COMPUTER GRAPHICS

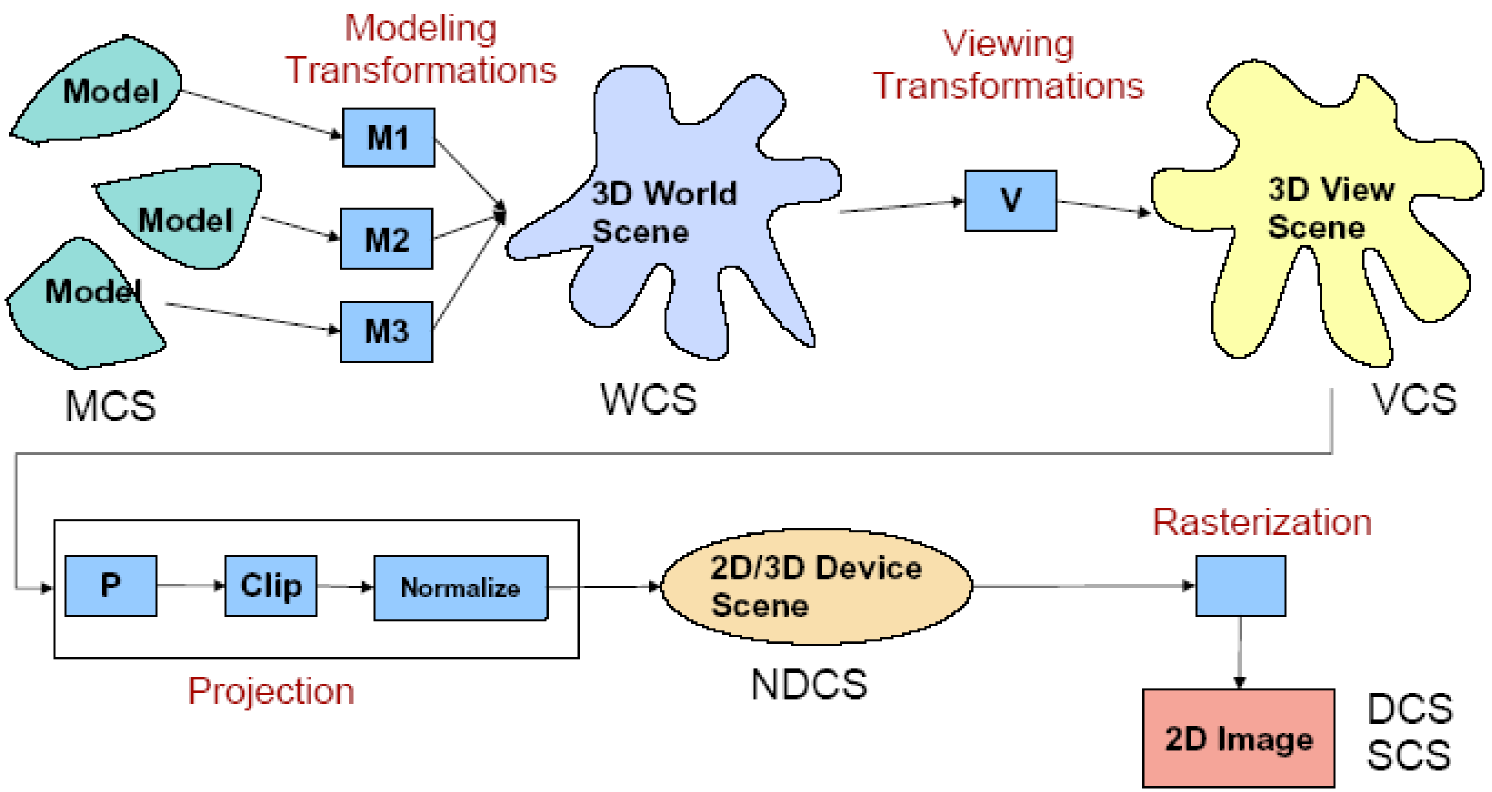
Two-Dimensional Viewing
Hearn & Baker Chapter 6

Slides are taken from Robert Thomsons notes.

OVERVIEW

- Two dimensional viewing pipeline
- The clipping window
- Normalizations and viewport transformations
- OpenGL 2D viewing functions
- Clipping algorithms
- Point, line, and fill-area clipping

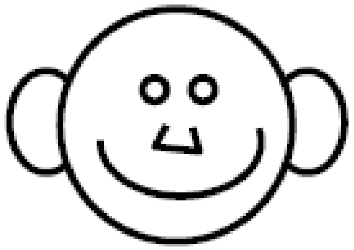
Viewing Pipeline Revisited



- Model coordinates to World coordinates: Modelling transformations

Model coordinates:
1 circle (head),
2 circles (eyes),
1 line group (nose),
1 arc (mouth),
2 arcs (ears).
With their relative
coordinates and sizes

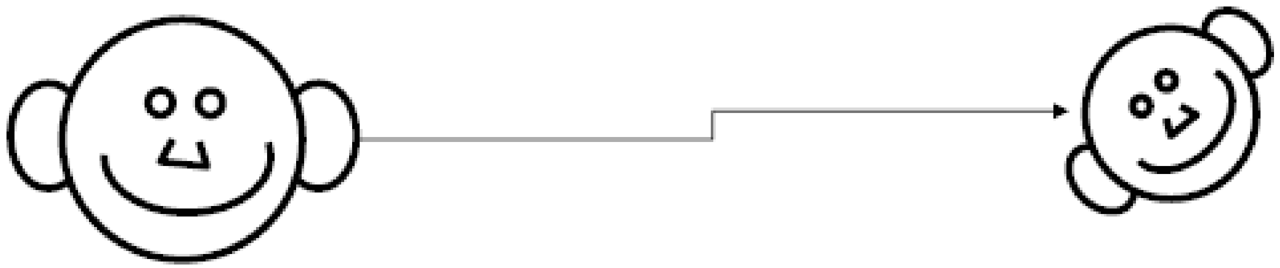
World coordinates:
All shapes with their
absolute coordinates and sizes.
circle(0,0,2)
circle(-.6,.8,.3) circle(.6,.8,.3)
lines[(-.4,0),(-.5,-.3),(.5,.3),(.4,0)]
arc(-.6,0,.6,0,1.8,180,360)
arc(-2.2,.2,-2.2,-.2,.8,45,315)
arc(2.2,.2,2.2,-.2,.8,225,135)



- World coordinates to Viewing coordinates: Viewing transformations

World coordinates

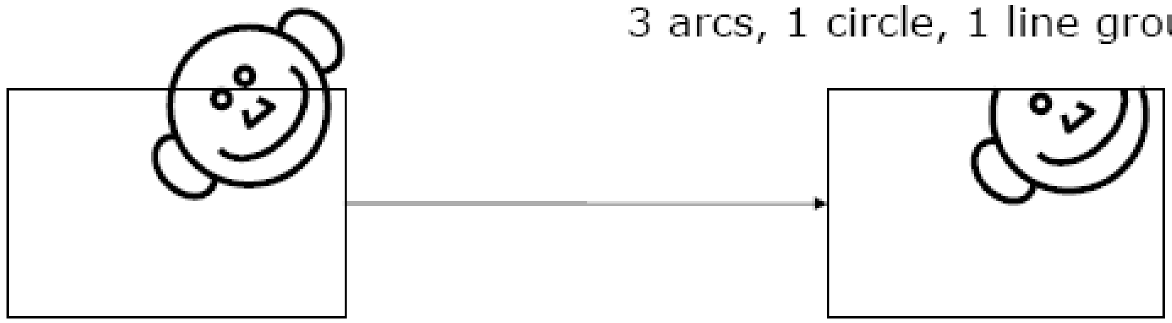
Viewing coordinates:
Viewers position and view angle. i.e. rotated/translated



- Projection: 3D to 2D. Clipping depends on viewing frame/volume. Normalization: device independent coordinates

Viewing coordinates:

Device Independent Coordinates:
Invisible shapes deleted, others reduced to visible parts.
3 arcs, 1 circle, 1 line group

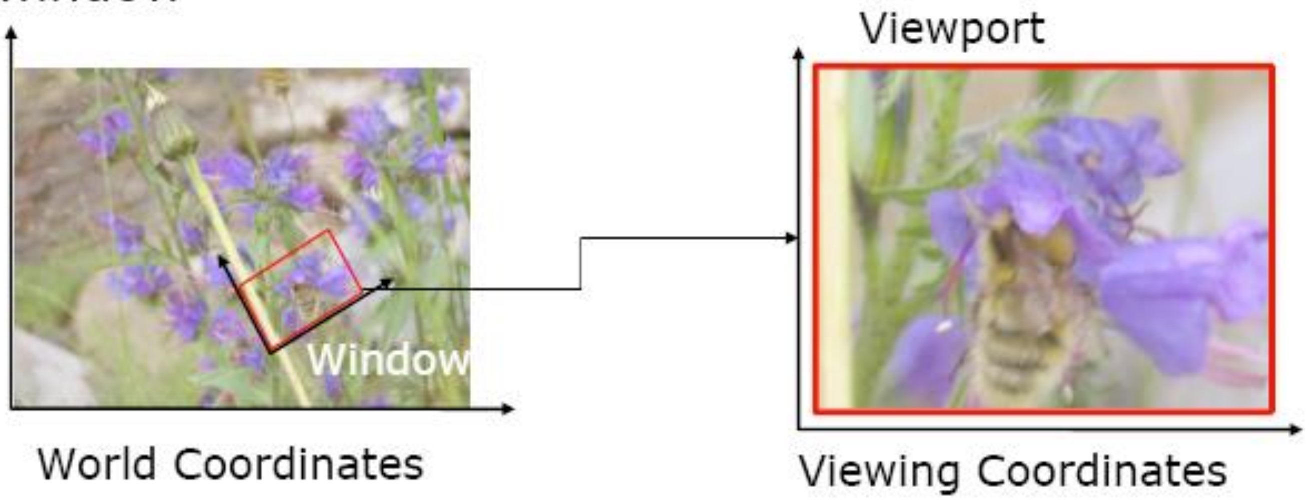


2-D viewing transformation pipeline

- Model coordinates
  - Construct world coordinate scene using modeling coordinate transformations $\rightarrow$
- World coordinates
  - Convert world coordinates to viewing coordinates $\rightarrow$
- Viewing coordinates
  - Transform viewing coordinates to normalised coordinates $\rightarrow$
- Normalized coordinates
  - Map normalized coordinates to device coordinates $\rightarrow$
- Device Coordinates

2D Viewing

- World coordinates to Viewing coordinates
- Window to Viewport.
  Window: A region of the scene selected for viewing (also called *clipping window*)
  Viewport: A region on display device for mapping to window



Graphics packages commonly allow only rectangular clipping windows aligned with the x- and y-axes

Must implement own clipping and coordinate transformations for other form of clipping

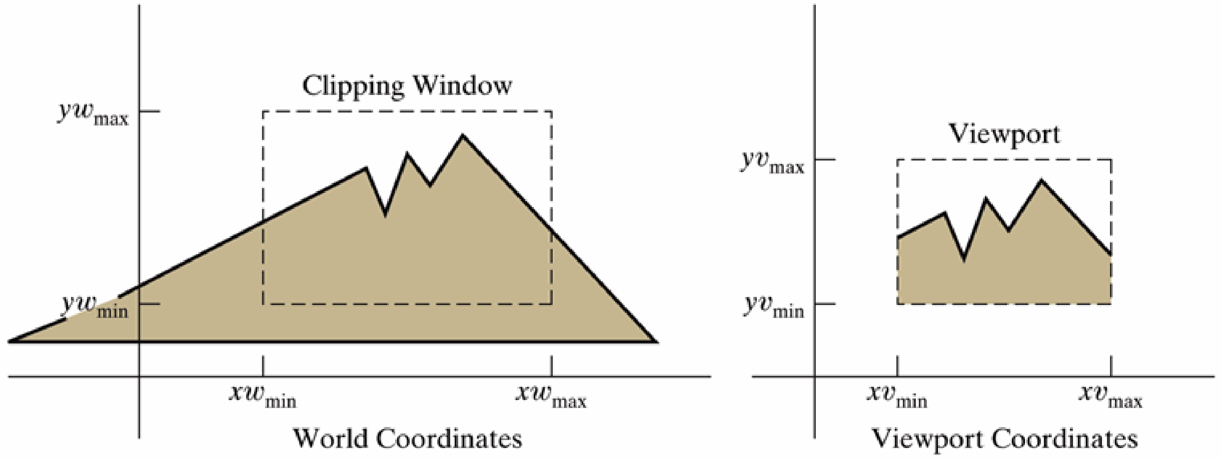
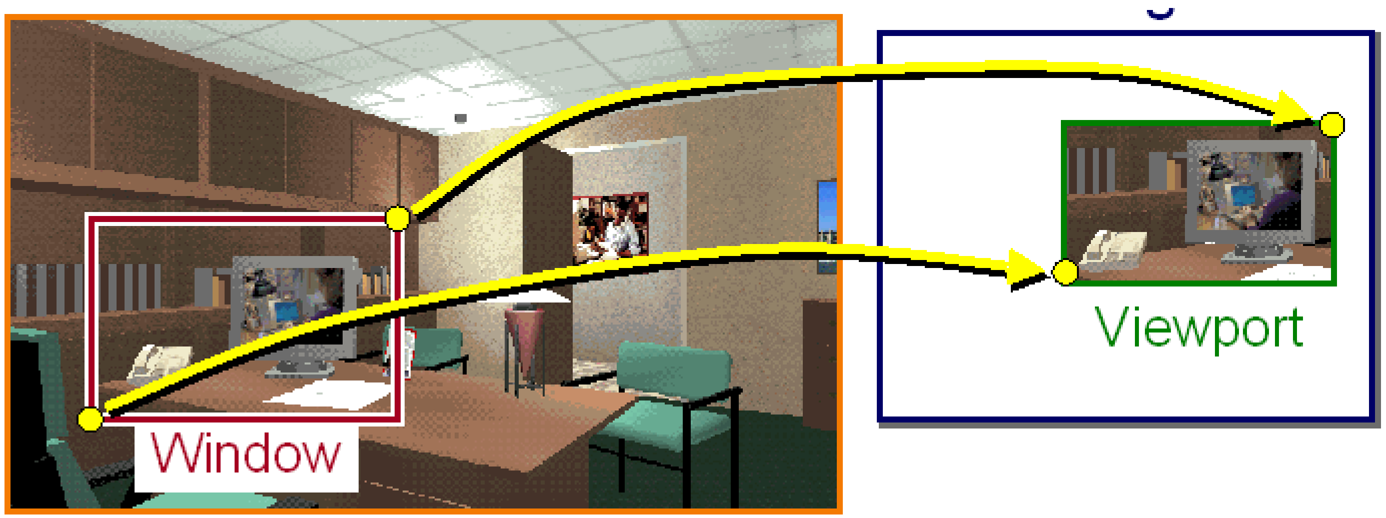


Figure 6-2

A clipping window and associated viewport, specified as rectangles aligned with the coordinate axes.

