

COMP3271 Computer Graphics

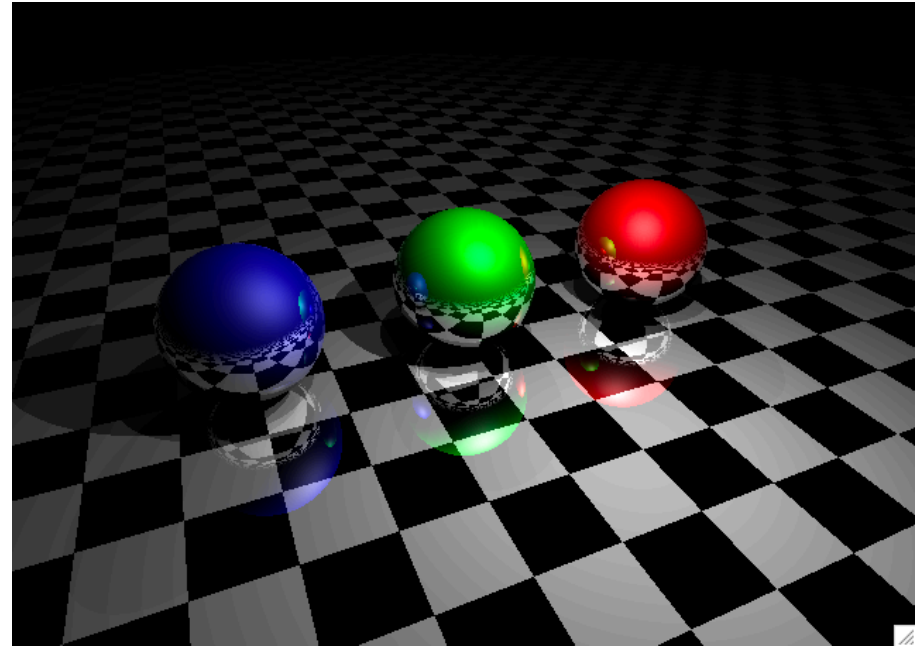
# Ray Tracing

---

2019-20

# Objectives

- Ray Casting
- Ray Tracing
- Speedup Techniques
- Intersection Tests



- POV-Ray - an open sourced raytracer:
  - <http://www.povray.org/>
  - <https://en.wikipedia.org/wiki/POV-Ray>

# Looking Back at Image Formation

Light is a stream of photons

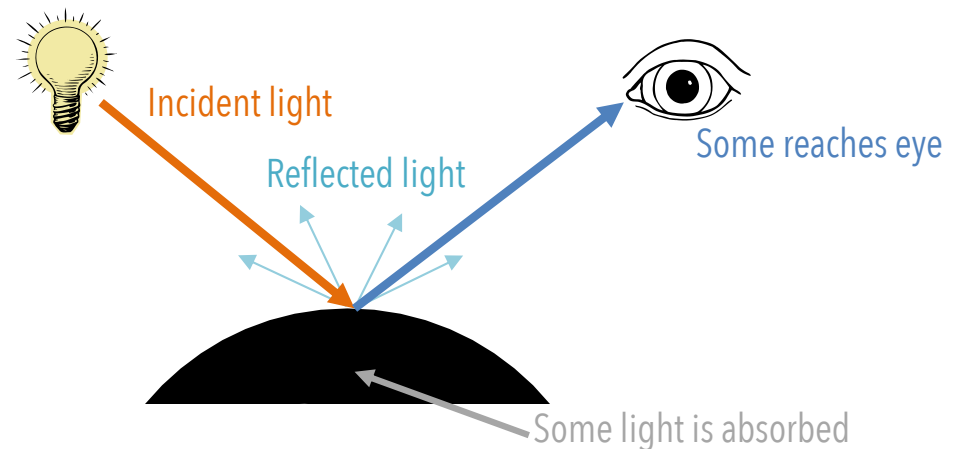
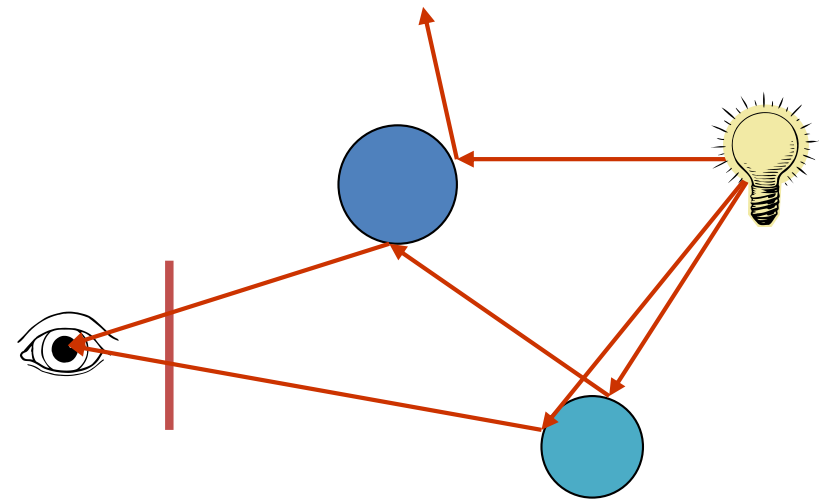
- which move in straight lines
- propagating from lights
- we ignore wave nature of light

Some rays strike the eye

- (passing through image plane)
- these rays form the image

Light interacts with surfaces

- objects absorb some light
- reflect some of the light
- may bounce off many objects



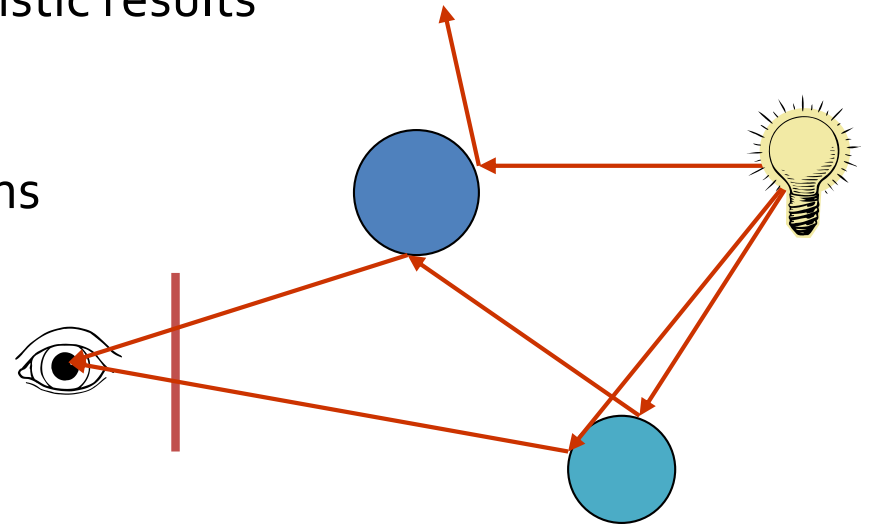
# Simulating Rays of Light in the World

We can render the scene by simulating physical light transport

- we hope that this produces more realistic results

Simulation would look like this:

- light source shoots rays in all directions
- rays bounce when they hit surfaces
- can ignore rays when
  - they fly off into empty space
  - almost all of their energy is absorbed
- record rays that strike the image plane
- we call this kind of simulation **forward ray tracing**



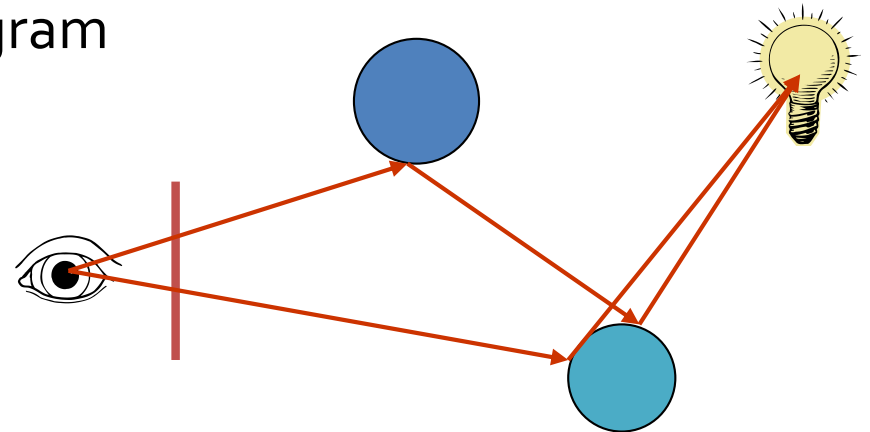
But there's a problem with this

- it can be extremely slow
- only a tiny fraction of light rays actually strike the eye

