Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT205 Computer Graphics

Clipping

Lecture 11

2022-23

Yong Yue

Topics for today

- Concepts
  - What is clipping?
  - Why is clipping important?
  - Clipping of points, lines and polygons
- Line clipping algorithms
  - Brute Force Simultaneous Equations
  - Brute Force Similar Triangles
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon clipping
- OpenGL functions

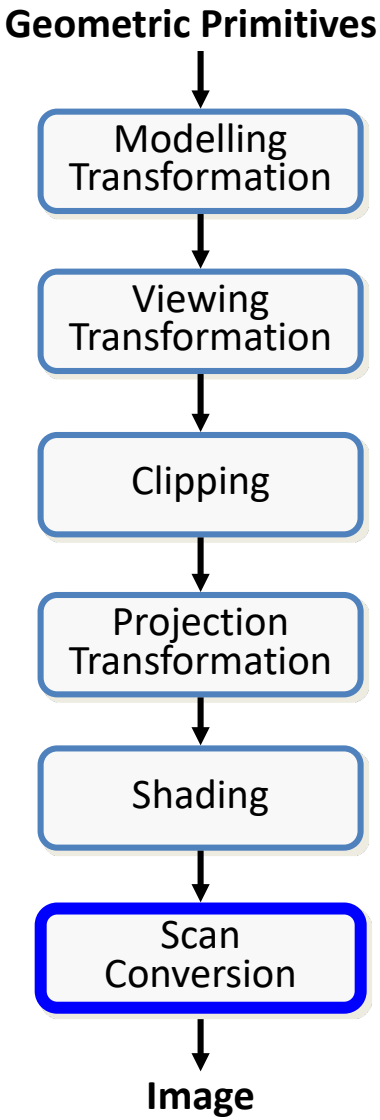
Clipping and rasterization

- Clipping – Remove objects or parts of objects that are outside the clipping window.
- Rasterization (scan conversion) – Convert high level object descriptions to pixel colours in the framebuffer.
- Graphics systems (e.g. OpenGL) do these for us – no explicit OpenGL functions needed for doing clipping and rasterization.

Why clipping

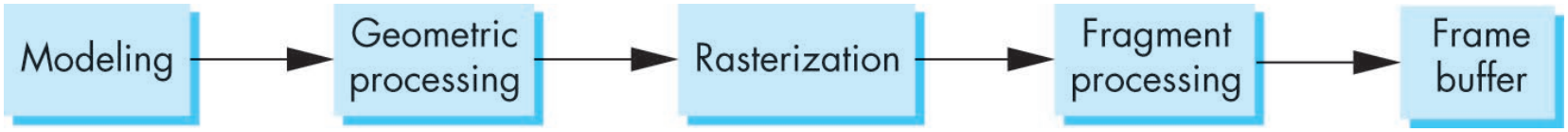
- Rasterization is very expensive
  - Approximately linear with the number of fragments created
  - Math and logic *per pixel*
- If we only rasterize what is actually viewable, we can save a lot of expensive computation
  - A few operations now can save many later

Rendering pipeline



Required tasks

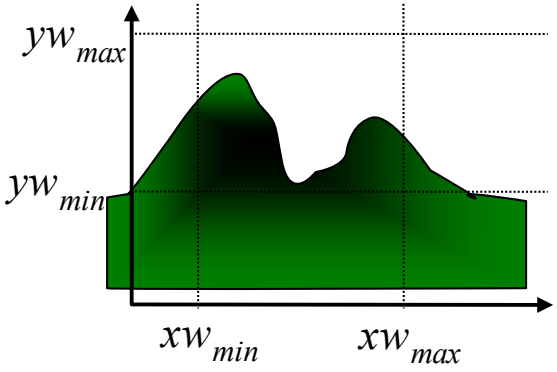
- Clipping
- Transformations
- Rasterization (scan conversion)
- Some tasks deferred until fragment processing
  - Hidden surface removal
  - Antialiasing

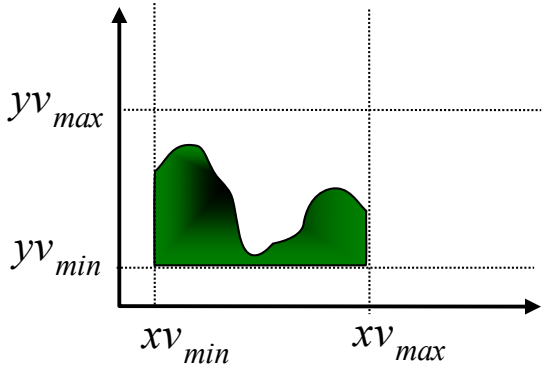


Rasterization meta algorithms

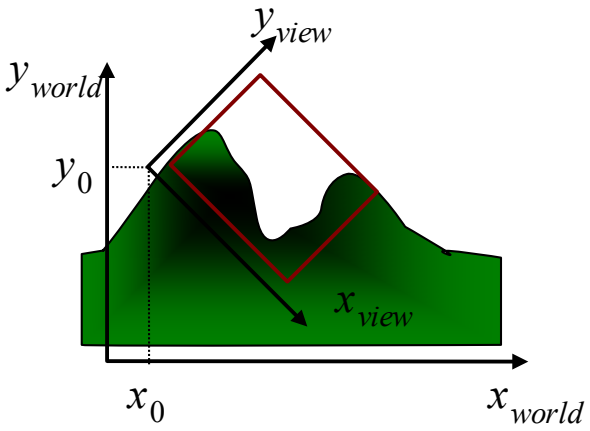
- Consider two approaches to rendering a scene with opaque objects
- For every pixel, determine which object that projects on the pixel is closest to the viewer and compute the shade of this pixel
  - Ray tracing paradigm
- For every object, determine which pixels it covers and shade these pixels
  - Pipeline approach
  - Must keep track of depths

