Assignment 7: Light That Bounces

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

Deadline: Wednesday 11th November 2026, **11:59 pm**

Introduction

Lecture 19 opened with three photographs your renderer cannot make. A white wall beside a red one, gone pink. A ceiling with no lamp pointing at it, lit anyway. The corner where two walls meet, darker than either wall. All three are the same thing: **light that arrived somewhere after bouncing off something else first**, and A6 did not follow it.

A6 did not follow it because A6's chalk asked a question that has no room for it. At a matte hit it asked *which lamps can see this point?*, added up their contributions, and stopped. Light that had come off the red wall was never a lamp, so it was never counted. The hole it left was papered over with `AmbientLight` — one flat number, 0.06, added everywhere, which is the same in a corner as in the open and the same beside a red wall as beside a white one.

This week the chalk asks a different question: *where did the light arriving here come from?* And since it cannot know, it **guesses** — it fires one ray in a random direction and asks what that ray sees — and then you average a great many guesses.

That is the whole assignment. It is about forty lines of code, of which yours are thirteen, and it is the largest change in the look of your renderer that this course contains.

The code is small, so the picture is the assignment. Part A is 30 points. **Part B — your scene — is 70.**

Objectives

- A lamp as a **surface that glows**, not a point with no size
- **Estimating** the rendering equation's integral instead of evaluating it: one random direction, many times, averaged
- **Indirect light** — colour bleeding, surfaces lit with no lamp in view, and corners that go dark on their own
- **Soft shadows** for free, because a lamp now has an area
- **Noise**, where it comes from, and the $1/\sqrt{N}$ that governs it
- **Gamma**: the difference between the light you computed and the number the screen wants
- Composing a picture when the light is no longer under your direct control

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 A7 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`.

This applies to the scene as much as to the code, and this week the scene is most of the marks. `NOTES.md` (see Part B) asks you to point at where the bounced light is in your own picture and say what put it there. Write it from your own image.

Getting Started

A) File Setup

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

`material.py`	(edit this) one new class — **T1 is here**
`tracer.py`	(edit this) the bounce and the screen — **T1–T3**
`scene.py`	(edit this) your scene — **all of Part B**
`strip.py`	*do not edit* — makes `convergence.ppm`. **You must submit its output**
`vec3.py`	*do not edit* — from A5
`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_a7.py`	*do not edit* — reference data used by `check.py`

`material.py` and `tracer.py` are A6's reference solution, handed back to you working, with this week's three jobs cut out of them. Nobody is blocked by an A6 bug. If you want to carry your own A6 files over instead, you may — but the given ones are the ones the checker was built against, and the three stubs are marked in both.

Standard-library Python throughout: no `pip`, no packages.

```
python3 check.py       # score Part A, as often as you like
python3 scene.py       # works, but slow
pypy3 scene.py         # the same picture, many times faster
pypy3 strip.py         # the convergence strip you must submit
```

Use PyPy from the first render. A path tracer casts tens of millions of rays for one clean picture. This is the week where the difference between `python3` and `pypy3` stops being a convenience and becomes the difference between composing your scene and waiting for it.

B) First, Watch A6 Break

Before you write anything, do this. Copy your **A6 scene.py** into this folder, run it, and look at what comes out.

It will still import — `AmbientLight` and `PointLight` are both still in `tracer.py` — and it will render

something wrong, because nothing calls them any more. Every matte surface in your old scene will be black, and only the mirrors and the glass will show anything at all.

That is the honest state of the renderer right now: **there is no light in your scene**, because in A7 light comes from surfaces and your A6 scene has none. The rest of the week is about putting it back, this time for real.

C) What Actually Changes

	A6 asked	**A7 asks**
at a matte hit	which lamps can see this point?	where did the light arriving here come from?
a lamp is	a point, with a position and no size	a surface that glows, hit by rays like any other
a shadow is	computed, with a shadow ray per lamp	what happens when a path fails to reach a lamp
a soft shadow	impossible: a point is either visible or not	automatic, because the lamp has an area
indirect light	a constant, `AMBIENT`	computed, because the ray keeps going
the answer is	clean, and wrong	grainy, and right

That last row is the week. A6's picture was perfectly smooth and had no idea what light does in a room. A7's has grain in it and is, in the limit, *correct* — and the grain goes away if you are willing to wait.

The A7 API at a glance. The left column is **given** and already works; the right column is **yours**.

Given (all of A5 and A6)	**Yours (T1–T3)**
`Vec3`, `Ray`, `Sphere`, `Plane`, `Group`	`Emissive` — in `material.py`
`HitRecord`, `Camera`, `render`	the `Emissive` branch of `trace`
`Lambertian`, `BlinnPhong`, `Reflective`, `Glass`	the matte branch of `trace`
`random_unit_vector`	`to_display`
`refract`, `schlick`, `shade_glass`	*bonus:* `uniform_bounce`
the mirror and glass branches of `trace`	

D) Where the π Went

You are about to write, as the entire body of a matte hit, `material.albedo * trace(...)`. Lecture 19 said chalk's BRDF is $\rho = R/\pi$. There is no π in that line, and it is important that you know this is not a shortcut — it **cancelled**.

