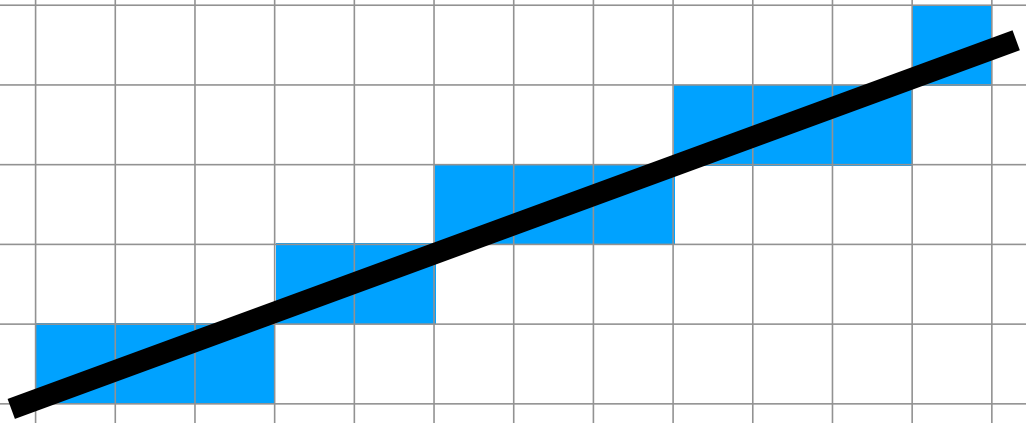


Computer Graphics

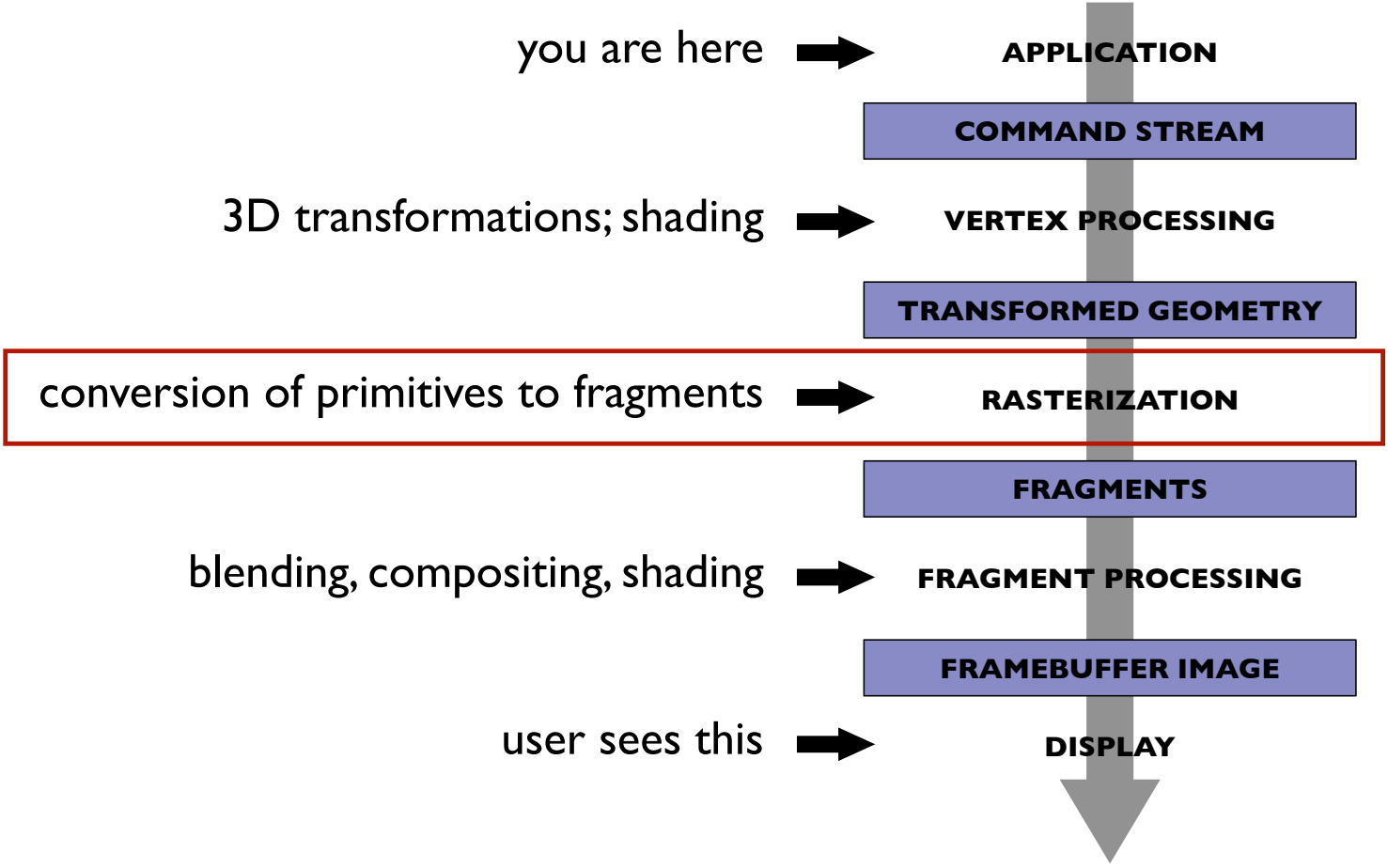
Lecture 25

Rasterizing Lines

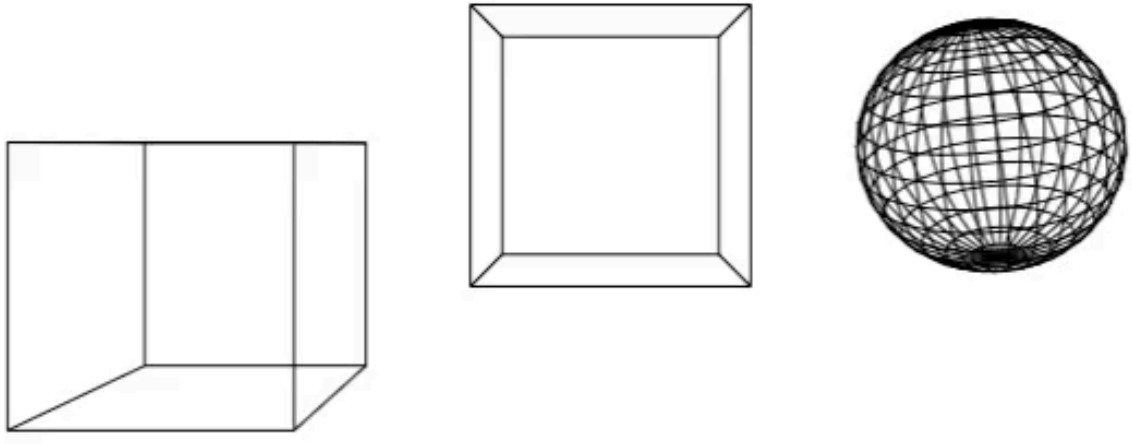
Announcements

- Midterm due tonight.
  Typo (#4 up vector) was fixed Saturday night.
- Final project feedback has been emailed. Next steps:
  - Accept final project assignment on GH classroom and create a repo
  - Make a reports/ directory and commit your proposal and my feedback (as a txt file is fine). Prepend filenames with dates so they sort chronologically.
- Today (midnight) is the **last day** to request P/NP grading.

Graphics Pipeline: Overview



