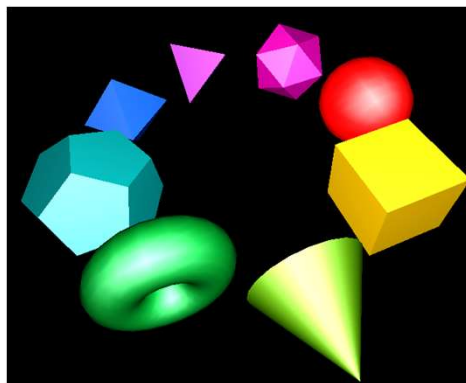
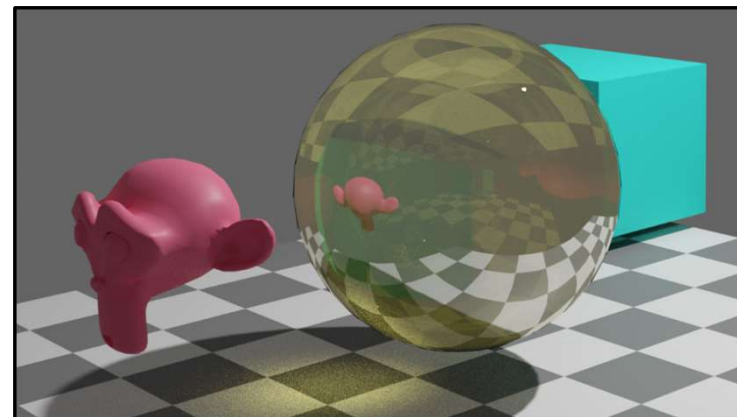
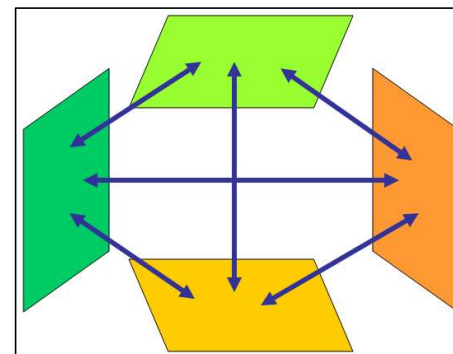


# Rendering

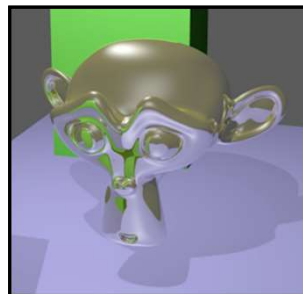
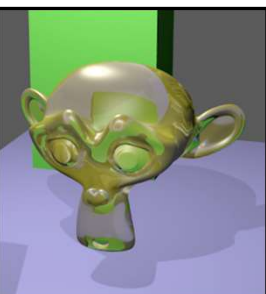
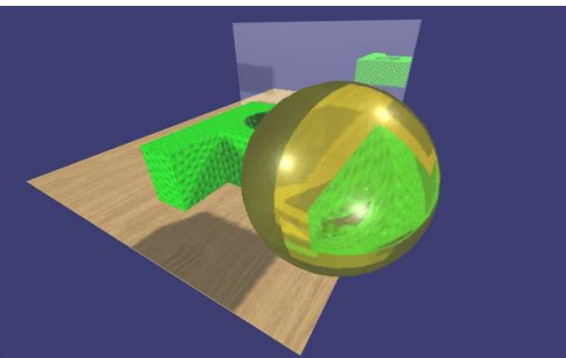


**Oregon State**  
University  
**Mike Bailey**

mjb@cs.oregonstate.edu



Rendering.pptx



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



**Oregon State**  
University  
Computer Graphics

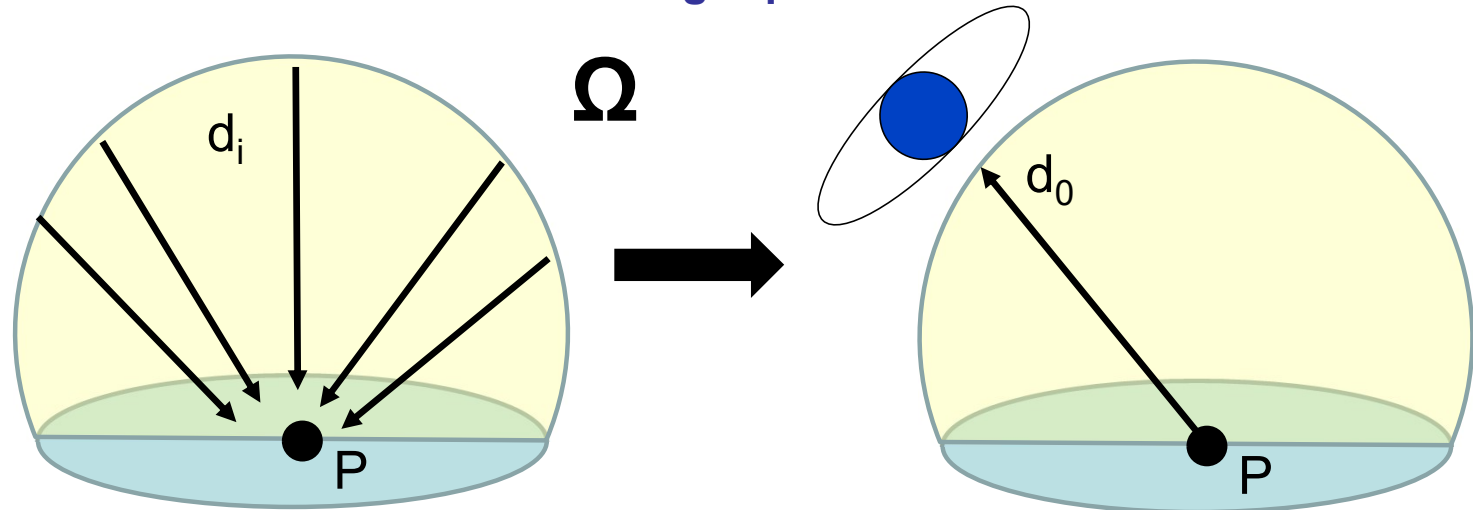
## Rendering

Rendering is the process of creating an image of a geometric model. There are questions you need to ask:

- For what purpose am I doing this?
- How realistic do I need this image to be?
- How much compute time do I have to create this image?
- Do I need to take lighting into account?
- Does the illumination need to be Global or will Local do?
- Do I need to create shadows?
- Do I need to create reflections and refractions?
- How good do the reflections and refractions need to be?



## The Rendering Equation



Light arriving at Point P from everywhere

Light departing from Point P in the direction that we are viewing the scene from

$$B(P, d_0, \lambda) = E(P, d_0, \lambda) + \int_{\Omega} B(P, d_i, \lambda) f(\lambda, d_i, d_0) (d_i \cdot \hat{n}) d\Omega$$

This is the true rendering situation. Essentially, it is an energy balance:

In English, this says that the Light Shining from a point P =

Light emitted by that point + Reflectivity \*  $\Sigma$ (Light arriving from all other points)

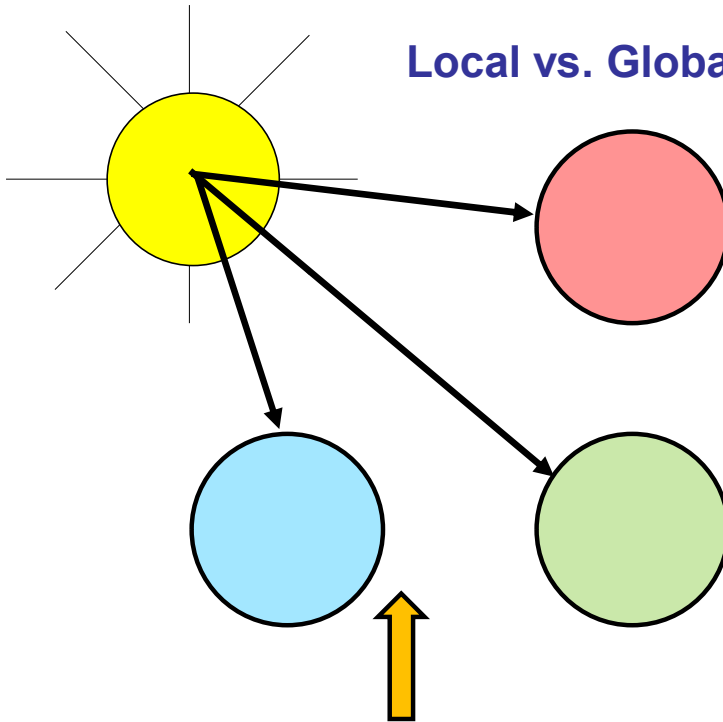
But, this is time-consuming to solve “exactly”.

So, we need to know **how much of an approximation we need**



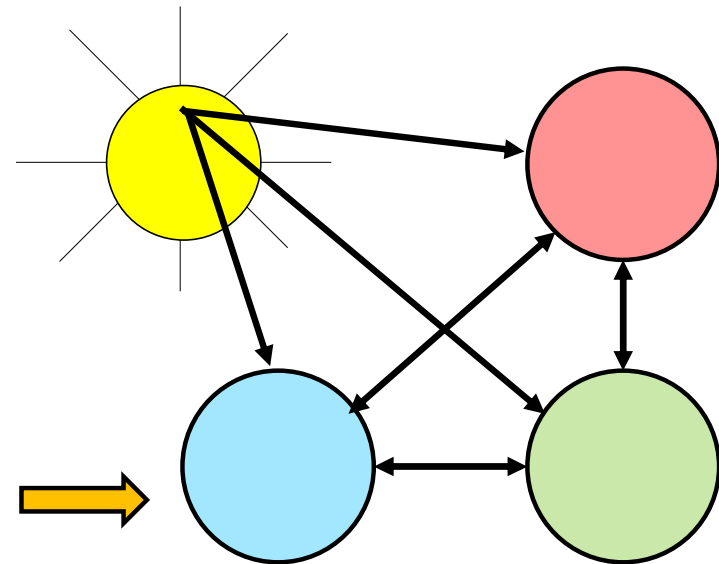
## Local vs. Global Illumination

**Local**



If the appearance of an object is only affected by its own characteristics and the characteristics of the light sources, then you have **Local Illumination**.

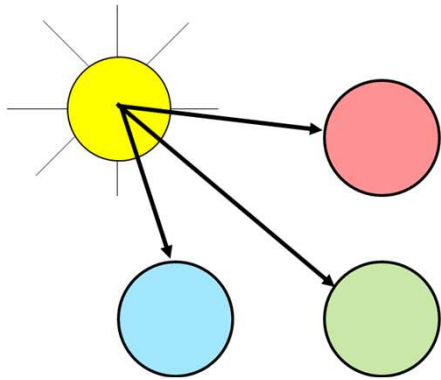
**Global**



If the appearance of an object is also affected by the appearances of other objects, then you have **Global Illumination**.

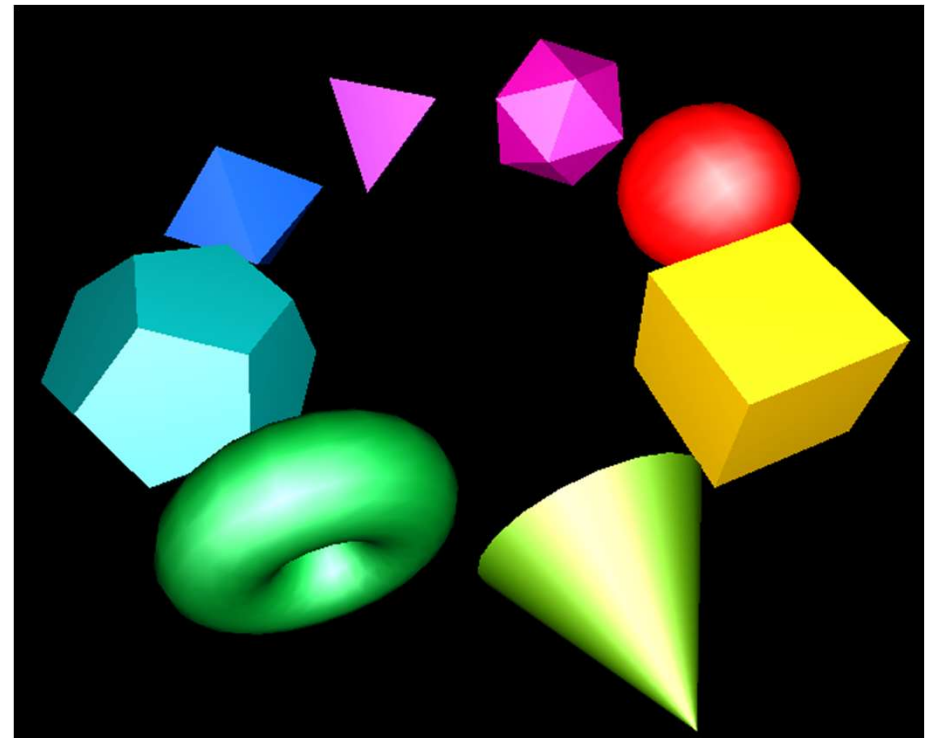


Oregon State  
University  
Computer Science



## Local Illumination at Work

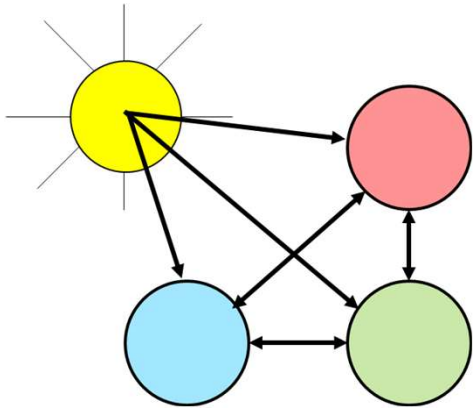
“If the appearance of an object is only affected by its own characteristics and the characteristics of the light sources, then you have **Local Illumination**.”



OpenGL rendering uses Local Illumination



## Global Illumination at Work

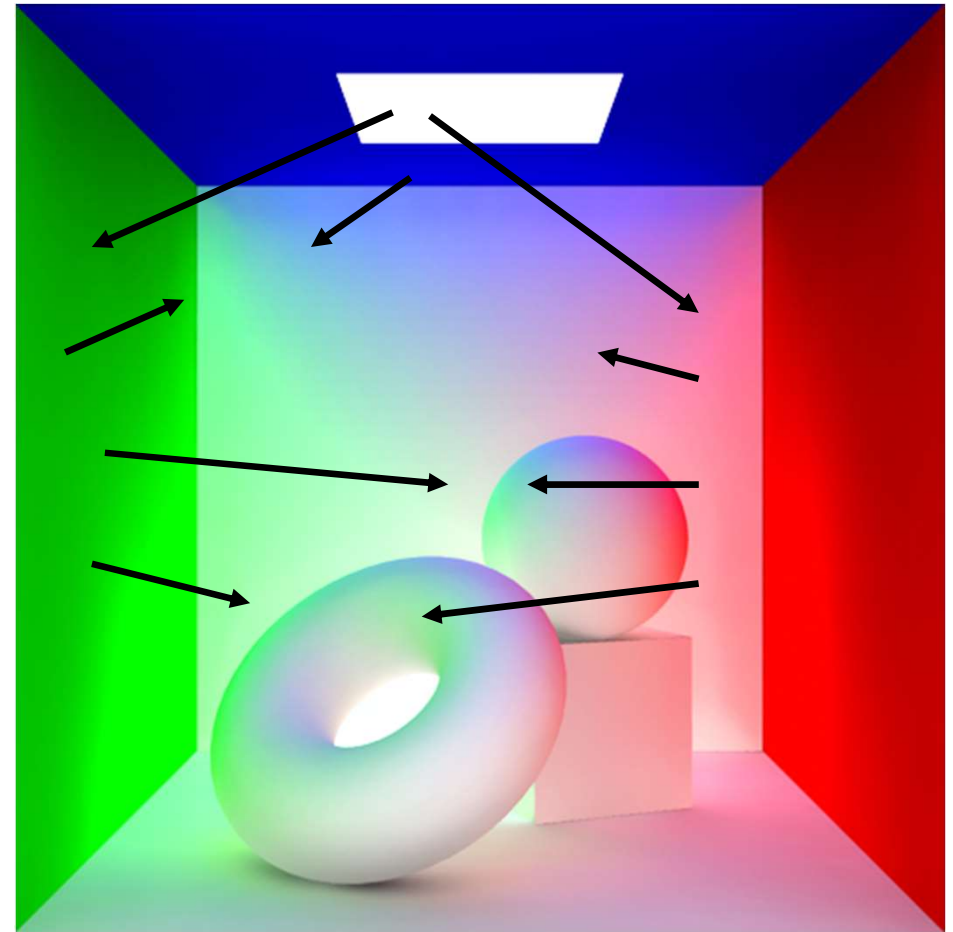


- The left wall is green.
- The right wall is red.
- The back wall is white.
- The ceiling is blue with a light source in the middle of it.
- The objects sitting on the floor are white.

