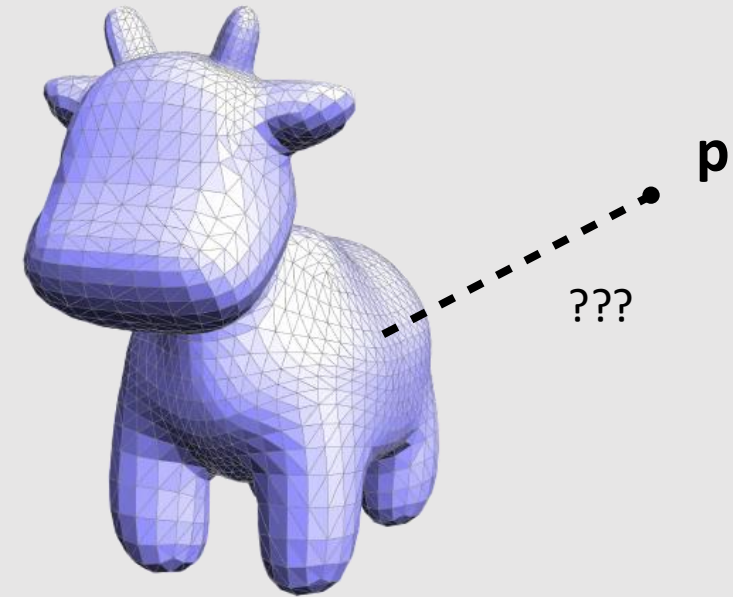


Geometric Queries and Spatial Data Structures

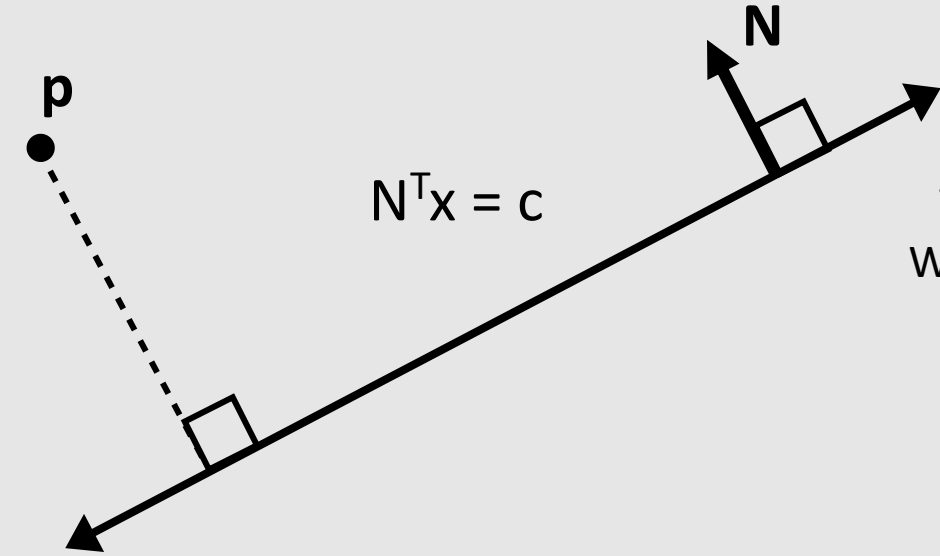
- **Geometric Queries**
- Ray-Triangle Intersections
- Bounding Volume Hierarchy
- Spatial-Partitioning Structures

Closest Point Queries

- **Problem:** given a point, in how do we find the closest point on a given surface?
- Several use cases:
 - Ray/mesh intersection in pathtracing
 - Kinematics/animation
 - GUI/user selection
 - When I click on a mesh, what point am I actually clicking on?



Closest Point on a Line



To find the closest point to p along $N^T x = c$
We can have p travel along N for some time t

$$N^T(p + tN) = c$$

Multiplying the terms out

$$N^T p + tN^T N = c$$

The unit norm multiplied by itself is 1

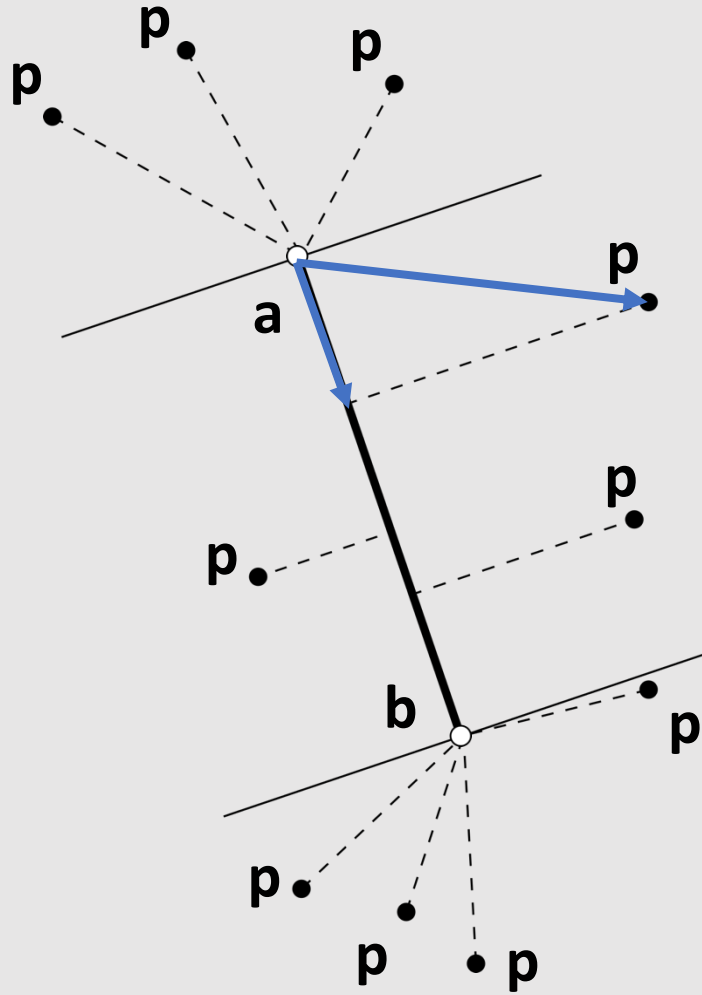
Solve for t

$$t = c - N^T p$$

Propagate p along N for time t

$$p + tN$$
$$p + (c - N^T p)N$$

Closest Point on a Line Segment



Compute the vector \mathbf{p} from the line base \mathbf{a} along the line

$$\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle$$

Normalize to get a time

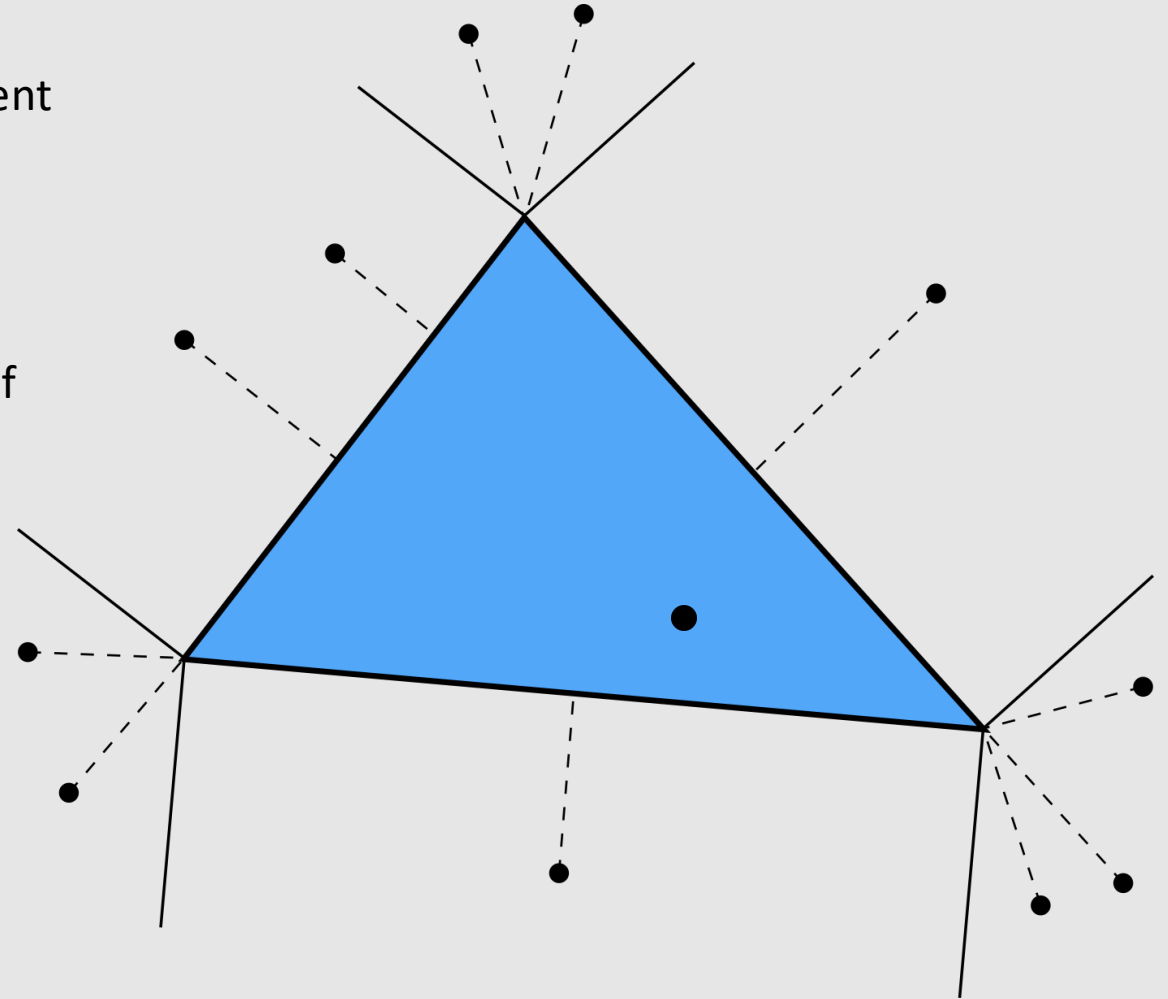
$$t = \frac{\langle \mathbf{p} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}{\langle \mathbf{b} - \mathbf{a}, \mathbf{b} - \mathbf{a} \rangle}$$

Clip time to range $[0,1]$ and interpolate

$$\mathbf{a} + (\mathbf{b} - \mathbf{a})t$$

Closest Point on a 2D Triangle

- Easy! Just compute closest point to each line segment
 - For each point, compute distance
 - Point with smallest distance wins
- What if the point is inside the triangle?
 - Even easier! The closest point is the point itself
 - Recall point-in-triangle tests



Closest Point on a 3D Triangle

- **Method #1: Projection****
 - Construct a plane that passes through the triangle
 - Can be done using cross product of edges
 - Project the point to the closest point on the plane
 - Same expression as with a line: $p + (c - N^T p)N$
 - Check if point is in triangle using half-plane test
 - Else, compute distance from each line segment in 3D
 - Same expression as with a 2D line segment
- **Method #2: Rotation****
 - Translate point + triangle so that triangle vertex v1 is at the origin
 - Rotate point + triangle so that triangle vertex v2 sits on the z-axis
 - Rotate point + triangle so that triangle vertex v3 sits in the plane x=0
 - Disregard x-coordinate of point
 - Problem reduces to closest point on 2D triangle

**<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.4264&rep=rep1&type=pdf>

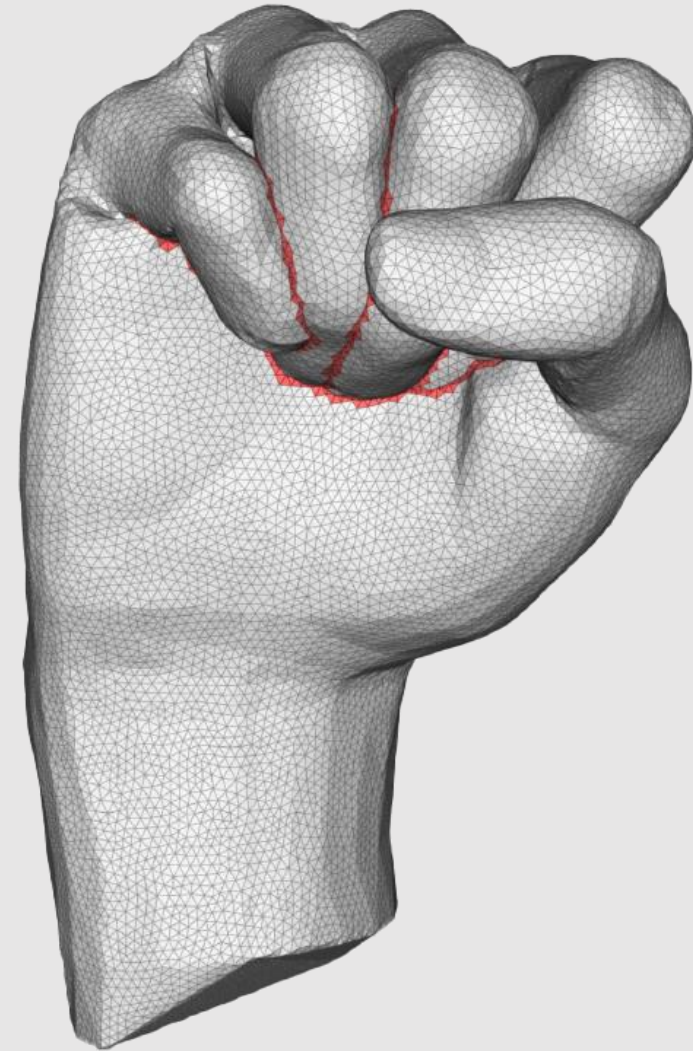
Closest Point on a 3D Triangle Mesh

- Conceptually easy!
 - Loop over every triangle
 - Compute closest point to current triangle
 - Keep track of globally closest point
- Not practical in real world
 - Meshes have billions of triangles
 - Programs make thousands of geometric queries a second
- Will look at better solutions a bit later

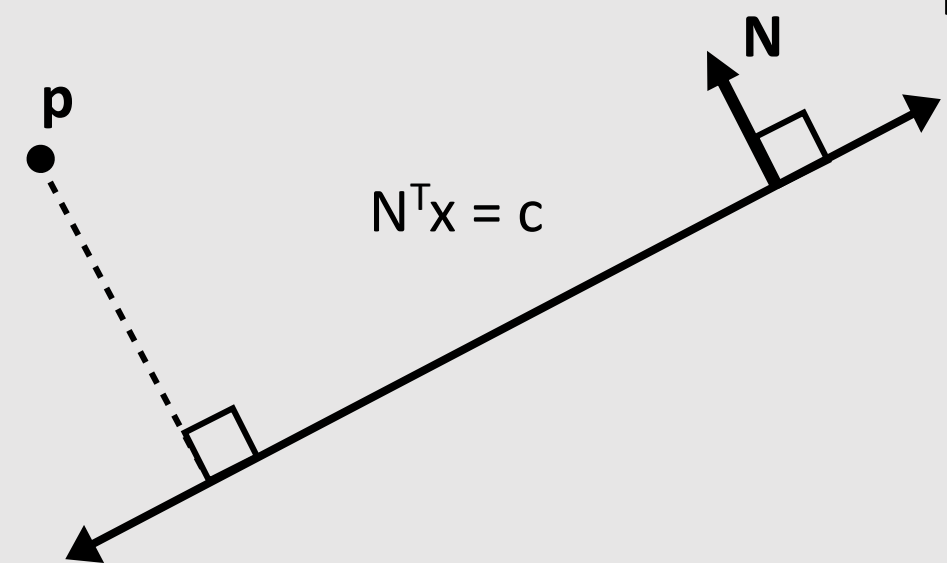


Mesh-Mesh Intersections

- Sometimes when editing geometry, a mesh will intersect with itself
- Likewise, sometimes when animating geometry, meshes will collide
- How do we check for/prevent collisions?



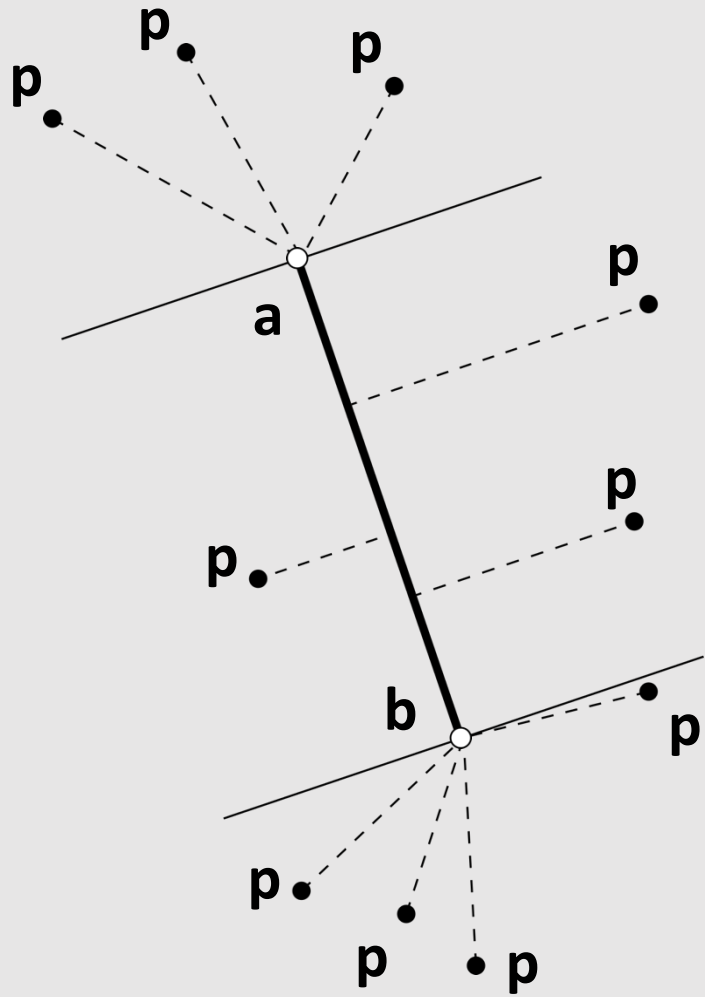
Point-Line Intersection



Just plug point in

$$N^T p = c?$$

Point-Line Segment Intersection



Check if adding distances equals net distance**

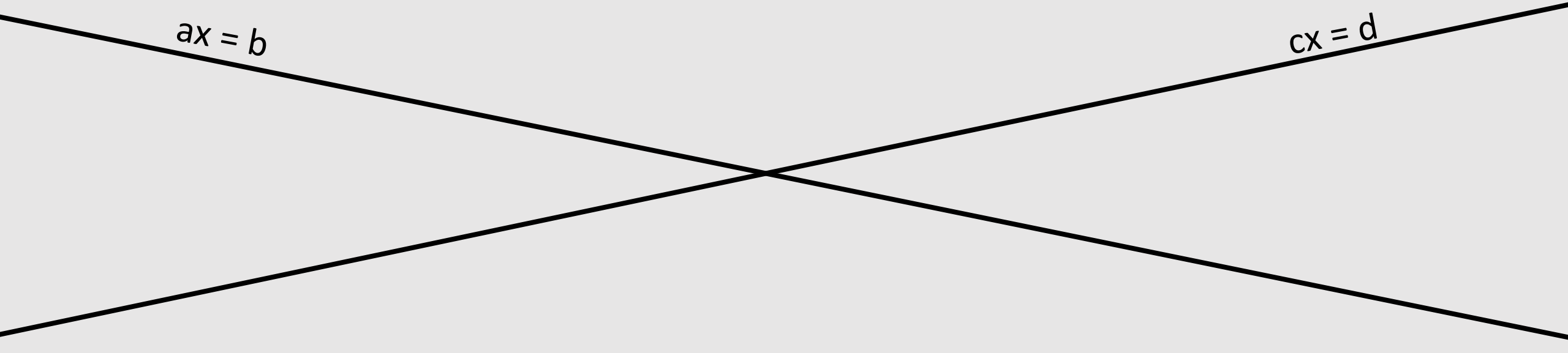
$$\text{dist}(a, p) + \text{dist}(p, b) = \text{dist}(a, b)$$

**Potential numeric stability issues

Line-Line Intersection

Two equations, two unknowns
Solve a linear system

$$\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

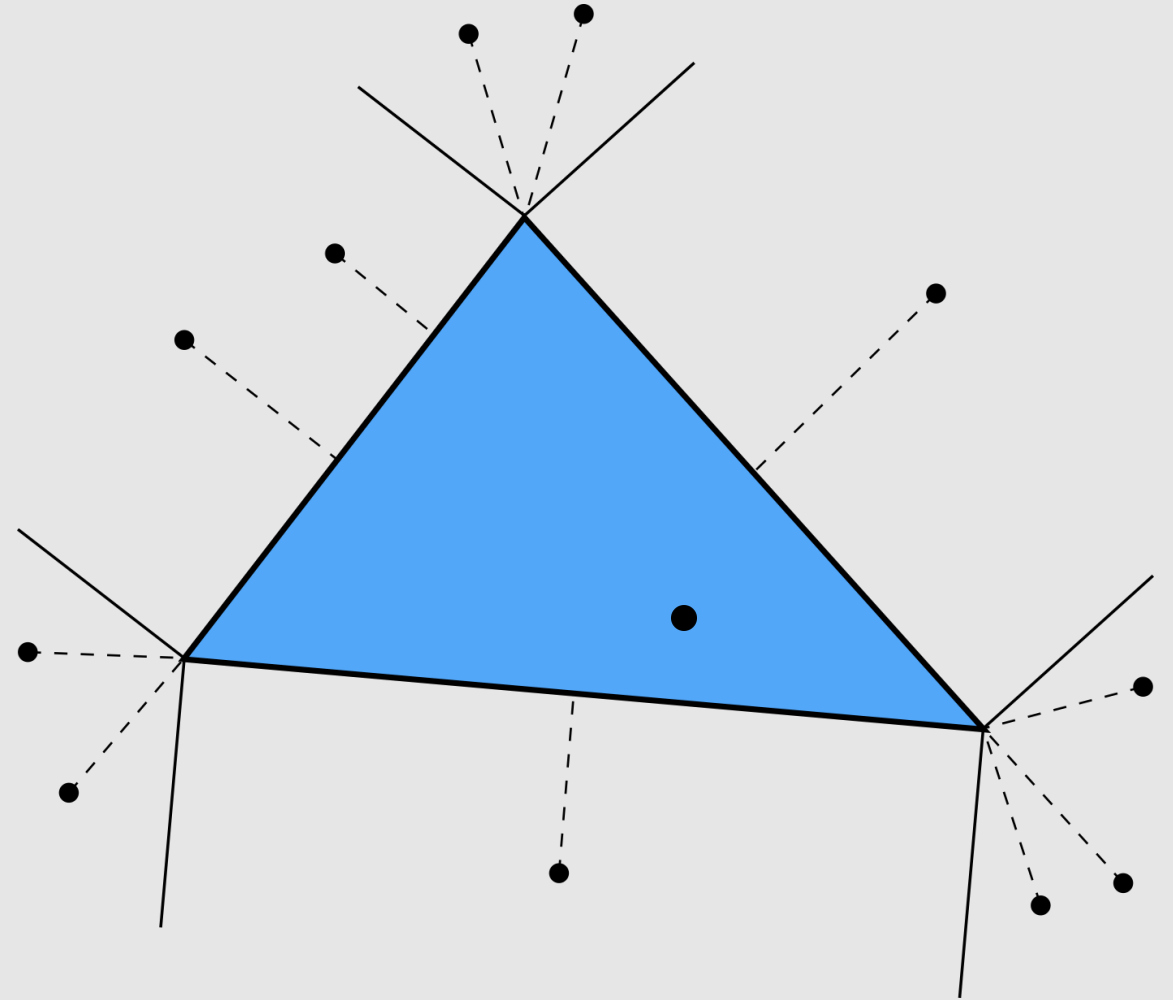


$ax = b$

$cx = d$

Point-Triangle Intersection

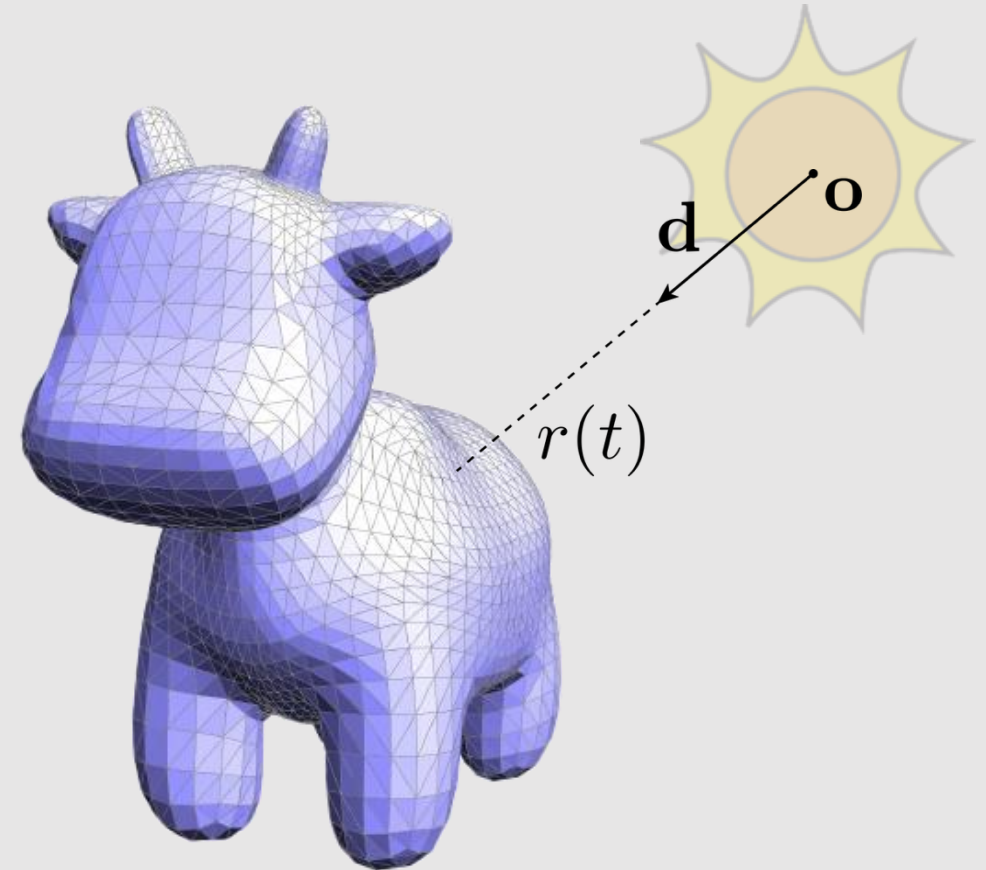
You know this :)



- Ray-Triangle Intersections
- Bounding Volume Hierarchy
- Spatial-Partitioning Structures

Ray-Mesh Intersection

- **We just saw** 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!



