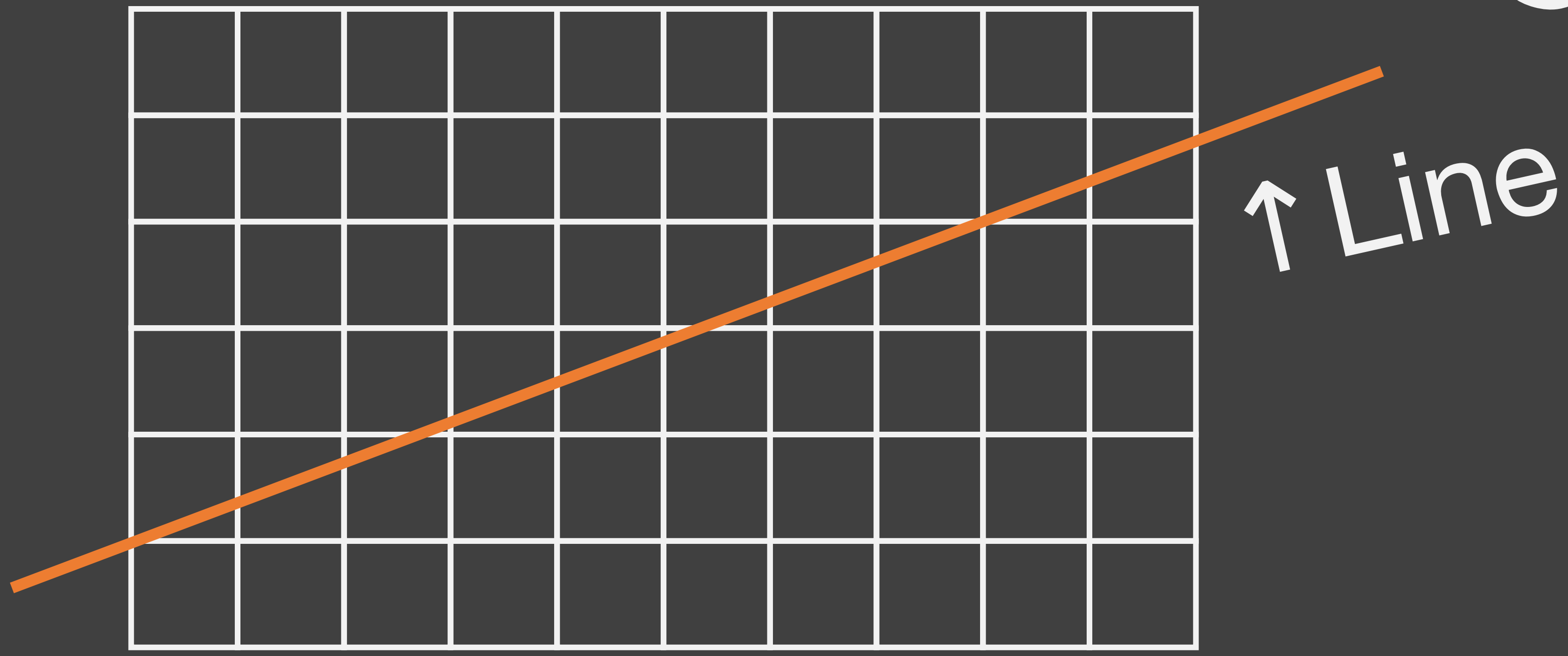
Line Drawing

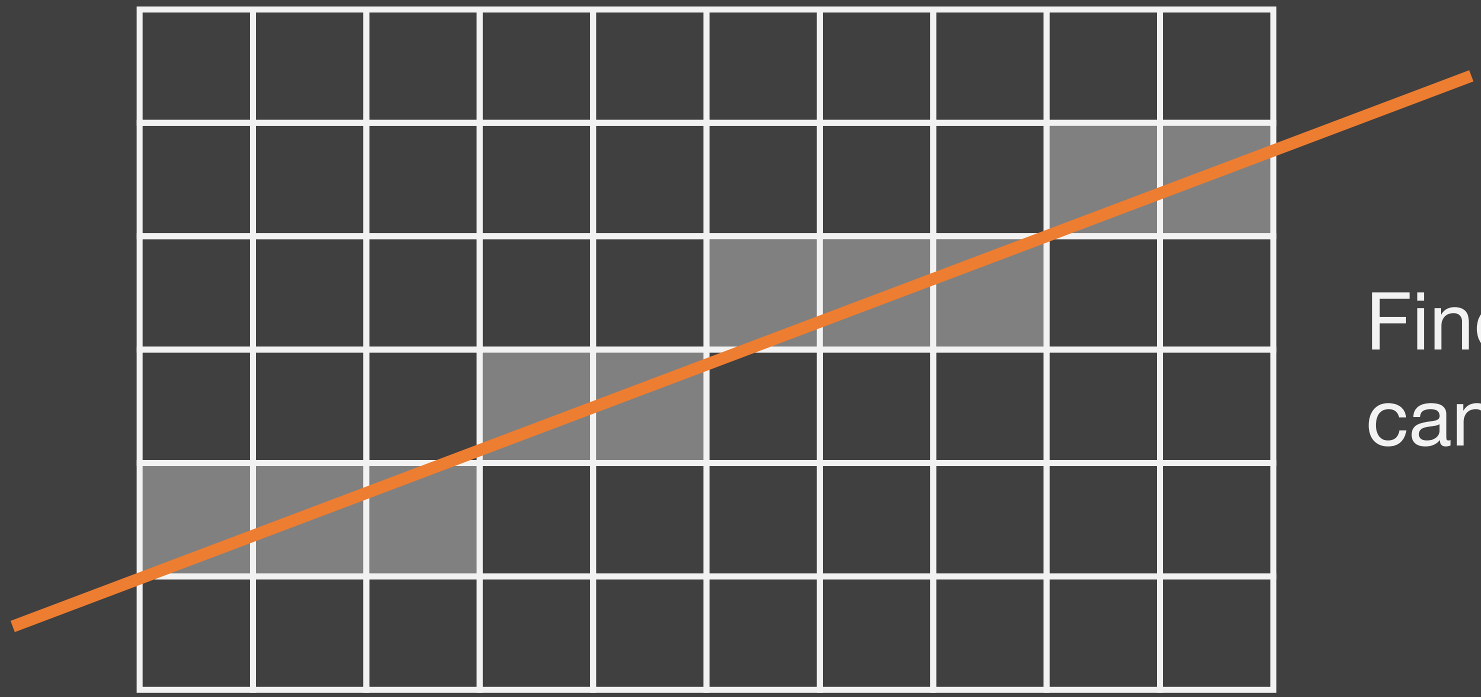
Algorithm, Collision Detection, and More