Clipping Window vs. Viewport

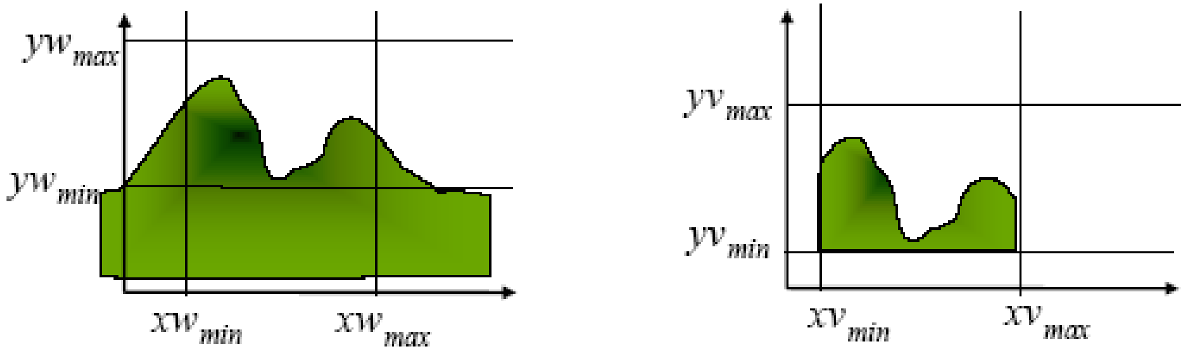
- The clipping window selects what we want to see in our virtual 2D world.
- The *viewport* indicates where it is to be viewed on the output device (or within the display window)
- By default the *viewport* has the same location and dimensions as the GLUT display window you create
  - But it can be modified so that only a part of the display window is used for OpenGL display


Window
Viewport

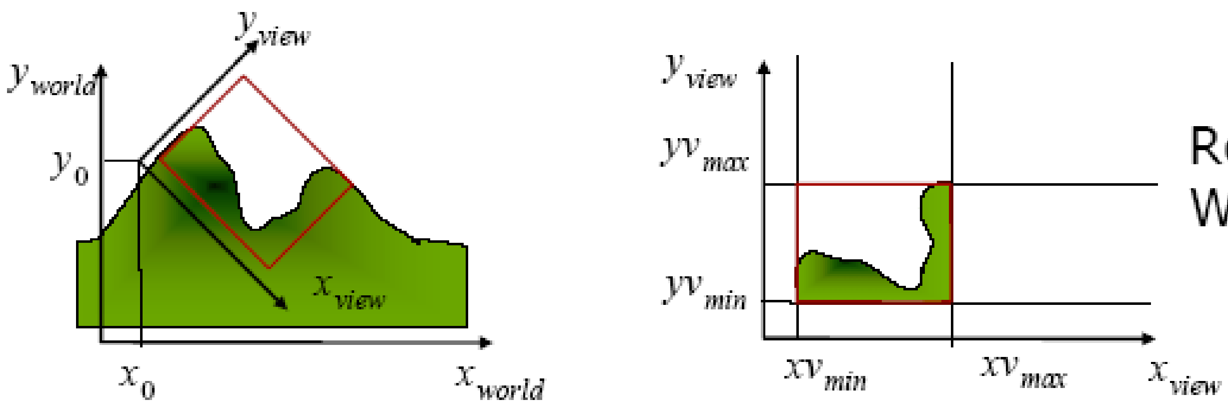
(clipping) window to viewport transformation

- Zooming
  - is successive mapping of different sized clipping windows to the viewport
    - reducing clip window : zoom in on part of scene
    - increase clip window : zoom out
- Panning
  - moving a fixed-size clipping window across the scene

The clipping window



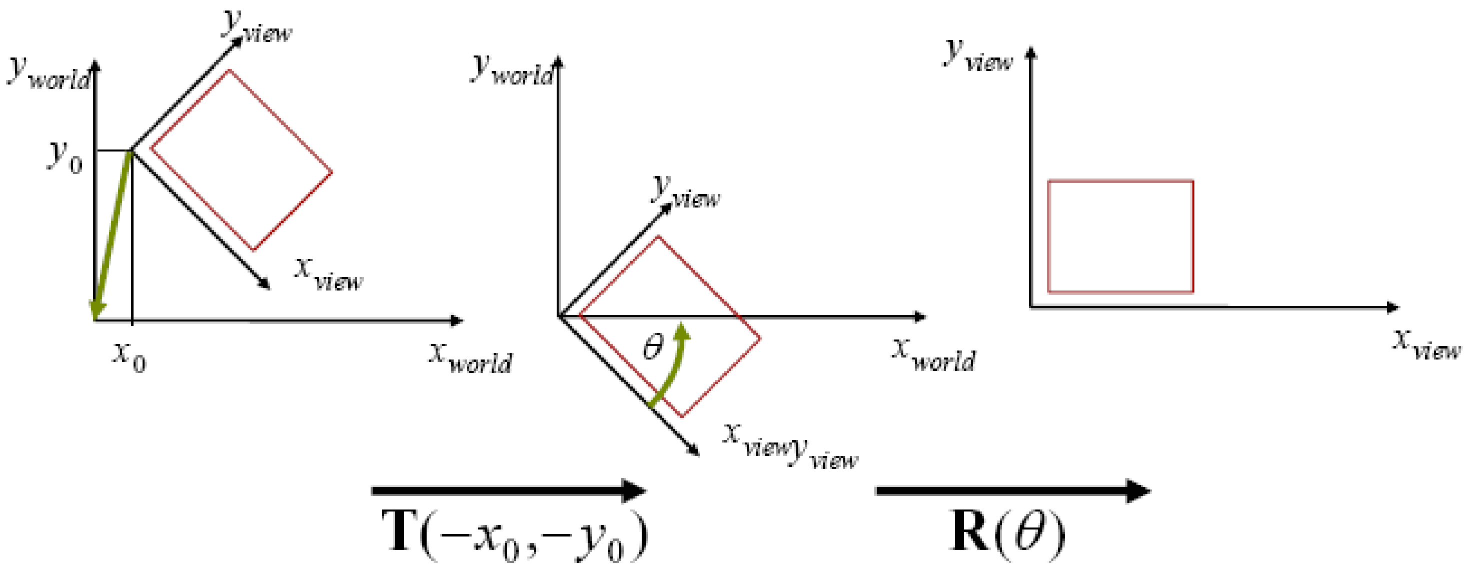
Rectangular Window



Rotated Window

From world coordinates to view coordinates

World-coordinates to Viewing Coordinates



- $M_{wc,vc} = R \cdot T$

Normalization and viewport transformations

- Some graphics packages combine normalisation and window-to-viewport transformations into a single operation
  - viewport coordinates are often given in the range 0 to 1.
    - Viewport is within a unit square
  - after clipping the unit square is mapped to the output display device

Window to viewport transformation

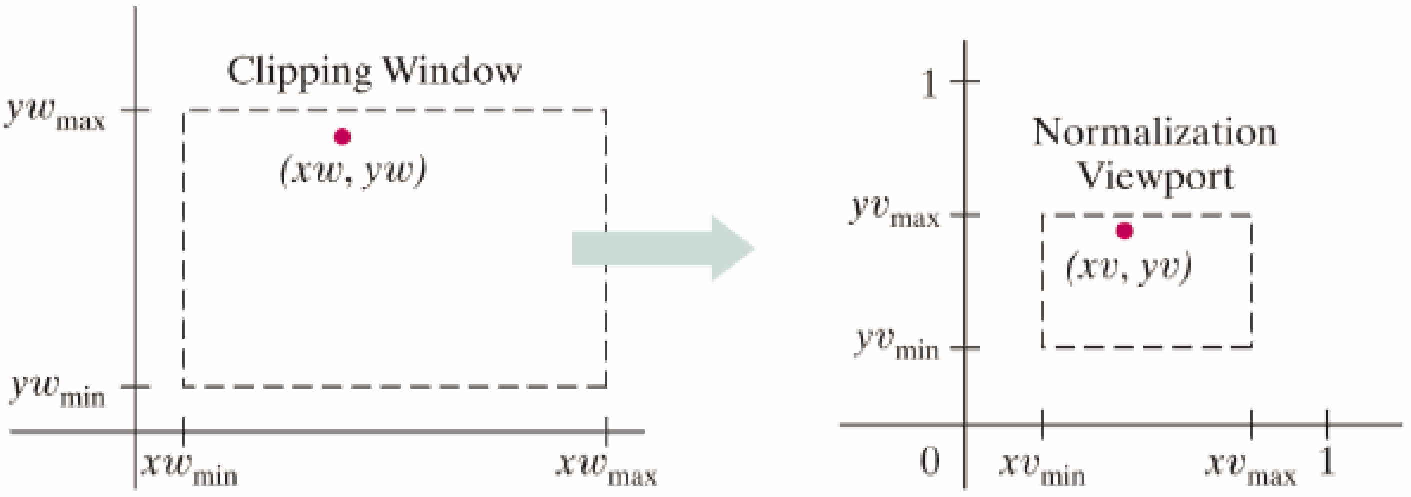


Figure 6-7

A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

Maintaining relative position of points within the two rectangles

- Coordinate transformation: Different sizes and/or height width ratios?

- For any point:

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

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

should hold.

$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}} \qquad \frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

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

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

- This can also be accomplished in 2 steps:
  1. Scale over the fixed point:

     $$\mathbf{S}(xw_{min}, yw_{min}, s_x, s_y)$$

  2. Translate lower-left corner of the clipping window to the lower-left corner of the viewport

     $$\mathbf{T}(xv_{min} - xw_{min}, yv_{min} - yw_{min})$$

Aspect ratio

- Relative proportions of objects are maintained only if the aspect ratio of the viewport is the same as the aspect ratio of the clipping window – i.e. only if the scaling factors *sx* and *sy* are the same
  - Otherwise world objects will be stretched or contracted in x or y directions when displayed

Normalization and viewport transformations

- In other graphics packages normalisation and clipping are applied before window-to-viewport transformation
  - viewport boundaries are specified in screen coordinates relative to the display window position

Transform Viewing Coordinates to Device Coordinates

- Convert object descriptions to normalized coordinates to make the viewing process independent of the requirements of any output device.
- Clip in normalised coordinates, then transfer the scene description to a viewport specified in screen coordinates
- Clipping algorithms in this transformation sequence are now standardised so that objects outside the boundaries $x=\pm1$, $y=\pm1$ are detected and removed from the scene description
- At the final step of the viewing transformation the objects in the viewport are positioned within the display window

Normalization and Viewport Transformation

- World coordinate clipping window
- Normalization square: usually [-1,1]x[-1,1]
- Device coordinate viewport

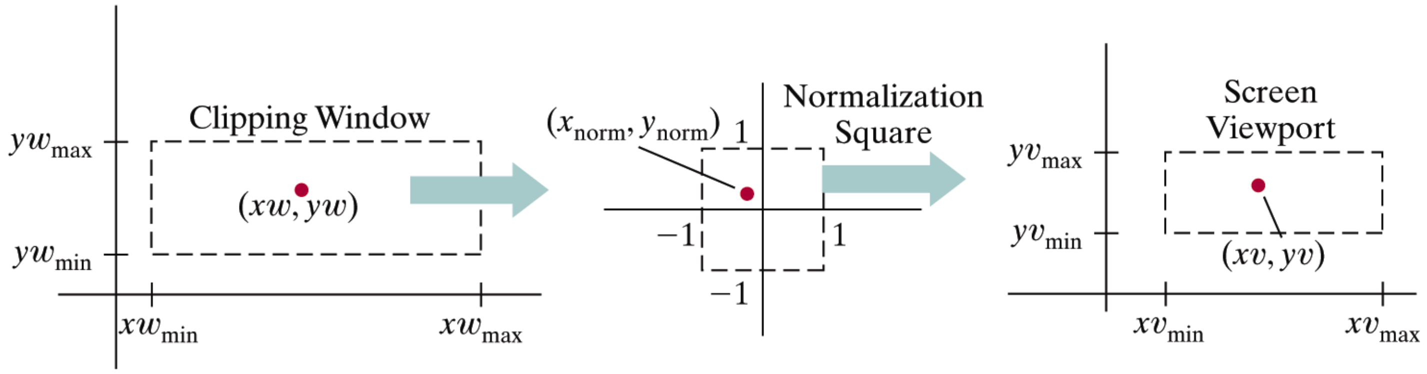


Figure 6-8

A point (xw, yw) in a clipping window is mapped to a normalized coordinate position (x_{norm}, y_{norm}), then to a screen-coordinate position (xv, yv) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates.

