Ray Casting

Based on slides of Thomas Funkhouser

3D Rendering

- The color of each pixel on the view plane depends on the radiance emanating from visible surfaces

Simplest method is ray casting

Rays through view plane

Eye position

View plane

1
Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance
Ray Casting

• Simple implementation:

```java
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```
Constructing Ray Through a Pixel

2D Example

\[ \Theta = \text{frustum half-angle} \]
\[ d = \text{distance to view plane} \]
\[ \text{right} = \text{towards x up} \]

\[ P_1 = P_0 + d \cdot \text{towards} - d \cdot \tan(\Theta) \cdot \text{right} \]
\[ P_2 = P_0 + d \cdot \text{towards} + d \cdot \tan(\Theta) \cdot \text{right} \]

\[ P = P1 + (i/\text{width} + 0.5) \cdot 2 \cdot d \cdot \tan(\Theta) \cdot \text{right} \]
\[ V = (P - P_0) / \|P - P_0\| \]

Ray: \( P = P_0 + tV \)
Ray Casting

• Simple implementation:

```java
Image RayCast(Camera camera, Scene scene, int width, int height) {
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

Ray-Scene Intersection

• Intersections with geometric primitives
  ◦ Sphere
  ◦ Triangle
  ◦ Groups of primitives (scene)

• Acceleration techniques
  ◦ Bounding volume hierarchies
  ◦ Spatial partitions
    • Uniform grids
    • Octrees
    • BSP trees
Ray-Sphere Intersection

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)

Substituting for \( P \), we get:
\[ |P_0 + tV - O|^2 - r^2 = 0 \]

Solve quadratic equation:
\[ at^2 + bt + c = 0 \]
where:
\( a = 1 \)
\( b = 2 V \cdot (P_0 - O) \)
\( c = |P_0 - O|^2 - r^2 = 0 \)

\( P = P_0 + tV \)

Ray-Sphere Intersection I

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)

Substituting for \( P \), we get:
\[ |P_0 + tV - O|^2 - r^2 = 0 \]

Solve quadratic equation:
\[ at^2 + bt + c = 0 \]
where:
\( a = 1 \)
\( b = 2 V \cdot (P_0 - O) \)
\( c = |P_0 - O|^2 - r^2 = 0 \)

\( P = P_0 + tV \)
Ray-Sphere Intersection II

Ray: \( P = P_0 + tV \)
Sphere: \( |P - O|^2 - r^2 = 0 \)

\[ L = O - P_0 \]
\[ t_{ca} = L \cdot V \]
if \( t_{ca} < 0 \) return 0
\[ d^2 = L \cdot L - t_{ca}^2 \]
if \( d^2 > r^2 \) return 0
\[ t_{hc} = \sqrt{r^2 - d^2} \]
\[ t = t_{ca} - t_{hc} \text{ and } t_{ca} + t_{hc} \]
\[ P = P_0 + tV \]

Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

\[ N = (P - O) / ||P - O|| \]
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Ray-Triangle Intersection

- First, intersect ray with plane
- Then, check if point is inside triangle
**Ray-Plane Intersection**

Ray: \( P = P_0 + tV \)
Plane: \( P \cdot N + d = 0 \)

Substituting for \( P \), we get:
\[(P_0 + tV) \cdot N + d = 0\]

Solution:
\[t = -(P_0 \cdot N + d) / (V \cdot N)\]
\[P = P_0 + tV\]

**Algebraic Method**

**Ray-Triangle Intersection I**

- Check if point is inside triangle algebraically

For each side of triangle
\[V_1 = T_1 - P\]
\[V_2 = T_2 - P\]
\[N_1 = V_2 \times V_1\]
Normalize \( N_1 \)
\[d_1 = -P_0 \cdot N_1\]
if \((P \cdot N_1 + d_1) < 0\)
return FALSE;
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

Compute $\alpha$, $\beta$:

$$P = \alpha (T_2 - T_1) + \beta (T_3 - T_1)$$

Check if point inside triangle.
- $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$
- $\alpha + \beta \leq 1$

Other Ray-Primitive Intersections

- Cone, cylinder, ellipsoid:
  - Similar to sphere
- Box
  - Intersect 3 front-facing planes, return closest
- Convex polygon
  - Same as triangle (check point-in-polygon algebraically)
- Concave polygon
  - Same plane intersection
  - More complex point-in-polygon test
Ray-Scene Intersection

- Find intersection with front-most primitive in group

Intersection FindIntersection(Ray ray, Scene scene)
{
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t < min_t) then
            min_primitive = primitive
            min_t = t
    }
    return Intersection(min_t, min_primitive)
}

Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
  - Triangle
  - Groups of primitives (scene)

- Acceleration techniques
  - Bounding volume hierarchies
  - Spatial partitions
    - Uniform grids
    - Octrees
    - BSP trees

Next Time!
Summary

- Writing a simple ray casting renderer is easy
  - Generate rays
  - Intersection tests
  - Lighting calculations

```java
Image RayCast(Camera camera, Scene scene, int width, int height) {
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

### Constructing Ray Through a Pixel

#### Ray: \( P = P_0 + tV \)
We need to determine $V_x$ and $V_y$

Ray: $P = P_0 + tV$

Camera Coordinate System

- Find the transformation matrix $M$ that rotate the world coordinate system to the camera coordinate system $(V_x, V_y, V_z)$ (normalized)

$M(0,0,1) = V_z$
Camera Coordinate System

- The vector X and Y are rotated by M

\[
\begin{align*}
M (0,0,1) &= V_z \\
\end{align*}
\]

The definition of the Matrix M

Let \( C_x \) and \( S_x \) denote \( \sin(x) \), \( \cos(x) \), respectively

\[
\begin{align*}
&\text{Rotate around z} \\
(0,0,1) \cdot \begin{bmatrix} C_z & S_z & 0 \\ -S_z & C_z & 0 \\ 0 & 0 & 1 \end{bmatrix} = (0,0,1) \\
&M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C_x & S_x \\ 0 & -S_x & C_x \end{bmatrix} \cdot \begin{bmatrix} C_y & 0 & S_y \\ 0 & 1 & 0 \\ -S_y & 0 & C_y \end{bmatrix} = \begin{bmatrix} C_y & 0 & S_y \\ -S_xSy & C_x & SxCy \\ -CxSy & -S_x & CxCy \end{bmatrix}
\end{align*}
\]
The definition of the Matrix M

Since:

\[(0,0,1) \cdot M = (-C_xS_y, -S_x, C_xC_y) = (V_z.x, V_z.y, V_z.z) = V_z = (a, b, c).\]

We get:

\[a = -C_xS_y; \quad b = -S_x; \quad c = C_xC_y,\]

or

\[S_x = -b; \quad C_x = \sqrt{1 - \text{sqr}(S_x)}; \quad S_y = -a/C_x; \quad C_y = c/C_x;\]

Compute the Camera Coordinate System

Now, use \(M\) to rotate the world coordinate vectors:

\[V_x = (1,0,0) \cdot M\]
\[V_y = (0,1,0) \cdot M\]
\[V_z = (0,0,1) \cdot M\]

Note that the vector \(V\) is normalized.
We need to determine $V_x$ and $V_y$

Let $f$ be the distance between the eye $E$ and the plane along $V_z$, and $w$ and $h$ the lengths of half the screen size.

$$P = E + V_z \cdot f$$

$$P_0 = P - w \cdot V_x - h \cdot V_y$$

The main loop

```java
Image RayCast(Camera camera, Scene scene, int width, int height) {
   Image image = new Image(width, height);
   Set $P_0$ (as in the previous slide);
   for (int $i = 0; i < height; i++) {
      $p = P_0$;
      for (int $j = 0; j < width; j++) {
         Ray ray = $E + t \cdot (p - E)$;
         Intersection hit = FindIntersection(ray, scene);
         image[$i$][$j$] = GetColor(hit);
         $p += V_x$; // move one pixel along the vector $V_x$
      }
      $P_0 += V_y$; // move one pixel along the vector $V_y$
   }
   return image;
}
```