# Ray Tracing (Shading)

CS 4620 Lecture 7


 
(with previous instructor Marschner)

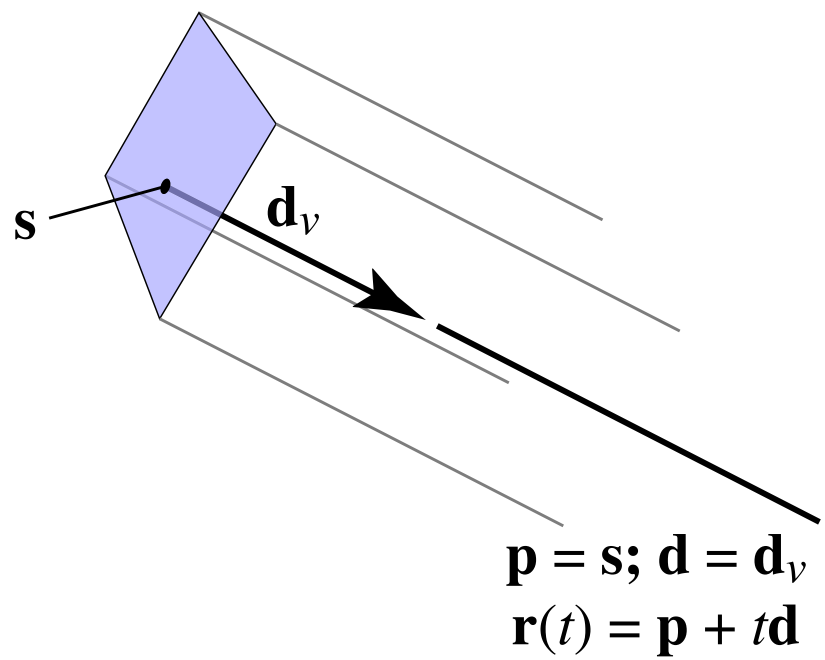
# Announcements

- A1 grading tonight
  - If you haven't signed up yet, do so immediately.

- A2 is out

# Generating eye rays—orthographic

- Just need to compute the view plane point **s**:




  – but where exactly is the view rectangle?

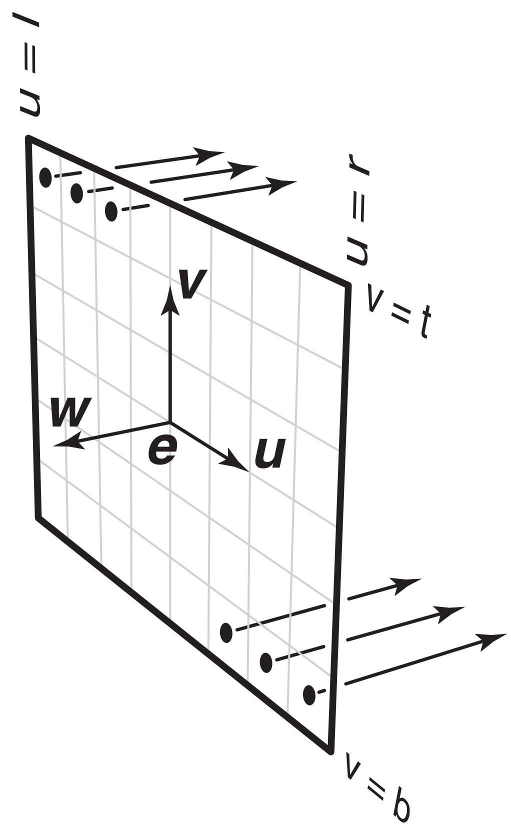
# Generating eye rays—orthographic

- Positioning the view rectangle
  - establish three vectors to be *camera basis:* **u**, **v**, **w**
  - view rectangle is in **u**–**v** plane, specified by l, r, t, b
  - now ray generation is easy:

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$

$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = -\mathbf{w}$$

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

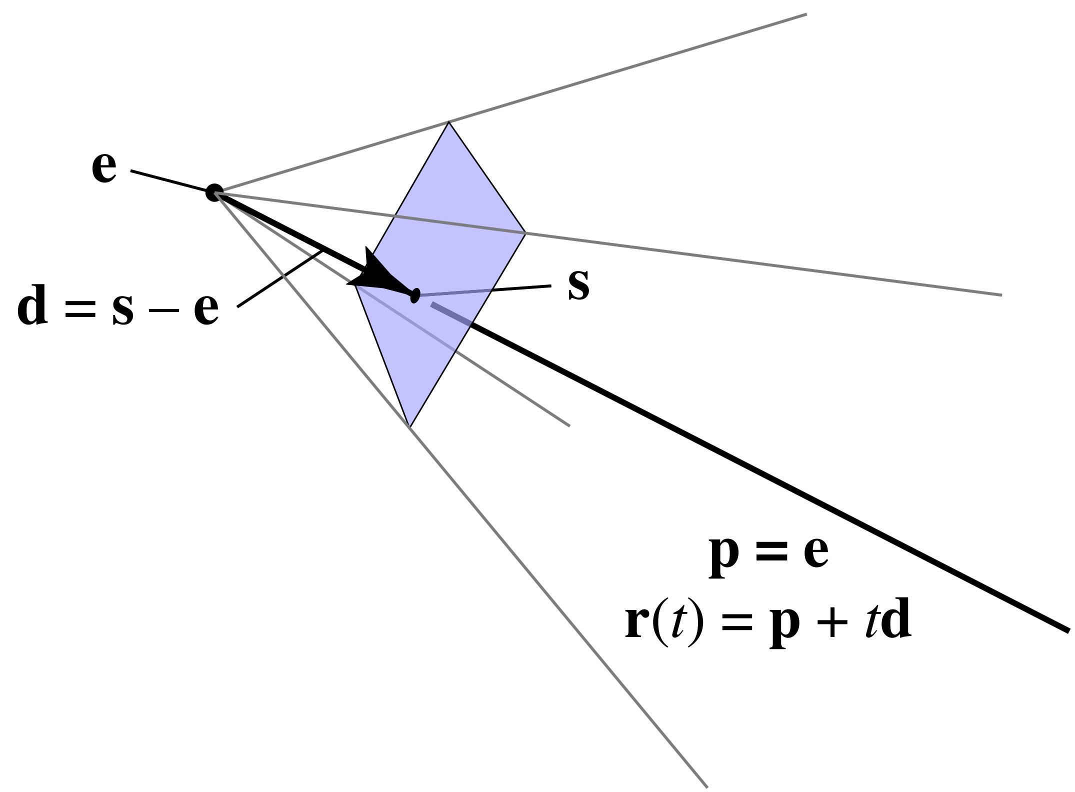


# Camera: more general

- Orthonormal bases
  - viewPoint == e
  - viewDir == -w, viewUp == v
    - Compute u from the above
    - Compute v from u and w

# Generating eye rays—perspective

- View rectangle needs to be away from viewpoint
- Distance is important: "focal length" of camera
  - still use camera frame but position view rect away from viewpoint
  - ray origin always **e**
  - ray direction now controlled by **s**



