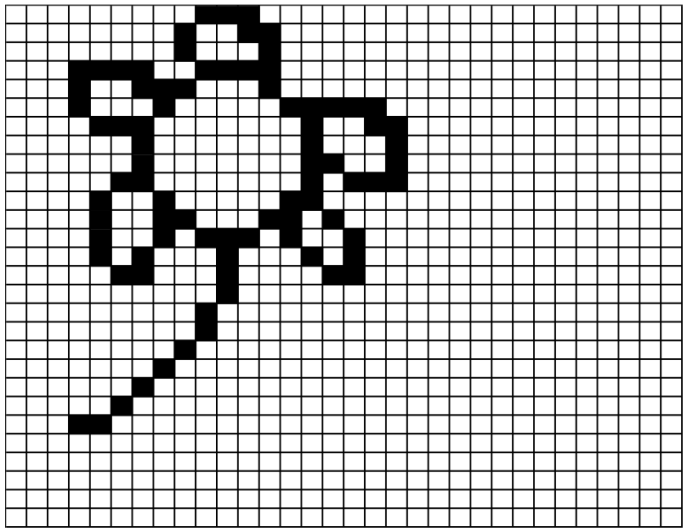
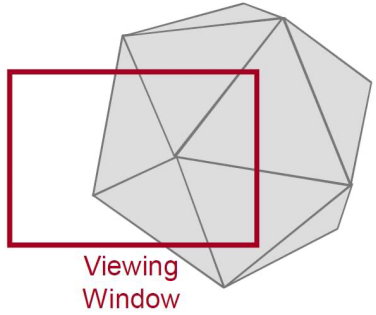
FACULTY OF ENGINEERING AND ARCHITECTURE

E016712: Computer Graphics

Rasterization and Clipping





Lecturers: Aleksandra Pizurica and Danilo Babin

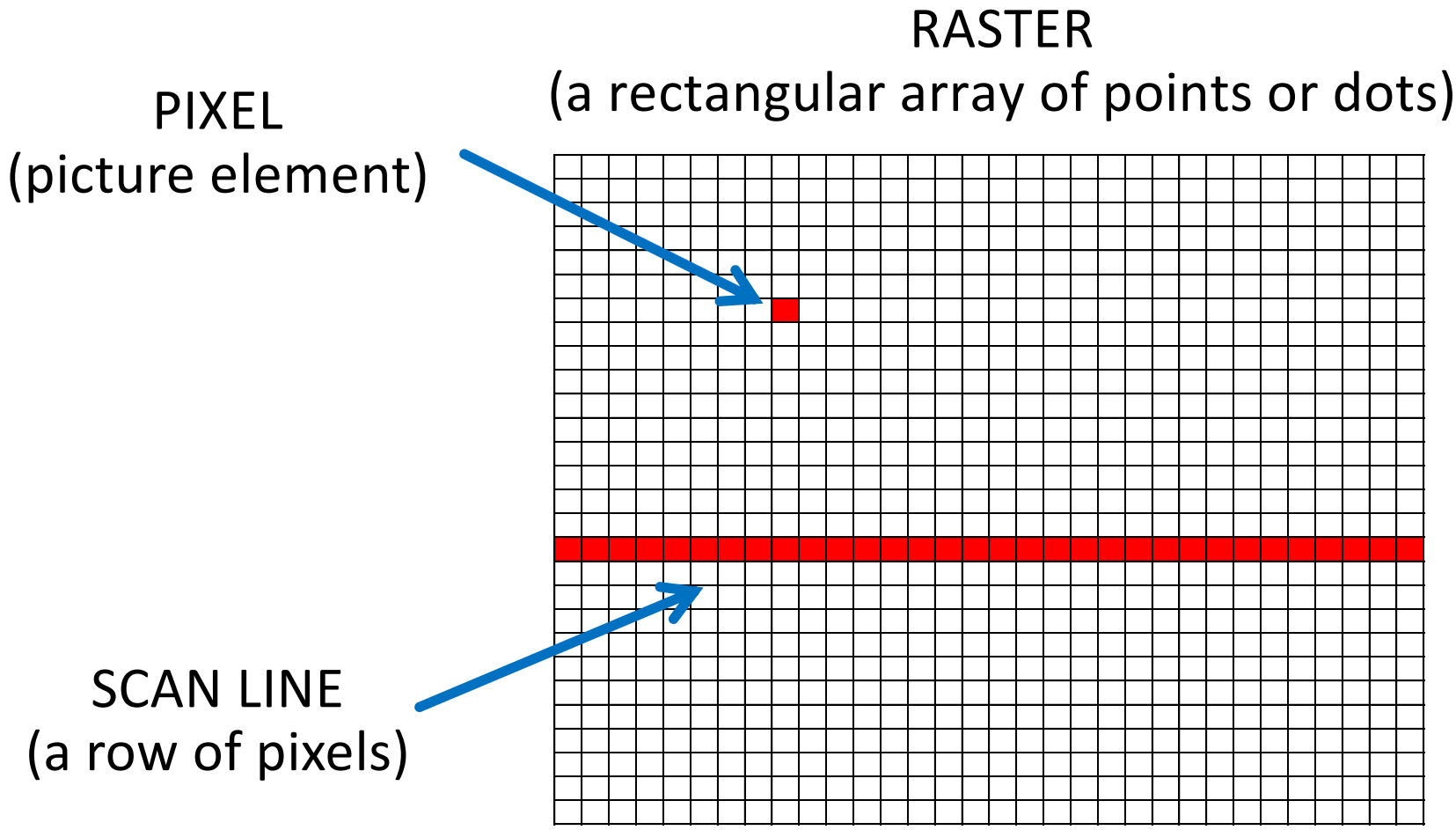
GHENT UNIVERSITY

Overview

- Scan conversion
- Polygon filling
- Clipping in 2D

Scan Conversion

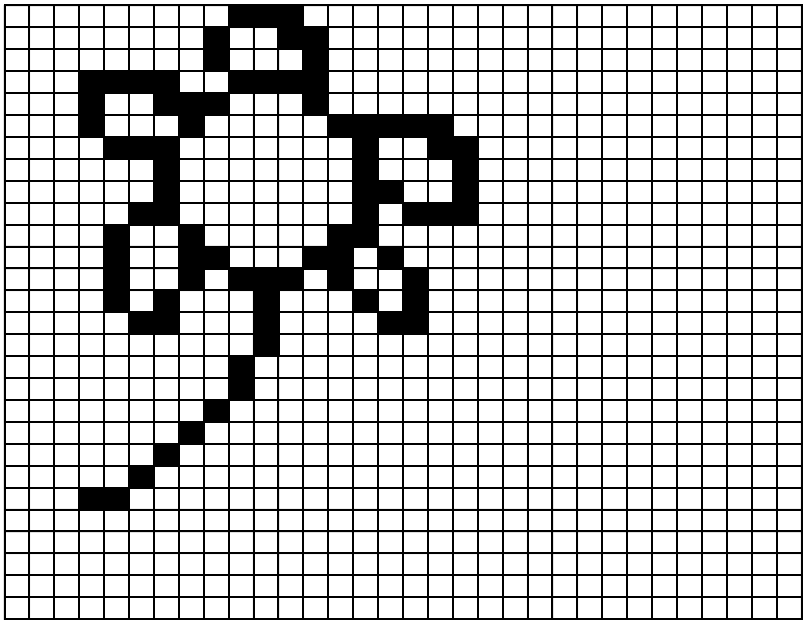
Raster Display



- Asset: Control of every picture element (rich patterns)
- Problem: limited resolution

Plotting in a raster display

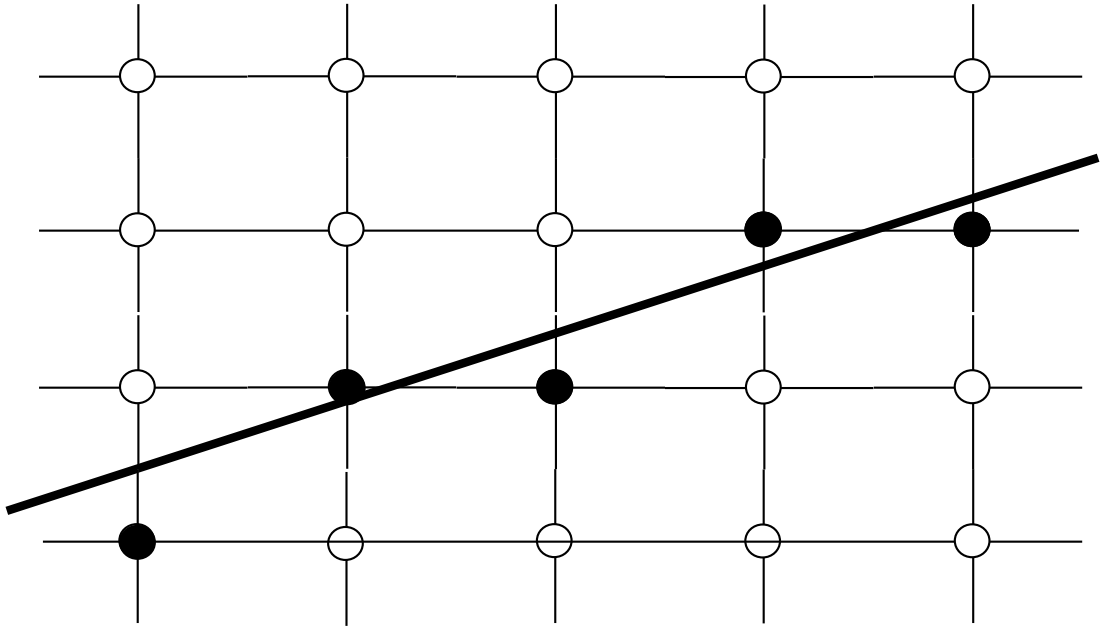
- Assume a bilevel display: each pixel is black or white
- Different patterns of dots are created on the screen by setting each pixel to black or white (i.e. turning it on or off)



- All the edges except perfectly horizontal and vertical ones show 'jaggies', i.e., staircasing effect

Line drawing

Line drawing: problem



- Line drawing on a raster grid involves approximation
- The process is called rasterization or scan-conversion

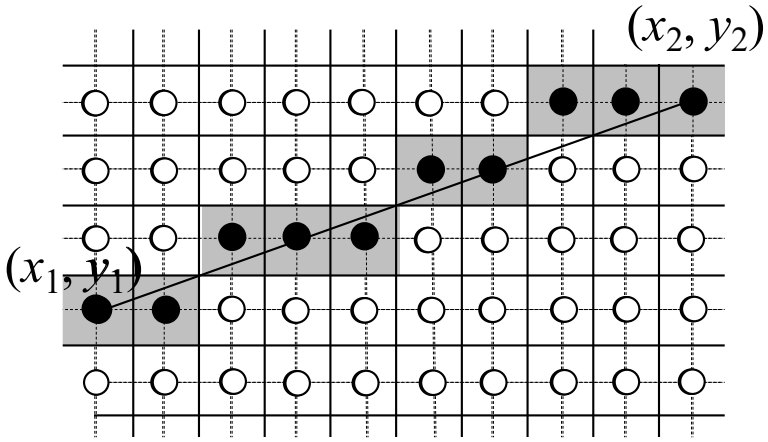
Line drawing: objectives

A line segment is defined by its end points (x_1, y_1) and (x_2, y_2) with integer coordinates.

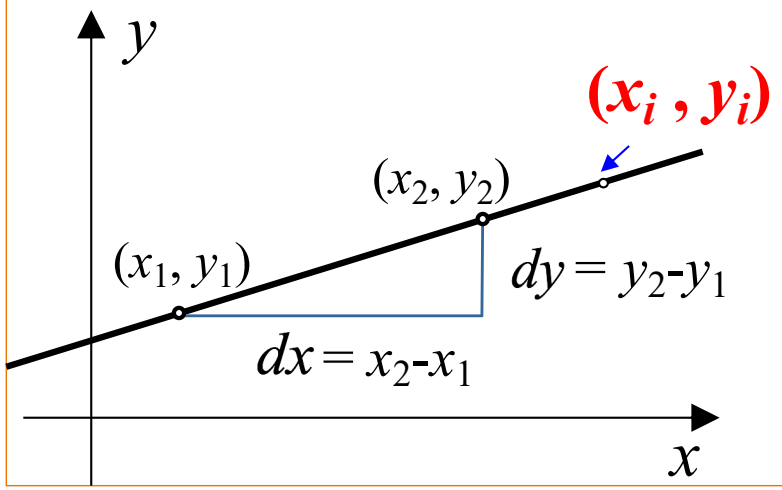
What is the best way to draw a line from (x_1, y_1) to (x_2, y_2)?

We require that this scan-converted line

- passes through endpoints
- appears straight
- appears smooth
- is independent of endpoint order
- has uniform brightness
- has slope-independent brightness
- is efficiently computed

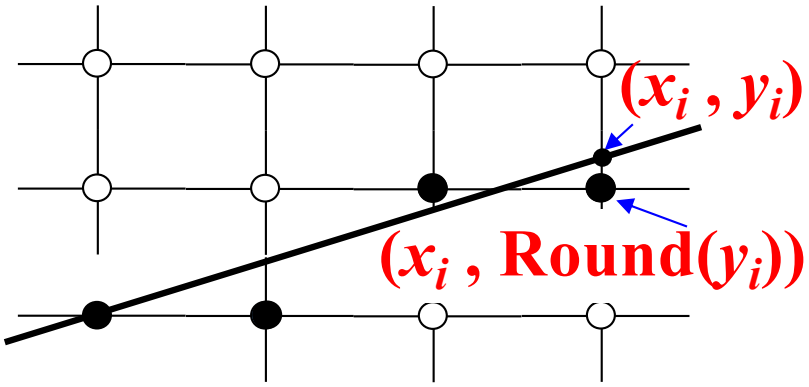


Line drawing : brute force approach



Line Characterisation

- Explicit: $y = mx + B$
- Implicit: $F(x,y) = ax + by + c = 0$
- Constant slope: $\frac{dy}{dx} = m$



The simplest strategy:

1) Compute m; $(0 \le m \le 1)$
2) Increment x by 1 starting with the leftmost point;
3) Calculate $y_i = mx_i + B$;
4) Intensify the pixel at $(x_i , \text{Round}(y_i))$, where $\text{Round}(y_i) = \text{Floor}(0.5+y_i)$

What is wrong with this approach?

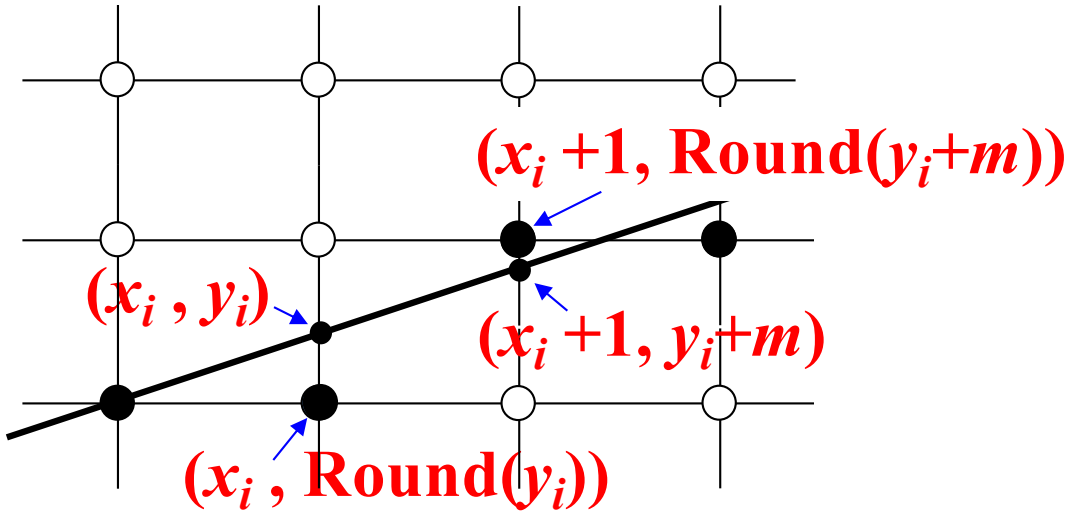
Line drawing: an incremental approach

The previous approach is inefficient because each iteration requires floating point operation *Multiply*, *Addition*, and *Floor*.

We can eliminate the multiplication by observing that:

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x$$

If $\Delta x = 1$, then $y_{i+1} = y_i + m$. Thus, a unit of change in x changes y by m, which is the slope of the line.



Midpoint Line Algorithm: intro

What is wrong with the incremental algorithm?

- Required floating-point operations (Round)
- The time-consuming floating-point operations are unnecessary because both endpoints are integers

Bresenham's classical algorithm (1965), also called midpoint line algorithm, is attractive because it uses only integer arithmetic

Main idea: decide the next step on the grid: choose among the possible successor pixels using only integer quantities

Midpoint Line Algorithm: idea

Previously selected pixel: $P\,(x_p\,,y_p)$

Assume the line slope is m, $0 \le m \le 1$

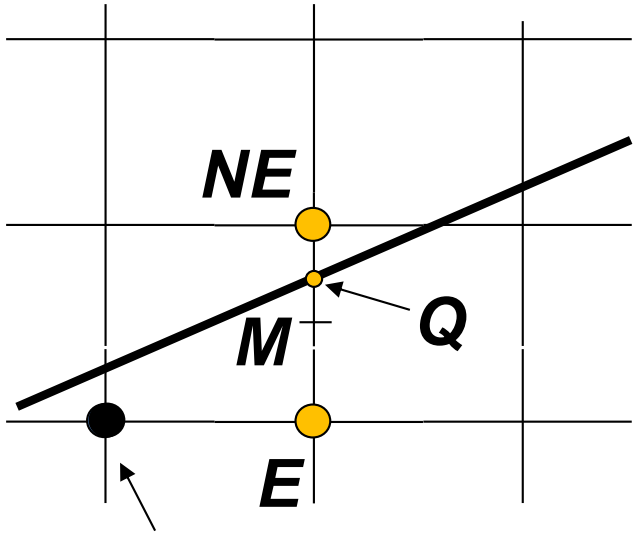
Choose between one increment to the right (the east pixel, *E*) or one increment to the right and one increment up (the northeast pixel, *NE*)

Q: the intersection point of the line being scan-converted with the grid line $x = x_p + 1$

M: the midpoint between *E* and *NE*

If *M* is below the line: choose *NE*;

if *M* is above the line: choose *E*



The problem becomes: Decide on which side of the line the midpoint lies

Midpoint Line Algorithm: point positioning

Explicit line form $y = \frac{dy}{dx}x + B$ gives:

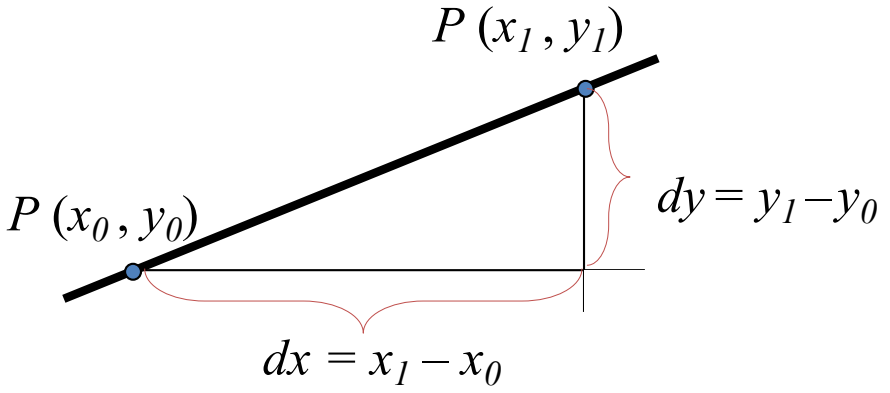
$$F(x,y) = x \cdot dy - y \cdot dx + Bdx = 0$$

Comparing with the implicit form $F(x,y) = ax + by + c = 0$ yields

$$a = dy,\ b = -dx,\ \text{and}\ c = Bdx$$

It can be shown that

$$F(x,y) \begin{cases} > 0 \text{ if } (x,y) \text{ is below the line} \\ = 0 \text{ if } (x,y) \text{ is on the line} \\ < 0 \text{ if } (x,y) \text{ is above the line} \end{cases}$$

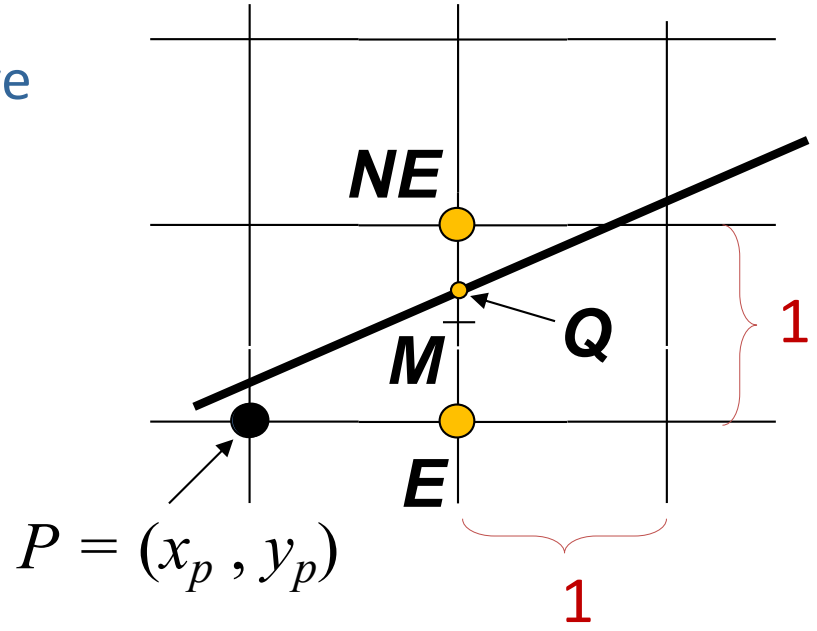


Midpoint Line Algorithm: decision variable (1)

To decide whether *M* lies below or above the line, we need only to compute

$$F(M) = F(x_p + 1, y_p + \frac{1}{2})$$

and to test its sign.

