A3: PathTracing

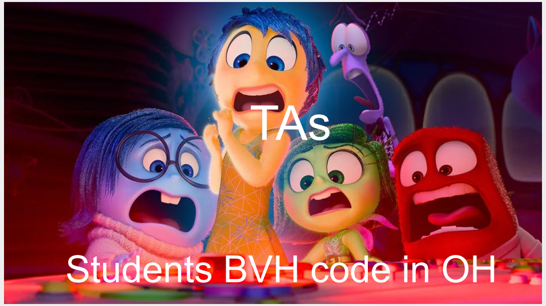
- Camera Rays
- Intersections
- BVH Construction
- BVH Navigation

Motivation

- Path tracing is everywhere!
- Many of the neat graphics renders you've seen, animations you've watched, or games you've played have probably been path traced!

How does it work?

- At a high level, we're trying to replicate what is happening in the real world
- We do this by simulating how light rays interact with our scene and shooting out millions of (or more) light rays and running this complex simulation to get our final image
- This doesn't mean our renders need to be photo realistic though

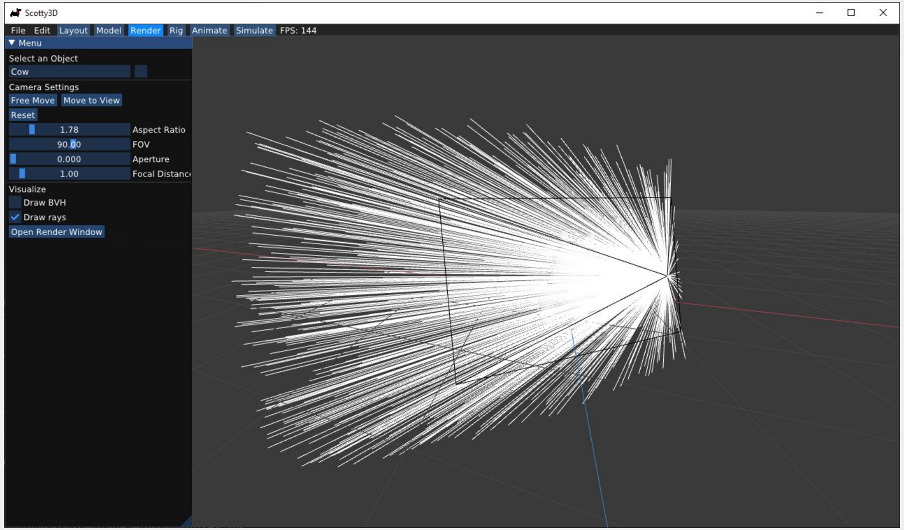


- Camera Rays
- Intersections
- BVH Construction
- BVH Navigation

Task 1: Camera Rays

- To path trace, we need to shoot out rays from the camera that bounce around the scene

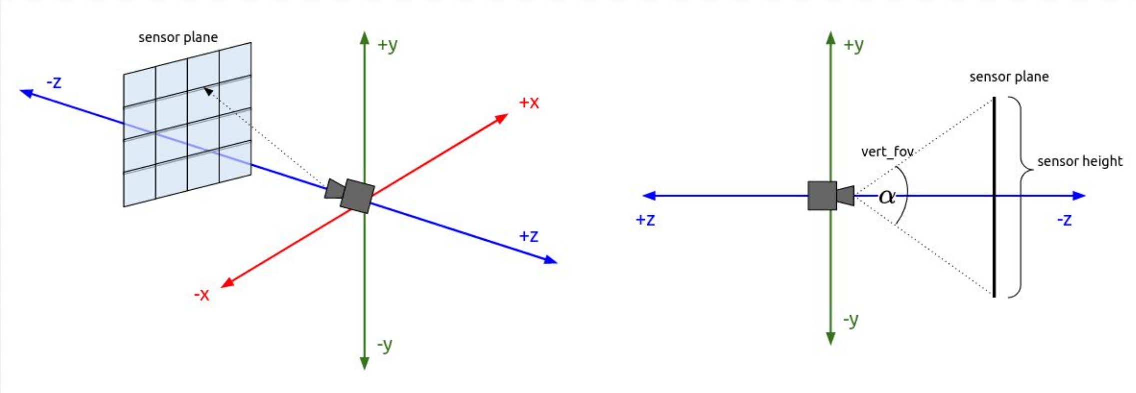
- This task is responsible for actually constructing the rays



Task 1: Camera Rays

- "Sample a ray that starts at the origin and passes through pixel (px,py)"
- To do this, we need the width and height of the sensor plane given:
    - **Vertical_fov**
    - **Aspect_ratio** = W/H

- How do we calculate the Width and height??



- Camera Rays
- Intersections
- BVH Construction
- BVH Navigation

Task 2 : Intersections

- We want to answer “does this ray hit this object?”
- We have two types of objects to check ray intersections against :
  Spheres (within the Shape class) and **Triangles** (as a part of a Tri_Mesh)

- For a given ray and shape, want to output :
  - **hit**: a boolean representing if there is a hit or not.
  - **distance**: the distance from the origin of the ray to the hit point
  - **position**: the position of the hit point
  - **uv**: The uv coordinates of the hit point on the surface
  - **origin**: the origin of the query ray

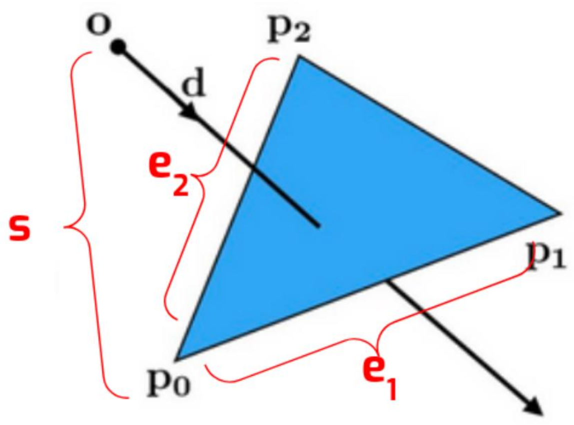
Step 1: Triangle::hit

Reference code uses Moller-Trombore algorithm:

- Parameterize points within the triangle with barycentric coordinates (u, v, w) :

$$\begin{aligned}\mathbf{P} &= w \cdot \mathbf{p}_0 + u \cdot \mathbf{p}_1 + v \cdot \mathbf{p}_2 \\ &= (1 - u - v) \cdot \mathbf{p}_0 + u \cdot \mathbf{p}_1 + v \cdot \mathbf{p}_2 \\ &= \mathbf{p}_0 + u \cdot (\mathbf{p}_1 - \mathbf{p}_0) + v \cdot (\mathbf{p}_2 - \mathbf{p}_0)\end{aligned}$$

- Parameterize input ray with **normalized** direction d, origin o, and time t (remember distance = rate x time!)
    - $\mathbf{P} = o + t*d$



Trace Triangle::hit(const Ray &ray);

in tri_mesh.cpp

Step 1: Triangle::hit

- Now we set them equal to each other!

