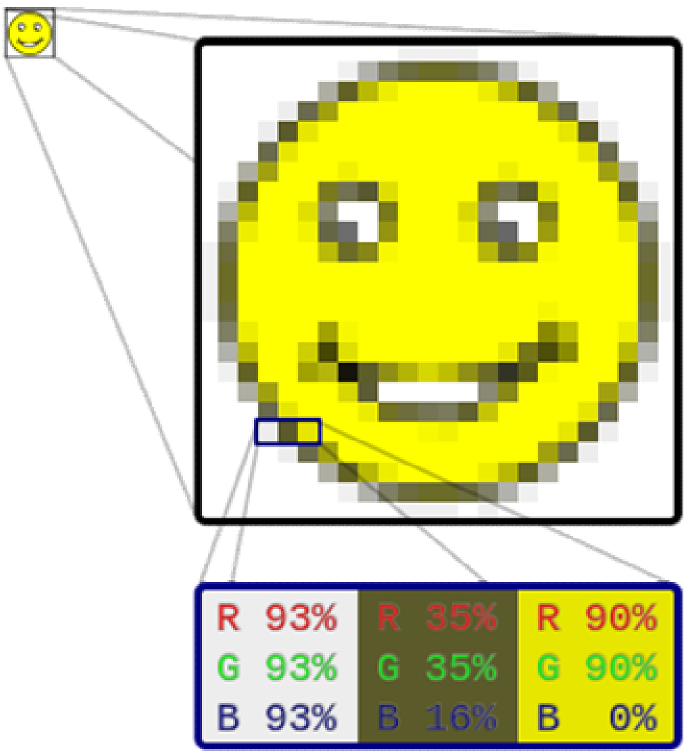
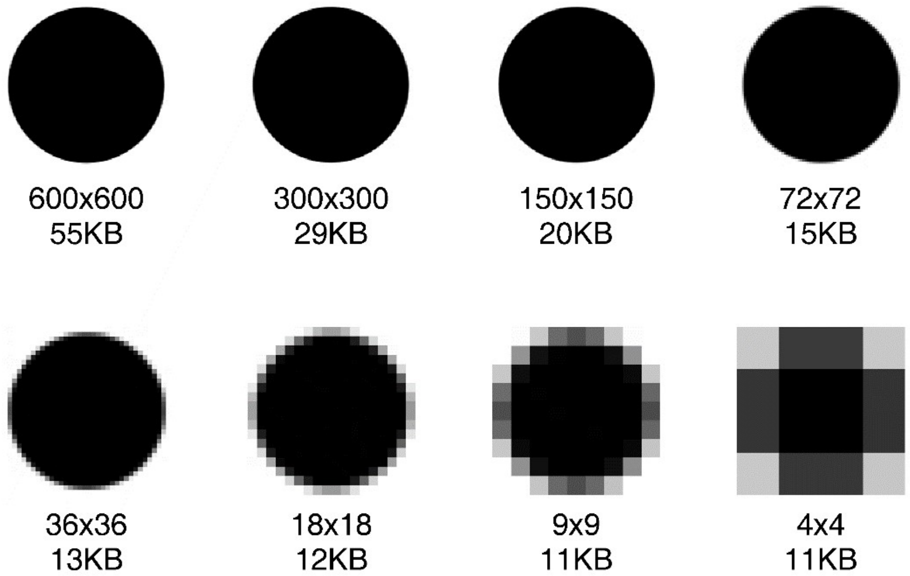
Raster Graphics

Raster Graphics

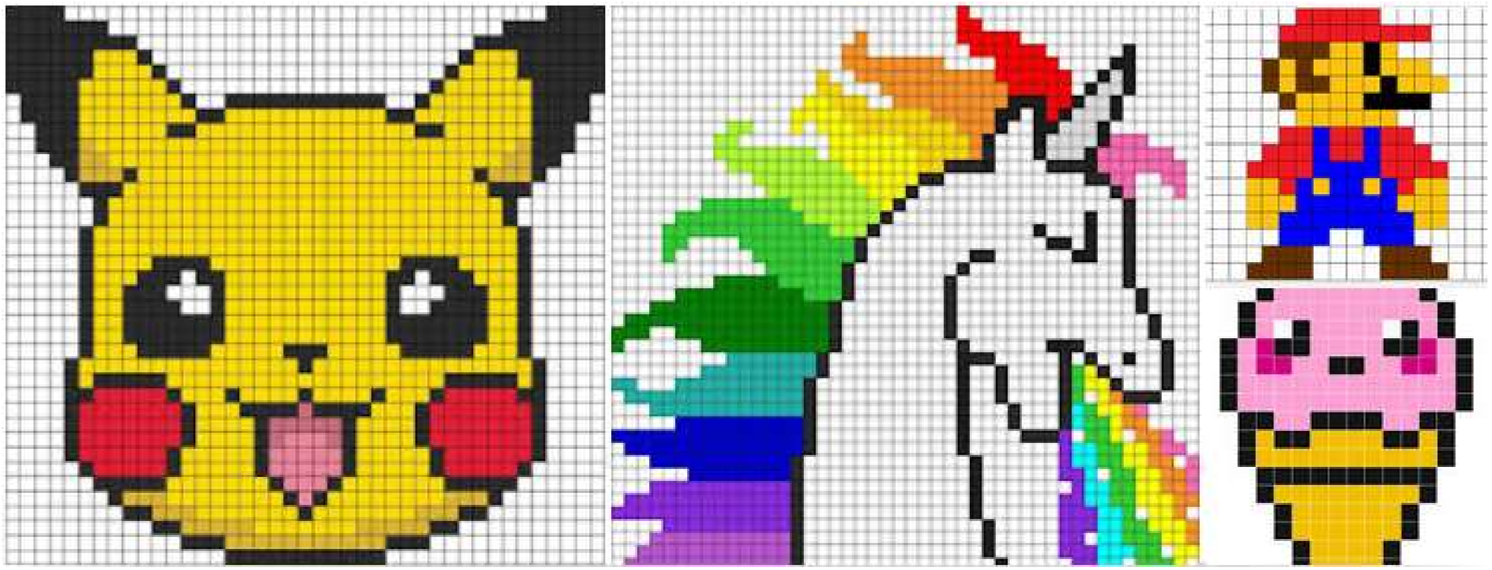
- Raster Image = rectangular grid of colored elements
- Higher realisme = higher memory requirements





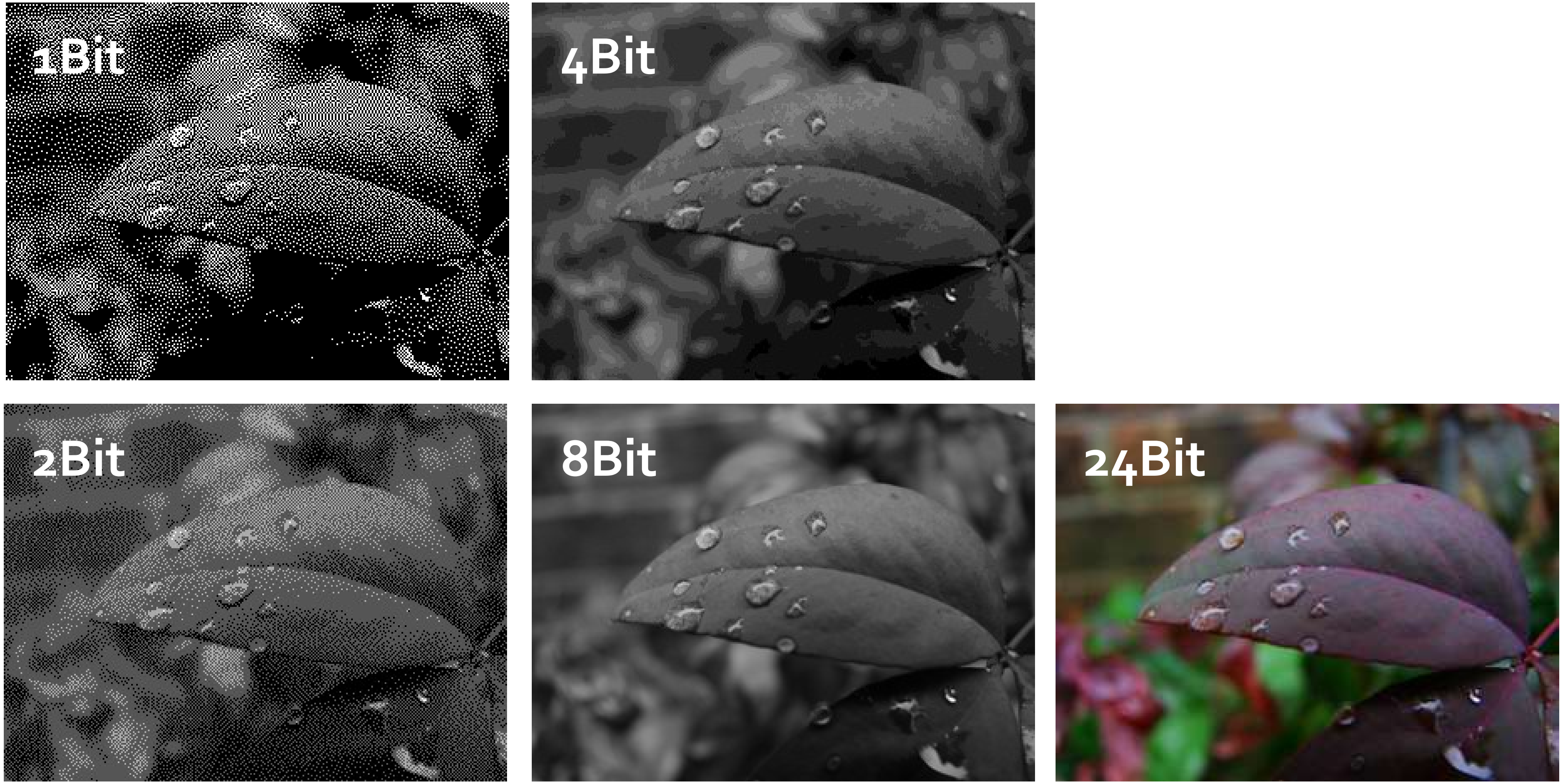
Pixel

- “Picture element”
- Physical point in a raster image
- Certain amount of bits per pixel



Bits per Pixel

- Amount of bits used to store color information

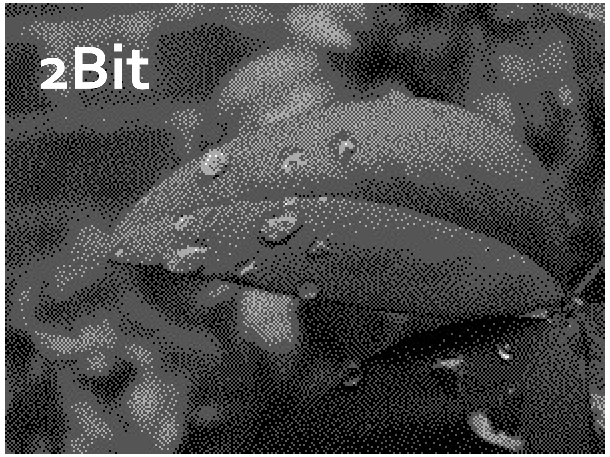


Bits per Pixel

0
1
2
3

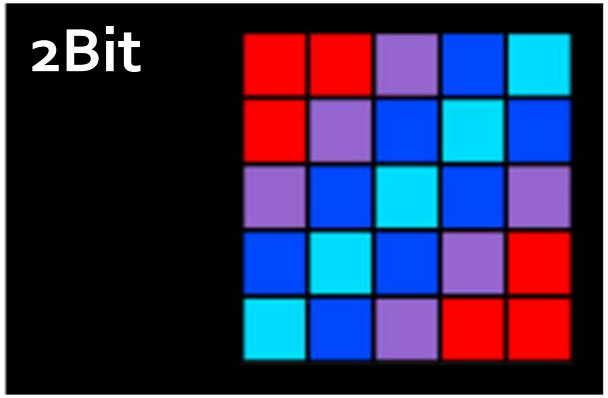
0
5
230
255

8bit	8bit	8bit
0	0	0
255	0	0
0	255	255
...	...	...



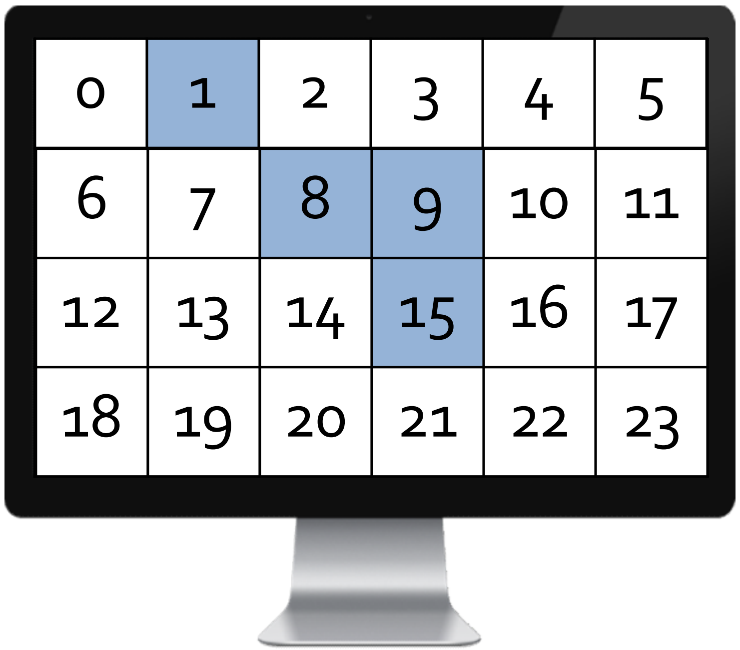
Color Tables

0	0	1	2	3
0	1	2	3	2
1	2	3	2	1
2	3	2	1	0
3	2	1	0	0



Frame Buffer

- A.k.a. frame store
- Raster image of monitor input
- Portion of RAM (often in video memory)
- RAM is usually 1 dimensional and linear

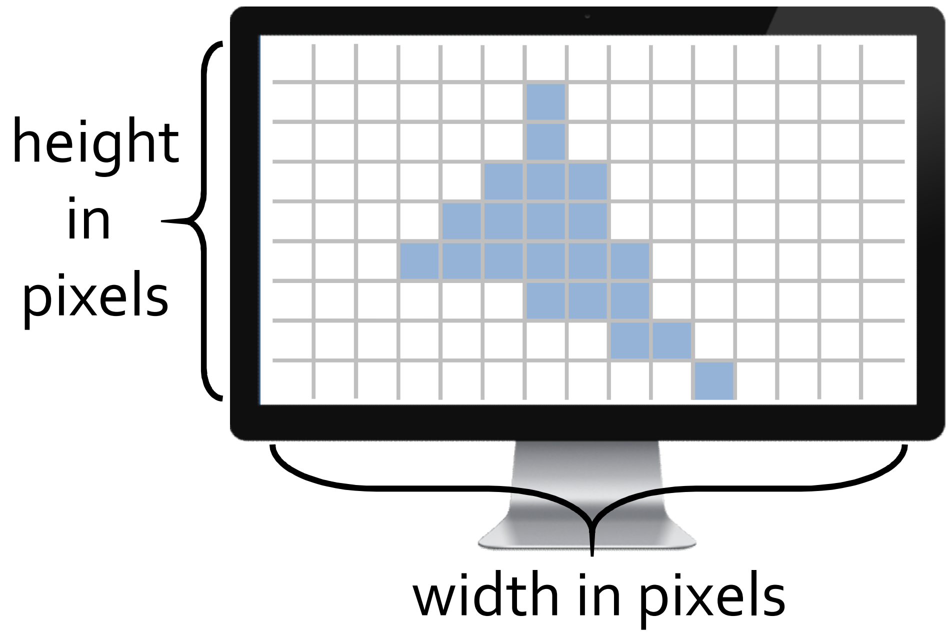


RAM (8Bit color)

#	Data
0	255
1	120
2	255
3	255
4	255
5	255
6	255
7	255
8	120
...	

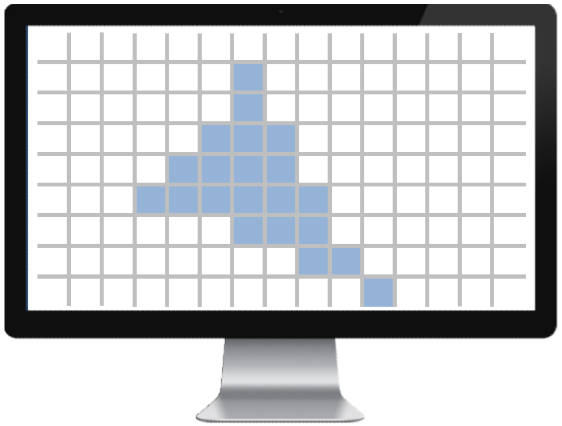
Frame Buffer

- A.k.a. frame store
- Portion of RAM (often in video memory)
- Raster image of monitor input
- Resolution
  - Width x height of pixels
  - VGA = 640*480
  - XGA= 1024*768
  - HD=1280*720
  - FullHD = 1920*1080



Frame Buffer Resolution

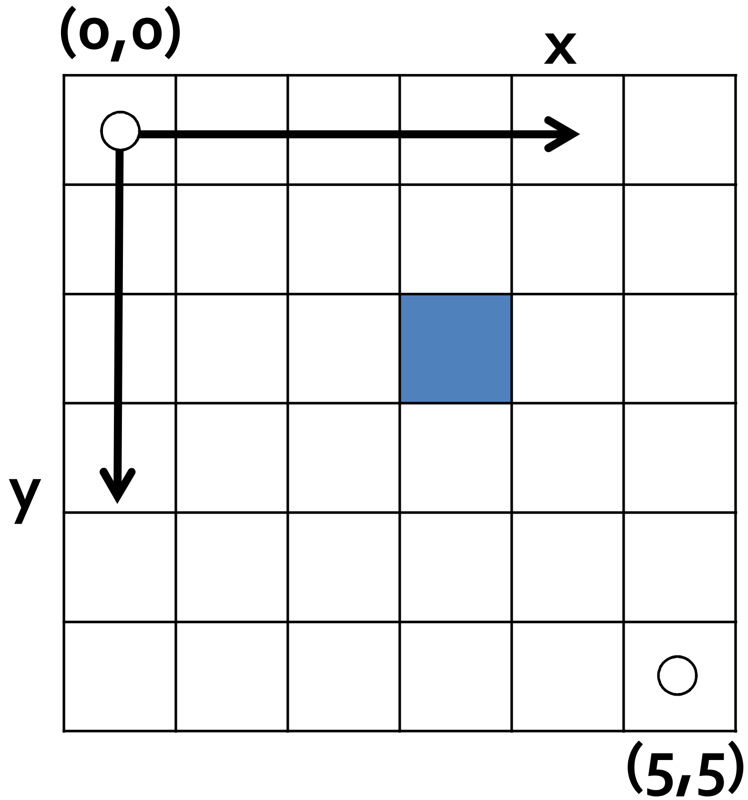
- Width x height of pixels
- VGA = 640*480, 8bit per pixel
  - 640*480*1 = 307KB
- XGA= 1024*768, 16bit per pixel
  - 1024*768*2= 1,5MB
- HD=1280*720, 24bit per pixel
  - 1280*720*3 = 2,6MB
- FullHD = 1920*1080, 32bit per pixel
  - 1920*1080*4= 8MB
- 4k = 3840*2160, 32bit per pixel
  - 3840*2160 *4= 32MB



Drawing Objects

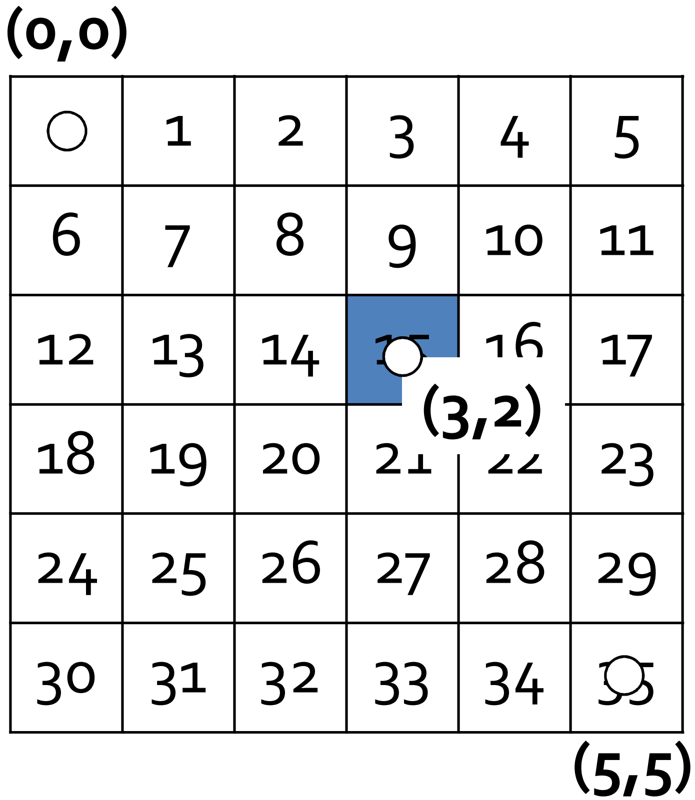
Drawing a Pixel

- Given is a pixel by coordinates and color
  `DrawPixel(x, y, color)`

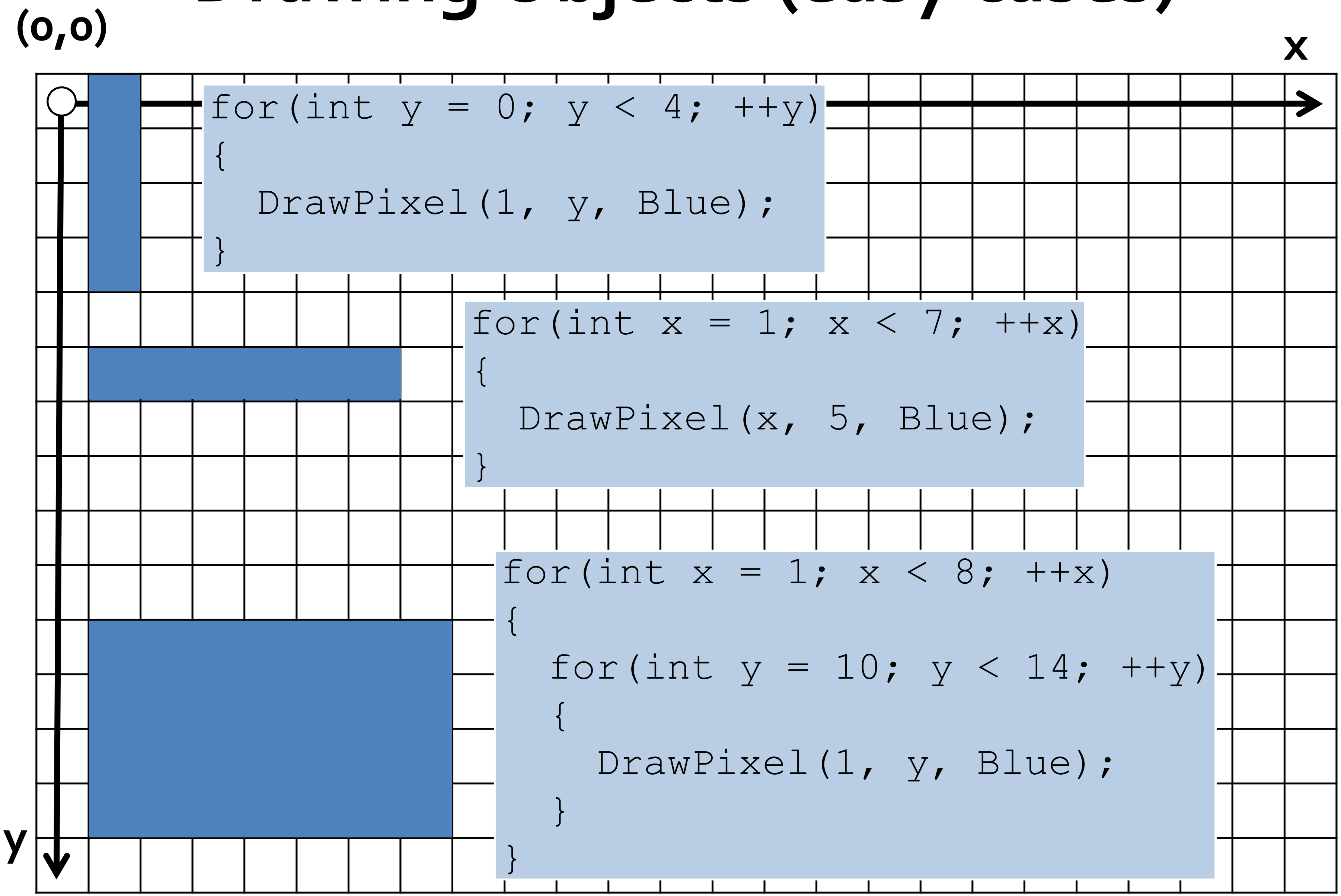


Drawing a Pixel

- Color assignment to location (memory address) in frame buffer
  `frameBuffer[addr] = BLUE;`
- Calculate address?
  `addr = y * width + x`



Drawing Objects (easy cases)

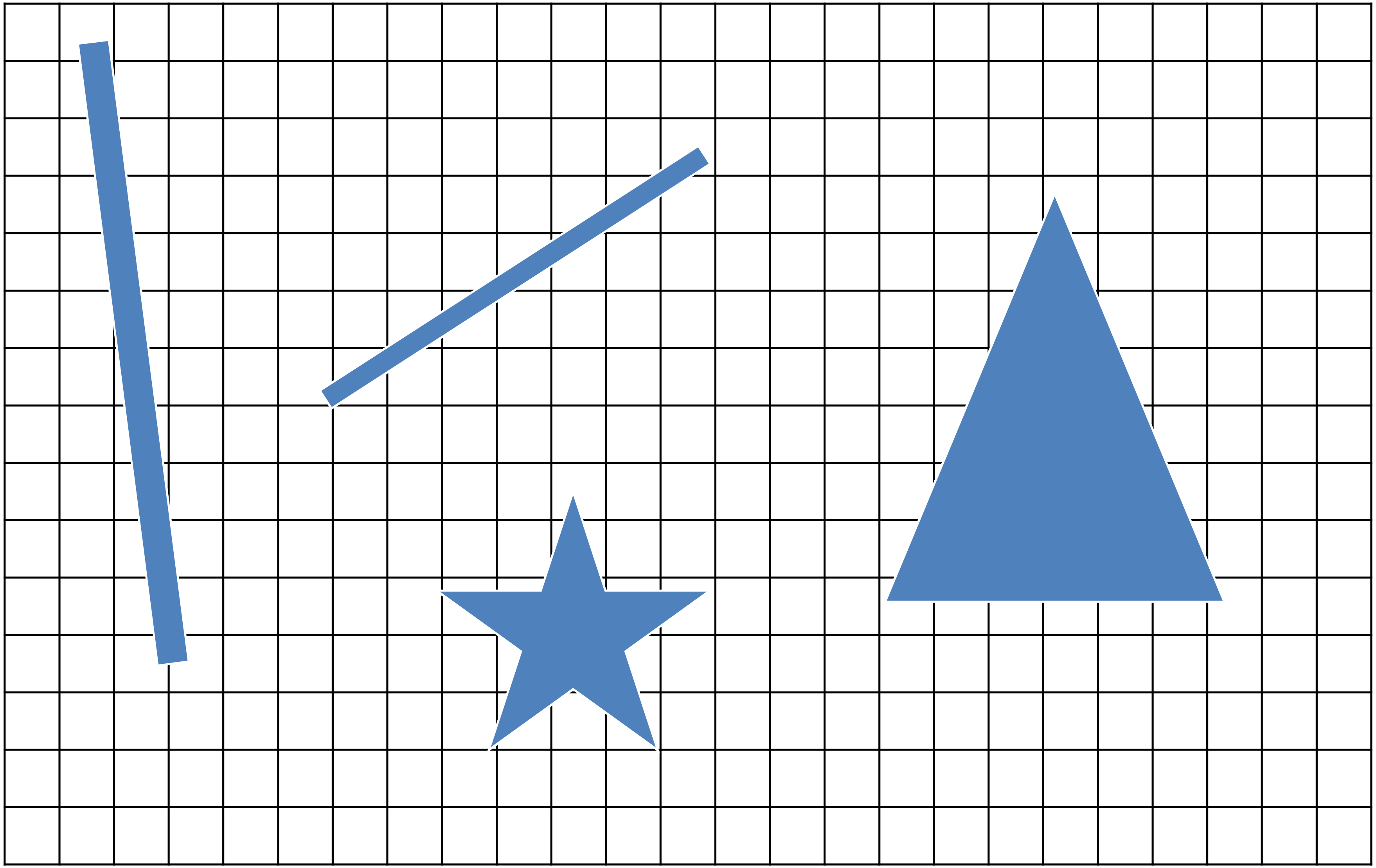



```
for(int y = 0; y < 4; ++y)
{
  DrawPixel(1, y, Blue);
}
```

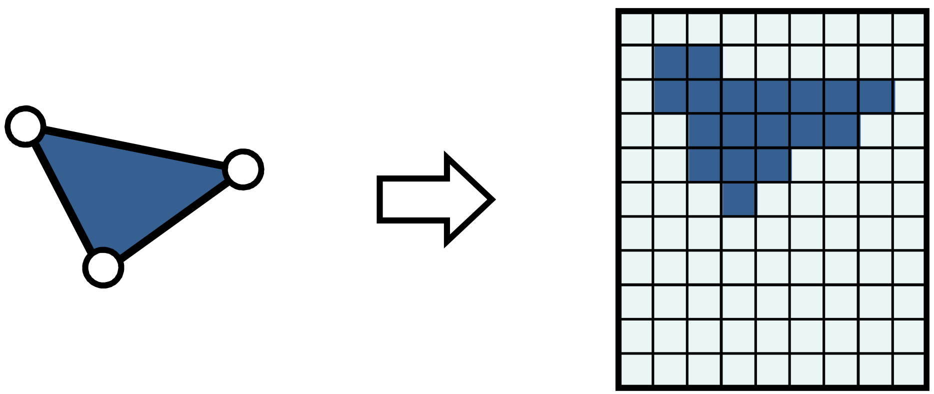
```
for(int x = 1; x < 7; ++x)
{
  DrawPixel(x, 5, Blue);
}
```

```
for(int x = 1; x < 8; ++x)
{
  for(int y = 10; y < 14; ++y)
  {
    DrawPixel(1, y, Blue);
  }
}
```