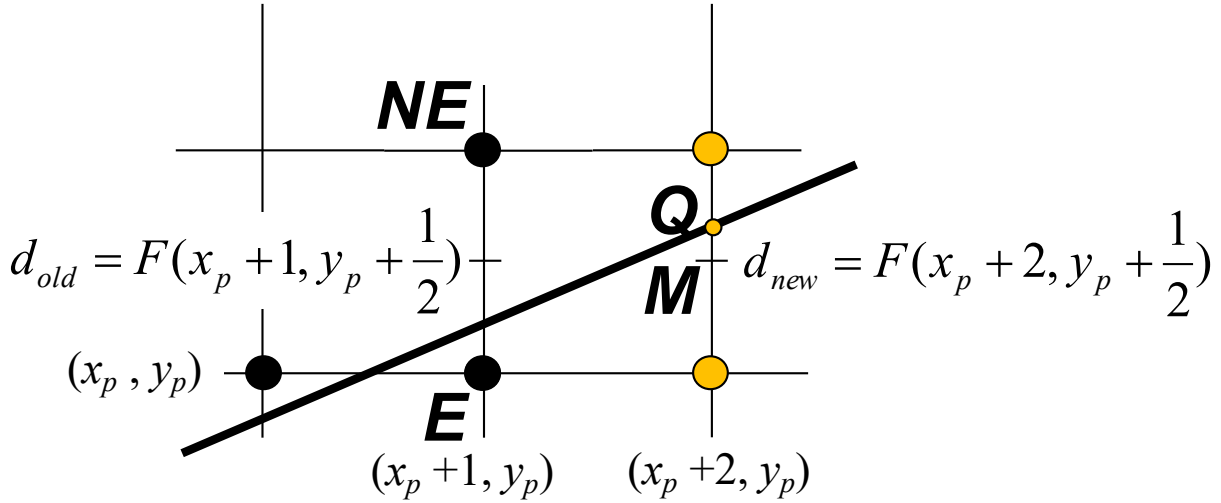


Define a decision variable d as:

$$d = F(x_p + 1, y_p + \frac{1}{2}) = a(x_p + 1) + b(y_p + \frac{1}{2}) + c$$

$$\begin{cases} > 0 & \text{, choose pixel } NE \\ = 0 & \text{, choose pixel } E \\ < 0 & \text{, choose pixel } E \end{cases}$$

Midpoint Line Algorithm: decision variable (2)



If *E* is chosen, *M* is incremented by one step in the x direction. Then

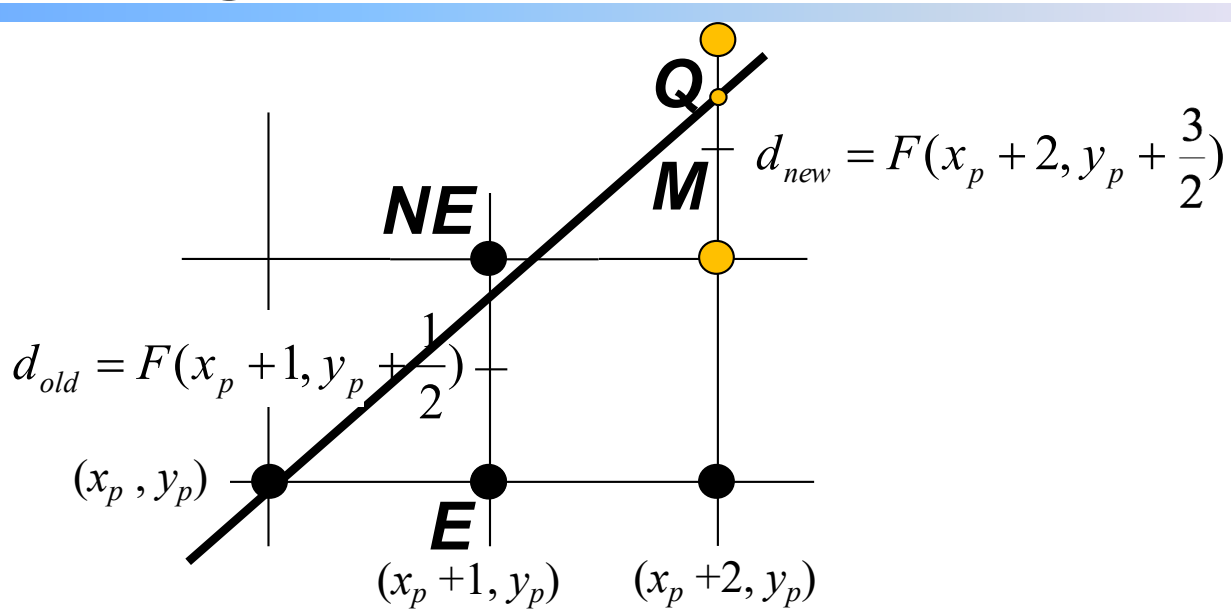
$$d_{new} = F(x_p + 2, y_p + \frac{1}{2}) = a(x_p + 2) + b(y_p + \frac{1}{2}) + c$$

$$= a(x_p + 1) + a + b(y_p + \frac{1}{2}) + c = F(x_p + 1, y_p + \frac{1}{2}) + a$$

$$= d_{old} + a = d_{old} + dy$$

$$(d_{new} - d_{old})|_E = \Delta_E = a = dy$$

Midpoint Line Algorithm: decision variable (3)



On the other hand, if *NE* is chosen

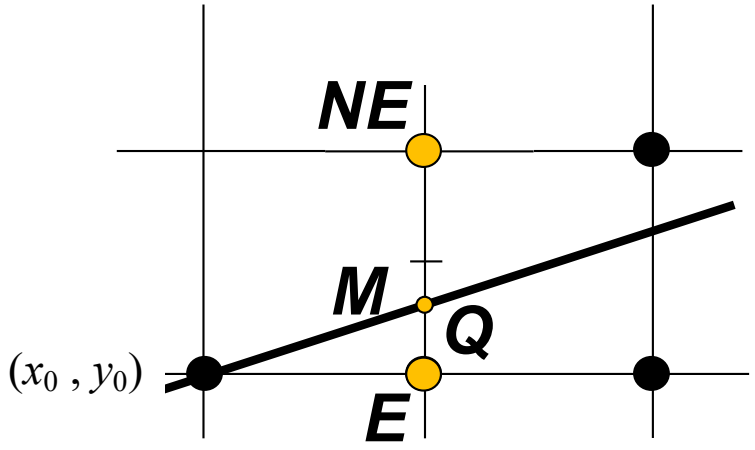
$$d_{new} = F(x_p + 2, y_p + \frac{3}{2}) = a(x_p + 2) + b(y_p + \frac{3}{2}) + c$$

$$= a(x_p + 1) + a + b(y_p + \frac{1}{2}) + b + c = F(x_p + 1, y_p + \frac{1}{2}) + a + b$$

$$= d_{old} + a + b = d_{old} + dy - dx$$

$$(d_{new} - d_{old})\big|_{NE} = \Delta_{NE} = a + b = dy - dx$$

Midpoint Line Algorithm: decision variable (4)



Initial condition:

$$F(x_0+1, y_0+\frac{1}{2}) = a(x_0+1)+b(y_0+\frac{1}{2})+c = F(x_0,y_0)+a+b/2$$

$$d_{start} = a+b/2 = dy-dx/2$$

Midpoint Line Algorithm: summary (1)

- Start from the chosen endpoint, find $d_{start} = a + b/2$ and based on its sign select the second pixel (*E* or *NE*)

- At each next step
  - Update d by adding either Δ_E or Δ_{NE} to the old value, depending on the choice of the previous pixel
  - choose the successor pixel as *E* or *NE* based on the sign of d

Implementation note:

To eliminate the fraction in d_{start}, multiply F by 2: $F(x,y) = 2(ax+by+c)$

- Thus, d and the increments Δ_E and Δ_{NE} are multiplied by 2
- This does not affect the sign of the decision variable d, which is all that matters for the midpoint test!

Midpoint Line Algorithm: summary (2)

Initialisation: $d_{start} = 2a + b = 2dy - dx$

where $dy = y_1 - y_0$, $dx = x_1 - x_0$.

Incremental update: 1) if E was chosen: $\Delta_E = 2dy$

$$d_{new} = d_{old} + \Delta_E$$

2) if NE was chosen: $\Delta_{NE} = 2(dy - dx)$

$$d_{new} = d_{old} + \Delta_{NE}$$

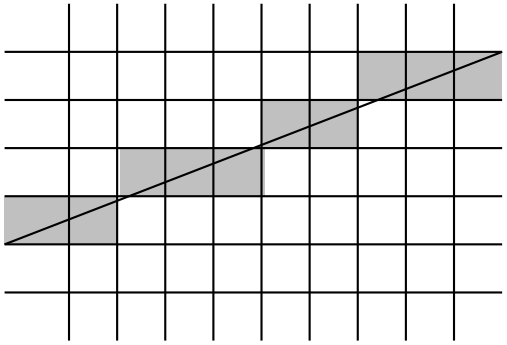
Note that evaluating d_{new} in any step requires only simple integer addition.

- No time-consuming multiplication involved
- Simple incremental updates
- Efficient algorithm

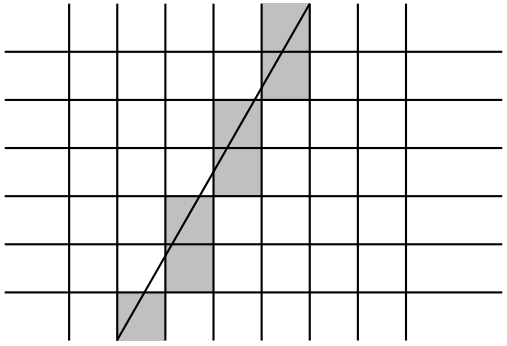
This works for those line with slope (0, 1). What about bigger slopes?

Line drawing: slope problem

- When the slope m is between 0 and 1 we can step along the x-axis.
- Other slopes can be handled by suitable reflections around the principal axes



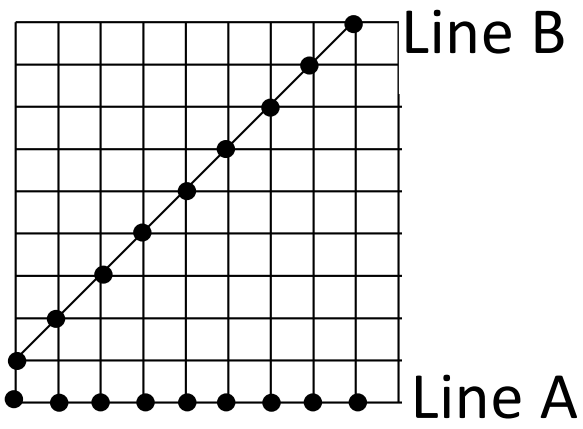
$m < 1$, can step along x.



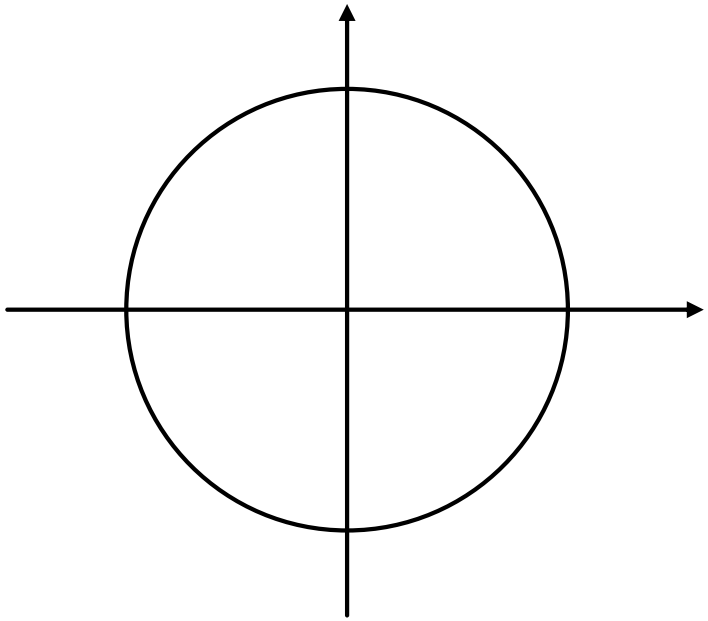
$m > 1$, cannot step along x.
(apply a suitable reflection)

Line drawing: Slope dependent intensity

- Problem: weaker intensity of diagonal lines
- Consider two scan-converted lines in the figure. The diagonal line, B, has a slope of 1 and hence is $\sqrt{2}$ times longer than the horizontal line A. Yet the same number of pixels is drawn to represent each line
- If the intensity of each pixel is I, then the intensity per unit length of line *A* is I, whereas for line *B* it is only $I / \sqrt{2}$



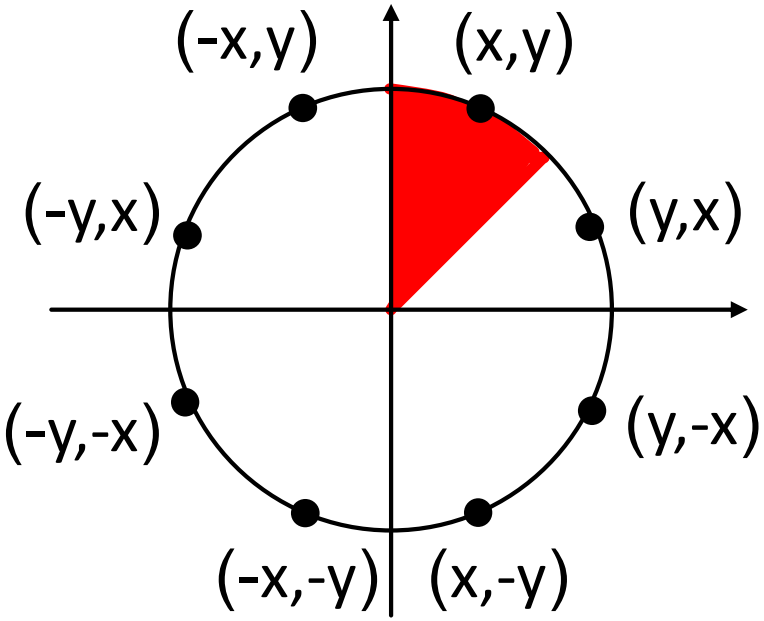
Scan converting circles



Suppose we want to rasterize a circle. Think of a smart algorithm that makes use of the circle symmetry to avoid unnecessary computations.

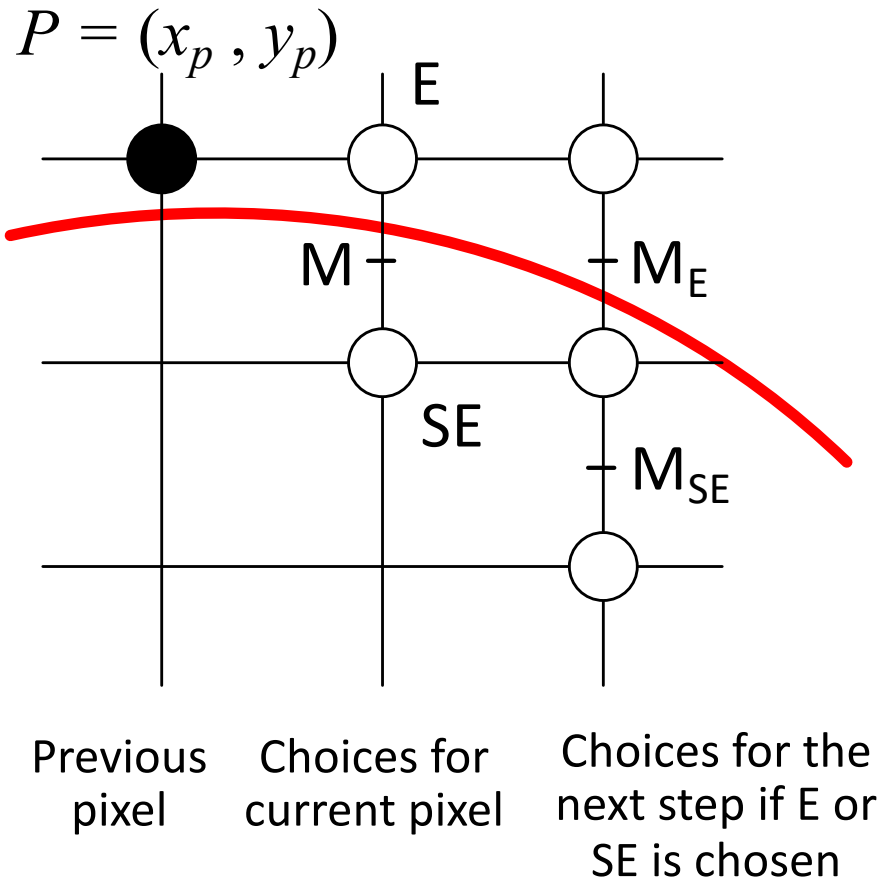
Which part of the circle do we need to scan convert, so that the rest follows by symmetry?

