Ray Tracing

Reading

Required:

- Chapter 10, up to section 10.9
- Section 6.2.2

Further reading:

- T. Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980. [Online]
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989. [In the lab.]
- K. Turkowski, "Properties of Surface Normal Transformations," Graphics Gems, 1990, pp. 539-547.

Geometric optics

Modern theories of light treat it as both a wave and a particle.

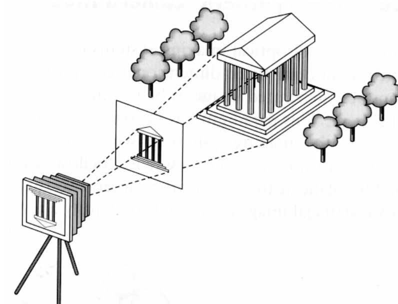
We will take a combined and somewhat simpler view of light – the view of **geometric optics**.

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 form the light sources to the eye, but the physics is invariant under path reversal (reciprocity).

Synthetic pinhole camera

The most common imaging model in graphics is the synthetic pinhole camera: light rays are collected through an infinitesimally small hole and recorded on an **image plane**.



For convenience, the image plane is usually placed in front of the camera, giving a non-inverted

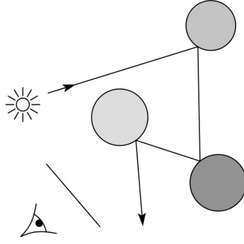
Viewing rays emanate from the **center of projection** (COP) at the center of the lens (or pinhole).

The image of an object point *P* is at the intersection of the viewing ray through *P* and the image plane.

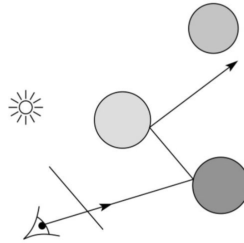
Eye vs. light ray tracing

Where does light begin?

At the light: light ray tracing (a.k.a., forward ray tracing or photon tracing)



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

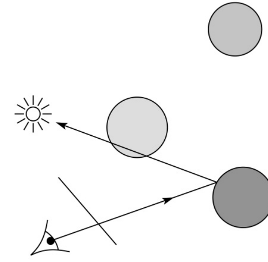


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

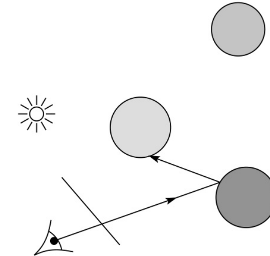
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

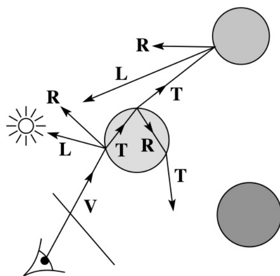
- Cast one eye ray + one ray to light



Whitted ray-tracing algorithm

In 1980, Turner Whitted introduced ray tracing to the graphics community.

- Combines eye ray tracing + rays to light
- Recursively traces rays



Algorithm:

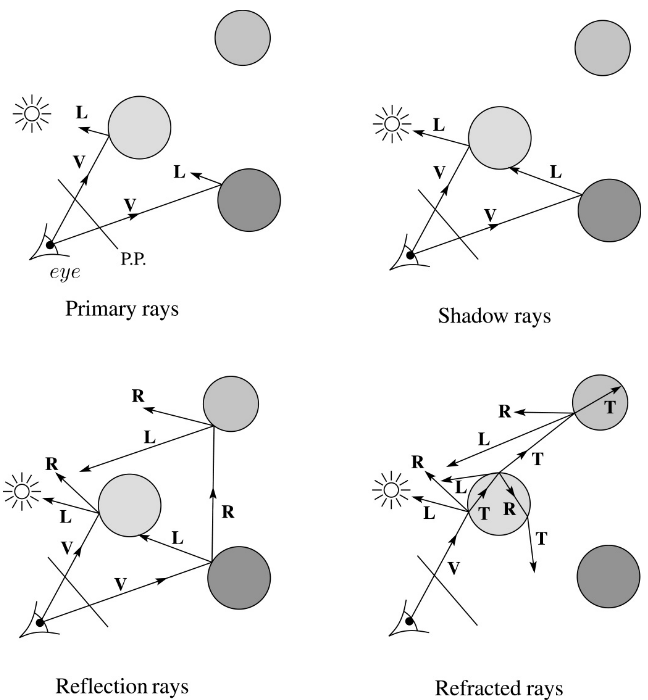
1. For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

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**.