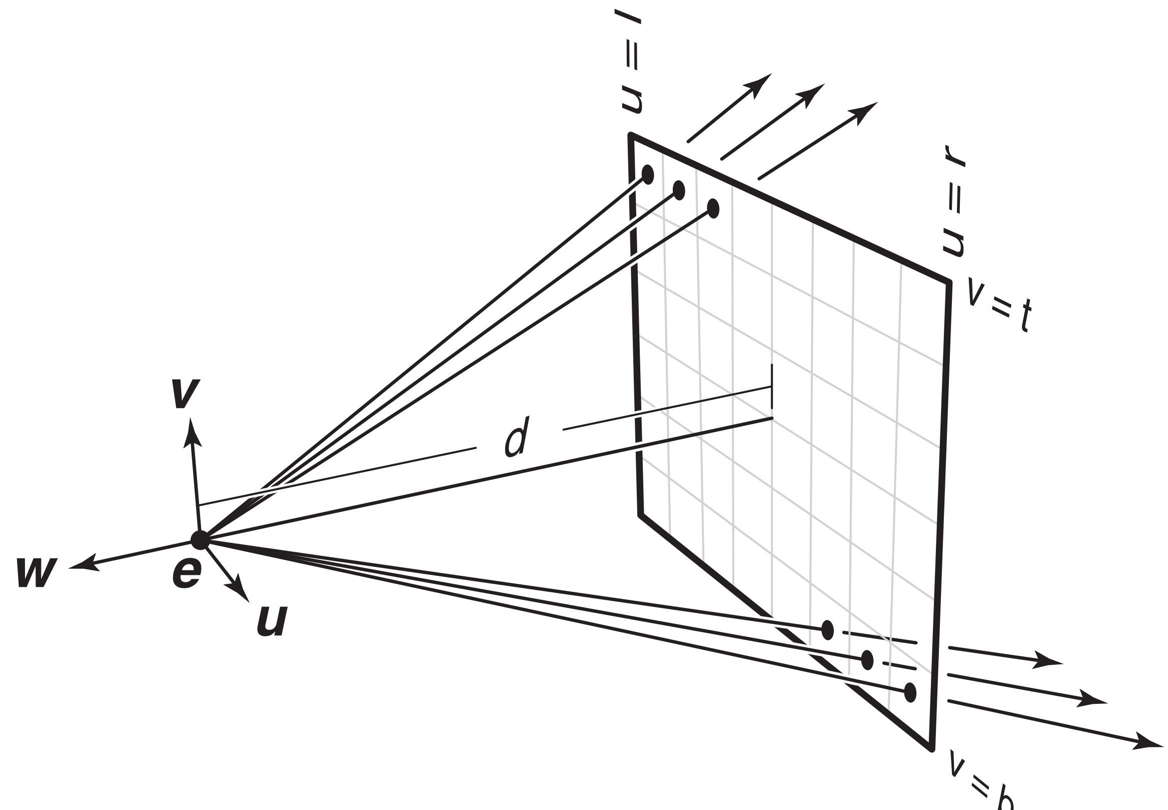
# Generating eye rays—perspective

- Compute **s** in the same way; just subtract *d***w**
  - coordinates of **s** are (*u*, *v*, –*d*)

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$

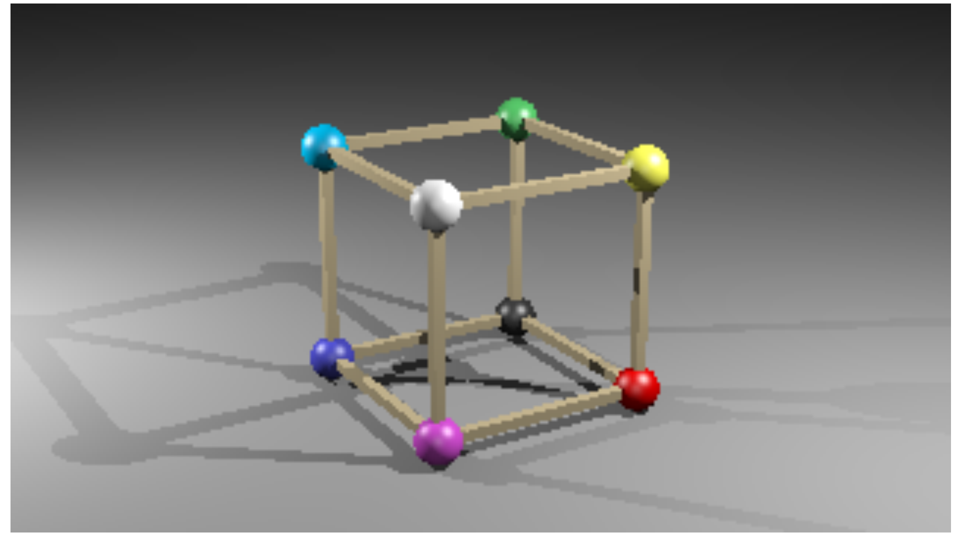
$$\mathbf{p} = \mathbf{e};\ \mathbf{d} = \mathbf{s} - \mathbf{e}$$