Scan converting circles



Eight way symmetry: If the point (x,y) is on the circle, then we can trivially compute seven other points on this circle

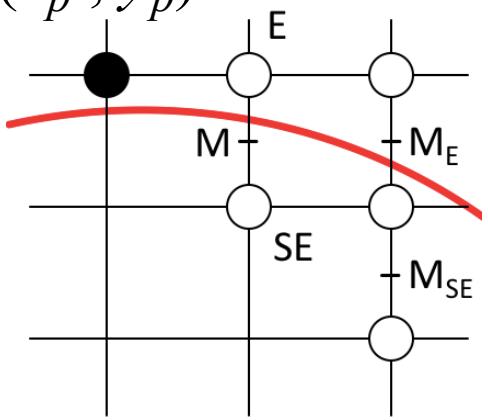
The midpoint circle scan conversion



The midpoint circle scan conversion

$$F(x,y) = x^2 + y^2 - R^2 = 0$$

$$P = (x_p , y_p)$$



The decision variable d is

$$d_{old} = F(x_p + 1, y_p - \frac{1}{2}) = (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2$$

If $d_{old} < 0$, *E* is chosen and the new d is:

$$d_{new} = F(x_p + 2, y_p - \frac{1}{2}) = (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2$$

resulting in $\Delta_E = 2x_p + 3.$

If $d_{old} \geq 0$, *SE* is chosen and the new decision variable is

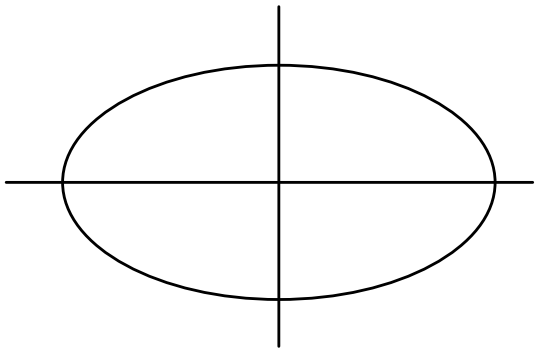
$$d_{new} = F(x_p + 2, y_p - \frac{3}{2}) = (x_p + 2)^2 + (y_p - \frac{3}{2})^2 - R^2$$

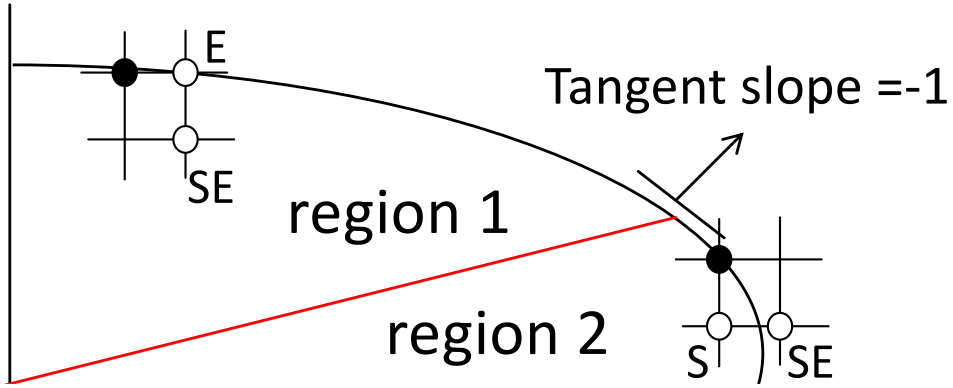
and hence $\Delta_{SE} = 2x_p + -2y_p + 5.$

Scan converting ellipses

The same reasoning can be applied for scan converting an ellipse

$$F(x,y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$$





Note: division into four quadrants
two regions in the first quadrant

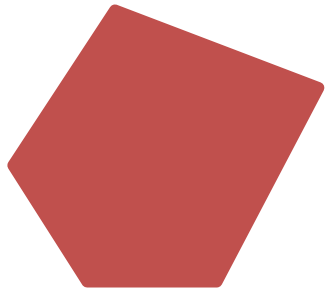
Next part: Polygon filling

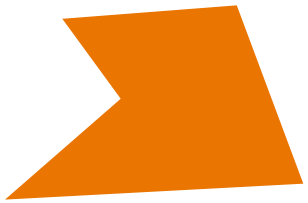
Polygons

Vertex = point in space (2D or 3D)

Polygon = ordered list of vertices

- Each vertex is connected with the next one in the list
- The last vertex is connected with the first one
- A polygon can contain holes
- A polygon can also contain self-intersections





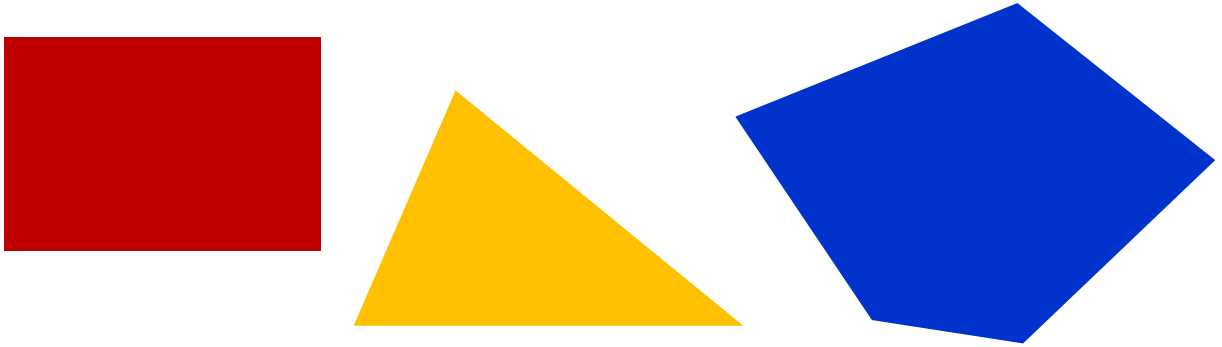
Simple polygon – no holes or self-intersections

- Such simple polygons are most interesting in Computer Graphics

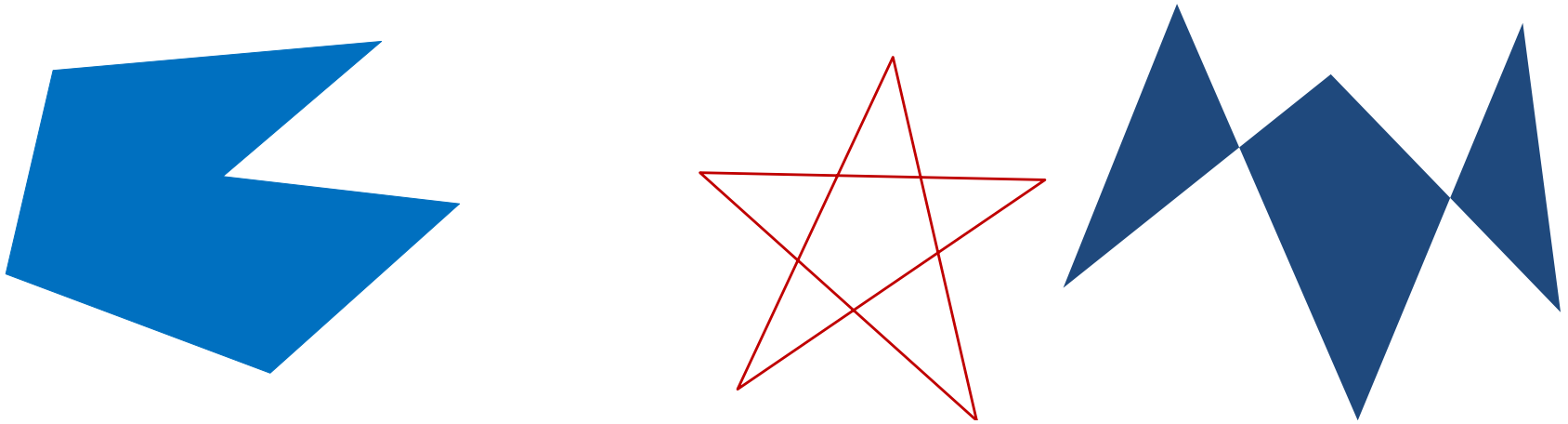
Efficient algorithms exist for polygon scan line rendering; this yields efficient algorithms for lighting, shading, texturing

By using a sufficient number of polygons, we can get close to any reasonable shape

Examples of polygons



Convex Polygons



Simple Concave

Complex (self-intersecting) polygons

Drawing modes for polygons

Draw lines along polygon edges

- Called **wireframe** mode
- Using e.g. midpoint line (Bresenham's) algorithm

Draw filled polygons

- **Shaded** polygons (shading modes)
  - **Flat shading** – constant color for whole polygon
  - **Gouraud shading** – interpolate vertex colors across the polygon
  - **Phong shading** – interpolate surface normals

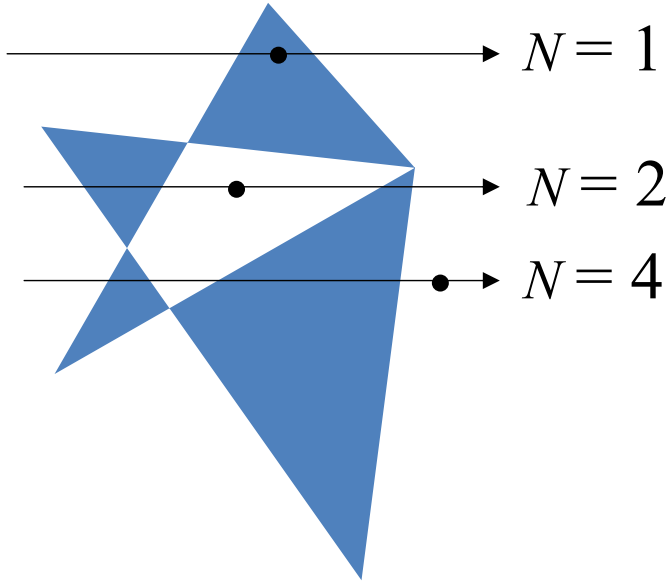
Polygon interior

We need to fill in (i.e. to color) only the pixels inside a polygon.

What is “inside” of a polygon ?

Parity (odd-even) rule is commonly used:

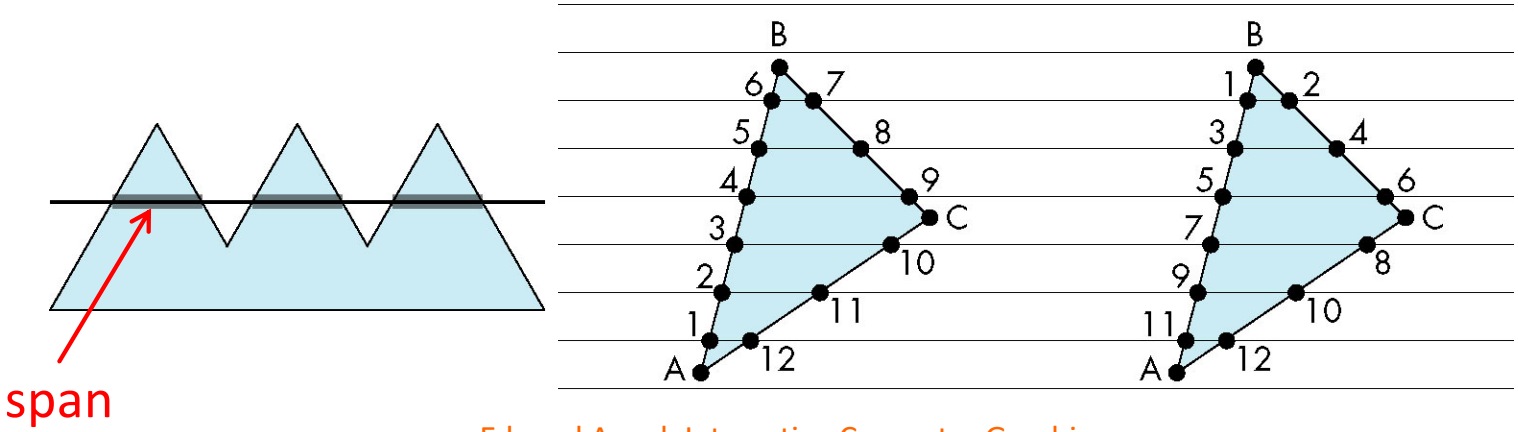
- Imagine a line passing through the point
- Count the number of intersections N with polygon edges
  - If N is odd, the point is inside
  - If N is even, the point is outside



Polygon filling: scan line approach

Span-filling is an important step in the whole polygon-filling algorithm, and is implemented by a three-step process:

- Find the intersections of the scan line with all edges of the polygon
- Sort the intersections by increasing x coordinates
- Fill in the 'inside' pixels between pairs of intersections



Edward Angel. Interactive Computer Graphics.

How do we find and sort the intersections efficiently?
How do we judge whether a pixel is inside or outside the polygon?

Polygon filling: edge coherence

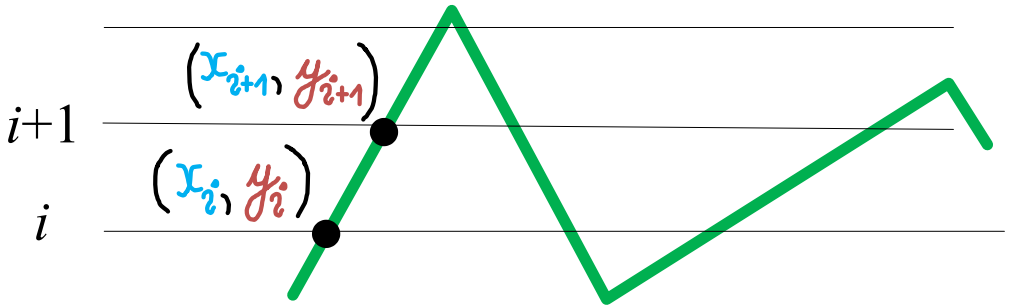
How to find all intersections between scan lines and edges?

A brute-force technique: test each polygon edge against each new scan line. This is inefficient and slow!

A better solution:

If the i-th scan line intersects with the edge at (x_i, y_i), and the edge slope is m, then the next scan line intersects with this edge at:

$$x_{i+1} = x_i + 1/m$$



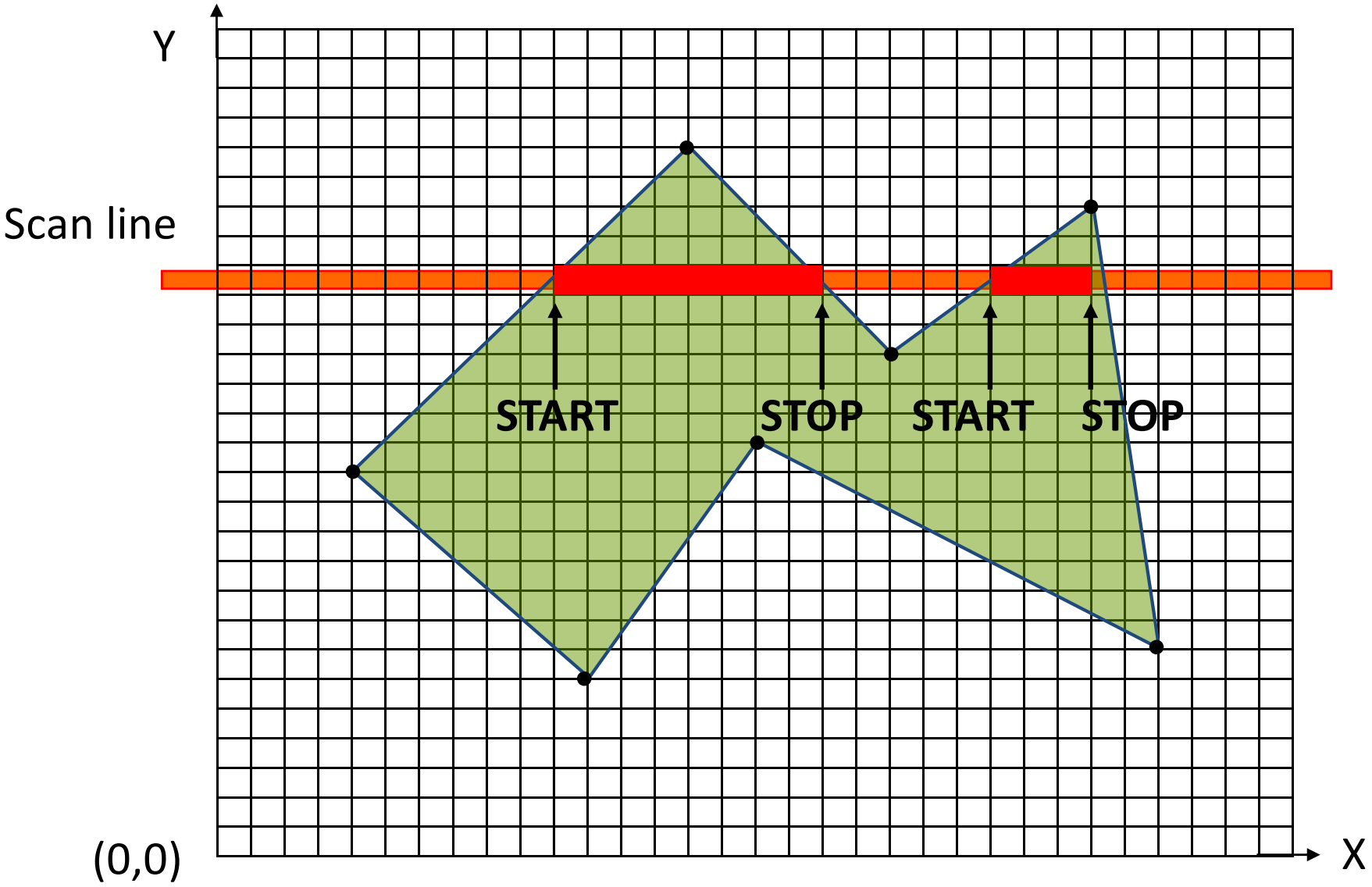
$$mx_{i+1} + B = mx_i + B + 1$$

y_{i+1} $\quad$ y_i

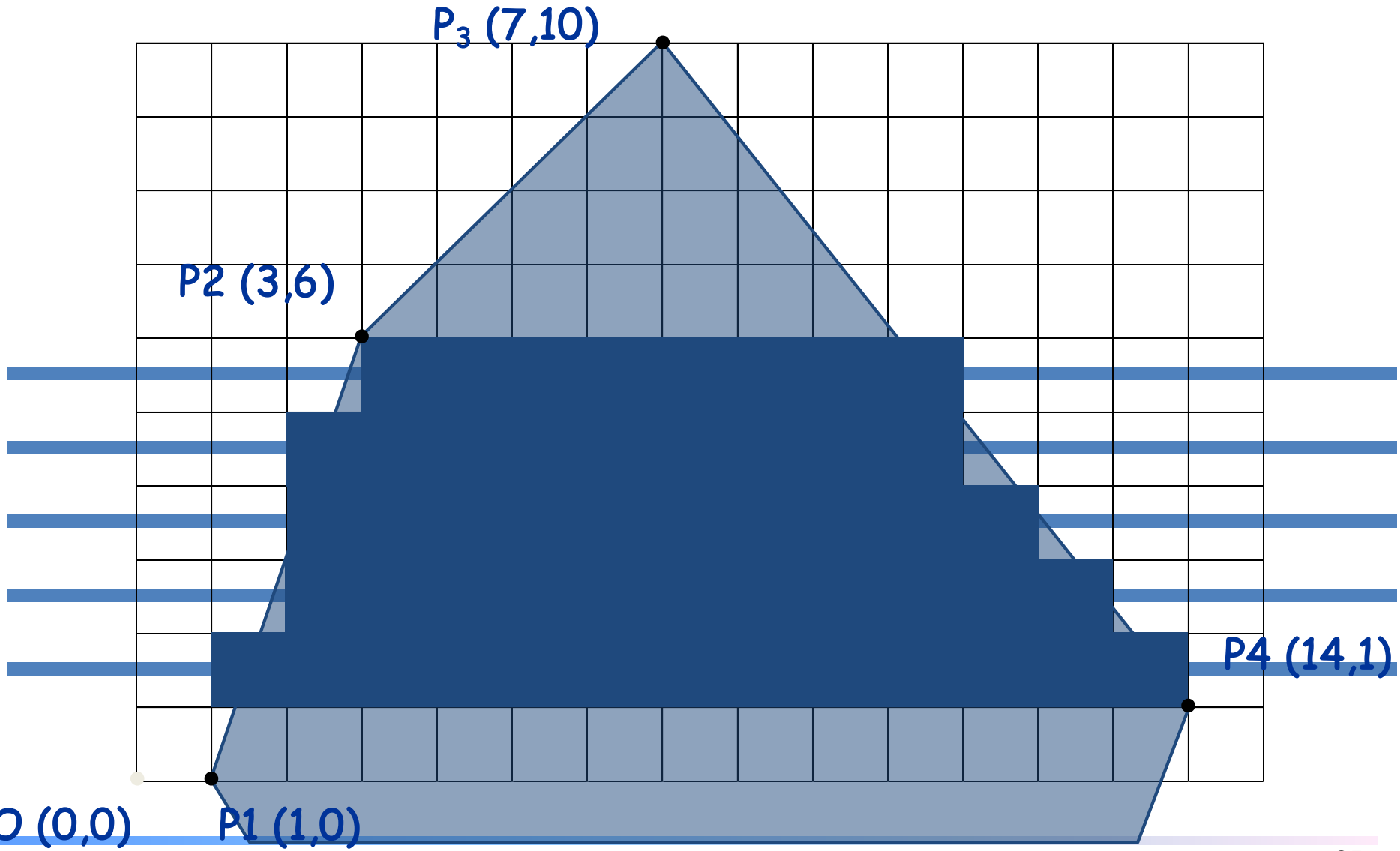
This is for the bottom-up direction. If we are filling top-down, then