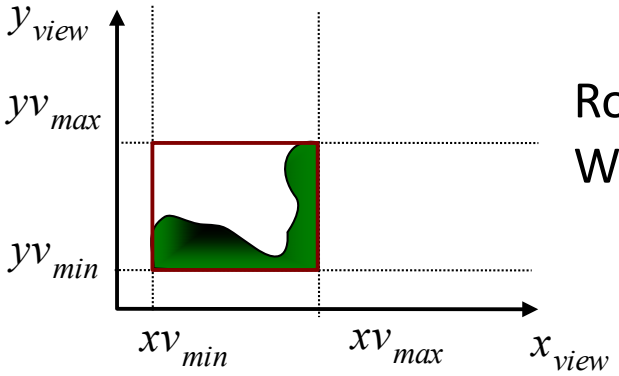
The clipping window





Rectangular Window





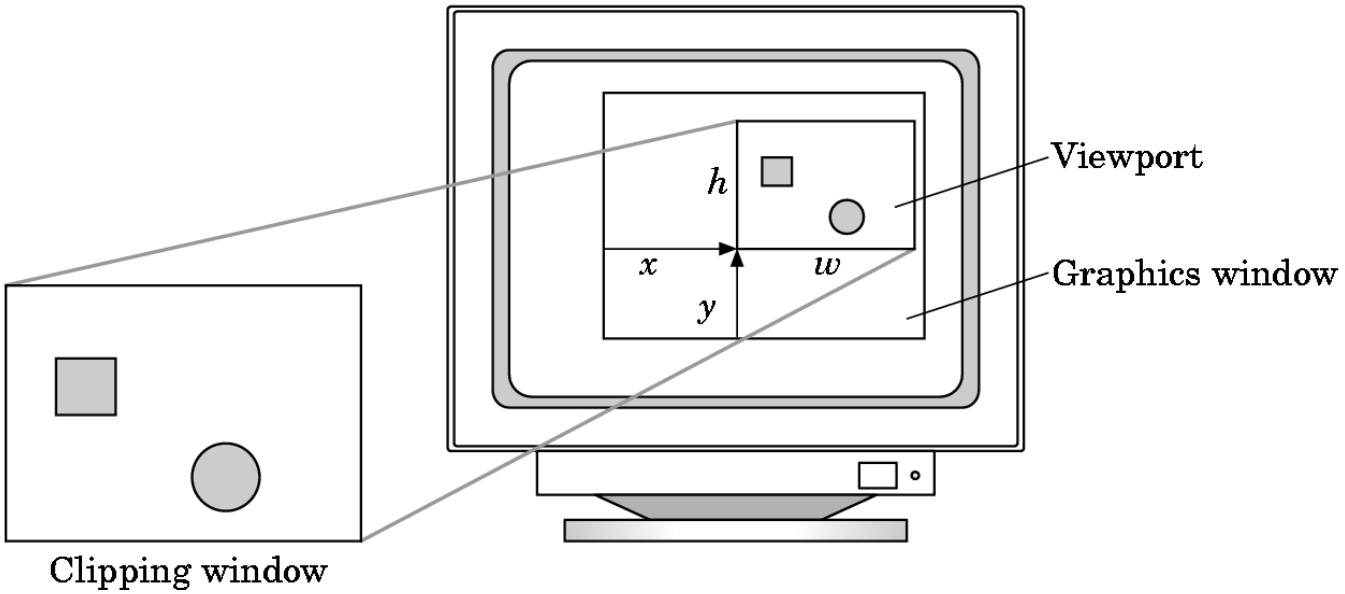
Rotated Window

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 of the GLUT display window we create (see next slide).

Clipping window and viewport

```
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)
```



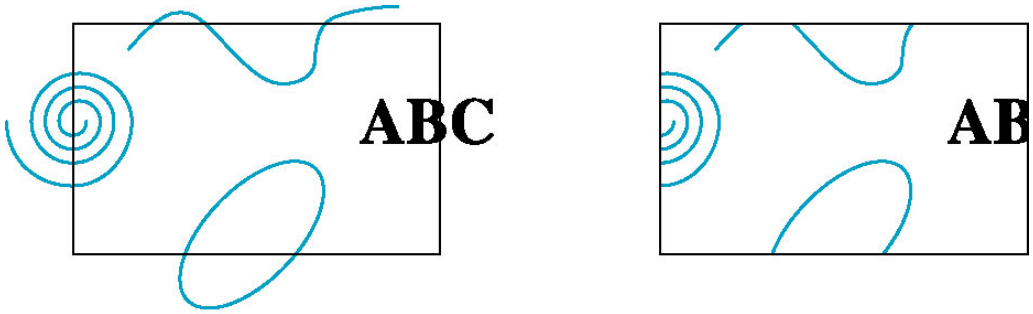
The viewport is part of the state

– changes between rendering objects or redisplay

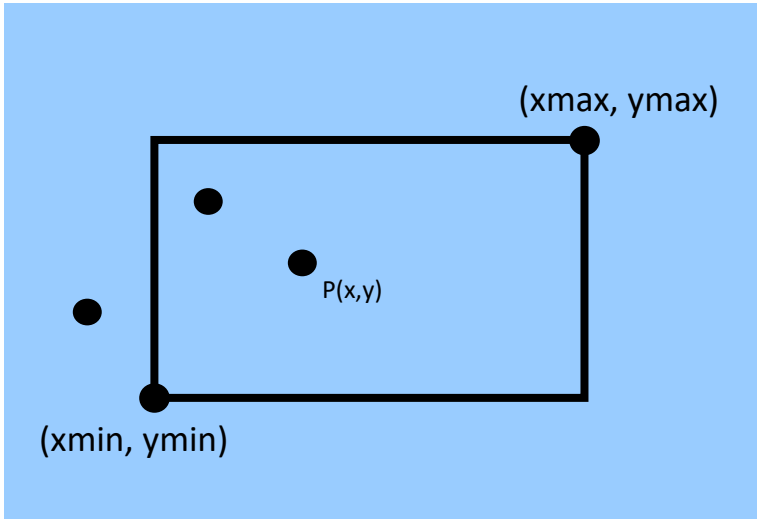
– different window-viewport transformations used to make the scene

Clipping primitives

- Different primitives can be handled in different ways
  - Points
  - Lines
  - Polygons
- 2D against clipping window
- 3D against clipping volume
- Easy for line segments and polygons
- Hard for curves and text - converted to lines and polygons first



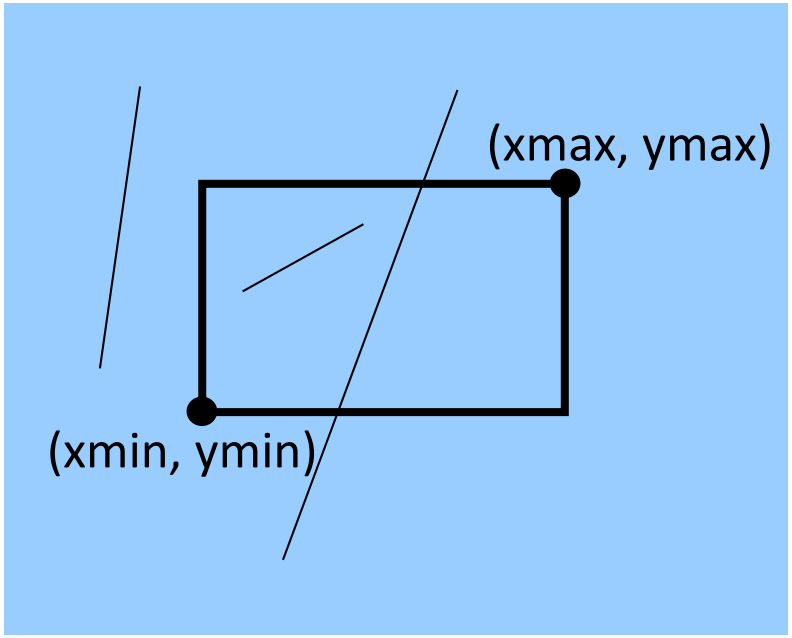
2D point clipping



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

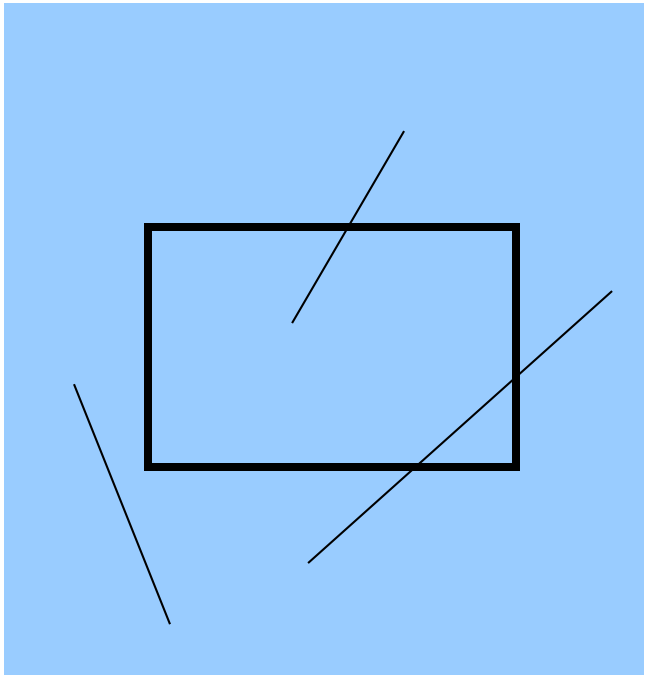
```
If (xmin <= x <= xmax) and (ymin <= y <= ymax)
        the point (x,y) is inside
else
        the point (x,y) is outside
```

