Ray Tracing

Reading

Foley et al., 16.12

Optional:


Geometric optics

We will take the view 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).

Forward Ray Tracing

- Rays emanate from light sources and bounce around in the scene.
- Rays that pass through the projection plane and enter the eye contribute to the final image.

What’s wrong with this method?
**Eye vs. Light**

- Starting at the light (a.k.a. forward ray tracing, photon tracing)
- Starting at the eye (a.k.a. backward ray tracing)

**Whitted ray-tracing algorithm**

1. For each pixel, trace a **primary ray** 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 (transmitted ray)** in direction $T$

**Reflection**

- Reflected light from objects behaves like specular reflection from light sources
  - Reflectivity is just specular color
  - Reflected light comes from direction of perfect specular reflection

**Refraction**

- Amount to transmit determined by transparency coefficient, which we store explicitly
- $T$ comes from Snell’s law
  \[ \eta_s \sin(\theta_i) = \eta_t \sin(\theta_t) \]
**Total Internal Reflection**

- When passing from a dense medium to a less dense medium, light is bent further away from the surface normal.
- Eventually, it can bend right past the surface!
- The $\theta_i$ that causes $\theta_t$ to exceed 90 degrees is called the critical angle ($\theta_c$). For $\theta_i$ greater than the critical angle, no light is transmitted.
- A check for TIR falls out of the construction of T

**Index of Refraction**

- Real-world index of refraction is a complicated physical property of the material.

<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>

- IOR also varies with wavelength, and even temperature!
- How can we account for wavelength dependence when ray tracing?
Shading

If \( I(P_0, \mathbf{u}) \) is the intensity seen from point \( P_0 \) along direction \( \mathbf{u} \)

\[
I(P_0, \mathbf{u}) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}}
\]

where

\[
I_{\text{direct}} = \text{Shade}(N, L, \mathbf{u}, R) \quad \text{(e.g. Phong shading model)}
\]

\[
I_{\text{reflected}} = k_r I(P, R)
\]

\[
I_{\text{transmitted}} = k_t I(P, T)
\]

Typically, we set \( k_r = k_s \) and \( k_t \)

Parts of a Ray Tracer

- What major components make up the core of a ray tracer?
  - Outer loop sends primary rays into the scene
  - Trace arbitrary ray and compute its color contribution as it travels through the scene
  - Shading model

\[
I = k_e + k_d I_d + \sum_l f(d_l) I_l \left[ k_d (N \cdot L_l)_+ + k_s (V \cdot R)_+ \right]
\]

Outer Loop

```c
void traceImage (scene)
{
    for each pixel (i,j) in the image {
        \( p = \text{pixelToWorld}(i,j) \)
        \( \mathbf{c} = \text{COP} \)
        \( \mathbf{u} = (p - \mathbf{c})/||p - \mathbf{c}|| \)
        \( I(i,j) = \text{traceRay} (\text{scene}, \mathbf{c}, \mathbf{u}) \)
    }
}
```

Trace Pseudocode

```c
color traceRay(point P_0, direction \mathbf{u})
{
    \( (P, O_i) = \text{intersect}(P_0, \mathbf{u}) \);
    \( I = 0 \)
    for each light source \( l \) {
        \( (P', \text{LightObj}) = \text{intersect}(P, \text{dir}(P, l)) \)
        if LightObj == l {
            \( I = I + I(l) \)
        }
    }
    \( I = I + \text{Obj.Kr} * \text{traceRay}(P, \mathbf{R}) \)
    \( I = I + \text{Obj.Kt} * \text{traceRay}(P, \mathbf{T}) \)
    return I
}
```
**TraceRay Pseudocode**

```plaintext
function traceRay(scene, P0, u) {
    (t, P, N, obj) ← scene.intersect(P0, u)
    I = shade(u, N, scene)
    R = reflectDirection(u, N)
    I ← I + obj.kr * traceRay(scene, P, R)
    if ray is entering object {
        (ni, nt) ← (index_of_air, obj.index)
    } else {
        (ni, nt) ← (obj.index, index_of_air)
    }
    if (notTIR(u, N, ni, nt)) {
        T = refractDirection(u, N, ni, nt)
        I ← I + obj.kt * traceRay(scene, P, T)
    }
    return I
}
```

**Controlling Tree Depth**

- Ideally, we’d spawn child rays at every object intersection forever, getting a “perfect” color for the primary ray.
- In practice, we need heuristics for bounding the depth of the tree (i.e., recursion depth)

**Shading Pseudocode**

```plaintext
function shade(obj, scene, P, N, u) {
    I ← obj.k_e + obj.k_s * scene->I_s
    for each light source ℓ {
        atten = distanceAttenuation(ℓ, P) *
            shadowAttenuation(ℓ, scene, P)
        I ← I + atten*(diffuse term + spec term)
    }
    return I
}
```

**Shadow attenuation pseudocode**

Check to see if a ray makes it to the light source.

```plaintext
function shadowAttenuation(ℓ, scene, P) {
    d = (ℓ.position - P).normalize()
    (t, Pl, N, obj) ← scene.intersect(P, d)
    if Pl is before the light source {
        atten = 0
    } else {
        atten = 1
    }
    return atten
}
```

Q: What if there are transparent objects along a path to the light source?
Ray-Object Intersection
- Must define different intersection routine for each primitive
- The bottleneck of the ray tracer, so make it fast!
- Most general formulation: find all roots of a function of one variable
- In practice, many optimized intersection tests exist (see Glassner)

Ray-Sphere Intersection
- Given a sphere centered at $P_c = [0,0,0]$ with radius $r$ and a ray $P(t) = P_0 + tu$, find the intersection(s) of $P(t)$ with the sphere.