$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

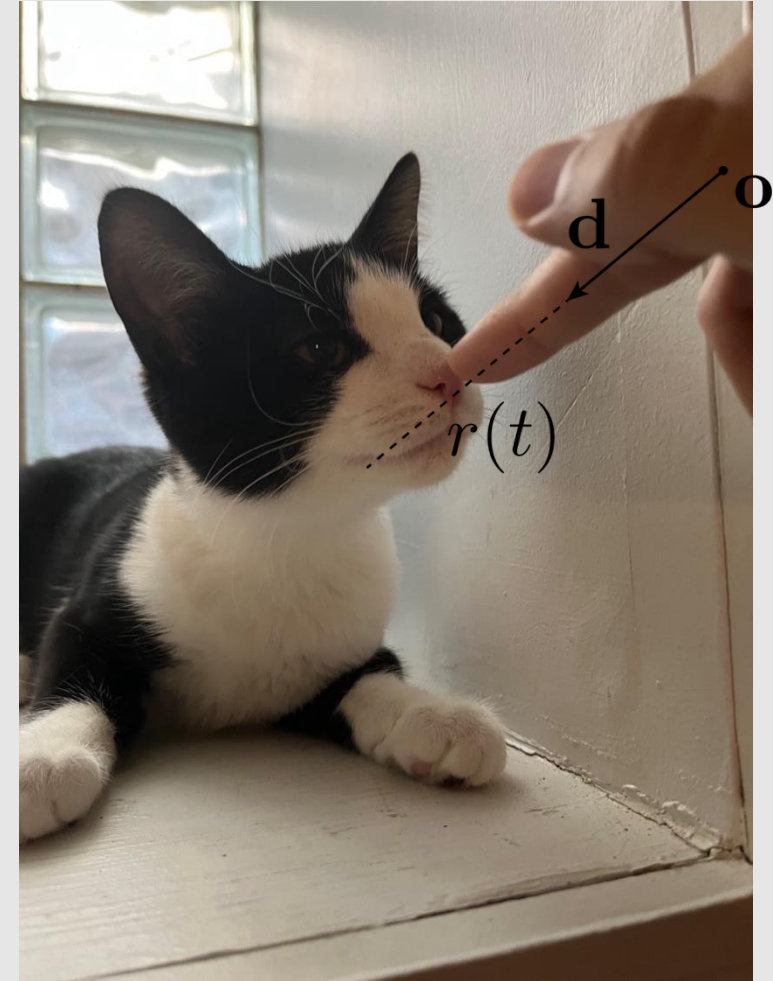
point along ray → $\mathbf{r}(t)$ ← time
origin → \mathbf{o} ← unit direction → \mathbf{d}

Ray-Mesh Intersection

- Spatial data structures that allows 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!

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

point along ray → $\mathbf{r}(t)$ ← time
origin → \mathbf{o} ← unit direction → \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}

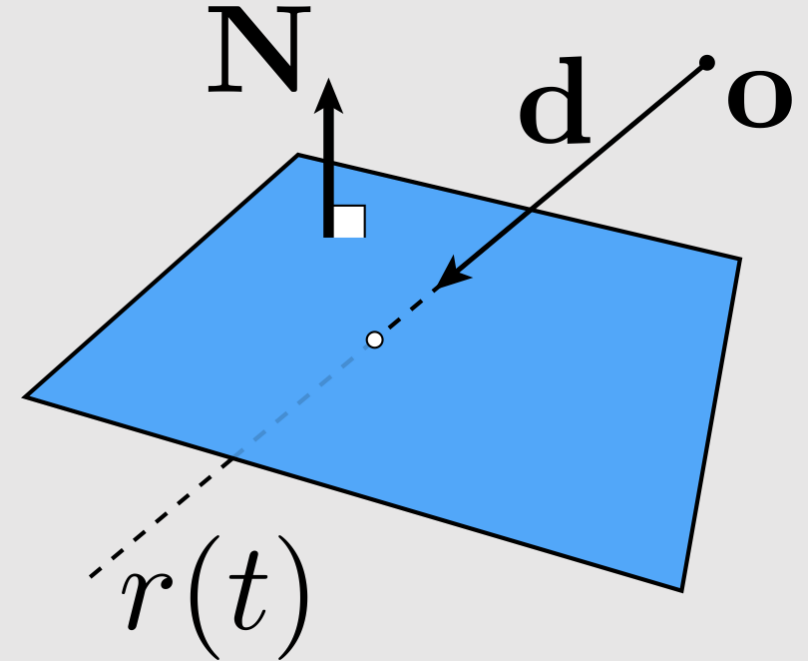
$$\mathbf{N}^T (\mathbf{o} + t\mathbf{d}) = c$$

Then solve for t

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

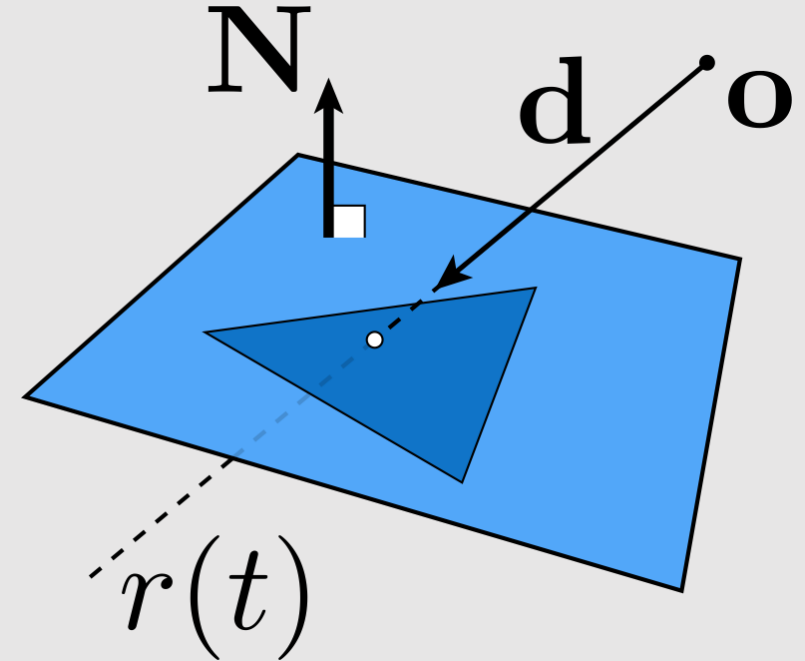
Substitute the time into the ray equation to find the intersection point

$$\mathbf{p} = \mathbf{o} + \left(\frac{c - \mathbf{N}^T \mathbf{o}}{\mathbf{N}^T \mathbf{d}} \right) \mathbf{d}$$



Ray-Triangle Intersection

- Not much different:
 - i) Compute ray-plane intersection to find point \mathbf{p} on plane
 - ii) Perform point-in-triangle test for point \mathbf{p}
 - Barycentric coordinates
- Not a very efficient algorithm...
 - Can we combine both steps into one?
 - **Idea:** set intersection and barycentric tests equal
- 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

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

Solve using Cramer's rule

$$\begin{aligned} \mathbf{s} &= \mathbf{o} - \mathbf{p}_0 \\ \mathbf{e}_1 &= \mathbf{p}_1 - \mathbf{p}_0 \\ \mathbf{e}_2 &= \mathbf{p}_2 - \mathbf{p}_0 \end{aligned}$$

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

$$\begin{aligned} a_1x + b_1y + c_1z &= d_1 \\ a_2x + b_2y + c_2z &= d_2 \\ a_3x + b_3y + c_3z &= d_3 \end{aligned} \quad \text{Let } D = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

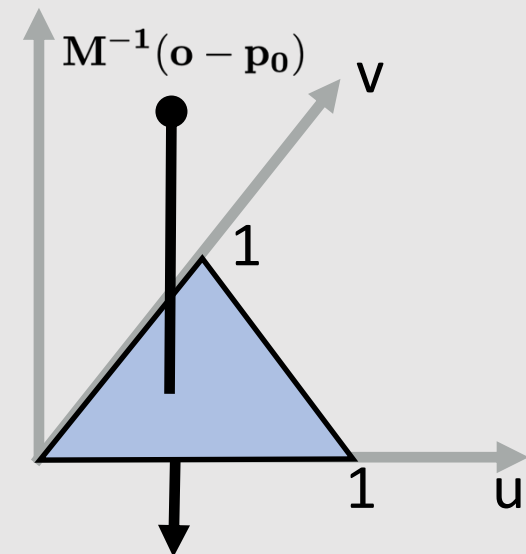
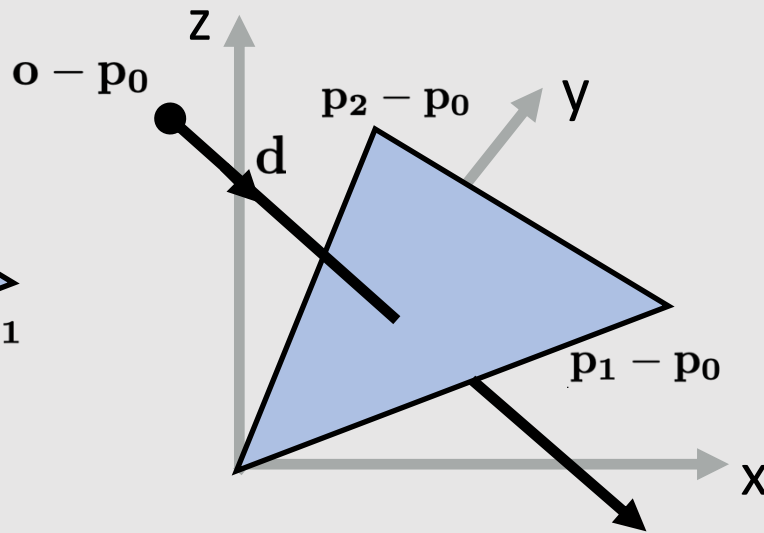
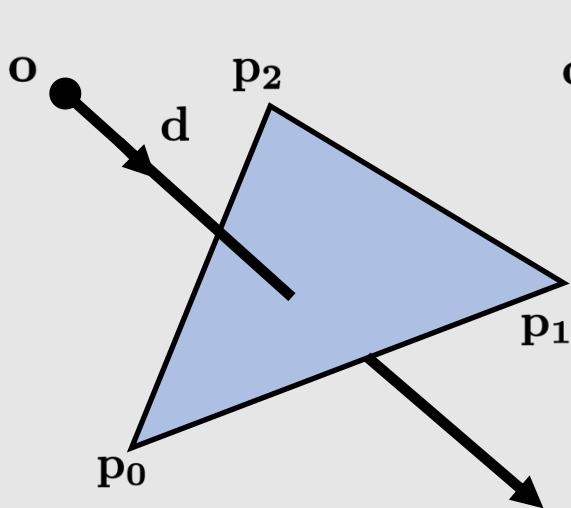
If $D \neq 0$ then

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{D} \quad y = \frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{D} \quad z = \frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{D}$$

Moller-Trumbore Visualized

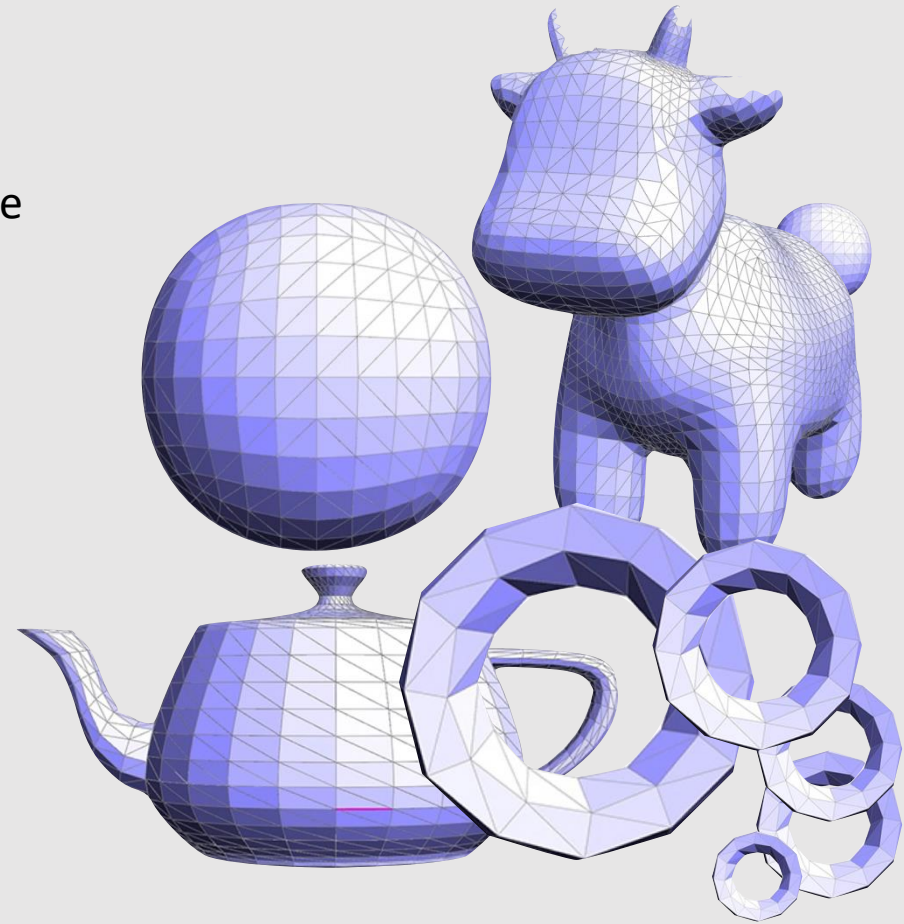
$$\begin{aligned}
 s &= [\mathbf{e}_1 \quad \mathbf{e}_2 \quad -\mathbf{d}] \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix} \\
 &= \\
 \mathbf{o} - \mathbf{p}_0 &= [\mathbf{p}_1 - \mathbf{p}_0 \quad \mathbf{p}_2 - \mathbf{p}_0 \quad -\mathbf{d}] \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix} \\
 &= \\
 \mathbf{o} - \mathbf{p}_0 &= \mathbf{M} \cdot \begin{bmatrix} u \\ v \\ t \end{bmatrix}
 \end{aligned}$$

- Matrix \mathbf{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!



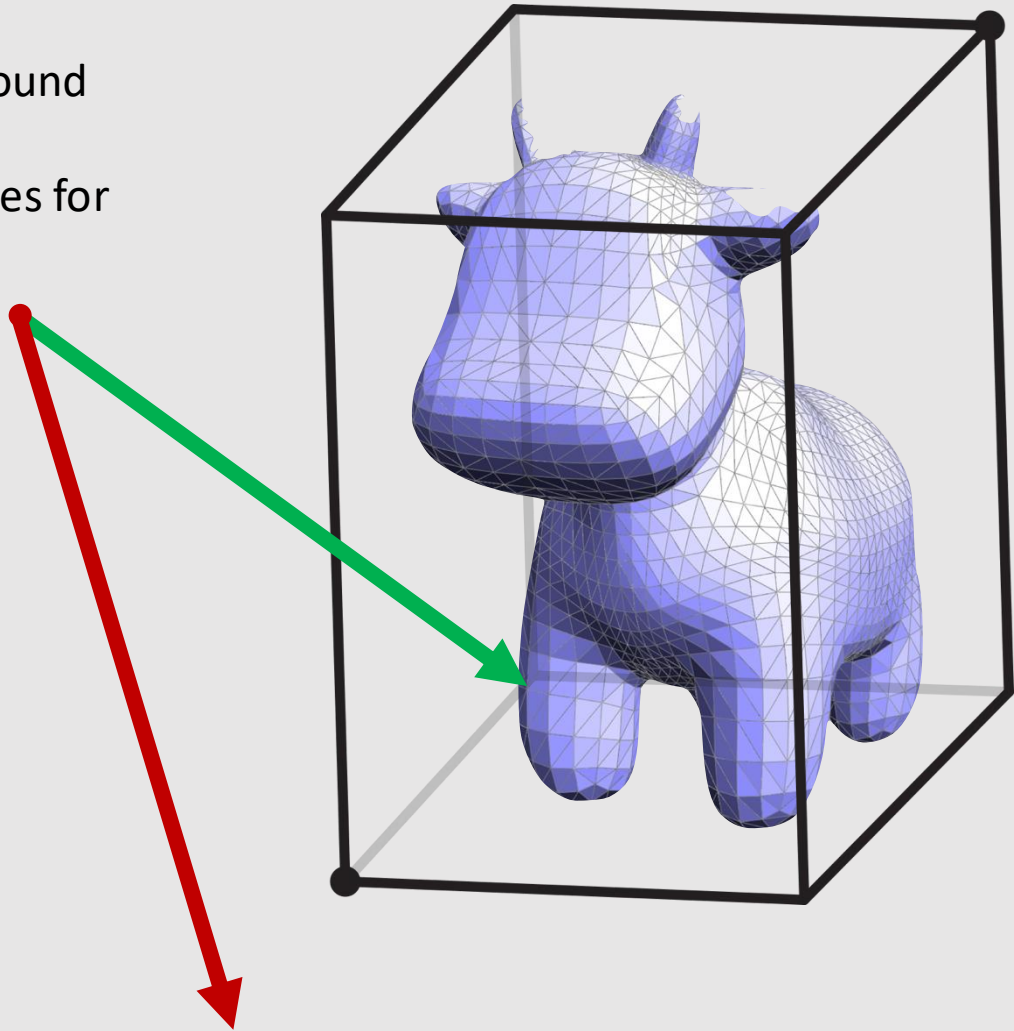
Spatial Data Structures

- 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

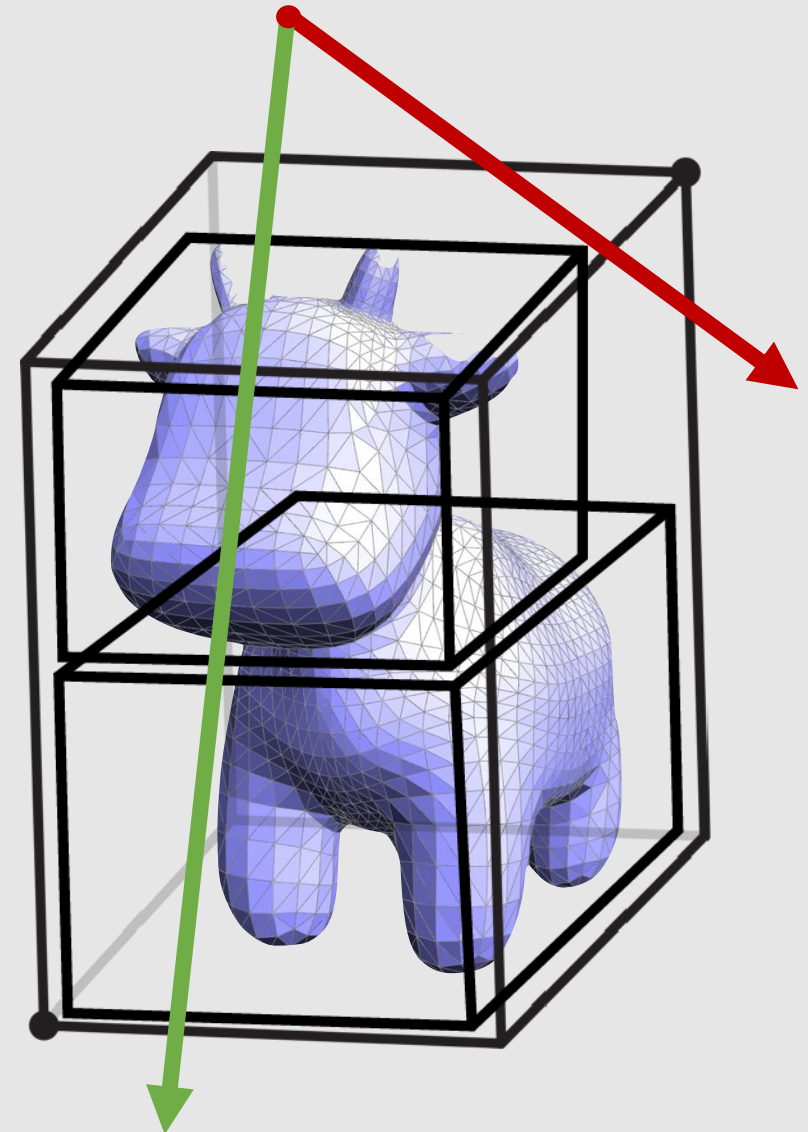
- Precompute the smallest axis-aligned bounding box around all primitives
 - Keep track of smallest and largest (x,y,z) coordinates for all primitives
- Check for ray-box intersection
 - If **misses**, we are done
 - If **passes**, check all triangles
- Saves time for rays that clearly miss the mesh, but...
 - Still $O(n)$ for rays that intersect the box



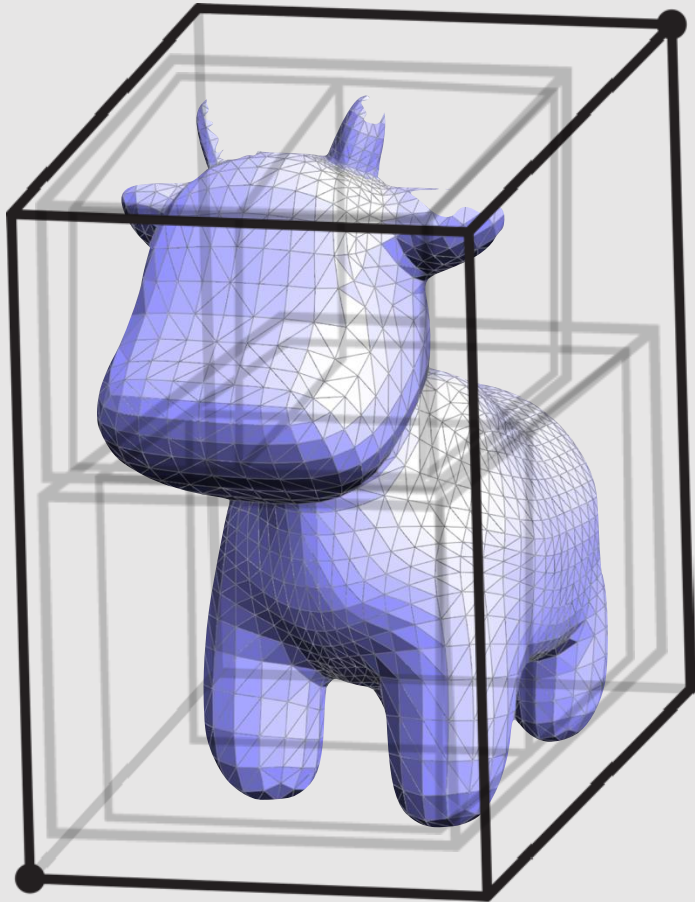
More Bounding Boxes

- 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

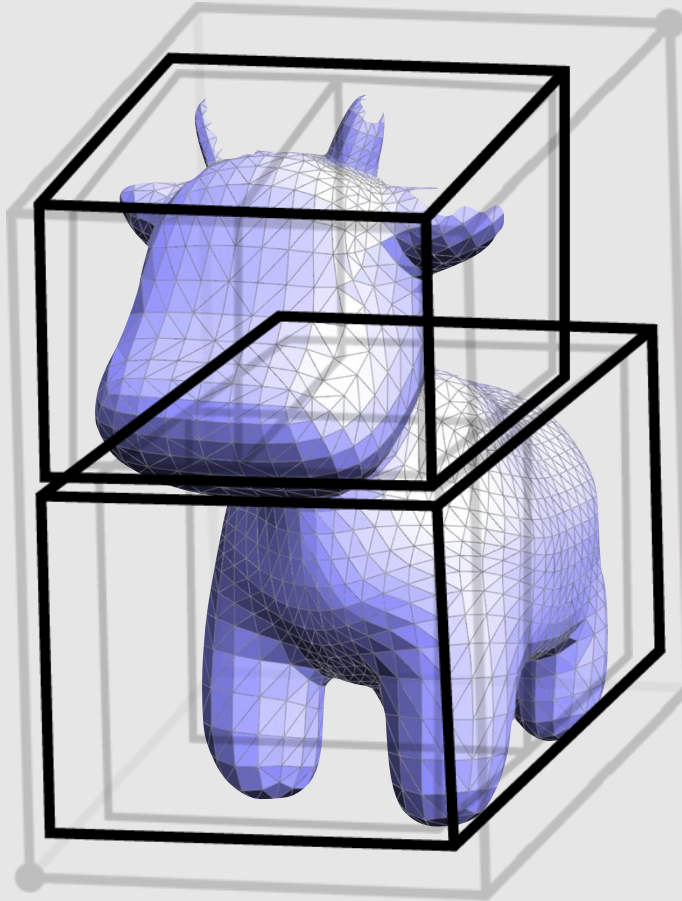
can we make this recursive?



