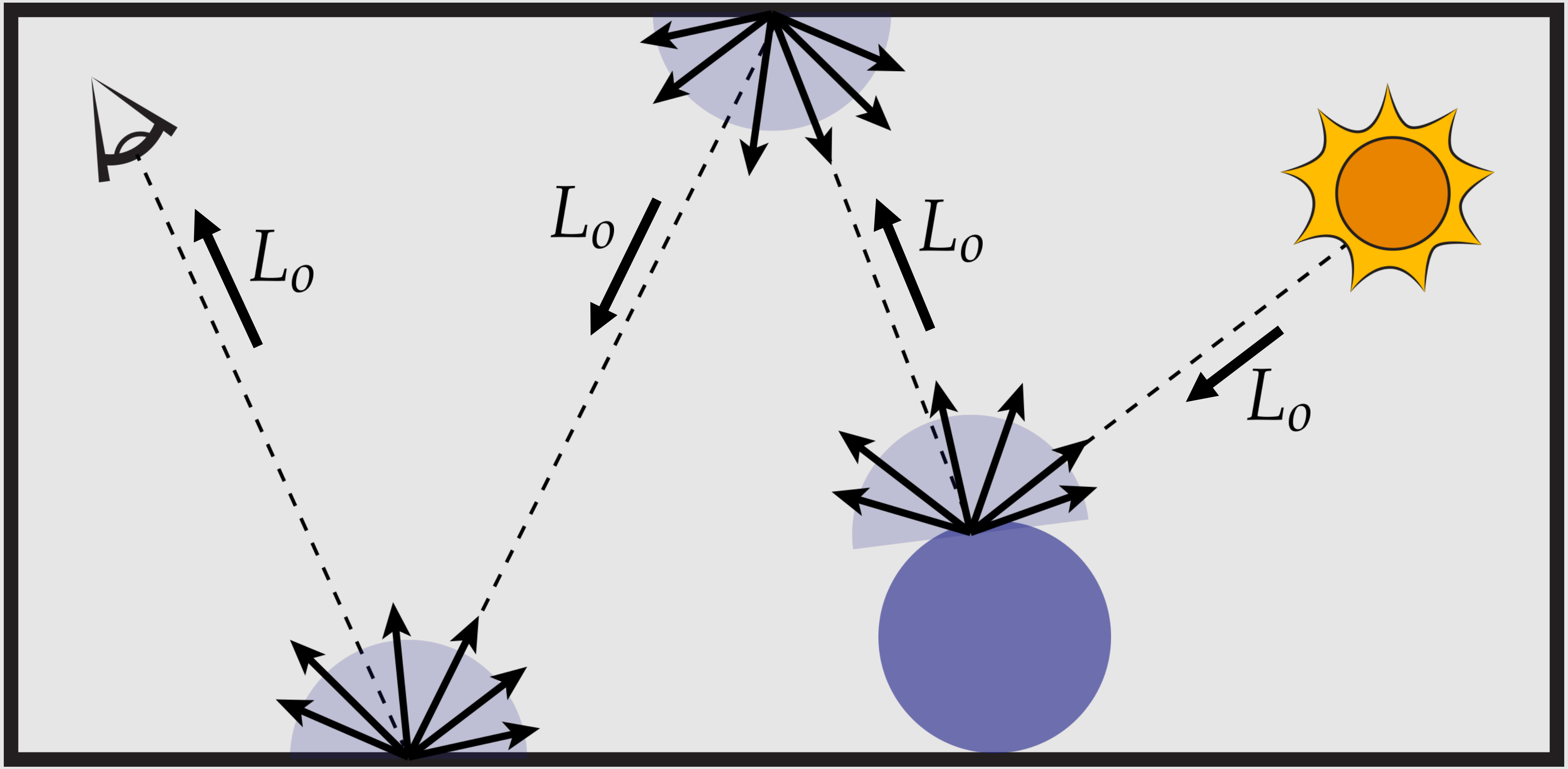
The Rendering Equation

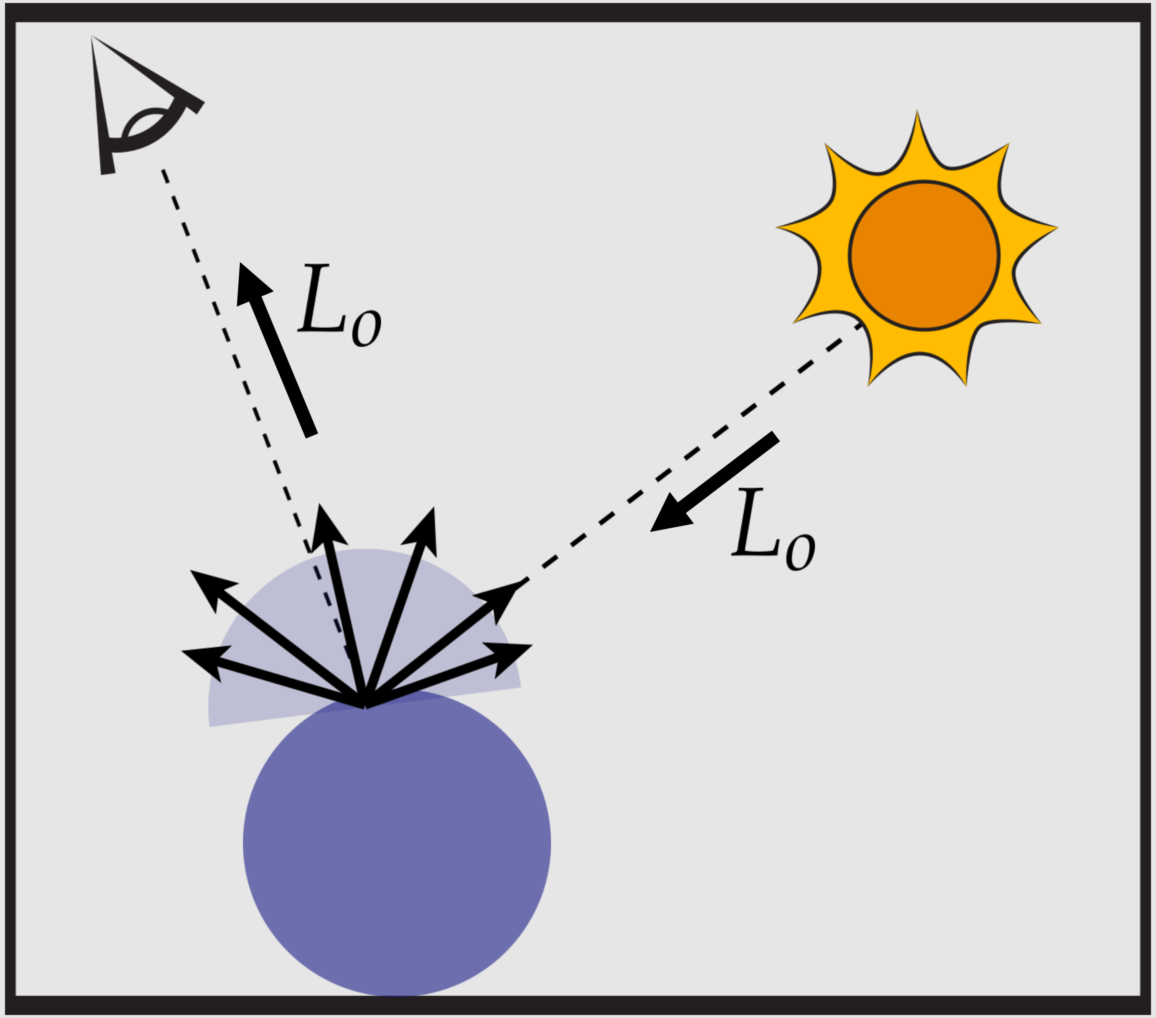
- The Rendering Equation
- A Simple Path-Tracer
- Camera Rays

Tracing Rays



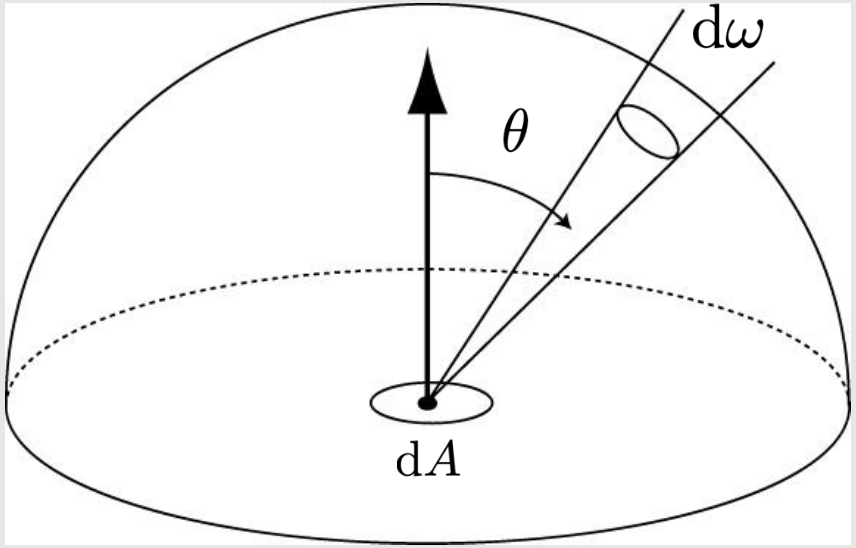
Tracing Rays

- **Goal:** trace light rays around the scene
  - Rays bounce around illuminating objects before reaching a camera
- Think of light rays as packets of info
  - When light hits an object, it picks up the object's color before moving onto the next object
- **Recall:** absorption spectrum
  - Any colors not absorbed are emitted back out



The Rendering Equation

$$E = \int_{\mathrm{H}^2} L(\omega) \cos\theta \, d\omega$$



The Rendering Equation should:

- Be recursive
- Have a base case
- Govern how light scatters (reflectance)

$$(\text{ recursive definition }) = (\text{ base case }) + \int_{\mathcal{H}^2} (\text{ scattering function }) * L_i(\mathbf{p}, \omega_i) \cos\theta \, d\omega_i$$

The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

$L_o(\mathbf{p},\omega_o)$ (recursive definition)

$L_e(\mathbf{p},\omega_o)$ (base case)

$f_r(\mathbf{p},\omega_i \to \omega_o)$ (scattering function)

$L_i(\mathbf{p},\omega_i)$ (previous recursive call)

The Rendering Equation

$$L_o(\mathbf{p}, \omega_o) \;=\; L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i \to \omega_o) L_i(\mathbf{p}, \omega_i) \cos\theta \, d\omega_i$$

$L_o(\mathbf{p}, \omega_o)$	outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o
$L_e(\mathbf{p}, \omega_o)$	emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o
$f_r(\mathbf{p}, \omega_i \to \omega_o)$	scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o
$L_i(\mathbf{p}, \omega_i)$	incoming radiance to point $\mathbf{p}$ from direction ω_i

The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \; = \; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

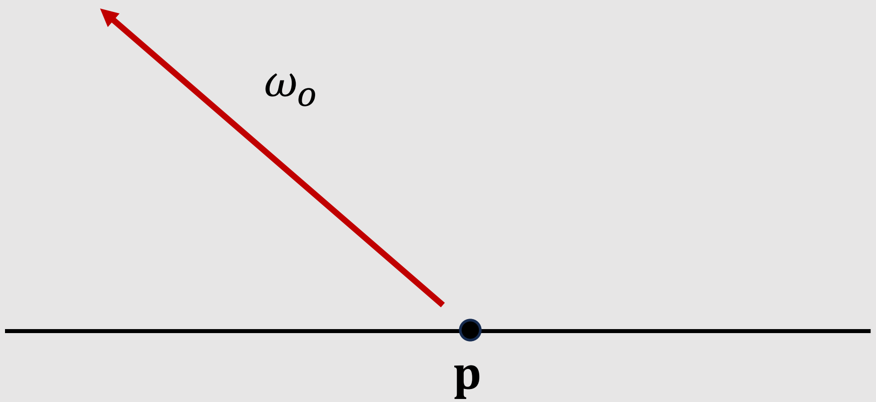
$L_o(\mathbf{p},\omega_o)$ outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o

$L_e(\mathbf{p},\omega_o)$ emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o

$f_r(\mathbf{p},\omega_i \to \omega_o)$ scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o

$L_i(\mathbf{p},\omega_i)$ incoming radiance to point $\mathbf{p}$ from direction ω_i

Outgoing Radiance



- To know what an object looks like, we want to know its **outgoing radiance**
  - Carries color and radiometry information

- Outgoing radiance parameterized by a ray with point $\mathbf{p}$ in outgoing direction ω_o
  - Where is the light coming from, and at what direction is it headed

- Want to solve for the outgoing radiance into the camera
  - The rendering equation helps us get there

The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

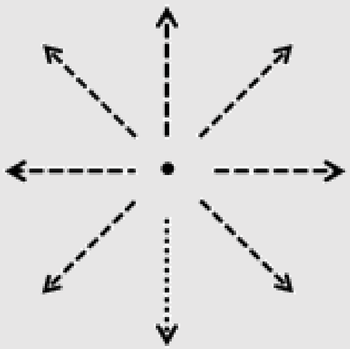
$L_o(\mathbf{p},\omega_o)$	outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o
$L_e(\mathbf{p},\omega_o)$	emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o
$f_r(\mathbf{p},\omega_i \to \omega_o)$	scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o
$L_i(\mathbf{p},\omega_i)$	incoming radiance to point $\mathbf{p}$ from direction ω_i

Recall: The Light Source


Kirby & The Forgotten Land (2022) Nintendo

- Light sources emit electromagnetic radiation that we view as light
  - In this class, we will treat light as a particle
  - Nice property: light paths are **ray-like**
    - We know how to work with rays
- Adding light into our scenes allow us to illuminate color
  - **A scene without lights will be just black**
  - Light bounces off objects (emittance), until it hits a sensor (eyes, camera, etc.)
- A light will have outgoing radiance at point $\mathbf{p}$ in some outgoing direction ω_o
  - The way $\mathbf{p}$ and ω_o are defined determines the light source!