Recall: Wireframe Rendering



$M = M_{vp}\, M_{proj}\, M_{view}\, M_{model}$

for each line segment $\mathbf{a}_i, \mathbf{b}_i$

  $\mathbf{p} = \mathbf{M}\, \mathbf{a}_i$

  $\mathbf{q} = \mathbf{M}\, \mathbf{b}_i$

  `draw_line(p, q)`

How do we do this?

Line Drawing

- This is a rasterization problem: given a primitive (line segment), generate fragments (pixels)

```
M = Mvp Mproj Mview Mmodel
for each line segment ai, bi
  p = M ai
  q = M bi
  draw_line(p, q)
```

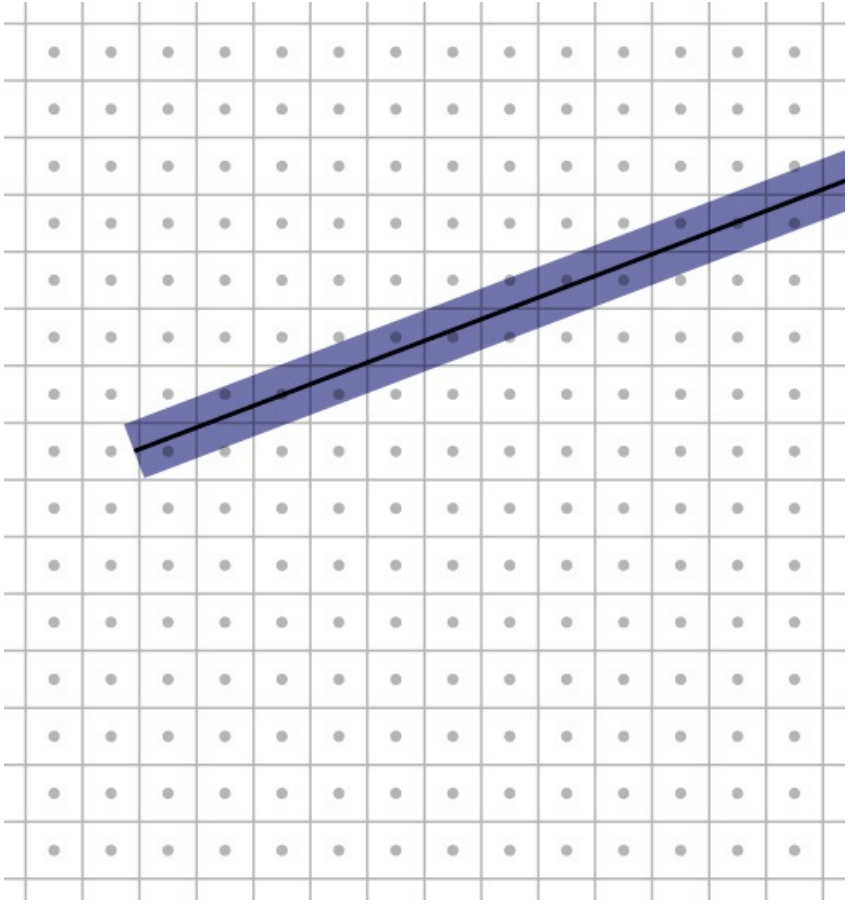
How do we do this?

Problem statement: Draw a line from p to q.

or: Decide which pixels to color to draw a line from p to q.