Drawing Objects (normal cases)

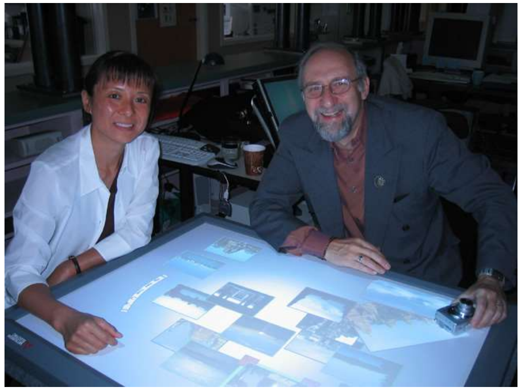


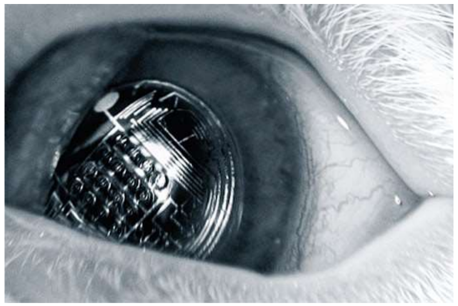
Rasterization



Rasterization

- Want to do vector graphics on a raster device



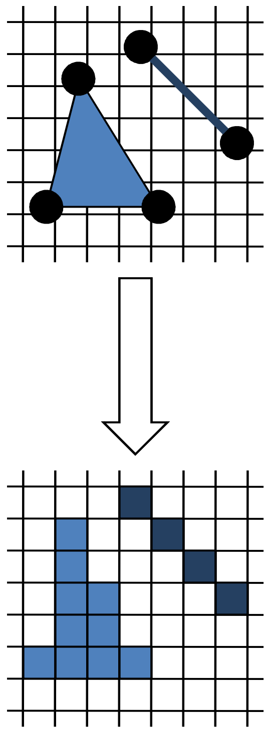


Output Primitives

- 2D
  - Points, lines
  - Polygons, circles, ellipses & other curves (also filled)
  - Characters (text)
- 3D
  - Triangles, polygons
  - Free form surfaces

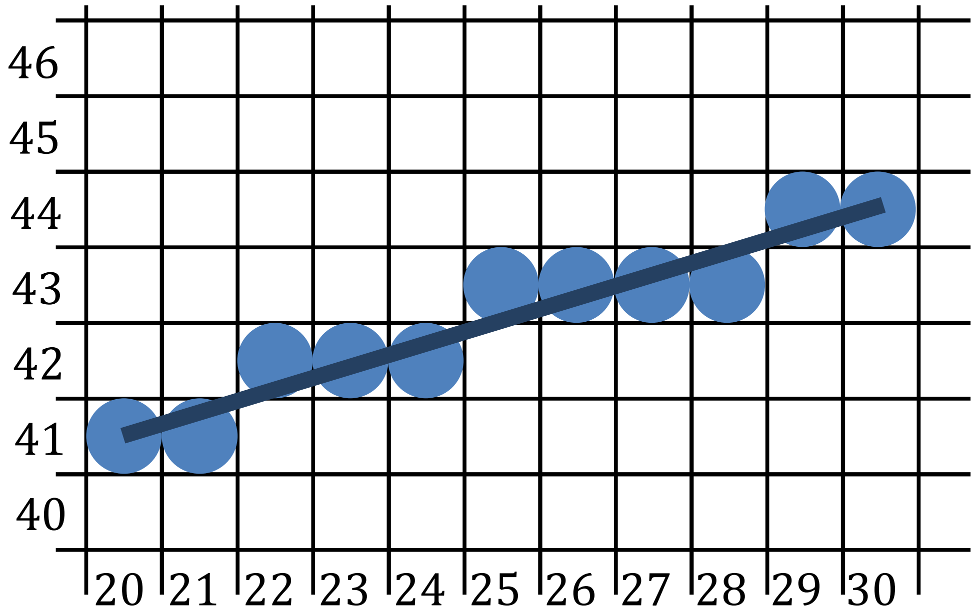
Rasterization

- Converts
  - Primitives
  - With floating point vertices



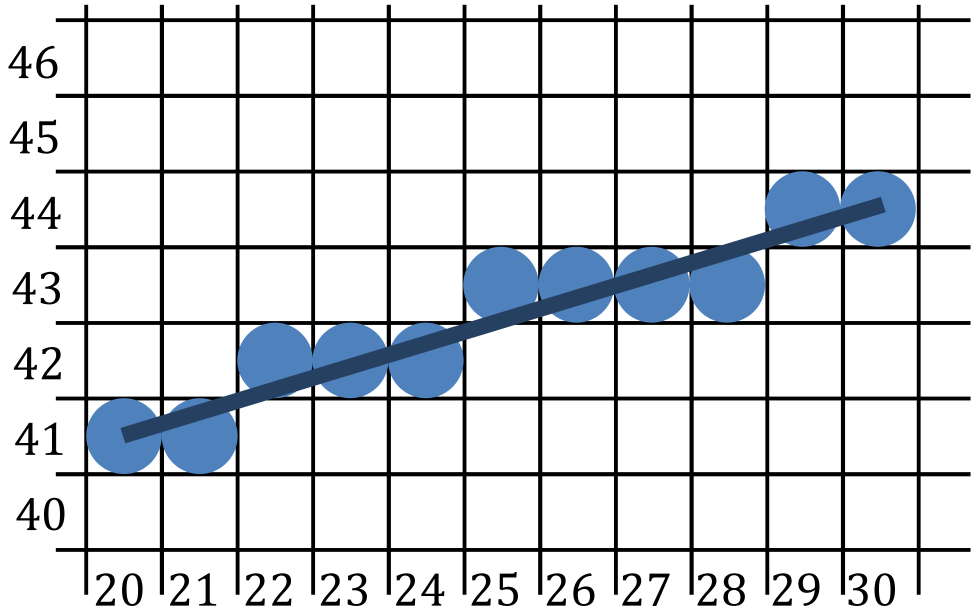
- into
  - Pixels
  - With integer coordinates

Rasterization of Lines



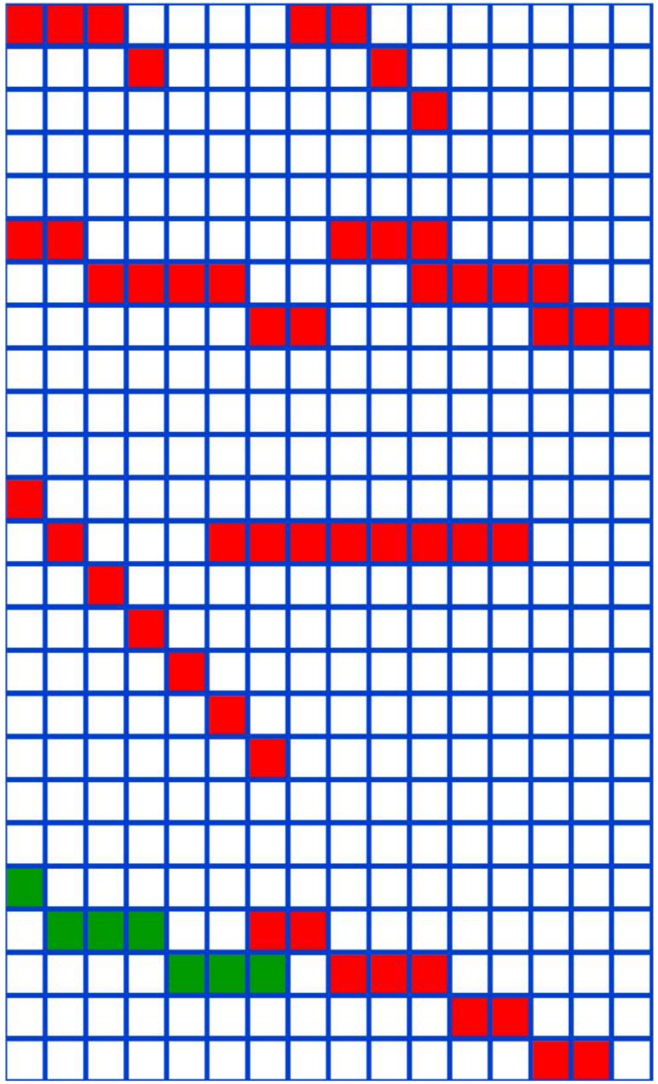
Drawing Lines

- Line is a series of pixel positions
- Intermediate discrete pixel positions calculated
- Staircase effect, “jaggies” (aliasing)



Raster Conversion of Lines

- Lines should appear straight
- Lines should appear uniformly bright
- Lightness should be independent of direction
- Endpoints should be "exact"

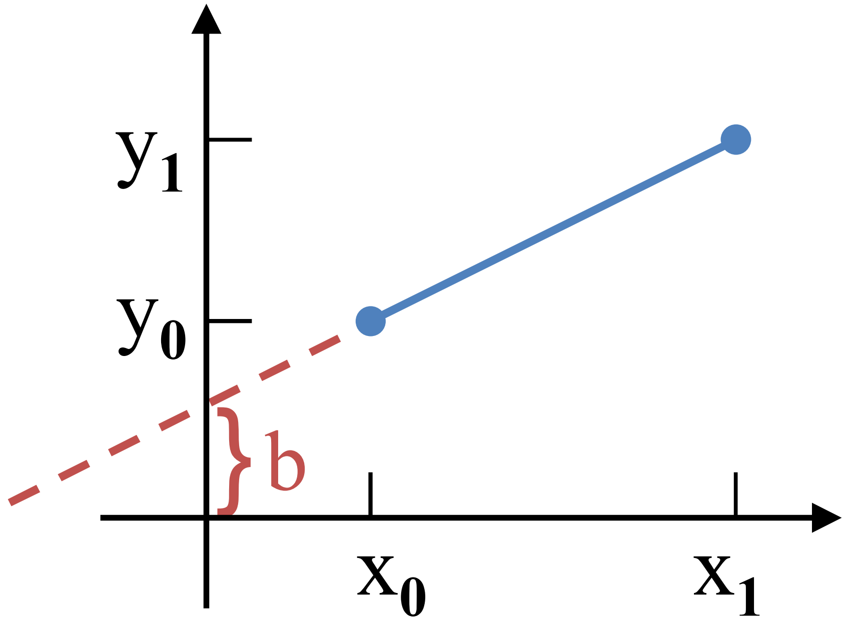


Line-Drawing Algorithms

- Line equation: $y = \mathrm{m} \cdot x + \mathrm{b}$
- Line path between two points:

$$\mathrm{m} = \frac{y_1 - y_0}{x_1 - x_0}$$

$$\mathrm{b} = y_0 - \mathrm{m} \cdot x_0$$



Example

$(x_0, y_0) = (20, 41)$
$(x_1, y_1) = (30, 44)$

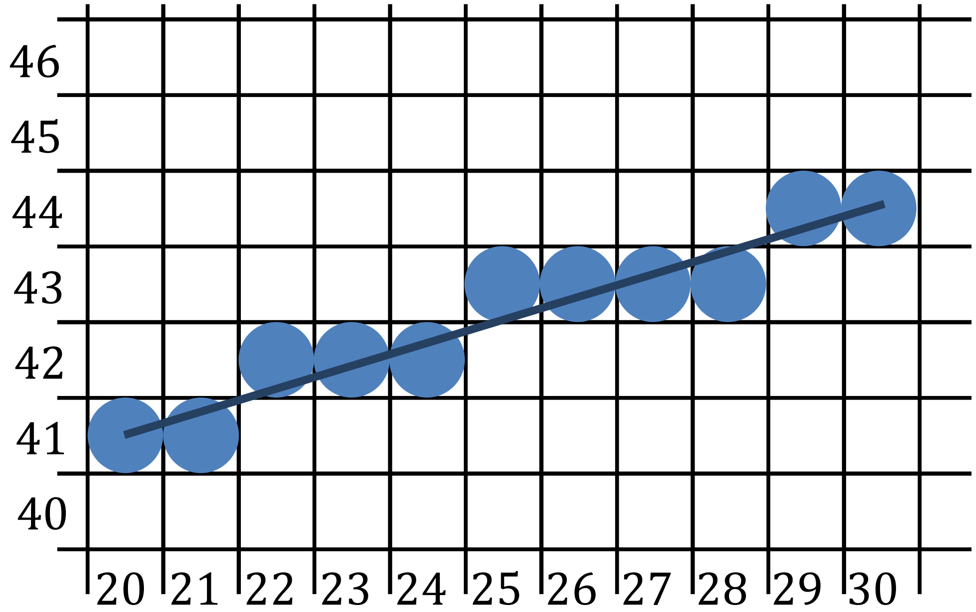
$$m = \frac{44-41}{30-20} = \frac{3}{10}$$

$$b = 41 - \frac{3}{10} \cdot 20 = 35$$

$$y = \frac{3}{10} \cdot x + 35$$

x	y
21	$\frac{413}{10} \approx 41$
22	42
23	42
24	42
25	43
26	43
27	43
28	43
29	44
30	44

Example



x	y
21	$\frac{413}{10} \approx 41$
22	42
23	42
24	42
25	43
26	43
27	43
28	43
29	44
30	44

Example 2

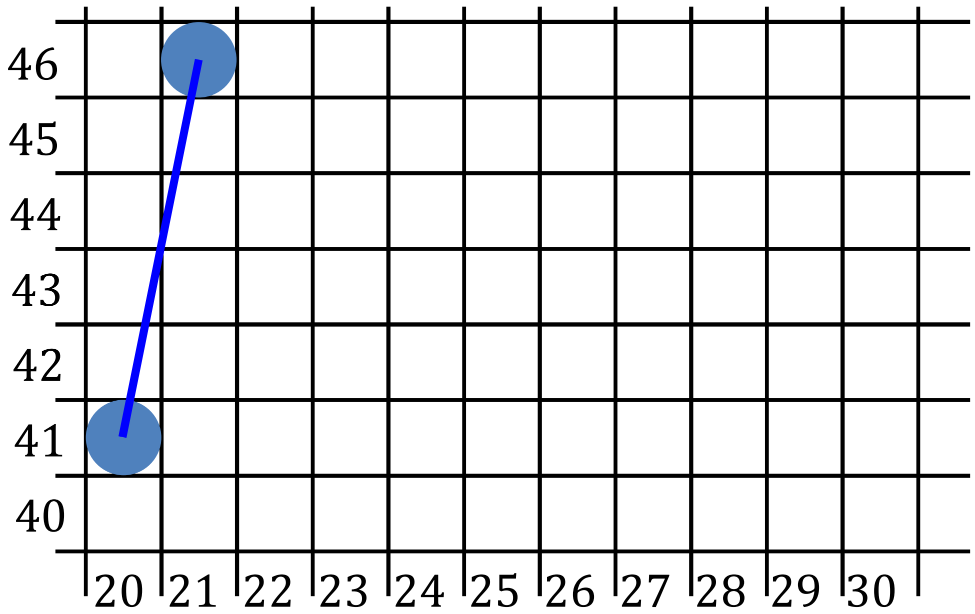
$(x_0, y_0) = (20, 41)$
$(x_1, y_1) = (21, 46)$

x	y
21	46

$$m = \frac{46-41}{21-20} = \frac{5}{1} = 5$$

$$b = 41 - 5 \cdot 20 = -59$$

$$y = 5 \cdot x - 59$$

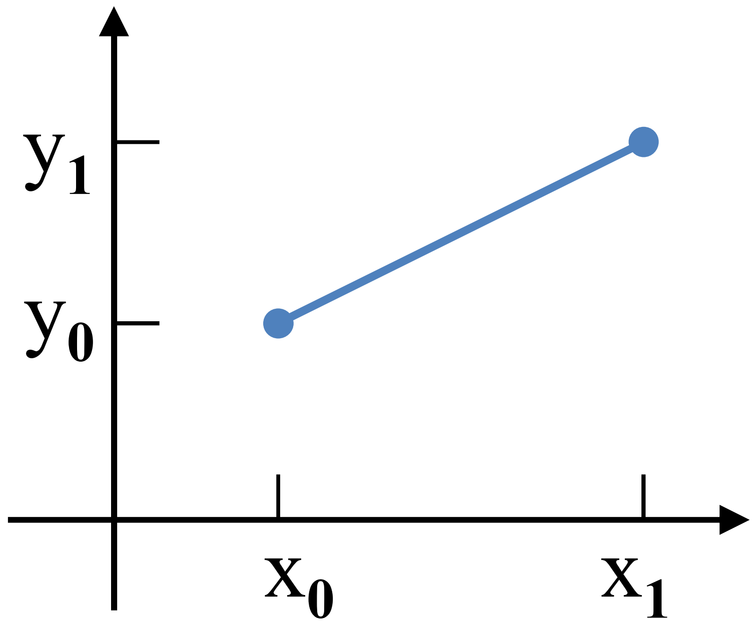


Résumé

- Quality
  - Works for some cases
    - If $m < 1$
- Performance
  - Division()
  - Round()
  - Floating point operation

DDA Line-Drawing Algorithm

- DDA (digital differential analyzer)
- Define $x_1 > x_0$ otherwise switch points
- $\Delta x = x_1 - x_0$
- $\Delta y = y_1 - y_0$
- Check if $|m| < 1$
  - Iterate along x
  - Otherwise iterate along y

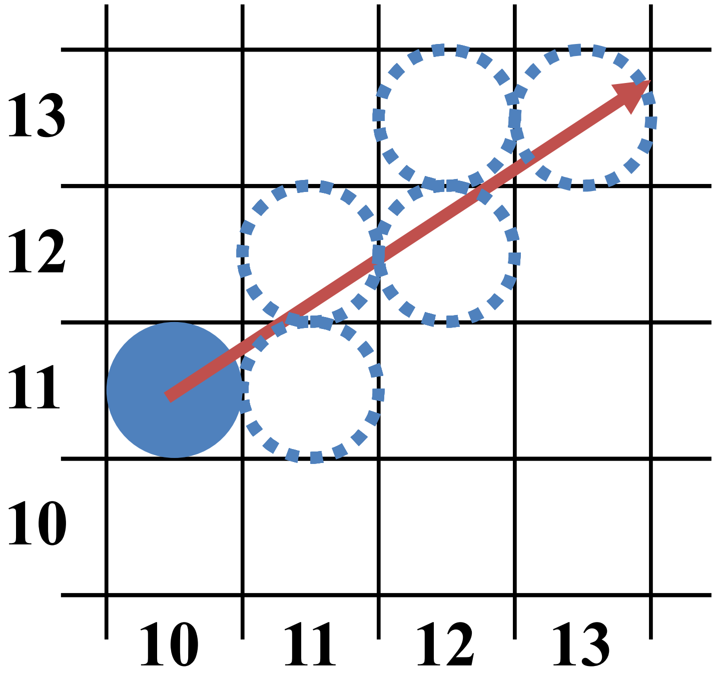


Résumé

- Quality
  - Works
- Performance
  - Division()
  - Round()
  - Floating point operation

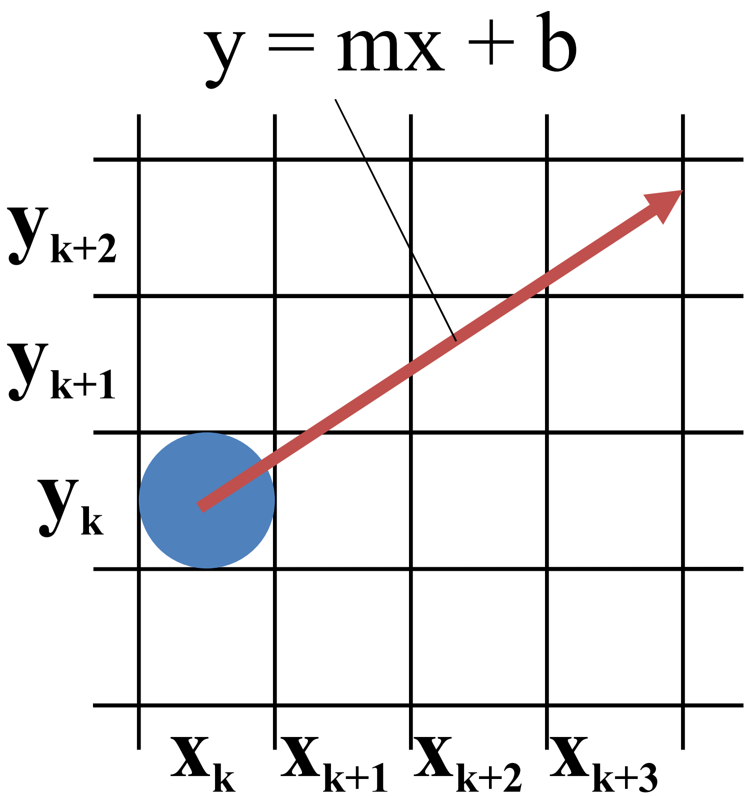
Bresenham's Line Algorithm

- Faster than simple DDA
  - Incremental integer calculations
  - Each step decision if draw upper or lower pixel



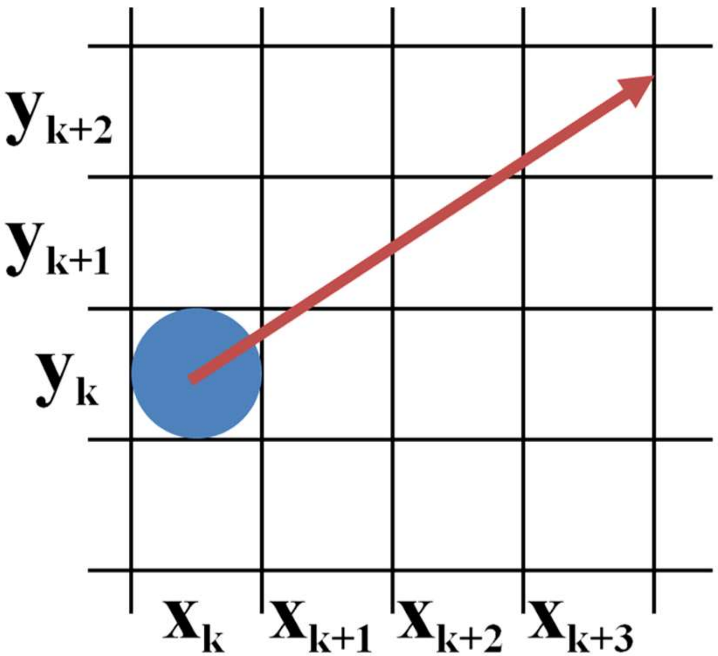
$$(x_0, y_0) = (10, 11)$$
$$(x_1, y_1) = (13, 13)$$

Bresenham's Line Algorithm

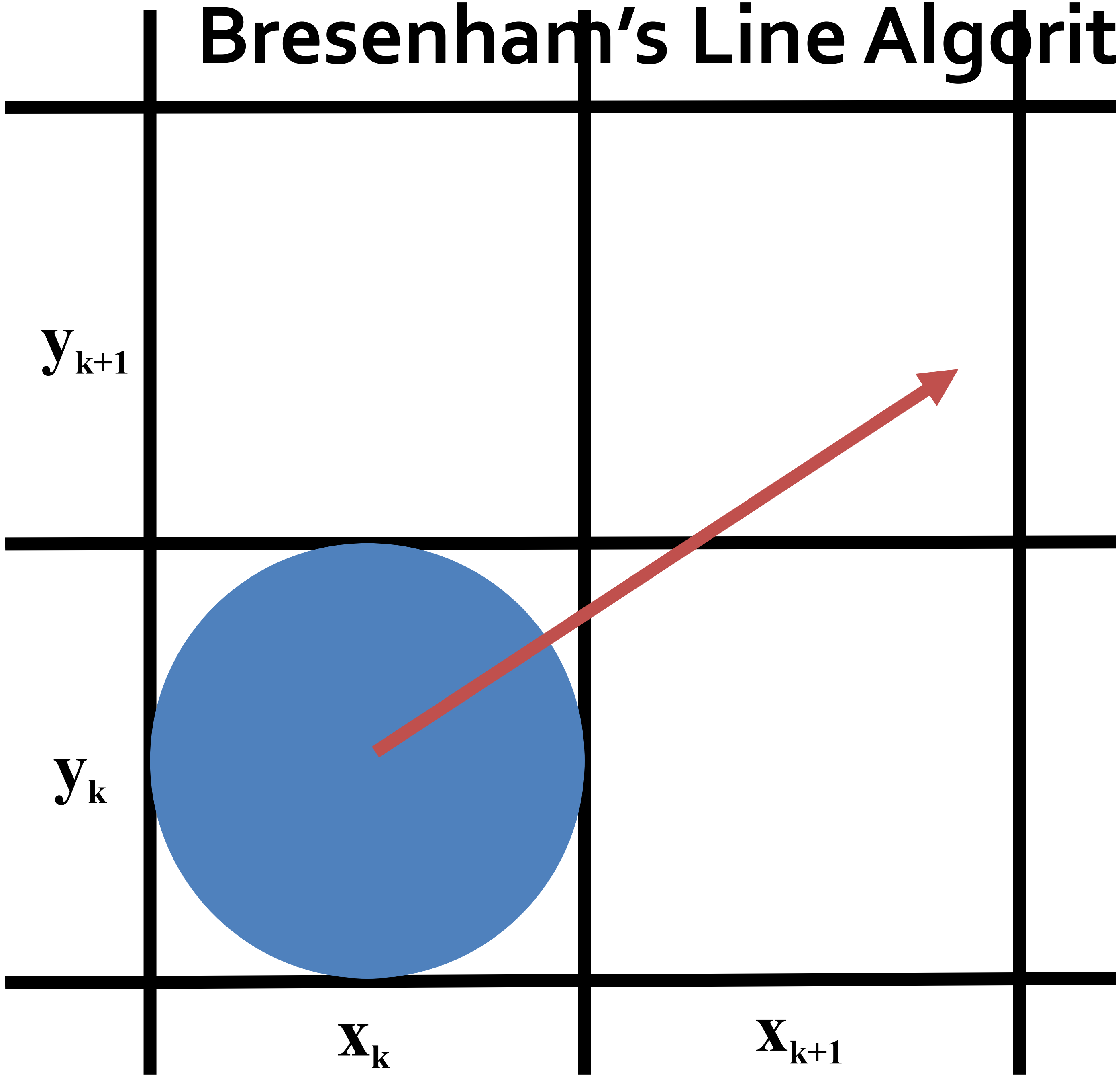


Section of the screen grid showing a pixel in column x_k on scan line y_k that is to be plotted along the path of a line segment with slope $0<m<1$

