Ray Tracing Basics

Reading

Required:

- Foley *et al.,* 16.12
- Important: study the textbook before attempting project 3 if you want to save hours of debugging!

Optional (recommended):

- Hearn & Baker, 14.6
- Glassner, chapter 1

Geometric optics

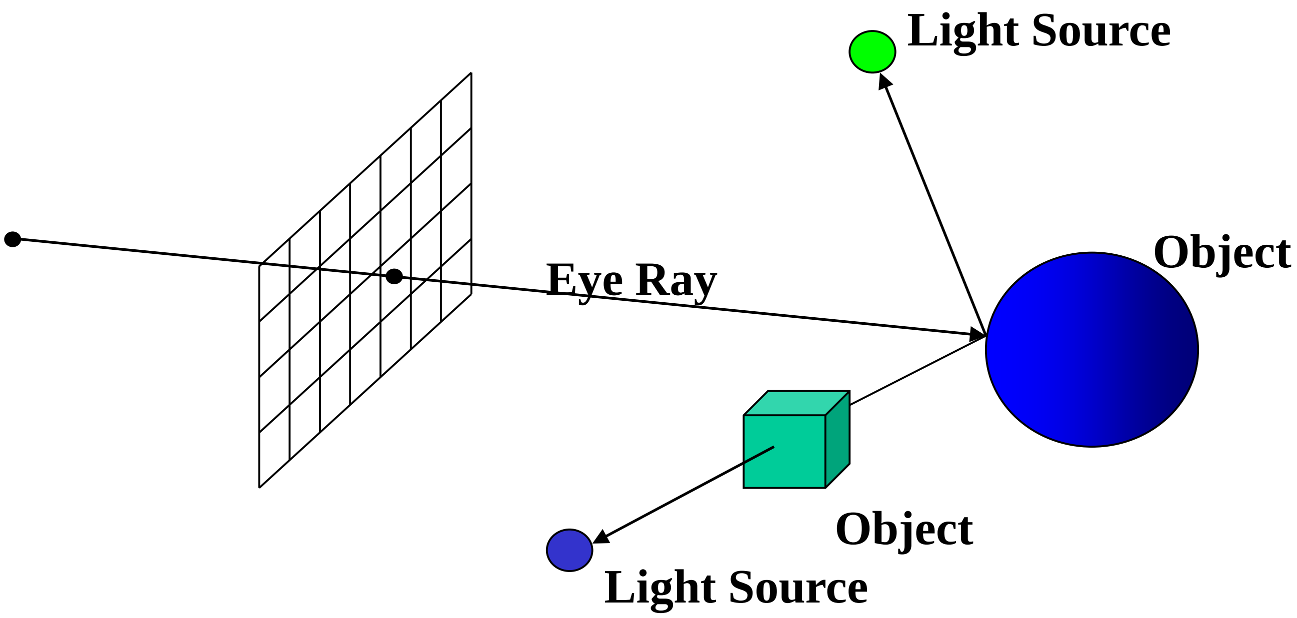
- Modern theories of light treat it as both a wave and a particle.
- We will take a combined and somewhat simpler view of light - the view of **geometric optics**.
- Here are the rules of geometric optics:
  - Light is a flow of photons with wavelengths. We'll call these flows "light rays."
  - Light rays travel in straight lines in free space.
  - Light rays do not interfere with each other as they cross.
  - Light rays obey the laws of reflection and refraction.
  - Light rays travel from the light sources to the eye, but the physics is invariant under path reversal (reciprocity)

Ray Tracing

- A term from optics
- A "physical" simulation of the particle theory of light
- In the 1960s, ray tracing seemed like a great idea, but nobody could do it well enough to beat cheaper image synthesis methods.
- These days, we can follow the simulation deeply enough to get great results!
- But there are some visual phenomena that ray tracing **cannot do.**

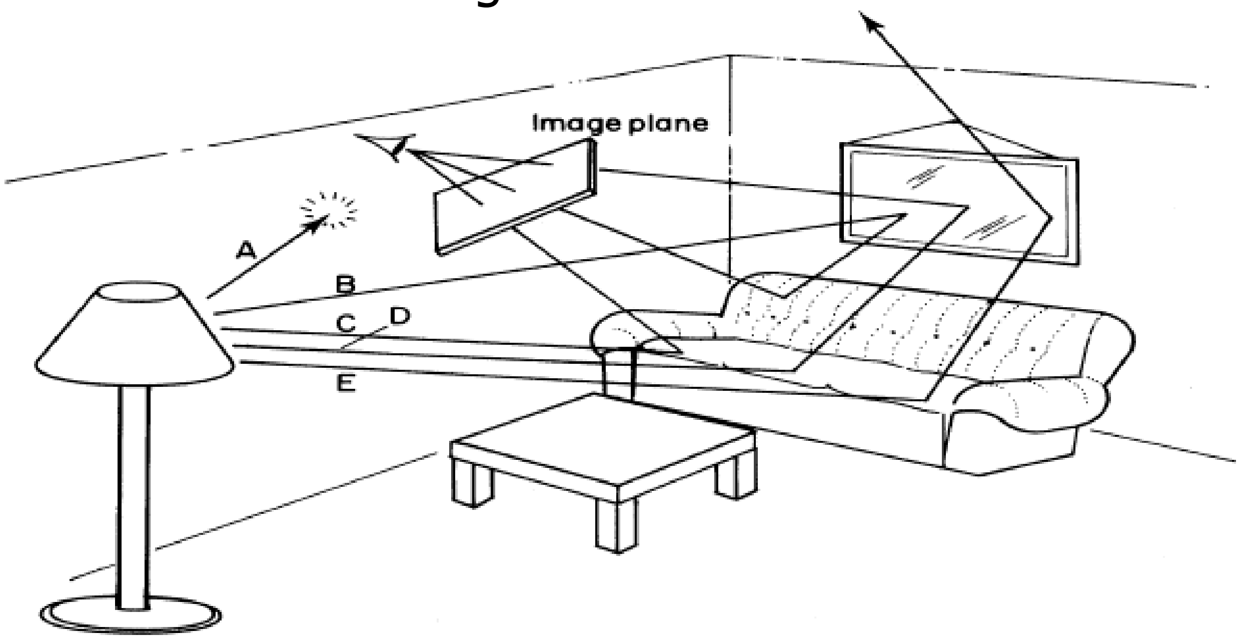
Why Ray Tracing?

