Assignment 4: Rays, Light and Shadow

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

Deadline: Friday 2nd October 2026, **11:59 pm**

Introduction

Everything you have drawn so far, you drew because you decided to. This week the picture decides. You will fire **rays** into a scene, find what they hit, and draw only what the light can reach — and shadows will appear on their own, because a shadow is not a shape you draw. It is the part your rays could not get to.

Objectives

- Writing a ray as $p(t) = \mathbf{o} + t\,\mathbf{d}$ and reading t as a distance
- Solving ray–segment and ray–circle intersection exactly
- Finding the nearest hit among many objects — the core of all rendering
- Using one ray as a visibility test, and guarding it against self-shadowing
- Turning visibility into brightness: falloff, facing, and occlusion
- Softening a hard edge by averaging many samples — the idea behind everything later in this course

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 A4 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 `raycast.py`, `light.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**:

`raycast.py`	(edit this) your ray library — **T1–T8 go here**
`light.py`	(edit this) your illumination code — **T9–T12 go here**
`scene.py`	(edit this) your lit scene — **all Part B work goes here**
`vec2.py`	*do not edit* — from A2
`curves.py`	*do not edit* — from A2
`mat3.py`	*do not edit* — from A3
`shapes.py`	*do not edit* — from A3, plus `polygon_segments`
`canvas.py`	*do not edit* — the `Canvas`/`Layer` classes
`painter.py`	*do not edit* — from A1
`encode.py`	*do not edit* — turns your frames into a video
`check.py`	*do not edit* — the autograder; run it as often as you like
`expected_a4.py`	*do not edit* — reference data used by `check.py`

`mat3.py` and `shapes.py` are the reference solutions to A3. Read them and compare against what you submitted, quietly, for your own benefit. You are not graded on them and you must not edit them.

As in A1–A3, there is **nothing to install**: A4 is standard-library Python throughout — no `pip`, no virtual environment, no third-party packages. Any Python 3.9+ interpreter works, **PyPy** included, and this week PyPy earns its keep: Part B fires hundreds of thousands of rays and fills millions of pixels. On a fast desktop the exemplar's 120 frames take about a minute under CPython and about two seconds under PyPy. The one external tool is `ffmpeg`, exactly as in A2 and A3:

```
python3 encode.py --check          # confirm ffmpeg is still reachable
```

B) The Ray Model

Three small classes, all **given**, carry everything in Part A:

- A `Ray` is an origin and a direction. The constructor **normalizes** the direction, which is the decision the rest of the assignment rests on: once $|\mathbf{d}| = 1$, the parameter t is a *distance in pixels*. Comparing two hits is then comparing two distances.
- A `Segment` is a wall: two endpoints. It is the only occluder shape you need, because the edges of any polygon are segments — and `shapes.polygon_segments` turns a transformed A3 shape into exactly that list.
- A `Hit` is what a ray found: `.t` (how far), `.point` (where), and `.segment` (on what). **A miss is `None`**, never a `Hit` with a nonsense t, so `if hit is not None:` is the only test you ever write.

Everything else — every function that actually intersects, tests or shades — is yours.

The A4 API at a glance. The left column is **given** and already works; the right column is **yours**.

Given (plumbing)	Yours (T1–T12)
`Ray(origin, direction)`	`ray.at(t)` `raycast.ray_at(o, d, t)`
`Segment(a, b)`	`raycast.ray_toward(origin, target)`
`seg.direction()` `seg.normal()`	`raycast.ray_segment(ray, seg)`
`Hit(t, point, segment)`	`raycast.ray_circle(ray, c, r)`
`shapes.polygon_segments(m, s)`	`raycast.nearest_hit(ray, segs, max_t)`
`EPSILON` `SHADOW_BIAS`	`raycast.march(ray, t_max, steps)`
	`raycast.is_visible(p, light, segs)`
	`raycast.cast_fan(o, segs, count)`
	`raycast.corner_directions(o, segs)`
	`light.attenuation` `light.lambert`
	`light.brightness(...)`
	`light.visibility_polygon(...)`
	`light.draw_light_polygon(...)`
	`light.shade_segments(...)`
	`light.area_light_samples(...)`
	`light.soft_shadow_factor(...)`

Part A — Rays and Light (70 pts)

Implement T1–T8 in `raycast.py` and T9–T12 in `light.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 what your functions draw, exactly), as in A2 and A3. **Grading runs additional hidden cases**: passing the checker is necessary, not sufficient — hardcoding its expected values earns a 0 for the task.

T1–T8 below are written in `raycast.py`. Open that file now — the stubs are in the same order as the tasks here, and each docstring repeats the contract you are reading. Nothing in this file draws anything: it is pure geometry, and it imports nothing beyond `math` and `vec2`.

T1 · the ray equation · 4 pts

The whole assignment in one line:

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

Write `Ray.at(t)` (the method, on a normalized direction) and `ray_at(origin, direction, t)` (the free function, on a raw direction). Then `ray_toward(origin, target)`, which builds the ray from one point aimed at another — one line, since the constructor normalizes for you.

Why normalize? Because it makes t mean *pixels*. With $|\mathbf{d}| = 1$, "the nearest hit" and "the shortest t" are the same statement, and a light's falloff can use t directly. Leave $\mathbf{d}$ unnormalized and t means "in units of however long $\mathbf{d}$ happened to be", which silently breaks every distance comparison you are about to write.

T2 · `ray_segment` · 9 pts

The workhorse of the assignment. Nearly everything below calls it, so get it right before moving on.

Set the ray equal to the segment's own parameterization and solve for both parameters at once:

$$\mathbf{o} + t\,\mathbf{d} \;=\; \mathbf{a} + u\,(\mathbf{b} - \mathbf{a})$$

Writing $\mathbf{v}_1 = \mathbf{o} - \mathbf{a}$, $\mathbf{v}_2 = \mathbf{b} - \mathbf{a}$ and $\mathbf{v}_3 = \text{perp}(\mathbf{d})$, the solution is

$$t = \frac{\mathbf{v}_2 \times \mathbf{v}_1}{\mathbf{v}_2 \cdot \mathbf{v}_3} \qquad\qquad u = \frac{\mathbf{v}_1 \cdot \mathbf{v}_3}{\mathbf{v}_2 \cdot \mathbf{v}_3}$$

where $\times$ is A2's scalar cross product. This is Lecture 12's Cramer's-rule solution in vector form: each determinant there is one of these cross or dot products with its sign flipped, and the flips cancel in both fractions.

Required behaviour:

- If $|\mathbf{v}_2 \cdot \mathbf{v}_3| <$ `EPSILON` the ray is **parallel** to the segment. Return `None`; do not divide.
- A hit counts only when **both** $t \geq 0$ and $0 \leq u \leq 1$. Otherwise return `None`.
- Otherwise return `Hit(t, ray.at(t), seg)`.

Those two conditions are the entire difference between a ray hitting a wall and two infinite lines crossing, and **dropping either one is the most common bug in this assignment**. Forget $t \geq 0$ and objects *behind* your light cast shadows; forget $u \leq 1$ and every wall extends forever in both directions.

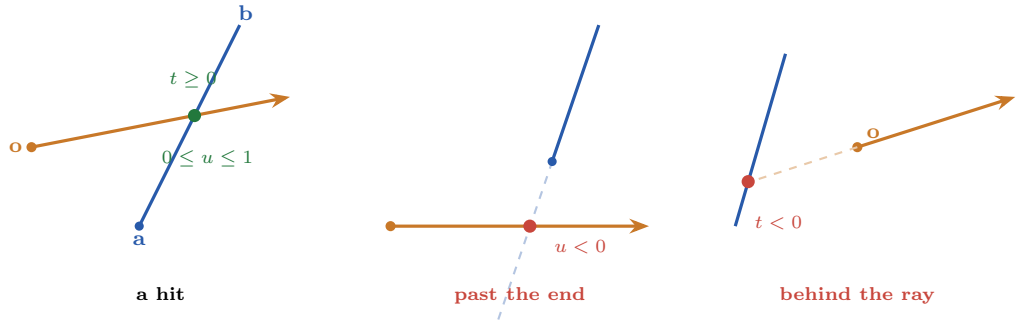

Figure 1: Both tests are needed. u keeps the hit *on the segment*; t keeps it *in front of* the ray.

T3 · `ray_circle` · 6 pts

Lecture 12's quadratic. A hit is a point of the ray at distance r from the centre $\mathbf{c}$: set $\|\mathbf{o} + t\,\mathbf{d} - \mathbf{c}\| = r$, square, expand, and use $\mathbf{d} \cdot \mathbf{d} = 1$:

$$t^2 + \beta\,t + \gamma = 0, \qquad \beta = 2\,\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}), \qquad \gamma = \|\mathbf{o} - \mathbf{c}\|^2 - r^2$$

with the two roots

$$t_0 = \frac{-\beta - \sqrt{\beta^2 - 4\gamma}}{2} \;\leq\; t_1 = \frac{-\beta + \sqrt{\beta^2 - 4\gamma}}{2}.$$

Required behaviour:

- $\beta^2 - 4\gamma < 0$: no roots, the ray misses. Return `None`.
- Otherwise take the **smaller non-negative root** — the near side is what the ray sees first. If the origin is *inside* the circle the near root is negative and the far one is correct; if *both* are negative the circle is entirely behind the ray, so return `None`.

- The `Hit`'s `segment` is `None`.

Notes (not graded). Lecture 14's ray–sphere intersection is this algebra again — *literally* the same. The 3D version changes only how many components the dot products have. Everything you debug here you will not have to debug again there.

T4 · `nearest_hit` · 6 pts

Test the ray against every segment and keep the hit with the **smallest** t — not the first one you happen to find. **Reordering the segment list must not change the answer.** Hits farther than `max_t` are ignored, which is what lets a shadow ray stop at the light instead of running to infinity.

This little loop is the whole of rendering in miniature, and it is also the reason Lectures 19 and 20 exist: it costs $O(n)$ per ray, which is fine for the dozens of segments here and hopeless for the millions in a real model. A bounding volume hierarchy replaces exactly this loop.

