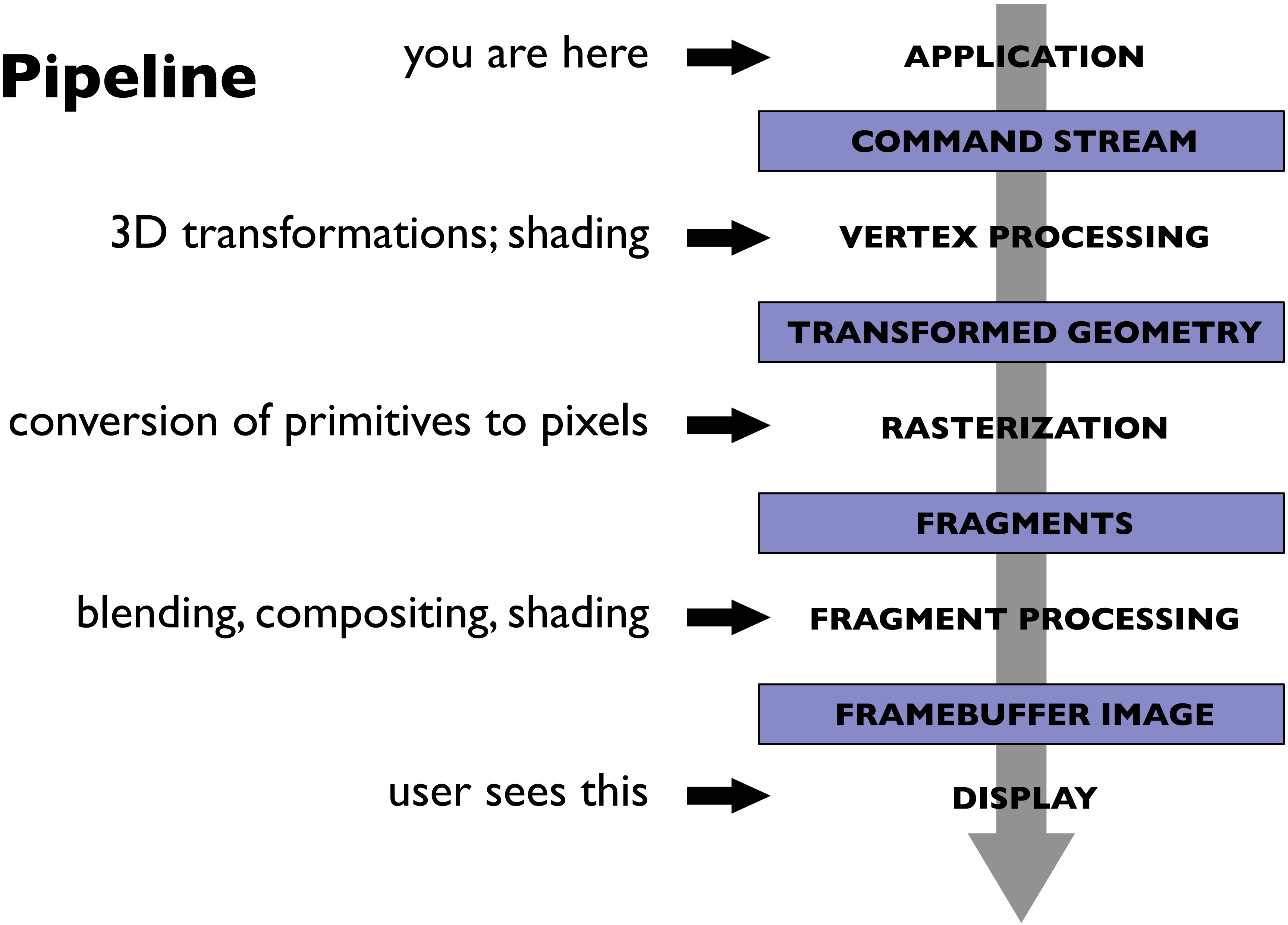
Rasterization

CS4620 Lecture 13

The graphics pipeline

- The standard approach to object-order graphics
- Many versions exist
  – software, e.g. Pixar's REYES architecture
    - many options for quality and flexibility
  – hardware, e.g. graphics cards in PCs
    - 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)

Pipeline


you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
FRAMEBUFFER IMAGE
user sees this
DISPLAY

Primitives

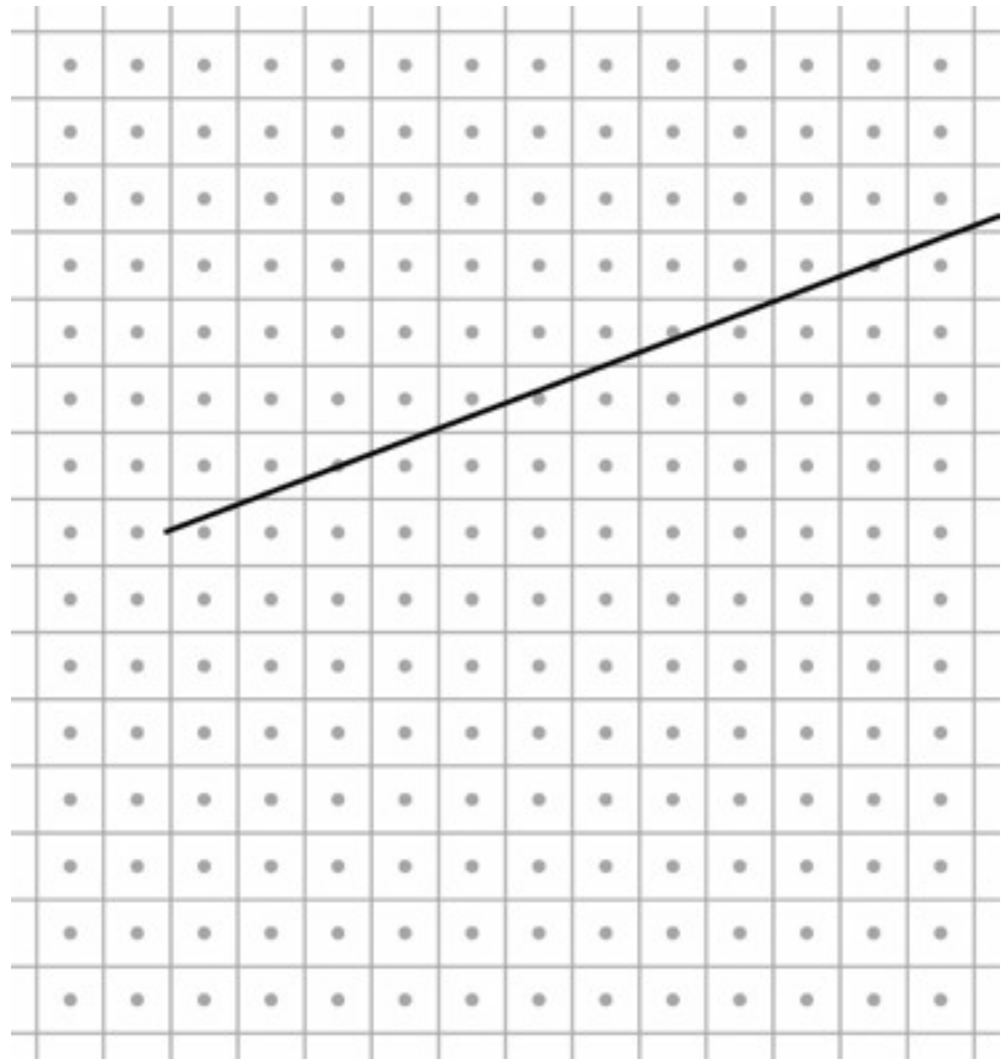
- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved regions? Approximate them with triangles
- Trend has been toward minimal primitives
  - simple,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

Rasterization

- First job: enumerate the pixels covered by a primitive
  - simple, aliased definition: pixels whose centers fall inside
- Second job: interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - will see applications later on

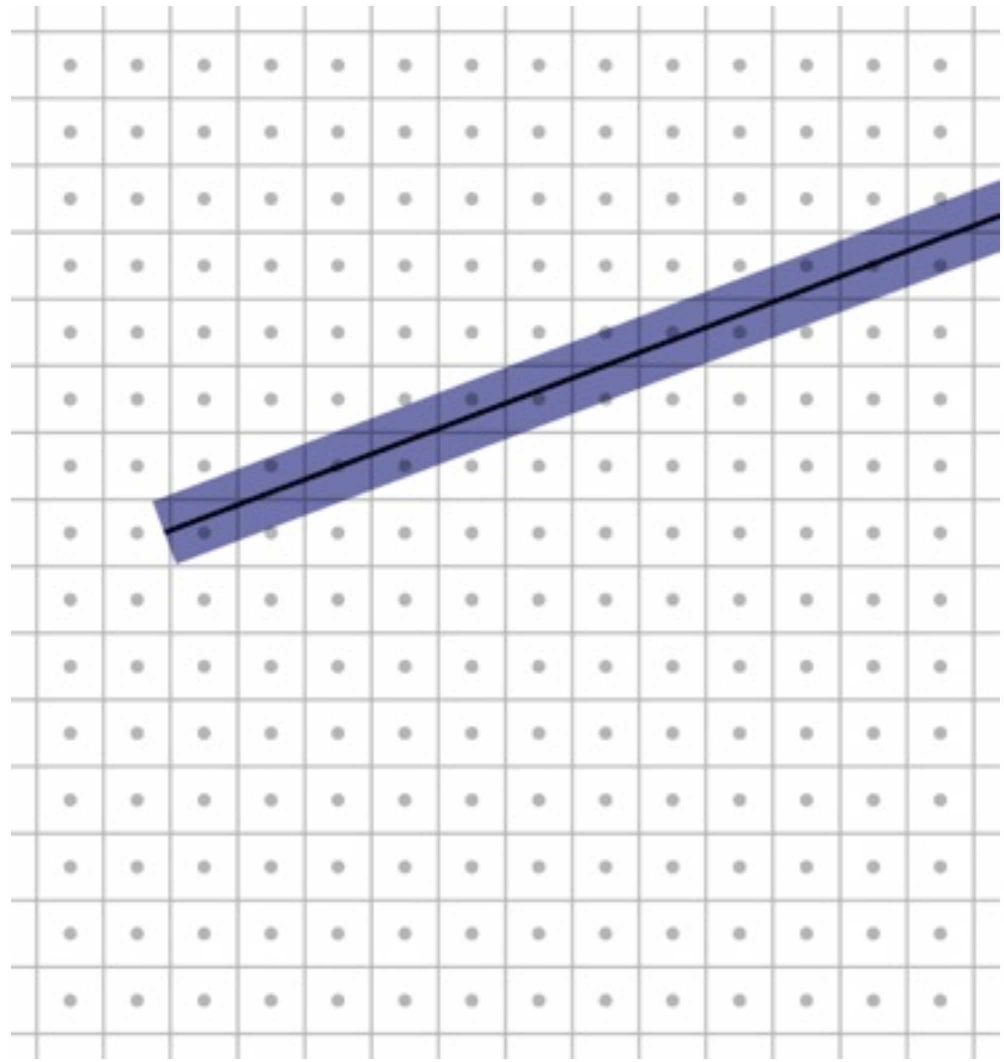
Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside



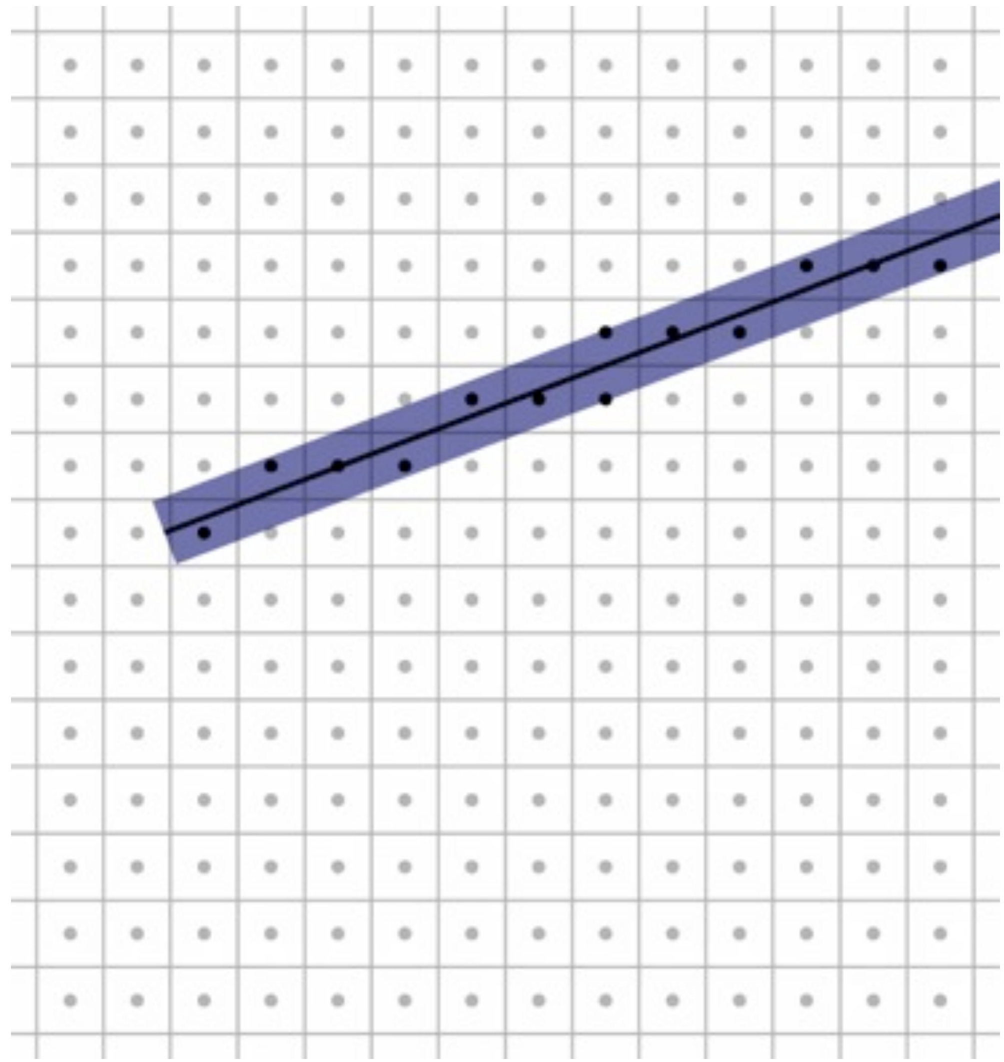
Rasterizing lines

- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside



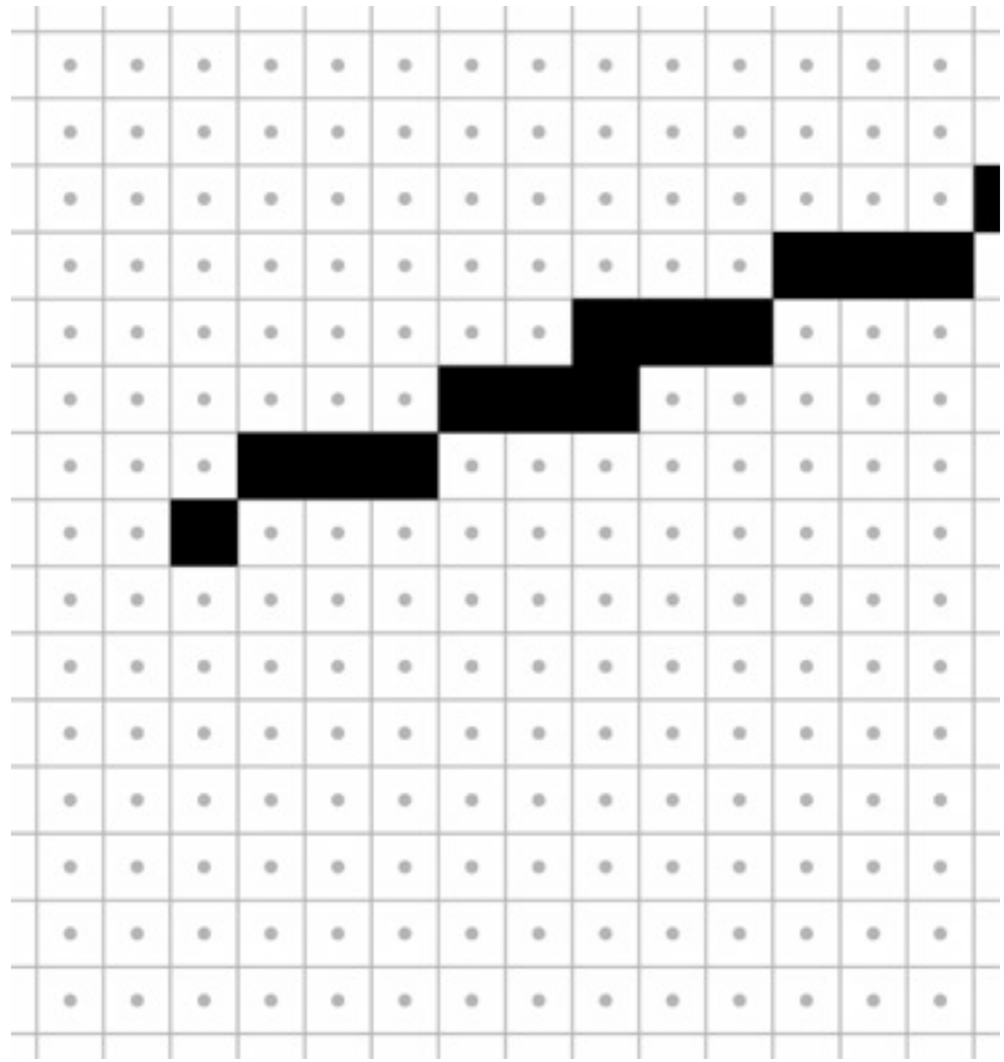
Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

