## Lecture 12:

# Accelerating Geometric Queries

---

### Computer Graphics
### CMU 15-462/15-662, Spring 2018

# Course roadmap

## Drawing Things

Key concepts:
Sampling (and anti-aliasing)
Coordinate Spaces and Transforms

- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- Occlusion and alpha compositing (+ the end-to-end GPU pipeline)

## Geometry

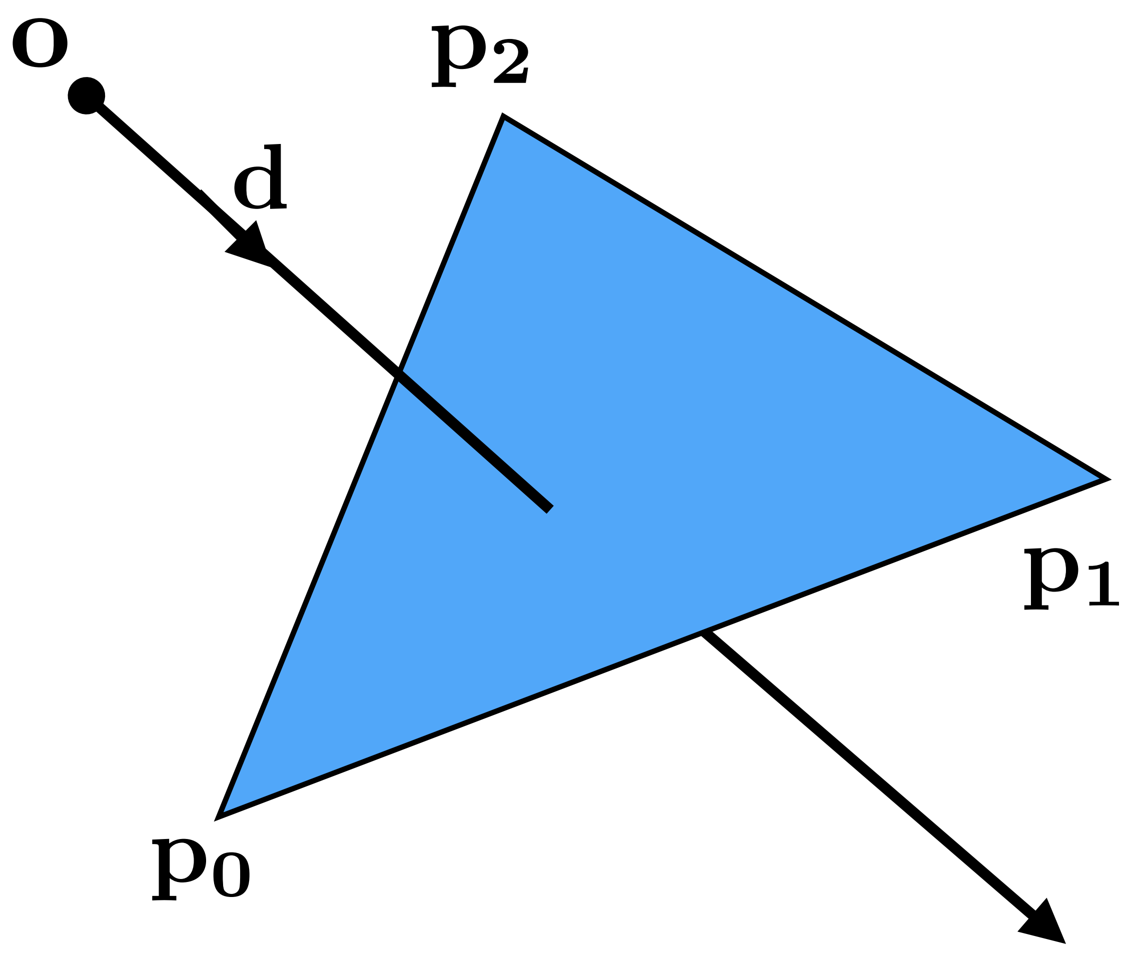
Key concepts:
Implicit vs. explicit representations
Manifold property of surfaces
Geometry processing as resampling

- Representing geometry and surfaces
- Properties of curves and surfaces, mesh representation
- Mesh processing operations
- Geometric queries (e.g., ray-triangle intersection test)
- Accelerating geometric queries (e.g., ray-mesh intersection)

