CSC 433/533
Computer Graphics

Alon Efrat
Credit: Joshua Levine

Lecture 10
Ray Tracing 2

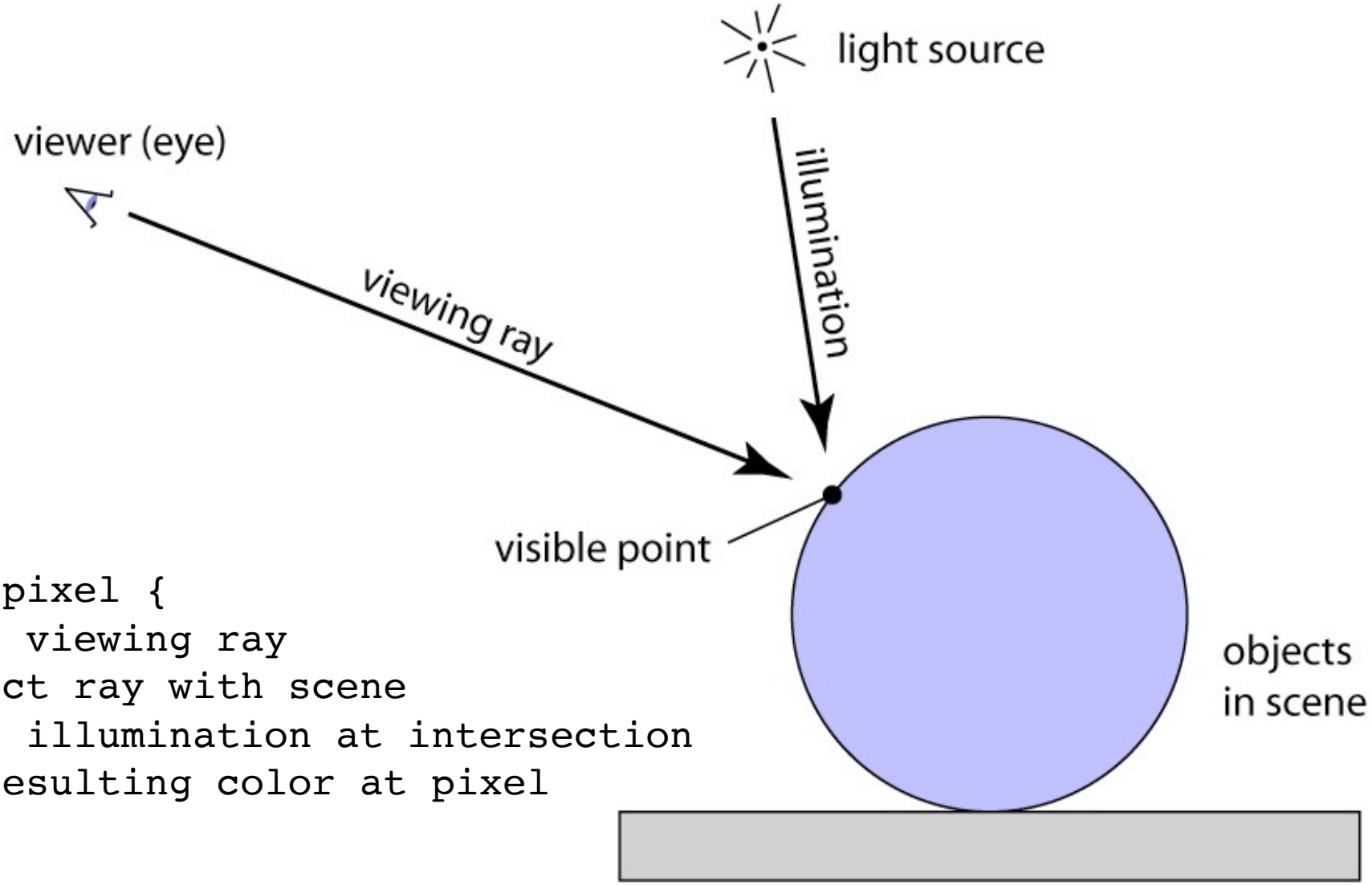
Oct 1, 2020

Today's Agenda

- Reminders:
  - A03, questions?
- Goals for today:
  - Discuss shapes
  - Introduce lighting and shading

