Ray Tracing

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.

Reading

Required:
- Shirley, section 10.1-10.7 (handout)
- Triangle intersection handout

Further reading:
- Shirley errata on syllabus page, needed if you work from his book instead of the handout, which has already been corrected.
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 1968, 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 $\mathbf{V}$ 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 $\mathbf{R}$
   - Refracted ray or transmitted ray in direction $\mathbf{T}$

Whitted algorithm (cont’d)

Let’s look at this in stages:

Primary rays

Shadow rays

Reflection rays

Reflected 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 \( t > 0 \):

\[
x(t) = P + td
\]

Let \( l(P, d) \) be the intensity seen along a ray. Then:

\[
l(P, d) = l_{\text{direct}}
\]

where

- \( l_{\text{direct}} \) is computed from the Phong model

**Ray-tracing pseudocode**

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

```plaintext
function traceimage(scene):
    for each pixel \((i, j)\) in image
        \( A = \text{pixelToWorld}(i, j) \)
        \( P = \text{COP} \)
        \( d = (A - P) / ||A - P|| \)
        \( l(i, j) = \text{traceRay}(scene, P, d) \)
    end for
end function
```

```plaintext
function traceRay(scene, \( P, d \)):
    \((t, N, mtrl) \leftarrow \text{sceneIntersect}(P, d)\)
    \( Q \leftarrow \text{ray}(\text{in} \ P, d) \) evaluated at \( t \)
    \( l = \text{shade}(A, \mu, N, \text{scene}, Q, \text{mtrl}) \)
    return \( l \)
end function
```

**Shading pseudocode**

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

```plaintext
function shade(mtrl, scene, \( Q, N, d \)):
    \( l \leftarrow \text{mtrl} \cdot k_d + \text{mtrl} \cdot k_a \cdot I_o \)
    for each light source \( L \) do:
        distance = \( d \cdot \text{distanceAttenuation}(Q) \)
        \( l \leftarrow l + \text{atten} \cdot (\text{diffuse term} + \text{specular term}) \)
    end for
    return \( l \)
end function
```

**Ray casting with shadows**

Now we'll add shadows by casting shadow rays:

```plaintext
[Diagram showing ray casting with shadows]
```
Shading with shadows

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

```python
function shade(material, scene, N, L):
    I ← material.I0 + material.L * I0
    for each light source L do:
        atten ← L → distance attenuation(0) *
        L ← shadow attenuation(scene, L)
        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:

```python
function point_light_shadow_attenuation(scene, P)
    d ← (this position - P).normalized
    (t, N, mtel) ← scene.intersect(P, d)
    Compute f_light
    if t < t_light then:
        atten ← 0
    else
        atten ← 1
    end if
    return atten
end function
```

For a directional light, \( f_{\text{light}} = \infty \).

Shading in "Trace"

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

- Distance attenuation is clamped to be at most 1:

  \[
  A^{\text{dist}} = \min \left\{ 1, \frac{1}{d + b_d d + c_d d^2} \right\}
  \]

- Shadow attenuation \( A^{\text{shadow}} \) is included.

Here's what it should look like:

\[
I = I_0 + \sum A_j^{\text{spec}} L_j (N \cdot L_j) + \sum A_j^{\text{diff}} L_j (N \cdot L_j) + \sum A_j^{\text{amb}} L_j
\]

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.

You must use the shading equation on this slide, not the one in Shirley's textbook.

Recursive ray tracing with reflection

Now we'll add reflection:
Shading with reflection

Let \( k(P, d) \) be the intensity seen along a ray. Then:

\[
k(P, d) = k_{\text{direct}} + k_{\text{reflected}}
\]

where

- \( k_{\text{direct}} \) is computed from the Phong model, plus shadow attenuation
- \( k_{\text{reflected}} = k_d \cdot (Q \cdot R) \)

Typically, we set \( k_d = k_s \)

Ray-tracing pseudocode, revisited

