Spatial Data Structures
• Ray-Triangle Intersections

• Bounding Volume Hierarchy

• Spatial-Partitioning Structures
Ray-Mesh Intersection

- **Last lecture:** closest triangle to a point

- What if we want to find the closest triangle a ray intersects?
  - A ray is a point + a direction vector
  - More constrained problem
  - Naïve approach still needs to check every triangle!

\[ \mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \]
Ray-Mesh Intersection

- Spatial data structures that allow us to compute ray-mesh intersections without having to check every triangle
- Think of building these structures as a preprocessing step
  - Building can take a while
  - Searching must be fast!

\[ \mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \]
Ray-Plane Intersection

Given a plane defined as

\[ \mathbf{N}^T \mathbf{x} = c \]

We can find the intersection point by plugging in the ray for \( \mathbf{x} \)

\[ \mathbf{N}^T (\mathbf{o} + t \mathbf{d}) = c \]

Then solve for \( t \)

\[ t = \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \]

Substitute the time into the ray equation to find the intersection point

\[ \mathbf{p} = \mathbf{o} + \left( \frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \right) \mathbf{d} \]
Ray-Triangle Intersection

- Not much different:
  - i) Compute ray-plane intersection to find point $p$ on plane
  - ii) Perform point-in-triangle test for point $p$
    - Barycentric coordinates

- Not a very efficient algorithm...
  - Can we combine both steps into one?
  - **Idea:** set intersection and barycentric tests equal

\[
o + td = (1 - u - v) \cdot p_0 + u \cdot p_1 + v \cdot p_2
\]

- If the intersection point lies within the triangle, the above equation will have a solution
Moller-Trumbore Algorithm

Given the below equation

\[ \mathbf{o} + t\mathbf{d} = (1 - u - v) \mathbf{p}_0 + u \mathbf{p}_1 + v \mathbf{p}_2 \]

Rearrange the terms until unknowns are on one side

\[ \mathbf{o} - \mathbf{p}_0 = u (\mathbf{p}_1 - \mathbf{p}_0) + v (\mathbf{p}_2 - \mathbf{p}_0) - t\mathbf{d} \]

Rewrite in terms of variables**

\[ \mathbf{s} = u \mathbf{e}_1 + v \mathbf{e}_2 - t\mathbf{d} \]

Rewrite as a matrix operation

\[ \mathbf{s} = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & -\mathbf{d} \end{bmatrix} \begin{bmatrix} u \\ v \\ t \end{bmatrix} \]

Solve using Cramer’s rule

\[ \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \frac{1}{(\mathbf{e}_1 \times \mathbf{d}) \cdot \mathbf{e}_2} \begin{bmatrix} -\left( \mathbf{s} \times \mathbf{e}_2 \right) \cdot \mathbf{d} \\ (\mathbf{e}_1 \times \mathbf{d}) \cdot \mathbf{s} \\ -\left( \mathbf{s} \times \mathbf{e}_2 \right) \cdot \mathbf{e}_1 \end{bmatrix} \]
\[ s = \begin{bmatrix} e_1 & e_2 & -d \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix} \]

\[ o - p_0 = \begin{bmatrix} p_1 - p_0 & p_2 - p_0 & -d \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix} \]

- Matrix \( M^{-1} \) transforms triangle to unit triangle at the origin with unit-length edges spanning \( u \) and \( v \)
  - Transforms ray to be orthogonal to the triangle

- \( Q: \) What if \( t \) is negative?
  - Ray intersection happens in negative direction!
Spatial Data Structures

• Naïve ray-mesh intersection requires checking every triangle for ray-triangle intersection
  • Meshes have millions to billions of triangles
  • \( O(n) \) execution

• Idea: sort triangles in a way where we can perform quick intersection tests on groups of triangles at a time
Bounding Box

- Precompute the smallest axis-aligned bounding box around all primitives
  - Keep track of smallest and largest \((x,y,z)\) coordinates for all primitives

- Check for ray-box intersection
  - If \textit{misses}, we are done
  - If \textit{passes}, check all triangles