Last Time

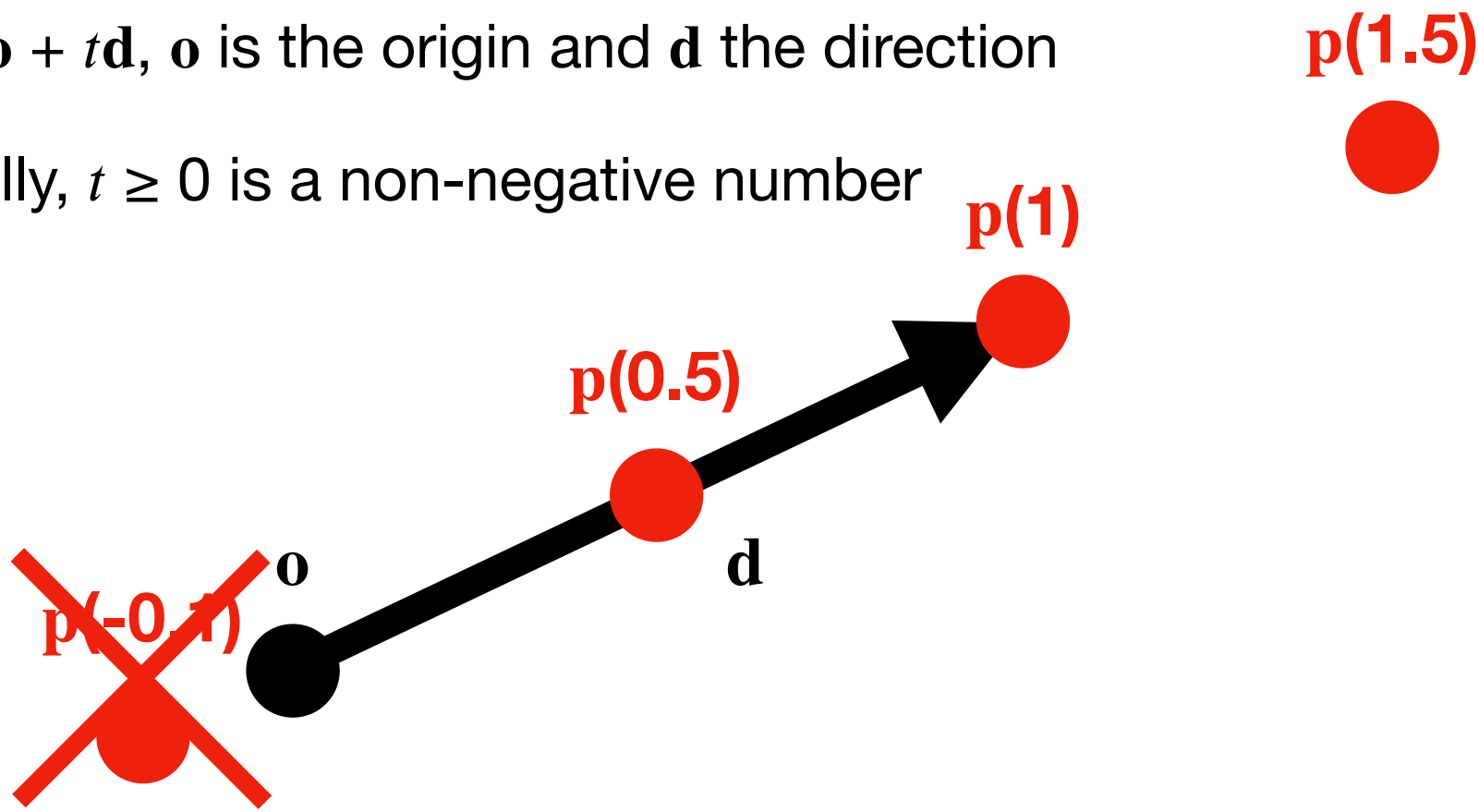
Ray Tracing Algorithm




```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Mathematical Description of a Ray

- Rays define a family of points, $\mathbf{p}(t)$, using a **parametric** definition
- $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$, $\mathbf{o}$ is the origin and $\mathbf{d}$ the direction
- Typically, $t \geq 0$ is a non-negative number



Intersecting Objects

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

From Planes to Triangles

- Given 3 points $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ on the triangle, can we define the plane of it?
- Recall: a plane is defined by a point $\mathbf{a}$ and a normal $\mathbf{n}$
- How to define the normal?
  - $\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$

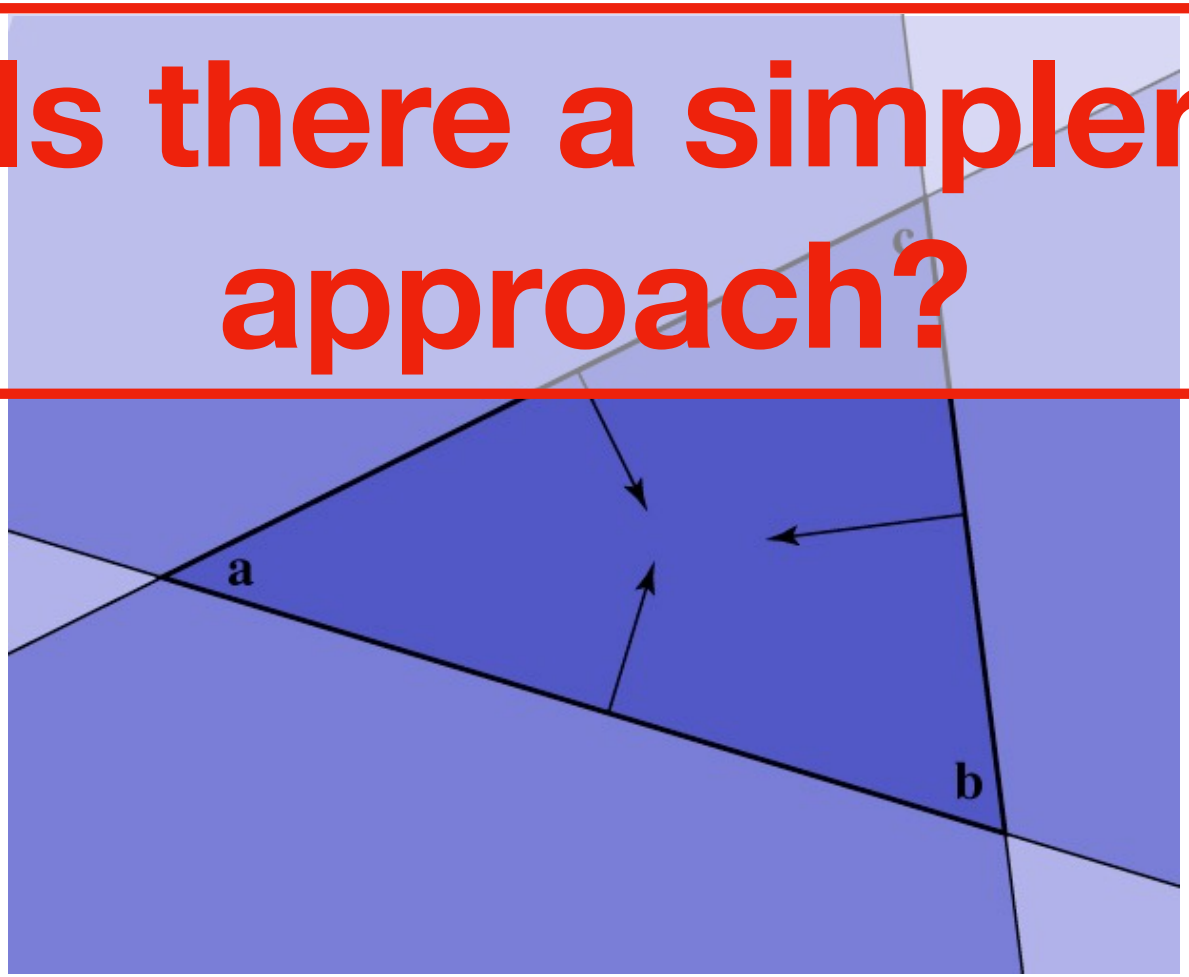
Ray-Triangle Intersection

- One approach is to satisfy 3 conditions:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on the plane: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$
  - Must be inside the triangle! How?

Point In Triangle

- In plane, triangle is the intersection of 3 half spaces
- Can check that the point is on the same side of these half spaces (perhaps after a transformation)

Is there a simpler approach?



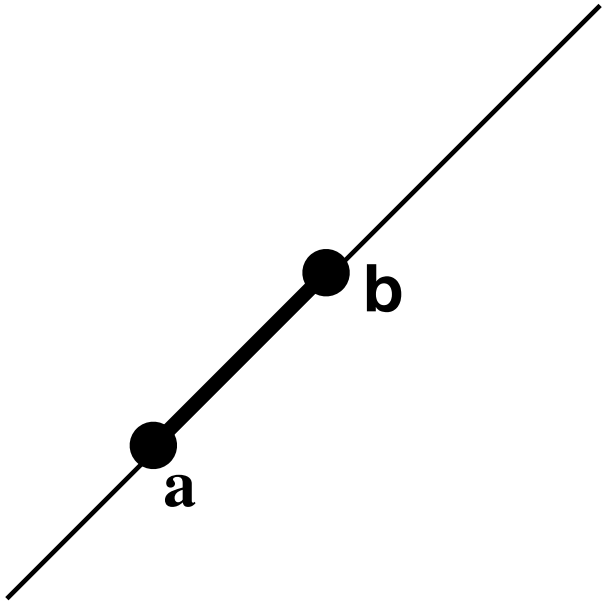
Warm-up

- Let, $\mathbf{a}$, $\mathbf{b}$ be points. We create a weighted combination of these points
-

$$\mathbf{p}(\alpha) = \alpha\mathbf{a} + (1-\alpha)\mathbf{b}$$

if $0 \leq \alpha \leq 1$ then $\mathbf{p}(\alpha)$ is on the segment **ab**

if $\alpha < 0$ or $\alpha > 1$ then $\mathbf{p}(\alpha)$ is not on the segment, but still the line passing through **a** and **b**



Same if we consider all combinations

$\alpha\mathbf{a} + \beta\mathbf{b}$ where $\alpha+\beta=1$

Now lets move to the weighted sum of 3 points **a,b,c**

Barycentric Coordinates

- A coordinate system to write all points $\mathbf{p}$ as a weighted sum of the vertices

  $\mathbf{p} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

  $\alpha + \beta + \gamma = 1$,

- Equivalently, α, β, γ are the proportions of area of subtriangles relative total area, A

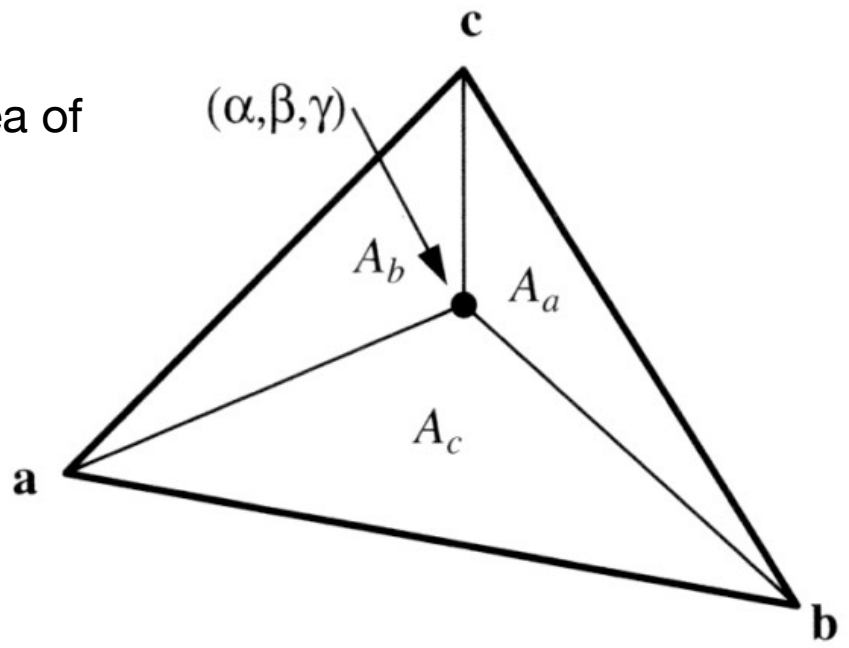
  $A_a / A = \alpha$

  $A_b / A = \beta$

  $A_c / A = \gamma$

- Triangle interior test:

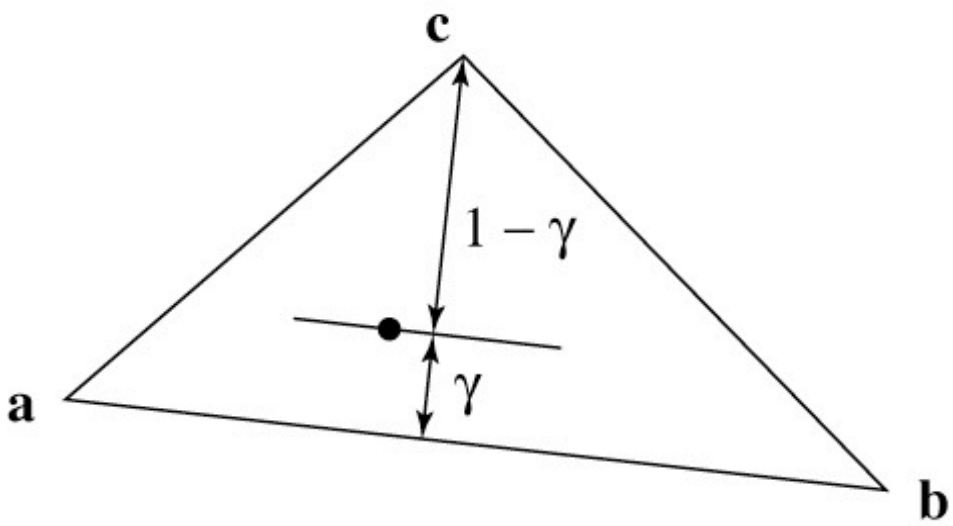
  $\alpha > 0$, $\beta > 0$, and $\gamma > 0$

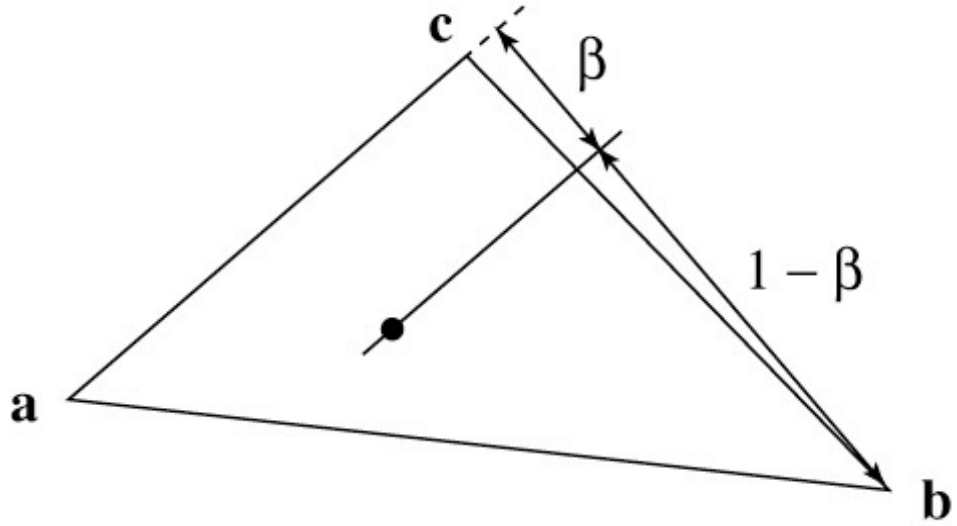


[Shirley 2000]

Barycentric Coordinates

- Also related to distances





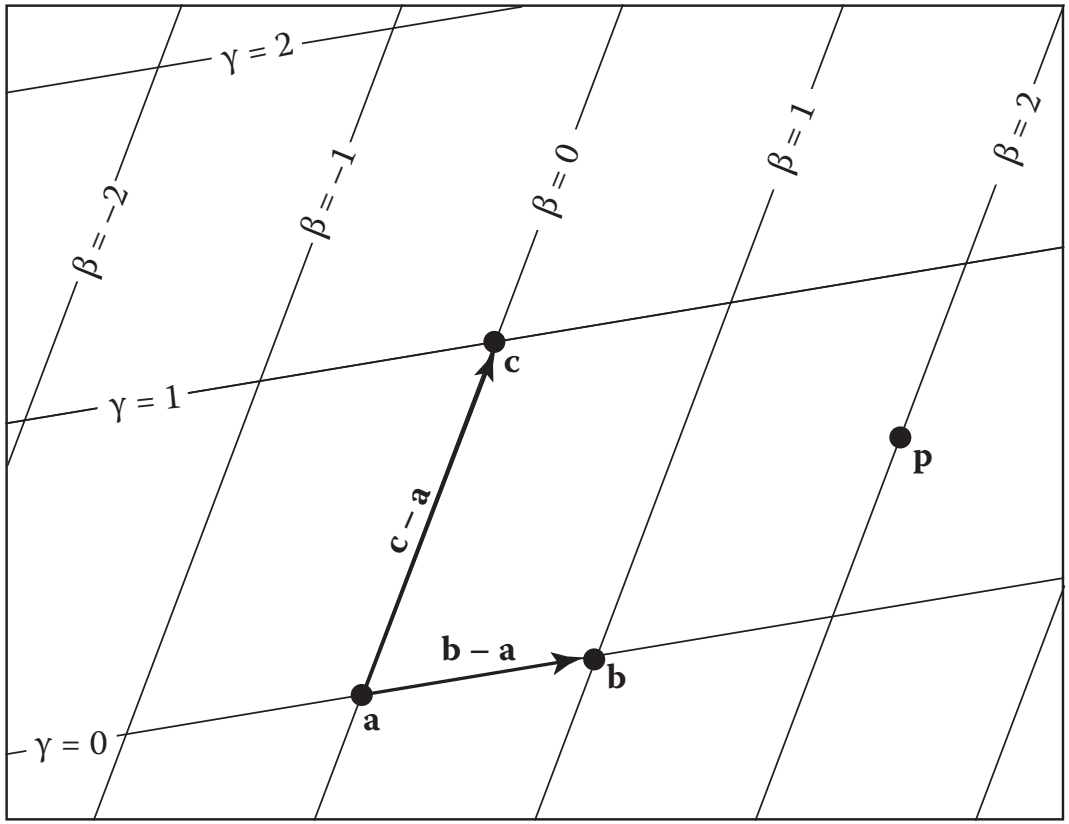
- And, they provide a basis relative to the edge vectors

$$\alpha = 1 - \beta - \gamma$$

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

Barycentric Coordinates

- This basis defines the plane of the triangle



- In this view, the triangle interior test becomes:

$$\beta > 0,\ \gamma > 0,\ \beta + \gamma \leq 1$$

Barycentric Ray-Triangle Intersection

- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be in the triangle: $\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$

- So, set them equal and solve for t, β, γ:

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

- This is possible to solve because you have 3 equations and 3 unknowns

Barycentric Ray-Triangle Intersection

$$\mathbf{o} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

$$\beta(\mathbf{a} - \mathbf{b}) + \gamma(\mathbf{a} - \mathbf{c}) + t\mathbf{d} = \mathbf{a} - \mathbf{o}$$

$$\begin{bmatrix} \mathbf{a} - \mathbf{b} & \mathbf{a} - \mathbf{c} & \mathbf{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{a} - \mathbf{o} \end{bmatrix}$$

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_{\mathbf{o}} \\ y_a - y_{\mathbf{o}} \\ z_a - z_{\mathbf{o}} \end{bmatrix}$$

- Cramer's rule good fast way to solve this system
  (see Ch. 4 for details and the closed form expressions)
- If you'll need to use it, we will provide the code.

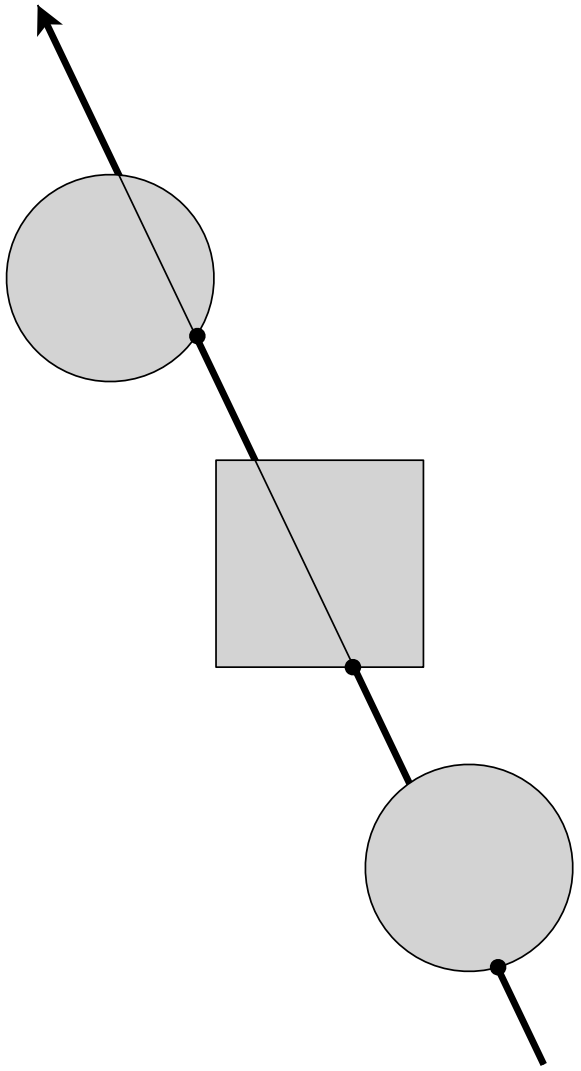
Intersection with Many Types of Shapes

- In a given scene, we also need to track which shape had the nearest hit point along the ray.

- This is easy to do by augmenting our interface to track a range of possible values for t, [t_{min}, t_{max}]:

```
intersect(eye, dir, t_min, t_max);
```

- After each intersection, we can then update the range



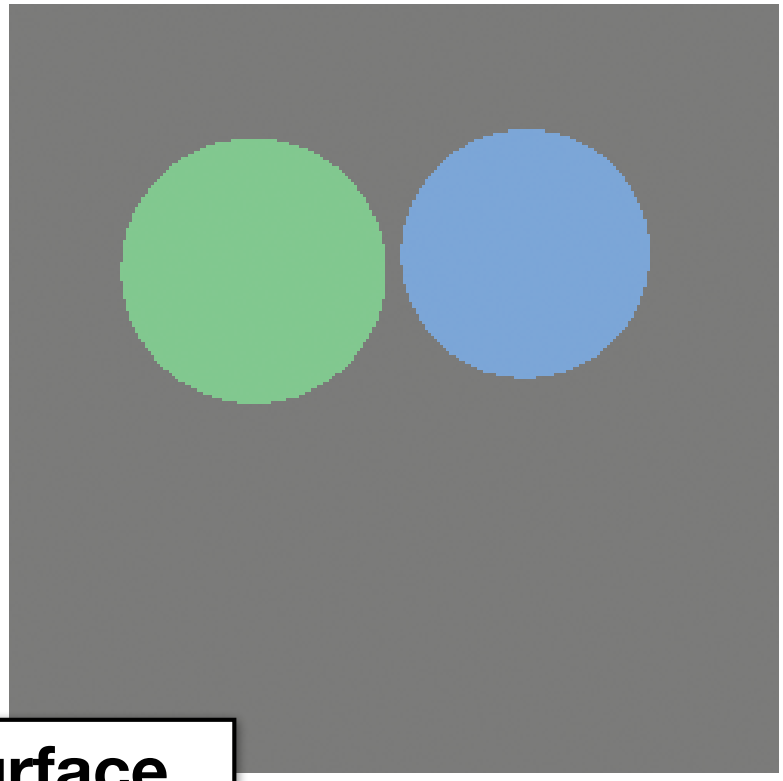
Illumination

```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

Our images so far

- With only eye-ray generation and scene intersection

```
for each pixel p in Image {
  let hit_surf = undefined;
  ...

  scene.surfaces.forEach( function(surf) {
    if (surf.intersect(eye, dir, ...)) {
      hit_surf = surf;
      ...
    }
  });

  c = hit_surf.ambient;
  Image.update(p, c);
}
```

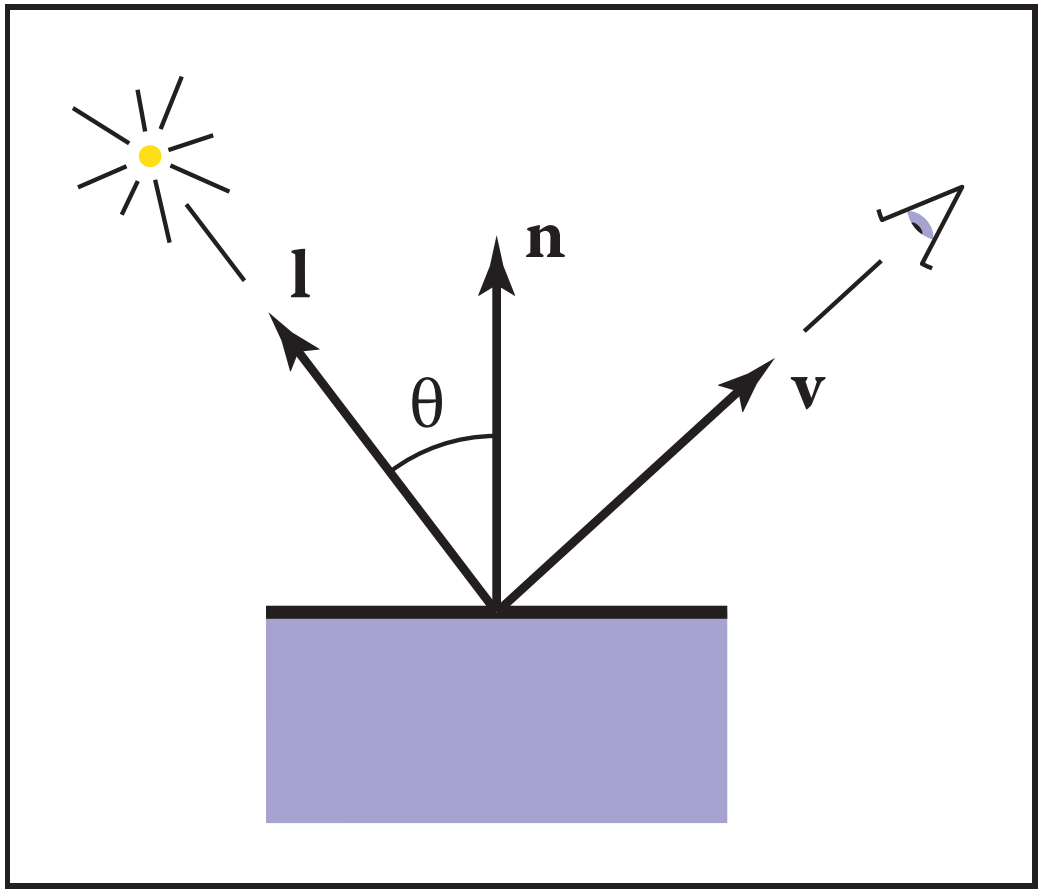


Each surface storing a single ambient color

Shading

- Goal: Compute light reflected toward camera

- Inputs:
  - eye direction
  - light direction (for each of many lights)
  - surface normal
  - surface parameters (color, shininess, ...)

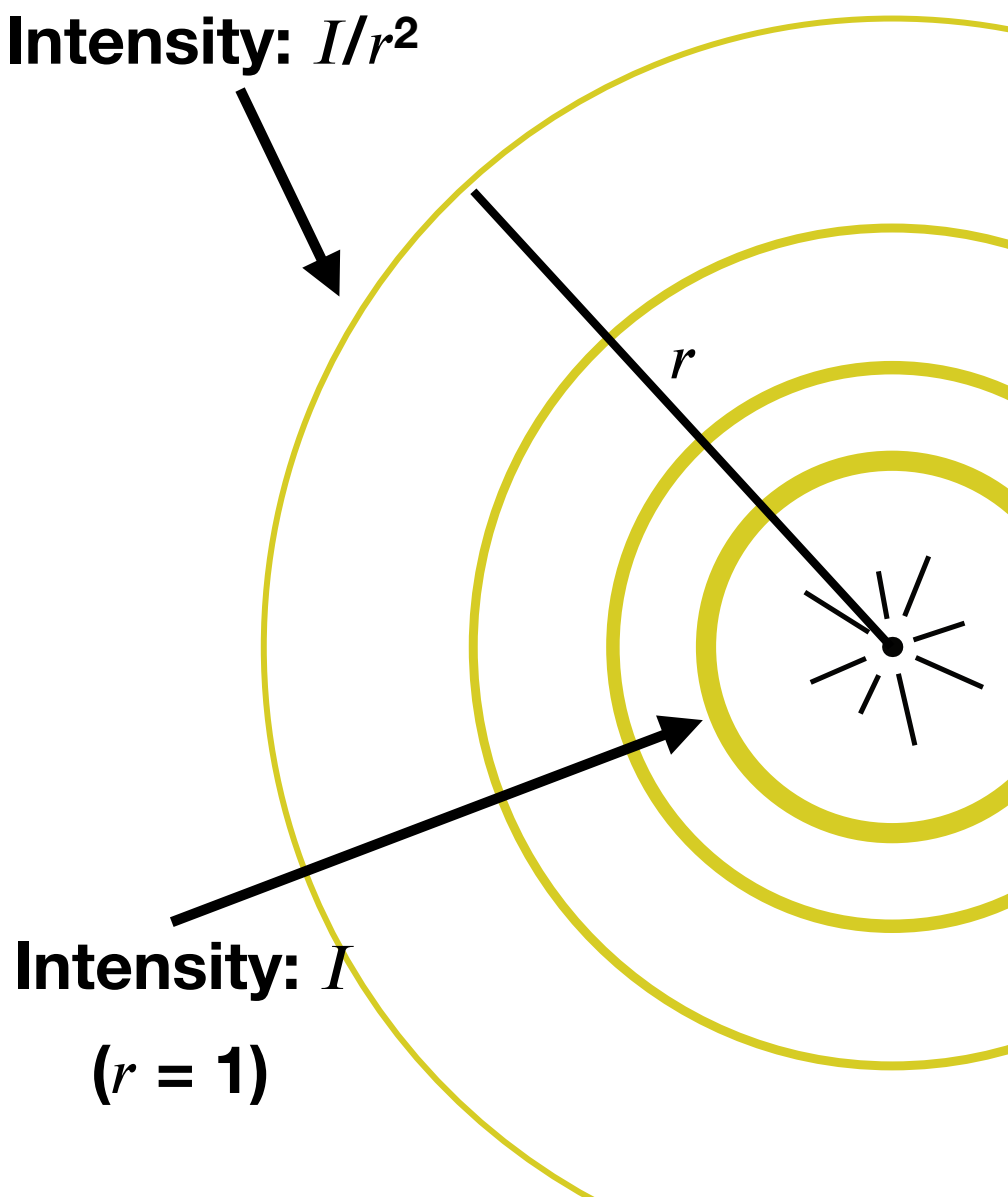


Normals

- The amount of light that reflects from a surface towards the eye depends on orientation of the surface at that point
- A **normal vector** describes the direction that is orthogonal to the surface at that point
- What are normal vectors for planes and triangles?
  - $\mathbf{n}$, the vector we already were storing!
- What are normal vectors for spheres?
  - Given a point $\mathbf{p}$ on the sphere $\mathbf{n} = (\mathbf{p} - \mathbf{c}) / \|\mathbf{p} - \mathbf{c}\|$

Light Sources

- There are many types of possible ways to model light, but for now we'll focus on **point lights**
- Point lights are defined by a position $\mathbf{p}$ that irradiates equally in all directions
- Technically, illumination from real point sources falls off relative to distance squared, but **<u>we will ignore this for now.</u>**



Shading Models

Ambient `shading" and Albedo

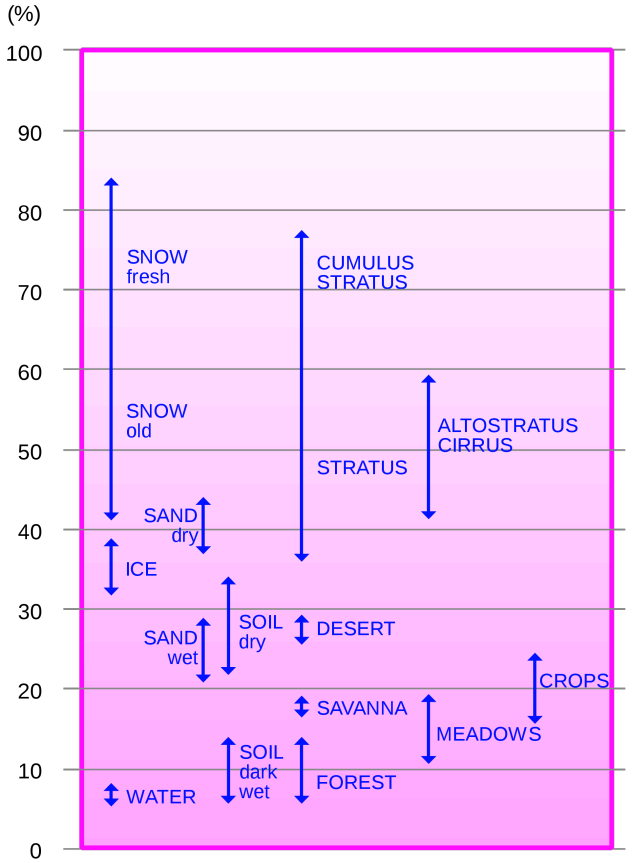
Ambient light - has no particular direction.

Every material has 3 coefficient describing the percentage of white light that it reflects in R, in G, and in B. The location of viewer and the location of the light-source are irrelevant.

When describing a scene to (Say) OpenGL, WebGL, processing.org etc, we could specify for every light source how much intensity it should emits (in RGB).

If a sphere has Ambient coefficient (0.1, 0.9, 0.9) it will look very dim in Red light, but bright in Blue or Green light.

If illuminated by while light, then the sphere color is cyan.



Albedo coefficient is a physical term that is related, but not identical

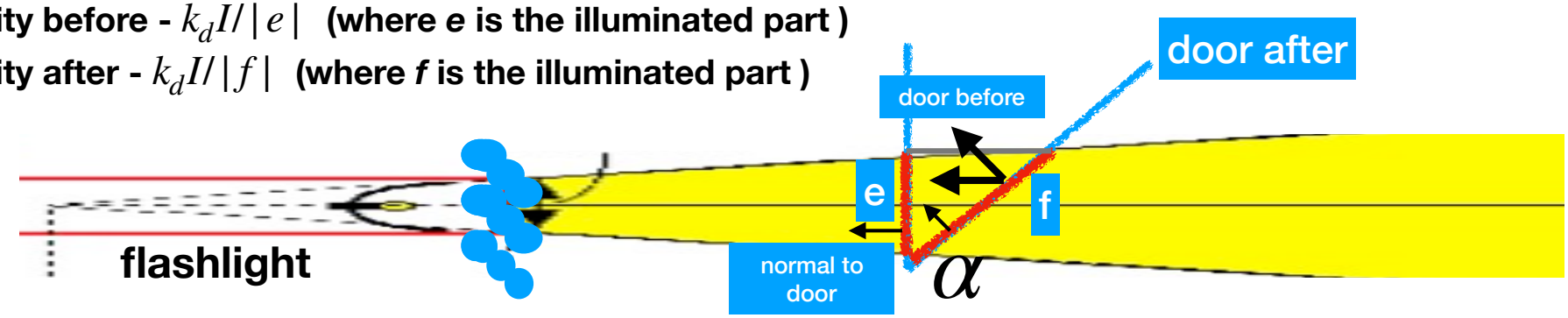
Lambertian (Diffuse) Shading

- Lets think about the intensity of the light in terms of energy reflected toward the viewer.

- Consider a door illuminated by a flashlight (see below).
- Lets think about the intensity reflected from the door as the door rotates.

- Let I denote the total light energy that the flashlight emits per second (can think about it as #photons / second)
- The Intensity of the light reflected from the door is $\frac{k_d \cdot I}{\textit{The area of the illumninated portion}}$.

- Intensity before - $k_d I/|e|$ (where *e* is the illuminated part)
- Intensity after - $k_d I/|f|$ (where *f* is the illuminated part)



$\frac{|e|}{|f|} = \cos\alpha$ or $|f| = |e|\frac{1}{\cos\alpha}$ Implying that $\frac{k_d I}{|f|} = \frac{k_d I}{|e|\frac{1}{\cos\alpha}} = \frac{k_d I}{|e|}\cos\alpha$

But I/|e| is the intensity of the reflected light for the before at the "before" stage.

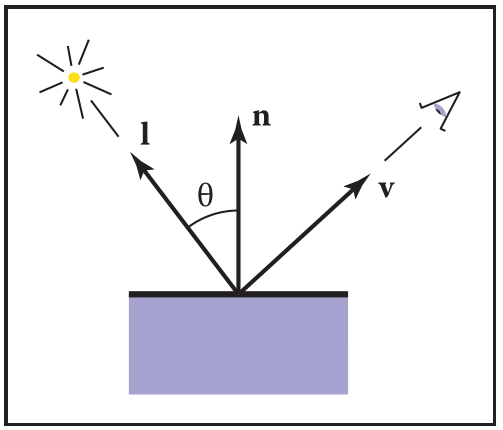
Conclusion - the intensity decrease by a factor of $\cos(\alpha)$

But $\cos(\alpha)$ is just the dot product of two vectors:

1) Normal of the door, and, 2) direction to the light source

Lambertian (Diffuse) Shading

- Simple model: amount of energy from a light source depends on the direction at which the light ray hits the surface



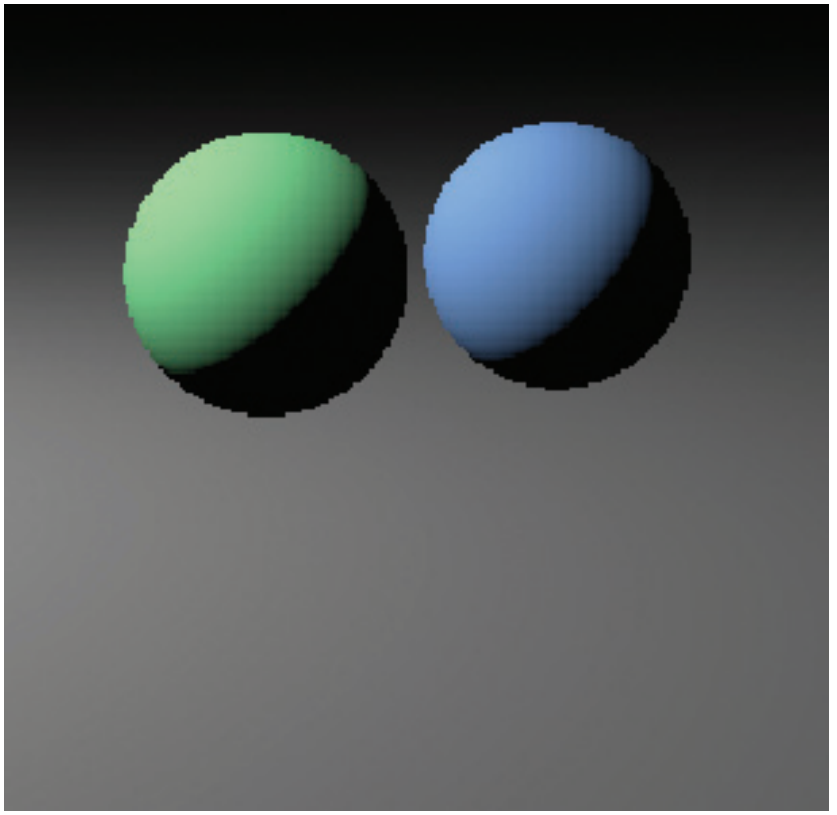
- Results in shading that is *view independent*

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

diffuse coefficient

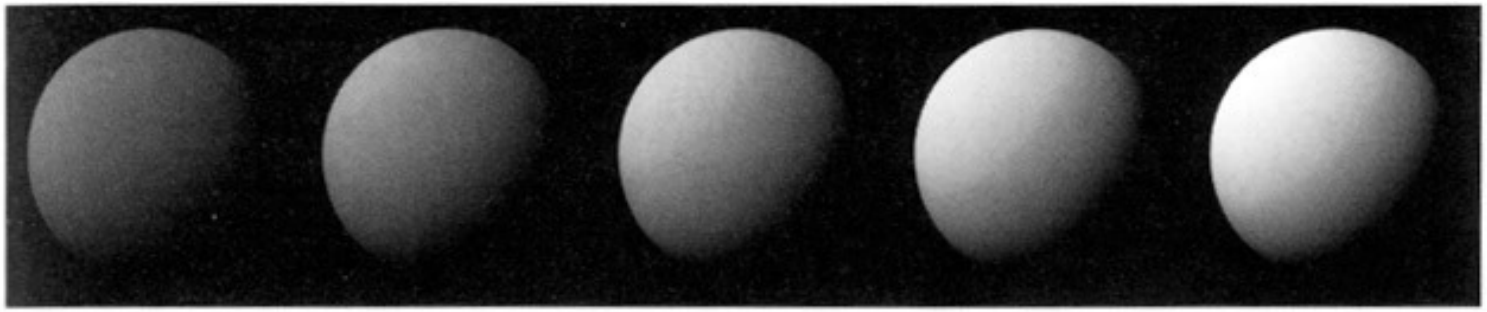
intensity/color of light

$\cos \theta$



Lambertian Shading

- k_d is a property of the surface itself (3 constants - one per each color channel)
- Produces matte appearance of varying intensities

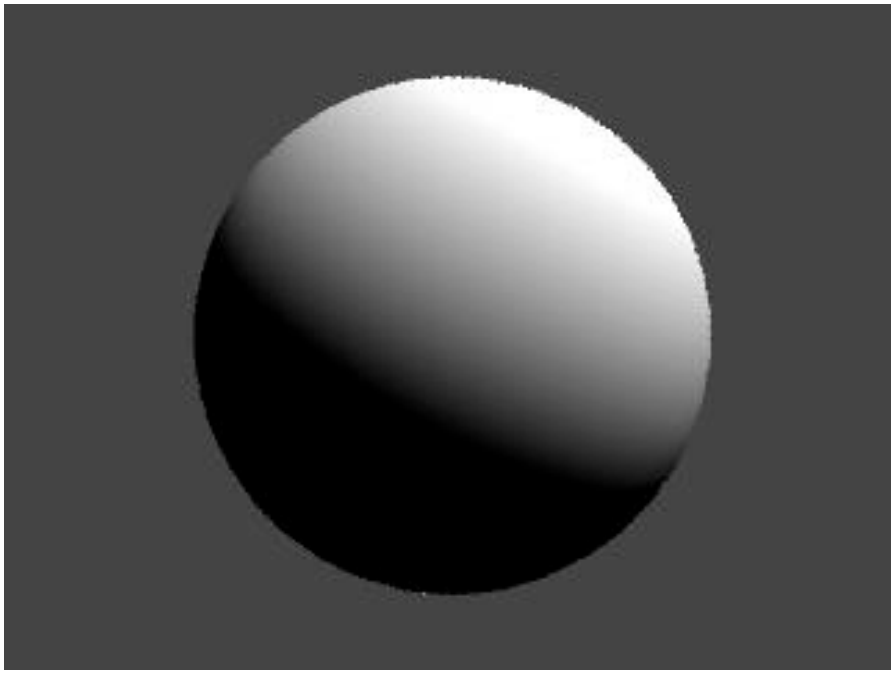


$k_d \longrightarrow$

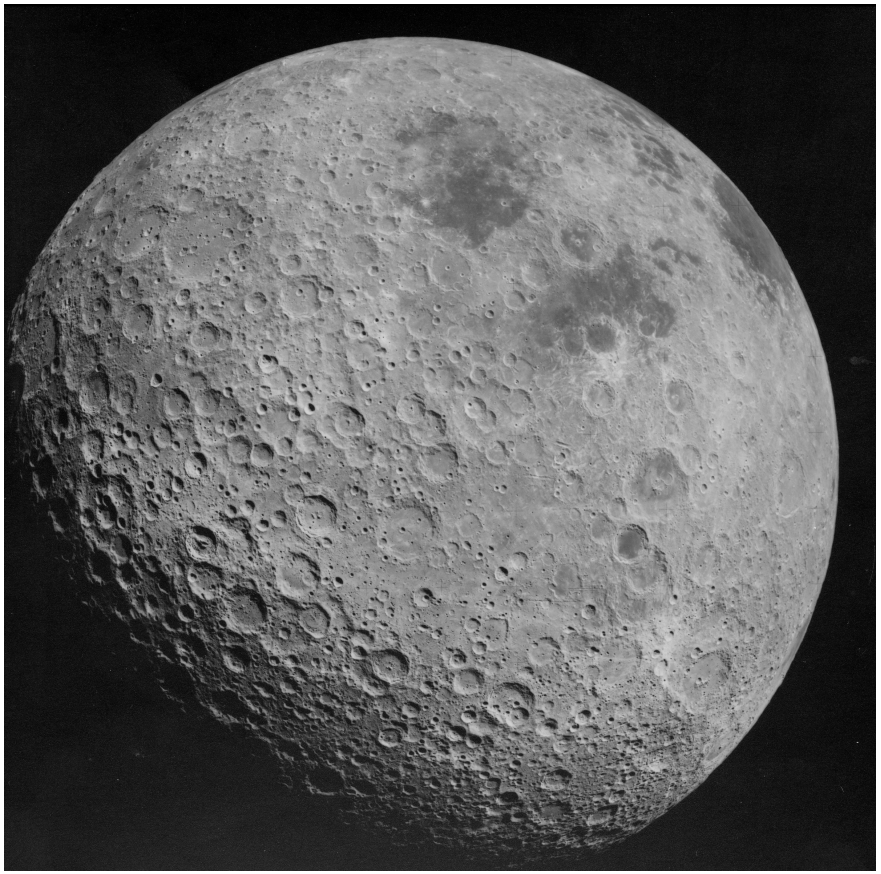
[Foley et al.]

The moon paradox

- why don’t we see this gradual shading when looking at the moon ?
-

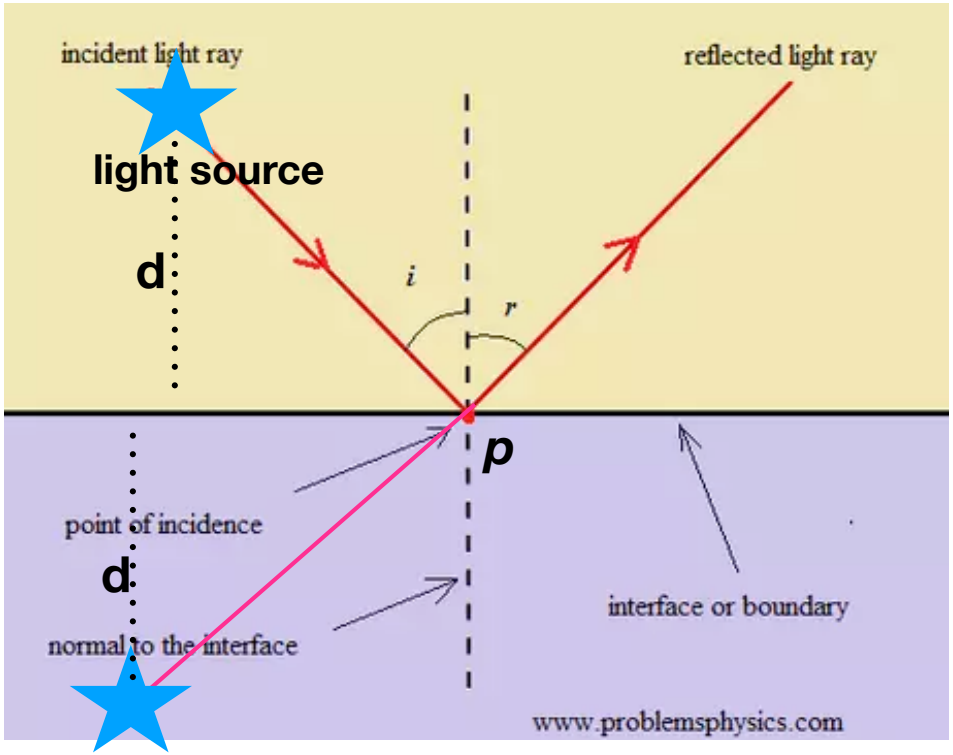


vs



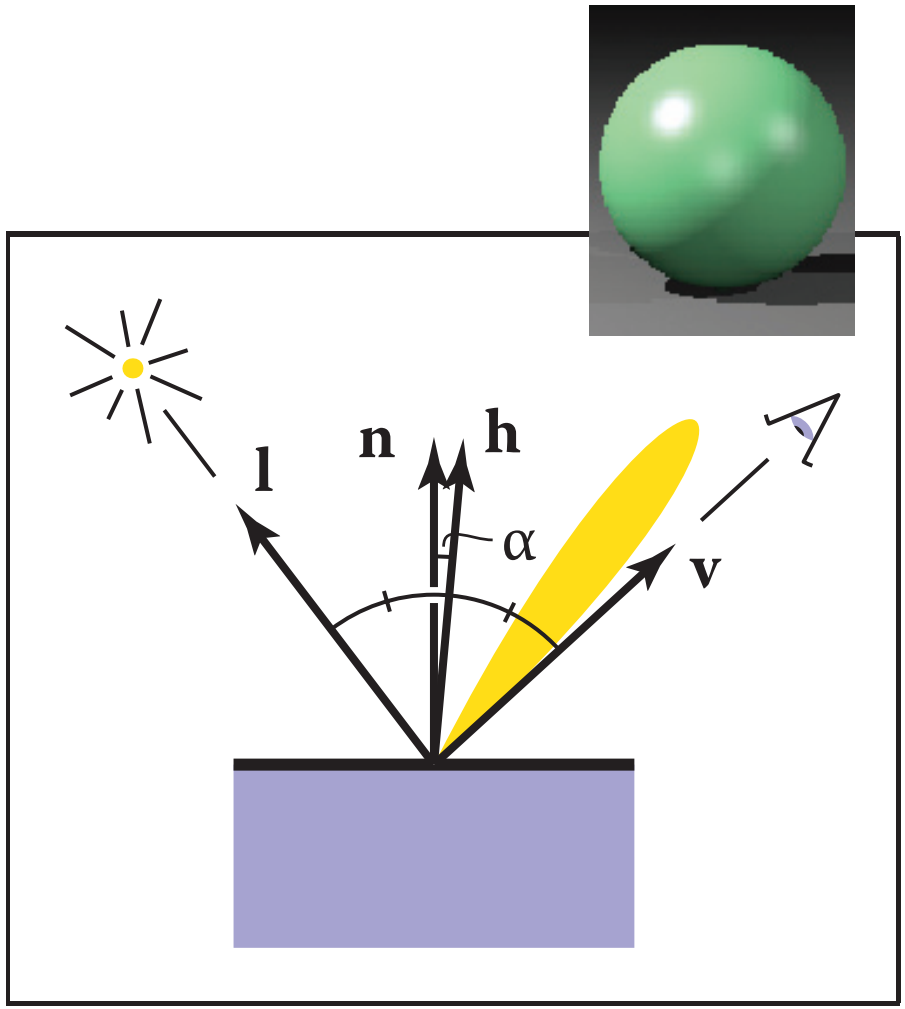
Toward Specular Shading: Perfect Mirror

- Many real surfaces show some degree of shininess that produce **specular** reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal

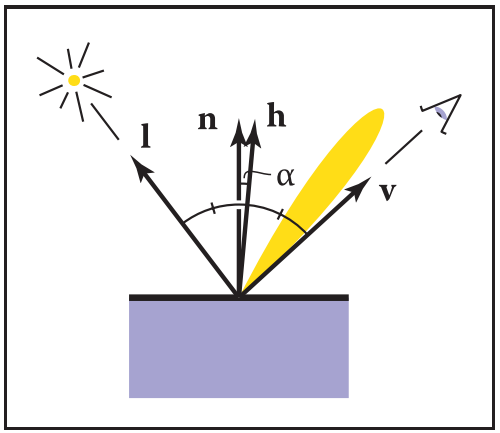


Blinn-Phong (Specular) Shading

- Many real surfaces show some degree of shininess that produce specular reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal



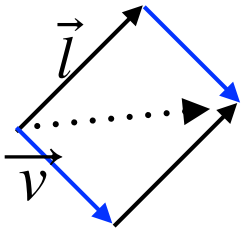
Blinn-Phong (Specular) Shading



- For any two **unit** vectors $\vec{v}$, $\vec{l}$, the vector is a bisector of the angle between these vectors.

- Normalize $\mathbf{v} + \mathbf{l}$

$$\mathbf{h} = (\mathbf{v} + \mathbf{l}) / \|\mathbf{v} + \mathbf{l}\|$$

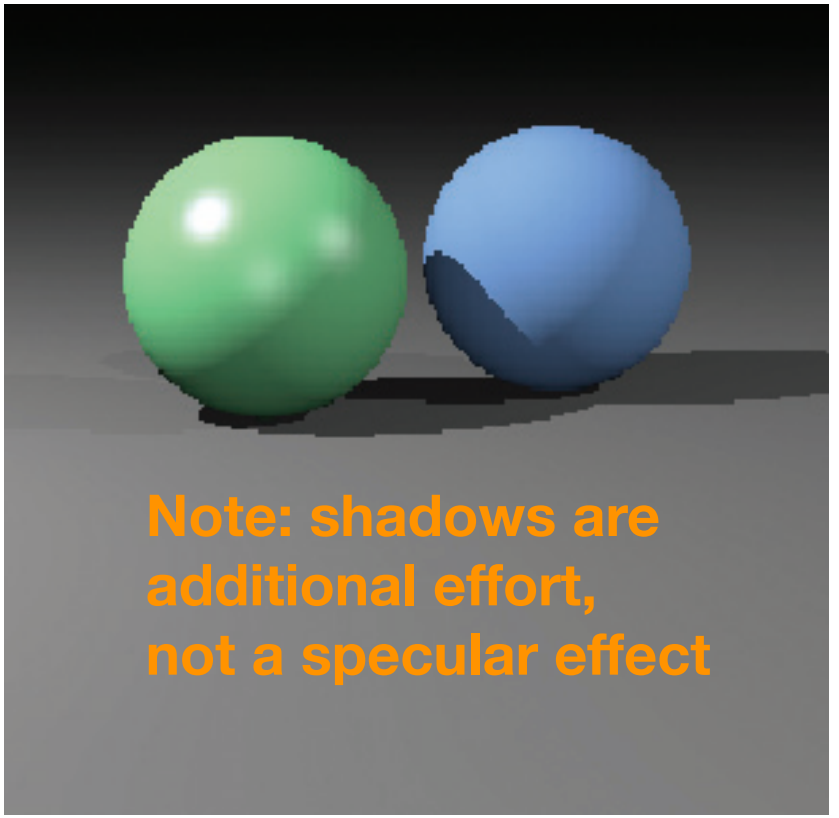


- In a perfect mirror, the 100% of the reflection occurs at the surface point where **h** is the normal **n**
- Diffuse reflection. Reflect large value for points where **h** is ``almost" **n**
- Phong heuristic:

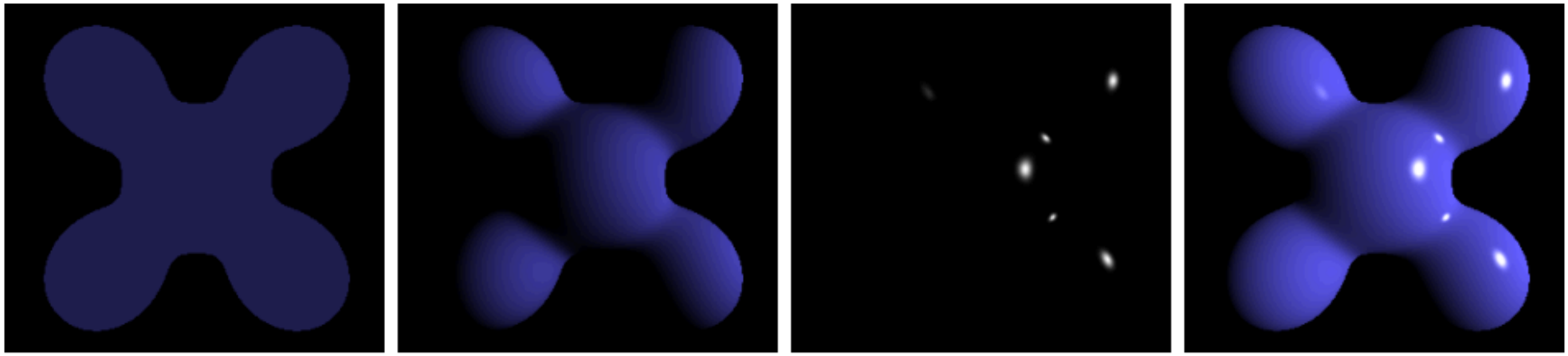
$$L_s = k_s I \max(0, \quad (\mathbf{n} \cdot \mathbf{h})^{p} \quad)$$

specular coefficient

Phong exponent



Blinn-Phong Decomposed

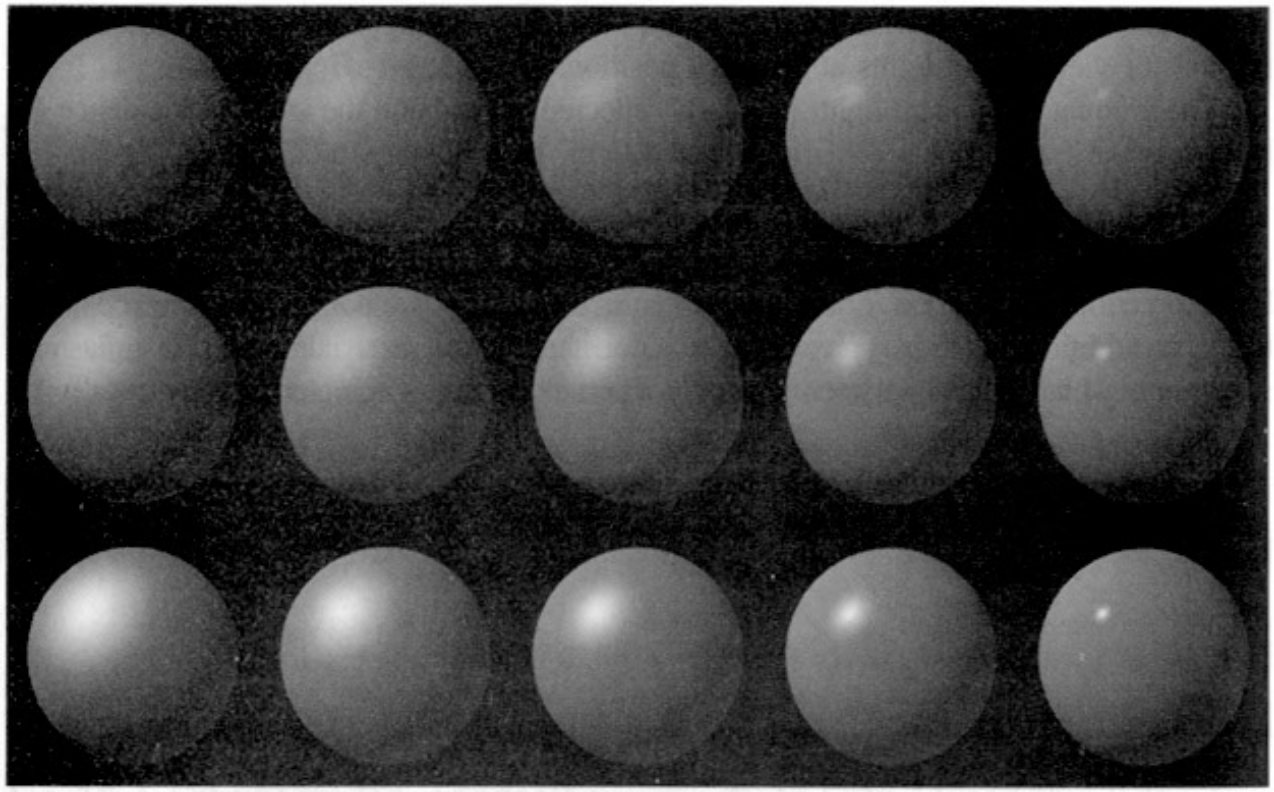


Ambient + Diffuse + Specular = Phong Reflection

https://en.wikipedia.org/wiki/Phong_shading

Blinn-Phong Shading

- Increasing p narrows the lobe
- This is kind of a hack, but it does look good



[Foley et al.]

Putting it all together

- Usually include ambient, diffuse, and specular in one model

$$L = L_a + L_d + L_s$$

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

- And, the final result accumulates for all lights in the scene

$$L = k_a I_a + \Sigma_i \left(k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l}_i) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p \right)$$

- Be careful of overflowing! You may need to clamp colors, especially if there are many lights.

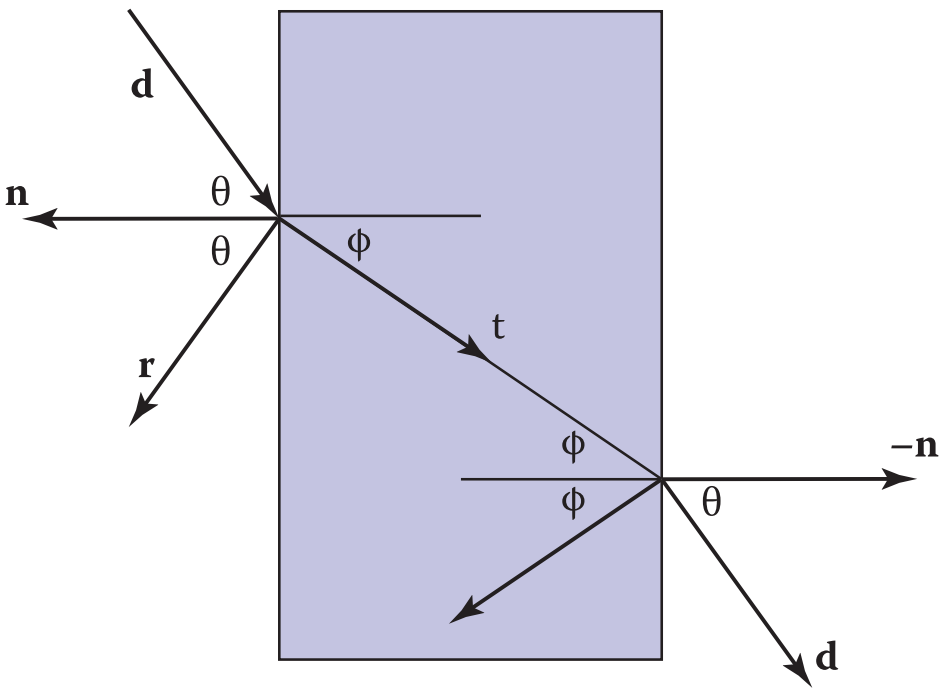
Simple Ray Tracer

```
function ray_cast(eye, dir, near, far) {
  let hit_surf = undefined;   let hit_rec = undefined;
  let t_min = 0;   let hit_t = Infinity;
  let color = background;    //default background color

  scene.surfaces.forEach( function(surf) {
    let intersect_rec = surf.hit(eye, dir, t_min, hit_t);
    if (intersect_rec.hit) {
      hit_surf = surf;
      hit_t = intersect_rec.t;
      hit_rec = intersect_rec;
    }
  });

  if (hit_surf !== undefined) {
    color = hit_surf.kA * Ia;
    scene.lights.forEach( function(light) {
      //compute l_i, h_i
      color = color + hit_surf.kD*I_i*max(0,n · l_i) + hit_surf.kS*I_i*max(0,n · h_i)^p;
    });
  }

  return color;
}
```

```
for each pixel p in Image {
  let [eye, dir] = camera.compute_ray(p);
  let c = ray_cast(eye, dir, 0, Infinity);
  image.update(p, c);
}
```

Snell's Law

- Governs the angle at which a refracted ray bends
- Computation based on refraction index of original medium, n, versus new index n_t
- $n_t \sin \theta = n \sin \phi$

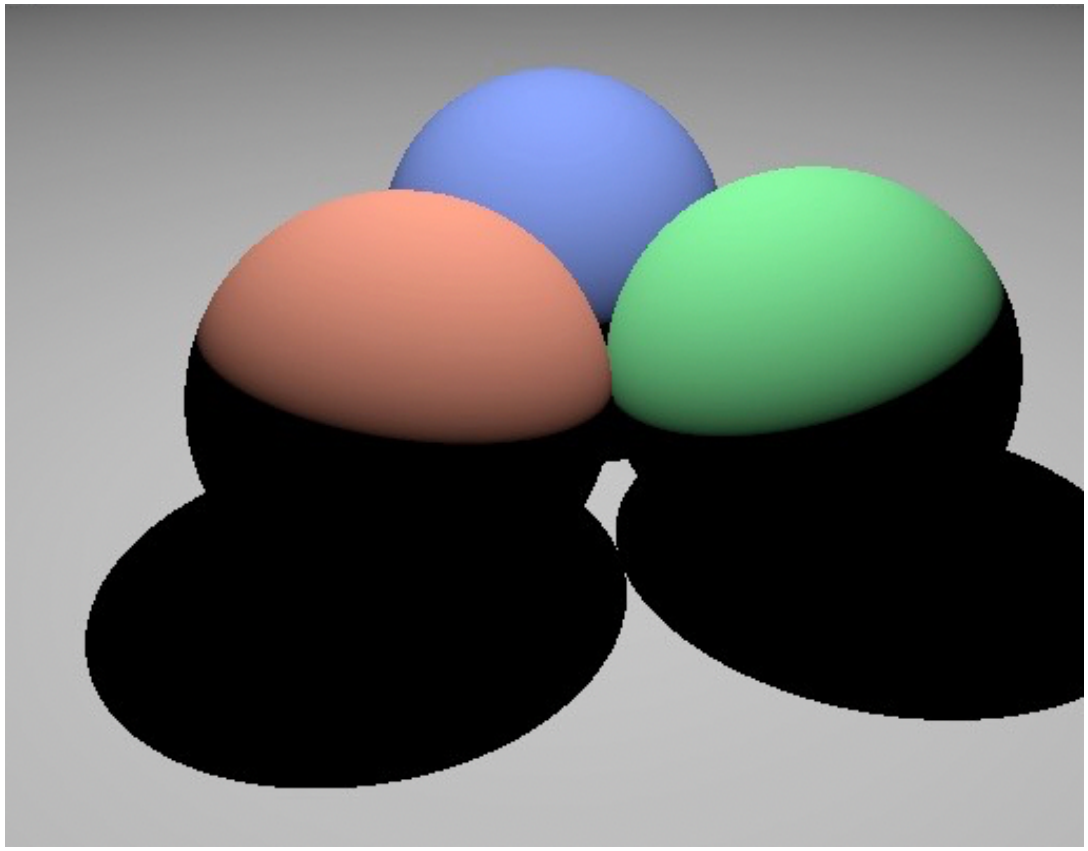



alamy stock photo
KC1E4Y
www.alamy.com

Recursive Ray Tracing

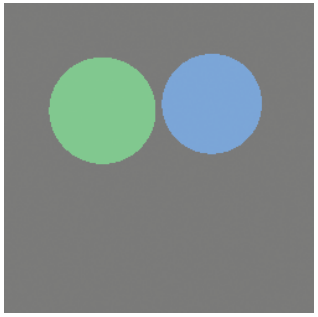
Shadows

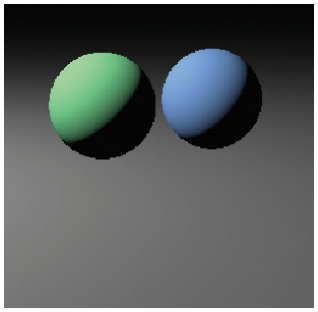
- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}

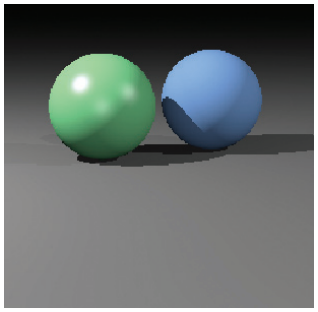


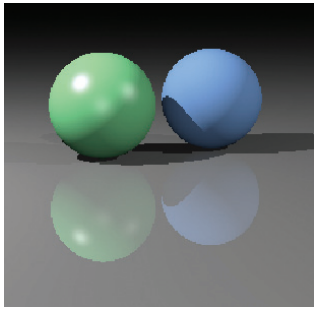
Recursive Ray Tracer

```
Color ray_cast(Ray ray, SurfaceList scene, float near, float far) {
  ...
  //initialize color;  compute hit_surf, hit_position;
  ...

  if (hit_surf is valid) {
    color = hit_surf.kA * Ia;









  }

  return color;
}
```





