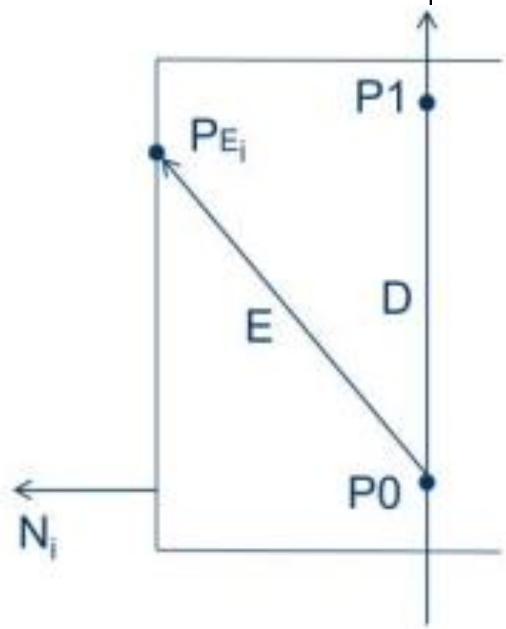
t exists when

(1) $N_i \neq 0$

(2) $D \neq 0 \Rightarrow P_0 \neq P_1$

(3) $N_i \cdot D \neq 0$

The Cyrus-Beck Algorithm



here

$$N_i \cdot D = 0$$

$$t = \frac{N_i \cdot \left[P_{E_i} - P_0\right]}{N_i \cdot D} = \frac{N_i \cdot E}{N_i \cdot D}$$

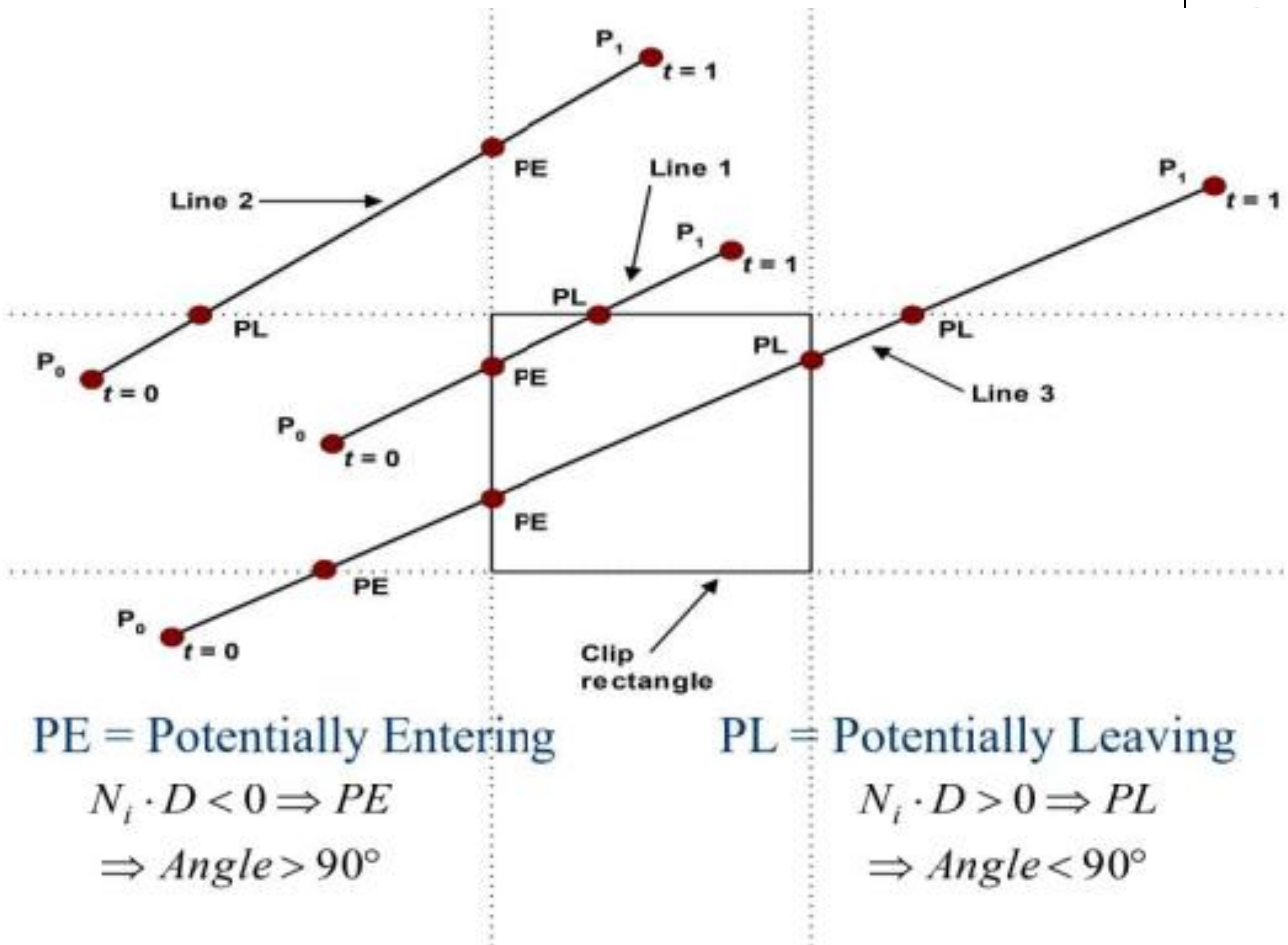
t exists when

(1) $N_i \neq 0$

(2) $D \neq 0 \Rightarrow P_0 \neq P_1$

(3) $N_i \cdot D \neq 0$

The Cyrus-Beck Algorithm



The Cyrus-Beck Algorithm

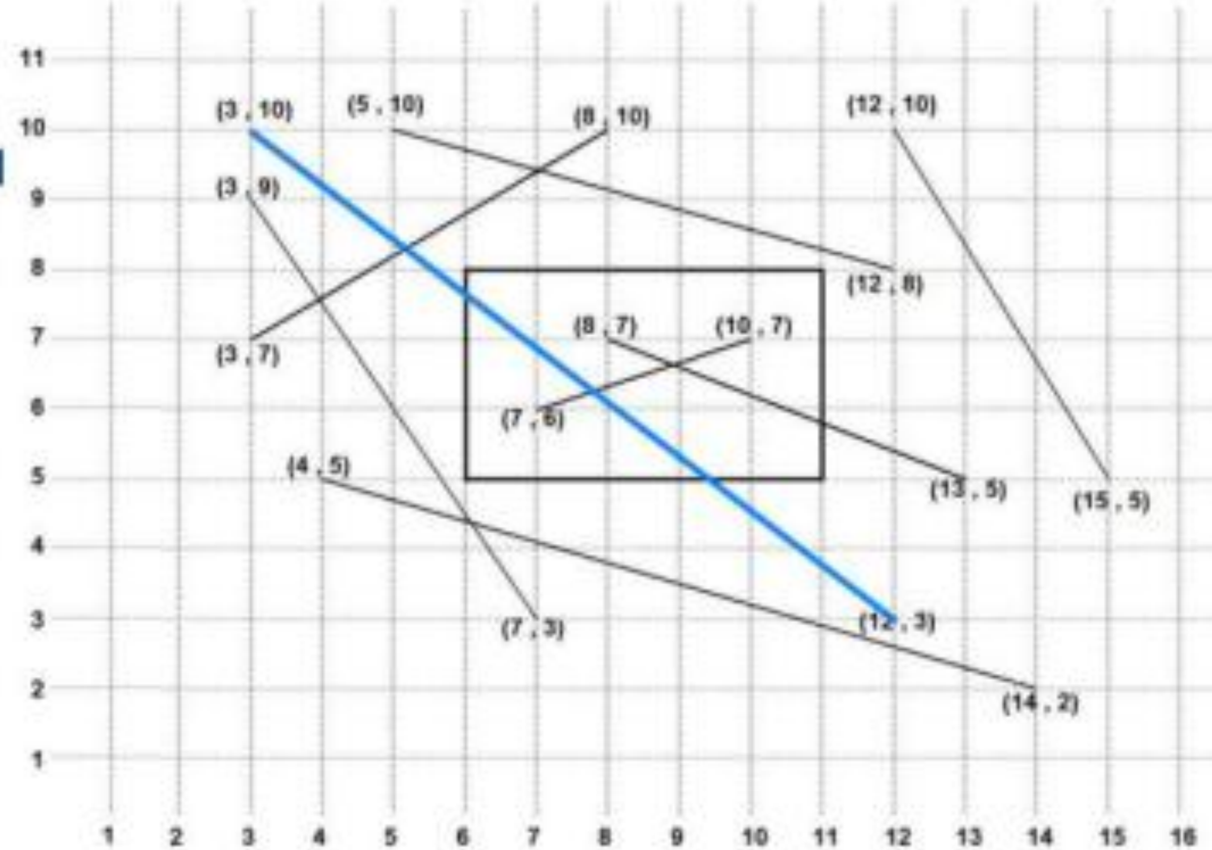
```
Precalculate Ni and PEi for each edge
for (each line segment to be clipped) {
    if (P1 == P0)
        line is degenerated, so clip as a point;
    else {
        tE = 0;  tL = 1;
        for (each candidate intersection with a clip edge) {
            if (Ni • D != 0) {   /* Ignore edges parallel to line */
                calculate t;
                use sign of Ni • D to categorize as PE or PL;
                if (PE) tE = max(tE , t);
                if (PL) tL = min(tL , t);
            }
        }
        if (tE > tL) return NULL;
        else return P(tE) and P(tL) as true clip intersection;
    }
}
```

Liang-Barsky Improvement

Clip edge$_i$	Normal N_i	P_{E_i}	$P_{E_i} - P_0$	$t = \frac{N_i \cdot (P_{E_i} - P_0)}{N_i \cdot D}$	$N_i \cdot D$
left: $x = x_{min}$	$(-1,0)$	(x_{min}, y)	$(x_{min} - x_0, y - y_0)$	$\frac{(x_{min} - x_0)}{(x_1 - x_0)}$	$-(x_1 - x_0)$
right: $x = x_{max}$	$(1,0)$	(x_{max}, y)	$(x_{max} - x_0, y - y_0)$	$\frac{(x_{max} - x_0)}{(x_1 - x_0)}$	$(x_1 - x_0)$
bottom: $y = y_{min}$	$(0,-1)$	(x, y_{min})	$(x - x_0, y_{min} - y_0)$	$\frac{(y_{min} - y_0)}{(y_1 - y_0)}$	$-(y_1 - y_0)$
top: $y = y_{max}$	$(0,1)$	(x, y_{max})	$(x - x_0, y_{max} - y_0)$	$\frac{(y_{max} - y_0)}{(y_1 - y_0)}$	$(y_1 - y_0)$