$$x_{i+1} = x_i - 1/m$$

Span filling

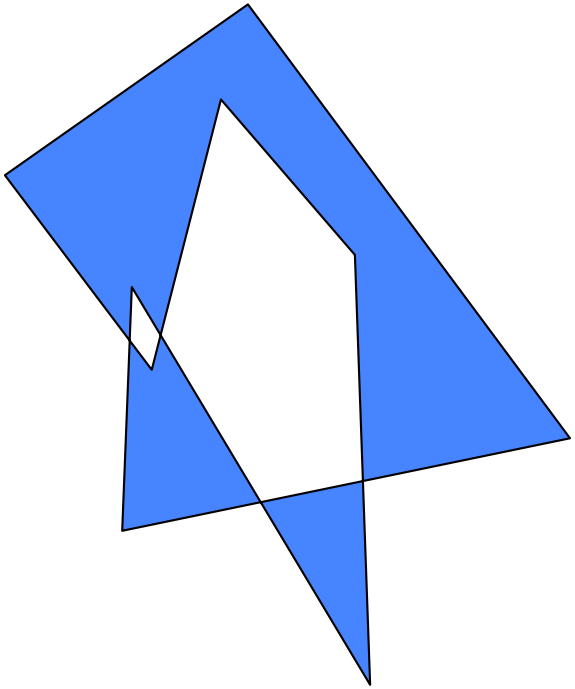


Rasterisation example

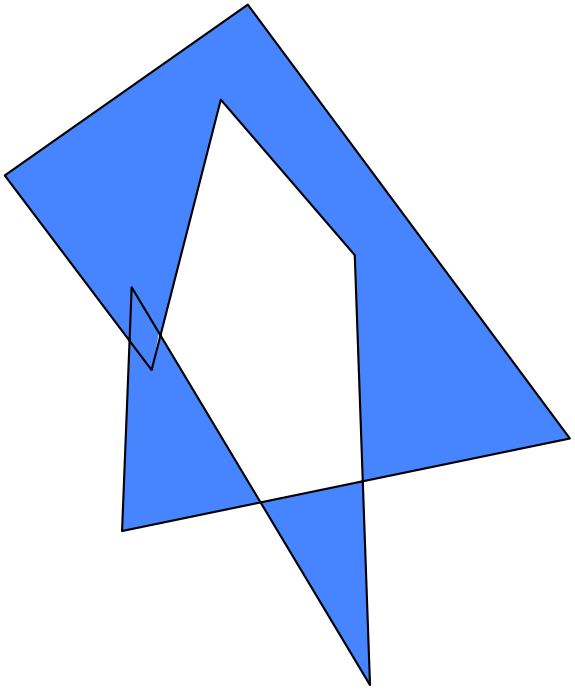


Alternative filling algorithms

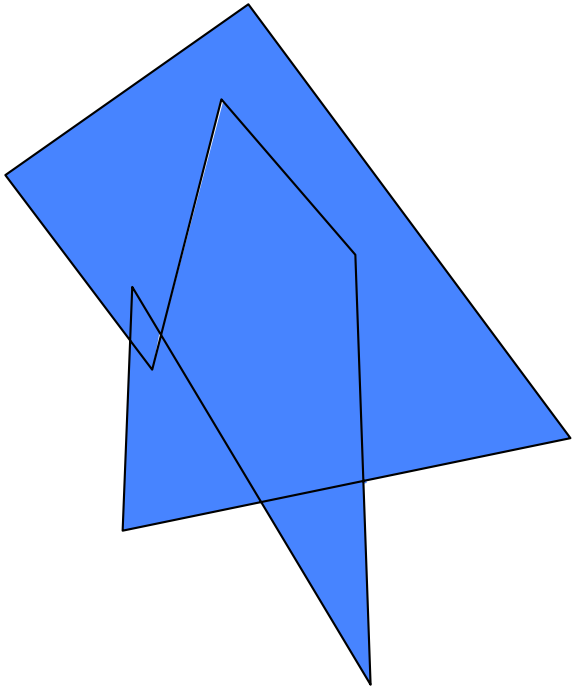
Different ways of filling a polyline



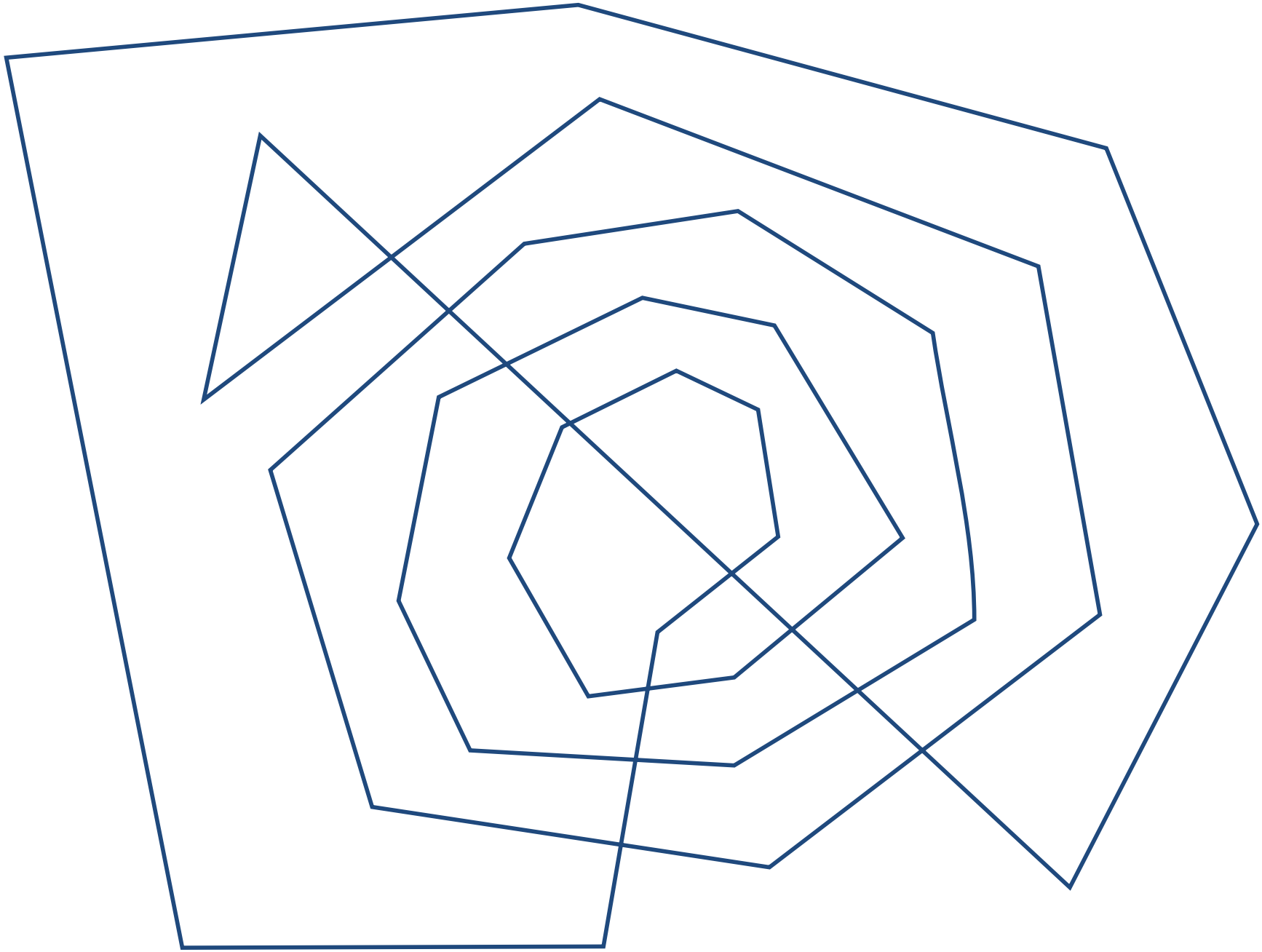
Parity rule

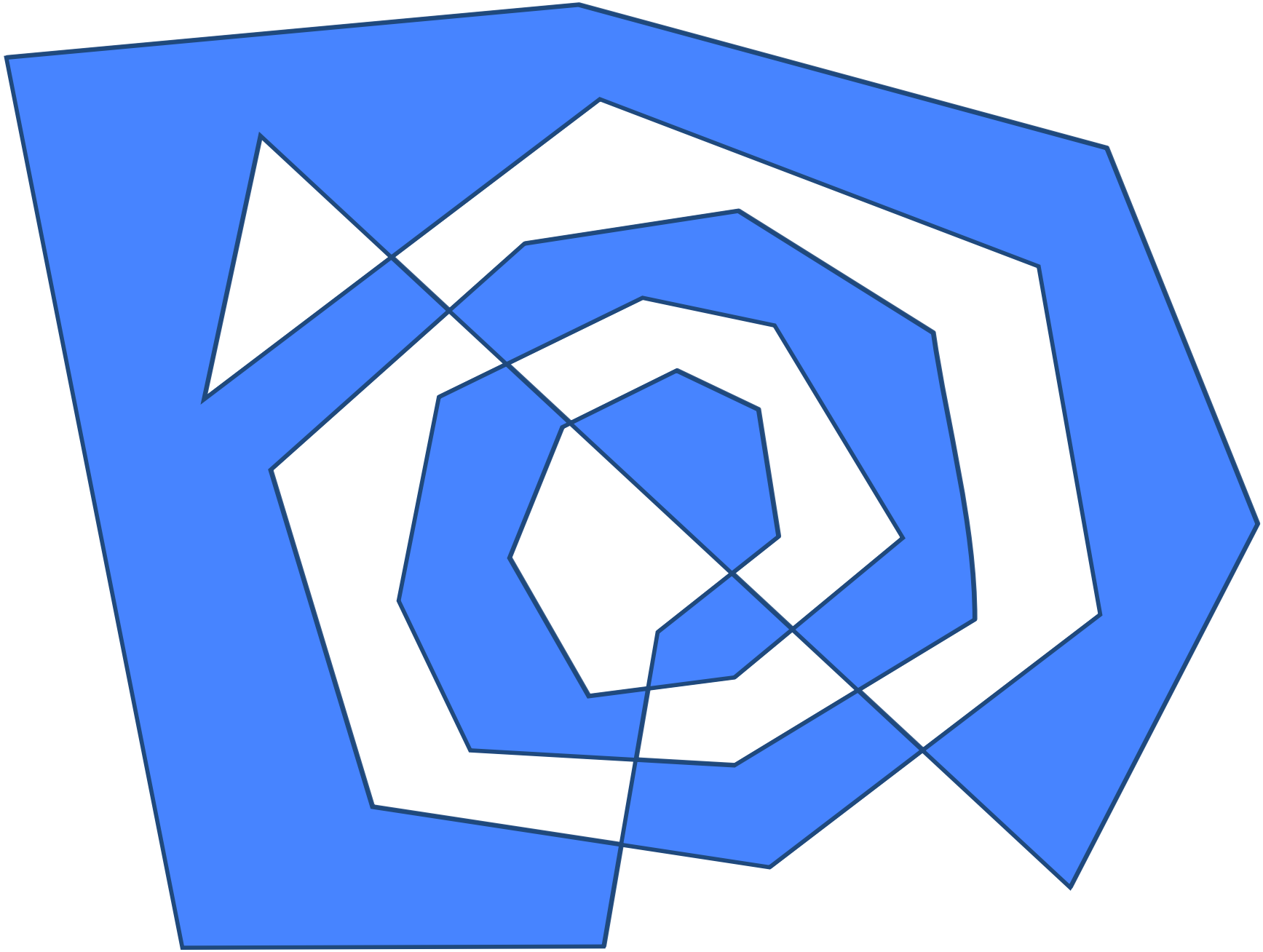


Nonzero winding rule



Nonexterior rule





Filled by parity rule

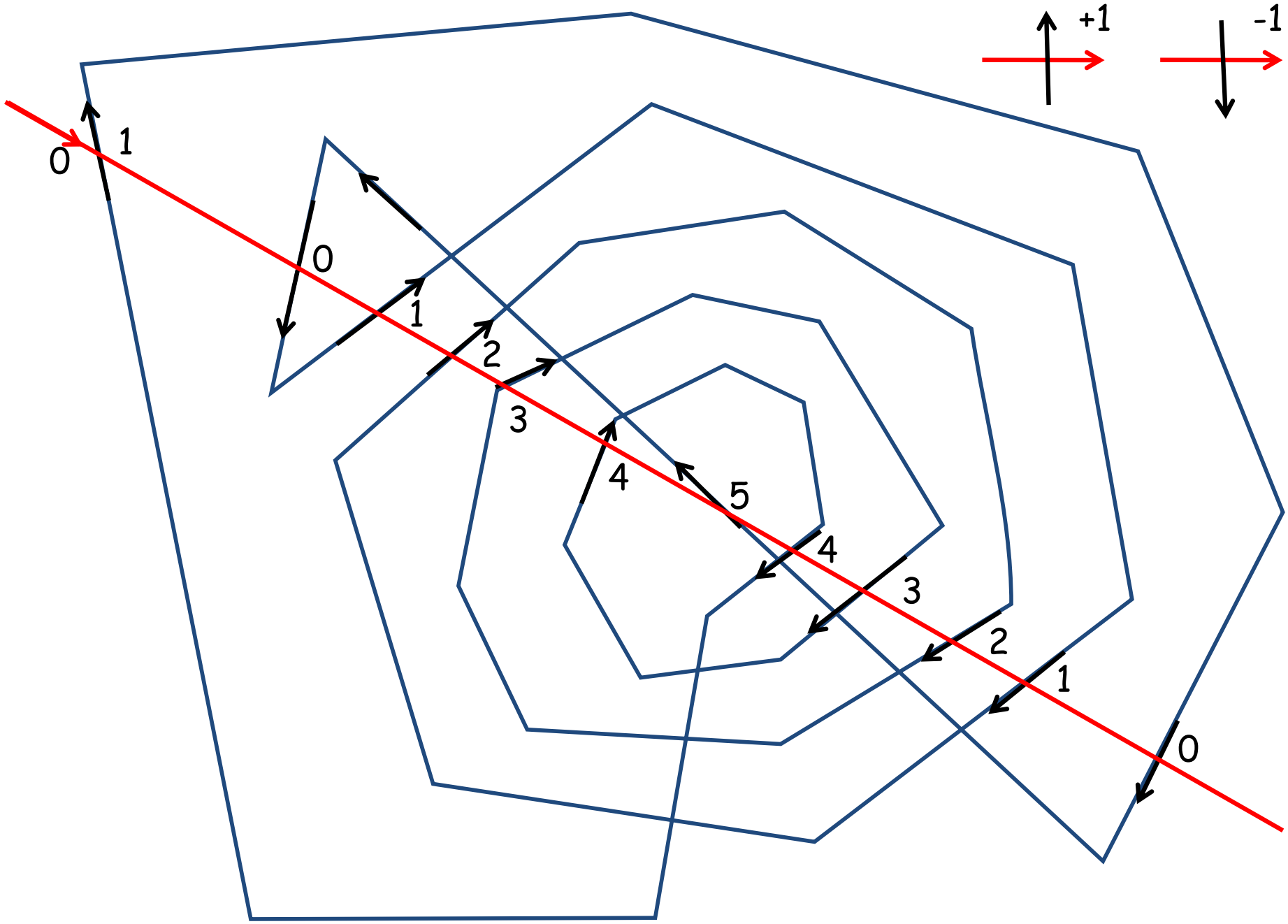
Filling: winding rules

Count the number of windings

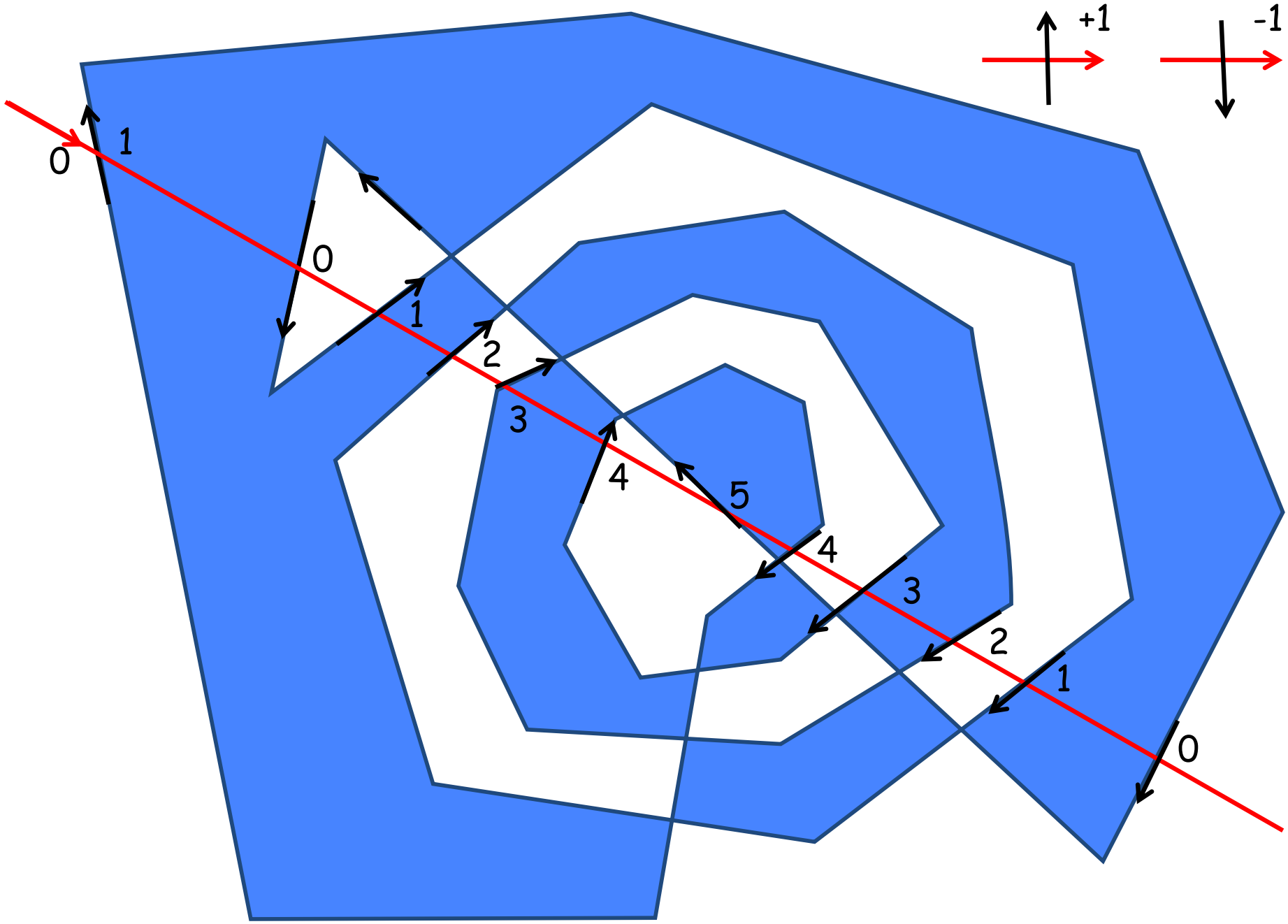
Assign a "winding index" i to each region