“If the appearance of an object is also affected by the appearances of other objects, then you have **Global Illumination.**”



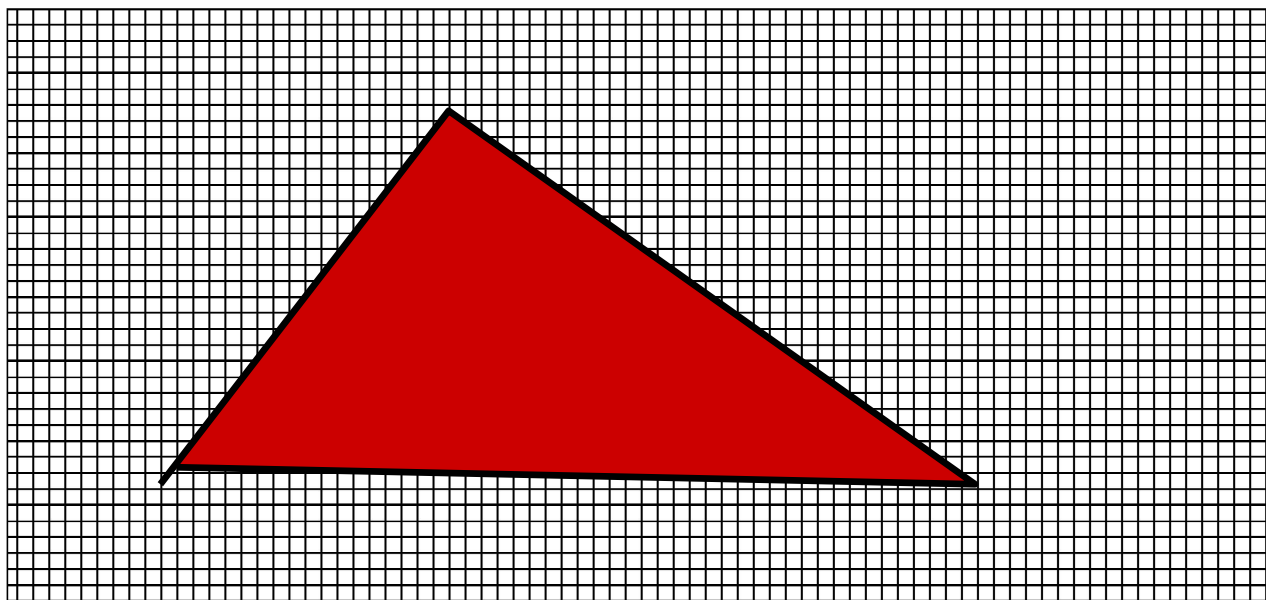
Oregon State  
University  
Computer Graphics



<http://www.swardson.com/unm/tutorials/mentalRay3/>

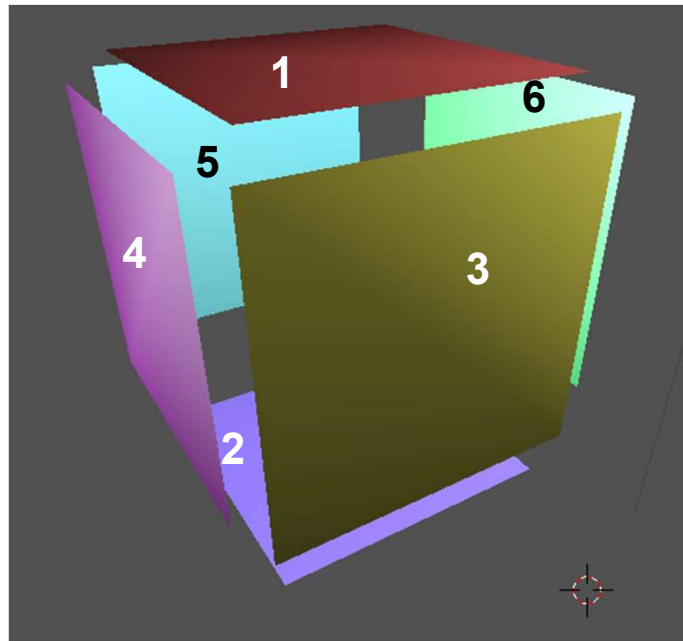
## How Graphics Hardware Renders: Start with the Object, Works Towards the Pixels

- This is the kind of rendering you get on a graphics card (e.g., OpenGL).
- You have been doing this all along.
- Start with the geometry and project it onto the pixels.



## Why do things in front look like they are *really* in front?

Your application might draw this cube's polygons in 1-2-3-4-5-6 order, but 1, 3, and 4 still need to look like they were drawn last:



**Solution #1:** Sort your polygons in 3D by depth and draw them back-to-front.

In this case 1-2-3-4-5-6 becomes 5-6-2-4-1-3.

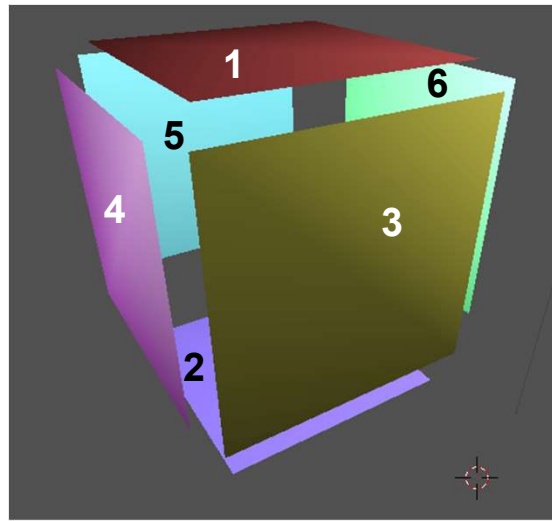
This is called the **Painter's Algorithm**. It sucked to have to do things this way.



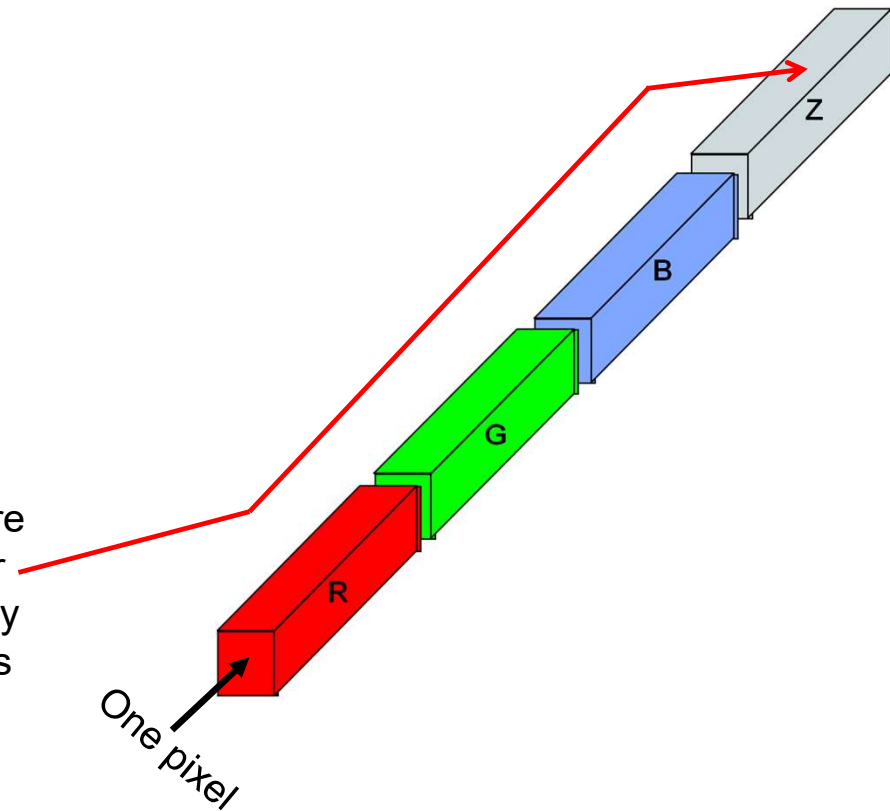


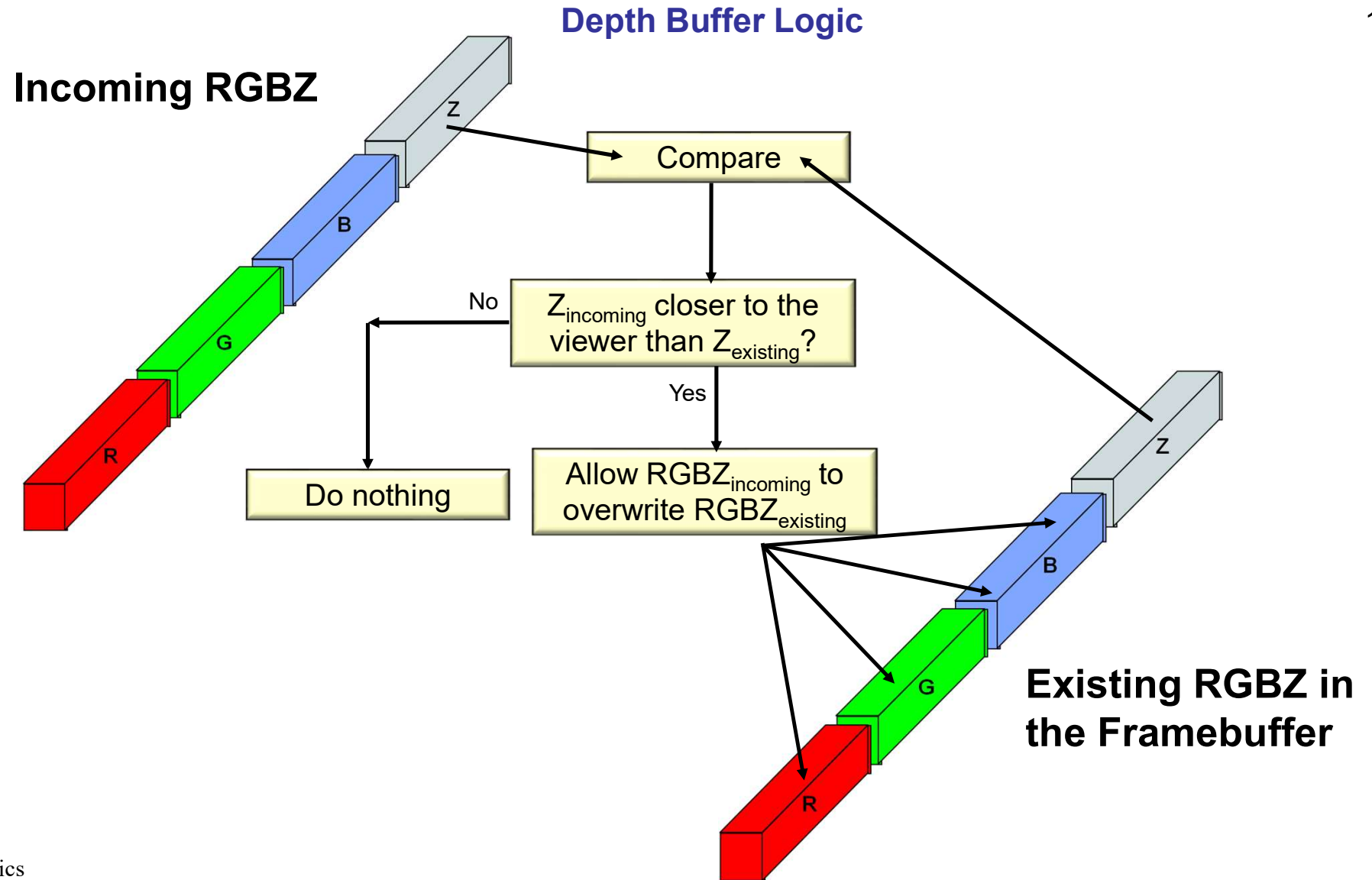
## Why do things in front look like they are *really* in front?

Your application might draw this cube's polygons in 1-2-3-4-5-6 order, but 1, 3, and 4 still need to look like they were drawn last:

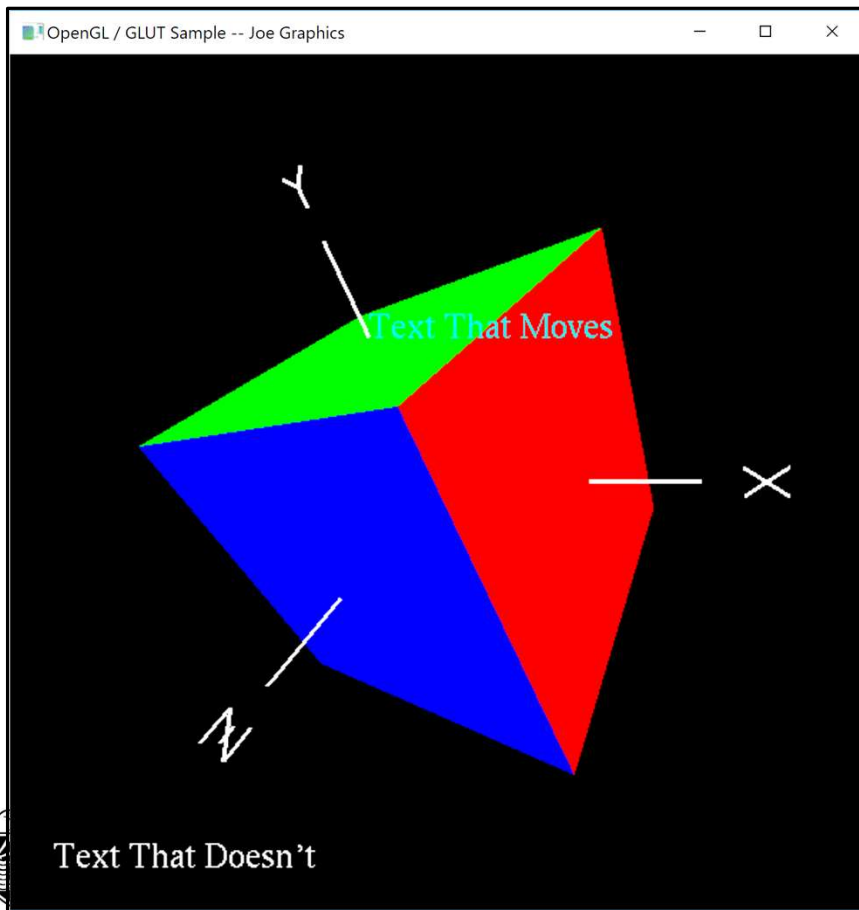


**Solution #2:** Add an extension to the framebuffer to store the depth of each pixel. This is called a **Depth-buffer** or **Z-buffer** (or **Zed-buffer** in other parts of the world). Only allow pixel stores when the depth of the incoming pixel is closer to the viewer than the pixel that is already in that spot in the framebuffer.