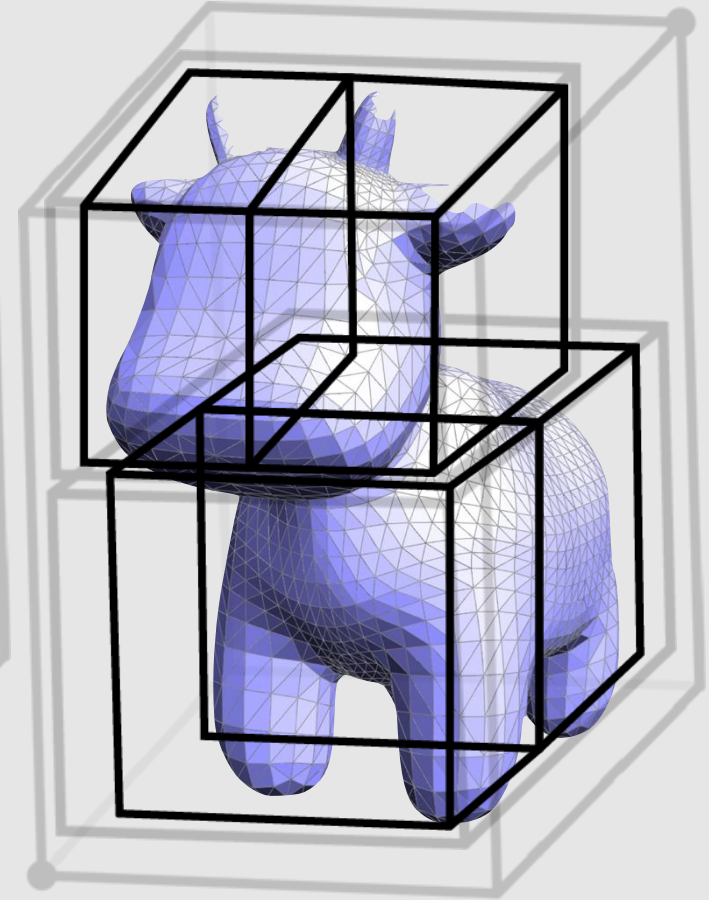
A Hierarchy of...Bounding Volumes?



[Level 0]



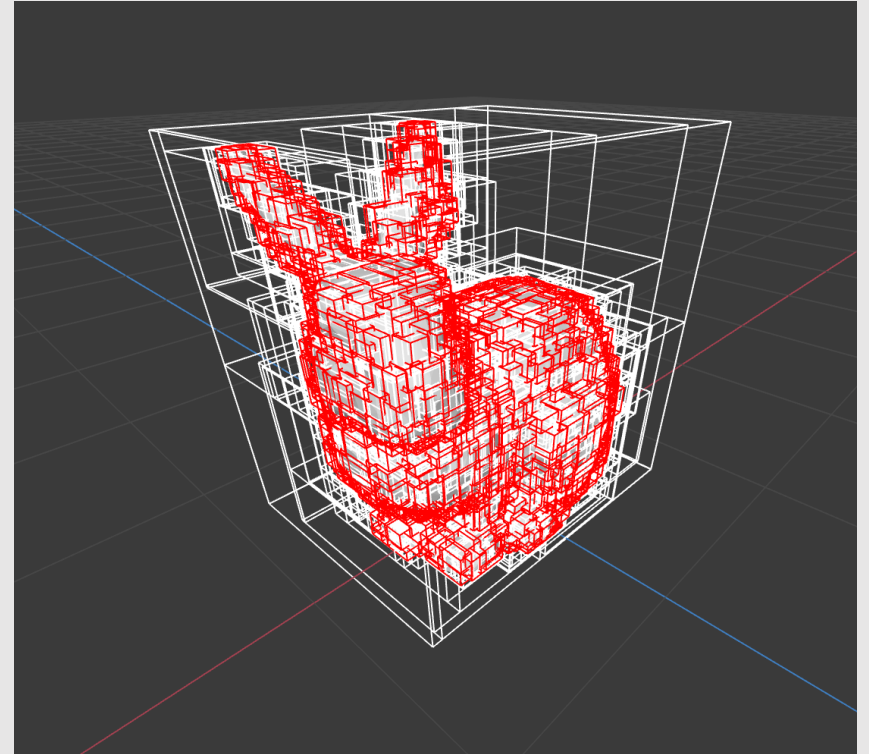
[Level 1]



[Level 2]

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

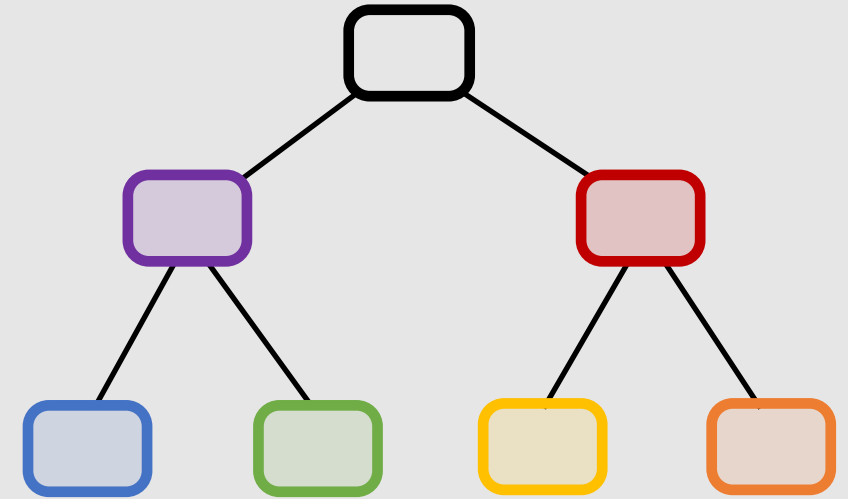
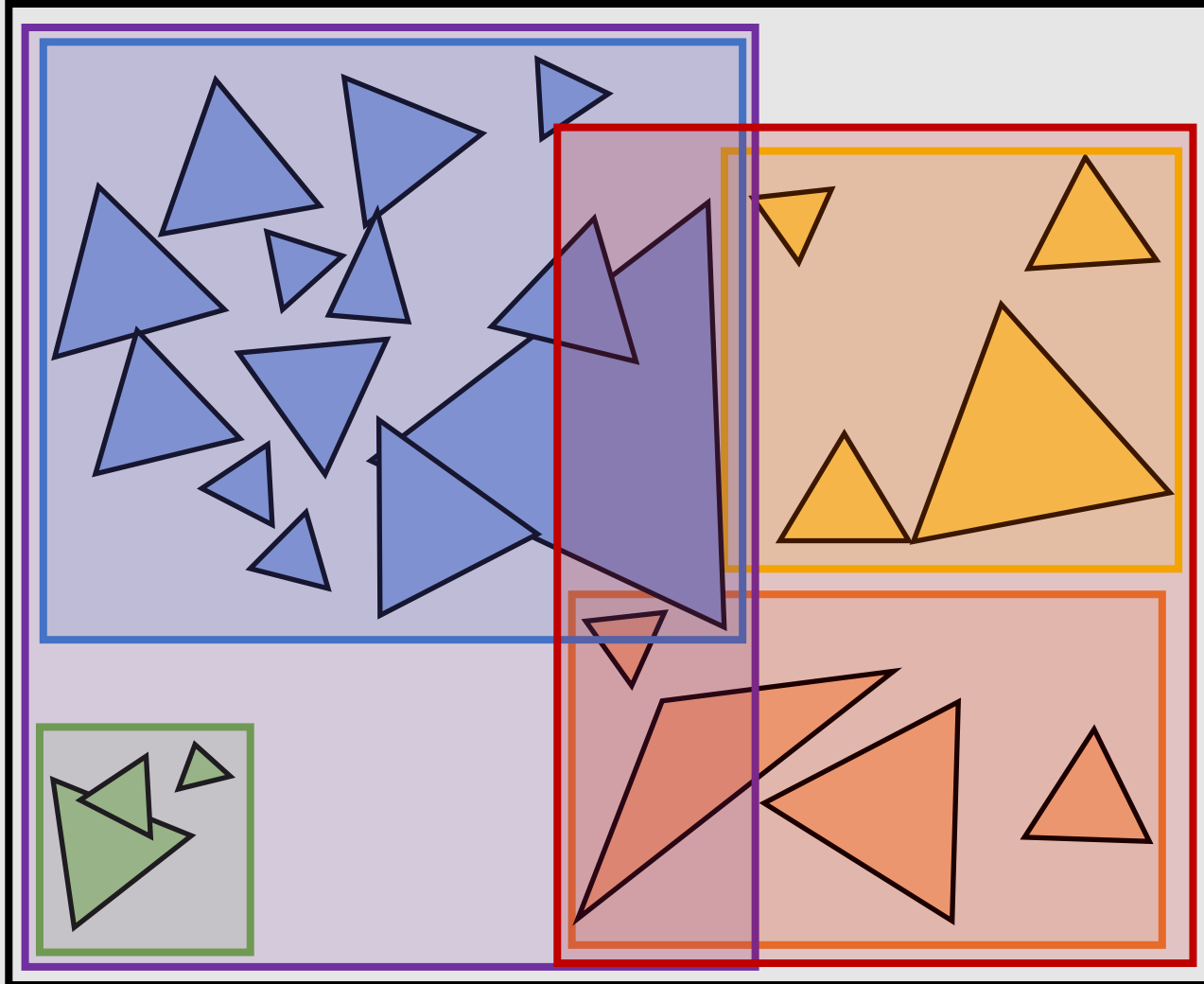


Stanford Bunny BVH visualizing 10th level

- ~~Ray-Triangle Intersections~~
- Bounding Volume Hierarchy
- Spatial-Partitioning Structures

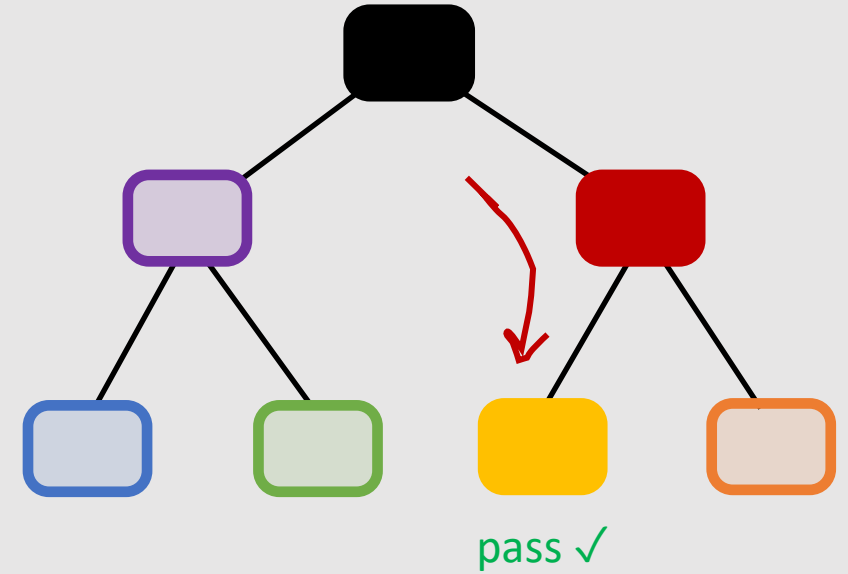
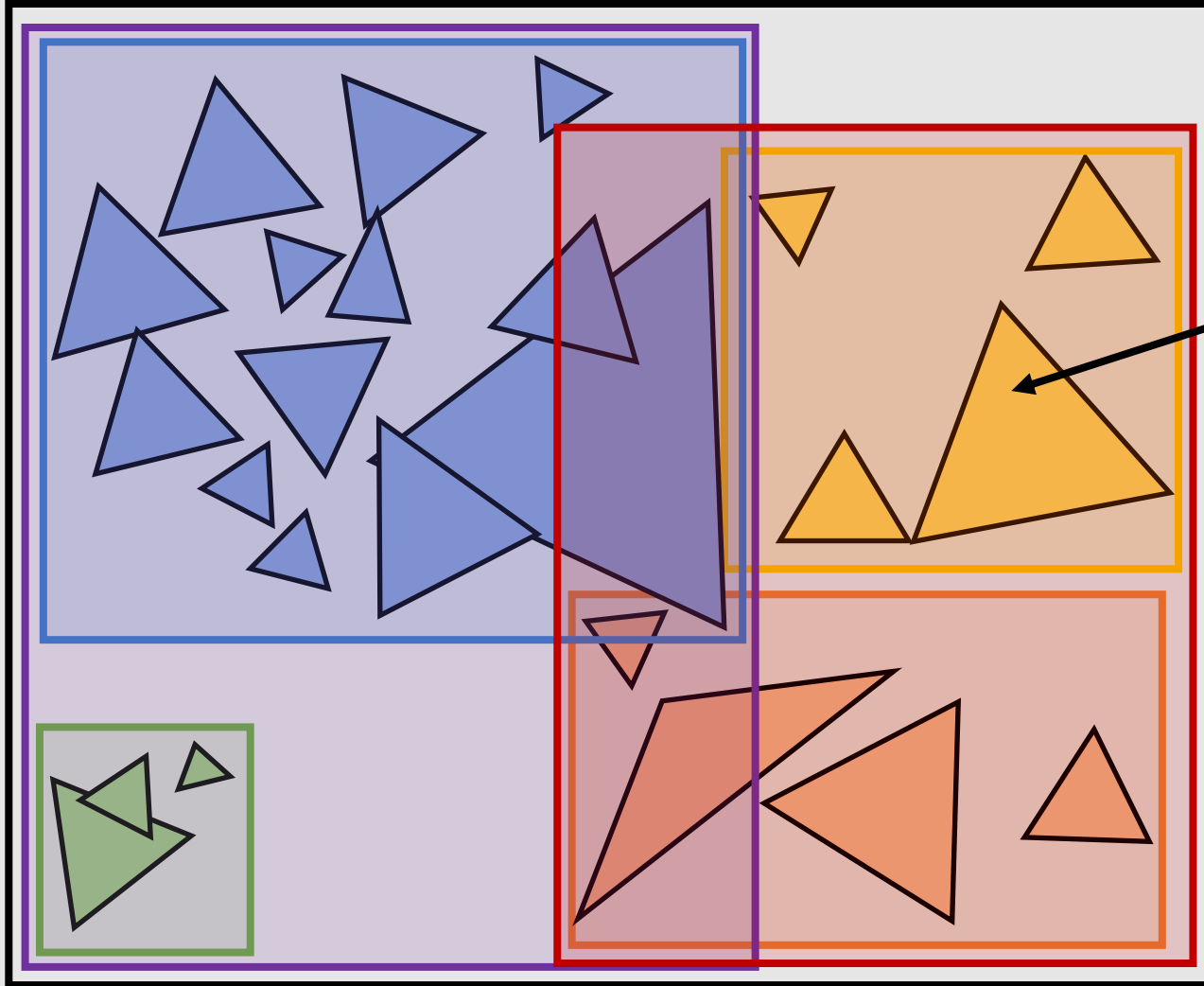
Let's look at an example

BVH Example

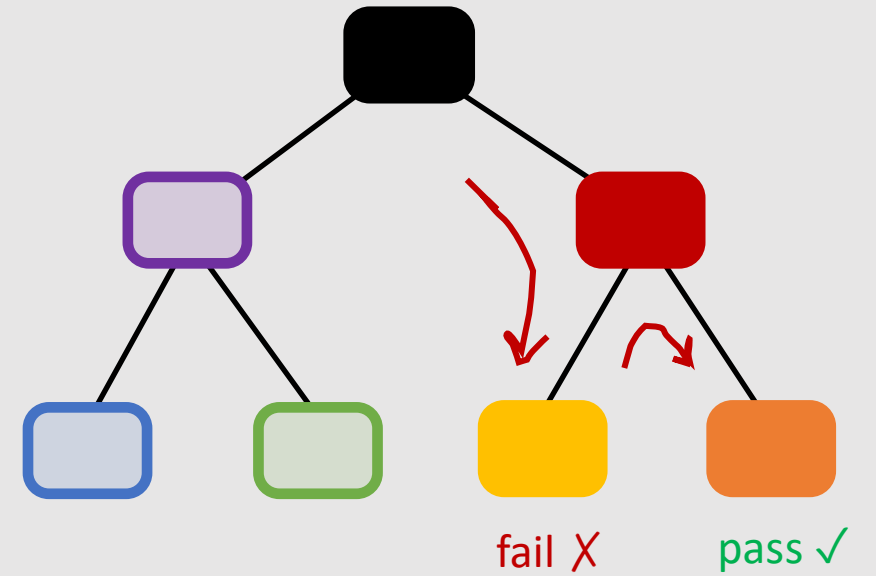
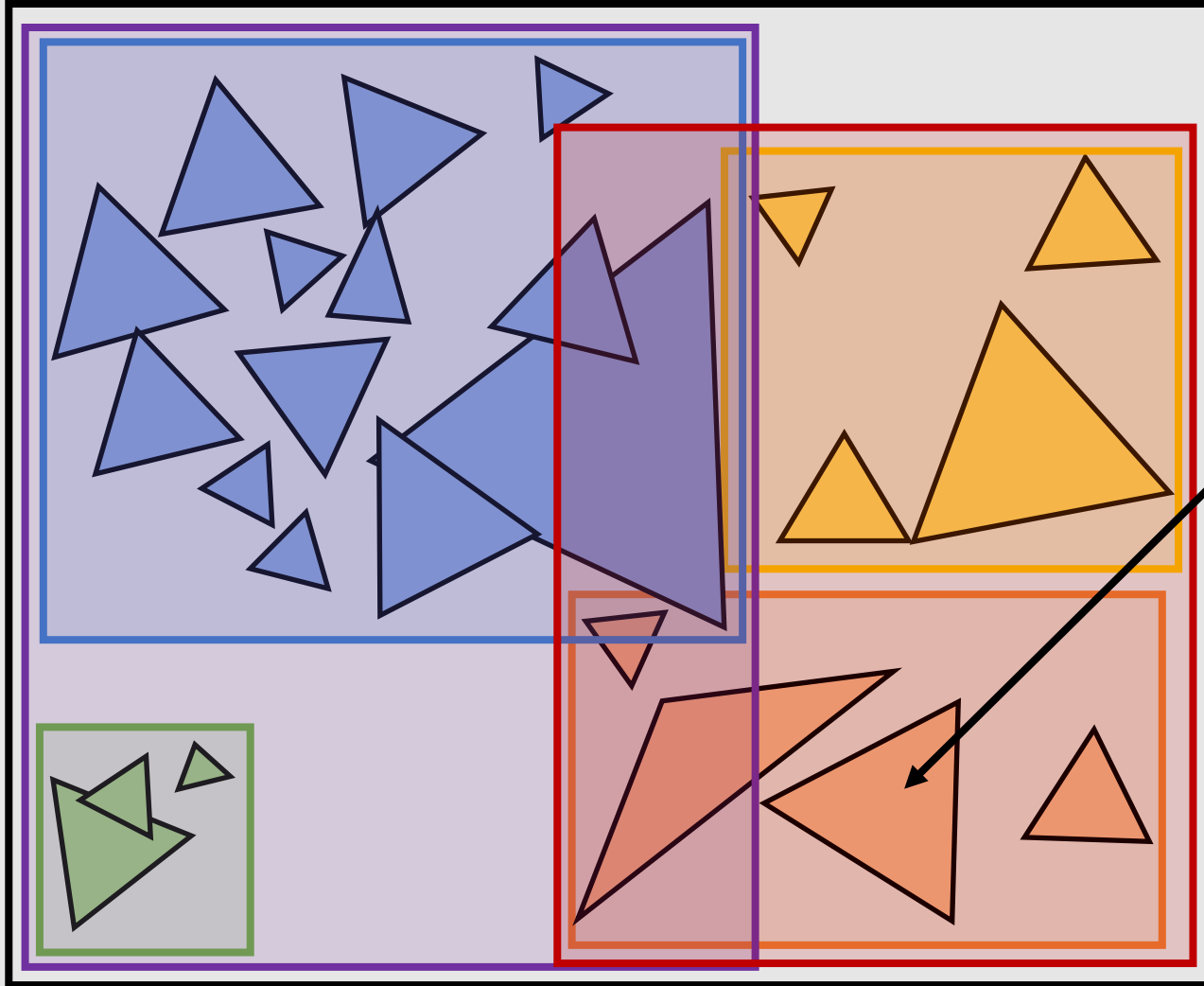


Bounding boxes will sometimes intersect!

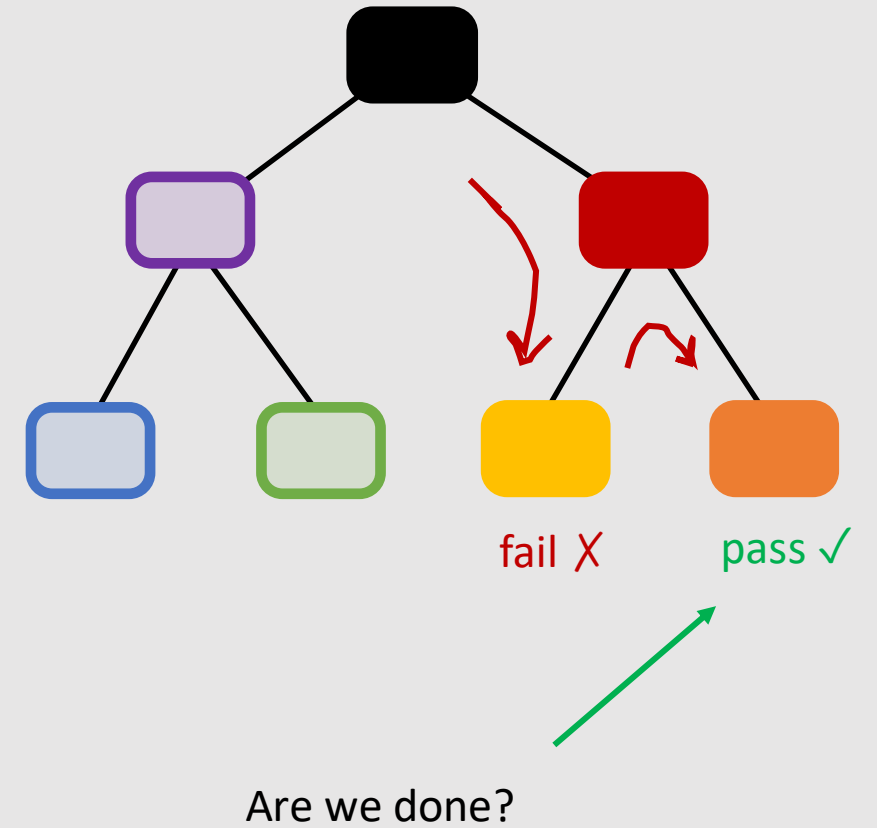
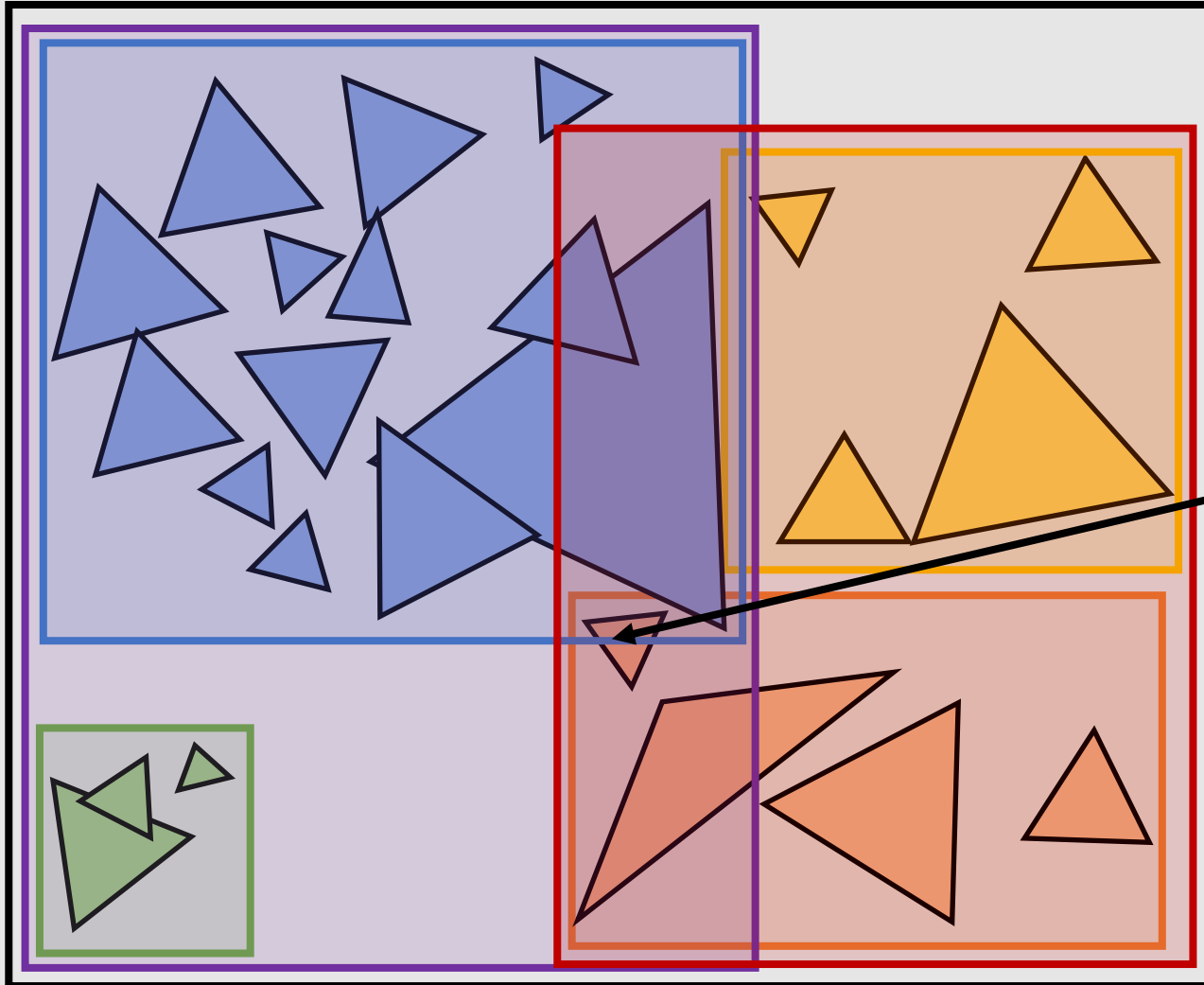
BVH Example



