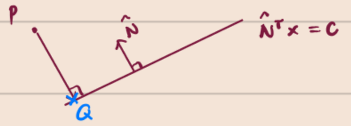
Chapter 6 Geometry Queries

I. Geometry Quests.

1. **Closest Point** on a geometry for point $P(p_1, p_2)$.

① **Point** Q to P: Q.

② **Line** $\hat{N}^T x = c$ to P   ($\hat{N}$: the unit normal)



As shown in the diagram, the closest point should be Q.

We know that $\hat{N}^T Q = c \Leftrightarrow \hat{N}^T(p + t\hat{N}) = c$ i.e. $p + t\hat{N} \Leftrightarrow p + (c - \hat{N}^T p)\hat{N}$

$\Leftrightarrow \hat{N}^T p + t\hat{N}^T\hat{N} = c$

$\Leftrightarrow t = c - \hat{N}^T p$

③ **Segment** Q_1Q_2 to P.

- Find closest point Q on the line where the segment locates.
- If Q is on Q_1Q_2 return Q.
- Else, check $\|PQ_1\|$ and $\|PQ_2\|$. Return the endpoint which has a smaller distance.

④ **Triangle** ABC to P

- If P is inside ABC, return P.
- Else, return min(seg(AB, P), seg(AC, P), seg(BC, P)).

⑤ **3D Triangle** ABC to P.

- Project P onto the plane where A, B, C coexist. (the projected point is P').
- Return Triangle(ABC, P').

⑥ **3D Mesh** to P

Intuitively just go over all the triangles if P is outside the mesh.

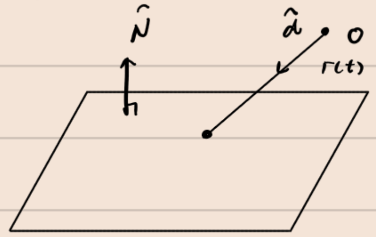
But this will be expensive. will be covered later.

2. **Ray-Mesh Intersection.**

Def (Ray) Ray is defined as a vector function of time. $\underset{\text{ray}}{\vec{r}}(\overset{\text{time}}{t}) = \underset{\text{origin}}{\vec{o}} + t\hat{d} \rightarrow \text{unit direction}$

Query Find the point where the ray intersect with the mesh.

① **Ray - plane** Intersection.



<u>Plane</u> : $\hat{N}^T x = c$

<u>Intersect</u> : Sol to $\hat{N}^T \vec{r}(t) = c$.

$\Leftrightarrow \hat{N}^T(\vec{o} + t\hat{d}) = c$

$\Leftrightarrow t = \dfrac{c - \hat{N}^T\vec{o}}{\hat{N}^T\hat{d}}$

Plug into $\vec{r}$: intersect @ $\vec{o} + \dfrac{c - \hat{N}^T\vec{o}}{\hat{N}^T\hat{d}}\hat{d}$

② **Ray - Triangle** Intersection

The triangle is inside a plane.

We check the Ray-Plane intersection first. then check if the intersecting point is inside the △.

③ **Mesh-Mesh** Intersection.

Level up from simpler primitives

- $\overset{P_1}{\text{Point}}$ - $\overset{P_2}{\text{Point}}$ Intersection : check if $P_1 = P_2$.
- $\overset{P}{\text{Point}}$ - $\overset{L}{\text{Line}}$ Intersection : check if P is on L.
- $\overset{L_1}{\text{Line}}$ - $\overset{L_2}{\text{Line}}$ Intersection : Point on L_1 & L_2 simultaneously

$$\begin{cases} L_1 : a\vec{x} = b \\ L_2 : c\vec{x} = d. \end{cases} \Rightarrow \text{solve} \begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

- $\overset{\Delta_1}{\text{Triangle}}$-$\overset{\Delta_2}{\text{Triangle}}$ Intersection : check edge-triangle intersection first, then interval test.

II. Spatial Acceleration Data structure.

1. **First Hit Problem**.

<u>Query</u> Given a scene defined by N primitives and a ray $\vec{r}$. find the closest point of intersection.

① Naïve algorithm O(N).

- Intersect with every △.
- Maintain the closest hit point.

② **Bounding Box** (BBox). worst case O(N).