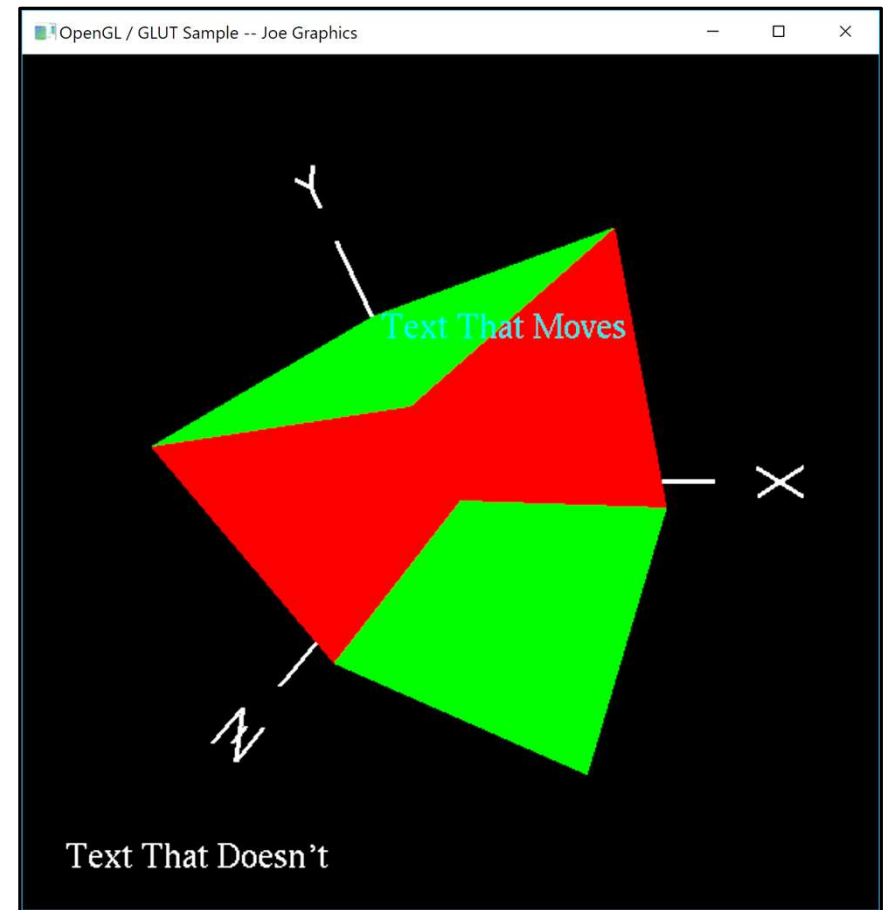




## Why do things in front look like they are *really* in front?

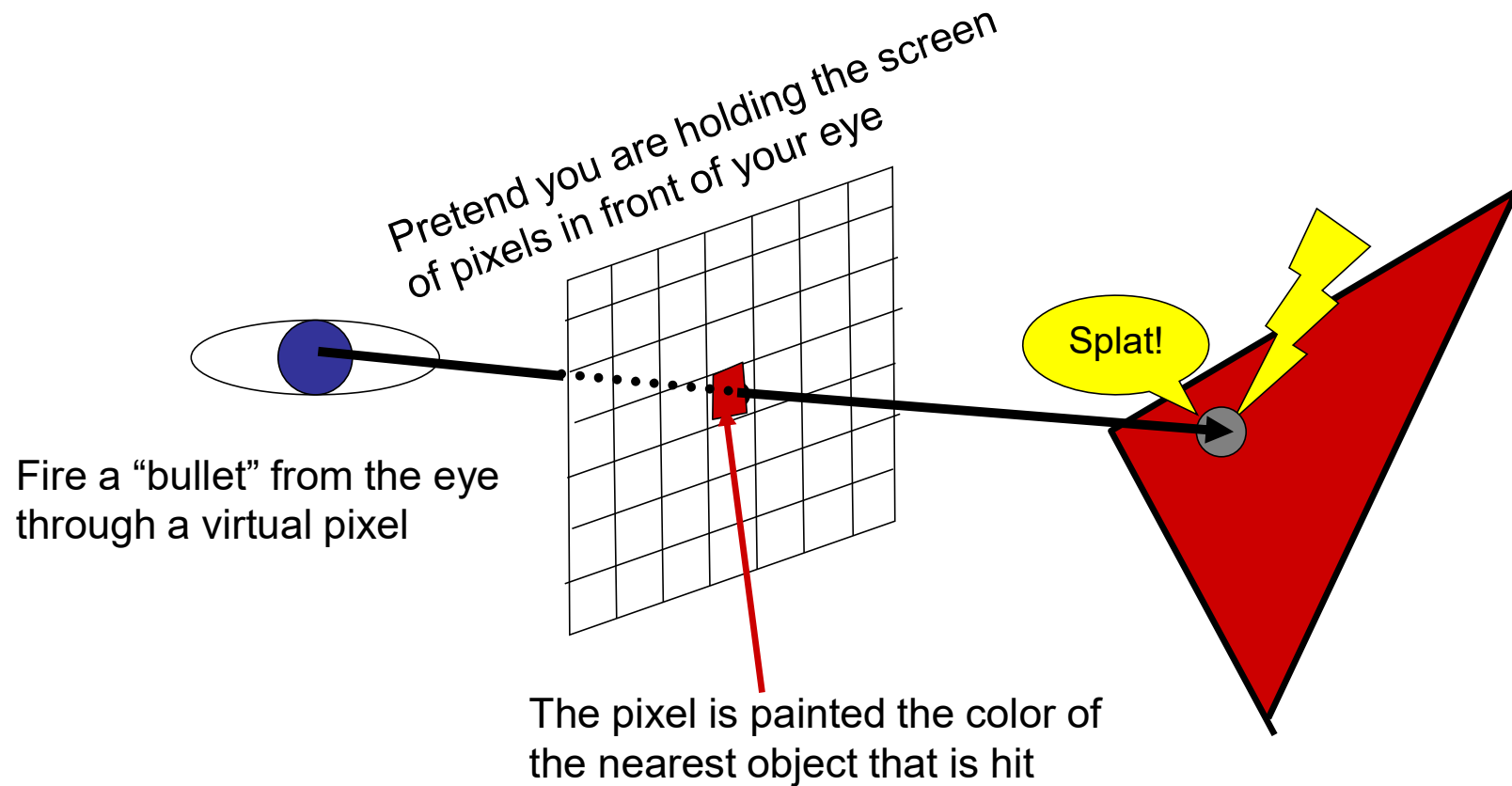


**With Depth Buffer**

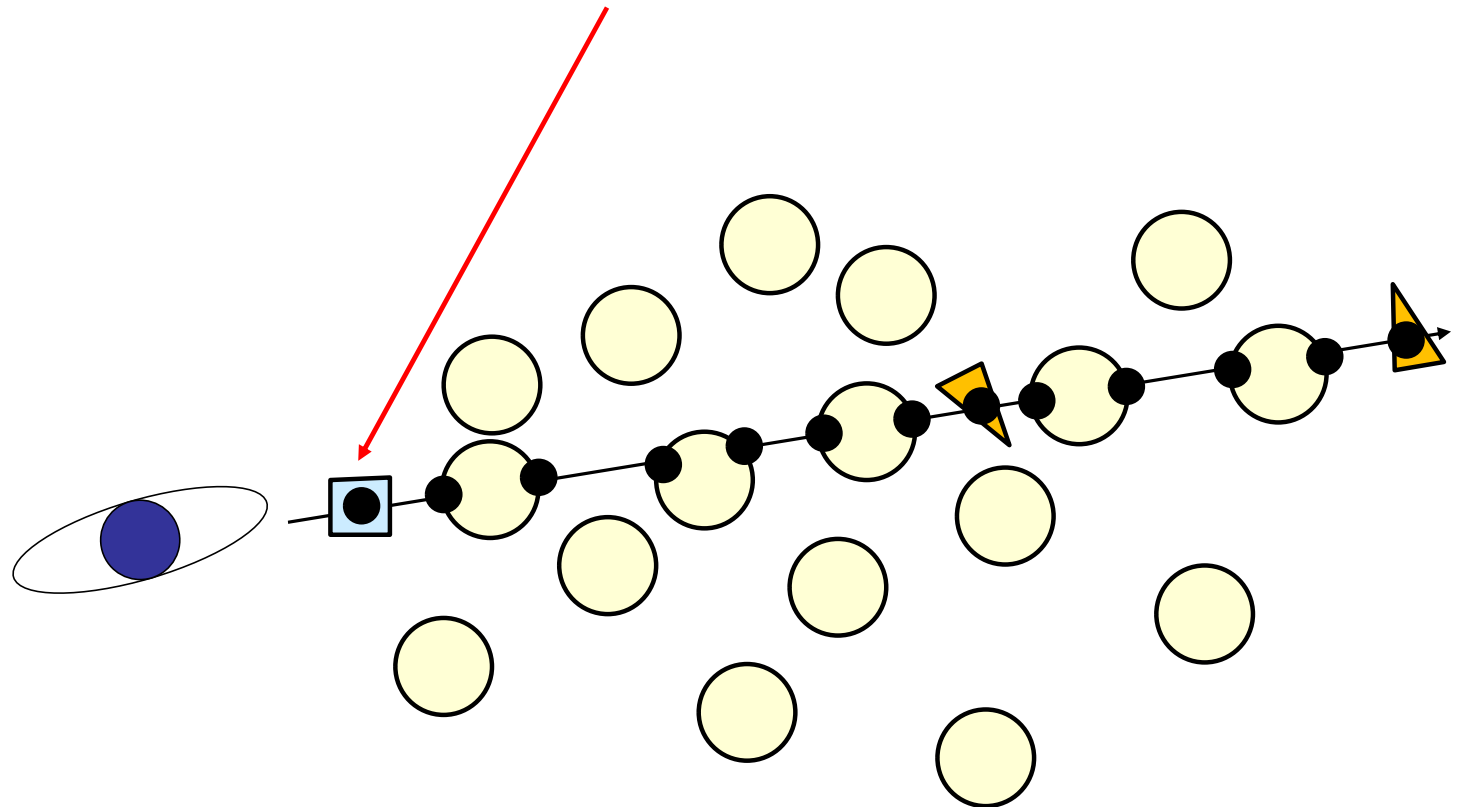


**Without Depth Buffer**

## Ray-Tracing: Start at the Pixels, Work Towards the Objects



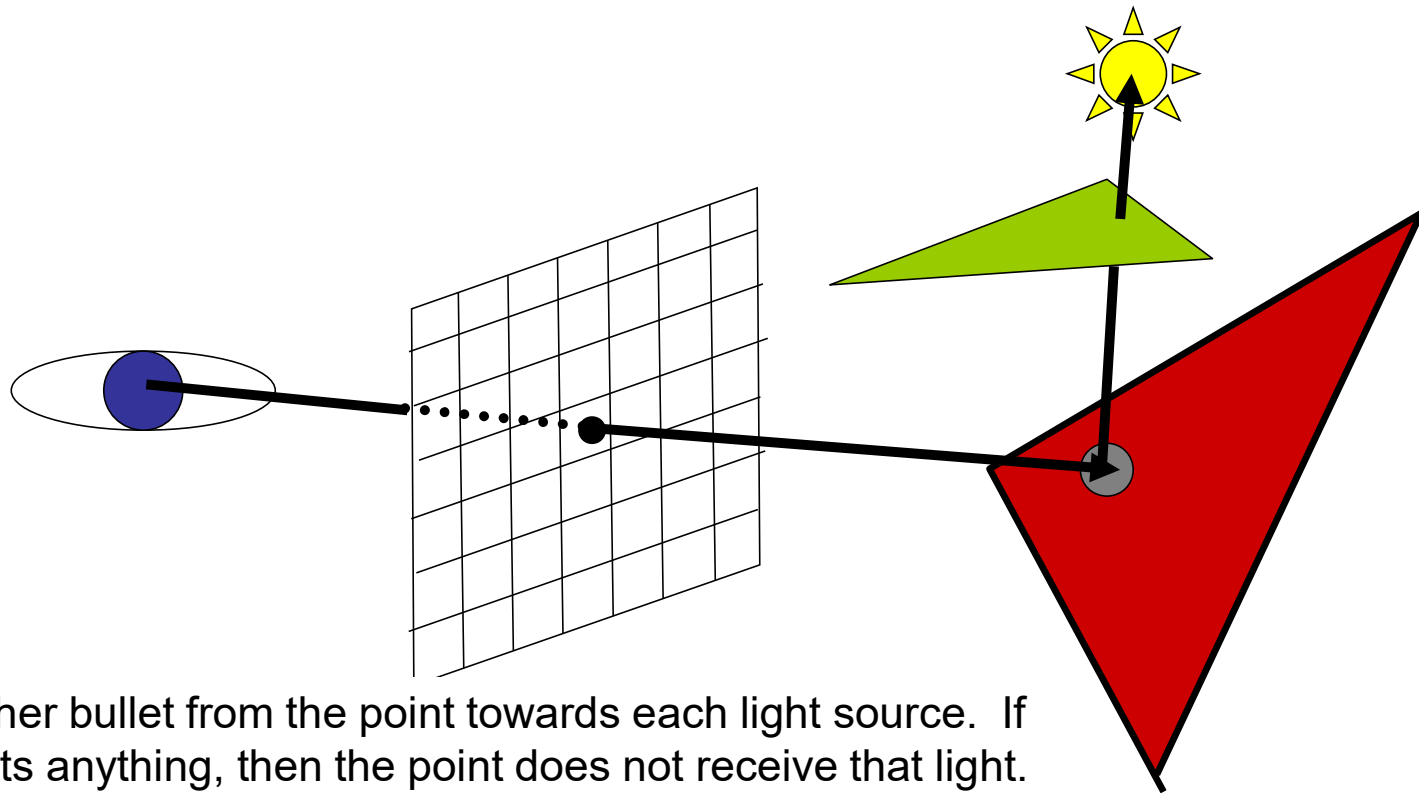
In a Ray-Tracing, each Ray (“bullet”) typically hits a lot of Things –  
You Need to Find All the Hits, then Find the Nearest Hit and work from There



## Ray-Tracing

14

It's also straightforward to see if the closest intersection point lies in a shadow:



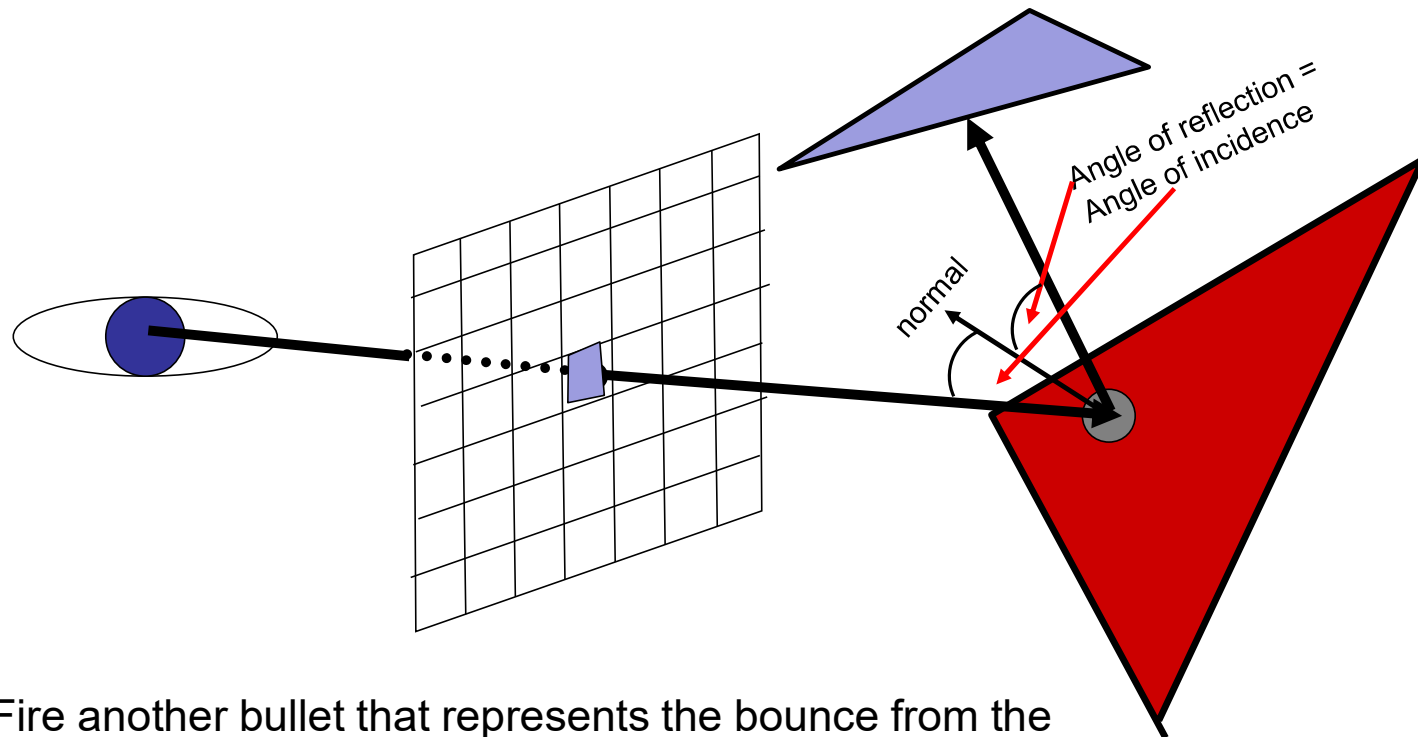
Oregon State  
University  
Computer Graphics

Fire another bullet from the point towards each light source. If the ray hits anything, then the point does not receive that light.