The Problem



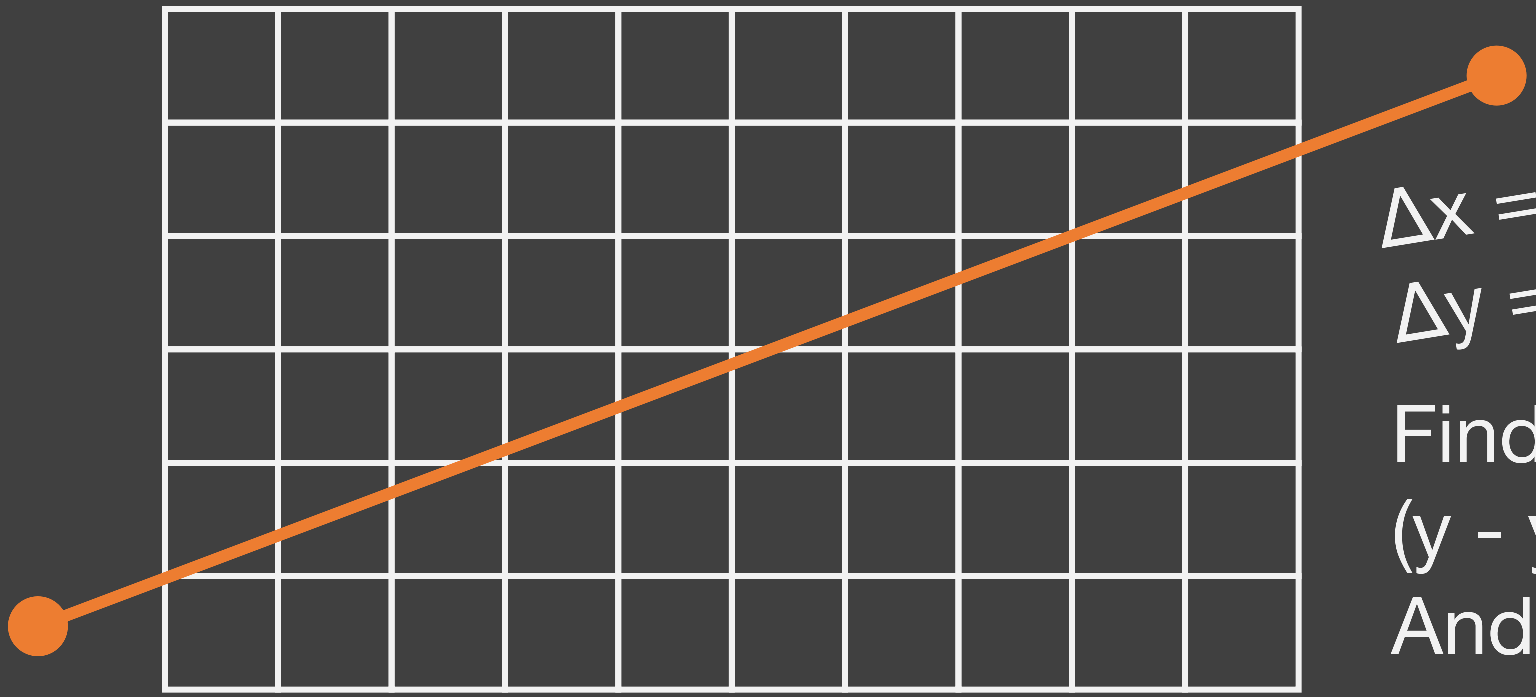
The Problem

Pixels ↓



Find the pixels that
can represent the line

Naïve Approach



$\Delta x = x_2 - x_1$

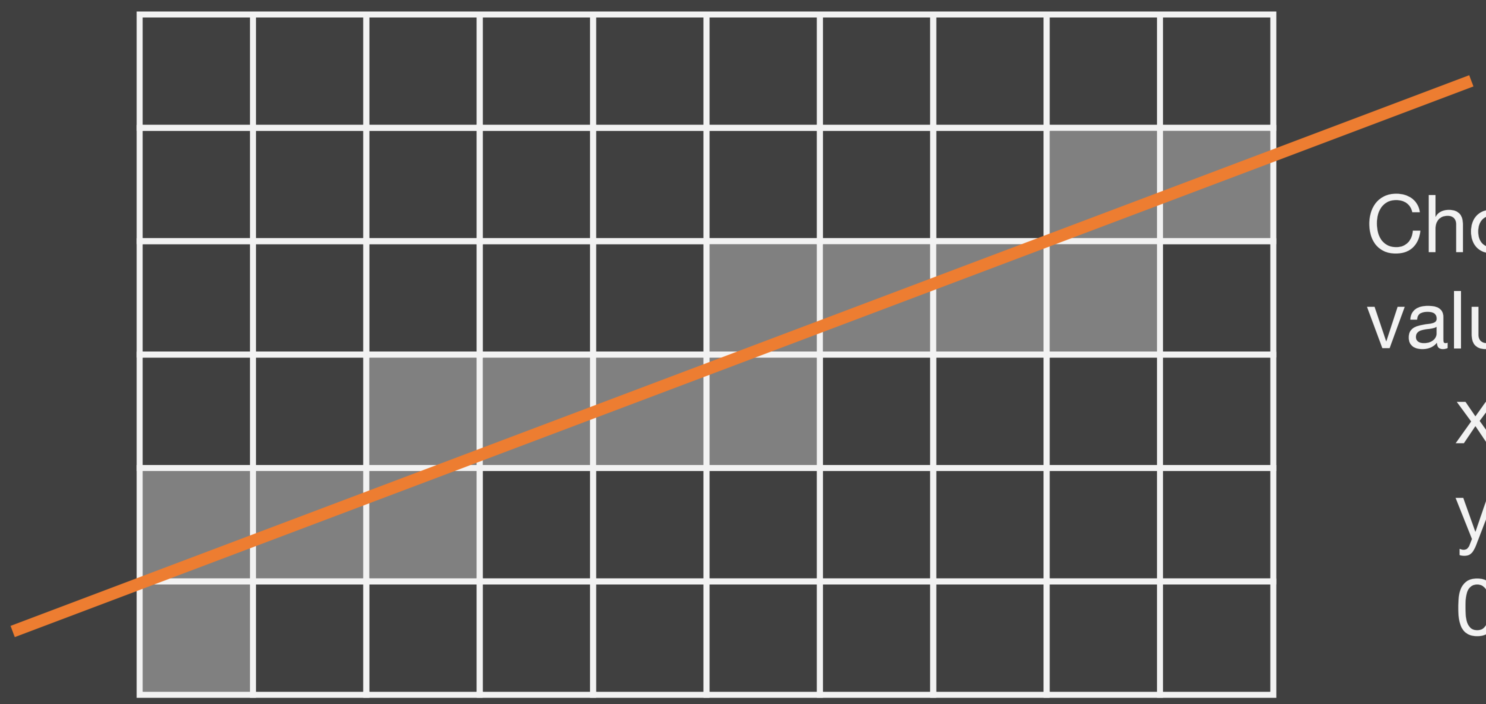
$\Delta y = y_2 - y_1$

Find some (x, y) that

$(y - y_1) / \Delta y = (x - x_1) / \Delta x$

