Lecture 2: Bresenham's Line Algorithm

Drawing Straight with Square Pixels

CS 453 — Computer Graphics

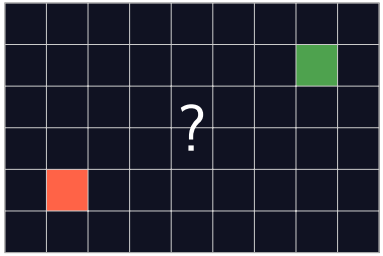
Fakhir Shaheen

Forman Christian University

Recap

- an image is a grid: `img[y][x]`, origin **top-left**, y down
- a color is three numbers: (R, G, B), each 0–255
- lerp: $C = A + t\,(B - A)$ — gradients

Today, One Question Only

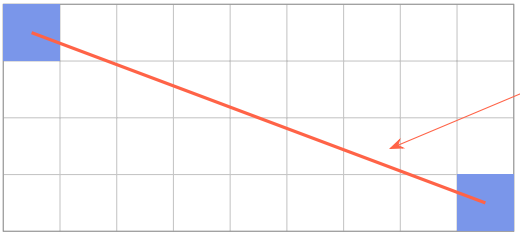


Given two pixels, which pixels in between should light up?

The Problem

The True Line vs. Our Grid

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



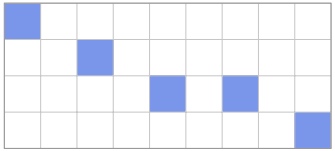
$(x_1, y_1) = (8, 3)$

The math line is **continuous**.

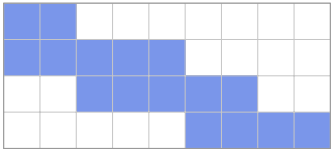
Pixels are **all-or-nothing**.

Rasterizing = deciding, pixel by pixel: *on* or *off*.

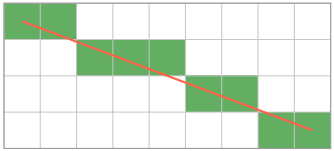
Three Possible Answers



× **gaps**



× **fat**



✓ **just right**

"Just right" is easy to point at.

Let's pin down what it promises.

The Contract

A good line-drawing routine guarantees:

- ✓ **both endpoints** lit
- ✓ **connected** — no holes
- ✓ **thin** — one pixel per column (or per row, if steep)
- ✓ **hugs** the ideal line
- ✓ **fast** — a screen redraws millions of pixels per frame

First Attempts (and How They Fail)

Attempt 1: Straight from Math Class

Assumption: `set_pixel(img, x, y, color)` does `img[y][x] = color`, silently skipping off-image coordinates *(part of A1)*

$y = mx + b$ — so walk x, compute y, round:

Attempt 1: Straight from Math Class

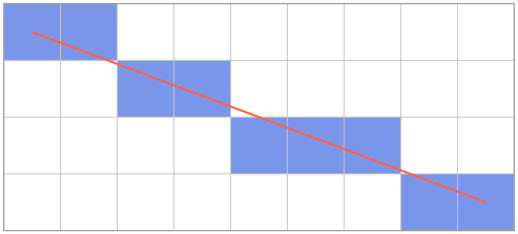
Assumption: `set_pixel(img, x, y, color)` does `img[y][x] = color`, silently skipping off-image coordinates *(part of A1)*

$y = mx + b$ — so walk x, compute y, round:

```
m = (y1 - y0) / (x1 - x0)
b = y0 - m * x0                    # how?
for x in range(x0, x1 + 1):
    y = int(m * x + b + 0.5)       # why add 0.5, int()?
    set_pixel(img, x, y, color)
```

(+ 0.5 then int = round to the nearest row; remember this trick)

First Test: Looks Perfect!

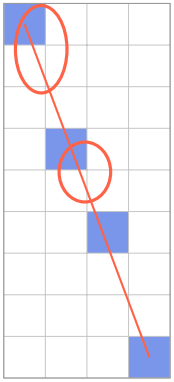


$(0, 0) \rightarrow (8, 3)$

✓ connected, thin, hugs the line.

Try more cases?

The Same Code on a Steep Line



$(0, 0) \to (3, 8)$

Gaps!

The loop runs once per **column**:

4 columns $\Rightarrow$ 4 pixels.

But the line crosses **9 rows**.

