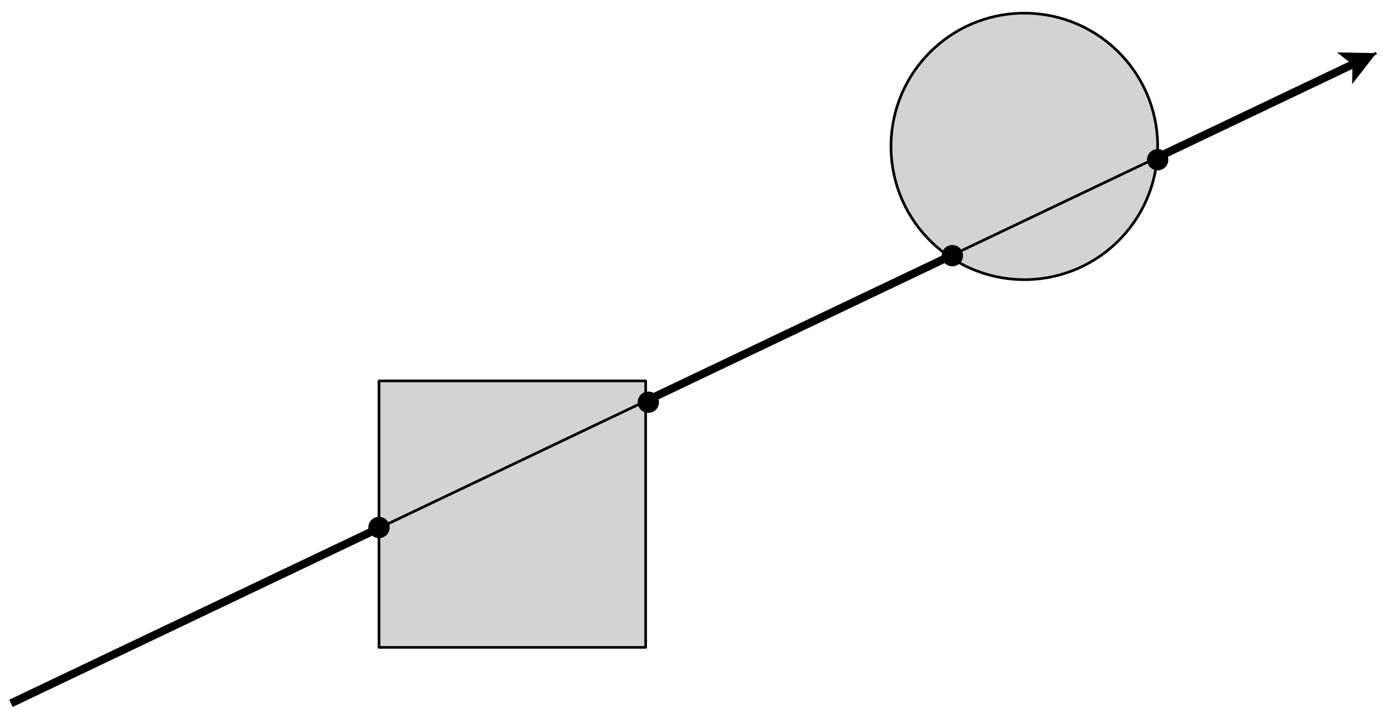
Ray Tracing: intersection and shading

CS 4620 Lecture 3

Ray intersection

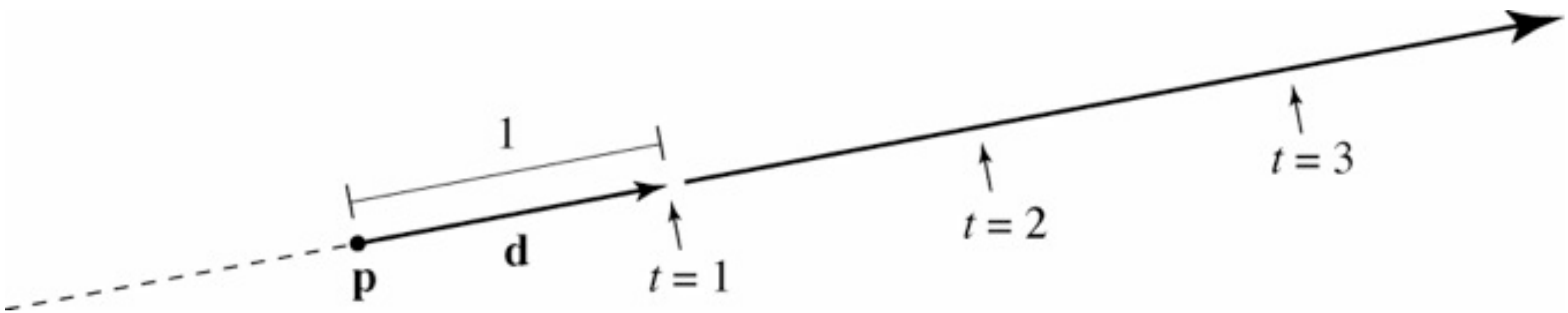


Ray: a half line

- Standard representation: point **p** and direction **d**

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

  – this is a *parametric equation* for the line
  – lets us directly generate the points on the line
  – if we restrict to $t > 0$ then we have a ray
  – note replacing **d** with a**d** doesn't change ray ($a > 0$)



Ray-sphere intersection: algebraic

- Condition 1: point is on ray

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Condition 2: point is on sphere
  - assume unit sphere; see Shirley or notes for general

$$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Substitute:

$$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - 1 = 0$$

  - this is a quadratic equation in *t*

Ray-sphere intersection: algebraic

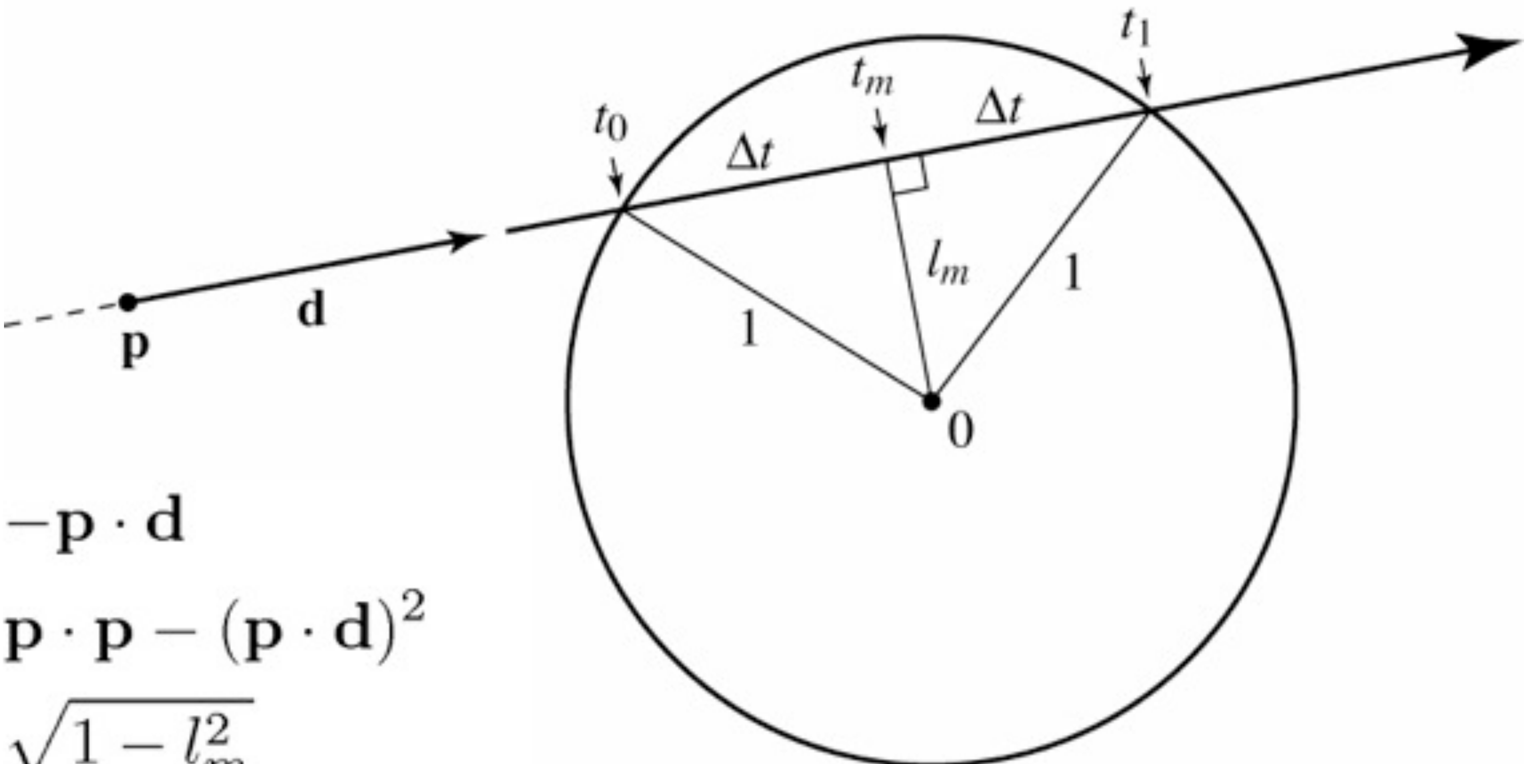
- Solution for *t* by quadratic formula:

$$t = \frac{-\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - (\mathbf{d} \cdot \mathbf{d})(\mathbf{p} \cdot \mathbf{p} - 1)}}{\mathbf{d} \cdot \mathbf{d}}$$

$$t = -\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

  - simpler form holds when **d** is a unit vector
    but we won't assume this in practice (reason later)
  - I'll use the unit-vector form to make the geometric interpretation

Ray-sphere intersection: geometric



$$t_m = -\mathbf{p} \cdot \mathbf{d}$$

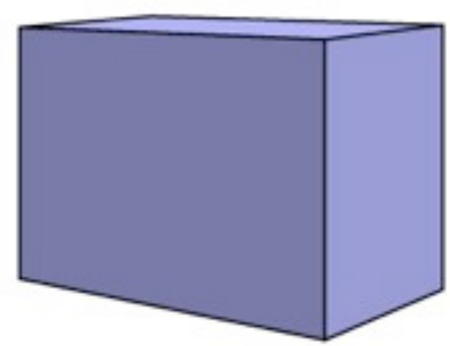
$$l_m^2 = \mathbf{p} \cdot \mathbf{p} - (\mathbf{p} \cdot \mathbf{d})^2$$

$$\begin{aligned} \Delta t &= \sqrt{1 - l_m^2} \\ &= \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1} \end{aligned}$$

$$t_{0,1} = t_m \pm \Delta t = -\mathbf{p} \cdot \mathbf{d} \pm \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

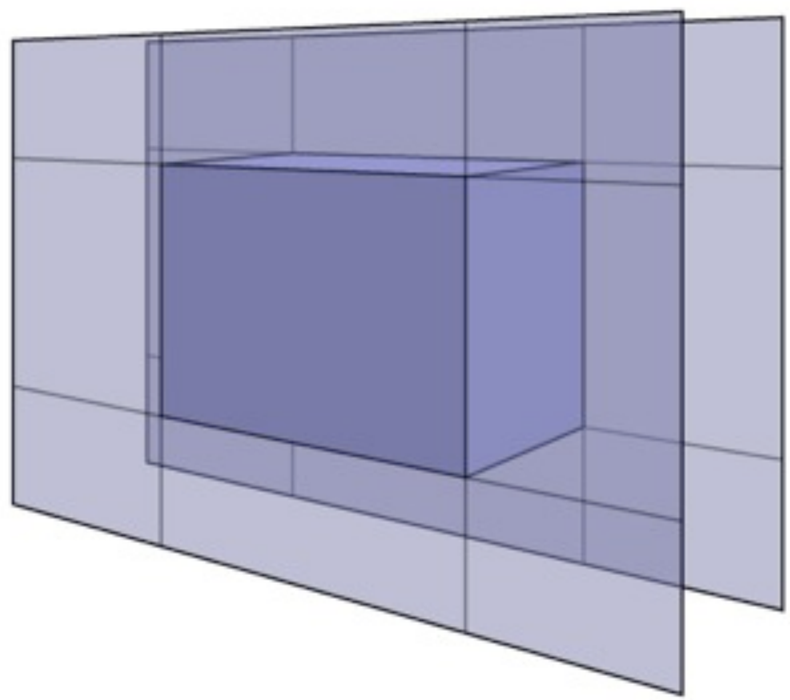
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