Object hierarchies and ray intersection
How do we intersect with primitives transformed with affine transformations?

Numerical Error
- Floating-point roundoff can add up in a ray tracer, and create unwanted artifacts
  - Example: intersection point calculated to be ever-so-slightly inside the intersecting object. How does this affect child rays?
- Solutions:
  - Perturb child rays
  - Use global ray epsilon
Plane Intersection

- We can write the equation of a plane as:
  \[ ax + by + cz + d = 0 \]

- The coefficients \( a, b, \) and \( c \) form a vector that is normal to the plane, \( \mathbf{n} = [a b c]^T \).
  Thus, we can re-write the plane equation as:
  \[ \mathbf{n} \cdot (\mathbf{P} + \mathbf{u}) + d = 0 \]

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

Ray-Polymesh Intersection

1. Use bounding sphere for fast failure
2. Test only front-facing polygons
3. Intersect ray with each polygon’s supporting plane
4. Use a point-in-polygon test
5. Intersection point is smallest \( t \)

Axis-Aligned Cube Intersection

- for each pair of parallel planes, compute \( t \) intersection values for both
- Let \( t_{\text{near}} \) be the smaller, \( t_{\text{far}} \) be the larger
- let \( t_1 = \text{largest } t_{\text{near}}, t_2 = \text{smallest } t_{\text{far}} \)
- ray intersects cube if \( t_1 < t_2 \)
- intersection point given by \( t_1 \)

Goodies

- There are some advanced ray tracing feature that self-respecting ray tracers shouldn’t be caught without:
  - Acceleration techniques
  - Antialiasing
  - CSG
  - Distribution ray tracing
Acceleration Techniques

- Problem: ray-object intersection is very expensive
  - make intersection tests faster
  - do fewer tests

Fast Failure

- We can greatly speed up ray-object intersection by identifying cheap tests that guarantee failure
- Example: if origin of ray is outside sphere and ray points away from sphere, fail immediately.

Hierarchical Bounding Volumes

- Arrange scene into a tree
  - Interior nodes contain primitives with very simple intersection tests (e.g., spheres). Each node’s volume contains all objects in subtree
  - Leaf nodes contain original geometry
- Like BSP trees, the potential benefits are big but the hierarchy is hard to build

Spatial Subdivision

- Divide up space and record what objects are in each cell
- Trace ray through voxel array
Antialiasing

- So far, we have traced one ray through each pixel in the final image. Is this an adequate description of the contents of the pixel?

- This quantization through inadequate sampling is a form of **aliasing**. Aliasing is visible as “jaggies” in the ray-traced image.

- We really need to colour the pixel based on the **average**

Supersampling

- We can approximate the average colour of a pixel’s area by firing multiple rays and averaging the result.

Adaptive Sampling

- Uniform supersampling can be wasteful if large parts of the pixel don’t change much.

- So we can subdivide regions of the pixel’s area only when the image changes in that area:

- How do we decide when to subdivide?
**CSG**

- CSG (constructive solid geometry) is an incredibly powerful way to create complex scenes from simple primitives.

- CSG is a modeling technique; basically, we only need to modify ray-object intersection.

**CSG Implementation**

- CSG intersections can be analyzed using “Roth diagrams”.
  - Maintain description of all intersections of ray with primitive
  - Functions to combine Roth diagrams under CSG operations

- An elegant and extremely slow system

**Distribution Ray Tracing**

- Usually known as “distributed ray tracing”, but it has nothing to do with distributed computing
- General idea: instead of firing one ray, fire multiple rays in a jittered grid

- Distributing over different dimensions gives different effects
- Example: what if we distribute rays over pixel area?

**Noise**

- Noise can be thought of as randomness added to the signal.
- The eye is relatively insensitive to noise.
DRT pseudocode

traceImage() looks basically the same, except now each pixel records the average color of jittered sub-pixel rays.

function traceImage (scene):
    for each pixel (i, j) in image do
        I(i, j) ← 0
        for each sub-pixel id in (i,j) do
            s ← pixelToWorld(jitter(i, j, id))
            p ← COP
            u ← (s - p).normalize()
            I(i, j) ← I(i, j) + traceRay(scene, p, u, id)
        end for
        I(i, j) ← I(i, j)/numSubPixels
    end for
end function

• A typical choice is numSubPixels = 4*4.

DRT pseudocode (cont’d)

• Now consider traceRay(), modified to handle (only) opaque glossy surfaces:

function traceRay(scene, p, u, id):
    (q, N, obj) ← intersect (scene, p, u)
    I ← shade(…)
    R ← jitteredReflectDirection(N, -u, id)
    I ← I + obj.k_t * traceRay(scene, q, R, id)
    return I
end function

Pre-sampling glossy reflections

Distributing Reflections

• Distributing rays over reflection direction gives:
Distributing Refractions

• Distributing rays over transmission direction gives:

Distributing Over Light Area

• Distributing over light area gives:

Distributing Over Aperature

• We can fake distribution through a lens by choosing a point on a finite aperture and tracing through the “in-focus point”.

Distributing Over Time

• We can endow models with velocity vectors and distribute rays over time. This gives:
Chaining the ray id’s

- In general, you can trace rays through a scene and keep track of their id’s to handle all of these effects: