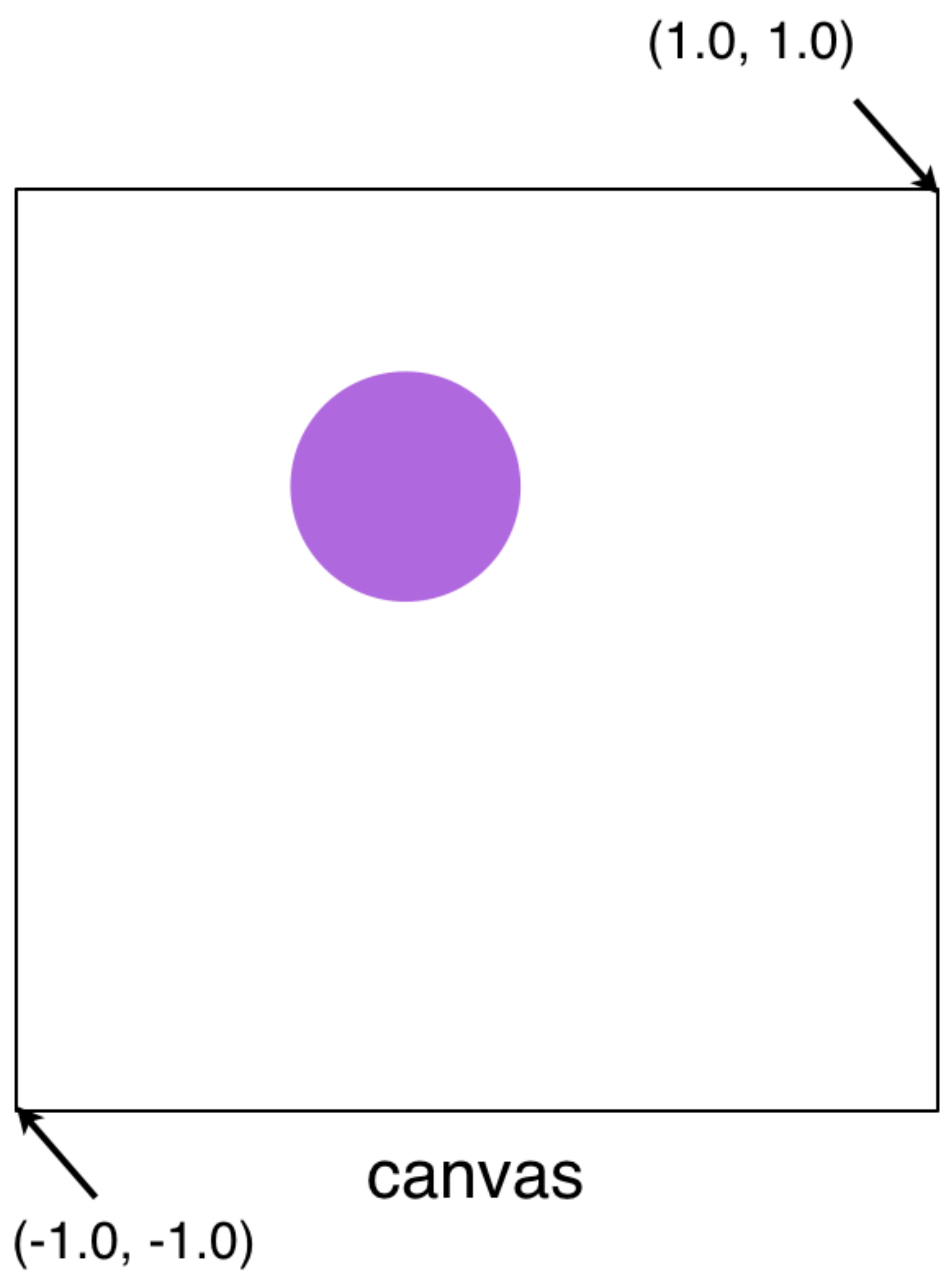
Computer Graphics

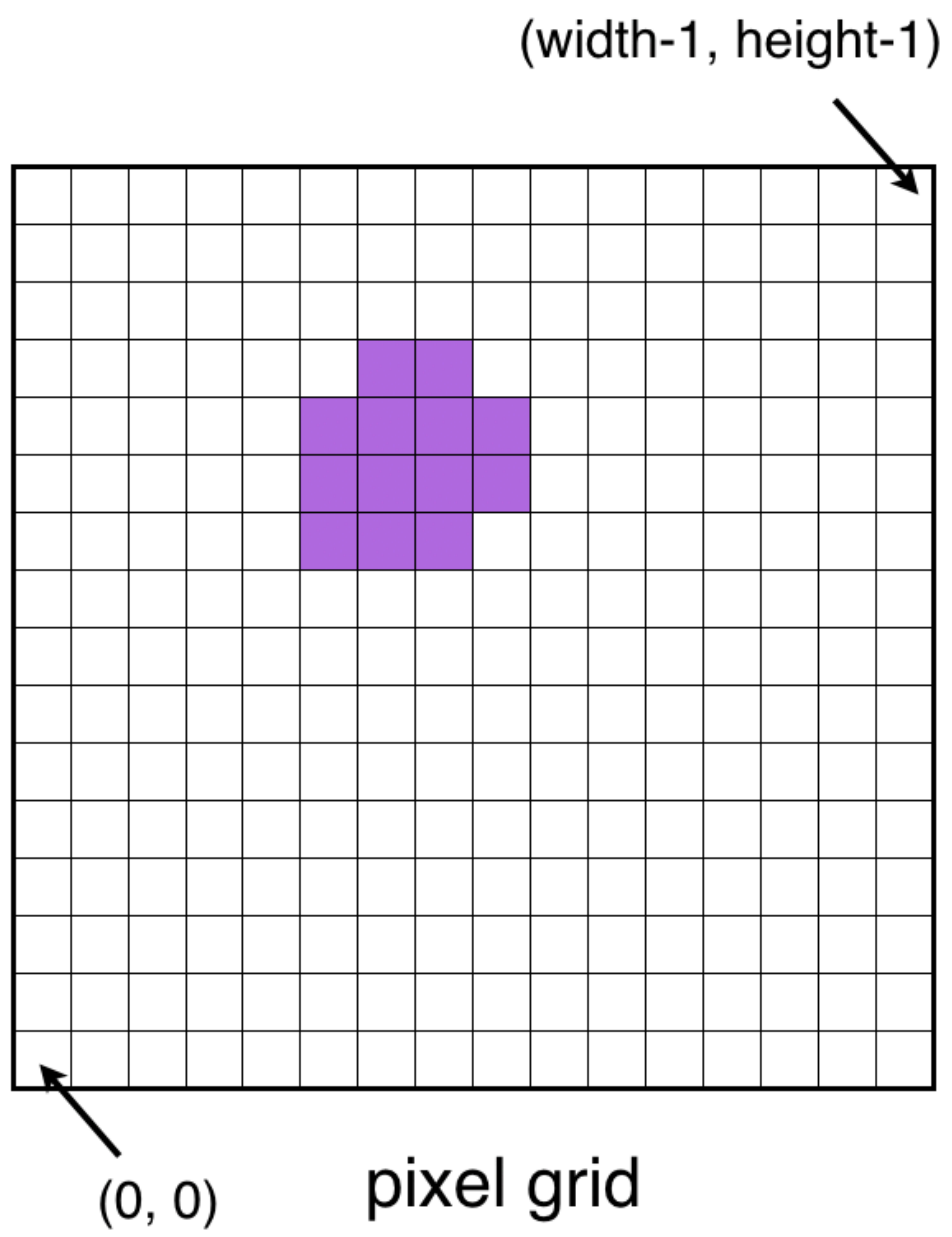
- Rasterization

Junjie Cao @ DLUT
Spring 2019
http://jjcao.github.io/ComputerGraphics/

Pleasure may come from illusion, but happiness can come only of reality.

2D Canvas





The graphics pipeline

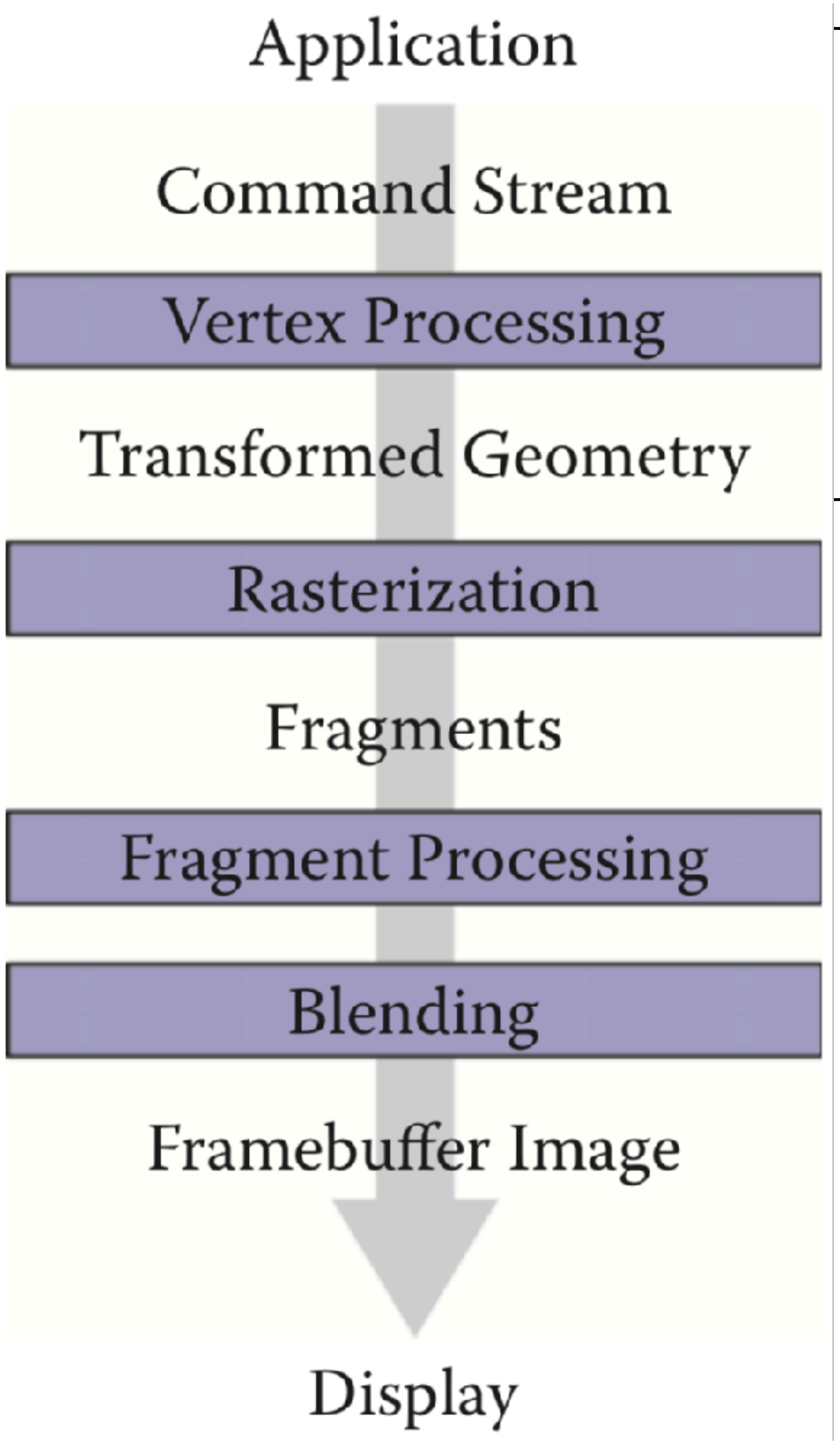
- **The 2nd major approach to rendering**
  - Image-order rendering: simpler, flexible, (usually) more execution time
  - Object-order rendering: efficiency

- **The standard approach to object-order graphics. Many versions exist**
  - software, e.g. Pixar's REYES architecture, used in film production
    - many options for quality and flexibility
  - hardware, e.g. graphics cards in PCs, for game, visualization, UI
    - amazing performance: millions of triangles per frame

- **We'll focus on an abstract version of hardware pipeline**

- **"Pipeline" because of the many stages**
  - very parallelizable
  - leads to remarkable performance of graphics cards (many times the flops of the CPU at ~1/5 the clock speed)

The graphics pipeline



Operations to geometry, matrix transformations => screen coords

Operations to fragments, HSR

The rasterizer breaks each primitive into a number of *fragments*, one for each pixel covered by the primitive.
various fragments corresponding to each pixel are combined in the *fragment blending stage*

Primitives

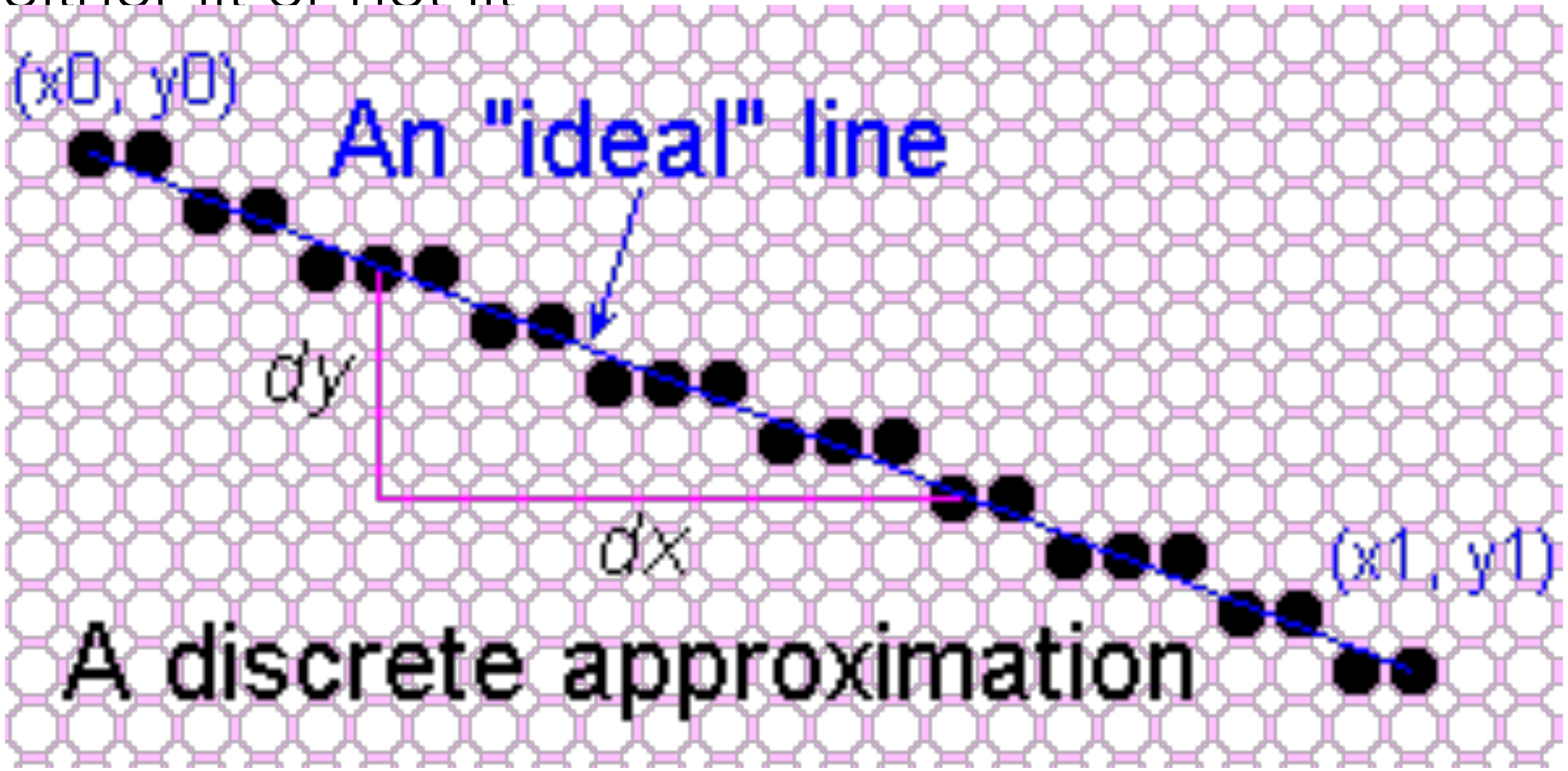
- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved surfaces? Approximate them with triangles
- Trend over the decades: toward minimal primitives
  - simple, uniform, repetitive: good for parallelism

Rasterization

- Input: primitives
- Output: fragments with attributes per pixel. |{Fragments_i}| = |objects covered the pixel|
  - First job: enumerate the pixels covered by a primitive
    - simple, aliased definition: pixels whose centers fall inside
  - Second job: interpolate attributes across the primitive
    - e.g. colors computed at vertices – e.g. normals at vertices
    - e.g. texture coordinates

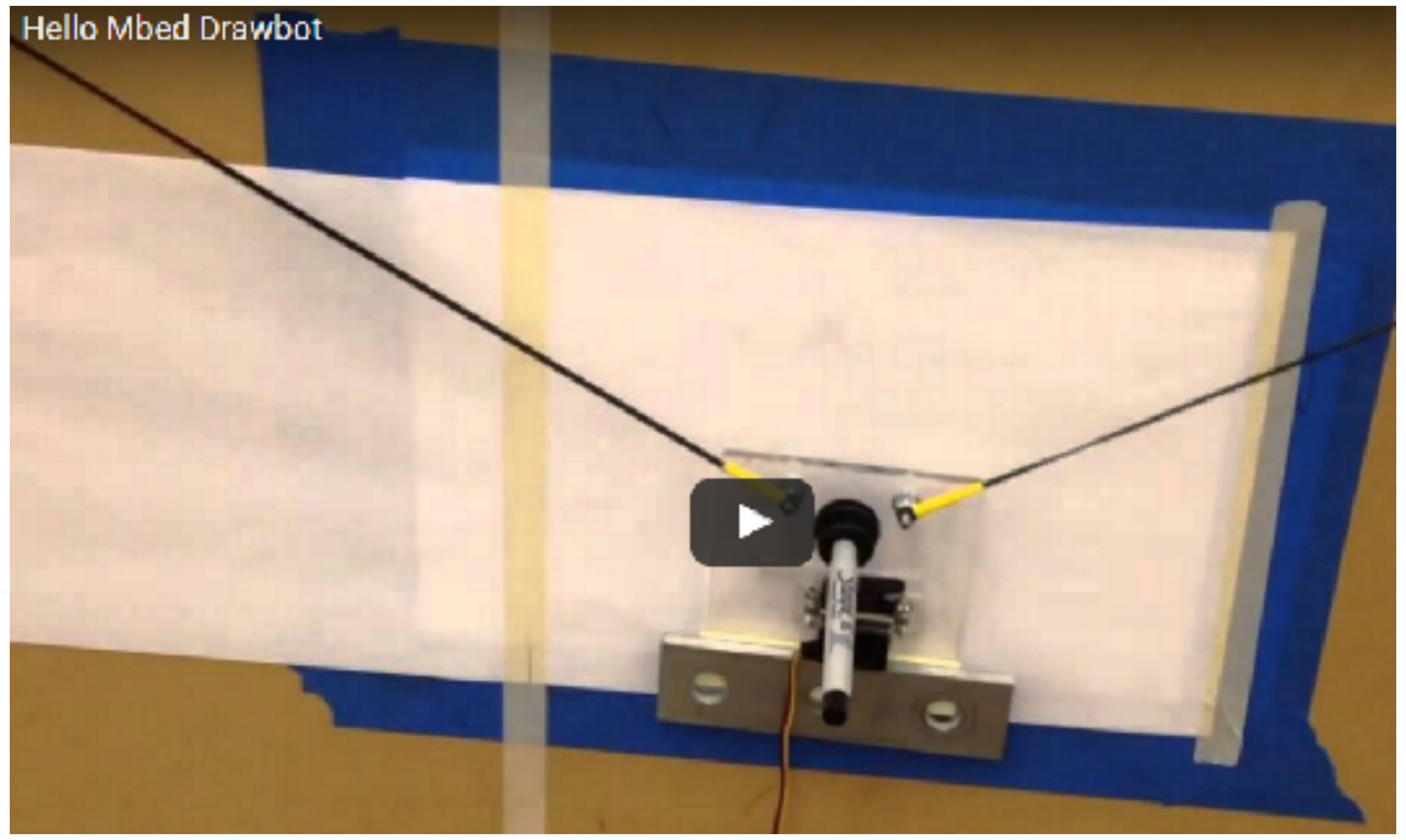
Towards the Ideal Line

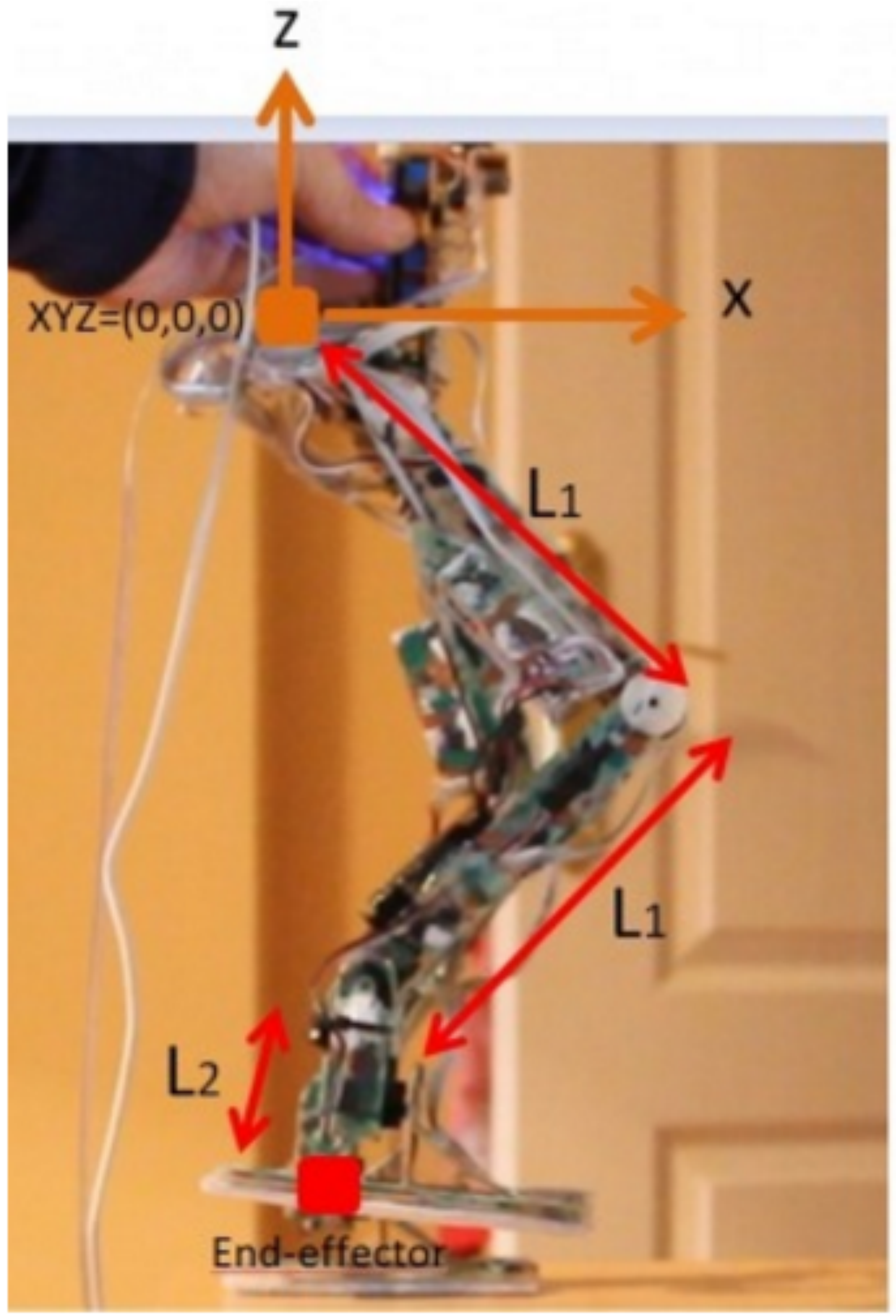
- We can only do a discrete approximation
- Illuminate pixels as close to the true path as possible, consider bi-level display only
  - Pixels are either lit or not lit

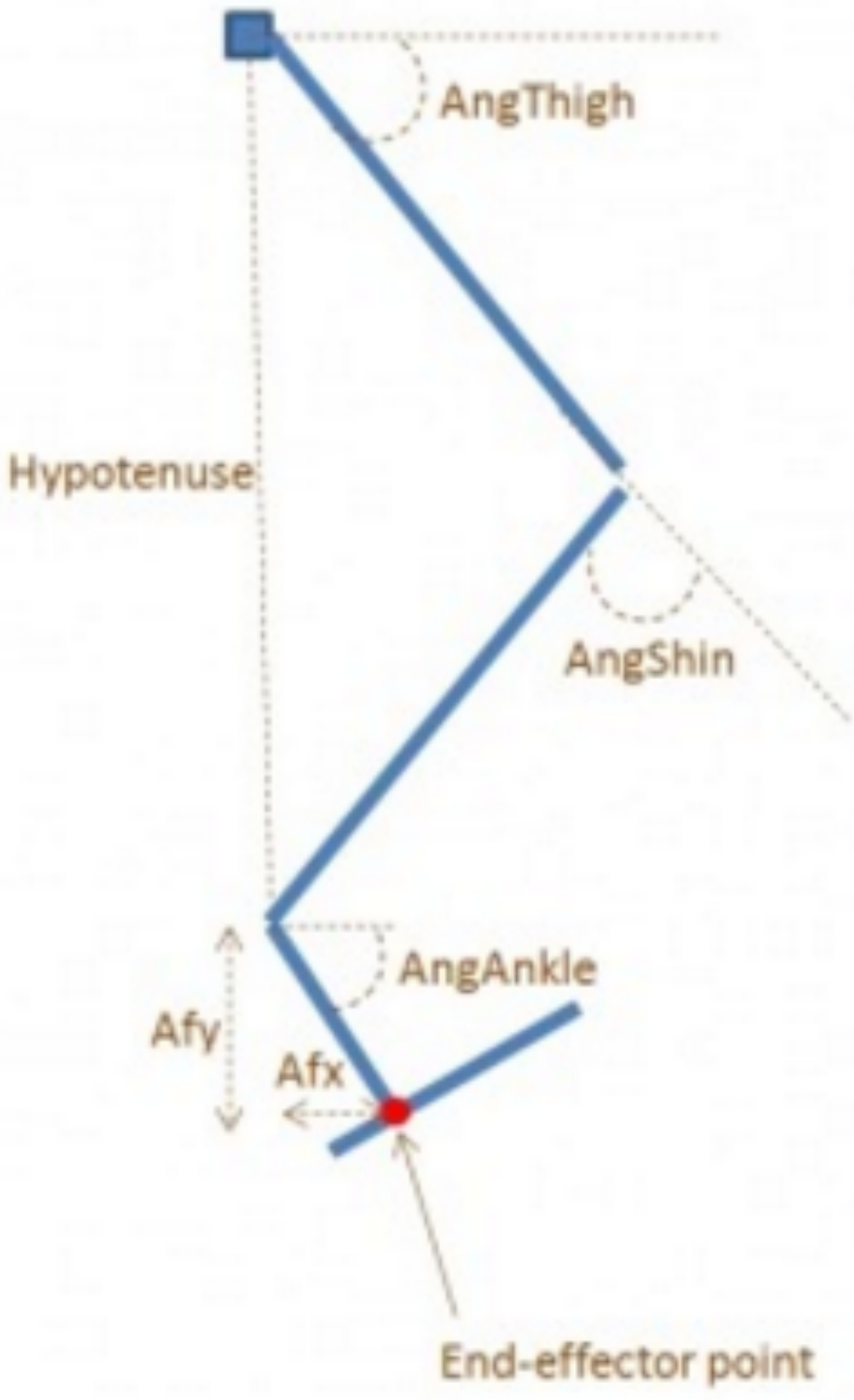


Applications

- Highly efficient
- Widely used
  - Robot
    - Path planning
    - Trajectory Generation







What is an *ideal* line

- Must appear straight and continuous
  - Only possible axis-aligned and 45^o lines
- Must interpolate both defining end points
- Must be efficient, drawn quickly
  - Lots of them are required!!!