Whitted algorithm (cont'd)

Let's look at this in stages:



Primary rays

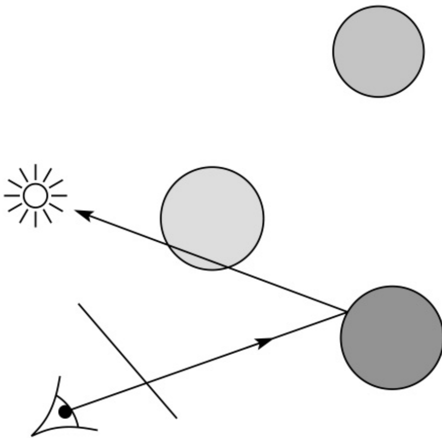
Shadow rays

Reflection rays

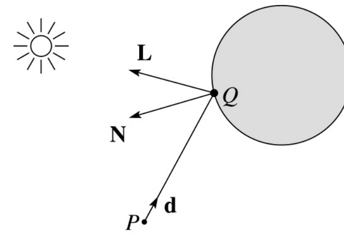
Refracted rays

Ray casting and local illumination

Now let's actually build the ray tracer in stages. We'll start with ray casting and local illumination:



Direct illumination



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

$$P + t\mathbf{d}$$

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

$$I(P, \mathbf{d}) = I_{\text{direct}}$$

where

- I_{direct} is computed from the Phong model

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
        A **=** *pixelToWorld*(i,j)
        P = **COP**
        d = (*A* - *P*)/|| *A* – *P*||
        I(i,j) = *traceRay*(scene, *P*, **d**)
    end for
end function

function *traceRay*(scene, *P*, **d**):
    (t, **N**, mtrl) ← scene.*intersect* (*P*, **d**)
    Q ← ray (*P*, **d**) evaluated at t
    I = *shade*(   )
    return I
end function

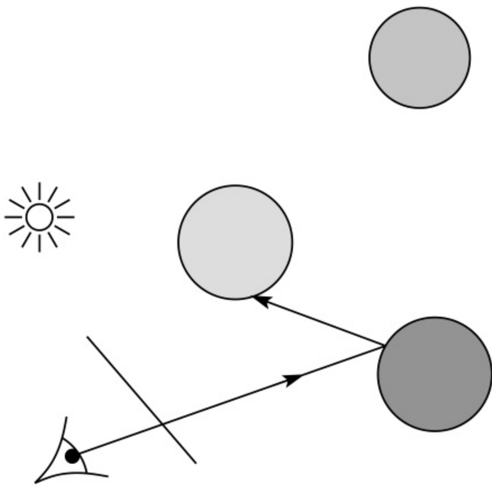
Shading pseudocode

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

function *shade*(mtrl, scene, *Q*, **N**, **d**):
    I ← mtrl.k_e + mtrl. k_a * I_a
    for each light source L **do**:
        atten = L -> *distanceAttenuation*(   )
        I ← I + atten*(diffuse term + specular term)
    end for
    return I
end function

Ray casting with shadows

Now we'll add shadows by casting shadow rays:



Shading with shadows

To include shadows, we need to modify the shade function:

function *shade*(mtrl, scene, *Q*, **N**, **d**):

   I ← mtrl.k_e + mtrl. k_a * I_a

   for each light source L **do**:

      atten = L -> *distanceAttenuation*(*Q*) *

         L -> *shadowAttenuation*(   )

      I ← I + atten*(diffuse term + specular 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 = (this.position - *P***)**.*normalize*()

   (t, **N**, mtrl) ← scene.*intersect*(*P*, **d**)

   Compute t_{light}

   if ($t < t_{light}$) **then**:

      atten = 0

   else

      atten = 1

   end if

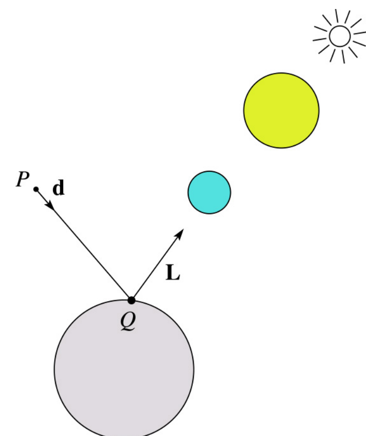
   return atten

end function

Shadow attenuation (cont'd)

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

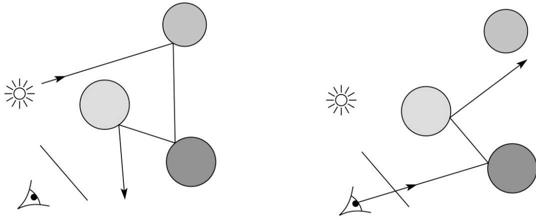
[Suppose for simplicity that each object has a multiplicative transparency constant, k_t.]



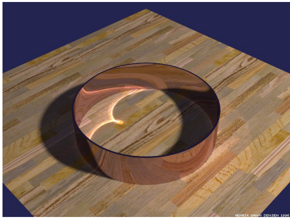
[See Shirley's Section 10.6 for discussion of "Beer's Law" for more realistic attenuation.]

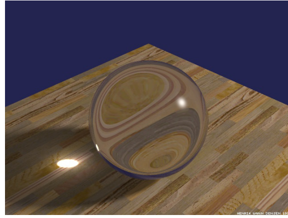
Photon mapping

Combine light ray tracing (photon tracing) and eye ray tracing:



…to get **photon mapping**.





Renderings by Henrik Wann Jensen: http://graphics.ucsd.edu/~henrik/images/caustics.html

Shading in "Trace"

The Trace project uses a version of the Phong shading equation we derived in class, with two modifications:

- Shadow attenuation is clamped to be at least 1:

$$A_j^{dist} = \min\left\{1, \frac{1}{a_j + b_j d_j + c_j d_j^2}\right\}$$

- Shadow attenuation A^{shadow} is included.

Here's what it should look like:

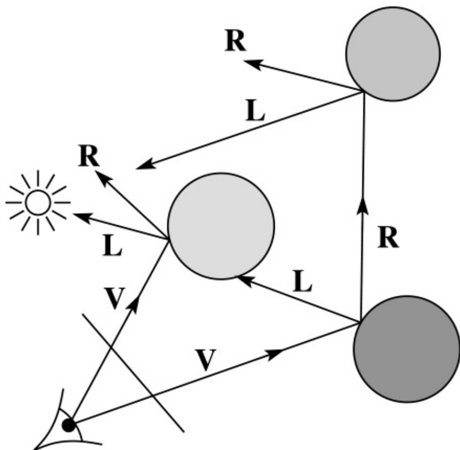
$$I = k_e + k_a L_a + \sum_j A_j^{shadow} A_j^{dist} L_j \left[k_d (\mathbf{N} \cdot \mathbf{L}_j)_+ + k_s (\mathbf{V} \cdot \mathbf{R}_j)_+^{n_s} \right]$$

I.e., we are not using the OpenGL shading equation, which is somewhat different.

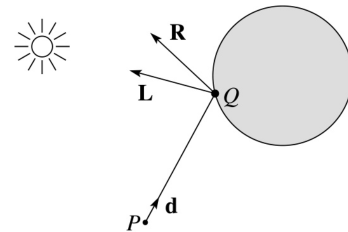
Note: the "R" here is the reflection of the *light* about the surface normal.

Recursive ray tracing with reflection

Now we'll add reflection:



Shading with reflection



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

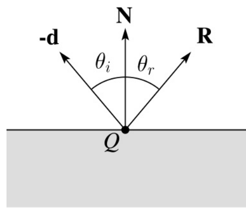
$$I(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}}$$

where

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

Typically, we set $k_r = k_s$.

Reflection



Law of reflection:

$$\theta_i = \theta_r$$

R is co-planar with **d** and **N**.

Ray-tracing pseudocode, revisited

function *traceRay*(scene, *P*, **d**):
    (t, **N**, mtrl) ← scene.*intersect* (*P*, **d**)
    Q ← ray (*P*, **d**) evaluated at t
    I = *shade*(scene, mtrl, *P*, **N**, -**d**)
    R = *reflectDirection*(   )
    I ← I + mtrl.k_r ∗ *traceRay*(scene, *Q*, **R**)
    return I
end function

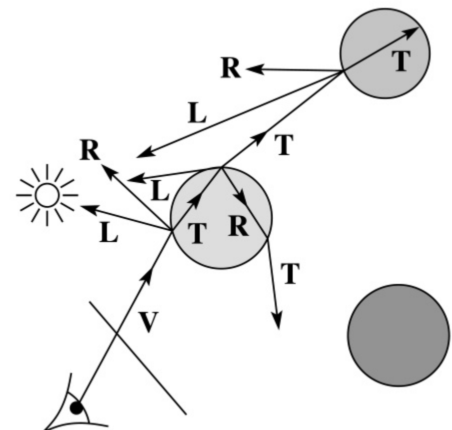
Terminating recursion

Q: How do you bottom out of recursive ray tracing?

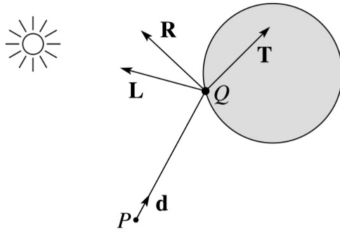
Possibilities:

Whitted ray tracing

Finally, we'll add refraction, giving us the Whitted ray tracing model:



Shading with reflection and refraction



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

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

where

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

Typically, we set $k_r = k_s$ and $k_t = 1 - k_s$ (or 0, if opaque).

[Generally, k_r and k_t are determined by "Fresnel reflection," which depends on angle of incidence and changes the polarization of the light. Shirley discusses an approximation in Section 10.6.]

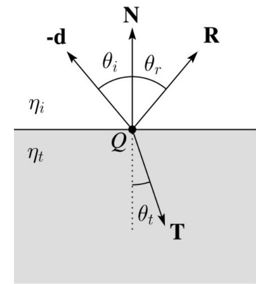
Refraction

Snell's law of refraction:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

where η_i, η_t are **indices of refraction**.

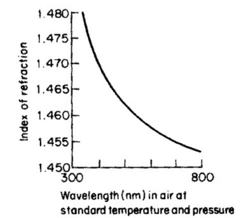
In all cases, **R** and **T** are co-planar with **d** and **N**.



The index of refraction is material dependent.

It can also vary with wavelength, an effect called **dispersion** that explains the colorful light rainbows from prisms. (We will generally assume no dispersion.)

Medium	Index of refraction
Vaccum	1
Air	1.0003
Water	1.33
Fused quartz	1.46
Glass, crown	1.52
Glass, dense flint	1.66
Diamond	2.42



Index of refraction variation for fused quartz

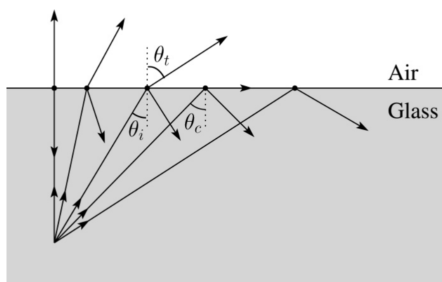
Total Internal Reflection

The equation for the angle of refraction can be computed from Snell's law:

What happens when $\eta_i > \eta_t$?

When θ_t is exactly 90°, we say that θ_I has achieved the "critical angle" θ_c.

For $\theta_I > \theta_c$, *no rays are transmitted*, and only reflection occurs, a phenomenon known as "total internal reflection" or TIR.