Point Light



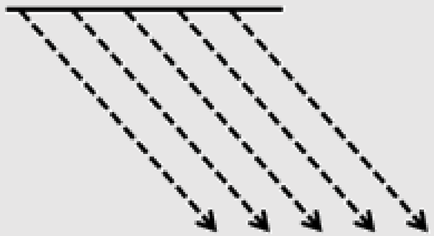
- Defined by:
  - $\mathbf{p}$ = [x, y, z] origin

- Light rays generated from all directions
- Intensity falls of with radius $\propto \frac{1}{r^2}$
- Very easy to check for visibility
  - Every point in active area

- Extension to Point Light: **Area Light**
  - Light generated from rectangle

- Extension to Point Light: **Spherical Light**
  - Light generated from sphere

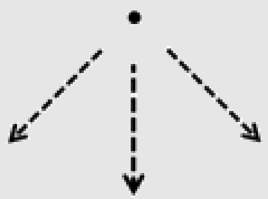
Directional Light



- Defined by:
  - ω_o= [x, y, z] direction
    - Can be simplified to ω_o= [x, y]
    - Normalized 3D coordinates can be written in 2D

- Light rays generated from infinity in the direction specified
- No fall-off of energy
- Very easy to check for visibility
  - Every point in active area

Spot Light



- Defined by:
  - $\mathbf{p}$ = [x, y, z] origin
  - ω_o = [x, y] direction (same optimization)
  - [hfov] horizontal field of view
  - [vfov] vertical field of view
    - Same parameters as a camera

- Light rays generated from directions within field of view
- Intensity falls of with radius $\propto \frac{1}{r^2}$
- Challenging to check for visibility
  - Point must fall in the light's field of view

Environmental Light

- Defined by:
    - An image!

- Sample light directly from an image
- No intensity falloff. Image distance is at infinity
- Very easy to check for visibility
    - Every point in active area

- We'll learn how to build this in a future lecture

Uncharted 4 (2016) Naughty Dog

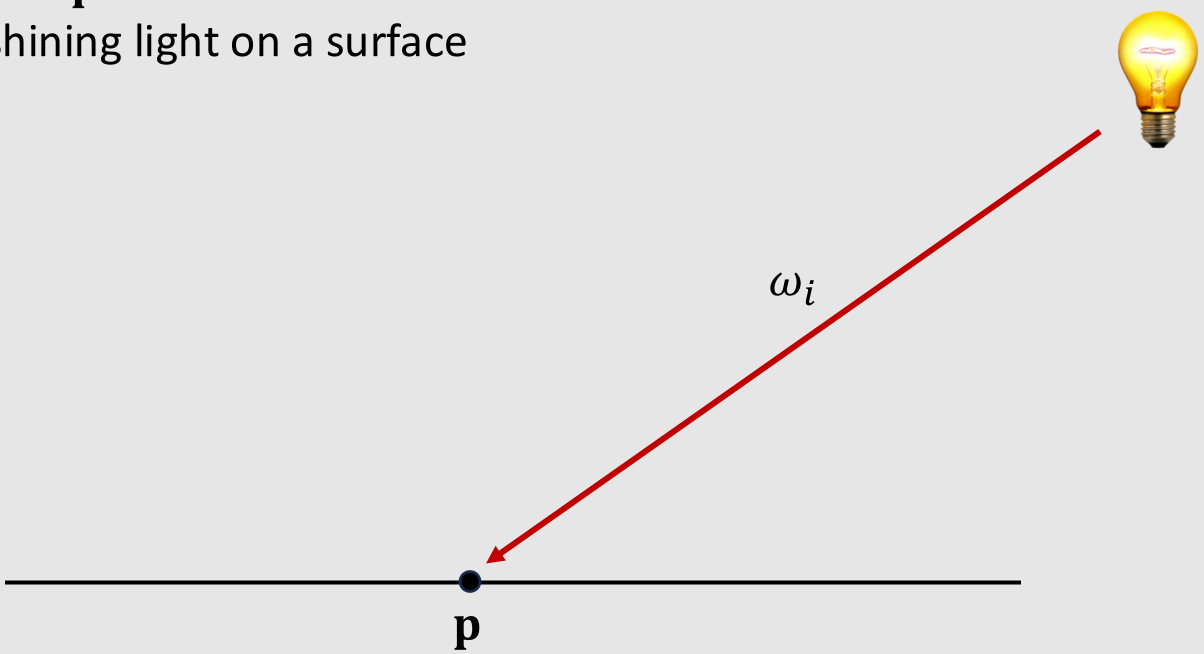
The Rendering Equation

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i \to \omega_o) L_i(\mathbf{p}, \omega_i) \cos\theta \, d\omega_i$$

$L_o(\mathbf{p}, \omega_o)$	outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o
$L_e(\mathbf{p}, \omega_o)$	emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o
$f_r(\mathbf{p}, \omega_i \to \omega_o)$	scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o
$L_i(\mathbf{p}, \omega_i)$	incoming radiance to point $\mathbf{p}$ from direction ω_i

Incoming Radiance

- Measures how much light is coming in from direction ω_i onto the incident surface point $\mathbf{p}$
  - **Example:** light source shining light on a surface



The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

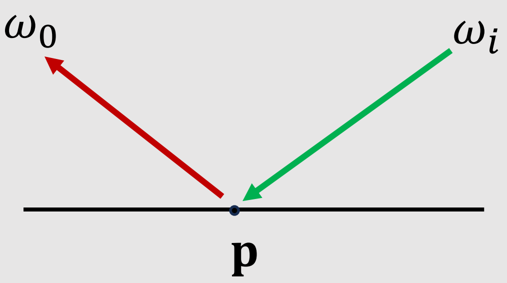
$L_o(\mathbf{p},\omega_o)$ outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o

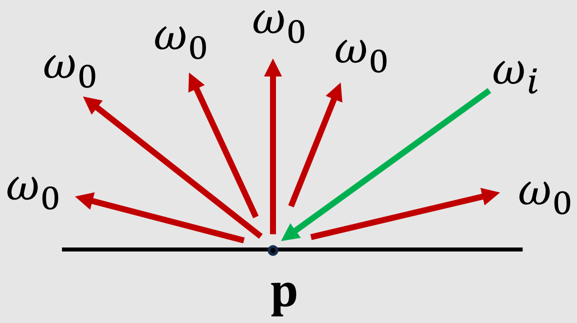
$L_e(\mathbf{p},\omega_o)$ emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o

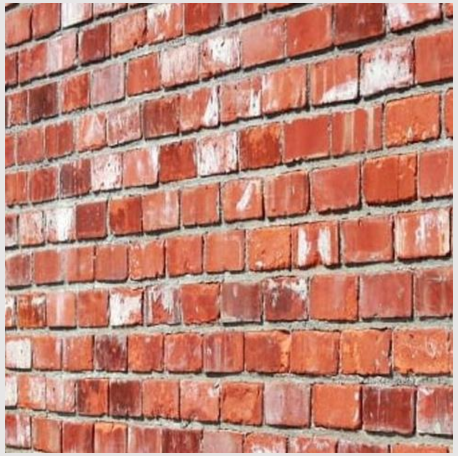
$f_r(\mathbf{p},\omega_i \to \omega_o)$ scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o

$L_i(\mathbf{p},\omega_i)$ incoming radiance to point $\mathbf{p}$ from direction ω_i

Reflecting Light



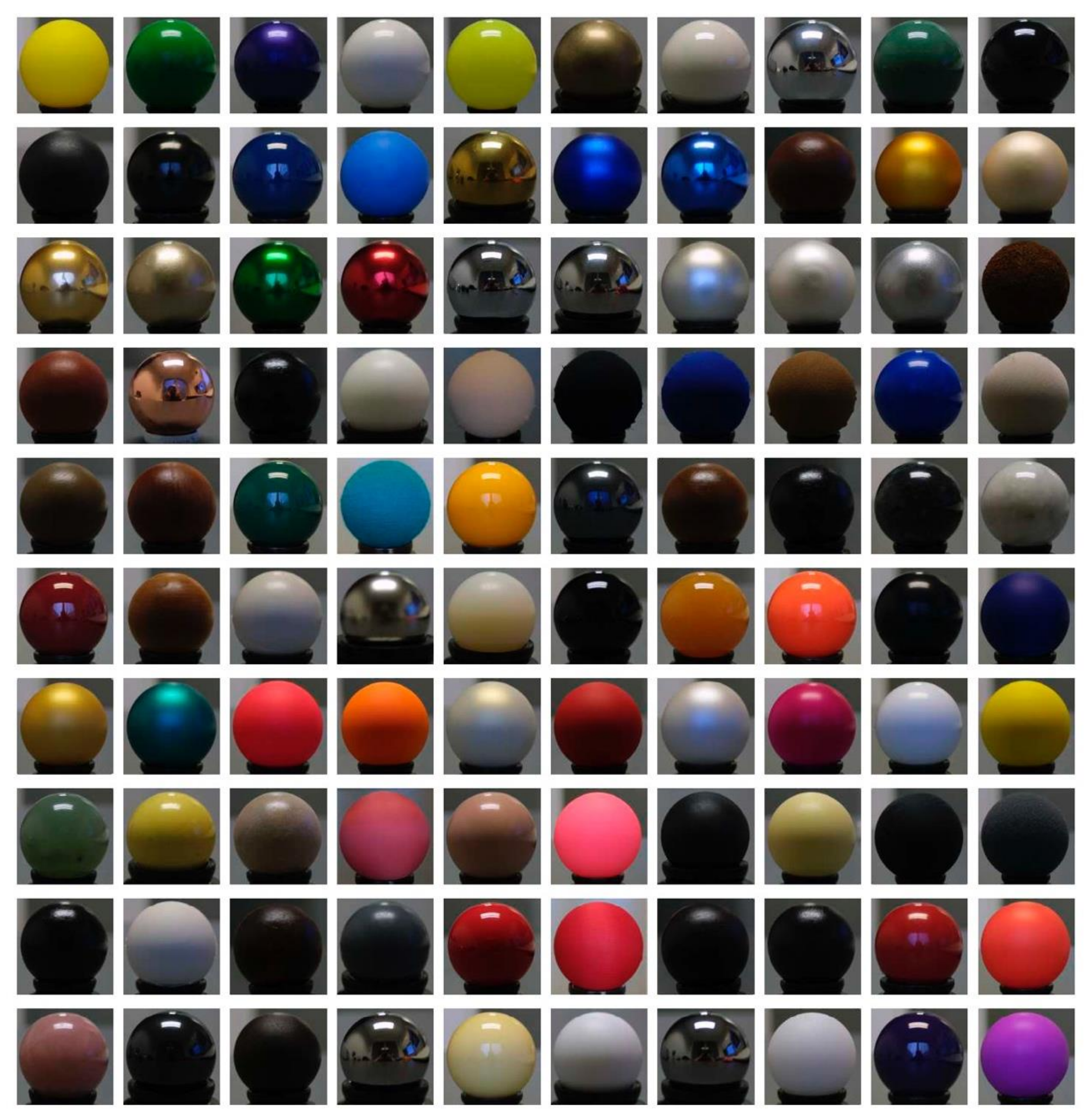




Some objects, like mirrors,
will reflect light in a single direction

Some objects, like brick walls,
will reflect light in all directions

There's A Lot Of BRDFs



The Rendering Equation

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$

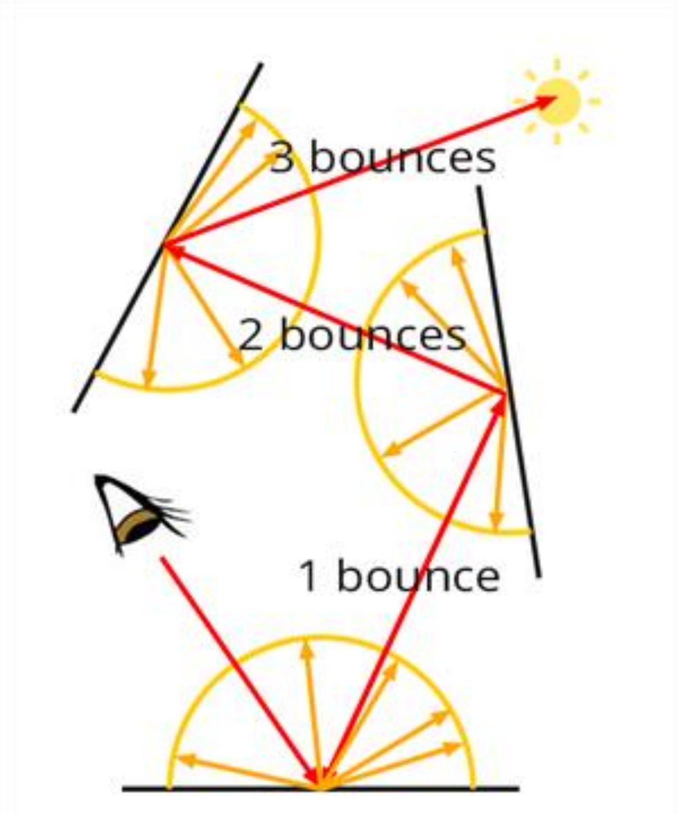
$L_o(\mathbf{p},\omega_o)$	outgoing radiance at point $\mathbf{p}$ in outgoing direction ω_o
$L_e(\mathbf{p},\omega_o)$	emitted radiance at point $\mathbf{p}$ in outgoing direction ω_o
$f_r(\mathbf{p},\omega_i \to \omega_o)$	scattering function at point $\mathbf{p}$ from incoming direction ω_i to outgoing direction ω_o
$L_i(\mathbf{p},\omega_i)$	incoming radiance to point $\mathbf{p}$ from direction ω_i

what about the integral?

