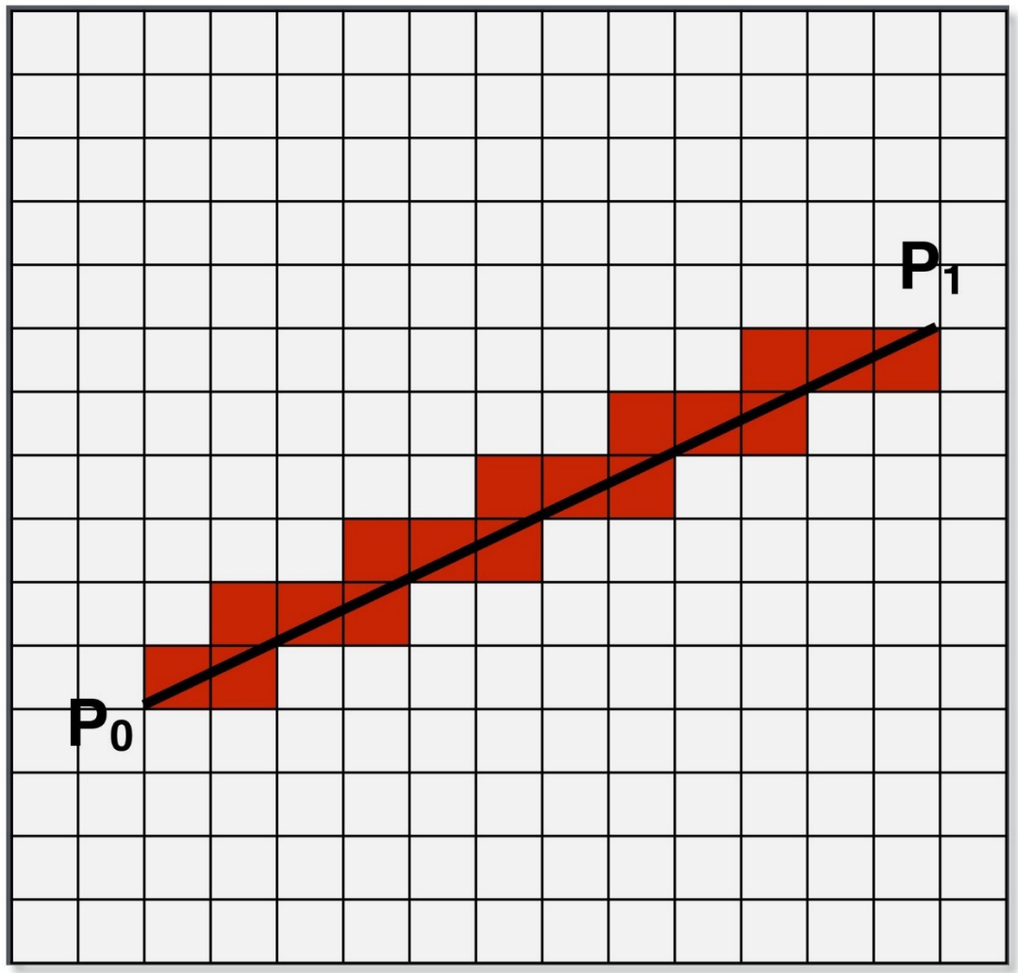
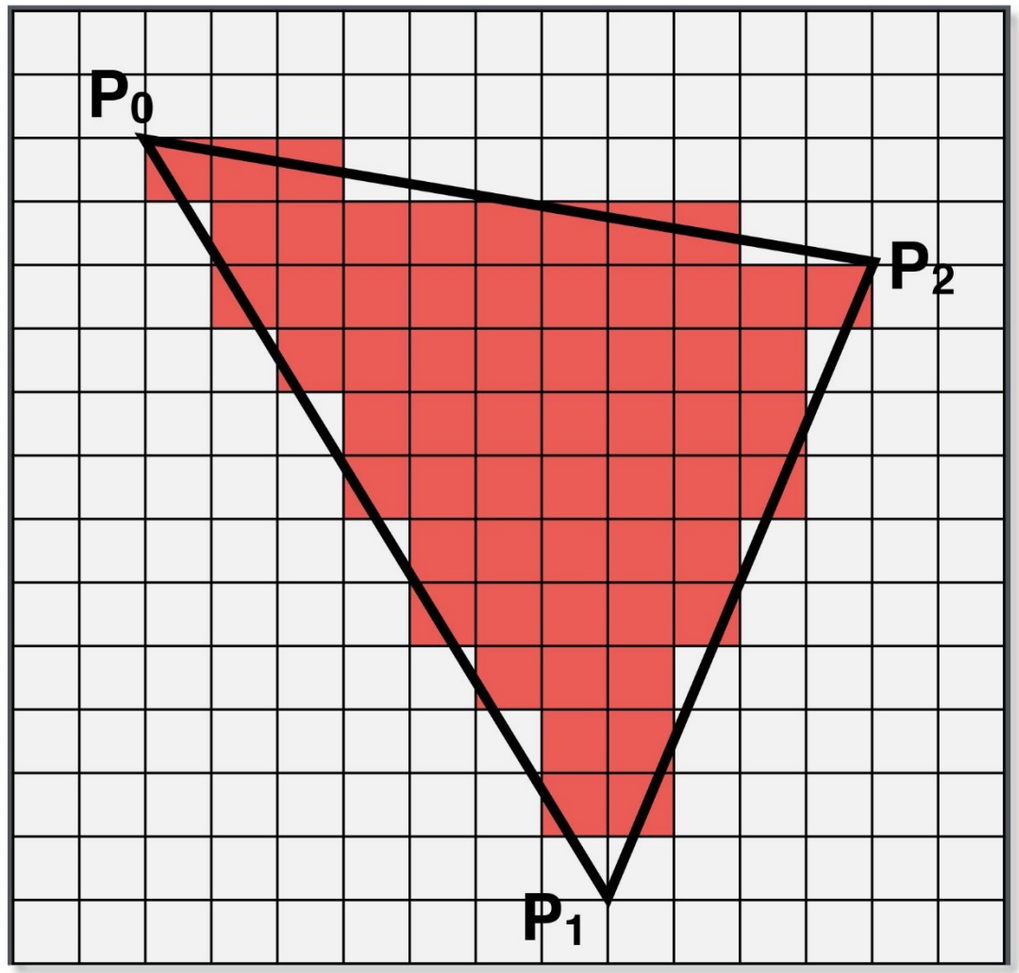
Rasterization

Example of rasterization





Rasterization

- **Rasterization** (or scan-conversion)

  process of converting two-dimensional primitives into a discrete pixel representation

  identify the pixels that best describe the primitives

- Two main **challenges**

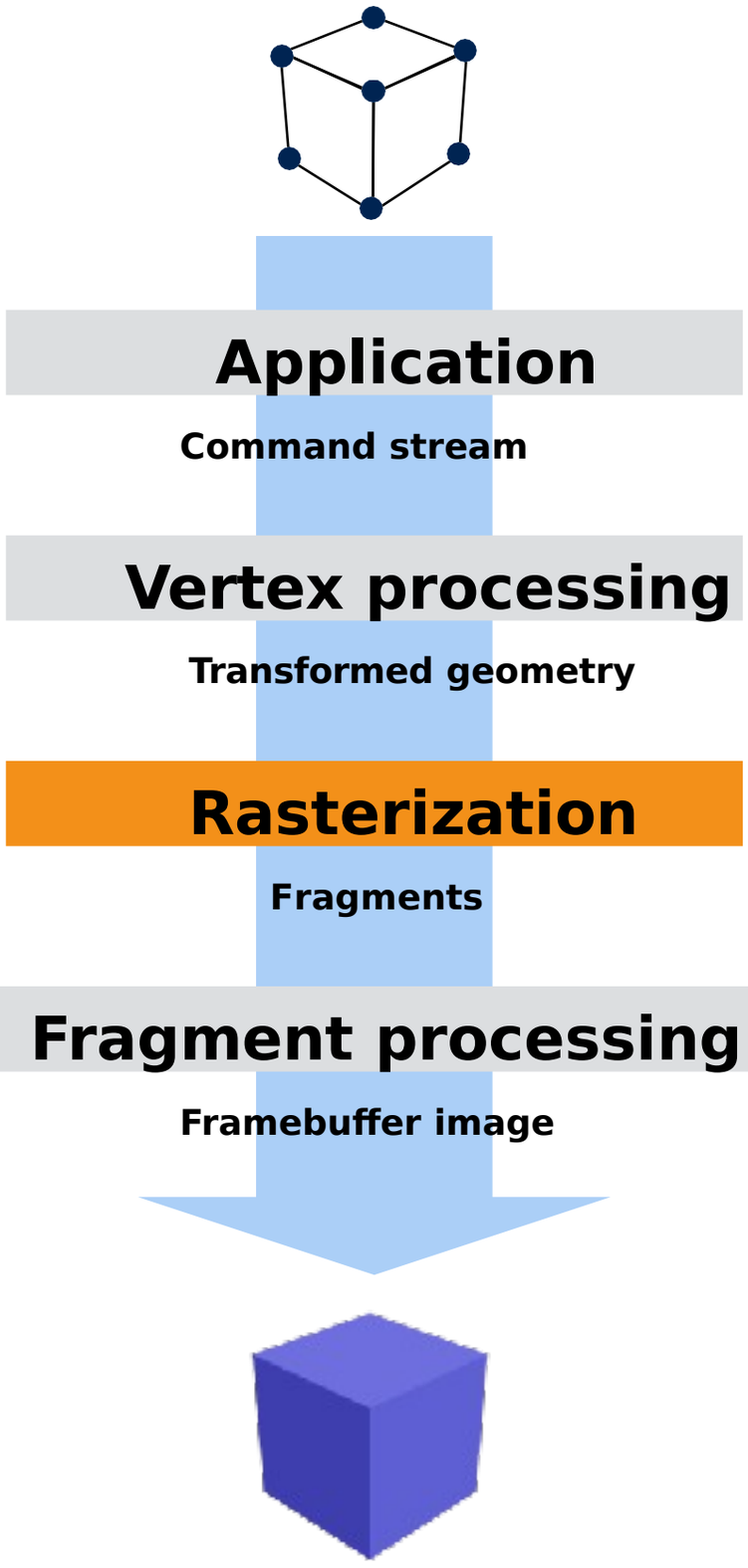
  to determine the pixels that accurately describe the primitive

  to be efficient

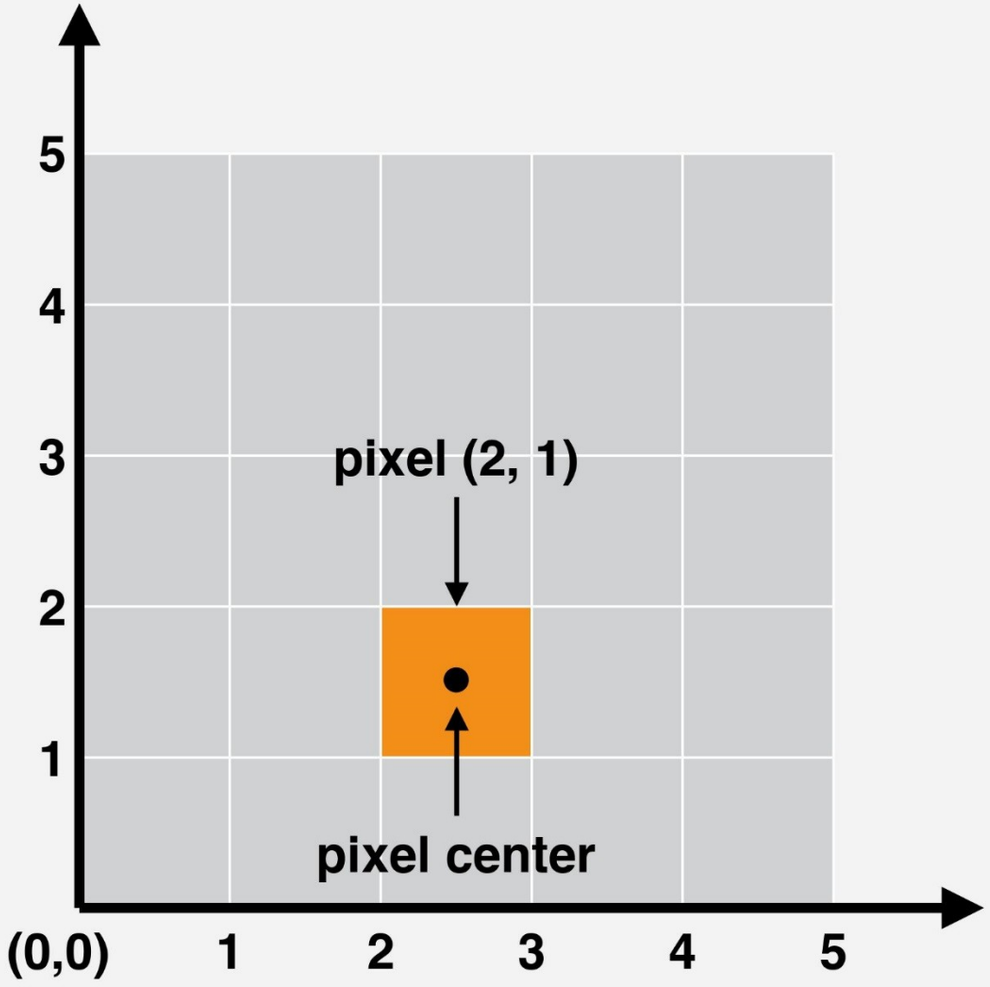
Real-time rendering pipeline

- **Rasterization**
  - **enumerates** the pixels that are covered by the primitive
  - **interpolates** values, called attributes, across the primitive
- The output of the rasterizer stage is a set of **fragments**, one for each pixel covered by the primitive
- Fragment data
  - raster position depth (z-value)
  - interpolated attributes (color, texture
  - coordinates, etc.)

    others
  -

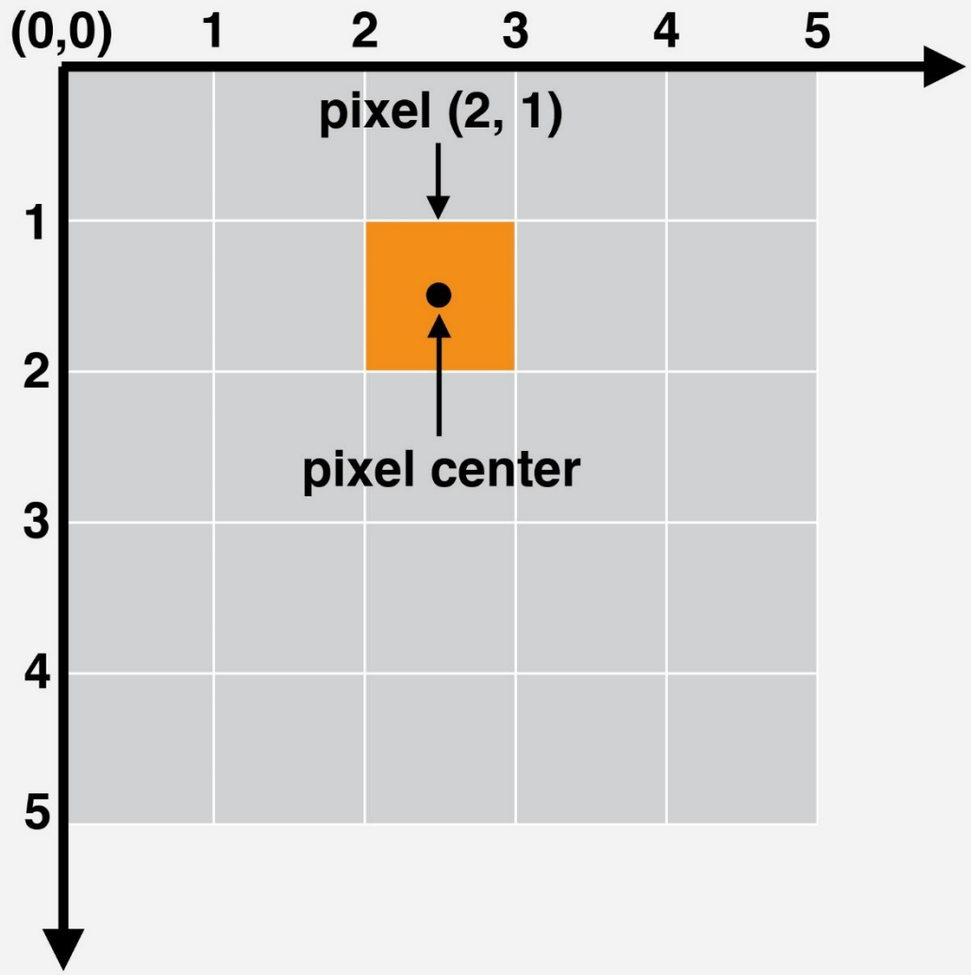


Pixel coordinates conventions



OpenGL

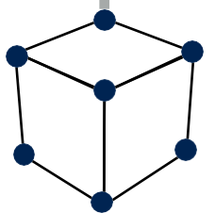
- pixel center located at **half integers**
- (0,0) located at screen **bottom left** corner



DirectX > 10

- pixel center located at **half integers**
- (0,0) located at screen **top left** corner

Rasterization



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

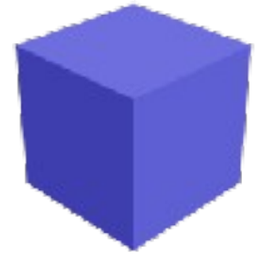
Window coordinates (plus z-value)

Rasterization

Screen coordinates

Fragment processing

Frame buffer



Line drawing

Line equation:

$$y = mx + b$$

$$b = y_1 - mx_1$$

Slope of the line:

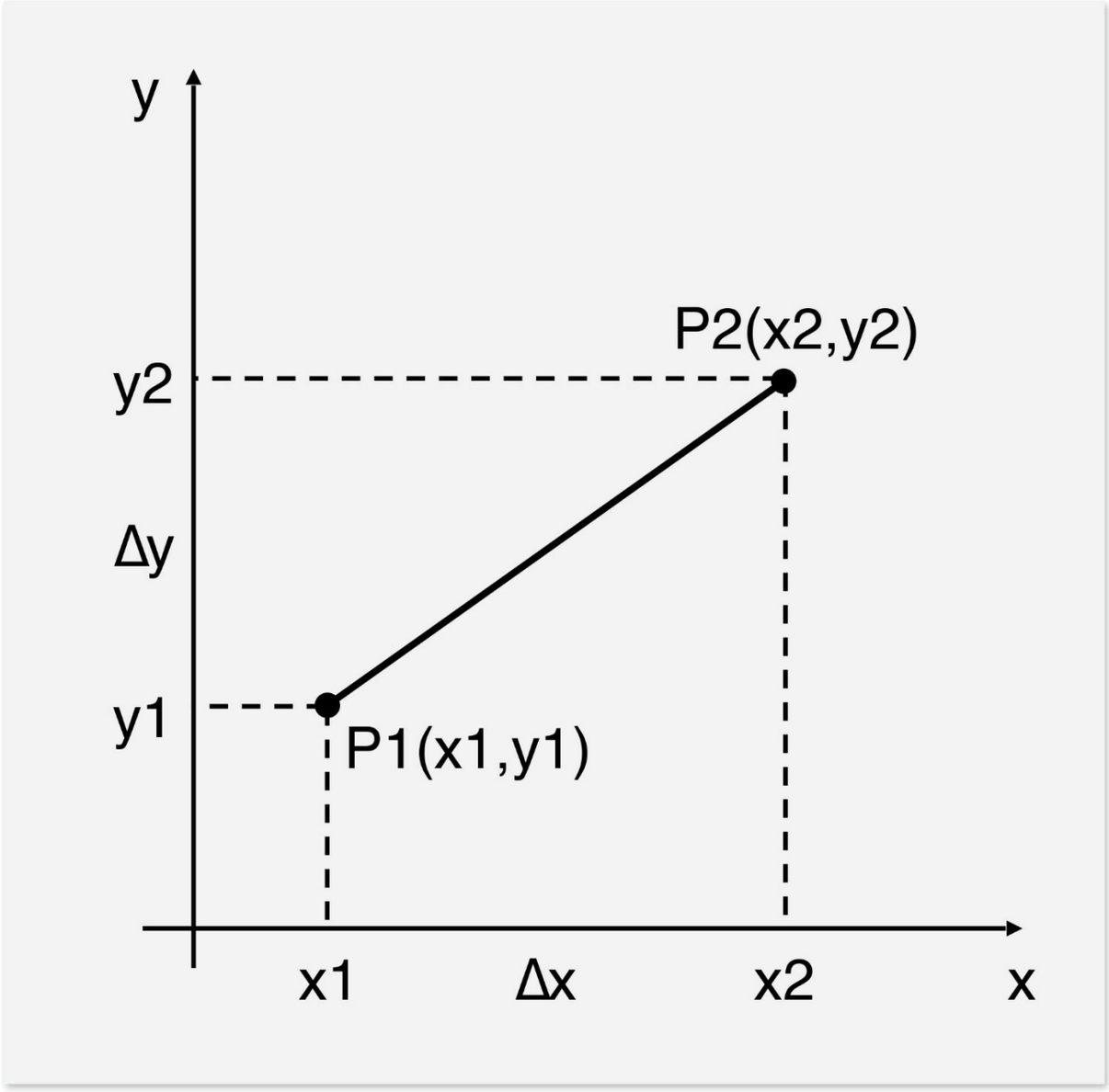
$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

$$\Delta y = m\Delta x$$

Assumptions:

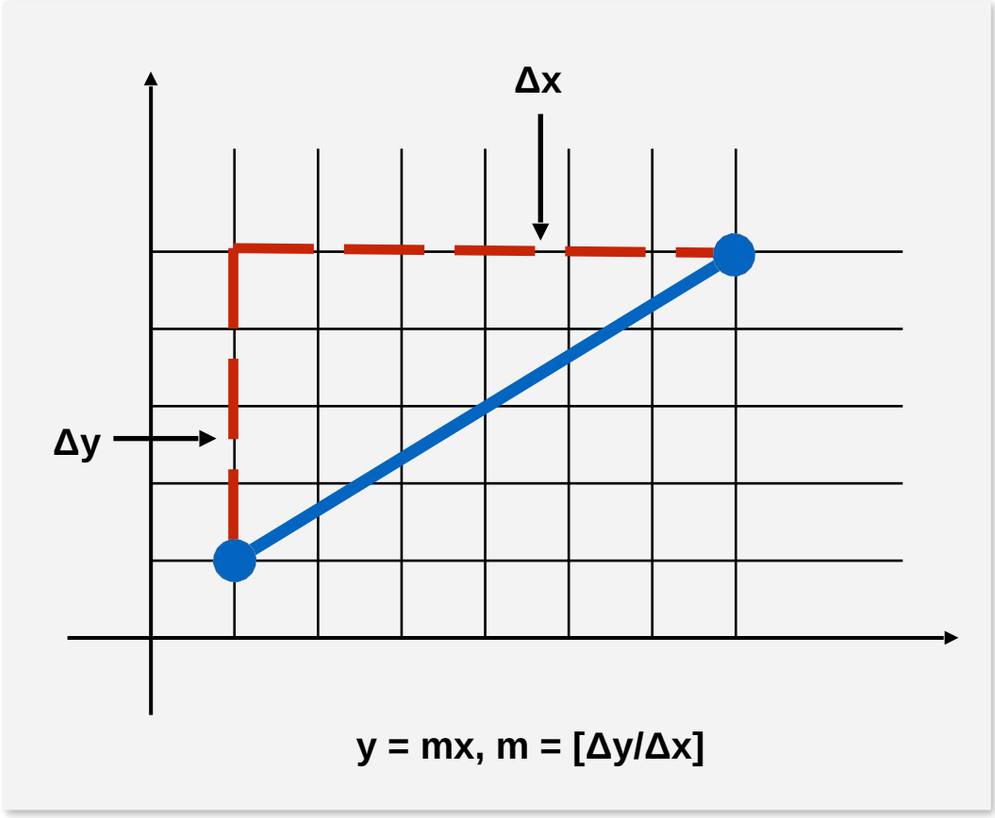
$$m > 0$$

$$x_1 < x_2$$



Basic incremental algorithm

```
void   Line(int x1, int y1,int x2,int y2){
    float  yt,yi;
    int dx = x2 - x1;
    int  dy = y2 – y1;
    float m = dy/(float)dx;

    for(int xi = 0;  xi < dx;  xi++){
        yt = m * xi + y1;
        yi = floor(yt);
        DisplayPixel(xi + x1,   yi);
    }
}
```



- Inefficient
- **Objective**: avoid floating-point multiplication

DDA (Digital Differential Analyzer) algorithm

$$\Delta y = m\Delta x, m > 0$$

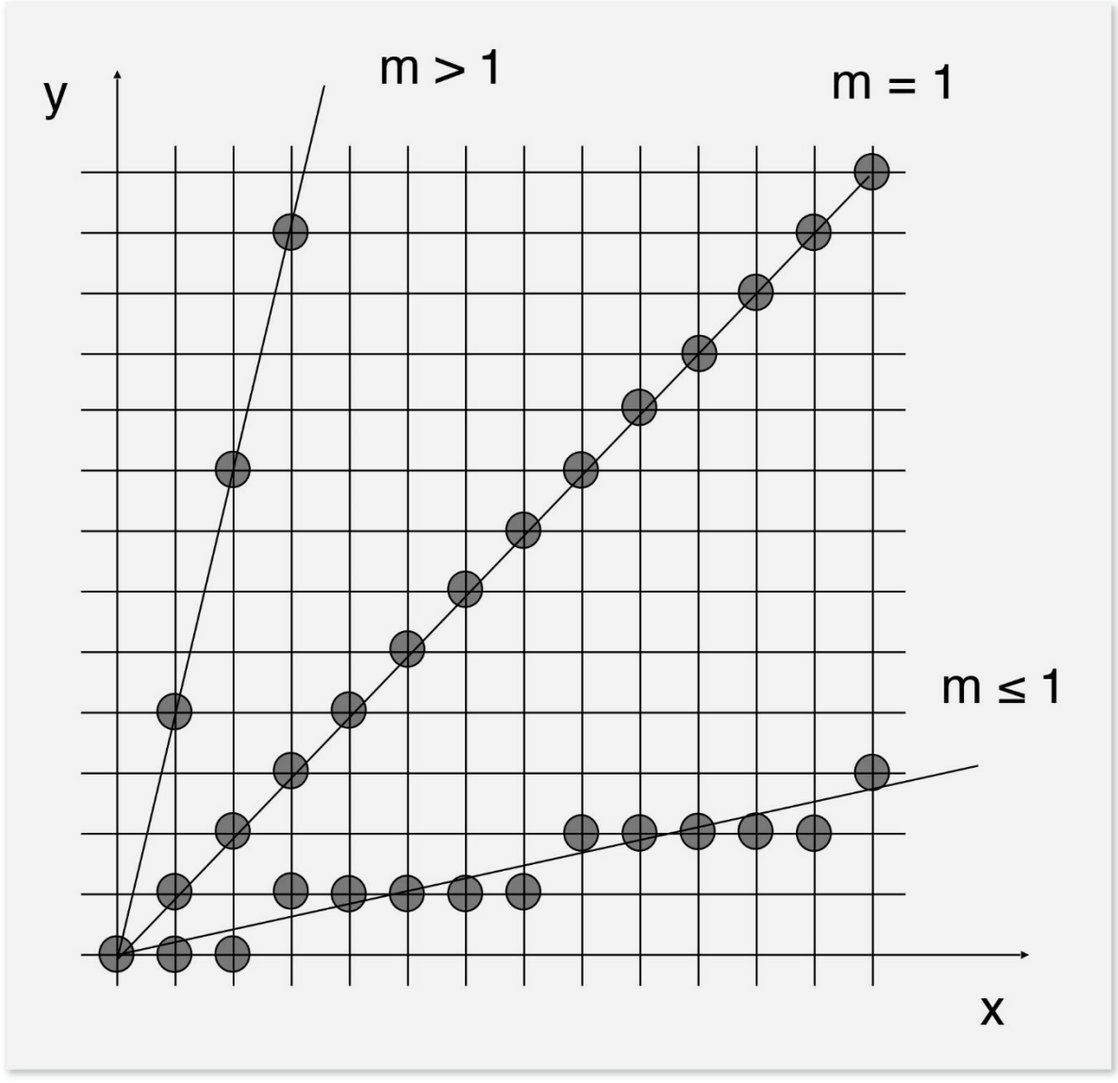
$$y_i = mx_i + b$$

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

$$y_{i+1} = mx_{i+1} + b = m(x_i + 1) + b$$

$$y_{i+1} = mx_i + b + m$$

$$y_{i+1} = y_i + m$$



DDA (Digital Differential Analyzer) algorithm

```
void LineDDA(int x1, int y1, int x2, int y2){
  int   dx =  x2 -  x1;
  int   dy =  y2 -  y1;
  int   x;
  float y;
  float m  =  dy /  (float)dx;
  x  =  x1;  y = y1;
  DisplayPixel(x1,  y1);

  for(int  k  =  1; k  <= dx;   k++)  {
     x +=  1;
     y +=  m;
     DisplayPixel(x,y);
  }
}
```

DDA (Digital Differential Analyzer) algorithm

Case 1: 0 < m < 1

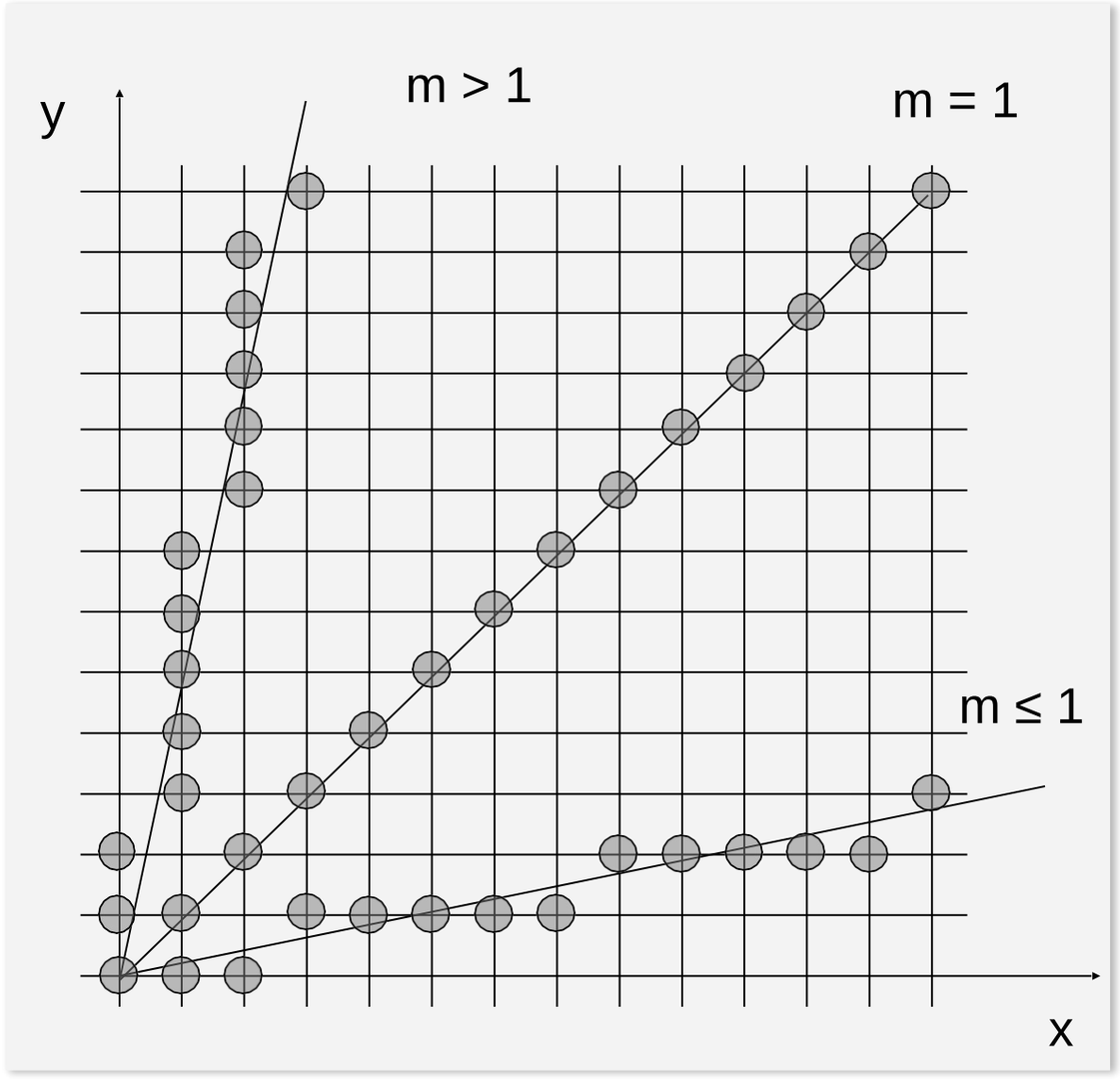
$$x_{i+1} = x_i + 1$$

$$y_{i+1} = y_i + m$$

Case 2: m > 1

$$x_{i+1} = x_i + \frac{1}{m}$$

$$y_{i+1} = y_i + 1$$



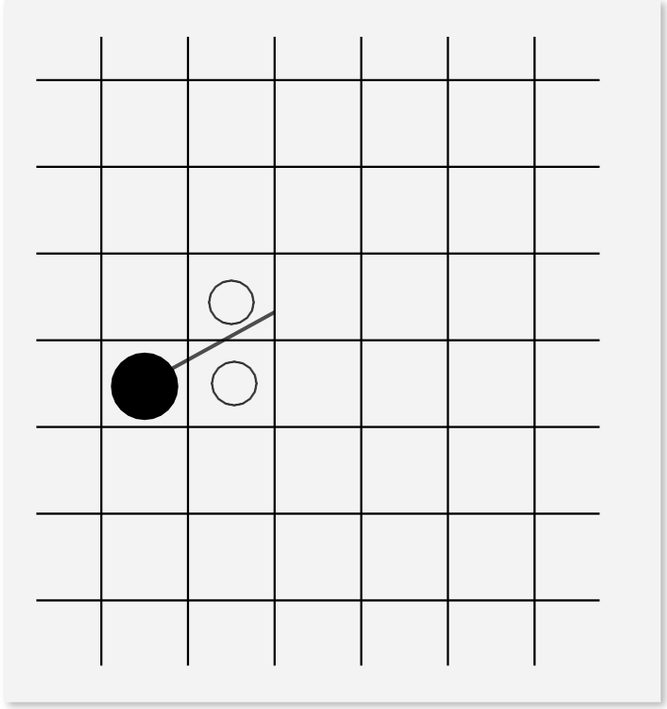
Bresenham's Line Algorithm

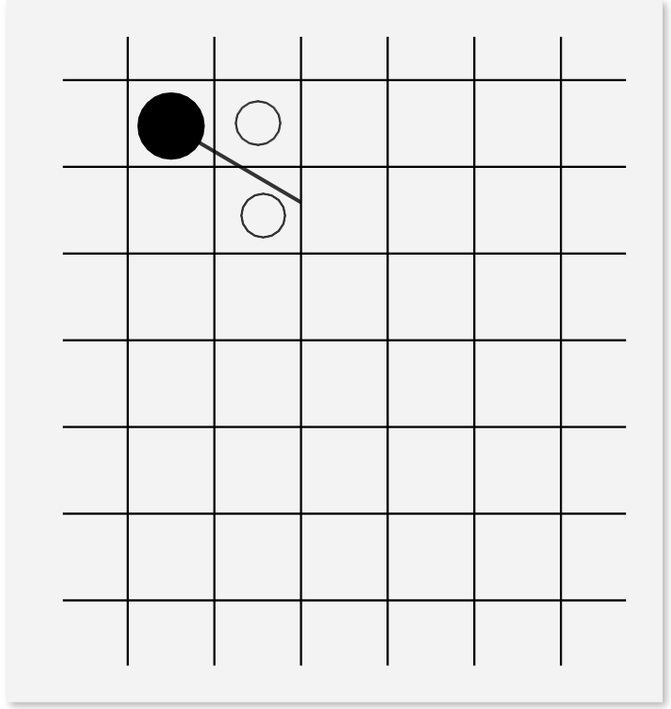
- Basic idea:

  Find the closest integer coordinates to the actual line path

  Use only integer arithmetic

  Compute iteratively the next point, $P_{i+1}=\mathbf{F}(P_i)$



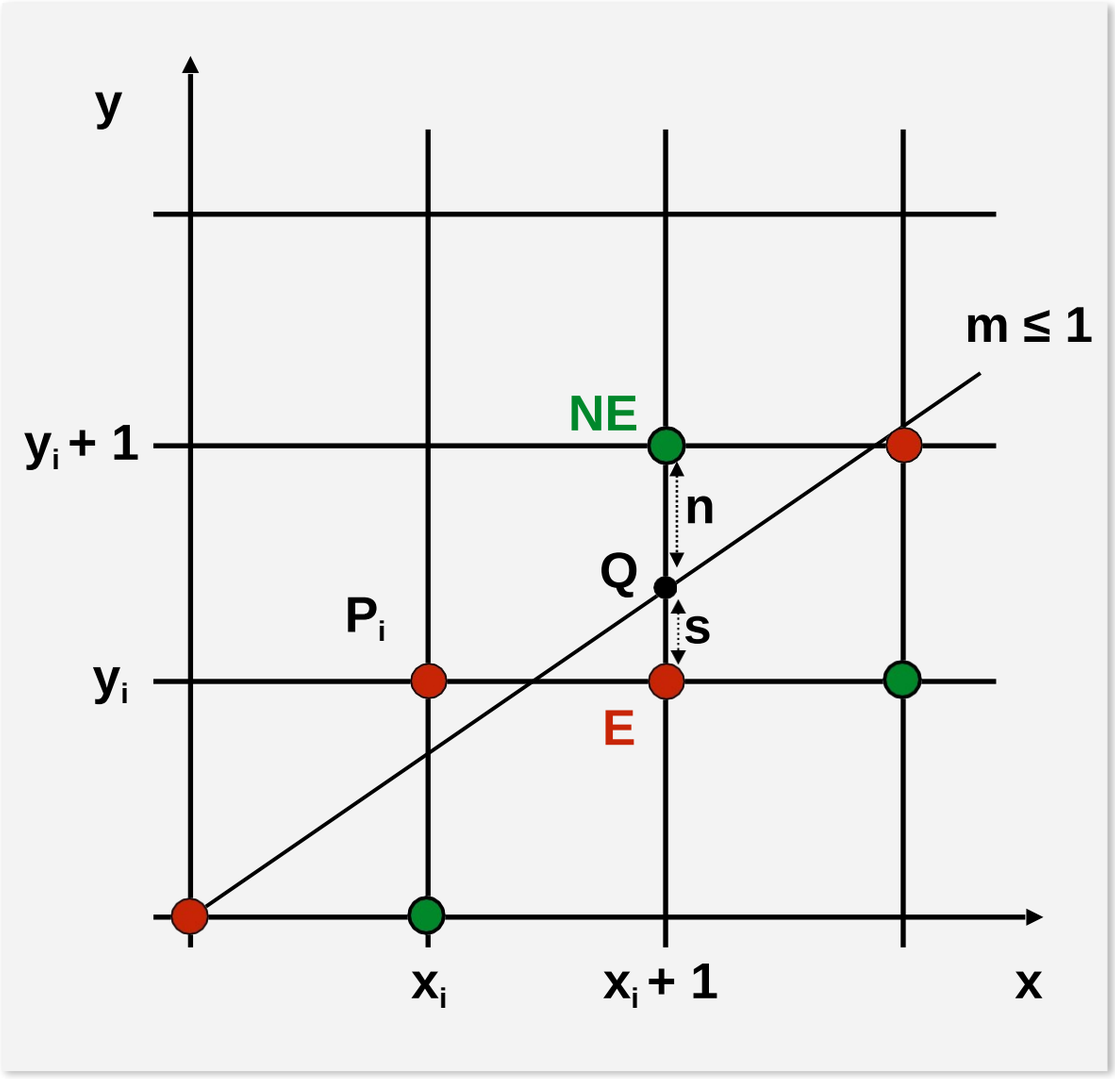


Bresenham method – basic idea

Basic approach:

Compute a decision variable $d_i=f(P_i, d_{i-1})$

Depending on the d_i value chose the next position **E** or **NE**



Bresenham algorithm

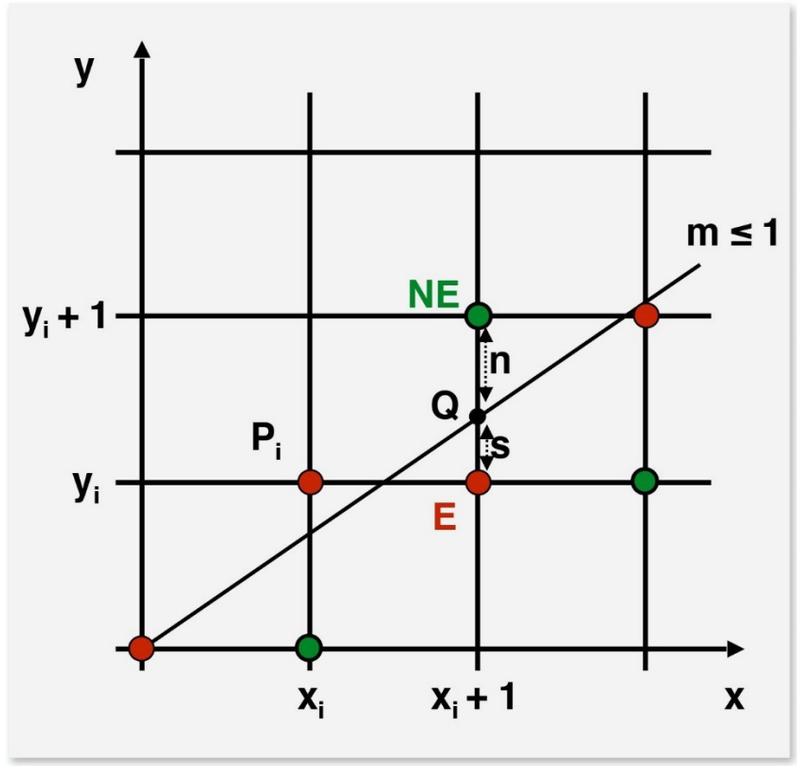
$$0 < m \leq 1,\ x_i < x_j,\ i < j$$

$$y = mx + b, m = \frac{\Delta y}{\Delta x}$$

$$s = y - y_i = m(x_i + 1) + b - y_i$$

$$n = y_i + 1 - y = y_i + 1 - m(x_i + 1) - b$$

$$k = s - n = 2m(x_i + 1) - 2y_i + 2b - 1$$



Consider: $d_i = k\Delta x = \Delta x(s - n)$

$$d_i = \Delta x(s - n) = 2\Delta y(x_i + 1) - 2\Delta x y_i + \Delta x(2b - 1)$$

$$d_i = 2\Delta y x_i - 2\Delta x y_i + const$$

if $d_i < 0$ choose $E(x_i + 1, y_i)$

if $d_i \geq 0$ choose $NE(x_i + 1, y_i + 1)$

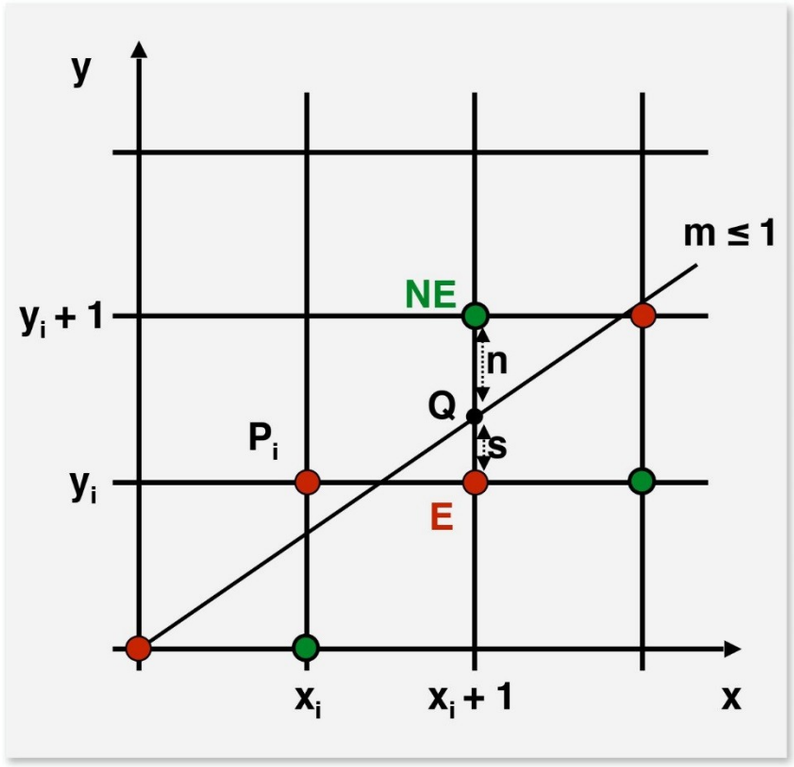
Bresenham algorithm

$$d_i = 2\Delta y x_i - 2\Delta x y_i + const$$

$$d_{i+1} = 2\Delta y x_{i+1} - 2\Delta x y_{i+1} + const$$

$$d_{i+1} - d_i = 2\Delta y(x_{i+1} - x_i) - 2\Delta x(y_{i+1} - y_i)$$

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i)$$



if $d_i < 0$ then $P_{i+1} = E$, and $y_{i+1} = y_i$
otherwise $P_{i+1} = NE$, and $y_{i+1} = y_i + 1$

$$d_{i+1} = d_i + 2\Delta y \text{ ,if } d_i < 0$$

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x \text{ ,otherwise}$$

The first value of the decision variable:

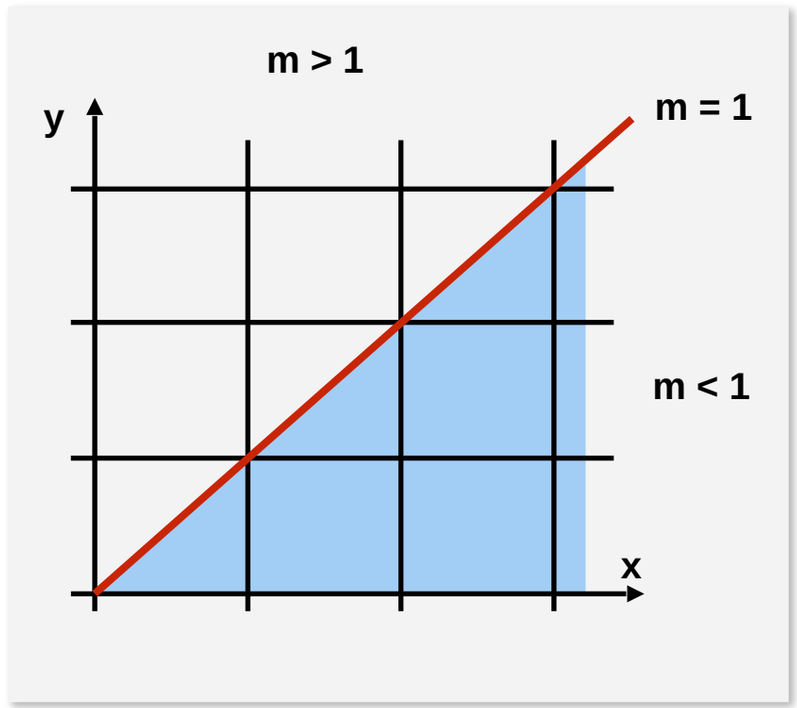
$$d_0 = 2\Delta y x_1 - 2\Delta x y_1 + [2\Delta y + \Delta x(2b - 1)]$$

$$y_1 = \frac{\Delta y}{\Delta x} x_1 + b \qquad \Delta x y_1 = \Delta y x_1 + b\Delta x \quad \Rightarrow \quad d_0 = 2\Delta y - \Delta x$$

$$\Delta x(2b - 1) = 2\Delta x y_1 - 2\Delta y x_1 - \Delta x$$

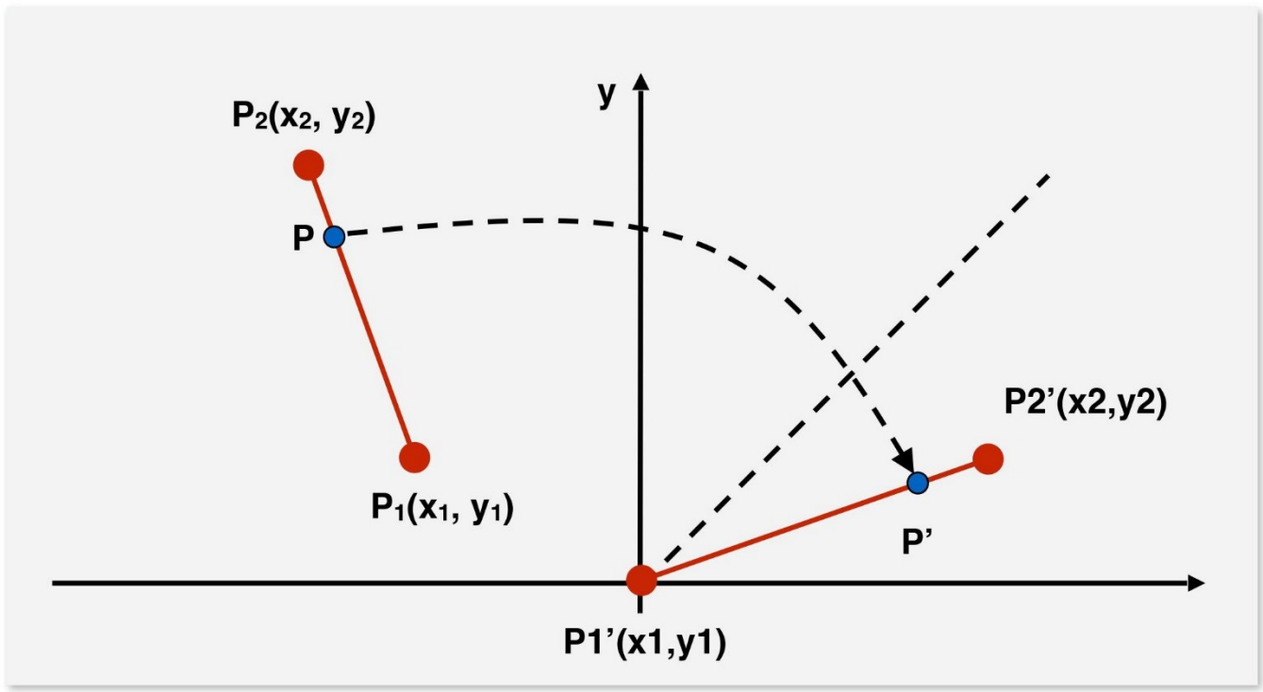
Bresenham algorithm

```
void LineBresenham(int x1, int y1, int x2, int y2){  int
   dx, dy, x, y, d, incrE, incrNE;

   dx = x2 - x1;
   dy = y2 - y1;
   d = 2 * dy - dx;
   incrE = 2 * dy;
   incrNE = 2 * (dy - dx);
   x = x1;
   y = y1;
   DisplayPixel(x, y);
    while(x  < x2){
       if(d  <= 0){
          d  = d +   incrE;
          x  = x +   1;
       }
       else{
          d  = d  +  incrNE;
          X  = x  +  1;
          y  = y  +  1;

       }
       DisplayPixel(x, y);
   }
}
```



General Bresenham's Line Algorithm

- Basic Bresenham line algorithm is given for line in the first octant
- To render a general line (P_1, P_2):
  - Transform the line (P_1, P_2) into the line (P_1', P_2') in the first octant
  - Compute the raster line (P_1', P_2')
  - For each point P'
    - Compute point P for the initial line
    - Render point P

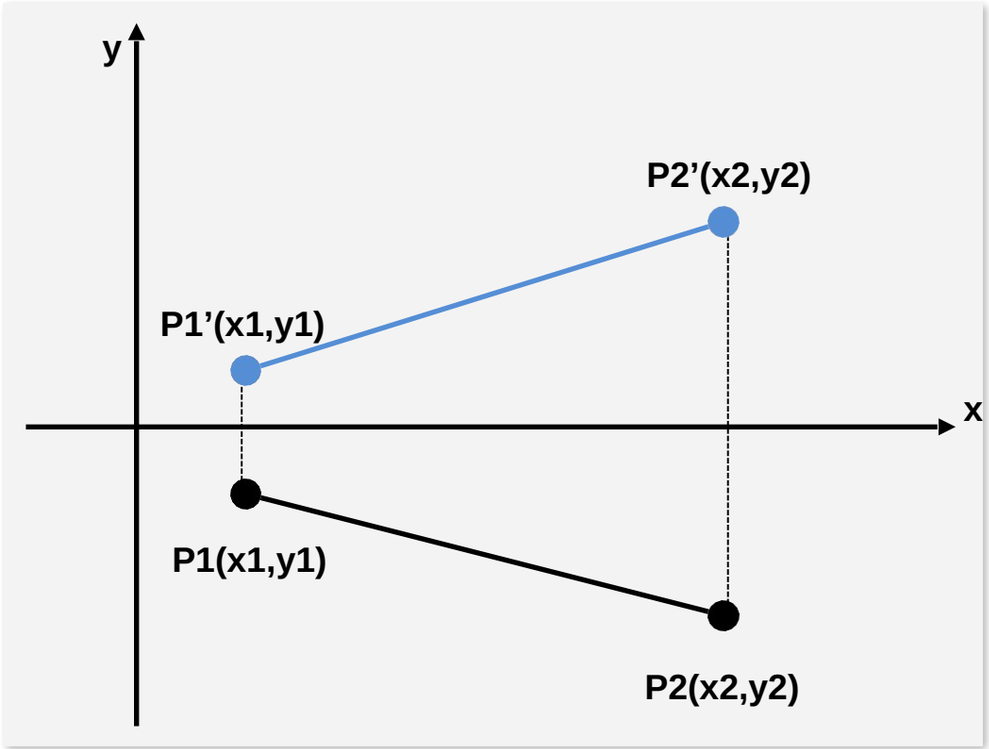


General Bresenham's Line Algorithm

$$0 > m > -1$$

Flip about x-axis

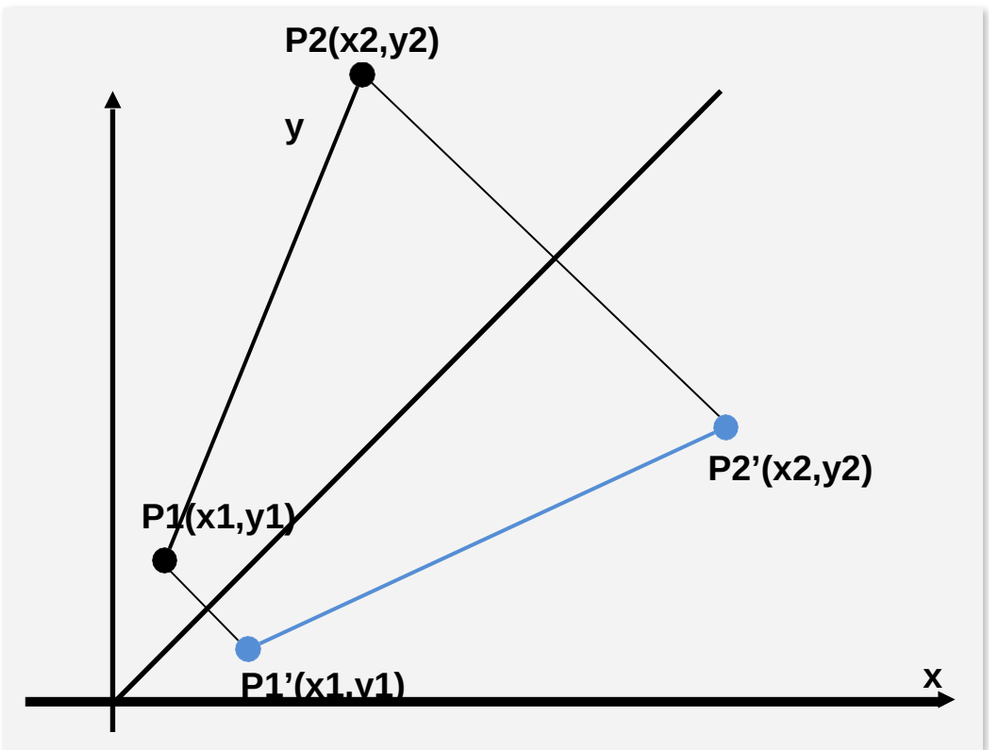
$$\begin{cases} x' = x \\ y' = -y \end{cases}$$



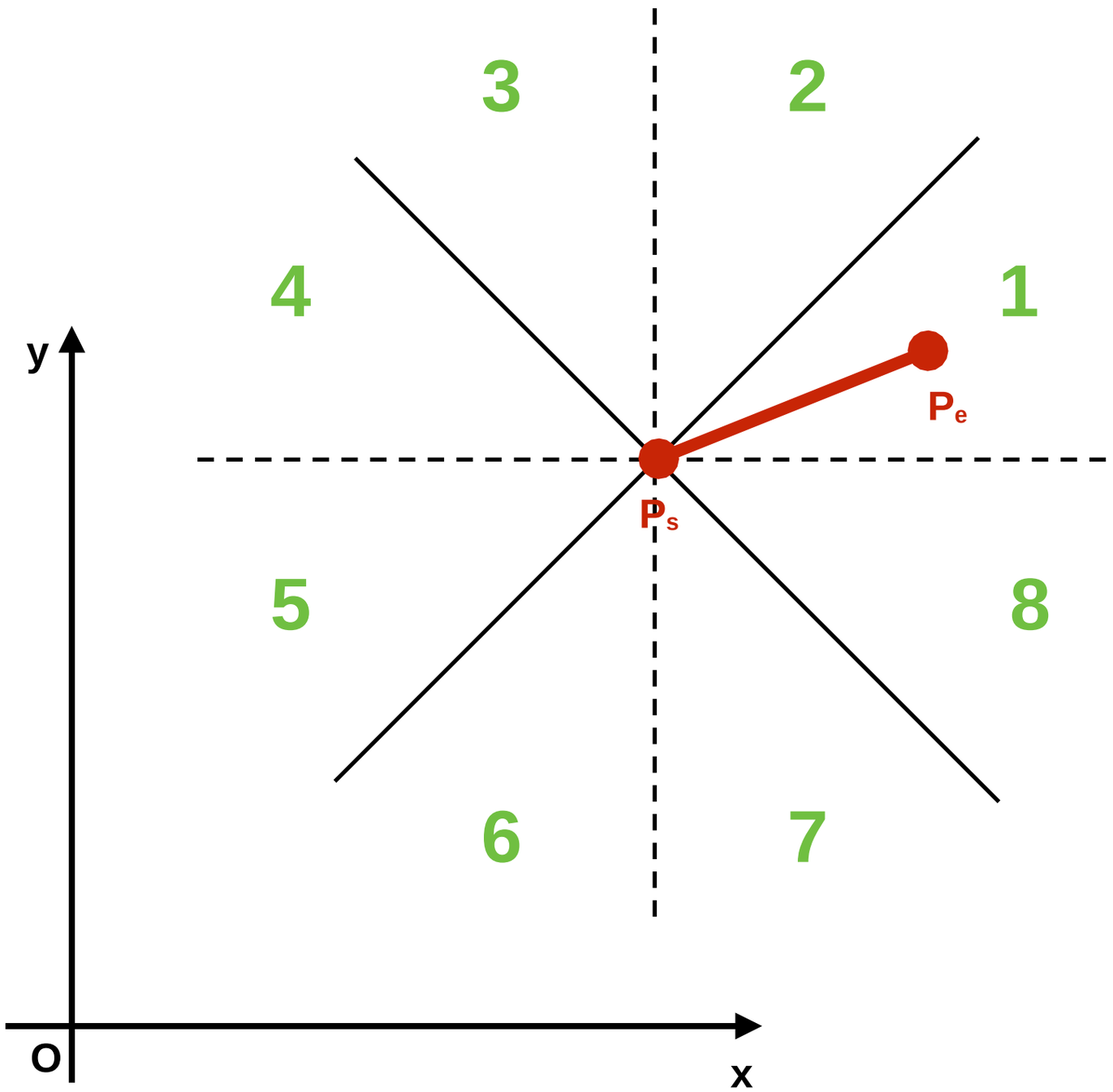
$$m > 1$$

Flip about y=x

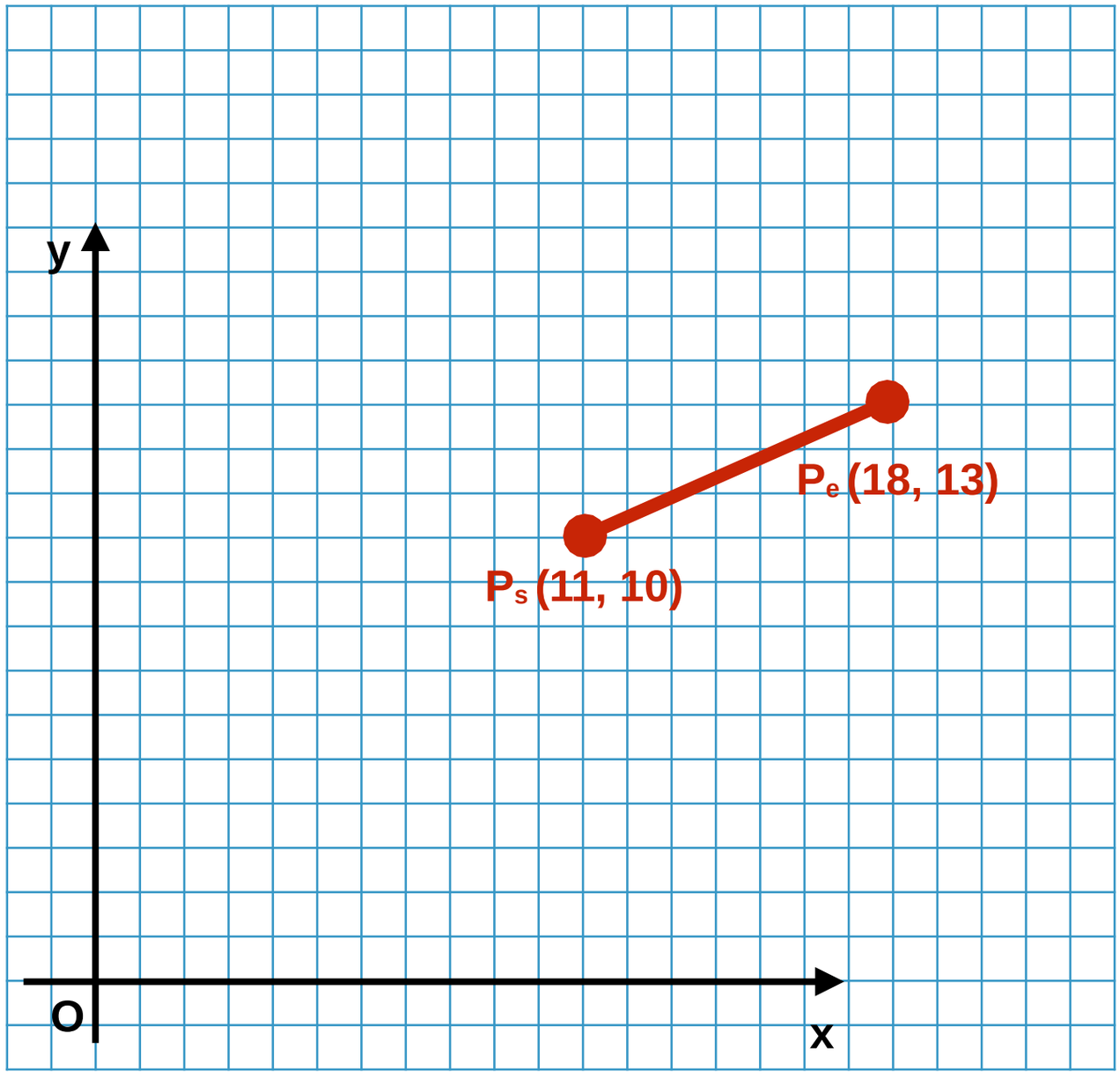
$$\begin{cases} x' = y \\ y' = x \end{cases}$$



