CSSE 351
Computer Graphics

DDAs and line drawing

Session schedule

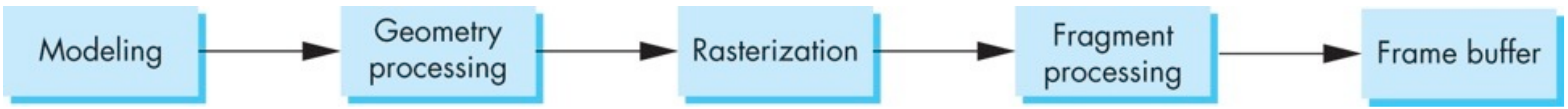
- Rasterization
- DDAs
- Line drawing

Render pipeline

- All geometry is in NDC
- No geometry out of view volume (NDC)

- Convert to fragments (pixels)

Render pipeline



- Render pipeline changes coordinate/vector spaces
- Ready for
  - Fragment conversion
  - Interpolation
  - Depth sorting

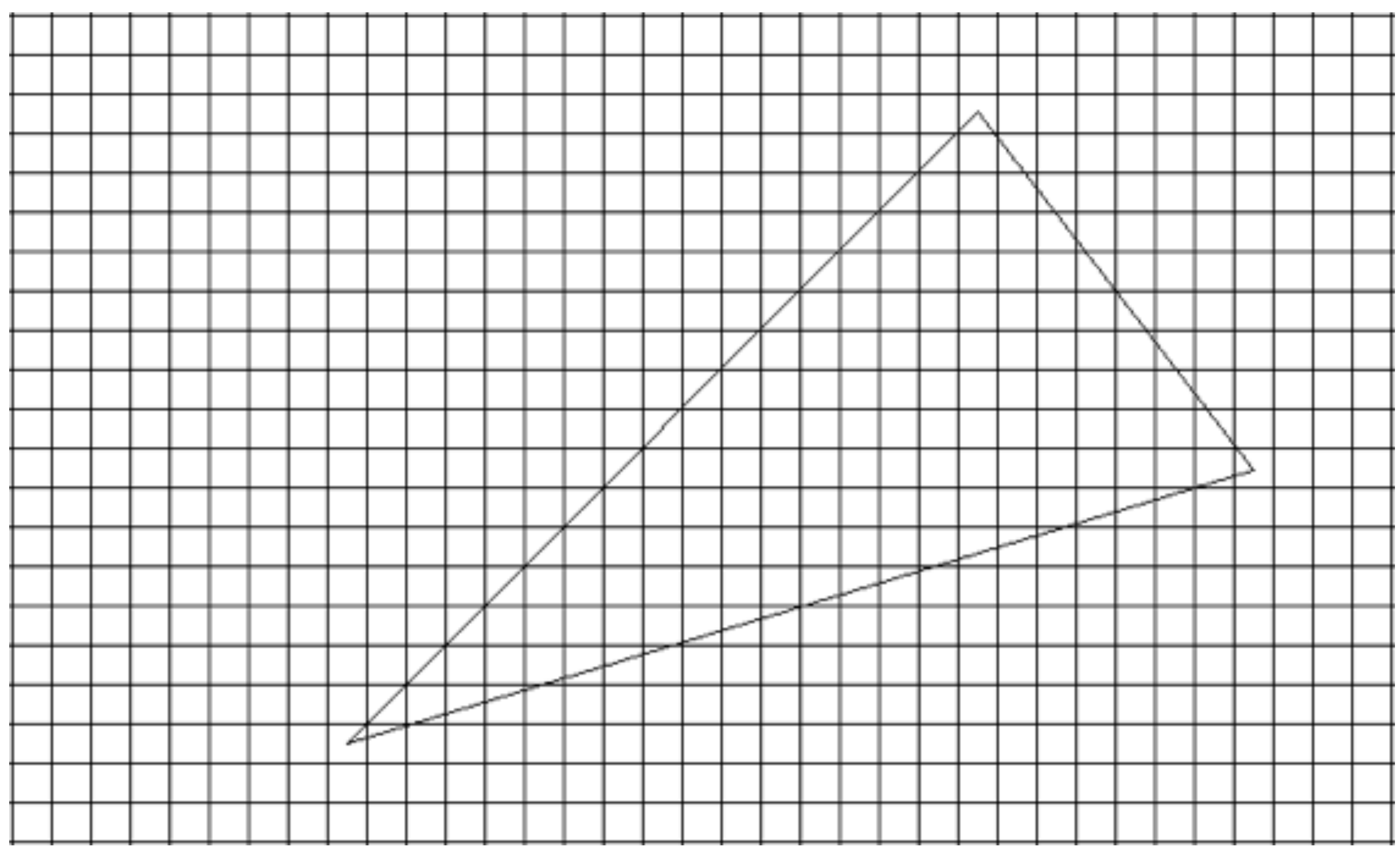
Rasterization

- Compute fragment locations in window coordinates
- Interpolate vertex attributes
- Compute fragment color

- Sort fragments by depth

On screen display

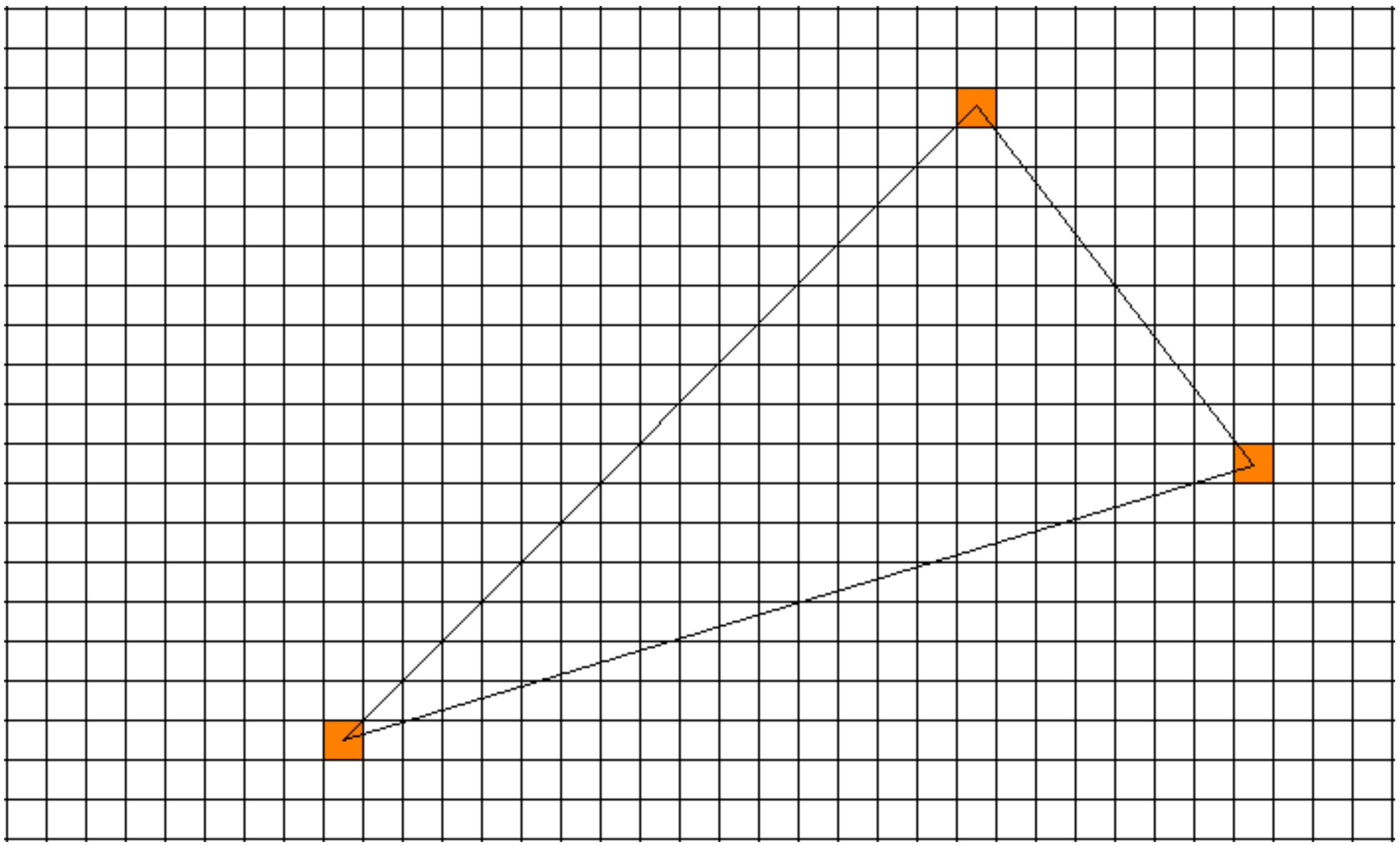
- Write some data to frame buffer
- Starting from geometric data



