Lecture 3: Color, Alpha & Fills

Finishing the Toolkit

CS 453 — Computer Graphics

Fakhir Shaheen

Forman Christian University

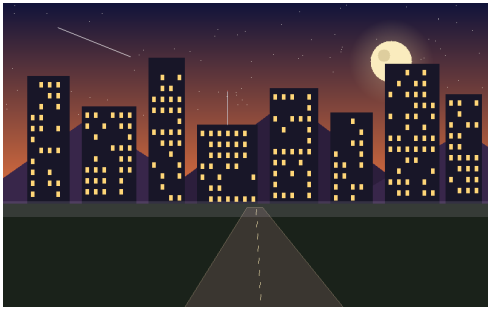
Where We Are

- **Monday**: pixels, coordinates, lerp, gradients
- **Wednesday**: Bresenham lines
- **Today**: the last three tools

(T8 is self-study by design. A1 is due **Friday 11 Sep, 11:59 pm**.)

Today's Three Tools

- **translucency**
  mixing colors — T4
- **discs**
  our first solid shape — T6
- **the paint bucket**
  filling regions — T7



the exemplar again — today we build its moon-glow, its discs, and its flood-filled road

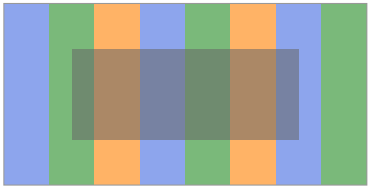
plus a short detour: circle **outlines** *(no task, but your scene may want them)*

Translucency

Everything We Draw Is Opaque

So far, writing a pixel **destroys** what was underneath.

`set_pixel`: detail **gone**



what we **want**: mist over the scene

Glass, mist, glow, smoke, shadow — all need pixels that **mix with** what is already there.

First Idea: Hardcode a "Shadow Gray"

Just pick a dark gray and paint it opaquely?

on grass: passable

on sand: **wrong**

A shadow is not gray.

It is a *darker version of whatever it falls on*.

What We Actually Want

The same "50% black" shadow, **mixed with** each background:

dark *green* — right

dark *orange* — right

A translucent color can't be precomputed —
the result must **depend on the background**.

Alpha Blending

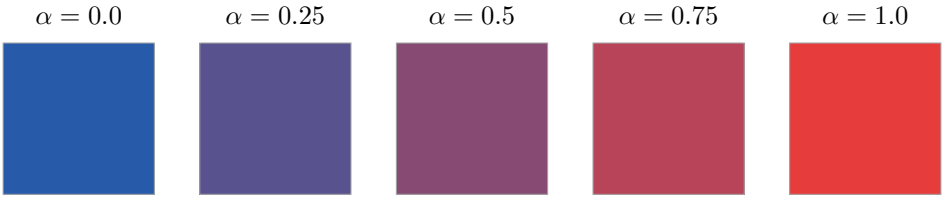
$$C = \alpha F + (1 - \alpha) B$$

F	foreground — the color you're painting
B	background — already in the image
α	how opaque, from 0 to 1

Take α parts of the new color, $1 - \alpha$ parts of the old. Apply **per channel**: once for r, once for g, once for b.

The Dial in Action

red $F = (230, 60, 60)$ painted over blue $B = (40, 90, 170)$:



$\alpha = 0$: background untouched $\alpha = 1$: exactly `set_pixel`

in between: a **mix**

You Already Know This Formula

Rearrange it:

$$\alpha F + (1 - \alpha)B \;=\; B + \alpha\,(F - B)$$

. . . that is **lerp**, with $t = \alpha$.

The Rounding Trap

Channels are **ints**. The blend produces **floats**.

```
# per channel:
C = int(alpha*F + (1-alpha)*B + 0.5)
```

- `int()` **truncates** toward zero:
  `int(127.9)` $\rightarrow$ `127`
- adding `0.5` first turns truncation into *round to nearest*:
  `int(127.5 + 0.5)` $\rightarrow$ `128`

Off by One. Who Cares?

Example. A 15% white disc over a black pixel:

$$0.15 \cdot 255 + 0.85 \cdot 0 = 38.25$$

A pixel can only store a whole number. So we store **38**.

Off by One. Who Cares?

Example. A 15% white disc over a black pixel:

$$0.15 \cdot 255 + 0.85 \cdot 0 = 38.25$$

A pixel can only store a whole number. So we store **38**.