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

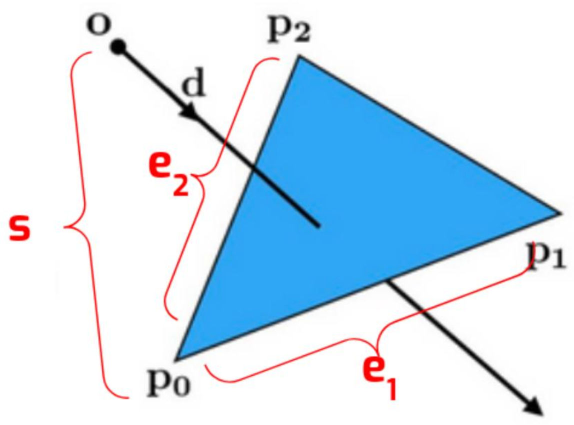
$$\mathbf{o} + t \cdot \mathbf{d} = \mathbf{p}_0 + u \cdot \mathbf{e}_1 + v \cdot \mathbf{e}_2$$

$$\implies \mathbf{o} - \mathbf{p}_0 = u \cdot \mathbf{e}_1 + v \cdot \mathbf{e}_2 + t \cdot (-\mathbf{d})$$

$$\implies \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & -\mathbf{d} \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p}_0 = \mathbf{s}$$

-Cramer's Rule reduces this to a fraction of determinants:

$$\begin{bmatrix} u \\ v \\ t \end{bmatrix} = \frac{1}{(\mathbf{e}_1 \times \mathbf{d}) \cdot \mathbf{e}_2} \cdot \begin{bmatrix} -(\mathbf{s} \times \mathbf{e}_2) \cdot \mathbf{d} \\ (\mathbf{e}_1 \times \mathbf{d}) \cdot \mathbf{s} \\ -(\mathbf{s} \times \mathbf{e}_2) \cdot \mathbf{e}_1 \end{bmatrix}$$



Trace Triangle::hit(const Ray &ray);

in tri_mesh.cpp

Step 1: Triangle::hit

Things to think about:

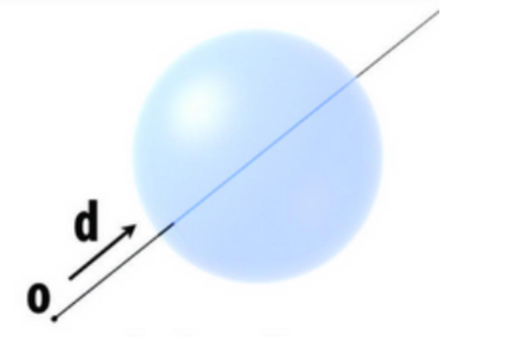
-This equation gives us barycentric coordinates **u** and **v**. How do we use these to tell if the ray intersection point is actually in the triangles?

-Look at the denominator $\frac{1}{(\mathbf{e}_1 \times \mathbf{d}) \cdot \mathbf{e}_2}$ What happens if we get 1/0. Can such a triangle be hit by a ray?

Step 2: Sphere::hit

-Instead of barycentric coordinates, use algebraic equation for sphere with center **c** and radius **r** :

$$||\mathbf{x}-\mathbf{c}||^2 - r^2 = 0.$$




```
Trace Sphere::hit(const Ray &ray);
```

in shapes.cpp

-Then we use the good ol' quadratic formula

$$\begin{aligned} ||x-c||^2 - r^2 &= 0 \\ ||\mathbf{o}+t\mathbf{d}||^2 - r^2 &= 0 \\ \underbrace{||\mathbf{d}||^2}_{a} \cdot t^2 + \underbrace{2\cdot(\mathbf{o}\cdot\mathbf{d})}_{b} \cdot t + \underbrace{||\mathbf{o}||^2 - r^2}_{c} &= 0 \end{aligned}$$

$$t = \frac{-2\cdot(\mathbf{o}\cdot\mathbf{d}) \pm \sqrt{4\cdot(\mathbf{o}\cdot\mathbf{d})^2 - 4\cdot||\mathbf{d}||^2\cdot(||\mathbf{o}||^2 - r^2)}}{2\cdot||\mathbf{d}||^2}$$

- Camera Rays
- Intersections
- BVH Construction
- BVH Navigation

Task 3: BVH

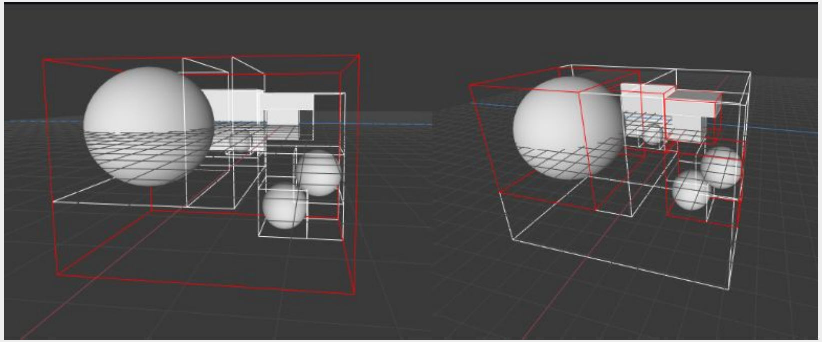
Bounding Volume Hierarchy

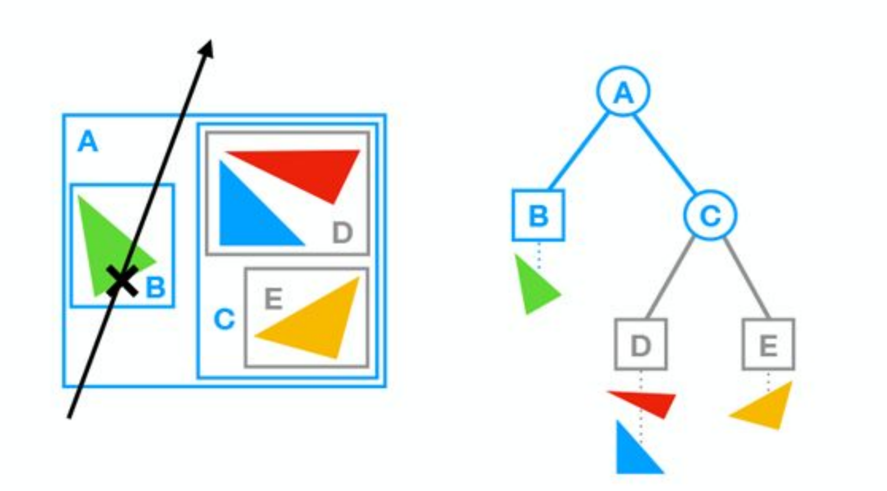
- Spatial Hierarchy
- Bounding Boxes (`bbox`) + Primitives
- Functions like a tree

Motivation:

Checking all the primitives at once for every ray hit is expensive.