And round x, y up

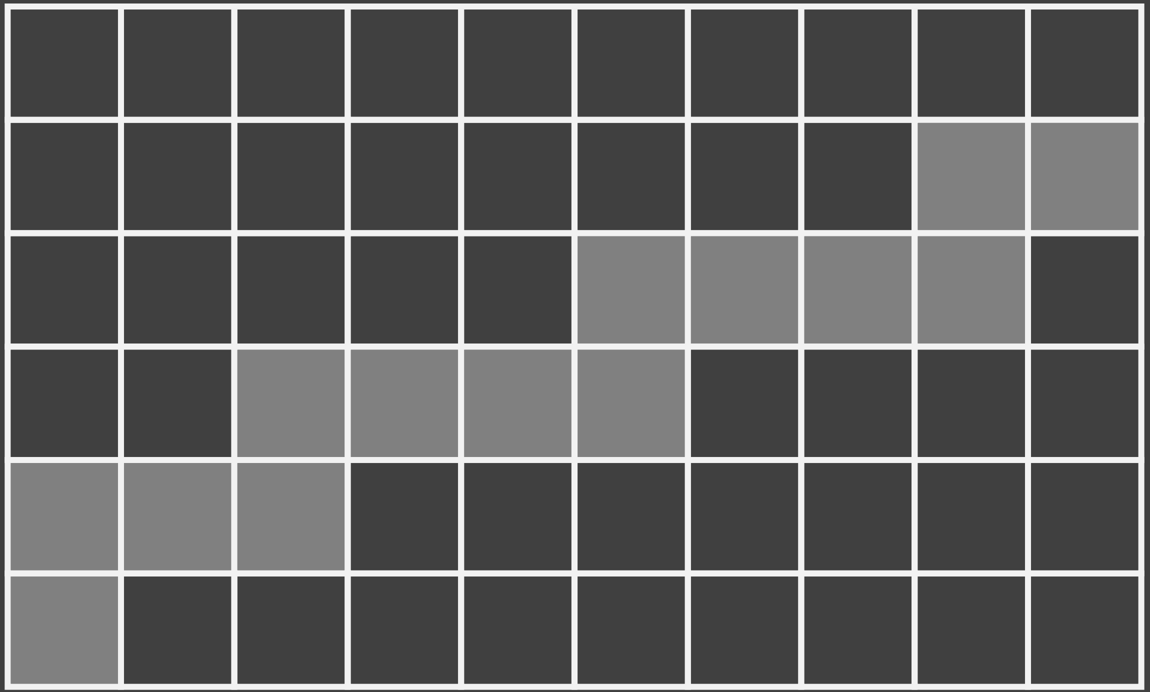
Naïve Approach #1



Choose as many values of t as possible

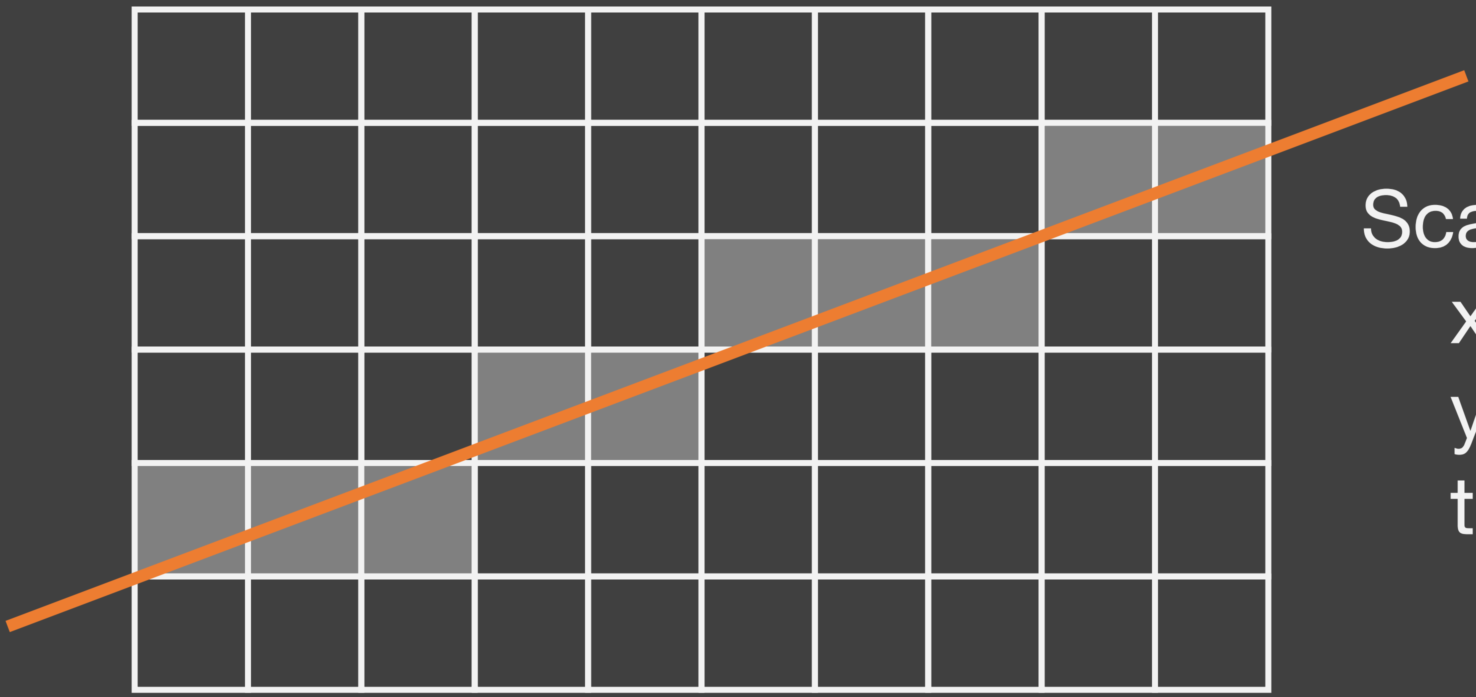
$$x = \|x_1 + t\,\Delta x\|$$
$$y = \|y_1 + t\,\Delta y\|$$
$$0 \le t \le 1$$

Naïve Approach #1



Too Dense!
(but sometimes useful)