Now a **second** disc on top. It blends over the 38 we stored:

$$0.15 \cdot 255 + 0.85 \cdot 38 = 70.55$$

Off by One. Who Cares?

Example. A 15% white disc over a black pixel:

$$0.15 \cdot 255 + 0.85 \cdot 0 = 38.25$$

A pixel can only store a whole number. So we store **38**.

Now a **second** disc on top. It blends over the 38 we stored:

$$0.15 \cdot 255 + 0.85 \cdot 38 = 70.55$$

Two choices:

chop the decimals:	`int(70.55)` $\rightarrow$ **70**
round to nearest:	`int(70.55 + 0.5)` $\rightarrow$ **71**

Off by One. Who Cares?

Example. A 15% white disc over a black pixel:

$$0.15 \cdot 255 + 0.85 \cdot 0 = 38.25$$

A pixel can only store a whole number. So we store **38**.

Now a **second** disc on top. It blends over the 38 we stored:

$$0.15 \cdot 255 + 0.85 \cdot 38 = 70.55$$

Two choices:

chop the decimals:	`int(70.55)` $\rightarrow$ **70**
round to nearest:	`int(70.55 + 0.5)` $\rightarrow$ **71**

Off by one. Who cares? ... *Nobody, until you stack more.*

Let's Give the Two Choices Names

Program A — *chops*

```
1 C = int(x)
```

after disc 2, the pixel holds **70**

Program B — *rounds*

```
1 C = int(x + 0.5)
```

after disc 2, the pixel holds **71**

Same disc. Same pixel. Different number stored.

Now we draw the **same four discs** with both programs and watch.

Why the Autograder Cares

The autograder compares your picture with mine, pixel by pixel.
Each color value may differ by **at most 2**.

- **Program A:** difference is 2 after three discs.
  One more slip and the test fails.
- **Program B:** difference is **0**, every time.
  My code rounds the same way.

Always use `int(x + 0.5)`

This was the A1 self check. T4 needs it, and T8 calls T4.

Stacking Faint Discs Makes a Glow

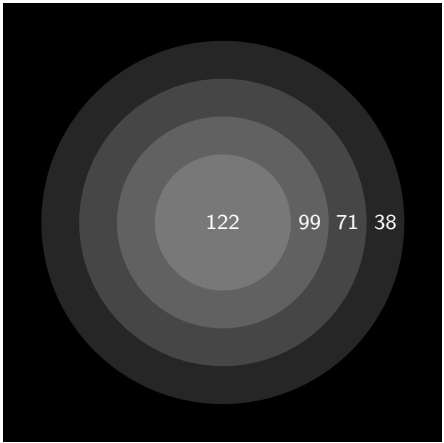
Draw a 15% white disc on black.
Then another on top. Then another.

discs	pixel value
1	38
2	71
3	99
4	122

Each disc adds 15% of the distance still left to white:

$71 + 0.15 \cdot (255 - 71) = 98.6 \to 99$

So it brightens fast at first, then slows down.
It can never pass 255.



pixel value where 1, 2, 3, 4 discs overlap

The exemplar's **moon glow** and **mist** are made exactly like this.

Circle Outlines

Attempt 1: Straight from Math Class, Again

Given: center (c_x, c_y) and radius r: Solve for y:

$$(x - c_x)^2 + (y - c_y)^2 = r^2 \quad \Longrightarrow \quad y \;=\; c_y \pm \sqrt{r^2 - (x - c_x)^2}$$

Walk x, compute *both* y's, round, plot:

Attempt 1: Straight from Math Class, Again

Given: center (c_x, c_y) and radius r: Solve for y:

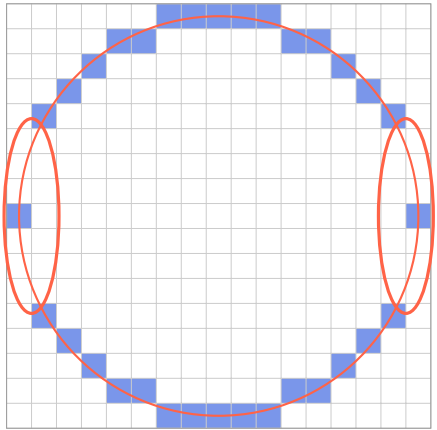
$$(x - c_x)^2 + (y - c_y)^2 = r^2 \quad \Longrightarrow \quad y \;=\; c_y \pm \sqrt{r^2 - (x - c_x)^2}$$

Walk x, compute *both* y's, round, plot:

```
for x in range(cx - r, cx + r + 1):
    h = math.sqrt(r*r - (x - cx)**2)    # rows from center to rim
    set_pixel(img, x, int(cy + h + 0.5), color)    # lower half
    set_pixel(img, x, int(cy - h + 0.5), color)    # upper half
```

h is the $\sqrt{\cdot}$ term: r at the top and bottom, 0 at the sides.