BVH's tree structure speeds it up from O(n) -> O(nlogn)

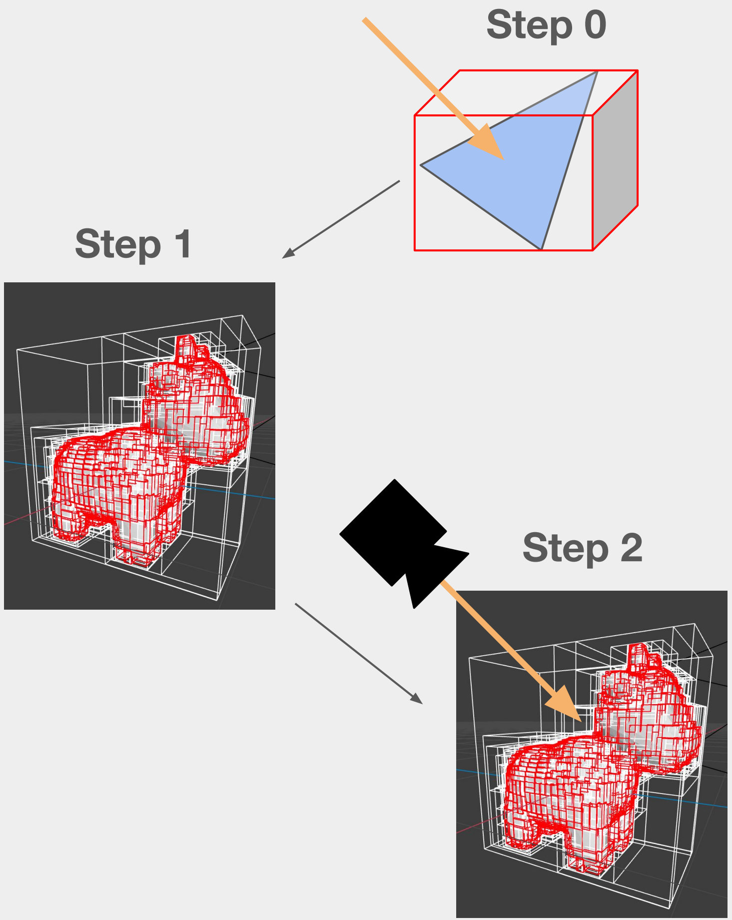




Task 3: BVH

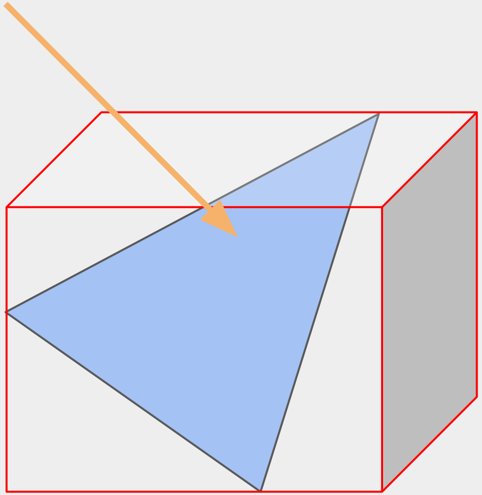
High-Level Procedure:

- **Step 0** - Implement `BBox` Intersection
- **Step 1** - Create BVH data structure using <u>Surface Area Heuristic</u> (to make Step 2 faster)
- **Step 2** - Implement Path Tracing with BVH data structure



