Finger Exercises 2

Bresenham's Line Algorithm

≈ 50 minutes, or three short sittings

CS 453 — Computer Graphics

What this is: Short drills that lock in Lecture 2 — line drawing, from "walk x, round y" to the full integer Bresenham loop. Nothing needs more than 5 minutes.

How to do it: On **paper** first, no computer. If an item takes longer than 5 minutes, skip it, finish the rest, come back. Afterwards, verifying your answers in Python is excellent practice — every trace here is a five-line program away from checking itself.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. The lecture's contract: a good line routine lights _____, leaves no _____, stays one pixel per _____, hugs the ideal line, and is _____.

A2. For a shallow left-to-right line, after painting pixel (x, y) the next pixel is either _____ or _____ — nothing else. Why can it not be $(x{+}1, y{+}2)$?

A3. In the float loop, `err` measures how far the true line has crept _____ the center of _____. We step down the moment it passes _____.

A4. DDA stands for _____, named for an early machine that integrated by _____.

A5. True or False: Bresenham's loop contains a division, but only outside the inner loop. True / False

Part B • Attempt 1: the Autopsy

CLASSIFY ≈ 5 min

The math-class loop from the lecture, exactly as first written:

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

For each call, tick what actually happens.

Call	good line	gaps	crash / nothing
`line(img, 0, 0, 8, 3, c)`	☐	☐	☐
`line(img, 0, 0, 3, 8, c)`	☐	☐	☐
`line(img, 5, 7, 5, 0, c)`	☐	☐	☐
`line(img, 8, 3, 0, 0, c)`	☐	☐	☐

B1. In one phrase: what hidden assumption do all three failures share? _____

Part C • Round Like a Rasterizer

COMPUTE ≈ 4 min

The lecture's trick: `int(v + 0.5)` rounds $v \geq 0$ to the nearest integer, because `int()` *truncates*.

C1. Evaluate: `int(1.9)` = _____ `int(1.9 + 0.5)` = _____ `int(2.5 + 0.5)` = _____

C2. Python's built-in `round()` gives `round(0.5)` = _____ and `round(1.5)` = _____. Why did the lecture avoid it for a line with slope $\frac{1}{2}$?

C3. The trick has a blind spot: `int(-1.2 + 0.5)` = _____, but the nearest integer to -1.2 is _____. Why does this never bite our line loop?

Part D • Trace the Float Loop

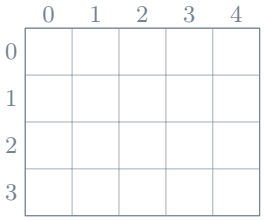
TRACE ≈ 6 min

The Insight-2 loop: start `err = 0`, and each column `err += m`; if `err > 0.5`, step down and `err -= 1`.

D1. Trace $(0,0) \to (4,3)$, so $m = 0.75$. The first row is done for you; remember a *tie* (`err` $= 0.5$ exactly) stays put.

column	`err` after `+= 0.75`	step down?	pixel painted
1	0.75	yes → `err` $= -0.25$	$(1,1)$
2	_____	_____	_____
3	_____	_____	_____
4	_____	_____	_____

Shade the five painted pixels (including the start) on the grid:



D2. After a step down, why `err -= 1`? One sentence — say what moved and what did not.

_____

Part E • Scale the Question

COMPUTE ≈ 5 min

Multiplying both sides of a comparison by the same positive number never changes the answer — so we scale the whole loop by $2\,dx$ until every fraction is gone.

E1. For $dy = 3$, $dx = 8$, fill in the integer twins:

`err += 3/8` → `E +=` _____ `err > 0.5` → `E >` _____ `err -= 1` → `E -=` _____

E2. Check the promise on one real decision. At the lecture's column 2 the float loop held `err` $= 0.75$. Scaled by 16: _____ versus threshold _____ — same verdict?

E3. Why is multiplying by dx alone not safe? Test it on a line with $dx = 7$: the threshold 0.5 becomes _____.

Part F • Find the Bug

FIX ≈ 7 min

Each loop below draws *something* — just not the right thing. Name the bug and give the one-line fix. The first two are shallow left-to-right loops; the third is the start of the full function.