Naïve Approach #2



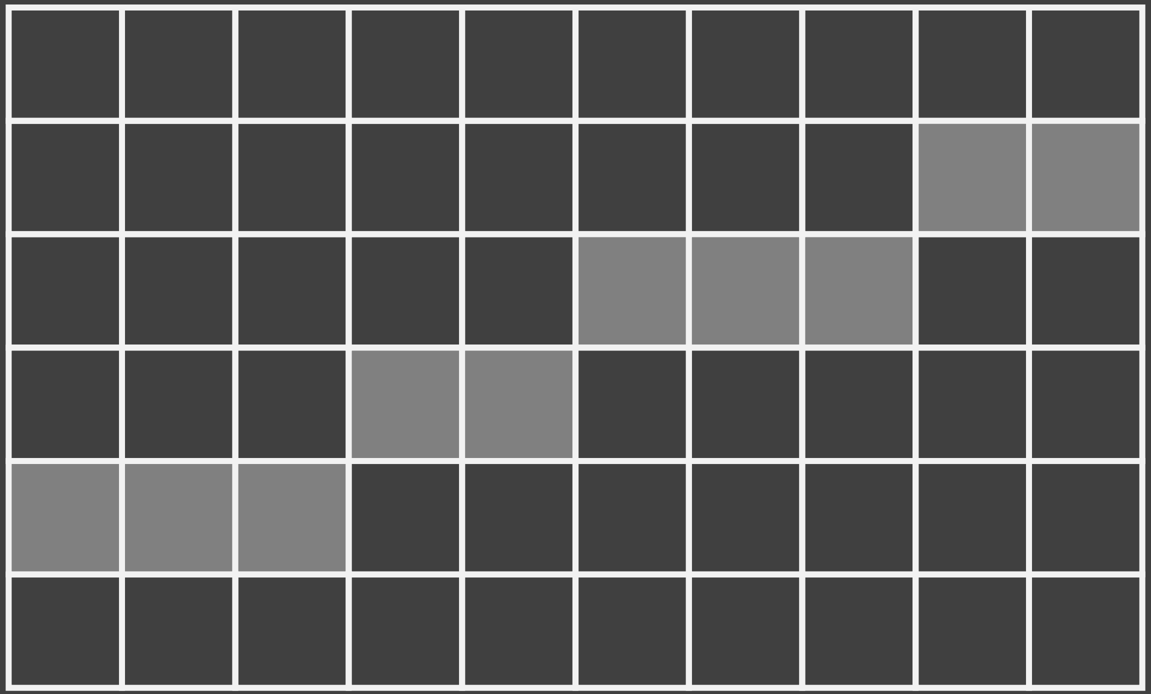
Scan by x

$$x = x_1 + t$$

$$y = \| y_1 + t\,\Delta y \,/\, \Delta x \|$$

$$t = 0, 1, 2, \ldots, \Delta x$$

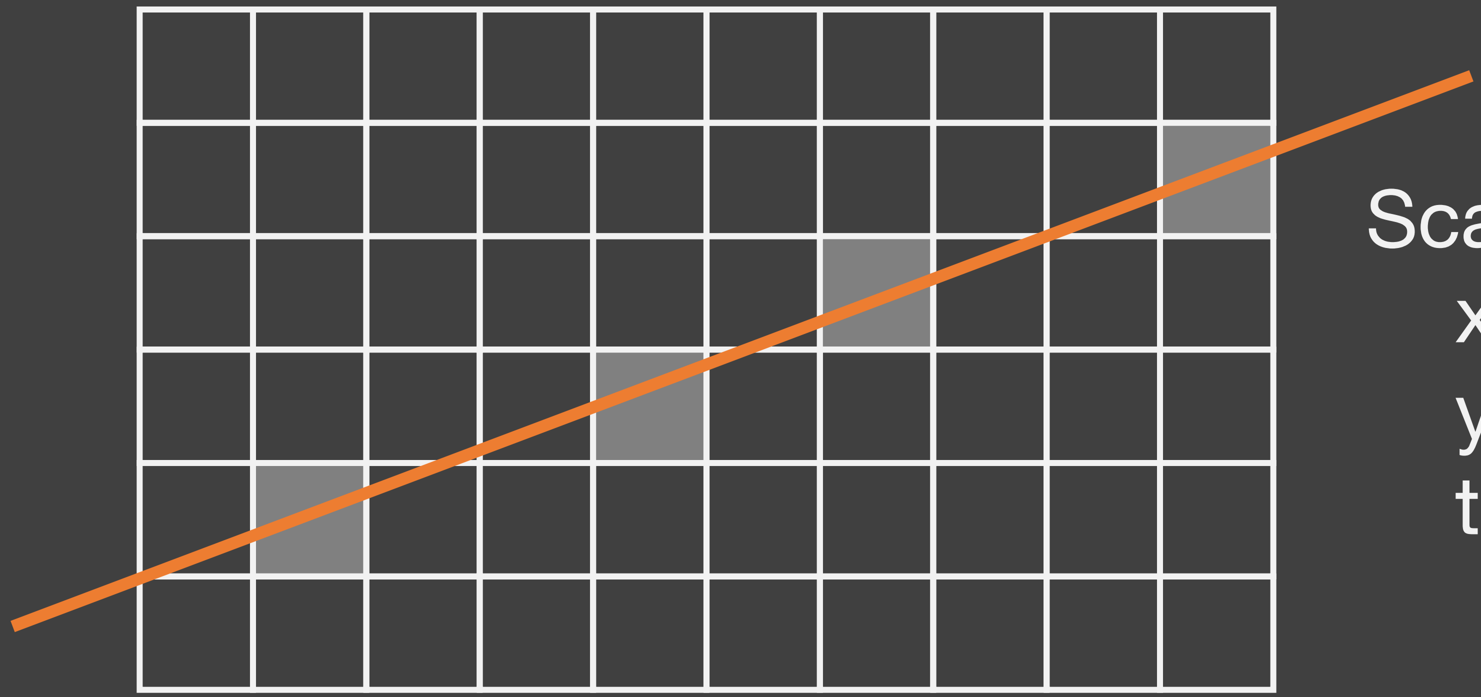
Naïve Approach #2



Looks good!

(but... what if...)

Naïve Approach #2



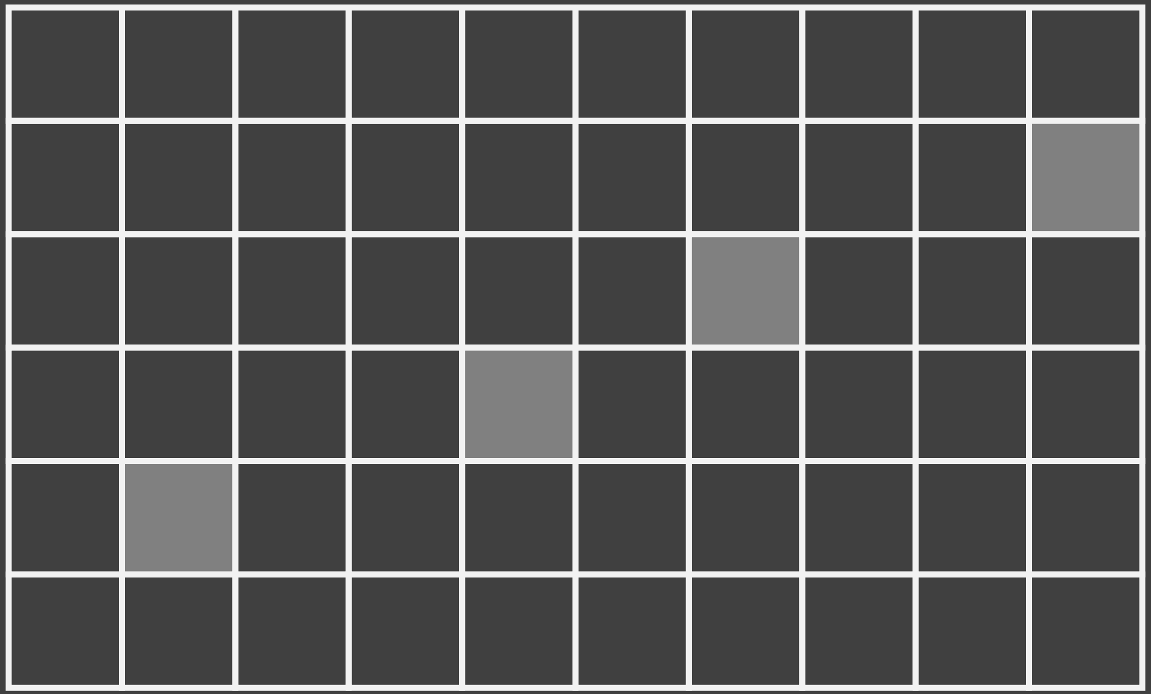
Scan by y?

$$x = \lVert x_1 + t\,\Delta x / \Delta y \rVert$$

$$y = y_1 + t$$

$$t = 0, 1, 2, \ldots, \Delta y$$

Naïve Approach #2



Too Sparse!