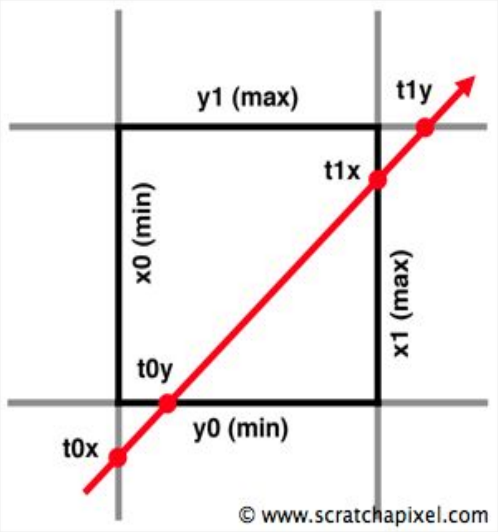
Task 3: BVH

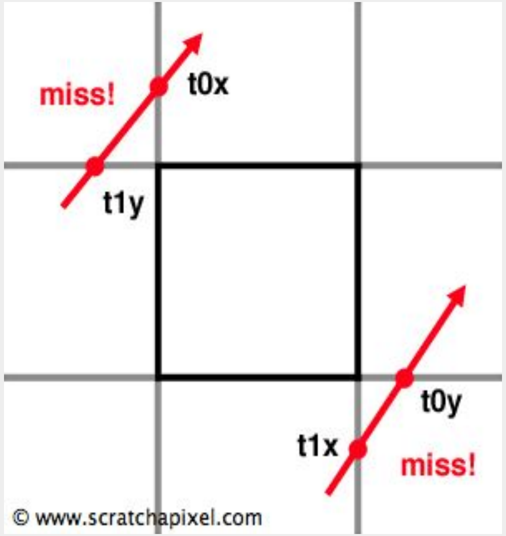
Step 0: Ray-BBox Intersection



Implement Ray-BBox intersection in BBox::hit

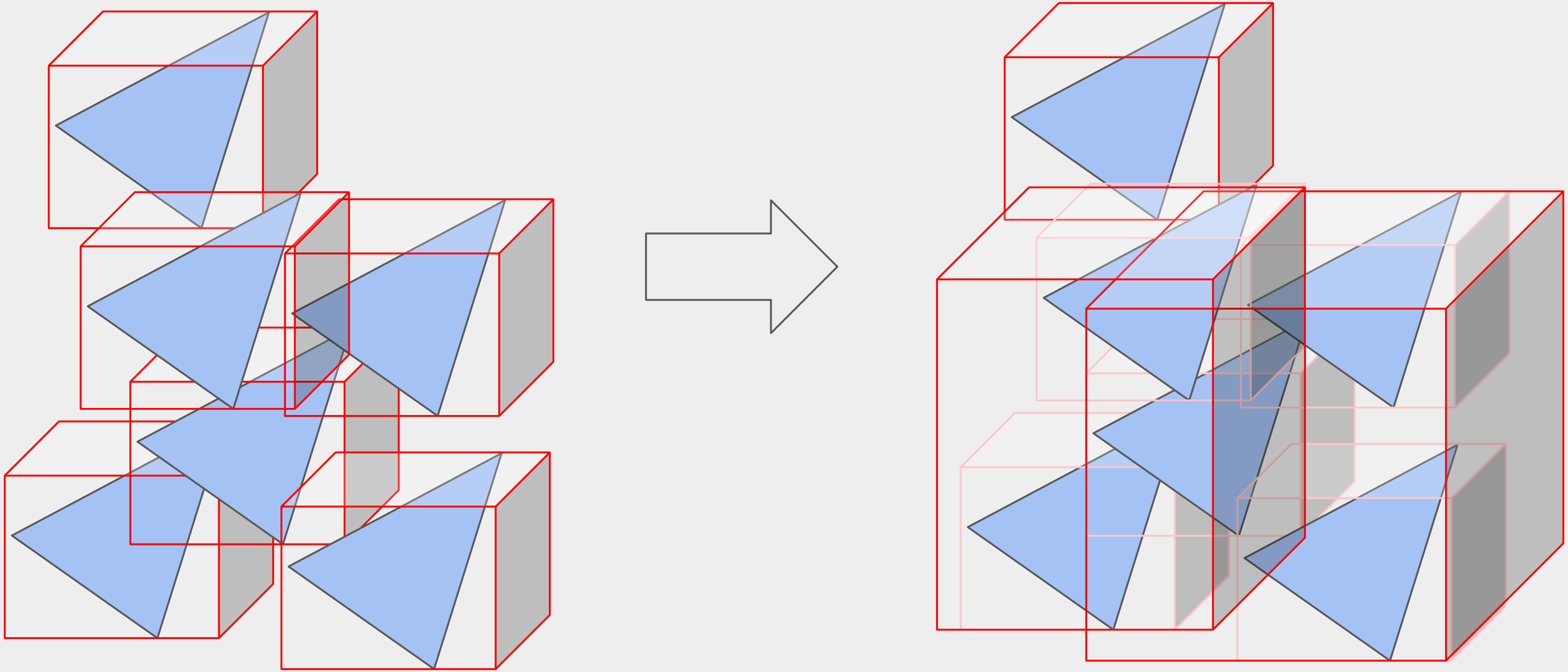
```
bool BBox::hit(const Ray &ray, Vec2
&times);
```





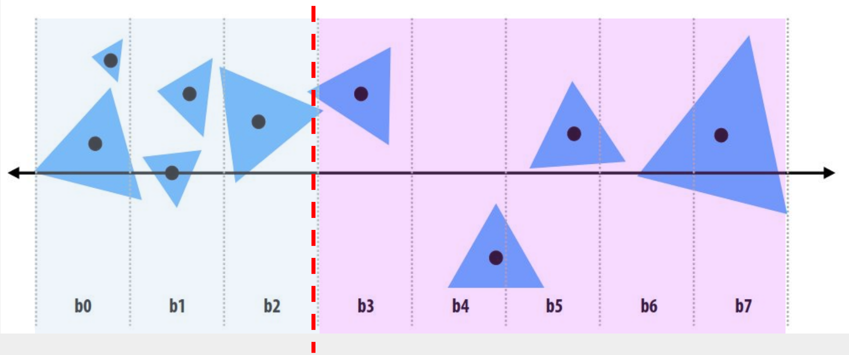
Task 3: BVH

How do I partition the 3D space (and the mesh)?



Task 3: BVH

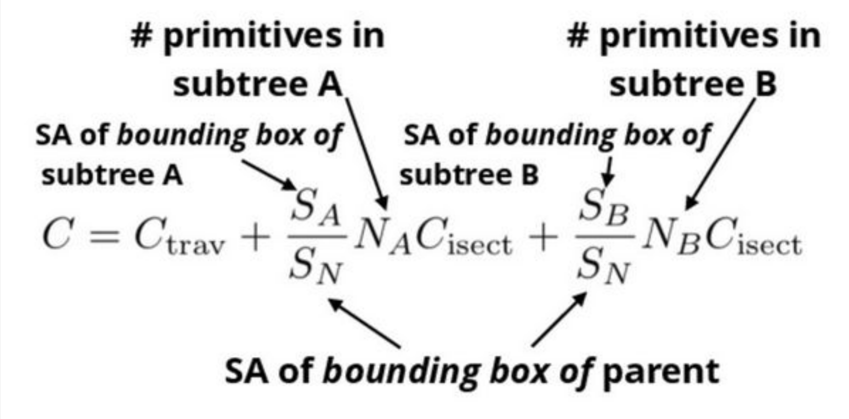
Want to optimally sort Primitives into buckets (or partitions)



Bucket A

Bucket B

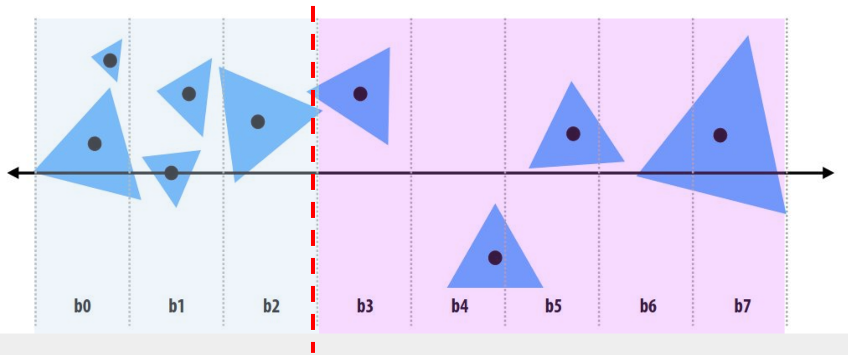
Surface Area Heuristic



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

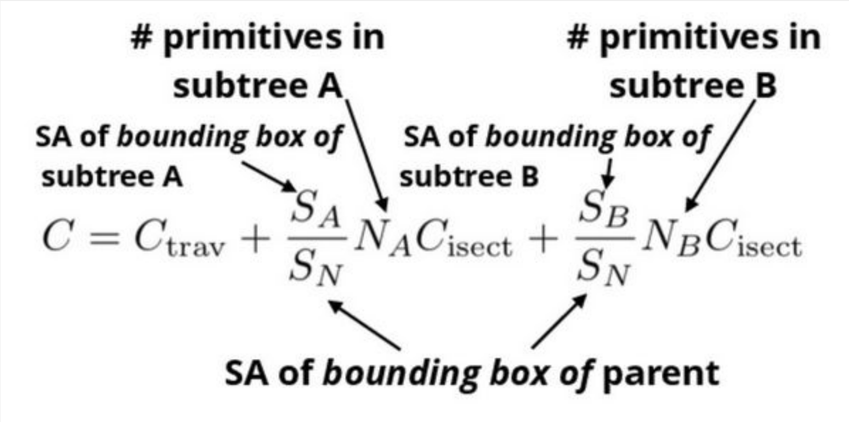
Task 3: BVH

Want to optimally sort Primitives into buckets (or partitions)