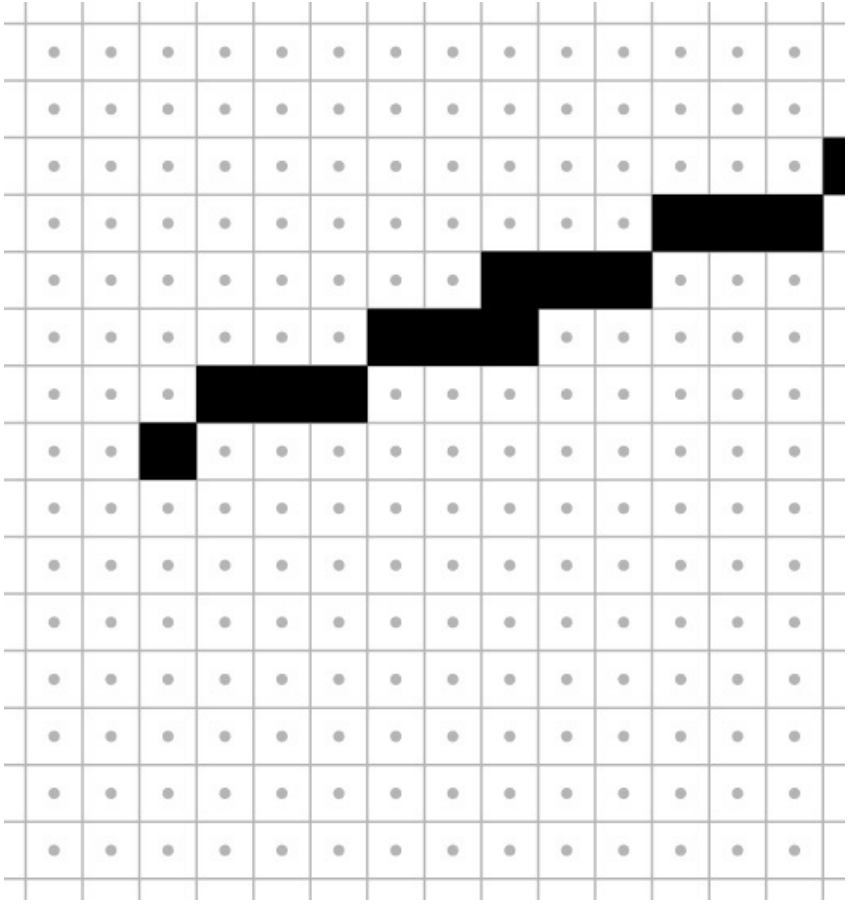
Rasterizing lines - possible definition

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

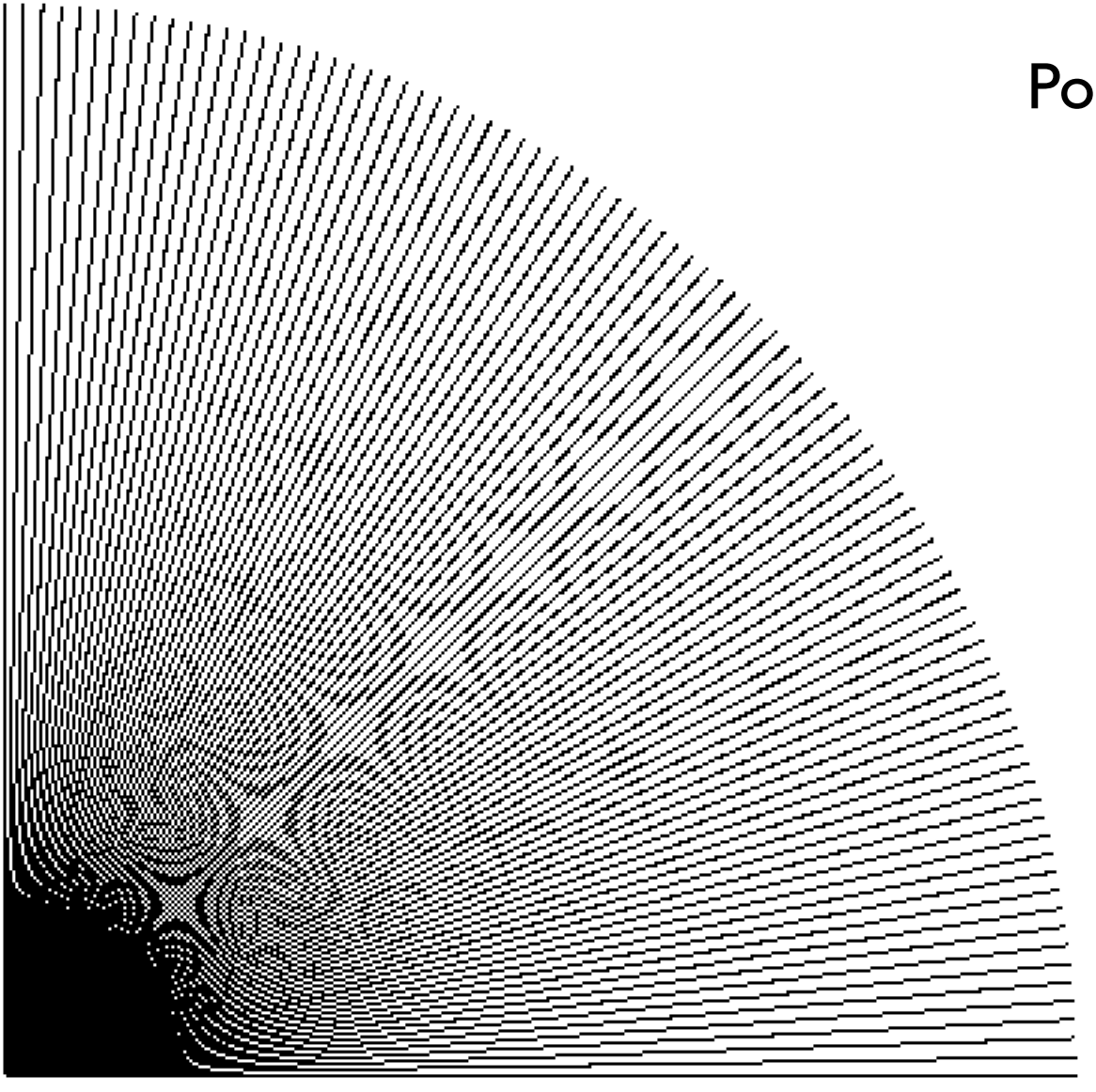


Point sampling