Lecture 19 ended at the rendering equation and the fact that its integral has no closed form:

$$L_o(\mathbf{x}, \boldsymbol{\omega}_o) = L_e(\mathbf{x}, \boldsymbol{\omega}_o) + \int_\Omega \rho \; L_i(\mathbf{x}, \boldsymbol{\omega}_i) \; \cos\theta \; d\sigma(\boldsymbol{\omega}_i)$$

Here is the one new idea of the week, and it is one line. If you cannot add up every direction, pick *one at random* and divide by how likely you were to pick it:

$$\int_\Omega f(\boldsymbol{\omega}) \, d\sigma \;\approx\; \frac{f(\boldsymbol{\omega}_1)}{p(\boldsymbol{\omega}_1)} \qquad \text{and, over } N \text{ picks,} \qquad \frac{1}{N} \sum_{k=1}^{N} \frac{f(\boldsymbol{\omega}_k)}{p(\boldsymbol{\omega}_k)}$$

That is the whole of Monte Carlo integration. The division by p is what makes it honest: a direction you were twice as likely to pick counts for half as much, so no direction gets over-counted just because you favoured it. The estimate is *wrong every single time* and *right on average*, and its error falls as $1/\sqrt{N}$ — which is the grain in your picture, and why four times the samples halves it.

Now put the two together. `normal + random_unit_vector()` does not pick directions evenly: it picks them with $p(\boldsymbol{\omega}) = \cos\theta/\pi$, clustered around the normal. (That is a fact about that particular trick; take it on trust this week.) So for one sample on chalk:

$$\frac{\rho\, L_i \cos\theta}{p} = \frac{(R/\pi)\, L_i \cos\theta}{\cos\theta/\pi} = \frac{R}{\pi} \cdot L_i \cos\theta \cdot \frac{\pi}{\cos\theta} = R\, L_i$$

Both π's gone, the cosine gone. **Multiply by the albedo and recurse.** That is why T2 is one line of arithmetic and not four.

This cancellation is not luck. Picking directions in proportion to how much they contribute — which is what the cosine does here — is called *importance sampling*, and it is the single most important technique in rendering. The bonus task makes you do it the other way, so that you can see what it was worth.

E) Light, and the Number the Screen Wants

There are two different kinds of number in your renderer this week, and mistaking one for the other is the single most expensive bug in the assignment. It costs an evening, every year, to several people.

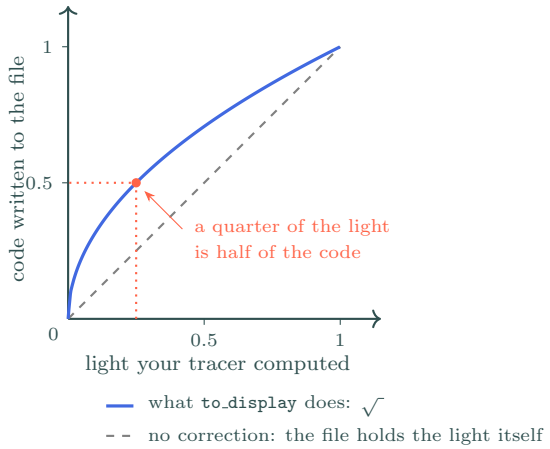
What `trace` returns is light. It is a radiance, in lecture 19's sense, and it is *linear*: twice the number means twice the photons. That is not a detail — it is the only reason the rest of the renderer is allowed to do arithmetic at all. Averaging a hundred samples, multiplying by an albedo, adding what two bounces brought back: every one of those operations assumes the numbers are proportional to light.

What a `.ppm` file holds is not light. It holds a *code* that the screen converts into light, and the conversion is deliberately not proportional:

$$\text{light out} \;\approx\; (\text{code sent})^{2.2}$$

Send 0.5 and you get about a fifth of the light of 1.0, not a half.

Why on earth. It began as physics: the electron gun in a CRT had roughly that response, so the signal was pre-distorted to cancel it. CRTs are gone and the curve stayed, on purpose, because it also happens to match the eye. We discriminate far better among dark tones than among bright ones, so an encoding that spends more of its 256 steps down at the dark end puts them where they can be seen. An 8-bit image stored *linearly* would band visibly in every shadow.



So the conversion happens once, at the very end. Raise to 1/2.2 to cancel the screen's 2.2. We use $\sqrt{}$, which is 1/2 — close enough that nobody can see the difference, and it is what *Ray Tracing in One Weekend* uses. (The real sRGB curve is piecewise, with a short linear segment near black. It does not matter here.)

Light	Byte, no $\sqrt{}$	Byte, with $\sqrt{}$	
0.01	2	25	a dark corner: invisible, or just visible
0.10	25	80	a shadowed floor
0.25	63	127	*this* is mid-grey, not 0.5
0.50	127	180	a well-lit wall
0.80	204	228	near-white; barely moves
1.00	255	255	the top, either way

Read the last two rows and then the first two. **The bright end hardly moves and the dark end moves by a factor of ten.** That is why the bug is easy to miss — your lamp still looks like a lamp — and why it ruins an A7 picture in particular.

Order matters, in two places, and both are in `render`.

1. **Average the samples first, then encode.** Encoding each sample and averaging the results is not the same number, and it is not the right one: it lifts your noise into a pale haze instead of letting the bright and dark errors cancel. Light adds; codes do not.
2. **Encode last, and only once.** `to_display` belongs between the average and `to_color()`, and nowhere else. A square root anywhere inside `trace` — on an albedo, on a lamp's radiance, on what a bounce returned — corrupts the arithmetic that follows it.