- Saves time for rays that clearly miss the mesh, but...
  - Still \(O(n)\) for rays that intersect the box
More Bounding Boxes

- What if we had 2 levels of bounding boxes?
  - Global bounding box
    - Head bounding box
    - Body bounding box
  
- Check for global ray-box intersection
  - If misses, we are done
  - If passes,
    - Check for head ray-box intersection
      - If misses, continue
      - If passes, check all triangles in head
    - Check for body ray-box intersection
      - If misses, continue
      - If passes, check all triangles in body

- Better, some rays can now pass the global bbox but neither the head/body bbox
  - We have tighter checks rays need to pass in order to search underlying triangles
A Hierarchy of...Bounding Volumes?
Bounding Volume Hierarchy (BVH)

- Recursively partition nodes into smaller nodes
  - Stop when node contains no more than several primitives

- The resulting **BVH** mimics a tree
  - Root node encompasses all primitives
  - Each non-root node has a parent
  - Each non-leaf node has two children
    - Some BVHs can have more than 2 children
    - Each leaf node points to a handful of primitives

Stanford Bunny BVH visualizing 10th level
• Ray-Triangle Intersections

• Bounding Volume Hierarchy

• Spatial-Partitioning Structures
Let’s look at an example
BVH Example

Bounding boxes will sometimes intersect!
BVH Example

fail X  pass ✓
BVH Example

Are we done?
We can find a closer triangle if we check here.

**Remember:** bounding boxes will intersect!
void hit(Ray* ray, BVHNode* node, HitInfo* best) {
    // test if ray hits node’s bbox
    HitInfo hit = intersect(ray, node->bbox);
    if (hit.prim == NULL || hit.t > best.t) return;
    // for leaves, check each primitive
    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = intersect(ray, p);
            if (hit.prim != NULL && hit.t < best.t) {
                best.prim = p;
                best.t = t;
            }
        }
    } else {
        // traverse BOTH children
        hit(ray, node->child1, best);
        hit(ray, node->child2, best);
    }
}

struct BVHNode {
    // is the node a leaf
    bool leaf;
    // min/max coordinates enclosing primitives
    Bbox bbox;
    // left child (can be NULL)
    BVHNode *child1;
    // right child (can be NULL)
    BVHNode *child2;
    // for leaves, stores primitives
    Primitive *primList;
}

struct HitInfo {
    // the primitive the ray hit
    Primitive *prim;
    // the time along the ray the hit occurred
    float t;
}
void hit(Ray* ray, BVHNode* node, HitInfo* best) {
    // test if ray hits node’s bbox
    HitInfo hit = intersect(ray, node->bbox);
    if (hit.prim == NULL || hit.t > best.t)
        return;
    // for leaves, check each primitive
    if (!node->leaf) {
        for (each primitive p in node->primList) {
            hit = intersect(ray, p);
            if (hit.prim != NULL && hit.t < best.t) {
                best.prim = p;
                best.t = t;
            }
        }
    } else {
        // traverse BOTH children
        hit(ray, node->child1, best);
        hit(ray, node->child2, best);
    }
}

We don’t ALWAYS need to check both children.
Recall the first example where we terminated after searching only the closer bbox.
void hit(Ray* ray, BVHNode* node, HitInfo* best) {
  if (node->leaf) {
    // same as previous slide
  } else {
    BVHNode* child1 = node->child1;
    BVHNode* child2 = node->child2;

    HitInfo hit1 = intersect(ray, child1->bbox);
    HitInfo hit2 = intersect(ray, child2->bbox);
    // pick node with better time
    BVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
    BVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

    hit(ray, first, best);
    if (hit2.t < best.t)
      hit(ray, second, best);
  }
}
void hit(Ray* ray, BVHNode* node, HitInfo* best) {
if (node->leaf) {
    // same as previous slide
} else {
    BVHNode* child1 = node->child1;
    BVHNode* child2 = node->child2;
    HitInfo hit1 = intersect(ray, child1->bbox);
    HitInfo hit2 = intersect(ray, child2->bbox);
    // pick node with better time
    BVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
    BVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;
    hit(ray, first, best);
    if (hit2.t < closest.t)
        hit(ray, second, best);
}
So we know how to traverse a BVH,
But how do we build one?
BVH Partitioning

