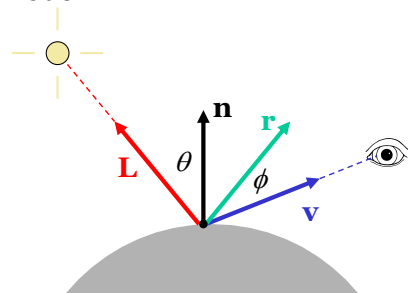


Looking at the Shading Model

Recall our current Phong illumination model

$$I = \sum_{\text{all lights } L} \overbrace{I_L k_d (\mathbf{n} \cdot \mathbf{L})}^{\text{diffuse reflection}} + \underbrace{I_L k_s (\mathbf{r} \cdot \mathbf{v})^n}_{\text{specular highlight}}$$



- sums contributions from all lights
- plus the global ambient glow

We'll add three important new components

- shadows, specular reflections, and specular transmission
- our general strategy: recursively trace rays to evaluate shading
- hence we call the method (recursive) ray tracing

Refractive Transparency

OpenGL supports limited transparency

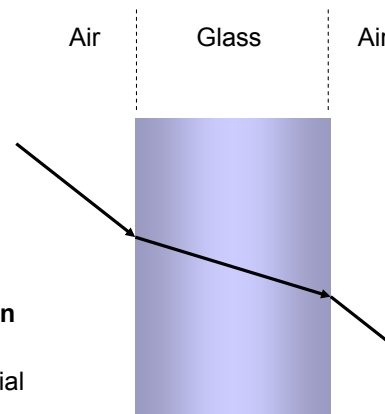
- enable alpha blending
- render objects back to front

Doesn't account for refraction

- light rays bent at material boundaries
- accounts for lenses among other

Can account for refraction like reflection

- when shading a given point
- trace a transmitted ray into the material
- need to compute refracted direction

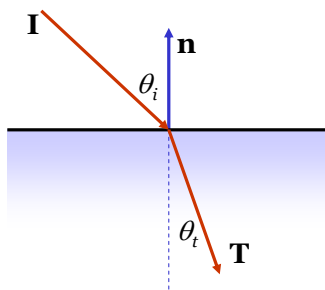


Refraction of Light

Rays transitioning between materials are bent around normal

- every material has an index of refraction

Material	Index of Refraction
<i>vacuum</i>	1.0
<i>ice</i>	1.309
<i>water</i>	1.333
<i>ethyl alcohol</i>	1.36
<i>glass</i>	1.5–1.6
<i>diamond</i>	2.417



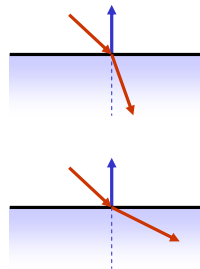
Angles with surface normal obey Snell's Law

$$\frac{\sin \theta_i}{\sin \theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i} \quad \text{where } \eta \text{ are indices of refraction}$$

Refraction of Light

Refractive indices determine amount of bending

- going from low index to higher index
 - ray is bent towards the normal
 - for example: air to glass
- going from high index to lower index
 - ray is bent away from the normal
 - for example: glass to water



Technically, this is a function of wavelength

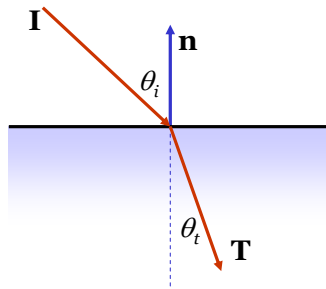
- that's why prisms work (and why you see rainbows)
- but for our purposes here, we'll ignore this detail

Computing the Transmitted Ray

Angles of the incoming & transmitted rays obey Snell's Law

$$\frac{\sin \theta_i}{\sin \theta_t} = \eta_i = \frac{\eta_t}{\eta_i}$$

- this isn't terribly convenient
- need transmitted direction *vector*



With a little math, we can derive the transmitted direction

$$\mathbf{T} = \eta \mathbf{I} + \left(\eta c - \sqrt{1 + \eta^2 (c^2 - 1)} \right) \mathbf{n}$$

$$\text{where } c = \cos \theta_i = -\mathbf{n} \cdot \mathbf{I} \quad \text{and} \quad \eta = \frac{\eta_i}{\eta_t}$$

