Rendering - I

CS475 / 675, Fall 2016

Siddhartha Chaudhuri

Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Andreas Suetterlin
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Thomas Suurland
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Henry Goss
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Sommerhaus PIU, Patrick Frey
(Take the challenge at http://area.autodesk.com/fakeorfoto)

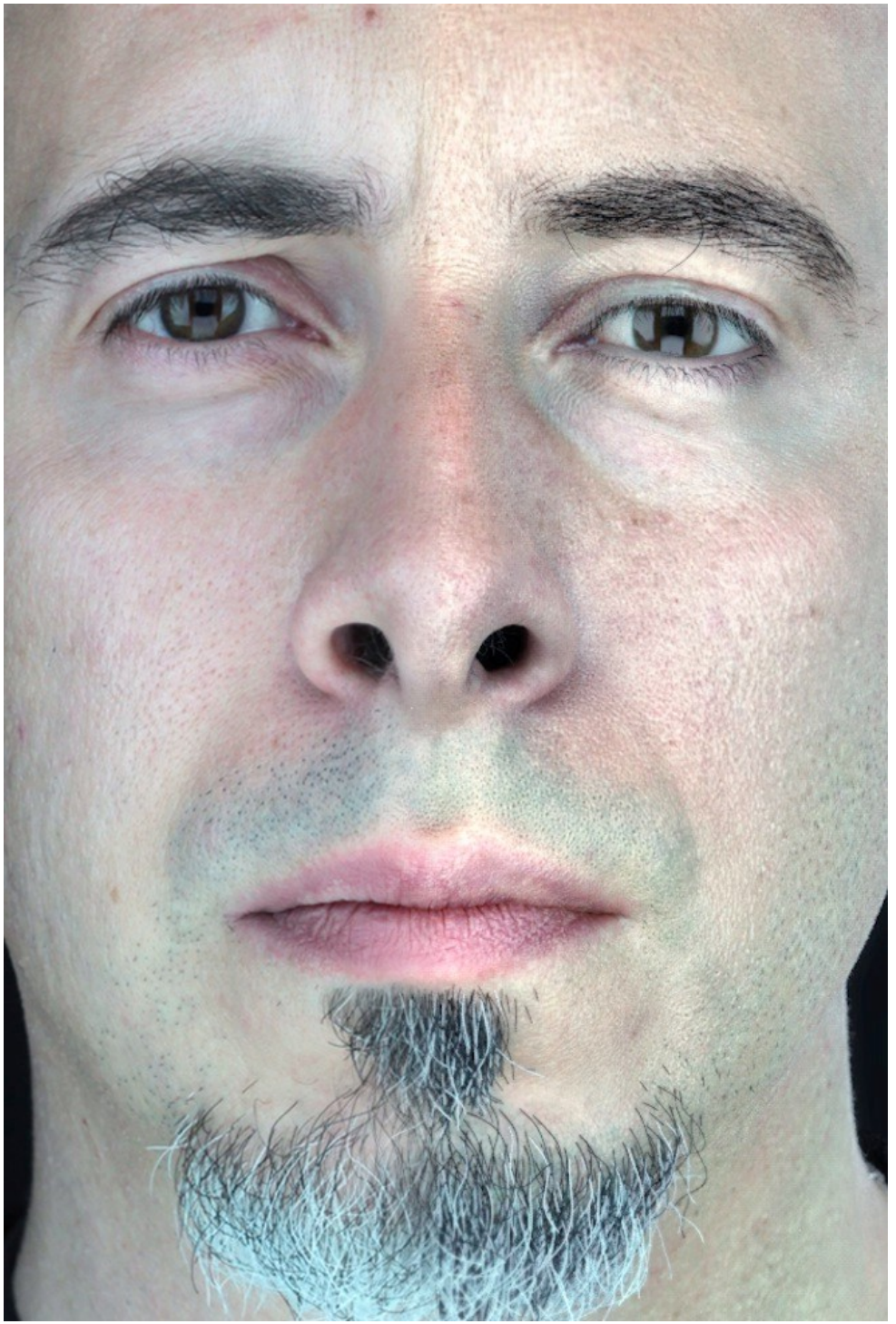
Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?



(Take the challenge at http://area.autodesk.com/fakeorfoto)

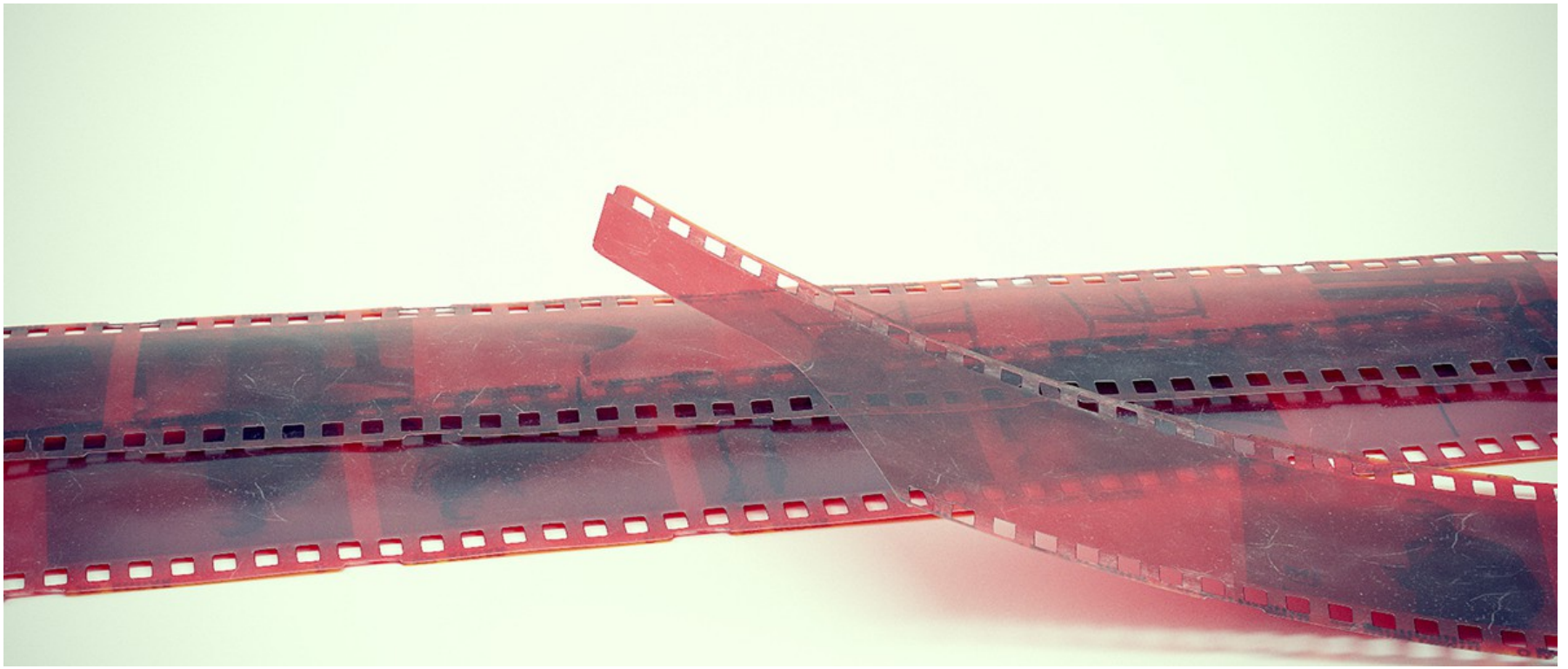
Fake or Foto?

Dan Roarty
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Dan Roarty
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?



(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

Matt Deakin
(Take the challenge at http://area.autodesk.com/fakeorfoto)

Fake or Foto?

(Michæl Paul Smith)

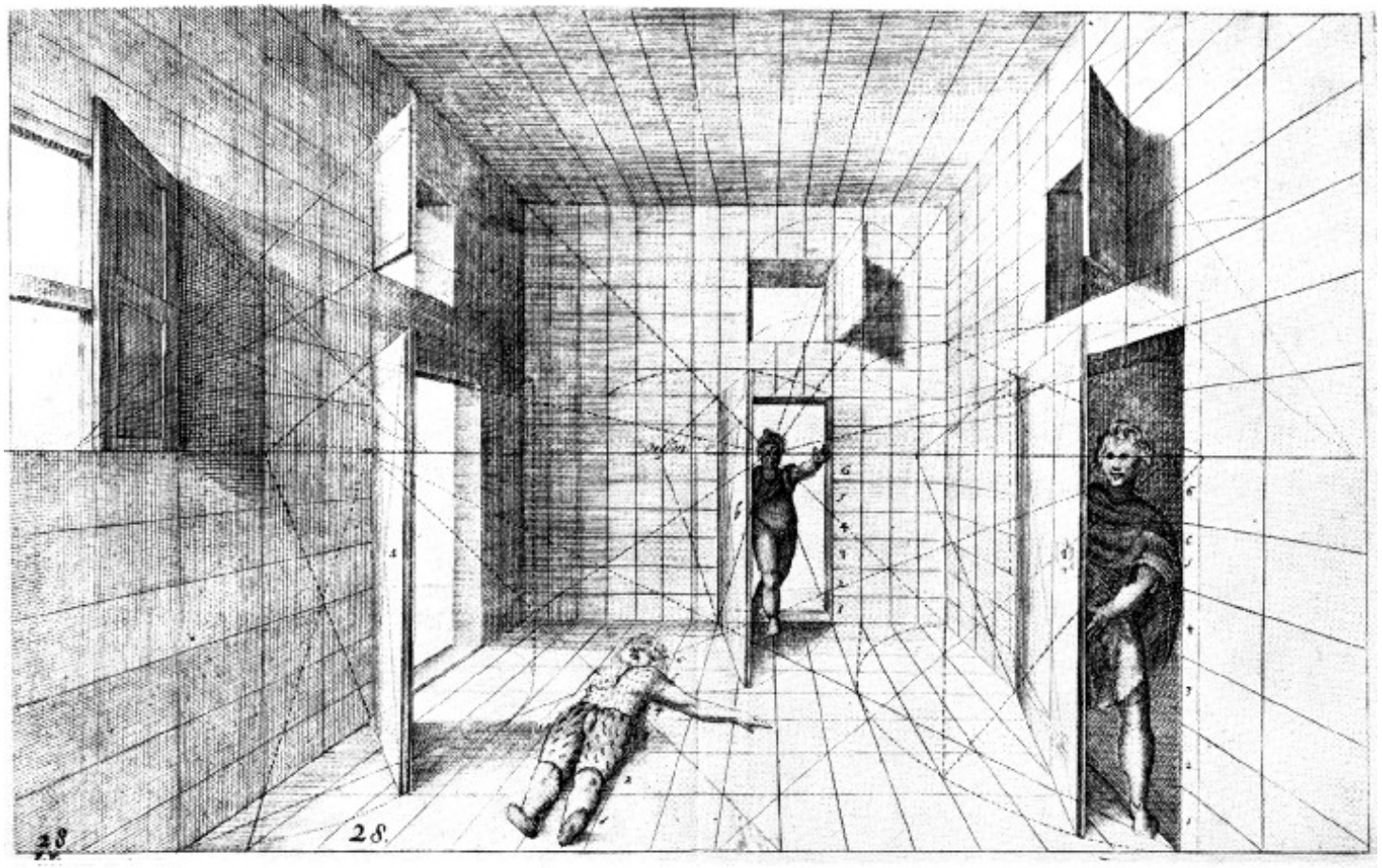
Uh...