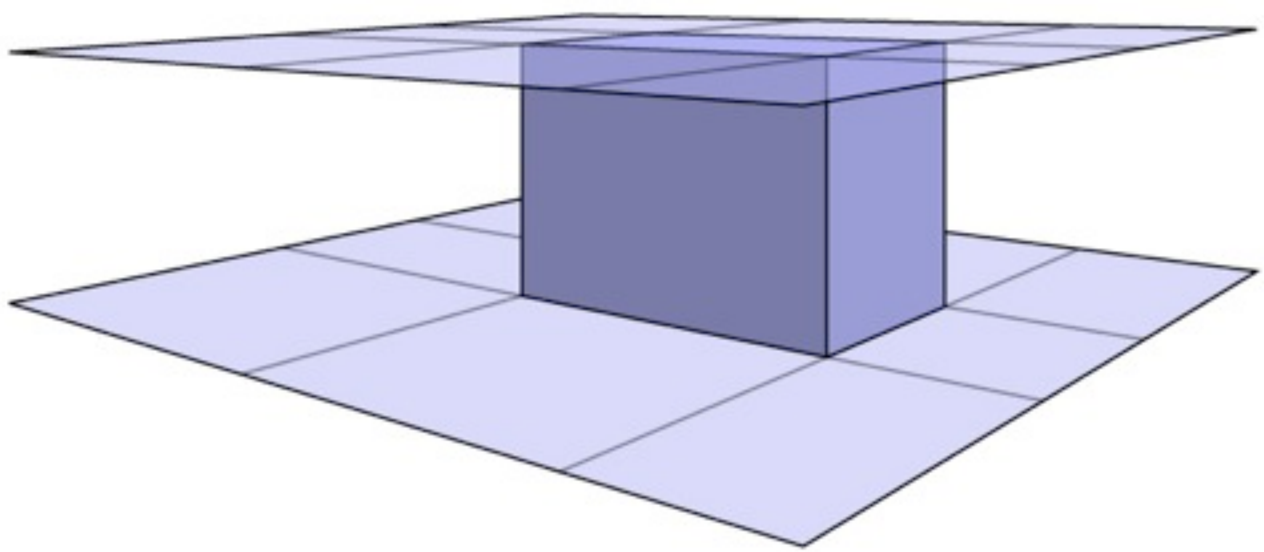
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



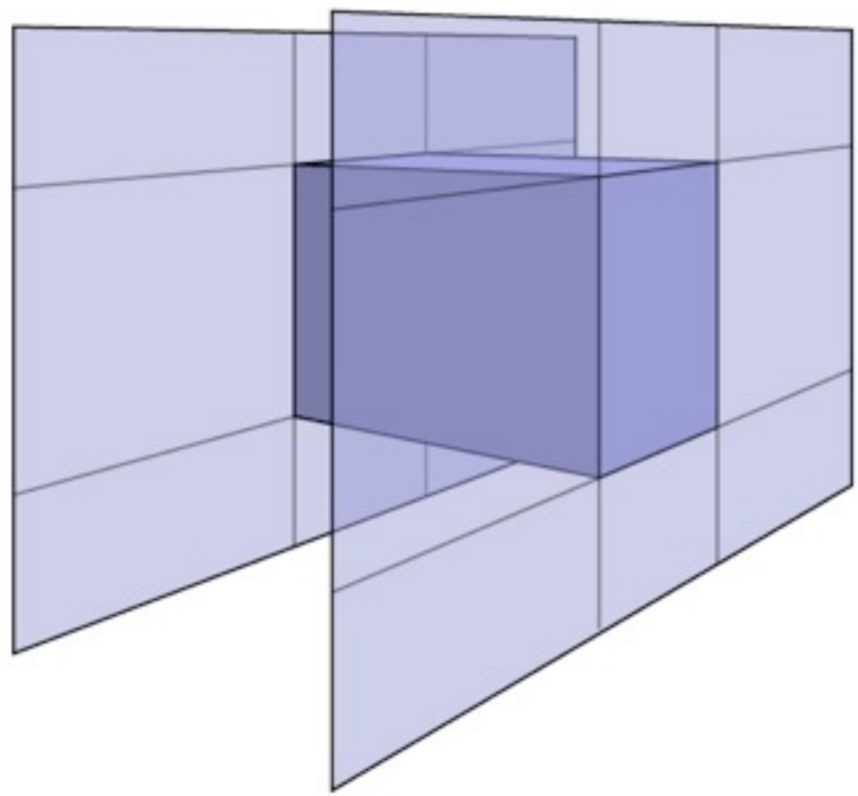
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



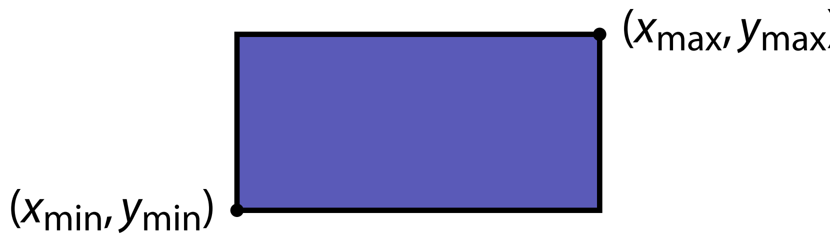
Ray-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



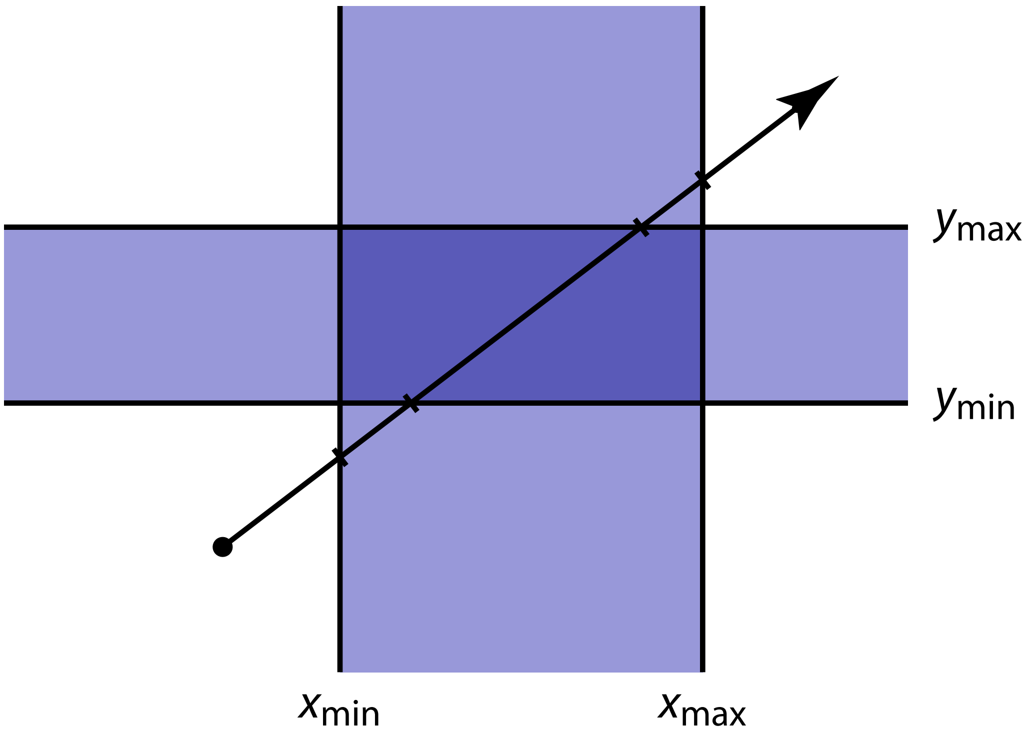
Ray-slab intersection

- 2D example
- 3D is the same!



Ray-slab intersection

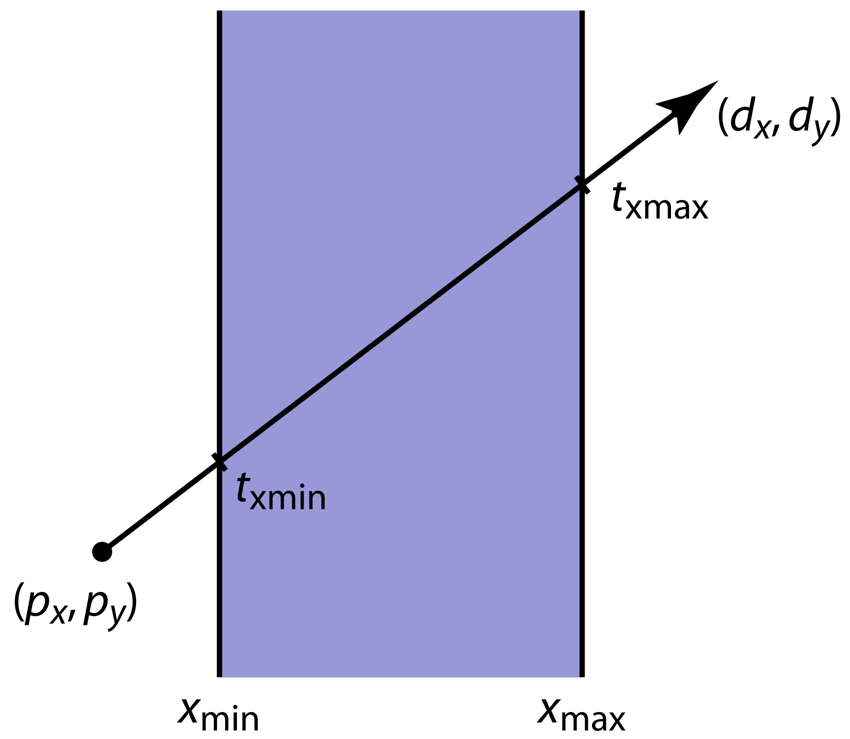
- 2D example
- 3D is the same!



Ray-slab intersection

- 2D example
- 3D is the same!

$$p_x + t_{x\min} d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

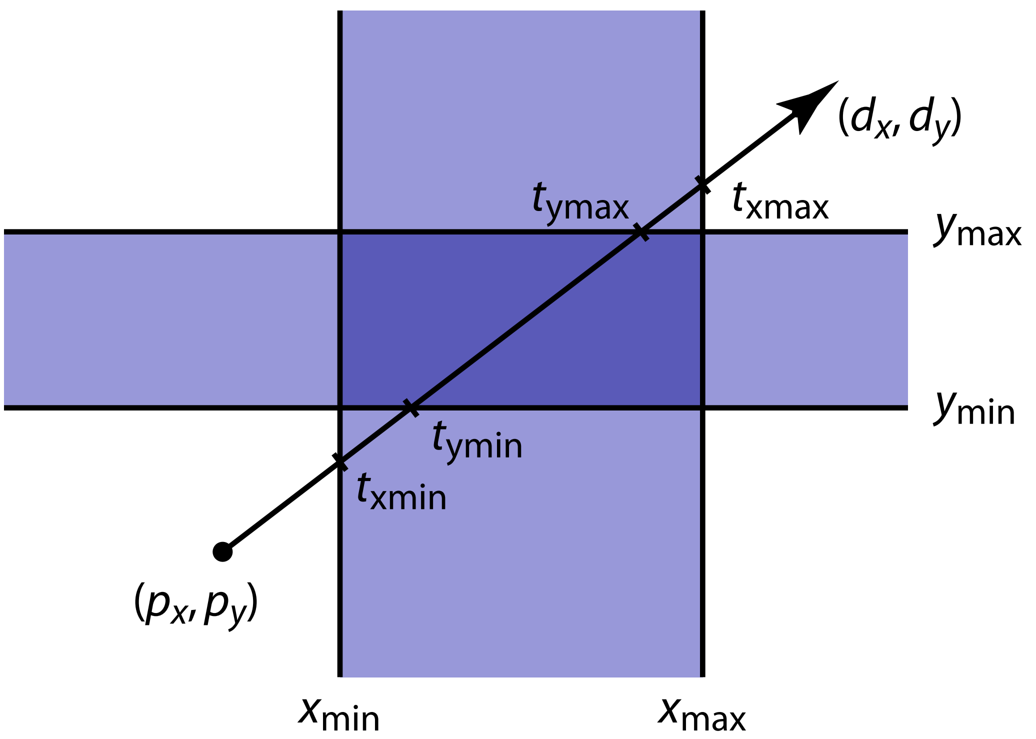


Ray-slab intersection

- 2D example
- 3D is the same!

