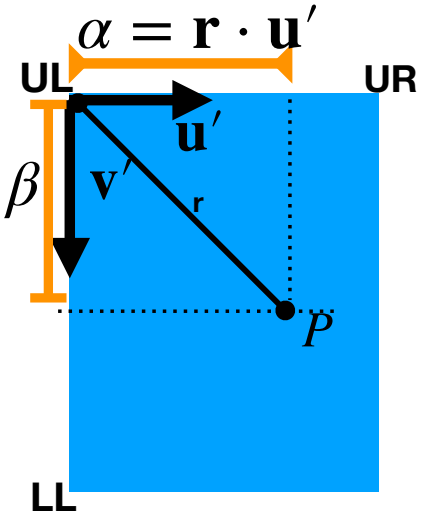
CSC 433/533
Computer Graphics

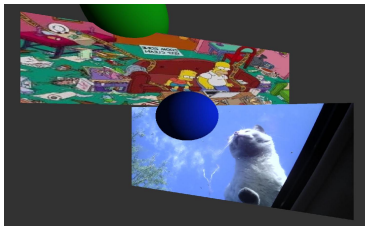
Alon Efrat
Credit: Joshua Levine

Finding the color of a point on a billboard

- For each billboard, you will be given 3 corners (UL,UR,LL)
- Let **u',v'** be orthonormal vectors (orthogonal and unit length). $\mathbf{u}' = \frac{UR - UL}{\|UR - UL\|}$
- Let P be a point on the plane containing the billboard. Let $\mathbf{r} = P - UL$.
- **Let** $\alpha = \mathbf{r} \cdot \mathbf{u}'$
- α is the length of the projection of **r** on **u'** .
- Other words. "shadow" that **r** casts on the line containing **u'**
- We can use α, β to determine if P is in the billboard, (how), and if yes, find the pixel of the image of the billboard at P.



$$\mathbf{P} = \mathbf{UL} + \underbrace{(\mathbf{r} \cdot \mathbf{u}')}_{\alpha}\,\mathbf{u}' + \underbrace{(\mathbf{r} \cdot \mathbf{v}')}_{\beta}\,\mathbf{v}'$$



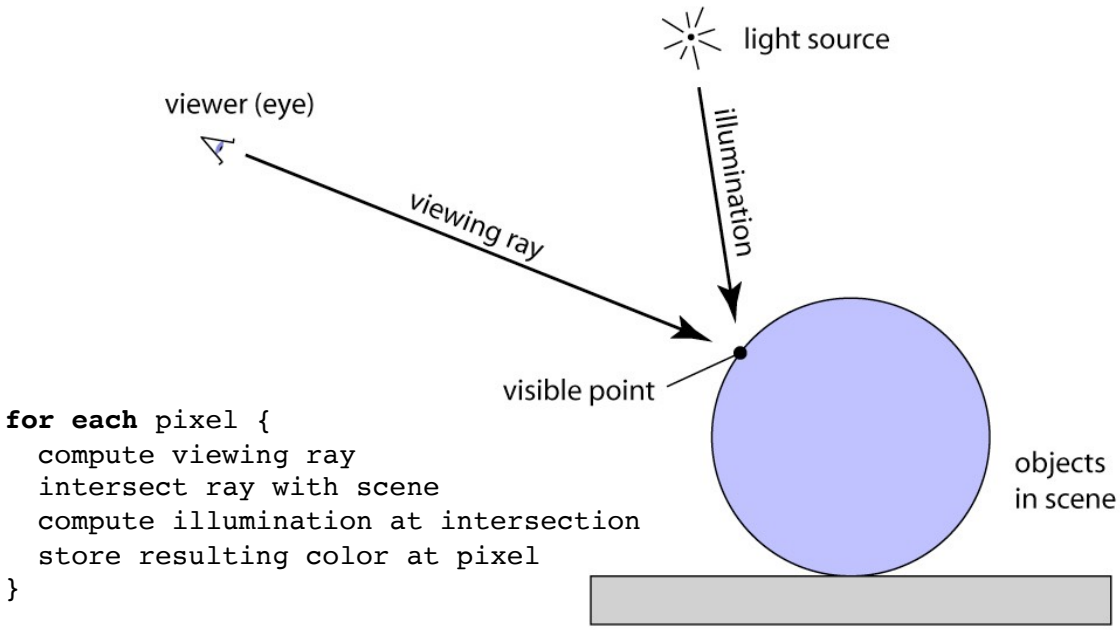
- Be aware that billboards are not necessarily vertical to the ground