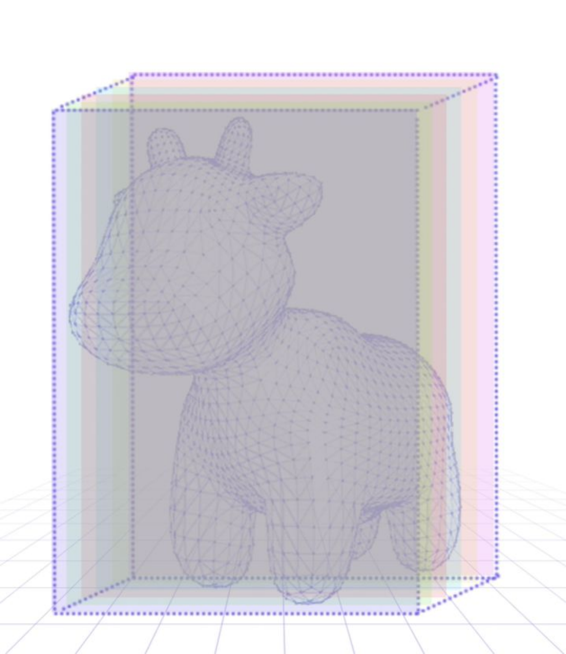
Bucket A   Bucket B

Surface Area Heuristic



C_{trav} , C_{isect} , S_N are constants

Task 3: BVH



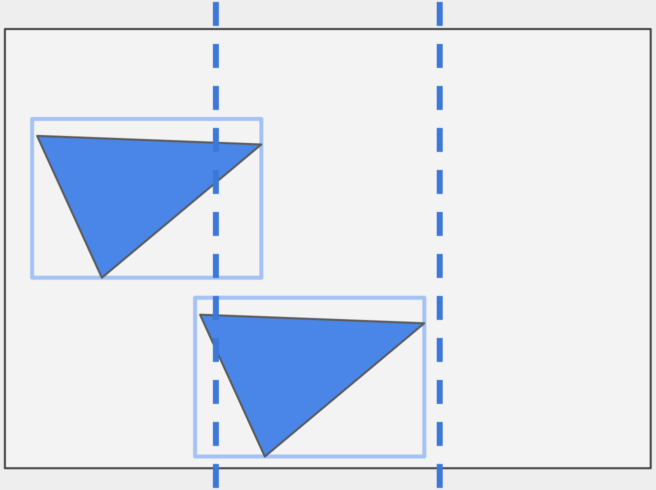
For each partition along the XYZ axis:

1. Define buckets along the line of partition
2. Calculate the BBox for the bucket based on the primitives in the bucket
3. Keep track of the best optimal partition

Construct the BVH based on the lowest cost partition found, and recurse on it(or make node leaf)

Task 3: BVH

Step 1: BVH Construction



```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

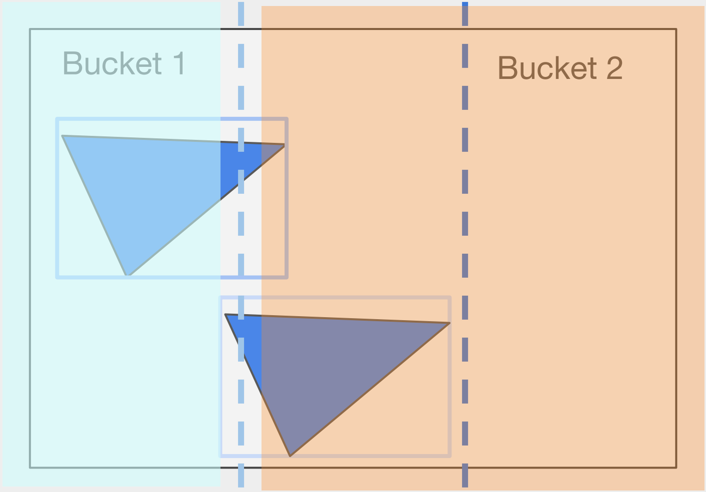
Task 3: BVH

Step 1: BVH Construction

Bucket = result from a possible (but maybe not the best) partition

In code: some variable that you will have to keep track of

Partition along x = 1, 2, 3 …