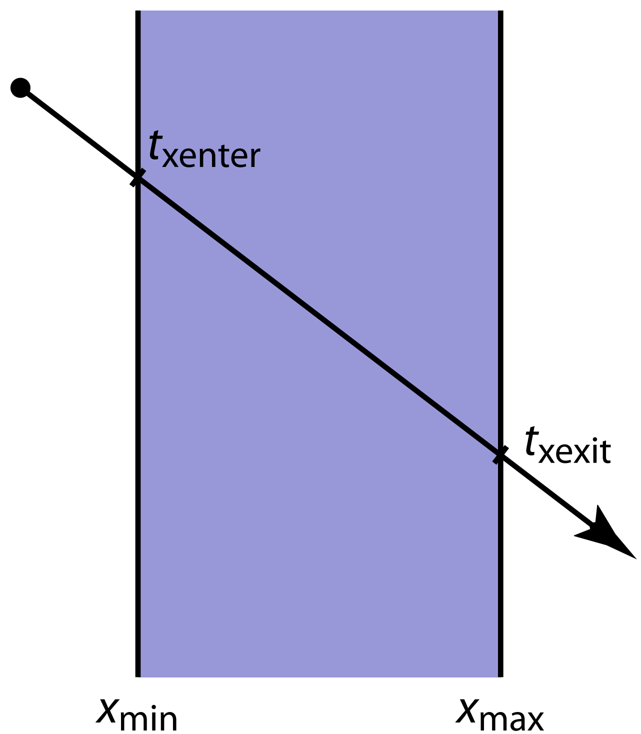
$$p_x + t_{x\mathrm{min}}\, d_x = x_{\mathrm{min}}$$
$$t_{x\mathrm{min}} = (x_{\mathrm{min}} - p_x)/d_x$$

$$p_y + t_{y\mathrm{min}}\, d_y = y_{\mathrm{min}}$$
$$t_{y\mathrm{min}} = (y_{\mathrm{min}} - p_y)/d_y$$



Intersecting intersections

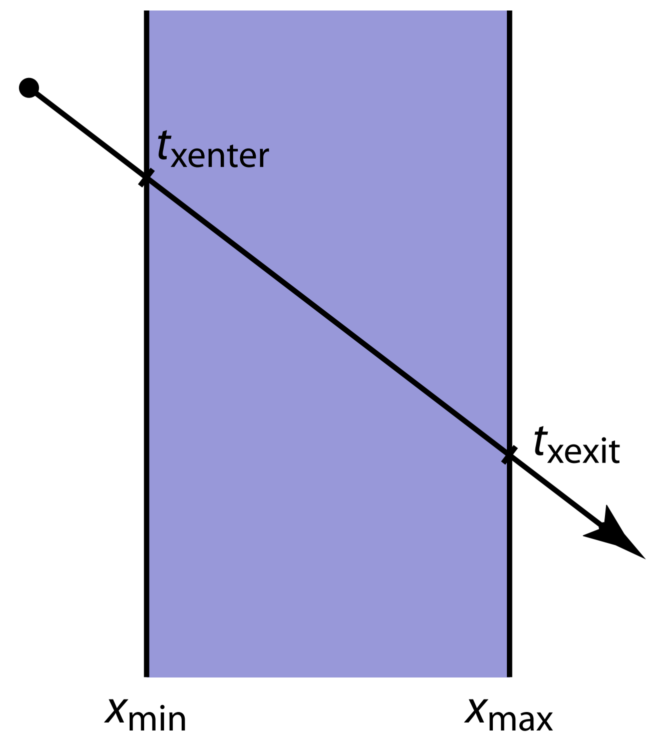
- Each intersection is an interval
- Want last entry point and first exit point



Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

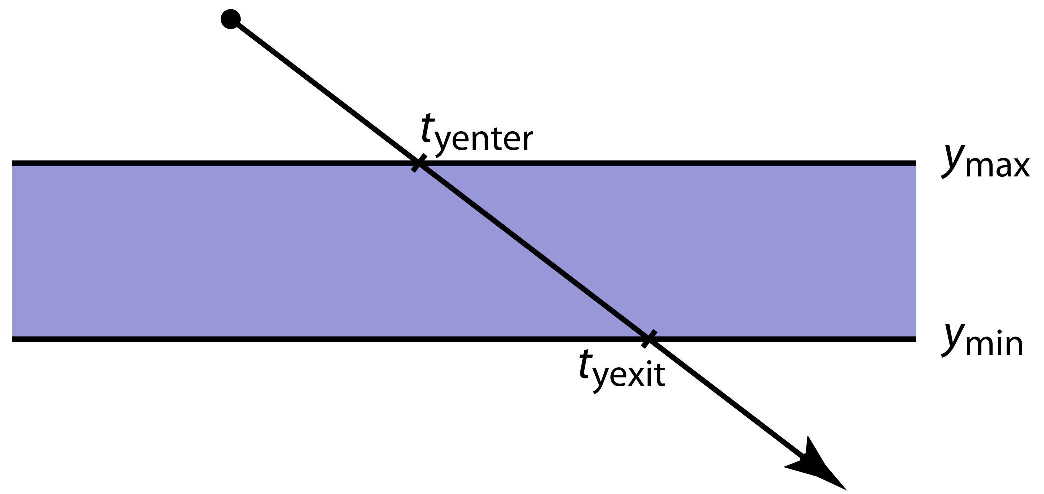
$$t_{x\text{enter}} = \min(t_{x\min}, t_{x\max})$$
$$t_{x\text{exit}} = \max(t_{x\min}, t_{x\max})$$



Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\min}, t_{x\max})$$
$$t_{x\text{exit}} = \max(t_{x\min}, t_{x\max})$$

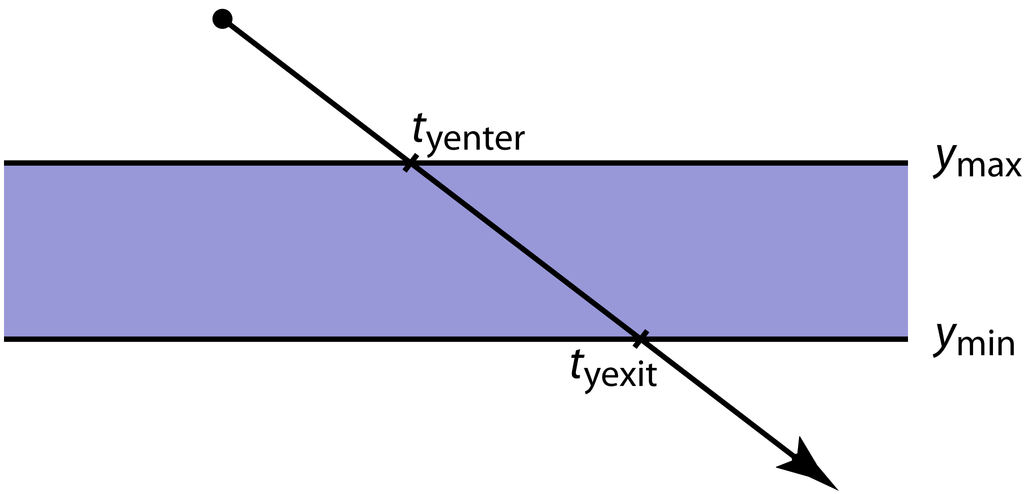


Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\text{min}}, t_{x\text{max}})$$
$$t_{x\text{exit}} = \max(t_{x\text{min}}, t_{x\text{max}})$$

$$t_{y\text{enter}} = \min(t_{y\text{min}}, t_{y\text{max}})$$
$$t_{y\text{exit}} = \max(t_{y\text{min}}, t_{y\text{max}})$$

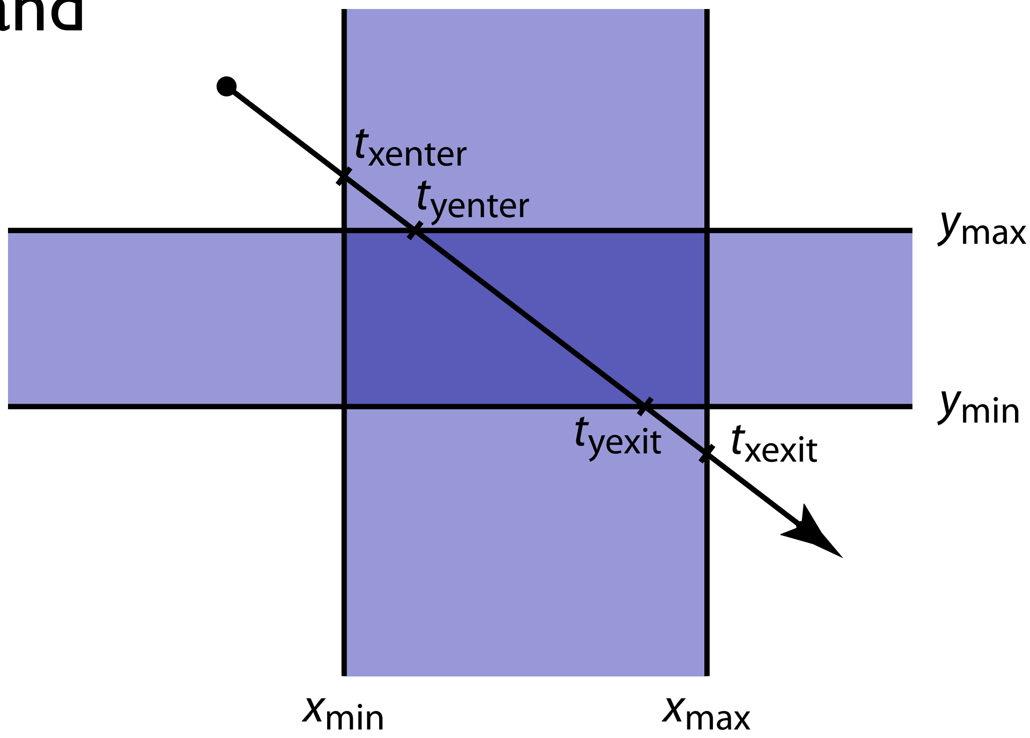


Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\min}, t_{x\max})$$
$$t_{x\text{exit}} = \max(t_{x\min}, t_{x\max})$$

$$t_{y\text{enter}} = \min(t_{y\min}, t_{y\max})$$
$$t_{y\text{exit}} = \max(t_{y\min}, t_{y\max})$$