Attempt 1: Gaps at the Sides



$r = 8$: **32** pixels, two holes

Gaps!

At the top: one pixel per column — fine.

At the sides: one step in x, and y jumps **4 rows**.

Column $c_x - 8$ has $y = c_y$; column $c_x - 7$ has $y = c_y \pm 4$. Seven rows, nothing drawn.

The Same Disease as Wednesday's Naive Line

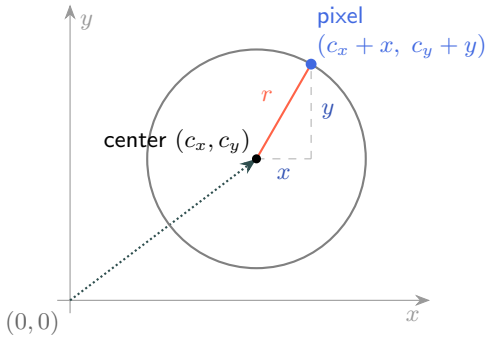
Wednesday: walking the wrong axis left gaps in steep lines.

The cure was: **walk the longer axis**.

A circle is shallow at the top, steep at the sides, and everything in between — **no single right axis**.

Unless. . . we only draw the part where there *is* one.

First, a Simpler Way to Name Points



Imagine the center is at $(0, 0)$.

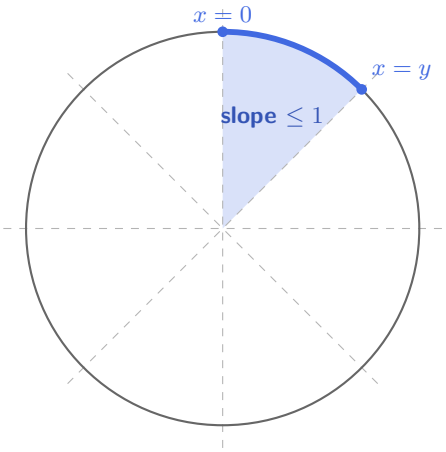
We first draw the circle as if its center is at the origin:

$$x^2 + y^2 = r^2$$

To plot it, add the center back:

pixel $= (c_x + x,\ c_y + y)$

Where Is the Circle Shallow?

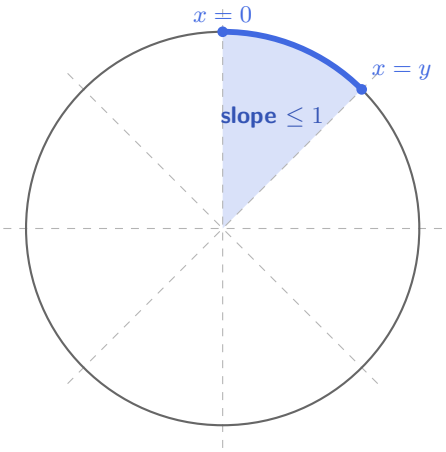


8 slices; only the shaded one is "one pixel per column" safe

In the shaded eighth:
at most one row per column.

Attempt 1 is **correct** here.

Where Is the Circle Shallow?



8 slices; only the shaded one is "one pixel per column" safe

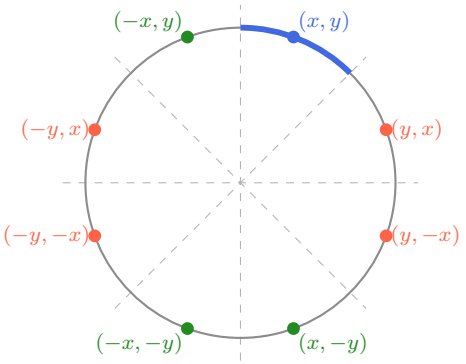
In the shaded eighth:
at most one row per column.