## Ray-Tracing

15

It's also straightforward to handle reflection



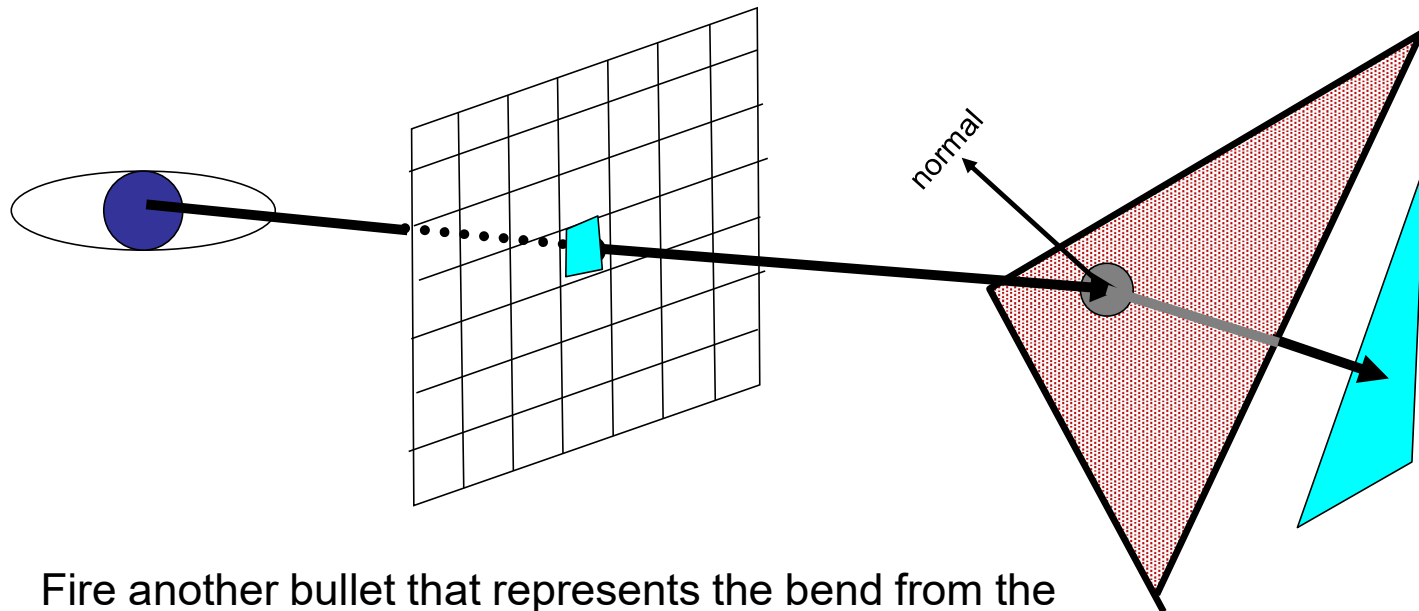
Fire another bullet that represents the bounce from the reflection. Paint the pixel the color that this ray sees.



## Ray-Tracing

16

It's also straightforward to handle refraction



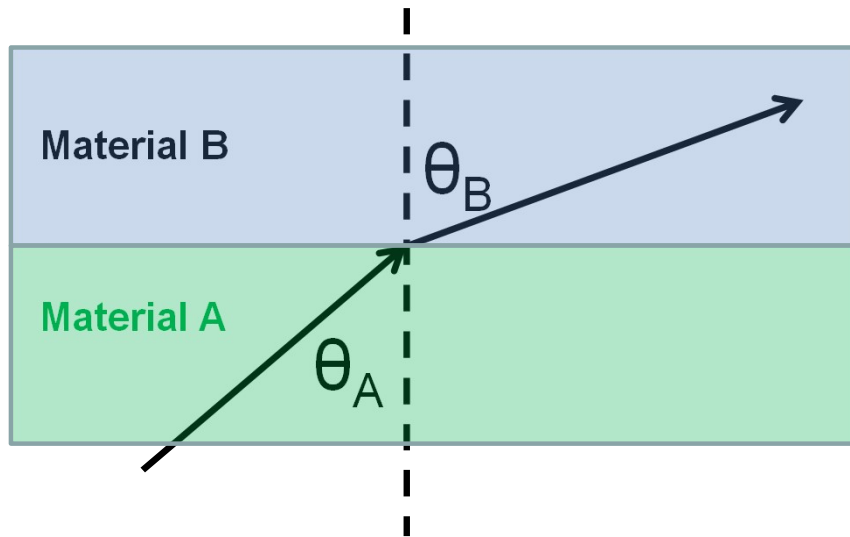
Fire another bullet that represents the bend from the refraction. Paint the pixel the color that this ray sees.





## The Physics of Refraction

17



Snell's Law of Refraction:

$$\frac{\sin \theta_B}{\sin \theta_A} = \frac{\eta_A}{\eta_B}$$



Oregon State

University

Computer Graphics

$\eta$   
↓

Material	Index of Refraction
Vacuum	1.00000
Air	1.00029
Ice	1.309
Water	1.333
Plexiglass	1.49
Glass	1.60
Diamond	2.42

[http://en.wikipedia.org/wiki/Refractive\\_index](http://en.wikipedia.org/wiki/Refractive_index)

## Refraction in Action ☺

18



© 2023, Grimmy, Inc.



Oregon State  
University  
Computer Graphics

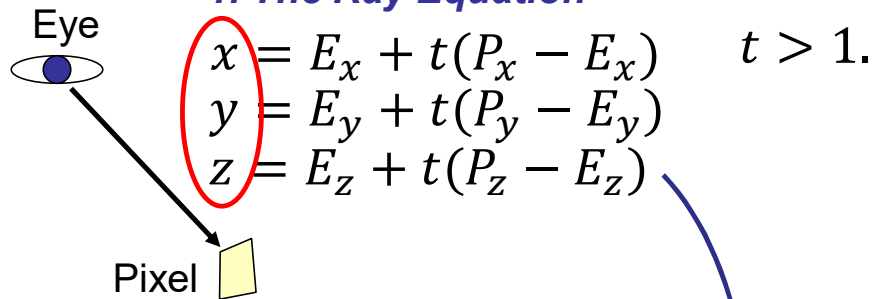
## An Example of How to Determine a Ray-Shape Intersection

### 1. The Ray Equation

Eye

$$\begin{aligned} x &= E_x + t(P_x - E_x) \\ y &= E_y + t(P_y - E_y) \\ z &= E_z + t(P_z - E_z) \end{aligned} \quad t > 1.$$

Pixel



### 4. Solve using the dreaded high school Quadratic Formula

$$At^2 + Bt + C = 0$$

### 2a. Substitute 3. Collect terms

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

### Three cases of possible solutions:

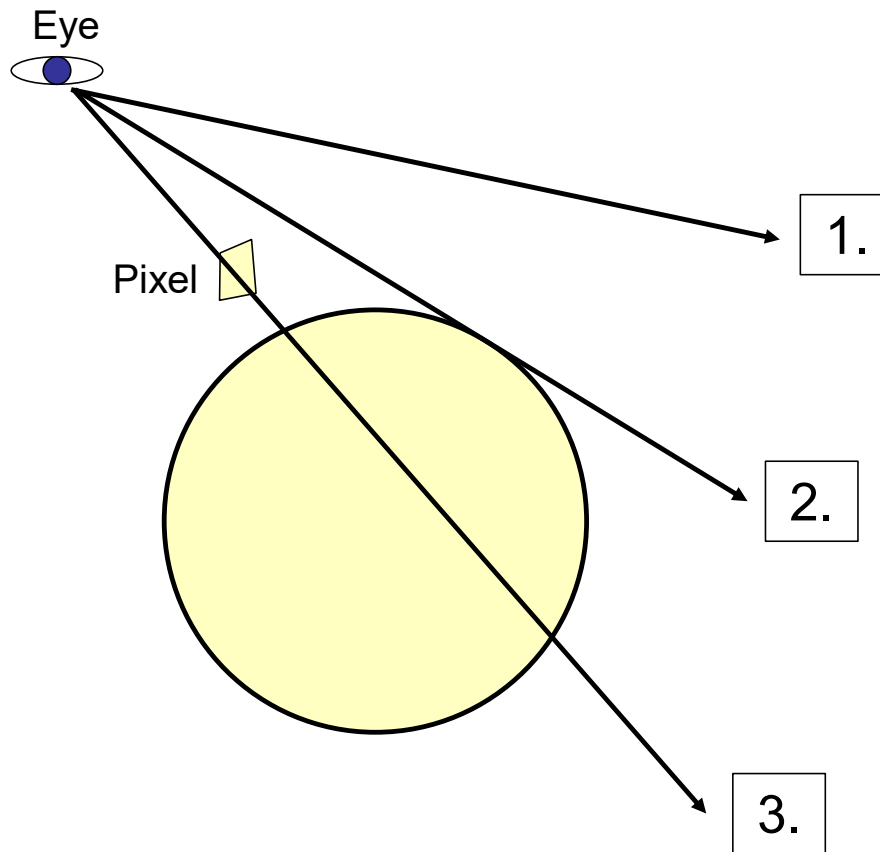
1. Both t's are complex: ray missed the sphere
2. Both t's are real and identical: ray is tangent to the sphere
3. Both t's are real and different: ray goes through the sphere

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2$$

### 2b. ... into the Sphere Equation and expand



## An Example of How to Determine a Ray-Shape Intersection

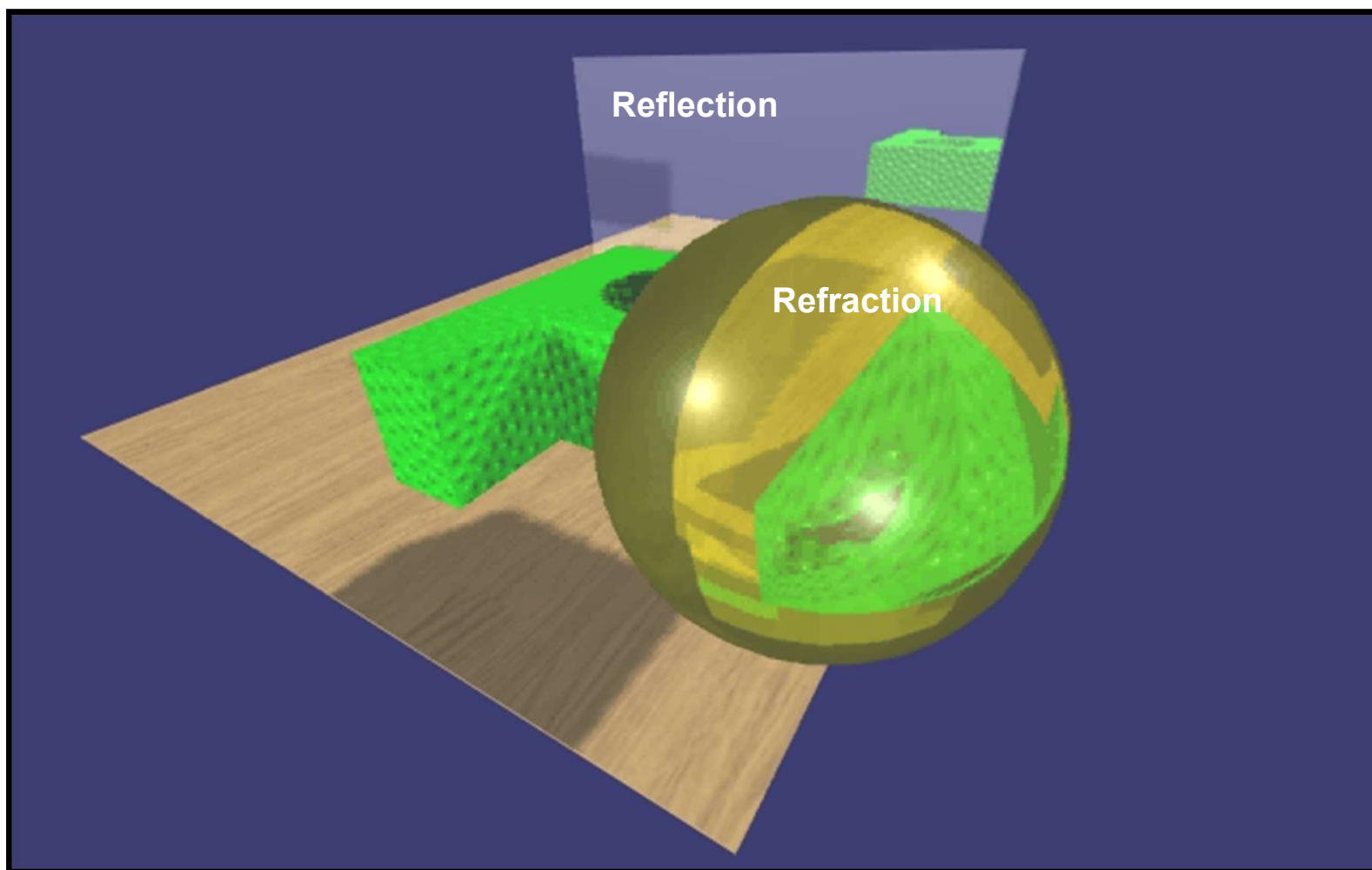


### Three cases of possible solutions:

1. Both  $t$ 's are complex: ray missed the sphere
2. Both  $t$ 's are real and identical: ray is tangent to the sphere
3. Both  $t$ 's are real and different: ray goes through the sphere