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

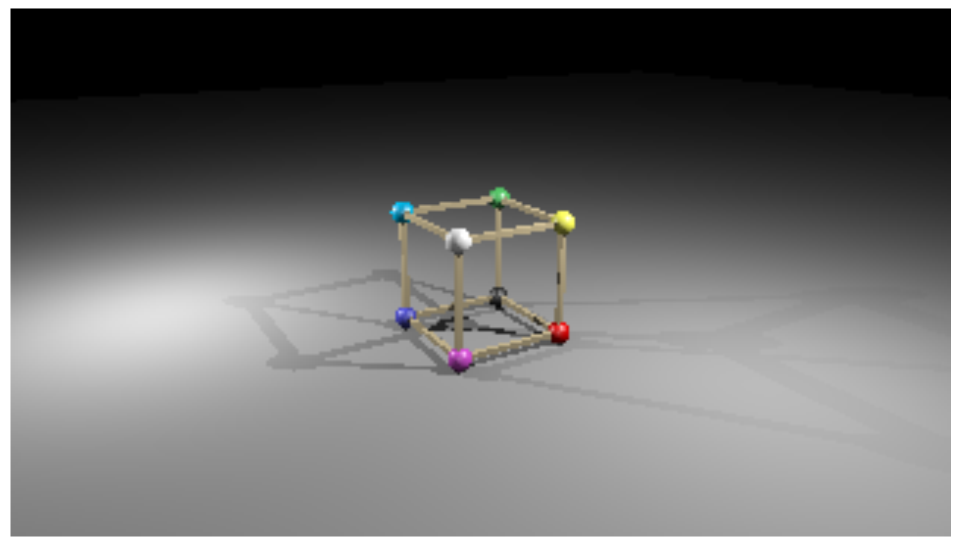


# Specifying views in Ray 1

```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>6</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>3</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



# Camera

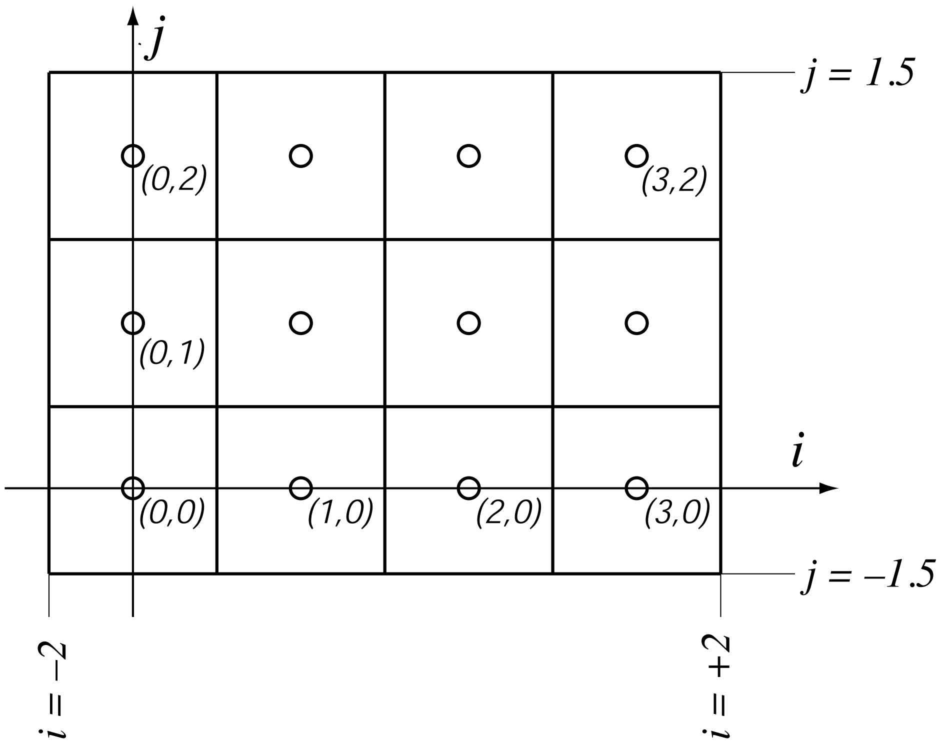
- Orthonormal bases
  - viewPoint == e
  - viewDir == -w, viewUp == v
    - Compute u from the above

l = -viewWidth/2

r = +viewWidth/2

n_x = imageWidth

# Where are the pixels located?



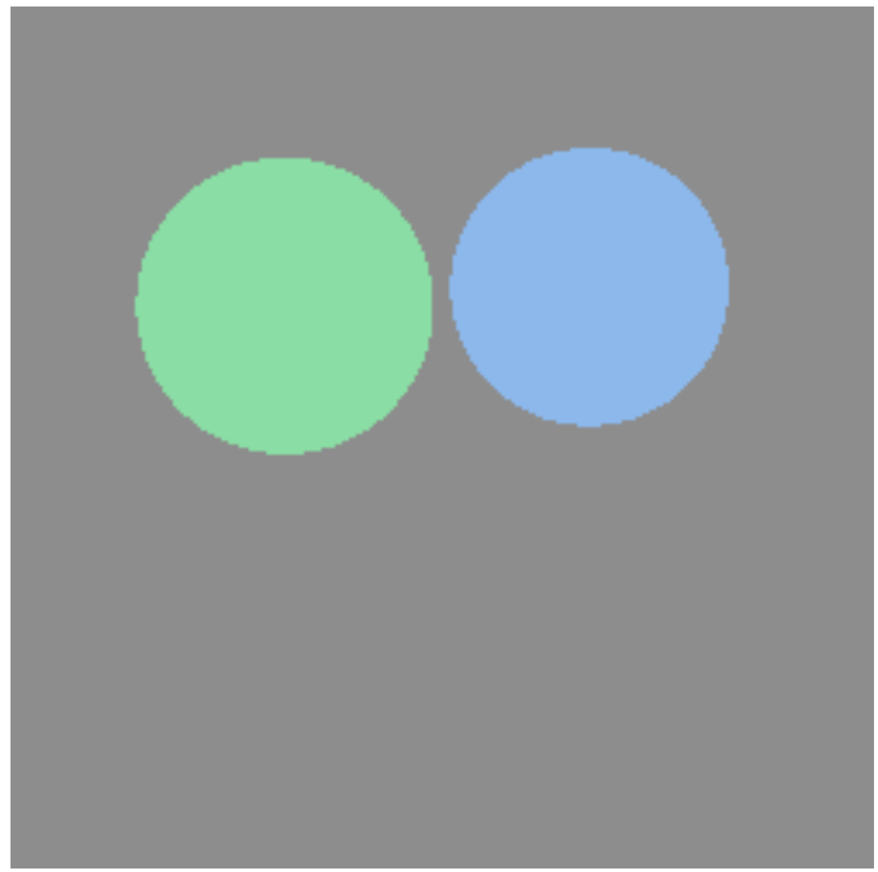

$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

# Ray Tracing: shading

# Image so far

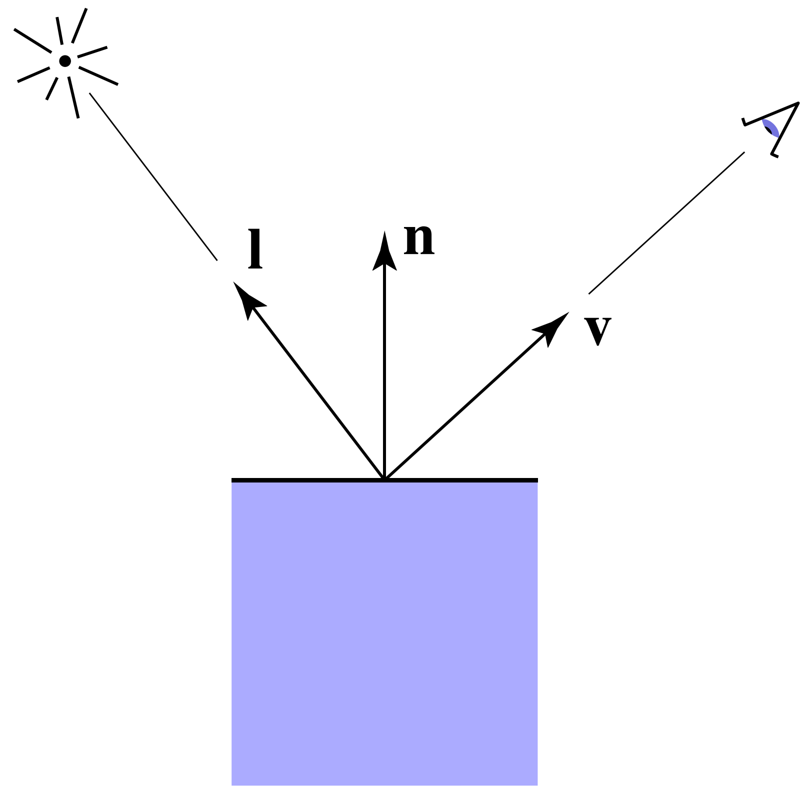
- With eye ray generation and scene intersection