Example

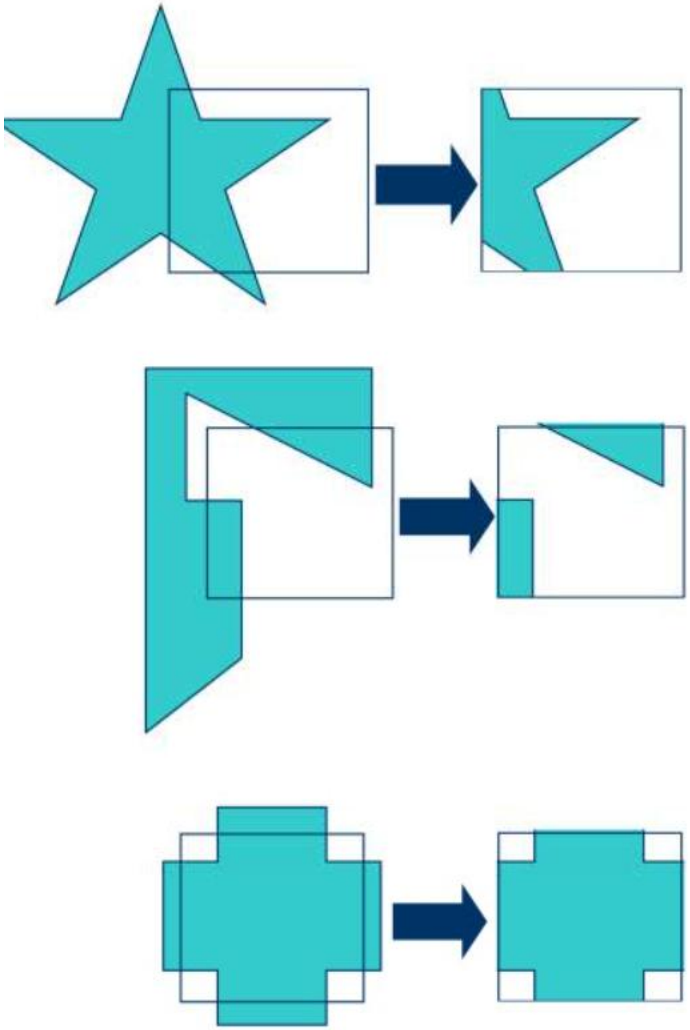
Pmin	6,5		
pmax	11,8		
	L1		
P0	3,10		
P1	12,3		
x1-x0=dx	9		
y1-y0=dy	-7		
tl=(xmin-x0)/dx	3/9 =	0.333	PE
N.D=-dx for left	-9 (-ve)		
tr=(xmax-x0)/dx	8/9 =	0.889	PL
N.D=dx for right	9 (+ve)		
tb=(ymin-y0)/dy	-5/(-7) =	0.714	PL
N.D=-dy for bottom	7 (+ve)		
tt=(ymax-y0)/dy	-2/(-7) =	0.286	PE
N.D=dy for top	-7 (-ve)		
te = max(0, all PE)	0.333		
tl = min(1, all PL)	0.714		



Comparison

- Cohen-Sutherland:
  - Repeated clipping is expensive
  - It is best used when trivial acceptance and rejection are possible for most lines
- Liang-Barsky:
  - Computation of t-intersections is cheap
  - Computation of (x,y) clip points is only done once
  - The algorithm doesn't consider trivial accepts/rejects
  - Best when many lines must be clipped

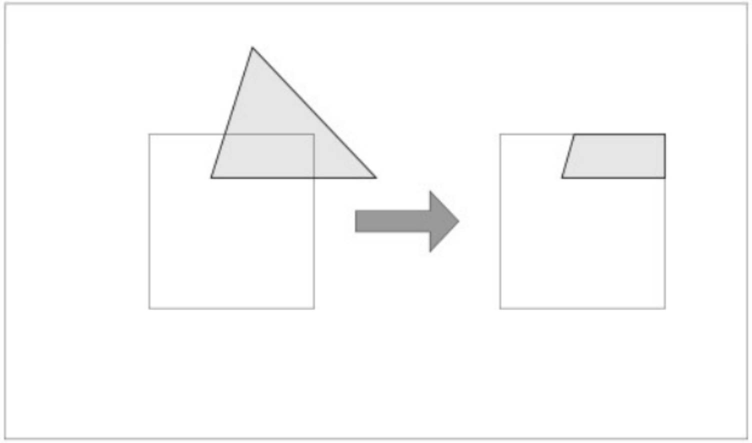
Polygon / Area Clipping



- Similarly to lines, areas must be clipped to a window boundary
- Consideration must be taken as to which portions of the area must be clipped

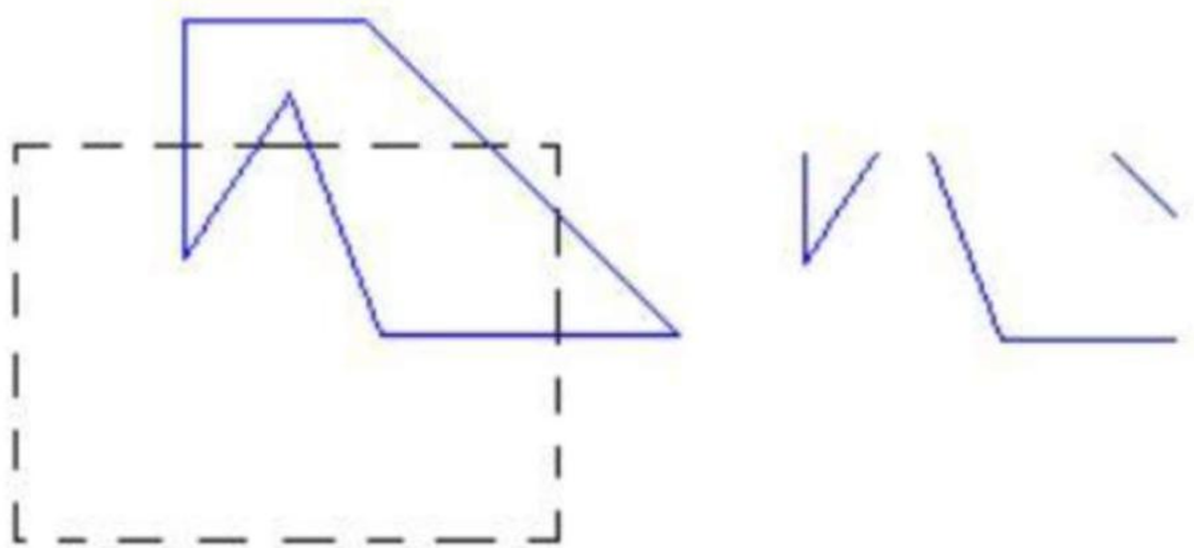
Polygon Clipping

- We know how to clip a single line segment
  - How about a polygon in 2D?
  - How about in 3D?
- Clipping polygons is more complex than clipping the individual lines
  - Input: polygon
  - Output: polygon, or nothing



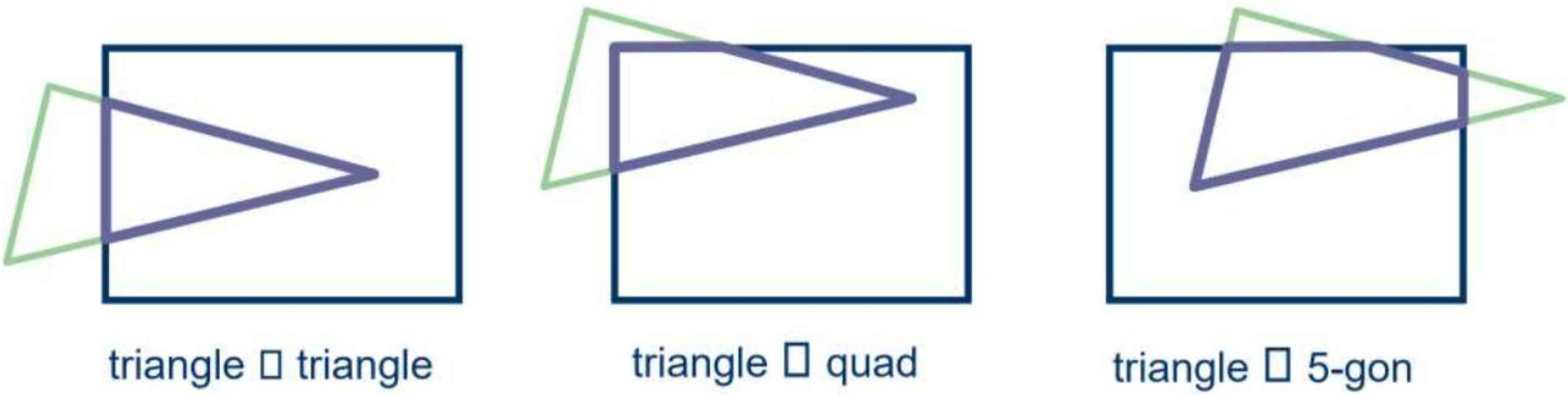
Polygon Clipping

- To clip a polygon, we cannot directly apply a line-clipping method to the individual polygon edges because this approach would produce a series of unconnected line segments as shown in the figure.



Why is Clipping Hard?