## Materials and Lighting

# Complexity of geometry

# Review: ray-triangle intersection



- **Find ray-plane intersection**

**Parametric equation of a ray:**

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

ray origin

normalized ray direction

**Plug equation for ray into implicit plane equation:**

$$\mathbf{N}^{\mathbf{T}}\mathbf{x} = c$$

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

**Solve for t corresponding to intersection point:**

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

- **Determine if point of intersection is within triangle**

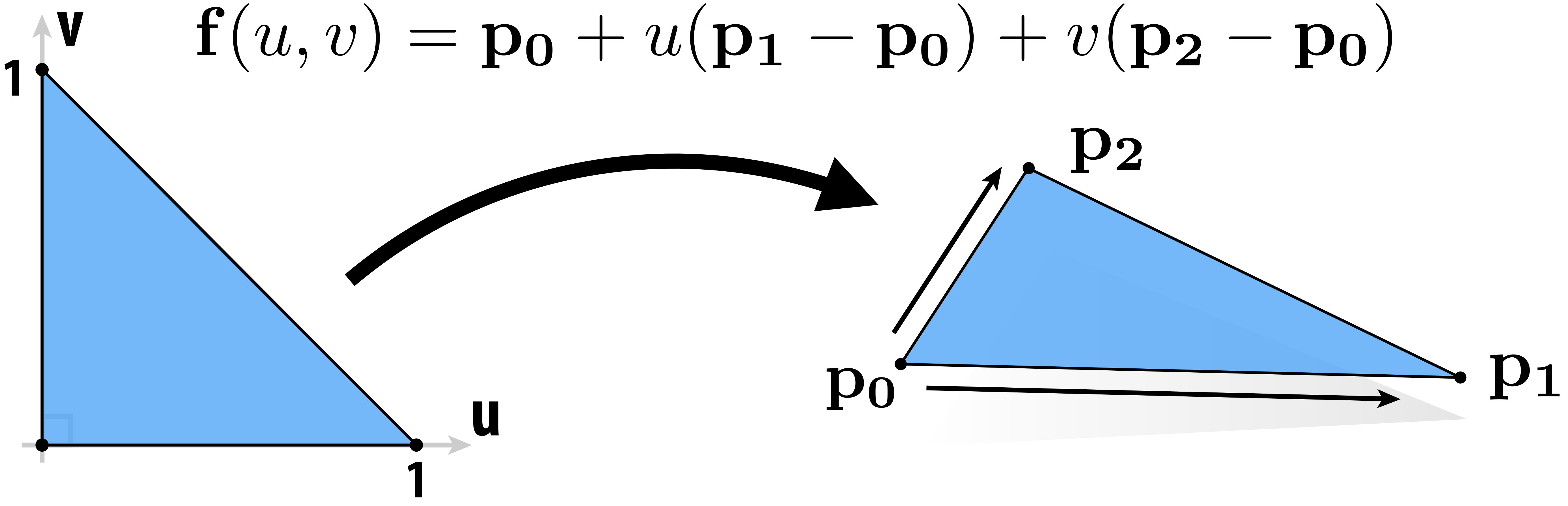
# Review: ray-triangle intersection

- **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**

$$\mathbf{f}(u,v) = \mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0})$$



# Ray-triangle intersection

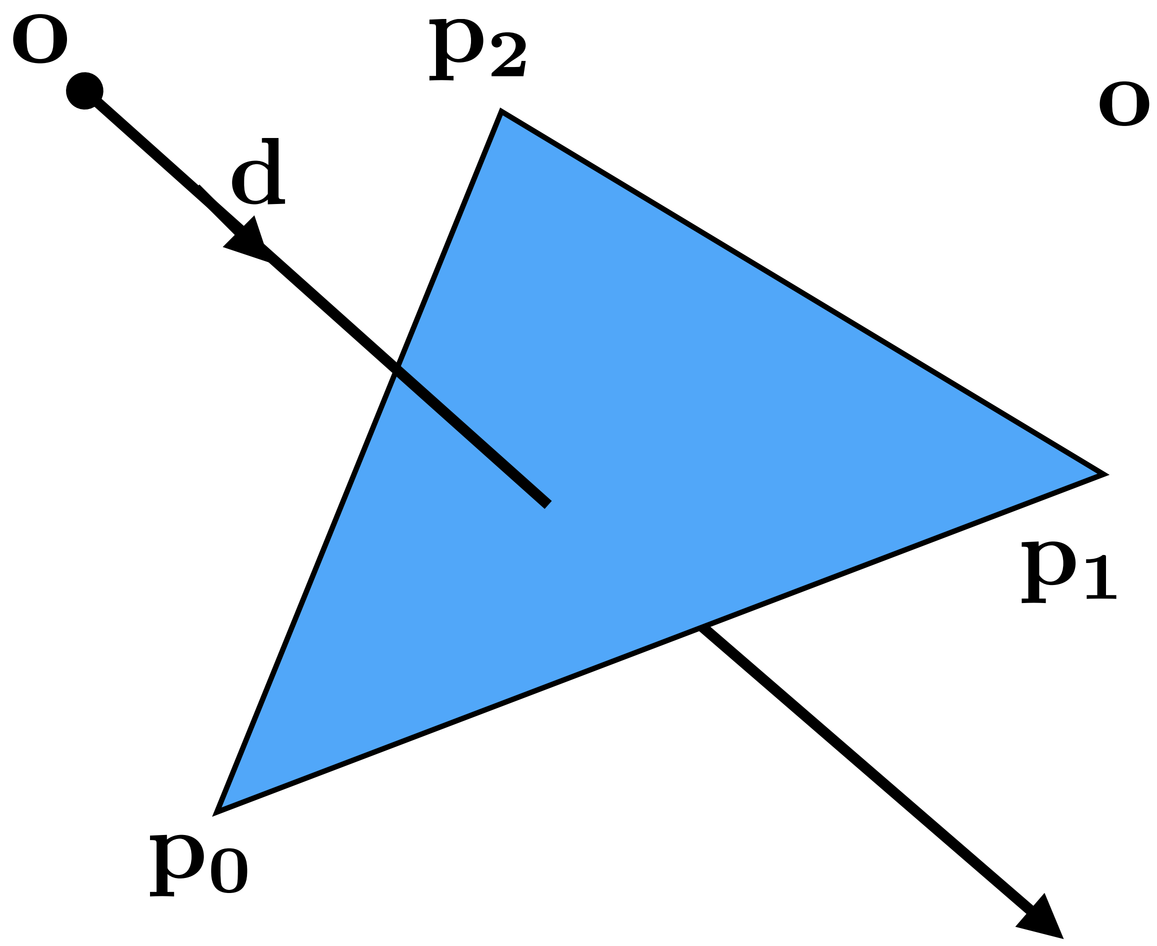
**Plug parametric ray equation directly into equation for points on triangle:**

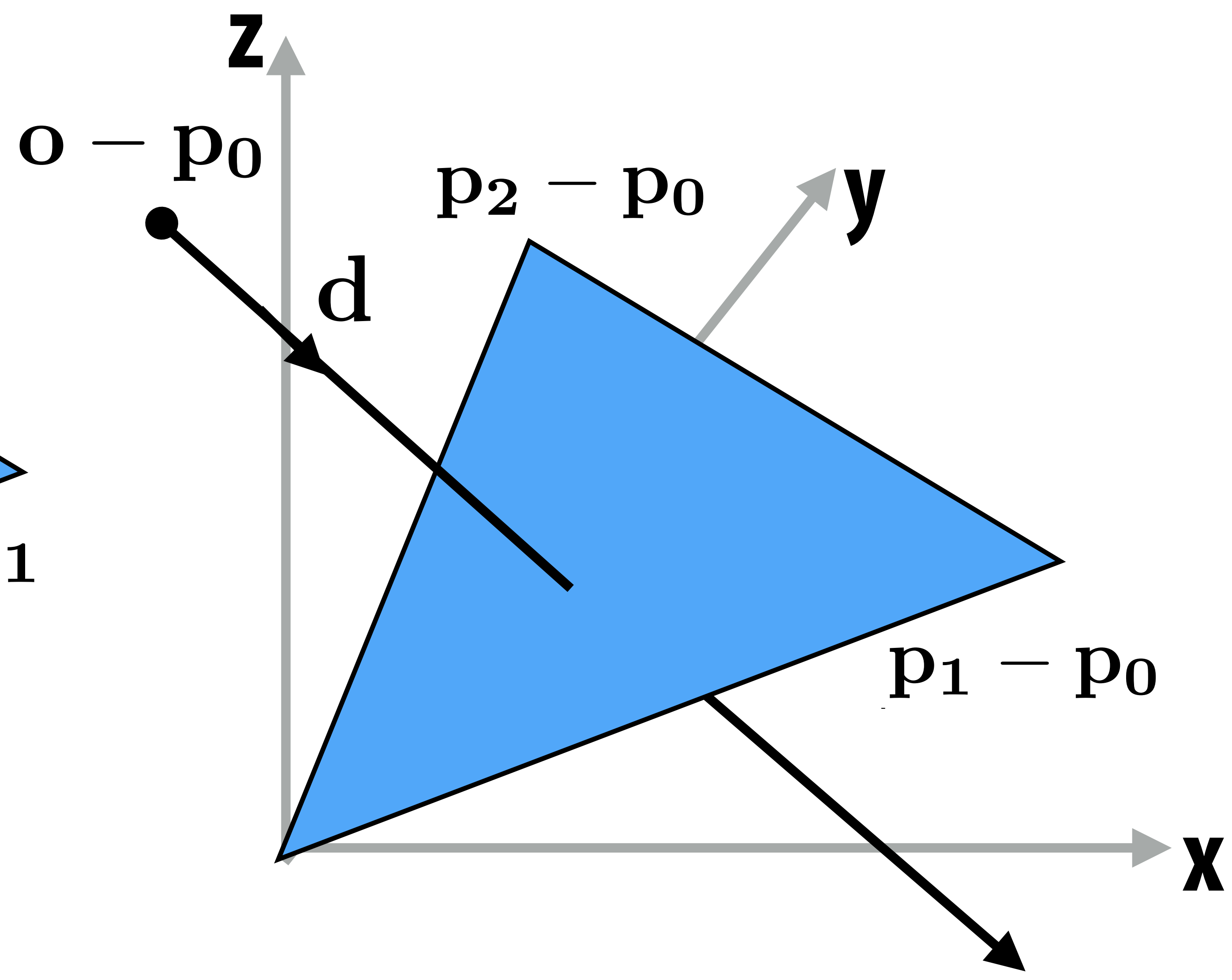
$$\mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0}) = \mathbf{o} + t\mathbf{d}$$

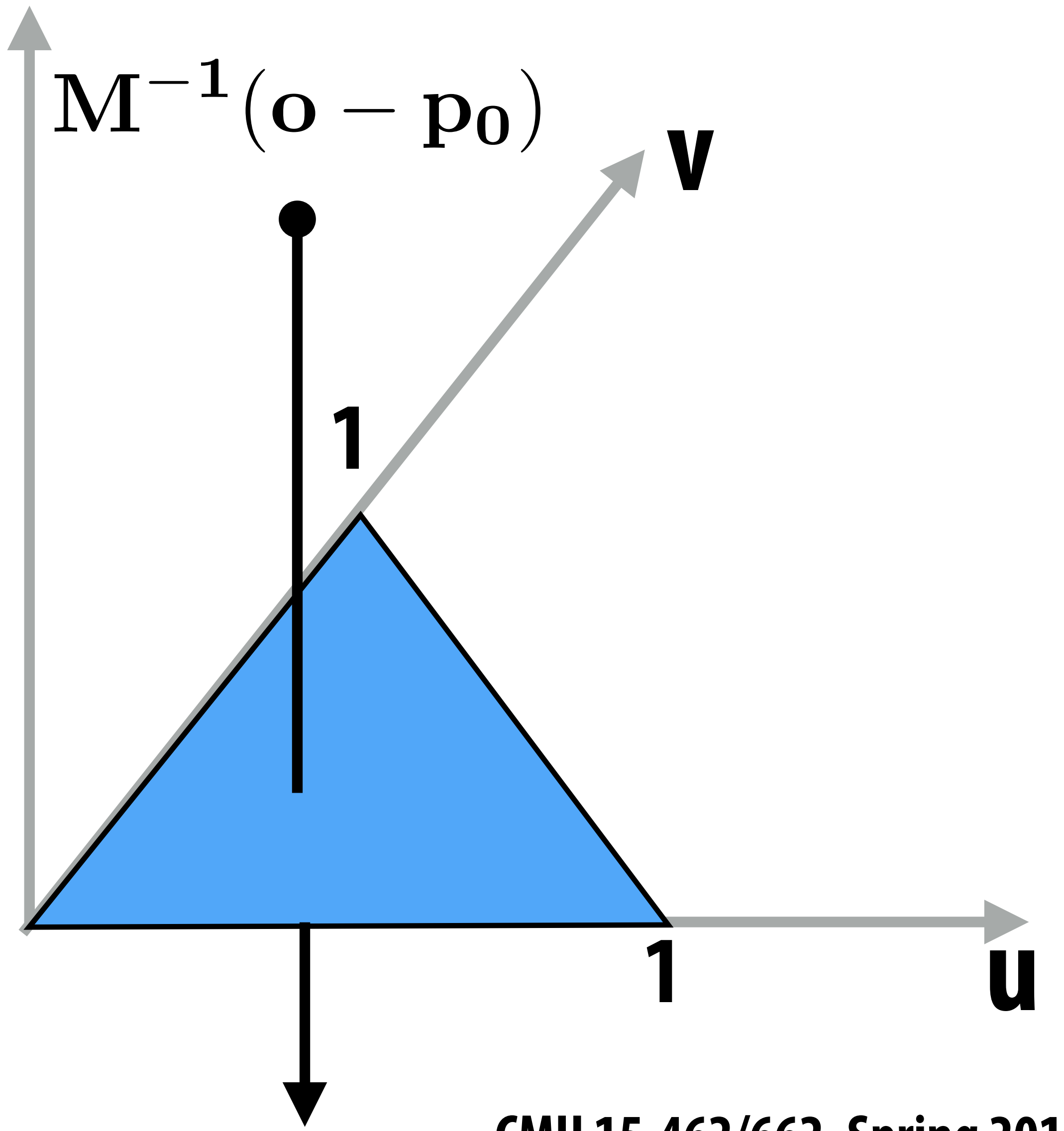
**Solve for u, v, t:**

$$\underbrace{\begin{bmatrix} \mathbf{p_1} - \mathbf{p_0} & \mathbf{p_2} - \mathbf{p_0} & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p_0}$$