Draw points

- Simplest data is to show points
  - Transform vertices
  - Convert to NDC, then viewport
  - Clamp/round to pixel value, show on screen!

Draw points



Drawing lines

- Transform vertices
- Convert to NDC, clip, convert to viewport
- Now have sets of lines in 2D space
  - Need to convert 2D geometry into pixels

Drawing lines

- Convert endpoints to pixel values
  x1, y1
  x2, y2
- Draw line between pixel values
- Use DDA (Digital Difference Analyzer)

DDA

- Compute line differential

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

- Restrict slope to

$$0 \leq m \leq 1$$

- Vertical change is then

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

DDA

- Using vertical change, make unit steps in x

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

$$\Delta x = 1$$

- Vertical change is then

$$\Delta y = m$$

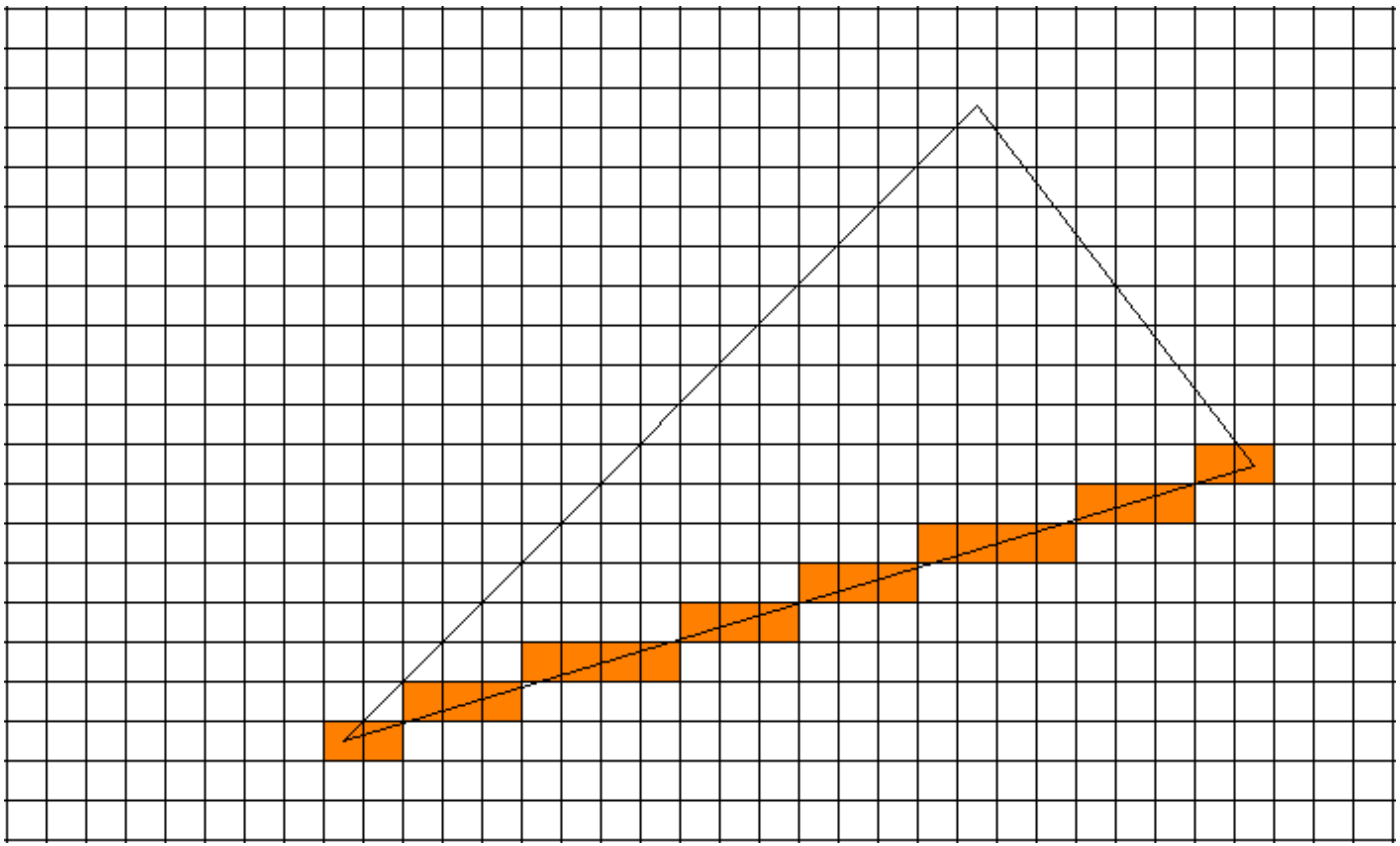
- Algorithm to draw line is...