OpenGL clipping routines use normalised coordinates in the range -1 to +1

Transform from clipping window into the normalization square

-1 for xv_min and yv_min

+1 for xv_max and yv_max

$$M_{window,normsq} = \begin{bmatrix} \dfrac{2}{xw_{\max} - xw_{\min}} & 0 & -\dfrac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \dfrac{2}{yw_{\max} - yw_{\min}} & -\dfrac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

Transform from normalization square into viewport

-1 for xw_min and yw_min

+1 for xw_max and yw_max

$$\mathrm{M}_{normsq,viewport} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

Aspect ratio

- As in the previous case, relative proportions of objects are maintained only if the aspect ratio of the viewport is the same as the aspect ratio of the clipping window
- If the viewport is mapped to the entire area of the display window and the size of the display window is changed, objects may be distorted unless the aspect ratio of the viewport is also adjusted

OpenGL 2D Viewing Functions

- OpenGL, GLU, and GLUT provide functions to specify clipping windows, viewports, and display windows

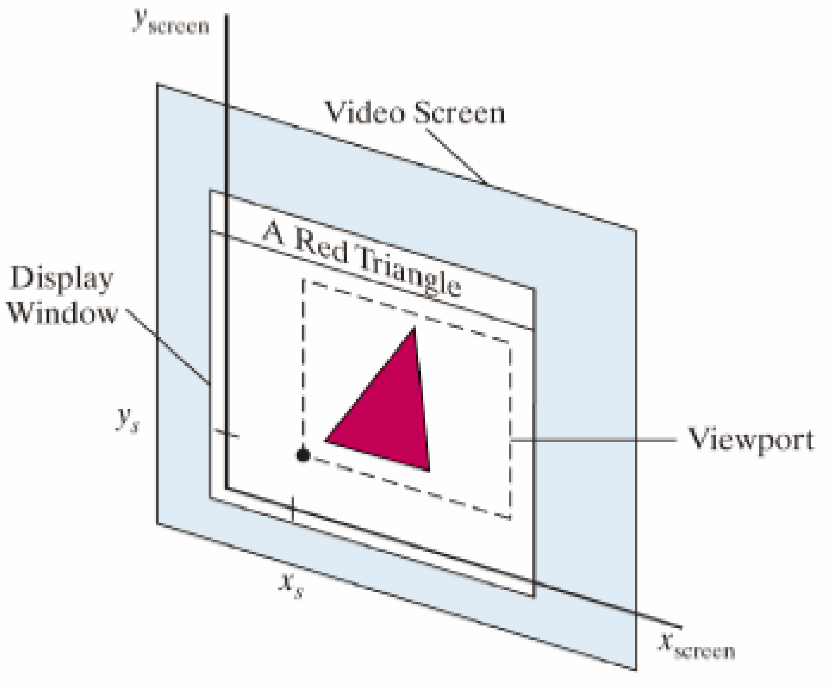


Figure 6-9

A viewport at coordinate position (x_s , y_s) within a display window.

Setting up a 2D Clipping-Window

- glMatrixMode (GL_PROJECTION)
- glLoadIdentity (); // reset, so that new viewing parameters are not combined with old ones (if any)
- gluOrtho2D (xwmin, xwmax, ywmin, ywmax);

or

- glOrtho (xwmin, xwmax, ywmin, ywmax, zwmin, zwmax);

- Objects within the clipping window are transformed to normalized coordinates (-1,1)

Setting up a Viewport

- glViewport (xvmin, yvmin, vpWidth, vpHeight);
- All the parameters are given in integer screen coordinates relative to the lower-left corner of the display window.
- If we do not invoke this function, by default, a viewport with the same size and position of the display window is used (i.e., all of the GLUT window is used for OpenGL display)

Creating a GLUT Display Window

- glutInitWindowPosition (xTopLeft, yTopLeft);
  - the integer parameters are relative to the top-left corner of the screen
- glutInitWindowSize (dwWidth, dwHeight);
- glutCreateWindow (“Title of Display Window”);
- glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB)
  - Specification of the buffer that will be used
- glClearColor (red, green, blue, alpha)
  - Specify the background color

Multiple GLUT windows

- Multiple windows may be created within an OpenGL program
  - Need *window ids* to manage multiple windows
  - windowID = glutCreateWindow(“Window1”);
  - glutDestroyWindow (windowID)

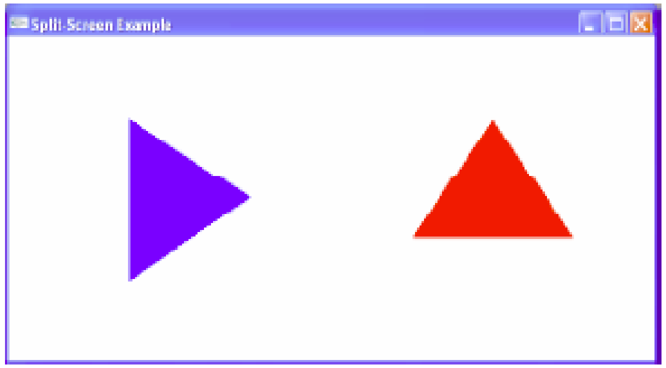
    // to destroy the window
- General functions (like glutInitDisplayMode) are applied to the current display window. We can set the current window to a specific window with:
  - glutSetWindow (windowID);

Other functions

- GLUT provide functions to relocate, resize, minimize, resize to fullscreen, change window title, hide, show, bring to front, or send to back, select a specific cursor for the current display window. (pages 309-311 in the textbook)

OpenGL 2D Viewing Example

- 2 Viewports
- One triangle is displayed in two colors and orientations in 2 viewports



glutInitWindowSize (600, 300);

```
glClear (GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 0.0, 1.0);
drawCenteredTriangle();

glColor3f(1.0, 0.0, 0.0);
glViewport(300, 0, 300, 300);
glRotatef(90.0, 0.0, 0.0, 1.0);
drawCenteredTriangle();
```

Clipping

- Remove portion of output primitives outside clipping window

- Two approaches
  - Clip during scan conversion: check each pixel against clip limits
  - Clip analytically, then scan-convert the modified primitives

Clipping

- Apart from clipping to the view volume, clipping is a basic operation in many other algorithms
  - Breaking space up into chunks
  - 2D drawing and windowing
  - Modelling
- May require more complex geometry than rectangular boxes

Two-Dimensional Clipping

- Point clipping – trivial
- Line clipping
  - Cohen-Sutherland
  - Liang-Barsky
  - Nicholl-Lee-Nicholl
- Fill-area clipping
  - Sutherland-Hodgeman
  - Weiler-Atherton
- Text clipping

Clipping Algorithms

- Clipping: identifying the parts of the objects that will be inside of the window.
- Everything outside the clipping window is eliminated from the scene description (i.e., not scan converted) for efficiency.

- Point clipping:

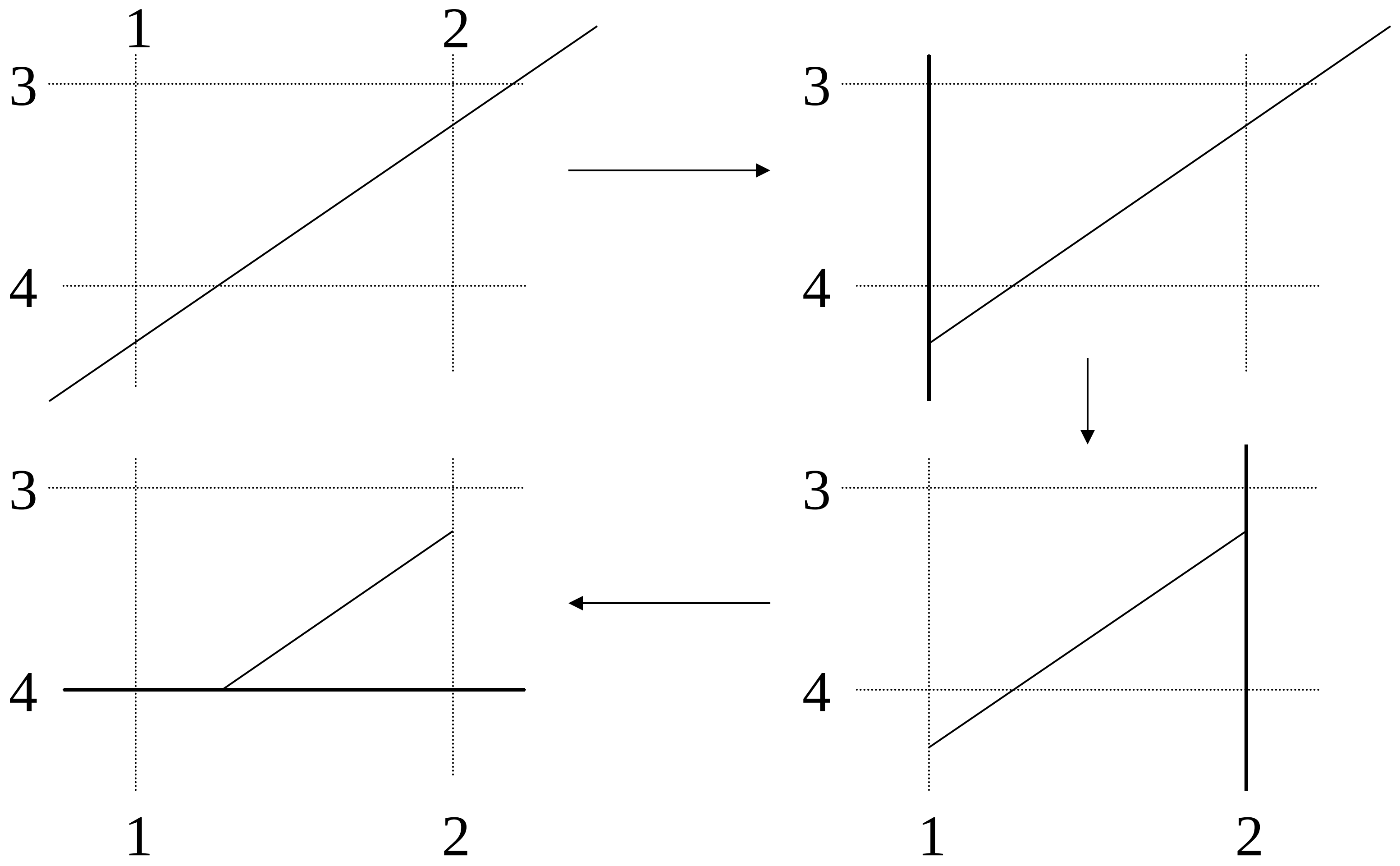
$$xw_{min} \le x \le xw_{max}$$

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

Cohen-Sutherland

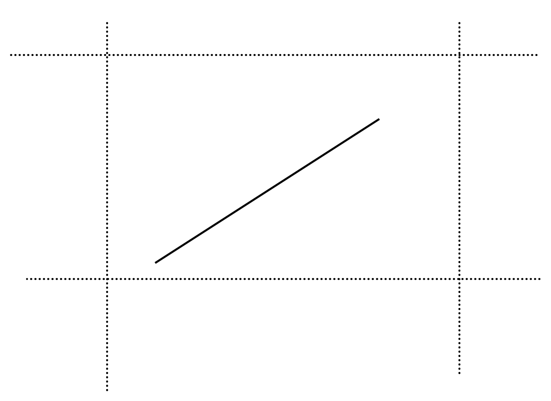
- Clip line against each edge of clip region in turn
  - If both endpoints outside, discard line and stop
  - If both endpoints in, continue to next edge (or finish)
  - If one in, one out, chop line at crossing pt and continue