```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

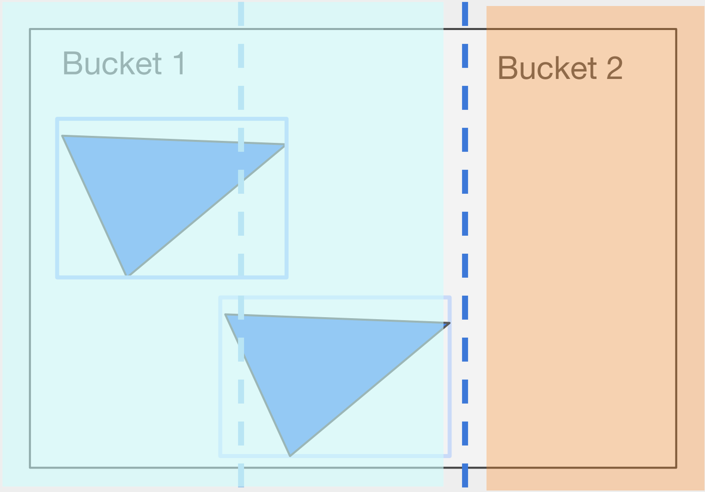
Task 3: BVH

Bucket = result from a possible (but maybe not the best) partition

In code: some variable that you will have to keep track of

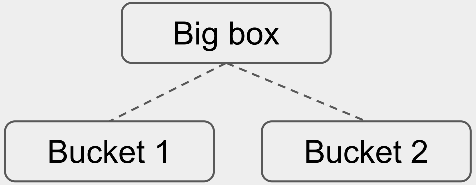
Partition along x = 1, 2, 3 …

Step 1: BVH Construction



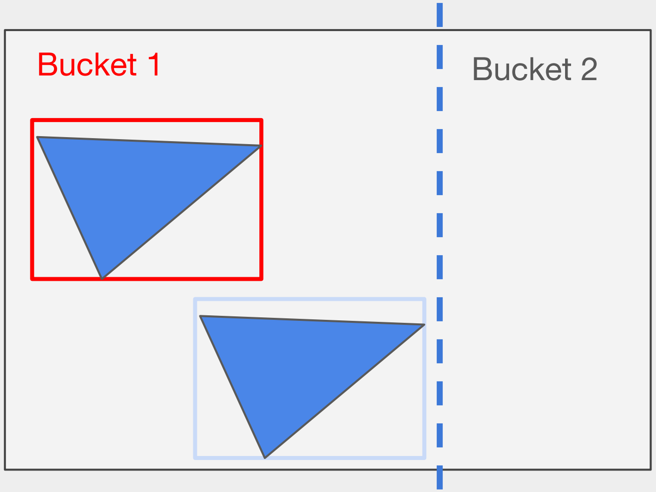

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH



<< nodes

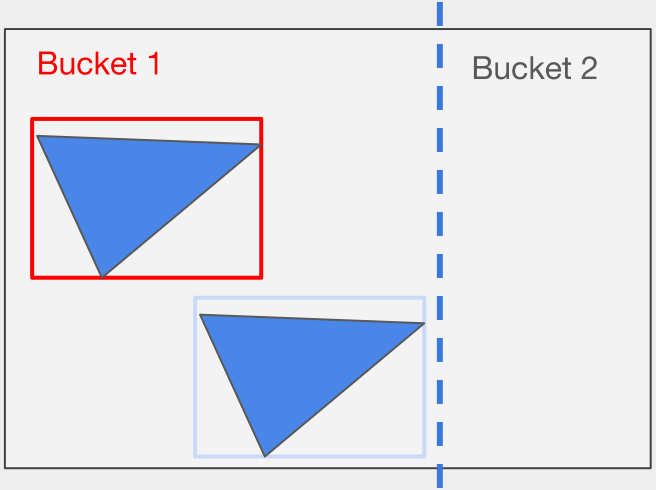
Step 1: BVH Construction




```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH

Step 1: BVH Construction



Bucket 1 Count: 1

Bucket 2 Count: 0

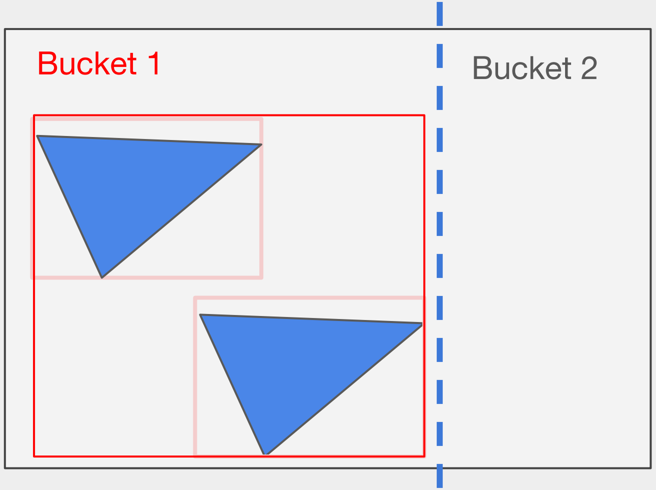
```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH

Step 1: BVH Construction

Partition 1

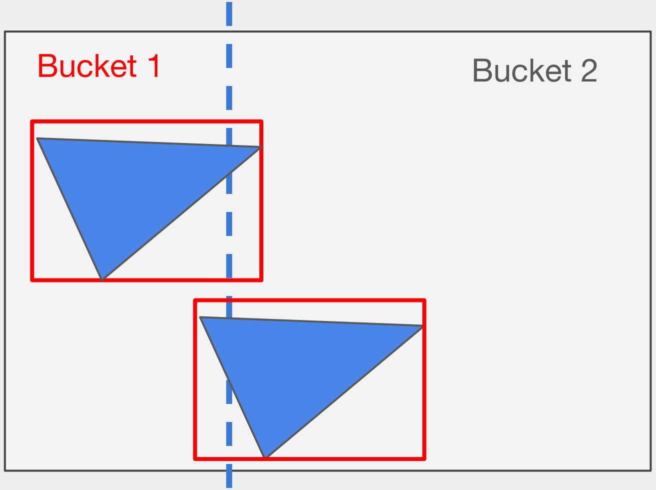
Bucket 1 Count: 2
Bucket 2 Count: 0




```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH

Step 1: BVH Construction



Partition 1

Bucket 1 Count: 1
Bucket 2 Count: 1

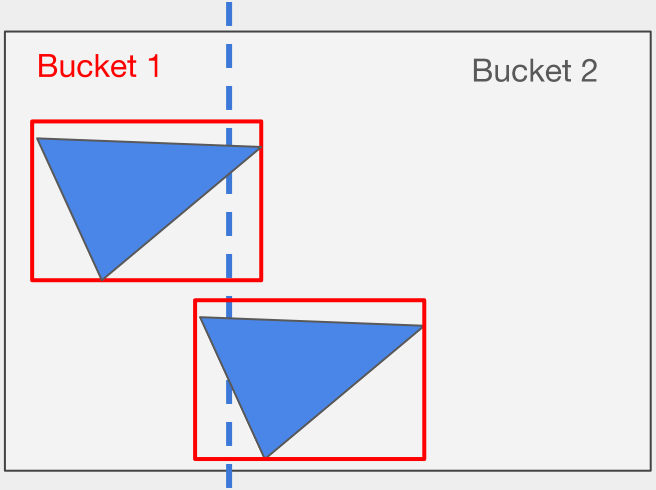
Partition 2

Bucket 1 Count: 2
Bucket 2 Count: 0

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH

Step 1: BVH Construction



Partition 1

Bucket 1 Count: 1
Bucket 2 Count: 1
SAH: big

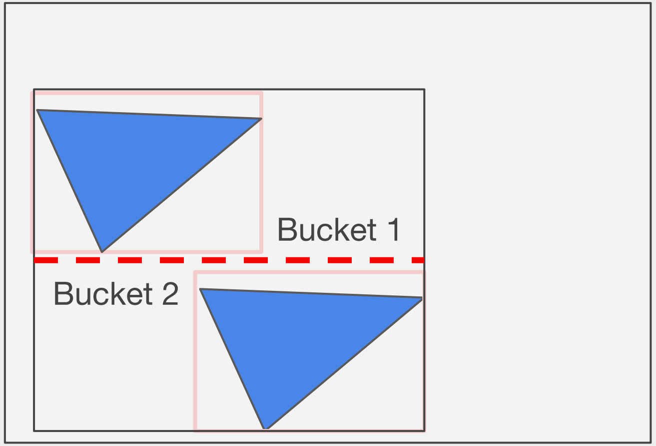
Partition 2

Bucket 1 Count: 2
Bucket 2 Count: 0
SAH: small

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Task 3: BVH

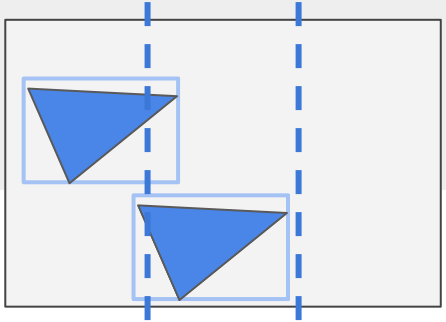
Step 1: BVH Construction



Stop if primitives <= `max_leaf_size`

