Juraj Onderik | onderik@sccg.sk

Clipping

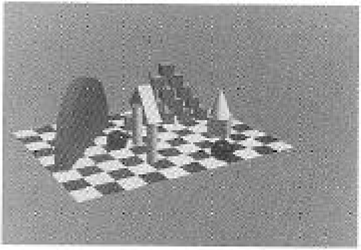
Lesson 06

Outline of Lesson 06

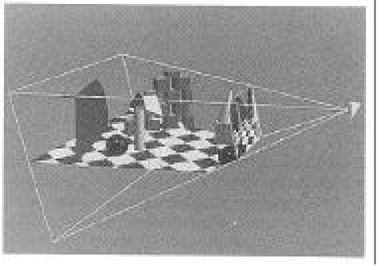
- Line clipping algorithms in the CG Pipeline
- Cohen-Sutherland
- Cyrus-Beck
- Nicholl-Lee-Nicholl

Transformations

- **Scene composition**
  - World space



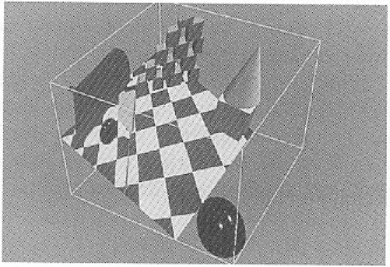
- **Viewing frustrum**
  - Eye position, orientation

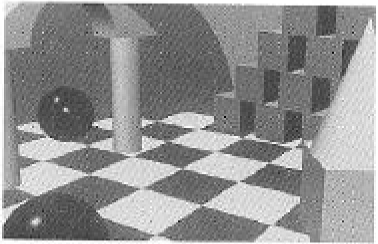


Transformation

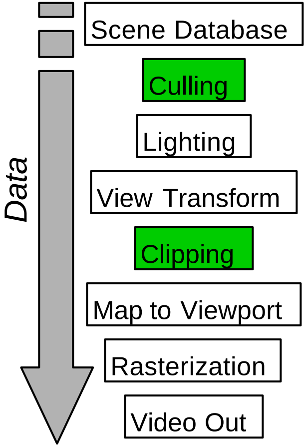
- **3D Screen space**
  - Clipped to frustrum
  - Distortion towards far clipping plane
  - Z-buffer occlusion detection

- **Projection to 2D**





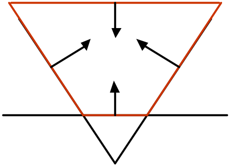
Where Culling & Clipping Fit In



- **Goal #1: Reject objects as early as possible**
  - this will save the most work
- **Goal #2: Rejection test must be efficient**
  - we're trying to *avoid* work
- **Generally perform culling early on**
  - remove objects wholly outside frustum
  - avoids lighting & transformation
- **And perform clipping later on**
  - cut off parts outside viewport
  - simplifies rasterization

View Frustum Culling

- **Discard any object outside viewing volume early on**
  - performed by application (or application framework)
- **Viewing volume is formed by 6 planes**
  - suppose all normals are oriented towards interior
  - then the interior is set of all points such that

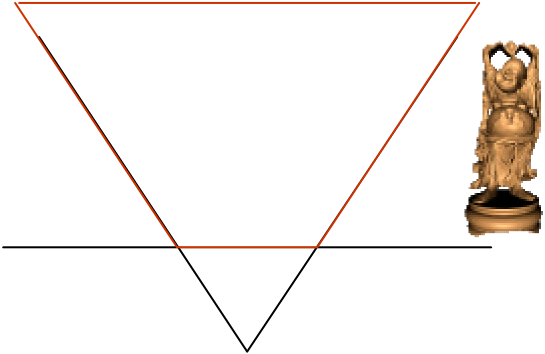


$$a_i x + b_i y + c_i z + d_i \geq 0$$

- **Given a set of polygons**
  - test for intersection with viewing volume
  - any polygon not intersecting frustum can be culled
- **What's wrong with this simple algorithm?**

Inefficient Per-Polygon Processing

- **What if a million polygon object is entirely outside frustum?**
  - we certainly don't want to test every one!

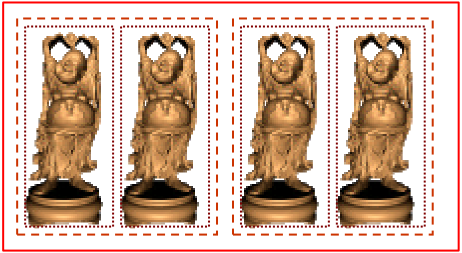


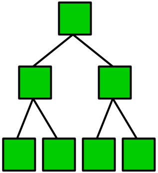
Culling with Bounding Volumes

- **Let's enclose our object in a *convex* volume**
  - bounding sphere
    - compact representation
    - may not fit object tightly
  - bounding box
    - axis-aligned or oriented with object
  - convex polytope
    - allows tightest fit
    - most expensive to deal with
- **Now test bounding volume first**
  - if outside frustum, reject object
  - otherwise visit individual components

Hierarchical Bounding Volumes

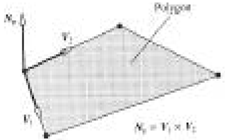
- **And we can do even better with a hierarchy of volumes**
- **Begin testing at the root node**
  - if outside, reject all objects
  - otherwise, recursively test sub-nodes
- **Of course this raises the question: how best to build this hierarchy?**





Backface Culling

- **Even for polygons inside frustum, some may be culled**
  - if we assume that our objects are closed



- **Consider polygon normal**

$$N_P = V_1 \times V_2$$

  - Oriented polygon edges V_1, V_2
  - if it's pointing towards the eye, we *may* be able to see it
  - pointing away means it's on the opposite side of the object

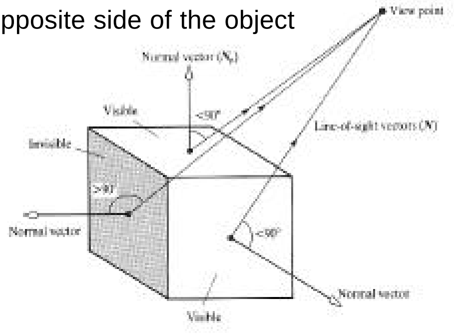
- **Line-of-sight vector** N

$$N_P \bullet N$$

> 0 : surface visible

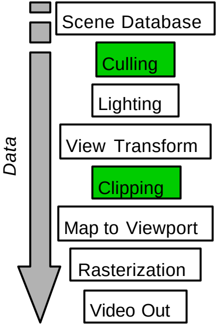
< 0 : surface not visible

⇒ Draw only visible surfaces

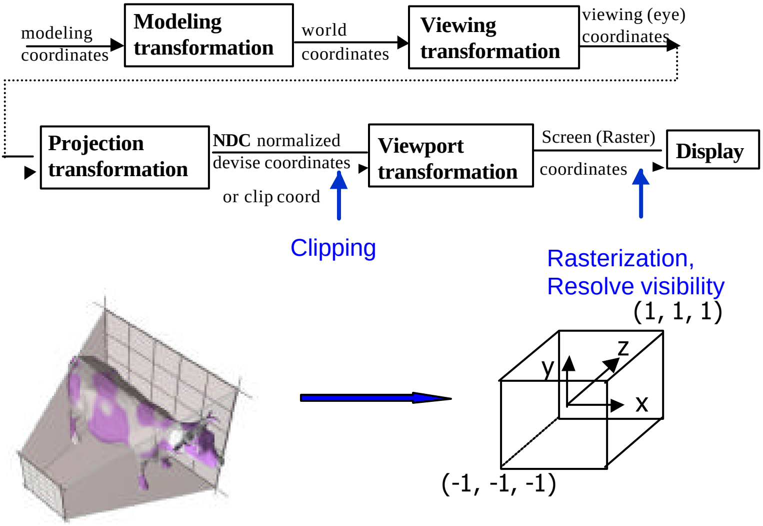


From Culling to Clipping

- **Culling tries to reject objects wholly outside viewing volume**
  - typically done by application
  - happens prior to lighting, transformation, …
- **Now, we want to cut off pieces outside frustum**
  - this is clipping
- **Clipping happens just prior to rasterization**
  - almost always done by graphics system
  - frequently implemented in hardware

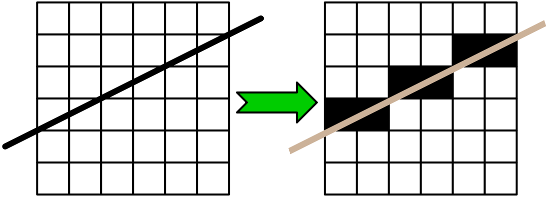


Transformations & Clipping



Why not Per-Pixel Clipping during Rasterization ?

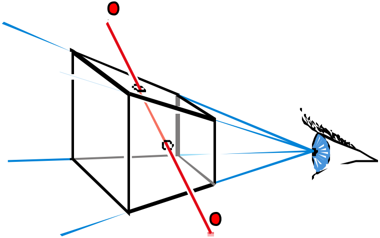
- **During rasterization, we visit every pixel covered by primitive**
  - if any pixel is outside the viewport, reject it

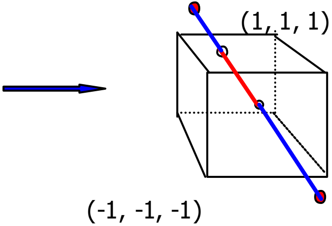


- **What's wrong with this?**
- **It can be pretty inefficient**
  - suppose a 1000 pixel polygon is completely outside viewport

Clipping

- After the mapping of the view volume (a frustum for perspective views; parallelepiped for orthographic views) to the canonical view volume. All vertices are in NDC.
- Primitives not within the canonical view volume are to be clipped. Clipping is more efficient and faster when carried out with NDC.