(so...)

Scan by the larger one between |Δx| and |Δy|

The naïve solution is

Digital Differential Analyzer (DDA)

It requires floating point arithmetic which is slow

Optimization?

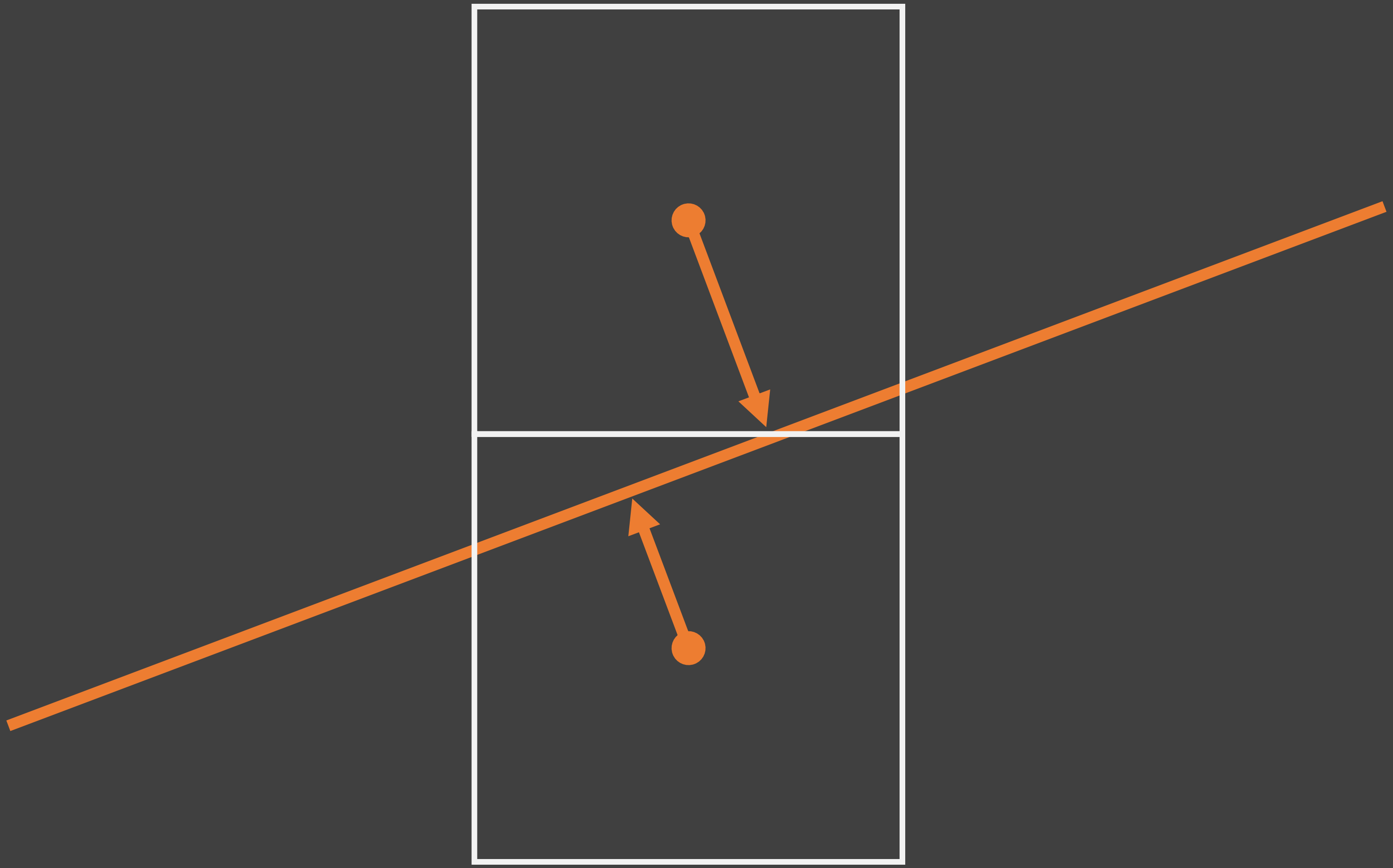
A better approach is

Bresenham's Line Algorithm*

(* Known as an earliest algorithm in computer graphics)

Optimization?