2D line clipping



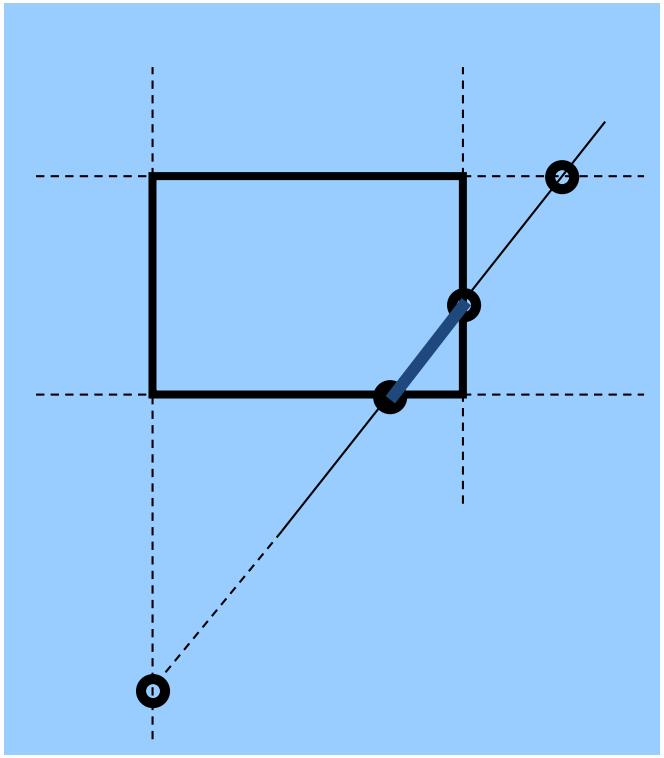
- Determine whether a line is inside, outside or partially inside the window.
- If a line is partially inside, we need to display the inside segment.

2D line clipping – non-trivial cases



- Lines that cannot be trivially rejected (i.e. entirely outside one of the 4 clipping window edges) or trivially accepted (i.e. entirely within the clipping window):
  - One point inside, and one point outside;
  - Both points are outside, but not “trivially” outside.
- Need to find the line segments that are inside.

2D line clipping – non-trivial clipping

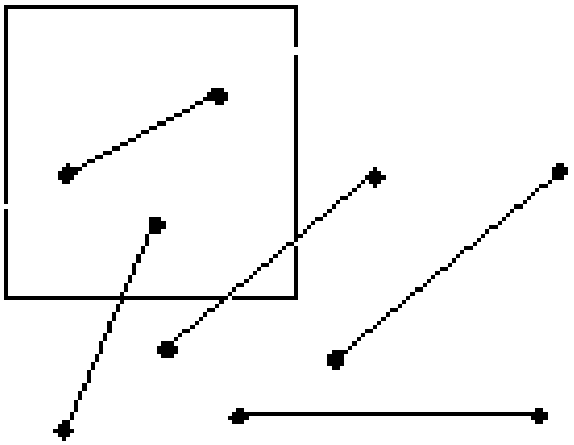


- Compute the line-window boundary edge intersection.
- There will be four intersections, but only one or two are on the window edges.
- These two points are the end points of the desired line segment.

Brute force clipping of lines (Simultaneous equations)

Brute force clipping: solve simultaneous equations using $y = mx + b$ for line and four clip edges

- Slope-intercept formula handles infinite lines only
- Does not handle vertical lines



Brute force clipping of lines (Similar triangles)

Clip a line against an edge of the window, e.g. the top edge, so we need to find out x' and y' (which is y_{max}).

Similar triangles

$A / B = C / D$

$A = (x_2 - x_1)$

$B = (y_2 - y_1)$

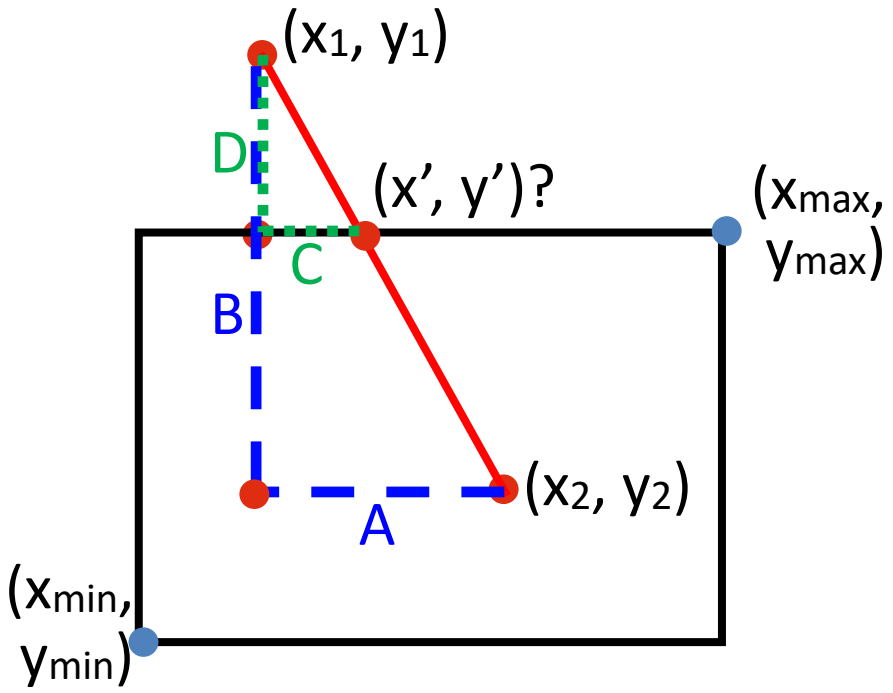
$C = (x' - x_1)$

$D = (y' - y_1) = (y_{max} - y_1)$

$\rightarrow C = A \bullet D / B$

$\rightarrow x' = x_1 + C$

$\rightarrow (x', y') = (x_1 + C, y_{max})$



Brute force clipping of lines (Similar triangles)

- The problem?
  Too expensive (the numbers below are for 2D)!
  - 4 floating point subtractions
  - 1 floating point multiplication
  - 1 floating point division
  - Repeat 4 times (once for each edge)

- We need to do better!