# (Backward) Ray Tracing

Fortunately, there's a simple solution to this problem

- we only care about light rays that eventually strike the eye
- so shoot rays from the eye out into the world
- just reverse arrows on the ray diagram



Traditionally, most ray-based renderers take this approach

- so we usually drop the “backward” from the name
- but keep in mind there are alternatives that don't

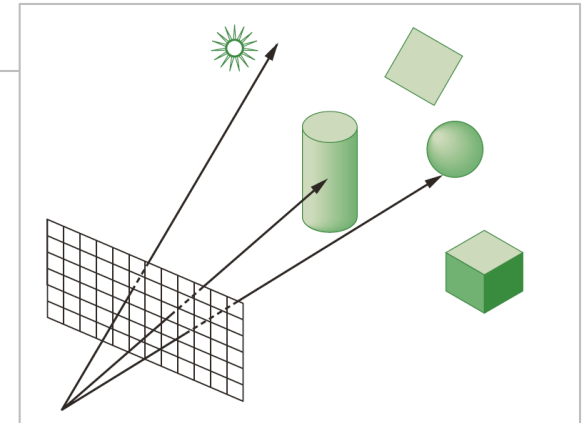
# Ray Casting: Simple Ray-Based Rendering

We can formulate a very simple rendering algorithm

- we'll ignore all this business about rays bouncing around
- just shoot rays into world, see what they strike, and shade

## Ray Casting Algorithm:

```
for all pixels (x,y)
  compute ray from eye through (x,y)
  compute intersections with all surfaces
  find surface with closest intersection
  shade this surface point (standard illumination equation)
  write this color into pixel (x,y)
```



What we need to resolve

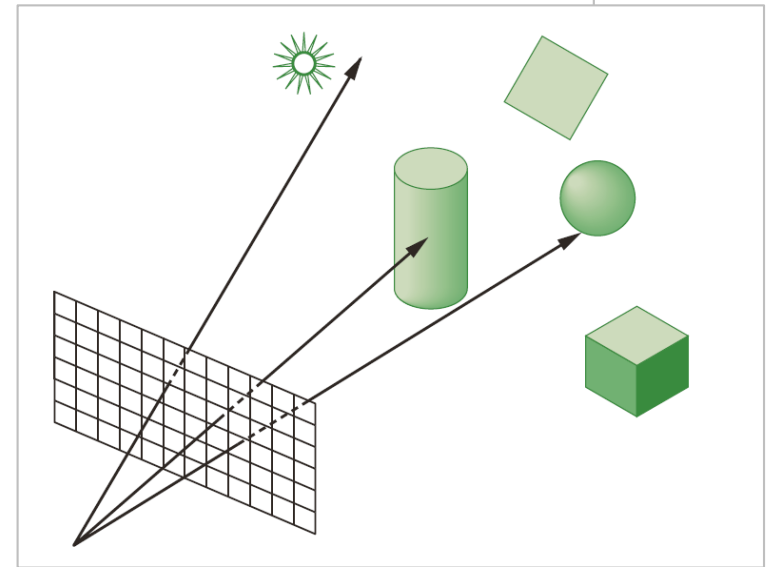
- how to represent rays & generate rays through screen
- how to compute intersections with objects in the world

# Pseudo-Code Outline of a Minimal Ray Caster

```
void raycast()  
  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
```

```
rgbColor shade(surface s, ray r, double t)  
  point x = r(t)  
  // evaluate (Phong) illumination equation  
  return color
```



# From Ray Casting to Ray Tracing

We developed the simple ray casting algorithm

- essentially, replacing z-buffer+rasterization with ray probes
- we still used the Phong illumination model
- thus the results look like OpenGL, only much slower

We want to extend this simple algorithm

- our main focus will be on developing a better shading model

## Ray Casting Algorithm:

for all pixels  $(x,y)$

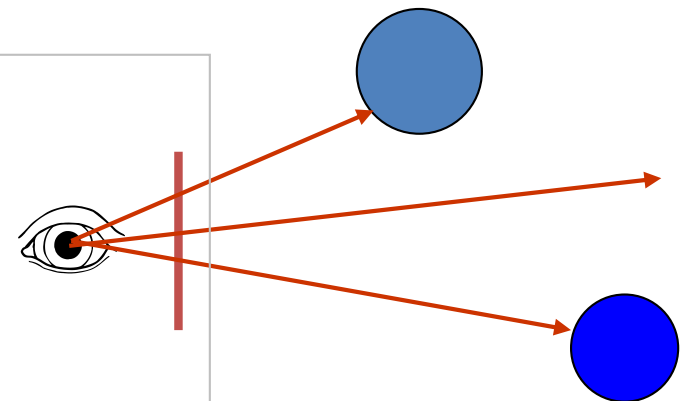
compute ray from eye through  $(x,y)$

compute intersections with all surfaces

find surface with closest intersection

compute color using Phong illumination model

write this color into pixel  $(x,y)$





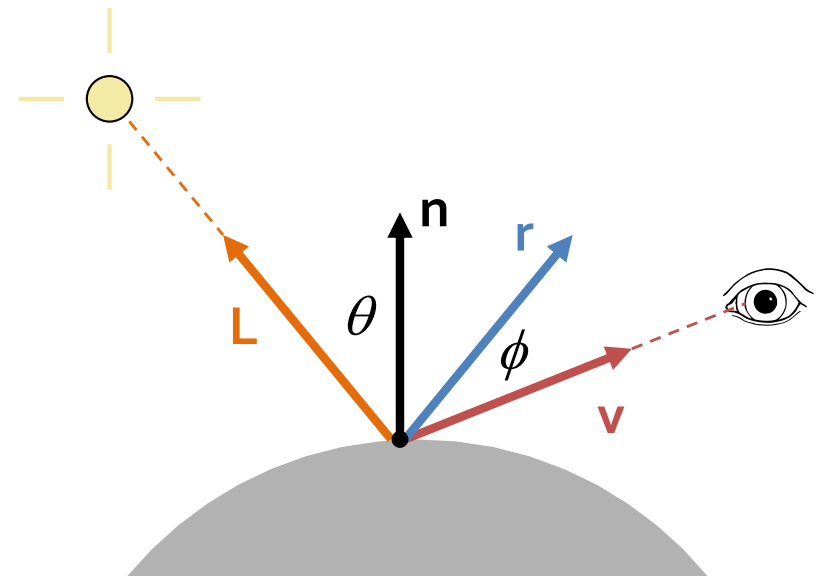
# Looking at the Shading Model

Recall our current Phong illumination model

$$I = \sum_{\text{all lights } L} I_L k_d (\mathbf{n} \cdot \mathbf{L}) + I_L k_s (\mathbf{r} \cdot \mathbf{v})^n$$

diffuse reflection      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**

# Adding Shadows

We've found the nearest surface hit for a given ray

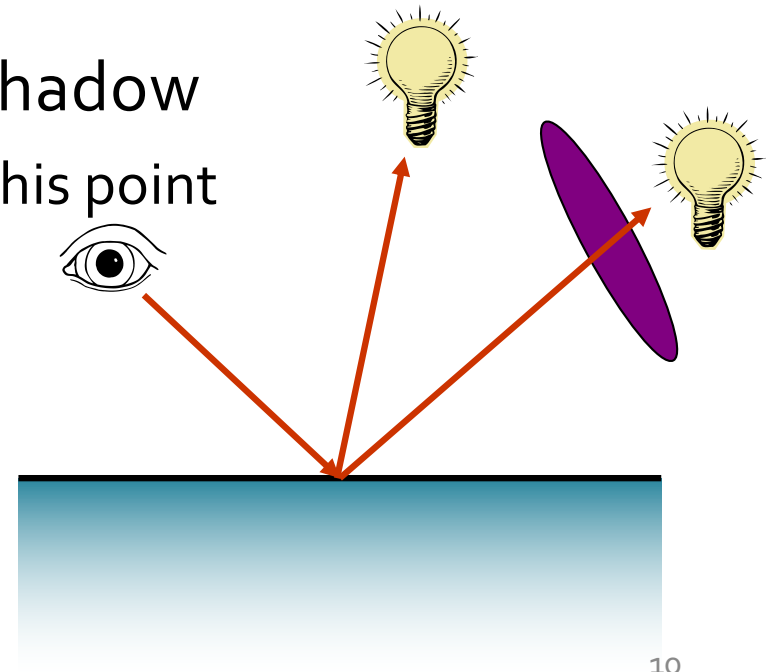
- now we want to shade this point
- for each light, we evaluate our diffuse & specular terms

