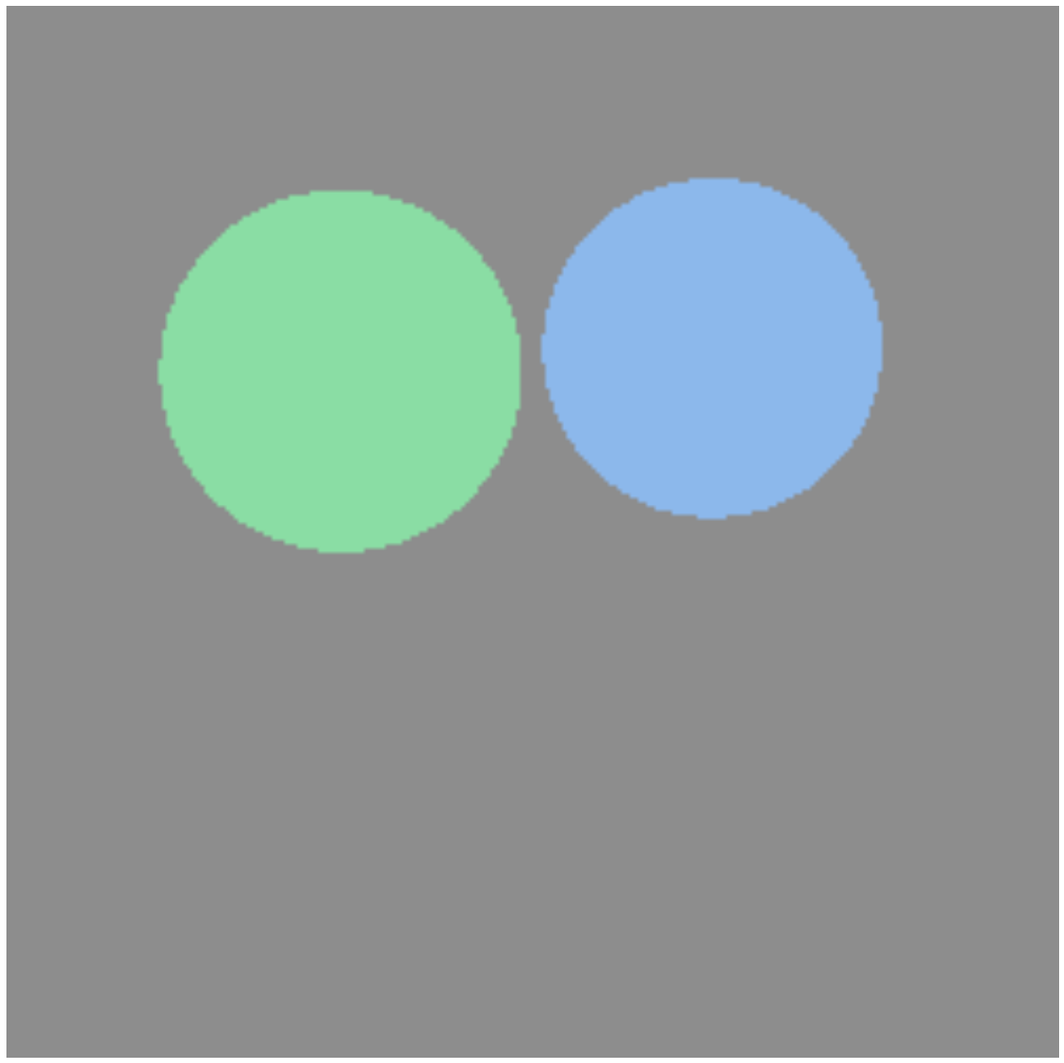
Ray Tracing: shading

CS 4620 Lecture 6

Image so far

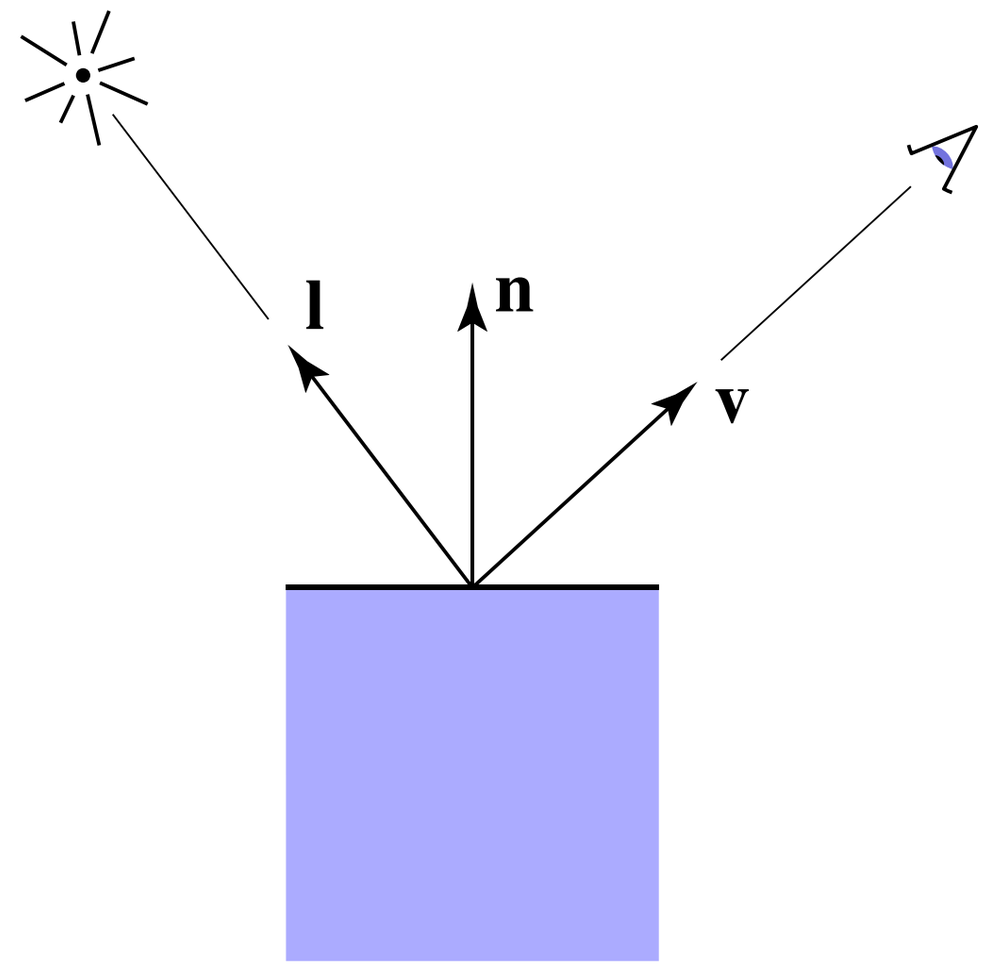
- **With eye ray generation and scene intersection**

```
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        c = scene.trace(ray, 0, +inf);
        image.set(ix, iy, c);
    }

...

Scene.trace(ray, tMin, tMax) {
    surface, t = surfs.intersect(ray, tMin, tMax);
    if (surface != null) return surface.color();
    else return black;
}
```