Slope $m = \frac{8}{3} \approx 2.7$: every step in x jumps almost 3 rows at once.
One pixel per column is only right when $|m| \leq 1$.

It Gets Worse: the Vertical Line

A vertical line has $x_1 = x_0$ ...

```
m = (y1 - y0) / (x1 - x0)
```

```
ZeroDivisionError
```

Crashed before drawing a single pixel.

And Worse Still: Right to Left

Endpoints given right-to-left, $x_1 < x_0$ ...

```
1 range(x0, x1 + 1)     # empty!
```

The loop body never runs.

Nothing at all is drawn — silently.

No crash, no error — the worst kind of bug.

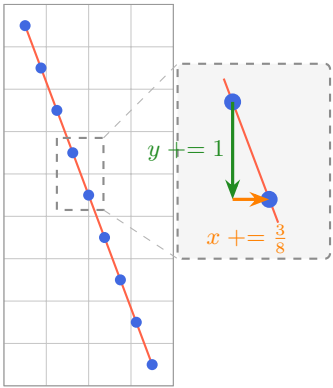
Attempt 1: the Scorecard

shallow, left→right	✓
steep	gaps
vertical	crash
right→left	blank

The lesson

"Loop over x" silently assumes x is the axis the line mostly travels along. Sometimes it isn't.

The Fix, in Pictures: Equal Steps Along the Line



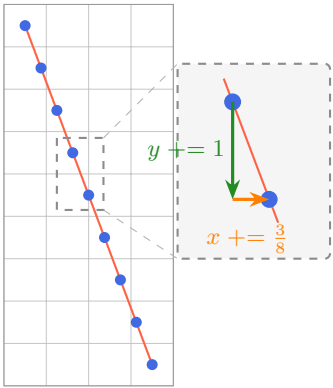
$(0, 0) \to (3, 8)$

The line spans **3** columns but **8** rows — y is the longer axis.

So take **8 equal steps**. Per step:

- y moves $8/8 = 1$ — one full row
- x moves $3/8$ — a *fraction* of a column

The Fix, in Pictures: Equal Steps Along the Line



$(0,0) \to (3,8)$

The line spans **3** columns but **8** rows — y is the longer axis.

So take **8 equal steps**. Per step:

- y moves $8/8 = 1$ — one full row
- x moves $3/8$ — a *fraction* of a column

That puts **9 evenly spaced points** *exactly on the true line* — one per row.

Round each to its nearest pixel. No row skipped, ever: neither increment exceeds 1.

Attempt 2: Walk the *Longer* Axis ("DDA")

Count steps along whichever axis spans **more** pixels:

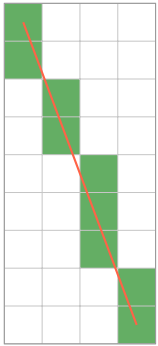
```
steps = max(abs(x1 - x0), abs(y1 - y0))
x_inc = (x1 - x0) / steps    # each <= 1
y_inc = (y1 - y0) / steps    # each <= 1

x, y = x0, y0
for i in range(steps + 1):
    set_pixel(img, int(x + 0.5), int(y + 0.5), color)
    x += x_inc
    y += y_inc
```

Each increment is ≤ 1, so no axis ever skips a pixel. Signs come along free.

DDA = (Digital) Differential Analyzer, an early computer that integrated by repeated adding

DDA: the Scorecard



the steep line, fixed

shallow ✓

steep ✓

vertical ✓

right→left ✓

9 steps, 9 pixels, no gaps —
one per **row** this time.

All four cases work. Are we done?

What DDA Pays Per Pixel

Look at the inner loop. Per pixel:

```
for i in range(steps + 1):
    set_pixel(img, int(x + 0.5), int(y + 0.5), color)
    x += x_inc
    y += y_inc
```

2 float adds + 2 float roundings

A full-HD frame has ∼2 million pixels,

redrawn **60 times a second**.

And in 1965, when this problem was first solved, there was no float hardware *at all*.

The Sneakier Problem: Reproducibility

Float rounding is subtle:

- machines and languages can disagree on **ties** by one pixel
- graders, GPUs, and plotters all need *exact, reproducible* answers — the same line must give the same pixels, everywhere, every time

The challenge

Same pixels as DDA — using **only integers**:

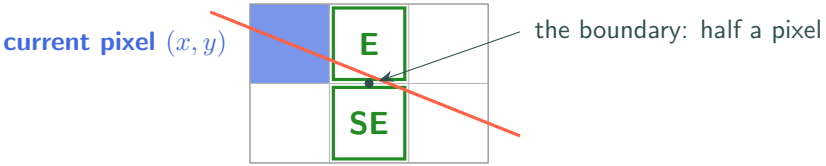
add, subtract, compare. No multiply in the loop. No divide *anywhere*.