For a given light, the point may or may not be in shadow

- if it is in shadow, don't add contribution for this light

We can easily test whether we're in shadow

- trace a new **shadow ray**, starting at this point
- heading towards the given light
- hit any surface closer than light?
  - yes: we're in shadow
  - else: no shadow



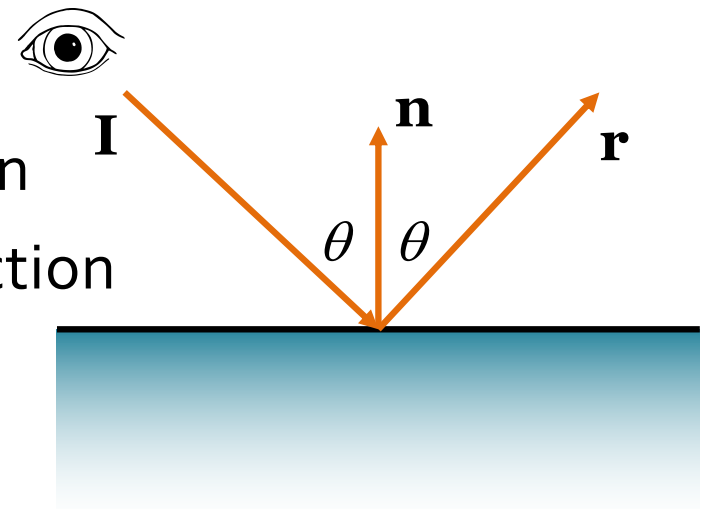
# Computing Correct Reflections

If we're tracing rays, we can easily compute correct reflections

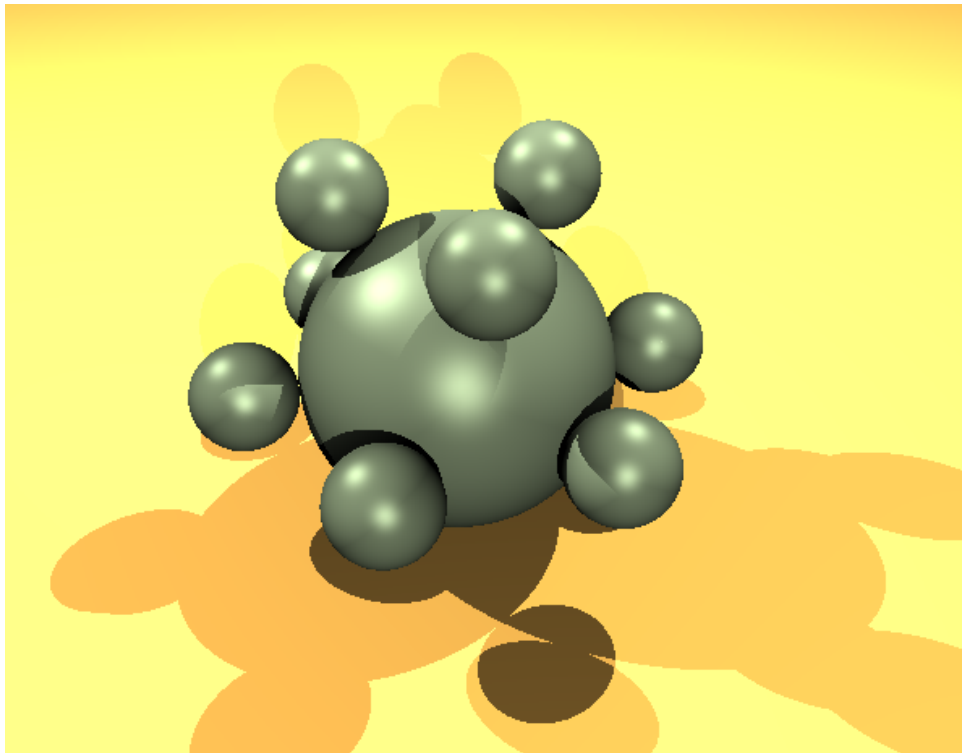
- at a given point, the perfect reflected direction is
$$\mathbf{r} = \mathbf{I} - 2(\mathbf{n} \cdot \mathbf{I})\mathbf{n}$$
- trace a ray going in that direction
- the returned color is the reflection

Remember to keep in mind

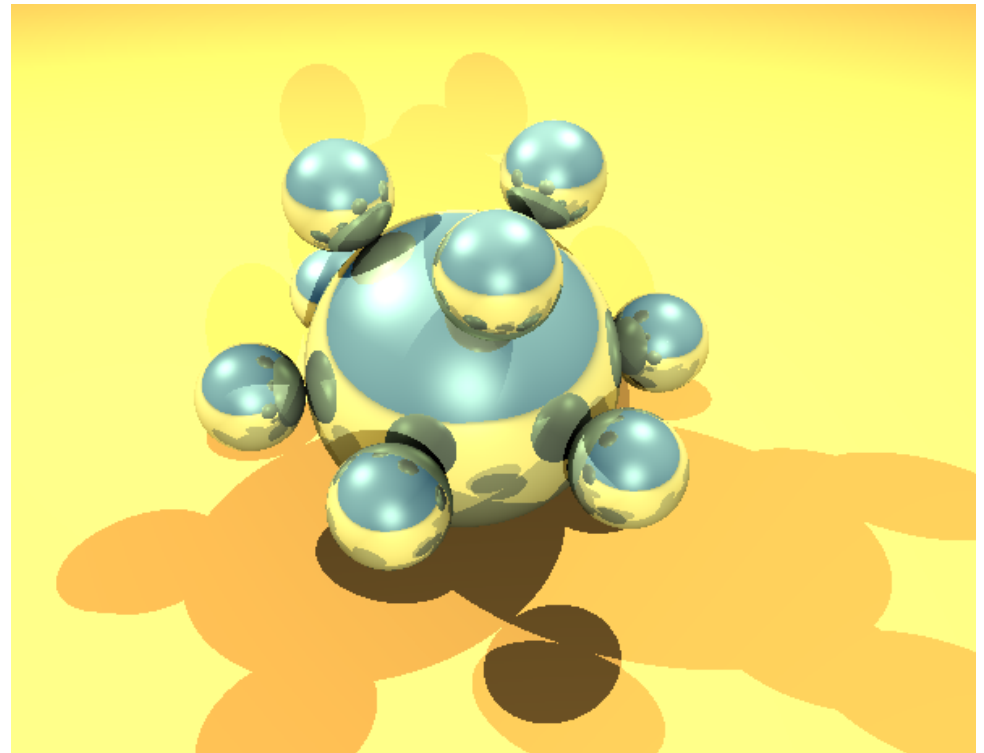
- these arrows are the tracing direction
- light is propagating in opposite direction