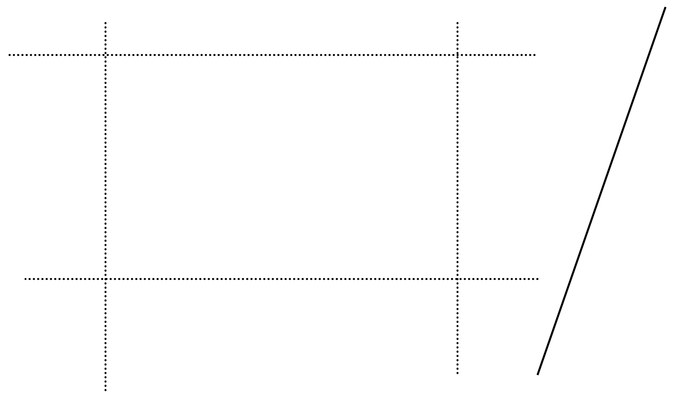
Cohen-Sutherland



Cohen-Sutherland

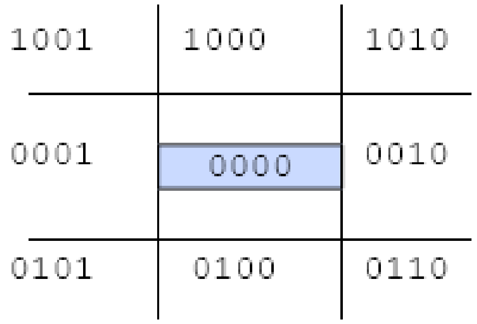
- Some cases lead to early acceptance or rejection
  - If both endpoints are inside all edges
  - If both endpoints are outside one edge





Cohen-Sutherland Line Clipping

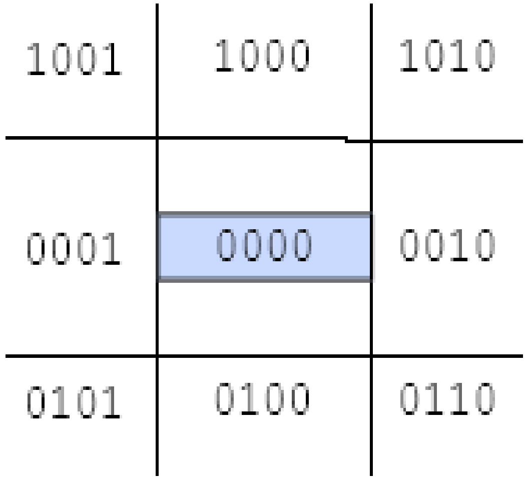
- Based on determination of completely invisible line segments by doing more tests before intersection tests.
- Define test bits for a point:
  bit 1: left of the left border
  bit 2: right of the right border
  bit 3: below the bottom border
  bit 4: above the top border



- When bits (also called *out bits*) are defined for start and end point of the line segment, a single bitwise operation defines the visibility of the line segment. How?

```
•  #define bits(x,y)  ((x<xmin)|(x>xmax)<<1|(y<ymin)<<2|(y>ymax)<<3)

   b1=bits(x1,y1) ; b2=bits(x2,y2);
   if (b1==0 && b2==0) {
   /* both end points inside
      trivial accept         */
   } else if (b1 & b2) {
   /* line is completely outside
      ignore it              */
   } else {
   /* needs further calculation*/
   }
```



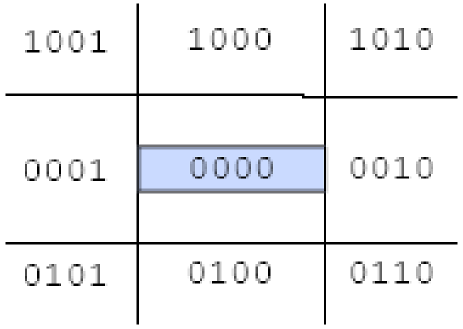
b1 OR b2 = 0 iff both end points inside clipping window. Accept

b1 AND b2 ≠ 0 iff line end points are in the same half-space defined by an edge of the clipping window. Reject

Else subdivide the line into two segments at the point where it crosses a clipping rectangle edge. Reject segment(s) outside the clipping edge

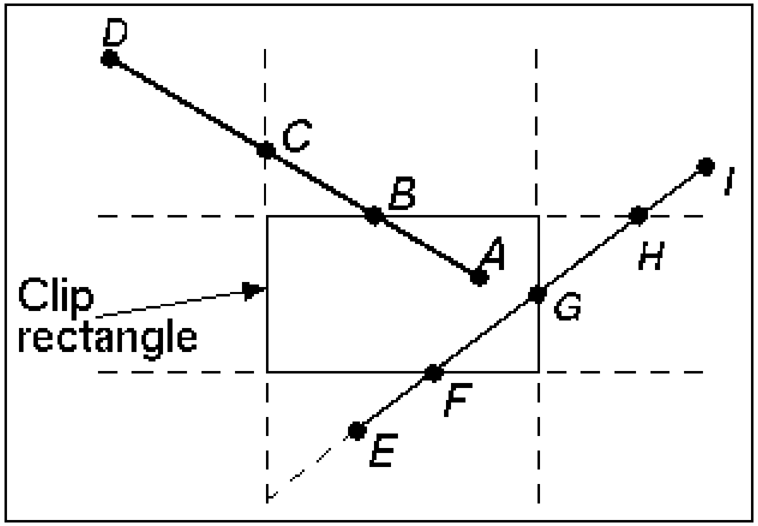
-
  ```
  #define bits(x,y) ((x<xmin)|(x>xmax)<<1|(y<ymin)<<2|(y>ymax)<<3)
  ```

```
b1=bits(x1,y1) ; b2=bits(x2,y2);
if (b1==0 && b2==0) {
/* both end points inside
   trivial accept          */
} else if (b1 & b2) {
/* line is completely outside
   ignore it               */
} else {
/* needs further calculation*/
}
```



- Process clipping boundaries in order and clip a section of the line during the process
- Use the bits calculated before for crossing tests (if one end is 1 and the other is 0 then the line crosses the boundary)
- Use slope-intercept line representation for intersection

The Cohen-Sutherland algorithm



Consider line AD (above). Point A has outcode 0000 and point D has outcode 1001. The line AD cannot be trivially accepted or rejected. D is chosen as the outside point. Its outcode shows that the line cuts the top and left edges (The bits for these edges are different in the two outcodes). Let the order in which the algorithm tests edges be top-to-bottom, left-to-right. The top edge is tested first, and line AD is split into lines DB and BA. Line BA is trivially accepted (both A and B have outcodes of 0000). Line DB is in the outside halfspace of the top edge, and gets rejected.

Cohen-Sutherland Algorithm

```
$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$
repeat    for $border = \{\text{LEFT}, \text{RIGHT}, \text{BOTTOM}, \text{TOP}\}$
    if $\neg(bits(x_1, y_1) \vee bits(x_2, y_2))$
        Both inside, accept line and terminate
    else if $bits(x_1, y_1) \wedge bits(x_2, y_2)$
        Both outside same region, reject line and terminate
    if $border = \text{LEFT} \vee border = \text{RIGHT}$
        $y_p = y_1 + m(x_{border} - x_1), x_p = x_{border}$
    else
        $x_p = x_1 + \frac{(y_{border} - y_1)}{m}, y_p = y_{border}$
    if $bordertest(x_1, y_1)$
        $x_1 = x_p; y_1 = y_p$
    else if $bordertest(x_2, y_2)$
        $x_2 = x_p; y_2 = y_p$
```

update line endpoint coordinates

Example

$bits(0,0) = 0101 \quad bits(8,5) = 0010$

$bits(0,0) \wedge bits(8,5) = 0000$

$m = \frac{(5-0)}{(8-0)} = \frac{5}{8}$

LEFT : $y_p = y_1 + \frac{5}{8}(1-0) = \frac{5}{8}$

$\text{LEFT}(P_1) \rightarrow P_1' = (1, 5/8)$

RIGHT : $y_p = y_1 + \frac{5}{8}(7-1) = \frac{5}{8} + \frac{5}{8}6 = \frac{35}{8}$

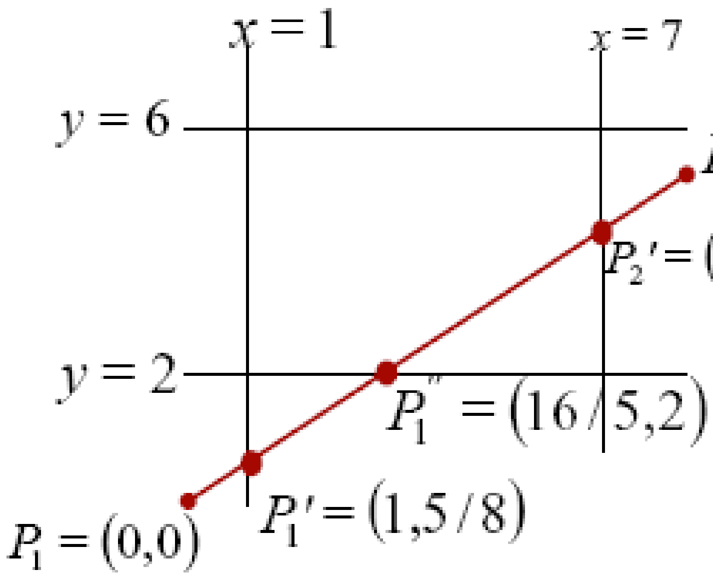
$\text{RIGHT}(P_2) \rightarrow P_2' = (7, 35/8)$

BOTTOM : $x_p = 1 + \left(2 - \frac{5}{8}\right)\frac{8}{5} = 1 + \frac{16}{5} - 1 = \frac{16}{5}$

$\text{BOTTOM}(P_1') \rightarrow P_1'' = (16/5, 2)$

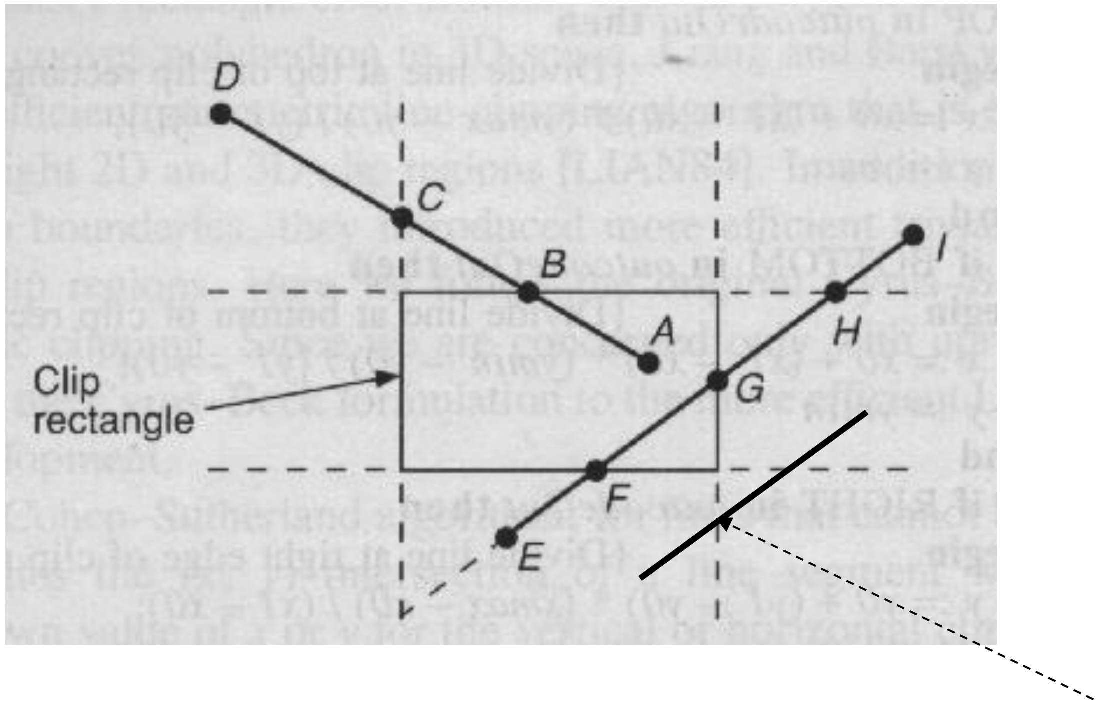
TOP : P_1'' inside, P_2' inside, so terminate

$L = (16/5, 2)$ to $(7, 35/8)$



Cohen-Sutherland Line Clipping

- Fixed order testing and clipping cause needless clipping (external intersections)



Extra clipping here

Cohen-Sutherland Line Clipping

- This algorithm can be very efficient if it can accept and reject primitives trivially
  - If clip window is large wrt scene data
    - Most primitives are accepted trivially
  - If clip window is much smaller than scene data
    - Most primitives are rejected trivially
- Good for hardware implementation

Liang-Barsky Line Clipping

Clipping: Overview of Steps

- Express line segments in parametric form
- Derive equations for testing if a point is inside the window
- Compute new parameter values for visible portion of line segment, if any
- Display visible portion of line segment

- The relative speed improvement over Sutherland-Cohen algorithm is as follows:
- 36% for 2D lines
  40% for 3D lines
  70% for 4D lines

Liang-Barsky Clipping

- Parametric clipping - view line in parametric form and reason about the parameter values
- More efficient, as not computing the coordinate values at irrelevant vertices
- Clipping conditions on parameter: Line is inside clip region for values of *t* such that:

$$x_{\min} \le x_1 + t\Delta x \le x_{\max} \qquad \Delta x = x_2 - x_1$$

$$y_{\min} \le y_1 + t\Delta y \le y_{\max} \qquad \Delta y = y_2 - y_1$$

Liang-Barsky (2)