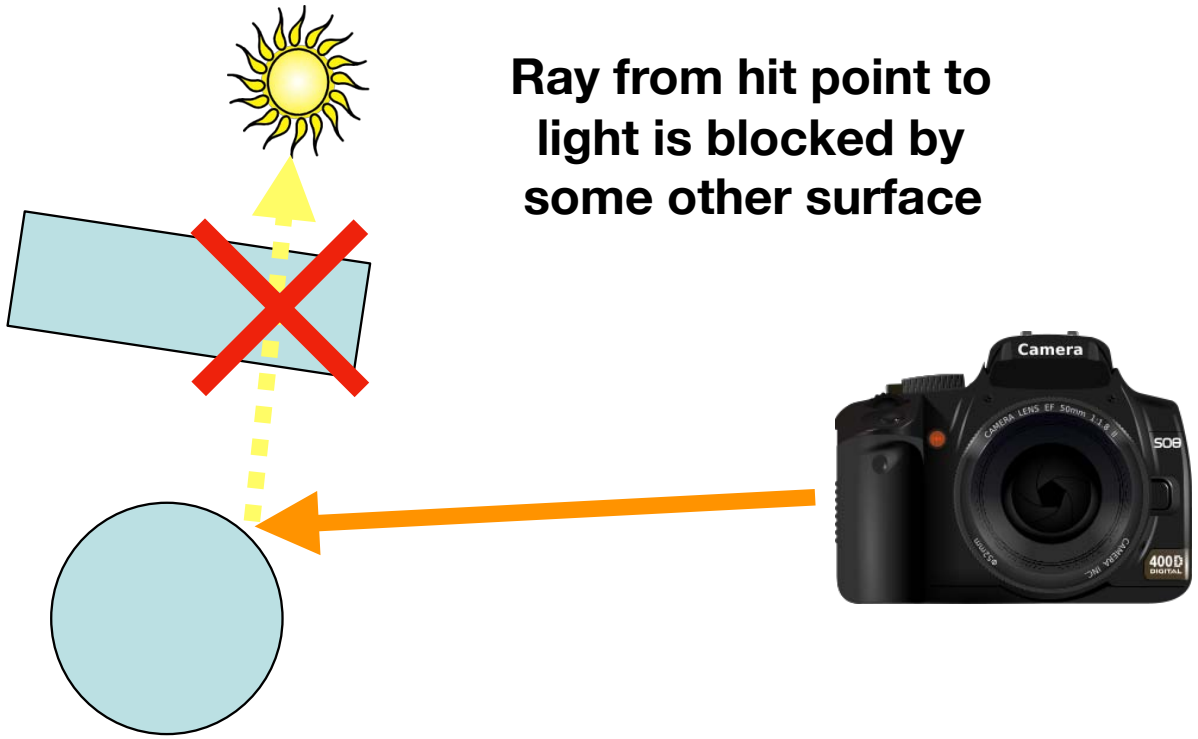




Shadows

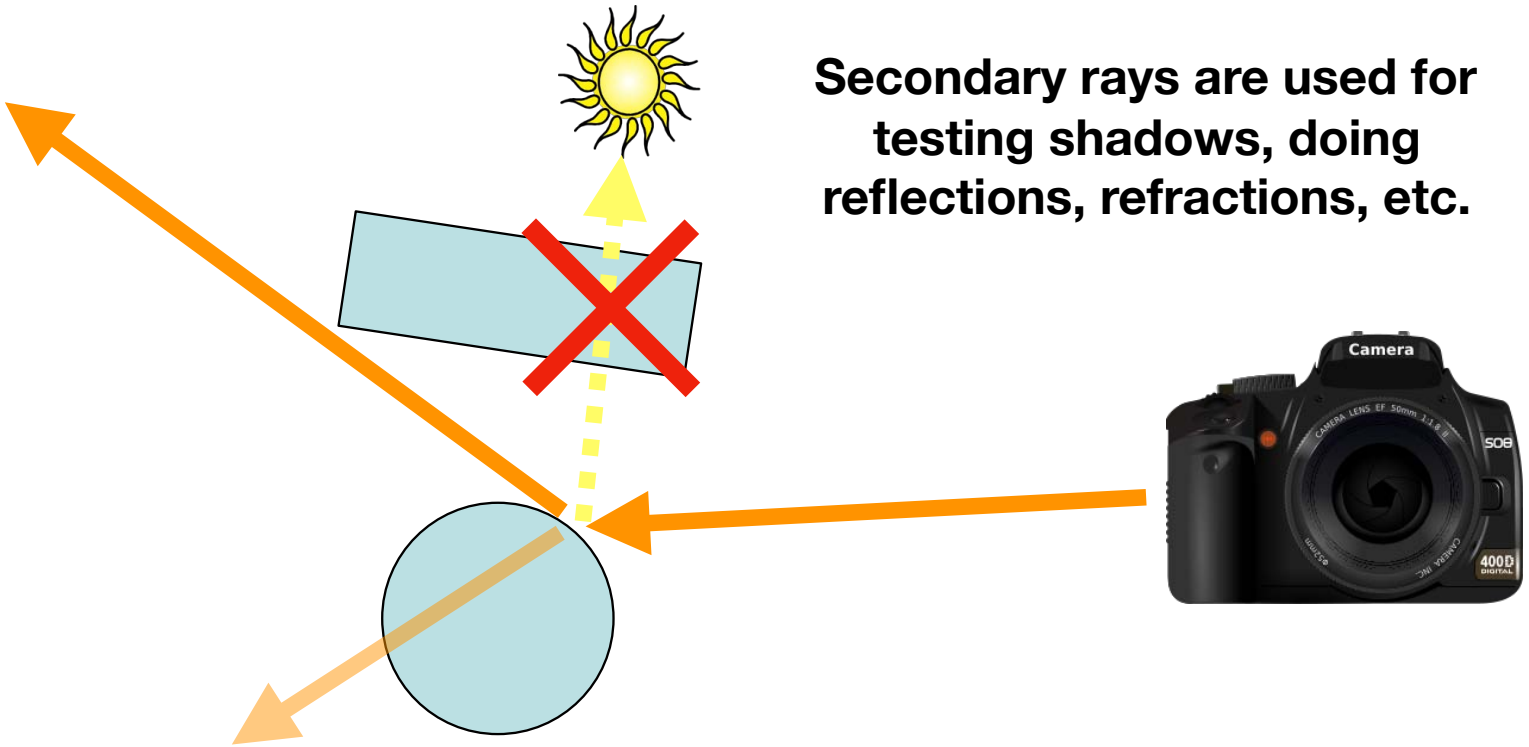
- Surface should only be illuminated if nothing blocks the light from hitting the surface

- This can be easily checked by intersecting a new ray with the scene!



Ray Casting vs Ray Tracing

- Ray casting: tracing rays from eyes only
- Ray tracing: tracing secondary rays



Secondary rays are used for testing shadows, doing reflections, refractions, etc.

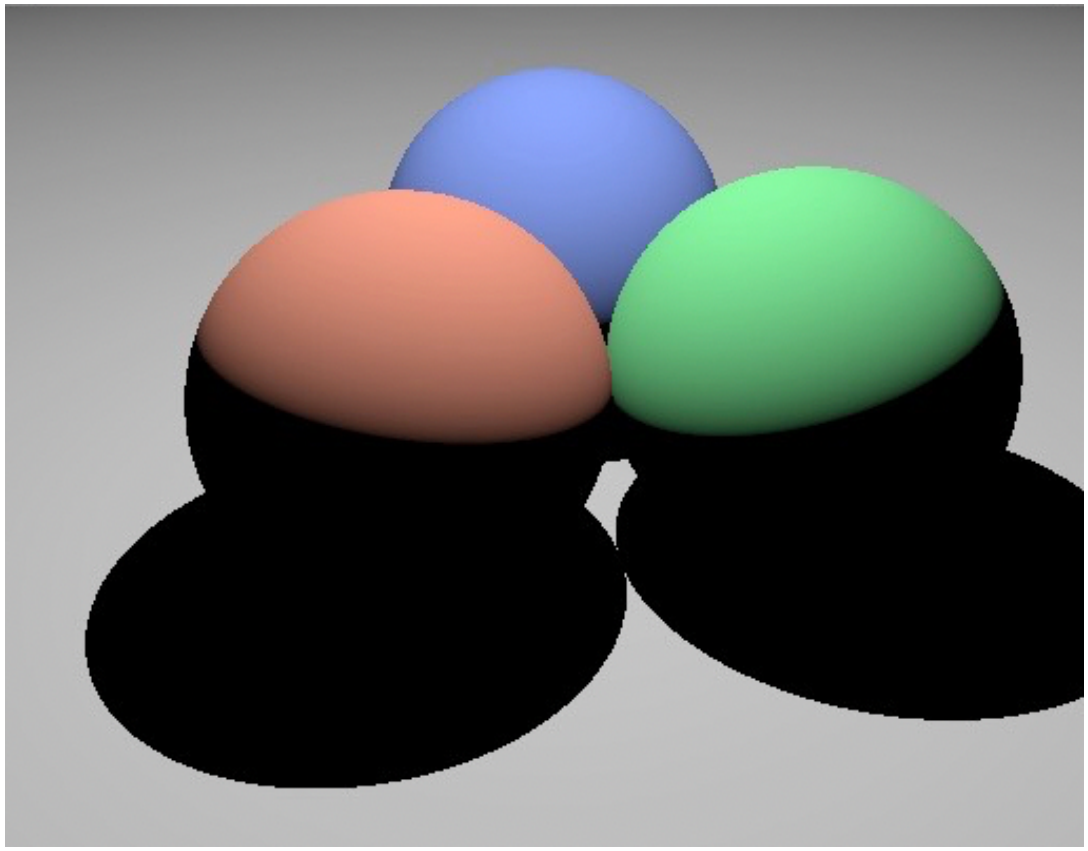
(hard) Shadows

- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible

- Be careful not to intersect with the object itself. Two solutions:

  - Only check for hits against all other surfaces

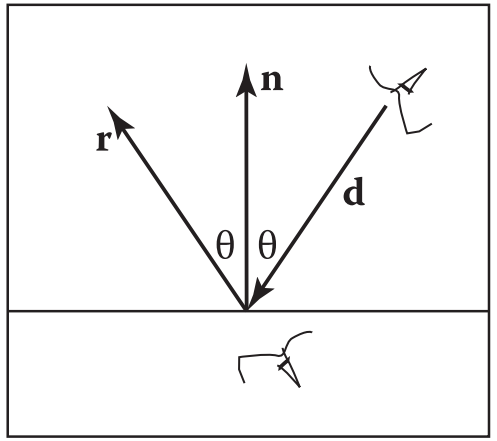
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}



Soft Shadows

Handling soft shadow will require **distributed ray shooting** - next class

Reflection



- Ideal **specular** reflection, or mirror reflection, can be modeled by casting another ray into the scene from the hit point
- Direction $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$
- One can then recursively accumulate some amount of color from whatever object this hits
- color += k_m `*ray_cast()`