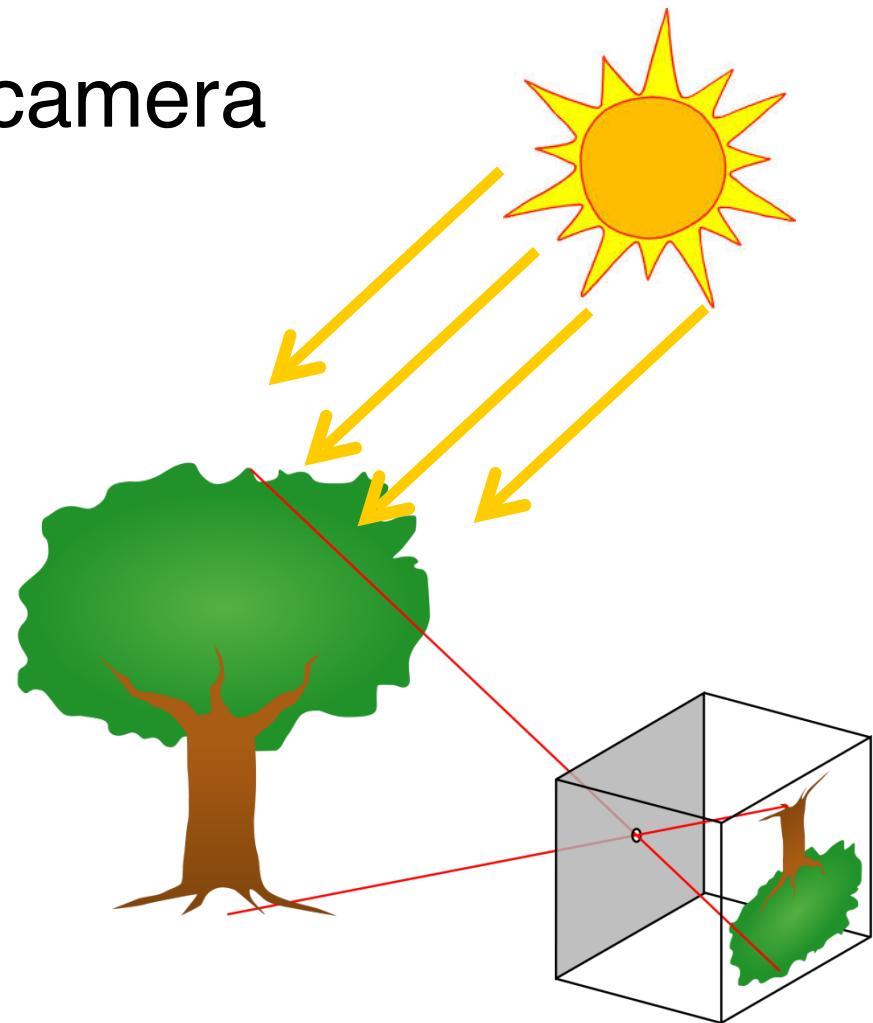


# **Basic Ray Tracing**

# Rendering: Reality

Eye acts as pinhole camera

Photons from light  
hit objects



# Rendering: Reality

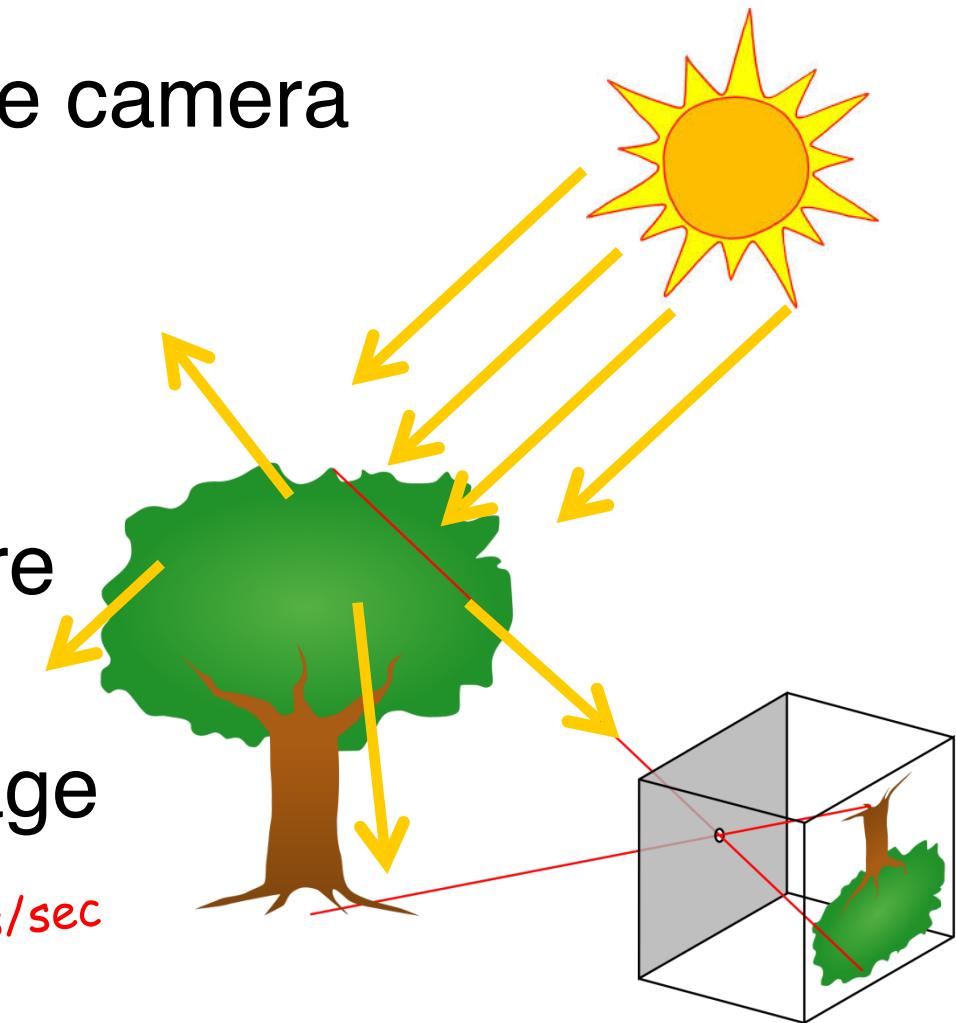
Eye acts as pinhole camera

Photons from light  
hit objects

Bounce everywhere

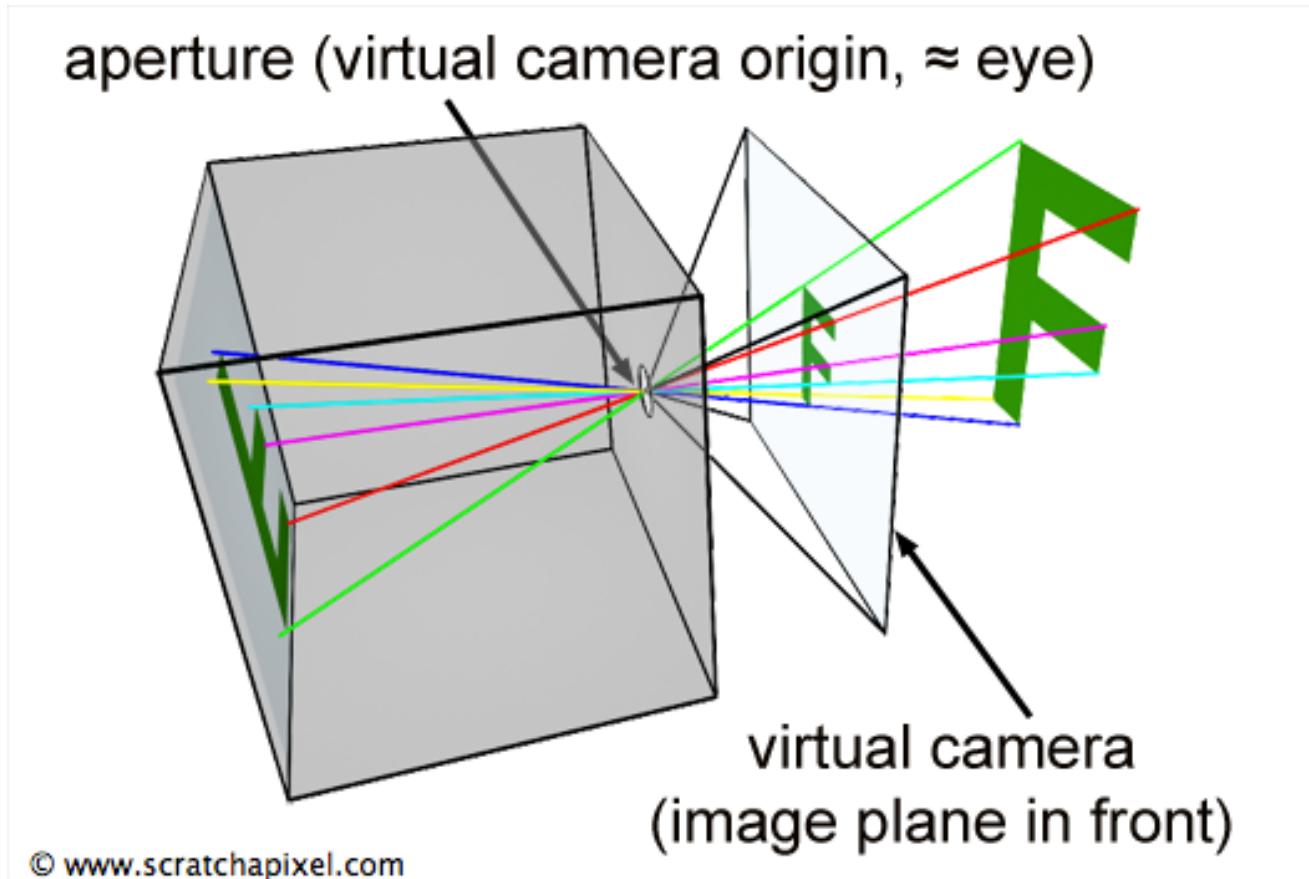
**Extremely** few  
hit eye, form image

one lightbulb =  $10^{19}$  photons/sec



# Synthetic Pinhole Camera

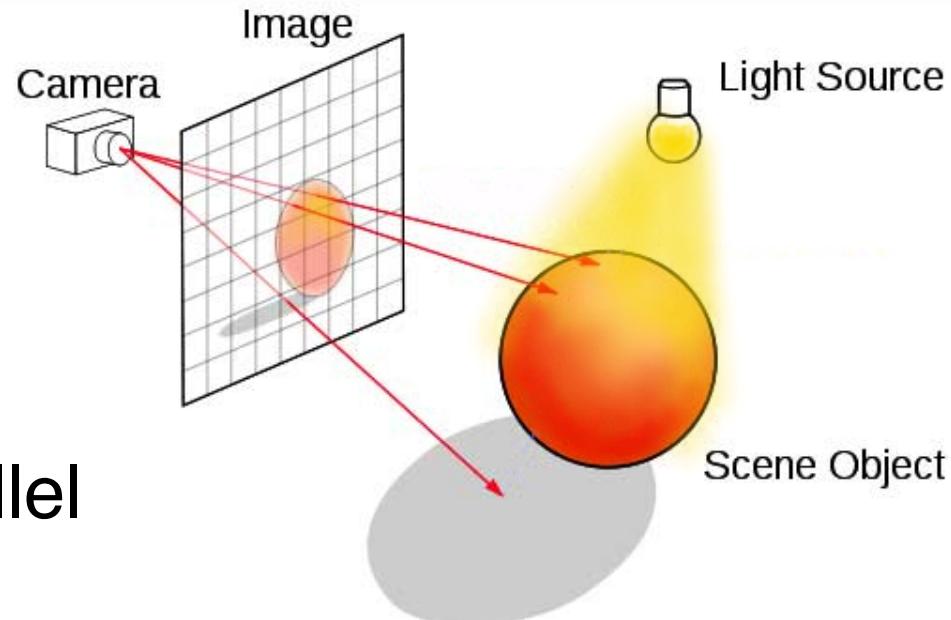
Useful abstraction: virtual image plane



# Rendering: Ray Tracing

Reverse of reality

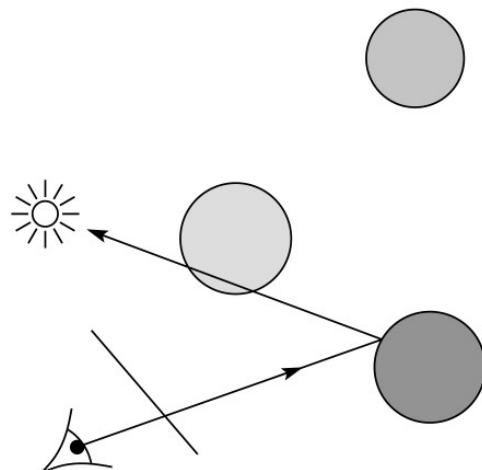
- Shoot rays through image plane
- See what they hit
- **Secondary** rays for:
  - Reflections
  - Shadows
- Embarrassingly parallel



# Local Illumination

Simplifying assumptions:

- Ignore everything except eye, light, and object
  - No shadows, reflections, etc



