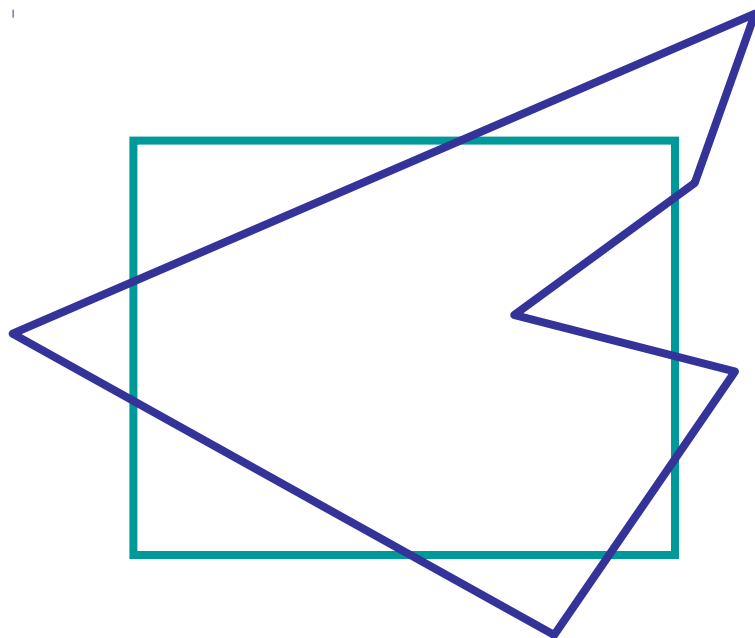
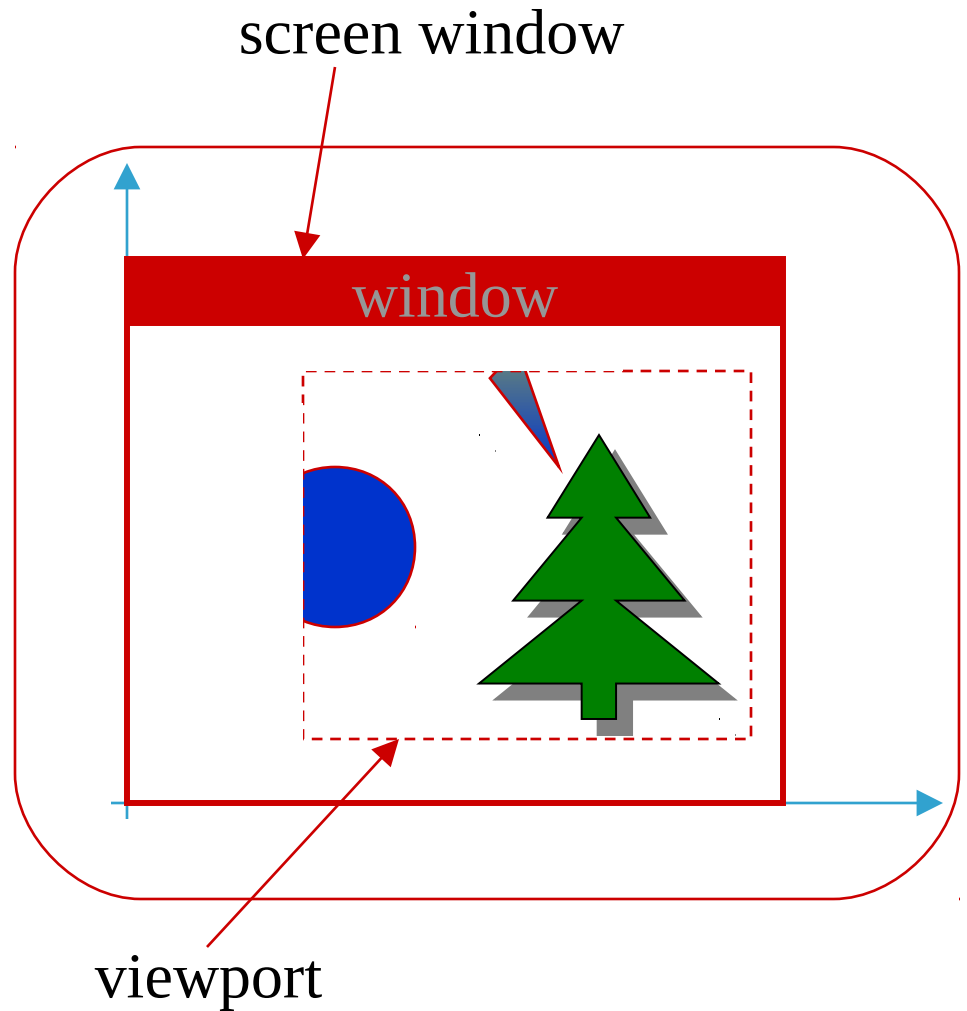
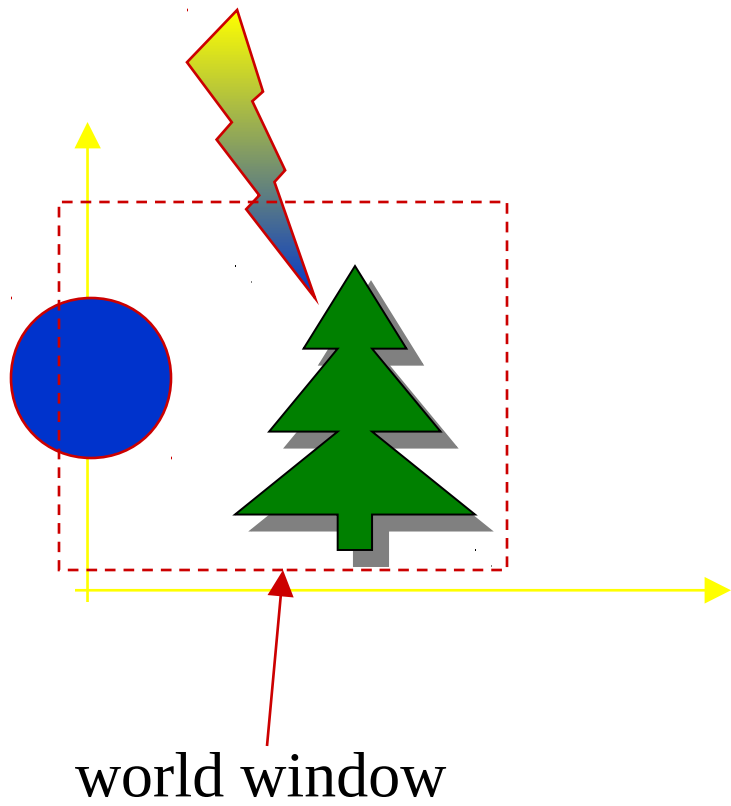


Rezanje črt in poligonov



World window & viewport



Clipping

- We have talked about 2D scan conversion of line-segments and polygons
- What if endpoints of line segments or vertices of polygons lie outside the visible device region?
- Need clipping!

Clipping

- Clipping of primitives is done usually before scan converting the primitives
- Reasons being
 - scan conversion needs to deal only with the clipped version of the primitive, which might be much smaller than its unclipped version
 - Primitives are usually defined in the real world, and their mapping from the real to the integer domain of the display might result in the overflowing of the integer values resulting in unnecessary artifacts

Clipping

- **Clipping:** Remove points outside a region of interest.
 - Want to discard everything that's outside of our window...
- **Point clipping:** Remove points outside window.
 - A point is either entirely inside the region or not.
- **Line clipping:** Remove portion of line segment outside window.
 - Line segments can straddle the region boundary.
 - Liang-Barsky algorithm efficiently clips line segments to a halfspace.
 - Halfspaces can be combined to bound a convex region.
 - Use *outcodes* to better organize combinations of halfspaces.
 - Can use some of the ideas in Liang-Barsky to clip points.