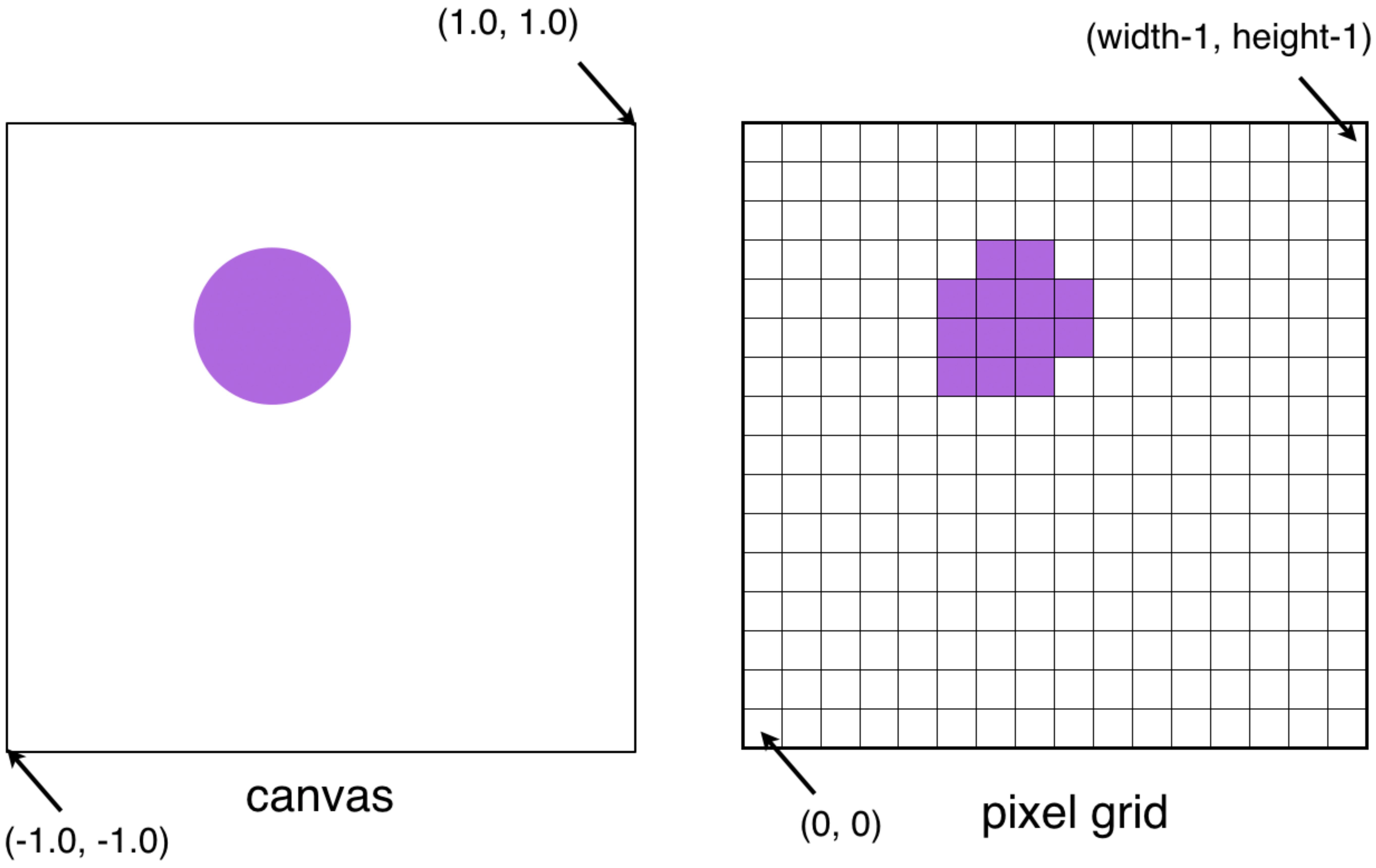
Implicit Geometry Representation

- Define a curve as zero set of 2D implicit function
  - $F(x,y) = 0$ → on curve
  - $F(x,y) < 0$ → inside curve
  - $F(x,y) > 0$ → outside curve
- Example: Circle with center (c_x, c_y) and radius r

$$F(x,y) = (x - c_x)^2 + (y - c_y)^2 - r^2$$

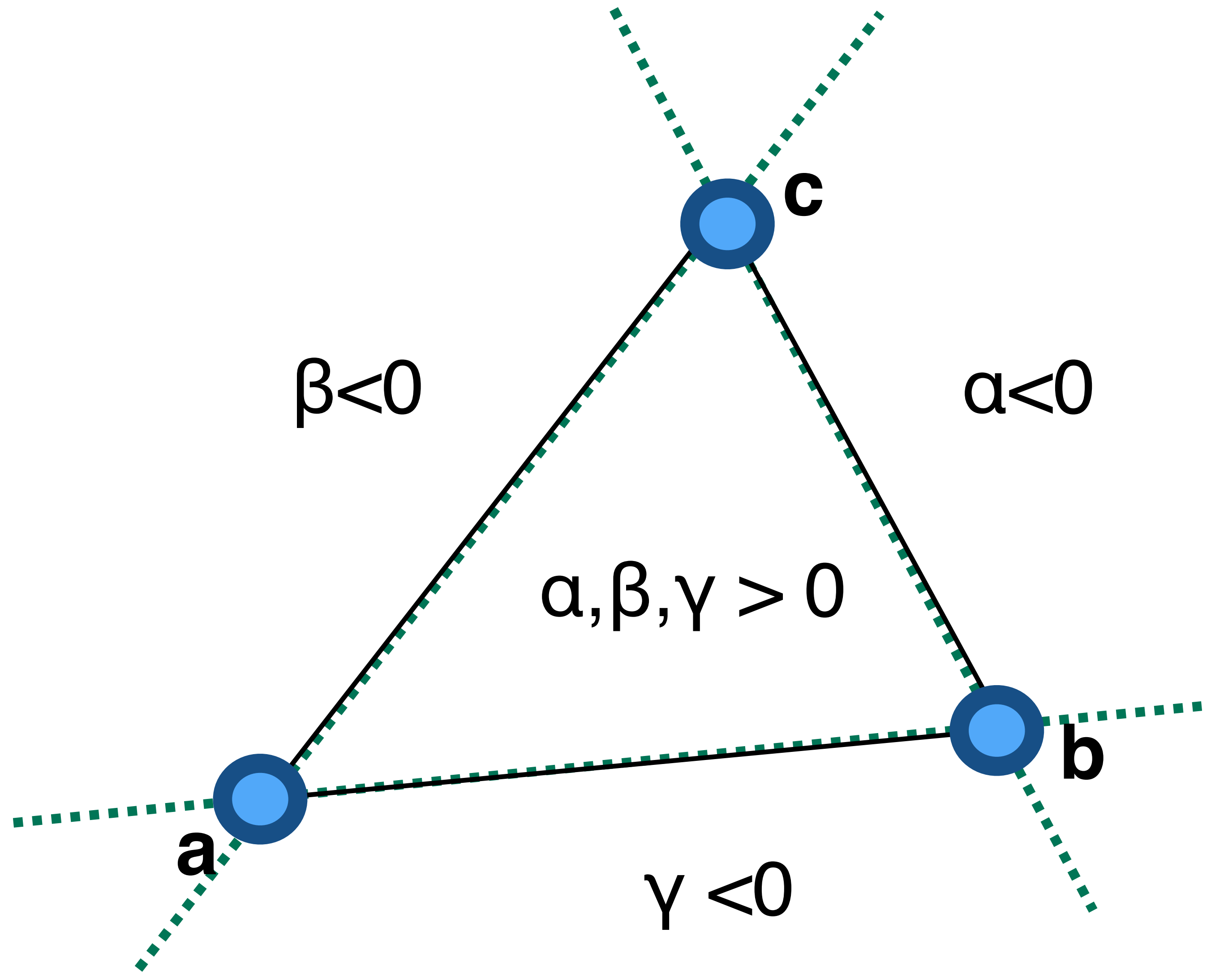
Implicit Rasterization

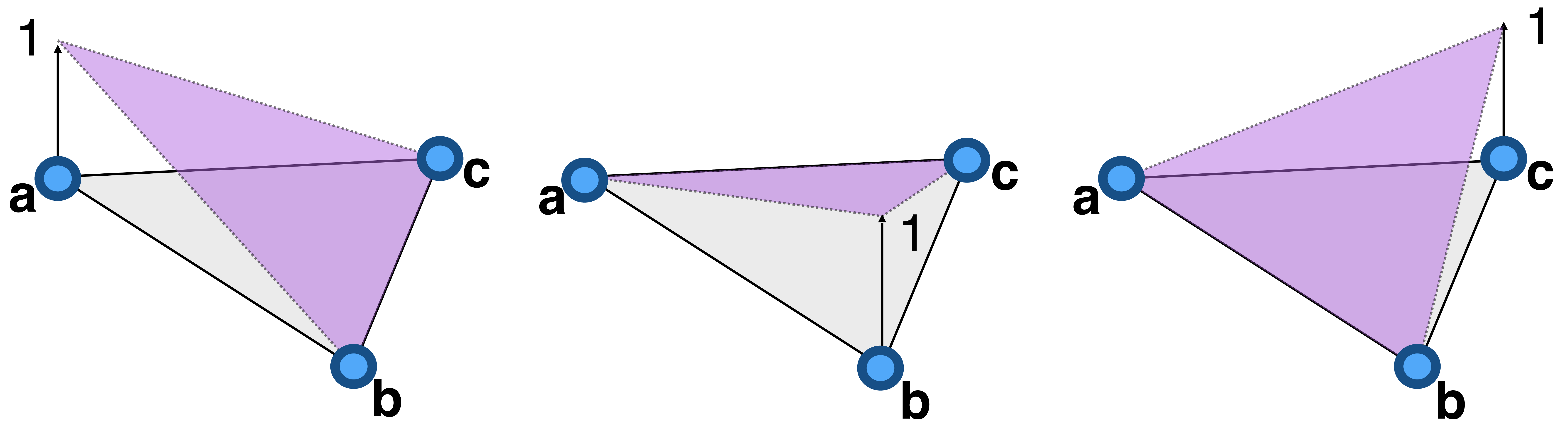
```
for all pixels (i,j)

        (x,y) = map_to_canvas (i,j)

      if F(x,y) < 0

        set_pixel (i,j, color)
```



Barycentric Interpolation

- Barycentric coordinates:
  - $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ with $\alpha + \beta + \gamma = 1$
  - Unique for non-collinear **a**,**b**,**c**
  - Ratio of triangle areas
  - $\alpha(\mathbf{p})$, $\beta(\mathbf{p})$, $\gamma(\mathbf{p})$ are linear functions

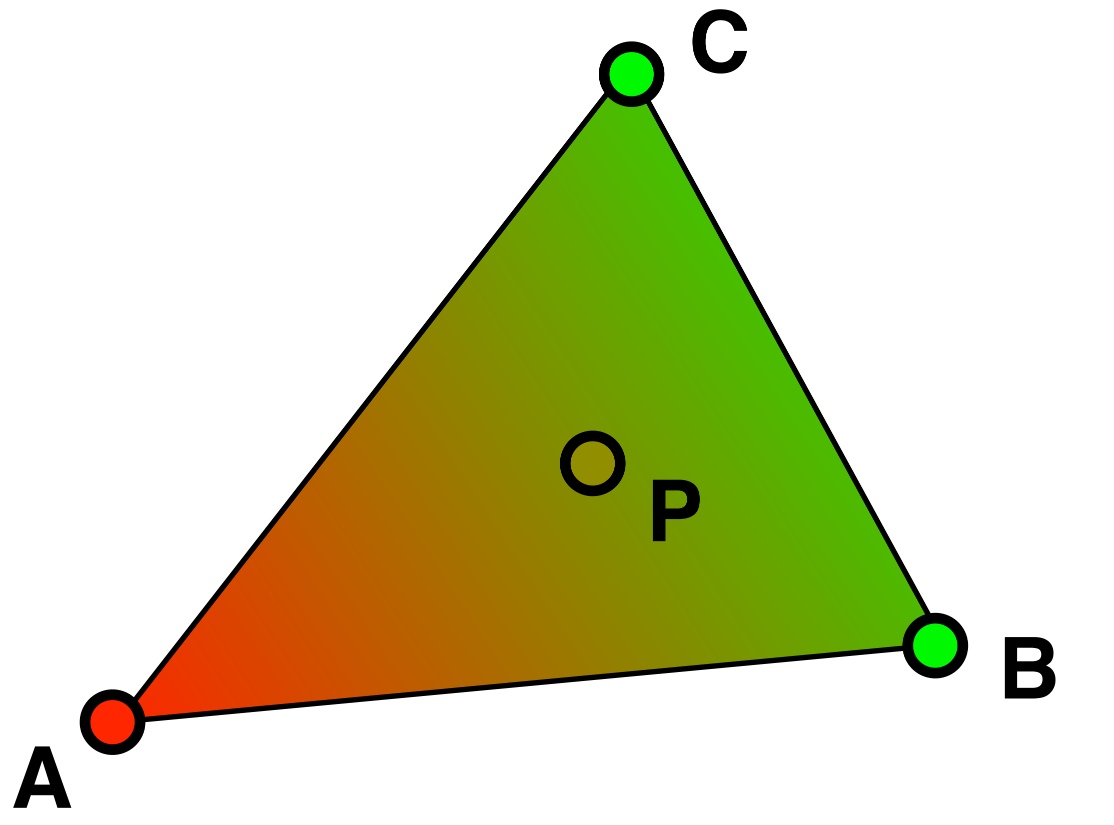




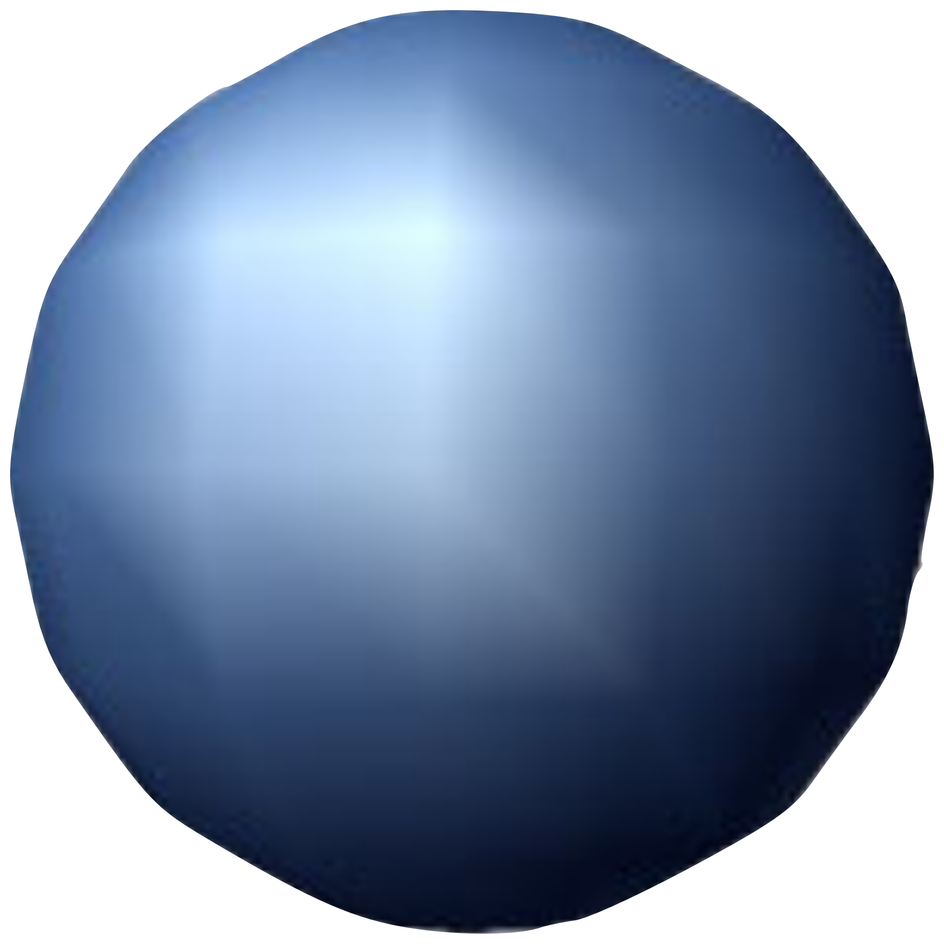
Barycentric Interpolation

- Barycentric coordinates:
  - $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$ with $\alpha + \beta + \gamma = 1$
  - Unique for non-collinear **a**,**b**,**c**
  - Ratio of triangle areas
  - $\alpha(\mathbf{p})$, $\beta(\mathbf{p})$, $\gamma(\mathbf{p})$ are linear functions
  - Gives inside/outside information
  - Use barycentric coordinates to interpolate vertex normals (or other data, e.g. colors)

$$\mathbf{n}(\mathbf{P}) = \alpha \cdot \mathbf{n}(\mathbf{A}) + \beta \cdot \mathbf{n}(\mathbf{B}) + \gamma \cdot \mathbf{n}(\mathbf{C})$$

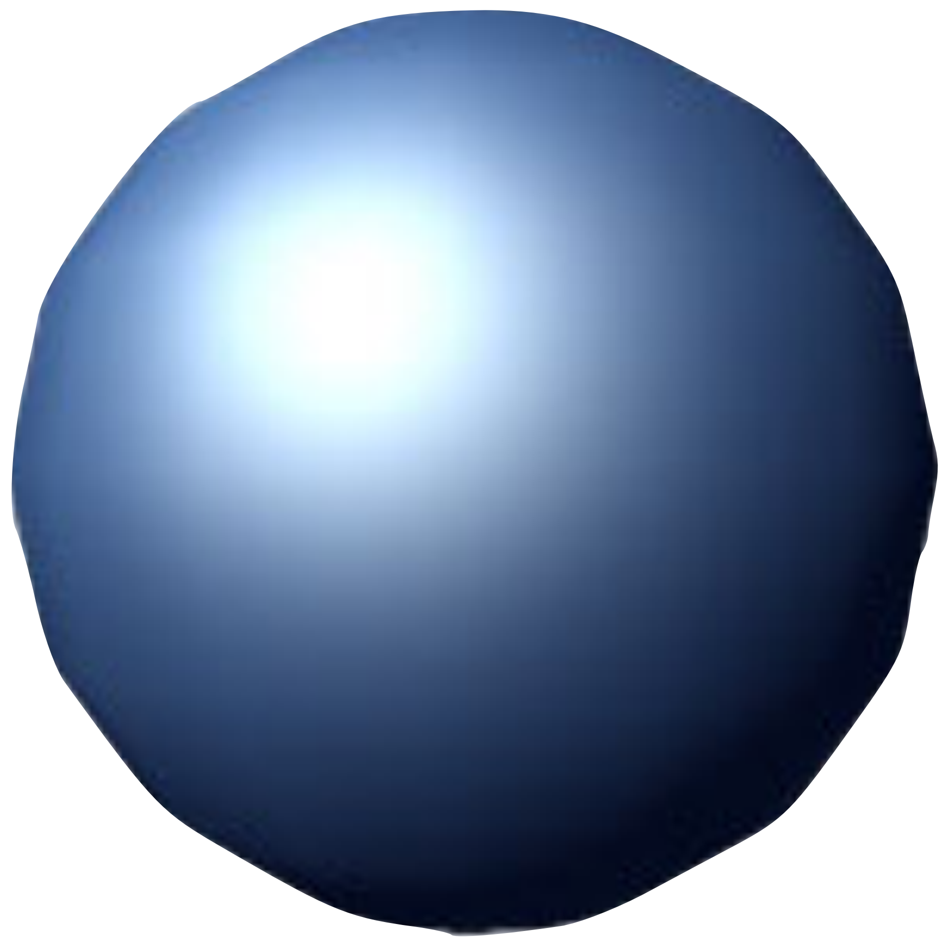


Per-vertex



Evaluate color on vertices, then interpolates it

Per-pixel

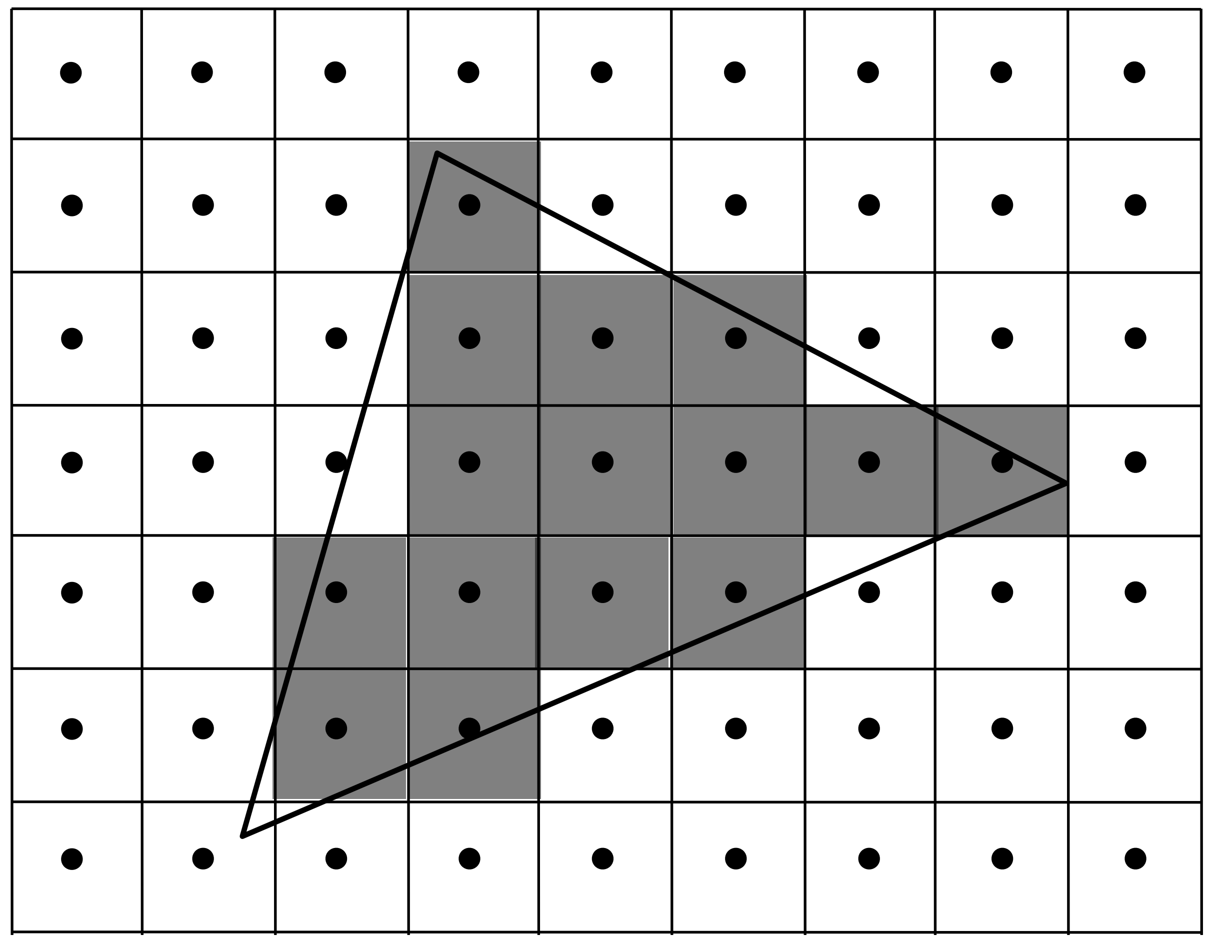


Interpolates positions and normals, then evaluate color on each pixel

Triangle Rasterization

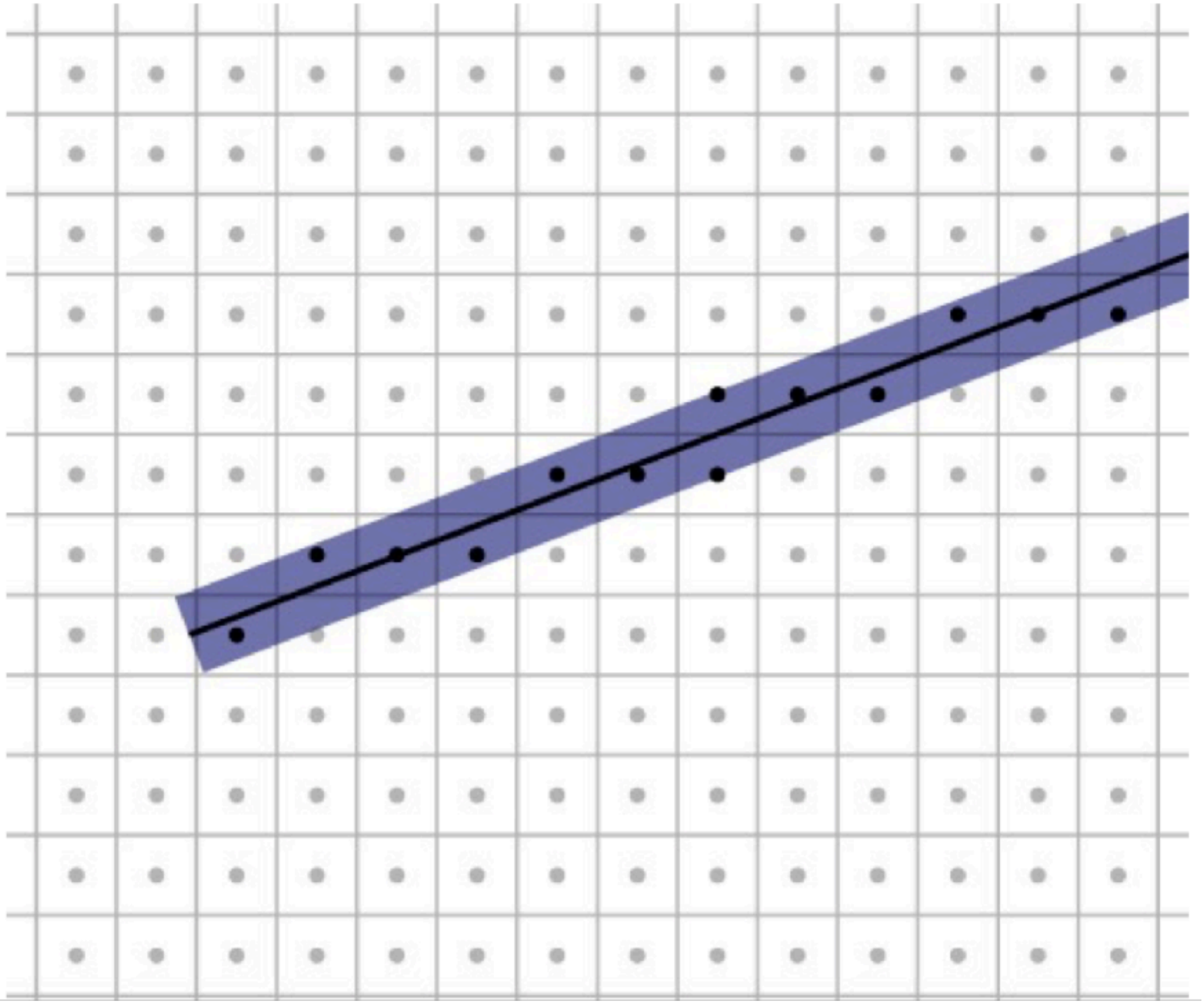
- Each triangle is represented as three 2D points (x_0, y_0), (x_1, y_1), (x_2, y_2)
- Rasterization using barycentric coordinates

for all y **do**
  for all x **do**
    compute (α,β,γ) for (x,y)
    if ($\alpha \in [0,1]$ and $\beta \in [0,1]$ and $\gamma \in [0,1]$
      set_pixel (x,y)



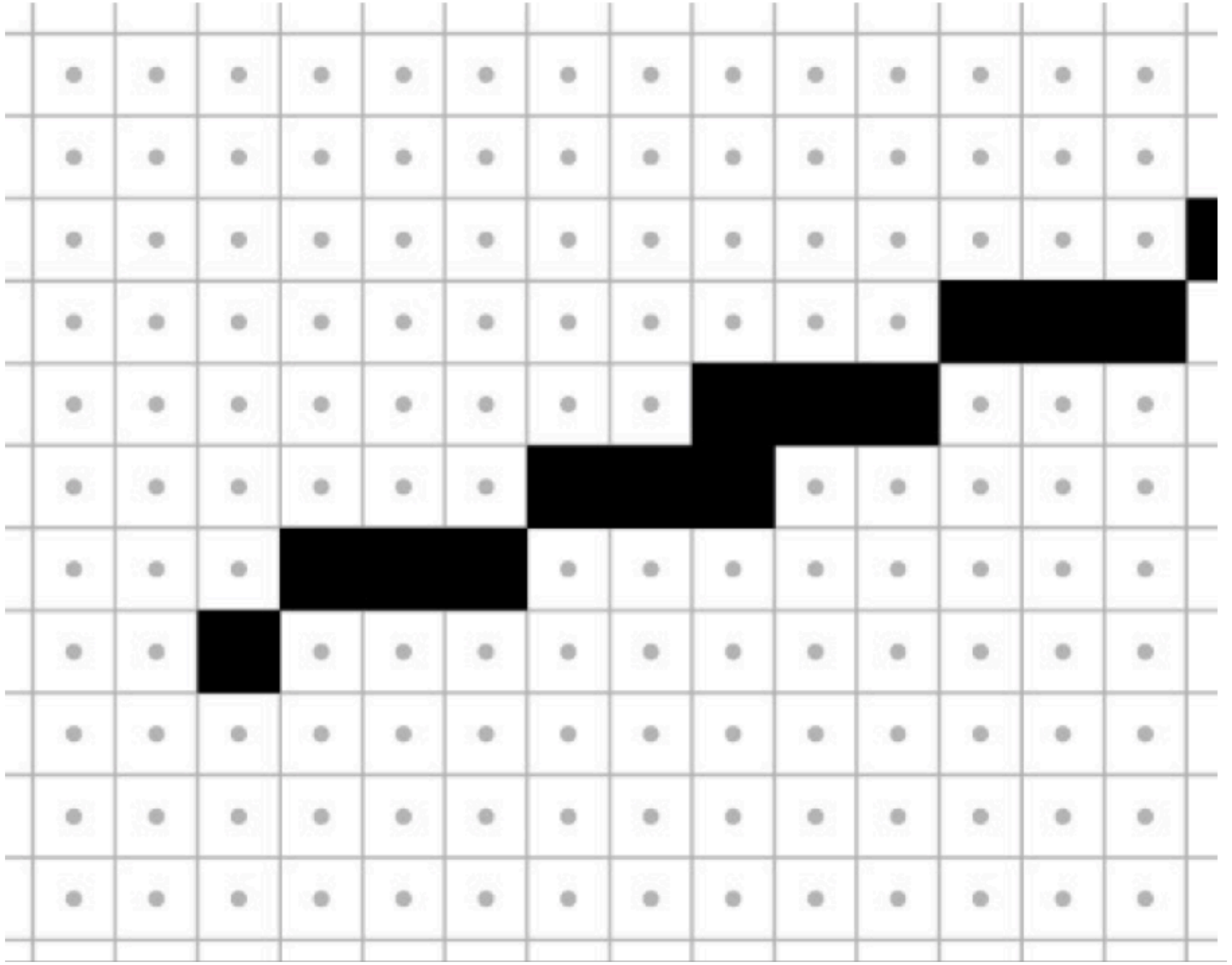
Rasterizing lines

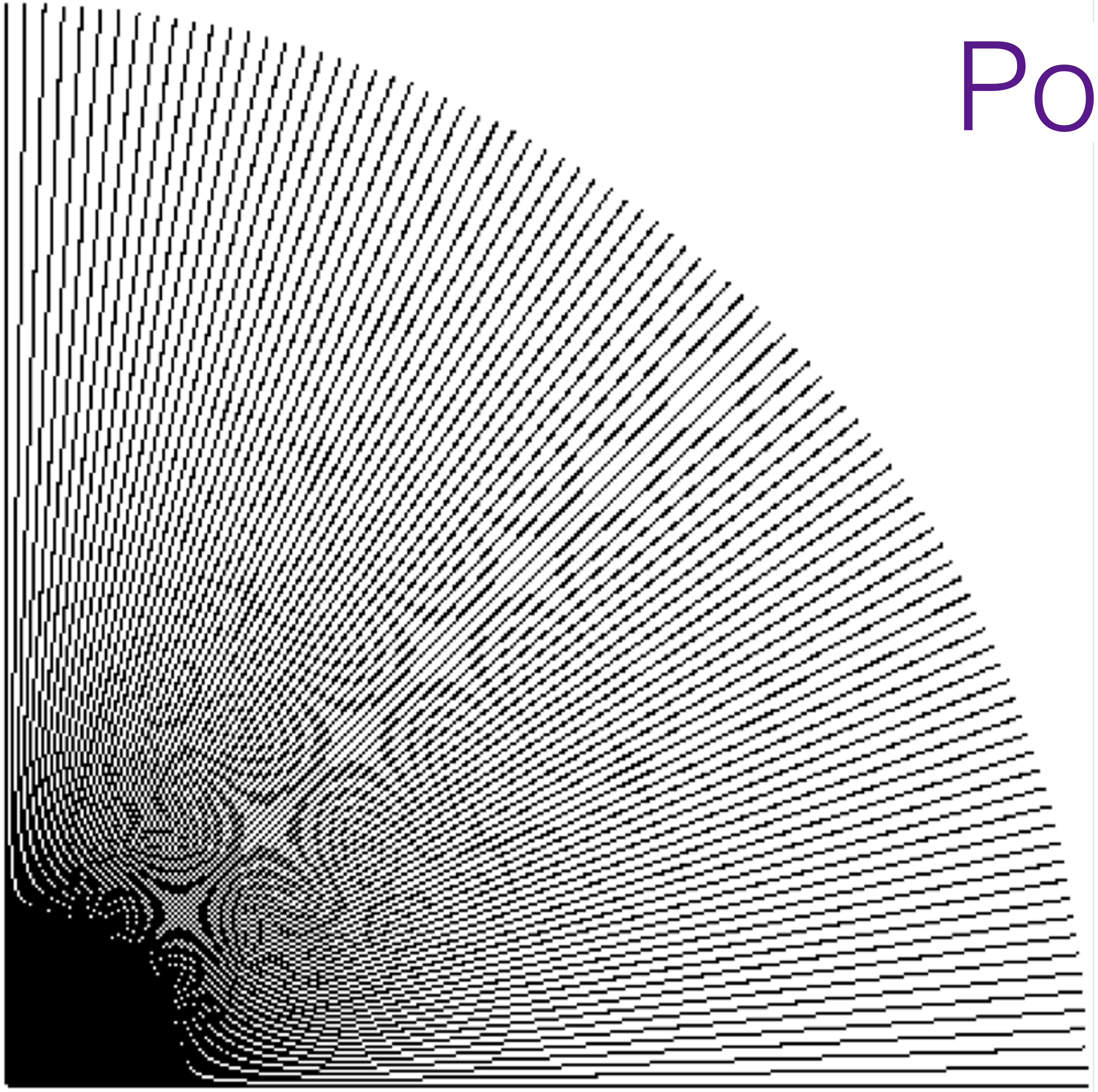
- Define line as a rectangle
- Specify by two endpoints
- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels



Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

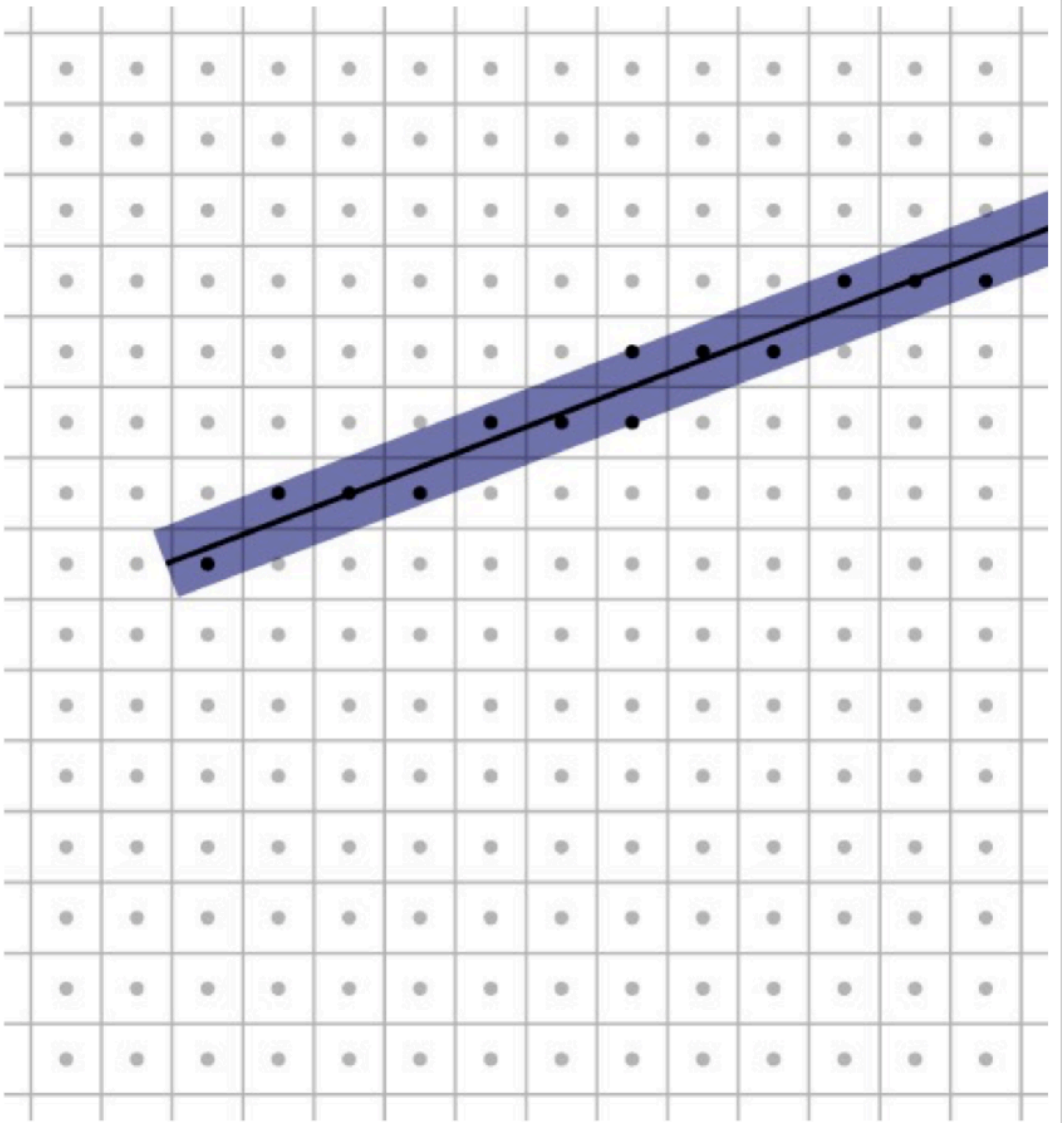




Point sampling in action

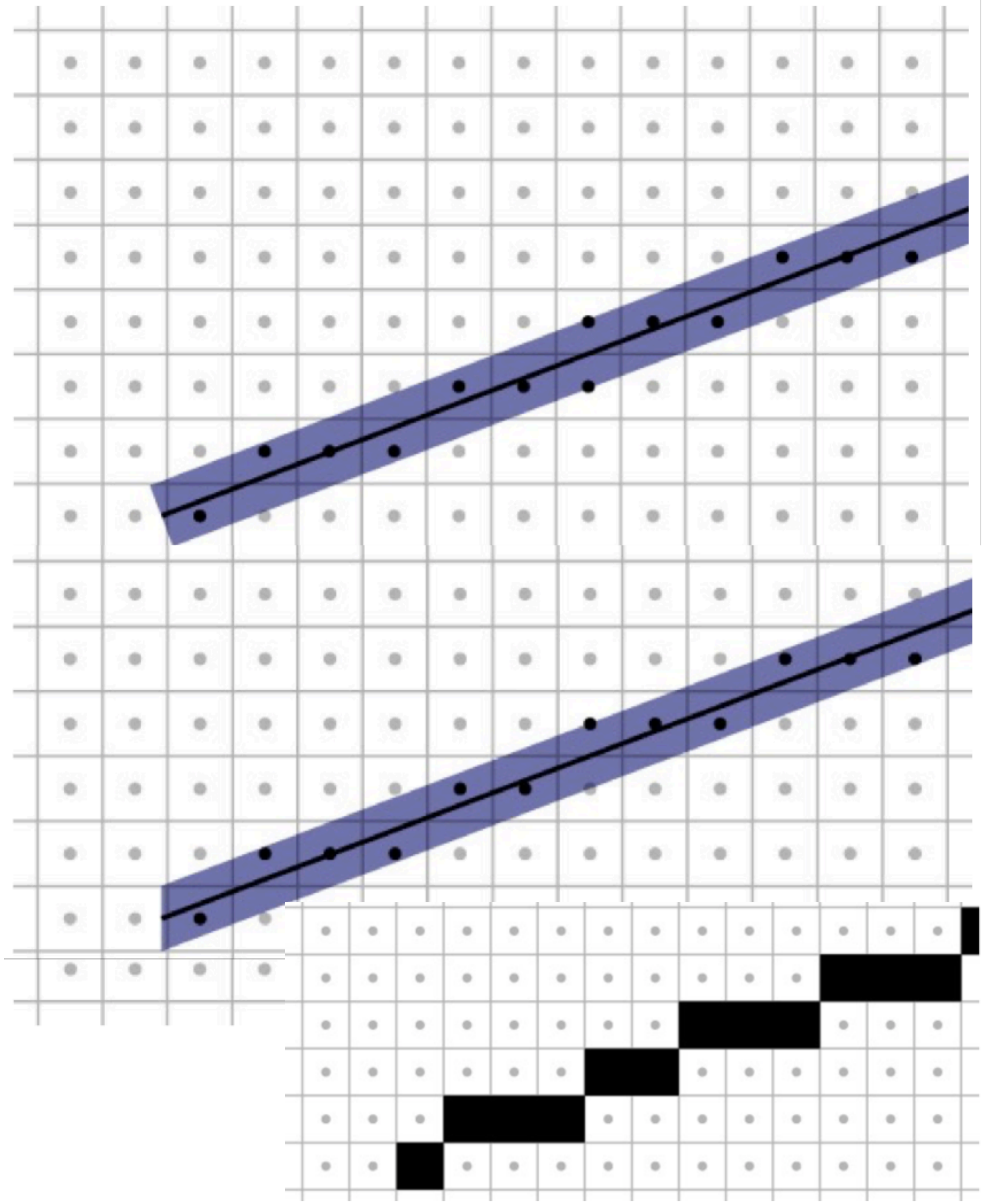
midpoint alg.

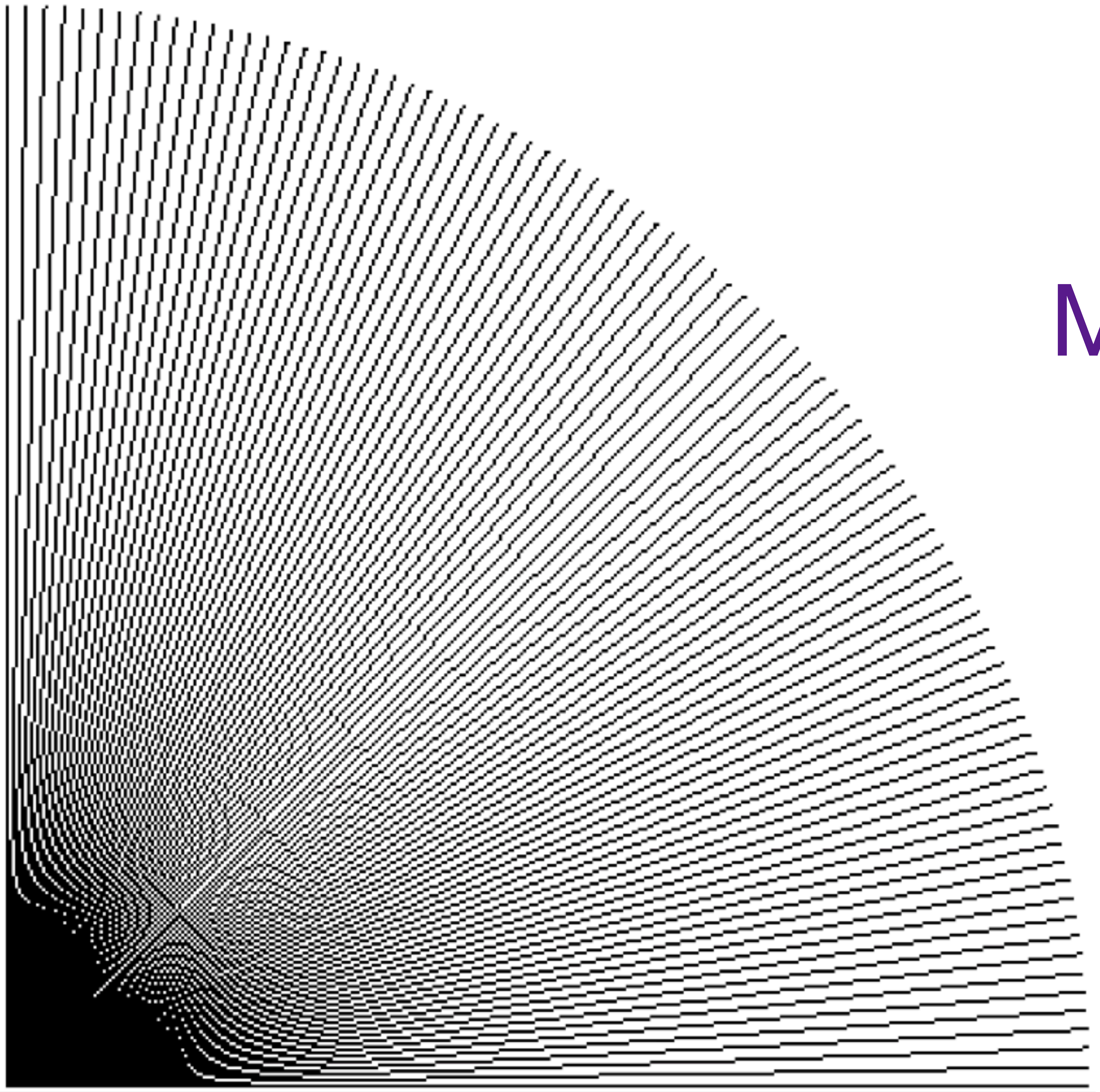
- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45^{o} lines are now thinner



midpoint alg.

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45^{o} lines are now thinner





Midpoint algorithm in action

History

- Bresenham's line algorithm is named after Jack Elton Bresenham who developed it in 1962 at IBM.

- The Calcomp 565 drum plotter, introduced in 1959, was one of the first computer graphics output devices sold.

- Later extended to *Bresenham's circle algorithm* or midpoint circle algorithm.

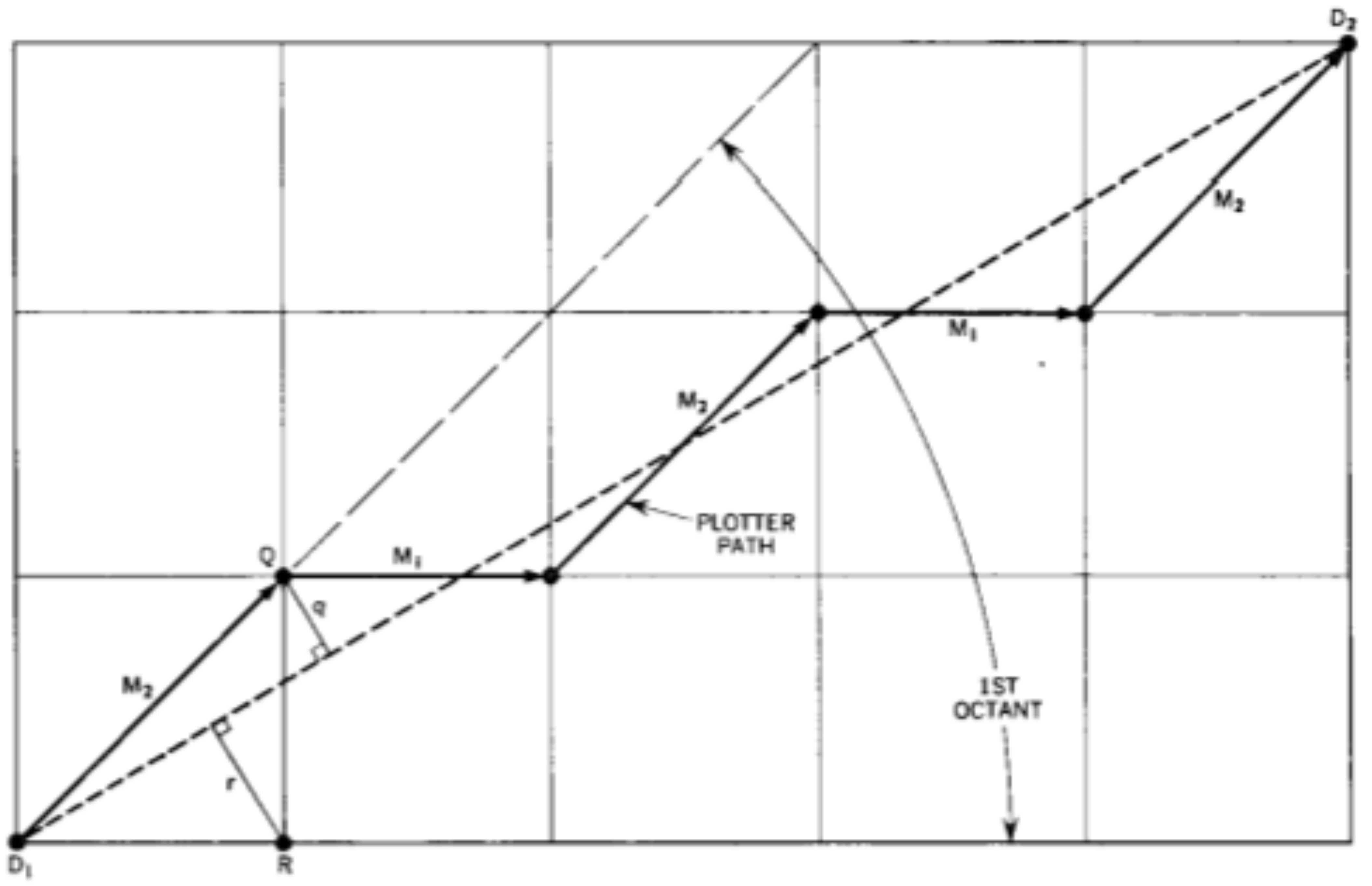
A Calcomp 565 drum plotter

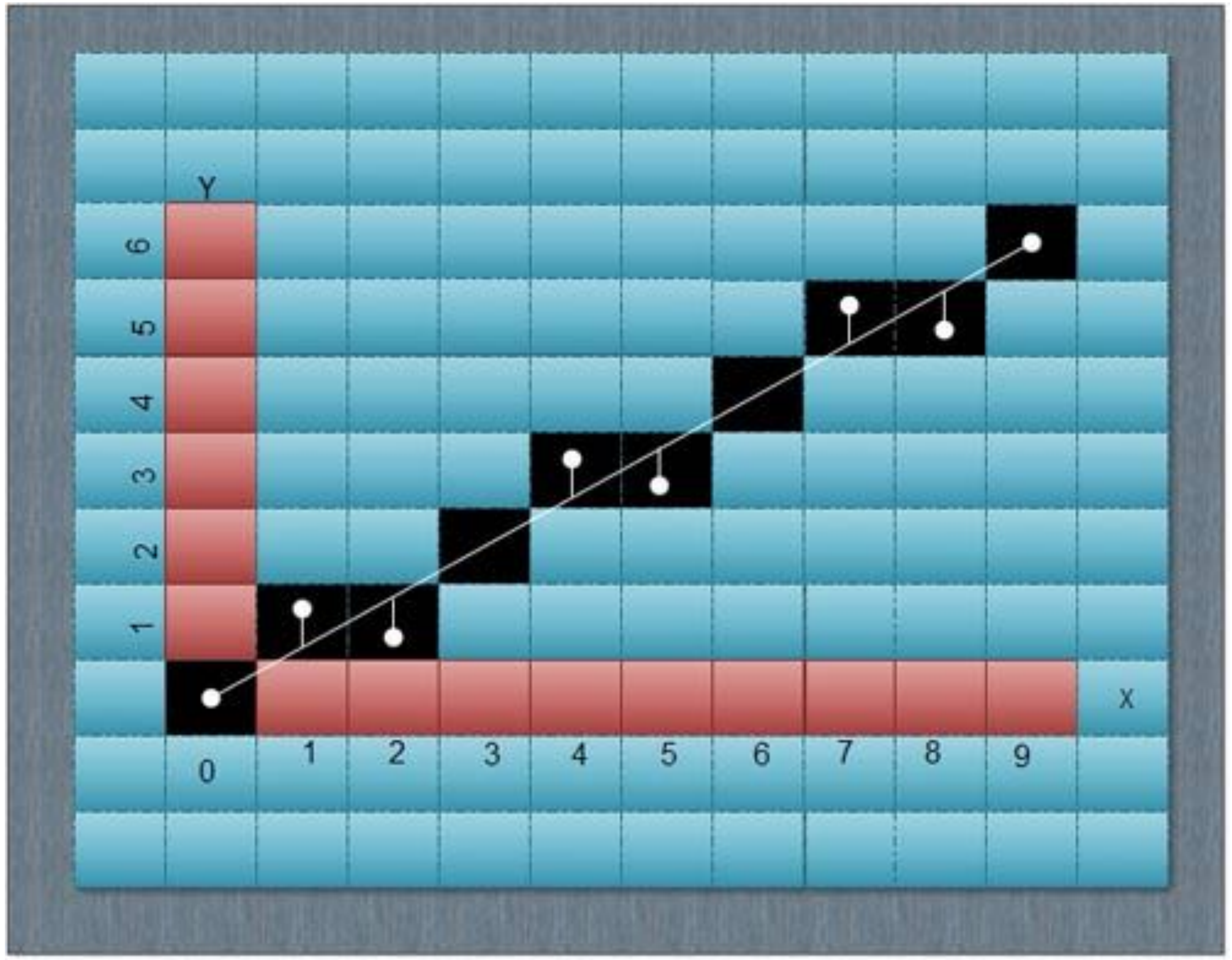
Closeup of Calcomp plotter right side, showing controls for manually moving the drum. Similar controls on the left move the pen carriage.

Algorithm for computer control of a digital plotter

- 1962 by [Jack Elton Bresenham]

Figure 3 Sequence of plotter movements





Comparison of r and q can be implemented by comparing hypotenuse since the two triangles are similar.

Computation of distance of the hypotenuse is simpler, see next page.

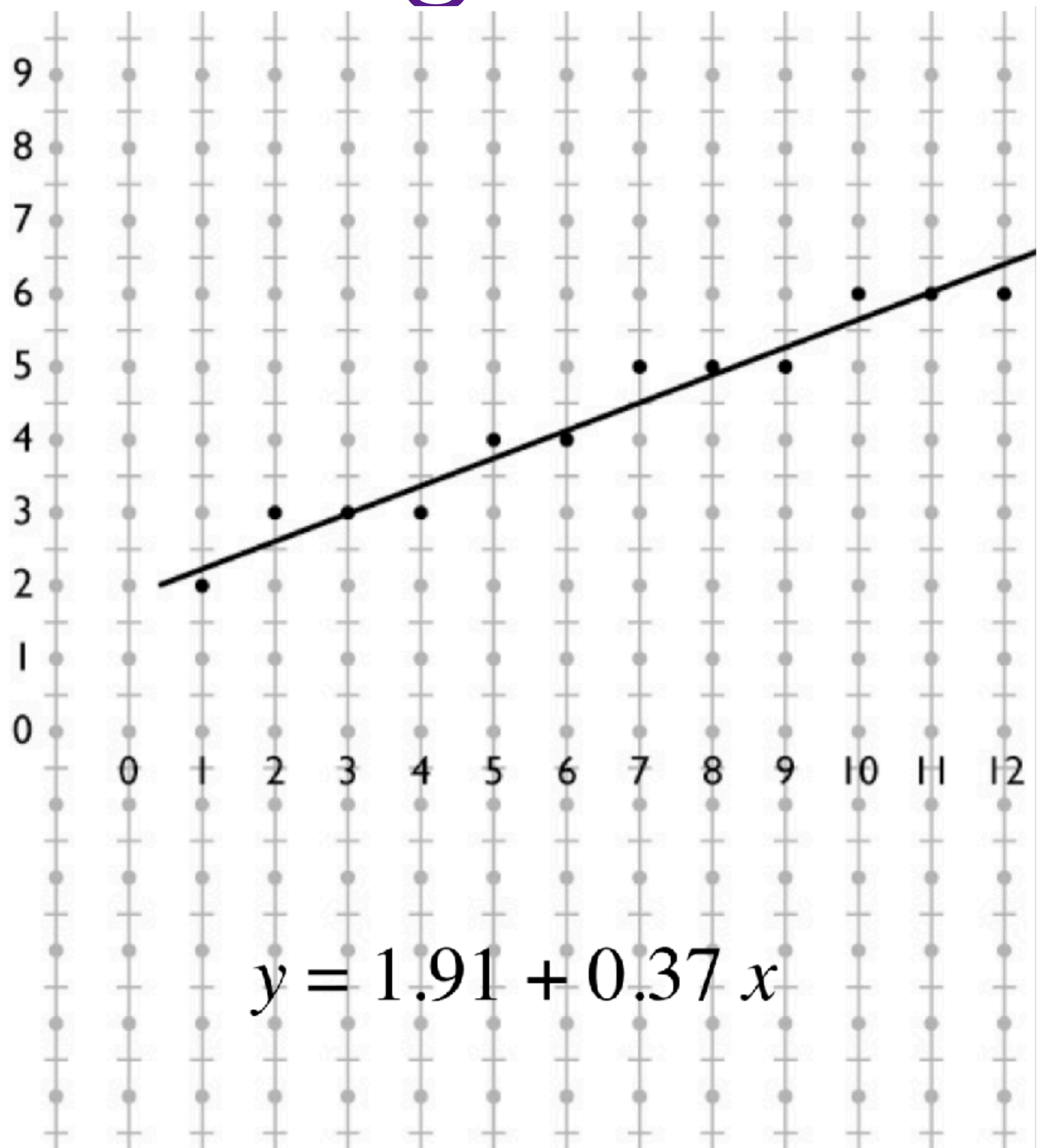
Algorithms for drawing lines

- line equation:
  $y=b+mx$

- Simple algorithm: evaluate line equation per column

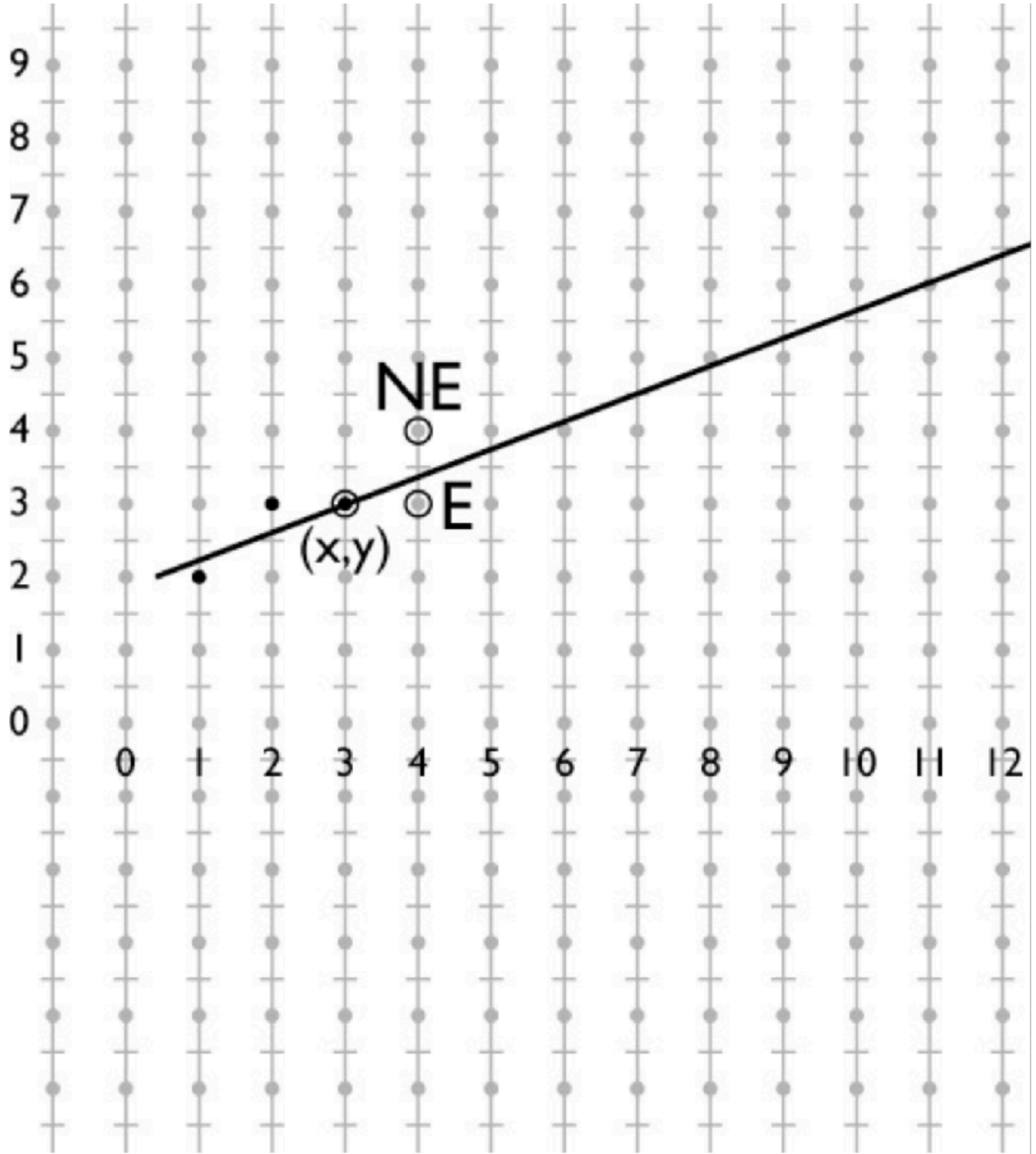
- W.l.o.g. $x0 < x1$;
  $0 \leq m \leq 1$

```
for x = ceil(x0) to floor(x1)
        y = b + m*x
        output(x, round(y))
```



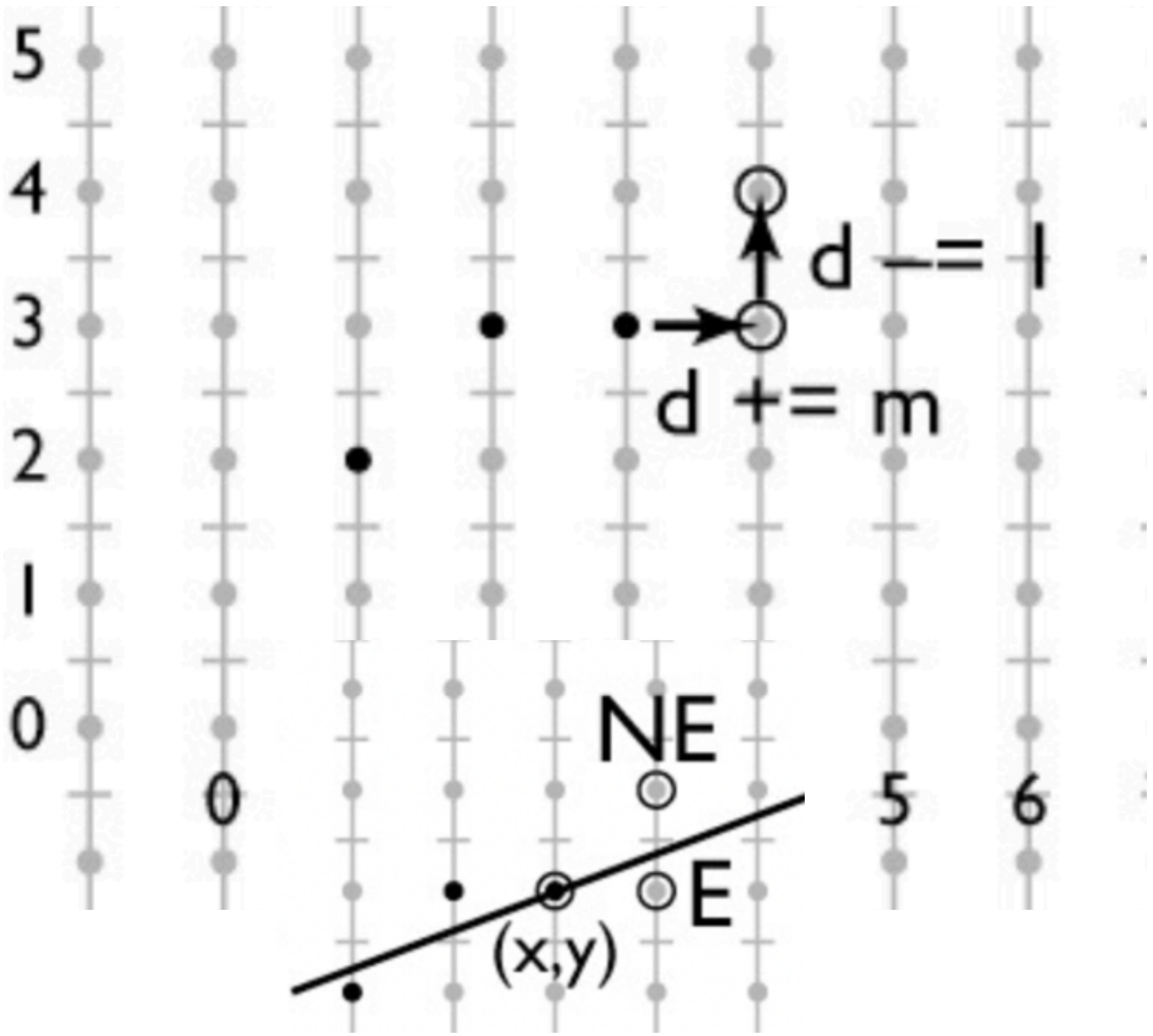
Optimizing line drawing

- Multiplying and rounding is slow
- At each pixel the only options are E and NE
- $d = m(x + 1) + b - y$
- $d > 0.5$ decides between
- E and NE



Optimizing line drawing

- $d = m(x+1) + b - y$
- If d > 0.5
  - y1 = y+1
  - d1 = $m(x+1+1) + b - y\text{-}1$
  - $= d + m - 1$
- $d < 0.5$
  - y1 = y
  - d1 = $m(x+1+1) + b - y$
  - $= d + m$



- Do that with addition
- Known as
  "DDA" (digital differential analyzer)

Mid-Point => Bresenham's line alg.

```
x = ceil(x0)

y = round(m*x + b)

d = m*(x + 1) + b – y

while x < floor(x1)
    if d > 0.5

        y += 1

        d –= 1

    x += 1

    d += m

    output(x, y)
```

- **Still have a "float" operation in calculation of "d"**
- If known 2 endpoints (x0, y0), (xn, yn), draw line **=> Δy=yn-y0, Δx=xn-x0 are integers**
- **Lets create a new decision operator by multiplying 2Δx (recall m=Δy/Δx)**

Bresenham line algorithm

- $d = m(x+1) + b - y$
- If d > 0.5
  - y1 = y+1
  - d1 = $m(x+1+1) + b - y-1$
  - $= d + m - 1$
- $d < 0.5$
  - y1 = y
  - d1 = $m(x+1+1) + b - y$
  - $= d + m$

- $2d\Delta x = 2\Delta y(x+1) + 2\Delta x(b - y)$
- If $2d\Delta x > \Delta x$
  - y1 = y+1
  - $2d1\Delta x = 2\Delta y\,(x+1+1) + 2\Delta x(b - y-1)$
  - $= 2d\Delta x + 2\Delta y - 2\Delta x$
- $d < 0.5$
  - y1 = y
  - $2d1\Delta x = m(x+1+1) + b - y$
  - $= 2d\,\Delta x + 2\Delta y$

Bresenham line algorithm

```
x = ceil(x0)

y = round(m*x + b)

d = m*(x + 1) + b – y

while x < floor(x1)
    if d > 0.5

        y += 1

        d –= 1

    x += 1

    d += m

    output(x, y)
```

```
x = x0

y = y0

p=2Δxd = 2Δy(x0 +1) + 2Δx(b – y0)

while x < xn
    if p > Δx

        y += 1

        p –= 2Δx

    x += 1

    p += 2Δy

    output(x, y)
```

Float?

