Accelerating Geometric Queries

Computer Graphics
CMU 15-462/15-662, Fall 2016
Geometric modeling and geometric queries

What point on the mesh is closest to \( p \)?
What point on the mesh is closest to \( p \)?

- Conceptually easy:
  - loop over all triangles
  - compute closest point to current triangle
  - keep globally closest point
- Q: What’s the cost? Does halfedge help?
- What if we have \textit{billions} of faces?
Different query: ray-mesh intersection

- A “ray” is an oriented line starting at a point
- Want to know where a ray pierces a surface
- Why?
  - GEOMETRY: inside-outside test
  - RENDERING: visibility, ray tracing
  - SIMULATION: collision detection
- Might pierce surface in many places!
Ray: parametric equation

\[ r(t) = o + td \]

- **Point along ray**
- **Origin**
- **Unit direction**
Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points $x$ such that $f(x) = 0$
- How do we find points where a ray intersects this surface?
  - we know all points along the ray: $r(t) = o + td$
  - replace “$x$” with “$r$”, solve for $t$
- Example: unit sphere
  
  $$f(x) = |x|^2 - 1$$
  
  $$f(r(t)) = |o + td|^2 - 1$$
  
  $$|d|^2 t^2 + 2(o \cdot d) t + |o|^2 - 1 = 0$$

  $$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  $a = |d|^2$
  $b = 2(o \cdot d)$
  $c = |o|^2 - 1$
Ray-plane intersection

• Suppose we have a plane $N^Tx = c$

• How do we find intersection with ray $r(t) = o + td$?

• Again, replace point $x$ with the ray equation:

$$N^T(o + td) = c$$

• Solve for $t$:

$$t = \frac{c - N^To}{N^Td}$$

• And plug $t$ back into ray equation:

$$r(t) = o + \frac{c - N^To}{N^Td}d$$
Ray-triangle intersection

• Triangle is in a plane...
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Why not compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point in triangle
  - Not much more to say!

• Actually, a *lot* more to say…
Ray-triangle intersection

- Parameterize triangle given by vertices \( \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2 \) using barycentric coordinates

\[
    f(u, v) = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2
\]

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

\[
    f(u, v) = \mathbf{p}_0 + u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0)
\]
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-scene intersection – a first optimization
Ray-axis-aligned-box intersection
Ray-axis-aligned-box intersection

Find intersection of ray with all planes of box:

$$\mathbf{N}^T(\mathbf{o} + td) = c$$

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

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

$$c = x_0$$

$$t = \frac{x_0 - \mathbf{o}_x}{d_x}$$

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 $t_{\text{min}}/t_{\text{max}}$ intervals

Intersections with $x$ planes
Intersections with $y$ planes
Final intersection result

How do we know when the ray misses the box?
Core methods for ray-primitive queries

Given primitive p:

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

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

\texttt{tri.bbox():}

\begin{align*}
\text{tri}_\text{min} &= \min(p0, \min(p1, p2)) \\
\text{tri}_\text{max} &= \max(p0, \max(p1, p2))
\end{align*}

\texttt{return \text{bbox}(\text{tri}_\text{min}, \text{tri}_\text{max})}
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”

```python
p_closest = NULL
t_closest = inf
for each primitive $p$ in scene:
    t = p.intersect(r)
    if $t \geq 0$ && $t < t_{\text{closest}}$:
        $t_{\text{closest}} = t$
        $p_{\text{closest}} = p$
```
Let’s look at a simpler problem

- Take a set of integers $S$

  10  123  20  100  6  25  64  11  200  30

- Given a new integer $k=18$, find the element in $S$ that is closest

Sort first:

6  10  11  20  25  30  64  100  123  200

Then what?
How do we organize scene primitives to enable fast ray-scene intersection queries?
Simple case (we’ve seen it already)

Ray misses bounding box of all primitives in scene

O(1) cost: requires 1 ray-box test
Another (should be) simple case

Ray hits bounding box, check all primitives
O(N) cost 😞
Another (should be) simple case

A bounding box of bounding boxes!
Another (should be) simple case

There is no reason to stop there!
Bounding volume hierarchy (BVH)

- Interior nodes:
  - Represent subset of primitives in scene
  - Store aggregate bounding box for all primitives in subtree
- Leaf nodes:
  - Contain list of primitives

Two different BVH organizations of the same scene containing 22 primitives. Leaf node are the same.