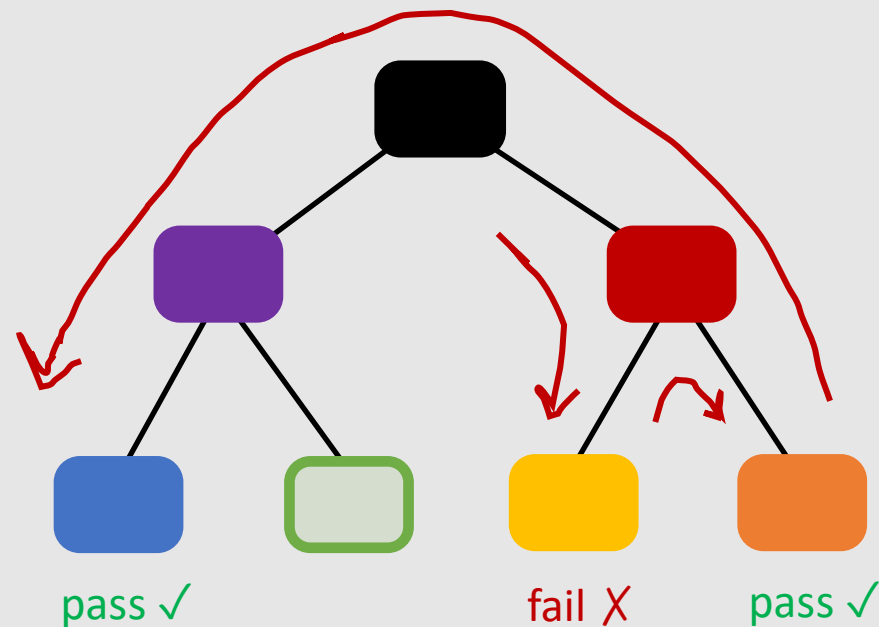
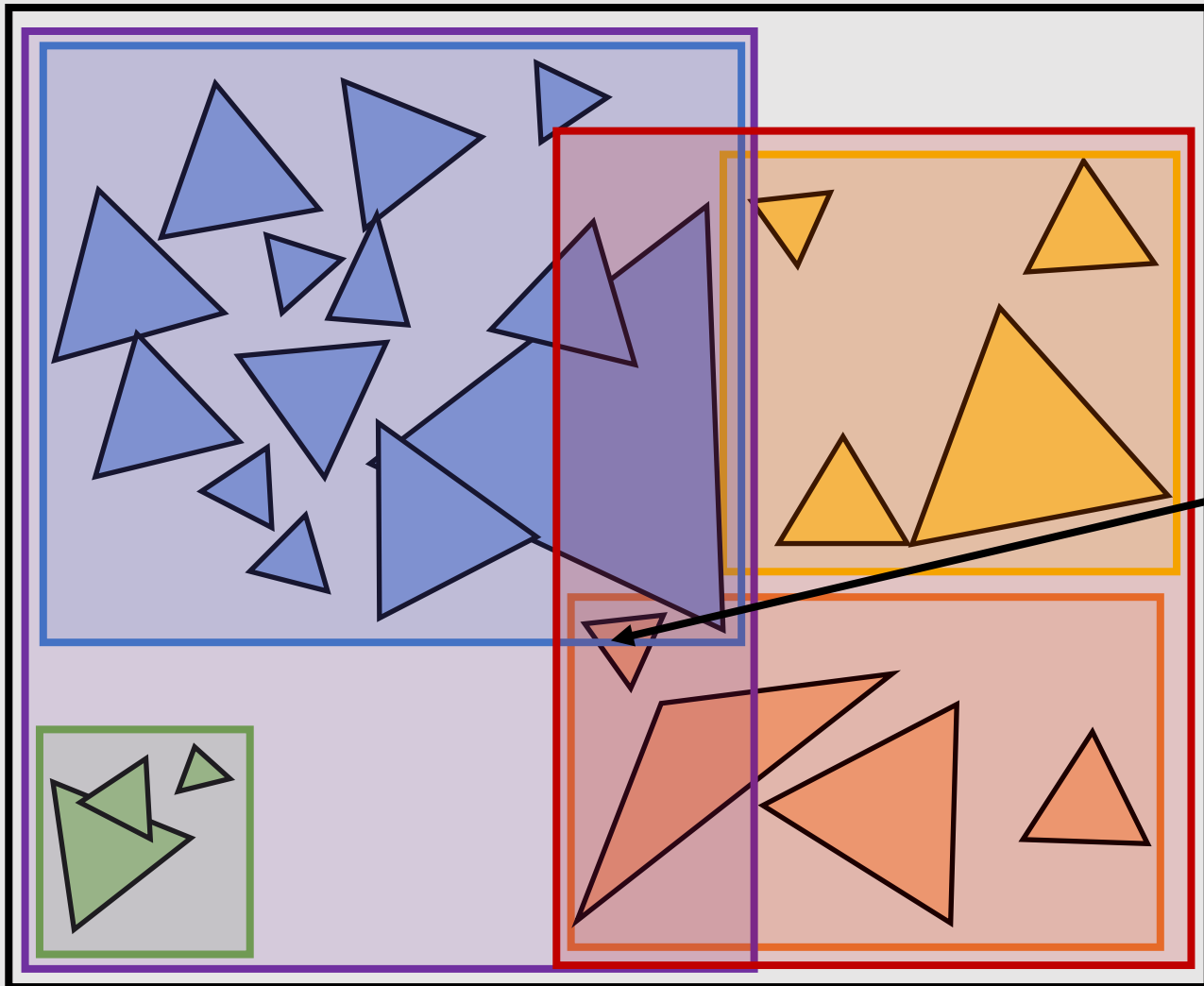
BVH Example



BVH Example



BVH Example



We can find a closer triangle if we check here
Remember: bounding boxes will intersect!

BVH Traversal

```
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;
}

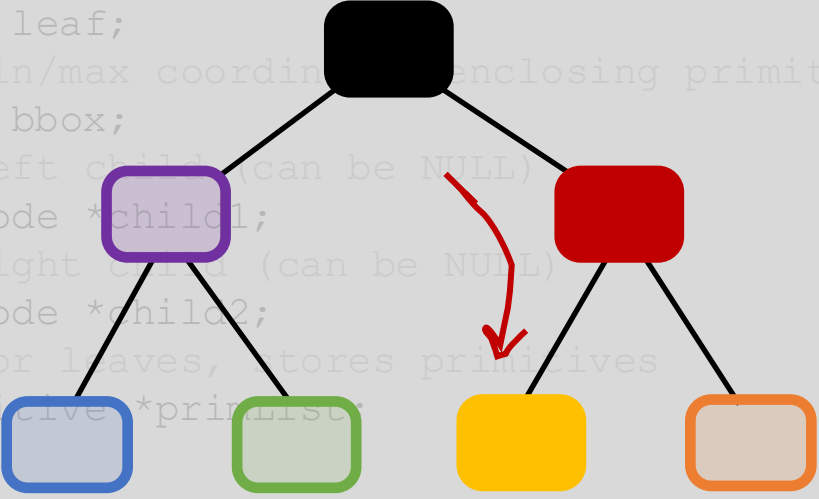
struct HitInfo {
    // the primitive the ray hit
    Primitive *prim;
    // the time along the ray the hit occurred
    float t;
}
```

```
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 = hit.t;
            }
        }
    } else {
        // traverse BOTH children
        hit(ray, node->child1, best);
        hit(ray, node->child2, best);
    }
}
```

BVH Traversal

```
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 *primitives;
}

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

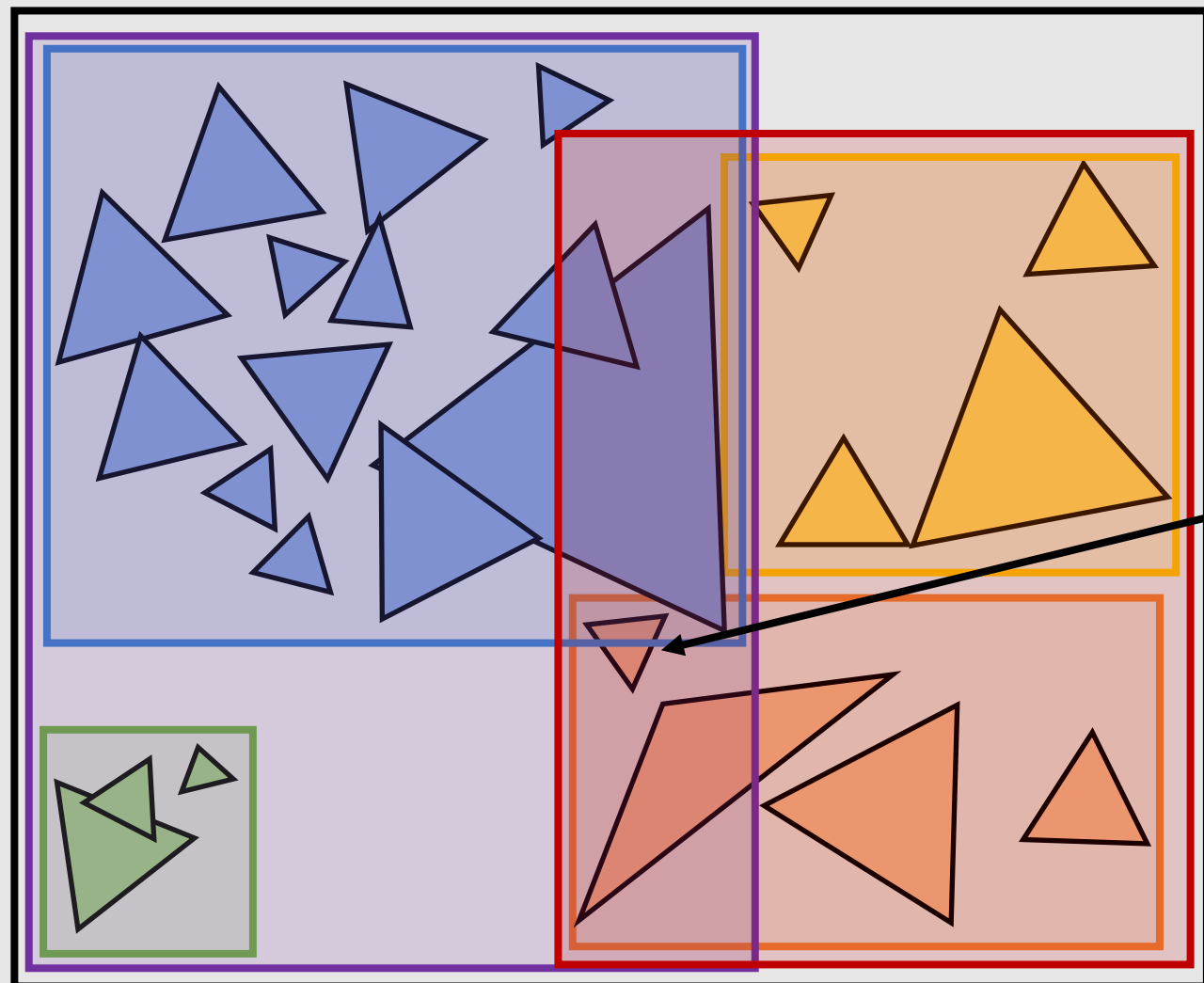


pass ✓

```
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 (Primitive* p : node->primitives) {
            if (hit.prim != NULL && hit.t < best.t) {
                best.prim = p;
                best.t = hit.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.**

Better BVH Traversal

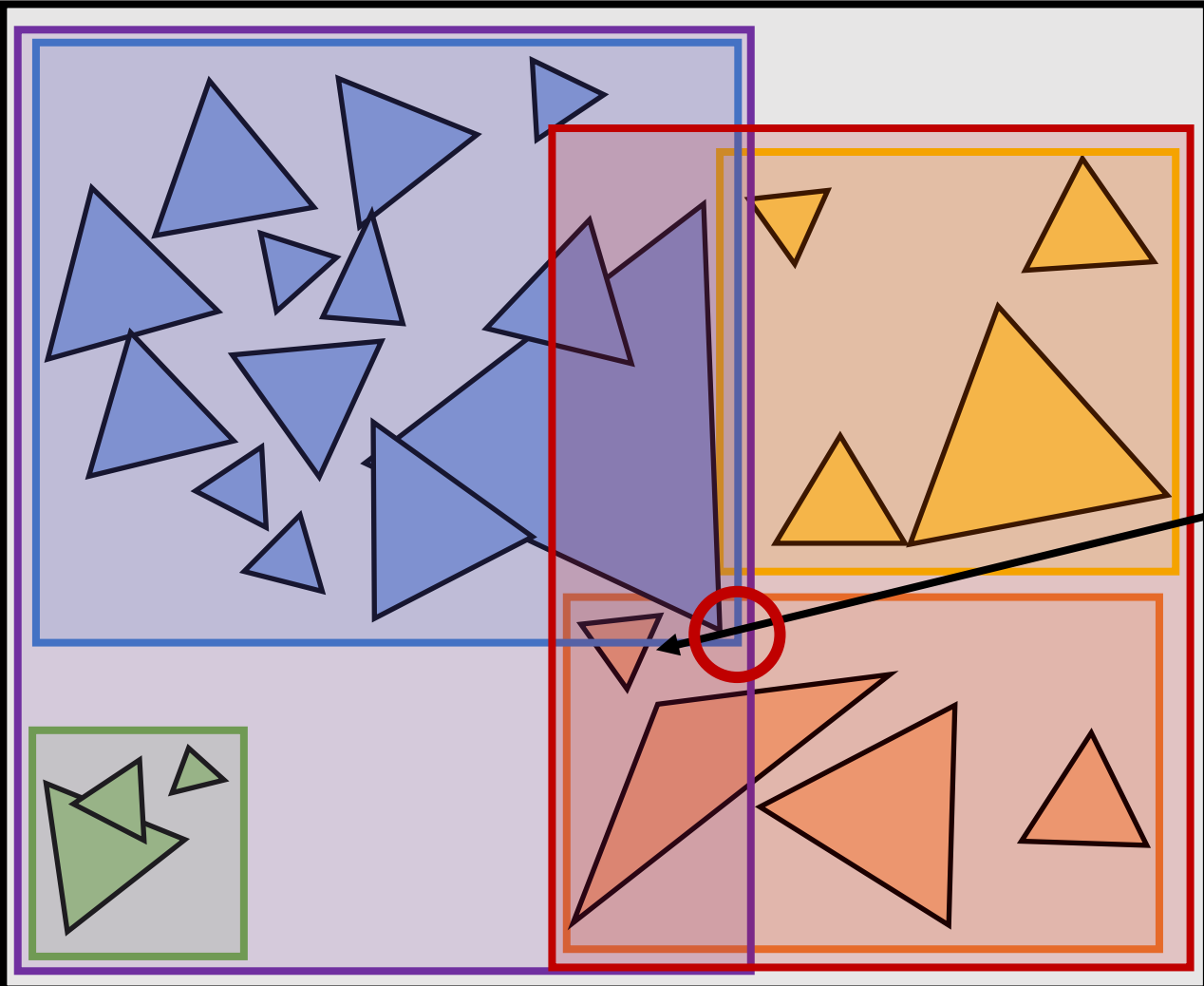


```
void hit(Ray* ray, BVHNode* node, HitInfo* best)
{
    if (node->leaf) {
        // same as previous slide
    } else {
        BVHNode* child1 = node->child1;
        BVHNode* child2 = node->child2;

        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);
    }
}
```

Better BVH Traversal



```
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;

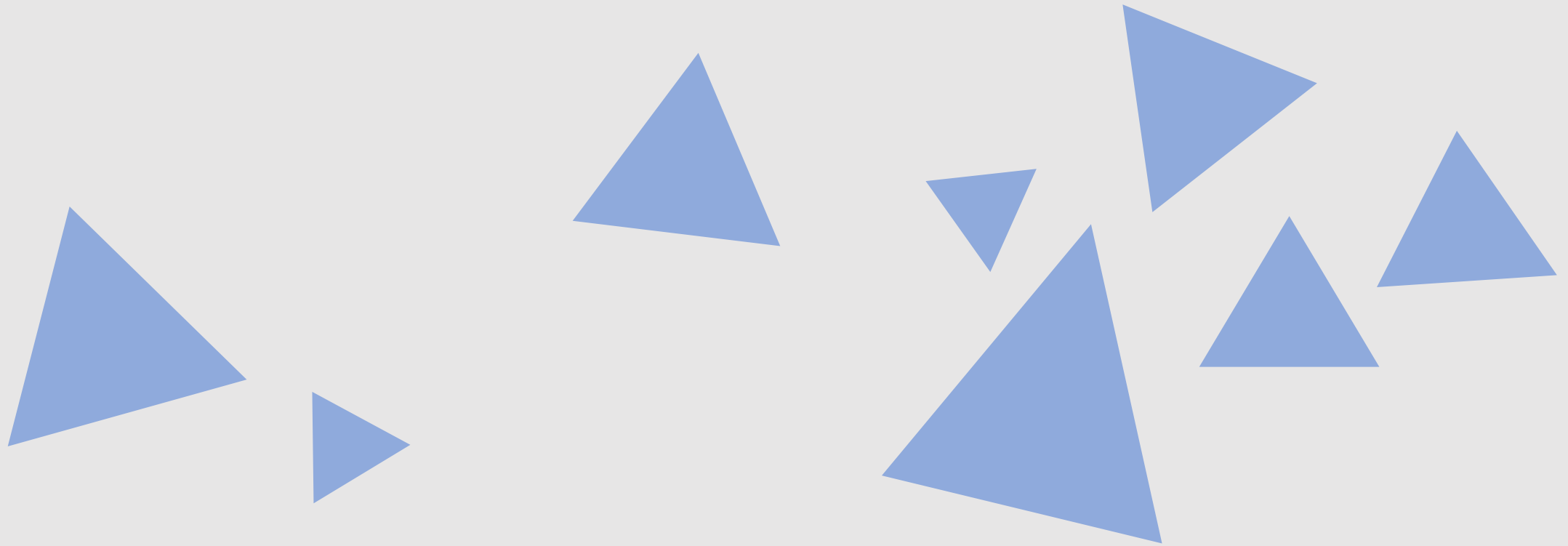
        hit(ray, first, best);
        if (hit2.t < closest.t)
            hit(ray, second, best);
    }
}
```

Only check far bbox if closest primitive in the near bbox is farther than the closest point intersected in the far bbox.

This means there's a potential to find a better primitive :)

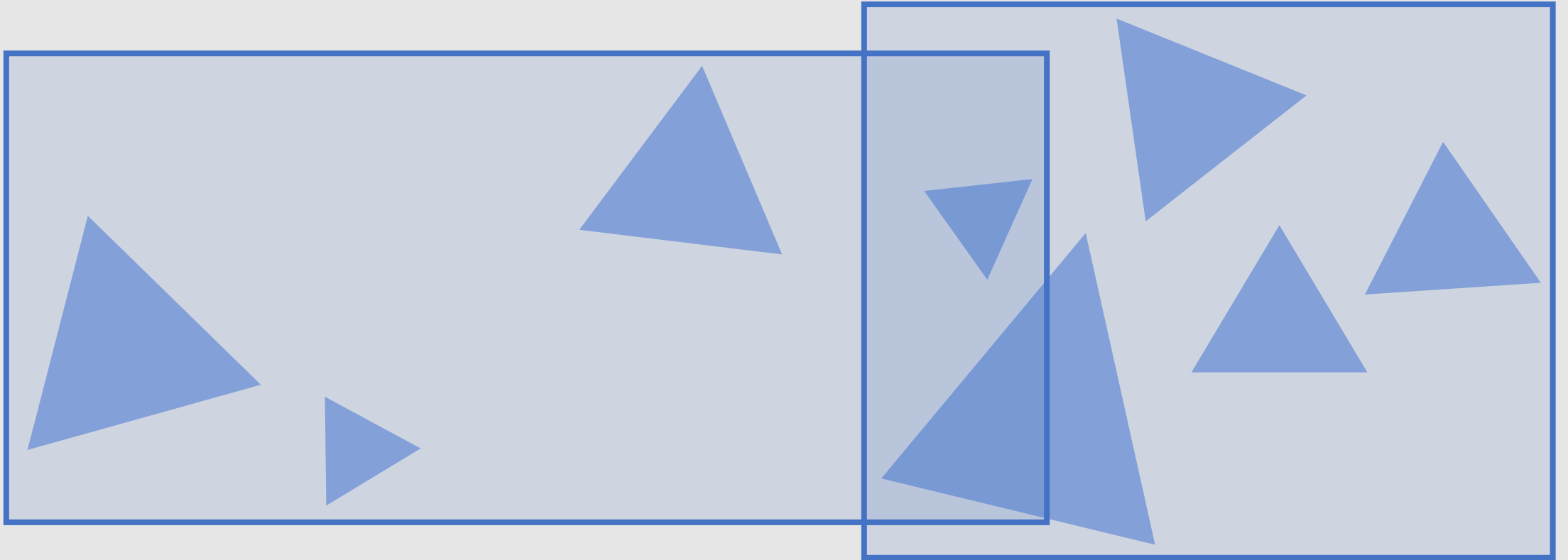
So we know how to traverse a BVH,
But how do we build one?

BVH Partitioning



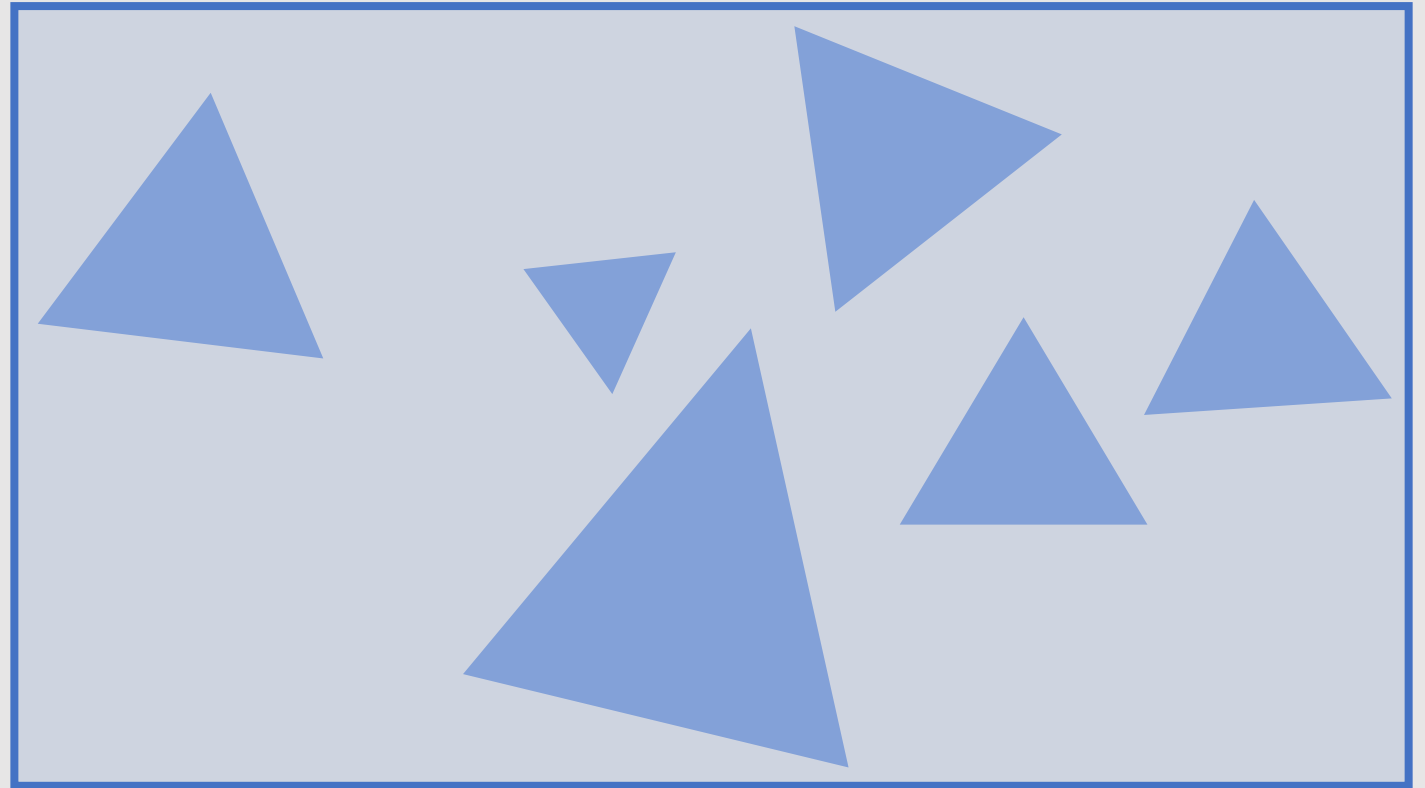
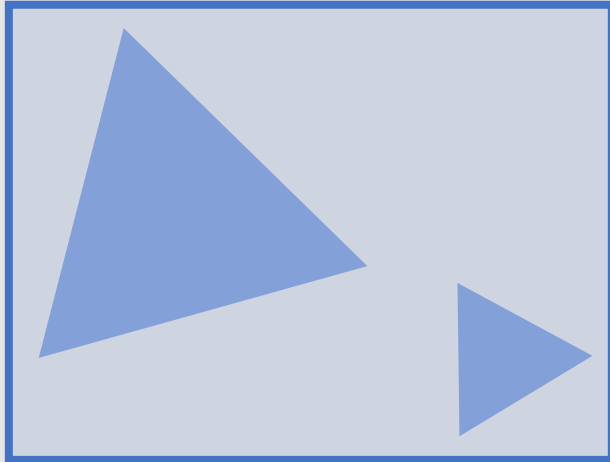
What is the best way to partition these primitives?