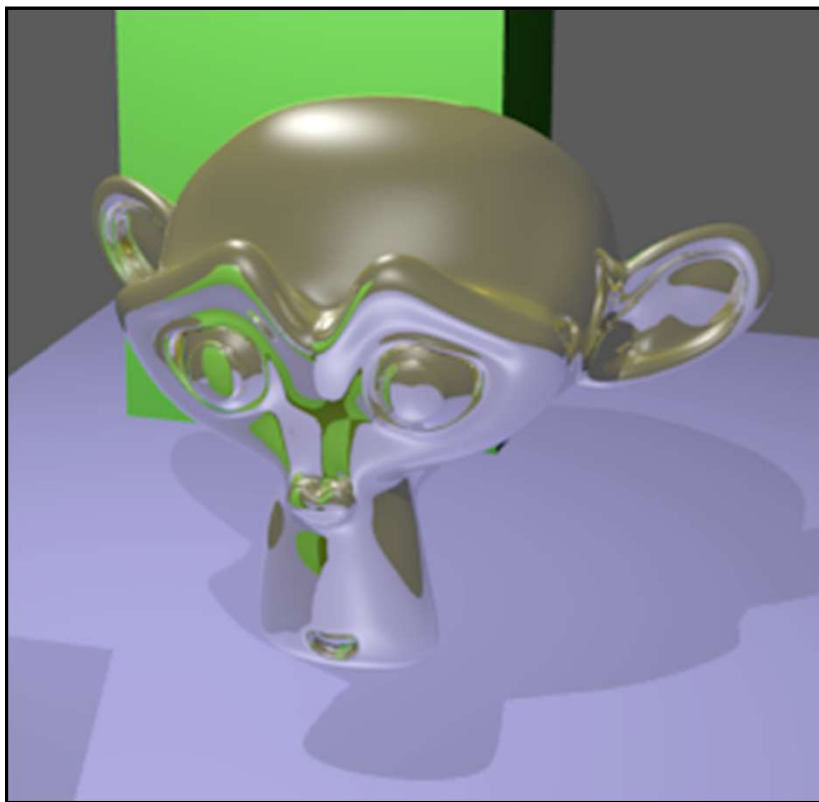
## IronCAD Ray-Tracing Example



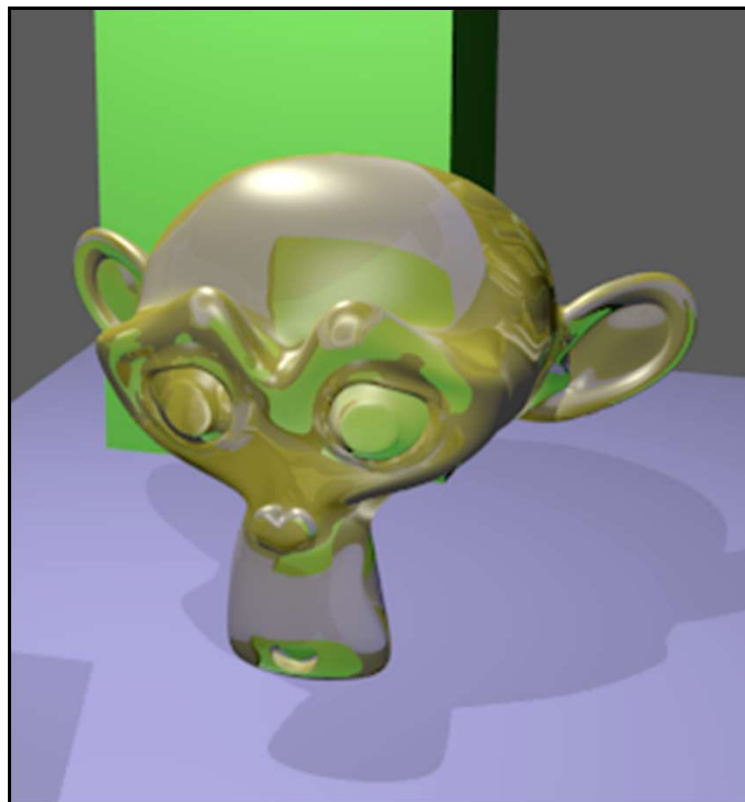
Oregon State  
University

Computer Graphics

## Blender Ray-Tracing Examples



Reflection

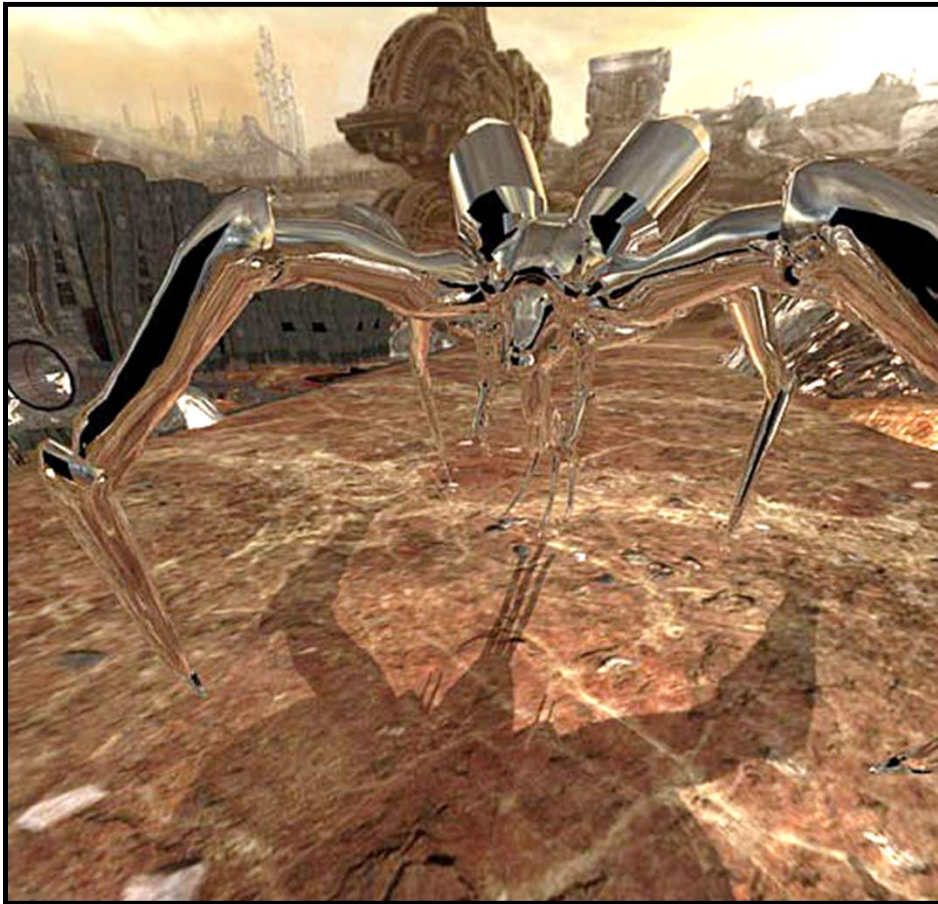


Refraction





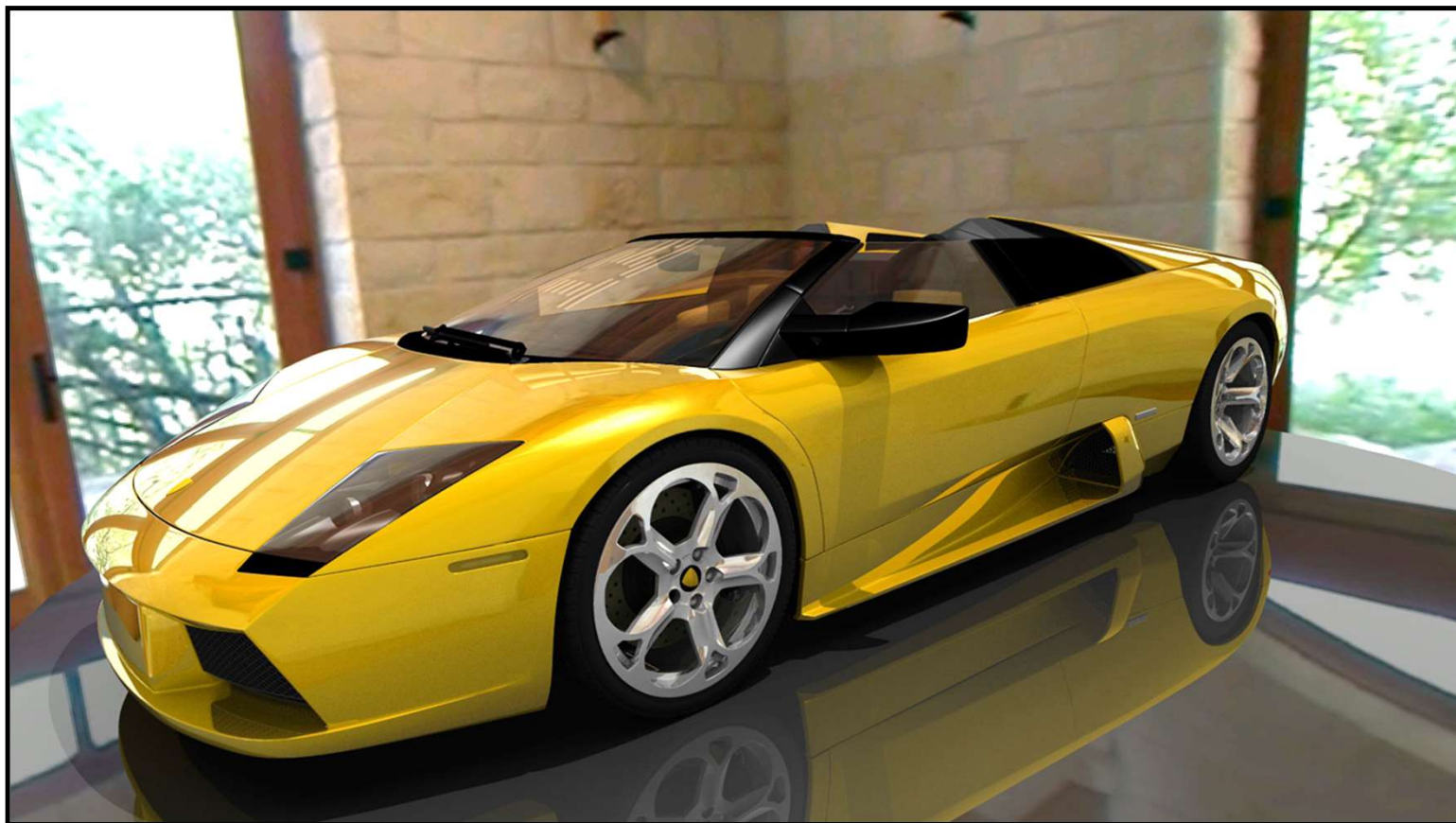
## More Ray-tracing Examples



Quake 4 Ray-Tracing Project



## More Ray-Tracing Examples





## More Ray-Tracing Examples

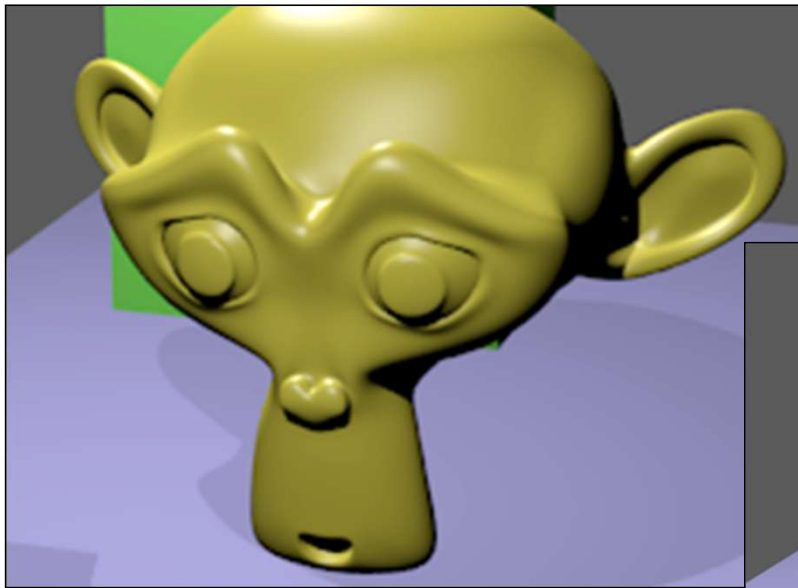


Bunkspeed



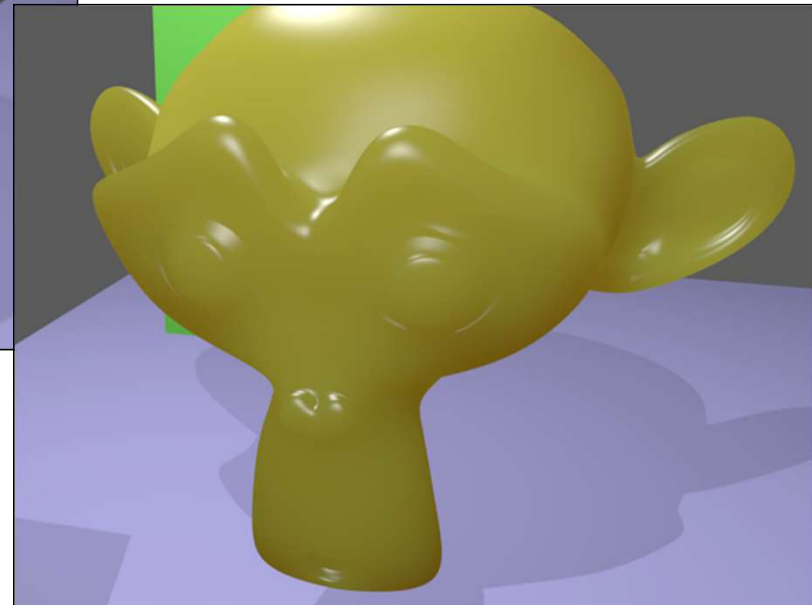
## Subsurface Scattering

- Subsurface Scattering mathematically models light bouncing around within an object before coming back out.
- This is a good way to render skin, wax, milk, paraffin, etc.



Original rendering

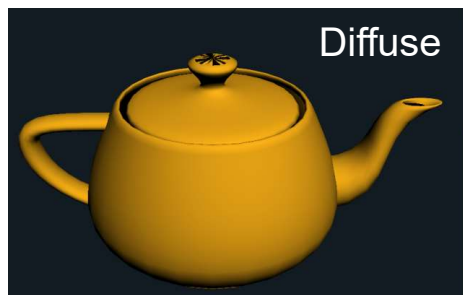
Subsurface scattering



## The Three Elements of OpenGL Lighting



+



+



=



The biggest problem with the Ambient-Diffuse-Specular way of computing lighting is that we are trying to match an appearance, not necessarily follow the laws of physics.

For example, using A-D-S, you can easily create a scene where the amount of light shining from the objects exceeds the amount of light that the light source is supplying!

This brings us to **Physically-Based Rendering (PBR)**.

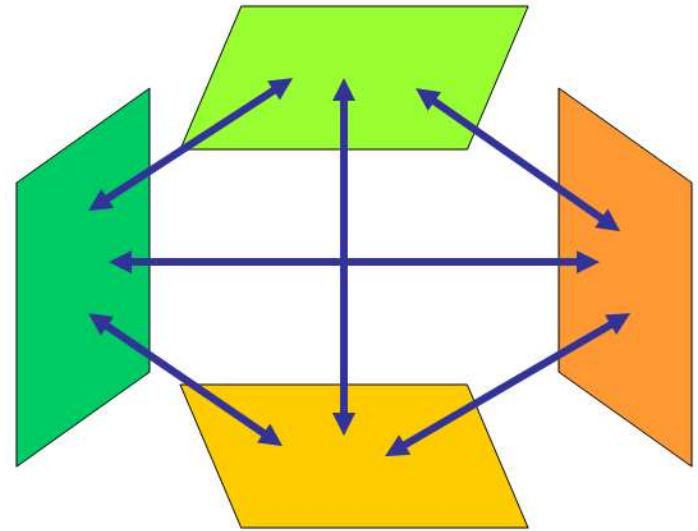


## Radiosity

Based on the idea that all surfaces gather light intensity from all other surfaces

The fundamental radiosity equation is an energy balance that says:

“The light energy leaving surface  $i$  equals the amount of light energy generated by surface  $i$  plus surface  $i$ ’s reflectivity times the amount of light energy arriving from all other surfaces”



$$B_i A_i = E_i A_i + \rho_i \sum_j B_j A_j F_{j \rightarrow i}$$



This is a good approximation to the **Rendering Equation**

## The Radiosity Equation

$$B_i A_i = E_i A_i + \rho_i \sum_j B_j A_j F_{j \rightarrow i}$$

$B_i$  is the light energy intensity shining from surface element  $i$

$A_i$  is the area of surface element  $i$

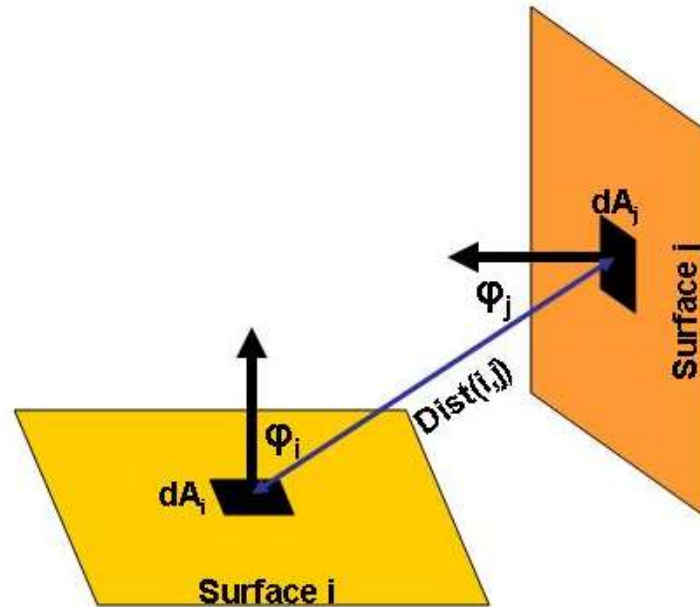
$E_i$  is the internally-generated light energy intensity for surface element  $i$

$\rho_i$  is surface element  $i$ 's reflectivity

$F_{j \rightarrow i}$  is referred to as the **Shape Factor**, and describes what percent of the energy leaving surface element  $j$  arrives at surface element  $i$



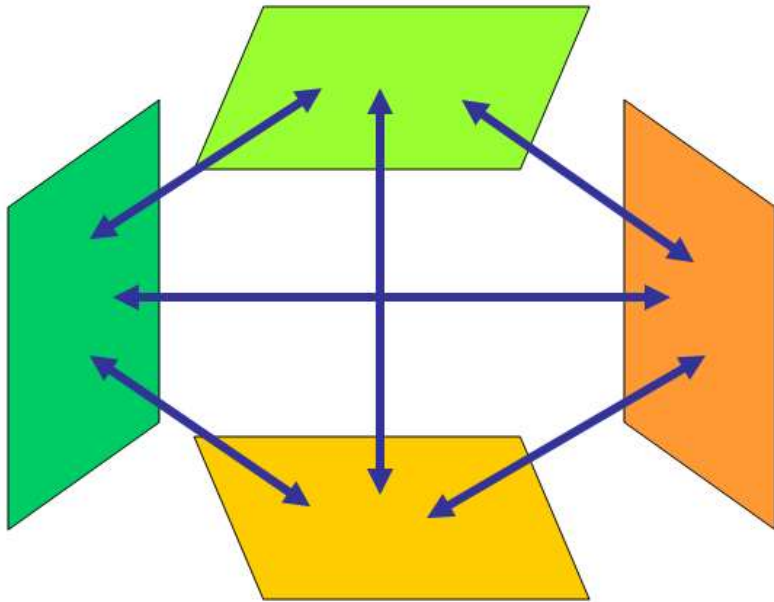
## The Radiosity Shape Factor



$$F_{j \rightarrow i} = \int_{A_i} \int_{A_j} visibility(di, dj) \frac{\cos \Theta_i \cos \Theta_j}{\pi Dist(di, dj)^2} dA_j dA_i$$



I know what you're thinking:  
It seems to you that the light just keeps propagating and you never actually get an answer?



To many people, radiosity seems like this:

Use x to get y  $y = 3x + 5$

Then use y to get x  $x = y - 7$

The equations are connected by curved arrows forming a cycle, with a large 'X' over the equations, indicating a common misconception of infinite recursion.

“x produces y, then y produces x,  
then x produces y, then ...”