# An Example: Adding Reflected Rays

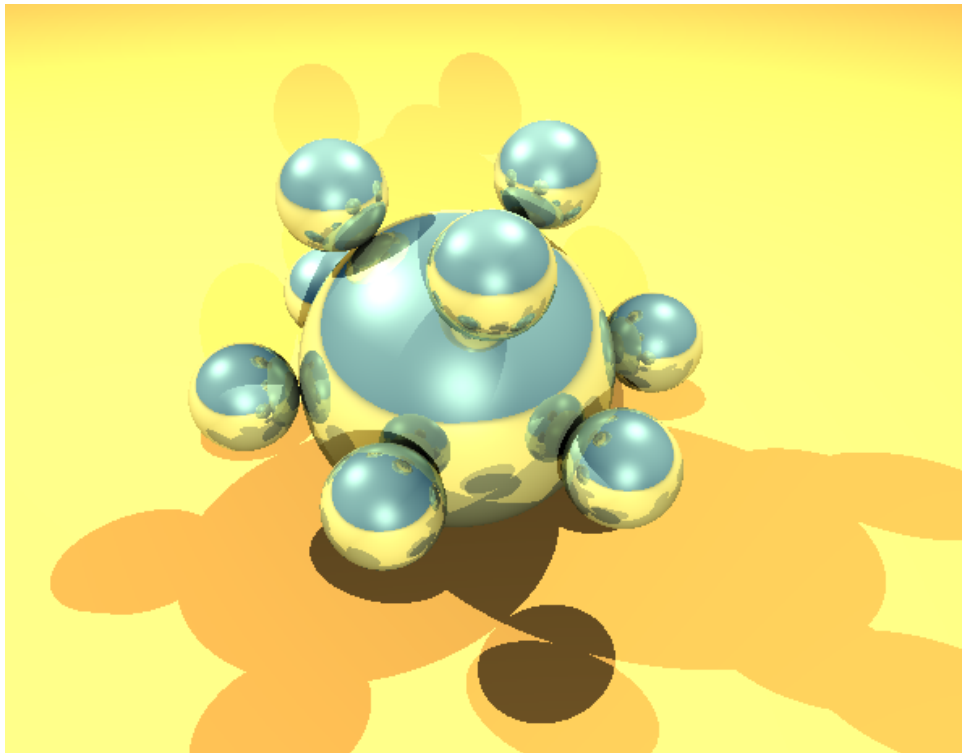


No recursive rays

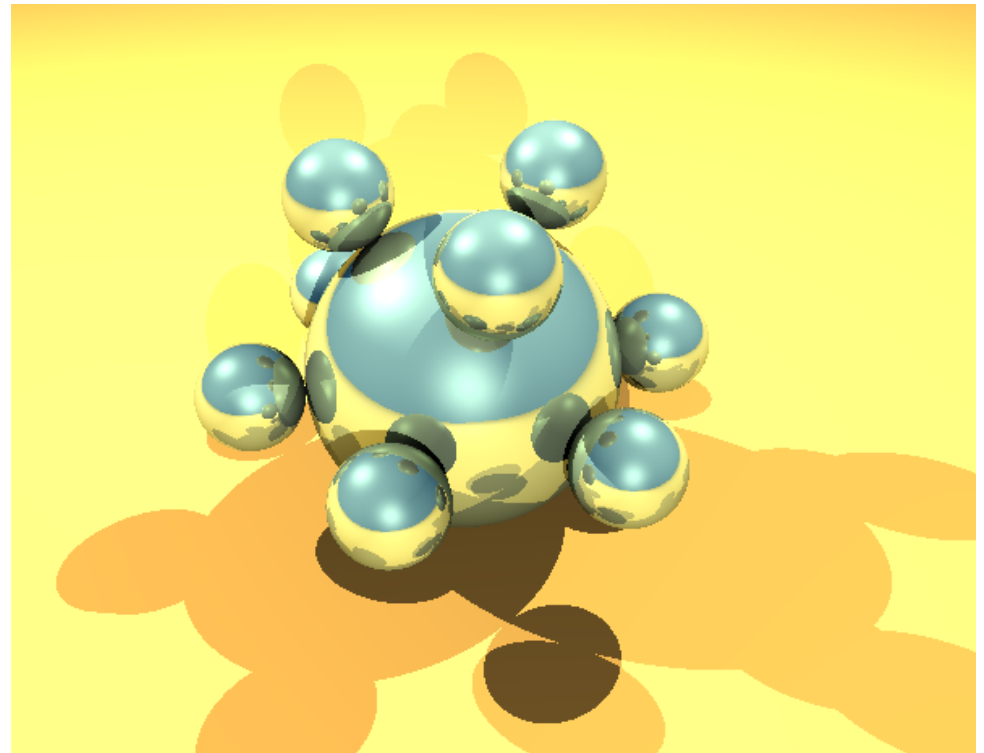


1 level of recursive reflection

# An Example: Adding Reflected Rays



2 levels of recursive reflection



1 level of recursive reflection

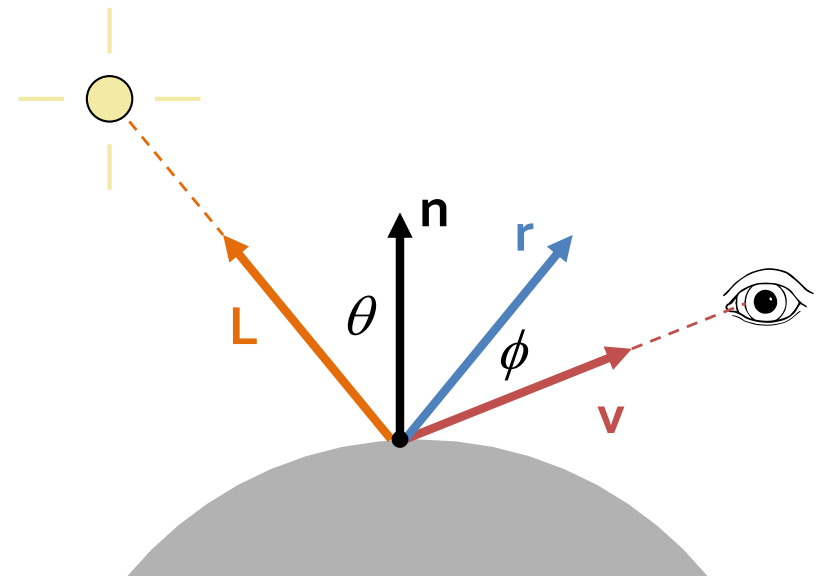
# Looking at the Shading Model

Recall our current Phong illumination model

$$I = \sum_{\text{all lights } L} I_L k_d (\mathbf{n} \cdot \mathbf{L}) + I_L k_s (\mathbf{r} \cdot \mathbf{v})^n$$

diffuse reflection      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

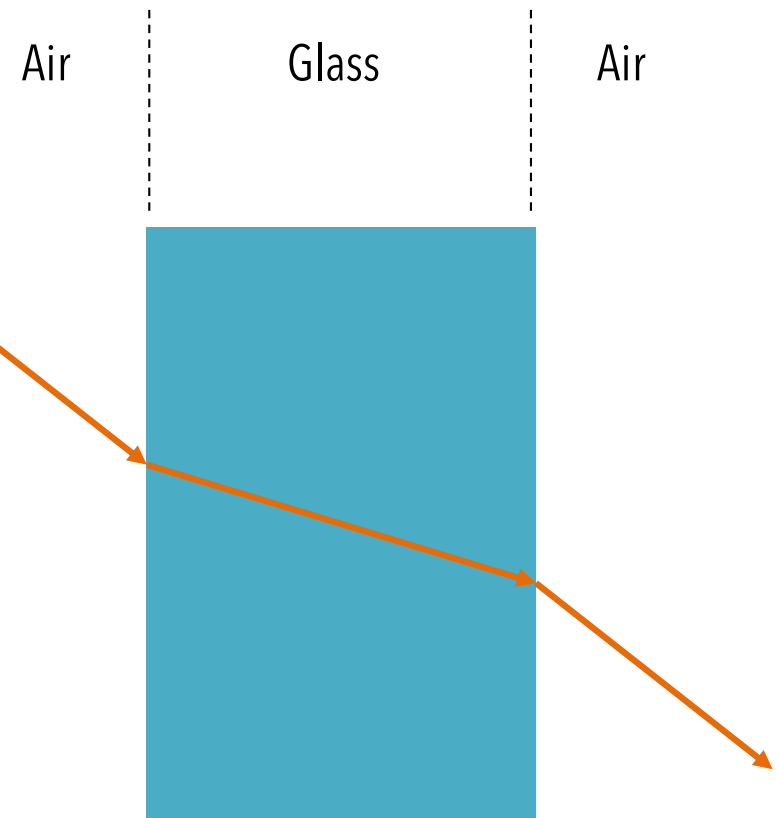
- enable alpha blending
- render objects back to front

OpenGL doesn't account for refraction

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

Ray Tracing can account for refraction like reflection

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

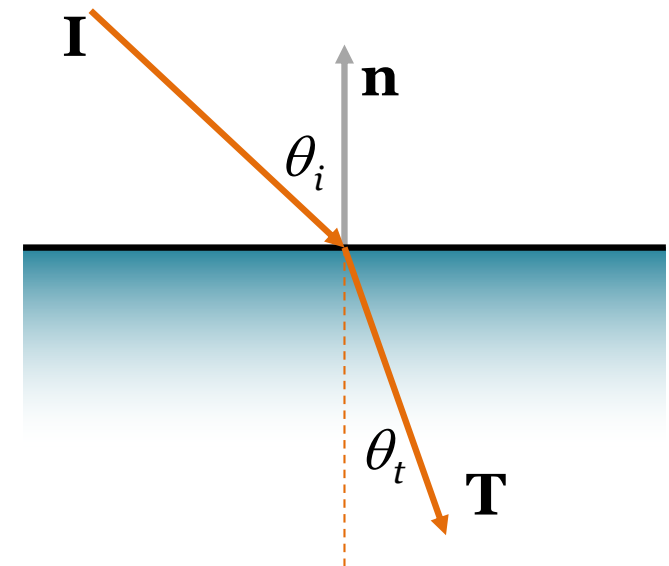


# Refraction of Light

Rays transiting between materials are bent around normal

- every material has an **index of refraction**

Material	Index of Refraction
vacuum	1.0
ice	1.309
water	1.333
ethyl alcohol	1.36
glass	1.5-1.6
diamond	2.417