Shading

- **Compute light reflected toward camera**
- **Inputs:**
  - eye direction
  - light direction
    (for each of many lights)
  - surface normal
  - surface parameters
    (color, roughness, …)



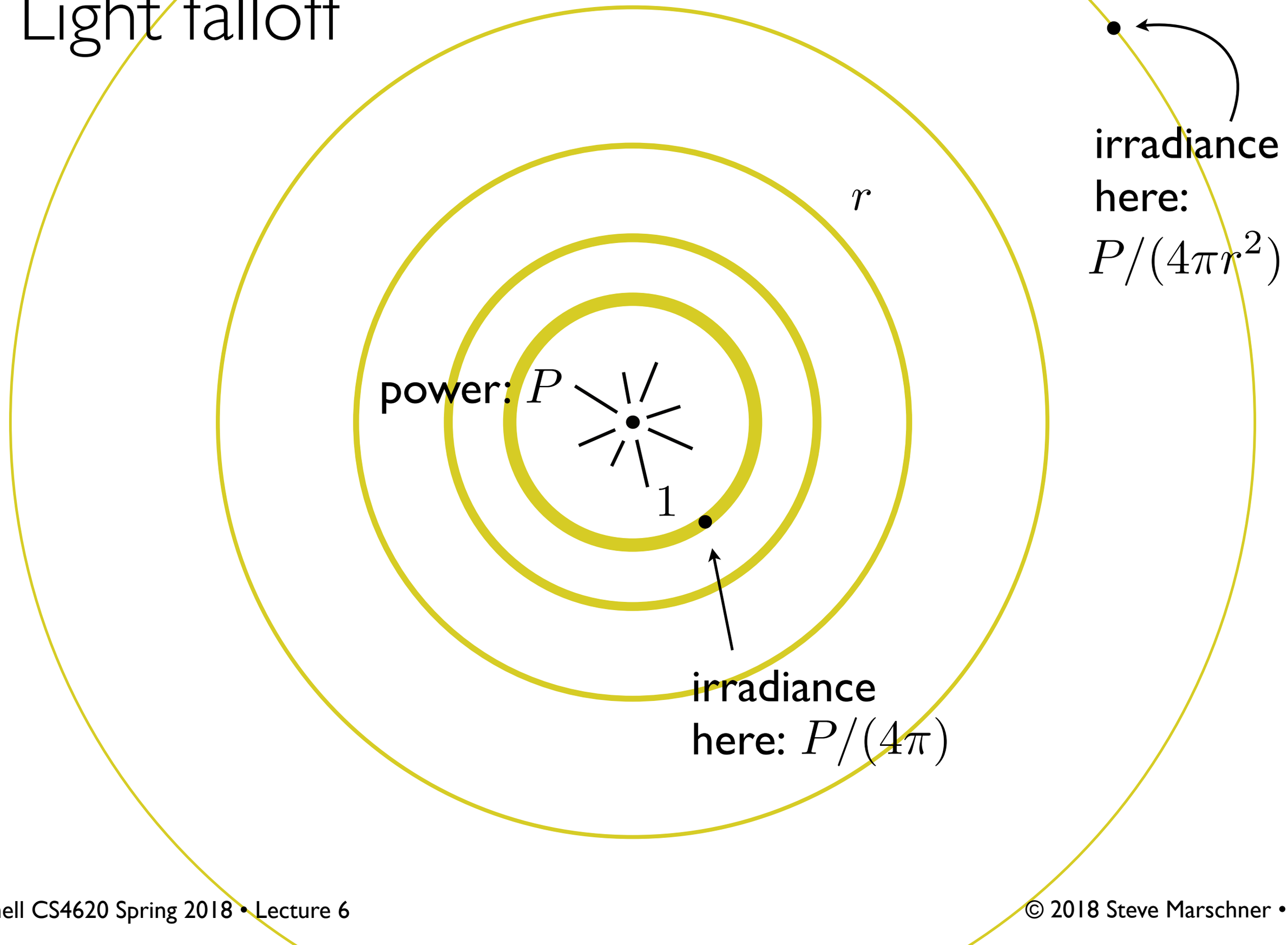
Shading philosophy

- **Goals of shading depend on purpose of image**
  - visualization, CAD: maximize visual clarity
  - visual effects, advertising: maximize resemblance to reality
  - animation, games: somewhere in between
- **Basic starting point: physics of light reflection**
  - a set of useful approximations to real surfaces
  - can remove things for simplicity/clarity
  - can add things for increased accuracy/realism