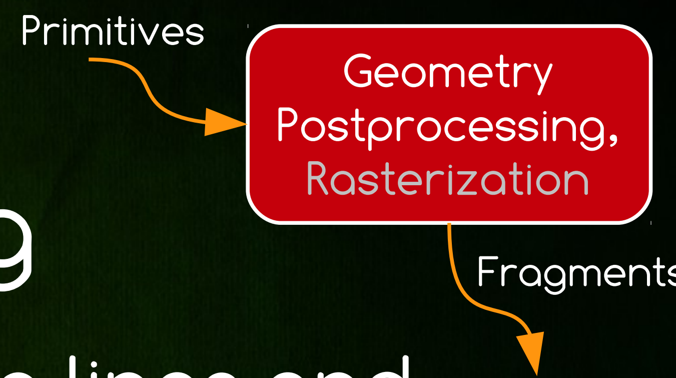




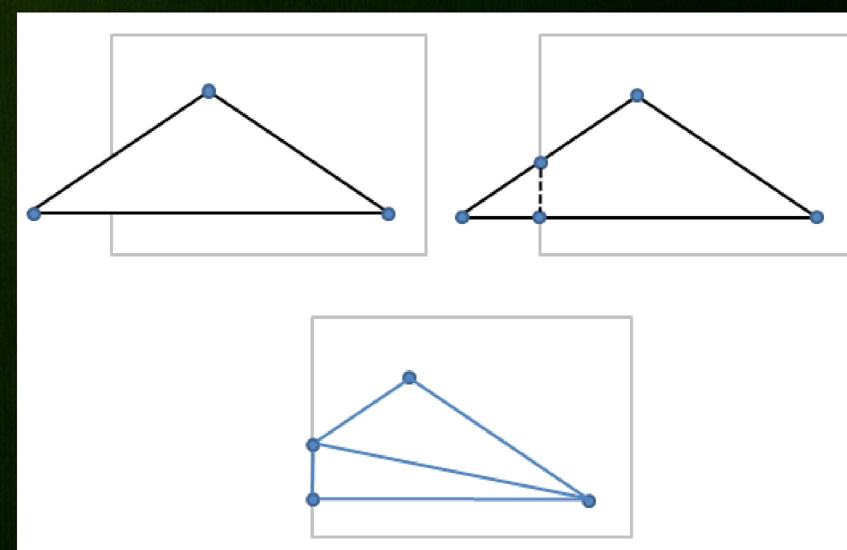
Point Clipping (Culling)

- **In 3D view space**
- **Vertex inside canonical view frustrum ?**
  - OpenGL: x,y,z [-1…1]
  - Direct3D: x,y [-1…1], z [0…1]

The CG Pipeline
Geometry Postprocessing



- During geometry postprocessing lines and triangles are clipped against the window
  - We can not write outside the frame buffer
- Clipping should be
  - Fast for many primitives
  - Implemented on HW (GPU)

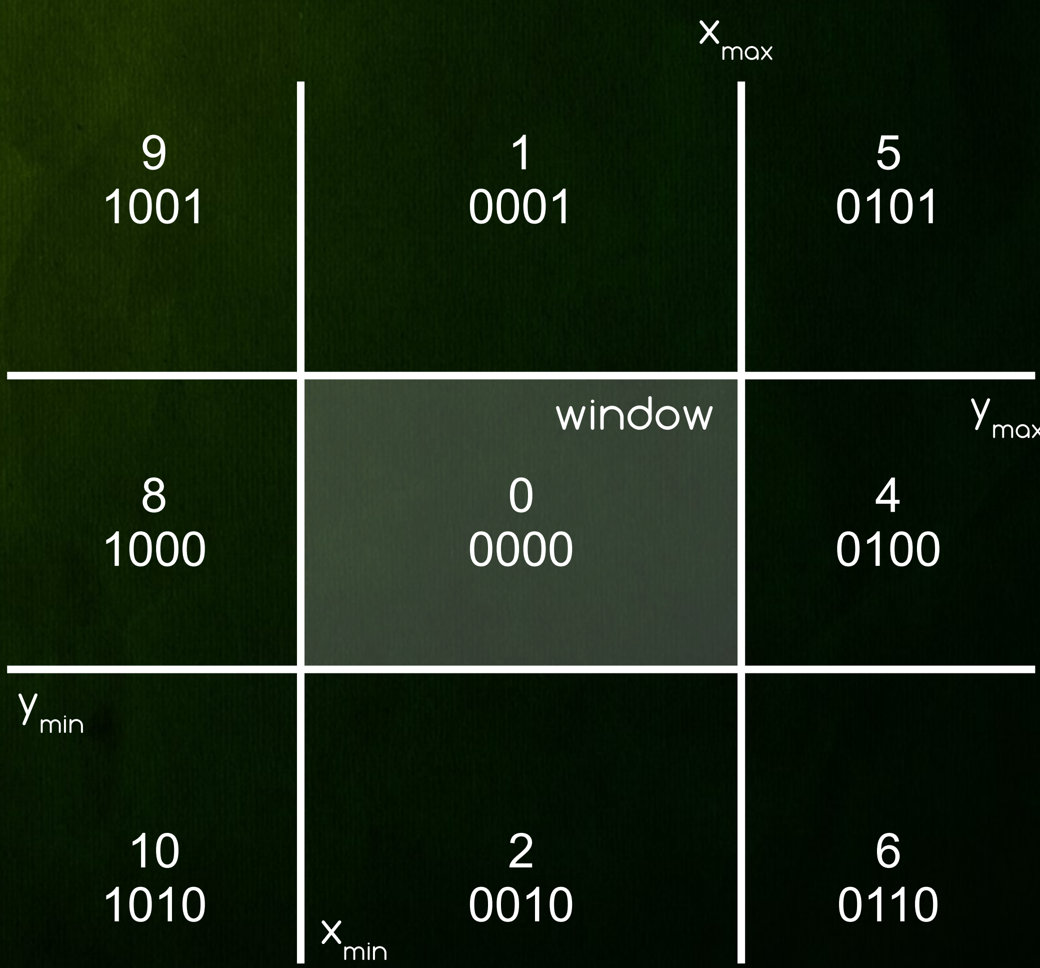


Cohen-Sutherland

- Main Purpose
  - Clipping lines against rectangular (axis aligned) 2D (3D) window
- Algorithm Principle
  - Divides a 2D (3D) space into 9 (27) regions
  - Efficiently determine the (portions of) lines that are visible in the window
  - Clip lines against window edges

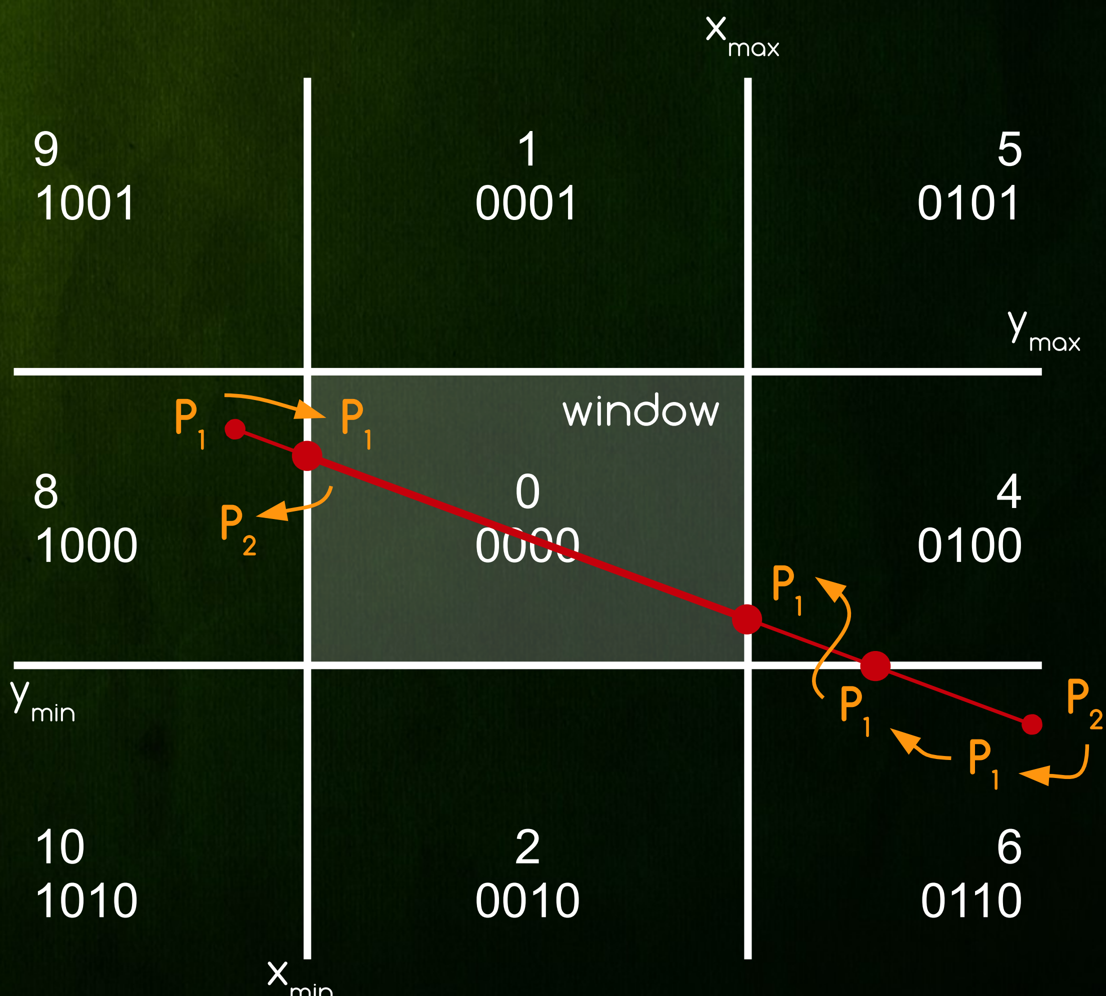
Cohen-Sutherland

- 9 codes (4bit) for each region: code = $b_3b_2b_1b_0$
- X cases
  - $b3 = (x < x_{min})\ ?\ 1 : 0$
  - $b2 = (x > x_{max})\ ?\ 1 : 0$
- Y Cases
  - $b1 = (y < y_{min})\ ?\ 1 : 0$
  - $b0 = (y > y_{max})\ ?\ 1 : 0$