Recap: Radiance In Rendering

- Surfaces are planar (Ex: triangles)
    - Light can enter surface from any angle around the hemisphere

- Outgoing radiance is a function of incoming radiance from every possible direction around the hemisphere



Scratch-A-Pixel (2018)

Just One Small Issue…

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o)+\int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o)L_i(\mathbf{p},\omega_i)\cos\theta\, d\omega_i$$

The integral assumes infinite sampling around the hemisphere

- Infinite lighting
- Infinite rays
- Infinite ray bounces

Computers can only process finite amounts of data

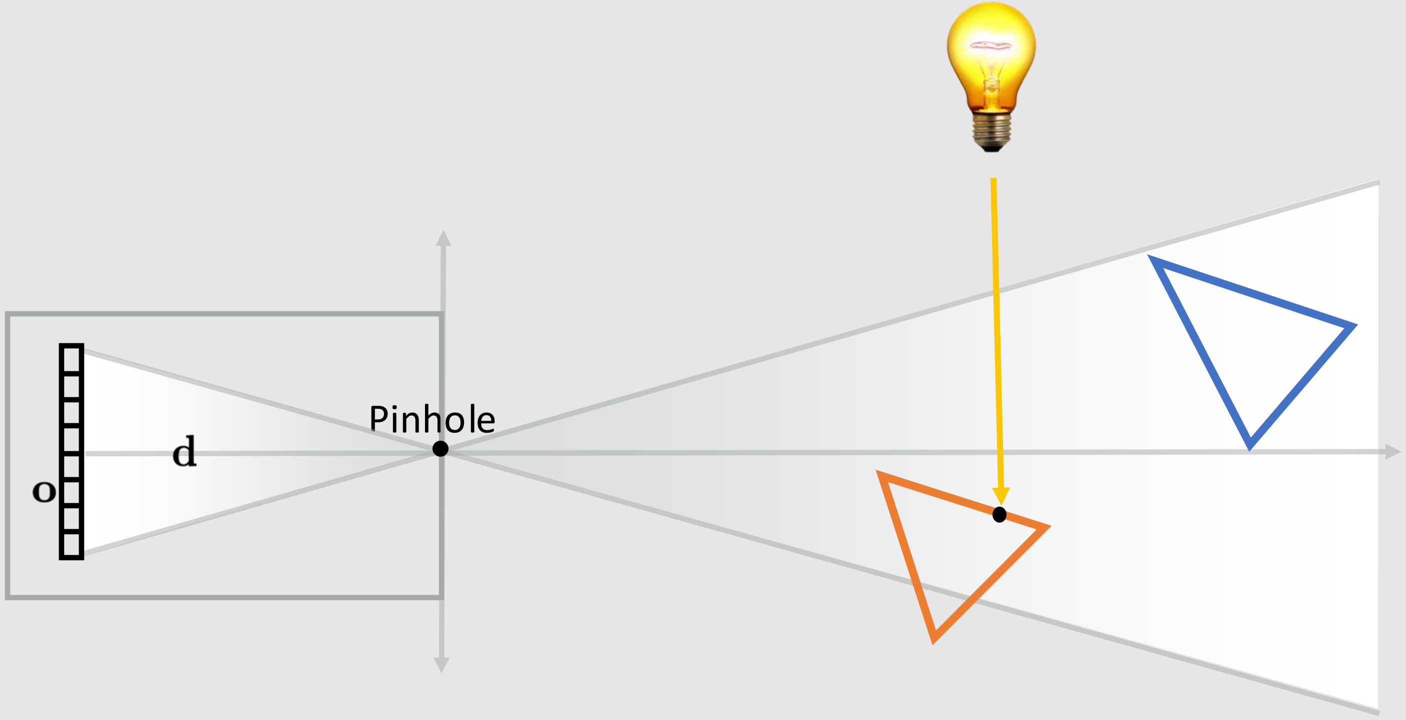
- Finite lighting
- Finite rays
- Finite ray bounces

- ~~The Rendering Equation~~
- A Simple Path-Tracer
- Camera Rays

https://tinyurl.com/362-mixer

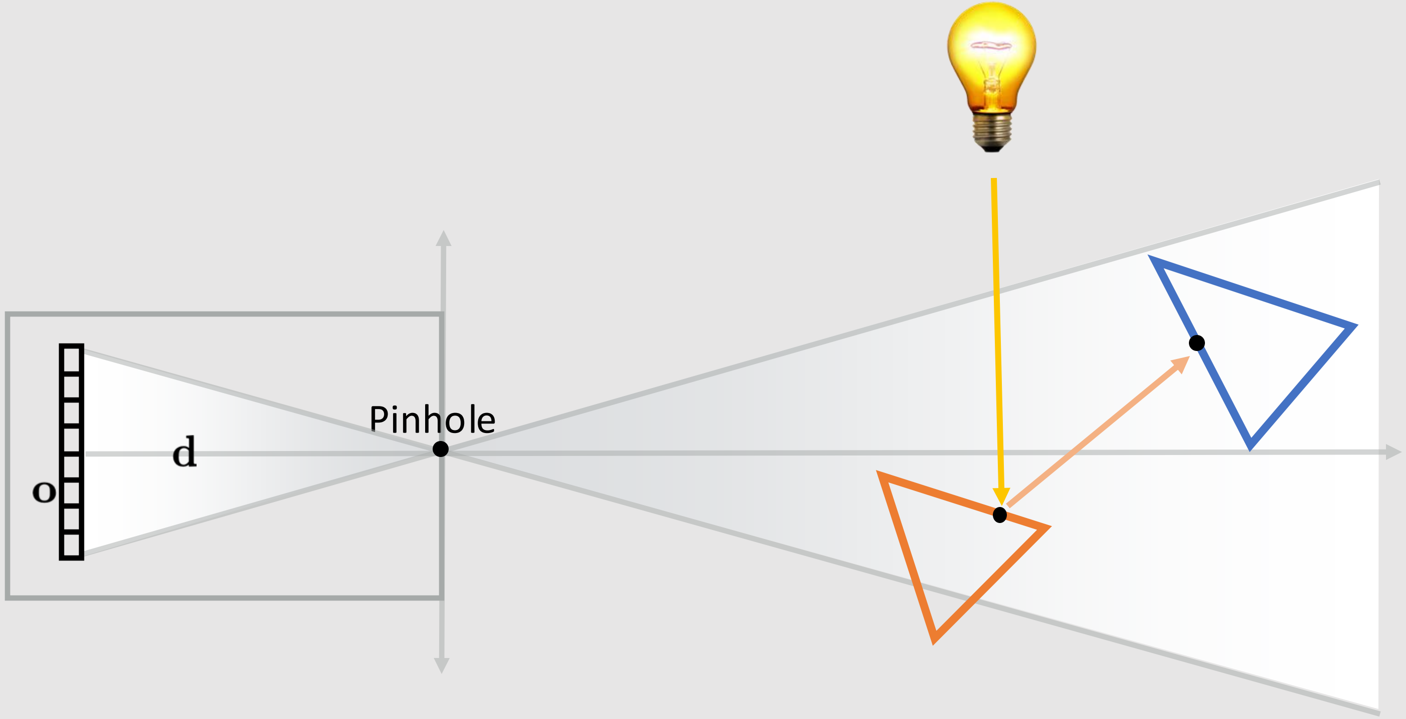
Example Of A Simple Renderer

- Yellow light ray generated from light source



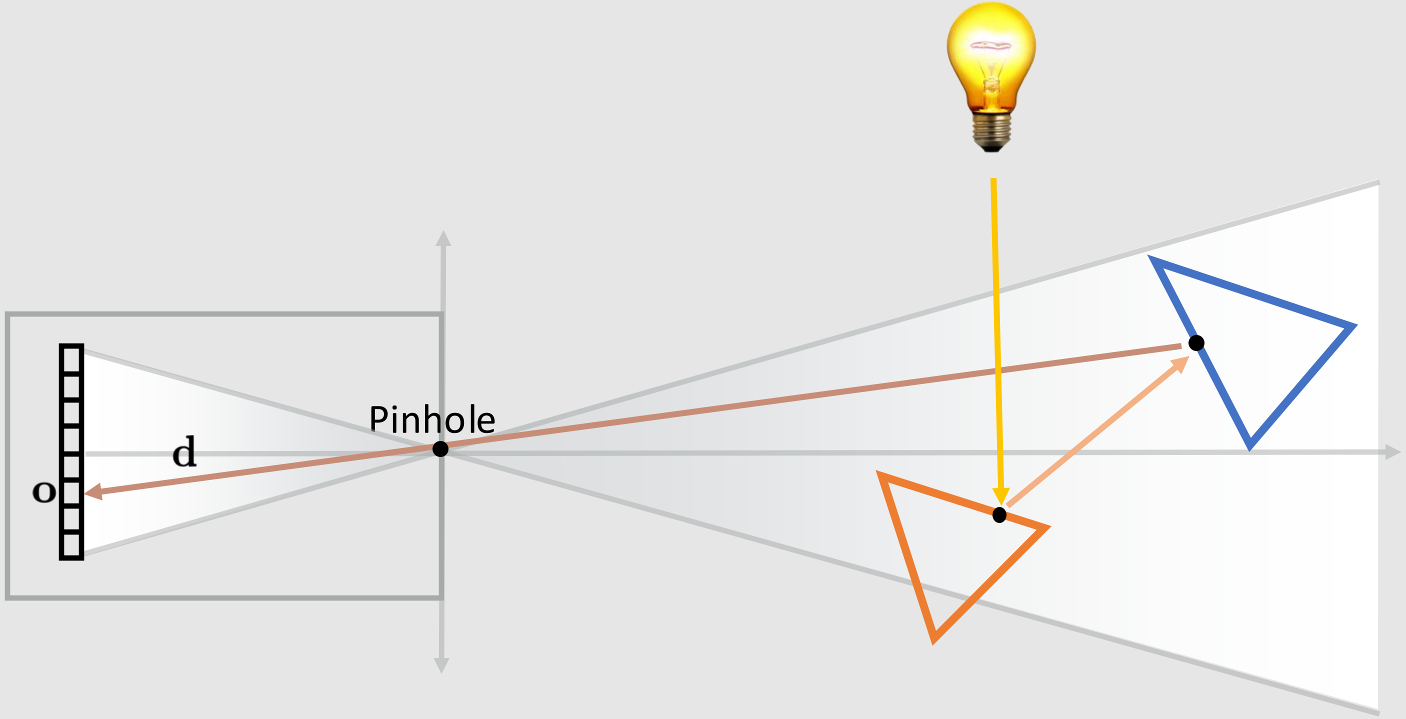
Example Of A Simple Renderer

- Yellow light ray generated from light source
- Ray hits orange specular surface
  - Emits a ray in reflected direction
  - Mixes yellow and orange color



