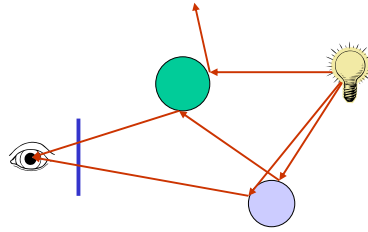


Looking Back at Image Formation

Light is a stream of photons

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

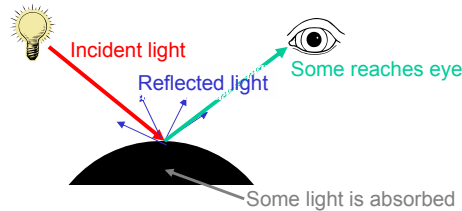


Some rays strike the eye

- (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



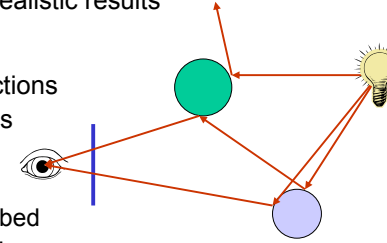
Simulating Rays of Light in the World

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

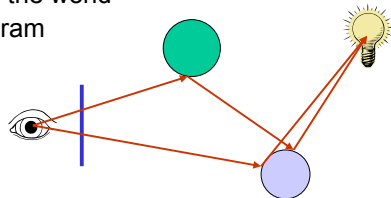
it can be extremely slow

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

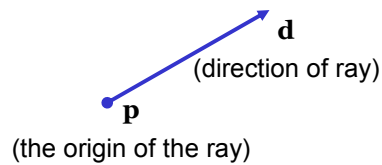
Representing Light Rays

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

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

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

$$\mathbf{x} = h(u, v)$$

- find location where distance between ray and surface is 0

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

- 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

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

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

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

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

- substitute the computed value of t back into the ray equation

Ray–Polygon Intersection

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

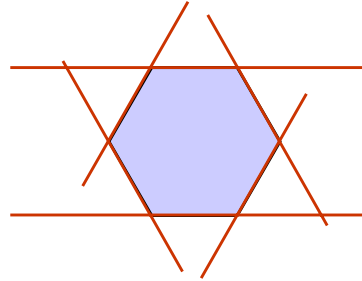
- we just need to determine whether this point is in the polygon
- there are many approaches to point-in-polygon testing

Point-in-Polygon Test for Convex Polygons

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

- construct lines 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

$$a_i x + b_i y + c_i \text{ all have same sign}$$



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 equation

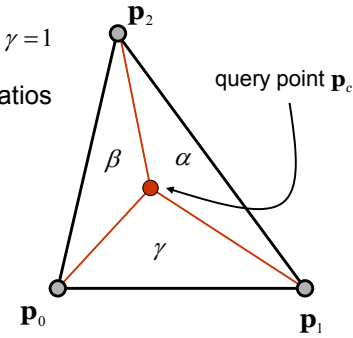
$$\mathbf{p} = \alpha \mathbf{p}_0 + \beta \mathbf{p}_1 + \gamma \mathbf{p}_2 \text{ where } \alpha + \beta + \gamma = 1$$

- 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

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

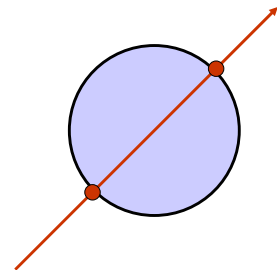
$$\mathbf{p} \cdot \mathbf{p} + 2t\mathbf{p} \cdot \mathbf{d} + t^2 \mathbf{d} \cdot \mathbf{d} - r^2 = 0$$

Which gives us a quadratic equation in t

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

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

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



Ray-Sphere Intersection

We can directly solve this quadratic equation

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

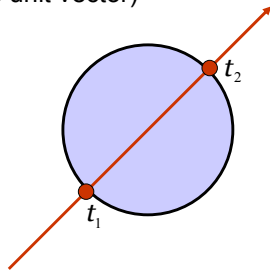
$$B = 2\mathbf{p} \cdot \mathbf{d}$$

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

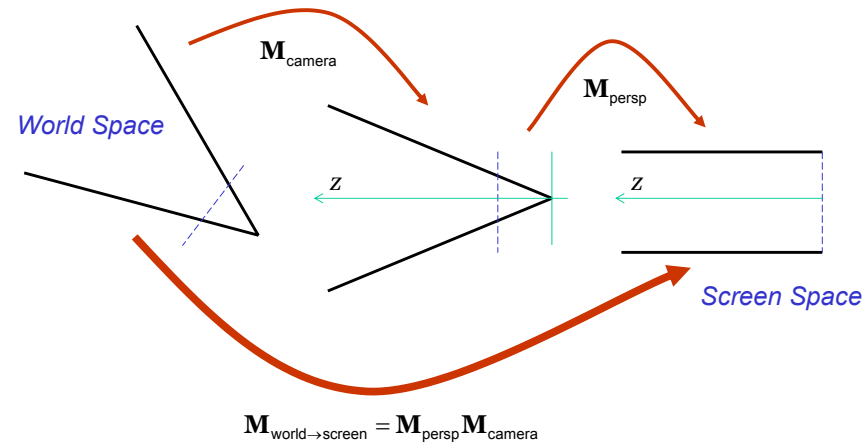
For the two intersection locations

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

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



The Geometry of Screen Space



Generating Eye Rays

Need to construct ray from eye through each pixel

Start with screen space coordinates

$$\text{pixel } \mathbf{q}_s = (x, y, 0, 1) \quad \text{eye } \mathbf{p}_c = (0, 0, 0, 1)$$