Clipping

- Lines outside of world window are not to be drawn.
- Graphics API clips them automatically.
- But clipping is a general tool in graphics!

Rezanje (clipping)

• Cohen-Sutherland

- Uporaba kode za hitro izločanje črt
- Izračun rezanja preostalih črt z oknom gledanja
 - Introduced parametric equations of lines to perform edge/viewport intersection tests
 - Truth in advertising, Cohen-Sutherland doesn't use parametric equations of lines
- Viewport intersection code:
 - $(x_1, y_1), (x_2, y_2)$ intersect with vertical edge at x_{right}
 - $y_{\text{intersect}} = y_1 + m(x_{\text{right}} - x_1), m = (y_2 - y_1)/(x_2 - x_1)$
 - $(x_1, y_1), (x_2, y_2)$ intersect with horizontal edge at y_{bottom}
 - $x_{\text{intersect}} = x_1 + (y_{\text{bottom}} - y_1)/m, m = (y_2 - y_1)/(x_2 - x_1)$

Parametrične enačbe

- Faster line clippers use parametric equations

- Line 0:

- $x^0 = x_0^0 + (x_1^0 - x_0^0) t^0$
 - $y^0 = y_0^0 + (y_1^0 - y_0^0) t^0$

- Viewport Edge L:

- $x^L = x_0^L + (x_1^L - x_0^L) t^L$
 - $y^L = y_0^L + (y_1^L - y_0^L) t^L$

- $x_0^0 + (x_1^0 - x_0^0) t^0 = x_0^L + (x_1^L - x_0^L) t^L$

- $y_0^0 + (y_1^0 - y_0^0) t^0 = y_0^L + (y_1^L - y_0^L) t^L$

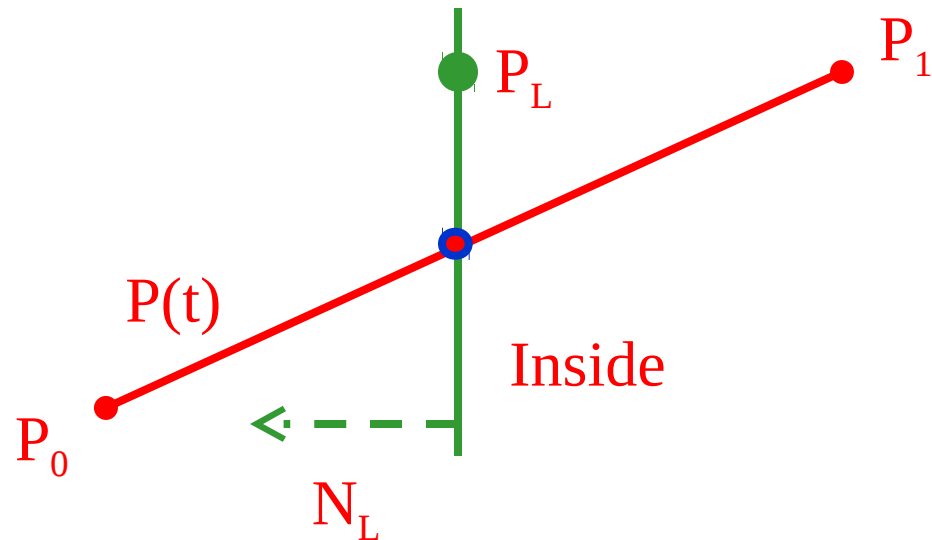
- Solve for t^0 and/or t^L

Algorithem Cyrus-Beck

- Use parametric equations of lines
- Optimize
- We saw that this could be expensive...
- Start with parametric equation of line:
 - $P(t) = P_0 + (P_1 - P_0) t$
- And a point and normal for each edge
 - P_L, N_L

Algorithem Cyrus-Beck

- $N_L \cdot [P(t) - P_L] = 0$



- Substitute line equation for $P(t)$

- Solve for t

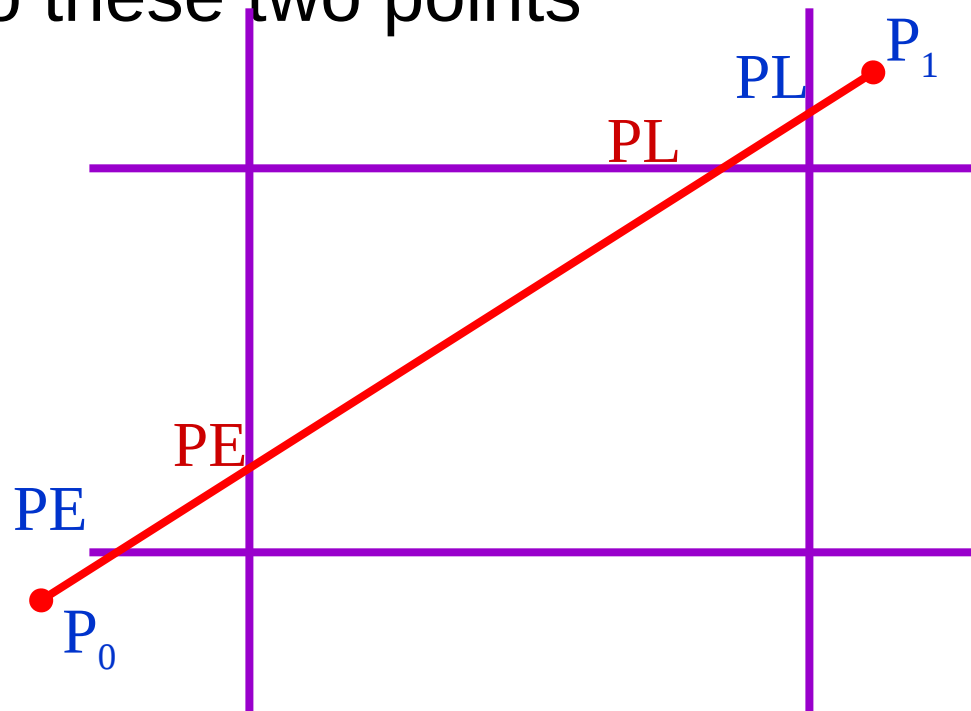
- $t = N_L [P_0 - P_L] / -N_L [P_1 - P_0]$

Algoritim Cyrus-Beck

- Compute t for line intersection with all four edges
- Discard all $(t < 0)$ and $(t > 1)$
- Classify each remaining intersection as
 - Potentially Entering (PE)
 - Potentially Leaving (PL)
- $N_L \cdot [P_1 - P_0] > 0$ implies PL
- $N_L \cdot [P_1 - P_0] < 0$ implies PE
 - Note that we computed this term in when computing t

Algoritim Cyrus-Beck

- Compute PE with largest t
- Compute PL with smallest t
- Clip to these two points



