Ray Tracing

Reading

Required:
- Watt, sections 1.3-1.4, 12.1-12.5.1 (handout)
- Triangle intersection handout

Further reading:
- Watt errata on syllabus page, needed if you work from his book instead of the handout.

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 $L_i$ 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:

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 \( t \):

\[ P + td \]

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

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

where

* \( I_{\text{direct}} \) is computed from the Phong model

Ray-tracing pseudocode

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

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

function traceRay(scene, \( P \), \( d \)):
    \((t, N, \text{mtrl}) \leftarrow \text{scene.intersect}(P, d)\)
    \( Q \leftarrow \text{ray}(P, d) \) evaluated at \( t \)
    \( I = \text{shade}(\ ) \)
    return \( I \)
end function

Shading pseudocode

Next, we need to calculate the color returned by the \textit{shade} function.

function shade(mtrl, scene, \( Q \), \( N \), \( d \)):
    \( I \leftarrow \text{mtrl.}k_e + \text{mtrl.}k_a \cdot I_a \)
    for each light source \( L \) do:
        atten = \( L \rightarrow \text{distanceAttenuation}(\ ) \)
        \( I \leftarrow I + \text{atten} \cdot (\text{diffuse term + specular term}) \)
    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:

```python
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:

```python
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, $k_t$.]

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:
  \[
  A_{\text{dist}}^j = \min \left\{ 1, \frac{1}{a_j + b_j d_j + c_j d_j^2} \right\}
  \]
- Shadow attenuation \( A_{\text{shadow}} \) is included.

Here’s what it should look like:

\[
I = I_0 + A_{\text{shadow}} A_{\text{dist}} L_j \left[ k_d (N \cdot L)_j + k_s (V \cdot R)_j \right] + \sum_j A_{\text{shadow}} A_{\text{dist}} L_j \left[ k_d (N \cdot L)_j + k_s (V \cdot R)_j \right]
\]

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:

![Recursive ray tracing diagram]

Shading with reflection

![Shading with reflection diagram]

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

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

where

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

Typically, we set \( k_r = k_s \).

Reflection

![Reflection diagram]

Law of reflection:

\[
\theta = \theta_r
\]

**R** is co-planar with \( d \) and \( N \).
Ray-tracing pseudocode, revisited

```python
function traceRay(scene, P, d):
    (t, N, mtrl) ← scene.intersect(P, d)
    Q ← ray(P, d) evaluated at t
    l = shade(scene, mtrl, P, N, -d)
    R = reflectDirection()
    l ← l + mtrl.kr * traceRay(scene, Q, R)
    return l
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:

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

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

where

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

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.]`
Refraction

Snell's law of refraction:
\[ \eta_i \sin \theta_i = \eta_t \sin \theta_t \]
where \( \eta_i, \eta_t \) 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.)

<table>
<thead>
<tr>
<th>Medium</th>
<th>Index of refraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacuum</td>
<td>1</td>
</tr>
<tr>
<td>Air</td>
<td>1.0003</td>
</tr>
<tr>
<td>Water</td>
<td>1.33</td>
</tr>
<tr>
<td>Fused quartz</td>
<td>1.46</td>
</tr>
<tr>
<td>Glass, crown</td>
<td>1.52</td>
</tr>
<tr>
<td>Glass, dense flint</td>
<td>1.66</td>
</tr>
<tr>
<td>Diamond</td>
<td>2.42</td>
</tr>
</tbody>
</table>

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 \( \theta_t \) is exactly 90°, we say that \( \theta_i \) has achieved the "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.

Watt handout

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

\[
\begin{align*}
\phi &= \theta_i \\
\theta &= \theta_t \\
\mu_1 &= \eta_i \\
\mu_2 &= \eta_t
\end{align*}
\]

Also, Watt had some 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 for corrections.

Ray-tracing pseudocode, revisited

function traceRay(scene, P, d):
    \((t, N, mtrl) \leftarrow scene\text{.intersect}(P, d)\)
    \(Q \leftarrow ray(P, d)\) evaluated at \(t\)
    \(I = shade(scene, mtrl, P, N, -d)\)
    \(R = reflectDirection(N, -d)\)
    \(I \leftarrow I + mtrl.k_r \ast traceRay(scene, Q, R)\)
    if ray is entering object then
        \(n_i = \text{index\_of\_air}\)
        \(n_t = mtrl.index\)
    else
        \(n_i = mtrl.index\)
        \(n_t = \text{index\_of\_air}\)
        if \(\text{notTIR}()\) then
            \(T = refractDirection()\)
            \(I \leftarrow I + mtrl.k_t \ast traceRay(scene, Q, T)\)
        end if
        return \(I\)
    end if
end function
Terminating recursion, incl. refraction

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

Intersecting rays with spheres

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

Intersecting rays with spheres

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

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, \( \mathbf{n} = [a \ b \ c]^T \). Thus, we can rewrite 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.

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 \( \alpha, \beta, \) and \( \gamma \) the barycentric coordinates of \( Q \) with respect to \( A, B, \) and \( C \).
**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!

![Barycentric coordinates diagram](image)

**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?

**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.