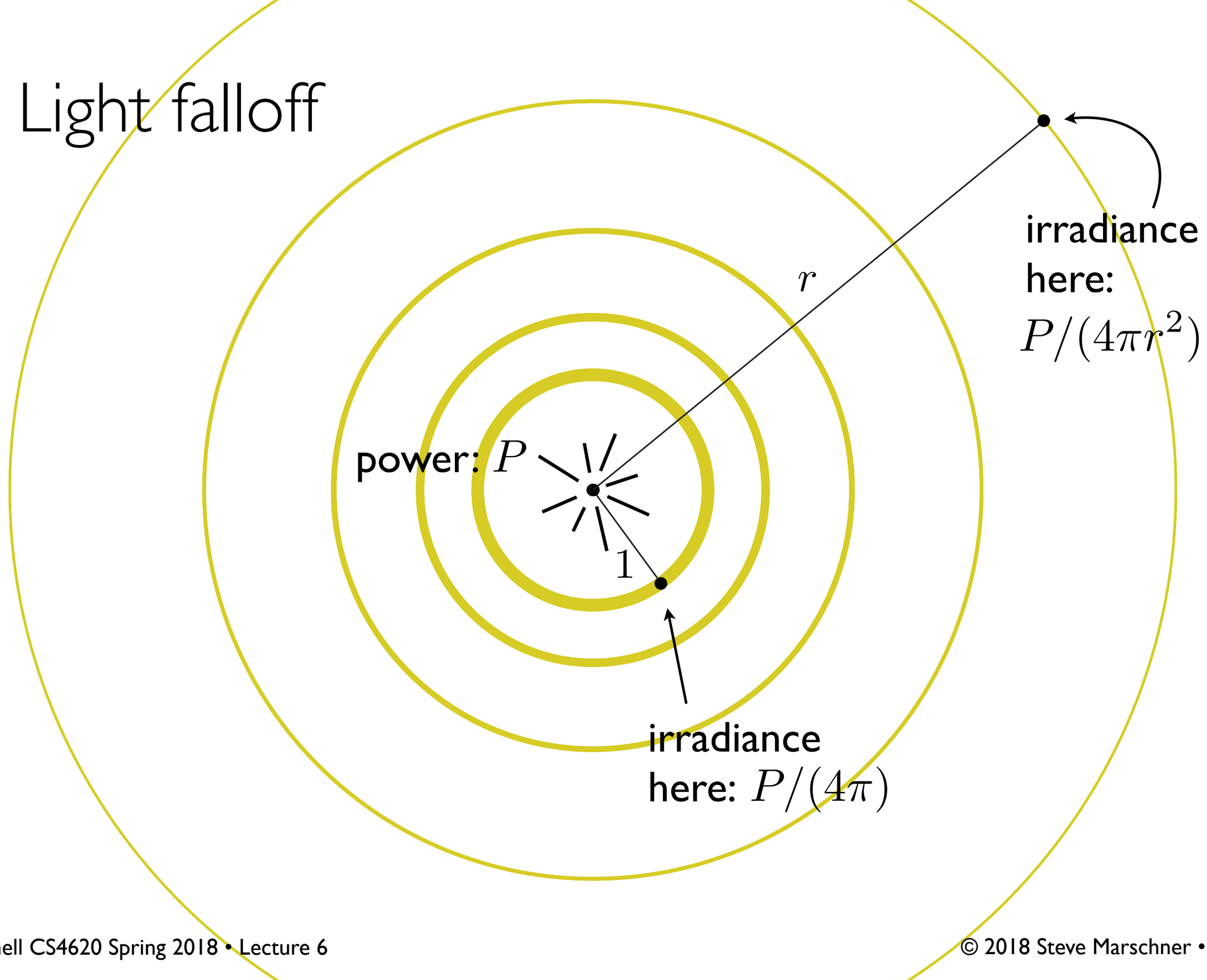
Light

- **Think of light as a flow of particles through space**
  - disregarding wave nature: polarization, interference, diffraction
  - for now disregarding color: only how much light
- **Sources of light**
  - point sources (a flashlight) ← we will stick to this for now.
  - directional sources (the sun)
  - area sources (a fluorescent tube)
  - environment sources (the sky)

Light falloff



Light falloff



Irradiance from isotropic point source

- **A sphere surrounding the source receives all the power**
- **A small, flat surface of area *A* facing the source receives a fraction (area of surface) / (area of sphere) of that power:**

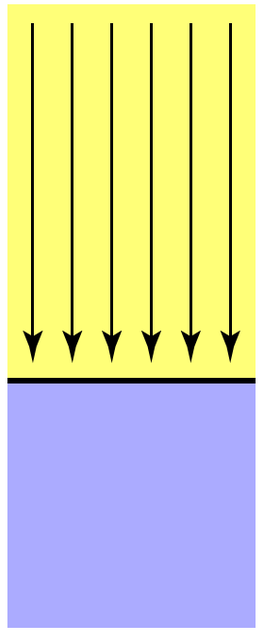
$$P_A = P\frac{A}{4\pi r^2}$$

- **Irradiance is power per unit area:**

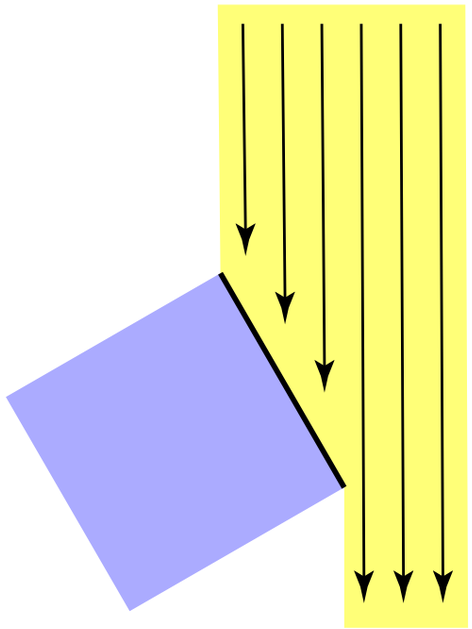
$$E = P_A/A = \frac{P}{4\pi r^2} = \frac{P}{4\pi}\frac{1}{r^2}$$

intensity geometry factor

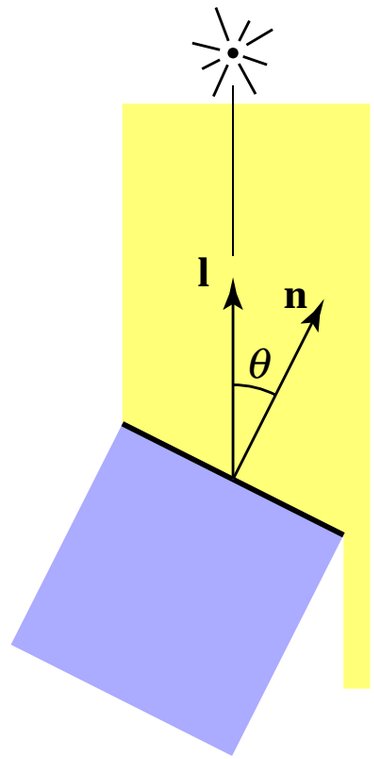
Lambert's cosine law



Top face of cube receives a certain amount of light



Top face of 60° rotated cube intercepts half the light



In general, light per unit area is proportional to $\cos\theta = \mathbf{l} \cdot \mathbf{n}$