(Michæl Paul Smith)

Outline

- Modeling a camera
- Tracking light
- Simulating surface appearance

Perspective

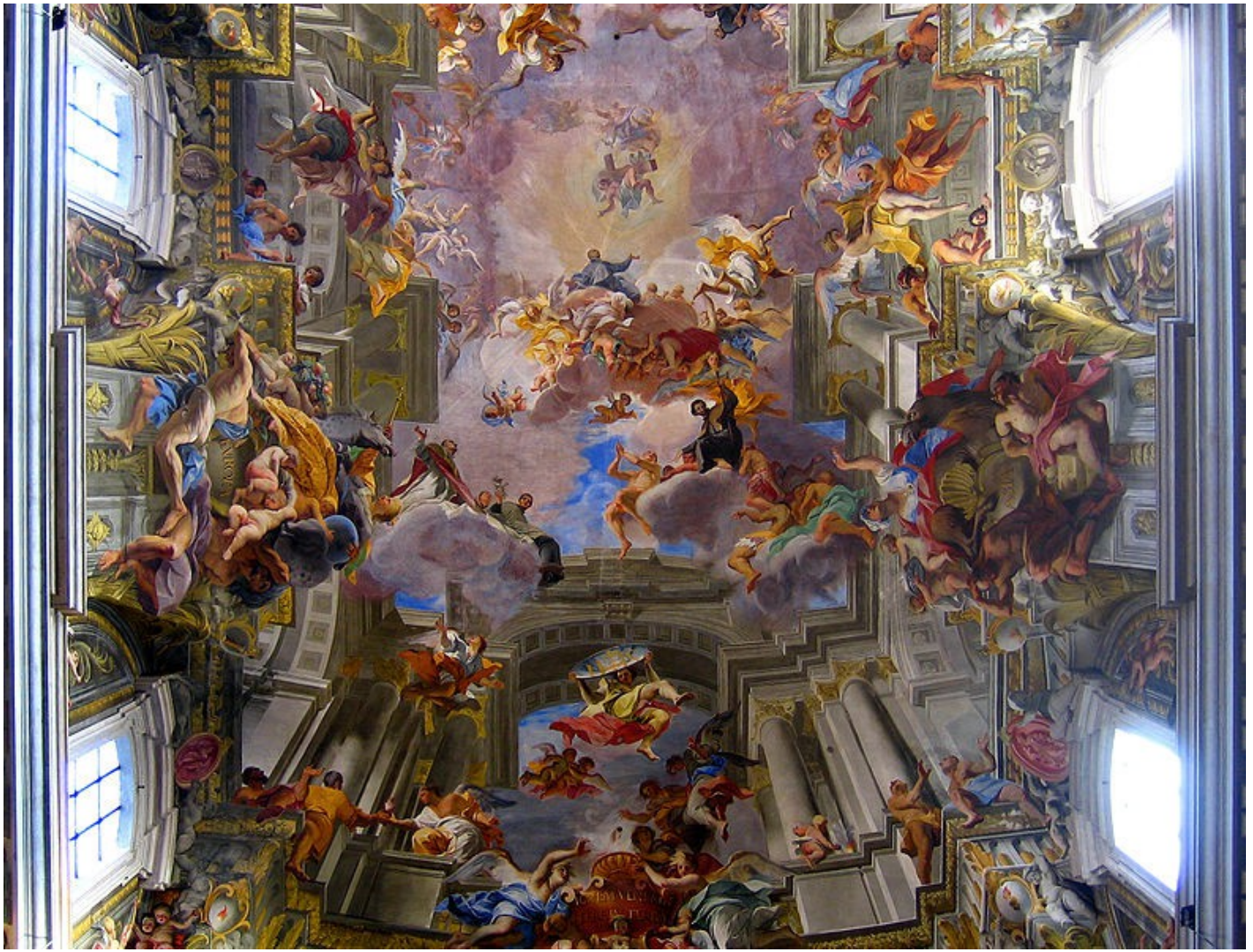


JAN VREDEMAN DE VRIES, *Perspective* (Leiden, 1604–5), plate 28. Courtesy, the Bancroft Library, Berkeley, California.

Creating the Illusion of Depth

Pietro Perugino, Sistine Chapel, Vatican City

Creating the Illusion of Depth



Andrea Pozzo, Chiesa di Sant'Ignazio, Rome

(Anthony Majanlahti)

Creating the Illusion of Depth

Andrea Pozzo, Jesuitenkirche, Vienna

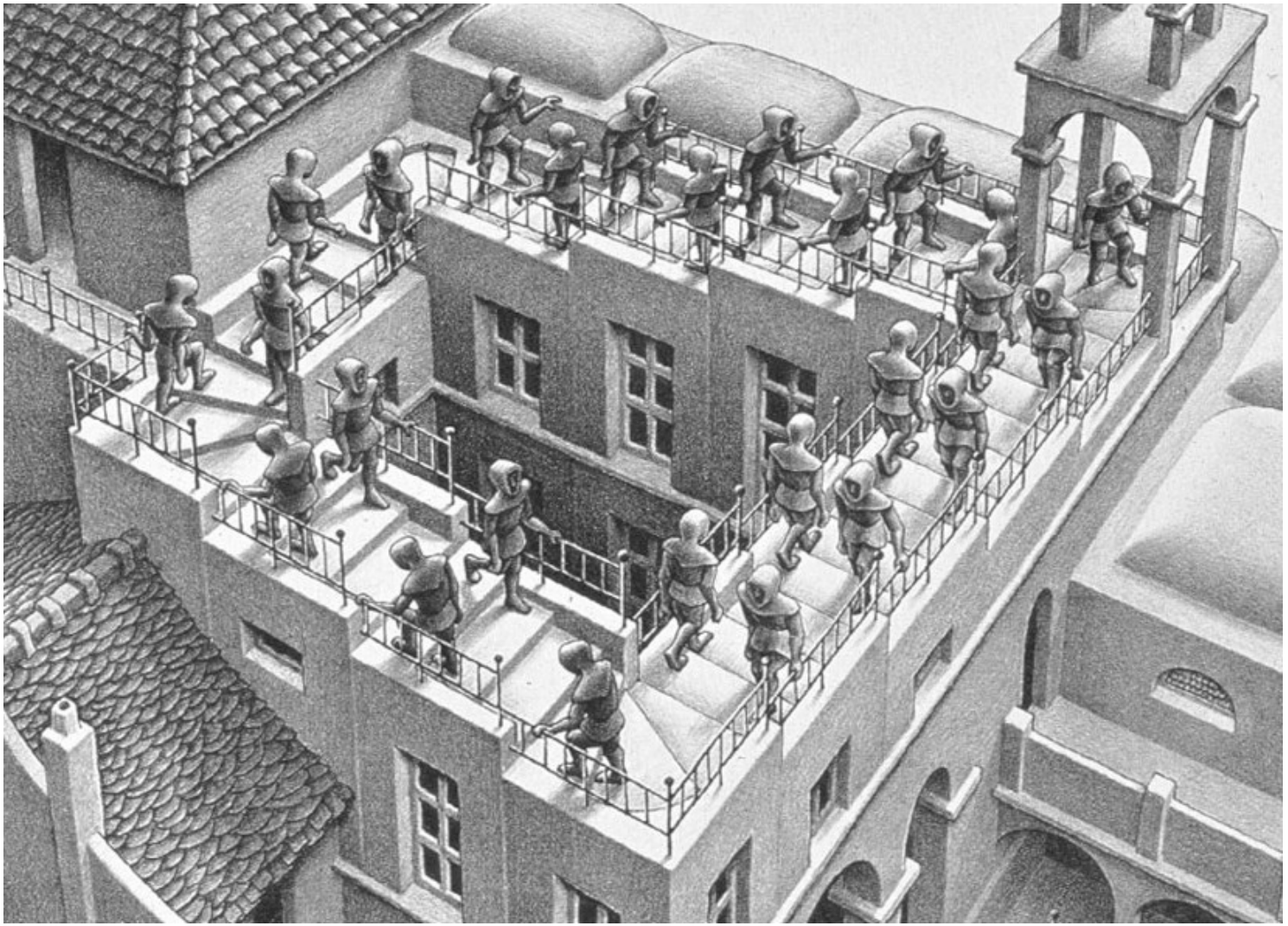
(Alberto Fernandez Fernandez)

Compensating for Perspective

Higher letters are bigger, to appear the same size from the ground (Taj Mahal, Agra)

(http://tylapens.blogspot.com)

