Ray Tracing

CSE 457
Winter 2014

Reading

Required:

- Shirley, section 10.1-10.7 (online handout)
- Triangle intersection (online handout)

Further reading:

- Shirley errata on syllabus page, needed if you work from his book instead of the handout, which has already been corrected.
- T. Whitted. An improved illumination model for shaded display. Communications of the ACM 23(6), 343-349, 1980.
- A. Glassner. An Introduction to Ray Tracing. Academic Press, 1989.
- 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**.

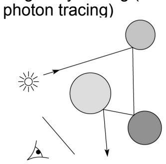
Here are the rules of geometric optics:

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

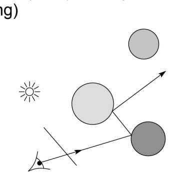
Eye vs. light ray tracing

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)

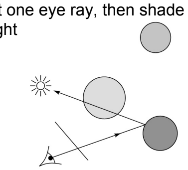


We will generally follow rays from the eye into the scene.

Precursors to ray tracing

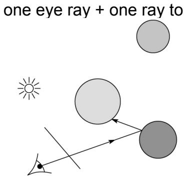
Local illumination

- Cast one eye ray, then shade according to light



Appel (1968)

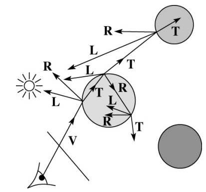
- Cast one eye ray + one ray to light



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:

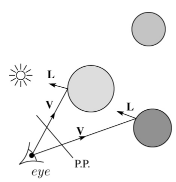
1. For each pixel, trace a **primary ray** in direction **V** to the first visible surface.

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

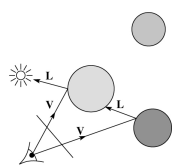
- **Shadow rays** in directions $\mathbf{L}_i$ to light sources
- **Reflected ray** in direction **R**.
- **Refracted ray** or **transmitted ray** in direction **T**.

Whitted algorithm (cont'd)

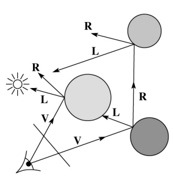
Let's look at this in stages:



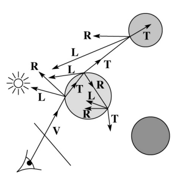
Primary rays



Shadow rays



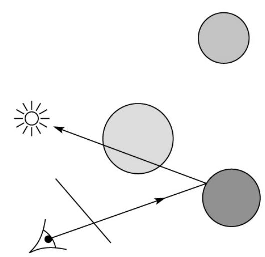
Reflection rays



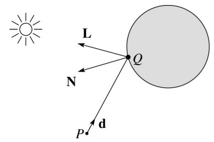
Refracted rays

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 $\mathbf{d}$ and is parameterized by $t > 0$:

$$\mathbf{r}(t) = P + t\mathbf{d}$$

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 Blinn-Phong model

Shading in "Trace"

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

- Distance attenuation is clamped to be at most 1:

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

- Shadow attenuation A^{shadow} is included.

Here's what it should look like:

$$I = k_e + k_a I_{La} +$$
$$\sum_j A_j^{shadow} A_j^{dist} I_{L,j} B_j \left[k_d \left(\mathbf{N} \cdot \mathbf{L}_j \right) + k_s \left(\mathbf{N} \cdot \mathbf{H}_j \right)^{n_s} \right]$$

This is the shading equation to use in the Trace project!