Not really – it is simply N equations, N unknowns – you solve for the unique solution

$$\begin{aligned} -3x + y &= 5 \\ x - y &= -7 \end{aligned}$$

$$\begin{aligned} x &= 1 \\ y &= 8 \end{aligned}$$



## The Radiosity Matrix Equation

$$B_i A_i = E_i A_i + \rho_i \sum_j B_j A_j F_{j \rightarrow i} \quad \Longrightarrow \quad B_i A_i - \rho_i \sum_j B_j A_j F_{j \rightarrow i} = E_i A_i$$

Expand for each surface element, and re-arrange  
to **solve for the surface intensities, the  $B$ 's**:

$$\begin{bmatrix} 1 - \rho_1 F_{1 \rightarrow 1} & -\rho_1 F_{1 \rightarrow 2} & \bullet \bullet \bullet & -\rho_1 F_{1 \rightarrow N} \\ -\rho_2 F_{2 \rightarrow 1} & 1 - \rho_2 F_{2 \rightarrow 2} & \bullet \bullet \bullet & -\rho_2 F_{2 \rightarrow N} \\ \bullet \bullet \bullet & \bullet \bullet \bullet & \bullet \bullet \bullet & \bullet \bullet \bullet \\ -\rho_N F_{N \rightarrow 1} & -\rho_N F_{N \rightarrow 2} & \bullet \bullet \bullet & 1 - \rho_N F_{N \rightarrow N} \end{bmatrix} \begin{Bmatrix} B_1 \\ B_2 \\ \bullet \bullet \bullet \\ B_N \end{Bmatrix} = \begin{Bmatrix} E_1 \\ E_2 \\ \bullet \bullet \bullet \\ E_N \end{Bmatrix}$$

This is a lot of equations!





## Radiosity Examples



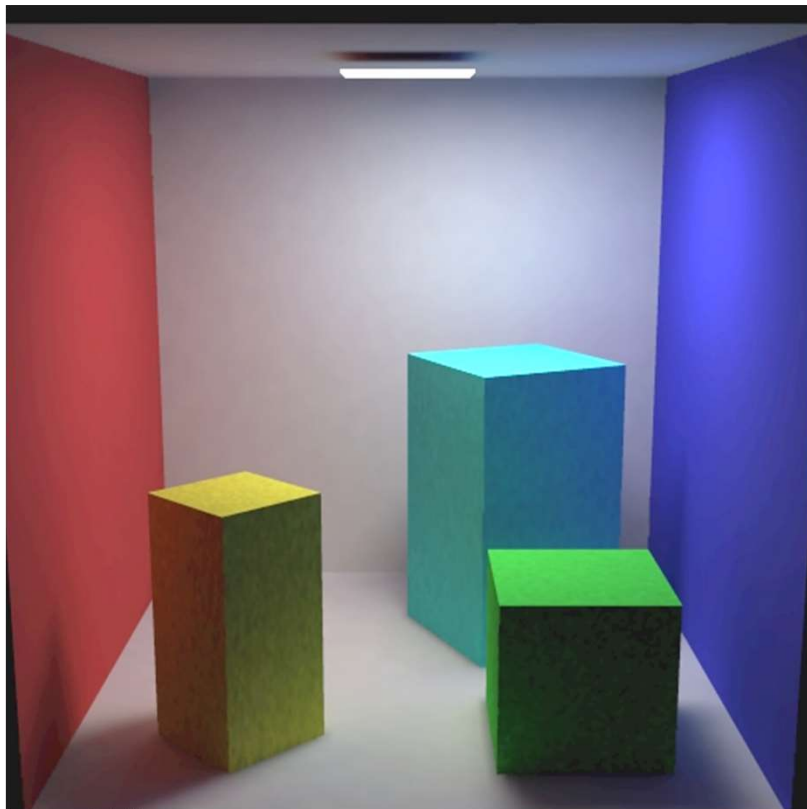
Cornell University



Cornell University



## Radiosity Examples



**AR Toolkit**

**Autodesk**

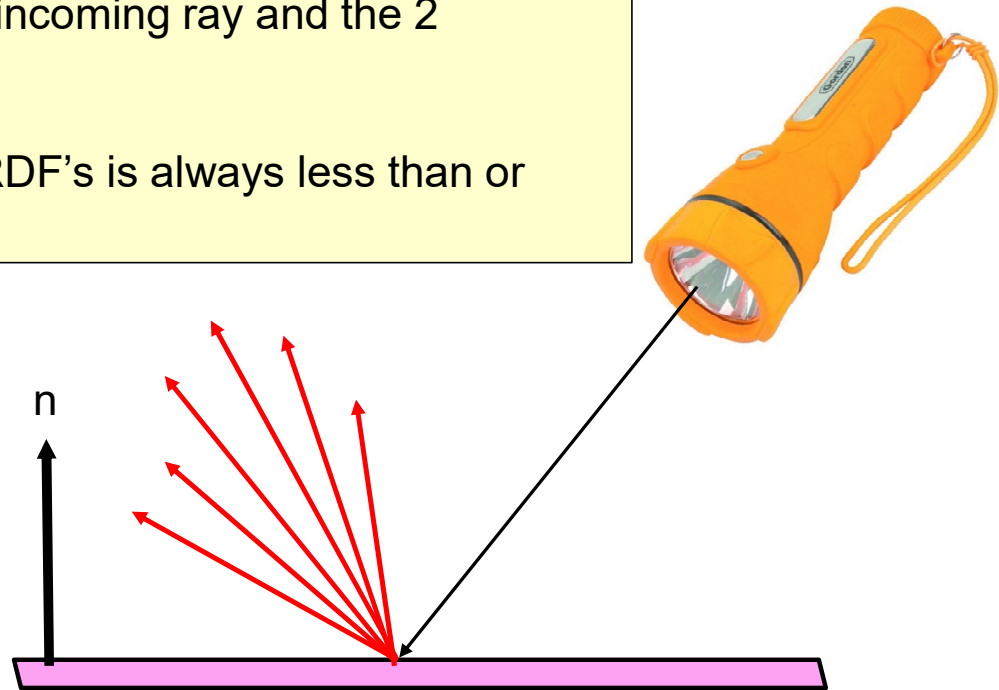
**Oregon State**  
University  
Computer Graphics

## Path Tracing: When light hits a surface, it bounces in particular ways depending on the angle and the material

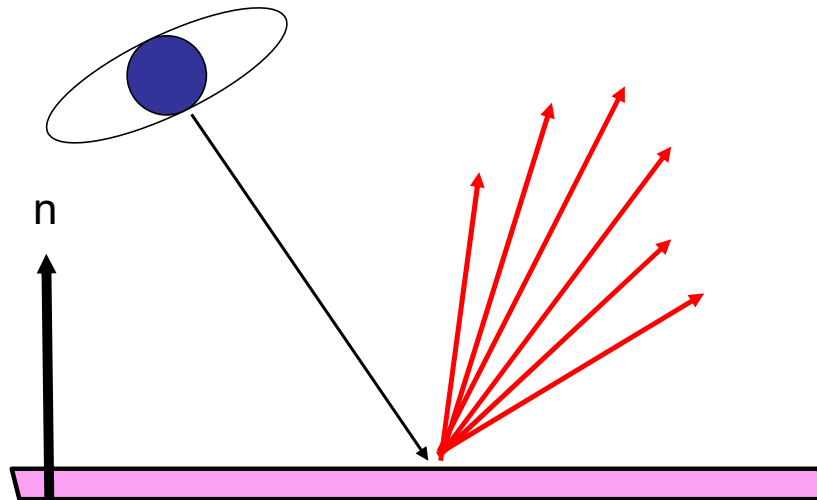
This distribution of bounced light rays is called the **Bidirectional Reflectance Distribution Function**, or **BRDF**.

For a given material, the BRDF behavior of a light ray is a function of 4 variables: the 2 spherical coordinates of the incoming ray and the 2 spherical coordinates of the outgoing ray.

The outgoing light energy in the outgoing BRDF's is always less than or equal to the amount of light that shines in.

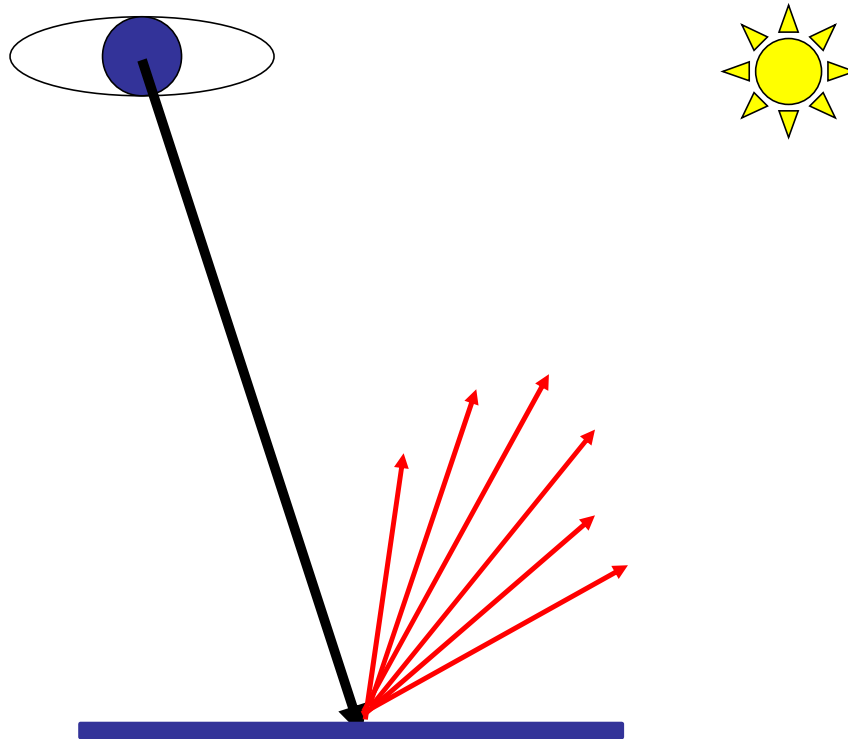


Usually it is easier to trace from the eye

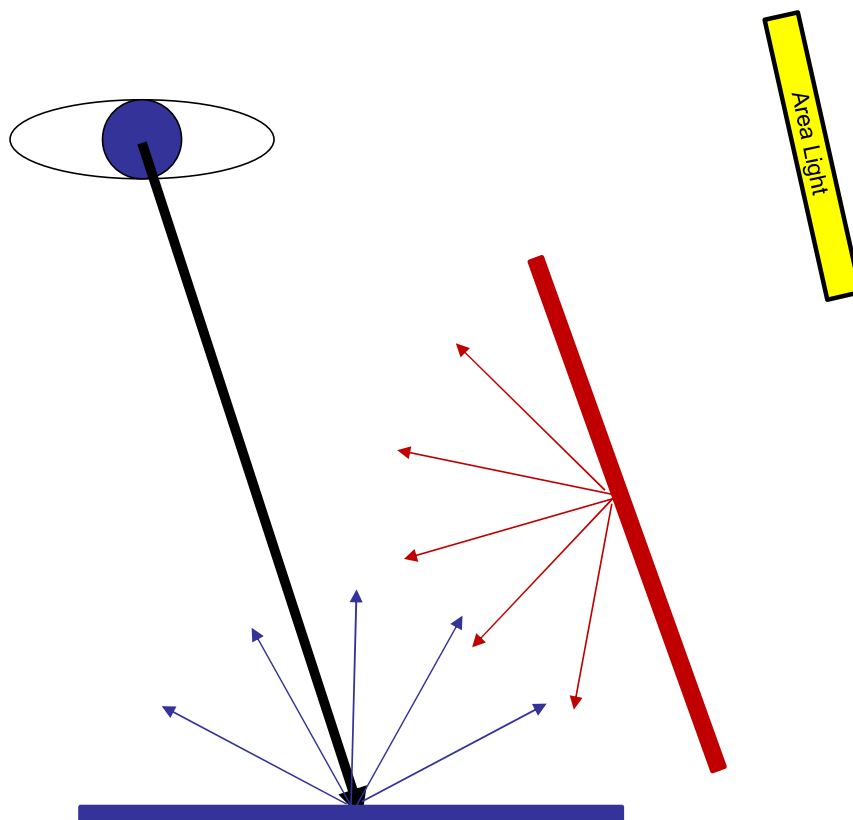


## Path Tracing

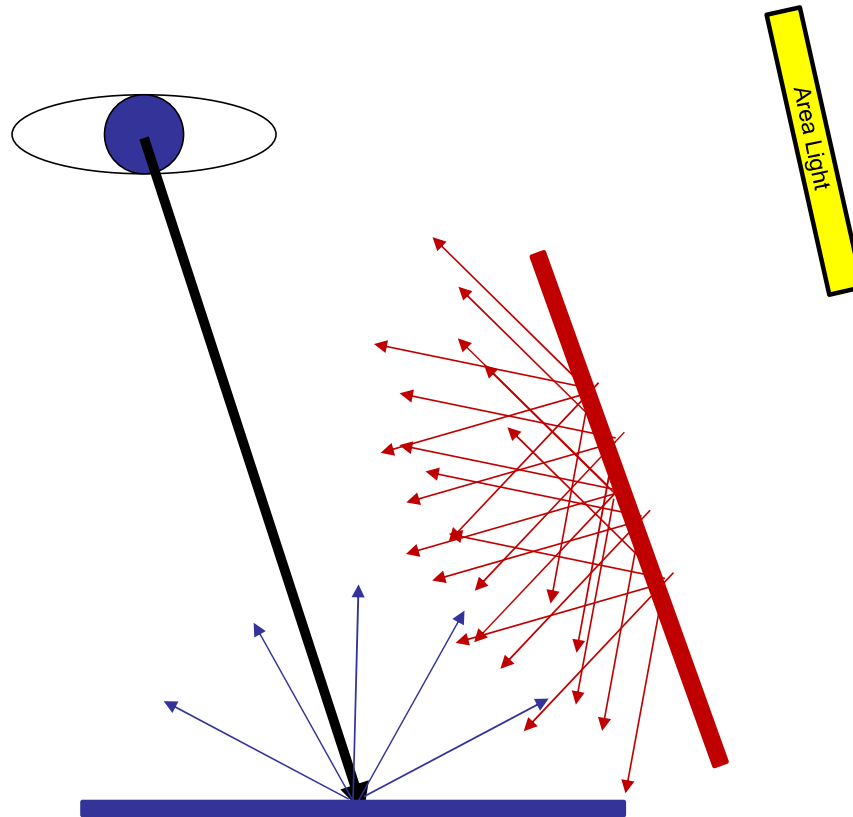
Somewhat like ray-tracing, somewhat like radiosity where light can bounce around the scene but this has more sophisticated effects.

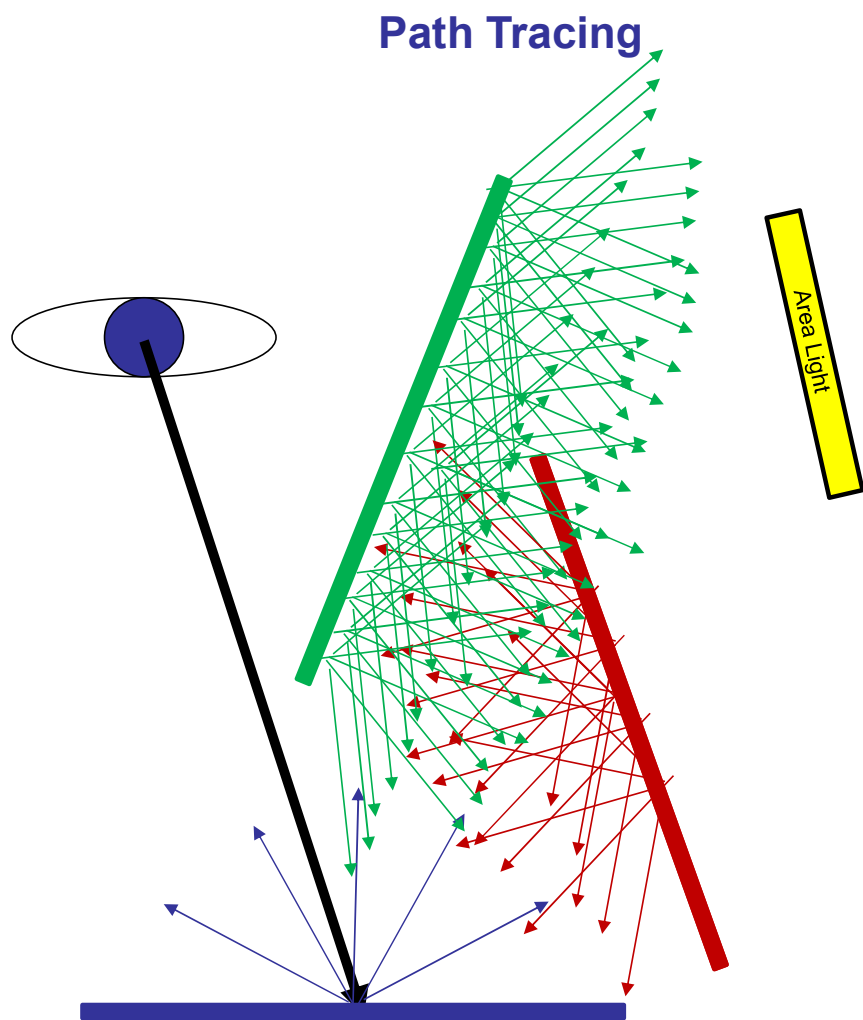


## Path Tracing



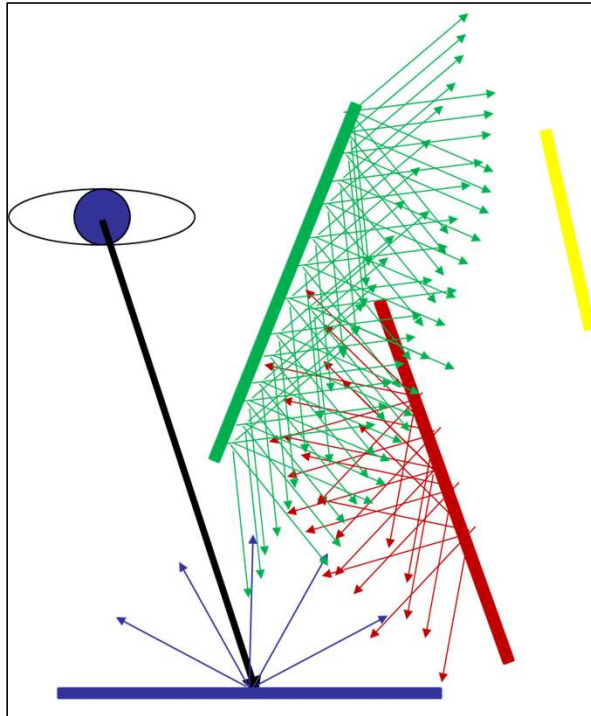
## Path Tracing







## Path Tracing



Clearly this is capable of spawning an infinite number of rays. How do we handle this?