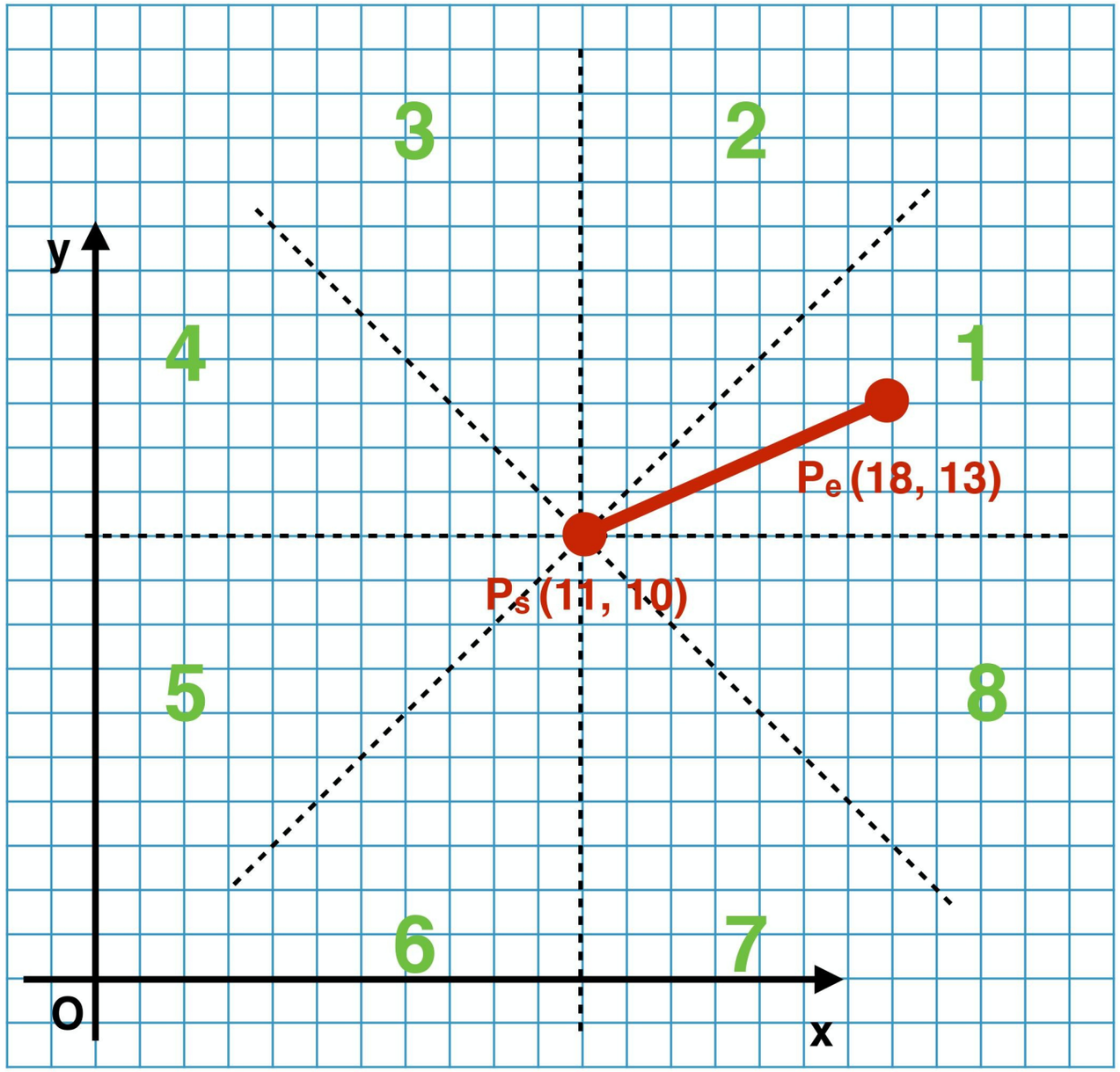
Line relative to octants



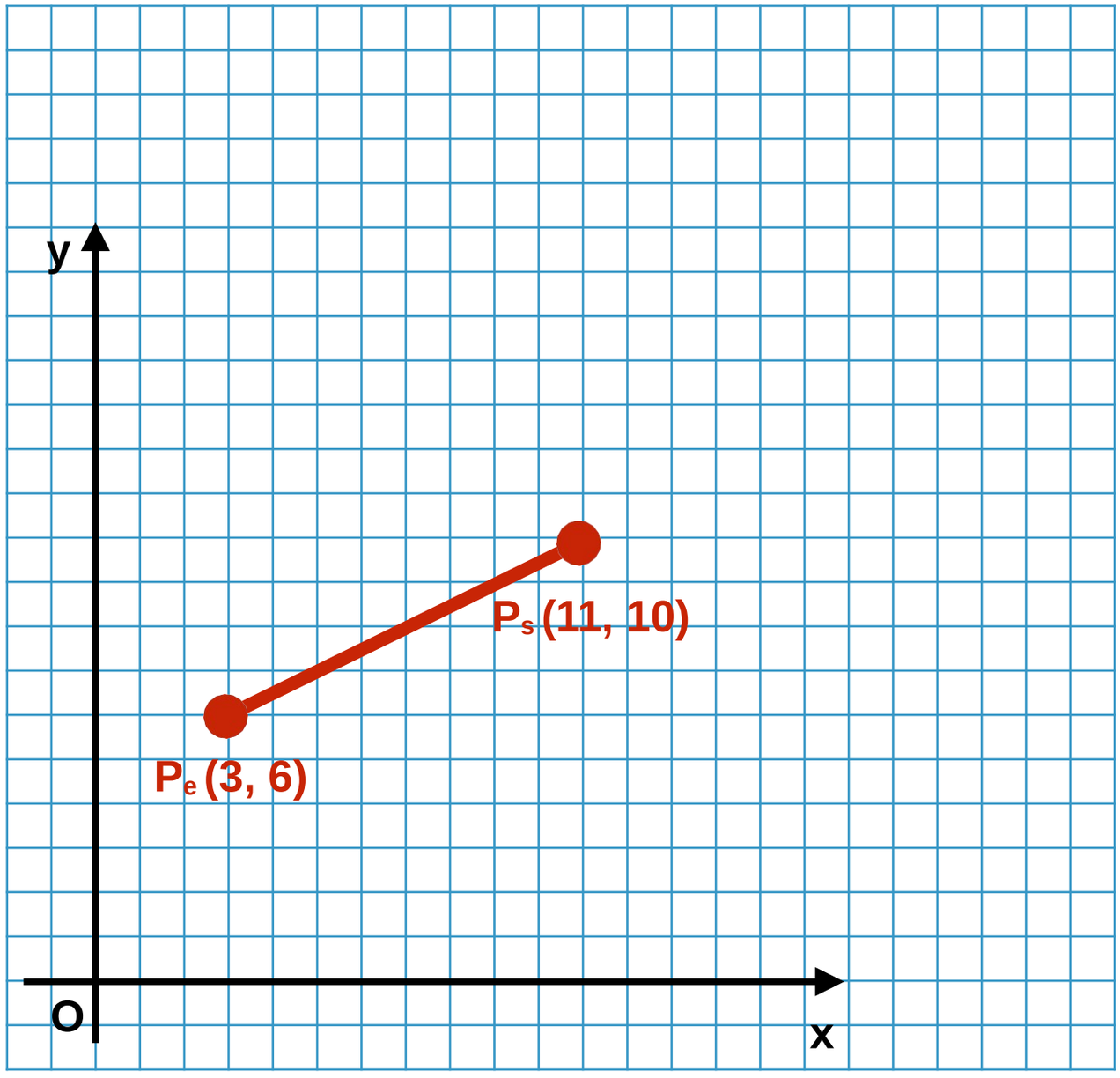
Line relative to octants



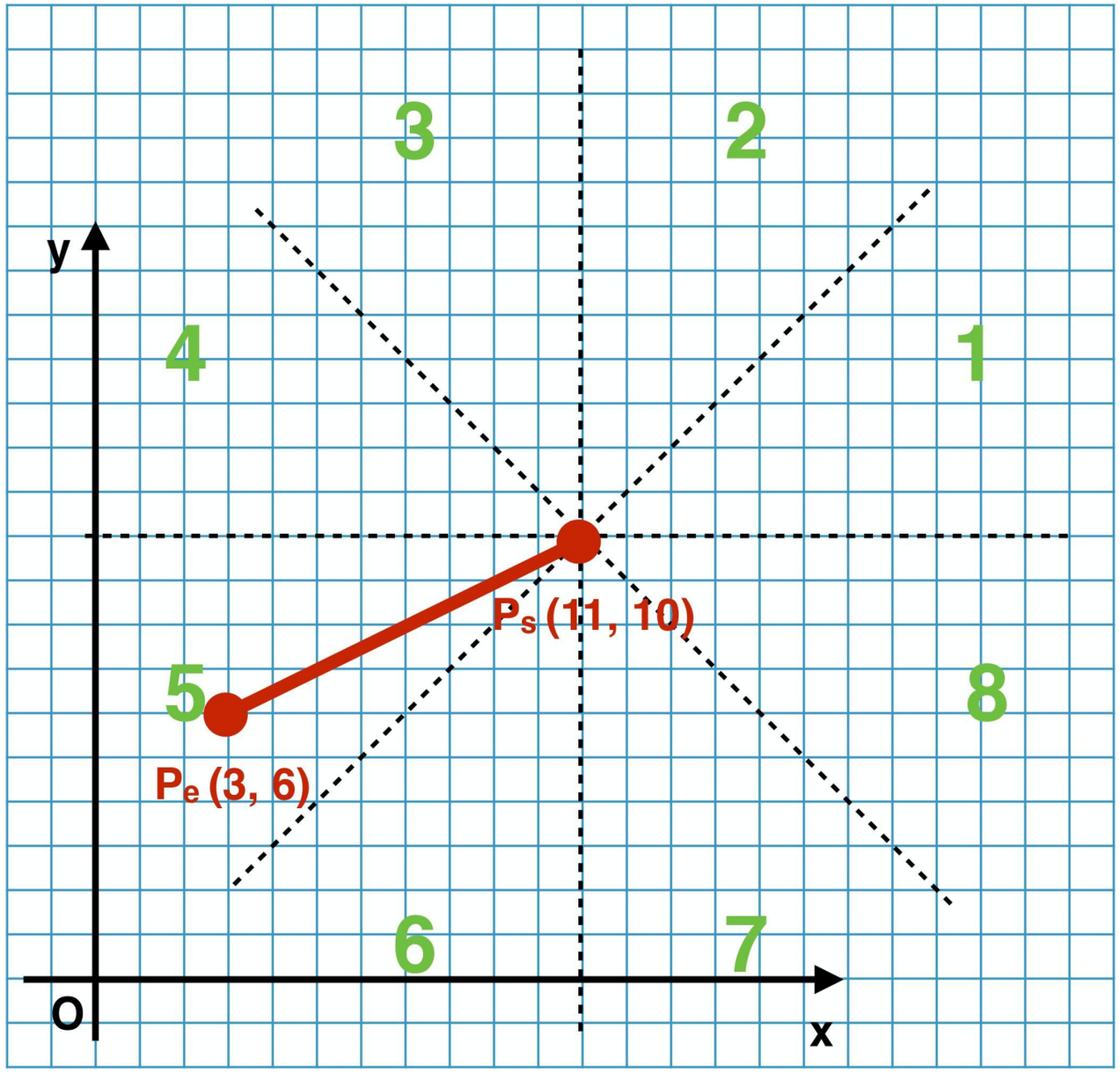
Line relative to octants



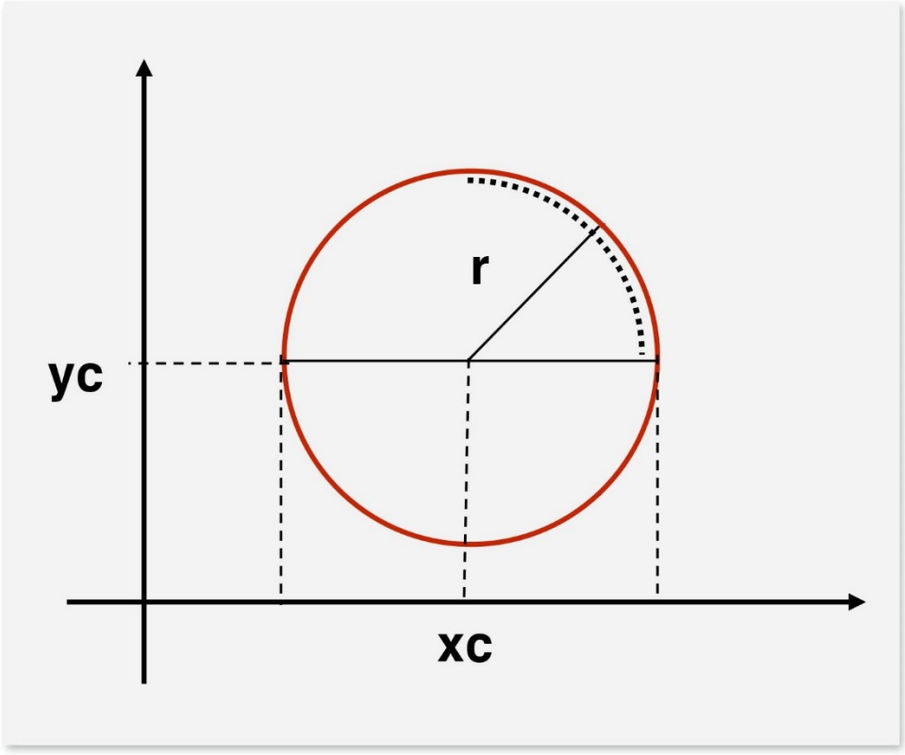
Line relative to octants

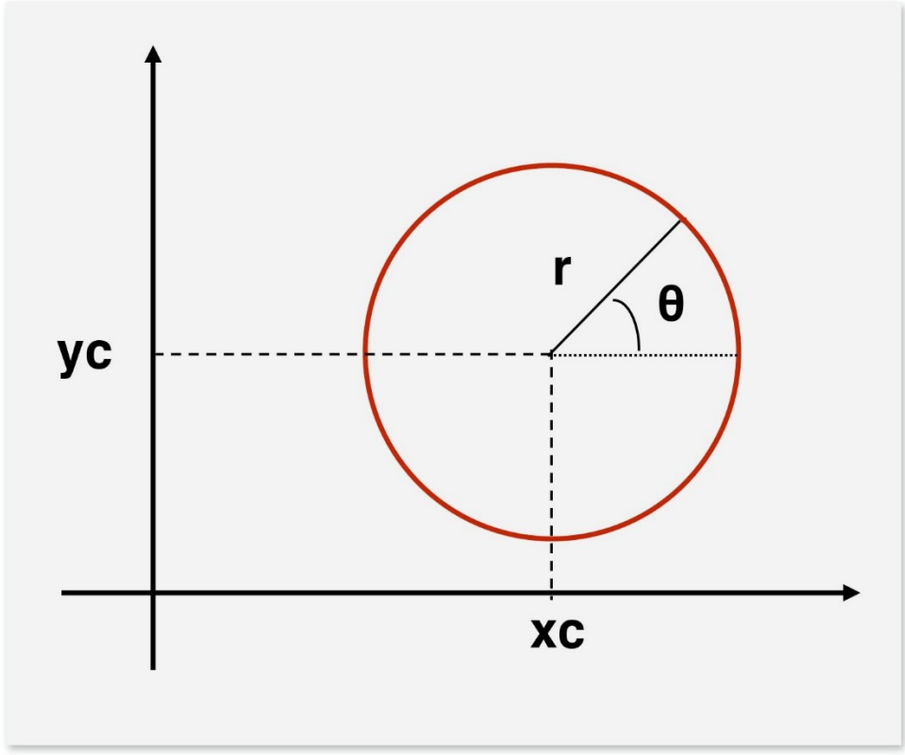


Line relative to octants



Circle Drawing





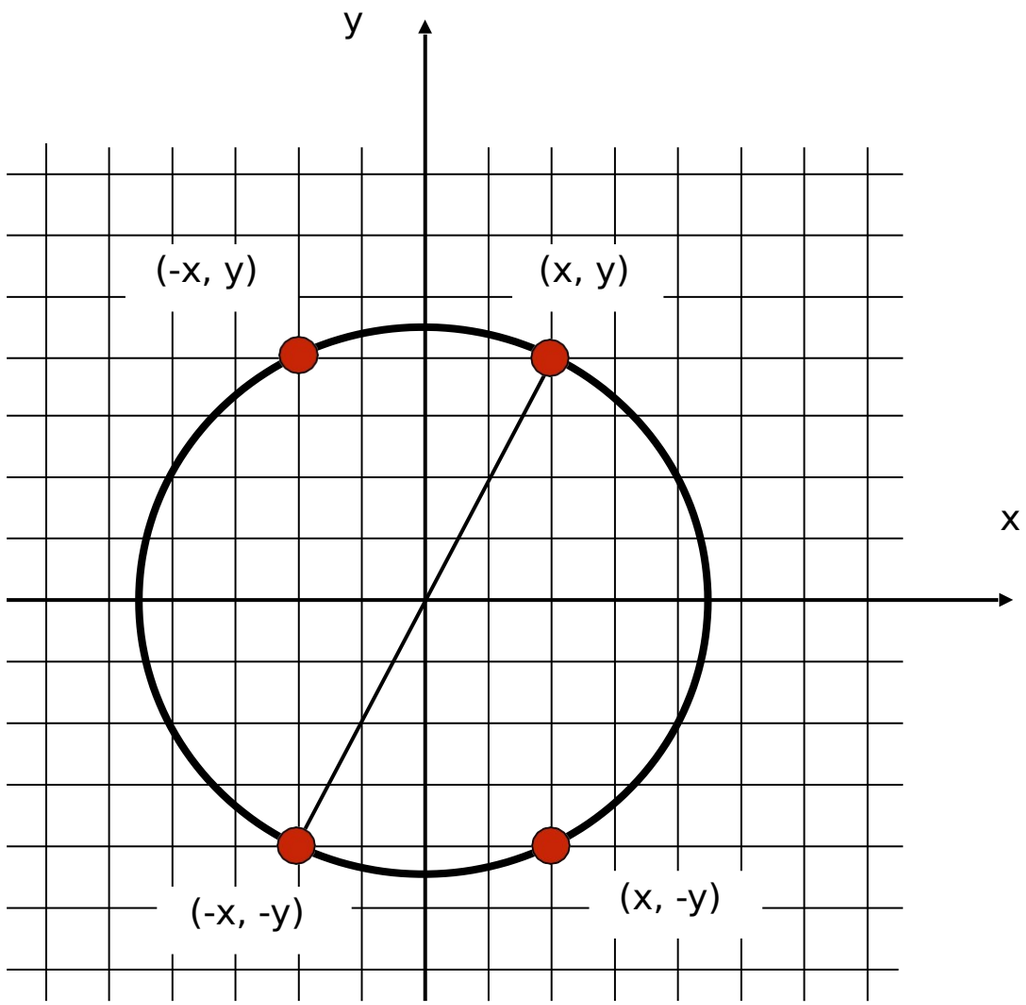
$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

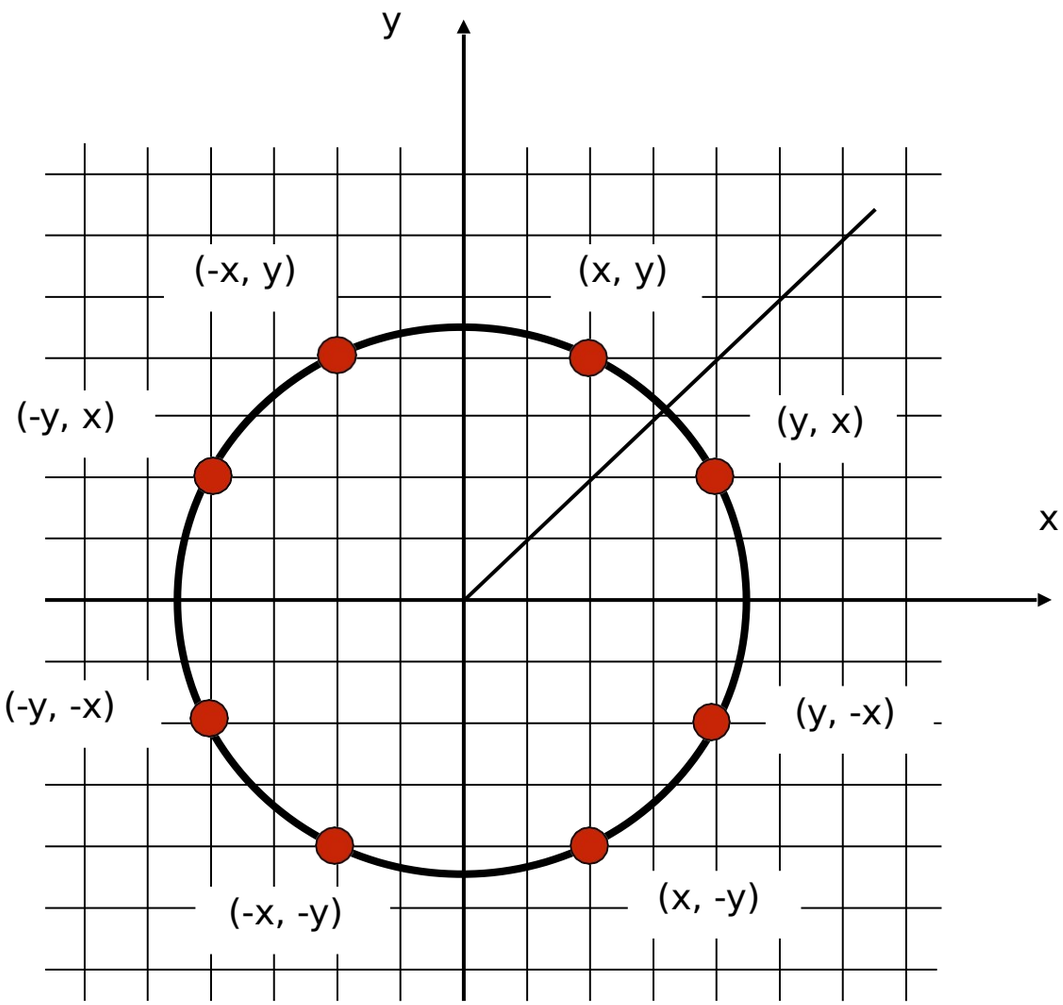
$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

$$x = x_c + r\cos\theta$$

$$y = y_c + r\sin\theta$$

Circle rendering by 4 and 8 way symmetry





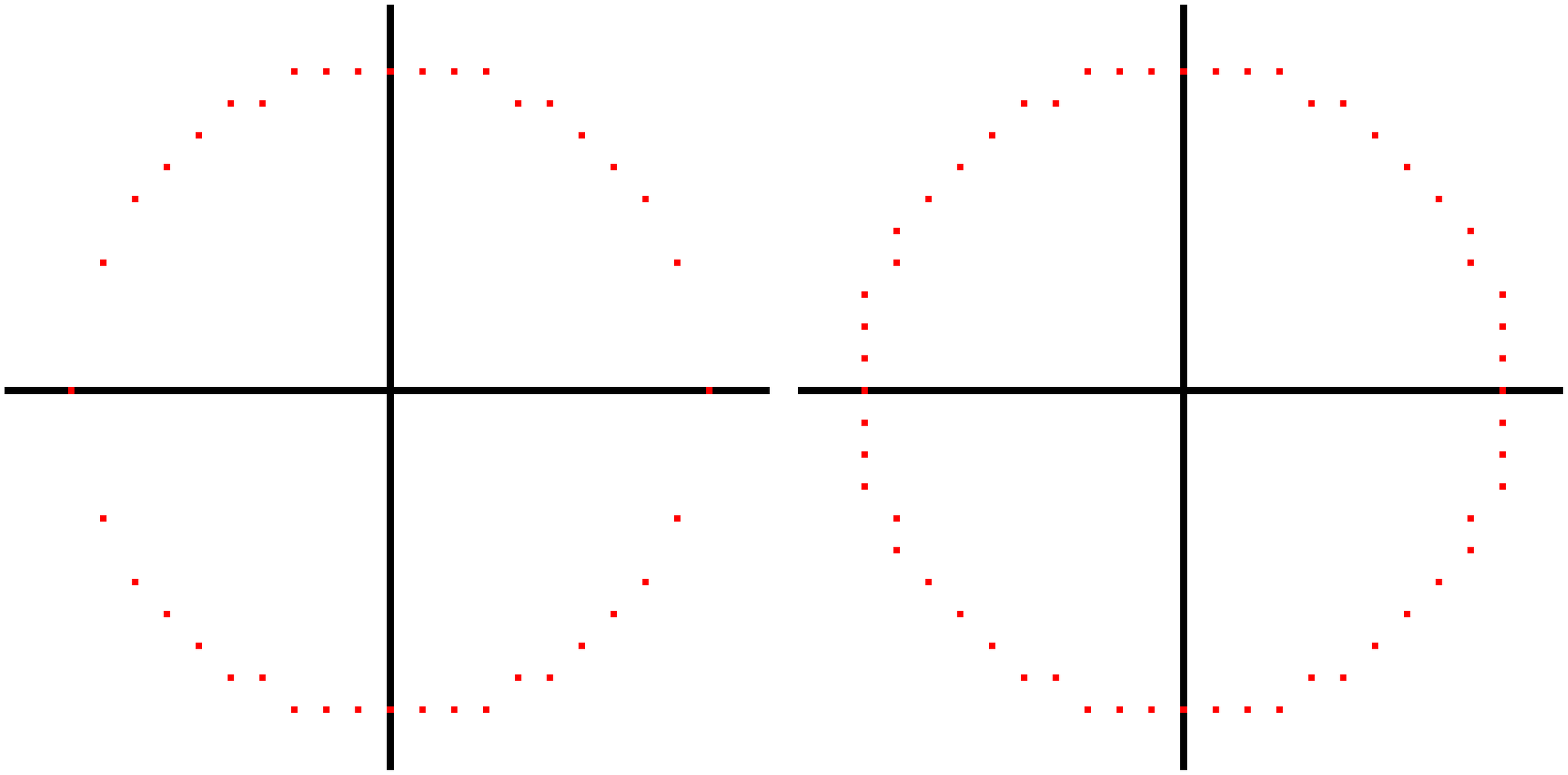
4 way symmetry algorithm

```
void CircleSym4(int xc, int yc, int r)
{  int x, y, r2;

   r2 = r * r;

    DisplayPixel(xc,   yc   +   r);
    DisplayPixel(xc,   yc   -   r);
    for(x = 1; x <= r; x++)
    {
       y = (int)(sqrt((float)(r2 - x * x)));
      DisplayPixel(xc  +   x,   yc  +  y);
      DisplayPixel(xc  +   x,   yc  -  y);
      DisplayPixel(xc  -   x,   yc  +  y);
      DisplayPixel(xc  -   x,   yc  -  y);
    }
}
```

8 way symmetry algorithm

```
void CircleSym8(int xc,int yc,int r){
  int x,y, r2;

 r2 = r * r;

 DisplayPixel(xc, yc + r);
 DisplayPixel(xc, yc - r);
 DisplayPixel(xc + r, yc);
 DisplayPixel(xc - r, yc);

 x = 1;
 y = (int)(sqrt((float)(r2 - x * x)));

 while(x < y){
     DisplayPixel(xc + x, yc + y);
     DisplayPixel(xc + x, yc - y);
     DisplayPixel(xc - x, yc + y);
     DisplayPixel(xc - x, yc - y);
     DisplayPixel(xc + y, yc + x);
     DisplayPixel(xc + y, yc - x);
     DisplayPixel(xc - y, yc + x);
     DisplayPixel(xc - y, yc - x);
     x++;
     y=(int)(sqrt((float)(r2-x*x)));
}


if(x == y)
{
    DisplayPixel(xc + x, yc + y);
    DisplayPixel(xc + x, yc - y);
    DisplayPixel(xc - x, yc + y);
    DisplayPixel(xc - x, yc - y);
  }
}
```

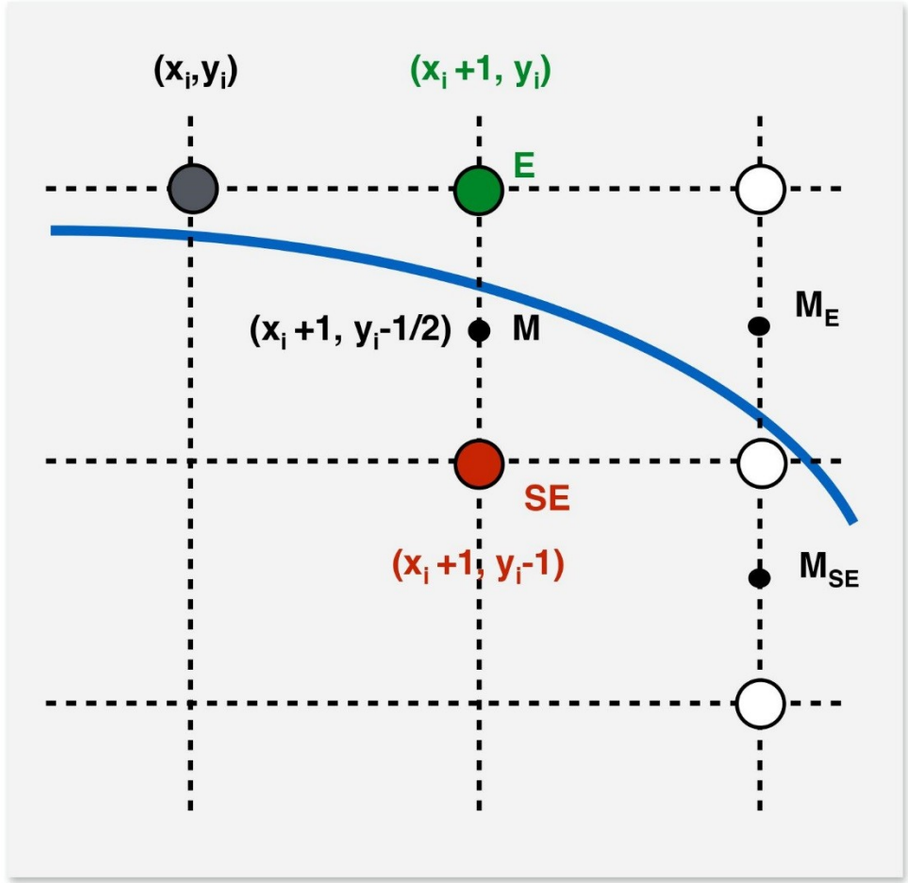
Circle rendering by 4 and 8 way symmetry



Midpoint circle algorithm

- Circle equation: $x^2 + y^2 = R^2$

- Considering $\mathbf{F}(x, y) = x^2 + y^2 - R^2$
  - **F(x,y) > 0** for **P(x,y) outside** of the circle
  - **F(x,y) = 0** for **P(x,y) on** the circle
  - **F(x,y) < 0** for **P(x,y) inside** of the circle



- Therefore:
  - if **F(M) < 0**, choose the next point **P(x_{i+1},y_{i+1}) = E(x_{i+1},y_i)**
  - otherwise, choose the next point **P(x_{i+1},y_{i+1}) = SE(x_{i+1},y_{i-1})**

Midpoint circle algorithm - computation

- Let us consider the decision variable:

$$d_i = \mathbf{F}(x_i + 1, y_i - \frac{1}{2}) = (x_i + 1)^2 + (y_i - \frac{1}{2})^2 - R^2$$

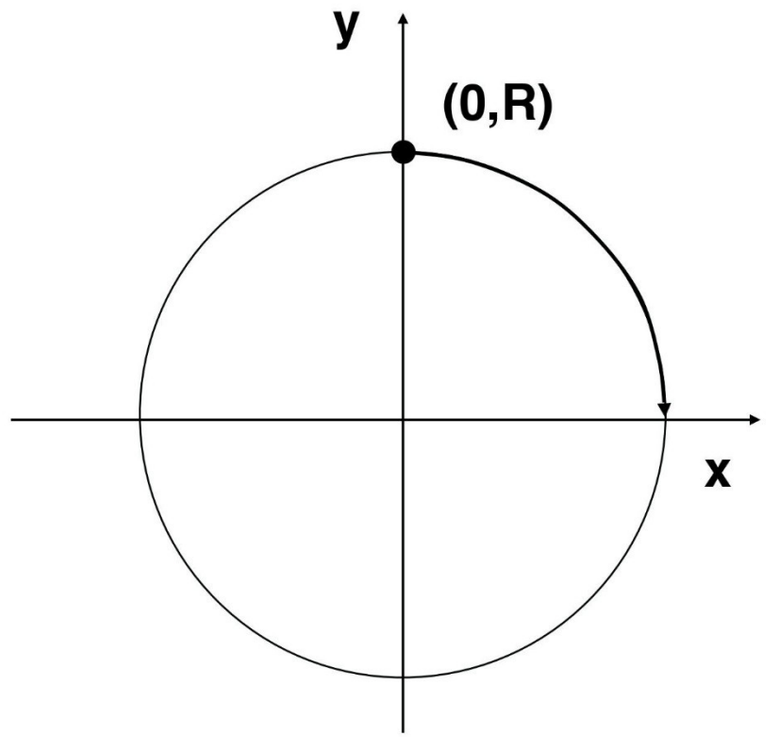
$$d_{i+1} = \mathbf{F}(x_{i+1} + 1, y_{i+1} - \frac{1}{2}) = (x_i + 2)^2 + (y_{i+1} - \frac{1}{2})^2 - R^2$$

- Where:

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

$$y_{i+1} = y_i, \text{ if } d_i < 0$$

$$y_{i+1} = y_i - 1, \text{ if } d_i \geq 0$$



$$d_{i+1} = d_i + 2x_i + 3, \text{ if } d_i < 0$$
$$d_{i+1} = d_i + 2(x_i - y_i) + 5, \text{ if } d_i \geq 0$$

- The starting point **$(x_0, y_0) = (0, R)$**

$$d_0 = \mathbf{F}(x_0 + 1, y_0 - \frac{1}{2}) = (0 + 1)^2 + (R - \frac{1}{2})^2 - R^2 = \frac{5}{4} - R$$

Midpoint circle algorithm - summary

- Choose the next point **P(x_{i+1},y_{i+1})**:

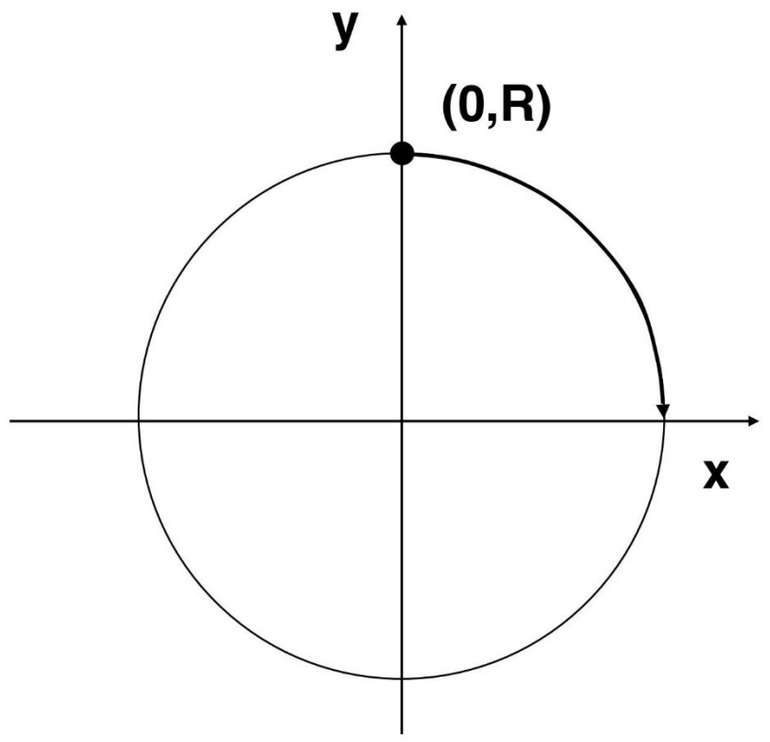
$$x_{i+1} = x_i + 1$$

$$y_{i+1} = y_i, \text{ if } d_i < 0$$

$$y_{i+1} = y_i - 1, \text{ if } d_i \geq 0$$

$$d_{i+1} = d_i + 2x_i + 3, \text{ if } d_i < 0$$

$$d_{i+1} = d_i + 2(x_i - y_i) + 5, \text{ if } d_i \geq 0$$



- The initial decision variable value:

$$d_0 = \mathbf{F}(0, R) = \frac{5}{4} - R$$

Midpoint circle algorithm

```
void MidpointCircle(int xc, int yc, int r)
{
   int x, y;  float d;


    x  =  0;
    y  =  r;
    d  =  5 /  4  - r;
   DisplayPixel(xc, yc + r);
   DisplayPixel(xc, yc - r);
   DisplayPixel(xc + r, yc);
   DisplayPixel(xc - r, yc);
   while(y > x){
      if(d < 0){
         d = d + 2 * x + 3;   x = x + 1;
      }
      else{
         d = d + 2*(x - y) +
         x = x +  1;
         y = y -  1;
      }
      DisplayCirclePoints(xc, yc, x, y);
   }
}
```

```
                                  5;
```

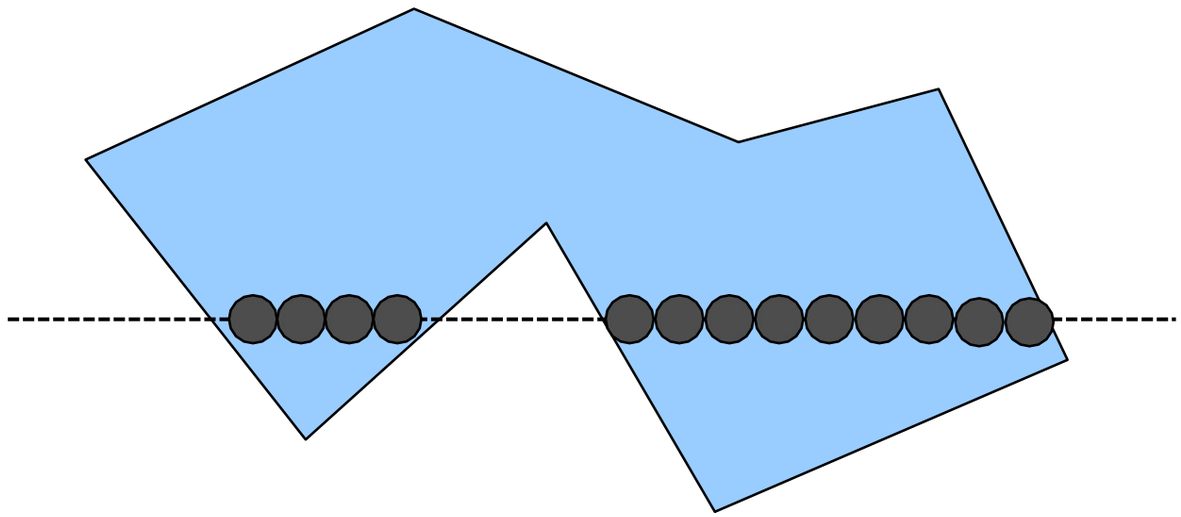
```
void DisplayCirclePoints(int xc, int yc,
                         int x, int y){

   DisplayPixel(xc  +  x,  yc  +  y);

   DisplayPixel(xc  +  x,  yc  -  y);

   DisplayPixel(xc  -  x,  yc  +  y);

   DisplayPixel(xc  -  x,  yc  -  y);

   DisplayPixel(xc  +  y,  yc  +  x);

   DisplayPixel(xc  +  y,  yc  -  x);

   DisplayPixel(xc  -  y,  yc  +  x);

   DisplayPixel(xc  -  y,  yc  -  x);

}
```

Polygon rasterization

- **Polygon scan-conversion**

  sweep the polygon by the scan line

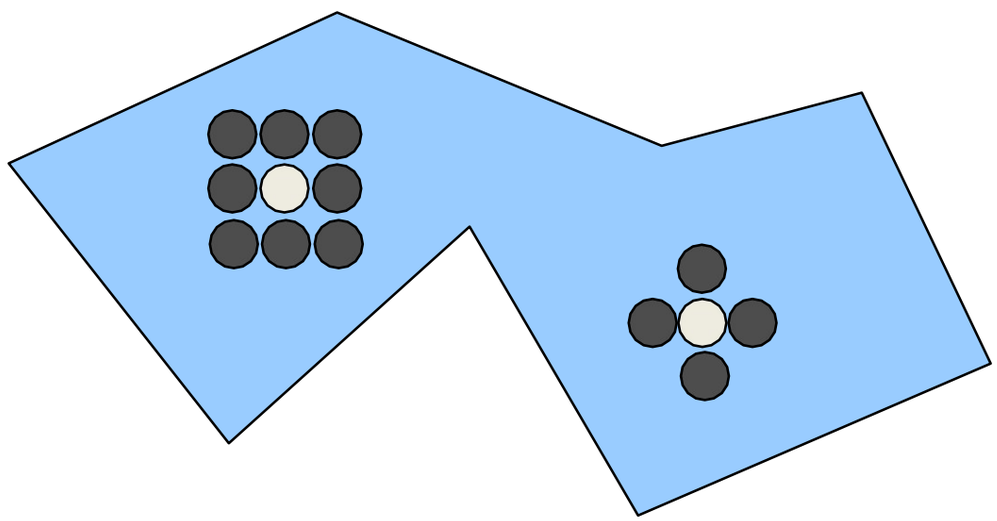
  set the pixels whose center is inside the polygon for each scan line