$\mathbf{M}^{-1}$ **transforms triangle back to unit triangle in u,v plane, and transforms ray's direction to be orthogonal to plane**







# Ray-primitive queries

**Given primitive p:**

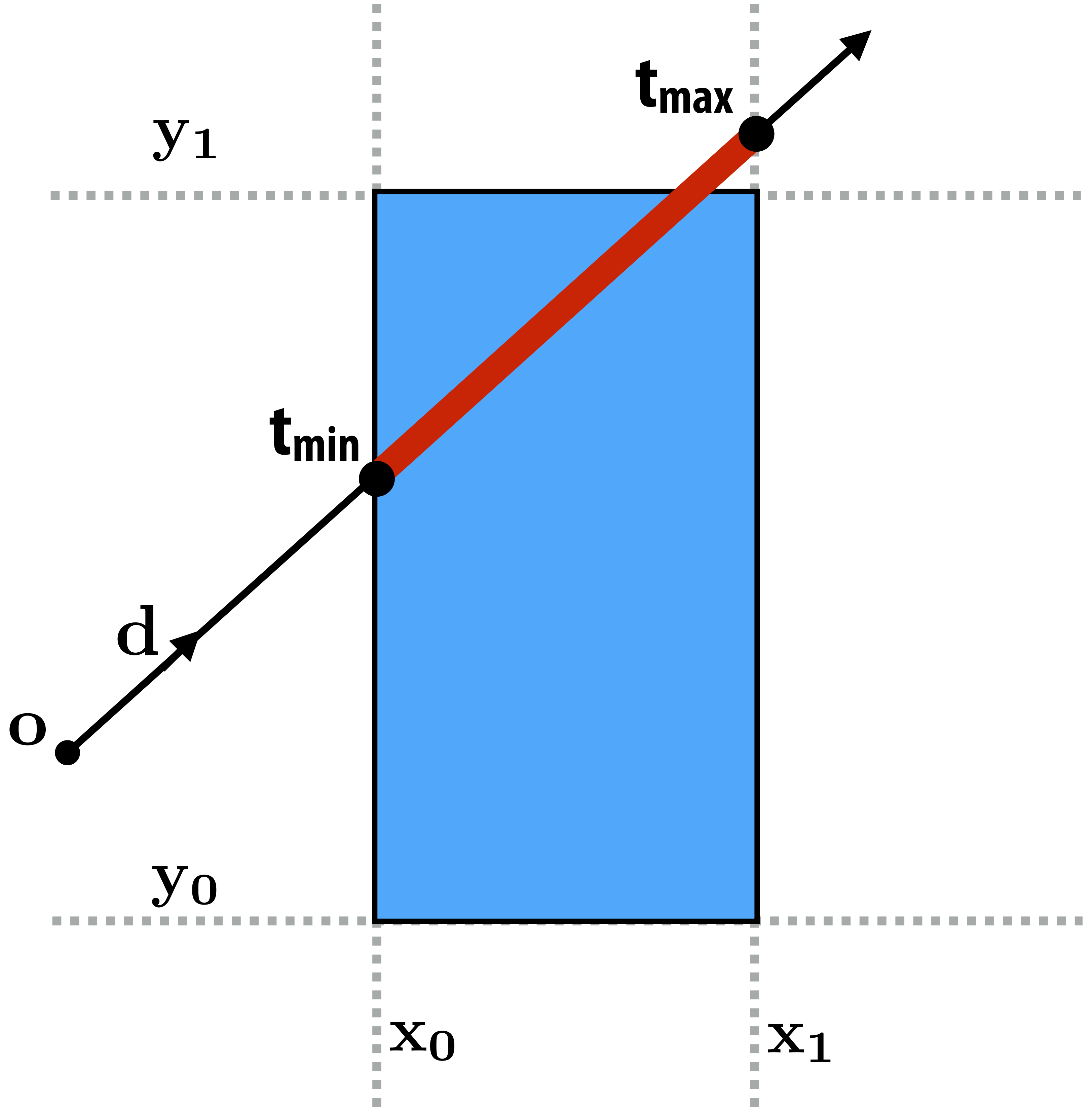
**p.intersect(r) returns value of t corresponding to the point of intersection with ray r**

**p.bbox() returns axis-aligned bounding box of the primitive**

```
tri.bbox():
   tri_min = min(p0, min(p1, p2))
   tri_max = max(p0, max(p1, p2))
   return bbox(tri_min, tri_max)
```