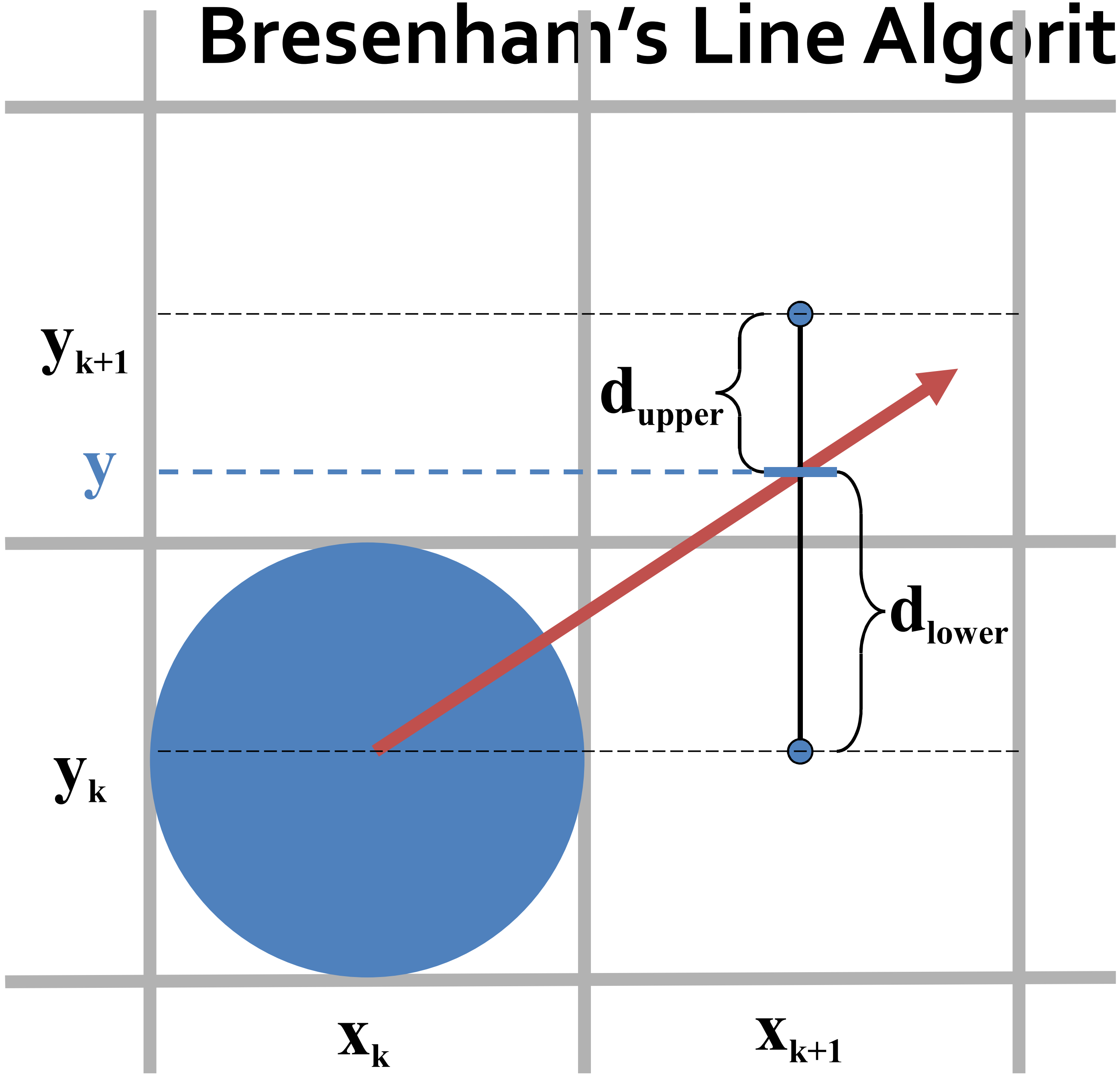
Bresenham's Line Algorithm



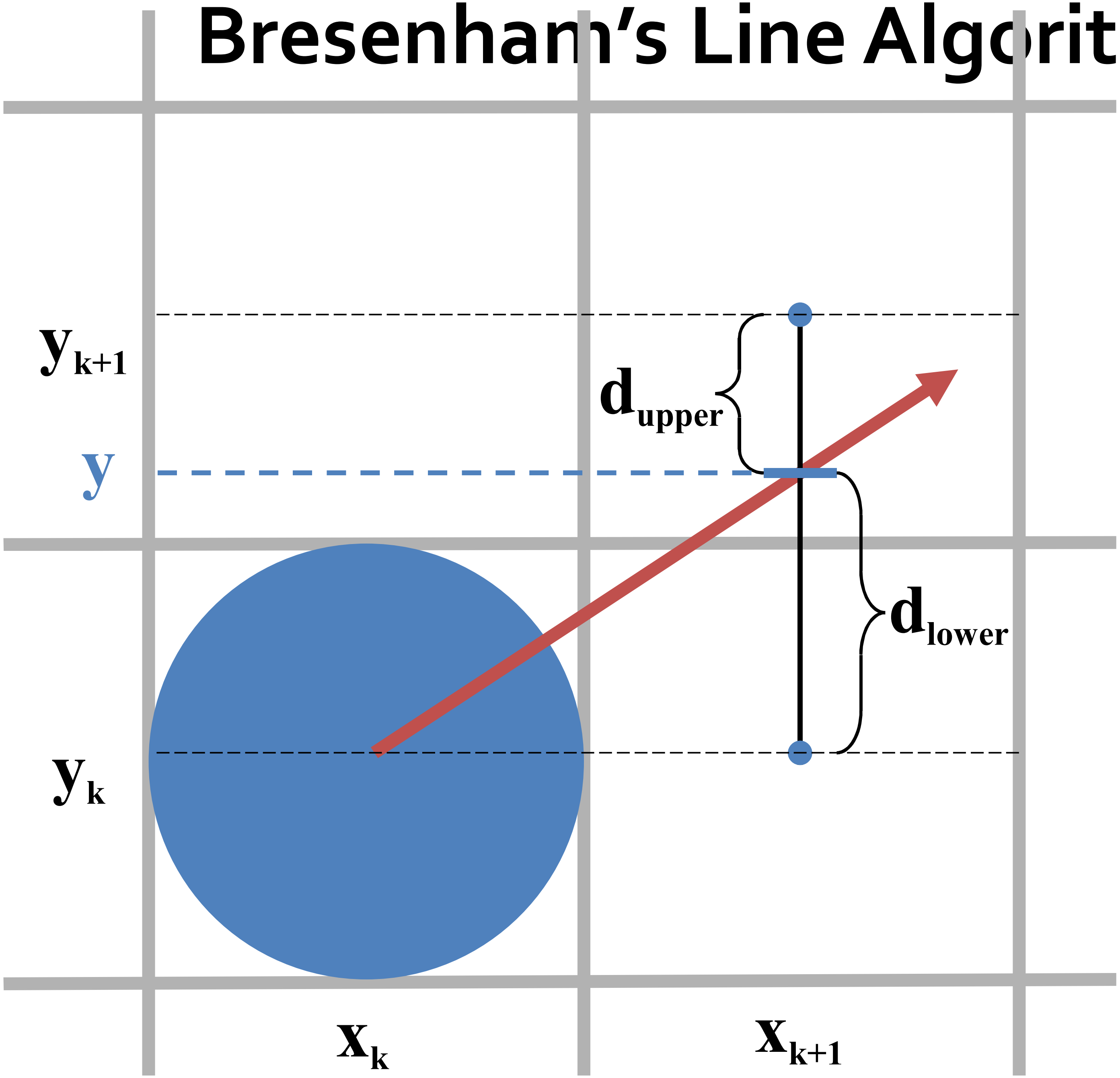
Bresenham's Line Algorithm (1/4)



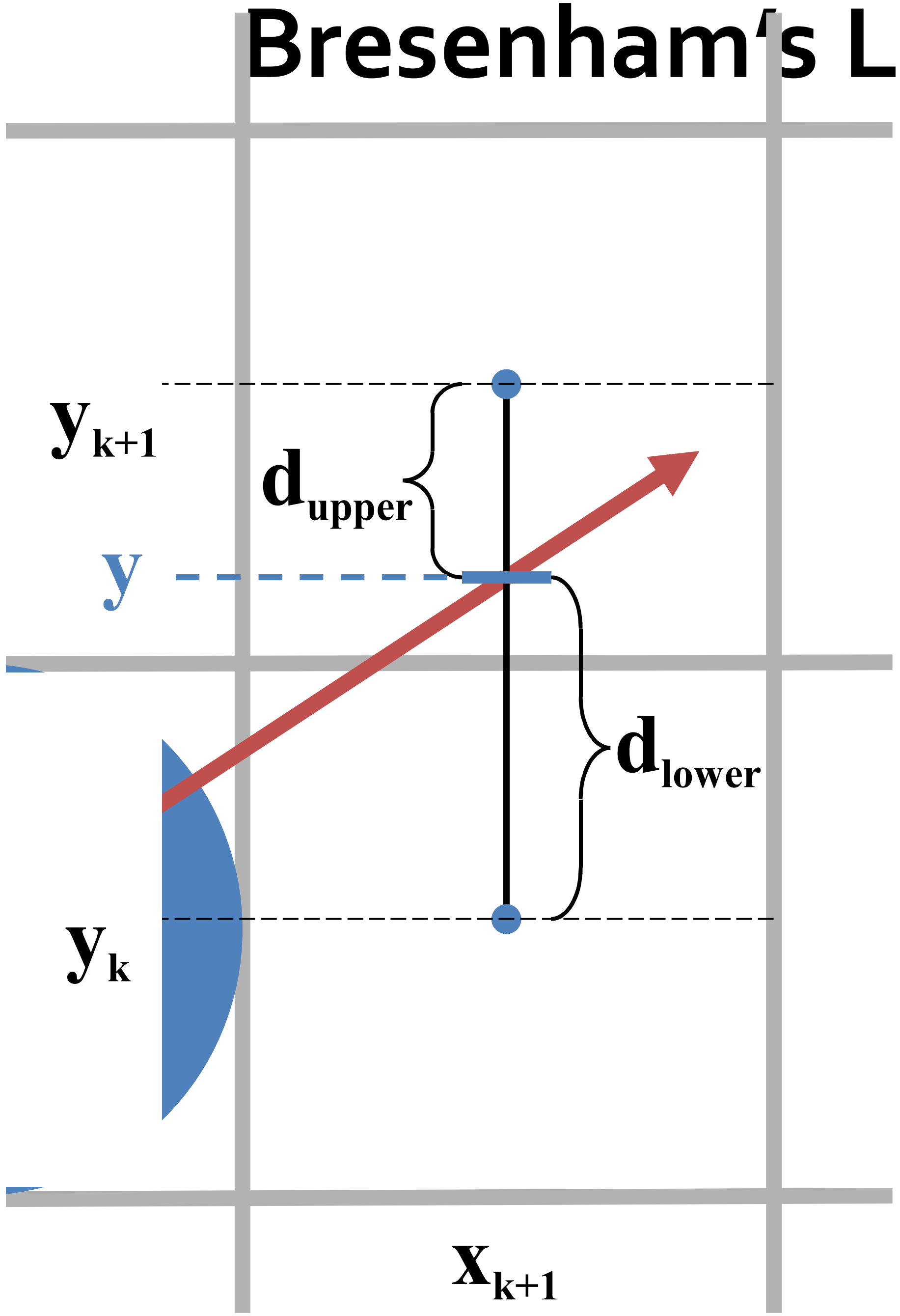
Bresenham's Line Algorithm (1/4)



Bresenham's Line Algorithm (1/4)



Bresenham's Line Algorithm (1/4)



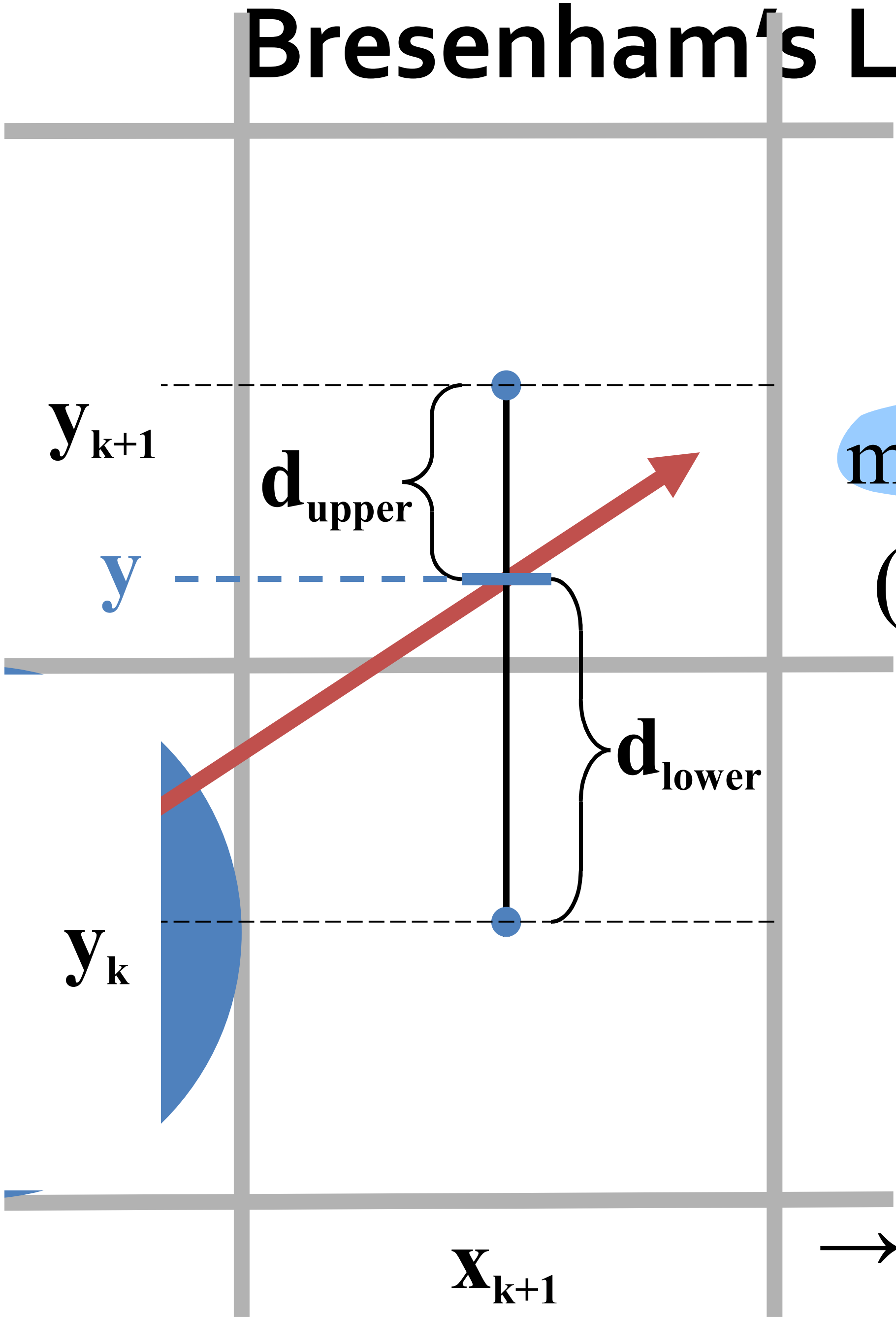
$$y = m\cdot(x_k + 1) + b$$

$$d_{lower} = y - y_k =$$
$$= m\cdot(x_k + 1) + b - y_k$$

$$d_{upper} = (y_k + 1) - y =$$
$$= y_k + 1 - m\cdot(x_k + 1) - b$$

$$d_{lower} - d_{upper} =$$
$$= 2m\cdot(x_k + 1) - 2y_k + 2b - 1$$

Bresenham's Line Algorithm (2/4)



$$d_{lower} - d_{upper} =$$
$$= 2m\cdot(x_k + 1) - 2y_k + 2b - 1$$

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

$$(\Delta x = x_1 - x_0, \Delta y = y_1 - y_0)$$

decision parameter:

$$p_k = \Delta x\cdot(d_{lower} - d_{upper}) =$$
$$= 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$$

$\rightarrow$ same sign as $(d_{lower} - d_{upper})$

Bresenham's Line Algorithm (3/4)

Current decision value:

$$p_{\mathbf{k}} = \Delta x \cdot (d_{\mathbf{lower}} - d_{\mathbf{upper}}) = 2\Delta y \cdot x_{\mathbf{k}} - 2\Delta x \cdot y_{\mathbf{k}} + c$$

Next decision value:

$$p_{\mathbf{k+1}} = p_{\mathbf{k}} + 2\Delta y - 2\Delta x \cdot (y_{\mathbf{k+1}} - y_{\mathbf{k}})$$

Starting decision value:

$$p_{\mathbf{0}} = 2\Delta y - \Delta x$$

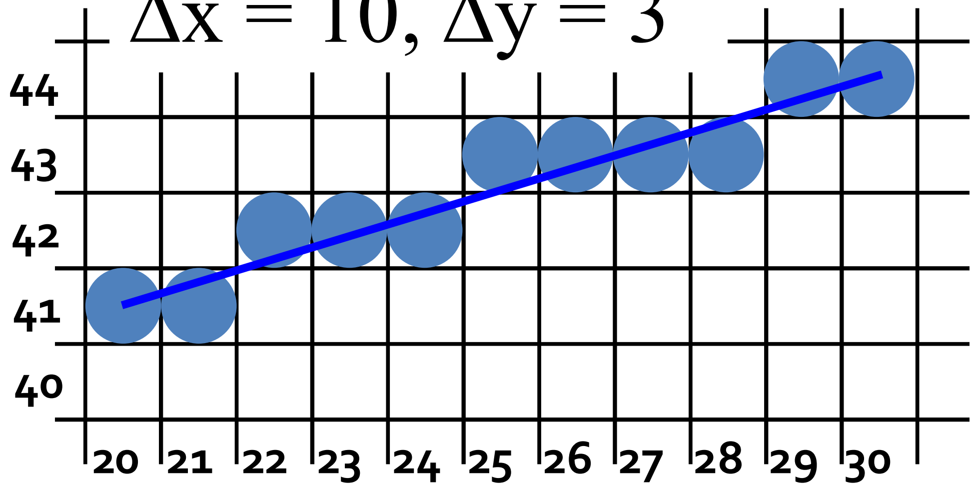
Bresenham's Line Algorithm (4/4)

1. Store left line endpoint in (x_0,y_0)
2. Draw pixel (x_0,y_0)
3. Calculate constants Δx, Δy, $2\Delta y$, $2\Delta y - 2\Delta x$, and obtain $p_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, perform test:
   if $p_k <= 0$
      then draw (x_k+1, y_k); $p_{k+1} = p_k + 2\Delta y$
      else draw (x_k+1, y_k+1); $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
5. Perform step 4 $(\Delta x - 1)$ times

Bresenham: Example

k	p_k	(x_{k+1}, y_{k+1})
		(20, 41)
0	−4	(21, 41)
1	2	(22, 42)
2	−12	(23, 42)
3	−6	(24, 42)
4	0	(25, 43)
5	−14	(26, 43)
6	−8	(27, 43)
7	−2	(28, 43)
8	4	(29, 44)
9	−10	(30, 44)

$\Delta x = 10,\ \Delta y = 3$



$p_0 = 2\Delta y - \Delta x$

if $p_k <= 0$

then draw pixel (x_k+1, y_k);

$$p_{k+1} = p_k + 2\Delta y$$

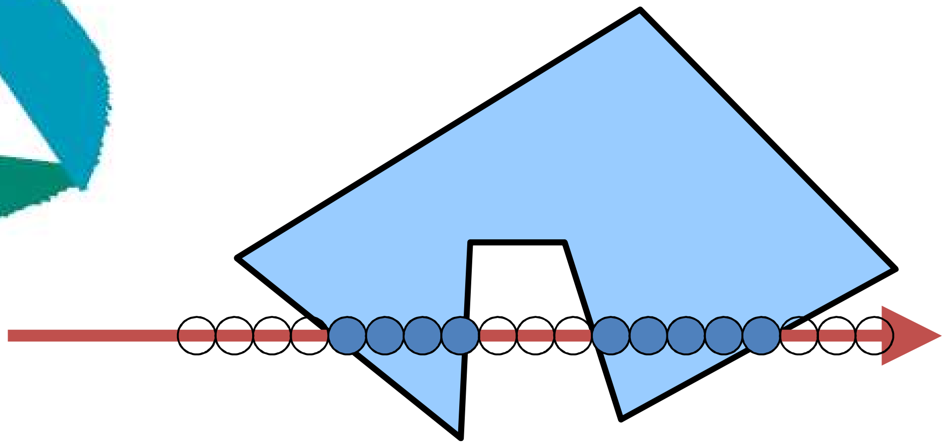
else draw pixel (x_k+1, y_k+1);

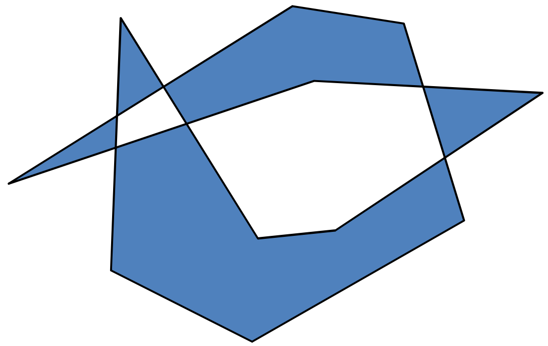
$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

Résumé

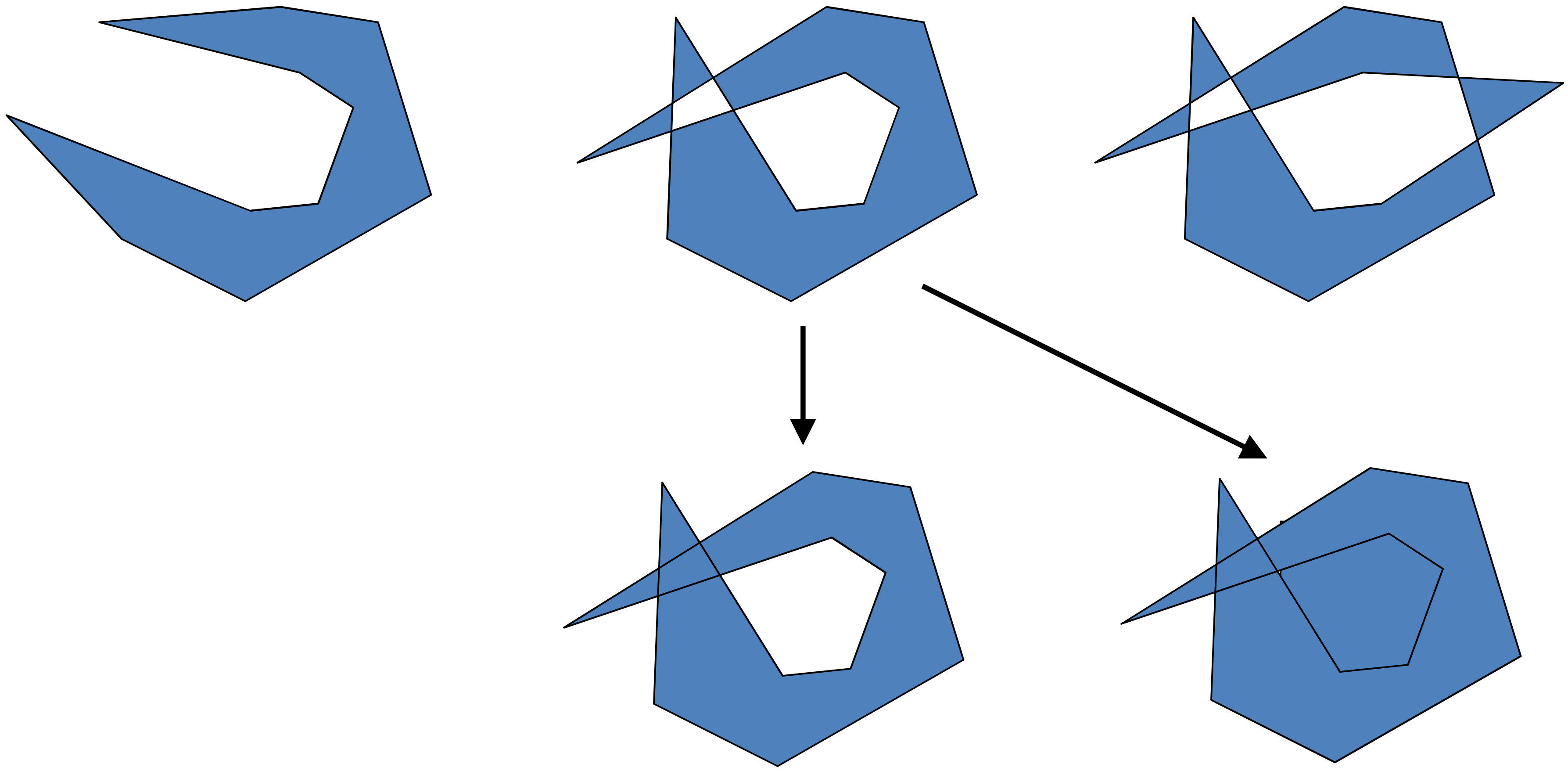
- Quality
  - Works
- Performance
  - No division()
  - No round()
  - No floating point operation
- Good idea
  - Adaptable to circles, other curves
  - Look at what cases are relevant in praxis

Polygon Filling



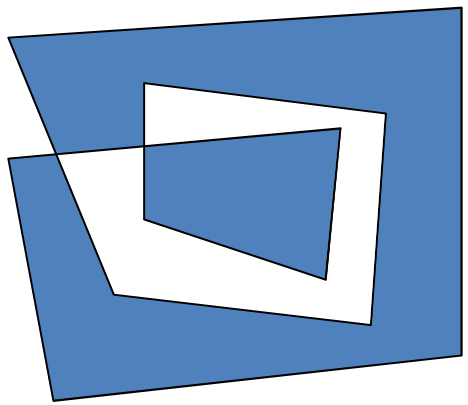


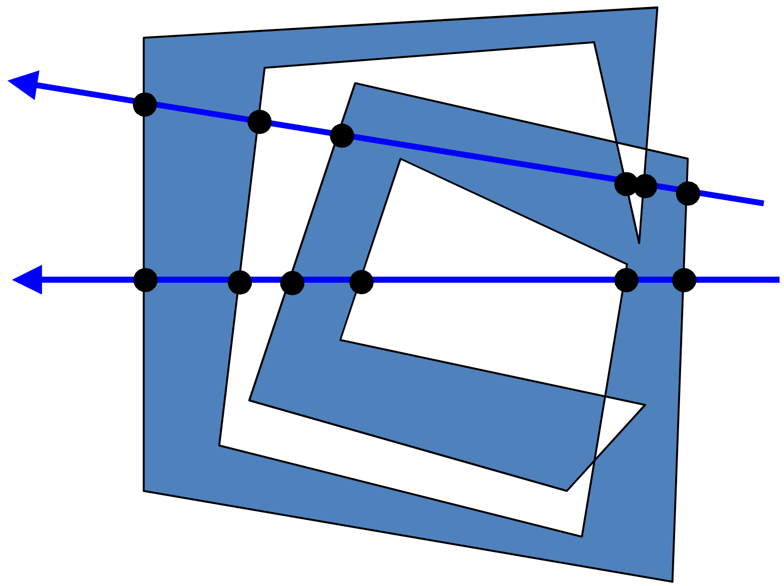
What is Inside a Polygon?



Odd-Even Rule

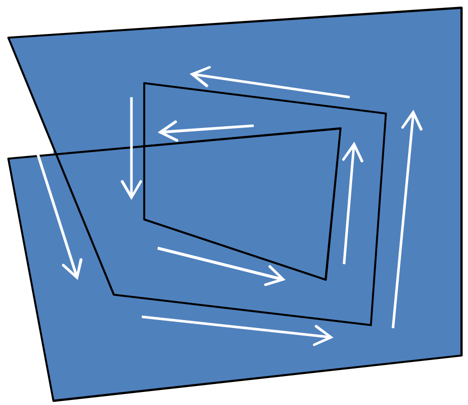
- Inside/outside switches at every edge
- Straight line to the outside:
  - Even # edge intersections = outside
  - Odd # edge intersections = inside

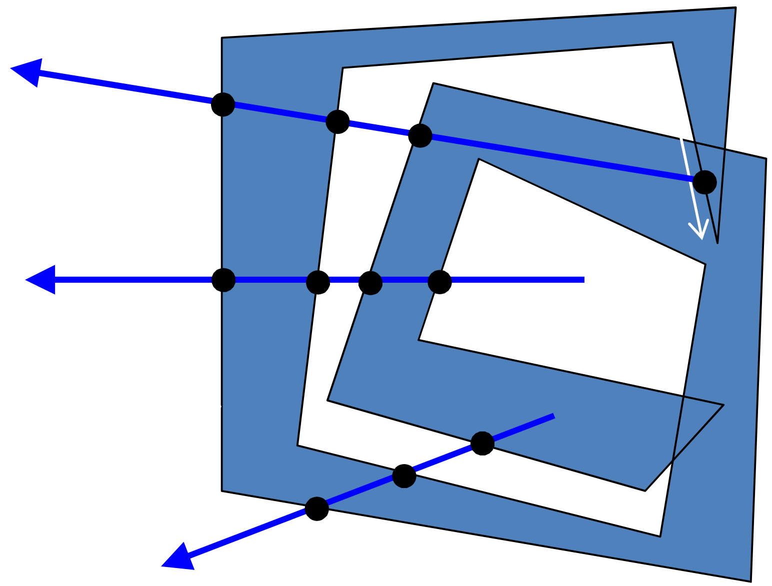




Nonzero Winding Number

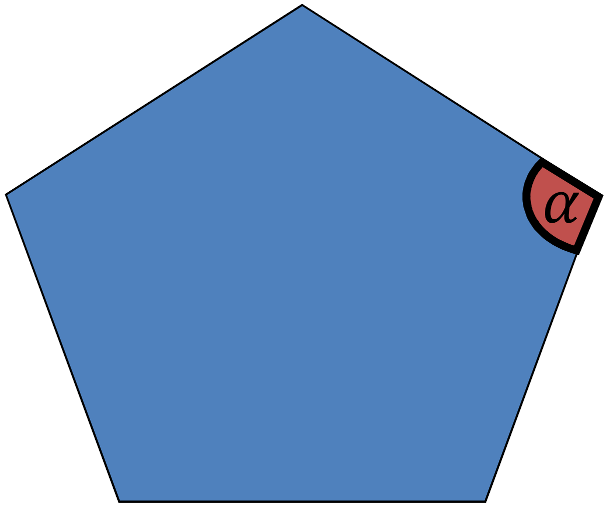
- Point is inside if polygon surrounds it
- Straight line to the outside:
  - same # edges up and down = outside
  - different # edges up and down = inside

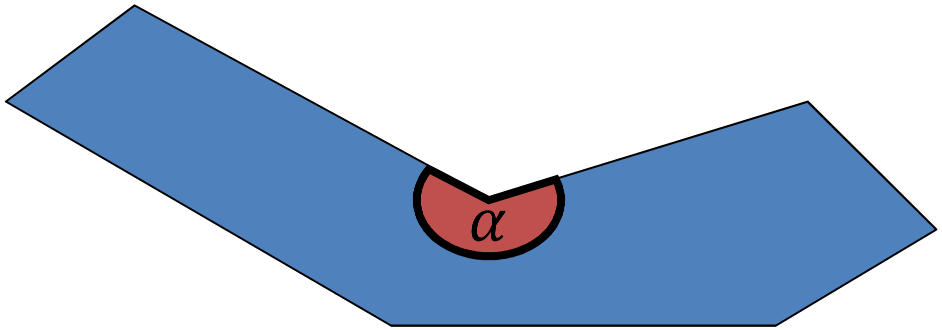




Polygon Classification

- **Convex**: no interior angle > 180°
- **Concave**: not convex

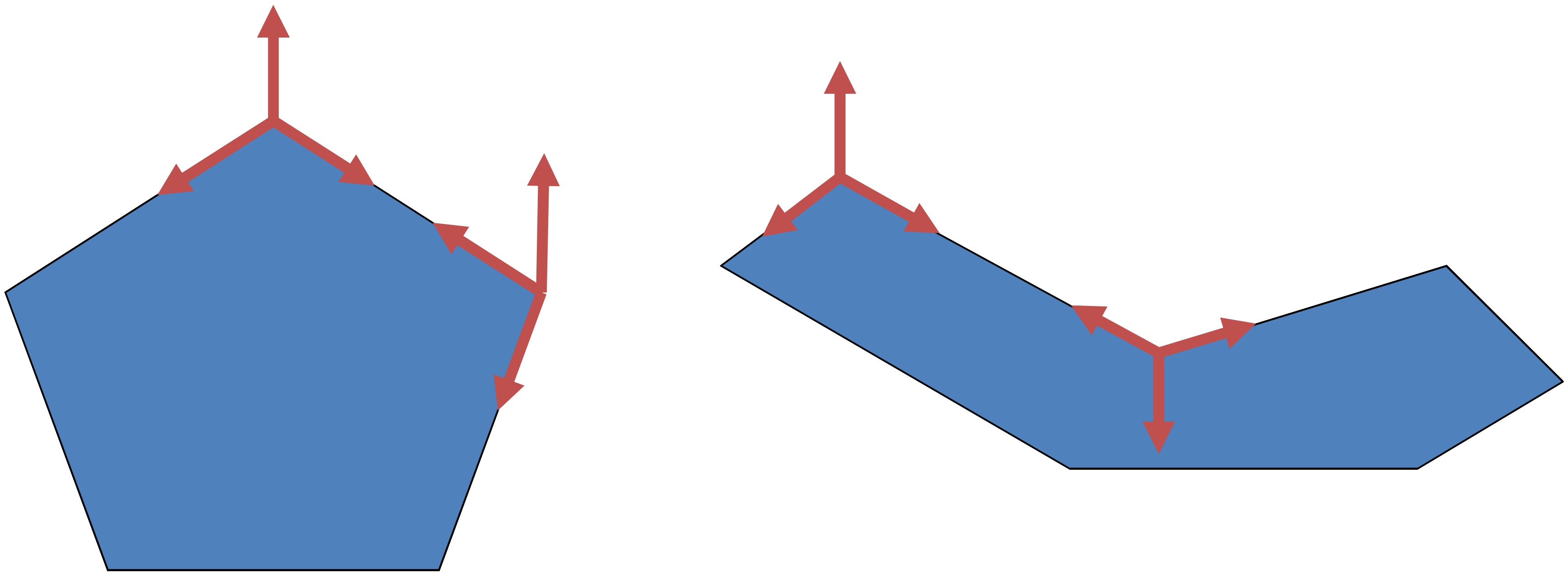




Polygon Classification – Vector Test

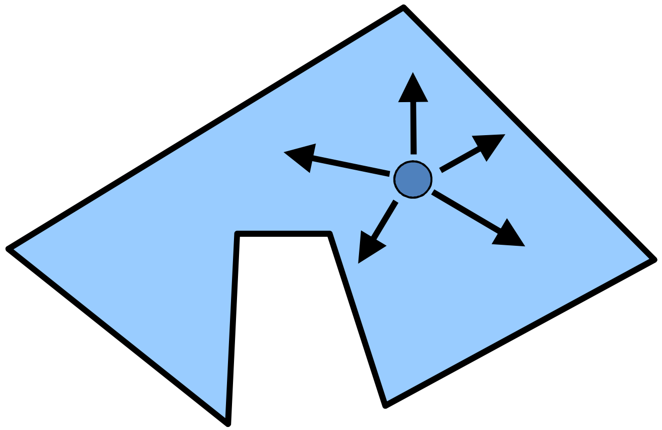
- All vector cross products have the same sign