T5 · `march` · 4 pts

Return `steps+1` points evenly spaced along the ray from $t = 0$ to $t =$ `t_max`, inclusive of both ends — the same convention as A2's `sample_parametric`, where `steps` counts the *gaps*. `steps < 1` returns just the origin.

Marching is the dumb, robust way to explore a ray: it is how you draw a visible beam, sample fog along its length, and — in Lecture 33 — raymarch a signed distance field on the GPU. Its weakness is worth knowing too: it can step straight *over* a thin occluder lying between two samples. Exact intersection (T2) never does.

T6 · `is_visible` · 7 pts

The shadow test, and it is one ray. Fire from the point toward the light and ask whether anything blocks it *before* the light.

Two subtleties, and both cause classic bugs:

1. **Self-shadowing.** A point sitting exactly on a wall is at $t = 0$ along its own shadow ray, and floating-point error will happily report that it hits itself. Every surface then shades itself and the image speckles black — the artifact called **shadow acne**. Start the ray `SHADOW_BIAS` pixels along its own direction, and shorten the max distance by the same amount.
2. **The light's distance is the limit,** not infinity. An occluder standing *behind* the light shades nothing.

Also guard the degenerate case: if the point *is* the light (distance below `EPSILON`), return `True`.

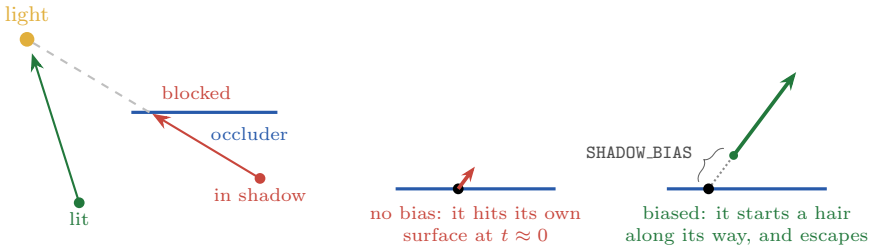

Figure 2: One ray, from the point toward the light, decides lit or shadowed. The bias, drawn hugely enlarged on the right, is what stops a surface from shadowing itself.

Notes (not graded). You will write this exact function again in Lecture 17, in 3D, for shadow rays. It is the same seven lines.

T7 · `cast_fan` · 4 pts

Fire `count` rays evenly around a full circle from `origin`, ray k at angle $2\pi k/$`count`, and return one **point** per ray: where it hit, or where it ran out at `max_distance`. This is A2's `circle_point` logic reused — a fan is a circle sampled by *direction* rather than by position. `count <= 0` returns `[]`.

Join those points in order and you have outlined the region the light can see. Do it, and you will notice the flaw that T8 exists to fix.

T8 · `corner_directions` · 7 pts

A uniform fan is blurry for a cheap reason: **a shadow's edge almost never lands exactly on one of your evenly spaced rays**, so the silhouette gets quantized to the fan's angular resolution. Adding more rays makes it less wrong and never right.

The fix is to stop sampling uniformly and **aim at the places where the answer can change** — and for straight-edged occluders those are exactly the segment **endpoints**.

For every endpoint of every segment, emit **three** angles: the angle to that corner, and that angle $\pm$ `nudge`. The middle ray stops at the corner. Of its two neighbours, the one on the open side slips past the corner and flies on to whatever is behind, while the other lands on the wall just beside it; which side is open depends on the geometry, so fire both. **That triple is what makes the edge sharp:** the polygon gets one point on the corner and the next one straight behind it.

Required behaviour:

- Use `math.atan2(d.y, d.x)`, and wrap into $[0, 2\pi)$ with `% (2 * math.pi)`.
- Return the list **sorted** — T10 fills a fan in this order, and unsorted angles make it criss-cross itself.
- Skip any endpoint coincident with `origin` (guard with `EPSILON`): a zero vector has no direction.
- Duplicates are fine; they cost one wasted ray and break nothing.

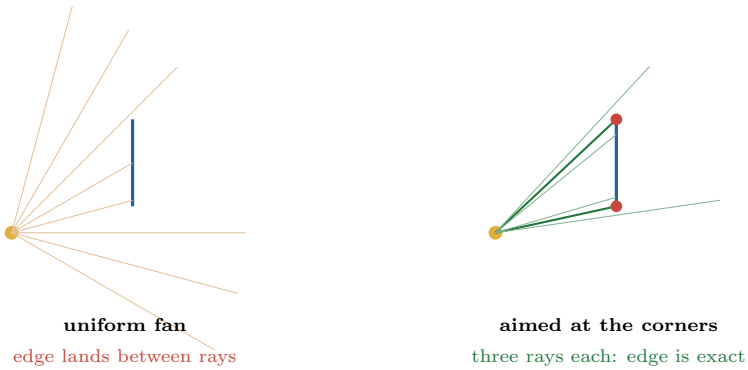

Figure 3: Sample where the function *changes*, not where it is convenient. The corners are the only places visibility can flip.

`raycast.py` **is done — T9–T12 below are written in** `light.py`**.** That file turns "what is in the way" into a picture: how bright a spot is, what region a lamp can see, and what a shadow's edge looks like when the lamp is not a single point.

