Assignment 6: Light, Shadow, Metal and Glass

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

Deadline: Friday 16th October 2026, **11:59 pm**

Introduction

Last week's picture was a diagram: flat colours and no shadows. This week you switch the lights on.

You will add lamps, shadows, a mirror and a glass ball to your tracer, one task at a time, and after almost every task the picture changes in a way you can see. Everything you need is on the slides of lectures 16 to 18, and the code is written the same way here as it is there.

Then Part B is yours: build a scene, light it, and render a picture you would be happy to show someone.

Objectives

- Point lights: the **cosine law**, and why light falls off with the **square** of the distance
- What a material does with the light: Lambertian R/π, and the Blinn–Phong highlight
- **Shadows as a visibility question**, answered by one more ray
- Mirrors: bouncing the ray, and multiplying what comes back by the albedo
- Glass: **Snell's law**, total internal reflection, and Schlick's approximation
- **Recursion in a renderer**, and why it needs a depth limit

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 A6 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 `material.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**:

`material.py`	(edit this) the materials — **T1–T2 go here**
`tracer.py`	(edit this) lights, `trace` and glass — **T3–T10 go here**
`scene.py`	(edit this) your scene — **all Part B work goes here**
`vec3.py`	*do not edit* — **A5's solution, given to you working**
`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_a6.py`	*do not edit* — reference data used by `check.py`

Note what is given. `vec3.py` is the *reference solution* from A5, so nobody is blocked this week by a cross product that was slightly wrong last week. The top half of `tracer.py` is A5's tracer, given and working too: `Ray`, `HitRecord`, `Sphere`, `Plane`, `Group`, `Camera`. Everything below the line marked `A6 STARTS HERE` is yours, except the pixel loop `render` at the very end.

Standard-library Python throughout: no `pip`, no packages. But **A6 is slower than A5**, and you should expect that: every pixel now sends a shadow ray *to every lamp*, and every mirror and glass surface sends the ray onward for up to eight more steps.

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

B) The Three New Ideas

Everything in A6 is one of these three, and they arrive in this order.

Idea	In one line	What it buys you
Light	$k \max(0,\ \mathbf{n} \cdot \mathbf{l})\, I/r^2$	Surfaces are bright where they face the lamp. Shape becomes visible.
Shadow	send a ray toward the lamp; is anything in the way?	Objects sit *on* the ground and relate to each other.
The ray continues	`trace` calls `trace`	Mirrors, glass, and reflections of reflections.

The A6 API at a glance. The left column is **given** and already works; the right column is **yours**.

Given	Yours (T1–T10)
`Vec3, dot, cross, lerp, to_color`	`Lambertian(albedo).evaluate(l, v, n)`
`Ray(eye, direction), .e, .d, .evaluate(t)`	`BlinnPhong(albedo, specular_exponent).evaluate`
`HitRecord: .surface .t .normal .front_face`	`Reflective(albedo)  Glass(refractive_index)`
`Sphere, Plane,` each with `.hit(ray, t0, t1)`	`AmbientLight(intensity).illuminate(ray, rec)`
`Group: .add(surface), .hit(ray)`	`PointLight(position, intensity).illuminate(ray, rec)`
`Camera, render`	`trace(ray, depth)`
`scene, lights, sky_bottom, sky_top`	`refract(d, n, ni, nt)  schlick(cos_theta, ni, nt)`
`random_unit_vector` (bonus)	`shade_glass(ray, rec, depth)`

C) The Names Are the Lectures' Names

From this week the code is written as on the slides of lectures 15 to 18, so that a slide and your file say the same thing. A5 used other names for the same objects. The given code has been rewritten for you; this table is for reading it.

In A5	From A6 on
`Ray3(origin, direction)`, `.origin`, `.direction`, `.at(t)`	`Ray(eye, direction)`, `.e`, `.d`, `.evaluate(t)`
`Hit3`, or `None` for a miss	`HitRecord`; a miss has `t == math.inf`
`hit.obj`, `hit.point`	`rec.surface`, `ray.evaluate(rec.t)`
`ray_sphere(ray, sphere)`	`sphere.hit(ray, t0, t1)`
`nearest_hit(ray, objects)`	`scene.hit(ray)`, where `scene` is a `Group`
`Sphere(center, radius, color)`	`Sphere(center, radius, material)`

Two names live at the top of `tracer.py`, as in the lectures: `scene`, the `Group` that holds every surface, and `lights`, the list of lights. A lamp reads `scene` for its shadow ray, and `trace` reads both. Your `scene.py` gives them their values before it renders (`tracer.scene = ...`), and the checker does the same before each of its cases. Inside `tracer.py` you only *read* them, by their plain names.

One field of the record matters more this week: `front_face`. In A5 every ray came from the camera and met the *outside* of things. In A6 a ray travels **through** a glass ball and meets the *inside* of that same ball on its way out. The given `hit` methods turn the stored normal to face the ray in both cases, and record which case it was. So **your code can always trust `rec.normal` to face the ray**, and T10 reads `rec.front_face` to ask the one question that matters: is this ray going *into* the glass, or coming *out*?