Tricks with Perspective

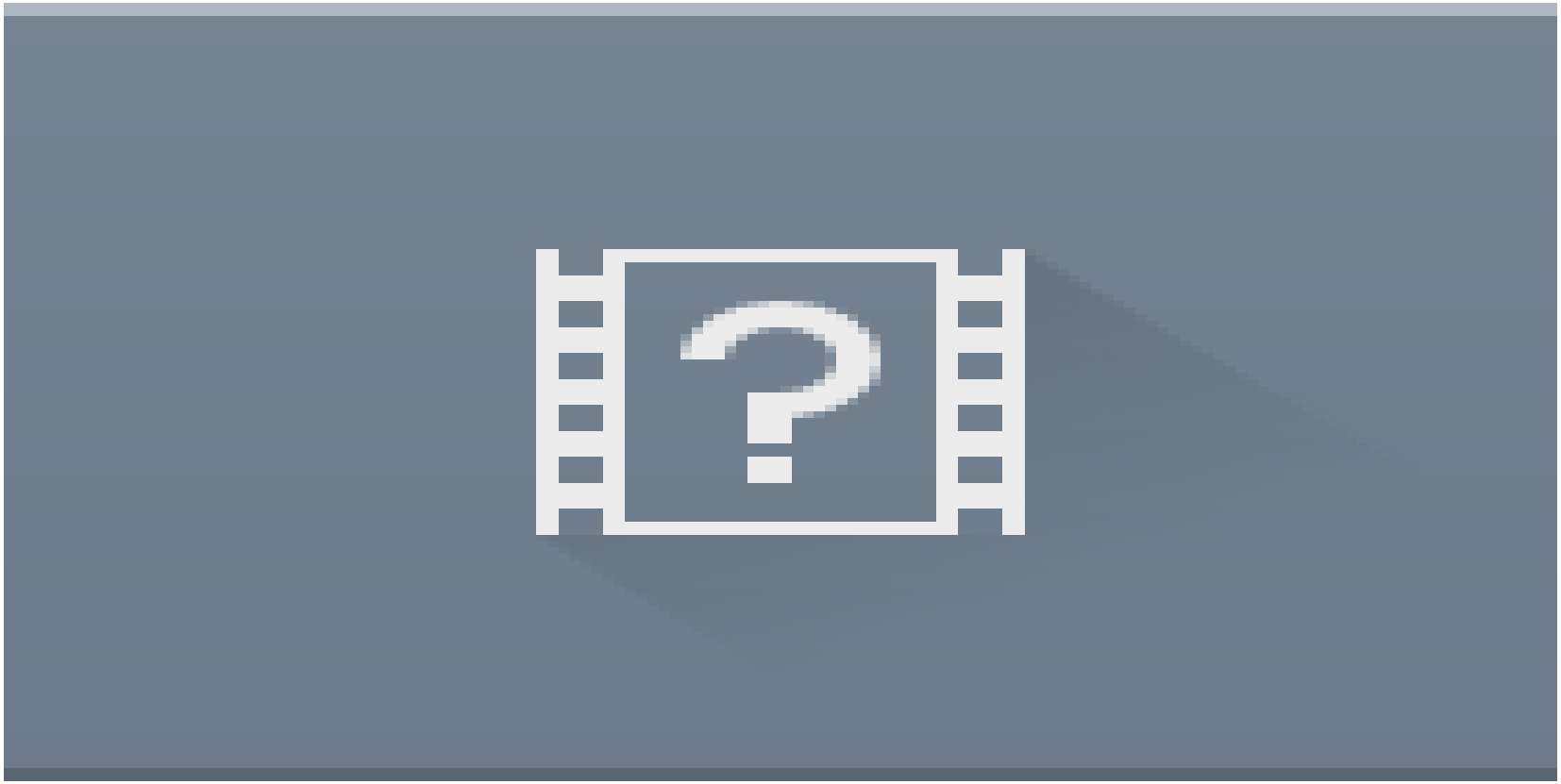


"Ascending and Descending", M. C. Escher

Tricks with Perspective

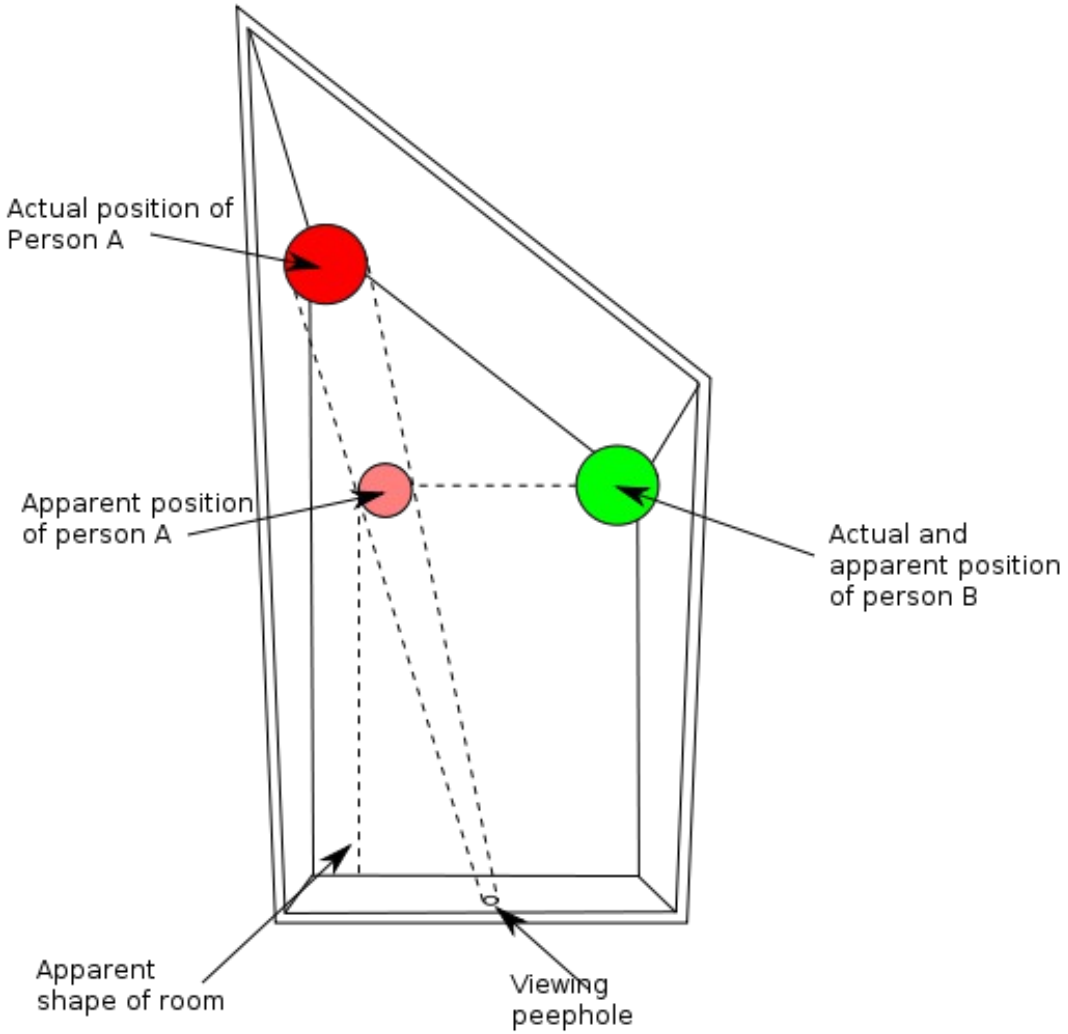
The Ames Room

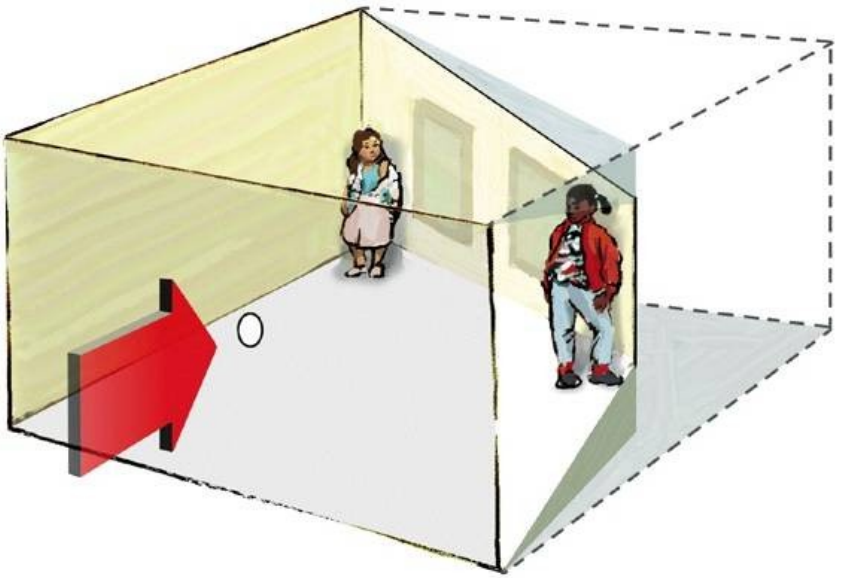
Tricks with Perspective



Ames Room, “The Mind's Eye”, BBC

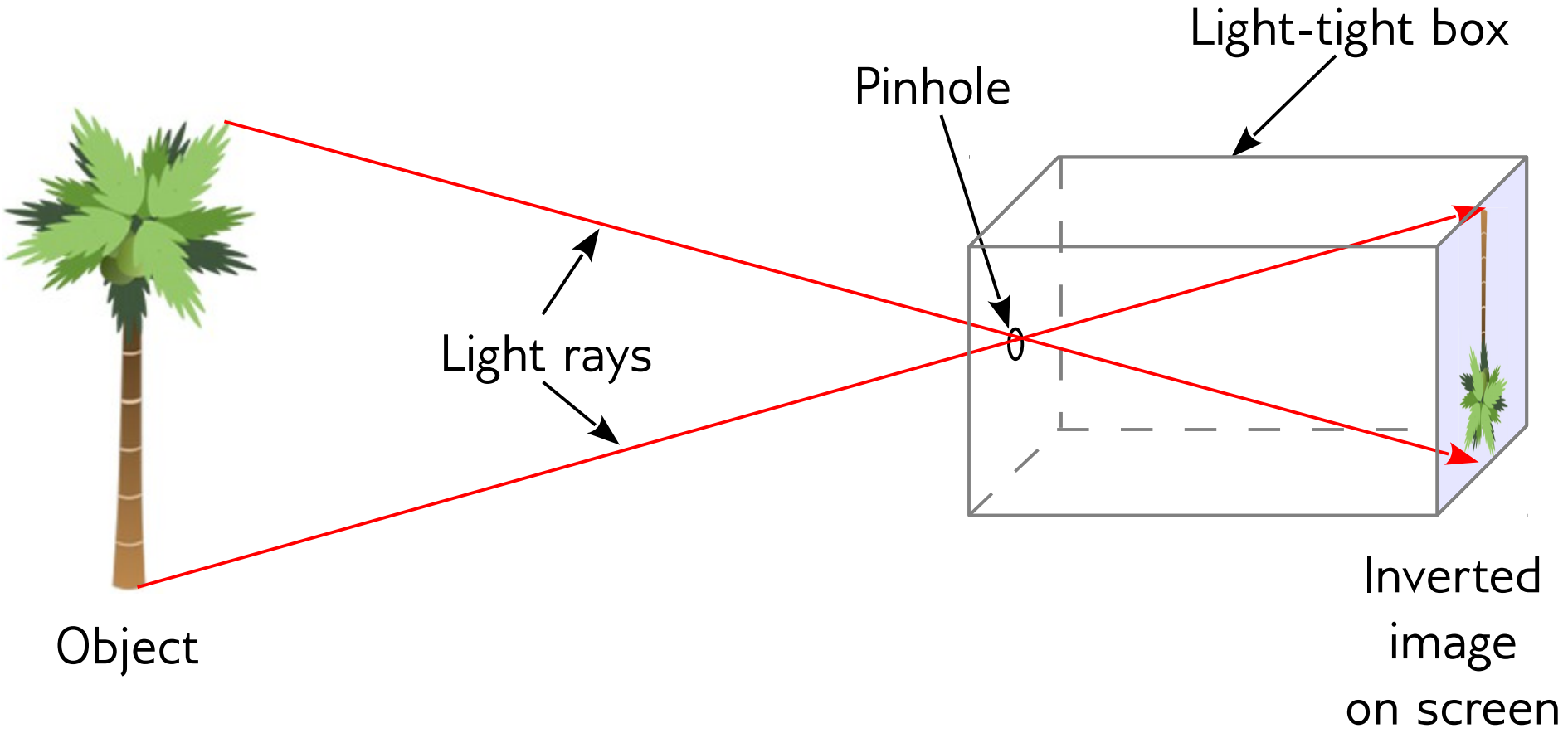