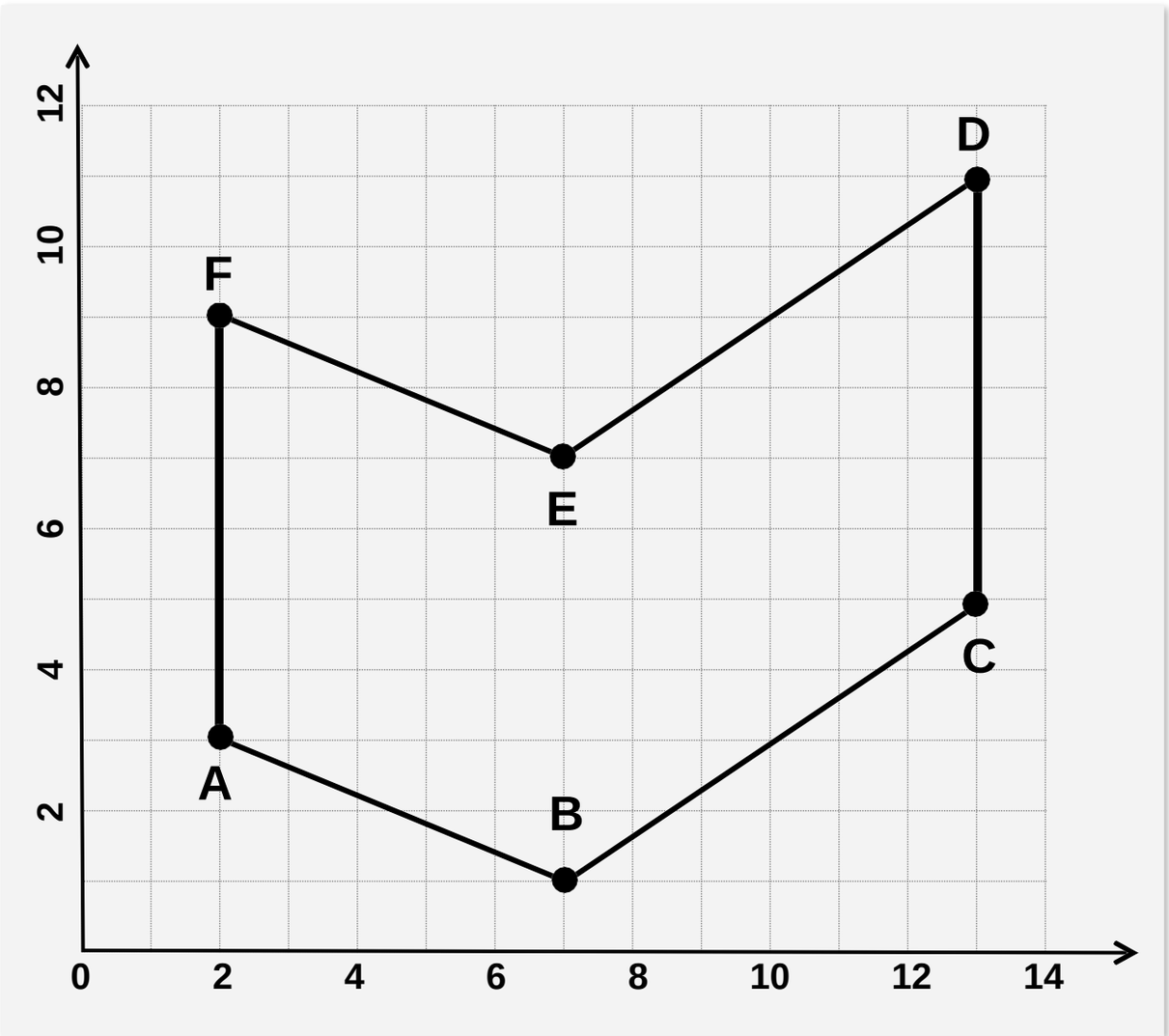
- **Polygon fill (Seed filling)**

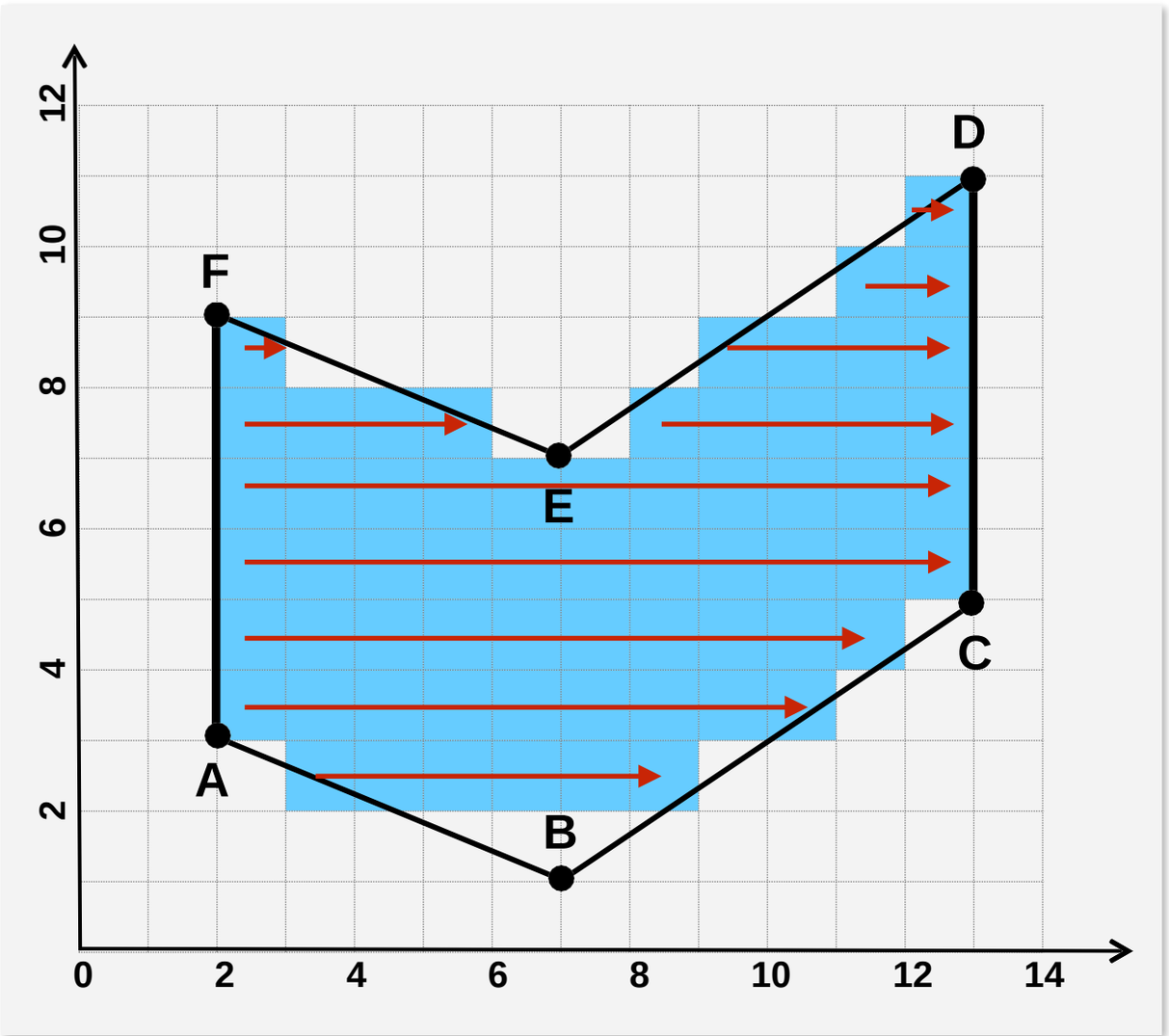
  choose a pixel inside the polygon

  grow until the whole polygon is filled



Scan-line





Scan-line algorithm

- **Edge table** - **ET**

  Contains all edges (sorted by their smaller y coordinate)

  Each entry contains:

  - $\mathbf{Y_{max}}$ coordinate of the edge
  - $\mathbf{X_{low}}$ coordinate at the bottom endpoint
    - The **X increment**

$\mathbf{Y_{max}}$	$\mathbf{X_{low}}$	**1/m**	

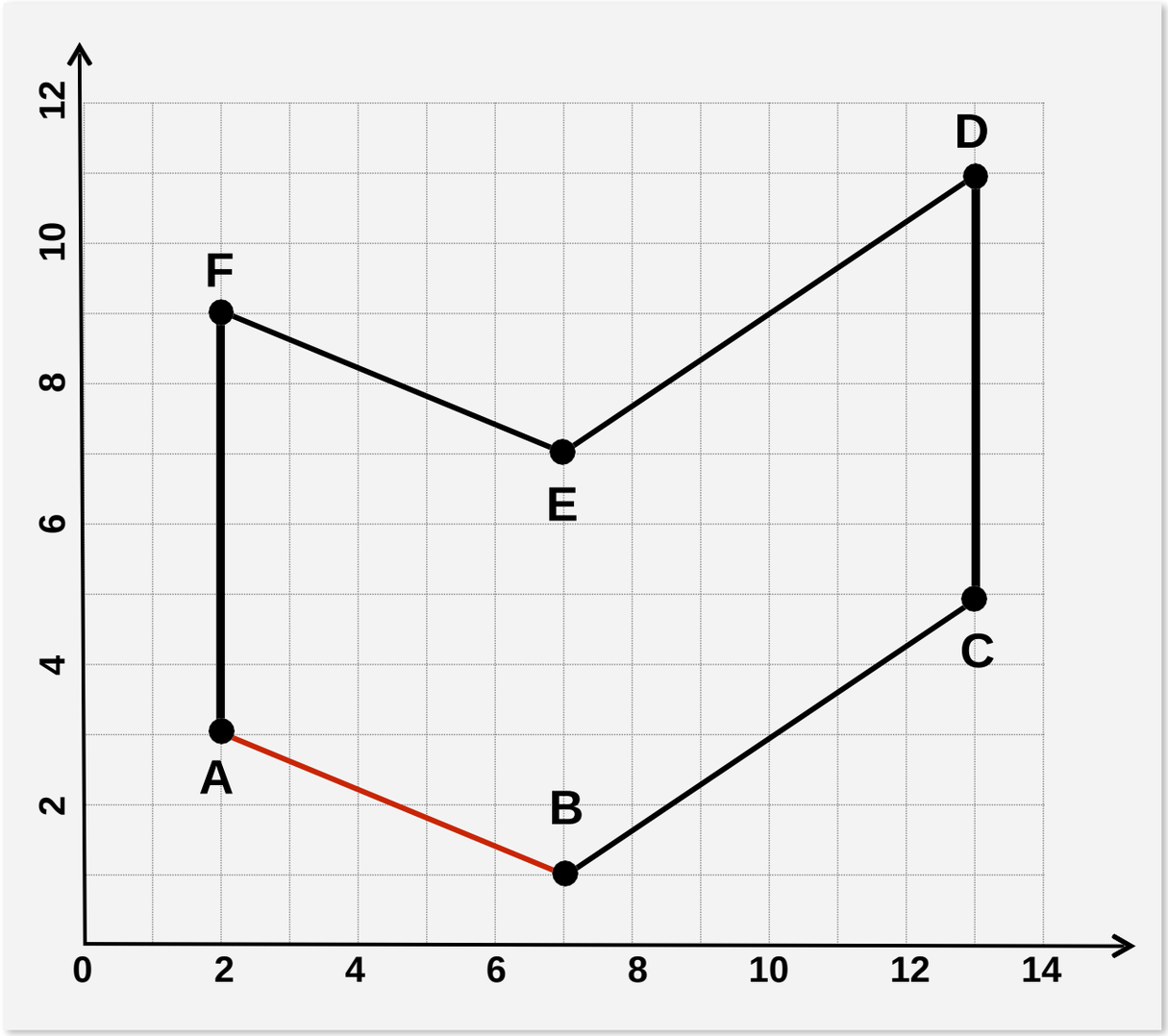
- **Active-edge table** - **AET**

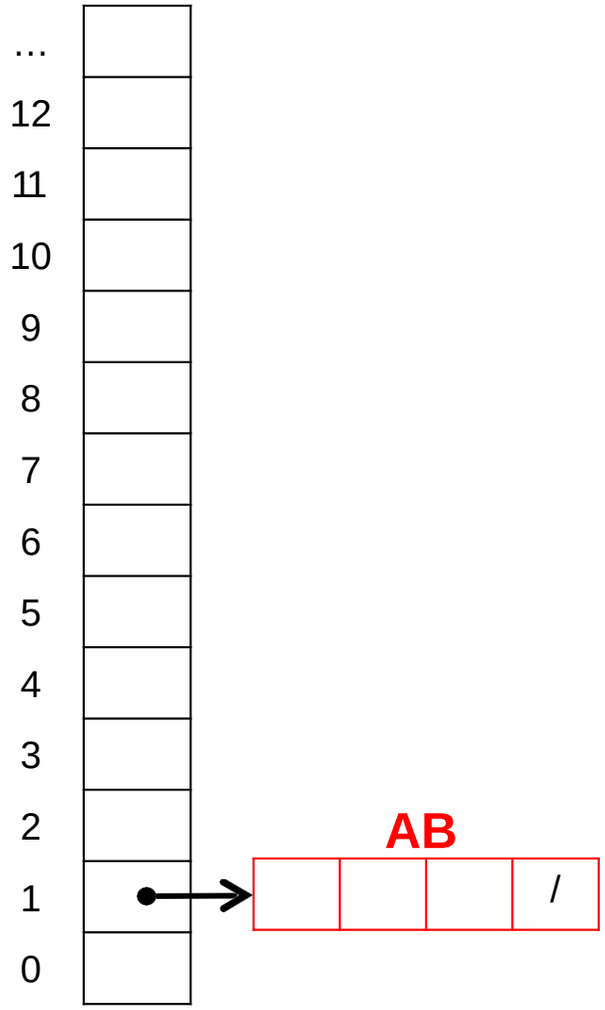
  The edges are sorted on their x intersection values