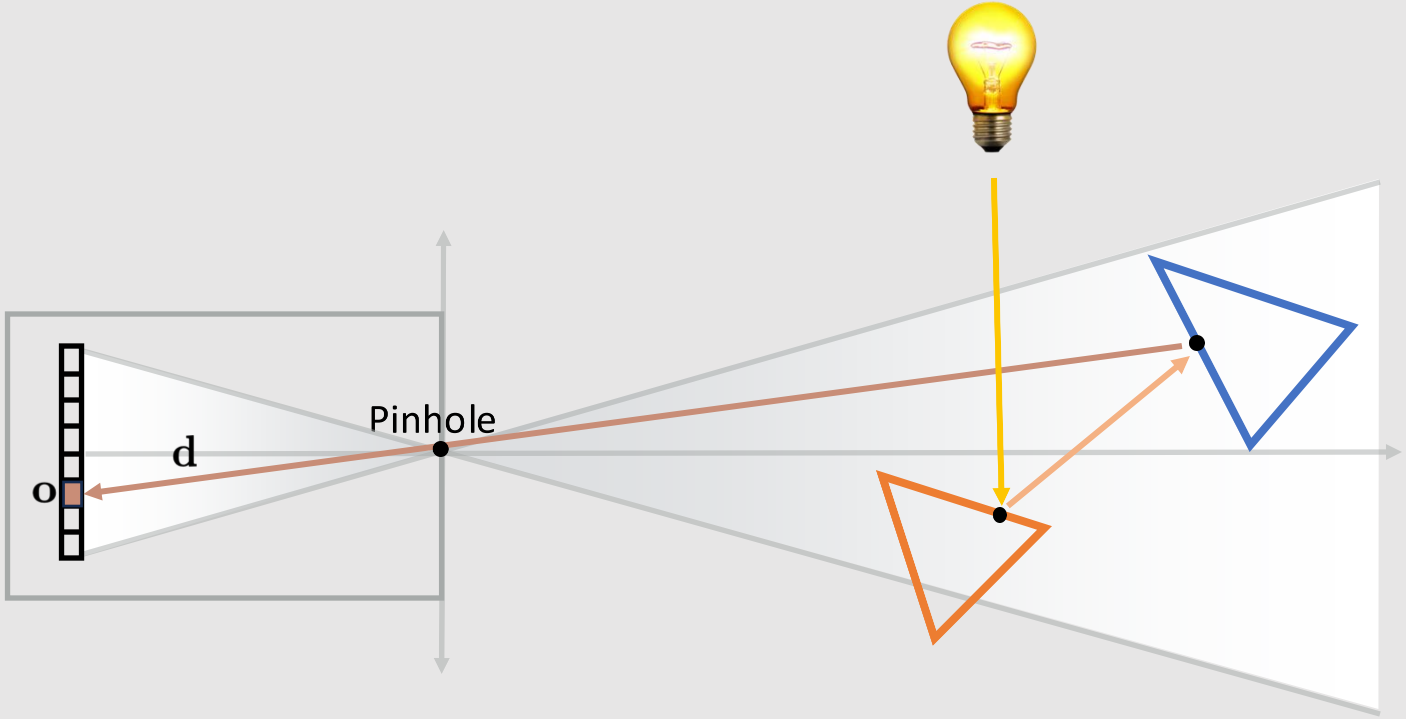
Example Of A Simple Renderer

- Yellow light ray generated from light source
- Ray hits orange specular surface
  - Emits a ray in reflected direction
  - Mixes yellow and orange color
- Ray hits blue specular surface
  - Emits a ray in reflected direction
  - Mixes blue and yellow and orange



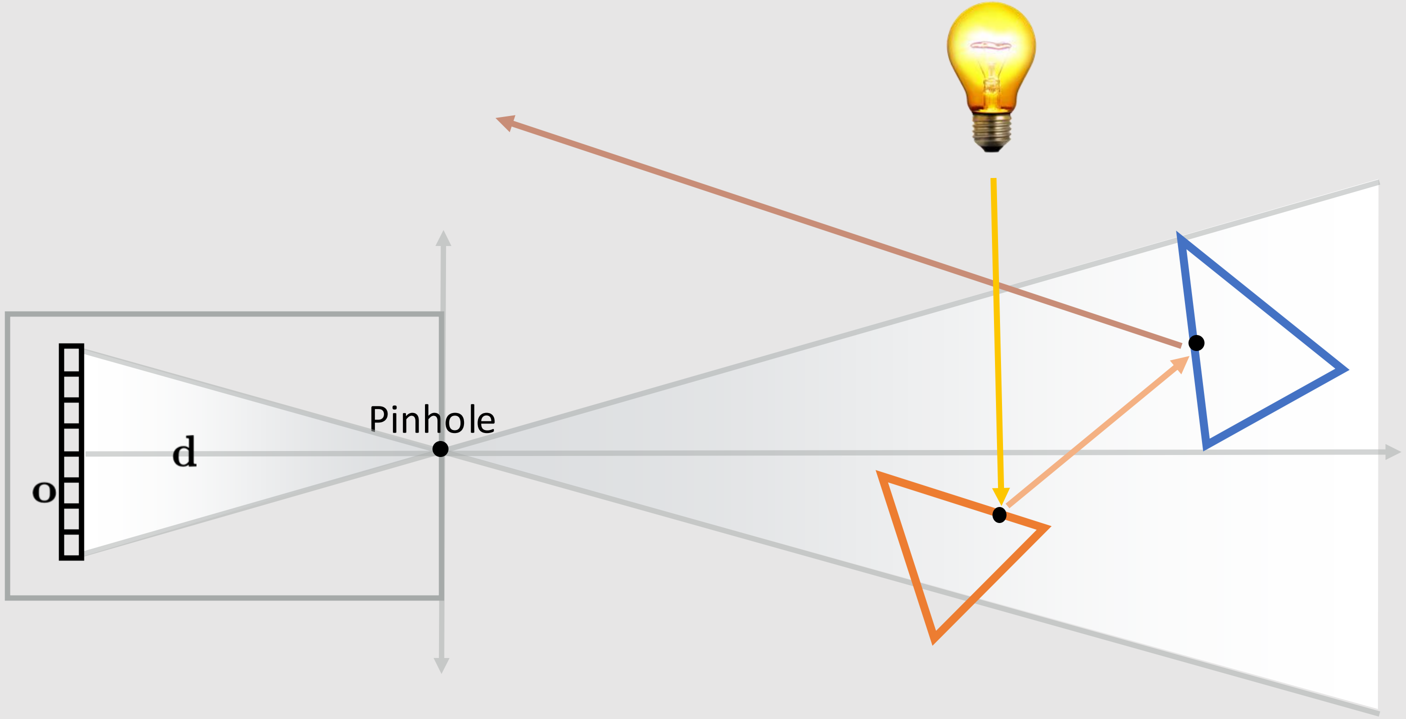
Example Of A Simple Renderer

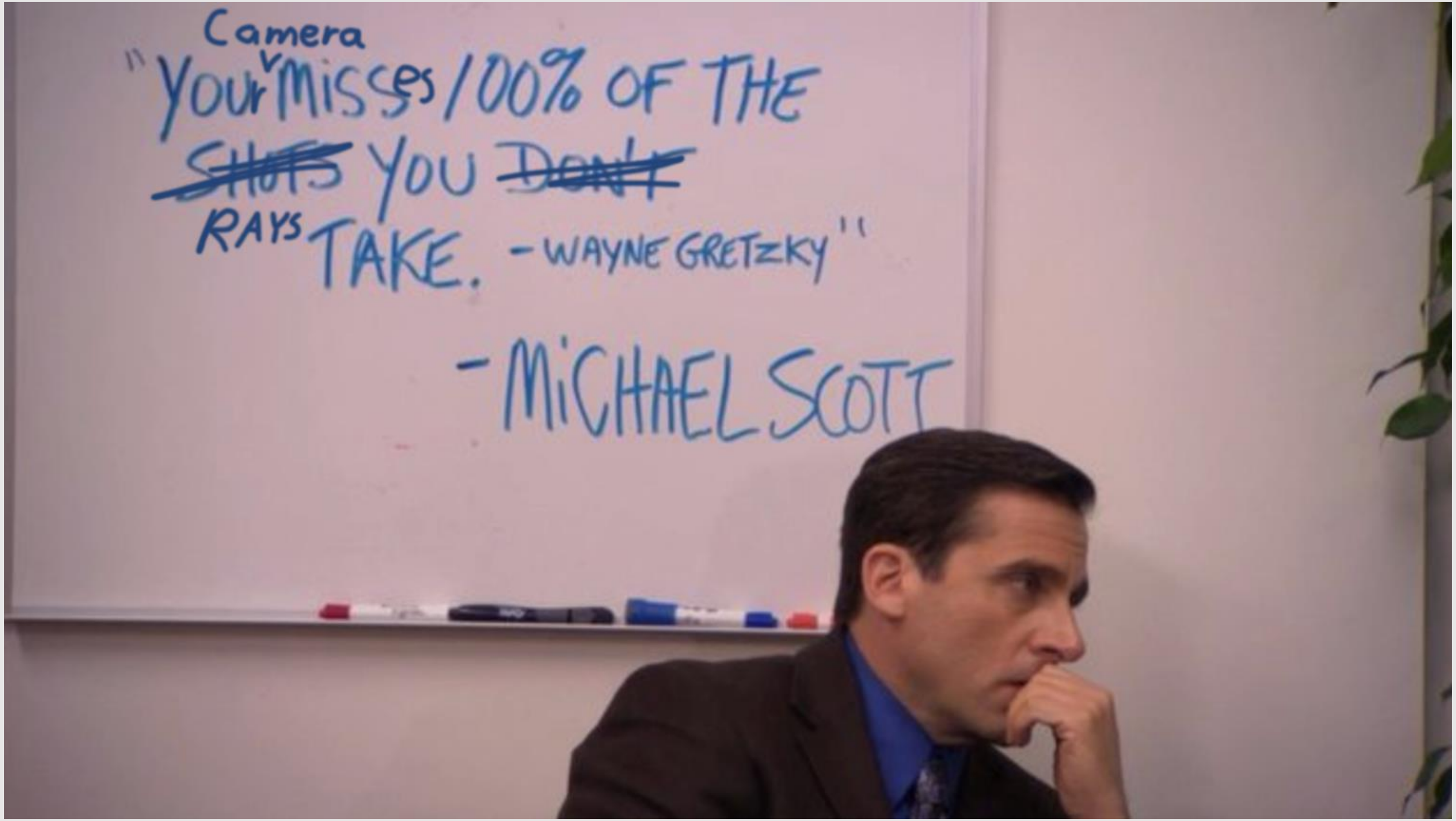
- Yellow light ray generated from light source
- Ray hits orange specular surface
  - Emits a ray in reflected direction
  - Mixes yellow and orange color
- Ray hits blue specular surface
  - Emits a ray in reflected direction
  - Mixes blue and yellow and orange
- Ray passes through pinhole camera
  - Light recorded on photoelectric cell
  - Incident pixel will be brown in final image



Example Of A Simple Renderer

- **Problem:** cannot always count on rays entering camera!
  - **Example:** if I turn the blue triangle a bit, the ray goes off into the void
- Compute wasted on a ray that doesn't contribute to the final image!




Camera
"Your Misses 100% OF THE
~~SHOTS~~ YOU ~~DON'T~~
RAYS TAKE. - WAYNE GRETZKY"
- MICHAEL SCOTT

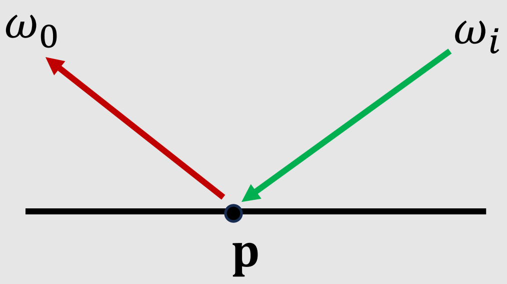
Idea: What if we trace a ray from the camera instead?

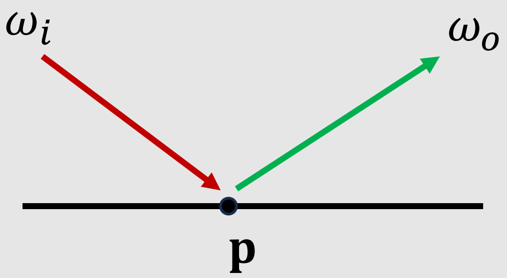
Hemholtz Reciprocity

- Reversing the order of incoming and outgoing light does not affect the BRDF evaluation

$$f_r(\mathbf{p}, \omega_i \rightarrow \omega_o) = f_r(\mathbf{p}, \omega_o \rightarrow \omega_i)$$

- Critical to reverse path-tracing algorithms
  - Allows us to trace rays backwards and still get the same BRDF effect