The Ames Room



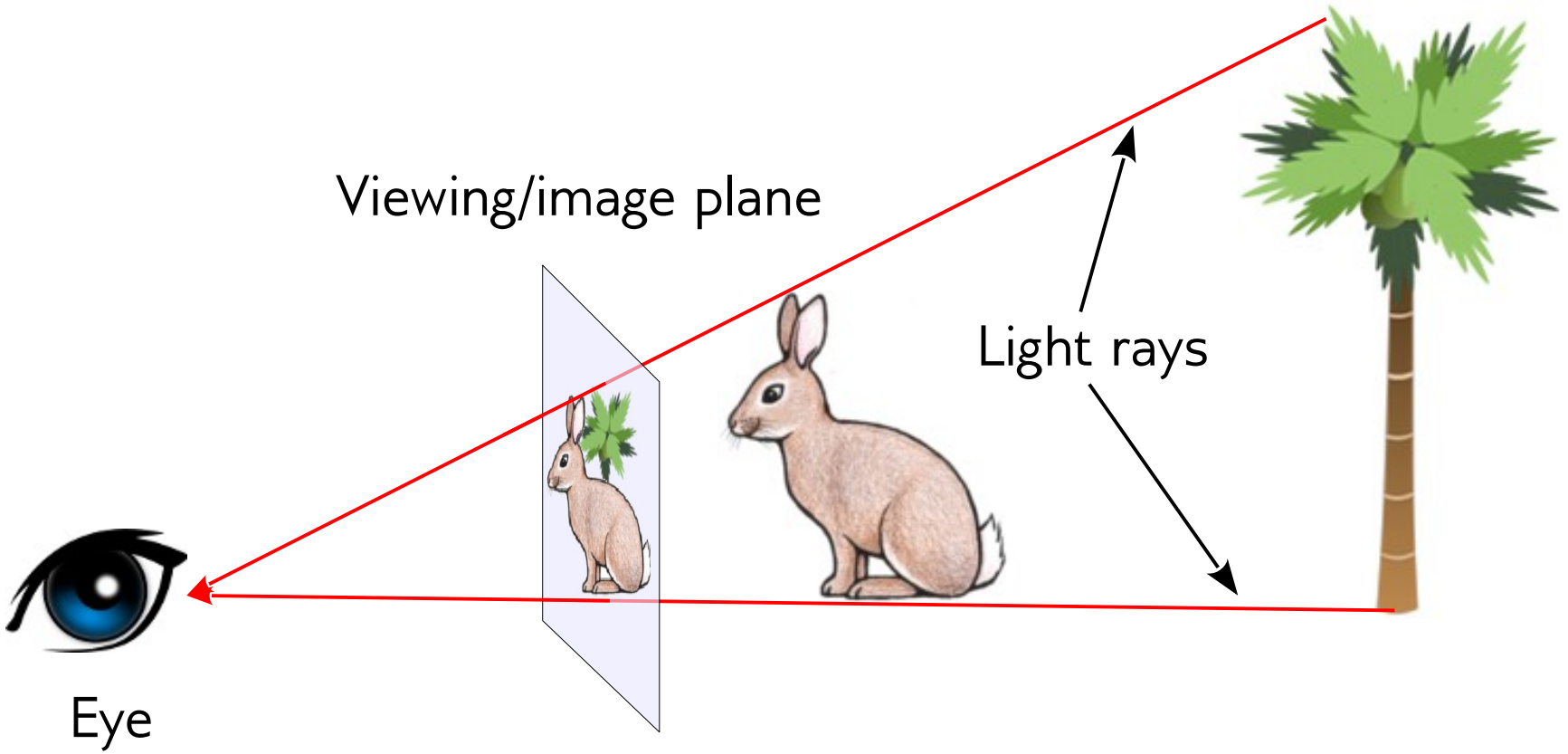


Wikipedia, Conway Psychology

Recall: Pinhole Camera Model

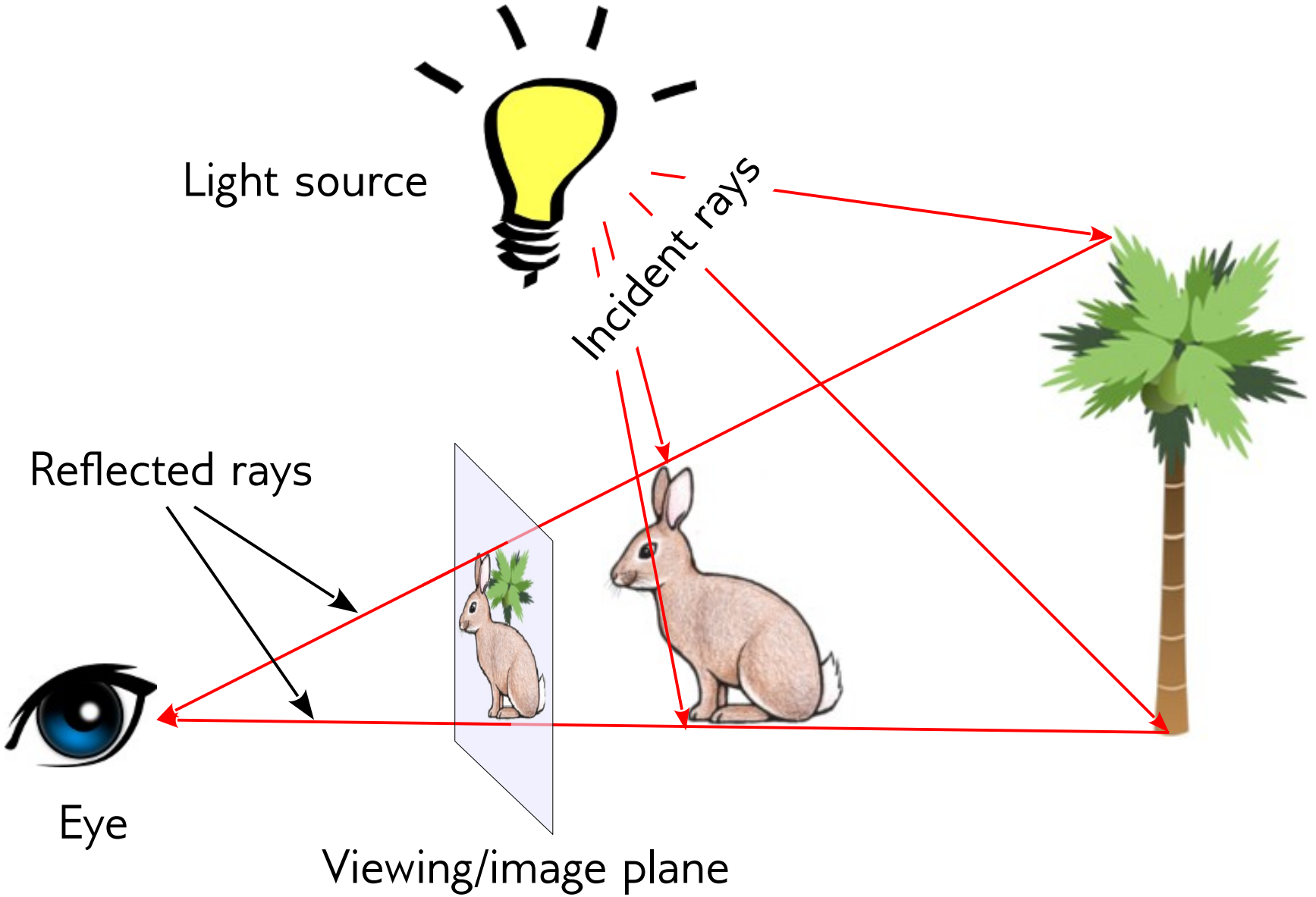


Perspective Camera for Graphics



Identical to pinhole camera except image plane is in front of eye, so image is not flipped

Where Does the Light Come From?



