Shading 3: Ray Tracing

Christian Miller
CS 354 - Fall 2011
Ray Tracing

• Both a shading and rendering technique
  • An alternative to rasterization
• Very physically intuitive
• The staple of high-quality renderings
• Very parallel, but very poor cache coherency and lots of unpredictable branching
  • That makes it slow
Why we do this
The basic idea

- “Fire” a ray from the camera through a pixel on the near plane and into the scene
- Find the first thing the ray hits
- Recursively fire more rays from the intersection:
  - Shadow rays check to see if a light is visible
  - Evaluate lighting model if it is
  - Refraction / reflection rays are like camera rays
The basic idea
Basic pseudocode

ray_trace(scene S, ray R)
  (hit, obj, P, N) = find_first_intersection(S, R)
  if (!hit) then return false

  color C = (0, 0, 0)

  foreach light L
    if (ray_trace(S, ray_to_light(P, L)))
      C += local_light_model(obj, P, N, L)

  if (obj is reflective)
    C += krfl * ray_trace(S, reflect_ray(N, R))

  if (obj is refractive)
    C += krfr * ray_trace(S, refract_ray(obj, N, R))

  return C
Ray refresher

- Directed semi-infinite parameterized line segment
- An origin point $\mathbf{o}$ and normalized direction vec. $\mathbf{d}$
- $t$ measures distance along ray from origin

$x = o + td$
Finding intersections

- We want to find the first object that the ray hits.
- That means the lowest value of $t > 0$.
- Compute intersection by $\mathbf{p} = \mathbf{o} + td$. 

\[ p = o + td \]
Ray-plane intersection

- Substitute ray equation into plane equation and solve for \( t \), check if it’s >0
- Divide by 0 if ray is parallel to plane!

\[
\begin{align*}
  \mathbf{x} &= \mathbf{o} + t\mathbf{d} \\
  \mathbf{N} \cdot \mathbf{x} + d &= 0 \\
  t &= \frac{-d - \mathbf{N} \cdot \mathbf{o}}{\mathbf{N} \cdot \mathbf{d}}
\end{align*}
\]
Ray-sphere intersection

- Same idea with a sphere: substitute ray equation into implicit equation for sphere and solve for $t$
- Gives us a quadratic equation

Mathematically:

1. **Ray Equation**:
   \[
   \mathbf{x} = \mathbf{o} + t\mathbf{d}
   \]

2. **Sphere Equation**:
   \[
   (\mathbf{x} - \mathbf{c})^2 = r^2
   \]

Using\(\mathbf{p}_1\),\(\mathbf{p}_2\), and their respective positions on the graph, we have:

- Point \(\mathbf{p}_1\): \[t = \frac{-d \cdot (\mathbf{o} - \mathbf{c}) + \sqrt{d^2(\mathbf{o} - \mathbf{c})^2 - (\mathbf{o} - \mathbf{c})^2 - r^2}}{d^2}
- Point \(\mathbf{p}_2\): \[t = \frac{-d \cdot (\mathbf{o} - \mathbf{c}) - \sqrt{d^2(\mathbf{o} - \mathbf{c})^2 - (\mathbf{o} - \mathbf{c})^2 - r^2}}{d^2}

The quadratic equation derived from these substitutions is:

\[
d^2t^2 + 2d \cdot (\mathbf{o} - \mathbf{c})t + (\mathbf{o} - \mathbf{c})^2 - r^2 = 0
\]
Ray-sphere intersection

Δ > 0
Δ = 0
Δ < 0

• Several cases depending on discriminant
• Basically a plane, but with a limited region to hit

• How do we tell if a point is inside the triangle?
Barycentric coords

- Represent a point inside a triangle as a convex combination of its vertices
- Coordinates all $\geq 0$, sum to 1 inside the triangle

\[ p = u v_0 + v v_1 + w v_2 \]

\[ u, v, w \geq 0 \]

\[ w = 1 - u - v \]
Point inside triangle

- Convert to barycentric coordinates first, then see if they’re all $\geq 0$
Point inside triangle

- Start by subtracting off $v_0$ to get vectors relative to that corner.

\[
R = p - v_0 \\
Q_1 = v_1 - v_0 \\
Q_2 = v_2 - v_0
\]
Point inside triangle

\[ \mathbf{R} = v\mathbf{Q}_1 + w\mathbf{Q}_2 \]

- Plug in \( \mathbf{R} \), but the resulting linear system is overconstrained (3 equations but only 2 unknowns)
Point inside triangle

- Instead, take dot product of $\mathbf{R}$ with both $\mathbf{Q}$’s
- Guaranteed to be solvable if $\mathbf{R} \neq \mathbf{Q}_1 \neq \mathbf{Q}_2$

\[
\mathbf{R} \cdot \mathbf{Q}_1 = v\mathbf{Q}_1^2 + w(\mathbf{Q}_1 \cdot \mathbf{Q}_2)
\]
\[
\mathbf{R} \cdot \mathbf{Q}_2 = v(\mathbf{Q}_1 \cdot \mathbf{Q}_2) + w\mathbf{Q}_2^2
\]
Point inside triangle

- This gives us a 2 x 2 matrix that’s invertible
- Solve for \( v \) and \( w \), then \( u = 1 - v - w \)
Barycentric coords

• Once you have the barycentric coordinates of a point, you can easily tell if it’s in the triangle

• Can also use them as relative weights for vertex attributes from the corners
  • Interpolate colors, normals, texcoords, etc.
Shading and Shadows

- Fire a ray from hit point to each light
- If those rays don’t hit anything, use Blinn-Phong to compute surface color
Reflection / refraction

- If a ray hits an object, it could bounce off or refract through it!
- Recurse by firing a new ray in that direction, then summing its color into the result
Reflection vector

- Goes out the same angle it came in, but in the opposite direction
- Simple to compute, be sure to check direction of $L$

$$R = 2(N \cdot L)N - L$$
Refraction

- Refraction is more complicated
- The angle of transmission depends on the indices of refraction for the two materials
- Angle computed using Snell’s Law

\[ \eta_i \sin(\theta_i) = \eta_R \sin(\theta_R) \]
Index of refraction

- The $\eta$ values in Snell’s Law are indices of refraction
- Basically describes the speed of light in a medium
- Vacuum = 1.0, air = 1.0003, water = 1.333, glass = 1.46, diamond = 2.42
- Determines how much rays are bent upon entering an object
\[
T = \left( \frac{\eta_L}{\eta_T} \mathbf{N} \cdot \mathbf{L} - \sqrt{1 - \frac{\eta_L^2}{\eta_T^2} (1 - (\mathbf{N} \cdot \mathbf{L})^2)} \right) \mathbf{N} - \frac{\eta_L}{\eta_T} \mathbf{L}
\]
Putting it together

- Fire a ray per pixel
- When it hits something, fire shadow rays to see if lights are occluded, then evaluate lighting model for the visible ones
- Recursively fire a reflection ray from the point of intersection, then blend that result into pixel by multiplying by some constant
- Do the same for refraction rays, as appropriate
Putting it together

• You need to put a cap on the level of recursion
  • Can use the blending weight to cull rays when they get too dim
  • Or just stop at a certain depth
• Can perform anti-aliasing by firing several rays per pixel with some jitter, then averaging them
Examples
Figures Courtesy...

- Real-Time Rendering, 3rd ed. [RTR]
  - Tomas Akenine-Moller, Eric Haines, Naty Hoffman

  - Eric Lengyel

  - Edward Angel, Dave Shreiner

- Wikipedia [WP]