Possible filling rules

- Fill with one color if $i > 0$ (non-zero winding fill)
- Fill with one color if $\mathrm{mod}(i, 2) = 0$ (parity fill)
- Fill with a separate color for each value of i



Count the number of windings



Count the number of windings

Clipping

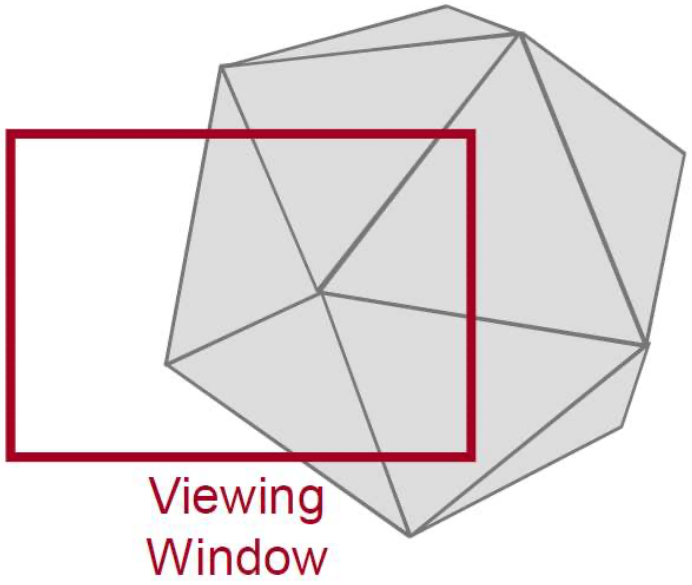
Clip a line segment at the edges of a rectangular window

Needs to be fast and robust

Robust means: works for special cases too, and preferably in the same way as for the normal cases

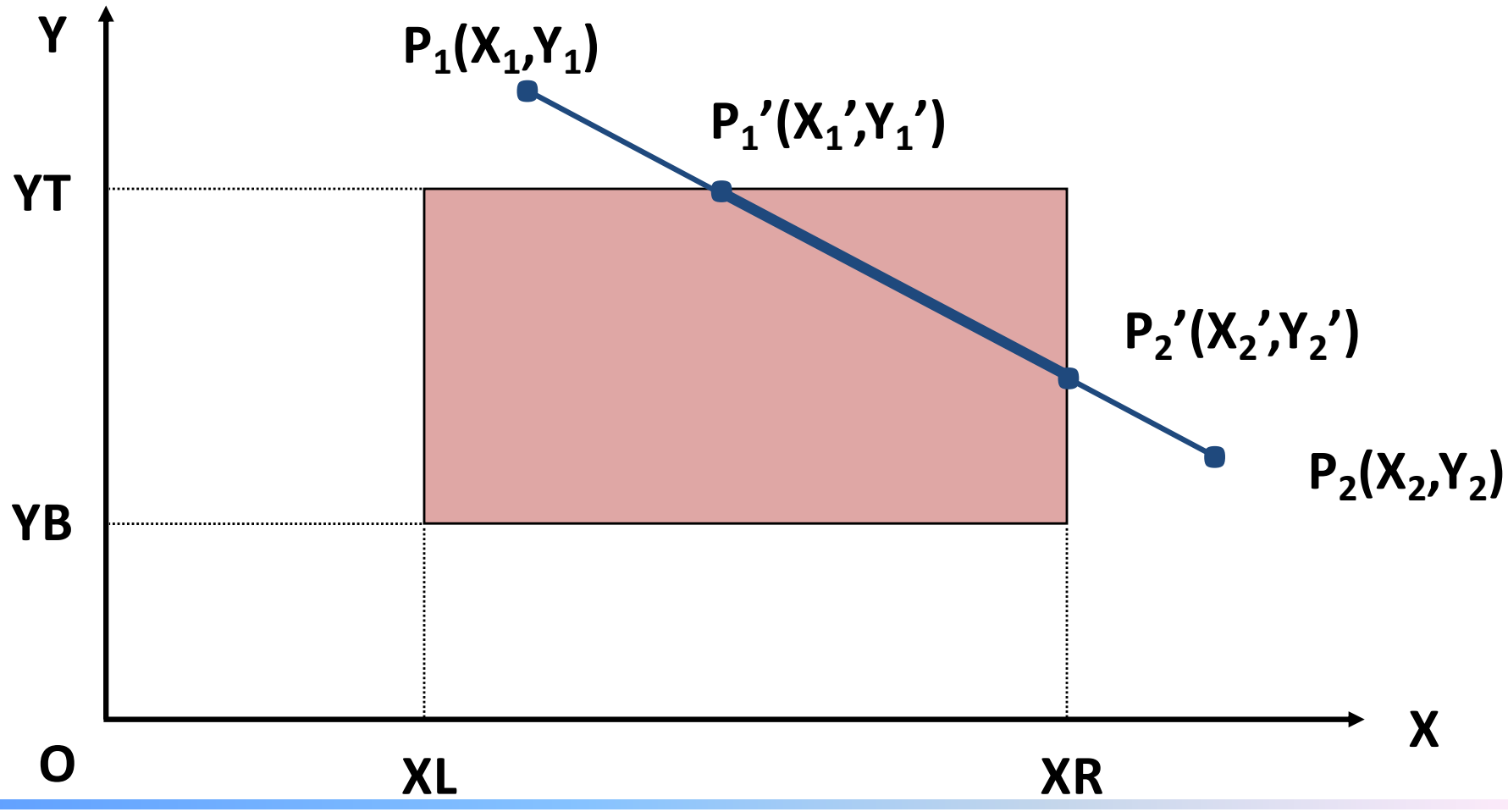
The method of Cohen-Sutherland (1974)

- Very simple
- Suitable for hardware implementation
- Can be directly extended to 3D

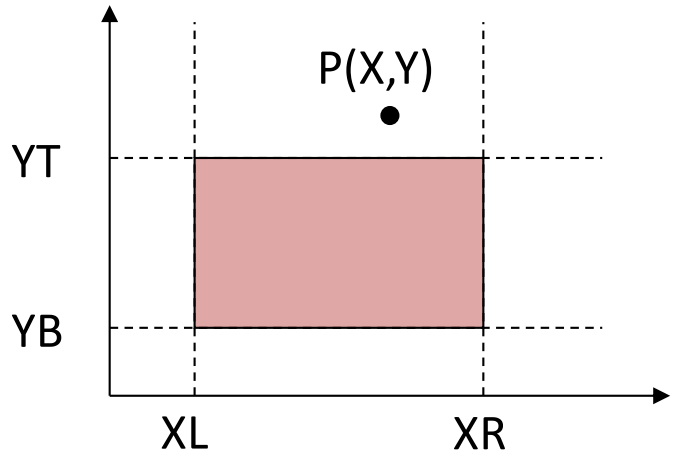


Cohen-Sutherland line clipping algorithm

Idea: Encode the position of the end points with respect to left, right, bottom and top window edges and cut accordingly.



Cohen-Sutherland line clipping algorithm



Assign to each end point a 4-bit code $\mathbf{k} = (k_1, k_2, k_3, k_4)$, $k_i \in \{0,1\}$

$k_1 = 1$ if X < XL (too much to the left);

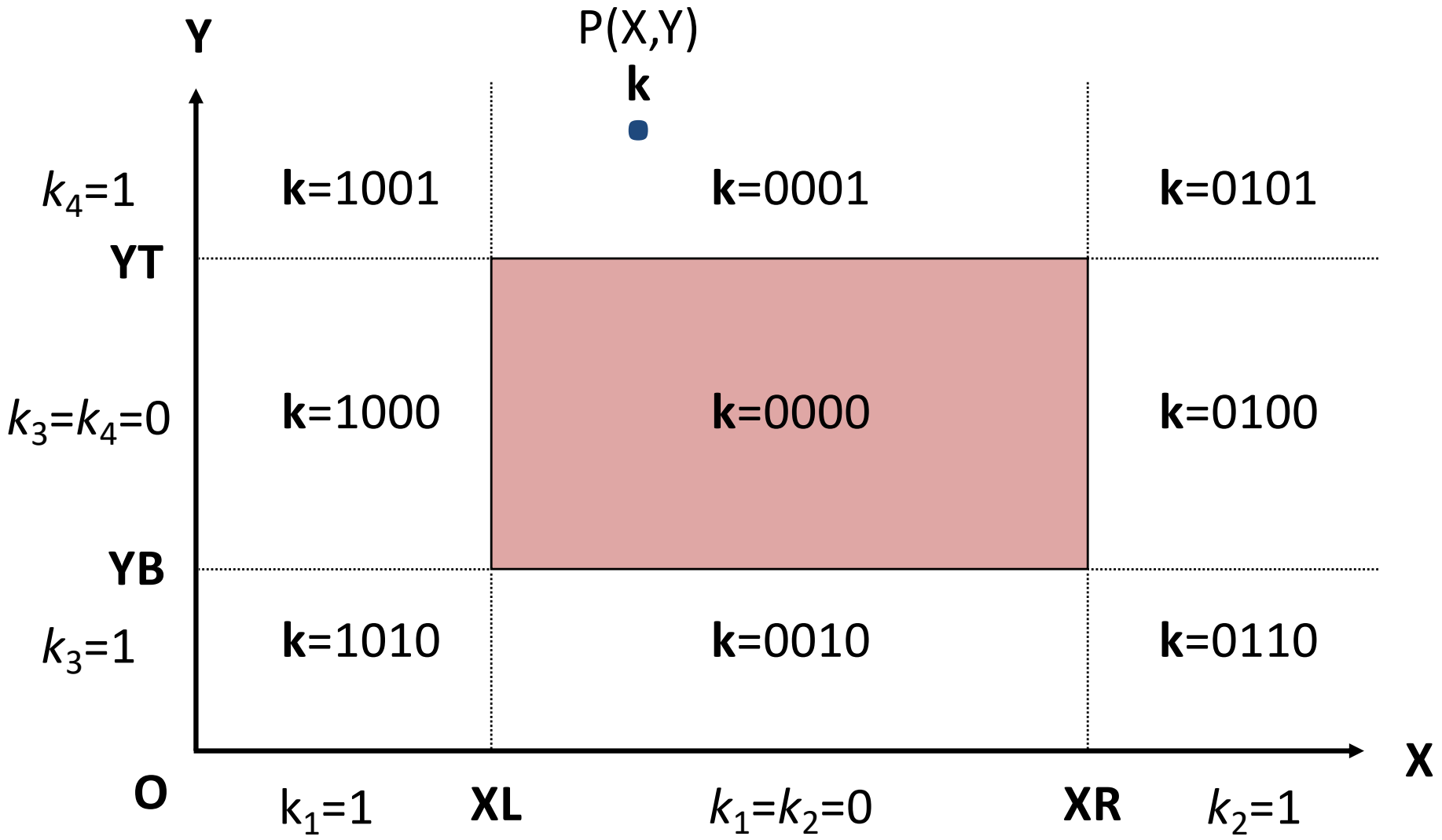
$k_2 = 1$ if X > XR (too much to the right)

$k_3 = 1$ if Y < YB (too low)

$k_4 = 1$ if Y > YT (too high)

Note: k_1 and k_2 cannot be simultaneously equal to 1, same holds for k_3 and k_4 → Hence 9 possible code words (**not** 2^4)

Cohen-Sutherland line clipping algorithm



Cohen-Sutherland line clipping algorithm

- Cohen-Sutherland tries to solve first simple (trivial) cases
- Assign 4-bit code words to end points: $P_1 \rightarrow \mathbf{k_1}$, $P_2 \rightarrow \mathbf{k_2}$

Step 1: if $\mathbf{k_1 = k_2 = 0}$, P_1P_2 is fully visible; otherwise go to Step 2

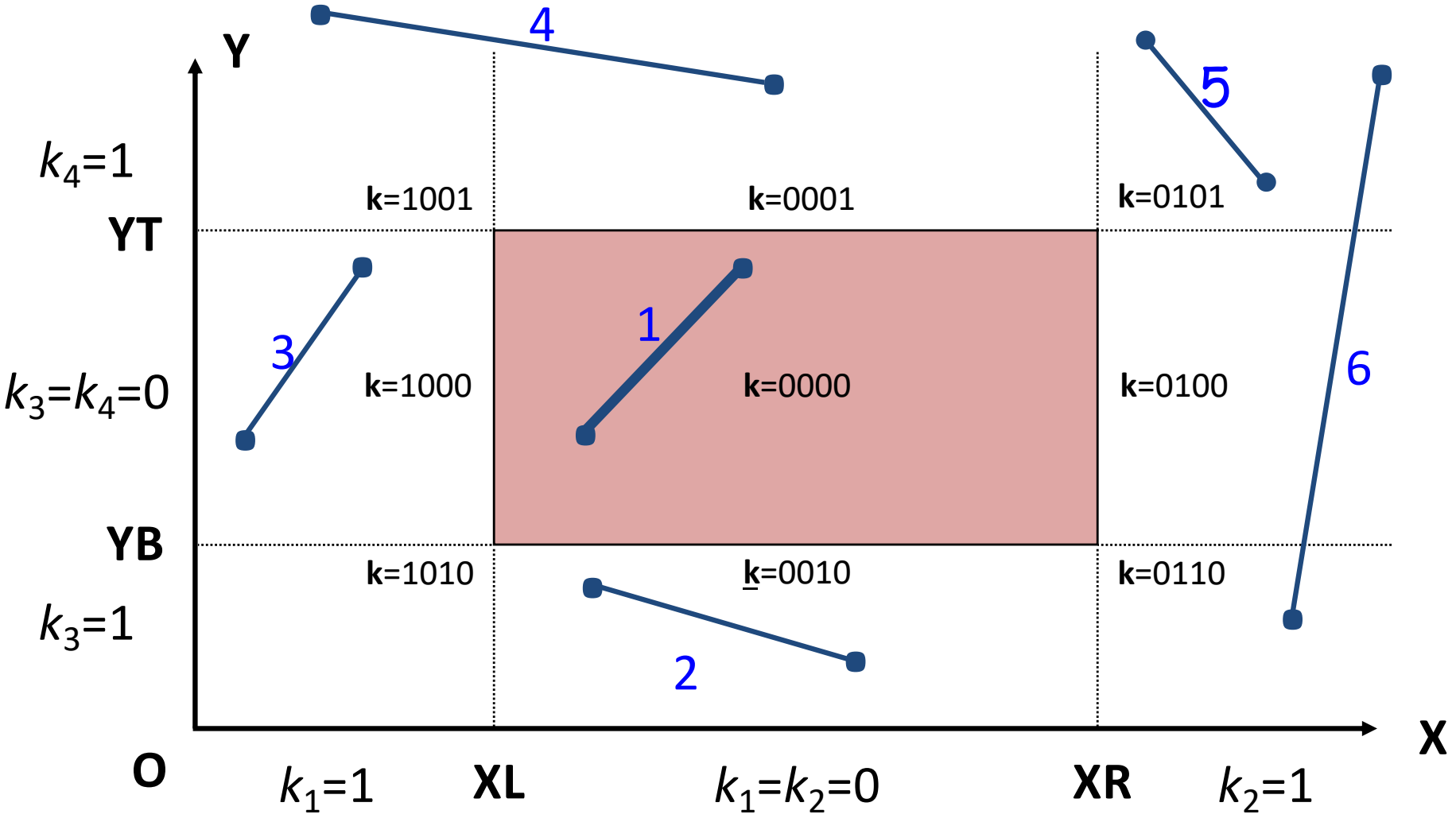
Step 2: Find bit per bit logic AND: $\mathbf{k = k_1 \wedge k_2}$.

If $\mathbf{k_1 \wedge k_2 \neq 0}$, P_1P_2 is fully and "trivially" non visible; otherwise go to Step 3

Step 3: find intersections with lines extending from window edges; Go to Step 1.