# “Ray Tracing is Slow”

Very true in the past;  
still true today

Ray tracing already  
used within the  
“raster” pipeline

Real-time, fully ray-  
traced scenes are  
coming



[Nvidia OptiX]

# Side Note

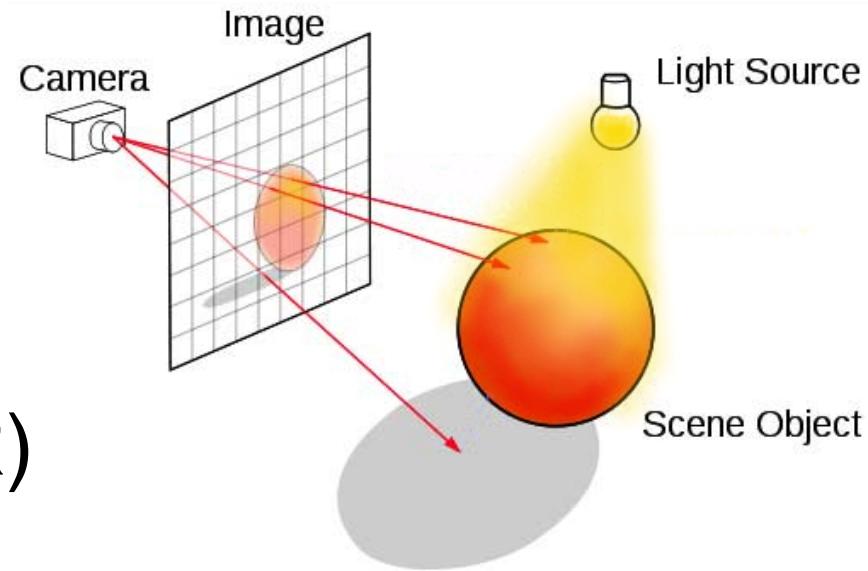
What does NVidia's RTX hardware do?



# Why Slow?

Naïve algorithm:  $O(NR)$

- R: number of rays
- N: number of objects

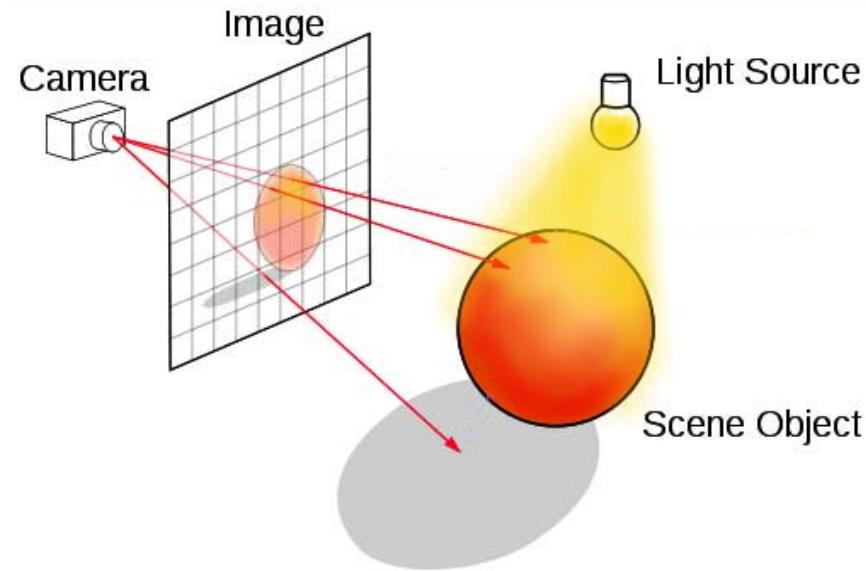


But rays can be cast in parallel

- each ray  $O(N)$
- even faster with good culling

# Why Slow?

Despite being parallel:



1. Poor cache coherence
  - Nearby rays can hit different geometry
2. Unpredictable
  - Must **shade** pixels whose rays hit object
  - May require tracing rays **recursively**

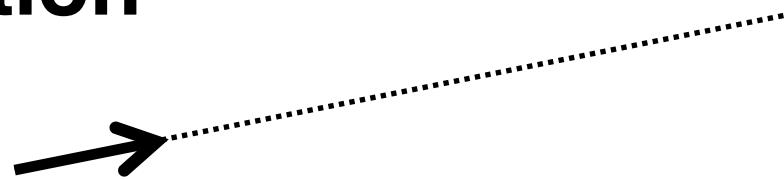
# Basic Algorithm

For each pixel:

- Shoot ray from camera through pixel
- Find first object it hits
- If it hits something
  - Shade that pixel
  - Shoot secondary rays