Irradiance from isotropic point source

- **A surface of area *A* facing at an angle to the source receives a factor of $\cos\theta$ less light:**

$$P_A = P\frac{A\cos\theta}{4\pi r^2}$$

- **Irradiance is power per unit area:**

$$E = P_A/A = \frac{P}{4\pi}\frac{\cos\theta}{r^2}$$

intensity geometry factor

Diffuse reflection

- **Simplest reflection model**
- **Reflected light is independent of view direction**
- **Reflected light is proportional to irradiance**
  - constant of proportionality is the diffuse reflection coefficient

$$L_d = k_d E$$

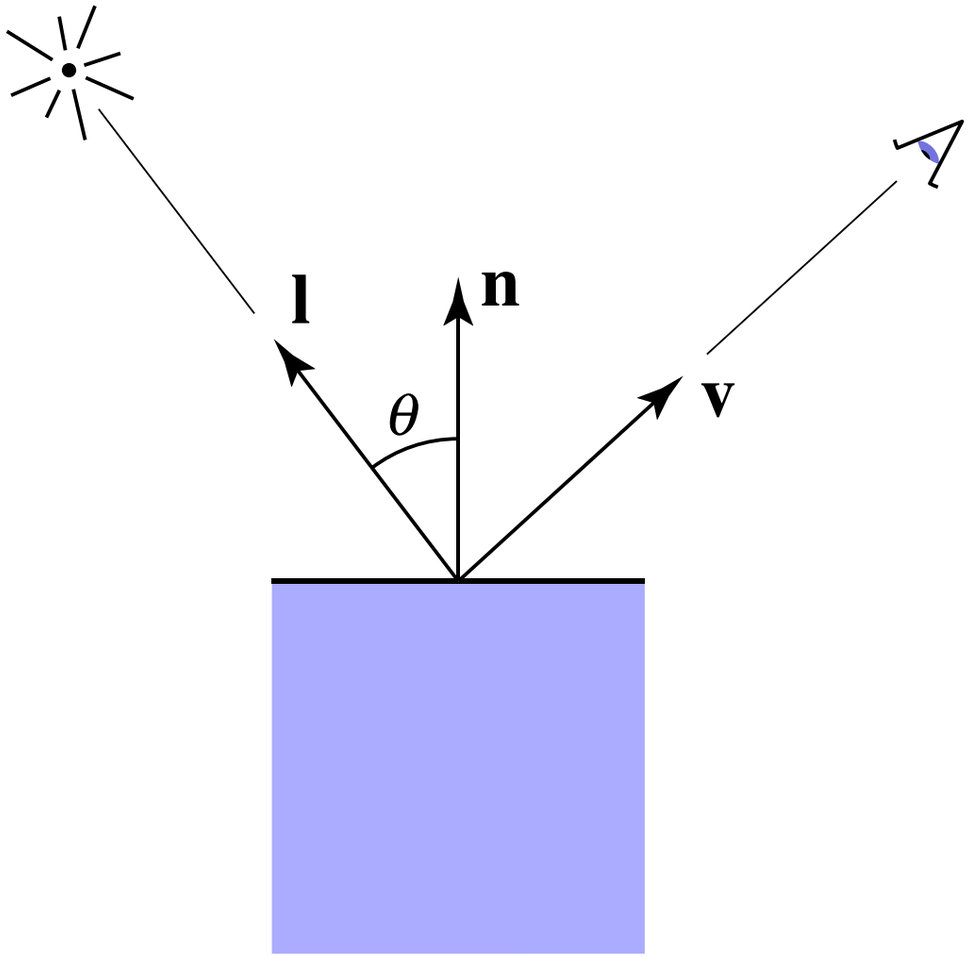
- **More useful to think in terms of reflectance**
  - reflectance is the fraction reflected (between 0 and 1)

$$L_d = \frac{R_d}{\pi} E$$

  - will have to explain the factor of pi later

Lambertian shading

- **Shading independent of view direction**



diffuse reflectance

irradiance from source

$$L_d = \frac{R}{\pi} \frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2} I$$

diffusely reflected radiance

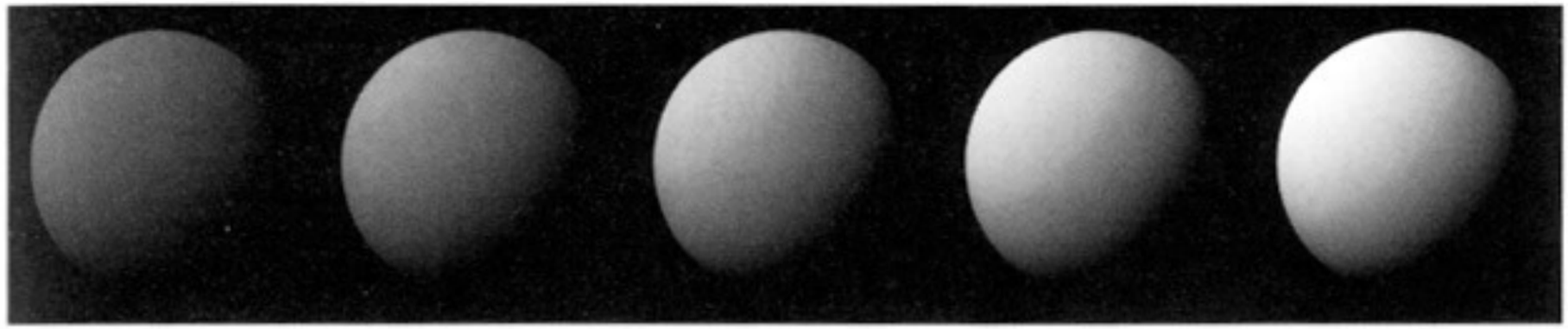
diffuse coefficient

distance to source

intensity of source

Lambertian shading

- **Produces matte appearance**



$k_d \longrightarrow$

[Foley et al.]