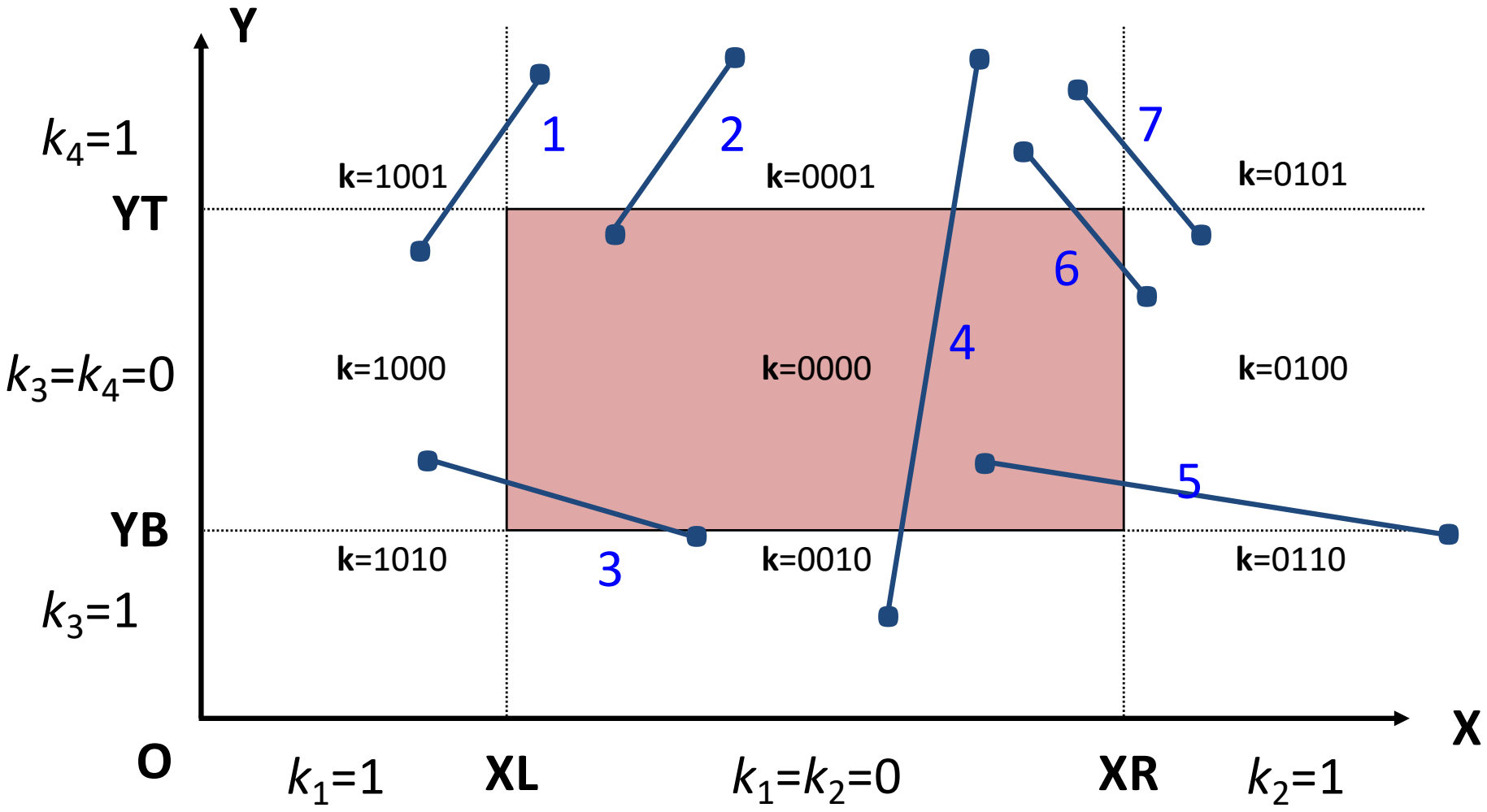
Cohen-Sutherland line clipping algorithm

Trivially visible ($k_1=k_2=0$) and trivially invisible ($k_1 \wedge k_2 \neq 0$) examples



Cohen-Sutherland line clipping algorithm

$k_1 \wedge k_2 = 0$, and not trivially visible can imply partially visible (subject to shortening) or invisible segment (additional testing needed)



Cohen-Sutherland line clipping algorithm

In Step 3, $\mathbf{k_1 \wedge k_2 = 0}$, but $\mathbf{k_1 \neq 0}$ or $\mathbf{k_2 \neq 0}$ (or both)

Otherwise the segment would be fully visible (Step 1)

Suppose that $\mathbf{k_1 \neq 0}$, which means that P_1 is outside

If not, switch P_1 and P_2 : $X_1 \leftrightarrow X_2$; $Y_1 \leftrightarrow Y_2$; $\mathbf{k_1 \leftrightarrow k_2}$

We need to "bring" P_1 on the edge of the window

Actually, replace P_1 by the intersection point of the segment P_1P_2 and the corresponding window edge.

If $k_1 = 1$, find intersection with the left edge XL

If $k_2 = 1$, find intersection with the the right edge XR

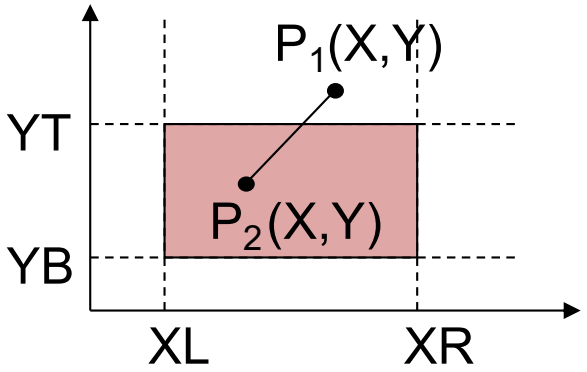
If $k_3 = 1$, find intersection with the bottom edge YB

If $k_4 = 1$ find intersection with the the top edge YT

Go to Step 1 and repeat until P_1' en P_2' are found

Cohen-Sutherland line clipping algorithm

Search for intersections



Intersection with XL

$Y_1 := Y_1 + (XL-X_1) * (Y_2-Y_1) / (X_2-X_1)$

$X_1 := XL$

Intersection with XR: same as above with XR instead of XL

Intersection with YB

$X_1 := X_1 + (YB-Y_1) * (X_2-X_1) / (Y_2-Y_1)$

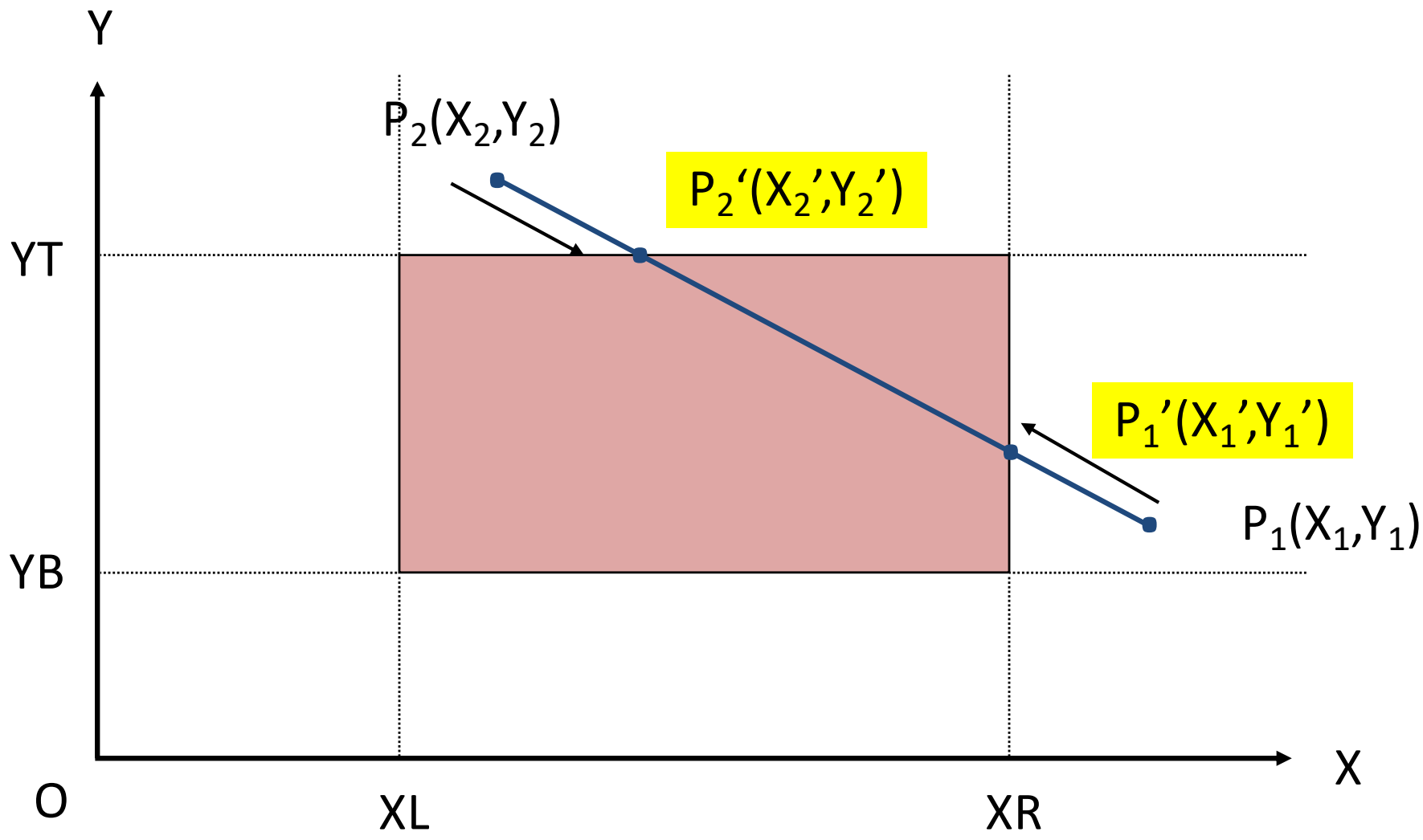
$Y_1 := YB$

Intersection with YT: same as above with YT instead of YB

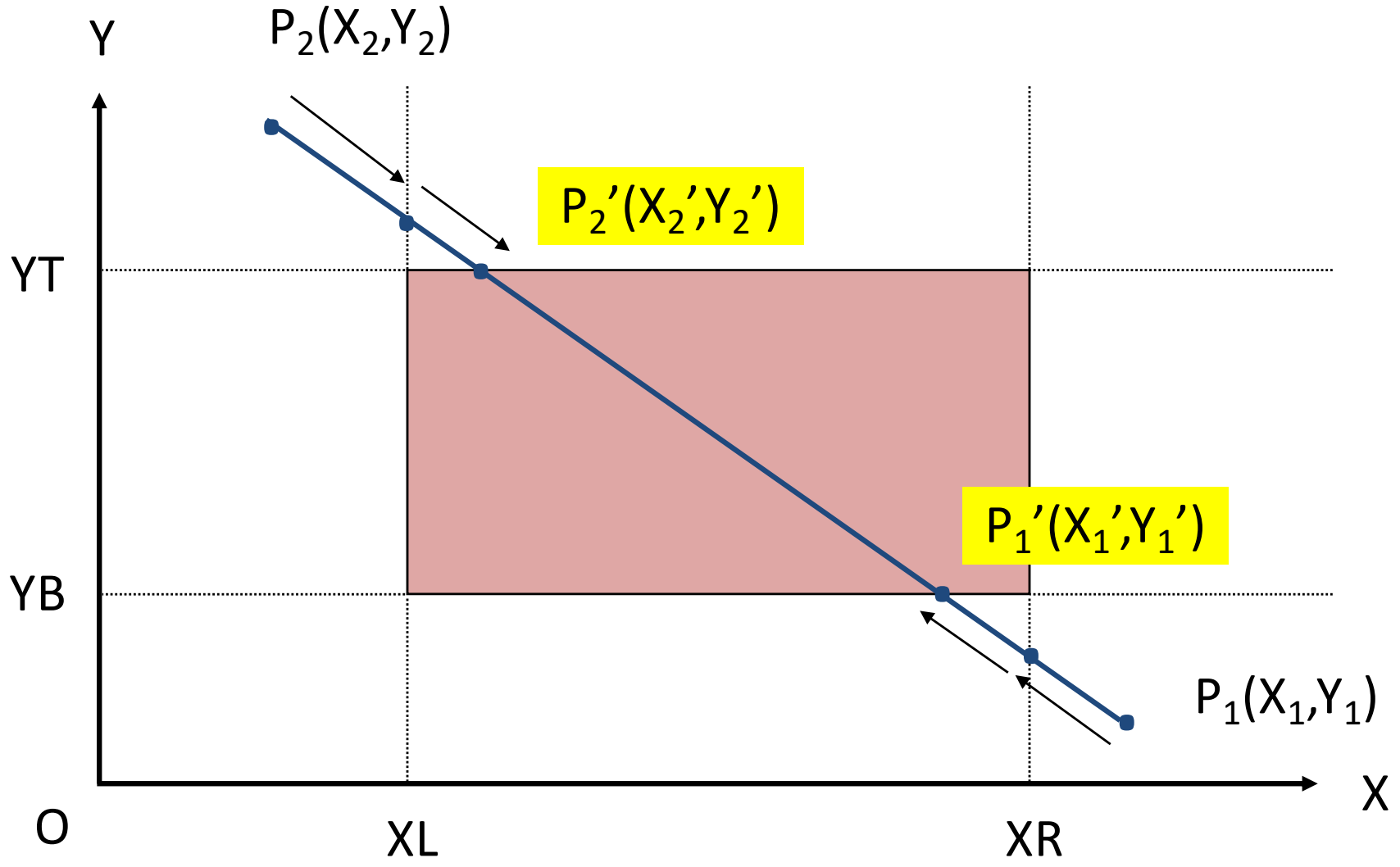
Denominators cannot be equal to 0 (no division by 0)

In the first case (bringing on XL), this would mean that the segment is vertical, and too much to the left → already eliminated

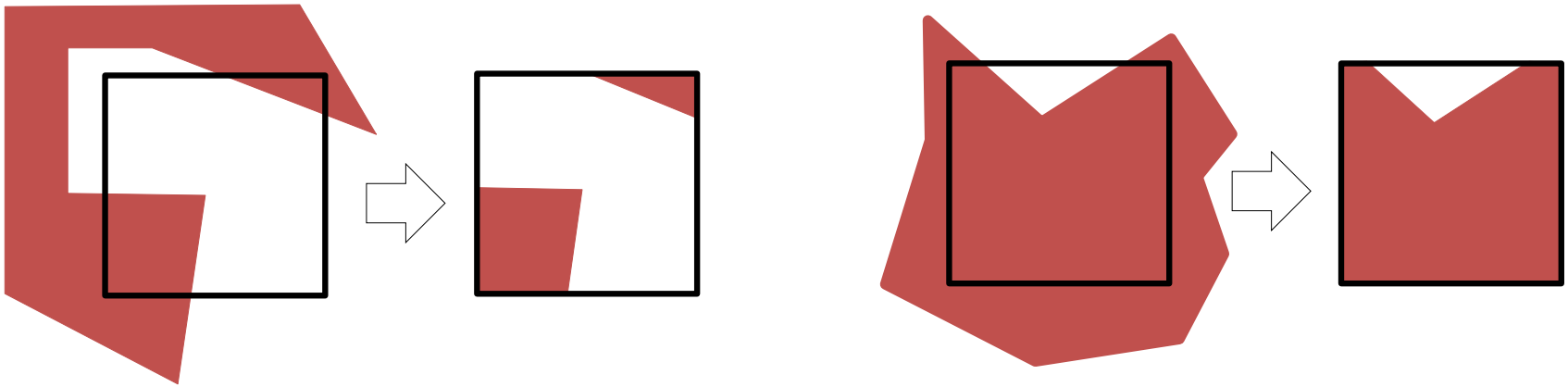
Cohen-Sutherland line clipping algorithm



Cohen-Sutherland line clipping algorithm



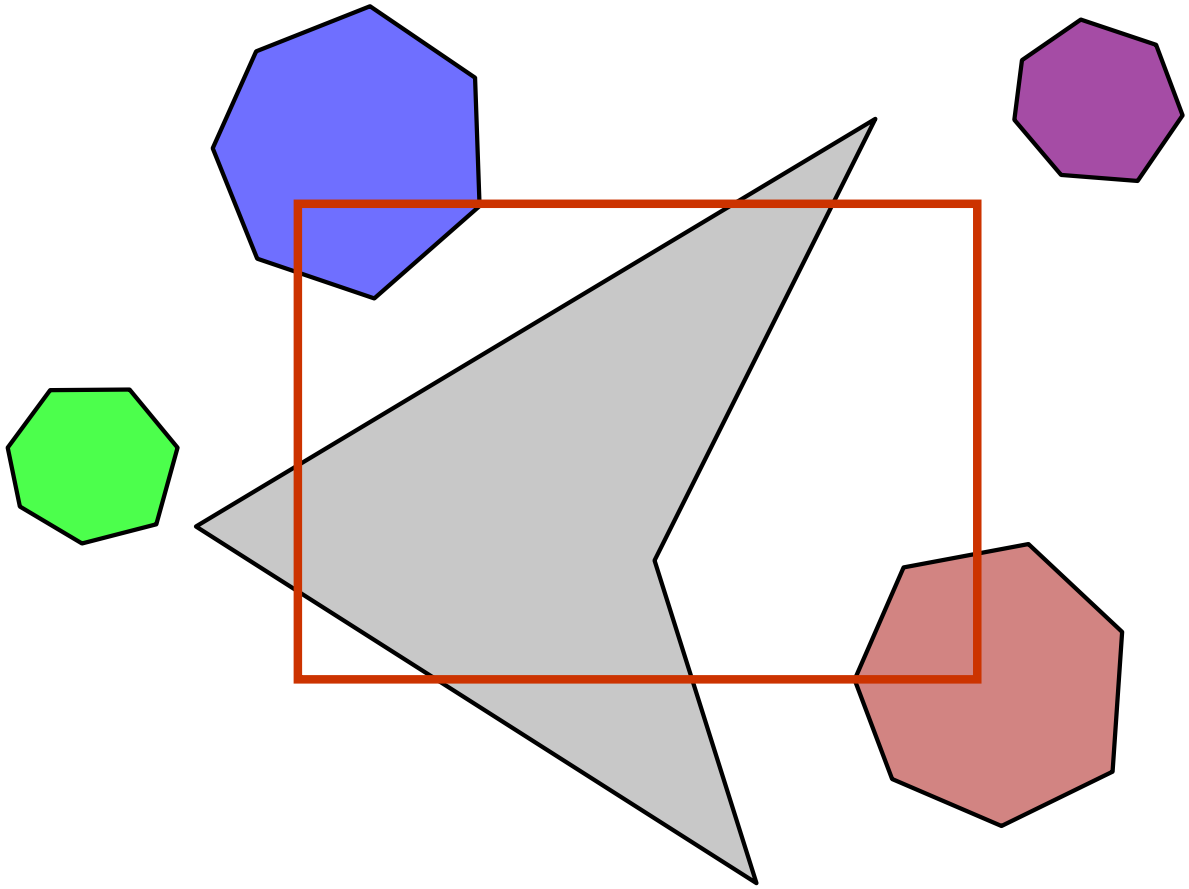
Polygon clipping



- Polygon clipping = scissoring according to clip window
- Must deal with many different cases
- Clipping a single polygon can result in multiple polygons