Find the distance between a pixel and the line

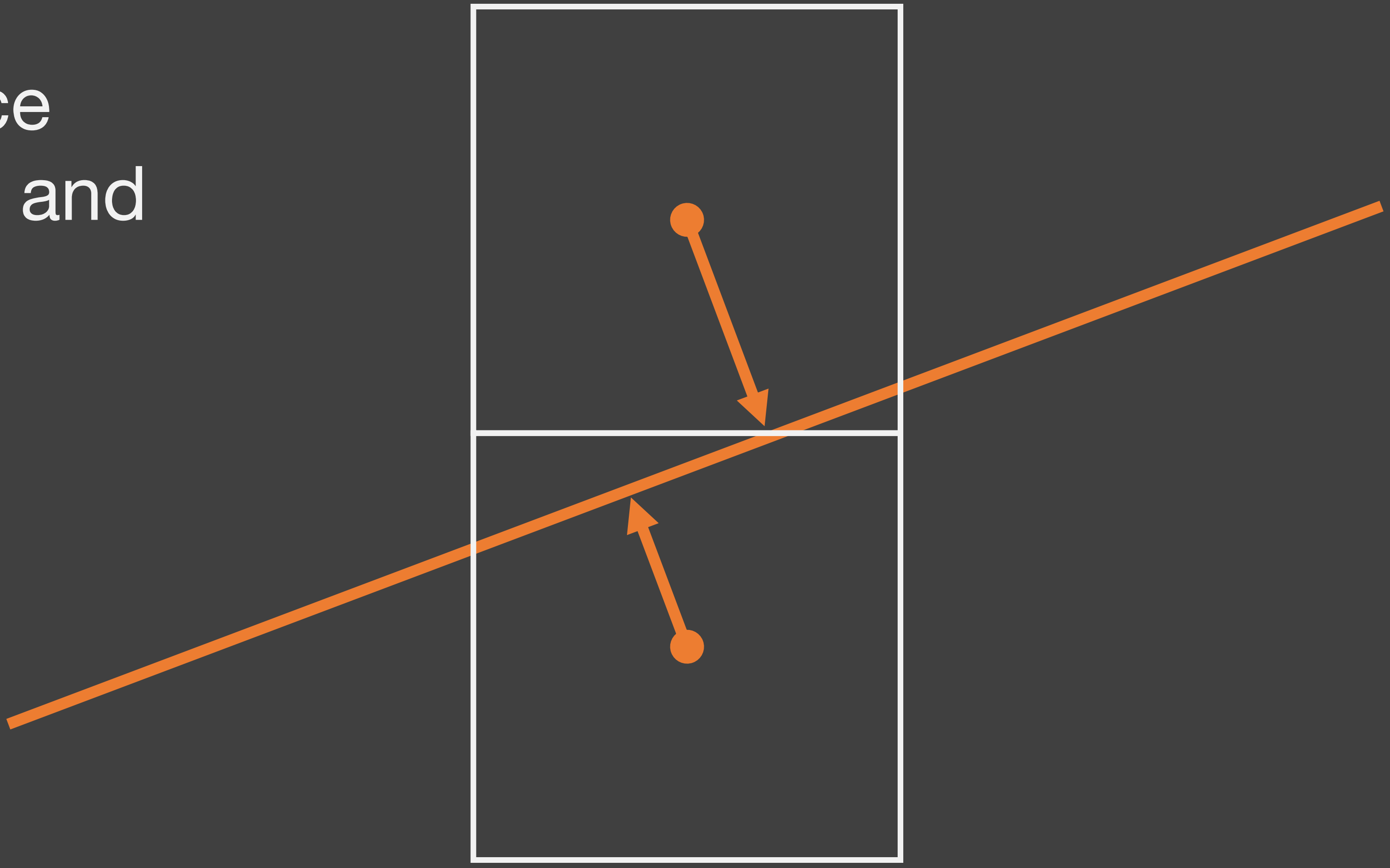


Bresenham's Algorithm

Bresenham's Algorithm

Find the distance between a pixel and the line

Does not calculate the actual distance but do comparison

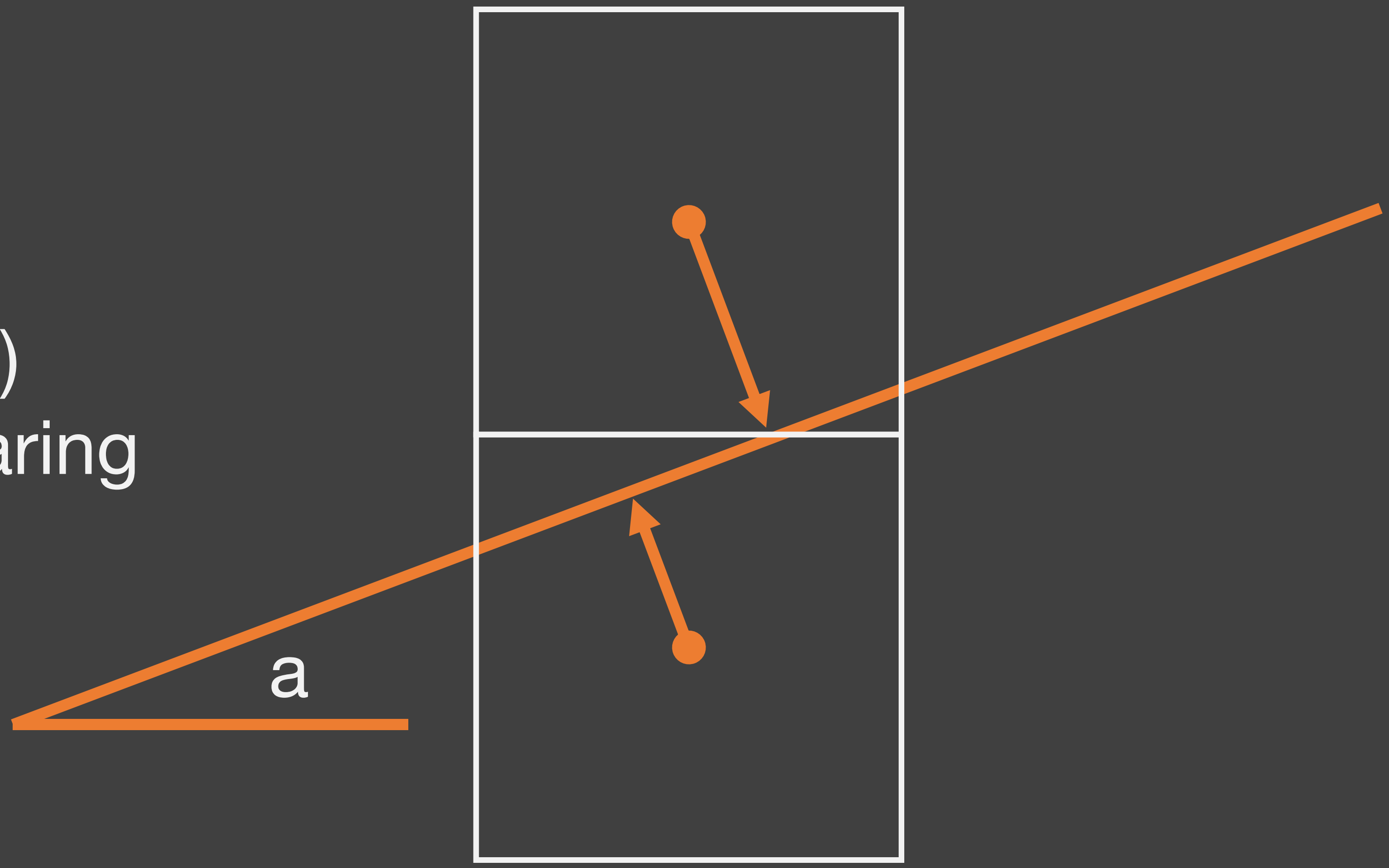


Comparing

$x \sin(a)$ and $y \cos(a)$

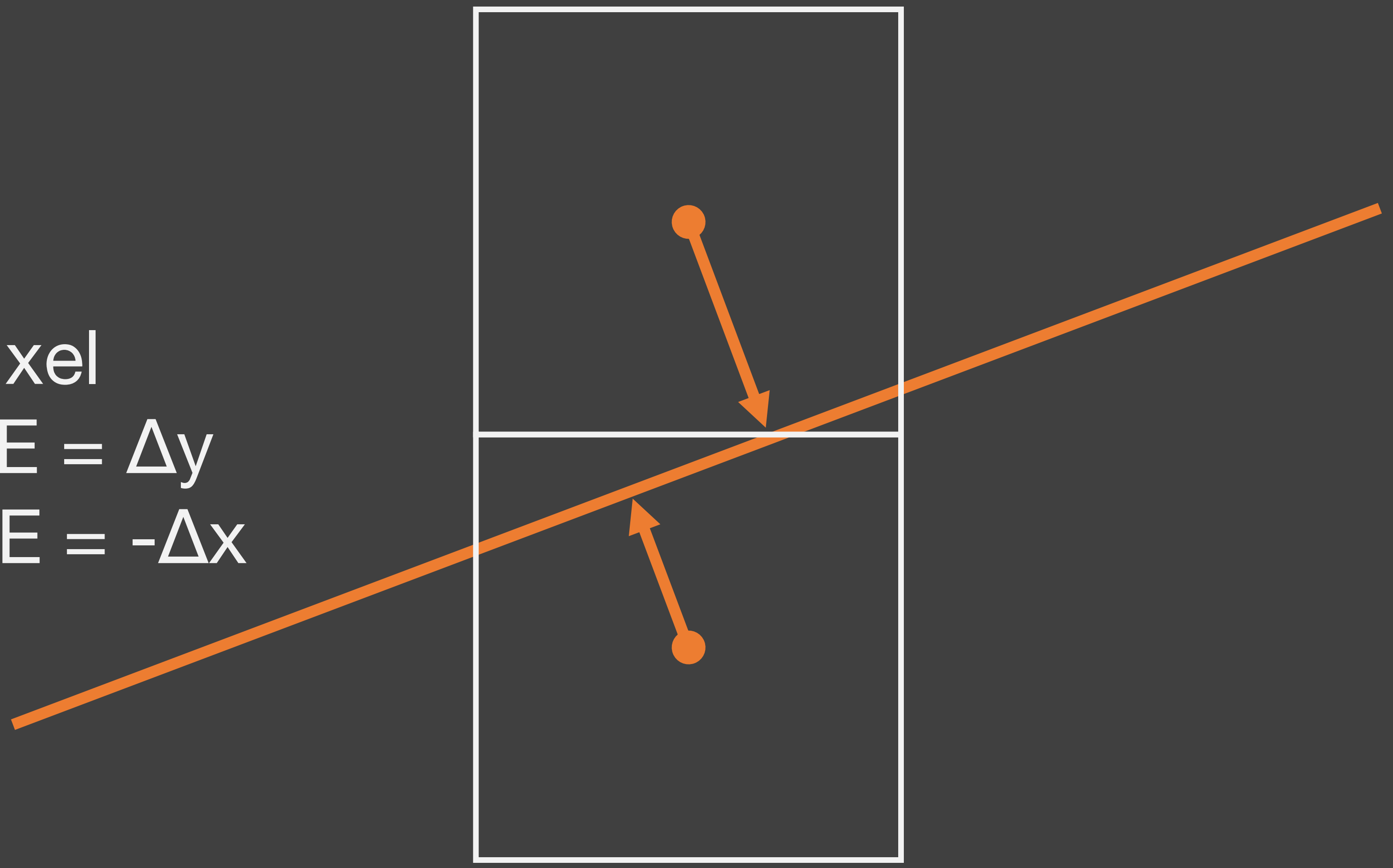
Is the same as comparing

$x\, \Delta y$ and $y\, \Delta x$



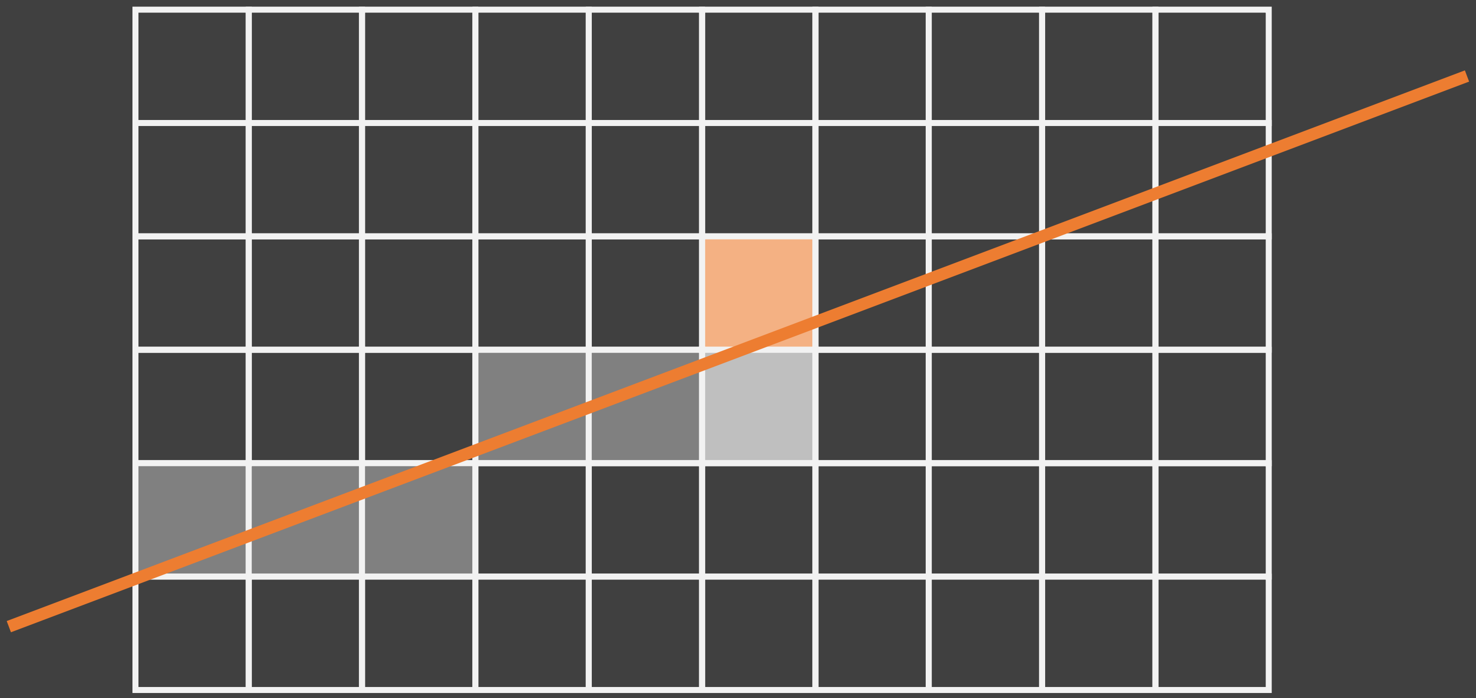
Bresenham's Algorithm

So, define an offset E