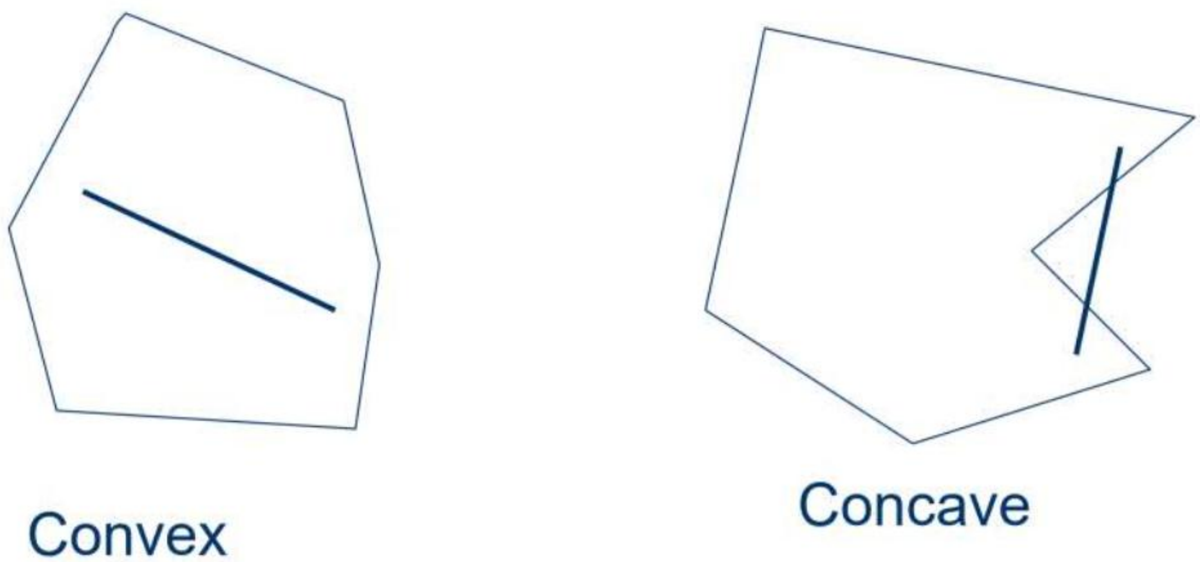
- What happens to a triangle during clipping?
- Possible outcomes:



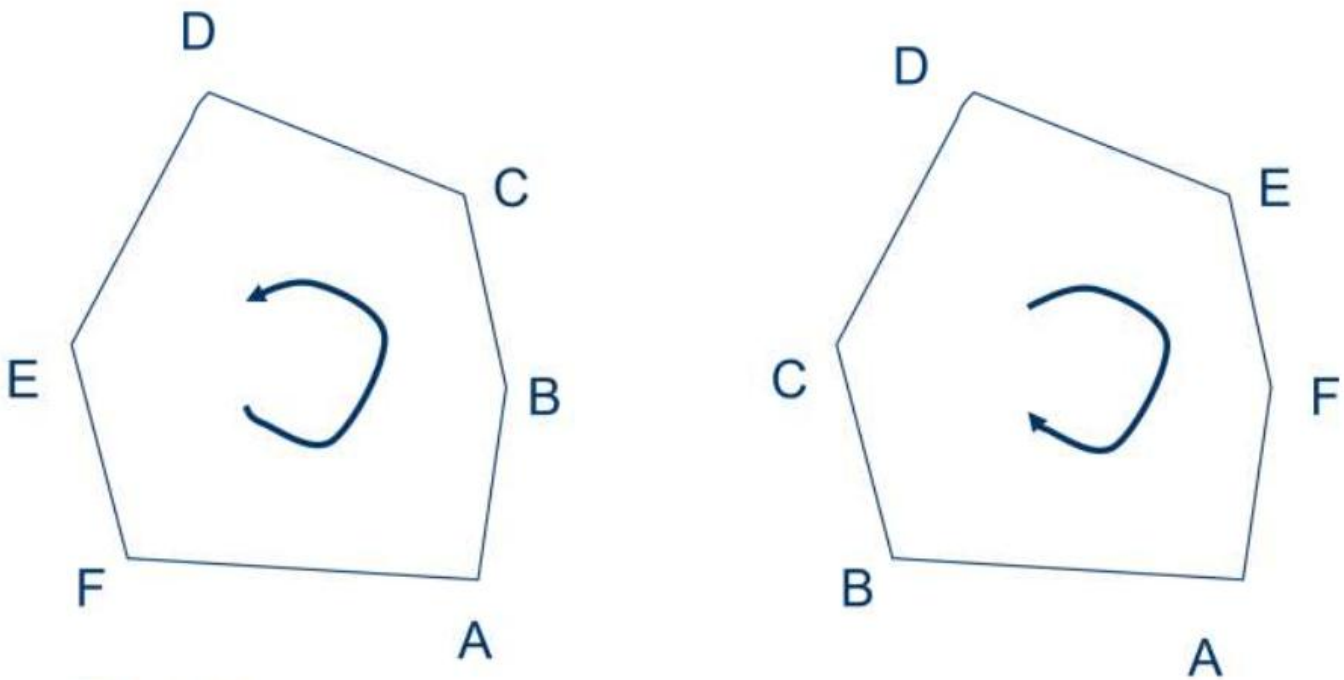
How many sides can a clipped triangle have?

Polygon Clipping

- Convex polygonal clipping window.
  - **Convex polygon**: if the line joining two interior points lies completely inside the polygon.
  - Otherwise, it is called a **concave polygon**



Polygon Clipping



Positive Orientation

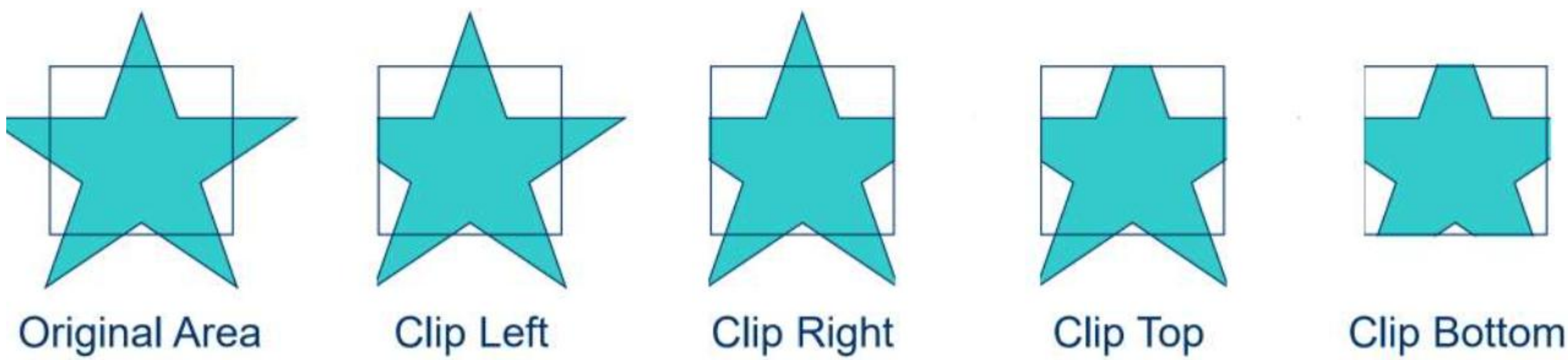
Negative Orientation

Polygon Clipping

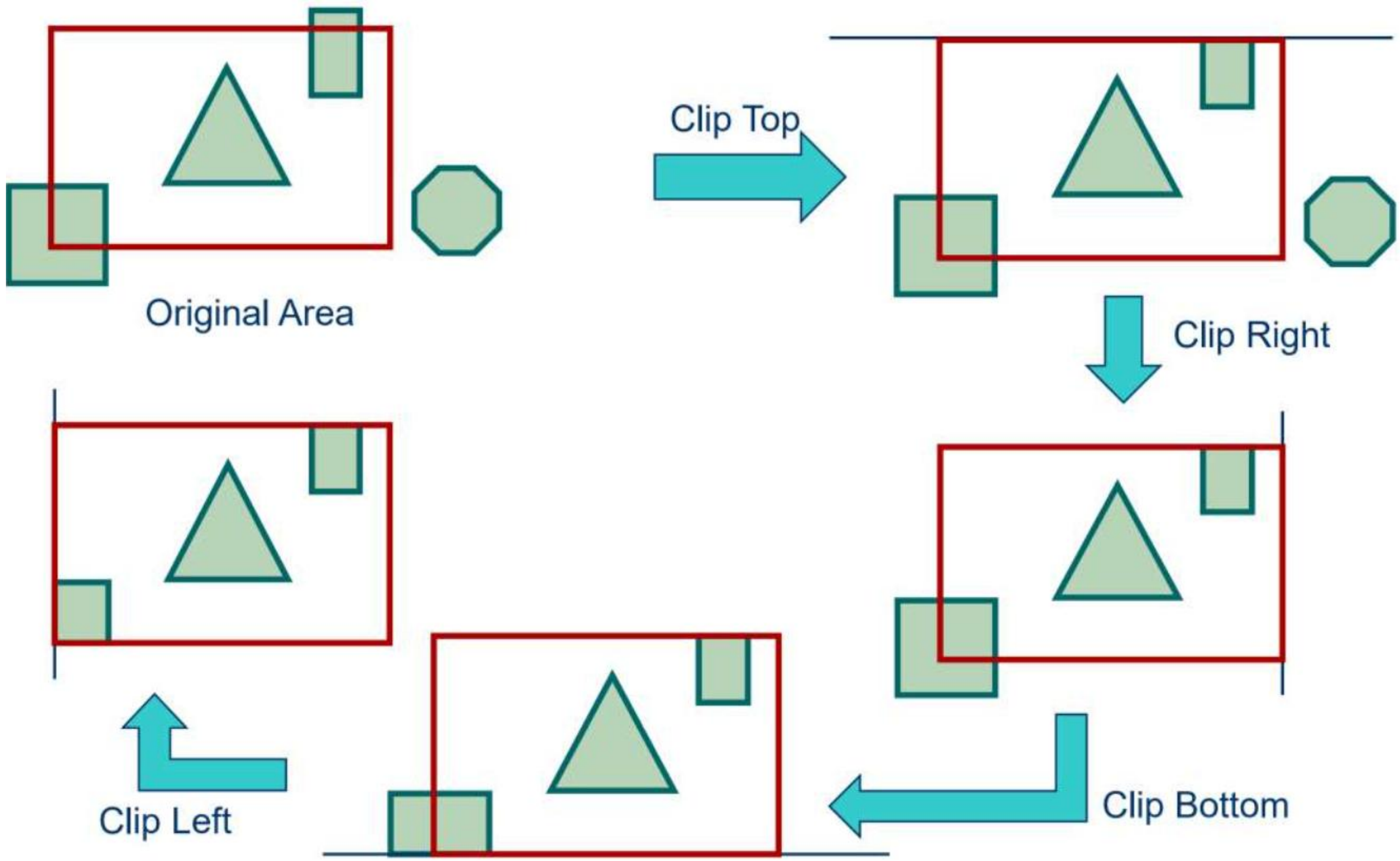
- The Left-hand side of any directed edge $\overline{P_{i-1}P_i}$ or $\overline{P_N P_1}$ points inside the polygon
- Let, a point P(x,y). If it is to the left of every edge of the polygon, it is inside the polygon