Diffuse shading

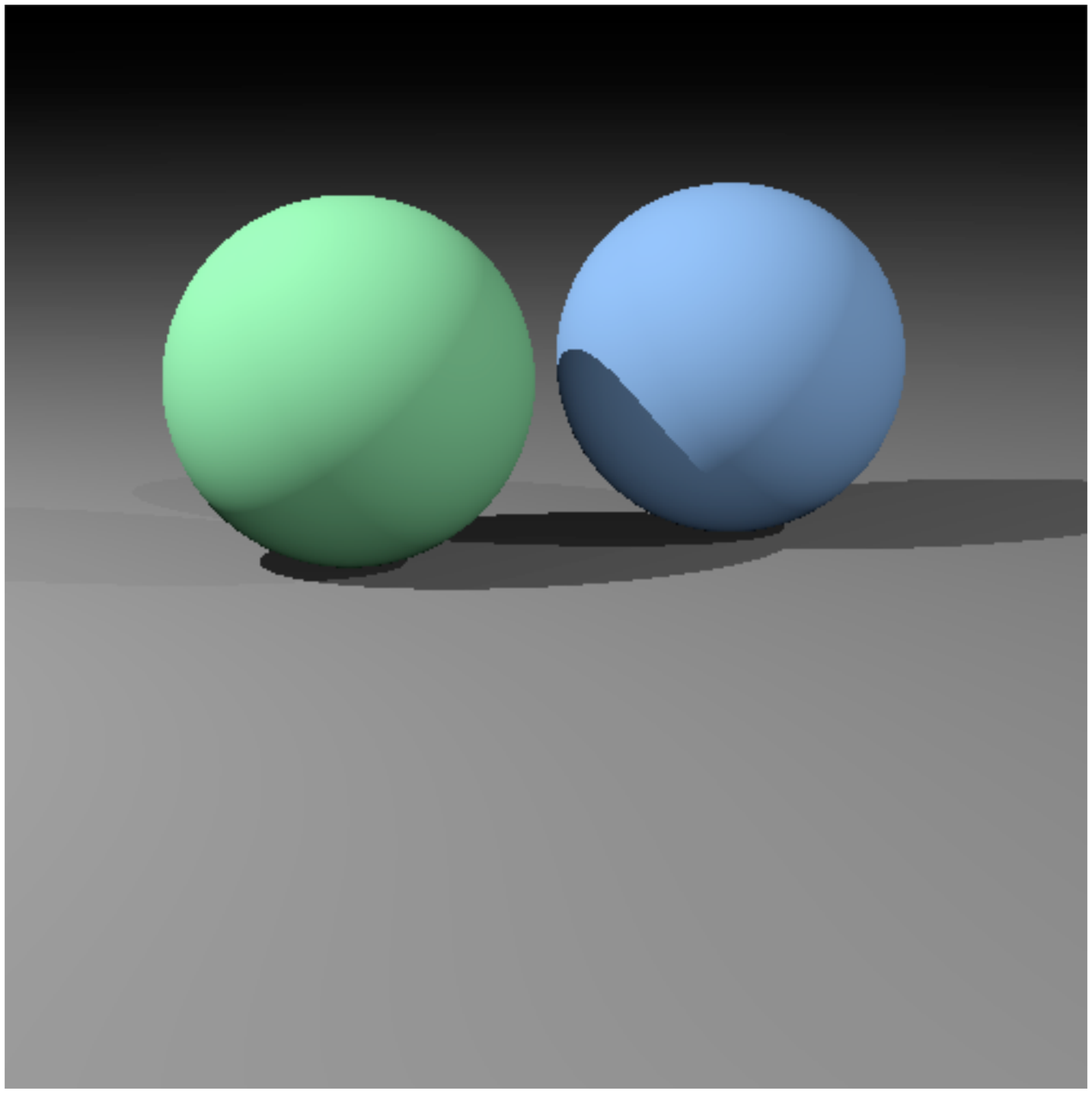
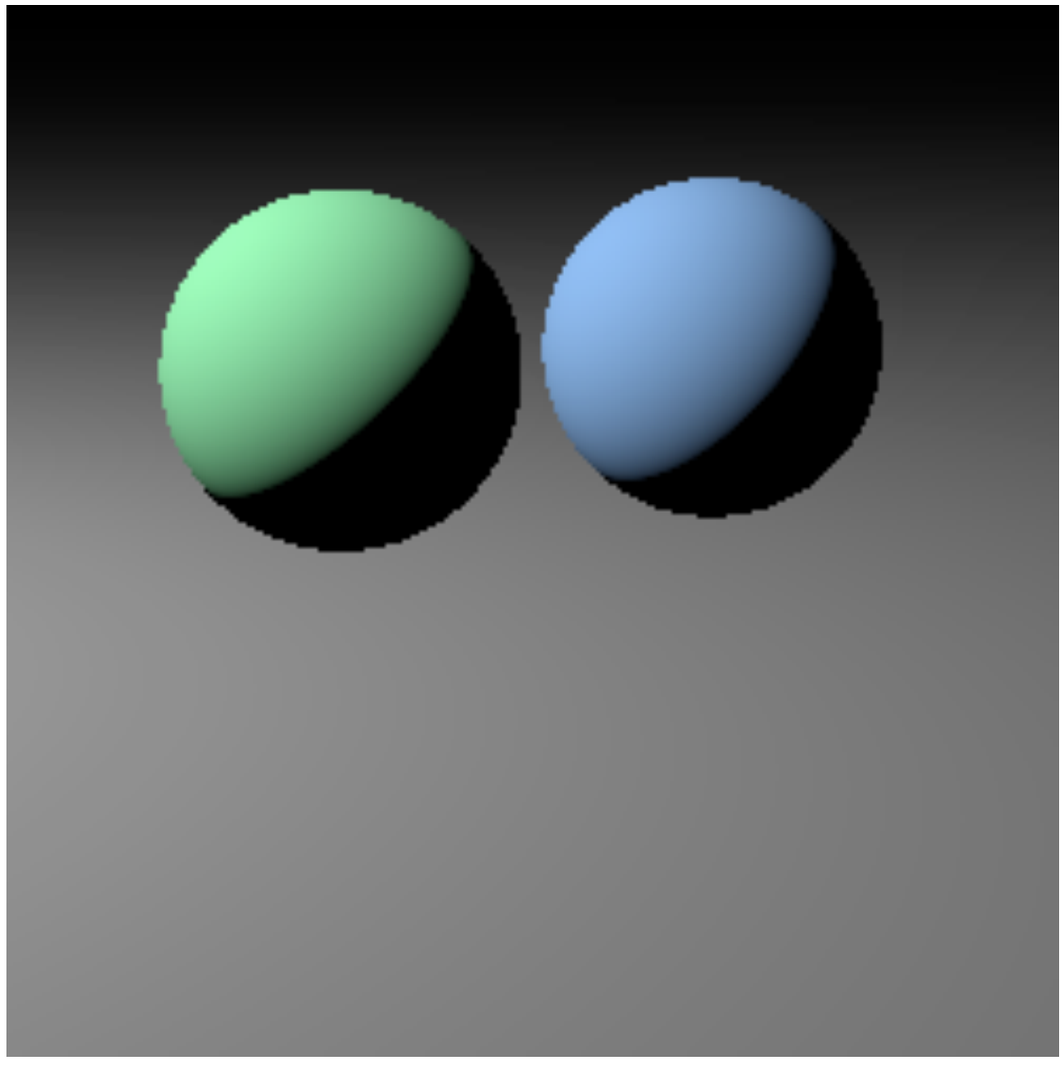