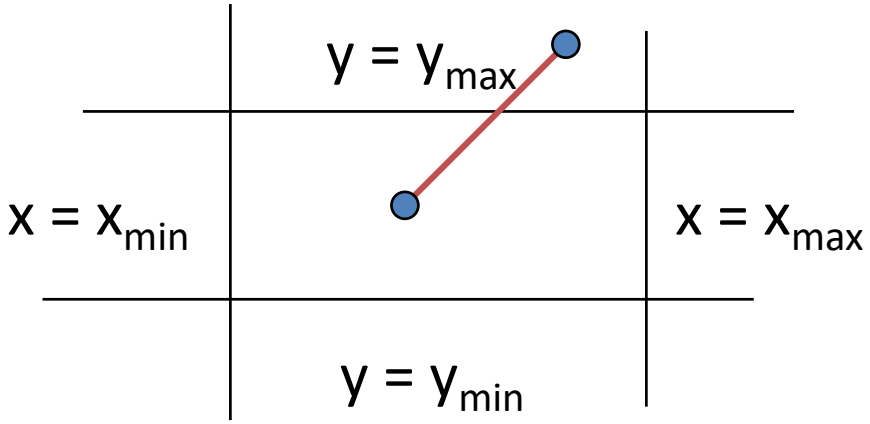
Cohen-Sutherland 2D line clipping (1)

- Cohen-Sutherland
  - organised
  - efficient
  - computes new end points for the lines that must be clipped

- How it works?
  - Creates an outcode for each end point that contains location information of the point with respect to the clipping window.
  - Uses outcodes for both end points to determine the configuration of the line with respect to the clipping window.
  - Computes new end points if necessary.
  - Extends easily to 3D (using 6 faces instead of 4 edges).

Cohen-Sutherland 2D line clipping (2)

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



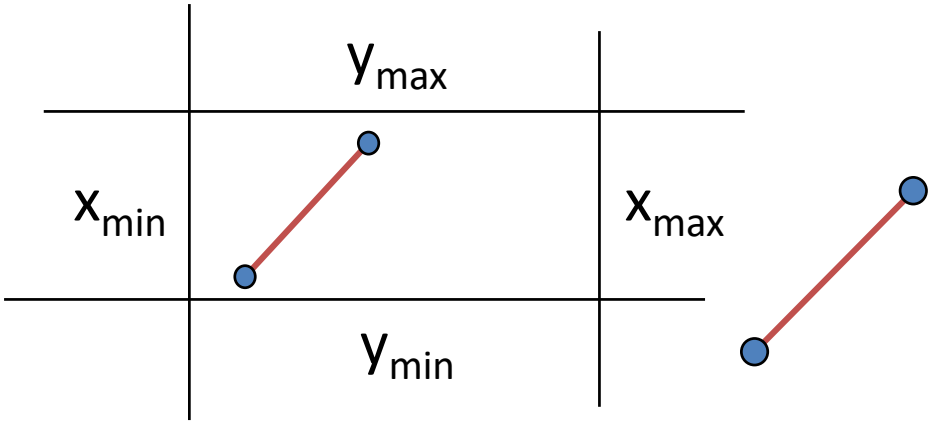
Cohen-Sutherland 2D line clipping (3)

- Case 1: both endpoints of line segment inside all four lines
  - Draw (trivial accept) line segment as is

Xmin <= x1 and x2 <= Xmax
Ymin <= y1 and y2 <= Ymax

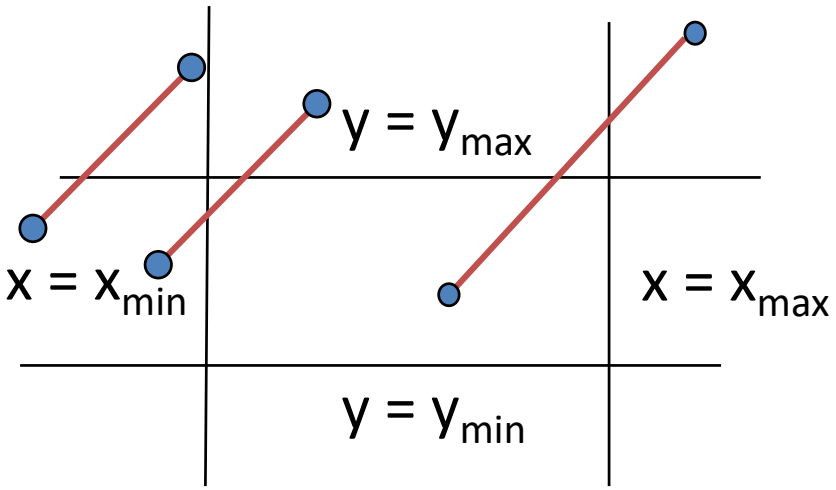
- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (trivial reject) the line segment

- x1 and x2 < Xmin OR
- x1 and x2 > Xmax OR
- y1 and y2 < Ymin OR
- y1 and y2 > Ymax



Cohen-Sutherland 2D line clipping (4)

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection

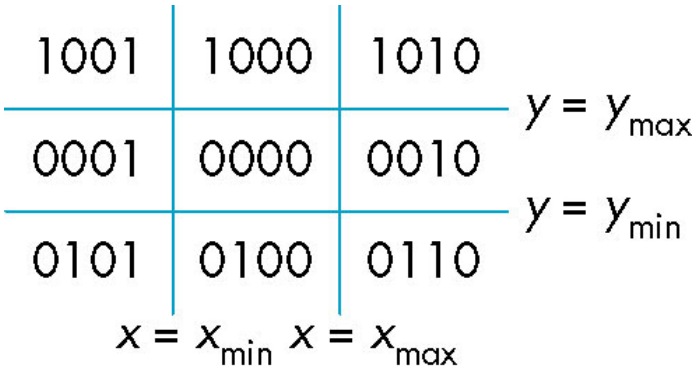


Cohen-Sutherland 2D line clipping – defining outcodes (1)

- Split plane into 9 regions.
- Assign each a 4-bit outcode (above, below, right and left).

b_0 = 1 if $x < x_{min}$, 0 otherwise
b_1 = 1 if $x > x_{max}$, 0 otherwise
b_2 = 1 if $y < y_{min}$, 0 otherwise
b_3 = 1 if $y > y_{max}$, 0 otherwise

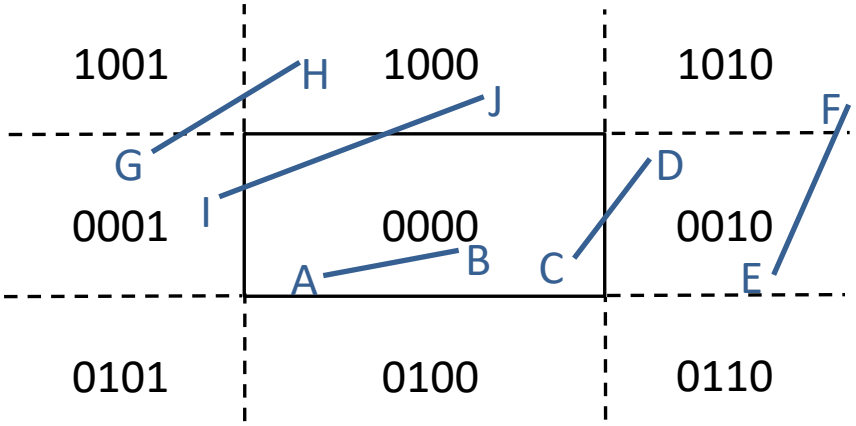
- Assign each endpoint an outcode.
- Computation of outcode requires at most 4 subtractions.