- So far, we can do **ray casting**:
  - for each pixel in the projection plane, find the object visible at that pixel and
  - color that pixel according to the object color
- What does this model miss? What if the object is reflective?



Forward Ray Tracing

- Rays emanate from light sources and bounce around in the scene.
- Rays that pass through the projection plane and enter the eye contribute to the final image.



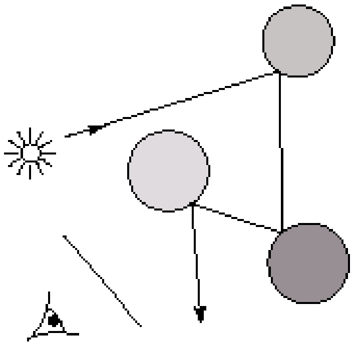
- What's wrong with this method?

Backward Ray Tracing

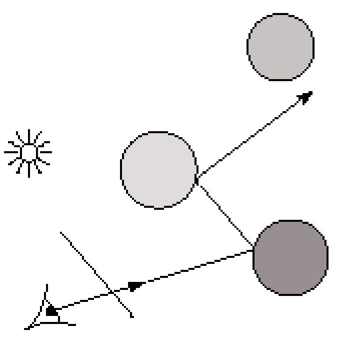
- (Efficiency!) Rather than propagating rays indiscriminately from light sources, we'd like to ask "which rays will definitely contribute to the final image?"
- We can get a good approximation of the answer by firing **rays from the eye**, through the projection plane and into the scene
  - These are the paths that light must have followed to affect the image

Eye vs. light ray tracing

- Where does light begin?
- At the light: light ray tracing (a.k.a. forward ray tracing or photon tracing)



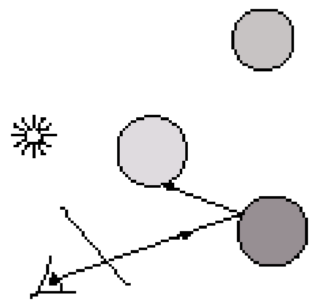
- At the eye: ray tracing (a.k.a. backward ray tracing)



Precursors to ray tracing

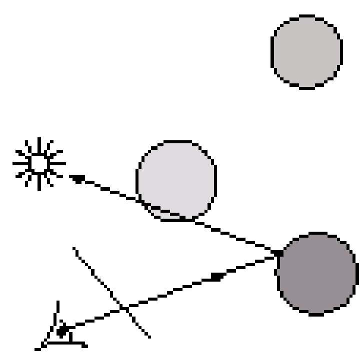
Local illumination

- Cast one eye ray, then shade according to light



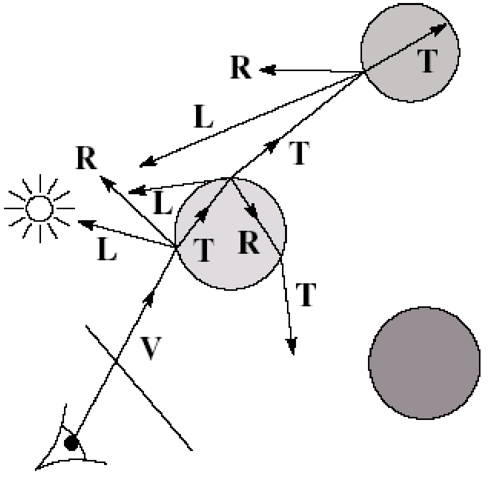
Appel (1968)

- Cast one eye ray + one ray to light



Whitted ray-tracing algorithm

- In 1980, Turner Whitted introduced ray tracing to the graphics community.
  - Combines backward ray tracing + rays to light
  - Recursively traces rays



Algorithm:

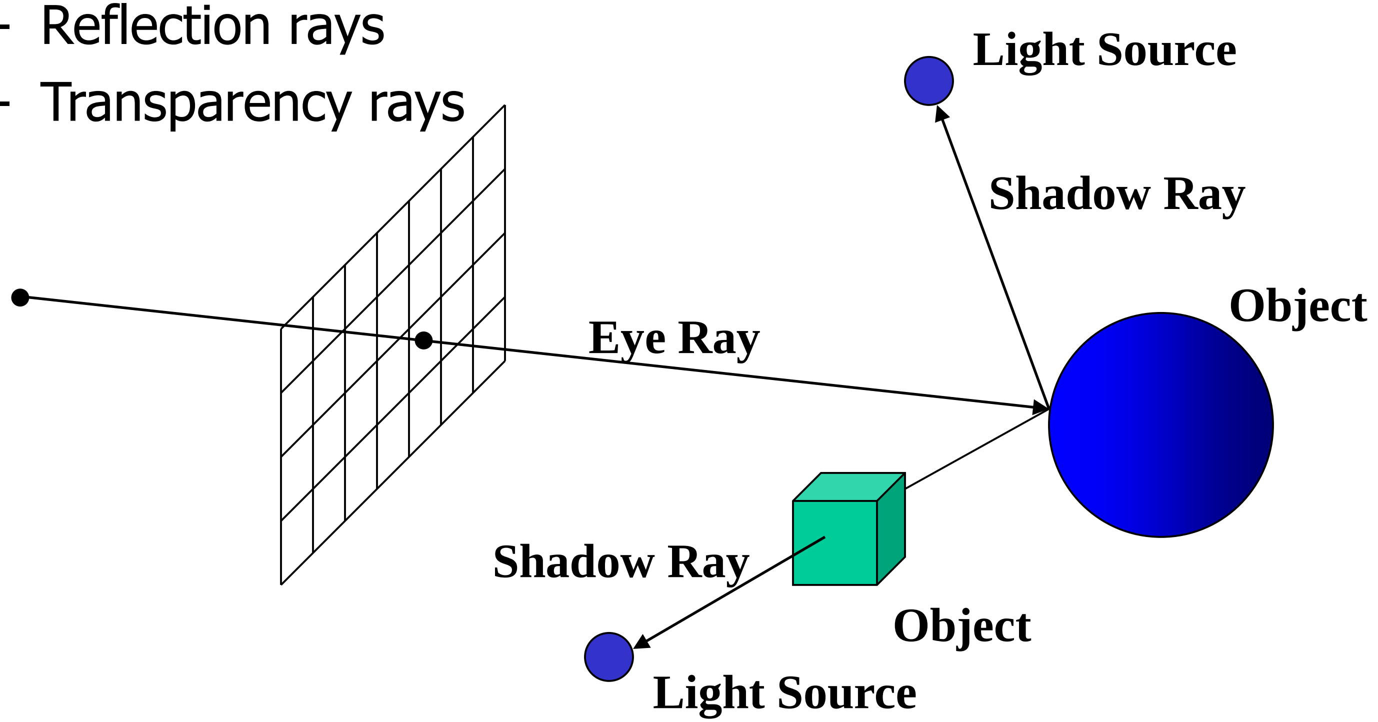
1. For each pixel, trace a **eye** (**primary**) **ray** in direction **V** to the first visible surface (or ray casting).