Part A — Lights, Shadows and Materials (70 pts)

Implement T1–T2 in `material.py` and T3–T10 in `tracer.py`. Each 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; each channel may differ by 1), as in A2–A5. **Grading runs additional hidden cases**: passing the checker is necessary, not sufficient — hardcoding its expected values earns a 0 for the task.

T1–T2 below are written in `material.py`. Four small classes. They are where the rest of the assignment finds the answer to "what is this surface made of?"

T1 · Lambertian, Reflective, Glass · 5 pts

In A5 a surface had a colour and nothing else. This week it has a **material**, and there are four kinds:

Class	Is like	What it stores
`Lambertian(albedo)`	chalk, paper	its reflectance R, a colour
`BlinnPhong(albedo, specular_exponent)`	plastic, polished wood	R, and the exponent p of its highlight
`Reflective(albedo)`	a mirror	the colour that its reflections are multiplied by
`Glass(refractive_index)`	a window, a marble	one number: 1.5 for glass, 1.33 for water

A material is one kind. A surface is matte, or it is a mirror, or it is glass; `trace` asks which, and does a different job for each (T6, T7, T10).

For T1, write the three constructors, which only store their arguments under the names in the table, and `Lambertian.evaluate`:

$$k \;=\; \frac{R}{\pi}$$

k is the material's factor: a lamp works out E, the light that *arrives* at a point, and $k\,E$ is the light that *leaves* toward the eye. A matte surface sends the same amount in every direction, so its k uses none of the three arrows that `evaluate(l, v, n)` is given. They are in the argument list because every material's `evaluate` has the same one: a lamp can then call it without asking what kind of material it has in hand. The π is lecture 16's: a surface that reflects all of its light, spread over the half of the directions above it, sends $1/\pi$ of it along each one.

T2 · BlinnPhong · 7 pts

A matte surface with a **highlight**: the bright spot on a billiard ball. The highlight is brightest where the surface is turned like a small mirror that sends the lamp's light straight to the eye. The test for that uses the **half vector h**, the direction halfway between the arrow to the lamp and the arrow to the eye:

$$\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{\|\mathbf{l}+\mathbf{v}\|} \qquad s = \max(0,\ \mathbf{n}\cdot\mathbf{h})^{p} \qquad k \;=\; \frac{R}{\pi} + R\,s$$

Where the normal points along $\mathbf{h}$, $s = 1$. A little away from there, $\mathbf{n}\cdot\mathbf{h}$ is a little under 1, and the power p decides how quickly the spot fades: $p = 10$ gives a wide soft spot, $p = 1000$ a small sharp one.

Two things to notice. The highlight is multiplied by R too, so the spot has the surface's own colour. And k can be far above 1 inside the spot: the pixel there is brighter than the screen can show, and `to_color` clamps it. That is what a highlight is.

> `material.py` **is done — T3–T10 below are written in** `tracer.py`, everything below the line marked `A6 STARTS HERE`.

T3 · AmbientLight · 3 pts

Every light has one method, `illuminate(ray, rec)`, which returns that light's **share** of the colour at the hit `rec`. The simplest light is the ambient light, which arrives from every side, the same amount I_a at every point:

$$\text{share} \;=\; R\,I_a$$

R is the albedo of the material that was hit. There is no arrow to the light in this formula, no normal, no distance and no shadow test: ambient light has no position.

Ambient light is a stand-in, and you should know it as one. Real shadows are not black: they are lit by light that bounced off the walls and the floor first. Following that light is Lecture 27 and A9. Until then a small constant stands in for all of it. **Keep it small**, 0.05 to 0.1.

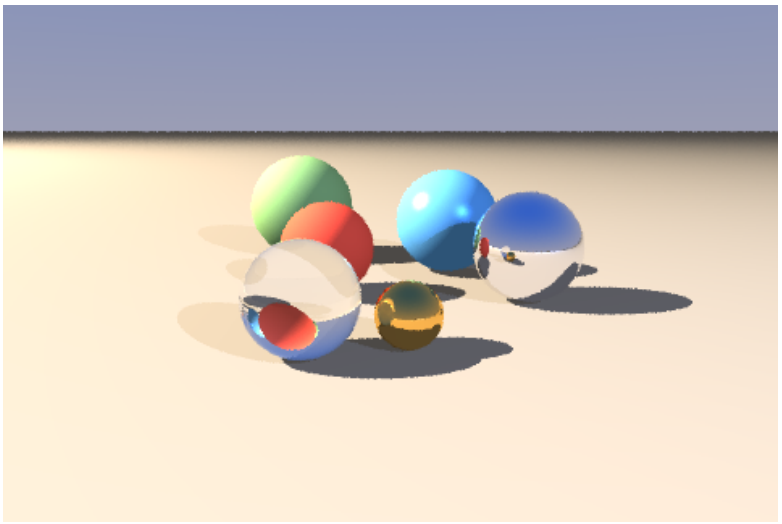

Figure 1: The exemplar scene with the ambient light's intensity raised to 0.6. Everything is still geometrically correct and the render is ruined: the contrast is gone and the shadows have faded. This is the commonest way a Part B submission loses craft marks.

T4 · PointLight: the light that arrives · 10 pts

A lamp is a position and an intensity I, a `Vec3`: its colour and its strength in one. Its `illuminate` works out everything from the ray and the hit record:

- $\mathbf{x}$ the hit point, `ray.evaluate(rec.t)`
- r the distance from $\mathbf{x}$ to the lamp
- $\mathbf{l}$ the unit arrow from $\mathbf{x}$ *toward* the lamp
- $\mathbf{n}$ the normal, `rec.normal`
- $\mathbf{v}$ the unit arrow from $\mathbf{x}$ toward the eye: the ray's direction, turned round

$$E \;=\; \max(0,\ \mathbf{n}\cdot\mathbf{l})\,\frac{I}{r^2} \qquad\qquad \text{share} \;=\; k\,E$$

E is the light that arrives, and k is the material's answer: `rec.surface.material.evaluate(l, v, n)`. The lamp does not ask what kind of material it is.

Why a dot product? A surface that faces the lamp catches a whole beam's worth of light. Tilt it, and that same beam is spread over a wider patch of surface, so each bit of surface gets less. How much wider is exactly $\cos\theta$, and $\cos\theta$ is $\mathbf{n}\cdot\mathbf{l}$ for unit vectors. This is **Lambert's cosine law**.

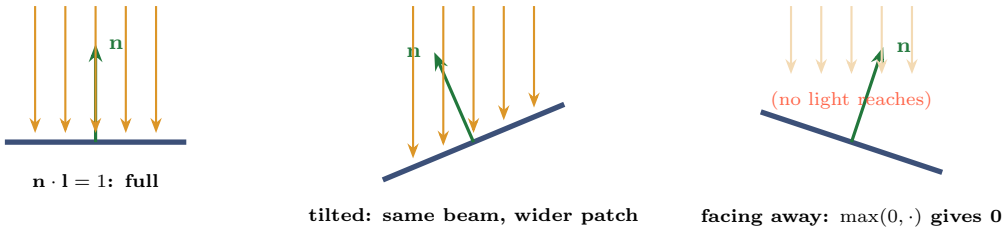

Figure 2: The same beam, three orientations. Tilting the surface spreads the light over more of it, and the spreading factor is the cosine — which is the dot product.

The `max` is not decoration. A surface that faces away from the lamp receives *no* light, not negative light. Leave it out and the negative amount is subtracted from your other lights.

Why r^2? The lamp's light spreads over a sphere, and a sphere's area grows with the square of its radius. Twice as far away, four times the area, a quarter of the light.

Mind the direction of l. It goes from the surface *to* the lamp. Get it backwards and every lit face goes dark while every dark face lights up.

A number to check by hand. A lamp of intensity $(30, 30, 30)$ hangs 3 above a matte floor of albedo $(0.8, 0.6, 0.4)$. Straight under it, $\mathbf{n} \cdot \mathbf{l} = 1$ and $r^2 = 9$, so $E = 30/9 = 3.33$ in each channel. The floor's factor is $k = R/\pi = (0.255,\ 0.191,\ 0.127)$, and the share is $k\,E = (0.849,\ 0.637,\ 0.424)$. This is the checker's first number for T4.

A practical consequence. The light is divided by r^2 and then by π, so a lamp 10 away with intensity 1 gives nothing you can see. **A lamp's intensity must grow with the square of its distance**: about $3r^2$ is a good first guess, which is 300 for a lamp 10 away and about 11 000 for one 60 away. The numbers are large, and that is correct.

T5 · PointLight: the shadow ray · 8 pts

This is the whole of shadows. *A shadow is not a darkness that gets drawn.* It is a question about **visibility** — can this point see the lamp? — and the scene already knows how to answer it. In `illuminate`, before the lines for E and k:

1. Send a ray from $\mathbf{x}$ along $\mathbf{l}$, toward the lamp.
2. Ask the scene what it hits, but only between 0.00001 and r: `scene.hit(Ray(x, l), 0.00001, r)`.
3. If it hits anything, this lamp's share is black.

Both ends of that interval are needed.

r, not infinity. A ray sent toward a lamp keeps going past it, so a surface on the *far side* of the lamp would "block" it.

0.00001**, not** 0**.** The shadow ray starts exactly *on* a surface, so with 0 it hits that same surface at once, and **every point in the scene reports itself as shadowed**: a black render, from one missing number. A4 called this **shadow acne**; it is the same bug, in 3D.

`scene` is the module-level name near the top of `tracer.py` (section C above).

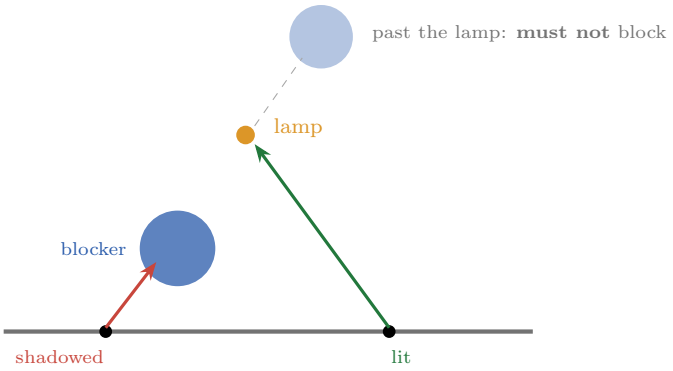

Figure 3: A shadow ray asks one question: between me and the lamp, is there anything? The two ends of the interval are what make the answer correct: 0.00001 at the near end, the distance r to the lamp at the far end.

