Spatial Data Structures
Complexity of geometry
Review: ray-triangle intersection

- Find ray-plane intersection

  Parametric equation of a ray:
  \[ r(t) = o + td \]

  Plug equation for ray into implicit plane equation:
  \[ N^T \mathbf{x} = c \]
  \[ N^T (o + td) = c \]

  Solve for \( t \) corresponding to intersection point:
  \[ t = \frac{c - N^T o}{N^T d} \]

- Determine if point of intersection is within triangle
Review: ray-triangle intersection

- Parameterize triangle given by vertices \( p_0, p_1, p_2 \) using barycentric coordinates

\[
f(u, v) = (1 - u - v)p_0 + up_1 + vp_2
\]

- Can think of a triangle as an affine map of the unit triangle

\[
f(u, v) = p_0 + u(p_1 - p_0) + v(p_2 - p_0)
\]
Ray-triangle intersection

Plug parametric ray equation directly into equation for points on triangle:

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

Solve for \( u, v, t \):

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

\( M^{-1} \) transforms triangle back to unit triangle in \( u,v \) plane, and transforms ray’s direction to be orthogonal to plane.
Ray-primitive queries

Given primitive p:

\[ p.\text{intersect}(r) \text{ returns value of } t \text{ corresponding to the point of intersection with ray } r \]

\[ p.\text{bbox}() \text{ returns axis-aligned bounding box of the primitive} \]

\[
\text{tri.bbox():}
\quad \text{tri}_\text{min} = \min(p0, \min(p1, p2))
\quad \text{tri}_\text{max} = \max(p0, \max(p1, p2))
\quad \text{return } \text{bbox}((\text{tri}_\text{min}, \text{tri}_\text{max})
\]
Ray-axis-aligned-box intersection

What is ray’s closest/farthest intersection with axis-aligned box?

Find intersection of ray with all planes of box:

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

Math simplifies greatly since plane is axis aligned (consider \(x=x_0\) plane in 2D):

\[ \mathbf{N}^T = [1 \ 0]^T \]

\[ \mathbf{c} = x_0 \]

\[ t = \frac{x_0 - \mathbf{o}_x}{\mathbf{d}_x} \]

Figure shows intersections with \(x=x_0\) and \(x=x_1\) planes.
Ray-axis-aligned-box intersection

Compute intersections with all planes, take intersection of $t_{\min}/t_{\max}$ intervals

Intersections with $x$ planes

Intersections with $y$ planes

Final intersection result

How do we know when the ray misses the box?

Note: $t_{\min} < 0$
Ray-scene intersection

Given a scene defined by a set of N primitives and a ray r, find the closest point of intersection of r with the scene

“Find the first primitive the ray hits”

\[
p_{\text{closest}} = \text{NULL} \\
t_{\text{closest}} = \infty \\
\text{for each primitive } p \text{ in scene:} \\
\quad t = p.\text{intersect}(r) \\
\quad \text{if } t \geq 0 \text{ and } t < t_{\text{closest}}: \\
\quad \quad t_{\text{closest}} = t \\
\quad \quad p_{\text{closest}} = p
\]

Complexity? \( O(N) \)

Can we do better?
A simpler problem

- Imagine I have a set of integers $S$
- Given an integer, say $k=18$, find the element of $S$ closest to $k$:

  10 123 2 100 6 25 64 11 200 30 950
  111 20
  8 1
  80

What’s the cost of finding $k$ in terms of the size $N$ of the set?

Can we do better?

Suppose we first sort the integers:

  1 2 6 8 10 11 111 23 20 25 30 64 80 100 950

How much does it now cost to find $k$ (including sorting)?

Cost for just ONE query: $O(n \log n)$
Amortized cost: $O(\log n)$

worse than before! :-)

…much better!
Can we also reorganize scene primitives to enable fast ray-scene intersection queries?
Simple case

Ray misses bounding box of all primitives in scene

Cost (misses box):
- preprocessing: \(O(n)\)
- ray-box test: \(O(1)\)
- amortized cost*: \(O(1)\)

*over many ray-scene intersection tests
Another (should be) simple case

Cost (hits box):
preprocessing: $O(n)$
ray-box test: $O(1)$
triangle tests: $O(n)$
amortized cost*: $O(n)$

Still no better than naïve algorithm (test all triangles)!

*over many ray-scene intersection tests
Q: How can we do better?

A: Use deep learning.

A: Apply this strategy hierarchically.
Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - Contain small list of primitives

- **Interior nodes:**
  - Proxy for a large subset of primitives
  - Stores bounding box for all primitives in subtree

Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?
Another BVH example

- BVH partitions each node's primitives into disjoint sets
  - Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)
Ray-scene intersection using a BVH

```c
struct BVHNode {
    bool leaf; // am I a leaf node?
    BBox bbox; // min/max coords of enclosed primitives
    BVHNode* child1; // “left” child (could be NULL)
    BVHNode* child2; // “right” child (could be NULL)
    Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
    Primitive* prim; // which primitive did the ray hit?
    float t; // at what t value?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
    HitInfo hit = intersect(ray, node->bbox); // test ray against node’s bounding box
    if (hit.prim == NULL || hit.t > closest.t)
        return; // don’t update the hit record

    if (node->leaf) {
        for (each primitive p in node->primList) {
            hit = intersect(ray, p);
            if (hit.prim != NULL && hit.t < closest.t) {
                closest.prim = p;
                closest.t = t;
            }
        }
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}
```
Improvement: “front-to-back” traversal

Invariant: only call find_closest_hit() if ray intersects bbox of node.