Cohen-Sutherland 2D line clipping – defining outcodes (2)

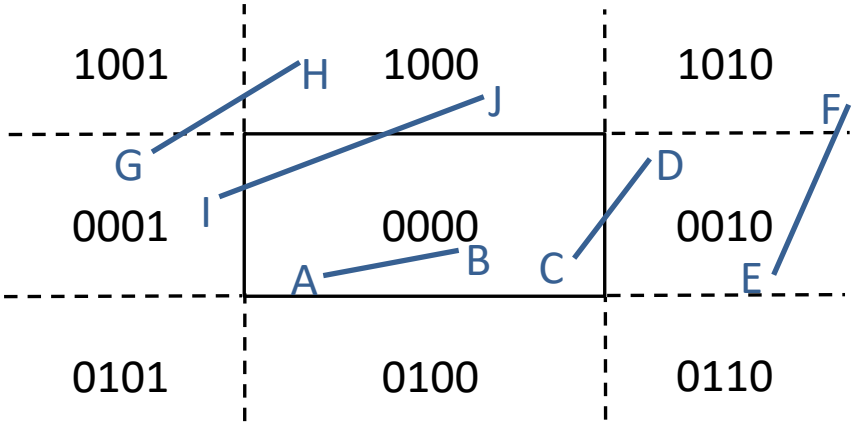
Consider the 5 cases below

- AB: outcode(A) = outcode(B) = 0000
  - Accept line segment (Trivial accept)
- CD: outcode (C) = 0000, outcode(D) = 0010 ≠ 0000
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from C to a point in a region with 2 ones in outcode, we might have to do two intersections



Cohen-Sutherland 2D line clipping – defining outcodes (3)

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) ≠ 0
  - Both outcodes (0010, 1010) have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - Trivial reject
- GH and IJ
  - Same outcodes (0001, 1000), neither 0 but logical AND yields 0
  - Shorten line segment by intersecting with one of window sides
  - Compute outcode of intersection (new endpoint of shortened line)
  - Re-execute algorithm

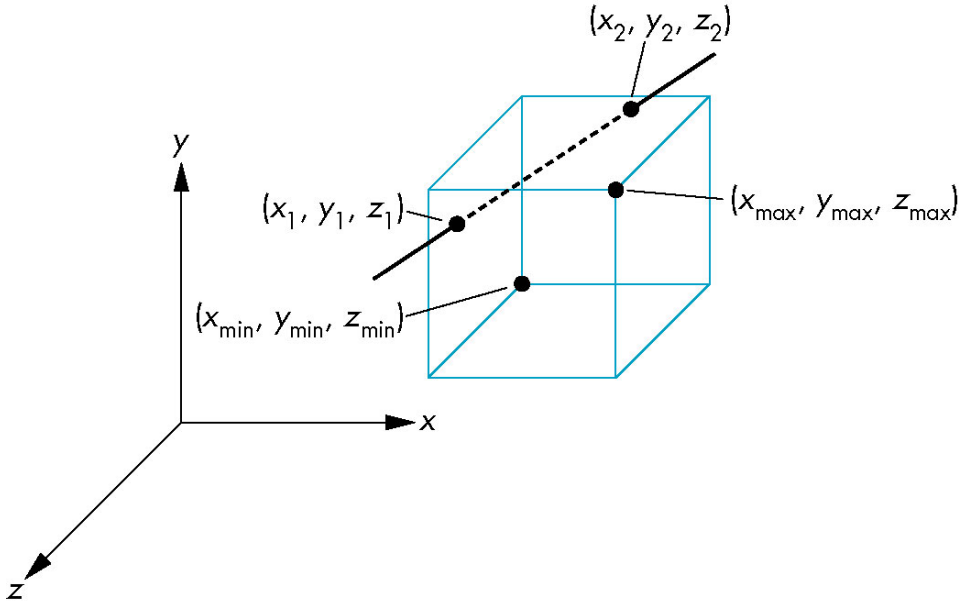


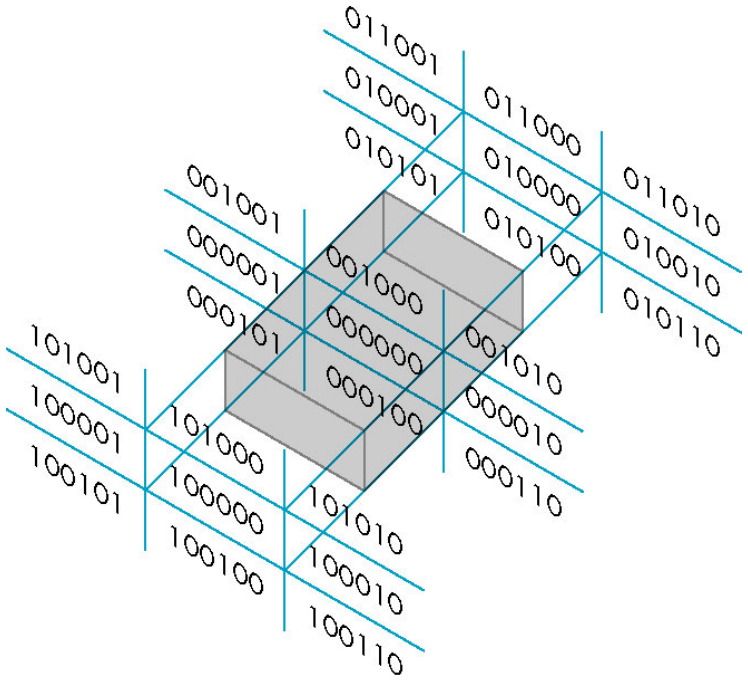
Cohen-Sutherland 2D line clipping: Efficiency

- In many applications, the clipping window is small relative to the size of the entire database.
- Most line segments are outside one or more sides of the window and can be eliminated based on their outcodes.
- Inefficient when code has to be re-executed for line segments that must be shortened in more than one step, because two intersections have to be calculated anyway, in addition to calculating the outcode (extra computing in such a case).

Cohen-Sutherland 3D line clipping

- Use 6-bit outcodes
- When needed, clip line segment against planes





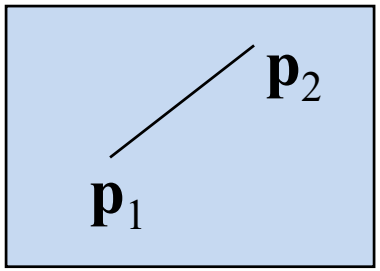
Liang-Barsky (Parametric line formulation) – 1

Consider the parametric form of a line segment

$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2, \quad 0 \leq \alpha \leq 1$$

or two scale expressions

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$
$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$