Attempt 1 is **correct** here.

Walk x from 0 until $x = y$:

$$x = \frac{r}{\sqrt{2}}$$

Draw One Eighth, Mirror It Seven Times



Wednesday's two mirrors, again:

- **flip signs**: $\pm x$, $\pm y$ $\times 4$
- **swap** $x \leftrightarrow y$ $\times 2$

One offset $\Rightarrow$ **8 pixels**
around (c_x, c_y)

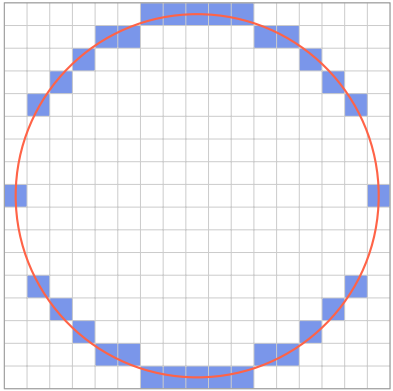
No fold, no unfold.
A circle is its own mirror image. Plot all eight at once.

Attempt 2: One Octant, Eight Pixels per Step

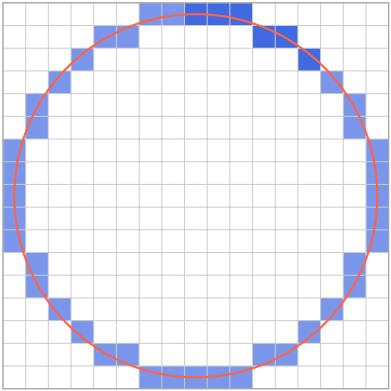
```
1  def draw_circle(img, cx, cy, r, color):
2      for x in range(int(r / math.sqrt(2)) + 1):  # up to x = y
3          y = math.sqrt(r*r - x*x)                 # a float. fine.
4          plot8(img, cx, cy, x, int(y + 0.5), color)
5
6  def plot8(img, cx, cy, x, y, color):
7      for dx, dy in ((x, y), (y, x)):          # mirror 2: swap
8          for sx in (+1, -1):                  # mirror 1: flip signs
9              for sy in (+1, -1):
10                 set_pixel(img, cx + sx*dx, cy + sy*dy, color)
```

x, y are offsets; `plot8` adds the center back.
Off-image pixels? `set_pixel` drops them. Clipping is still free.

Attempt 1 vs. Attempt 2



walk x all the way: **32** pixels

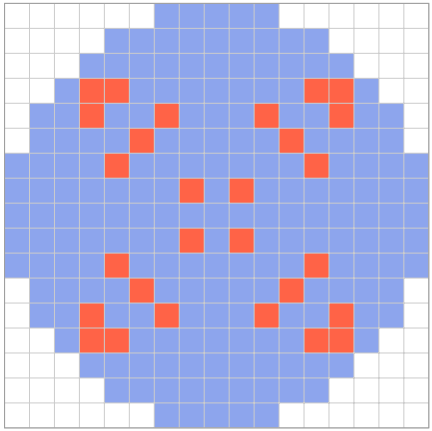


6 computed, **44** pixels, closed

Same formula, same rounding — **6 square roots** instead of 17,
and no gaps: the dark cells are the only ones we computed.

Discs

Now Fill It: Outlines All the Way Down?



$r = 8, 7, \ldots, 0$: **28** holes

Drawing concentric circles results in gaps.

Try again.

Flip the Question: Ask Every Pixel

Don't compute *where the circle goes*.

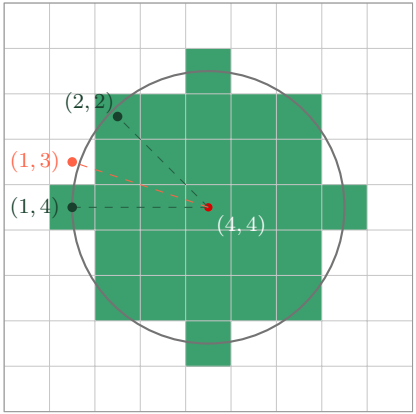
Ask each pixel:

"am I inside?"