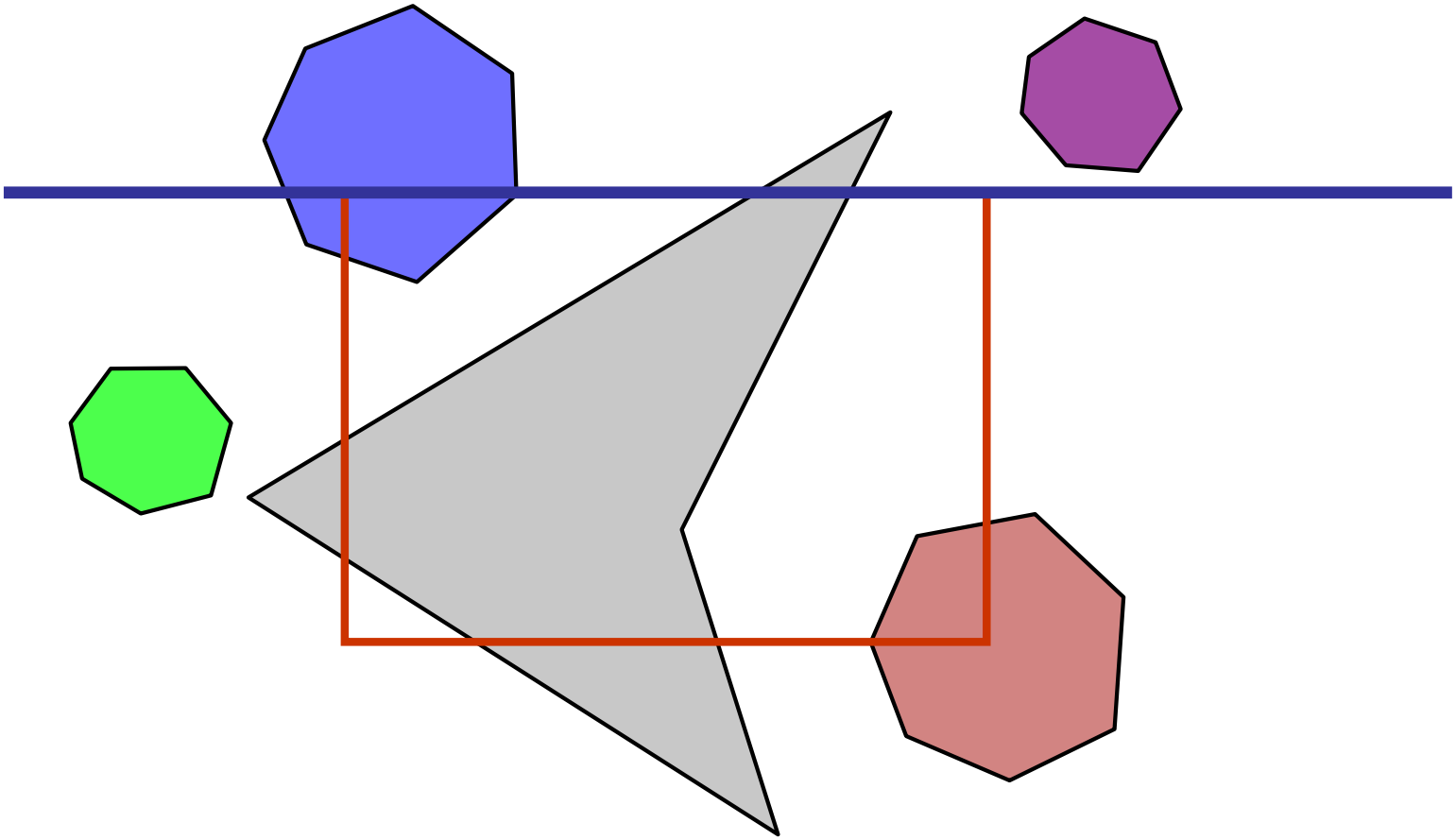
Polygon clipping

- Find the parts of polygons inside the clip window



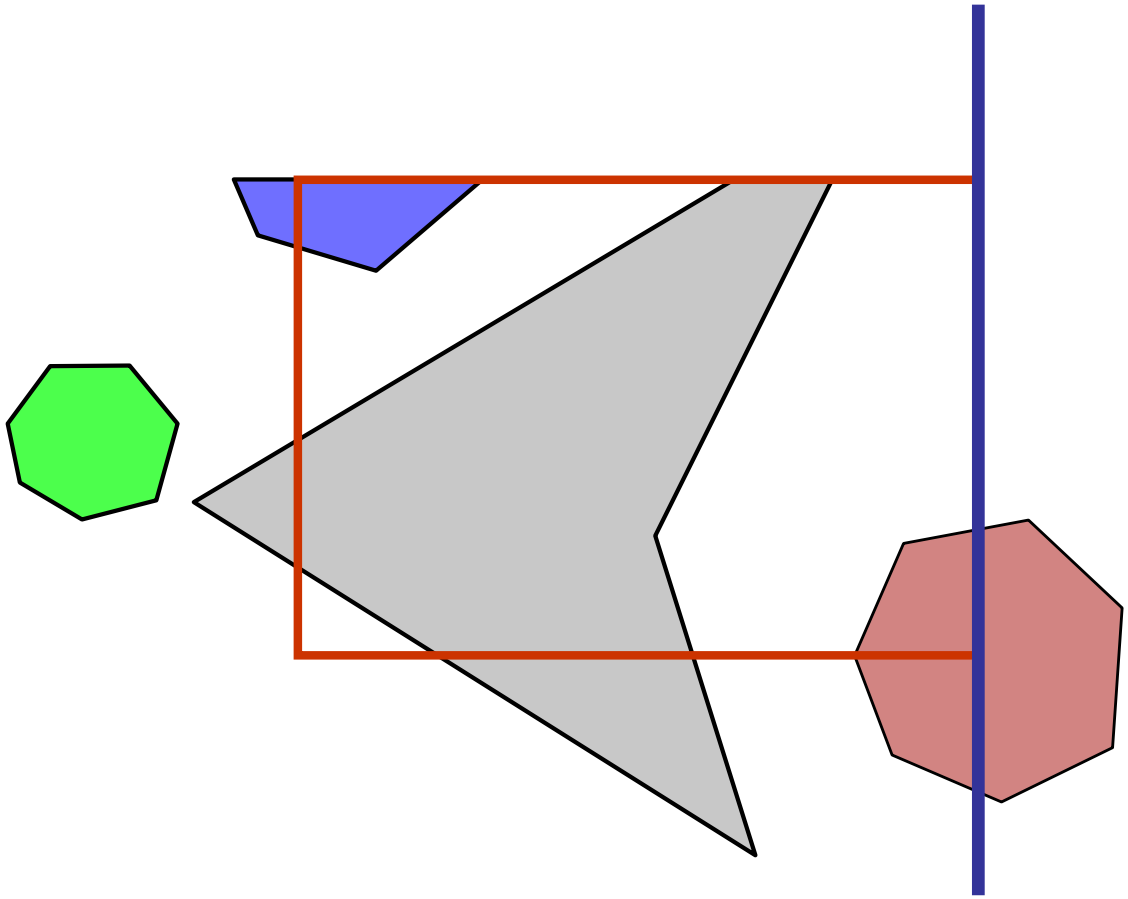
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time



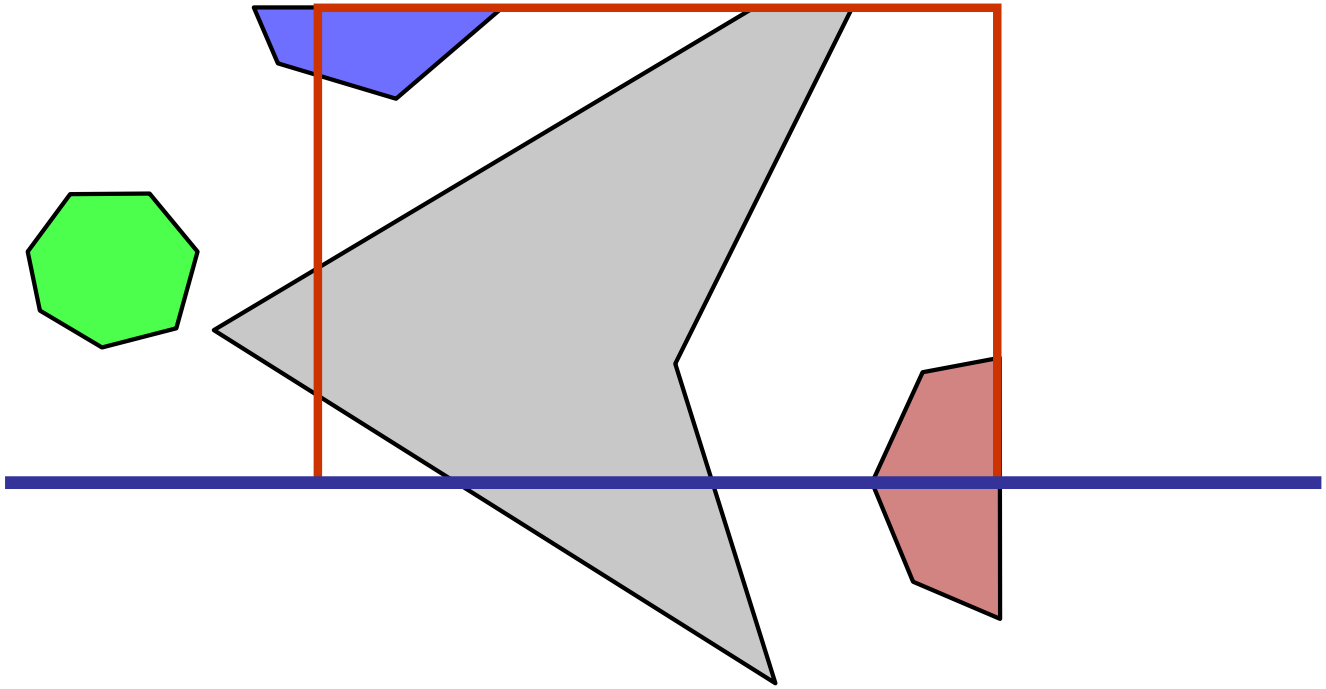
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time



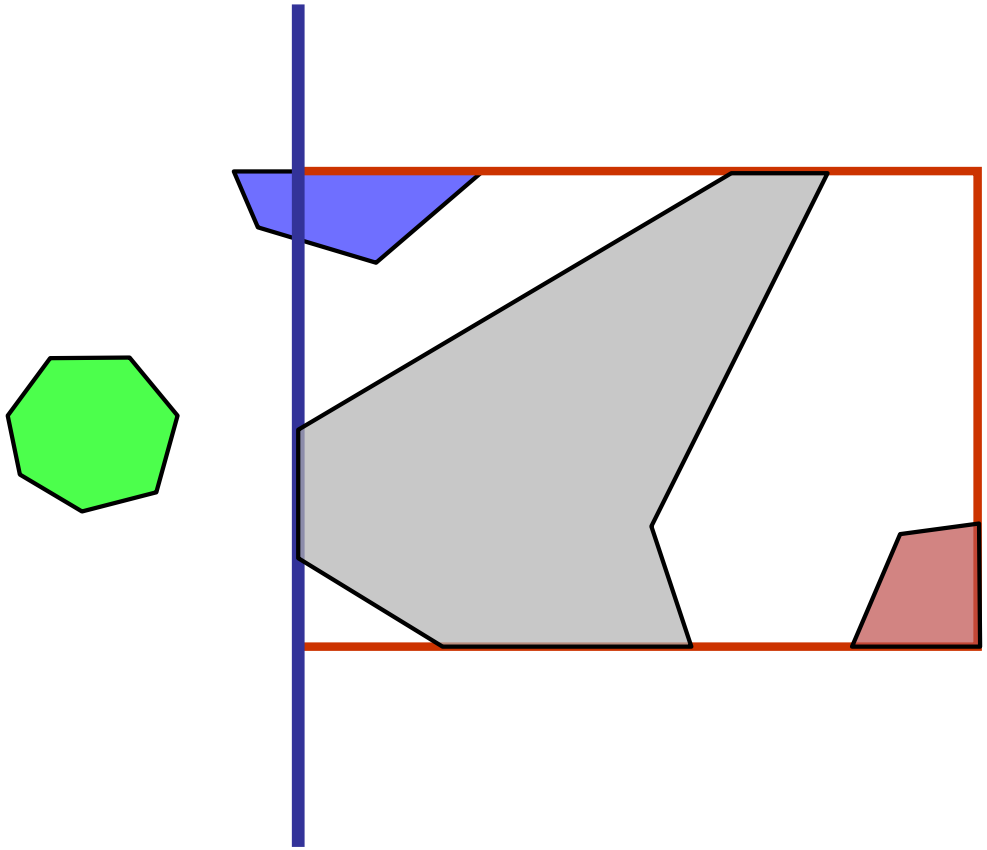
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time



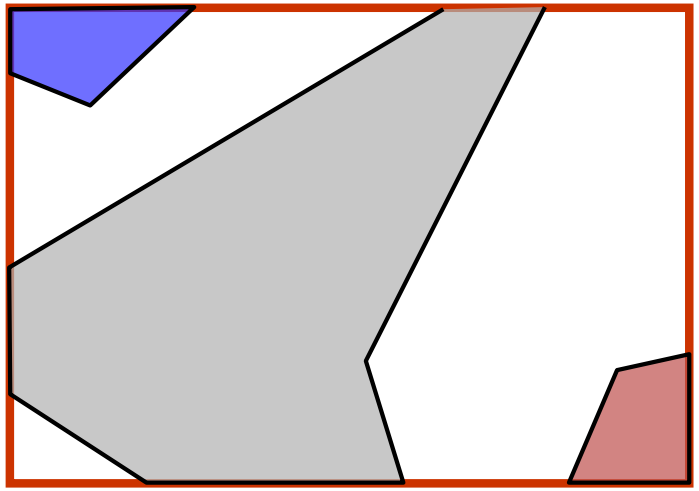
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time



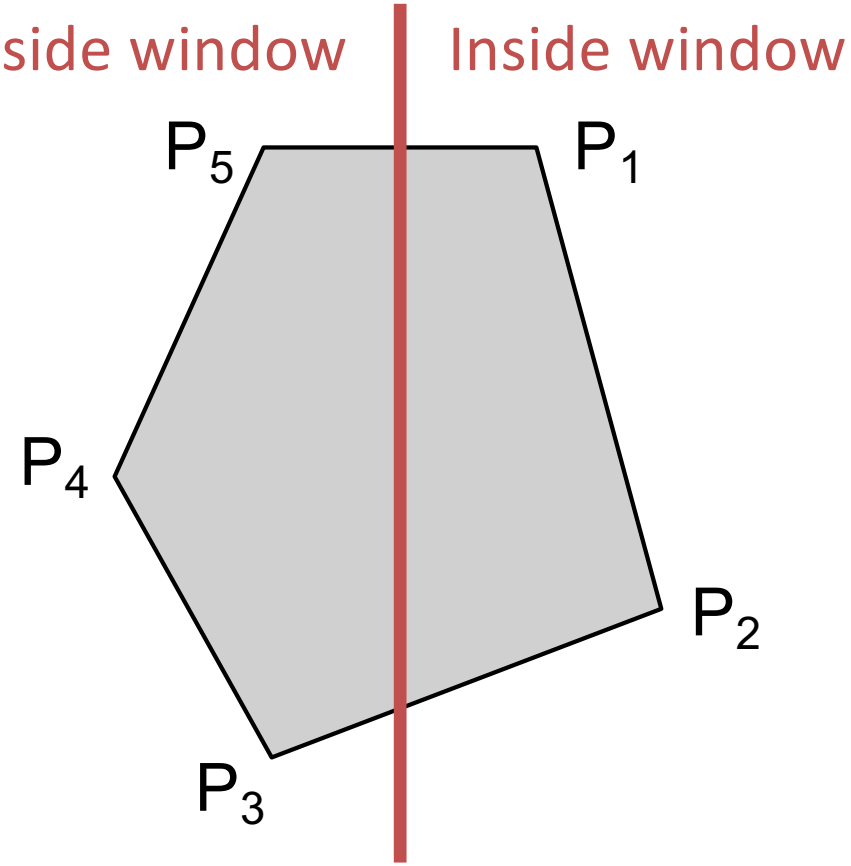
Polygon clipping

- Clipping to one window boundary at a time



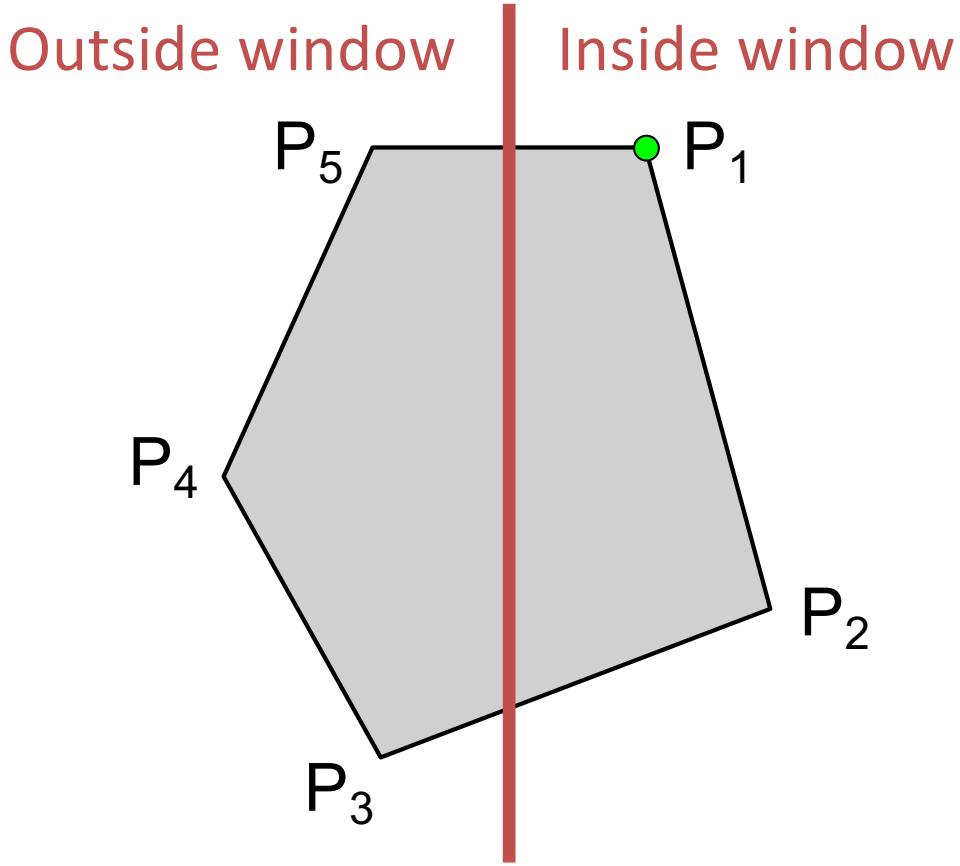
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



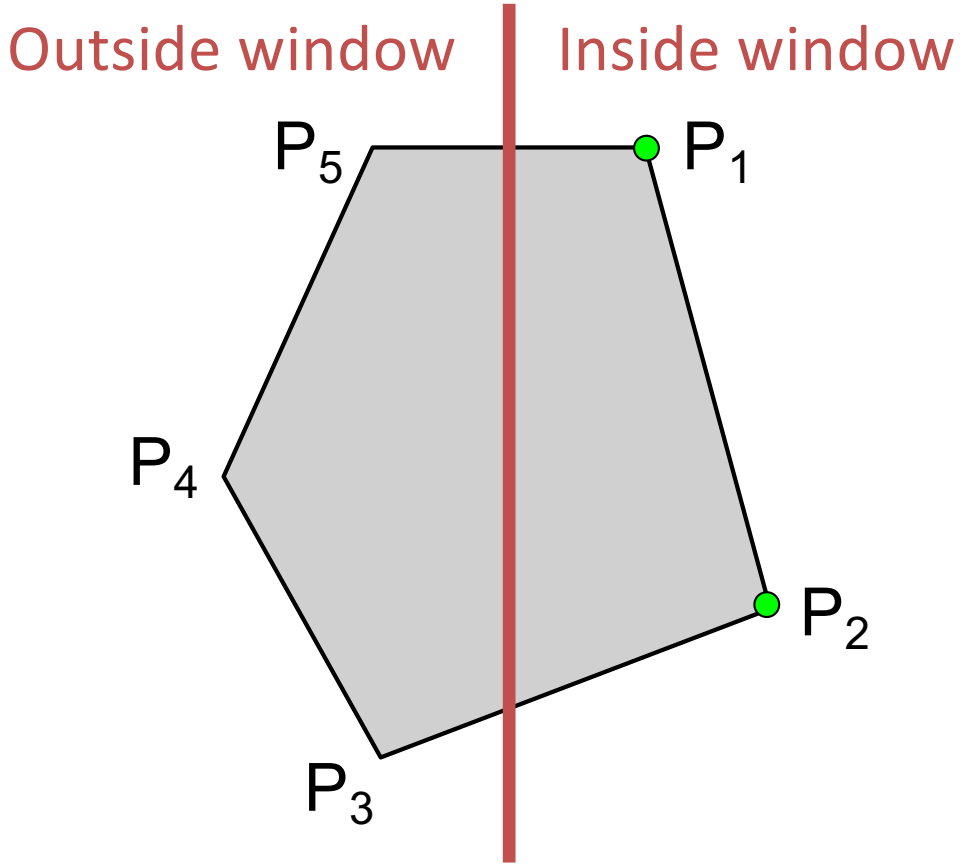
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