```plaintext
function traceRay(scene, P, d):
    (C, N, mtl) ← scene.intersect(P, d)
    O ← ray(P, d) evaluated at t
    l = shade(scene, mtl, O, N, d)
    R = reflectDirection(-d, N)
    l ← l + mtl.k_r * traceRay(scene, O, R)
    return l
end function
```

Terminating recursion

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

Possibilities:

- Depth threshold
- \( \sum k_i < \text{threshold} \)
- Early/adaptive ray termination
Whitted ray tracing

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

Shading with reflection and refraction

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

\[
I(P, d) = I_{\text{direct}} + I_{\text{refracted}} + I_{\text{transmitted}}
\]

where

- \( I_{\text{direct}} \) is computed from the Phong model, plus shadow attenuation
- \( I_{\text{refracted}} = k_r I(Q, R) \)
- \( I_{\text{transmitted}} = k_t I(Q, T) \)

Typically, we set \( k_r = k_t \) and \( k_t = 1 - k_r \) (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. This is discussed in Shirley's textbook and can be implemented for extra credit.)

Refraction

Snell's law of refraction:

\[
\frac{n_1}{n_2} \sin \theta_1 = \sin \theta_2
\]

where \( n_1 \) and \( n_2 \) are indices of refraction.

In all cases, \( R \) and \( T \) are coplanar 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.)

Total Internal Reflection

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

\[
\theta_2 = \sin^{-1} \left( \frac{n_2}{n_1} \sin \theta_1 \right)
\]

What happens when \( \theta_1 > \theta_c \)?

When \( \theta_1 \) is exactly 90°, we say that \( \theta_1 \) has achieved the "critical angle" \( \theta_c \).

For \( \theta_1 > \theta_c \), 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:

\[
\begin{align*}
\mathbf{r} &= \mathbf{R} \\
\mathbf{t} &= T \\
\mathbf{g} &= G \\
\mathbf{d} &= D \\
\mathbf{n} &= N \\
\mathbf{t}_e &= t_e
\end{align*}
\]

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