2. For each intersection, trace **secondary rays**.

- Shadow rays in directions $\mathbf{L}_i$ to light sources.
- Reflected ray in direction **R**.
- Refracted ray or transmitted ray in direction **T**.

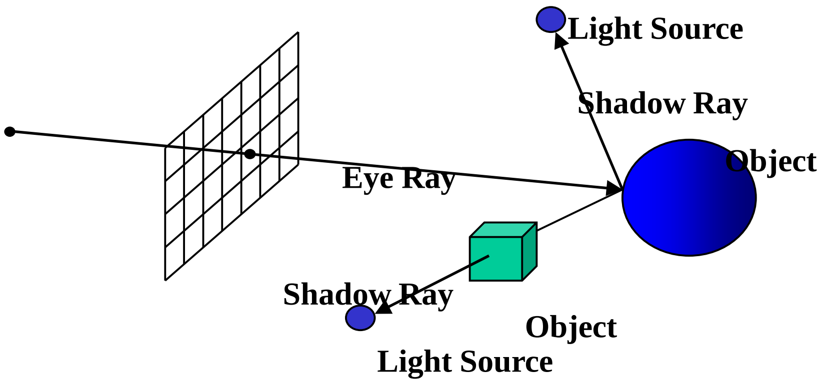
Kinds of (Backward) Rays

- Three kinds of rays
  - Shadow rays
  - Reflection rays
  - Transparency rays



Kinds of (Backward) Rays

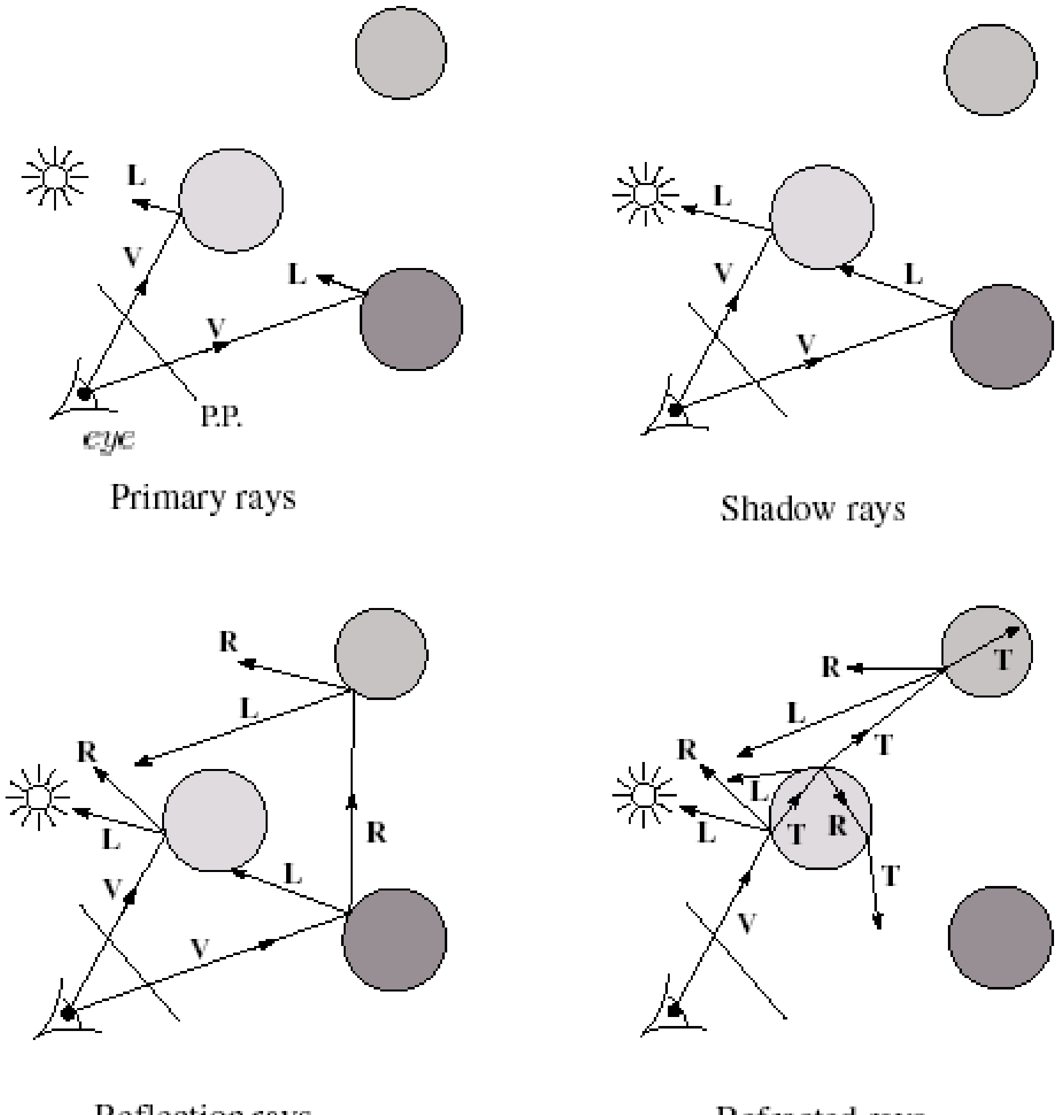
- A ray that leaves the eye and travels out to the scene is called a **primary ray**.
- When a ray hits an object, we spawn three new (backward) rays to collect light that must contribute to the incoming primary ray:
  - **Shadow rays** to light sources, used to attenuate incoming light when applying the shading model
  - **Reflection rays**, which model light bouncing off of other surfaces before hitting this surface
  - **Transparency rays**, which model light refracting through the surface before leaving along the primary ray