Algoritem Cyrus-Beck

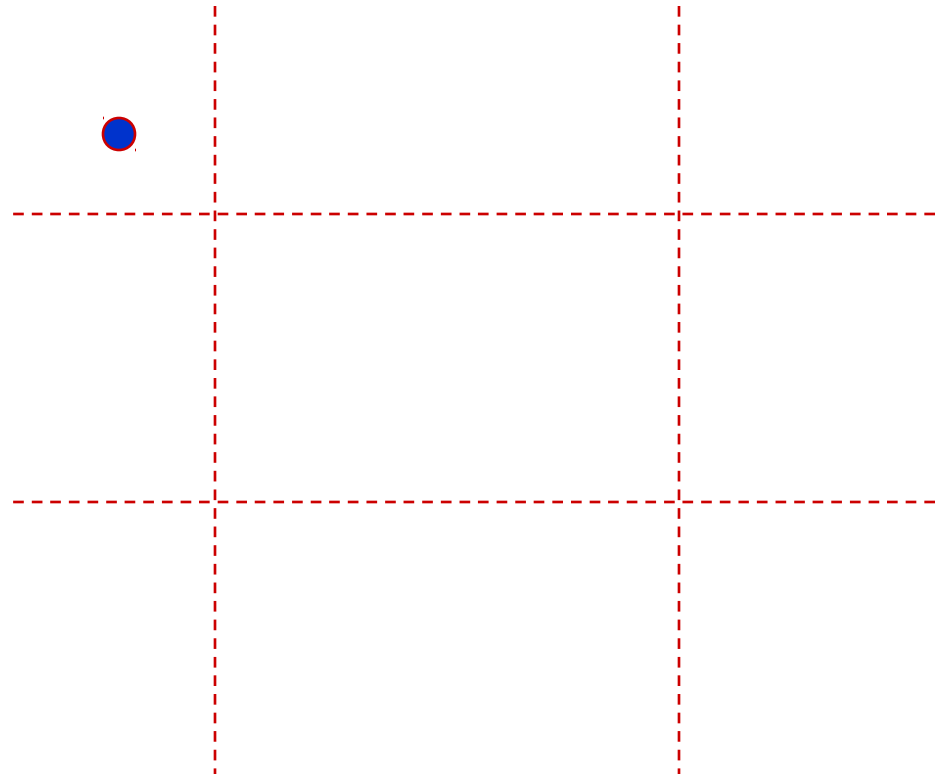
- Because of horizontal and vertical clip lines:
 - Many computations reduce
- Normals: $(-1, 0)$, $(1, 0)$, $(0, -1)$, $(0, 1)$
- Pick constant points on edges
- solution for t:
 - $-(x_0 - x_{\text{left}}) / (x_1 - x_0)$
 - $(x_0 - x_{\text{right}}) / -(x_1 - x_0)$
 - $-(y_0 - y_{\text{bottom}}) / (y_1 - y_0)$
 - $(y_0 - y_{\text{top}}) / -(y_1 - y_0)$

Cohen-Sutherland region outcodes

- 4 bits:

TTFF

Left of window?
Above window?
Right of window?
Below window?



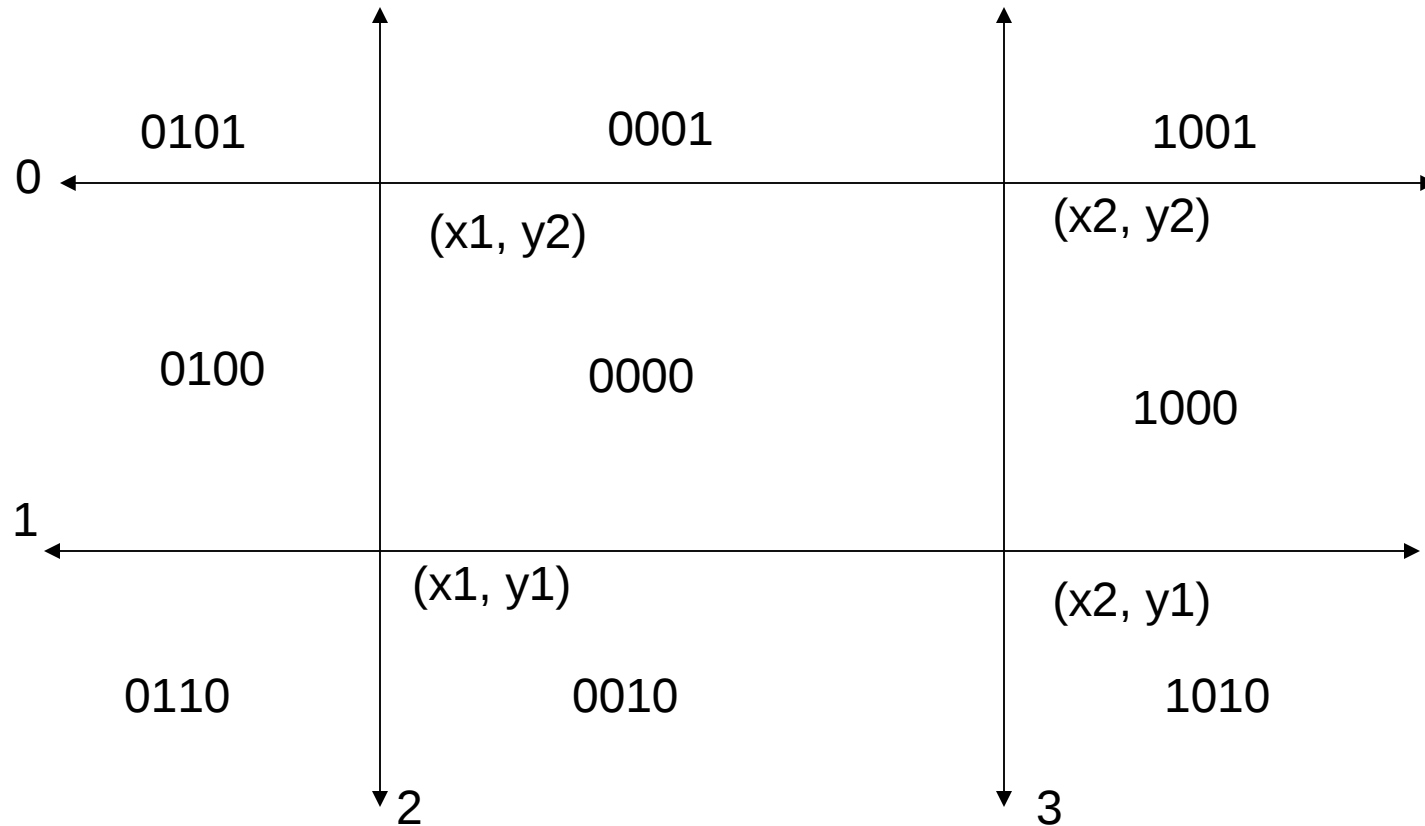
Cohen-Sutherland region outcodes

TFFF	FTFF	FTTF
TFFF	FFFF	FFTF
TFFT	FFFT	FFTT

- Trivial accept: both endpoints are FFFF
- Trivial reject: both endpoints have T in the same position

Cohen-Sutherland Algorithm

Half space code $(x < x_2) \mid (x > x_1) \mid (y > y_1) \mid (y < y_2)$



Cohen-Sutherland Algorithm

- Computing the code for a point is trivial
 - Just use comparison
- Trivial rejection is performed using the logical **and** of the two endpoints
 - A line segment is rejected if any bit of the **and** result is 1.
Why?