- $2d\Delta x = 2\Delta y(x+1) + 2\Delta x(b - y)$
- If $2d\Delta x > \Delta x$
  - $y1 = y+1$
  - $2d1\Delta x = 2\Delta y\,(x+1+1) + 2\Delta x(b - y-1)$
  - $= 2d\Delta x + 2\Delta y - 2\Delta x$
- $d < 0.5$
  - $y1 = y$
  - $2d1\Delta x = m(x+1+1) + b - y$
  - $= 2d\,\Delta x + 2\Delta y$

Bresenham line algorithm

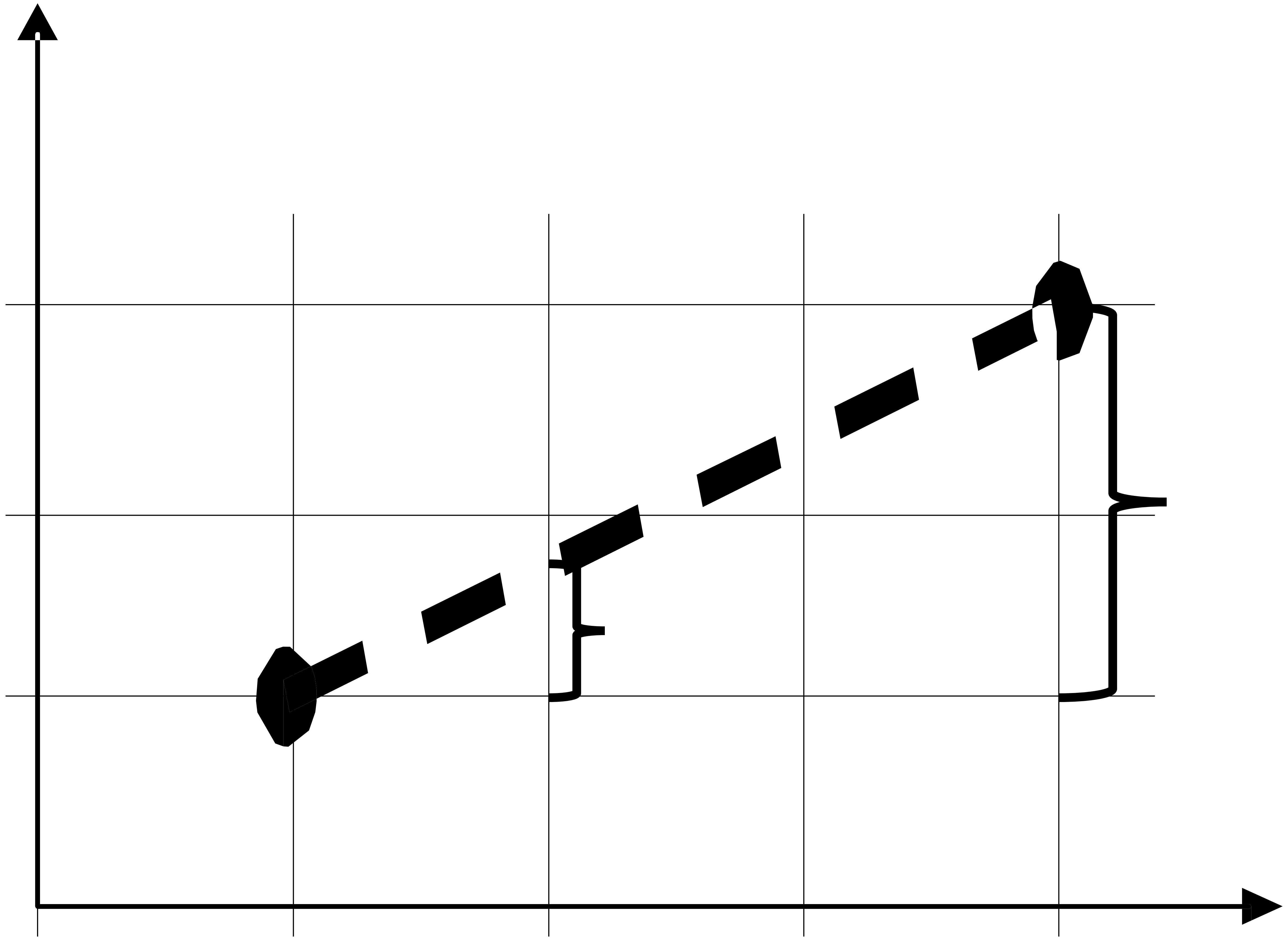
```
x = x0

y = y0

p = 2Δy
while x < xn
    if p > Δx

        y += 1

        p -= 2Δx

    x += 1

    p += 2Δy

    output(x, y)
```

Multiplication?

$$\mathbf{p} = 2\Delta x d = 2\Delta y(x0+1) + 2\Delta x(b - y0)$$

$$= 2\Delta y(x0+1) - 2\Delta x m x0$$

$$= 2\Delta y(x0+1) - 2\Delta y x0$$



Bresenham line algorithm

```
x = x0

y = y0

p = 2Δy
while x < xn
    if p > Δx

        y += 1

        p –= 2Δx

    x += 1

    p += 2Δy

    output(x, y)
```

```
x = x0
; y = y0
;
a= 2Δx; c = 2Δy;
p = c
while x < xn
    if p > Δx

        y += 1

        p –= a

    x += 1

    p += c
    output(x, y)
```

Note -- main loop:

- Only integer math.
- No float representation, or operations needed.
- No multiplication

Bresenham Line Algorithm

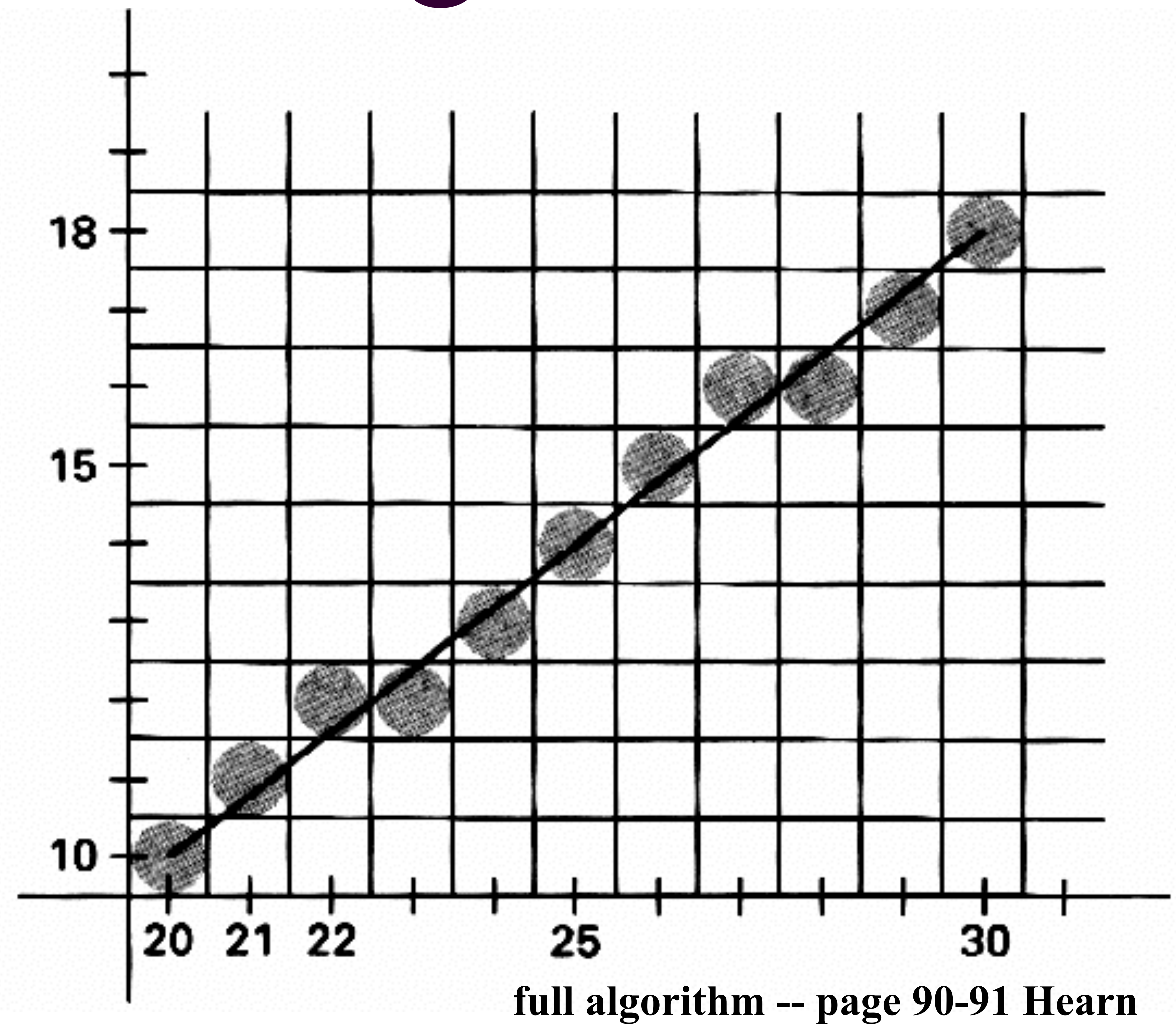
Example:

(20,10) to (30,18)

$\Delta x = 10, \quad \Delta y = 8$

(slope 0.8)

k	P_k	(x_{k+1}, y_{k+1})	k	p_k	(x_{k+1}, y_{k+1})
0	6	(21,11)	5	6	(26,15)
1	2	(22,12)	6	2	(27,16)
2	-2	(23,12)	7	-2	(28,16)
3	14	(24,13)	8	14	(29,17)
4	10	(25,14)	9	10	(30,18)



full algorithm -- page 90-91 Hearn

- **adjusts for slope m>1**
- **re-orders x1,x2,y1,y2 as necessary**

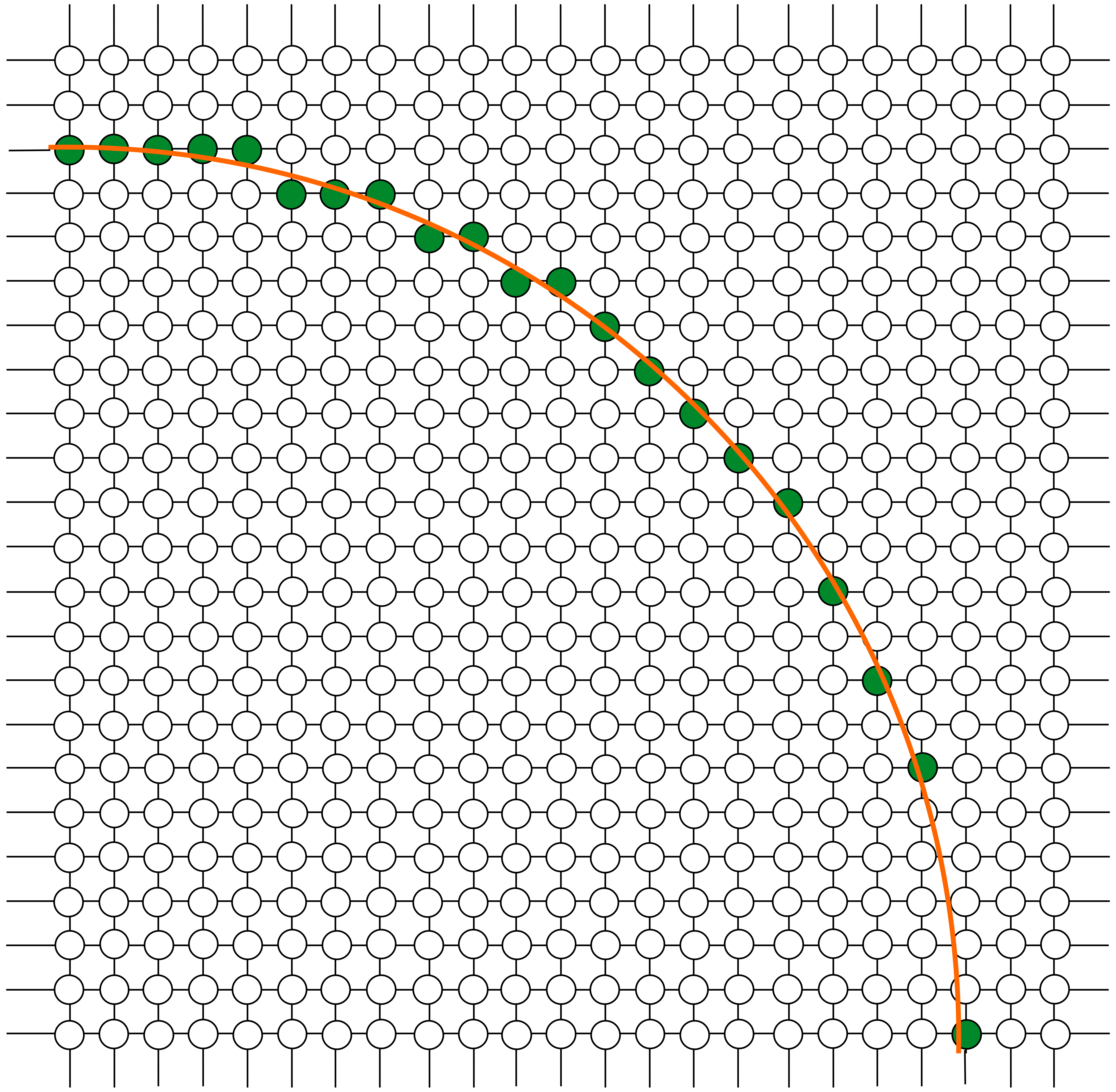
http://www.cosc.canterbury.ac.nz/people/mukundan/cogr/LineMP.html

A Simple Circle Drawing Algorithm

- The equation for a circle is: $x^2 + y^2 = r^2$
- where r is the radius of the circle
- So, we can write a simple circle drawing algorithm by solving the equation for y at unit x intervals using:

$$y = \pm\sqrt{r^2 - x^2}$$

A Simple Circle Drawing Algorithm (cont…)



$$y_0 = \sqrt{20^2 - 0^2} \approx 20$$

$$y_1 = \sqrt{20^2 - 1^2} \approx 20$$

$$y_2 = \sqrt{20^2 - 2^2} \approx 20$$

$$\vdots$$

$$y_{19} = \sqrt{20^2 - 19^2} \approx 6$$

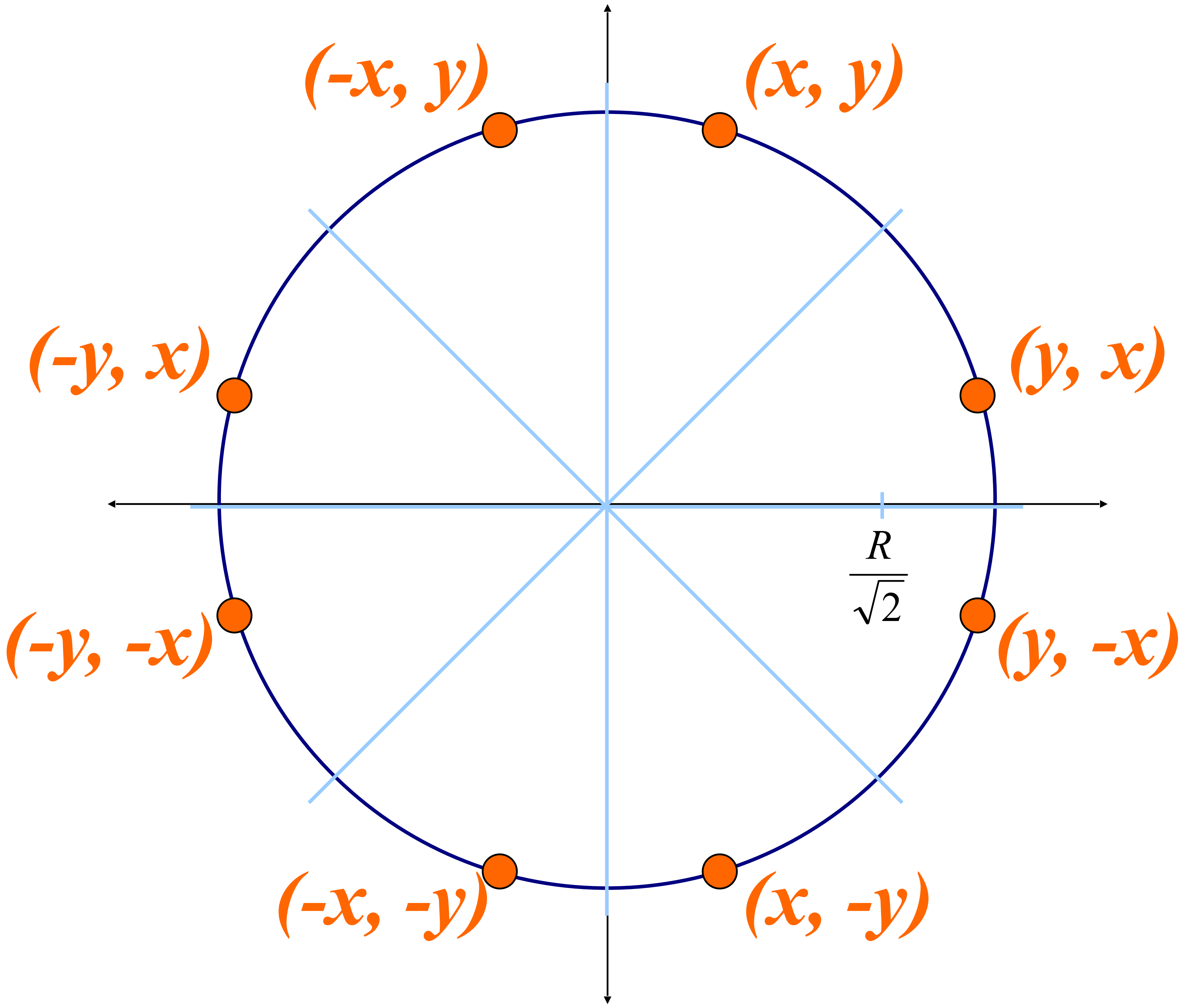
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

A Simple Circle Drawing Algorithm (cont…)

- However, unsurprisingly this is not a brilliant solution!
- Firstly, the resulting circle has large gaps where the slope approaches the vertical
- Secondly, the calculations are not very efficient
  - The square (multiply) operations
  - The square root operation – try really hard to avoid these!
- We need a more efficient, more accurate solution

Eight-Way Symmetry

- The first thing we can notice to make our circle drawing algorithm more efficient is that circles centred at $(0, 0)$ have *eight-way symmetry*



Mid-Point Circle Algorithm

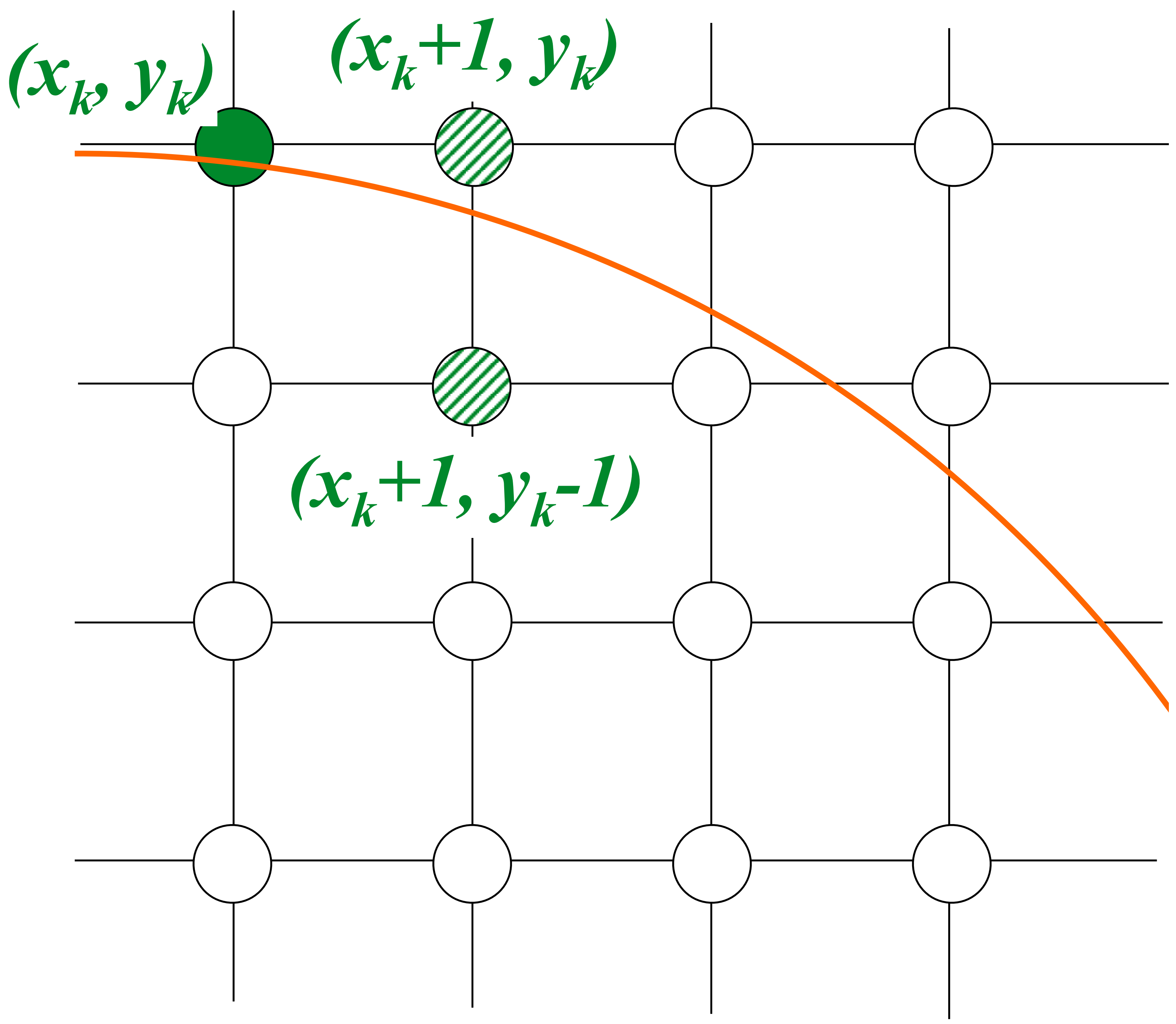
- Similarly to the case with lines, there is an incremental algorithm for drawing circles – the *mid-point circle algorithm*

- In the mid-point circle algorithm we use eight-way symmetry so only ever calculate the points for the top right eighth of a circle, and then use symmetry to get the rest of the points

The mid-point circle algorithm was developed by Jack Bresenham, who we heard about earlier. Bresenham's patent for the algorithm can be viewed here.

Mid-Point Circle Algorithm (cont…)

- Assume that we have just plotted point (x_k, y_k)
- The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)
- We would like to choose the point that is nearest to the actual circle
- So how do we make this choice?



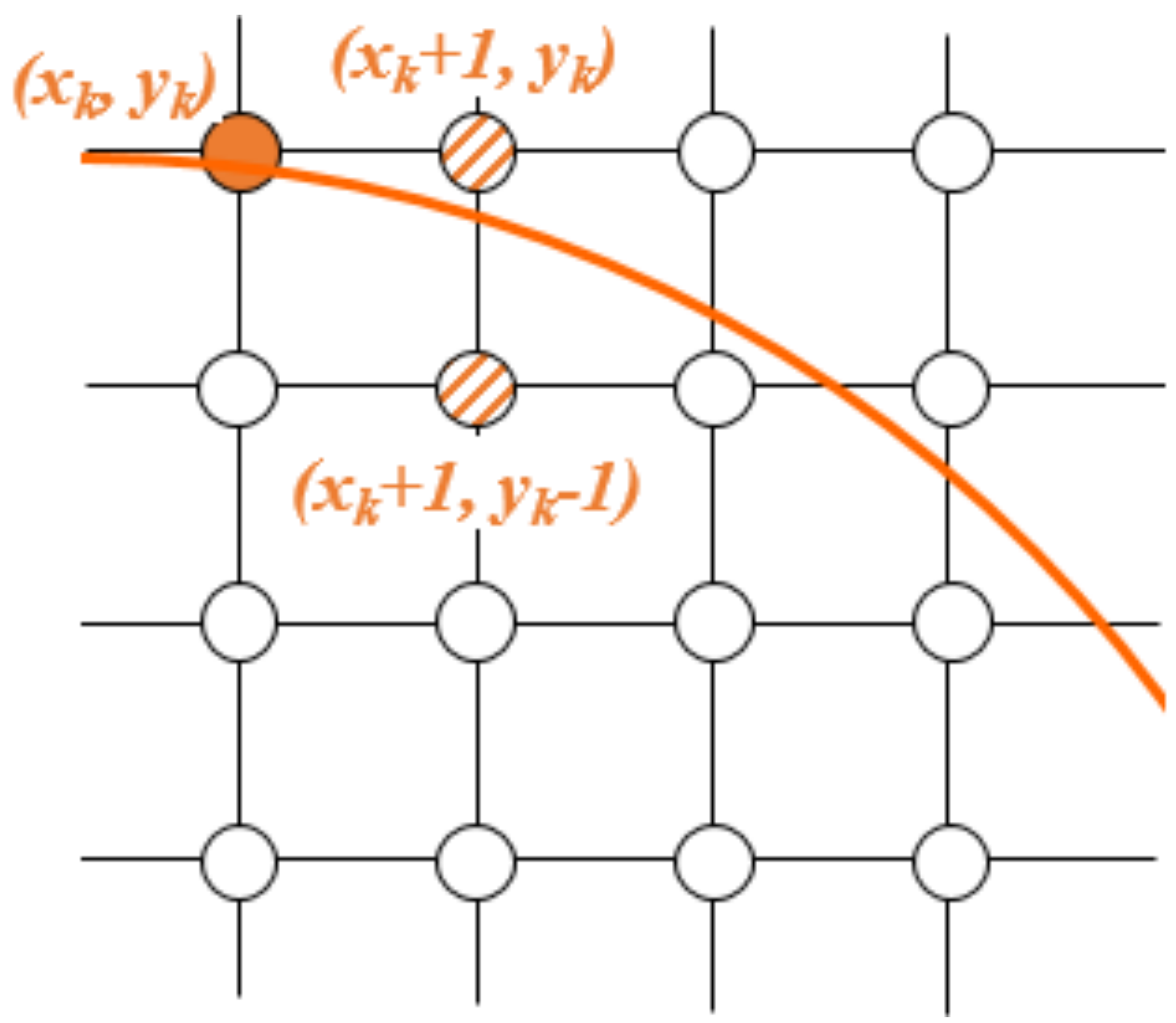
Mid-Point Circle Algorithm (cont…)

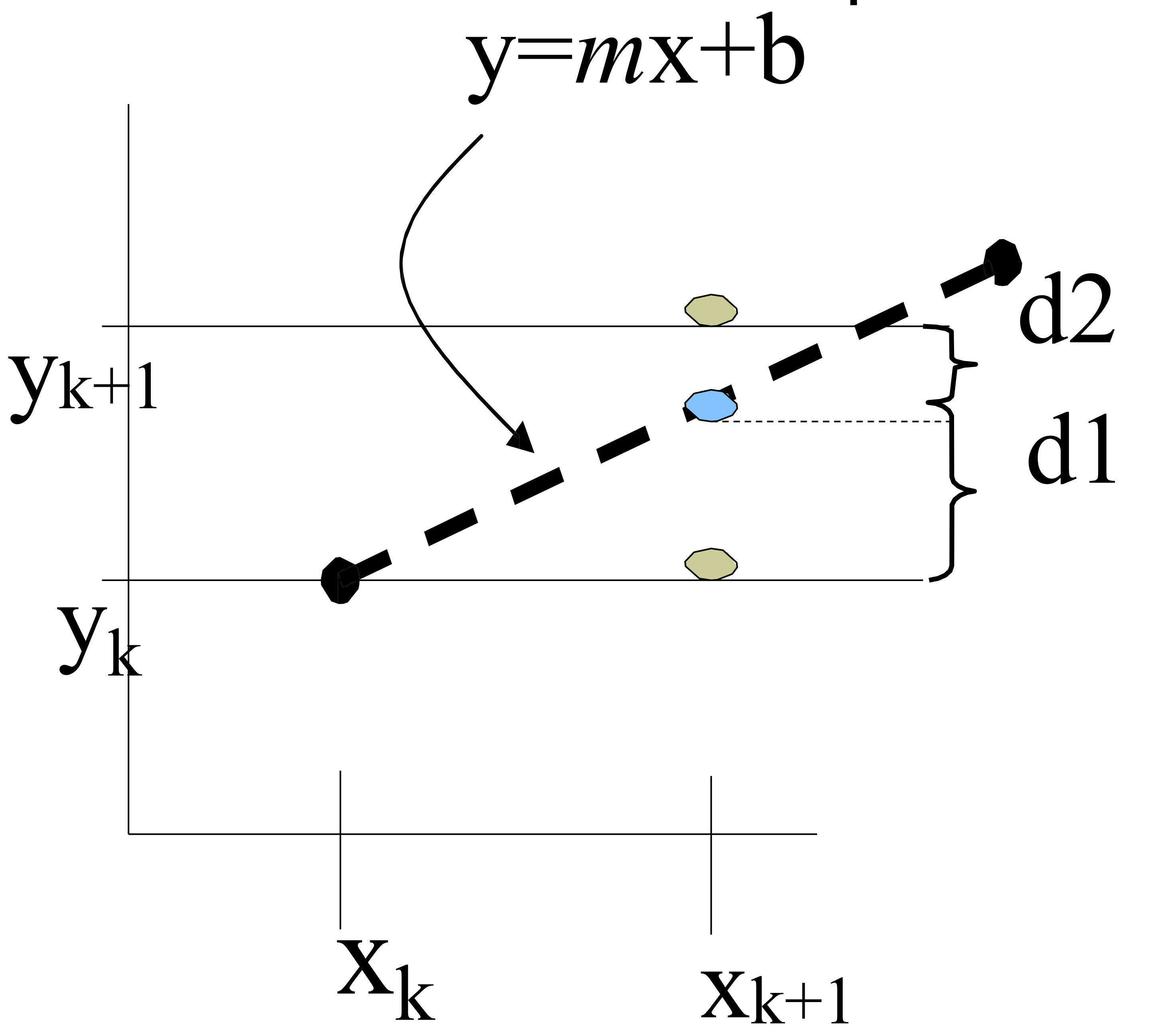
- Let's re-jig the equation of the circle slightly: $f_{circ}(x, y) = x^2 + y^2 - r^2$