```
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) {
    bool didhit = surfs.intersect(hit,ray, tMin, tMax);
    if (didhit) return hit.surface.color();
    else return black;
}
```



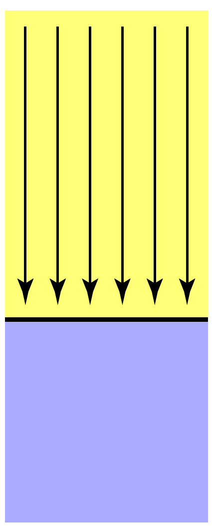
# Shading

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

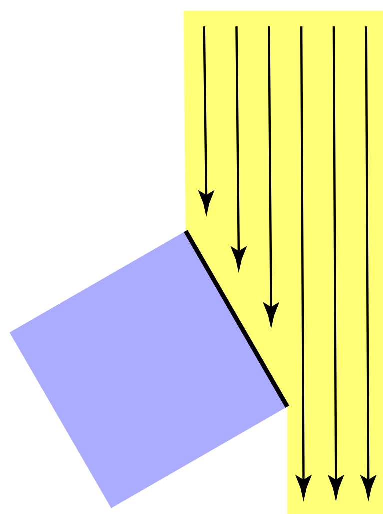


# Diffuse reflection

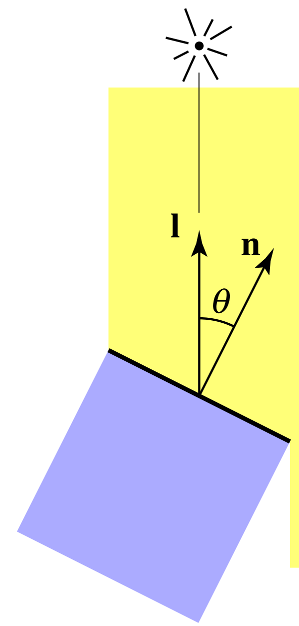
- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law



Top face of cube receives a certain amount of light

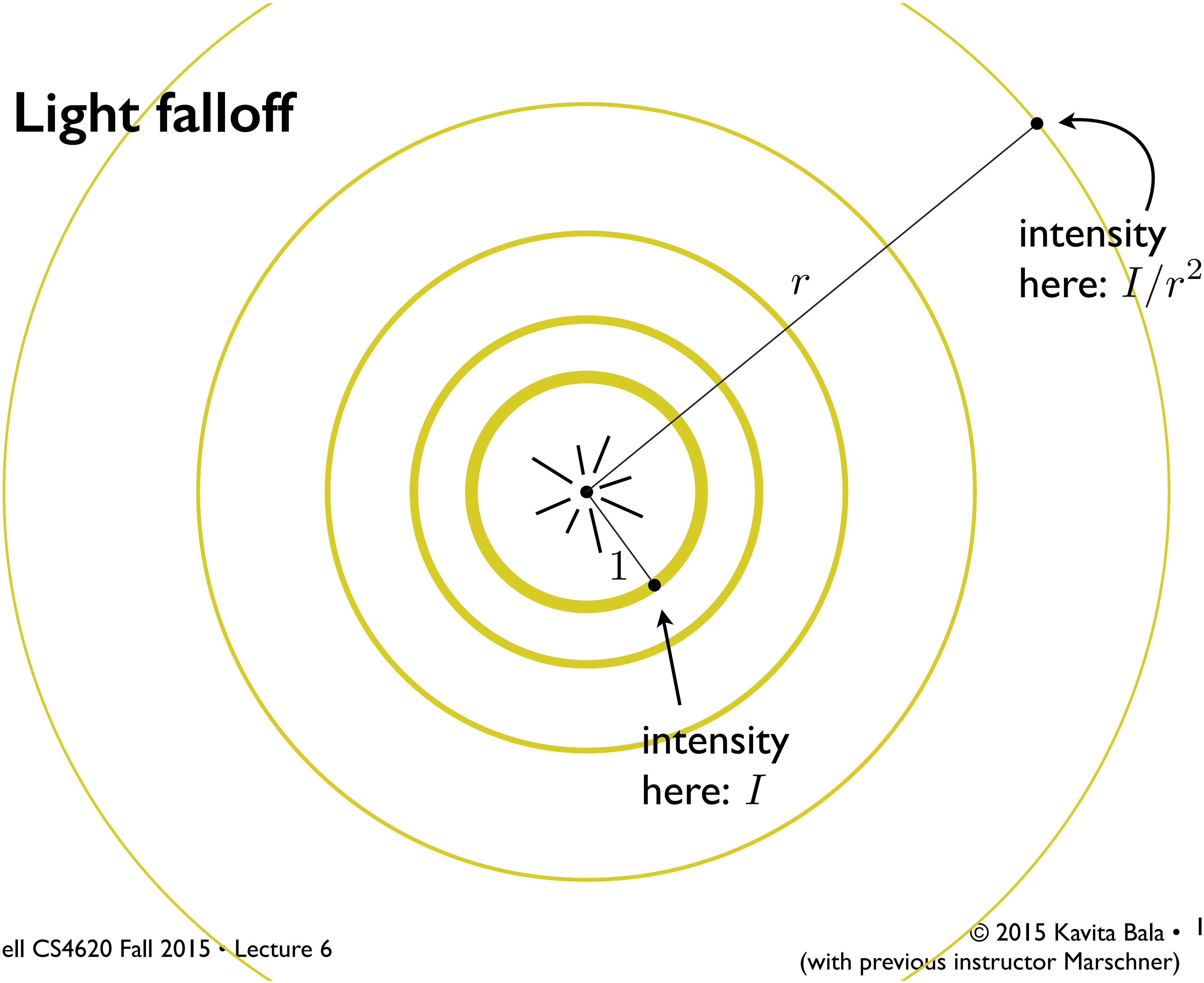


Top face of 60° rotated cube intercepts half the light



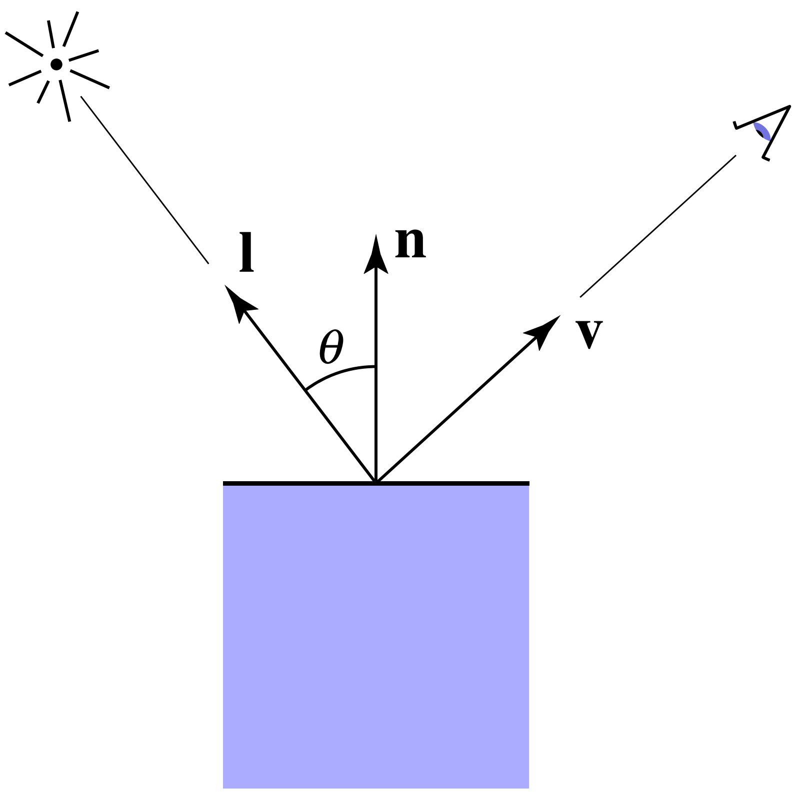

In general, light per unit area is proportional to $\cos\theta = \mathbf{l} \cdot \mathbf{n}$

# Light falloff



# Lambertian shading

- Shading independent of view direction




illumination from source

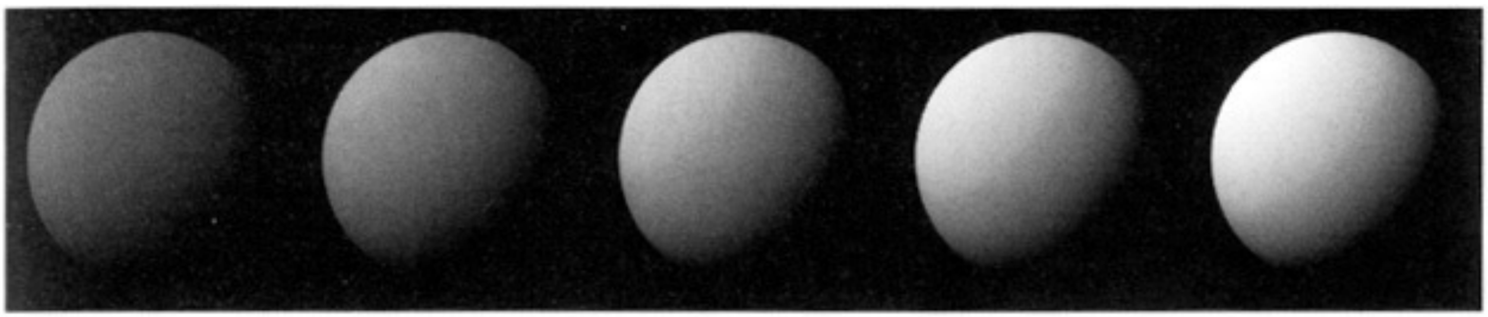
$$L_d = k_d \, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffusely reflected light

diffuse coefficient

# Lambertian shading

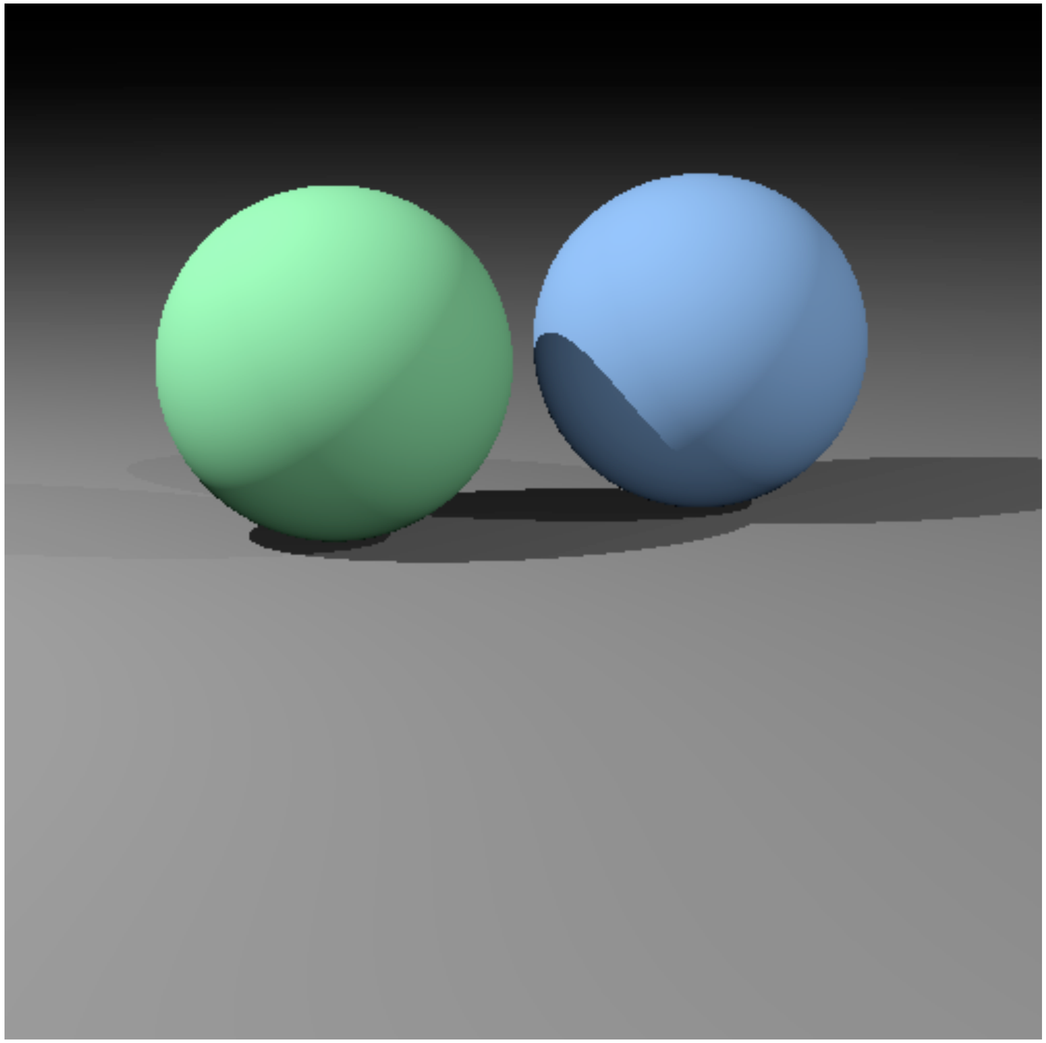
- Produces matte appearance



$k_d \longrightarrow$

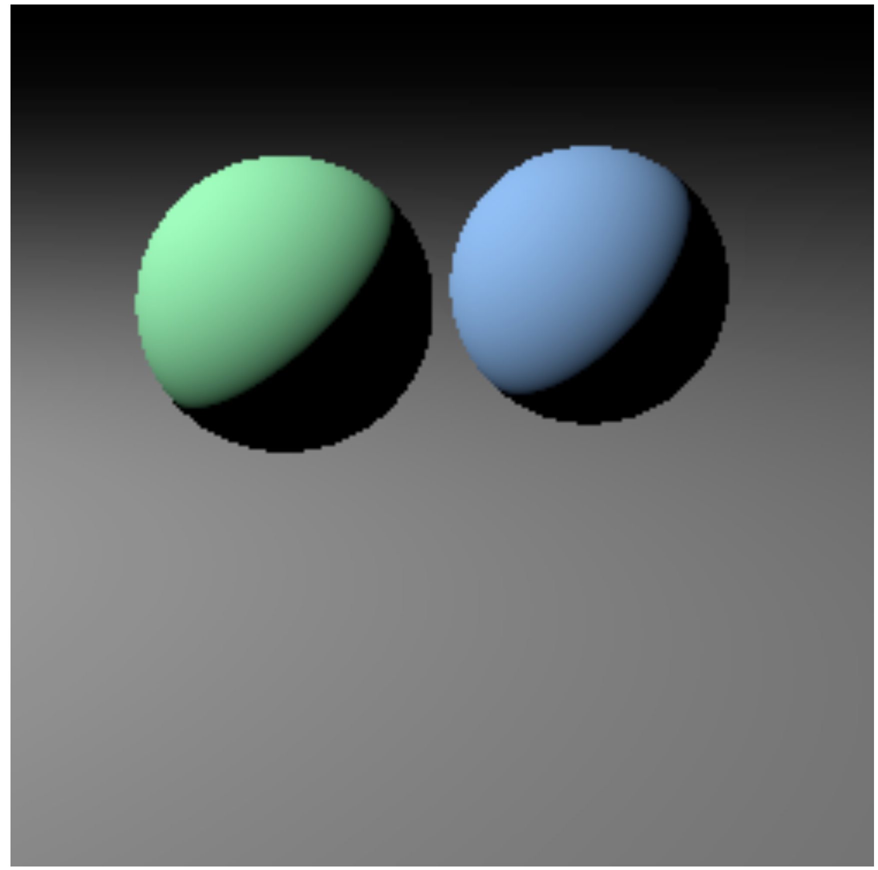
[Foley et al.]

# Diffuse shading



# Image so far

