Lecture 16, Part A:

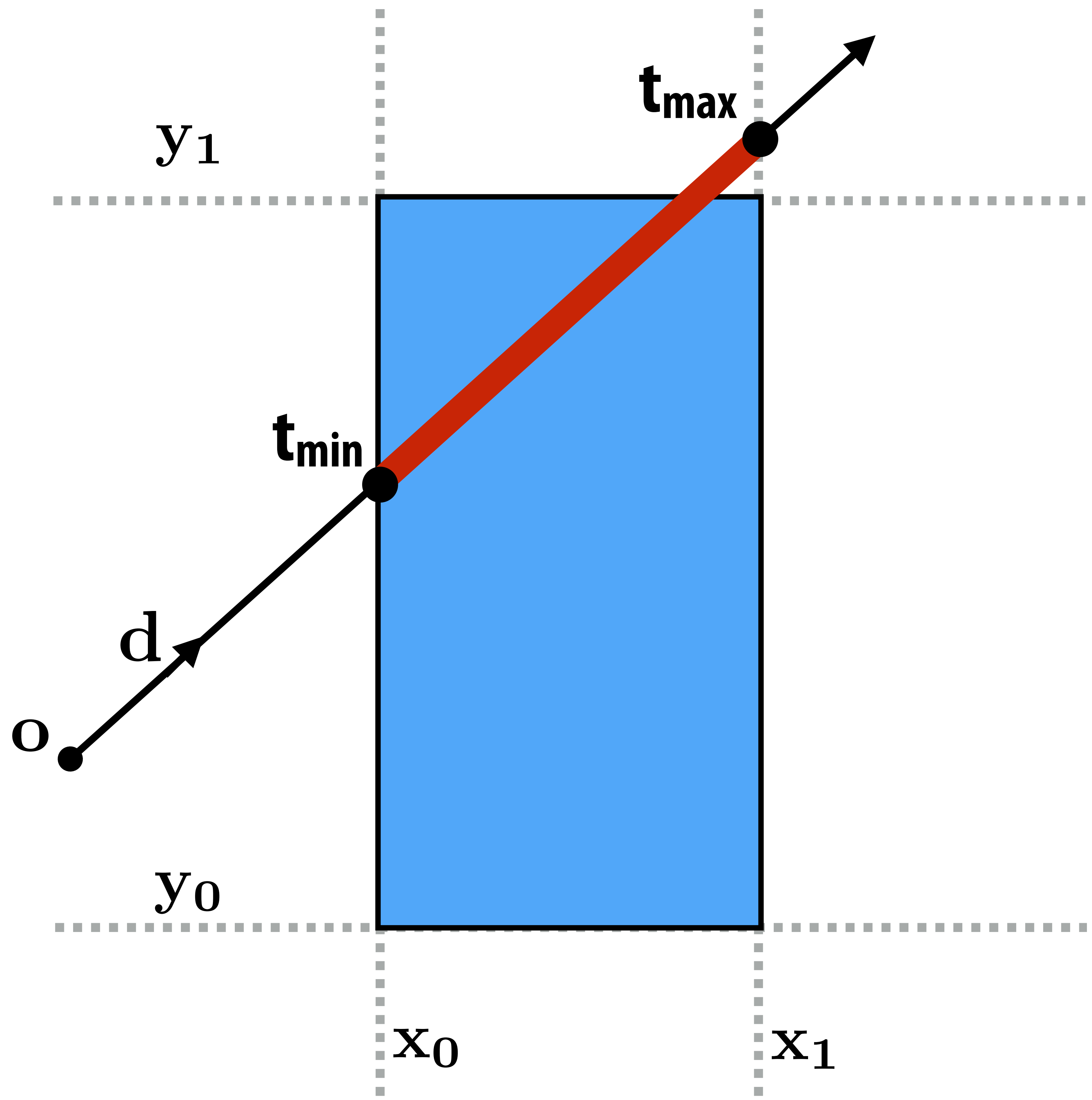
Bounding Volume Hierarchies

Computer Graphics

CMU 15-462/15-662, Spring 2017

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:

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

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

$$\mathbf{N}^{\mathbf{T}} = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$$