- The equation evaluates as follows:

$$f_{circ}(x, y) \begin{cases} < 0, \text{ if } (x, y) \text{ is inside the circle boundary} \\ = 0, \text{ if } (x, y) \text{ is on the circle boundary} \\ > 0, \text{ if } (x, y) \text{ is outside the circle boundary} \end{cases}$$

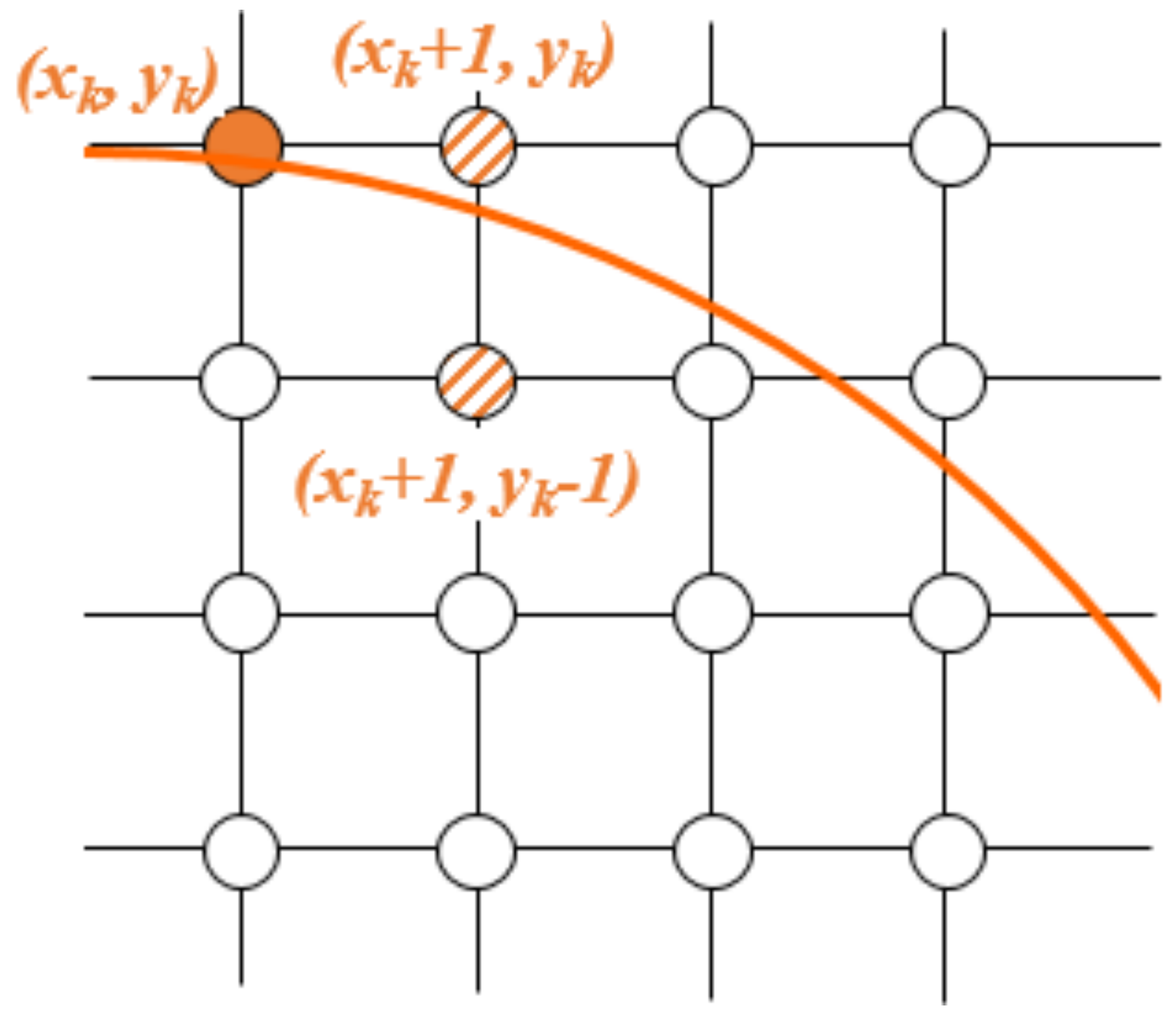
- By evaluating this function at the midpoint between the candidate pixels we can make our decision





Mid-Point Circle Algorithm (cont…)

- Assuming we have just plotted the pixel at (x_k, y_k) so we need to choose between (x_k+1, y_k) and (x_k+1, y_k-1)



- Our decision variable can be defined as:

$$p_k = f_{circ}(x_k + 1, y_k - \tfrac{1}{2})$$
$$= (x_k + 1)^2 + (y_k - \tfrac{1}{2})^2 - r^2$$

- If $p_k < 0$ the midpoint is inside the circle and and the pixel at y_k is closer to the circle

- Otherwise the midpoint is outside and y_k-1 is closer

Mid-Point Circle Algorithm (cont…)

- To ensure things are as efficient as possible we can do all of our calculations incrementally
- First consider:

$$p_{k+1} = f_{circ}\left(x_{k+1}+1, y_{k+1}-\tfrac{1}{2}\right)$$
$$= [(x_k+1)+1]^2 + \left(y_{k+1}-\tfrac{1}{2}\right)^2 - r^2$$

- or: $$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$
- where y_{k+1} is either y_k or y_k-1 depending on the sign of p_k

Mid-Point Circle Algorithm (cont…)

- The first decision variable is given as:

$$p_0 = f_{circ}(1, r - \tfrac{1}{2})$$
$$= 1 + (r - \tfrac{1}{2})^2 - r^2$$
$$= \tfrac{5}{4} - r$$

- $p_k < 0 \Rightarrow y_{k+1} = y_k$:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

- $p_k > 0 \Rightarrow y_{k+1} = y_k - 1$:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_k + 2$$

$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

The Mid-Point Circle Algorithm

1. Input radius r and circle centre (x_c, y_c), then set the coordinates for the first point on the circumference of a circle centred on the origin as:

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as:

$$p_0 = \frac{5}{4} - r$$

3. Starting with $k = 0$ at each position x_k, perform the following test. If $p_k < 0$, the next point along the circle centred on $(0, 0)$ is (x_k+1, y_k) and:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

The Mid-Point Circle Algorithm (cont…)

4. Otherwise the next point along the circle is (x_k+1, y_k-1) and:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

5. Determine symmetry points in the other seven octants

6. Move each calculated pixel position (x, y) onto the circular path centred at (x_c, y_c) to plot the coordinate values:

$$x = x + x_c \qquad y = y + y_c$$

7. Repeat steps 3 to 5 until $x >= y$

Mid-Point Circle Algorithm Summary

- The key insights in the mid-point circle algorithm are:
  - Eight-way symmetry can hugely reduce the work in drawing a circle
  - Moving in unit steps along the x axis at each point along the circle's edge we need to choose between two possible y coordinates

Linear interpolation

- **We often attach attributes to vertices**
  - e.g. computed diffuse color of a hair being drawn using lines
  - want color to vary smoothly along a chain of line segments
- **Recall basic definition**
  - 1D: $f(x) = (1 - \alpha)\, y_0 + \alpha\, y_1$
  - where $\alpha = (x - x_0) / (x_1 - x_0)$
- **In the 2D case of a line segment, alpha is just the fraction of the distance from (x_0, y_0) to (x_1, y_1)**

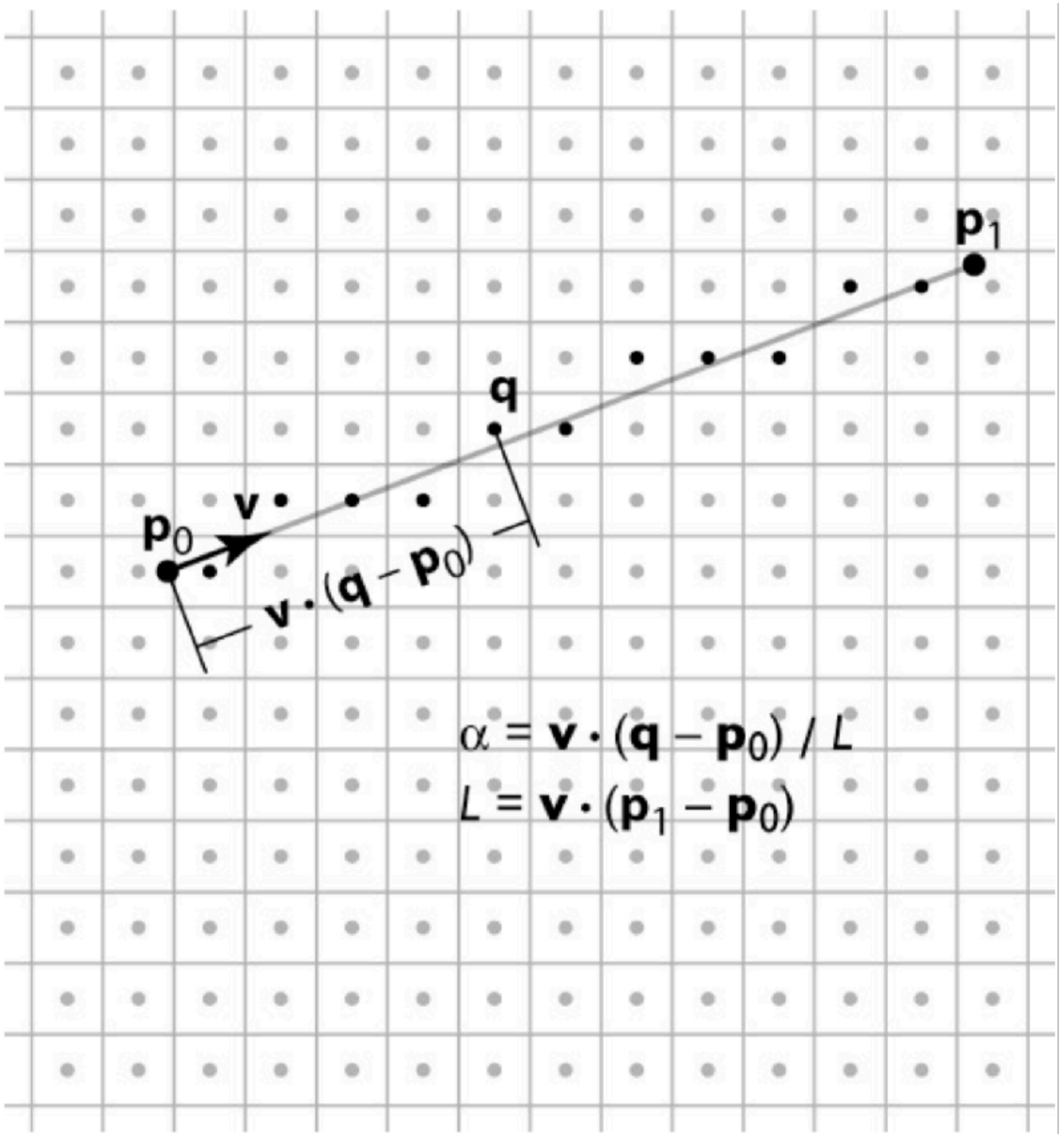
Linear interpolation

- Pixels are not exactly on the line

- Define 2D function by projection on line

– this is linear in 2D

– therefore can use DDA to interpolate

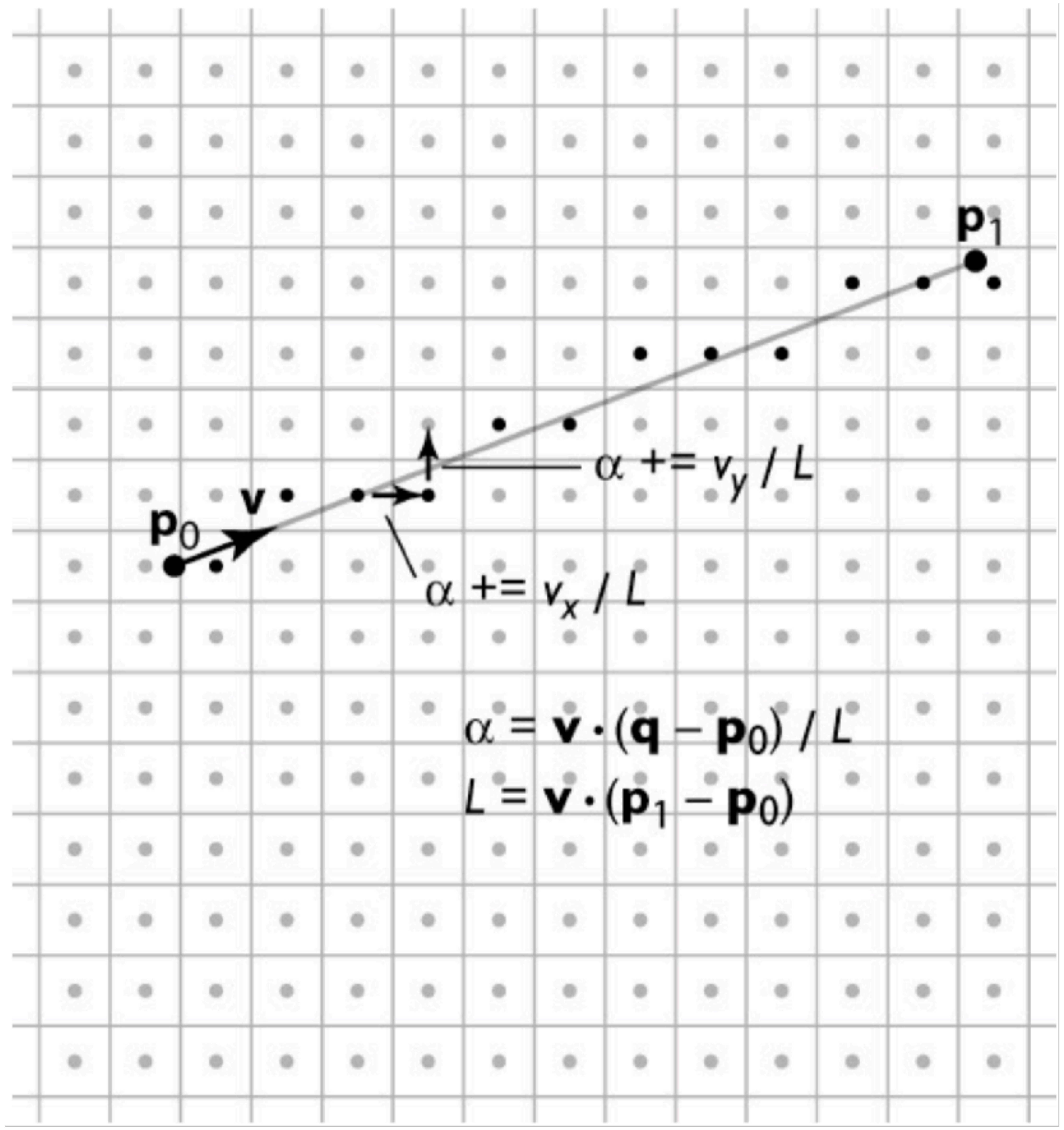


Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line

– this is linear in 2D

– therefore can use DDA to interpolate

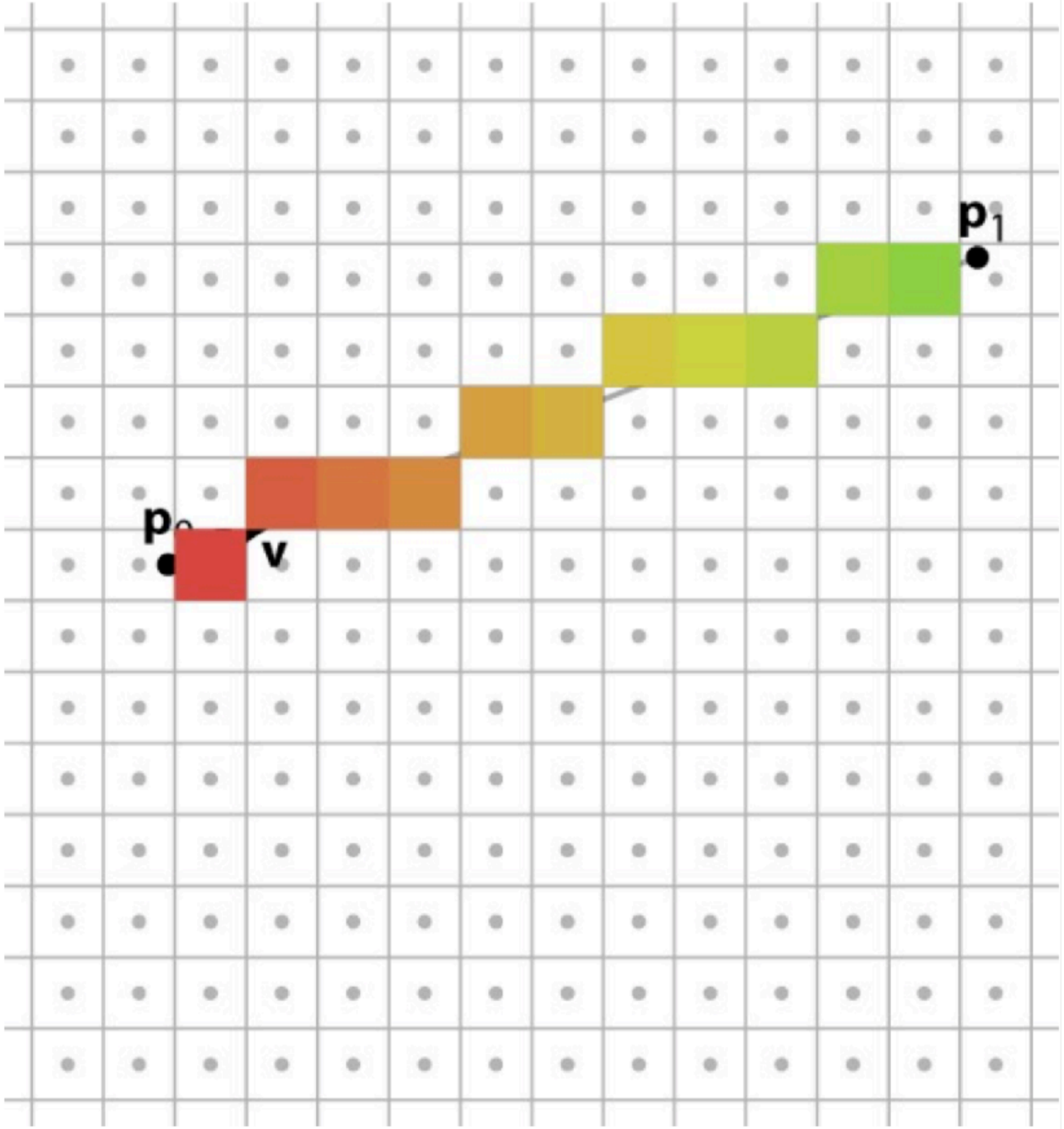


Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line

– this is linear in 2D

– therefore can use DDA to interpolate



Alternate interpretation

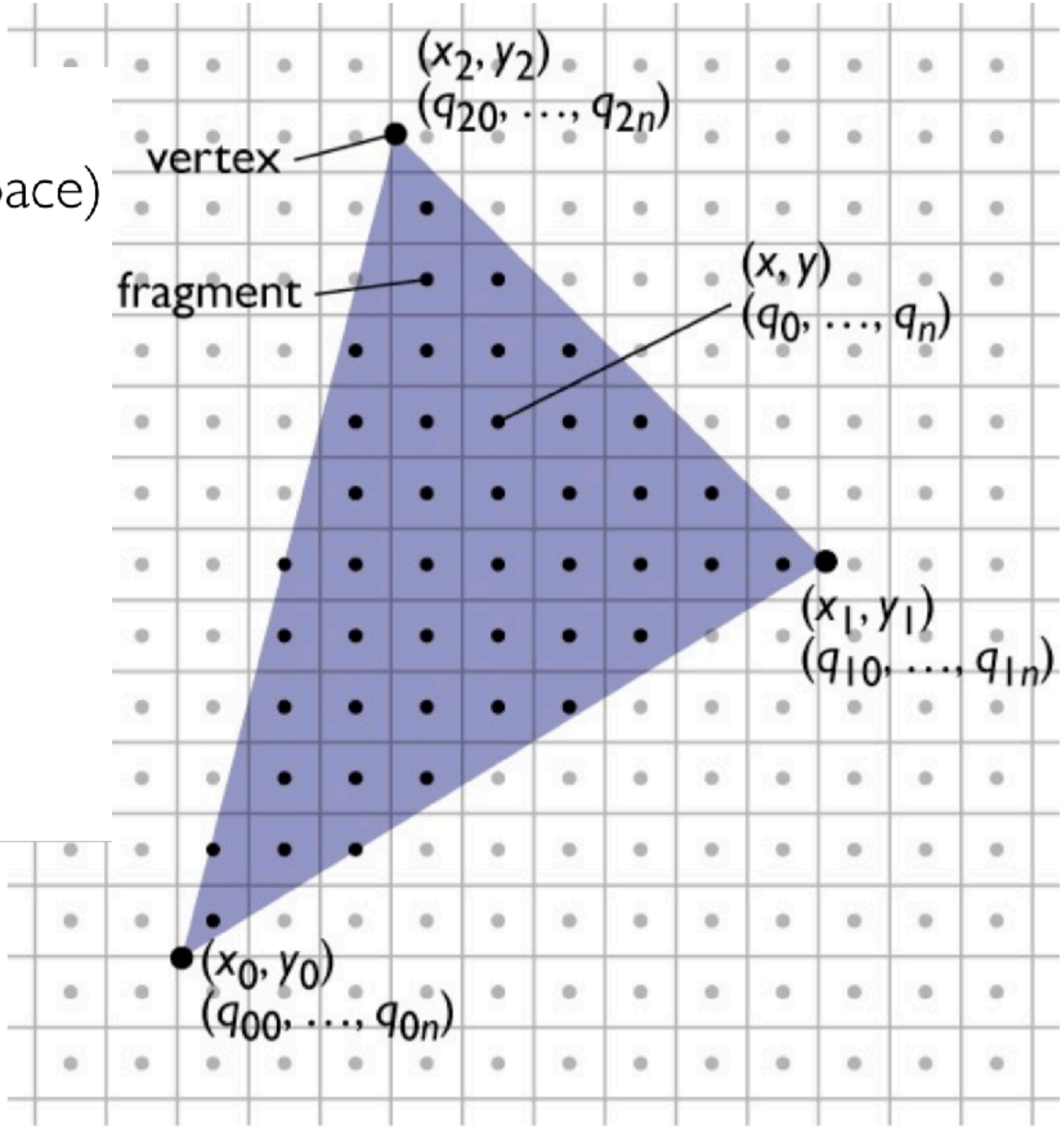
- We are updating d and α as we step from pixel to pixel
  - d tells us how far from the line we are
  - α tells us how far along the line we are
- So d and α are coordinates in a coordinate system oriented to the line

Rasterizing triangles

- The most common case in most applications
  - with good antialiasing can be the only case
  - some systems render a line as two skinny triangles
- Triangle represented by three vertices
- Simple way to think of algorithm follows the pixel-walk interpretation of line rasterization
  - walk from pixel to pixel over (at least) the polygon's area
  - evaluate linear functions as you go
  - use those functions to decide which pixels are inside

Rasterizing triangles

- **Input:**
  - three 2D points (the triangle's vertices in pixel space)
    - $(x_0, y_0); (x_1, y_1); (x_2, y_2)$
  - parameter values at each vertex
    - $q_{00}, \ldots, q_{0n}; q_{10}, \ldots, q_{1n}; q_{20}, \ldots, q_{2n}$
- **Output: a list of fragments, each with**
  - the integer pixel coordinates (x, y)
  - interpolated parameter values $q_0, \ldots, q_n$



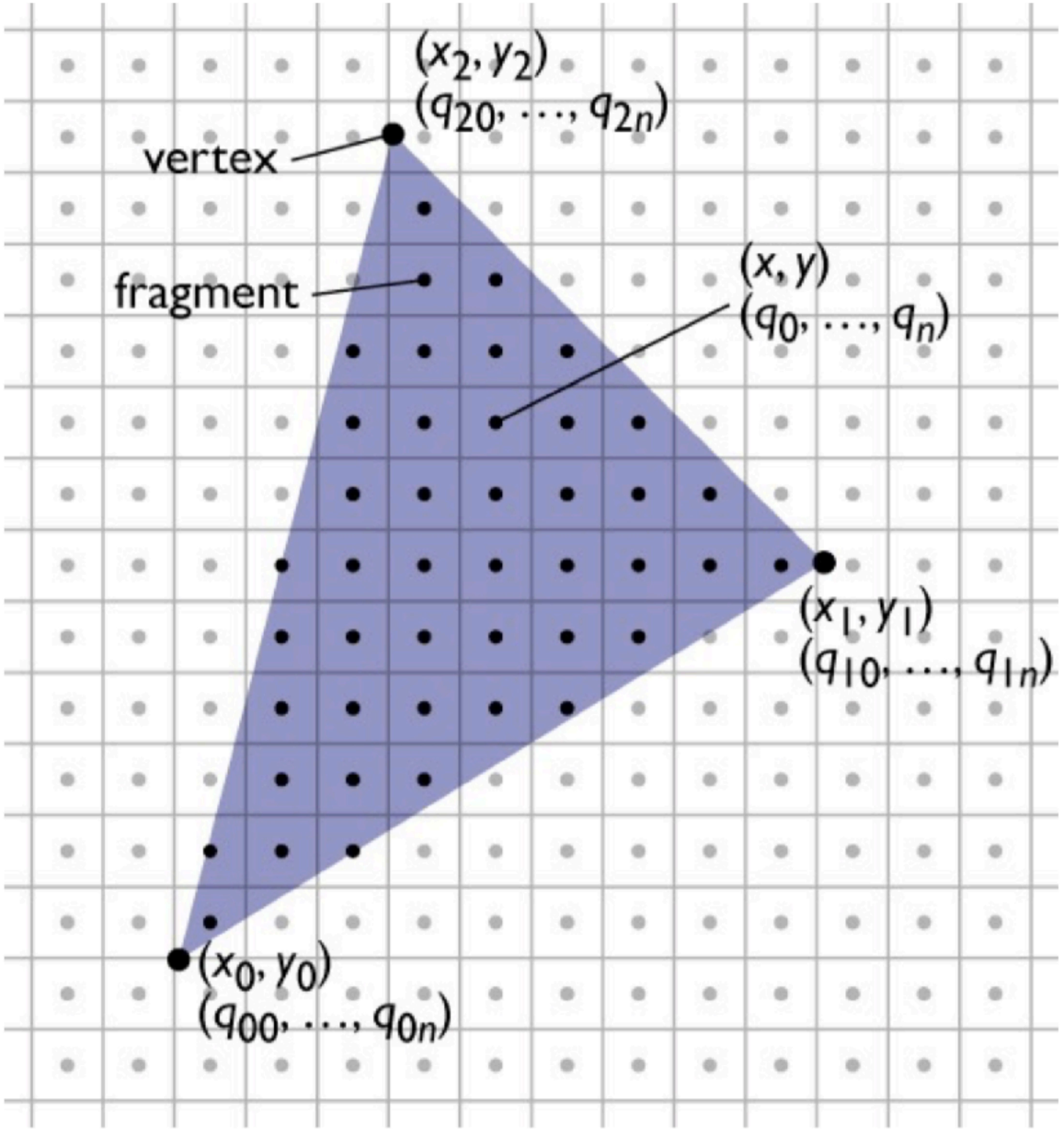
Rasterizing triangles

- Summary

1 evaluation of linear functions on pixel grid

2 these functions are defined by parameter values at vertices

3 using extra parameters to determine fragment set



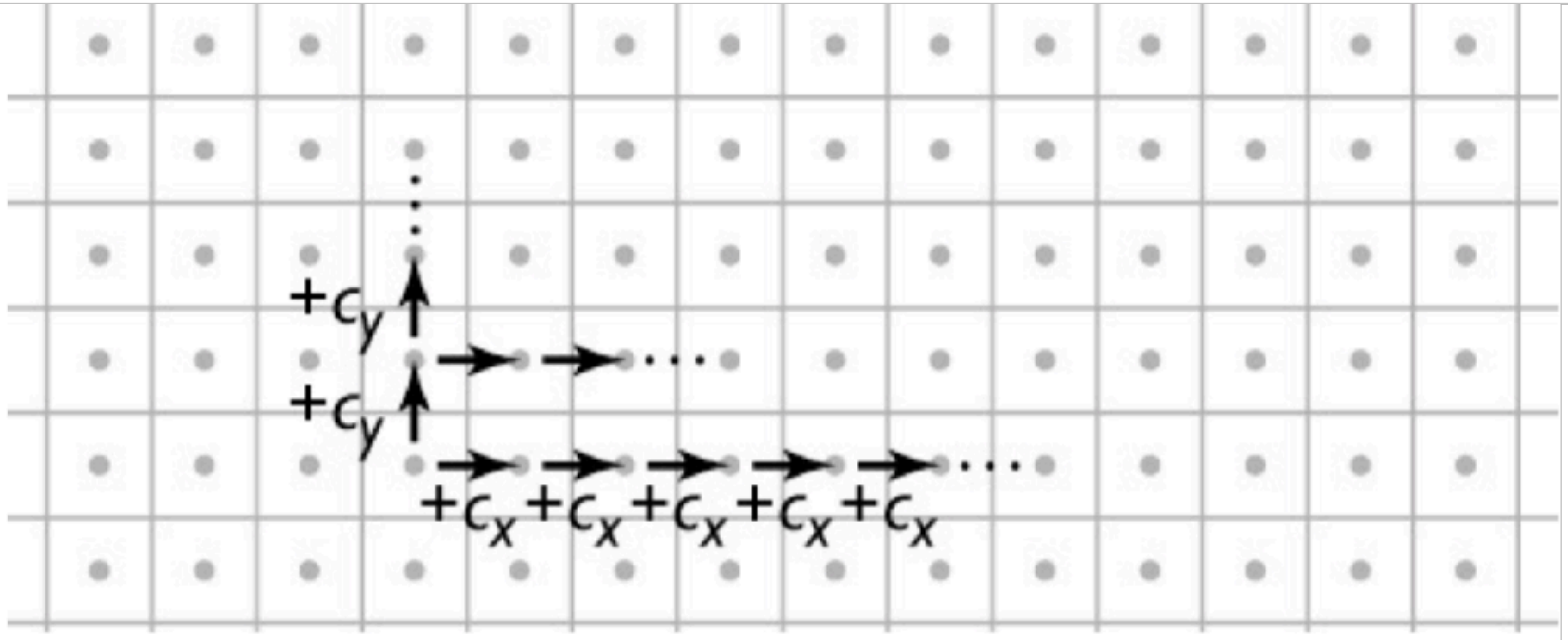
1. Incremental linear evaluation

- **A linear (affine, really) function on the plane is:**

$$q(x, y) = c_x x + c_y y + c_k$$

- **Linear functions are efficient to evaluate on a grid:**

$$q(x+1, y) = c_x(x+1) + c_y y + c_k = q(x, y) + c_x$$
$$q(x, y+1) = c_x x + c_y(y+1) + c_k = q(x, y) + c_y$$