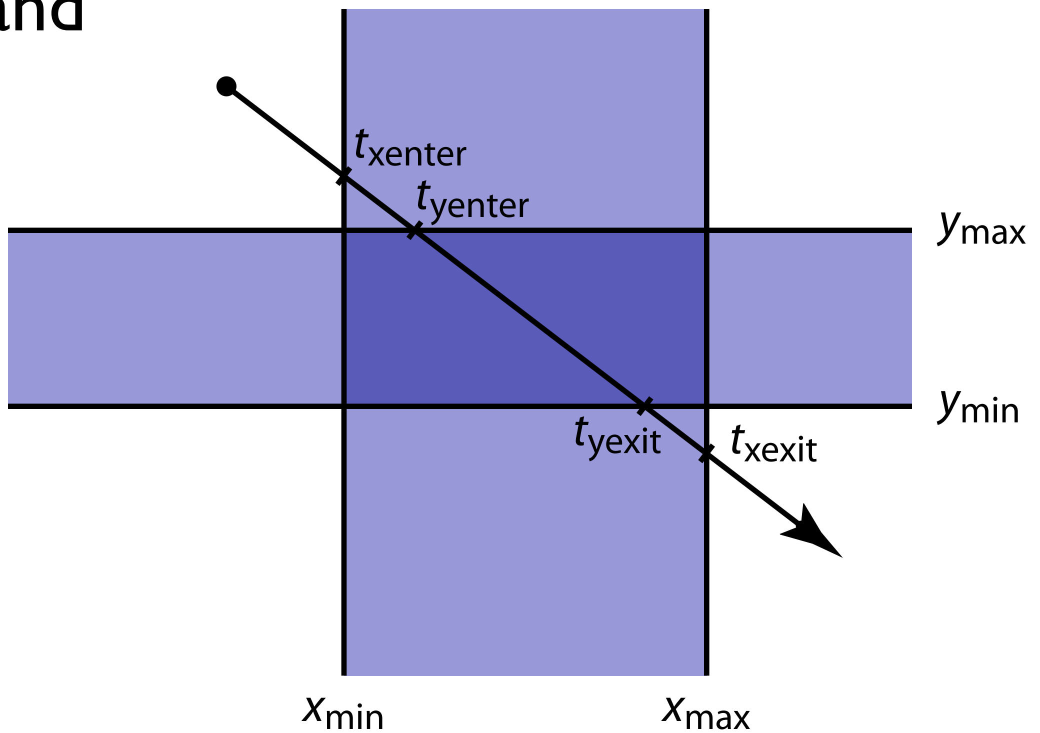
Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

$$t_{x\text{enter}} = \min(t_{x\min}, t_{x\max})$$
$$t_{x\text{exit}} = \max(t_{x\min}, t_{x\max})$$

$$t_{y\text{enter}} = \min(t_{y\min}, t_{y\max})$$
$$t_{y\text{exit}} = \max(t_{y\min}, t_{y\max})$$

$$t_{\text{enter}} = \max(t_{x\text{enter}}, t_{y\text{enter}})$$
$$t_{\text{exit}} = \min(t_{x\text{exit}}, t_{y\text{exit}})$$



Ray-triangle intersection

- Condition 1: point is on ray

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Condition 2: point is on plane

$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0$$

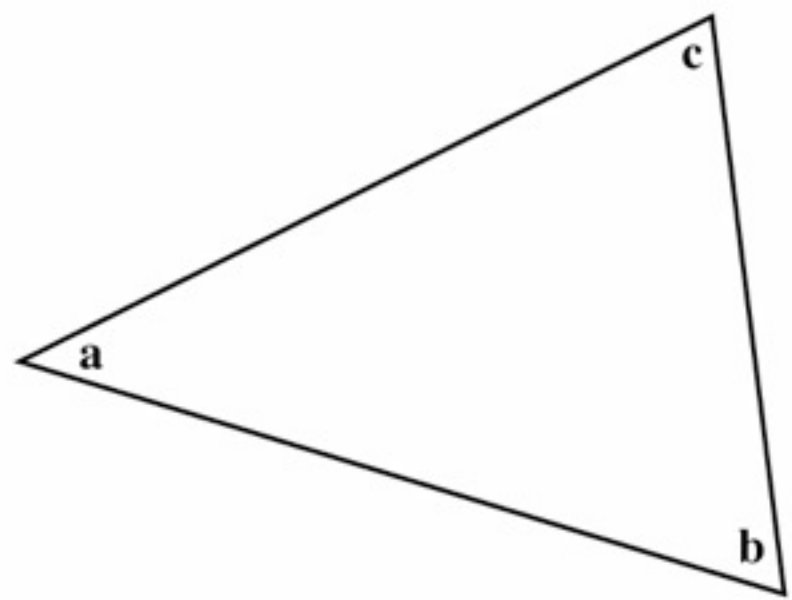
- Condition 3: point is on the inside of all three edges
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

$$(\mathbf{p} + t\mathbf{d} - \mathbf{a}) \cdot \mathbf{n} = 0$$

$$t = \frac{(\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

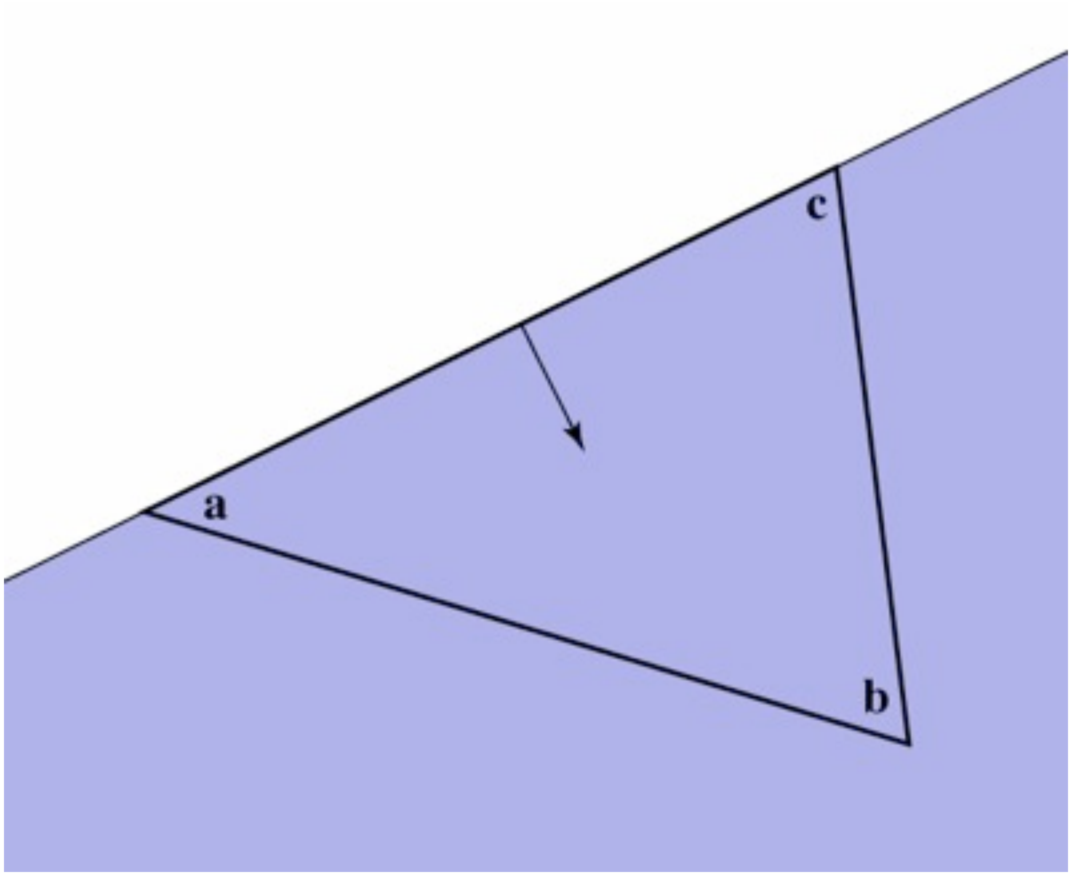
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces



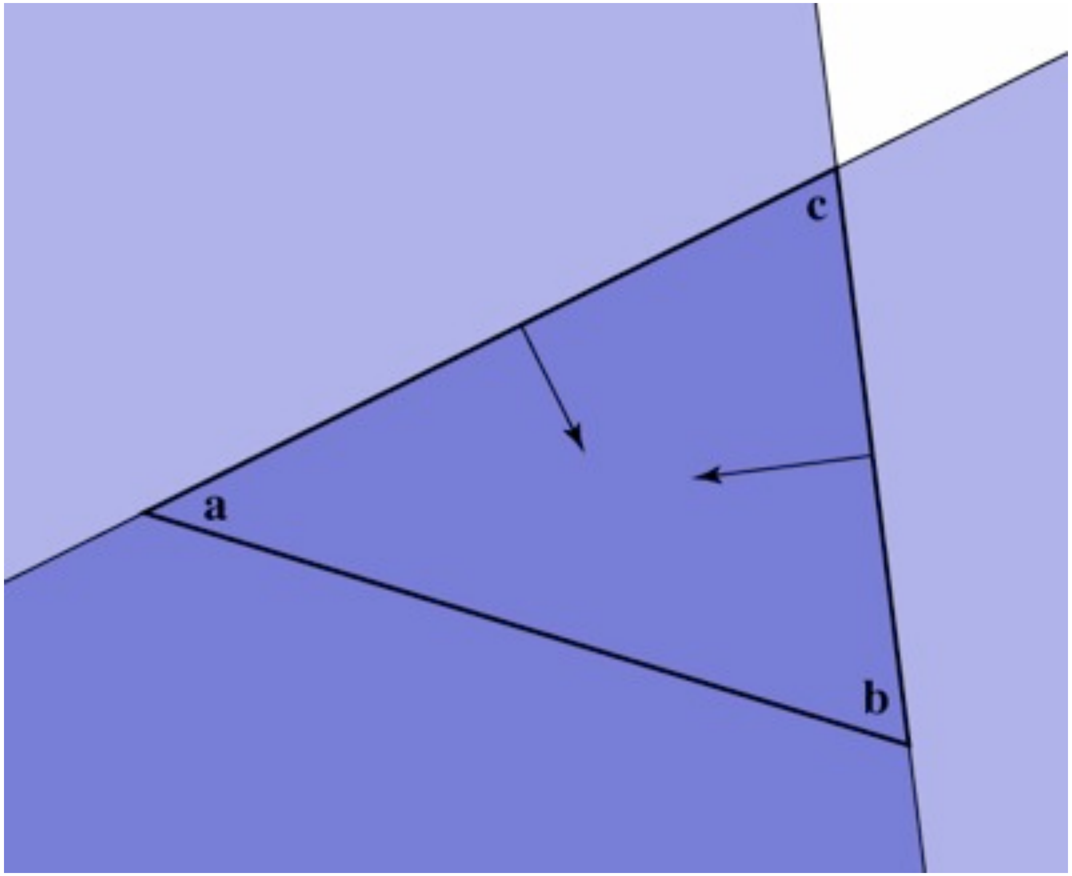
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces



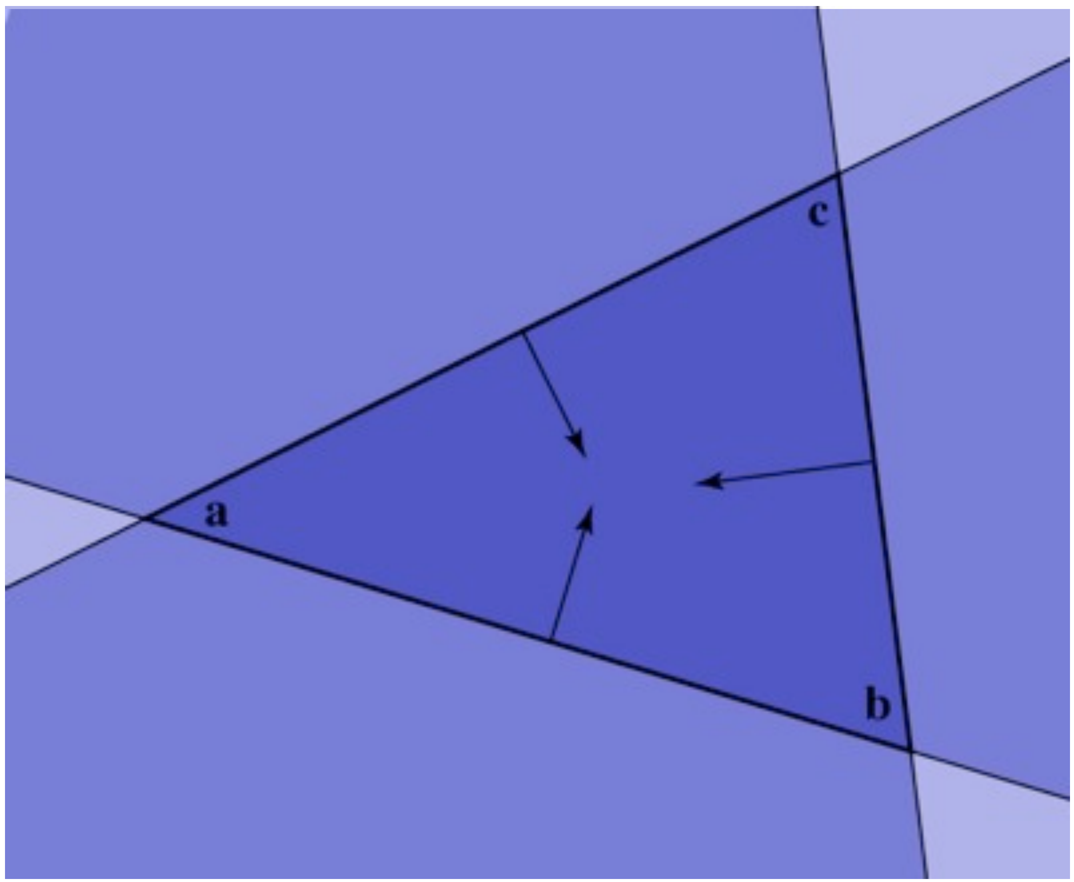
Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces



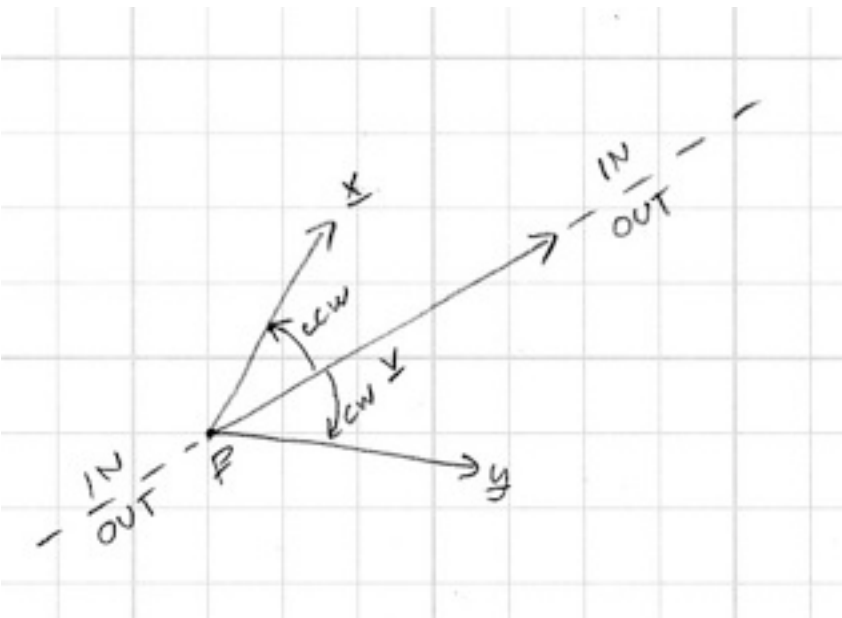
Ray-triangle intersection

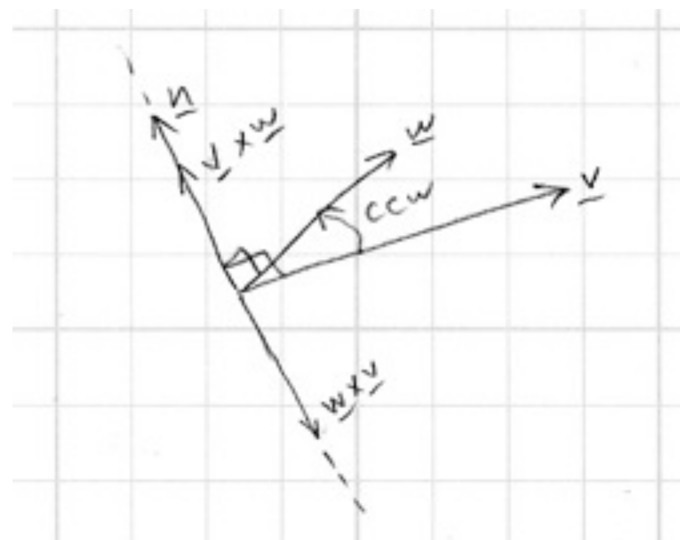
- In plane, triangle is the intersection of 3 half spaces



Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  - vector of edge to vector to **x**
- Use cross product to decide



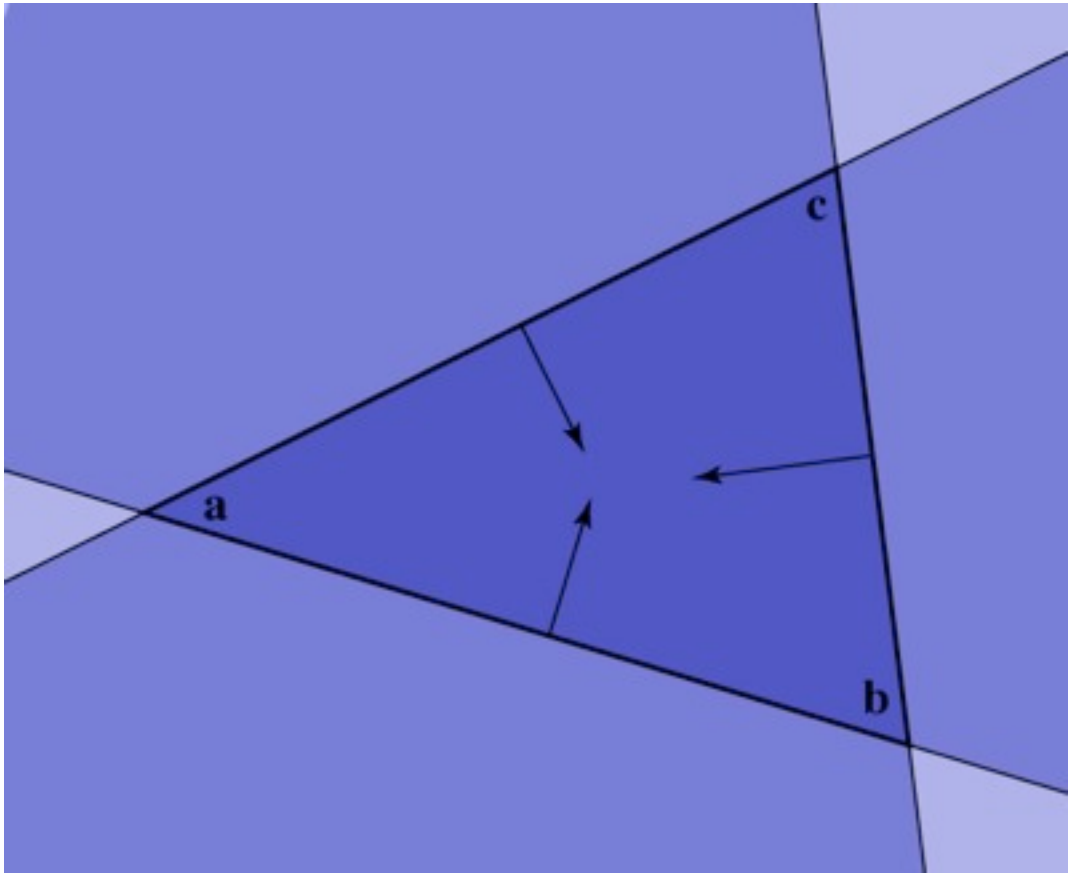


Ray-triangle intersection

$$(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} > 0$$

$$(\mathbf{c} - \mathbf{b}) \times (\mathbf{x} - \mathbf{b}) \cdot \mathbf{n} > 0$$

$$(\mathbf{a} - \mathbf{c}) \times (\mathbf{x} - \mathbf{c}) \cdot \mathbf{n} > 0$$



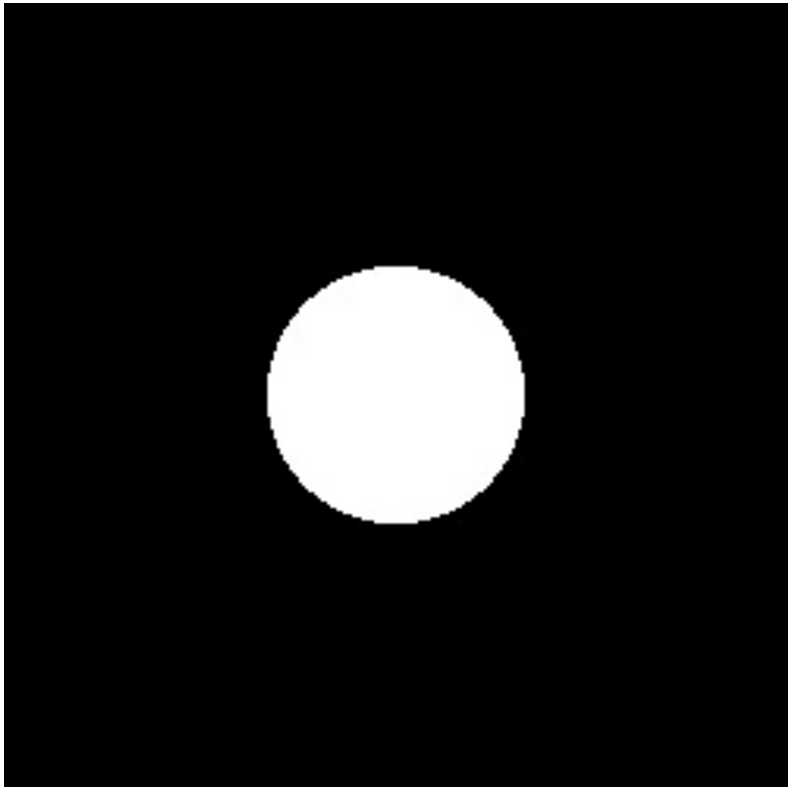
Ray-triangle intersection

- See book for a more efficient method based on linear systems
  - (don’t need this for Ray 1 anyhow—but stash away for Ray 2)

Image so far

- With eye ray generation and sphere intersection

```
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        hitSurface, t = s.intersect(ray, 0, +inf)
        if hitSurface is not null
            image.set(ix, iy, white);
    }
```



Intersection against many shapes

- The basic idea is:

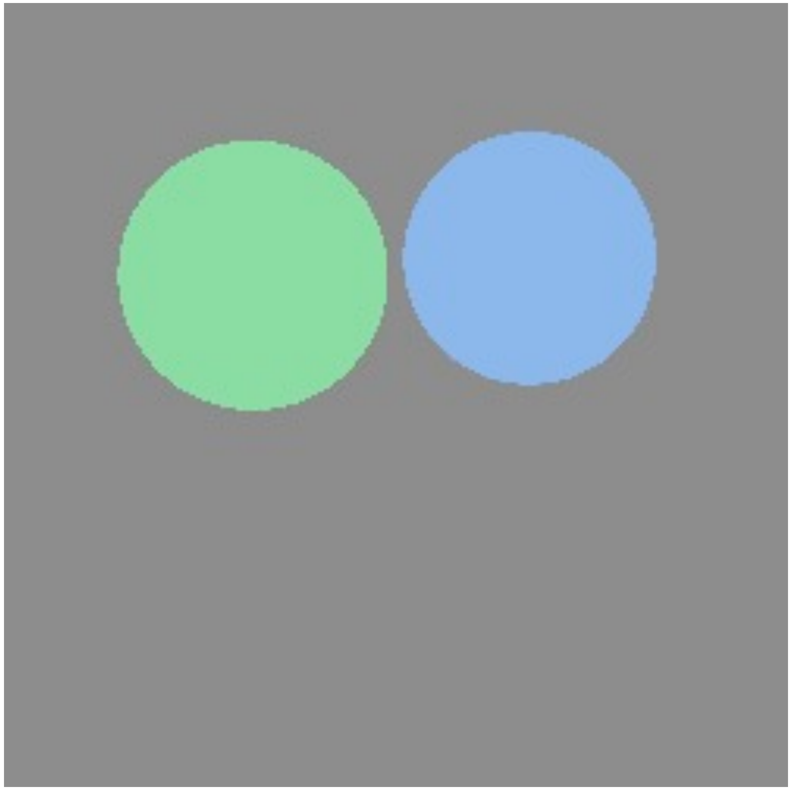
```
Group.intersect (ray, tMin, tMax) {
    tBest = +inf; firstSurface = null;
    for surface in surfaceList {
        hitSurface, t = surface.intersect(ray, tMin, tBest);
        if hitSurface is not null {
            tBest = t;
            firstSurface = hitSurface;
        }
    }
return hitSurface, tBest;
}
```

  – this is linear in the number of shapes
    but there are sublinear methods (acceleration structures)

Image so far

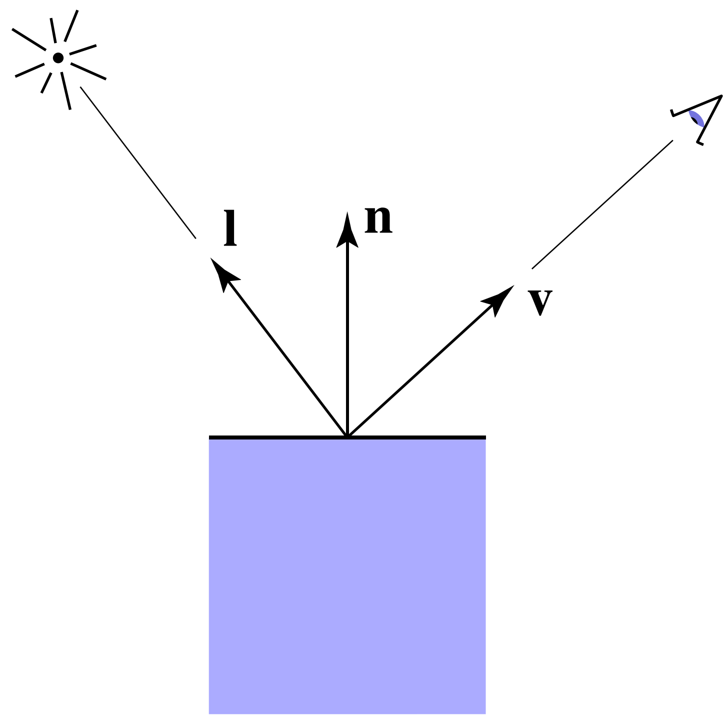
- With eye ray generation and scene intersection

```
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;
}
```



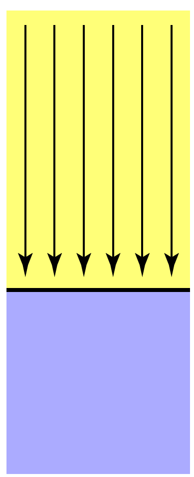
Shading

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

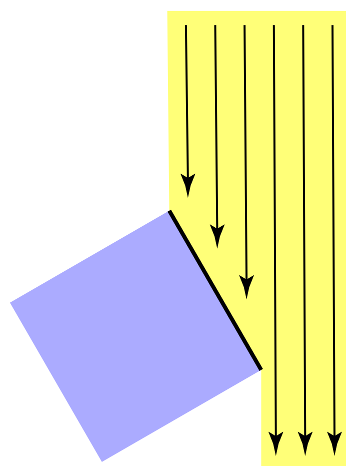


Diffuse reflection

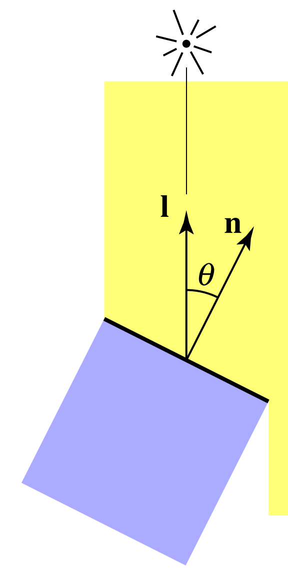
- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law



Top face of cube receives a certain amount of light



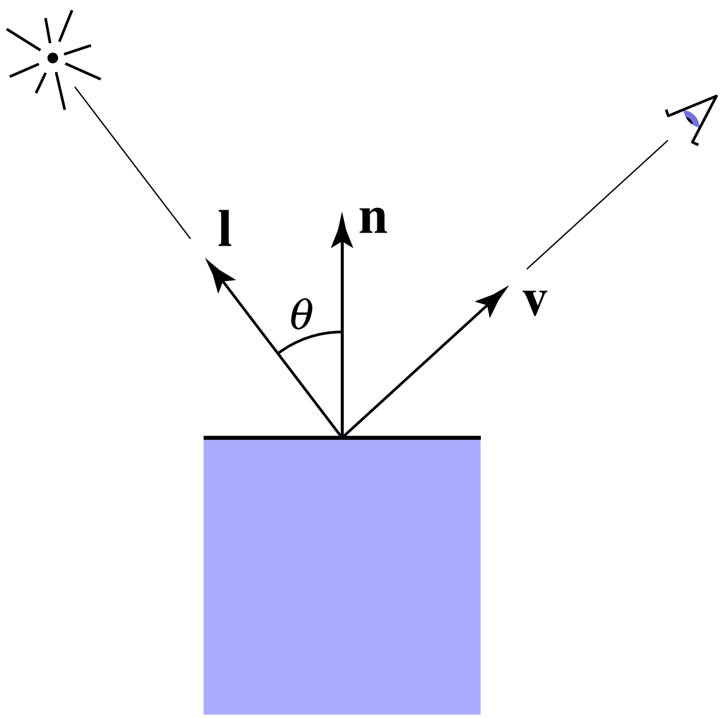
Top face of 60° rotated cube intercepts half the light



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

Lambertian shading

- Shading independent of view direction



illumination from source

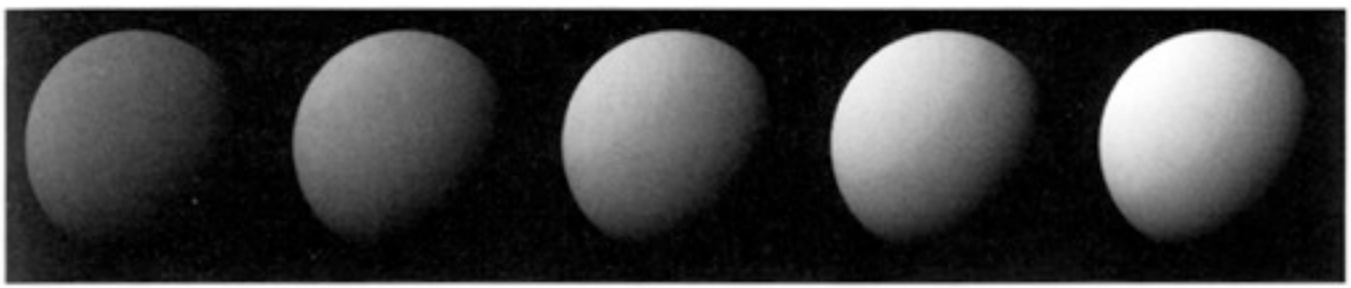
$$L_d = k_d \, I \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse coefficient

diffusely reflected light

Lambertian shading

- Produces matte appearance



$k_d \longrightarrow$

[Foley et al.]

Diffuse shading

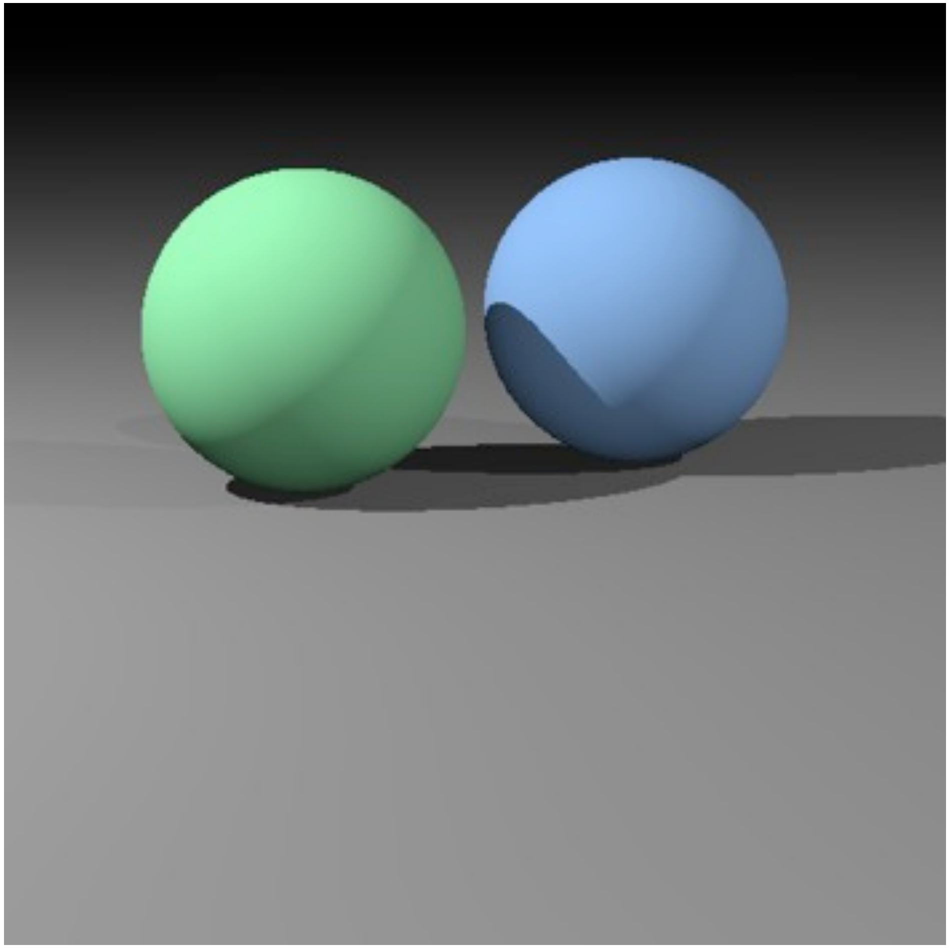
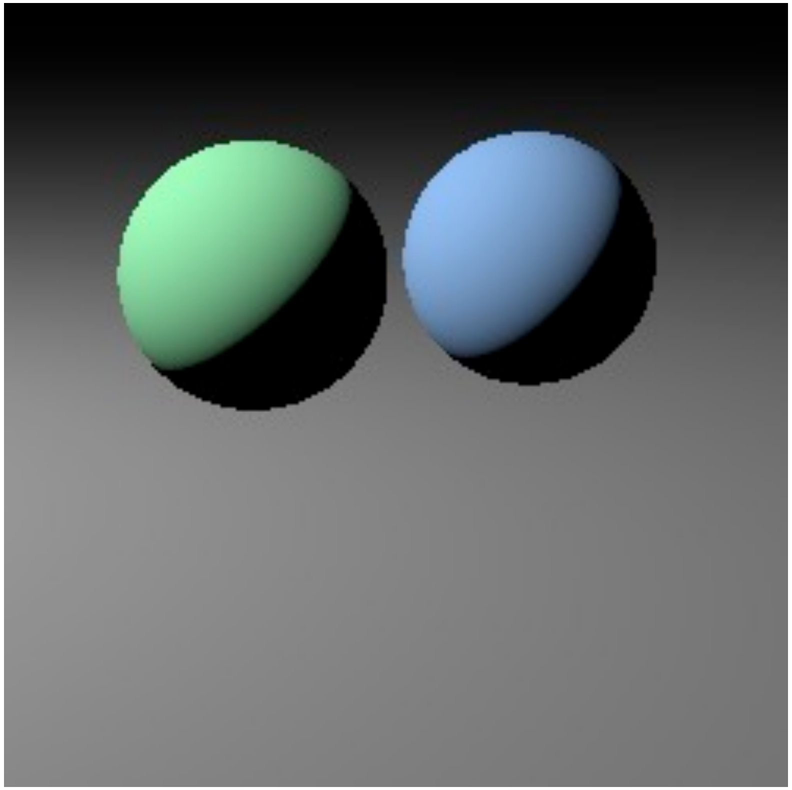