# 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:**

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

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

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

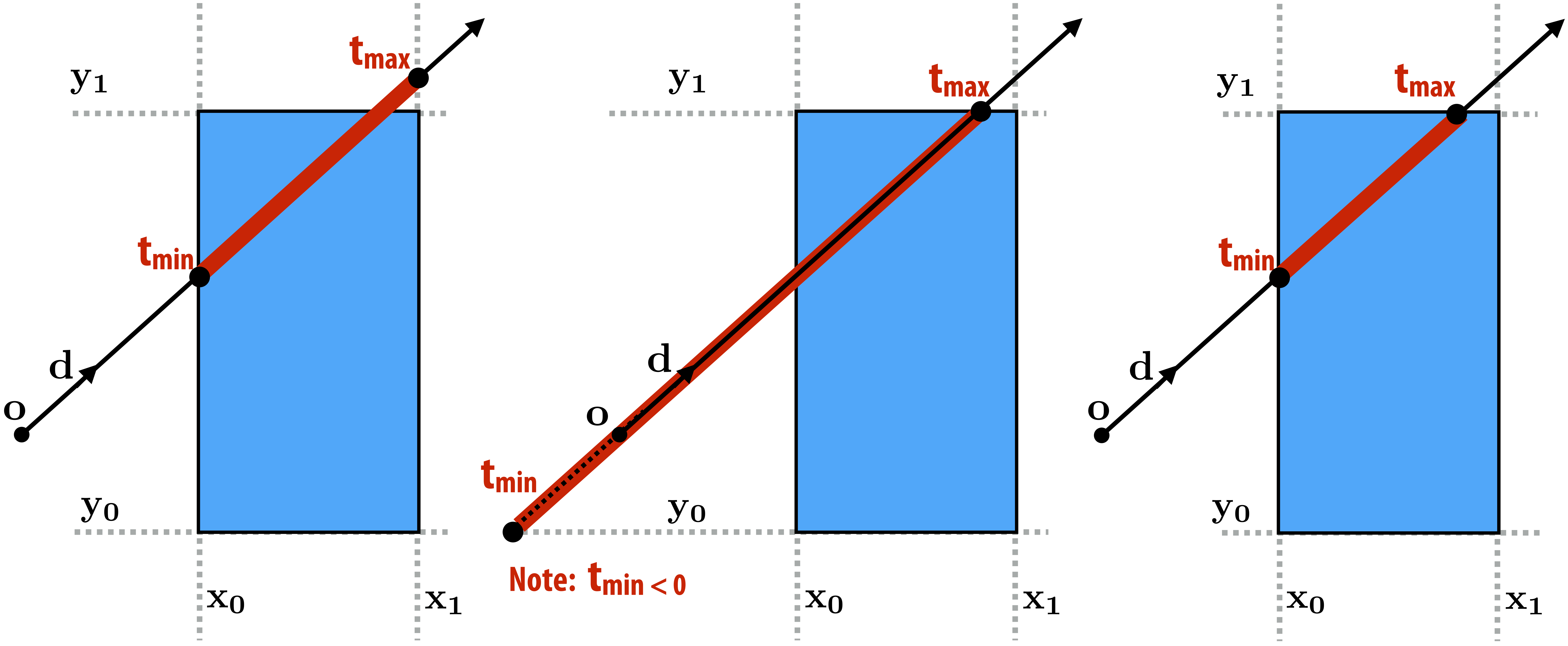
$$c = x_0$$

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

**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**



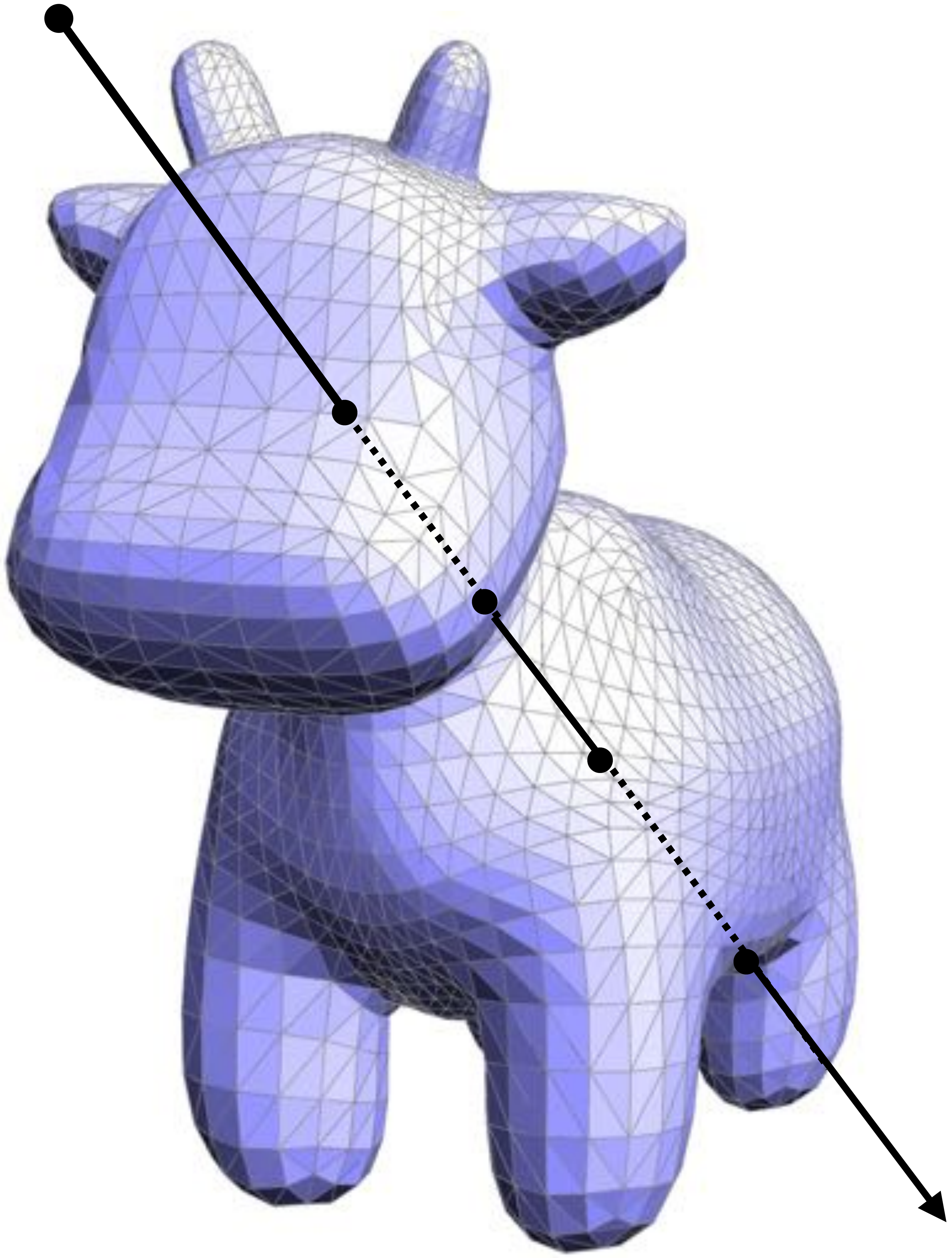
**Intersections with *x* planes** **Intersections with *y* planes** **Final intersection result**

**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"**



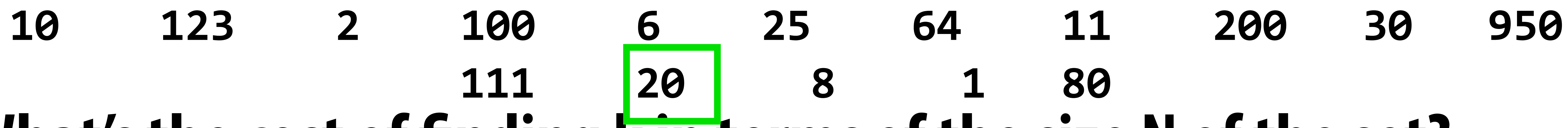
```
p_closest = NULL
t_closest = inf
for each primitive p in scene:
    t = p.intersect(r)
    if t >= 0 && t < t_closest:
        t_closest = t
        p_closest = p
```

**Complexity?** $O(N)$

**Can we do better?**

# A simpler problem

- Imagine I have a set of integers S
- Given an integer, say k=18, find the element of S closest to k:

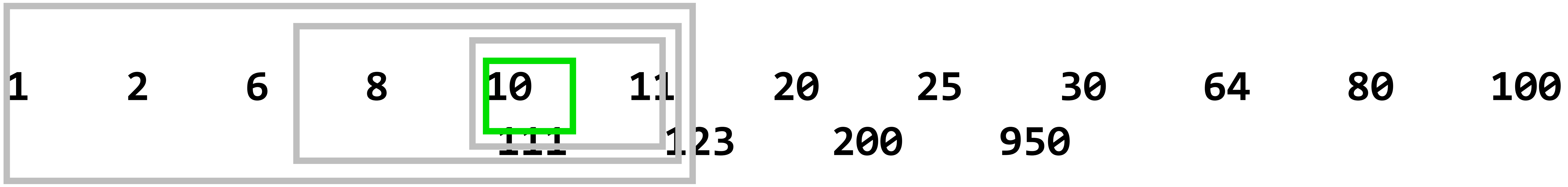


```
10    123    2    100    6    25    64    11    200    30    950
                  111    20      8      1    80
```

What's the cost of finding k in terms of the size N of the set?

Can we do better?

Suppose we first sort the integers:



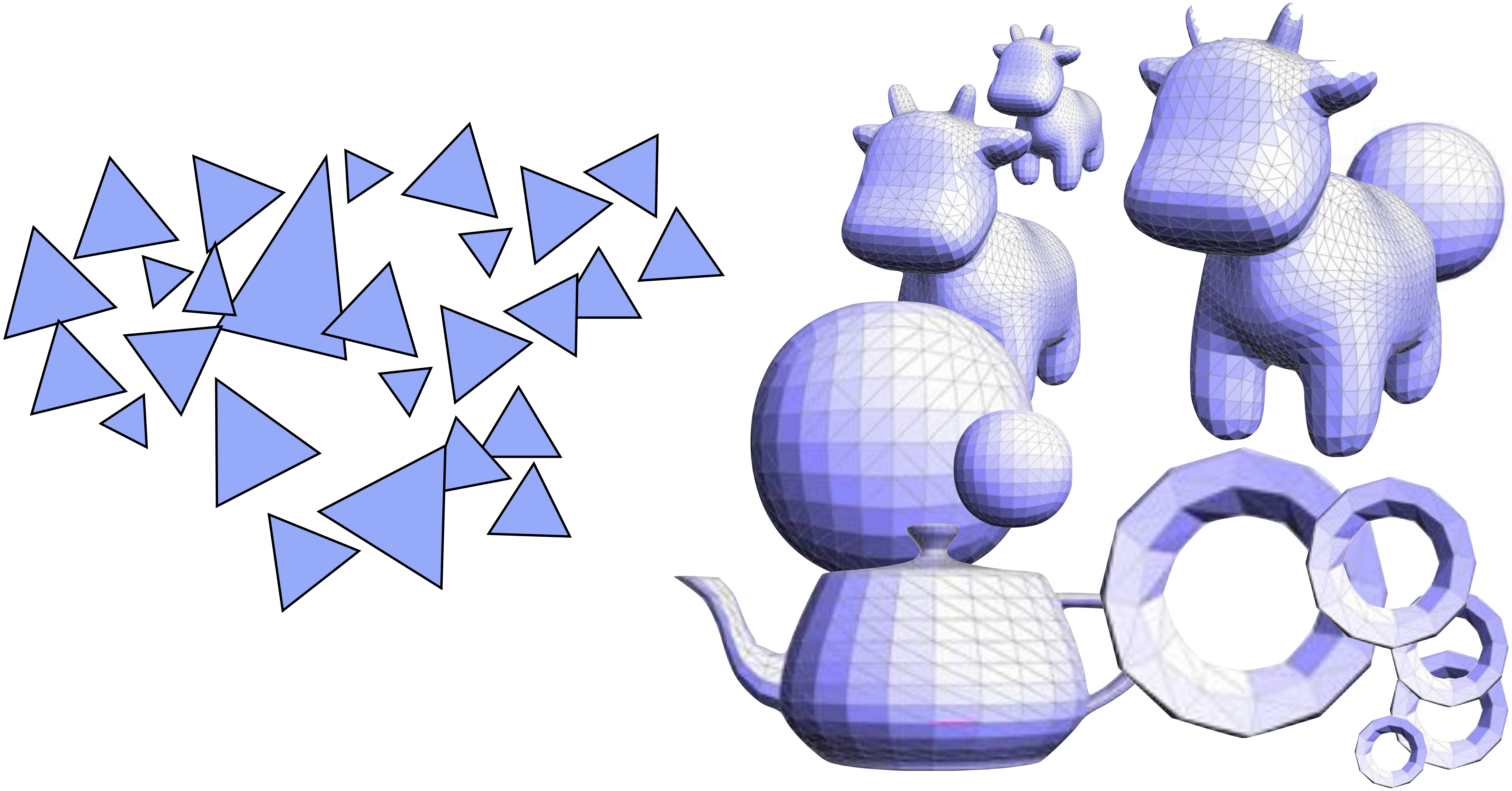
```
1    2    6    8    10    11    20    25    30    64    80    100
                      111      123     200     950
```

How much does it now cost to find k (including sorting)?

