Ray Tracing

CS 4620 Lecture 4

A Little Background

- Ray casting
  - Process of shooting rays into scene to get pixel colors
  - Nonrecursive, i.e., no interreflections
  - Origin: Arthur Appel, 1968 (earlier work by others for nonrendering)

- Ray tracing
  - Process of shooting rays into scene and resolving reflections and refractions to get pixel colors
  - Recursive by nature
  - Origin: Turner Whitted, 1980

Motivation

The number of reflections a "ray" can take and how it is affected each time it encounters a surface is all controlled via software settings during ray tracing. Here, each ray was allowed to reflect up to 16 times. Multiple "reflections of reflections" can thus be seen.
Motivation

"A render of a few spheres, created in Rhinoceros 3D and rendered using V-Ray. This render features: Depth of field, hexagonal aperture (and consequently hexagonal bokeh), Fresnel reflections, area lights, global illumination, diffuse interreflection, ambient occlusion etc."

[Mimigu 2009 (Wikipedia)]

Motivation

"Recursive raytracing of a sphere, which incorporates the effects of diffuse interreflection, depth-of-field, ambient occlusion, area light sources, and Fresnel reflection."

[Tim Babb, 2008 (Wikimedia Commons)]

Motivation

"This image was created by Gilles Tran with POV-Ray 3.6 using Radiosity."

[Zoom]
Ray tracing idea

for each pixel {
  compute viewing ray
  intersect ray with scene
  compute illumination at visible point
  put result into image
}

Generating eye rays

• Use window analogy directly
Vector math review

- Vectors and points
- Vector operations
  - addition
  - scalar product
- More products
  - dot product
  - cross product
- Bases and orthogonality

Generating eye rays—orthographic

- Just need to compute the view plane point \( s \):
  \[
  p = s; \ d = d_v
  \]
  \[
  r(t) = p + t d
  \]
- but where exactly is the view rectangle?

Generating eye rays—orthographic

- Positioning the view rectangle
  - establish three vectors to be camera basis: \( u, v, w \)
  - view rectangle is in \( u-v \) plane, specified by \( l, r, t, b \)
  - now ray generation is easy:
  \[
  s = e + u u + v v
  \]
  \[
  p = s; \ d = -w
  \]
  \[
  r(t) = p + t d
  \]

Generating eye rays—perspective

- View rectangle needs to be away from viewpoint
- Distance is important: “focal length” of camera
  - still use camera frame but position view rect away from viewpoint
  - ray origin always \( e \)
  - ray direction now controlled by \( s \)
Generating eye rays—perspective

- Compute \( s \) in the same way; just subtract \( dw \)
  - coordinates of \( s \) are \((u, v, -d)\)

\[
s = e + uu + vv - dw
\]

\[
p = e; \quad d = s - e
\]

\[
r(t) = p + td
\]

Pixel-to-image mapping

- One last detail: \((u, v)\) coords of a pixel

\[
\begin{align*}
  u &= l + (r - l)(i + 0.5)/n_x \\
  v &= b + (t - b)(j + 0.5)/n_y
\end{align*}
\]

Ray: a half line

- Standard representation: point \( p \) and direction \( d \)
  \[
r(t) = p + td
\]
  - this is a parametric equation for the line
  - lets us directly generate the points on the line
  - if we restrict to \( t > 0 \) then we have a ray
  - note replacing \( d \) with \( ad \) doesn’t change ray \((a > 0)\)
Ray-sphere intersection: algebraic

• Condition 1: point is on ray
  \[ r(t) = p + td \]

• Condition 2: point is on sphere
  – assume unit sphere; see Shirley or notes for general
  \[ \|x\| = 1 \Leftrightarrow \|x\|^2 = 1 \]
  \[ f(x) = x \cdot x - 1 = 0 \]

• Substitute:
  \[ (p + td) \cdot (p + td) - 1 = 0 \]
  – this is a quadratic equation in \( t \)

Ray-sphere intersection: geometric

Ray-box intersection

• Could intersect with 6 faces individually
• Better way: box is the intersection of 3 slabs
Ray-slab intersection