BVH Partitioning



We can split them into equal # of primitives...
...but bboxes take up large area

BVH Partitioning



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:

$$C = C_{trav} + p_A C_A + p_B C_B$$

- 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

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

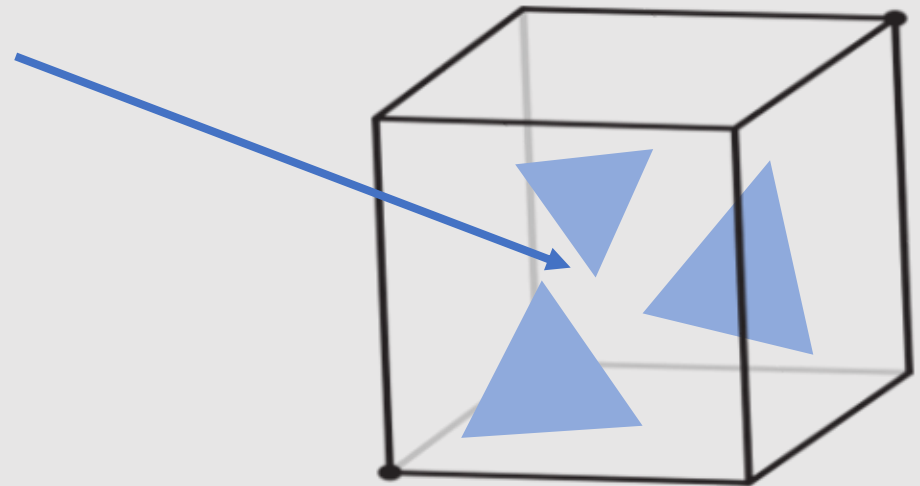
- The cost of intersecting a node is:

$$C = C_{trav} + p_A C_A + p_B C_B$$

- 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:

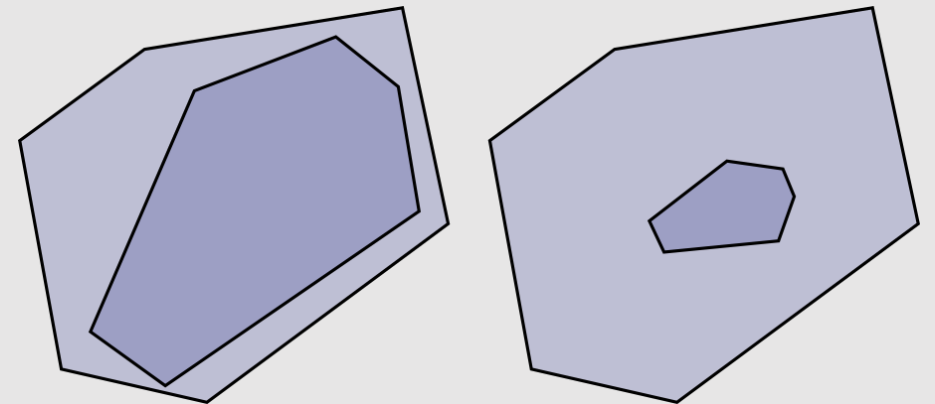
$$C = C_{trav} + p_A C_A + p_B C_B$$

- 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

- 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:

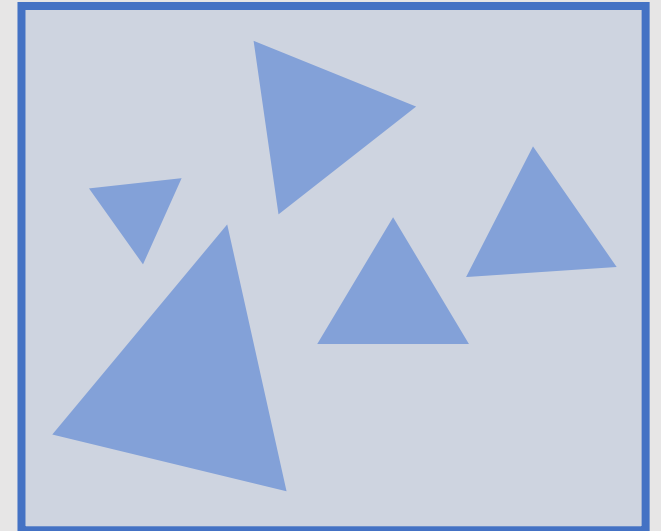
$$C = C_{trav} + p_A C_A + p_B C_B$$

- 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

- For a node C_A , this is the cost of checking all primitives held by this box

- All triangles have the same cost C_{tri}
- For N_A triangles, cost is $N_A C_{tri}$



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

- New equation:

$$C = C_{trav} + \frac{S_A}{S_C} N_A C_{tri} + \frac{S_B}{S_C} N_B C_{tri}$$

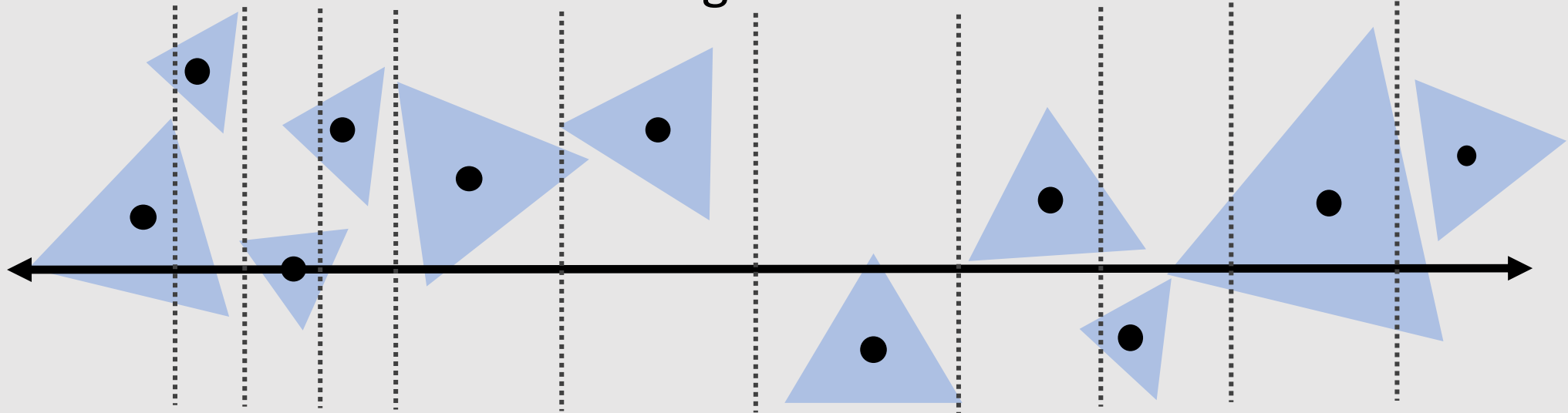
- C_{trav} , C_{tri} and S_C are constants, so we can remove them when computing the minimum cost:

$$C' = S_A N_A + S_B N_B$$

- Minimizes surface area deviation
- Minimizes primitive deviation

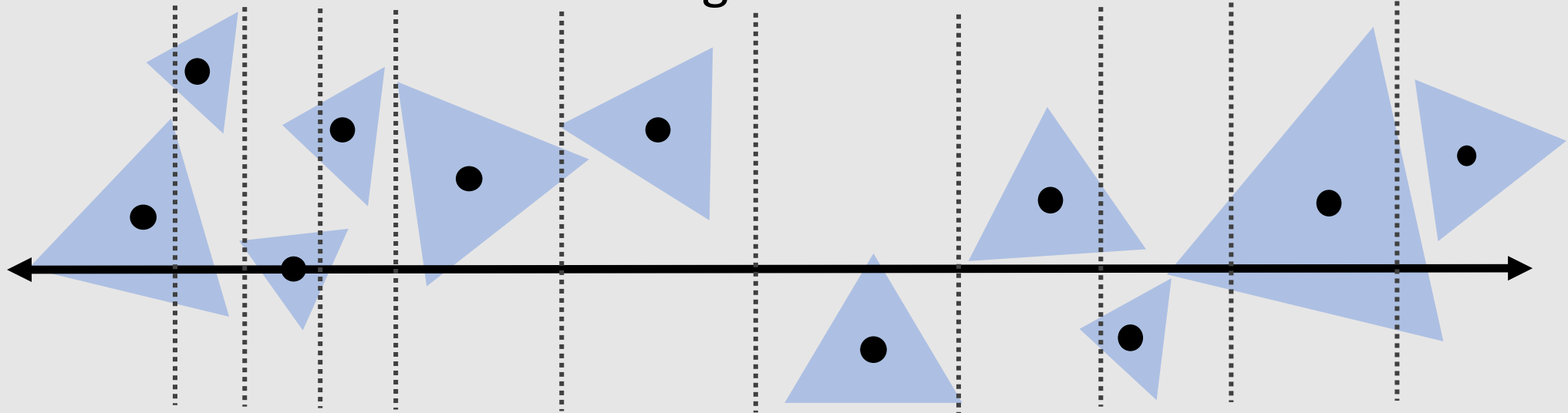
We know what a good partition is,
but how do we actually build a partition

Building Partitions



```
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);
```

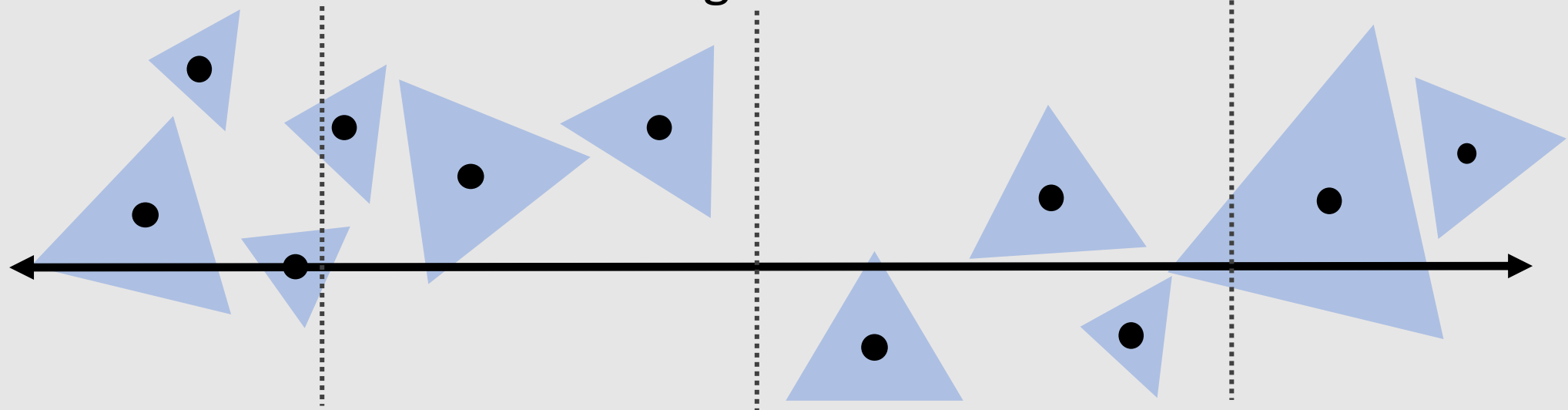
Building Partitions



```
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!

Building Partitions

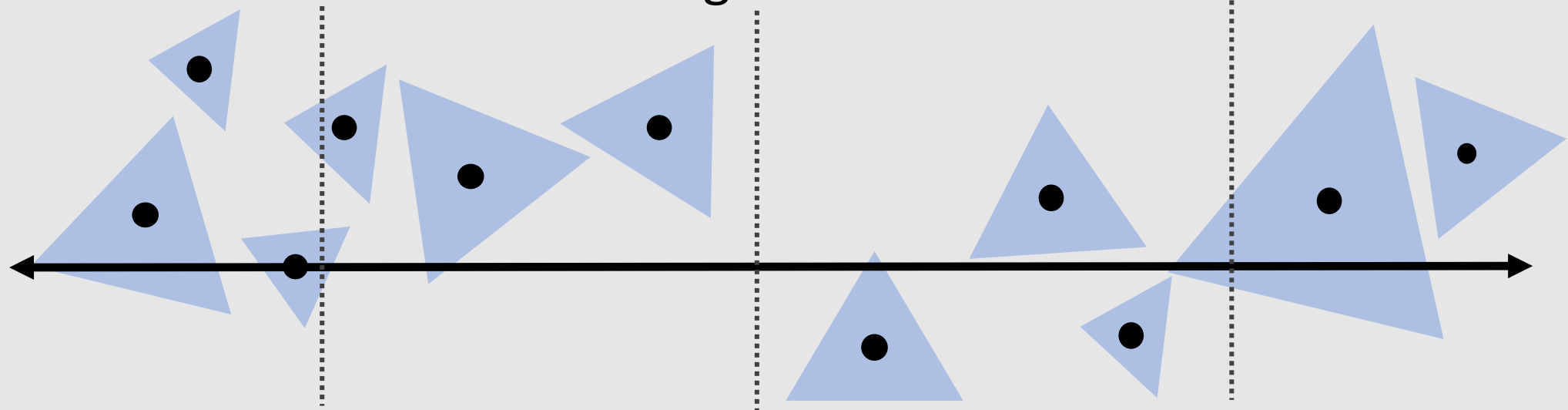


```
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    for(int i = 0; i < n; i+=32) { // check every B primitives (B = 32)
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);

        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);
```

Building Partitions



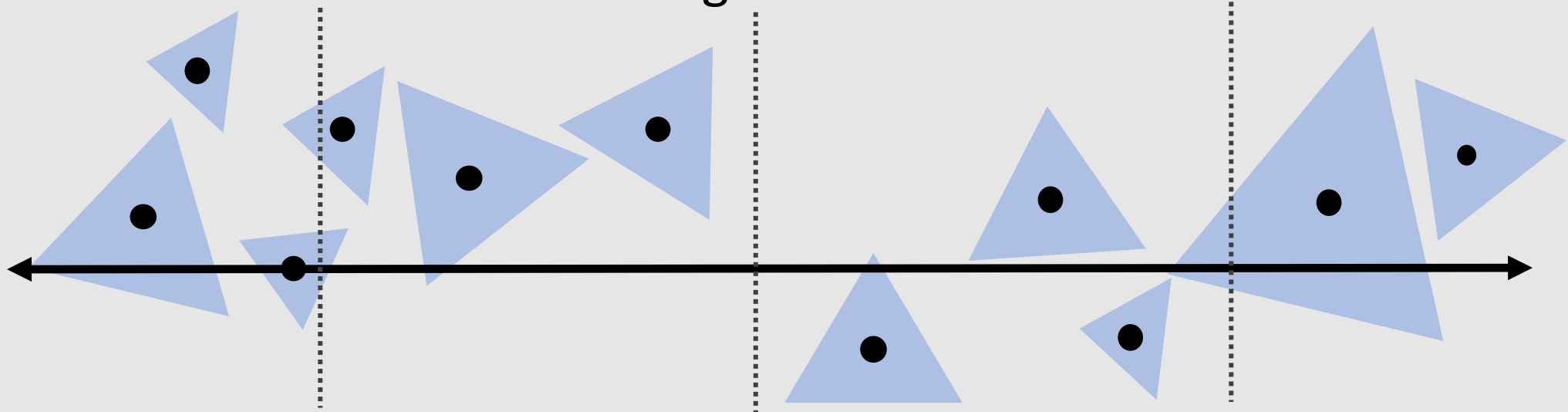
```
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    for(int i = 0; i < n; i+=32) { // check every B primitives (B = 32)
        a = bbox(primitives[0,i]);
        b = bbox(primitives[i,n]);

        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);
```

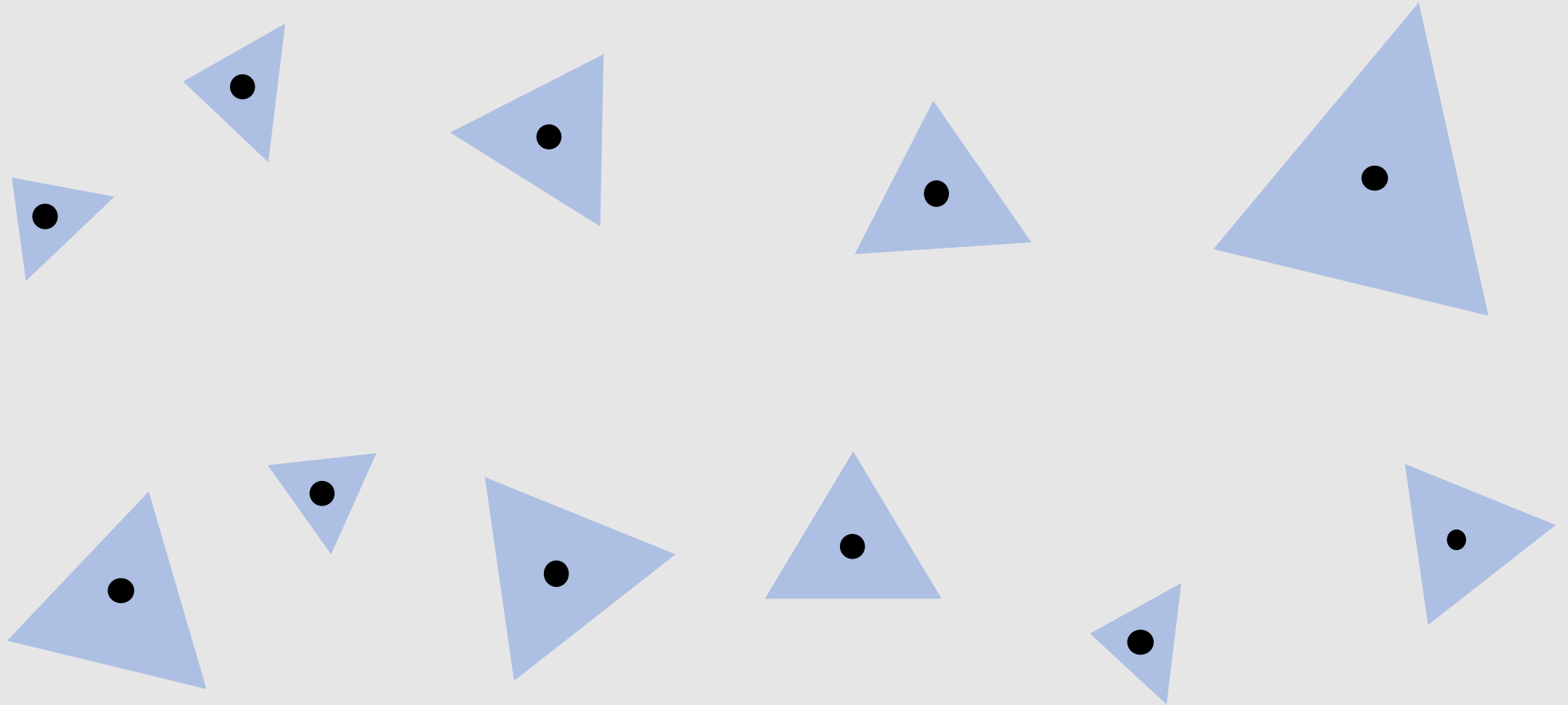
Still a lot of iterating over primitives each loop!

Building Partitions

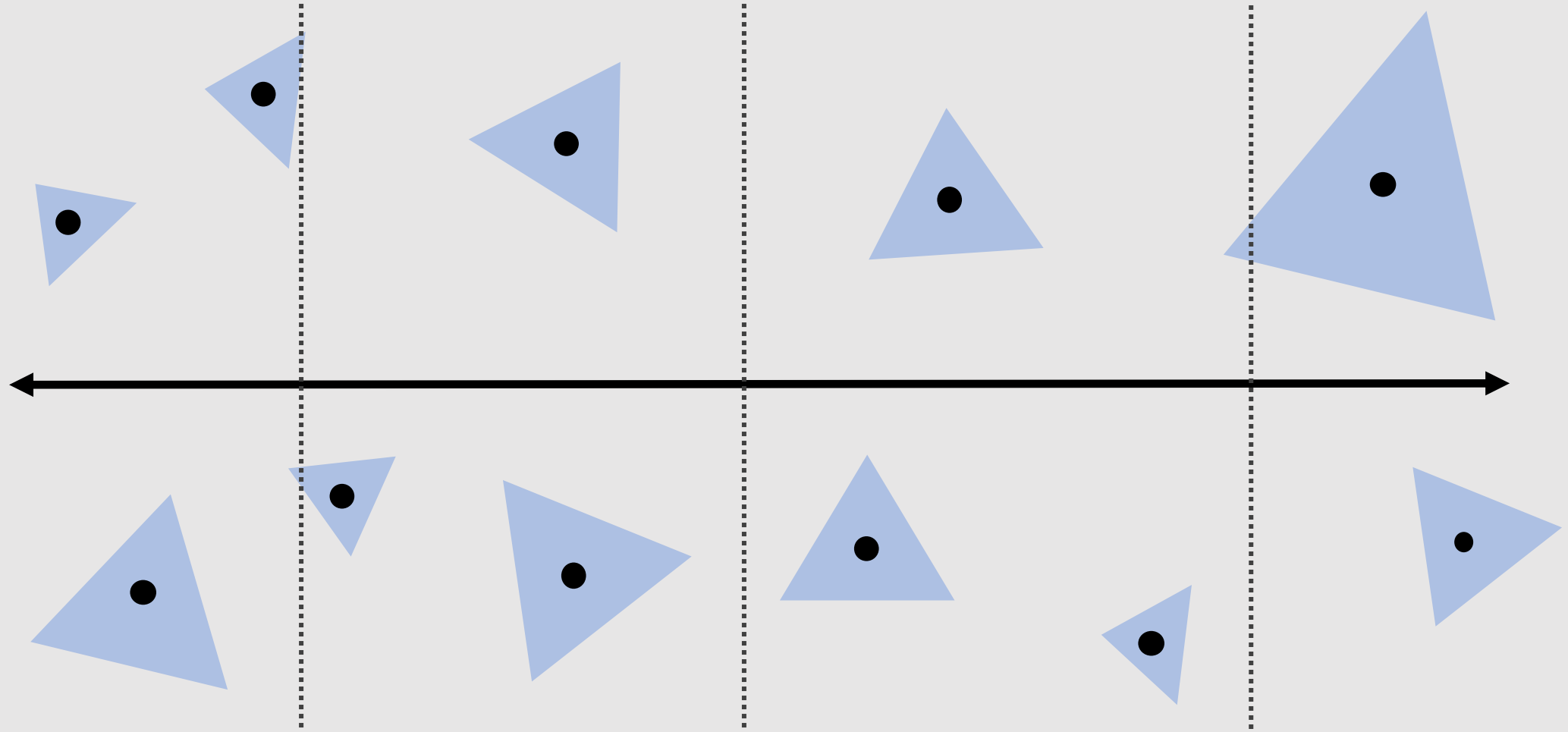


```
for(axis : [x, y, z]) {
    sort(primitives, axis);
    n = primitives.length();
    bin_n = bin.length();
    for(int i = 0; i < n; i++) {
        bin = compute_bucket(primitives[i].centroid) // find bin that triangle lies in
        bin.bbox.add(primitives[i]); // add triangle to bin
    }
    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

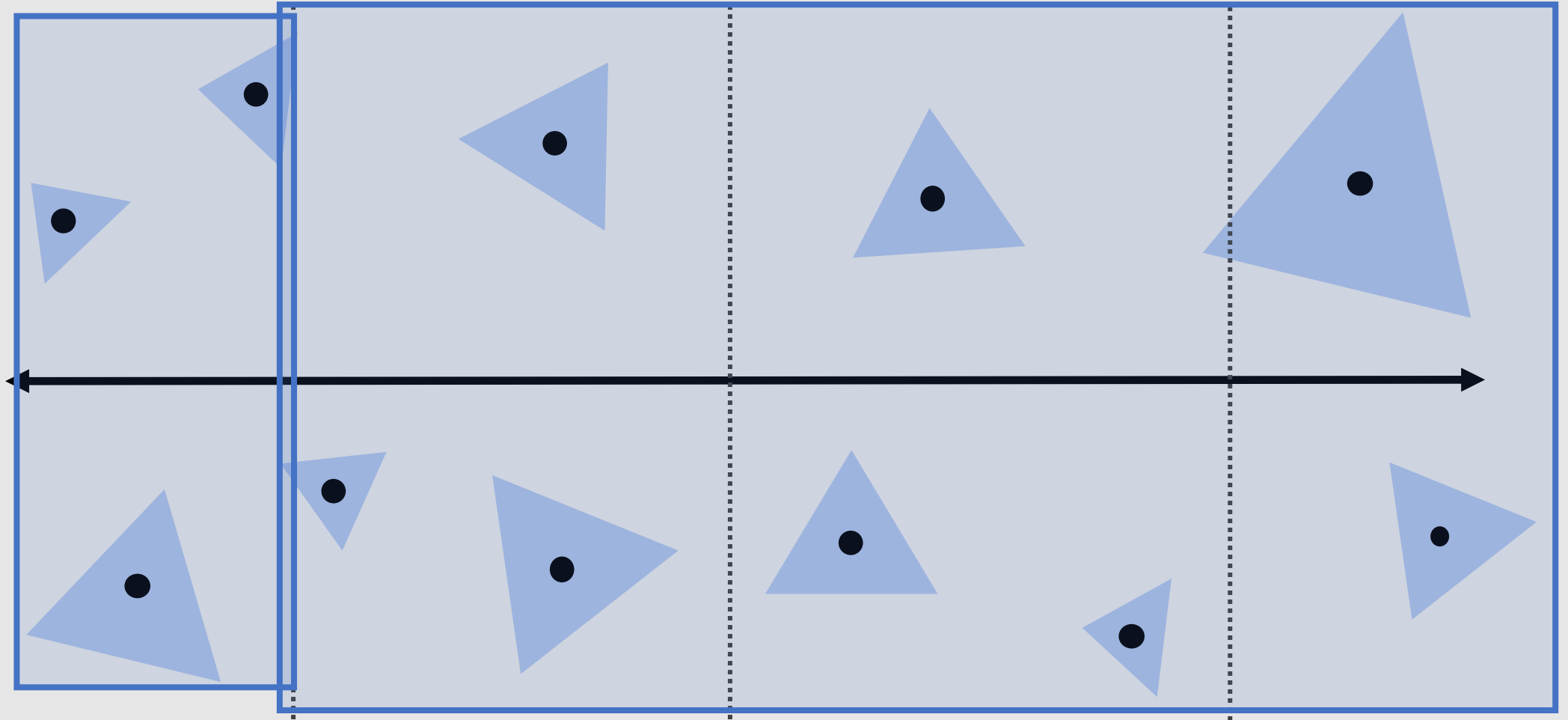


Building Partitions Example



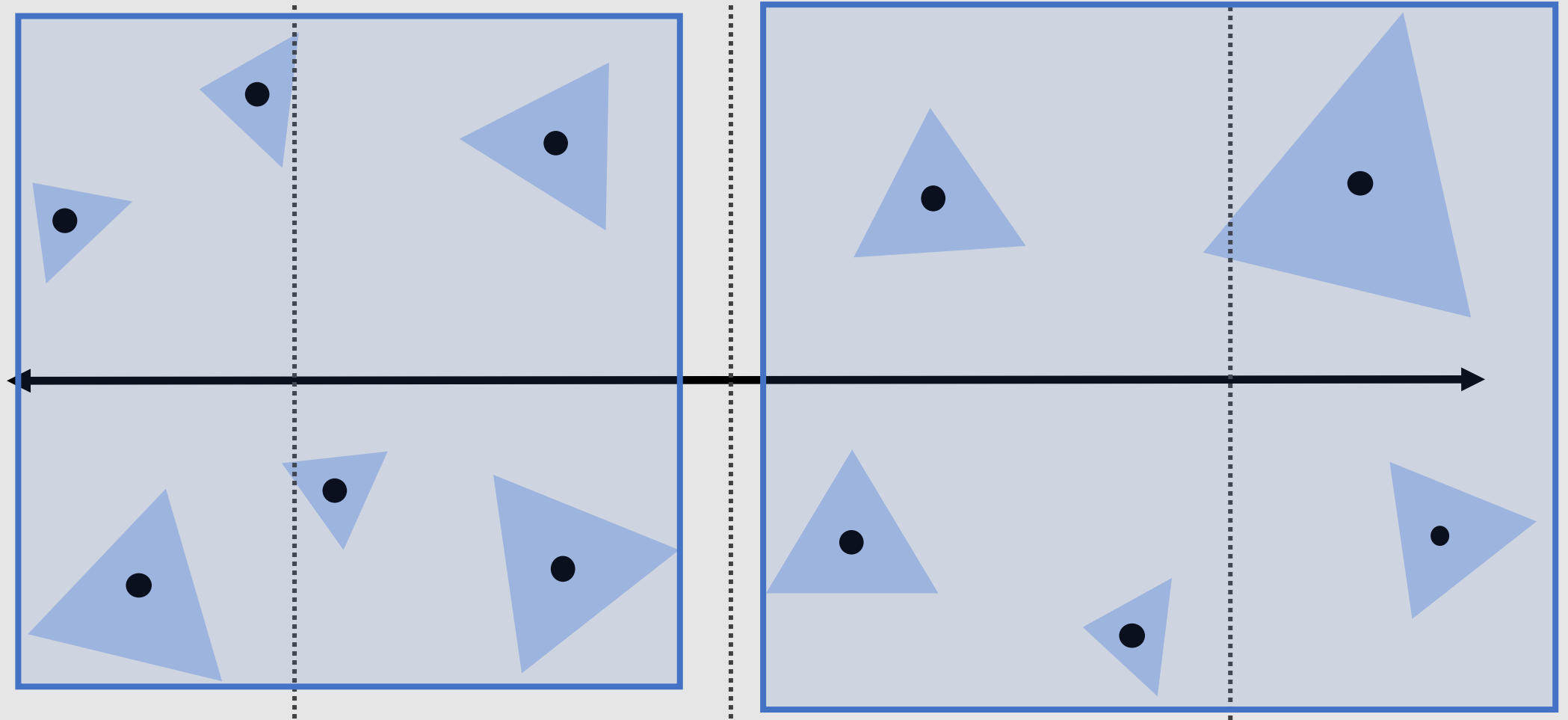
[x-axis binning]