Cohen-Sutherland Algorithm

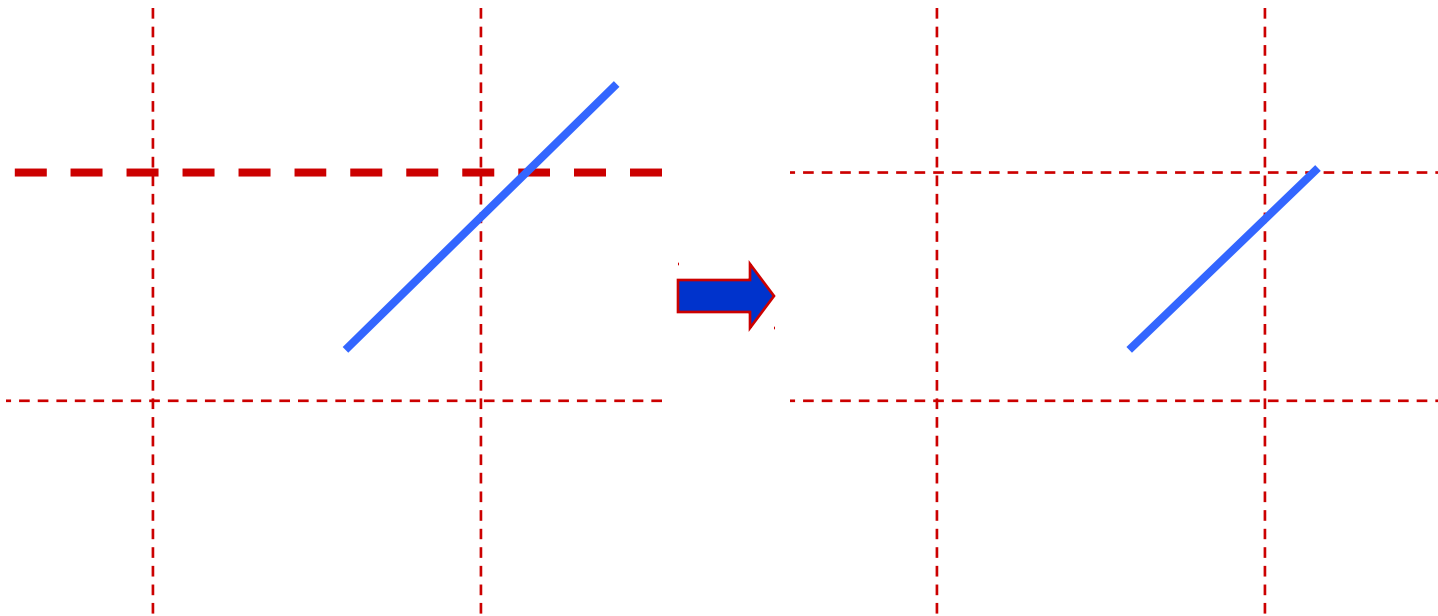
- Now we can efficiently reject lines completely to the left, right, top, or bottom of the rectangle.
- If the line cannot be trivially rejected (what cases?), the line is split in half at a clip line.
- Not that about one half of the line can be rejected trivially
- This method is efficient for large or small windows.

Cohen-Sutherland Algorithm

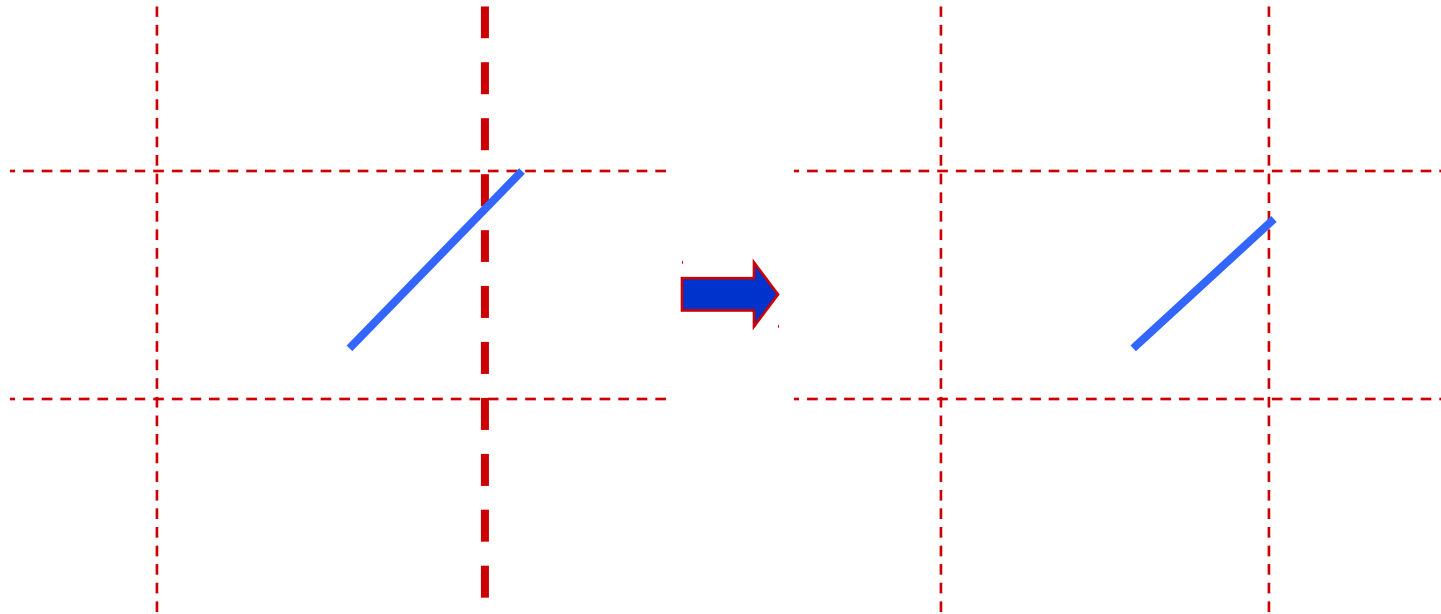
```
clip (int Ax, int Ay, int Bx, int By)
{
    int cA = code(Ax, Ay);
    int cB = code(Bx, By);
    while (cA | cB) {
        if(cA & cB) return;           // rejected
        if(cA) {
            update Ax, Ay to the clip line depending
            on which outer region the point is in
            cA = code(Ax, Ay);
        } else {
            update Bx, By to the clip line depending
            on which outer region the point is in
            cB = code(Bx, By);
        }
    }
    drawLine(Ax, Ay, Bx, By);
}
```

Cohen-Sutherland: chopping

- If segment is neither trivial accept or reject:
 - Clip against edges of window in turn



Cohen-Sutherland: chopping



Trivial accept

Cohen-Sutherland line clipper

- `int clipSegment (point p1, point p2)`

Do {

 If (trivial accept) return (1)

 If (trivial reject) return (0)

 If (p1 is outside)

 if (p1 is left) chop left

 else if (p1 is right) chop right

 ...