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

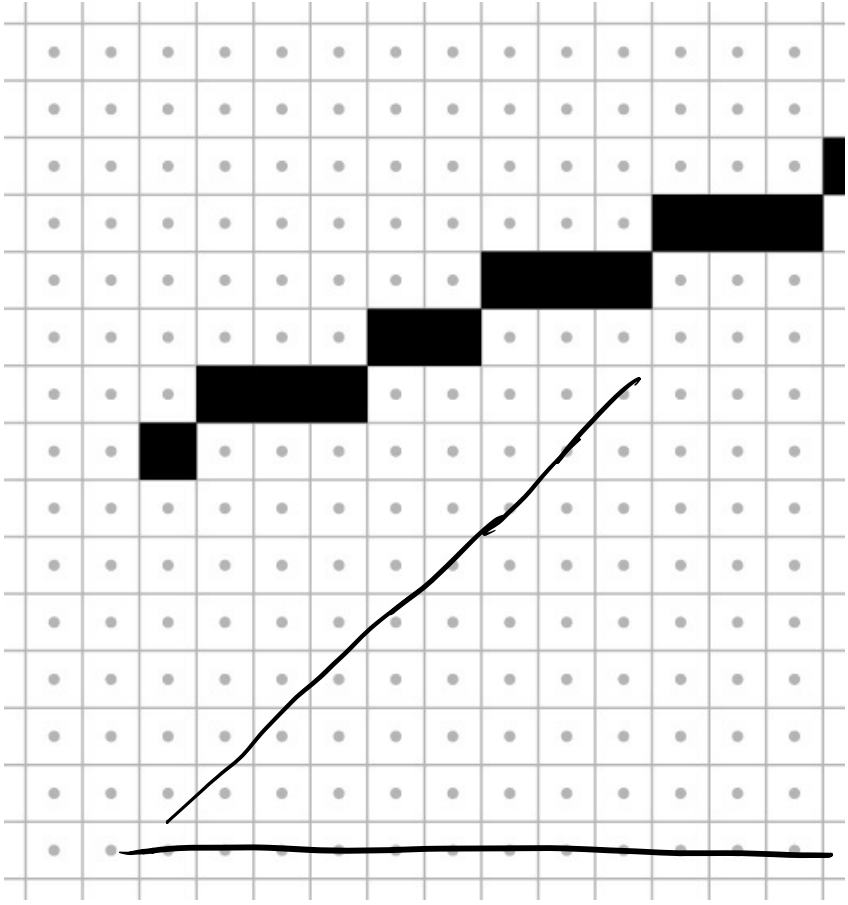


Point sampling in action

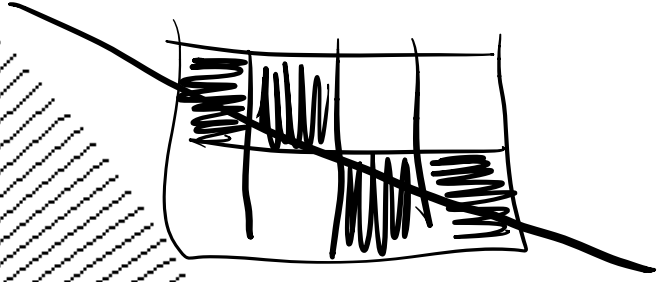


Bresenham lines (midpoint alg.)