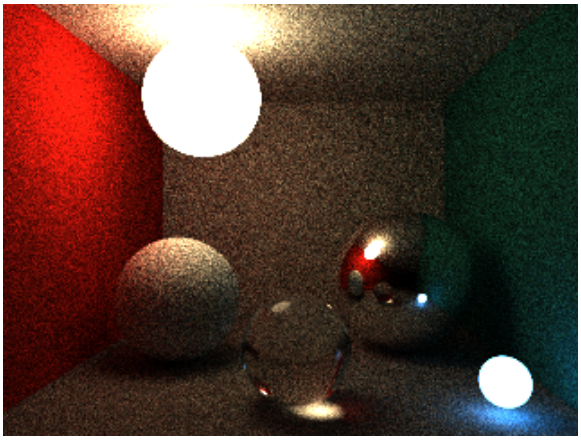
A note on the other direction. The albedos you pick by eye — `Vec3(0.8, 0.2, 0.2)` for a red wall — are strictly speaking already encoded numbers, of the kind a colour picker shows you. A "middle grey" that reads as 0.5 on screen is about $0.5^{2.2} = 0.22$ of the light. **You are not asked to correct for this**, because you are choosing albedos by eye and judging the result by eye, so the loop closes — but it explains why a wall you set to 0.5 comes out darker than you expected, and it is the reason real engines decode texture images on the way in.

Why A6 did not need this. It did, and it got away with it. A6's images were lit by lamps aimed squarely at the things you were meant to look at, so the picture lived at the bright end of the table where the curve barely bends; what it got wrong was down in the shadows, where there was nothing to see anyway. **An A7 image is made of mid-tones.** Every surface lit by bounced light — which is now most of them — sits in the middle of that table, which is exactly where the curve bends hardest. Leave the square root out of the exemplar and its average pixel falls from 109 to 62, and its *mid-tones* fall by more than half.

And the clamp is a tone map. Deciding that everything above 1 becomes pure white is the crudest possible answer to "this scene contains a lamp eight times brighter than paper, and the screen has one white". It is the right answer for this week. Lecture 35 replaces it with a curve that rolls the highlights off instead of cutting them, and lecture 30 writes this same square root in WGSL — so the move you are about to make by hand is one you will meet twice more on the GPU.

Part A — The Glow, The Bounce and The Screen (30 pts)

Three tasks. The stubs are marked in `material.py` and `tracer.py`, each with its spec in the docstring above it, and `check.py` will score you at any time.



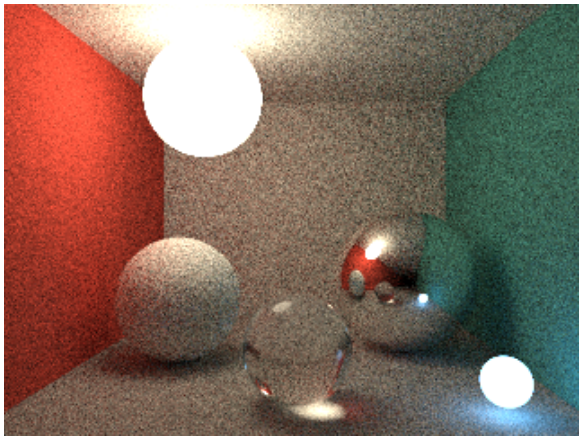


Figure 1: The same render, the same lamp, the same samples. **Left:** the clamp alone — T3's square root left out. **Right:** T3 as specified. The bright things barely move ($\sqrt{0.9} = 0.95$), which is why this is easy to miss; the mid-tones and the shadows move a great deal ($\sqrt{0.1} = 0.32$, three times brighter), and they are most of the picture. Compare the ceiling, the shadowed floor and the dark side of the white ball. This is one `math.sqrt`, and students lose evenings to it every year.

Task		**Where**	**Pts**
T1	`Emissive`, and the branch that ends a path	`material.py`, `tracer.py`	6
T2	the bounce	`tracer.py`	16
T3	`to_display`	`tracer.py`	8
Bonus	`uniform_bounce`	`tracer.py`	*+5*

T1 · the lamp is a surface · 6 pts

In `material.py`, a class with one line in it:

```
class Emissive:
    def __init__(self, color):
        self.color = color      # this is L_e
```

In `trace`, a branch that must come *before* the matte one:

```
if isinstance(material, Emissive):
    return material.color
```

No recursion. No multiplication. **No depth test.** A lamp is as bright as it is, however many bounces the path took to arrive, and there is nothing on the far side of it to ask about.

The numbers go above 1, and that is the point. An albedo is a fraction — a surface cannot reflect more light than it receives, so it lives in $[0, 1]$. A radiance is not a fraction and has no ceiling. A lamp that is to light a room is `Vec3(8, 8, 8)`; a lamp of `Vec3(1, 1, 1)` is about as bright as a sheet of white paper in daylight and will light nothing at all. If your first render is black, this is the first thing to check.

What you have just bought, for free:

- **Soft shadows.** A point in the band at the edge of a shadow can see part of the lamp, so some of its

random rays reach it and some do not. The fraction that do is exactly the brightness of the penumbra. Nobody writes a line of code for this; it is what a lamp with an area does.

- **The $1/r^2$ falloff.** A lamp twice as far away covers a quarter as much of the dome of directions above a point, so a quarter as many of that point's random rays find it.
- **Lamps you can see.** Look straight at it and you see the lamp, because it is a surface and rays hit it.

T2 · the bounce · 16 pts

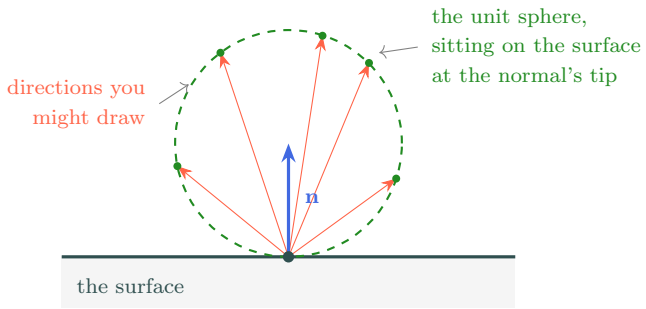
The matte branch of `trace`. This replaces A6's loop over `lights` entirely — that loop is gone, and `lights` is never read again.

```
elif isinstance(material, (Lambertian, BlinnPhong)):
    if depth == 0:
        return Vec3(0, 0, 0)
    x = ray.evaluate(hit.t)
    n = hit.normal
    d = n + random_unit_vector()
    if d.length_squared() < 1e-12:
        d = n
    return material.albedo * trace(Ray(x, d), depth - 1)
```

Six lines, each load-bearing:

1. `if depth == 0: return Vec3(0,0,0).` Without it the recursion never ends: a room of matte walls has no exit, and the ray bounces until Python's stack gives out. Black is the honest answer — we stopped looking, so we found nothing. It is the same test A6 put on its mirror, for the same reason.

2. `n + random_unit_vector().` A random unit vector is a random point on a sphere of radius 1. Add the normal and that sphere sits *on* the surface, touching it at the hit point — so every direction you can draw from it points away from the surface, and they cluster around the normal. The cluster is the $\cos\theta$ of lecture 19, arriving for free.



3. The guard. If `random_unit_vector()` happens to come back as very nearly $-\mathbf{n}$, then `d` is very nearly zero. `Ray`'s constructor normalizes, the zero vector normalizes to the zero vector, and a ray with no direction divides by zero inside `Sphere.hit`. It is rare and it *will* happen in a render of a hundred thousand pixels.

4. `material.albedo *.` The fraction that survives the bounce, applied *once per bounce*. This is where a red wall makes the light red — and where a path through three walls of albedo 0.5 comes home at one eighth strength. It is also why bright rooms need albedos near 0.8: at 0.4 the light dies after two bounces.

5. `trace(..., depth - 1).` The recursion. Your chalk is now a mirror that has forgotten which way it was pointing.

Call `random_unit_vector()` exactly ONCE per matte hit, after the depth test. The checker seeds Python's random module and compares numbers. Call it twice, or call it before testing the depth, and your pictures will be perfectly fine and your marks will not be.

Blinn-Phong goes down the same branch, by its albedo — note that the `isinstance` test names both. Its highlight was computed from the direction to a lamp, and there is no lamp to ask. That is not a loss: a Blinn-Phong highlight was always a cheap imitation of a blurred mirror reflection, and you have the real thing in `Reflective`.

T3 · `to_display` · 8 pts

```
def to_display(color):
    def channel(v):
        v = max(0.0, min(1.0, v))
        return math.sqrt(v)
    return Vec3(channel(color.x), channel(color.y), channel(color.z))
```

Clamp first, then the square root, per channel. `render` calls this on each pixel after averaging the samples and before `to_color()` — and nothing else in the renderer may call it.

The two steps are the two halves of section E, and that is where the reasoning is. In short: the clamp is there because a radiance has no ceiling and a screen does, and the square root is there because a screen raises what it is sent to the power of about 2.2 before lighting the pixel. Figure 1 is what the exemplar looks like with the root left out.

The marks are in the details section E spells out, so if `t3a` or `t3b` fails, check in this order: are you clamping *before* the root, so that a negative channel never reaches `math.sqrt`? Is it $\sqrt{v}$ and not $v/2.2$ or $v^{2.2}$ — both of which are plausible-looking and make the picture darker still? And is `render` averaging before it calls you, rather than after?

Bonus · `uniform_bounce` · +5 pts

Pick the direction the *other* way — evenly over the dome, every direction above the surface equally likely — and make it produce the same picture.

```
d = random_unit_vector()
if d.dot(n) < 0:
    d = -d                          # flip it to the right side
return 2 * albedo * n.dot(d) * trace(Ray(x, d), depth - 1)
```

The $2\cos\theta$ is the estimator again, with the even pick's $p = 1/(2\pi)$ in place of $\cos\theta/\pi$:

$$\frac{\rho L \cos\theta}{p} = \frac{R}{\pi} L \cos\theta \cdot 2\pi = 2RL\cos\theta$$

Write it as a separate function (the stub is there); `check.py` calls it by name and still grades `trace` on T2's behaviour.

To see the point, call it from `trace`'s matte branch instead of your T2 code, render your Part B scene at the same sample count, and put the two side by side. They converge to the same image and the even pick is visibly noisier — it spends most of its samples near the horizon, where the cosine means they barely count. Put T2 back afterwards. Mention what you saw in `NOTES.md` and the bonus is yours.

Part B — A Room Full of Bounced Light (70 pts)

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

> **Every light in your scene is a surface. No `PointLight`, no `AmbientLight` — and the picture must show light that has BOUNCED.**

A6's scene rule was about materials. This week's is about **light**. A picture with one bright lamp and everything else black could have been made in A6 with a shadow ray. What could not: a wall lit only by light reflected off the floor; a white surface gone pink because it sits beside a red one; the soft darkening where two surfaces meet.

Part B is 70 of this week's 100 points. That is deliberate. The code is thirteen lines; the understanding is in what you do with it.

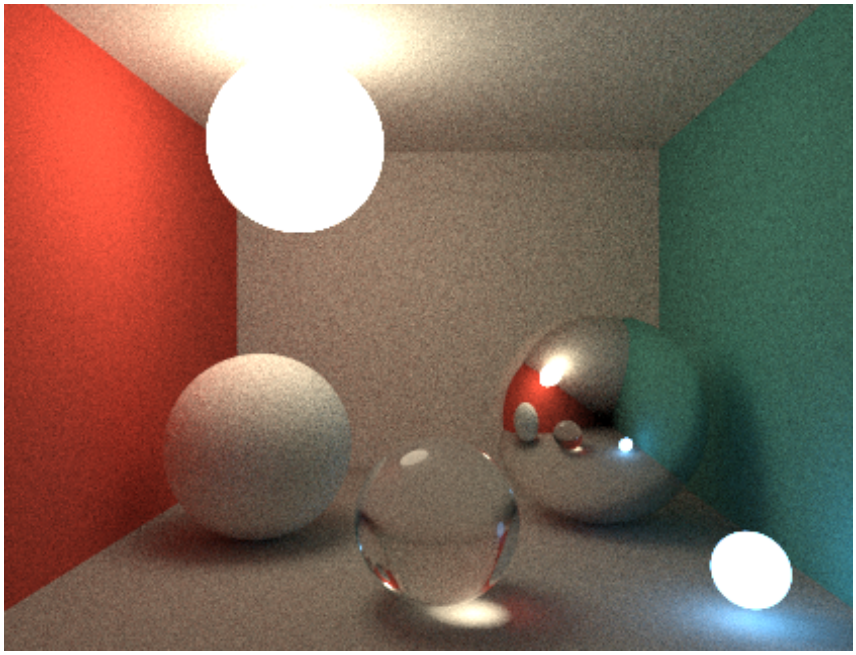


Figure 2: The instructor exemplar, "The Red Room" (420×320, 512 samples per pixel, 10 bounces, about four minutes under PyPy). **There are two lights in this scene and both are spheres.** Everything else you can see is light that has bounced: the warm tint down the left of the white ball, thrown by a wall that emits nothing; the red in the mirror; the lit ceiling, which no lamp points at; the soft band at the edge of every shadow, which exists only because the lamp has a radius. There is still visible grain at 512 samples — that is what $1/\sqrt{N}$ means, and you are not expected to out-render it. **Your scene should be yours, not this one.**

Two routes. Pick one, and say which in `NOTES.md`.

	The sky	What it is like to work with
Route A a sealed room, Fig. 2	`SKY_BOTTOM` and `SKY_TOP` both black	Total control, and total responsibility: if you put no lamp in, you get a black rectangle. Shows bounced light most strongly, because there is nothing else. Slower to converge.
Route B under a sky, Fig. 3	a dimmed sky, plus **at least one** `Emissive` surface	The sky is a dome lamp wrapped around everything, so nothing is ever fully black and the picture converges faster. Easier to make pretty, harder to make *about* the bounce.

Both are worth full marks. Route B must not use the lectures' bright daylight sky unchanged — at `Vec3(1,1,1)` it does all the work and your `Emissive` surface becomes decoration. Dim it to something like a third of that.

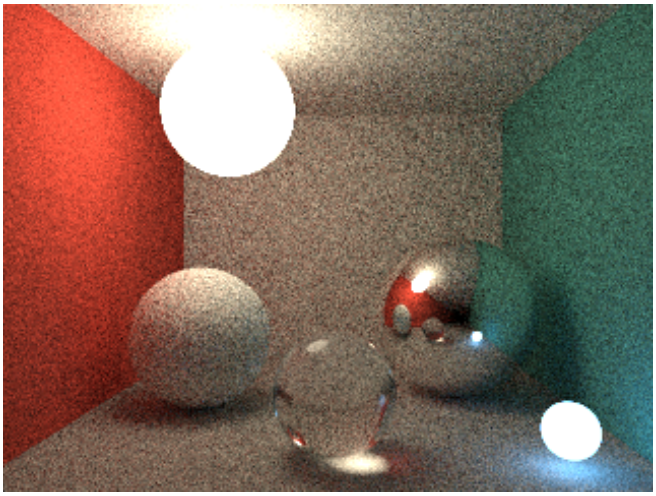


Figure 3: The exemplar's room, unchanged, with Route B's sky switched on at about a third of the lectures' daylight. Nothing is fully black any more, because every ray that escapes brings a little sky home through every bounce on the way; the picture is softer, cooler and converges faster, and the lamp has correspondingly less of the work to do. Compare Figure 2, which is the same room sealed.

