Assignment 5: Into the Third Dimension

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

Deadline: Sunday 11th October 2026, **11:59 pm**

Introduction

For four weeks you have been putting things on the screen. This week you stop. You will describe a **place** — where things are, how big they are, what color — and then ask, one pixel at a time, "what would I see from here?"

Nothing in your scene file will position a circle. Circles will appear because **spheres are there**. That is the whole difference, and everything else in this course follows from it.

Objectives

- Extending your vector library to 3D — and noticing how little changes
- The **cross product**: the one operation with no 2D equivalent, and what it is for
- Solving ray–sphere and ray–plane intersection exactly
- Building a **camera** from a position, a target and a field of view
- Firing one ray per pixel and finding the nearest hit — your first ray tracer
- Softening jagged edges by averaging samples inside a pixel

Collaboration and AI Policy

All work is individual. Students are NOT permitted to look at or copy each other's code, code structure, or scenes. If caught, both parties receive a 0 in A5 and the case is reported.

Per the course policy: AI assistants may be used to *explain* concepts and to *debug code you have already written*, but not to generate solutions or scenes. Whatever you use, disclose it in the header of `vec3.py`, `tracer.py` and `scene.py`. You must be able to explain, and modify on request, any line you submit.

Getting Started

A) File Setup

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

`vec3.py`	(edit this) your 3D vector library — **T1–T4 go here**
`tracer.py`	(edit this) your ray tracer — **T5–T11 go here**
`scene.py`	(edit this) your scene — **all Part B work goes here**
`vec2.py`	*do not edit* — from A2, for reference
`canvas.py`	*do not edit* — the `Canvas` class
`painter.py`	*do not edit* — from A1
`check.py`	*do not edit* — the autograder; run it as often as you like
`expected_a5.py`	*do not edit* — reference data used by `check.py`

As in A1–A4, there is **nothing to install**: A5 is standard-library Python throughout — no `pip`, no virtual environment, no third-party packages. **And this week there is no `ffmpeg` either**, because A5 produces a single image rather than a video.

What A5 *does* need is patience, or PyPy. This is the first program you have written where the computer has to think: every pixel fires rays, and every ray is tested against every object. A 480×320 render at 2×2 samples is over a million rays.

```
python3 scene.py        # works, but slower
pypy3 scene.py          # the same picture, many times faster
```

B) What Actually Changes in 3D

Less than you expect. Almost every formula you already know gains one term and keeps its meaning:

Idea	**A2, in 2D**	**A5, in 3D**
dot product	$a_x b_x + a_y b_y$	$a_x b_x + a_y b_y + a_z b_z$ *(one more term)*
length	$\sqrt{v \cdot v}$	$\sqrt{v \cdot v}$ *(identical)*
normalize	$v/\lvert v\rvert$	$v/\lvert v\rvert$ *(identical)*
reflect	$d - 2(d \cdot n)\, n$	$d - 2(d \cdot n)\, n$ *(identical)*
cross product	a **number** (signed area)	a **vector**, perpendicular to both

Only the last row is genuinely new, and it is new for an interesting reason. In 2D there was only one direction perpendicular to a plane, and it pointed out of the page — nowhere the maths could reach — so the cross product collapsed to the single number you used for side tests. In 3D that direction is a real direction, so the answer is a real vector.

That vector is what makes a camera possible, and it is the only tool that can give you one.

The A5 API at a glance. The left column is **given** and already works; the right column is **yours**.

Given (plumbing)	Yours (T1–T11)
`Vec3(x, y, z)`, `.x .y .z`	`+ - * / -` and `v * v` (colors)
`Ray3(origin, direction)`	`dot length normalized`
`Sphere(center, radius, color)`	`cross reflect lerp to_color`
`Plane(point, normal, color)`	`ray.at(t) ray_toward(o, target)`
`Hit3(t, point, normal, obj)`	`ray_sphere(ray, sphere)`
`Camera(pos, target, fov, aspect)`	`ray_plane(ray, plane)`
`T_MIN EPSILON`	`nearest_hit(ray, objects)`
`FOG_DISTANCE`	`Camera._build_basis()`
	`Camera.ray_through(s, t)`
	`sample_offsets(n)`
	`sky_color shade trace render`

C) The Shape of a Ray Tracer

The entire program, before you write any of it:

```
for every pixel:
    build the ray through that pixel      # T9, the camera
    find the nearest thing it hits        # T6, T7, T8
    decide what color that is             # T11
```

Three steps. A5 does the first two *properly* and deliberately fakes the third, because deciding colors properly means **lighting** — and lighting is next week.