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

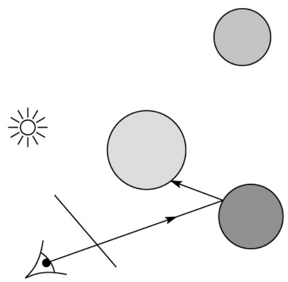
Shading pseudocode

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_{La}
  for each light source Light **do**:
    atten = Light -> *distanceAttenuation*(   )
    L = Light -> getDirection (   )
    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_{La}

for each light source Light **do**:

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

Light -> *shadowAttenuation*( )

L = Light -> getDirection (Q)

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 = getDirection(P)

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

Compute t_{light}

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

atten = (0, 0, 0)

else

atten = (1, 1, 1)

end if

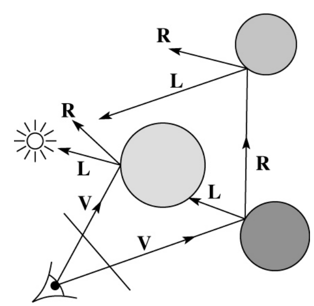
return atten

end function

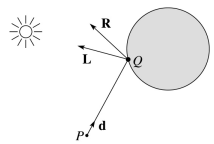
Note: we will later handle color-filtered shadowing, so this function needs to return a *color* value.

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:

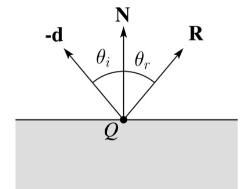
$$I(P, \mathbf{d}) = I_{\text{direct}} + I_{\text{reflected}}$$

where

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

Typically, we set $k_r = k_s$. (k_r is a color value.)