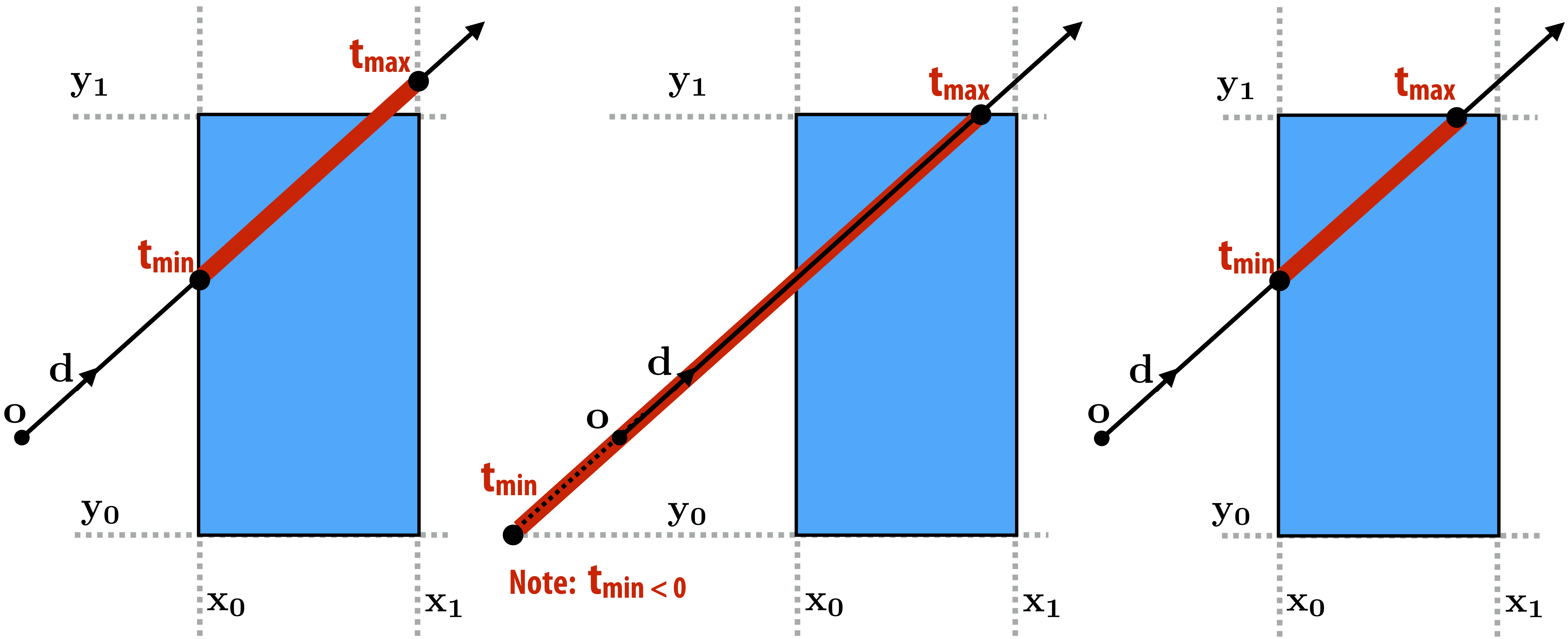
$$c = x_0$$

$$t = \frac{x_0 - \mathbf{o_x}}{\mathbf{d_x}}$$

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



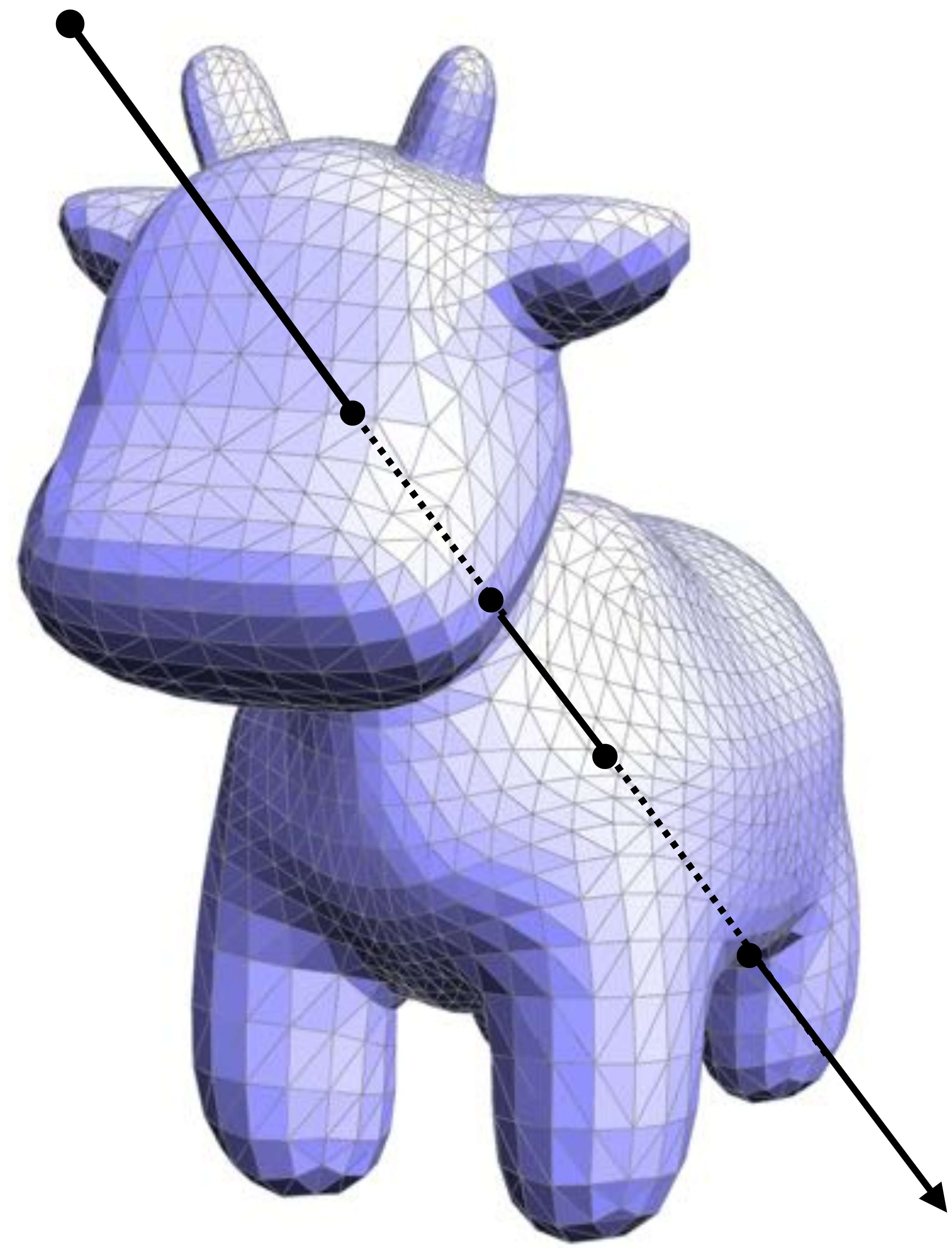
Intersections with *x* planes | **Intersections with *y* planes** | **Final intersection result**

How do we know when the ray misses the box?