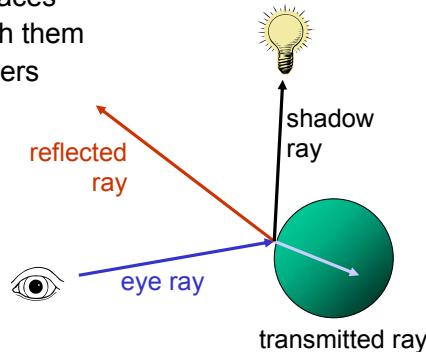
Classification of Rays

We've now seen four kinds of rays in the world

- **eye rays** that leave the eye through a pixel
- **reflected rays** that bounce off surfaces
- **transmitted rays** that travel through them
- **shadow rays** which test for occluders

Every surface intersection spawns

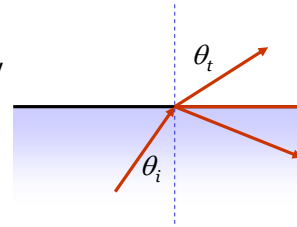
- 1 reflected ray
- 1 transmitted ray
- 1 shadow ray per light



One Last Refractive Detail

When entering material of lower index

- ray bends outward from normal
- what if the angle is more than 90°?
 - ray is actually reflected off the boundary
 - this is called **total internal reflection**
 - and it's why fiber optics work



Total internal reflection occurs when

$$\theta_i > \theta_{\text{critical}} \quad \text{where} \quad \theta_{\text{critical}} = \sin^{-1} \frac{\eta_t}{\eta_i}$$

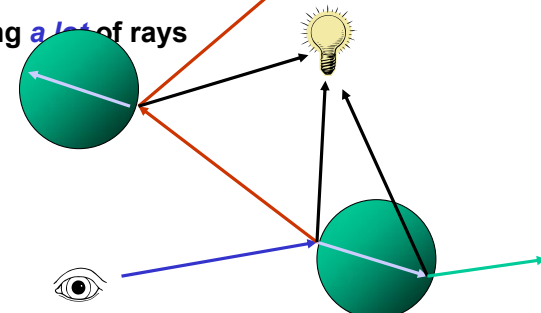
- just need to check for this critical angle
- if above it, use specular reflection for "transmission"
- if we're exactly at the critical angle, things are a little weird

Ray Recursion

Recursive ray tracing spawns a whole tree of rays

- when eye ray hits a surface, we spawn reflected & transmission rays
- when either of them hits a surface, they spawn 2 more
- typically impose maximum recursion limit

We will wind up tracing **a lot** of rays



Structure of a Simple Ray Tracer

```
void raytrace()
for all pixels (x,y)
    image(x,y) = trace(compute_eye_ray(x,y))

rgbColor trace(ray r)
for all surfaces s
    t = compute_intersection(r, s)
    closest_t = MIN(closest_t, t)

if( hit_an_object )
    return shade(s, r, closest_t)
else
    return background_color
```

This all looks identical to a simple ray caster

Structure of a Simple Ray Tracer

```
rgbColor shade(surface s, ray r, double t)
point x = r(t)
rgbColor color = black

for each light source L
    if( closest_hit(shadow_ray(x, L)) >= distance(L) )
        color += shade_phong(s, x)

color += k_specular * trace(reflected_ray(s,r,x))

color += k_transmit * trace(transmitted_ray(s,r,x))

return color
```

Here's where *ray tracing* is different from *ray casting*

- it's the recursive calls to trace()
- to resolve shadows, reflection, and transmission

Antialiasing with Ray Tracing

Remember our simple approach to antialiasing

- render at a higher resolution than desired
- down-sample to output resolution

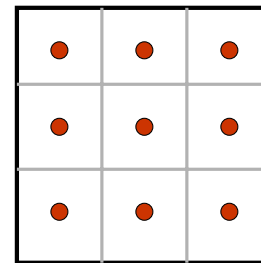
This is particularly convenient to accomplish in ray tracing

- we can supersample every pixel
- instead of shooting 1 ray through the center of each pixel
- we can shoot k rays through different parts of the same pixel