  $\Rightarrow$ convex

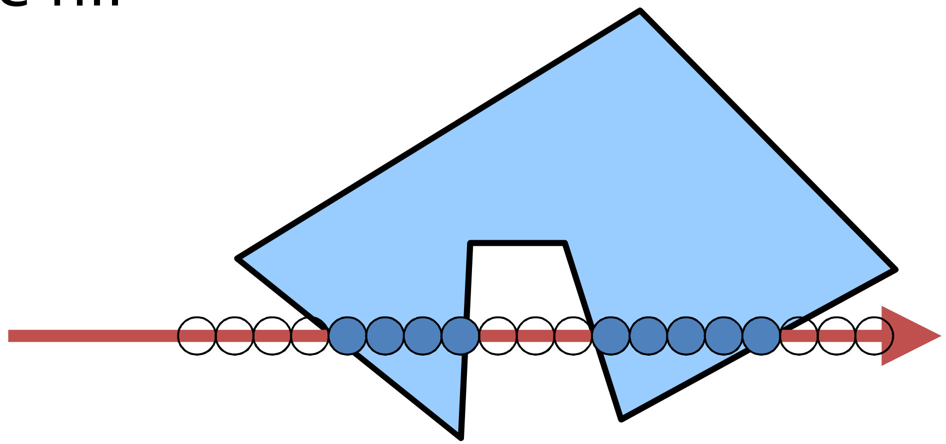


Fill Algorithms

- Flood fill



- Scan-line fill

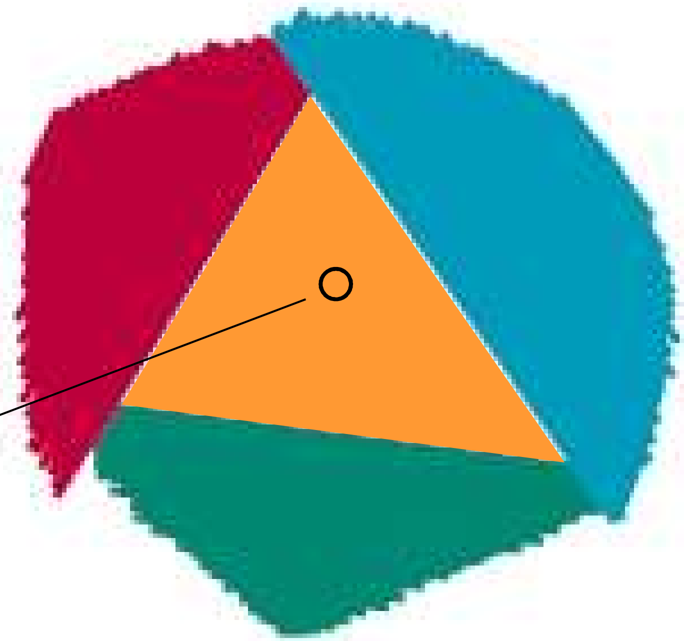


Flood-Fill Algorithm

- Pixel filling of area
  - Start from interior point
  - “Flood” internal region

Flood-Fill: Boundary and Seed Point

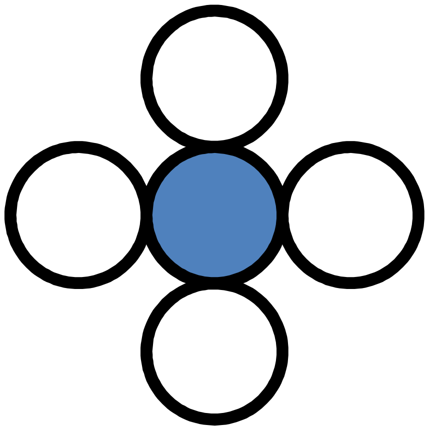
- Area must be distinguishable from boundaries
- Example
  - Area defined within multiple color boundaries

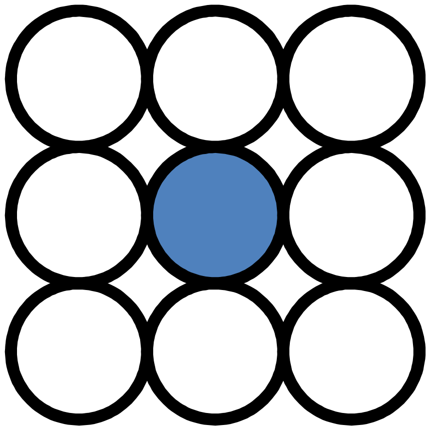


seed point

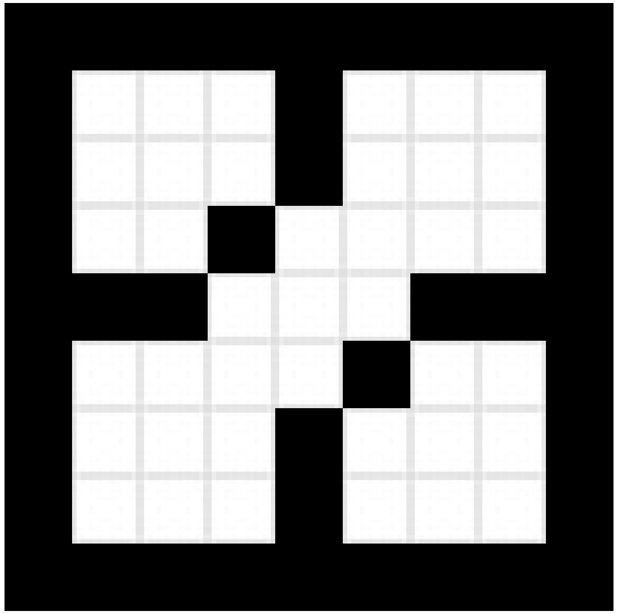
Flood-Fill: Who is my Neighbour?

- *4-connected* means, that a connection is only valid in these 4 directions
- *8-connected* means, that a connection is valid in these 8 directions

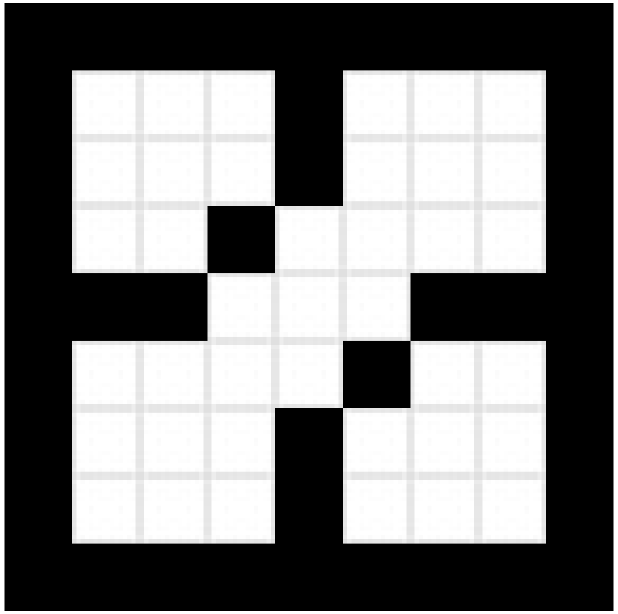




Flood-Fill: Connectedness

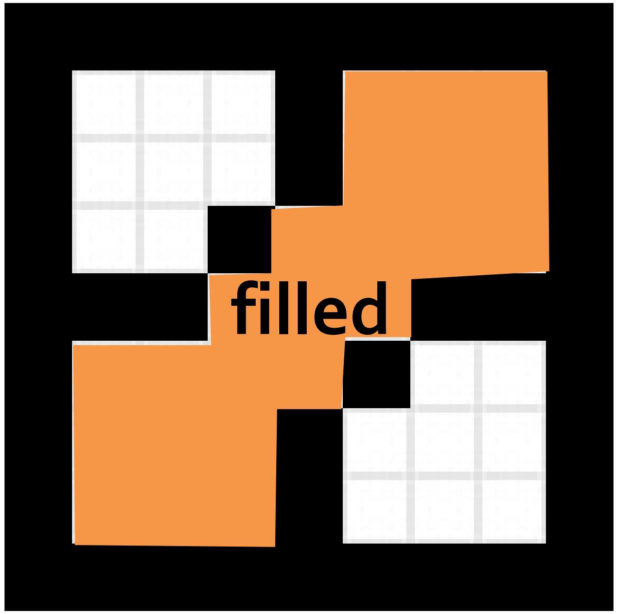


4-connected

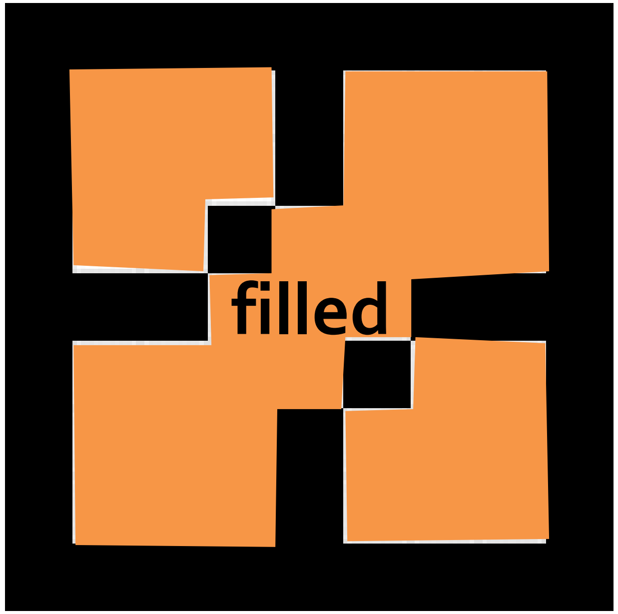


8-connected

Flood-Fill: Connectedness



4-connected

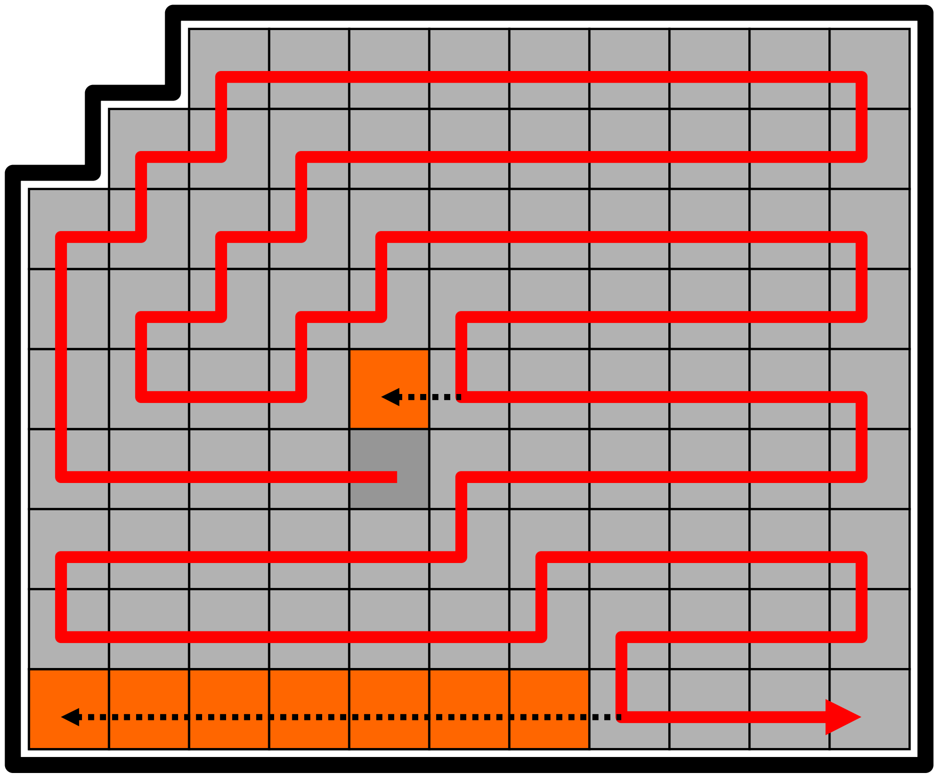


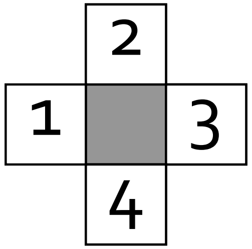
8-connected

Simple Flood-Fill Algorithm

```
void floodFill4(x, y, new, old)
{
  int color = getPixel (x, y);
  if (color == old) {
    drawPixel (x, y, new);
    floodFill4 (x-1, y, new, old);   // left
    floodFill4 (x, y+1, new, old);   // up
    floodFill4 (x+1, y, new, old);   // right
    floodFill4 (x, y-1, new, old);   // down
  }
}
```

Bad Behavior of Simple Flood-Fill

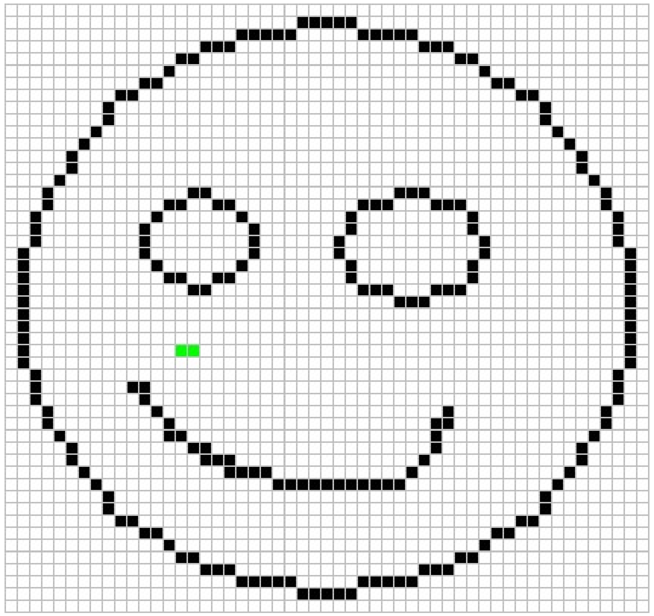




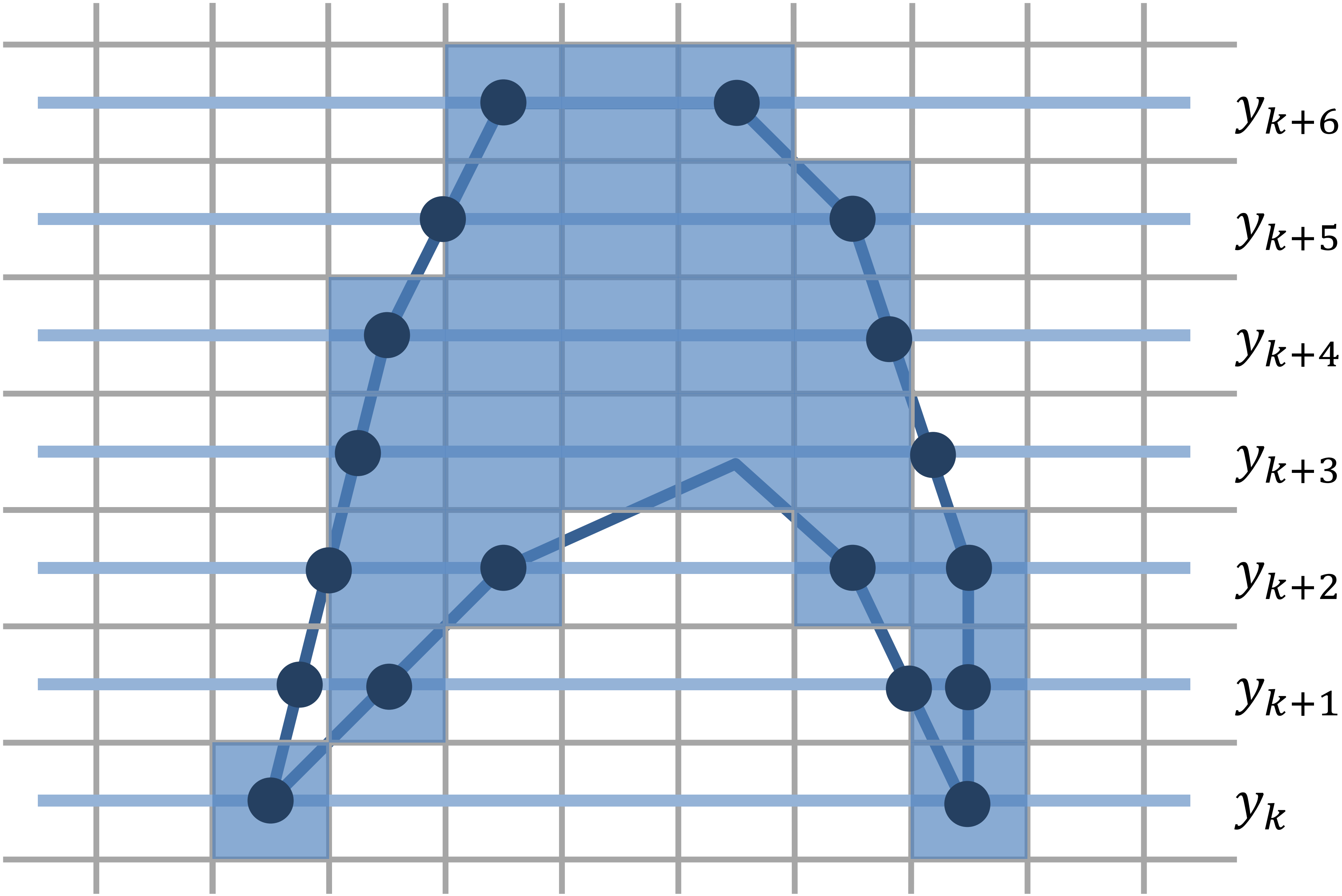
recursion sequence

Span Flood-Fill Algorithm

- FloodFill4 produces too high stacks (recursion!)
- Solution
  - Incremental horizontal fill (left to right)
  - Recursive vertical fill (first up then down)



Scan-Line Fill



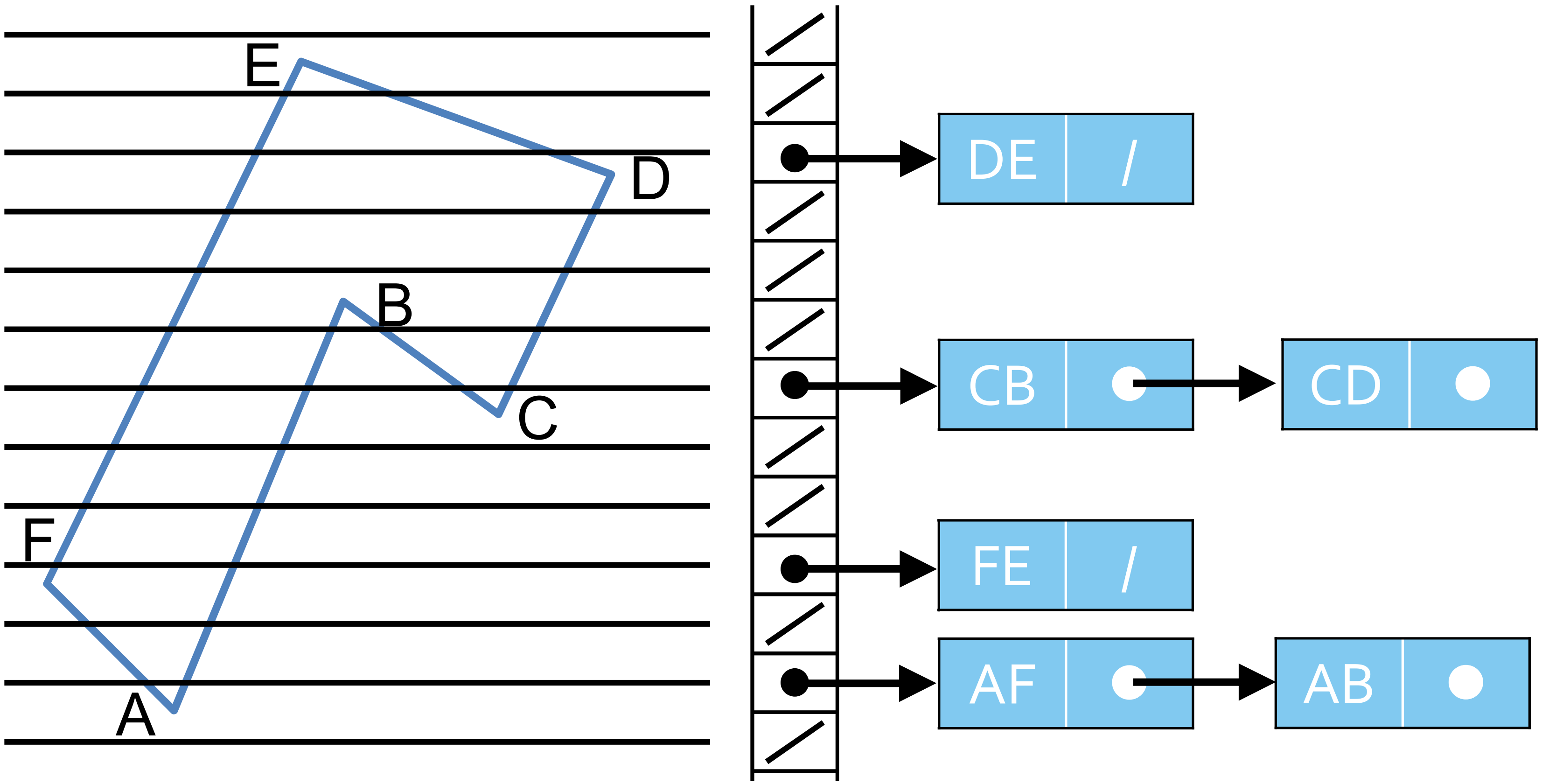
Per Scan-Line

- Find active edges
- Find Intersection points
- Color in-between

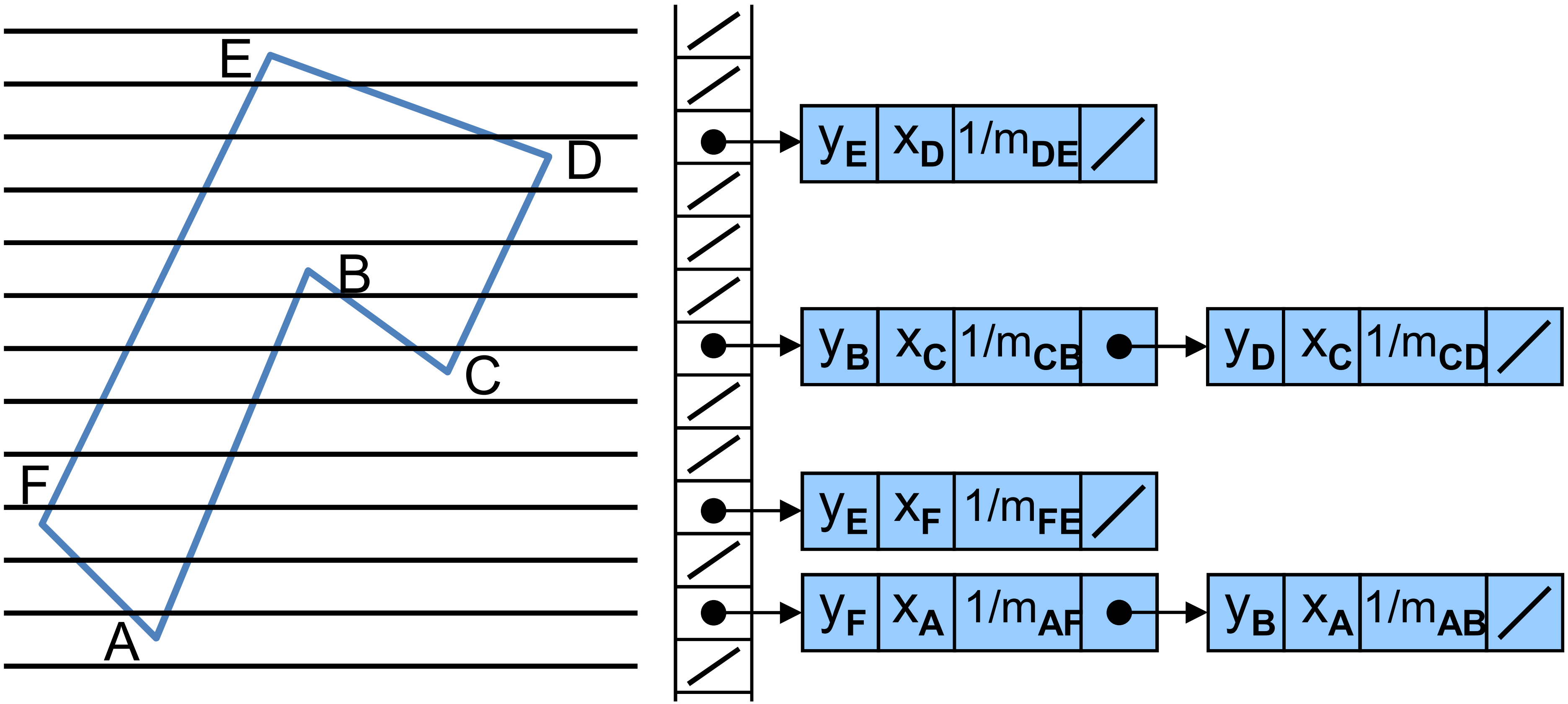
Scan-Line Fill: Incremental Update

- $y_{k+1} = y_k + 1$
- $x_{k+1} = x_k + \frac{1}{m}$
- Efficient data structure: Sorted Edge Table