When move by one pixel
- X's contribution to E = Δy
- Y's contribution to E = -Δx



Bresenham's Algorithm

Scan by x
In each step, choose the y that makes |E| smallest



Bresenham's Algorithm

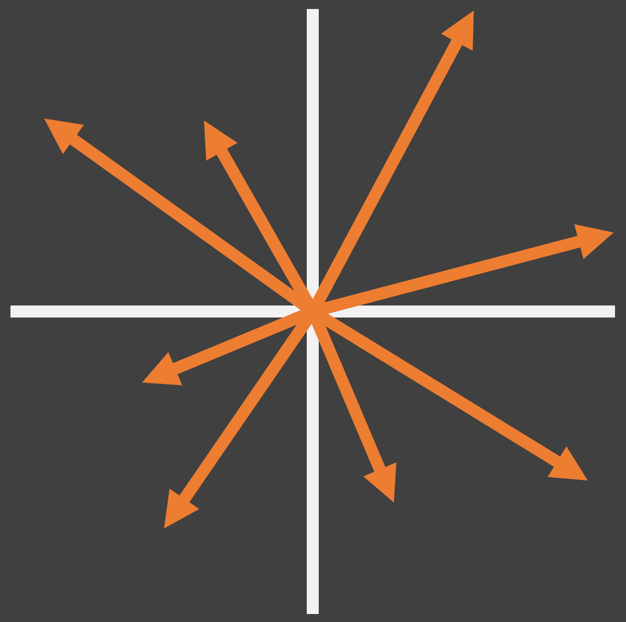
Bresenham's Algorithm Pseudocode

```
dx = x2 - x1
dy = y2 - y1
error = 0

y = y1
for x = x1 to x2 do
    plot(x, y)
    error = error + dy
    while error >= dx / 2 do
        y = y + 1
        error = error - dx
    end while
end for
```

Wait!!
Did we forget something?