DDA line drawing

```
m = (y2-y1)/(x2-x1)

for x = x1 to x2
    y+=m
    color_pixel(x, round(y))
```

DDA line drawing



DDA

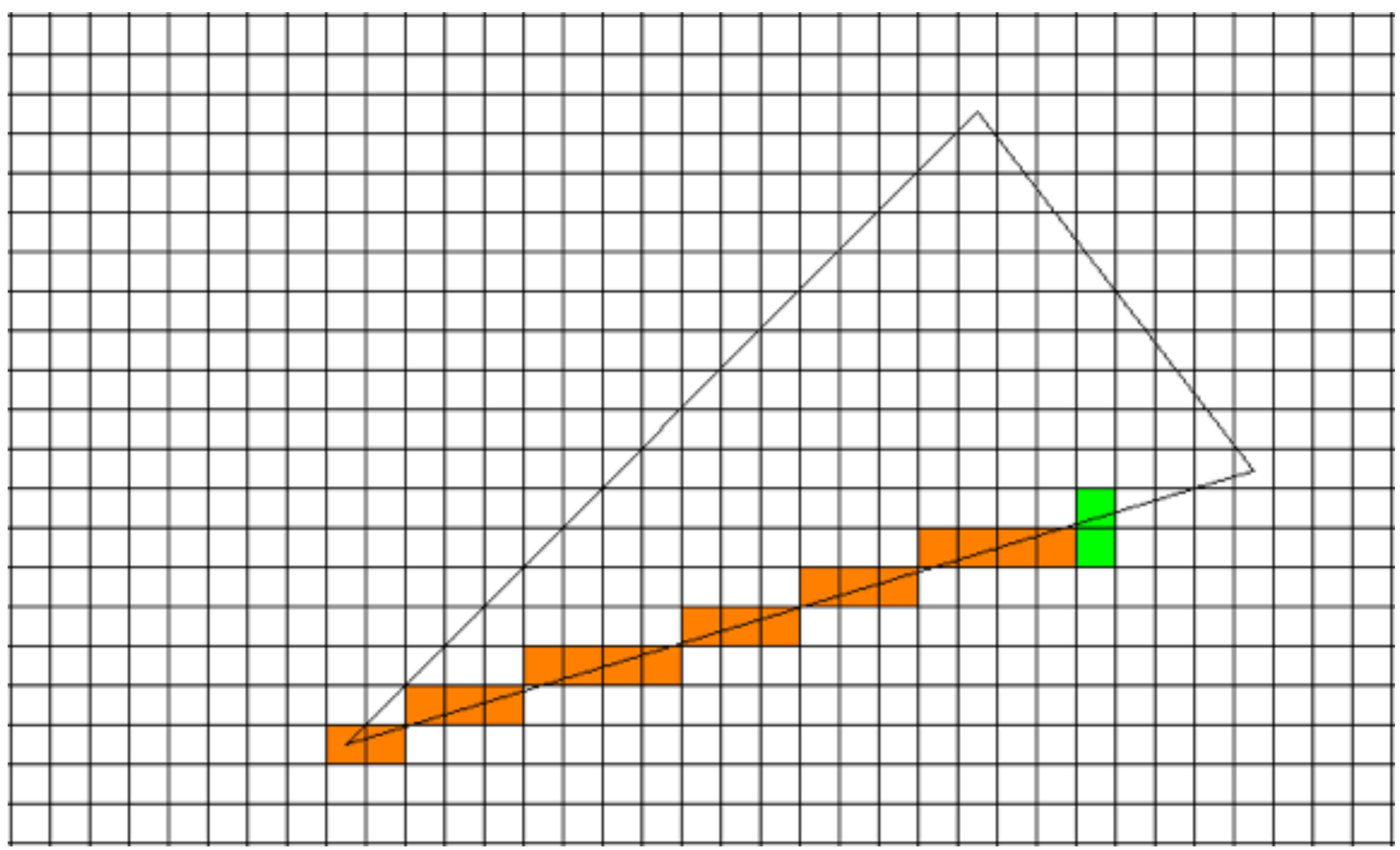
- Requires floating point operations
- Possible to draw lines with only integers
  - Bresenham's line drawing algorithm
  - We will cover simpler midpoint version