Jack Bresenham solved exactly this — and sixty years later it is still the standard answer.

Bresenham's Idea

Insight 1: There Are Only Two Candidates

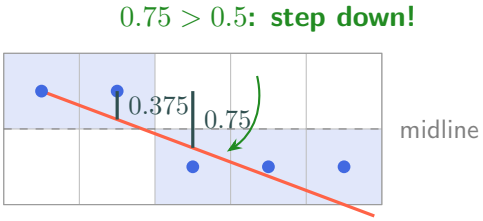
Shallow case ($0 \leq m \leq 1$), moving left to right, y growing down:



Next column, the line drops by $m \leq 1$. The next pixel is

E $(x{+}1,\ y)$ or **SE** $(x{+}1,\ y{+}1)$ — nothing else.

Insight 2: Track the *Miss*, Not the Position



Keep one number, `err`: how far the true line has **crept below** the row we are painting.

Each column it grows by m.

Past **half a pixel** $\Rightarrow$ drop a row, measure from there.

Insight 2, as Code

```
m = dy / dx          # shallow: m <= 1
err = 0.0
y = y0
for x in range(x0, x1 + 1):
    set_pixel(img, x, y, color)
    err += m
    if err > 0.5:    # past the midline?
        y += 1       # step down (SE)
        err -= 1.0   # measure from the
                     # new row *
```

✓ same pixels as DDA

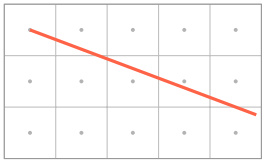
× still floats: `dy/dx`, `0.5`, `1.0`

* `err -= 1`: just changed rows, `err` is measured from my row, so re-measure: subtract 1.

One step from the finish line: get rid of the fractions.

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

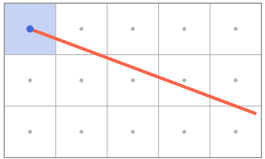


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

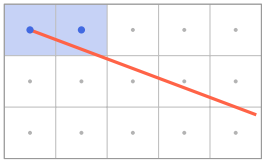


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

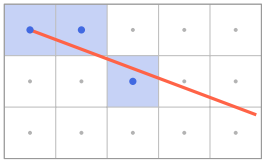


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	
1	$(1,0)$	0.75	yes	$y = 1$, `err` $= -0.25$

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

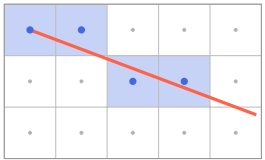


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	
1	$(1,0)$	0.75	yes	$y = 1$, `err` $= -0.25$
2	$(2,1)$	0.125	no	

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

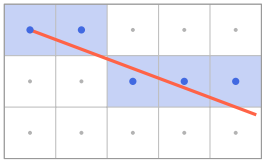


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	
1	$(1,0)$	0.75	yes	$y = 1$, `err` $= -0.25$
2	$(2,1)$	0.125	no	
3	$(3,1)$	0.5	no — a tie stays	

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.

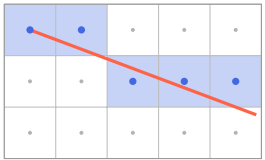


first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	
1	$(1,0)$	0.75	yes	$y = 1$, `err` $= -0.25$
2	$(2,1)$	0.125	no	
3	$(3,1)$	0.5	no — a tie stays	
4	$(4,1)$	0.875	yes	$y = 2$, `err` $= -0.125$

Trace It: `err` Doing Its Job

Line $(0,0) \to (8,3)$, so $m = 3/8 = 0.375$. Start: `y = 0`, `err = 0`. Each column: paint, then `err += m`; if `err > 0.5`: `y += 1` *and* `err -= 1` — always together.



first five columns

x	paint	`err += 0.375`	past 0.5?	bookkeeping
0	$(0,0)$	0.375	no	
1	$(1,0)$	0.75	yes	$y = 1$, `err` $= -0.25$
2	$(2,1)$	0.125	no	
3	$(3,1)$	0.5	no — a tie stays	
4	$(4,1)$	0.875	yes	$y = 2$, `err` $= -0.125$

`y` holds whole rows, `err` the leftover fraction: y + `err` $= 0.375\,x$ — **at every step.**
Stepping down just moves 1.0 from one pocket to the other.