Cohen-Sutherland

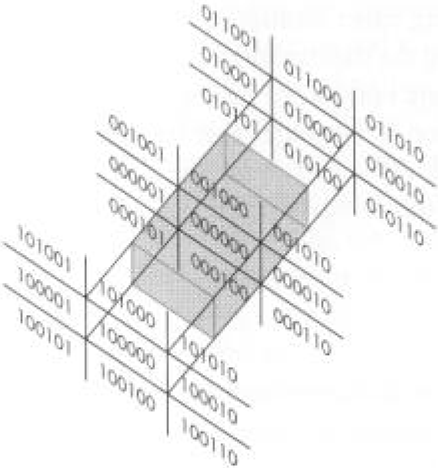
- Execution example
  - Clip P_1 against x_{min}
  - Swap P_1 and P_2
  - Clip P_1 against y_{min}
  - Clip P_1 against x_{max}
  - Done with P1P2



Cohen-Sutherland

```
c2 = code(x2, y2);
while (false) {
    c1 = code(x1, y1);
    if (c1 & c2 != 0) return false;
    else if (c1 | c2 == 0) return true;
    else {
        if (c1 == 0) { swap(x1, x2); swap(y1, y2); swap(c1, c2); }
        else if (c1 ∈ {1, 5, 9} ) { x1 = x1 + (x2-x1) * (ymax-y1) / (y2-y1); y1 = ymax; }
        else if (c1 ∈ {2, 6, 10} ) { x1 = x1 + (x2-x1) * (ymin-y1) / (y2-y1); y1 = ymin; }
        else if (c1 ∈ {4, 5, 6} ) { y1 = y1 + (y2-y1) * (xmax-x1) / (x2-x1); x1 = xmax; }
        else if (c1 ∈ {8, 9, 10} ) { y1 = y1 + (y2-y1) * (xmin-x1) / (x2-x1); x1 = xmin; }
    }
}
```

OutCode in 3D

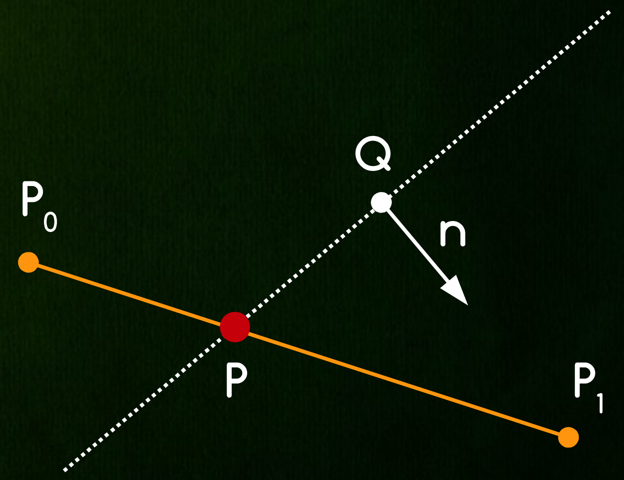


Cyrus-Beck

- Main Purpose
  - Clipping lines against any convex polygon
- Algorithm Principle
  - Find line parameter of intersection with each edge of polygon
  - Update min and max line parameter to be inside the halfspace of each edge
  - If min < max calculate clipped line segment points

Cyrus-Beck

- Intersection of hyperplane and line segment
  - Hyperplane (origin O, normal n)
  - Line segment (start point P0, end point P1)
- P lies on line segment
  - P = P0 + t (P1 - P0) | 0 <= t <= 1
- P lies on hyperplane
  - (P - Q) * n = 0



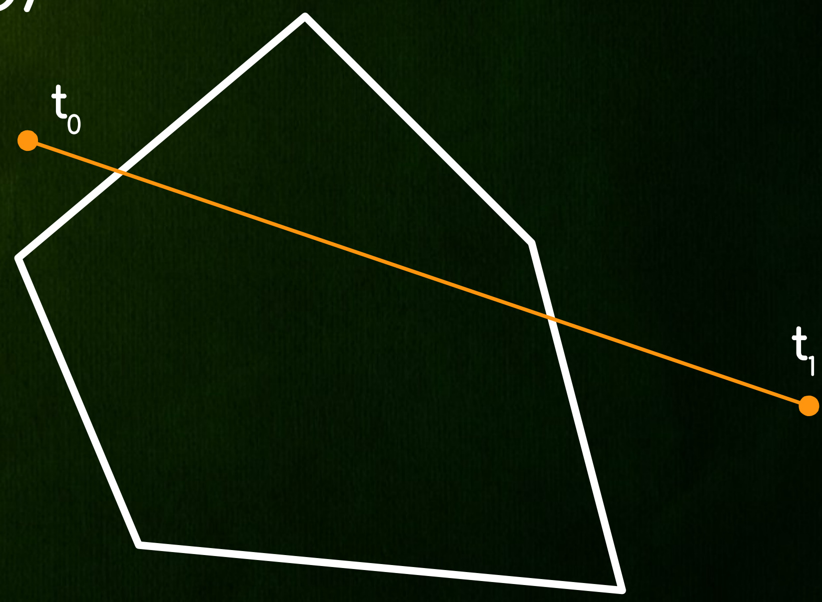
- Solve t = (Q-P0) * n / (P1-P0) * n
  - dq = (Q-P0) * n | d1 = (P1-P0) * n → **t = dq / d1**

Cyrus-Beck

- Instead of calculating new intersected points Cyrus-Beck operates only on line parameters t0 and t1 - this is faster
- First set t0 = 0 and t1 = 1 (original line segment)
- For each edge find intersection parameter t and set
  - If (d1 > 0) t0 = max(t, t0) (out-to-in case)
  - If (d1 < 0) t1 = min(t,t1) (in-to-out case)
- This will find the smallest intersection interval
- At the end find new P0 and P1 for t0 and t1

Cyrus-Beck

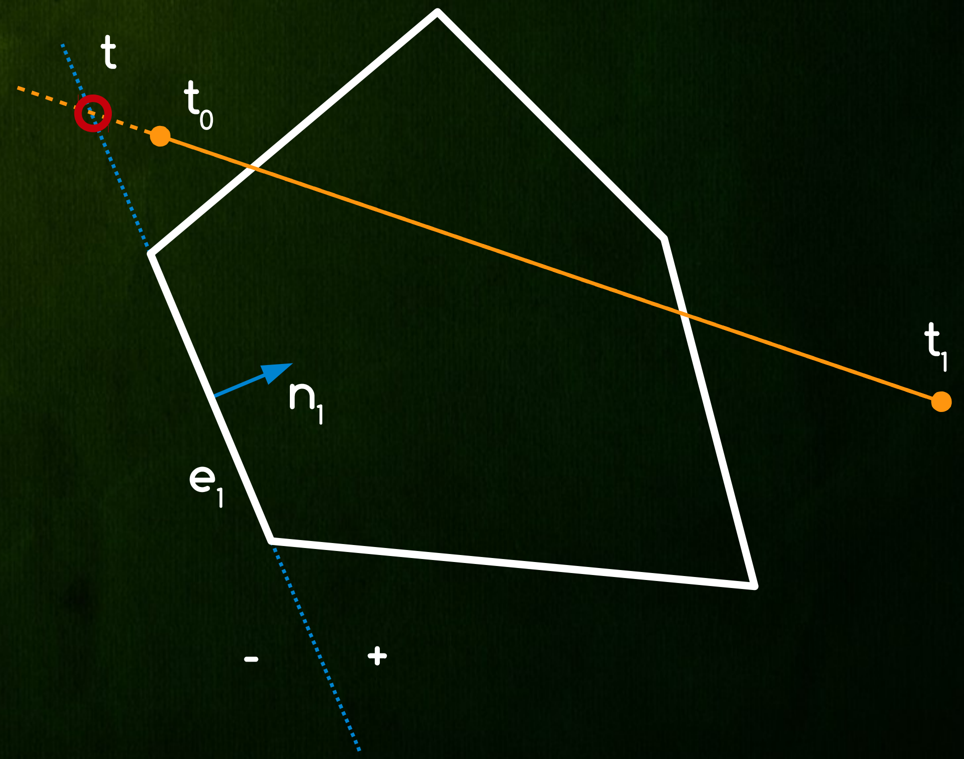
- Input: Convex polygon and line segment
- Output: Clipped line segment being fully inside given polygon (or nothing)