Sutherland-Hodgman Algorithm

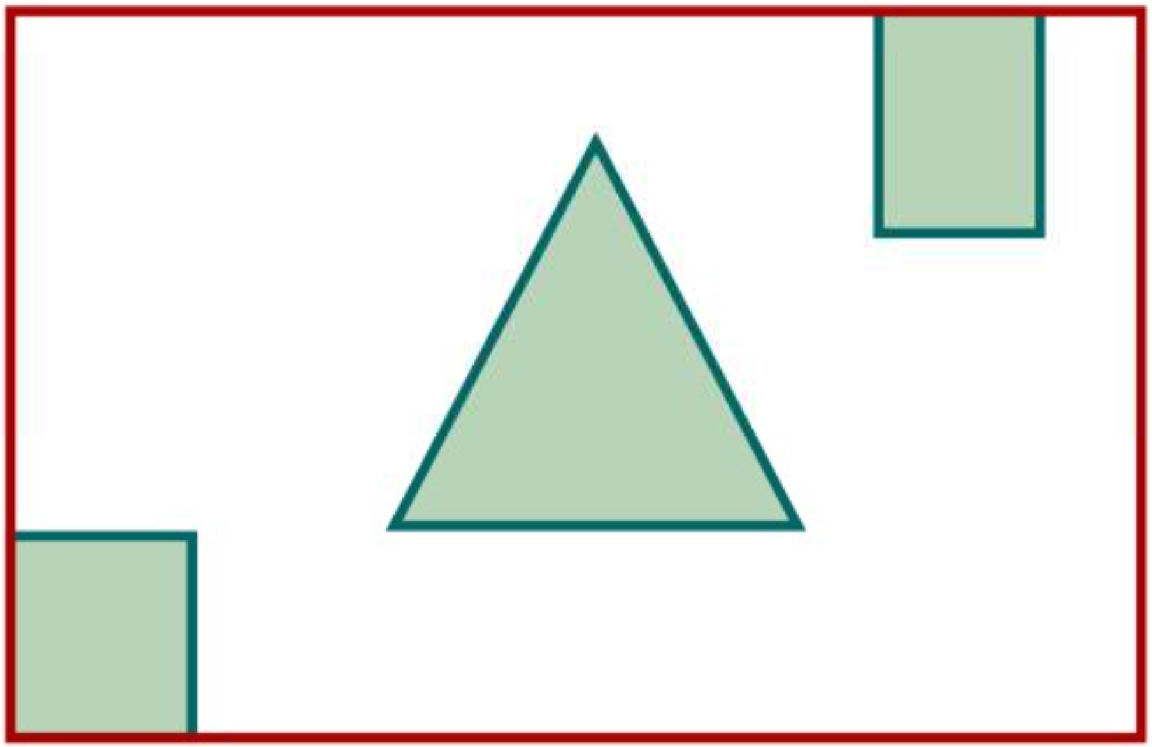
- A technique for clipping areas developed by Sutherland & Hodgman
- Put simply the polygon is clipped by comparing it against each boundary in turn



Sutherland-Hodgman Polygon Clipping



Polygon Clipping



After Clipping

Sutherland-Hodgman Algorithm

- P_1, P_2, ..., P_N be vertex list of the polygon to be clipped (**subject polygon**).
- Edge E, defined by points A and B, be any edge of the positively oriented, convex **clipping polygon**
- Vertex output list - a list containing the vertices that are to be displayed after clipping

Sutherland-Hodgman Algorithm

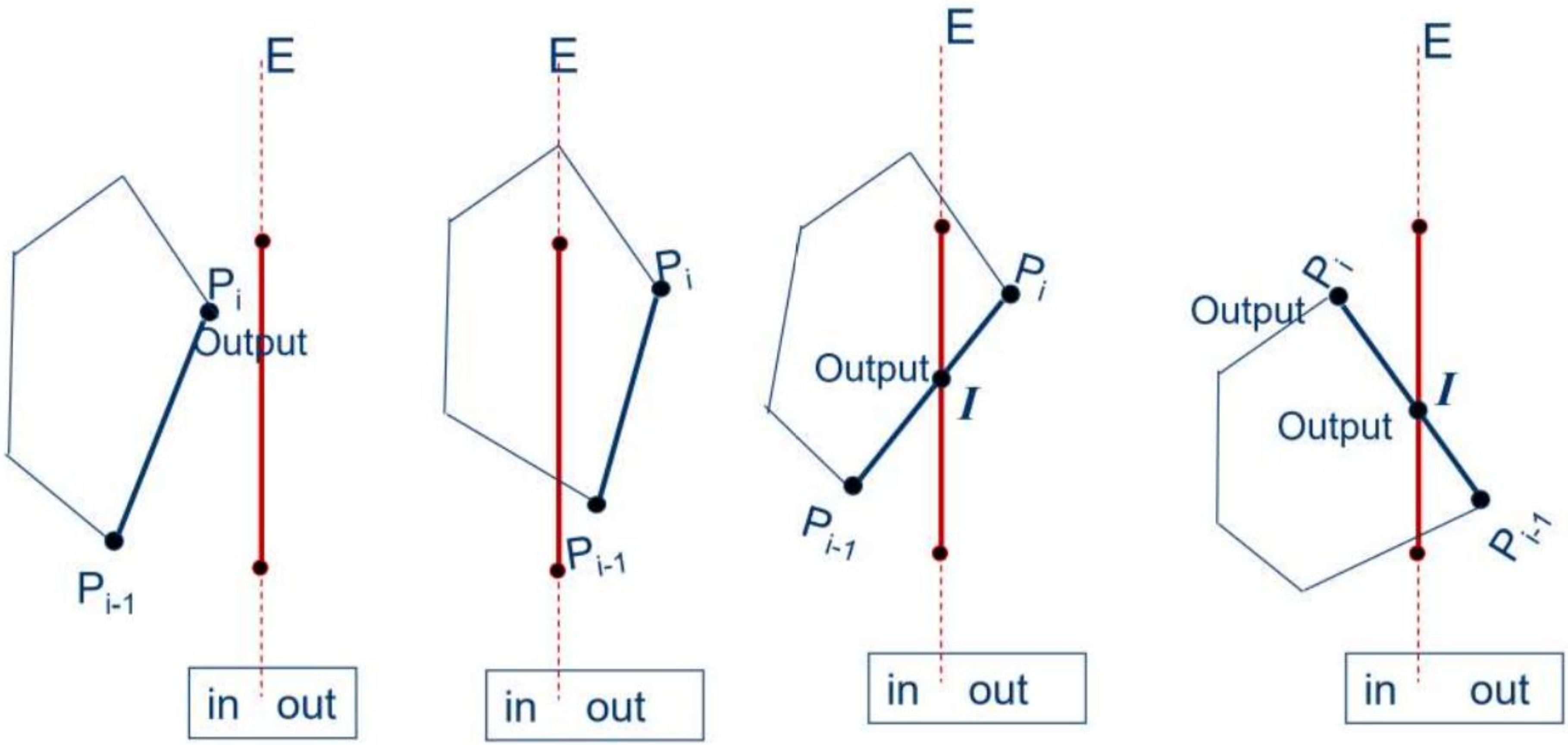
Consider edge $\overline{P_{i\_1}P_i}$

- If both P_{i-1} and P_i are left of the edge E, place P_i to the vertex output list
- If both P_{i-1} and P_i are right of the edge, place nothing to the vertex output list
- If P_{i-1} is left and P_i is right of the edge, find intersect point **I** and place **I** to the vertex output list
- If P_{i-1} is right and P_i is left of the edge, find intersect point **I** and place both **I** and P_i to the vertex output list

The Algorithm proceeds by passing each clipped polygon to the next edge of the window.

Sutherland-Hodgman Algorithm

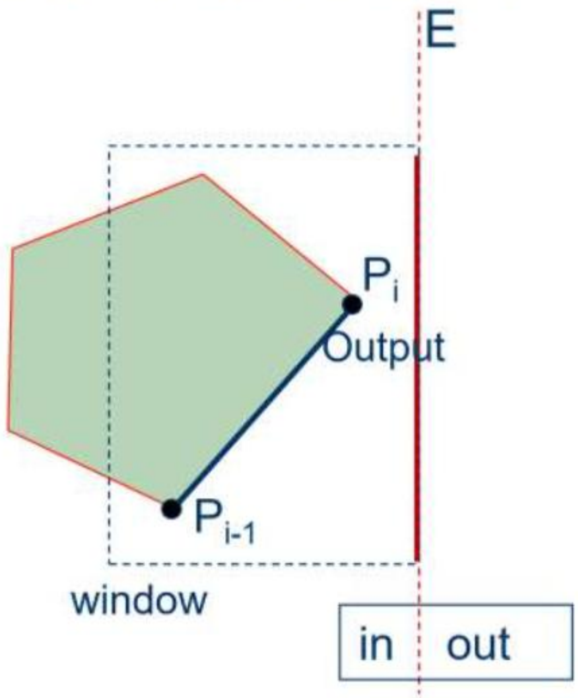
- 4 cases



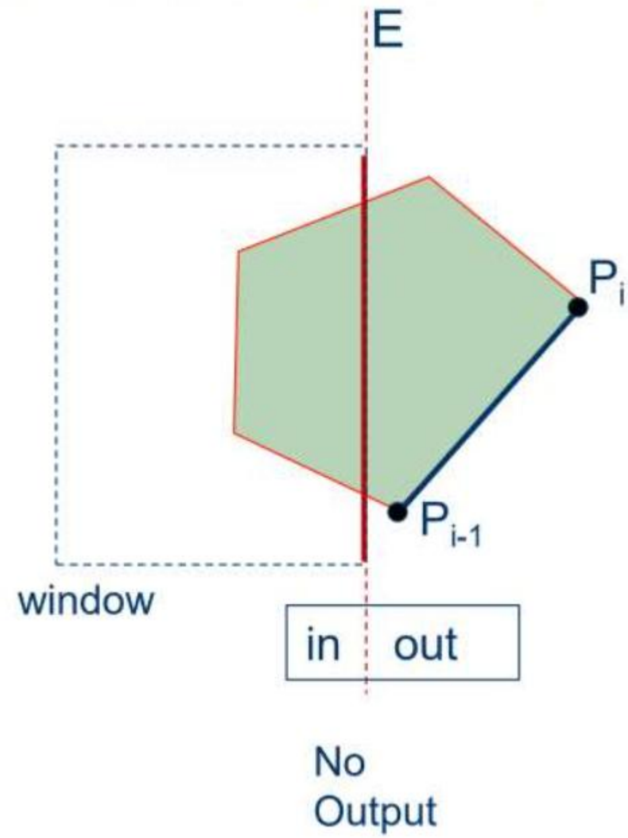
Sutherland-Hodgman Algorithm

- 4 cases

both P_{i-1} and P_i are left / inside



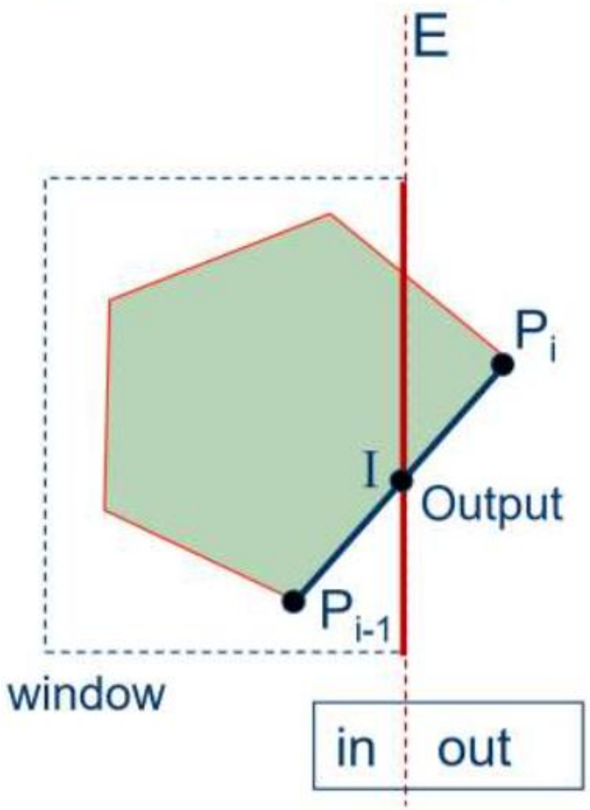
both P_{i-1} and P_i are right / outside