Incremental linear evaluation

```
linEval(xm, xM, ym, yM, cx, cy, ck) {

  // setup
  qRow = cx*xm + cy*ym + ck;

  // traversal
  for y = ym to yM {
    qPix = qRow;
    for x = xm to xM {
      output(x, y, qPix);
      qPix += cx;
    }
    qRow += cy;
  }
}
```

$c_x = .005; c_y = .005; c_k = 0$

(image size 100x100)

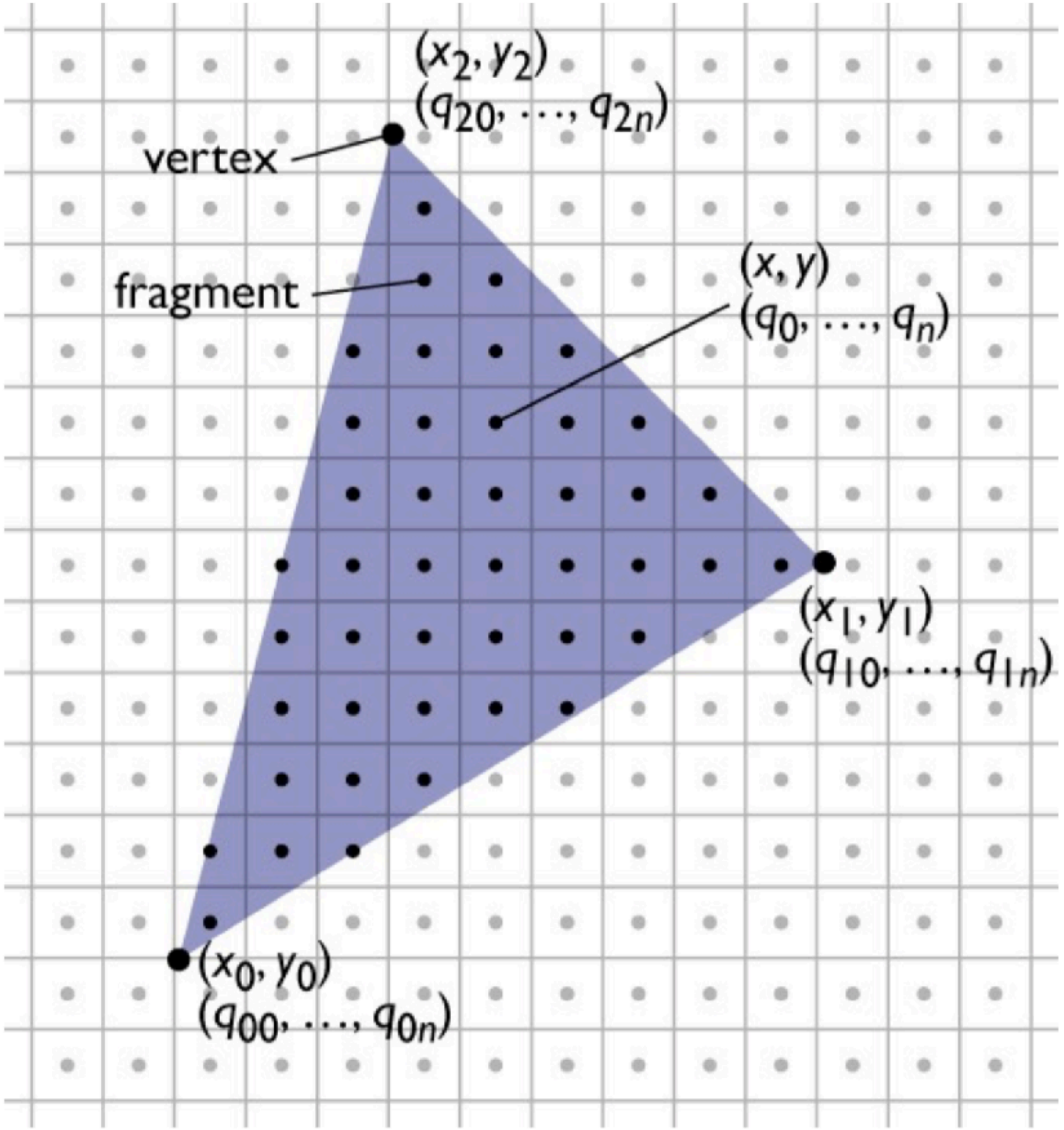
Rasterizing triangles

- Summary

1 evaluation of linear functions on pixel grid

2 these functions are defined by parameter values at vertices

3 using extra parameters to determine fragment set



2. Defining parameter functions

- **To interpolate parameters across a triangle we need to find the c_x, c_y, and c_k that define the (unique) linear function that matches the given values at all 3 vertices**
  - this is 3 constraints on 3 unknown coefficients:

$$c_x x_0 + c_y y_0 + c_k = q_0$$
$$c_x x_1 + c_y y_1 + c_k = q_1$$
$$c_x x_2 + c_y y_2 + c_k = q_2$$

(each states that the function agrees with the given value at one vertex)

  - leading to a 3x3 matrix equation for the coefficients:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_k \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}$$

(singular iff triangle is degenerate)

Defining parameter functions

- **More efficient version: shift origin to (x_0, y_0)**

$$q(x, y) = c_x(x - x_0) + c_y(y - y_0) + q_0$$
$$q(x_1, y_1) = c_x(x_1 - x_0) + c_y(y_1 - y_0) + q_0 = q_1$$
$$q(x_2, y_2) = c_x(x_2 - x_0) + c_y(y_2 - y_0) + q_0 = q_2$$

  - now this is a 2x2 linear system (since q_0 falls out):

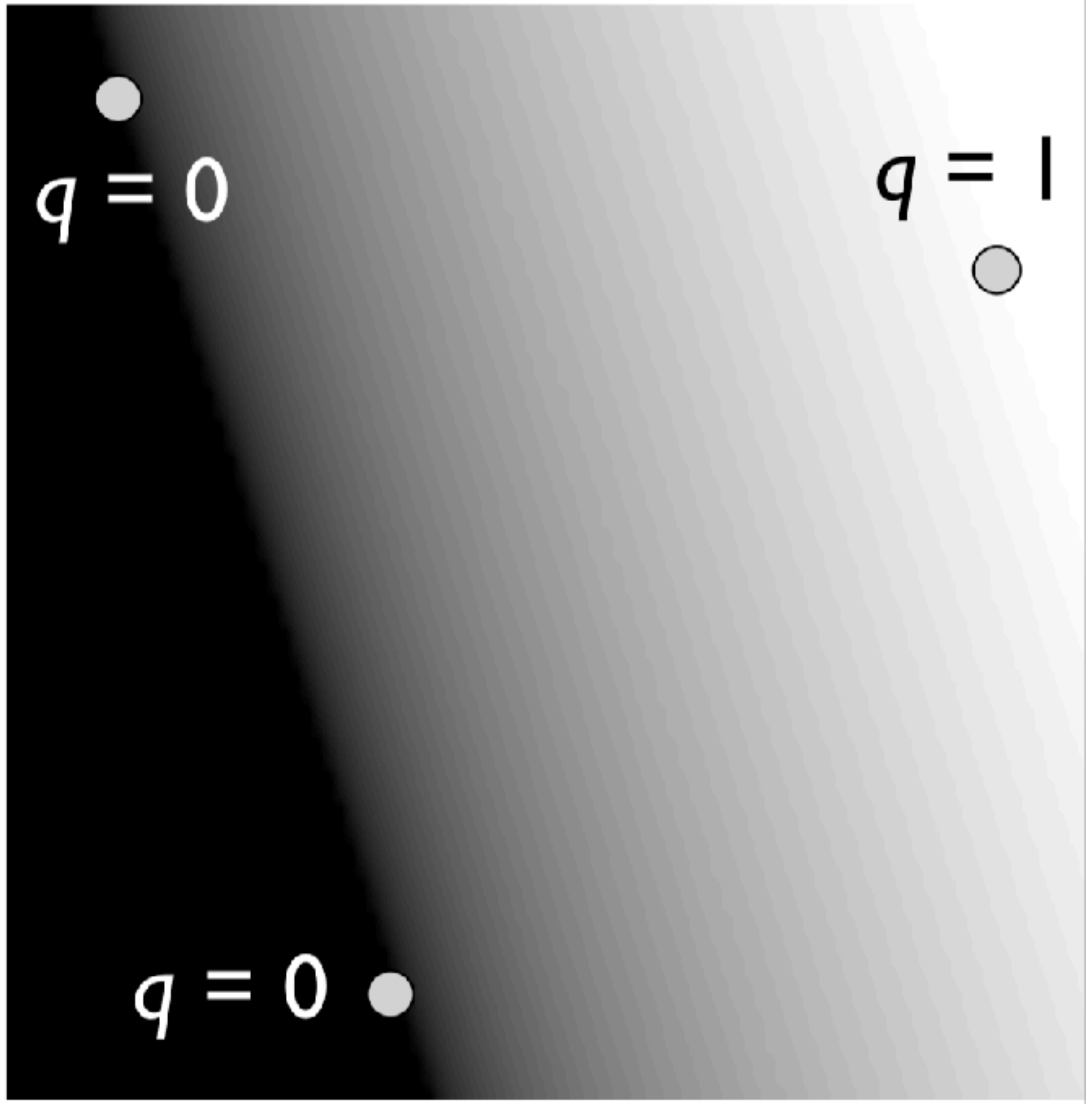
$$\begin{bmatrix} (x_1 - x_0) & (y_1 - y_0) \\ (x_2 - x_0) & (y_2 - y_0) \end{bmatrix} \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} q_1 - q_0 \\ q_2 - q_0 \end{bmatrix}$$

  - solve using Cramer's rule (see Shirley):

$$c_x = (\Delta q_1 \Delta y_2 - \Delta q_2 \Delta y_1)/(\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)$$
$$c_y = (\Delta q_2 \Delta x_1 - \Delta q_1 \Delta x_2)/(\Delta x_1 \Delta y_2 - \Delta x_2 \Delta y_1)$$

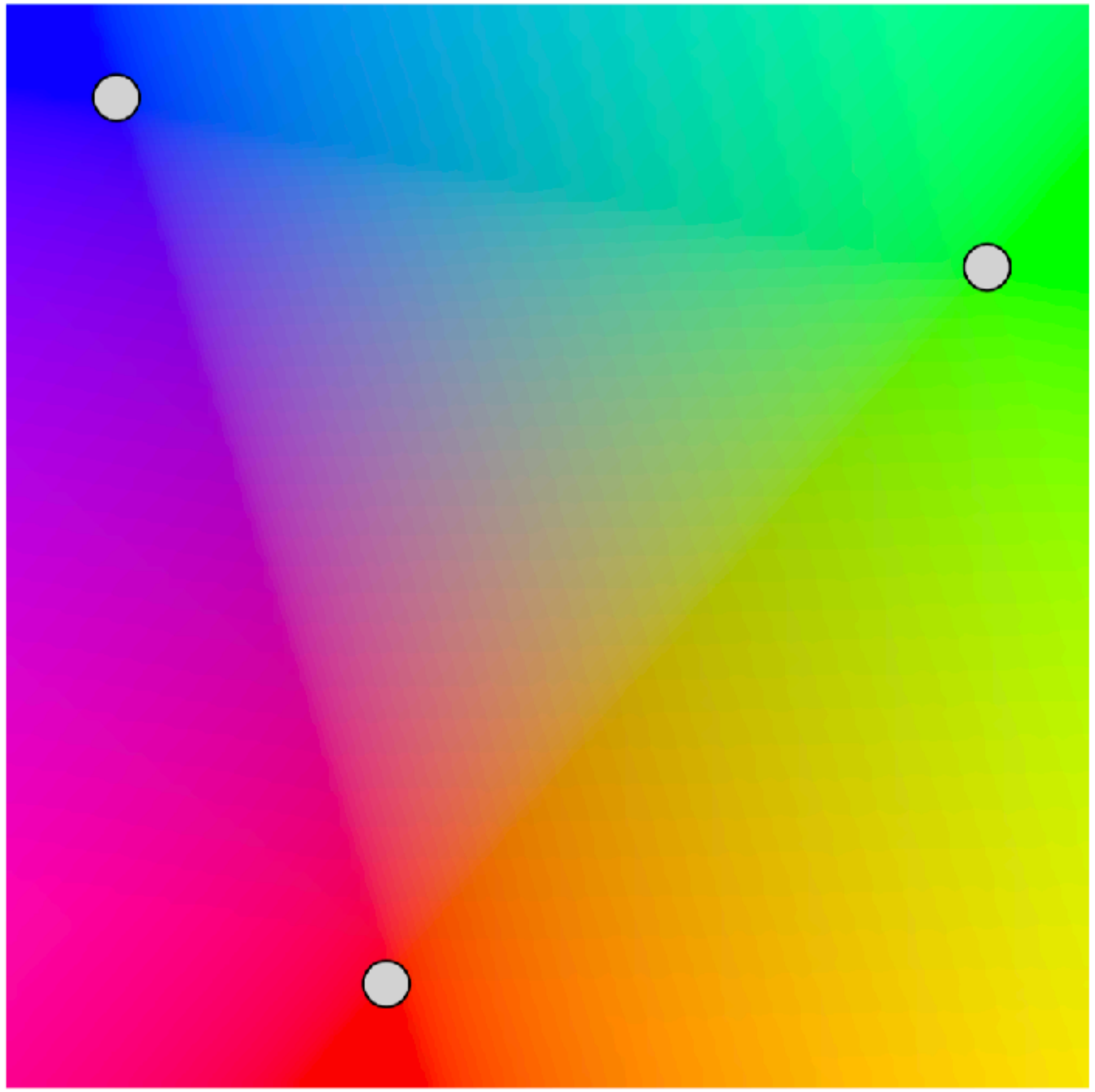
Defining parameter functions

```
linInterp(xm, xM, ym, yM, x0, y0, q0,
          x1, y1, q1, x2, y2, q2) {

    // setup
    det = (x1-x0)*(y2-y0) - (x2-x0)*(y1-y0);
    cx = ((q1-q0)*(y2-y0) - (q2-q0)*(y1-y0)) / det;
    cy = ((q2-q0)*(x1-x0) - (q1-q0)*(x2-x0)) / det;
    qRow = cx*(xm-x0) + cy*(ym-y0) + q0;

    // traversal (same as before)
    for y = ym to yM {
        qPix = qRow;
        for x = xm to xM {
            output(x, y, qPix);
            qPix += cx;
        }
        qRow += cy;
    }
}
```



Interpolating several parameters

```
linInterp(xm, xM, ym, yM, n, x0, y0, q0[],
        x1, y1, q1[], x2, y2, q2[]) {

  // setup
  for k = 0 to n-1
     // compute cx[k], cy[k], qRow[k]
     // from q0[k], q1[k], q2[k]

  // traversal
  for y = ym to yM {
     for k = 1 to n, qPix[k] = qRow[k];
     for x = xm to xM {
        output(x, y, qPix);
        for k = 1 to n, qPix[k] += cx[k];
     }
     for k = 1 to n, qRow[k] += cy[k];
  }
}
```



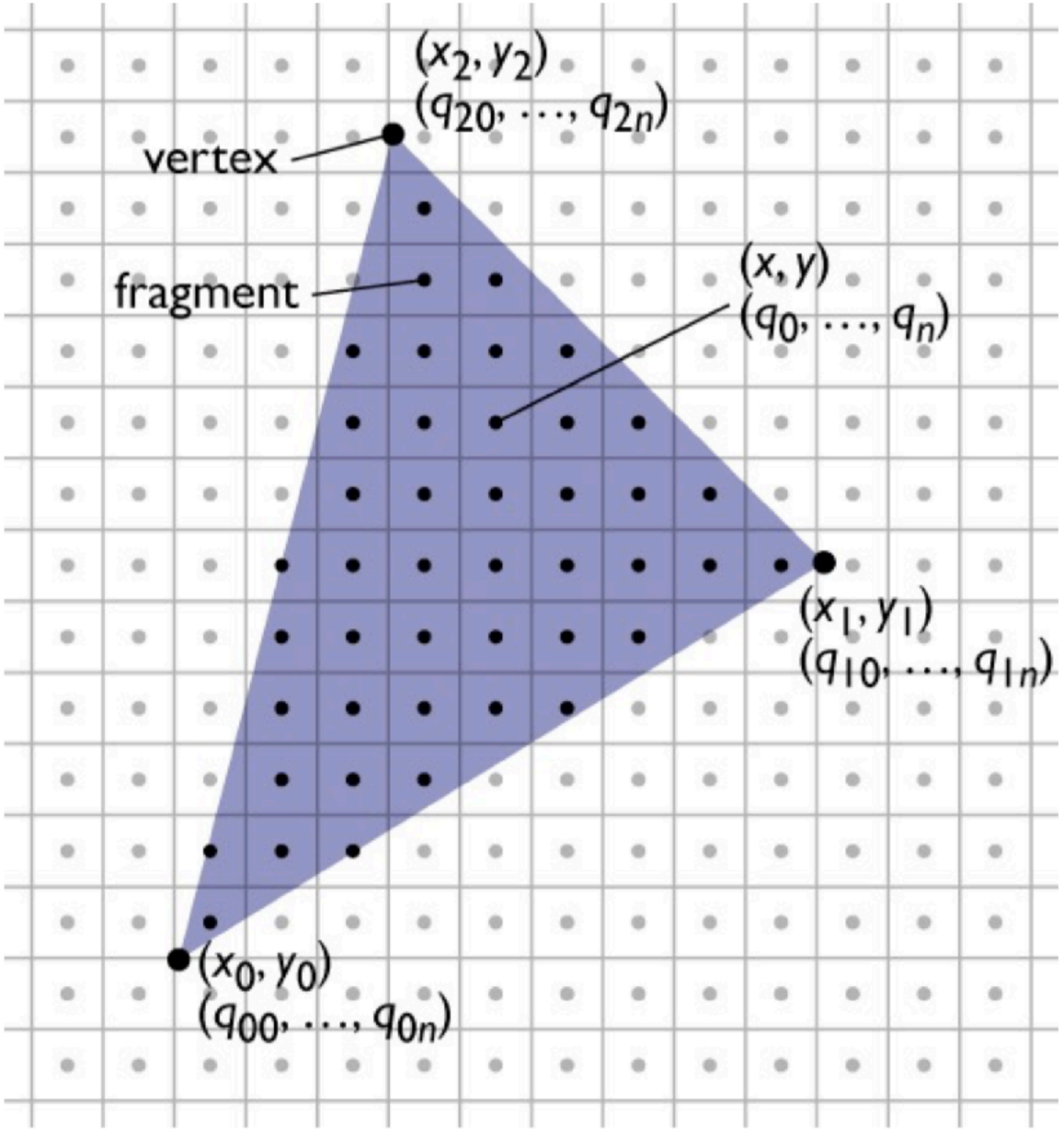
Rasterizing triangles

- Summary

1 evaluation of linear functions on pixel grid

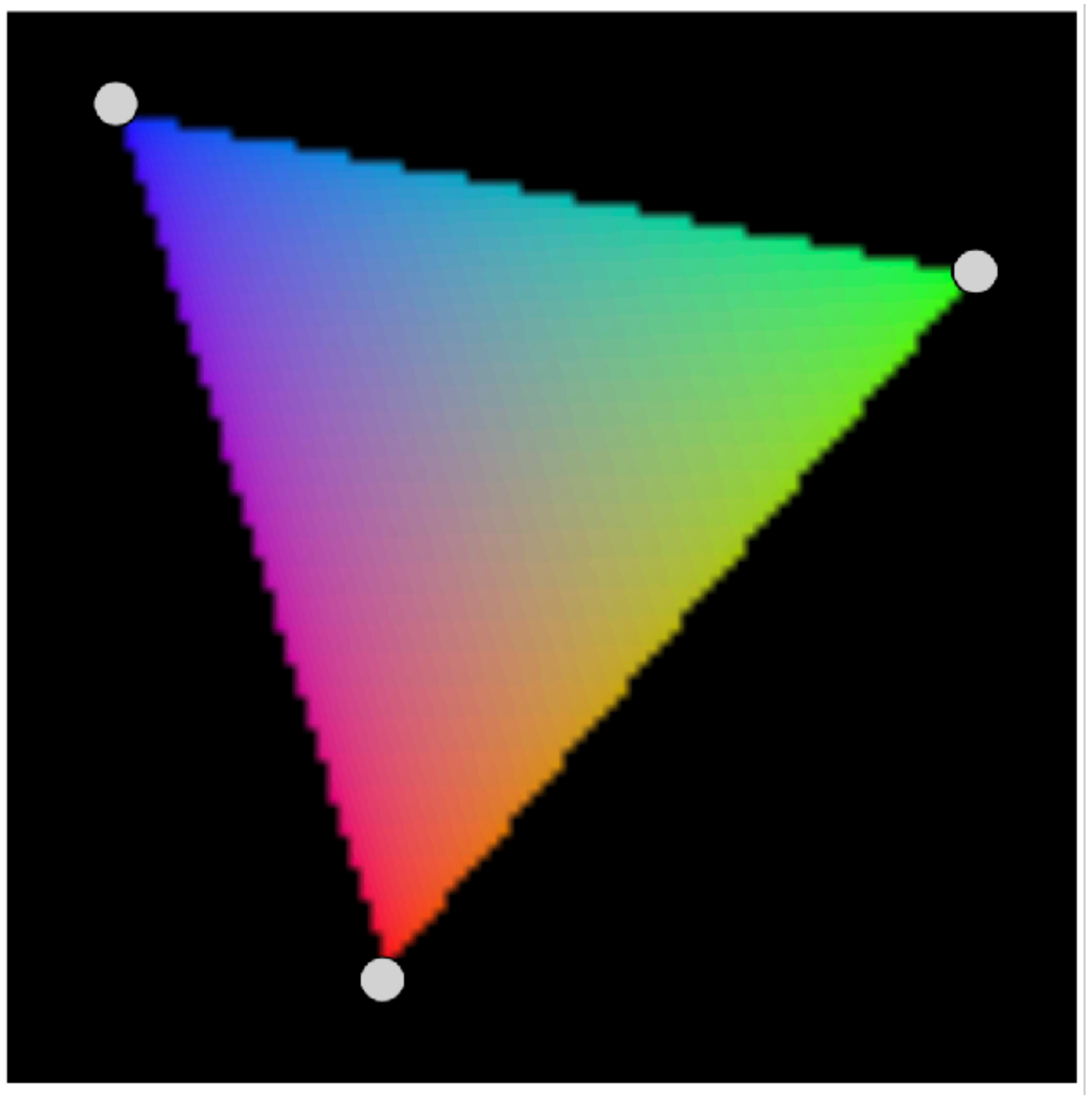
2 these functions are defined by parameter values at vertices

3 using extra parameters to determine fragment set



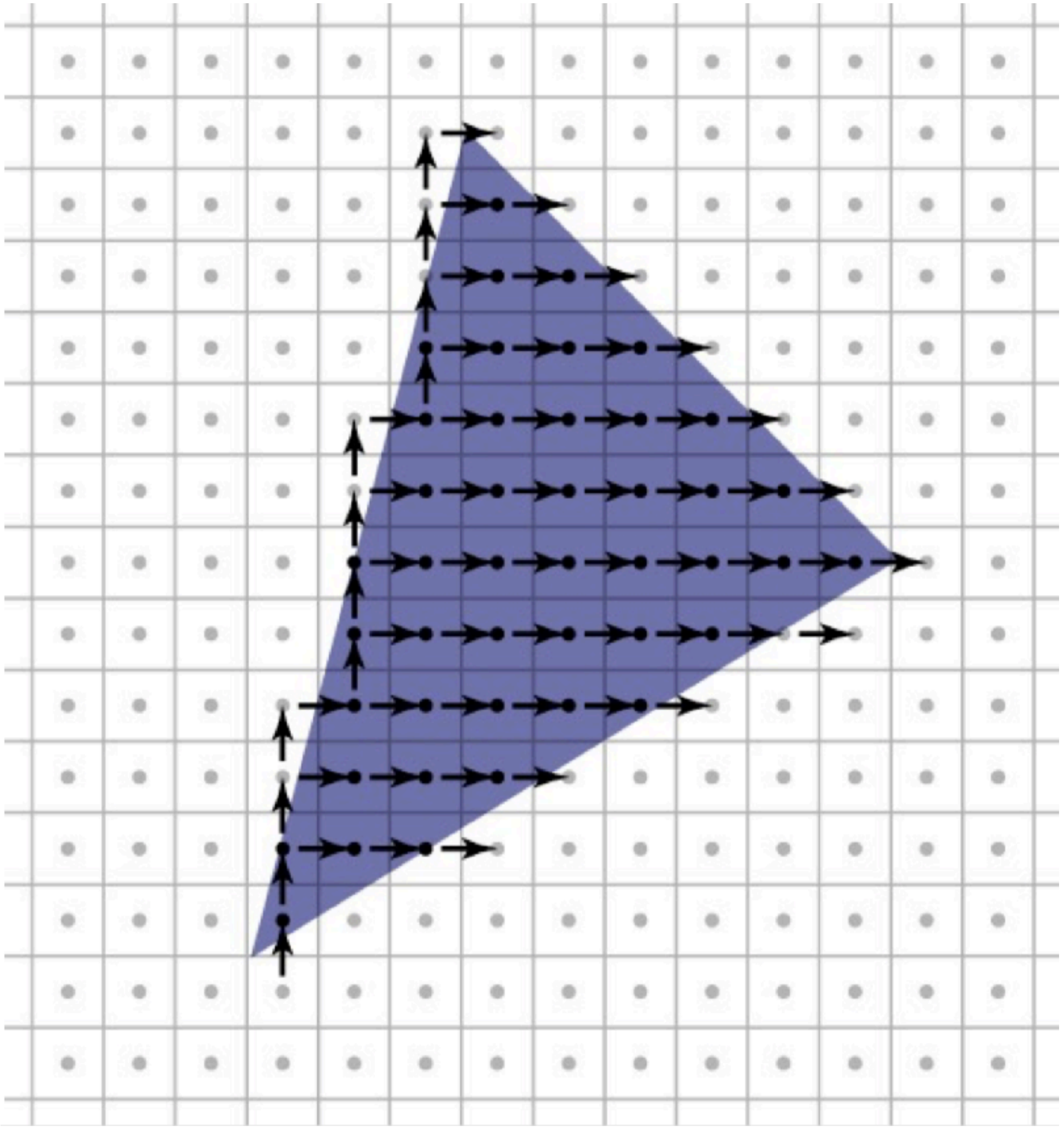
3. Clipping to the triangle

- Interpolate three barycentric coordinates across the plane
  - recall each barycentric coord is 1 at one vert. and 0 at the other two
- Output fragments only when all three are > 0.



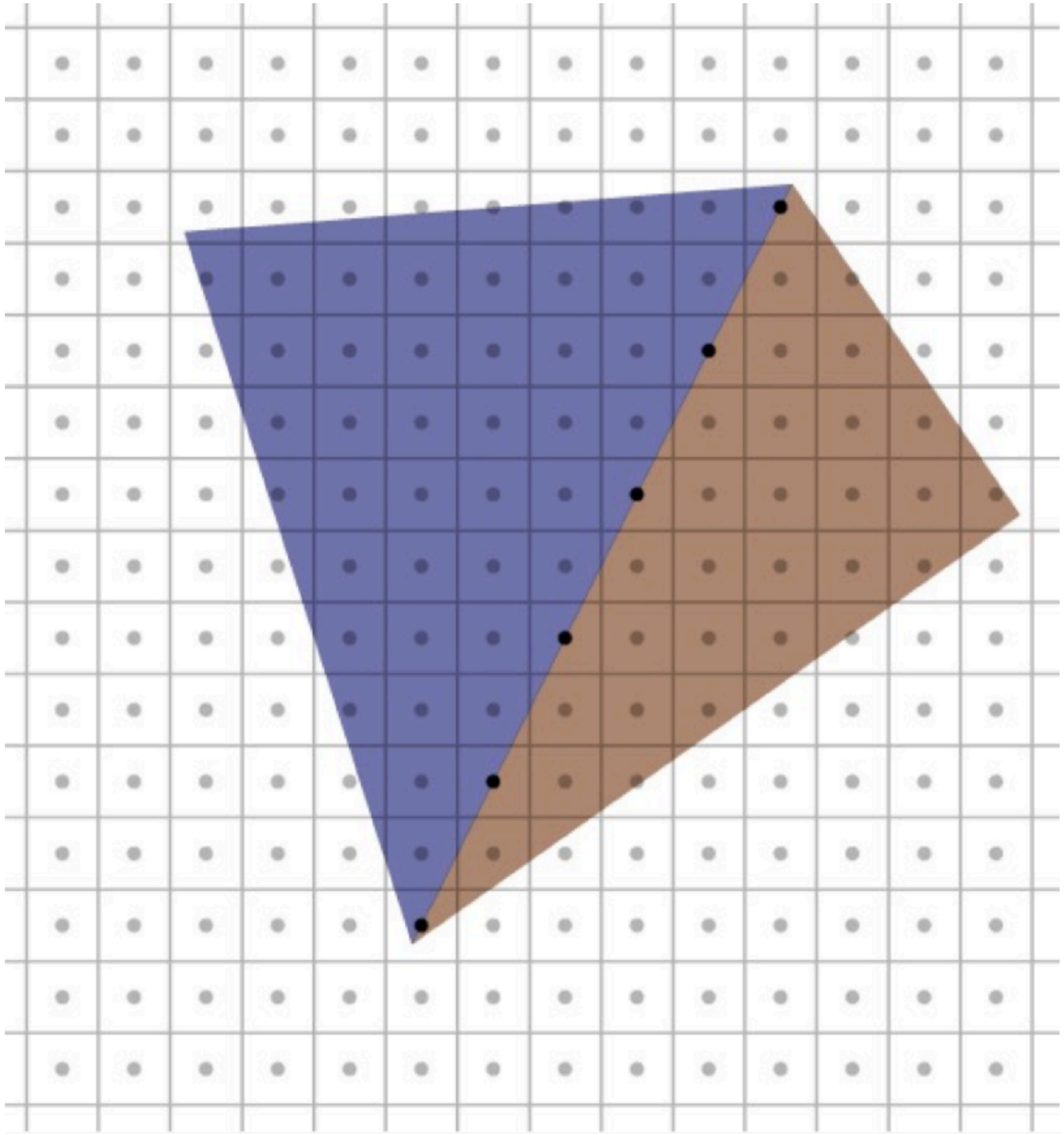
Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels (BBox)
- Interpolate linear functions
- Use those functions to determine when to emit a fragment



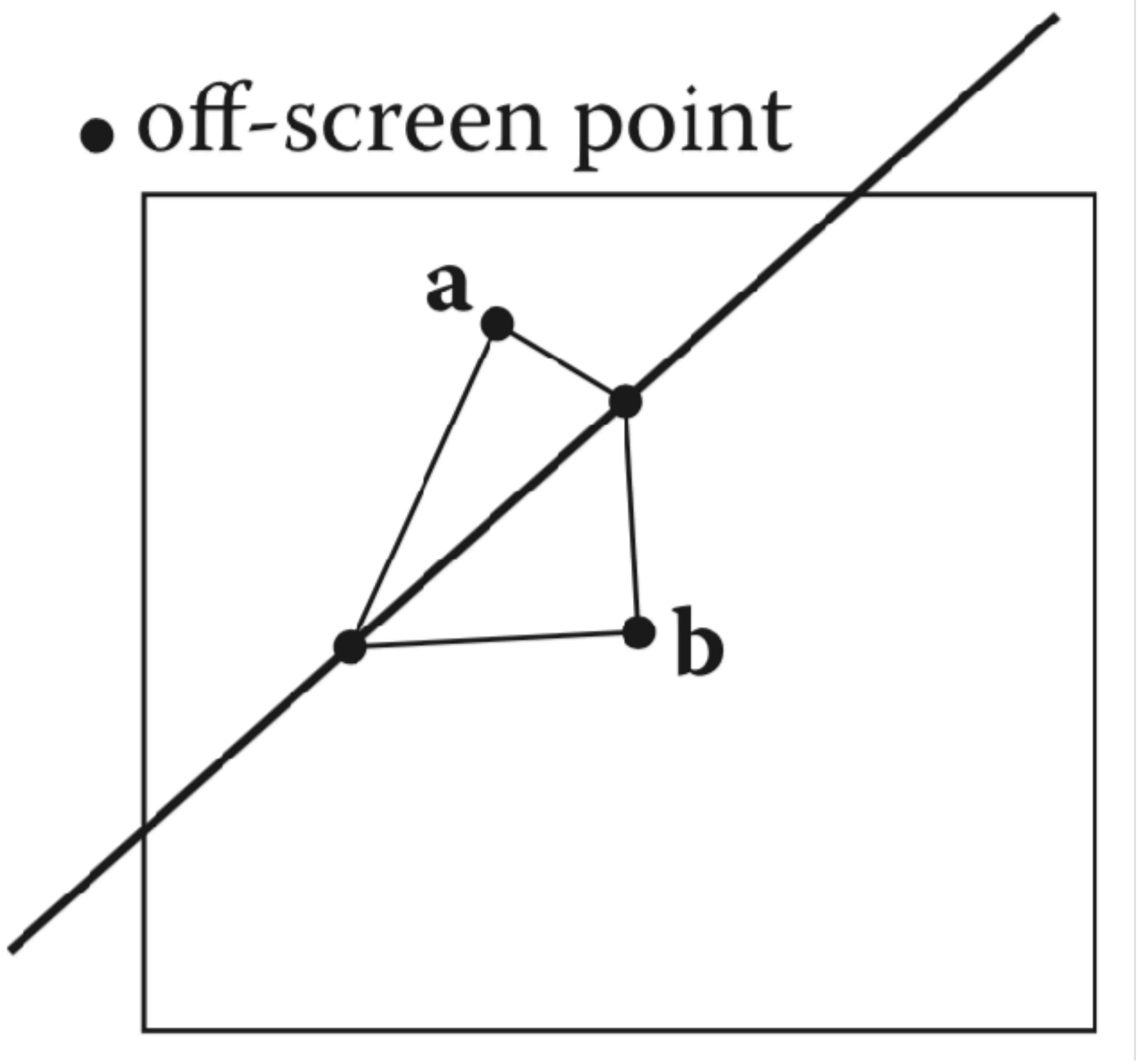
Rasterizing triangles

- Exercise caution with rounding and arbitrary decisions
  - need to visit these pixels once: no hole
  - but it's important not to visit them twice!