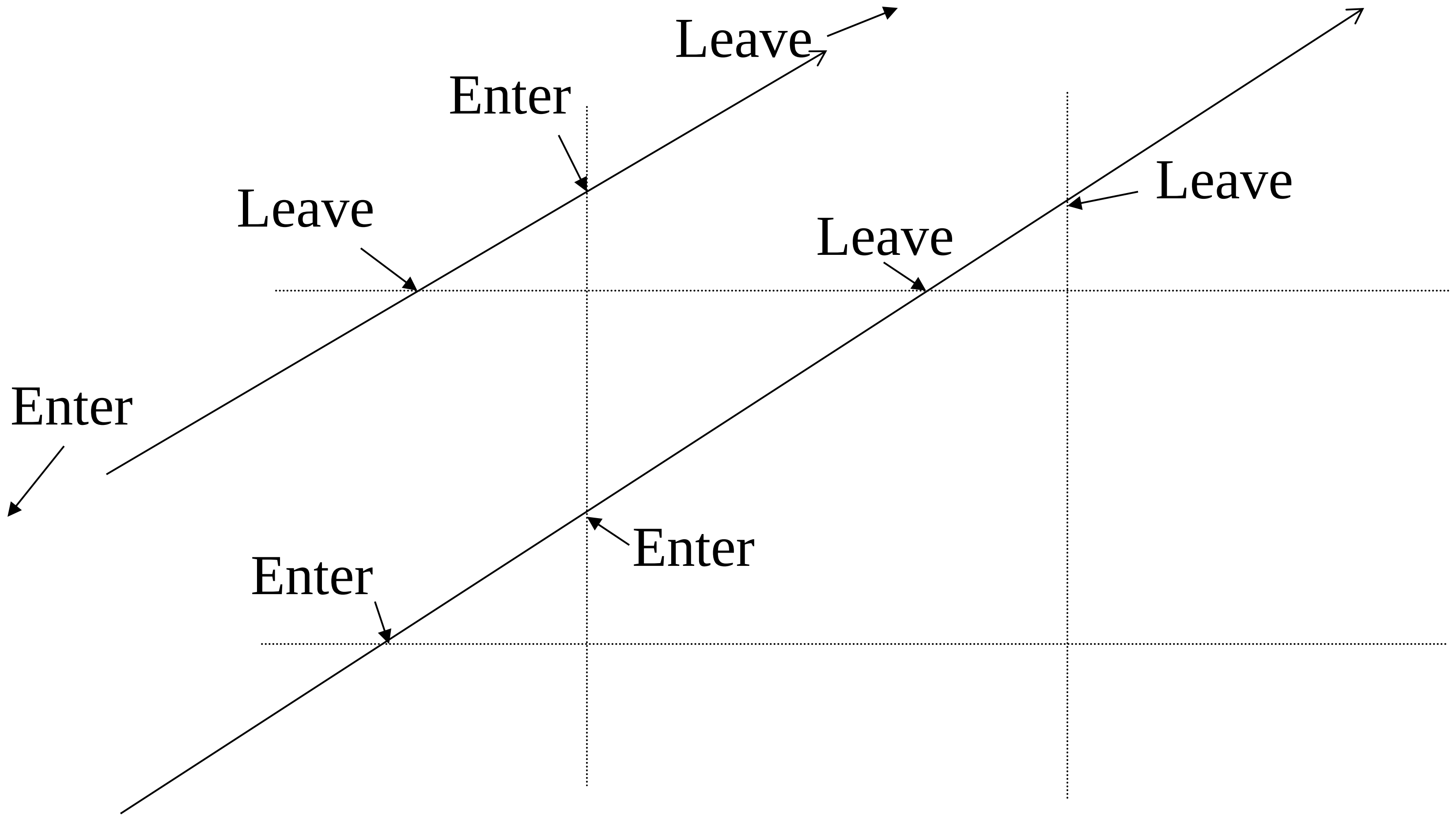
- Infinite line intersects clip region edges when:

$$t_k = \frac{q_k}{p_k} \qquad \text{where} \qquad \begin{array}{ll} p_1 = -\Delta x & q_1 = x_1 - x_{\min} \\ p_2 = \Delta x & q_2 = x_{\max} - x_1 \\ p_3 = -\Delta y & q_3 = y_1 - y_{\min} \\ p_4 = \Delta y & q_4 = y_{\max} - y_1 \end{array}$$

Liang-Barsky (3)

- When p_k<0, as t increases line goes from outside to inside - enter
- When p_k>0, line goes from inside to outside - leave
- When p_k=0, line is parallel to an edge (clipping is easy)
- If there is a segment of the line inside the clip region, sequence of infinite line intersections must go: enter, enter, leave, leave

Liang-Barsky (4)

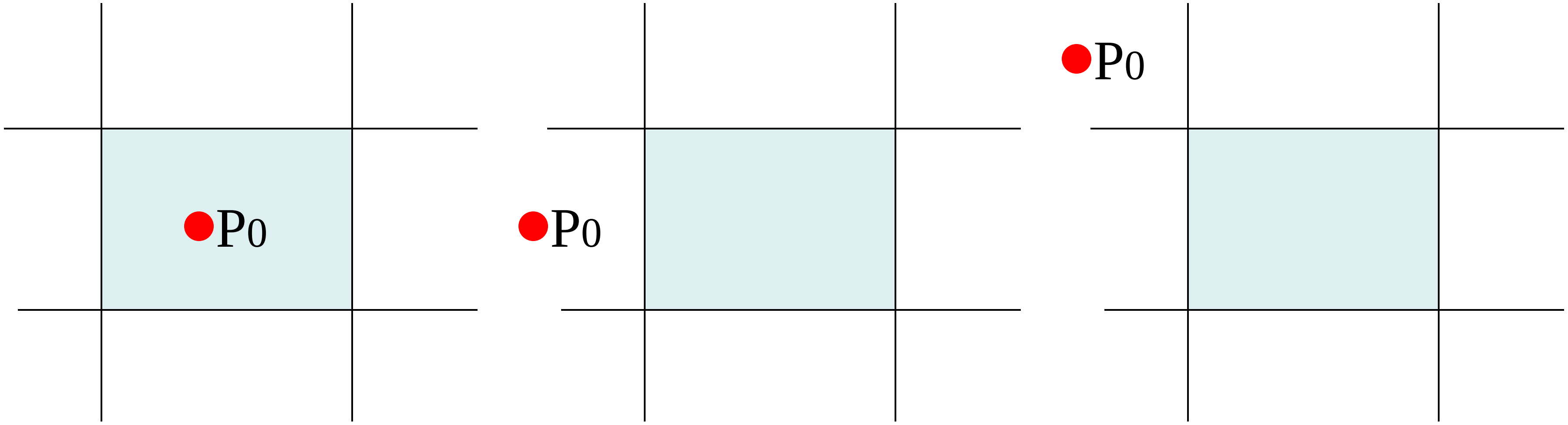


Liang-Barsky - Algorithm

- Compute entering *t* values, which are q_k/p_k for each $p_k<0$
- Compute leaving *t* values, which are q_k/p_k for each $p_k>0$
- Parameter value for small *t* end of line is:t_{small}= max(0, entering *t*’s)
- parameter value for large t end of line is: t_{large}=min(1, leaving *t*’s)
- if $t_{small}<t_{large}$, there is a line segment - compute endpoints by substituting *t* values

Nicholl-Lee-Nicholl Line Clipping

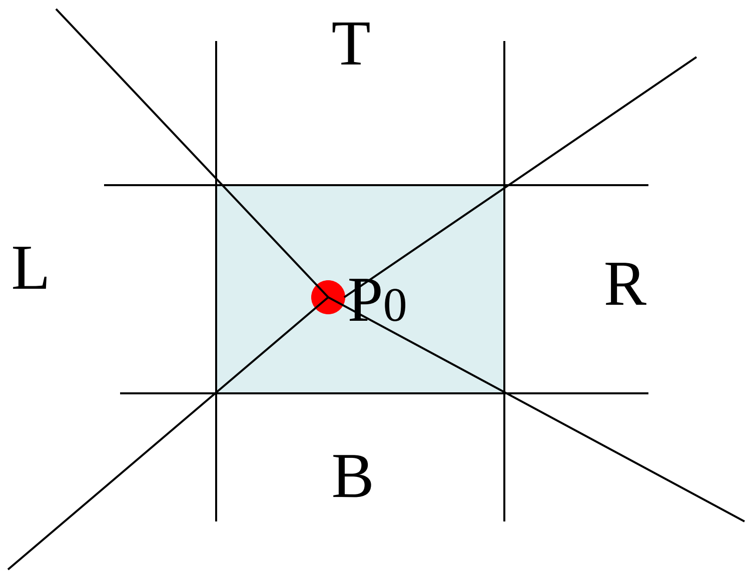
- Creates more testing regions around the clipping window
  - Avoids multiple line-intersection calculations
- Initial testing to determine if a line segment is completely inside the clipping window can be done using previous methods
- If trivial acceptance or rejection is not possible the NLN algorithm sets up additional regions



For line with endpoints P_0P_{end}, there are three different positions to consider - all others can be derived from these by symmetry considerations

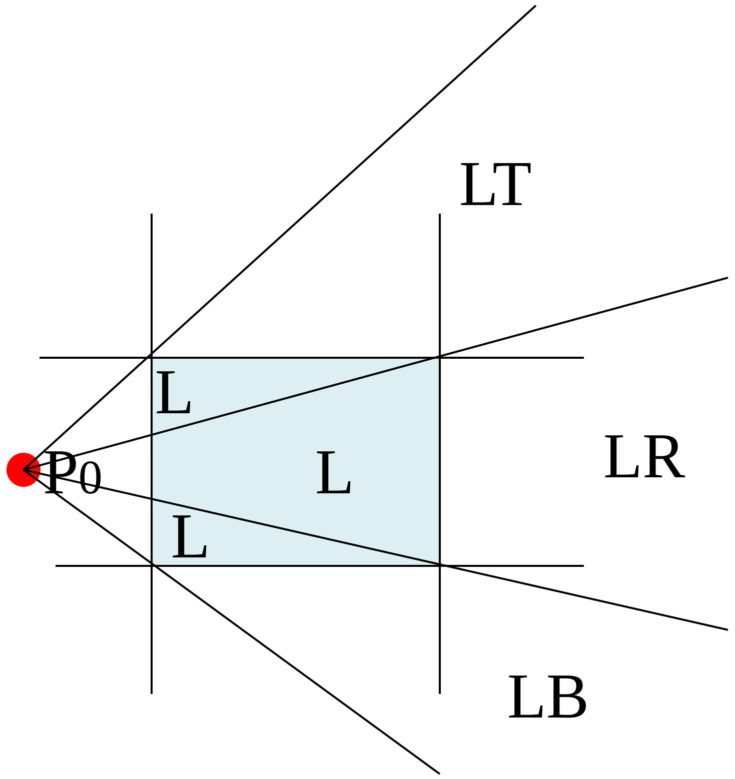
For each case, we generate specialized test regions for other endpoint P_{end}, which use simple tests (slope, >, <), and tells us which edges to clip against.

Case 1



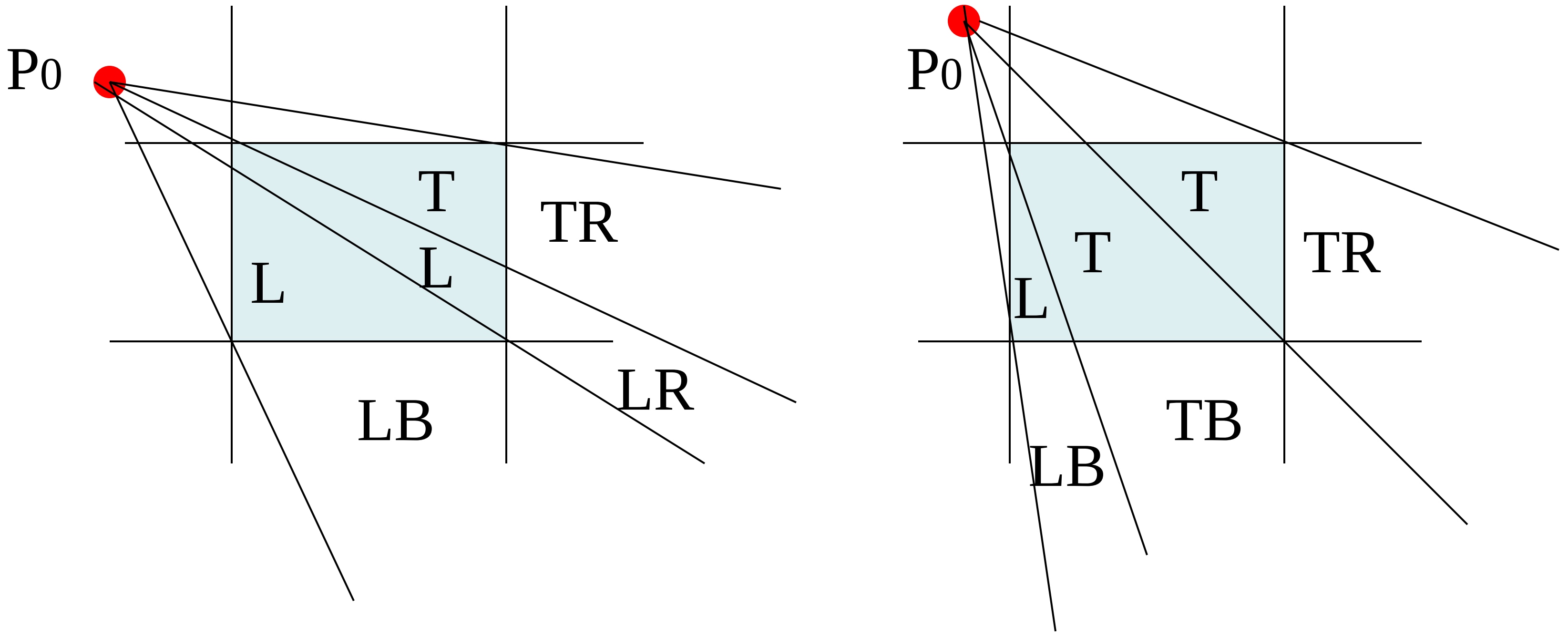
Find which of the four regions P_{end} lies in, then calculate the line intersection with the corresponding boundary