Building Partitions Example



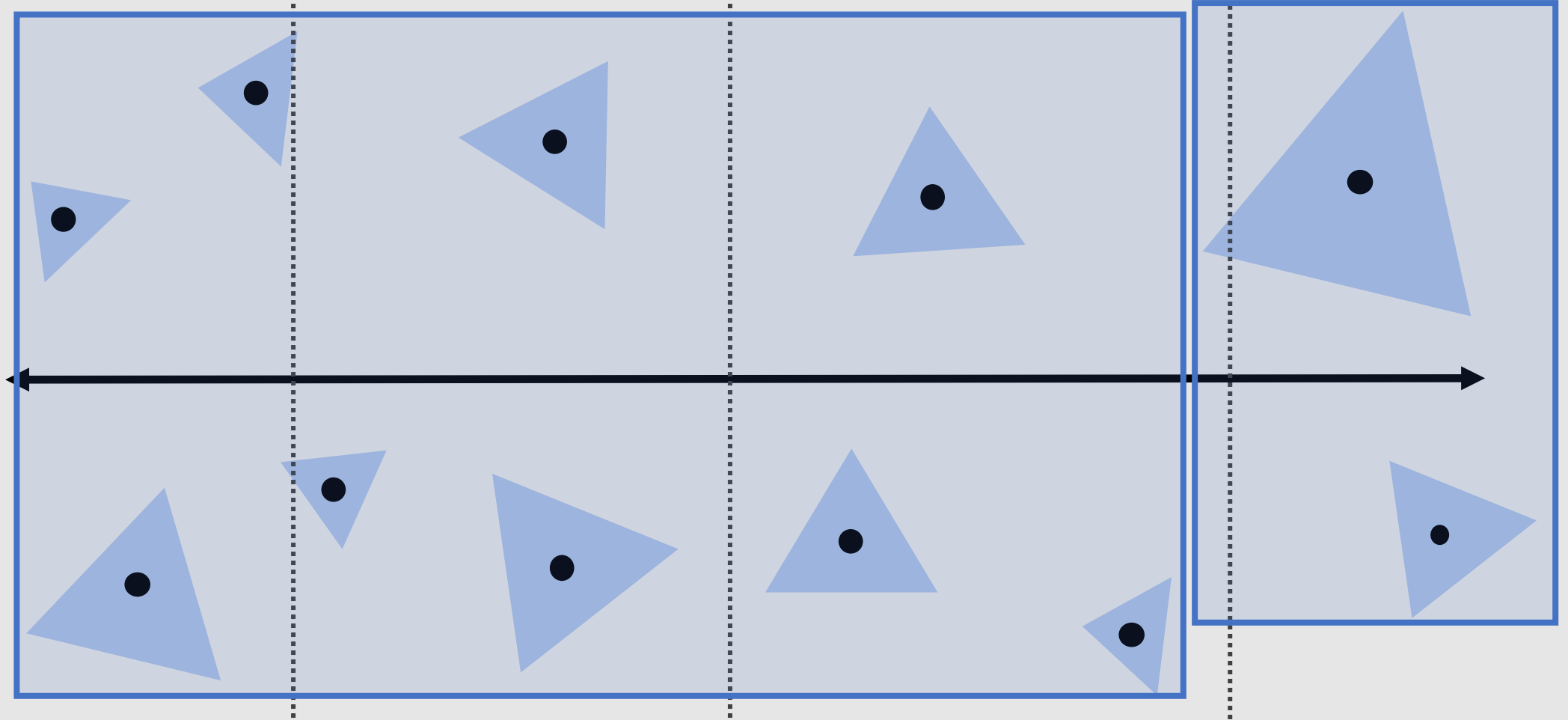
$$\text{Cost} = 3 \text{ prims} * (0.15) + 8 \text{ prims} * (0.87)$$

Building Partitions Example



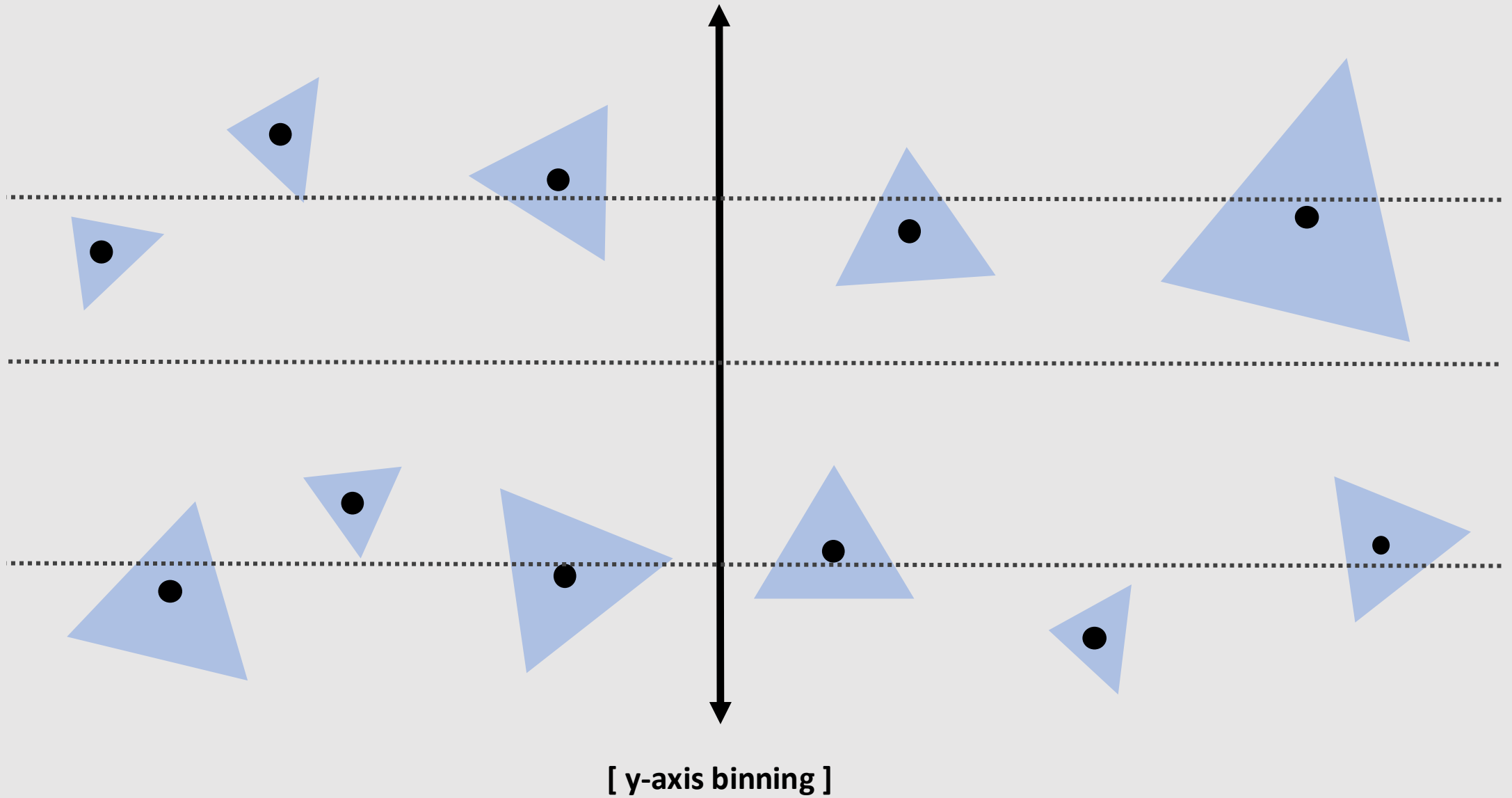
$$\text{Cost} = 6 \text{ prims} * (0.38) + 5 \text{ prims} * (0.43)$$

Building Partitions Example

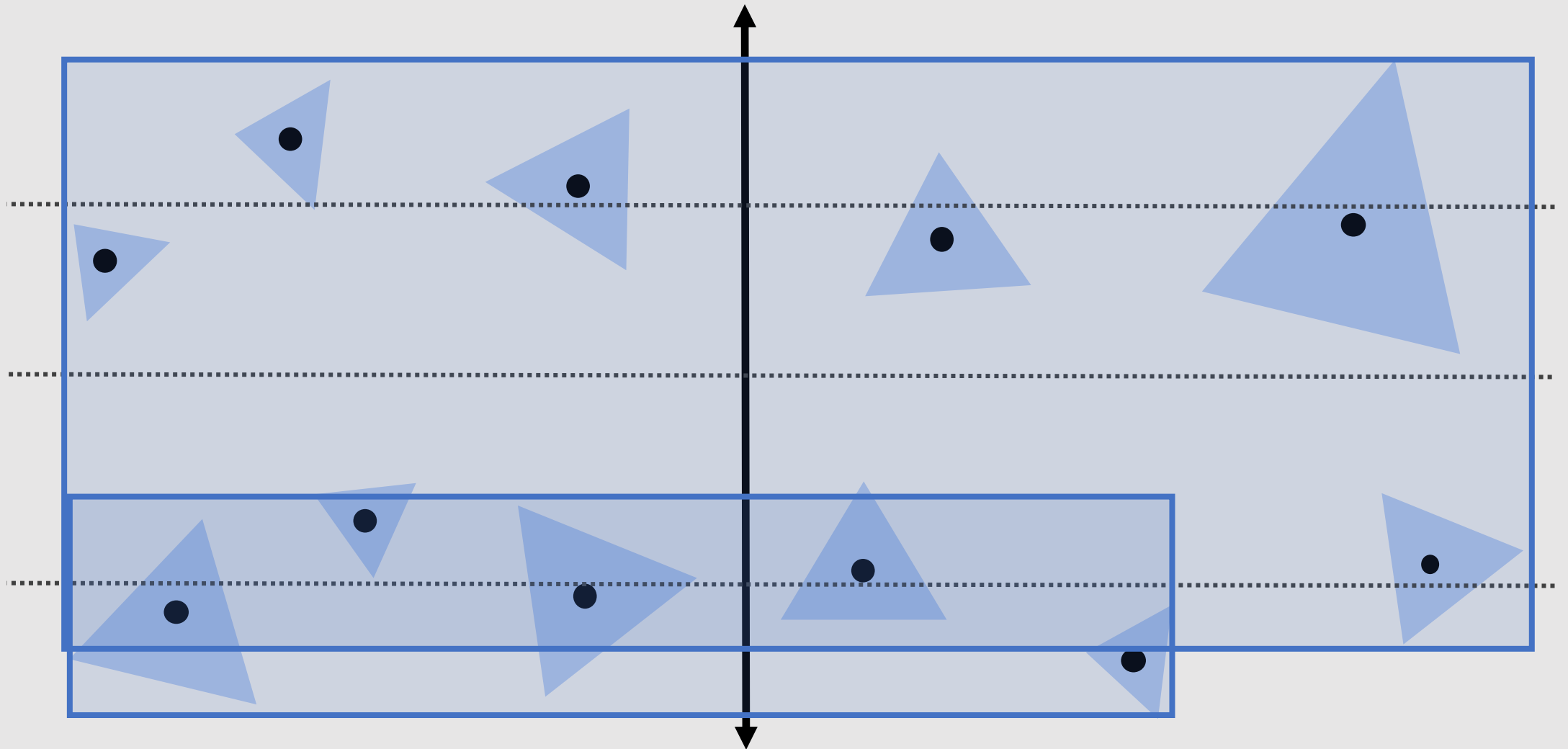


$$\text{Cost} = 9 \text{ prims} * (0.81) + 2 \text{ prims} * (0.18)$$

Building Partitions Example

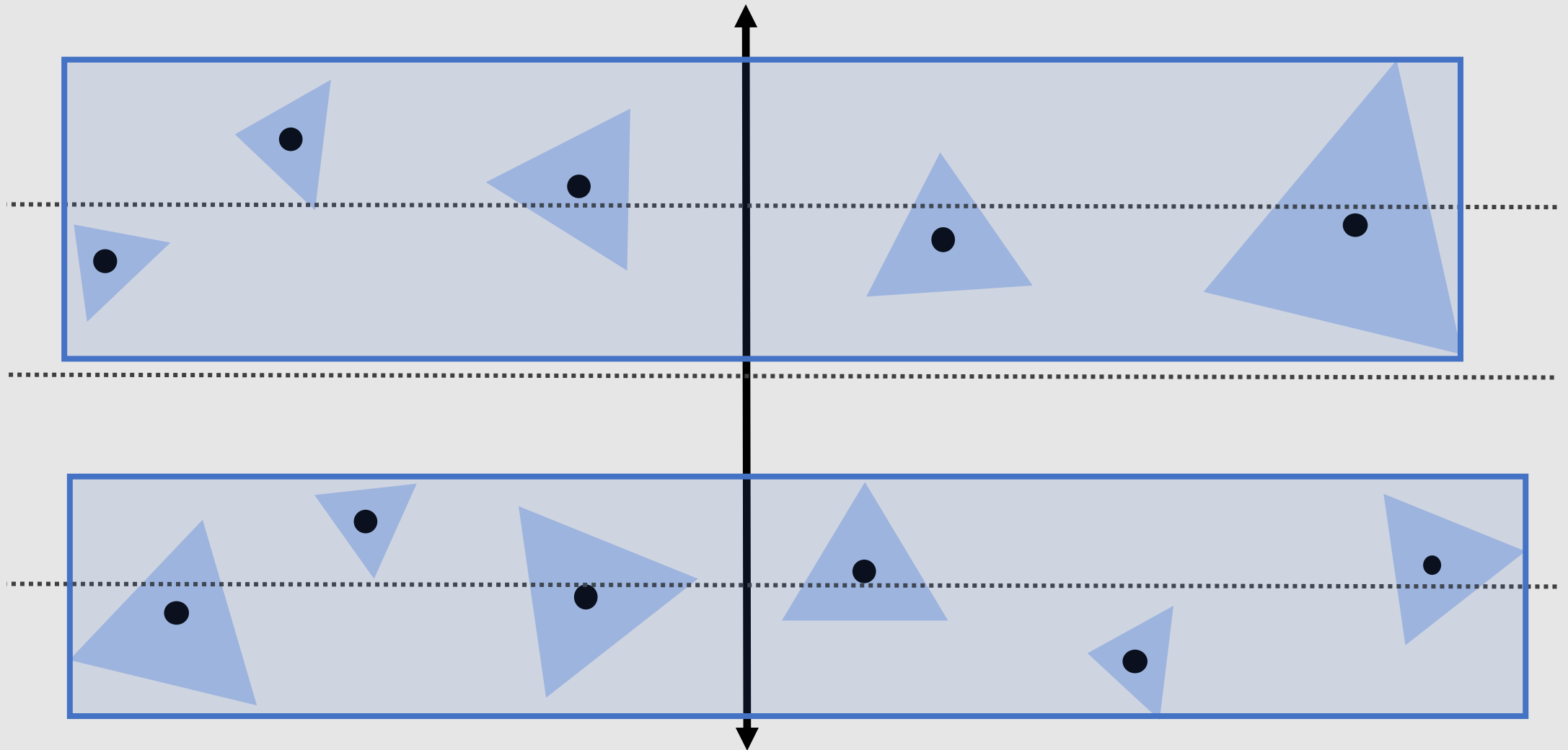


Building Partitions Example



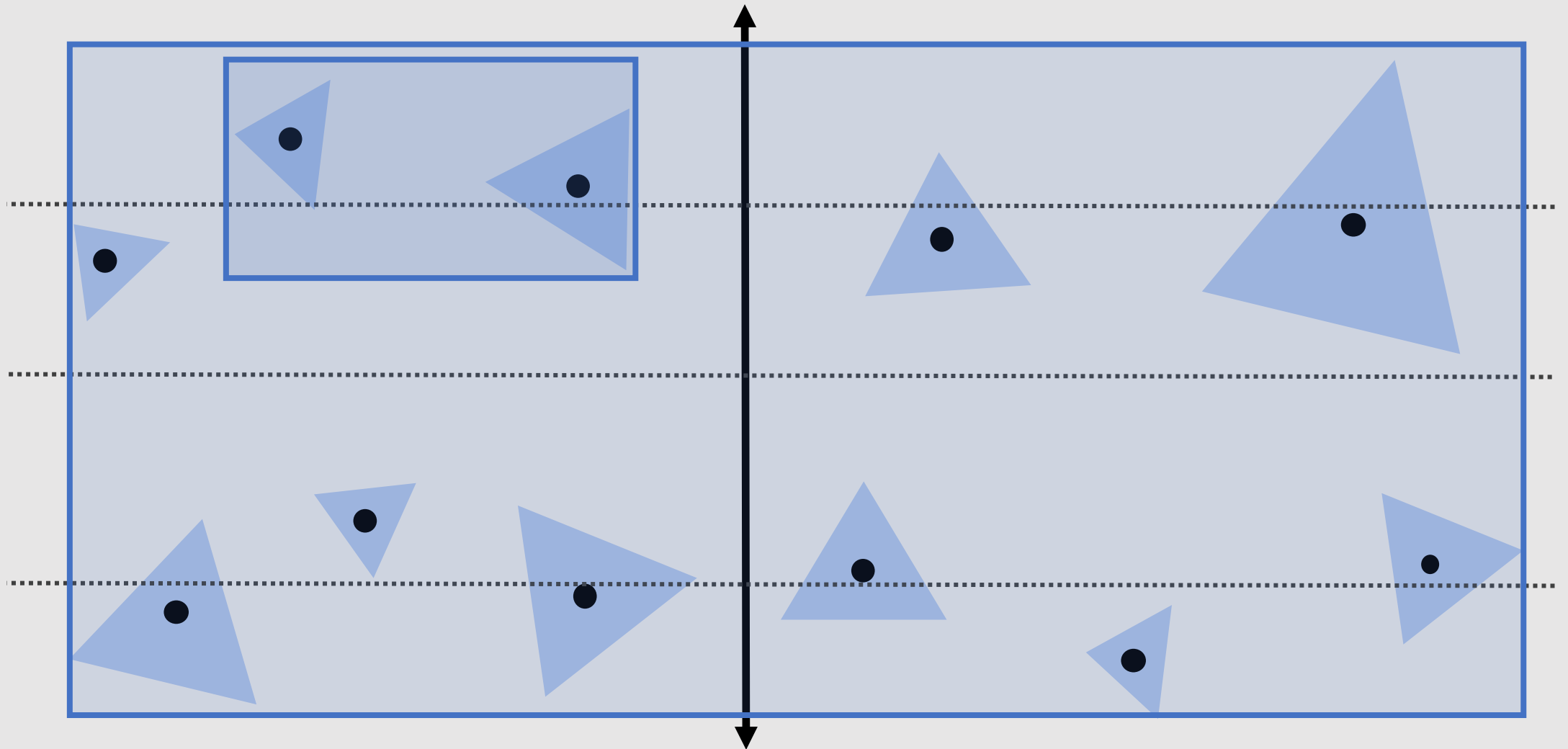
$$\text{Cost} = 3 \text{ prims} * (0.19) + 8 \text{ prims} * (0.91)$$

Building Partitions Example



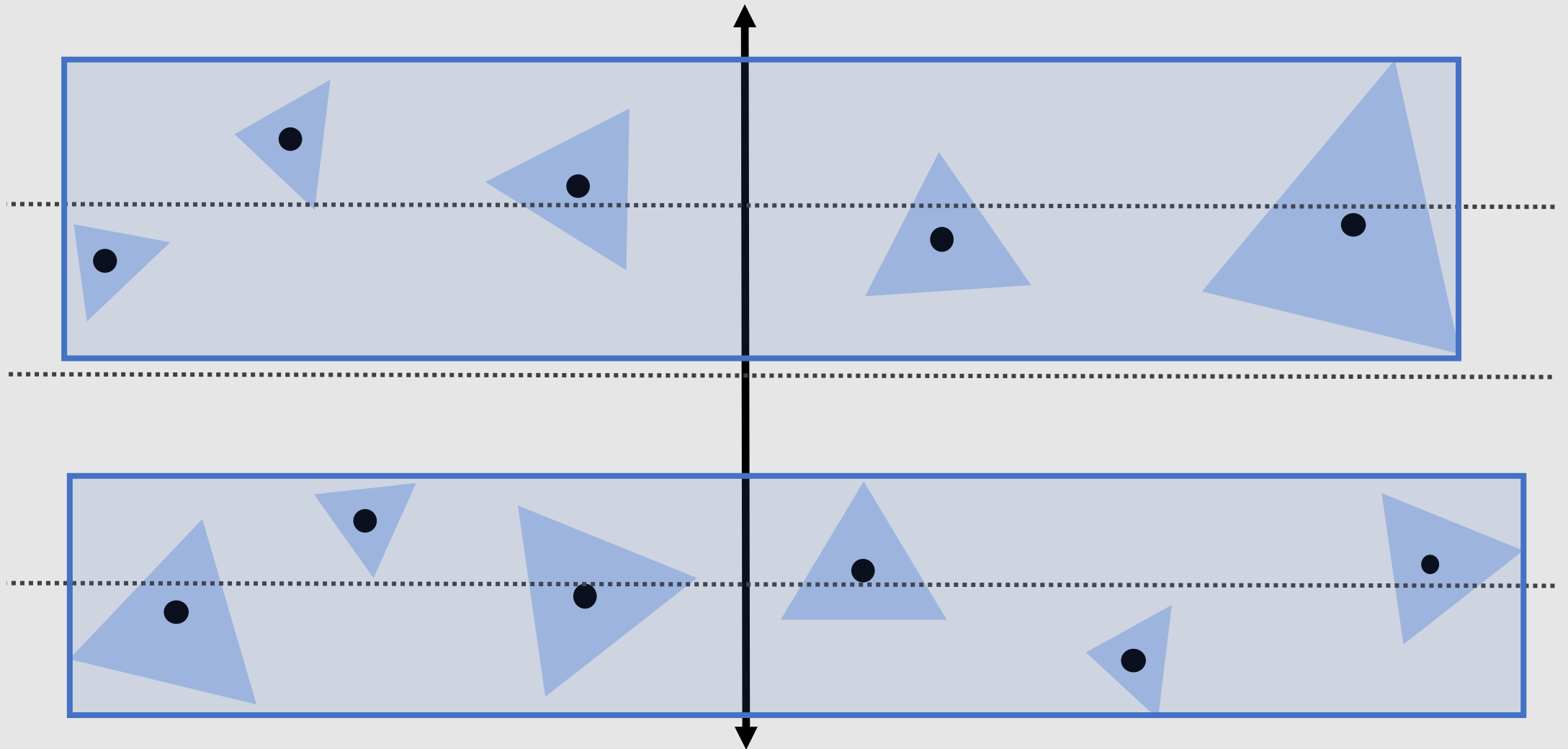
$$\text{Cost} = 6 \text{ prims} * (0.32) + 5 \text{ prims} * (0.36)$$