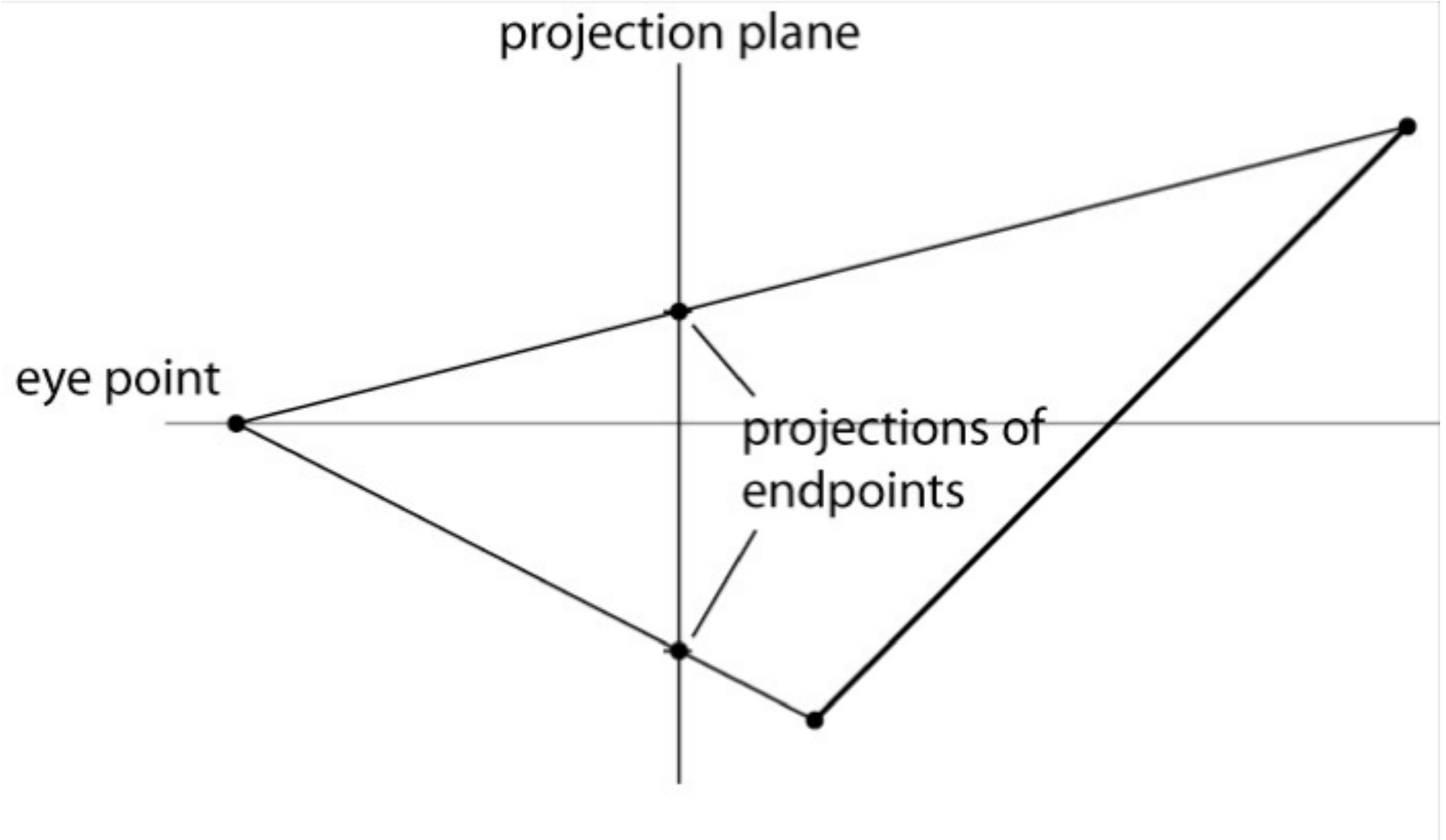
Rasterizing triangles

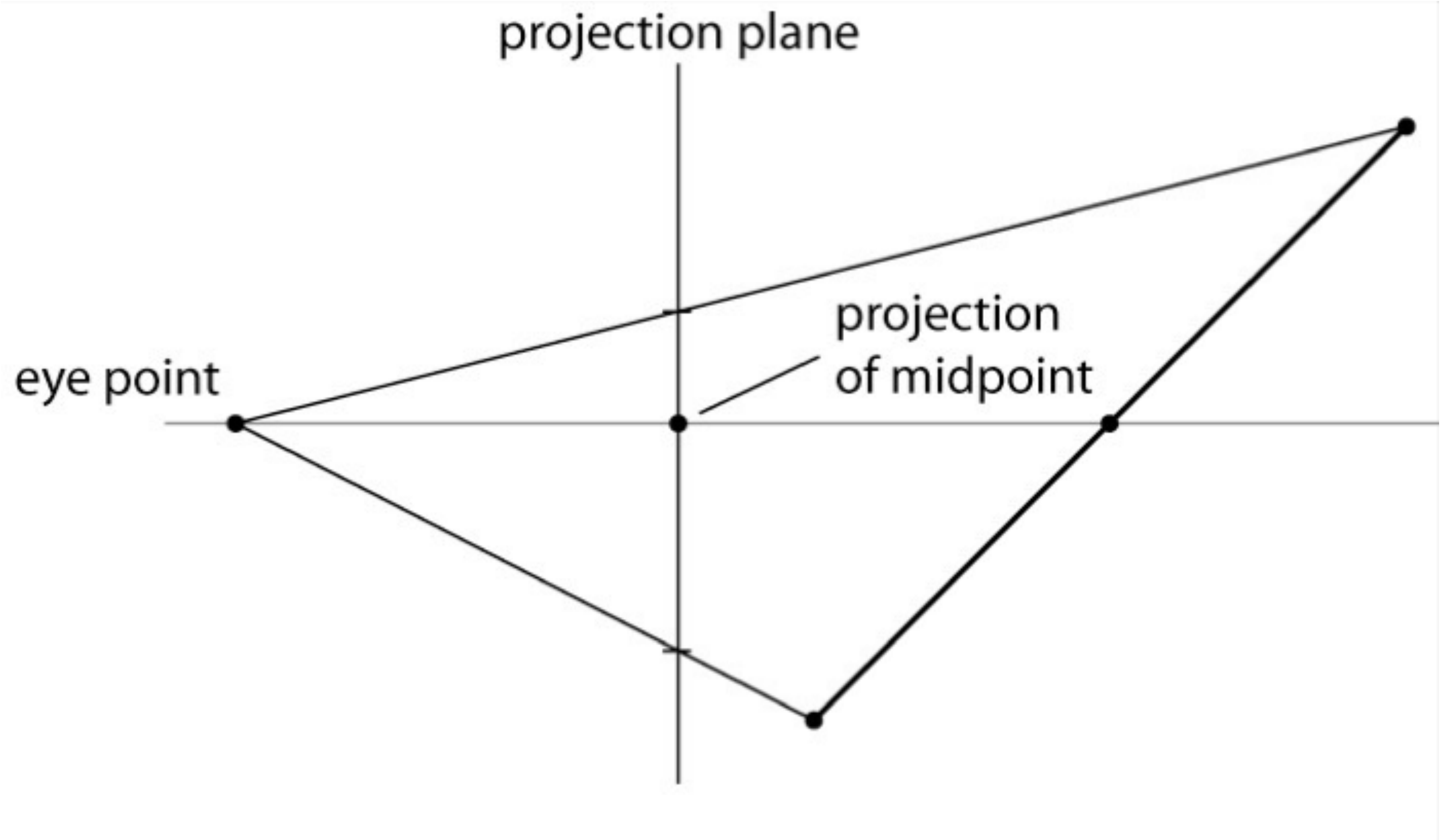
- Exercise caution with rounding and arbitrary decisions
  - need to visit these pixels once: no hole
  - but it's important not to visit them twice!
- Consistency
  - Coordinate inner contradiction via a global view: off-screen point p



Perspective and interpolation

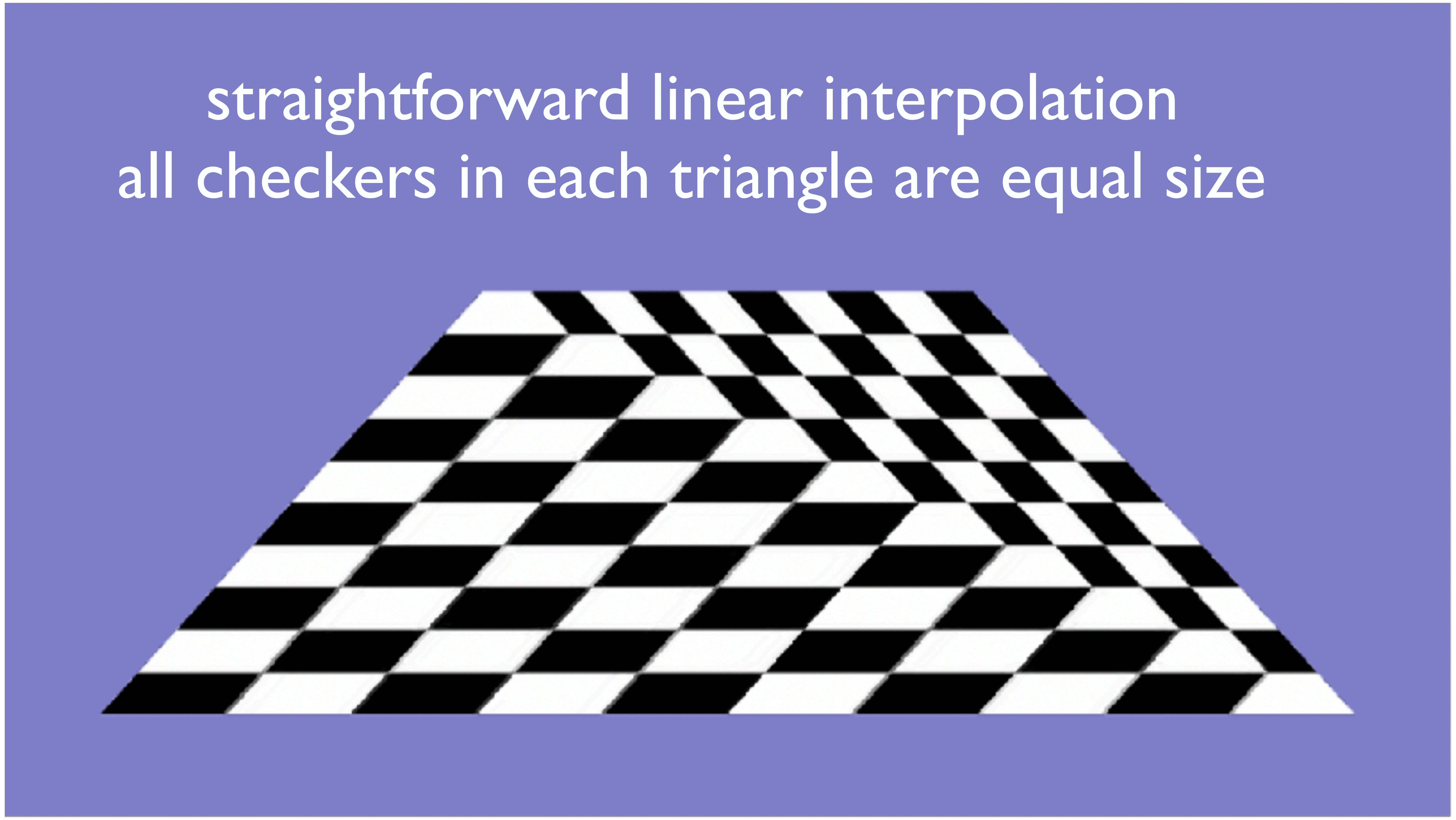
- **interpolating values in screen space is not the whole story**
  - often we are interpolating values that are supposed to vary linearly in the scene
  - because perspective projection
    does not preserve ratios of lengths, these values *should not vary linearly in screen space*





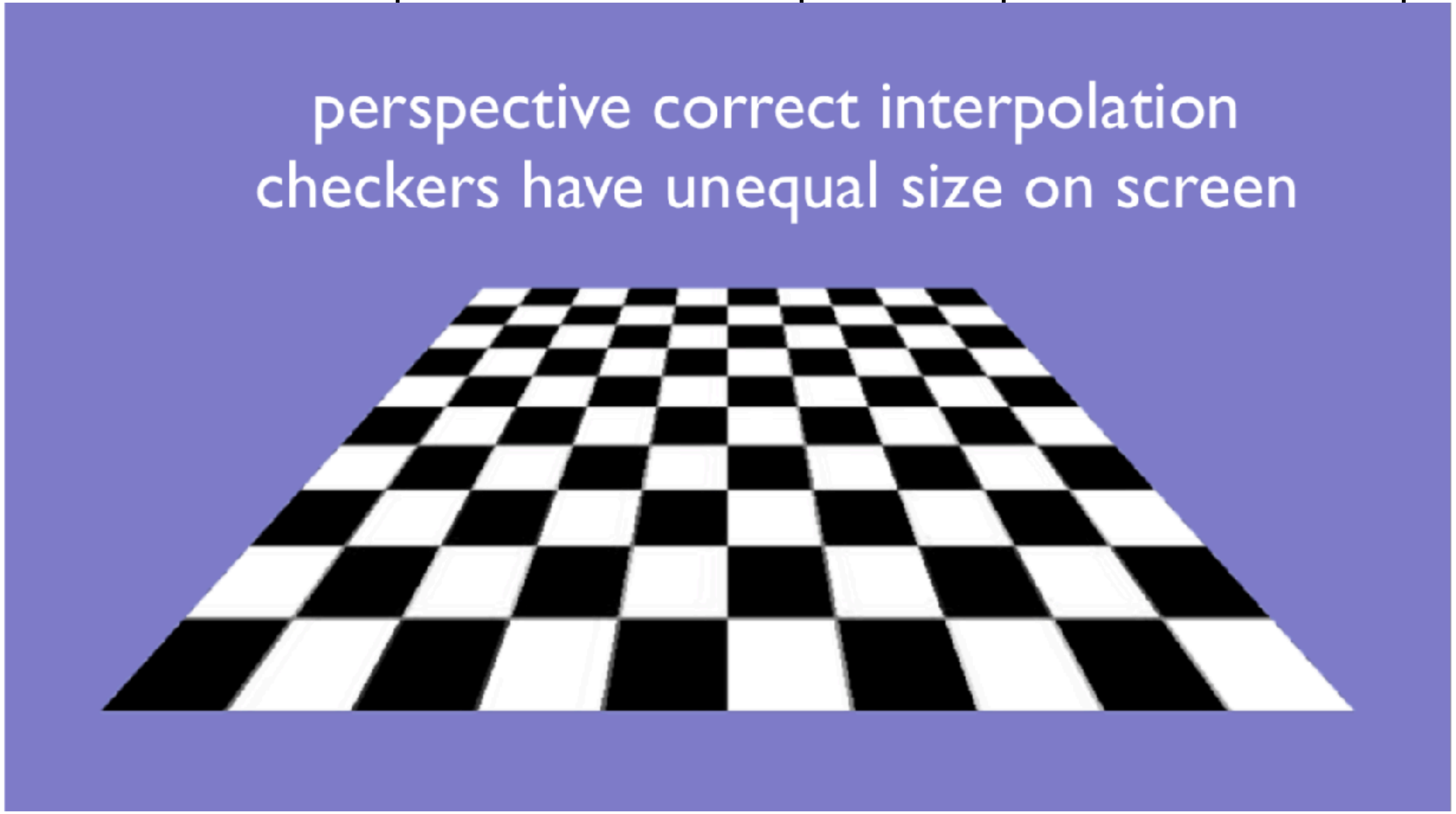
Perspective and interpolation

- Texture coordinates are the canonical example
  - equal steps in screen space are unequal steps in texture space



Perspective and interpolation

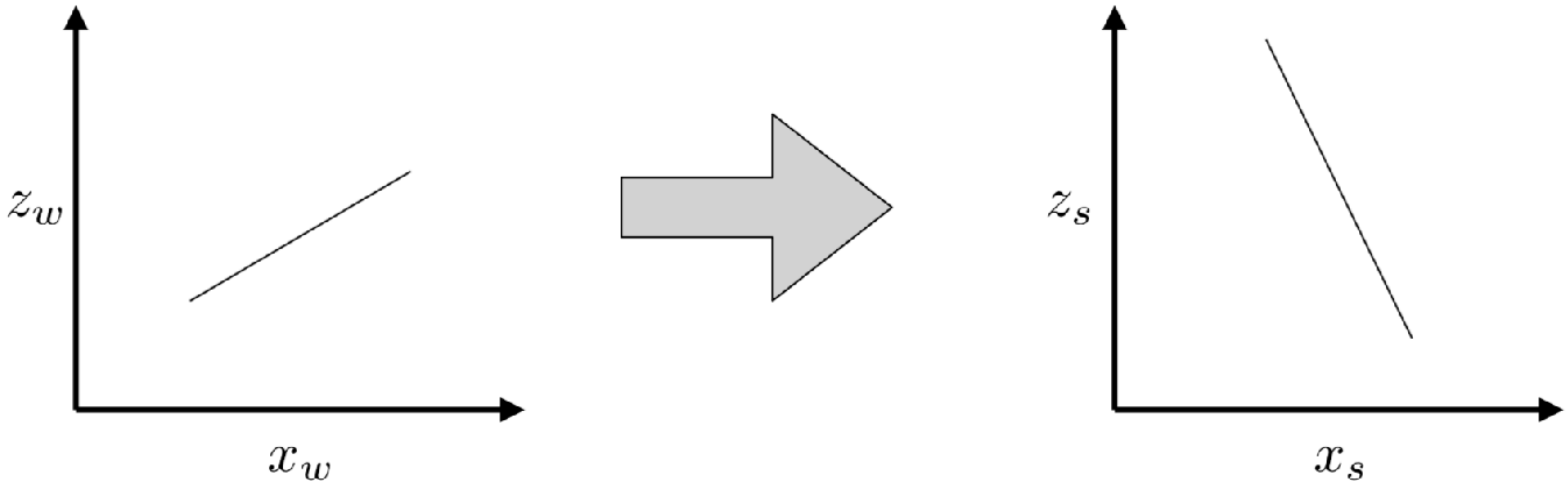
- Texture coordinates are the canonical example
  - equal steps in screen space are unequal steps in texture space



Perspective correct interpolation

- Linear interpolation still suffices if we do it the right way
  - remember projective transformations preserve straight lines

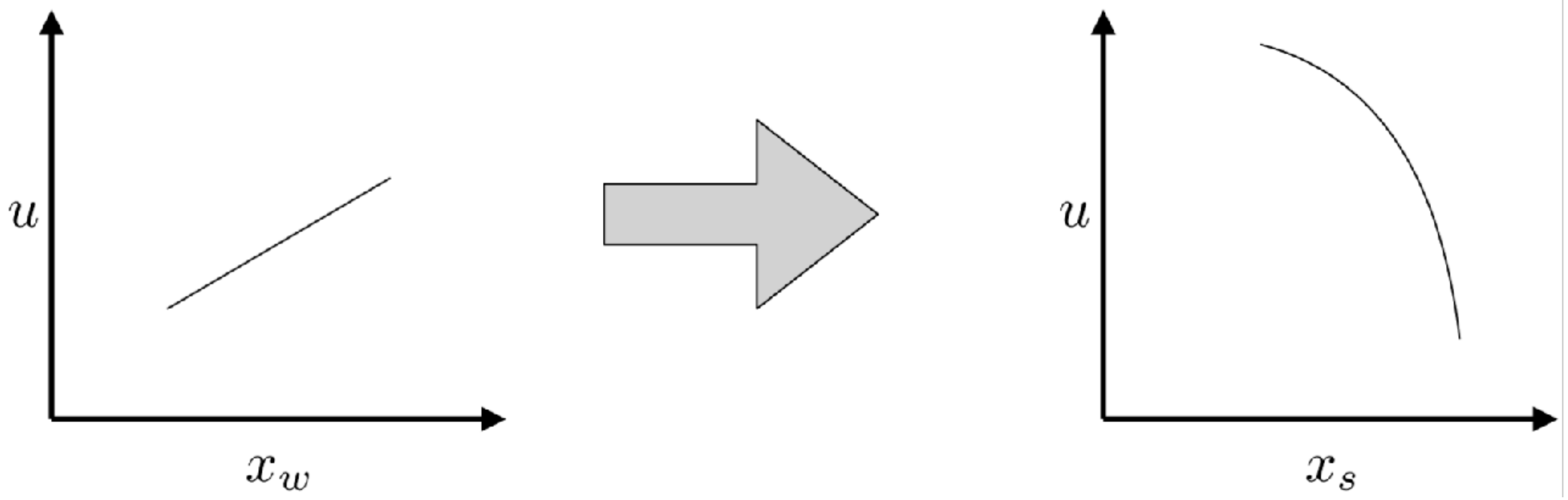
$$\begin{bmatrix} x_w \\ z_w \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x_c \\ z_c \\ w_c \end{bmatrix} \rightarrow \begin{bmatrix} x_c/w_c \\ z_c/w_c \\ 1 \end{bmatrix} = \begin{bmatrix} x_s \\ z_s \\ 1 \end{bmatrix}$$



Perspective correct interpolation

- Linear interpolation still suffices if we do it the right way
  - remember projective transformations preserve straight lines
  - just carrying the tex, coord. along is not a projective transform.

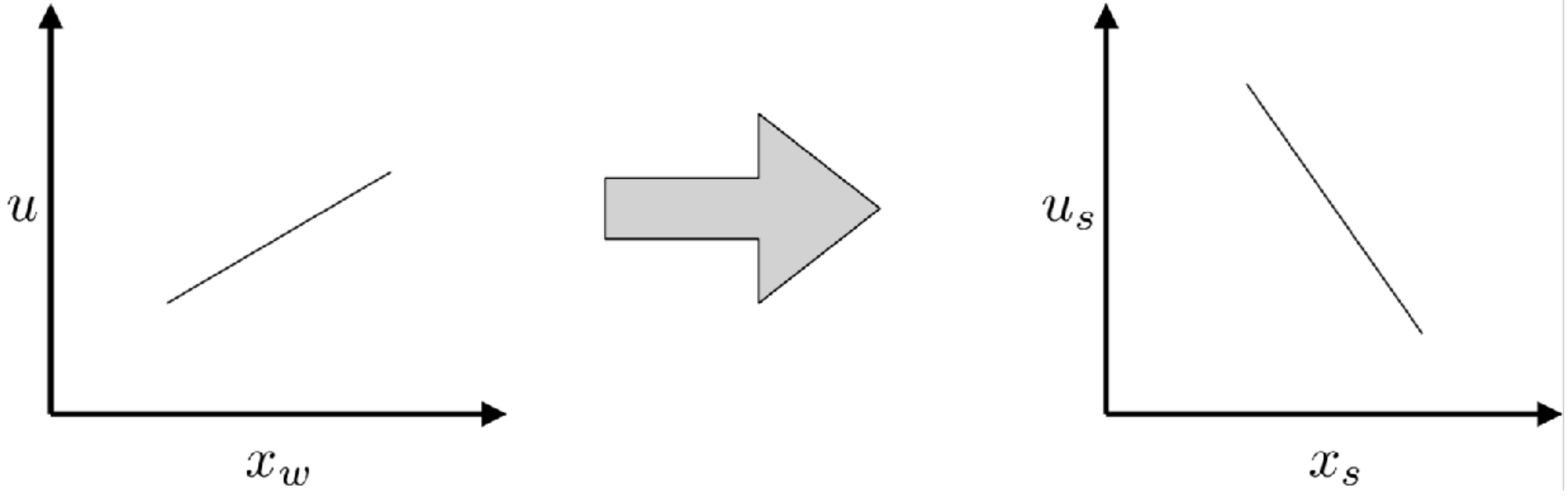
$$\begin{bmatrix} x_w \\ z_w \\ 1 \\ u \end{bmatrix} \to \begin{bmatrix} x_c \\ z_c \\ w_c \\ u \end{bmatrix} \to \begin{bmatrix} x_c/w_c \\ z_c/w_c \\ 1 \\ u \end{bmatrix} = \begin{bmatrix} x_s \\ z_s \\ 1 \\ u \end{bmatrix}$$



Perspective correct interpolation

- Solution: treat u and v as additional coordinates in the projective transformation
  - now the full transformation on (x, y, z, u, v) is projective

$$\begin{bmatrix} x_w \\ z_w \\ u \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x_c \\ z_c \\ u \\ w_c \end{bmatrix} \rightarrow \begin{bmatrix} x_c/w_c \\ z_c/w_c \\ u/w_c \\ 1 \end{bmatrix} = \begin{bmatrix} x_s \\ z_s \\ u_s \\ 1 \end{bmatrix}$$

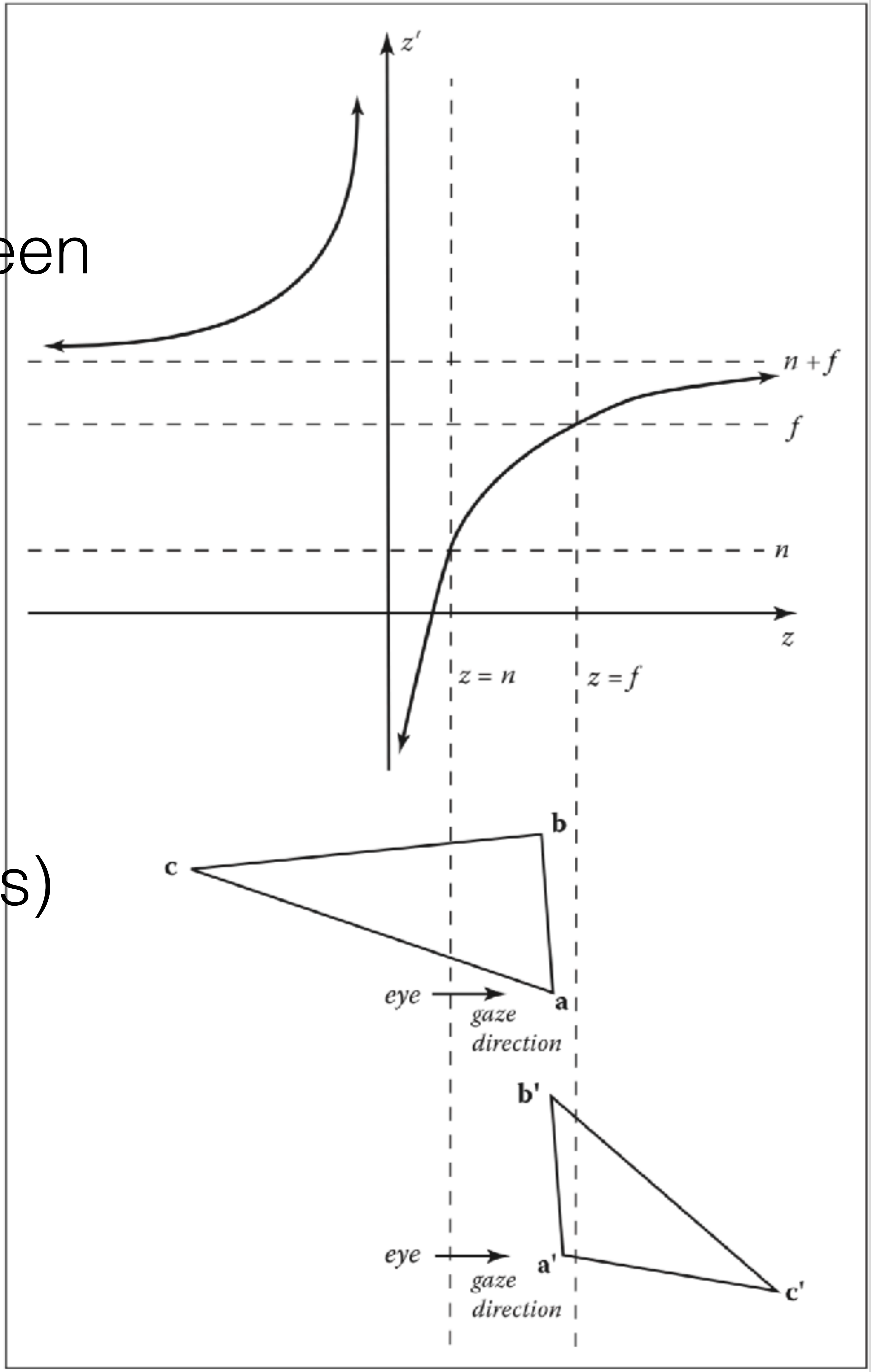


Perspective correct interpolation

- Bottom line: treat all attributes the same as (x, y, z)
  - divide them by w before interpolation
  - interpolate quantities u/w, etc., linearly across screen
  - also interpolate 1/w as an additional attribute
  - divide interpolated u/w by 1/w to recover u