One test, straight from the definition of a circle:

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

The Inside Test, Pixel by Pixel



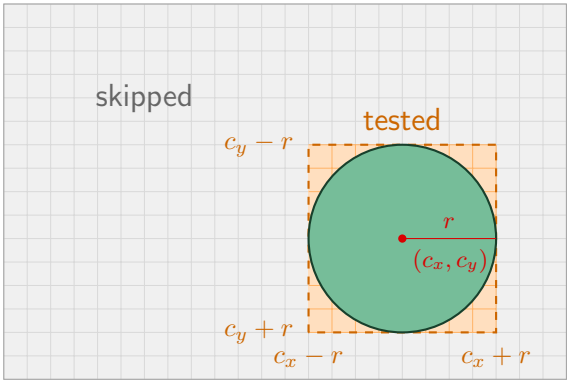
$(2,2)$: $d^2 = 8 \leq 9$ **in**

$(1,4)$: $d^2 = 9 \leq 9$ **in**

$(1,3)$: $d^2 = 10 > 9$ **out**

```
fill_circle(img, 4, 4, 3, green)
```

Only Test the Pixels Near the Circle



Don't test every pixel.

Test only the **box** around the circle.

441 tests, not **240,000**.
(radius 10, 600×400 image)

<=, not <: rim pixels are inside.

The Code: Filling a Disc

```
1 def fill_circle(img, cx, cy, r, color):
2     for y in range(cy - r, cy + r + 1):
3         for x in range(cx - r, cx + r + 1):
4             if (x - cx)**2 + (y - cy)**2 <= r*r:
5                 set_pixel(img, x, y, color)
```

lines 2–3: walk the **box** around the circle

line 4: the inside test — squared, with <=

line 5: `set_pixel` ignores anything off the image

This Pattern Has a Future

"For each pixel, evaluate a formula; act on the answer"

today — Python, T6

```python
if (x-cx)**2 + (y-cy)**2 <= r*r:
    set_pixel(img, x, y, col)
```

Week 12 — GLSL, on the GPU

```
if (length(p - c) <= r)
    fragColor = col;
```

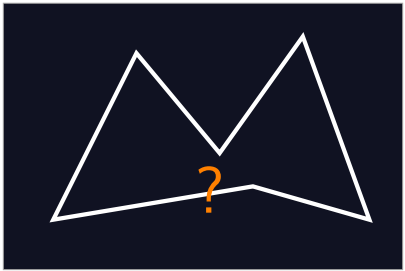
A GPU **fragment shader** runs exactly this —
one tiny program per pixel, millions in parallel.

You are writing your first shader today; it just runs on one core.

The Paint Bucket

Regions With No Formula

A disc had an inequality. But this?

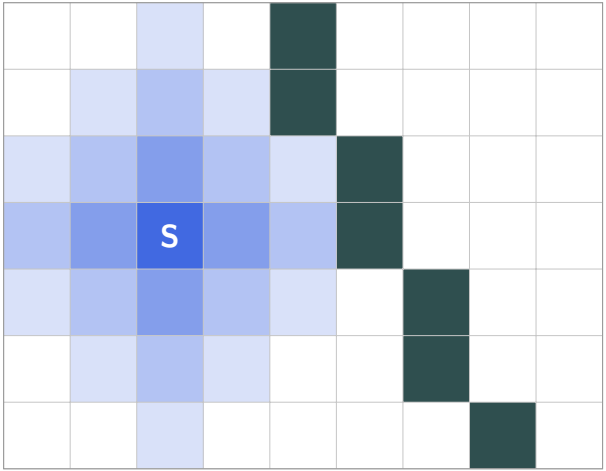


Six `bresenham_line` calls —

no inequality describes the inside.

Every paint program solves this with the **bucket tool**. Today: what actually happens when you click it.

The Idea: Paint That Spreads



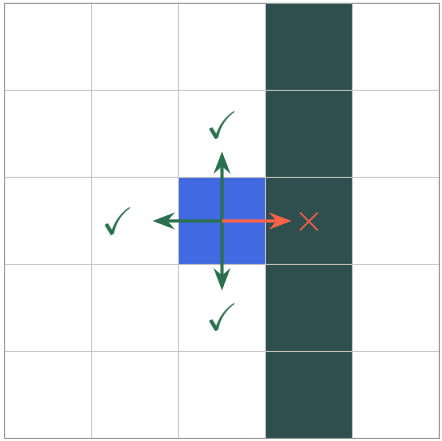
Pick one pixel.
Call it the **seed**, **S**.

Give it a new colour.
That is the **paint**.