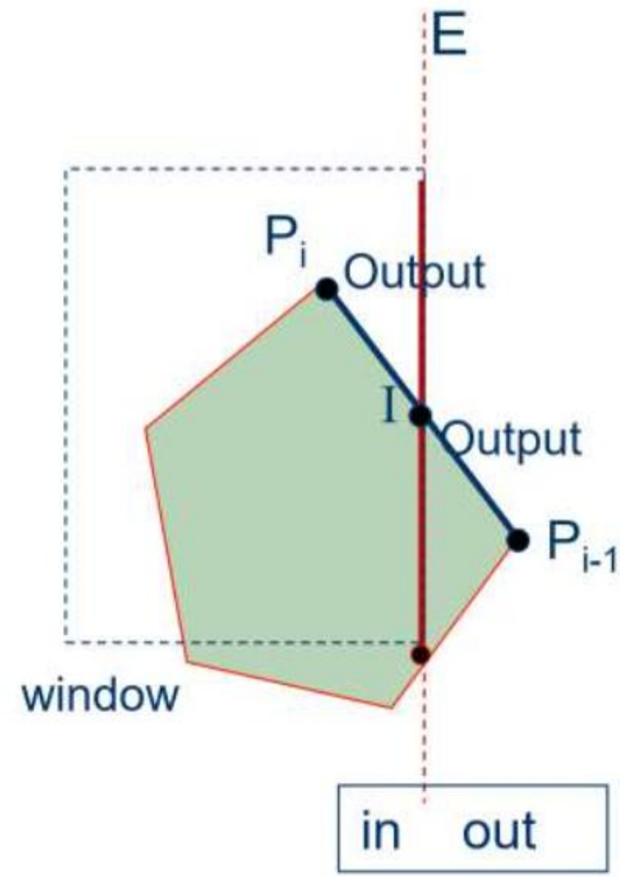
Sutherland-Hodgman Algorithm

- 4 cases

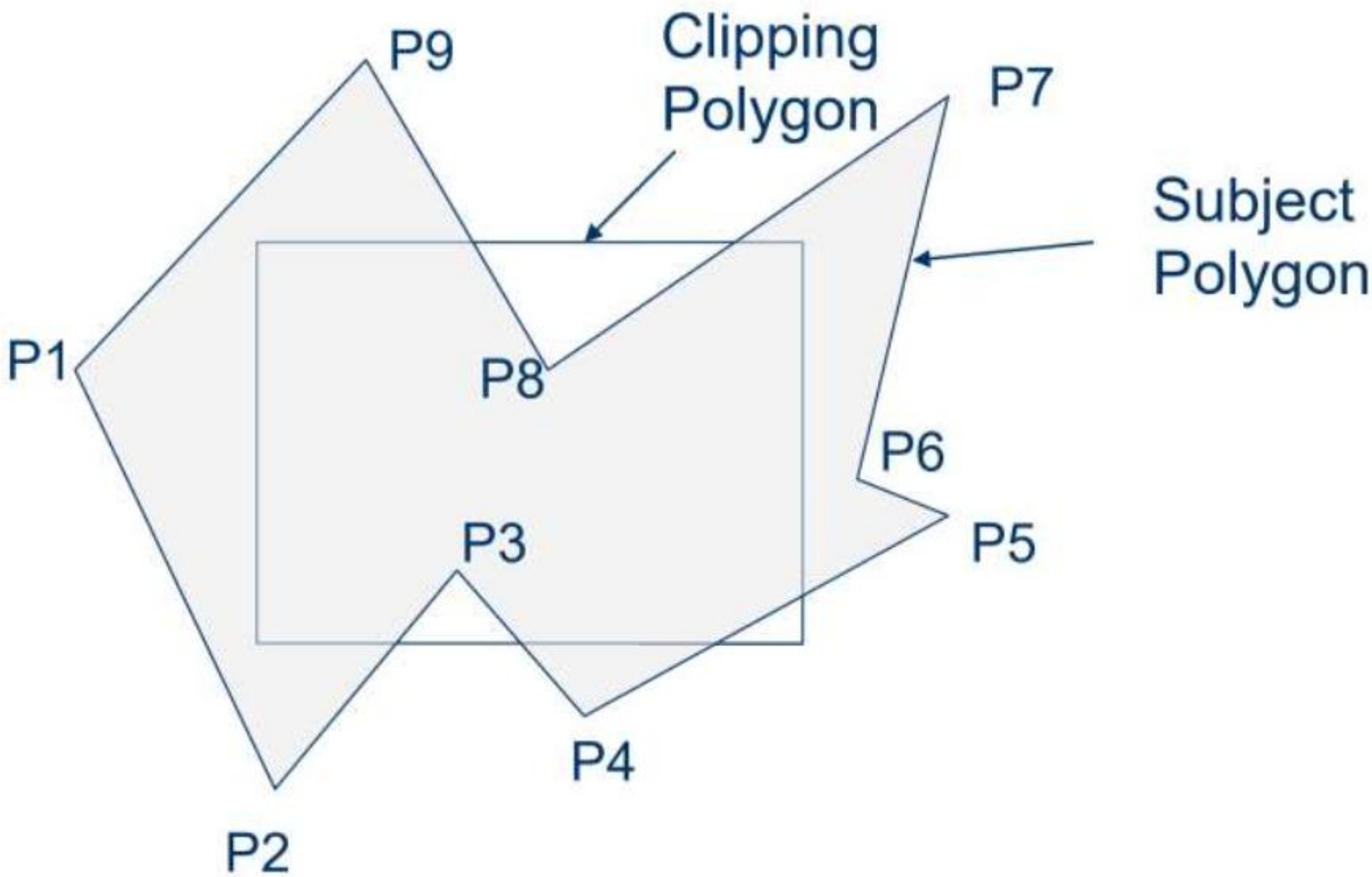
P_{i-1} is left and P_i is right



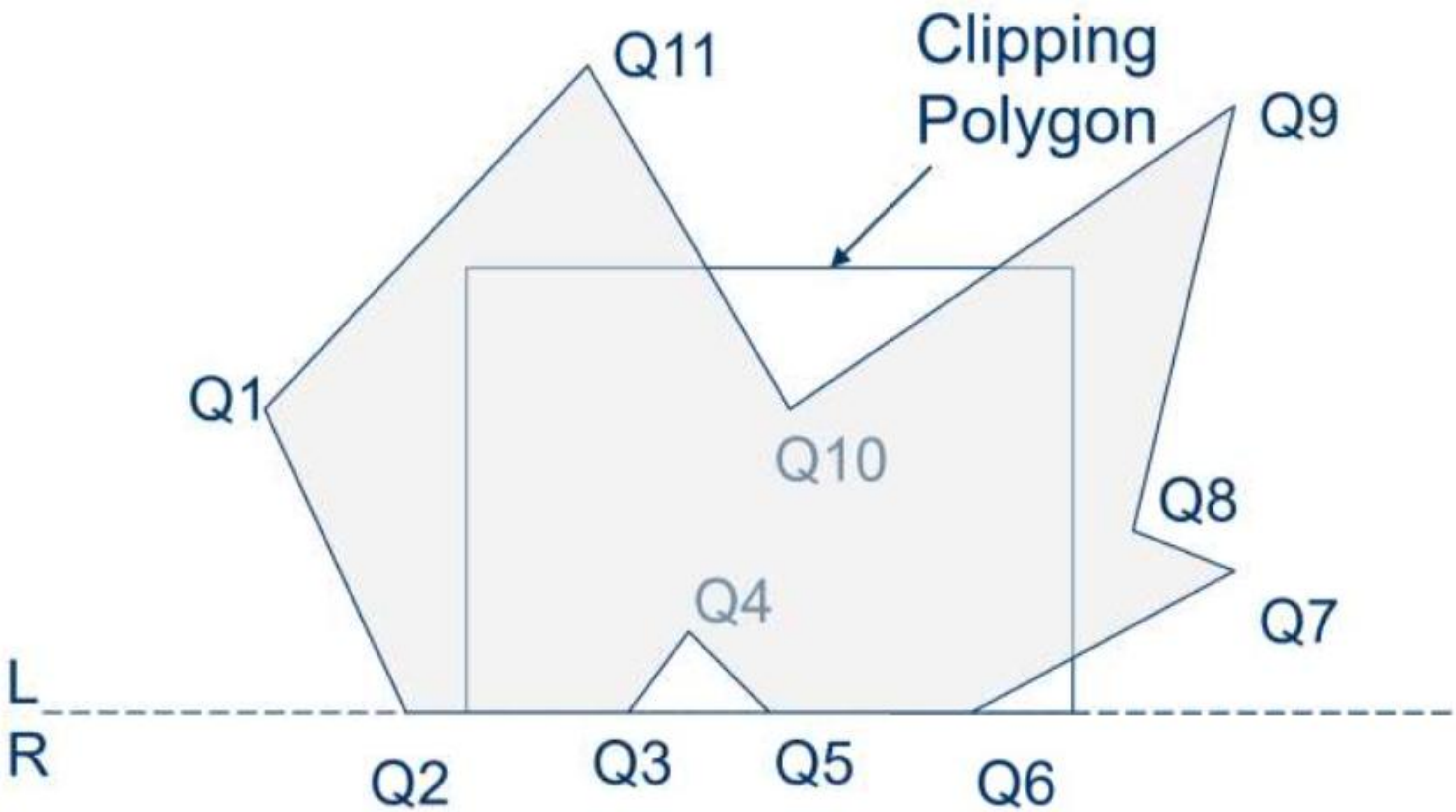
P_{i-1} is right and P_i is left



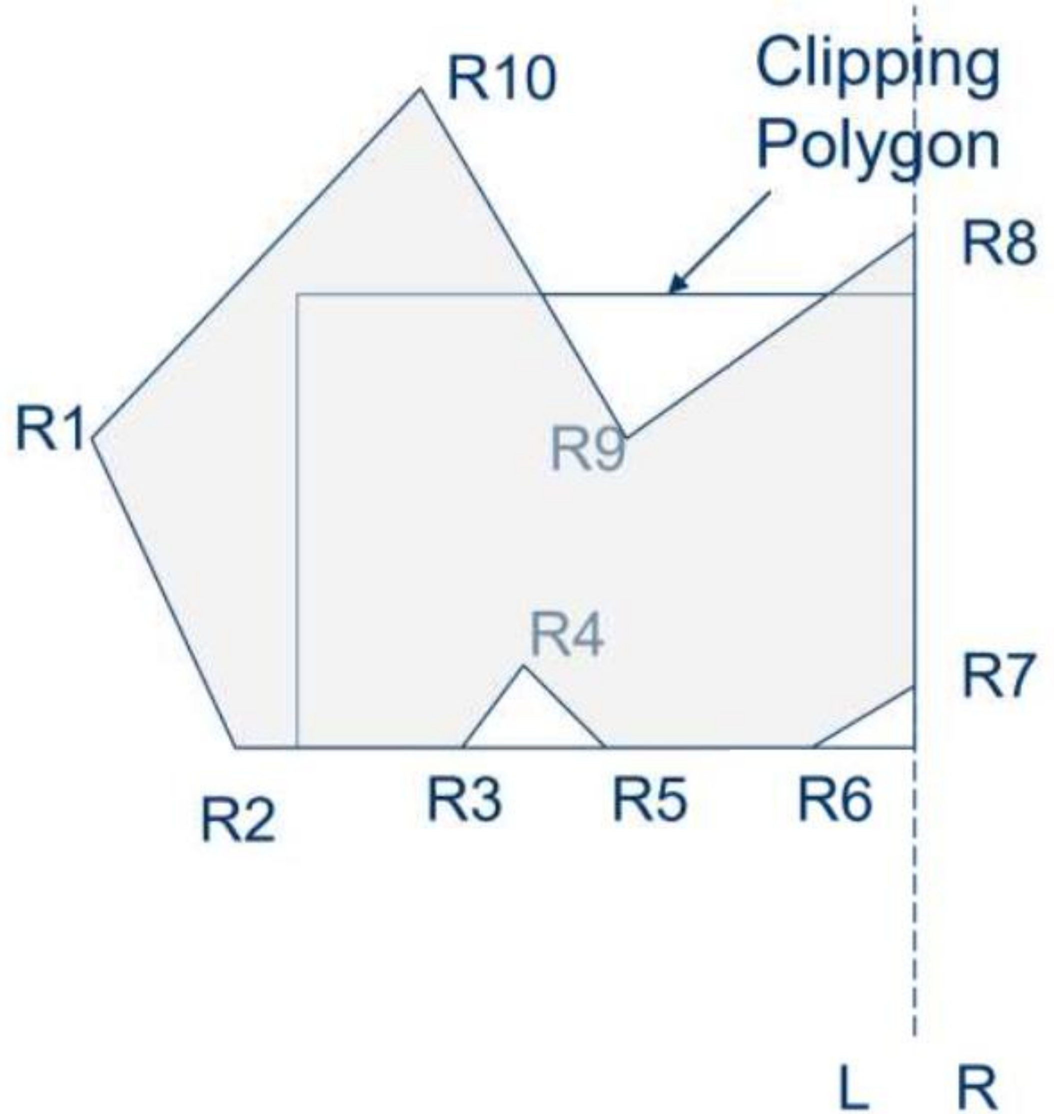