Building Partitions Example



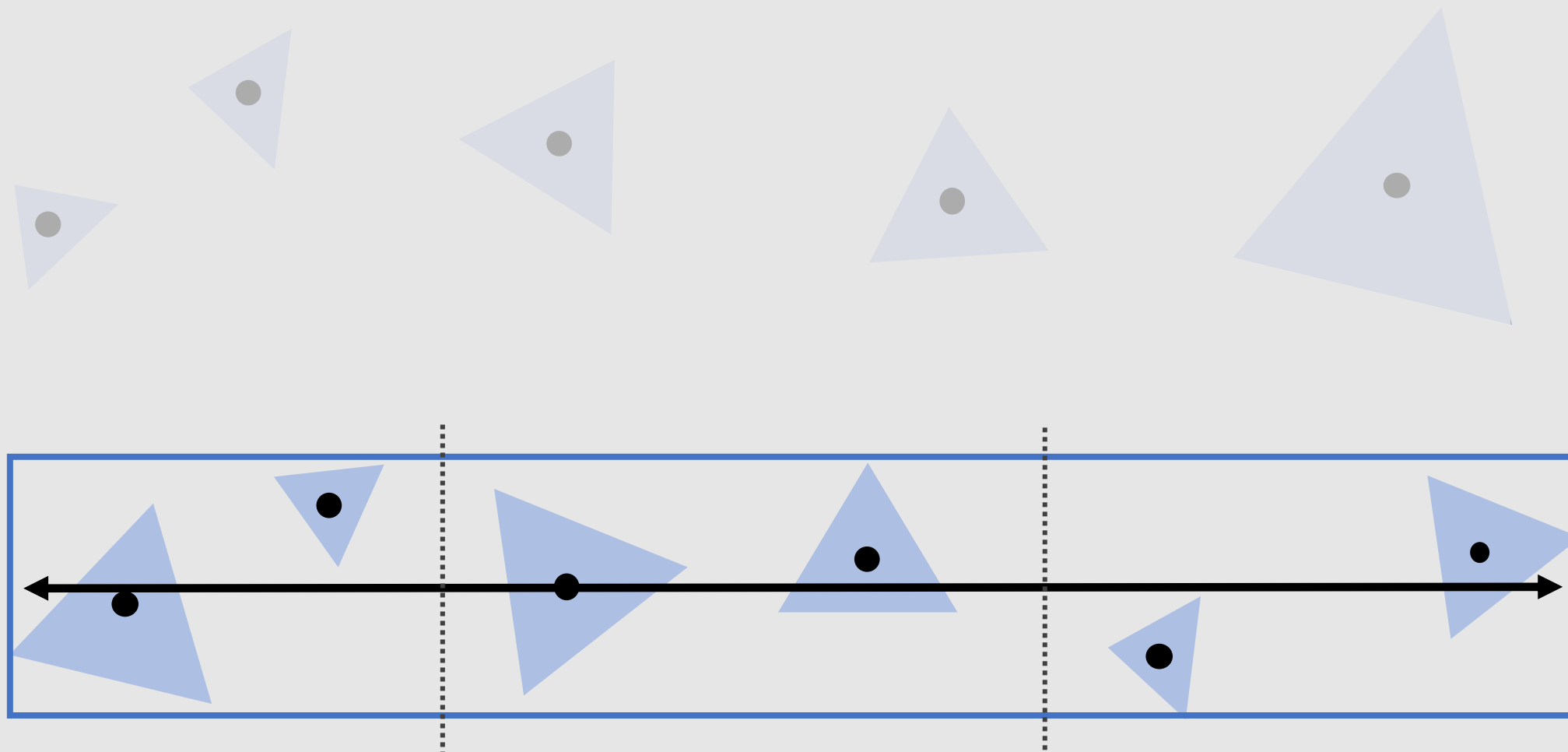
$$\text{Cost} = 9 \text{ prims} * (0.94) + 2 \text{ prims} * (0.13)$$

Building Partitions Example



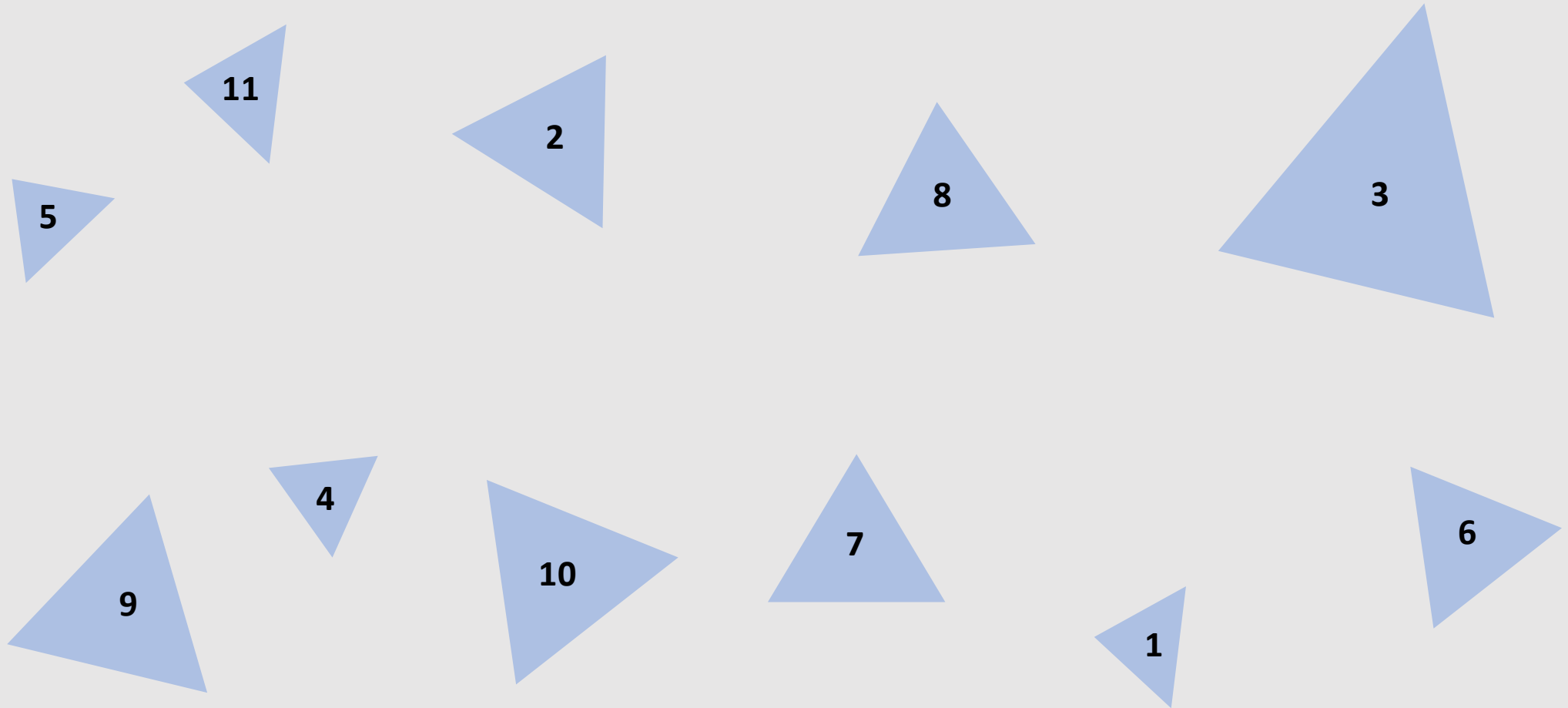
Best Partition

Building Partitions Example



Recurse with each child node

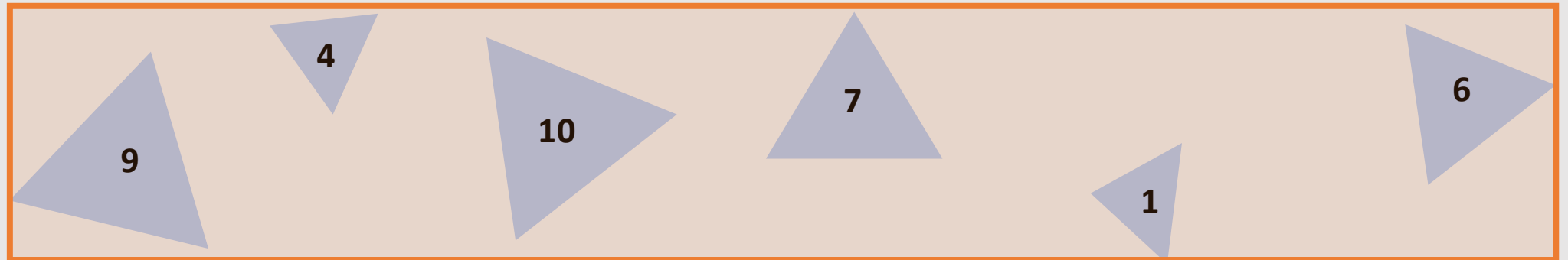
What About Ordering?



primitives

1	2	3	4	5	6	7	8	9	10	11
---	---	---	---	---	---	---	---	---	----	----

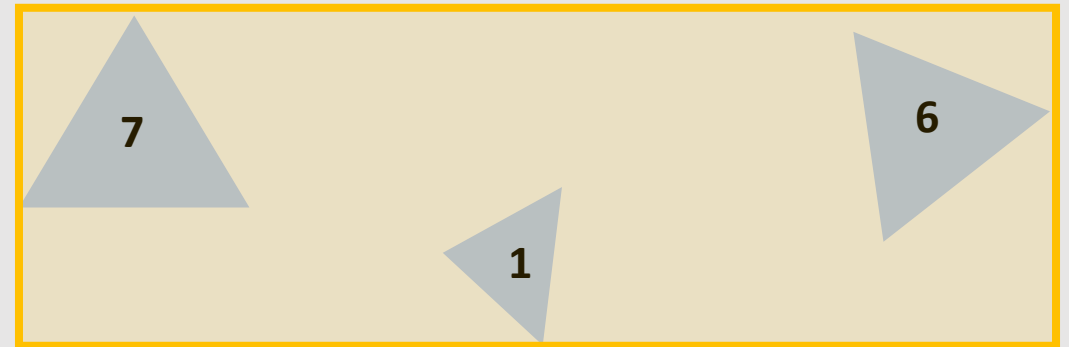
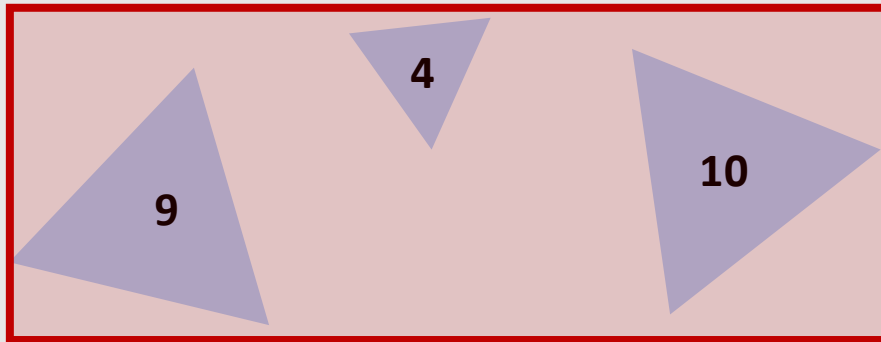
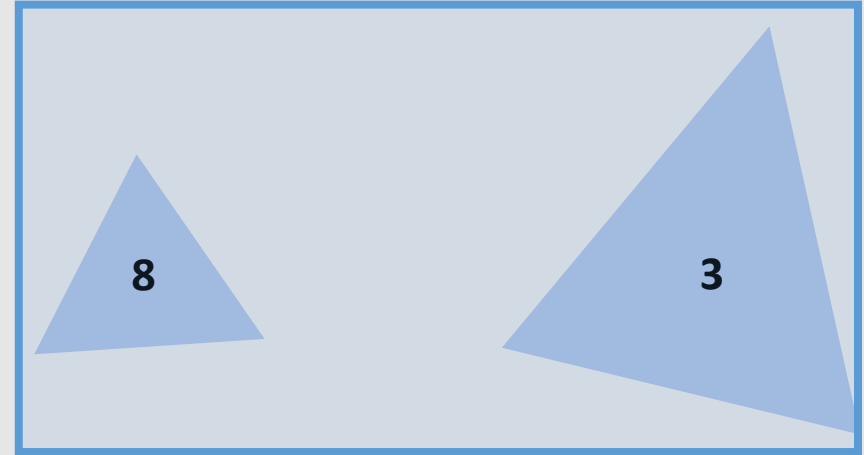
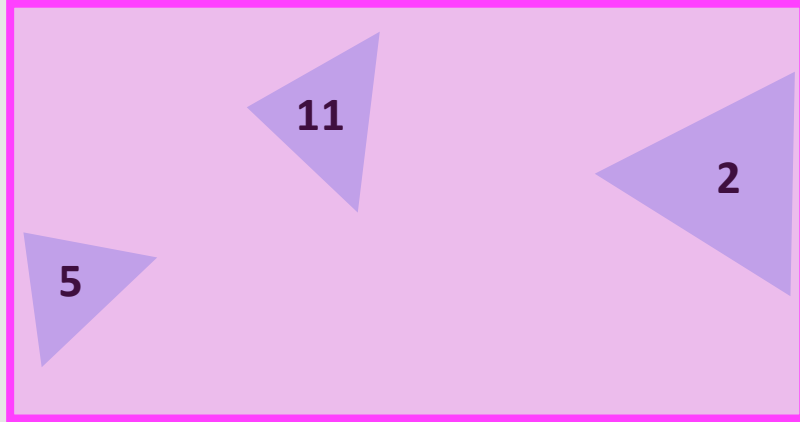
What About Ordering?



primitives

1	9	10	7	6	4	5	3	8	2	11
---	---	----	---	---	---	---	---	---	---	----

What About Ordering?

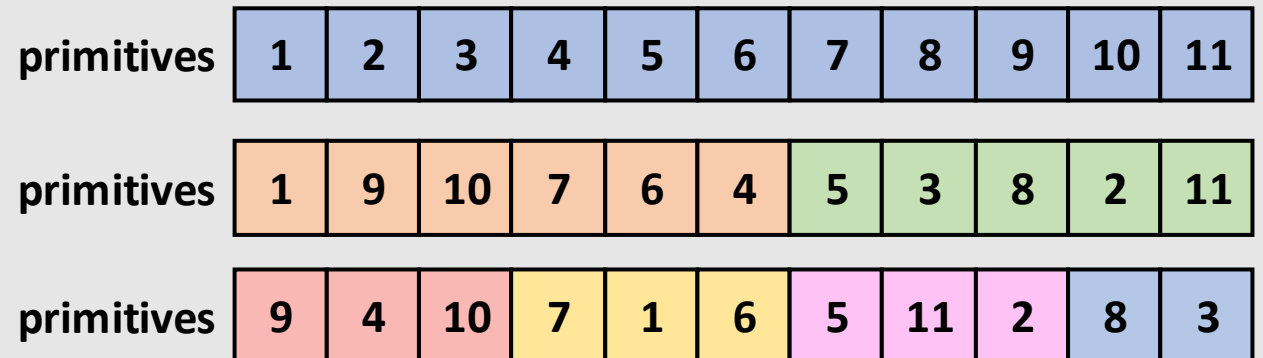


primitives

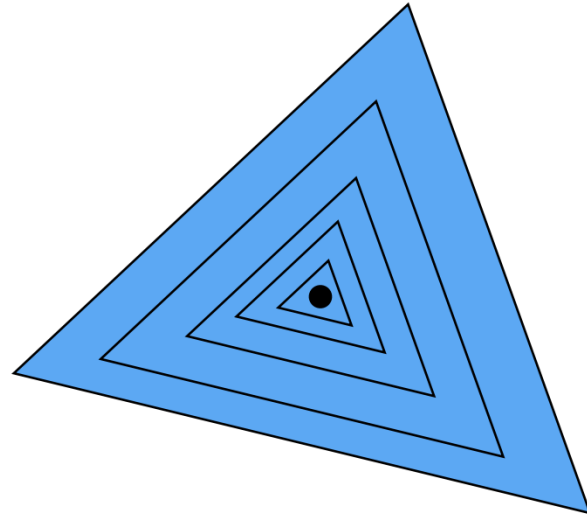
9	4	10	7	1	6	5	11	2	8	3
---	---	----	---	---	---	---	----	---	---	---

What About Ordering?

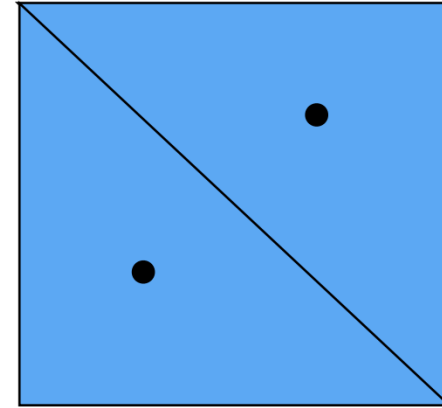
- Sort by partition axis
- Each node saves index start/end range for primitives it is responsible for
 - Combination of children node primitives should match parent node primitives
 - **Example:** all red and yellow primitives encased in orange primitive list
- When partitioning a node along an axis, 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

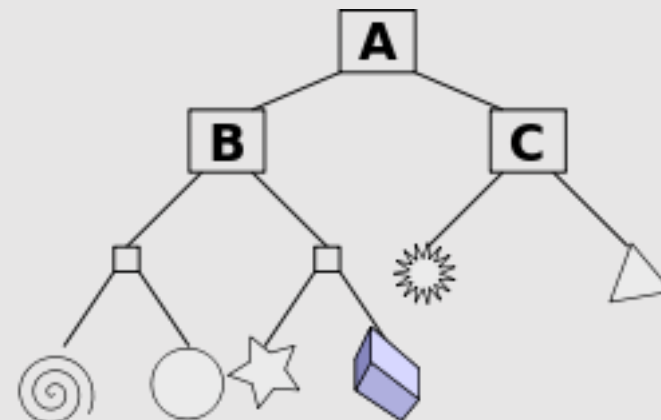
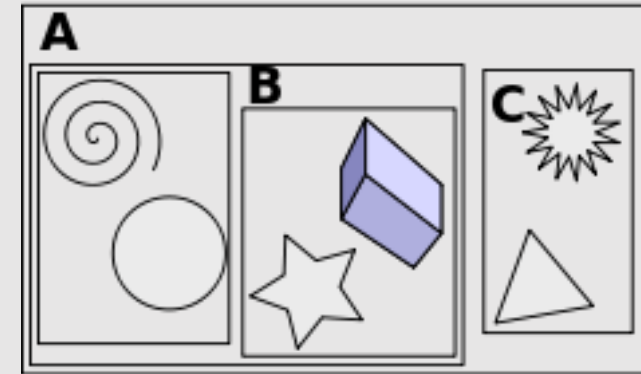
Building the BVH:

- 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:

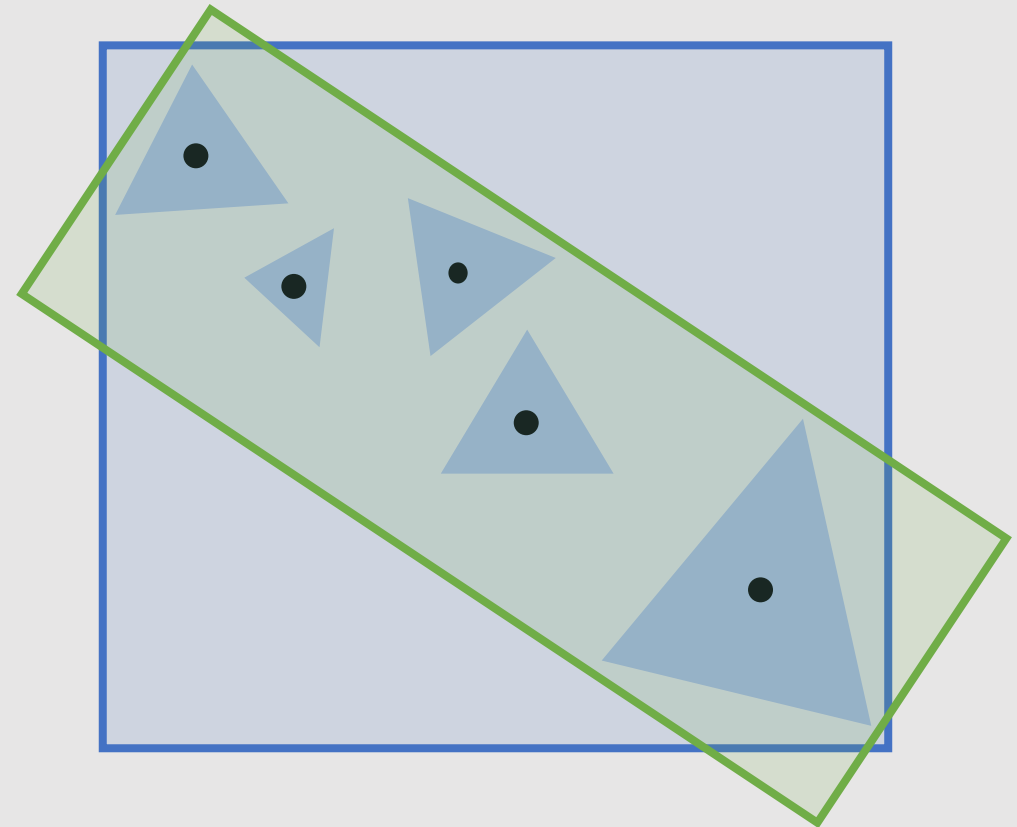
- 1) Check if ray hits current node bbox
- 2) If hit, find which child node is closer to ray
- 3) Recurse down closer child
- 4) If the farther child node is closer to the ray than the hit discovered, recurse down the farther child

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?**

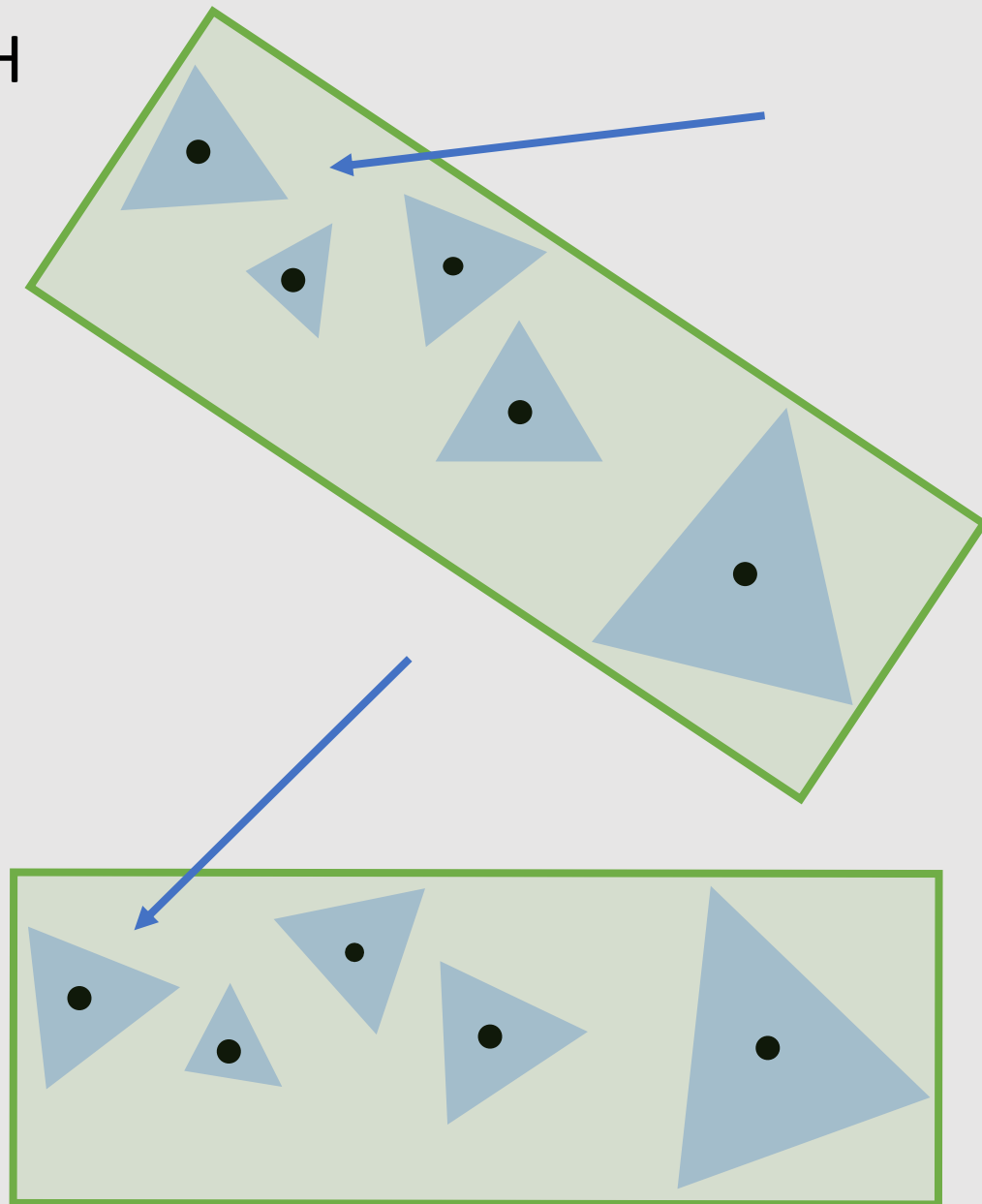
- Yes, and no.

$$C = C_{trav} + \frac{S_A}{S_C} N_A C_{tri} + \frac{S_B}{S_C} N_B C_{tri}$$

- Surface area ratio $\frac{S_A}{S_C}$ decreases with better-fitting bboxes
- Bounding box intersection cost C_{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

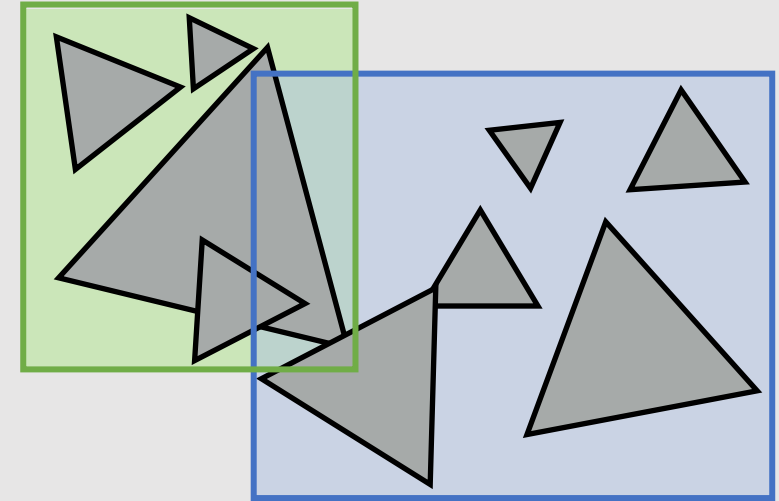


- ~~Ray-Triangle Intersections~~
- ~~Bounding Volume Hierarchy~~
- Spatial-Partitioning Structures