- Set clipping parameters
  - ➔ t0 = 0, t1 = 1

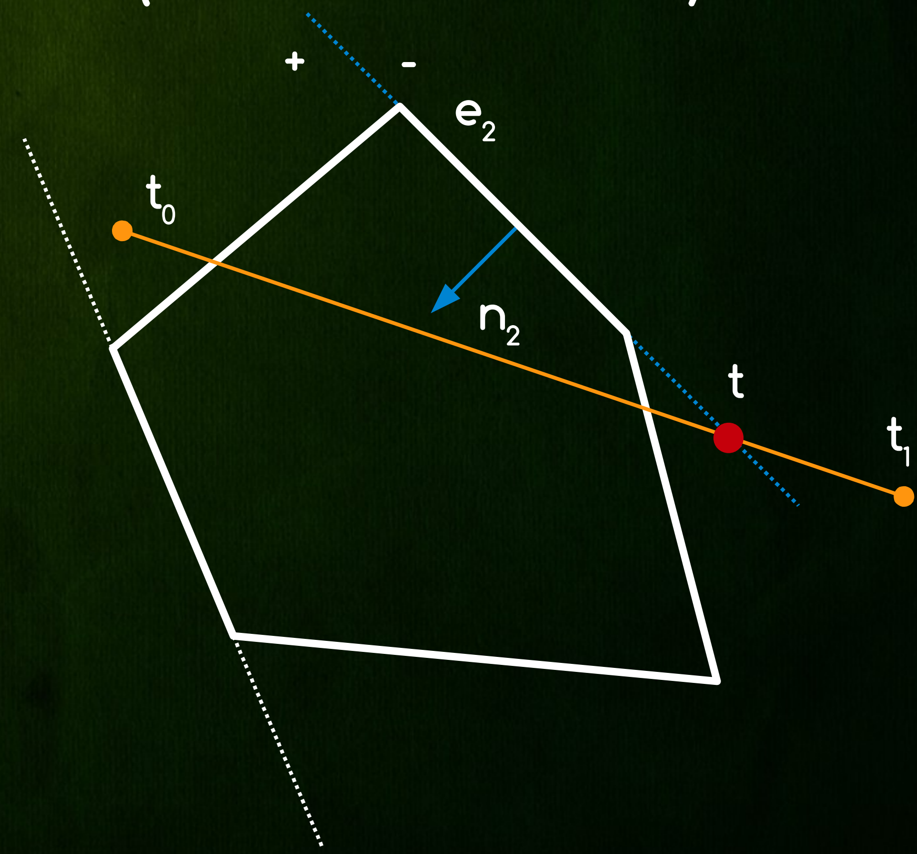
Cyrus-Beck

- Find intersection parameter t with edge e1
- d1 = (P1-P0) *n1 > 0 → clip t0 (out-to-in case)
- t0 = max(t,t0)
  - Since t < t0
  - No update is done



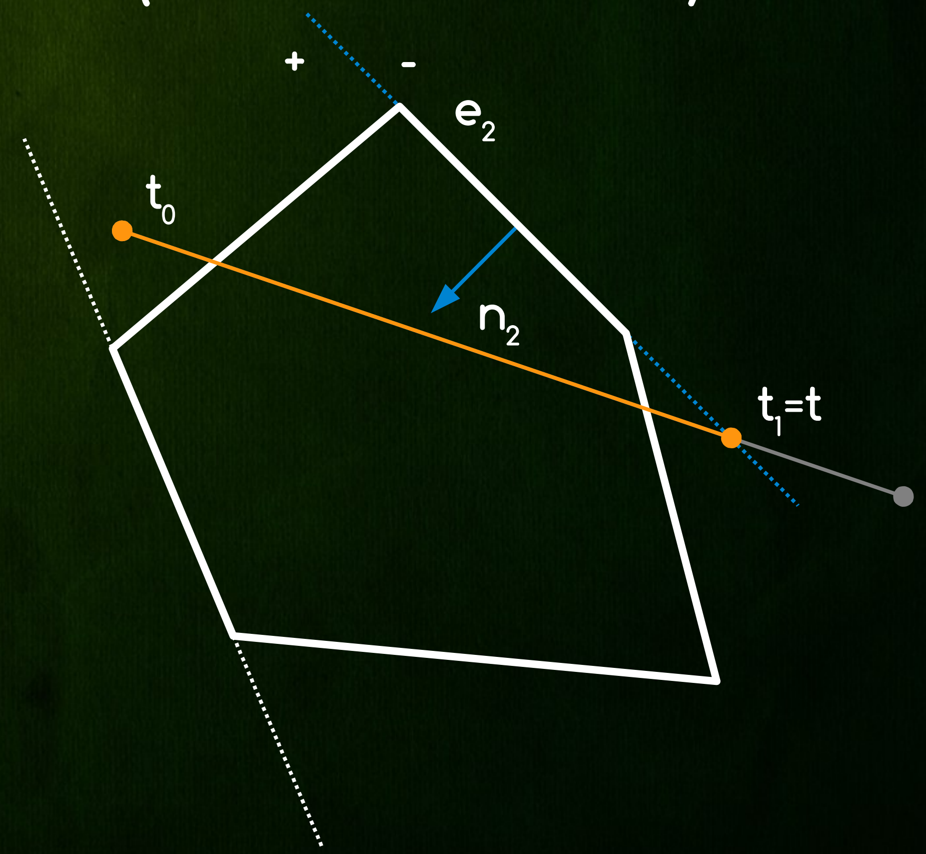
Cyrus-Beck

- Find intersection parameter t with edge e2
- d1 = (P1-P0) *n2 < 0 → clip t1 (in-to-out case)



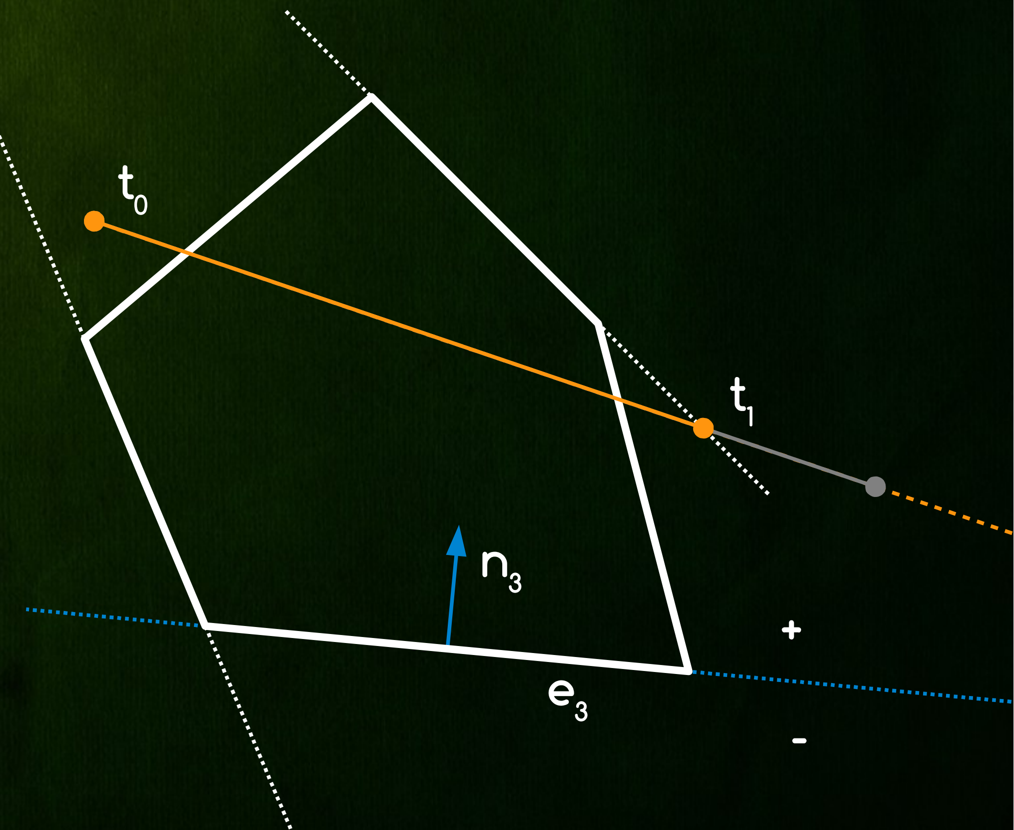
Cyrus-Beck

- Find intersection parameter t with edge e2
- d1 = (P1-P0) *n2 < 0 → clip t1 (in-to-out case)
- t1 = min(t,t1)
  - Since t < t1
  - We update t1 = t



