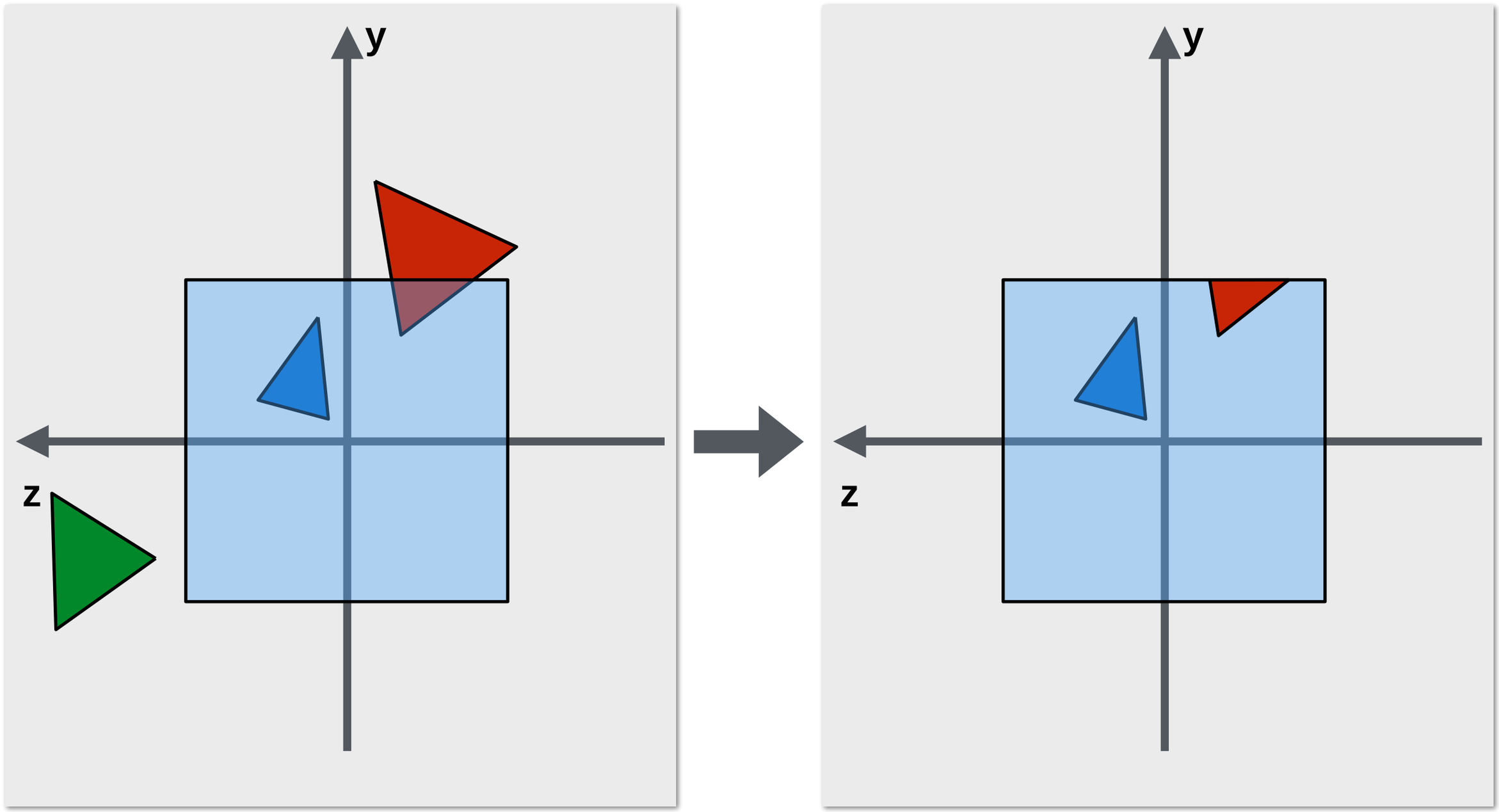
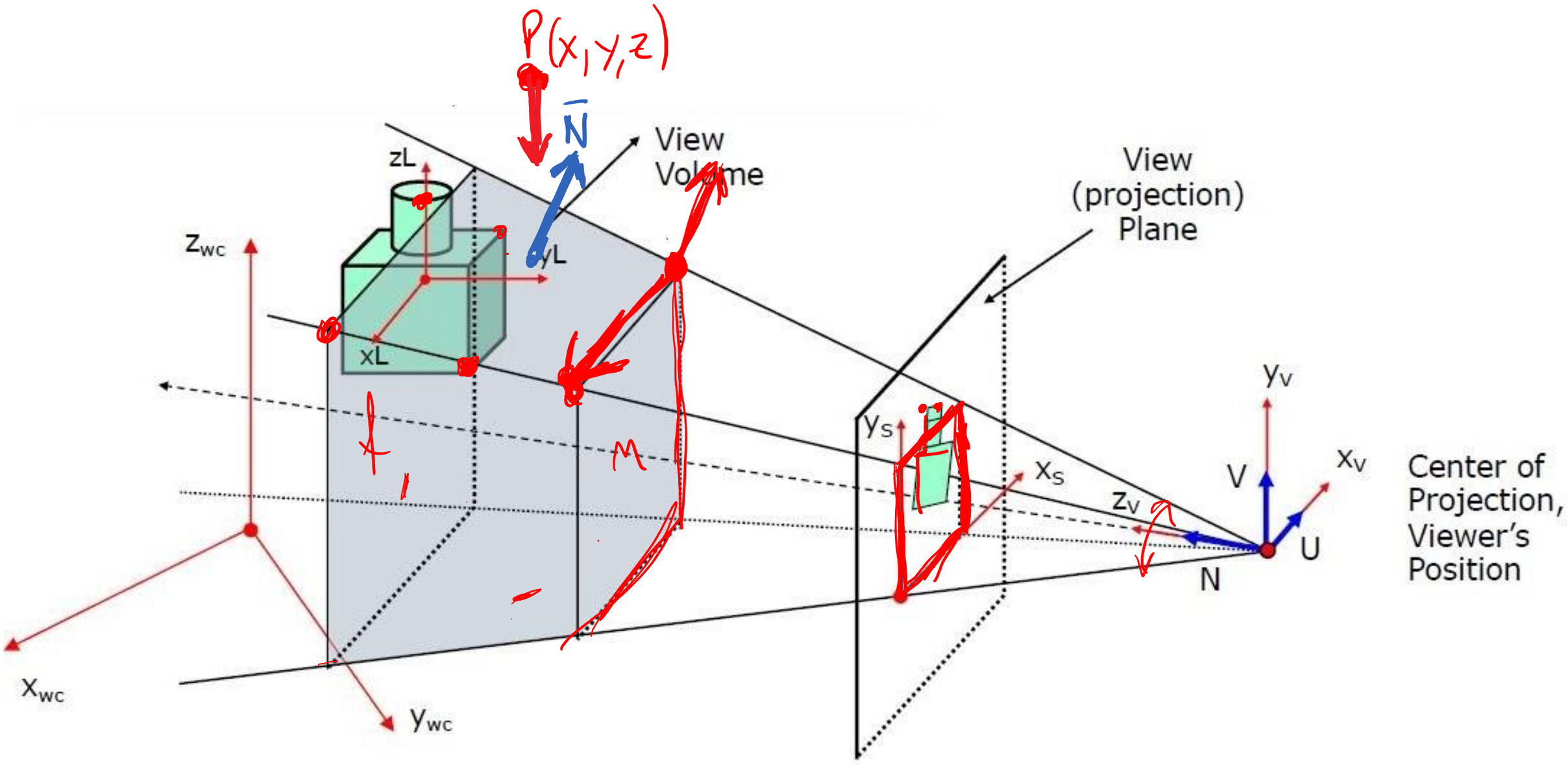
Clipping

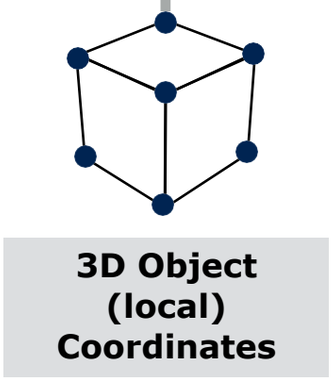
Clipping



Clipping


P(x,y,z)
N
View Volume
View (projection) Plane
zL
yL
xL
Zwc
Xwc
Ywc
ys
xs
zv
yv
xv
V
U
N
Center of Projection, Viewer's Position

Clipping



3D Object (local) Coordinates

Modeling Tranformations

3D World Coordinates

Transform to viewing coordinates

3D Viewing (Eye) Coordinates

Per-vertex shading

Projection transformation

3D Clip Coordinates

Primitive setup

Clipping

Perspective divide

Normalized device coordinates

Viewport transformation

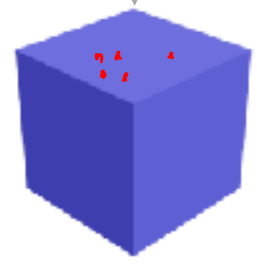
Window coordinates (plus z-value)

Rasterization

Screen coordinates

Fragment processing

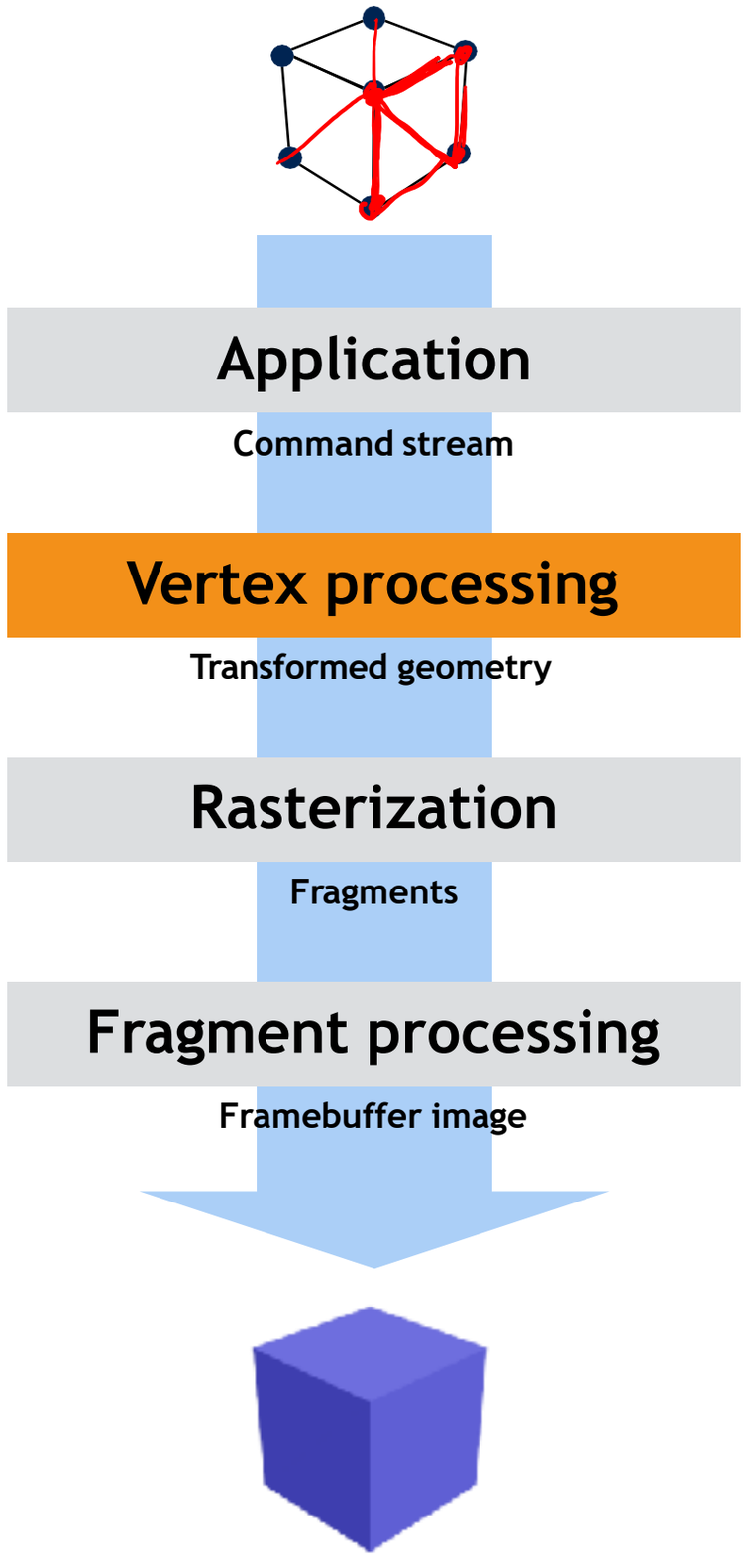
Frame buffer



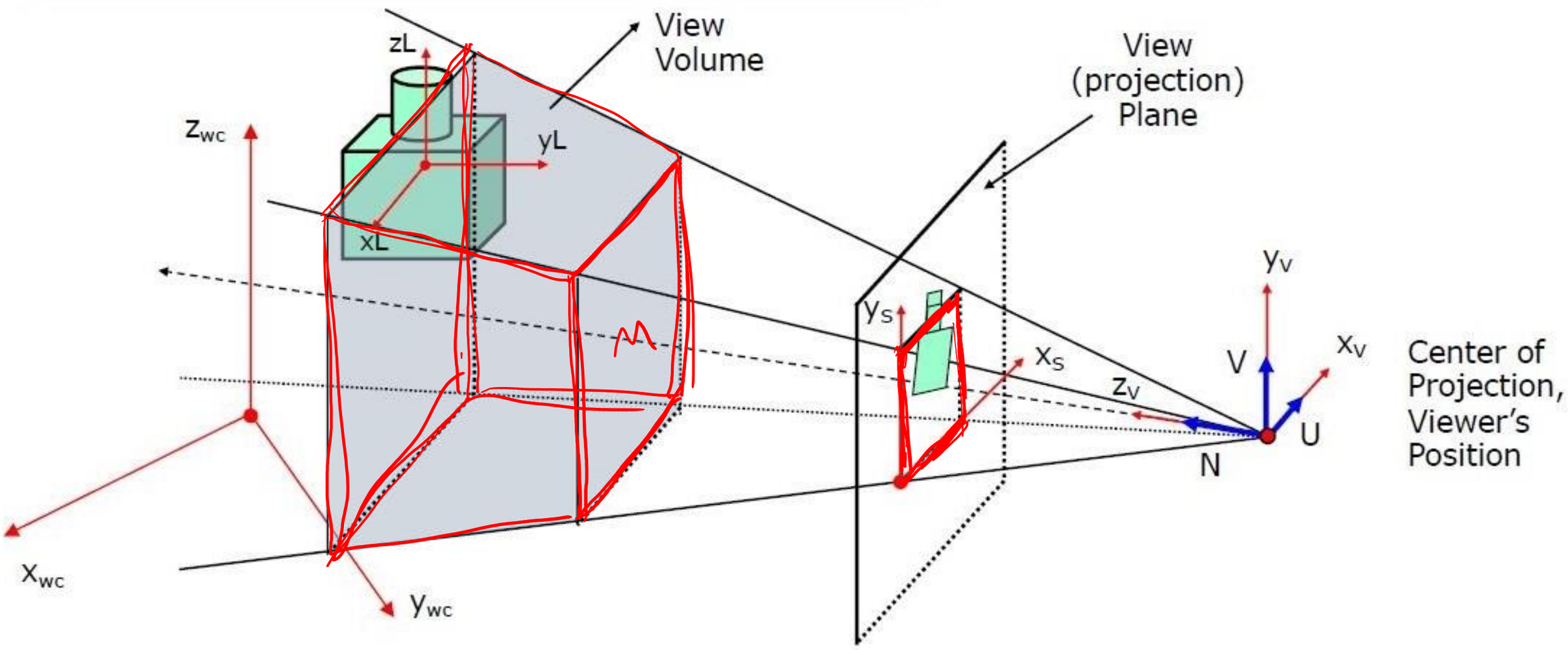
Clipping

- Avoid drawing parts of primitives that are outside of the window
- Common approaches for implementing clipping:
  - in world coordinates using the six planes of the frustum
  - in the 4D transformed space before the homogeneous (perspective) divide

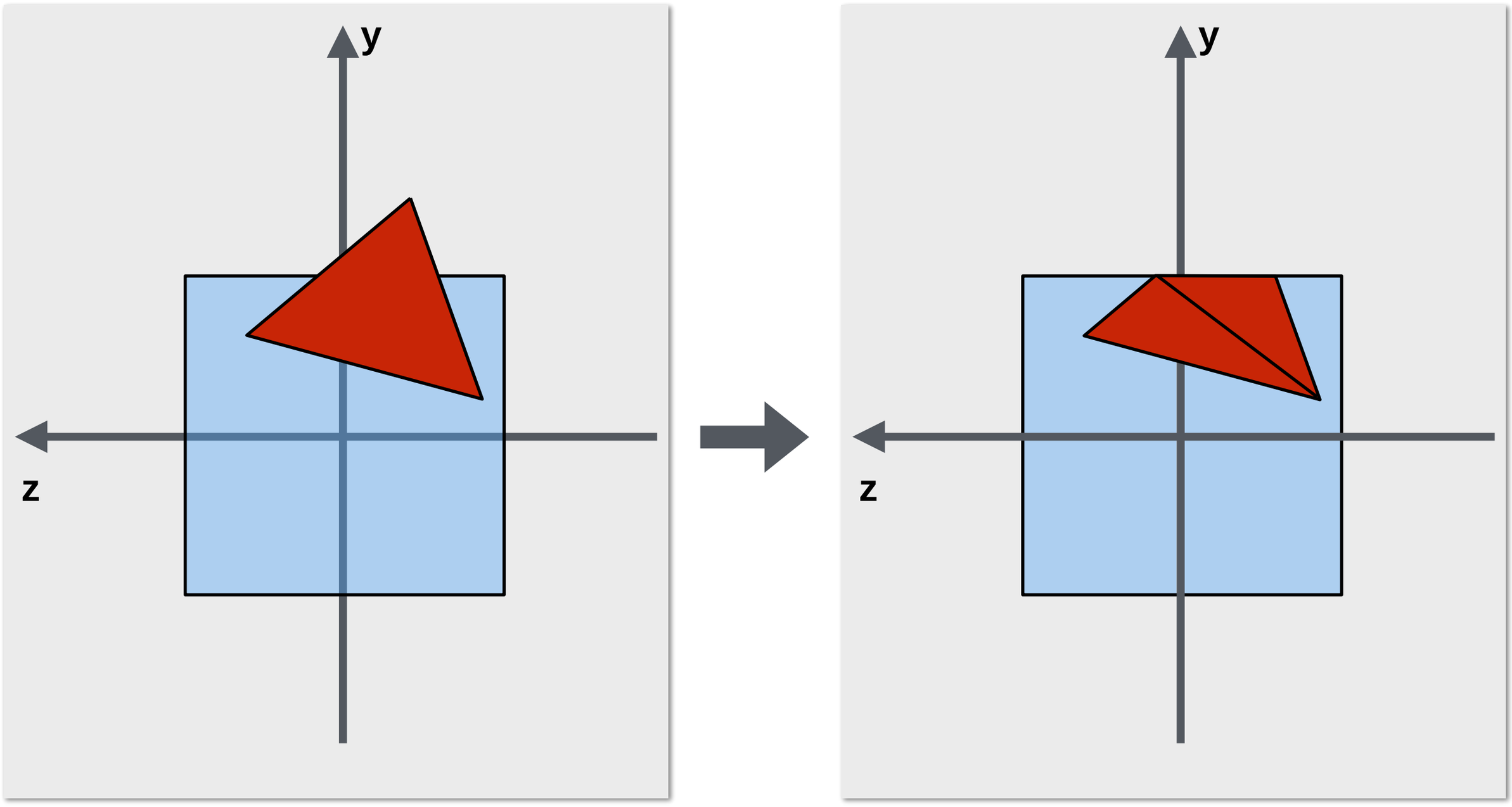
```
for each of six planes do
    if (triangle entirely outside of plane) then
        break (triangle is not visible)
    else if triangle spans plane then
    clip triangle
    if (quadrilateral is left) then
        break into two triangles
```



Clipping


View
Volume
View
(projection)
Plane
zL
yL
xL
z_{wc}
x_{wc}
y_{wc}
y_s
x_s
z_v
y_v
x_v
V
U
N
Center of
Projection,
Viewer's
Position

Clipping

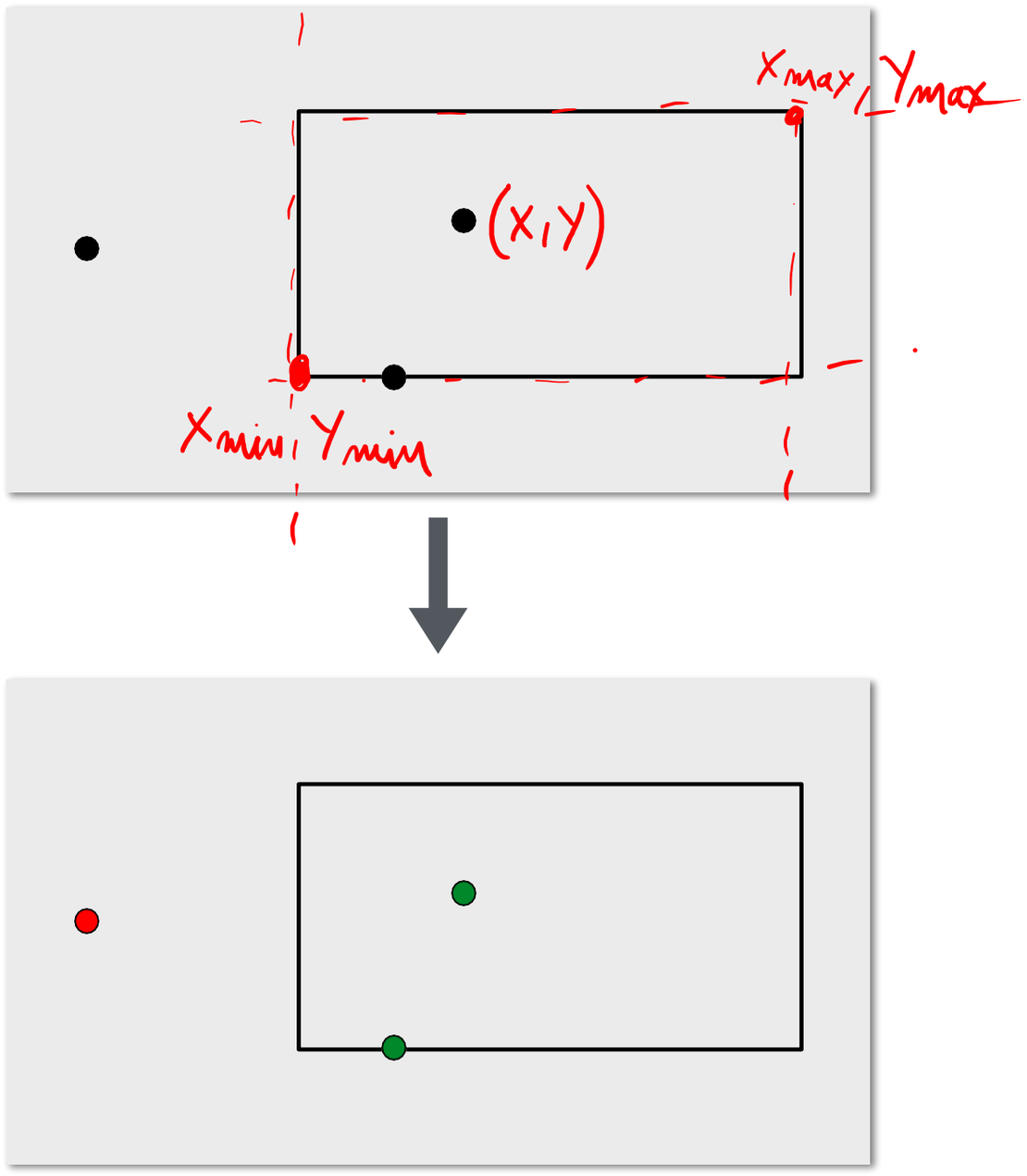


Clipping

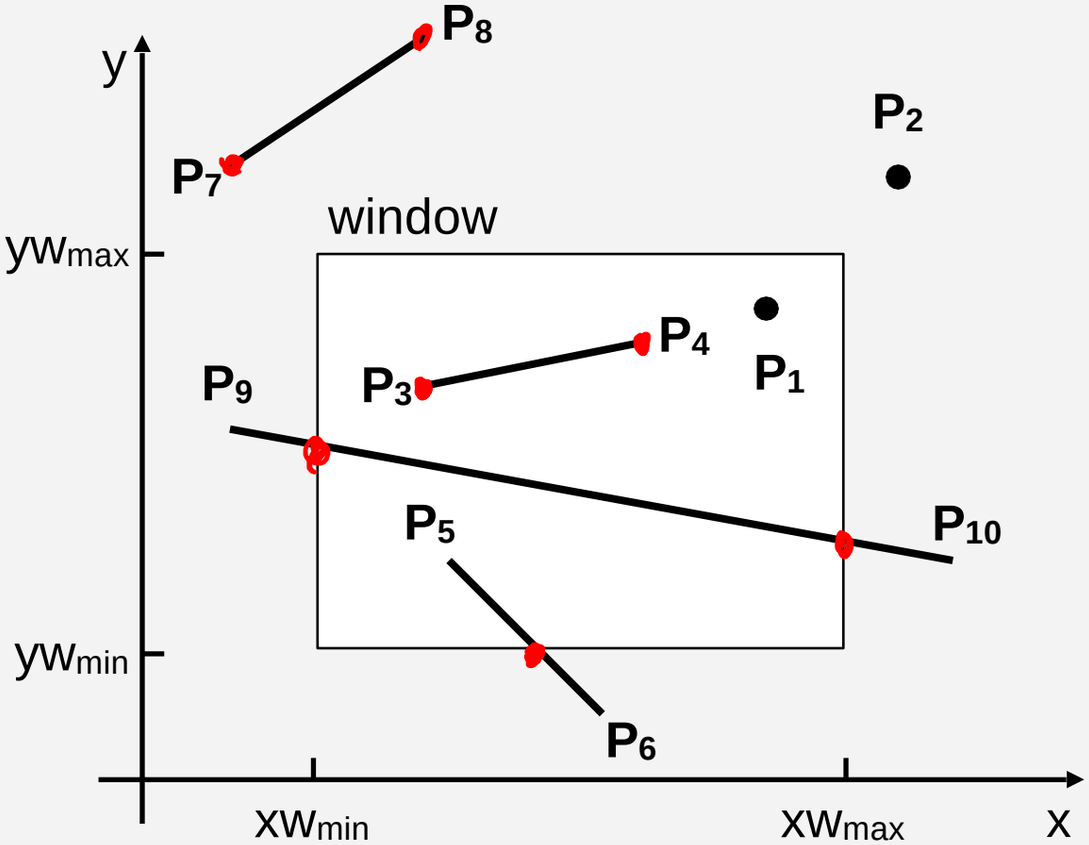
- **Frustum culling**
  - Eliminate polygons that are entirely outside of the view frustum
- **Near-plane clipping**
  - Clip polygons against the near plane
- **Whole-frustum clipping**
  - Clip polygons to the side and far planes

Point clipping

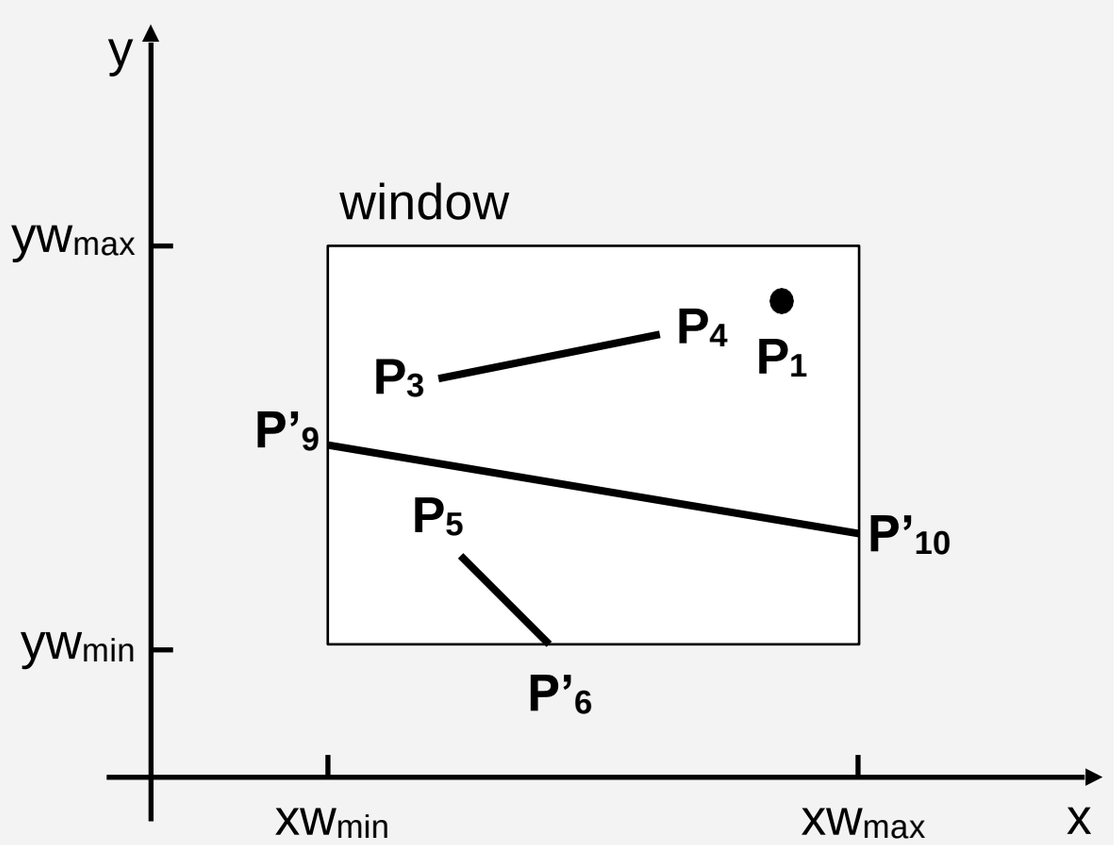
- Inside outside test:
  - $x_{min} \leq x \leq x_{max}$
  - $y_{min} \leq y \leq y_{max}$