Change the Units, Keep the Answer

Who is taller — $1.7\,\text{m}$ or $1.5\,\text{m}$?

Measure the same two people in centimeters: 170 vs 150.

Same winner. Always.

Change the Units, Keep the Answer

Who is taller — $1.7\,\text{m}$ or $1.5\,\text{m}$?

Measure the same two people in centimeters: 170 vs 150.

Same winner. Always.

Multiplying *both* sides of a comparison by the same **positive** number never flips it.

Change the Units, Keep the Answer

Who is taller — $1.7\,\text{m}$ or $1.5\,\text{m}$?

Measure the same two people in centimeters: 170 vs 150.

Same winner. Always.

Multiplying *both* sides of a comparison by the same **positive** number never flips it.

And our loop never uses `err`'s *value* — it only ever asks one comparison:

`err > 0.5` $\xrightarrow{\times 16}$ `16*err > 8` same answer, every column $\Rightarrow$ **same pixels**.

Shopping for a Multiplier

The fractions to clear (our line: $dy = 3$, $dx = 8$): $\frac{3}{8}, \quad 0.5, \quad 1$

Shopping for a Multiplier

The fractions to clear (our line: $dy = 3$, $dx = 8$): $\frac{3}{8}$, 0.5, 1

try $\times 8$: 3 ✓ 4 ✓ 8 ✓ lucky! $dx = 7$ gives $0.5 \times 7 = 3.5$ $\times$

Shopping for a Multiplier

The fractions to clear (our line: $dy = 3$, $dx = 8$): $\frac{3}{8}$, 0.5, 1

try $\times 8$:	3 ✓	4 ✓	8 ✓	lucky! $dx = 7$ gives $0.5 \times 7 = 3.5$ $\times$
try $\times 2$:	0.75 $\times$	1 ✓	2 ✓	the $/dx$ survives

Shopping for a Multiplier

The fractions to clear (our line: $dy = 3$, $dx = 8$): $\frac{3}{8}$, 0.5, 1

try $\times 8$:	3 ✓	4 ✓	8 ✓	lucky! $dx = 7$ gives $0.5 \times 7 = 3.5$ $\times$
try $\times 2$:	0.75 $\times$	1 ✓	2 ✓	the $/dx$ survives
try $\times \mathbf{2\,dx}$:	$\mathbf{6}$ ✓	$\mathbf{8}$ ✓	$\mathbf{16}$ ✓	**whole, for every line**

The **2** kills the 0.5; the $\mathbf{dx}$ kills the $/dx$. In general:

$$\frac{dy}{dx} \times 2\,dx = 2\,dy \qquad 0.5 \times 2\,dx = dx \qquad 1 \times 2\,dx = 2\,dx$$

No division survives — integers in, integers forever.

Insight 3: Scale Until the Fractions Vanish

`err` only ever meets three values:

$$m = \frac{dy}{dx}, \quad 0.5, \quad 1.$$

Multiply *everything* by $2\,dx$:

float version		**integer version**
`err += dy/dx`	$\longrightarrow$	`E += 2*dy`
`err > 0.5`	$\longrightarrow$	`E > dx`
`err -= 1.0`	$\longrightarrow$	`E -= 2*dx`

Scaling both sides of a comparison by the same positive number never changes the *decision*. That's the whole trick.

The Integer Loop

```
E, y = 0, y0
for x in range(x0, x1 + 1):
    set_pixel(img, x, y, color)
    E += 2 * dy
    if E > dx:
        y += 1
        E -= 2 * dx
```

✓ identical pixels

✓ integers only

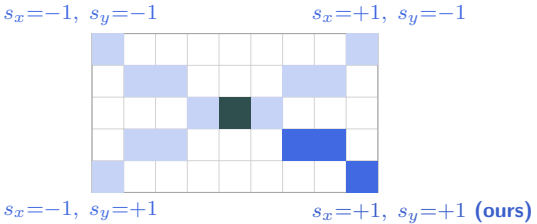
✓ add & compare in the loop

× One limitation left: shallow only, left-to-right only.