The paint spreads to the pixels next to it, wave by wave.

The dark outline **stops** it.

The Spreading Rule



same colour → paint outline → stop

From a painted pixel, look at its **4 neighbours**:
up, down, left, right.

Same colour the seed started with?
Paint it. It spreads next.

Any other colour?
Stop.

First Attempt: Recursion — Beautiful...

```
1  def flood_fill(img, x, y, new):
2      w, h = size_of(img)          # (guards omitted)
3      old = img[y][x]              # remember the seed's colour
4      img[y][x] = new              # paint this pixel
5      for nx, ny in ((x+1,y), (x-1,y), (x,y+1), (x,y-1)):
6          if 0 <= nx < w and 0 <= ny < h:    # on the image?
7              if img[ny][nx] == old:         # still old colour?
8                  flood_fill(img, nx, ny, new)
```

. . . and Doomed

```
RecursionError:
maximum recursion depth
exceeded
```

- each unfinished pixel = one **stack frame**
- Python's stack limit: $\sim$1000 frames
- a 300×300 region = **90,000** pixels

. . . and Doomed

```
RecursionError:
maximum recursion depth
exceeded
```

- each unfinished pixel = one **stack frame**
- Python's stack limit: $\sim$1000 frames
- a 300×300 region = **90,000** pixels

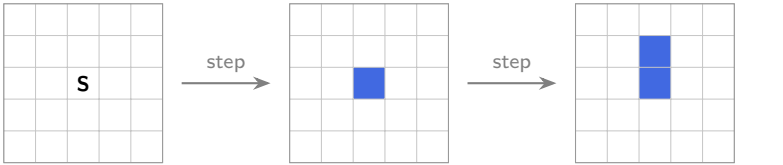
Problem: the call stack is too small.

Fix: use a Python list instead. No recursion.

The autograder tests a very large region.

The Fix: Keep a To-Do List

1. Start: the list holds only the seed.
2. Take one pixel off the list.
3. Off the image, or not the old colour? **Skip it.**
4. Otherwise: **paint it**. Put its 4 neighbours on the list.
5. Repeat until the list is empty.



start
list: (2,2)

took (2,2): painted it
list: (3,2) (1,2) (2,3) (2,1)

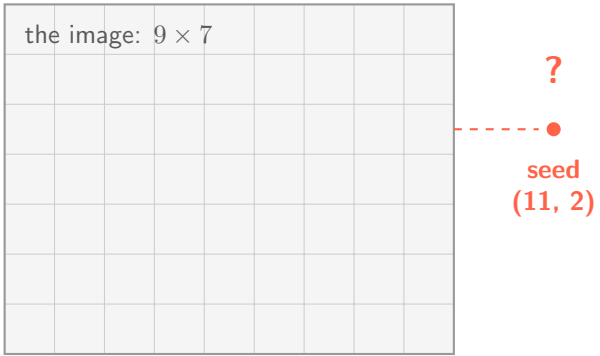
took (2,1): painted it
list: (3,2) (1,2) (2,3)
(3,1) (1,1) (2,2) (2,0)

(2,2) is on the list again. Already painted, so it will be skipped. Duplicates are fine.

The Fix: The Code

```
1  def flood_fill(img, x, y, new):
2      w, h = size_of(img)           # (guards on the next slide)
3      old = img[y][x]               # the colour we spread over
4      todo = [(x, y)]               # the list: only the seed
5      while todo:                   # until the list is empty
6          cx, cy = todo.pop()       # take one pixel
7          if not (0 <= cx < w and 0 <= cy < h):
8              continue              # off the image: skip
9          if img[cy][cx] != old:
10             continue              # not the old colour: skip
11         img[cy][cx] = new         # paint it
12         todo.append((cx+1, cy))   # its 4 neighbours go on the list
13         todo.append((cx-1, cy))
14         todo.append((cx, cy+1))
15         todo.append((cx, cy-1))
```

Guard 1: Seed Off the Image



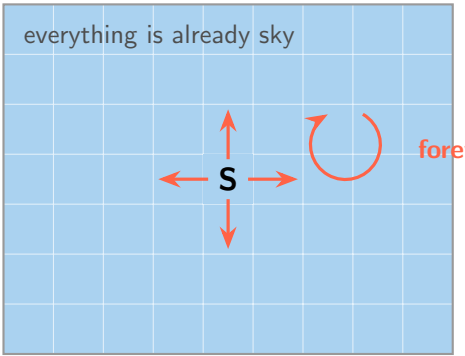
Seed outside the image?