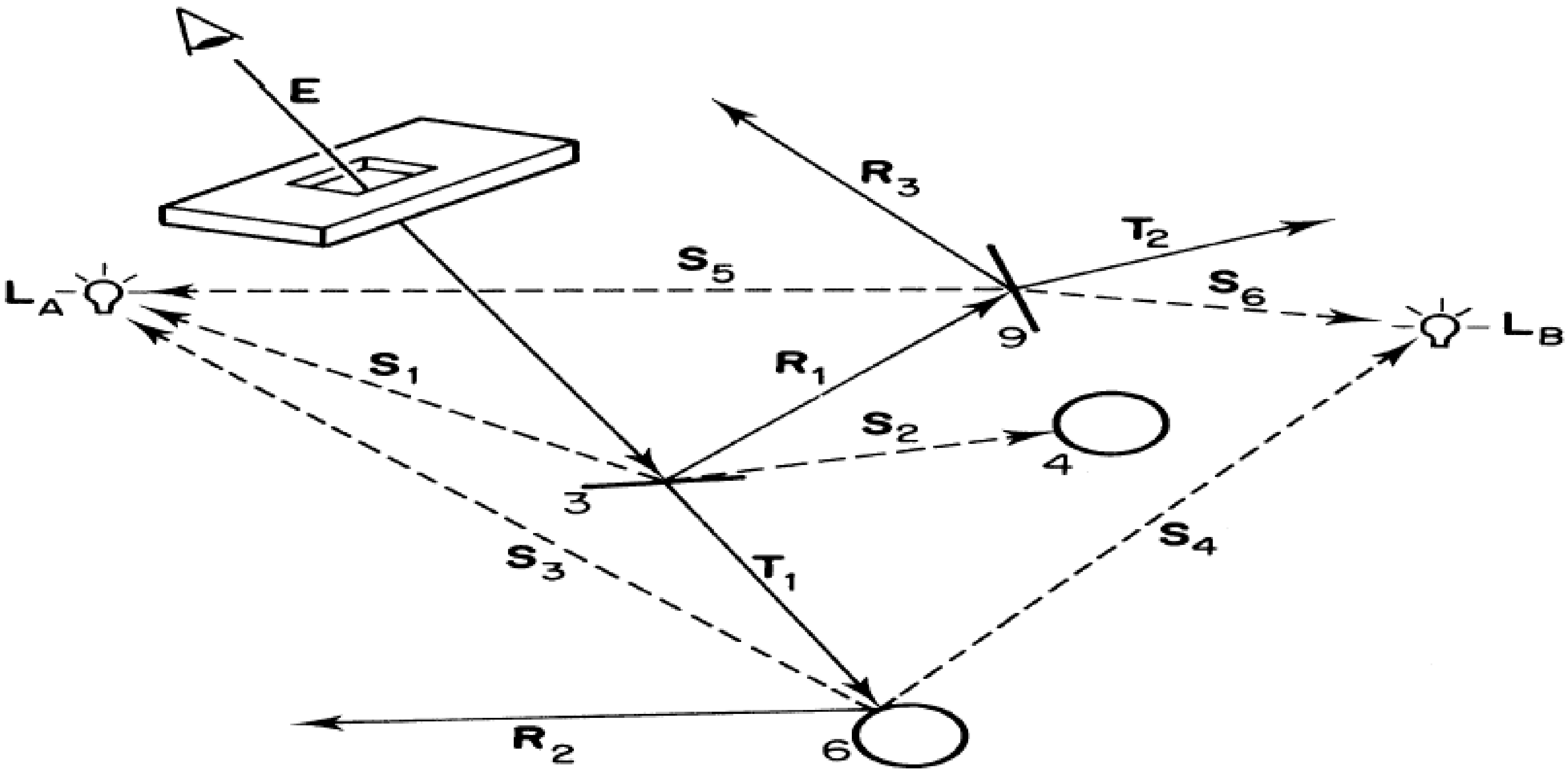
- Shadow rays stop at light sources, but reflection and transparency rays behave just like primary rays!