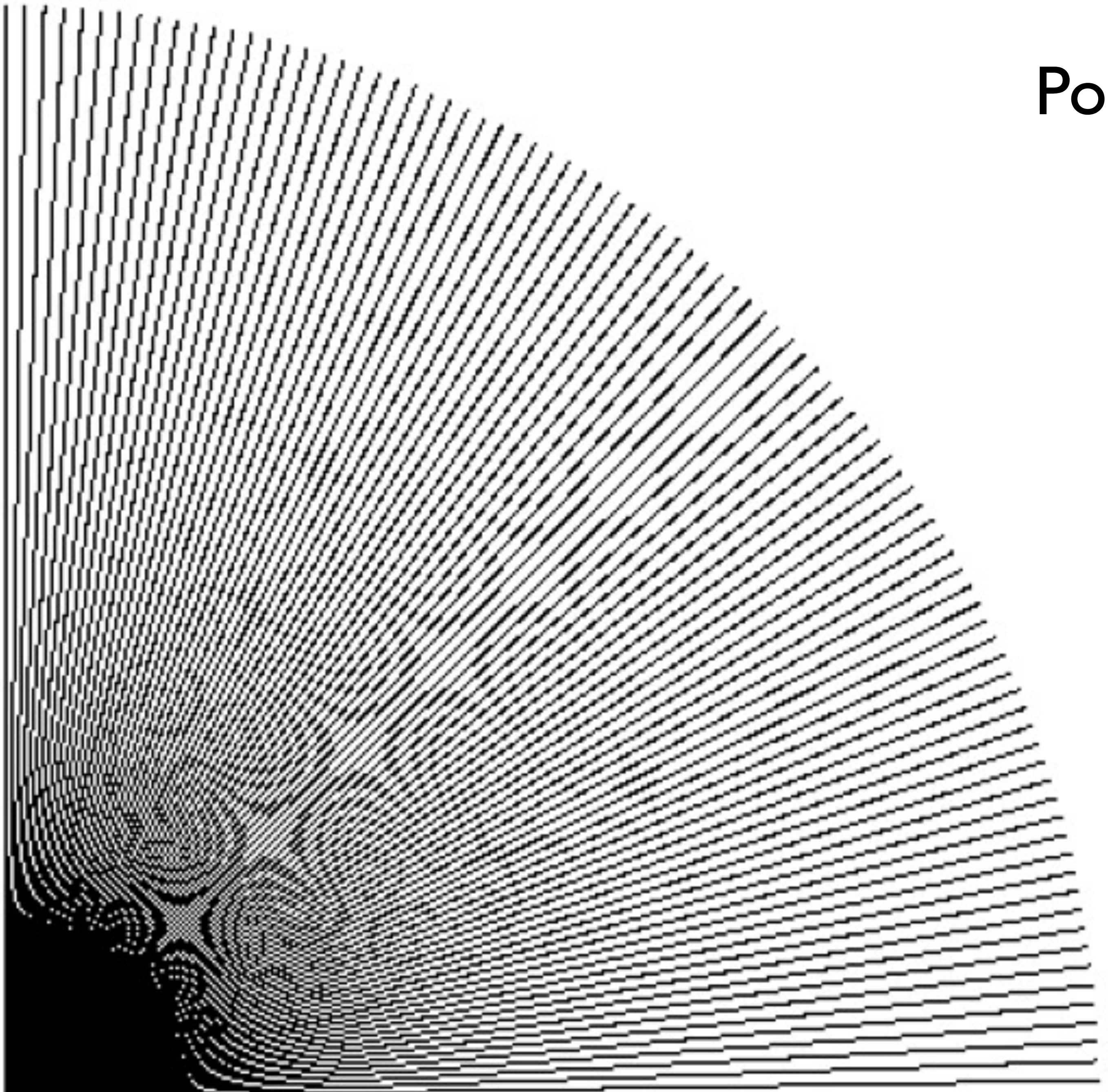


Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

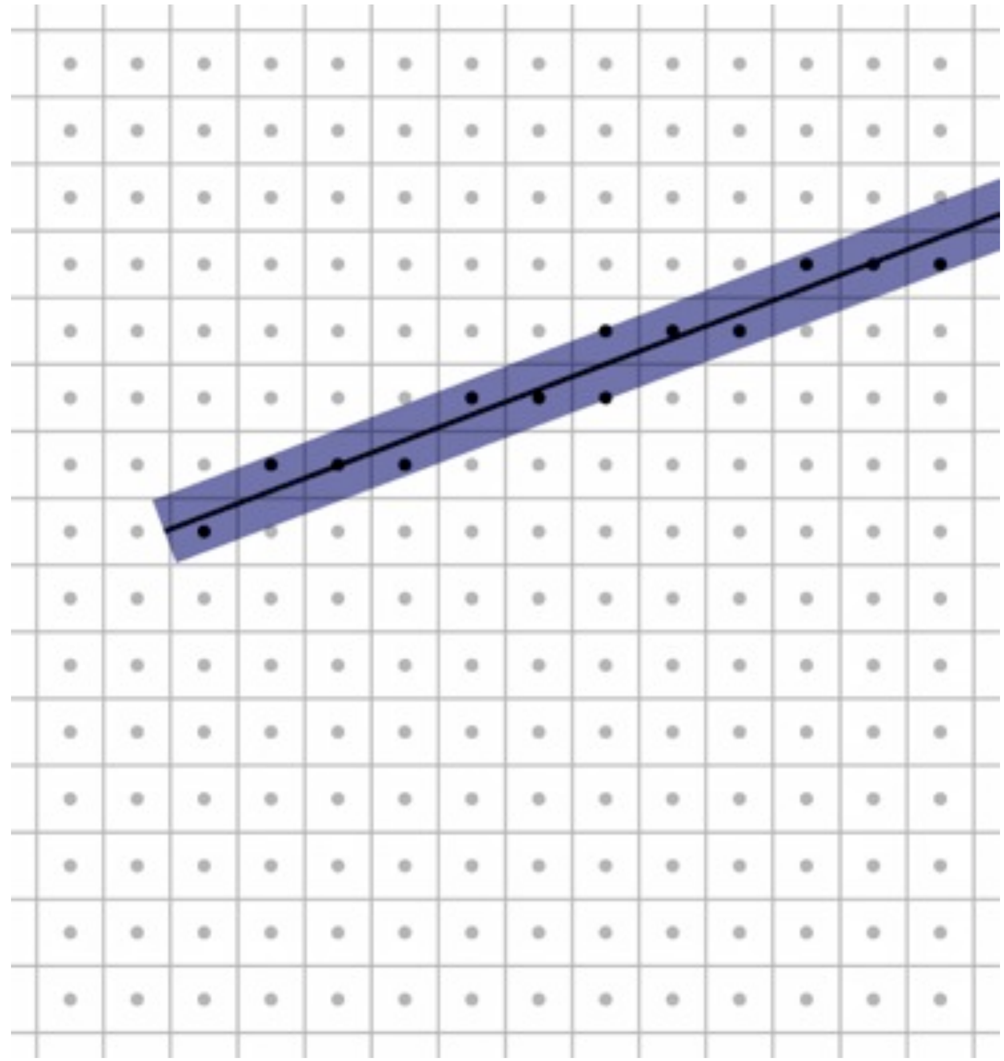


Point sampling in action



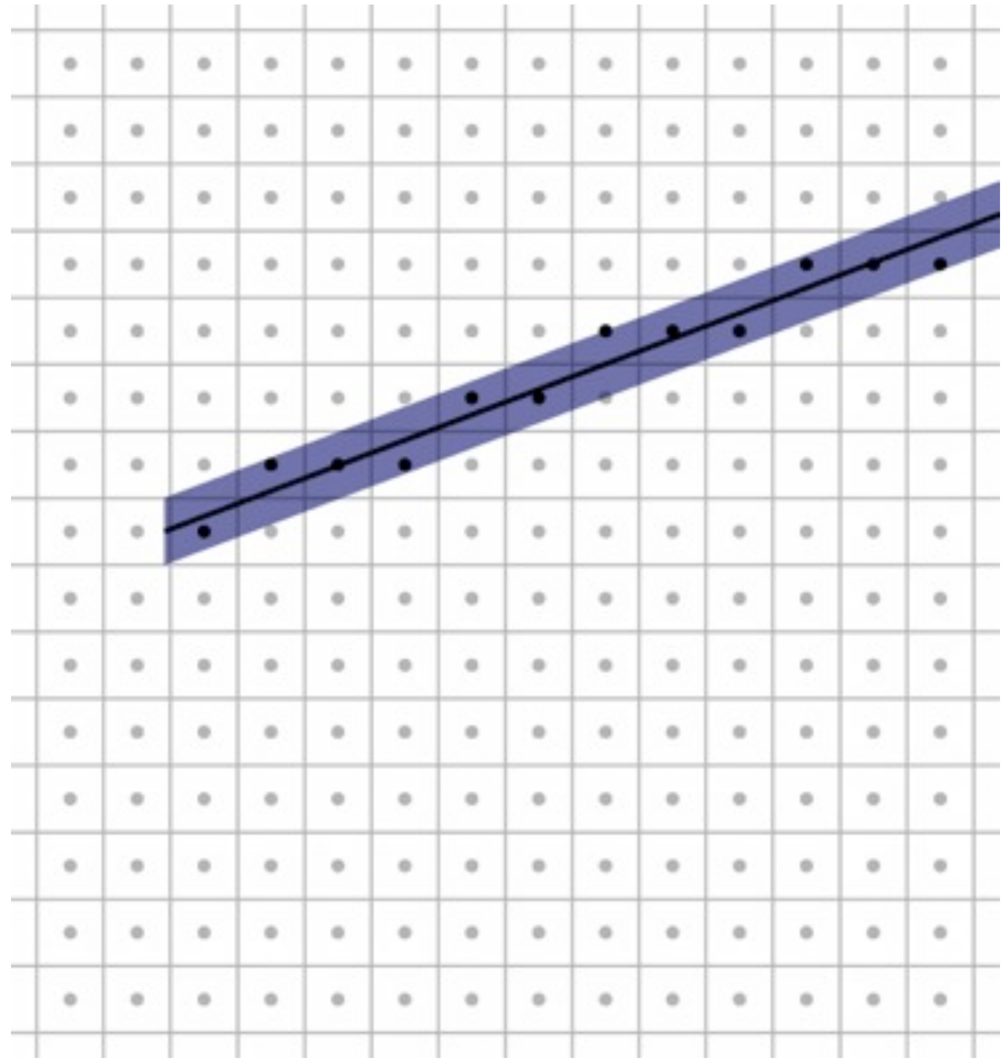
Bresenham lines (midpoint alg.)

- 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° lines are now thinner



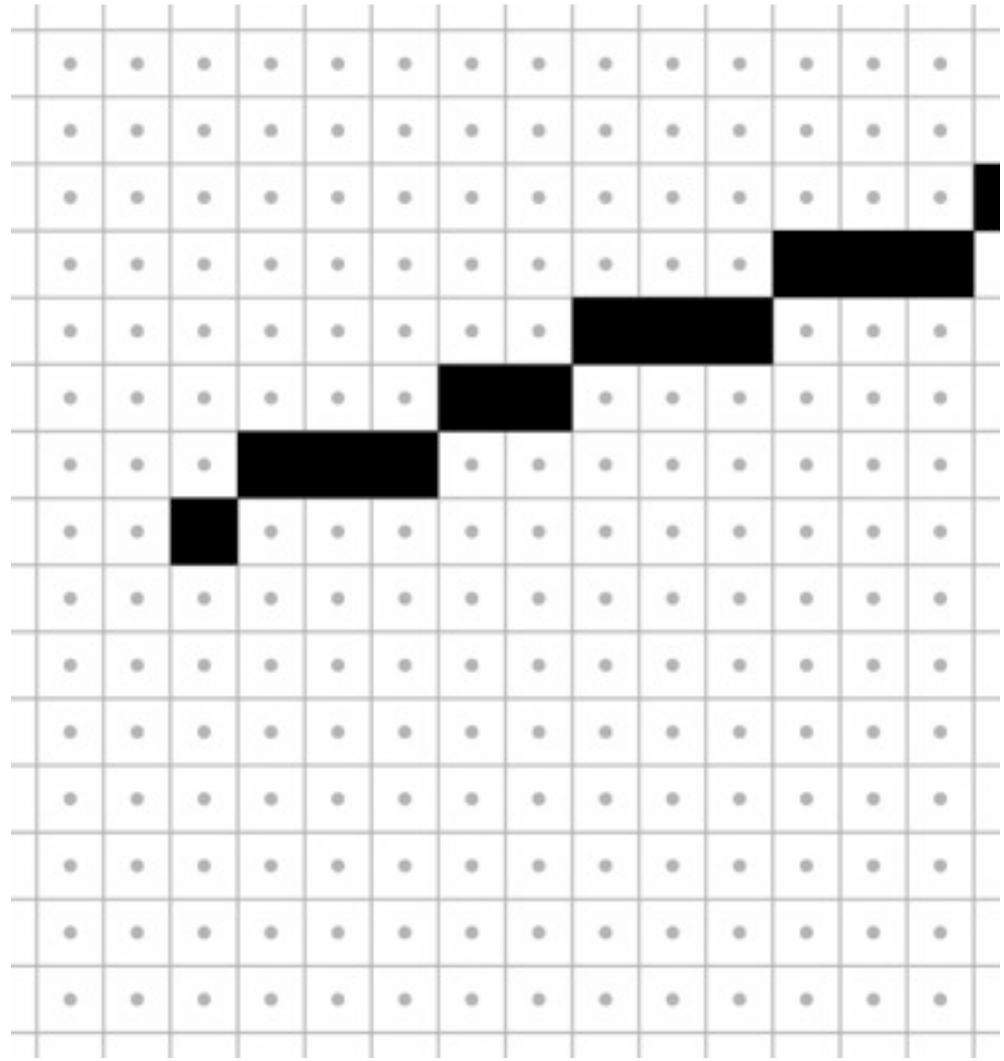
Bresenham lines (midpoint alg.)

- 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° lines are now thinner

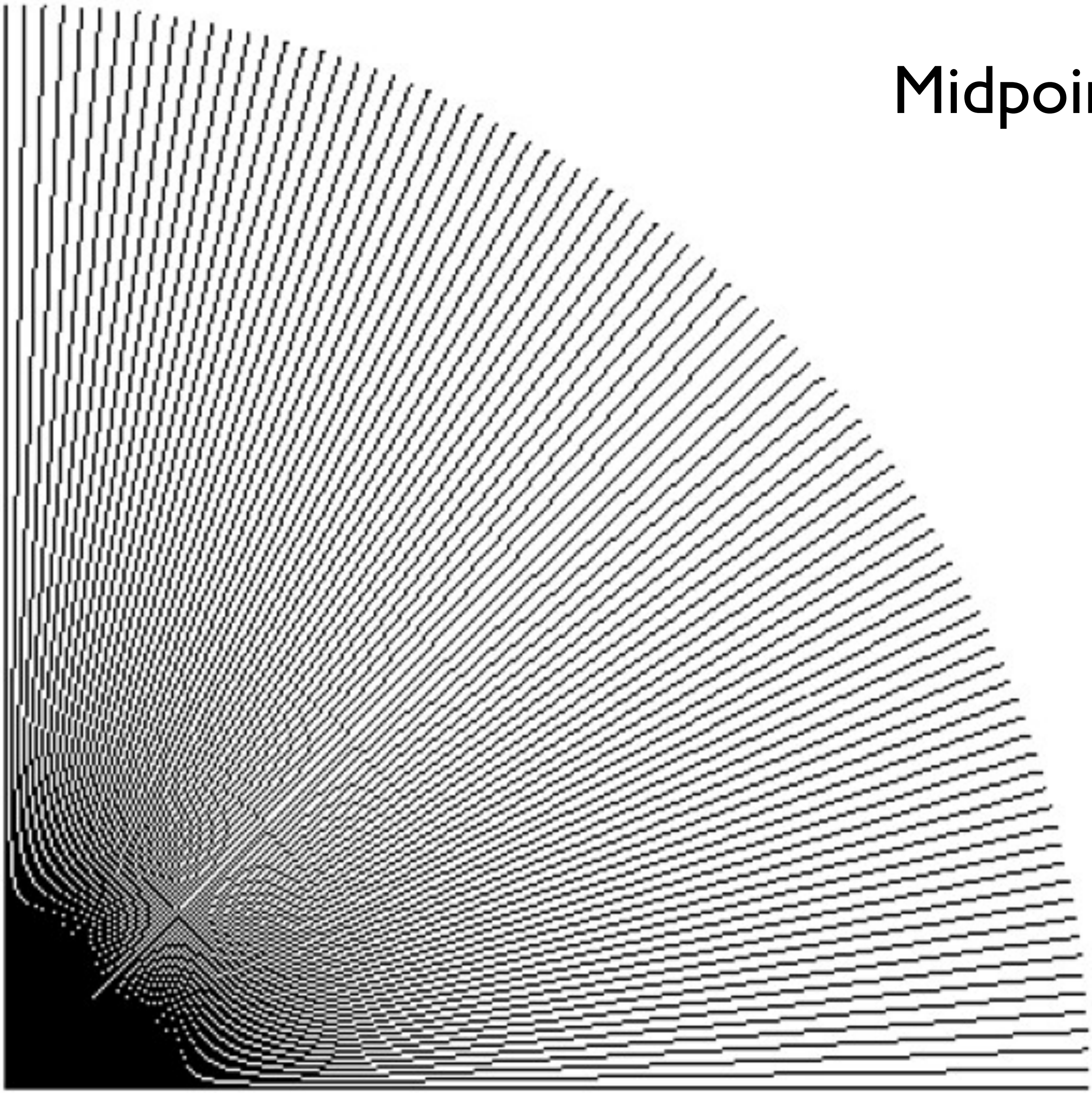


Bresenham lines (midpoint alg.)