Ray Tracing 2

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
}
```

Intersecting Objects

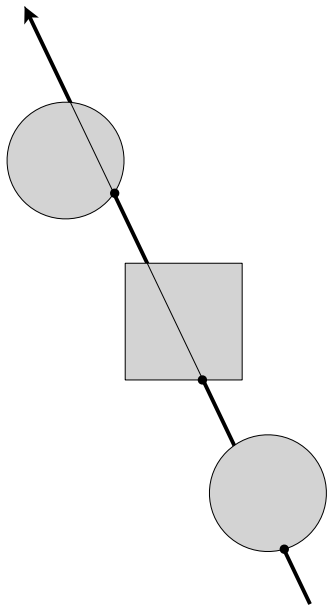
```
for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at intersection
  store resulting color at pixel
}
```

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



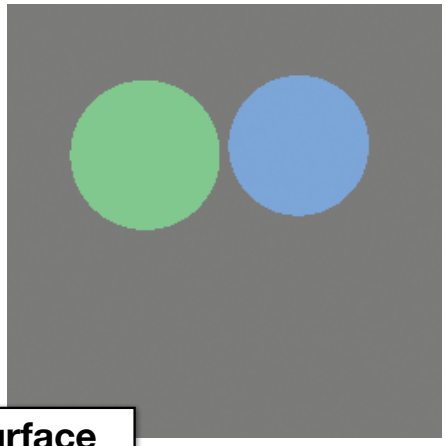
Illumination

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

Our images so far

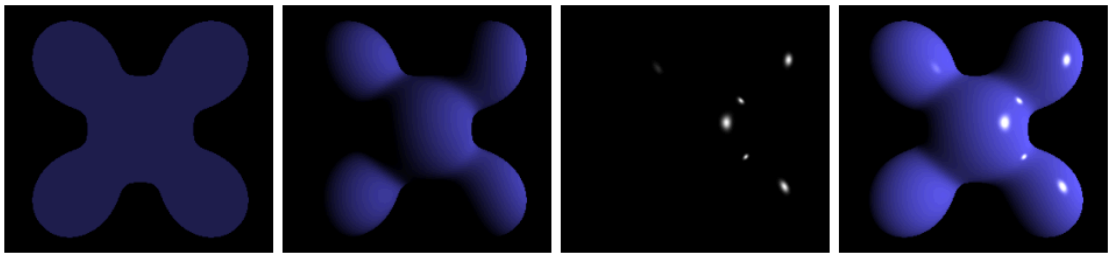
- With only eye-ray generation and scene intersection

```
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

Today: shading

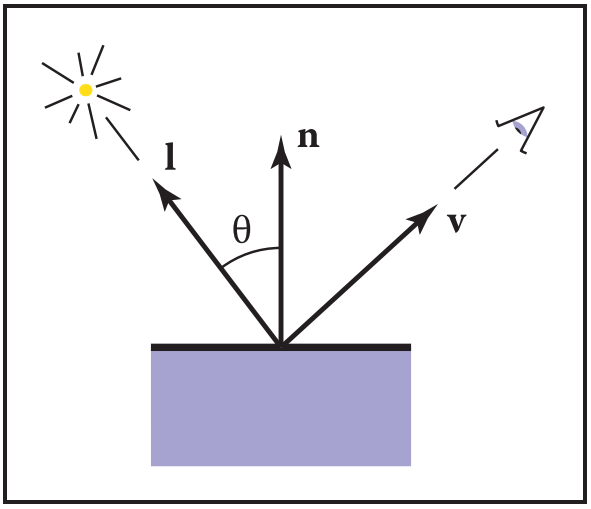


From this (ambient shading) + Diffuse Shading + Specular Shading ⇒ this

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

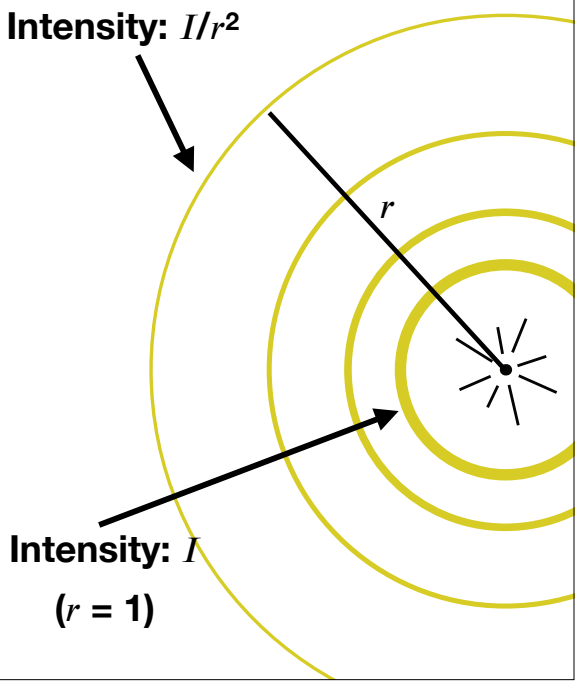
Shading

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



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 **we will ignore this for now.**



Shading Models

Just to be sure:

Shading ≠ Shadows

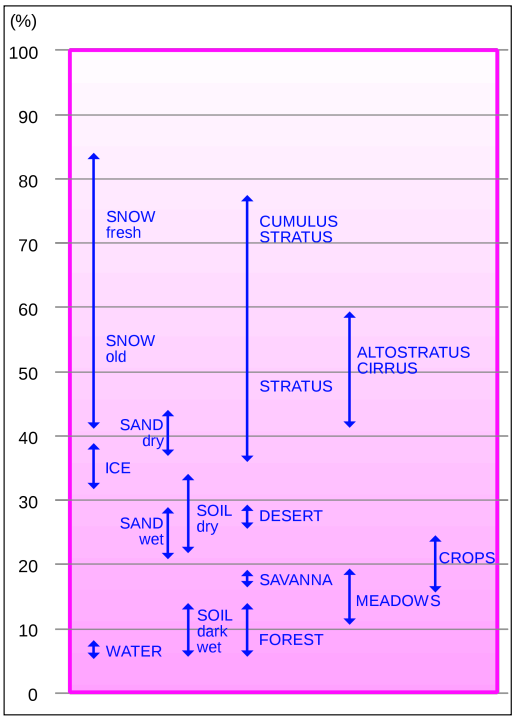
- **Shadows** are casted by occluding sources of light.
- **Shading** of a surface - changing of intensity of the **reflected** light due to surface properties ad geometry, and its locations in 3D with respect to locations of viewer and light source.

We will cover Diffuse shading and Specular Shading. We will study a trick that is easy to program, and ``looks" like physical diffuse shading.

Ambient coefficient ≠ Albedo coefficient

- Albedo coefficient - percentage of white light reflected by the object
- White light -might contains all visible frequencies, not only RGB.

- No attention to color.

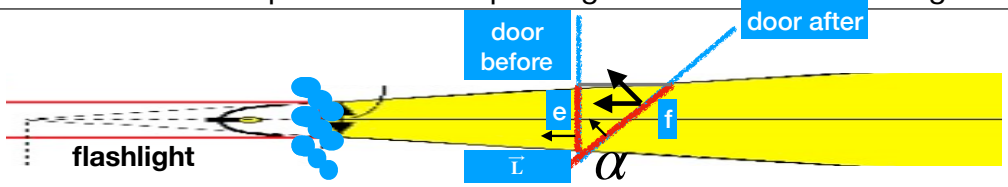


Ambient ``shading" and Albedo

- Ambient light - has no particular direction.
- Every material has 3 coefficients $(\mathbf{k_d}\,.\,\mathbf{r},\ \mathbf{k_d}\,.\,\mathbf{g},,\ \mathbf{k_d}\,.\,\mathbf{b})$.
- $\mathbf{k_d}\,.\,\mathbf{b}$ specifies the percentage of **blue** light that the surface reflects (obviously, as blue light).
- The location of viewer and the location of the light-source are irrelevant.
- If a sphere has Ambient coefficient $(k_d\,.\,r,\ k_d\,.\,g,\ k_d\,.\,b) = (0.1,\ 0.9,\ 0.9)$ it looks very dim in Red light, but bright in Blue or Green light.
- If illuminated by while light, then the sphere color is cyan.
- When describing a scene to (say) OpenGL, WebGL, processing.org etc, we could specify for every light source how much intensity it emits (in RGB).
- In reality, there is no ambient light.
- In OpenGL, we could specify 3 sets of coefficients (for ambient, for diffuse, and for specular. We can also specify the scene ambient RGB.
- E.g. specifying the ambient light in the scene as (0.3, 0.1, 0.9), and a sphere with k_d=(0, 0, 0.5), will be seen with $RGB = (0,\ 0,\ 0.45)$

Lambertian (Diffuse) Shading

- Consider a door illuminated by a flashlight (see below).
- Lets think about the intensity reflected from the door as the door rotates.
- I denotes the intensity. Think about I as #photons$/inch^2$
- Let e be a portion of the door with area 1_{in^2}. The number of photons falling on e is I.
- Now open the door (without moving e). Let f be the area of the shadow that e casts on the door. The area of f is $1_{in^2}/\cos\alpha$ (where α is the angle of the door)
- The same amount of photos that are passing via e are falling on a large area

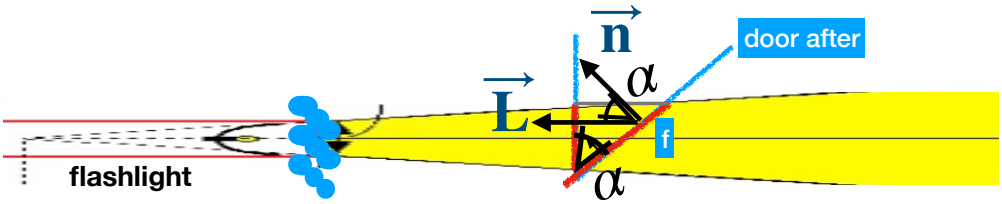


Intensity of the light on f = #photons falling on 1_{in^2}

The number of photons on e and on f is the same, but the area

increases to $1_{in^2}/\cos\alpha$, so intensity now is $I/f = I/\frac{1}{\cos\alpha} = I\cos\alpha$

Lambertian (Diffuse) Shading



Intensity of the light on f = #photons falling on 1_{in^2}

The number of photons on e and on f is the same, but the area

increases to $1_{in^2}/\cos\alpha$, so intensity now is $I/f = I/\frac{1}{\cos\alpha} = I\cos\alpha$

Let $\vec{\mathbf{L}}$ be a unit vector from f toward the light source, and let $\vec{\mathbf{n}}$ be the normal to the door.

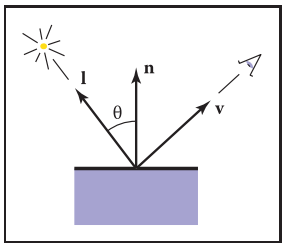
$\cos\alpha = \vec{\mathbf{L}} \cdot \vec{\mathbf{n}}$

The intensity of light reflected from f is intensity of light hitting f times k_d

Conclusion: To create diffuse shading, render f with RGB= $k_d\ I\ \vec{L} \cdot \vec{n}$

Lambertian (Diffuse) Shading

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



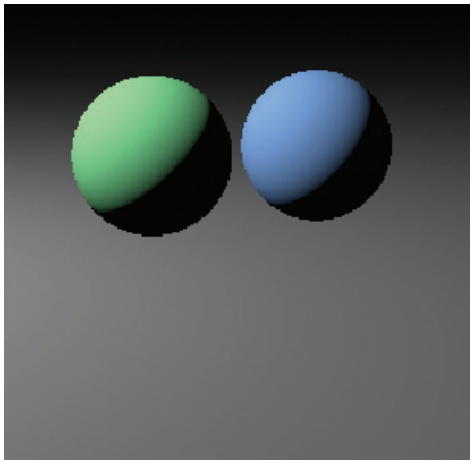
- Results in shading that is *view independent*

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

diffuse coefficient

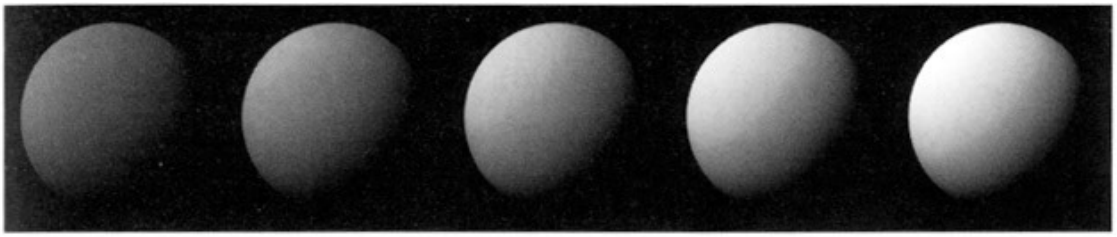
intensity/color of light

cos θ



Lambertian Shading

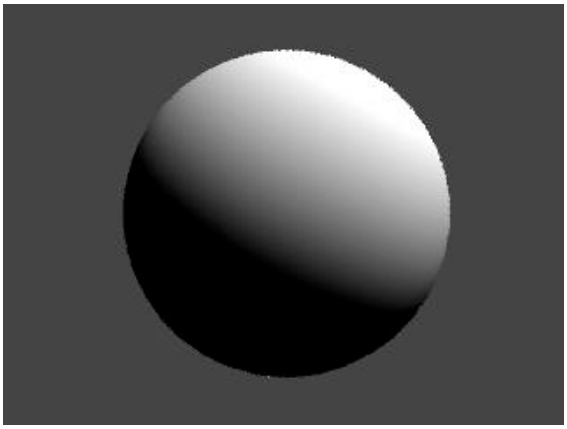
- k_d is a property of the surface itself (3 constants - one per each color channel)
- Produces matte appearance of varying intensities



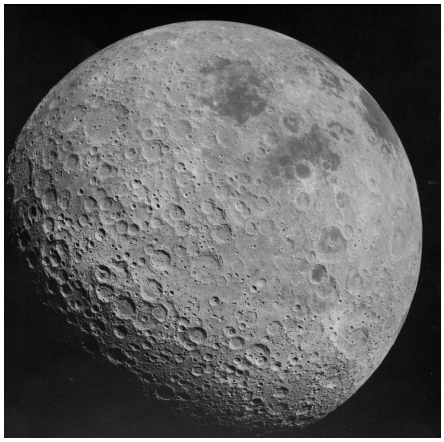
$k_d \longrightarrow$

The moon paradox

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



vs



How do we find a normal to a billboard ? Also a hint about for hw3

- Let h be the place containing a billboard.
- To know when a ray $r = o + t\,\overrightarrow{d}$ hits a plane containing a billboard, we need the billboard normal. How ?

$$\overrightarrow{U} = UR - UL$$
$$\overrightarrow{V} = LL - UL$$
$$\overrightarrow{n'} = (\overrightarrow{V}) \times (\overrightarrow{U})$$

(note that these are not unit vectors)

Ray-Sphere Intersection

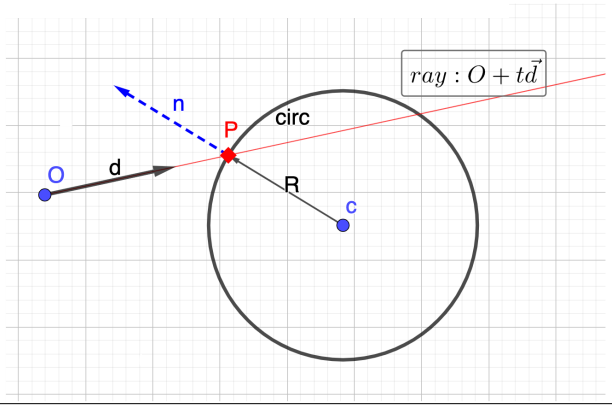
- Two conditions must be satisfied:
  - Must be on a ray: $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$
  - Must be on a sphere: $f(\mathbf{p}) = (\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$
- Can substitute the equations and solve for t in $f(\mathbf{p}(t))$:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

- Solving for t is a quadratic equation $At^2 + Bt + C = 0$,

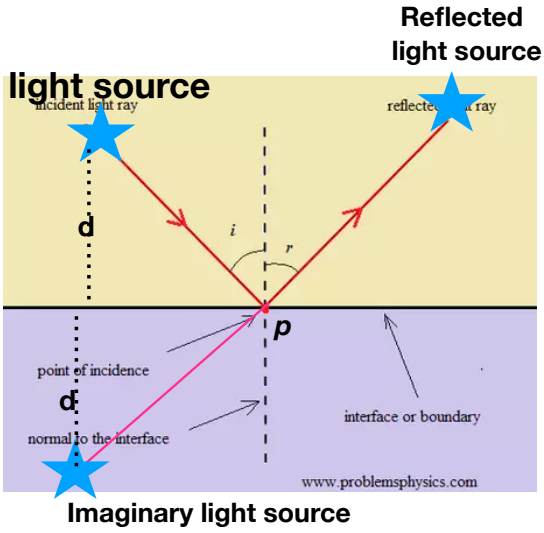
  where $A = (\mathbf{d} \cdot \mathbf{d})$; $B = 2\mathbf{d}(\mathbf{o} - \mathbf{c})$; $C = (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2$

$$\overrightarrow{n} = P - c$$

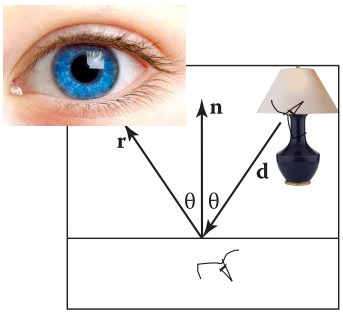


Toward Specular Shading: Perfect Mirror

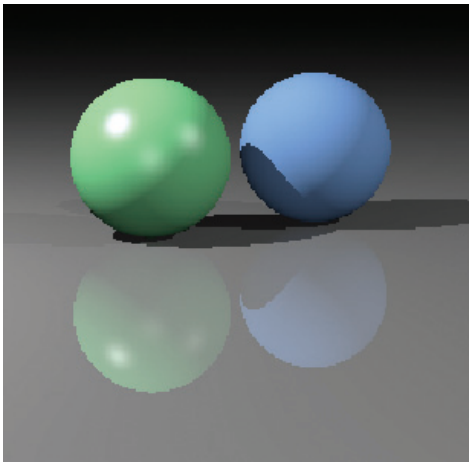
- Many real surfaces show some degree of shininess that produce **specular** reflections
- These effects move as the viewpoint changes (as oppose to diffuse and ambient shading)
- Idea: produce reflection when $\mathbf{v}$ and $\mathbf{l}$ are symmetrically positioned across the surface normal



Reflection

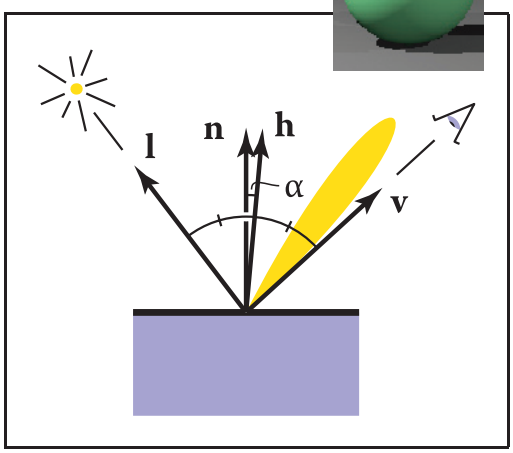


- Ideal **specular** reflection, or mirror reflection, can be modeled by casting another ray into the scene from the hit point
- Direction $\mathbf{r} = \mathbf{d} - 2(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$
- **r**- reflected ray toward the eye, **d -** ray from lamp. **n** is a unit vector orthogonal to the plane.
- Proof
  - (handwave) **r**=(r.x, r.y) and **d**=(d.x,d.y)
  - **r** and **d** have the same x-value, but opposite y-value: r.x=d.x and r.y= -r.y = r.y+(-2r.y)= r.y -2 (n · r)
  - $(\mathbf{d} \cdot \mathbf{n})\mathbf{n}$=(0, r.y).
- One can then recursively accumulate some amount of color from whatever object this hits
- color += k_m `*ray_cast()`

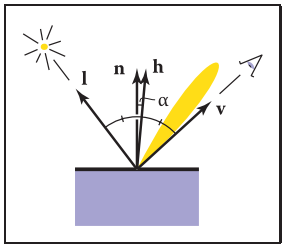


Blinn-Phong (Specular) Shading

- 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 **v** and **l** are symmetrically positioned across the surface normal



Blinn-Phong (Specular) Shading



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

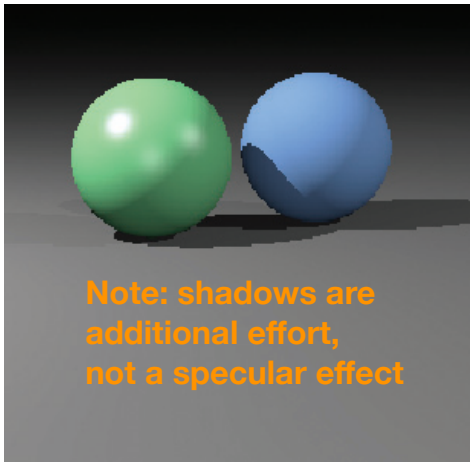
$$\mathbf{h} = (\mathbf{v} + \mathbf{l}) / \|\mathbf{v} + \mathbf{l}\|$$

- 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:

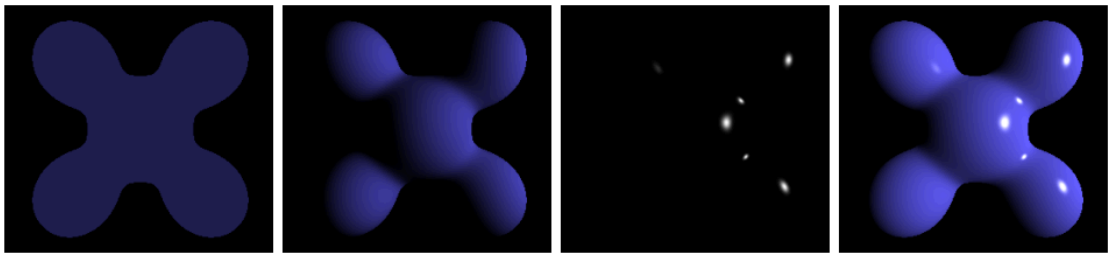
$$L_s = k_s I \max(0, \quad (\mathbf{n} \cdot \mathbf{h})^{p.} \ )$$

specular coefficient

Phong exponent



Blinn-Phong Decomposed

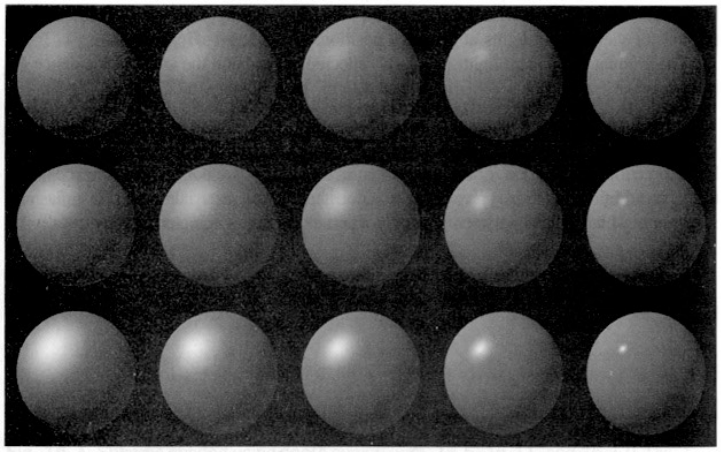


Ambient + Diffuse + Specular = Phong Reflection

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

$$L = L_a + L_d + L_s$$

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

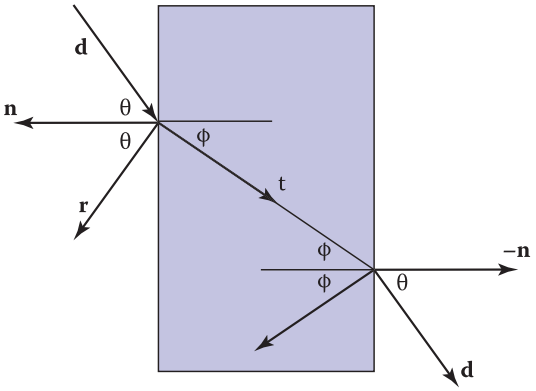
- And, the final result accumulates for all lights in the scene

