CS4620/5620: Lecture 34

Ray Tracing

Announcements

• 4621
  • Friday (animation): Nov 16

• PPA2 due today

• HW 3 due right before class on Friday
Ray tracing algorithm

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

Generating eye rays—perspective

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

\[
\begin{align*}
  s &= e + u\mathbf{u} + v\mathbf{v} - d\mathbf{w} \\
  p &= e; \quad d = s - e \\
  r(t) &= p + td
\end{align*}
\]

Parallel projection
same direction, different origins

Perspective projection
same origin, different directions

\( s = e + u\mathbf{u} + v\mathbf{v} - d\mathbf{w} \)

\( p = e; \quad d = s - e \)

\( r(t) = p + td \)
**Pixel-to-image mapping**

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

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

**PA3A camera**

- `viewPoint == e`
- `projNormal == w, viewUp == up`
  - Compute \(u, v\) from the above

- \(l = -\text{viewWidth}/2\)
- \(r = +\text{viewWidth}/2\)
- \(n_x = \text{imageWidth}\)
Ray intersection

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 for general
  \[ \|x\| = 1 \iff \|x\|^2 = 1 \]
  \[ f(x) = x \cdot x - 1 = 0 \]

- 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:
  \[ t = \frac{-d \cdot p \pm \sqrt{(d \cdot p)^2 - (d \cdot d)(p \cdot p - 1)}}{d \cdot d} \]
  \[ t = -d \cdot p \pm \sqrt{(d \cdot p)^2 - p \cdot p + 1} \]
  - simpler form holds when \( d \) is a unit vector
  but don’t necessarily assume this (for potential performance reasons)
  - discriminant intuition?
  - use the unit-vector form to make the geometric interpretation
Ray-sphere intersection: geometric

\[ t_m = -p \cdot d \]
\[ l_m^2 = p \cdot p - (p \cdot d)^2 \]
\[ \Delta t = \sqrt{1 - l_m^2} \]
\[ = \sqrt{(p \cdot d)^2 - p \cdot p + 1} \]
\[ t_{0,1} = t_m \pm \Delta t = -p \cdot d \pm \sqrt{(p \cdot d)^2 - p \cdot p + 1} \]

Normal for sphere
Image so far

- With eye ray generation and sphere intersection

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);
  }

Transforming objects

- In modeling, we’ve seen the usefulness of transformations
  - How to do the same in RT?

- Take spheres as an example: want to support transformed spheres

- Option 1: transform sphere into world coordinates
  - Write code to intersect arbitrary ellipsoids

- Option 2: transform ray into sphere’s coordinates
  - Then just use existing sphere intersection routine
Intersecting transformed objects

Implementing RT transforms

- Create wrapper object “TransformedSurface”
  - Has a transform M and a reference to a surface S
  - To intersect:
    - Transform ray to local coords (by inverse of M)
    - Call surface.intersect
    - Transform hit data back to global coords (by M)
      - Intersection point
      - Surface normal
      - Any other relevant data (maybe none)
Transforming normal vectors

- Transforming surface normals
  - differences of points (and therefore tangents) transform OK
  - normals do not --> use inverse transpose matrix

\[
\begin{align*}
  t \cdot n &= t^T n = 0 \\
  \text{want: } M t \cdot X n &= t^T M^T X n = 0 \\
  \text{so set } X &= (M^T)^{-1} \\
  \text{then: } M t \cdot X n &= t^T M^T (M^T)^{-1} n = t^T n = 0
\end{align*}
\]

Groups, transforms, hierarchies

- Often it’s useful to transform several objects at once
  - Add “SurfaceGroup” as a subclass of Surface
    - Has a list of surfaces
    - Returns closest intersection

- With TransformedSurface and SurfaceGroup you can put transforms below transforms
  - Voilà! A transformation hierarchy.
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!

\[ p_x + t_{x_{\min}} d_x = x_{\min} \]
\[ t_{x_{\min}} = \frac{(x_{\min} - p_x)}{d_x} \]

\[ p_y + t_{y_{\min}} d_y = y_{\min} \]
\[ t_{y_{\min}} = \frac{(y_{\min} - p_y)}{d_y} \]
Intersecting intersections

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

\[
\begin{align*}
    t_{\text{min}} &= \max(t_{x_{\text{min}}}, t_{y_{\text{min}}}) \\
    t_{\text{max}} &= \min(t_{x_{\text{max}}}, t_{y_{\text{max}}})
\end{align*}
\]

Does it work?

- \(d_x\) positive or negative
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 \( t \):
  \[ (p + td - a) \cdot n = 0 \]
  \[ t = \frac{(a - p) \cdot n}{d \cdot n} \]

Ray-triangle intersection

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

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

- See book for a more efficient method based on linear systems
  - (don’t need this for PA3A anyhow—but stash away for PA3B)
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);
  }
...

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