Line clipping



Before clipping



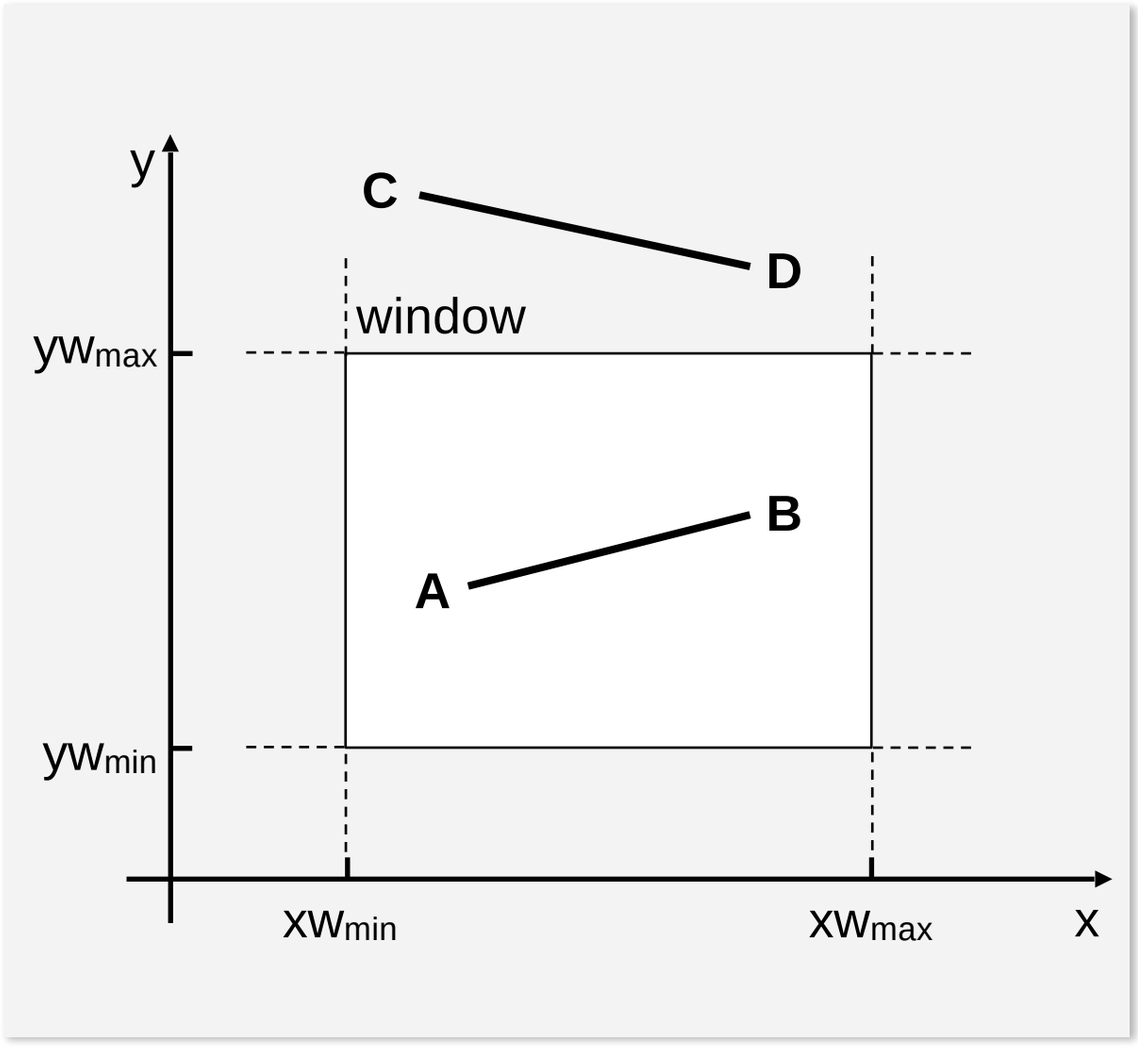
After clipping

Line clipping

- Line definition:
  - **set of points** → point clipping
  - **vector** →
    - endpoints clipping
    - binary clipping
    - parametric line clipping
    - Cohen-Sutherland algorithm

Endpoints clipping

- Could be two cases:
  - trivially accepted (AB)
  - trivially rejected (CD)

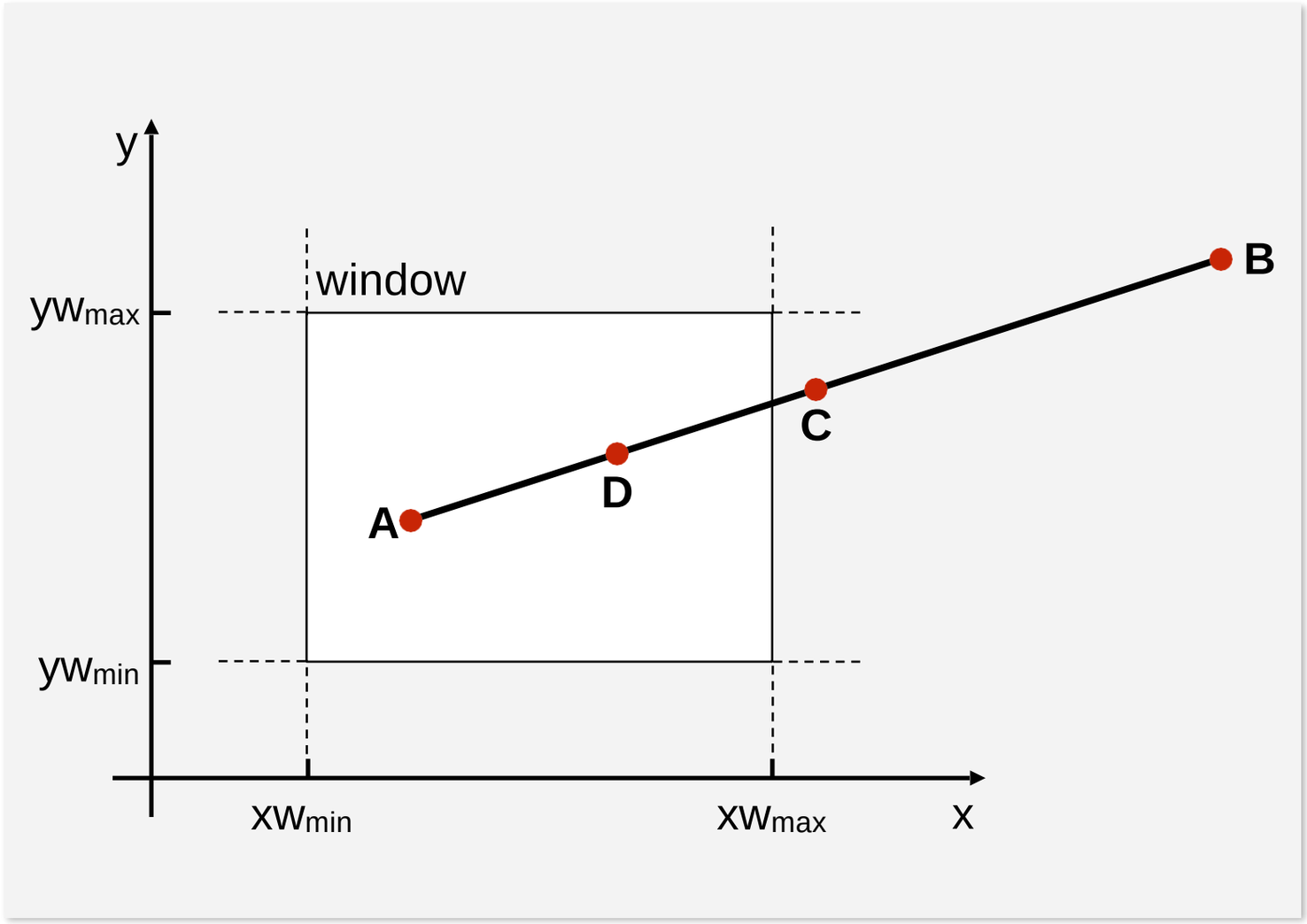


Binary clipping

1. step: AB
2. step: AC, CB (tr. rejected)

3.3 step: AD (tr. acc), DC

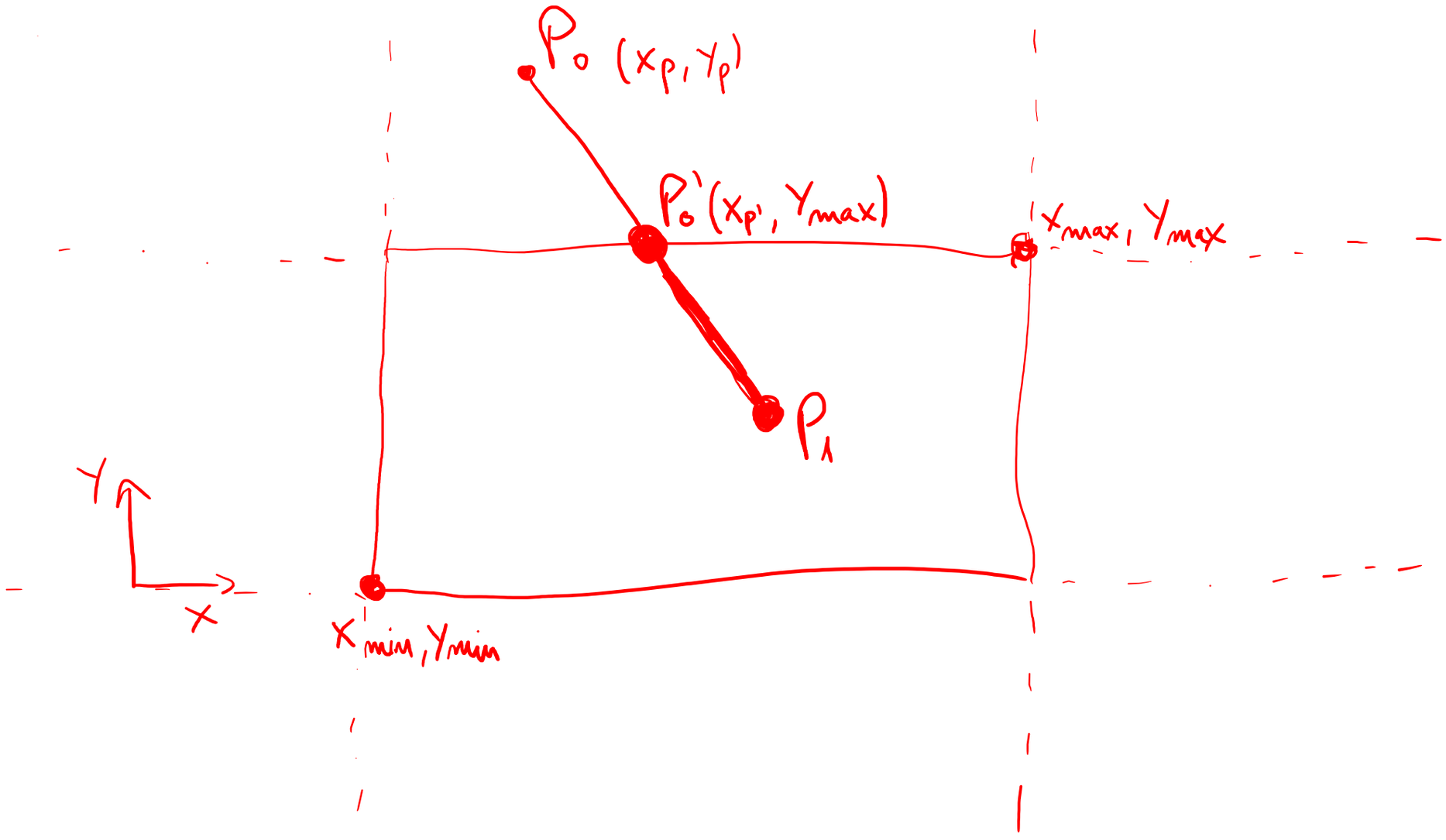
4. . . .



Cohen-Sutherland clipping algorithm

- Basic Idea
  - Encode the line endpoints
  - Successively divide the line segments so that they are completely contained in the window or completely lies outside window
  - Division occurs at the boundary of window

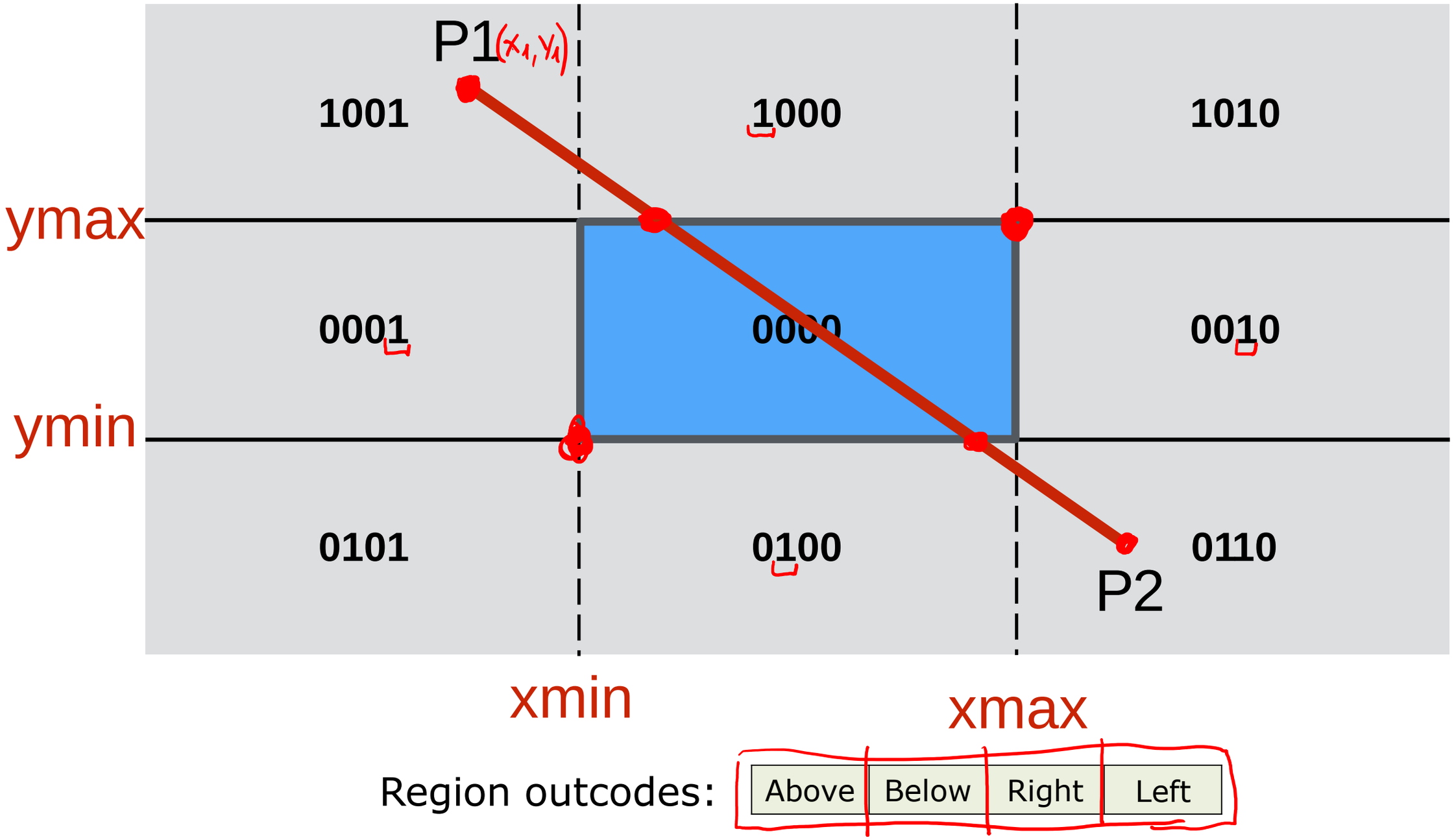
Cohen-Sutherland clipping algorithm



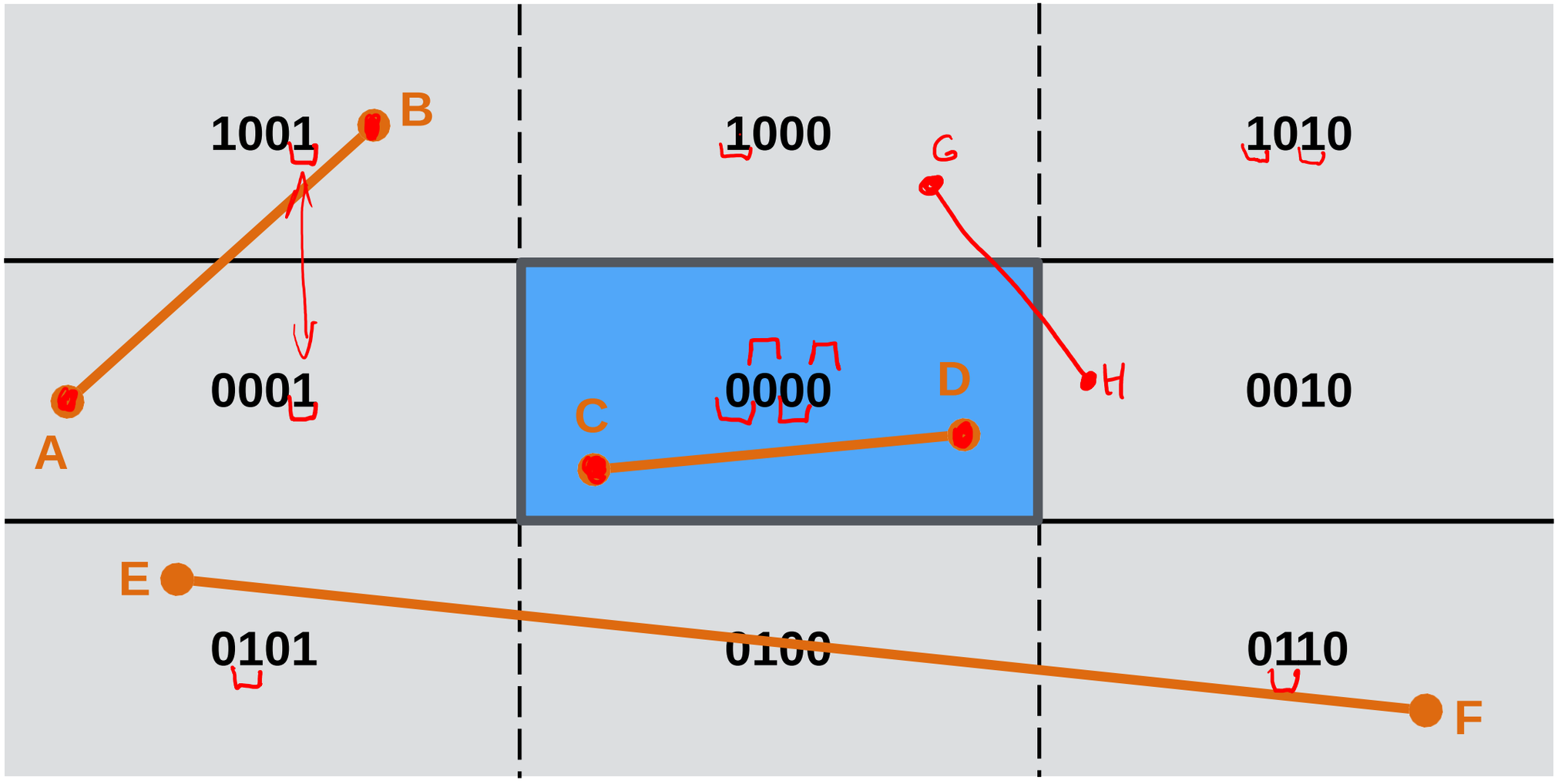
Clipping: $P_0 \Rightarrow P_0'(x_{p'}, Y_{max})$

$x_{p'} = ?$

Cohen-Sutherland clipping algorithm



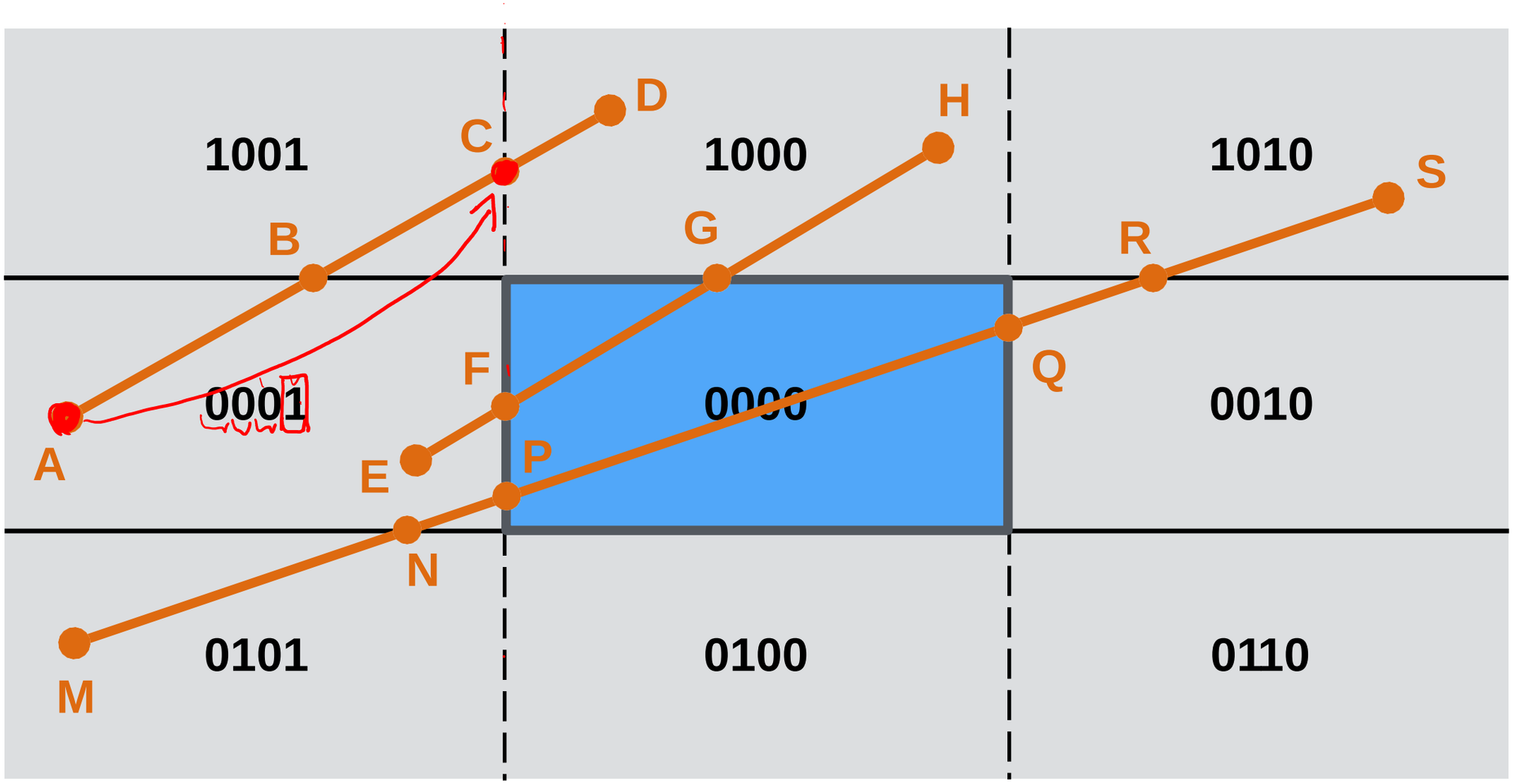
Cohen-Sutherland clipping algorithm



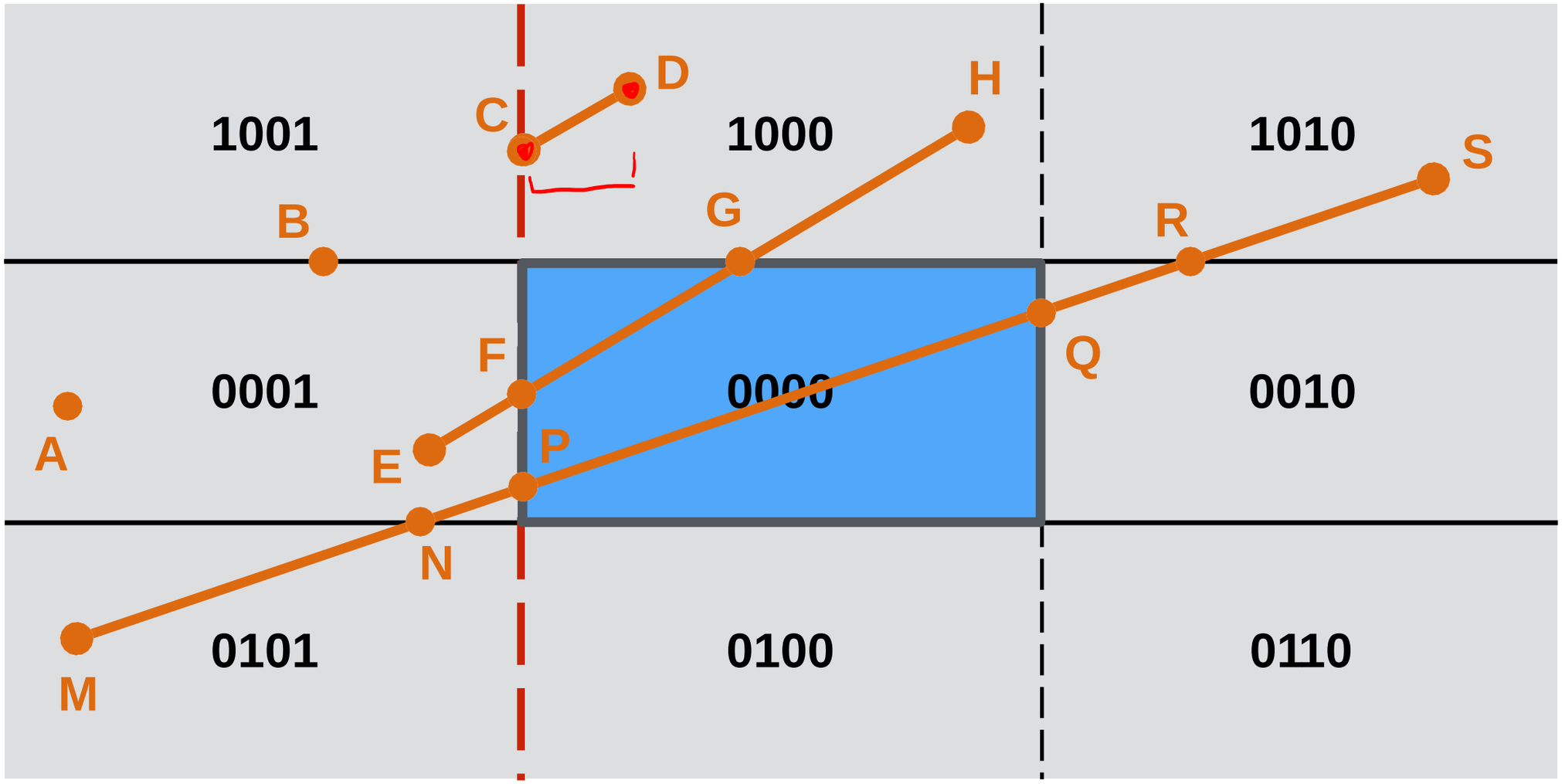
trivially accepted: **CD**

trivially rejected: **AB**, **EF**

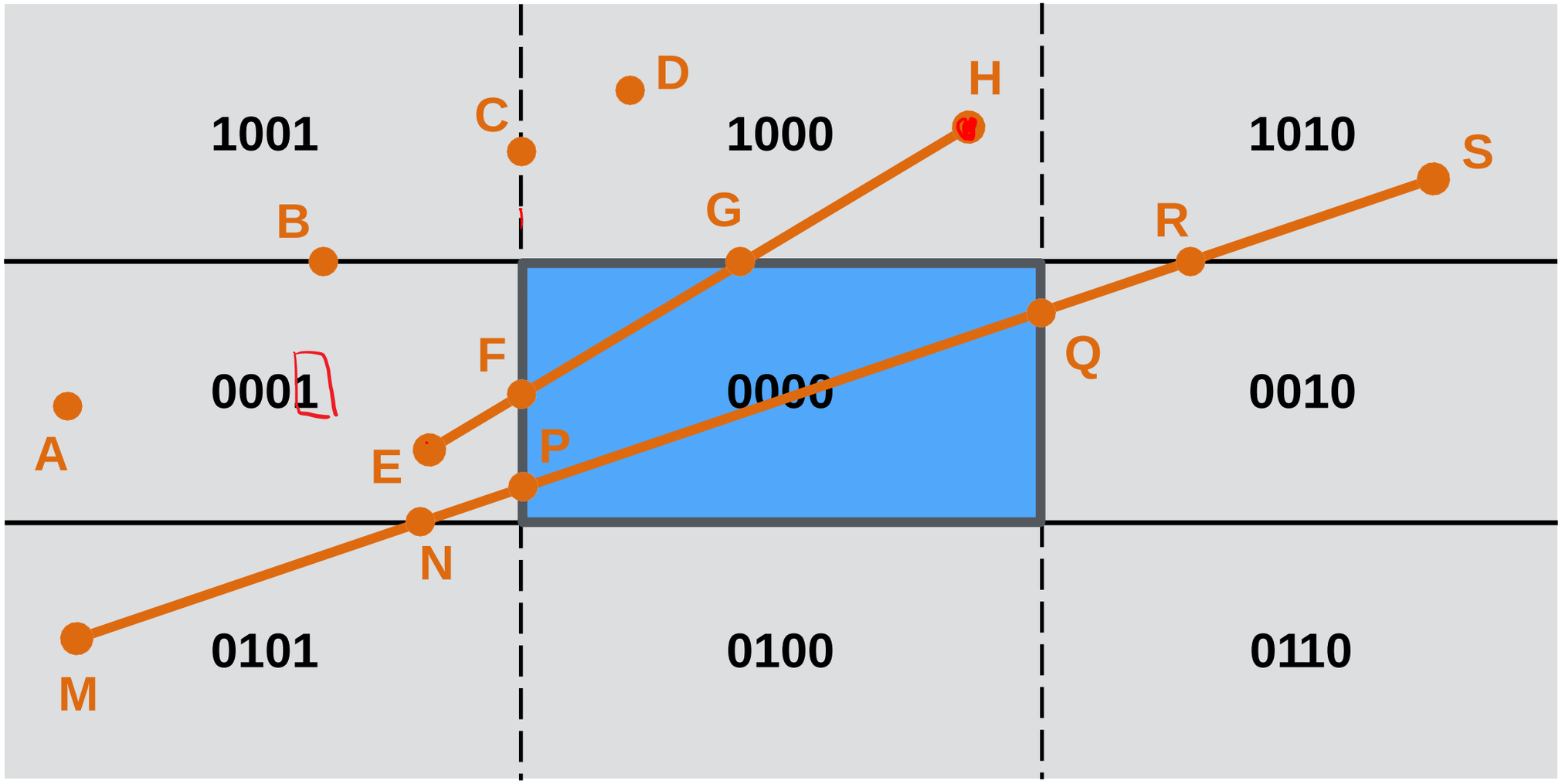
Cohen-Sutherland clipping algorithm



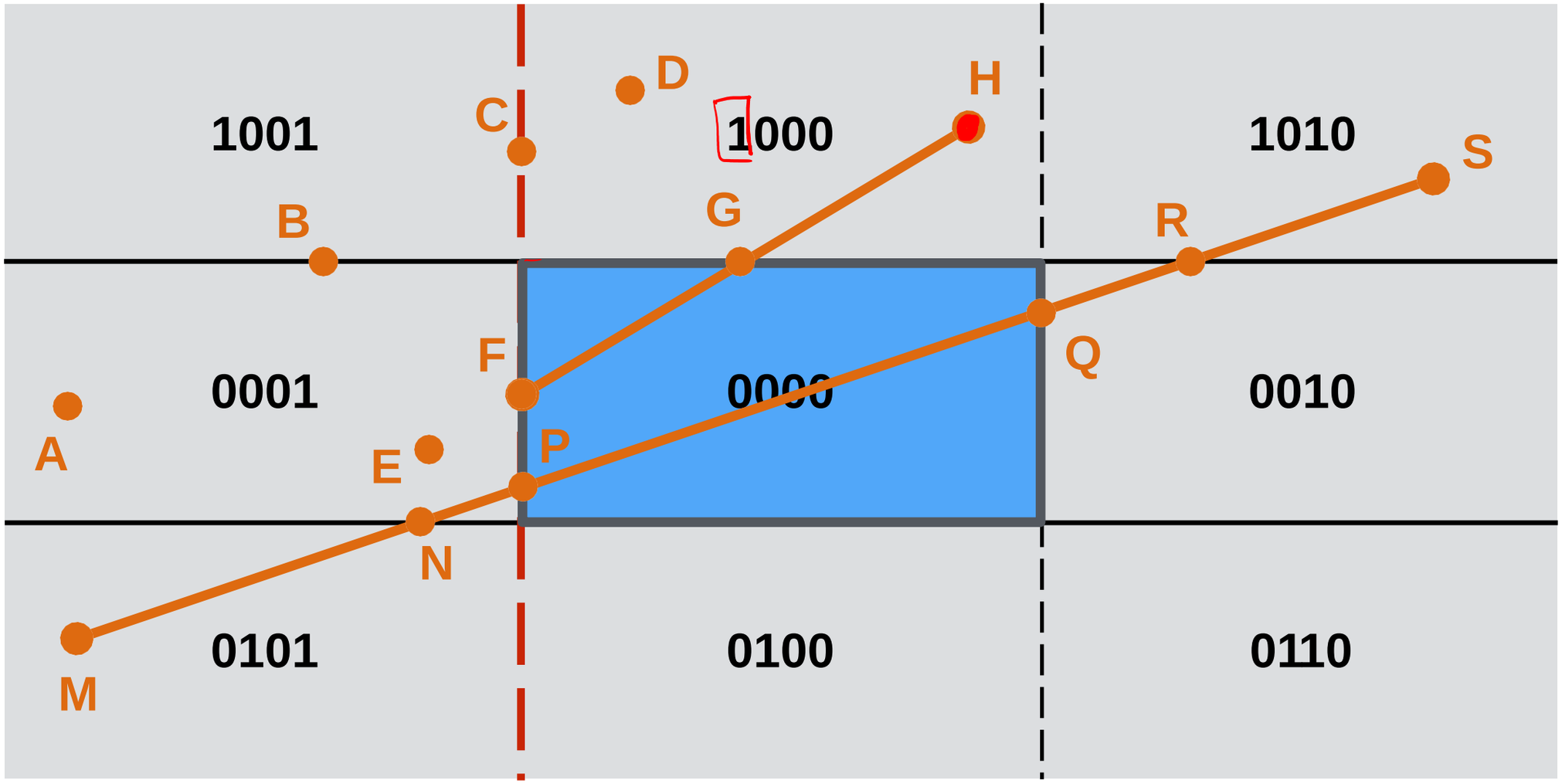
Cohen-Sutherland clipping algorithm



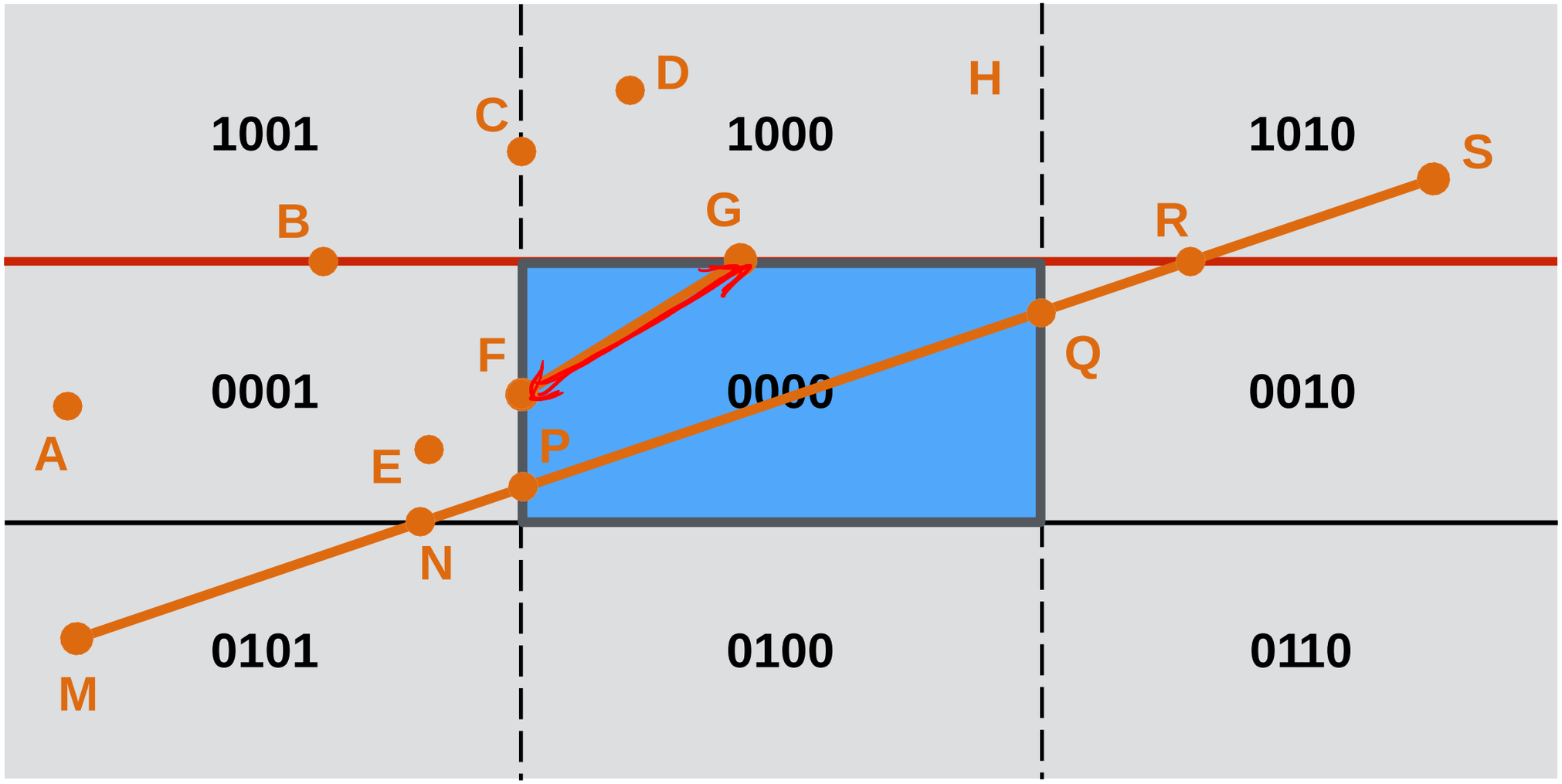
Cohen-Sutherland clipping algorithm



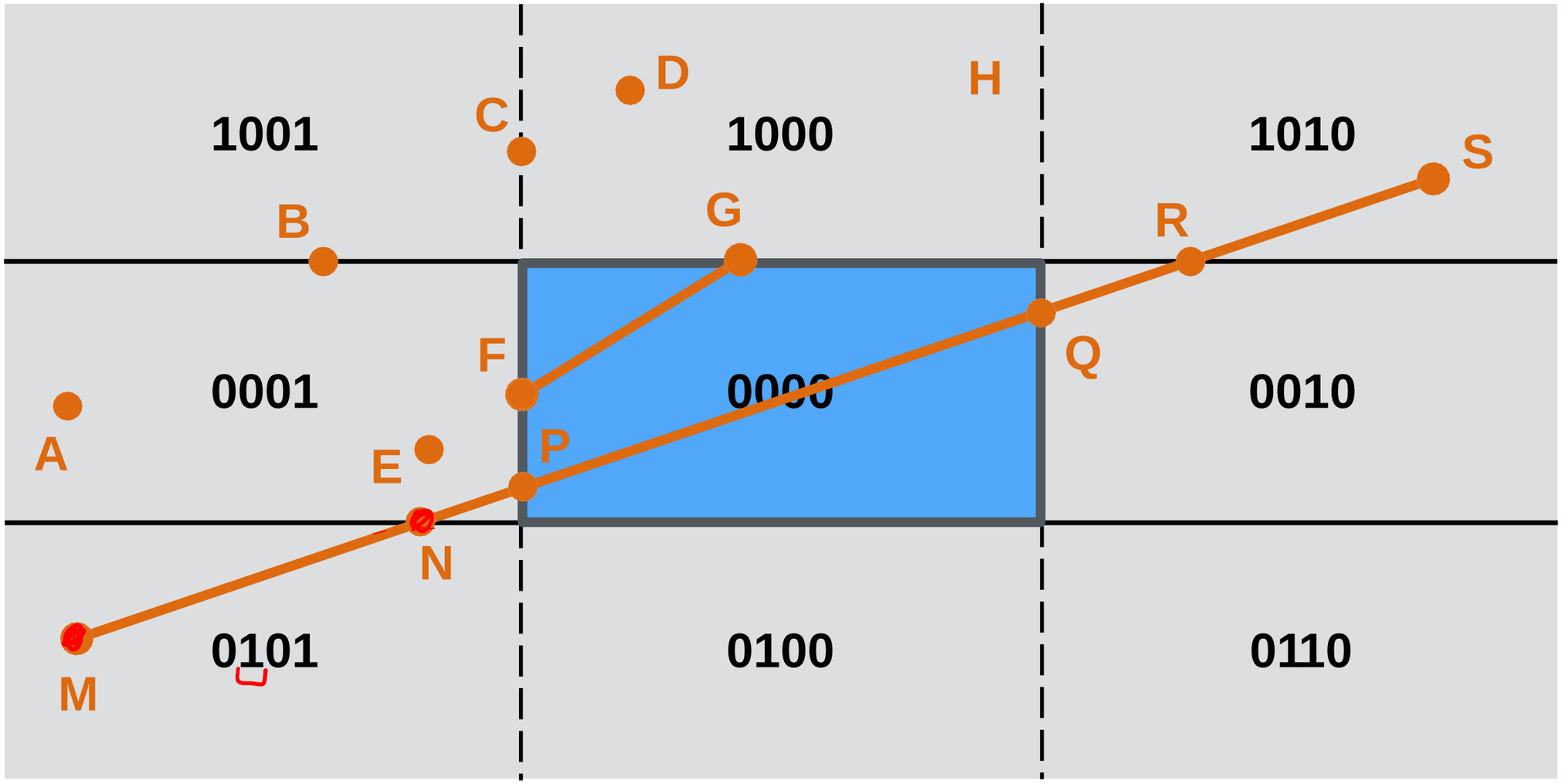
Cohen-Sutherland clipping algorithm



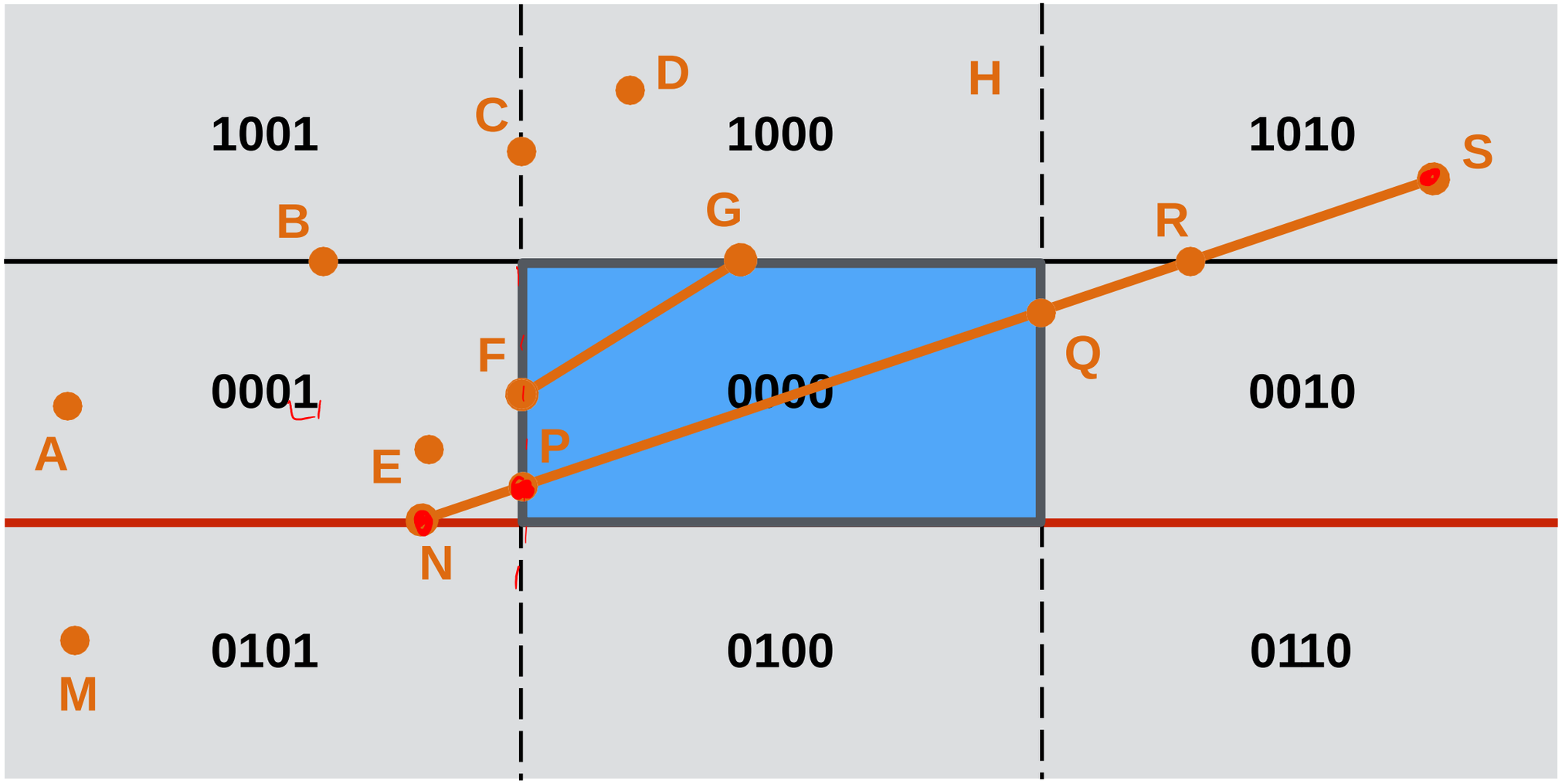
Cohen-Sutherland clipping algorithm



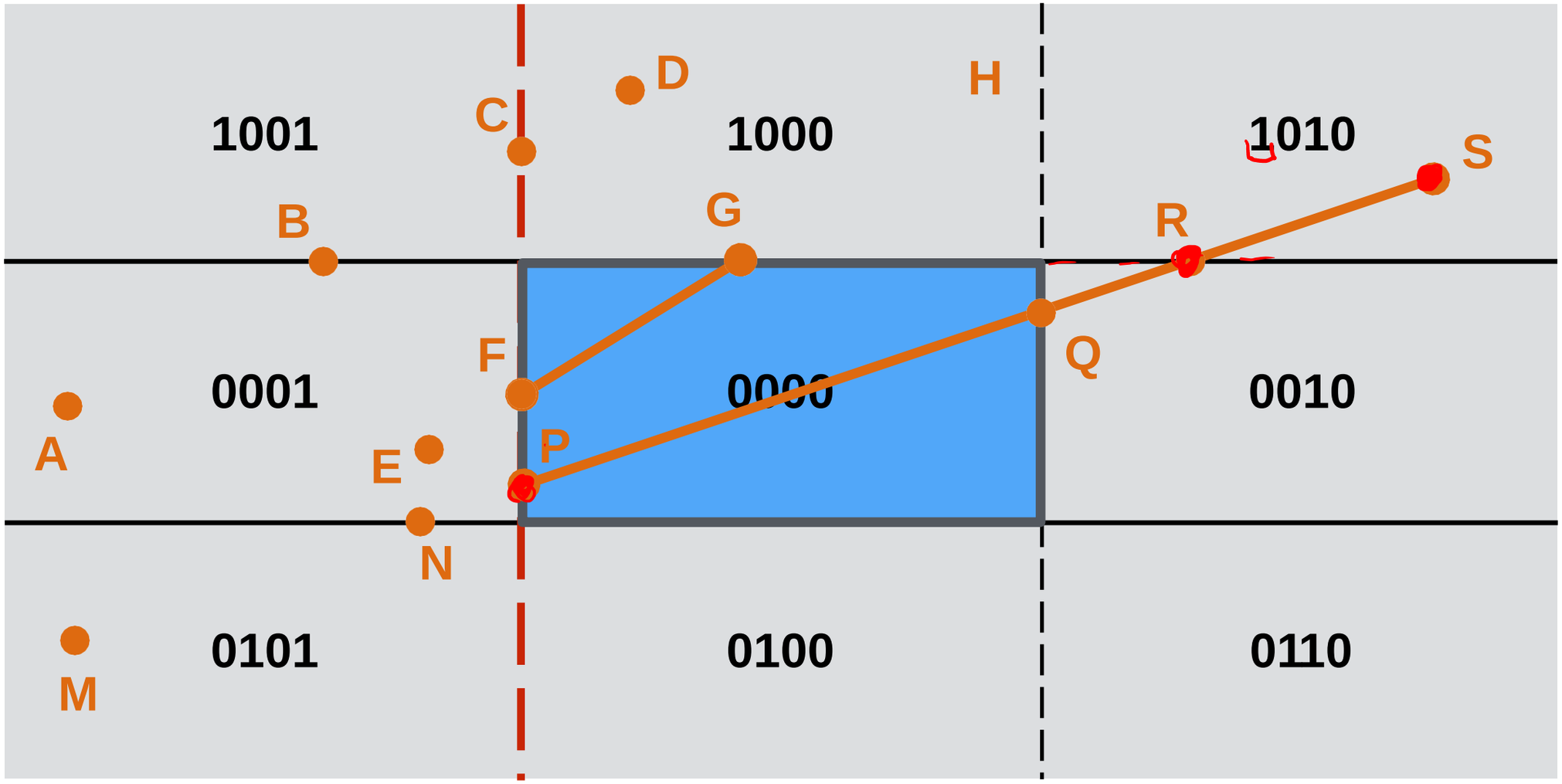
Cohen-Sutherland clipping algorithm



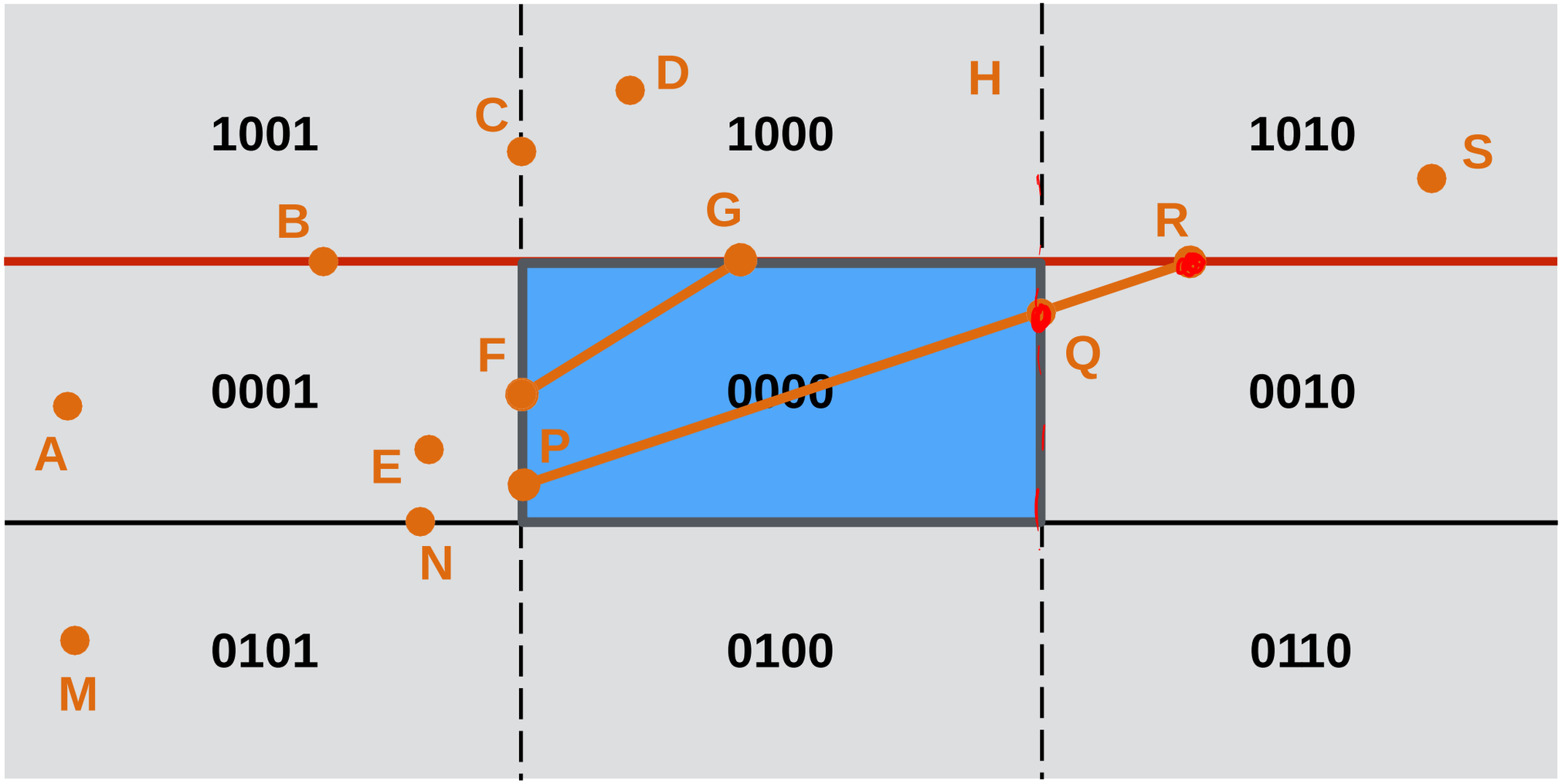
Cohen-Sutherland clipping algorithm



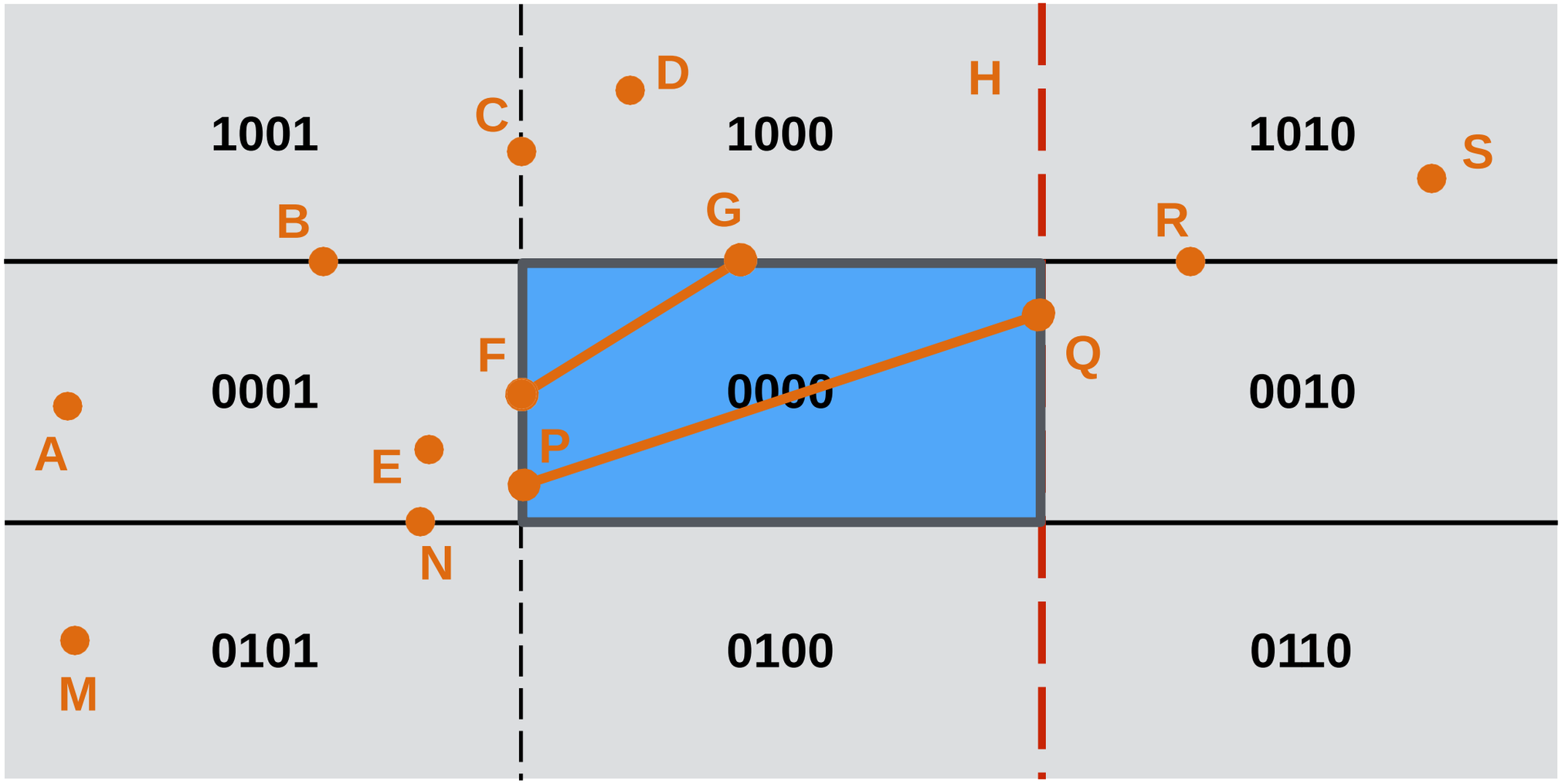
Cohen-Sutherland clipping algorithm



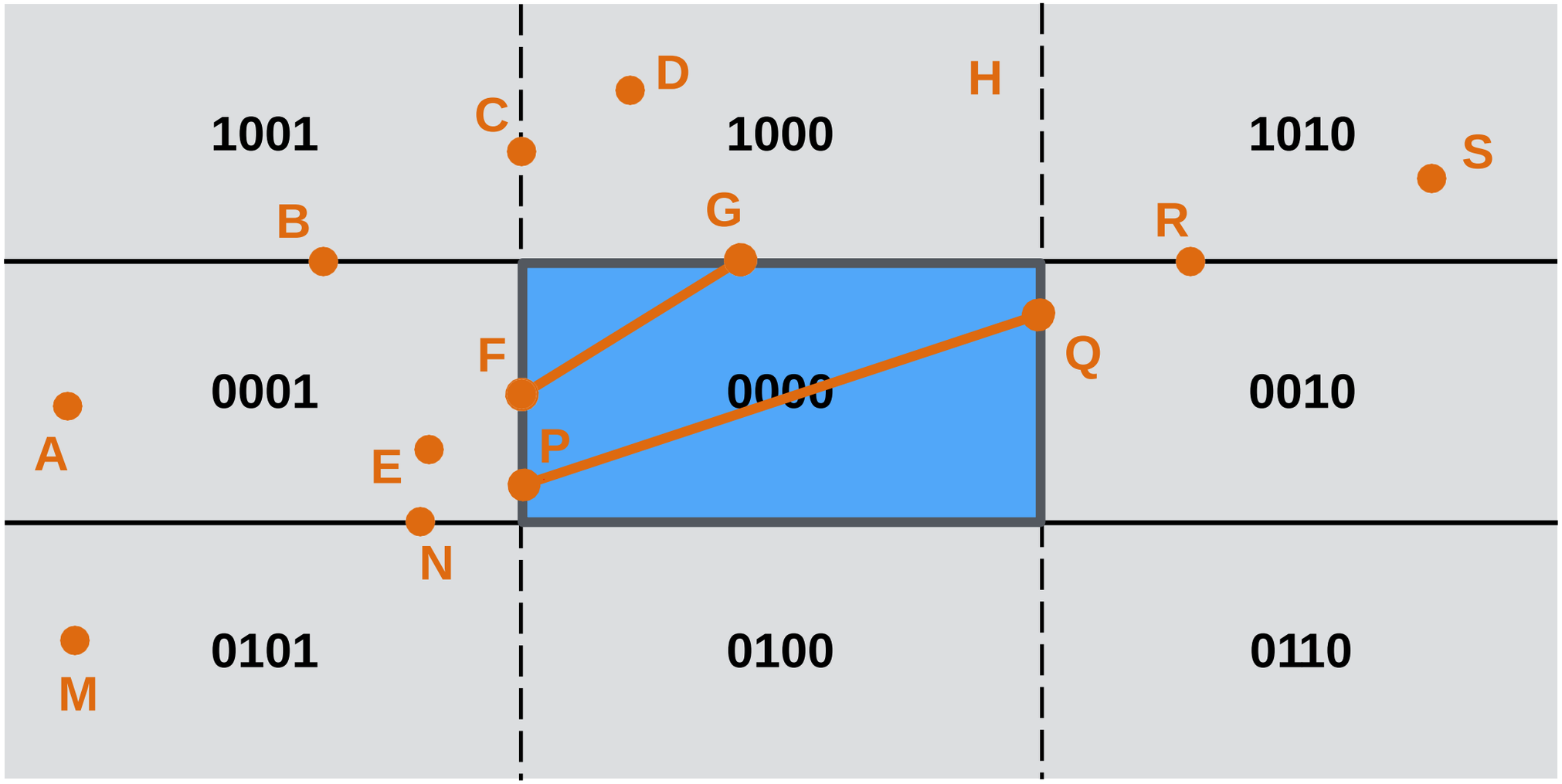
Cohen-Sutherland clipping algorithm



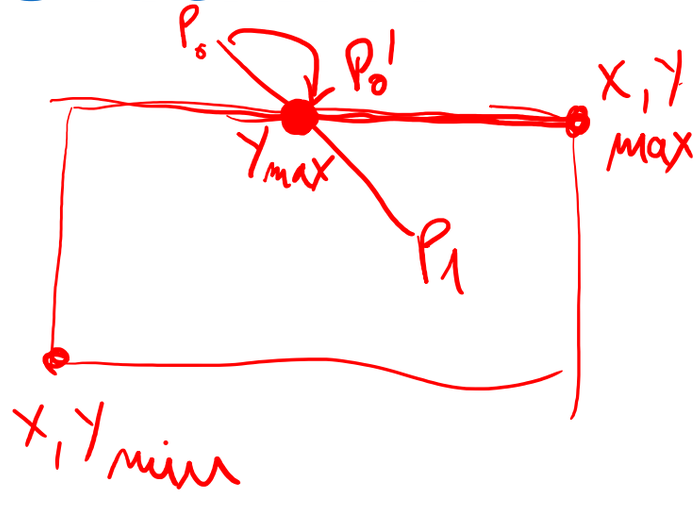
Cohen-Sutherland clipping algorithm



Cohen-Sutherland clipping algorithm



Cohen-Sutherland Algorithm




```
repeat{
    ComputeOutCode(x0, y0, outcode0);
    ComputeOutCode(x1, y1, outcode1);

    check for trivial reject or trivial accept;

    if (x0, y0 is the inside point){
            invert(x0, y0, x1, y1)
            invert(outcode0, outcode1)
    }

    if(outcode & TOP){
        x0 = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);    // divide line at top of clip rectangle
        y0 = ymax;
    }else if(outcode & BOTTOM){
        x0 = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);    // divide line at bottom of clip rectangle
        y0 = ymin;
    }else if(outcode & RIGHT){
        y0 = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);    // divide line at right of clip rectangle
        x0 = xmax;
    }else if(outcode & LEFT){
        y0 = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);    // divide line at left of clip rectangle
        x0 = xmin;
    }

}until done
```

1. Calc. coduri
2. S.A. ⊻ S.R. ?
3. if P0 inside ⇒ switch P0 & P1
4. Calc. inters.

up
down
right
left

Cohen-Sutherland Algorithm

$$P(t) = (1-t)\cdot P_0 + t\cdot P_1$$

$t=0 \Rightarrow P(t)=P_0$

$t=1 \Rightarrow P(t)=P_1$

repeat{

ComputeOutCode(x0, y0, outcode0);
ComputeOutCode(x1, y1, outcode1);

check for trivial reject or trivial accept;

if (x0, y0 is the inside point){
invert(x0, y0, x1, y1)
invert(outcode0, outcode1)
}

if(outcode & TOP){
x0 = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);
y0 = ymax;
}**else if**(outcode & BOTTOM){
x0 = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);
y0 = ymin;
}**else if**(outcode & RIGHT){
y0 = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);
x0 = xmax;
}**else if**(outcode & LEFT){
y0 = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);
x0 = xmin;
}

}**until** done

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

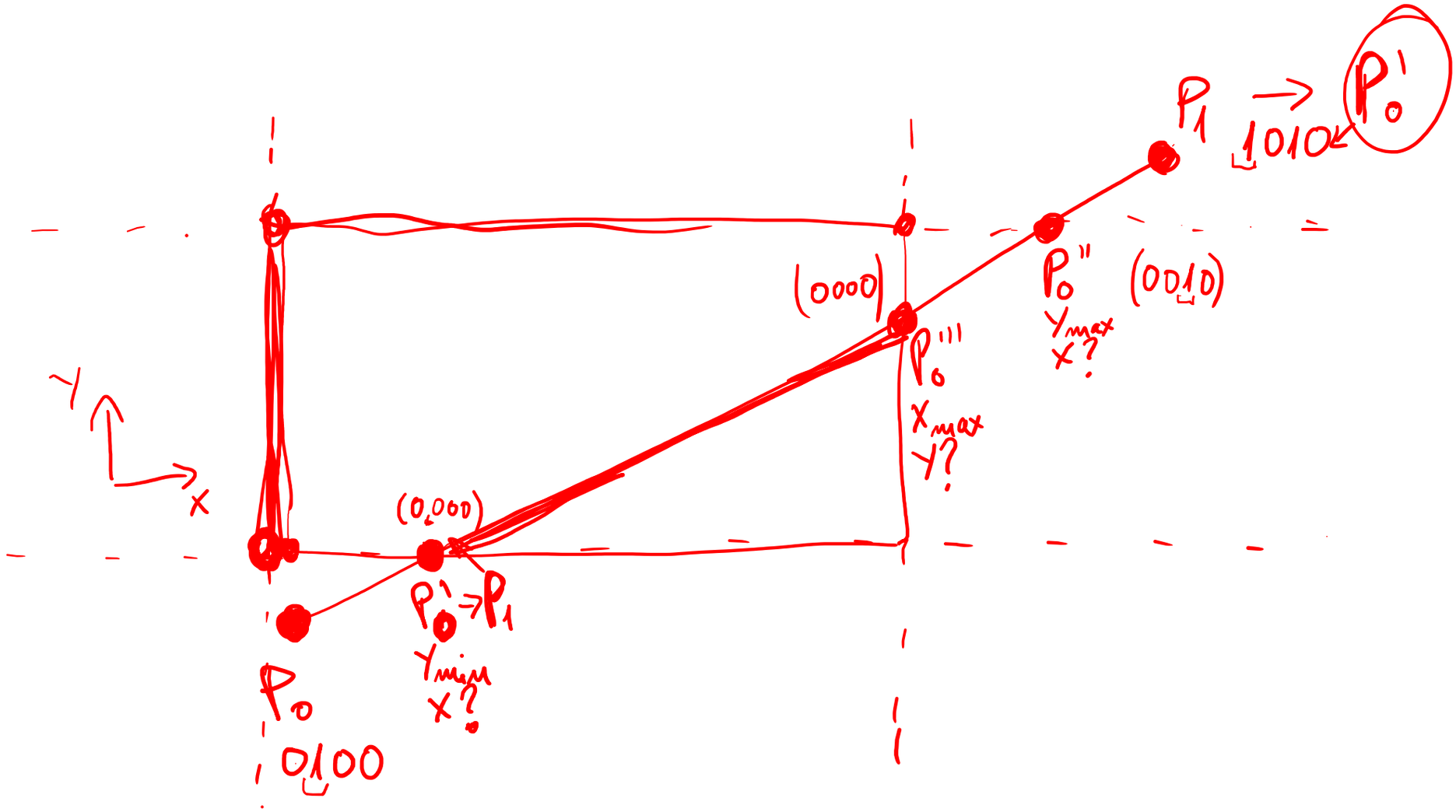
$$x' = x_0 + (x_1 - x_0)t$$

$$y' = y_0 + (y_1 - y_0)t = y_{max}$$

$$t = \frac{y_{max} - y_0}{y_1 - y_0}$$

$$x' = x_0 + (x_1 - x_0)\frac{y_{max} - y_0}{y_1 - y_0}$$

Cohen-Sutherland clipping algorithm



Ecuația parametrică a unei drepte

$$\mathbf{P(t)} = (1 - t)\mathbf{P_0} + t\mathbf{P_1}$$

(bazată pe combinația afină a doua puncte)

$t \in [0,1]$

$t = 0.3 \Rightarrow P(t) = ?$

Se aplică pe coordonate

$X_{(t=0.3)} = (1-t) \cdot X_{P_1} + t \cdot X_{P_2} =$

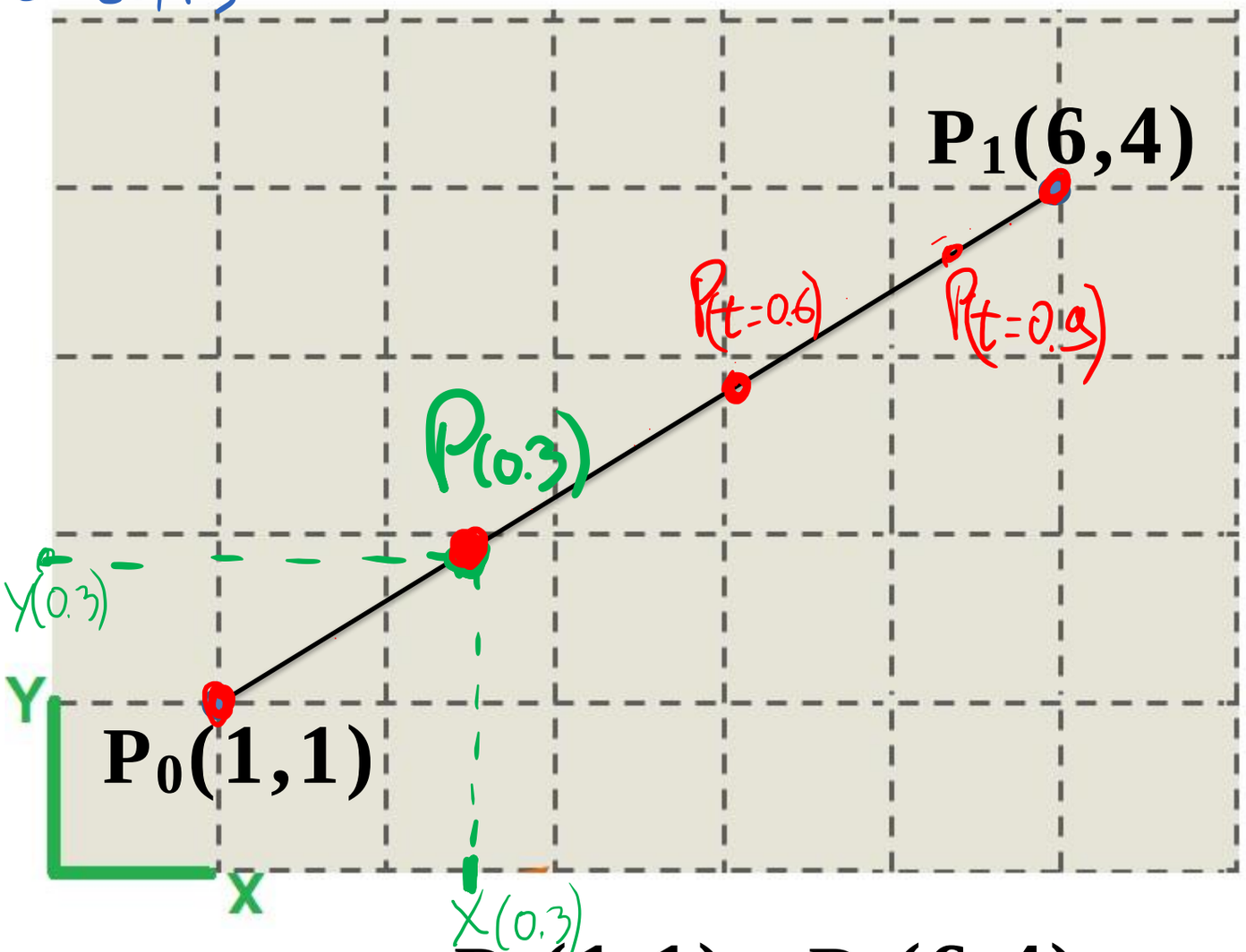
$= 0.7 \cdot 1 + 0.3 \cdot 6 =$

$= 0.7 + 1.8 = 2.5$

$Y_{(t=0.3)} = (1-t) \cdot Y_{P_1} + t \cdot Y_{P_2} =$

$= 0.7 \cdot 1 + 0.3 \cdot 4 =$

$= 0.7 + 1.2 = 1.9$



$\Rightarrow P(0.3) = (2.5, 1.9)$

Parametric line clipping

Basic approach:

1. Oriented line segment and window edge,

   e.g. line from (x_0, y_0) to (x_1, y_1):

   $\mathbf{P} = \mathbf{P_0} + t_{line}(\mathbf{P_1} - \mathbf{P_0})$, where $t_{line} \in [0, 1]$

2. Solve the system of two equations:

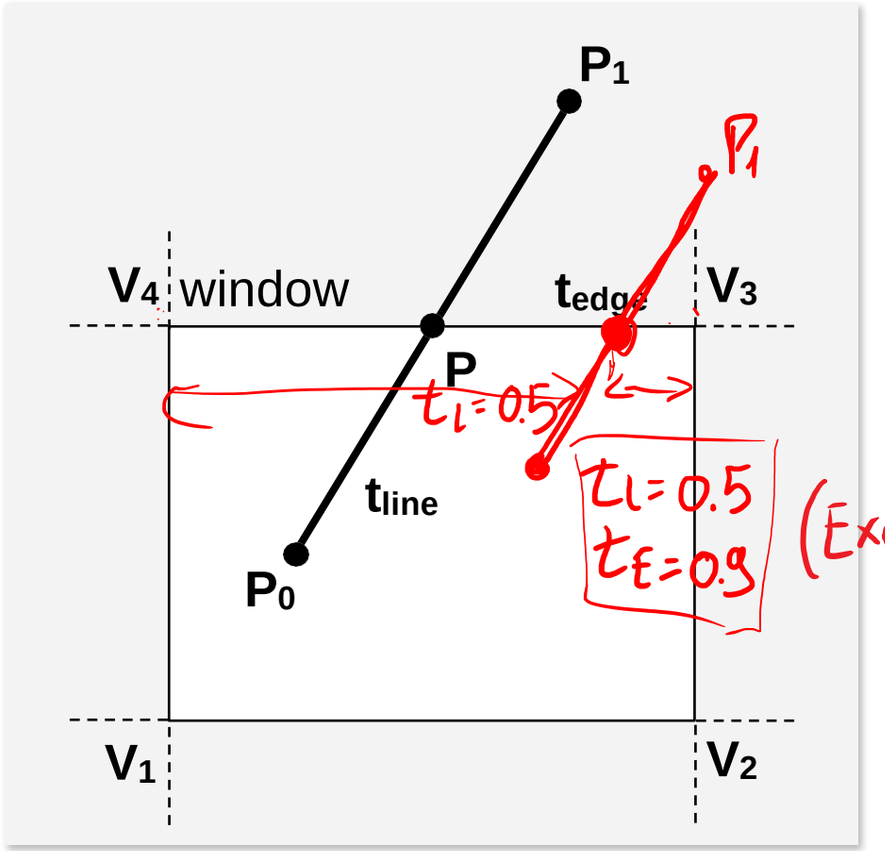
   $\mathbf{P} = \mathbf{P_0} + t_{line}(\mathbf{P_1} - \mathbf{P_0})$

   $\mathbf{P} = \mathbf{V_i} + t_{edge}(\mathbf{V_{i+1}} - \mathbf{V_i})$

3. The line and edge intersects each other

   if both t_{edge} and $t_{line} \in [0, 1]$

4. Solution: the clipped line is $\mathbf{P_0P(t)}$, $t_{line} \in [0, 1]$



Other parametric line clipping algorithms:

- 1978 Cyrus-Beck algorithm
- 1984 Liang-Barsky algorithm (improved form of the Cyrus-Beck algorithm)

Cyrus-Beck algorithm

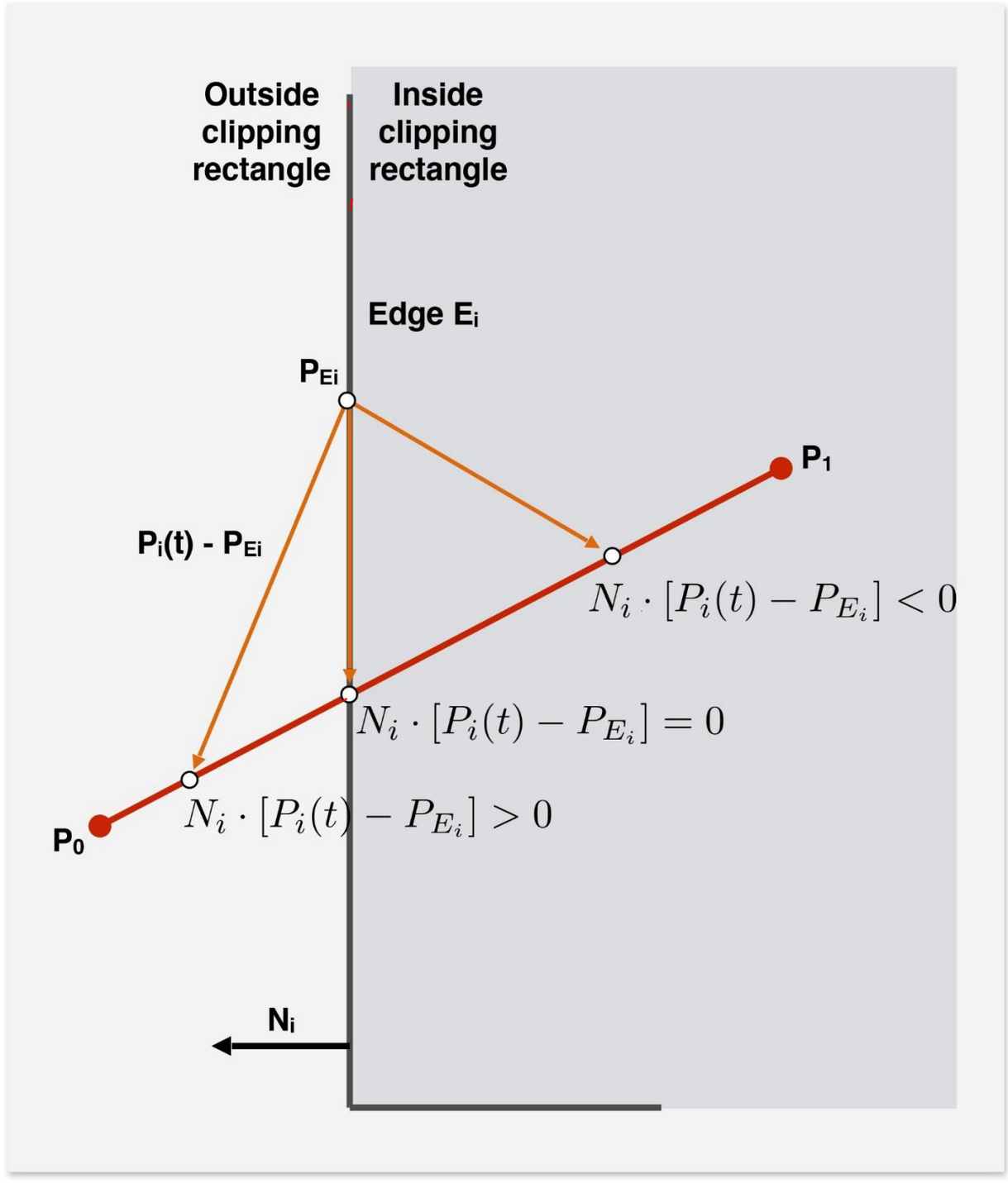
Published in 1978 by Cyrus, M., Beck, J, “Generalized Two- and Three-Dimensional Clipping”, Computers and Graphics, vol 3(1)

$$N_i \cdot [P_i(t) - P_{E_i}] = \|N_i\| \|P_i(t) - P_{E_i}\| \cos\theta$$

$N_i \cdot [P_i(t) - P_{E_i}] < 0$ ➡ P inside

$N_i \cdot [P_i(t) - P_{E_i}] = 0$ ➡ P on the edge

$N_i \cdot [P_i(t) - P_{E_i}] > 0$ ➡ P outside



Cyrus-Beck algorithm

Solve the equation

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

$$\mathbf{P(t)} = (1 - t)\mathbf{P_0} + t\mathbf{P_1}$$

Substitute $P(t) = P_0 + (P_1 - P_0)t$

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

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot [P_1 - P_0]t = 0$$

Let $D = (P_1 - P_0)$ be the vector from P_0 to P_1, then

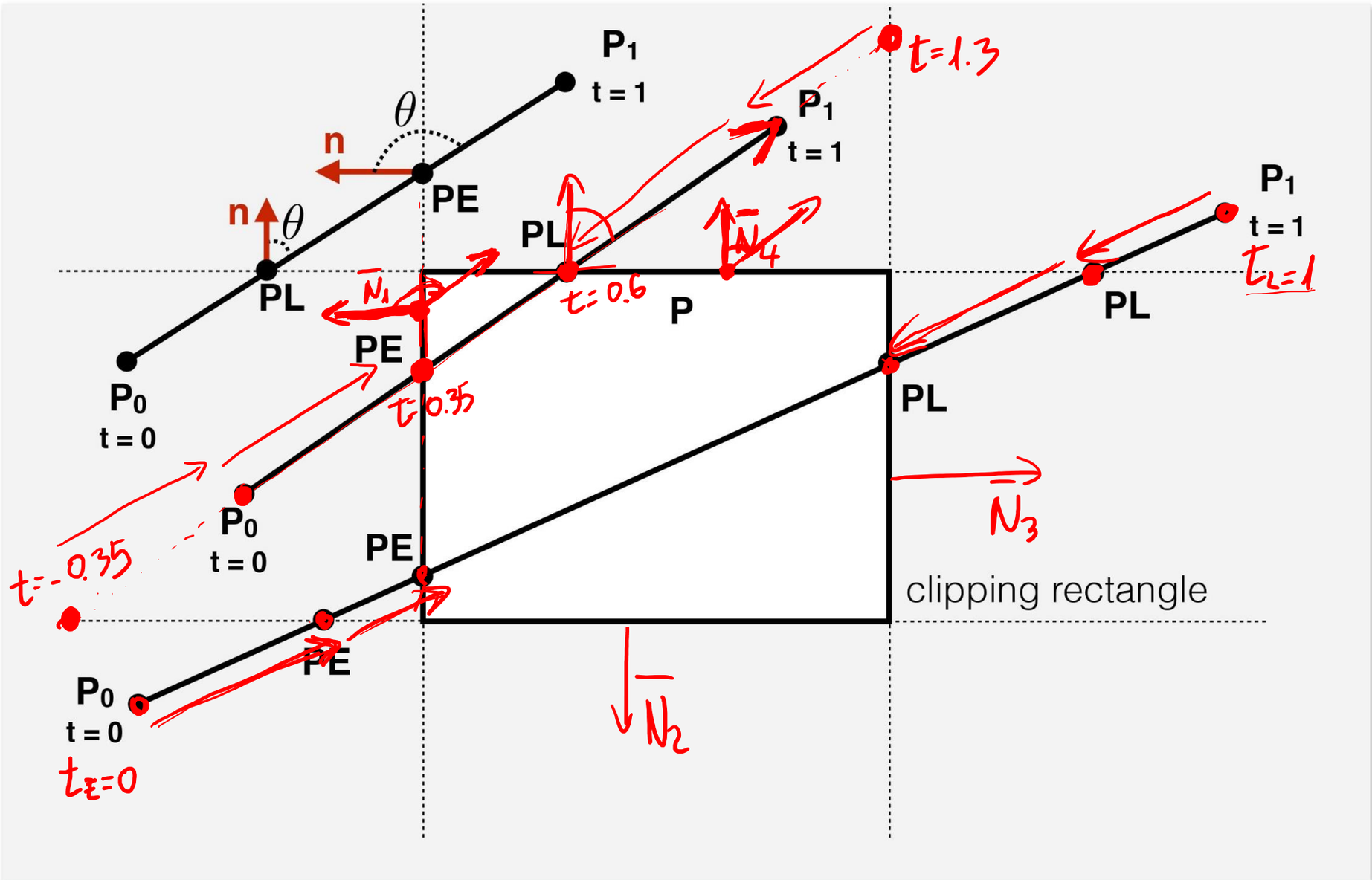
$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}$$

-> apl. pt. fiecare edge de clipping

The algorithm checks that

$N_i \neq 0$	the normal should not be 0 (error case)
$D \neq 0$	$P_1 \neq P_0$
$N_i \cdot D \neq 0$	the edge E_i and the line P_0 to P_1 are not parallel

Cyrus-Beck algorithm



PE = entering point: $N_i \cdot D < 0, \cos\theta < 0, \theta > 90^o$

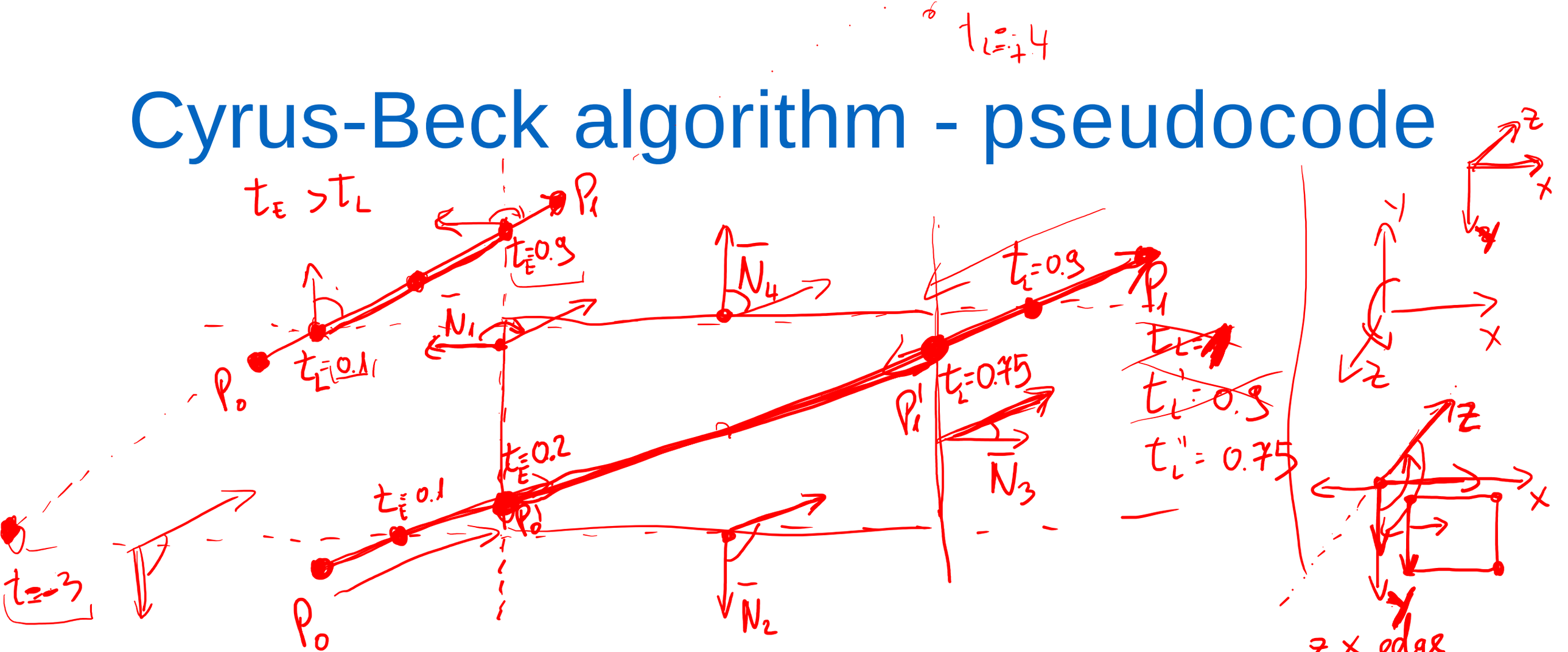
PL = leaving point: $N_i \cdot D > 0, \cos\theta > 0, \theta < 90^o$