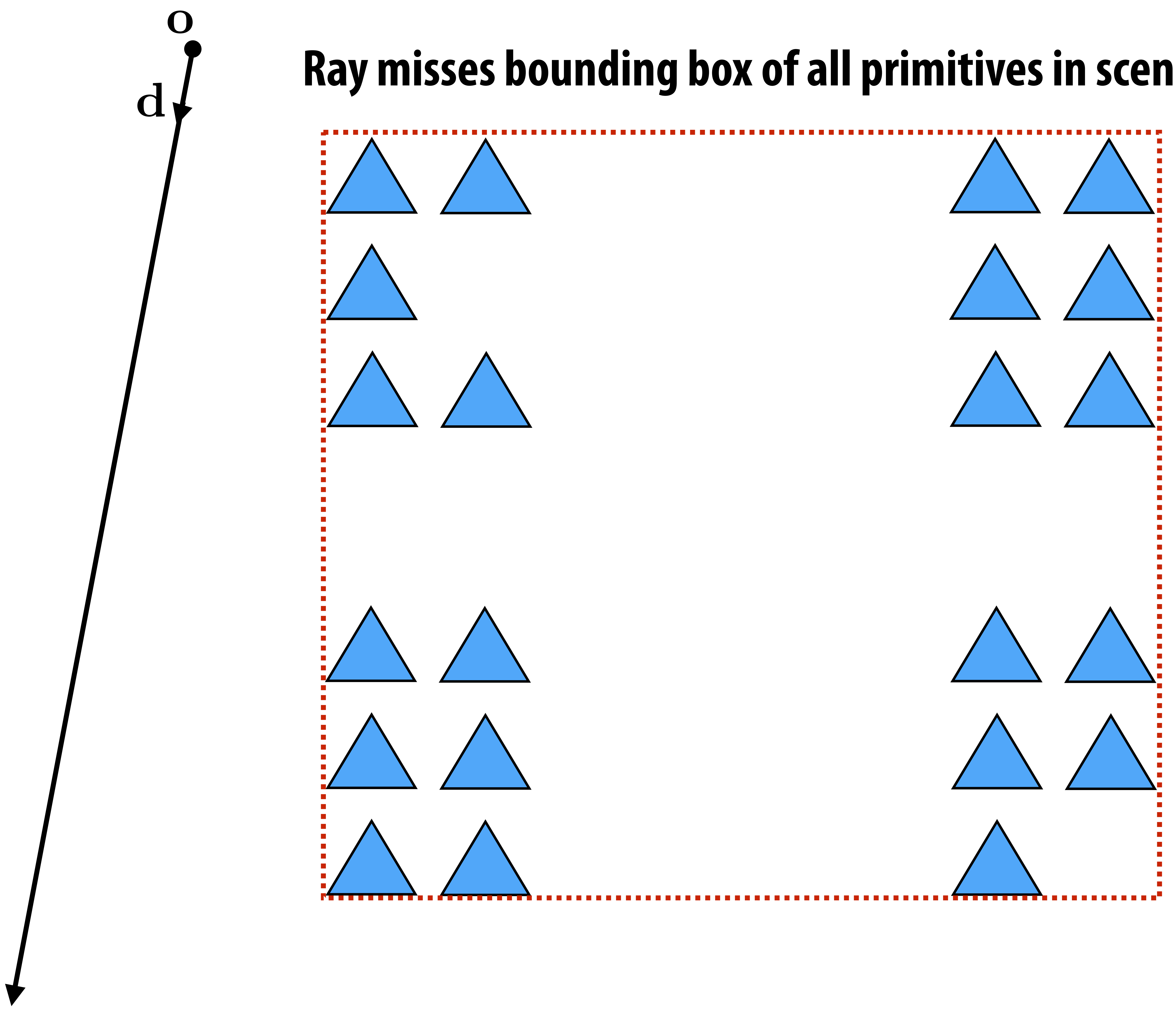
Cost for just ONE query: O(n log n) worse than before! :-)

Amortized cost: O(log n) ...much better!

# Can we also reorganize scene primitives to enable fast ray-scene intersection queries?



# Simple case



Ray misses bounding box of all primitives in scene

**Cost (misses box):**

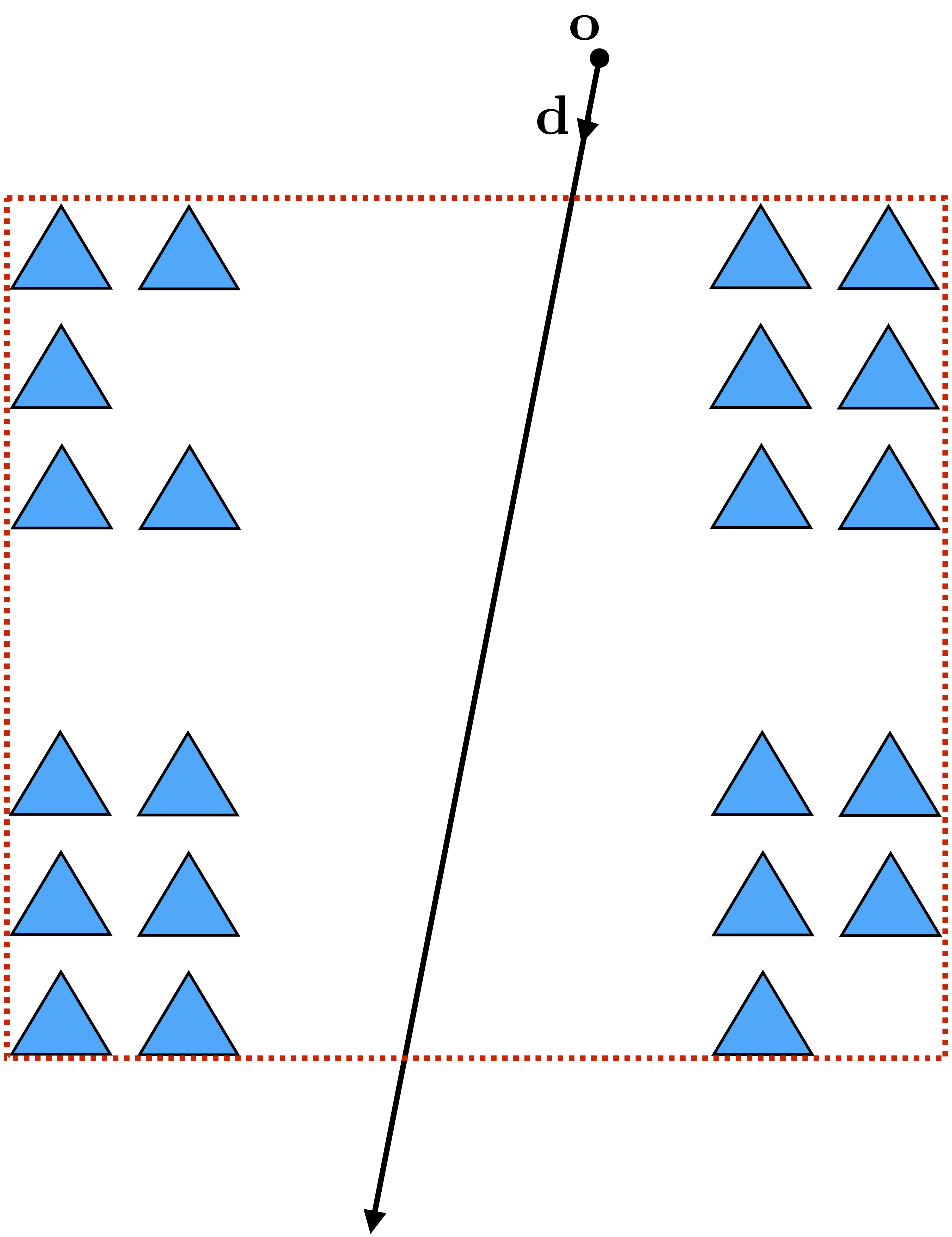
preprocessing: O(n)

ray-box test: O(1)

amortized cost*: O(1)

*over many ray-scene intersection tests

# Another (should be) simple case



**Cost (hits box):**

preprocessing: O(n)

ray-box test: O(1)

triangle tests: O(n)

amortized cost*: O(n)