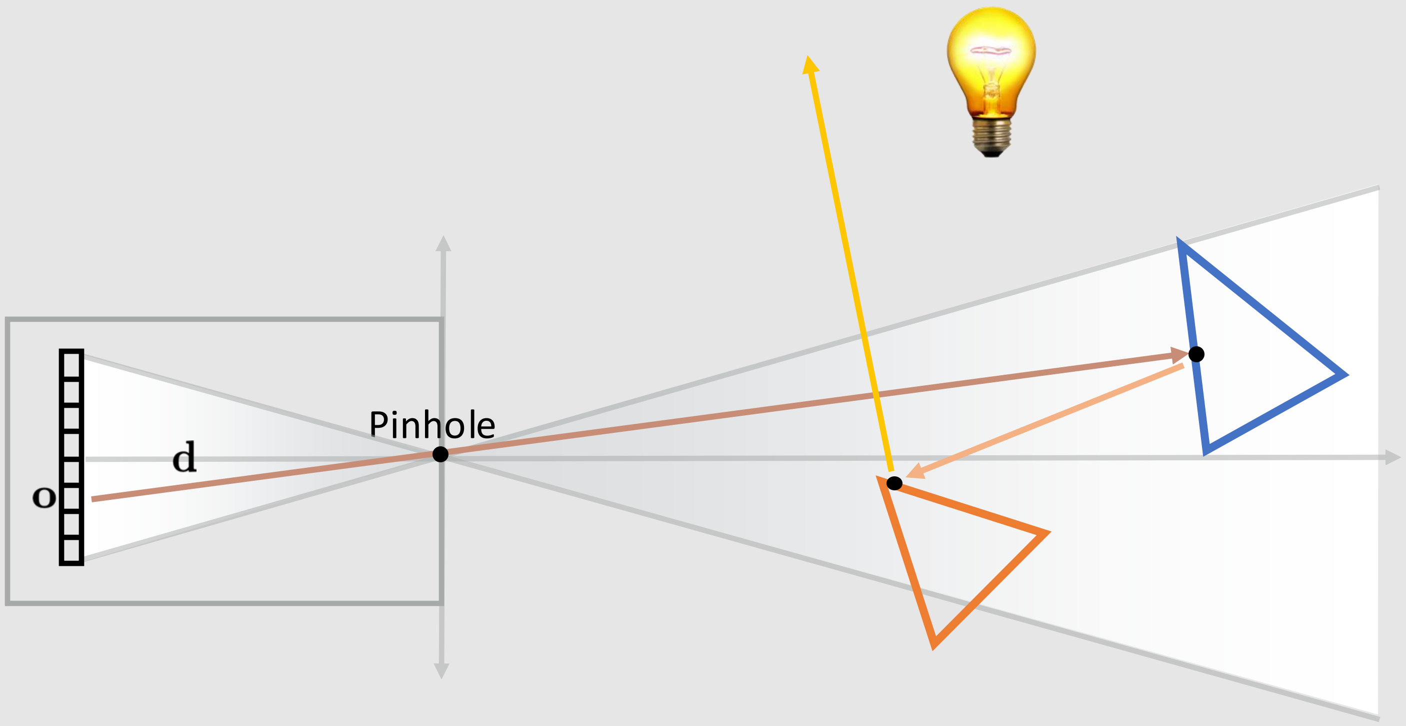




Example Of A Simple Backwards Renderer

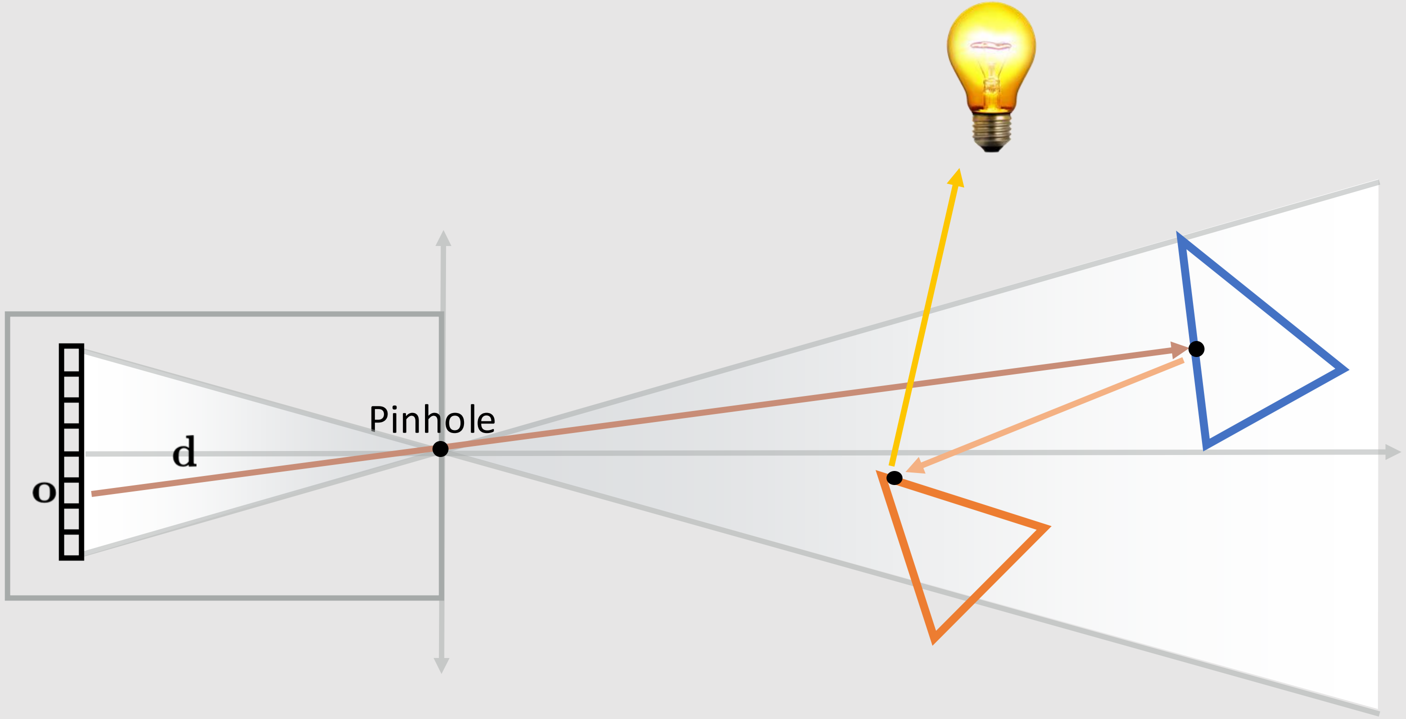
- Rays now traced out from the camera
  - Ray origin is pixel, direction faces pinhole
- **Issue #1:** How do we know the color of the rays now things are backwards?
- **Issue #2:** Rays still go to infinity!

Let's start with this



Example Of A Simple Backwards Renderer

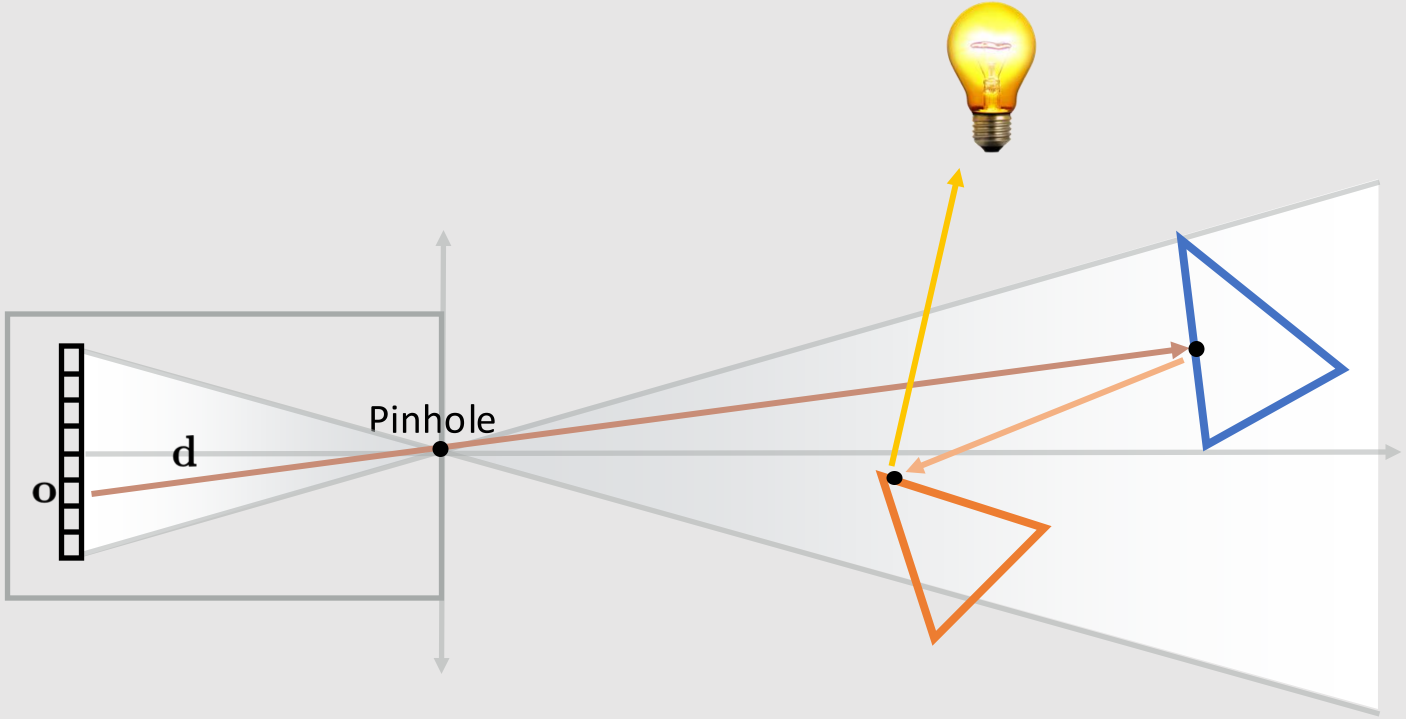
- **Issue #2:** Rays still go to infinity!
- After n-bounces, **terminate** the ray by constructing the ray towards the light source
  - If scene has multiple lights, pick one
- **Only works for BDRFs that are not ideal specular** (Ex: mirror, glass)!
  - If ideal specular, then continue to trace the ray until a non ideal specular surface is hit



Example Of A Simple Backwards Renderer

- **Issue #1:** How do we know the color of the rays now things are backwards?

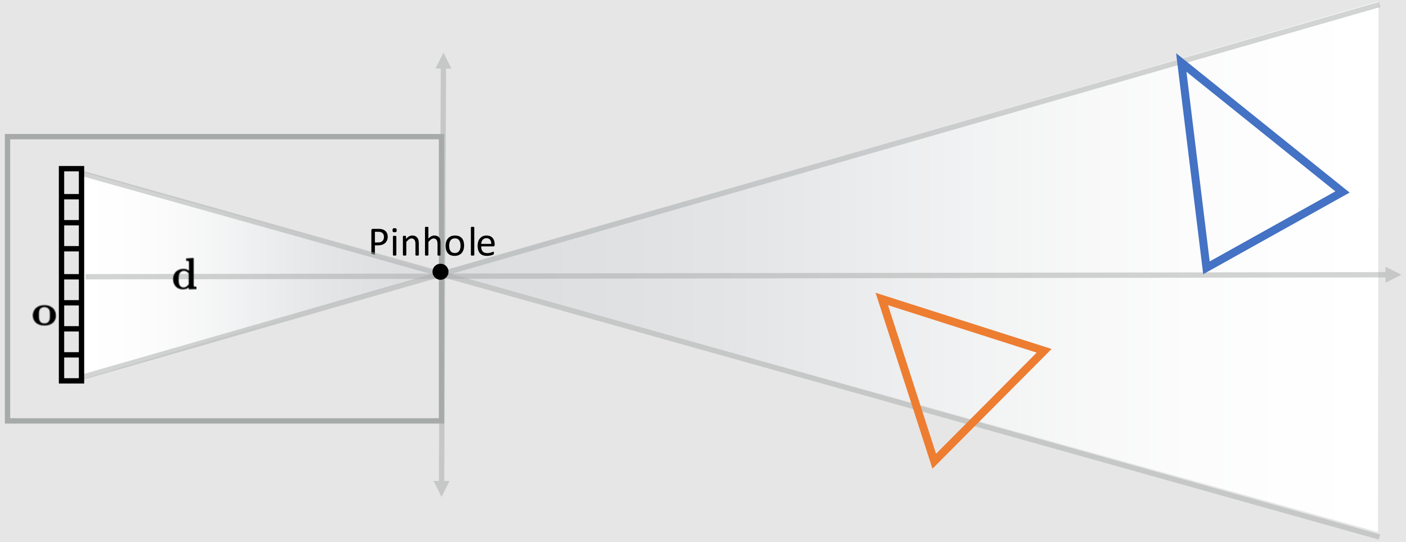
- Split the renderer into two parts:
  - **Path-trace** to find a path to the light source
  - **Backpropagate** the colors back to the pixel



Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o)+\int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \rightarrow \omega_o)L_i(\mathbf{p},\omega_i)\cos\theta\, d\omega_i$$



$L(pixel) =$

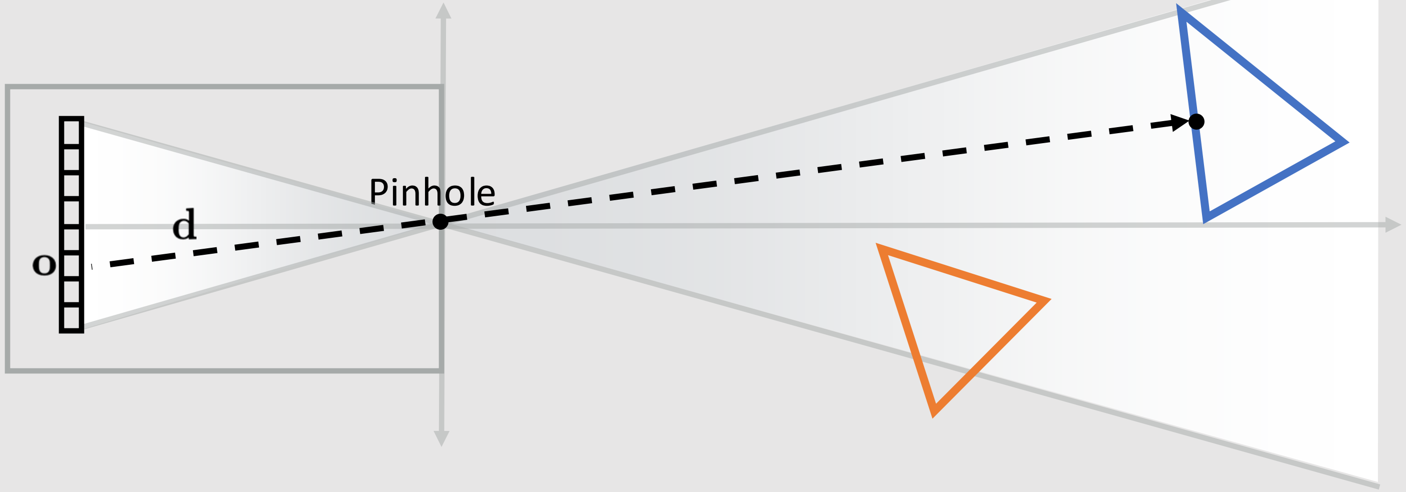
$L(pixel) =$

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o)+\int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \rightarrow \omega_o)L_i(\mathbf{p},\omega_i)\cos\theta\, d\omega_i$$