# Shoot Rays From Camera

Ray has **origin** and **direction**



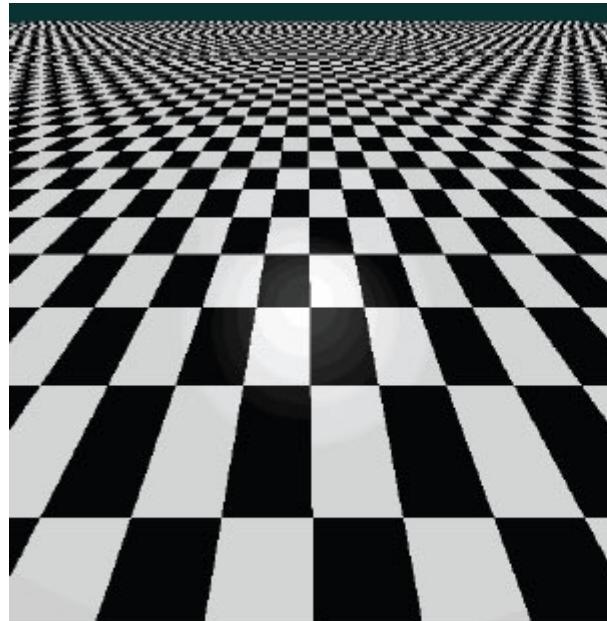
Points on ray are the **positive span**

How to create a ray?

# Shoot Rays From Camera

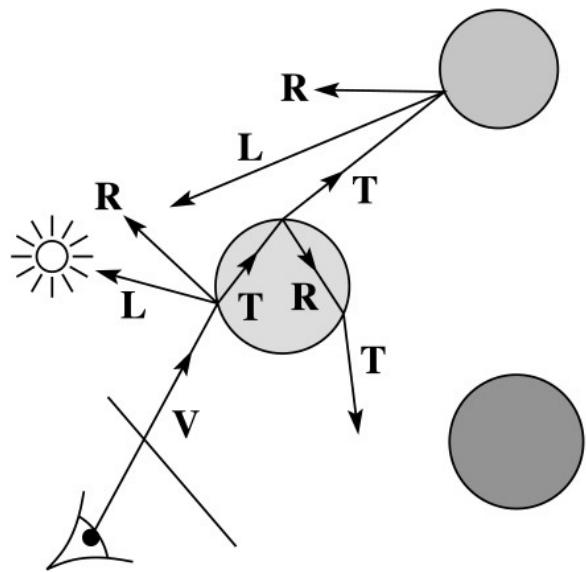
Creating a ray:

- Origin is eye
- Pick direction to pierce **center** of pixel



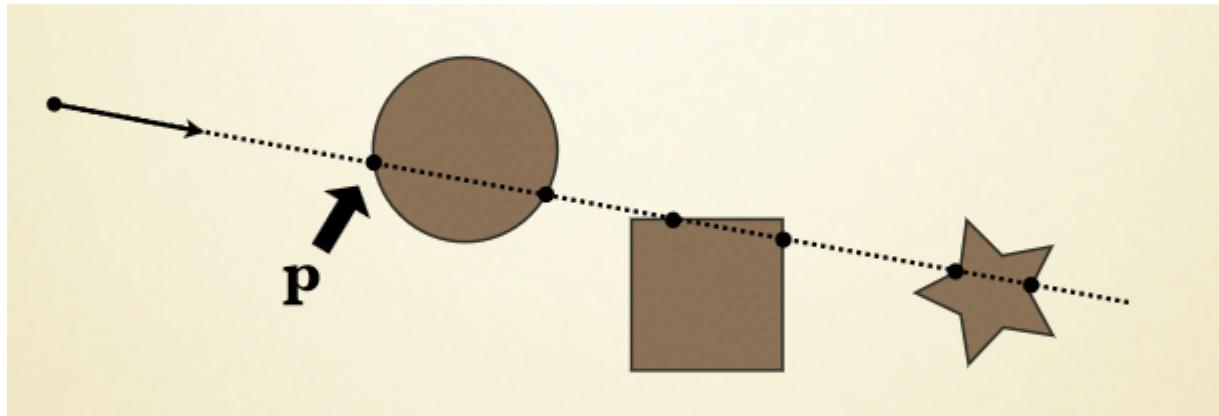
# Whitted-style Ray Tracing

- Turner Whitted introduced ray tracing to graphics in 1980
  - Combines eye ray tracing + rays to light and recursive tracing
- Algorithm:
  1. For each pixel, trace primary ray in direction **V** to the first visible surface.
  2. For each intersection trace secondary rays:
    - Shadow in direction **L** to light sources
    - Reflected in direction **R**
    - Refracted (transmitted) in direction **T**
  3. Calculate shading of pixel based on light attenuation



# Find First Object Hit By Ray

**Collision detection:** find all values of  $t$  where ray hits object boundary



Take smallest **positive** value of  $t$

# **When Did We Hit an Object?**

How do we know?

How can we calculate this efficiently?

# Efficient Approximations

Multiple approximate checks eliminate candidates more efficiently than a single, accurate check

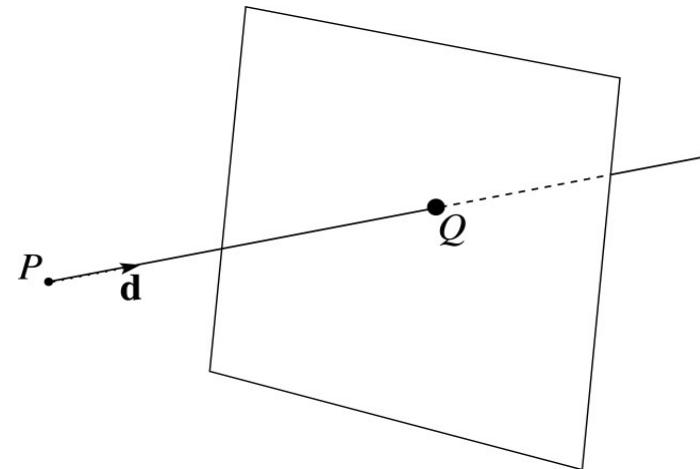
Checks (in order):

- Ray-Plane intersection
- Ray-Triangle intersection
- Position of intersection on triangle

# Ray-Plane Collision Detection

Plane specified by:

- Point on plane
- Plane normal



In-class Activity:

Use the plane equation to determine where point Q is based on the ray origin  $P$  and direction  $\vec{d}$  assuming we already know at least one other point on this plane

$$N\cdot Q+d=0$$

$$N\cdot(\mathsf{P}+\overrightarrow{\mathsf{d}}t)+d=0$$

$$N\cdot \mathsf{P}+N\cdot \overrightarrow{\mathsf{d}}t=-d$$

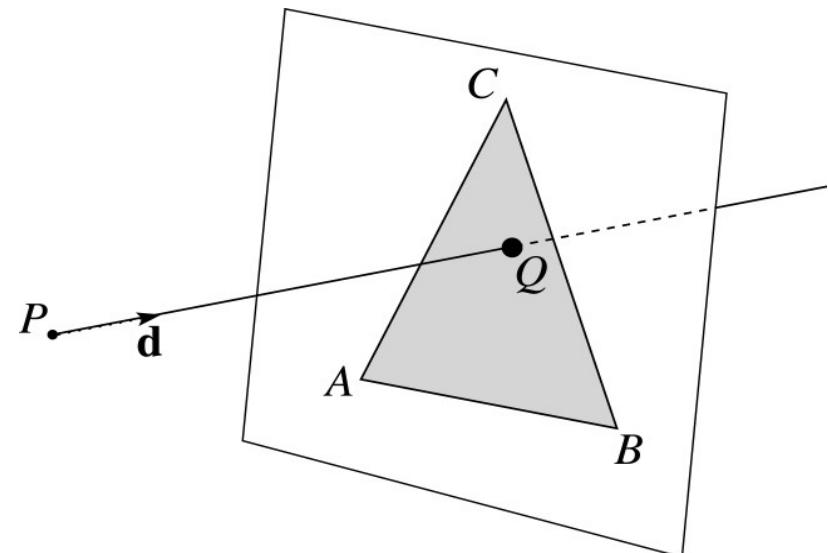
$$N\cdot \overrightarrow{\mathsf{d}}t = - (d + N\cdot \mathsf{P})$$

$$t=-\frac{N\cdot \mathsf{P}+d}{N\cdot \overrightarrow{\mathsf{d}}}$$

$$Q=\mathsf{P}+\overrightarrow{\mathsf{d}}t$$

# Ray-Triangle Collision Detection

- Intersect ray with triangle's supporting plane:  
$$N = (A - C) \times (B - C)$$
- Check if inside triangle



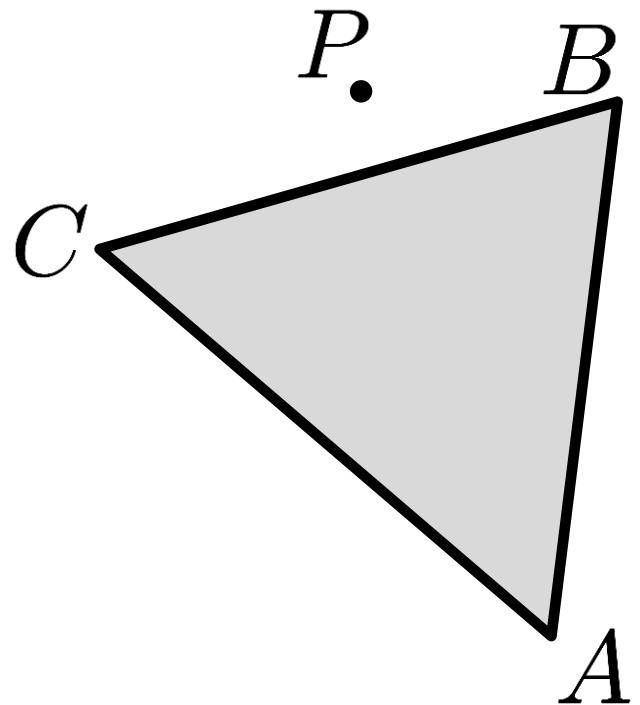
# How to Check if Inside?