```
For axis x,y,z:
    Initialize buckets
    For each primitive p in node:
        B = compute_bucket(p.centroid)
        B.bbox.enclose(p.bbox)
        B.prim_count++
    For each of |B| - 1 possible
    partitions
        Evaluate cost (SAH), keep track
        of lowest cost partition
Recurse on lowest cost partition found
(or make node leaf)
```

Another way to think about it



For axis *x*, *y*, *z*:

  For partitions *partition* along current *axis*:

    divide primitives into *left*, *right* according to *partition*

    evaluate *SAH cost*

    keep track of the *best partition*

Recurse on *best partition*

Helpful functions!

```
auto it = std::partition(primitives.begin() + bdata.start,
                         primitives.begin() + bdata.start + bdata.range,
                         [split_dim, split_val](const Primitive& p) {
                           return p.bbox().center()[split_dim] < split_val;
                       });
```

```
std::partition(v.begin(), v.end(), [](int i){return i % 2 == 0;});
```

```
Original vector:
    0 1 2 3 4 5 6 7 8 9
Partitioned vector:
    0 8 2 6 4  *  5 3 7 1 9
Unsorted list:
    1 30 -4 3 5 -4 1 6 -8 2 -5 64 1 92
```

***** is where auto **it** is at

Note that the elements are not sorted within the subgroups themselves. You may want to use std::sort to sort them.

Helpful functions!

`Bbox.enclose` - lets one box enclose another

`Bbox.center` - center of the box

`primitives(in build)` - vector/array of primitives

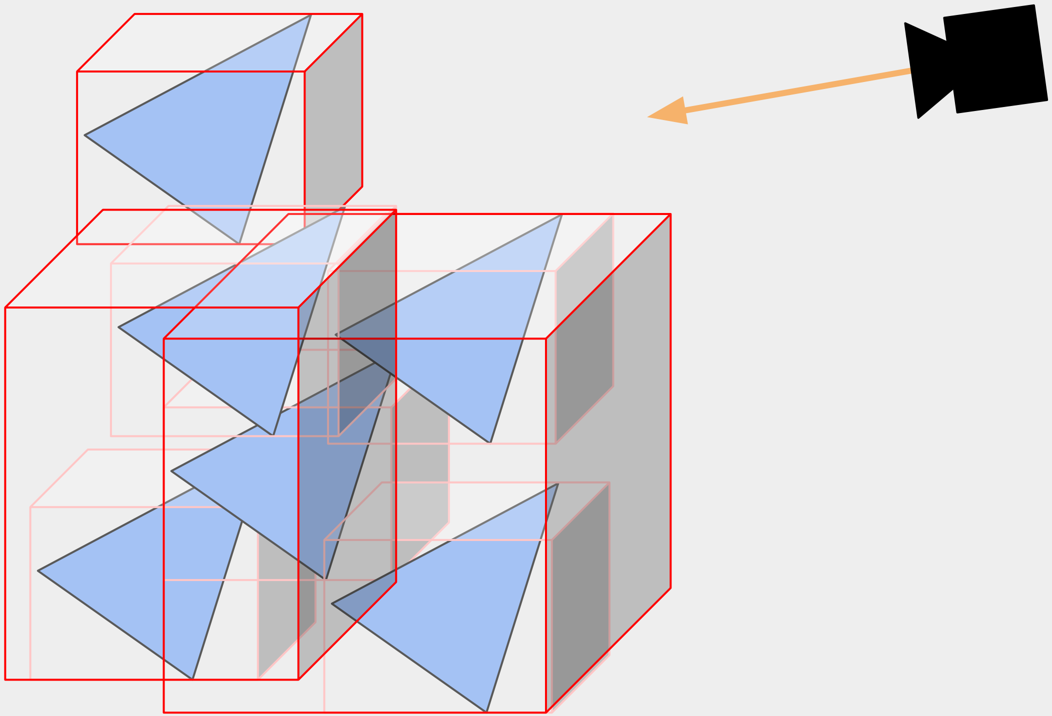
`Node` - used to construct the tree