- 2D example
- 3D is the same!

\[ p_x + t_x \min d_x = x_{\min} \]
\[ t_x = (x_{\min} - p_x) / d_x \]
\[ p_y + t_y \min d_y = y_{\min} \]
\[ t_y = (y_{\min} - p_y) / d_y \]

Intersecting intersections

- Each intersection is an interval
- Want last entry point and first exit point

\[ t_{\min} = \max(t_x \min, t_y \min) \]
\[ t_{\max} = \min(t_x \max, t_y \max) \]

Ray-triangle intersection

- Condition 1: point is on ray
  \[ r(t) = p + td \]
- Condition 2: point is on plane
  \[ (x - a) \cdot n = 0 \]
- Condition 3: point is on the inside of all three edges
- First solve 1 & 2 (ray–plane intersection)
  - substitute and solve for \( t \):
  \[ (p + td - a) \cdot n = 0 \]
  \[ t = \frac{(a - p) \cdot n}{d \cdot n} \]
Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  - vector of edge to vector to \( x \)
- Use cross product to decide

Ray-triangle intersection

- See book for a more efficient method based on linear systems
  - (don’t need this for Ray 1 anyhow—but stash away for Ray 2)

Ray-triangle intersection

\[
(b - a) \times (x - a) \cdot n > 0 \\
(c - b) \times (x - b) \cdot n > 0 \\
(a - c) \times (x - c) \cdot n > 0
\]

Image so far

- With eye ray generation and sphere intersection

```java
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
  for 0 <= ix < nx {
    ray = camera.getRay(ix, iy);
    hitSurface, t = s.intersect(ray, 0, +inf)
    if hitSurface is not null
      image.set(ix, iy, white);
  }
```
**Intersection against many shapes**

- The basic idea is:

```java
Group.intersect (ray, tMin, tMax) {
  tBest = +inf; firstSurface = null;
  for surface in surfaceList {
    hitSurface, t = surface.intersect(ray, tMin, tBest);
    if hitSurface is not null {
      tBest = t;
      firstSurface = hitSurface;
    }
  }
  return hitSurface, tBest;
}
```

- this is linear in the number of shapes
- but there are sublinear methods (acceleration structures)

**Image so far**

- With eye ray generation and scene intersection

```java
for 0 <= iy < ny
  for 0 <= ix < nx {
    ray = camera.getRay(ix, iy);
    c = scene.trace(ray, 0, +inf);
    image.set(ix, iy, c);
  }
```

```java
Scene.trace(ray, tMin, tMax) {
  surface, t = surf.intersect(ray, tMin, tMax);
  if (surface != null) return surface.color();
  else return black;
}
```

**Shading**

- Compute light reflected toward camera

**Inputs:**
- eye direction
- light direction
  (for each of many lights)
- surface normal
- surface parameters
  (color, shininess, …)

**Diffuse reflection**

- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert's cosine law

![Diffuse reflection diagram](image-url)
**Lambertian shading**

- Shading independent of view direction

\[ L_d = k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) \]

**Produces matte appearance**

**Image so far**

```java
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}

...Surface.shade(ray, point, normal, light) {
    v = -normalize(ray.direction);
    l = normalize(light.pos - point);
    // compute shading
```

**Diffuse shading**

- Produces matte appearance

**Image so far**

```java
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}

...Surface.shade(ray, point, normal, light) {
    v = -normalize(ray.direction);
    l = normalize(light.pos - point);
    // compute shading
```

**Image so far**

```java
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}

...Surface.shade(ray, point, normal, light) {
    v = -normalize(ray.direction);
    l = normalize(light.pos - point);
    // compute shading
```

**Image so far**

```java
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point, normal, light);
    }
    else return backgroundColor;
}

...Surface.shade(ray, point, normal, light) {
    v = -normalize(ray.direction);
    l = normalize(light.pos - point);
    // compute shading
```
Shadows

• Surface is only illuminated if nothing blocks its view of the light.
• With ray tracing it’s easy to check
  – just intersect a ray with the scene!