Part A — 3D Vectors and the Tracer (70 pts)

Implement T1–T4 in `vec3.py` and T5–T11 in `tracer.py`. Each function's docstring in the starter repeats its exact contract. Check yourself anytime:

```
python3 check.py
```

The checker runs **value cases** (comparing what your functions return, within 10^{-6}) and **pixel cases** (comparing whole tiny renders, exactly), as in A2–A4. **Grading runs additional hidden cases**: passing the checker is necessary, not sufficient — hardcoding its expected values earns a 0 for the task.

> **T1–T4 below are written in `vec3.py`.** Open that file now. Most of it is A2's `Vec2` with one more component, so if you find yourself thinking "this is the same as last time" — yes. That is the lesson. Only **T3** is genuinely new.

T1 · Vec3 arithmetic · 4 pts

Add, subtract, negate, scale and divide, all returning **new** vectors — `Vec3` is immutable, exactly as `Vec2` was.

One addition worth noticing: `__mul__` must accept **either a number or another** `Vec3`. Multiplying two vectors componentwise is what lets this one class carry **colors** as well as positions:

`Vec3(0.9, 0.2, 0.2) * Vec3(0.5, 0.5, 1.0)` — a red surface under a blue tint

A color really is three numbers that add and scale like a vector, so giving it its own class would be duplicated code. Division by (near) zero returns the zero vector rather than raising — the same guard style as A2.

T2 · dot, length, normalize · 5 pts

The same formulas as A2, with one more term:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \qquad |\mathbf{v}| = \sqrt{\mathbf{v} \cdot \mathbf{v}}$$

Write `length_squared()` as well, and **use it**. It skips the square root, and whenever you only need to *compare* two lengths — or test one against a radius — comparing the squares gives the same answer, because taking a square root never reorders positive numbers. In a program that runs a million times, a skipped `sqrt` is not nothing.

Guard the zero vector in `normalized()`, as in A2: it has no direction, and it turns up more often than you would think, because `normalized()` is usually called on the difference of two points.

T3 · the cross product · 6 pts

The one genuinely new operation in this assignment.

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y,\ \ a_z b_x - a_x b_z,\ \ a_x b_y - a_y b_x)$$

Each component is built from the *other* two coordinates, and the middle one runs "backwards" — that middle component is the one people get wrong, so check it: it is $a_z b_x - a_x b_z$.

Three properties, and you should know all three by heart:

- **It is perpendicular to both inputs.** $(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{a} = 0$ and $(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{b} = 0$, always. Hand it two directions and it hands back the third axis. *This is the property T9 is built on.*
- **Order matters:** $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$. Swap the operands and the result flips, which is why the order of your camera's cross products decides whether your image comes out mirrored.
- **Parallel inputs give zero.** Since $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta$, two vectors pointing the same way cross to the zero vector. **This is not a rare edge case** — it is exactly what happens when a camera looks straight down, and it is why T9 needs a guard.

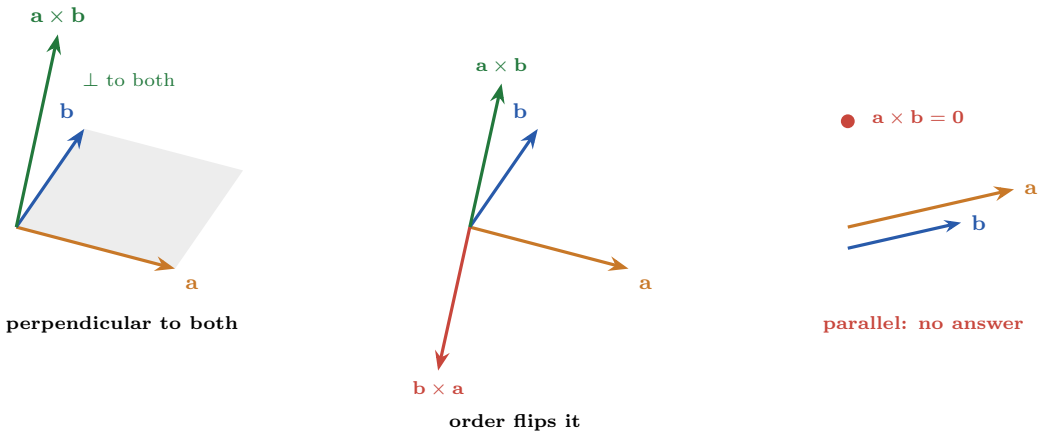

Figure 1: The cross product hands you the axis your two inputs do not span — unless they do not span a plane at all, in which case there is no such axis and you get zero.

T4 · reflect, lerp, to_color · 5 pts

Three short ones.

1. `reflect(normal)` — letter for letter A2's formula, $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$: subtract twice the part pointing into the surface, and what is left points back out. You will use it for mirrors in Lecture 17.