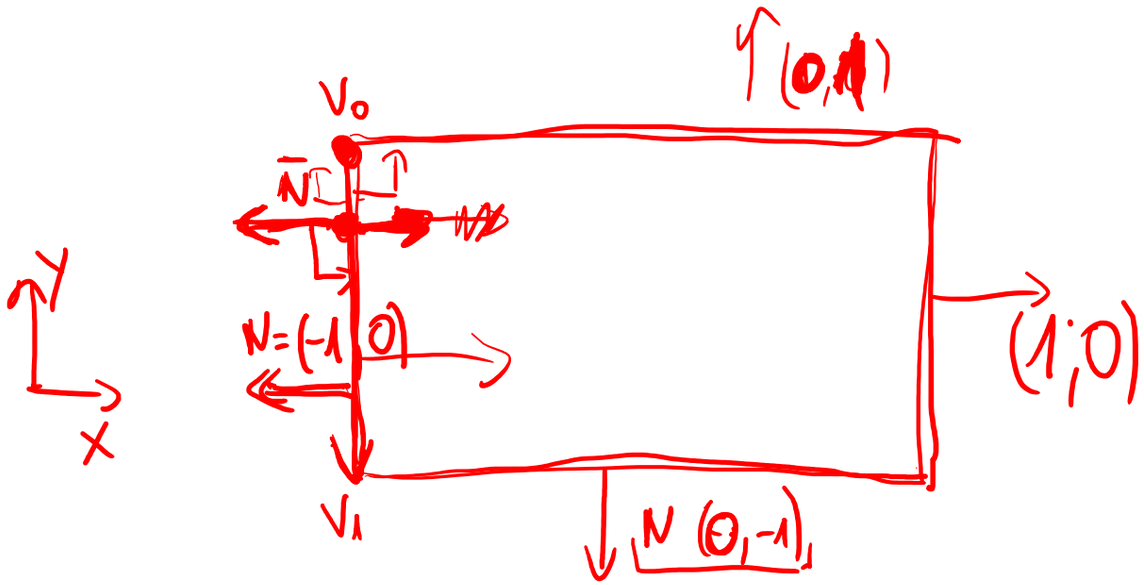
$D = (P_1 - P_0)$

Cyrus-Beck algorithm - pseudocode

```
precalculate Ni and select a PEi for each edge;
for (each line segment to be clipped){
   if (P1 = P0)
      line degenerates to a point, so clip as a point;
   else
      tE = 0; tL = 1;
      for (each candidate compute intersection with a clipping edge){
         if (Ni * D !=0){ //ignore edges parallel to line
            calculate t;
            use sign of Ni * D to categorize as PE (potentially entering) or PL (potentially
            leaving);
            if (PE)
               tE = max (tE , t);
            if (PL)
               tL = min (tL , t);
         }
      }
      if (tE > tL)
         return  NULL
      else
        return P(tE ) and  P(tL ) as true clip intersections
}
```

Cyrus-Beck algorithm - pseudocode



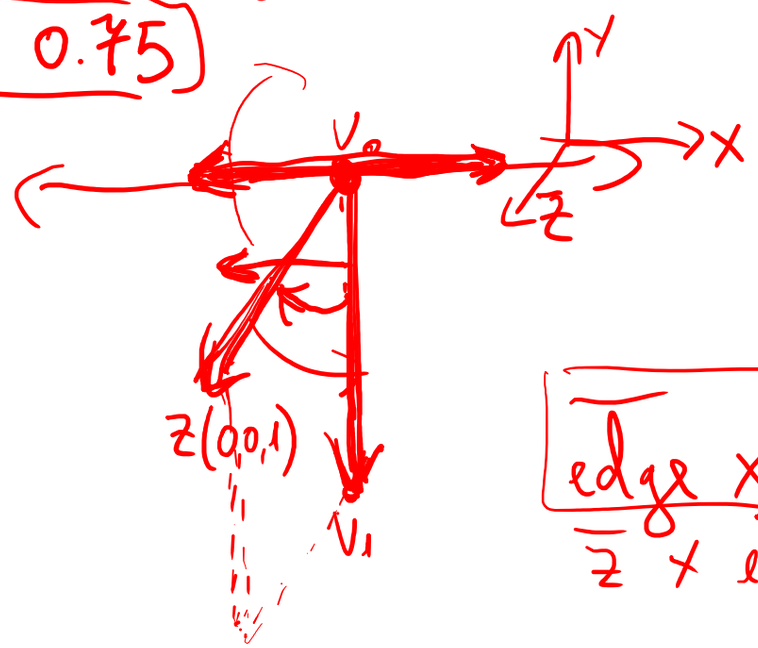


$t_E = 0.2$

$t_L = 0.75$

$\Rightarrow$

$P(t) = (1-t) \cdot P_0 + t \cdot P_1$

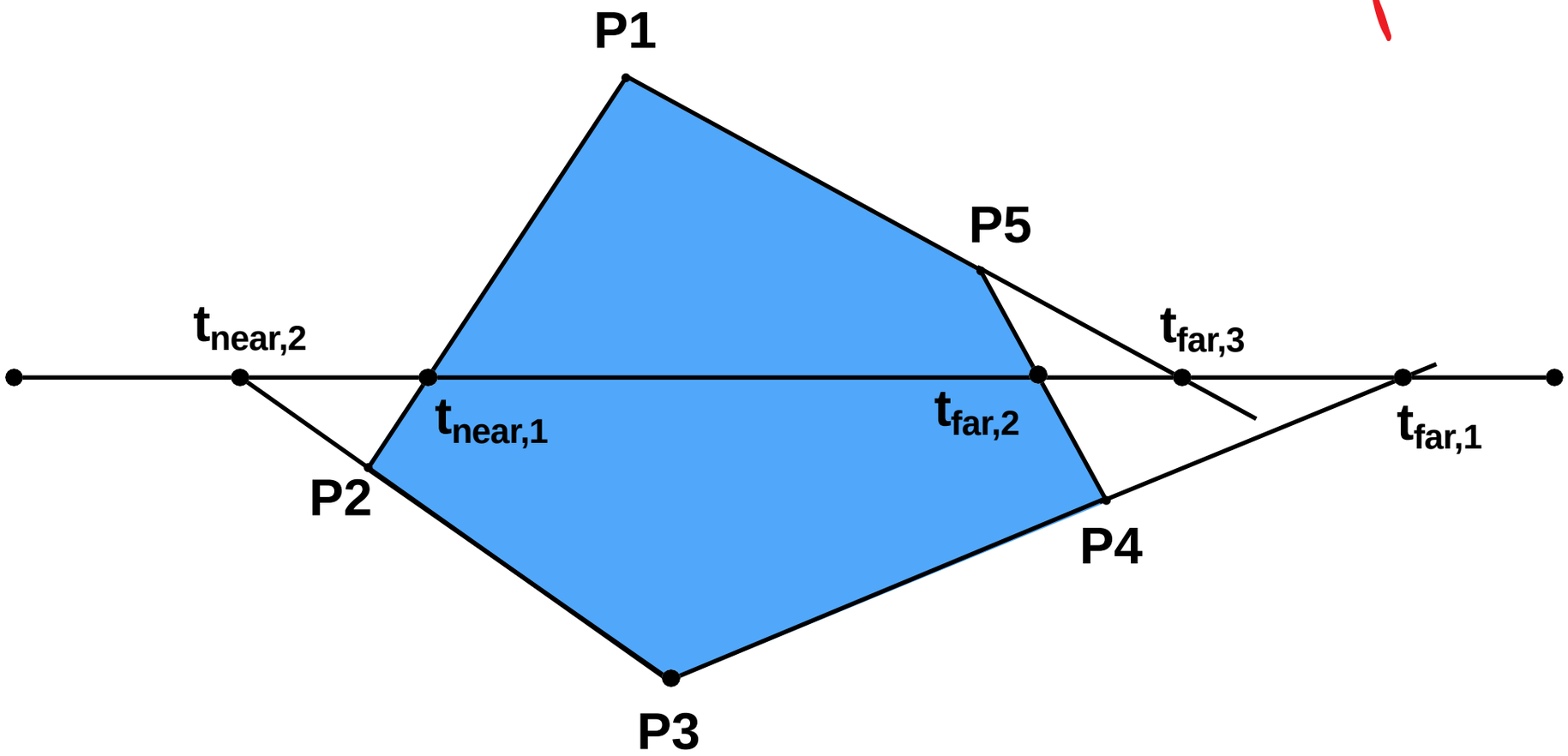


$\overline{edge} \times \bar{z}$

$\bar{z} \times edge$

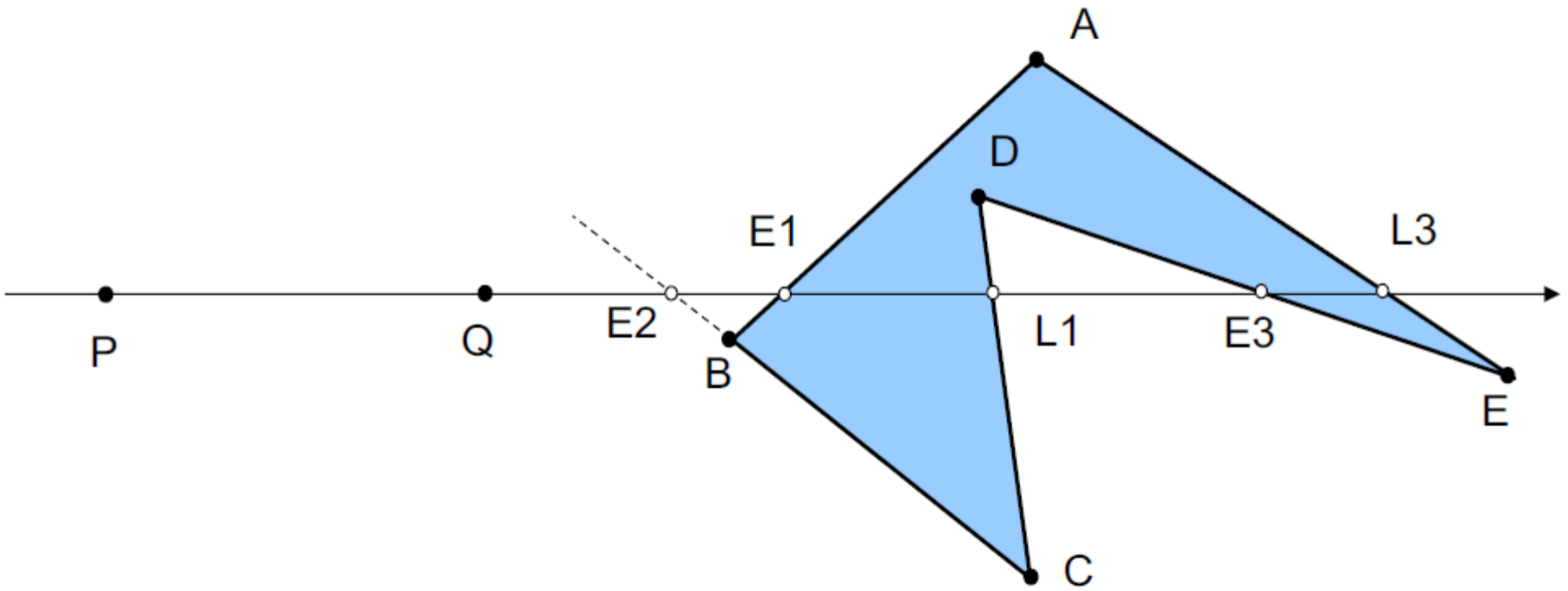
Cyrus-Beck algorithm

- poligon convex folosit ca "fereastră" de decupare



Cyrus-Beck algorithm

requires a convex polygon!



Liang-Barsky line clipping algorithm

- Published in 1984 by Liang, Y-D., Barsky, B.A., “A New Concept and Method for Line Clipping”, ACM TOG, vol 3(1).
- Simplifies the Cyrus-Beck algorithm for the particular case of clipping window as an horizontal rectangle:

1. One coordinate of each normal is 0.
2. The dot product $N_i \cdot (P_0 - P_{Ei})$ determining whether the endpoint P0 lies inside or outside a specific edge, reduces to the directed horizontal or vertical distance from the point to the edge.
3. The denominator dot product $N_i \cdot D$, which determines whether the intersection is potentially entering or leaving, reduces to $\pm dx$ or dy:
   - If dx >0, the line moves from left to right and is PE for the left edge, PL for the right edge, and so on.
4. The parameter t, the ratio of numerator and denominator, reduces to the distance to an edge divided by dx or dy, exactly the constant of proportionality we could calculate directly from the parametric line formulation.

Liang-Barsky algorithm - parameters

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

Liang-Barsky algorithm

```
boolean ClipT(double denom, double num, double *tE, double *tL)
{
/* computes a new value of tE or tL for an interior intersection of a line segment and edge. Parameter
denom is −(Ni · D), which reduces to ±Δx, Δy for upright rectangle. Its sign determines whether the
intersection is PE or PL. parameter num is Ni · D(P0-PEi) for a particular edge/line combination, which
reduces to directed horizontal and vertical distances from P0 to an edge; If the line segment can be
trivially rejected, false is returned; if it cannot be, true is returned and the value of tE or tL is
adjusted, if needed, for the portion of the segment that is inside the edge*/
    double r;
    if(denom > 0){                  // PE intersection
        t = num / demon;            // value of t at the intersection
        if(t > tL)                  // tE and tL crossover
          return FALSE: // so prepare to reject line
        else if(t > tE)     // a new tE has been found
          tE = t;
    }else if (denom < 0){           // PL intersection
        t = num / denom;            // value of t at the intersection
        if(t < tE)                  // tE and tL crossover
          return FALSE; // so prepare to reject line
        else                // a new tL has been found
          tL = t;
    }else if (num > 0)              // line on outside of edge
        return FALSE;
    return TRUE;
}
```

Liang-Barsky algorithm

```
void Clip2D(double *x0, double *y0, double *x1, double *y1, boolean *visible)
{/* clip a line with endpoints (x0,y0) and (x1,y1), against the rectangle with corners (xmin,ymin) and (xmax,ymax).
The flag visible is true if a clipped segment is returned in the var endpoint parameters. If the line is rejected, the
endpoints are not changed and visible is set to false*/
        double dx = *x1 - *x0;
        double dy = *y1 - *y0;
        *visible = FALSE;
        //first test for degenerate line and clip the point; ClipPoint returns true if the point lies inside the clip rectangle
        if(dx == 0 && dy == 0 && ClipPoint(*x0, *y0))
                *visible = TRUE;
        else{
                double tE = 0.0;
                double tL = 1.0;
                if(ClipT(dx, xmin - *x0, &tE, &tL))                  // inside against left edge
                   if(ClipT(-dx, *x0 – xmax, &tE, &tL))              // inside against right edge
                      if(ClipT(dy, ymin - *y0, &tE, &tL))            // inside against bottom edge
                        if(ClipT(-dy, *y0 – ymax, &tE, &tL)){        // inside against top edge
                           *visible = TRUE;
                            if(tL < 1){ //compute PL intersection, if tL has moved
                             *x1 = *x0 + tL * dx;
                             *y1 = *y0 + tL * dy;
                             }
                             if(tE > 0){ //compute PE intersection, if tE has moved
                             *x0 += tE * dx;
                             *y0 += tE * dy;
                             }

        }
}
```

Conclusions

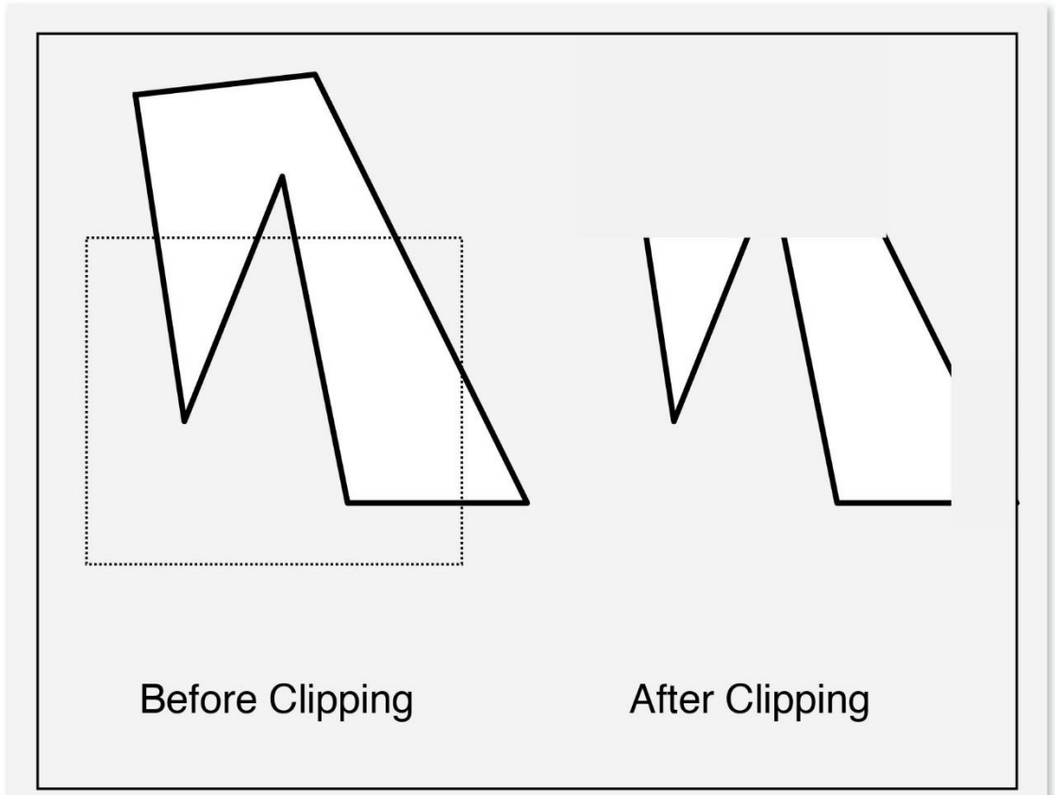
- Cohen-Sutherland algorithm is efficient when:
  - Outcode testing can be done cheaply, e.g. by doing bitwise operations in assembly language
  - Trivial acceptance or rejection is applicable to the majority of line segments
- Parametric line clipping is efficient when:
  - Many line segments need to be clipped, since the actual calculations of the coordinates of the intersection points is postponed until needed, and testing can be done on parameter values.
- Liang-Barsky algorithm is more efficient than Cyrus-Beck
  - Trivial rejection testing can avoid calculation of all four parameter values for lines that do not intersect the clip rectangle.

Polygon Clipping

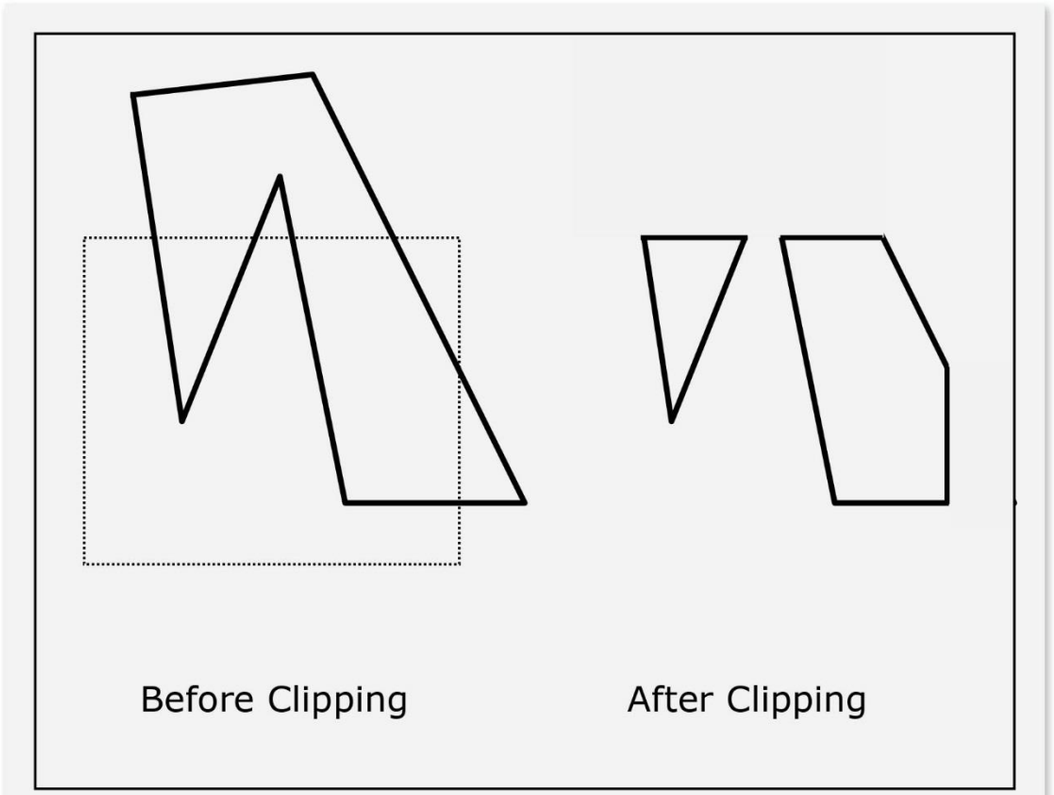
- Polygon definition:
  - set of points → point clipping
  - set of lines → line clipping
  - sequence of lines →
    - Sutherland – Hodgman algorithm
    - Weiler-Atherton algorithm

- General definition:
  - Sequence of vertices with implicit relationships
  - By clipping: 1 polygon (v1, v2, …., vn) →
    - p polygons (A11, A12, …, A1k; …Ap1, Ap2, …,Apq)

Polygon Clipping



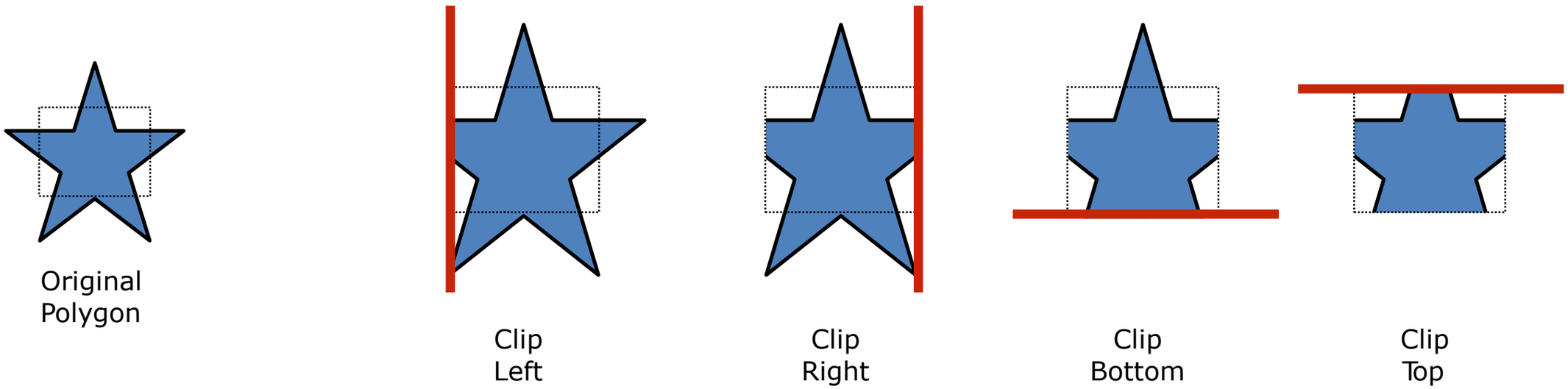
Line clipping



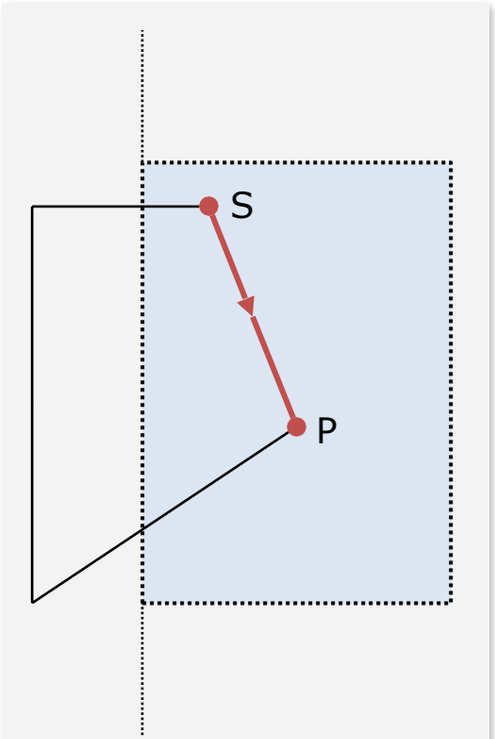
Polygon clipping

Sutherland Hodgman algorithm

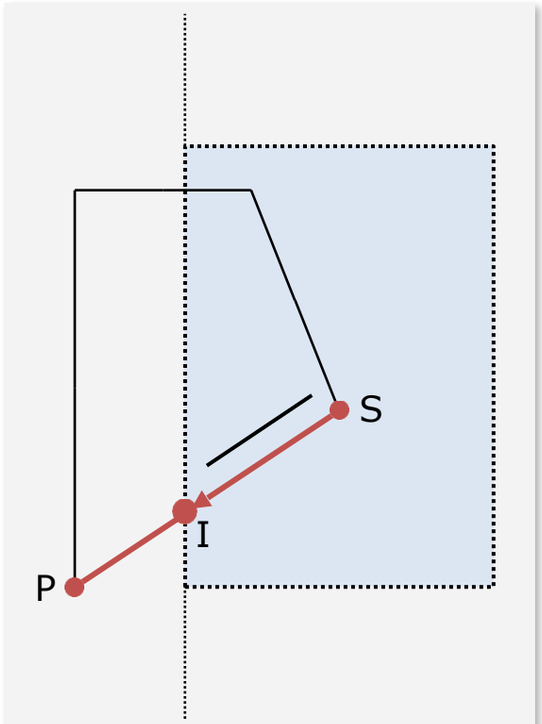
- Reentrant polygon clipping
- Published in 1974 by Sutherland, I.E., Hodgman, G.W., “Reentrant Polygon Clipping”, CACM, vol 17(1).



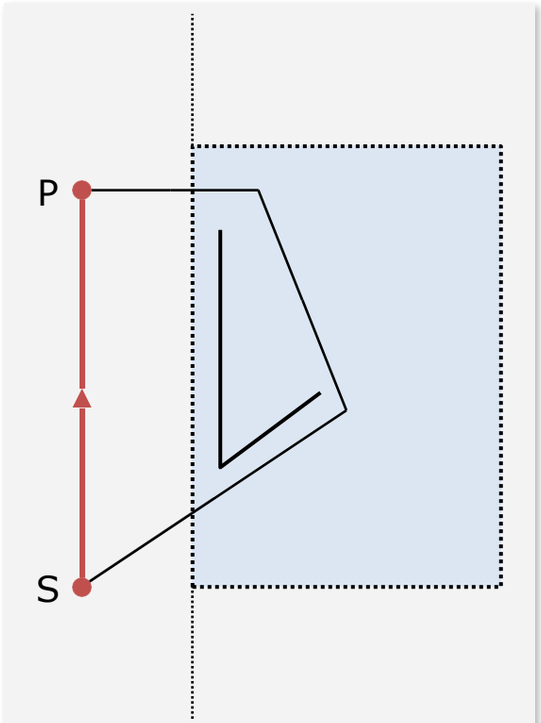
Window – polygon edge relationships



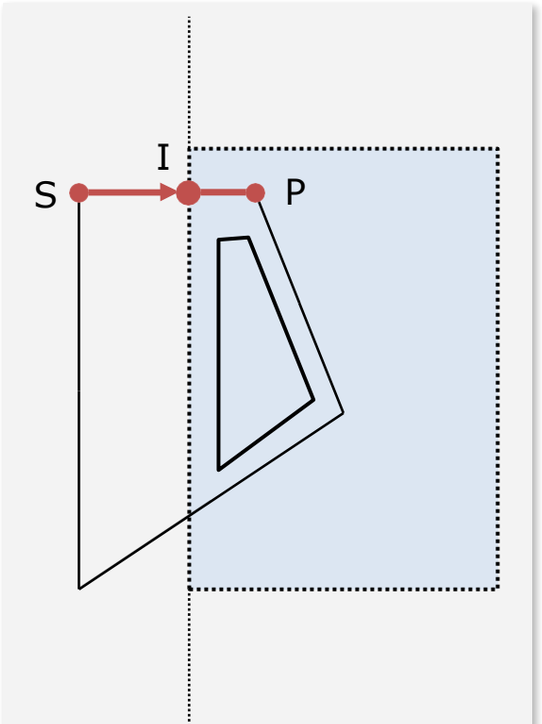
(a) Save P



(b) Save I



(c) No points saved



(d) Save I, P

Sutherland Hodgman algorithm

Input vertex list: inv[] = {v1, v2,..., vn}
Output vertex list: outv[ ] = {v1', v2',..., vp'}

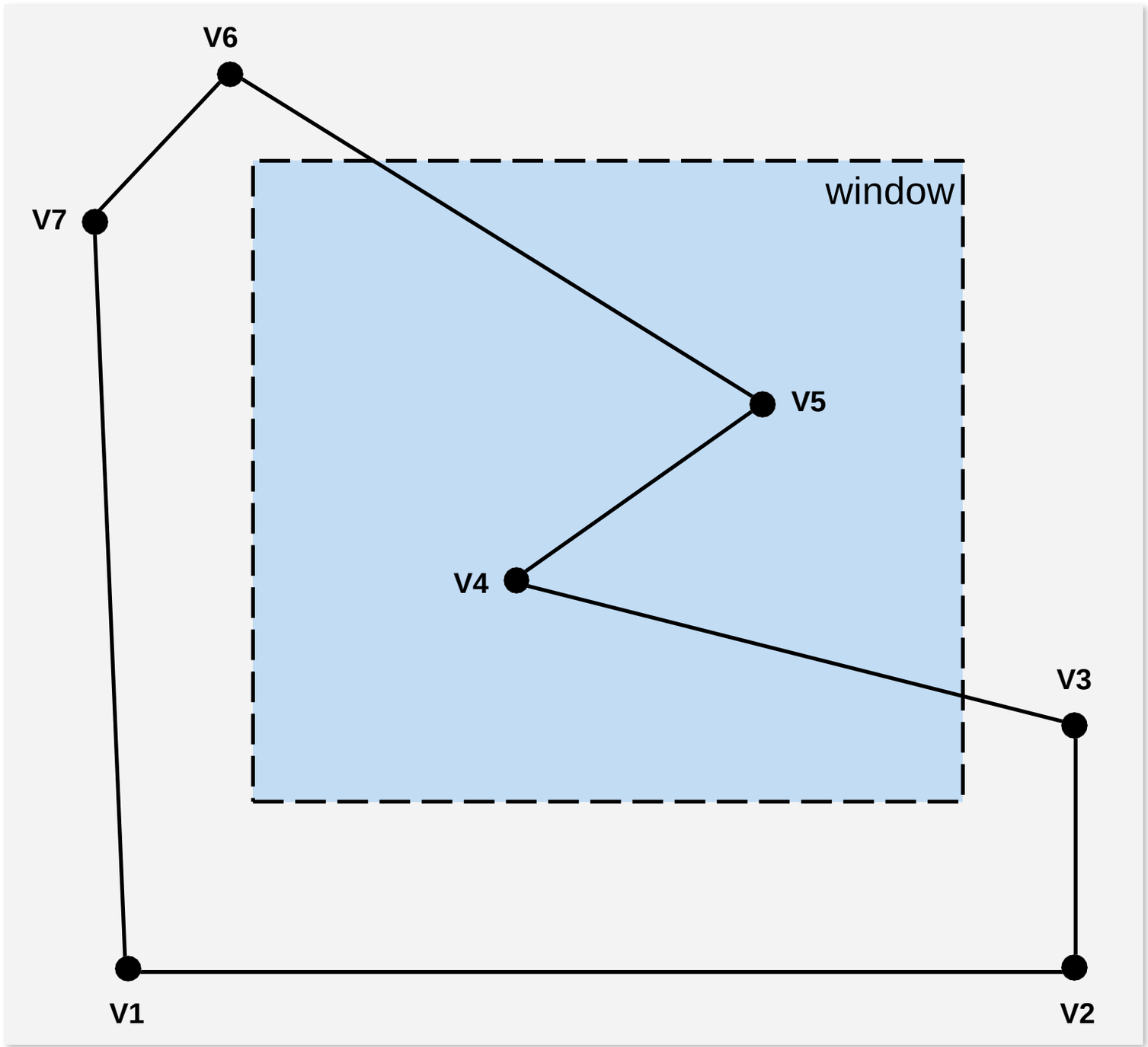
```
ClippingSH(vertexList inputVertices, vertexList outputVertices, windowEdge clippingEdges[4])
{
    for(each clipping edge){
        EdgeClippingSH(inputVertices, outputVertices, clippingEdges[currentEdge]);
        inputVertices = outputVertices; // update the current edge list
    }
}
```