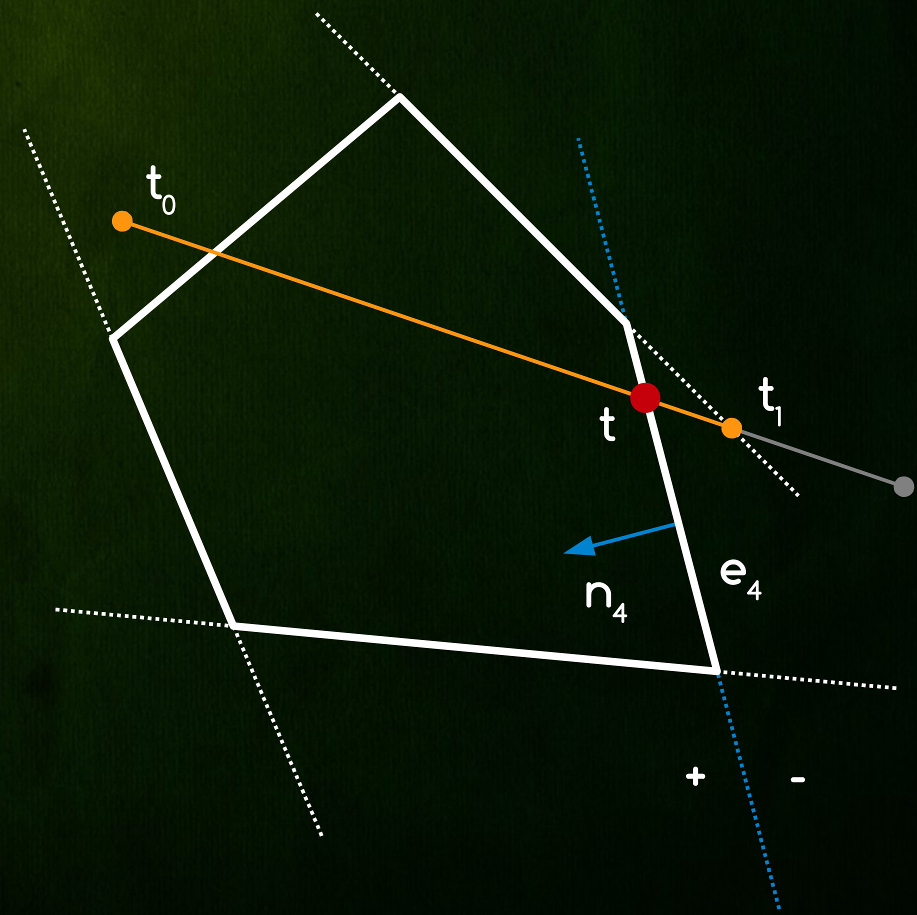
Liang-Barsky

- Find intersection parameter t with edge e3
- d1 = (P1-P0) *n3 < 0 → clip t1 (in-to-out case)
- t1 = min(t,t1)
  - Since t > t1
  - No update is done



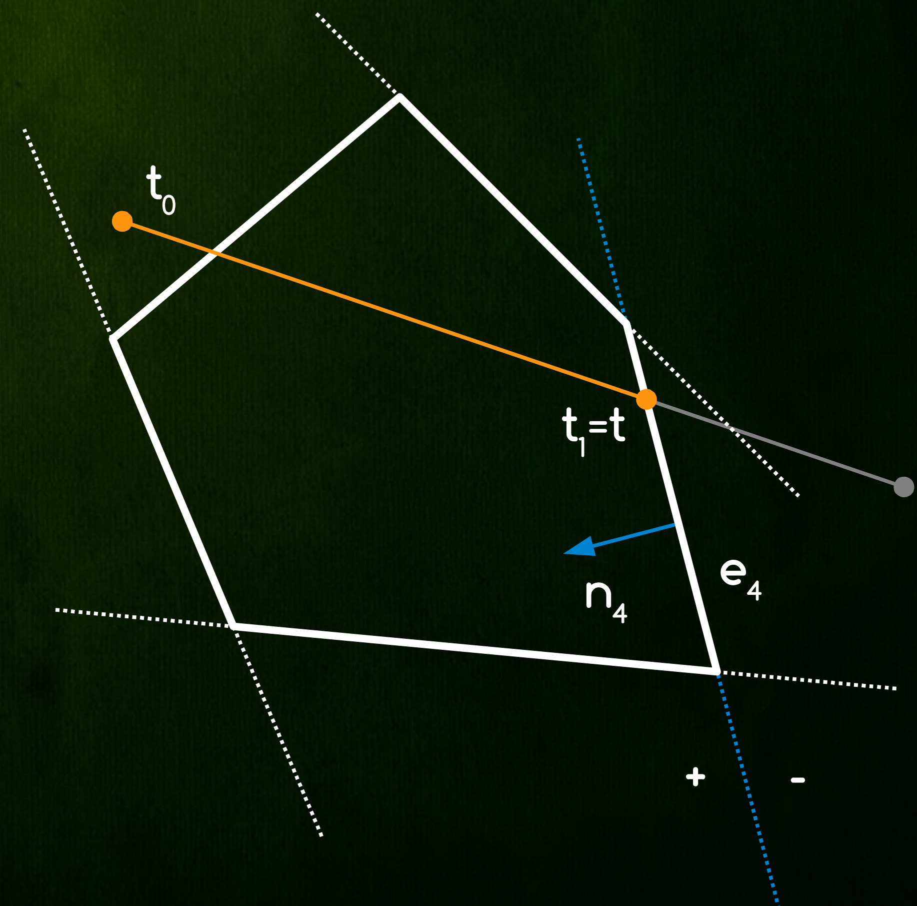
Cyrus-Beck

- Find intersection parameter t with edge e4
- d1 = (P1-P0) *n4 < 0 → clip t1 (in-to-out case)



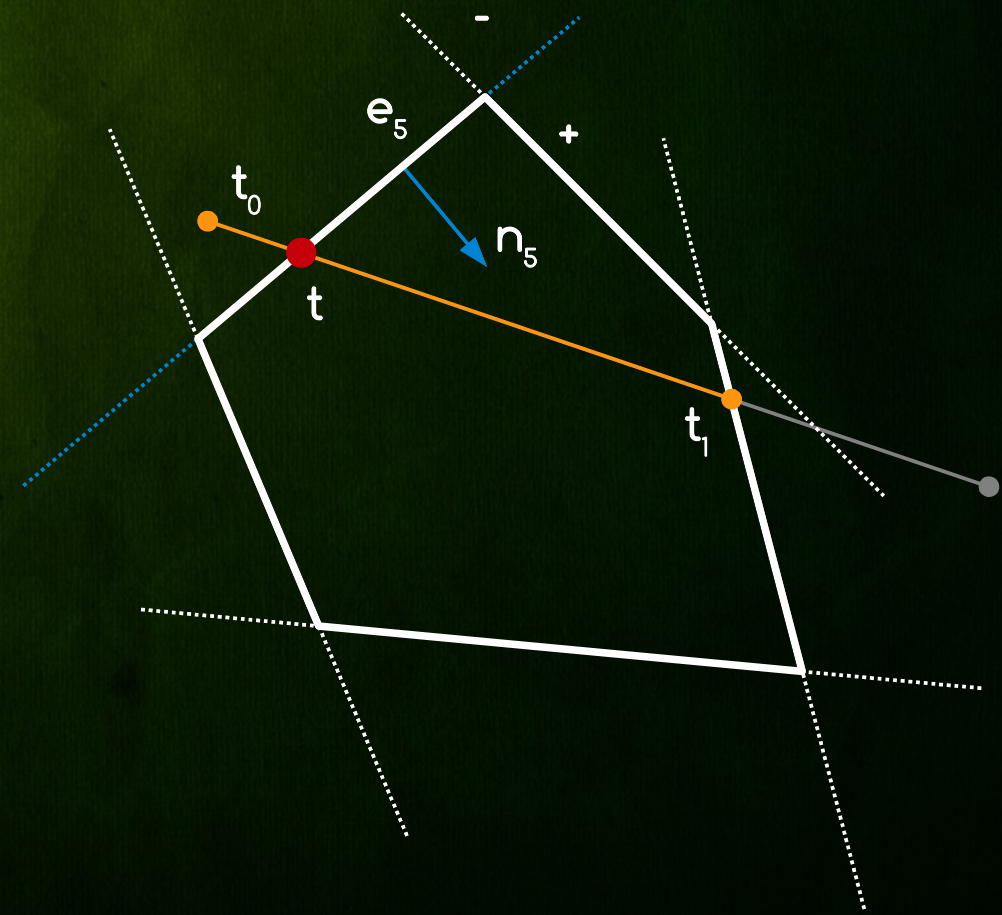
Cyrus-Beck

- Find intersection parameter t with edge e4
- d1 = (P1-P0) *n4 < 0 → clip t1 (in-to-out case)
- t1 = min(t,t1)
  - Since t < t1
  - We update t1 = t