2. `lerp(other, t)` — blend from self at $t = 0$ to `other` at $t = 1$: $\mathbf{a} + t(\mathbf{b} - \mathbf{a})$. It is how A5 paints its sky and how it fades distant things into haze.

3. `to_color()` — read this vector as a color in $[0, 1]$, scale by 255, and **clamp** into $[0, 255]$ before truncating. The clamp is the part that matters: a channel that computes to 1.3 must become 255, not wrap around to 76 and put a bright hole in the middle of your highlight.

> `vec3.py` **is done — T5–T11 below are written in** `tracer.py`**.** That file is the ray tracer proper: what a ray hits, where the rays come from, and what color each pixel ends up.

T5 · the ray equation in 3D · 3 pts

Unchanged from A4:

$$p(t) \;=\; \mathbf{o} \;+\; t\,\mathbf{d}, \qquad t \geq 0$$

Write `Ray3.at(t)` and `ray_toward(origin, target)`. The constructor normalizes the direction for you, and for the same reason it did last week: with $|\mathbf{d}| = 1$, t **is a distance**, so "nearest hit" and "smallest t" are the same sentence.

T6 · ray_sphere · 9 pts

This is A4's `ray_circle`**, word for word**, with three-component dot products instead of two. A point is on the sphere when $|\mathbf{p} - \mathbf{c}|^2 = r^2$; substituting $p(t)$ and writing $\mathbf{oc} = \mathbf{o} - \mathbf{c}$ gives a quadratic in t:

$$a = \mathbf{d} \cdot \mathbf{d}\ (= 1) \qquad b = 2\,(\mathbf{d} \cdot \mathbf{oc}) \qquad c = \mathbf{oc} \cdot \mathbf{oc} - r^2$$

Required behaviour:

- Discriminant $b^2 - 4ac < 0$: the ray misses. Return `None`.
- Otherwise take the **nearest root inside the window** [`t_min`, `t_max`]: try the near root $(-b - \sqrt{\cdot})/2a$ first, then fall back to the far one. The fallback is what handles a ray starting *inside* the sphere, and a `t_min` that lands past the near face.
- The **normal** is $(\mathbf{p} - \mathbf{c})/r$ — a unit vector straight out from the centre. Dividing by the radius is the cheap way to say `normalized()`.

Why the window exists. Without `t_min`, a ray leaving a surface finds that same surface again at $t \approx 0.0000001$ and everything goes black. That is A4's shadow acne, in 3D, and `T_MIN` is A4's `SHADOW_BIAS` under a new name.

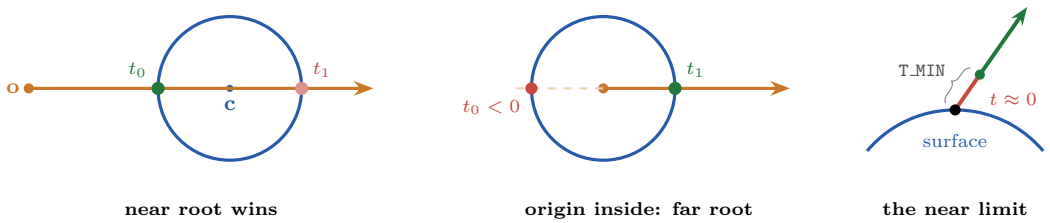

Figure 2: The t window does two jobs: it picks the surface you can actually see, and it stops a ray from re-hitting the surface it just left.

Notes (not graded). You solved this exact quadratic in A4 for circles. That is not a coincidence and it is not padding — it is the point. A sphere is a circle that stopped caring how many dimensions it was in, and everything you debugged last week you do not have to debug again.

T7 · `ray_plane` · 5 pts

A point is on the plane when the vector to it is perpendicular to the normal: $(\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0$. Substituting $p(t)$ and solving:

$$t = \frac{(\mathbf{p}_0 - \mathbf{o}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

The denominator is the whole story. When $\mathbf{d} \cdot \mathbf{n}$ is near zero the ray runs **parallel** to the plane: it either never meets it or lies entirely within it, and in neither case is there a single crossing to report. Return `None`; **do not divide**. Reject any t outside $[$`t_min`, `t_max`$]$, exactly as in T6.

An infinite plane is a bargain: one division, and it gives your spheres something to *sit on*. That is most of what makes a render read as a place rather than as objects floating in a void.

T8 · `nearest_hit` · 6 pts

`objects` is a **mixed** list of `Spheres` and `Planes`. Send each to the right intersector and keep the hit with the **smallest** t. **Reordering the list must not change the answer.**

A tidy trick: shrink `t_max` to the best t found so far, so every later object is asked "can you beat this?" rather than "what do you hit?". Both versions are correct; the shrinking one does less work.

