Finger Exercises 1

What Graphics Is; First Pixels & Gradients

≈ 50 minutes, or three short sittings

CS 453 — Computer Graphics

What this is: Short drills that lock in Lecture 1. 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. Most items are three lines of code away from checking themselves.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Fill in the blanks. A *pixel* is short for _____. An image of resolution $W \times H$ contains _____ pixels, so a 600×400 image contains _____ of them.

A2. Pixel $(0, 0)$ sits at the _____ corner of the image; x grows _____ and y grows _____ — flipped from math class, because screens count rows the way you read.

A3. Each color channel is one _____, holding a value from _____ to _____. Three channels give $256^3 \approx$ _____ million colors. A color with all three channels equal looks _____.

A4. Circle **True** or **False**:

(a) Screens mix colors the same way paint does. True / False

(b) In `img[y][x]`, the first index picks the *row*. True / False

(c) A binary PPM file stores its pixels compressed, like PNG. True / False

(d) In plain Python, `img[-1][0]` raises an `IndexError`. True / False

(e) "Visit every pixel, compute its color" is the skeleton of every renderer we build this semester. True / False

Part B • What Color Is That Pixel?

TRACE ≈ 7 min

B1.

```
navy = (16, 18, 34)
red  = (255, 0, 0)
# 8 wide, 6 tall
img = [[navy] * 8 for _ in range(6)]
img[2][5] = red
```

Color at coordinate $(x, y) = (5, 2)$: _____

Color at coordinate $(x, y) = (2, 5)$: _____

B2.

```
img = [[(0, 0, 0)] * 7 for _ in range(7)]
for y in range(7):
    for x in range(7):
        img[y][x] = (40 * x, 0, 40 * y)
```

Color at $(3, 0)$: _____

Color at $(0, 6)$: _____

Color at $(6, 6)$: _____

Does the *blue* channel rise left→right or top→bottom? _____

B3. Match each snippet to the picture it paints. Assume the loop `for y ...for x ...` visits every pixel of a $W \times H$ image, and A, B are two different colors.

```
(i)   img[y][x] = A if y < H // 2 else B
(ii)  img[y][x] = lerp_color(A, B, x / (W - 1))
(iii) img[y][x] = A if (x // 8 + y // 8) % 2 == 0 else B
```

(P) a checkerboard of 8-pixel squares: _____ (Q) a smooth left-to-right fade: _____ (R) two solid horizontal bands: _____

B4.

```
navy = (16, 18, 34)
img = [[navy] * 640 for _ in range(420)]
print(len(img), len(img[0]))
```

It prints: _____

Which number is the width? _____

Part C • Bytes on Disk

COMPUTE ≈ 7 min

Recall from the slides: a P6 file is a text header, then 3 raw bytes per pixel, row after row from the top — so pixel (x, y) starts at byte $3\,(y \cdot W + x)$ of the pixel data.

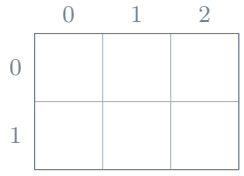
C1. A 600×400 image. How many pixels, and how many *bytes of pixel data* does its P6 file contain?
pixels: _____ pixel-data bytes: _____

C2. In a 640-wide image, at which byte of the pixel data does pixel $(x, y) = (10, 5)$ begin?
byte: _____

C3. Same image. Byte 9634 of the pixel data belongs to which pixel — and which channel of it?
pixel $(x, y) =$ _____ channel: _____

C4. A complete P6 file begins `P6`, then `3 2`, then `255`, and its pixel data is the 18 bytes
`255 0 0  0 255 0  0 0 255  0 0 0  255 255 255  128 128 128`

Label every cell of the 3×2 image with its color *name* (red, gray, . . .):



Which pixel is white? $(x, y) =$ _____

Total file size, header included: _____ bytes

Part D • Lerp Until It Is Reflex

COMPUTE ≈ 6 min

$\text{lerp}(a, b, t) = a + t\,(b - a)$: at $t = 0$ you get a, at $t = 1$ you get b. Say it out loud: "lerp".