Scan-Line Fill: Sorted Edge Table

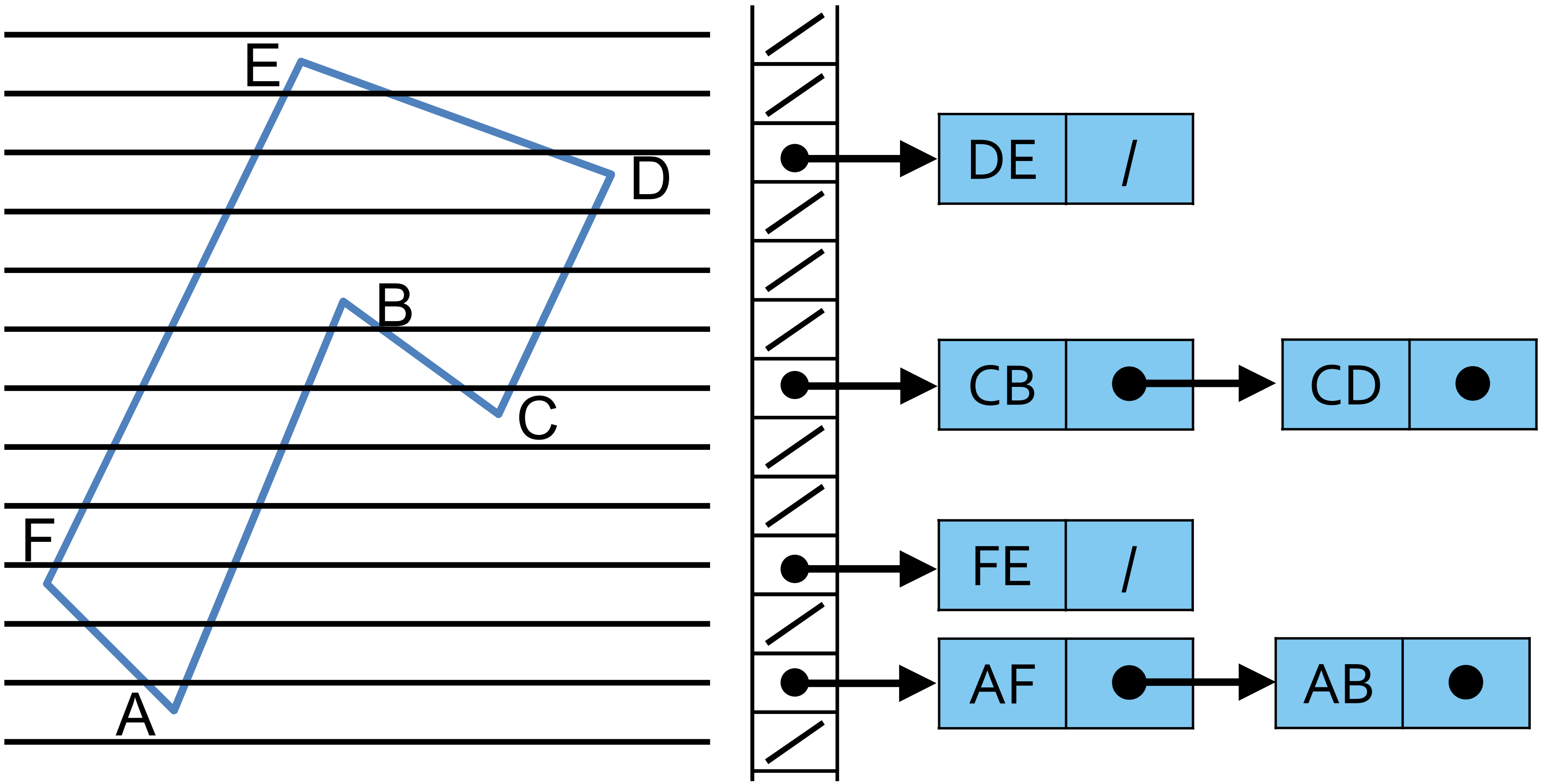


Scan-Line Fill: Sorted Edge Table

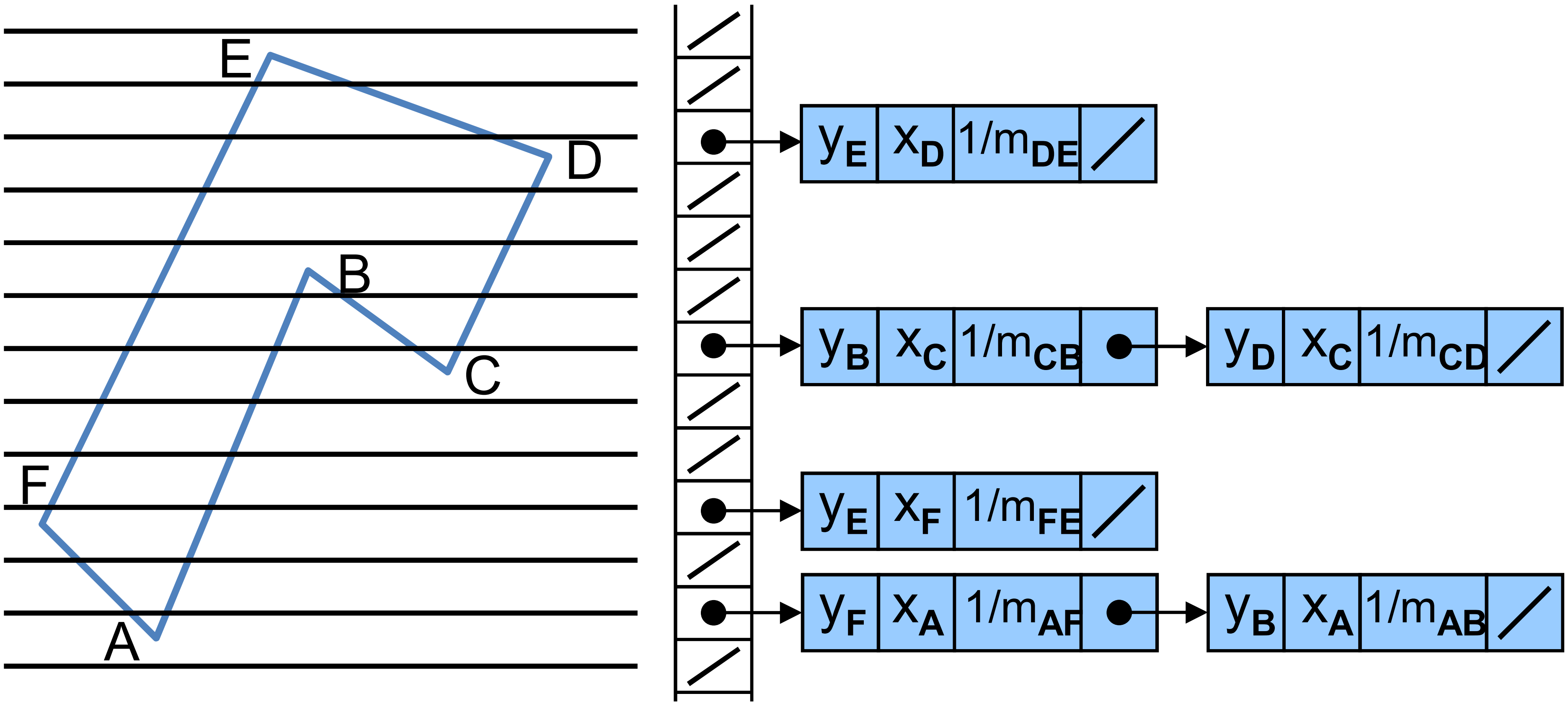


- Edge entry: [max y-value, x-start, inverse slope]

Scan-Line Fill: Sorted Edge Table

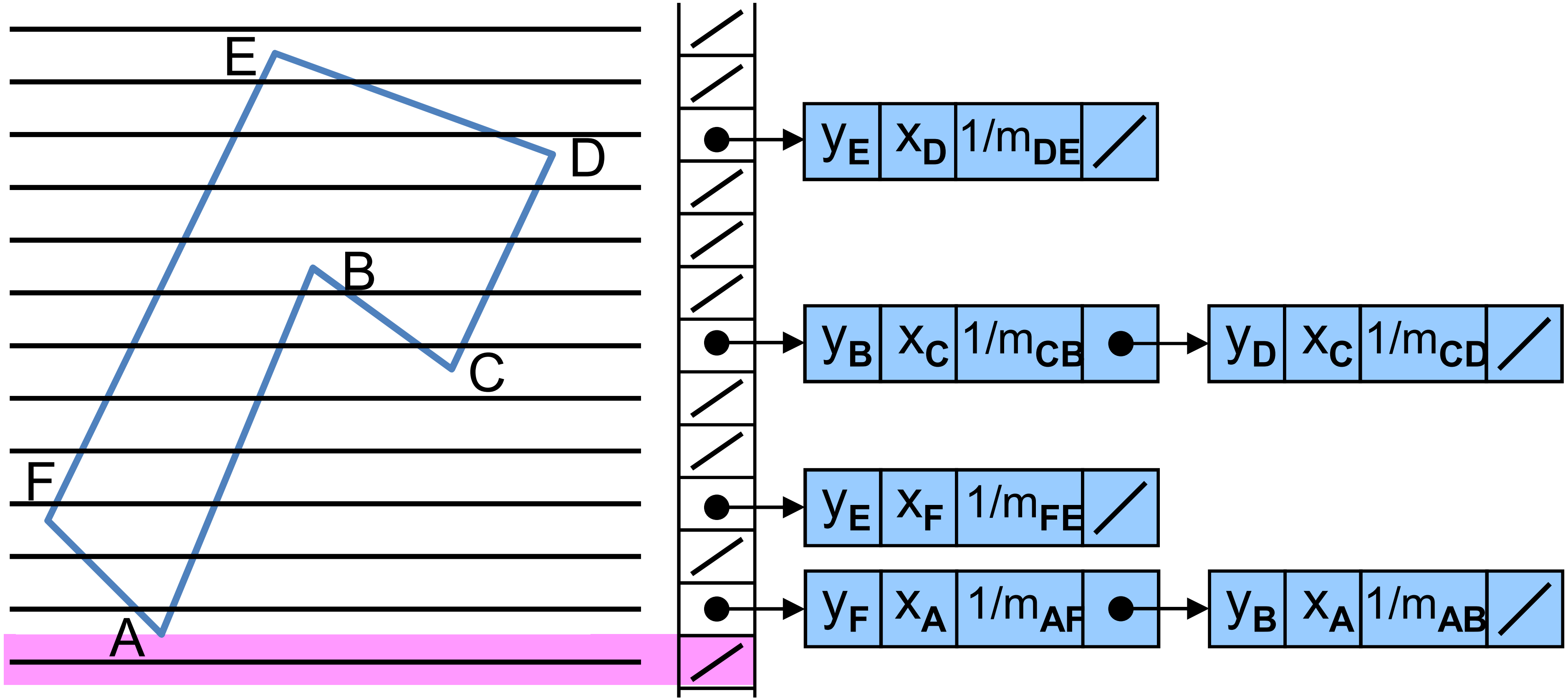


Scan-Line Fill: Sorted Edge Table



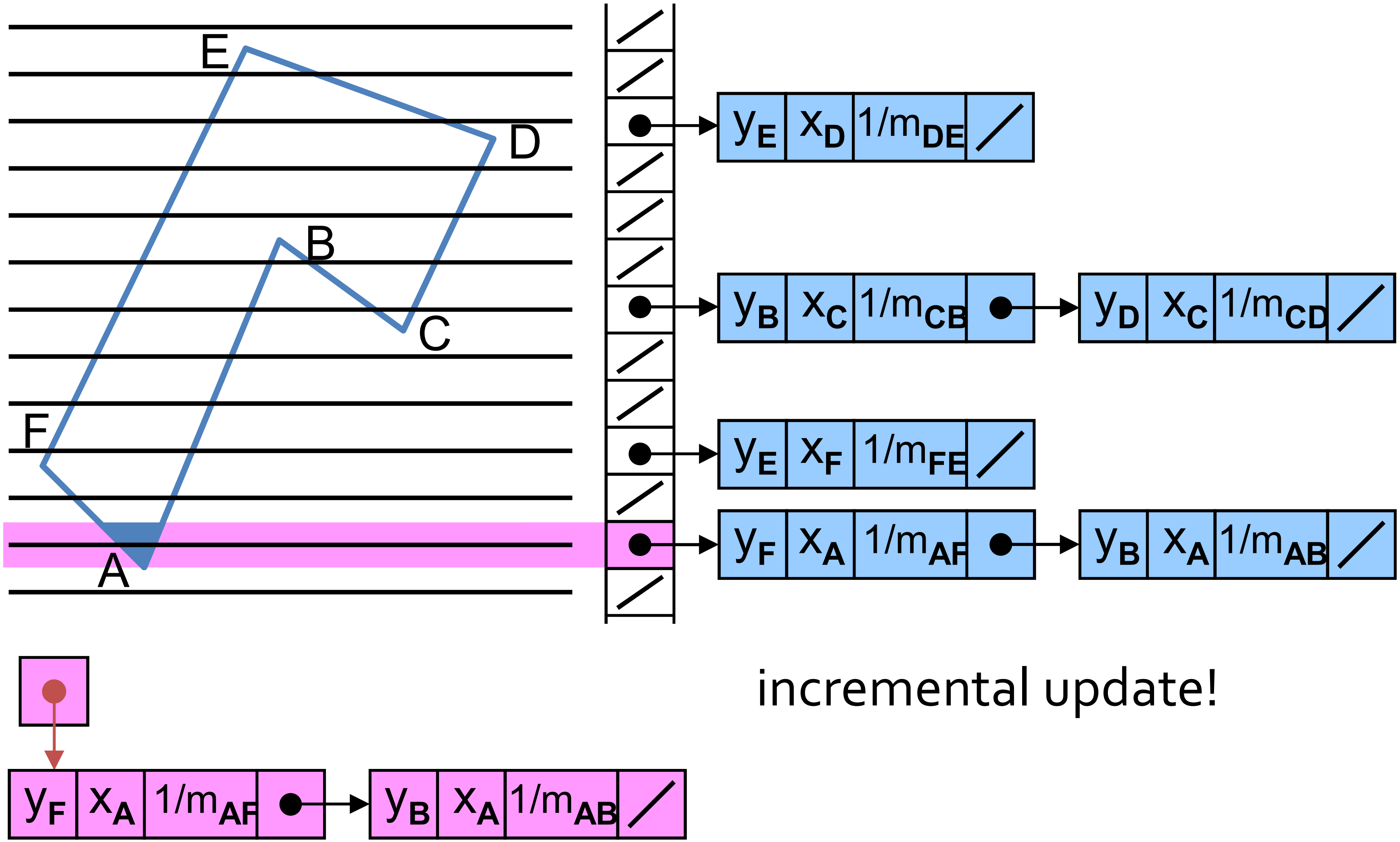
- Edge entry: [max y-value, x-start, inverse slope]

Sorted Edge Table / Active Edge List



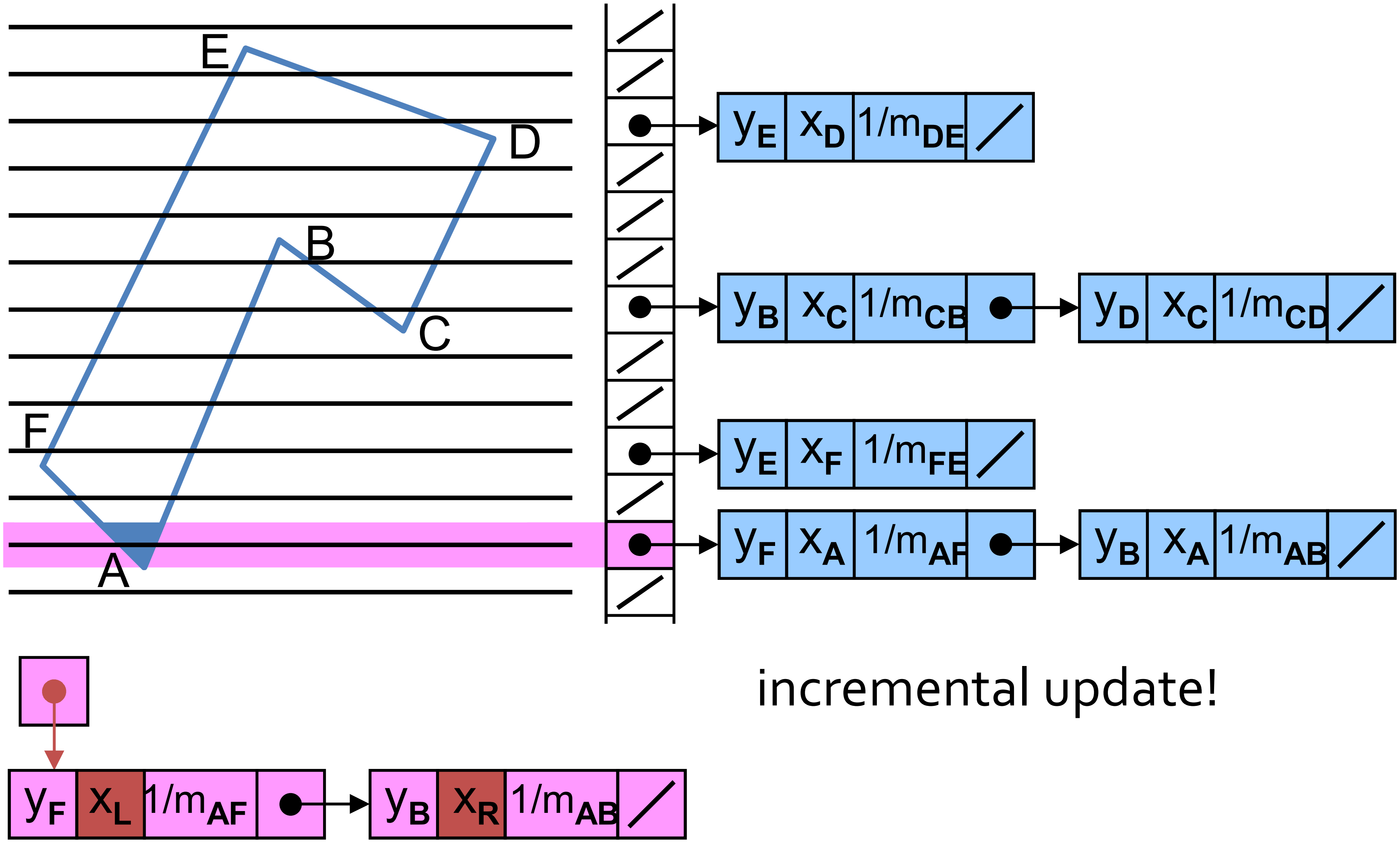
Active Edge List

Sorted Edge Table / Active Edge List



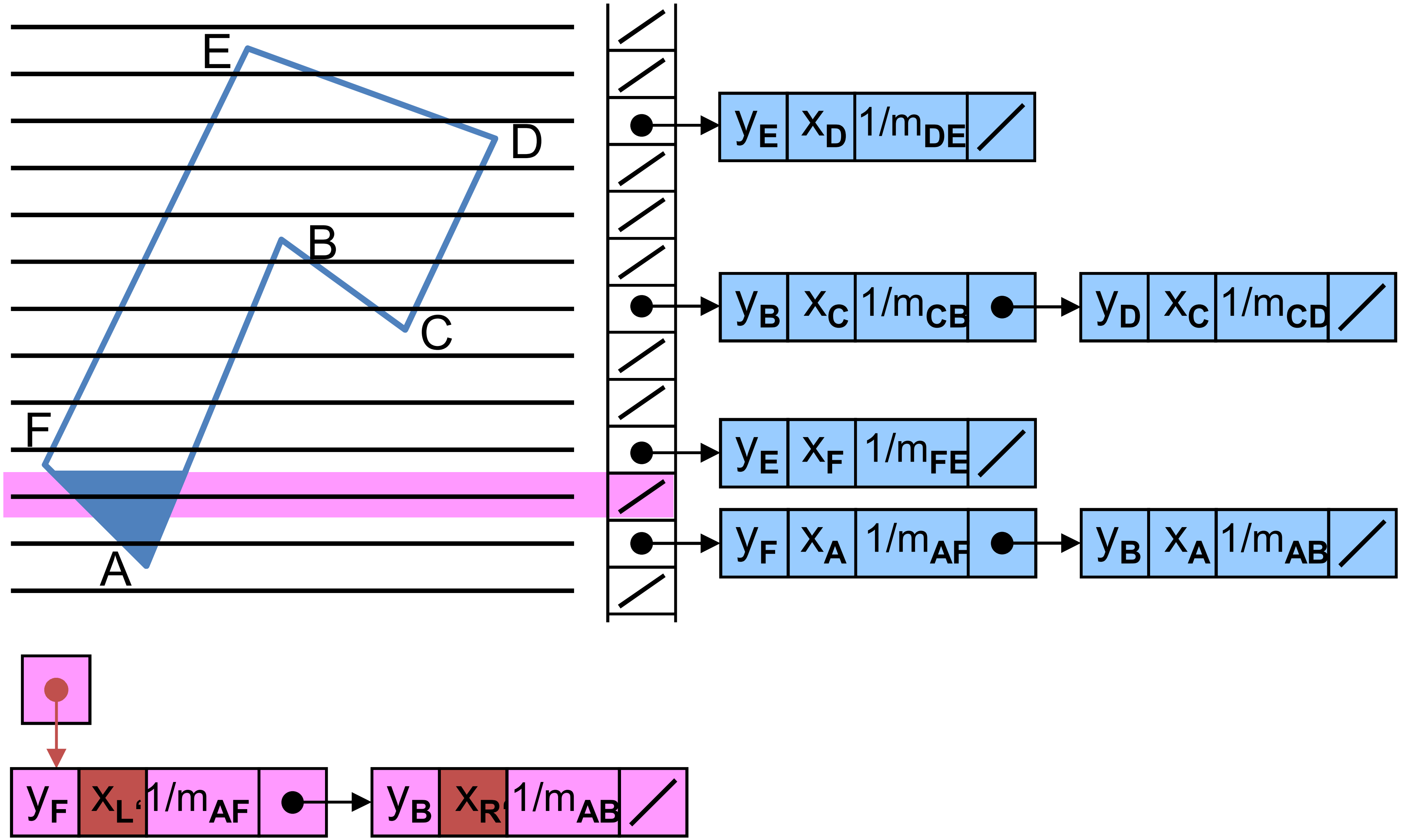
incremental update!

Sorted Edge Table / Active Edge List

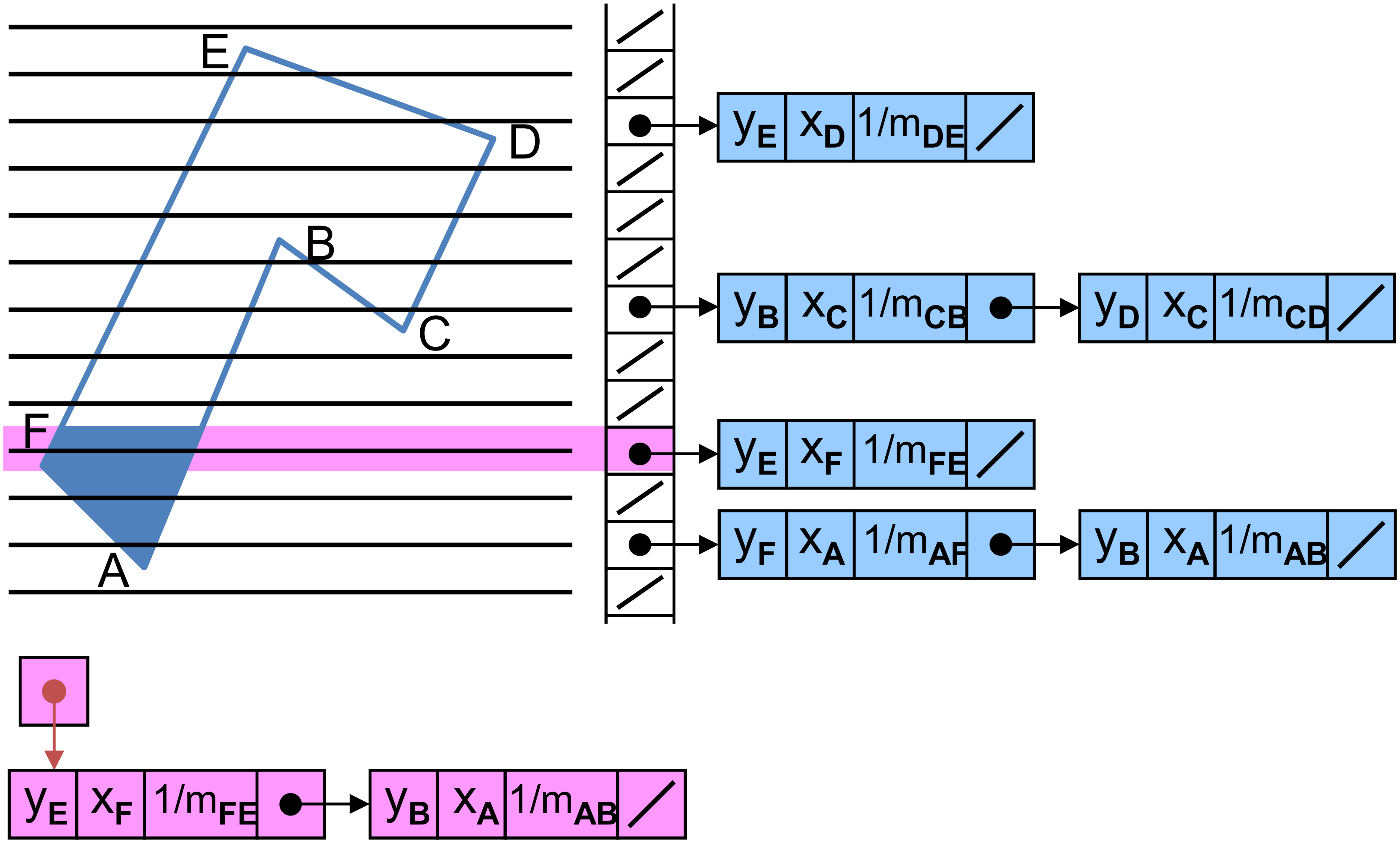


incremental update!