One Loop for All Eight Directions

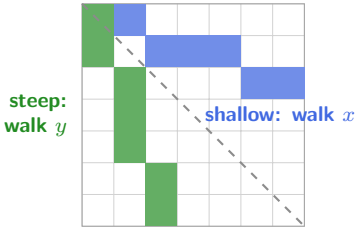
Our loop only draws **one kind** of line: rightward, downward, shallow. Two mirrors buy the rest.

Mirror 1: flip the step signs



same pixels, reflected — reverse the endpoints, or step y by -1

Mirror 2: swap $x \leftrightarrow y$

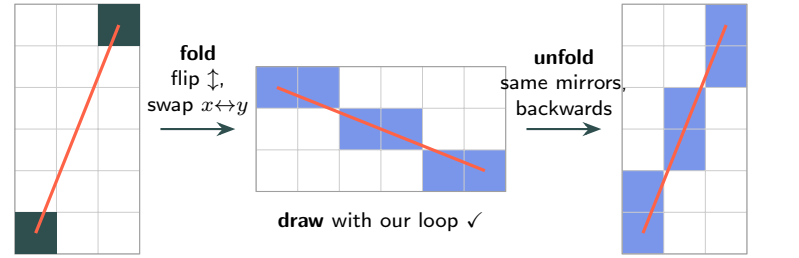


if $dy > dx$: walk rows, nudge columns — same loop, letters swapped

4 sign choices $\times$ 2 walk axes $= 8$ octants — **still one loop**. Let's build it.

Fold, Draw, Unfold

We can only draw one octant: rightward, downward, shallow. Want this steep, *upward* line?



Mirrors undo themselves: fold the *endpoints* in, draw, unfold the *pixels* out.

In code: two little swaps fold the endpoints in, and a swapped `set_pixel` unfolds each pixel out.

Coming up.

When to Fold? Ask the Endpoints

Three yes/no questions — each a single comparison of the endpoints:

question	**test**	**if yes, fold**
going left?	`x1 < x0`	swap the endpoints
going up?	`y1 < y0`	step y with `sy = -1`
steep?	`abs(y1-y0) > abs(x1-x0)`	swap $x \leftrightarrow y$ in both

Our example $(0, 5) \to (2, 0)$: left? $2 \geq 0$: **no**. up? $0 < 5$: **yes** $\Rightarrow$ `sy = -1`. steep? $5 > 2$: **yes** $\Rightarrow$ swap.

Already shallow, rightward, downward? Three no's — **the fold does nothing.**

The Algorithm — One Loop, All Eight Octants

```
1  def bresenham_line(img, x0, y0, x1, y1, color):
2      steep = abs(y1 - y0) > abs(x1 - x0)
3      if steep:                  # fold: swap x <-> y
4          x0, y0, x1, y1 = y0, x0, y1, x1
5      if x0 > x1:                # fold: left to right
6          x0, y0, x1, y1 = x1, y1, x0, y0
7      dx = x1 - x0
8      dy = abs(y1 - y0)
9      sy = 1 if y0 < y1 else -1
10     E, y = 0, y0
11     for x in range(x0, x1 + 1):
12         if steep:                        # unfold
13             set_pixel(img, y, x, color)
14         else:
15             set_pixel(img, x, y, color)
16         E += 2 * dy
17         if E > dx:
18             y += sy
19             E -= 2 * dx
```

- lines 2–6: the **fold** — two endpoint swaps, nothing else
- the loop *is* the integer loop you traced — only `y += sy` is new
- `if steep`: paint (y, x) — the **unfold**, pixel by pixel
- ties (`E == dx`) stay — same rule as your trace

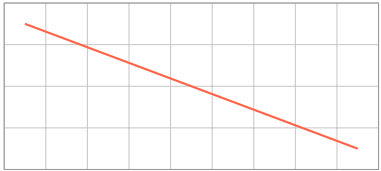
Learn this exact variant

Friday's assignment will ask for precisely this form — other correct variants differ on tie pixels.