Sutherland Hodgman algorithm

```
EdgeClippingSH(vertexList inputVertices, vertexList outputVertices, windowEdge
    clippingEdge)
{
    vertex i,p,s;
    select s to be the last vertex in inputVertices;

    for(each vertex p from inputVertices){
        if(ToWindow(p, clippingEdge))                    //case a and d
           if(ToWindow(s, clippingEdge))                 //case a
           write vertex p into outputVertices;
        else{
           i = Intersection(s, p, clippingEdge);
           write intersection i into outputVertices;
           write vertex p into outputVertices;
        }
        else if(ToWindow(s, clippingEdge)){              //case b
        i = Intersection(s, p, clippingEdge);
        write intersection i into outputVertices;
        }                                                //for case c don't write anything
    s = p;                                               //update the start vertex
    }
}
```

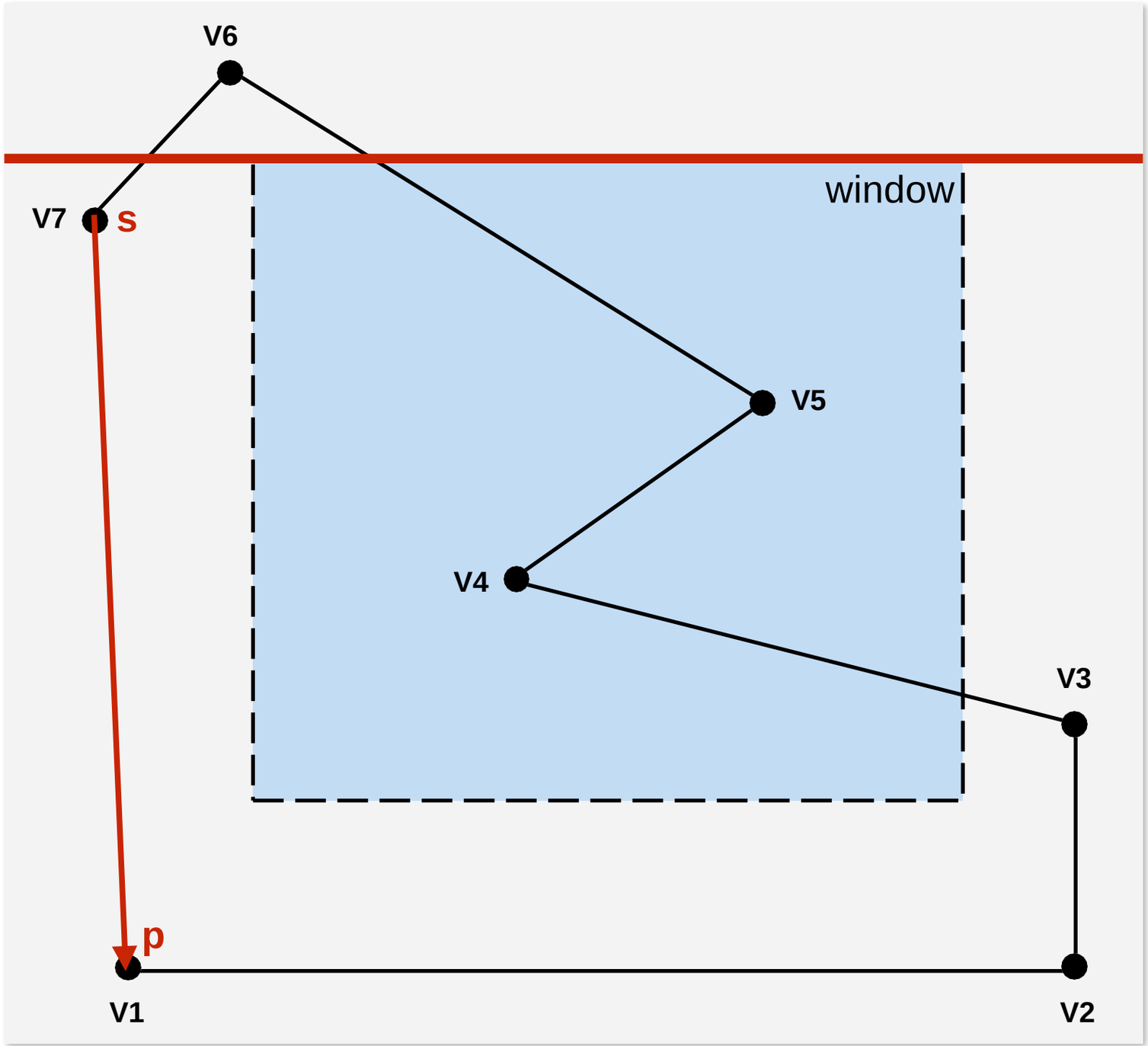
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices:

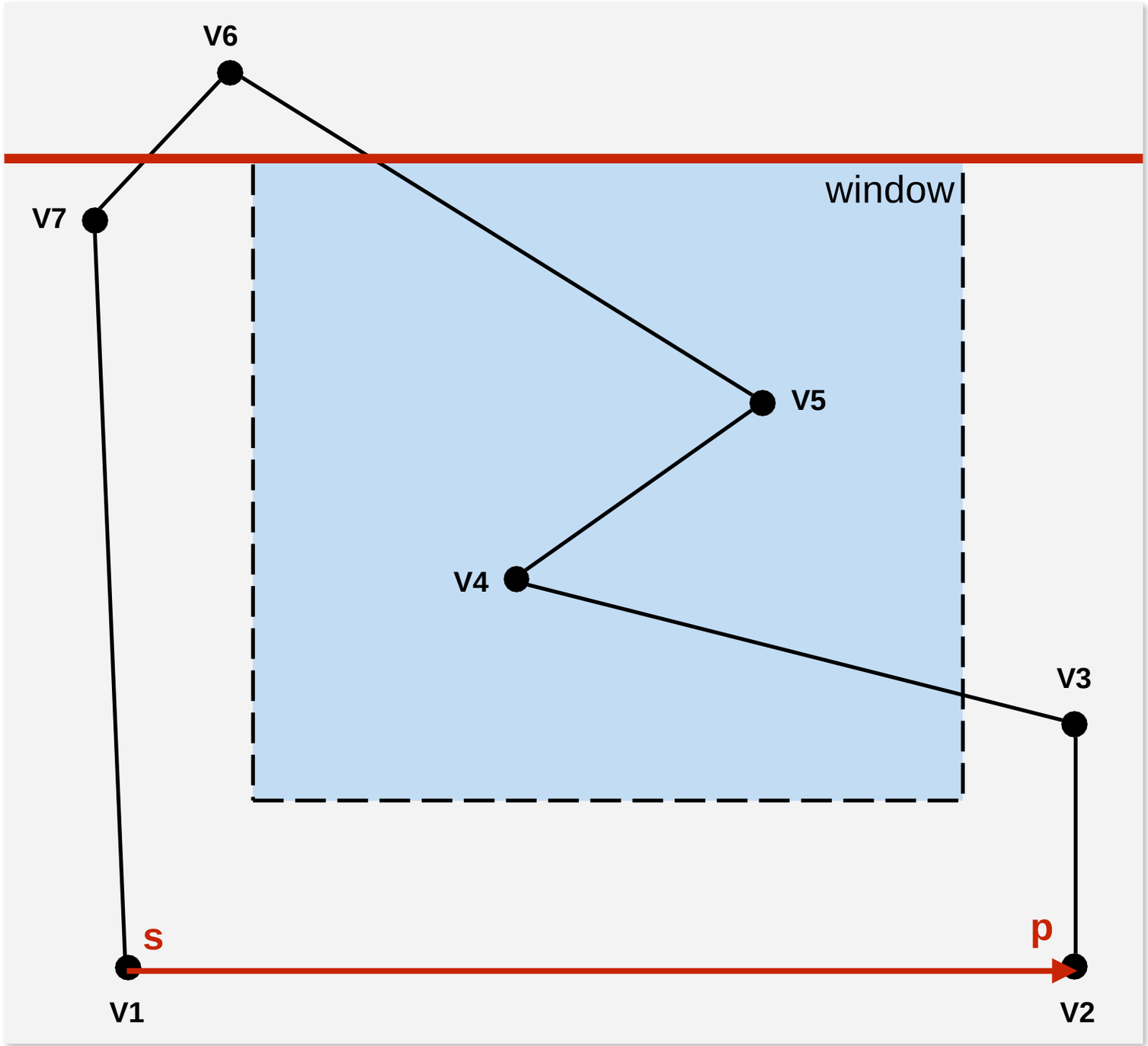
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1

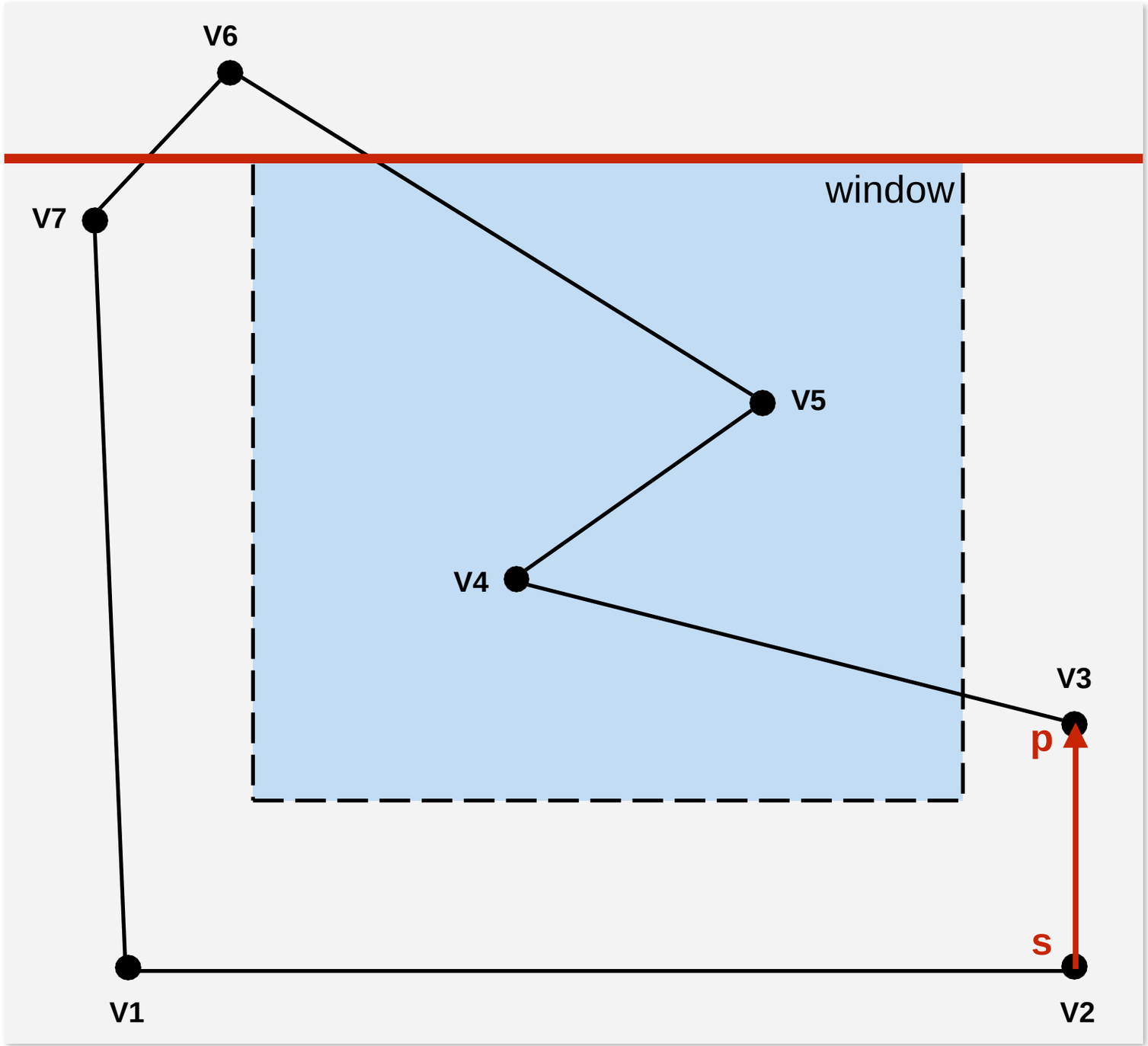
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2

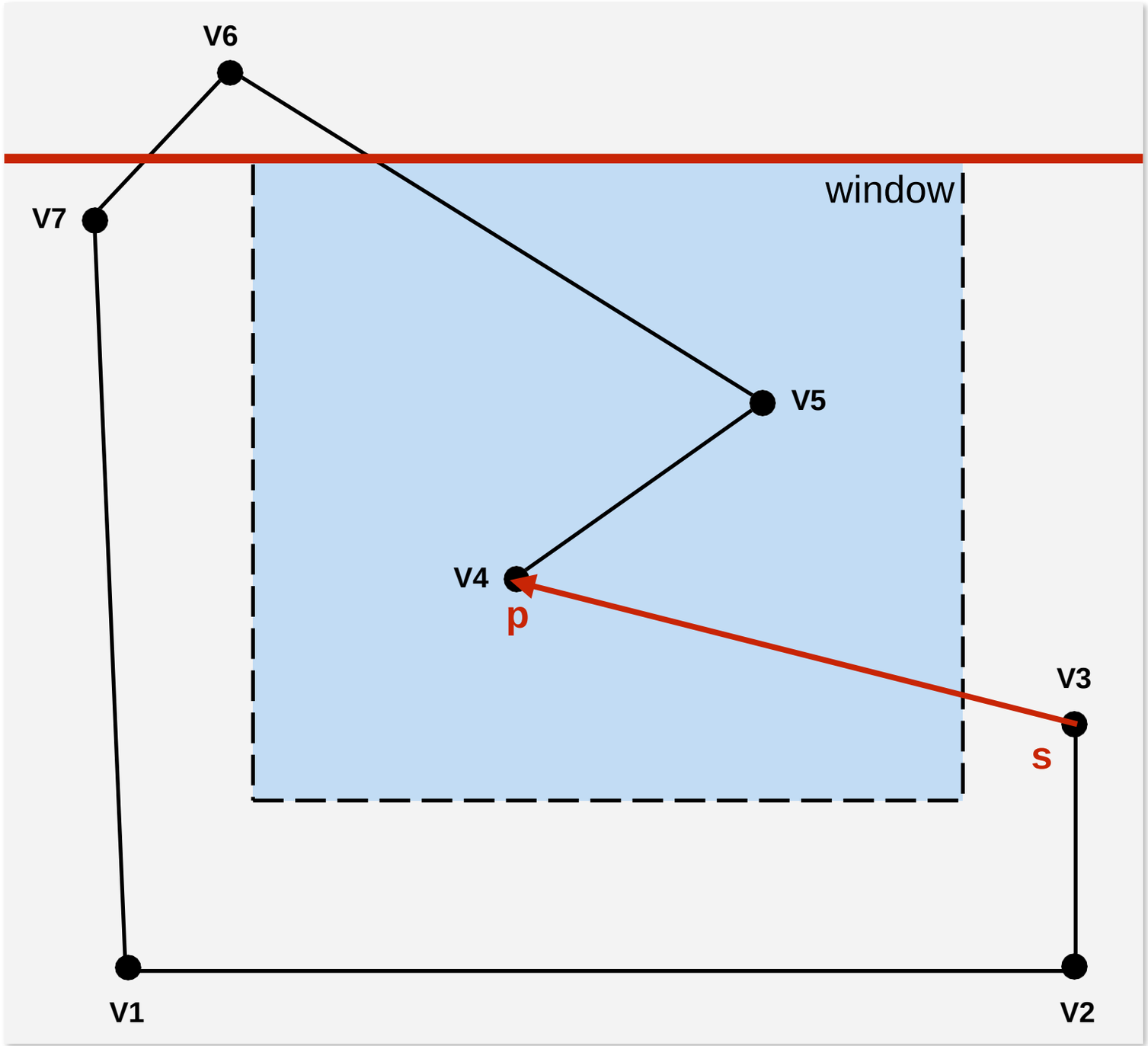
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3

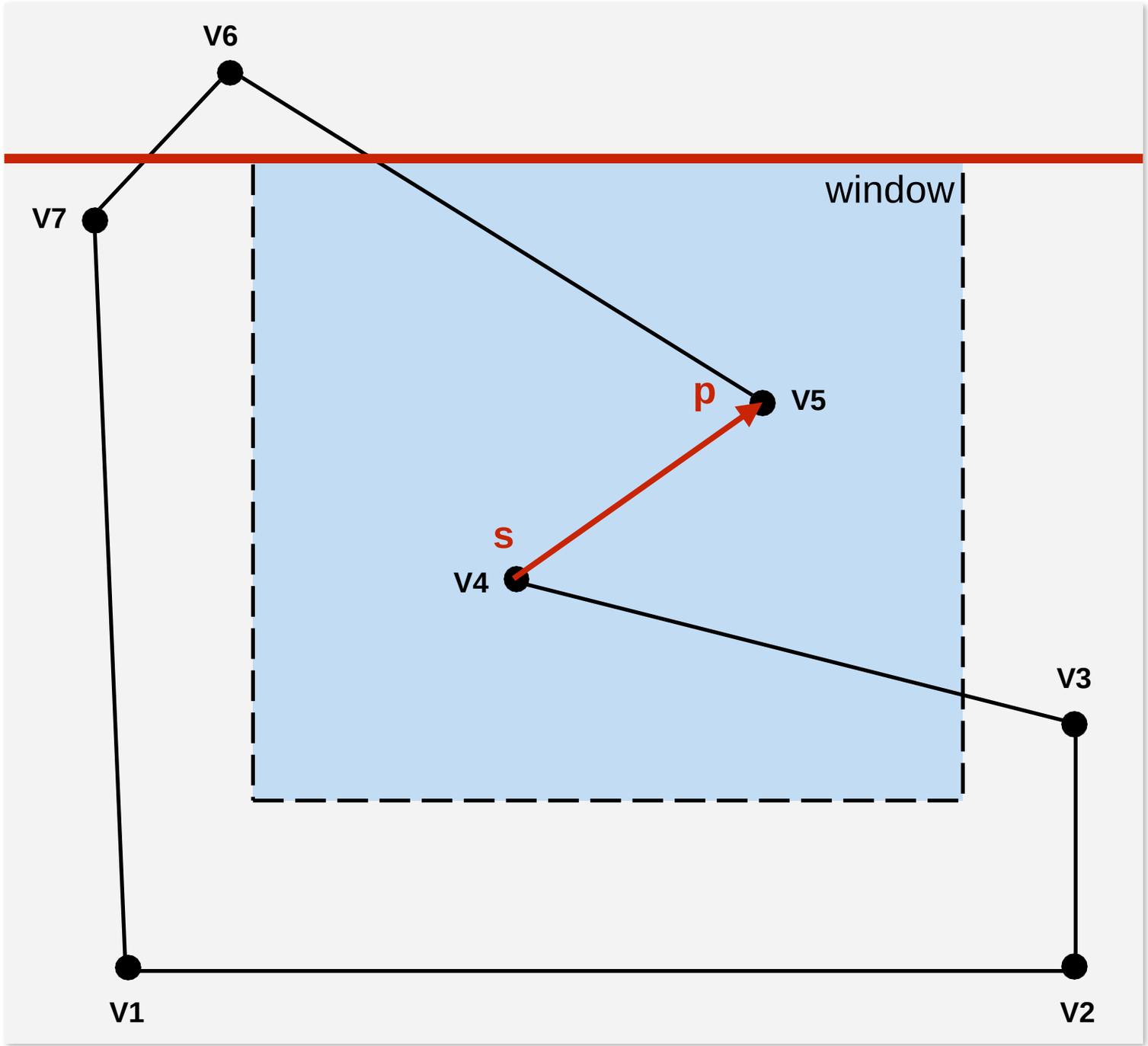
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3, V4

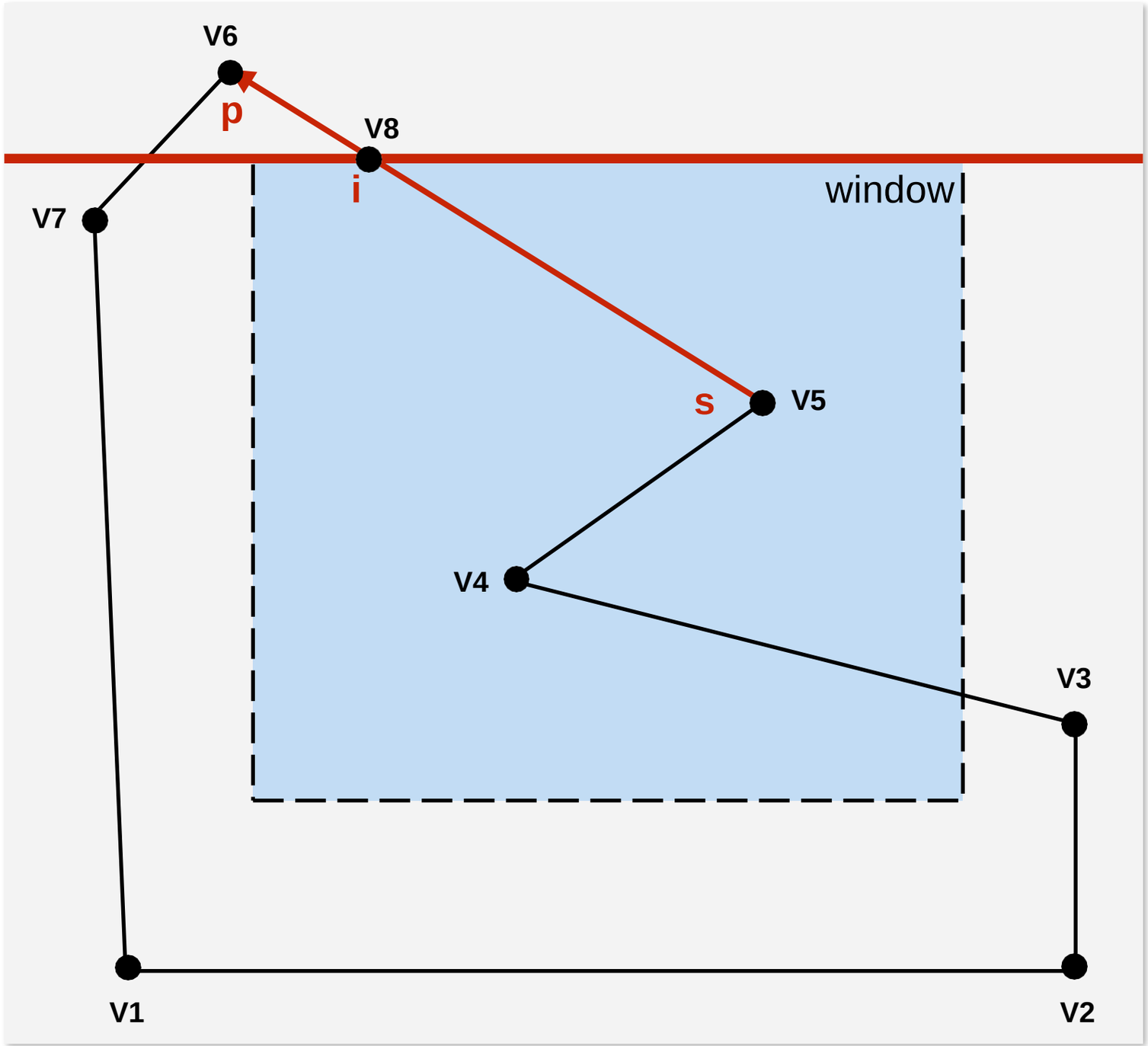
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3, V4, V5

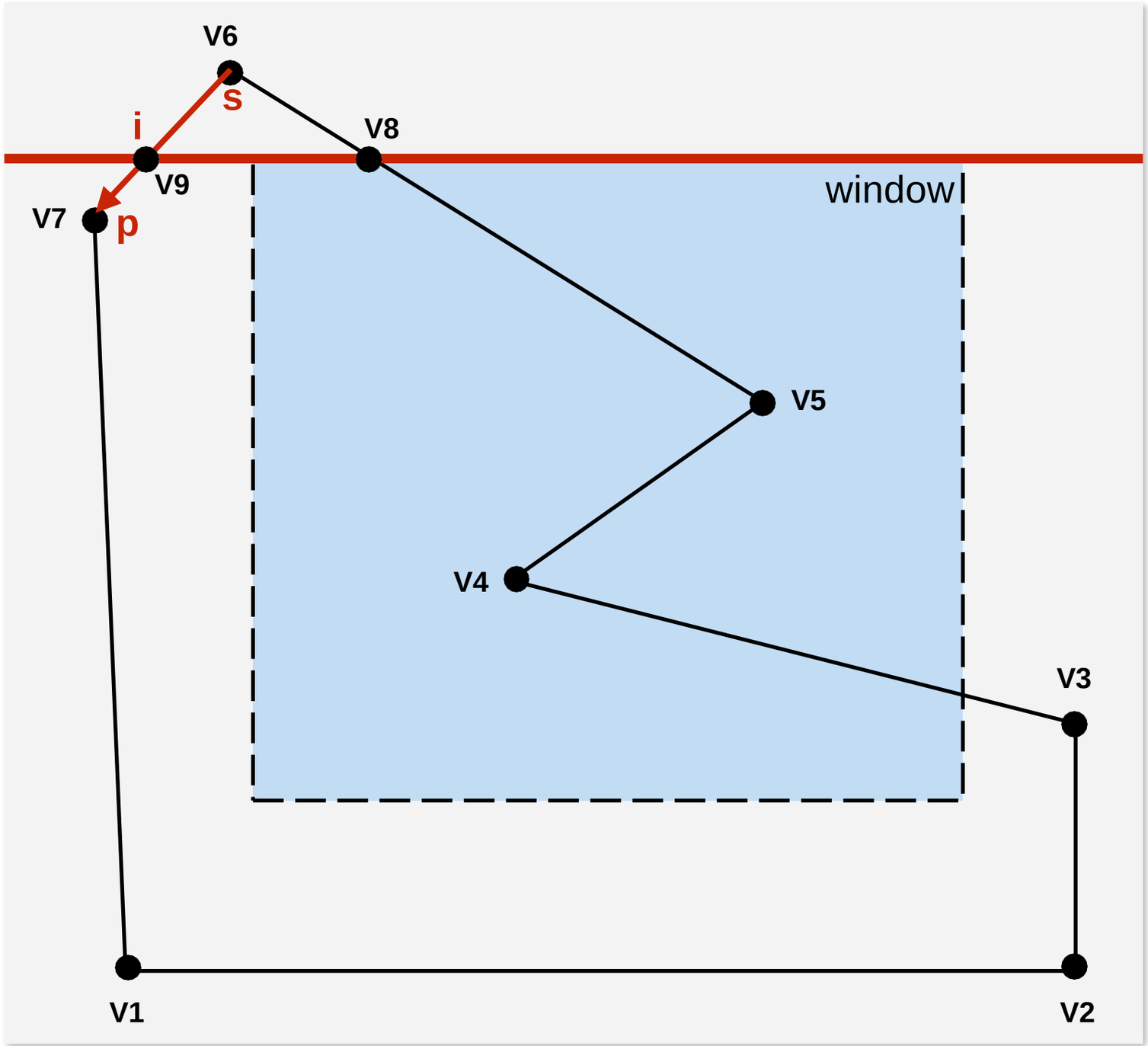
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3, V4, V5, V8