**Monte Carlo** simulation to the rescue!

Each time a ray hits a surface, use the equation at that point. Continue until:

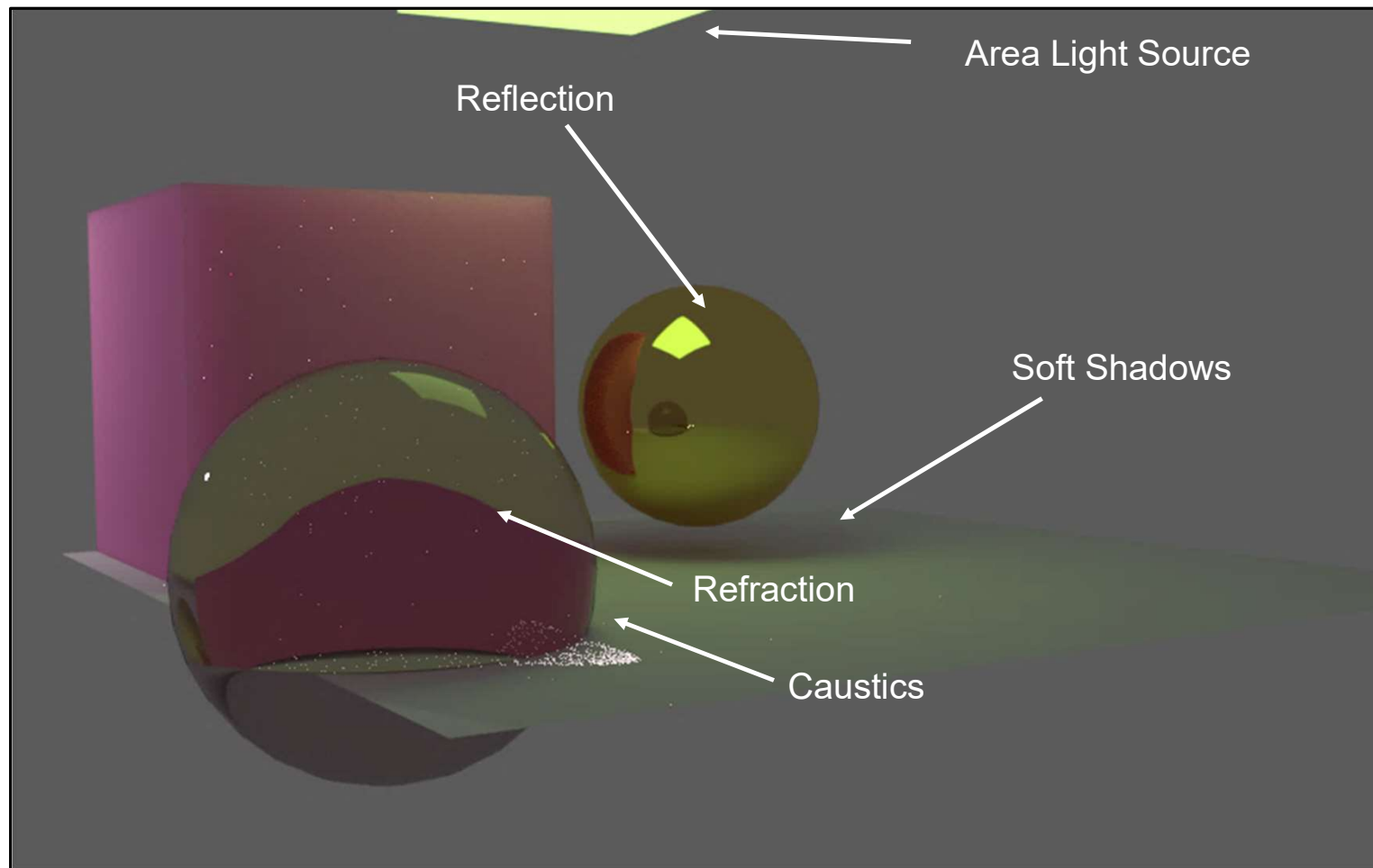
1. Nothing is hit
2. A light is hit
3. Some maximum number of bounces are found

$$LightGathered = \frac{\sum_{0}^{N-1} ResultOfRaysCastInRandomDirection}{N}$$

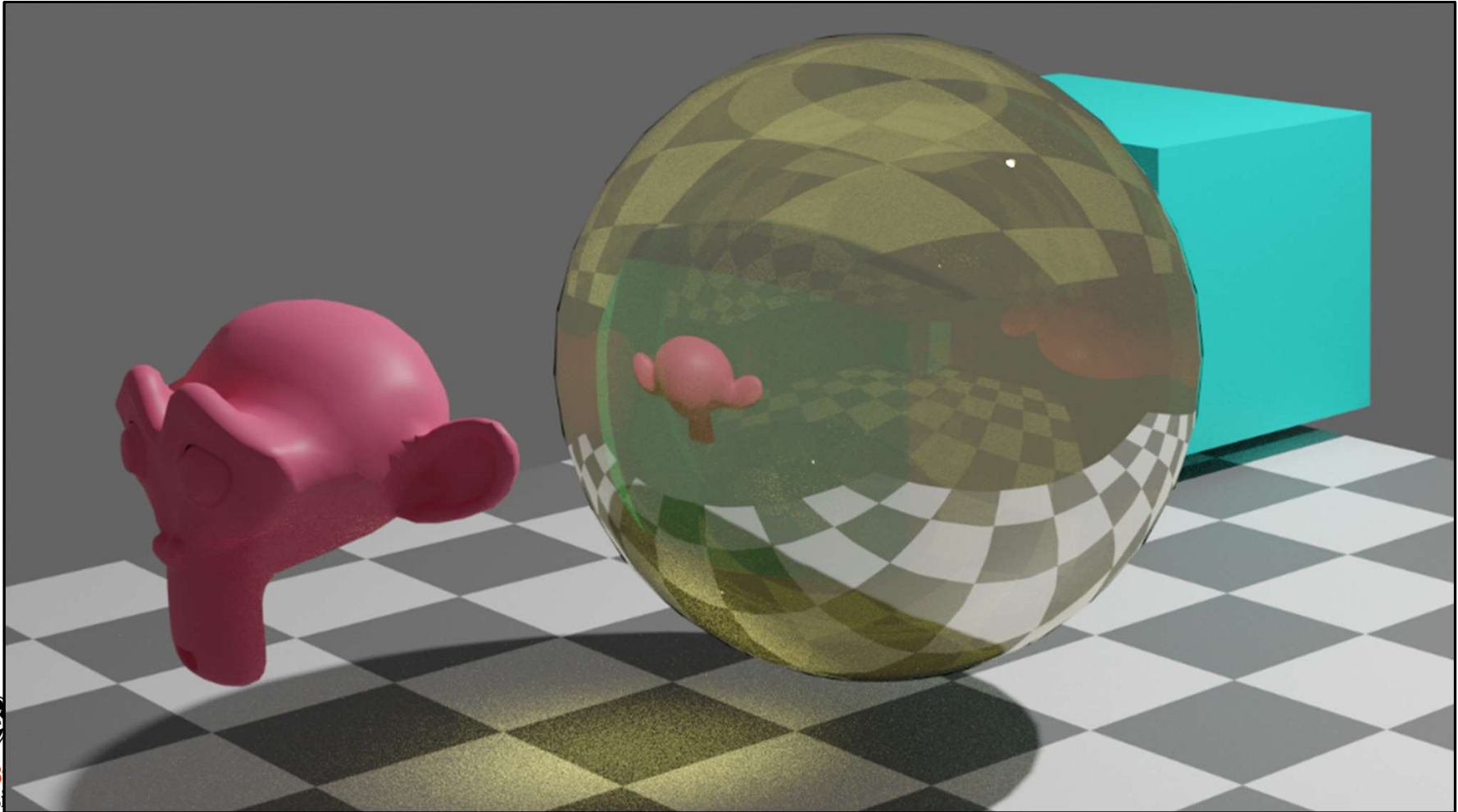
**Recurse by applying this equation for all ray hits (yikes!)**