Let's Run It by Hand: $(0, 0) \to (8, 3)$

steep? $3 > 8$: no. left→right already. $dx = 8,\ dy = 3,\ s_y = +1,\ \texttt{E} = 0$

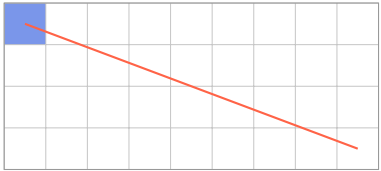
plot	`E += 6`	`E > 8?`	then



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8,\ dy = 3,\ s_y = +1,\ \texttt{E} = 0$

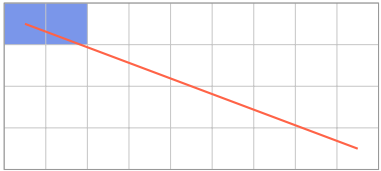
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

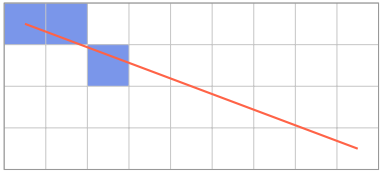
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

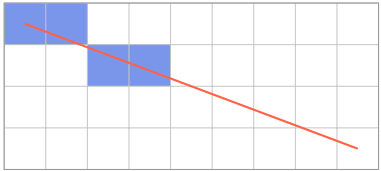
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$
$(2,1)$	2	no	



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

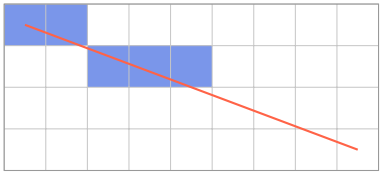
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$
$(2,1)$	2	no	
$(3,1)$	8	no — a tie stays	



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

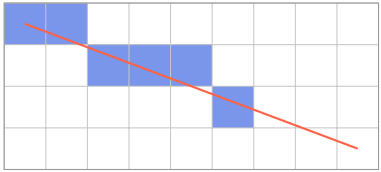
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$
$(2,1)$	2	no	
$(3,1)$	8	no — a tie stays	
$(4,1)$	14	**yes**	$y = 2$, $\texttt{E} = -2$



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\mathtt{E} = 0$

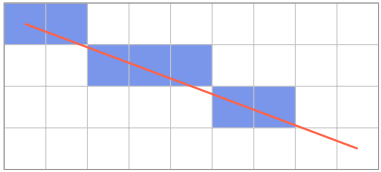
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\mathtt{E} = -4$
$(2,1)$	2	no	
$(3,1)$	8	no — a tie stays	
$(4,1)$	14	**yes**	$y = 2$, $\mathtt{E} = -2$
$(5,2)$	4	no	



Let's Run It by Hand: $(0, 0) \to (8, 3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

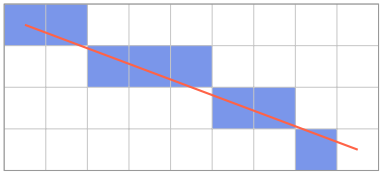
plot	`E += 6`	`E > 8?`	then
$(0, 0)$	6	no	
$(1, 0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$
$(2, 1)$	2	no	
$(3, 1)$	8	no — a tie stays	
$(4, 1)$	14	**yes**	$y = 2$, $\texttt{E} = -2$
$(5, 2)$	4	no	
$(6, 2)$	10	**yes**	$y = 3$, $\texttt{E} = -6$



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\texttt{E} = 0$

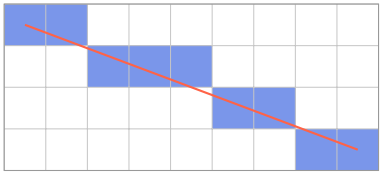
plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\texttt{E} = -4$
$(2,1)$	2	no	
$(3,1)$	8	no — a tie stays	
$(4,1)$	14	**yes**	$y = 2$, $\texttt{E} = -2$
$(5,2)$	4	no	
$(6,2)$	10	**yes**	$y = 3$, $\texttt{E} = -6$
$(7,3)$	0	no	



Let's Run It by Hand: $(0,0) \to (8,3)$

steep? $3 > 8$: no. left→right already. $dx = 8$, $dy = 3$, $s_y = +1$, $\mathtt{E} = 0$

plot	`E += 6`	`E > 8?`	then
$(0,0)$	6	no	
$(1,0)$	12	**yes**	$y = 1$, $\mathtt{E} = -4$
$(2,1)$	2	no	
$(3,1)$	8	no — a tie stays	
$(4,1)$	14	**yes**	$y = 2$, $\mathtt{E} = -2$
$(5,2)$	4	no	
$(6,2)$	10	**yes**	$y = 3$, $\mathtt{E} = -6$
$(7,3)$	0	no	
$(8,3)$	x reached x_1 — loop ends		



row runs of 2, 3, 2, 2 pixels — evenly spread, never uniform

Do this trace yourself for $(0,0) \to (5,2)$ before Friday — it is the fastest way to *own* the loop.