(#ProtectYourCervicalSpine)



There are eight cases

$x_1 > x_2$ vs. $x_1 \leq x_2$

$y_1 > y_2$ vs. $y_1 \leq y_2$

$|\Delta x| > |\Delta y|$ vs. $|\Delta x| \leq |\Delta y|$

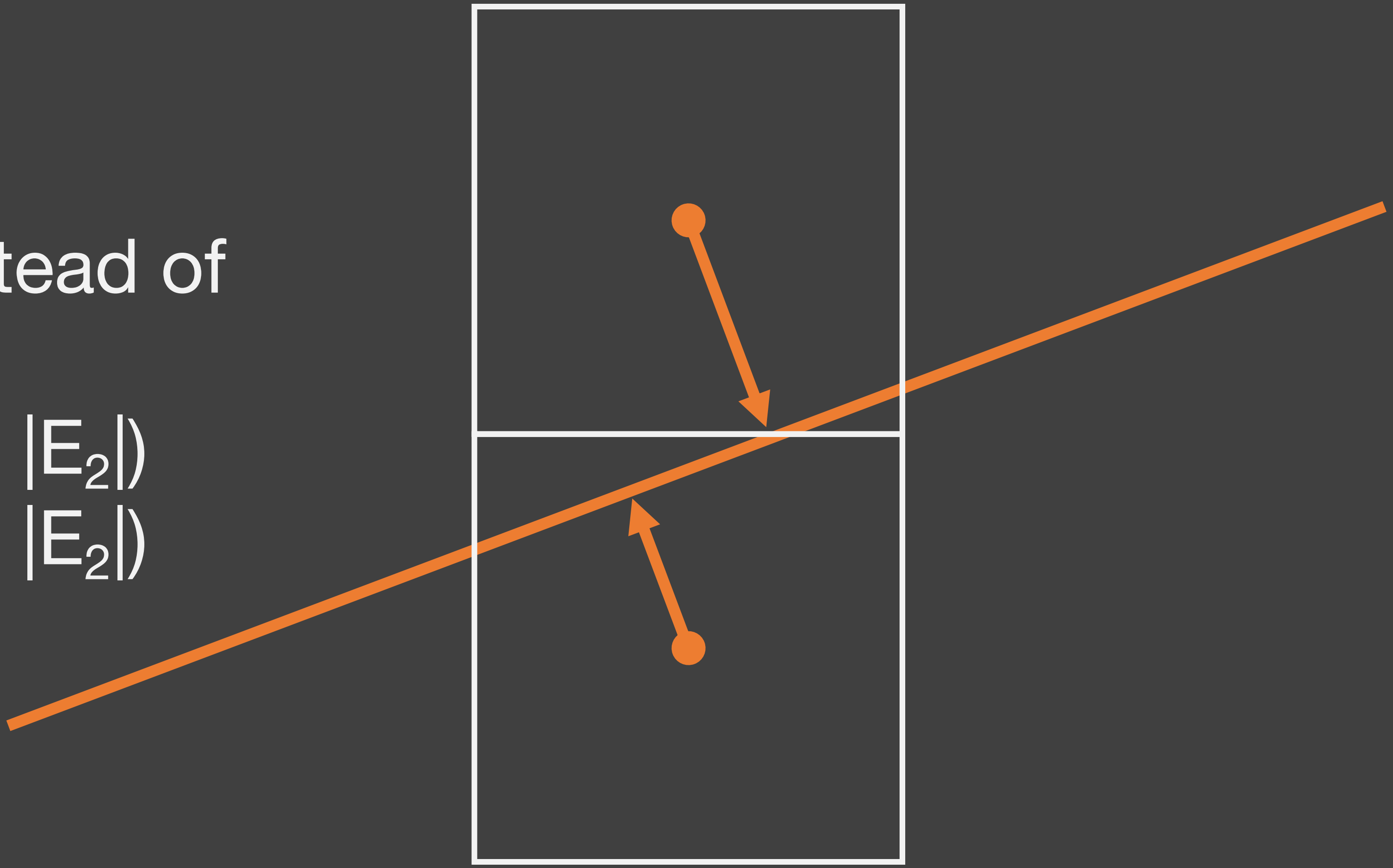
One more thing?

Anti-aliasing!

One m thing?

Render two pixels instead of choosing one

$$\text{color}_1 = |E_2| \,/\, (|E_1| + |E_2|)$$
$$\text{color}_2 = |E_1| \,/\, (|E_1| + |E_2|)$$



Xiaolin Wu's Algorithm

Line Drawing and Collision Detection

CTU Open 2011
Simple Polygon

Give points of a polygon

Points are given in form of (X_i, Y_i)
X_i and Y_i are integers, $1 \leq X_i, Y_i \leq 30000$

Determine if it is self-intersecting

CTU Open 2011
Simple Polygon

There are ≤ 150 test cases
In each test case, the number of points ≤ 40000
CPU time limit is 4s

Simple $O(n^2)$ solution will cause TLE
$40000 \times 40000 \times 150 = 2.4 \times 10^{11}$

CTU Open 2011
Simple Polygon

Solution: Collision detection
HCZ's approach: Line drawing

Draw each line discretely
Check if two line intersect only if they share the same pixel

CTU Open 2011
Simple Polygon

There are two tricky points

The original size is 30000 × 30000
To draw the lines faster, downscale it

(#Magic)
Choosing a proper downscale rate is important

CTU Open 2011
Simple Polygon

There are two tricky points

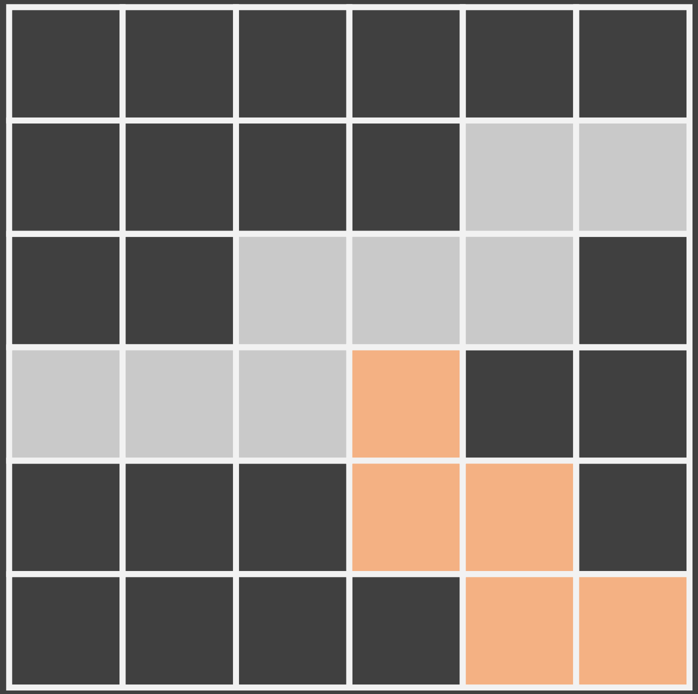
Lines may intersect without pixel intersection

It sounds tricky...
What happened there?

CTU Open 2011
Simple Polygon

CTU Open 2011
Simple Polygon

Create a "wall" to solve this problem



Modified Bresenham's Algorithm

```
dx = x2 - x1
dy = y2 - y1
error = 0

x = x1
y = y1
while x != x2 and y != y2 do
    plot(x, y)
    error1 = abs(error + dy)
    error2 = abs(error - dx)
    if error1 < error2 then
        x = x + 1
        error = error1
    else
        y = y + 1
        error = error2
    end if
end while
plot(x, y)
```