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_closest = NULL
t_closest = inf
for each primitive p in scene:
   t = p.intersect(r)
   if t >= 0 && t < t_closest:
      t_closest = t
      p_closest = p
```

Complexity: $O(N)$

A simpler problem

- Imagine I have a set of integers S
- Given a new integer k, find the element in S that is closest to k:

`10    123    20    100    6    25    64    11    200    30`

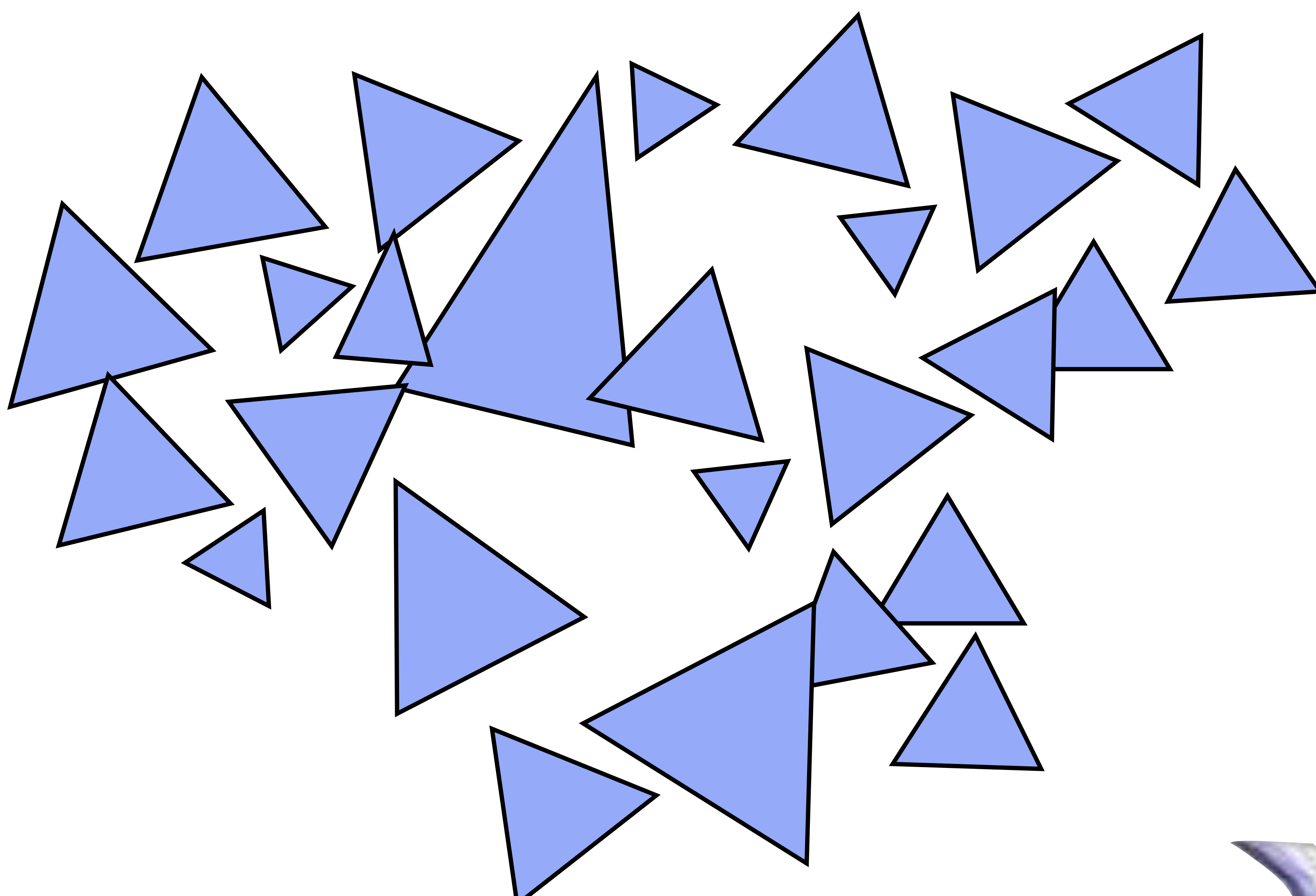
Example: k=18

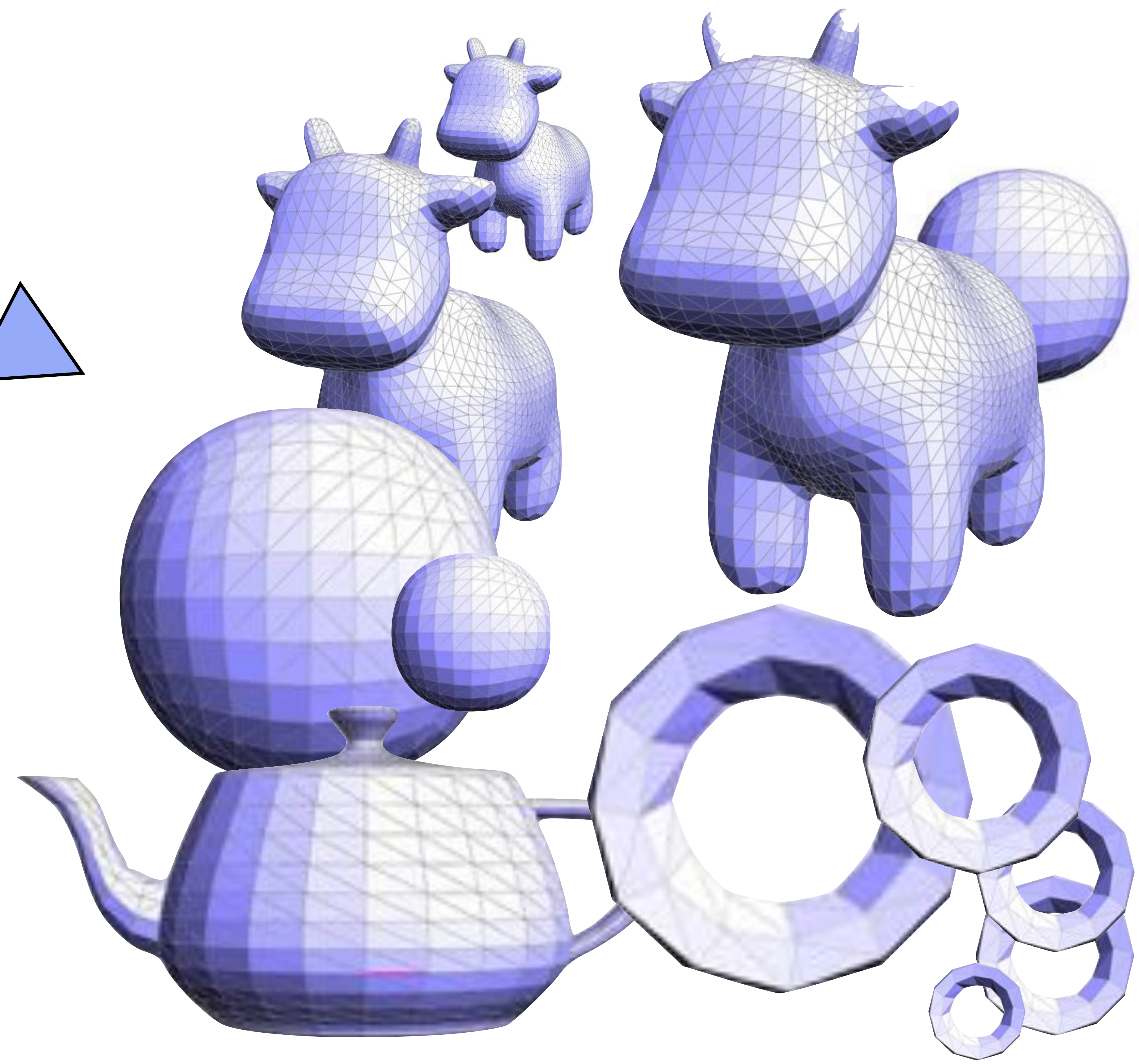
Sort integers:

`6    10    11    20    25    30    64    100    123    200`

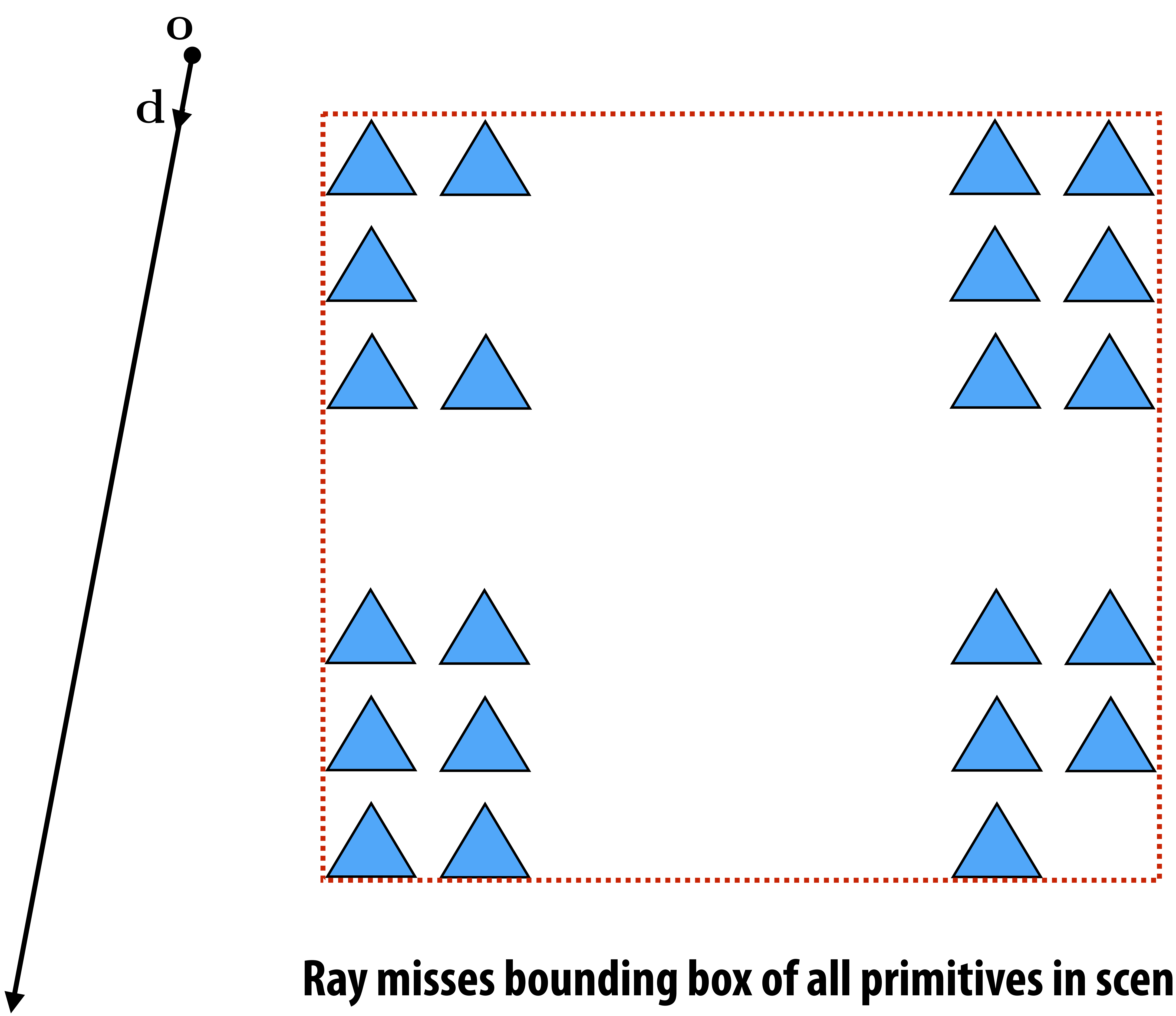
How would you perform a modified binary search?

How do we organize scene primitives to enable fast ray-scene intersection queries?