F1. Symptom: the line tracks correctly to the first step down — then turns into a 45° staircase for the rest of its length.

```
err = 0.0
y = y0
for x in range(x0, x1 + 1):
    set_pixel(img, x, y, color)
    err += m
    if err > 0.5:
        y += 1
```

Bug & fix:

_____

_____

F2. Symptom: same 45° staircase — but this one starts at the very first column.

```
E = 0
y = y0
for x in range(x0, x1 + 1):
    set_pixel(img, x, y, color)
    E += 2 * dy
    if E > 0.5:
        y += 1
        E -= 2 * dx
```

Bug & fix:

F3. Symptom: right-to-left lines silently vanish — steep ones, however, are fine.

```
steep = abs(y1 - y0) > abs(x1 - x0)
if steep:
    x0, y0, x1, y1 = y0, x0, y1, x1
dx = x1 - x0
dy = abs(y1 - y0)
sy = 1 if y0 < y1 else -1
E, y = 0, y0
for x in range(x0, x1 + 1):
    ...
```

Bug & fix:

Part G • Fold or Not? Setup Drill

CLASSIFY ≈ 6 min

Before its loop starts, the final function decides: is the line *steep* (fold: swap $x \leftrightarrow y$ in both endpoints)? Do the endpoints need swapping so it walks left to right? Then it reads off dx, dy and s_y, with `E` $= 0$. Fill the table — *after* all folding.

Call	steep?	walks from	dx	dy	s_y
`bresenham_line(img, 2, 3, 9, 6, c)`					
`bresenham_line(img, 7, 2, 3, 10, c)`					
`bresenham_line(img, 4, 9, 4, 1, c)`					
`bresenham_line(img, 0, 0, 5, 5, c)`					

G1. For the diagonal call (row 4): with $dx = dy = 5$, what does `E` do each column, and what does the loop draw?

G2. For the vertical call (row 3): after the folds, what does the loop actually walk — and how do the painted pixels end up vertical?

Part H • Run the Real Loop

TRACE ≈ 7 min

The lecture asked you to trace $(0, 0) \to (5, 2)$ before Friday. Here is the scaffold — so no excuses. Setup: not steep, already left-to-right, $dx = 5$, $dy = 2$, $s_y = +1$, `E` $= 0$. Each turn: plot, then `E += 4`; if `E > 5`: `y += 1` and `E -= 10`.

plot	`E` after `+= 4`	`E > 5`?	then
$(0, 0)$	4	no	—

Part I • One Sentence Each

EXPLAIN ≈ 4 min

I1. Attempt 1 and DDA both draw correct-looking lines on a shallow test. Give the *two* reasons the lecture still threw floats out.

I2. The final loop only ever walks x. How do steep lines still come out right?

I3. The second fold draws a right-to-left segment left to right instead. Why is that legitimate?

I4. Modern GPUs have fast float hardware, yet their rasterizers still snap coordinates to a grid and decide with integer arithmetic. Why?

Stuck on more than three items? Re-read the Lecture 2 slides — every answer above comes straight from them. **Done and confident?** Close everything and, from memory, write the full `bresenham_line` on paper — setup and loop, nineteen lines. Then trace $(0, 0) \to (7, 5)$ through it. If your trace lights $(7, 5)$ last, Friday will feel like more of the same.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides.

A1. both **endpoints**; no **holes** (connected); one pixel per **column — or per row, if steep** (thin); **fast**.

A2. E = $(x{+}1,\ y)$ or SE = $(x{+}1,\ y{+}1)$. A shallow line has slope ≤ 1, so in one column it drops at most one row — it cannot reach two rows down.

A3. **below** the center of **the row we are currently painting**; past **half a pixel** (the midline).

A4. **Digital Differential Analyzer**; by **repeated adding** — exactly what the loop does.

A5. **False** — there is *no divide anywhere*: `dx`, `dy` come from subtraction, and the $\times 2\,dx$ scaling removed every fraction. Add, subtract, compare — that is the whole loop.

B. Row 1: **good** — shallow, left to right is the one case it handles. Row 2: **gaps** — steep: one pixel per column, but the line crosses several rows per column. Row 3: **crash** — vertical means `x1 - x0 = 0`, a `ZeroDivisionError` before the loop even starts. Row 4: **nothing** — `range(8, 1)` is empty, so the loop body never runs; no error, no pixels, the worst kind of bug.

B1. "Loop over x" assumes x is the axis the line mainly travels along (and travels left to right) — sometimes it is not.

C1. `1` (truncation drags down); `2`; `3` — the trick sends exact halves *up*.

C2. `0` and `2` — banker's rounding sends halves to the nearest *even* integer. A slope of 1/2 hits an exact half every other column, so the staircase comes out lumpy (runs of 2, 1, 3, 1) instead of even.

C3. `int(-0.7)` = `0`, but the nearest integer is -1: truncation goes *toward zero*, so the trick only rounds correctly for $v \geq 0$. Our rows and columns are never negative — off-image writes are dropped before this matters.