**Still no better than naïve algorithm (test all triangles)!**

*over many ray-scene intersection tests

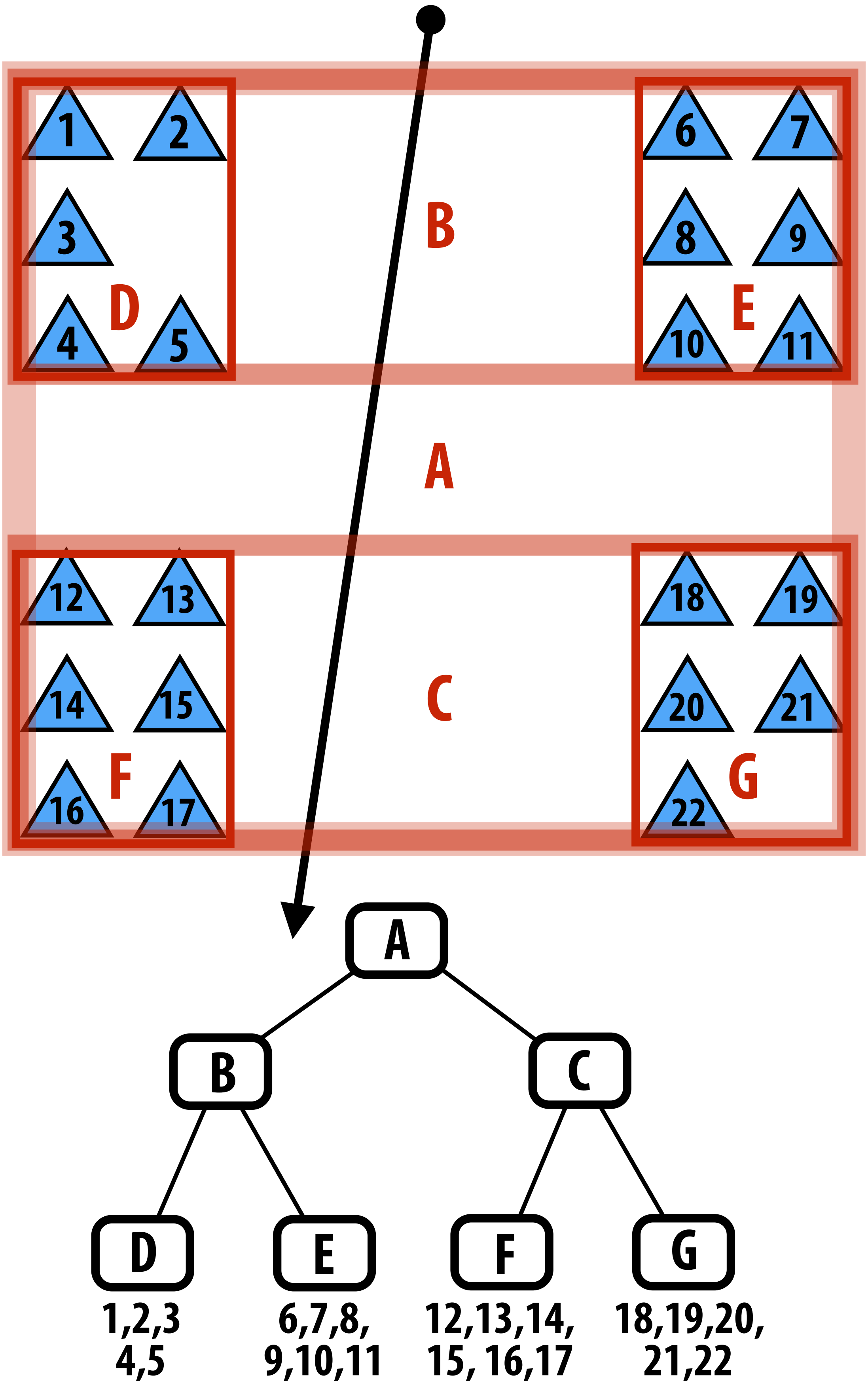
**Q: How can we do better?**

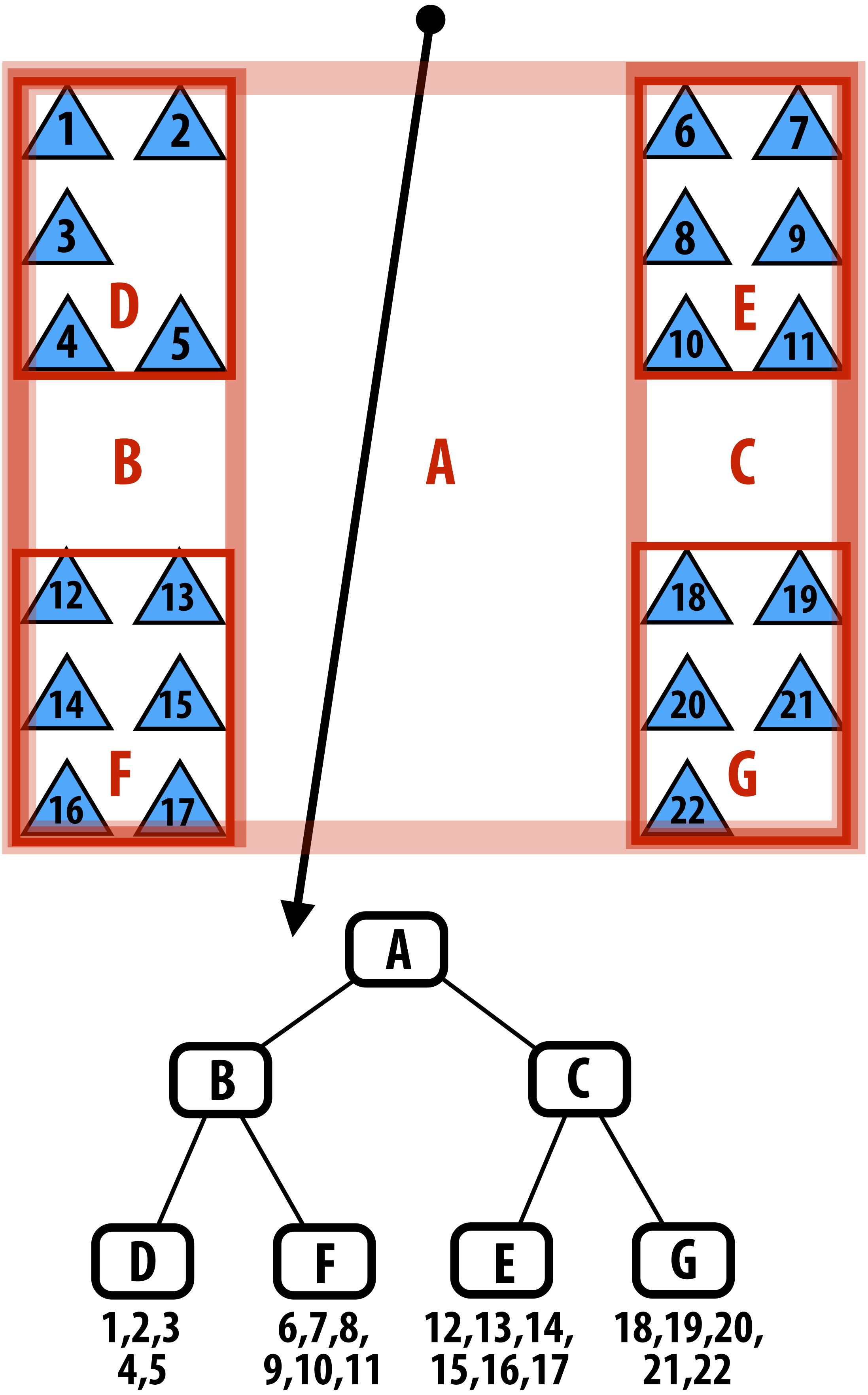
~~**A: Use deep learning.**~~

**A: Apply this strategy hierarchically.**

# Bounding volume hierarchy (BVH)

- **Leaf nodes:**
  - Contain small list of primitives
- **Interior nodes:**
  - Proxy for a large subset of primitives
  - Stores bounding box for all primitives in subtree