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

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

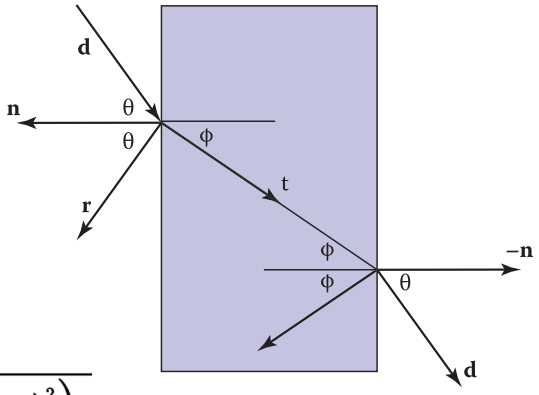
Refraction and Snell's Law

- Governs the angle at which a refracted ray bends
- Computation based on *refraction index (confusingly denoted n_t)* of the mediums. The mediums here are air and glass. They air has refraction index n=1, while the glass has refraction index n_t
- Snell law: $n_t \sin\theta = n \sin\phi$



Snell's Law

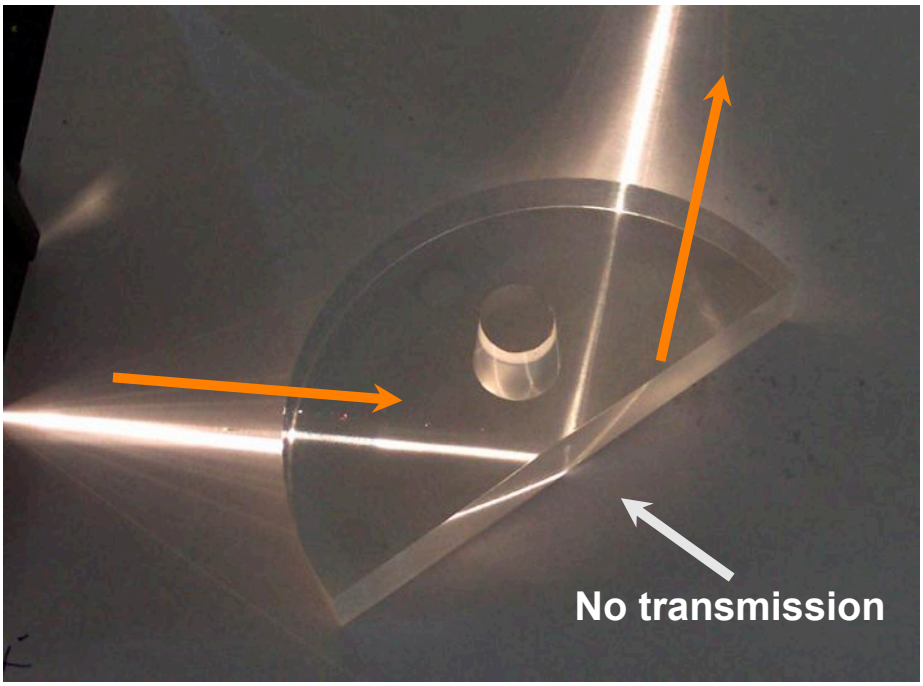
- Working with cosine's are easier because we can use dot products
- Can derive the vector for the refraction direction **t** as



$$\mathbf{t} = \frac{n(\mathbf{d}+\mathbf{n}\cos\theta)}{n_t} - \mathbf{n}\cos\phi$$

$$= \frac{n(\mathbf{d}-\mathbf{n}(\mathbf{d}\cdot\mathbf{n}))}{n_t} - \mathbf{n}\sqrt{1 - \frac{n^2\left(1-(\mathbf{d}\cdot\mathbf{n})^2\right)}{n_t^2}}$$