- 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° lines are now thinner



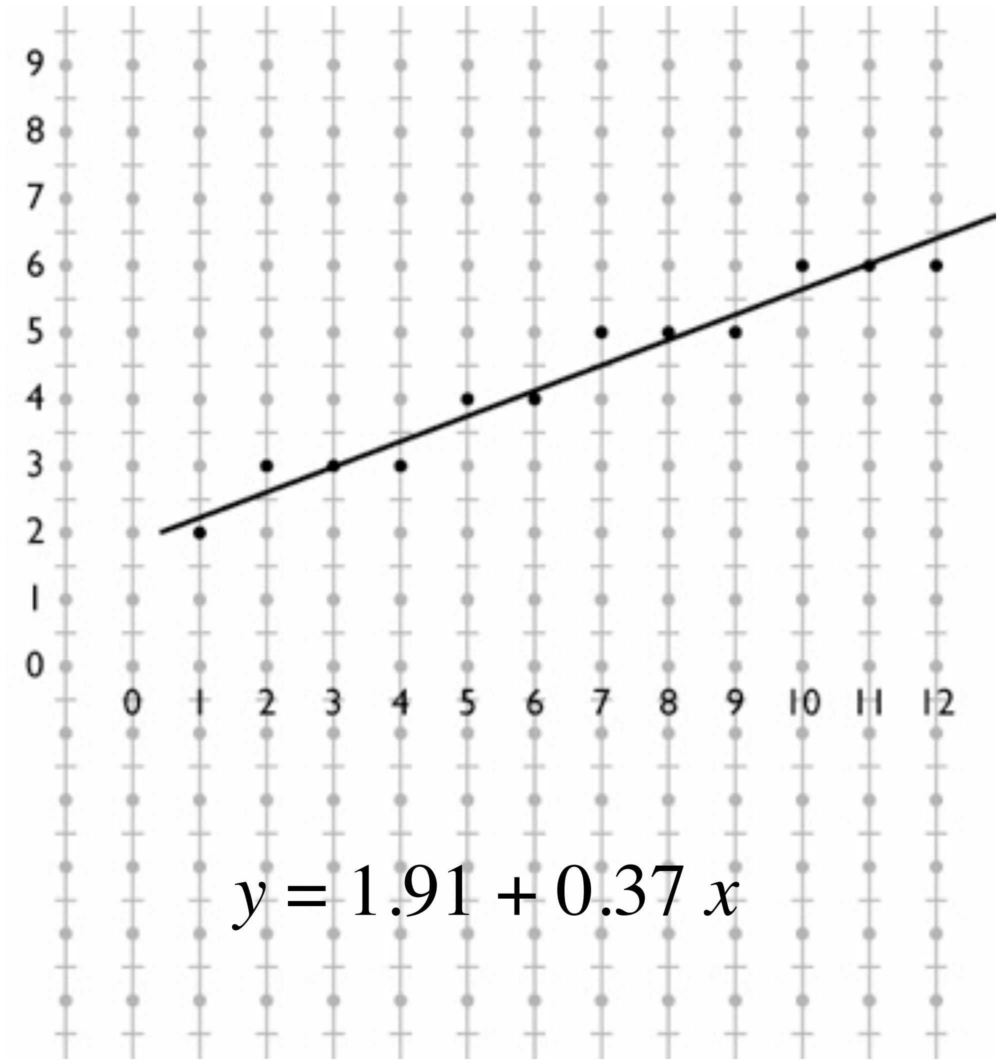
Midpoint algorithm in action



Algorithms for drawing lines

- line equation:
  $y = b + m\,x$
- Simple algorithm: evaluate line equation per column
- W.l.o.g. $x_0 < x_1$;
  $0 \leq m \leq 1$

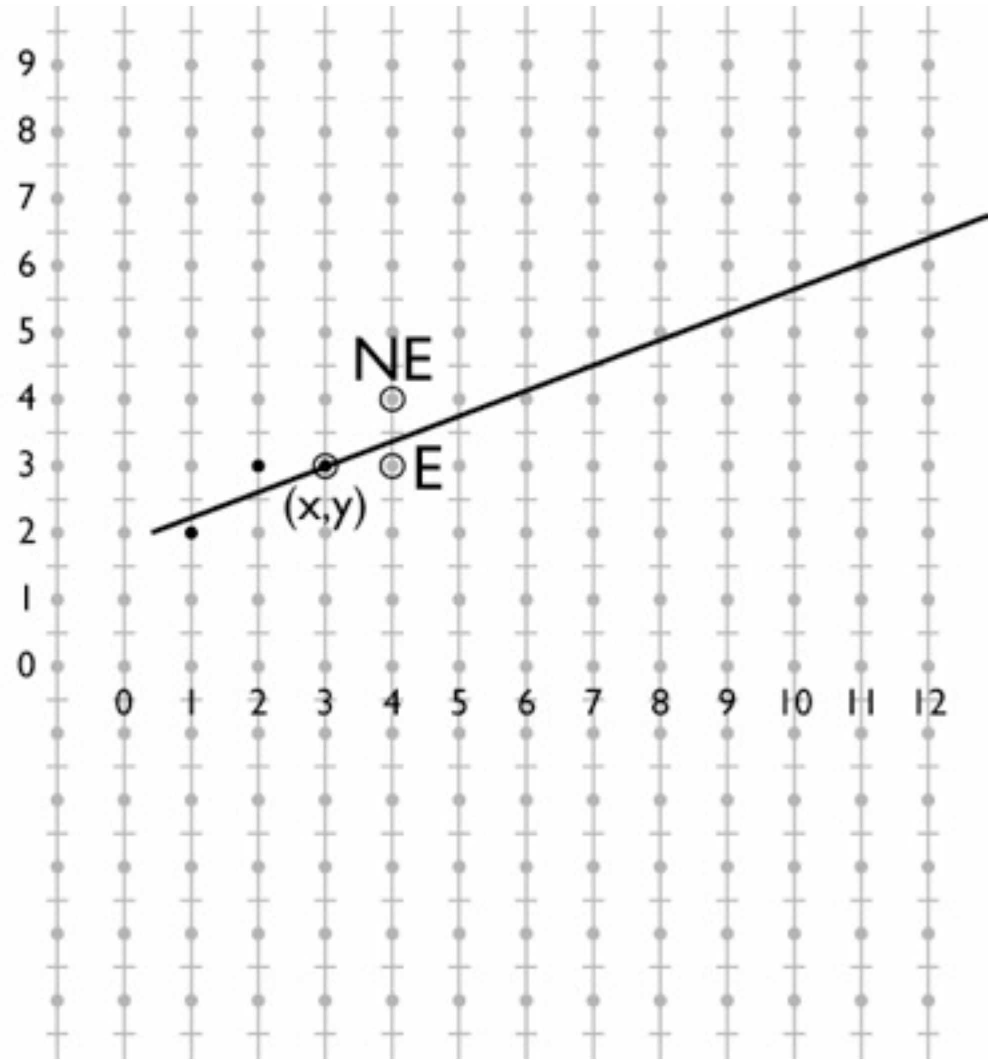
```
for x = ceil(x0) to floor(x1)
    y = b + m*x
    output(x, round(y))
```