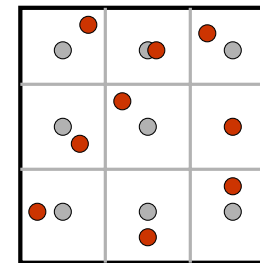
Typically, we subdivide every pixel into a uniform grid

- and shoot a ray through each sub-pixel
- for better results, we typically *jitter* every sub-pixel sample
 - rather than shooting ray through the center of the sub-pixel
 - add a random offset away from the center
 - this helps reduce aliasing by adding noise to the result

Applying Supersampling



3x3 supersampling



3x3 supersampling
with jitter



1 sample/pixel



3x3=9

Ray Tracing Efficiently

The primary path to efficiency

- avoid tracing rays whenever possible
- and above all, avoid ray–surface intersection tests

A ray tracing system can be easily overcome with rays

- minimum 1 eye ray per pixel, many more with supersampling
- recursion depth k yields $2^{k+1} - 1$ rays traced per eye ray
 - counting reflection & transmission rays but *not* shadow rays

Consider this example

- image resolution of $1024 \times 768 = 786,432$ pixels
- 3×3 supersampling = 7 million eye rays
- recursion depth 5 = $63 \times 7 = 441$ million
- each tested against 10,000 polygons
- 4.4 trillion intersection tests (ignoring shadow rays)

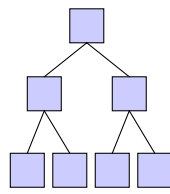
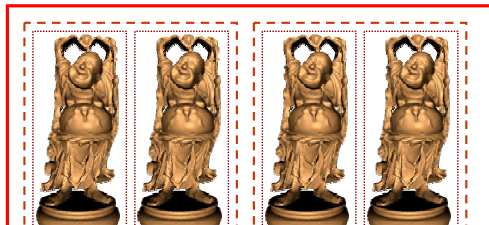
Hierarchical Bounding Volumes

Begin by intersecting ray with the root volume

- if no intersection, ignore all child volumes
- otherwise, recursively test child volumes

We'll only test objects whose bounding volume is actually hit by ray

- hopefully ignoring most of the scene
- but this depends a lot on the structure of the hierarchy



Spatial Data Structures

Probably the single most important efficiency improvement

- divide space into cells
- record what geometry lies in each cell
- first test rays against cell
- only check geometry within cell if the ray actually hits the cell

Several data structures in common practice

- hierarchical bounding volumes
- BSP trees
- octrees
- regular 3-D grids

BSP Trees, Octrees, and Grids

Use BSP tree to traverse scene from front to back

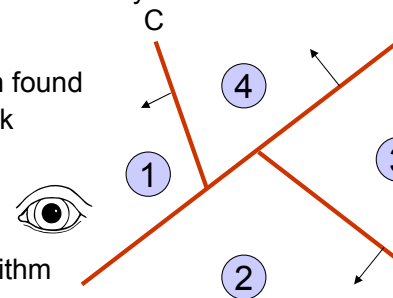
- start at root plane
- figure out which side the viewpoint is on
- descend on that side first; do this recursively

Helps us avoid unnecessary work

- can tell when closest intersection found
- because we're going front to back

Can use grids & octrees similarly

- traverse grid with 3-D DDA algorithm
- octree a little trickier



Recall the Structure of Shading Procedure

```
rgbColor shade(surface s, ray r, double t)
  point x = r(t)
  rgbColor color = black

  for each light source L
    if( closest_hit(shadow_ray(x, L)) >= distance(L) )
      color += shade_phong(s, x)

  color += k_specular * trace(reflected_ray(s, r, x))

  color += k_transmit * trace(transmitted_ray(s, r, x))

  return color
```

What does this simulate?

- Phong illumination+shadows (direct lighting of plastic)
- perfect specular reflection (mirrors, chrome)
- perfect specular transmission (glass, crystal)

Distributed Ray Tracing

A powerful method for extending classical ray tracing

- also known as **distribution ray tracing** & **stochastic ray tracing**
- has *nothing* to do with parallel computation over a network

Simulate phenomena by distributing multiple rays

- ...over a pixel for spatial antialiasing
- ...over time for temporal antialiasing
- ...over a light source to simulate soft shadows
- ...over a cone of directions for glossy reflections
- ...over a lens for depth of field

