Intersection and Acceleration

Michael Kazhdan
(601.457/657)

HB Ch. 14.1, 14.2
FvDFH 16.1, 16.2
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:

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
Ray-Triangle Intersection

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

Ray: \( p(t) = p_0 + t \cdot \vec{v}, \quad (0 \leq t < \infty) \)
Plane: \( \Phi(p) = \langle p, \vec{n} \rangle - d = 0 \)

Substituting for \( p \), we get:
\( \Phi(t) = \langle p_0 + t \cdot \vec{v}, \vec{n} \rangle - d = 0 \)

Solution:
\[
t = -\frac{\langle p_0, \vec{n} \rangle - d}{\langle \vec{v}, \vec{n} \rangle}
\]
Check if point is inside triangle parametrically

A point \( p \) is inside the triangle iff. it can be expressed as the weighted average of the corners:

\[
p = \alpha \cdot T_1 + \beta \cdot T_2 + \gamma \cdot T_3
\]

where:
\[
\alpha + \beta + \gamma = 1
\]
\[
0 \leq \alpha, \beta, \gamma
\]
Ray-Triangle Intersection II

Check if point is inside triangle parametrically

- Intersect the triangle's plane with the ray to get $p$.
- Solve for $\alpha, \beta, \gamma$ such that:
  \[ p = \alpha \cdot T_1 + \beta \cdot T_2 + \gamma \cdot T_3 \]
- Check if $p$ is in the triangle:
  \[ 0 \leq \alpha, \beta, \gamma \]
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

In general, given \( p \in \mathbb{R}^3 \) and given three points \( \{T_1, T_2, T_3\} \subset \mathbb{R}^3 \) (in general position) we can solve for \( \alpha, \beta, \gamma \in \mathbb{R} \) such that:

\[
p = \alpha T_1 + \beta T_2 + \gamma T_3
\]

If \( p \) is in the plane spanned by \( \{T_1, T_2, T_3\} \):

\[
\alpha + \beta + \gamma = 1
\]

If \( p \) is inside the triangle with vertices \( \{T_1, T_2, T_3\} \):

\[
\alpha, \beta, \gamma \geq 0
\]
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

In general, given \( p \in \mathbb{R}^3 \) and given three points \( \{T_1, T_2, T_3\} \subset \mathbb{R}^3 \) (in general position) we can solve for \( \alpha, \beta, \gamma \in \mathbb{R} \) such that:

\[
p = \alpha T_1 + \beta T_2 + \gamma T_3
\]

To get \( \alpha, \beta, \gamma \), solve the system:

\[
\begin{pmatrix}
T_1^x & T_2^x & T_3^x \\
T_1^y & T_2^y & T_3^y \\
T_1^z & T_2^z & T_1^1
\end{pmatrix}
\begin{pmatrix}
\alpha \\
\beta \\
\gamma
\end{pmatrix} =
\begin{pmatrix}
p^x \\
p^y \\
p^z
\end{pmatrix}
\]
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

In general, given \( p \in \mathbb{R}^3 \) and given three points \( \{T_1, T_2, T_3\} \subset \mathbb{R}^3 \) (in general position) we can solve for \( \alpha, \beta, \gamma \in \mathbb{R} \) such that:

\[
p = \alpha T_1 + \beta T_2 + \gamma T_3
\]

To get \( \alpha, \beta, \gamma \), solve the system:

\[
\begin{pmatrix}
T_1^x & T_2^x & T_3^x \\
T_1^y & T_2^y & T_3^y \\
T_1^z & T_2^z & T_1^z
\end{pmatrix}
\begin{pmatrix}
\alpha \\
\beta \\
\gamma
\end{pmatrix} =
\begin{pmatrix}
p^x \\
p^y \\
p^z
\end{pmatrix}
\]

This will fail if the vertices \( \{T_1, T_2, T_3\} \) lie in a plane through the origin.
Ray-Triangle Intersection II

- Check if point is inside triangle parametrically

In general, given \( p \in \mathbb{R}^3 \) and given three points \( \{T_1, T_2, T_3\} \subset \mathbb{R}^3 \) (in general position) we can solve for \( \alpha, \beta, \gamma \in \mathbb{R} \) such that:

\[
p = \alpha T_1 + \beta T_2 + \gamma T_3
\]

If \( p \) is in the plane spanned by \( \{T_1, T_2, T_3\} \) we can translate so that \( T_1 \) is at the origin and solve for \( \beta, \gamma \):

\[
\begin{pmatrix}
T_2^x - T_1^x & T_3^x - T_1^x \\
T_2^y - T_1^y & T_3^y - T_1^y \\
T_2^z - T_1^z & T_3^z - T_1^z
\end{pmatrix}
\begin{pmatrix}
\beta \\
\gamma
\end{pmatrix}
=
\begin{pmatrix}
p^x - T_1^x \\
p^y - T_1^y \\
p^z - T_1^z
\end{pmatrix}
\]
• Check if point is inside triangle parametrically