  Fill the spans defined by pairs

  Need to be updated for each scan line

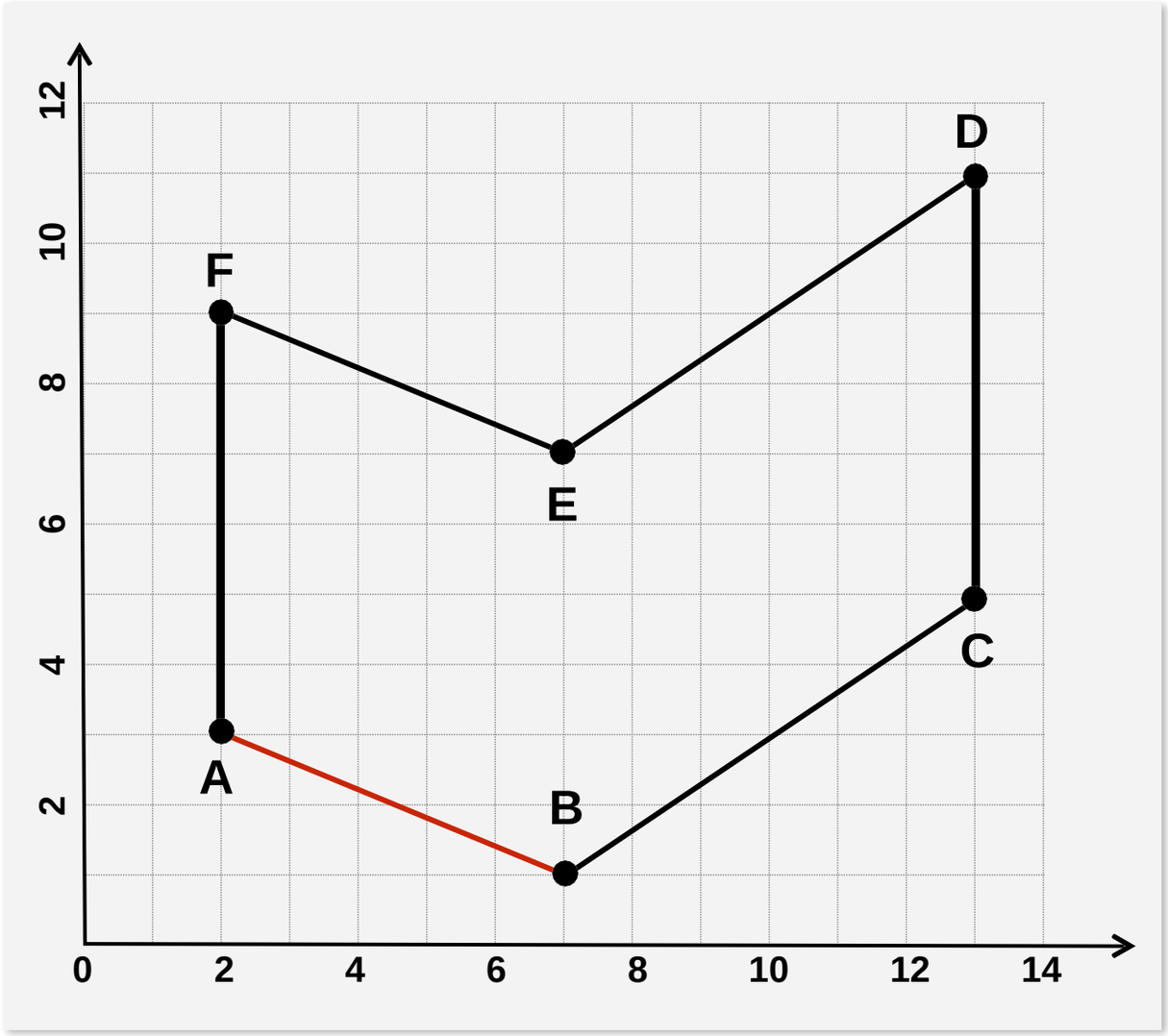
Scan-line algorithm

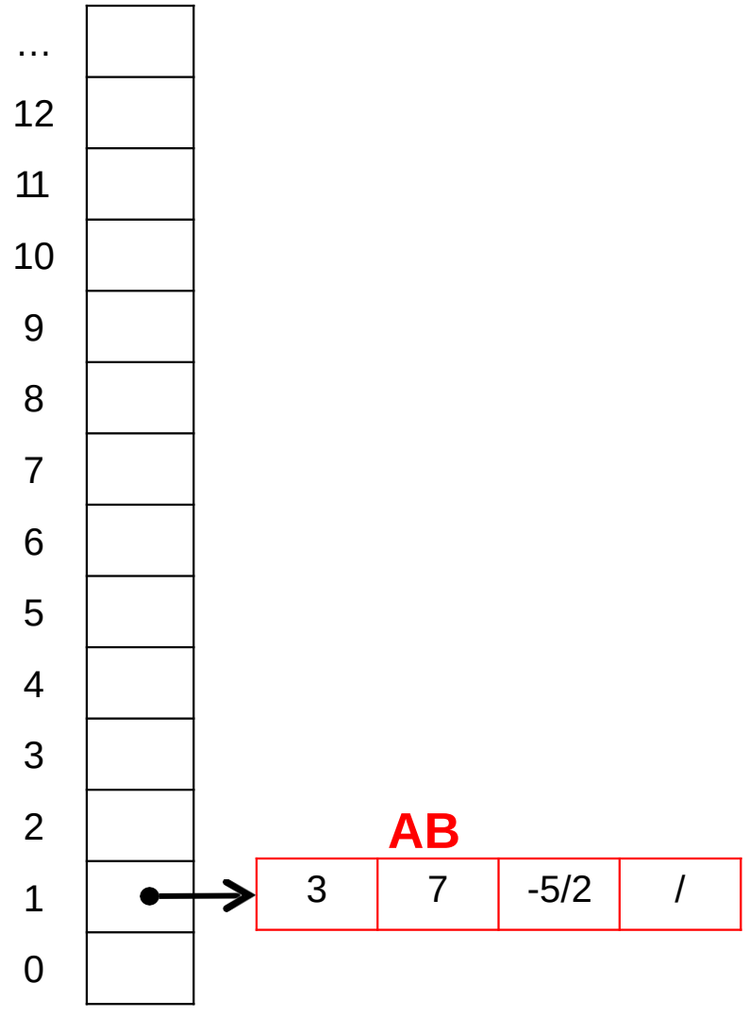




Y_{max}	X_{low}	1/m	

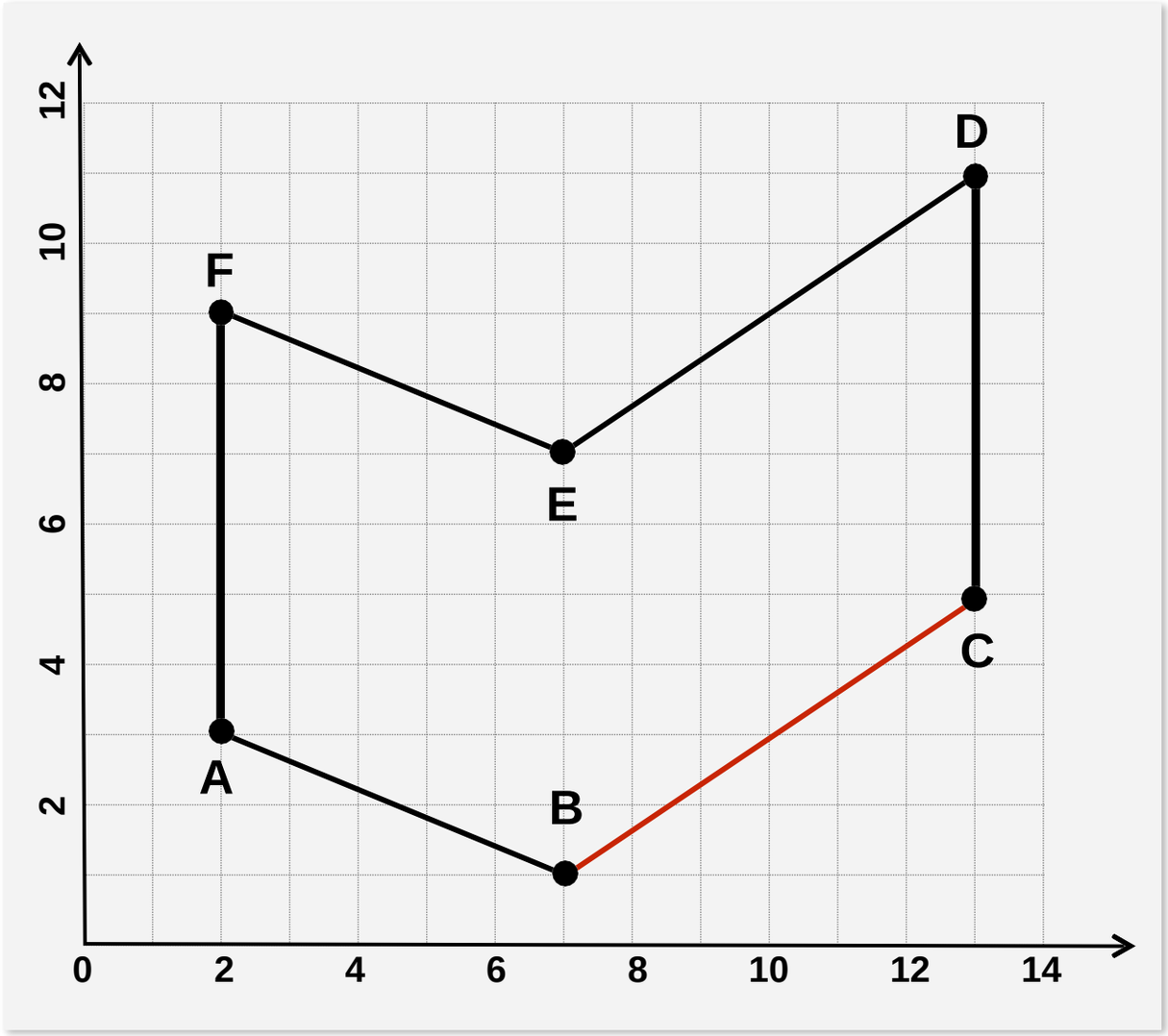
Scan-line algorithm

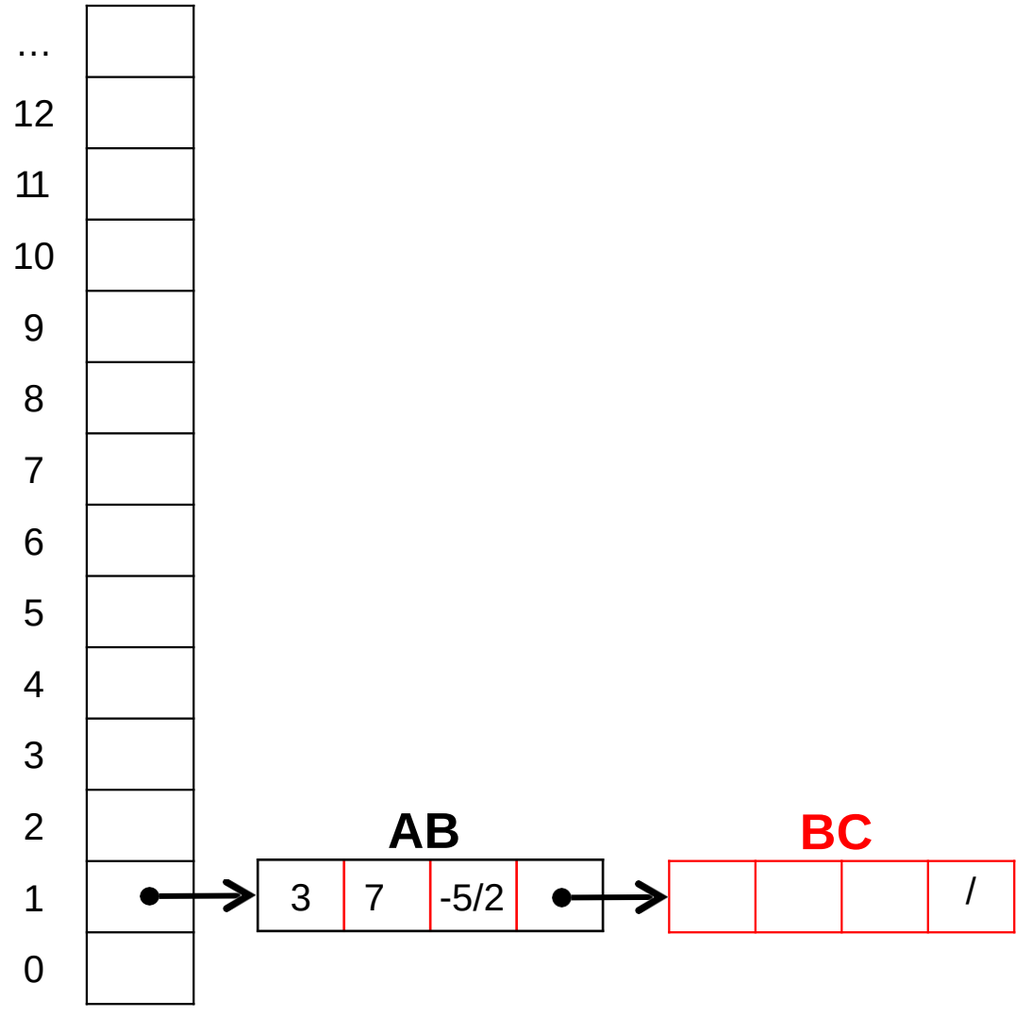




Y_{max}	X_{low}	1/m	

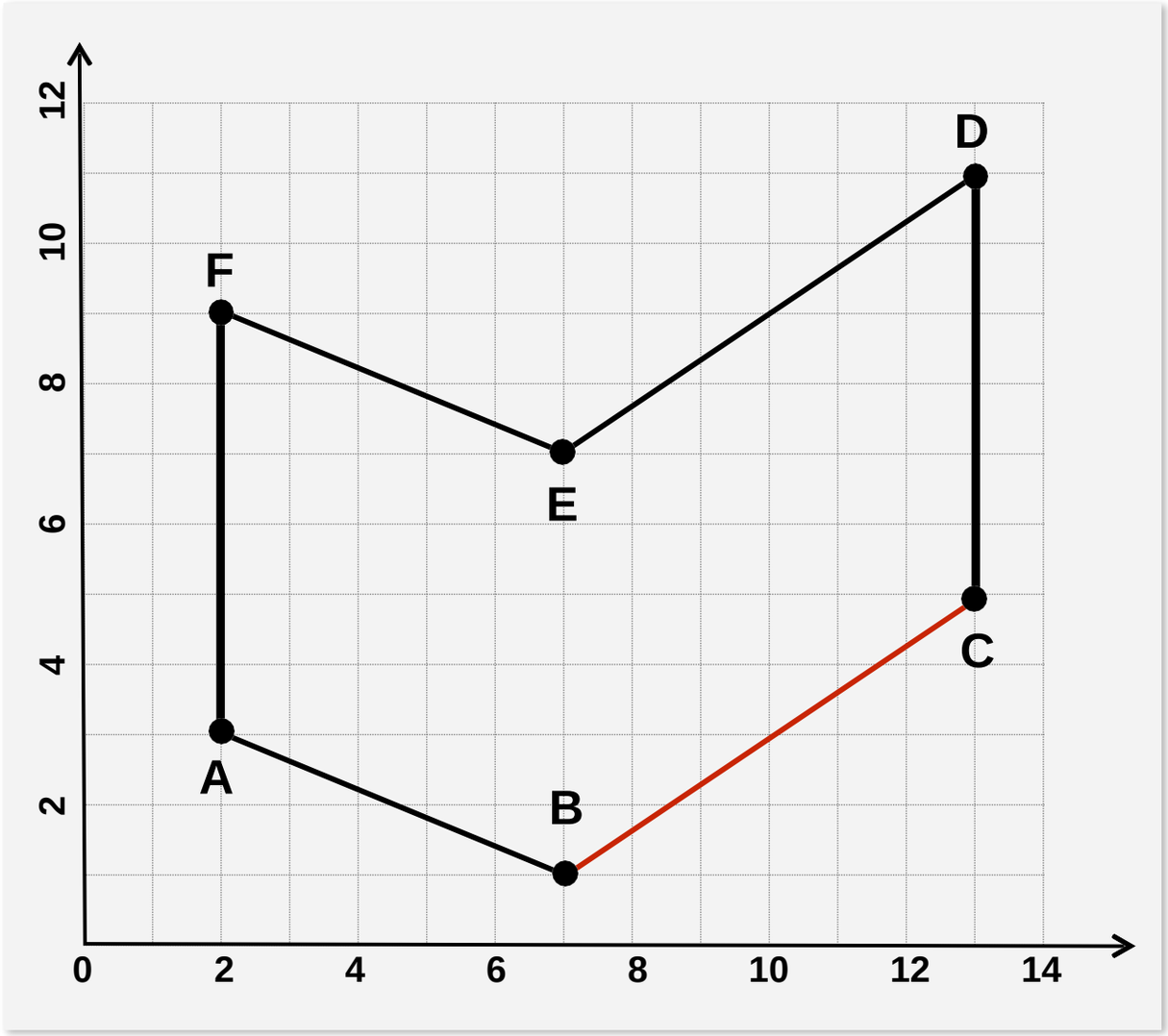
Scan-line algorithm

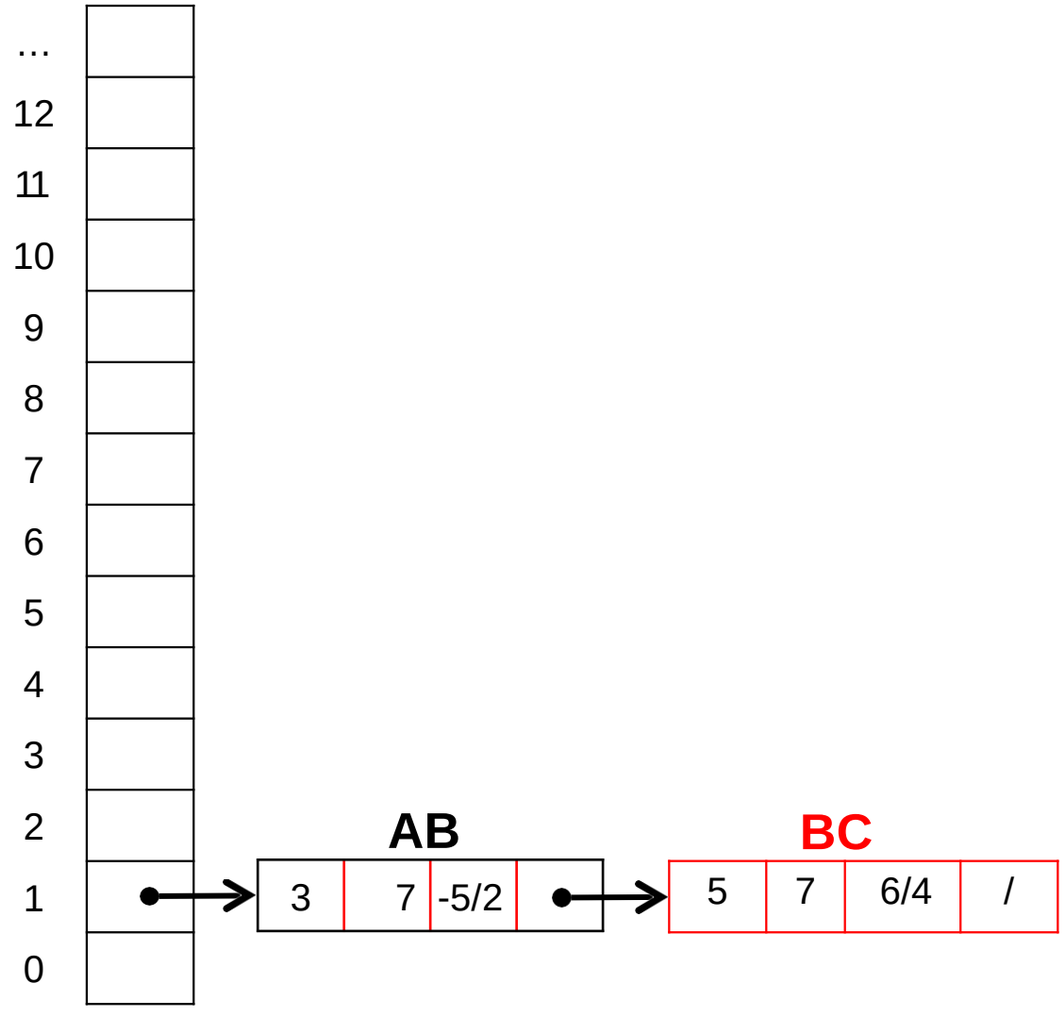




Y_{max}	X_{low}	1/m	

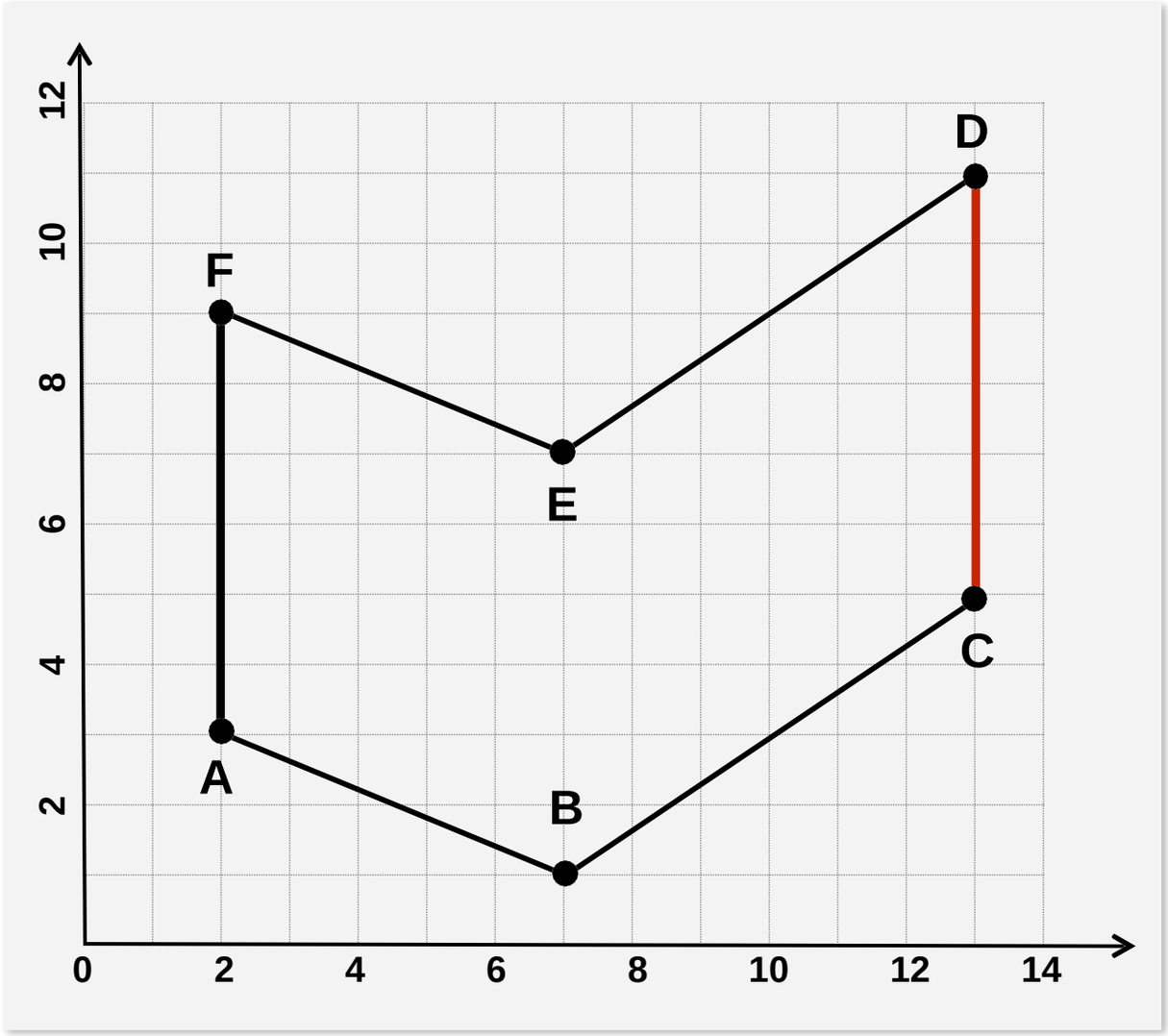
Scan-line algorithm

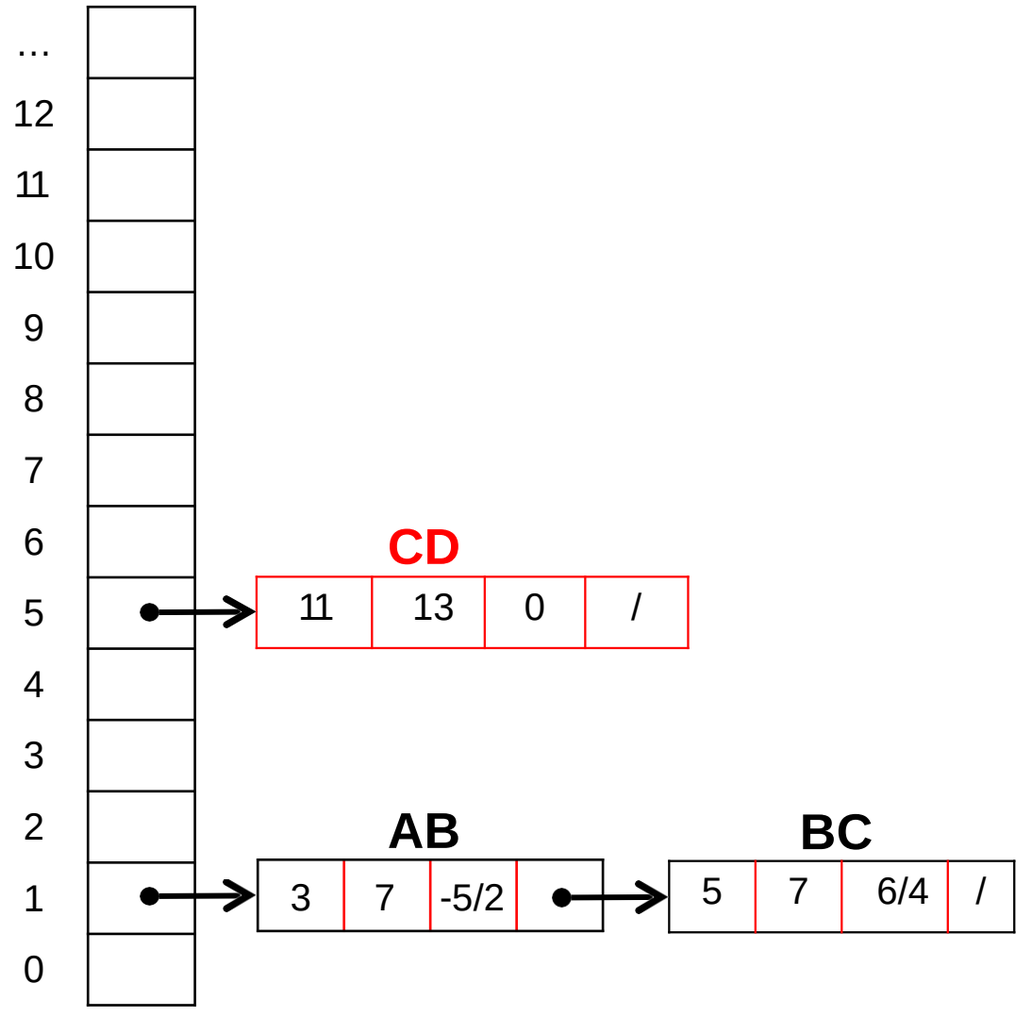




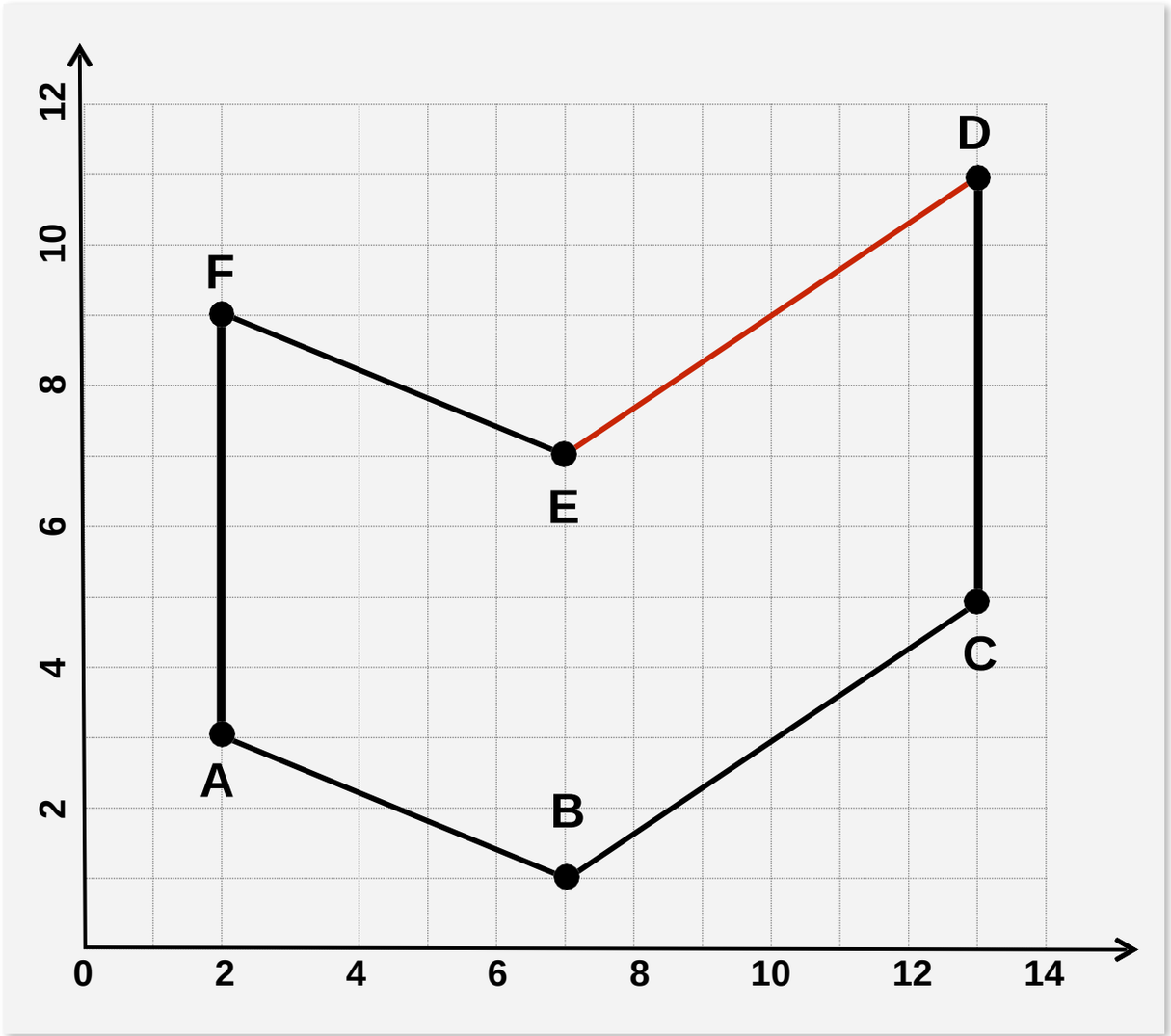
Y_{max}	X_{low}	1/m	

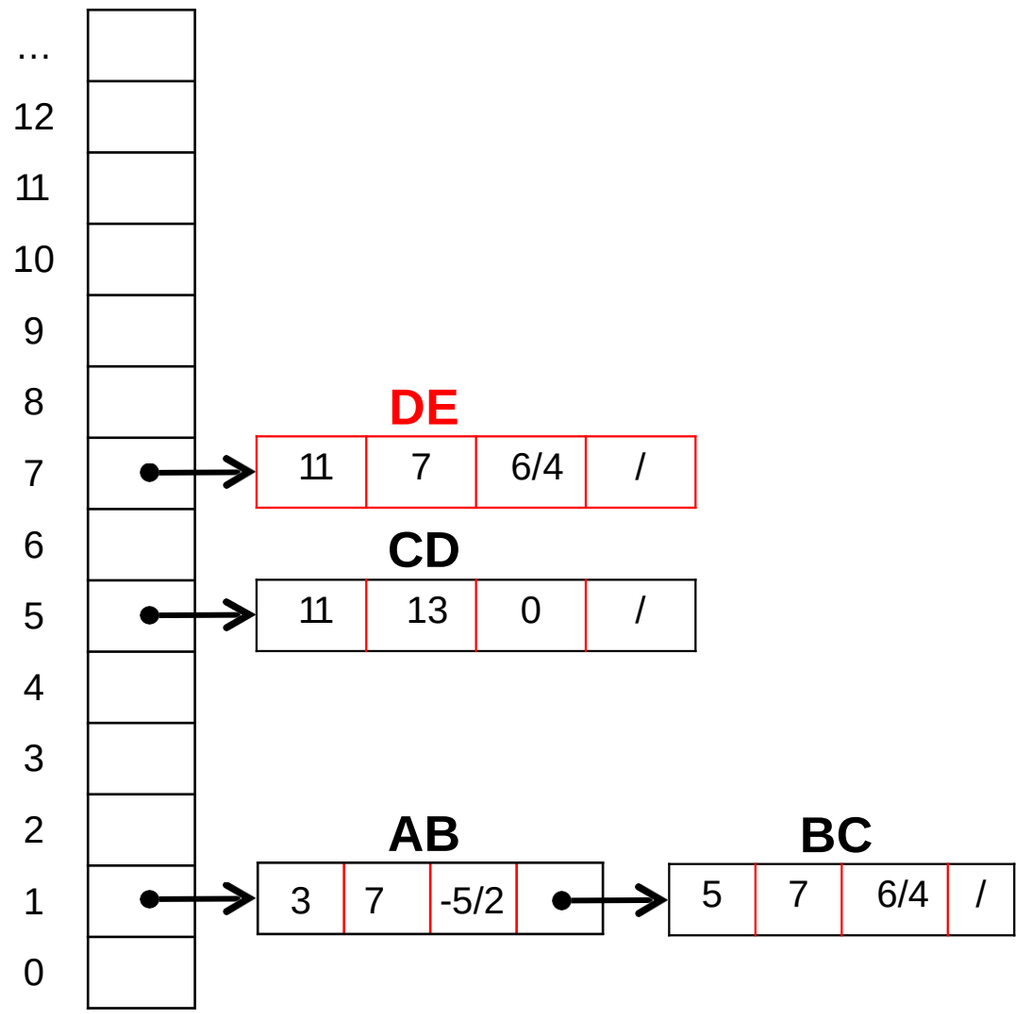
Scan-line algorithm



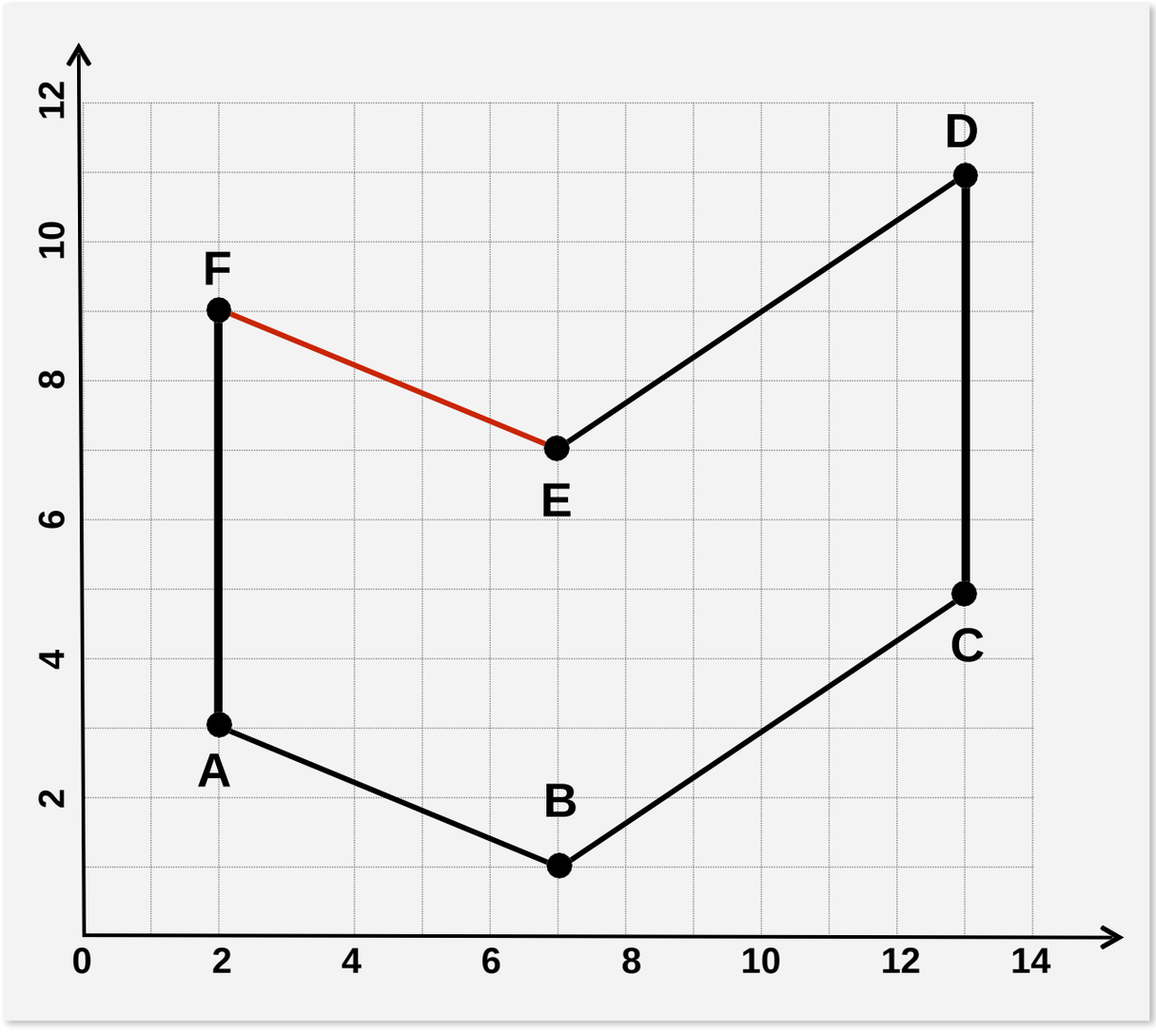


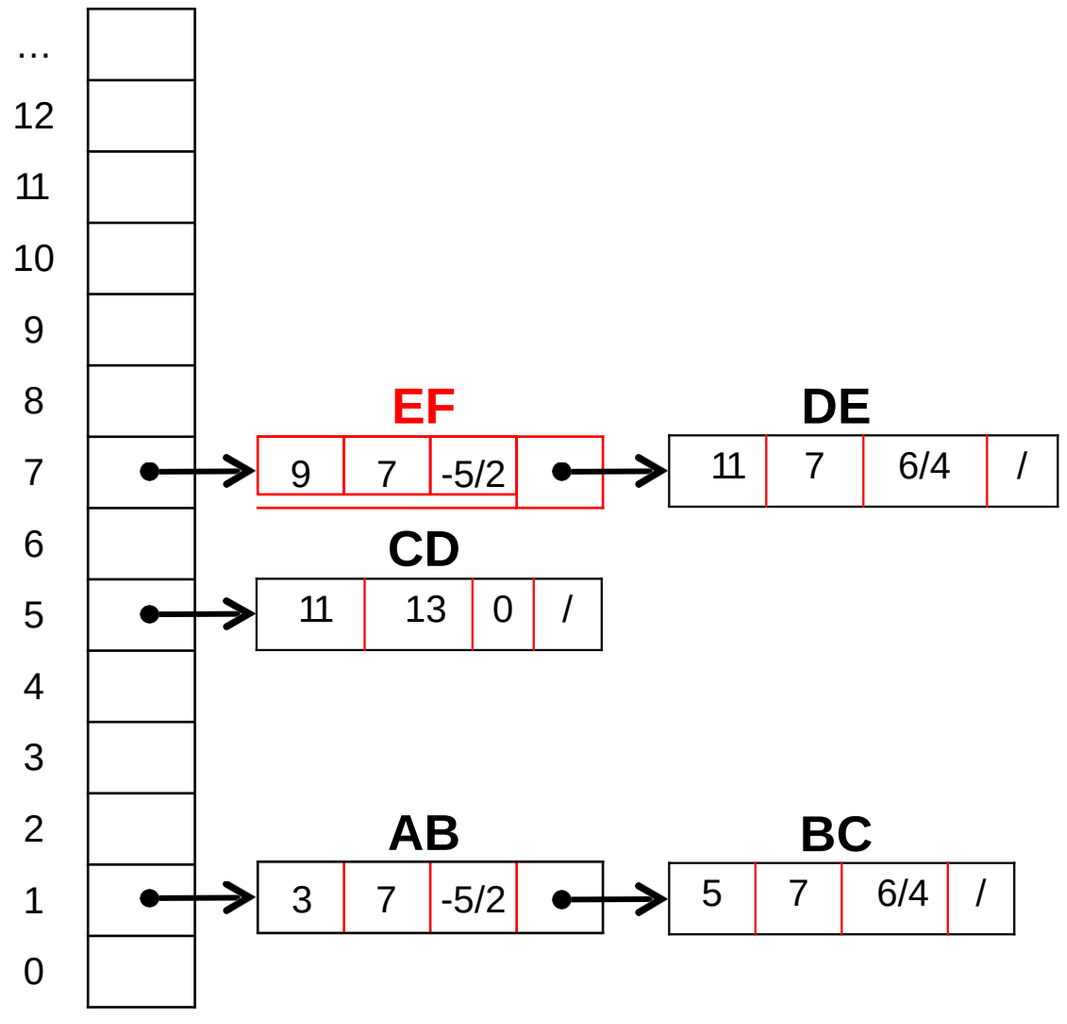
Scan-line algorithm



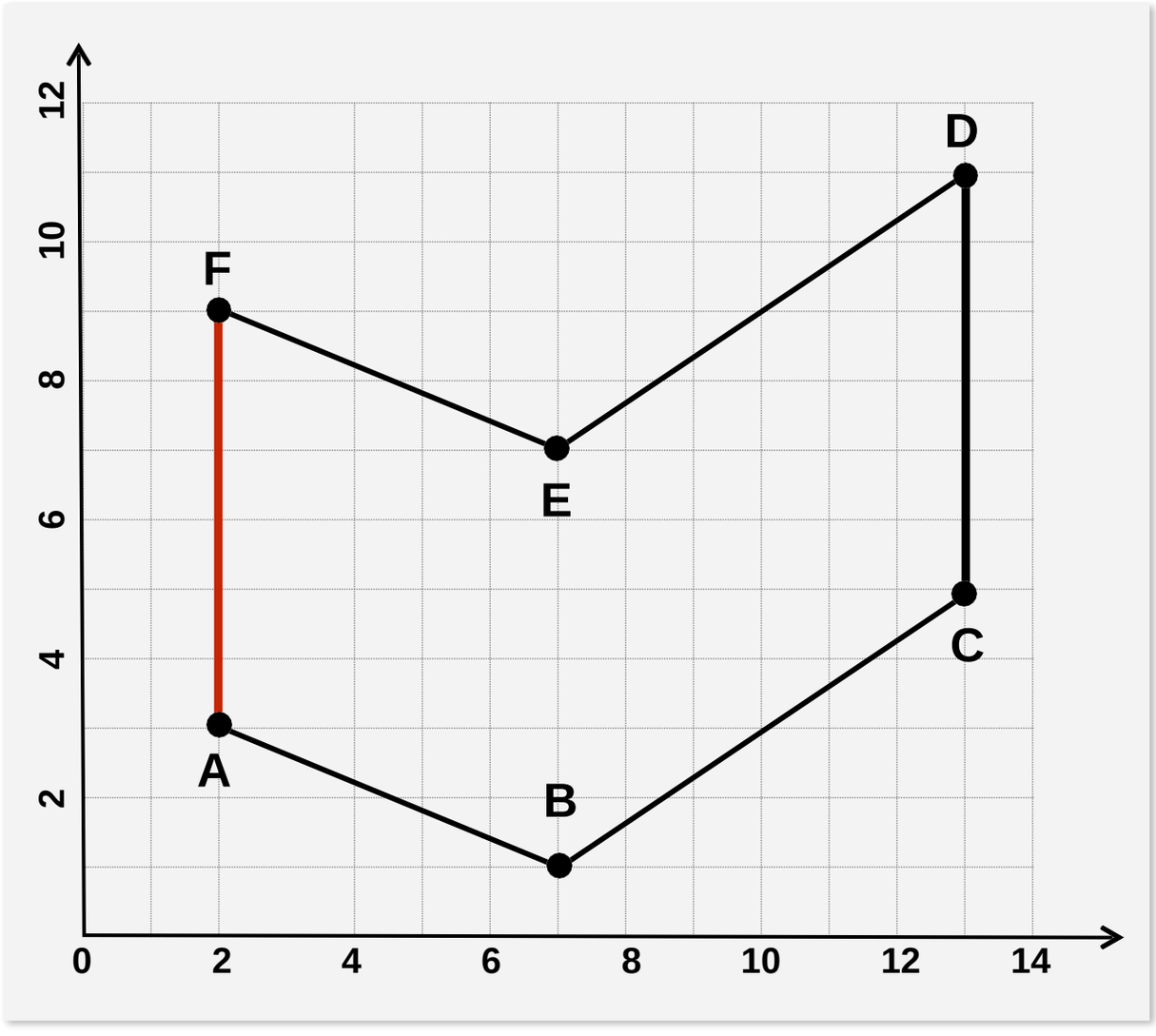


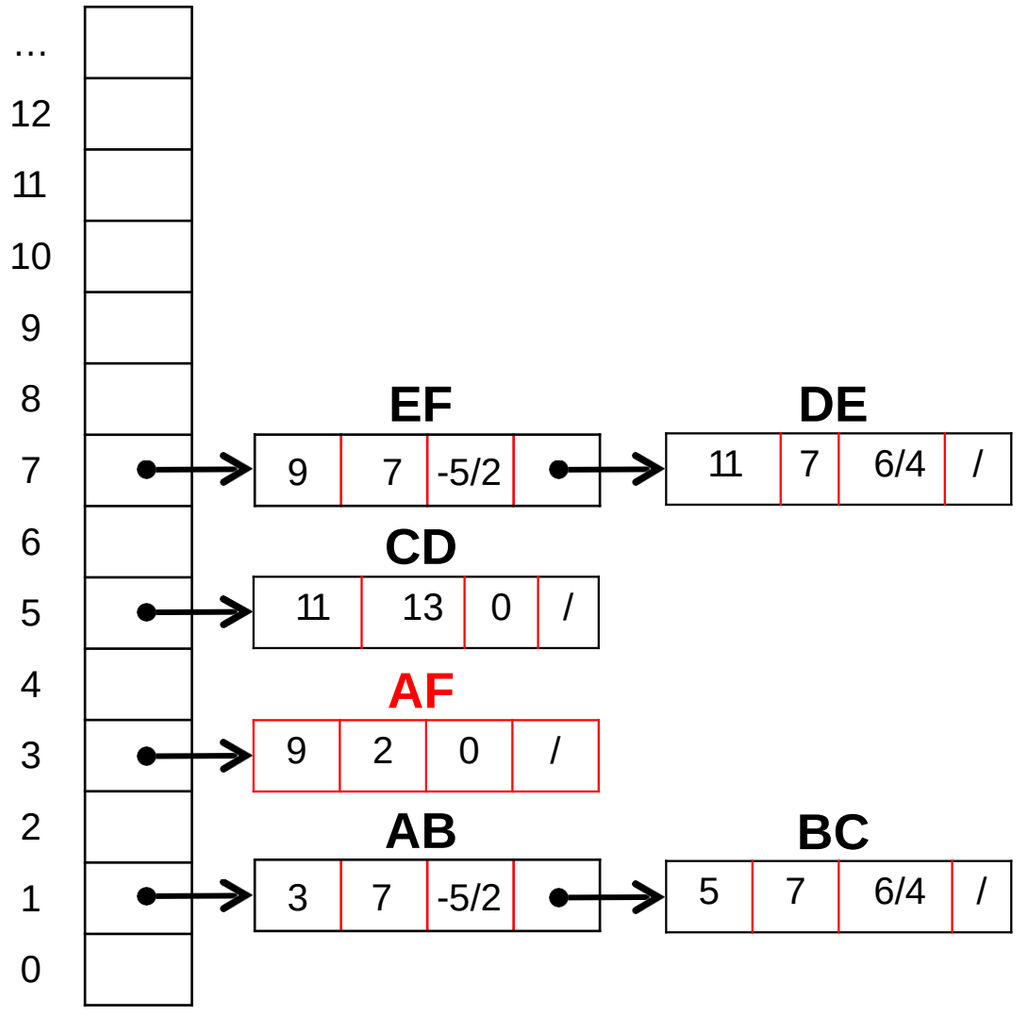
Scan-line algorithm





Scan-line algorithm

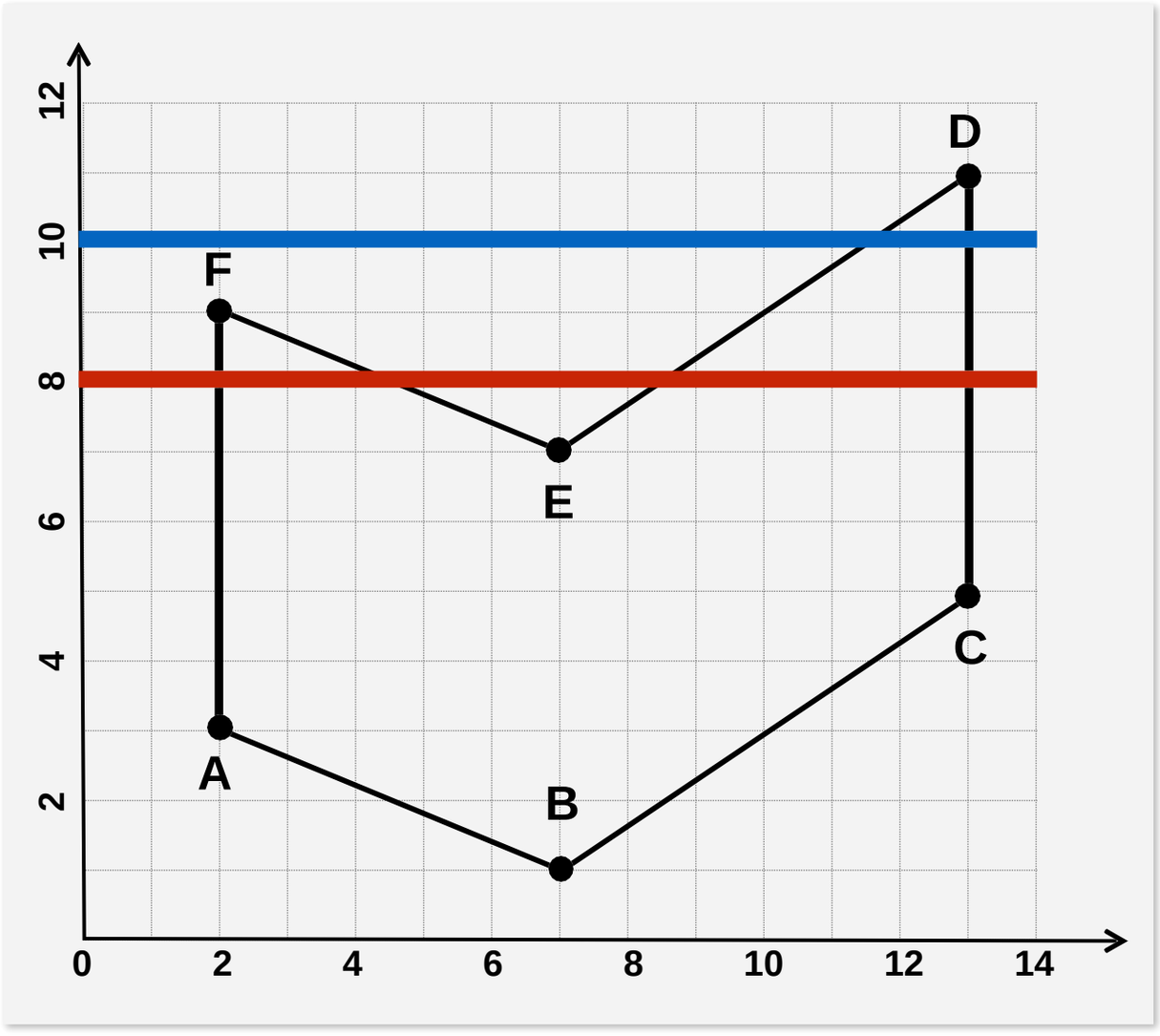


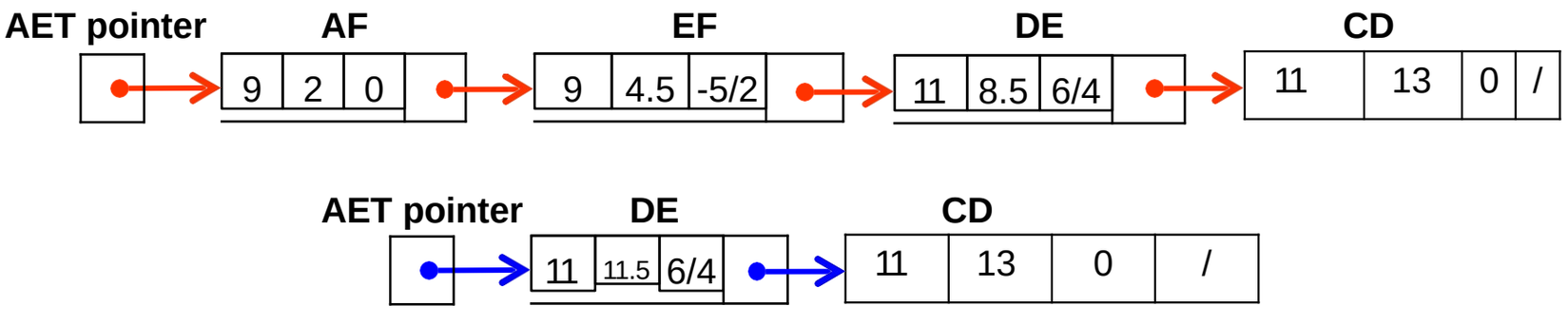


Scan-line algorithm

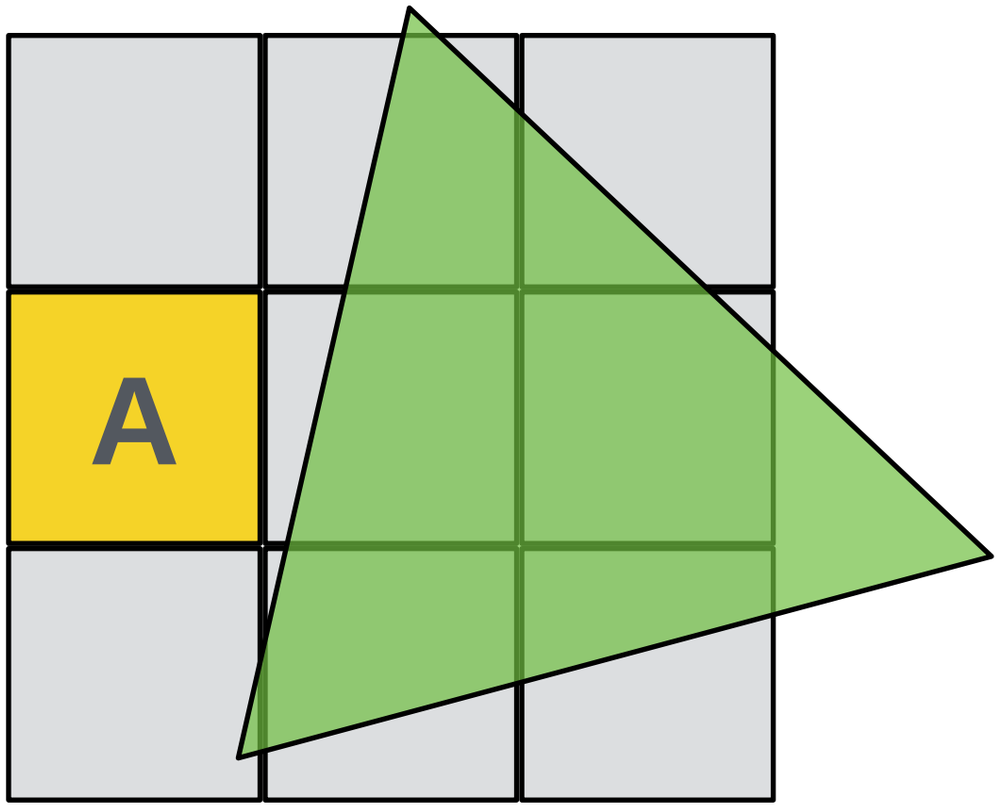
- Set y to the smallest y coordinate that has an entry in the **ET**
- Initialize the **AET** to be empty
- Repeat until the **AET** and **ET** are empty
  - Move from **ET** bucket y to the **AET** those edges whose $\mathbf{y_{min} = y}$
  - Remove from the AET those entries for which $\mathbf{y = y_{max}}$, then sort the **AET** on x
  - Fill in desired pixel values on scan line y by using pairs of x coordinates from the **AET**
  - Increment y by 1 (to the next scan line)
  - For each nonvertical edge remaining in the **AET**, update x for the new y