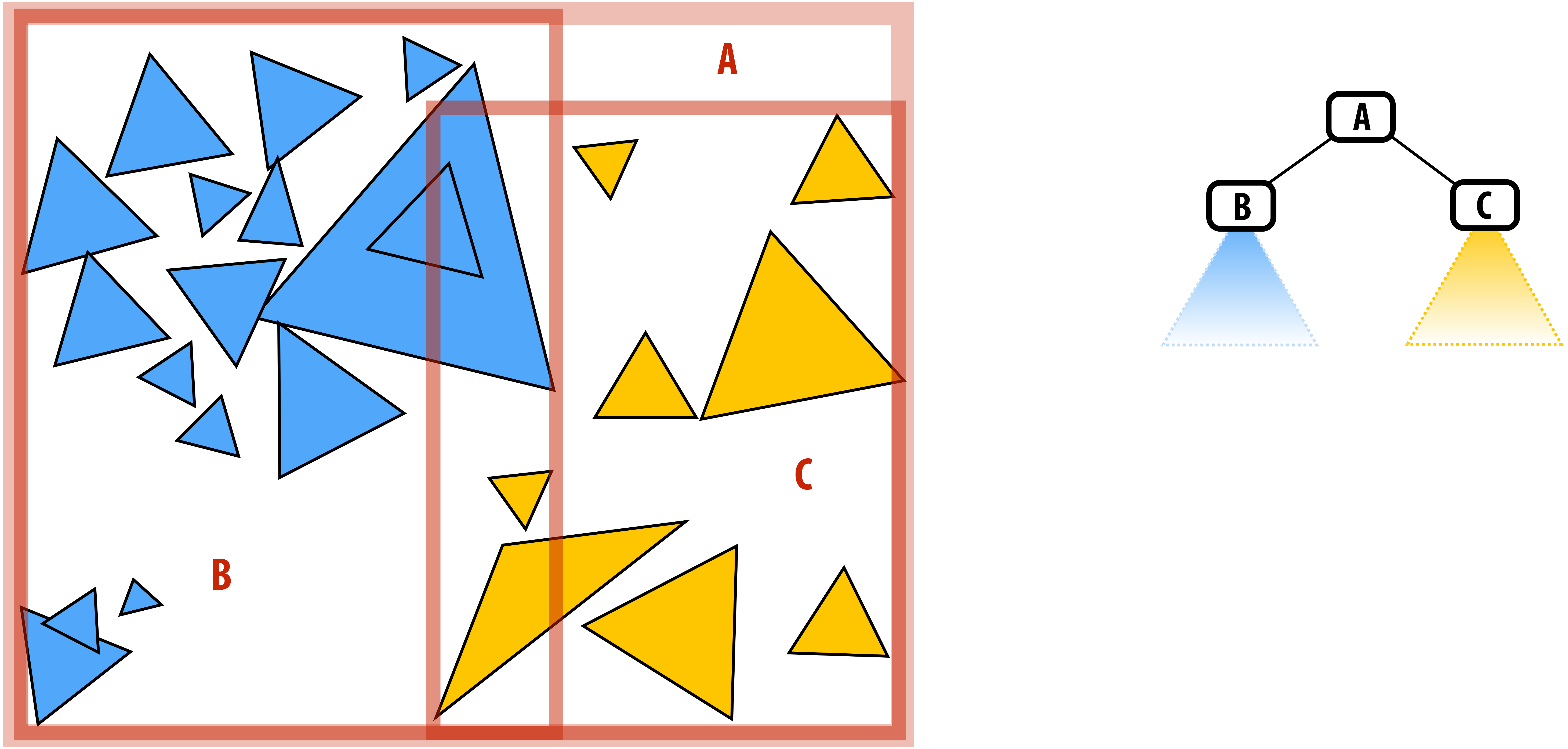


Left: two different BVH organizations of the same scene containing 22 primitives.

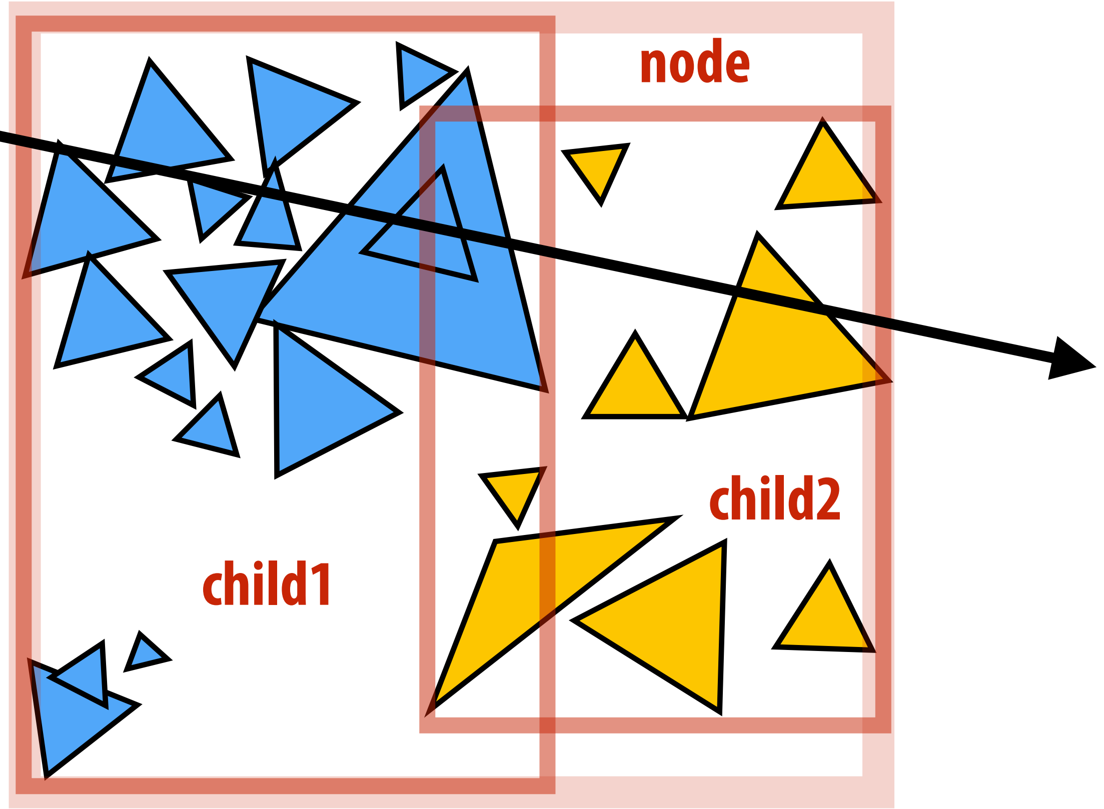
Is one BVH better than the other?

# 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)



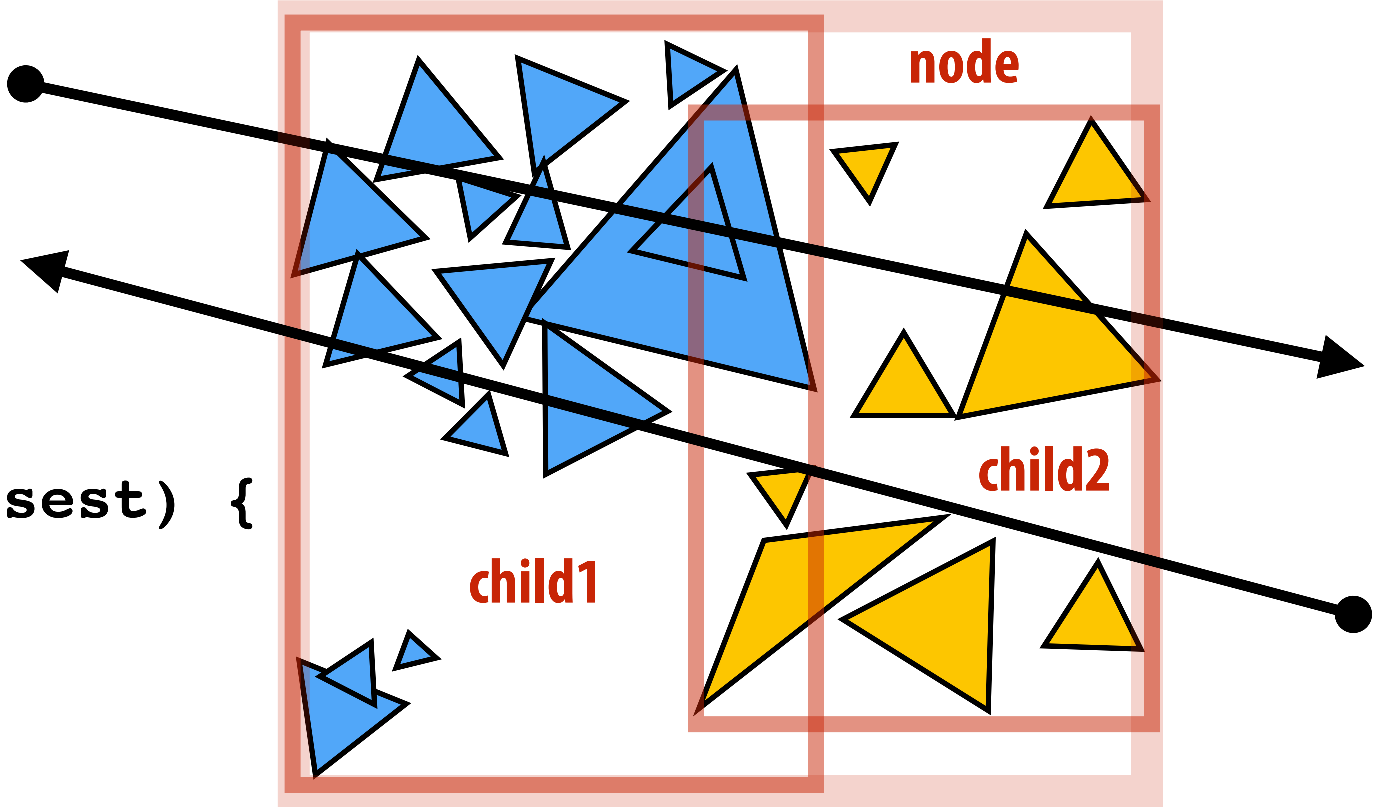
# Ray-scene intersection using a BVH



```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // “left” child (could be NULL)
   BVHNode* child2; // “right” child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;   // at what t value?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node’s bounding box
   if (hit.prim == NULL || hit.t > closest.t))
      return; // don’t update the hit record

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && hit.t < closest.t) {
            closest.prim = p;
            closest.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, HitInfo* closest) {

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit && t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      HitInfo hit1 = intersect(ray, node->child1->bbox);
      HitInfo hit2 = intersect(ray, node->child2->bbox);

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit2.t <= hit1.t) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (second child’s t is closer than closest.t)
         find_closest_hit(ray, second, closest); // why might we still need to do this?
   }
}
```