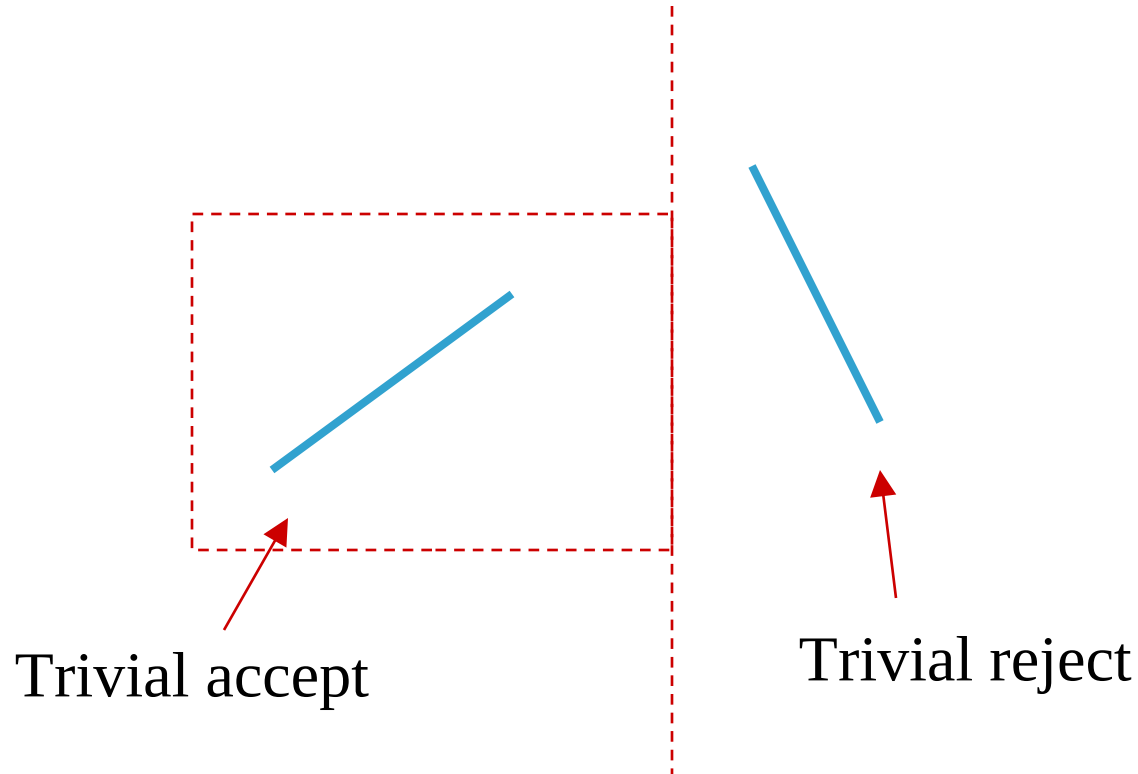
 If (p2 is outside)

 ...

} while (1)

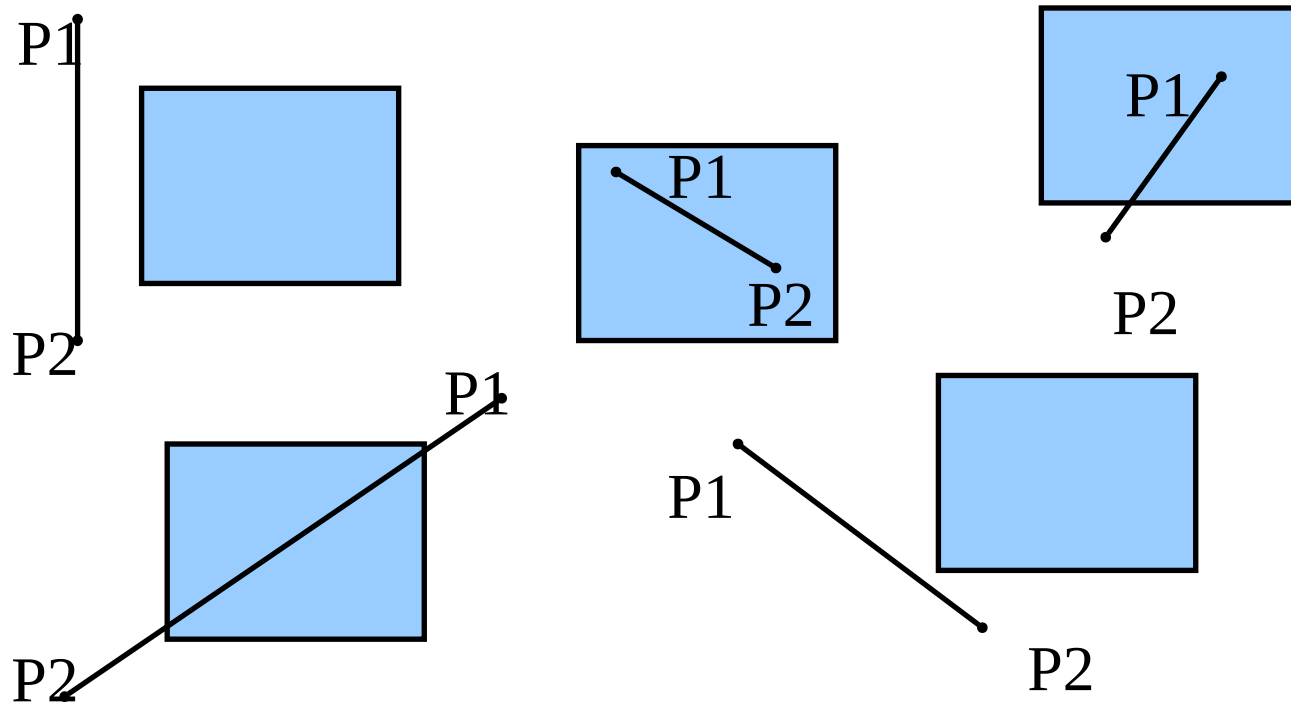
Cohen-Sutherland clipping

- Trivial accept/reject test!



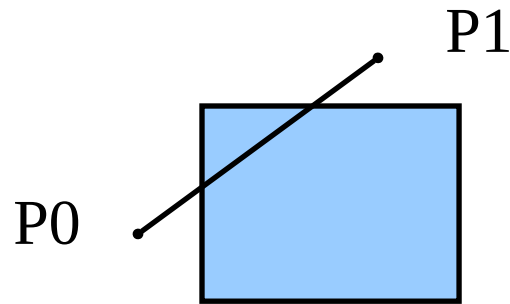
Trivially accept or trivially reject

- 0000 for both endpoints = accept
- matching 1's in any position for both endpoints = reject



Calculate clipped endpoints

$$y = y_0 + \text{slope} * (x - x_0) \quad x = x_0 + (1 / \text{slope}) * (y - y_0)$$



P0: Clip left

$$x = x_{min}$$

$$y = y_0 + [(y_1 - y_0) / (x_1 - x_0)] * (x_{min} - x_0)$$

P1: Clip top

$$y = y_{max}$$

$$x = x_0 + [(x_1 - x_0) / (y_1 - y_0)] * (y_{max} - y_0)$$

Comparison

● Cohen-Sutherland

- Repeated clipping is expensive
- Best used when trivial acceptance and rejection is possible for most lines

● Cyrus-Beck

- Computation of t-intersections is cheap
- Computation of (x,y) clip points is only done once
- Algorithm doesn't consider trivial accepts/rejects
- Best when many lines must be clipped

● Liang-Barsky: Optimized Cyrus-Beck

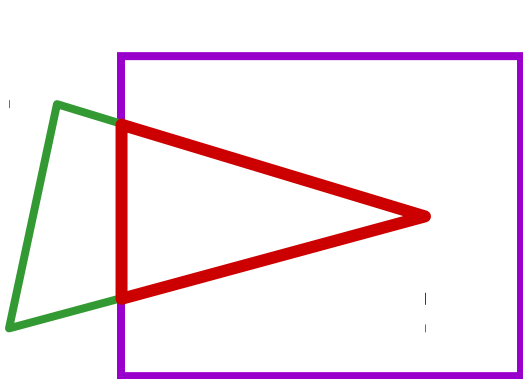
● Nicholl et al.: Fastest, but doesn't do 3D

Clipping Polygons

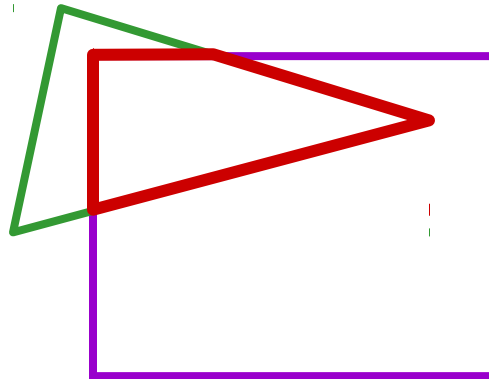
- Clipping polygons is more complex than clipping the individual lines
 - Input: polygon
 - Output: original polygon, new polygon, or nothing
- *When can we trivially accept/reject a polygon as opposed to the line segments that make up the polygon?*

Why Is Clipping Hard?