$y = 1.91 + 0.37\,x$

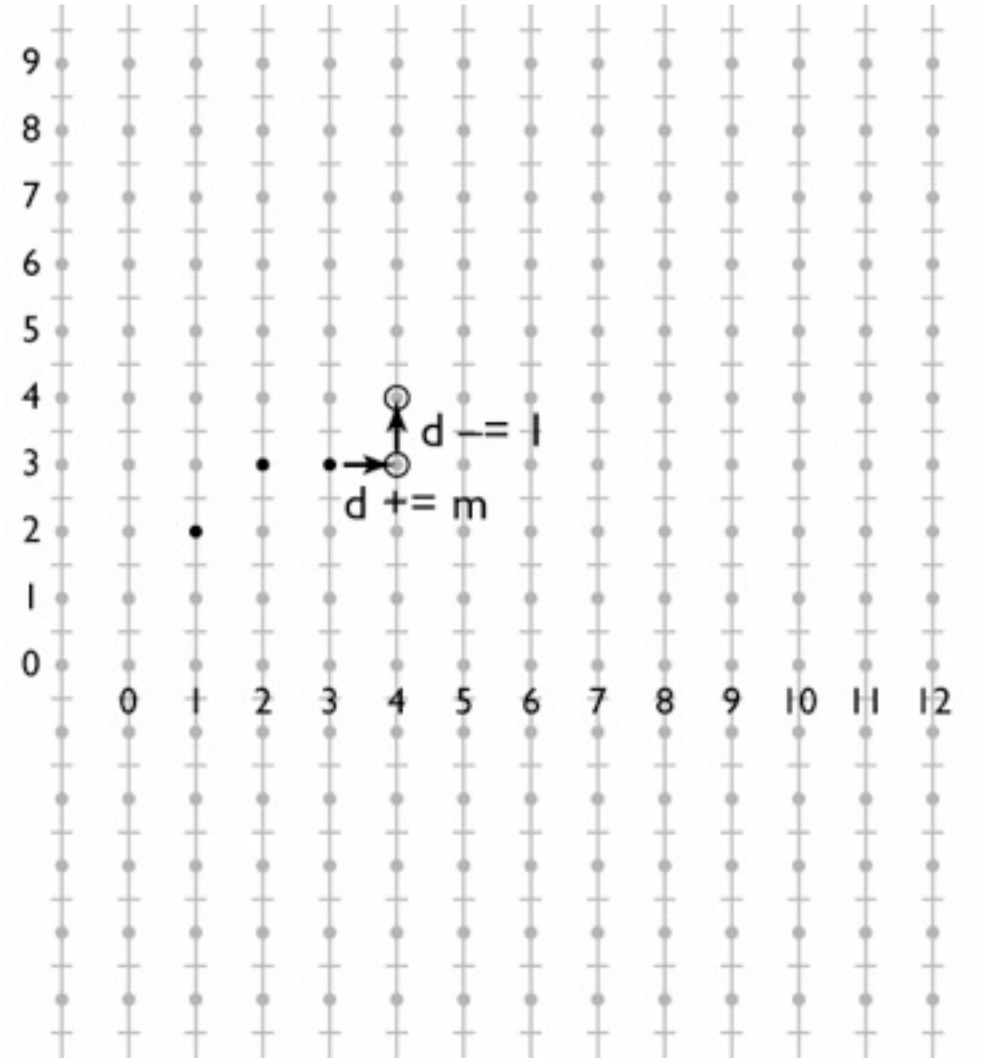
Optimizing line drawing

- 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



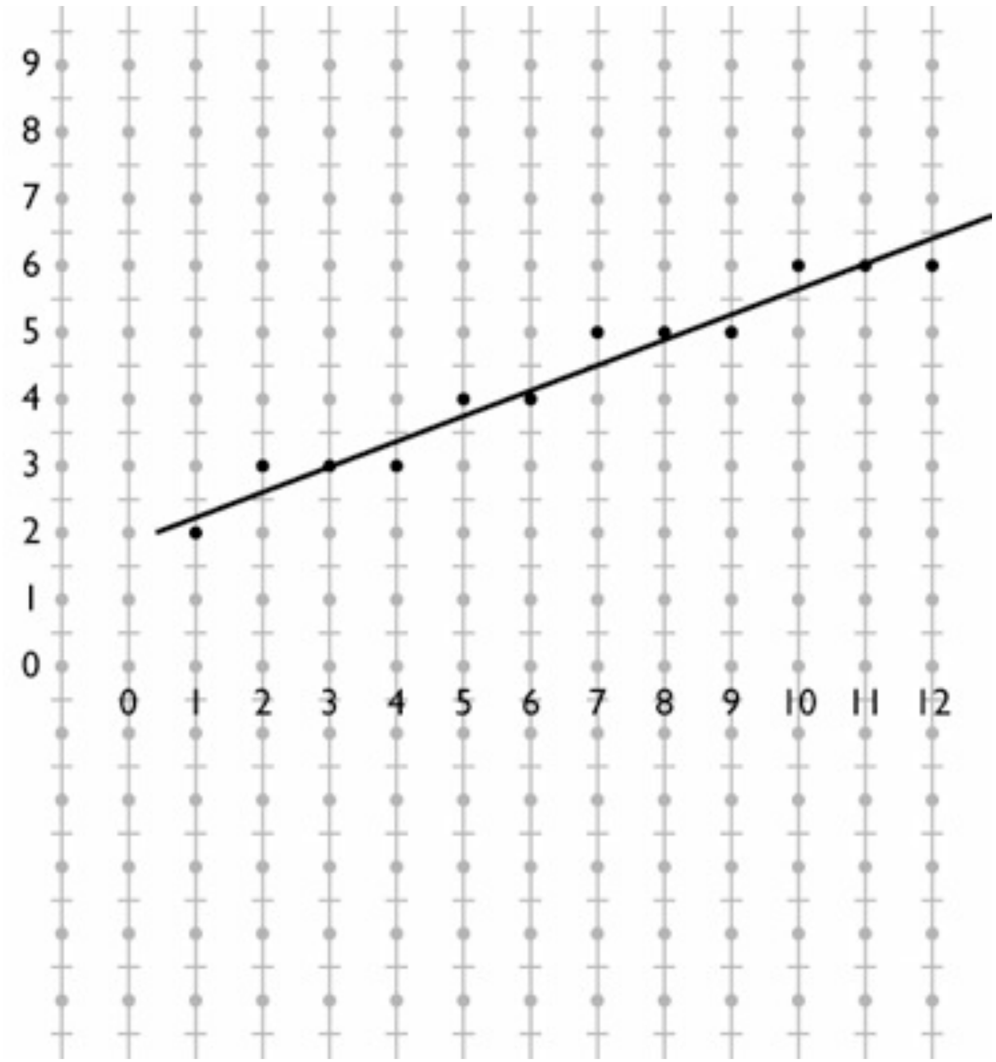
Optimizing line drawing

- $d = m(x + 1) + b - y$
- Only need to update d for integer steps in x and y
- Do that with addition
- Known as "DDA" (digital differential analyzer)



Midpoint 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)
```

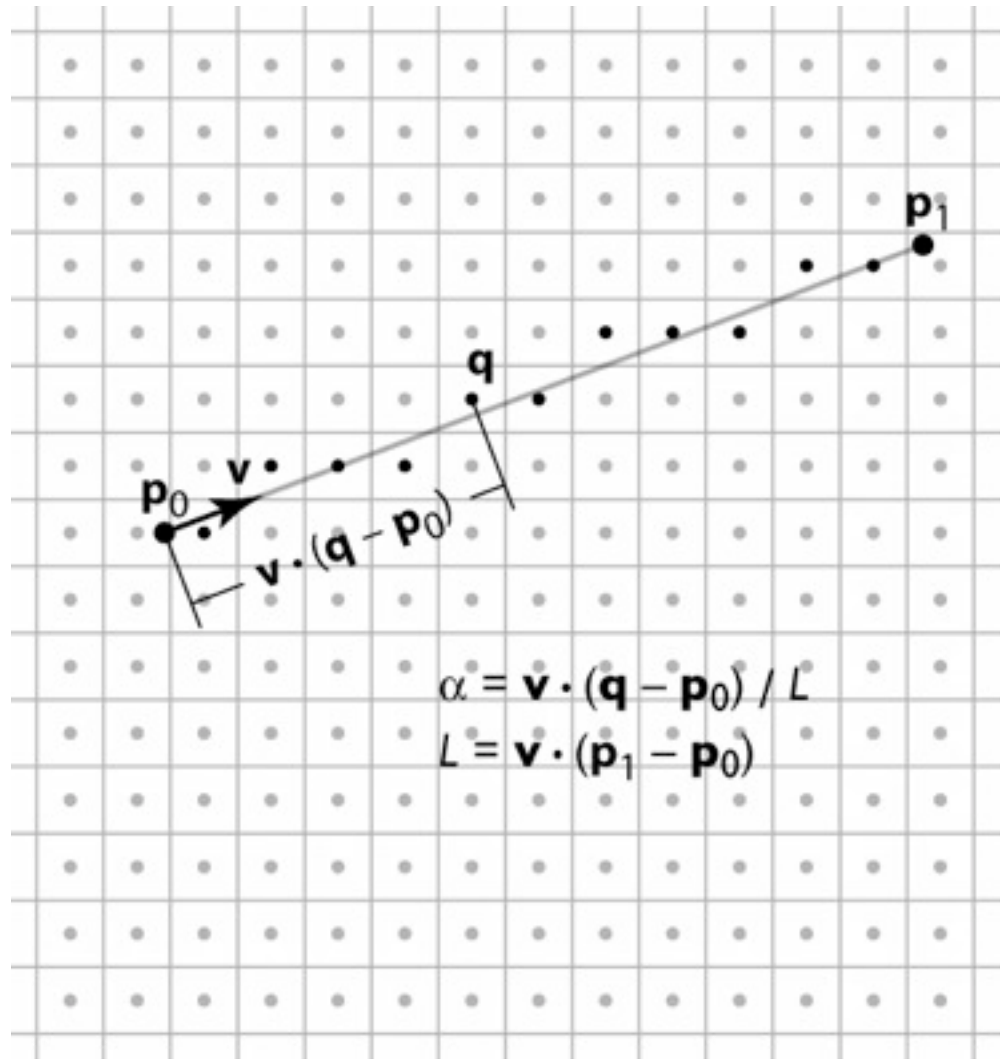


Linear interpolation

- 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)

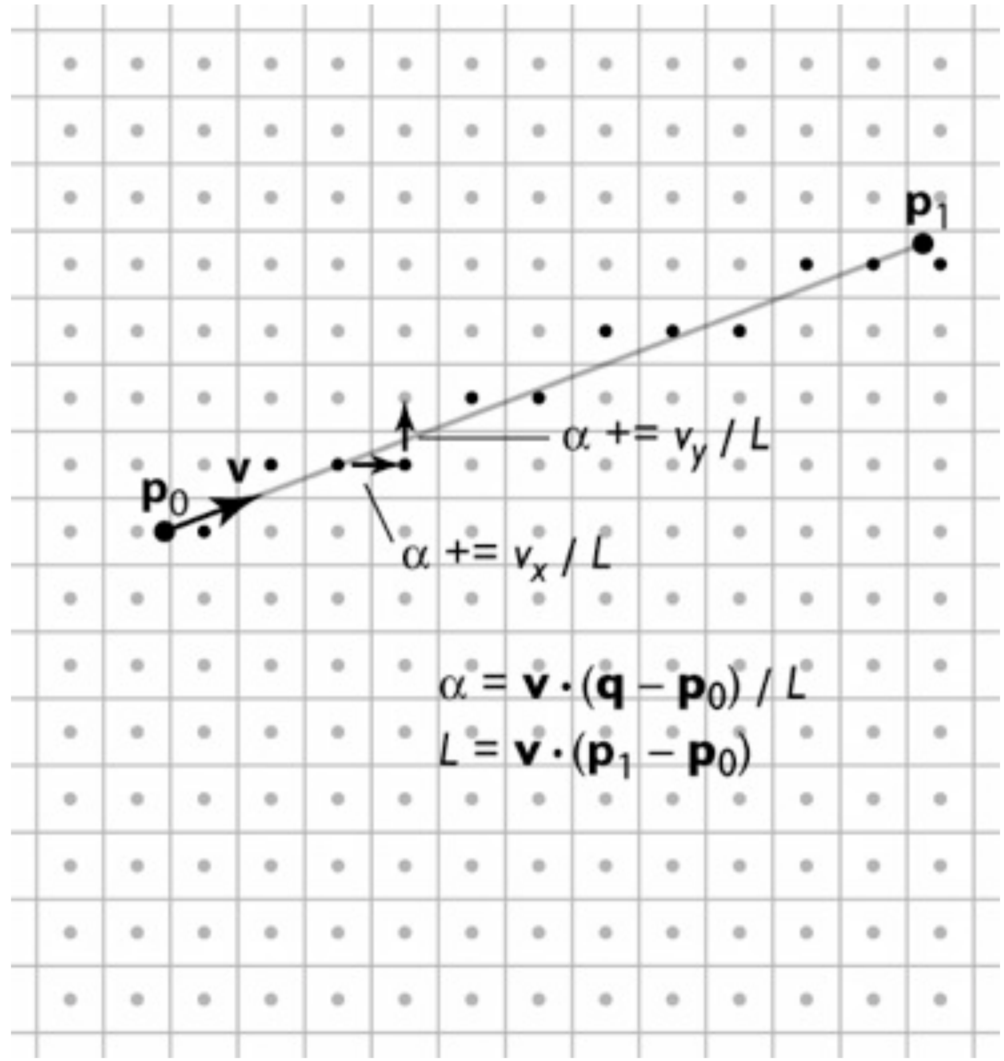
Linear interpolation

- 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



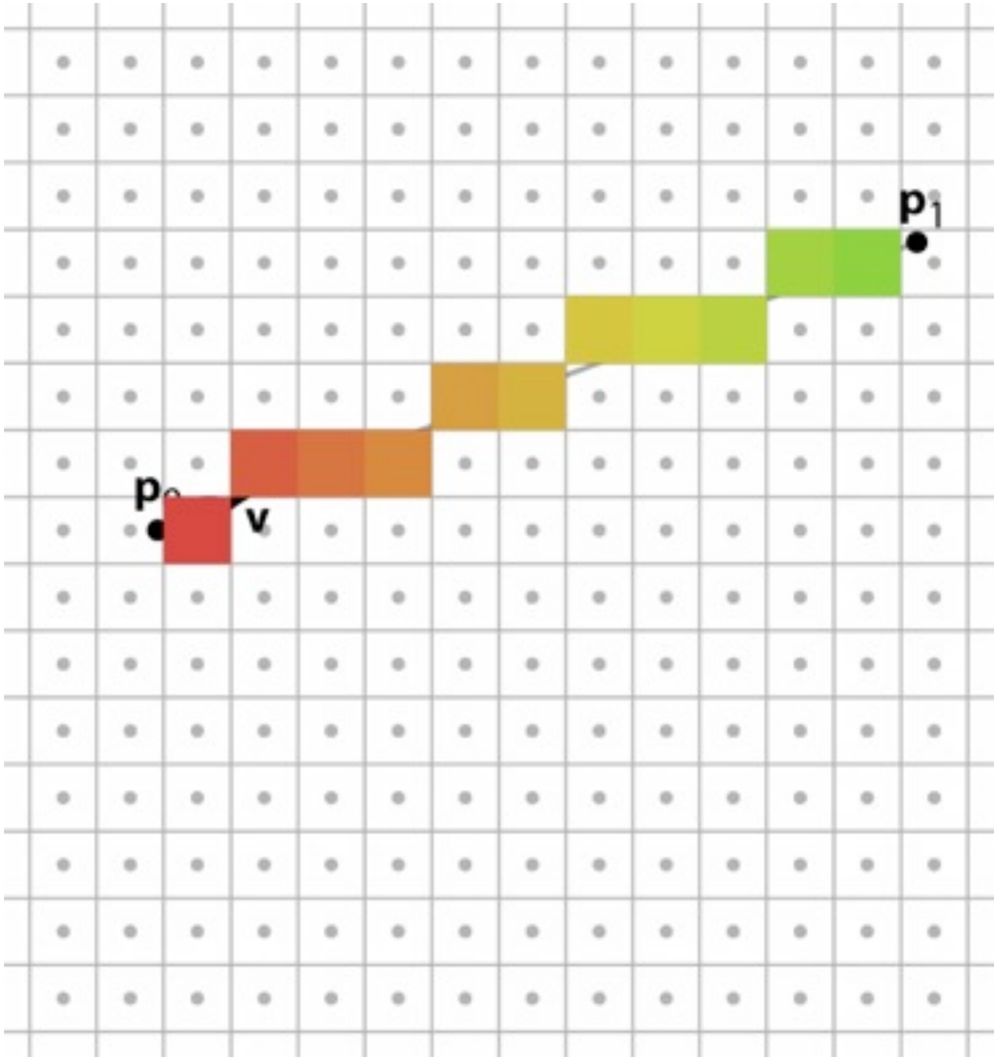
Linear interpolation

- 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



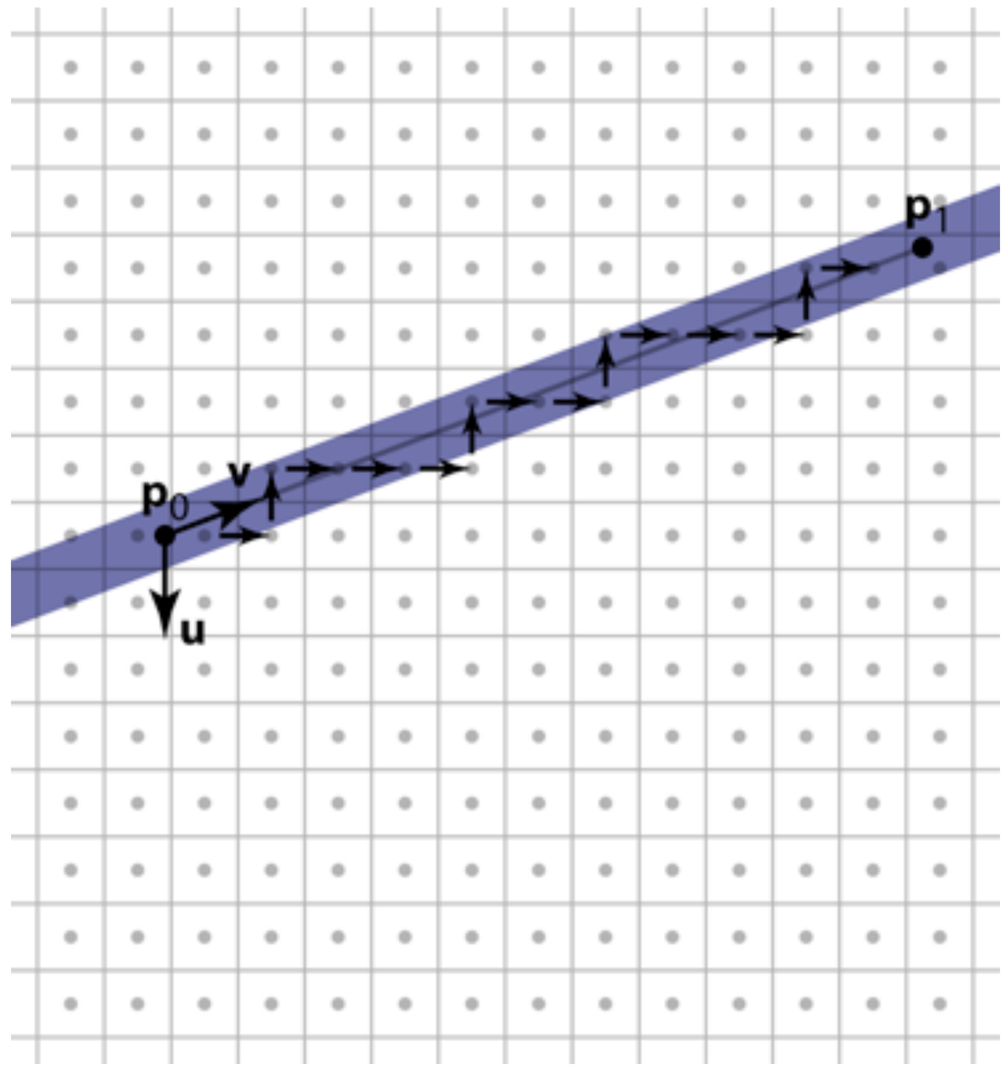
Alternate interpretation

- 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

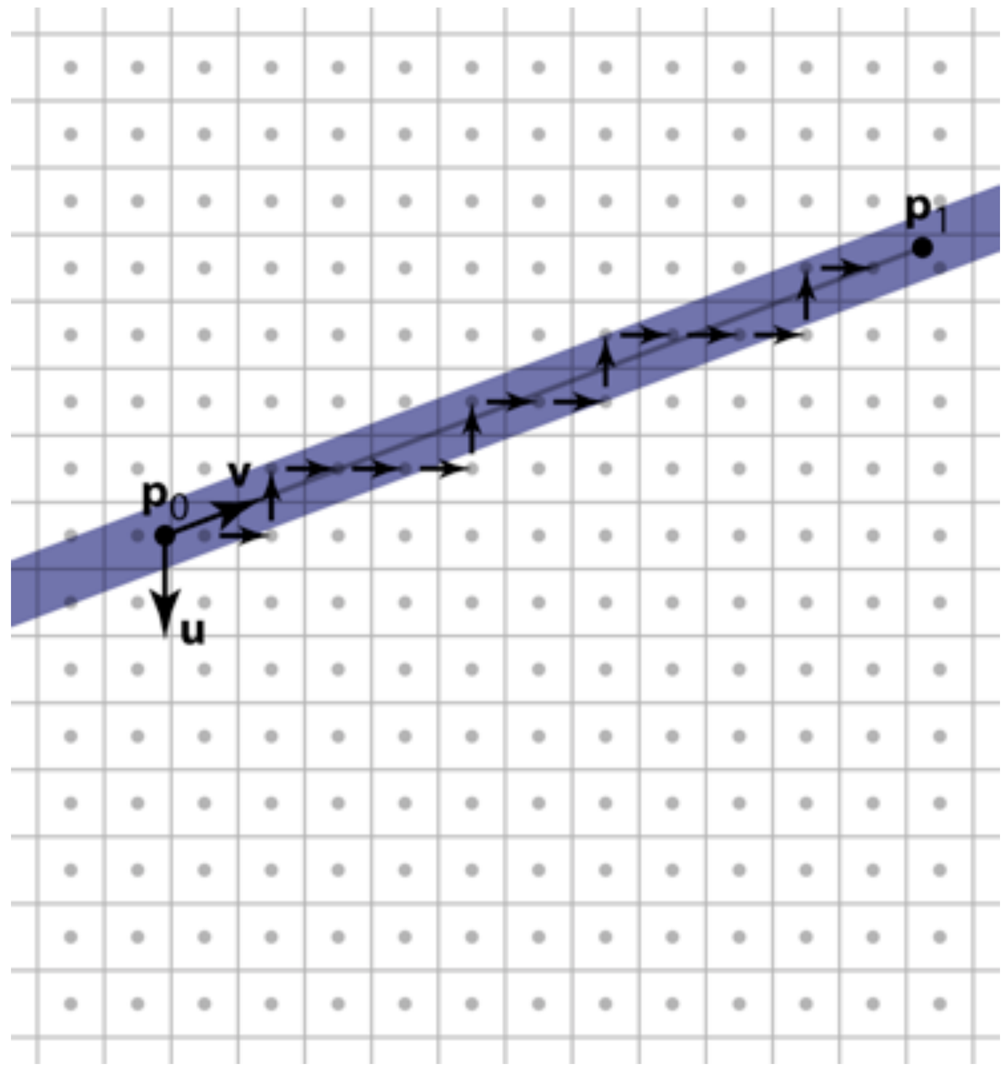
Alternate interpretation

- View loop as visiting all pixels the line passes through
  - Interpolate d and α for each pixel
  - Only output frag. if pixel is in band
- This makes linear interpolation the primary operation



Pixel-walk line rasterization