What is the best way to partition these primitives?
BVH Partitioning

We can split them into equal # of primitives...
...but bboxes take up large area
We can split them into the smallest possible bboxes...
...but some bboxes will have many more primitives
Surface Area Heuristic

- The cost of intersecting a node is:
  \[
  C = C_{\text{trav}} + p_A C_A + p_B C_B
  \]
- Where:
  - \( C_{\text{trav}} \) measures the cost of intersecting the current node’s bbox
  - \( p_A \) measures the probability of a ray intersecting child node \( A \) given it intersects the parent node of \( A \)
  - \( C_A \) measures the cost of intersecting a primitive in child node \( A \)’s subtree

Surface Area Heuristic gives us a quantitative way of telling us if a partition is good
A better partition will have a lower cost
Surface Area Heuristic

- The cost of intersecting a node is:
  \[ C = C_{trav} + p_A C_A + p_B C_B \]

- Where:
  - \( C_{trav} \) measures the cost of intersecting the current node’s bbox
  - \( p_A \) measures the probability of a ray intersecting child node \( A \) given it intersects the parent node of \( A \)
  - \( C_A \) measures the cost of intersecting a primitive in child node \( A \)’s subtree

- Fixed cost associated with bbox intersection
- Having too large a BVH depth means we have to check too many bboxes before finding a primitive
Surface Area Heuristic

- The cost of intersecting a node is:
  \[ C = C_{\text{trav}} + p_A C_A + p_B C_B \]
- Where:
  - \( C_{\text{trav}} \) measures the cost of intersecting the current node’s bbox
  - \( p_A \) measures the probability of a ray intersecting child node \( A \) given it intersects the parent node of \( A \)
  - \( C_A \) measures the cost of intersecting a primitive in child node \( A \)’s subtree

- For a convex object \( A \) inside a parent convex object \( B \), the probability that a random ray that hits \( B \) also hits \( A \) is given by the ratio of the surface areas \( S_A \) and \( S_B \) of these objects:
  \[ P(\text{hit} A | \text{hit} B) = \frac{S_A}{S_B} \]
Surface Area Heuristic

- The cost of intersecting a node is:
  \[ C = C_{\text{trav}} + p_A C_A + p_B C_B \]

- Where:
  - \( C_{\text{trav}} \) measures the cost of intersecting the current node's bbox
  - \( p_A \) measures the probability of a ray intersecting child node \( A \) given it intersects the parent node of \( A \)
  - \( C_A \) measures the cost of intersecting a primitive in child node \( A \)'s subtree

- For a node \( C_A \), this is the cost of checking all primitives held by this box
  - All triangles have the same cost \( C_{\text{tri}} \)
  - For \( N_A \) triangles, cost is \( N_A C_{\text{tri}} \)
Surface Area Heuristic

- The cost of intersecting a node is:
  \[ C = C_{\text{trav}} + p_A C_A + p_B C_B \]

- Where:
  - \( C_{\text{trav}} \) measures the cost of intersecting the current node’s bbox
  - \( p_A \) measures the probability of a ray intersecting child node \( A \) given it intersects the parent node of \( A \)
  - \( C_A \) measures the cost of intersecting a primitive in child node \( A \)’s subtree

- New equation:
  \[ C = C_{\text{trav}} + \frac{S_A}{S_C} N_A C_{\text{tri}} + \frac{S_B}{S_C} N_B C_{\text{tri}} \]
  - \( C_{\text{trav}}, C_{\text{tri}} \) and \( S_C \) are constants, so we can remove them when computing the minimum cost:
  \[ C' = S_A N_A + S_B N_B \]
  - Minimizes surface area deviation
  - Minimizes primitive deviation
We know what a good partition is, but how do we actually build a partition
for(axis : [x, y, z]) {
    // check all axis-aligned partitions
    sort(primitives, axis);  // sort primitives by centroid
    n = primitives.length();
    for(int i = 0; i < n; i++) {
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);
        // surface area heuristic
        cost = a.area * i + b.area * (n - i);
        if(cost < best_cost) { best_cost = cost; best_partition = i; best_axis = axis; }
    }
}