Workflow — follow this order

1. Compose at 200×150 with `SAMPLES_PER_PIXEL = 8`. It will be a blizzard. **Ignore that completely** — you are placing things, not rendering.
2. Get the **layout** right, then the **lamps**, then the **camera**. In that order, and do not move on until each is settled.
3. Check that the lighting reads at `SAMPLES_PER_PIXEL = 64` before committing to anything final.
4. *Then* raise resolution and samples together, start it, and go and have dinner.

The one number that governs your evening: cost is linear in pixels, in samples and in depth. 400×300 at 256 samples is **sixteen times** the work of the same image at 16.

Four things worth doing

1. Put a saturated colour next to a white surface. It is the cheapest way to make bounced light unmistakable, and it is the effect no previous assignment could produce at all. A red wall and a white ball, a hand's width apart, with nothing between them.

2. Choose your lamp's radius deliberately. It is the softness of every shadow in the picture, and it is now a compositional decision rather than a parameter to tune — see Figure 4.

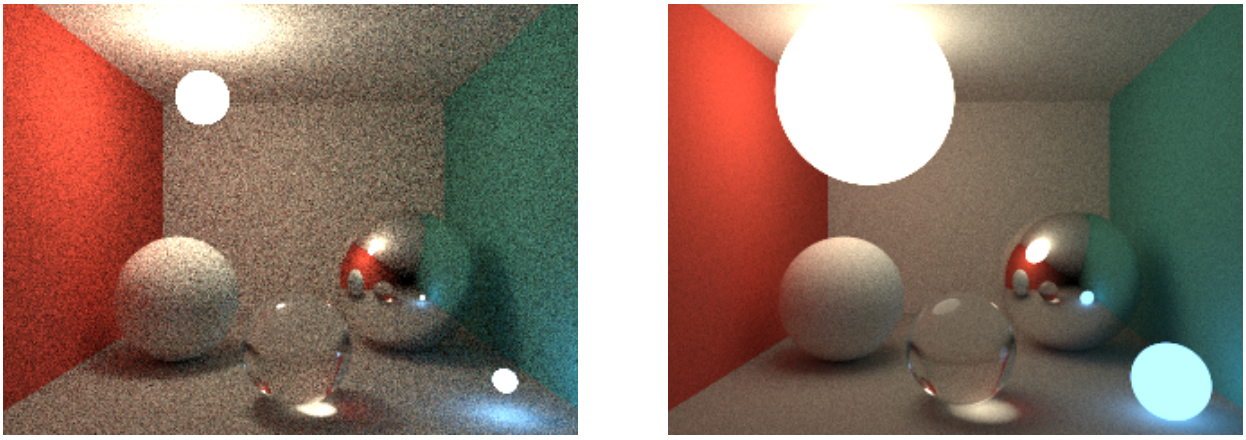


Figure 4: The same room, the same camera and the same *total* light: the left lamp has under a third of the radius of the right one and about eleven times its radiance, so the two deliver the same total. **Left:** a small lamp — a narrow penumbra, crisp shadows, high contrast, and visibly more noise, because a point's random rays find a small target far less often and bring back much more when they do. **Right:** a large lamp — a wide soft band at every shadow edge, gentle modelling, and a picture that converges faster at the same sample count. Both are 512 samples per pixel. Neither is better; they are different photographs, and the choice is yours.

3. Let something be lit only indirectly. A surface that faces away from every lamp, lit purely by what bounced onto it, is the single most convincing thing in a path-traced image — and it is the thing your A6 renderer could not draw at all. Figure 5 is the fastest way to check whether your own picture has any.

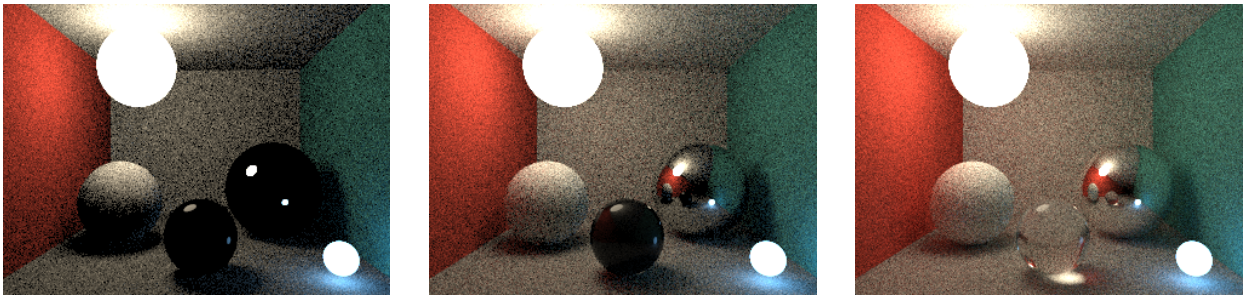


Figure 5: Lecture 19's "bounce 0, bounce 1, every bounce", rendered by your own code — just set `MAX_DEPTH`. **Left, `MAX_DEPTH = 1`:** only paths that hit a lamp on their first bounce survive. This is roughly what A6 drew, minus the ambient fudge: direct light, and black everywhere else. **Middle, 2:** one bounce of indirect light, and the room appears. **Right, 10:** every bounce. Note how much of the final picture is *not* in the left one. Render this triple of your own scene while you work — it is the fastest way to see whether your picture is actually about the bounce.

4. Keep your albedos near 0.8, and your floor darker than your walls. Each bounce multiplies by the albedo, so dark rooms swallow light and go slow and murky. A dark floor under bright walls is what

keeps your spheres from dissolving into the ground.

Five traps worth naming

1. The black render. Almost always one of three things: no `Emissive` surface in the scene at all; a lamp with a radiance below 1; or `MAX_DEPTH` left at 0 or 1 in a sealed room. Check in that order.

2. The muddy render. `to_display` is not being called, or its square root is missing. Do not compensate by turning the lamp up — see Figure 1.

3. Turning the lamp up to fix the noise. Noise is not brightness. It comes from too few samples and it falls as $1/\sqrt{N}$ — the only cures are more samples, a bigger lamp, or brighter albedos so that paths carry more light home.

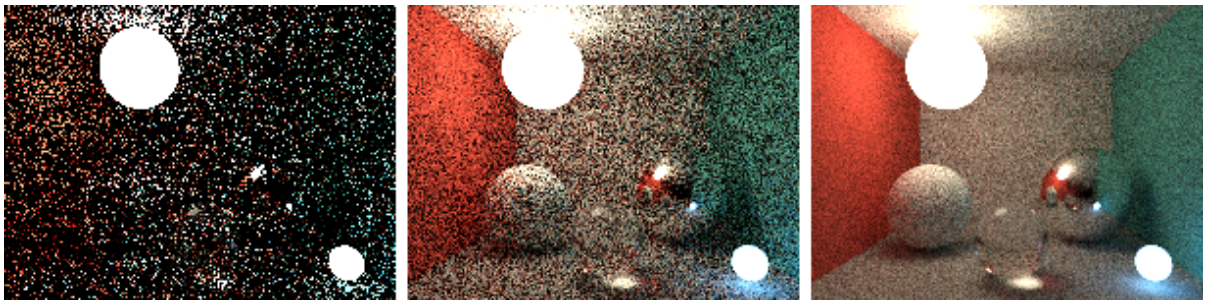


Figure 6: `convergence.ppm`, exactly as `pypy3 strip.py` produces it, on the exemplar scene. **1, 16 and 128 samples per pixel**, left to right, for one, six and forty-five seconds of work. At 1 the estimate is almost pure error — most pixels fired a single path that found nothing and came home black. At 16 the room is legible and the composition can be judged, which is where you should be doing all of your work. At 128 it is a picture. Each step is eight times the samples and therefore a bit under three times less grain: that is $1/\sqrt{N}$, and it is why the last clean image always costs more than you expect. **You submit this strip for your own scene.**

4. A render you cannot afford. Decide your final resolution and sample count *before* the last day, time a small version, and multiply. A render you started at 11pm and could not finish is worth zero, and no extension is given for one.

5. Leaving `AmbientLight` in. It is still in `tracer.py` and nothing calls it, so you will not get an error — you will get a scene that quietly ignores a line you wrote. Delete the light list from your `scene.py` altogether.

B1 · Compliance Checklist · 30 pts

Objective, and checked by eye in under a minute each. Do not lose these.

Requirement	Pts
`scene.ppm` submitted, at least 400 × 300, at least 128 samples per pixel	5
Every light is an `Emissive` surface. No `PointLight`, no `AmbientLight`, no light list	5
At least one **saturated colour beside a neutral surface**, with the bleeding visible in the image	5
At least one surface **lit only indirectly** — no lamp in its line of sight	5
`convergence.ppm`, the output of `pypy3 strip.py` on your own scene, unedited — see Figure 6	4
`NOTES.md`: your route (A or B), and **four to six sentences** naming where the bounced light is in *your* image and which surface put it there	4
Headers filled in — name, roll number, AI disclosure — in all three edited files	2
`python3 scene.py` runs from a clean copy of the starter with your three files dropped in	0

That last row is zero points and is not a formality: if it does not run, **none** of Part B can be marked.

B2 · Craft Rubric · 40 pts

Five criteria, eight points each. Marked against what the image actually shows, not against what `scene.py` intended.

Criterion	What full marks looks like	Pts
Composition	The camera is placed, not defaulted. Objects read as a deliberate arrangement with a subject and a background; nothing important is clipped, crowded into a corner, or hidden behind something else.	8
The lamp as a choice	Size, position and colour are all doing work and you can say why. A big soft key, or a small hard one, chosen — not the starter's sphere moved slightly.	8
Bounced light as a subject	The indirect light is not merely present, it is *used*: a coloured surface placed to tint a neutral one, a region lit only by bounce, a corner whose darkening you composed around.	8
Material contrast	Matte against mirror, or against glass, arranged so that each explains the other. A scene of five matte spheres earns few of these.	8
Finish	Noise under control for the resolution submitted; exposure such that the lamp is the brightest thing and the shadows are not crushed; albedos chosen rather than left at defaults.	8

A note on effort and hardware. The rubric asks for noise *under control*, not noise eliminated. The exemplar has visible grain. 400 × 300 at 256 samples with a reasonably sized lamp is a full-marks render, and it takes a few minutes under PyPy on any machine in the lab. Nobody is rewarded for owning a faster computer, and a scene whose composition is good at 128 samples will score above one whose composition is poor at 2000.