Angles with surface normal obey Snell's Law

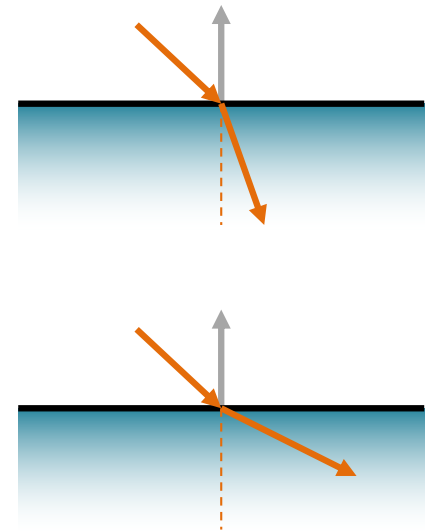
$$\frac{\sin \theta_i}{\sin \theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i} \quad \text{where } \eta \text{ are indices of refraction}$$



# 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

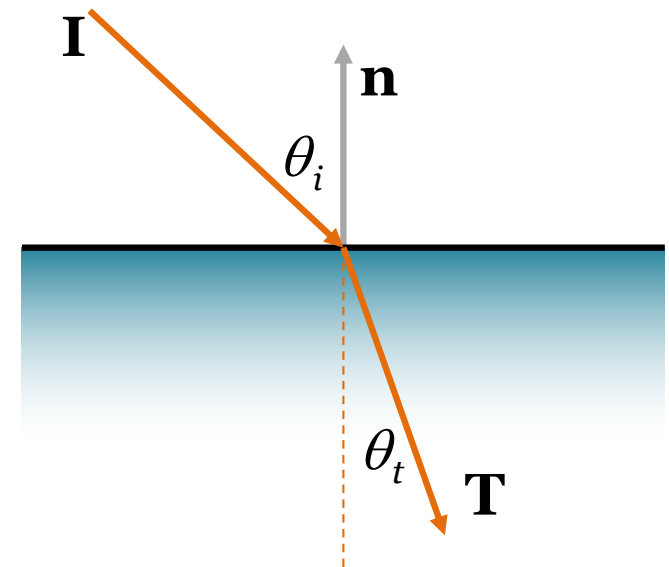
$$\frac{\sin \theta_i}{\sin \theta_t} = \eta_{ti} = \frac{\eta_t}{\eta_i}$$

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

With a little math, we can derive the  
transmitted direction

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

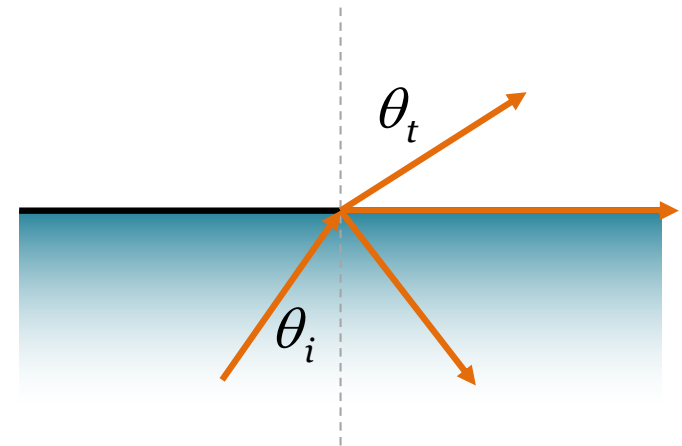
$$\text{where } c = \cos \theta_i = -\mathbf{n} \cdot \mathbf{I} \quad \text{and} \quad \eta = \frac{\eta_i}{\eta_t}$$



# One Last Refractive Detail

When entering material of lower index

- ray bends outward from normal
- what if the angle is more than  $90^\circ$ ?
  - ray is actually reflected off the boundary
  - this is called **total internal reflection**
  - and it's how 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