Case 2



Find which of the four regions P_{end} lies in, then calculate the line intersection with the corresponding boundary

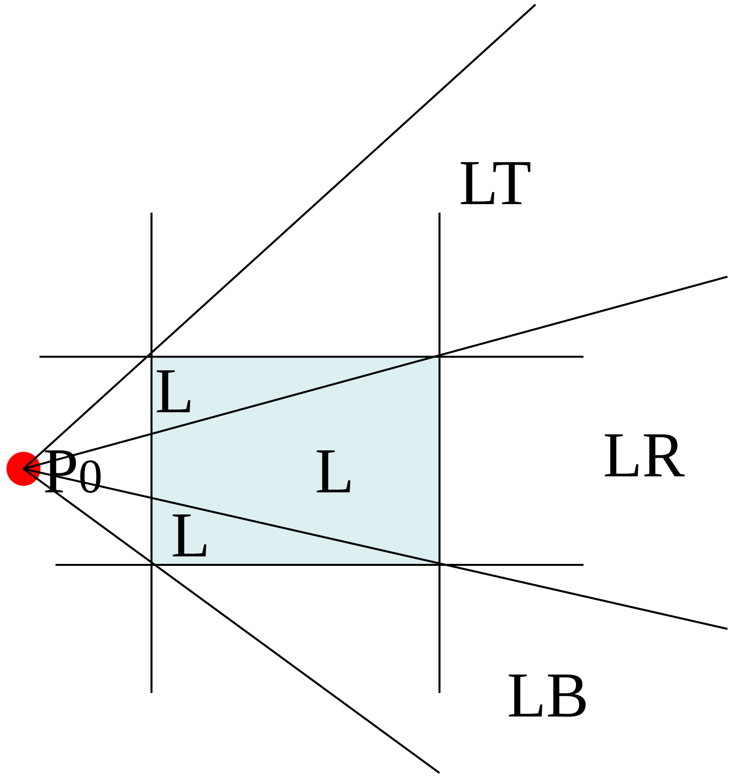
Case 3 : 2 possibilities



Find which of the five regions P_{end} lies in, then calculate the line intersection with the corresponding boundary

N-L-N Line clipping

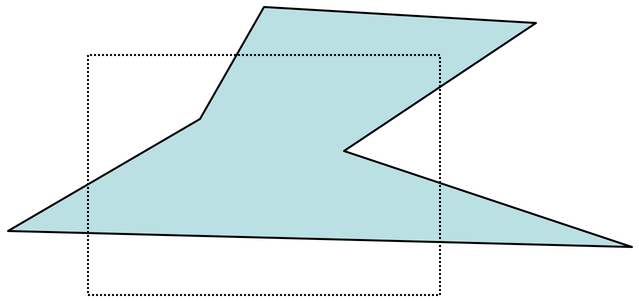
- To determine in which region P_{end} lies we compare the slope of $P_{end}P_o$ to the slopes of the boundaries of the NLN regions

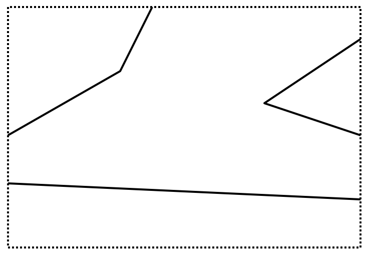


Number of cases explodes in 3D, making algorithm unsuitable

Polygon clipping

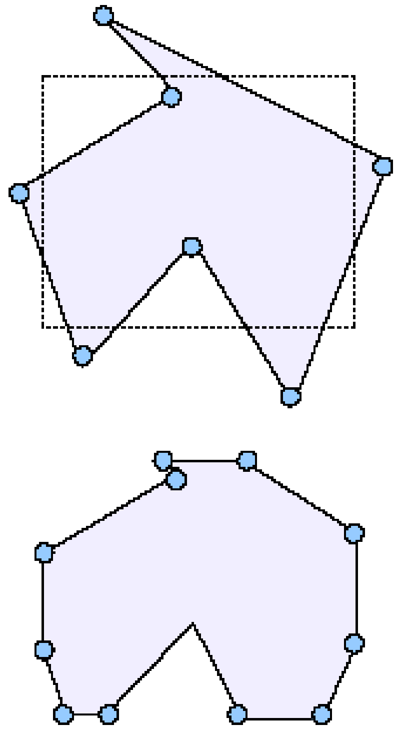
- Clipping a polygon fill area needs more than line-clipping of the polygon edges
  - would produce and unconnected set of lines
- Must generate one or more closed polylines, which can be filled with the assigned colour or pattern





Polygon Clipping

- Find the vertices of the new polygon(s) inside the window.
- Sutherland-Hodgeman Polygon Clipping: Check each edge of the polygon against all window boundaries. Modify the vertices based on transitions. Transfer the new edges to the next clipping boundary.



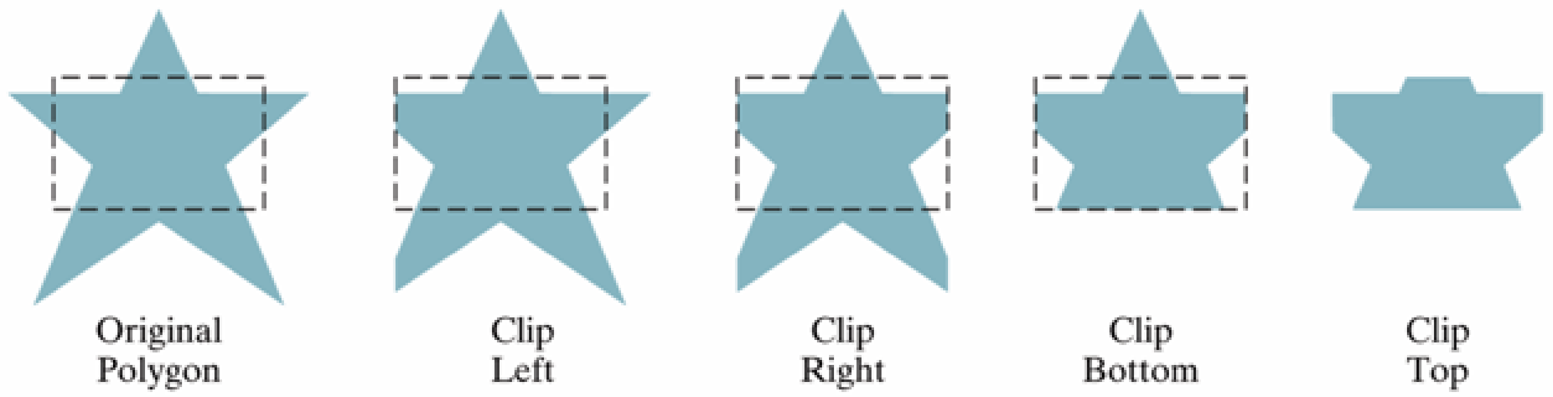
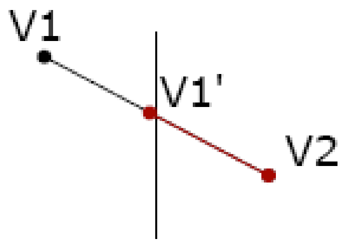


Figure 6-23

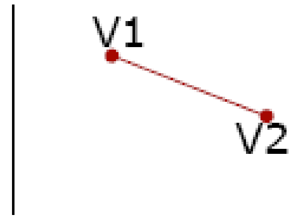
Processing a polygon fill area against successive clipping-window boundaries.

Sutherland-Hodgeman Polygon Clipping

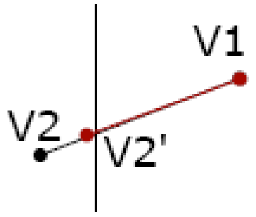
- Traverse edges for borders; 4 cases:
  - V1 outside, V2 inside: take V1' and V2



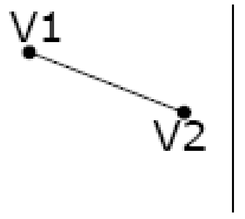
  - V1 inside, V2 inside: take V1 and V2

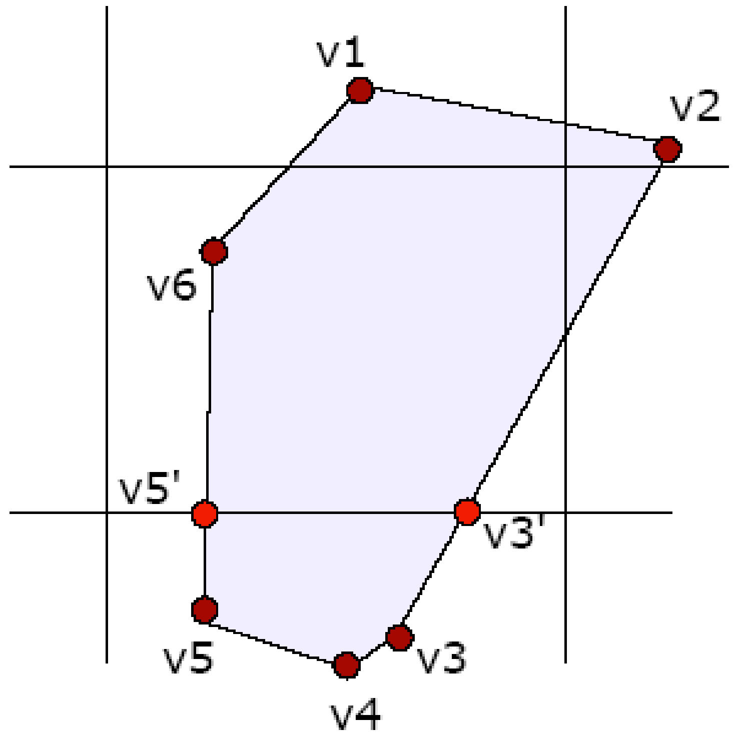


  - V1 inside, V2 outside: take V1 and V2'



  - V1 outside, V2 outside: take none





- Left border:

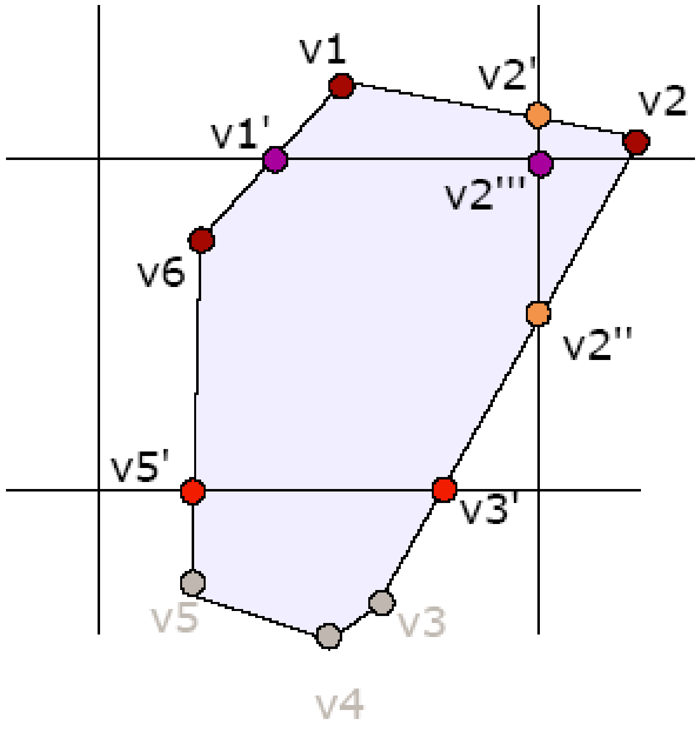
v1 v2	both inside	v1 v2
v2 v3	both inside	v2 v3
.....	" "	....

v1,v2,v3,v4v5,v6,v1

- Bottom Border:

v1 v2	both inside	v1 v2
v2 v3	v2 i, v3 o	v2 v3'
v3 v4	both outside	none
v4 v5	both outside	none
v5 v6	v5 o, v6 i	v5' v6
v6 v1	both inside	v6 v1

v1,v2,v3',v5',v6,v1



- v1,v2,v3',v5',v6,v1
- Right border:

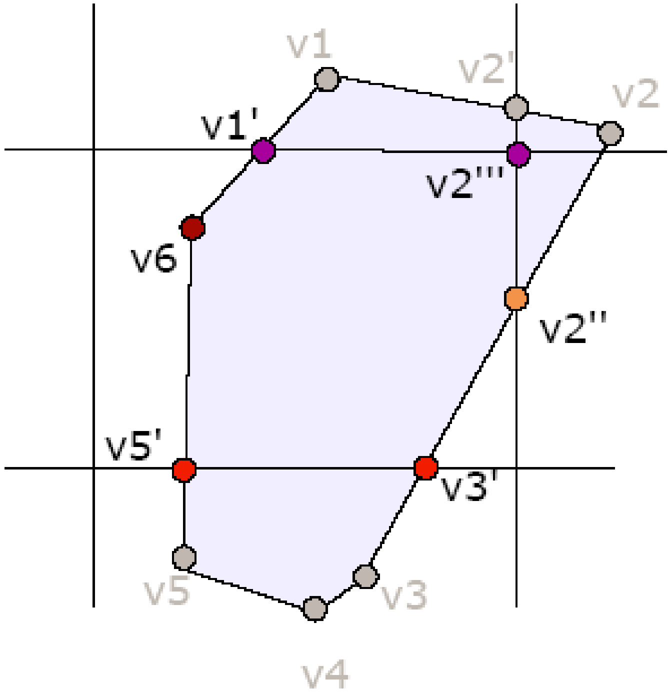
v1 v2	v1 i, v2 o	v1 v2'
v2 v3'	v2 o, v3'i	v2'' v3'
v3' v5'	both inside	v3' v5'
v5' v6	both inside	v5' v6
v6 v1	both inside	v6 v1

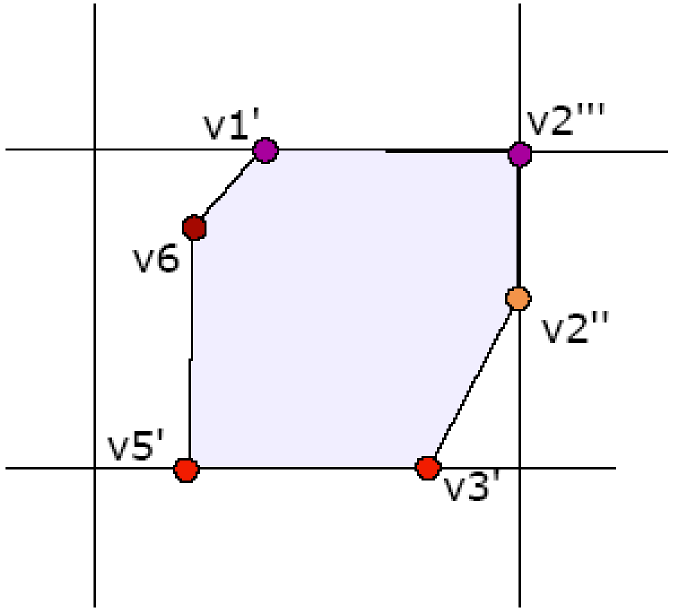
  v1,v2',v2'',v3',v5',v6,v1

- Top Border:

v1 v2'	both outside	none
v2' v2''	v2' o, v2'' i	v2''' v2''
v2'' v3'	both inside	v2'' v3'
v3' v5'	both inside	v3' v5'
v5' v6	both inside	v5' v6
v6 v1	v6 i, v1 o	v6 v1'

  v2''',v2'',v3',v5',v6,v1'


v1
v2'
v2
v1'
v2'''
v6
v2''
v5'
v3'
v5
v3
v4


v1'
v2'''
v6
v2''
v5'
v3'

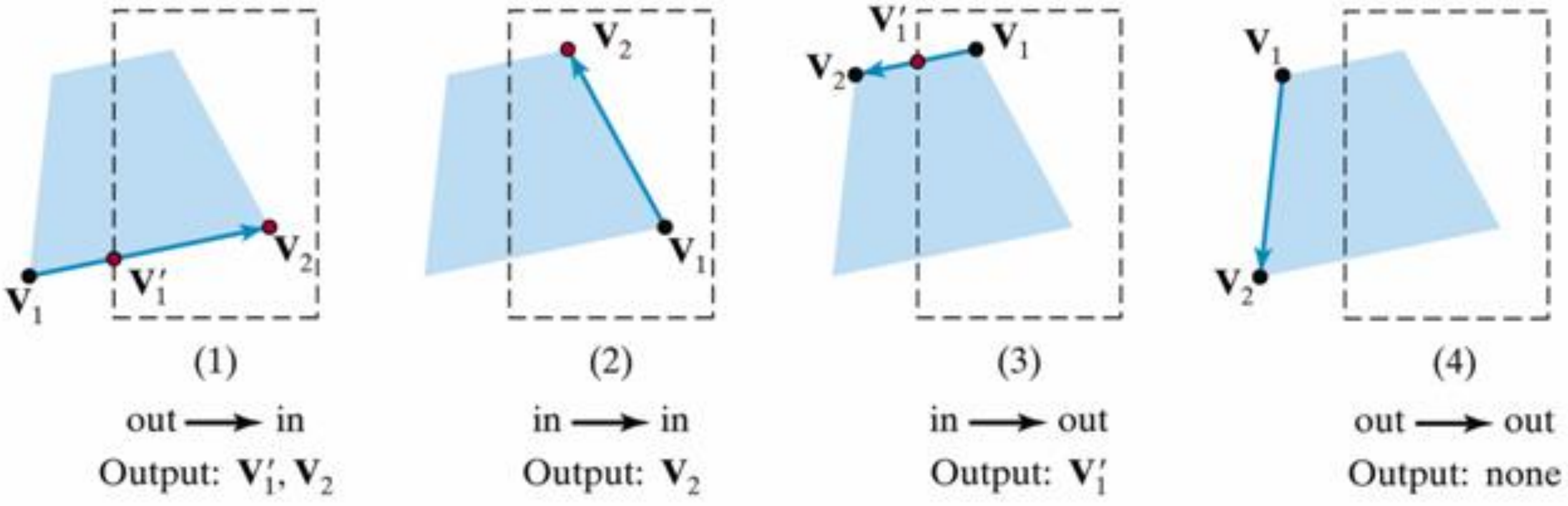
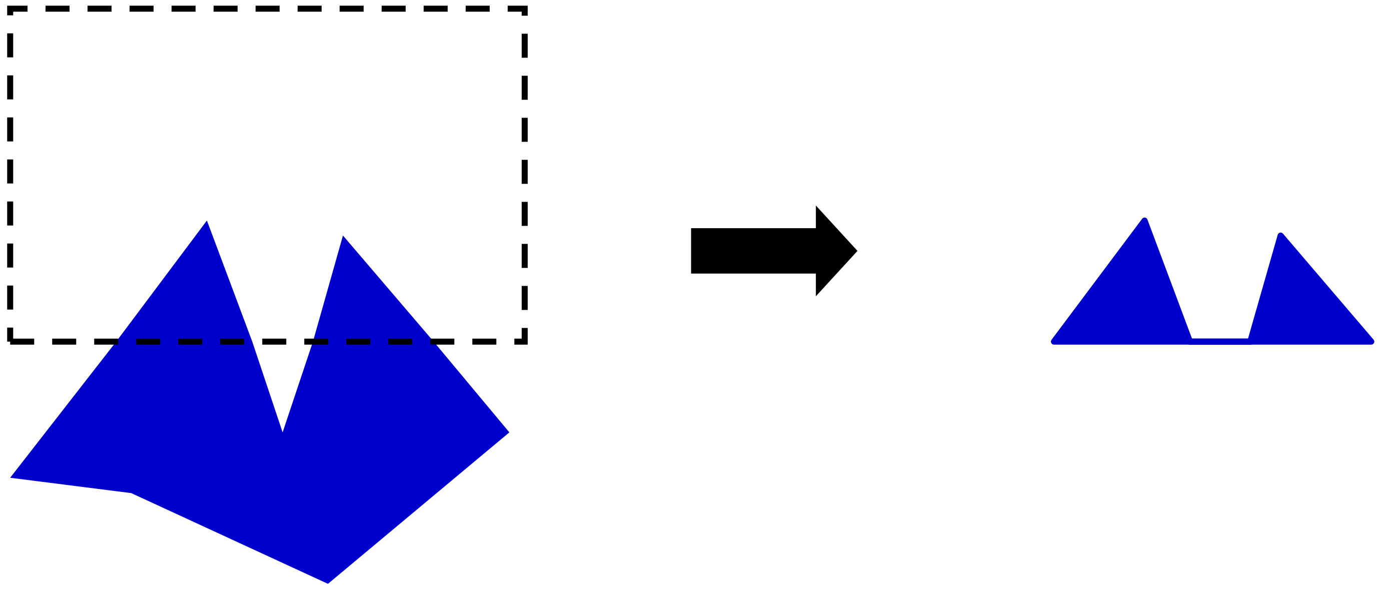


Figure 6-26

The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

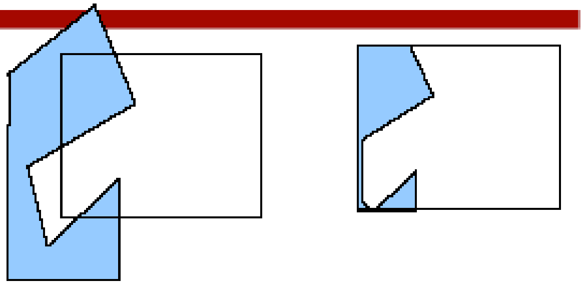
Sutherland-Hodgman Polygon Clipping

- The algorithm correctly clips convex polygons, but may display extraneous lines for concave polygons



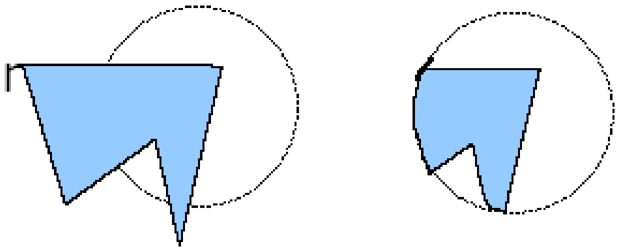
Other Issues in Clipping

- Problem in Sutherland-Hodges. Weiler-Atherton has a solution

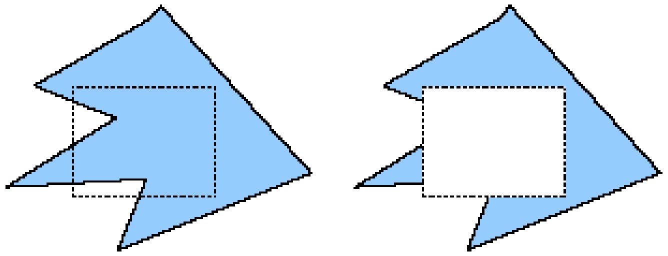


- Clipping other shapes: Circle, Ellipse, Curves.

- Clipping a shape against another shape



- Clipping the exteriors.



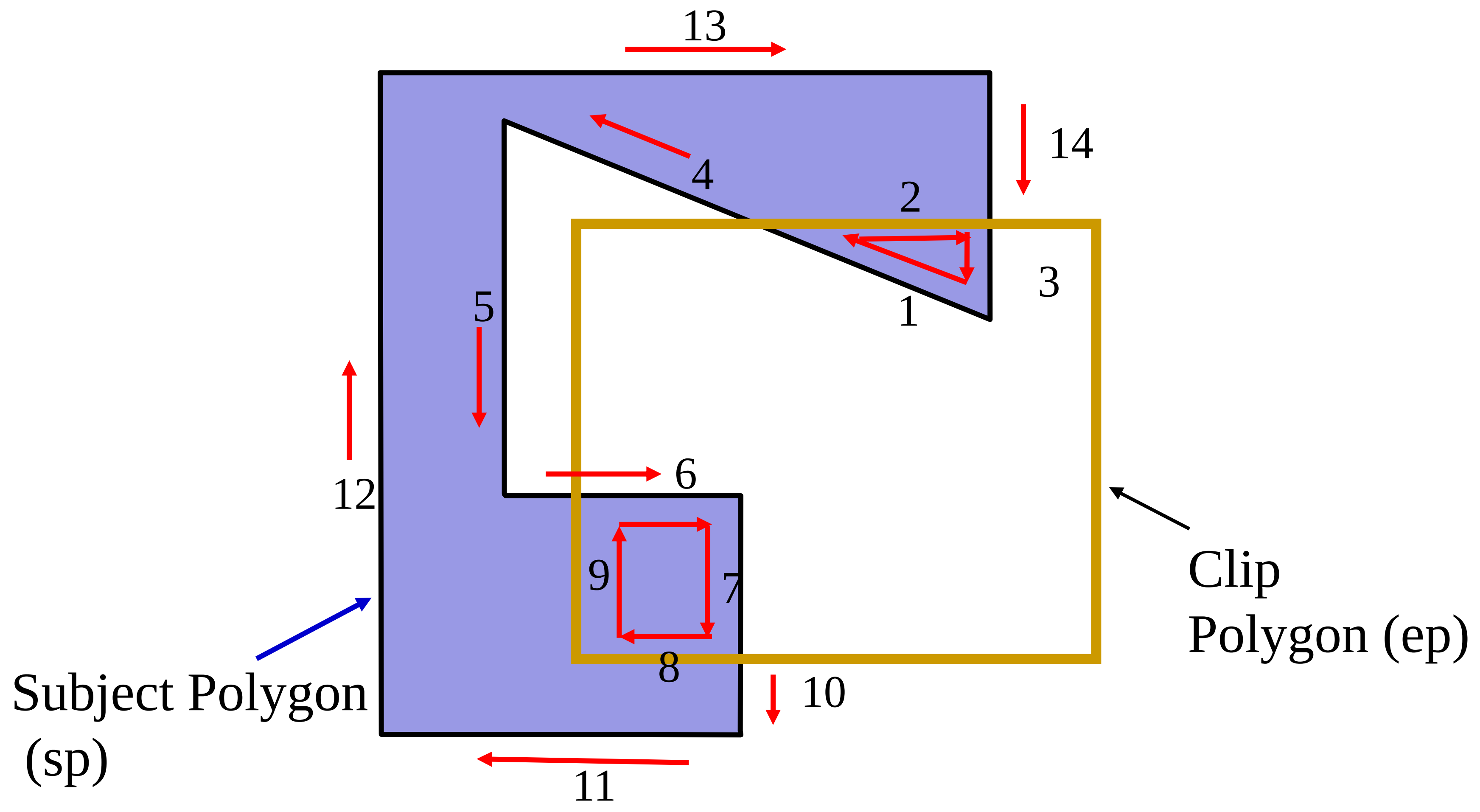
Weiler-Atherton Polygon Clipping

- Another approach to polygon clipping
- No extra clipping outside window
- Works for arbitrary shapes
- Avoids degenerate polygons