T9 · `brightness` · 6 pts

Three small functions, and the third is the product of the first two with the shadow test.

1. `attenuation(distance, radius)` — how much power survives the trip:

$$f = \text{clamp}\left(1 - \frac{\text{distance}}{\text{radius}},\, 0,\, 1\right) \qquad \text{return } f^2$$

Squaring makes the light drop fastest near the source, as real light does, and then ease gently into zero at the rim, so the edge of its reach leaves no visible line; a straight $1 - d/R$ would stop with a crease. (Real light falls off as $1/d^2$, which never quite reaches zero and so is awkward for a picture with edges. This is a common graphics compromise, and the idea behind a game engine's "light radius" slider.) A `radius <= 0` returns 0.

2. `lambert(point, normal, light_pos)` — how squarely the surface faces the light:

$$\max\left(0,\ \hat{\mathbf{n}} \cdot \hat{\mathbf{l}}\right)$$

Normalize both. A surface facing the light head-on gets 1, edge-on gets 0, and facing away gets a negative dot product that must **clamp** to 0.

3. `brightness(...)` — multiply all three factors:

$$\texttt{visibility} \times \texttt{falloff} \times \texttt{facing}$$

Multiplying is right because any one of them being zero means no light arrives: in shadow, out of range, and facing away are three different ways to be dark.

Test the cheap factors first and return early. If the point is out of range or facing away, you never pay for the shadow ray — and the shadow ray is by far the most expensive part. That single early-out is worth several times the render time on a scene with a few lights.

Notes (not graded). Lambert's cosine law is the single most-used formula in shading, and Lecture 16 is this exact function in 3D. The dot product does not care how many dimensions you hand it.

T10 · the light polygon · 6 pts

1. `visibility_polygon(origin, segments, max_distance)` — take T8's angles, fire one ray along each, and collect where each stops. Because T8 sorted its angles, the points come back in angle order.

Required: with *no* segments there are no corners to aim at, so fall back to a uniform fan (`cast_fan` with 64 rays). Without that, an empty scene produces an empty polygon and your light silently vanishes.

The one precondition: every ray must end on a wall. Between two neighbouring rays the fan draws a straight edge. That edge is exact when both rays stop on the same wall, and wrong when they fly off into open space: it cuts straight across the dark, and if two neighbouring corners are more than half a turn apart, the closing triangle covers the wrong side of the light altogether. The checker's rooms are closed. In Part B, keep yours closed too (trap 2).

2. `draw_light_polygon(img, origin, points, color)` — fill it as a **triangle fan hubbed at the light**: triangles $(\mathbf{o}, p_0, p_1), (\mathbf{o}, p_1, p_2), \ldots$, closing p_{n-1} back to p_0. Fewer than two points draws nothing.

Inside a closed room, a fan from the light is **always** valid, however concave the lit region looks — every point of that region is by definition visible from the light, which is precisely what a fan needs. Contrast A3's `fill_shape`, where a fan from an arbitrary vertex spills outside a concave polygon. Same technique; here you can say exactly why it is safe.

T11 · `shade_segments` · 6 pts

A wall's brightness *varies along its length*, so one flat color will not do. Chop each segment into `samples` pieces; for each piece take its **midpoint**, sum `brightness()` over every light, scale the colors, and stroke the piece with `curves.thick_line`.

`lights` is a list of `(position, radius, color)` tuples. Use each light's *own* color, so two differently coloured lamps mix on the wall the way they should. The reference mix, which the pixel case checks:

```
channel = base_color[i] * 0.15 + accumulated_light[i] * 0.85
```

truncated with `int()` and capped at 255.

Treat every segment as two-sided. A wall in a 2D scene should light from whichever side the lamp is on, but `Segment.normal()` returns one of the two perpendiculars based on the order of `a` and `b` — an accident of how the geometry was authored. Flip it toward the light before shading:

```
n = normal if normal.dot(light_pos - mid) > 0 else normal * -1
```

Skip this and half of every closed shape comes out unlit for no reason visible in the picture, which is a genuinely nasty bug to find from the output alone.

T12 · soft shadows · 5 pts

A point light gives one hard edge, because a point is either visible or it is not. A real lamp has **size**, and a spot near the shadow's border can see part of the lamp and not the rest — which is what a penumbra actually is.

1. `area_light_samples(center, axis, count)` — the light runs from **center** − **axis** to **center** + **axis** (Lecture 12's $\mathbf{p}_0$ and $\mathbf{p}_1$). Return `count` points at the *centre* of each share of the length:

$$\text{sample}_k = \textbf{center} + \textbf{axis}\left(\frac{2(k+0.5)}{\texttt{count}} - 1\right)$$

Check it at the ends: `count = 2` gives the quarter and three-quarter points, **not** the endpoints. Sampling the endpoints double-counts the extremes and puts a sample exactly on a corner of the light, where a grazing occluder makes the answer jumpy.

2. `soft_shadow_factor(...)` — run `is_visible` once per sample and return the **fraction** that succeeded: 1.0 fully lit, 0.0 the **umbra**, anything between the **penumbra**. `count <= 0` returns 1.0.