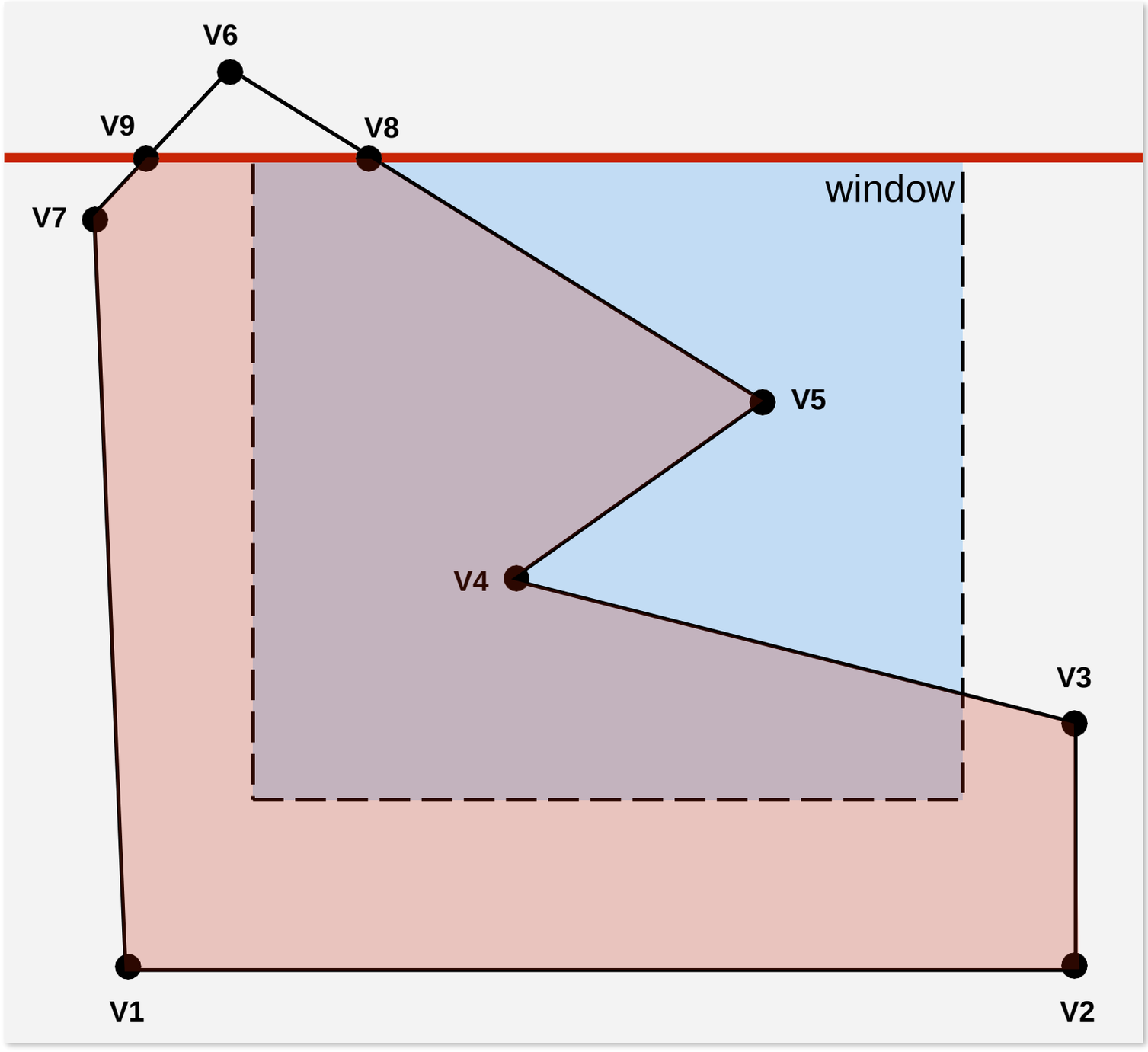
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3, V4, V5, V8, V9, V7

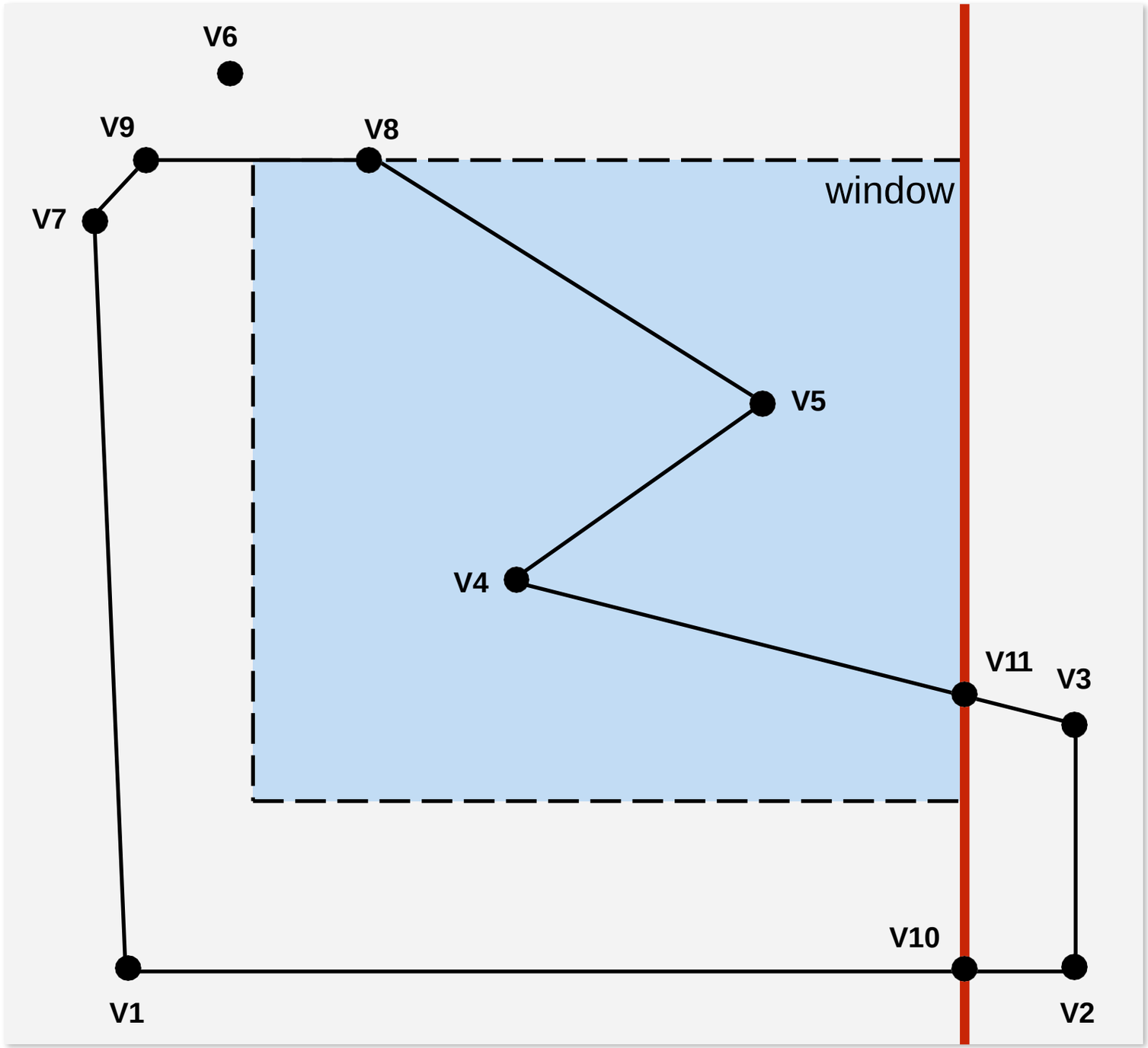
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V1, V2, V3, V4, V5, V8, V9, V7

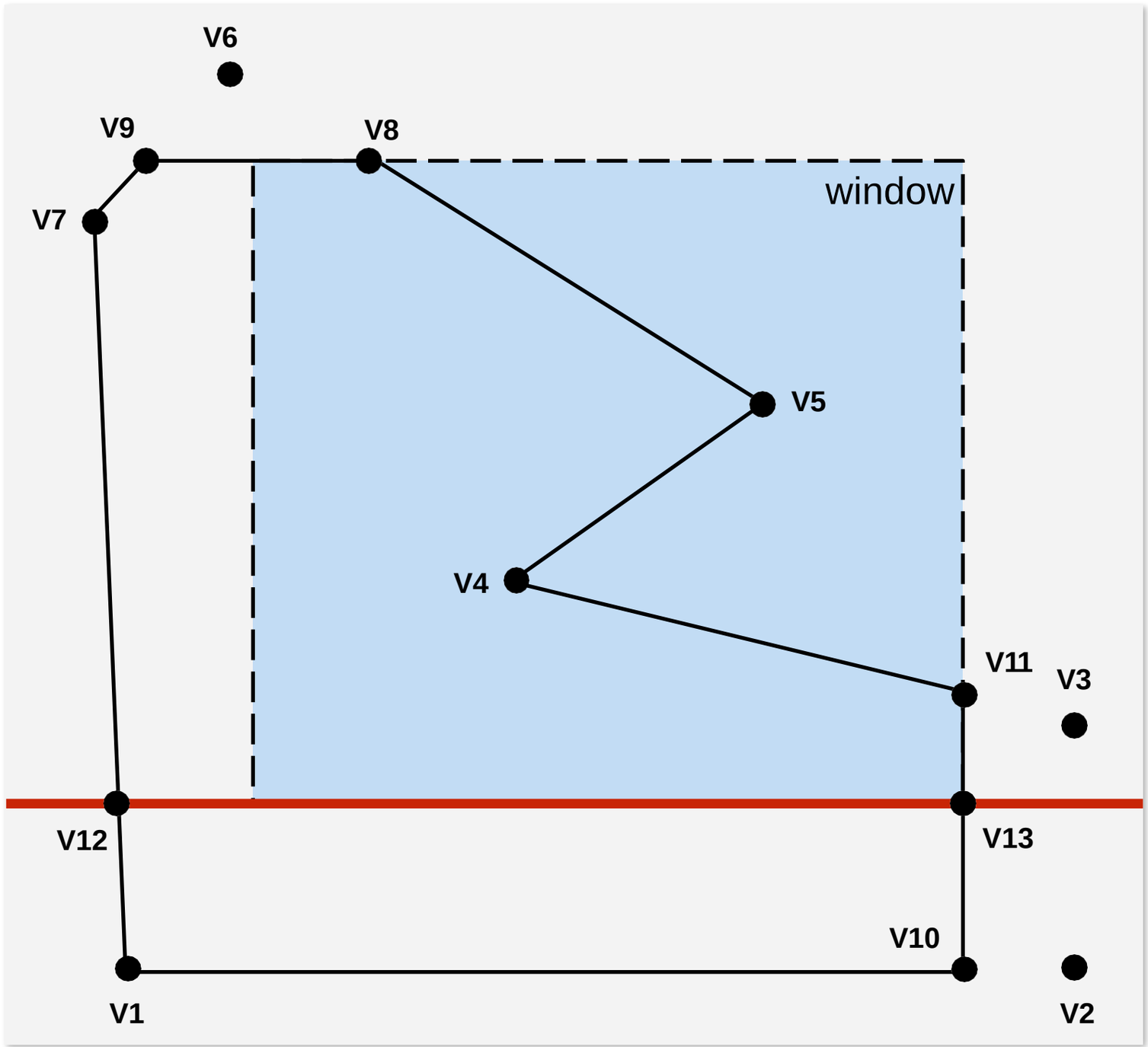
Sutherland Hodgman algorithm



inputVertices: V1, V2, V3, V4, V5, V8, V9, V7

outputVertices: V1, V10, V11, V4, V5, V8, V9, V7

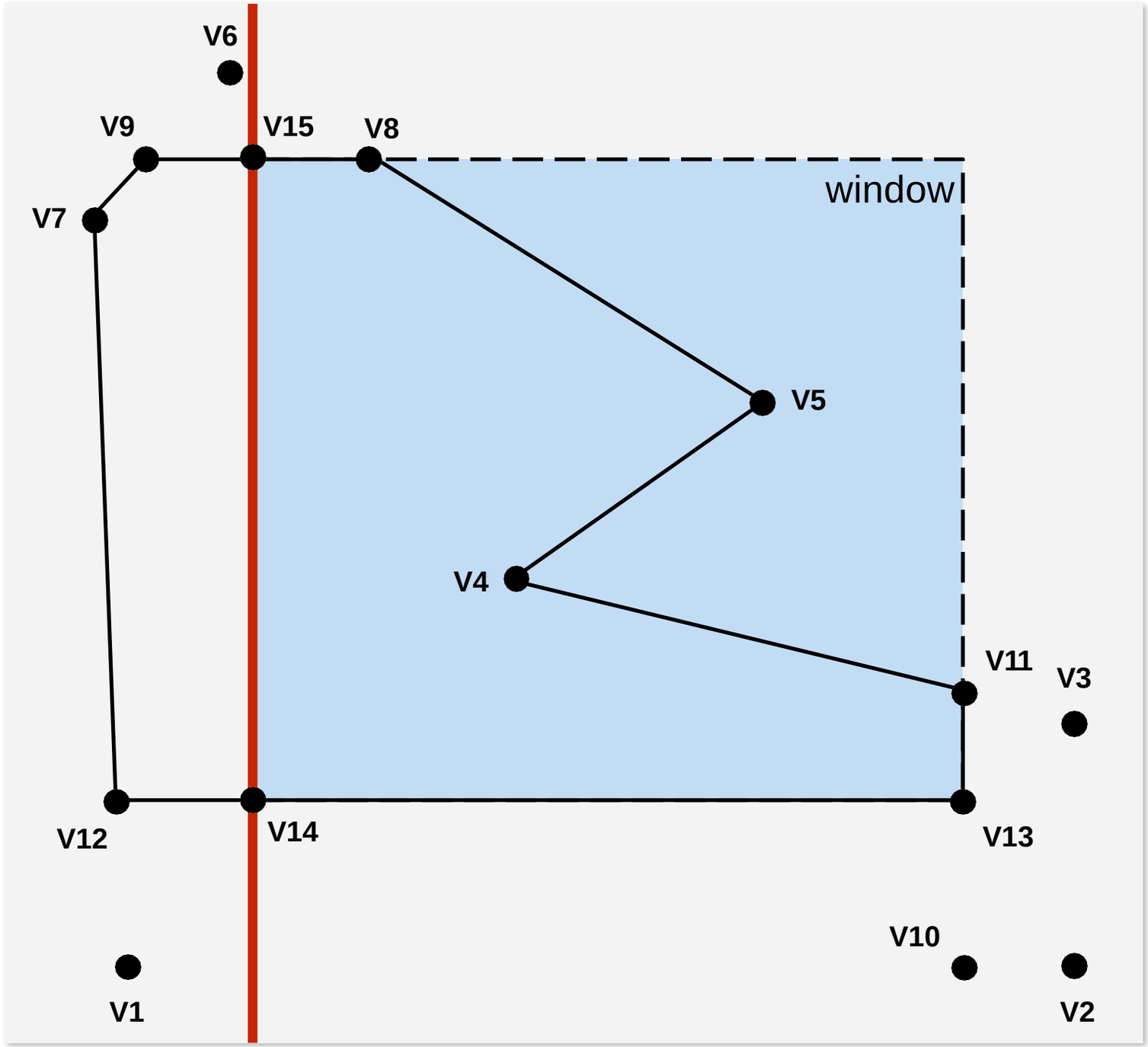
Sutherland Hodgman algorithm



inputVertices: V1, V10, V11, V4, V5, V8, V9, V7

outputVertices: V12, V13, V11, V4, V5, V8, V9, V7

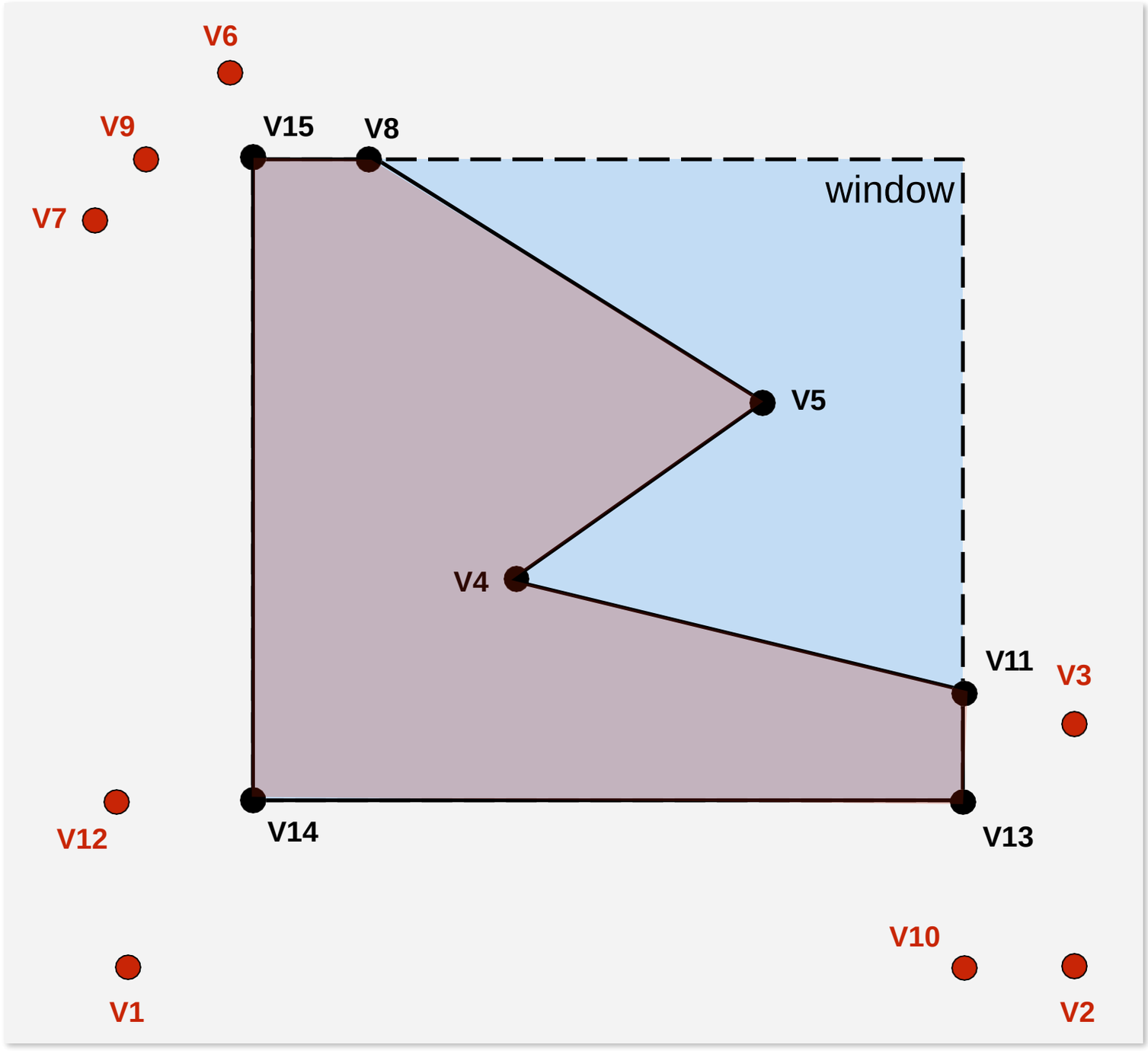
Sutherland Hodgman algorithm



inputVertices: V12, V13, V11, V4, V5, V8, V9, V7

outputVertices: V14, V13, V11, V4, V5, V8, V15

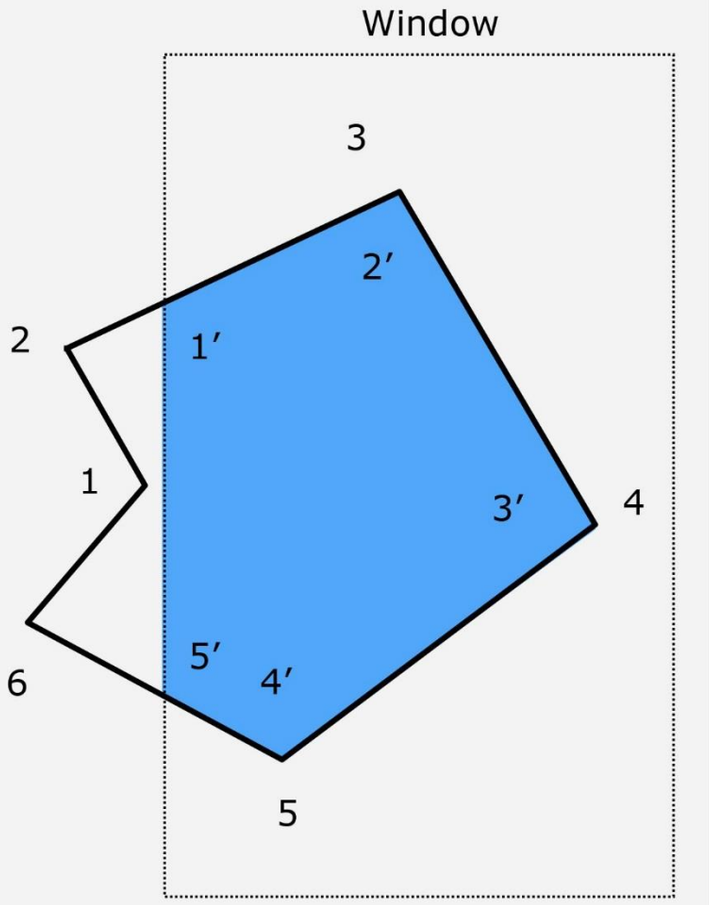
Sutherland Hodgman algorithm

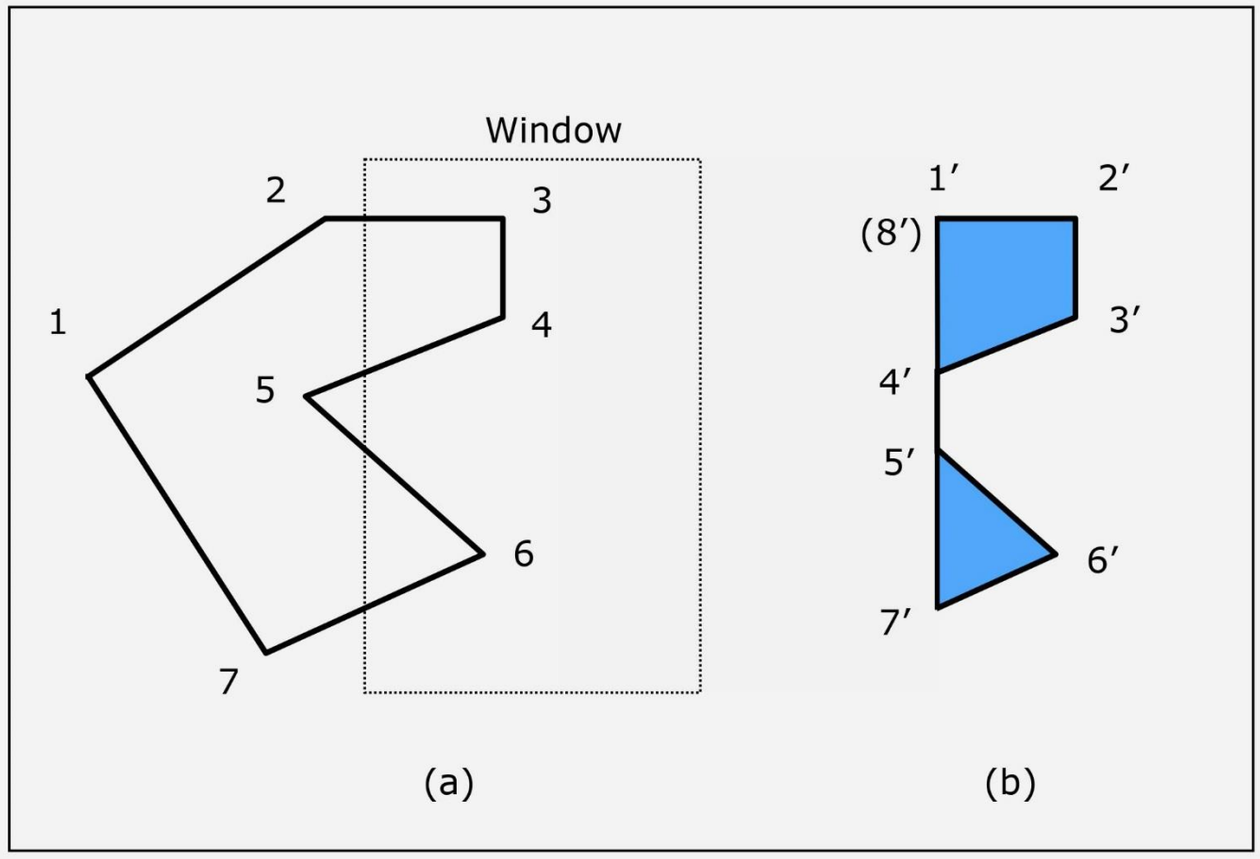


inputVertices: V1, V2, V3, V4, V5, V6, V7

outputVertices: V14, V13, V11, V4, V5, V8, V15

Sutherland Hodgman algorithm



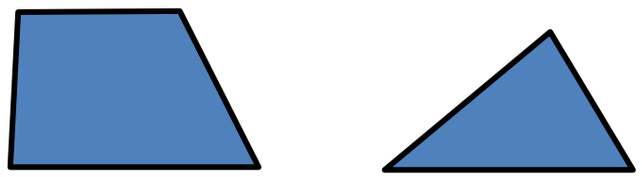


Weiler-Atherton polygon algorithm (1980)

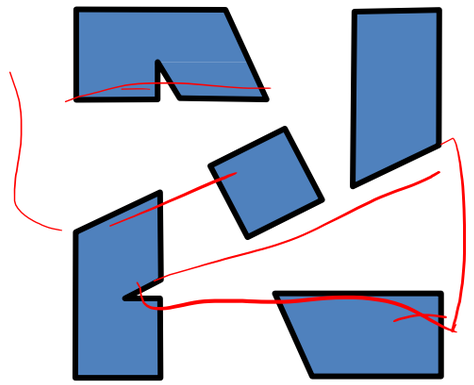
- Published in 1980 in SIGRAPH'80, "Polygon Comparison Using a Graph Representation", by Weiler, K.
- Clipping one polygon to another arbitrary polygon
- The clipped polygon may be disconnected and may be nonconvex even the original polygon was convex
- Works for any number of polygons
- Basic principle:
  - The edge of the polygon A and the polygon to be clipped B divide the drawing surface into disjoint regions
  - Each of these regions is entirely in A, enirely in B, entirely contained in both, or contained in neither
  - The clipped input polygon consists of the regions contained in both A and B