Ray-tracing pseudocode, revisited

```
function traceRay(scene, P, d):
    (t, N, mtrl) ← scene.intersect (P, d)
    Q ← ray (P, d) evaluated at t
    I = shade(scene, mtrl, P, N, -d)
    R = reflectDirection(N, -d)
    I ← I + mtrl.kr ∗ 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.kt ∗ traceRay(scene, Q, T)
    end if
    return I
end function
```

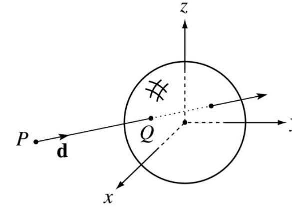
Terminating recursion, incl. refraction

Q: *Now* how do you bottom out of recursive ray tracing?

Intersecting rays with spheres

Now we've done everything except figure out what that "scene.*intersect*(*P*, **d**)" function does.

Mostly, it calls each object to find out the t value at which the ray intersects the object. Let's start with intersecting spheres…



Given:

- The coordinates of a point along a ray passing through *P* in the direction **d** are:

$$x = P_x + td_x$$
$$y = P_y + td_y$$
$$z = P_z + td_z$$

- A unit sphere *S* centered at the origin defined by the equation:

Find: The *t* at which the ray intersects *S*.

Intersecting rays with spheres

Solution by substitution:

$$x^2 + y^2 + z^2 - 1 = 0$$
$$(P_x + td_x)^2 + (P_y + td_y)^2 + (P_z + td_z)^2 - 1 = 0$$
$$at^2 + bt + c = 0$$

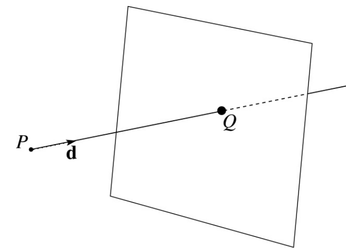
where

$$a = d_x^2 + d_y^2 + d_z^2$$
$$b = 2(P_x d_x + P_y d_y + P_z d_z)$$
$$c = P_x^2 + P_y^2 + P_z^2 - 1$$

Q: What are the solutions of the quadratic equation in *t* and what do they mean?

Q: What is the normal to the sphere at a point (*x,y,z*) on the sphere?

Ray-plane intersection



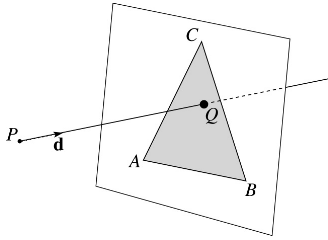
We can write the equation of a plane as:

$$ax + by + cz = d$$

The coefficients *a*, *b*, and *c* form a vector that is normal to the plane, **n** = [*a b c*]^T. Thus, we can re-write the plane equation as:

We can solve for the intersection parameter (and thus the point):

Ray-triangle intersection



To intersect with a triangle, we first solve for the equation of its supporting plane.

How might we compute the (un-normalized) normal?

Given this normal, how would we compute *d*?

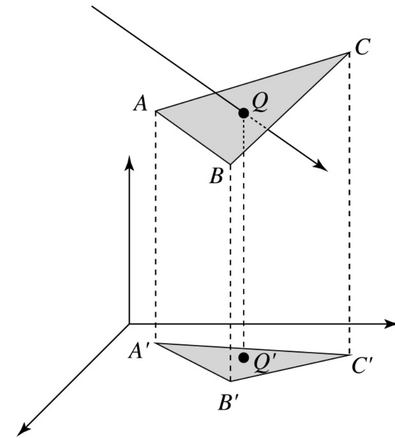
Using these coefficients, we can solve for *Q*. Now, we need to decide if *Q* is inside or outside of the triangle.

Solution 1: compute barycentric coordinates from 3D points.

What do you do with the barycentric coordinates?

Ray-triangle intersection

Solution 2: project down a dimension and compute barycentric coordinates from 2D points.



Why is solution 2 possible? Why is it legal? Why is it desirable? Which axis should you "project away"?

Interpolating vertex properties

The barycentric coordinates can also be used to interpolate vertex properties such as:

- material properties
- texture coordinates
- normals

For example:

$$k_d(Q) = \alpha k_d(A) + \beta k_d(B) + \gamma k_d(C)$$

Interpolating normals, known as Phong interpolation, gives triangle meshes a smooth shading appearance. (Note: don't forget to normalize interpolated normals.)

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?

Intersecting with xformed geometry

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

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

Intersecting with xformed geometry

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