And transform them to world space

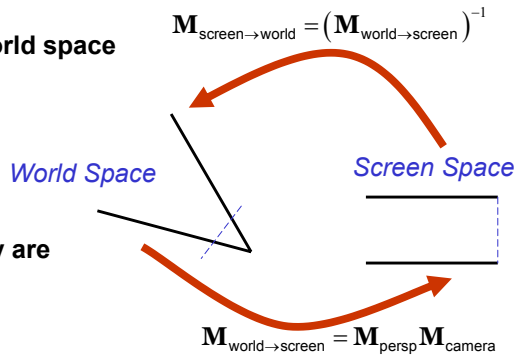
$$\mathbf{q}_w = \mathbf{M}_{\text{screen} \rightarrow \text{world}} \mathbf{q}_s$$

$$\mathbf{p}_w = \mathbf{M}_{\text{camera}}^{-1} \mathbf{p}_c$$

Origin and direction of ray are

$$\mathbf{p} = \mathbf{p}_w$$

$$\mathbf{d} = (\mathbf{q}_w - \mathbf{p}_w) / \|\mathbf{q}_w - \mathbf{p}_w\|$$



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)

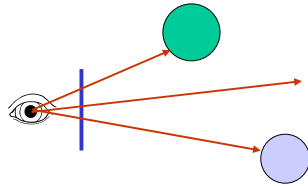
  if( hit_an_object )
    return shade(s, r, closest_t)
  else
    return background_color

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

From Ray Casting to Ray Tracing

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)



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

Adding Shadows

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

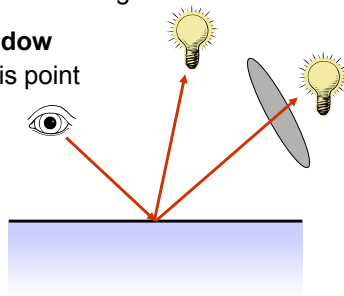
- now we want to shade this point
- for each light, we evaluate our diffuse & specular terms

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

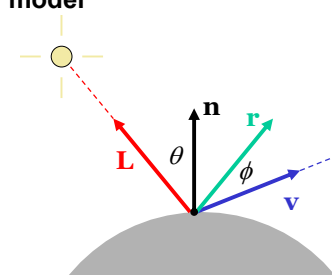


Looking at the Shading Model

Recall our current Phong illumination model

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

- 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**

Computing Correct Reflections

We've talked about two methods to simulate reflections

- environment maps — physically incorrect results
- render mirrored geometry — only works for planar mirrors

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

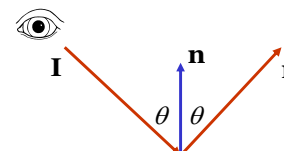
- at a given point, the perfect reflected direction is

$$\mathbf{r} = \mathbf{I} - 2(\mathbf{n} \cdot \mathbf{I})\mathbf{n}$$

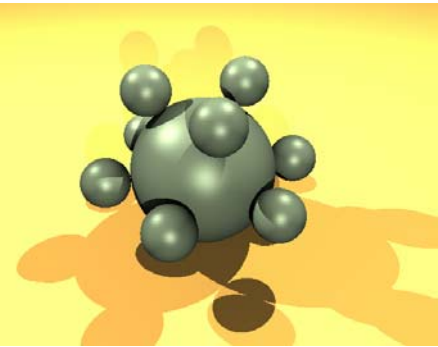
- trace a ray going in that direction
- the returned color is the reflection

Remember to keep in mind

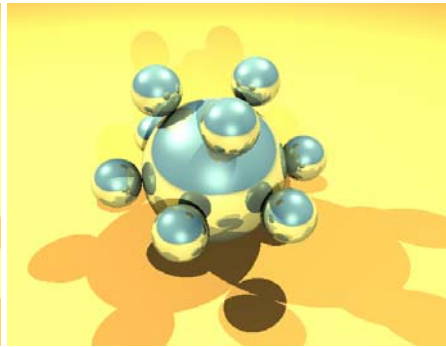
- these arrows are the tracing direction
- light is propagating in opposite direction



An Example: Adding Reflected Rays

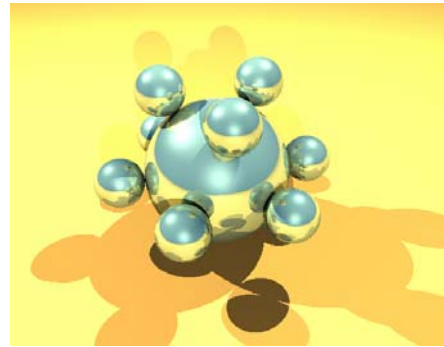


No recursive rays

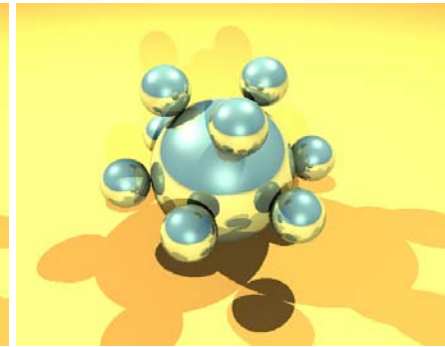


1 Level of recursive reflection

An Example: Adding Reflected Rays



2 Levels of recursive reflection



1 Level of recursive reflection