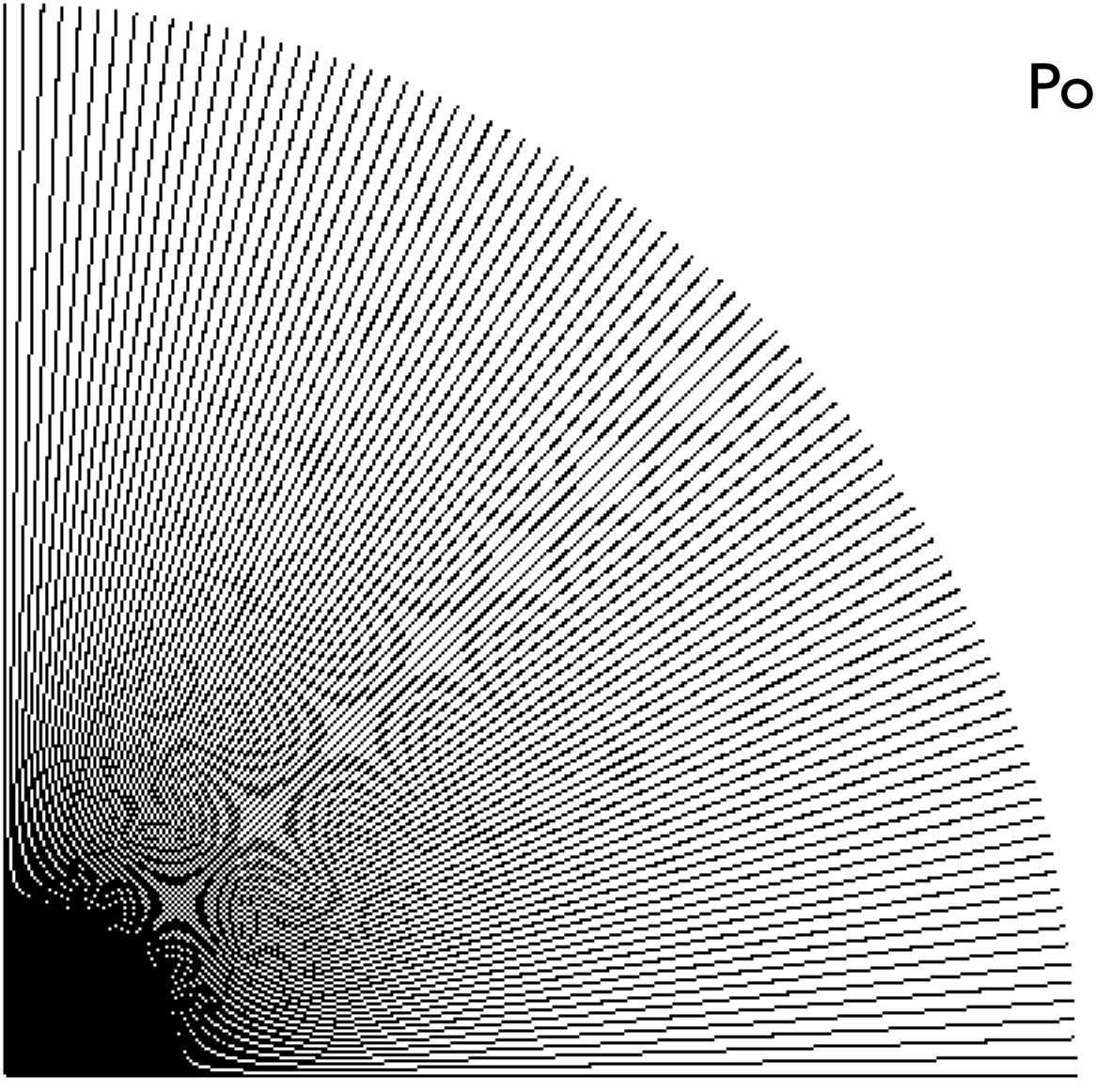
- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner



Midpoint algorithm in action



Point sampling in action

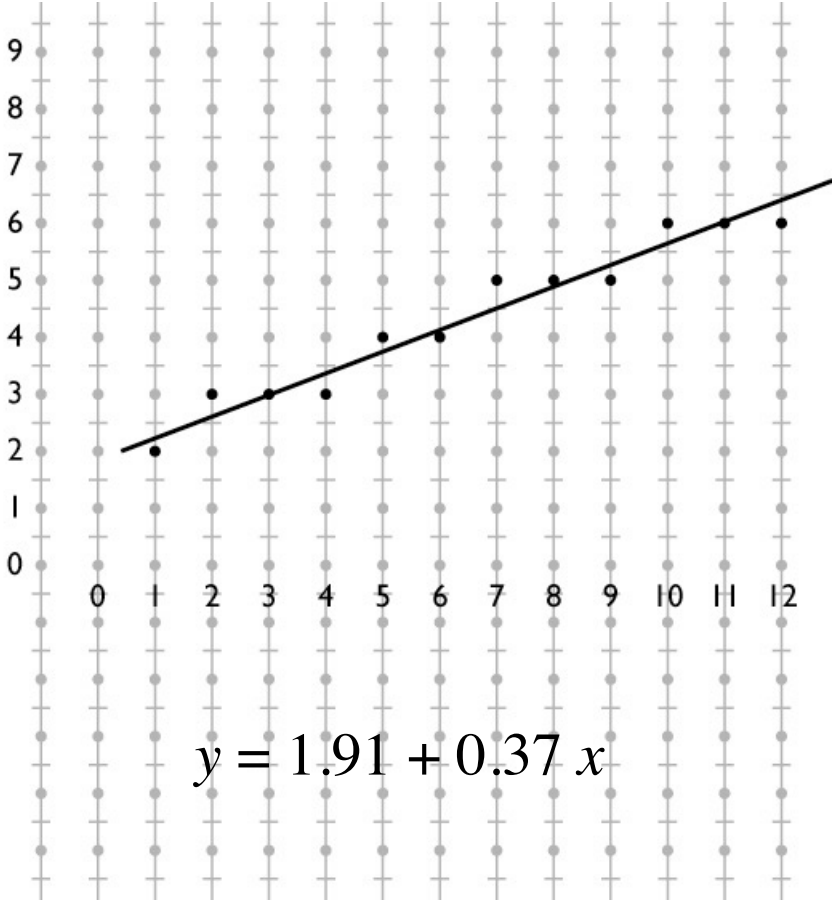


Notes:
Midpoint Algorithm

Midpoint Algorithm

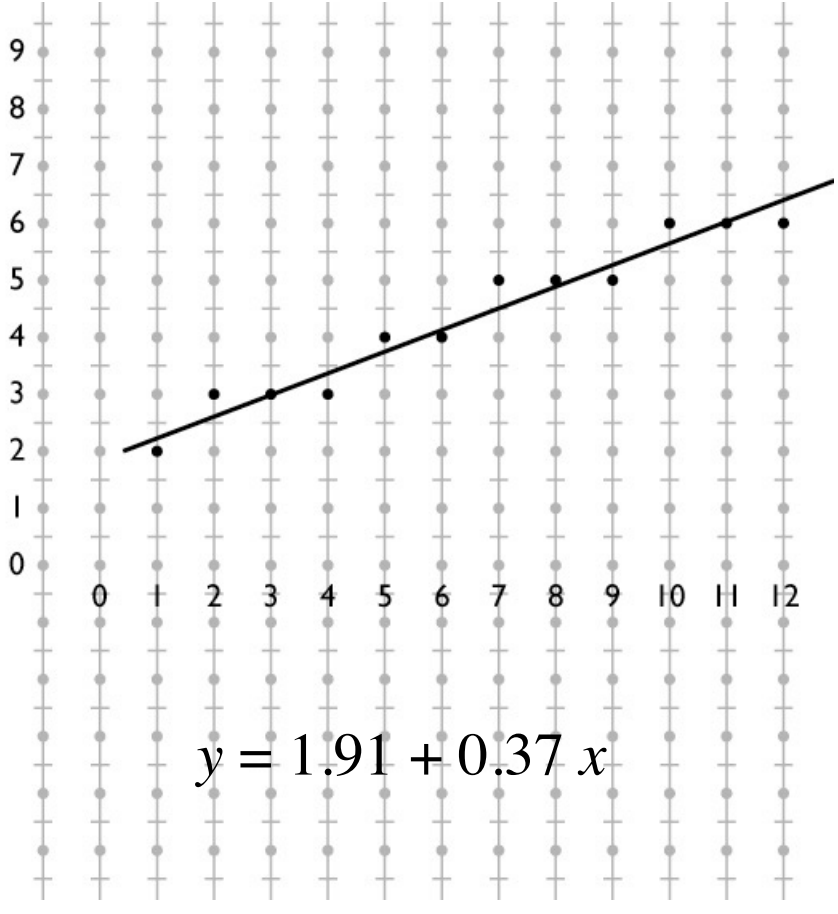
- line equation:
  $y = b + m\,x$
- Simple algorithm: evaluate line equation per column
- W.l.o.g. $x_0 < x_1$;
  $0 \le m \le 1$

Algorithm:



Midpoint Algorithm

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



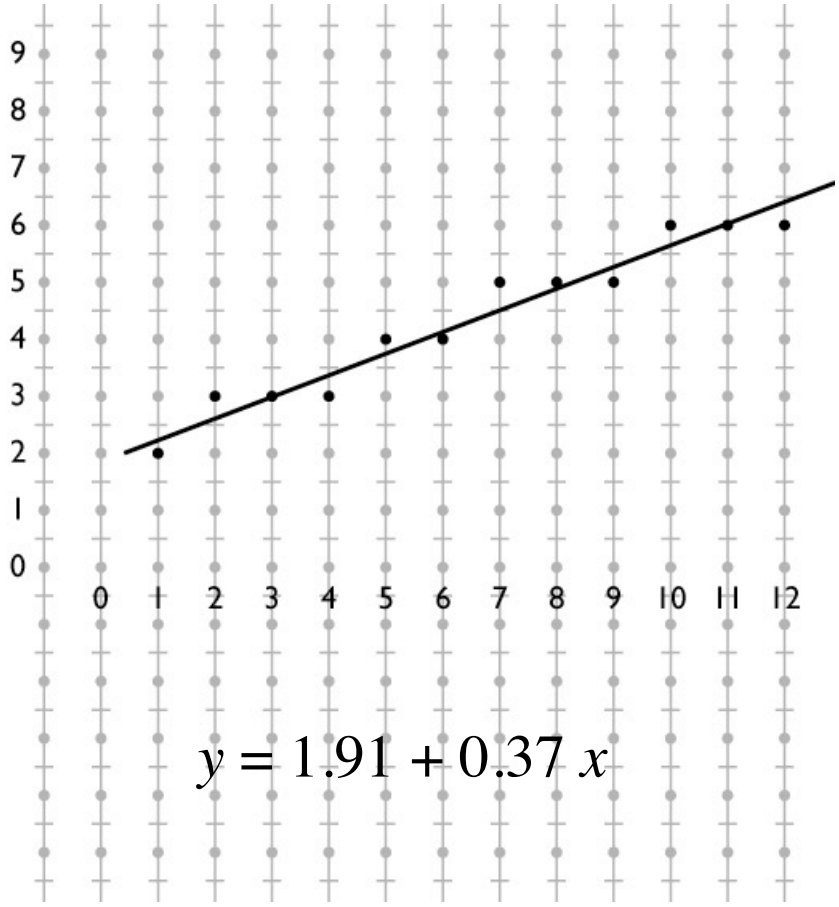
Algorithm:

```
// compute m, b

for x = ceil(x0) to floor(x1)
    y = b + m*x
   // Ex: what goes here?
```