Image so far

```
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return backgroundColor;
}

...

Surface.shade(ray, point, normal, light) {
    v = −normalize(ray.direction);
    l = normalize(light.pos − point);
    // compute shading
}
```

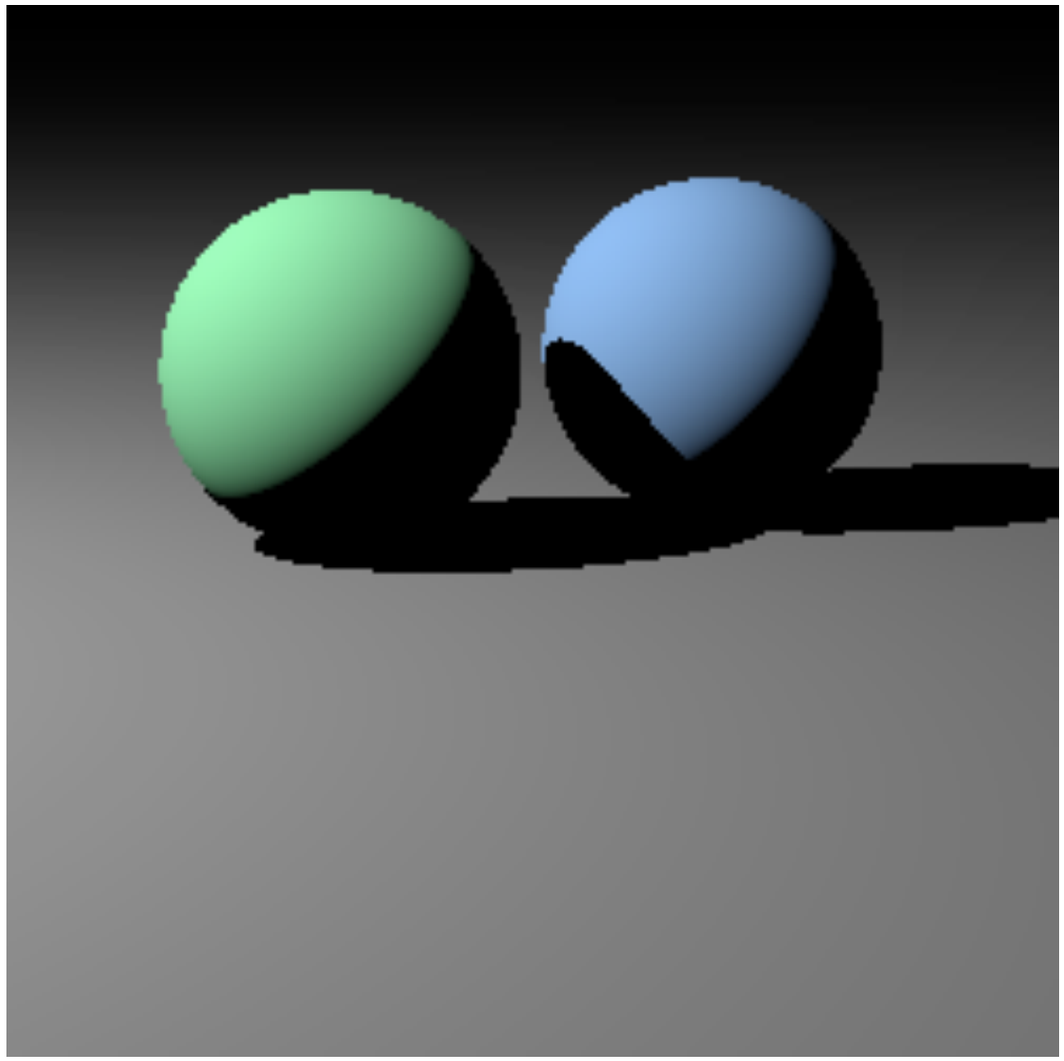


Shadows

- **Surface is only illuminated if nothing blocks the light**
  - i.e. if the surface can "see" the light
- **With ray tracing it's easy to check**
  - just intersect a ray with the scene!