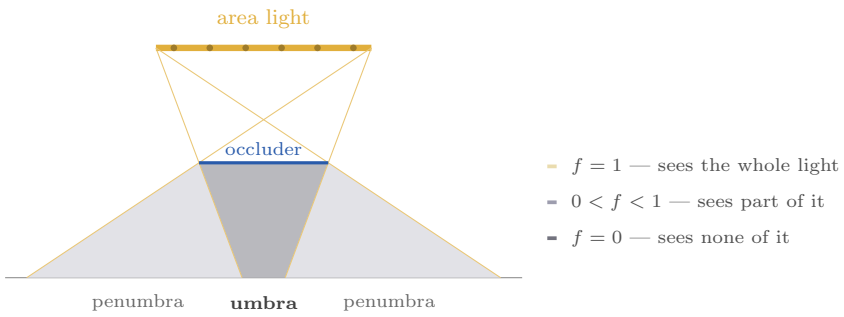

Figure 4: `soft_shadow_factor` is the fraction of the light a point can see. The penumbra is where that fraction is strictly between 0 and 1.

Notes (not graded). That fraction is the single most transferable idea in this assignment. **Replacing one binary yes/no with the average of many samples** is how graphics softens hard edges everywhere: soft shadows here, antialiasing in Lecture 15, depth of field in Lecture 21, and the entirety of Monte Carlo path tracing in Lecture 27. The cost is equally general — `count` times the rays, for an answer that is still an estimate.

Part B — Your Lit Scene (30 pts)

Using your toolkit, design and render an animated **lit** scene in `scene.py`: a place made of occluders, where what you can see is decided by what the light can reach. Render it as a sequence of frames and encode it to `scene.mp4`.

What you build is up to you, and we mean that. It does not have to be a room, it does not have to be realistic, and it does not have to obey physics. One candle or a whole light show; cosy, eerie, funny, abstract or flatly impossible — all of it is welcome. The B1 checklist below is the only hard constraint. Subject, style, palette, canvas shape, length and tone are yours, and B2 rewards ambition. If you have an idea that seems too strange for a graphics assignment, that is the one to make.

The rule that defines this assignment: you are not drawing shadows. You are drawing *light*, and the shadows are what is left over. If you find yourself drawing a dark polygon by hand to fake a shadow, stop — that polygon should be the *absence* of a light polygon.

The test of whether you got it right is one line long. **Move the light and every shadow in your scene must move correctly**, with no other edits. With several lights, that holds for each one. B1 requires exactly that, and the grader will try it. Two things this does *not* rule out:

- **Drawing the occluders themselves.** A block filled solid, a silhouette, a lamp's housing: that is the object, not its shadow. Only the dark region *behind* an object must come from the rays.
- **Decoration that casts no shadow.** A sky, stars, a floor pattern, titles: paint them however you like, ideally once, in the background. Only something that should cast a shadow has to be an occluder the rays can hit.

Where you could take it

The orbiting lamp is deliberately plain: one light, boxes, a loop. Rays can do far stranger things, and each row below leans on something this week's toolkit is uniquely good at. **None of it is required and none of it is a template** — it is here to show how big the space is. Borrow a technique, not a subject.

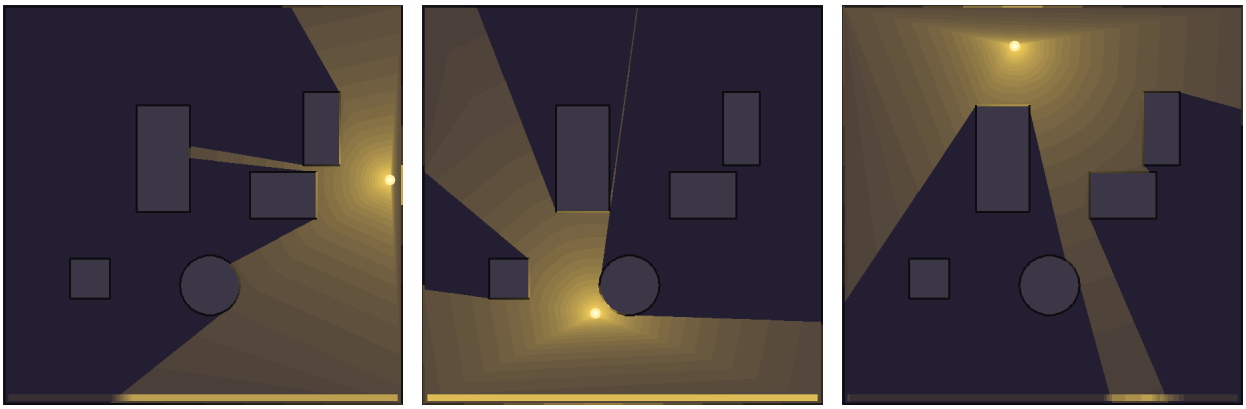

Figure 5: Three frames from the instructor exemplar, "The Orbiting Lamp" (frames 0, 40 and 80 of 120), built entirely on this assignment's own machinery. A lamp orbits a room of blocks and a round post, and every shadow wedge in the picture is simply the region the light polygon could not reach: nothing anywhere in the code draws a shadow. The falloff is the same polygon drawn at shrinking reaches, dim to bright, and the strip of floor along the bottom wall is lit by the lamp treated as a *bar*, so its shadow edges are soft where everything else's are hard. Your scene should be *yours*, not this one — and it need not be a room at all.