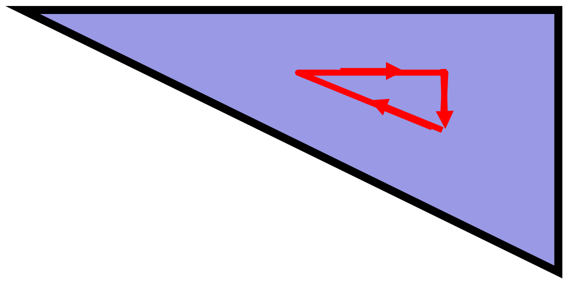
Outline of Weiler algorithm:

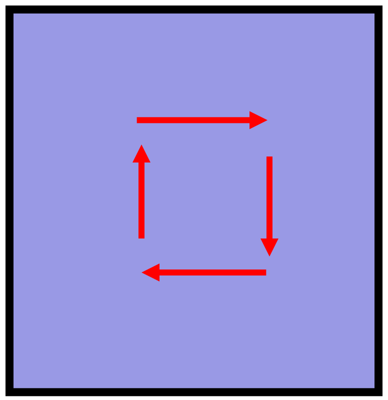
- – Replace crossing points with vertices
- – Form linked lists of edges
- – Change links at vertices
- – Enumerate polygon patches

Weiler-Atherton Clipping
clockwise orientation of subject polygon



Gives “Right” Answer





Weiler-Atherton Clipping (clockwise orientation of polygon)

- Start at first (inside) vertex
- Traverse polygon until hitting a window boundary
- Output intersection point *i*
- Turn *right*
- Follow window boundary until next intersection

Weiler-Atherton Clipping

- Output second intersection
- Turn *right,* again, and follow subject polygon until closed
- Continue on subject polygon from first intersection point.
- Repeat processing until complete

Generalizations of W-A

- Can be extended to complex situations, arbitrary windows
- Stability issues can arise for such cases

Weiler-Atherton Polygon Clipping
counter-clockwise orientation of subject polygon

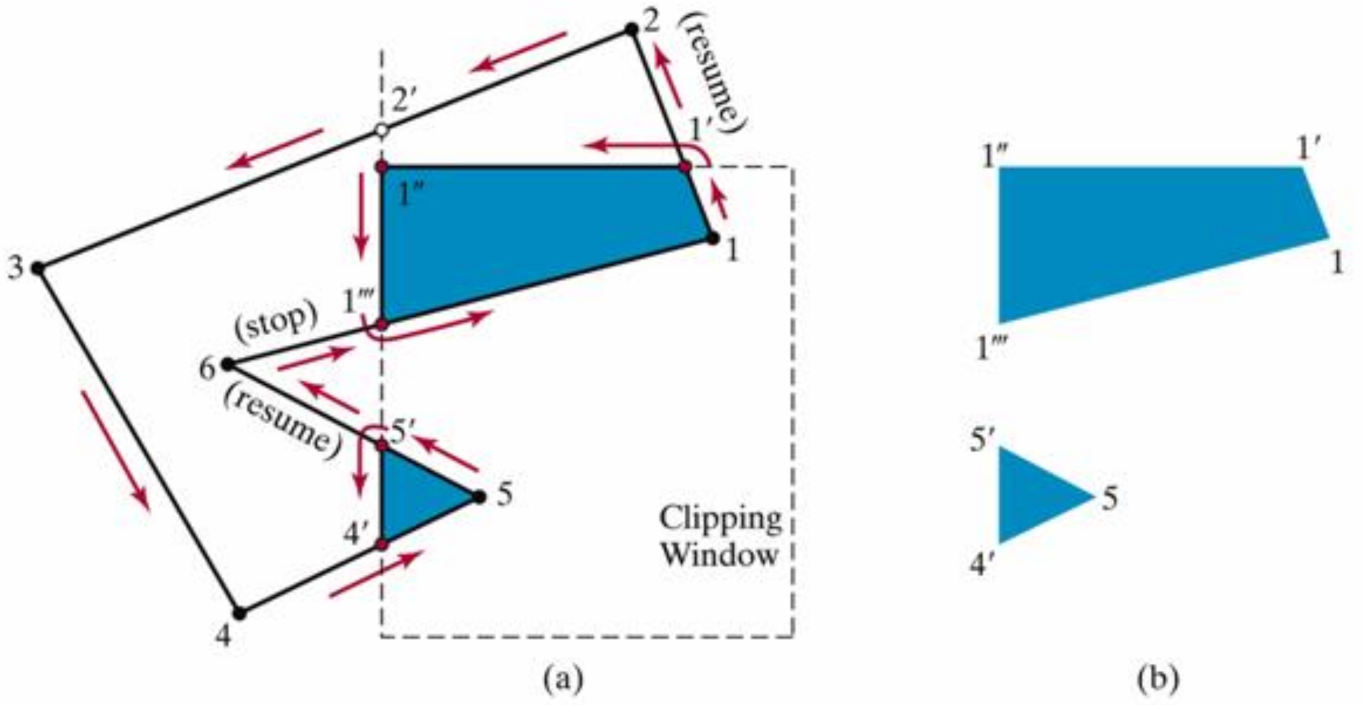
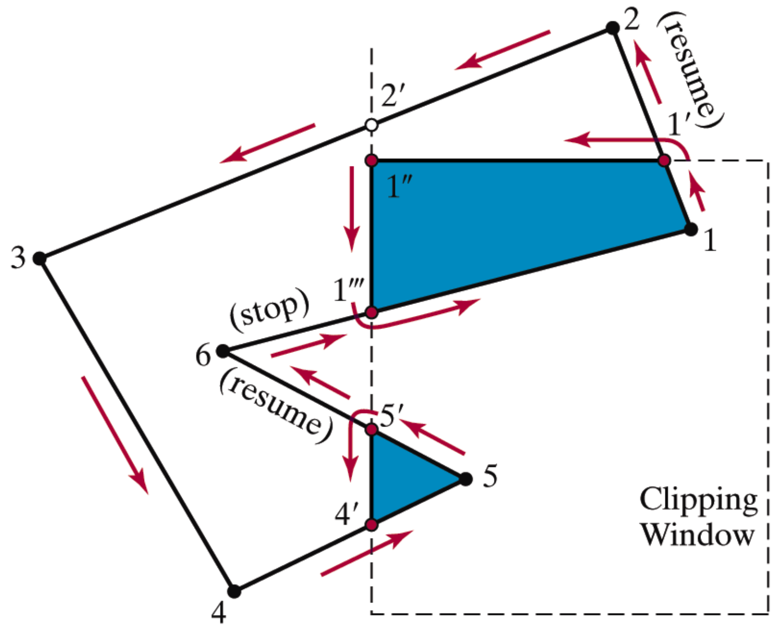


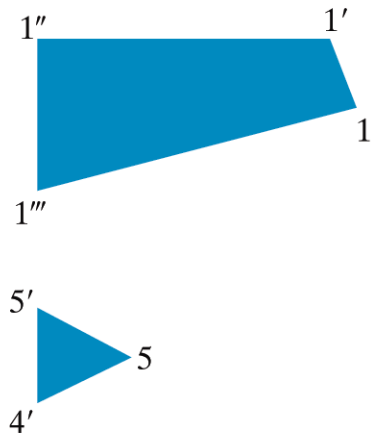
Figure 6-29

A concave polygon (a), defined with the vertex list {1, 2, 3, 4, 5, 6 }, is clipped using the Weiler-Atherton algorithm to generate the two lists {1, 1′,1″,1‴} and {4′,5,5′}, which represent the separate polygon fill areas shown in (b).

Weiler-Atherton Polygon Clipping
counter-clockwise orientation of subject polygon

- For an outside-to-inside pair of vertices, follow the polygon boundary
- For an inside-to-outside pair of vertices, follow the window boundary in a counter-clockwise direction





Weiler-Atherton Polygon Clipping

- Polygon clipping using nonrectangular polygon clip windows

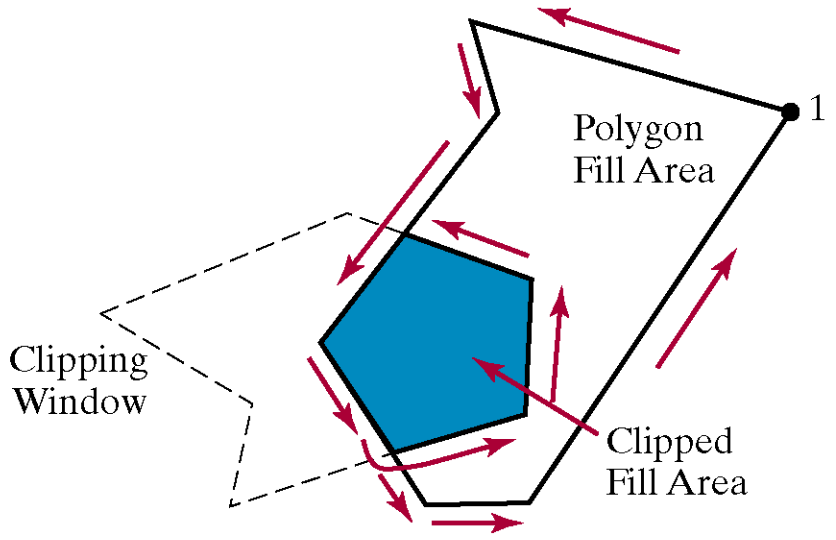
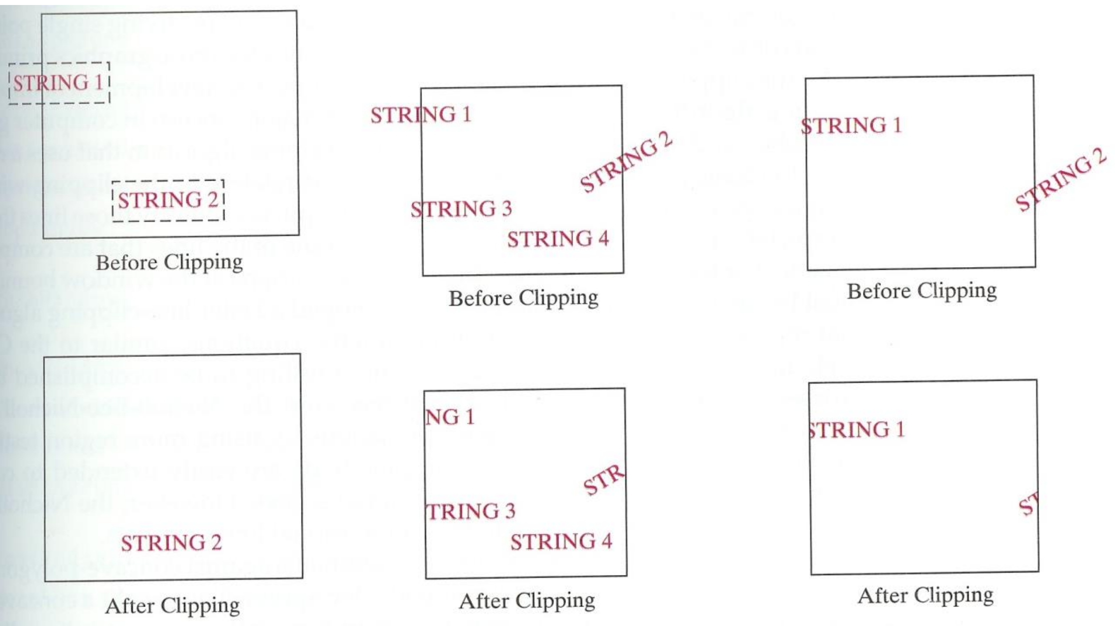


Figure 6-30

Clipping a polygon fill area against a concave-polygon clipping window using the Weiler-Atherton algorithm.

Text Clipping

- All-or-none text clipping
  - Using boundary box for the entire text
- All-or-non character clipping
  - Using boundary box for each individual character
- Character clipping
  - Vector font: Clip boundary polygons or curves
  - Bitmap font: Clip individual pixels


STRING 1
STRING 2
Before Clipping
STRING 2
After Clipping
STRING 1
STRING 2
STRING 3
STRING 4
Before Clipping
NG 1
STR
TRING 3
STRING 4
After Clipping
STRING 1
STRING 2
Before Clipping
STRING 1
ST
After Clipping