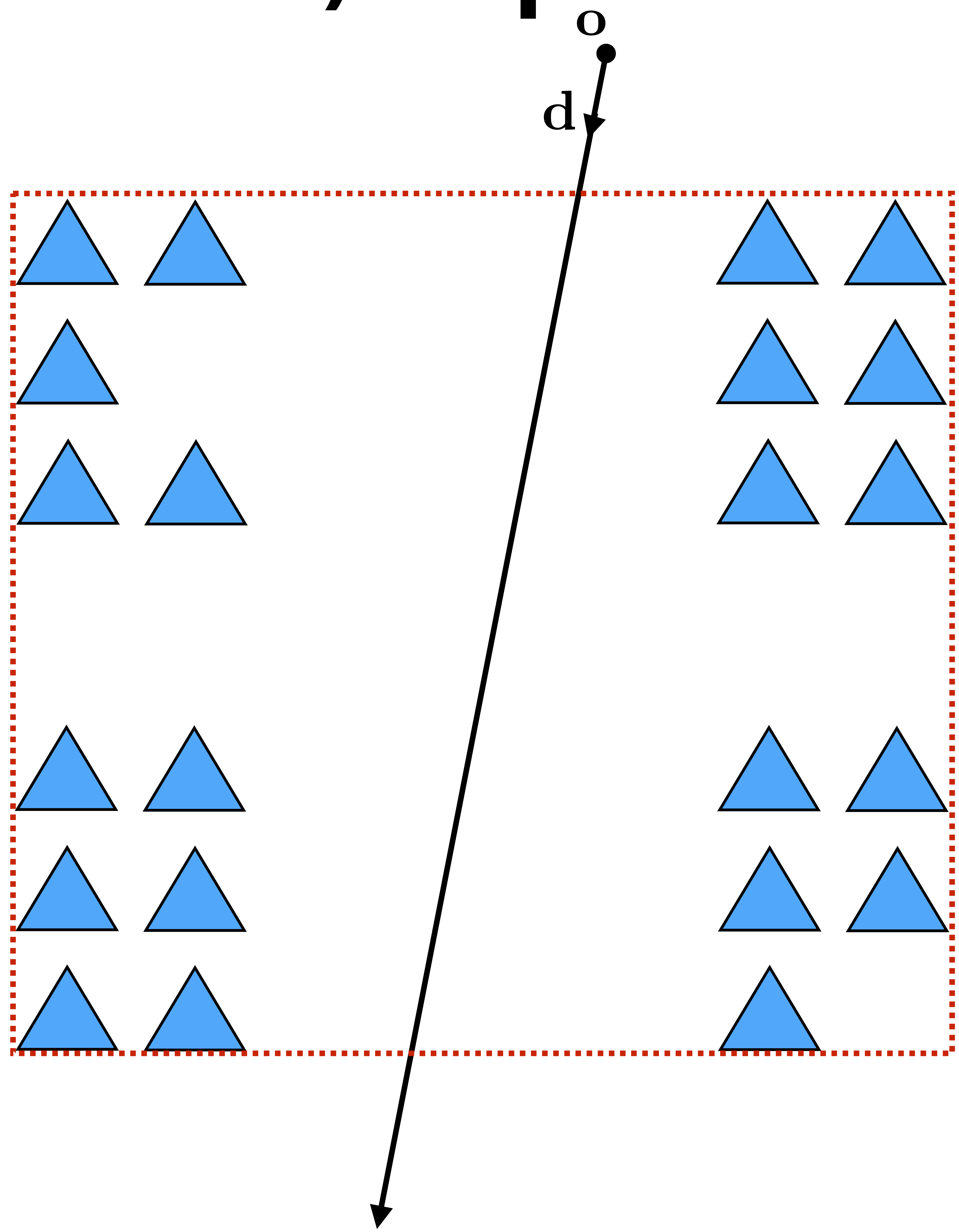
Simple case



Ray misses bounding box of all primitives in scene

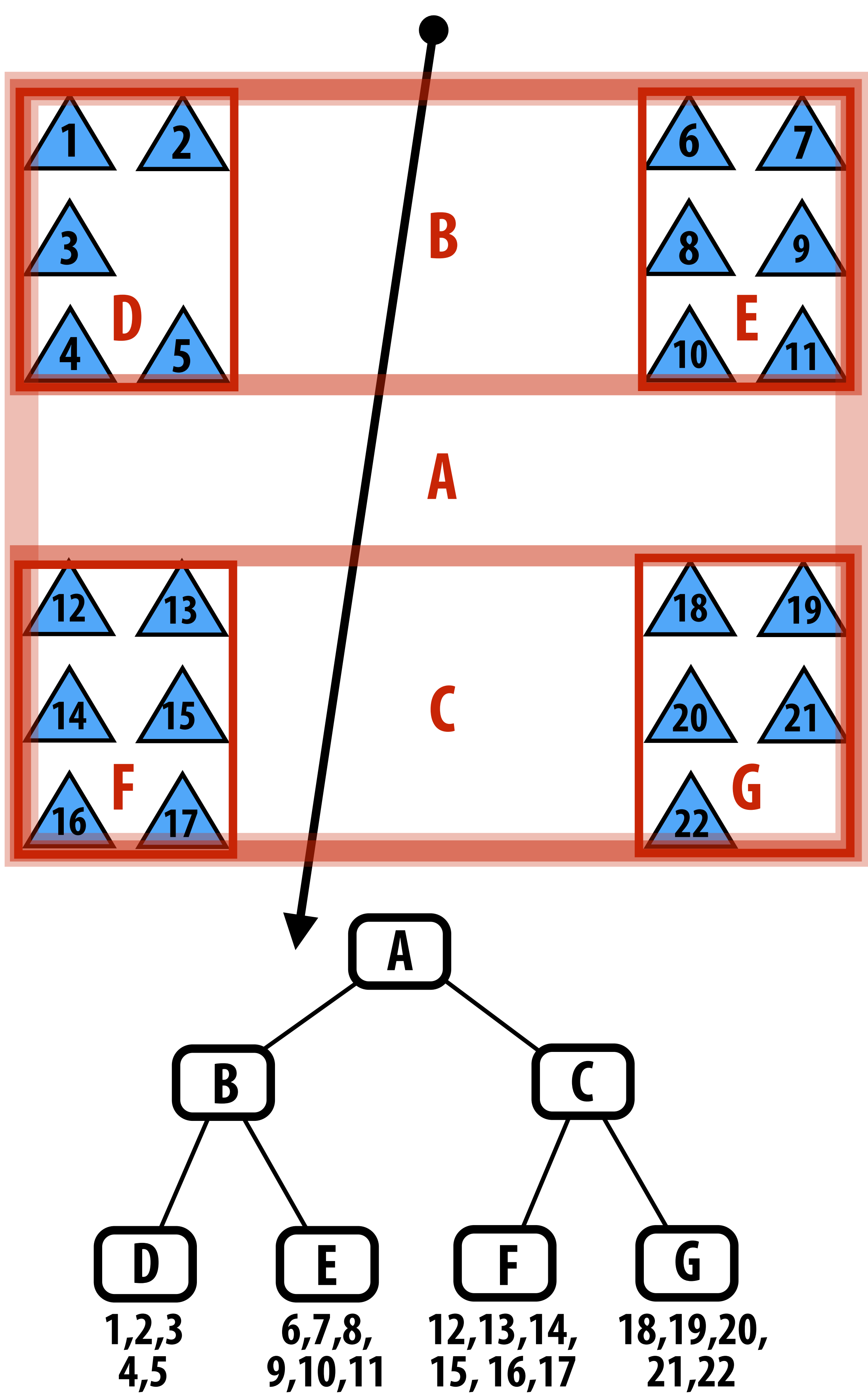
O(1) cost: requires 1 ray-box test

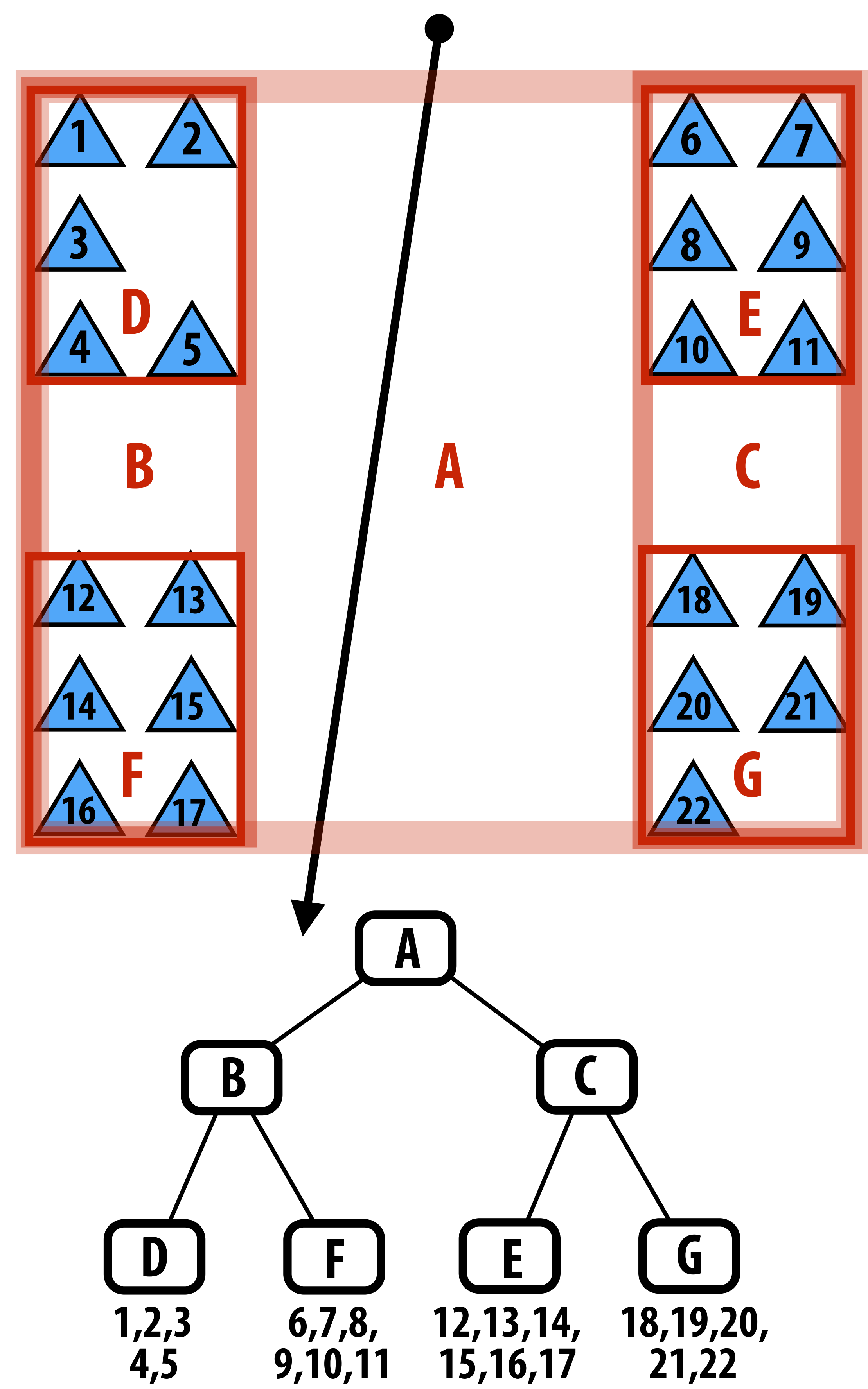
Another (should be) simple case



Bounding volume hierarchy (BVH)

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