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, Q, N, -d)
        R = reflectDirection(                    )
        I ← I + mtrl.k_r * traceRay(scene, Q, R)
        return I
end function
```

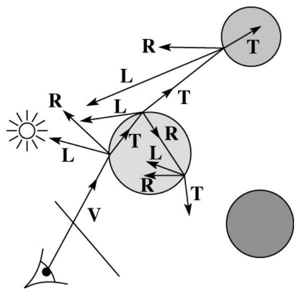

Terminating recursion

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

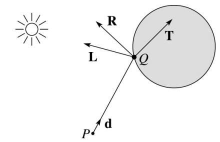
Possibilities:

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 Blinn-Phong model, plus shadow attenuation
- $I_{\text{reflected}} = k_r\, I\,(Q, \mathbf{R})$
- $I_{\text{transmitted}} = k_t I\,(Q, \mathbf{T})$

Typically, we set $k_r = k_s$ and $k_t = 1 - k_s$ (or (0,0,0), if opaque, where k_t is a color value).

[Generally, k_r and k_t are determined by "Fresnel reflection," which depends on angle of incidence and changes the polarization of the light. This is discussed in Shirley's textbook and can be implemented for extra credit.]

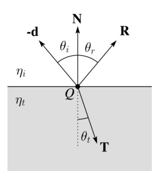
Refraction

Snell's law of refraction:

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

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

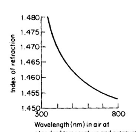
In all cases, **R** and **T** are co-planar with **d** and **N**.



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

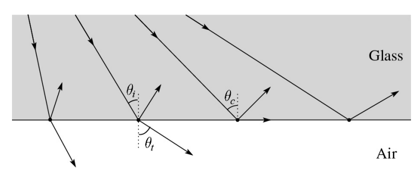
Total Internal Reflection

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.



Shirley handout

Shirley uses different symbols. Here is the translation between them:

r = R

t = T

$\phi = \theta_t$

$\theta = \theta_r = \theta_i$

$n = \eta_i$

$n_t = \eta_t$

Also, Shirley has two important errors that have already been corrected in the handout.

But, if you're consulting the original text, be sure to refer to the errata posted on the syllabus and on the project page for corrections.

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, *Q*, **N**, -**d**)
- **R** = *reflectDirection*(**N**, -**d**)
- 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

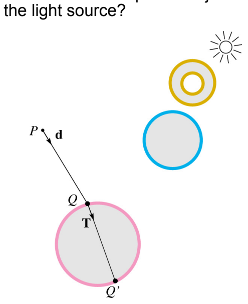
end function

Terminating recursion, incl. refraction

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

Shadow attenuation (cont'd)

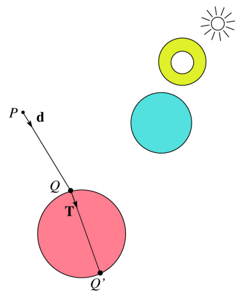
Q: What if there are transparent objects along a path to the light source?



We'll take the view that the color is really only at the surface, like a glass object with a colored transparency coating on it. In this case, we multiply in the transparency constant, k_t, every time an object is entered or exited, possibly more than once for the same object.

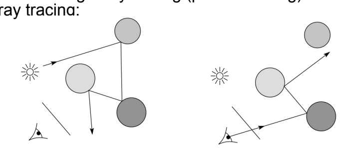
Shadow attenuation (cont'd)

Another model would be to treat the glass as solidly colored in the interior. Shirley's textbook describes a the resulting volumetric attenuation based on Beer's Law, which you can implement for extra credit.

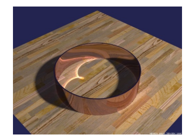


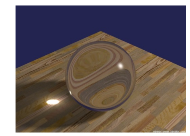
Photon mapping

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

Normals and shading when inside

When a ray is inside an object and intersects the object's surface on the way out, the normal will be pointing ***away*** from the ray (i.e., the normal always points to the outside by default).

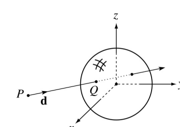
You must ***negate*** the normal before doing any of the shading, reflection, and refraction that follows.

Finally, when shading a point inside of an object, apply k_t to the ambient component, since that "ambient light" had to pass through the object to get there in the first place.

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:

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

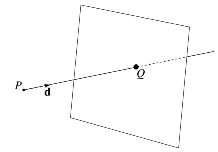
where

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

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



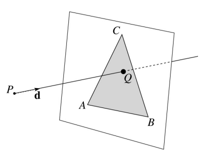
We can write the equation of a plane as:

$$ax + by + cz = k$$

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.

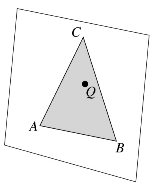
How might we compute the (un-normalized) normal?

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

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

3D inside-outside test

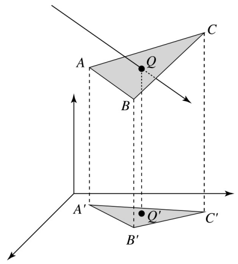
One way to do this “inside-outside test,” is to see if *Q* lies on the left side of each edge as we move counterclockwise around the triangle.



How might we use cross products to do this?

2D inside-outside test

Without loss of generality, we can perform this same test after projecting down a dimension:



If Q' is inside of $A'B'C'$, then Q is inside of ABC.

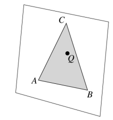
Why is this projection desirable?

Which axis should you “project away”?

Barycentric coordinates

As we’ll see in a moment, it is often useful to represent Q as an **affine combination** of A, B, and C:

$$Q = \alpha A + \beta B + \gamma C$$



where:

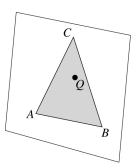
$$\alpha + \beta + \gamma = 1$$

We call α, β, and γ, the **barycentric coordinates** of Q with respect to A, B, and C.

Computing barycentric coordinates

Given a point Q that is inside of triangle ABC, we can solve for Q’s barycentric coordinates in a simple way:

$$\alpha = \frac{\text{Area}(QBC)}{\text{Area}(ABC)} \quad \beta = \frac{\text{Area}(AQC)}{\text{Area}(ABC)} \quad \gamma = \frac{\text{Area}(ABQ)}{\text{Area}(ABC)}$$



How can cross products help here?

In the end, these calculations can be performed in the 2D projection as well!

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?

Summary

What to take home from this lecture:

- The meanings of all the boldfaced terms.
- Enough to implement basic recursive ray tracing.
- How reflection and transmission directions are computed.
- How ray-object intersection tests are performed on spheres, planes, and triangles
- How barycentric coordinates within triangles are computed
- How ray epsilons are used.