- Using triangle edges
- Using barycentric coordinates
- Using projections

# Ray-Triangle Collision Detection

Normal:

$$\hat{n} = \frac{(B-A) \times (C-A)}{\|(B-A) \times (C-A)\|}$$



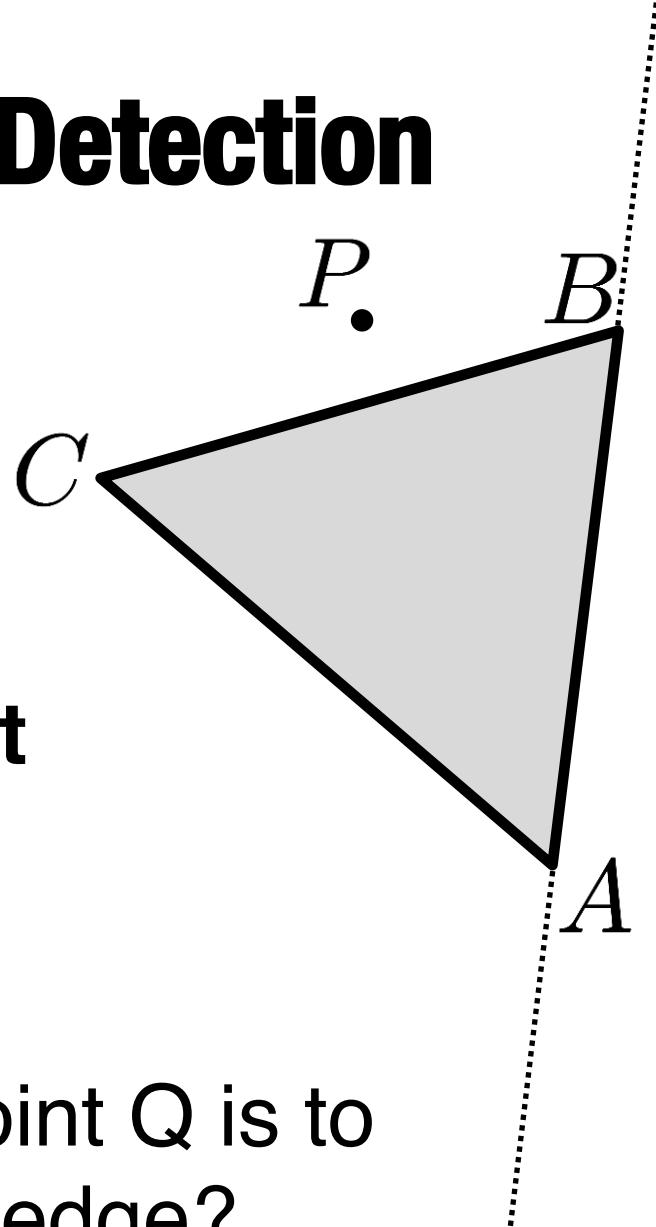
# Ray-Triangle Collision Detection

Normal:

$$\hat{n} = \frac{(B-A) \times (C-A)}{\|(B-A) \times (C-A)\|}$$

Idea: if P inside, must be **left** of line AB

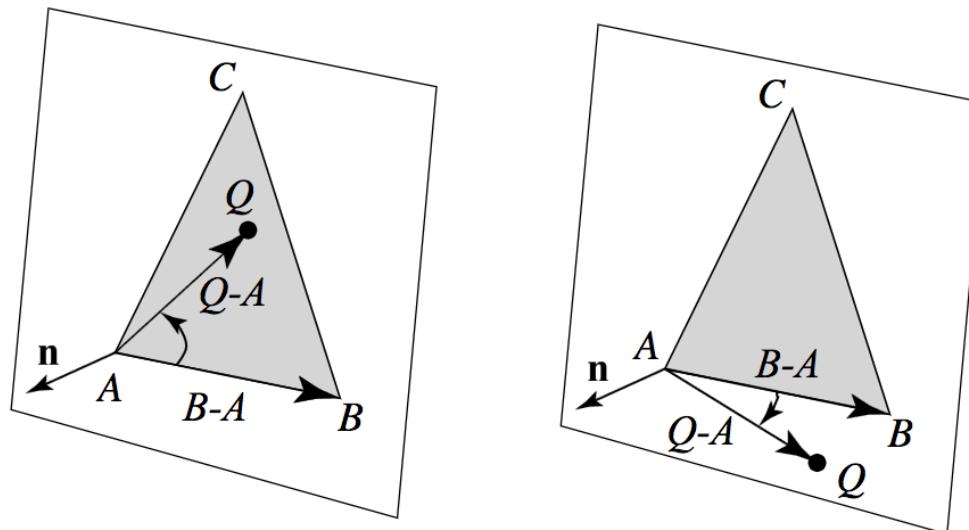
How can we determine if point Q is to the left or right of a triangle edge?



# Intuition

Cross product will point in opposite direction if point Q is to the right

Therefore dot product will now be negative  
( $\cos\Theta < 0$  if  $\Theta > 90^\circ$ )



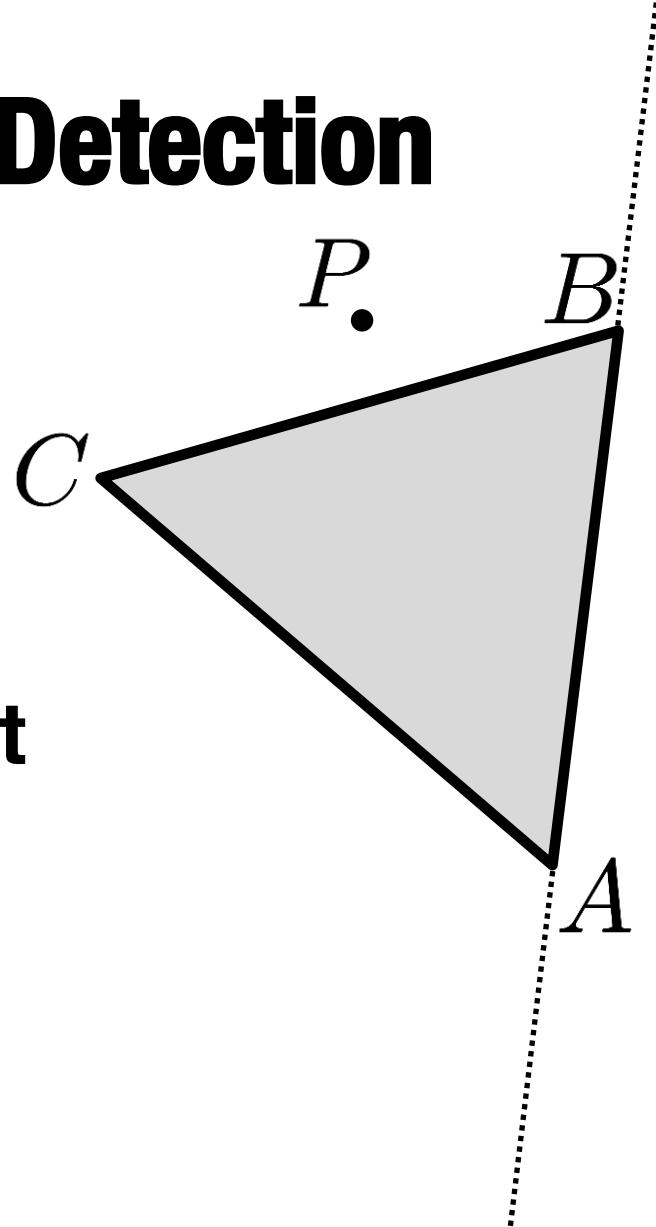
# Ray-Triangle Collision Detection

Normal:

$$\hat{n} = \frac{(B-A) \times (C-A)}{\|(B-A) \times (C-A)\|}$$

Idea: if P inside, must be **left** of line AB

$$(B - A) \times (P - A) \cdot \hat{n} \geq 0$$



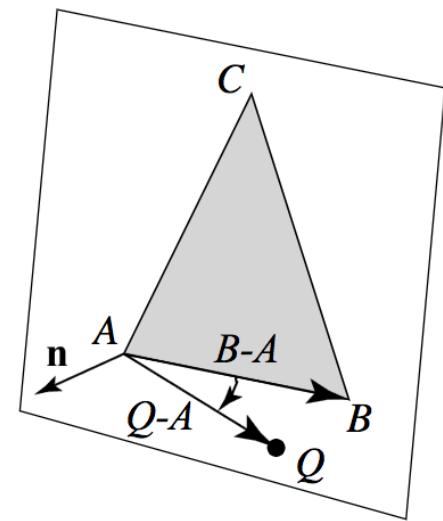
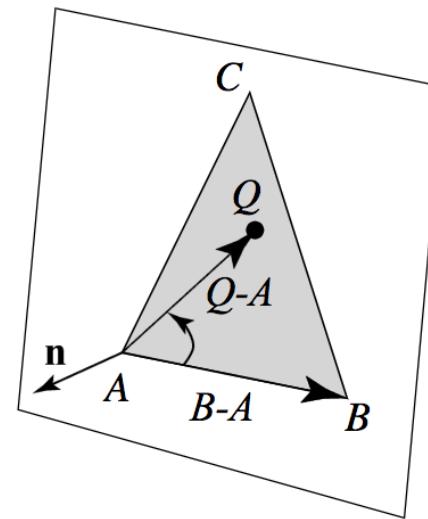
# Inside-Outside Test

Check that point Q is to the left of all edges:

$$[(B-A) \times (Q-A)] \cdot n \geq 0$$

$$[(C-B) \times (Q-B)] \cdot n \geq 0$$

$$[(A-C) \times (Q-C)] \cdot n \geq 0$$



If it passes all three tests, it is inside the triangle

# Caveat

The inside-outside method may have some corner cases that result in unshaded fragments on the first assignment

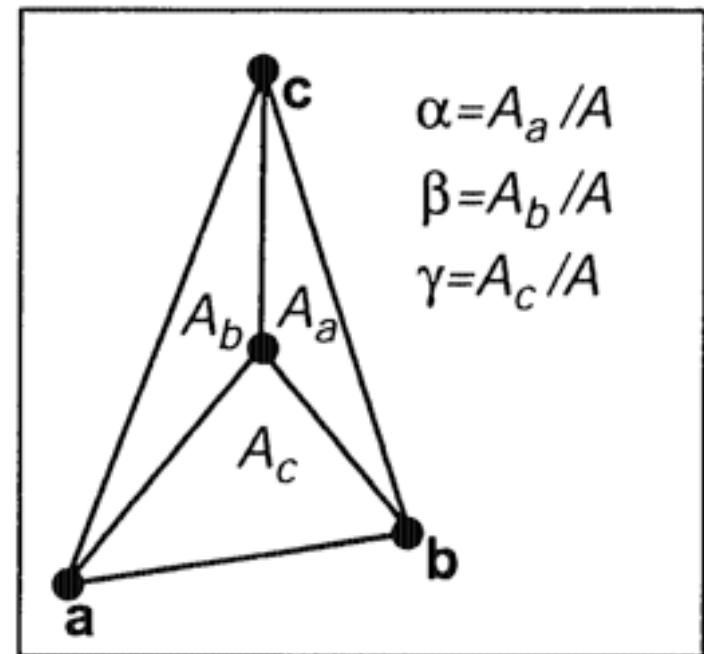
We will not mark off points for that if you've implemented the inside-outside test correctly but better methods exist...

# Barycentric Coordinates

Affine frame defined by origin  
( $t = c$ ) and vectors from  $c$  ( $v = a-c$ ,  $w = b-c$ )

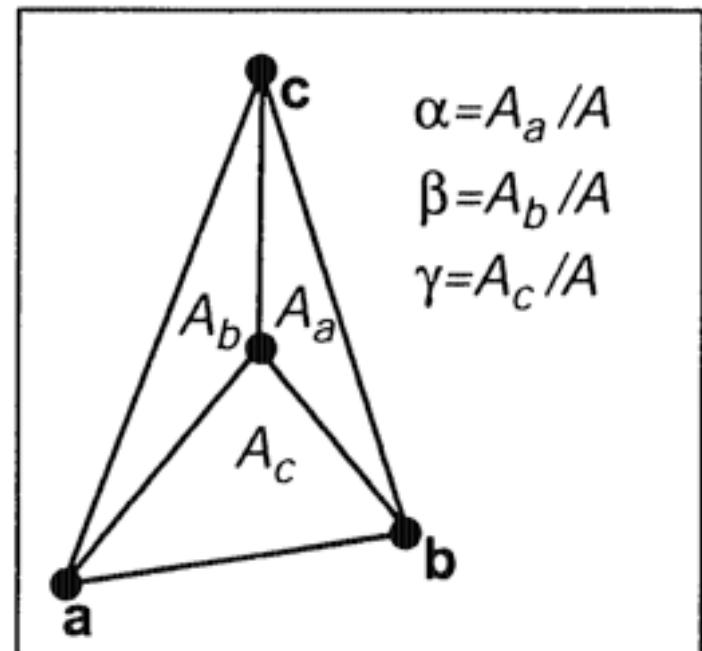
Point can be represented  
using area coordinates  $\alpha$ ,  $\beta$ ,  
 $\gamma$  (ratio between sub-area  
and total triangle area):

$$Q = \alpha a + \beta b + \gamma c$$



# Barycentric Coordinates

What does these area coordinates tell us?



# Barycentric Coordinates

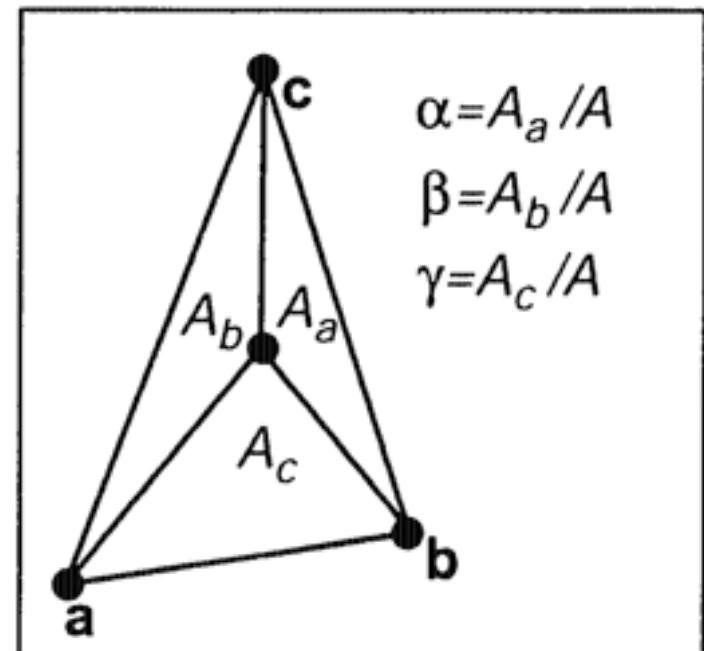
If point Q's

$$\alpha, \beta, \gamma \geq 0$$

and

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

then Q is within the triangle!



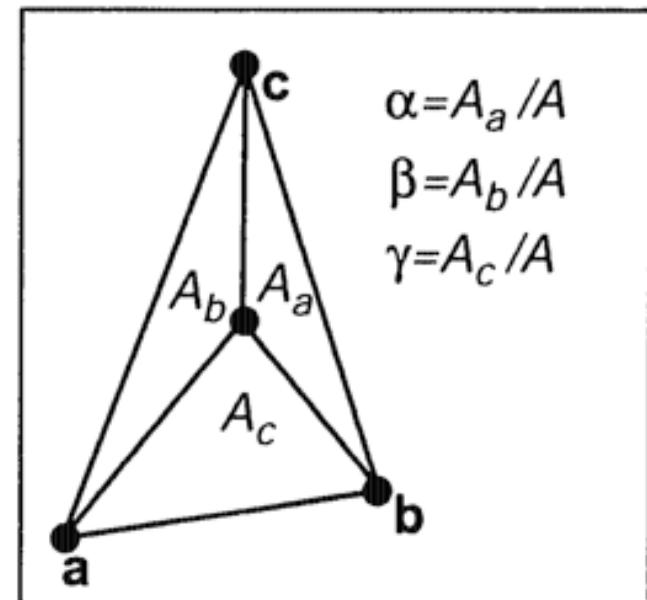
# Barycentric Coordinates

Proportional to lengths of crossproducts:

$$A_a = \|((C-B) \times (Q-B))\|/2$$

$$A_b = \|((A-C) \times (Q-C))\|/2$$

$$A_c = \|((B-A) \times (Q-A))\|/2$$



# Beyond Triangle Intersections...

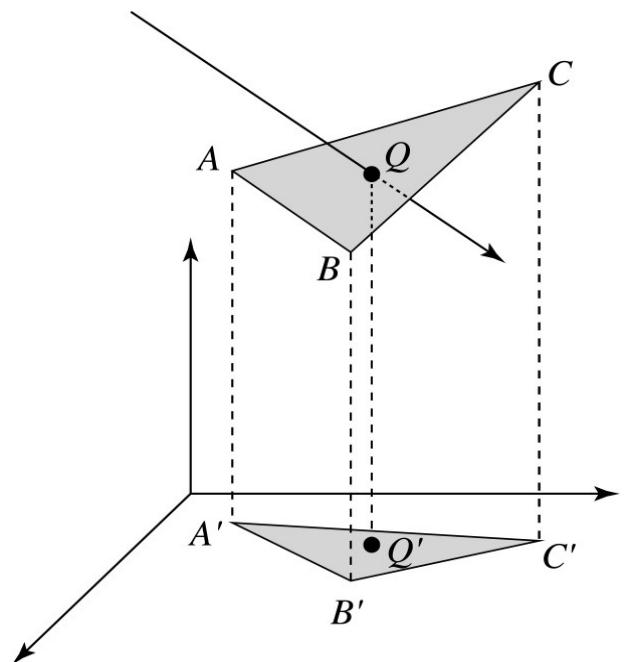
- Barycentric coordinates can interpolate
  - Vertex properties
  - Material properties
  - Texture coordinates
  - Normals

$$k_d(Q) = \alpha k_d(A) + \beta k_d(B) + \gamma k_d(C)$$

- Used everywhere!

# Barycentric Coordinates in 2D

Project down into 2D and compute barycentric coordinates



# Möller-Trumbore Triangle Intersect

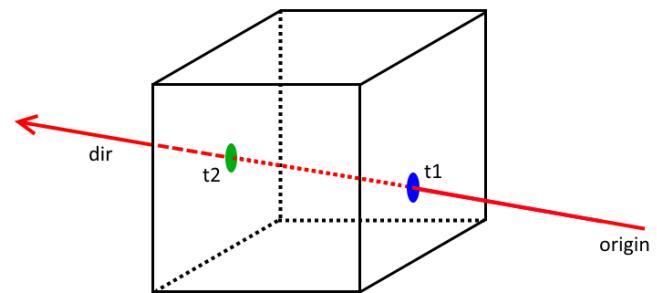
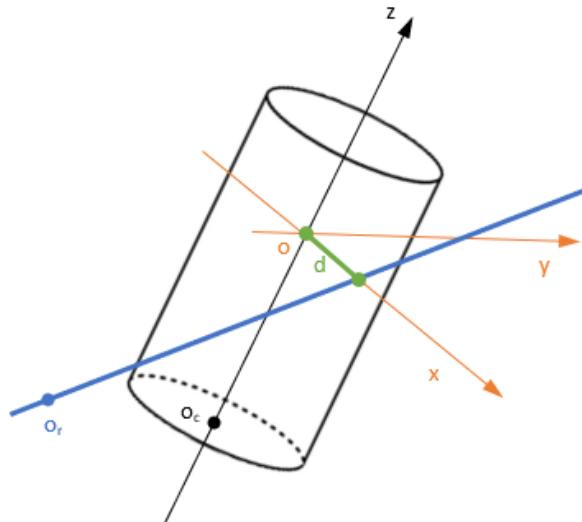
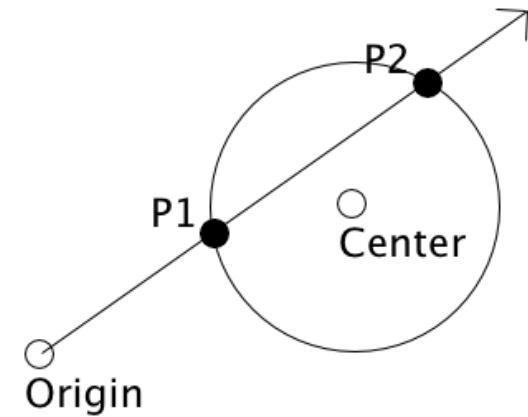
- Introduced as an optimized triangle-ray intersection test
- Based on the barycentric parameterization
  - Direction of ray intersection from ray origin becomes 3rd axis (uw are barycentric axes)
- Still commonly used

Full details here:

<https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/moller-trumbore-ray-triangle-intersection>

# Other Common Intersects

- Sphere
- Box
- Cylinder



# Ray Tracing: Shading

- Shading colors the pixels
- Color depends on:
  - Object material
  - Incoming lights
  - Angle of viewer

# Object Materials

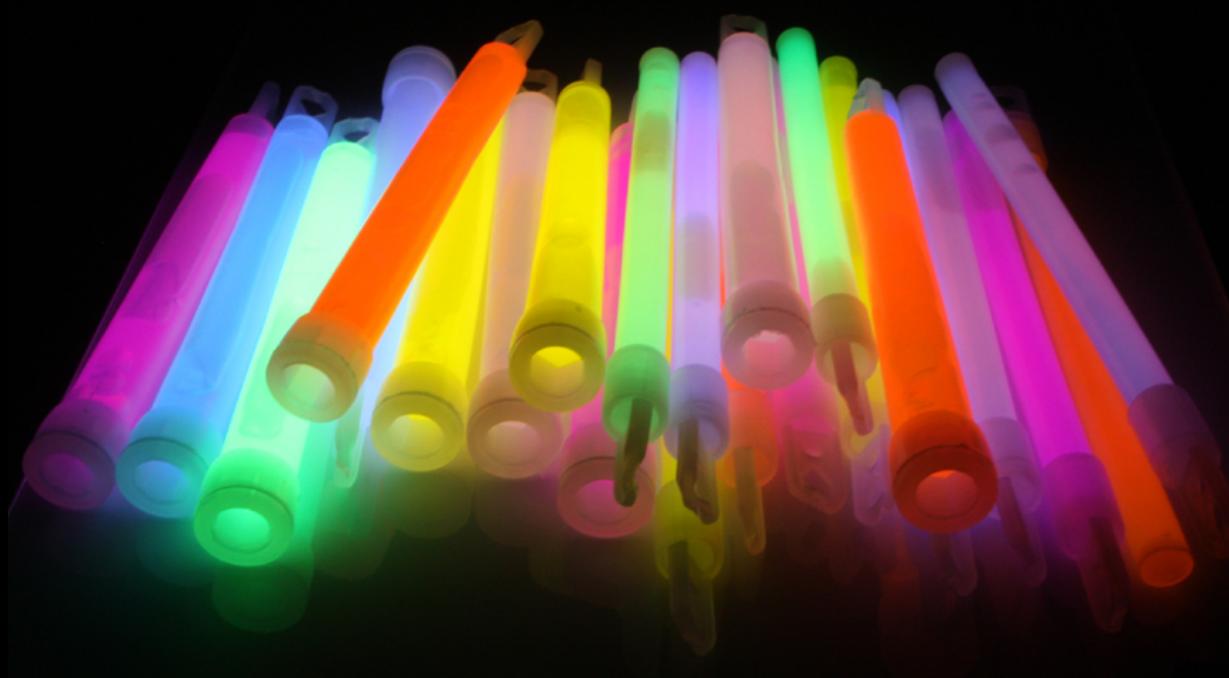
Different materials can behave **very** differently

- opaque vs translucent vs transparent
- shiny vs dull

We classify different responses to light into “types”

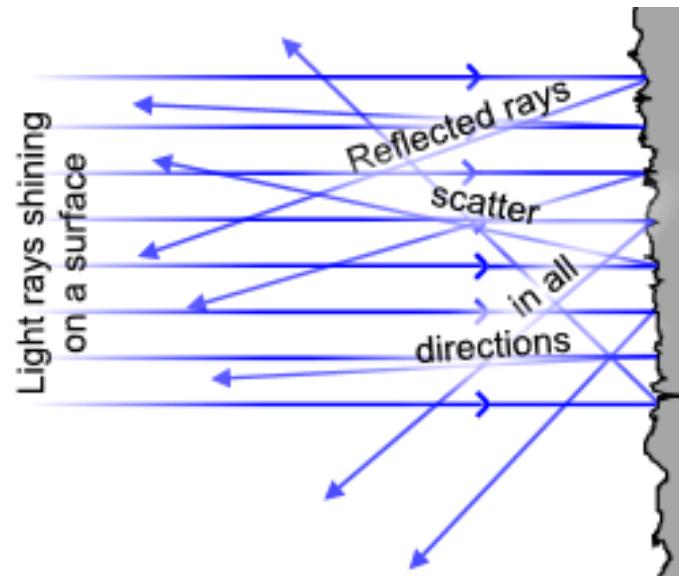
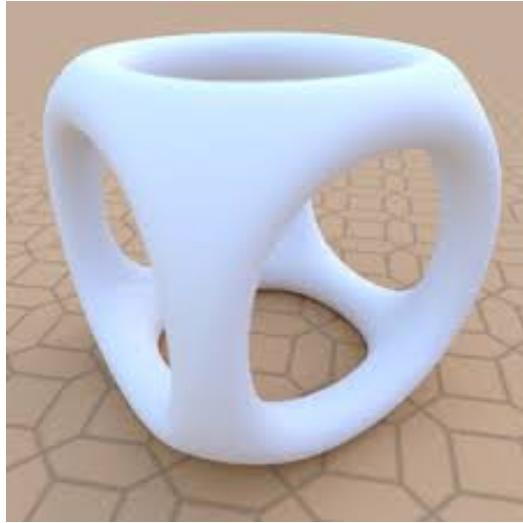
# Emissive Lighting