Algorithms for drawing lines

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



Algorithm:

```
// compute m, b

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


Optimizing Line Drawing

Can we take stuff out of the inner loop?

Exercise: optimize this

```
function slow_line(p1, p2):
  // compute m, b
  for x = ceil(x0) to floor(x1)
      y = b + m*x
      draw(x, round(y))
```

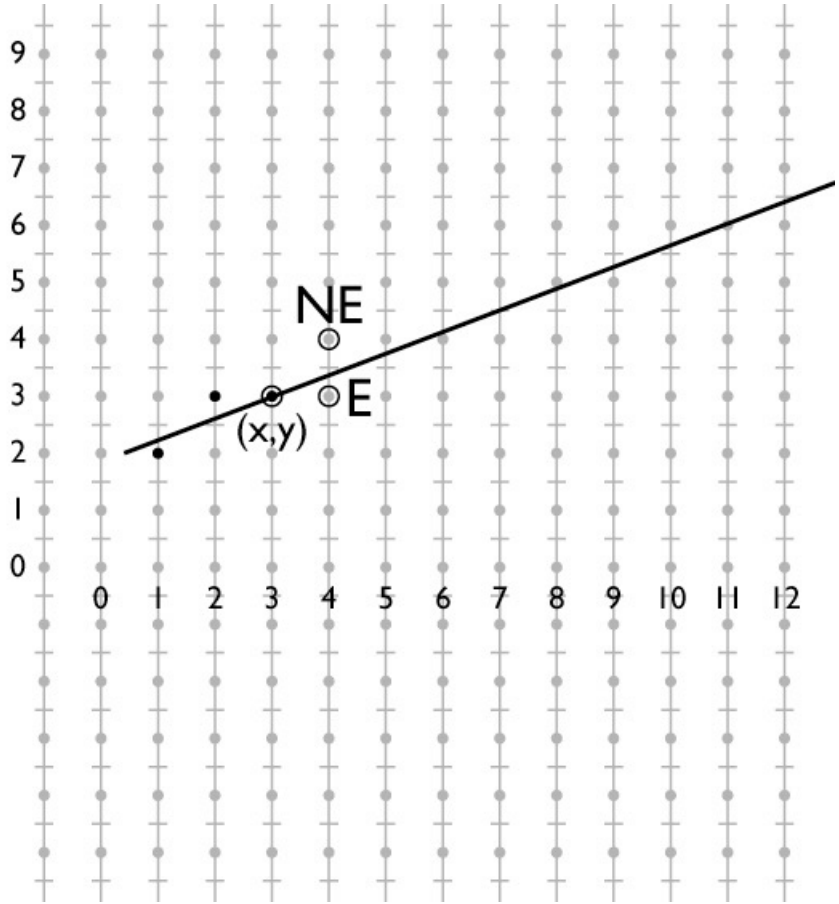
```
function fast_line(p1, p2):
  // compute m, b

  for x = ceil(x0) to floor(x1)