Primitive vs. Spatial

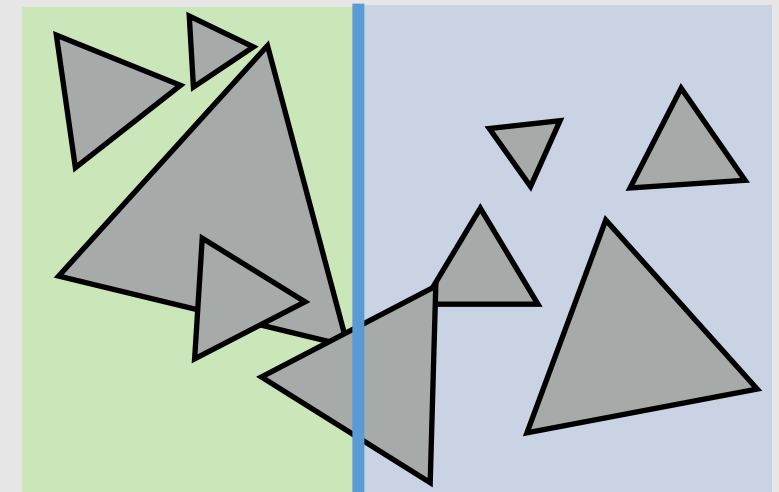
- **Primitive Partitioning**

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

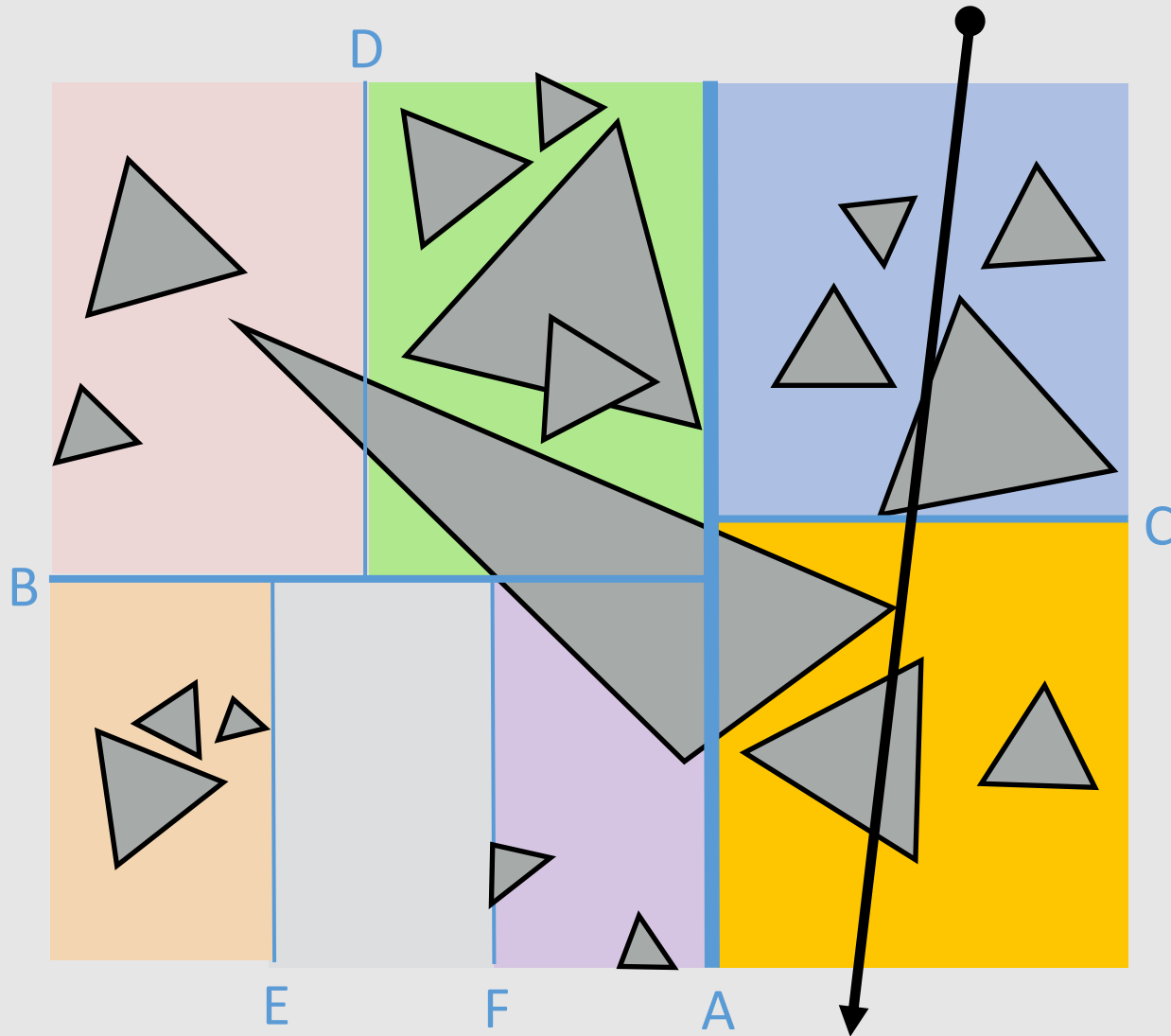


- **Spatial Partitioning**

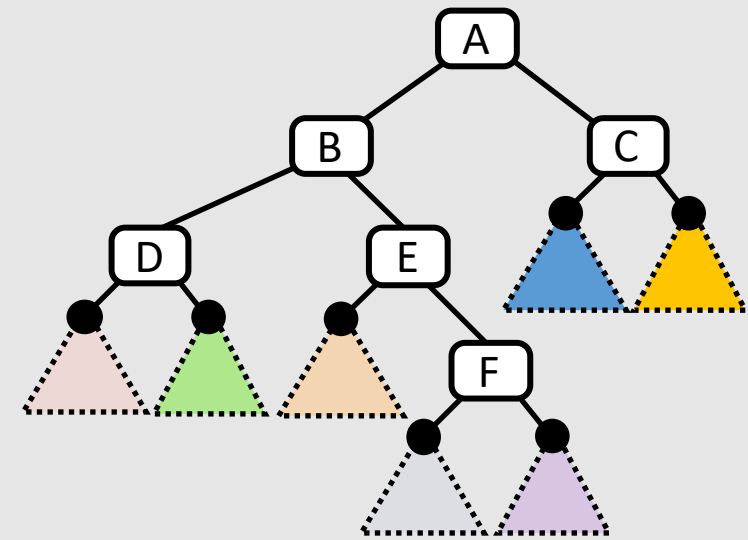
- K-D Trees
- Uniform Grid
- Quad/Octree
 - [+] No volume overlap
 - [+] Can terminate on first hit
 - [-] Higher potential for empty space
 - [-] May intersect primitive multiple times



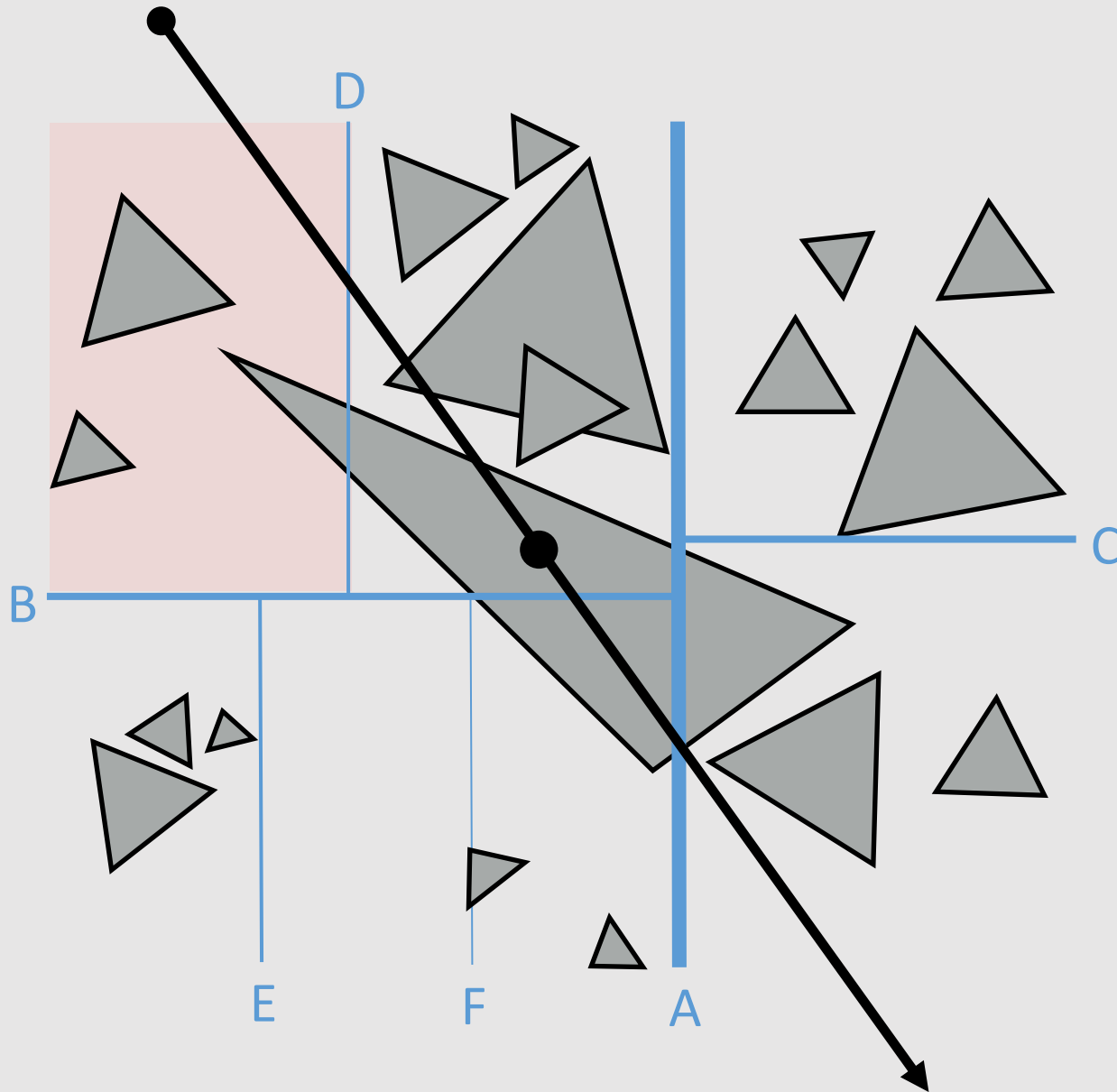
K-D Trees



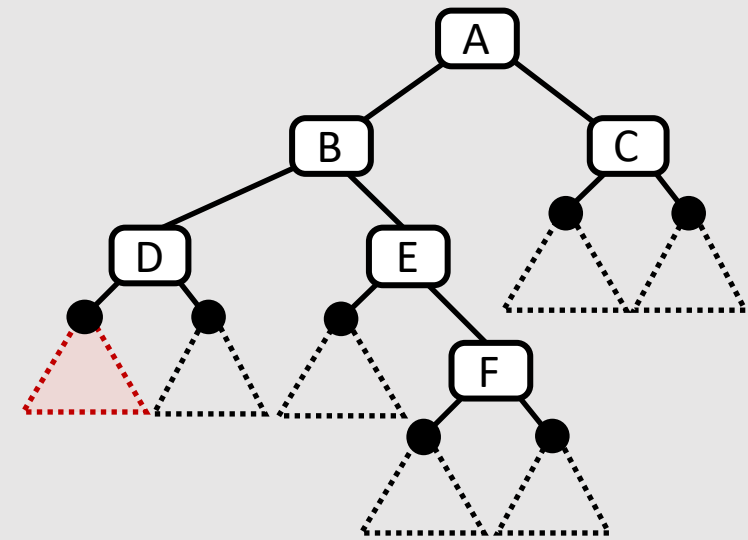
- Recursively partition space via axis-aligned partitioning planes
 - Interior nodes correspond to spatial splits
 - Node traversal proceeds in front-to-back order
 - Unlike BVH, can terminate search after first hit is found
 - Still $O(\log(N))$ performance



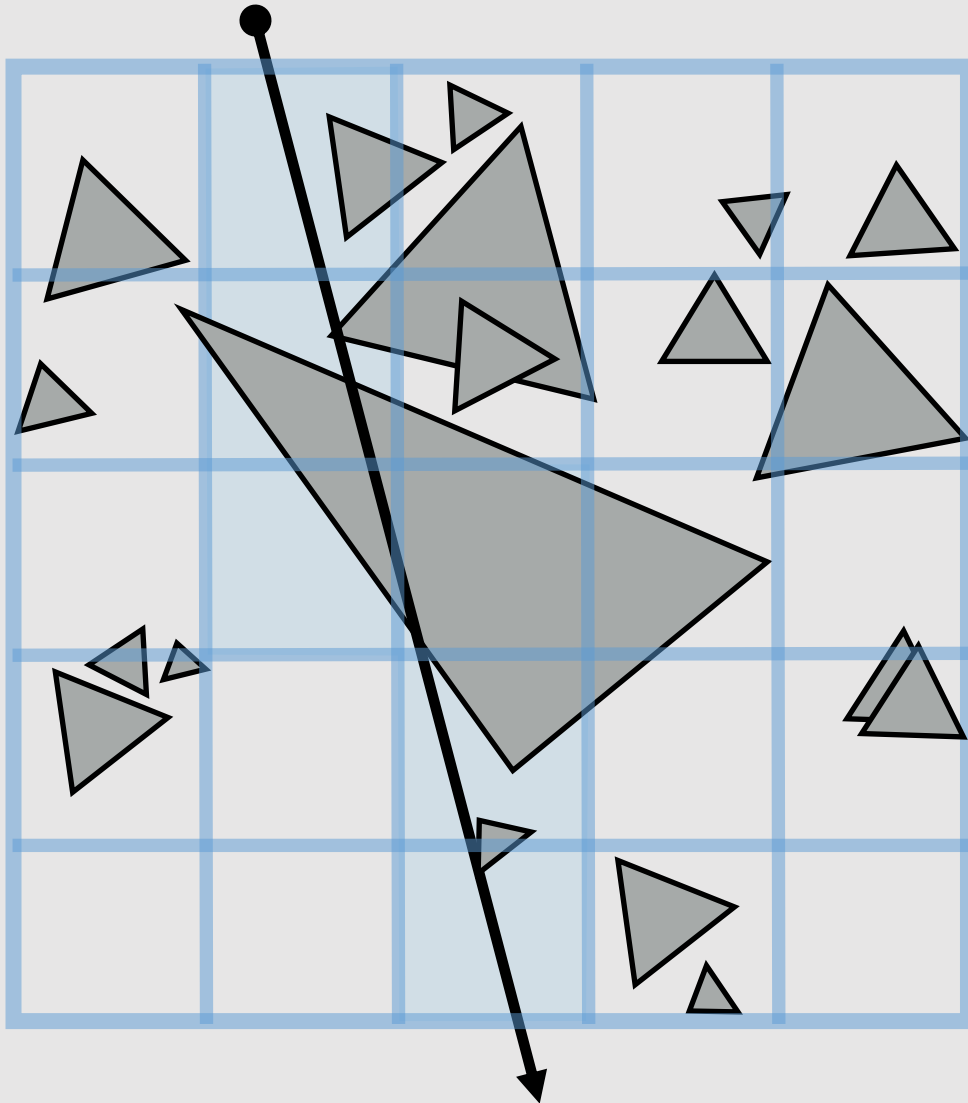
K-D Trees



- **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

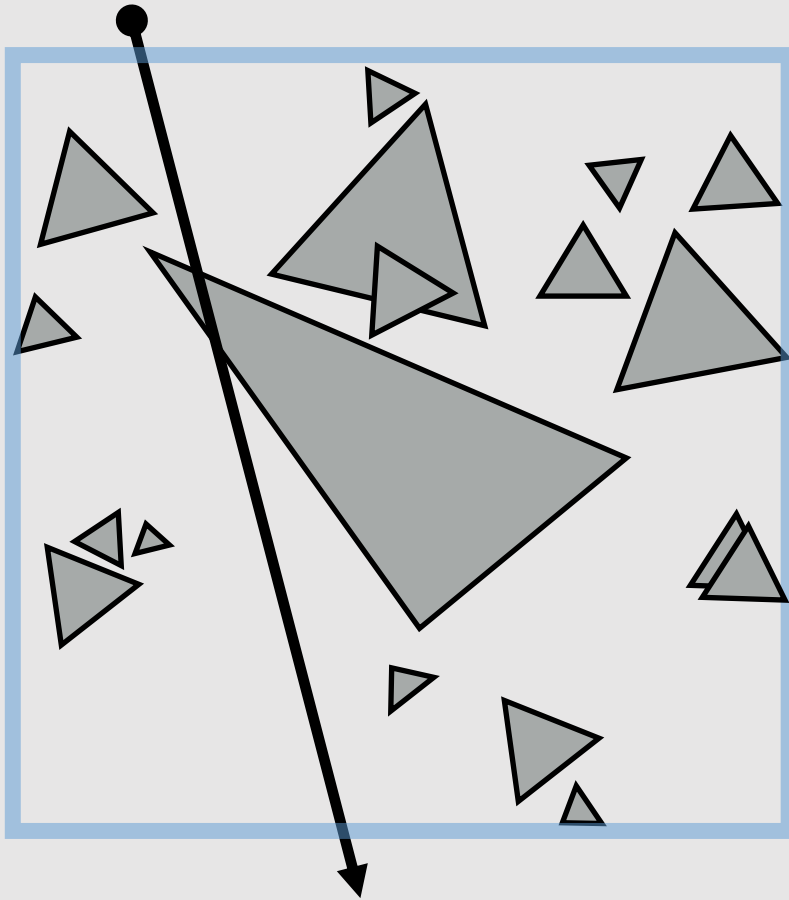


Uniform Grid

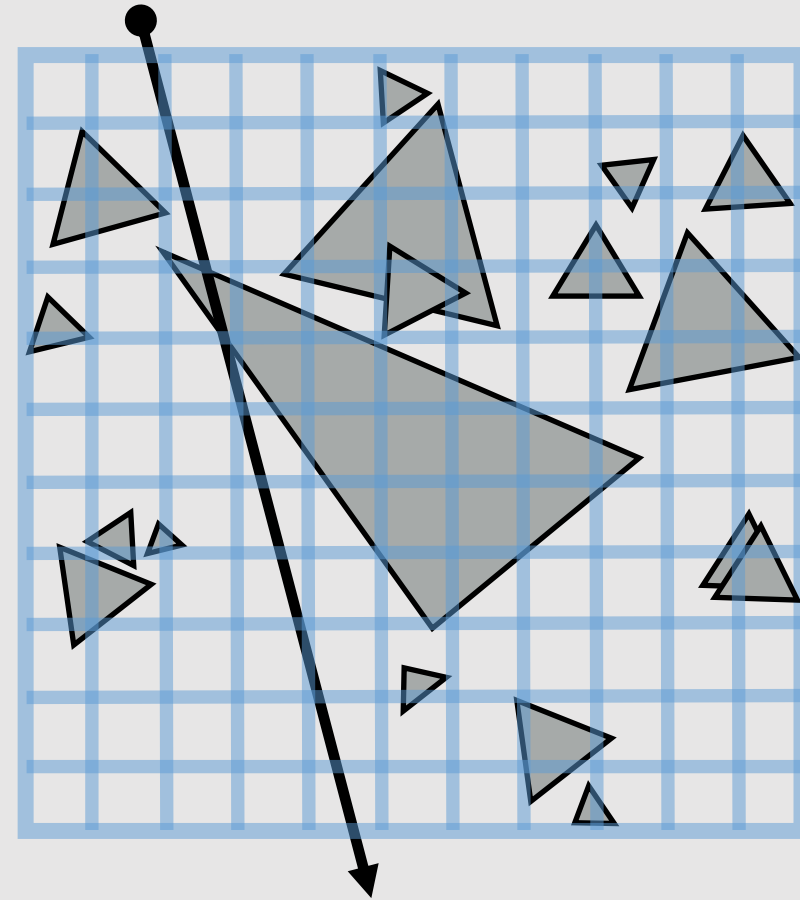


- Partition space into equal sized volumes (volume-elements or **“voxels”**)
- Each voxel contains primitives that overlap
- Walk ray through volume in order
 - Very efficient implementation possible (think: 3D line rasterization)
 - Only consider intersection with primitives in voxels the ray intersects
- 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

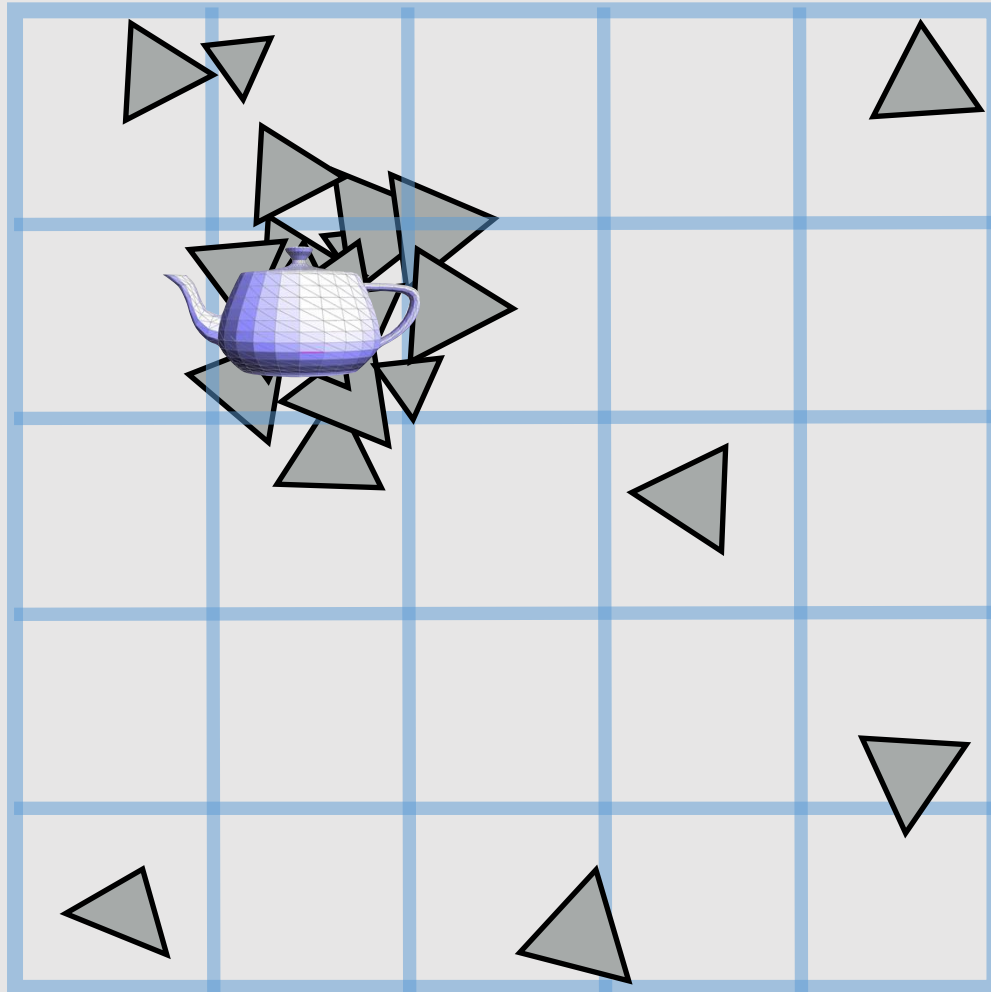


Too few cells
Requires checking every primitive



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



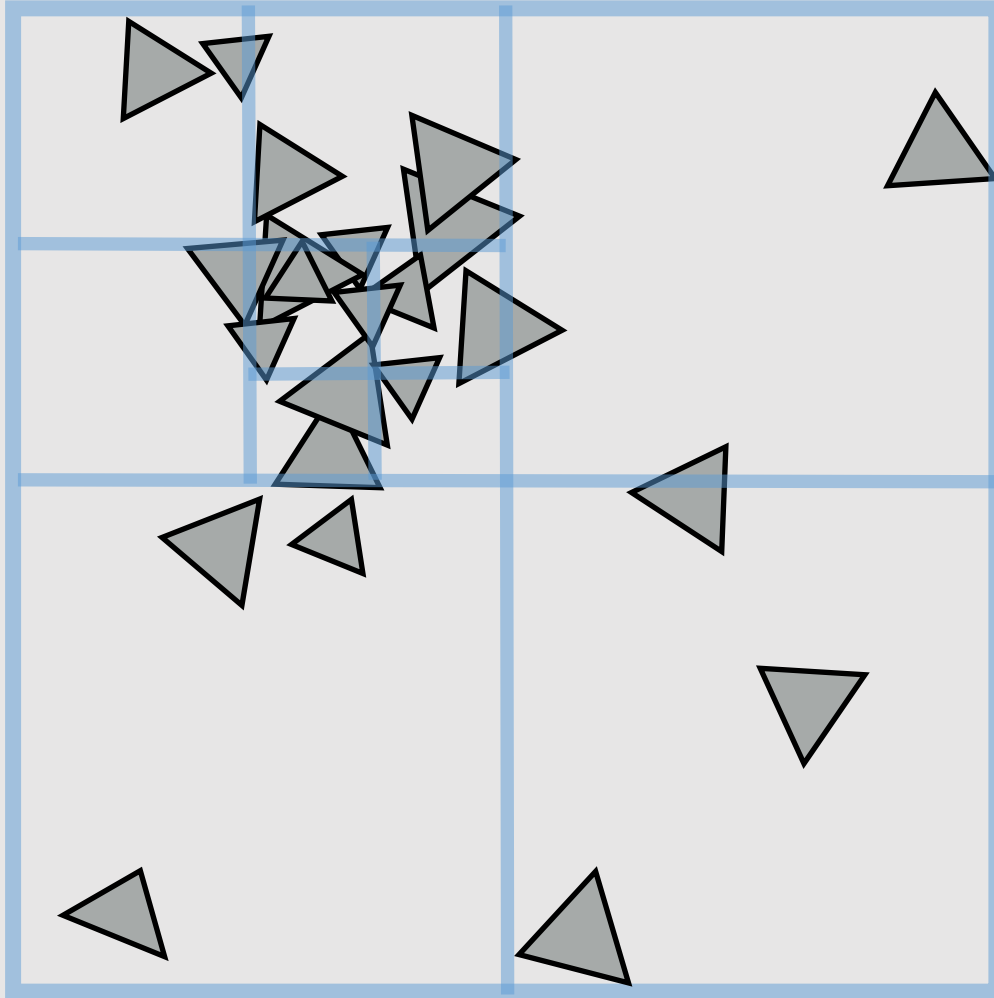
Monsters University (2013) Pixar

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

	[Spatial]	[Primitive]	[Build Speed]	[Search Speed]
BVH	X	✓	X	✓
K-D Tree	✓	X	X	✓
Uniform Grid	✓	X	✓	X
Quad/Octree	✓	X	X	✓