This is A4's `nearest_hit` again, and the warning stands. It costs $O(n)$ per ray, and at 480×320 with 2×2 samples you are running it over a million times. **Lecture 20's bounding volume hierarchy replaces exactly this loop**, and by then you will want it.

T9 · the camera · 12 pts

The one hard idea in A5. You are given a position and a target, and you must produce a ray through any pixel. Three steps.

Step 1 — build the camera's own three axes. You know where it looks. You do *not* know which way is "right" or "up" for the camera, and this is what the cross product is for:

```
forward = (target - position).normalized()
right   = forward.cross(world_up).normalized()
up      = right.cross(forward).normalized()
```

The second cross is not redundant. The world's "up" is only a *hint*, and it is usually *not* perpendicular to forward — any tilted camera makes it so. Crossing twice squeezes out the part that was not perpendicular and leaves three axes at exact right angles. Three mutually perpendicular unit vectors are called an **orthonormal basis**, and building one is the standard move whenever a thing needs its own coordinate system. You will do it again for every object in Lecture 34.

Guard the degenerate case. If forward is parallel to `world_up` — you are looking *straight down* — the first cross gives the zero vector and everything after it is garbage. Fall back to a different hint. This is T3's third property, arriving exactly where it was promised.

Step 2 — size the image plane. Put a virtual rectangle one unit in front of the camera. The field of view is the vertical angle it can see, so:

$$\text{half_height} = \tan\left(\frac{\text{fov}}{2}\right) \qquad \text{half_width} = \text{half_height} \times \text{aspect}$$

That is trigonometry and nothing more: the rectangle is one unit away, so the opposite side of the angle is the tangent. A wider fov makes a bigger rectangle at the same distance — which is precisely what a wide-angle lens does.

Step 3 — aim a ray at a point on that rectangle. `ray_through(s, t)` takes fractions: $s = 0$ is the **left** edge, $s = 1$ the **right**; $t = 0$ is the **top**, $t = 1$ the **bottom**.

$$u = (2s - 1) \cdot \text{half_width} \qquad v = (1 - 2t) \cdot \text{half_height}$$

```
direction = forward + right * u + up * v
```

The two minus signs do different jobs. $2s - 1$ recentres the fraction onto $[-1, 1]$. $1 - 2t$ recentres *and flips*, because the image plane's up axis and the screen's y axis run opposite ways — your canvas has had y growing downward since A1. Get that flip wrong and your render comes out **upside down**, which is a bug you can spot instantly and fix in one character.

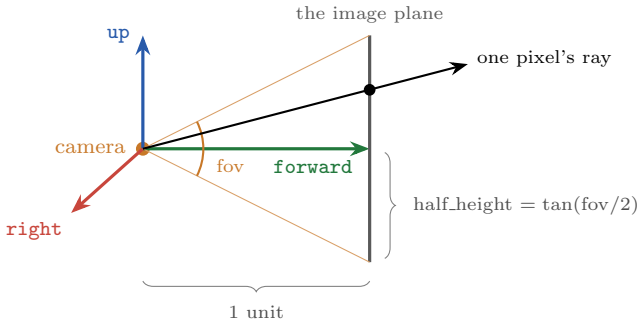

Figure 3: The camera builds its own three axes, hangs a rectangle one unit in front sized by the field of view, and fires a ray at a point on it. Every pixel is one such point.

T10 · sample offsets · 6 pts

A pixel is a little square, not a point. Fire one ray through its centre and every edge in your image becomes a staircase, because each pixel is forced to answer a yes/no question — sphere or sky? — about an area that is genuinely part of both. That staircase is **aliasing**.

The fix is A4's soft-shadow trick, reused exactly: **replace one binary answer with the average of many**. Fire several rays through different points inside the pixel and average their colors, and a pixel that is 30% sphere comes out 30% sphere-colored.

Lay the samples on an $n \times n$ grid, each at the **centre** of its own cell — the same midpoint rule as A4's `area_light_samples`, and for the same reason:

$$\text{offset}(i, j) = \left(\frac{i + 0.5}{n}, \frac{j + 0.5}{n} \right)$$

Required: return them in row-major order (j outermost, i innermost), and $n \leq 1$ returns the single centre sample `[(0.5, 0.5)]` — which is what "antialiasing off" means.

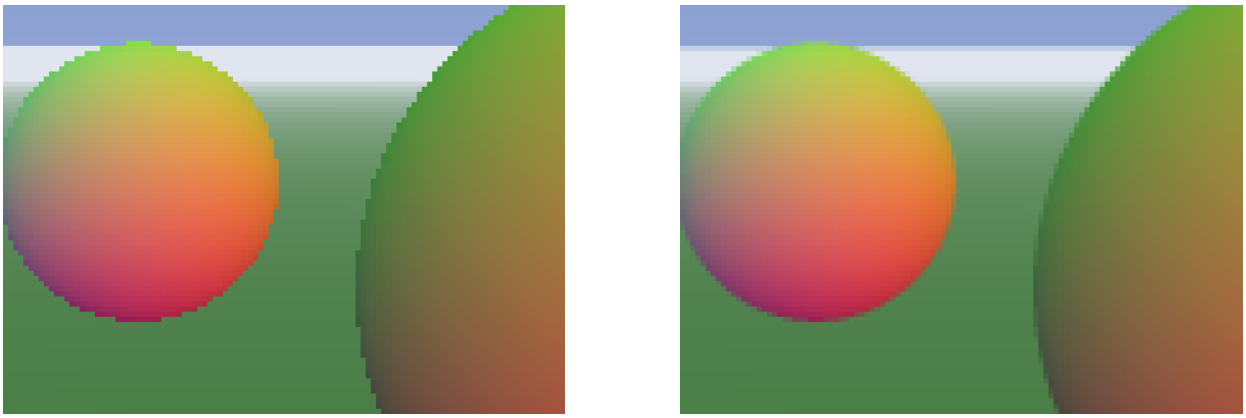

Figure 4: The same crop at 1 sample per pixel (left) and 3×3 (right), both magnified $4\times$ so the pixels are visible. Nothing about the geometry changed — only how many questions each pixel asked.

The cost, stated plainly: $n \times n$ rays per pixel. $n = 2$ is four times the render time for a visibly cleaner edge; $n = 4$ is sixteen times for a mildly cleaner one. **Build at 1, ship at 2 or 3.**

T11 · the render loop · 9 pts

Four functions, and together they are the ray tracer.

1. `sky_color(ray)` — what a ray sees when it hits nothing. A flat background is a missed opportunity; a gradient costs one `lerp`:

```
h = 0.5 * (direction.y + 1.0)     bottom.lerp(top, h)
```

Because the direction is normalized, `y` is already in $[-1, 1]$, so that one line is the entire remap.

The default colors are fixed, and `check.py` compares against them exactly. They are constants at the top of `tracer.py`; use them whenever the matching argument is `None`:

Constant	Value	Used when
`DEFAULT_SKY_TOP`	`Vec3(0.35, 0.55, 0.95)`	`sky_color`'s `top` is `None`
`DEFAULT_SKY_BOTTOM`	`Vec3(0.92, 0.90, 0.85)`	`sky_color`'s `bottom` is `None`
`DEFAULT_FOG_COLOR`	`Vec3(0.86, 0.89, 0.94)`	`shade`'s `fog_color` is `None`

In `trace`, the fog color handed to `shade` is the sky's bottom color, so when `sky_bottom` is `None` it is `DEFAULT_SKY_BOTTOM`, not `None`.

2. `shade(hit, ray)` — **A5's deliberate placeholder for lighting.** There are no lights this week, so a surface is colored by **which way it faces**:

```
n_color = normal * 0.5 + Vec3(0.5, 0.5, 0.5)     base * n_color
```

A surface facing $+x$ goes red, $+y$ green, $+z$ blue, so a sphere comes out with a smooth rainbow shell — and **that smoothness is the proof your normals are right**. Flat facets or harsh seams mean the bug is in your normal, not your shading. This is the most useful debugging view in ray tracing, which is why it is worth having before you have lights.

Then **fade toward the haze with distance**:

```
f = min(hit.t / fog_distance, 1.0)     color.lerp(fog_color, f)
```

This one is not decoration, and it is worth knowing *why* it is here. Coloring purely by normal means a **flat ground plane, whose normal is identical everywhere, renders as one uniform slab** with no depth information in it at all — your geometry is correct and your picture still looks flat, which is a confusing place to be. Fading with distance puts the depth back. It is also a real technique with a real name, **aerial perspective**, and painters were using it centuries before anyone rendered a pixel.

3. `trace(ray, objects)` — nearest hit, or the sky. Two lines, and they are the two this whole assignment builds toward. In Lecture 17 this function grows a recursive call for mirrors; in Lecture 27 it becomes a path tracer. **The shape never changes.**

4. `render(img, camera, objects, samples)` — the loop. For each pixel, for each offset:

```
s = (x + dx) / width     t = (y + dy) / height
```

accumulate `trace(...)`, then **average by how many samples you actually took** — `len(offsets)`, not what you hoped for.

Divide by `width`, not `width - 1`. With `width`, the offset 0.5 lands exactly on the centre of pixel x, which is what you want to sample. With `width - 1` the image stretches by half a pixel and the corner samples sit on the rim.

Part B — Your 3D Scene (30 pts)

Using your tracer, design and render **one still image** of a 3D scene of your own, in `scene.py`, saved as `scene.ppm`.

The rule that defines this assignment: you are not drawing this picture. You are describing a place and photographing it. Nothing in `scene.py` should position anything *on the screen* — you place objects in the *world*, put a camera somewhere, and the picture is what the camera reports.

The test of whether you got it right is one line long. **Move only the camera, and the whole picture must recompose itself correctly** — perspective, occlusion, and every silhouette. B1 requires exactly that, and the grader will try it.

Workflow — follow this order

1. Keep the canvas **small** (240×150) and `SAMPLES = 1` while you iterate. You will be re-rendering a lot.
2. Get the **layout and camera** right first. A bad camera cannot be rescued by more pixels, and this is where nearly all of B2's marks live.
3. *Then* raise the resolution and set `SAMPLES = 2` or `3` for the final render — that last step multiplies your wait by 4 or 9.

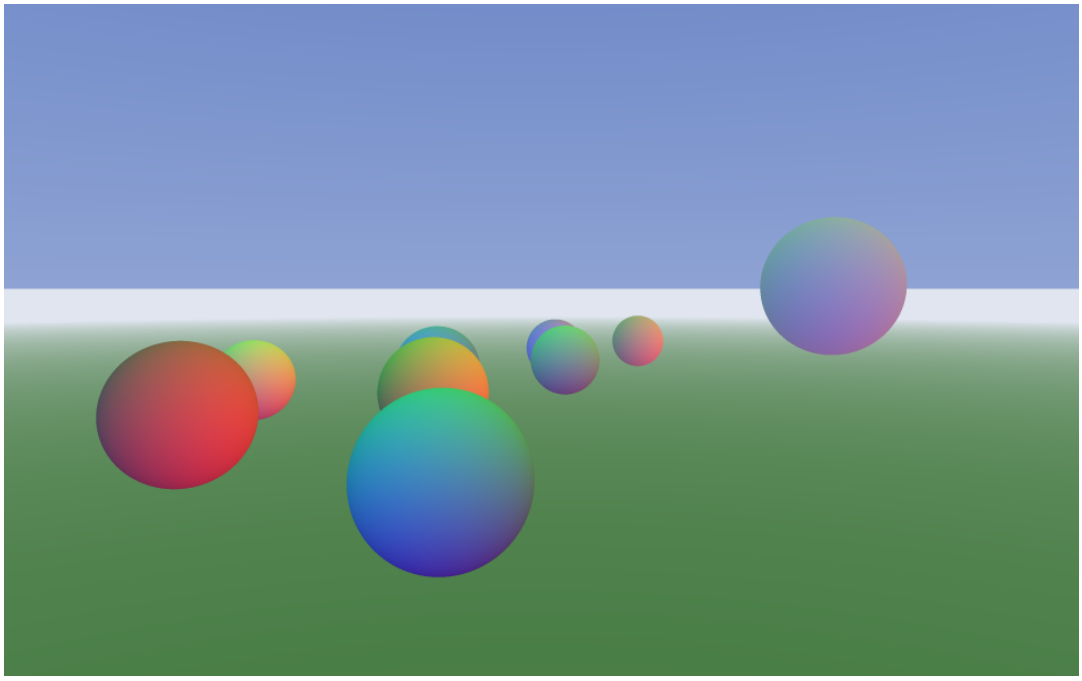

Figure 5: The instructor exemplar, "Marbles on a Long Table" (960×600, 3×3 samples). Every marble in the receding row is *almost the same size* — the shrinking is perspective doing the work, not a size gradient typed into the scene. The camera sits low and off to one side so the row runs diagonally across the frame; a line seen end-on carries no depth information at all. Your scene should be *yours*, not this one.

```
python3 scene.py       # writes scene.ppm
ffmpeg -i scene.ppm scene.png     # if you want a PNG to look at
```

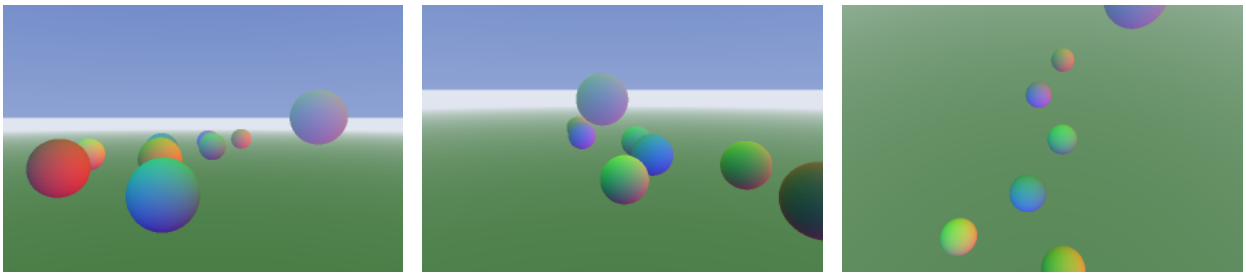

Figure 6: The **same scene file**, three camera positions, nothing else changed — including the last one looking straight down, which is the degenerate case T9 guards against. This is what "you described a place" buys you, and it is what the grader will check.

Four traps worth naming

All four cost people an evening every year.

1. Spheres that sink or float. A sphere of radius r rests on a ground plane at height y_0 when its centre is at $y = y_0 + r$ — **not** at y_0. This is the single commonest Part B bug, and it is far easier to see in the render than to spot in your numbers.

2. A camera inside an object. If your camera sits inside a sphere, every ray hits that sphere's far wall and your image is one flat color. This is A4's "light inside its own geometry", wearing a hat.

3. An upside-down render. The v flip in T9. One character.

4. Nothing overlaps. Objects arranged so that none hides any other read as flat no matter how correct the maths, because **overlap is what tells the eye "in front of"**. Move the camera until things cut across each other.

If your render is slow

It will be. Three fixes, in order of how much they buy you:

Use PyPy. A5 is arithmetic in a loop, which is exactly what a JIT is good at — the same code, one word of the command changed.

Turn `SAMPLES` down while iterating. It multiplies your ray count by n^2 directly. Build at 1, ship at 2 or 3.

Render small while composing. 240×150 is a quarter of the pixels of 480×320 and tells you everything you need to know about whether the camera is in the right place.

B1 · Compliance Checklist · 16 pts

Binary items, checked by the grader exactly as written. The checklist also 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 480×320	1
At least **6 spheres**, of at least **3 different sizes**	2
A ground plane, with every sphere **sitting on it** (none floating or sunk)	2
At least one sphere partly **behind** another, from your camera	3
Final render uses `SAMPLES` ≥ 2 (antialiasing on)	2
Your **own** sky colors, not the two the starter ships with	2
Moving **only** the camera recomposes the scene correctly	1
Any randomness is seeded — the same image on every run	1
Total	**16**

Two of these are faked every year, so to be explicit. The **overlap** must be genuine occlusion — one sphere in front of another from the camera's point of view, not two spheres that merely look close together. And **sitting on the ground** means $y = y_0 + r$ for every sphere, not for most of them.

B2 · Craft Rubric · 14 pts

Criterion	0 — absent	1 — attempted	2 — competent	3 — excellent
Composition & camera (0–4, doubled at 2)	Camera arbitrary; subject clipped or tiny	Everything is in frame, but the viewpoint says nothing	Camera placed deliberately; the frame is balanced and the subject reads	The viewpoint IS the idea — angle, height and fov chosen together
Depth & overlap	Flat; nothing occludes anything	Some overlap, but it reads as clutter	Clear near/mid/far; overlap used to establish order	Depth is unmistakable — perspective, overlap and fade all agree
Scene design	Objects scattered at random	Some grouping; sizes and colors arbitrary	Deliberate arrangement; sizes and palette chosen	The arrangement has an idea behind it, and you can state it
Originality & polish	Reproduces the starter's row of six, or the handout's marbles	Derivative with small changes	Clearly original; minor artifacts (jaggies, sunk spheres)	Original and artifact-free; ships at a resolution that shows it off

Composition & camera is scored 0–4 (it is what this assignment is about); the other three are 0–3. Total 14.

Calibration: *competent* on every axis earns 9/14 — a solid outcome. Top marks are reserved for work that would headline the class gallery. Submitting the starter's six spheres, lightly recolored, caps *Originality* at 1.

Two cheap upgrades, both nearly free. First, **lower your camera**. Most submissions are shot from head height looking down, which turns a scene into a diagram; drop the camera to object height and the same objects become a place. Second, **put something big far away** — one large sphere near the horizon terminates the recession and gives the eye somewhere to land, and the distance fade does the rest.

Submission

Submit **one zip file** named `RollNo_A5.zip` (e.g. `231234567_A5.zip`) on the course website, containing exactly:

`vec3.py` `tracer.py` `scene.py` `scene.ppm`

If `scene.ppm` exceeds the upload limit, convert it to PNG (`ffmpeg -i scene.ppm scene.png`) and submit that instead — but **do not** reduce your resolution below 480×320 to make it fit.

Do **not** include any of the following: `vec2.py`, `canvas.py`, `painter.py`, `check.py`, `expected_a5.py`, `__pycache__`, or any intermediate renders. We run your submission against our own copies of the given files. All three `.py` files must have the header block (name, roll number, AI disclosure) filled in.

The deadline is hard: no late submissions, per the course outline. A missed deadline scores zero. Documented medical or university grounds are the only exception.

Appendix A: The 3D Cheat Sheet

Every question this assignment asks, and the one line that answers it.

You want	Call	The idea
the third axis	`a.cross(b)`	perpendicular to both; zero if they are parallel
a point along a ray	`ray.at(t)`	$\mathbf{o} + t\,\mathbf{d}$; with $\lvert\mathbf{d}\rvert = 1$, t is a distance
does this ray hit that sphere	`ray_sphere(ray, s)`	a quadratic; nearest root inside $[t_{\min}, t_{\max}]$
does it hit the ground	`ray_plane(ray, p)`	one division; guard $\mathbf{d} \cdot \mathbf{n} \approx 0$
what does it hit first	`nearest_hit(ray, objs)`	smallest t wins, whatever the list order
the ray for a pixel	`cam.ray_through(s, t)`	two crosses for the axes, tan(fov/2) for the size
smoother edges	`sample_offsets(n)`	n^2 rays per pixel, averaged
the color of a ray	`trace(ray, objs)`	nearest hit, or the sky

The single rule to carry away: you did not draw the picture. You described a place, and asked what it looks like from here.

Appendix B: Debugging a Ray Tracer

Rendering bugs are quiet: you get an image, it is just the wrong one. Almost all of them are on this list.

Symptom	Almost always
The image is upside down	the v flip: use $(1 - 2t)$, not $(2t - 1)$ (T9)
The image is mirrored left-to-right	the cross products are in the wrong order (T9)
Everything is one flat color	the camera is inside an object, or `forward` is parallel to `world_up`
Spheres look like ellipses	`aspect` not passed, or passed as h/w
Spheres sink into or float above the ground	centre at y_0 instead of $y_0 + r$
Black speckles all over a surface	`t_min` missing: the ray re-hits its own surface (T6)
A far object draws in front of a near one	keeping the first hit instead of the smallest t (T8)
The whole image is sky	every ray misses — objects are behind the camera (check the sign of z)
Edges are a hard staircase	`SAMPLES = 1`; that is aliasing, not a bug (T10)
The image is stretched by half a pixel	dividing by `width - 1` instead of `width` (T11)
Colors wrap: bright spots go dark	`to_color()` is not clamping (T4)
Render takes minutes	`SAMPLES` too high, or resolution too high, or not using PyPy

The best single debugging tool you have is `shade`'s normal coloring. A correct sphere is a smooth rainbow shell. If yours has facets, seams, or a hard edge running through it, stop looking at your camera — the bug is in the normal.

Appendix C: A 3D Scene in Twenty Lines

The skeleton every Part B scene shares. Everything interesting is in `build_scene` and the camera; the loop never changes.

```
GROUND_Y = 0.0

def build_scene():
    objects = [Plane(Vec3(0, GROUND_Y, 0), Vec3(0, 1, 0),
                     Vec3(0.5, 0.5, 0.55))]

    # (x, z, radius, color) - note the centre is at GROUND_Y + radius,
    # which is what makes a sphere REST on the plane instead of sinking
    for (x, z, r, color) in MY_LAYOUT:
        objects.append(Sphere(Vec3(x, GROUND_Y + r, z), r, color))

    return objects

def render_scene():
    img = Canvas(WIDTH, HEIGHT, (0, 0, 0))
    objects = build_scene()
    camera = Camera(CAMERA_POSITION, CAMERA_TARGET, CAMERA_FOV,
                    WIDTH / HEIGHT)      # move THIS and everything follows

    offsets = tracer.sample_offsets(SAMPLES)
    inv = 1.0 / len(offsets)

    for y in range(HEIGHT):
        for x in range(WIDTH):
            total = Vec3(0, 0, 0)
            for (dx, dy) in offsets:
                ray = camera.ray_through((x + dx) / WIDTH,
                                         (y + dy) / HEIGHT)
                total = total + tracer.trace(ray, objects,
                                             SKY_TOP, SKY_BOTTOM)
            painter.set_pixel(img, x, y, (total * inv).to_color())
    return img
```

Note what is *not* here: nothing decides where a circle goes on the screen. The camera decides, and it decides correctly for any camera you give it — which is the property worth protecting. The moment you find yourself nudging an object to make it look right *from this one angle*, you have started drawing again.