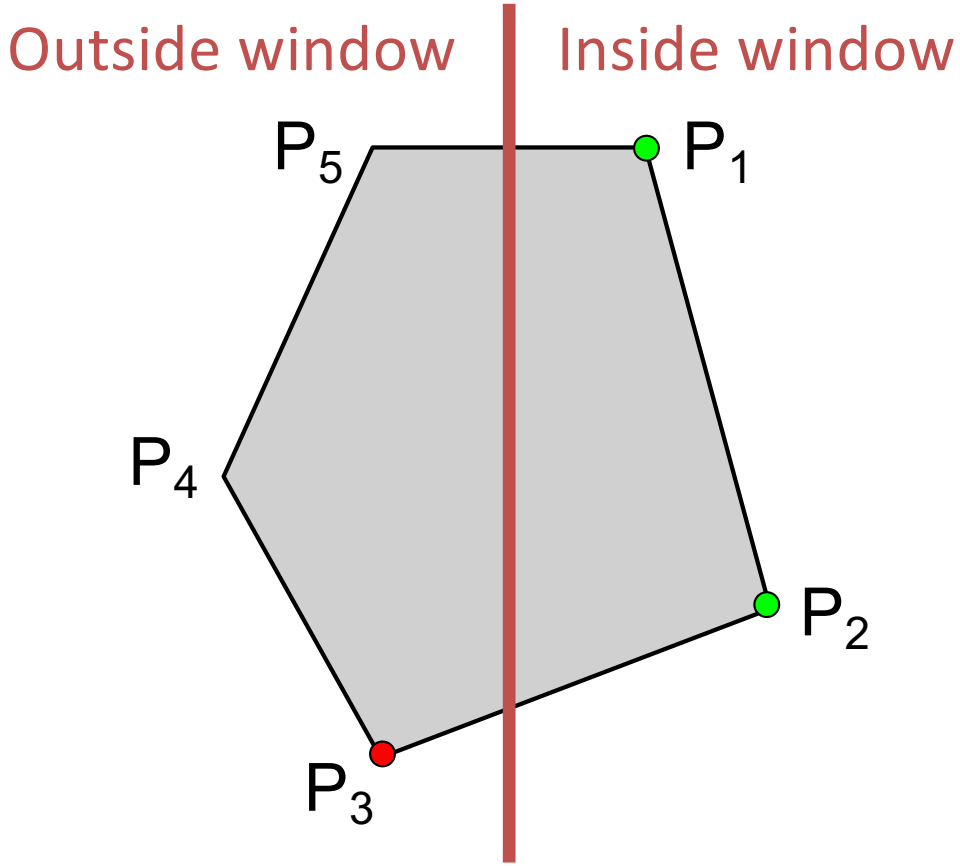
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



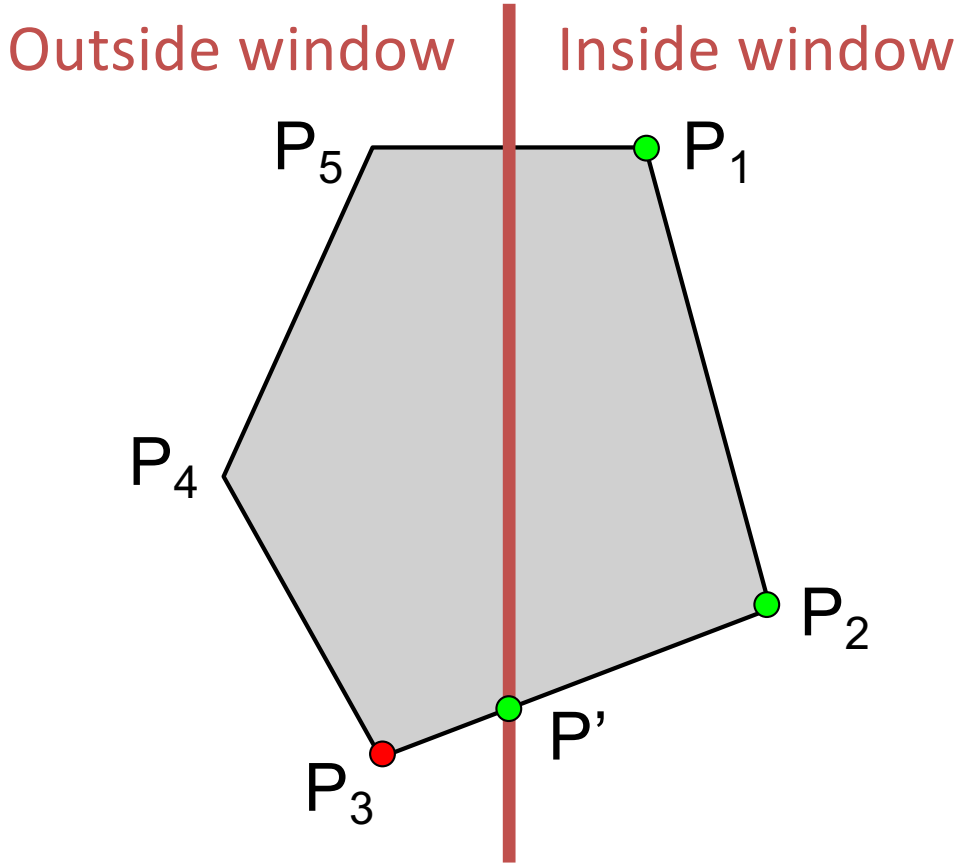
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



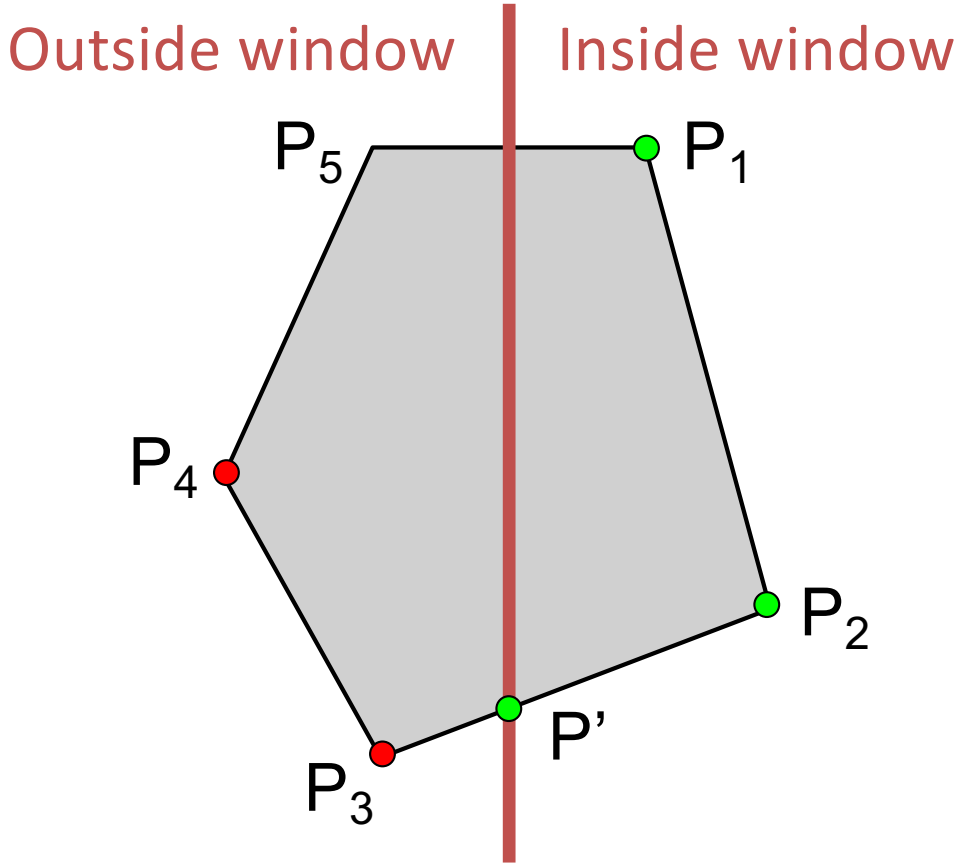
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



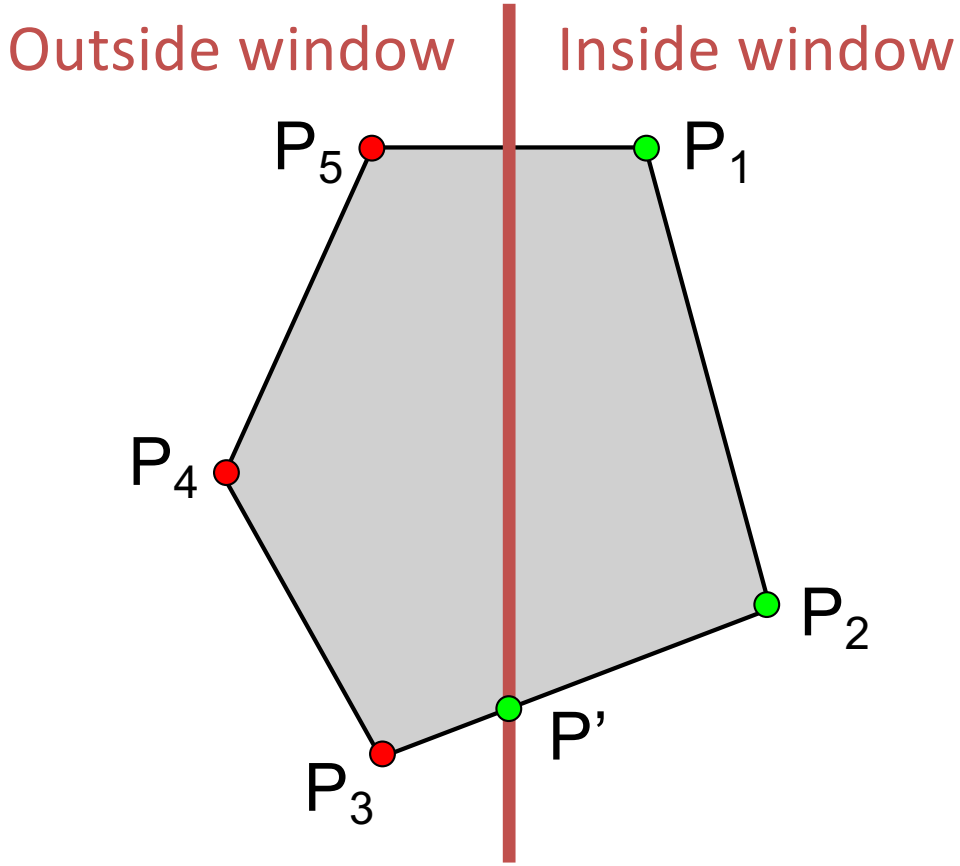
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



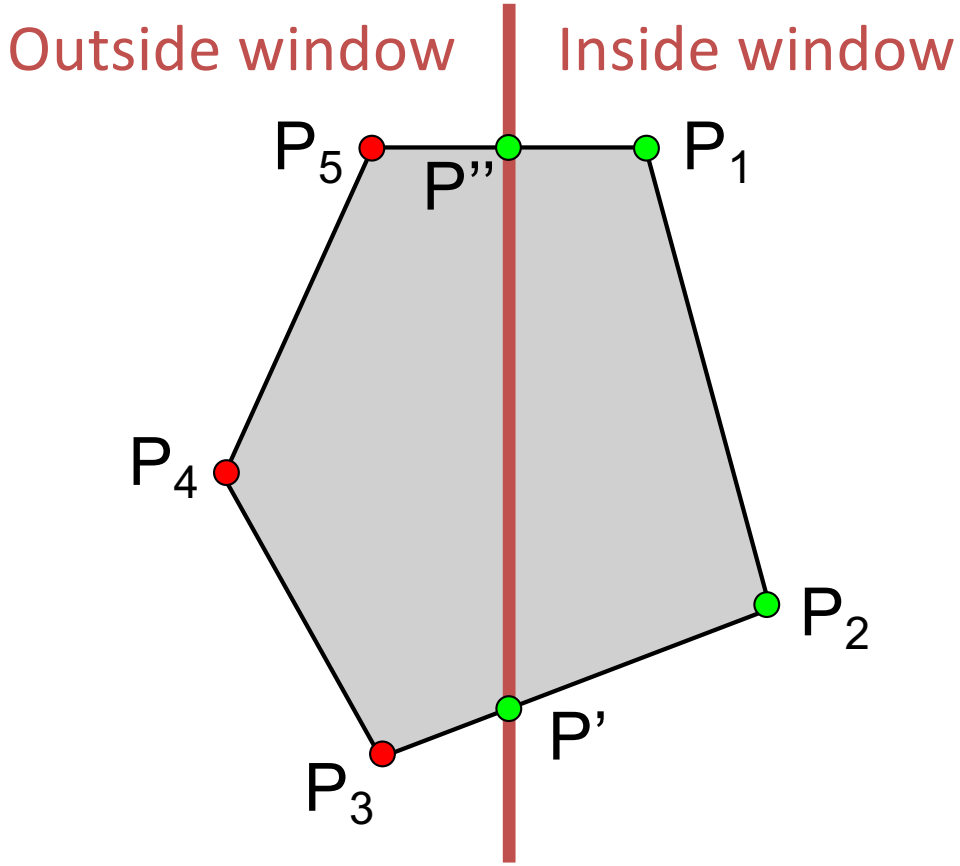
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



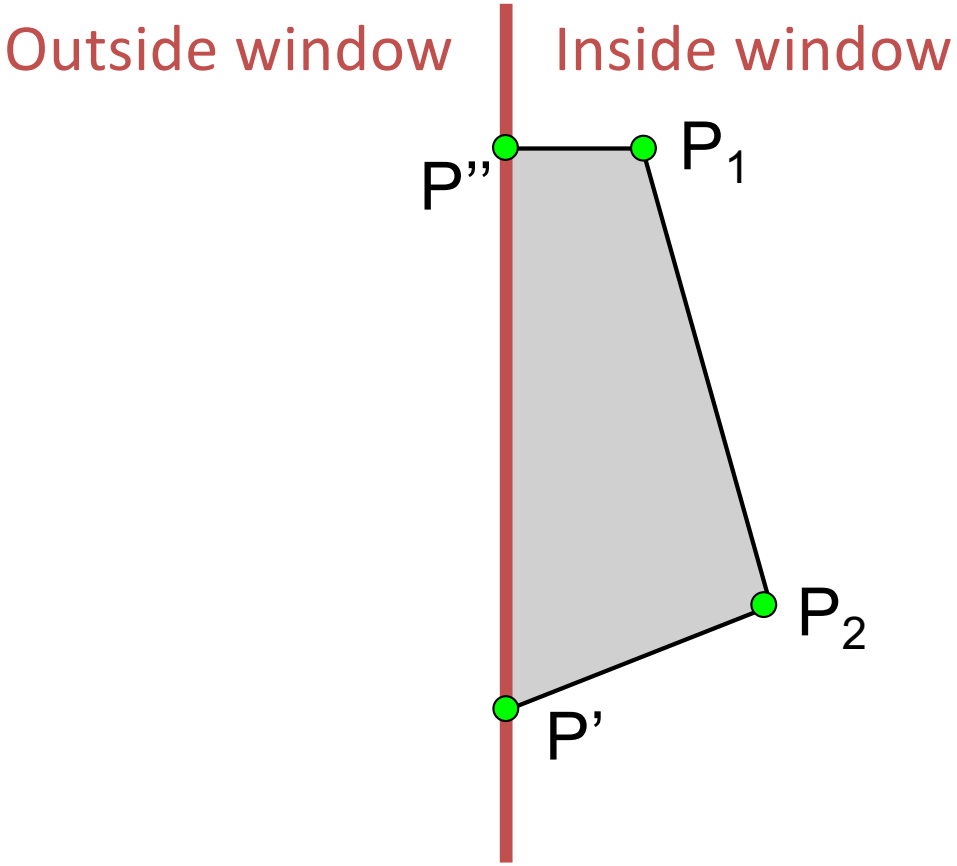
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary

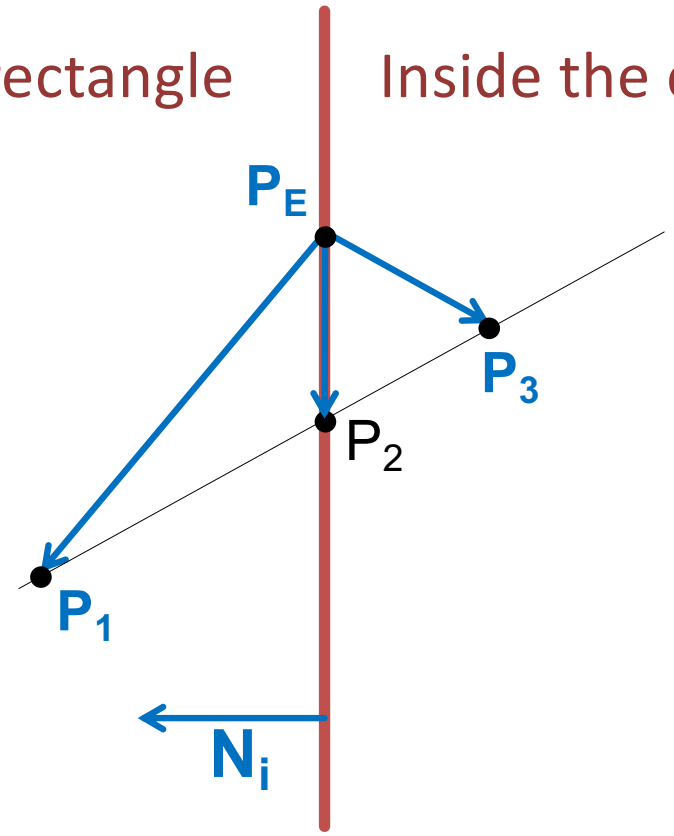


Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary



Inside test



$N_i \cdot (P_1-P_E) > 0$ (outside)

$N_i \cdot (P_2-P_E) = 0$ (on edge)

$N_i \cdot (P_3-P_E) < 0$ (inside)

- Define edge's outward normal N_i
- For a point P_i test the sign of the dot product $N_i \cdot (P_i-P_E)$

Summary

- Scan conversion of lines: midpoint line algorithm
- Scan conversion of circles and ellipses: use symmetries
- Clipping in 2D: Cohen-Sutherland algorithm based on assigning 4-bit code words to the end points of the line
- Clipping polygons – inside test