Take a look at the YouTube video: <https://youtu.be/NAaHPRsveJk>

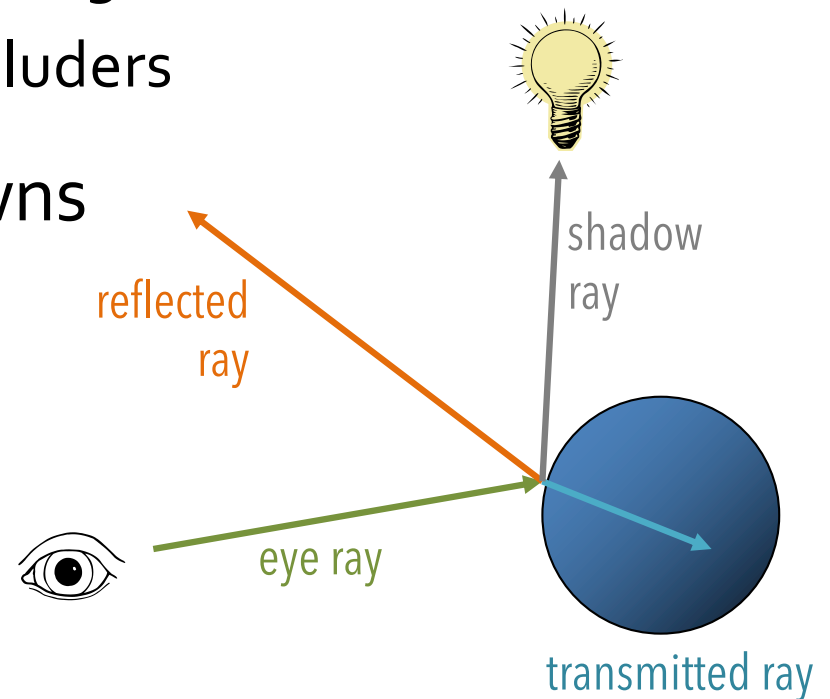
# Classification of Rays

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

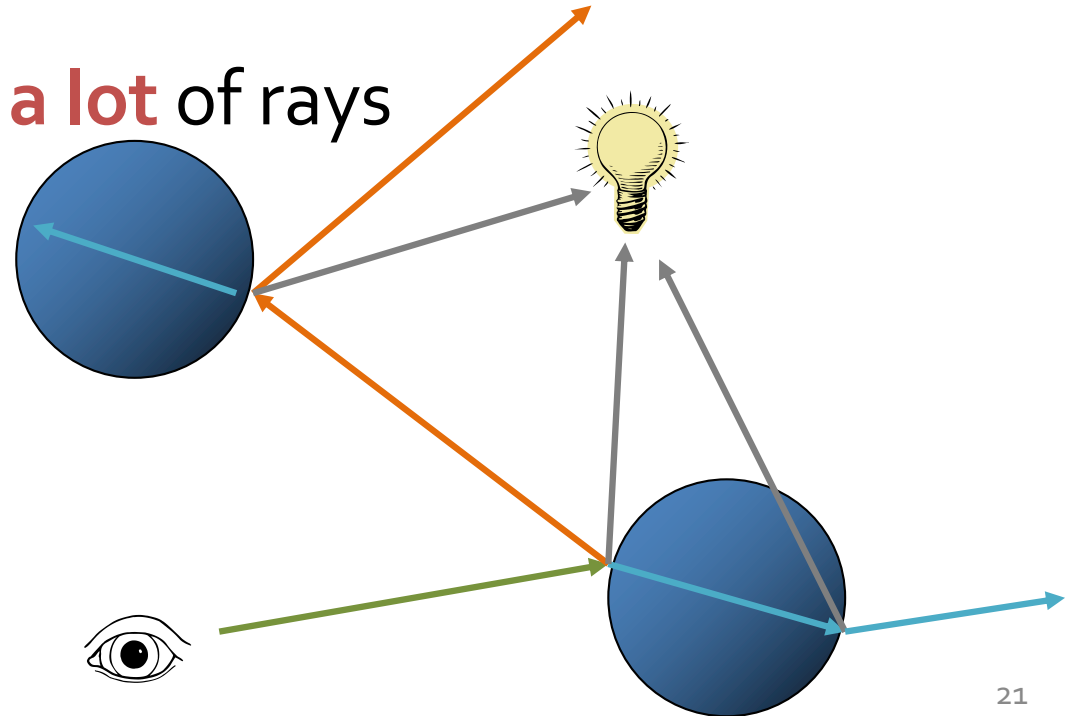


# Ray Recursion

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

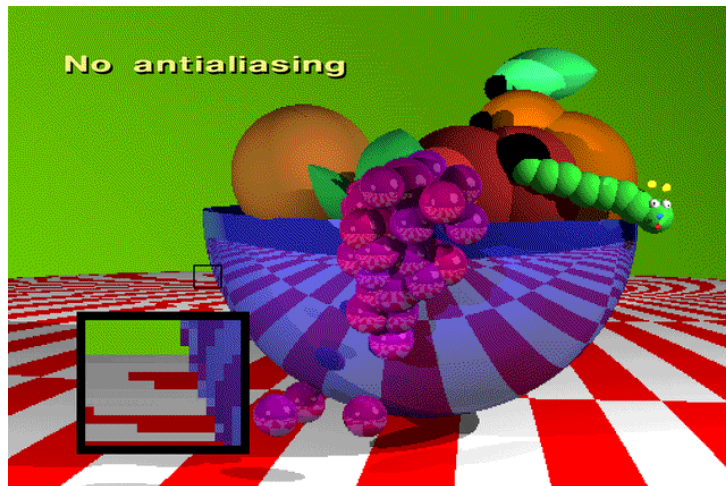
# Ray Tracing Efficiently

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





# Ray Tracing Efficiently

Consider this example

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

# 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

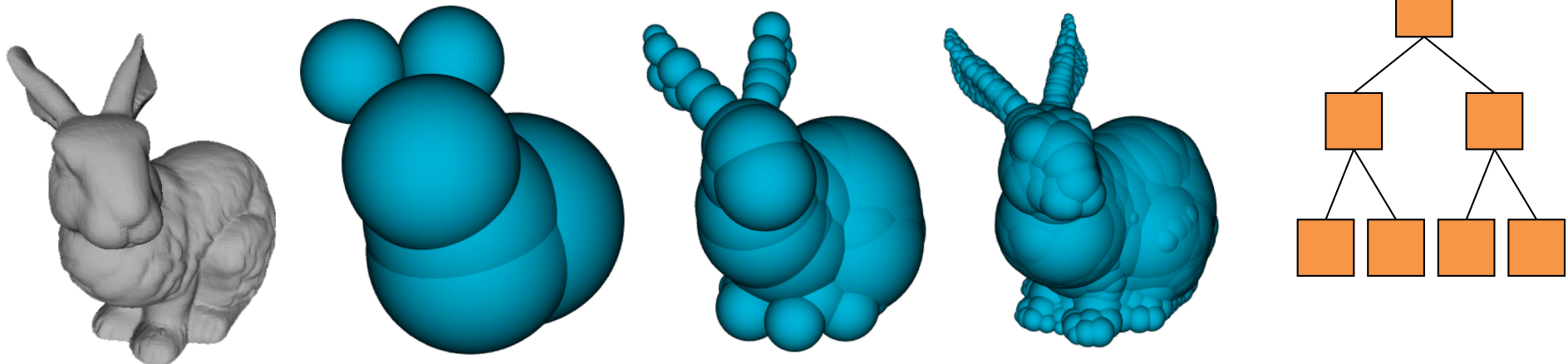
# Hierarchical Bounding Volumes

Begin by intersecting ray with the root volume

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

Test only 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



# BSP Trees, Octrees, and Grids

Use BSP tree to traverse scene **from front to back**

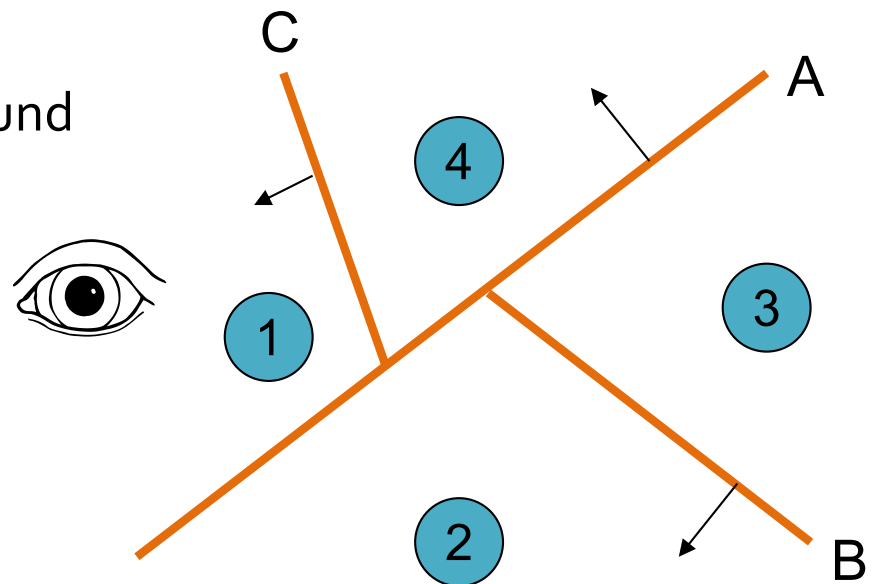
- start at root plane
- figure out which side the viewpoint is on
- descend on that side first; do this recursively

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 3D DDA algorithm
- octree a little trickier