```
Scene.trace(Ray ray, tMin, tMax) {
    bool didhit = intersect(hit, ray, tMin, tMax);
    if didhit {
        point = ray.evaluate(hit.t);
        normal = hit.surface.getNormal(point);
        return hit.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
}
```

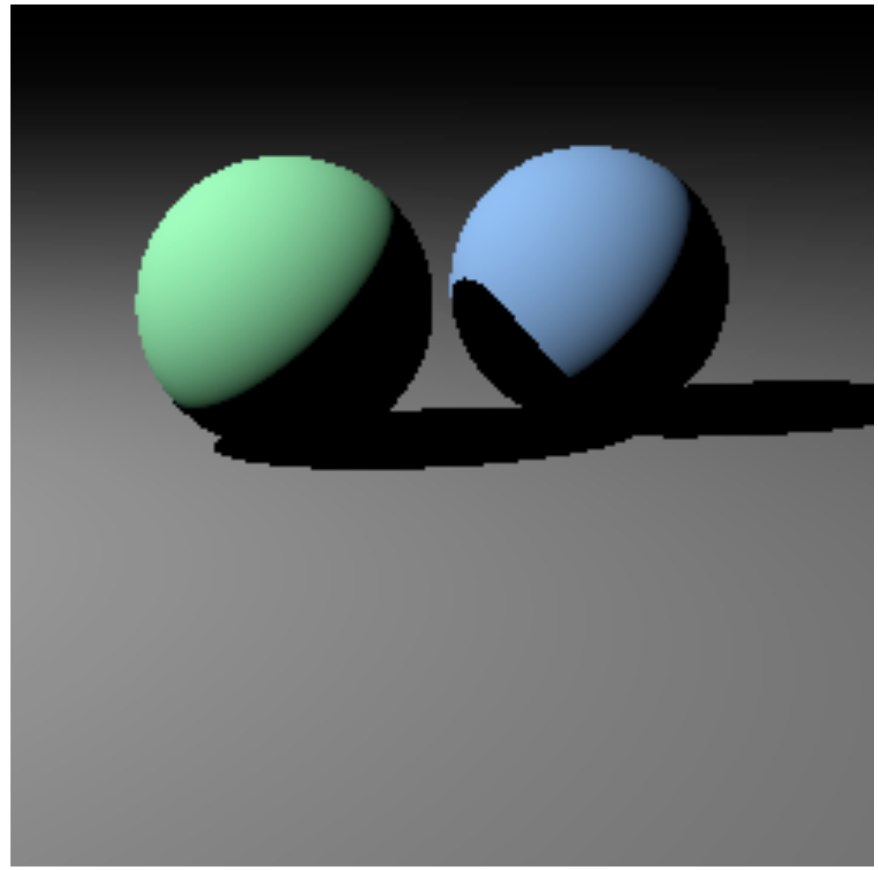


# Shadows

- 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!

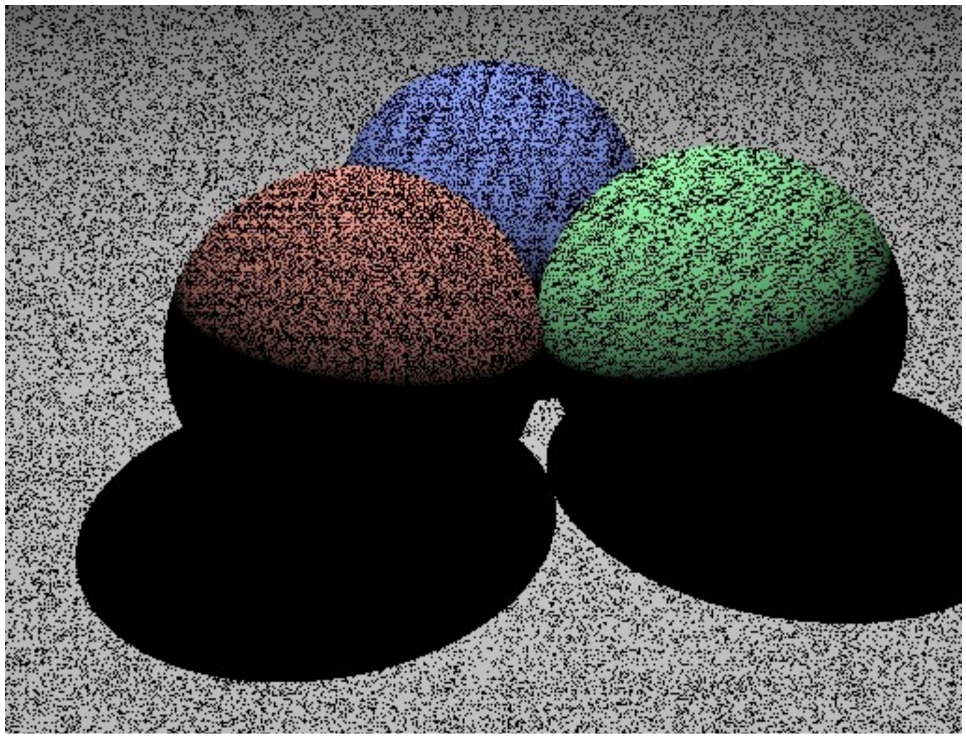
# Image so far