In general, given \( p \in \mathbb{R}^3 \) and given three points \( \{T_1, T_2, T_3\} \subset \mathbb{R}^3 \) (in general position) we can solve for \( \alpha, \beta, \gamma \in \mathbb{R} \) such that:

\[
p = \alpha T_1 + \beta T_2 + \gamma T_3
\]

If \( p \) is in the plane spanned by \( \{T_1, T_2, T_3\} \), we can solve for \( \beta, \gamma \):

\[
\begin{pmatrix}
T_2^x - T_1^x & T_3^x - T_1^x \\
T_2^y - T_1^y & T_3^y - T_1^y \\
T_2^z - T_1^z & T_3^z - T_1^z
\end{pmatrix}
\begin{pmatrix}
\beta \\
\gamma
\end{pmatrix}
= \begin{pmatrix}
p^x - T_1^x \\
p^y - T_1^y \\
p^z - T_1^z
\end{pmatrix}
\]

This is an over-constrained system but a solution exists since \( p \) is in the plane spanned by \( \{T_1, T_2, T_3\} \).

After solving for \( \beta \) and \( \gamma \), we can set:

\[
\alpha = 1 - \beta - \gamma
\]
Other Ray-Primitive Intersections

• Cone, cylinder, ellipsoid:
  ◦ Similar to sphere

• Box
  ◦ Intersect 3 front-facing planes, return closest

• Convex polygon
  » Find the intersection of the ray with the plane
  » Check that the intersection is above every triangle generated by the ray source and polygon edge.

• Concave polygon
  ◦ Same plane intersection
  ◦ More complex point-in-polygon test
Ray-Scene Intersection

• Intersections with geometric primitives
  ◦ Sphere
  ◦ Triangle

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

A direct (naïve) approach generates the image:

Intersection FindIntersection( Ray ray, Scene scene )
{
    ( min_t , min_shape ) = ( -1 , NULL )
    For each primitive in scene
    {
        t = Intersect( ray , primitive );
        if( t>0 and (t < min_t or min_t<0 ) )
            min_shape = primitive
            min_t = t
    }
    return Intersection( min_t , min_shape )
}
Overview

• Acceleration techniques
  ◦ Bounding volume hierarchies
  ◦ Space partitions
    » Uniform (Voxel) grids
    » Octrees
    » BSP Trees
Intersection Testing

Accelerated techniques try to leverage:

- **Grouping:**
  Discard groups of primitives that are guaranteed to be missed by the ray.

- **Ordering:**
  Test nearer intersections first and allow for early termination if there is a hit.
Bounding Volumes

• Check for intersection with the bounding volume:
  - Bounding cubes
  - Bounding boxes
  - Bounding spheres
  - Etc.

Stuff that’s easy to intersect
Bounding Volumes

• Check for intersection with the bounding volume
  ◦ If the ray misses the bounding volume, it can’t intersect its contents

Still need to check for intersections with shape.
Bounding Volume Hierarchies

• Build hierarchy of bounding volumes
  ◦ Bounding volume of a parent node contains all children
Bounding Volume Hierarchies

- Grouping acceleration

```c
FindIntersection( Ray ray, Node node )
{
    ( min_t , min_shape ) = ( -1 , NULL )

    if( !intersect ( node.boundingVolume ) ) // Test Bounding box
        return ( -1 , NULL );

    foreach shape // Test node's shape
    {
        t = Intersect( shape )
        if( t>0 && (t<min_t || min_t<0) ) ( min_t , min_shape ) = ( t , shape )
    }

    for each child node // Test node's children
    {
        ( t , shape ) = FindIntersection( ray, child )
        if( t>0 && (t<min_t || min_t<0) ) ( min_t , min_shape ) = ( t , shape )
    }

    return ( min_t , min_shape );
}
```
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if you hit the bounding volume
Bounding Volume Hierarchies

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

- Don’t need to test shapes A or B
- Need to test groups 1, 2, and 3
- Need to test shapes C, D, E, and F
Bounding Volume Hierarchies

- Grouping + Ordering acceleration

```c
FindIntersection( Ray ray , Node node )
{
    // Find intersections with the shapes of the node
    ... 
    // Find intersections with child node bounding volumes
    ... 
    // Sort child bounding volume intersections front to back
    ...
    // Process intersections (checking for early termination)
    for each child node whose bounding box is intersected
    {
        if( min_t < bv_t[child] ) break;
        ( t , shape ) = FindIntersection( ray , child );
        if( t>0 && ( t < min_t || min_t<0 ) ) ( min_t , min_shape ) = ( t , shape )
    }
    return ( min_t , min_shape );
}
```
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect nodes only if you haven’t hit anything closer
Bounding Volume Hierarchies