// create children bounding boxes based on best axis and partition location
partition(best_axis, best_partition);
for(axis : [x, y, z]) { // check all axis-aligned partitions
    sort(primitives, axis); // sort primitives by centroid
    n = primitives.length();
    for(int i = 0; i < n; i++) {
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);
        // surface area heuristic
        cost = a.area * i + b.area * (n - i);
        if(cost < best_cost) {
            best_cost = cost;
            best_partition = i;
            best_axis = axis;
        }
    }
}
// create children bounding boxes based on best axis and partition location
partition(best_axis, best_partition);

Checking every partition in a scene with millions of primitives is incredibly expensive!
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    for(int i = 0; i < n; i+=32) { // check every B primitives (B = 32)
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);

        cost = a.area * i + b.area * (n - i);
        if(cost < best_cost) { best_cost = cost; best_partition = i; best_axis = axis; }
    }
}

partition(best_axis, best_partition);
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    for(int i = 0; i < n; i+=32) {
        // check every B primitives (B = 32)
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);

        cost = a.area * i + b.area * (n - i);
        if(cost < best_cost) { best_cost = cost; best_partition = i; best_axis = axis; }
    }
}

partition(best_axis, best_partition);

Still a lot of iterating over primitives each loop!
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    bin_n = bin.length();
    for(int i = 0; i < n; i++) {
        bin = compute_bucket(primitives[i].centroid) // find bin that triangle lies in
        bin.bbox.add(primitives[i]); } // add triangle to bin
    for(int j = 0; j < bin_n; j++) {
        a = bbox(bin[0,j]); // add bins to partitions instead of triangles
        b = bbox(bin[j, bin_n]); // add bins to partitions instead of triangles
        // same as before
    }
}
Building Partitions Example
Building Partitions Example

[ x-axis binning ]
Building Partitions Example

Cost = 3 prims * (0.15) + 8 prims * (0.87)
Building Partitions Example

Cost = 6 prims \times (0.38) + 5 prims \times (0.43)
Building Partitions Example

Cost = 9 prims * (0.81) + 2 prims * (0.18)
Building Partitions Example

[ y-axis binning ]
Building Partitions Example

Cost = 3 prims * (0.19) + 8 prims * (0.91)
Building Partitions Example

Cost = 6 prims * (0.32) + 5 prims * (0.36)
Building Partitions Example

Cost = 9 prims * (0.94) + 2 prims * (0.13)
Building Partitions Example

Best Partition
Building Partitions Example

Recurse with each child node
What About Ordering?
What About Ordering?
What About Ordering?
What About Ordering?

- Sort by partition axis
- Each node saves index start/end range for primitives it is responsible for
  - Combination of children node primitives should match parent node primitives
  - **Example**: all red and yellow primitives encased in orange primitive list
- When partitioning a node along an axis, should only sort for primitives in node’s range!
- Storing a BVH in memory requires storing the primitive index order, as well as the start/end indices of each node and their connectivity (parent/child) to the tree.
Edge Cases

[ primitives with same centroid ]
[ overlapping bboxes ]

In these cases, pick a random partition
BVH Review

Building the BVH:
1) Pick axis \([x, y, z]\)
   1) Sort primitives on axis by centroid
   2) Bin primitives (\(B = 32\))
   3) Partition primitives by bin along axis
   4) Compute SAH, saving best result
2) Construct 2 child nodes from best SAH result
3) Recurse until few primitives (< 4) left in node

Traversing the BVH:
1) Check if ray hits current node bbox
2) If hit, find which child node is closer to ray
3) Recurse down closer child
4) If the farther child node is closer to the ray than the hit discovered, recurse down the farther child