Whitted algorithm

Let's look at this in stages:

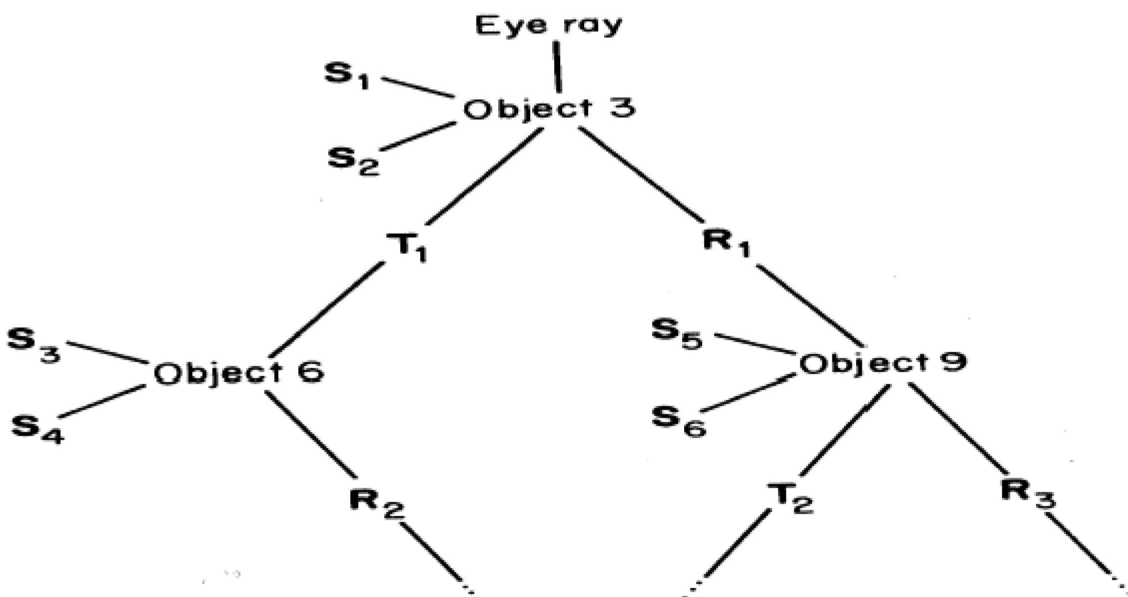


Example of Ray Tracing



Ray Tree

- A primary ray hits a surface and spawns reflection and transparency rays. Those rays may hit surfaces and spawn their own rays, etc.
- We can represent this process schematically using a **ray tree**:



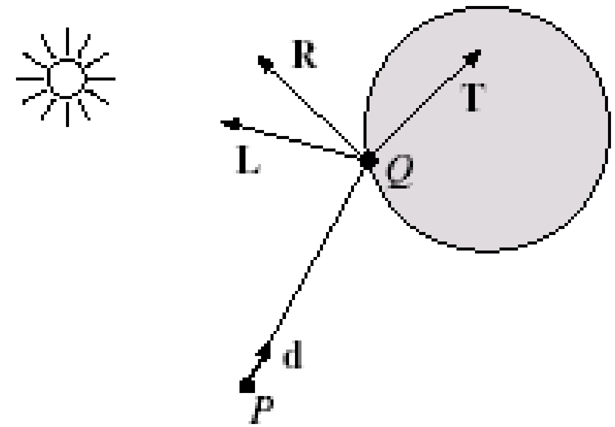
Controlling Tree Depth

- Ideally, we'd spawn child rays at every object intersection forever, getting a "perfect" color for the primary ray.
- In practice, we need heuristics for bounding the depth of the tree (i.e., recursion depth)

Shading

- A ray is defined by an origin **p** and a unit direction **d** and is parameterized by t:

  $$\mathbf{p} + t\,\mathbf{d}$$



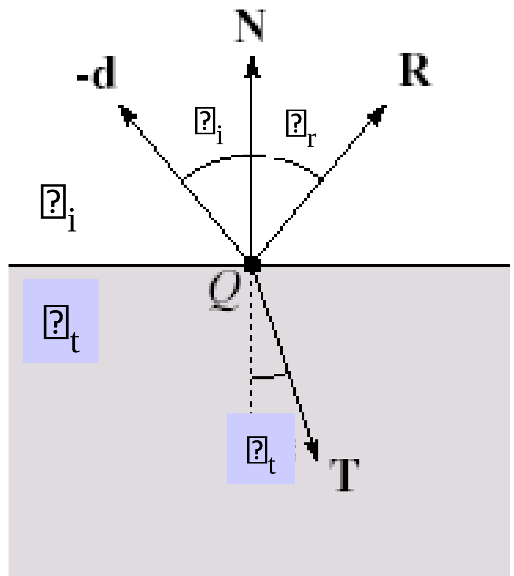
- Let $I(P,\mathbf{d})$ be the intensity seen along that ray. Then:

  $$I(P,\mathbf{d}) = I_{direct} + I_{reflected} + I_{transmitted}$$

  where
  - I_{direct} is computed from the Phong model
  - $I_{reflected} = k_r I(Q,\mathbf{R})$
  - $I_{transmitted} = k_t I(Q,\mathbf{T})$

- Typically, we set $k_r = k_s$ and $k_t = 1 - k_s$ if there is no diffuse reflection.

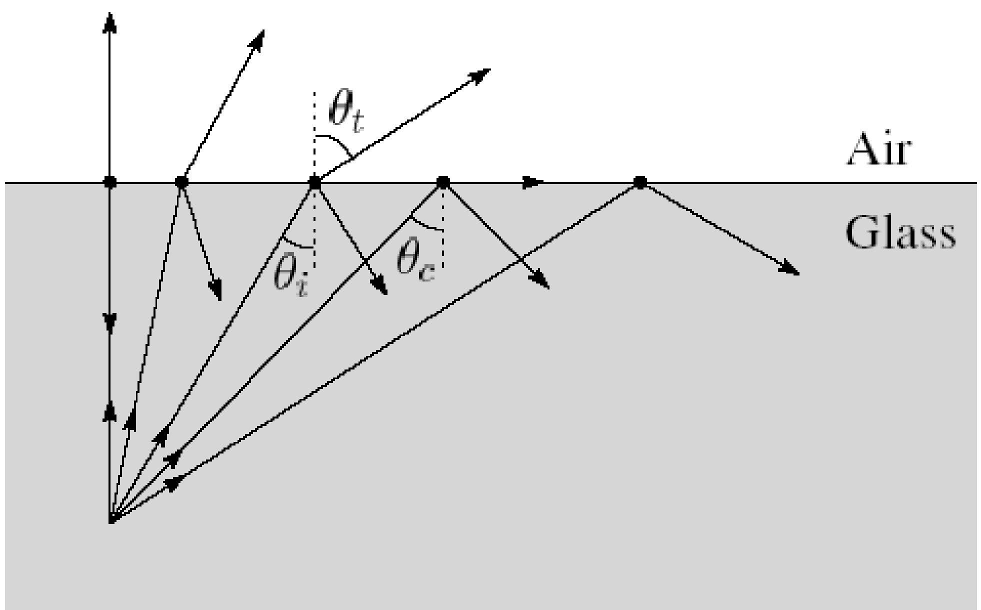
Reflection and transmission



- Law of reflection: $\square_i = \square_r$
- Snell's law of refraction: $\square_i \sin\square_i = \square_t \sin\square_t$
  where $\square_i$ and $\square_t$ are **indices of refraction**.

Total internal reflection

- The equation for the angle of refraction can be computed from Snell's law.
- What happens when $\theta_i > \theta_t$?
- When θ_t is exactly 90 deg, we say that θ_i has achieved "critical angle" $\theta_{c.}$
- For $\theta_i > \theta_{c.}$, no rays are transmitted, and only reflection occurs, a phenomenon known as "total internal reflection" or TIR.



A Recursive Ray Tracer

- Now, put everything together…

Ray tracing pseudocode

- We build a ray traced image by casting rays through each of the pixels:

```
function traceImage (scene):
    for each pixel (i,j) in image
        S = pixelToWorld(i,j)
        P = COP
        d = (S - P)/|| S – P||
        I(i,j) = traceRay(scene, P, d)
    end for
end function
```

Ray tracing pseudocode (cont)

```
function traceRay(scene, P, d):
    (t, N, mtrl) ← scene.intersect (P, d)
    Q ← ray (P, d) evaluated at t
    I = shade(                    )
    R = reflectDirection(         )
    I ← I + mtrl.k_r * traceRay(scene, Q, R)
    if ray is entering object then
        n_i = index_of_air
        n_t = mtrl.index
    else
        n_i = mtrl.index
        n_t = index_of_air
    if (notTIR (                         )) then
        T = refractDirection (                    )
        I ← I + mtrl.k_t * traceRay(scene, Q, T)
    end if
    return I
end function
```

Shade

- Next, we need to calculate the color returned by the *shade* function.

function *shade*(mtrl, scene, Q, $\mathbf{N}$, $\mathbf{d}$):

  I ← mtrl.k_e + mtrl. k_a * scene->I_a

  for each light source ℓ **do**:

    atten = ℓ -> *distanceAttenuation*(   ) *

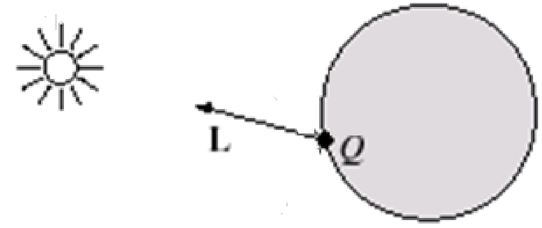
      ℓ -> *shadowAttenuation*(   )

    I ← I + atten*(diffuse term + spec term)

  end for

  return I

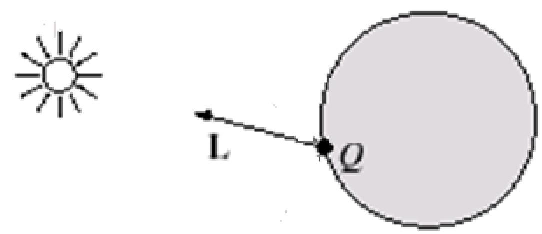
end function



Shadow attenuation

- Computing a shadow can be as simple as checking to see if a ray makes it to the light source.
- For a point light source:

function *PointLight::shadowAttenuation*(scene, *P*)
  d = (ℓ.position - *P*)*.normalize*()
  (t, **N**, mtrl) ← scene.*intersect*(*P*, **d**)
  Q ← ray(t)
  if *Q* is before the light source **then**:
    atten = 0
  else
    atten = 1
  end if
  return atten
end function



- Q: What if there are transparent objects along a path to the light source?

Epsilons!!

- Due to finite precision arithmetic, we do not always get the exact intersection at a surface.
- Q: What kinds of problems might this cause?
- Q: How might we resolve this?

How to resolve this?

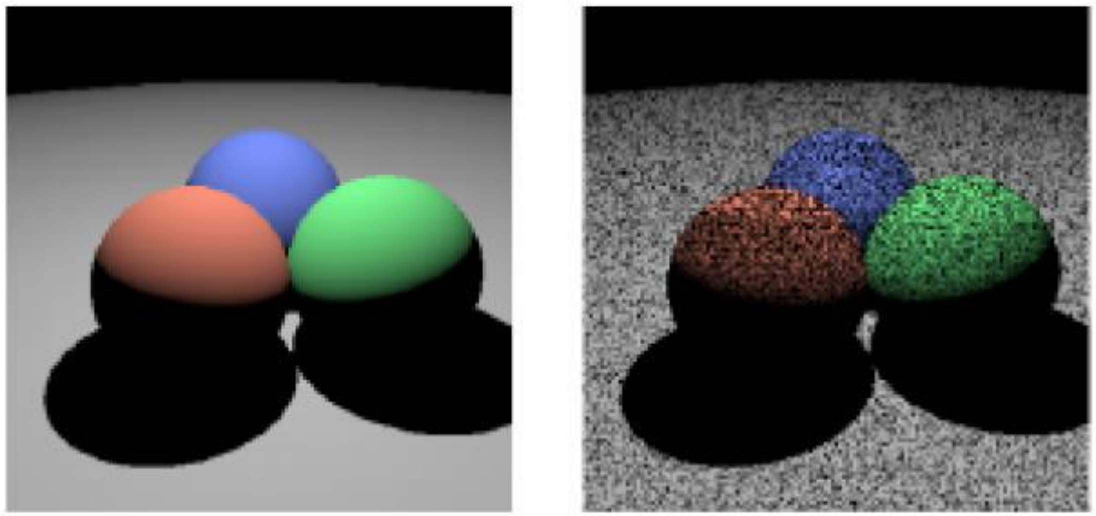
```
#define <math.h>

#define ZERO(a) (fabs(a) < 1e-5)

double t = 0.0;

// for ray
vec3 r = P + d*t;
// if you have done some double precision computation on t
// no shadow attenuation
if (ZERO(t))
 … ..
```

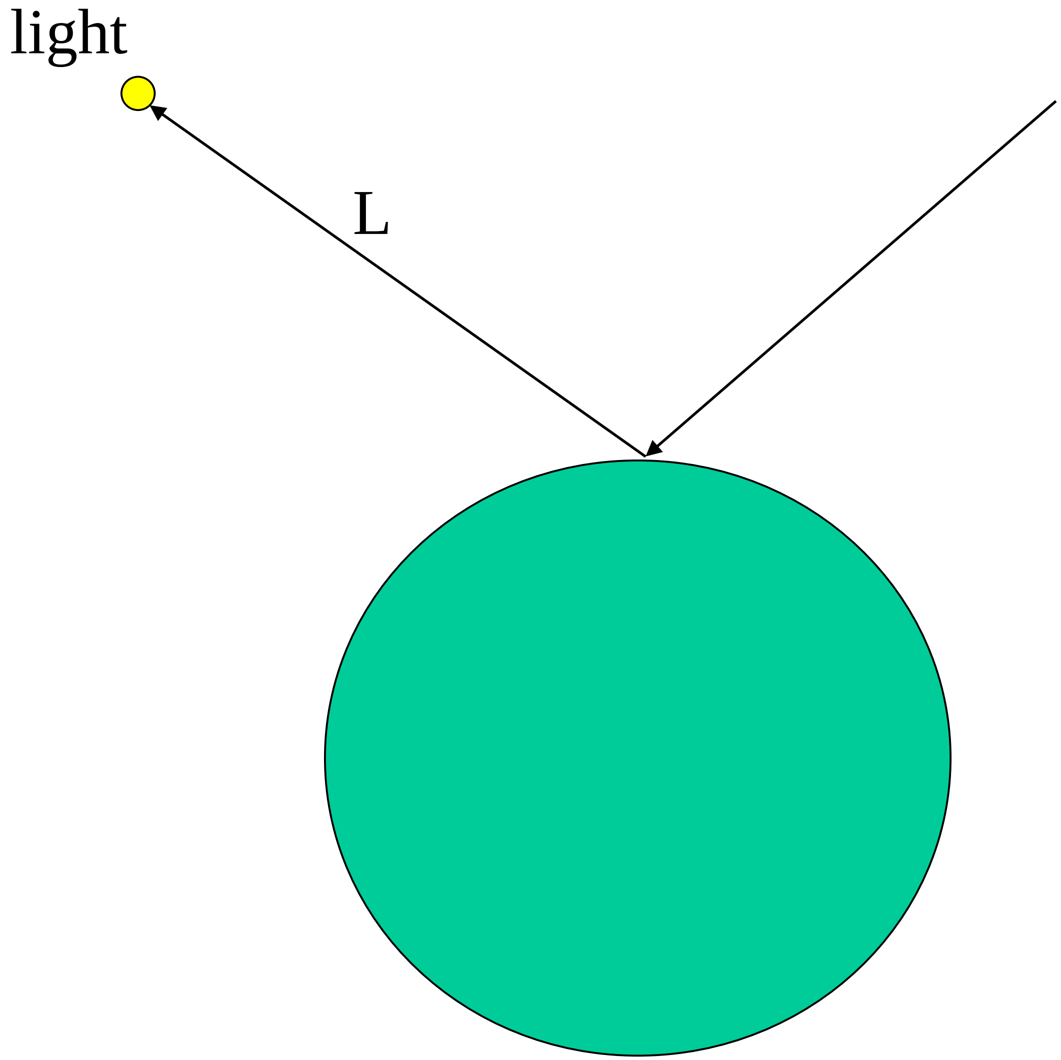
The following shows the correct and incorrect ray-traced result of a sample scene.

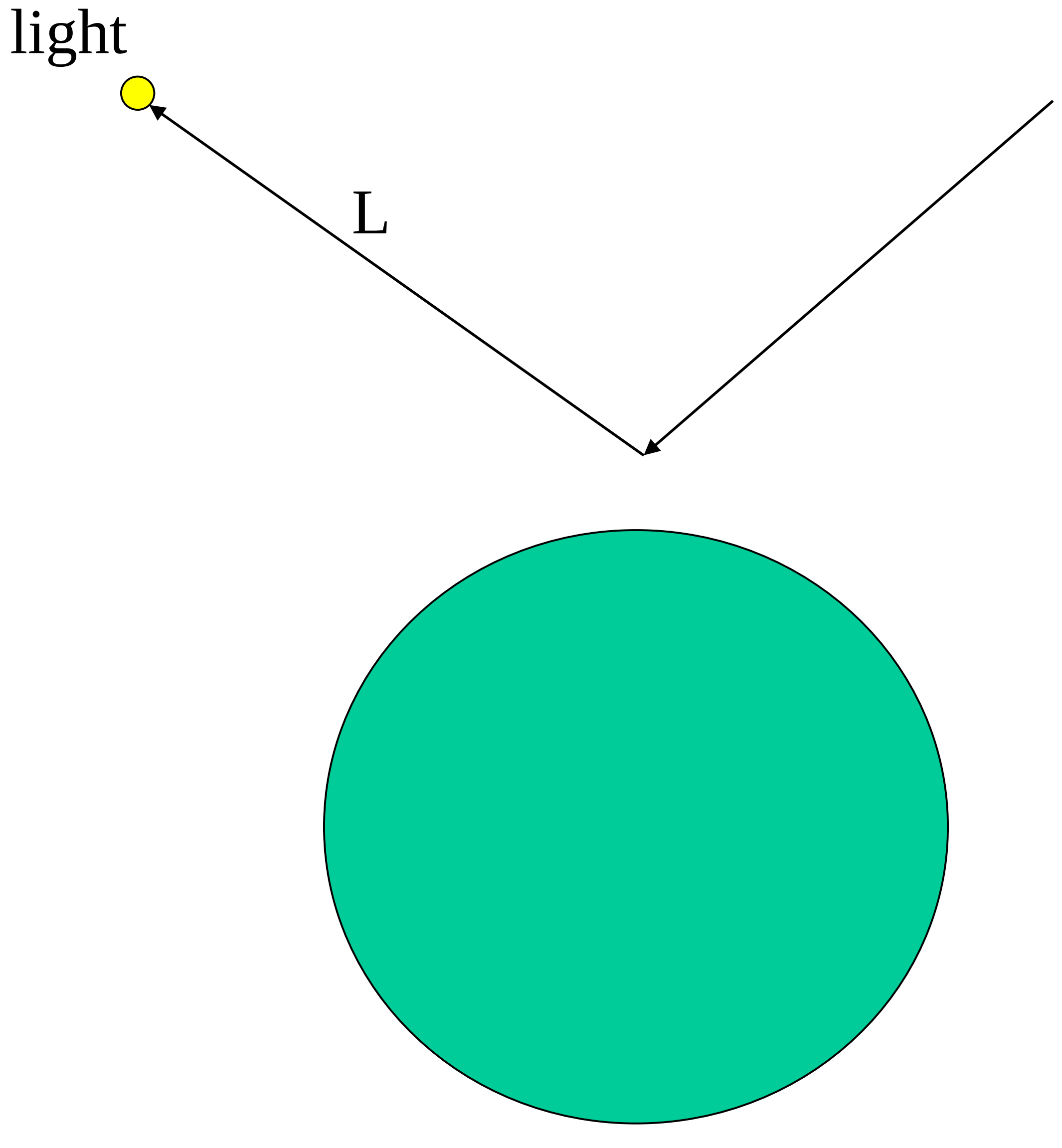


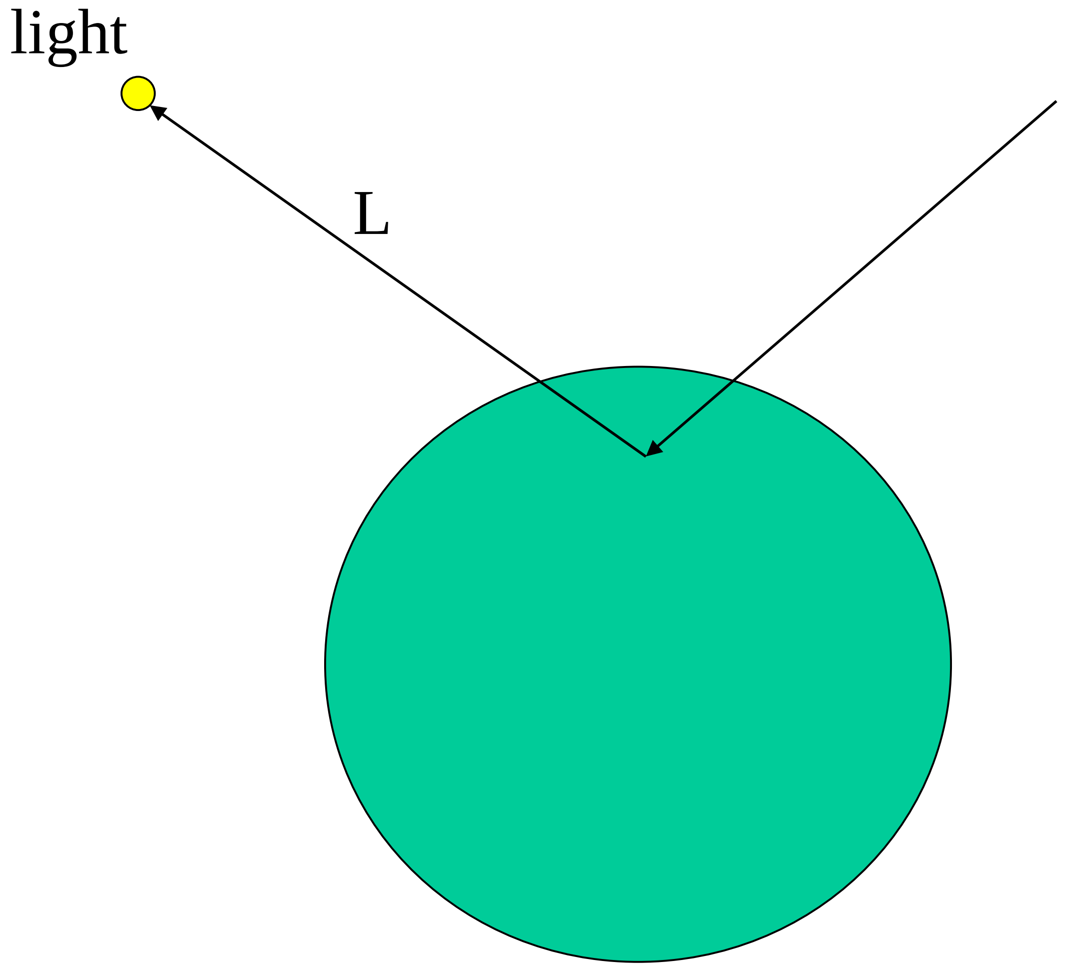
Which of the following are likely bugs?

I. ray generation bug
II. ray intersection bug
III. shading computation bug

(A) I and II
(B) II and III
(C) I and III
(D) II only
(E) III only


light
L


light
L


light
L

Intersecting with xformed geometry

- In general, objects will be placed using transformations. What if object being intersected were transformed by a matrix M?

- Apply M^{-1} to the ray first and intersect in object (local) coordinates!

- The intersected normal is in object (local) coordinates. How do we transform it to world coordinates?

Summary

- Understanding of basic ray tracing concepts
- Forward vs. backward tracing
- Classification of rays
- The ray tree
- Terminating recursion