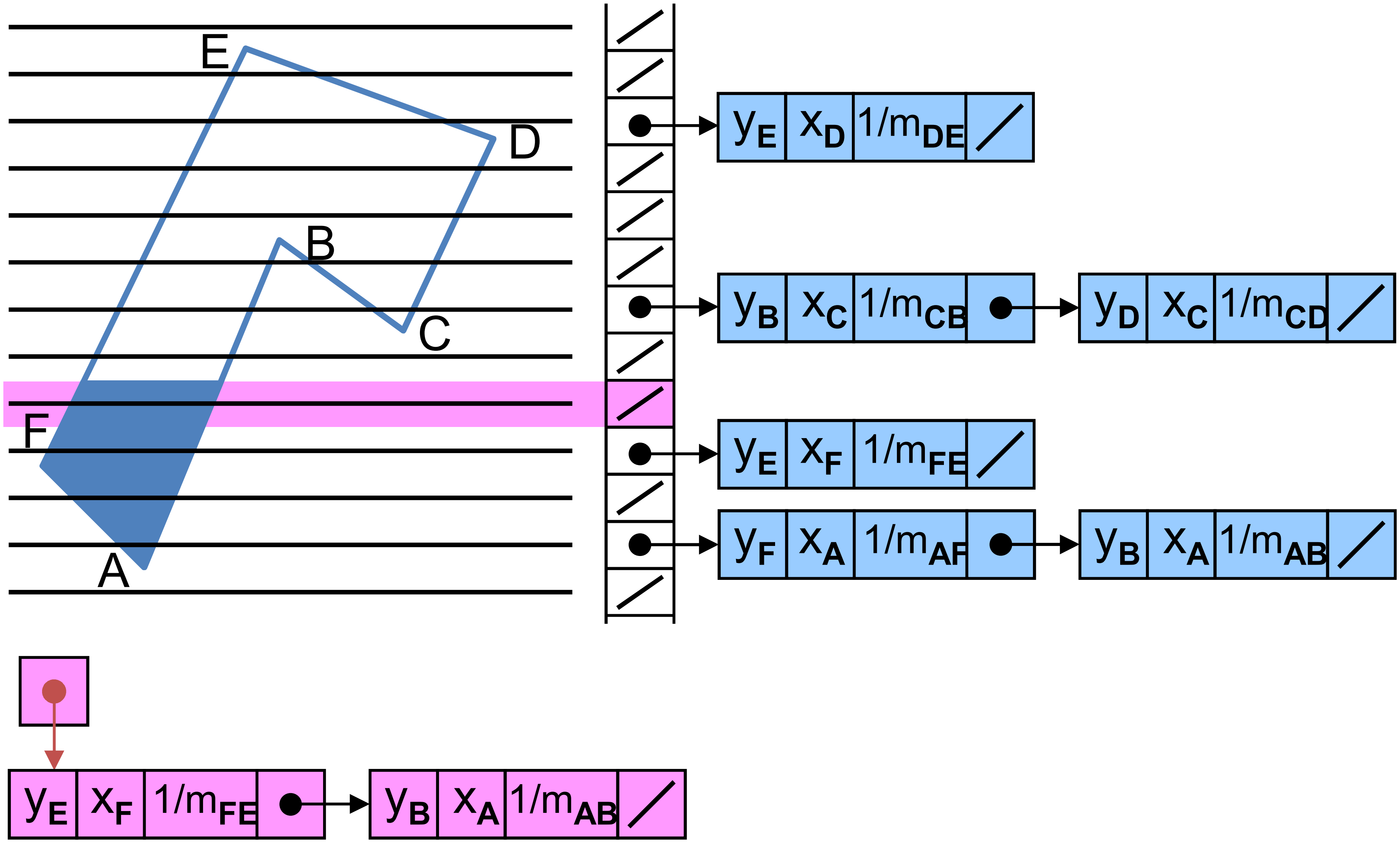
Sorted Edge Table / Active Edge List



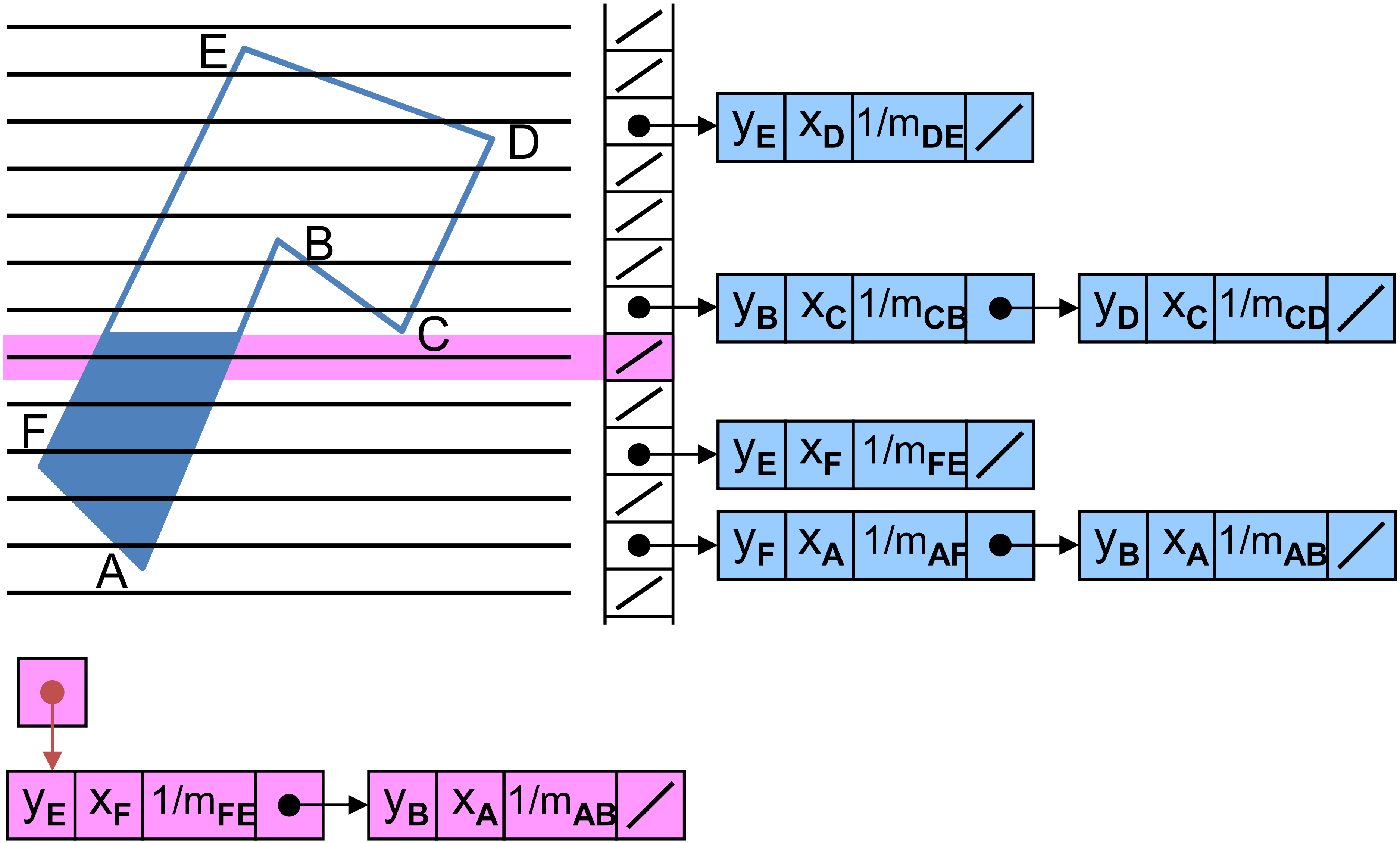
Sorted Edge Table / Active Edge List



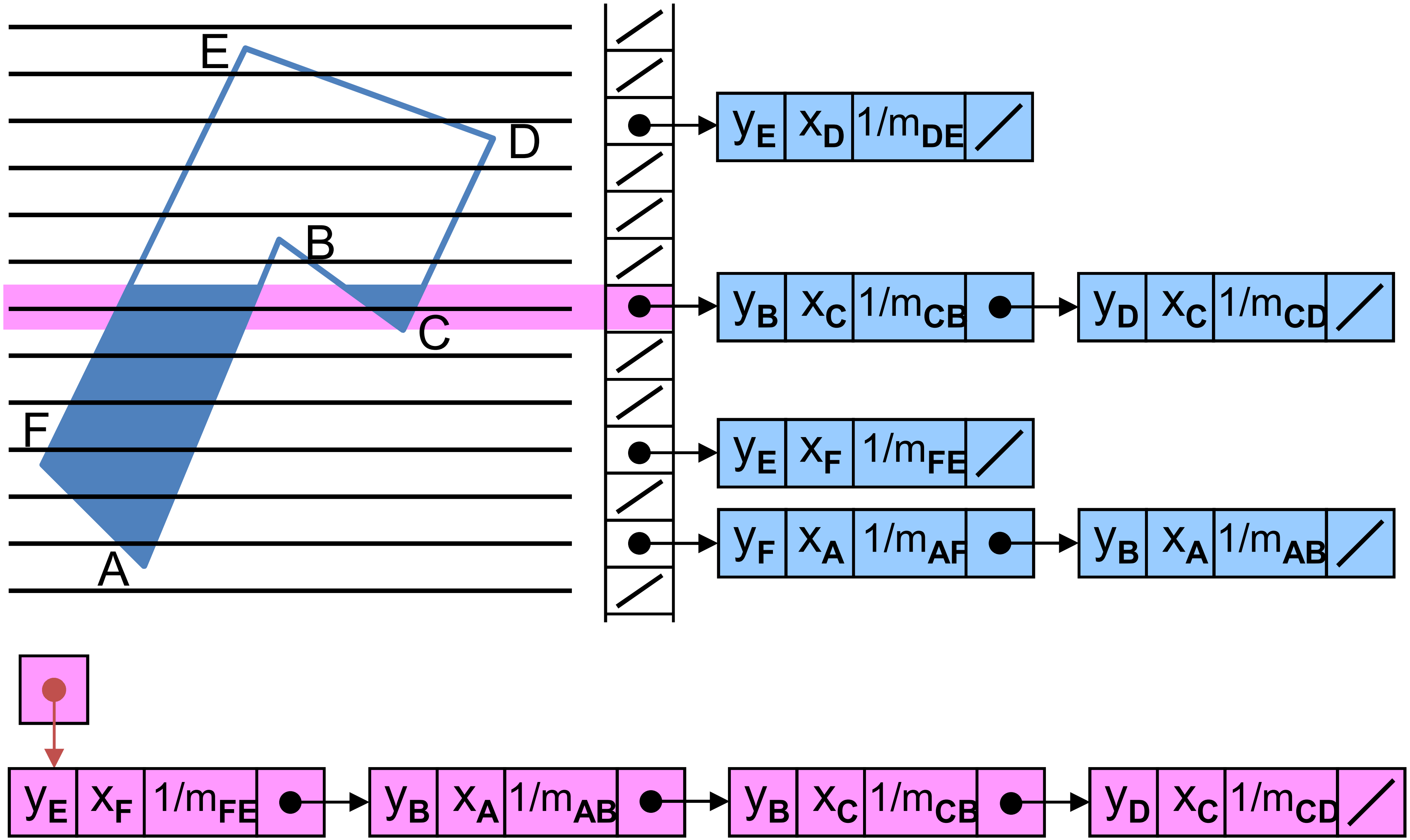
Sorted Edge Table / Active Edge List



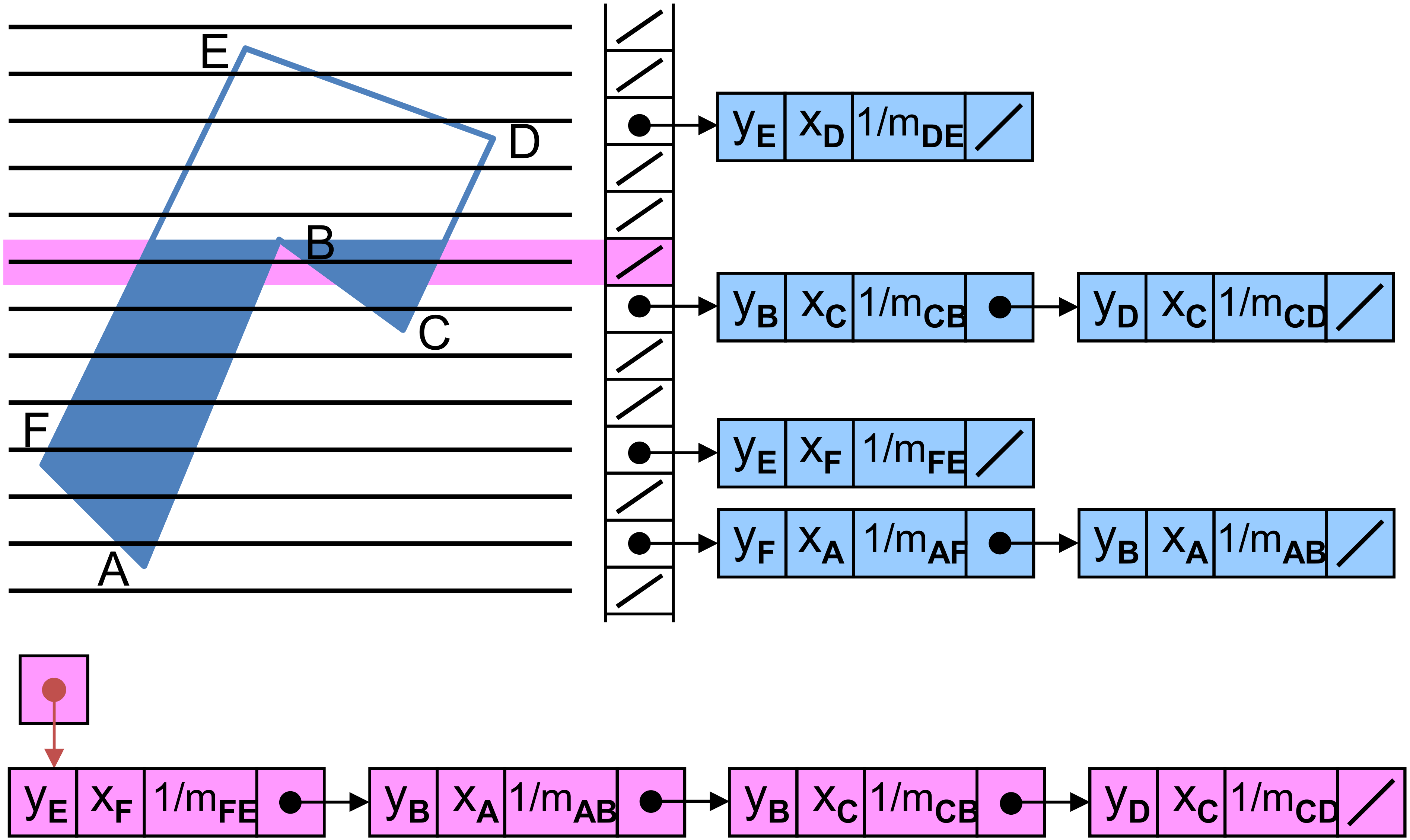
Sorted Edge Table / Active Edge List



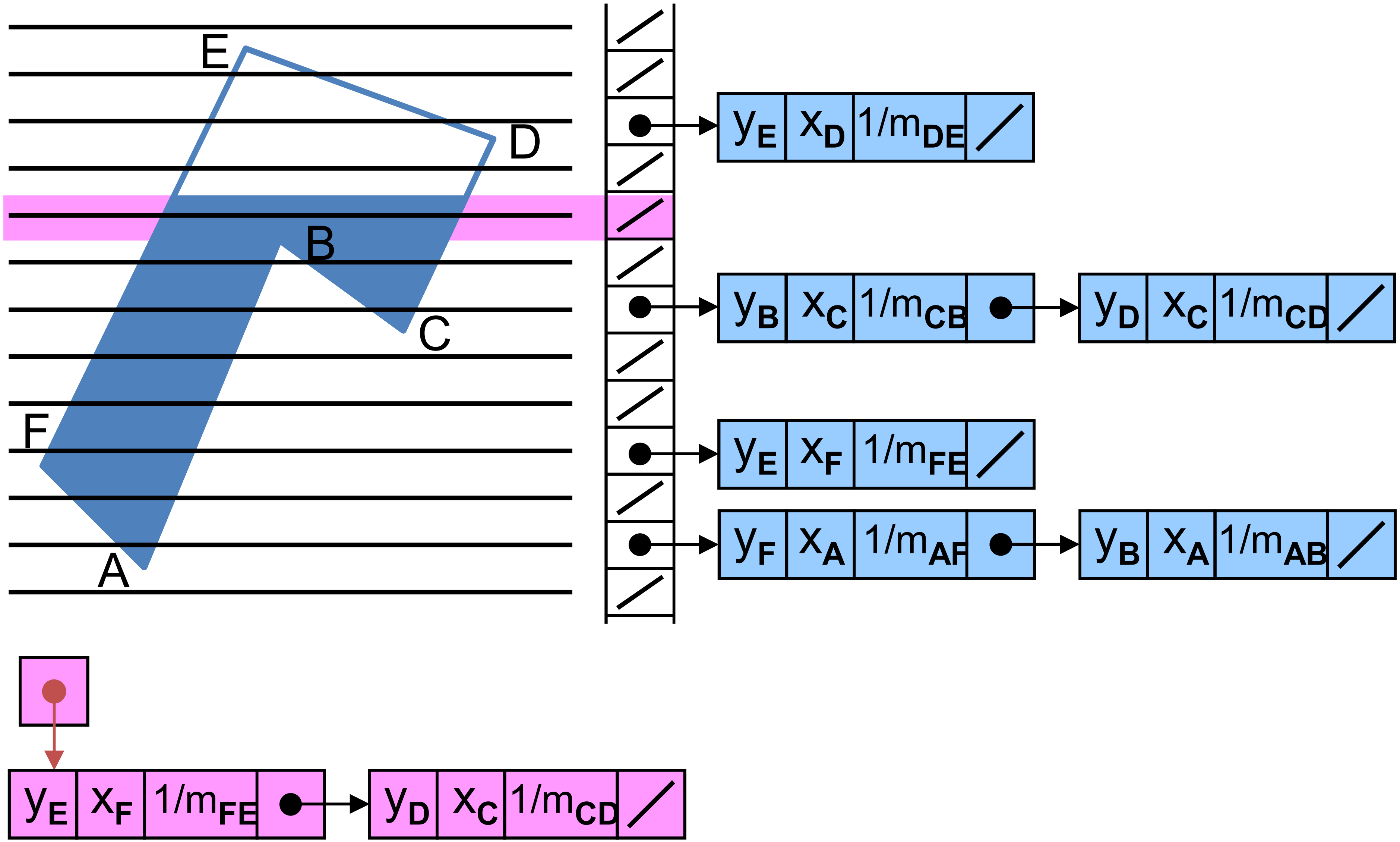
Sorted Edge Table / Active Edge List



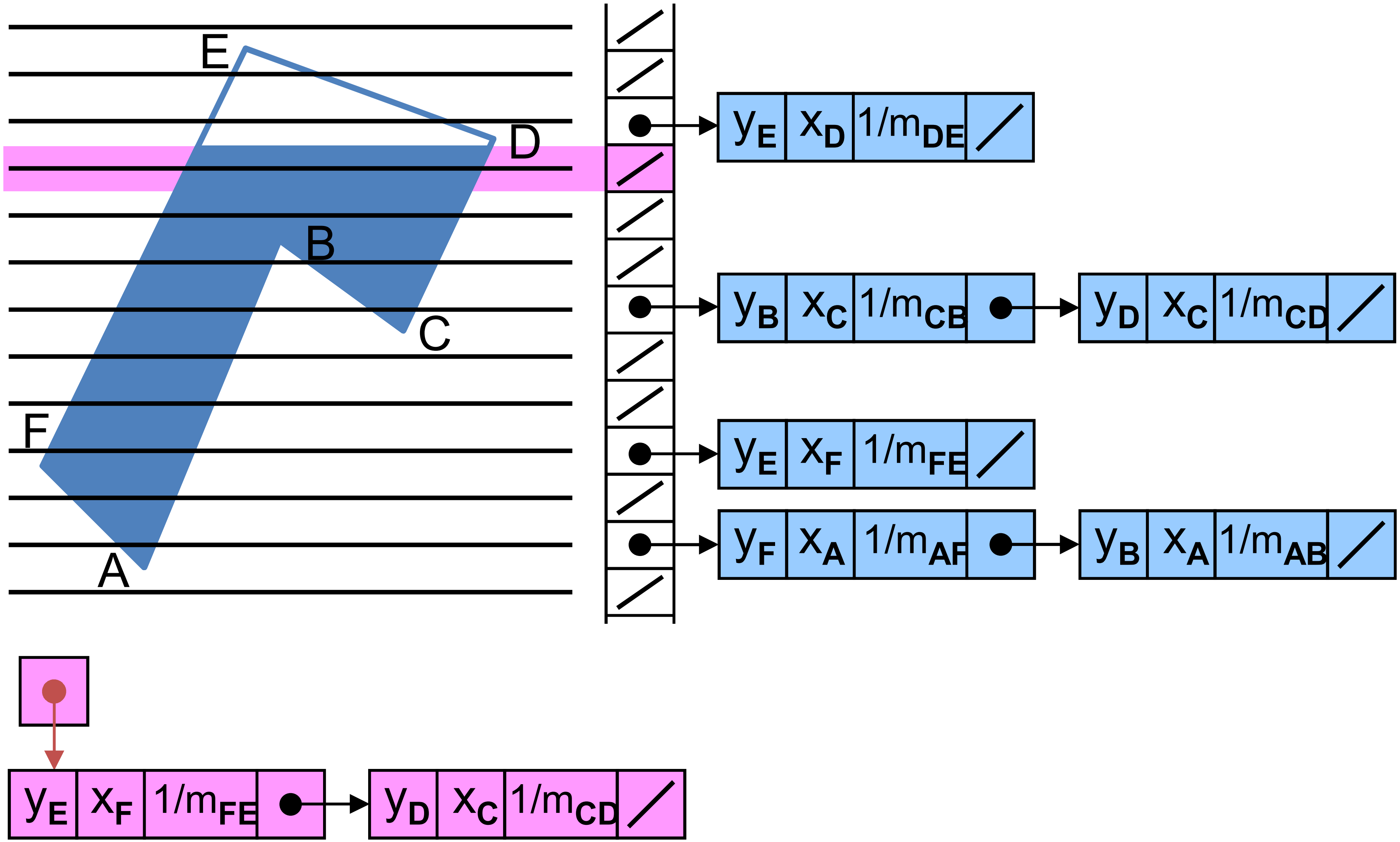
Sorted Edge Table / Active Edge List



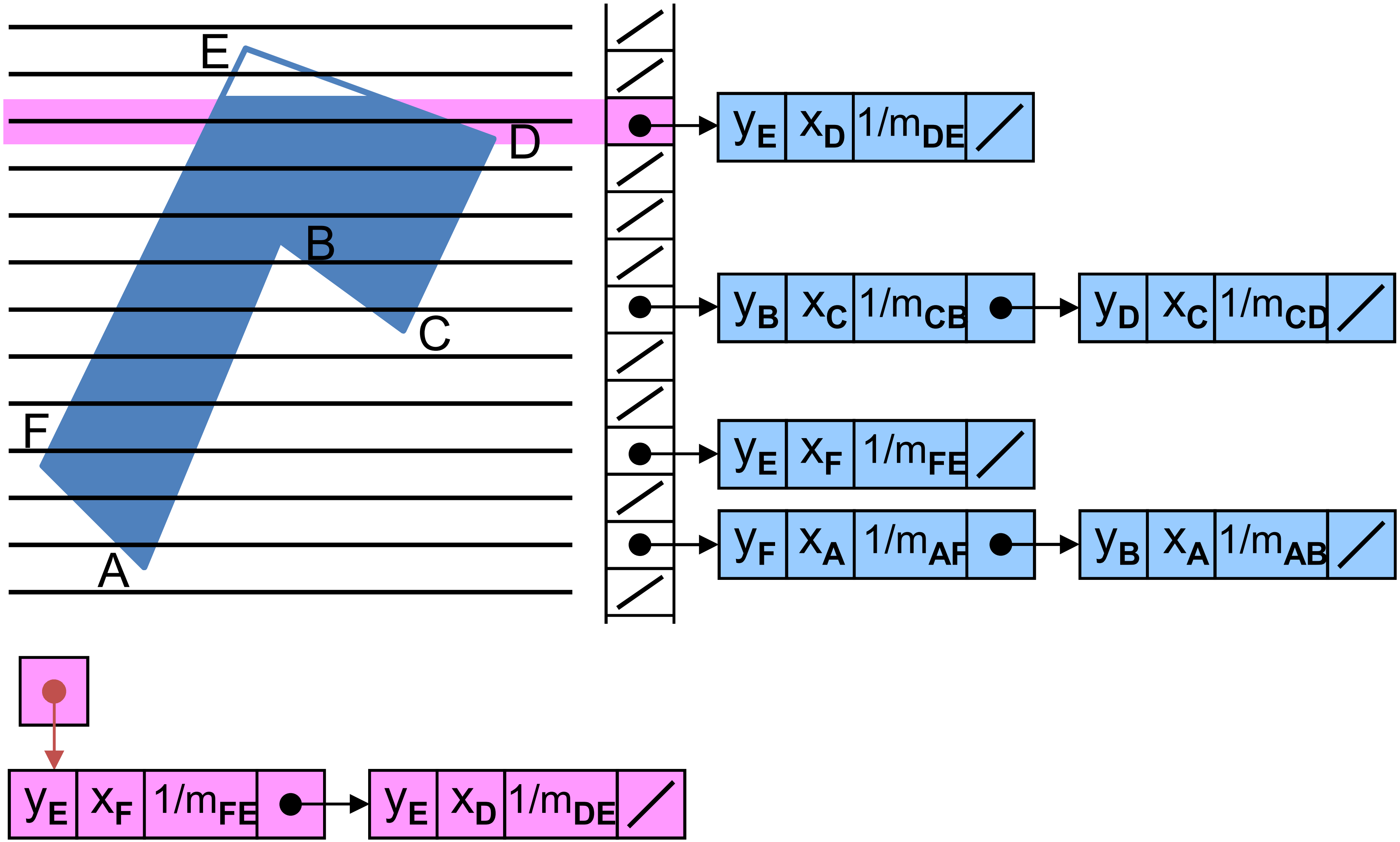
Sorted Edge Table / Active Edge List



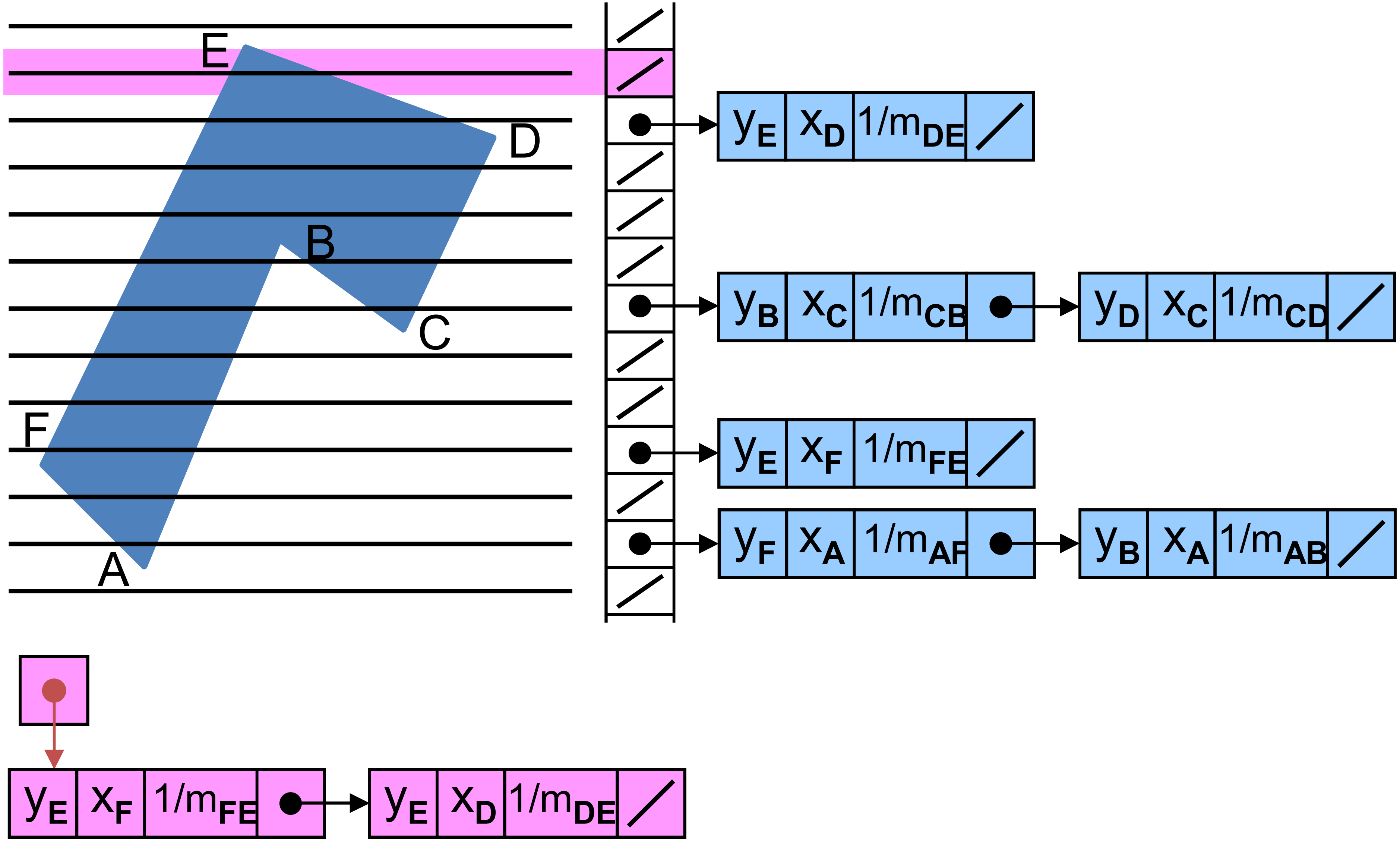
Sorted Edge Table / Active Edge List



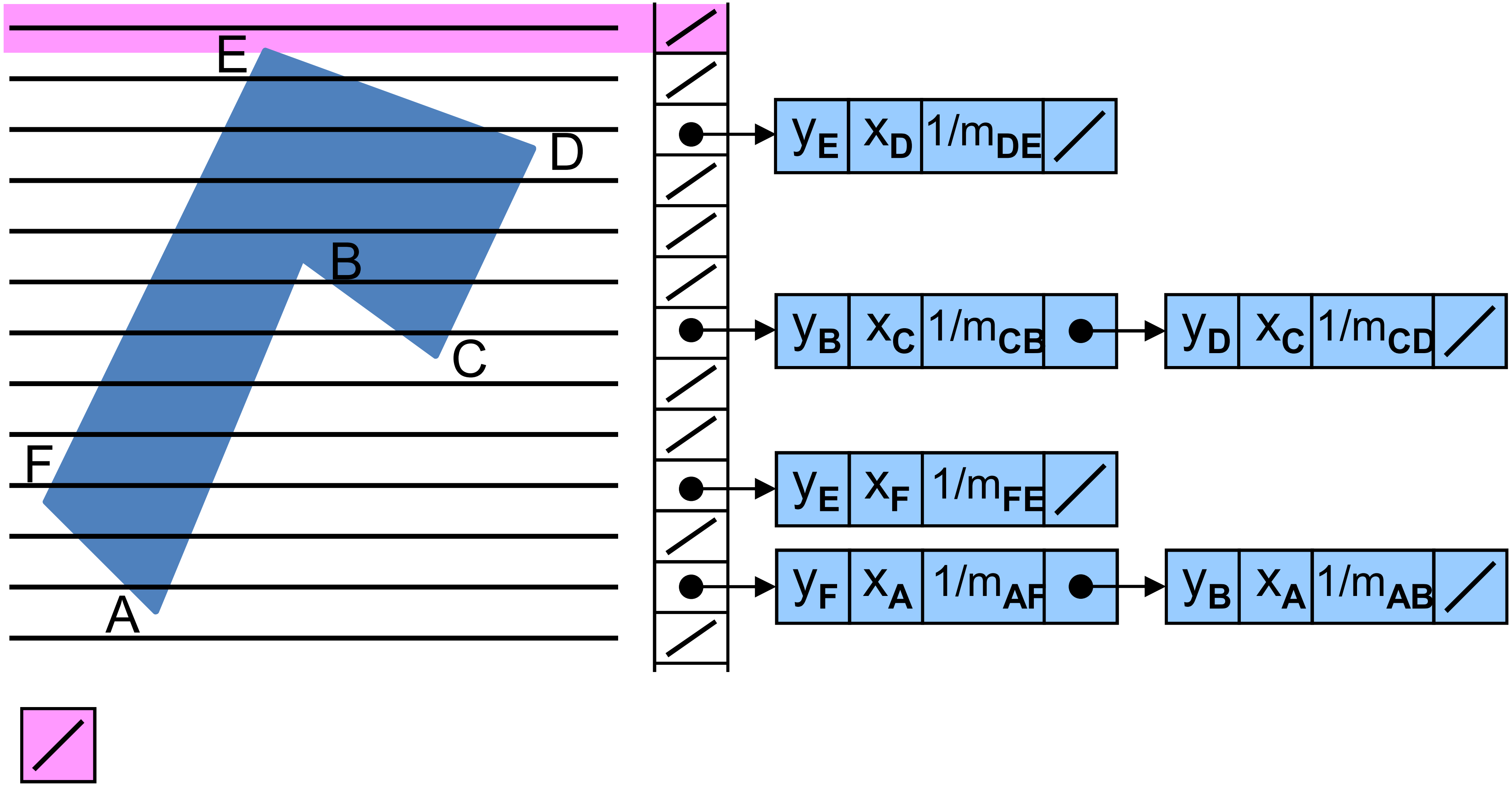
Sorted Edge Table / Active Edge List



Sorted Edge Table / Active Edge List

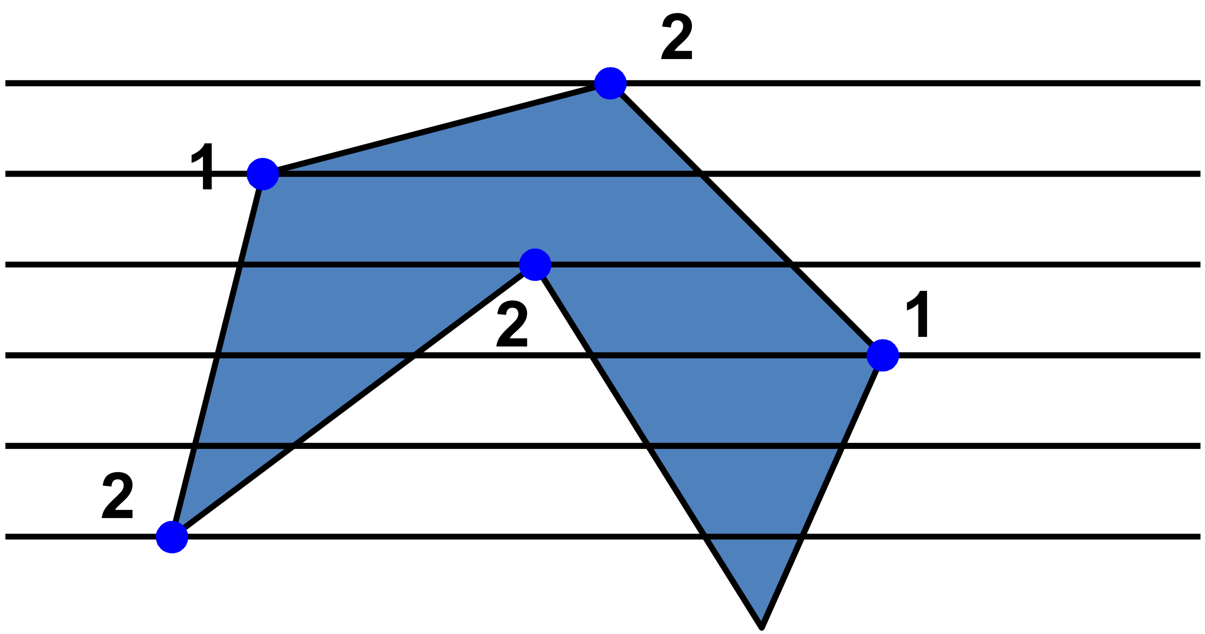


Sorted Edge Table / Active Edge List

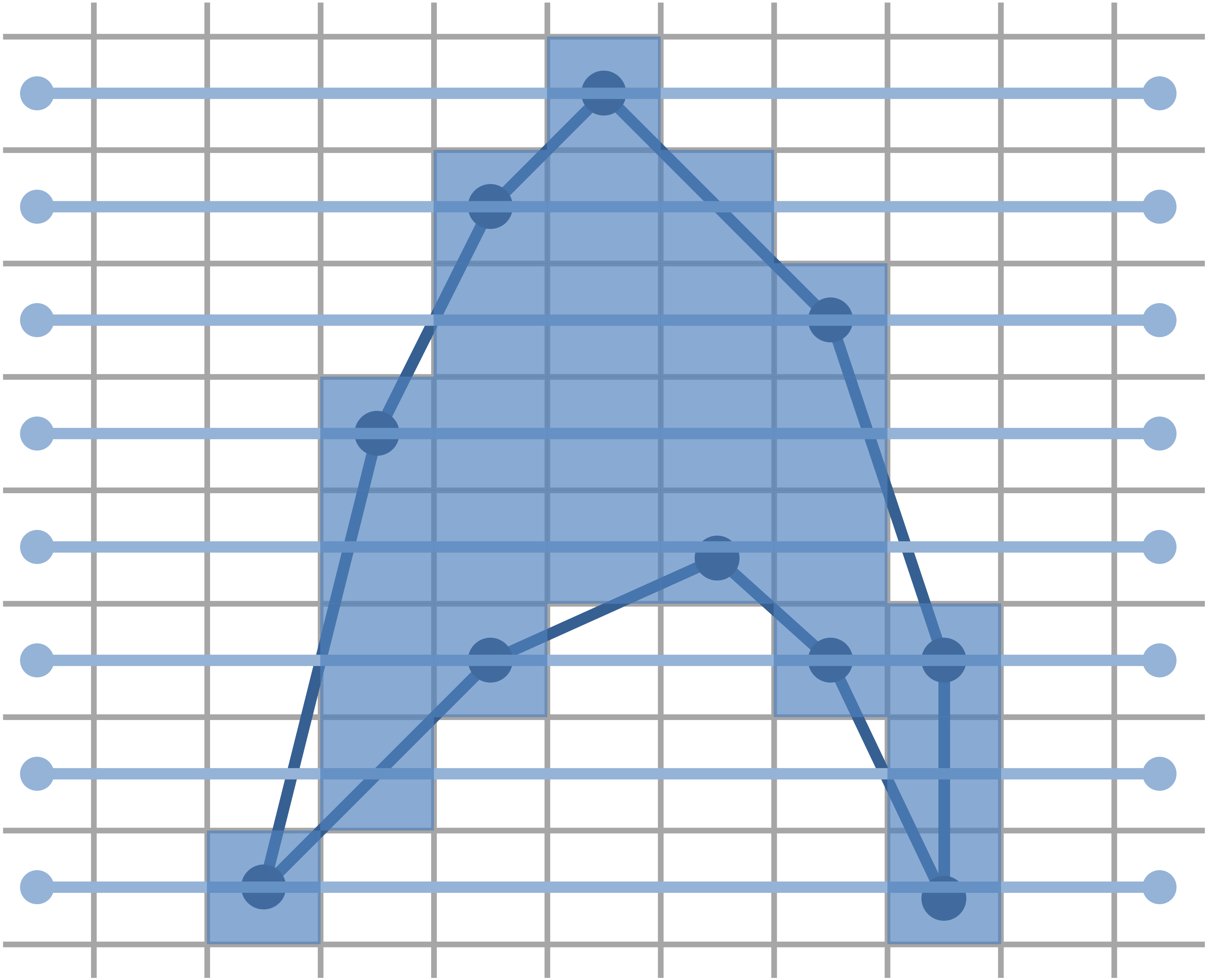


Scan-Line Fill: Intersecting Vertices

- Special case
  - Scan lines that intersect polygon vertices
  - → either special handling (1 or 2 intersections?)
  - → or move vertices up or down by ε

