# Assignment 1: A Scene from Scratch

*Weightage:* ***3.6%*** (1/11 of the 40% assignment component)

**Deadline:** Friday $11^{th}$ September 2026, **11:59 pm**

## Introduction

Build a small 2D drawing toolkit — pixels, rectangles, gradients, alpha blending, Bresenham lines, circles, flood fill — in plain Python, then use it to render a scene of your own design.

### Objectives

- Treat an image as a 2D grid of RGB values, origin at the **top-left**
- Implement classic raster algorithms exactly to specification
- Composite with alpha: $C = \alpha F + (1 - \alpha)B$

### Collaboration and AI Policy

**All work is individual. Do not look at or copy another student's code, code structure, or scene. Both parties receive 0 in A1 and the case is reported.**

AI assistants may *explain* concepts and *debug code you wrote*; they may not generate solutions or scenes. Disclose any use in the header of `painter.py` and `scene.py`. You must be able to explain and modify any line you submit; spot checks happen.

## Getting Started

### A) File Setup

Download the starter folder and keep **all files in one directory**:

| | |
|---|---|
| `painter.py` | your toolkit — **all Part A work goes here** |
| `scene.py` | your scene — **all Part B work goes here** |
| `check.py` | the autograder; run it as often as you like |
| `expected_a1.py` | reference data used by `check.py` — do not edit |
| `smoke_test.py` | Part 0 environment check |

## B) Requirements

Python 3.9 or newer. A1 uses only the standard library — nothing to `pip install`.

**Viewing a `.ppm` file.** GIMP, IrfanView, macOS Preview, and the VS Code extension *PPM/PGM/PBM Viewer* open PPM directly. Otherwise convert:

```
ffmpeg -i scene.ppm scene.png
magick scene.ppm scene.png
```

## C) Part 0 — Smoke Test (0 pts, required)

From the starter folder, run:

```
python3 smoke_test.py
```

It checks the Python version, imports `painter.py`, and writes a $64 \times 64$ image `setup_ok.ppm`. It must print `ALL GOOD` and the image must match Figure 1. If not, ask on Discord or in office hours this week.

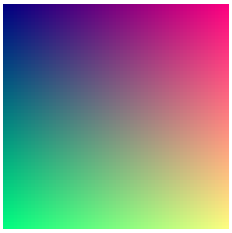

Figure 1: `setup_ok.ppm`, enlarged: $r = 4x$, $g = 4y$, $b = 128$. Red grows rightward ($x$), green grows *downward* ($y$).

## D) The Image Model

An image is exactly what it was in Lecture 1: a list of rows, each row a list of `(r, g, b)` tuples, written to disk as a PPM. The `painter` below is *your own* `painter.py`, the file you build in Part A, which `scene.py` imports. Its top carries four short GIVEN functions so that the checker, the smoke test and your scene all build images the same way:

- `save_ppm` is the function from Lecture 1, unchanged. It is the only code in A1 that touches a file.
- `new_image(w, h, color)` is the lecture's `[[color] * w for _ in range(h)]` — a fresh row each time, never `[[color] * w] * h`.
- `size_of(img)` is `(len(img[0]), len(img))`, i.e. `(width, height)`.
- `load_ppm` reads a PPM back; only the smoke test uses it.

So a scene file is the lecture's sketchbook with the helpers called by name:

```
import painter   # your file, same folder

img = painter.new_image(640, 420, (16, 18, 34))  # == [[(16,18,34)]*640 for _ in range(420)]
img[5][10] = (255, 0, 0)                # write:  row 5, column 10
print(img[5][10])                       # -> (255, 0, 0)
print(painter.size_of(img))             # -> (640, 420)
painter.save_ppm(img, "scene.ppm")      # Lecture 1's save_ppm
```

- $x$ grows rightward, $y$ grows downward; $(0, 0)$ is the top-left.
- A color is a tuple `(r, g, b)` of ints in 0..255.
- **Indexing is `img[y][x]`**: row first, then column.
- Python does not stop off-image writes: an index past the end raises `IndexError`, and a negative one silently wraps (`img[-1]` is the last row). Guarding both is T1.

**Rules.** `new_image`, `size_of`, `save_ppm`, `load_ppm` and `fill_triangle` are given; do not modify them. Do not change any function signature. Standard library only. For seeded randomness use `random.Random(7)`.

## Part A — The Toolkit (70 pts)

Implement the eight functions below in `painter.py`. Each docstring in the starter states the exact contract. Check your work anytime:

```
python3 check.py
```

Grading runs additional hidden cases. Hardcoding the checker's expected values earns 0 for that task.

### T1 · `set_pixel` · 6 pts

Write one pixel; **silently do nothing** if $(x, y)$ is off the image. All other functions draw through this one and inherit its clipping.