Light generated within material



# Diffuse Reflection

Light comes in, bounces out randomly

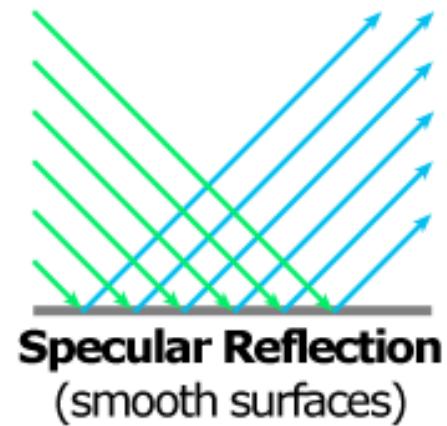
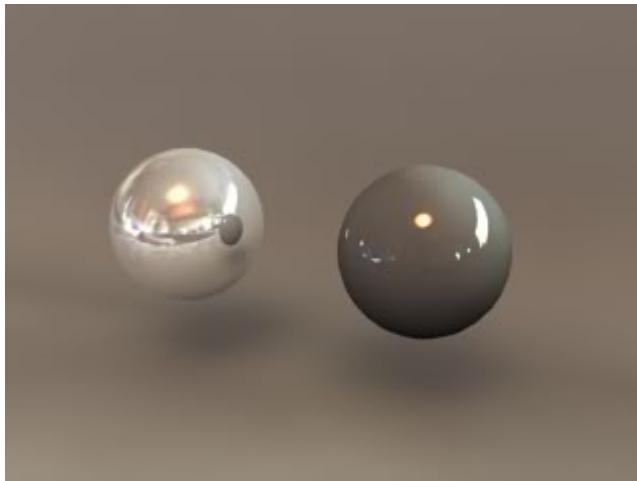


Typical for “rough” unpolished materials

View angle doesn't matter

# Specular Reflection

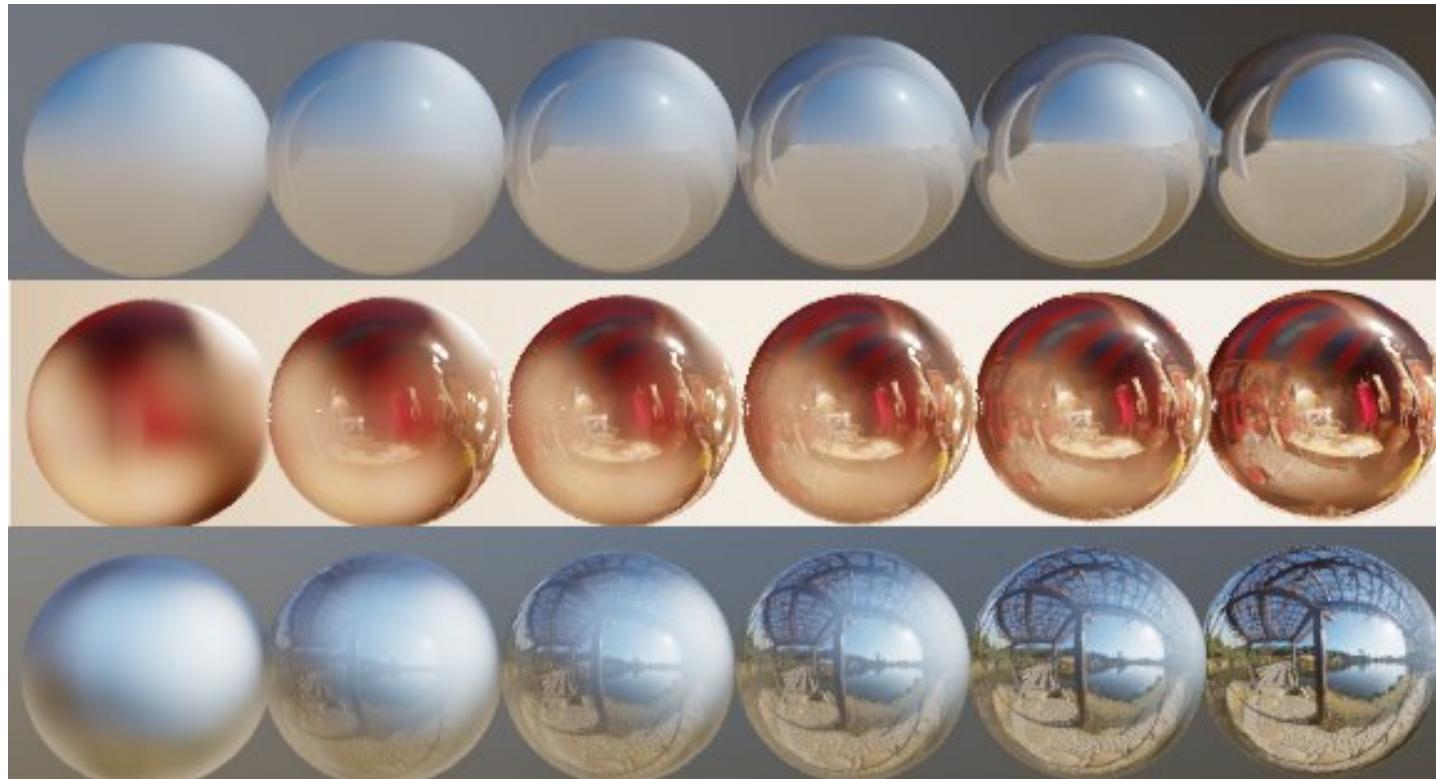
Light reflects perfectly



Typical for smooth, “polished” surfaces

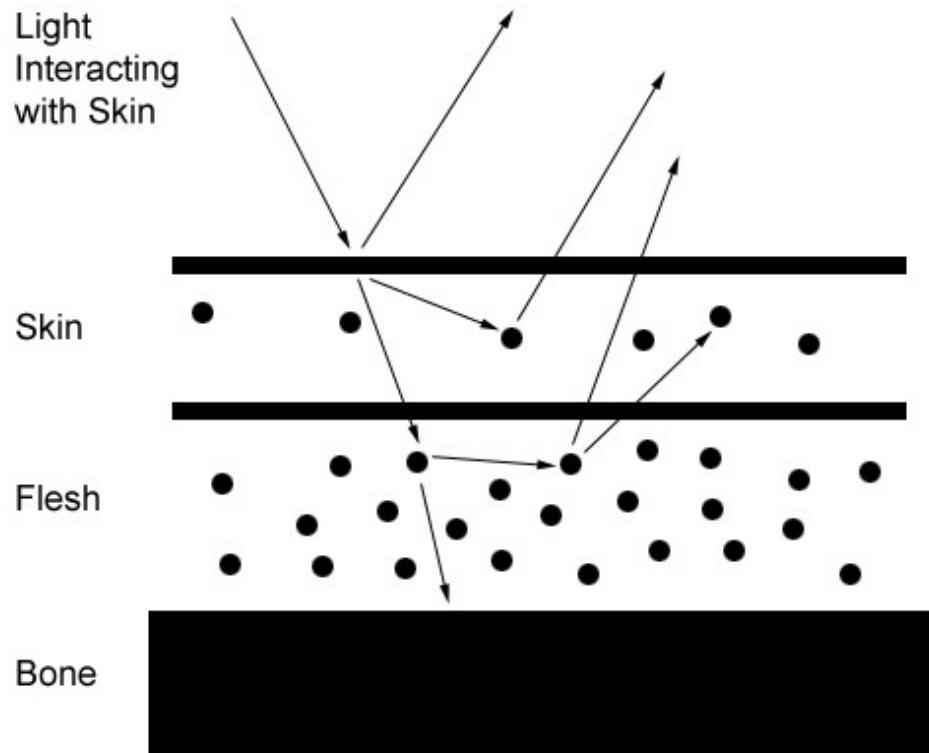
# General Opaque Materials

Diffuse-specular spectrum:



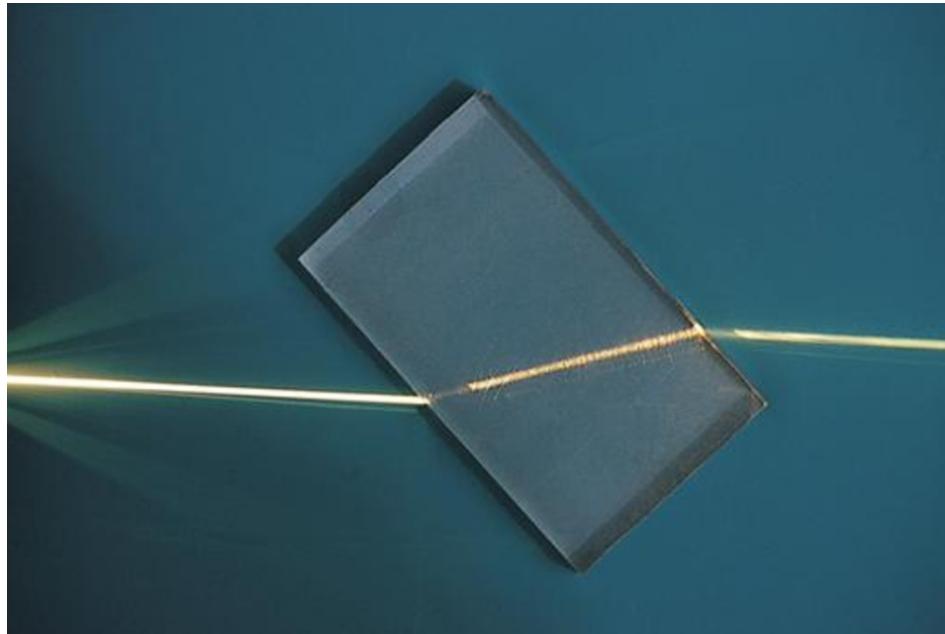
# What About Translucent?

## Subsurface Scattering



# What About Translucent?

Subsurface Scattering  
Refraction



# What About Translucent?

Subsurface Scattering

Refraction

Structural Color

....

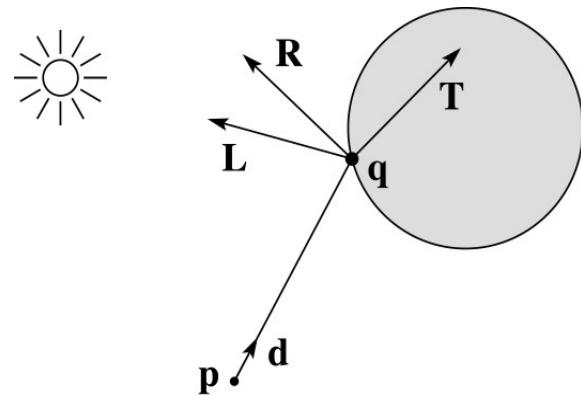
Not today.



# Phong Shading Model

We'll talk about the specific math behind shading models later. For now, let's focus on the “ray-tracing” aspect of shading...

# Ray Tracing: Shading



Let  $I(P, d)$  be the intensity along ray  $P + td$

$$I(P, d) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}$$

- $I_{\text{direct}}$  computed from Phong model
- $I_{\text{reflected}} = k_r I(Q, R)$
- $I_{\text{transmitted}} = k_t I(Q, T)$

# Reflection and Transmission

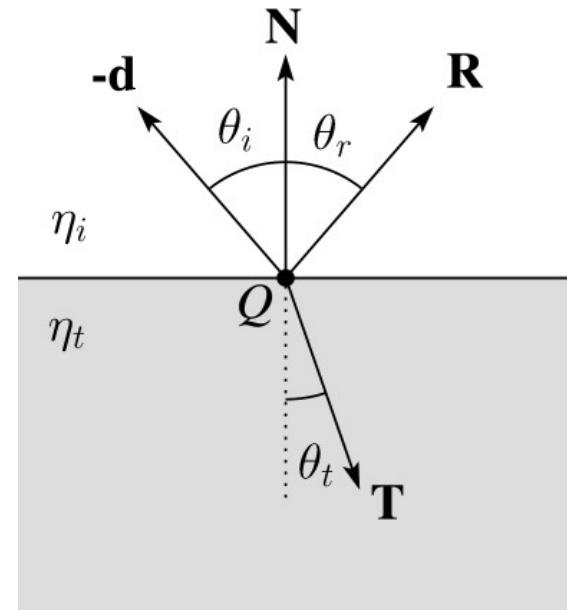
Law of reflection:

$$\theta_i = \theta_r$$

Snell's law of refraction:

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t$$

( $\eta$  is index of refraction)



# What is this effect?

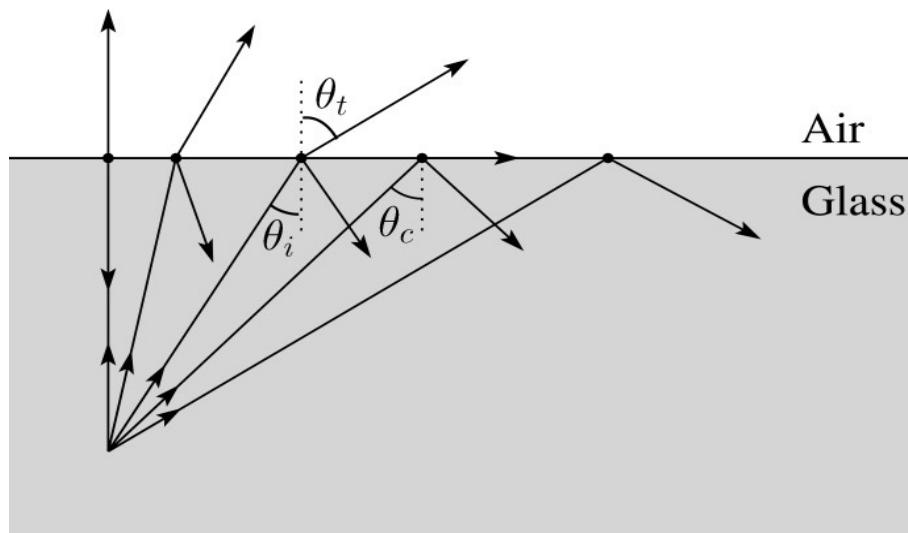


# Total Internal Reflection

- Occurs if:
  - $\eta_i > \eta_t$  (density of current medium > density of other medium)
  - $\Theta_i > \Theta_c$  (angle of incidence > critical angle)
- Critical angle is an angle of incidence that provides an angle of refraction of  $90^\circ$
- No transmission occurs – only reflection

# Critical Angle in TIR

- If  $\theta_t = 90^\circ$ , light moves along boundary surface
- If  $\theta_t > 90^\circ$ , light is reflected within current medium



# Light and Shadow Attenuation

Light attenuation:

- Light farther from the source contributes less intensity

Shadow attenuation:

- If light source is blocked from point on an object, object is in shadow
- Attenuation is 0 if completely blocked
- Some attenuation for translucent objects

# Light Attenuation

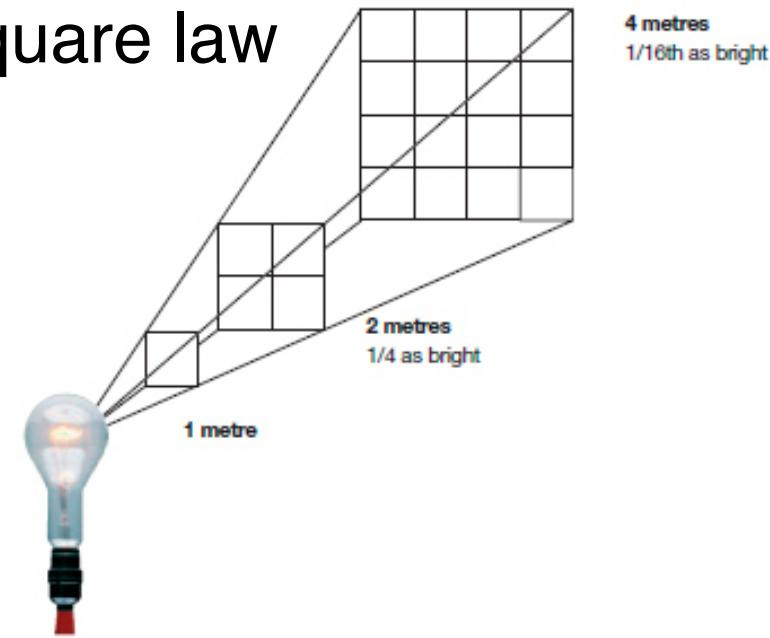
Real light attenuation: inverse square law

Tends to look bad: too dim  
or washed out

So, we cheat:

$d$  is light-to-point distance

Tweak constant & linear terms to taste:



$$f_{atten}(d) = \frac{1}{a + bd + cd^2}$$

# Shooting Shadow Rays