Example



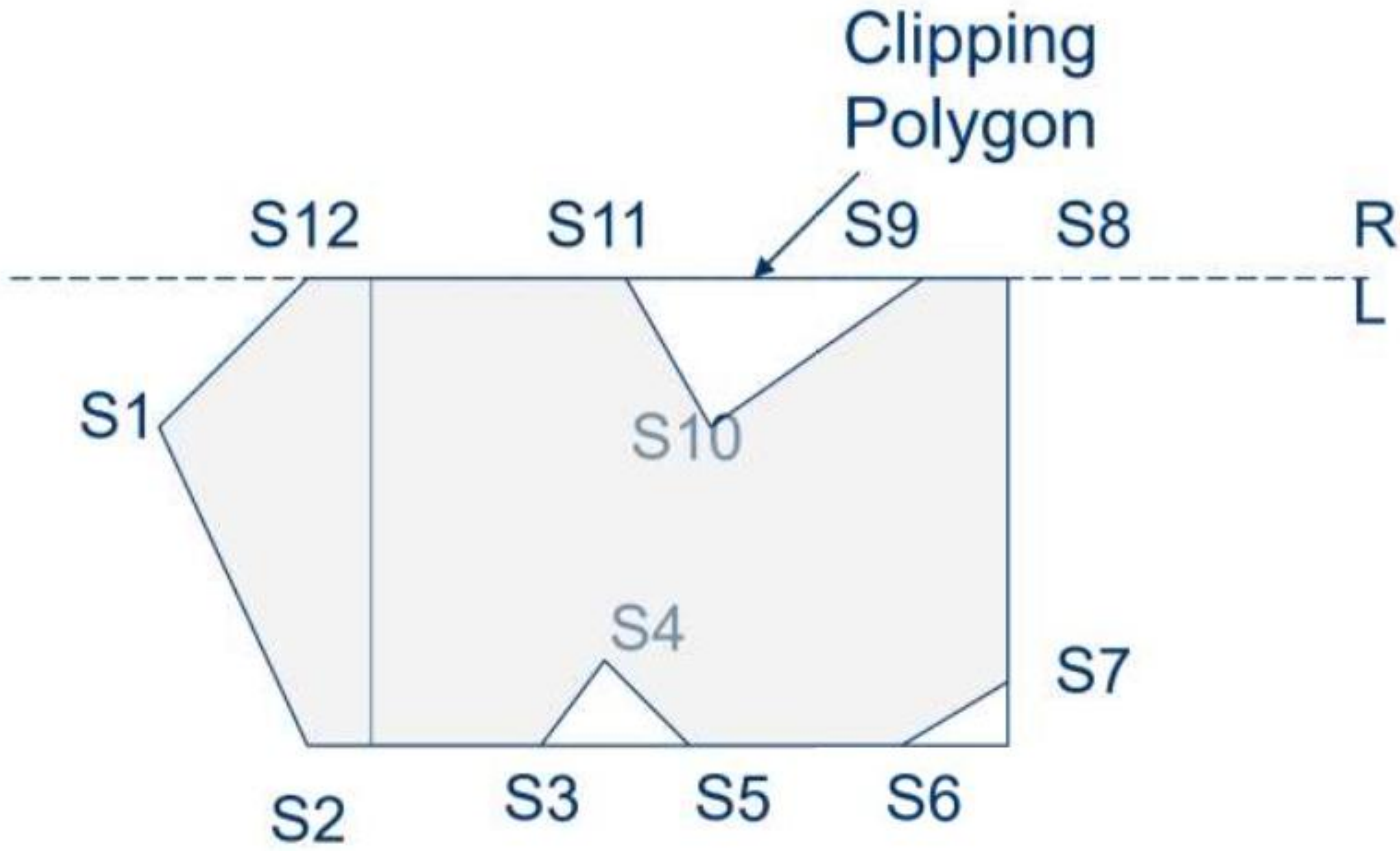
Example



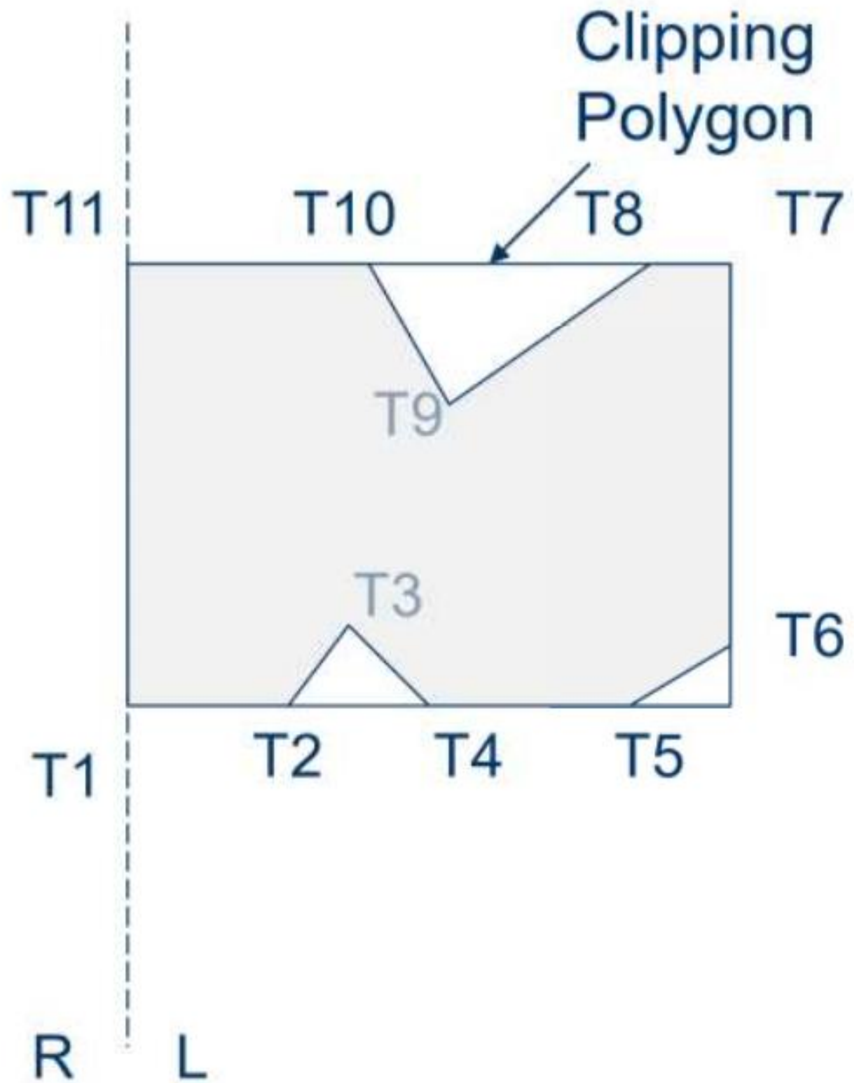
Example



Example



Example



Point-to-line test

- A very general test to determine if a point p is “inside” a line L or plane L for 3D, defined by q and n:

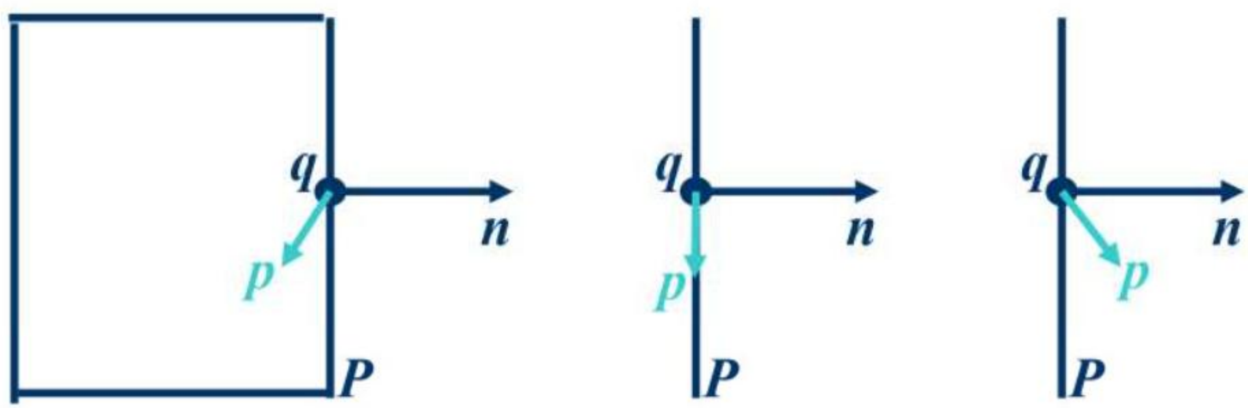
  $(p - q) \cdot n < 0$: p inside $\boldsymbol{L}$

  $(p - q) \cdot n = 0$: p on $\boldsymbol{L}$

  $(p - q) \cdot n > 0$: p outside $\boldsymbol{L}$

Remember: $\boldsymbol{A} \cdot \boldsymbol{B} = |\boldsymbol{A}|\,|\boldsymbol{B}| \cos(\theta) = \boldsymbol{A}_x . \boldsymbol{B}_x + \boldsymbol{A}_y . \boldsymbol{B}_y$

θ = angle between $\boldsymbol{A}$ and $\boldsymbol{B}$



Finding Line-edge interactions

- Use parametric definition of Line and edge:

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

- Line intersects an clipping edge where L(t) is on E
  - q is a point on E
  - n is normal to E

$$(L(t) - q) \cdot n = 0$$

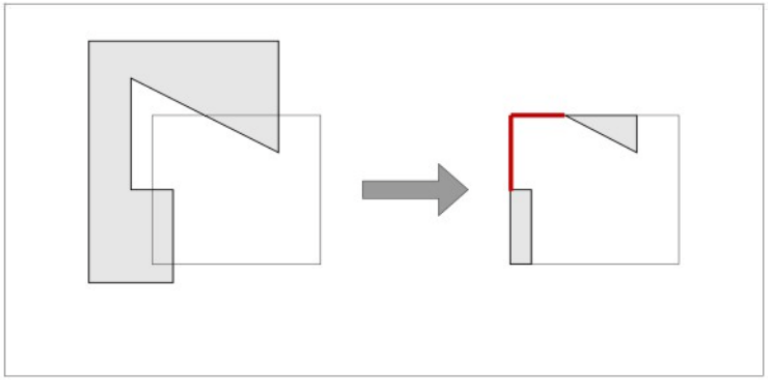
$$(P_0 + t(P_1 - P_0) - q) \cdot n = 0$$

$$t = [(q - P_0) \cdot n] / [(P_1 - P_0) \cdot n]$$

  - The intersection point $I = L(t)$ for this value of t

Unwanted effect

- The Sutherland-Hodgman algorithm correctly clips convex polygons, but concave polygons may be displayed with extraneous lines
- Since there is only one output vertex list, the last vertex in the list is always joined to the first vertex.



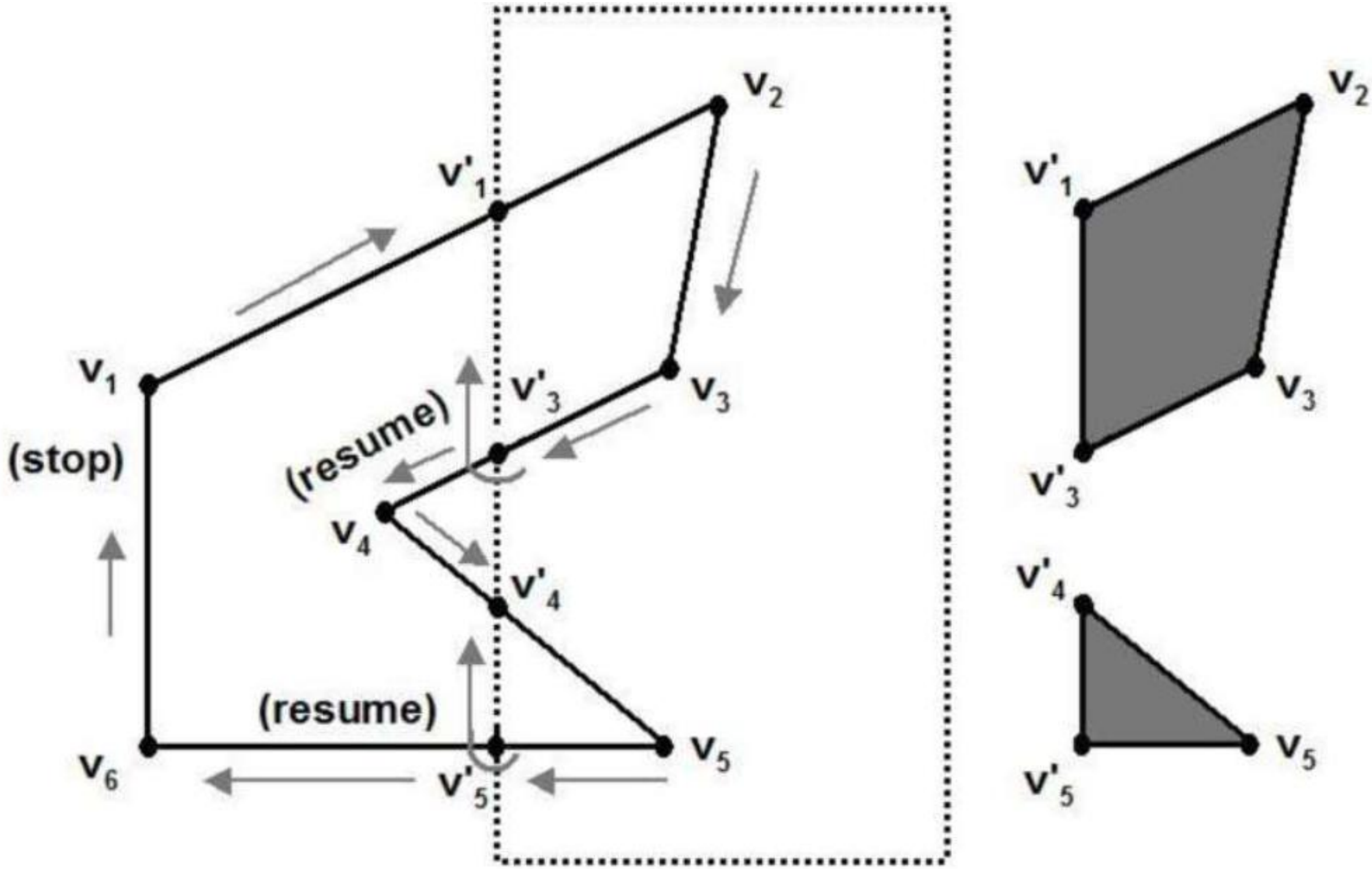
Weiler-Atherton Polygon Clipping

- can be used to clip either a convex or a concave polygon.
- The basic idea of this algorithm is that instead of proceeding around the polygon edges as vertices are processed, we will follow the window boundaries.
- The path we follow depends on:
  - polygon-processing direction (clockwise or counterclockwise)
  - The pair of polygon vertices
    - outside-to-inside or inside-to-outside.

Weiler-Atherton Polygon Clipping

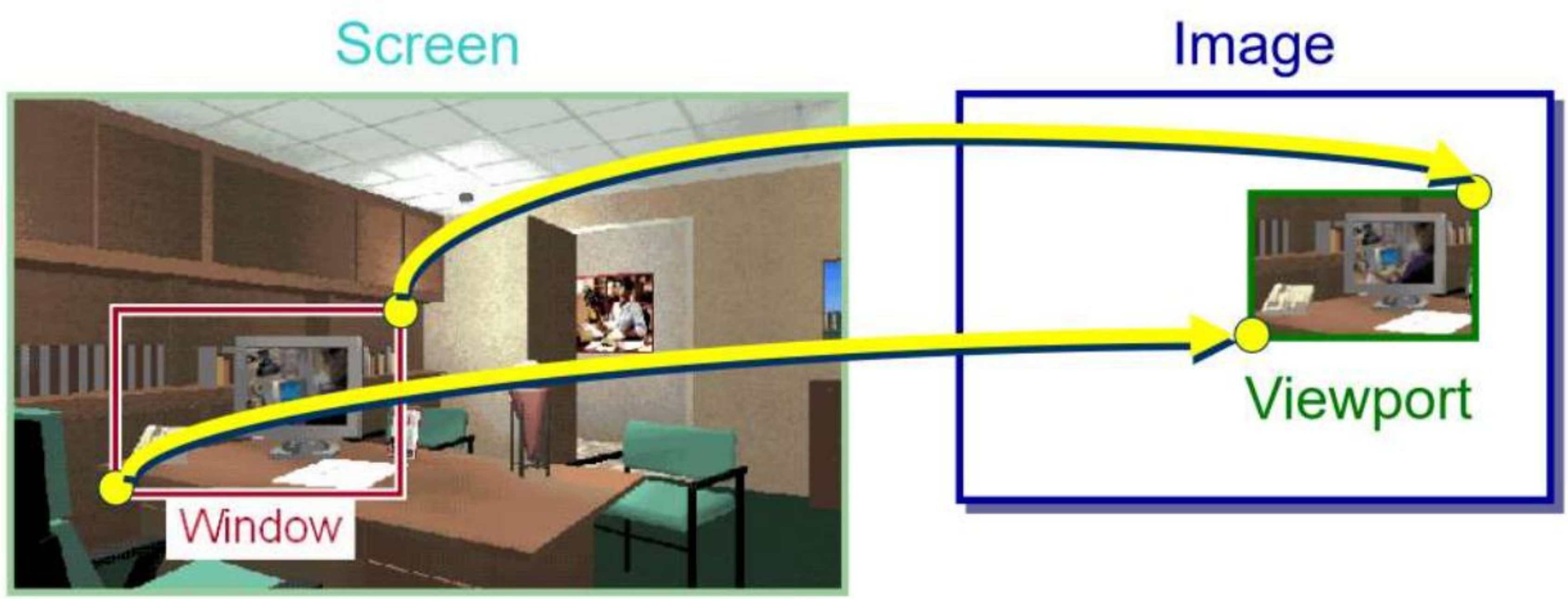
- For clockwise processing of polygon vertices, we use the following rules:
- For an outside-to-inside pair of vertices, follow polygon boundaries.
- For an inside-to-outside pair of vertices, follow window boundaries in a clockwise direction.