```
x = ceil(x0)
y = round(m*x + b)
d = m*x + b − y
while x < floor(x1)
    if d > 0.5
        y += 1; d −= 1;
    else
        x += 1; d += m;
    if −0.5 < d ≤ 0.5
        output(x, y)
```



Rasterizing triangles

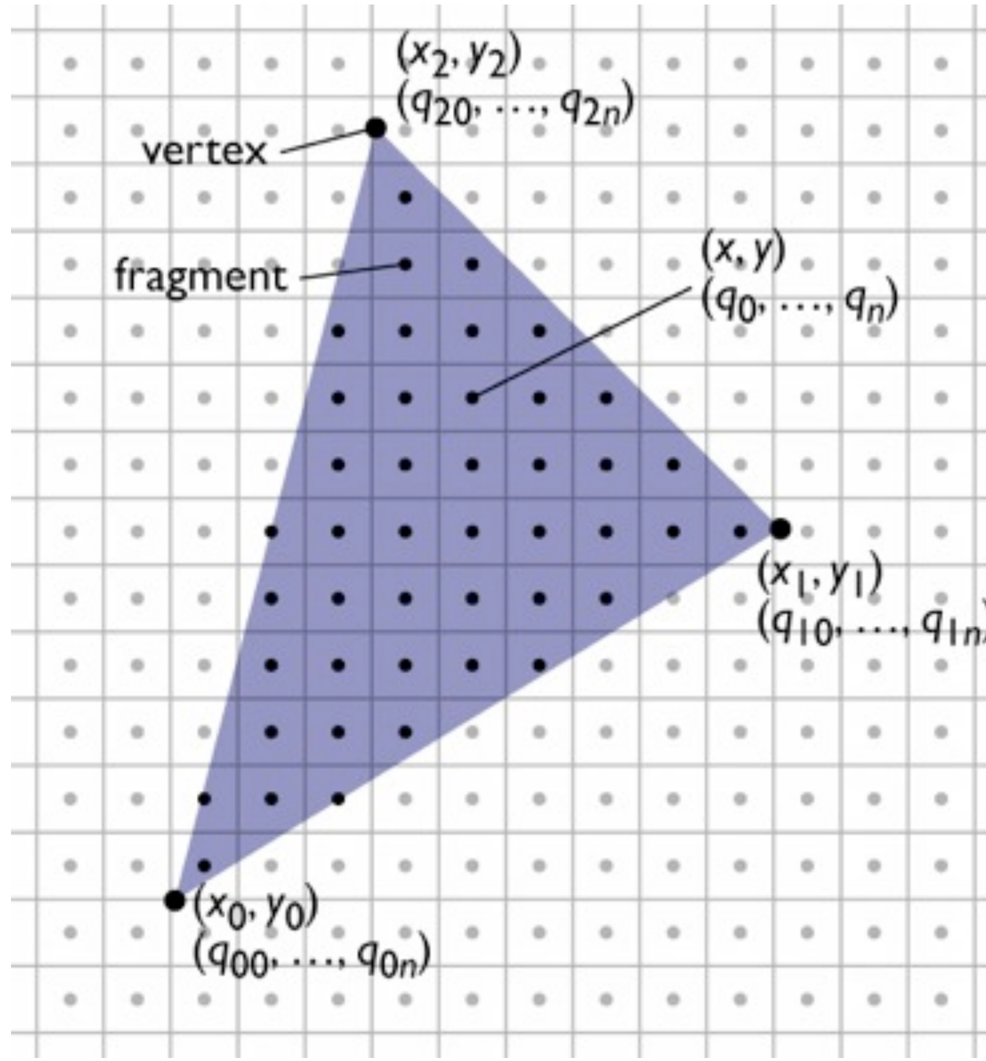
- The most common case in most applications
  - with good antialiasing can be the only case
  - some systems render a line as two skinny triangles
- Triangle represented by three vertices
- Simple way to think of algorithm follows the pixel-walk interpretation of line rasterization
  - walk from pixel 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. functions defined by parameter values at vertices
  3. using extra parameters to determine fragment set



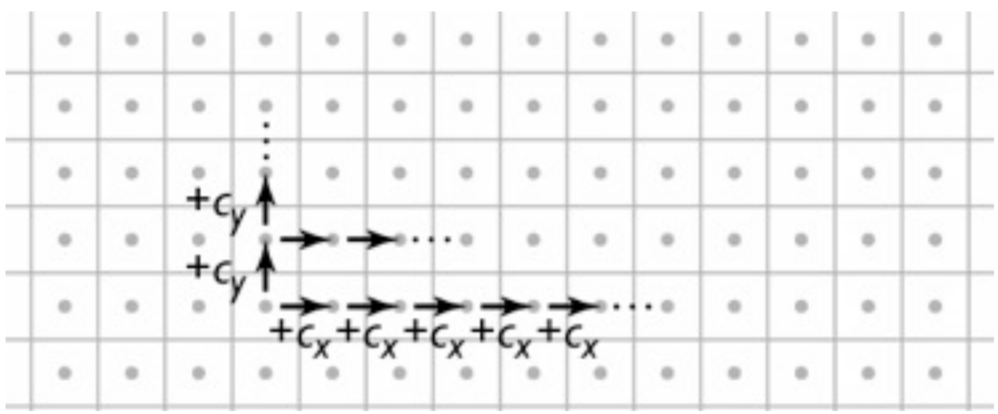
Incremental linear evaluation

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

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

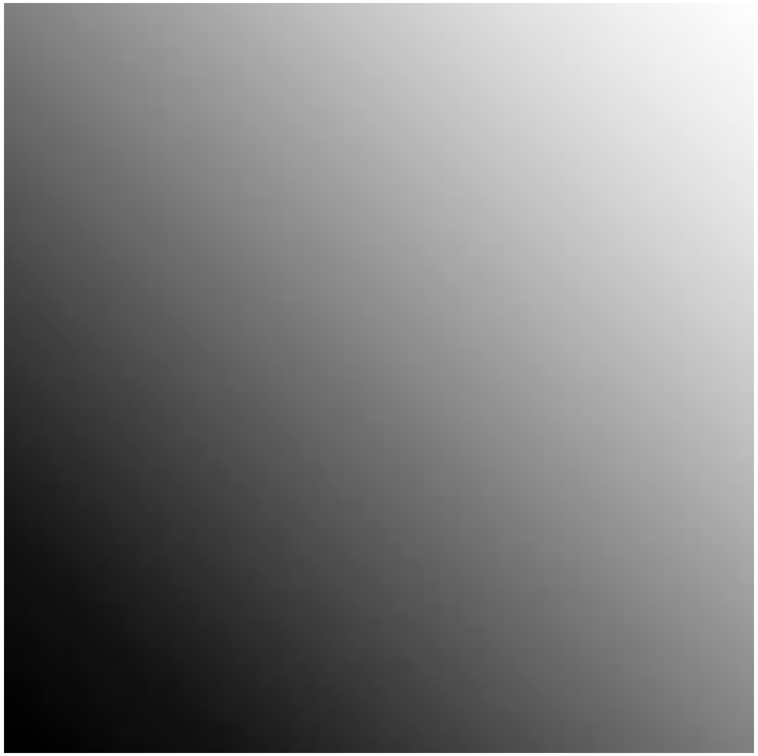
- Linear functions are efficient to evaluate on a grid:

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



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;
  }
}
```