One Way to Render a Scene

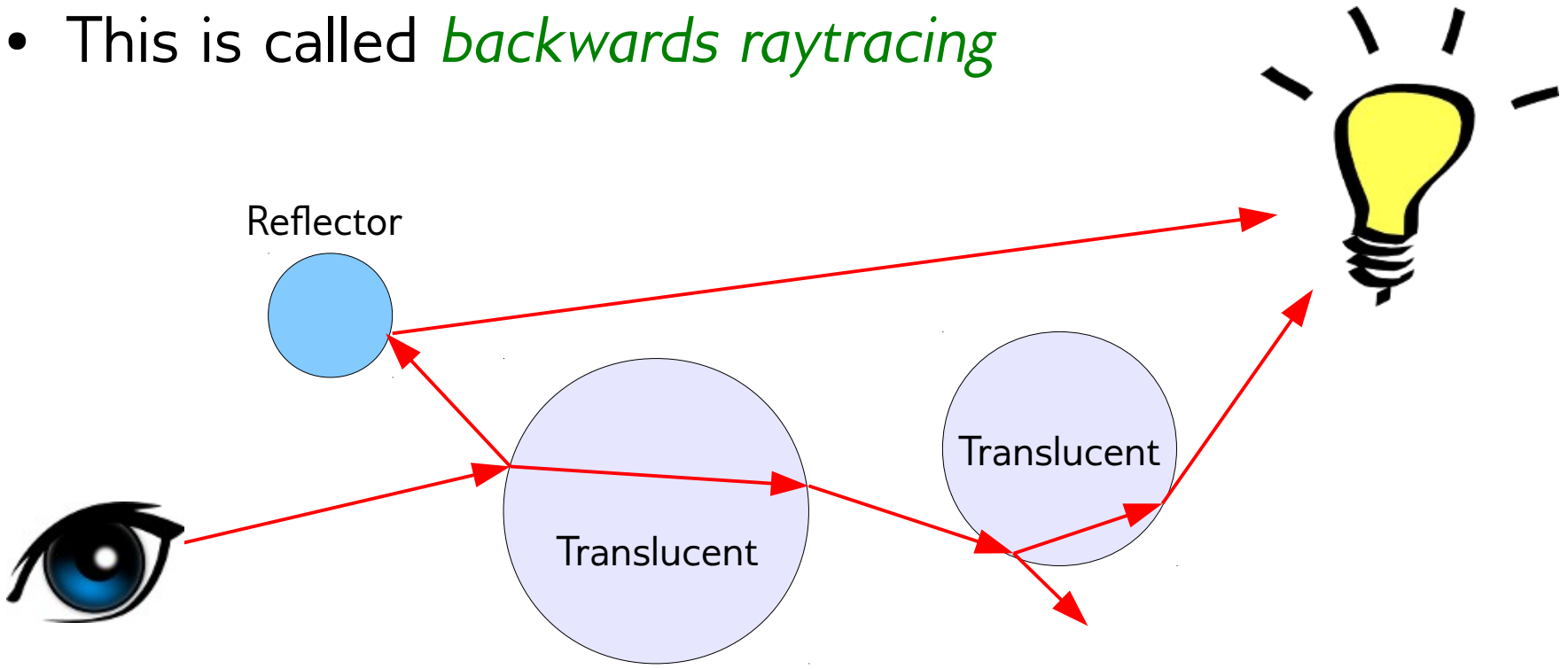
- Track rays of light as they leave the (virtual) light source, bounce around the (virtual) scene and finally enter the (virtual) eye
  - Very inefficient!
  - Most rays ***don't*** enter the eye
  - (This is called *forward raytracing*, btw)

Observation

Path of light is ***reversible***

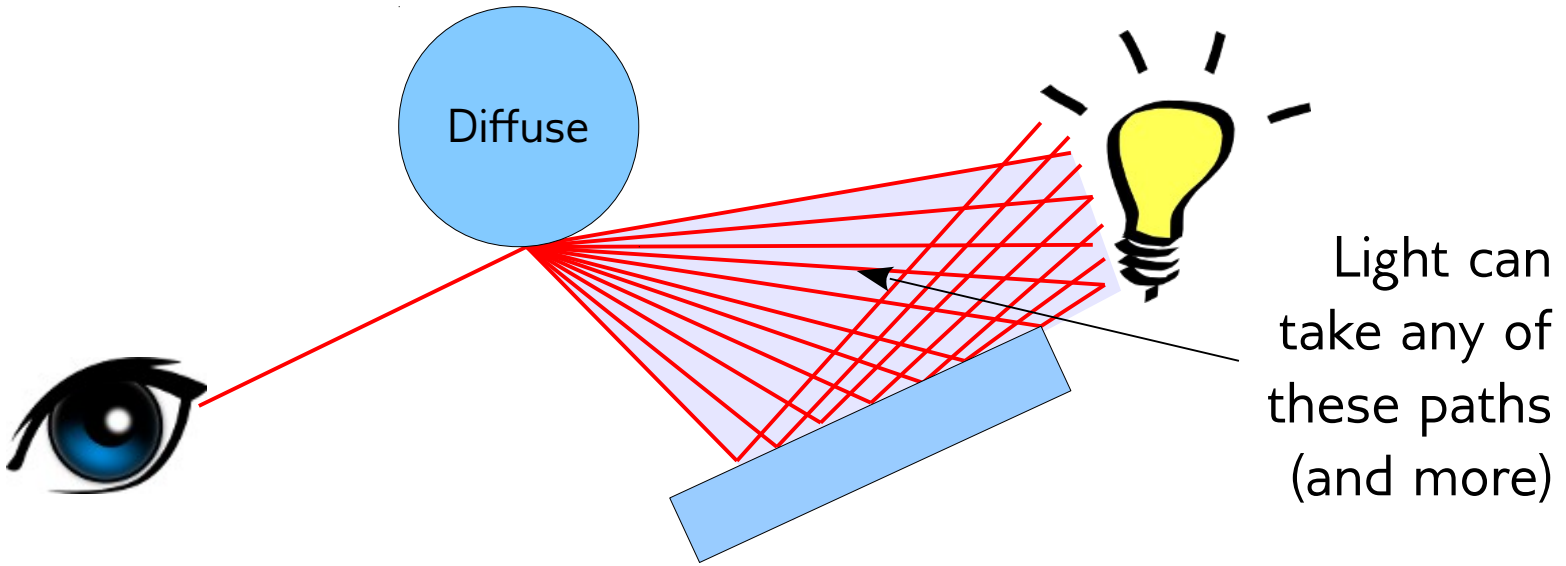
A Better Way

- Track light ***backwards*** from the eye to the light source
  - One path per image plane point
  - This is called *backwards raytracing*



Problem: Diffuse Surfaces

- If objects are not perfect reflectors, many incident rays contribute to a single reflected direction
  - So we have to go backwards along a whole range of directions after we hit the first object - inefficient!

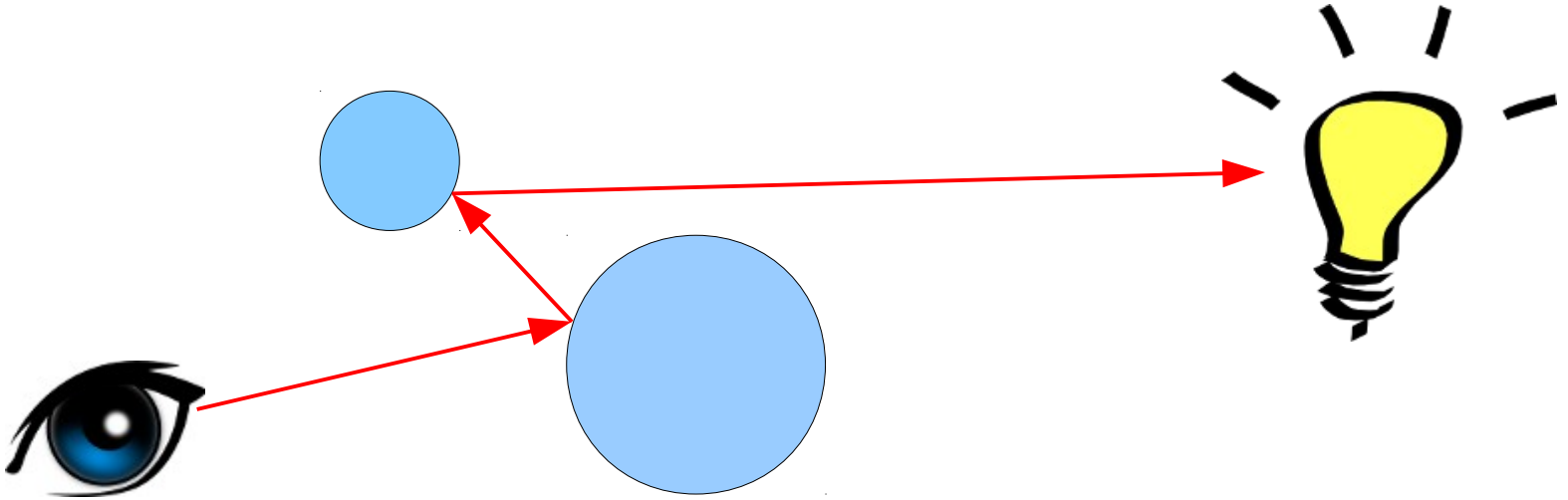


But there is hope...

- Most of the time, it's possible to approximate surface appearance without waiting for the backwards path to hit a light source
  - Stop after only a few bounces
- We'll decouple the light tracking
  - Assume every object behaves
    - partly as a perfect mirror
    - partly as a transparent medium (without scattering)
    - partly as a diffuse reflector

The Perfect Reflector Component

- Trace a single ray backwards, reflecting it whenever it intersects a surface
- Going forward, we lose light with each reflection
- Going backward, we ***accumulate*** light