```
class Node {
    BBox bbox;
    size_t start, size, l, r;
```

- Camera Rays
- Intersections
- BVH Construction
- BVH Navigation

BVH Navigation

So how do we actually find out which primitive a light ray is hitting?



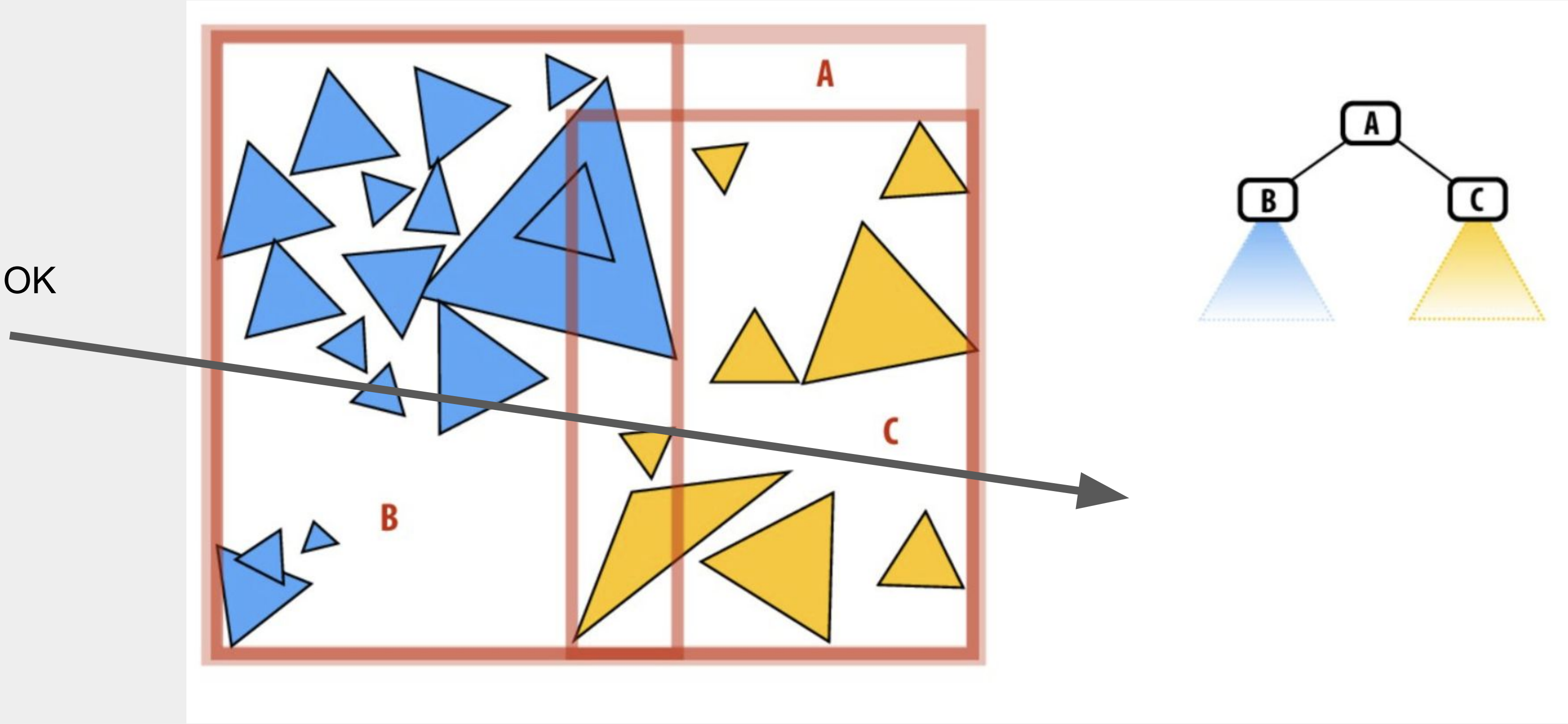
BVH Traversal

Implement `Trace BVH<Primitive>::hit(const Ray& ray);`

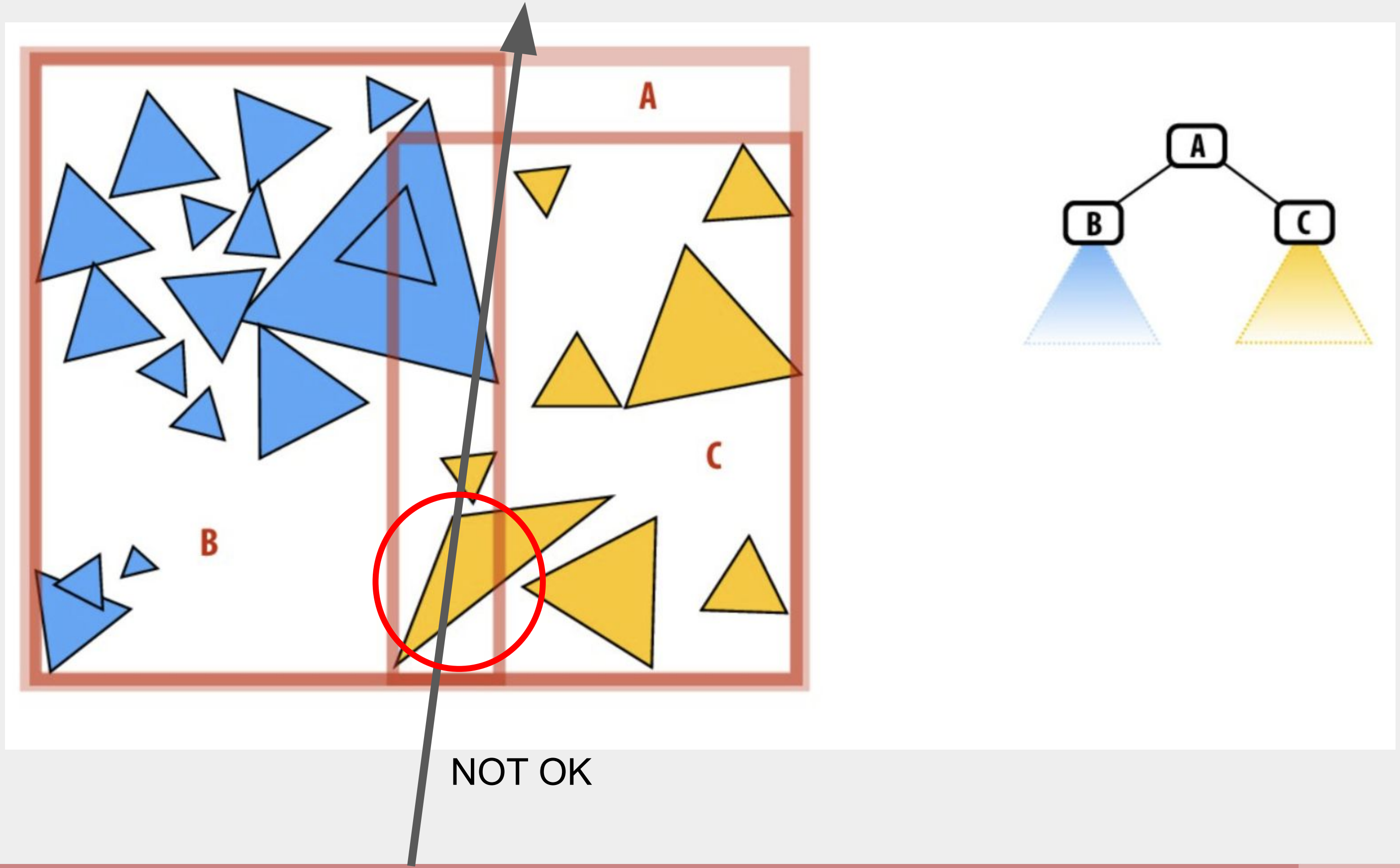
```
struct BVHNode {
  // is the node a leaf
  bool leaf;
  // min/max coordinates enclosing primitives
  Bbox bbox;
  // left child (can be NULL)
  BVHNode *child1;
  // right child (can be NULL)
  BVHNode *child2;
  // for leaves, stores primitives
  Primitive *primList;
}

struct HitInfo {
  // the primitive the ray hit
  Primitive *prim;
  // the time along the ray the hit occured
  float t;
}
```

```
void hit(Ray* ray, BVHNode* node, HitInfo* best)
{
  if (node->leaf) {
    // check all primitives in leaf for closest
  } 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 = (hit1.t <= hit2.t) ?
                       child2 : child1;

    hit(ray, first, best);
    if (hit2.t < best.t)
      hit(ray, second, best);
  }
}
```

Why do we need to check the other child as well?



Why do we need to check the other child as well?



NOT OK