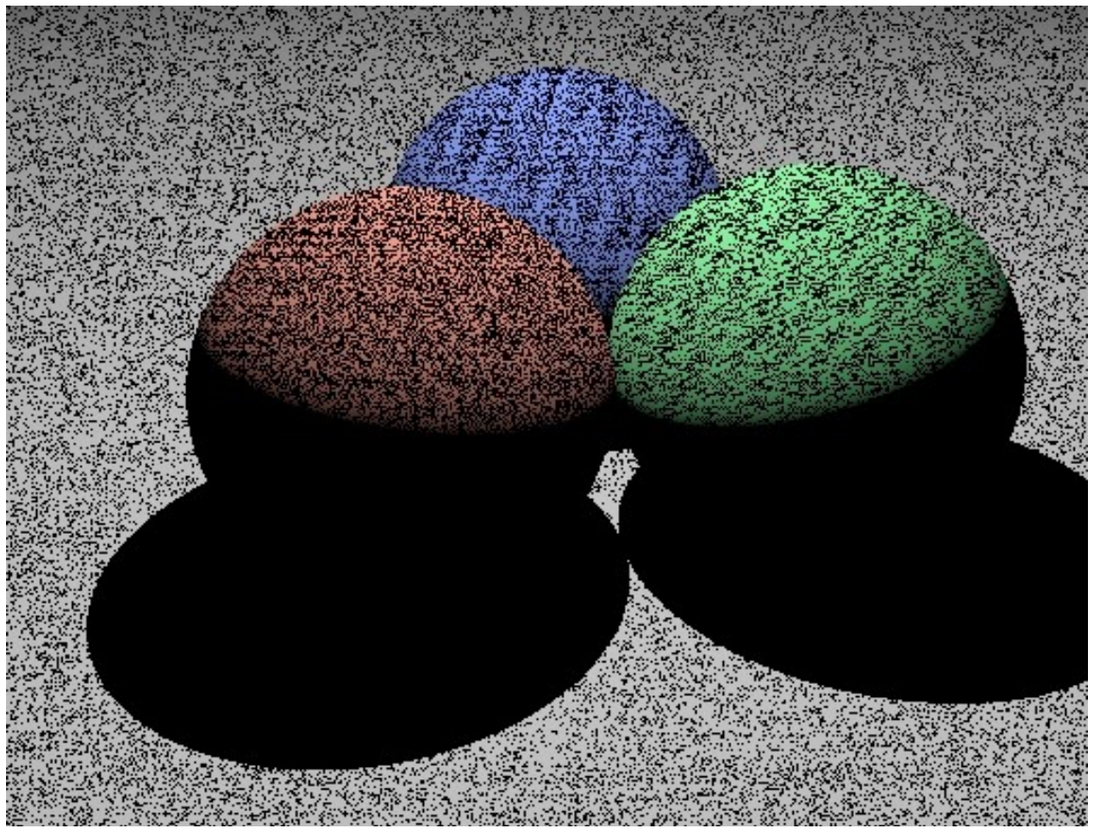
Image so far

```
Surface.shade(ray, point, normal, light) {
    shadRay = (point, light.pos − point);
    if (shadRay not blocked) {
        v = −normalize(ray.direction);
        l = normalize(light.pos − point);
        // compute shading
    }
    return black;
}
```



Shadow rounding errors

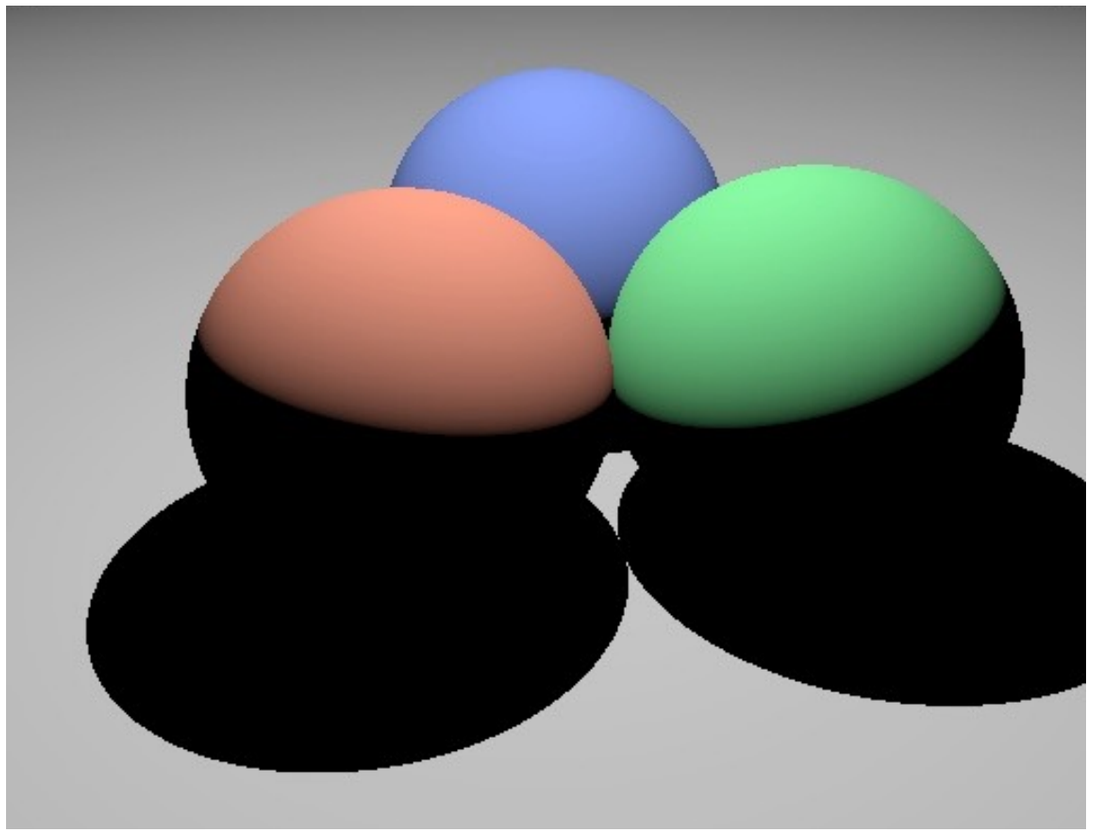
- **Don’t fall victim to one of the classic blunders:**



- **What’s going on?**
  - hint: at what ***t*** does the shadow ray intersect the surface you’re shading?

