Ray Casting

Thomas Funkhouser
Princeton University
COS 426, Spring 2004

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;
}
```

3D Rendering

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

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
  - \( \theta = \) frustum half-angle
  - \( d = \) distance to view plane
  - \( \text{right} = \text{towards x up} \)

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

\[
P = P_1 + ((\text{width} + 0.5) \cdot (P_2 - P_1)) \\
v = (P - P_2) / |P - P_1|
\]

---

**Ray-Casting**

- Simple implementation:

```
rayCastRayThroughPixel(camera, i, j) 
| Image image = new Image(width, height); 
| for (int i = 0; i < width; i++) 
| for (int j = 0; j < height; j++) 
| Ray ray = rayCastRayThroughPixel(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**

- Algebraic Method

\[
ray: P = P_0 + tv \\
sphere: |P - O|^2 - r^2 = 0
\]

Substituting for \( P \), we get:

\[
|P_0 + tv - O|^2 - r^2 = 0
\]

Solve quadratic equation:

\[
a t^2 + bt + c = 0
\]

where:

\[
a = 1 \\
b = 2 \cdot v \cdot (P_0 - O) \\
c = |P_0 - O|^2 - r^2 = 0
\]

\[
P = P_0 + tv
\]
Ray-Sphere Intersection

- Need normal vector at intersection for lighting calculations

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

Ray-Plane Intersection

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

- Check if point is inside triangle parametrically

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

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, Scene) {
    min_t = infinity
    min_primitive = NULL
    For each primitive in scene {
        t = Intersect(ray, primitive);
        if (t > 0 && 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

Bounding Volumes

- Check for intersection with simple shape first

```
\[
\text{Bounding Volume Hierarchies:}\ 
\text{min\_primitive = \text{NULL}} \\
\text{min\_primitive = \text{primitive}} \\
\text{Spatial partitions:}\ 
\text{D} \\
\text{Triangle} \\
\text{For each primitive in scene {}} \\
\text{Check if point inside triangle.} \\
\text{0 \leq \alpha \leq 1\ and \ 0 \leq \beta \leq 1\} \\
\text{\alpha + \beta \leq 1}\}
```
Bounding Volumes

- Check for intersection with simple shape first
  - If ray doesn’t intersect bounding volume, then it doesn’t intersect its contents

Bounding Volume Hierarchies I

- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

Bounding Volume Hierarchies III

- Sort hits & detect early termination

```c
FindIntersection(Ray ray, Node node)
{
  // Find intersections with child node bounding volumes
  ... // Sort intersections front to back
  // Process intersections (checking for early termination)
  min_t = infinity;
  for each intersected child i {
    if (min_t < bv[i].t) break;
    shape_i = FindIntersection(ray, child);
    if (shape_i < min_t) { min_t = shape_i; }
  }
  return min_t;
}
```

Bounding Volumes

- Check for intersection with simple shape first
  - If ray doesn’t intersect bounding volume, then it doesn’t intersect its contents

Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

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
**Uniform Grid**

- Construct uniform grid over scene
  - Index primitives according to overlaps with grid cells

**Uniform Grid**

- Trace rays through grid cells
  - Fast
  - Incremental

- Only check primitives in intersected grid cells

**Potential problem:**
- **Too little benefit** if grid is too coarse
- **Too much cost** if grid is too fine

**Uniform Grid**

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells

**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

**Octree**

- Construct adaptive grid over scene
  - Recursively subdivide box-shaped cells into 8 octants
  - Index primitives by overlaps with cells

**Octree**

- Trace rays through neighbor cells
  - Fewer cells
  - More complex neighbor finding

- Trade-off fewer cells for more expensive traversal
Ray-Scene Intersection

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

- Acceleration techniques
  - Spatial partitions
    - Uniform grid
    - Octrees
    - BSP trees

Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Every cell is a convex polyhedron

• Recursively partition space by planes

- Memory coherence
  - BSP construction enables simple front-to-back traversal

• Trace rays by recursion on tree

Binary Space Partition (BSP) Tree

• Simple recursive algorithms
  - Example: point finding

RayTreeIntersect(Ray ray, Node node, double min, double max)
{
  if (Node is a leaf)
    return intersection of closest primitive in cell, or NULL if none
  else
    dist = distance of the ray point to split plane of node
    near_child = child of node that contains the origin of Ray
    far_child = other child of node
    if the interval to look is on near side
      return RayTreeIntersect(ray, near_child, min, max)
    else if the interval to look is on far side
      return RayTreeIntersect(ray, far_child, min, max)
    else if the interval to look is on both side
      return RayTreeIntersect(ray, near_child, min, max)
    else return RayTreeIntersect(ray, far_child, dist, max)
}
Acceleration

- Intersection acceleration techniques are important
  - Bounding volume hierarchies
  - Spatial partitions
- General concepts
  - Sort objects spatially
  - Make trivial rejections quick
  - Utilize coherence when possible

Expected time is sub-linear in number of primitives

Summary

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

Next Time is Illumination!

Heckbert’s business card ray trace

```c
typedef struct{double x,y,z;vec c;black} Point; Sphere* intersect(Point P, ray R)
{ best=0; tmin=1e30; sph=5; while(sph-->sph) //for (int j = 0; j < height; j++) {
  P = sph+5; //get the intersection point of the sphere with the ray
  if(d<0) //if the ray intersects the sphere
    N = vunit(-P); //calculate the normal at the intersection point
    d = dot(N, P); //calculate the distance from the origin to the intersection point
    if(d < tmin) //if the distance is smaller than the previous minimum
      return (N, d); //return the new minimum distance and the normal
    tmin = d; //update the minimum distance
  } //end while
  return (black, 0); //if the ray does not intersect the sphere
}```