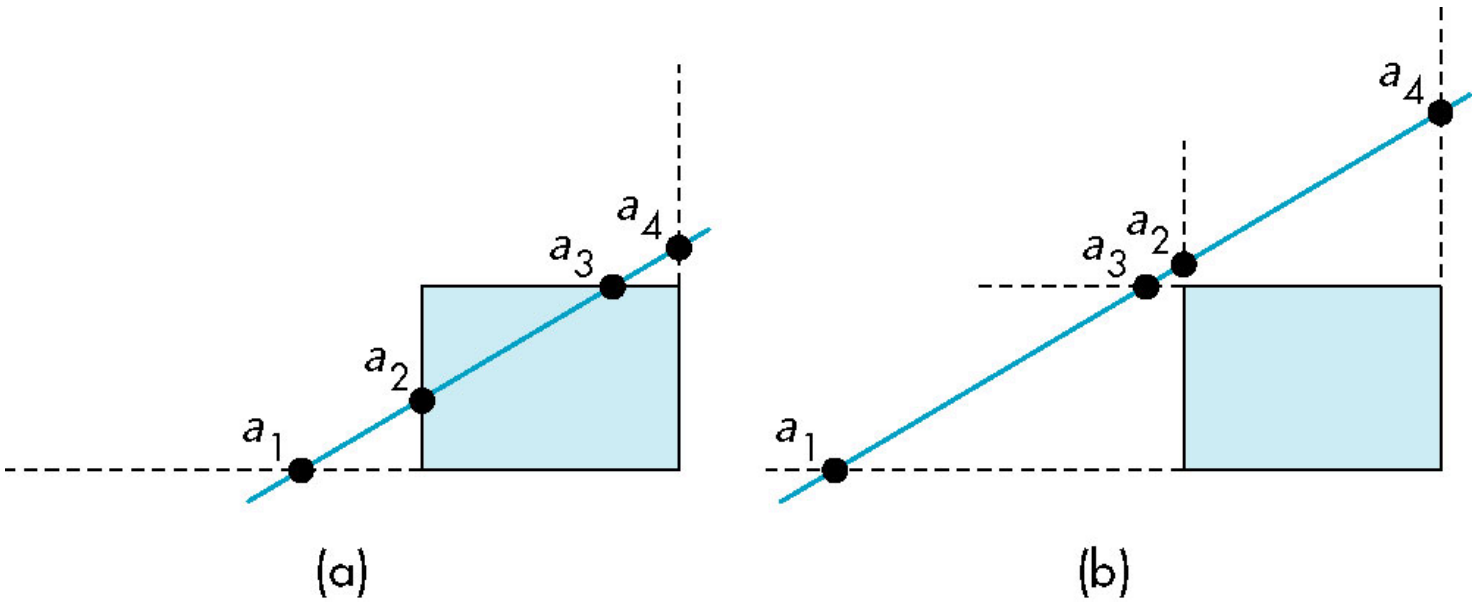


We can distinguish the cases by looking at the ordering of the α values where the line is determined by the line segment crossing the lines that determine the window.

Liang-Barsky (Parametric line formulation) – 2

- In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$
  Intersect right, top, left and bottom: shorten

- In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$
  Intersect right, left, top and bottom: reject



Liang-Barsky (Parametric line formulation) – 3

Advantages:

- Can accept/reject as easily as with Cohen-Sutherland;
- Using values of α, we do not have to use the algorithm recursively as with the Cohen-Sutherland method;
- Extends to 3D.

Plane-line intersections

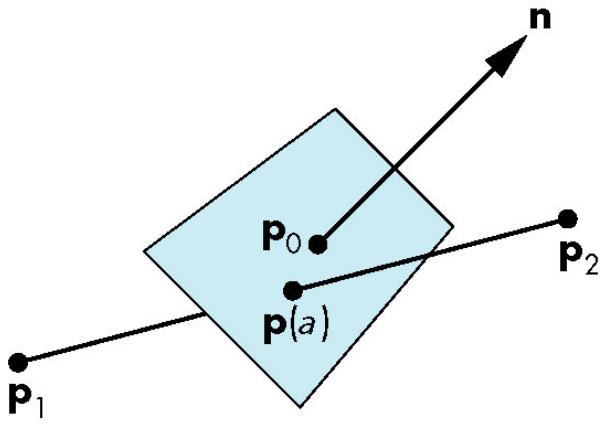
If we write the line and plane equations in matrix form (where **n** is the normal to the plane and $\mathbf{p}_0$ is a point on the plane), we must solve the equations

$$\mathbf{p}(\alpha) = (1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2, \quad 0 \le \alpha \le 1$$
$$\mathbf{n} \cdot (\mathbf{p}(\alpha) - \mathbf{p}_0) = 0$$

For the α corresponding to the point of intersection, this value is

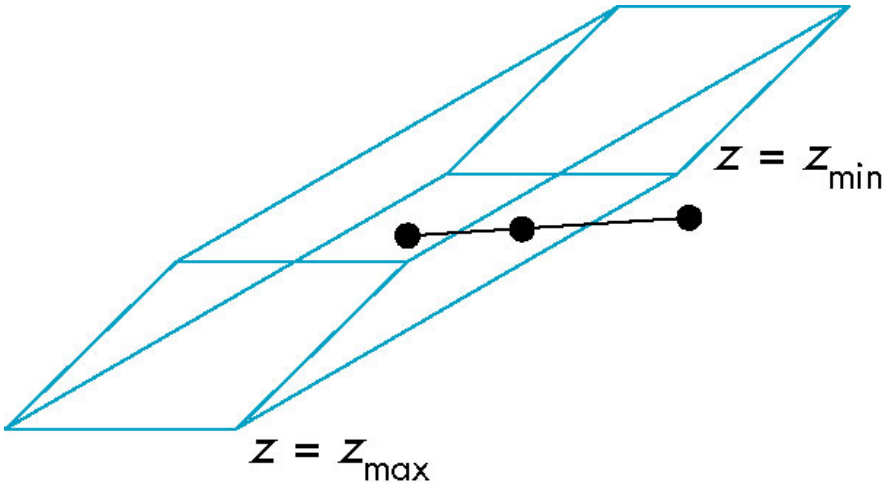
$$\alpha = \frac{\mathbf{n} \bullet (\mathbf{p}_o - \mathbf{p}_1)}{\mathbf{n} \bullet (\mathbf{p}_2 - \mathbf{p}_1)}$$

The intersection requires 6 multiplications and 1 division.



General 3D clipping

- General clipping in 3D requires intersection of line segments against arbitrary plane.
- Example: oblique view

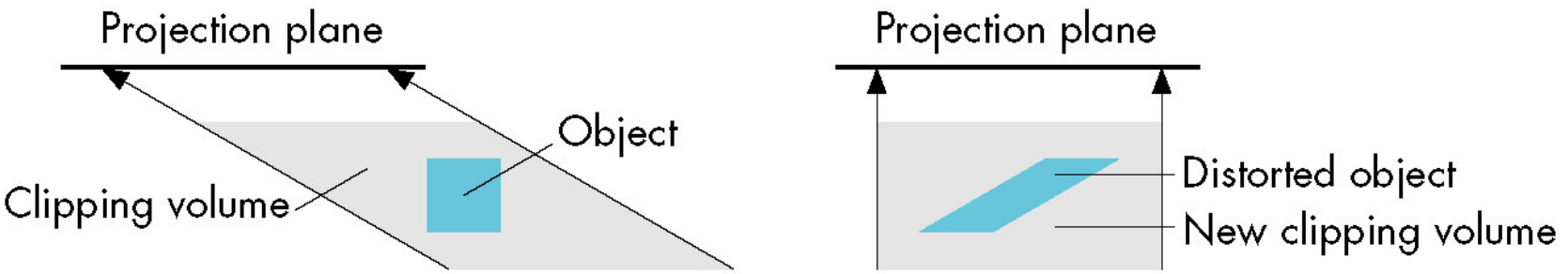


Normalised form

Normalisation is part of viewing (pre clipping) but after normalisation, we clip against sides of right parallelepiped.

Typical intersection calculation now requires only a floating point subtraction, e.g. is $x > x_{max}$?

Top view



Before normalisation

After normalisation

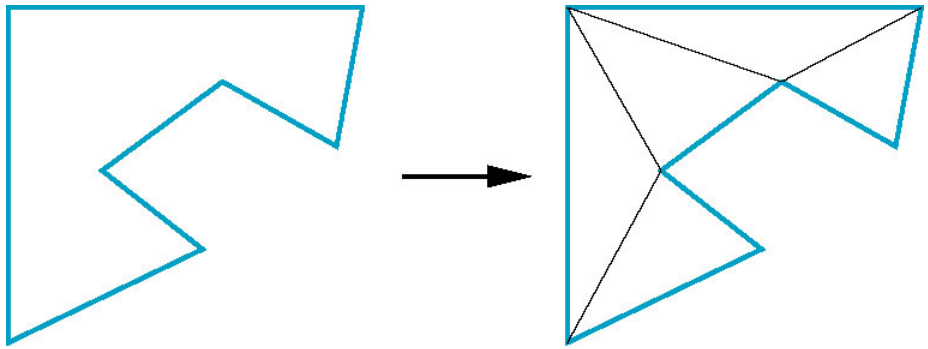
Polygon clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment;
  - Clipping a polygon can yield multiple polygons.