Midpoint line drawing

- Make some assumptions
  - x1 < x2 (swap if needed)
  - Slope is (0,1]
  - Lines have no gaps, diagonal pixels connect
- How does this help?

Midpoint line drawing

- Lines must go right or right+up!
- Just draw increasing x, and move up sometimes



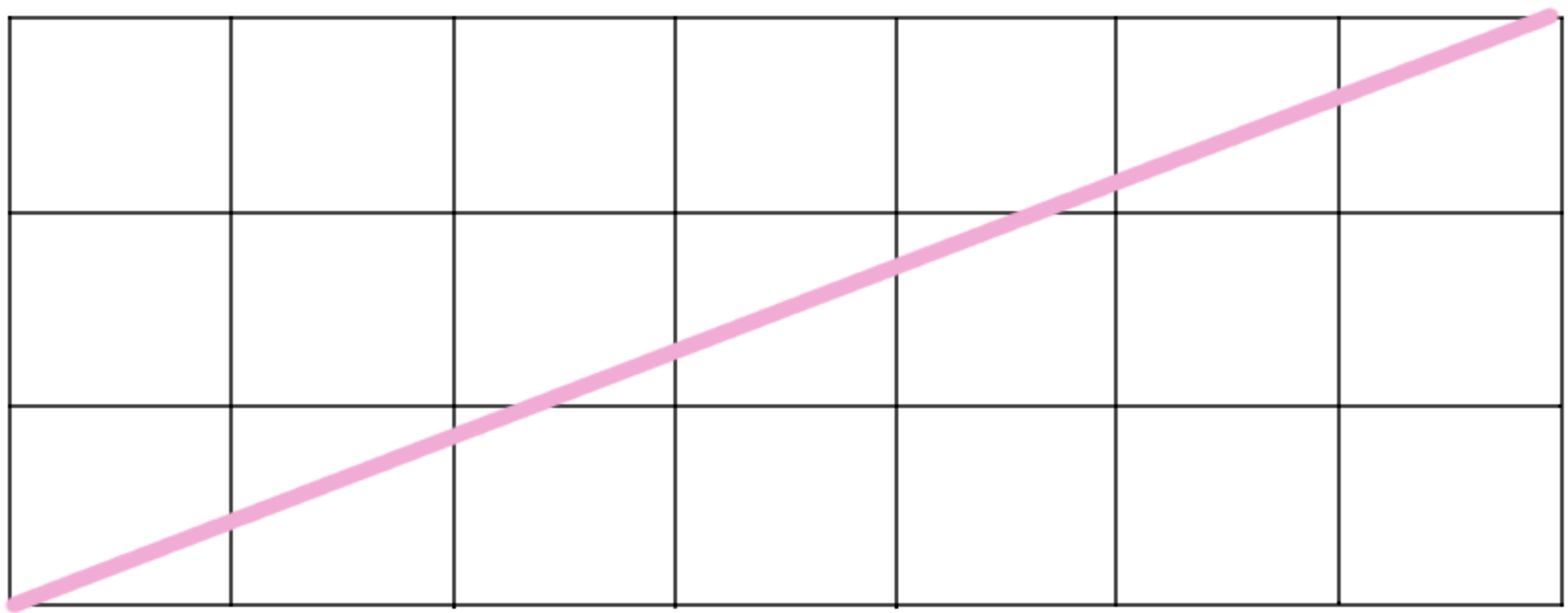
Midpoint line drawing

- Resulting code:

```
y = y0
for x = x0 to x1
    draw(x, y)
    if(some condition)
        y = y + 1
```