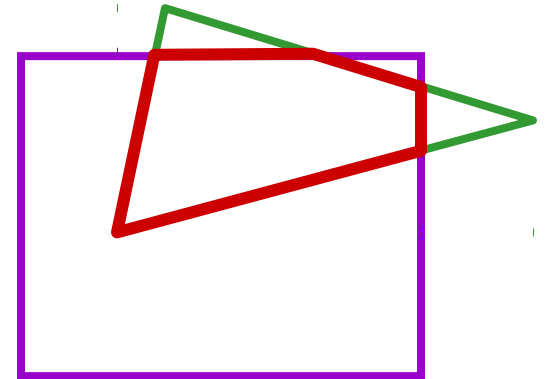
- *What happens to a triangle during clipping?*
- Possible outcomes:



triangle triangle



triangle quad

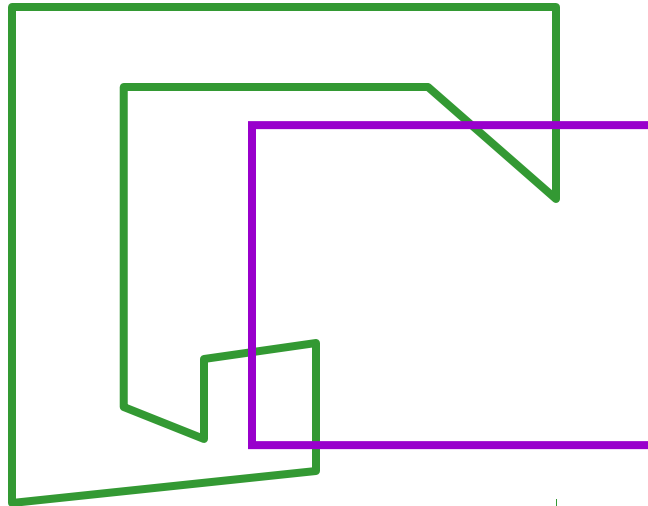


triangle 5-gon

- *How many sides can a clipped triangle have?*

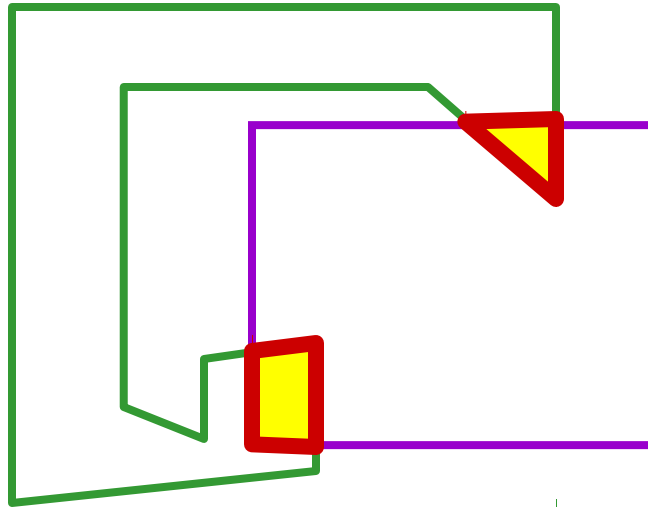
Why Is Clipping Hard?

- A really tough case:



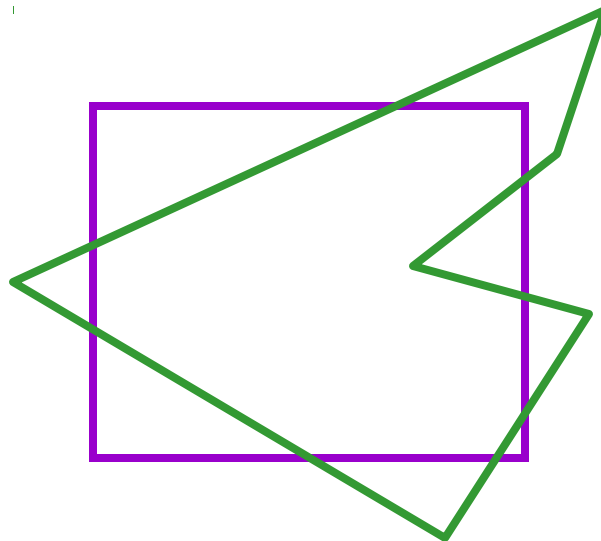
Why Is Clipping Hard?

- A really tough case:



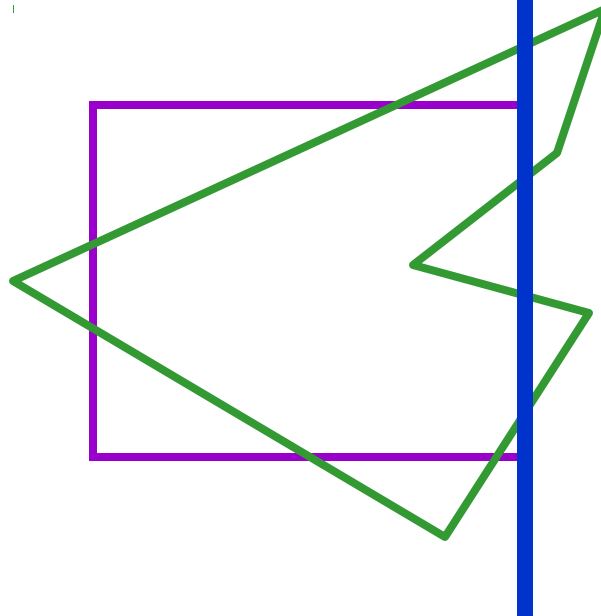
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation



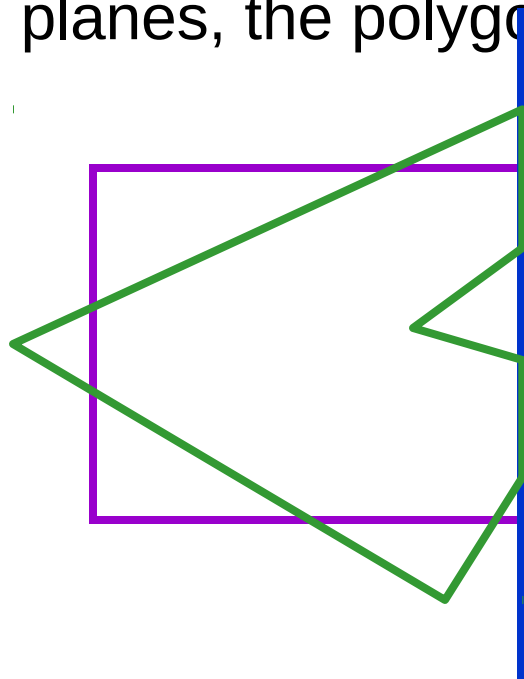
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