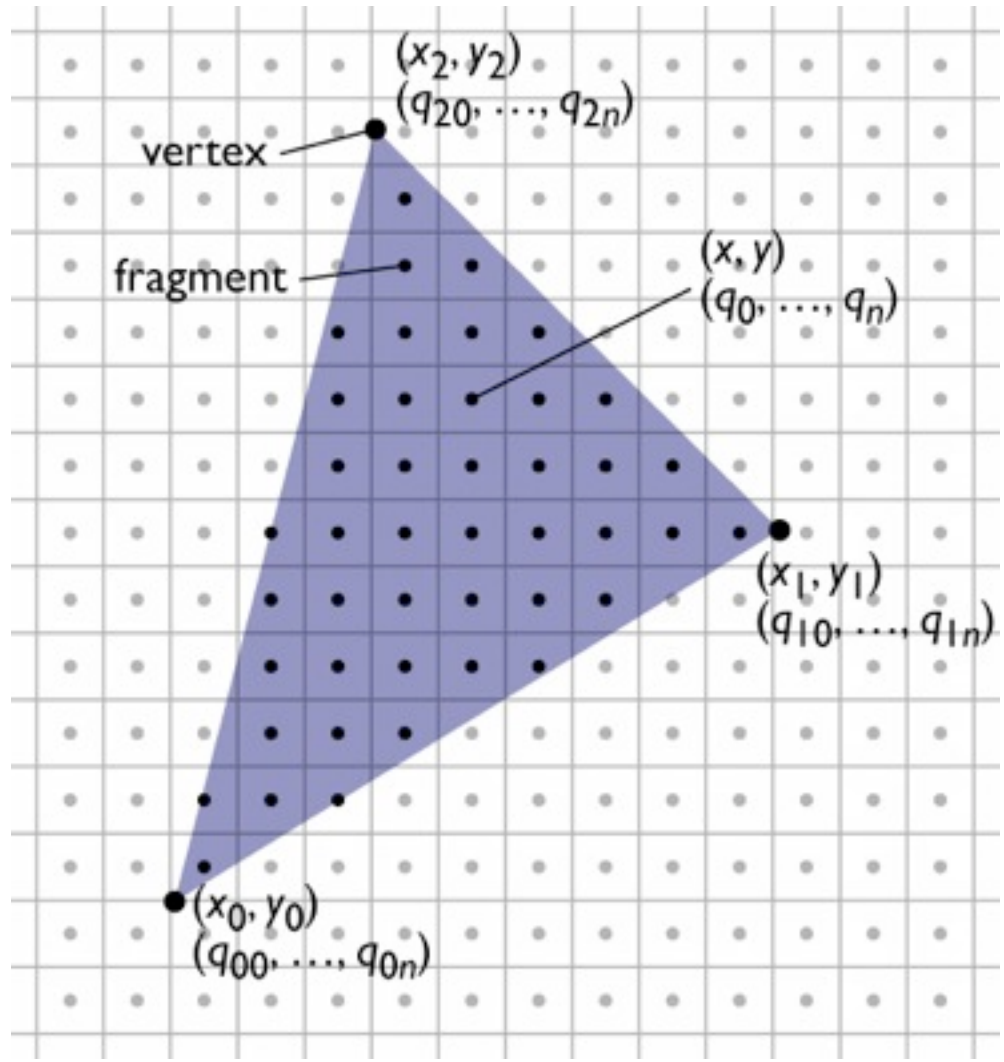


$c_x = .005; c_y = .005; c_k = 0$

(image size 100x100)

Rasterizing triangles

- Summary
  1 evaluation of linear functions on pixel grid
  2 functions defined by parameter values at vertices
  3 using extra parameters to determine fragment set



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):

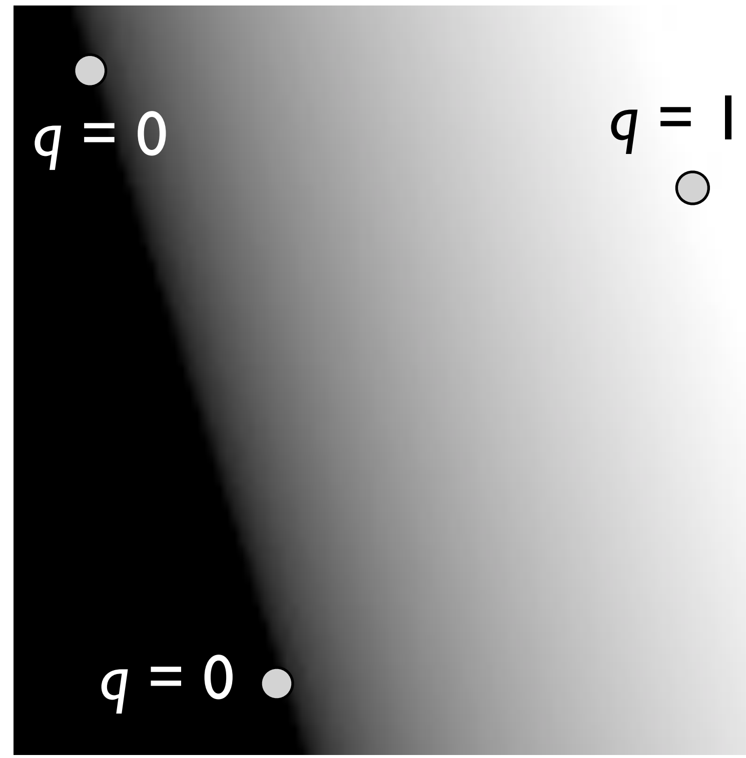
$$\begin{bmatrix} (x_1 - x_0) & (y_1 - y_0) \\ (x_2 - x_0) & (y_2 - y_0) \end{bmatrix} \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} q_1 - q_0 \\ q_2 - q_0 \end{bmatrix}$$

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

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

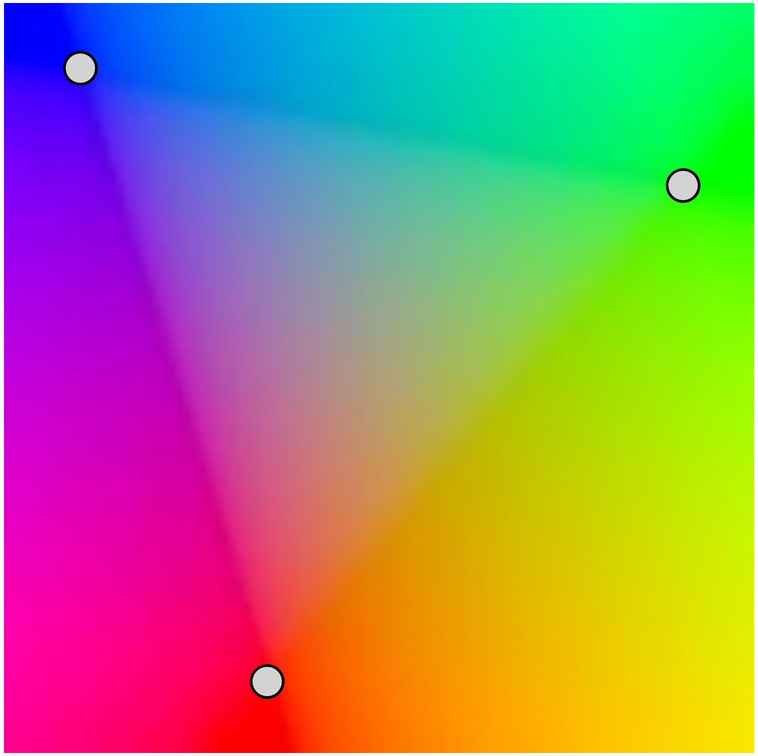
Defining parameter functions

```
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;
  }
}
```