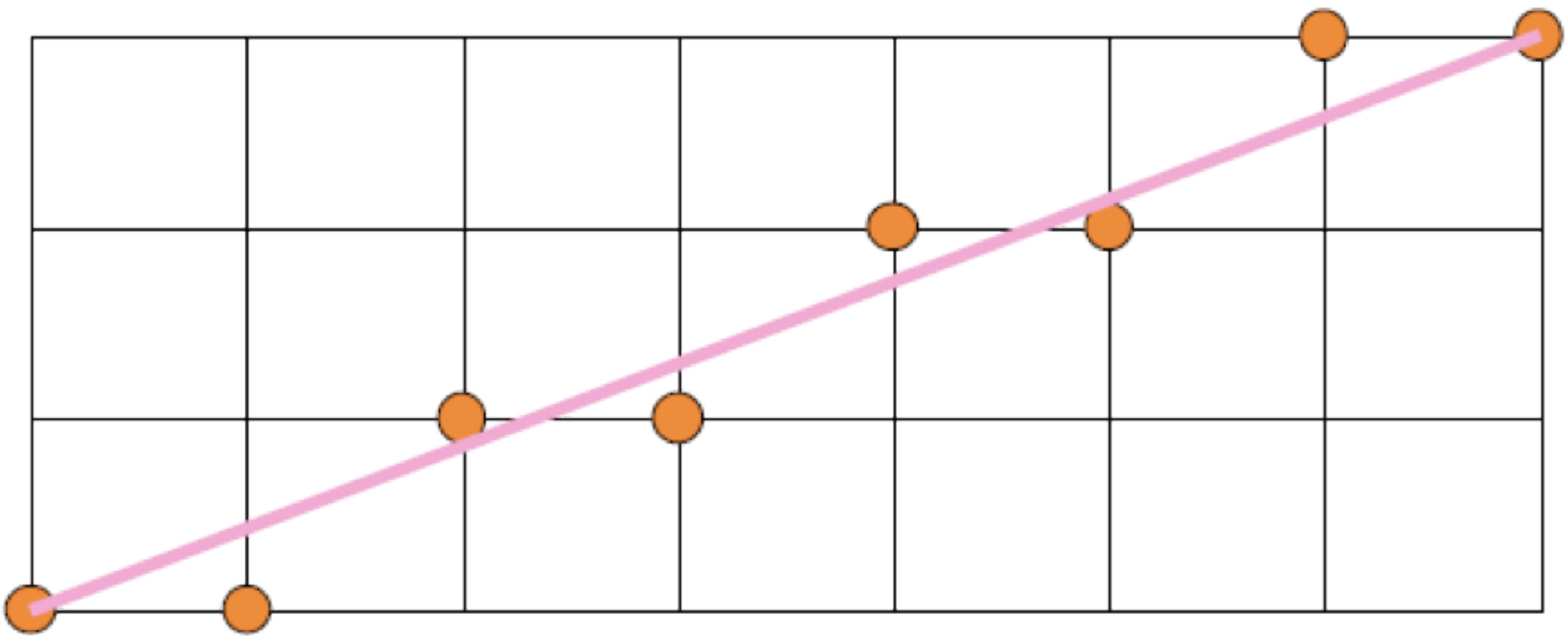
Midpoint condition

-



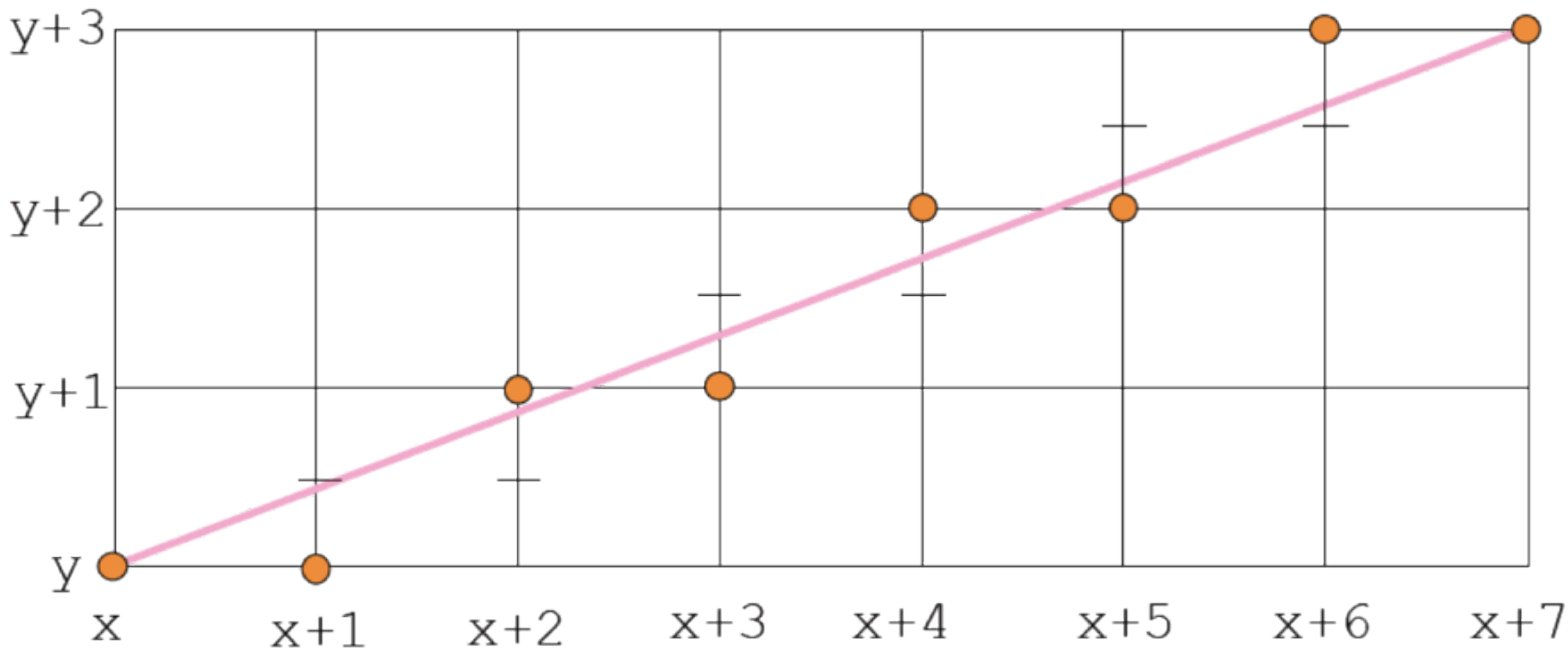
Midpoint condition

- Closest pixel is ‘on the line’
  - Orange dots are pixel centers



Midpoint condition

- Check if line is above or below midpoint



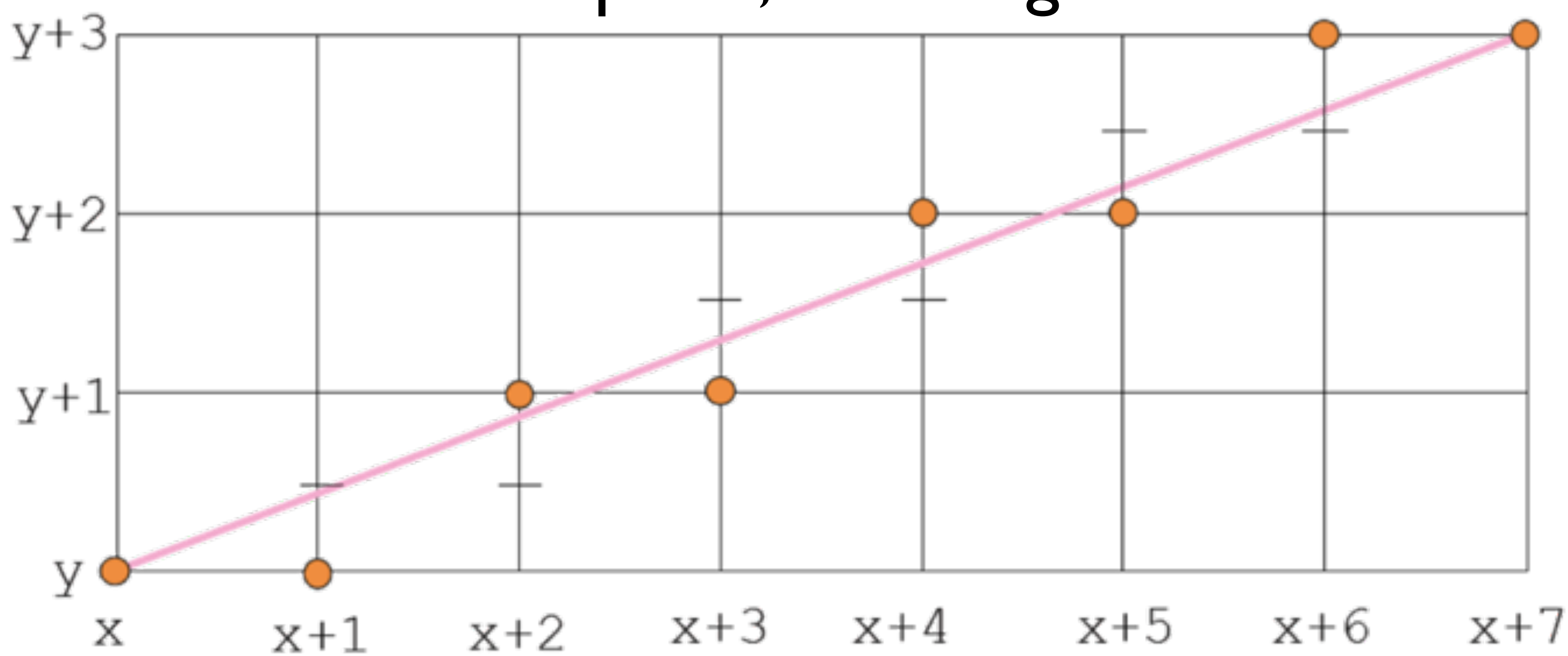
Midpoint condition

- Check if line is above or below test point
- Use implicit line equation
  - 0 when on the line
  - < 0 when below line
  - > 0 when above line

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 = 0$$

Midpoint condition

- Compute implicit line equation for 2D line