- Use hierarchy to accelerate ray intersections
  - Intersect nodes only if you haven’t hit anything closer

- Don’t need to test shapes A, B, D, E, or F
- Need to test groups 1, 2, and 3
- Need to test shape C
Ray-Scene Intersection

• Intersections with geometric primitives
  ◦ Sphere
  ◦ Triangle

» Acceleration techniques
  ◦ Bounding volume hierarchies
  ◦ Spatial partitions
    » Uniform (Voxel) grids
    » Octrees
    » BSP trees
Uniform (Voxel) Grid

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

- A primitive may belong to multiple cells
- A cell may have multiple primitives
Uniform (Voxel) Grid

- Trace rays through grid cells
  - Fast
  - Incremental

Only check primitives in intersected grid cells
Uniform (Voxel) Grid

- Potential problem:
  - How choose suitable grid resolution?

  Too little benefit if grid is too coarse

  Too much cost if grid is too fine
Ray-Scene Intersection

• Intersections with geometric primitives
  ○ Sphere
  ○ Triangle

  » Acceleration techniques
  ○ Bounding volume hierarchies
  ○ Spatial partitions
    » Uniform (Voxel) grids
  » Octrees
  » BSP trees
Octrees

• We can think of a voxel grid as a tree.
  ◦ The root node is the entire region
  ◦ Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

- We can think of a voxel grid as a tree.
  - The root node is the entire region
  - Each node has eight children obtained by subdividing the parent into eight equal regions
Octrees

• In an octree, we only subdivide regions that contain more than one shape.
Octrees
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
- Adaptively determines grid resolution.
Octrees

- In an octree, we only subdivide regions that contain more than one shape.
- Adaptively determines grid resolution.

Efficiently tracing a ray through an adaptive octree is trickier than tracing a ray through a regular grid!
Overview

• Acceleration techniques
  ◦ Bounding volume hierarchies
  ◦ Spatial partitions
    » Uniform (Voxel) grids
    » Octrees
    » BSP trees
      – $k$-D trees
$k$-D Trees

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**$k$-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**k-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**$k$-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**k-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**k-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.
**k-D Trees**

- Alternate between splitting along the $x$-axis, $y$-axis, and $z$-axis.

**Note:**
- Either primitives need to be split, or they belong to multiple nodes.

**Limitations:**
- The splitting planes still have to be axis-aligned
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Recursively partition space by planes
  - Generate a tree structure where the leaves store the shapes.
Binary Space Partition (BSP) Tree

- Example: Point Intersection
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    - Left of 1 (root) → 2
Binary Space Partition (BSP) Tree

• Example: Point Intersection
  ◦ Recursively test what side we are on
    » Left of 2 → 4
Binary Space Partition (BSP) Tree

- Example: Point Intersection
  - Recursively test what side we are on
    » Right of 4 → Test B
Binary Space Partition (BSP) Tree

• Example: Point Intersection
  ○ Recursively test what side we are on
    » Missed B. No intersection!
Binary Space Partition (BSP) Tree

• Example: Point Intersection
  ◦ Recursively test what side we are on
    » Missed B. No intersection!

Worst-case / Expected complexity: proportional to the depth of the tree
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 1
  ○ Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the left of 1
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 1
  ◦ Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to the right of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 1
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - Intersection with C. Done!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - Test half to the left of 1
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - Test half to the right of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Missed C. Recurse!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to left of 2
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - Test half to left of 4
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 2
  ○ Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Missed A. Recurse!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    - No half to right of 4.
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to right of 1
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 2
  ◦ Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Test half to left of 3
Binary Space Partition (BSP) Tree

• Example: Ray Intersection 2
  ◦ Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Intersection with D. Done!
Binary Space Partition (BSP) Tree

- Example: Ray Intersection 2
  - Recursively split the ray and test nearer and farther halves, nearest first. Stop once you hit something:
    » Intersection with D. Done!

Worst-case: Proportional to the number of nodes in the tree
Expected: substantially faster

How should we choose the splitting planes?
Binary Space Partition (BSP) Tree

```
RayTreeIntersect( Ray ray , Node node )
{
    if ( Node is a leaf ) return intersection of closest primitive in cell, or NULL if none
    else
    {
        // Find near and far children
        near_child = child of node that contains the origin of Ray
        far_child = other child of node

        // Recurse down near child first
        isect = RayTreeIntersect( ray, near_child )
        if( isect ) return isect  // If there's a hit, we are done

        // If there's no hit, test the far child
        return RayTreeIntersect( ray, far_child )
    }
}
```