Image so far

Surface.shade(ray, point, normal, light) {
  shadRay = (point, light.pos – point);
  if (shadRay not blocked) {
    v = –normalize(ray.direction);
    l = normalize(light.pos – point);
    // compute shading
  }
  return black;
}

Shadow rounding errors

• Don’t fall victim to one of the classic blunders:

  • What’s going on?
    – hint: at what t does the shadow ray intersect the surface you’re shading?

• Solution: shadow rays start a tiny distance from the surface

  • Do this by moving the start point, or by limiting the t range
Multiple lights

- Important to fill in black shadows
- Just loop over lights, add contributions
- Ambient shading
  - black shadows are not really right
  - one solution: dim light at camera
  - alternative: add a constant “ambient” color to the shading…

Specular shading (Blinn-Phong)

- Intensity depends on view direction
  - bright near mirror configuration

Image so far

```cpp
shade(ray, point, normal, lights) {
  result = ambient;
  for light in lights {
    if (shadow ray not blocked) {
      result += shading contribution;
    }
  }
  return result;
}
```

Specular shading (Blinn-Phong)

- Close to mirror ⇔ half vector near normal
  - Measure “near” by dot product of unit vectors

$$\begin{align*}
  \mathbf{h} &= \text{bisector}(\mathbf{v}, \mathbf{l}) \\
  &= \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|} \\
  L_s &= k_s I \max(0, \cos \alpha)^p \\
  &= k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p
\end{align*}$$
Phong model—plots

- Increasing $n$ narrows the lobe

![Phong model plots](image)

Specular shading

- Equation: $k_s p$

![Specular shading](image)

Diffuse + Phong shading

- Shading combination

![Diffuse + Phong shading](image)

Ambient shading

- Shading that does not depend on anything
  - Add constant color to account for disregarded illumination and fill in black shadows

$$L_a = k_a I_a$$
Putting it together

- Usually include ambient, diffuse, Phong in one model

\[ L = L_a + L_d + L_s \]
\[ = k_a I_a + k_d I \max(0, n \cdot l) + k_s I \max(0, n \cdot h)^p \]

- The final result is the sum over many lights

\[ L = L_a + \sum_{i=1}^{N} [(L_d)_i + (L_s)_i] \]
\[ L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, n \cdot l_i) + k_s I_i \max(0, n \cdot h_i)^p] \]

Mirror reflection

- Consider perfectly shiny surface
  - there isn’t a highlight
  - instead there’s a reflection of other objects

- Can render this using recursive ray tracing
  - to find out mirror reflection color, ask what color is seen from surface point in reflection direction
  - already computing reflection direction for Phong…

- “Glazed” material has mirror reflection and diffuse

\[ L = L_a + L_d + L_m \]
- where \( L_m \) is evaluated by tracing a new ray
**Diffuse + mirror reflection (glazed)**

(glazed material on floor)

---

**Ray tracer architecture 101**

- You want a class called Ray
  - point and direction; evaluate(t)
  - possible: tMin, tMax

- Some things can be intersected with rays
  - individual surfaces
  - groups of surfaces (acceleration goes here)
  - the whole scene
  - make these all subclasses of Surface
  - limit the range of valid t values (e.g. shadow rays)

- Once you have the visible intersection, compute the color
  - may want to separate shading code from geometry
  - separate class: Material (each Surface holds a reference to one)
  - its job is to compute the color

---

**Architectural practicalities**

- **Return values**
  - surface intersection tends to want to return multiple values
    - t, surface or shader, normal vector, maybe surface point
  - in many programming languages (e.g. Java) this is a pain
  - typical solution: an intersection record
    - a class with fields for all these things
    - keep track of the intersection record for the closest intersection
    - be careful of accidental aliasing (which is very easy if you’re new to Java)

- **Efficiency**
  - in Java the (or, a) key to being fast is to minimize creation of objects
  - what objects are created for every ray? try to find a place for them where you can reuse them.
  - Shadow rays can be cheaper (any intersection will do, don’t need closest)
  - but: “First Get it Right, Then Make it Fast”