- Intersect △ , no emission □



$$L(pixel) = L_e(ray_1) + f_r(obj_1)[\qquad\qquad\qquad]$$

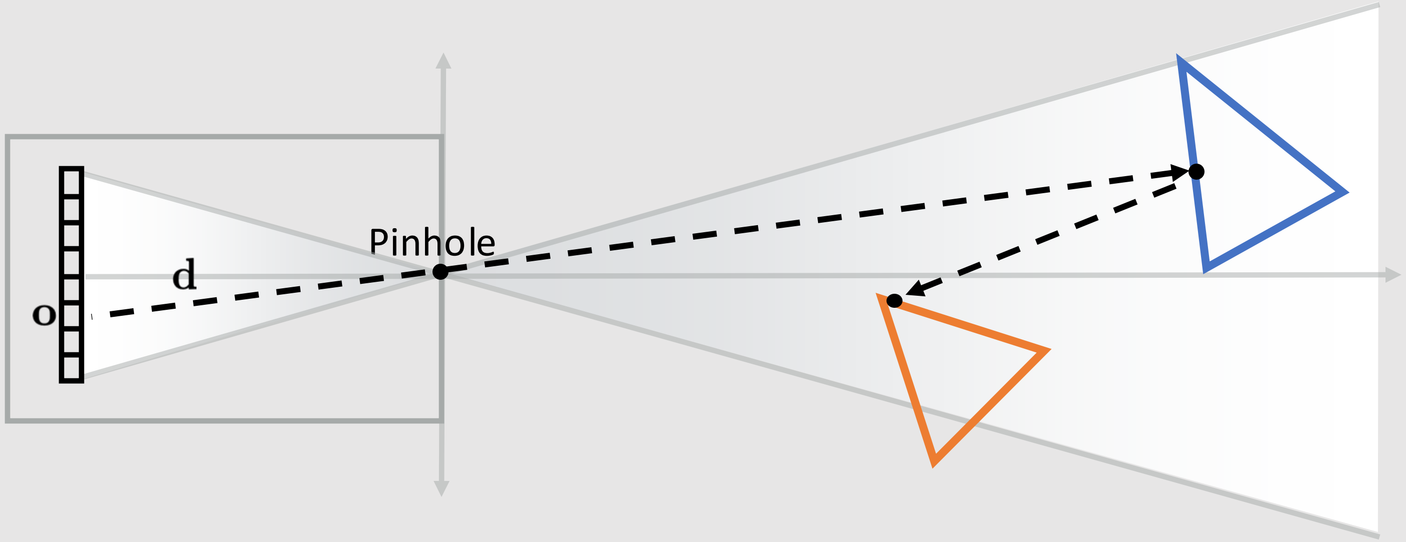
$$L(pixel) = \square + f_r(\triangle)[\qquad\qquad]$$

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \to \omega_o) L_i(\mathbf{p},\omega_i)\cos\theta\, d\omega_i$$

- Intersect △ , no emission □
- Intersect △ , no emission □



$$L(pixel) = L_e(ray_1) + f_r(obj_1)[L_e(ray_2) + f_r(obj_2)[\qquad\quad]$$

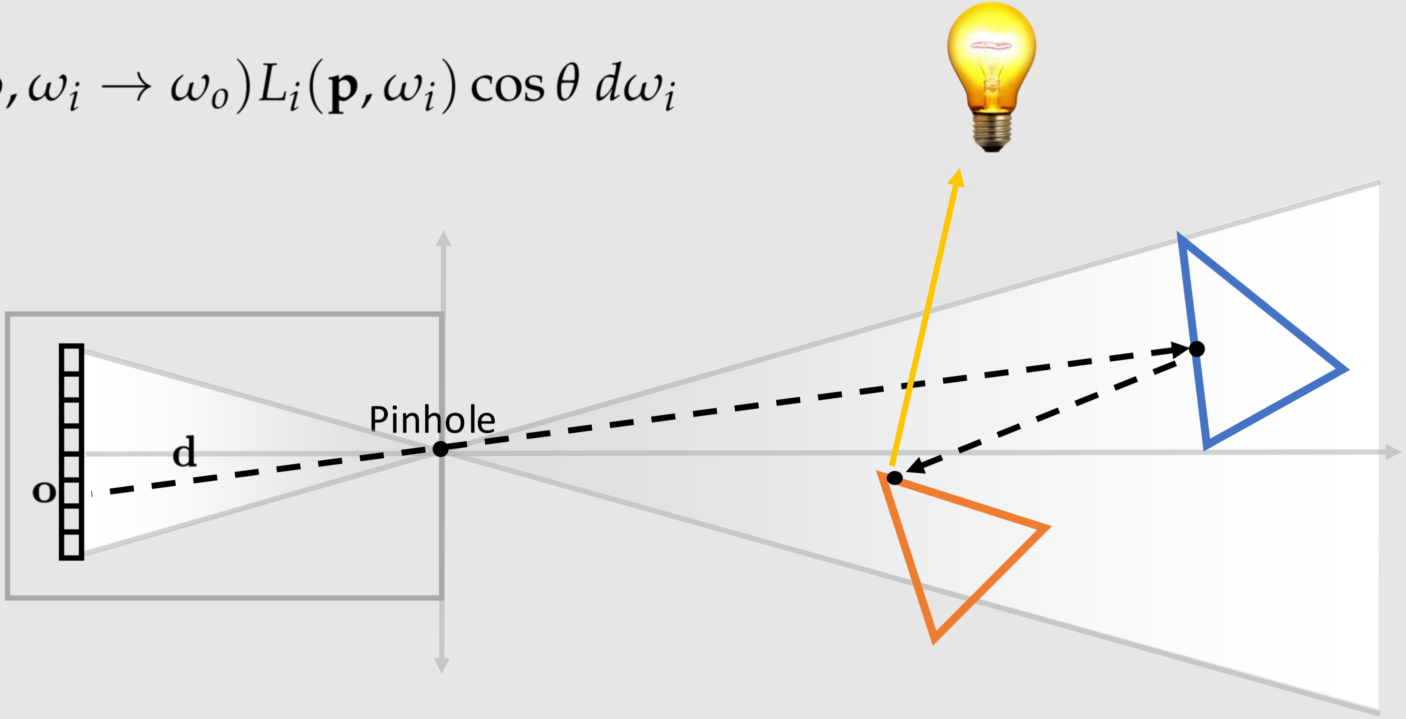
$$L(pixel) = \square + f_r(\triangle)[\;\square + f_r(\triangle)[\qquad]]$$

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p}, \omega_i \to \omega_o) L_i(\mathbf{p}, \omega_i) \cos\theta \, d\omega_i$$

- Intersect △ (blue), no emission □
- Intersect △ (orange), no emission □
- Ray terminate, emission ■ (yellow)



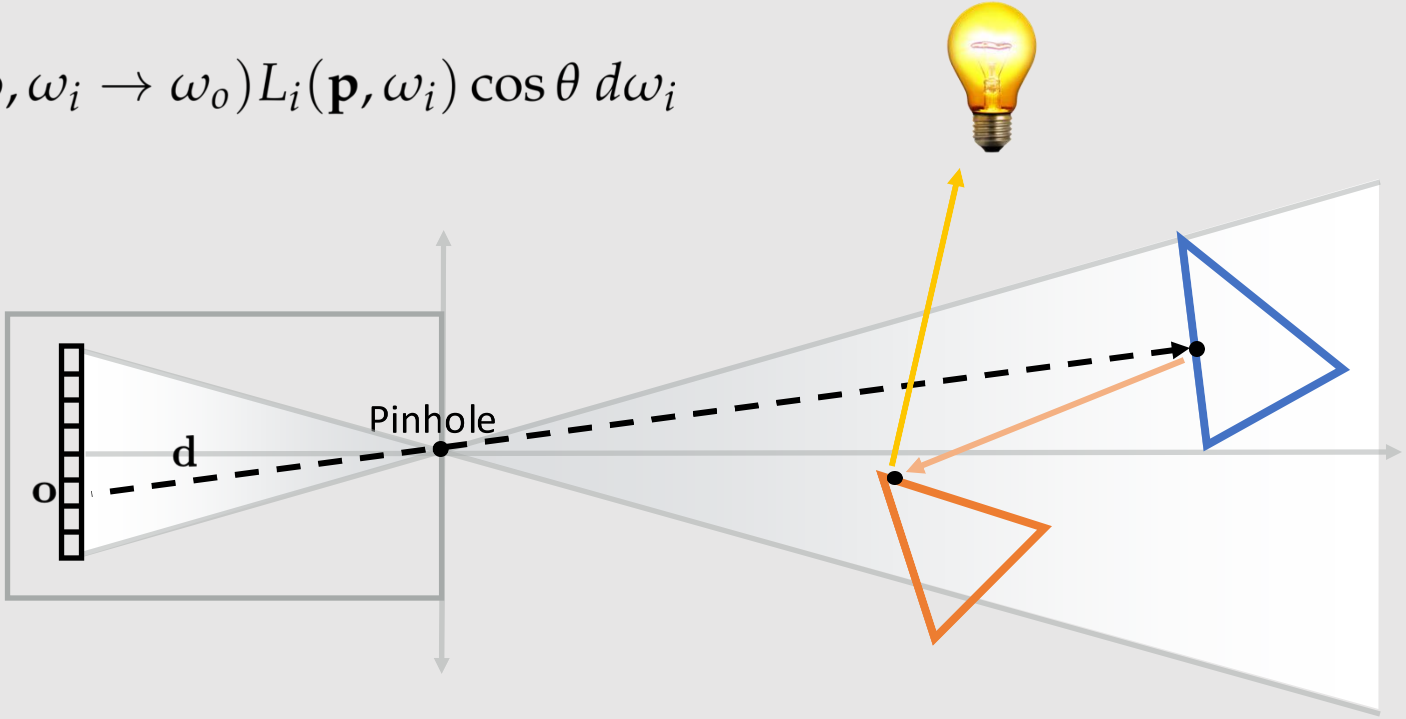
$$L(pixel) = L_e(ray_1) + f_r(obj_1)[L_e(ray_2) + f_r(obj_2)[L_e(ray_3)]]$$

$$L(pixel) = \square + f_r(\triangle_{blue})[\ \square + f_r(\triangle_{orange})[\ \blacksquare_{yellow}\ ]]$$

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) = L_e(\mathbf{p},\omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i \rightarrow \omega_o) L_i(\mathbf{p},\omega_i) \cos\theta \, d\omega_i$$