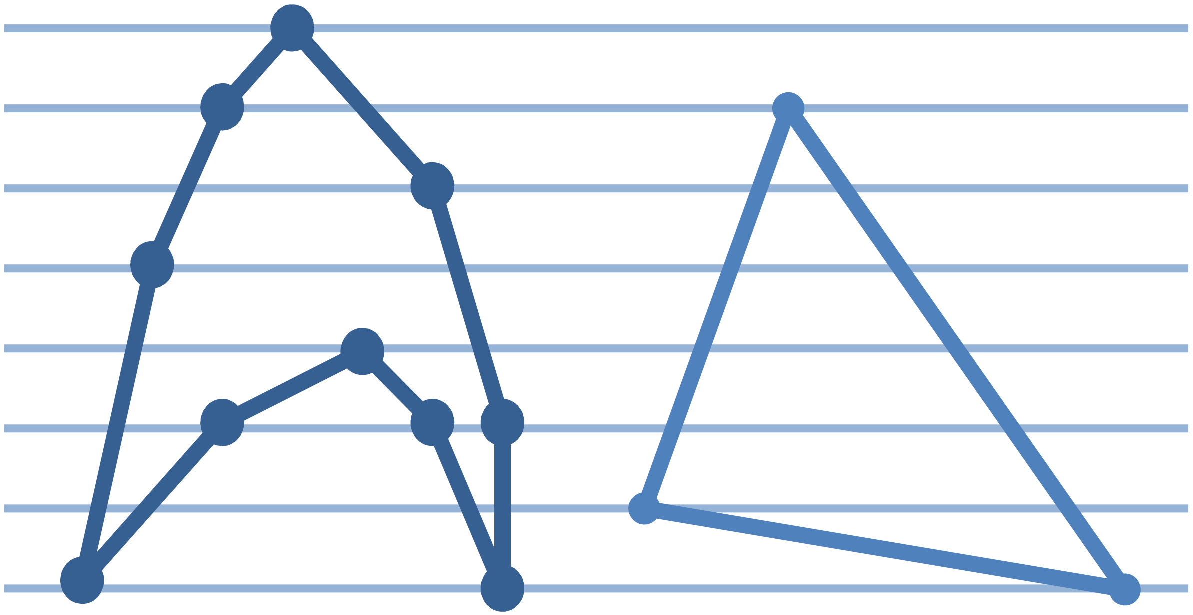


Triangle Rasterization



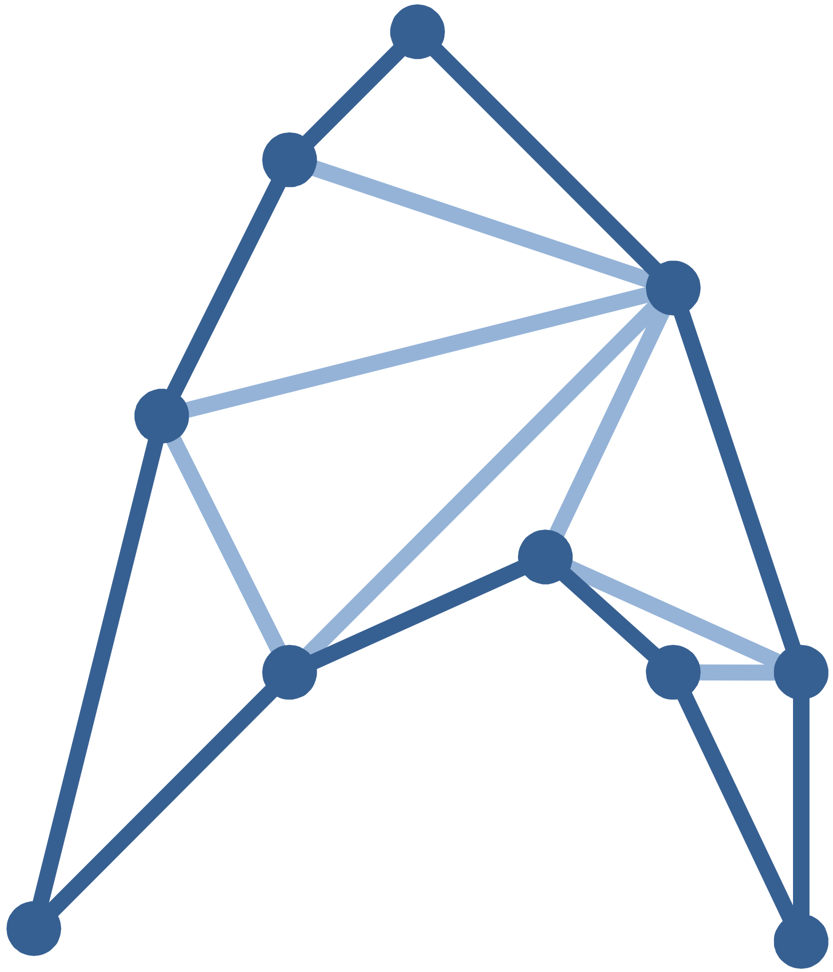
Triangles – Why?

1. Easy to specify
2. Always convex and planar
3. Going to 3D is easy
4. All polygons can be broken into triangles



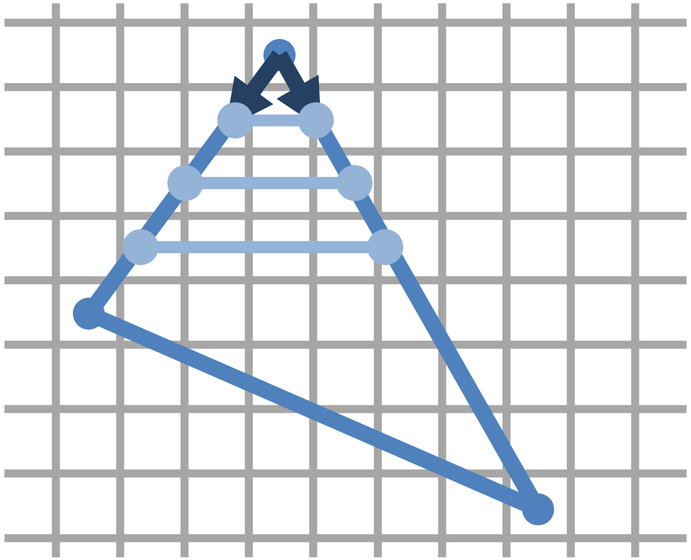
Triangulation

- Breaking a polygon into triangles
  - Delaunay-Triangulation

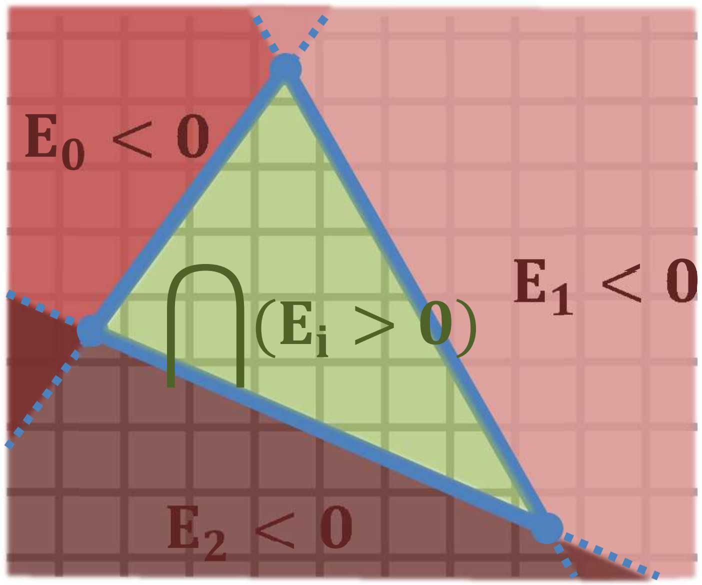


Scan Converting a Triangle

- Edge Walking

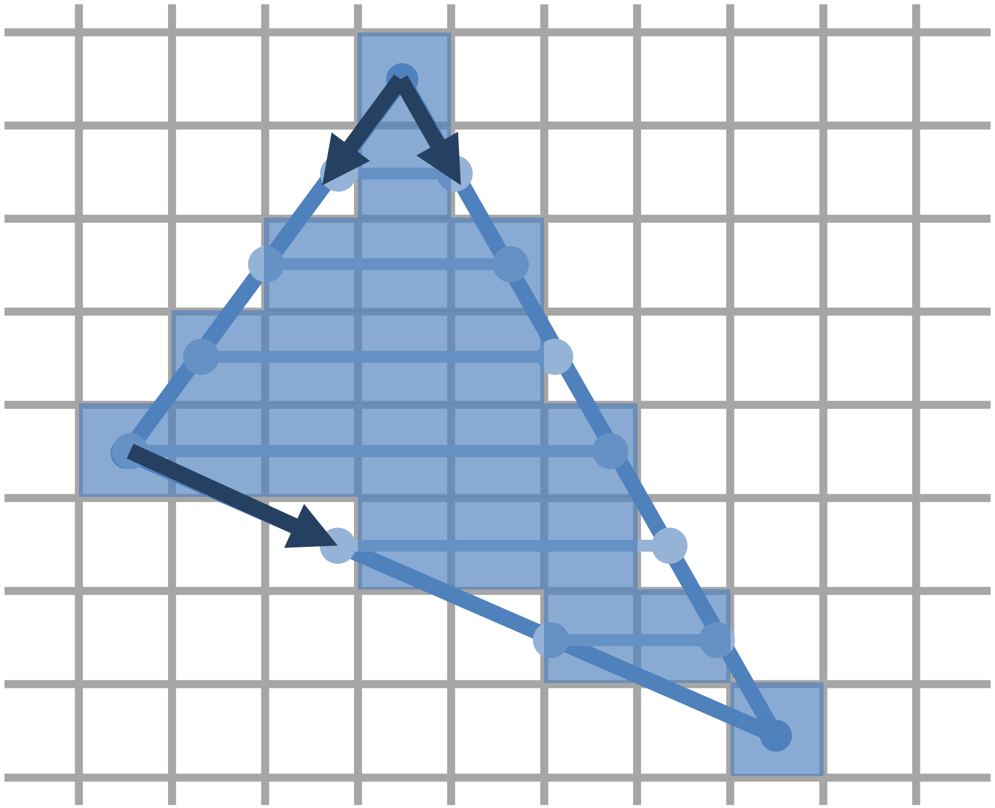


- Edge Equations



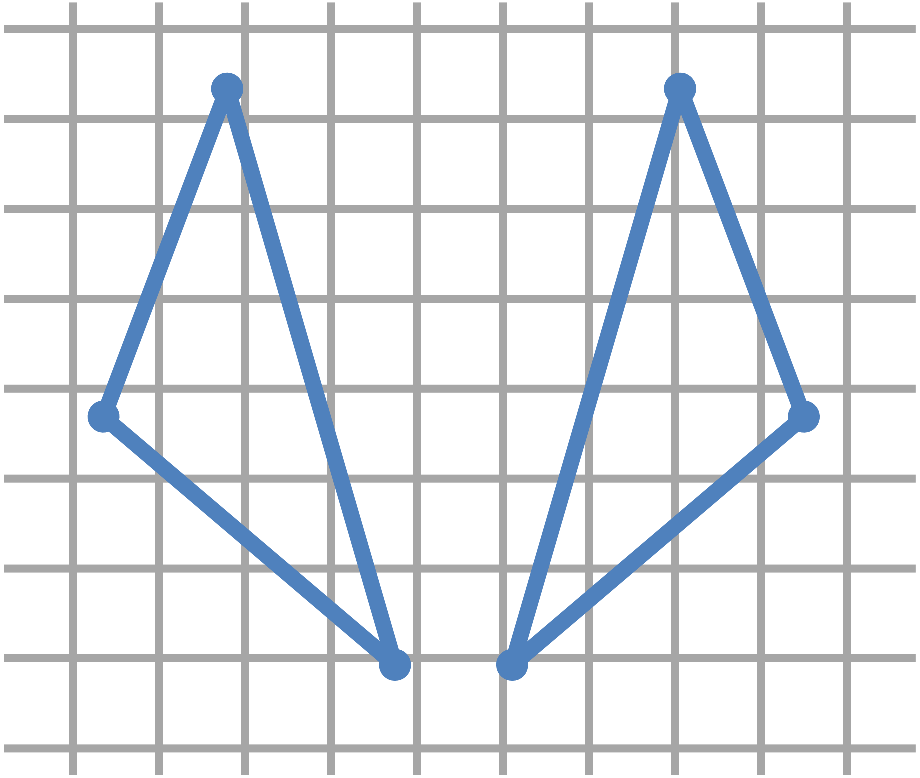
Edge Walking

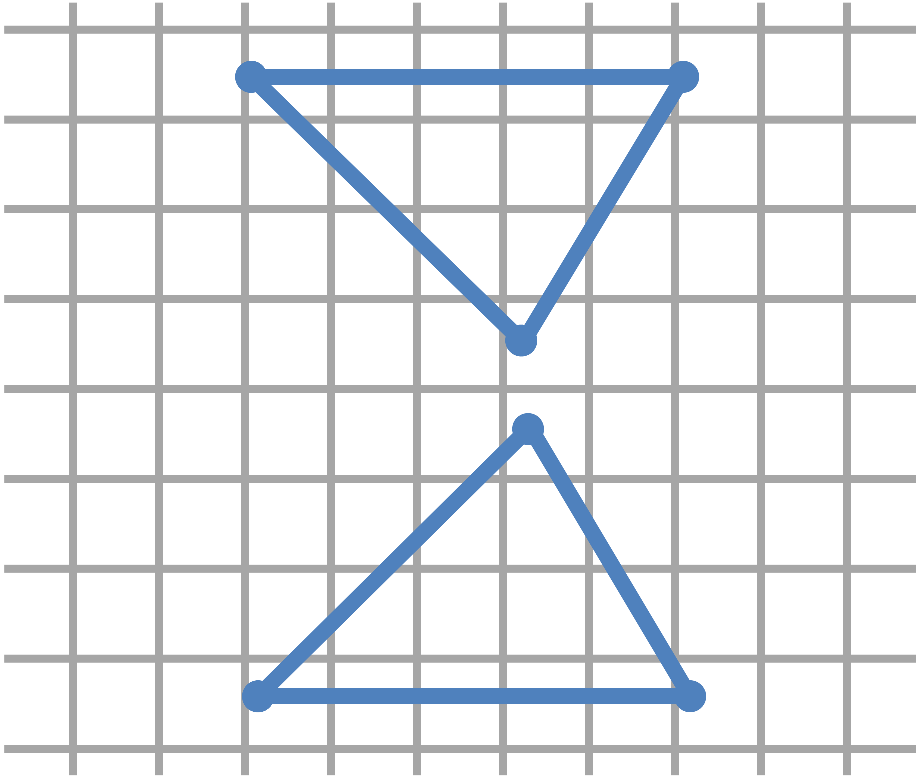
1. Sort vertices in y
2. Walk down edges from extremal y-point
3. Compute spans
4. Switch in 3rd edge
5. Repeat 2 and 3
   until lowest point



Possible Cases

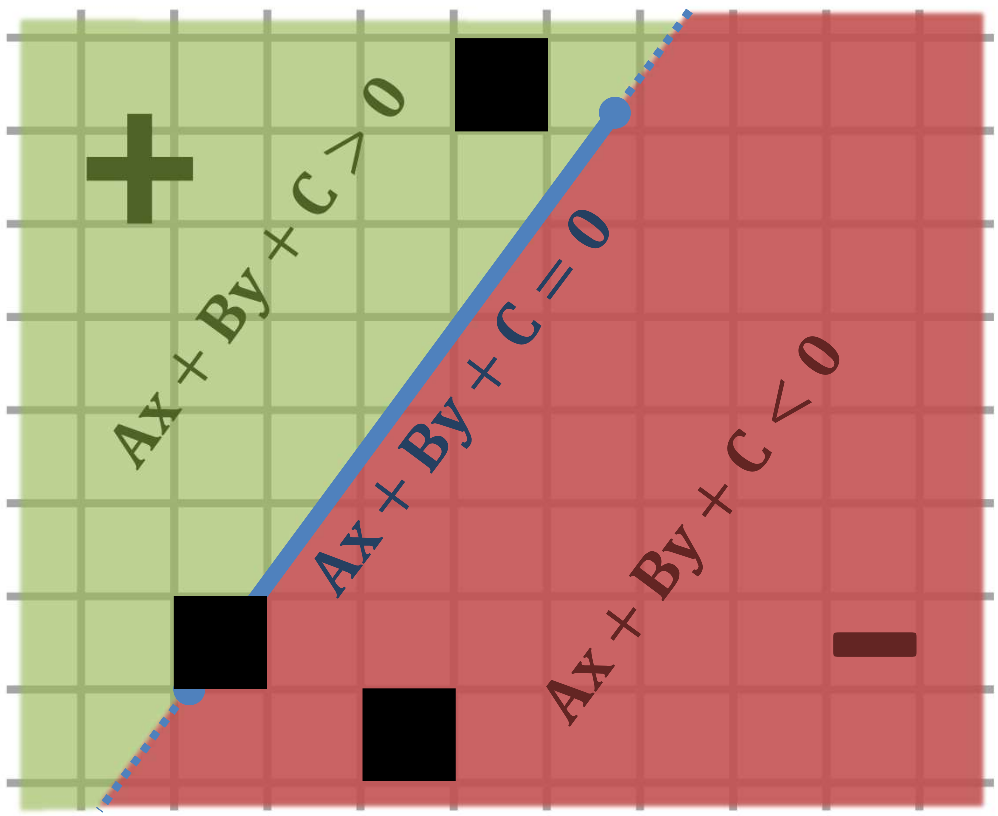
- Left or right y middle point
- 2 highest/lowest points





Edge Equations

- Defines positive/negative half-spaces
- Reverse spaces by multiplication by -1
- $E(x, y) = Ax + By + C$
- Value for pixels?
  - $E(P_x, P_y)$



Given 2 points $\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$, compute A,B,C

1. Setup equation system

$$A x_0 + B y_0 + C = 0 \quad A x_1 + B y_1 + C = 0$$

2. Matrix representation

$$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} + \begin{bmatrix} C \\ C \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \leftrightarrow \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = -C \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

3. Solve

$$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{\begin{vmatrix} x_0 & y_0 \\ x_1 & y_1 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} 1 & y_0 \\ 1 & y_1 \end{vmatrix} \\ \begin{vmatrix} x_0 & 1 \\ x_1 & 1 \end{vmatrix} \end{bmatrix} = \frac{-C}{x_0 y_1 - y_0 x_1} \begin{bmatrix} y_1 - y_0 \\ x_0 - x_1 \end{bmatrix}$$

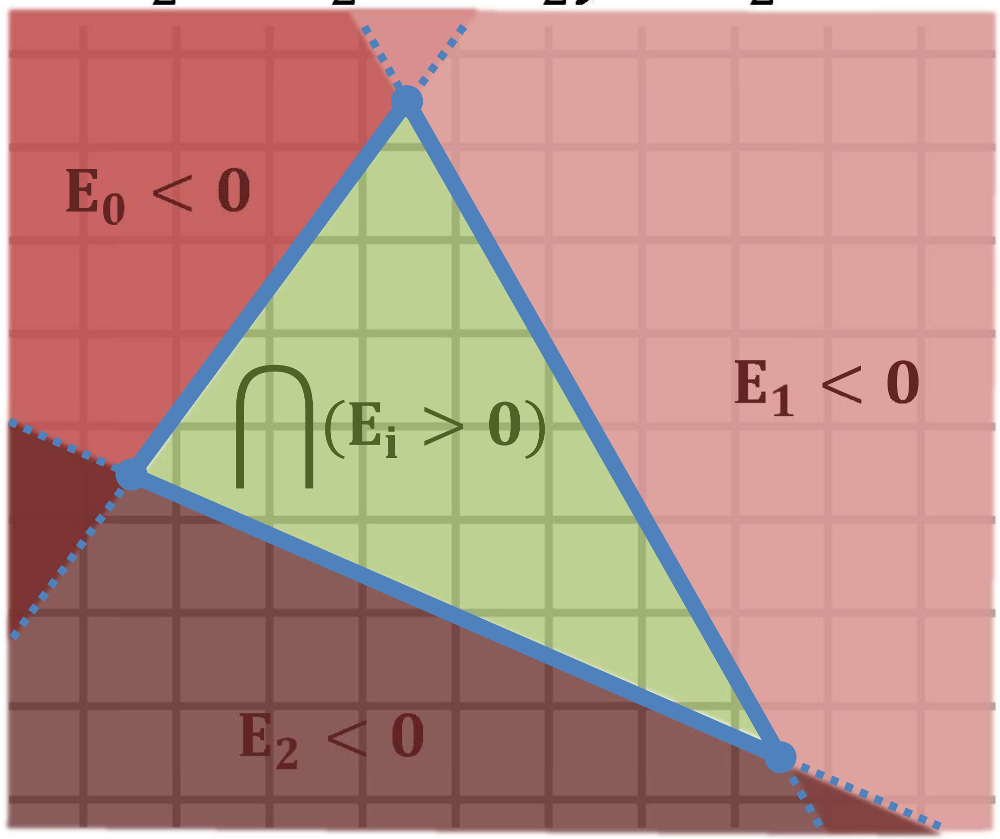
4. Choose C

Edge Equations for the Triangle

$$E_0 = A_0 x + B_0 y + C_0 = 0$$

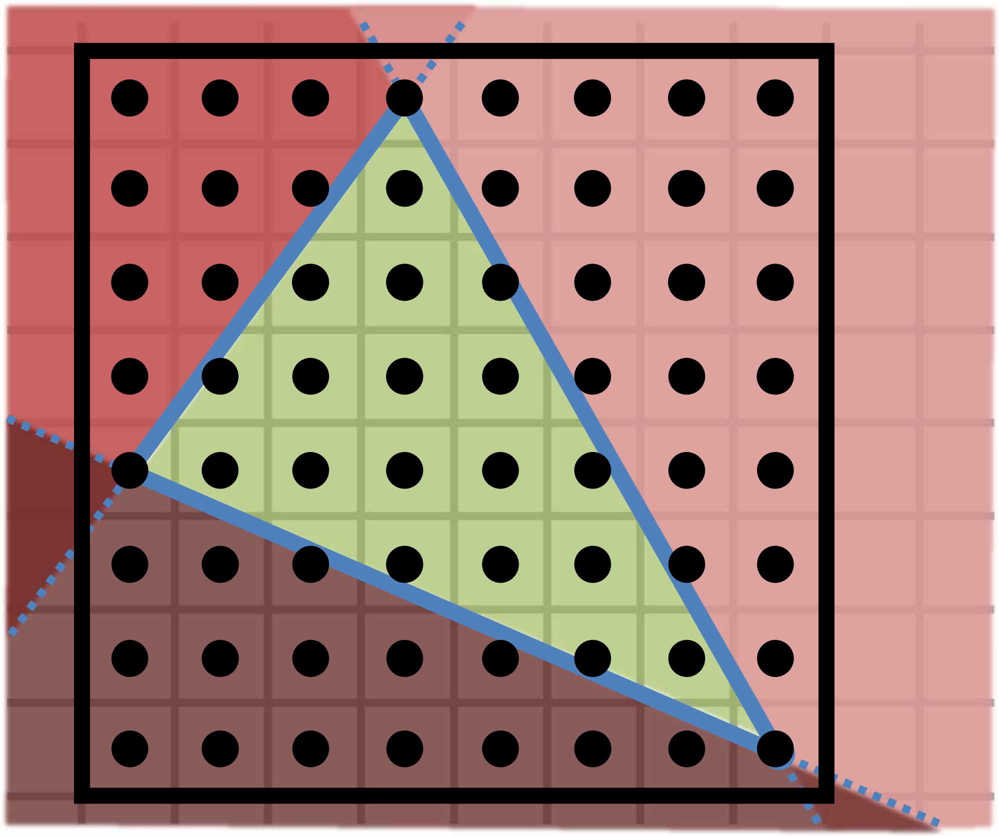
$$E_1 = A_1 x + B_1 y + C_1 = 0$$

$$E_2 = A_2 x + B_2 y + C_2 = 0$$



Testing Pixels

- Find bounding box
- Test $\bigcap(\mathbf{E_i} > \mathbf{0})$ for each pixel
- Happy?