D1. Column 2: `err` $= -0.25 + 0.75 = 0.5$ — a tie, stays, paints $(2, 1)$. Column 3: `err` $= 1.25$, steps $(\rightarrow 0.25)$, paints $(3, 2)$. Column 4: `err` $= 1.0$, steps $(\rightarrow 0)$, paints $(4, 3)$ — the far endpoint, as it must. Full pixel list: $(0, 0)$, $(1, 1)$, $(2, 1)$, $(3, 2)$, $(4, 3)$.

D2. The line never moved — *our measuring row* dropped by exactly one row, so the measured gap "line minus my row" must drop by 1 too. (Ladder version: climb down one rung and a mark on the wall sits 1 higher *relative to you*.)

E1. `E += 6` $(= 2\,dy)$; `E > 8` $(= dx)$; `E -= 16` $(= 2\,dx)$.

E2. 12 versus 8 — and $12 > 8$ exactly as $0.75 > 0.5$: same verdict, same pixel. Scaling a comparison by a positive number is like measuring both heights in centimeters instead of meters.

E3. $0.5 \times 7 = 3.5$ — still fractional whenever dx is odd. The extra factor of 2 exists purely to kill the 0.5; the dx kills the $/dx$ in the slope.

F1. The step down never pays for itself: `err -= 1` is missing, so once `err` crosses 0.5 it stays there and *every* later column steps down. Add `err -= 1` inside the `if`.

F2. Half-converted: the adds were scaled by $2\,dx$ but the threshold was not — in the scaled world the midline is `dx`, not `0.5`. Since `E` jumps to $2\,dy$ immediately, `E > 0.5` fires on column one. Fix: `if E > dx`. When you scale a comparison, you must scale *both* sides.

F3. The second fold is missing: nothing guarantees $x_0 \leq x_1$, so for a right-to-left call `range(x0, x1 + 1)` is empty and the loop never runs — Attempt 1's "blank" bug, reborn. Fix: after the steep swap, add `if x0 > x1:` swap the two endpoints. (Steep right-to-left lines survived only because the first swap happened to put their x in order.)

G. Row 1: not steep, no swap — walks from $(2, 3)$; $dx = 7$, $dy = 3$, $s_y = +1$: the home octant, untouched. Row 2: steep $(8 > 4)$ — swapping $x \leftrightarrow y$ gives $(2, 7) \rightarrow (10, 3)$, already left-to-right; $dx = 8$, $dy = 4$, $s_y = -1$. Row 3: steep $(8 > 0)$ — the swap gives $(9, 4) \rightarrow (1, 4)$, which then needs the endpoint swap: walks from $(1, 4)$; $dx = 8$, $dy = 0$, $s_y = -1$ (harmless — y never steps). Row 4: $5 > 5$ is false, so *not* steep — walks from $(0, 0)$; $dx = 5$, $dy = 5$, $s_y = +1$.

G1. `E` jumps by $2\,dy = 10$, which passes $dx = 5$ *every* column — so y steps every turn and `E` nets $10 - 10 = 0$. The loop draws the perfect staircase, no special case anywhere.

G2. It walks the folded x from 1 to 9 with $dy = 0$, so `E` stays 0 and y stays 4 — a horizontal walk. The `steep` unfold paints `set_pixel(img, y, x)`, flipping each pixel back to column 4, rows 1..9: the vertical line that crashed Attempt 1 is now just another line.

H. Plot $(1, 0)$: `E` $= 8 > 5$ — step, $y = 1$, `E` $= -2$. Plot $(2, 1)$: `E` $= 2$, no. Plot $(3, 1)$: `E` $= 6 > 5$ — step, $y = 2$, `E` $= -4$. Plot $(4, 2)$: `E` $= 0$, no. Plot $(5, 2)$: x reached x_1 — the loop ends. Pixels: $(0, 0)$, $(1, 0)$, $(2, 1)$, $(3, 1)$, $(4, 2)$, $(5, 2)$ — row runs of 2, 2, 2: both endpoints lit, connected, thin.

I1. Cost — two float adds and two roundings per pixel, millions of pixels per frame (and in 1965, no float hardware at all); and reproducibility — float rounding can differ by one pixel across machines, while graders, GPUs and plotters need the same line to give the same pixels everywhere, every time.

I2. They are *folded* first — swapping $x \leftrightarrow y$ in both endpoints turns a steep line into a shallow one, so the folded x *is* the true long axis; the swapped `set_pixel` unfolds every pixel back as it is painted.

I3. A segment is a *set* of pixels, not a direction: same endpoints, same folded walk, same pixels either way — and walking left to right turns Attempt 1's silent empty-`range` bug into a non-issue.

I4. Determinism, not speed: integer tests give bit-identical answers regardless of evaluation order, so two triangles sharing an edge produce exactly matching pixels — no cracks, no double-drawn seams. Bresenham's real legacy is exactness, and it is still inside every GPU.