Viewing and Ray Tracing

CS 4620 Lecture 4
Projection

- To render an image of a 3D scene, we *project* it onto a plane
- Most common projection type is *perspective projection*
Two approaches to rendering
Two approaches to rendering

\begin{verbatim}
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
\end{verbatim}

object order
  or
rasterization
Two approaches to rendering

\begin{align*}
\textbf{object order} & \quad \textbf{image order} \\
\text{or} & \quad \text{or} \\
\textbf{rasterization} & \quad \textbf{ray tracing}
\end{align*}
Two approaches to rendering

for each object in the scene {
    for each pixel in the image {
        if (object affects pixel) {
            do something
        }
    }
}

object order
or
rasterization

for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}

image order
or
ray tracing

We will do this first
Ray tracing idea

• Start with a pixel—what belongs at that pixel?
• Set of points that project to a point in the image: a ray
Ray tracing idea

- Start with a pixel—what belongs at that pixel?
- Set of points that project to a point in the image: a ray
Ray tracing idea

viewer (eye)

light source

objects in scene
Ray tracing idea
Ray tracing idea

viewer (eye)

light source

viewing ray

visible point

objects in scene
Ray tracing idea

- Viewer (eye)
- Viewing ray
- Visible point
- Light source
- Illumination
- Objects in scene
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—planar projection

• Ray origin (varying): pixel position on viewing window
• Ray direction (constant): view direction
Generating eye rays—perspective

- Ray origin (constant): viewpoint
- Ray direction (varying): toward pixel position on viewing window
Software interface for cameras

- Key operation: generate ray for image position

```java
class Camera {
    ...
    Ray generateRay(int col, int row);
}
```

- Modularity problem: Camera shouldn’t have to worry about image resolution
  - better solution: normalized coordinates

```java
class Camera {
    ...
    Ray generateRay(float u, float v);
}
```
Specifying views in Ray 1

<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>6</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>

<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>3</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
Pixel-to-image mapping

- One last detail: exactly where are pixels located?

\[
\begin{align*}
    u &= (i + 0.5)/n_x \\
    v &= (j + 0.5)/n_y
\end{align*}
\]
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 \( \alpha d \) doesn’t change ray (\( \alpha > 0 \))
Ray-sphere intersection: algebraic

• Condition 1: point is on ray
  \[ \mathbf{r}(t) = \mathbf{p} + t\mathbf{d} \]

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

• Substitute:
  \[ (\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - 1 = 0 \]
  – 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 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

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

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

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

• In plane, triangle is the intersection of 3 half spaces
Deciding about insideness

- Need to check whether hit point is inside 3 edges
  - easiest to do in 2D coordinates on the plane
- Will also need to know where we are in the triangle
  - for textures, shading, etc. … next couple of lectures
- Efficient solution: transform to coordinates aligned to the triangle
Barycentric coordinates

• A coordinate system for triangles
  – algebraic viewpoint:
    \[ p = \alpha a + \beta b + \gamma c \]
    \[ \alpha + \beta + \gamma = 1 \]
  – geometric viewpoint (areas):
• Triangle interior test:
  \[ \alpha > 0; \ \beta > 0; \ \gamma > 0 \]
Barycentric coordinates

- A coordinate system for triangles
  - geometric viewpoint: distances
  - linear viewpoint: basis of edges

\[
\alpha = 1 - \beta - \gamma \\
p = a + \beta(b - a) + \gamma(c - a)
\]
Barycentric coordinates

• Linear viewpoint: basis for the plane

– in this view, the triangle interior test is just

\[ \beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1 \]
Barycentric ray-triangle intersection

• Every point on the plane can be written in the form:
  \[ \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a}) \]
  for some numbers \( \beta \) and \( \gamma \).

• If the point is also on the ray then it is
  \[ \mathbf{p} + t \mathbf{d} \]
  for some number \( t \).

• Set them equal: 3 linear equations in 3 variables
  \[ \mathbf{p} + t \mathbf{d} = \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a}) \]
  ...solve them to get \( t, \beta, \) and \( \gamma \) all at once!
Barycentric ray-triangle intersection

\[ p + td = a + \beta(b - a) + \gamma(c - a) \]
\[ \beta(a - b) + \gamma(a - c) + td = a - p \]

\[
\begin{bmatrix}
  a - b & a - c & d
\end{bmatrix}
\begin{bmatrix}
  \beta \\
  \gamma \\
  t
\end{bmatrix}
= 
\begin{bmatrix}
  a - p
\end{bmatrix}
\]

\[
\begin{bmatrix}
  x_a - x_b & x_a - x_c & x_d \\
  y_a - y_b & y_a - y_c & y_d \\
  z_a - z_b & z_a - z_c & z_d
\end{bmatrix}
\begin{bmatrix}
  \beta \\
  \gamma \\
  t
\end{bmatrix}
= 
\begin{bmatrix}
  x_a - x_p \\
  y_a - y_p \\
  z_a - z_p
\end{bmatrix}
\]

Cramer’s rule is a good fast way to solve this system
(see text Ch. 2 and Ch. 4 for details)
Ray intersection in software

- All surfaces need to be able to intersect rays with themselves.

```java
class Surface {
    ...
    abstract boolean intersect(IntersectionRecord result, Ray r);
}
```

```java
class IntersectionRecord {
    float t;
    Vector3 hitLocation;
    Vector3 normal;
    ...
}
```
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);
    }
Ray intersection in software

- Scenes usually have many objects
- Need to find the first intersection along the ray
  - that is, the one with the smallest positive $t$ value
- Loop over objects
  - ignore those that don’t intersect
  - keep track of the closest seen so far
  - Convenient to give rays an ending $t$ value for this purpose (then they are really segments)
Intersection against many shapes

- The basic idea is:

```java
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)