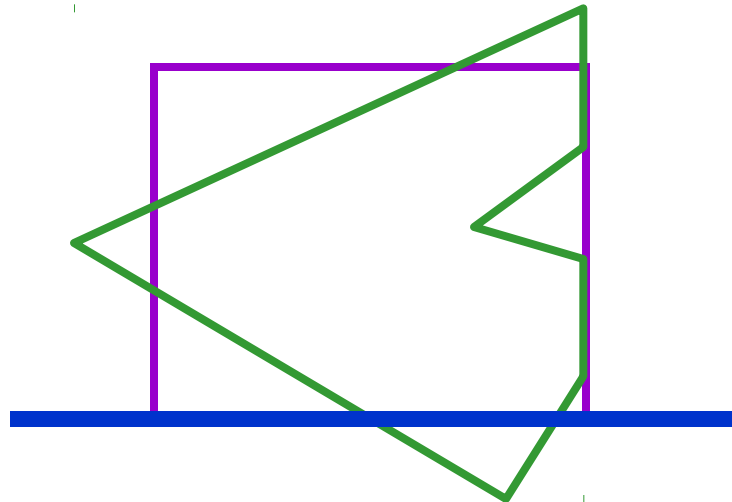
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



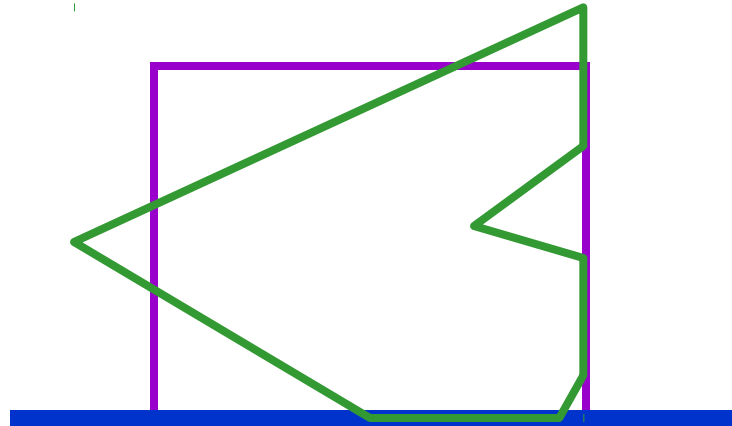
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



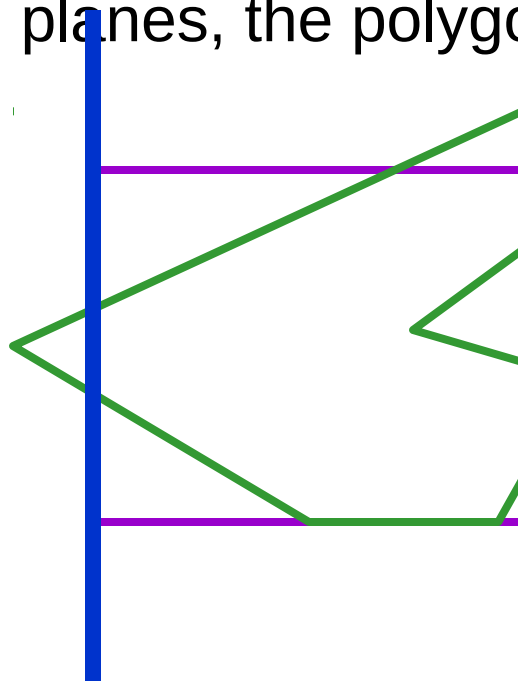
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



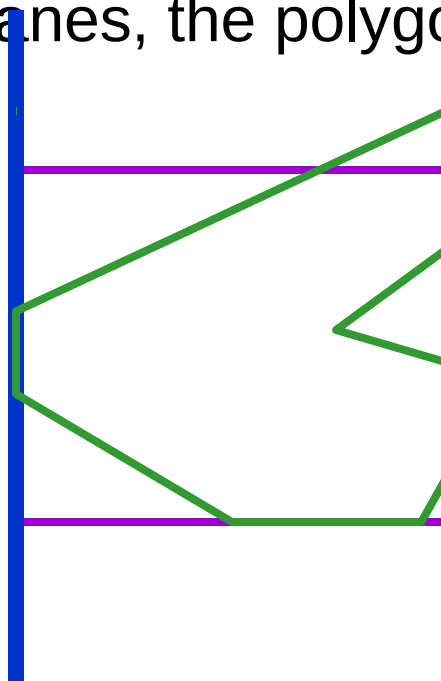
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



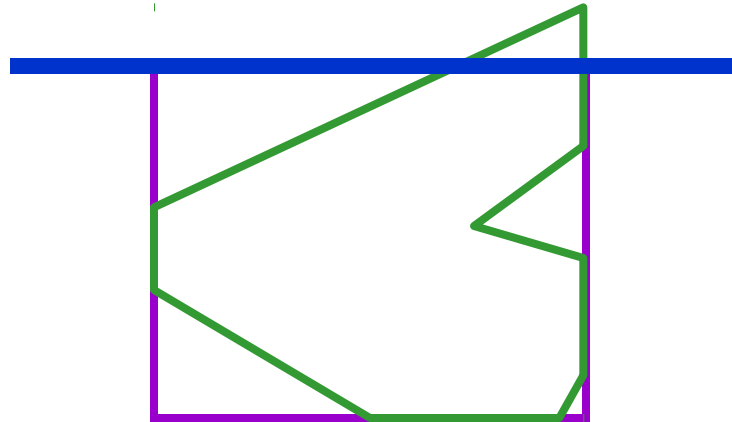
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



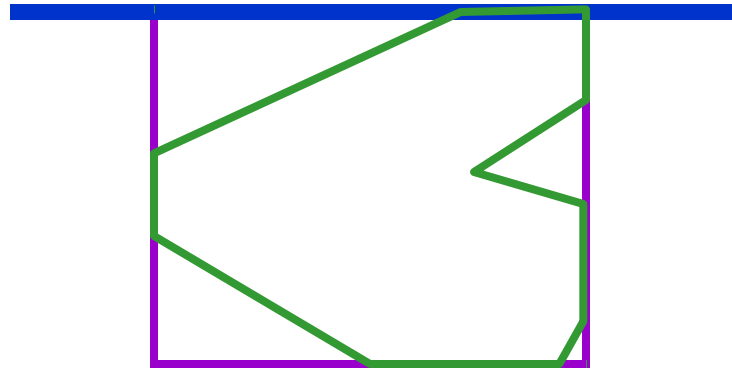
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



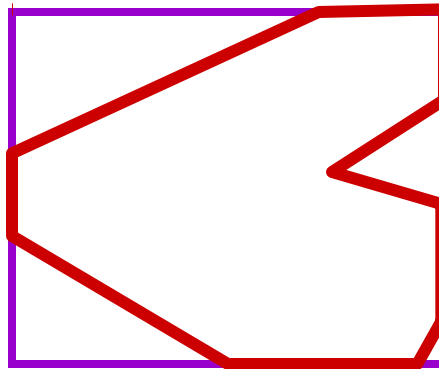
Sutherland-Hodgeman Clipping

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



Sutherland-Hodgeman Clipping: The Algorithm

- Basic idea:
 - Consider each edge of the viewport individually
 - Clip the polygon against the edge equation
 - After doing all planes, the polygon is fully clipped



Sutherland-Hodgeman Clipping

- Input/output for algorithm:
 - Input: list of polygon vertices in order
 - Output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- Note: this is exactly what we expect from the clipping operation against each edge