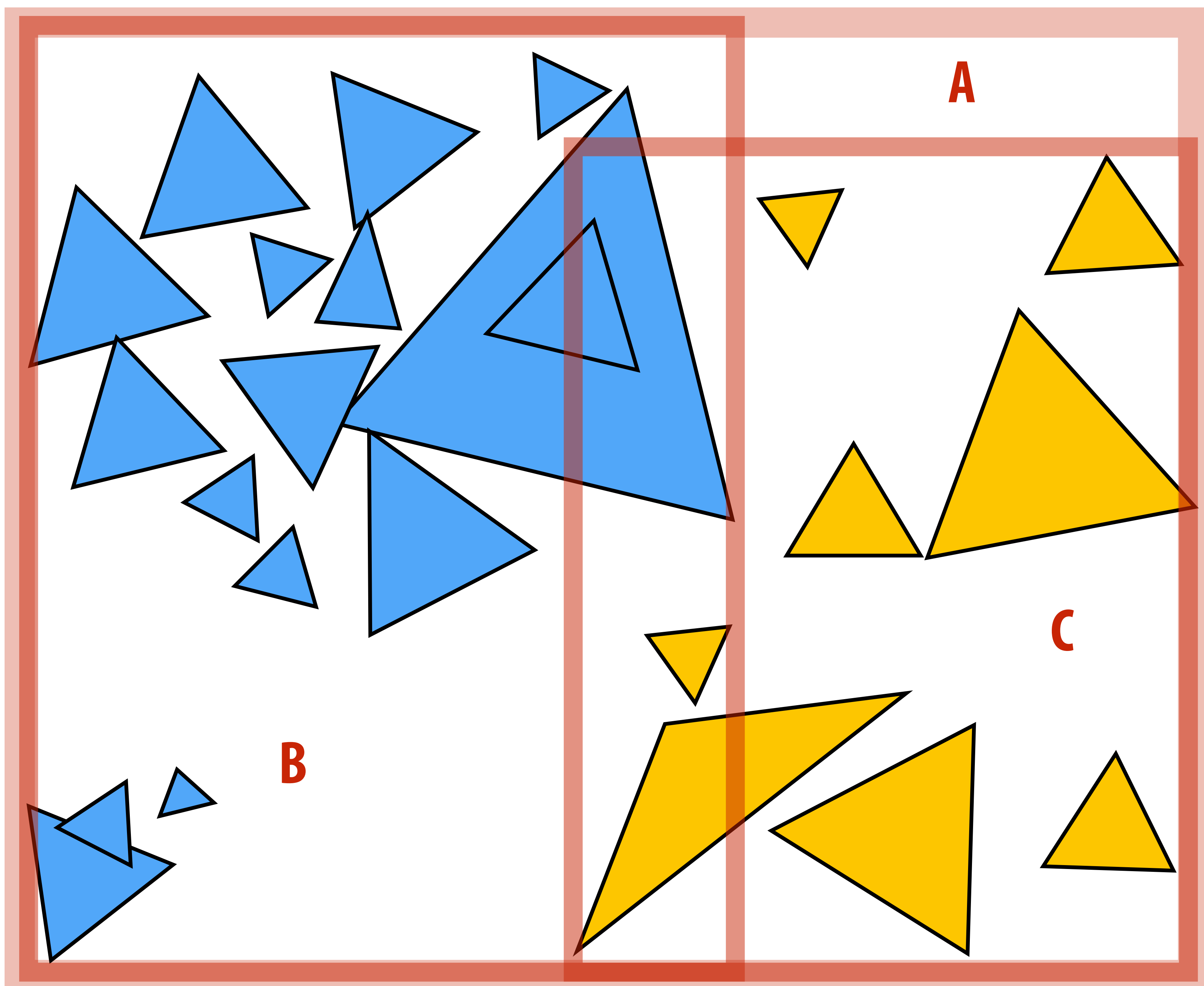


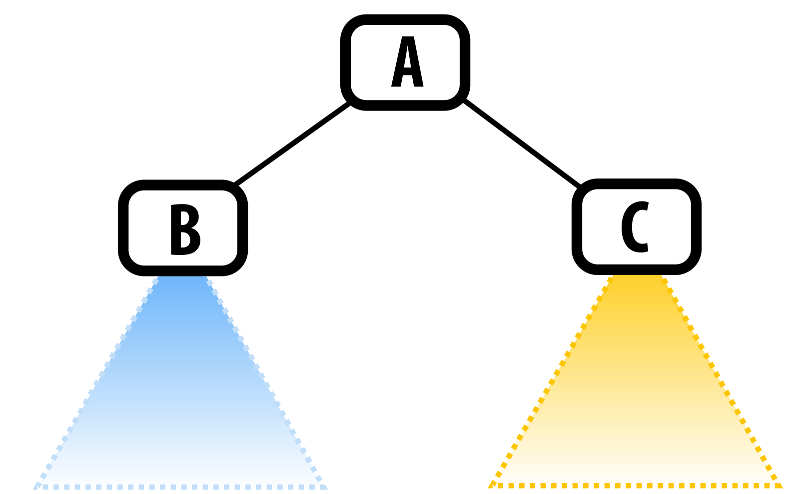
Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Another BVH example

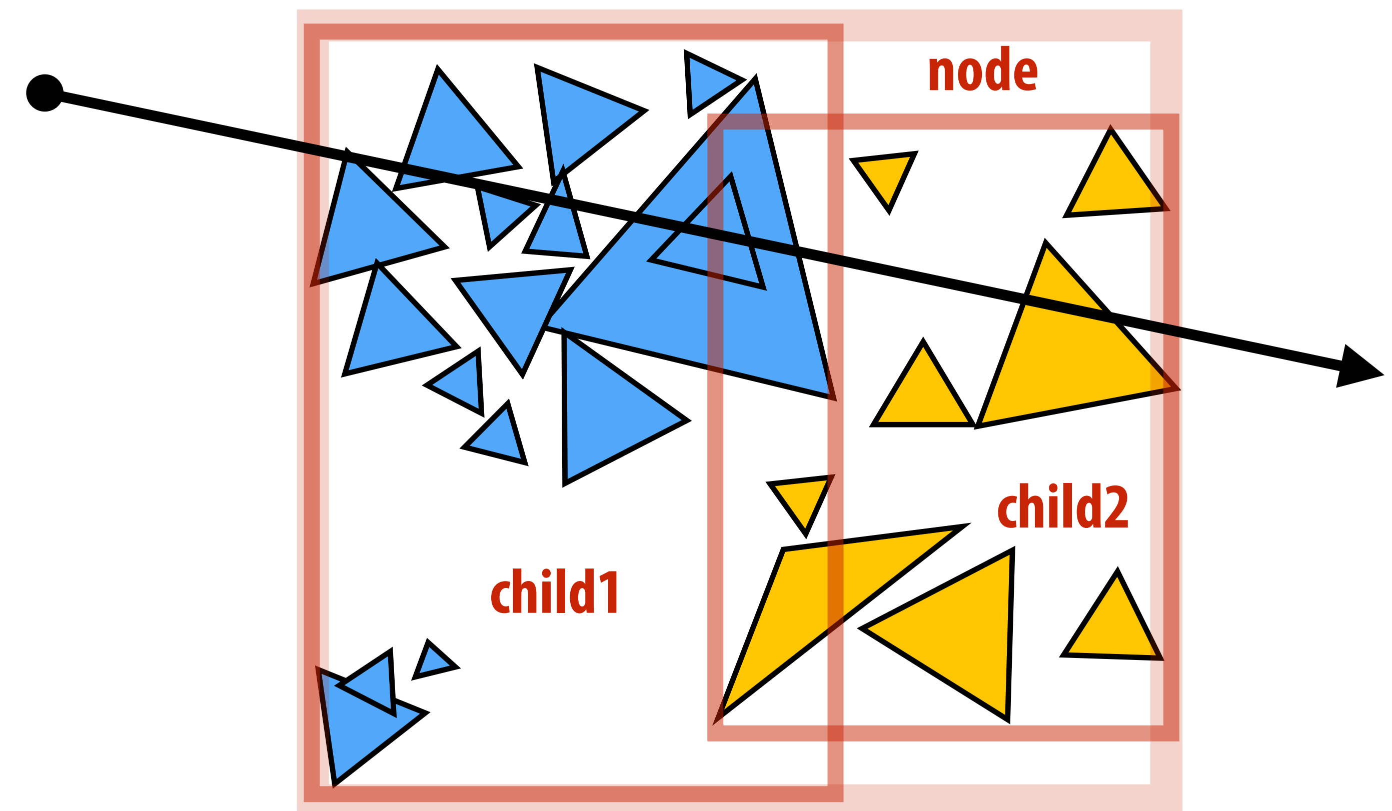
- BVH partitions each node's primitives into disjoints sets
  - Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)





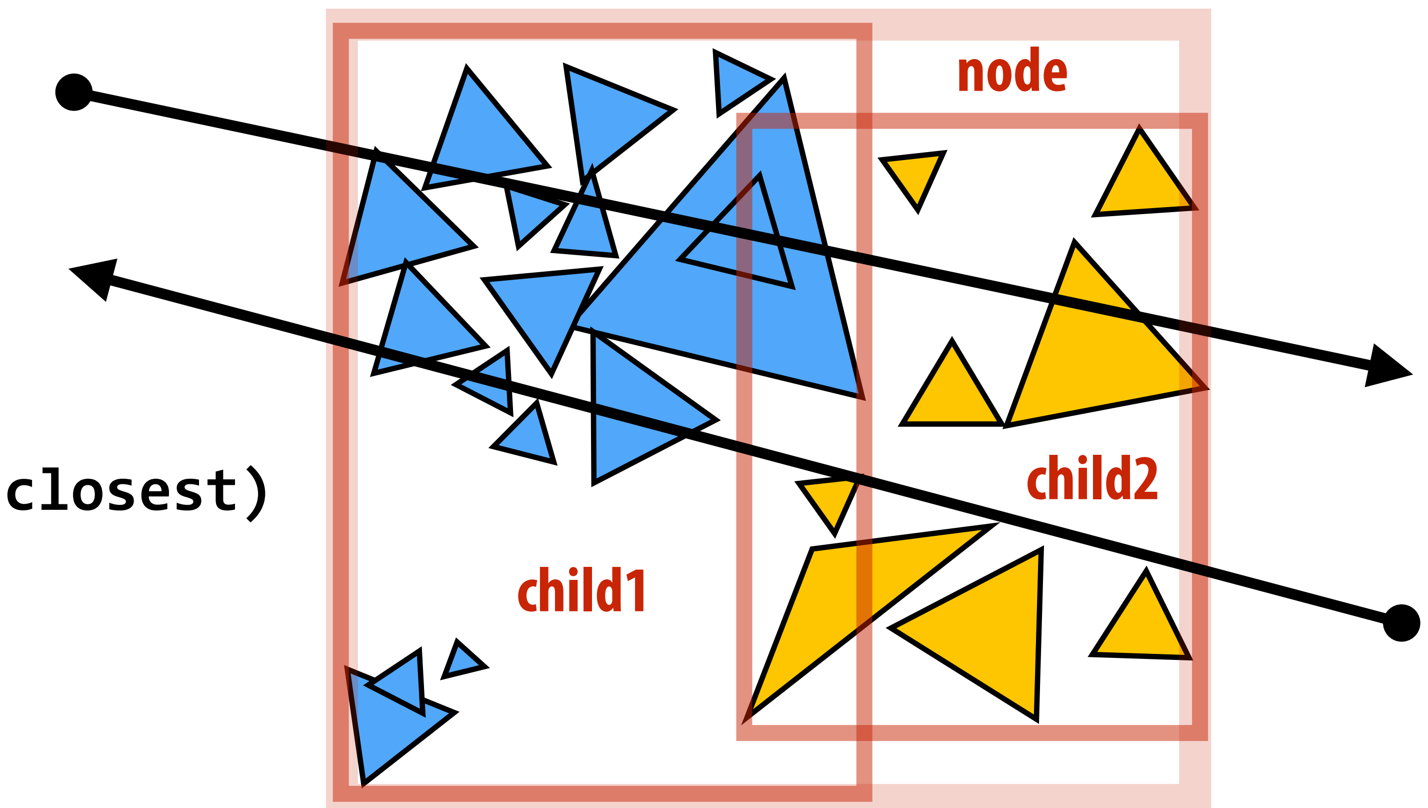
Ray-scene intersection using a BVH