Shadow rounding errors

- **Solution: shadow rays start a tiny distance from the surface**



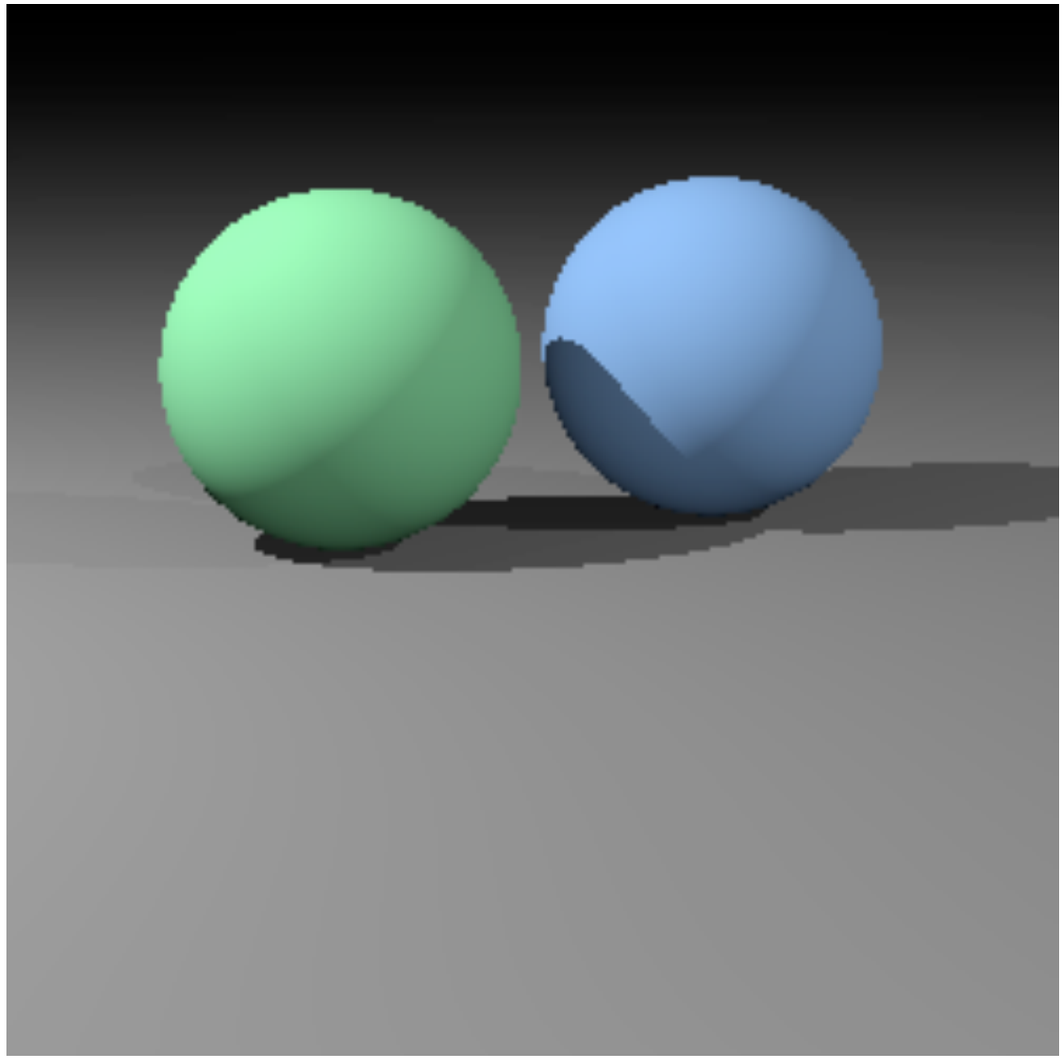
- **Do this by moving the start point, or by limiting the *t* range**

Multiple lights

- **Important to fill in black shadows**
- **Just loop over lights, add contributions**
- **Ambient shading**
  - black shadows are not really right
  - one solution: dim light at camera
  - alternative: add a constant "ambient" color to the shading…

Image so far

```
shade(ray, point, normal, lights) {
    result = ambient;
    for light in lights {
        if (shadow ray not blocked) {
            result += shading contribution;
        }
    }
    return result;
}
```



Specular shading

(under construction)

Ray tracer architecture 101

- **You want a class called Ray**
  - point and direction; evaluate(t)
  - possible: tMin, tMax
- **Some things can be intersected with rays**
  - individual surfaces
  - groups of surfaces (acceleration goes here)
  - the whole scene
  - make these all subclasses of Surface
  - limit the range of valid *t* values (e.g. shadow rays)
- **Once you have the visible intersection, compute the color**
  - may want to separate shading code from geometry
  - separate class: Material (each Surface holds a reference to one)
  - its job is to compute the color

Architectural practicalities

- **Return values**
  - surface intersection tends to want to return multiple values
    - *t*, surface or shader, normal vector, maybe surface point
  - in many programming languages (e.g. Java) this is a pain
  - typical solution: an *intersection record*
    - a class with fields for all these things
    - keep track of the intersection record for the closest intersection
    - be careful of accidental aliasing (which is very easy if you're new to Java)
- **Efficiency**
  - in Java the (or, a) key to being fast is to minimize creation of objects
  - what objects are created for every ray? try to find a place for them where you can reuse them.
  - Shadow rays can be cheaper (any intersection will do, don't need closest)
  - but: "First Get it Right, Then Make it Fast"