Weiler-Atherton Polygon Clipping



Viewport Transformation

- Transform 2D Geometric Primitives from Screen Coordinate System (Projection Coordinates) to Image Coordinate System (Device Coordinates)

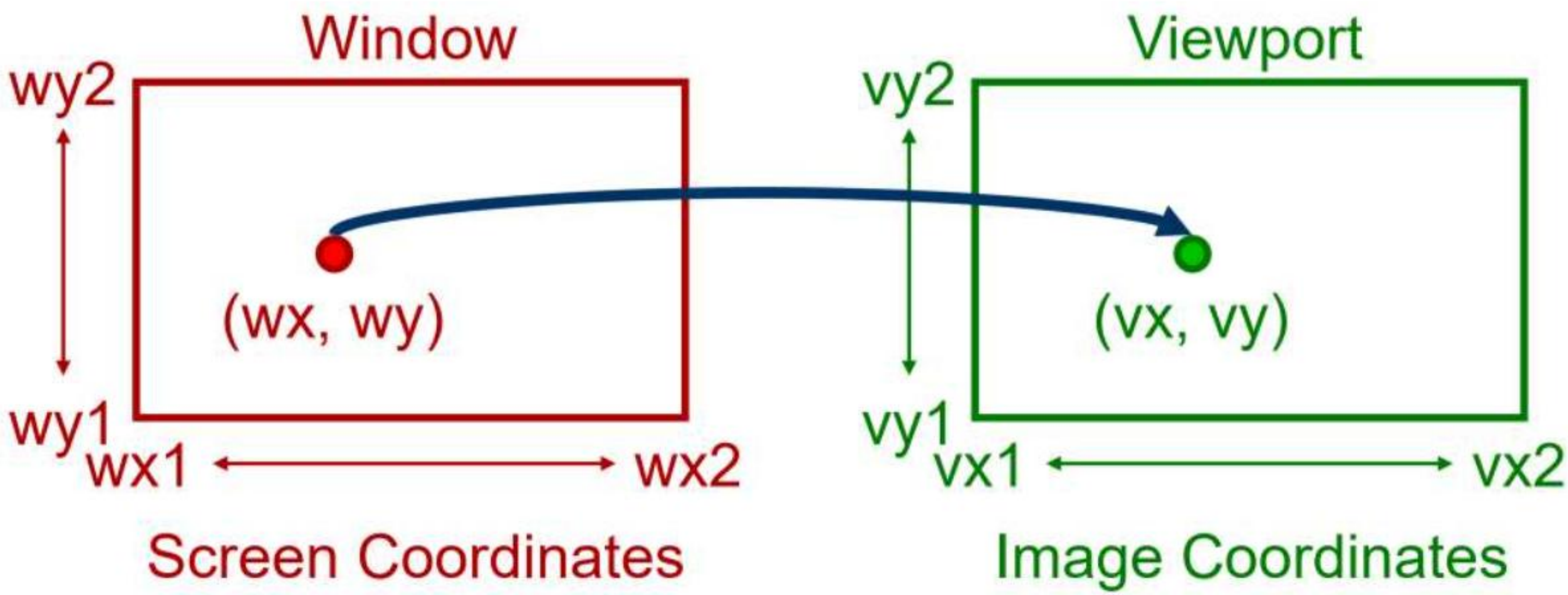


Window vs Viewport

- Window
  - World-coordinate area selected for display
  - What is to be viewed
- Viewport
  - Area on the display device to which a window is mapped
  - Where it is to be displayed

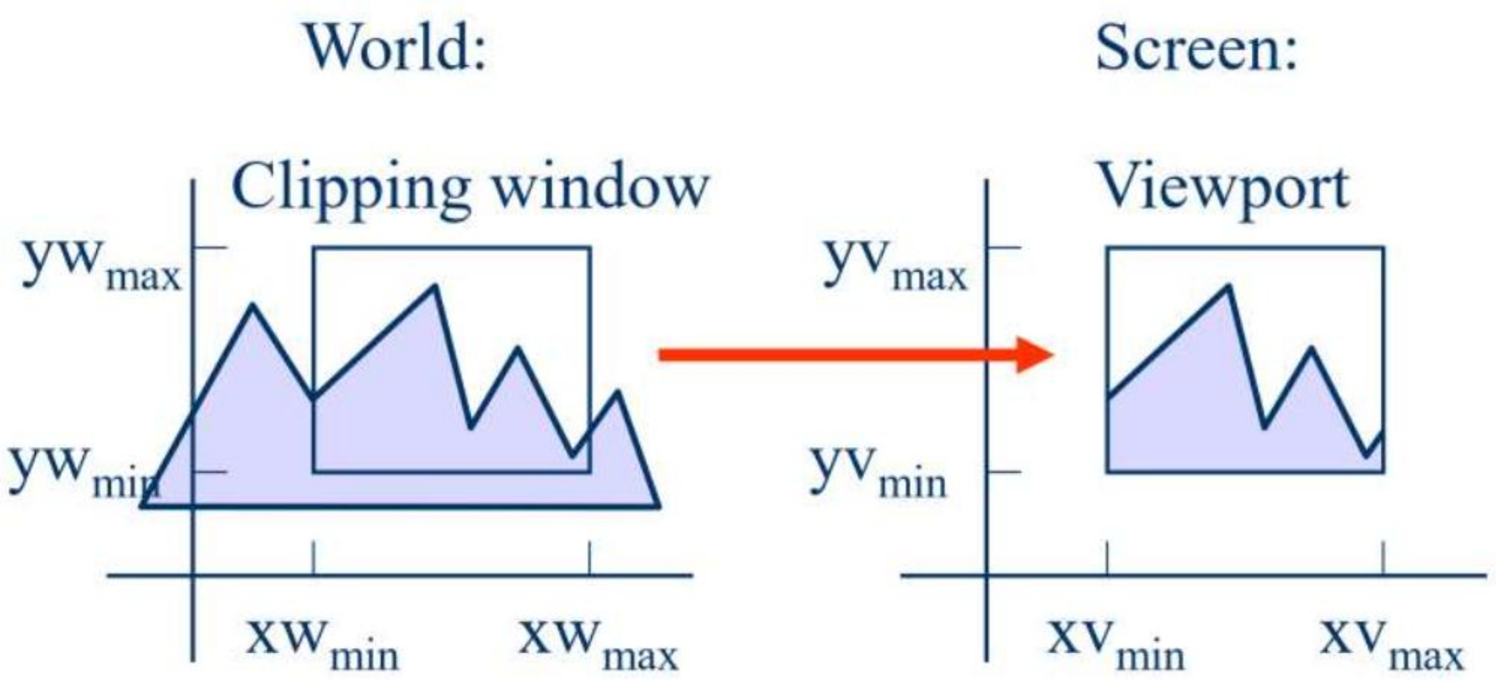
Viewport Transformation

- Window-to-Viewport Mapping




```
vx = vx1 + (wx - wx1) * (vx2 - vx1) / (wx2 - wx1);
vy = vy1 + (wy - wy1) * (vy2 - vy1) / (wy2 - wy1);
```

Viewport Transformation



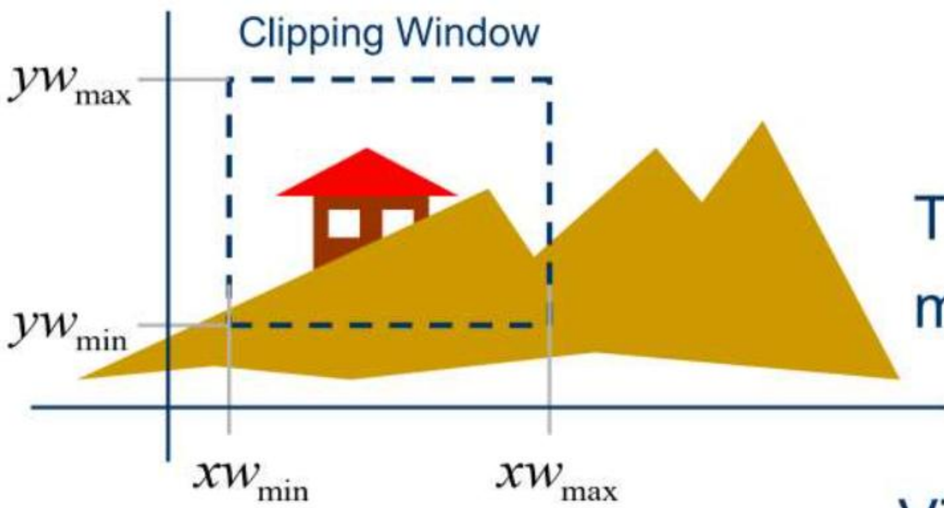
Clipping window:

What do we want to see?

Viewport:

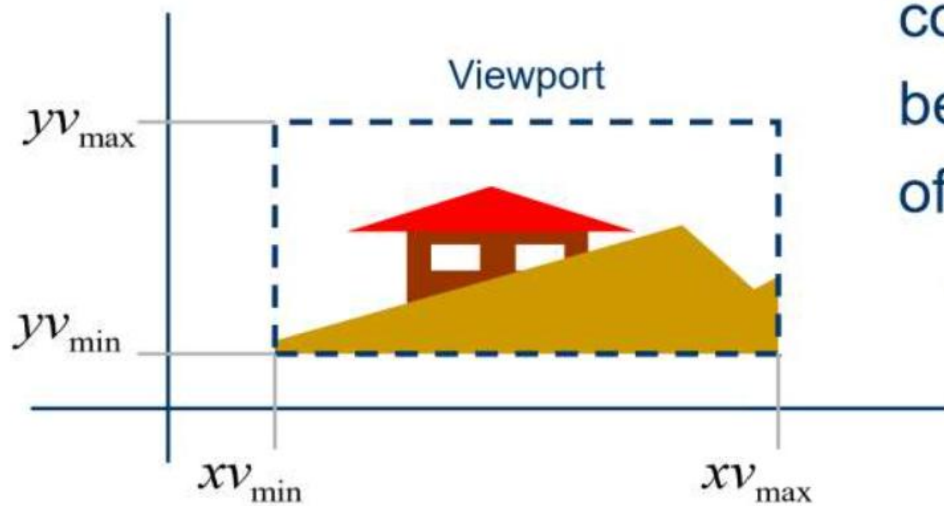
Where do we want to see it?

Viewport Transformation



The clipping window is mapped into a viewport.

Viewing world has its own coordinates, which may be a non-uniform scaling of world coordinates.



Viewport Coordinates

References

- Angel and Shreiner, Interactive Computer Graphics, 6th edition
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition
- Dr. Sk. Md. Masudul Ahsan, (2022), L7 Clipping hov [PDF document], Khulna University of Engineering & Technology.