```
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) || (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);
   }
}
```



How could this occur?

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, 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)

```
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) );
   }
}
```

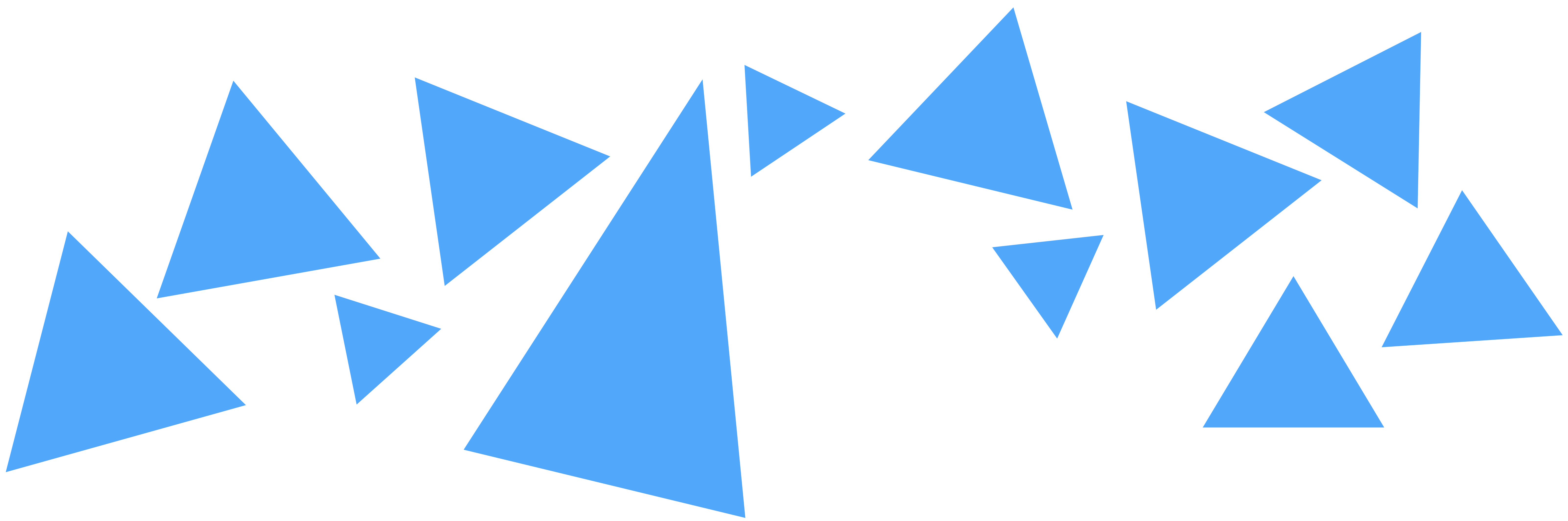
Interesting question of which child to enter first. How might you make a good decision?

For a given set of primitives, there are many possible BVHs

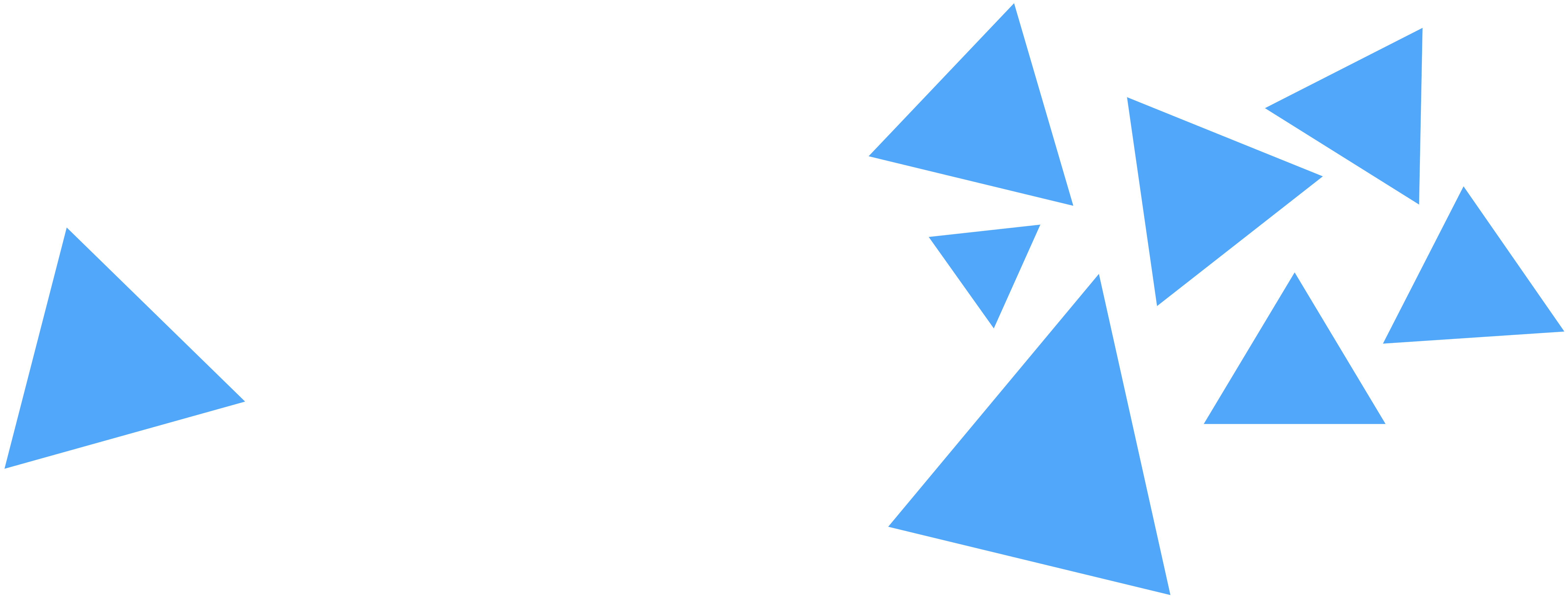
(2^N-2 ways to partition N primitives into two groups)

How do we build a high-quality BVH?

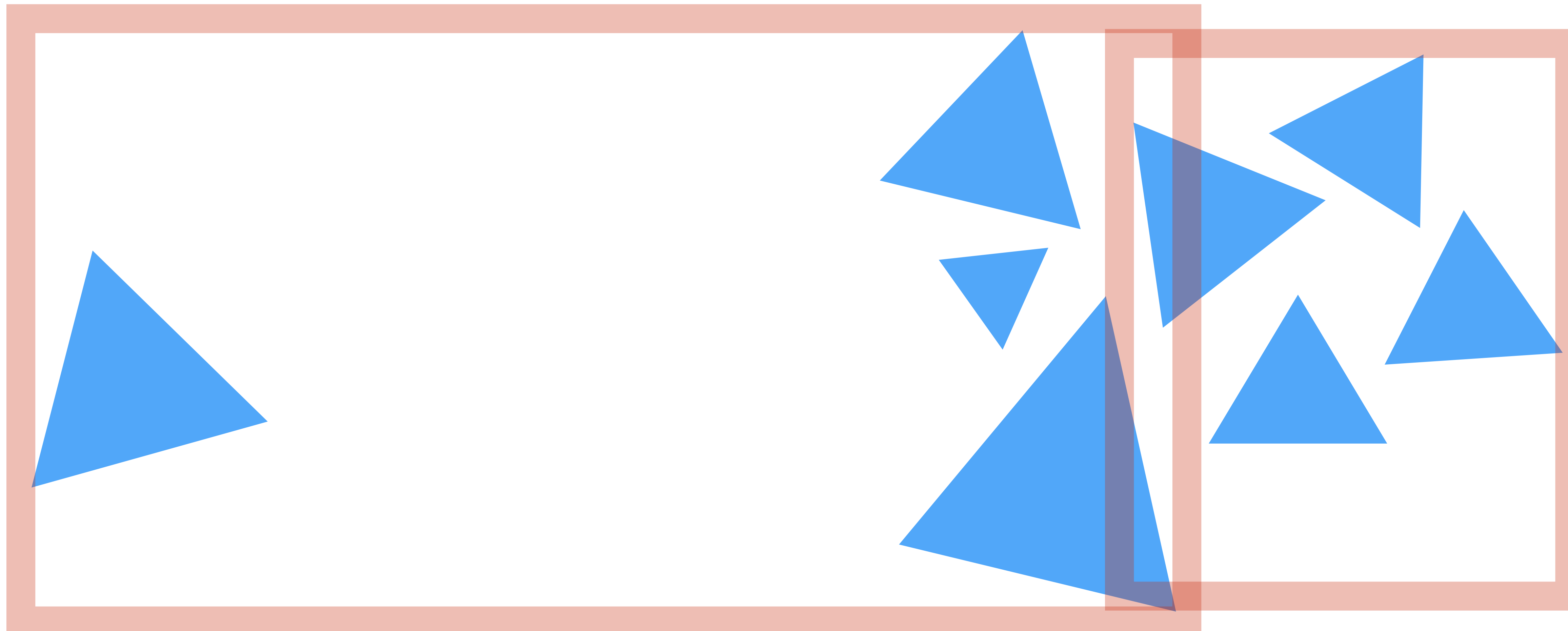
How would you partition these triangles into two groups?



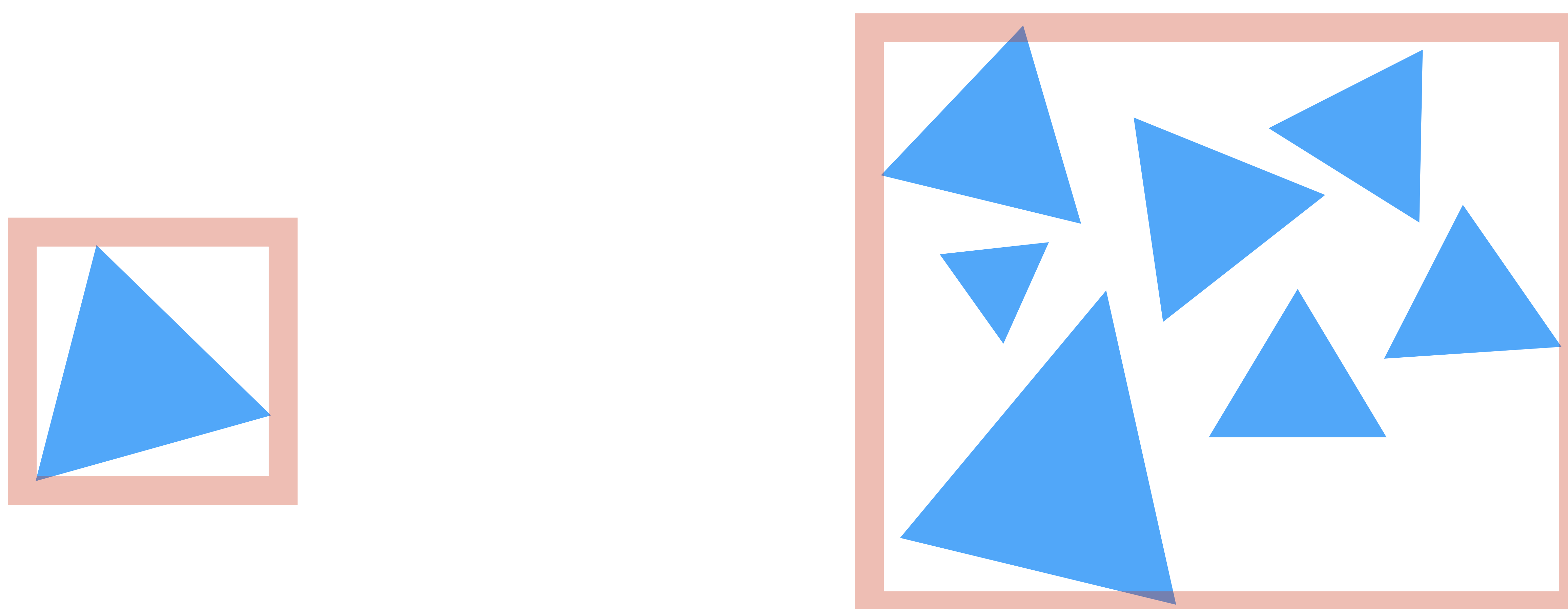
What about these?



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 <u>cost</u> 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 <u>expected cost</u> 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_{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:

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

Where: $N_A = |A|, N_B = |B|$

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):

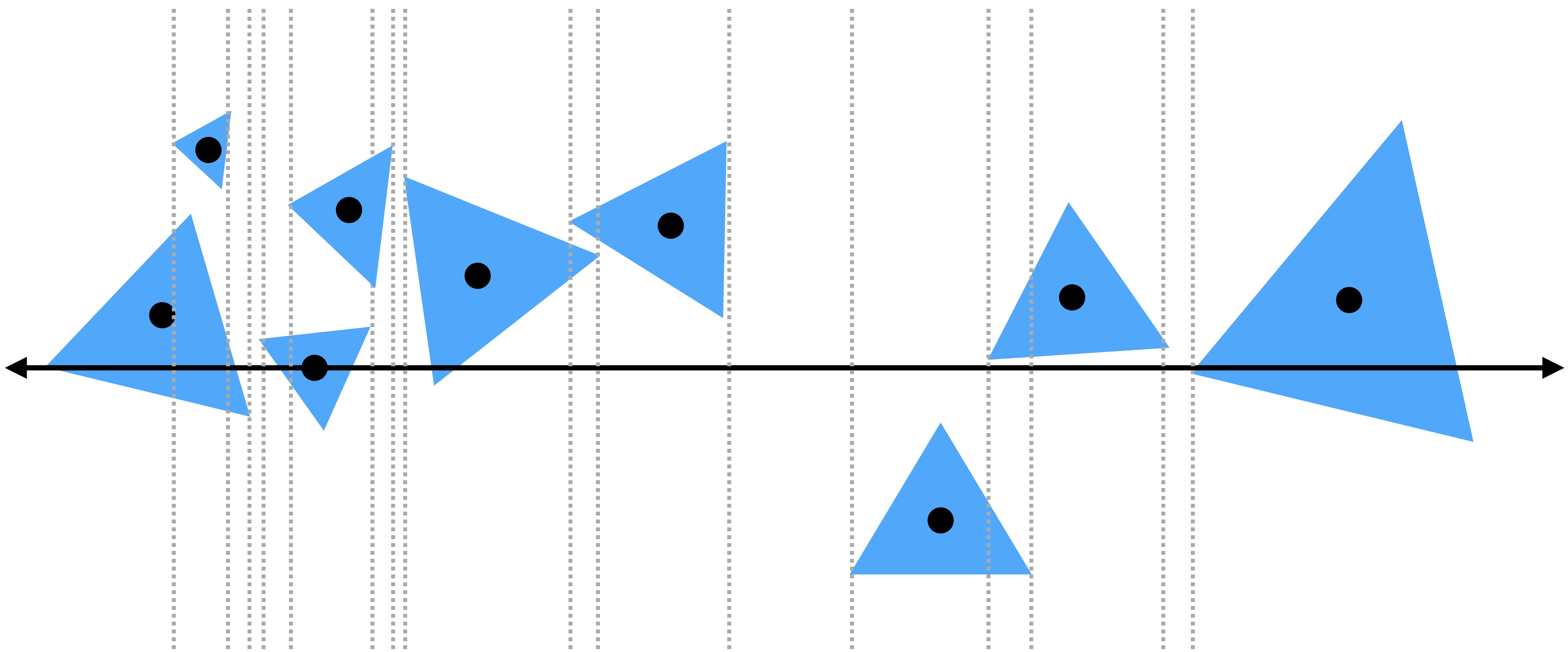
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

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

- **Rays are randomly distributed**