```
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

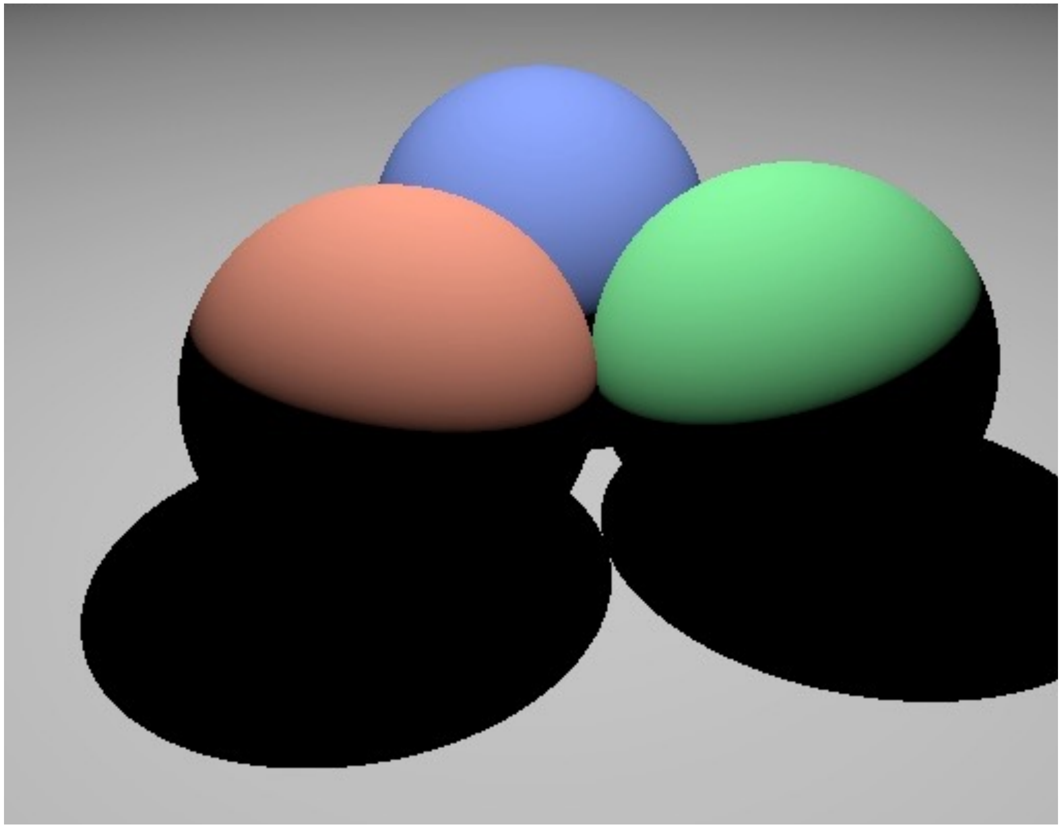
- Don't fall victim to one of the classic blunders:



- What's going on?
  - hint: at what *t* does shadow ray intersect the surface?