## Physically-Based Rendering using the Blender *Cycles* Renderer



## Physically-Based Rendering using the Blender *Cycles* Renderer



Oregon S  
Univers

Computer Graphics

## Interesting Mix of Surface Properties: Mmmmm, Gummies!

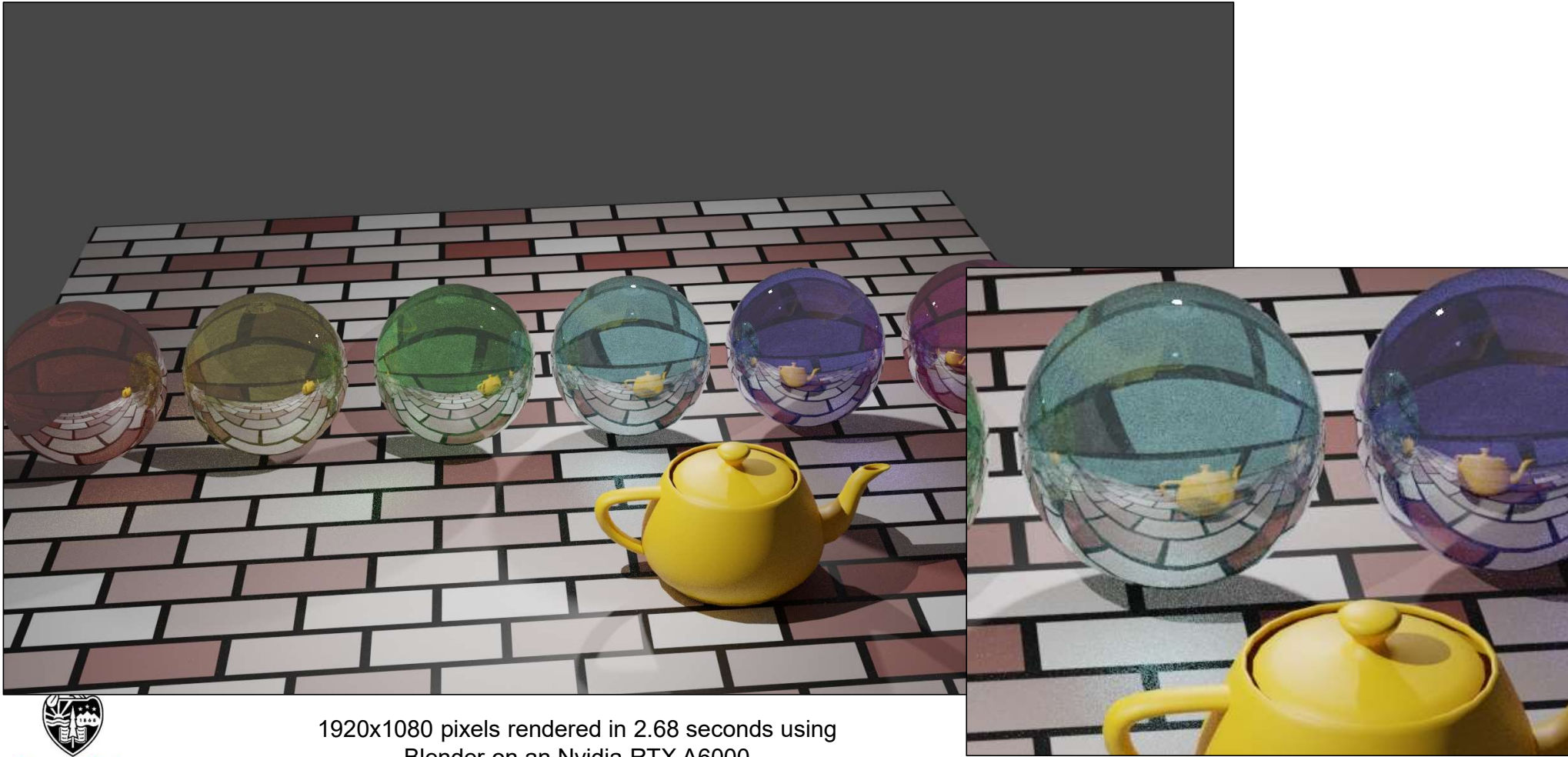


Image by Grace Todd

Oregon State  
University  
Computer Graphics



## Hardware Ray-Tracing



1920x1080 pixels rendered in 2.68 seconds using  
Blender on an Nvidia RTX A6000



Oregon State  
University  
Computer Graphics

## Another Physically-Based Rendering Example

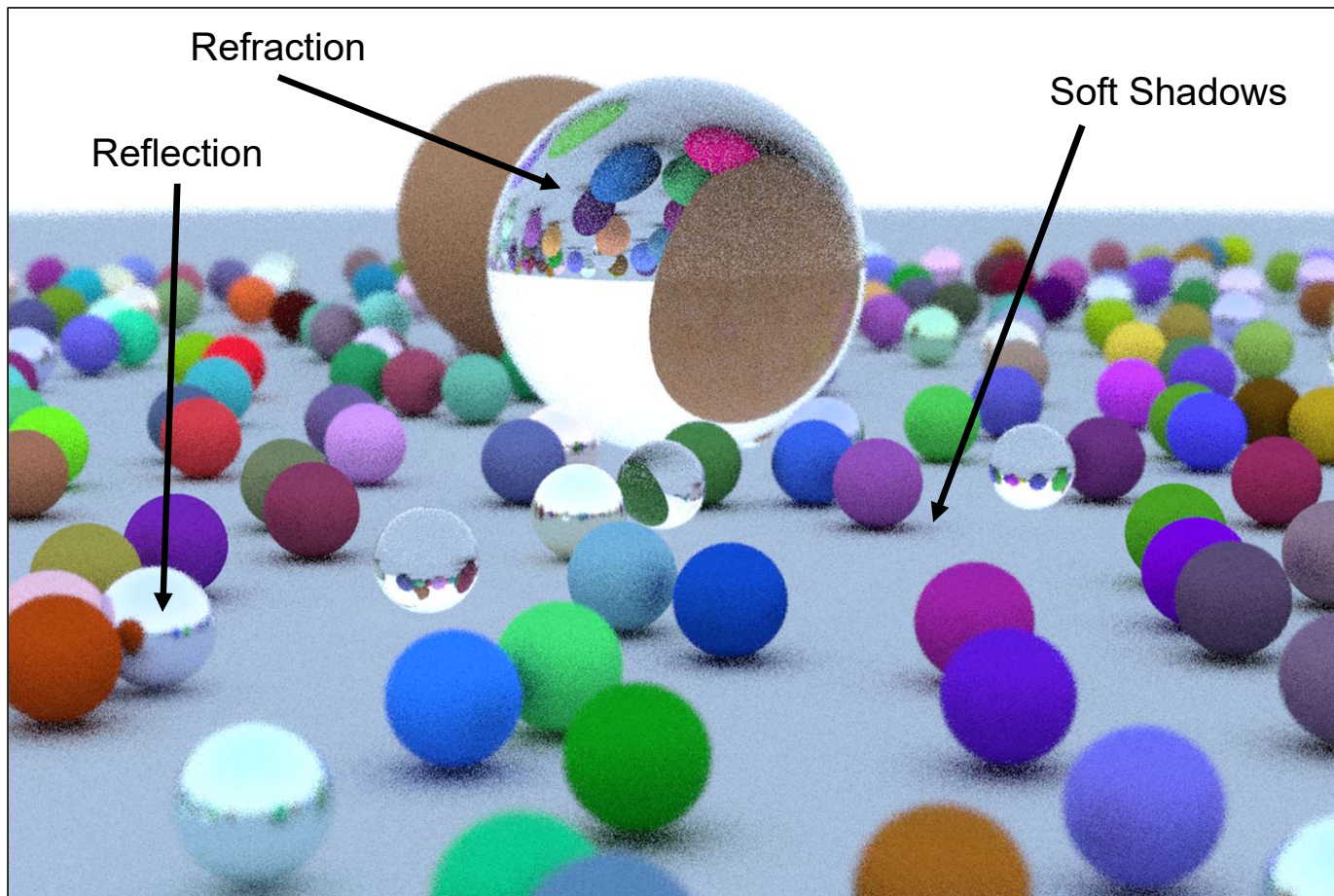
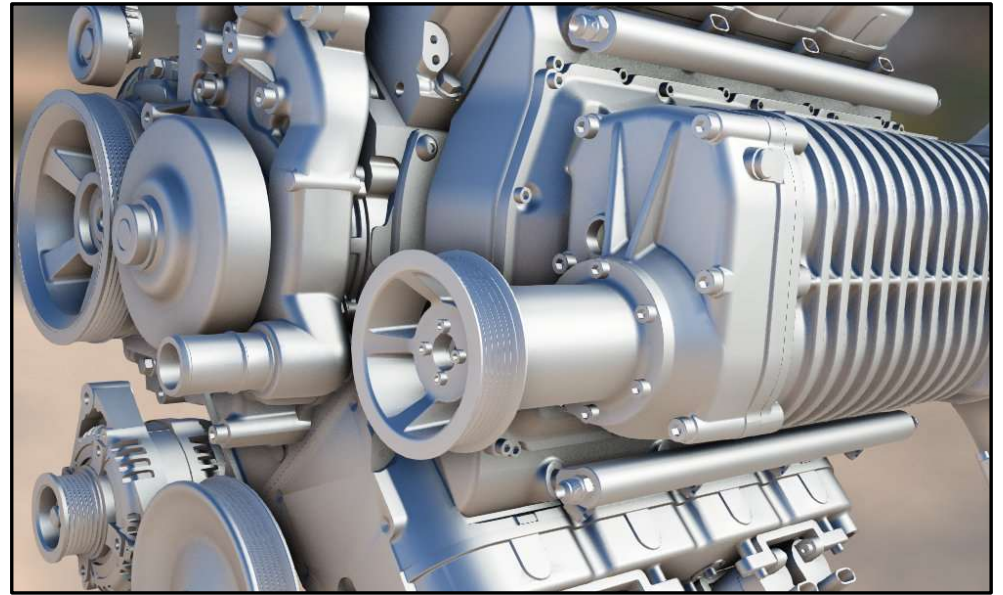
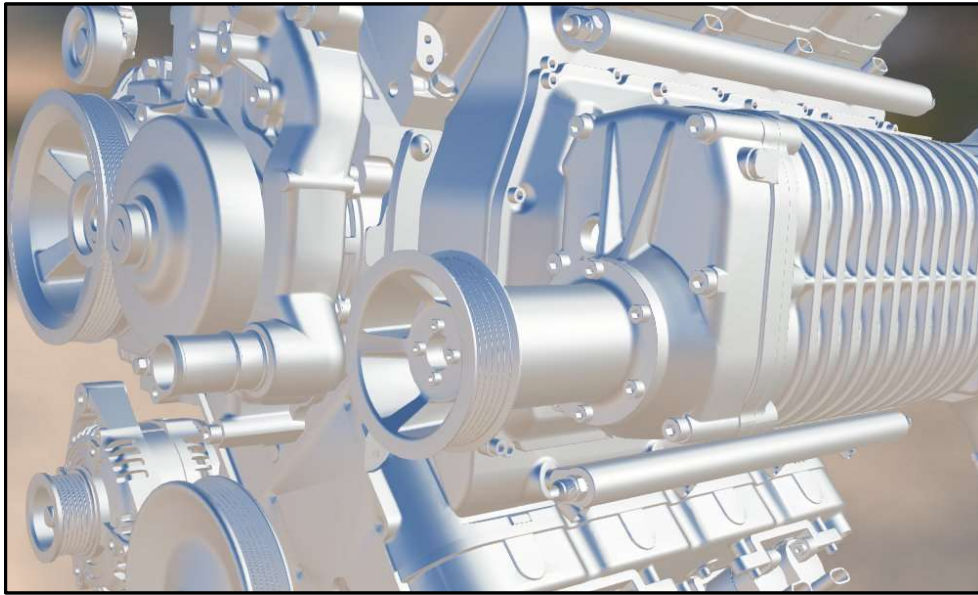


Image by Josiah Blaisdell





## An Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAOO)

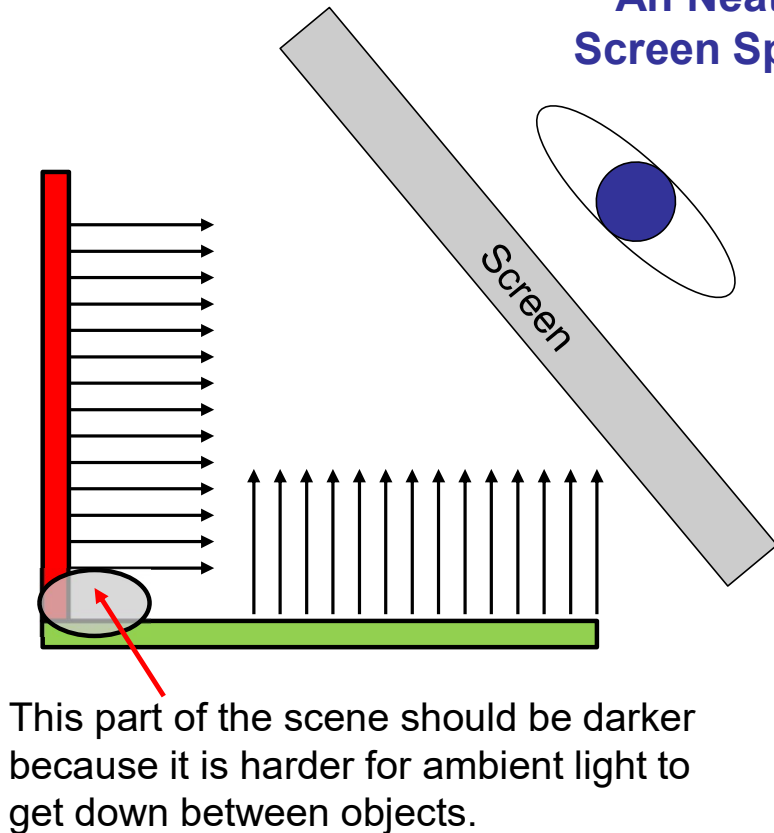


Kitware



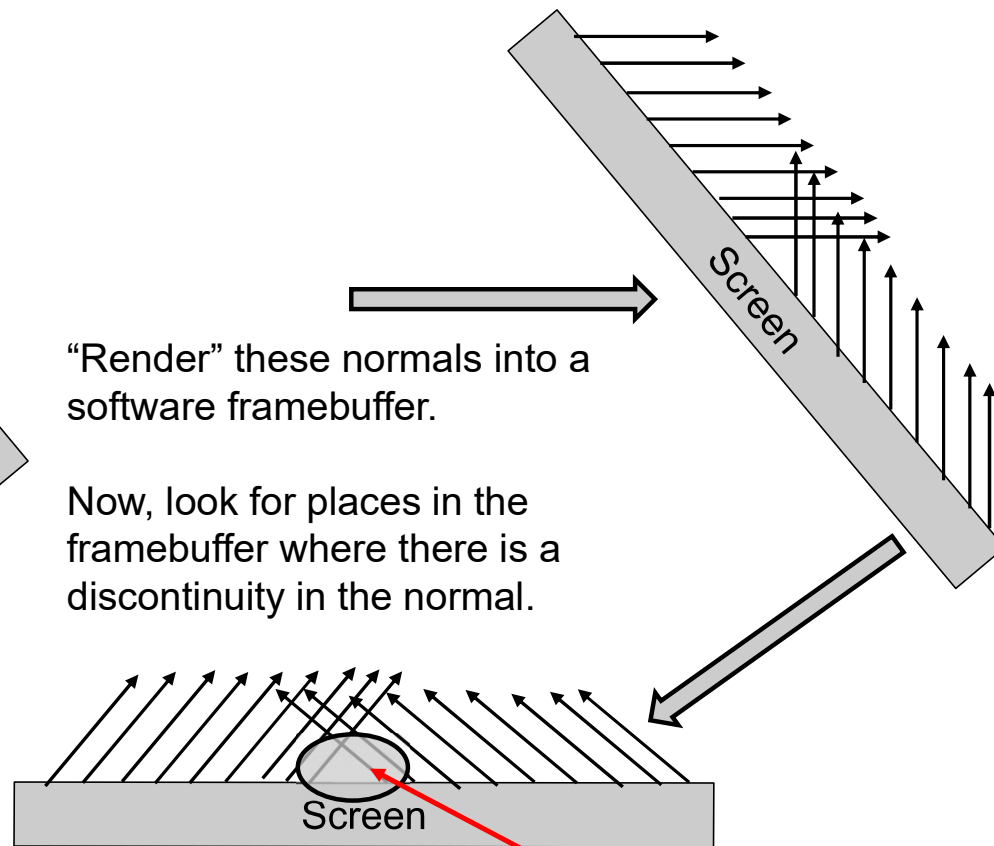
Oregon State  
University  
Computer Graphics

## An Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAO)



"Render" these normals into a software framebuffer.

Now, look for places in the framebuffer where there is a discontinuity in the normal.

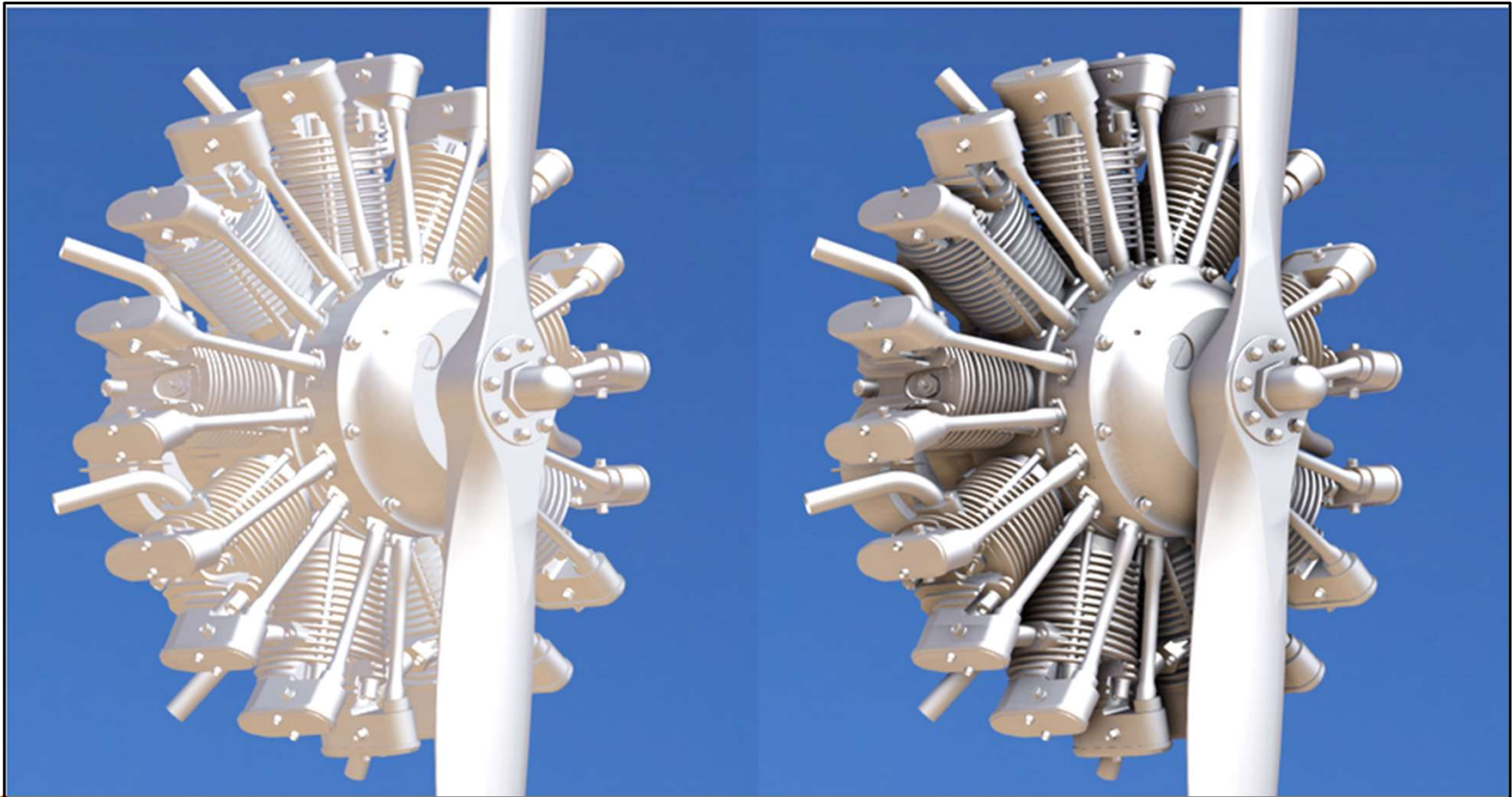




## A Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAO)

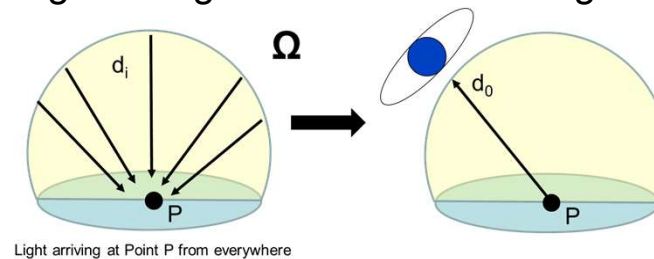


## A Neat Global Illumination-ish Trick: Screen Space Ambient Occlusion (SSAO)



## Something New: Neural Radiance Fields -- NeRFs

What if you want to know what an object looks like no matter where other light is coming from and no matter where you view it from? You could go through the whole rendering thing from every viewing angle...



...but that would be time consuming and would preclude any sort of real-time scene manipulation.

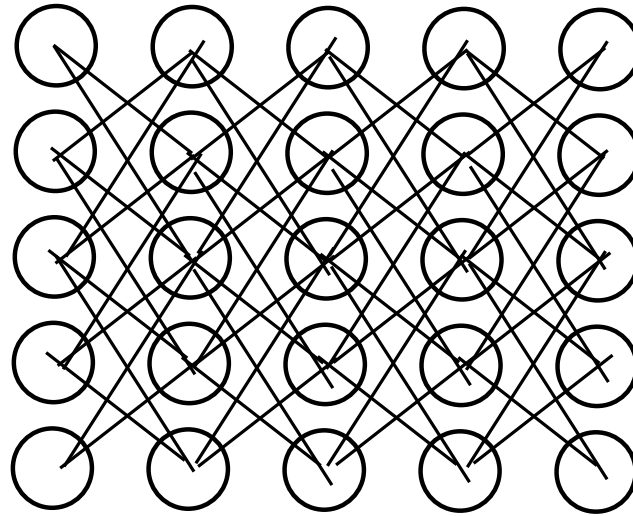
In the NeRFs technique, you precompute, for every incoming light direction how much of that ends up in every outgoing viewing direction. Then, when interacting with the scene, you don't need to do any actual "rendering". You just track the radiance outputs from an object and use those as inputs to other objects and use those precomputed values to see what comes out of that object.



How can we lookup that information quickly?

## NeRFs: Machine Learning to the Rescue!

For each object, you train a neural network ...



...on the pre-rendered data so that a radiance input to that object can quickly be turned into a radiance output from that object

*This is very new technique, but something worth keeping an eye on!*