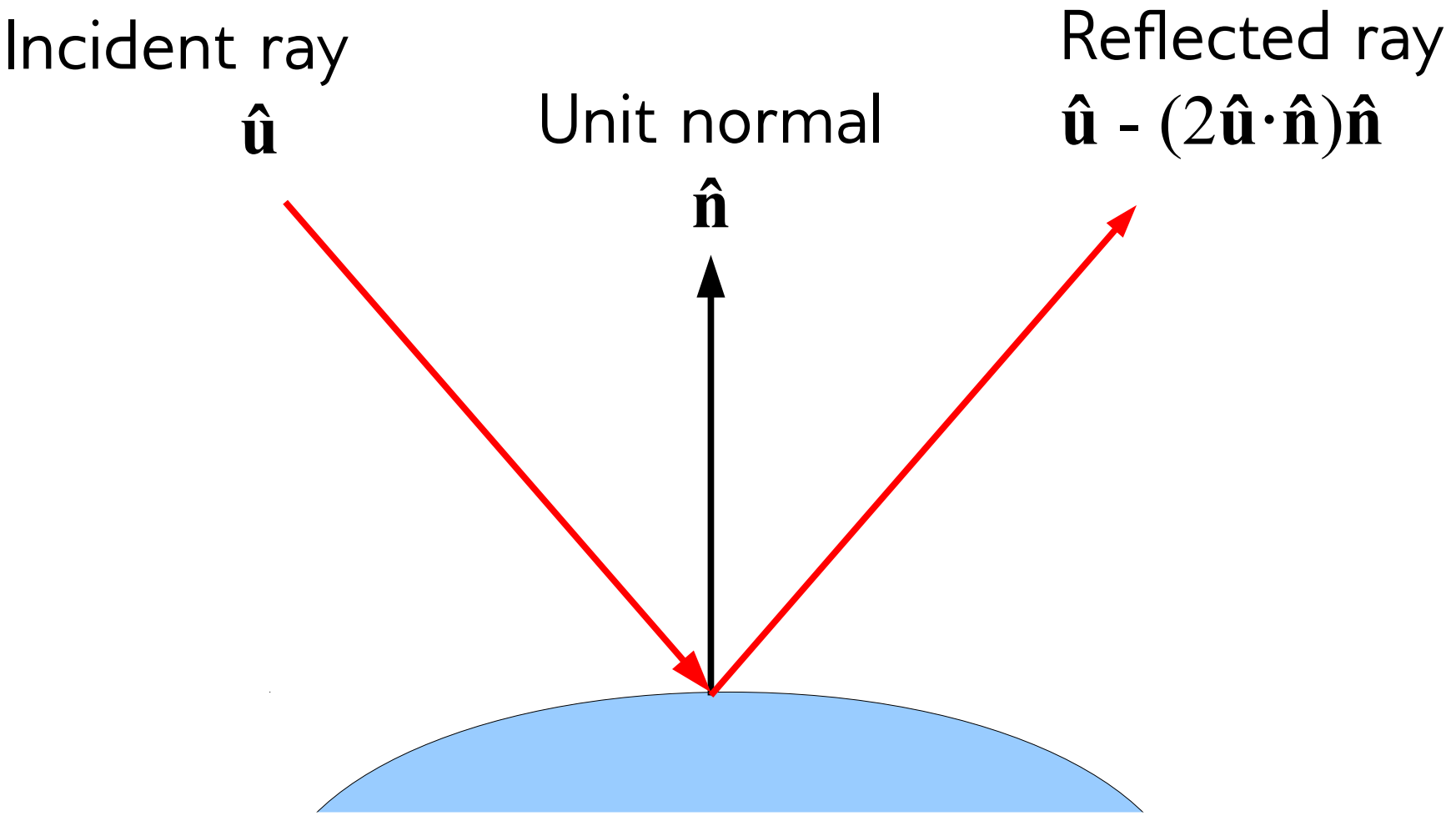


Recursive Raytracing Loop – ver. 1

```
function traceRay(ray) returns Color
    (obj, intersection) = getFirstIntersection(ray)
    if obj is a light source
        return getLightColor(obj)
    else
        reflected_ray = getReflectedRay(ray, intersection)
        reflectance = getSurfaceReflectance(obj, intersection)
        return reflectance * traceRay(reflected_ray)
    end if
end function
```

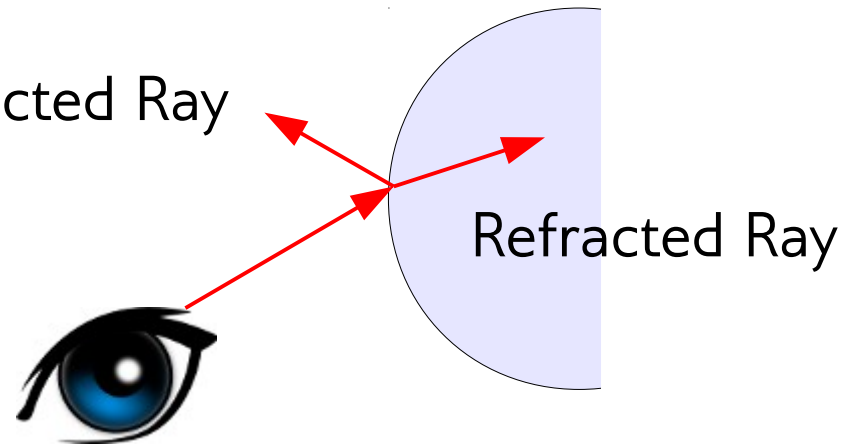
Note: Ray has origin and direction

Finding the Reflected Ray



The Transparent Component

- Whenever the backwards ray hits a surface, also trace a "*refracted*" ray through the object
  - Now we have two rays, reflected and refracted, for each backwards ray
  - Exponential growth in number of paths, so we can't trace back too far



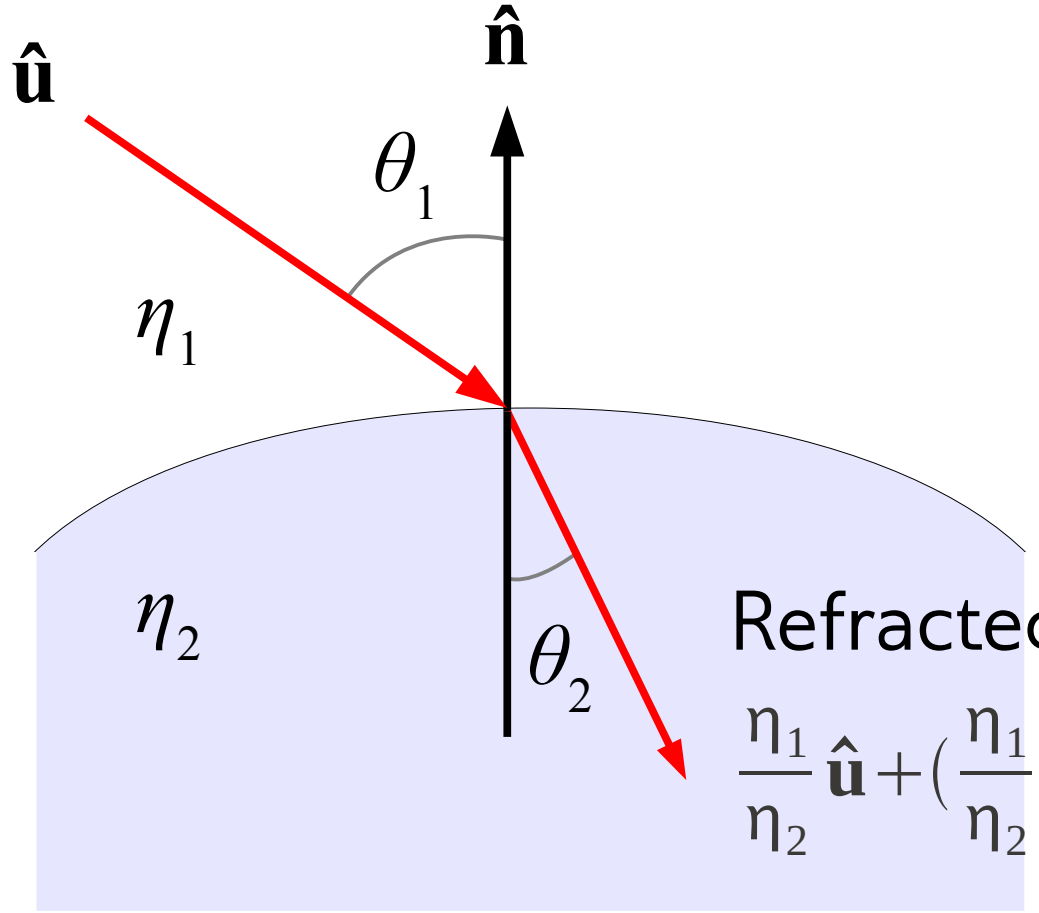
Recursive Raytracing Loop – ver. 2

```
function traceRay(ray) returns Color
    (obj, intersection) = getFirstIntersection(ray)
    if obj is a light source
        return getLightColor(obj)
    else
        reflected_ray = getReflectedRay(ray, intersection)
        reflectance = getSurfaceReflectance(obj, intersection)

        refracted_ray = getRefractedRay(ray, intersection)
        transmittance = getSurfaceTransmittance(obj, intersection)

        return   reflectance * traceRay(reflected_ray)
               + transmittance * traceRay(refracted_ray)
    end if
end function
```

Finding the Refracted Ray (Snell's Law)



$$\cos\theta_1=-\hat{\mathbf{u}}\cdot\hat{\mathbf{n}},\quad \cos\theta_2=\sqrt{1-\sin^2\theta_2},\quad \sin^2\theta_2=\left(\frac{\eta_1}{\eta_2}\right)^2(1-\cos^2\theta_1)$$

The Diffuse Component

- Commonly broken down [Phong 1973] into:
  - *Lambertian*
    - Dœsn't depend on viewing direction

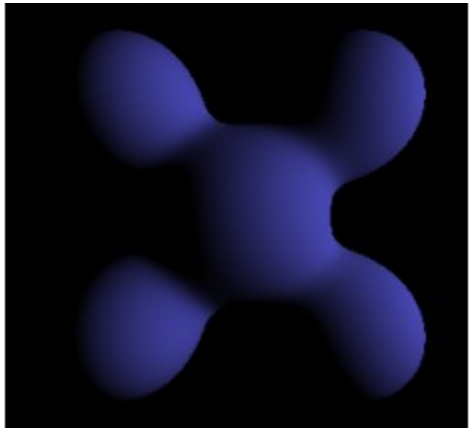
  - Approximately *Specular* (shiny)
    - Depends on viewing direction

  - *Ambient*: Light that has bounced around so much it uniformly lights the scene even in the darkest corners
    - prevents shadows from appearing completely black

Putting It All Together

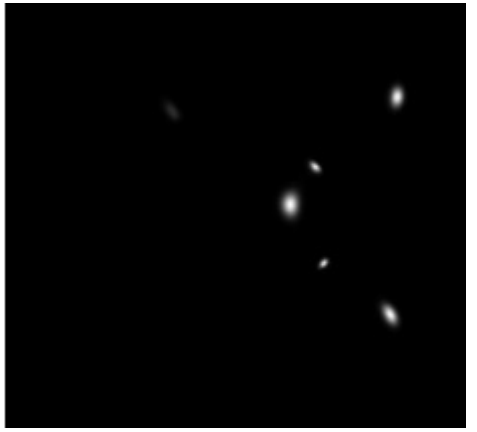
Ambient

+



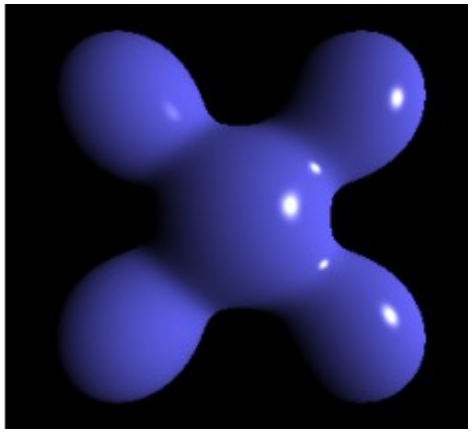
Lambertian

+



(Approx.) Specular

=



(Wikipedia)

Caution!