# Shadow rounding errors

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

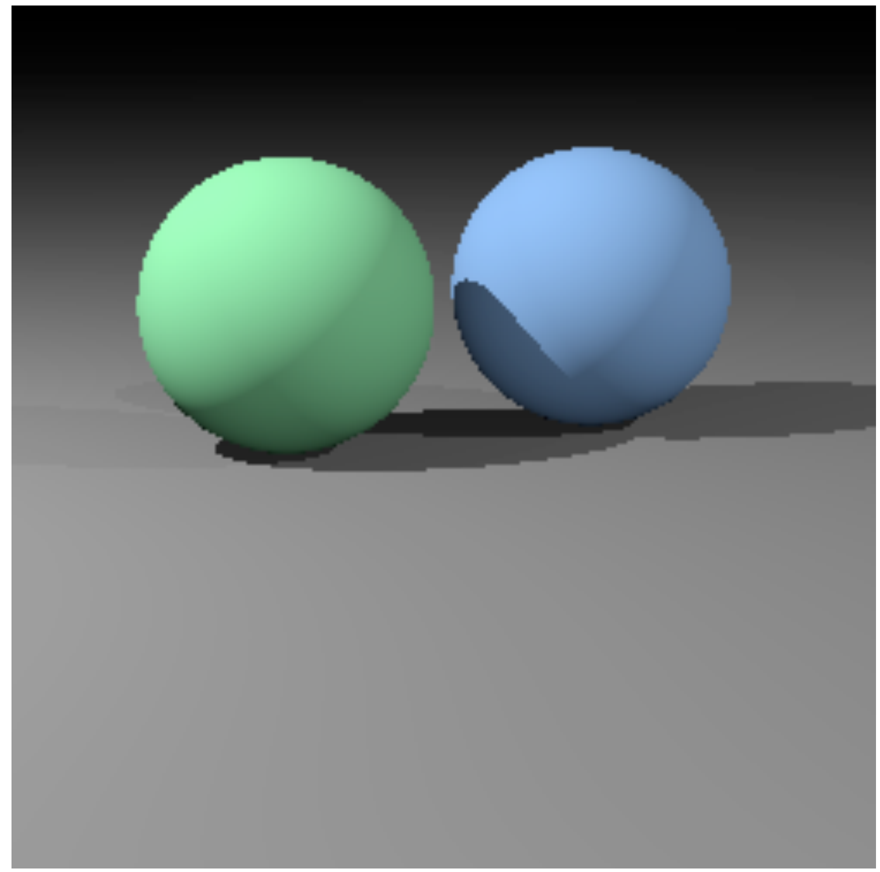


# Multiple lights

- 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…

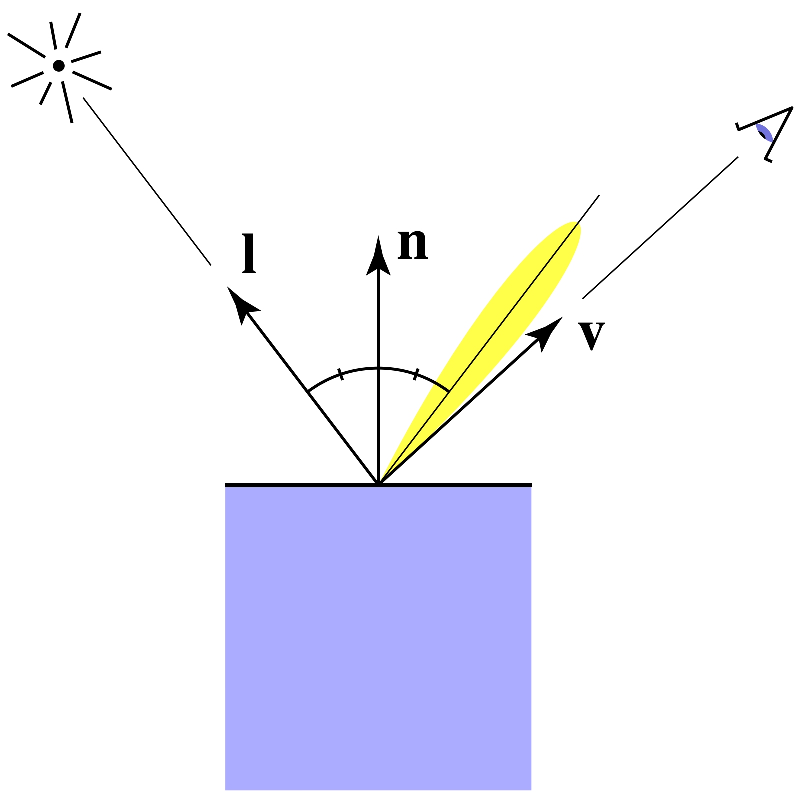
# Image so far

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



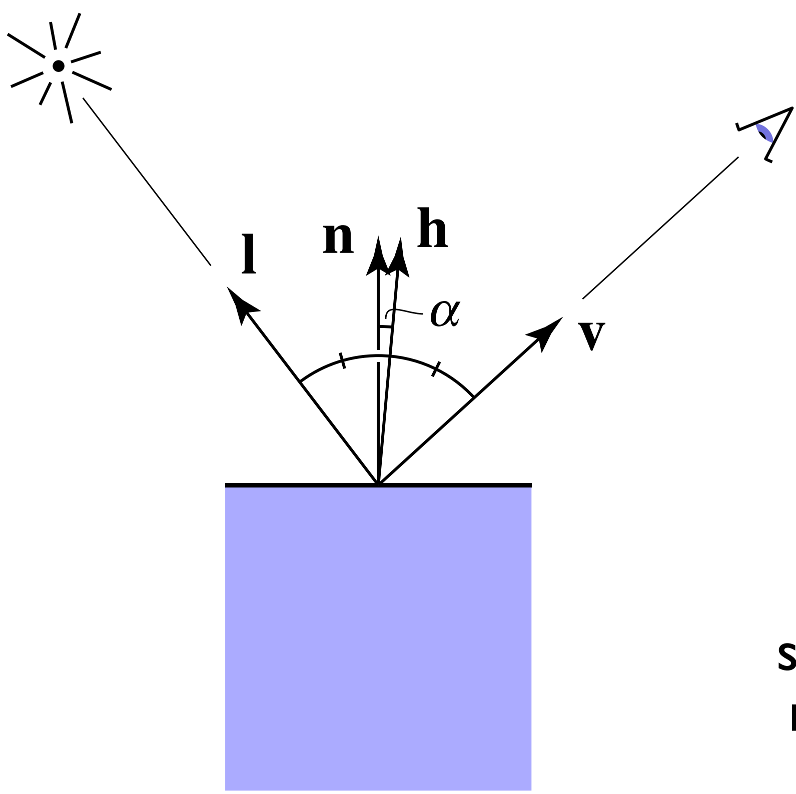
# Specular shading (Blinn-Phong)

- 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




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

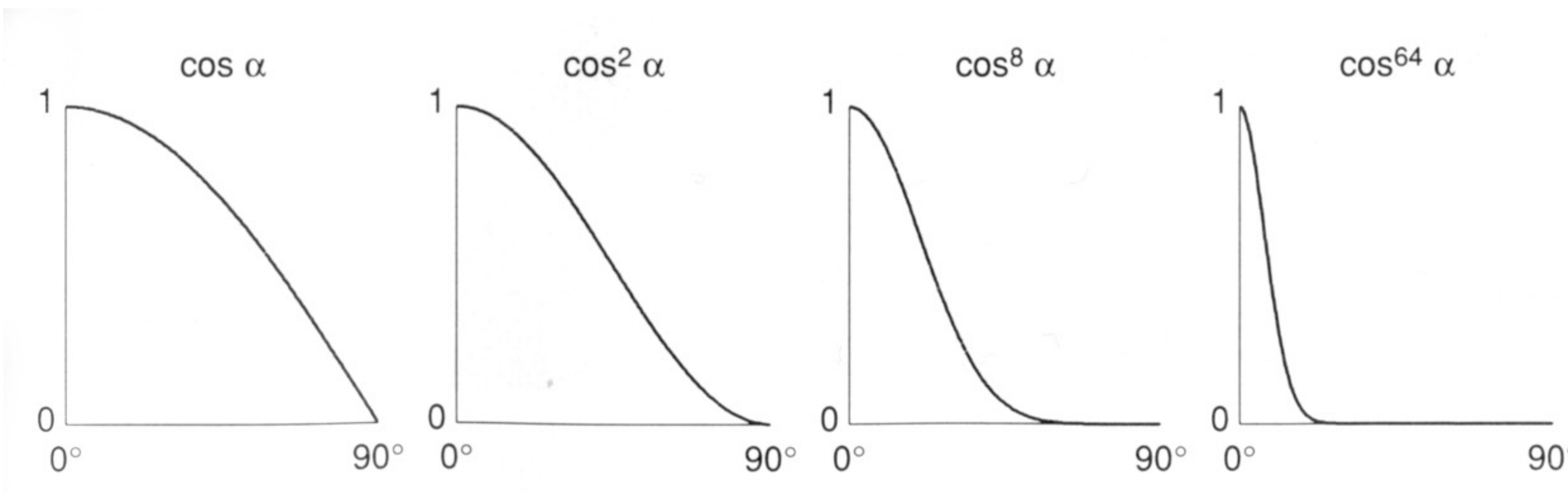
$$L_s = k_s\,(I/r^2)\max(0, \cos\alpha)^p = k_s\,(I/r^2)\max(0, \mathbf{n} \cdot \mathbf{h})^p$$

