Ray Casting

Jason Lawrence

CS 4810: Graphics

Acknowledgment: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin
Traditional Pinhole Camera

• The film sits behind the pinhole of the camera.
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
- Rays come in from the outside, pass through the pinhole, and hit the film plane.
Traditional Pinhole Camera

- The film sits behind the pinhole of the camera.
- Rays come in from the outside, pass through the pinhole, and hit the film plane.

Photograph is upside down
Virtual Camera

• The film sits in front of the pinhole of the camera.
Virtual Camera

- The film sits in front of the pinhole of the camera.
- Rays come in from the outside, pass through the film plane, and hit the pinhole.
Virtual Camera

• The film sits in front of the pinhole of the camera.
• Rays come in from the outside, pass through the film plane, and hit the pinhole.
Overview

• Ray Casting
  o What do we see?
  o How does it look?
Ray Casting

• Rendering model

• Intersections with geometric primitives
  o Sphere
  o Triangle

• Acceleration techniques
  o Bounding volume hierarchies
  o Spatial partitions
    » Uniform grids
    » Octrees
    » BSP trees
Ray Casting

- We invert the process of image generation by sending rays **out** from the pinhole, and then we find the first intersection of the ray with the scene.
Ray Casting

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

- Where are we looking?
- What are we seeing?
- What does it look like?
Constructing a Ray Through a Pixel
Constructing a Ray Through a Pixel

The ray has to originate at $P_0$, the position of the camera. So the equation for the ray is of the form:

$$\text{Ray} = P_0 + tV$$
Constructing a Ray Through a Pixel

If the ray passes through the point $P$, then the solution for $V$ is:

$$V = \frac{(P - P_0)}{||P - P_0||}$$
If P represents the (i,j)-th pixel of the image, what is the position of P?
Constructing Ray Through a Pixel

• 2D Example: Side view of camera at $P_0$

  • What is the position of the $i$-th pixel $P[i]$?

  $\theta = \text{frustum half-angle (given), or field of view}$
  
  $d = \text{distance to view plane (arbitrary = you pick)}$
Constructing Ray Through a Pixel

• 2D Example: Side view of camera at $P_0$

What is the position of the $i$-th pixel $P[i]$?

$\theta =$ frustum half-angle (given), or field of view
$d =$ distance to view plane (arbitrary = you pick)

$P_1 = P_0 + d \cdot \text{towards} - d \cdot \text{tan(} \theta \text{)} \cdot \text{up}$
$P_2 = P_0 + d \cdot \text{towards} + d \cdot \text{tan(} \theta \text{)} \cdot \text{up}$
Constructing Ray Through a Pixel

- 2D Example: Side view of camera at $P_0$

  **What is the position of the $i$-th pixel?**

  $\theta =$ frustum half-angle (given), or field of view
  $d =$ distance to view plane (arbitrary = you pick)

  
  \[
  P_1 = P_0 + d\text{towards} - d\text{tan}(\theta)\text{up}
  \]
  
  \[
  P_2 = P_0 + d\text{towards} + d\text{tan}(\theta)\text{up}
  \]
  
  \[
  P[i] = P_1 + \left((i+0.5)/\text{height}\right)(P_2-P_1)
  = P_1 + \left((i+0.5)/\text{height}\right)2d\text{tan}(\theta)\text{up}
  \]
### Constructing Ray Through a Pixel

- **2D Example:**
  - The ray passing through the $i$-th pixel is defined by:

\[
\text{Ray} = P_0 + tV
\]

- **Where:**
  - $P_0$ is the camera position
  - $V$ is the direction to the $i$-th pixel:
    \[
    V = (P[i] - P_0) / \|P[i] - P_0\|
    \]
  - $P[i]$ is the $i$-th pixel location:
    \[
    P[i] = P_1 + \left((i+0.5) / \text{height}\right) \cdot (P_2 - P_1)
    \]
  - $P_1$ and $P_2$ are the endpoints of the view plane:
    - $P_1 = P_0 + d \cdot \text{towards} - d \cdot \tan(\theta) \cdot \text{up}$
    - $P_2 = P_0 + d \cdot \text{towards} + d \cdot \tan(\theta) \cdot \text{up}$
Ray Casting

• 2D implementation:

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

- Figuring out how to do this in 3D is assignment 2

\[ \text{2} \times d \times \tan(\theta) \]
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 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

• Acceleration techniques
  • Bounding volume hierarchies
  • Spatial partitions
    » Uniform (Voxel) grids
    » Octrees
    » BSP trees
Ray-Sphere Intersection

\[ \text{Ray: } P = P_0 + tV \]
\[ \text{Sphere: } |P - O|^2 - r^2 = 0 \]
Ray-Sphere Intersection I

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

Substituting for \( P \), we get:
\(|P_0 + tV - O|^2 - r^2 = 0\)
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:
$$a = 1$$
$$b = 2 V \cdot (P_0 - O)$$
$$c = |P_0 - O|^2 - r^2 = 0$$

Algebraic Method
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:
\( a = 1 \)
\( b = 2V \cdot (P_0 - O) \)
\( c = |P_0 - O|^2 - r^2 = 0 \)

Generally, there are two solutions to the quadratic equation, giving rise to points \( P \) and \( P' \).
You want to return the first hit.
Ray-Sphere Intersection II

Ray: \( \mathbf{P} = \mathbf{P}_0 + t \mathbf{V} \)

Sphere: \( |\mathbf{P} - \mathbf{O}|^2 - r^2 = 0 \)

\( \mathbf{L} = \mathbf{O} - \mathbf{P}_0 \)

Geometric Method
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$ (assumes $V$ is unit length)
Ray-Sphere Intersection II

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

\[ L = O - P_0 \]

\[ t_{ca} = L \cdot V \text{ (assumes } V \text{ is unit length)} \]

\[ d^2 = L \cdot L - t_{ca}^2 \]

if \( d^2 > r^2 \) return 0
Ray-Sphere Intersection II

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

\[ L = O - P_0 \]

\( t_{ca} = L \cdot V \) (assumes \( V \) is unit length)

\( d^2 = L \cdot L - t_{ca}^2 \)
if \( d^2 > r^2 \) return 0

\( t_{hc} = \sqrt{r^2 - d^2} \)
\( t = t_{ca} - t_{hc} \) and \( t_{ca} + t_{hc} \)
Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

\[ N = \frac{(P - O)}{||P - O||} \]
Ray-Scene Intersection

- Intersections with geometric primitives
  - Sphere
    - Triangle

- 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:
\[
t = -\frac{(P_0 \cdot N + d)}{(V \cdot N)}
\]
Ray-Triangle Intersection I

• Check if point is inside triangle algebraically

For each side of triangle

\[ V_1 = T_1 - P_0 \]
\[ V_2 = T_2 - P_0 \]
\[ N_1 = V_2 \times V_1 \]

if \((P - P_0) \cdot N_1 < 0\)
return FALSE;
end
Ray-Triangle Intersection II

• Check if point is inside triangle parametrically

Every point P inside the triangle can be expressed as:

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

where:

\[ 0 \leq \alpha \leq 1 \text{ and } 0 \leq \beta \leq 1 \]

\[ \alpha + \beta \leq 1 \]
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

Solve for $\alpha$, $\beta$ such that:

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

Check if point inside triangle.

$$0 \leq \alpha \leq 1 \text{ and } 0 \leq \beta \leq 1$$

$$\alpha + \beta \leq 1$$
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 = \infty
    min_shape = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 and t < min_t) then
            min_shape = primitive
            min_t = t
    }
}
return Intersection(min_t, min_shape)
Next Lecture

- Intersections with geometric primitives
  - Sphere
  - Triangle

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