D1. Evaluate, no calculator:
$\text{lerp}(10, 30, 0.25) =$ _____ $\text{lerp}(0, 255, 0.2) =$ _____ $\text{lerp}(40, 20, 0.5) =$ _____

D2. $A = (0, 0, 0)$, $B = (200, 100, 50)$.
$\text{lerp\_color}(A, B, 0.5) =$ _____ $\text{lerp\_color}(A, B, 0.1) =$ _____

D3. The lecture's sky: a 6-row image fades from `(20,30,120)` at row 0 to `(240,150,60)` at row 5, using `t = y / (H - 1)`. For row $y = 3$:
$t =$ _____ red channel $= \text{lerp}(20, 240, t) =$ _____

D4. An image 101 rows tall gets the lecture's full-height gradient, color A at the top row, B at the bottom row. At which row is the color *exactly* halfway between A and B? _____

D5. Going backwards: for what t does $\text{lerp}(20, 240, t) = 108$? _____

Part E • Find the Bug

FIX ≈ 8 min

Each snippet compiles and runs — and each one has bitten a real student in a real week 1. Name the bug and give the one-line fix.

E1. Symptom: the student sets *one* pixel, but a full-height red line appears at column 3.

```
navy = (16, 18, 34)
img = [[navy] * 640] * 420
img[0][3] = (255, 0, 0)
```

Bug & fix:
_____

_____

E2. Symptom: crashes with `IndexError` partway through the loop.

```python
img = [[(0, 0, 0)] * 640 for _ in range(420)]
for y in range(420):
    for x in range(640):
        img[x][y] = (x % 256, y % 256, 0)
```

Bug & fix:

E3. Symptom: a student writes a helper meant to paint a solid $w \times h$ block with top-left (x, y) — but asking for a 4×3 block colors a 5×4 one.

```python
def rect(img, x, y, w, h, color):
    for yy in range(y, y + h + 1):
        for xx in range(x, x + w + 1):
            img[yy][xx] = color
```

Bug & fix:

E4. Symptom: shapes pushed off the *left* edge mysteriously reappear on the *right* edge.

```python
def put_pixel(img, x, y, color):
    w, h = len(img[0]), len(img)
    if x < w and y < h:
        img[y][x] = color
```

Bug & fix:

E5. Symptom: two students both code the lecture's gradient. Their images look *identical to the eye*, yet comparing the files byte-for-byte shows some channels differ by exactly 1. Student one wrote the slide's `lerp_color`; student two "improved" it:

```python
# student one (the slide):
channel = int(a + t * (b - a))
# student two:
channel = round(a + t * (b - a))
```

What happened?

Part F • Will It Draw?

CLASSIFY ≈ 4 min

Suppose you finish the safe helper from E4: `put_pixel(img, x, y, c)` writes the pixel only when (x, y) is actually on the image, and `rect(img, x, y, w, h, c)` paints its $w \times h$ block by calling `put_pixel` in the fixed E3 loop. The image is 640×420 (columns 0..639, rows 0..419). Tick one column per call.

Call	draws a pixel	silently ignored
`put_pixel(img, 0, 0, c)`	☐	☐
`put_pixel(img, 639, 419, c)`	☐	☐
`put_pixel(img, 640, 419, c)`	☐	☐
`put_pixel(img, -1, 5, c)`	☐	☐
`put_pixel(img, 5, 420, c)`	☐	☐
`put_pixel(img, 3, -1, c)`	☐	☐