- Intersect △ (blue), no emission □
- Intersect △ (orange), no emission □
- Ray terminate, emission ■ (yellow)

$$L(pixel) = L_e(ray_1) + f_r(obj_1)[L_o(ray_2)]$$

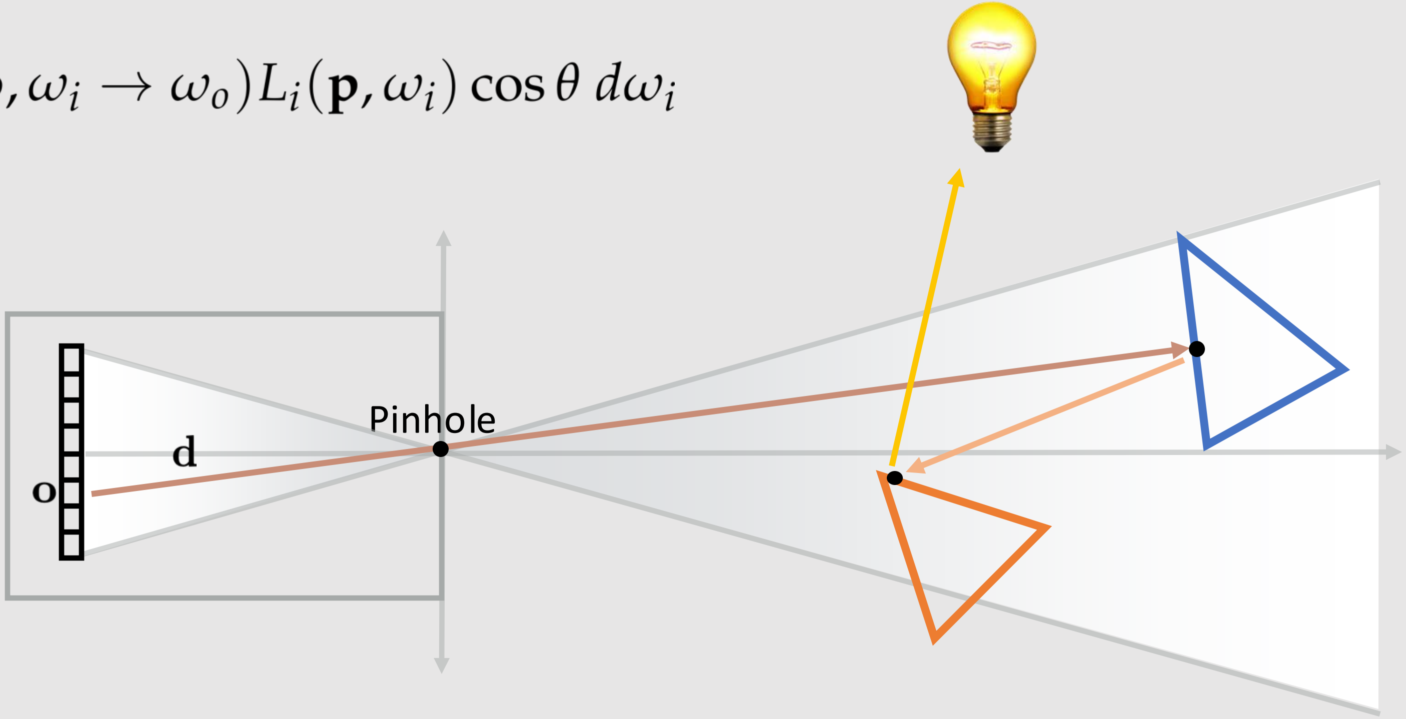
$$L(pixel) = \square + f_r(\triangle)[\blacksquare]$$

Example Of A Simple Backwards Renderer

[ray depth 2]

$$L_o(\mathbf{p},\omega_o) \;=\; L_e(\mathbf{p},\omega_o)+\int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i\rightarrow\omega_o)L_i(\mathbf{p},\omega_i)\cos\theta\, d\omega_i$$

- Intersect △, no emission □
- Intersect △, no emission □
- Ray terminate, emission ■



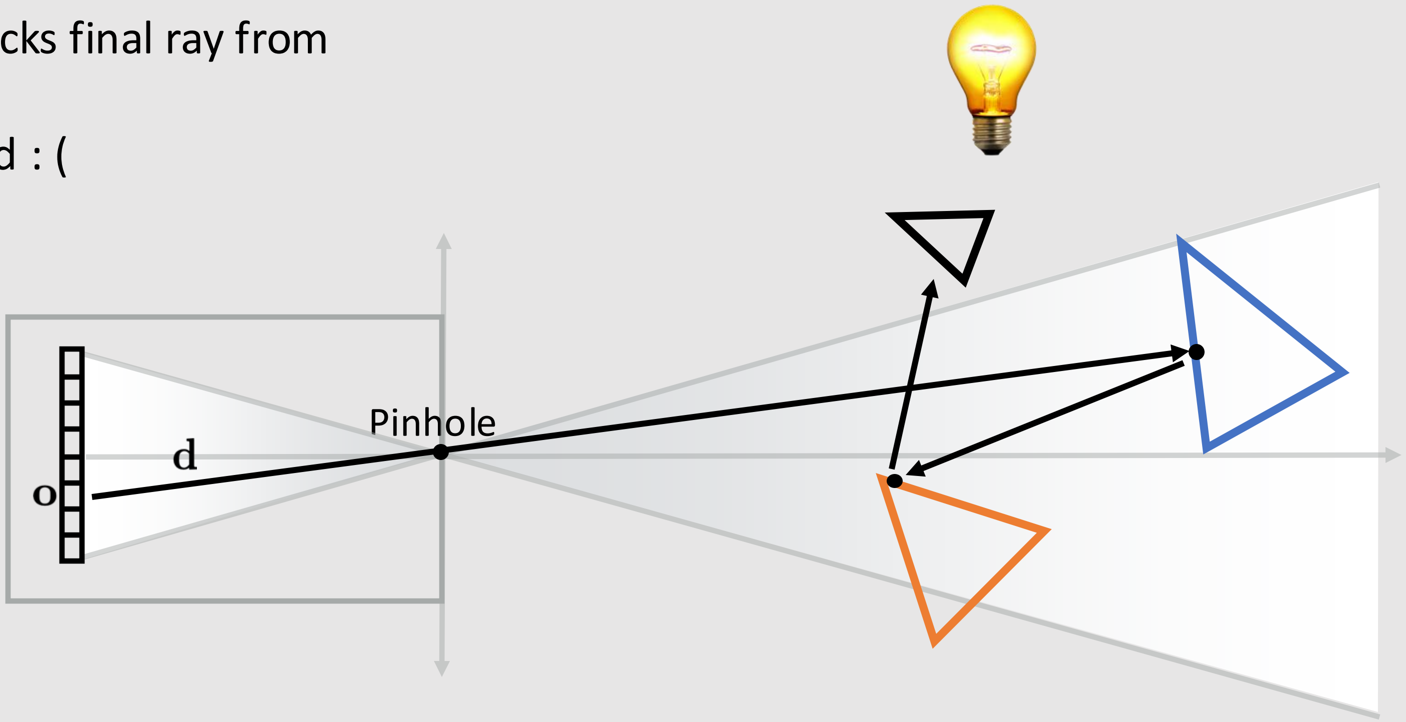
$$L(pixel) = L_o(ray_1)$$

$$L(pixel\,) = \blacksquare$$

Terminating Emission Occlusion

[ray depth 2]

- Possibility that geometry in the scene blocks final ray from reaching light source
  - No contribution returned, ray wasted : (



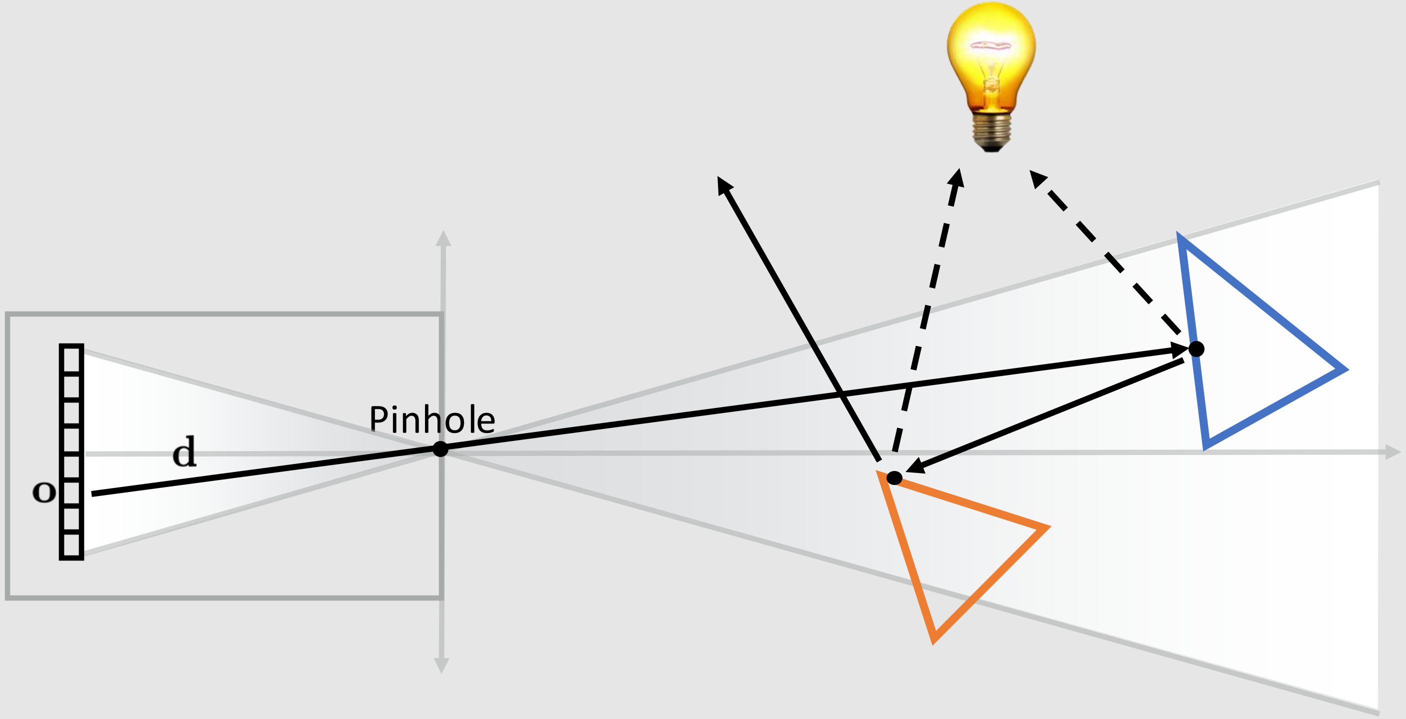
- Intersect △ , no emission □
- Intersect △ , no emission □
- Ray terminate, emission □

$$L(pixel) = L_o(ray_1)$$

$$L(pixel\,) = \square$$

Next Event Estimation (NEE)

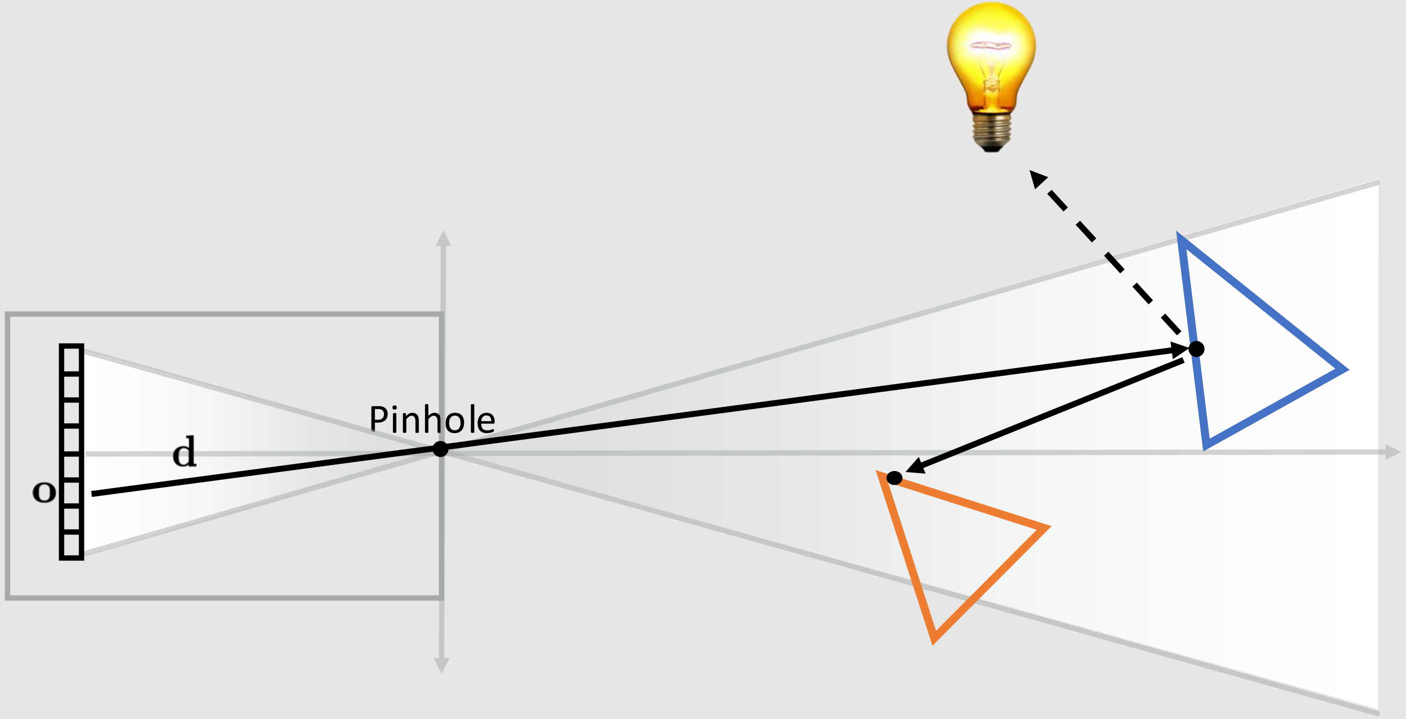
- Extension to Backwards Path Tracing
  - **At each ray bounce, trace two new rays:**
    - A ray generated by the BRDF
    - A ray towards the light
  - Average samples together
  - Can only be done for diffuse surfaces!
- No need to trace ray to light source explicitly on termination
  - Taken care of at each ray bounce
- **Issue:** requires a lot of ray traces!