- The Lambertian component is often called the "diffuse" component when the context is clear
- Hence this shading model is often called Ambient-Diffuse-Specular (ADS)
  - … although "diffuse" here means Lambertian
  - … and "specular" means *approximately* specular
    - (a perfectly specular object is a mirror)

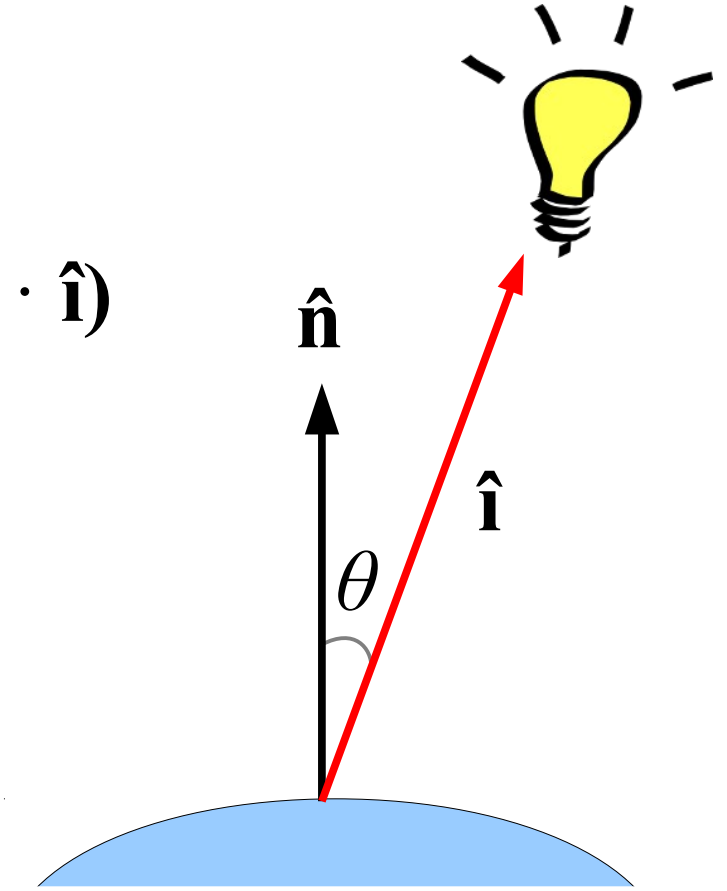
Lambertian Shading

- Intuition: Slanted illumination ⇒ dimmer, all directions receive same reflected light
- Modeled via falloff function

$$I_{\mathrm{L}} = I_{\mathrm{light}} k_{\mathrm{L}} \cos\theta = I_{\mathrm{light}} k_{\mathrm{L}} (\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})$$

  where $\hat{\mathbf{l}}$ is unit vector *from* point *to* light source

- Independent of viewing direction



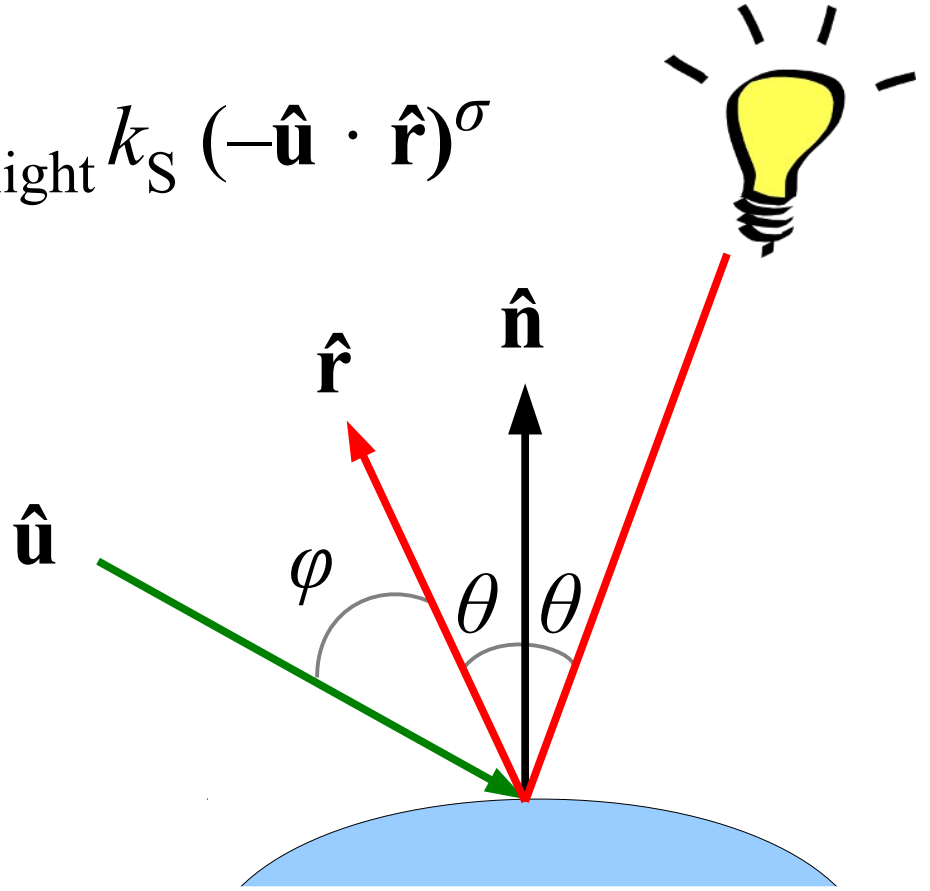
(Approximately) Specular Shading

- Intuition: More light is reflected along directions close to the "perfect" reflected ray

$$I_S = I_{light} k_S (\cos \varphi)^\sigma = I_{light} k_S (-\hat{\mathbf{u}} \cdot \hat{\mathbf{r}})^\sigma$$

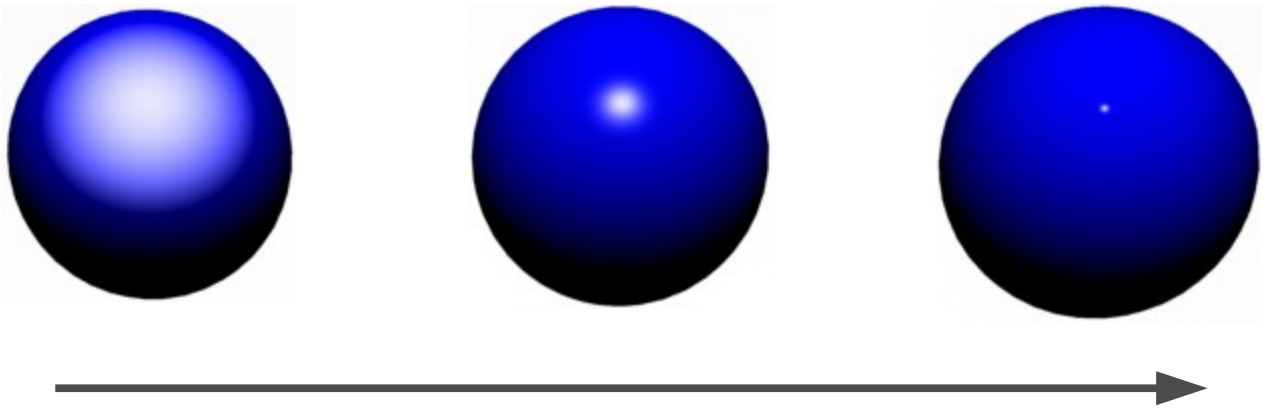
where $\hat{\mathbf{r}}$ is ideal reflection direction

- Depends on viewing direction
- (aka "Phong highlights")



(Approximately) Specular Shading

- The specular exponent σ controls the size of the highlight



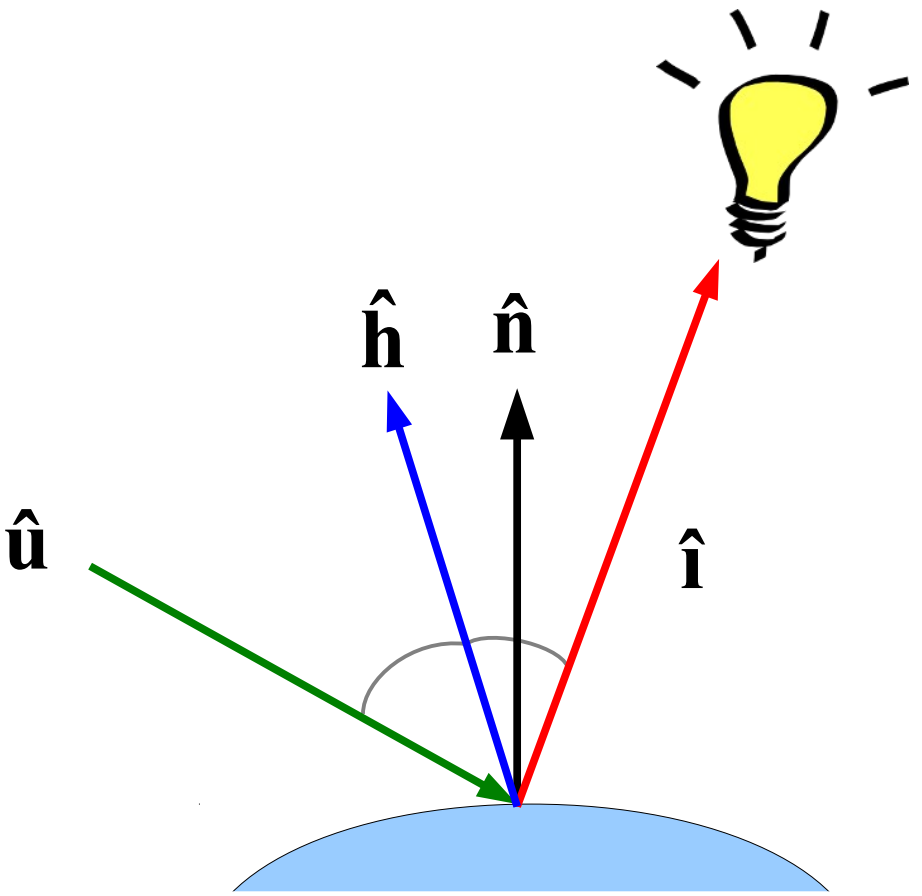
Increasing σ

Simplification: The Blinn-Phong Model

- Replace $-\hat{\mathbf{u}} \cdot \hat{\mathbf{r}}$ with $\hat{\mathbf{n}} \cdot \hat{\mathbf{h}}$
- $\hat{\mathbf{h}}$ is the "halfway vector"