### T2 · `fill_rect` · 8 pts

Color a solid $w \times h$ block whose **top-left corner** is $(x, y)$ (Figure 2). Call `set_pixel` in a nested loop; clipping comes for free.

### T3 · `vertical_gradient` · 10 pts

Fade from one color at row $y_{top}$ to another at row $y_{bottom}$, **both inclusive**. Each row is one flat color across the full image width (use `size_of`). Lerp each channel per row, exactly as follows; the checker compares pixels exactly:

```
t = (y - y_top) / max(y_bottom - y_top, 1)
channel = int(top + t * (bottom - top))     # int() truncates; no +0.5 here
```

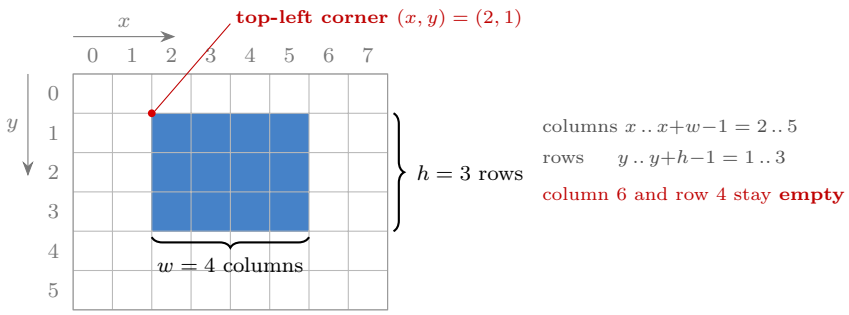


Figure 2: `fill_rect(img, 2, 1, 4, 3, blue)` on an $8 \times 6$ image. The last colored column is $x+w-1$, not $x+w$.

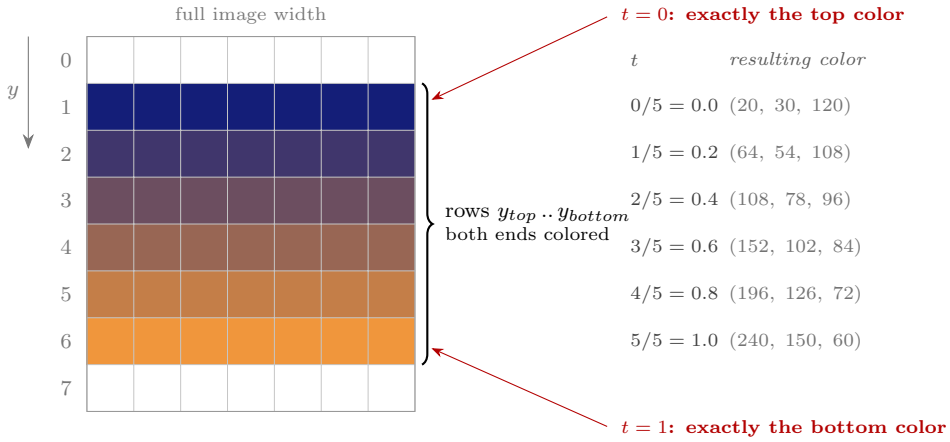


Figure 3: `vertical_gradient(img, 1, 6, (20,30,120), (240,150,60))`. Rows 0 and 7 are untouched.

## T4 · `blend_pixel` · 8 pts

Paint a translucent color: mix foreground $F$ (the color passed in) into background $B$ (the pixel already in the image) with $\alpha \in [0, 1]$, the fraction of $F$ kept (Figure 4).

$$C = \alpha F + (1 - \alpha)B$$

Compute each channel as

```
C = int(alpha * F + (1 - alpha) * B + 0.5)
```

The $+0.5$ rounds to nearest; plain `int()` truncates and drifts darker with every stacked layer (Lecture 3, *The Rounding Trap*).

## T5 · `bresenham_line` · 14 pts

Integer line from $(x_0, y_0)$ to $(x_1, y_1)$, **both endpoints included**, in every octant, including the single-point case. Draw through `set_pixel`. Implement **exactly** the Lecture 2 variant in Appendix A: other correct formulations differ by a pixel on some lines, and the checker expects this one.

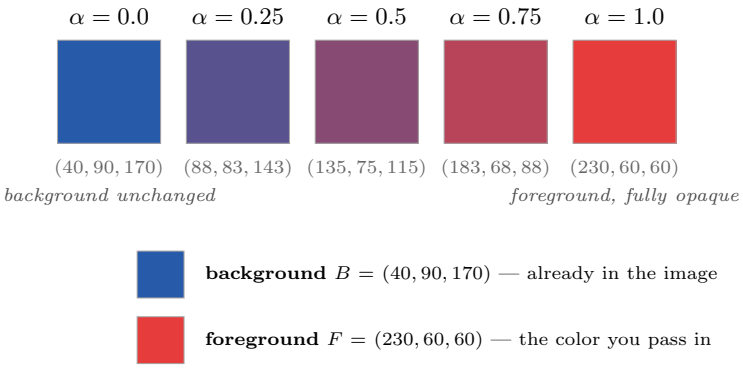


Figure 4: Red over blue at five alphas. At $\alpha = 0$ nothing changes; at $\alpha = 1$ the pixel is overwritten.