Do nothing.

Same rule as `set_pixel`:
never crash, just ignore it.

```
if not on image:  return
```

Guard 2: New Colour = Seed's Colour



sky on sky: nothing changes

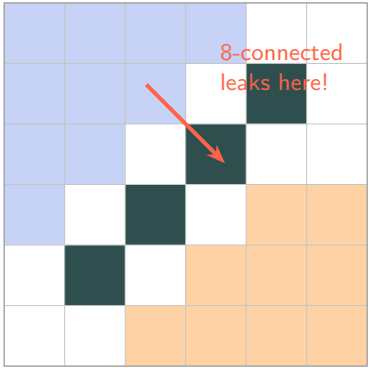
`flood_fill(img, 4, 3, sky)`
... but $(4, 3)$ is already sky.

Painting changes nothing.
Every neighbour still matches `old`.
The list **never empties**.

Infinite loop

```
if new == old:  return
```

Why 4 Neighbors, Not 8?

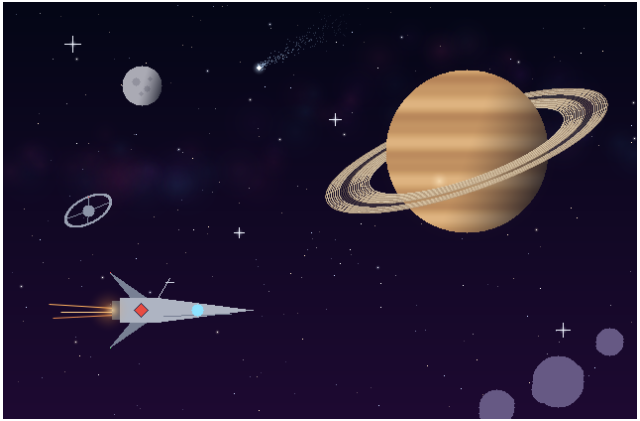


- Bresenham lines take **diagonal steps** — their pixels touch corner-to-corner
- an **8-connected** fill moves corner-to-corner too ⇒ it slips *through* and floods your sky

A **4-connected** fill cannot cross the outline.

Fill and outline are a matched pair: diagonal-stepping lines need a non-diagonal fill.

Assignment 1



a scene built with only these tools

A1: Painter

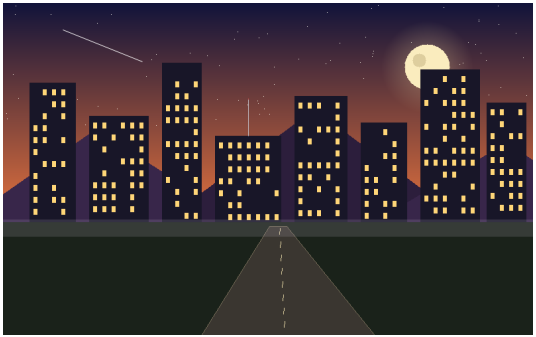
Part A — the toolkit
T1–T8, eight functions — 70 pts

Part B — your own scene
anything you like — 30 pts

Released: **today**

Due: **Fri 11 Sep, 11:59 pm**

Part B: Where Craft Points Live



every element traces back to a toolkit call

Paint like a stage set — back to front:

1. `vertical_gradient` — the sky
2. seeded stars — `set_pixel` + `random.Random(7)`
3. glow & mist — stacked `blend_pixel` discs
4. solid shapes — `fill_rect`, `fill_circle`, triangles
5. outlines — `bresenham_line`, then `flood_fill` inside
6. one `aa_line` flourish on top

Later layers overwrite (or blend with!) earlier ones — **order is composition**.

Craft rubric: 12 pts — palette, depth, ambition, polish.

Exit Ticket

Today's questions *(2 minutes)*

(a) `blend_pixel` with $\alpha = 0.25$:
foreground red channel 200,
background red channel 40.
Resulting red channel?

(b) `flood_fill` where the new color equals the seed's color — without the guard, what happens, and why?

CS 453 — Exit Ticket #3

Name: _____

Roll no: _____

(a) _____

(b) _____

hand in on your way out

Monday: vectors & the dot product — the one piece of math under everything from Week 2 on.