- However, clipping a convex polygon can yield at most one other polygon.

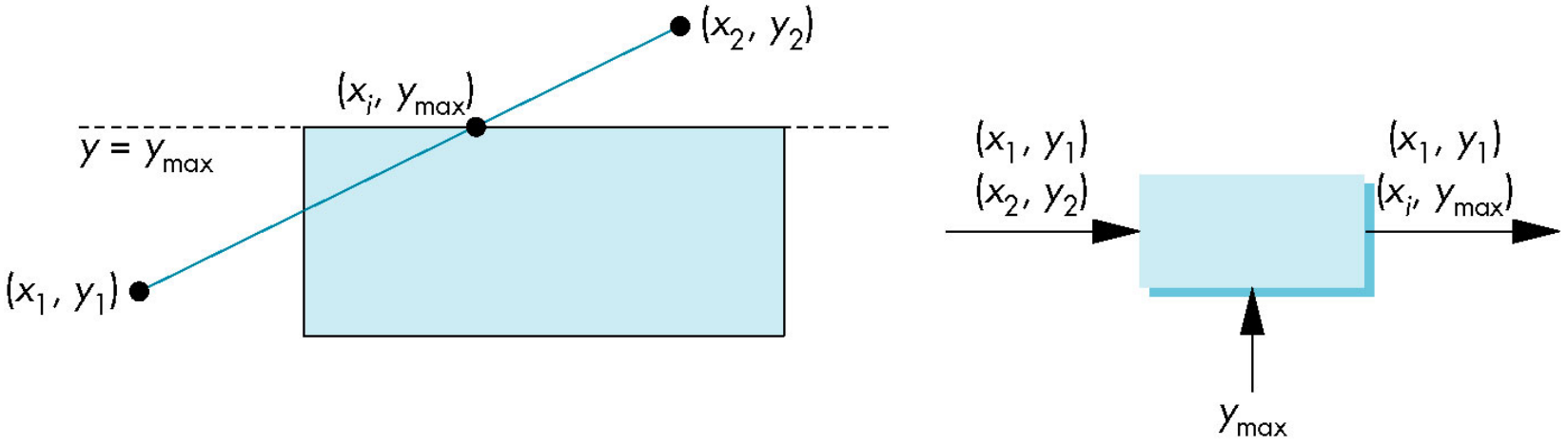
Convexity and tessellation

- One strategy is to replace non-convex (*concave*) polygons with a set of triangular polygons (a *tessellation*).
- Also makes fill easier.
- Tessellation code in GLU library.



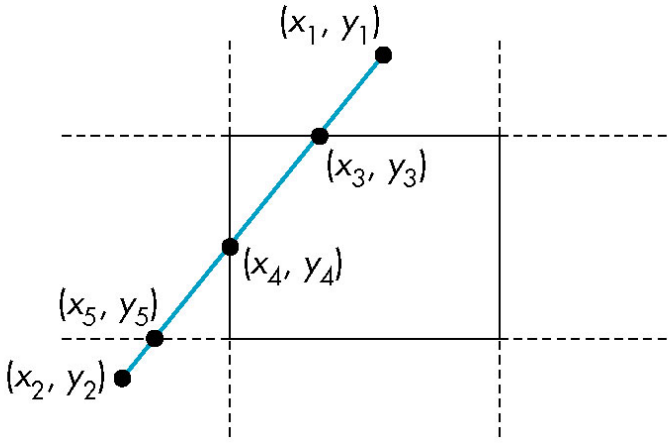
Clipping as a black box

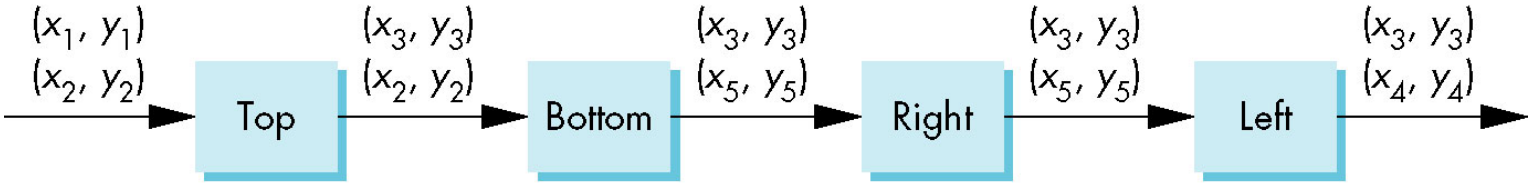
We can consider line segment clipping as a process that takes in two vertices and produces either no vertices or the vertices of a clipped line segment.



Pipeline clipping of line segments

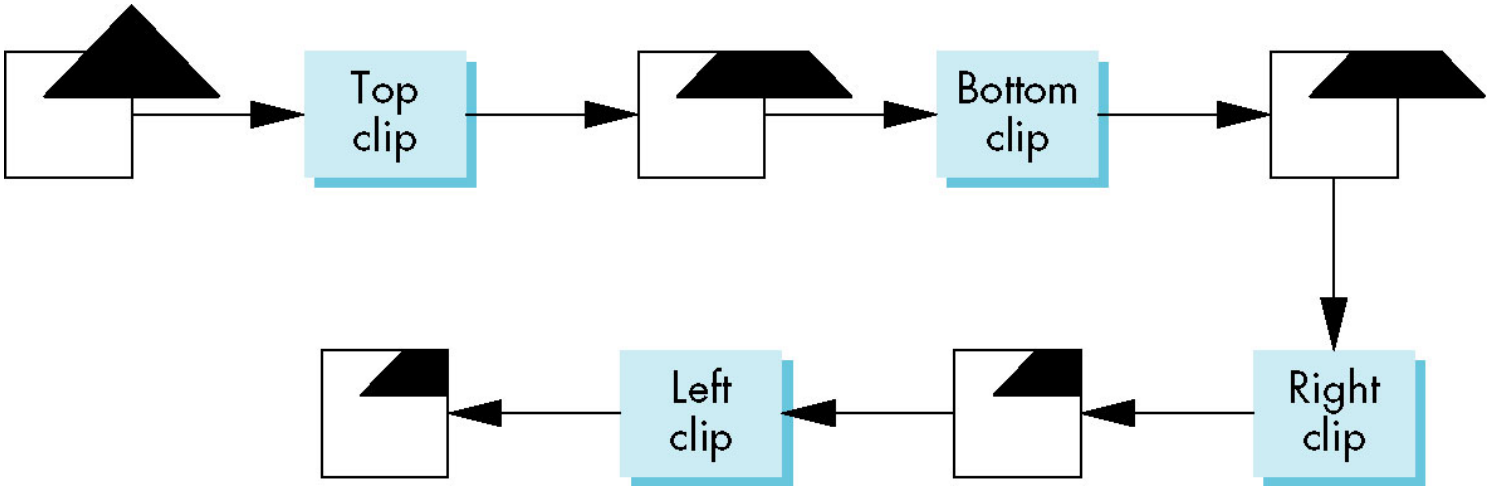
Clipping against each side of window is independent of other sides so we can use four independent clippers in a pipeline.





Pipeline clipping of polygons

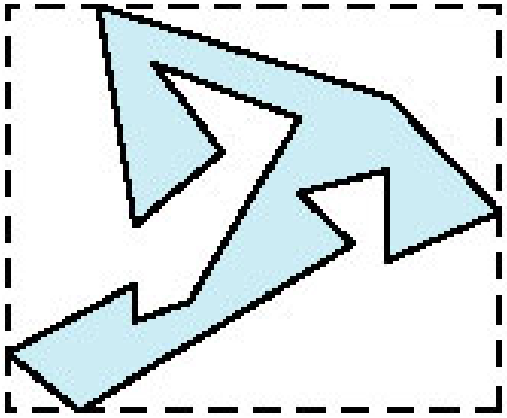
- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency



Bounding boxes (1)

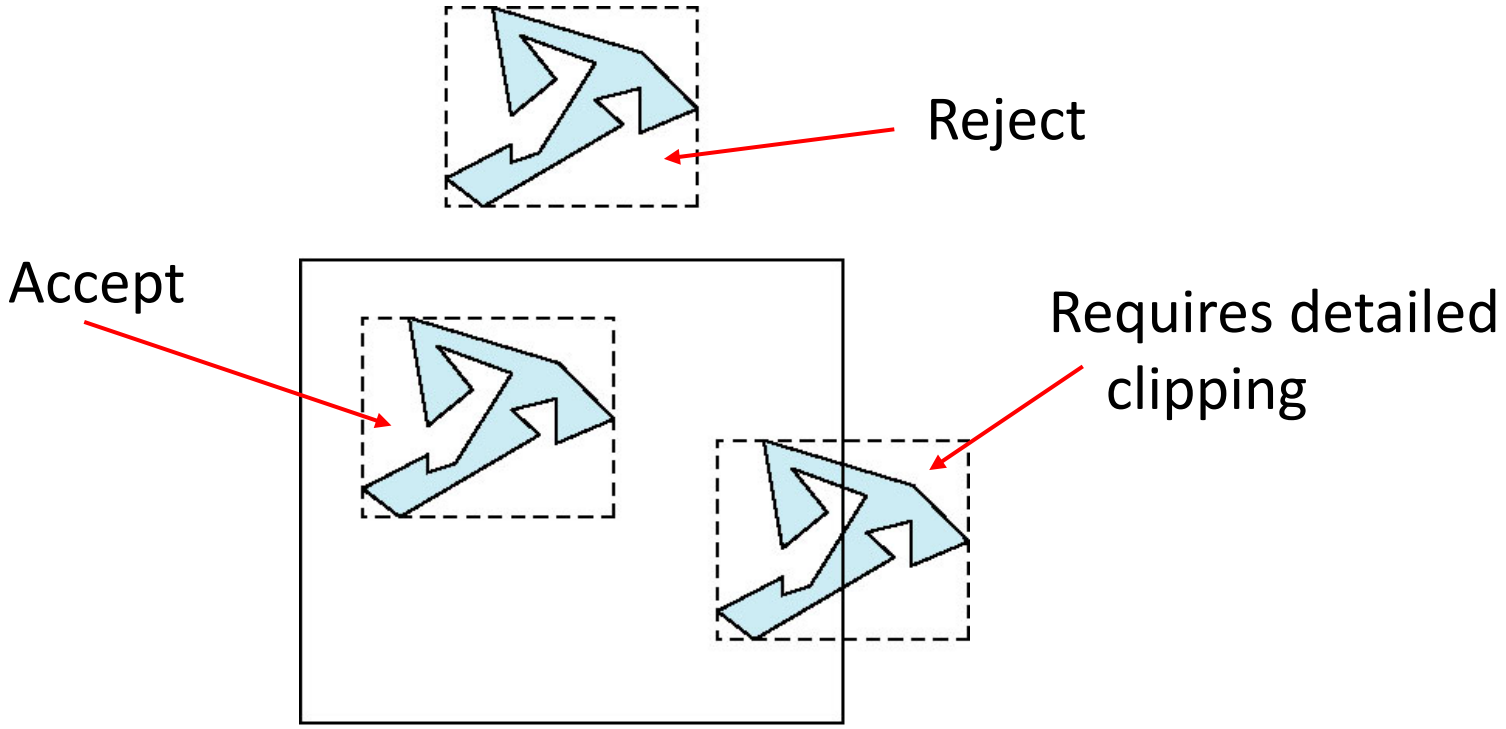
Rather than doing clipping on a complex polygon, we can use an *axis-aligned bounding box* or *extent*:

- Smallest rectangle aligned with axes that encloses the polygon
- Simple to compute: max and min of x and y



Bounding boxes (2)

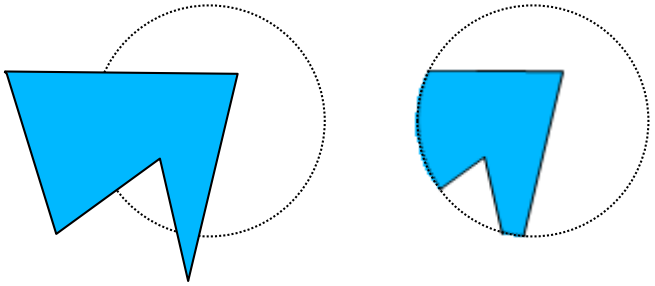
We can usually determine accept/reject based only on bounding box.



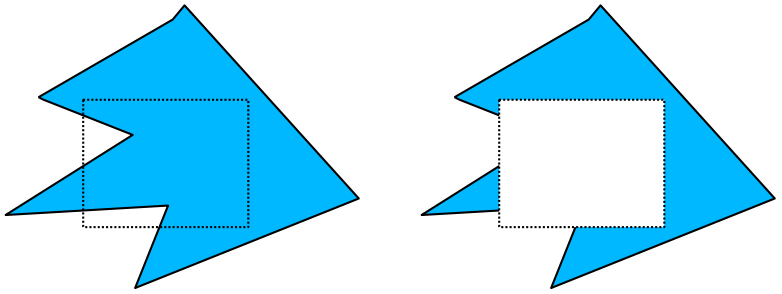
Other issues in clipping

- Clipping other shapes:
  Circle, Ellipse, Curves

- Clipping a shape against another shape.



- Clipping the interior



Setting up a 2D viewport in OpenGL

```
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).

Setting up a clipping-window in OpenGL

```
glMatrixMode(GL_PROJECTION)

glLoadIdentity();  // reset, so that new viewing parameters
                   // are not combined with old ones (if any)

gluOrtho2D(xwmin, xwmax, ywmin, ywmax);
glOrtho(xwmin, xwmax, ywmin, ywmax, near, far);
gluPerspective(vof, aspectratio, near, far);
glFrustum(xwmin, xwmax, ywmin, ywmax, near, far);
```

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

Summary

- Concepts of clipping
  - What is clipping and why it is important
  - Points, lines and polygons
- Line clipping algorithms
  - Brute Force Simultaneous Equations
  - Brute Force Similar Triangles
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon clipping in brief
- OpenGL functions
  - `glViewport()`
  - `glMatrixMode() / glLoadIdentity()`
  - `glOrtho() / gluOrtho2D()`
    `gluPerspective / glFrustum()`