## T6 · `fill_circle` · 8 pts

A filled disc of radius $r$ centered at $(c_x, c_y)$. Loop over the bounding box $c_x{-}r \,..\, c_x{+}r$, $c_y{-}r \,..\, c_y{+}r$ and color each pixel that passes

$$(x - c_x)^2 + (y - c_y)^2 \;\le\; r^2$$

(Figure 5). Use $\le$, not $<$. Draw through `set_pixel`.

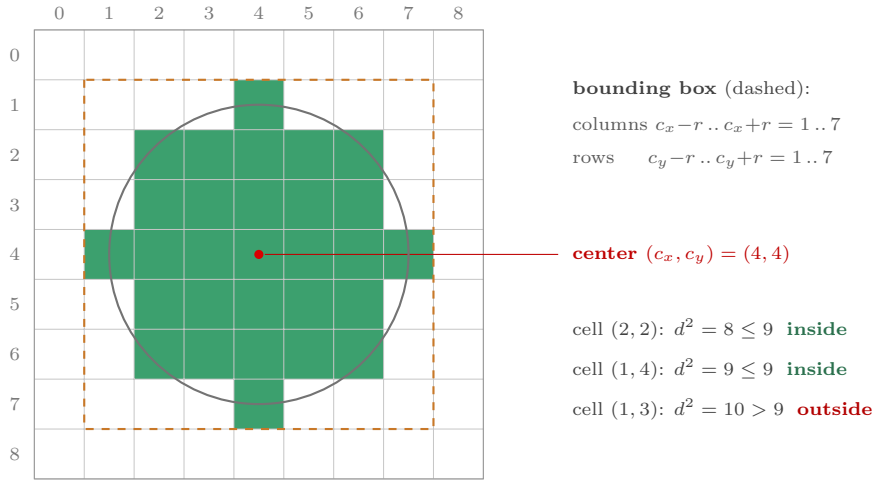


Figure 5: `fill_circle(img, 4, 4, 3, green)`. Every cell in the dashed bounding box is tested; 29 of 49 pass.

## T7 · `flood_fill` · 8 pts

4-connected flood fill from a seed: recolor the connected region whose color *exactly* matches the seed's. Two guards: an off-image seed is a no-op; a seed already of the new color is a no-op (otherwise the fill never terminates).

**Must be iterative** (Lecture 3): keep a stack or queue of pixels, pop one, skip it if off-image or no longer the old color, else paint it and push its four neighbours. Recursion overflows the call stack; the checker's $300 \times 300$ case catches it. Use 4 neighbours, not 8: an 8-connected fill leaks through the diagonal steps of a Bresenham outline.

### T8 · `aa_line` · 8 pts

Anti-aliased line using Xiaolin Wu's algorithm, as presented in the Wikipedia article *"Xiaolin Wu's line algorithm"*. This is not taught in class; reading and implementing it is part of the task. Plot a pixel with coverage $c$ by calling

```
blend_pixel(img, px, py, color, c)
```

The checker allows ±2 per channel: enough for float rounding, not for a different algorithm. Get T4 right first.

### Given: `fill_triangle` · 0 pts

A scanline triangle fill is provided (Appendix B). Use it freely in your scene. It draws through your `set_pixel`, so it works once T1 does.

## Part B — Your Scene (30 pts)

Using only your toolkit (plus `fill_triangle`), render an original scene in `scene.py`, saved as `scene.ppm`. Do not reproduce the exemplar below or the lecture sample scenes (space, Jupiter, mountains).

Paint back to front: background gradient, then translucent layers, then solid shapes, then outlines flood-filled from inside, then anti-aliased details on top. Later calls overwrite or blend with earlier ones.

Figure 6: Instructor exemplar: gradient sky, seeded stars, blended glow and mist, Bresenham road edges flood-filled inside, anti-aliased shooting star.

### B1 · Compliance Checklist · 18 pts

Binary items, graded exactly as written. The same checklist appears as comments in the starter `scene.py`.

| Item | Pts |
|---|---|
| `python3 scene.py` runs from a clean folder and writes `scene.ppm` | 2 |
| Image is at least $600 \times 400$ | 1 |
| Background uses `vertical_gradient` | 2 |
| At least 3 visible `bresenham_line` segments | 2 |
| At least 2 `fill_circle` uses | 2 |
| At least 1 `fill_rect` | 1 |
| At least 1 region colored via `flood_fill`, marked with a comment | 3 |
| At least 1 translucent element drawn via `blend_pixel` | 3 |
| Any randomness is seeded — the same image on every run | 1 |
| Header block (name, roll no, AI disclosure) is filled in | 1 |
| **Total** | **18** |

### B2 · Craft Rubric · 12 pts

Four criteria, each scored 0–3 against the anchors below.

| Criterion | 0 — absent | 1 — attempted | 2 — competent | 3 — excellent |
|---|---|---|---|---|
| **Composition & depth** | Shapes scattered with no relationship | Some grouping; no layering | Clear fore/mid/background; deliberate placement | The eye is led through the image; overlap and scale create convincing depth |
| **Color & light** | Arbitrary colors; no palette | Palette attempted but muddy or clashing | Cohesive palette; gradient and translucency support a mood | A deliberate lighting story (time of day, glow, atmosphere), cleanly executed |
| **Technical ambition** | Bare-minimum checklist calls | Primitives combined into at least one compound object | Several compound objects, or a custom helper built atop the toolkit | Beyond the spec: expressive AA lines, glow or mist effects, patterns, silhouettes |
| **Originality & polish** | Reproduces the exemplar or a lecture sample scene | Derivative with small changes | Clearly original; minor artifacts (gaps, stray pixels) | Original and artifact-free at 100% zoom |

**Calibration:** competent (2) on every axis earns 8/12. Reproducing the exemplar or a lecture sample scene caps *Originality* at 1.

## Submission

Submit one zip named `RollNo_A1.zip` (e.g. `231234567_A1.zip`) on the course website, containing exactly:

`painter.py` `scene.py` `scene.ppm`

Do not include `check.py`, `expected_a1.py`, or `__pycache__`. Both `.py` files must have the header block (name, roll number, AI disclosure) filled in.

**No late submissions. A missed deadline scores zero.** Documented medical or university grounds are the only exception.

## Appendix A: Bresenham's Line (as taught in Lecture 2)

Reproduced verbatim from Lecture 2. Note that a tie (`E == dx`) stays on the current row; the checker depends on it.

```
def bresenham_line(img, x0, y0, x1, y1, color):
    steep = abs(y1 - y0) > abs(x1 - x0)
    if steep:               # fold: swap x <-> y
        x0, y0, x1, y1 = y0, x0, y1, x1
    if x0 > x1:             # fold: left to right
        x0, y0, x1, y1 = x1, y1, x0, y0
    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):
        if steep:                    # unfold
            set_pixel(img, y, x, color)
        else:
            set_pixel(img, x, y, color)
        E += 2 * dy
        if E > dx:
            y += sy
            E -= 2 * dx
```

## Appendix B: The Given `fill_triangle`

Provided in `painter.py`; do not modify it.

```
def fill_triangle(img, x0, y0, x1, y1, x2, y2, color):
    verts = sorted([(x0, y0), (x1, y1), (x2, y2)], key=lambda v: v[1])
    (ax, ay), (bx, by), (cx, cy) = verts

    def lerp_x(y, pa, pb):
        if pb[1] == pa[1]:
            return pa[0]
        return pa[0] + (y - pa[1]) * (pb[0] - pa[0]) / (pb[1] - pa[1])

    for y in range(ay, cy + 1):
        xl = lerp_x(y, (ax, ay), (cx, cy))
        if y < by:
            xr = lerp_x(y, (ax, ay), (bx, by))
        else:
            xr = lerp_x(y, (bx, by), (cx, cy))
        for x in range(int(min(xl, xr)), int(max(xl, xr)) + 1):
            set_pixel(img, x, y, color)
```

Vertices are sorted by $y$ into top $a$, middle $b$, bottom $c$. For each scanline from $a_y$ to $c_y$, `lerp_x` finds where the left and right edges cross that row, and the span between them is filled. The long edge $a \to c$ is always one side; the other switches from $a \to b$ to $b \to c$ at the middle vertex. We derive this properly later in the course.

## Appendix C: The PPM Format

`save_ppm` is given; this is for reference. A binary PPM is a header followed by raw pixels, nothing else:

```
P6\n800 500\n255\n<raw bytes: r,g,b, r,g,b, ...>
```

- `P6`: magic number for binary portable pixmap.
- `800 500`: width, then height.
- `255`: maximum channel value, so each channel is one byte.
- Then $3 \times 800 \times 500$ bytes in row-major order: top row left to right, then the next row down. Pixel $(x, y)$ sits at byte offset $3\,(y \cdot \text{width} + x)$.