Every point is either lit by a lamp or not, which is why point lights give sharp shadow edges. Soft shadows need a lamp with *size*, sampled many times: that was A4's area light, and it returns in Lecture 21.

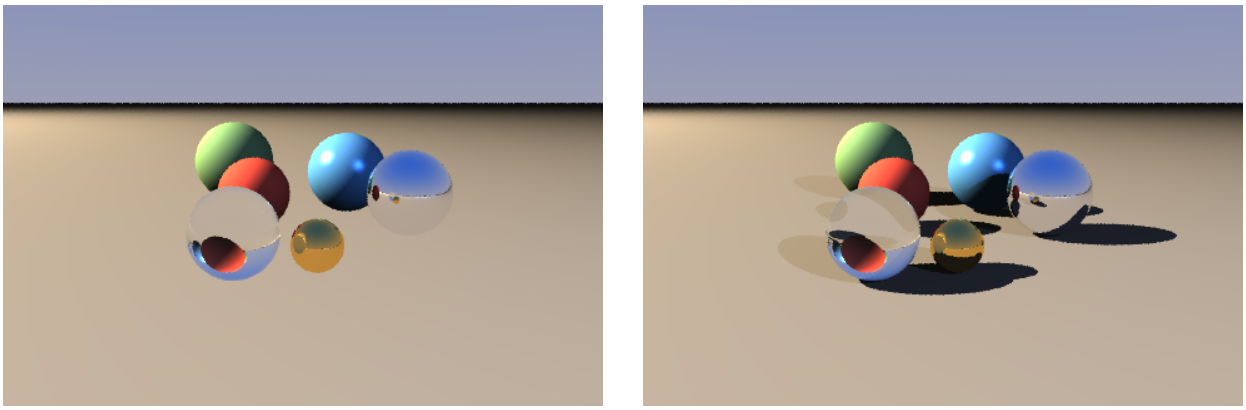

Figure 4: The exemplar scene with the shadow ray left out (left) and put in (right). Nothing else changed. Without shadows the balls float free of the ground; the shadows are what put them *in* the scene.

T6 · trace: the sky and matte surfaces · 8 pts

`trace(ray, depth)` returns the colour seen along a ray. It asks the scene for the nearest hit, with `scene.hit(ray)`, and then does one of several things. T6 is the first two.

A miss. The record's `t` is `math.inf`. Return the sky's colour, a blend of the two sky colours by the height of the ray's direction:

```
a = 0.5 * (ray.d.y + 1.0)
return sky_bottom.lerp(sky_top, a)
```

Test the miss *first*. Everything after it may then assume that there is a hit.

A hit on a Lambertian or a BlinnPhong surface. Start from black and add the share of every light in the list:

```
c = Vec3(0,0,0)
for light in lights:
    c = c + light.illuminate(ray, hit)
return c
```

Light adds. Two lamps on one surface give the *sum* of what each delivers; light does not average. The loop does not know what kind of light it has in hand: a lamp answers with $k\,E$, the ambient light with $R\,I_a$.

Which material is it? Ask with `isinstance`:

```
material = hit.surface.material
if isinstance(material, (Lambertian, BlinnPhong)):
    ...
elif isinstance(material, Reflective):      # T7
    ...
elif isinstance(material, Glass):           # T10
    ...
```

`depth` is not used by these two branches. A matte hit ends the ray.

T7 · trace: mirrors · 8 pts

The idea this assignment is built toward: the ray does not stop. A hit on a `Reflective` surface asks no lamp at all. Its colour is whatever the *bounced* ray sees:

$$\mathbf{r} \;=\; \mathbf{d} - 2(\mathbf{d}\cdot\mathbf{n})\,\mathbf{n} \qquad\qquad \text{colour} \;=\; R \;\cdot\; \texttt{trace}(\text{the ray from } \mathbf{x} \text{ along } \mathbf{r},\; \texttt{depth} - 1)$$

Angle in equals angle out. The new ray starts at the hit point, and `scene.hit`'s default $t_0 = 0.00001$ is what stops it from hitting the mirror it has just left.

If `depth` is 0, return black. Without this test, two mirrors that face each other call `trace` until Python's stack gives out. Each call passes on `depth - 1`, so the chain ends.

The multiplication by R is where a gold mirror gets its gold. A mirror of albedo $(1.0, 0.8, 0.4)$ removes some blue from everything it reflects, *once for every bounce*, so a reflection of a reflection is twice as gold.