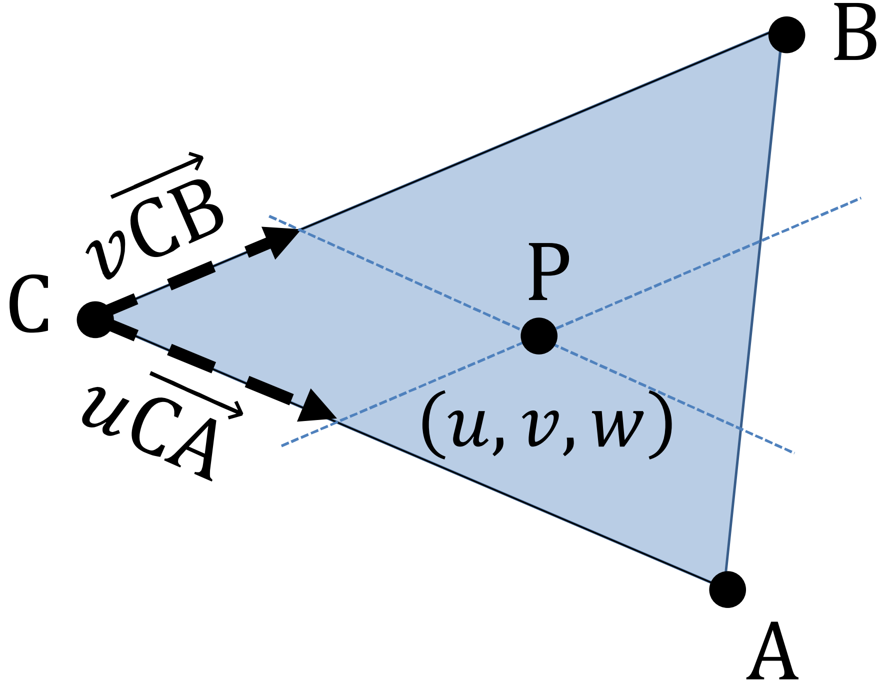
Barycentric Coordinates of P

- Define $\mathrm{P} = \mathrm{C} + u\overrightarrow{\mathrm{CA}} + v\overrightarrow{\mathrm{CB}}$

$$= u\mathrm{A} + v\mathrm{B} + (1 - u - v)\mathrm{C}$$

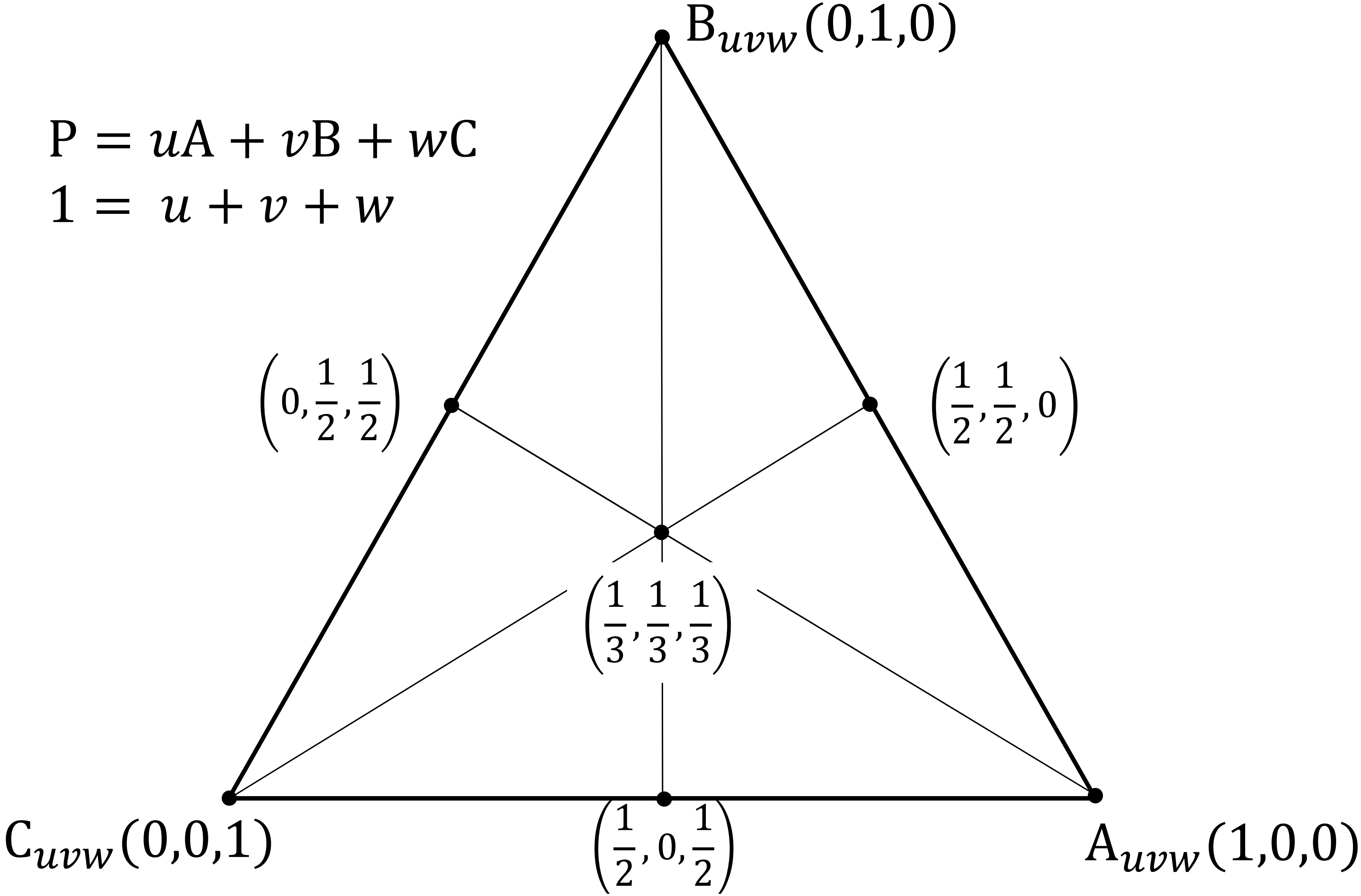
$$= u\mathrm{A} + v\mathrm{B} + w\mathrm{C} \quad \text{with} \quad 1 = u + v + w$$

- Triangle can also be 3d



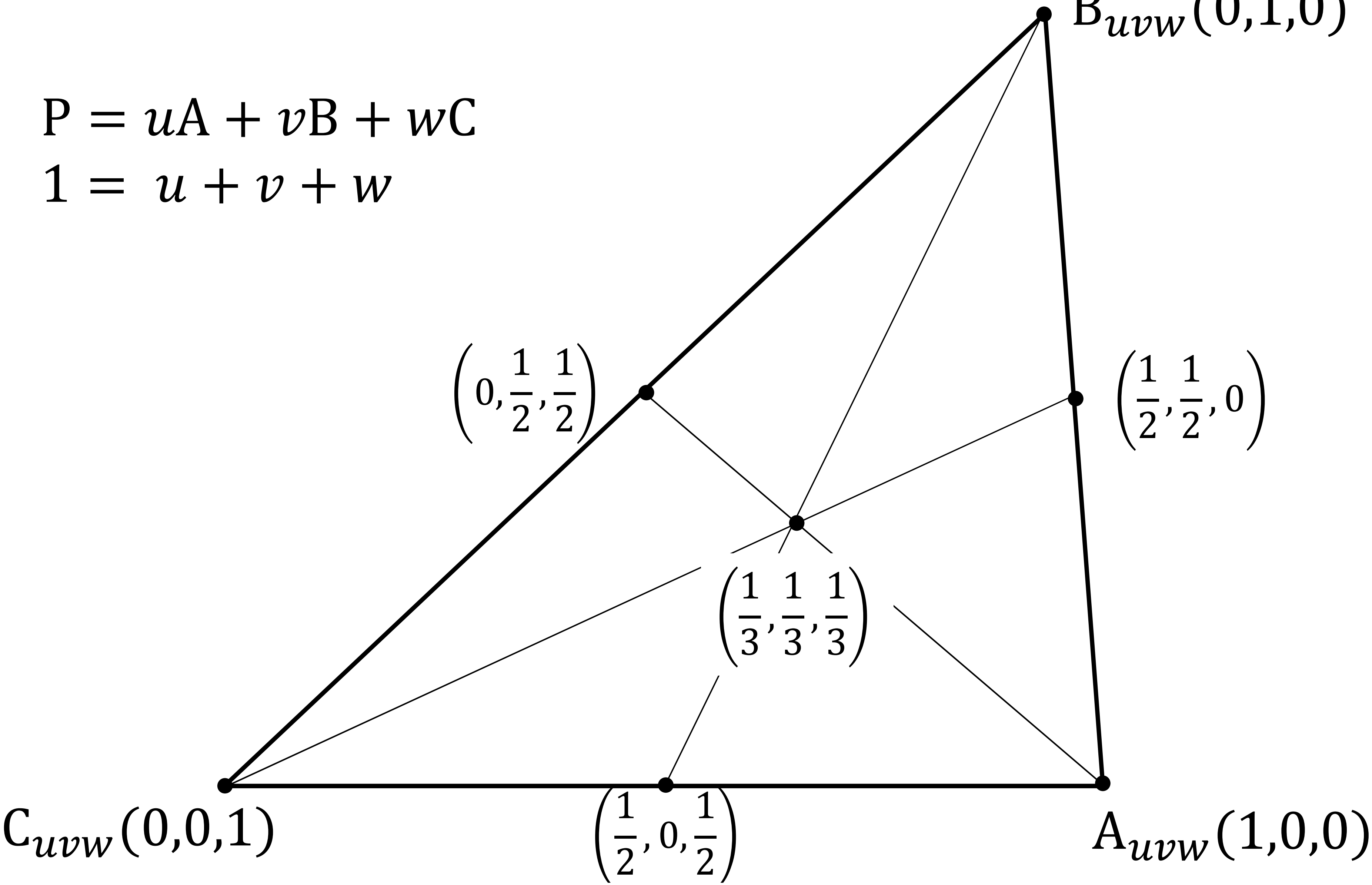
BC – Special Points

$$P = uA + vB + wC$$
$$1 = u + v + w$$



Barycentric Coordinates – Invariance

$$P = uA + vB + wC$$
$$1 = u + v + w$$



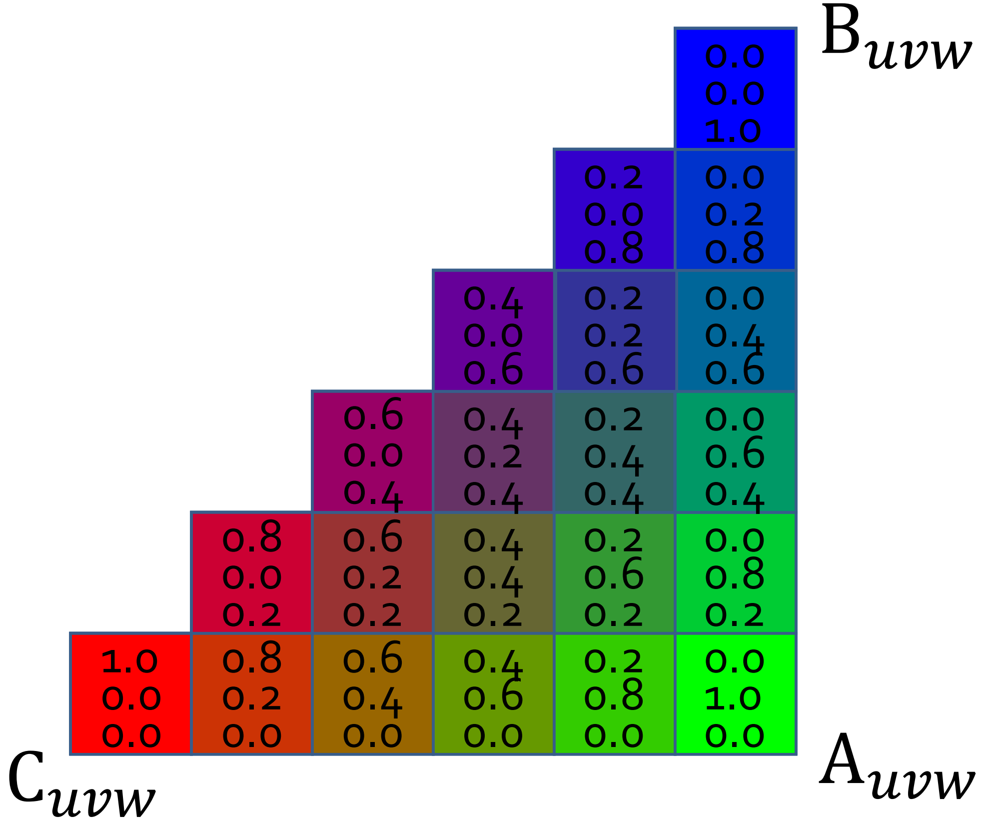
BC – Inside Triangle Test

- Also outside triangle
- In triangle if (u, v, w) all same sign
  - For CCW $(u, v, w) \geq 0$

1.2 -1.4 1.2	1.0 -1.2 1.2	0.8 -1.0 1.2	0.6 -0.8 1.2	0.4 -0.6 1.2	0.2 -0.4 1.2	0.0 -0.2 1.2	-0.2 0.0 1.2
1.2 -1.2 1.0	1.0 -1.0 1.0	0.8 -0.8 1.0	0.6 -0.6 1.0	0.4 -0.4 1.0	0.2 -0.2 1.0	0.0 0.0 1.0	-0.2 0.2 1.0
1.2 -1.0 0.8	1.0 -0.8 0.8	0.8 -0.6 0.8	0.6 -0.4 0.8	0.4 -0.2 0.8	0.2 0.0 0.8	0.0 0.2 0.8	-0.2 0.4 0.8
1.2 -0.8 0.6	1.0 -0.6 0.6	0.8 -0.4 0.6	0.6 -0.2 0.6	0.4 0.0 0.6	0.2 0.2 0.6	0.0 0.4 0.6	-0.2 0.6 0.6
1.2 -0.6 0.4	1.0 -0.4 0.4	0.8 -0.2 0.4	0.6 0.0 0.4	0.4 0.2 0.4	0.2 0.4 0.4	0.0 0.6 0.4	-0.2 0.8 0.4
1.2 -0.4 0.2	1.0 -0.2 0.2	0.8 0.0 0.2	0.6 0.2 0.2	0.4 0.4 0.2	0.2 0.6 0.2	0.0 0.8 0.2	-0.2 1.0 0.2
1.2 -0.2 0.0	1.0 0.0 0.0	0.8 0.2 0.0	0.6 0.4 0.0	0.4 0.6 0.0	0.2 0.8 0.0	0.0 1.0 0.0	-0.2 1.2 0.0
1.2 0.0 -0.2	1.0 0.2 -0.2	0.8 0.4 -0.2	0.6 0.6 -0.2	0.4 0.8 -0.2	0.2 1.0 -0.2	0.0 1.2 -0.2	-0.2 1.4 -0.2

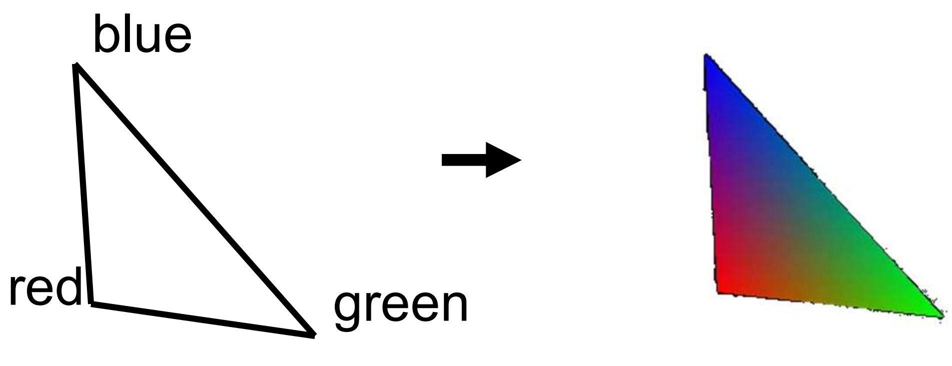
BC – Color Interpolation

- $\mathrm{P} = u\mathrm{A} + v\mathrm{B} + w\mathrm{C}$
- $\mathrm{P} = u\langle Green\rangle + v\langle Blue\rangle + w\langle Red\rangle$
- A.k.a. Gouraud interpolation



Interpolation

- Interpolate per point (a.k.a vertex) attributes (ex.: colors, z-value) over the triangle
- Attribute value for a point P
  - Easy with barycentric coordinates
  - $\mathrm{P} = u\mathrm{A} + v\mathrm{B} + w\mathrm{C}$
  - $P_{attrib.} = uA_{attrib.} + vB_{attrib.} + wC_{attrib.}$

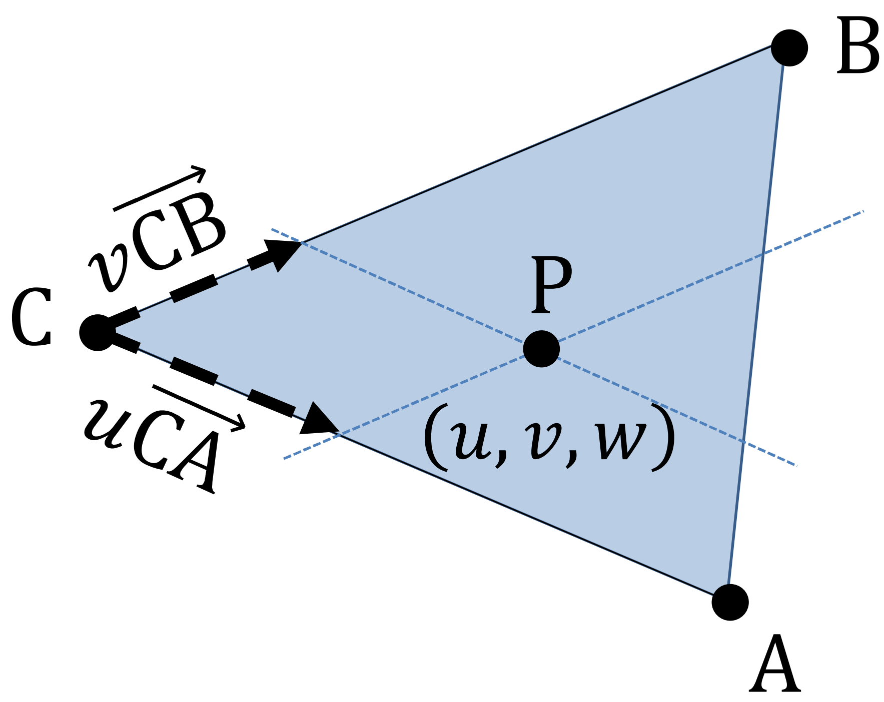


Barycentric Coordinates of P (2D)

$$\mathrm{P} = \mathrm{C} + u\overrightarrow{\mathrm{CA}} + v\overrightarrow{\mathrm{CB}}$$

$$\begin{pmatrix}\overrightarrow{\mathrm{CA}} & \overrightarrow{\mathrm{CB}}\end{pmatrix}\begin{pmatrix}u\\ v\end{pmatrix} = \mathrm{P} - \mathrm{C}$$

$$\begin{pmatrix}\mathrm{A} - \mathrm{C} & \mathrm{B} - \mathrm{C}\end{pmatrix}\begin{pmatrix}u\\ v\end{pmatrix} = \mathrm{P} - \mathrm{C}$$



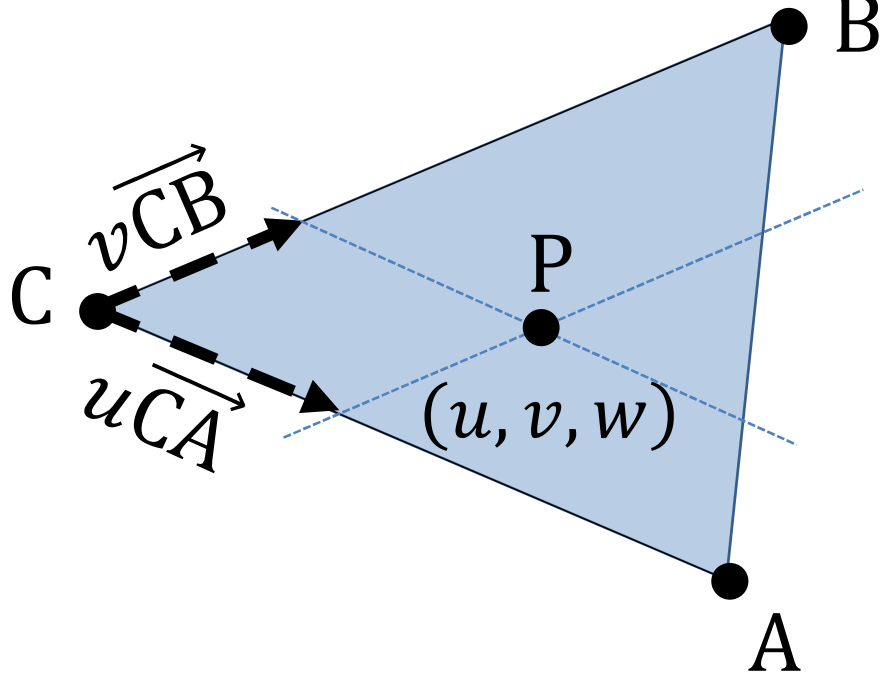
Barycentric Coordinates of P (2D)

- Cramer's Rule

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{|A-C \quad B-C|} \begin{pmatrix} |P-C \quad B-C| \\ |A-C \quad P-C| \end{pmatrix}$$

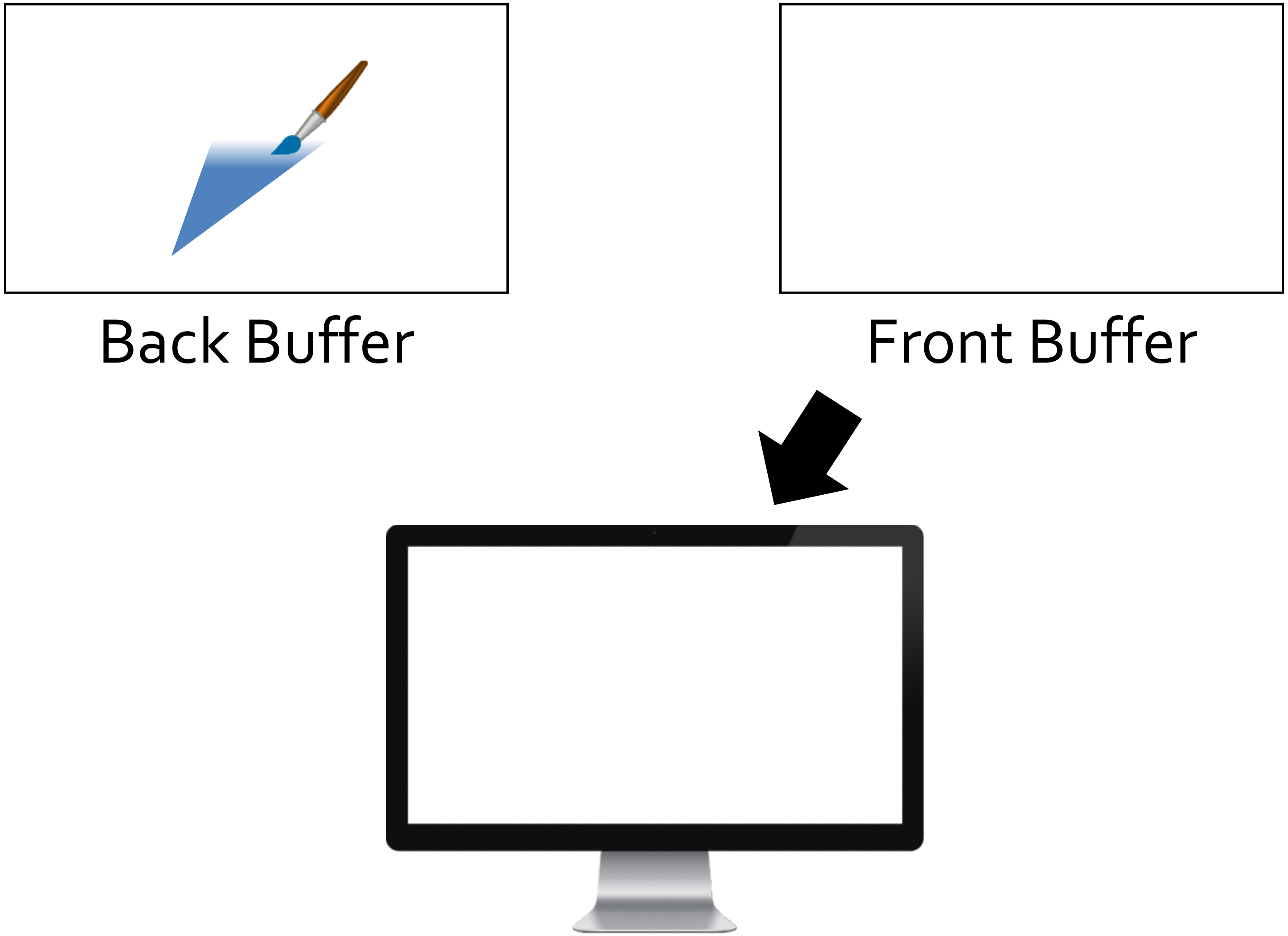
- Point is inside triangle iff (means if and only if)

$$u \geq 0 \ \cap v \geq 0 \cap (u+v) \leq 1$$

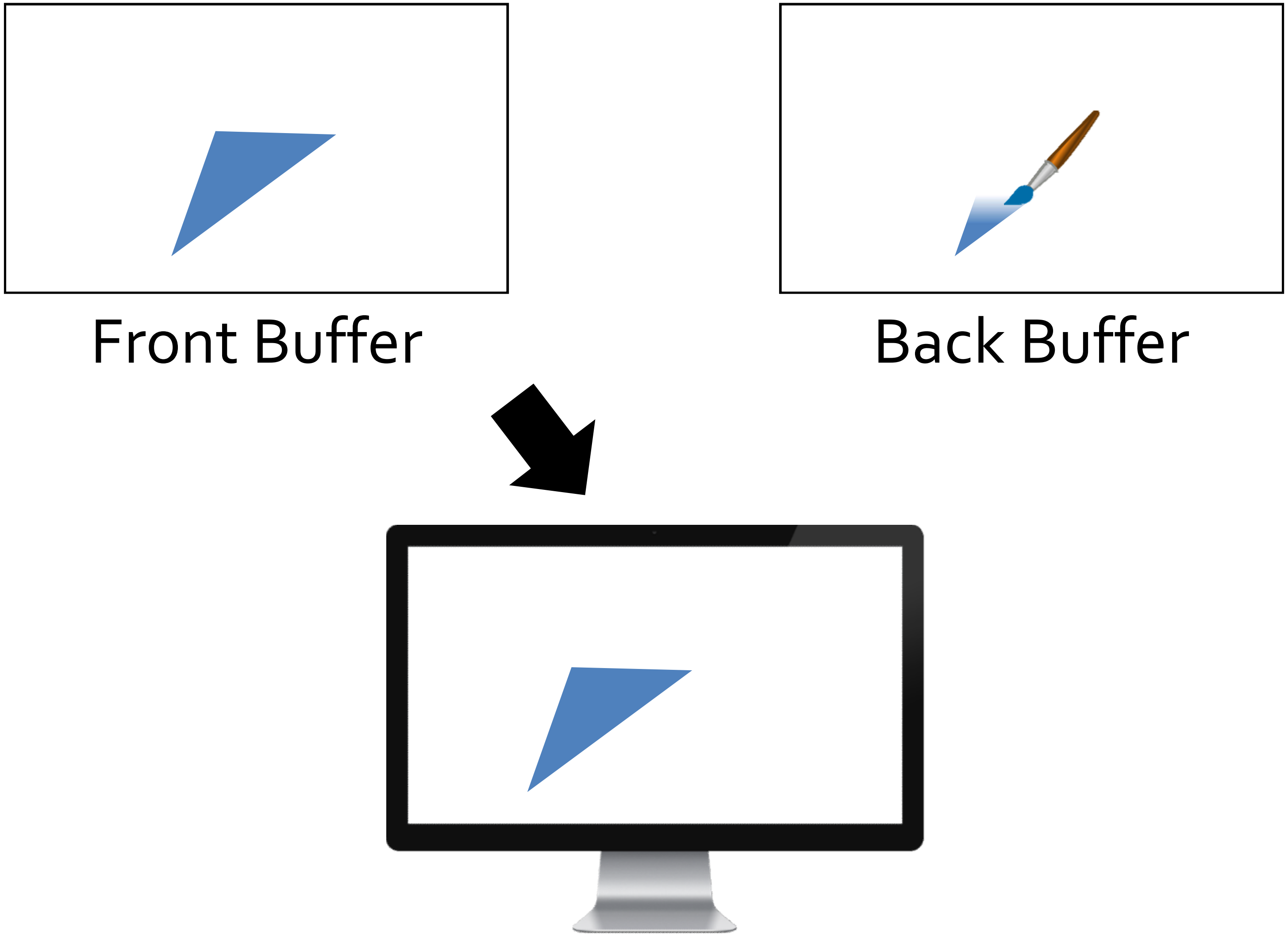


Rasterization in Time

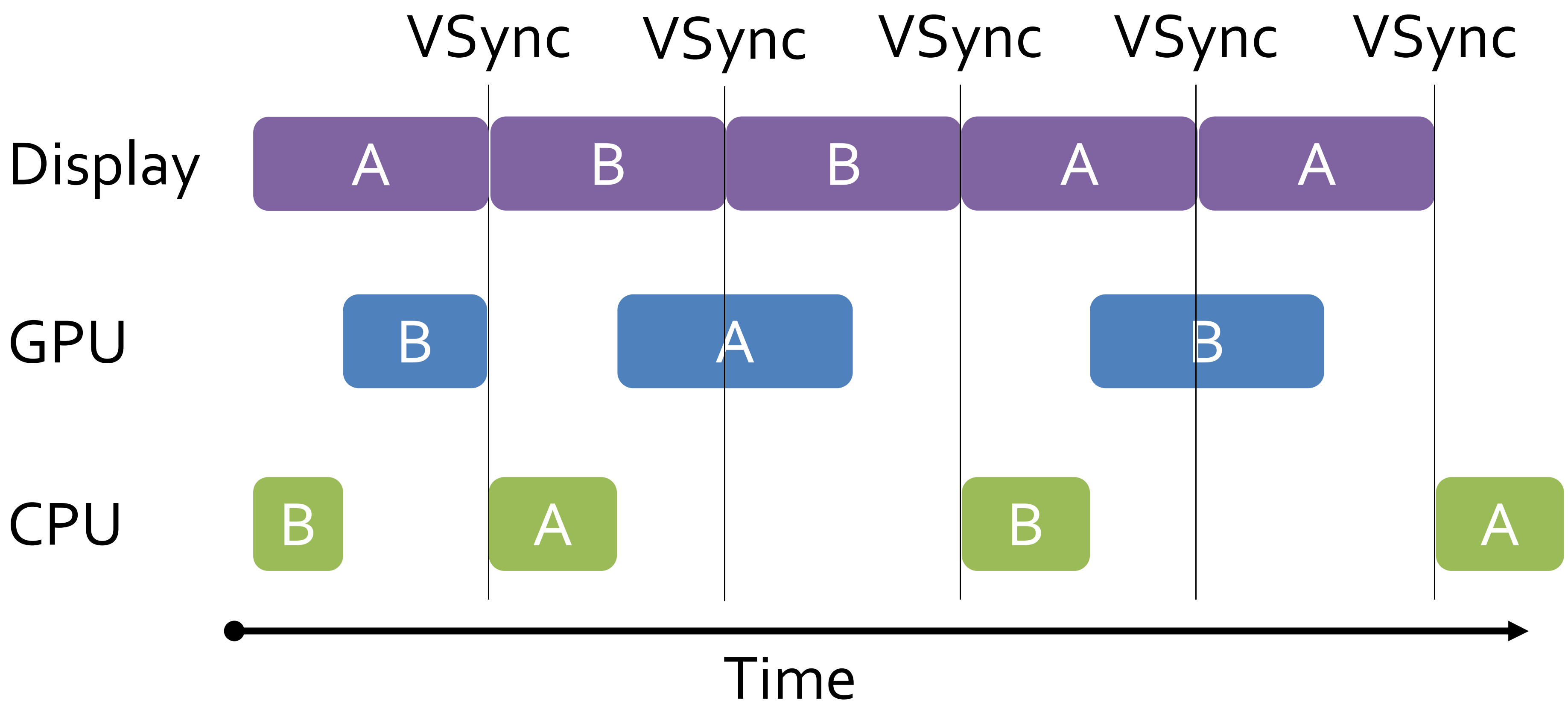
Double Buffering (2 Frame Buffer)



Double Buffering



Vertical Synchronisation (VSync)



Tripple Buffering