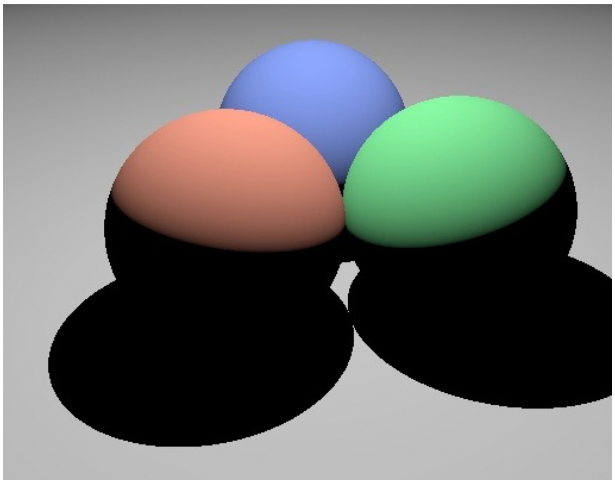
Total Internal Reflection



Recursive Ray Tracing

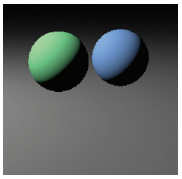
Shadows

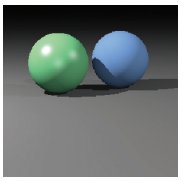
- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}

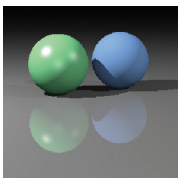


Recursive Ray Tracer

```
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;
}
```

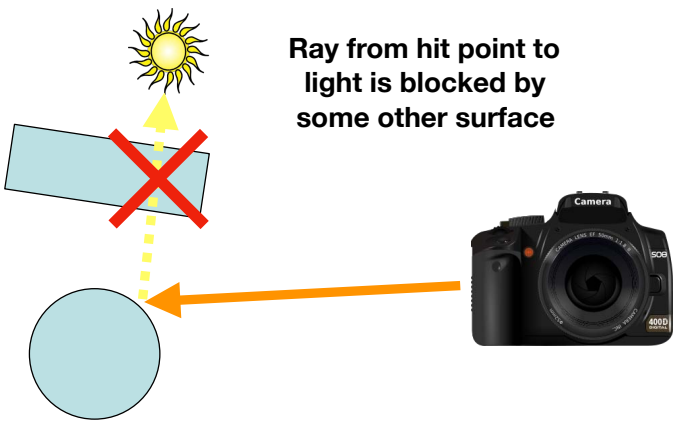






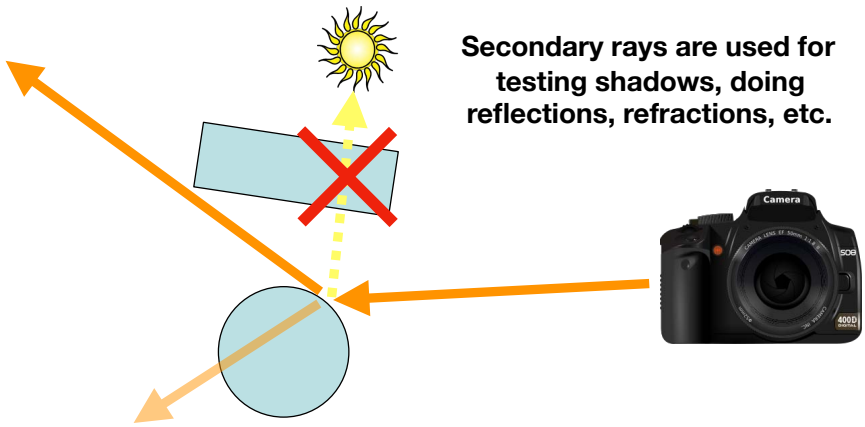
Shadows

- 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!



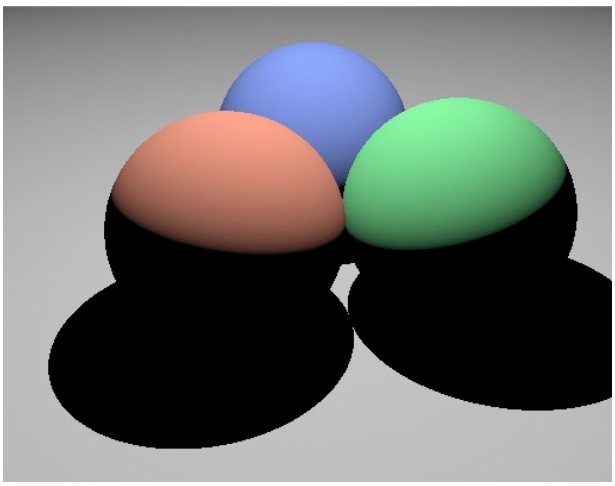
Ray Casting vs Ray Tracing

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



(hard) Shadows

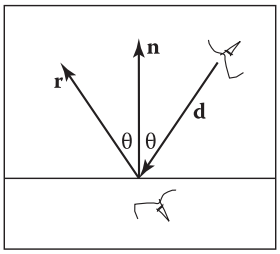
- Idea: after finding the closest hit, cast a ray to each light source to determine if it is visible
- Be careful not to intersect with the object itself. Two solutions:
  - Only check for hits against all other surfaces
  - Start shadow rays a tiny distance away from the hit point by adjusting t_{min}



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()`