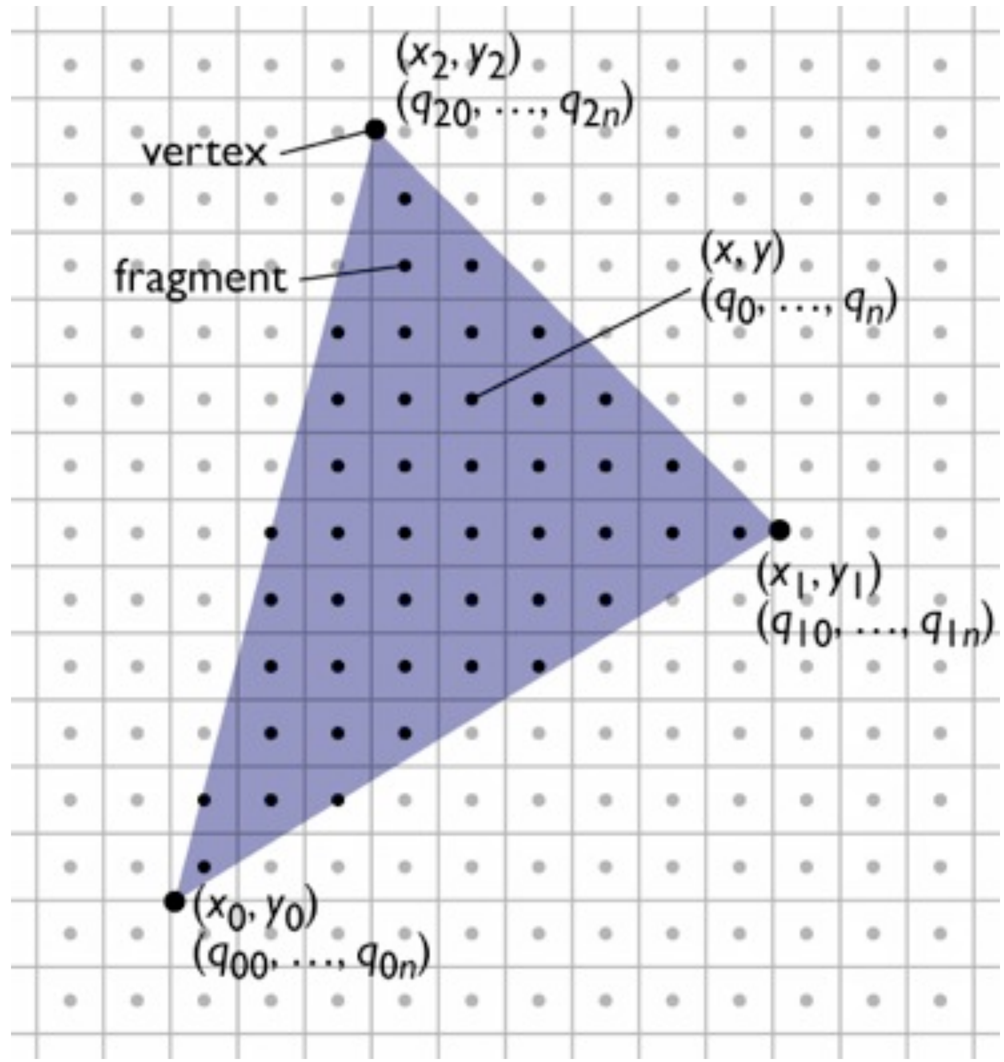
Interpolating several parameters

```
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];
  }
}
```



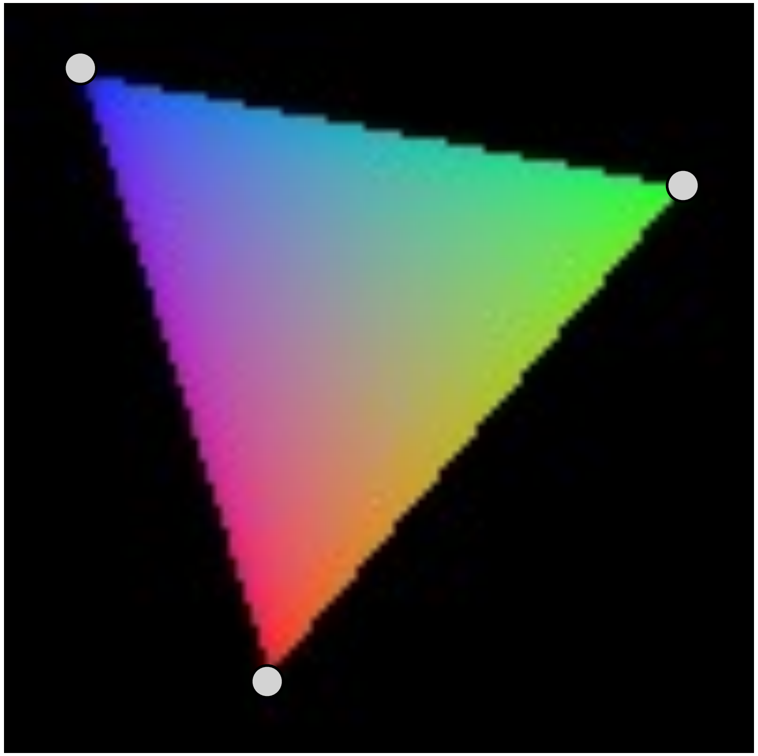
Rasterizing triangles

- Summary
  1. evaluation of linear functions on pixel grid
  2. functions defined by parameter values at vertices
  3. using extra parameters to determine fragment set



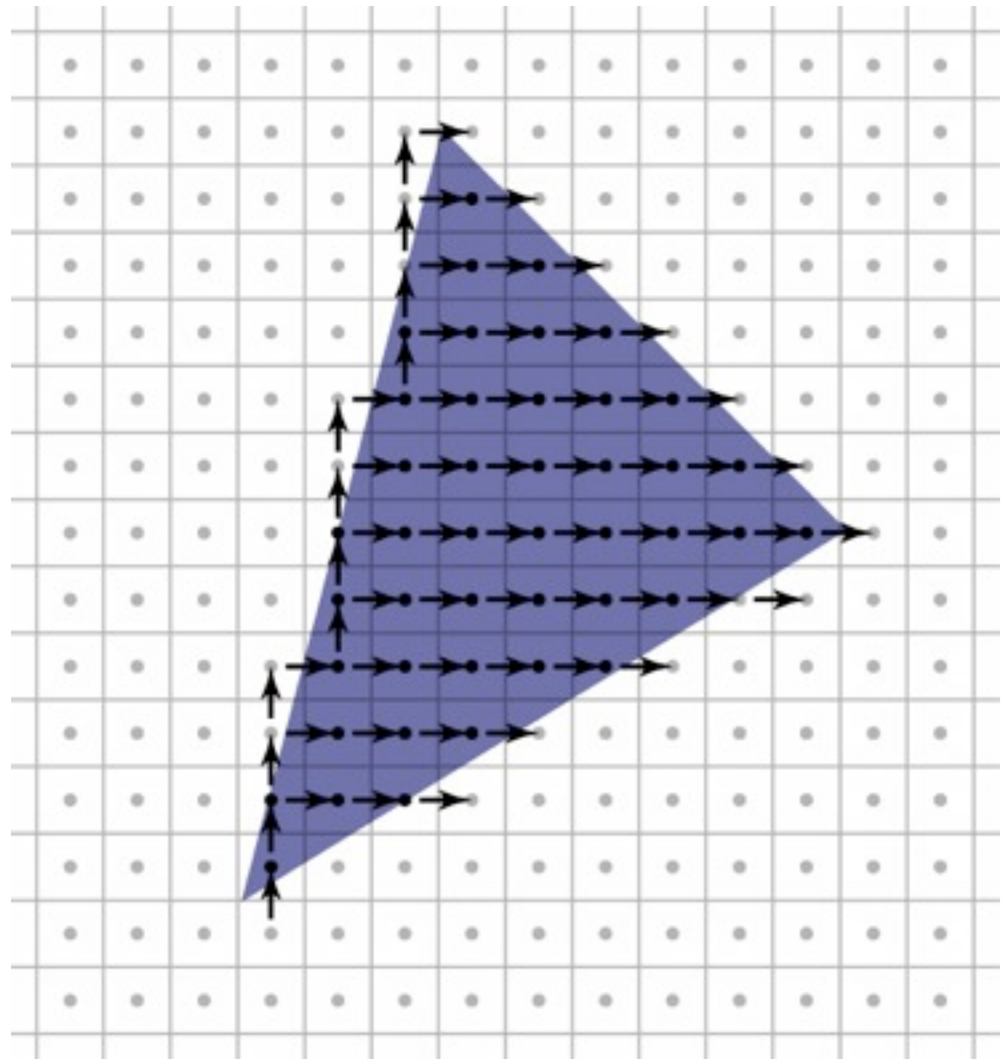
Clipping to the triangle

- 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.



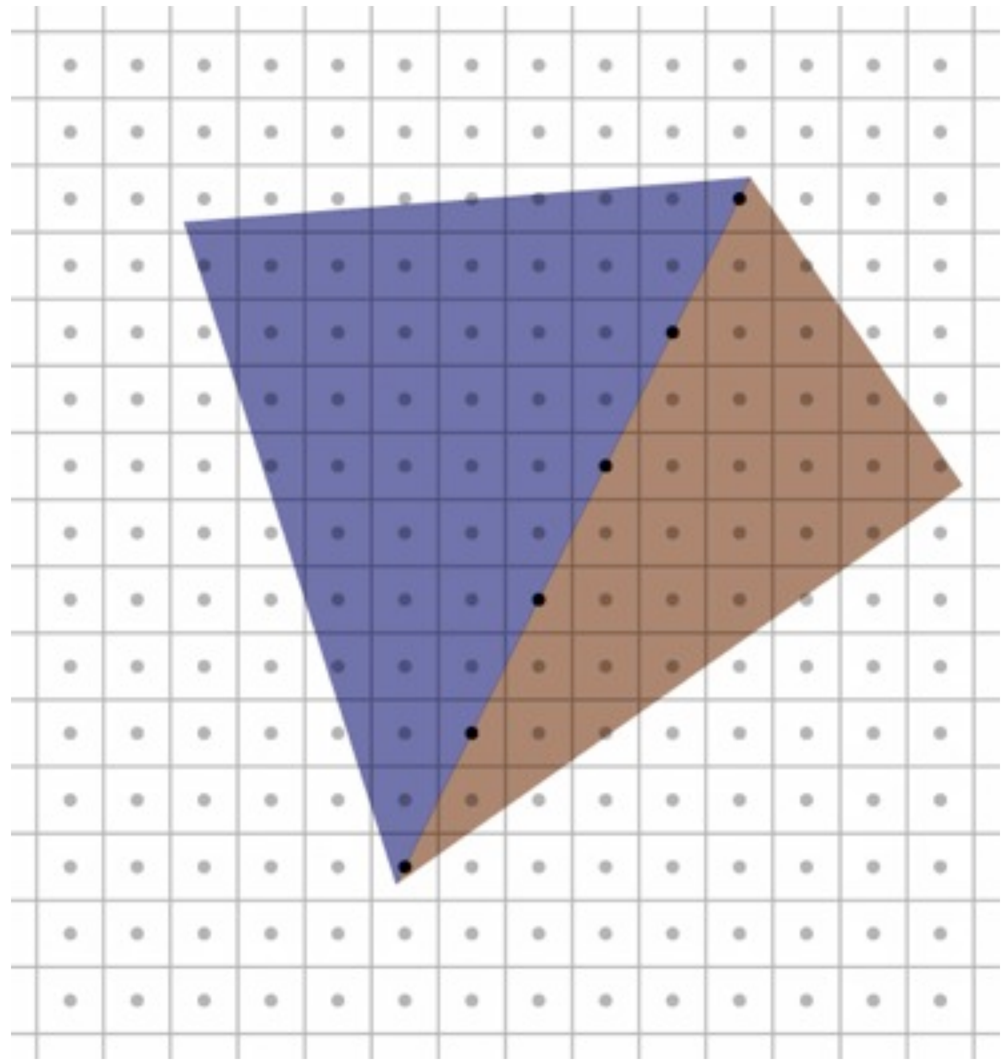
Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels you want
- 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
  - but it's important not to visit them twice!



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)