- **Rays are not occluded**

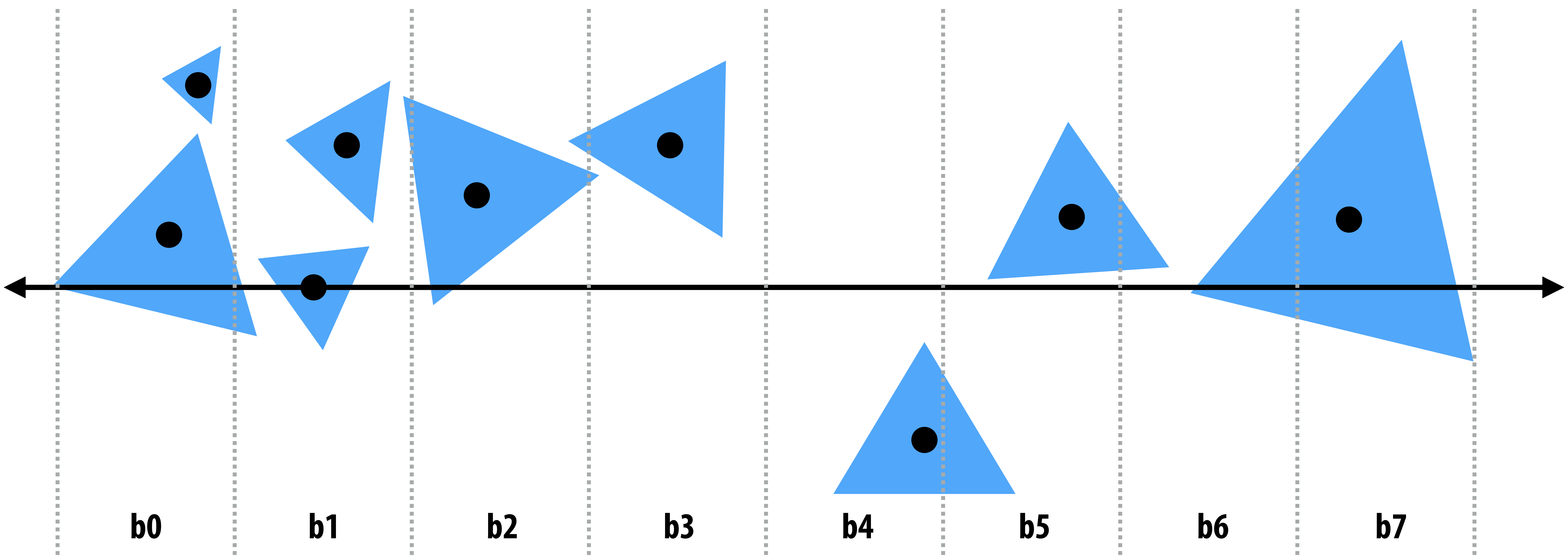
Implementing partitions

- 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
  - 2N-2 possible splitting positions for node with N primitives. (Why?)



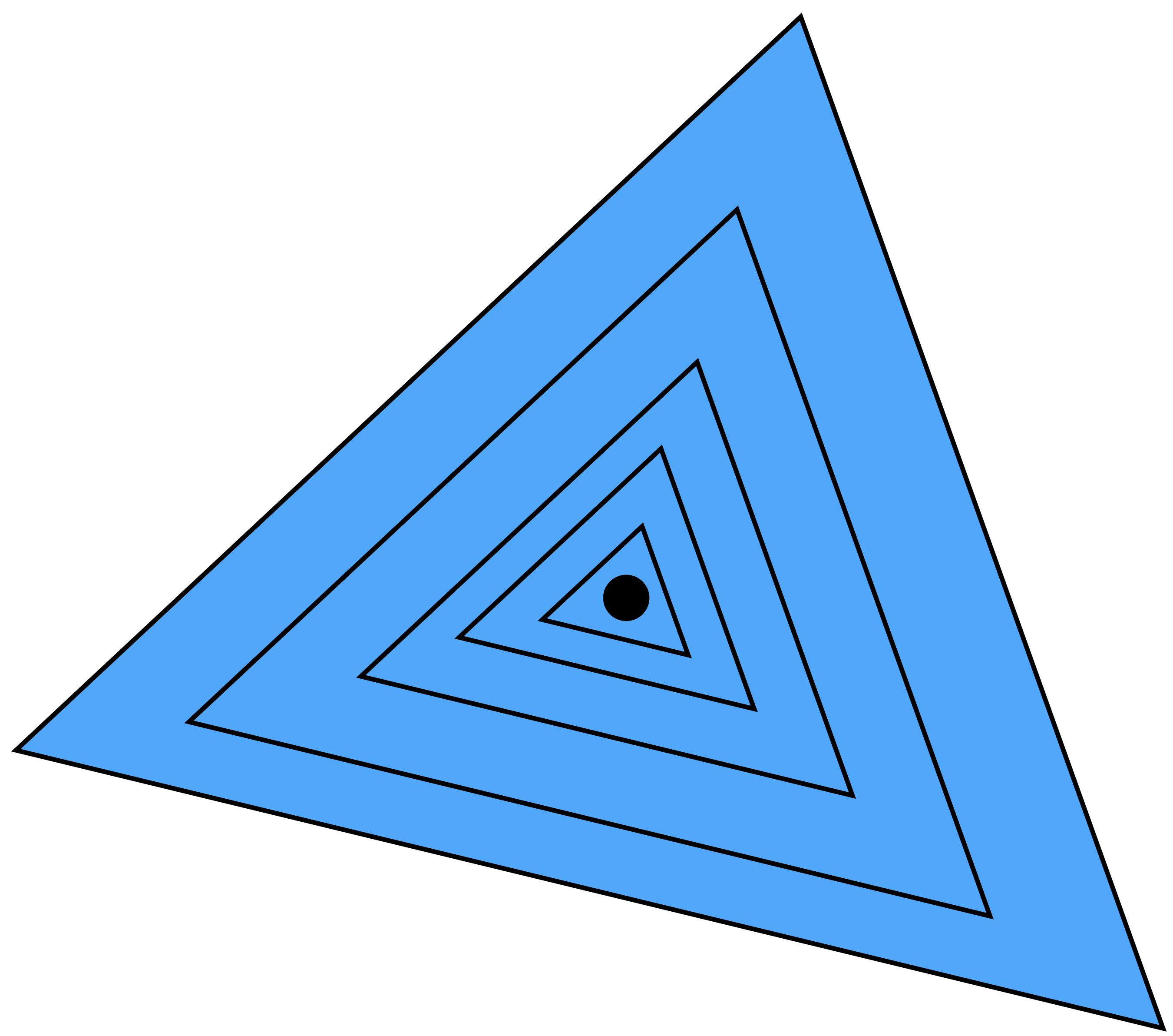
Efficiently implementing partitioning

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

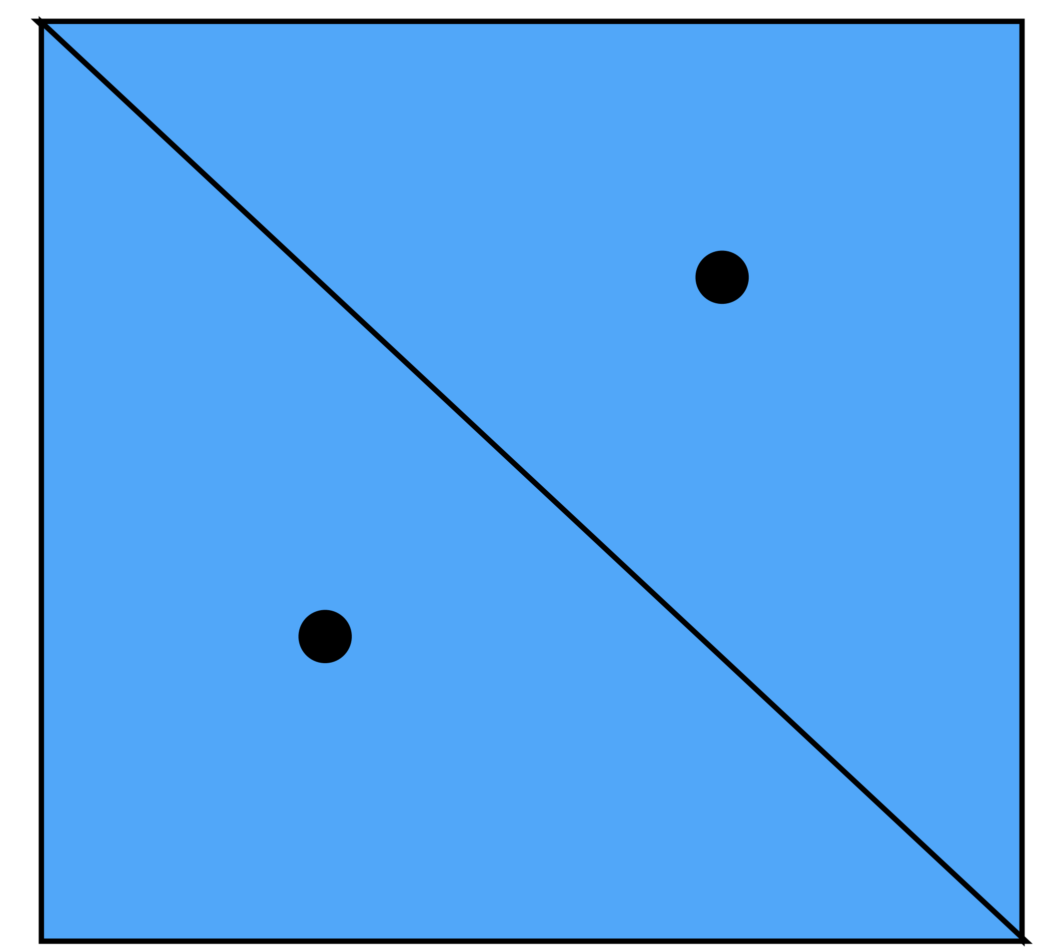



```
For each axis: x,y,z:
   initialize buckets
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Execute lowest cost partitioning found (or make node a leaf)
```

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)

What you should know:

- Compute ray - bounding box intersection
- Construct a bounding box hierarchy for a given collection of objects.
- Calculate traversal order of a bounding box hierarchy for a given ray.
- What is the Surface Area Heuristic (SAH) and what goals is it trying to achieve?
- Explain how to choose a bounding box partition using the SAH
- (from last week) Be able to distinguish between object-centric (primitive partitioning) acceleration structures and space-centric (space-partitioning) acceleration structures
- (from last week) Know the difference between these acceleration structures, how to build them, how to traverse them, and when to use each type:
  - bounding box and bounding sphere hierarchies
  - KD-trees
  - octrees
  - grids