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit && t < closest.t) {
                closest.prim = p;
                closest.t = t;
            }
        }
    } else {
        HitInfo hit1 = intersect(ray, node->child1->bbox);
        HitInfo hit2 = intersect(ray, node->child2->bbox);

        NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
        NVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

        find_closest_hit(ray, first, closest);
        if (second child’s t is closer than closest.t)
            find_closest_hit(ray, second, closest); // why might we still need to do this?
    }
}
For a given set of primitives, there are many possible BVHs
($2^N/2$ ways to partition $N$ primitives into two groups)

Q: How do we build a high-quality BVH?
How would you partition these triangles into two groups?
What about these?
Intuition about a “good” partition?

Partition into child nodes with equal numbers of primitives

Better partition
Intuition: want small bounding boxes (minimize overlap between children, avoid empty space)
What are we really trying to do?

A good partitioning minimizes the cost of finding the closest intersection of a ray with primitives in the node.

If a node is a leaf node (no partitioning):

\[ C = \sum_{i=1}^{N} C_{\text{isect}}(i) \]

Where \( C_{\text{isect}}(i) \) is the cost of ray-primitive intersection for primitive \( i \) in the node.

\[ = NC_{\text{isect}} \]

(Common to assume all primitives have the same cost)
Cost of making a partition

The expected cost of ray-node intersection, given that the node’s primitives are partitioned into child sets A and B is:

\[ C = C_{\text{trav}} + p_A C_A + p_B C_B \]

- \( C_{\text{trav}} \) is the cost of traversing an interior node (e.g., load data, bbox check)
- \( C_A \) and \( C_B \) are the costs of intersection with the resultant child subtrees
- \( p_A \) and \( p_B \) are the probability a ray intersects the bbox of the child nodes A and B

Primitive count is common approximation for child node costs:

\[ C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}} \]

Remaining question: how do we get the probabilities \( p_A, p_B \)?
For convex object A inside 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.

$$P(\text{hit } A | \text{hit } B) = \frac{S_A}{S_B}$$

Leads to surface area heuristic (SAH):

$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

Assumptions of the SAH (which may not hold in practice!):

- Rays are randomly distributed
- Rays are not occluded
Implementing partitions

- Constrain search for good partitions to axis-aligned spatial partitions
  - Choose an axis; choose a split plane on that axis
  - Partition primitives by the side of splitting plane their centroid lies
  - SAH changes only when split plane moves past triangle boundary
  - Have to consider rather large number of possible split planes…
Efficiently implementing partitioning

- Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: $B < 32$)

For each axis: $x, y, z$:
- Initialize buckets
- For each primitive $p$ in node:
  - $b = \text{compute_bucket}(p.\text{centroid})$
  - $b.\text{bbox}.\text{union}(p.\text{bbox})$
  - $b.\text{prim_count}++$

For each of the $B-1$ possible partitioning planes evaluate SAH
Recurse on lowest cost partition found (or make node a leaf)
Troublesome cases

- All primitives with same centroid (all primitives end up in same partition)
- All primitives with same bbox (ray often ends up visiting both partitions)

In general, different strategies may work better for different types of geometry / different distributions of primitives...
Primitive-partitioning acceleration structures vs. space-partitioning structures

- Primitive partitioning (bounding volume hierarchy): partitions node’s primitives into disjoint sets (but sets may overlap in space)

- Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)
K-D tree

- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in front-to-back order
  - Unlike BVH, can terminate search after first hit is found.
Challenge: objects overlap multiple nodes

- Want node traversal to proceed in front-to-back order so traversal can terminate search after first hit found.

Triangle 1 overlaps multiple nodes.
Ray hits triangle 1 when in highlighted leaf cell.
But intersection with triangle 2 is closer! (Haven’t traversed to that node yet)

Solution: require primitive intersection point to be within current leaf node.
(primitives may be intersected multiple times by same ray *)

* Caching or “mailboxing” can be used to avoid repeated intersections
Uniform grid

- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- 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 should the grid resolution be?

Too few grids cell: degenerates to brute-force approach

Too many grid cells: incur significant cost traversing through cells with empty space
Heuristic

- Choose number of voxels \( \sim \) total number of primitives
  (constant prims per voxel — assuming uniform distribution of primitives)

Intersection cost: \( O\left(\sqrt[3]{N}\right)\)

(Q: Which grows faster, cube root of \( N \) or \( \log(N) \)?)
Uniform distribution of primitives

Uniform grids work well for large collections of objects that are uniform in size and distribution.

Terrain / height fields:

[Image credit: Misuba Renderer]

Grass:

[Image credit: www.kevinboulanger.net/grass.html]
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)

"Teapot in a stadium problem"
Scene has large spatial extent.
Contains a high-resolution object that has small spatial extent (ends up in one grid cell)
Non-uniform distribution of geometric detail

[Image credit: Pixar]
Quad-tree / octree

Like uniform grid: easy to build (don’t have to choose partition planes)

Has greater ability to adapt to location of scene geometry than uniform grid.

But lower intersection performance than K-D tree (only limited ability to adapt)

Quad-tree: nodes have 4 children (partitions 2D space)
Octree: nodes have 8 children (partitions 3D space)
Summary of spatial acceleration structures: Choose the right structure for the job!

- **Primitive vs. spatial partitioning:**
  - Primitive partitioning: partition sets of objects
    - Bounded number of BVH nodes, simpler to update if primitives in scene change position
  - Spatial partitioning: partition space
    - Traverse space in order (first intersection is closest intersection), may intersect primitive multiple times

- **Adaptive structures (BVH, K-D tree)**
  - More costly to construct (must be able to amortize cost over many geometric queries)
  - Better intersection performance under non-uniform distribution of primitives

- **Non-adaptive accelerations structures (uniform grids)**
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed

- Many, many combinations thereof...
Next time: light