Ray Tracing

CS 465 Lecture 3

Ray tracing algorithm

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

Plane projection in drawing

The concept of the plane plane may be better understood by thinking through a window or other transparent plane, and a method is used to make the illustrations. The idea is to think of the objects in the plane as being "projected" onto the plane, where the objects are projected onto a two-dimensional plane and their edges are projected onto the plane. This process is what is meant by "plane projection."
Plane projection in photography

- This is another model for what we are doing
  - applies more directly in realistic rendering

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

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: algebraic

- Solution for \( t \) by quadratic formula:
  \[ t = \frac{-d \cdot p \pm \sqrt{(d \cdot p)^2 - (d \cdot d)(p \cdot p - 1)}}{d \cdot d} \]

  - simpler form holds when \( d \) is a unit vector but we won’t assume this in practice (reason later)
  - I’ll use the unit-vector form to make the geometric interpretation

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!

\[
\begin{align*}
px + t_{xmin} \cdot dx &= x_{\text{min}} \\
\frac{t_{xmin}}{y_{\text{min}}} &= \frac{x_{\text{min}} - px}{dx} \\
p_y + t_{ymin} \cdot dy &= y_{\text{min}} \\
\frac{t_{ymin}}{y_{\text{min}}} &= \frac{y_{\text{min}} - p_y}{dy}
\end{align*}
\]

Intersecting intersections

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

\[
\begin{align*}
t_{\text{min}} &= \max(t_{xmin}, t_{ymin}) \\
t_{\text{max}} &= \min(t_{xmax}, t_{ymin})
\end{align*}
\]

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 \( t \):
    \[
    \begin{align*}
    (p + td - a) \cdot n &= 0 \\
    t &= \frac{(a - p) \cdot n}{d \cdot n}
    \end{align*}
    \]
- In plane, triangle is the intersection of 3 half spaces

Ray-triangle intersection

- In plane, triangle is the intersection of 3 half spaces
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

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

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);
    if (s.intersect(ray, 0, +inf) < +inf)
      image.set(ix, iy, white);
  }
```

Intersection against many shapes

- The basic idea is:
  ```java
  hit (ray, tMin, tMax) {
    tBest = +inf; hitSurface = null;
    for surface in surfaceList {
      t = surface.intersect(ray, tMin, tMax);
      if t < tBest {
        tBest = t;
        hitSurface = surface;
      }
    }
    return hitSurface, t;
  } 
  ```
- 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

\[
\text{for } 0 \leq iy < ny \\
\text{for } 0 \leq ix < nx \\
\begin{array}{l}
\text{ray } = \text{camera.getRay}(ix, iy); \\
\text{c } = \text{scene.trace}(\text{ray}, 0, +\infty); \\
\text{image.set}(ix, iy, c);
\end{array}
\]

\[
\text{...}
\]

\[
\text{trace}(\text{ray}, t_{\text{Min}}, t_{\text{Max}}) \\
\begin{array}{l}
\text{surface, } t = \text{hit}(\text{ray}, t_{\text{Min}}, t_{\text{Max}}); \\
\text{if (surface }\neq \text{ null) return surface.color();} \\
\text{else return black;}
\end{array}
\]


Shading

- Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction
  (for each of many lights)
  - surface normal
  - surface parameters
    (color, shininess, …)
- More on this in the next lecture

Image so far

\[
\text{trace}(\text{Ray } \text{ray}, t_{\text{Min}}, t_{\text{Max}}) \\
\begin{array}{l}
\text{surface, } t = \text{hit}(\text{ray}, t_{\text{Min}}, t_{\text{Max}}); \\
\text{if (surface }\neq \text{ null) }
\begin{array}{l}
\text{point } = \text{ray.evaluate}(t); \\
\text{normal } = \text{surface.getNormal(point);} \\
\text{return surface.shade(\text{ray, point,}} \\
\text{normal, light);}
\end{array}
\end{array}
\]

\[
\text{...}
\]

\[
\text{shade}(\text{ray, point, normal, light}) \\
\begin{array}{l}
\text{v_E } = \text{normalize}(\text{ray.direction);} \\
\text{v_L } = \text{normalize(\text{light.pos - point;)} } \\
\text{// compute shading}
\end{array}
\]

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!
shade(ray, point, normal, light) {
    shadRay = (point, light.pos - point);
    if (shadRay not blocked) {
        v_E = –normalize(ray.direction);
        v_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: all surface receive a bit more light
    • just add a constant "ambient" color to the shading…
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;
}
```

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
  - the whole scene
  - often need to be able to limit the range (e.g. shadow rays)
- Once you have the visible intersection, compute the color
  - this is an object that’s associated with the object you hit
  - 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: “Get it Right, Then Make it Fast”