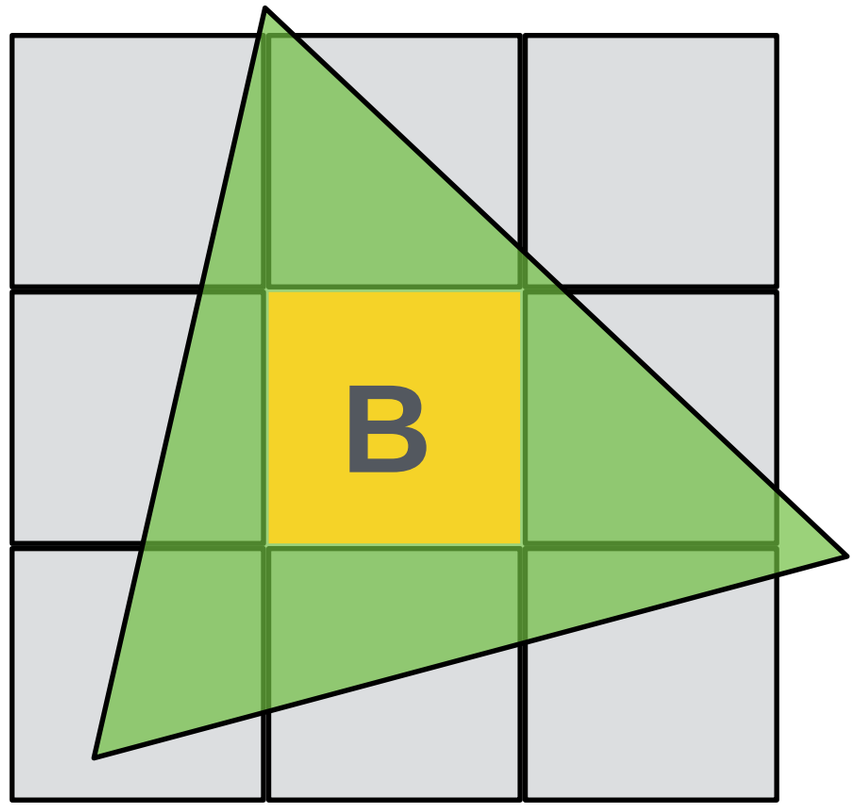
Scan-line algorithm



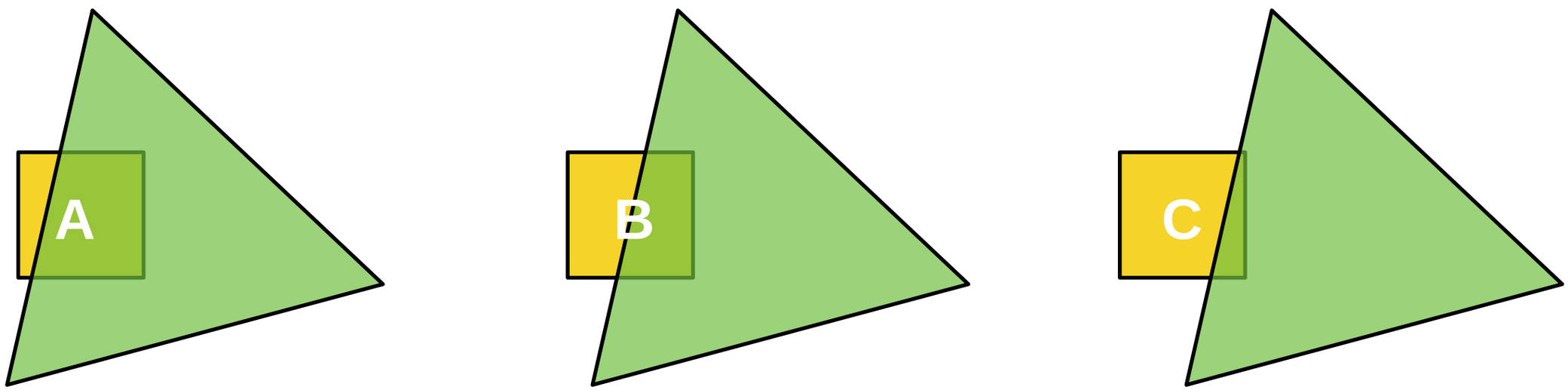


Which pixel is inside a triangle?

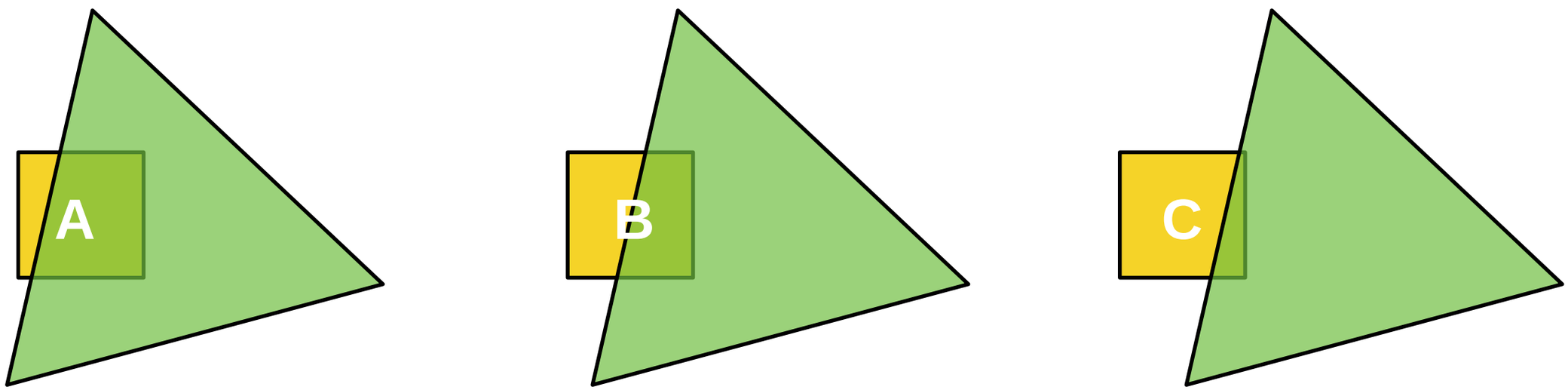




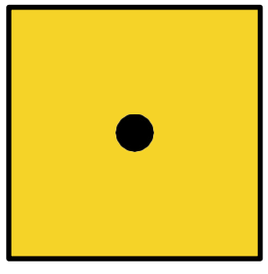
Which pixel is inside a triangle?



Which pixel is inside a triangle?



Solution: sample at the pixel's center

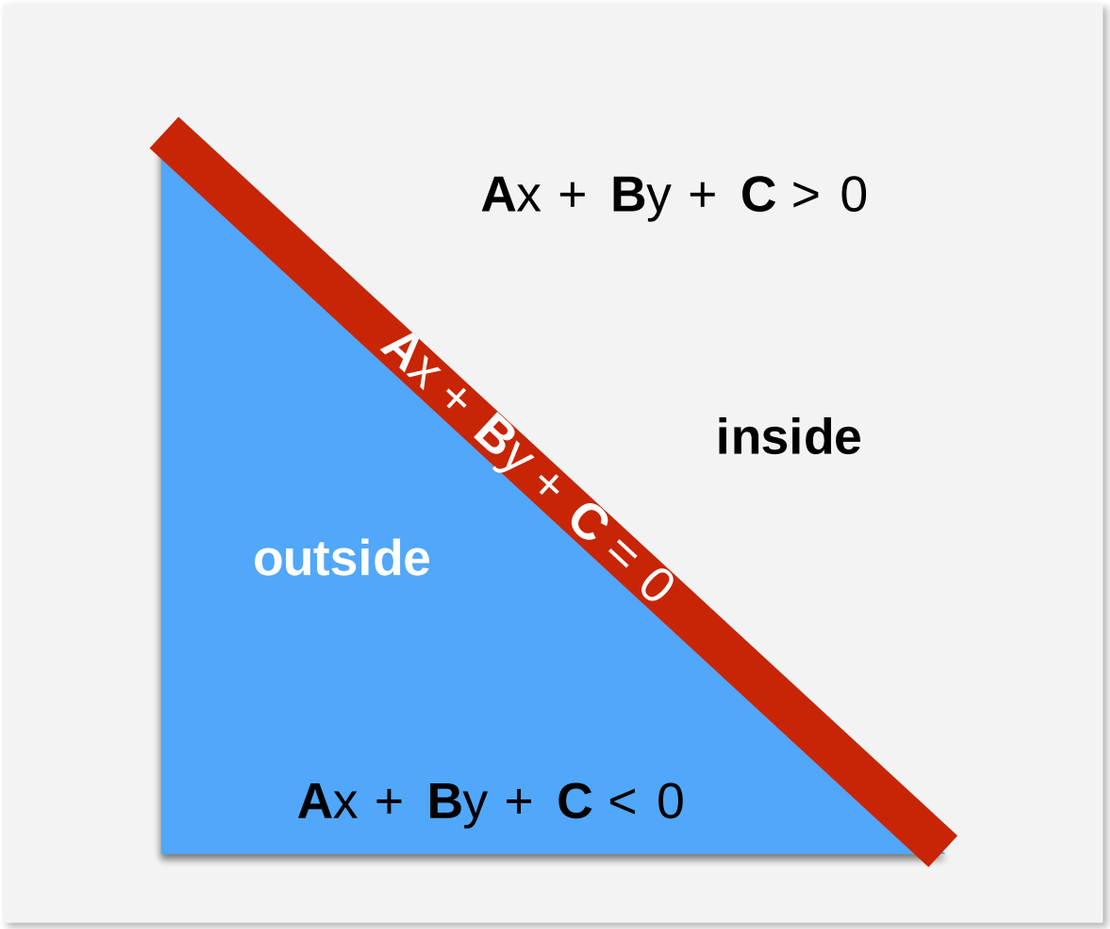


How do we determine if a sample is inside a triangle?

1. compute the **edge equations** (orient edge equations so that the negative half space be on the triangle's exterior)

2. scan through each pixel and **evaluate** against all edge equations

3. **set** pixel if all three edge equations > 0



Triangle rasterization

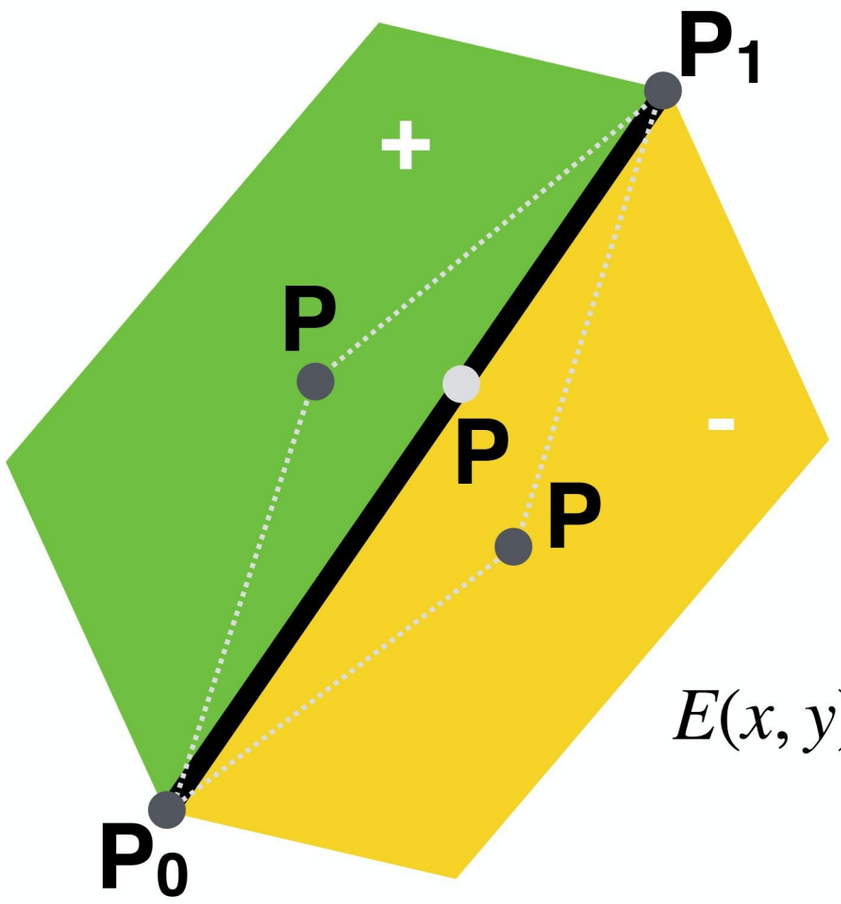
- General implicit form of a line (in 2D):

$$Ax + By + C = 0$$

- Implicit form of a line through two points $A(x_a, y_a)$ and $B(x_b, y_b)$:

$$(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a = 0$$

Edge equation



$$E(x,y) = \begin{vmatrix} P_{0x} & P_{1x} & P_x \\ P_{0y} & P_{1y} & P_y \\ 1 & 1 & 1 \end{vmatrix}$$

(subtract the first column from the other two and then develop the determinant with respect to the third row)

$$E(x,y) = \begin{vmatrix} P_{0x} & P_{1x} - P_{0x} & P_x - P_{0x} \\ P_{0y} & P_{1y} - P_{0y} & P_y - P_{0y} \\ 1 & 0 & 0 \end{vmatrix} = \begin{vmatrix} P_{1x} - P_{0x} & P_x - P_{0x} \\ P_{1y} - P_{0y} & P_y - P_{0y} \end{vmatrix}$$

$$E(x,y) = (P_{1x} - P_{0x})(P_y - P_{0y}) - (P_{1y} - P_{0y})(P_x - P_{0x})$$

(rewrite to the implicit form)

$$E(x,y) = aP_x + bP_y + c$$

$$a = -(P_{1y} - P_{0y})$$
$$b = P_{1x} - P_{0x}$$
$$c = P_{0x}P_{1y} - P_{0y}P_{1x}$$

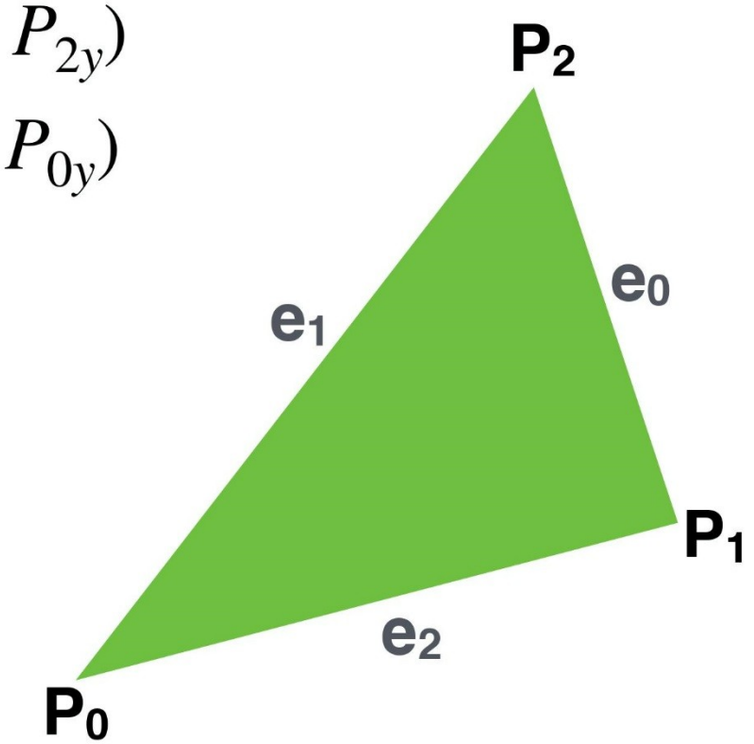
Edge equation

Define the edge equations $\mathbf{e_i(x,y)}$ by using two vertices, $\mathbf{p_j}$ and $\mathbf{p_k}$ such that $\mathbf{i \neq j}$ and $\mathbf{i \neq k}$:

$$e_0(x,y) = -(P_{2y} - P_{1y})(P_x - P_{1x}) + (P_{2x} - P_{1x})(P_y - P_{1y})$$

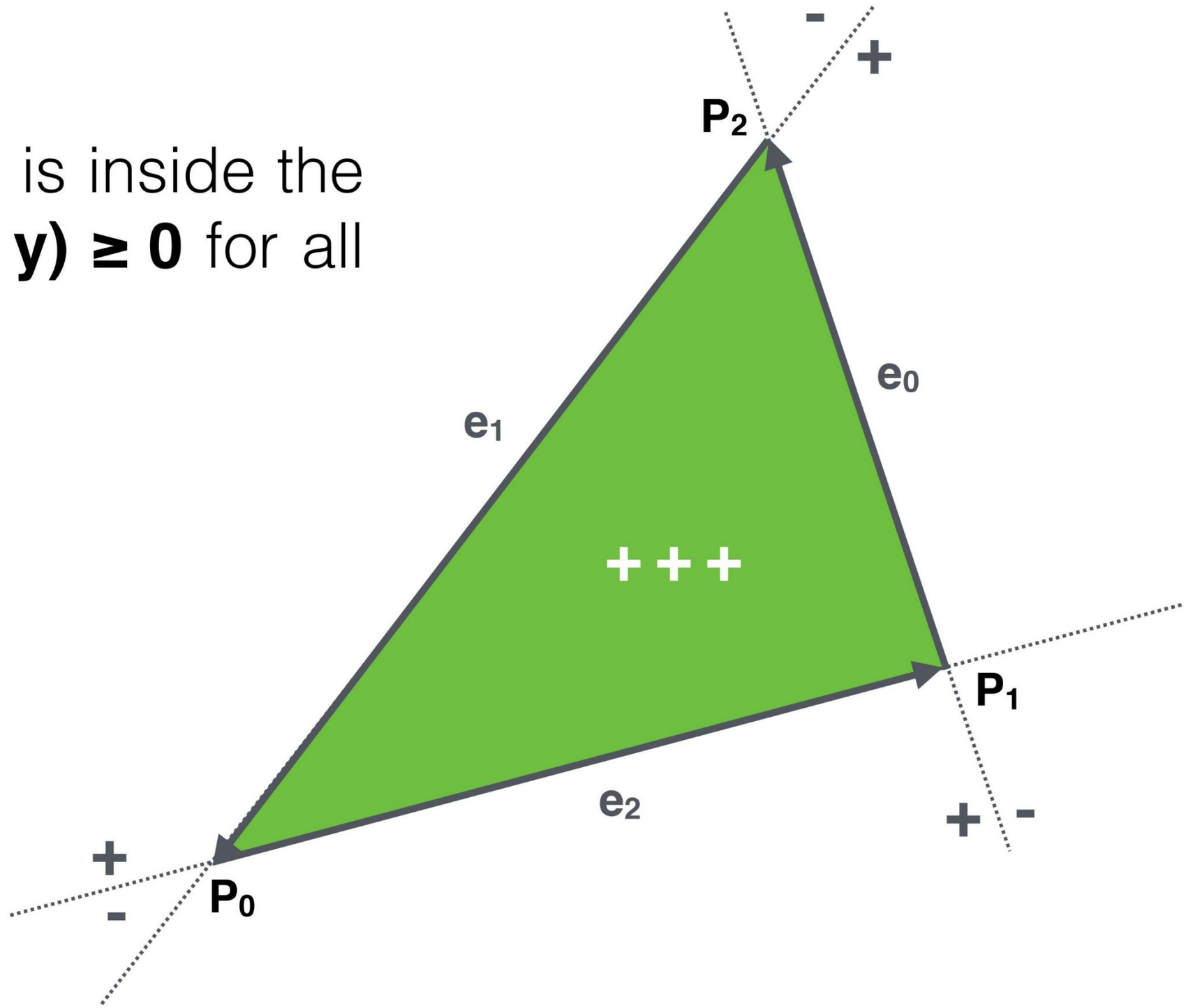
$$e_1(x,y) = -(P_{0y} - P_{2y})(P_x - P_{2x}) + (P_{0x} - P_{2x})(P_y - P_{2y})$$

$$e_2(x,y) = -(P_{1y} - P_{0y})(P_x - P_{0x}) + (P_{1x} - P_{0x})(P_y - P_{0y})$$

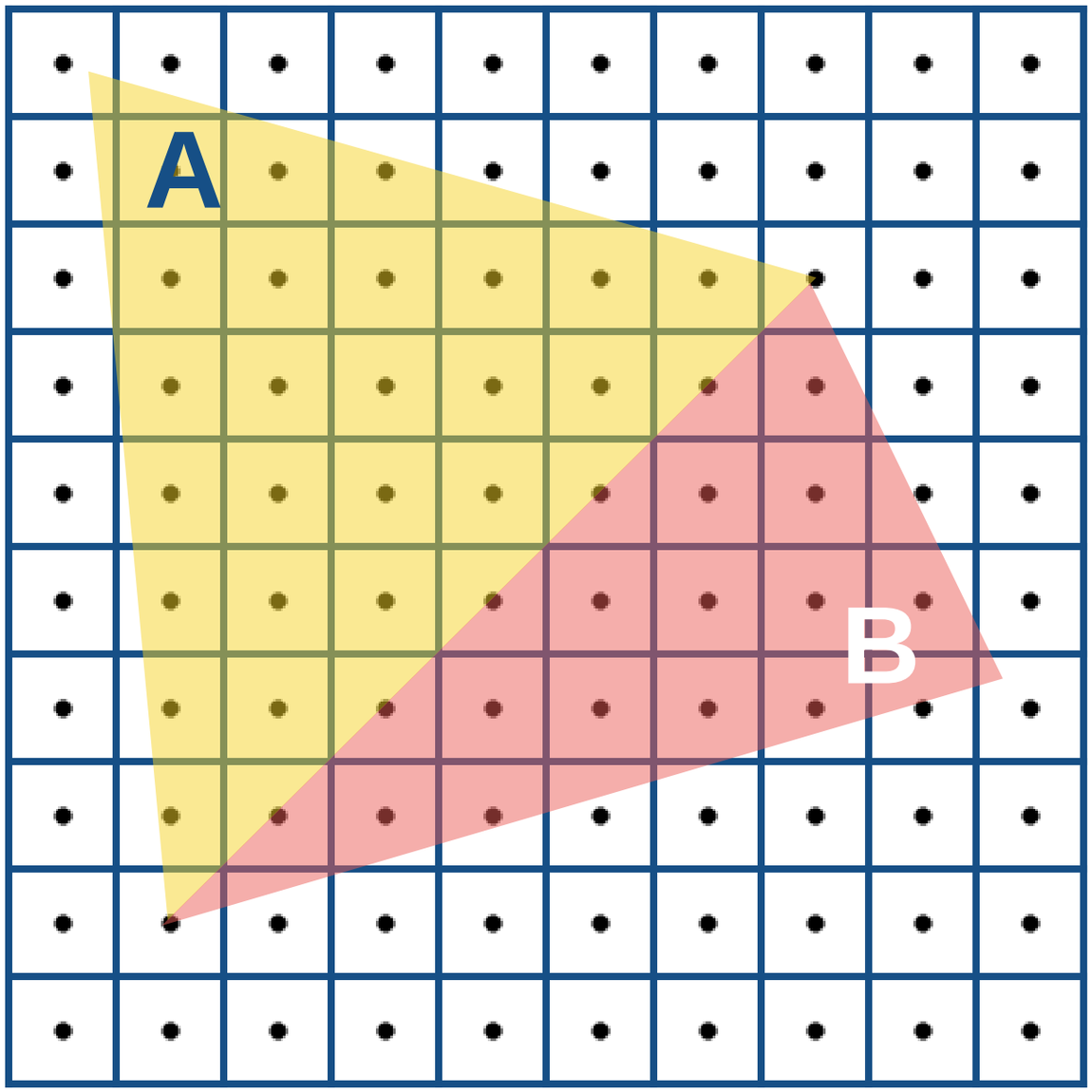


Inside test

A point **P(x, y)** is inside the triangle if $\mathbf{e_i(x, y) \geq 0}$ for all $\mathbf{i \in [0, 1, 2]}$

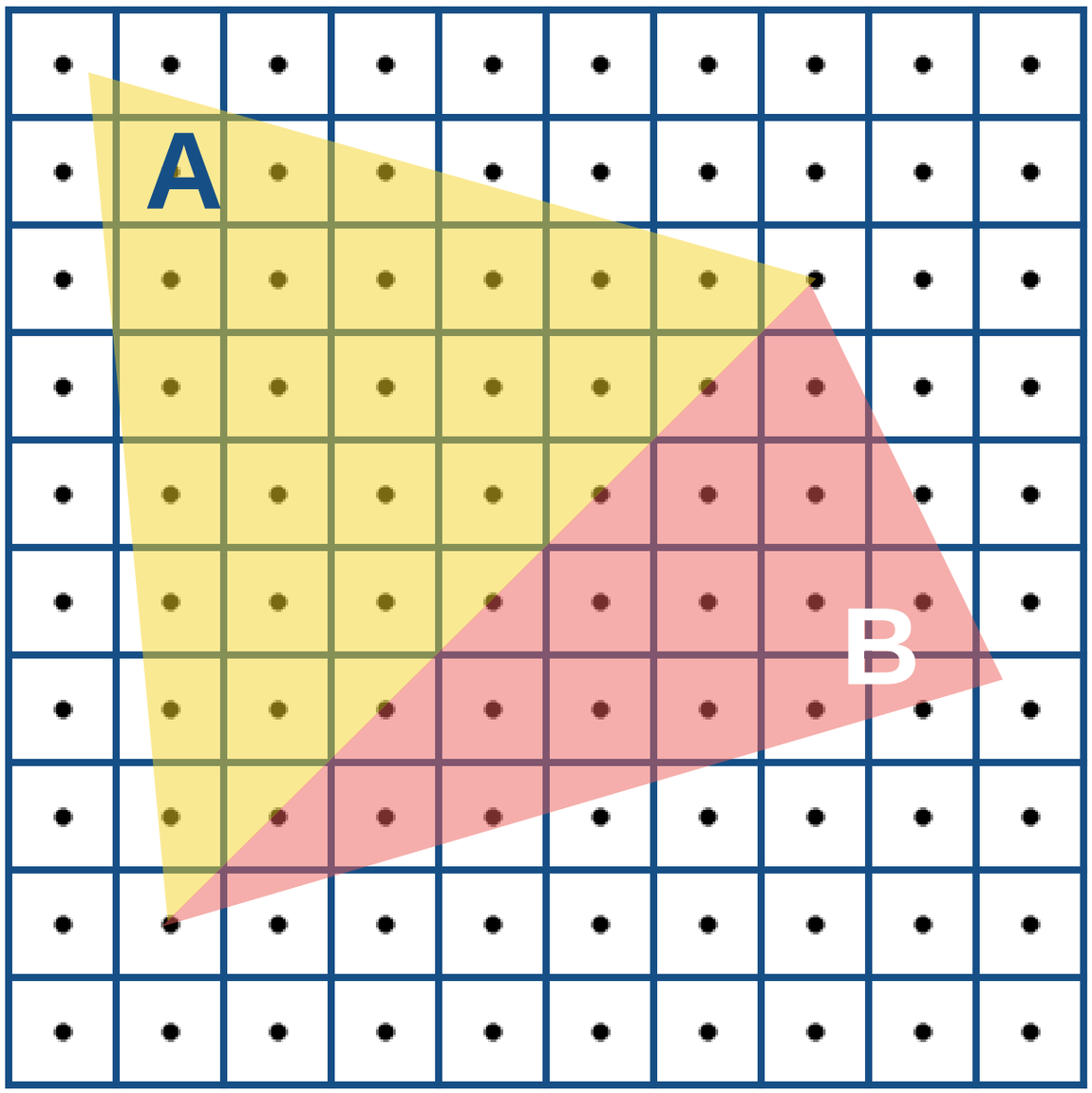


Shared edges



Question: Does the pixels on the shared edge belong to A or B?

Shared edges



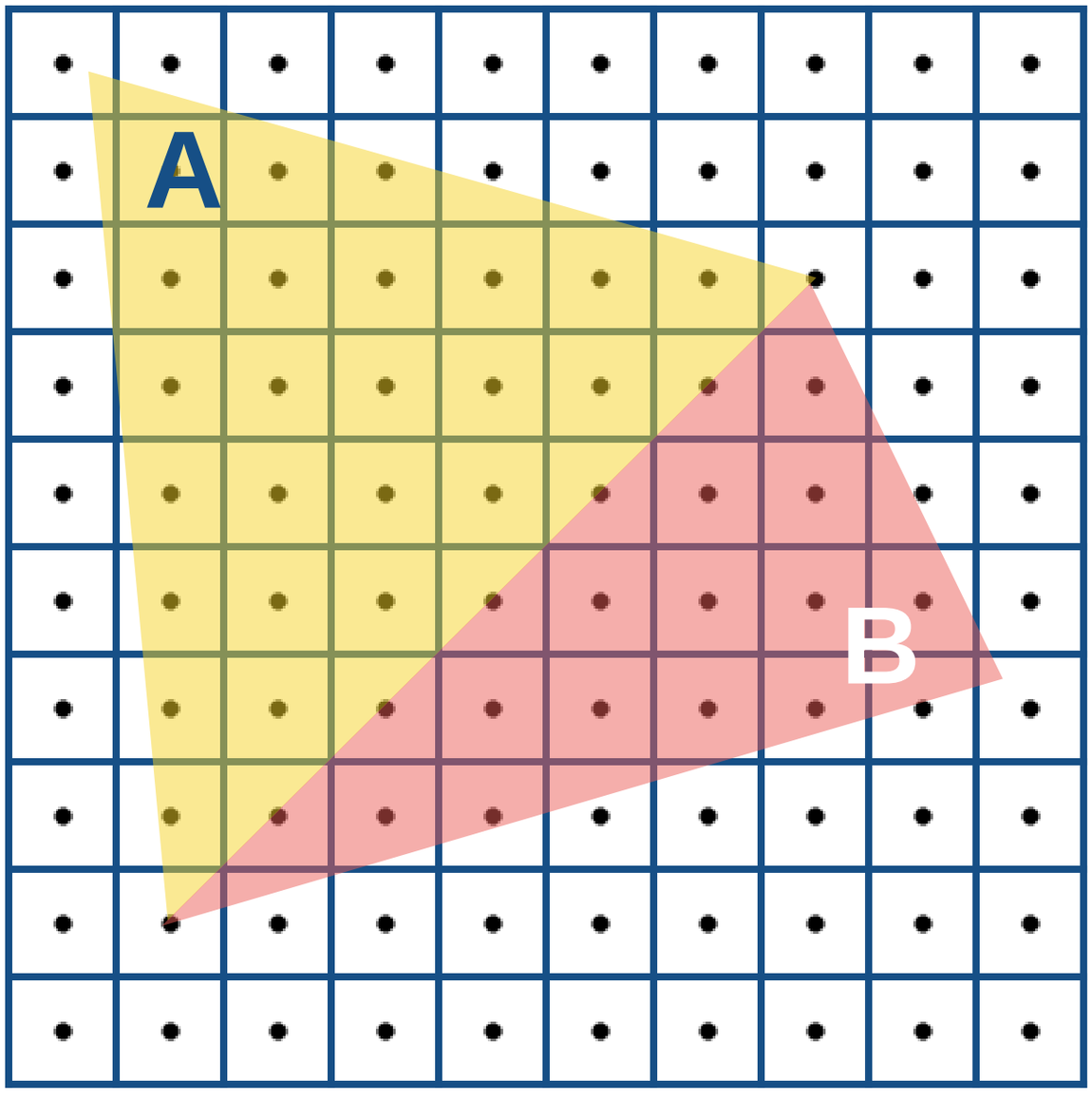
Question: Does the pixels on the shared edge belong to A or B?

Solution: Choose one and only one of A or B

Should avoid:

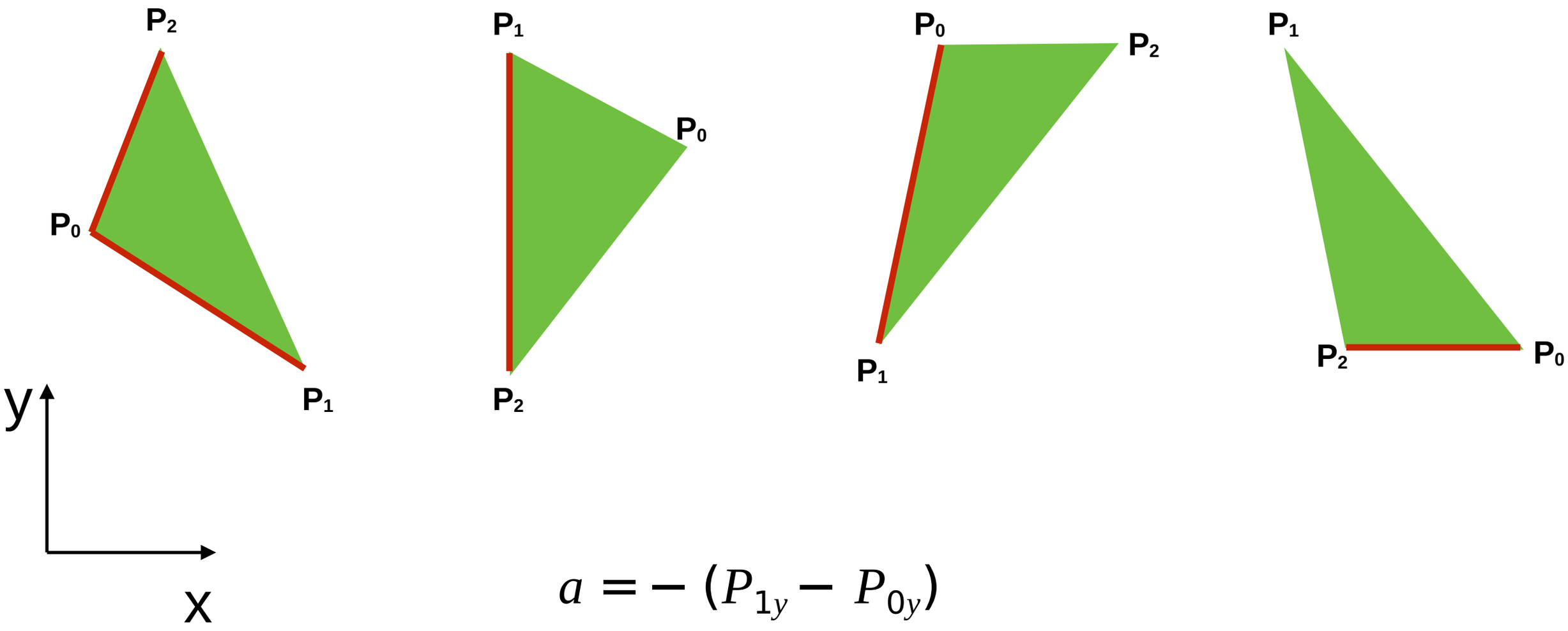
- Cracks between triangles
- Overlapping triangles

Tiebreaker rule



- **Include** points that are fully inside the triangle **(e > 0)**
- **Exclude** points that are fully outside **(e < 0)**
- **Include** points that are on the **left** edges
- **Include** points that are on the **bottom horizontal** edges

Left and bottom horizontal edges



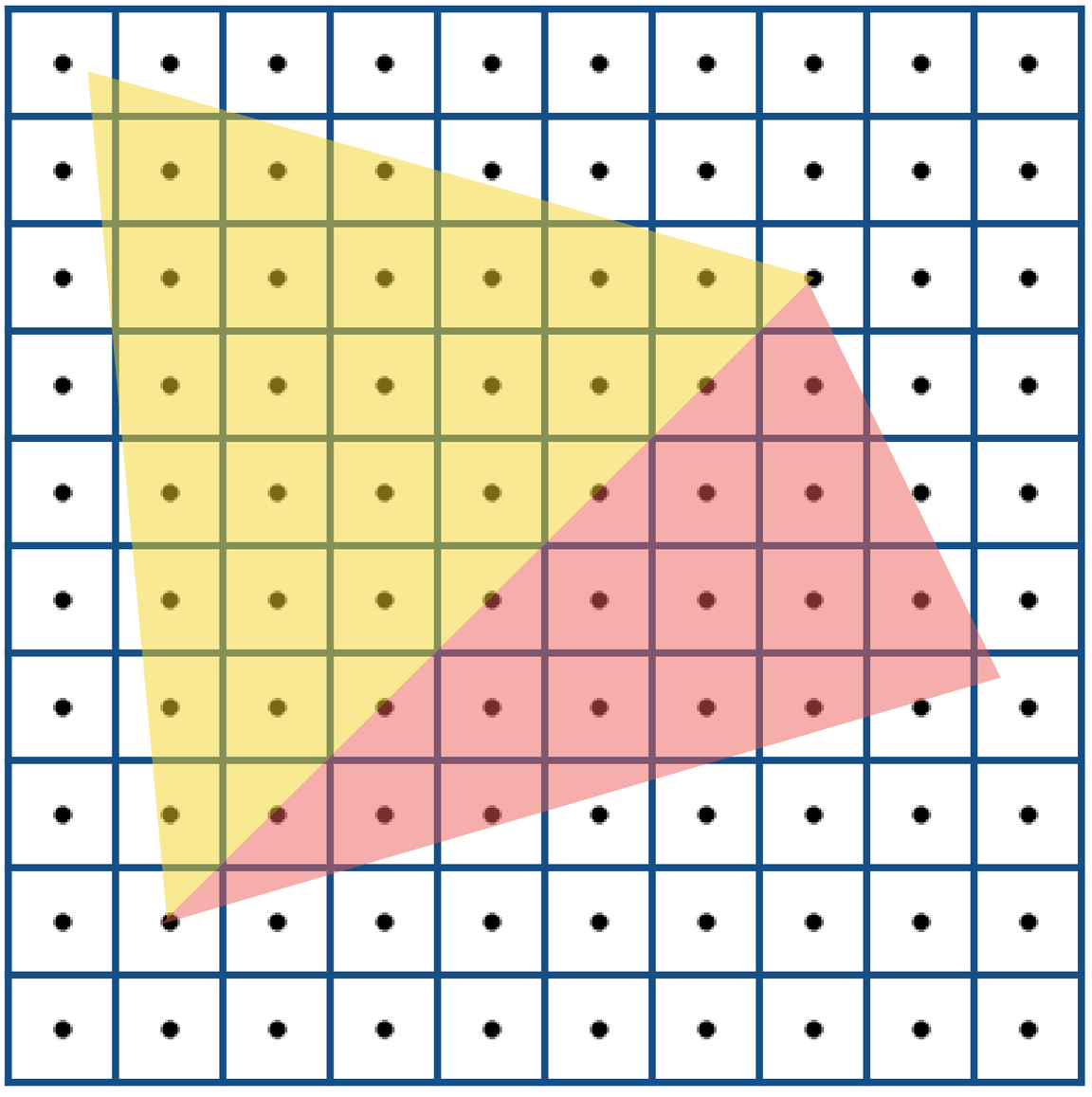
$$a = -(P_{1y} - P_{0y})$$

$$b = P_{1x} - P_{0x}$$

$$c = P_{0x}P_{1y} - P_{0y}P_{1x}$$

Tiebreaker rule

```
bool inside(e, x, y)
      if e(x, y) > 0 return true
      if e(x, y) < 0 return false
      if a > 0 return true
      if a < 0 return false
      if a = 0 && b > 0 return true
      return false
```



Triangle Traversal

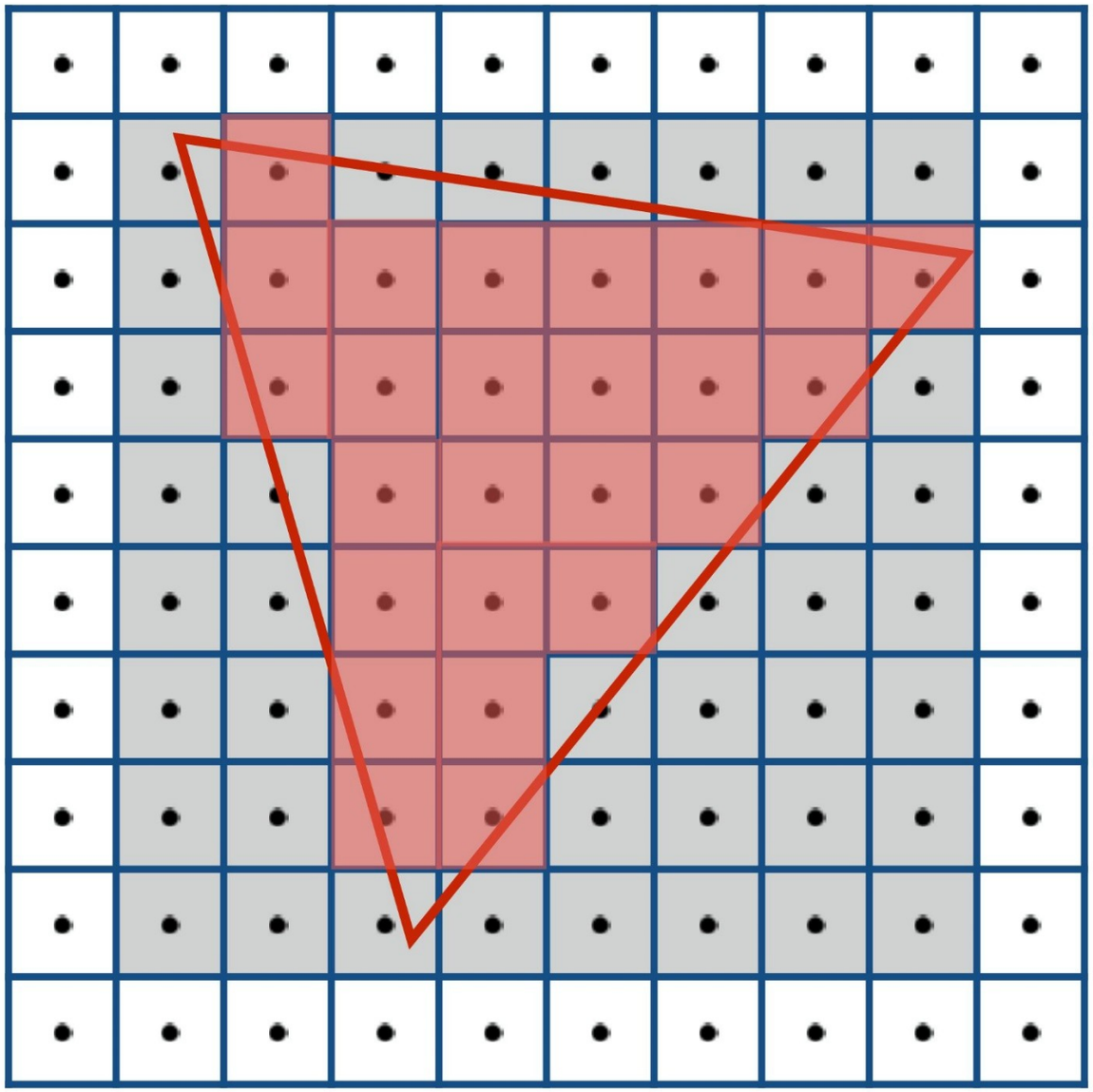
- The procedure that finds the pixels that are inside the triangle
- This process is often also referred to as rasterization
- Simple (and naive) triangle traversal algorithm:

  execute **inside()** function for every pixel **(x, y)** on screen, and for every edge

Bounding Box Traversal

- Compute a bounding box of the triangle
- Execute the **Inside(x,y)** test for each edge for each center point of the pixels inside the bounding box
- The pixels can be visited in any order, e.g., left-to-right/bottom-to-top

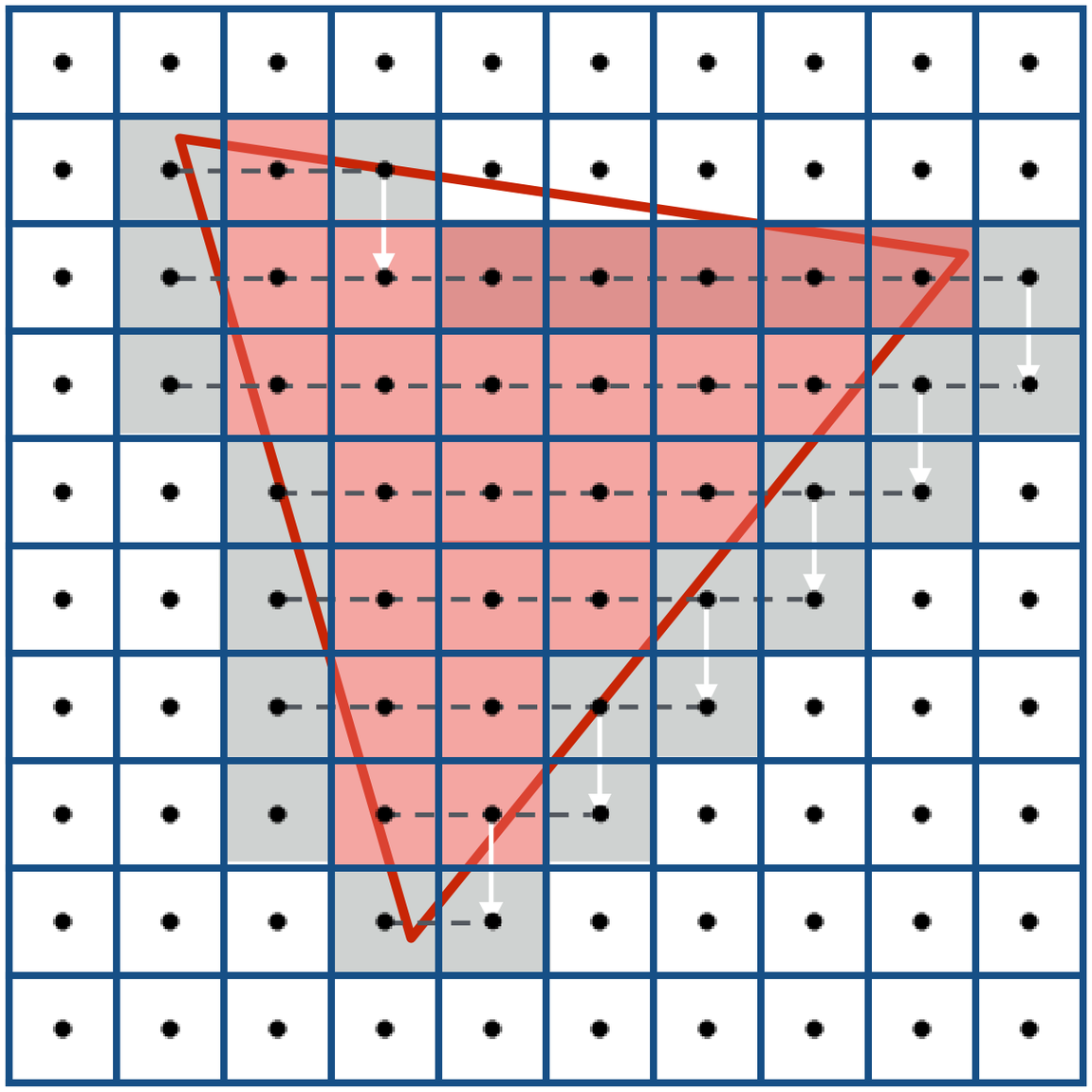
Example



(The circles represent the location of the sample points. The light gray pixels are included in the bounding box, but only the red pixels are actually inside the triangle.)

Backtrack traversal

- Start at the pixel center below the topmost vertex
- Process each scanline always from left to right
- Find a pixel that is outside to the left, continue to traverse to the right, stop when a pixel that is outside to the right of the triangle is found
- Continue to traverse to the pixel below the current pixel
- Backtrack to the left until a pixel that is outside to the left of the triangle is found, and so on
- Obs: Traversal never need to go outside the bounding box
- Obs: When visiting do not compute depth, color, etc.



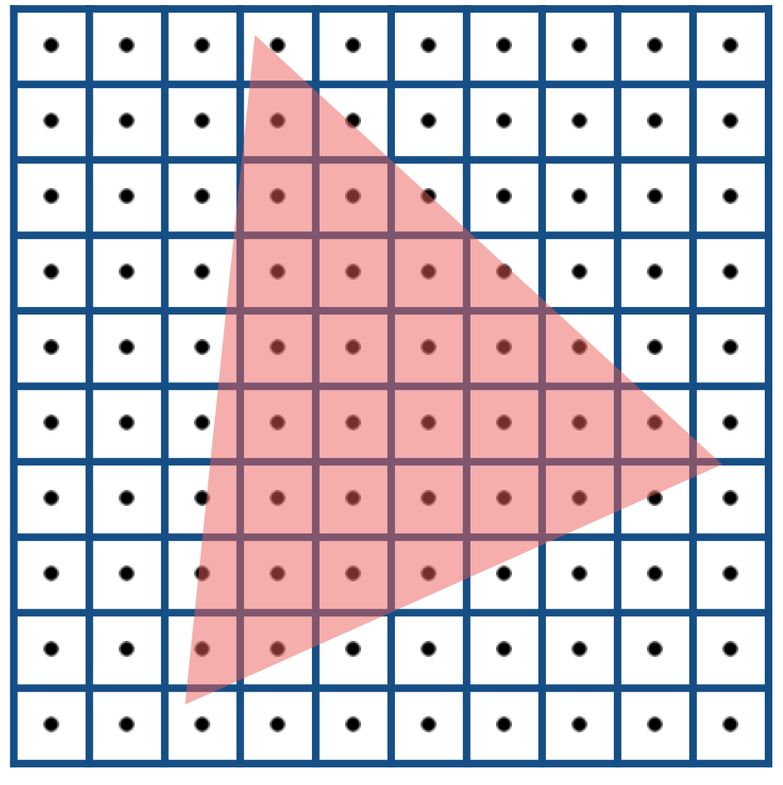
Neighbouring pixels

Evaluate the neighbouring pixel (in the positive x-direction) by:

$$e(x+1,y) = a(x+1) + by + c = e(x,\ y)\ +\ a$$

Evaluate the four neighbouring pixels by:

$$e(x+1,y) = e(x,\ y)\ +\ a$$
$$e(x-1,y) = e(x,\ y)\ -\ a$$
$$e(x,\ y+1) = e(x,\ y)\ +\ b$$
$$e(x,\ y-1) = e(x,\ y)\ -\ b$$



Backtrack traversal

- ***outside to the left***

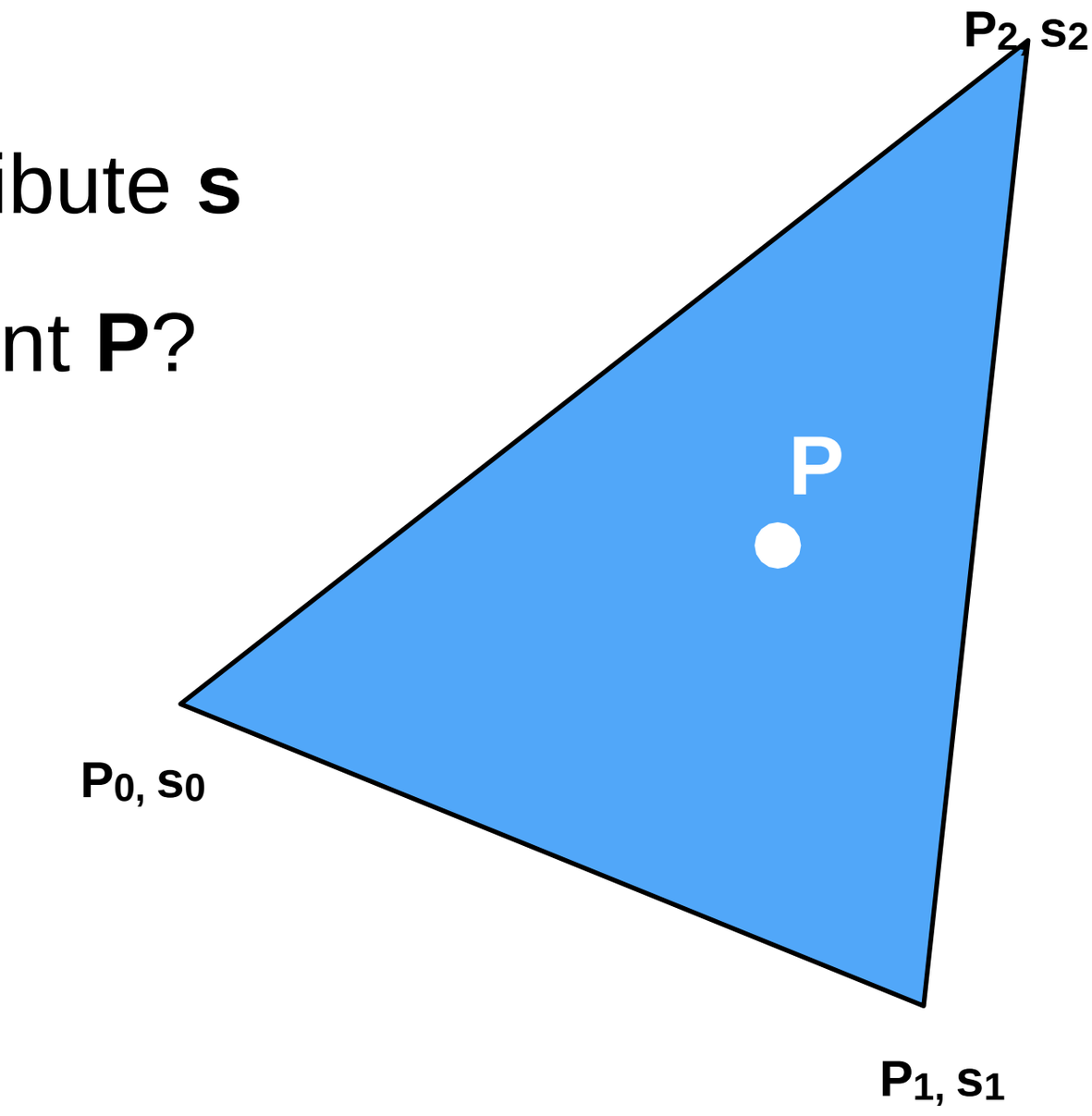
  point that is outside at least one edge function with $a > 0$

- ***outside to the right***

  point that is outside at least one edge function with $a < 0$

Interpolate parameters across triangles

- Each vertex has an attribute **s**
- What value has **s** at point **P**?
- **s** should vary smoothly across the triangle
- **Solution:** barycentric coordinates



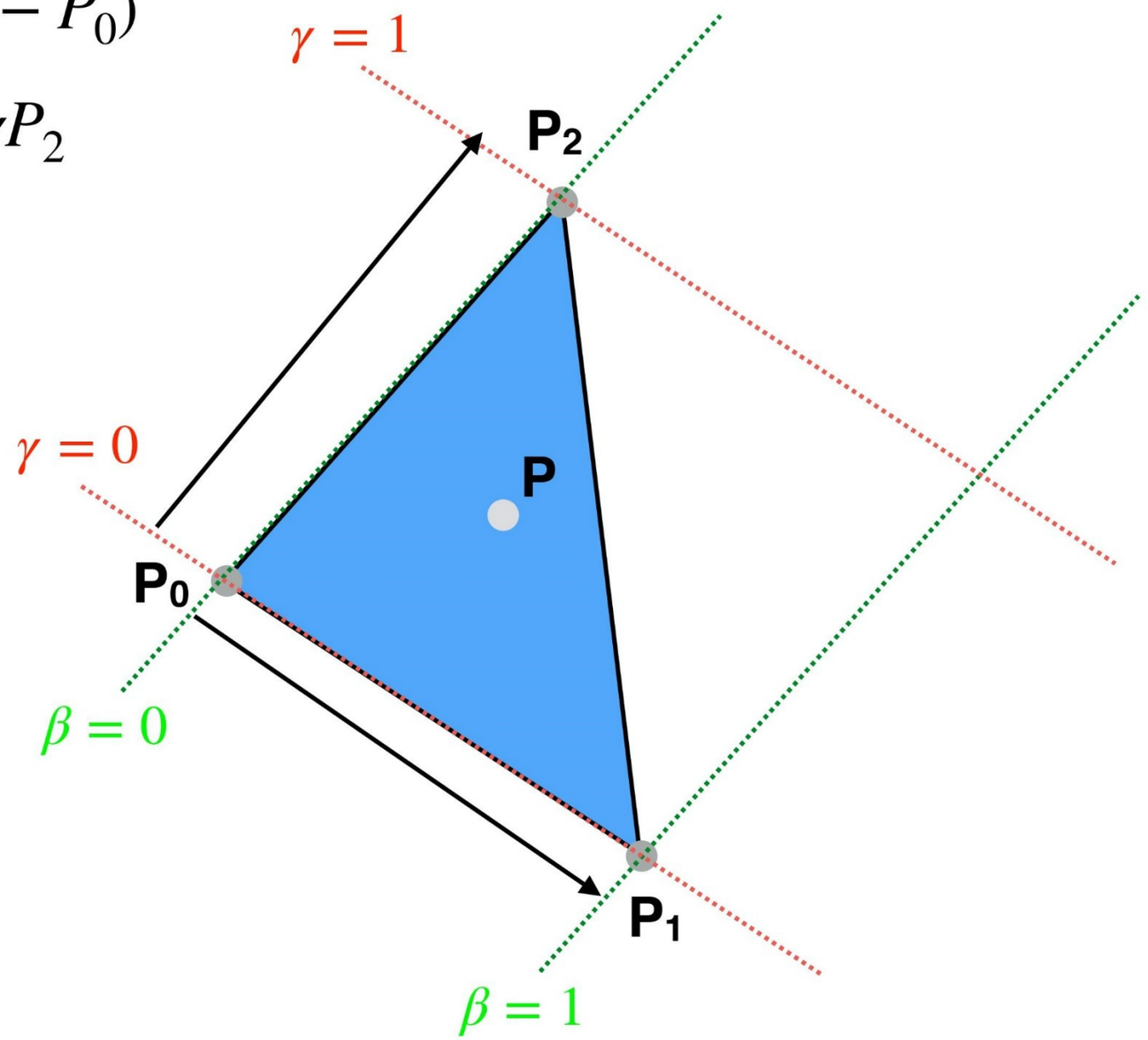
Barycentric Coordinates

- Any point **P** is a linear combination:

$$P = P_0 + \beta(P_1 - P_0) + \gamma(P_2 - P_0)$$

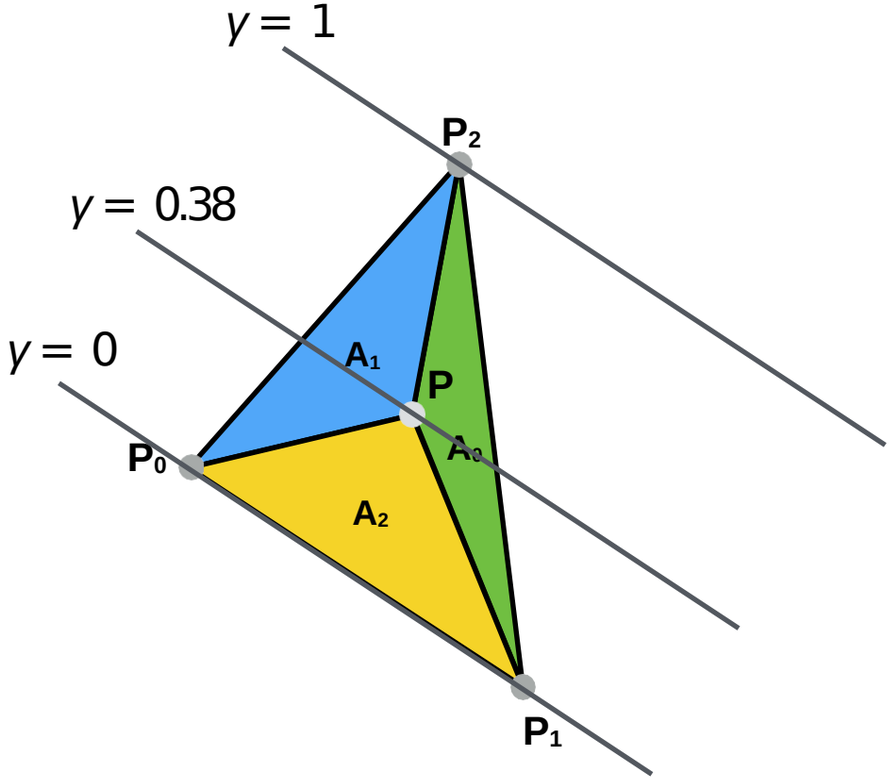
$$P = (1 - \beta - \gamma)P_0 + \beta P_1 + \gamma P_2$$

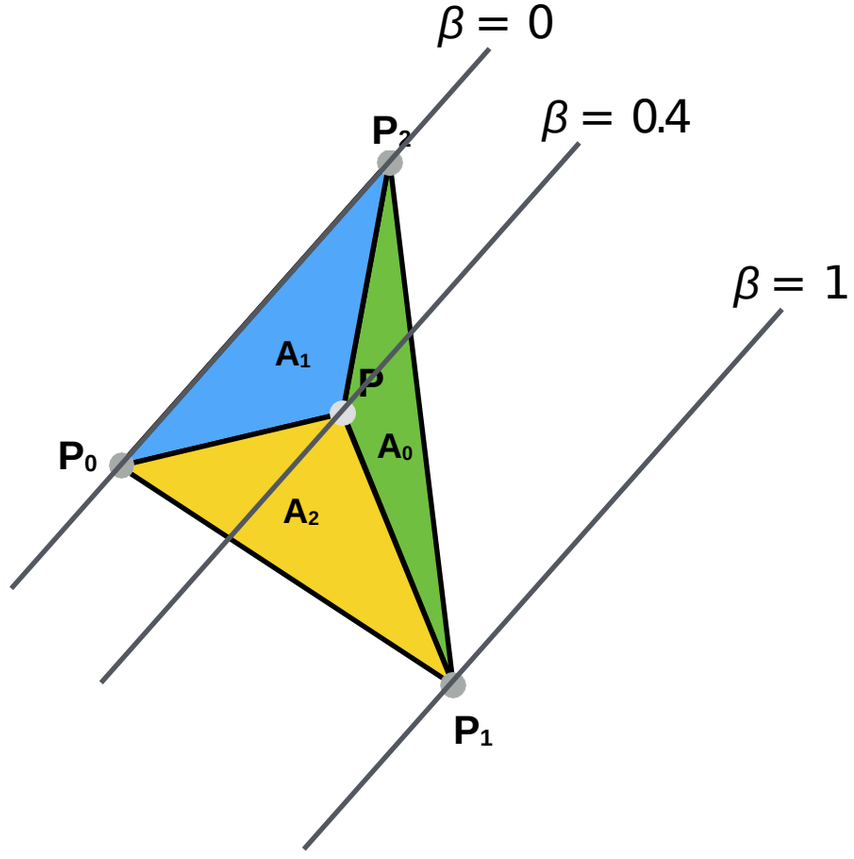
$$P = \alpha P_0 + \beta P_1 + \gamma P_2$$



Barycentric Coordinates

- The barycentric coordinates are proportional to the signed areas ($\mathbf{A_0}$,$\mathbf{A_1}$,$\mathbf{A_2}$) of the sub-triangles formed by a triangle edge and **P**





Barycentric Coordinates

- The barycentric coordinates are proportional to the signed areas (**A0**,**A1**,**A2**) of the sub-triangles formed by a triangle edge and **P**:

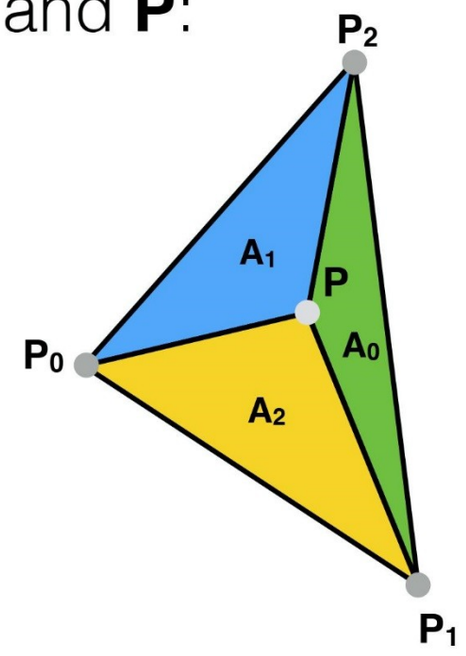
$$\alpha = \frac{A_0}{A_\triangle} \qquad \beta = \frac{A_1}{A_\triangle} \qquad \gamma = \frac{A_2}{A_\triangle}$$

$$\alpha + \beta + \gamma = 1$$

$$A_0 = \frac{1}{2}((P_x - P_{1x})(P_{1y} - P_{2y}) - (P_y - P_{1y})(P_{1x} - P_{2x}))$$

$$A_1 = \frac{1}{2}((P_x - P_{0x})(P_{2y} - P_{0y}) - (P_y - P_{0y})(P_{2x} - P_{0x}))$$

$$A_2 = \frac{1}{2}((P_x - P_{0x})(P_{0y} - P_{1y}) - (P_y - P_{0y})(P_{0x} - P_{1x}))$$



- For a point **P** inside the triangle:

$$\beta \geq 0 \qquad \gamma \geq 0 \quad \beta + \gamma \leq 1$$

Barycentric coordinates

- Barycentric coordinates describe a point **p** as an affine combination of the triangle vertices:

$$P = \alpha P_0 + \beta P_1 + \gamma P_2 \qquad \alpha + \beta + \gamma = 1$$

- For any point p inside the triangle specified by vertices:

$$0 < \alpha < 1$$
$$0 < \beta < 1$$
$$0 < \gamma < 1$$

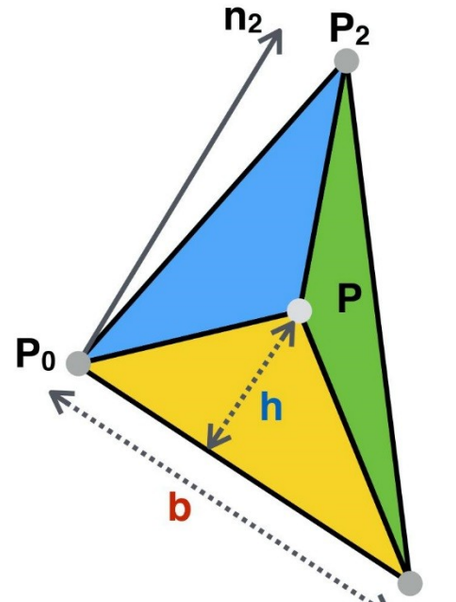
- Point on an edge if one coefficient is 0, point on a vertex if two coefficients are 0

Barycentric Coordinates from edge equations

$$e_2(P) = -(P_{1y} - P_{0y})(P_x - P_{0x}) + (P_{1x} - P_{0x})(P_y - P_{0y})$$

$$e_2(P) = \mathbf{n_2} \cdot (P - P_0) = \|n_2\| \|P - P_0\| cos\theta$$

$$e_2(P) = bh$$



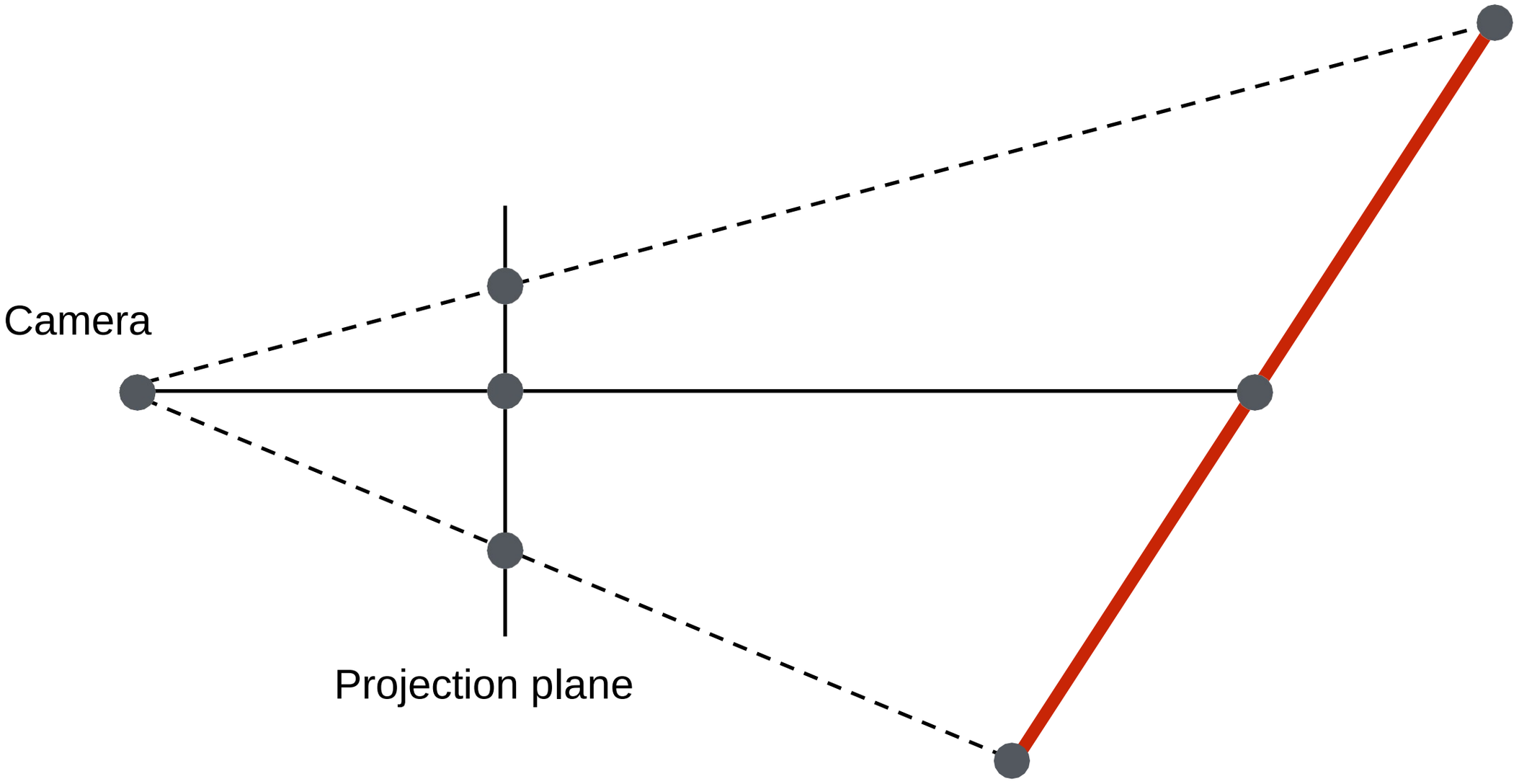
The edge function returns twice the area

$$\alpha = \frac{e_0(x,y)}{2A_\triangle}$$

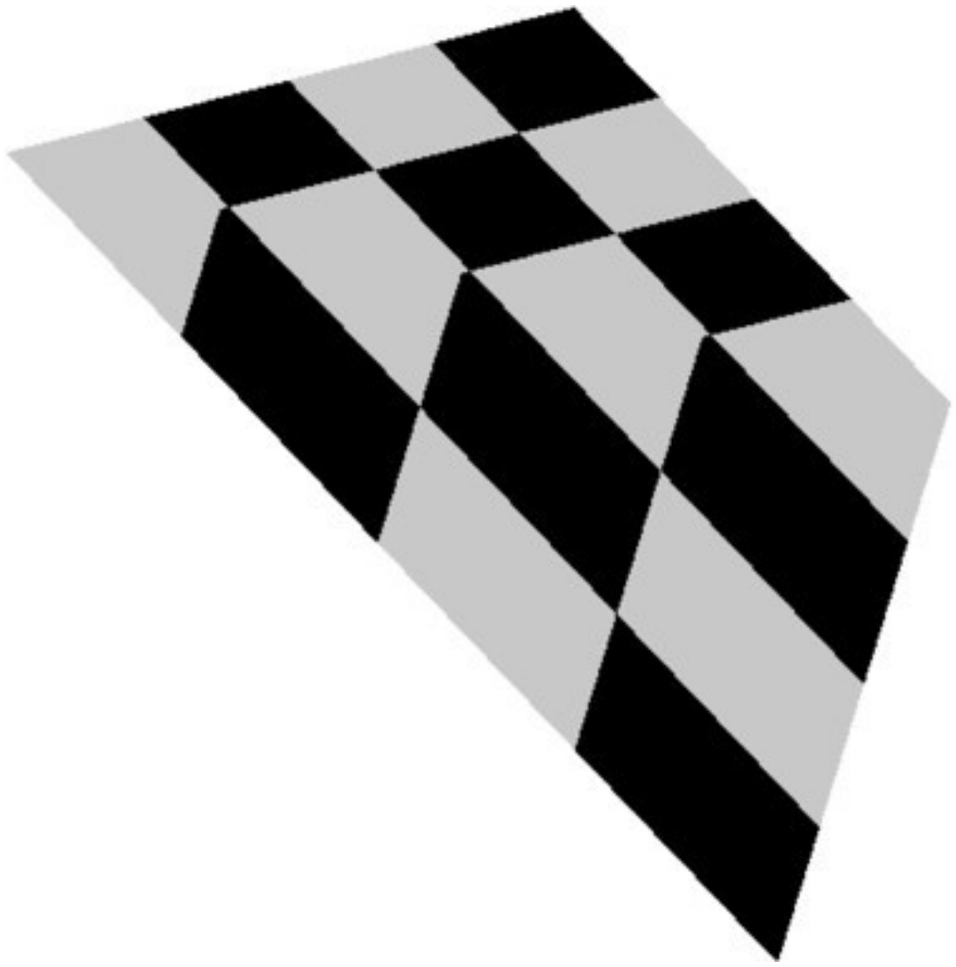
$$\beta = \frac{e_1(x,y)}{2A_\triangle}$$

$$\gamma = \frac{e_2(x,y)}{2A_\triangle}$$

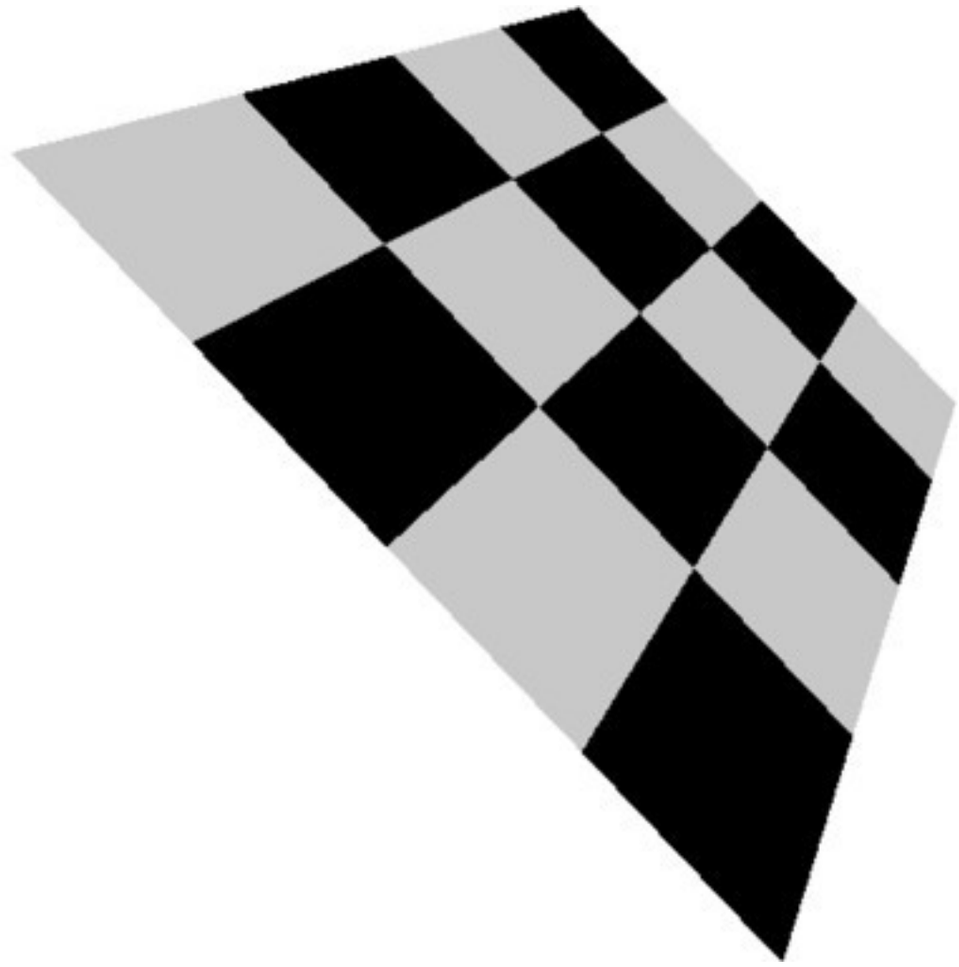
Attribute interpolation



Attribute interpolation



without perspective correction



with perspective correction

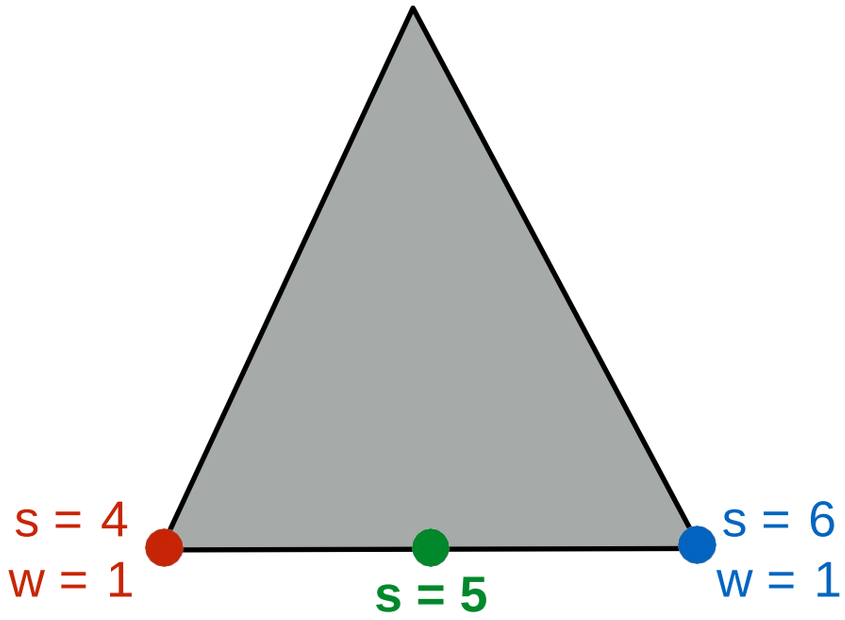
Perspective-Correct Interpolation Coordinates

- A set of coordinates, that are similar to the barycentric coordinates, but computed with perspective in mind
- Note that these coordinates are not proportional to the areas of the sub-triangles
- Requires a division per pixel
- Assume that each vertex has a parameter **s**
- Linearly interpolate **s/w** and **1/w**

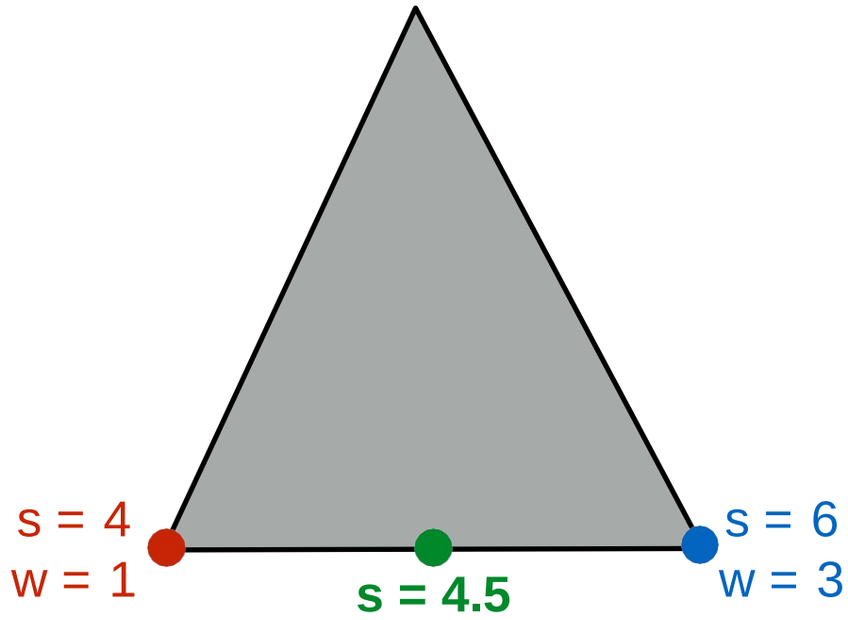
$$\mathbf{s} = \frac{\mathbf{IntepolatedValue(s/w)}}{\mathbf{IntepolatedValue(1/w)}}$$

Example

Orthographic projection



Perspective projection



Interpolate s/w: left
corner = 4/1
right corner = 6/3
midpoint value = 3

Interpolate 1/w: left
corner = 1/1
right corner = 1/3
midpoint value = 2/3

midpoint value = 4.5

Perspective-Correct Interpolation Coordinates

$$f_0(x,y) = \frac{e_0(x,y)}{w_0}$$

$$f_1(x,y) = \frac{e_1(x,y)}{w_1}$$

$$f_2(x,y) = \frac{e_2(x,y)}{w_2}$$

Auxiliary functions

$$\widetilde{\alpha} = \frac{f_0(x,y)}{f_0(x,y) + f_1(x,y) + f_2(x,y)}$$

$$\widetilde{\beta} = \frac{f_1(x,y)}{f_0(x,y) + f_1(x,y) + f_2(x,y)}$$

$$\widetilde{\gamma} = \frac{f_2(x,y)}{f_0(x,y) + f_1(x,y) + f_2(x,y)}$$

Barycentric Perspective-Correct Interpolation Coordinates

Triangle setup and per-pixel computations

$a_i, b_i, c_i, i \in [1,2,3]$	Edge functions
$\frac{1}{2A_{\triangle}}$	Half reciprocal of triangle area
$\frac{1}{w_i}, i \in [1,2,3]$	Reciprocal of w-coordinates

Triangle setup computations

$e_i(x, y)$	Evaluate edge functions at (x, y)
α, β, γ	Barycentric coordinates
$d(x, y)$	Per-pixel depth
$f_i(x, y)$	Evaluation of per-pixel f-values
$\widetilde{\alpha}, \widetilde{\beta}, \widetilde{\gamma}$	Perspectively-correct interpolation coordinates
$s(x, y)$	Interpolation of all desired parameters

Per-pixel computations

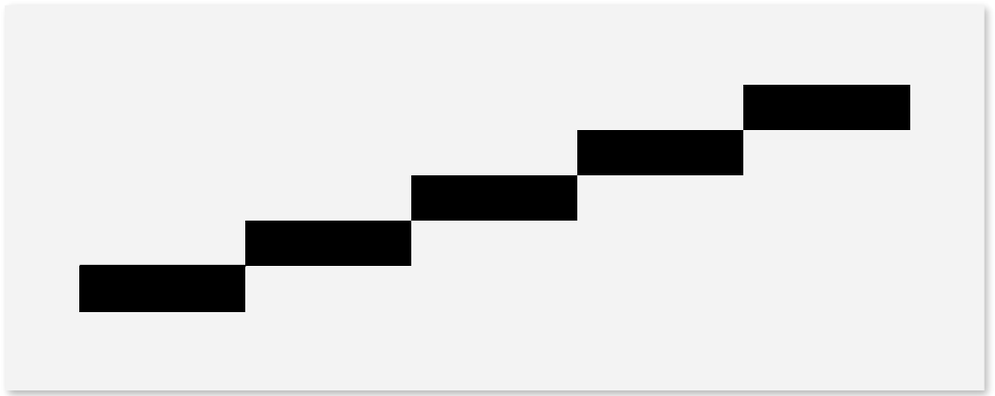
Antialiasing

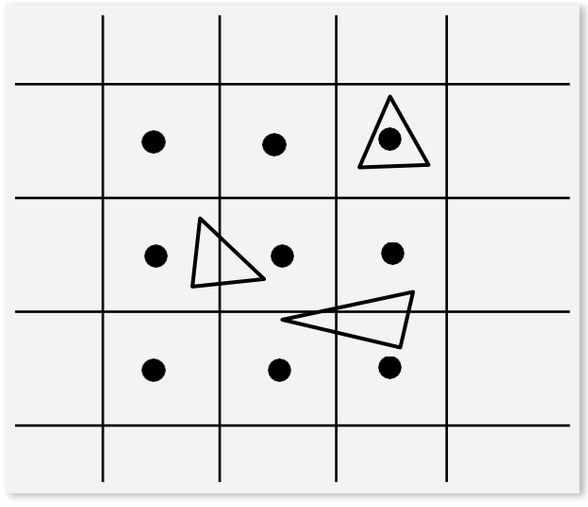
- Removing the stair-step appearance of a line
- Need some compensation in line-drawing algorithm
- Jugged edges
- Small objects

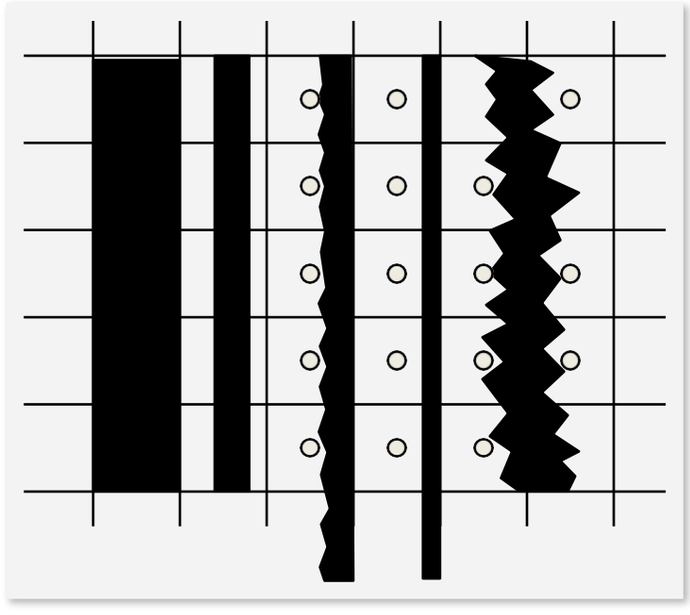
  unvisible, disturbed

- Textures

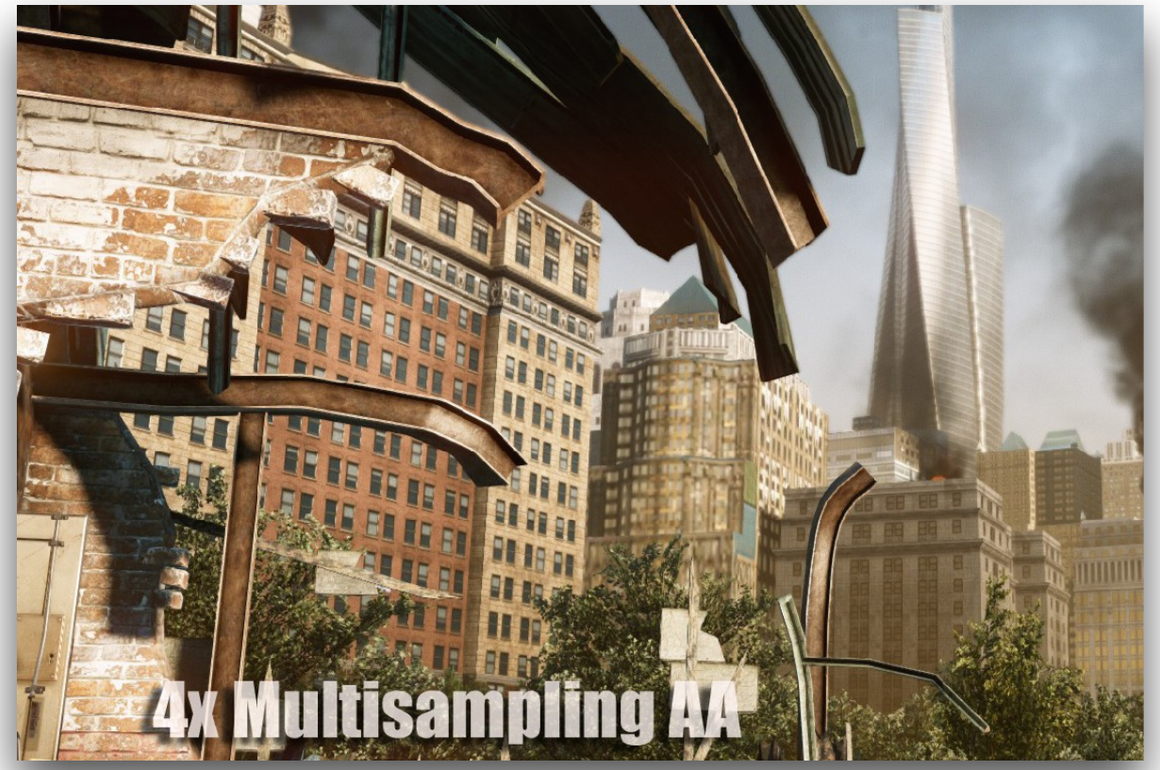
  toothed edges







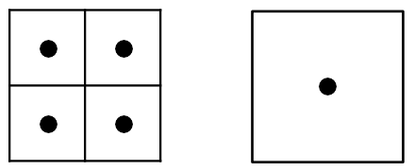
Antialiasing

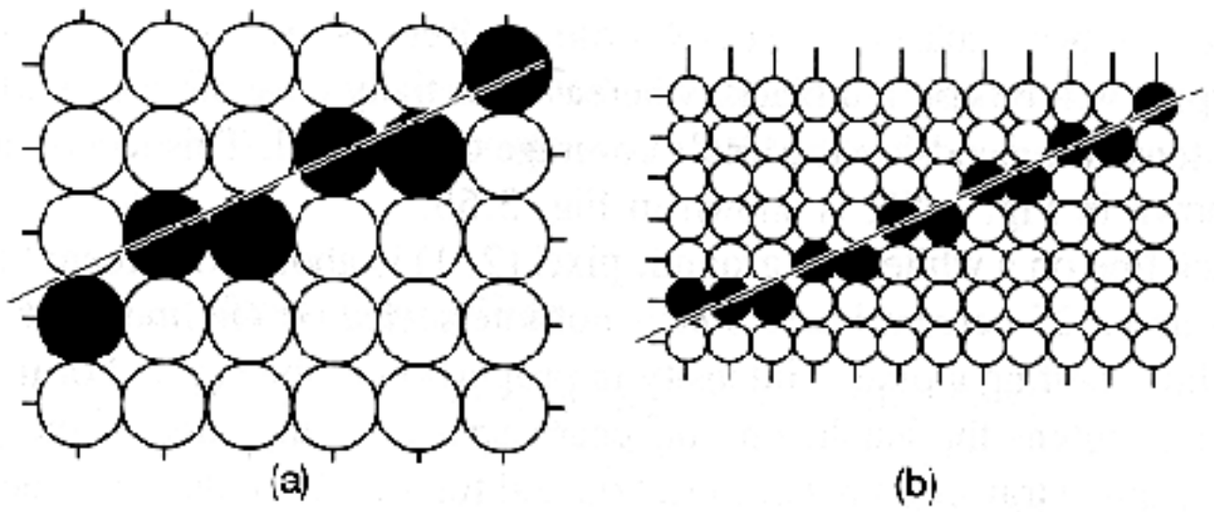


[http://images.pcworld.com/images/article/2011/04/aavs4xms-5159947.jpg]

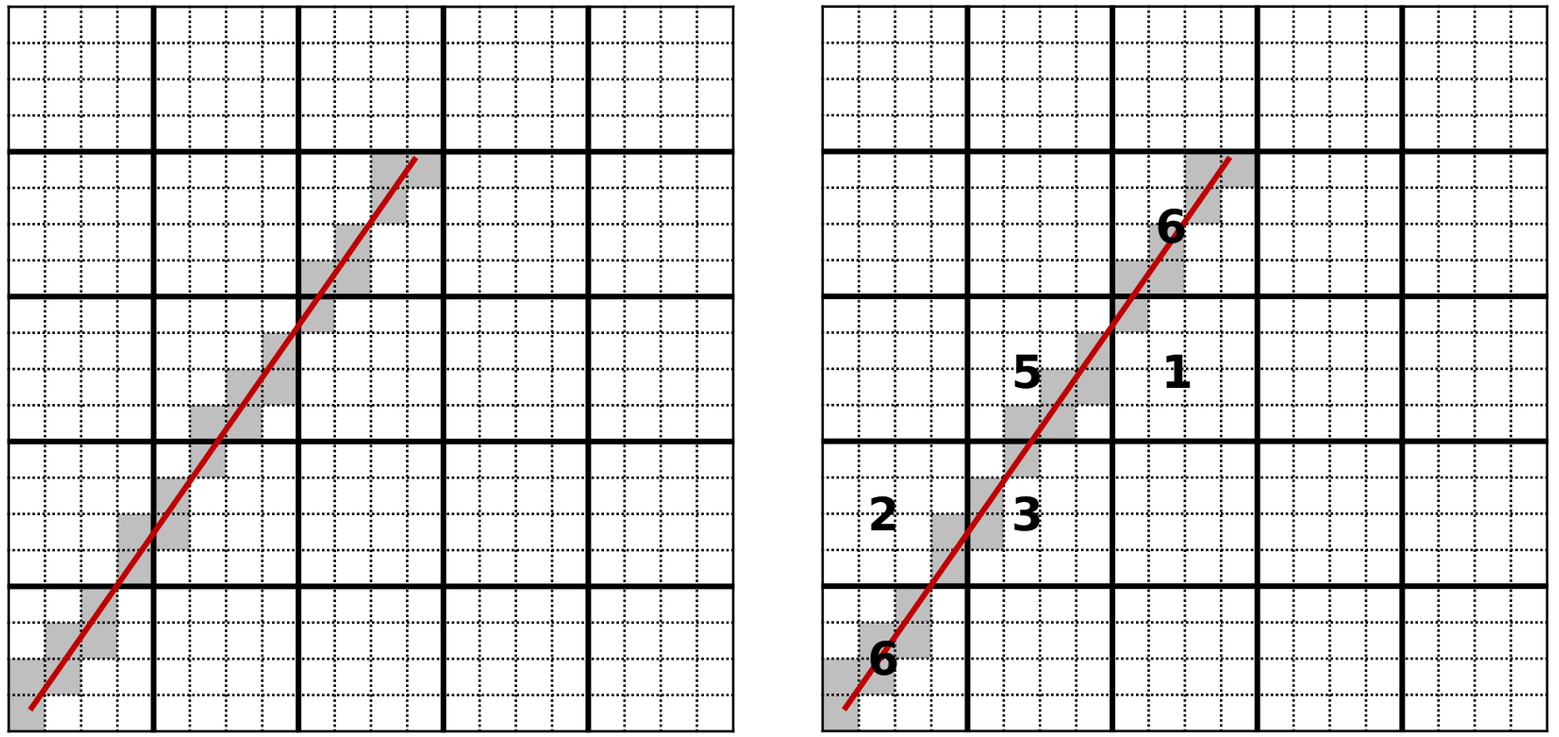
Supersampling (Postfiltering)

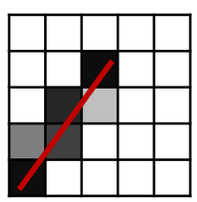
- Increase resolution





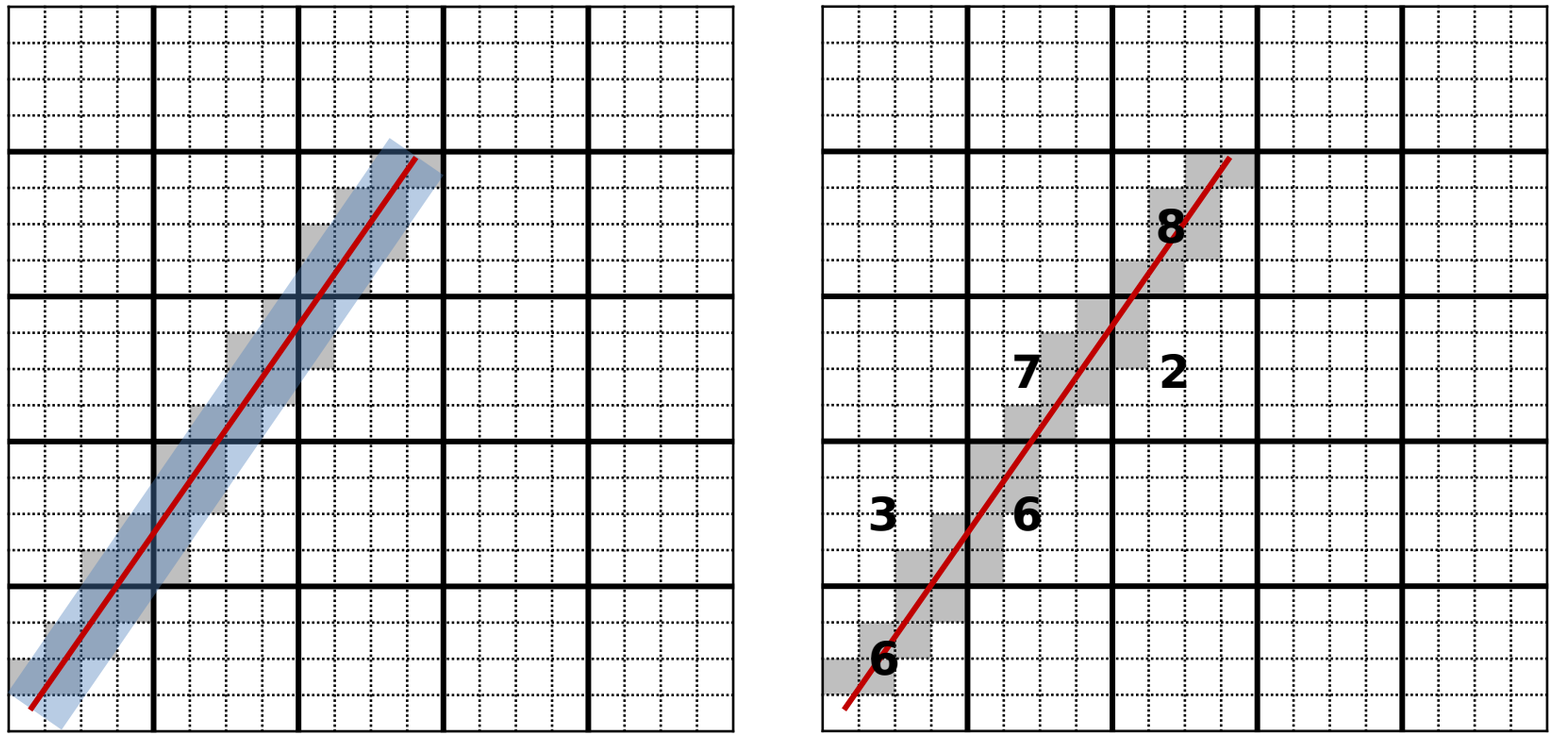
Supersampling (Postfiltering)

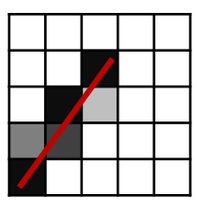




The intensity of each pixel depends on the number of subpixels intersected by the line

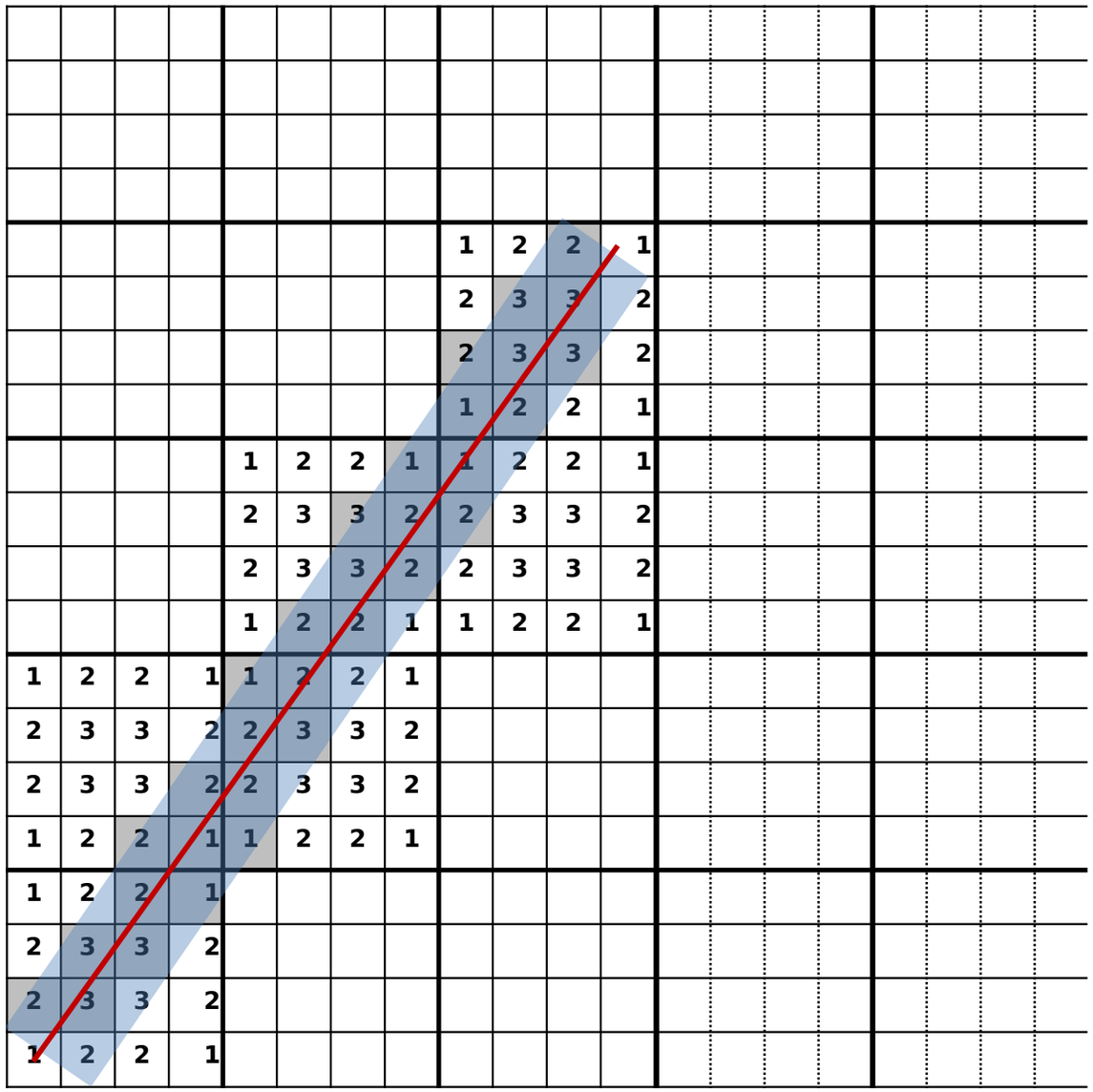
Supersampling (Postfiltering)





The intensity of each pixel depends on the number of subpixels that are inside the rectangle (their center is inside the rectangle

Weighted supersampling

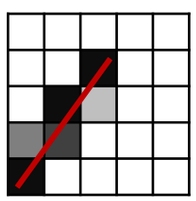


1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

Unweighted

1	2	2	1
2	3	3	2
2	3	3	2
1	2	2	1

Weighted



Assign weights to each subpixel (higher values near the center of the pixel)