- **Coloured lights.** Give each lamp its own colour and its own light polygon on a `Layer`, and pass all of them to `shade_segments`. Wherever one lamp is blocked and another is not, the shadow takes the other lamp's colour: red, green and blue lamps throw cyan, magenta and yellow shadows. Stage lights, a disco, a police car at night.
- **Light shaped by geometry.** Three sides of a box around a lamp turn it into a beam; rotate the box and the beam sweeps. A lighthouse, a searchlight, a guard's cone of vision in a stealth game. The beam is just the light polygon — no cone maths anywhere.
- **Occluders from a scene graph.** A3's tree still works. `mat3.draw_tree` hands its `img` argument straight to each node's `draw`, so pass it a list and let each `draw` append `shapes.polygon_segments(world, shape)`: the tree now collects this frame's occluders instead of pixels. A windmill, a walking figure, clock hands sweeping shadows across a dial.
- **Slits and slats.** A row of short segments splits one lamp into fingers of light. Turn each slat about its own centre and the fingers open and close: venetian blinds, a fence, a cage, a picket gate.
- **Many occluders, seeded.** A seeded random scatter — a forest of trunks, a field of stones, an asteroid belt — or a maze grown by recursion. The raycaster does not care how many there are, but your clock does: every corner fires three rays and every ray tests every segment, so the cost grows with the *square* of the segment count.
- **The penumbra as the subject.** Move an occluder between a long lamp and a wall, and T12 does the rest: its shadow is crisp beside the wall and blurs as it rises toward the lamp. A hand approaching a wall, a moth circling a lantern.
- **A light that is carried.** The lamp's position can come from anywhere: a node's world matrix (`mat3.world_matrix(node).apply_point(Vec2(0, 0))`), a point on an A2 Bézier path, a particle. A lantern swinging on a rope, a firefly, headlights on a winding road. Leave the carrier's own body out of the occluder list, or the lamp sits inside it (trap 1 below).

Write whatever helpers you like in `scene.py` — your own shapes, Bézier outlines from A2's `curves`, A3's scene graph, a palette that shifts with `u`. Two practical limits: everything has to live in `scene.py` (it is the only scene file you submit), and it stays standard-library Python.

Workflow — follow this order

1. `python3 encode.py --check` — confirm ffmpeg still works.
2. Set `FRAMES = 1` and get a **single frame** right. Iterating on one frame is seconds; on 120 it is minutes.
3. **Turn on** `DEBUG_RAYS` and look at where your rays actually go before you trust the picture.
4. Keep `SOFT_SAMPLES` low (4) while building; raise it (16+) only for the final render.
5. Raise `FRAMES`, render, encode, watch it. Then adjust.

```
python3 scene.py        # writes frames/frame_0000.ppm, frame_0001.ppm, ...
python3 encode.py       # combines them into scene.mp4
```

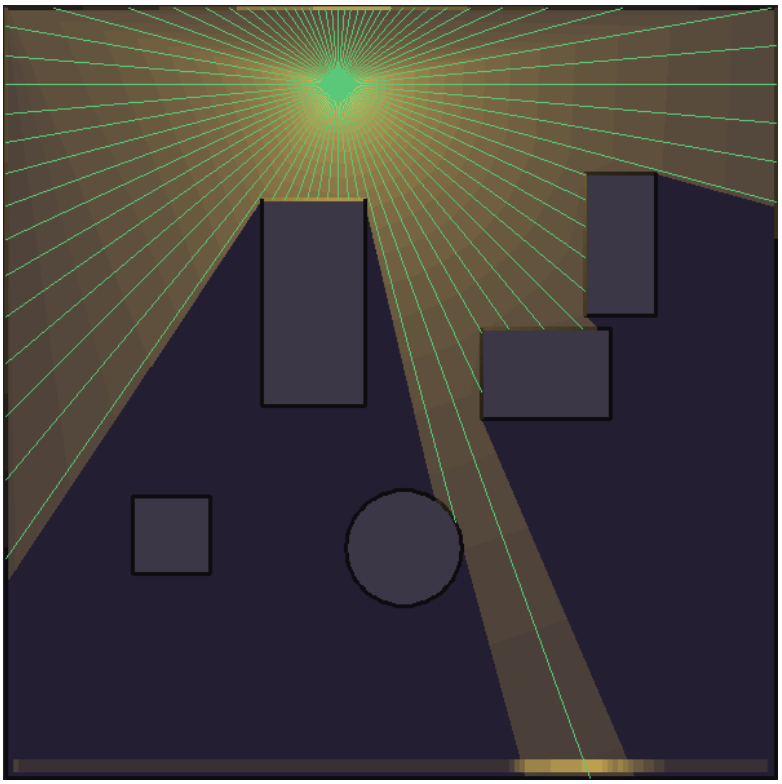

Figure 6: The exemplar with `DEBUG_RAYS = True`. Every ray from the lamp is drawn, and you can see at a glance where each one stops — on a block, on the post, on the far wall. A lighting bug is invisible in the final picture and obvious here. Turn this off before submitting.

Three traps worth naming

These cost people an evening every year, and `DEBUG_RAYS` shows the first two in seconds.

1. A light inside its own geometry lights nothing. If your lamp sits on the wall it is mounted on, every one of its rays is stopped at $t \approx 0$ by that wall, and the scene stays black. Put the lamp *outside* the occluders — this is T6's self-shadowing problem again, at scene scale.

2. An open scene breaks the light polygon. T10's fan joins neighbouring rays with straight edges, and that is exact only when both rays end on a wall. Leave the scene open and rays fly off into nothing: whole

shadows vanish, and a lamp whose corner rays leave a gap of more than half a turn lights the wrong side of itself. **Close the scene**, however open it is meant to look. Four segments just outside the canvas edge will do: they are never drawn, and every ray ends on one.

3. A soft shadow you cannot see scores nothing. The penumbra's width is set by where the occluder sits. Beside the surface its edge is nearly hard; as it moves toward the light the edge widens and fades, and an occluder narrower than the lamp eventually loses its umbra altogether. Halfway between light and surface is a good place to start: the penumbra there is about as wide as the lamp itself. And make sure the occluder actually stands between the light and the surface you are sampling.

If your render is slow

Rays are not free, and neither are pixels. On a fast desktop the starter's 90 frames take about 11 seconds under CPython and the exemplar's 120 about a minute — and most of the exemplar's minute goes on *filling* its light, not on casting rays. Four fixes, roughly in order of how much they buy you:

Use PyPy. A4 is arithmetic in a loop, which is exactly what a JIT is good at. Under PyPy the same two renders take under a second and about two seconds — the same code, one word of the command changed.

Cache what does not depend on `u`. Same rule as A2 and A3, now applied to geometry as well as pixels: if an occluder never moves, its segments do not need rebuilding every frame, and your background certainly does not need repainting. The starter shows the background; Appendix C shows the occluders.

```
BACKGROUND = make_background()      # painted ONCE, at import

def render_frame(i):
    img = BACKGROUND.copy()         # not a repaint - a copy
    ...
```

Count your light passes. Every `draw_light_polygon` call fills every pixel it covers, one Python call per pixel: a pass that lights a whole 640×400 room is a quarter of a million of them, and pasting a `Layer` visits every pixel again. Three passes cost three times one. Build with one; add the rest for the final render.

Turn `SOFT_SAMPLES` down while iterating. It multiplies the soft shadow's ray count directly. Build at 4, ship at 16.

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 `frames/frame_0000.ppm` …	2
At least 60 frames; canvas at least 480×320	1
`python3 encode.py` produces a playable `scene.mp4` from your frames	2
At least **8 occluder segments**, at least one of them **moving**	2
A light polygon drawn with your `visibility_polygon` (T10)	3
At least one surface lit with `brightness()` — falloff **and** facing	2
One **visible** soft shadow, from `soft_shadow_factor` (T12)	2
Moving **only** the light's position moves every shadow correctly	1
Any randomness is seeded — the same movie on every run	1
Total	**16**

Two of these are faked every year, so to be explicit. The **soft shadow** must be visible *as* a soft shadow: a penumbra several pixels wide that a marker can point at. Computing a factor and drawing a hard edge with it scores 0. And the **moving occluder** means its shadow moves because *it* moved — not because you moved the light.

The minimums are only minimums. A longer film, a bigger or oddly shaped canvas (square, tall, letterboxed), a dozen lights and hundreds of occluders are all fine; the only ceilings are the upload limit and your render time.

B2 · Craft Rubric · 14 pts

Criterion	0 — absent	1 — attempted	2 — competent	3 — excellent
Lighting design (worth 0/1/3/5)	One light, flat; shadows incidental	Light works, but placement is arbitrary and nothing is revealed by it	Light is placed to show the geometry off; falloff and occlusion both read	Light shapes the scene — hard and soft edges used deliberately, geometry shaped to shape the light
Motion quality	Nothing moves, or motion jitters	Movement happens but is uniformly linear and abrupt	Smooth, continuous motion; easing or varied pacing used deliberately	Motion has intent — the sweep pauses where it matters, movers arrive on cue, or a stylised rhythm that is plainly deliberate
Composition & design	Elements scattered; no framing	Some structure; foreground and background undifferentiated	Clear staging; the lit region is easy to read	Deliberate composition that holds up *throughout* the motion, not just on frame 0
Originality & ambition	Reproduces the starter room or the handout's orbiting lamp	Derivative with small changes	Clearly original; minor artifacts (popping, gaps, stray pixels)	Original and polished — or a genuinely bold idea whose rough edges are the price of its reach; the loop or ending is considered

Lighting design counts extra (it is what this assignment is about): its four columns are worth 0, 1, 3 and 5 points. The other three criteria score as labelled, 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 room, lightly recolored, caps *Originality* at 1.