Single Sample Importance Sampling

- Extension to Backwards Path Tracing
  - **At each ray bounce, pick one:**
    - A ray generated by the BRDF
    - A ray towards the light
  - Can only be done for diffuse surfaces!
  - Sample between rays with uniform probability

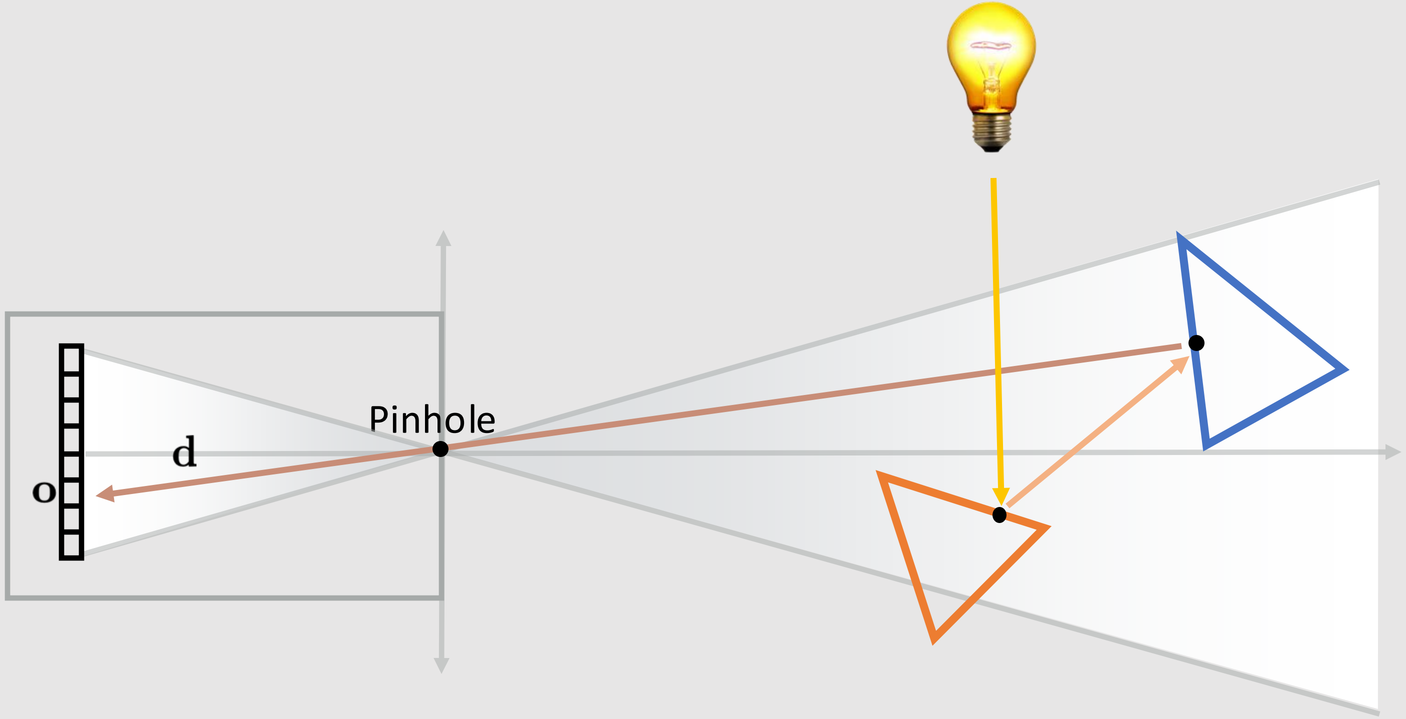
- You will implement this in Scotty3D



If we can connect the final ray to whatever our target is,
why can’t we just use Forward Path Tracing?

Problem With Forward Renderer

- **Terminating ray must go through pinhole!**
- Cannot chose which pixel sensor the light ray will hit
  - Leads to uneven distribution of light samples onto final image sensor
- Backwards Renderer allows us to generate even number of rays from sensor
  - Leads to higher-quality image



Side Note: Why Is Everything In Focus?

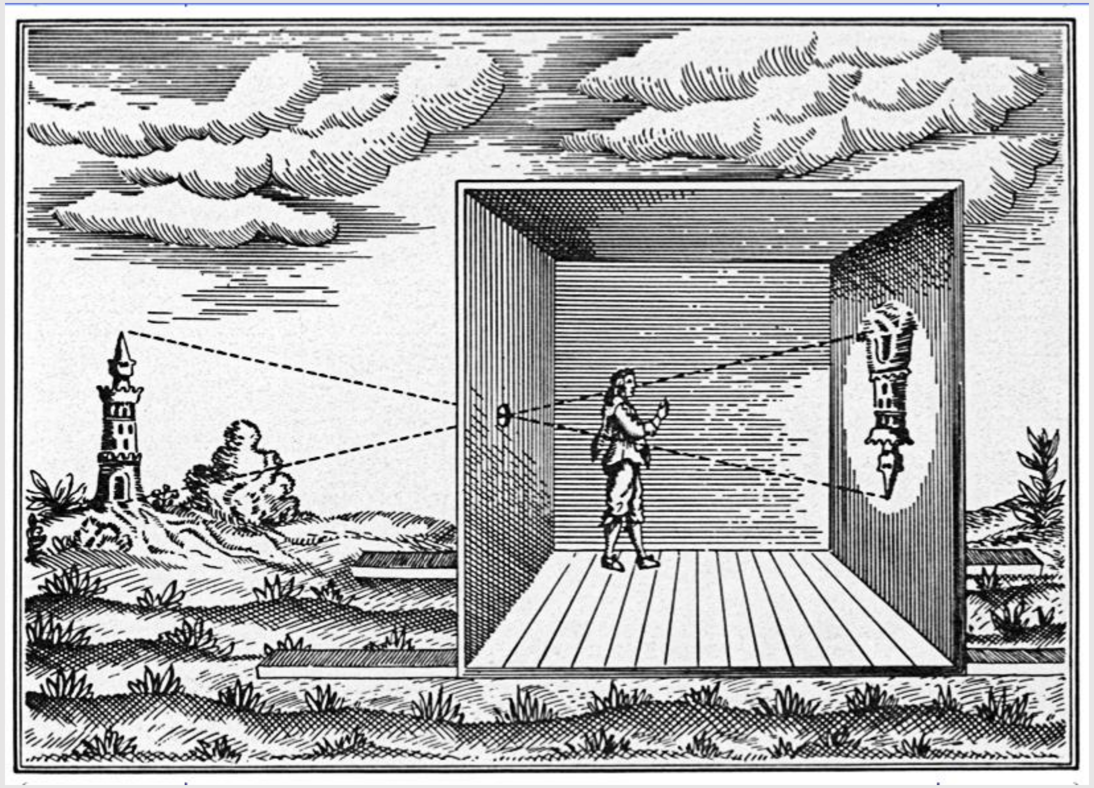
Cyberpunk 2077 (2020) CD Projekt

Side Note: Why Is Everything In Focus?

- When rendering, we can render everything clearly
  - No need to set focal distance
  - No blur like with real cameras

- Rendering uses pinhole cameras
  - Light isn't spread out across multiple sensors
  - Produces clear images everywhere

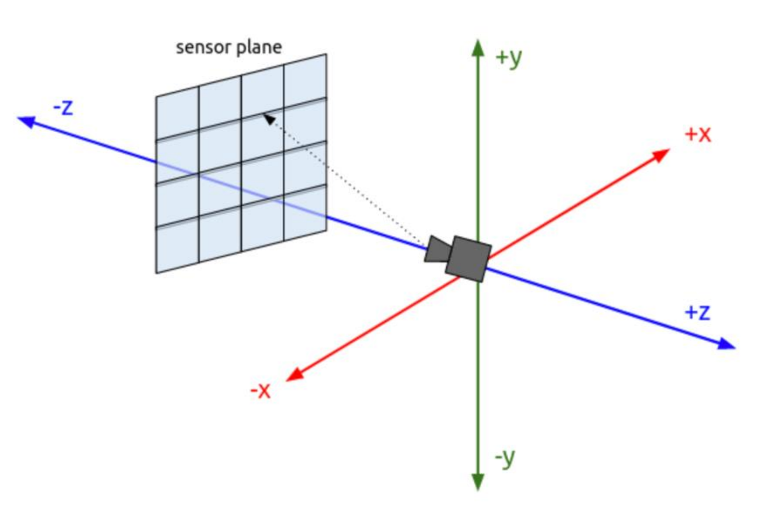
- Renderers can use pinhole, cameras cannot
  - Pinhole rendering takes in less light
    - Requires longer exposure
  - Render can freeze digital scene
  - Camera cannot freeze physical scene
    - Needs to increase aperture
    - Leads to blurring at different distances

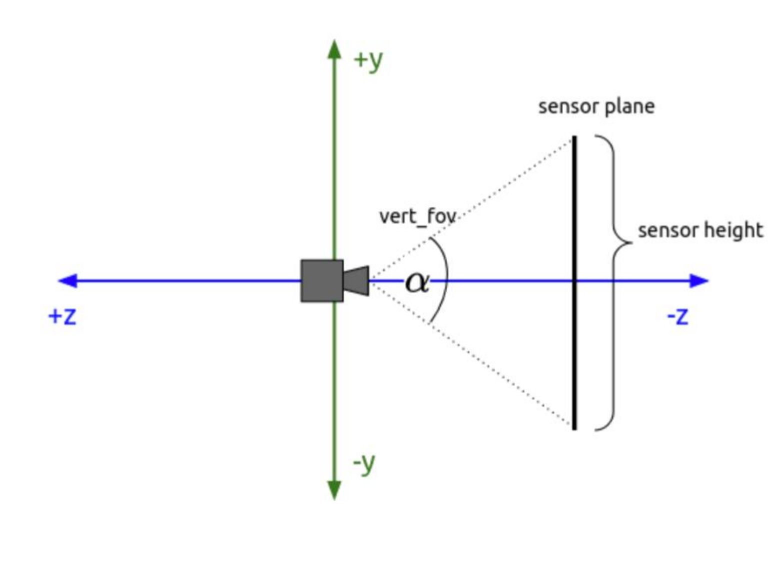


- ~~The Rendering Equation~~
- ~~A Simple Path Tracer~~
- Camera Rays

Camera Properties

- **Goal:** render an image of a given width and height
  - Think of the sensor image in front of the camera 1 unit away in the –z direction
- Construct rays from the camera origin to a point on the sensor
  - Where on the sensor depends on what sampling method
- Instead of with and height, we are given the **vertical field of view (vfov)** and **aspect ratio** of the sensor image
  - Vertical FOV measures how wide vertically the camera can see
  - Aspect ratio is the ratio of width/height



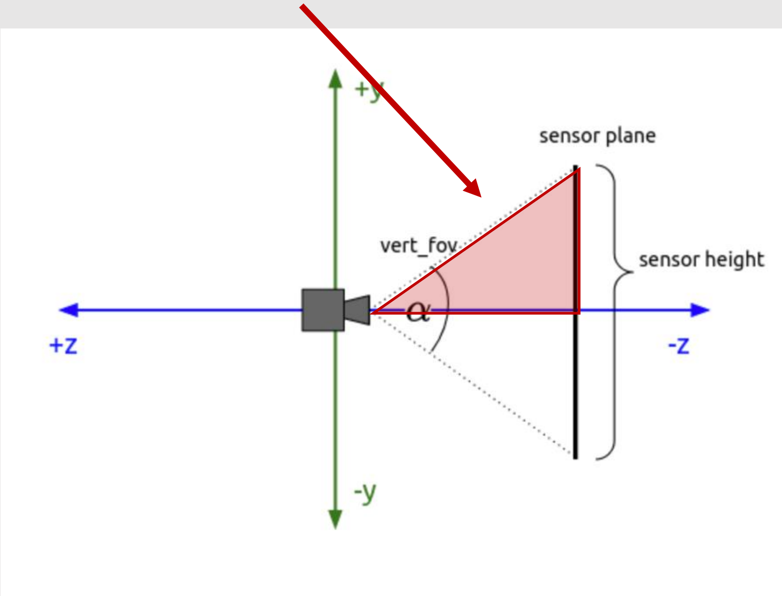


Generating Camera Rays

```
Ray Camera::sample_ray(rng, x, y)
{
  // create offset using rng
  Vec2 o = sample(rng);
  // normalize coordinates
  float xn = (x + o.x) / film.wth;
  float yn = (y + o.y) / film.hgt;

  // computing height is an exercise to reader
  float hgt = // TODO: some trig
  // aspect ratio tells us ratio of wth/hgt
  float wth = hgt * aspect_ratio;

  // convert to 2D sensor coordinates
  float x_cord = xn * wth;
  float y_cord = yn * hgt;

  // construct ray from camera origin to sensor
  // sensor is 1 unit away in -z dir
  Ray r(Vec3(), Vec3(x_cord, y_cord, -1.0f));
  return {r, pdf(o)};
}
```

- Solve for width and height
- Generate point on sensor plane using any sampler
  - In our example we use random sampling
- Build a ray from the camera to the sample point on the sensor

Triangle! Just use trig!



Supersampling Camera Rays

- Similar to rasterization, can trace multiple rays per pixel
  - Resolve samples by averaging
- Many different sampling methods to chose from:
  - Jittered Sampling
  - Multi-jittered sampling
  - N-Rooks sampling
  - Sobol sequence sampling
  - Halton sequence sampling
  - Hammersley sequence sampling
- Visualizer built in Scotty3D to see ray distribution