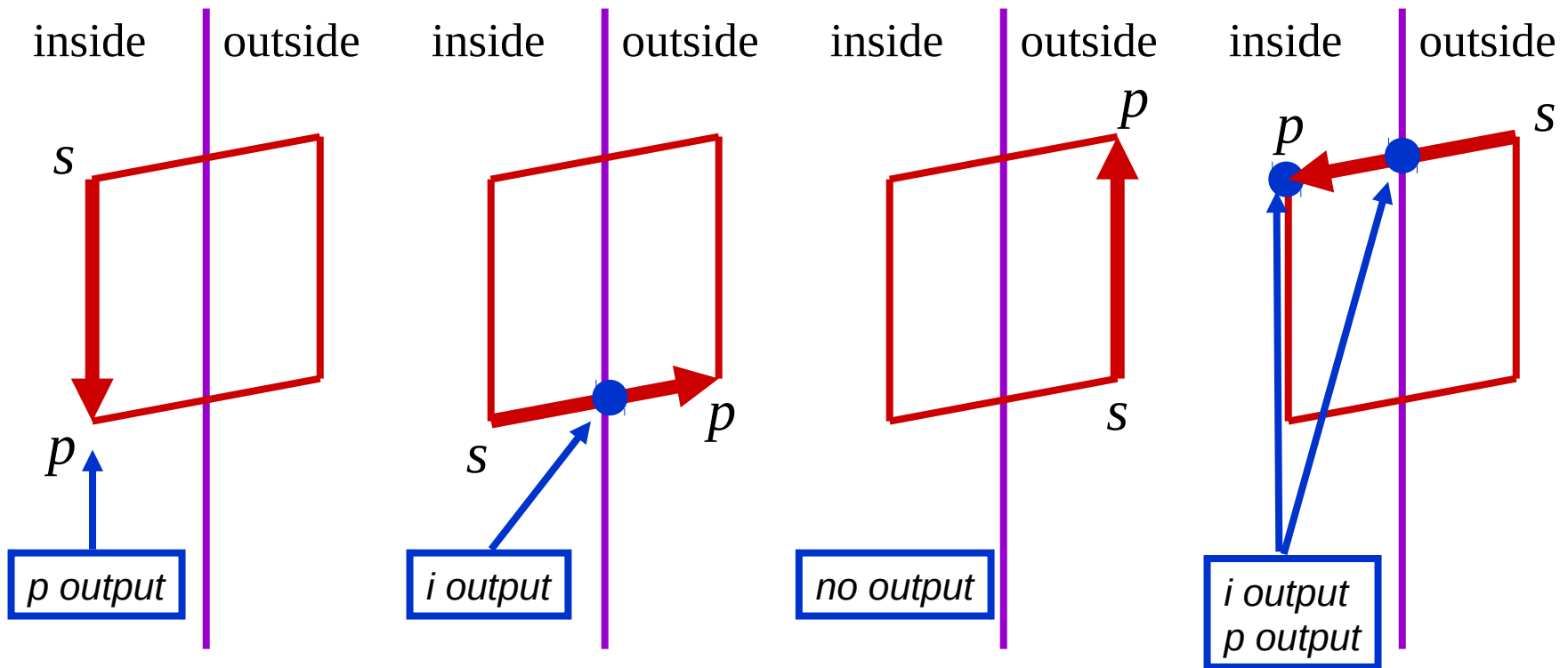
Sutherland-Hodgeman Clipping

- Sutherland-Hodgman basic routine:
 - Go around polygon one vertex at a time
 - Current vertex has position p
 - Previous vertex had position s , and it has been added to the output if appropriate

Sutherland-Hodgeman Clipping

- Edge from s to p takes one of four cases:

(Purple line can be a line or a plane)



Sutherland-Hodgeman Clipping

• Four cases:

- s inside plane and p inside plane
 - Add p to output
 - Note: s has already been added
- s inside plane and p outside plane
 - Find intersection point i
 - Add i to output
- s outside plane and p outside plane
 - Add nothing
- s outside plane and p inside plane
 - Find intersection point i
 - Add i to output, followed by p

Point-to-Plane test

- A very general test to determine if a point p is “inside” a plane P , defined by q and n :

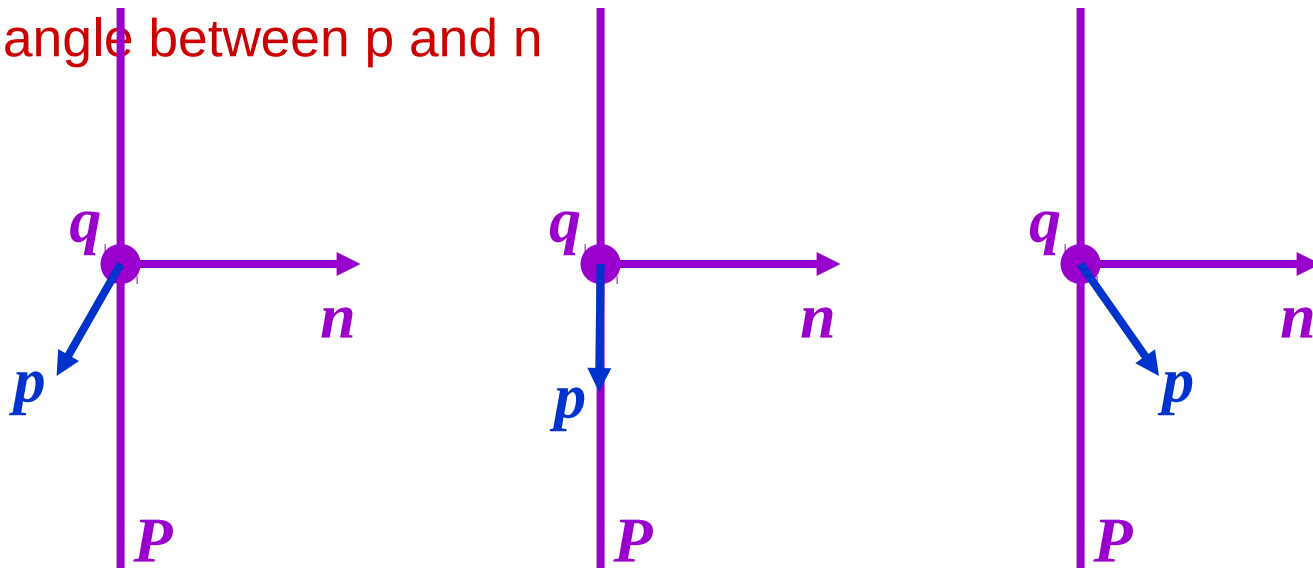
$$(p - q) \cdot n < 0: \quad p \text{ inside } P$$

$$(p - q) \cdot n = 0: \quad p \text{ on } P$$

$$(p - q) \cdot n > 0: \quad p \text{ outside } P$$

Remember: $p \cdot n = |p| |n| \cos(\theta)$

θ = angle between p and n



Finding Line-Plane Intersections

- Use parametric definition of edge:

$$\mathbf{L}(t) = \mathbf{L}_0 + (\mathbf{L}_1 - \mathbf{L}_0)t$$

- If $t = 0$ then $\mathbf{L}(t) = \mathbf{L}_0$
- If $t = 1$ then $\mathbf{L}(t) = \mathbf{L}_1$
- Otherwise, $\mathbf{L}(t)$ is part way from \mathbf{L}_0 to \mathbf{L}_1

Finding Line-Plane Intersections

- Edge intersects plane P where $E(t)$ is on P
 - q is a point on P
 - n is normal to P

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

$$t = [(q - L_0) \cdot n] / [(L_1 - L_0) \cdot n]$$

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

Line-Plane Intersections

- Note that the length of \mathbf{n} doesn't affect result:
- Again, lots of opportunity for optimization

An Example with a non-convex polygon