Q: Which one is better?
A less-structured BVH example

- BVH partitions each node’s primitives into disjoint sets
  - Note: The sets can still be overlapping in space!
Ray-scene intersection using a BVH

struct BVHNode {
    bool leaf;
    BBox bbox;
    BVHNode* child1;
    BVHNode* child2;
    Primitive* primList;
};

struct ClosestHitInfo {
    Primitive prim;
    float min_t;
};

void find_closest_hit(Ray* ray, BVHNode* node, ClosestHitInfo* closest) {
    if (!intersect(ray, node->bbox))
        return;

    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit && t < closest->min_t) {
                closest->prim = p;
                closest->min_t = t;
            }
        }
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}

Hmmm… this is still checking all the primitives in the scene.
Ray-scene intersection using a BVH

```cpp
struct BVHNode {
    bool leaf;
    BBox bbox;
    BVHNode* child1;
    BVHNode* child2;
    Primitive* primList;
};

struct ClosestHitInfo {
    Primitive prim;
    float min_t;
};

void find_closest_hit(Ray* ray, BVHNode* node, ClosestHitInfo* closest) {
    if (!intersect(ray, node->bbox) || (closest point on box is farther than closest.min_t))
        return;
    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit && t < closest.min_t) {
                closest.prim = p;
                closest.min_t = t;
            }
        }
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}
```

What if ray points the other way?
Improvement: “front-to-back” traversal

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

```c
void find_closest_hit(Ray* ray, BVHNode* node, ClosestHitInfo* closest) {
    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit && t < closest.min_t) {
                closest.prim = p;
                closest.min_t = t;
            }
        }
    } else {
        (hit1, min_t1) = intersect(ray, node->child1->bbox);
        (hit2, min_t2) = intersect(ray, node->child2->bbox);

        NVHNode* first = (min_t1 <= min_t2) ? child1 : child2;
        NVHNode* second = (min_t1 <= min_t2) ? child2 : child1;

        find_closest_hit(ray, first, closest);
        if (second child’s min_t is closer than closest.min_t) {
            find_closest_hit(ray, second, closest);
        }
    }
}
```

“Front to back” traversal. Traverse to closest child node first. Why?
Another type of query: any hit

Sometimes it’s useful to know if the ray hits ANY primitive in the scene at all (don’t care about distance to first hit)

```cpp
bool find_any_hit(Ray* ray, BVHNode* node) {
    if (!intersect(ray, node->bbox))
        return false;

    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit)
                return true;
        }
    } else {
        return ( find_closest_hit(ray, node->child1, closest) ||
                find_closest_hit(ray, node->child2, closest) );
    }
}
```

Traversal order should attempt to maximize chance of any primitive being hit!
Two different BVH trees. Which one is better?
For a given set of primitives, there are many possible BVHs

Q: how many ways are there to partition N primitives into two groups?

A: \(2^{N-2}\)

So, how do we build a high-quality BVH tree?
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

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.

\[ = N C_{\text{isect}} \]

(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 child nodes
- \( p_A \) and \( p_B \) are probabilities ray intersects bbox of child node

**Primitive count is a common approximation for child node costs:**

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

Where: \( N_A = |A|, N_B = |B| \)
Estimating probabilities

For a given direction, the number of rays that hits an object is proportional to the projected area.
Estimating probabilities

For rays with endpoint at fixed distance, probability that a ray hits object is proportional to the solid angle subtended by its surface.
Estimating probabilities

For convex objects and rays with endpoints that are far away, probability that a ray hits object is approximately proportional to surface area
Estimating probabilities

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.

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

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 (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
Efficiently implementing partitioning

- Efficient 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
Execute lowest cost partitioning 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)
Primitive-partitioning acceleration
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)
Uniform grid

- Partition space into equal sized volumes (“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)

Intersection cost: $O(\sqrt[3]{N})$
Non-uniform distribution of geometric detail requires adaptive grids

[Image credit: Pixar]
K-D trees

- Recursively partition space via axis-aligned planes
  - Interior nodes correspond to spatial splits (still correspond to spatial volume)
  - 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 *)
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)
Summary of accelerating geometric queries: 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 construction 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