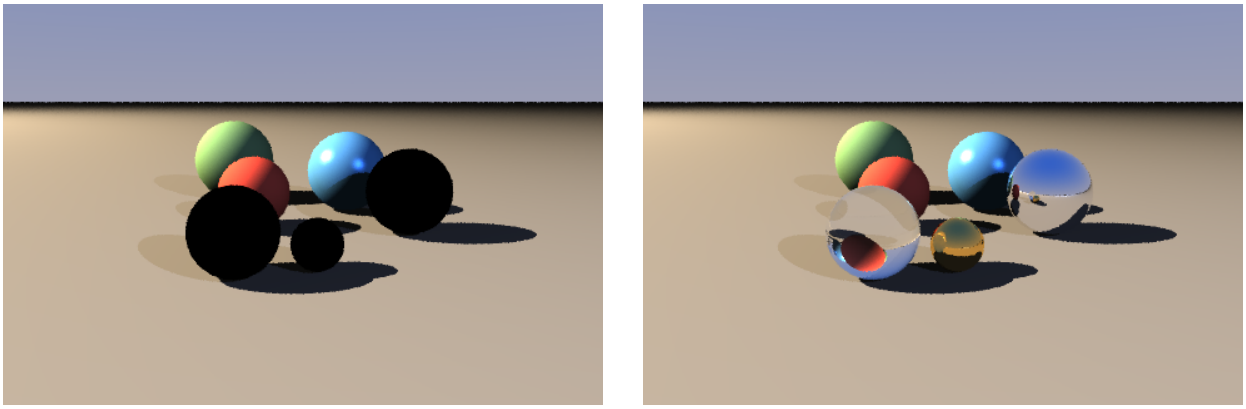

Figure 5: The exemplar scene traced with `depth = 0` (left) and 8 (right). With no bounce allowed, every mirror and every glass surface is black, because *all* of their colour comes from where the ray goes next. The matte surfaces are the same in both.

T8 · refract · 8 pts

Light travels slower in glass than in air, so a ray that crosses the boundary at an angle changes direction: the reason a straw in a glass of water looks broken. **Snell's law** states it:

$$n_i \sin\theta_i \;=\; n_t \sin\theta_t$$

n_i is the refractive index on the side the ray comes *from*, and n_t on the side it goes *into*. Air into glass is $n_i = 1.0,\ n_t = 1.5$; glass back out to air is the other way round.

`refract(d, n, ni, nt)` returns the bent direction $\mathbf{t}$. $\mathbf{d}$ is the unit direction of the ray and $\mathbf{n}$ the unit normal on the side the ray comes from, so $\mathbf{d}\cdot\mathbf{n}$ is negative. As in lecture 18, $\mathbf{t}$ is built from two parts, one perpendicular to the normal (along the surface) and one parallel to the normal:

$$\mathbf{t}_\perp = \frac{n_i}{n_t}\big(\mathbf{d} - (\mathbf{d}\cdot\mathbf{n})\,\mathbf{n}\big) \qquad \mathbf{t}_\parallel = -\sqrt{1 - \frac{n_i^2\,\big(1-(\mathbf{d}\cdot\mathbf{n})^2\big)}{n_t^2}}\;\mathbf{n} \qquad \mathbf{t} = \mathbf{t}_\perp + \mathbf{t}_\parallel$$

Total internal reflection. On the way *out* of glass at a shallow angle, the number under the square root is negative: Snell's law asks for a sine greater than 1, and no angle has one. The ray cannot leave; it is reflected back inside. Test for it *before* the square root, and return `None`.

This is not an obscure case. It is why the underside of a glass marble looks silvered, why optical fibre works, and why the surface of a swimming pool looks like a mirror from underneath.

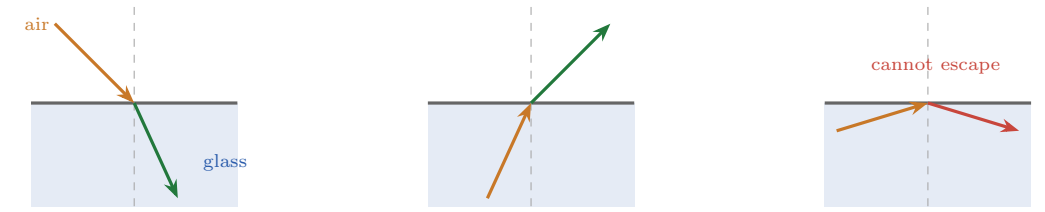

Figure 6: The three things a ray can do at a glass boundary. The third is what `refract` reports by returning `None`.

T9 · schlick · 4 pts

Glass does *both* things at once: some light passes through and some bounces off, and how much of each depends on the angle. Look straight down into a pond and you see the fish; look across it at a shallow angle and you see the sky.

The exact answer is the Fresnel equations, which are long. Christophe Schlick's 1994 approximation is two lines:

$$R_0 = \left(\frac{n_t - n_i}{n_t + n_i}\right)^2 \qquad R(\theta) = R_0 + (1 - R_0)\,(1 - \cos\theta)^5$$

R is the fraction of the light that is *reflected*. R_0 is its value head on: 0.04 for glass in air, which is why a window that you face directly is mostly transparent. The fifth power keeps R near R_0 until the angle is steep, and then takes it quickly to 1.

T10 · shade_glass · 9 pts

The colour of a hit on glass. It traces **two** rays, the bounced one and the bent one, and adds their colours, weighted by Schlick's R:

$$\text{colour} \;=\; R\,c_r + (1 - R)\,c_t$$

1. **If `depth` is 0, return black.**
2. **Which way is the ray going?** `rec.front_face` answers. Into the glass: $n_i = 1.0$, $n_t =$ the glass's index. Out of it: the other way round.
3. **The bounced ray.** **r** as for a mirror; $c_r =$ `trace` of the ray from **x** along **r**, at `depth - 1`.
4. **The bent ray.** **t** = `refract(d, n, ni, nt)`. If it is `None`, return c_r alone: all of the light was reflected.
5. Otherwise $c_t =$ `trace` of the ray from **x** along **t**, at `depth - 1`, and $R =$ `schlick(-d.dot(n), ni, nt)`.