```python
function traceRay(scene, \mathbf{P}, \mathbf{d}):
    \mathbf{t}, \mathbf{N}, \mathbf{mtr} \leftarrow \text{scene.intersect}(\mathbf{P}, \mathbf{d})
    \mathbf{O} \leftarrow \text{ray(\mathbf{P}, \mathbf{d}) evaluated at } \mathbf{t}
    \mathbf{I} = \text{shade(scene, mtr, } \mathbf{O}, \mathbf{N}, -\mathbf{d})
    \mathbf{R} = \text{reflectDirection(} \mathbf{N}, -\mathbf{d})
    \mathbf{I} \leftarrow \mathbf{I} \times \text{mtr.K} \times \text{traceRay(scene, } \mathbf{O}, \mathbf{R})
    \text{if ray is entering object then}
        \text{n.i = index of air}
        \mathbf{n.t} = \text{mtr index}
    \text{else}
        \mathbf{n.i} = \text{mtr.in dex}
        \mathbf{n.t} = \text{index of air}
        \text{if } (\text{not in scene}(\mathbf{N}, \mathbf{d}_2, \mathbf{A}; \mathbf{N}_i)) \text{ then}
            \mathbf{T} = \text{refracDirection}()
            \mathbf{I} \leftarrow \mathbf{I} \times \text{mtr.K} \times \text{traceRay(scene, } \mathbf{O}, \mathbf{T})
        \text{end if}
    \text{return } \mathbf{I}
\end{function}
```

Q: How do we decide if a ray is entering the object?

Terminating recursion, incl. refraction

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

\[
\begin{align*}
\mathbf{N}_f \mathbf{d} = (\mathbf{K}) \mathbf{K}_r \mathbf{K}_t \mathbf{K}_o
\end{align*}
\]

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, $k_p$ which gets factored in every time an object is entered, possibly more than once for the same object.

Shirley's textbook describes a better attenuation model 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.

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.

Intersecting rays with spheres

Now we've done everything except figure out what the "nene_intersect2, d2" function does.

Mostly it calls each object to find out the texel 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:
  
  \[ x' = x + t \cdot d_x, \quad y' = y + t \cdot d_y, \quad z' = z + t \cdot d_z \]

- A unit sphere centered at the origin defined by the equation:
  
  \[ x'^2 + y'^2 + z'^2 = 1 \]

Find: the rat which the ray intersects $P$.

Intersecting rays with spheres

Solution by substitution:

\[ \frac{(x_0 + t \cdot d_x)^2 + (y_0 + t \cdot d_y)^2 + (z_0 + t \cdot d_z)^2 - 1}{\alpha t^2 + \beta t + \gamma} = 0 \]

where

\[ \alpha = d_x^2 + d_y^2 + d_z^2 \]
\[ \beta = 2(x_0 d_x + y_0 d_y + z_0 d_z) \]
\[ \gamma = x_0^2 + y_0^2 + z_0^2 - 1 \]

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

\[ t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

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

\[ N = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \]
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, \( \mathbf{n} = (a, b, c)^T \). Thus, we can rewrite the plane equation as:

\[ \mathbf{n} \cdot \mathbf{r} = d \]

where \( \mathbf{r} = (x, y, z)^T \) is a point in space. We can solve for the intersection parameter (and thus the point):

\[ \mathbf{n} \cdot \mathbf{r} = d \]
\[ \mathbf{n} \cdot (\mathbf{r} + \mathbf{t} \mathbf{n}) = d \]
\[ \mathbf{n} \cdot \mathbf{r} + \mathbf{n} \cdot \mathbf{t} \mathbf{n} = d \]
\[ \mathbf{n} \cdot \mathbf{r} = d \]

If \( \mathbf{n} \cdot \mathbf{r} = 0 \) then \( \mathbf{r} \) is outside the plane.

3D inside-outside test

One way to do this “inside-outside test” is to see if \( \mathbf{Q} \) lies on the left side of each edge as we move counterclockwise around the triangle.

\[
[(\mathbf{B} - \mathbf{A}) \times (\mathbf{Q} - \mathbf{A})] \cdot \mathbf{n} \geq 0
\]

and

\[
[(\mathbf{C} - \mathbf{A}) \times (\mathbf{Q} - \mathbf{A})] \cdot \mathbf{n} \geq 0
\]

If \( \mathbf{Q} \) is inside \( \triangle ABC \), then \( \mathbf{Q} \) is inside \( \triangle ABC \).

Why is this projection desirable? Faster.

Which axis should you “project away”? Axis corresponding to largest component of \( \mathbf{n} \).

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?

\[ \mathbf{N} = (\mathbf{C} - \mathbf{B}) \times (\mathbf{A} - \mathbf{B}) \]

Given this normal, how would we compute \( \mathbf{d} \)?

\[ \mathbf{n} \cdot \mathbf{A} = d = \mathbf{n} \cdot \mathbf{B} = \mathbf{n} \cdot \mathbf{C} \]

Using these coefficients, we can solve for \( \mathbf{Q} \). Now, we need to decide if \( \mathbf{Q} \) is inside or outside of the triangle.

2D inside-outside test

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

If \( \mathbf{Q} \) is inside \( \triangle ABC \), then \( \mathbf{Q} \) is inside \( \triangle ABC \).
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 $\alpha$, $\beta$, and $\gamma$ the barycentric coordinates of $Q$ with respect to $A$, $B$, and $C$.

$$\begin{bmatrix}
\begin{array}{c}
A_x \\
B_x \\
C_x
\end{array}
\end{bmatrix} + \begin{bmatrix}
\begin{array}{c}
\alpha \\
\beta \\
\gamma
\end{array}
\end{bmatrix} = \begin{bmatrix}
\begin{array}{c}
Q_x \\
Q_y \\
Q_z
\end{array}
\end{bmatrix}$$

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

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

$$Q = (1 - \beta - \gamma) A + \beta B + \gamma C$$

$$\alpha = 1 - \beta - \gamma$$

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:

$$\alpha = \frac{\text{Area}(BCQ)}{\text{Area}(ABC)}$$

$$\beta = \frac{\text{Area}(ACQ)}{\text{Area}(ABC)}$$

$$\gamma = 1 - \alpha - \beta$$

How can cross products help here?

$$\text{Area}(ABQ) = \frac{1}{2} |(B - A) \times (C - A)|$$

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.)

$$N_a = \alpha N_A + \beta N_B + \gamma N_C$$

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?

$$t < \text{RAY - EPSILON} \Rightarrow \text{discard}$$
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 epsilon are used.