Traversal cost is \(O(\log(N))\), same as tree-search
Axis-Aligned BVH

• **What is an axis-aligned BVH?**
  • By searching for partitions along the axes \([x,y,z]\), we are constraining ourselves to build partitions with bounding boxes that are axis-aligned

• **How do we make a non-axis-aligned BVH?**
  • Simple! Just search for partitions that are not constrained to \([x,y,z]\)
    • Easy in theory, difficult in practice

• **What are the pros/cons of non-axis-aligned BVH?**
  • \([+]\) Better SAH
  • \([+]\) Nodes have less likelihood of having empty space
  • \([-]\) More work to compute partitions
  • \([-]\) Larger intersection cost for non-aligned bboxes
  • \([-]\) More memory overhead
Axis-Aligned BVH

• Are non-axis-aligned BVHs actually faster?
  • Yes, and no.
  \[ C = C_{\text{trav}} + \frac{S_A}{S_C} N_A C_{\text{tri}} + \frac{S_B}{S_C} N_B C_{\text{tri}} \]
  • Surface area ratio \( \frac{S_A}{S_C} \) decreases with better-fitting bboxes
  • Bounding box intersection cost \( C_{\text{trav}} \) increases with more compute required to check unaligned bbox

• How to check for intersection with non-axis-aligned bbox?
  • Bbox now has an extra transform matrix \( T \) taking it from the parent’s coordinate space to its own coordinate space
    • Apply the inverse transform to the bbox and ray and compute axis-aligned intersections
    • Larger memory overhead, now need to store the transform with each node
- Ray-Triangle Intersections
- Bounding Volume Hierarchy
- Spatial-Partitioning Structures
Primitives vs. Spatial

**Primitive Partitioning**
- Bounding Volume Hierarchy
  - [+] More flexible to geometry
  - [+] Easier to update (animation)
  - [-] Volumes can overlap
  - [-] Unable to terminate on first hit

**Spatial Partitioning**
- K-D Trees
- Uniform Grid
- Quad/Octree
  - [+] No volume overlap
  - [+] Can terminate on first hit
  - [-] Higher potential for empty space
  - [-] May intersect primitive multiple times
K-D Trees

• Recursively partition space via axis-aligned partitioning planes
  • Interior nodes correspond to spatial splits
  • Node traversal proceeds in front-to-back order
  • Unlike BVH, can terminate search after first hit is found
  • Still $O(\log(N))$ performance
Consider: Triangle 1 overlaps multiple zones
- Triangle 1 is checked for intersection when checking red zone first
  - Ray intersects triangle 1
  - But triangle 2 is closer
- Requirement: intersection point must lie within zone
Uniform Grid

- Partition space into equal sized volumes (volume-elements or "voxels")
- Each voxel contains primitives that overlap
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with primitives in voxels the ray intersects
- What is a good number of voxels?
  - Should be proportional to total number of primitives $N$
  - Number of cells traversed is proportional to $O(\sqrt[3]{N})$
    - A line going through a cube is a cubed root
    - Still not as good as $O(\log(N))$
Uniform Grid

Too few cells
Requires checking every primitive

Too many cells
Walking through a lot of empty space
Uniform Grid

- Uniform grid cannot adapt to non-uniform distribution of geometry in scene
  - Unlike K-D tree, location of spatial partitions is not dependent on scene geometry

Monsters University (2013) Pixar
Where Uniform Grids Work

Legend of Zelda: Tears of the Kingdom (2023) Nintendo
Quad-Tree/Octree

- Like uniform grid, easy to build
- Has greater ability to adapt to location of scene geometry than uniform grid
  - Still not as good adaptability as K-D tree

- **Quad-tree**: nodes have 4 children
  - Partitions 2D space

- **Octree**: nodes have 8 children
  - Partitions 3D space
## Spatial Data Structures Review

<table>
<thead>
<tr>
<th></th>
<th>[ Spatial ]</th>
<th>[ Primitive ]</th>
<th>[ Build Speed ]</th>
<th>[ Search Speed ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BVH</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>K-D Tree</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Quad/Octree</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>