Then give `trace` its last branch: a hit on `Glass` returns `shade_glass(ray, hit, depth)`.

Getting n_i and n_t the wrong way round is the classic bug: the ball renders as a dark blob in place of a lens.

Why both rays, and not one chosen at random. You could follow the bounced ray with probability R and the bent one otherwise; averaged over enough samples that is correct, and it is what A9's path tracer will do. But "enough" means *hundreds* of samples per pixel. At the few you can afford this week, choosing at random shows as speckle all over the glass. Tracing both costs one more ray and is exact.

A ray that passes straight through a glass ball uses up two steps of `depth`: one going in, one coming out. That is why `depth = 1` is not enough to see through glass.

Bonus · FuzzyMetal · 5 extra pts

Not part of the 70. Brushed metal: a mirror whose bounce is nudged a little at random, so that its reflections are blurred.

In `material.py`: the class `FuzzyMetal(albedo, fuzz)`, which stores its albedo and stores `fuzz` **clamped into** $[0, 1]$. A fuzz of 0 is a perfect mirror; 0.3 is brushed aluminium.

In `trace`: one more `elif`. It is the `Reflective` branch with two more lines after **r** is computed:

```
    r = r + material.fuzz * random_unit_vector()
    if r.dot(n) <= 0:
        return Vec3(0,0,0)       # nudged into the surface
```

`random_unit_vector` is given. Call it exactly **once** for each hit, or the checker's numbers will not match yours. A fuzzy surface needs many samples per pixel to look smooth: at 4 it is grainy, at 50 it is brushed metal.

The checker reports the bonus on its own line, after the total. The 5 points are added to your assignment mark, which cannot go above 100.

Part B — Your Lit Scene (30 pts)

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

The rule from A5 has not changed: you are not drawing this picture. You place objects *and lamps* in the world, put a camera somewhere, and the picture is what the camera reports. Move only the camera and the whole thing recomposes itself — including every shadow, every reflection and every refraction, all correctly.

The subject, the style and the mood are yours. A still life, a planet and its moons, a hall of mirrors, something abstract or impossible: all are welcome. The B1 checklist is the only hard constraint.

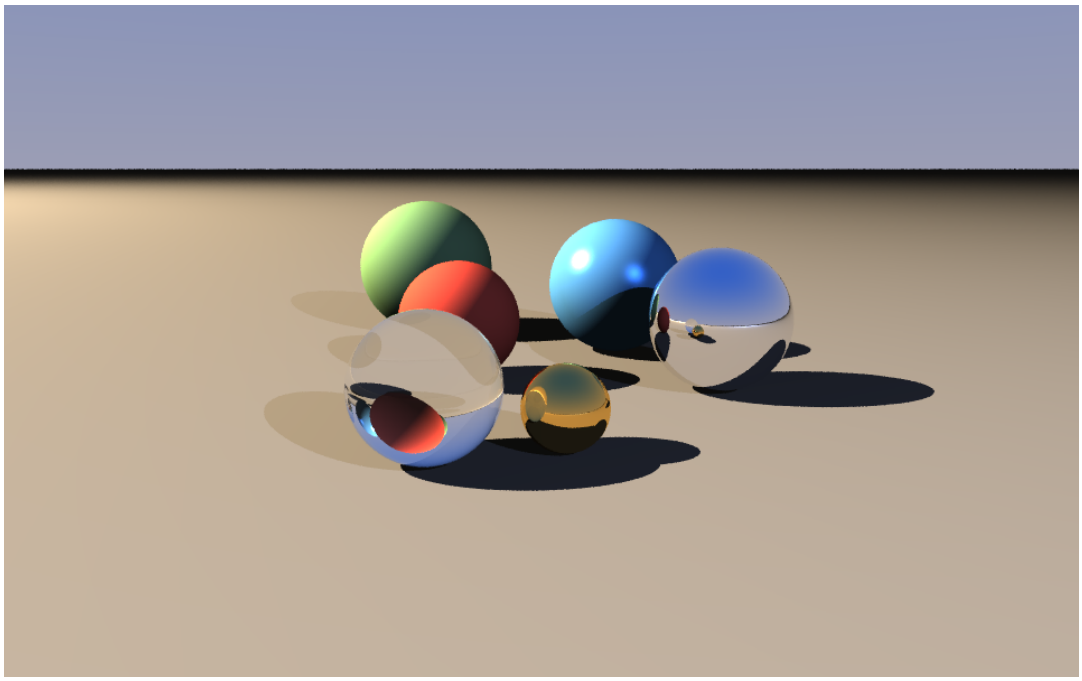

Figure 7: The instructor exemplar, "Late Afternoon, Four Materials" (960×600, 9 samples per pixel). One warm key lamp, far off to the left, throws the long shadows; a dimmer cool fill from the right keeps the dark sides from going black. The glass ball sits in front of the red one *on purpose*: the red ball seen through it, small and upside down, is the most convincing proof that refraction works. The mirror is placed where it catches the other balls and not blank ground. The blue ball is Blinn–Phong, and carries the key lamp's highlight. Your scene should be *yours*, not this one.