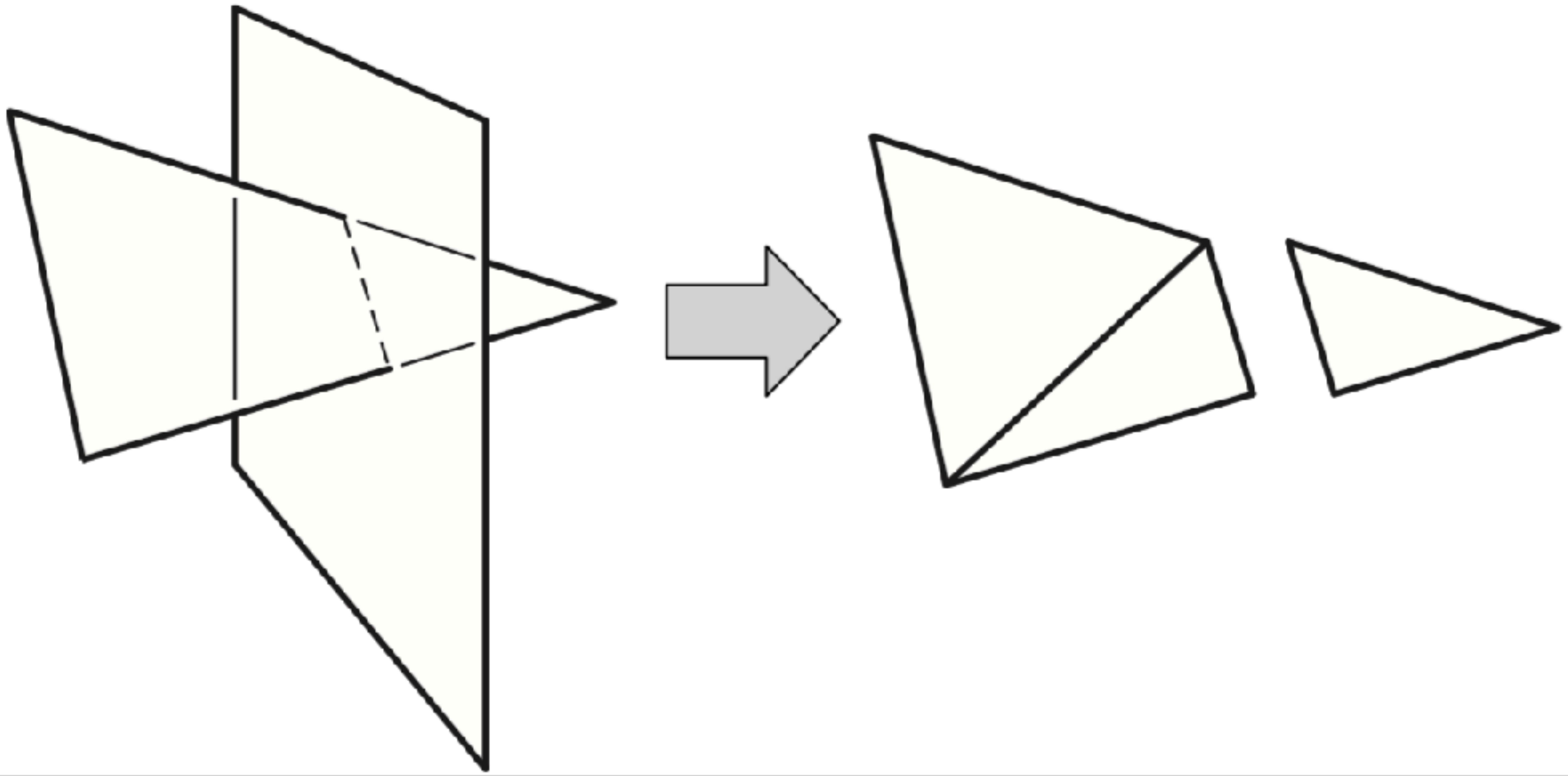
Clipping

- Rasterizer tends to assume triangles are on screen
  - particularly problematic to have triangles crossing
    the plane $z = 0$
- After projection, before perspective divide
  - clip against the planes x, y, $z = 1, -1$ (6 planes)
  - primitive operation: clip triangle against axis-aligned plane



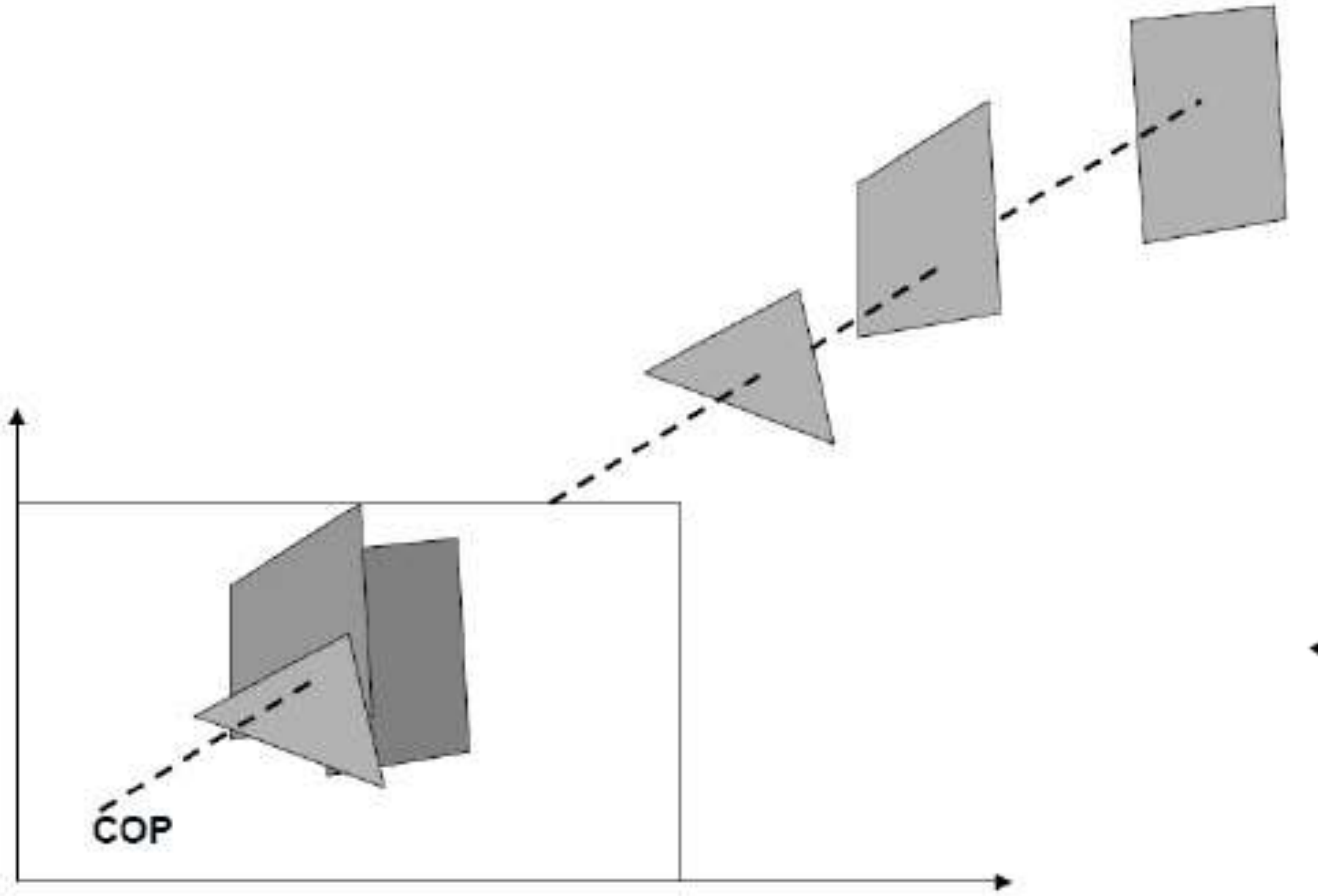
Clipping a triangle against a plane

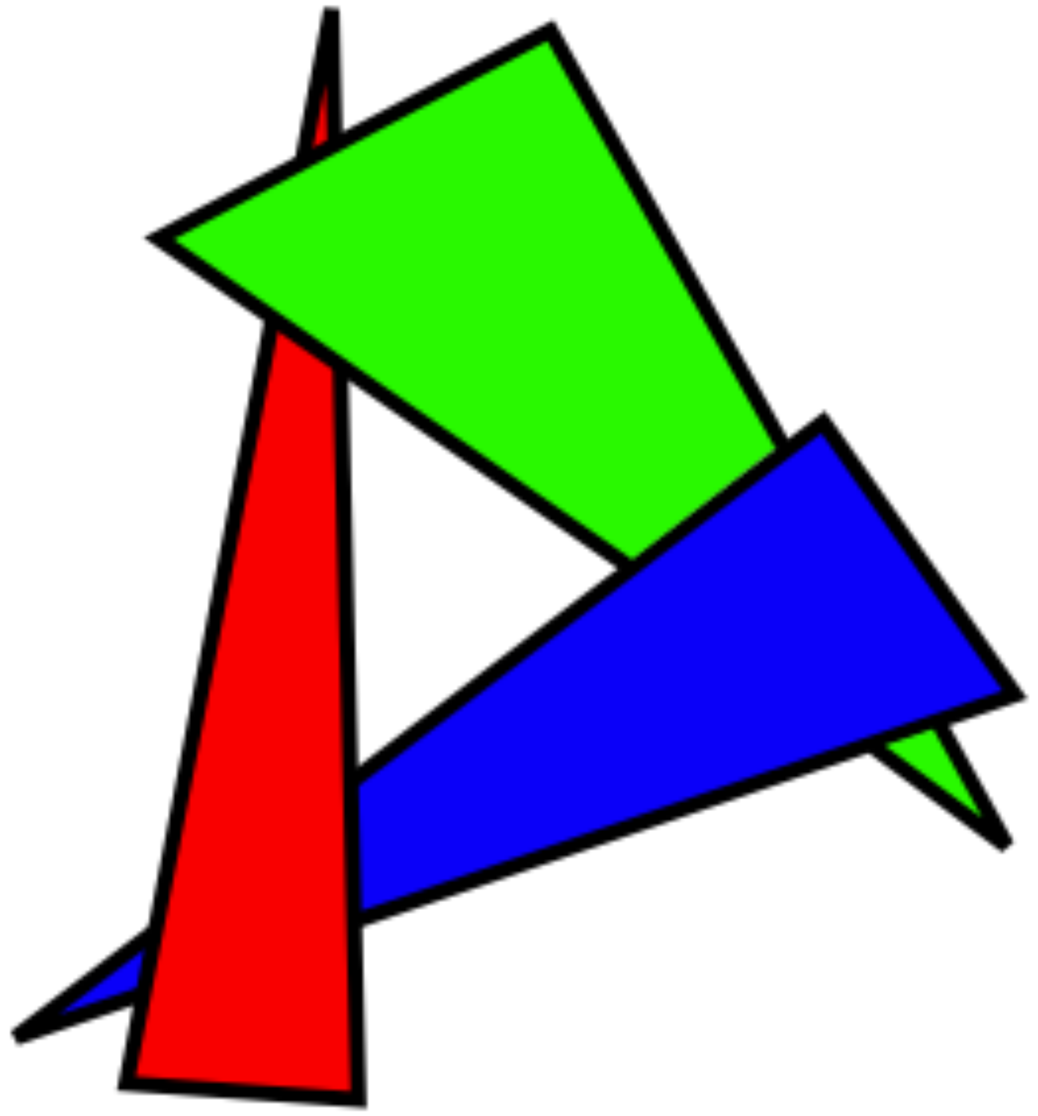
- 4 cases, based on sidedness of vertices
  - all in (keep)
  - all out (discard)
  - one in, two out (one clipped triangle)
  - two in, one out (two clipped triangles)



Objects Depth Sorting

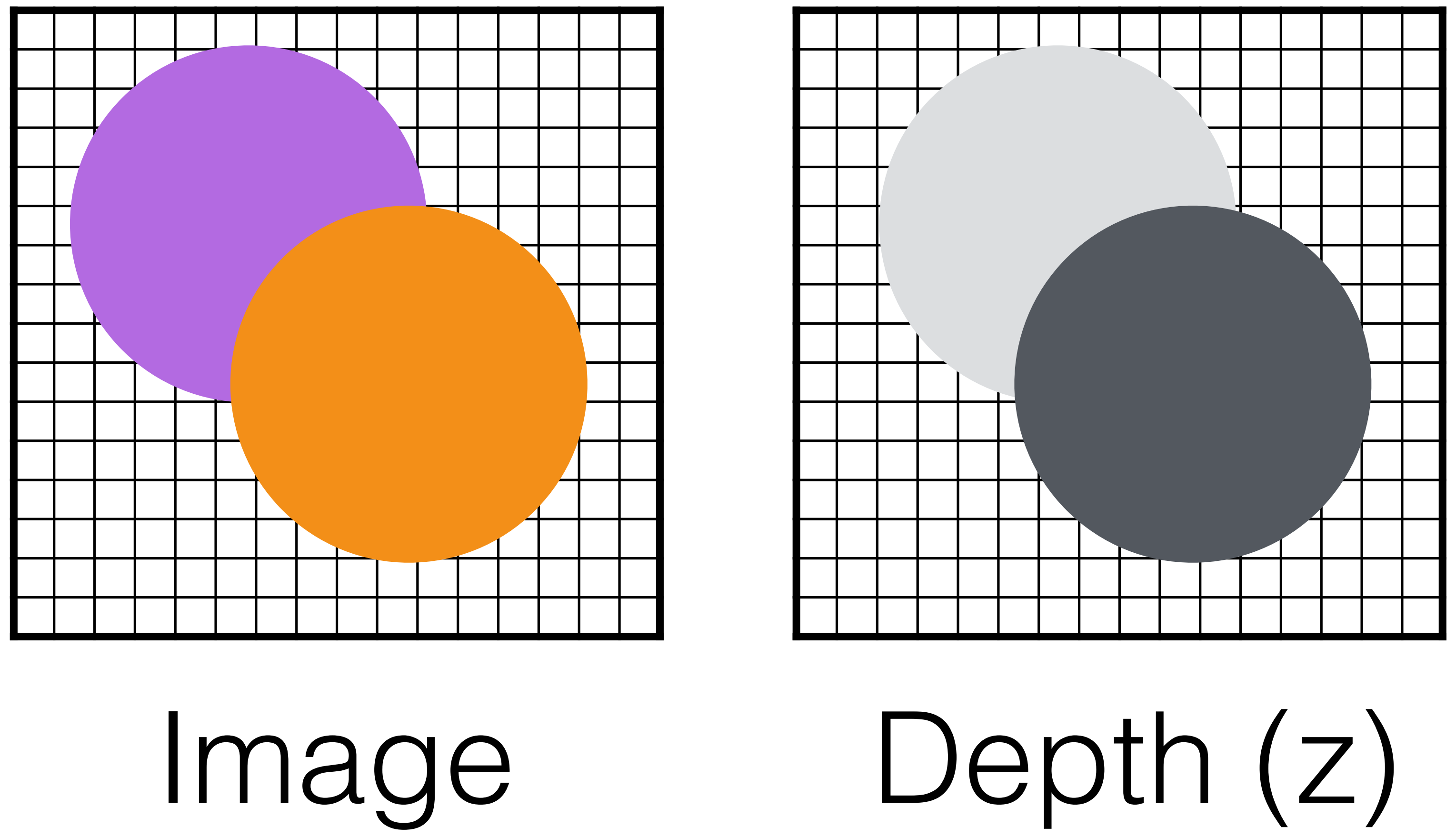
- To handle occlusion, you can sort all the objects in a scene by depth





- This is not always possible!

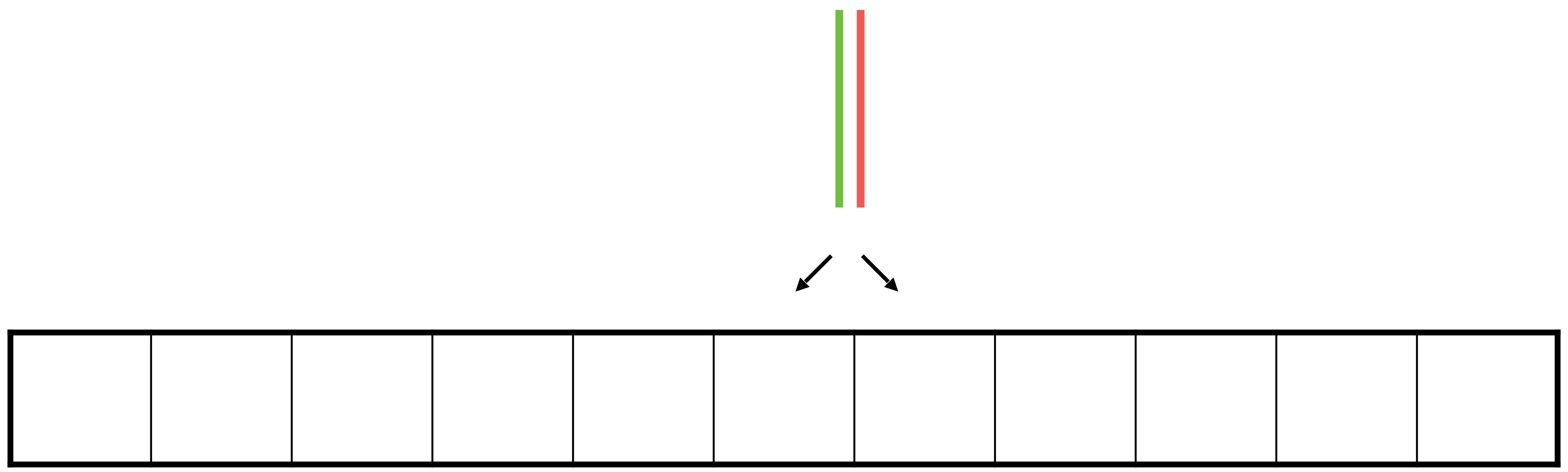
z-buffering



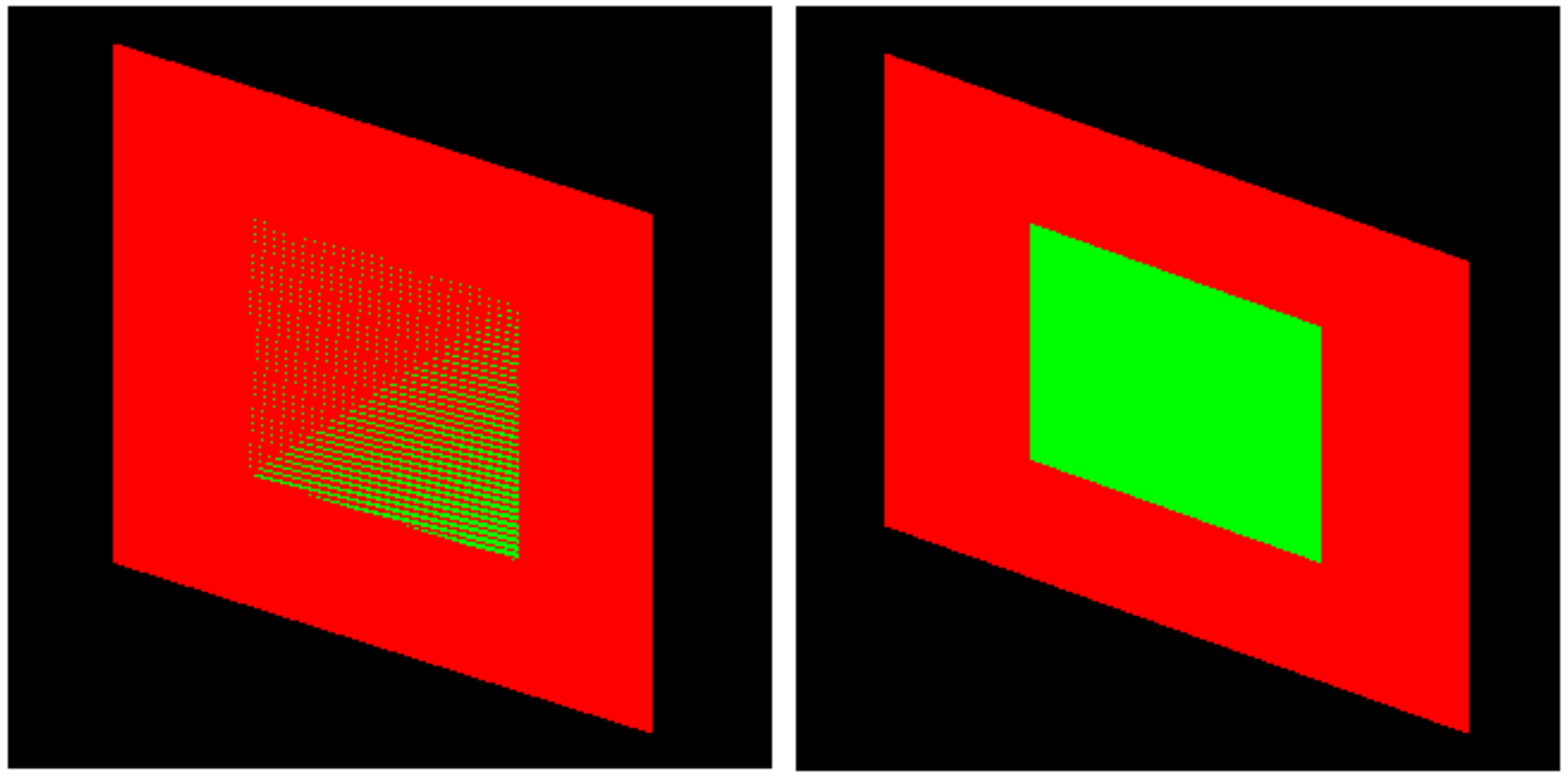
- You render the image both in the Image and in the depth buffer, where you store only the depth
- When a new fragment comes in, you draw it in the image only if it is closer
- This always work and it is cheap to evaluate! It is the default in all graphics hardware
- You still have to sort for transparency…

z-buffer quantization and “z-fighting”

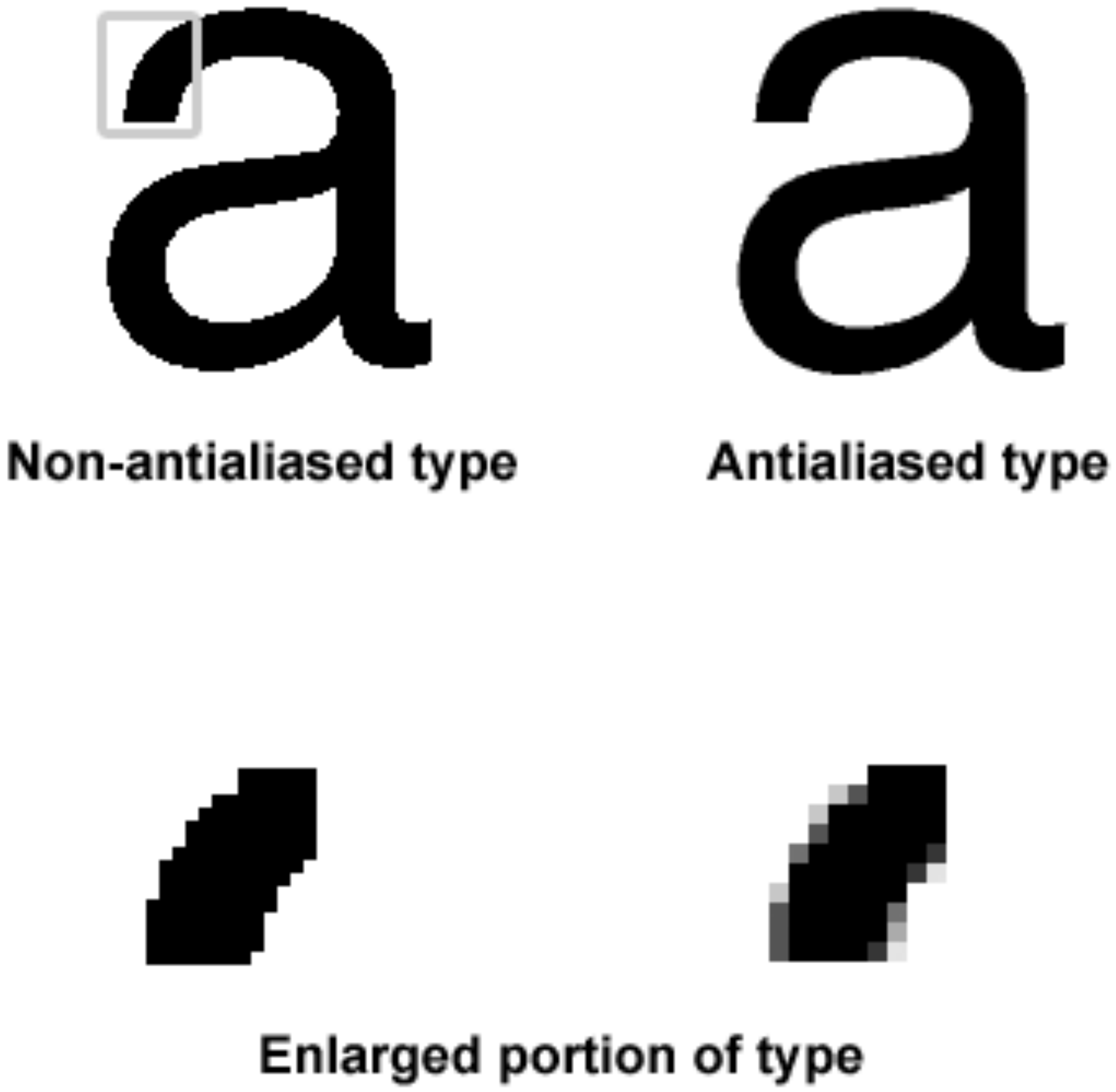
- The z-buffer is quantized (the number of bits is heavily dependent on the hardware platform)



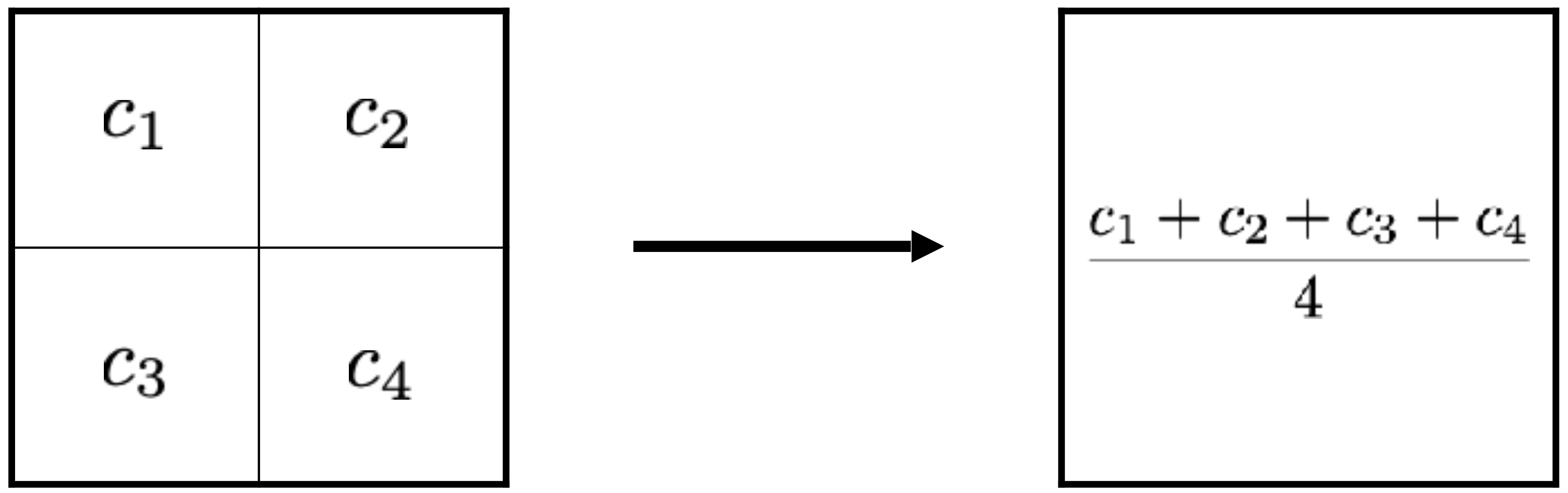
- Two close object might be quantized differently, leading to strange artifacts, usually called “z-fighting”



Super Sampling Anti-Aliasing

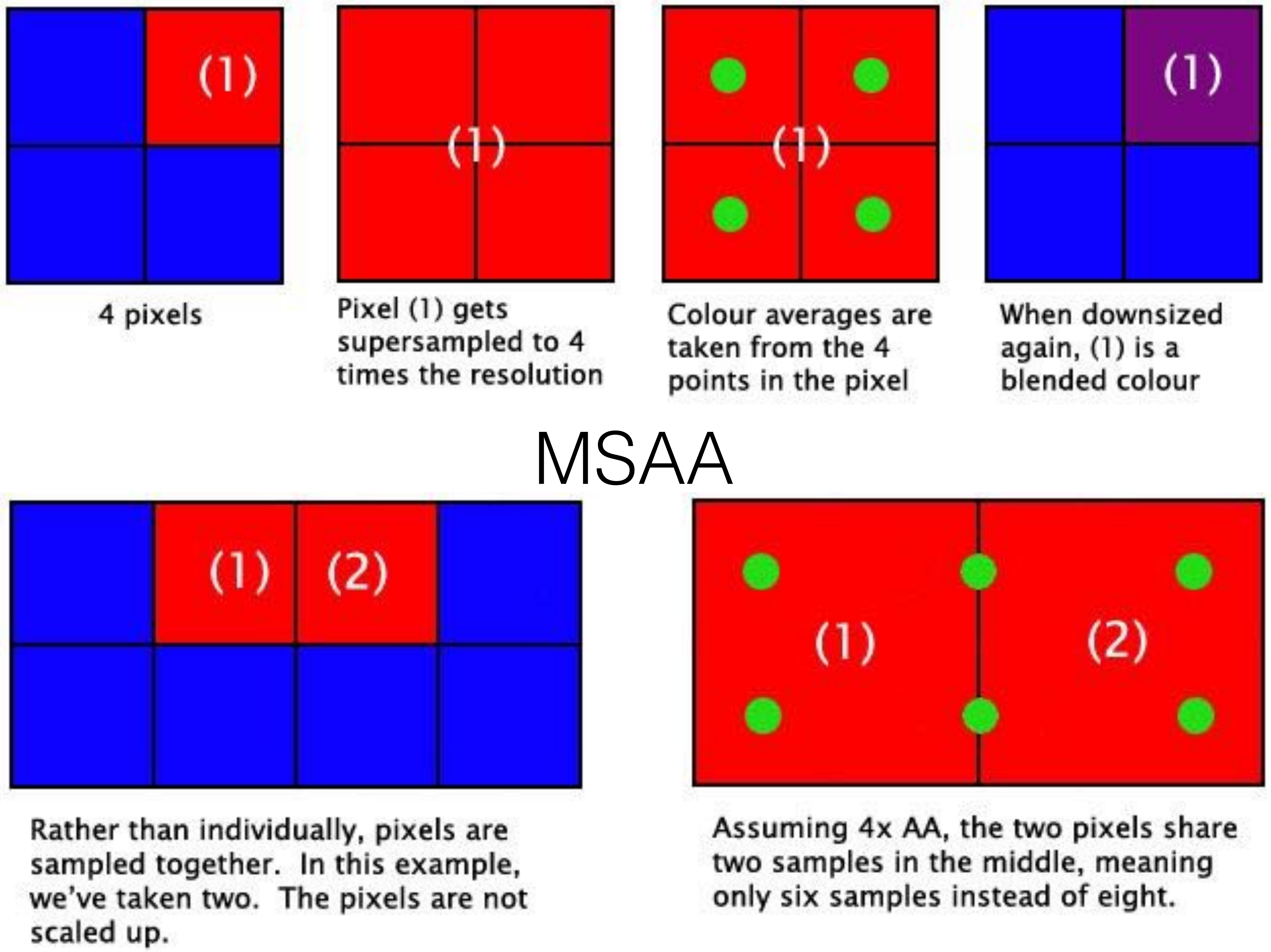


- Render nxn pixels instead of one
- Assign the average to the pixel



Many different names and variants

- SSAA (FSAA)
- MSAA
- CSAA
- EQAA
- FXAA
- TX AA



MSAA

References

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 8