- Test next pixel midpoint
  - If line above midpoint, move up+right
  - If line below midpoint, move right



Midpoint line drawing

- Condition checks if line above midpoint
- By seeing if midpoint is below line

$$f(x+1, y+0.5) < 0$$

- If so, move right and move up

Midpoint line drawing

- Code becomes (with line equation f):

```
y = y0
for x = x0 to x1
    draw(x, y)
    if( f(x+1, y+0.5) < 0 )
        y = y + 1
```

Optimize

- Avoid evaluating full line equation
- Precompute midpoint and increment
- Line:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 = 0$$

- Move right:

$$f(x + 1, y) = f(x, y) + (y_0 - y_1)$$

- Move up+right:

$$f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0)$$

Incremental midpoint

- Code:

```
y = y0
d = f(x0 + 1, y0 + 0.5)
for x = x0 to x1
    draw(x, y)
    if(d < 0)
        y = y + 1
        d = d + (x1-x0) + (y0-y1)
    else
        d = d + (y0-y1)
```

Optimize

- Last optimization is to remove floating point...

- Might discuss much later

Using midpoint & DDAs

- But, but, but!
  - Other slopes?

- Swap order of line endpoints
- Symmetric around origin
- Swap x and y, minor adjustment to calculations (plus vs. minus)