Clipping polygons against polygons

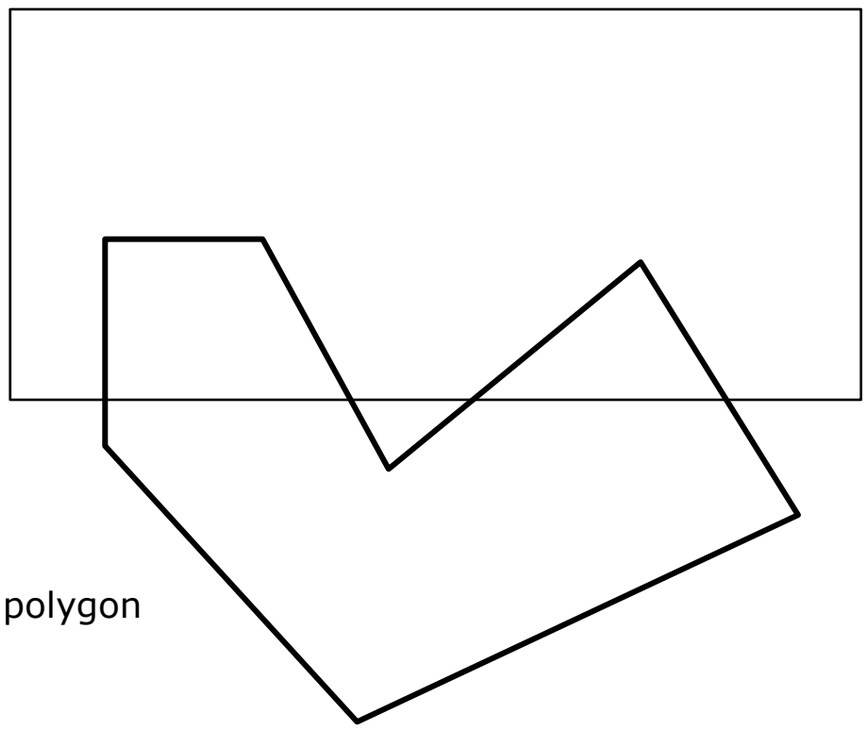
clipped polygons

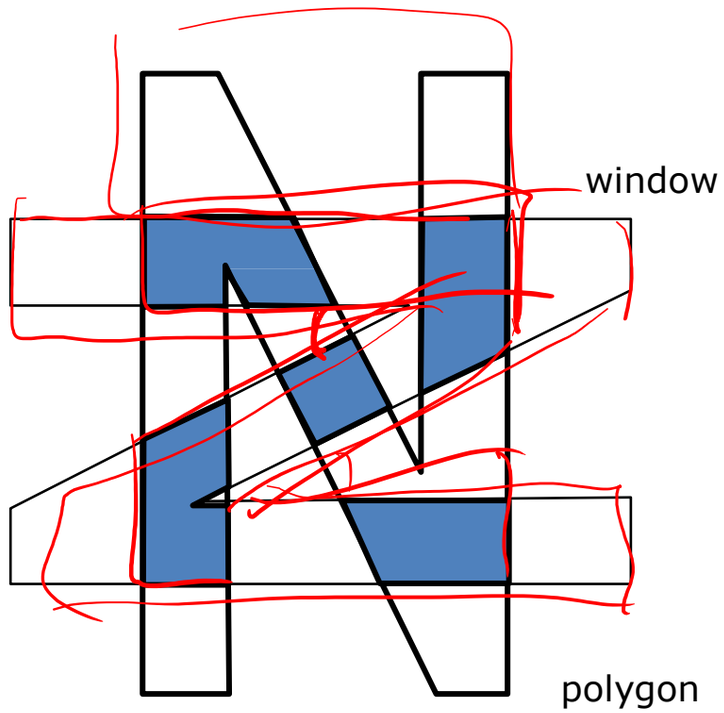


clipped polygons



window



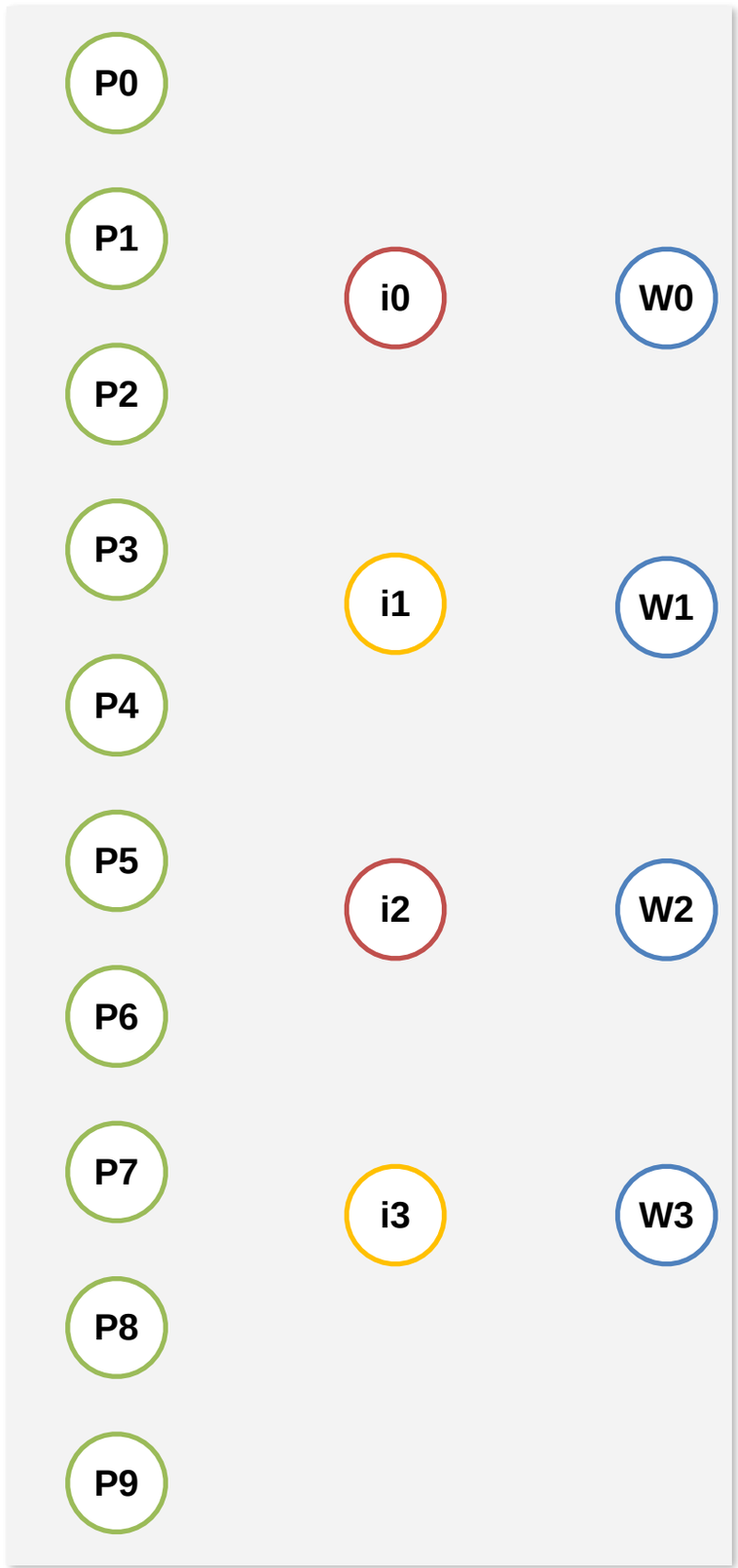


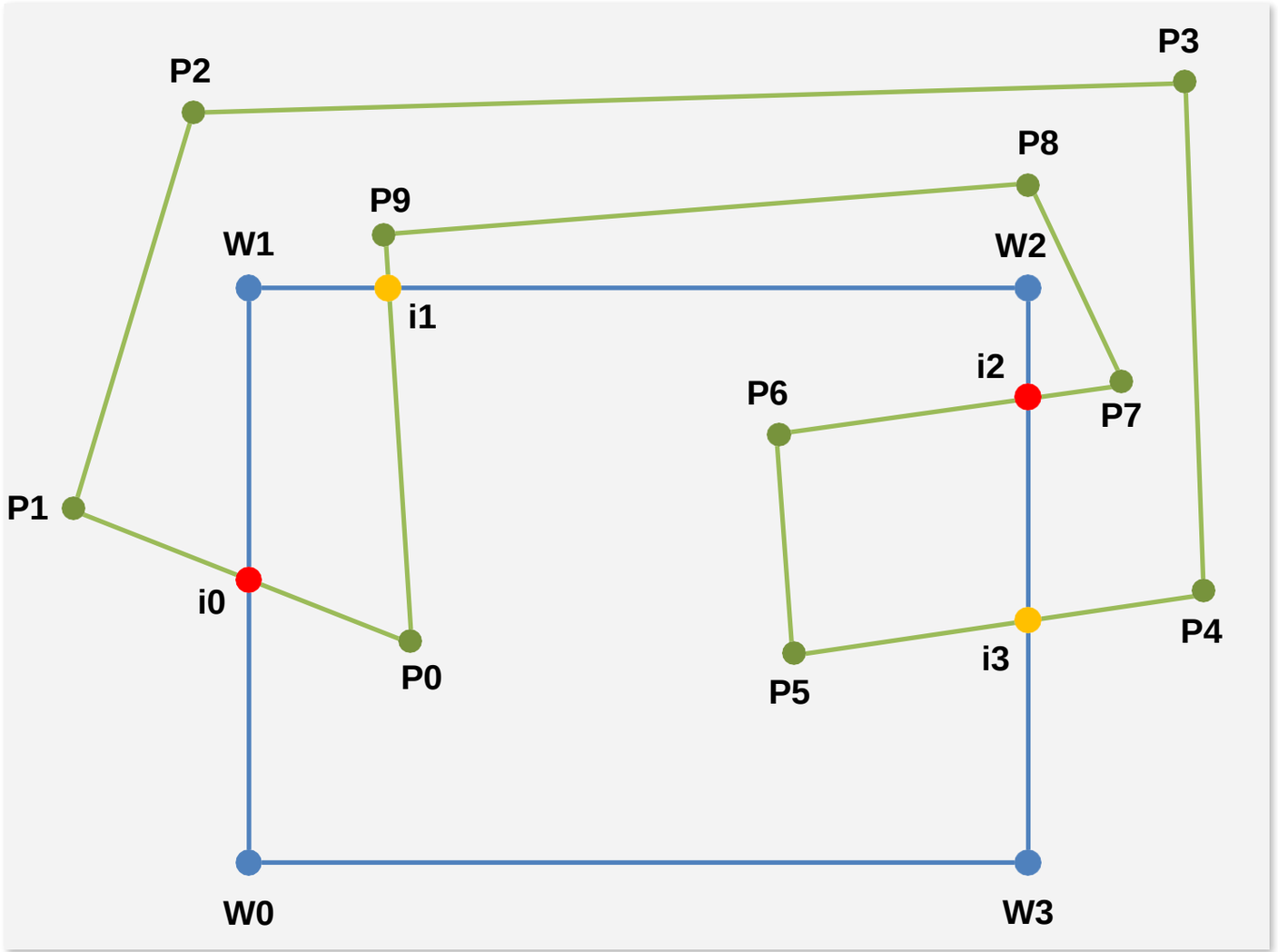
Weiler-Atherton algorithm

- Input vertices:
  - polygon vertices
  - clipping region vertices
  - intersection vertices

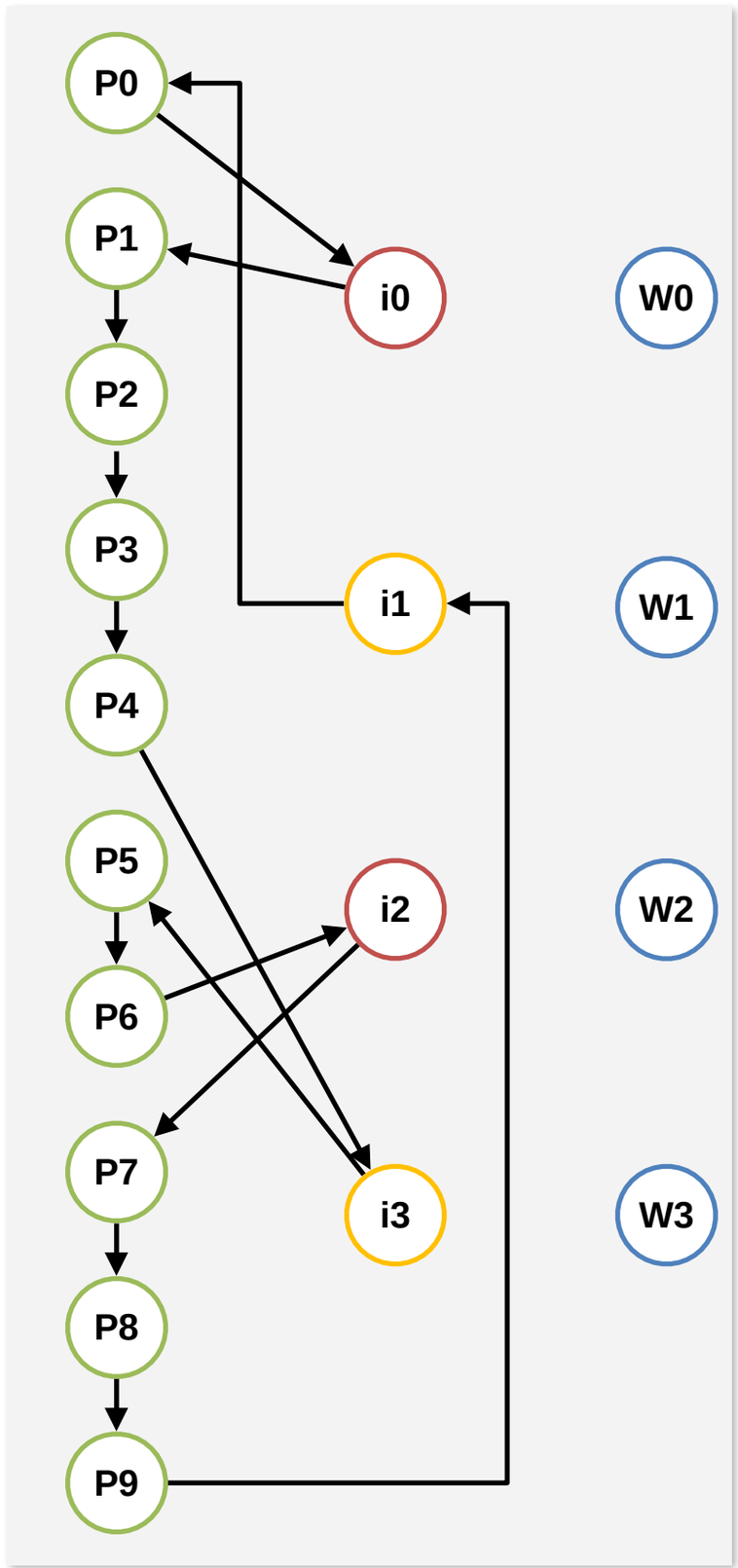
- Walk along the boundary of the polygon and include also the intersection vertices (distinguish the outgoing intersections from the incoming intersections)
- Walk along the boundary of the clipping region and include also the intersection vertices

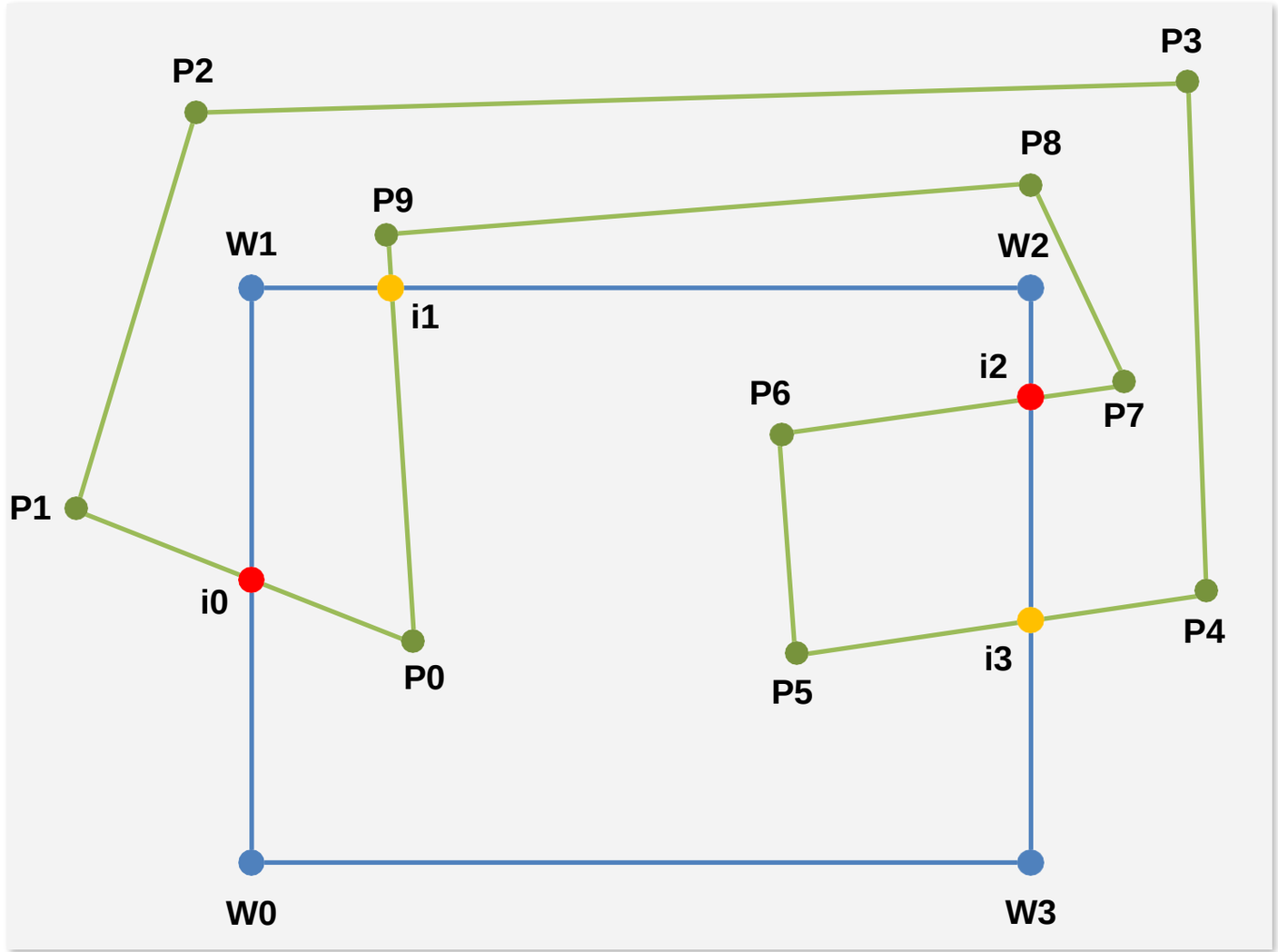
Weiler-Atherton algorithm



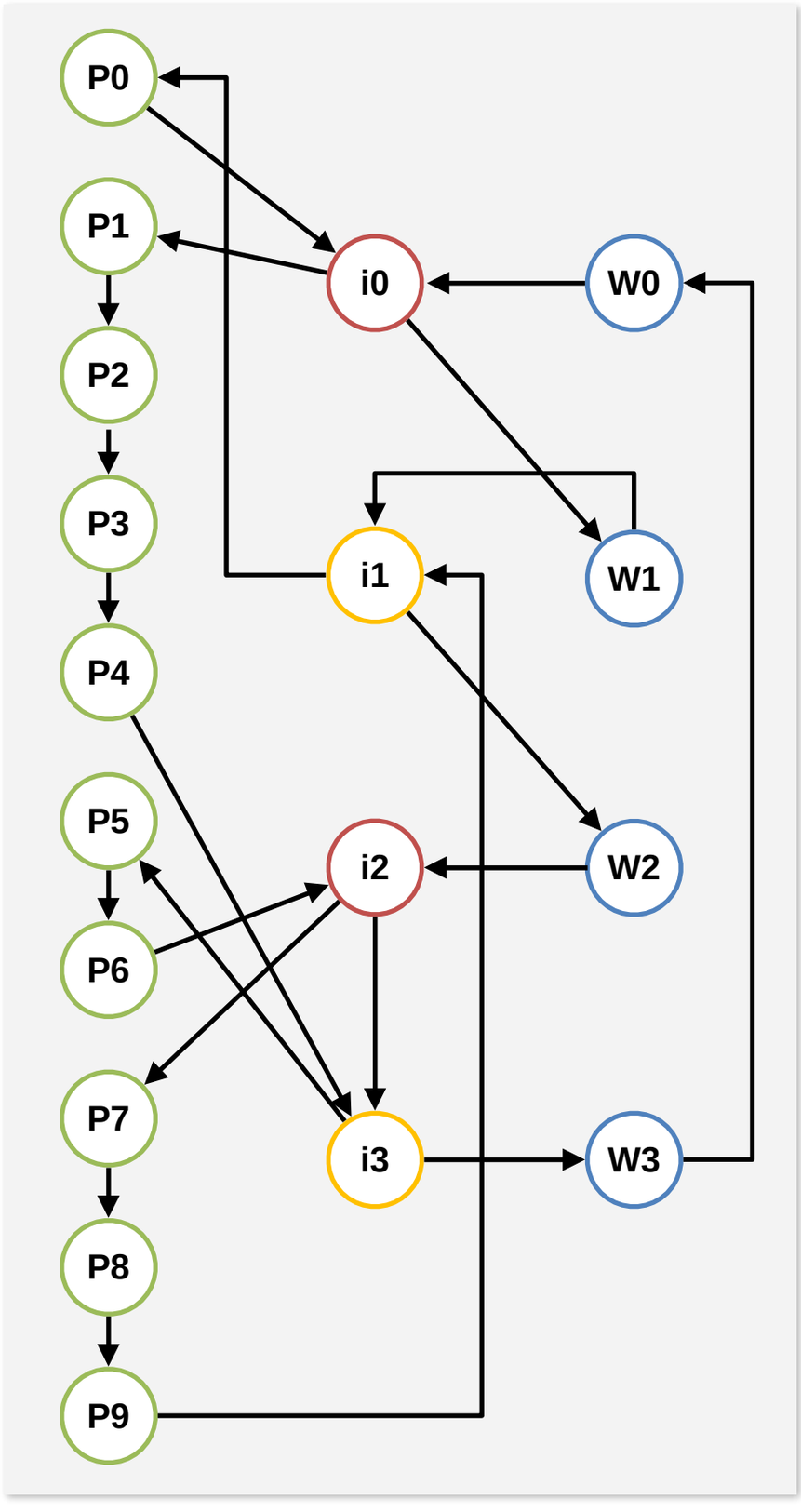


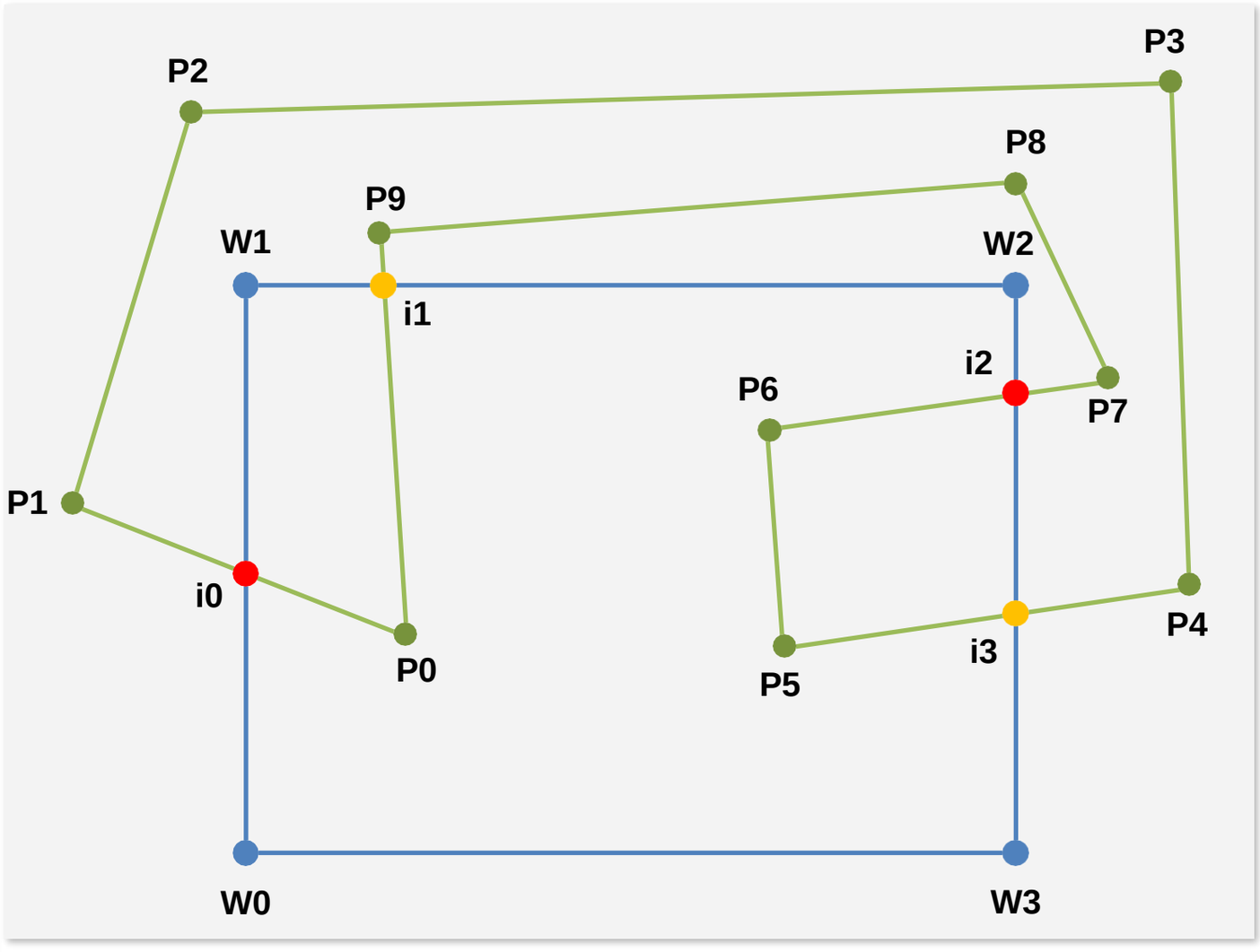
Weiler-Atherton algorithm





Weiler-Atherton algorithm

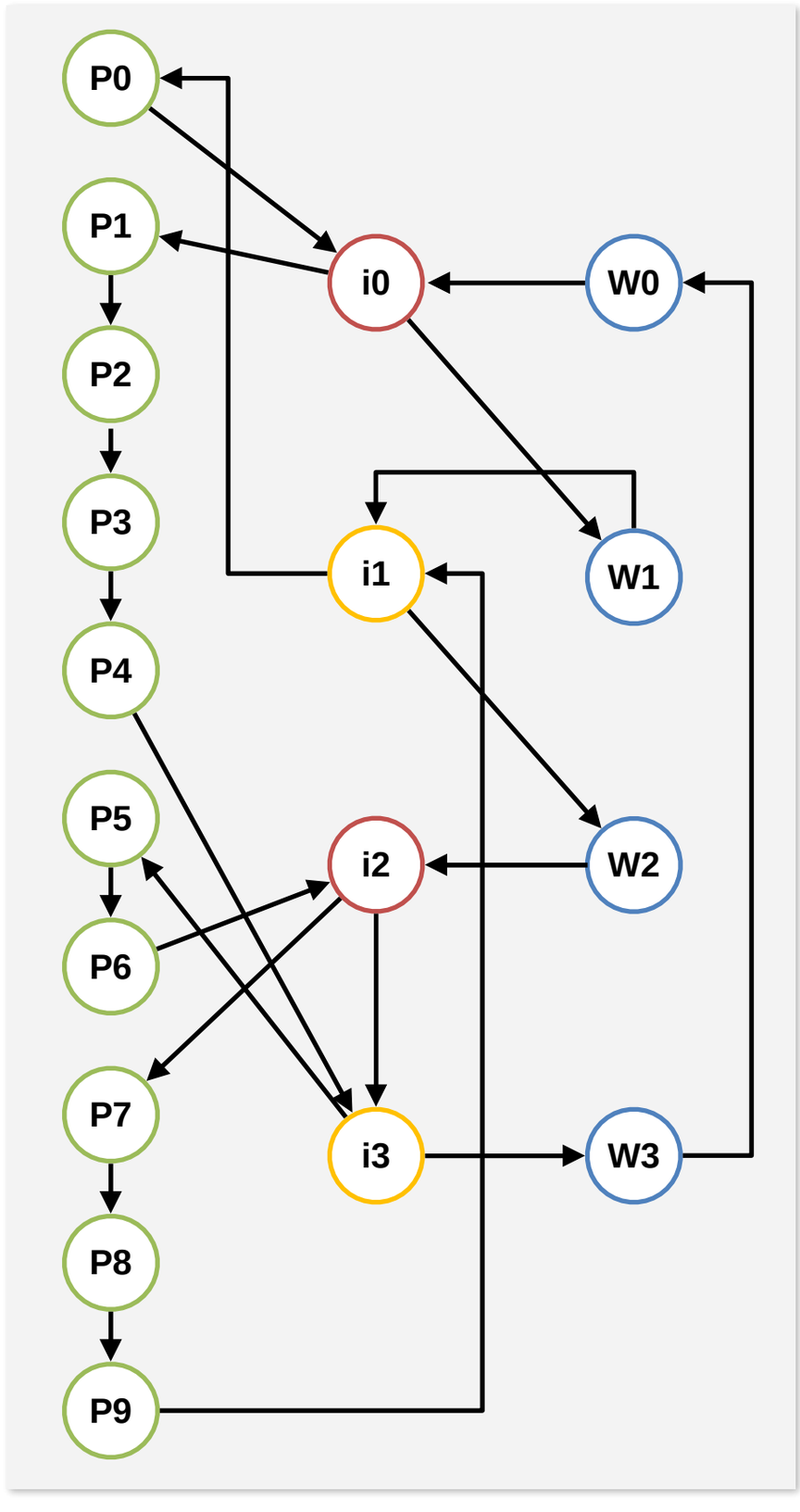


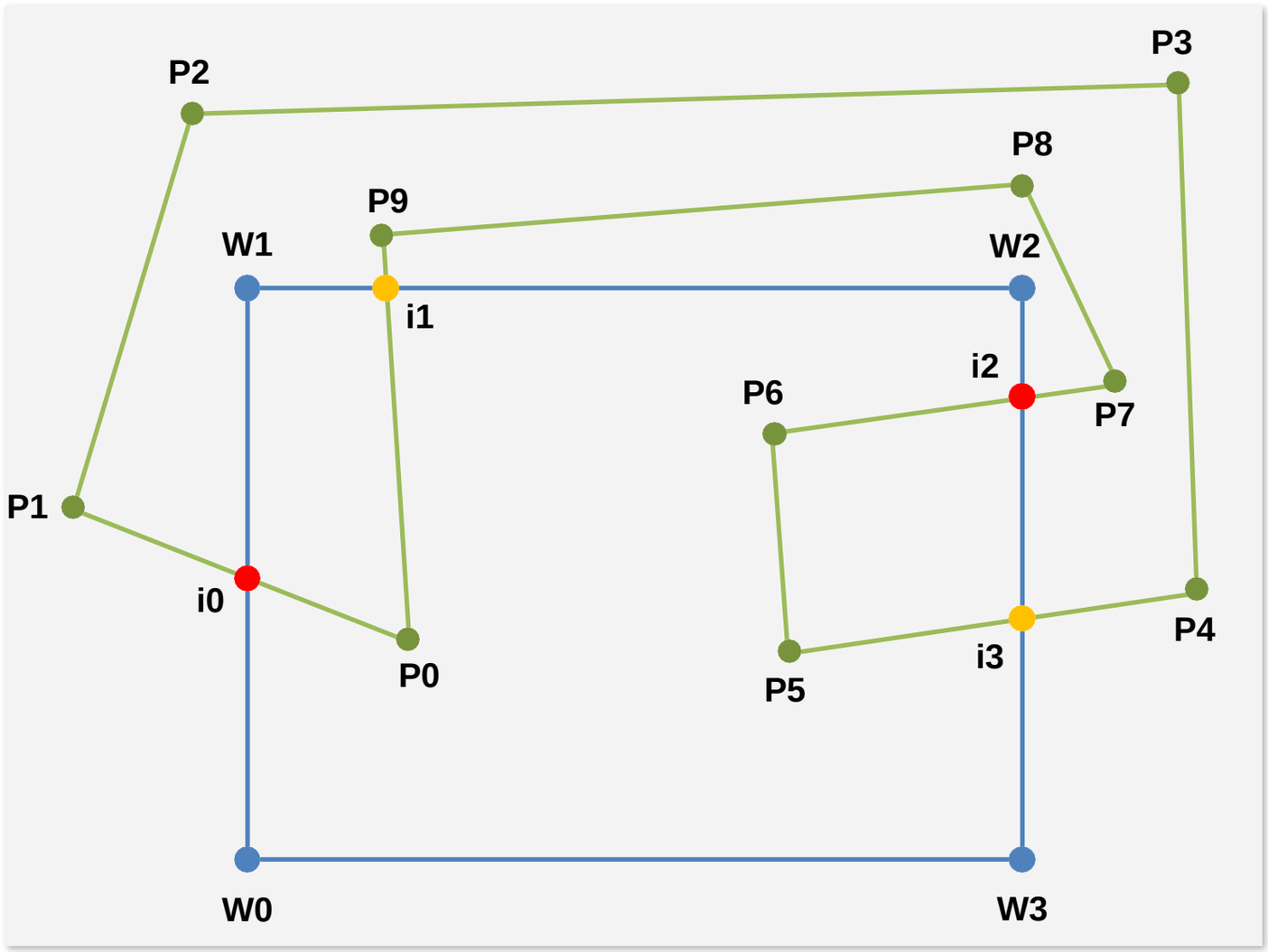


Weiler-Atherton algorithm

1. Start from an outgoing intersection vertex and add all the vertices from the boundary of the clipping region
2. After adding an incoming intersection vertex add all the vertices from the boundary of the polygon
3. Continue this process until we reach the starting vertex
4. Start with the next unvisited outgoing intersection vertex and continue with steps 1-3 until no unvisited outgoing intersection vertices are left

Weiler-Atherton algorithm





i0->W1->i1->P0->i0

i2->i3->P5->P6->i2