Ambition is rewarded, not punished. Realism is optional and physics is optional; only intent is required. If you attempt something genuinely strange or hard and a few seams show, the marker weighs the reach, not only the finish: a safe idea polished to a shine and a wild idea with a few rough edges can both earn a 3 on *Originality & ambition*.

Two cheap upgrades. First, **draw the light polygon two or three times at different reaches**, with low alpha — attenuation only ever reaches your *surfaces*, never the air, so stacking a couple of ranges is the cheapest way to get falloff into the lit region itself. A short pass comes out as a polygon rather than a circle, because its outline bends only where a corner ray points; that faceting is part of the look, not a bug. Second, **shape your light with geometry**: a hood over a lamp, a doorway, a slot in a wall. It costs a few more segments and it is the difference between a lamp that looks placed and one that looks painted on.

Submission

Submit **one zip file** named `RollNo_A4.zip` (e.g. `231234567_A4.zip`) on the course website, containing exactly:

`raycast.py` `light.py` `scene.py` `scene.mp4`

Do **not** include any of the following: the `frames/` folder (it is hundreds of frames), `vec2.py`, `curves.py`, `mat3.py`, `shapes.py`, `canvas.py`, `painter.py`, `encode.py`, `check.py`, `expected_a4.py`, `__pycache__`, or the ffmpeg binary. 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.

If `scene.mp4` exceeds the upload limit, reduce your canvas size or frame count and re-encode — do not submit a partial video.

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 Ray Cheat Sheet

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

You want	Call	The idea
a point along a ray	`ray.at(t)`	$\mathbf{o} + t\,\mathbf{d}$; with $\lvert\mathbf{d}\rvert = 1$, t is pixels
does this ray hit that wall	`ray_segment(ray, seg)`	solve for t and u; need $t \geq 0$ *and* $0 \leq u \leq 1$
what does it hit first	`nearest_hit(ray, segs)`	smallest t wins; `max_t` clips
is this point lit	`is_visible(p, light, segs)`	one ray, started `SHADOW_BIAS` along, stopped at the light
how bright is it	`brightness(...)`	visibility $\times$ falloff $\times$ facing
what can the lamp see	`visibility_polygon(...)`	rays aimed at every corner, in angle order
how soft is the edge	`soft_shadow_factor(...)`	the fraction of the lamp that is visible

The single rule to carry away: a shadow is not drawn. It is the region no ray reached.

Appendix B: Debugging Rays

Lighting bugs are invisible in the output and obvious in the rays. When something is wrong, turn on `DEBUG_RAYS` and work down this list.

Symptom	Almost always
The whole scene is black	the light sits *inside* an occluder — every ray stops at $t \approx 0$
Half the scene goes dark, or whole shadows vanish	an open scene: rays fly off into nothing; close it with walls (T10, trap 2)
Everything speckles black	shadow acne: missing `SHADOW_BIAS` in `is_visible` (T6)
Objects behind the light cast shadows	`max_t` not passed to `nearest_hit` (T6)
Walls extend forever	the $u \leq 1$ test is missing in `ray_segment` (T2)
Shadows appear behind the ray	the $t \geq 0$ test is missing (T2)
The light polygon criss-crosses itself	angles not sorted (T8), or the fan is not hubbed at the light (T10)
Shadow edges are stair-stepped	using a uniform fan instead of corner directions (T7 vs T8)
Half of every shape is unlit	the normal is not flipped toward the light (T11)
The nearest object is not the one hit	keeping the first hit instead of the smallest t (T4)
Render takes minutes, not seconds	CPython instead of PyPy, too many light passes, `SOFT_SAMPLES` too high, or the background repainted per frame

Appendix C: A Lit Scene in Twenty Lines

The skeleton every Part B scene shares. Everything interesting is in the geometry and the light's position; the drawing never changes.

```
def build_occluders(u):              # u runs 0.0 -> 1.0
    segs = list(STATIC_OCCLUDERS)    # walls + fixed props, built ONCE

    # a moving occluder is just one more segment
    angle = 0.9 + 0.7 * math.sin(2.0 * math.pi * u)
    tip = HINGE + Vec2(math.cos(angle), math.sin(angle)) * 110.0
    segs.append(Segment(HINGE, tip))
    return segs

def render_frame(i):
    u = i / max(FRAMES - 1, 1)
    img = BACKGROUND.copy()          # not a repaint - a copy

    occluders = build_occluders(u)
    lamp = light_position(u)         # move THIS and every shadow follows

    # what the lamp can see, drawn at two ranges for falloff
    glow = Layer(WIDTH, HEIGHT)
    for reach, alpha in ((900.0, 40), (260.0, 50)):
        poly = light.visibility_polygon(lamp, occluders, reach)
        light.draw_light_polygon(glow, lamp, poly, (255, 210, 140, alpha))
    img.paste(glow)

    # the walls, lit properly rather than filled flat
    light.shade_segments(img, occluders, [(lamp, 420.0, LAMP_COLOR)], WALL)
    return img
```

Note what is *not* here: nothing computes a shadow. The dark parts of the frame are dark because no ray reached them, and that is the property worth protecting — the moment you start drawing a shadow by hand, you have lost the thing this assignment was teaching.