One Pixel Differs from Attempt 1 — On Purpose

Column 4: Attempt 1 put it on row 2.

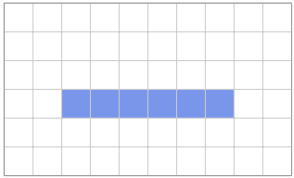
Bresenham keeps it on row 1.

The true line hits that pixel corner *exactly* — a **tie**.

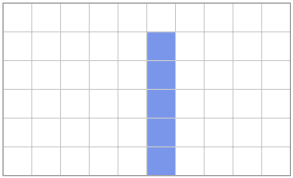
Tie-breaking is what separates the correct variants from each other.

That is why Friday's assignment will pin *one* variant: so everyone's pixels agree on every tie.

The Awkward Cases — No Extra Code Needed

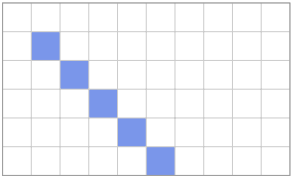


horizontal

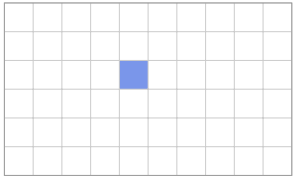


vertical

$dy = 0$: E stays 0 — y never steps, so it folds into a horizontal walk



diagonal

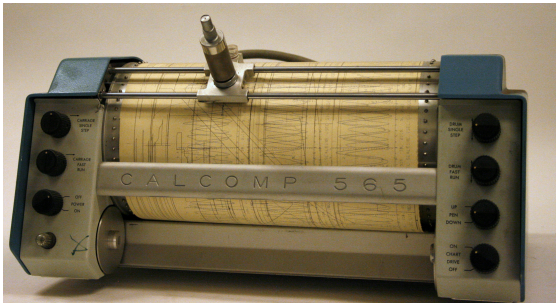


a point

$dy = dx$: E passes dx every column the loop runs exactly once

No division anywhere $\Rightarrow$ nothing to crash.

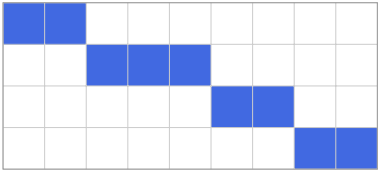
1962: Why Integers Were Not Optional



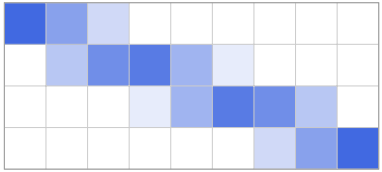
- Jack Bresenham, IBM, 1962 — published 1965
- target: the **Calcomp 565 pen plotter**, whose pen moves in fixed $1/100''$ steps in 8 directions — *exactly* our E / SE step
- the computer driving it had no float hardware; every cycle counted

Sixty years on, the same loop sits in GPU rasterizers, plotters, CNC machines and `painter.py`. Good constraints age well.

The Price of All-or-Nothing: Jaggies



`bresenham_line`: on / off



anti-aliased line: partial coverage

The staircase ("**aliasing**") is the cost of yes/no pixels.

The fix: pay each pixel in proportion to how much of the line covers it.

The version on the right is Xiaolin Wu's algorithm (1991). We will **deliberately not lecture it**: reading and implementing an algorithm from the literature is a skill this course trains. It needs color blending, which we build Friday.

Your Turn

Before Friday

1. Re-do today's trace **on paper** for $(0, 0) \to (5, 2)$ while it is fresh
   — one plotted pixel per loop turn; watch `err` swing between its bounds
2. Write the loop from memory, on paper, then check it against today's slide
   — you will implement it for real on Friday, so make the shape stick now
3. Skim the Wikipedia article *"Xiaolin Wu's line algorithm"*
   — just to see its shape; it needs the blending we build Friday

Exit Ticket

Today's questions *(2 minutes)*

(a) "Loop over x, round y" — why does it leave gaps on steep lines?

(b) For `bresenham_line(img, 4, 9, 1, 7, c)` is it steep? which endpoint does the walk start from? `dx`, `dy`, `sy`?

CS 453 — Exit Ticket #2

Name: _____

Roll no: _____

(a) _____

(b) _____

hand in on your way out

Friday: translucency, circles, and how paint "knows" where to stop: fills.