      draw(x, round(y))
```

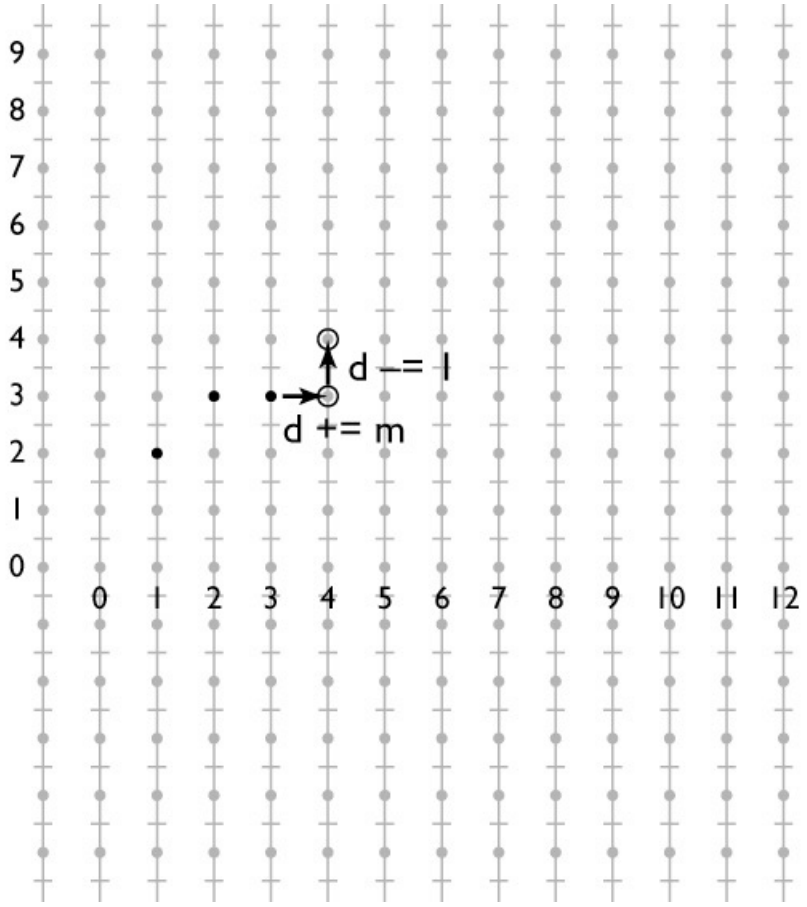
Optimizing Line Drawing Even More

- Rounding is slow too
- At each pixel the only options are E and NE
- Track distance to line:
  - $d = m(x + 1) + b - y$
  - $d > 0.5$ decides between E and NE



Optimizing Line Drawing Even More

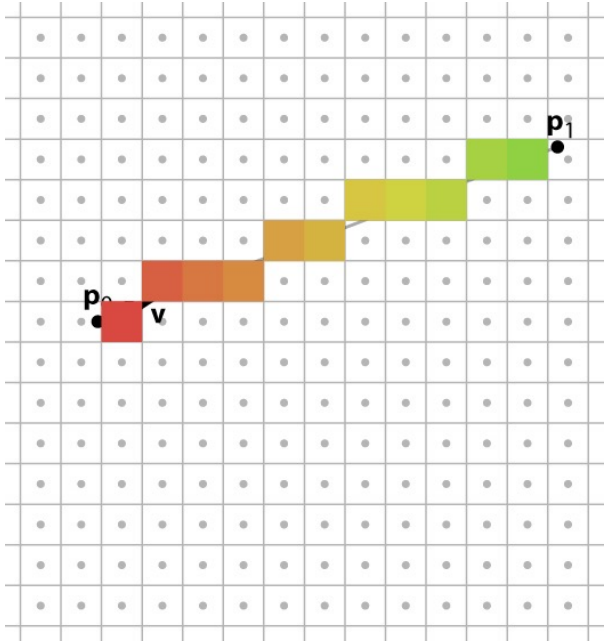
- $d = m(x + 1) + b - y$
- Only need to update d for integer steps in x and y
- Do that with addition
- Known as "DDA" (digital differential analyzer)



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

- Same machinery as we used for y works for other values!

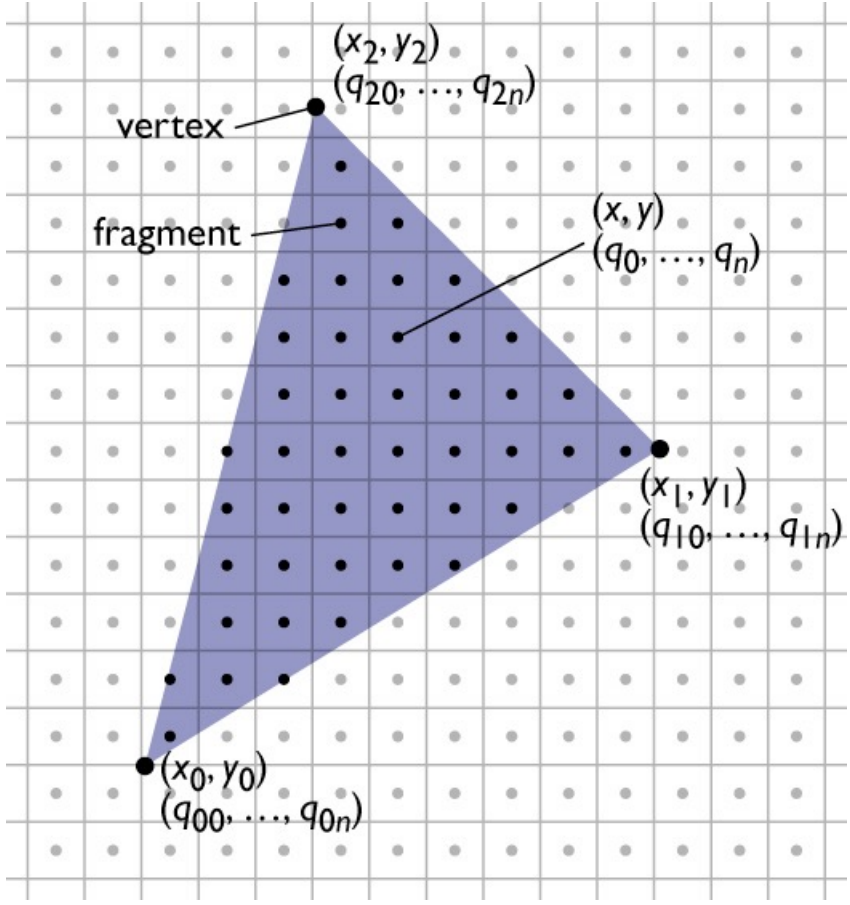


Rasterizing triangles

- 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



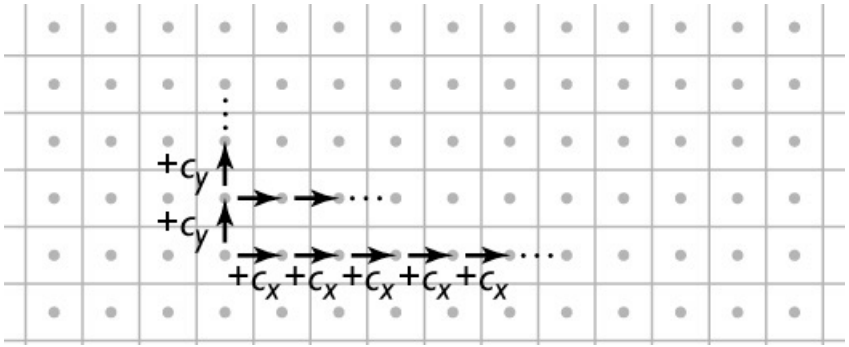
Incremental linear evaluation

- 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:

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



Pixel-walk (Pineda) rasterization

- Conservatively visit a superset of the pixels you want
- Interpolate linear functions
  - barycentric coords (determines when to emit a fragment)
  - colors
  - normals
  - whatever else!