(axis-aligned) "AABB"

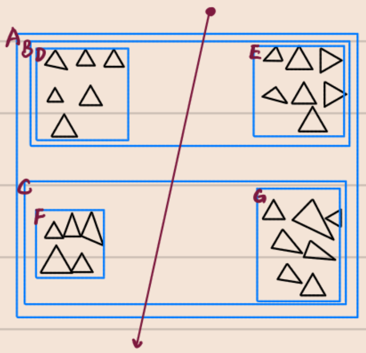
- Precompute the smallest **bounding box** around all primitives. → BBox can be build by looping over vertices.
- Intersect the ray with bbox.
  - If misses. the ray must touch no primitive, then we're done.
  - If hit. then try all △.

⚠ For axis-aligned bbox. the ray-box intersection may be conceptually easier.

since axis-aligned.

$\hat{N}^T(\vec{o} + t\hat{d}) = c \Rightarrow N^T = [1 \ 0]^T. \quad c = x_0. \quad \Rightarrow t = \dfrac{x_0 - \vec{o}_x}{\hat{d}_x}$

③ Bounding Volume Hierarchy (BVH)



Tree structure {
- Leaf : containing all small list of primitives.
- Interior : • Proxy for larger subset of primitives.
  • Stores bbox for primitives in subtree.

Algo (Recursive Iteration).

```
FindClosestHit ( Ray ray, BVHNode node, HitInfo closest ) {

    HitInfo hit = intersect ( ray, node.bbox );

    if ( hit.prim == NULL || hit.t > closest.t )    // haven't intersect with any prim or not first hit

        return;

    if ( node.isLeaf ) {

        foreach ( primitive p in node.primList ) {

            hit = intersect ( ray, p ) ;

            if ( hit.prim != NULL && hit.t < closest.t ) {    // update if the prim is closer.

                closest.prim = p;

                closest.t = t;

            }

        }

    } else {

        FindClosestHit ( ray, node.childL, closest ) ;

        FindClosestHit ( ray, node.childR, closest ) ;

    }

}
```

could possibly do better if we traverse in a way that is likely to terminate earlier. Since ray is straight.

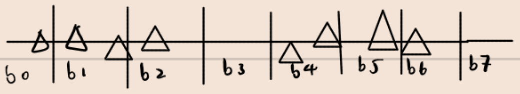
Data Structure

```
struct BVHNode {

    bool isLeaf;

    BBox bbox;

    BVHNode childL, childR;

    Primitive [] primitives;

}
```

```
struct HitInfo {

        Primitive prim;

        float t;

}
```

How to Quickly Build BVH ? (Efficient Modern Approximation)

- Split primitives into B buckets. (usually B < 32.).

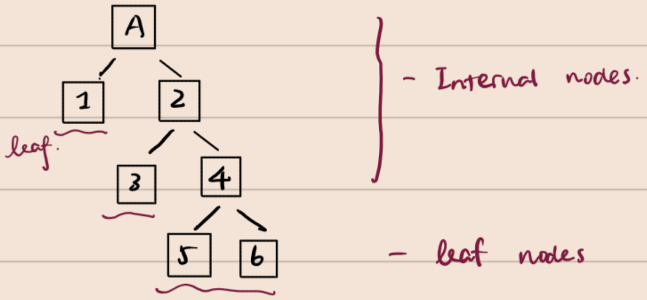


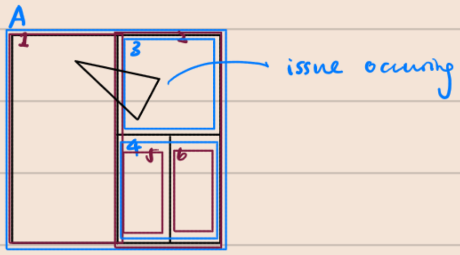
- Compute the bucket for each primitive. (with centroid probably).
- Union the bbox with the original bbox in this bucket.
- For B-1 possible partitioning planes. choose the lowest cost.

Issue   Sets may overlap.

④ KD-Tree.

Recursively partition space via axis-aligned partitioning planes.





Internal :
- partition plane (axis being aligned, position).
- pointers to children.

Leaf :
- primitive List.

Algorithm   If intersect with internal : check both children.

otherwise. skip.

Issue   A primitive may occur in multiple leaves.