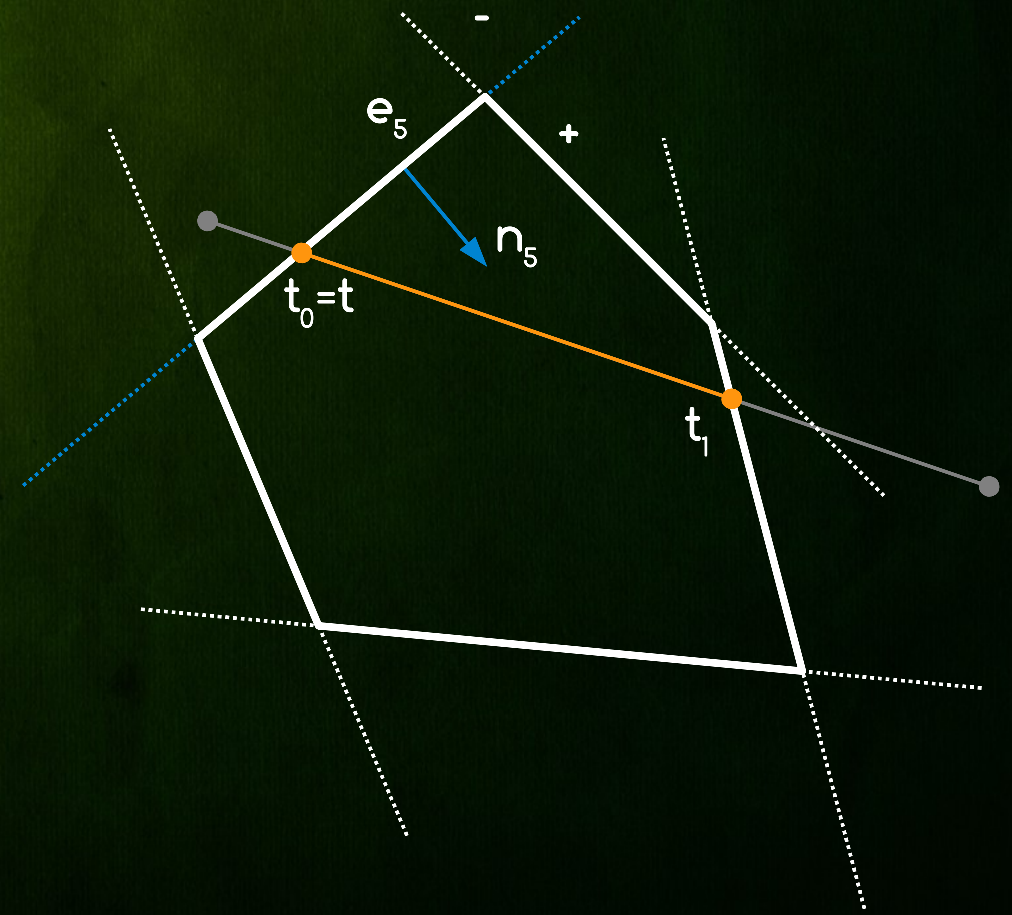
Cyrus-Beck

- Find intersection parameter t with edge e5
- d1 = (P1-P0) *n5 > 0 → clip t0 (out-to-in case)



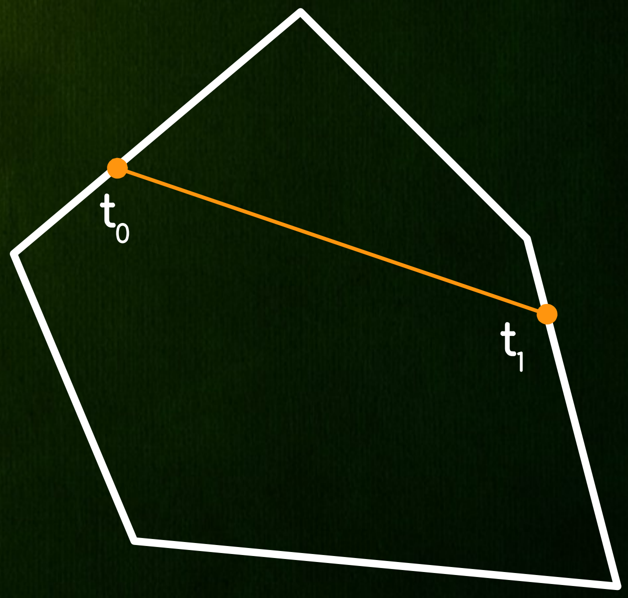
Cyrus-Beck

- Find intersection parameter t with edge e5
- d1 = (P1-P0) *n5 > 0 → clip t0 (out-to-in case)
- t0 = max (t,t0)
  - Since t > t0
  - We update t0 = t



Cyrus-Beck

- No more edges to update with
- If t0 > t1 whole line segment is outside of polygon
- If t0 <= t1 clip line
  - P0' = P0 + t0 (P1-P0)
  - P1' = P0 + t1 (P1-P0)



Cyrus-Beck

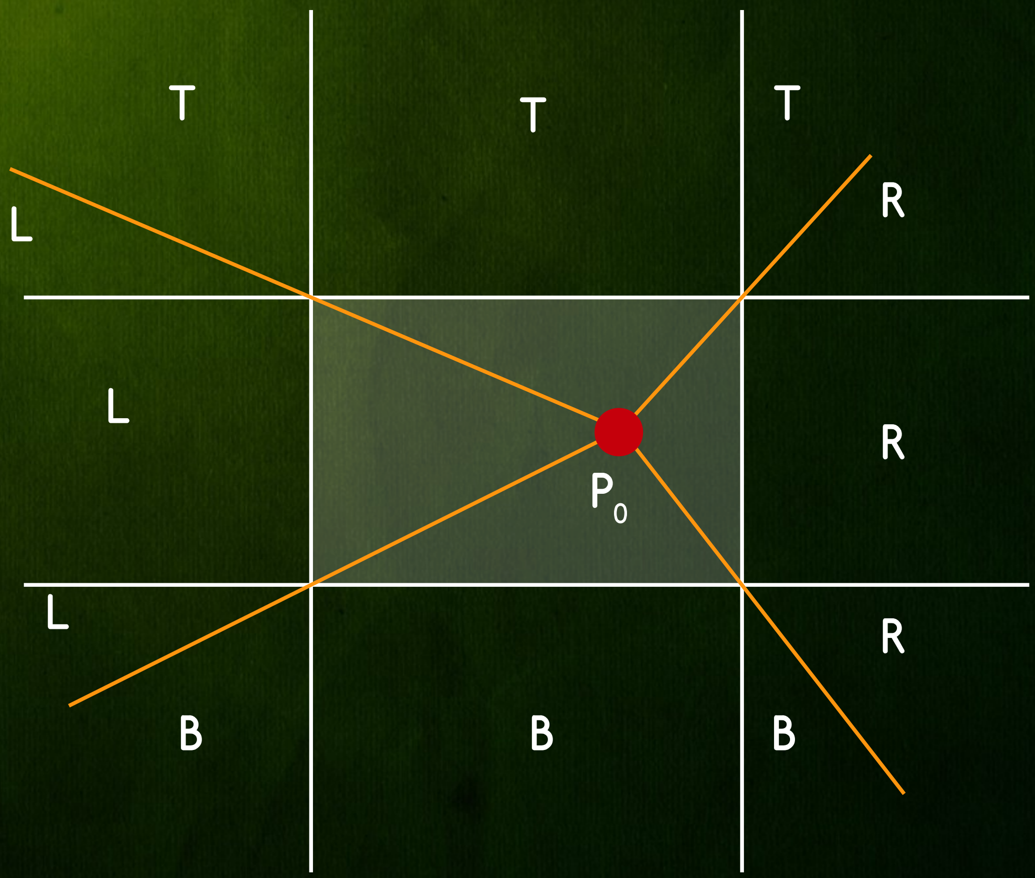
- $t_0 = 0; t_1 = 1;$
- foreach edge $e_i = (q_i, n_i)$ {
  - $d_1 = (p_1 - p_0) * n_i;$ $d_q = (q_i - p_0) * n_i;$
  - if $(d_1 > 0)$ { $t = d_q / d_1; t_0 = max(t, t_0);$ } else
  - if $(d_1 < 0)$ { $t = d_q / d_1; t_1 = min(t, t_1);$ } else
  - if $((p_0 - q_i) * n_i < 0)$ return false;   // line is outside of poly
- }
- if $(t_0 < t_1)$ return true; else return false;

Nicholl-Lee-Nicholl

- Main Purpose
  - Clipping lines against rectangular (axis aligned) 2D only window
- Algorithm Principle
  - Categorize first point of line segment similarly to Cohen-Sutherland
  - Virtual cast 4 rays from P0 through 4 corners of window and categorize all regions between rays. In each segment we know which window edges we have to clip with
  - Clip line segment with selected edges

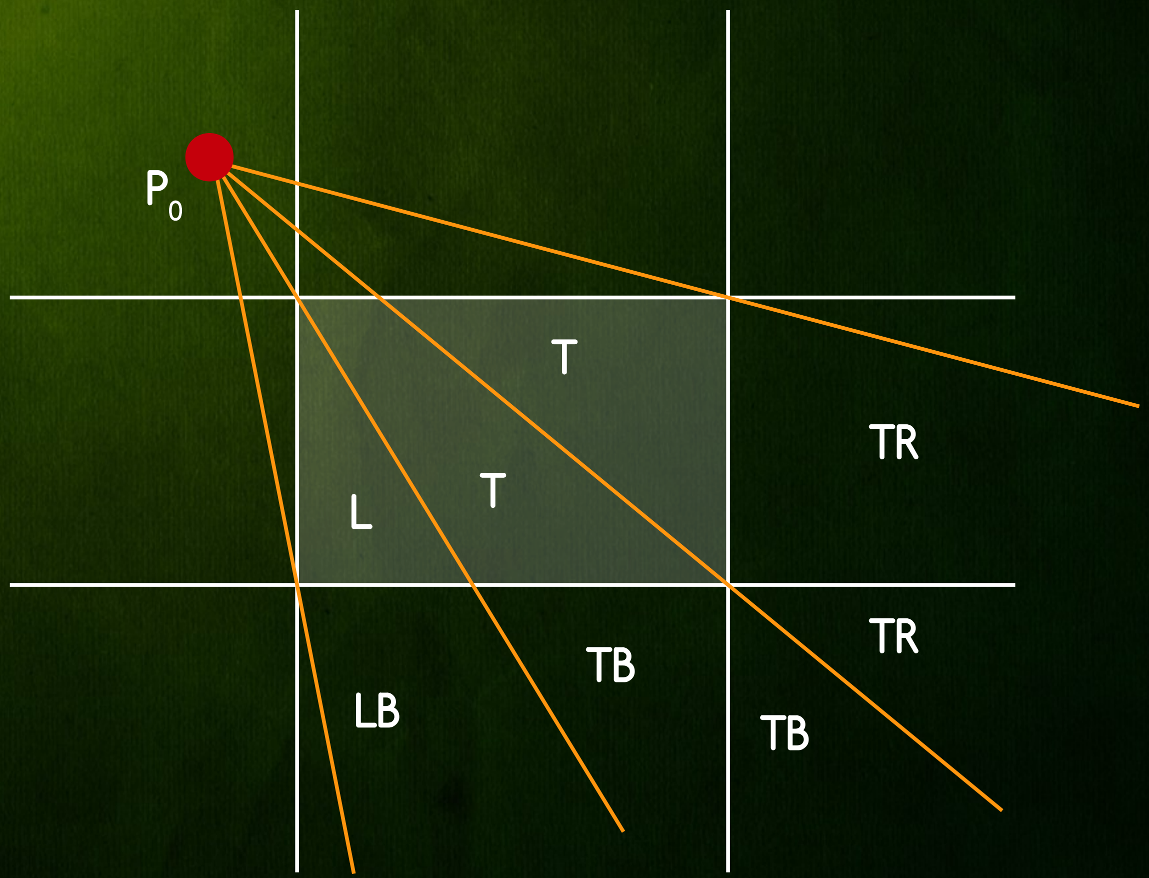
Nicholl-Lee-Nicholl

- Window region



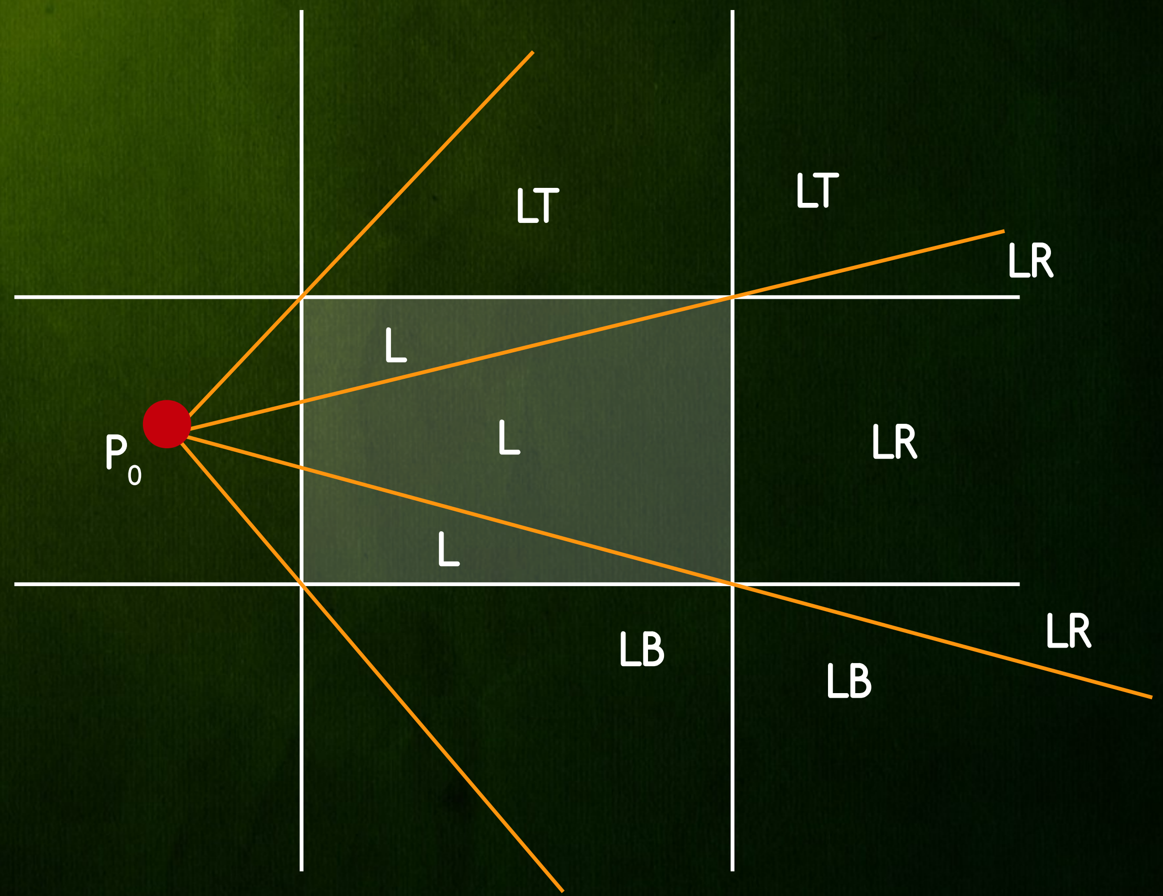
Nicholl-Lee-Nicholl

- Corner region



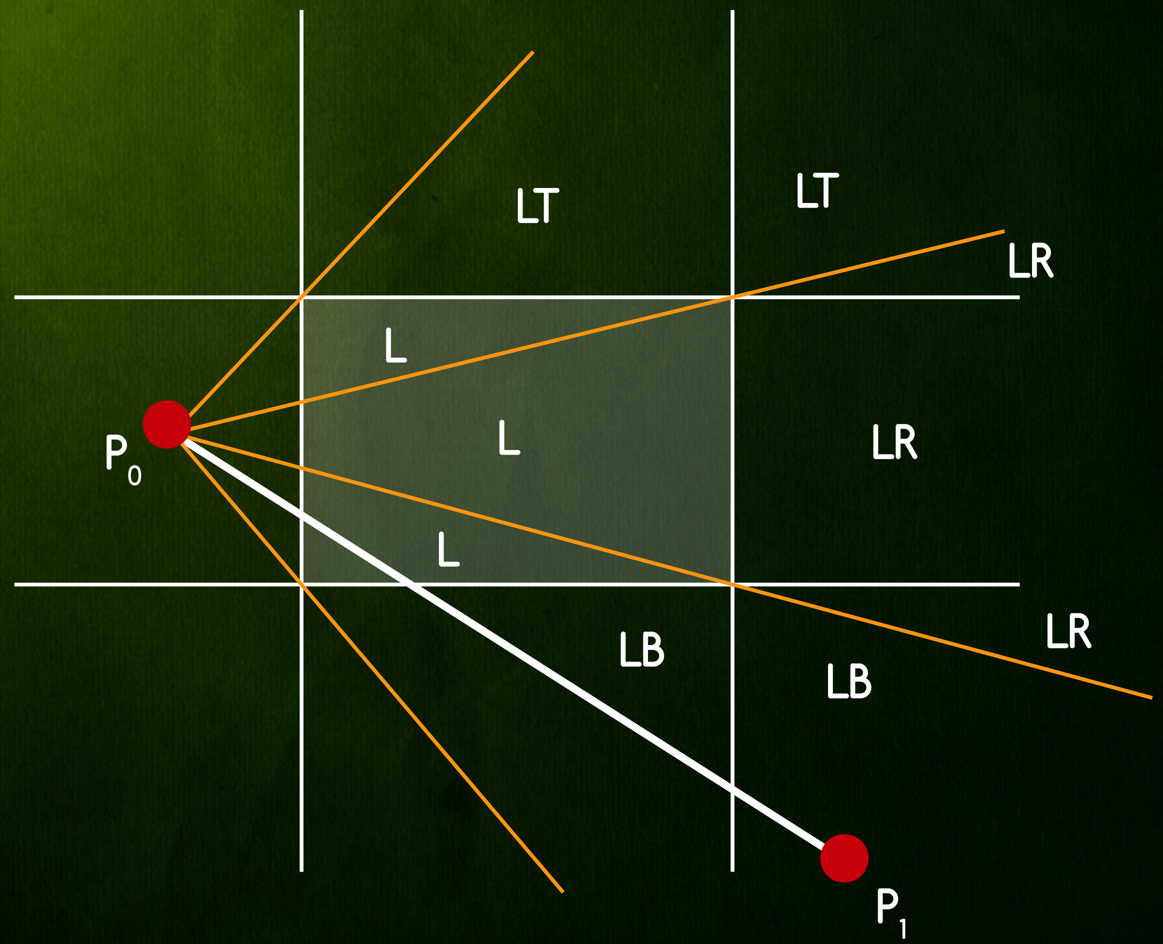
Nicholl-Lee-Nicholl

- Edge region



Nicholl-Lee-Nicholl

- Edge region Example



Nicholl-Lee-Nicholl

```
procedure LeftEdgeRegionCase (ref real x1, y1, x2, y2; ref boolean visible)
begin
    real dx, dy;

    if  x2 < xmin
        then  visible := false
        else  if  y2 < ymin
            then  LeftBottom (xmin,ymin,xmax,ymax,x1,y1,x2,y2,visible)
            else  if  y2 > ymax
                then
                    begin
                        { Use symmetry to reduce to LeftBottom case }
                        y1 := –y1;   y2 := –y2;   { reflect about x-axis }
                        LeftBottom (xmin,–ymax,xmax, –ymin,x1,y1,x2,y2,visible);
                        y1 := –y1;   y2 := –y2;   { reflect back }
                    end
                else
                    begin
                        dx := x2 – x1;   dy := y2 – y1;
                        if  x2 > xmax  then
                            begin
                                y2 := y1 + dy*(xmax – x1)/dx;    x2 := xmax;
                            end;
                        y1 := y1 + dy*(xmin – x1)/dx;    x1 := xmin;
                        visible := true;
                    end
end;
```

Nicholl-Lee-Nicholl

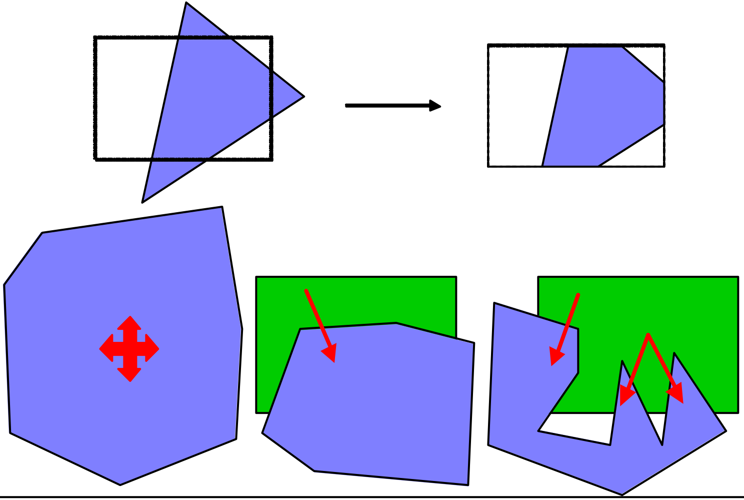
```
procedure LeftBottom (     real  xmin, ymin, xmax, ymax;
                       ref real  x1, y1, x2, y2; ref boolean  visible)
begin
    real dx, dy, a, b, c;

    dx := x2 – x1;            dy := y2 – y1;
    a   := (xmin – x1)*dy;    b   := (ymin – y1)*dx;
    if  b > a
        then  visible := false  { (x2,y2) is below ray from (x1,y1) to bottom left corner }
        else
            begin
                visible := true;
                if  x2 < xmax
                    then
                        begin   x2 := x1 + b/dy;    y2 := ymin;    end
                    else
                        begin
                            c := (xmax – x1)*dy;
                            if  b > c
                                then   { (x2,y2) is between rays from (x1,y1) to
                                         bottom left and right corner }
                                    begin   x2 := x1 + b/dy;    y2 := ymin;    end
                                else
                                    begin   y2 := y1 + c/dx;    x2 := xmax;   end
                        end;
            end;
    y1 := y1 + a/dx;   x1 := xmin;
end;
```

Clipping Algorithms Summary

- Cohen-Sutherland
  - Repeated clipping is expensive
  - Best when trivial accepts/rejects occur often
- Cyrus-Beck
  - Cheap intersection parameter calculation
  - Points are clipped only once at the and
  - Best when most lines have to be clipped
- Liang-Barsky - optimized Cyrus-Beck for window
- Nicholl et. al. - Fastest, not applicable in 3D

2D Polygon Clipping

- **Given an initial polygon, find areas within viewport**
  - this will yield one *or more* polygons



Sutherland-Hodgman Algorithm

- How to clip a polygon against a single plane?

  When the polygon is being clipped by one side of the window, traverse the polygon in a clockwise fashion

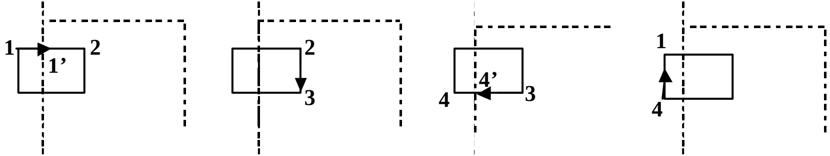
- Since each edge of the polygon is individually compared with the clipping plane, only the relationship between a single edge and a single clipping plane need be considered.

- The order in which the polygon is clipped against the various window boundaries is immaterial.

Sutherland-Hodgman

- While traversing the polygon, there are only four possibilities for each edge, namely:
  - going in of the window
  - two endpoints are inside the window (*i.e.* on visible side of clipping boundary)
  - going out of the window
  - two endpoints are outside the window

– output the intersection point and visible terminating vertex



out to in
Save: 1', 2

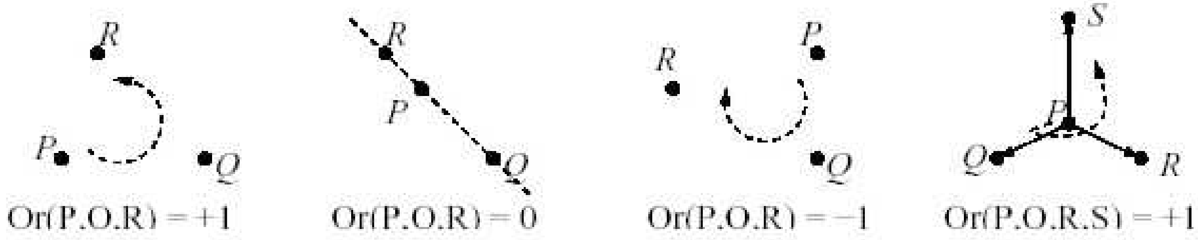
in to in
Save: 3

in to out
Save: 4'

out to out
Save: none

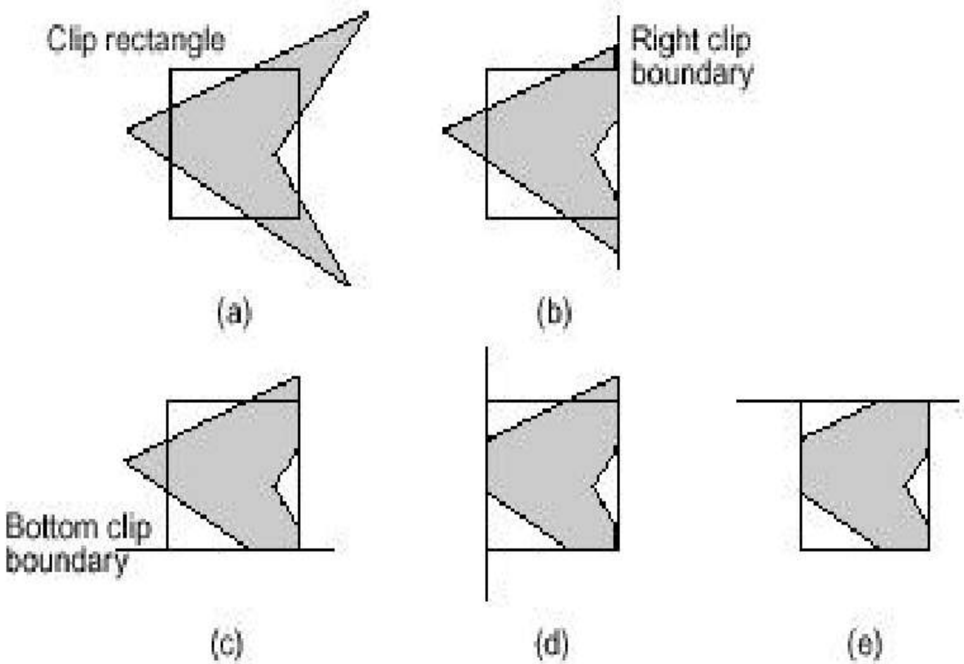
Oriented Function

- $s_i = Or_2(A_i, P, Q)$



$$Or_2(P,Q,R) = p_x(q_y - r_y) + p_y(r_x - q_x) + q_x r_y - q_y r_x$$

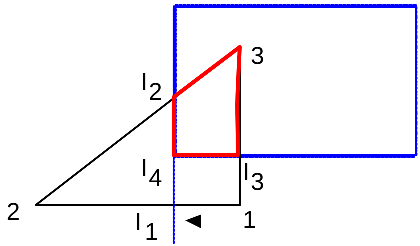
$$Or_2(P,Q,R) = sign\begin{vmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{vmatrix}$$

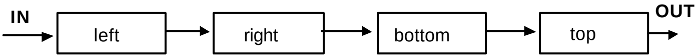


Polygon clipping

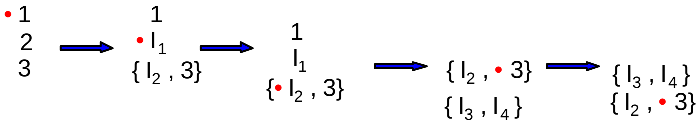
s_i	s_{i+1}	poloha hrany	do zoznamu sa pridáva
$+$	$+$	vnútri	A_{i+1}
$+$	0	vnútri	A_{i+1}
$+$	$-$	vychádza	C
0	$+$	vchádza, A_i na hranici	A_{i+1}
0	0	celá na hranici	A_{i+1} ak $s_{i+2} > 0$, inak $\emptyset$
0	$-$	mimo, A_i na hranici	$\emptyset$
$-$	$+$	vchádza	C, A_{i+1}
$-$	0	A_{i+1} na hranici	A_{i+1} ak $s_{i+2} > 0$, inak $\emptyset$
$-$	$-$	mimo	$\emptyset$

Sutherland-Hodgman





Sucessive processing sequence:




The End