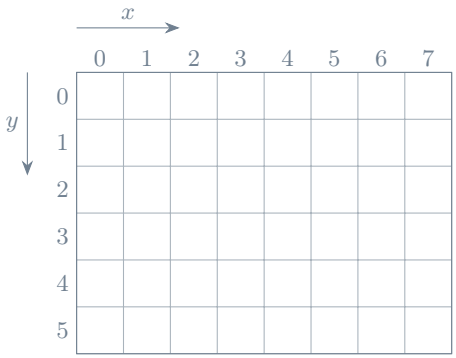
F1. `rect(img, 630, 410, 20, 20, c)` on that same image — how many pixels actually get colored? _____

F2. `rect(img, -3, -3, 5, 5, c)` — how many pixels now? _____

Part G • Design on Paper

DESIGN ≈ 8 min

G1. Shade exactly the cells that `rect(img, 5, 2, 4, 3, c)` (Part F's clipped rectangle helper) colors on this 8×6 image, then count them.



Pixels colored: _____

G2. Write the one-line loop body that turns a $W \times H$ image into a *horizontal* grayscale gradient: pure black at the left column, pure white at the right column, every row identical.

```
for y in range(H):
    for x in range(W):
        img[y][x] = ____________________________________
```

G3. Using *only* `rect` calls, draw a 2-pixel-thick red border around the full edge of a 640×420 image. What is the fewest number of calls, and what are they?

calls needed: _____

_____

_____

G4. Write a one-line expression for the loop body that paints *vertical stripes* 4 pixels wide, alternating colors A and B, starting with A at the left edge.

`img[y][x] =` _____

Part H • Lerp Tricks You Will Want on Friday

COMPUTE ≈ 5 min

The slides used `lerp_color` to fade a sky — but nothing says its two endpoints must be a top color and a bottom color. Lerping a pixel's *current* color partway toward some other color is how mist, glow and glass will be faked all semester. You can already compute all of it.

H1. Mist. A night-sky pixel is $(20, 30, 120)$. Move it 30% of the way toward white $(255, 255, 255)$ — that is, `lerp_color` of the two at $t = 0.3$. Work out all three channels, using the slide's `int(...)` version.

result = _____

H2. Glow, twice. You move a pixel *halfway* toward white, then take the result and move it halfway toward white *again*. Is that the same as going all the way to white? If not, what single t is it equivalent to?

_____

H3. Detective work. A vertical gradient runs from color A at row 0 to an unknown color B at row 4 of a 5-row image ($t = y/4$). You are told $A = (20, 20, 20)$, and row 1 came out as $(75, 60, 45)$. Recover B.

$B =$ _____

Part I • One Sentence Each

EXPLAIN ≈ 4 min

I1. Why is the indexing `img[y][x]` — row first — rather than `img[x][y]`? (Hint: think about the file on disk.)

_____

I2. Part F's `rect` contains no bounds-checking code of its own, yet never crashes at the image edges. Why is that a *good* design, not luck?

_____

I3. In the gradient mapping `t = y / (H - 1)`, why divide by $H - 1$ and not H?

_____

I4. A friend says: "PNG is a fundamentally different kind of thing from your list of rows." Correct them in one sentence.

Stuck on more than three items? Re-read the Lecture 1 slides — everything above comes straight from them. **Done and confident?** On paper, from memory, write the ten-or-so lines that produce the lecture's `setup_ok` gradient image and save it as a PPM. If you can do that, Friday's assignment 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. **pic**ture **el**ement; $W \cdot H$ pixels; $600 \cdot 400 = 240{,}000$.

A2. **top-left**; x grows **rightward**; y grows **downward**.

A3. one **byte**; `0` to `255`; $\approx$ **16.7** million; equal channels look **gray** (with `(0,0,0)` black and `(255,255,255)` white at the ends).

A4. (a) **False** — screens *emit* light, so colors add; paint absorbs. (b) **True** — row first, then column. (c) **False** — PPM is a text header plus raw bytes, a memory dump; PNG and JPEG mostly *compress* those same bytes. (d) **False** — it silently wraps around to the *last* row, which is why the lecture said keeping writes on-image is *your* job. (e) **True**.