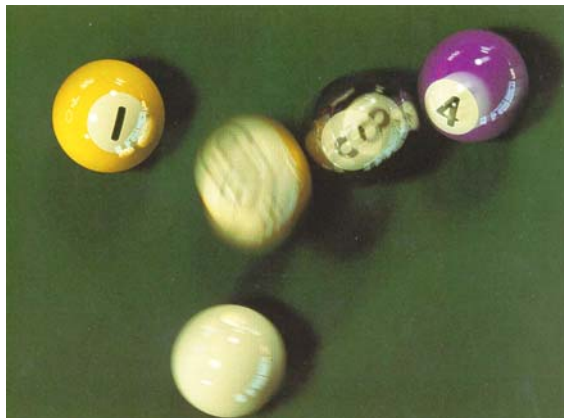
To get good results we must do this with care

- select a good method for distributing rays
- don't want number of rays traced to expand out of control

From the First Distributed Ray Tracer

Note the various phenomena

- glossy reflections
- soft shadows
- motion blur
 - including of reflections
- antialiasing is implicit
- used 16 samples/pixel



Robert L. Cook and Thomas Porter and Loren Carpenter

Glossy Reflections

Our current method simulates perfect specular reflection

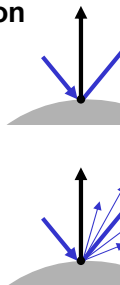
- is only really true for perfect mirrors
- simulates metals like chrome fairly well

Most surfaces are imperfect specular reflectors

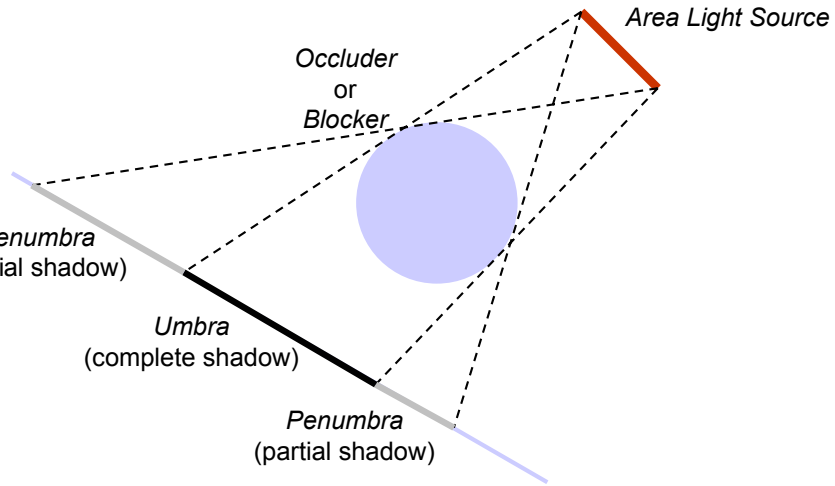
- reflect rays in cone around perfect reflection direction
- glossy reflections rather than mirror images
- Phong model tries to fake this kind of thing

We can directly simulate real glossy reflection

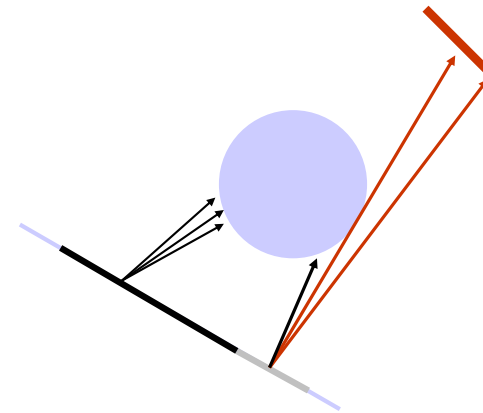
- for an incoming ray direction
 - instead of always shooting ray in perfect reflected direction
 - stochastically sample rays within cone of reflected directions
- strength of reflection drops off rapidly from mirror direction
- probability of sampling that direction should fall off similarly



The Structure of Soft Shadows



Soft Shadows



Our previous shadow method

- for the point we're shading
- cast a ray towards point light
- hit surface before light = shadow
- otherwise no shadow

Extends directly to area lights

- sample multiple spots on light
- look at fraction hitting surfaces
- indicates level of shadow
 - none hit = full illumination
 - all hit = full shadow
 - some hit = partial shadow