Workflow — follow this order

1. Keep the canvas **small** (240×150) and `SAMPLES_PER_PIXEL = 1` while you iterate.
2. Get the **layout** right, then the **lights**. In that order, and spend real time on the lights — they matter more to the final picture than the objects do.
3. Add the **mirrors and the glass last**. They are the slowest things in the scene and the hardest to judge small.
4. *Then* raise the resolution and set `SAMPLES_PER_PIXEL` to 4 or more.

Lighting, in four rules

Almost every mark in B2's lighting criterion comes down to these.

1. One lamp does the work. A single bright lamp, off to one side, gives strong readable shadows and a clear sense of shape. Lighting everything evenly from the front — from beside the camera — is the fastest way to make a 3D scene look flat, and it is what most first attempts do.

2. The second lamp is dimmer, and a different colour. Its job is to fill the dark sides so that they are not black. Warm key, cool fill is the standard setup on a film set, and it is two lines of code here.

3. Low lamps make long shadows, and long shadows make depth. A lamp directly overhead throws a short shadow straight underneath each object. Lower it and push it sideways, and the shadows stretch across the floor and tie everything together.

4. Near lamps make pools, far lamps make daylight. A lamp a few units from the scene lights a bright pool around itself, and the floor fades to black a little further out: a room at night. A lamp a hundred units away or more lights everything in view almost evenly, like the sun; the exemplar's key lamp is about 175 away, with an intensity near 200 000. Both are right; choose on purpose, and remember that the intensity must grow with the square of the distance (T4).

Four traps worth naming

1. Intensity 1 looks like nothing. The division by r^2 and by π is doing that, not a bug in your code. Start at about $3r^2$ (T4).

2. A mirror in an empty corner looks like a grey ball. A mirror has no colour of its own: it shows you the rest of the scene. Put something *worth reflecting* where it can see it.

3. Glass with nothing behind it is invisible. The whole point of glass is what you see *through* it. Put a colourful object directly behind your glass ball, and it will appear through the glass small and upside down.

4. Too much ambient light ruins everything. It is the fastest way to erase the shadows you have just implemented. Keep it between 0.05 and 0.1; see Figure 1 for what happens if you do not.

If your render is slow

It will be, more than A5 was. In order of how much they buy you: **use PyPy**; **turn `SAMPLES_PER_PIXEL` down** while iterating; **lower `MAX_DEPTH` to 2 or 3** while composing (reflections of reflections go dark, but layout and lighting still read); **render small**.

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 surfaces**, including a ground plane	1
At least one **Reflective**, one **Glass** and one **BlinnPhong** surface	3
At least **2 point lights**, of different colours, and an ambient light	2
A visible **shadow** that clearly belongs to a specific object	3
Final render uses `SAMPLES_PER_PIXEL` ≥ 4 (antialiasing on)	2
Every sphere **sits on** the ground ($y = y_0 + r$)	1
The render is **repeatable** — the same image on every run	1
Total	**16**

Two of these get faked every year, so to be explicit. The **glass must be visibly glass** — something recognisable seen through it, not a transparent ball against an empty background where nobody can tell. And the **shadow must be attributable**: a grader should be able to point at a dark patch and say which object made it.

B2 · Craft Rubric · 14 pts

Criterion	0 — absent	1 — attempted	2 — competent	3 — excellent
Lighting design (worth 0, 1, 3, 5)	One lamp beside the camera; flat	Lit from somewhere, but shadows are stubby or absent	A clear key light with readable shadows, plus a fill	Key and fill deliberately placed and coloured; the light is the idea
Materials	Everything matte, or mirror and glass present but unreadable	Mirror and glass present; neither shows what it does	Both read clearly: reflections and refraction visible	Materials placed to explain each other; coloured mirrors and highlights used well
Composition & camera	Camera arbitrary; subject clipped or tiny	Everything in frame, but the viewpoint says nothing	Deliberate camera; the frame is balanced and the subject reads	The viewpoint IS the idea — angle, height and fov chosen together
Originality & ambition	Reproduces the starter's scene, or the handout's	Derivative with small changes	Clearly original; a safe idea, carried out cleanly	A bold idea of your own. Rough edges are forgiven; playing safe is not rewarded

Lighting design is worth 0, 1, 3 or 5 for its four columns (it is what this assignment is about); the other three are worth 0 to 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 scene, lightly recoloured, caps *Originality* at 1.

Two cheap upgrades, both nearly free. First, **move your key lamp lower and further to the side**: it costs one line and it is the difference between short shadows and long ones. Second, **put a colourful object directly behind your glass**: the refracted, flipped image of it is the most impressive thing in an A6 render, and there is nothing to see if nothing is there to refract.