B1. $(5, 2)$ is **red**: `img[2][5]` means row 2, column 5, i.e. coordinate $(x, y) = (5, 2)$. $(2, 5)$ is still **navy** — that is row 5, column 2 (`img[5][2]`), which was never written. If you gave the same answer twice, re-read the `img[y][x]` slide before Friday.

B2. $(3, 0) = (120, 0, 0)$; $(0, 6) = (0, 0, 240)$; $(6, 6) = (240, 0, 240)$. Blue is $40y$, so it rises **top to bottom** — reading axes off an image like this is the "reading an image like a graphics programmer" skill from the `setup_ok` slide.

B3. P–(iii), Q–(ii), R–(i). The give-aways: (i) depends only on y with a hard test, so bands; (ii) depends only on x *smoothly*, so a fade; (iii) mixes both coordinates with integer division, so squares.

B4. `420 640`. The width is **640** (`len(img[0])`, the columns in one row); `len(img)` counts *rows*, the height. We *say* "640 by 420" but Python hands the sizes back height-first — mixing the two orders up is a classic week-1 bug.

C1. 240,000 pixels; $3 \times 240{,}000 = 720{,}000$ bytes (plus a header of about 15 characters — the header is text, the rest is a memory dump).

C2. $3\,(5 \cdot 640 + 10) = 3 \cdot 3210 =$ **9630**.

C3. $9634 = 3 \cdot 3211 + 1$, so it is pixel index 3211, channel offset 1 = **green**. And $3211 = 5 \cdot 640 + 11$, so the pixel is $(x, y) = (11, 5)$ — the pixel right after the one in C2.

C4. Top row, left to right: red, green, blue. Bottom row: black, white, gray. White is at $(x, y) = (1, 1)$ — byte count 12..14 is pixel index 4, and $4 = 1 \cdot 3 + 1$. File size: the header `P6, 3 2, 255` with its three newlines is 11 bytes, plus 18 pixel bytes = **29 bytes**.

D1. 15; 51; 30 — note the last one: lerp works perfectly well "downhill" when $b < a$.

D2. $(100, 50, 25)$ — the exit-ticket question — and $(20, 10, 5)$. Each channel travels the same fraction t of its own distance.

D3. $t = 3/5 = 0.6$; red $= 20 + 0.6 \cdot 220 =$ **152**. (Green: $30 + 0.6 \cdot 120 = 102$; blue: $120 - 0.6 \cdot 60 = 84$ — the whole row is $(152, 102, 84)$.)

D4. Row **50**: $t = 50/(101 - 1) = 0.5$. Halfway down means halfway through the color — that linearity is the whole point of lerp.

D5. $t = (108 - 20)/(240 - 20) = 88/220 =$ **0.4**. "Undoing" a lerp like this is called *inverse lerp*; you will meet it again when we hit curves and textures.

E1. `[[navy] * 640] * 420` makes 420 *aliases of one single row*, so writing any pixel writes it in every row at once. Fix: build each row fresh with the slide's comprehension — `[[navy] * 640 for _ in range(420)]`. The lecture flagged this exact trap; it bites someone every year.

E2. The indices are swapped: it should be `img[y][x]` — row first. It crashes the moment `x` reaches 420, because there is no row 420 in a 420-row image. (On a *square* image this bug does not crash at all — it silently transposes your picture, which is far nastier. Non-square test images are your friend.)

E3. Off-by-one: `range(y, y + h)` *already* covers the h rows $y \,..\, y + h - 1$ — Python ranges exclude their end — so the `+ 1`s paint an extra row and column. Drop both. Off-by-ones at rectangle edges are the classic raster bug; train your eye for them now.

E4. Only the *upper* bounds are checked. A negative `x` slips through, and Python's negative indexing wraps it to the other side of the row. Fix: `if 0 <= x < w and 0 <= y < h`. One correct bounds check like this, at the bottom of your code, keeps everything you draw safe at the edges — remember it on Friday.

E5. `int()` *truncates* while `round()` rounds to nearest: for a value like 166.67 they give 166 and 167. Both images are fine art — but they are not the same file. The lesson for graded work: when a written spec gives an exact formula, implement *that* formula, because "equally reasonable" is not "equal".

F. Draws: rows 1 and 2 only — $(0, 0)$ and $(639, 419)$ are the two extreme *valid* corners. All four others are ignored: 640 and 419 fail because valid columns stop at 639 and valid rows at 419; the negative ones must be caught explicitly or Python wraps them ($x = -1$ would paint the right edge, $y = -1$ the bottom row).

F1. $10 \times 10 =$ **100**. The rectangle *wants* columns 630..649 and rows 410..429, but only 630..639 and 410..419 exist. Note `rect` itself needs no bounds logic — it draws through `put_pixel`, which clips.

F2. $2 \times 2 =$ **4**: of columns $-3..1$ only 0..1 survive, same for rows. A rectangle hanging off the top-left keeps only

its bottom-right corner.

G1. Columns 5..7, rows 2..4 — **9** pixels. The rectangle asks for columns 5..8, but column 8 does not exist on an 8-wide image, so the right column clips away. (Nothing clips vertically: rows 2..4 all exist.)

G2. `g = int(255 * x / (W - 1))` then `img[y][x] = (g, g, g)` — or as one line, `img[y][x] = (int(255 * x / (W - 1)),) * 3`. The essentials: it must depend on x (not y), divide by $W - 1$ (so the last column reaches exactly 255), and set all three channels equal (grays).

G3. **4** calls: top strip `rect(img, 0, 0, 640, 2, red)`, bottom strip `rect(img, 0, 418, 640, 2, red)`, left strip `rect(img, 0, 0, 2, 420, red)`, right strip `rect(img, 638, 0, 2, 420, red)`. The corners are painted twice — perfectly fine, since the writes are opaque and identical. Composing big things from a few primitive calls is how every scene gets built.

G4. `A if (x // 4) % 2 == 0 else B`. Integer division groups columns into blocks of 4; the parity of the block index alternates the color. Compare B3's checkerboard — same trick, one axis instead of two.

H1. $(90, 97, 160)$. Red: $20 + 0.3 \cdot 235 = 90.5 \to 90$; green: $30 + 0.3 \cdot 225 = 97.5 \to 97$; blue: $120 + 0.3 \cdot 135 = 160.5 \to 160$. The pixel keeps its character but pales — that *is* mist.

H2. No. After the first step the pixel is $\frac{1}{2}$ of the way there; the second step covers half of the *remaining* distance, landing at $t = 0.75$ — not 1.0. Each pass halves what is left, so repeated soft passes approach white but never reach it. Layered translucent effects behave exactly like this.

H3. $B = (240, 180, 120)$. Row 1 sits at $t = 1/4$, so each channel has covered a quarter of its journey: $B = A + 4\,(C - A)$, e.g. red $= 20 + 4 \cdot 55 = 240$. Running a lerp backwards like this is how you will debug gradients whose endpoints "look wrong".

I1. Because the image is stored, in memory and in the PPM file, as one row after another (row-major order) — so the outer index naturally selects a row, and pixel (x, y) lands at byte $3\,(y \cdot W + x)$.

I2. It draws through `put_pixel`, whose clipping silently drops any off-image pixel — one correct check at the bottom protects every drawing function built on top of it, so the bounds logic exists in exactly one place.

I3. The last row has index $H - 1$, so dividing by $H - 1$ makes t reach exactly 1.0 there; dividing by H would leave the bottom row just short of the target color, and the endpoint colors would never quite appear.

I4. Every raster format is a header plus that same grid of bytes — PPM writes them raw, and PNG or JPEG mostly just *compress* the same numbers.