# Shadows

## Soft Shadows – A Quick Recap



- Area lights give soft shadows

• Point light

 Area light

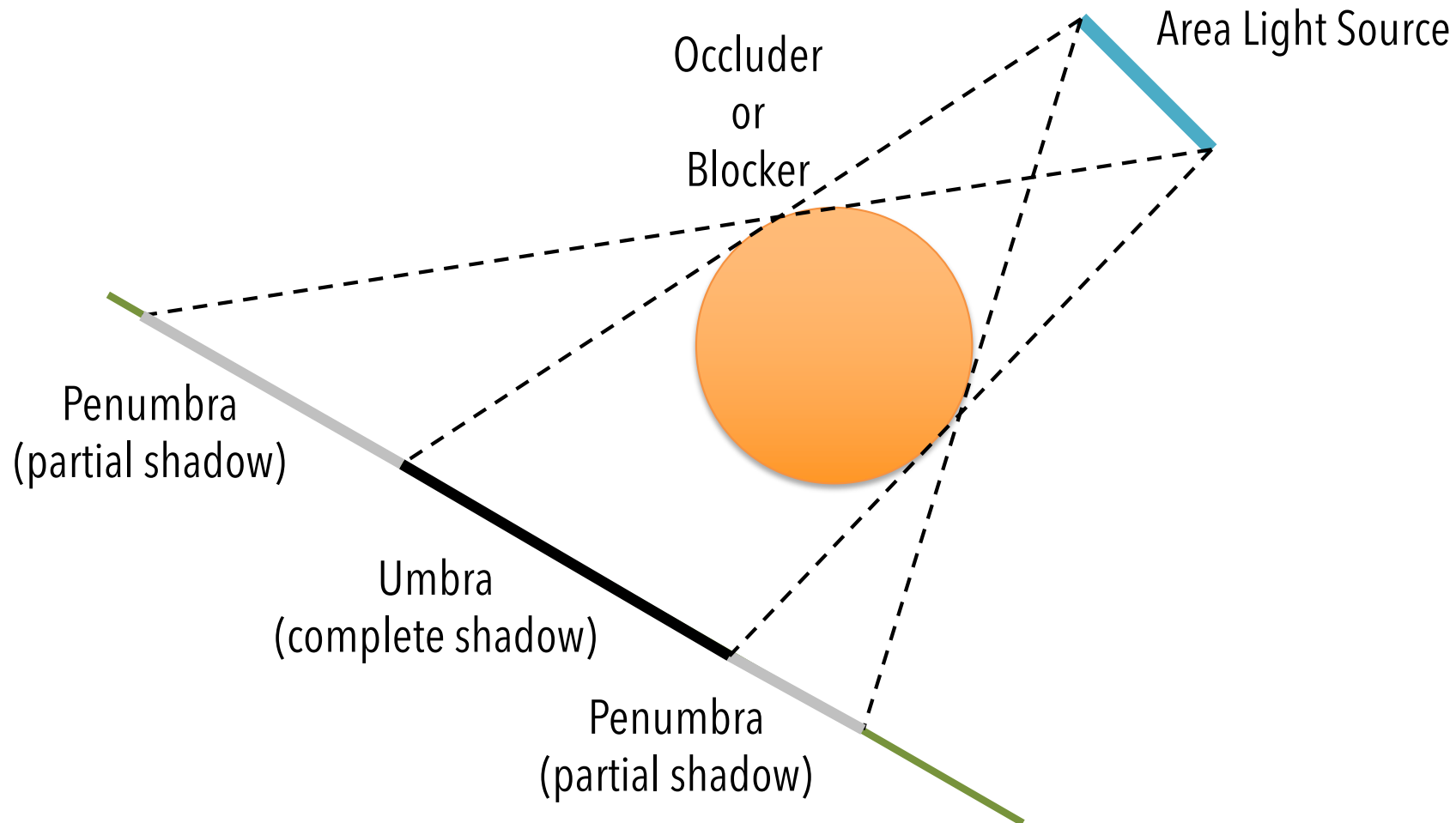


Hard shadow



Soft shadow

# The Structure of Soft Shadows



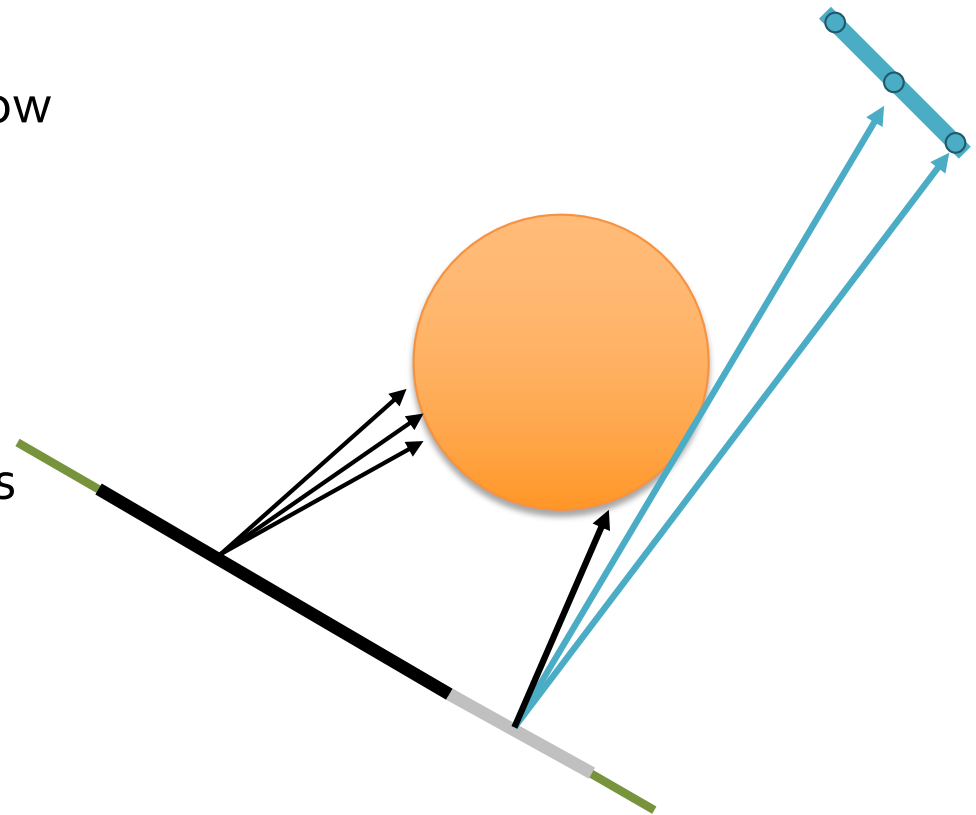
# Soft Shadows

## 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



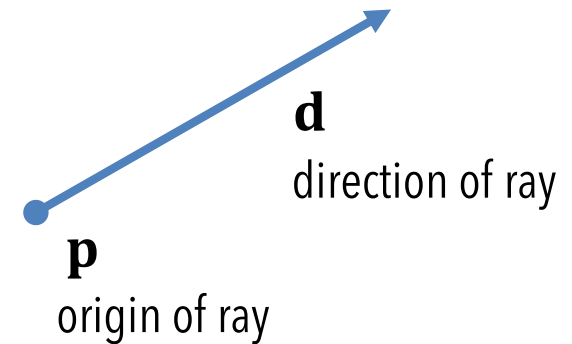
# Representing Light Rays

Geometrically, a ray is just a starting point plus a direction

- the set of all points described by

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{d} \quad \text{where } t > 0$$

- implementation tip:**  
make sure  $\mathbf{d}$  is unit vector



Each ray will return some amount of light from the world

- for implementation purposes, an **RGB color**



# Computing Ray–Surface Intersections

We start with an equation of our ray

$$\mathbf{x}(t) = \mathbf{p} + t\mathbf{d} \quad \text{where } t > 0$$

General idea:

write equation for point on both ray & surface

- for an implicit surface  $f(\mathbf{x}) = 0$ 
  - substitute ray equation

$$f(\mathbf{p} + t\mathbf{d}) = 0$$

- for a parametric surface  $\mathbf{x} = S(u, v)$ 
  - find location where distance between ray and surface is 0

$$S(u, v) - (\mathbf{p} + t\mathbf{d}) = 0$$

in general, both approaches can require expensive root finding

# Ray–Plane Intersection

For specific surfaces, can derive more efficient methods

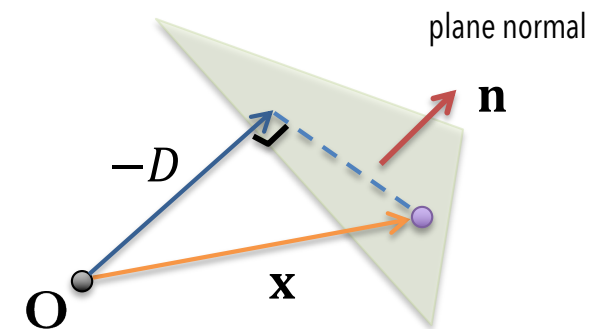
We start with the equation for the plane  $\mathbf{n} \cdot \mathbf{x} + D = 0$

Then substitute the ray equation into it and solve for  $t$

$$\mathbf{n} \cdot (\mathbf{p} + t\mathbf{d}) + D = 0$$

$$\mathbf{n} \cdot \mathbf{p} + t\mathbf{n} \cdot \mathbf{d} + D = 0$$

$$t_0 = \frac{-(\mathbf{n} \cdot \mathbf{p} + D)}{\mathbf{n} \cdot \mathbf{d}}$$



The ray hits the plane if and only if  $t_0 \geq 0$

To find the actual intersection point of the ray with the plane

- substitute the solution  $t_0$  back into the ray equation

# Ray–Polygon Intersection

First, find the intersection of the ray and the polygon's plane

Then we just need to determine whether this point is in the polygon

- there are many approaches to point-in-polygon testing

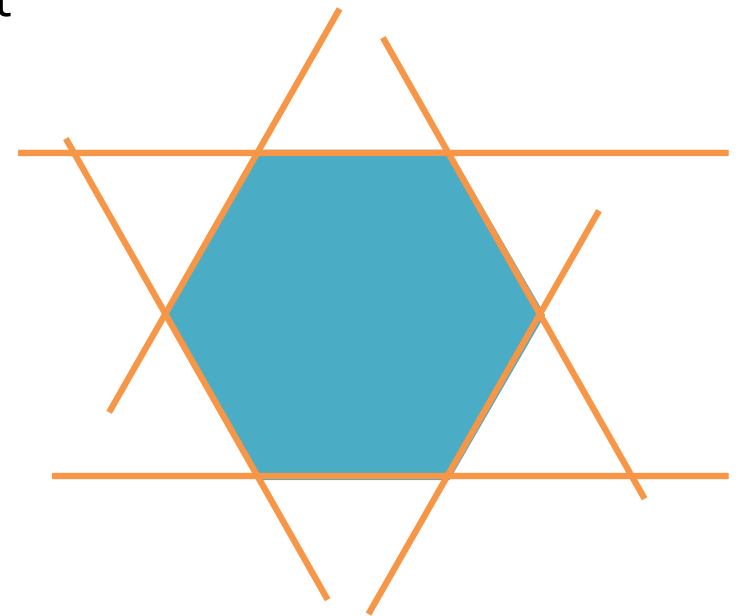
# Point-in-Polygon Test for Convex Polygons

For general **convex polygons**, we can use **half-space tests**

- construct lines  $a_i x + b_i y + c_i = 0$  through each edge of the polygon
- must make sure that normals are consistently oriented
  - either they all point in or they all point out
- the point  $(x, y)$  is in the polygon if  $a_i x + b_i y + c_i$  all have the same sign

Note that this also works in 3-D

- construct planes through each edge
- perpendicular to polygon plane
- point must lie on inside of all of them



# Point-in-Triangle Tests

With triangles, can make good use of **barycentric coordinates**

- all points in triangle satisfy the equation

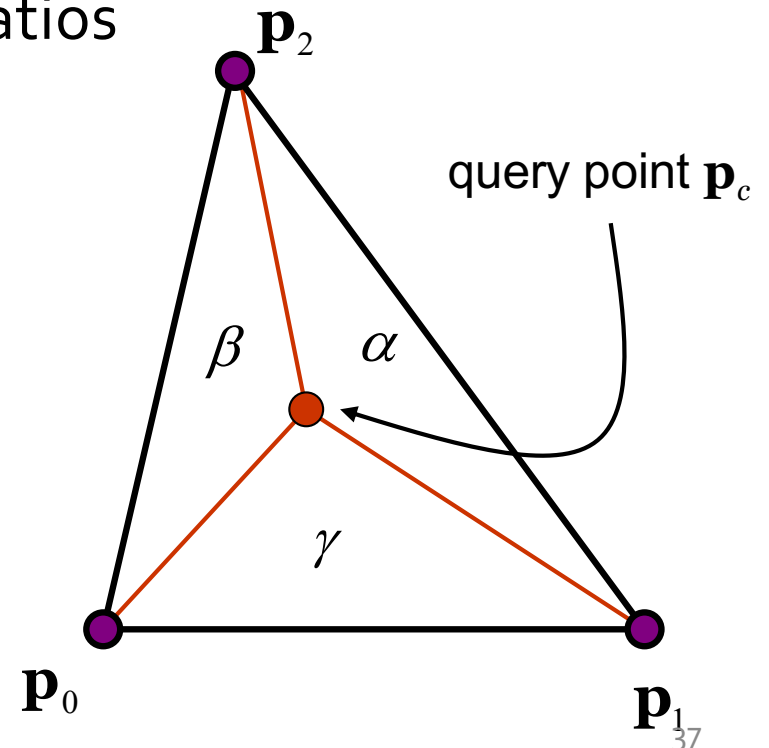
$$\mathbf{p} = \alpha \mathbf{p}_0 + \beta \mathbf{p}_1 + \gamma \mathbf{p}_2 \text{ where } \alpha + \beta + \gamma = 1$$

- these coefficients are triangle area ratios

$$\alpha = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_1, \mathbf{p}_2)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

$$\beta = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_2, \mathbf{p}_0)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)}$$

$$\gamma = \frac{\text{Area}(\mathbf{p}_c, \mathbf{p}_0, \mathbf{p}_1)}{\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)} = 1 - \alpha - \beta$$



# Point-in-Triangle Test

We can compute the 2-D area of a triangle as

$$\text{Area}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2) = \frac{1}{2} \begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix} = \frac{(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)}{2}$$

- note: this is the **signed area** of the triangle
  - it's positive if points are in counter-clockwise order
  - and negative if they're in clockwise order

So, to figure out if a given point is in the triangle

- compute it's three barycentric coordinates
- point is inside the triangle exactly when  $\alpha, \beta, \gamma > 0$
- note that this can also be made to work directly in 3-D

# Ray–Sphere Intersection

Consider a sphere centered at the origin

$$x^2 + y^2 + z^2 - r^2 = \mathbf{x} \cdot \mathbf{x} - r^2 = 0$$

We substitute the ray equation

$$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - r^2 = 0$$

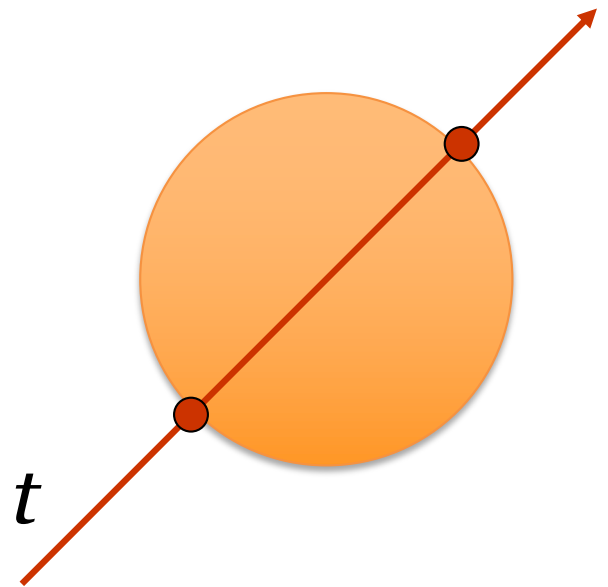
$$\mathbf{p} \cdot \mathbf{p} + 2t\mathbf{p} \cdot \mathbf{d} + t^2\mathbf{d} \cdot \mathbf{d} - r^2 = 0$$

Which gives us a quadratic equation in  $t$

$$At^2 + Bt + C = 0 \quad \text{where} \quad A = \mathbf{d} \cdot \mathbf{d} = 1 \quad (\mathbf{d} \text{ is unit vector})$$

$$B = 2\mathbf{p} \cdot \mathbf{d}$$

$$C = \mathbf{p} \cdot \mathbf{p} - r^2$$



# Ray–Sphere Intersection

We can directly solve this quadratic equation

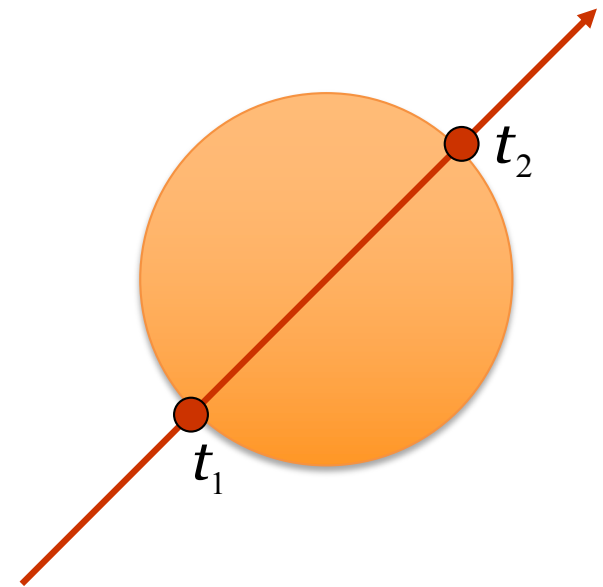
$$At^2 + Bt + C = 0 \quad \text{where} \quad A = \mathbf{d} \cdot \mathbf{d} = 1 \quad (\mathbf{d} \text{ is unit vector})$$

$$B = 2\mathbf{p} \cdot \mathbf{d}$$

$$C = \mathbf{p} \cdot \mathbf{p} - r^2$$

For the two intersection locations

$$t_1 = \frac{-B - \sqrt{B^2 - 4C}}{2} \quad t_2 = \frac{-B + \sqrt{B^2 - 4C}}{2}$$



- the smaller (non-negative) one is the closest ray intersection
- a negative discriminant means that the ray missed the sphere



# Ray-Quadrics Intersection

For general quadrics with equation  $Q: \mathbf{X}^T \mathbf{A} \mathbf{X} = 0$  where  $\mathbf{X}$  is the homogeneous coordinates  $(x, y, z, 1)^T$ , and  $\mathbf{A}$  is a  $4 \times 4$  real symmetric matrix,

substituting the ray equation  $\mathbf{p} + t\mathbf{d}$  to  $Q$ , we have

$$\begin{aligned} (\mathbf{p} + t\mathbf{d})^T \mathbf{A} (\mathbf{p} + t\mathbf{d}) &= 0 \\ \mathbf{p}^T \mathbf{A} \mathbf{p} + 2\mathbf{p}^T \mathbf{A} \mathbf{d} t + \mathbf{d}^T \mathbf{A} \mathbf{d} t^2 &= 0 \quad (*) \end{aligned}$$

The determinant of this equation is

$$\Delta = 4[(\mathbf{p}^T \mathbf{A} \mathbf{d})^2 - (\mathbf{p}^T \mathbf{A} \mathbf{p})(\mathbf{d}^T \mathbf{A} \mathbf{d})].$$

Ray intersects the quadric

if and only if  $\Delta \geq 0$  and a root of  $(*)$  is non-negative