Submission

Submit a single ZIP named `<your roll number>_a7.zip`, containing **exactly these six files, no folder inside**:

`material.py`	your `Emissive`
`tracer.py`	your `trace` and `to_display`
`scene.py`	your scene
`scene.ppm`	your render, at least 400×300
`convergence.ppm`	the output of `pypy3 strip.py`, unedited
`NOTES.md`	route, and four to six sentences

Do **not** include `vec3.py`, `canvas.py`, `painter.py`, `vec2.py`, `check.py`, `expected_a7.py`, `strip.py` or `__pycache__`. We have those.

Before you zip: unpack a fresh copy of the starter into an empty folder, drop your three edited files in, and run `python3 check.py` and then `python3 scene.py`. If both work there, they will work here.

`scene.py` **must keep these names**, because `strip.py` and the grader read them:

```
build_scene()                              returns a Group
CAMERA_POSITION, CAMERA_TARGET, CAMERA_FOV the camera
SKY_BOTTOM, SKY_TOP                        both black for route A
MAX_DEPTH                                  how many bounces
```

`NOTES.md` is four to six sentences and nothing more. Say which route you took, then point at your own picture: where is the bounced light, and which surface put it there? One sentence of the form "the left side of the tall ball is pink because the red wall at $x = -3$ is throwing light onto it" is worth more than a paragraph about Monte Carlo. If you did the bonus, add one sentence about what you saw.

Appendix A: The A7 Cheat Sheet

Thing	**What to remember**
A lamp	an `Emissive` surface in `scene`, not an entry in `lights`. Radiance above 1 — 8 is a reasonable start.
A matte hit	black at depth 0; otherwise `albedo * trace(Ray(x, n + random_unit_vector()), depth - 1)`.
The π	cancelled, against $p = \cos\theta/\pi$. Do not put it back.
The estimator	$f(\omega)/p(\omega)$, averaged over N picks. Error falls as $1/\sqrt{N}$.
Soft shadows	not written. They are what a lamp with a radius does.
Ambient light	deleted. The corners go dark by themselves now.
A Blinn-Phong highlight	gone. Use `Reflective` if you want a bright spot.
`to_display`	clamp into $[0, 1]$, then `math.sqrt`. Average the samples *first*.
A black picture	no `Emissive`; or radiance below 1; or `MAX_DEPTH` too small for a sealed room.
A muddy picture	the square root. Always the square root.
Four times the samples	half the noise. Not a quarter.
`random_unit_vector()`	exactly once per matte hit, after the depth test.

Appendix B: Debugging A7

Everything is black. In order: is there an `Emissive` surface in the `Group`? Is its radiance above 1? Is `MAX_DEPTH` at least 4? Is `SKY_BOTTOM` black with no lamp to replace it? Set the sky to `Vec3(1,1,1)` temporarily — if the picture appears, your lamp is the problem and not your `trace`.

Everything is white in the middle and black at the edges. The lamp is far too bright, almost certainly because the square root is missing and you compensated. Put `to_display` right and take the lamp back down to 8.

The picture is grey mush with no shadows. `AmbientLight` is probably still in your scene *and* you have added a branch that calls it. Nothing in A7 calls `illuminate`.

`ZeroDivisionError` in `Sphere.hit`. The degenerate-direction guard in T2 is missing. A random unit vector came back as nearly $-\mathbf{n}$, `d` came out as nearly zero, and a ray with no direction has no quadratic to solve.

`RecursionError`. The depth test in the matte branch is missing or is testing the wrong thing. Every matte branch must be able to stop.

The render takes forever. You are on `python3`. Use `pypy3`. If you already are: your albedos may be near 1, so paths never die out — or `MAX_DEPTH` is 50 when 8 would do.

Only the bounce cases fail, and my picture looks fine. Count your calls to `random_unit_vector()`. There must be exactly one per matte hit, and it must come after the depth test. A second call anywhere — in a retry loop, in a guard, in a debug line — shifts every random number that follows.

The mirror or the glass stopped working. You edited a branch you were given. `check.py`'s two unscored `Given, unchanged` cases exist to tell you this; diff against the starter.

My shadows have a hard edge. Your lamp is very small, or very far away. Both are legitimate choices; if you did not intend it, grow the radius.

There is a bright speckle of single white pixels. A path happened to hit the lamp directly after a bounce and came home carrying a radiance of 8. It is real, it is correct, and it averages out — that is *fireflies*, and more samples is the only honest cure.

Appendix C: A7 in Thirteen Lines

Everything you write this week, with the docstrings removed:

```python
# material.py
class Emissive:
    def __init__(self, color):
        self.color = color

# tracer.py, inside trace()
if isinstance(material, Emissive):
    return material.color

elif isinstance(material, (Lambertian, BlinnPhong)):
    if depth == 0:
        return Vec3(0, 0, 0)
    x = ray.evaluate(hit.t)
    n = hit.normal
    d = n + random_unit_vector()
    if d.length_squared() < 1e-12:
        d = n
    return material.albedo * trace(Ray(x, d), depth - 1)
```

```python
# tracer.py
def to_display(color):
    def channel(v):
        return math.sqrt(max(0.0, min(1.0, v)))
    return Vec3(channel(color.x), channel(color.y), channel(color.z))
```

That is the whole of Part A. Fifty years of rendering research sits behind those thirteen lines, and the other seventy points of this assignment are about what you can see once they are written.