Submission

Submit **one zip file** named `RollNo_A6.zip` (e.g. `231234567_A6.zip`) on the course website, containing exactly:

`material.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: `vec3.py`, `vec2.py`, `canvas.py`, `painter.py`, `check.py`, `expected_a6.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 Lighting Cheat Sheet

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

You want	Call	The idea
a matte surface's factor	`Lambertian.evaluate(l, v, n)`	R/π
the same with a highlight	`BlinnPhong.evaluate(l, v, n)`	$R/\pi + R\max(0, \mathbf{n}\cdot\mathbf{h})^p$
the ambient light's share	`AmbientLight.illuminate(ray, rec)`	$R\,I_a$; no arrows, no shadow
a lamp's share	`PointLight.illuminate(ray, rec)`	$k\,\max(0, \mathbf{n}\cdot\mathbf{l})\,I/r^2$, or black in shadow
is this point in shadow	`scene.hit(Ray(x, l), 0.00001, r)`	a ray toward the lamp; both ends matter
the nearest hit of a ray	`scene.hit(ray)`	a miss has `t == math.inf`
the bounce off a mirror	$\mathbf{d} - 2(\mathbf{d}\cdot\mathbf{n})\mathbf{n}$	then `albedo * trace(..., depth - 1)`
the bend through glass	`refract(d, n, ni, nt)`	Snell; `None` means it cannot get out
how much glass reflects	`schlick(cos_theta, ni, nt)`	0.04 head on, 1 at a grazing angle
the colour of glass	`shade_glass(ray, rec, depth)`	$R\,c_r + (1-R)\,c_t$
the colour of a ray	`trace(ray, depth)`	the sky, or one branch for each kind of material

The single rule to carry away: light is not painted onto surfaces. It arrives from somewhere, it is blocked by things in the way, and it keeps travelling after it hits.

Appendix B: Debugging a Lit Ray Tracer

Rendering bugs are quiet: you get an image, it is just the wrong one. Almost all of A6's are on this list.

Symptom	Almost always
Matte surfaces are black, or nearly	the shadow ray starts at $t_0 = 0$: every point shadows itself (T5)
Everything is dark but not black	only the ambient share is arriving: `PointLight.illuminate` returns black (T4)
Lit faces are dark, dark faces are lit	**l** points from the lamp to the surface; it must point the other way (T4)
Barely visible even with a bright lamp	intensity too low for the distance. Try $3r^2$ (T4)
Lit surfaces are far too bright	the π is missing from `evaluate` (T1, T2)
No shadows at all	the shadow ray's interval is empty, or its result is never tested (T5)
Shadows in impossible places	surfaces *beyond* the lamp are blocking: t_1 is not r (T5)
Shadows are grey; no contrast	the ambient light is far too strong
No highlight on a BlinnPhong ball	**v** points along the ray; it must point back toward the eye (T4)
The highlight is a huge white disc	the exponent is missing, or far too small (T2)
Mirrors and glass are pure black	`depth` is 0 where `render` is called, or `trace` passes on `depth` in place of `depth - 1` and Python stops it
`TypeError` about `NoneType`	`trace` has no branch for that material, and returned nothing
The glass ball is a dark blob	n_i and n_t are the wrong way round: `front_face` used backwards (T10)
Speckles on the glass	choosing one of the two rays at random; trace both (T10)
A mirror looks like a grey ball	it has nothing worth reflecting near it: a scene problem, not a code one
The far floor is black	the lamp is near, and $1/r^2$ is doing its work. Move the lamp far away and raise its intensity
`RecursionError`	a mirror or glass branch does not test `depth == 0`, or does not pass `depth - 1` (T7, T10)

The best single debugging move is to turn things off. Render with one lamp and only matte surfaces first: if they are lit and shadowed correctly, T1–T6 are right, and any remaining bug is in T7–T10. `MAX_DEPTH = 0` is the other half of the same trick: it shows everything that does *not* depend on recursion.

Appendix C: A Lit Scene in Twenty-Five Lines

The skeleton every Part B scene shares. Everything interesting is in `build_scene`, `build_lights` and the camera.

```
GROUND_Y = 0.0

def build_scene():
    scene = Group()
    scene.add(Plane(Vec3(0, GROUND_Y, 0), Vec3(0, 1, 0),
                    Lambertian(Vec3(0.5, 0.5, 0.55))))

    # (x, z, radius, material) - the centre is at GROUND_Y + radius,
    # which is what makes a sphere REST on the plane
```

```
    for (x, z, r, material) in MY_LAYOUT:
        scene.add(Sphere(Vec3(x, GROUND_Y + r, z), r, material))
    return scene

def build_lights():
    return [
        AmbientLight(0.08),
        # KEY: bright, warm, far off to one side. Casts the shadows.
        PointLight(Vec3(-40, 35, 10), Vec3(16000, 14800, 12600)),
        # FILL: dimmer, cooler, from the other side. Softens the darks.
        PointLight(Vec3(25, 20, 20), Vec3(1100, 1400, 2200)),
    ]

def render_scene():
    tracer.scene = build_scene()         # the two module-level names
    tracer.lights = build_lights()
    tracer.sky_bottom = SKY_BOTTOM
    tracer.sky_top = SKY_TOP
    img = Canvas(WIDTH, HEIGHT, (0, 0, 0))
    camera = Camera(CAMERA_POSITION, CAMERA_TARGET, CAMERA_FOV,
                    WIDTH / HEIGHT)      # move THIS and everything follows
    tracer.render(img, camera, SAMPLES_PER_PIXEL, MAX_DEPTH)
    return img
```

Note what is *not* here: nothing decides where anything appears on the screen, and nothing paints a shadow. The camera decides the first; the shadow rays decide the second. The moment you find yourself darkening a patch of ground by hand to make it look right, you have started drawing again.