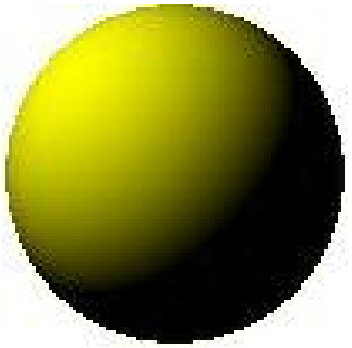
$$\frac{\hat{\mathbf{i}} - \hat{\mathbf{u}}}{\|\hat{\mathbf{i}} - \hat{\mathbf{u}}\|}$$

- Looks very like the original Phong highlight
- No need to compute $\hat{\mathbf{r}}$

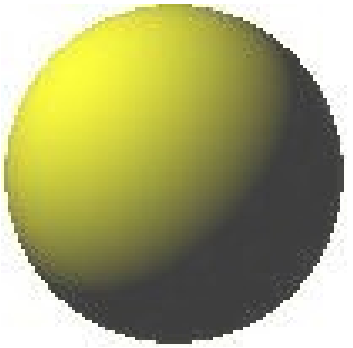


Ambient Shading

- Constant value $k_{\mathrm{A}} I_{\mathrm{ambient}}$ added to reflected light
- Uniform over an object, not directly defined in terms of light sources



Low ambient term

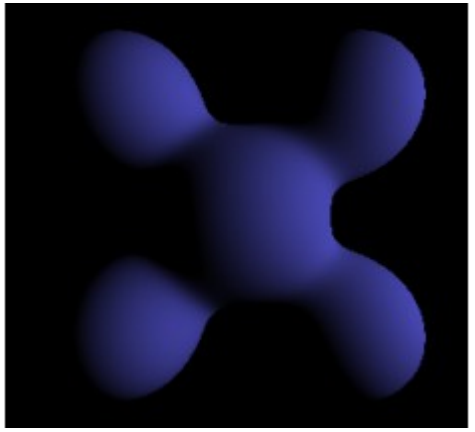


High ambient term

Let's look at it all again...

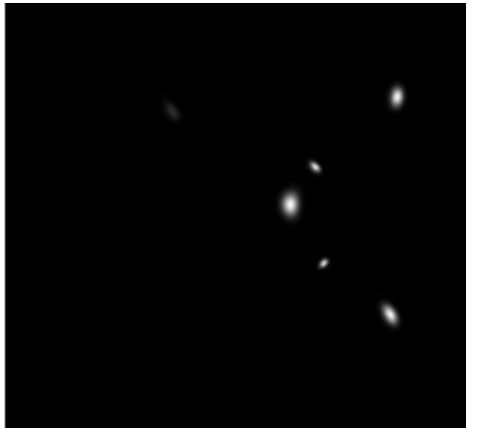
Ambient

+



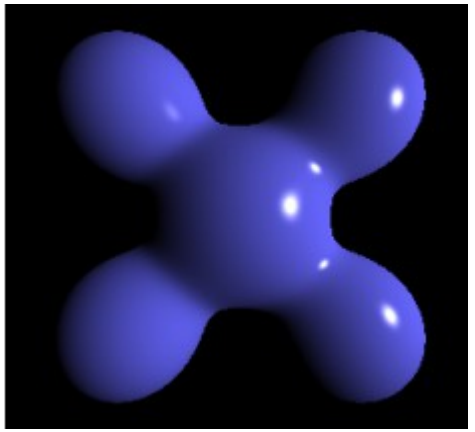
Lambertian

+



(Approx.) Specular

=



(Wikipedia)

Material

- An object's “material” can be described with its ambient, lambertian and specular cœfficients

$$(k_{\mathrm{A}}, k_{\mathrm{L}}, k_{\mathrm{S}}, \sigma)$$

- In a color space, k_{A}, k_{L} and k_{S} are defined separately for each primary
  - e.g. $k_{\mathrm{L}} = (k_{\mathrm{L}}{}^{\mathrm{red}}, k_{\mathrm{L}}{}^{\mathrm{green}}, k_{\mathrm{L}}{}^{\mathrm{blue}})$

Only an Approximation!

- The Ambient-Lambertian-Specular ("Phong") reflection model is an empirical model that approximates real world materials
- The *bidirectional reflectance distribution function* (BRDF) tells us how light is actually reflected in various directions
  - It can be measured in a real-world setup using a *goniophotometer*
- Sophisticated renderers use BRDFs

(STIL S.A.)

Extending BRDFs

Without
subsurface scattering

With
subsurface scattering

Jensen, Marschner, Levoy and Hanrahan, 2001

Recursive Raytracing Loop – ver. 3

```
function traceRay(ray) returns Color
    (obj, intersection) = getFirstIntersection(ray)
    if obj is a light source
        return getLightColor(obj)
    else
        diffuse_color = getDiffuseShading(obj, intersection)  // ADS/BRDF/radiosity/...

        reflected_ray = getReflectedRay(ray, intersection)
        reflectance = getSurfaceReflectance(obj, intersection)

        refracted_ray = getRefractedRay(ray, intersection)
        transmittance = getSurfaceTransmittance(obj, intersection)

        return combine(    diffuse_color,
                           reflectance * traceRay(reflected_ray),
                           transmittance * traceRay(refracted_ray)    )
    end if
end function
```

Raytraced Image

Rendered in POV-Ray by Gilles Tran

Computing Intersections

- We must evaluate which object is hit first by a ray, and where
- Usually:
  - Represent surface by equation
  - Substitute formula of ray
  - Solve for intersection point

Let's see an example...

- Sphere has equation $\|\mathbf{p} - \mathbf{c}\|^2 = r^2$
  - $\mathbf{c}$ is center and r is radius
- A point on the ray can be written as $\mathbf{p}_0 + t\mathbf{u}$
  - $\mathbf{u}$ is direction and $\mathbf{p}_0$ is origin of ray
- Substituting: $\|\mathbf{p}_0 + t\mathbf{u} - \mathbf{c}\|^2 = r^2$
- This is a quadratic equation in t (expand it!) which can be solved for t
  - **Note:** The parametric form of the ray implies that the closest surface is the one with the smallest positive t

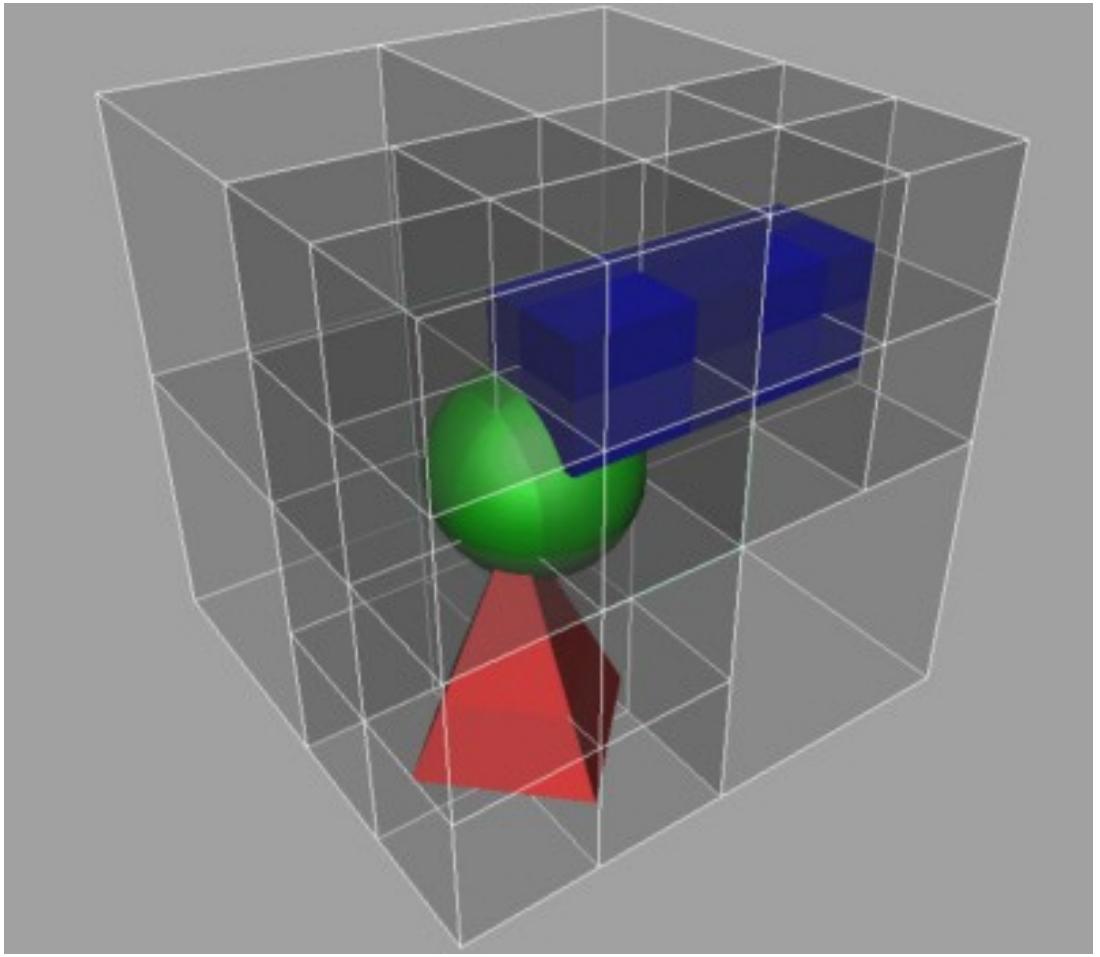
Efficiency Concerns

- Expensive to test if ray intersects every object in the scene
- Some speedup with *bounding volumes*
  - Simple object (e.g. sphere/cube) enclosing a complex shape
  - Test for intersection with bounding volume first
  - If ray doesn't hit bounding volume, it doesn't hit object either

Efficiency Concerns

- Huge speedup with *space subdivision* methods
  - Hierarchy of bounding volumes
  - Octree
  - kD-Tree
  - ...
- General idea:
  - Hierarchically partition space (root encloses all objects)
  - Traverse hierarchy top-down
    - Consider a subtree only if its root cell is intersected

Example: Octree



Shadows

- When computing diffuse shading, we assumed the surface was directly visible from the light source
- If some other object blocks the path, we must ignore this component
- To test this:
  - Create a ray from the surface point to the light source
  - Check if the ray hits anything before the light
  - Can reuse intersection code!

Fun Things to Try

- Learn POV-Ray: http://www.povray.org
  - Open-source raytracer
  - Set up your scene and lighting with a script
  - See their Hall of Fame! http://hof.povray.org
- Find out more about "*trompe-l'œil*"
- Think about the different ways in which the lighting model we described falls short of reality