Image so far

```
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
}
```

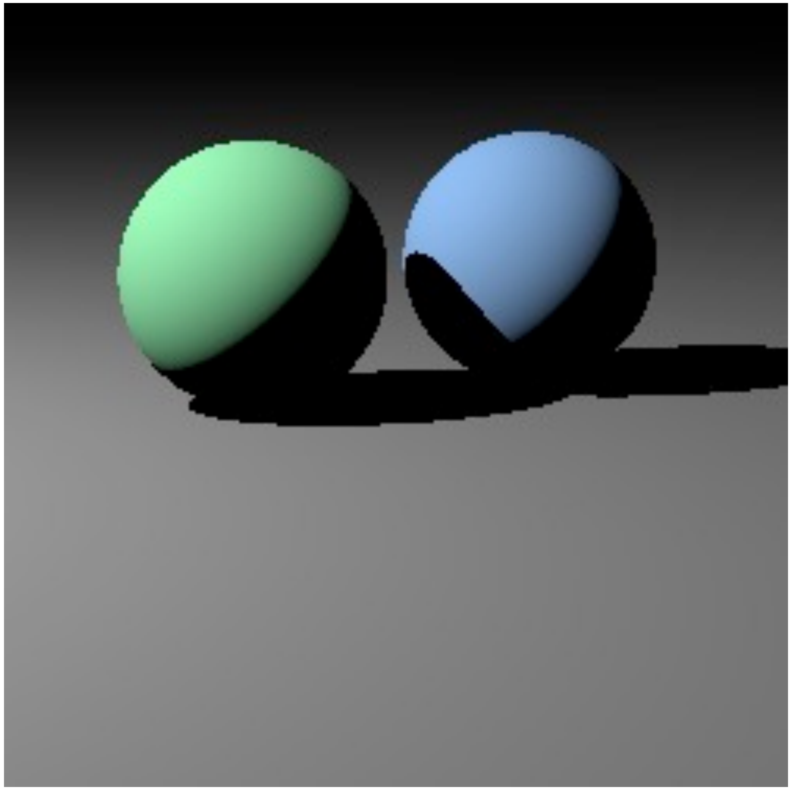


Shadows

- Surface is only illuminated if nothing blocks its view of the light.
- With ray tracing it's easy to check
  - just intersect a ray with the scene!