$L_s$ ↑ specularly reflected light

$k_s$ ↑ specular coefficient

# Phong model—plots

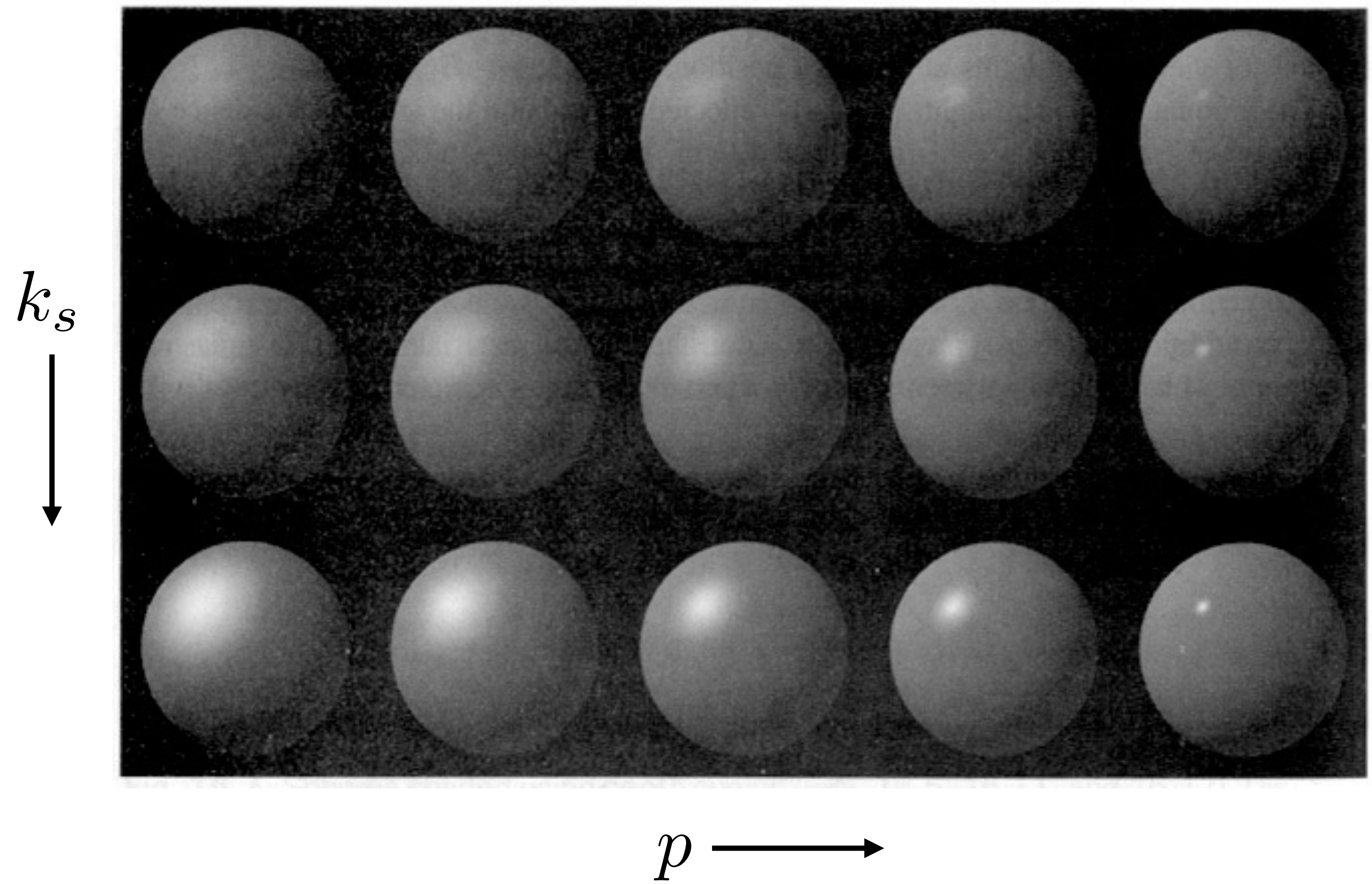
- Increasing *p* narrows the lobe




[Foley et al.]

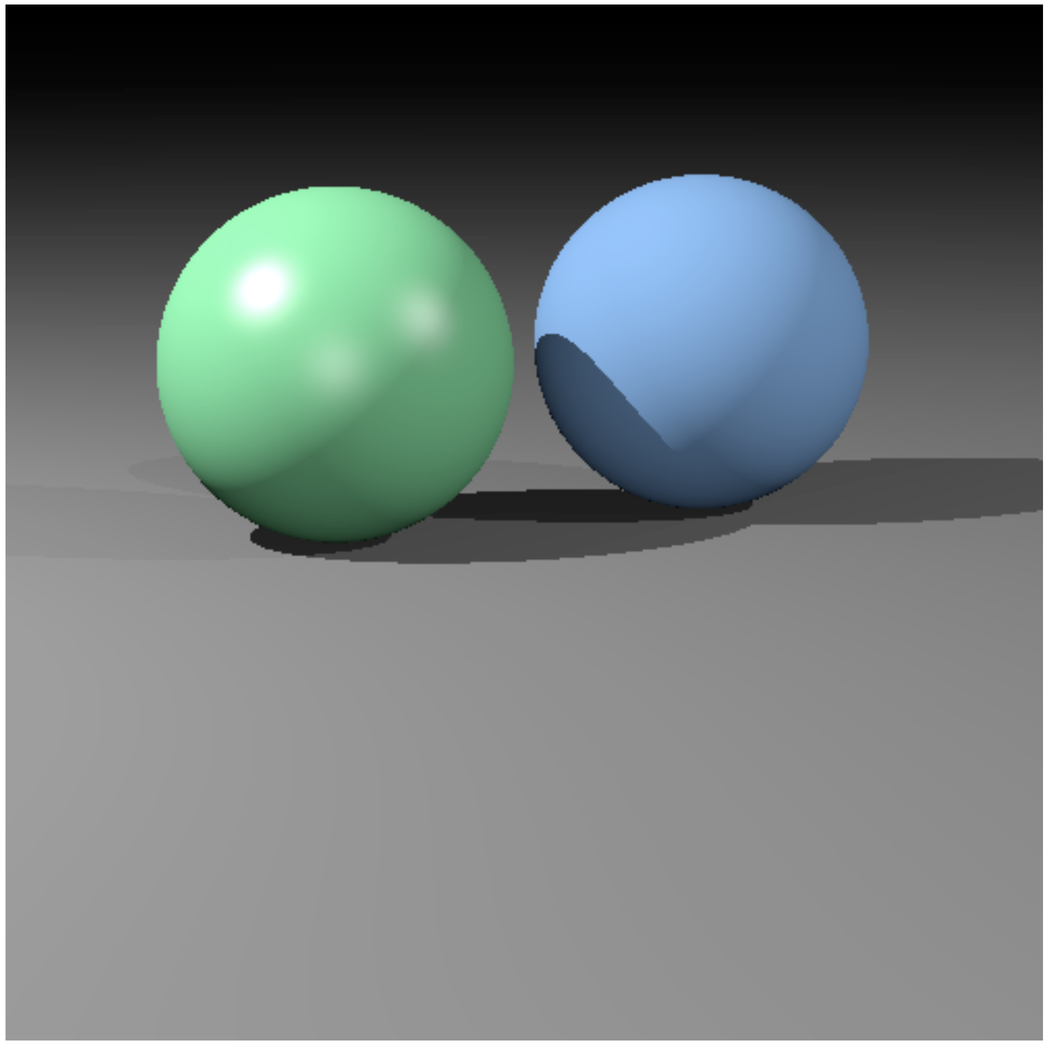
# Specular shading

- Blinn-Phong



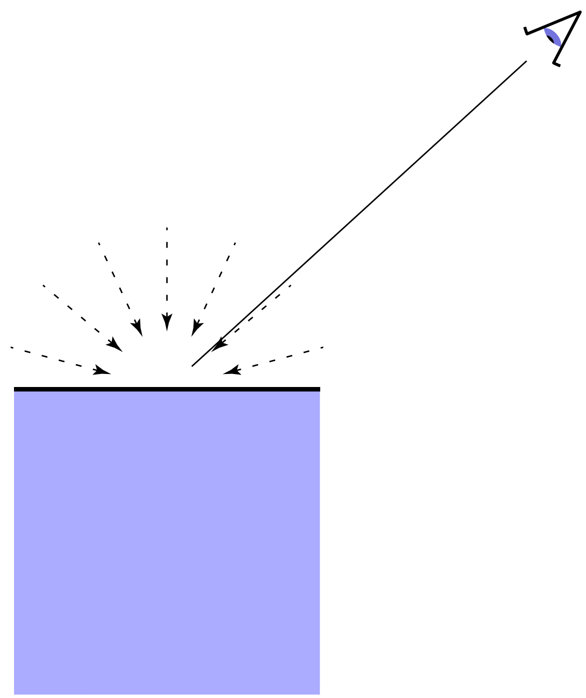

[Foley et al.]

# 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

$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p
\end{aligned}
$$

- The final result is the sum over many lights

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

$$
L = k_a\, I_a + \sum_{i=1}^{N} \Big[ k_d\, (I_i/r_i^2) \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s\, (I_i/r_i^2) \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \Big]
$$