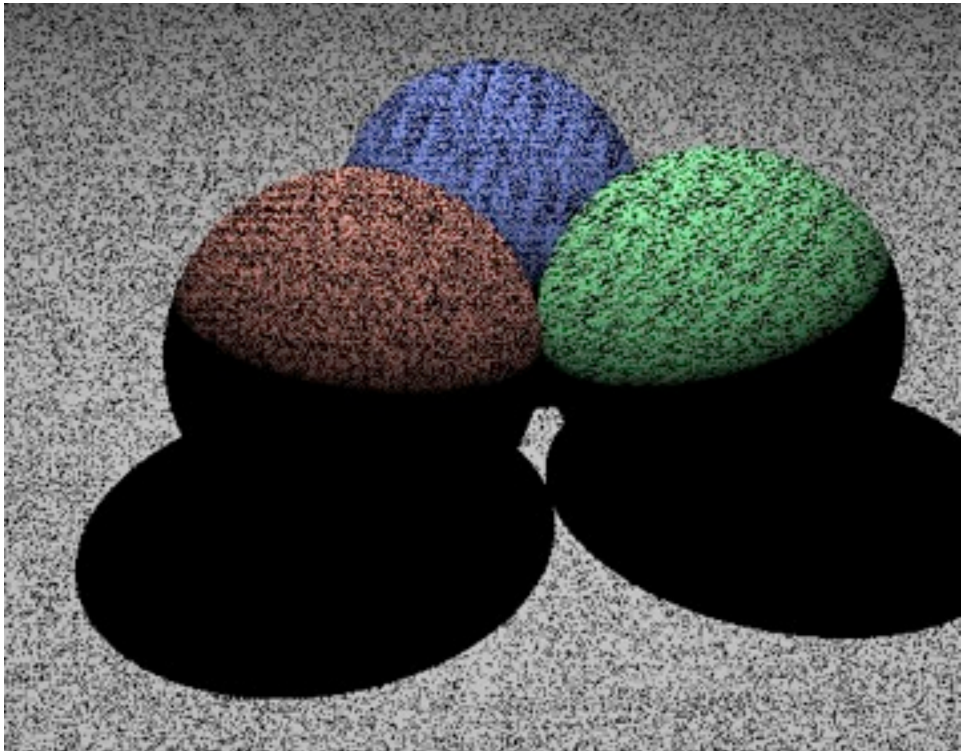
Image so far

```
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

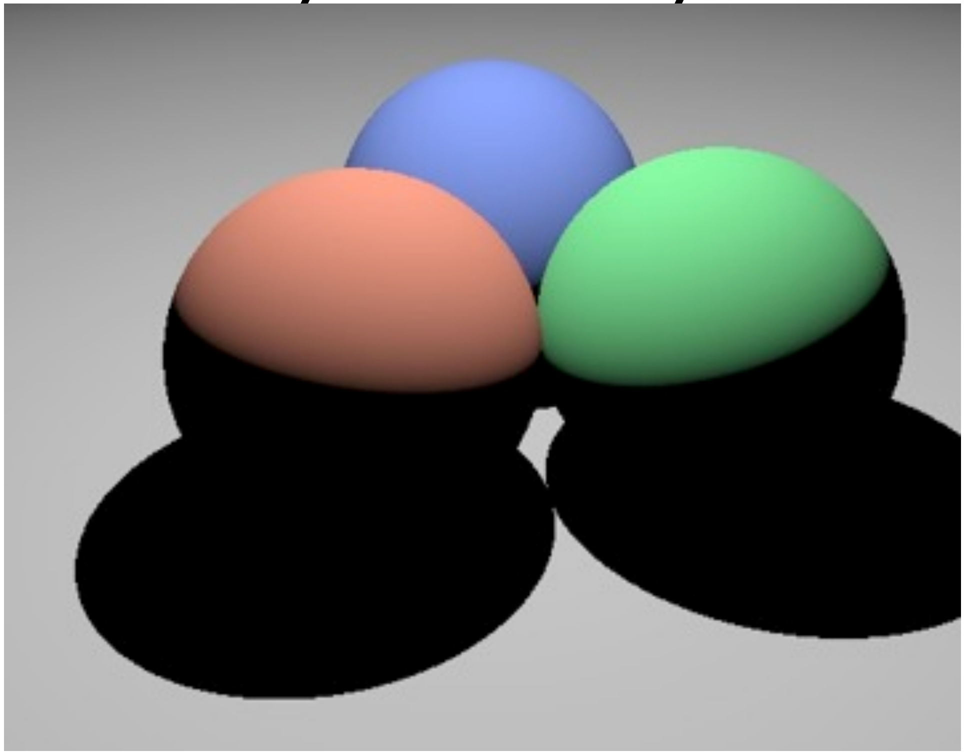
- Don’t fall victim to one of the classic blunders:



- 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



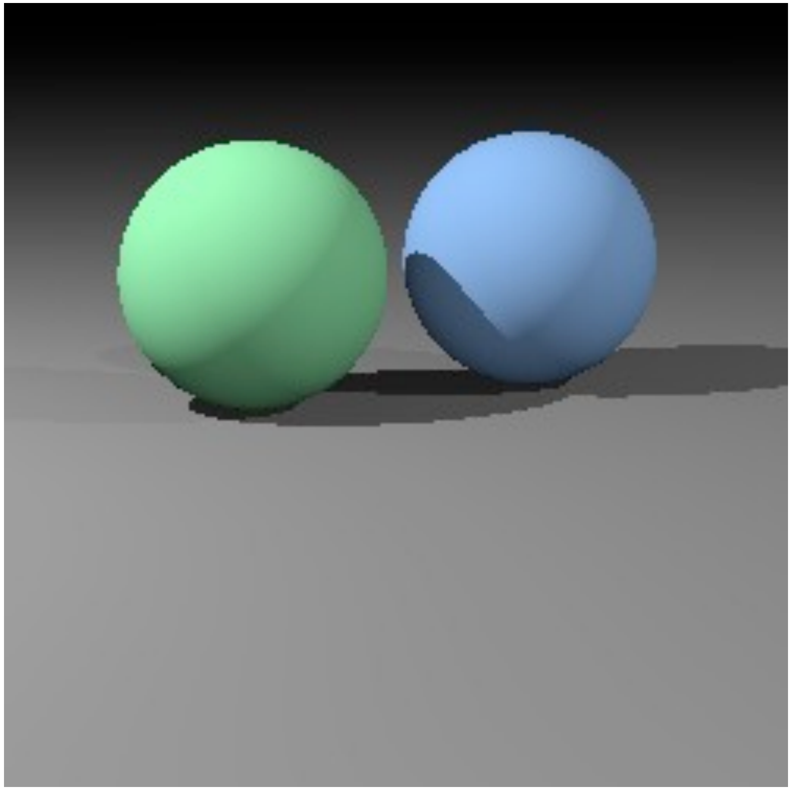
- Do this by moving the start point, or by limiting the *t* range

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…

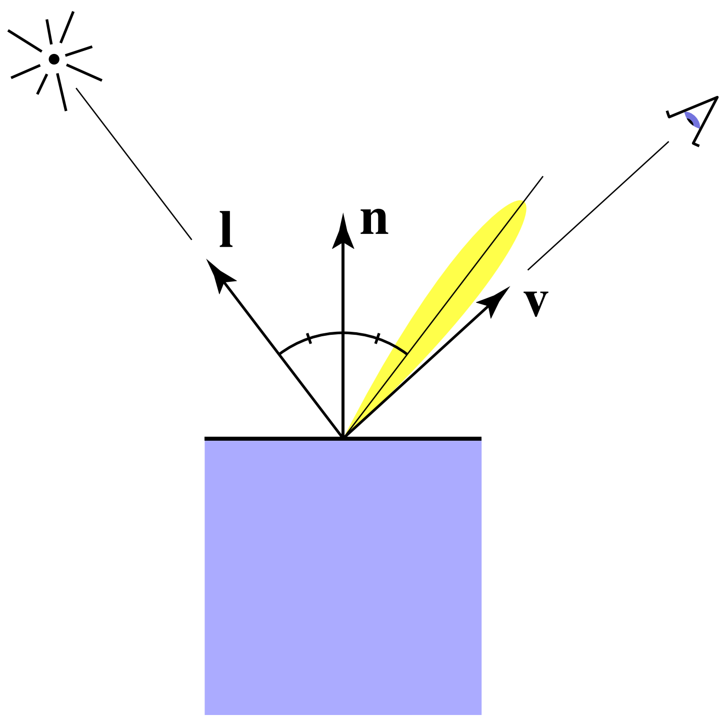
Image so far

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



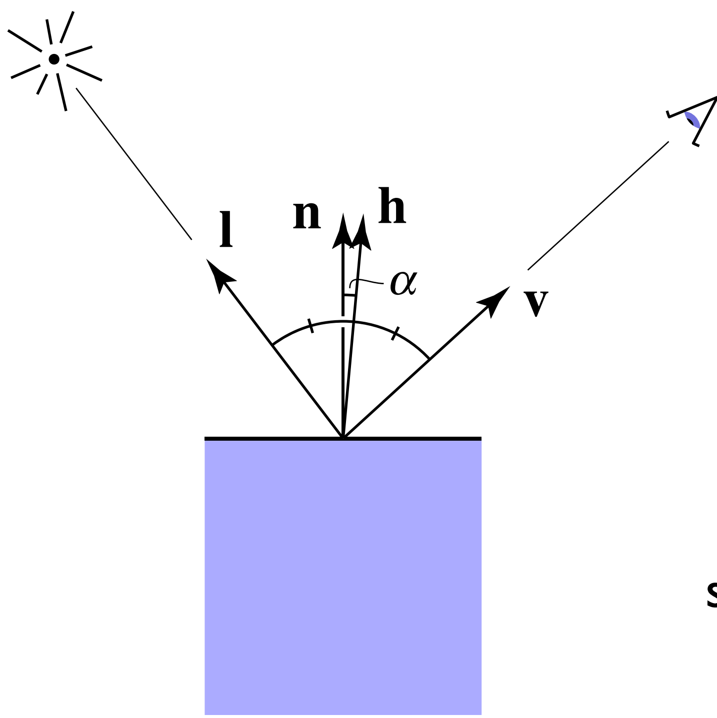
Specular shading (Blinn-Phong)

- Intensity depends on view direction
  - bright near mirror configuration



Specular shading (Blinn-Phong)

- Close to mirror ⇔ half vector near normal
  - Measure “near” by dot product of unit vectors



$$\mathbf{h} = \mathrm{bisector}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

$$L_s = k_s\, I \max(0, \cos\alpha)^p = k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

specularly reflected light

specular coefficient

Phong model—plots

- Increasing *n* narrows the lobe

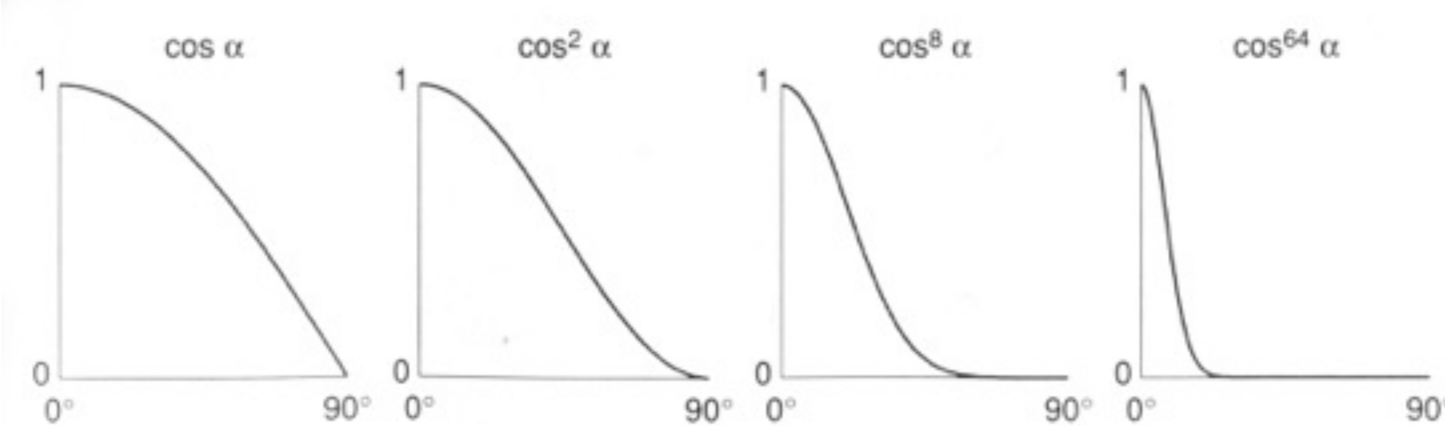
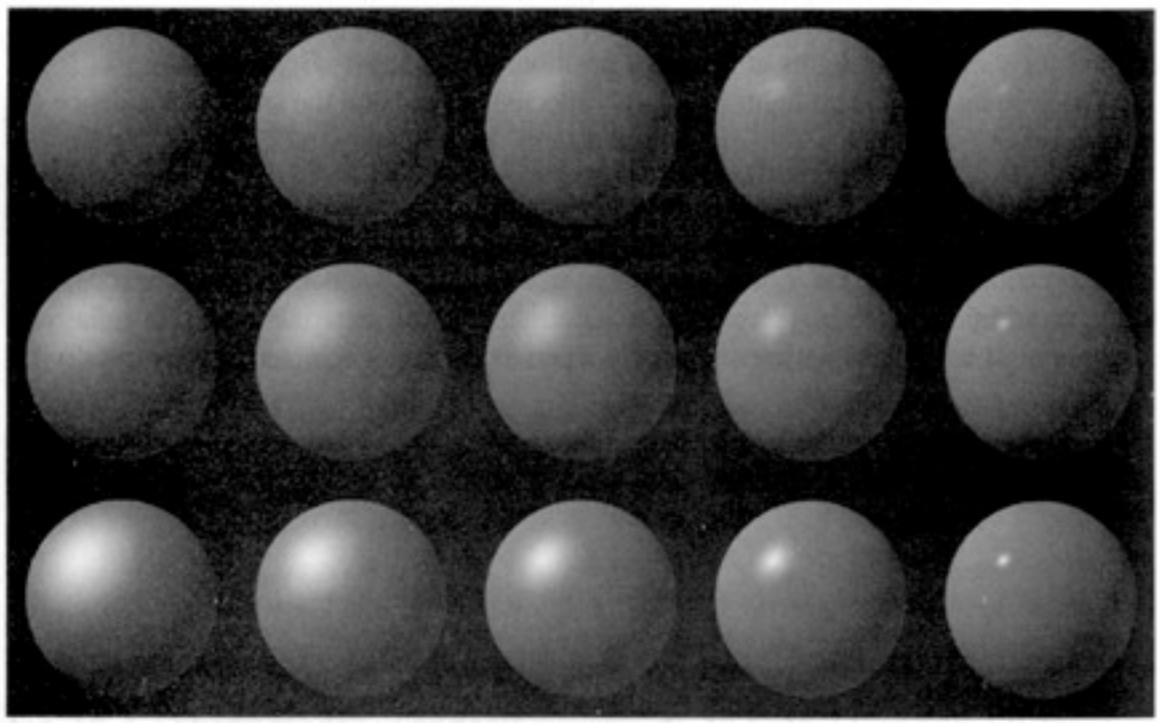


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

[Foley et al.]

Specular shading

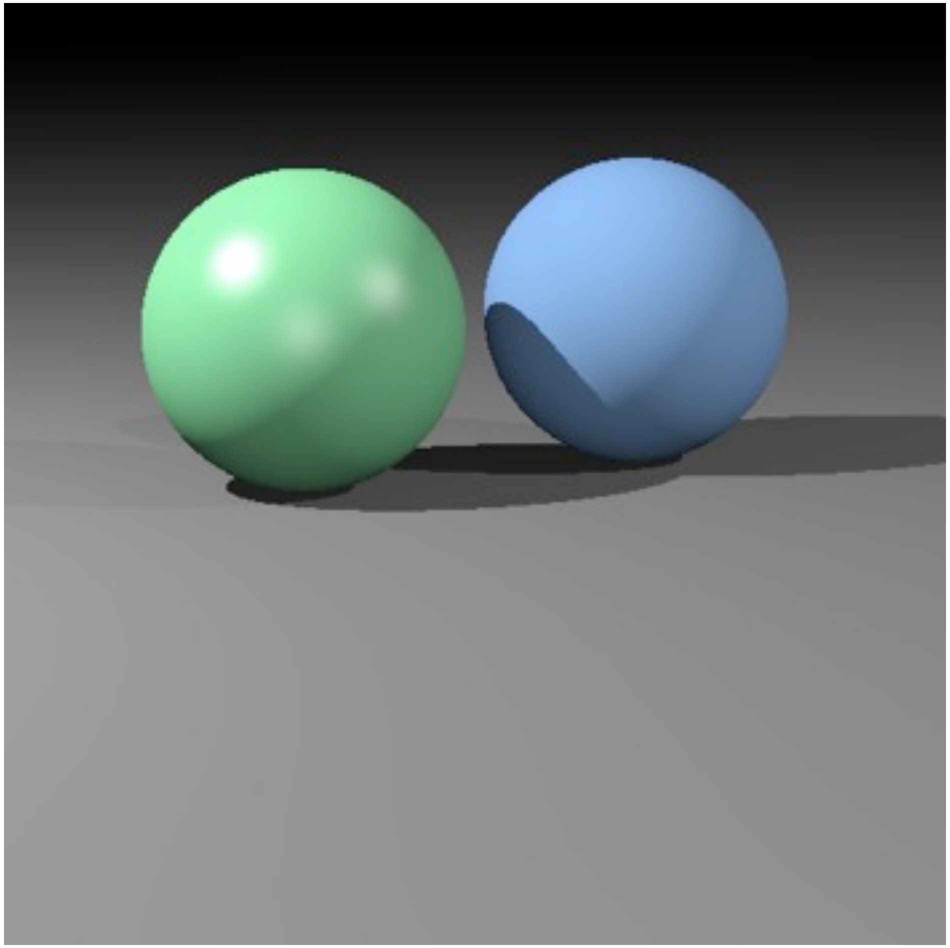
k_s



p

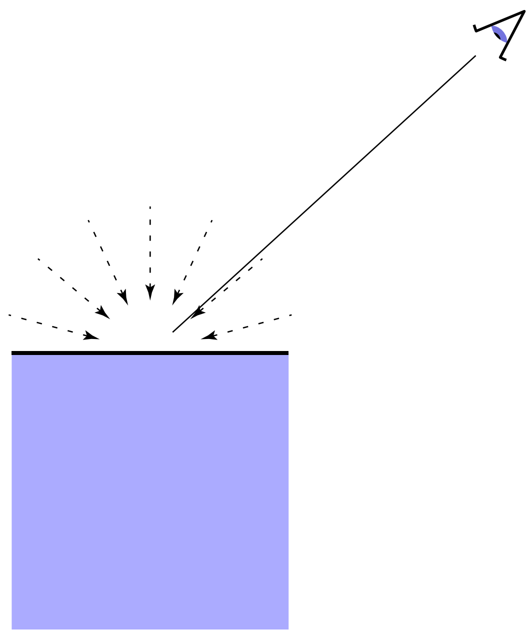
[Foley et al.]

Diffuse + Phong shading



Ambient shading

- Shading that does not depend on anything
  - add constant color to account for disregarded illumination and fill in black shadows



$$L_a = k_a \, I_a$$

ambient coefficient

reflected ambient light

Putting it together

- Usually include ambient, diffuse, Phong in one model

$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, I \max(0, \mathbf{n} \cdot \mathbf{h})^p
\end{aligned}
$$

- The final result is the sum over many lights

$$
L = L_a + \sum_{i=1}^{N} \left[(L_d)_i + (L_s)_i \right]
$$

$$
L = k_a\, I_a + \sum_{i=1}^{N} \left[k_d\, I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s\, I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \right]
$$

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 re
  - Shadow rays can be cheaper (any intersection will do, don't need closest)
  - but: "First Get it Right, Then Make it Fast"