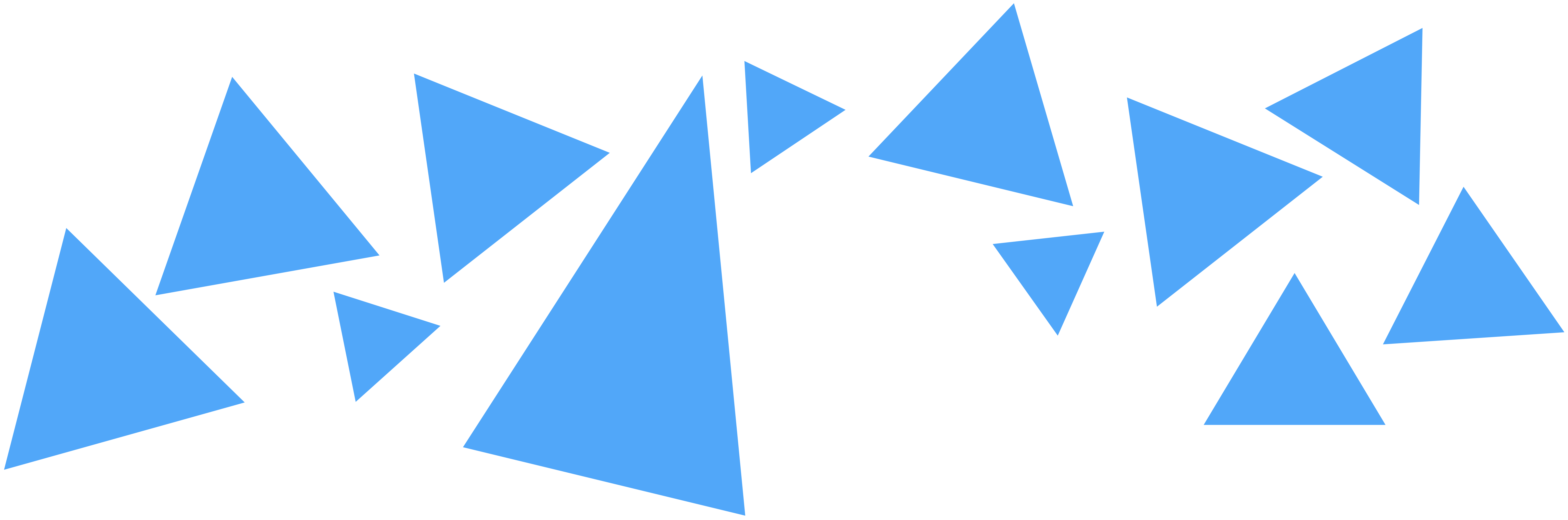
“Front to back” traversal. Traverse to closest child node first. Why?

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

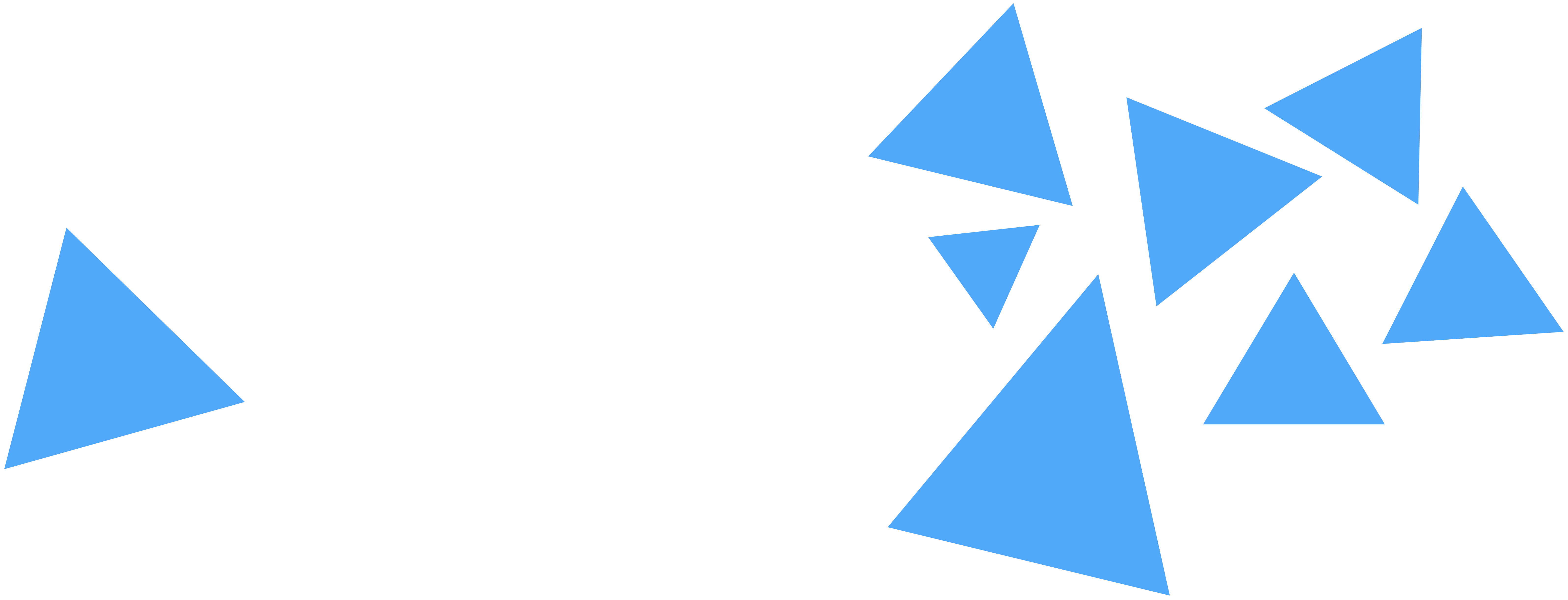
**($2^N/2$ ways to partition N primitives into two groups)**

## Q: How do we build a high-quality BVH?

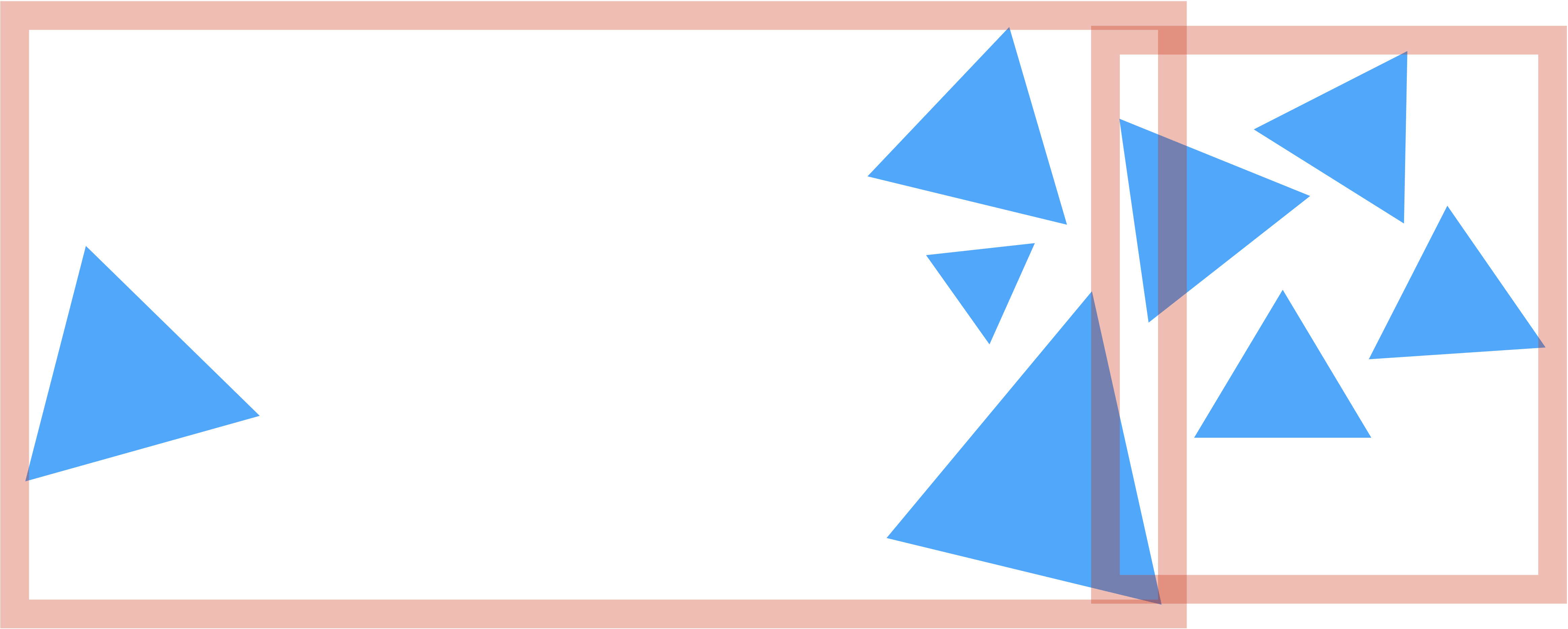
# How would you partition these triangles into two groups?



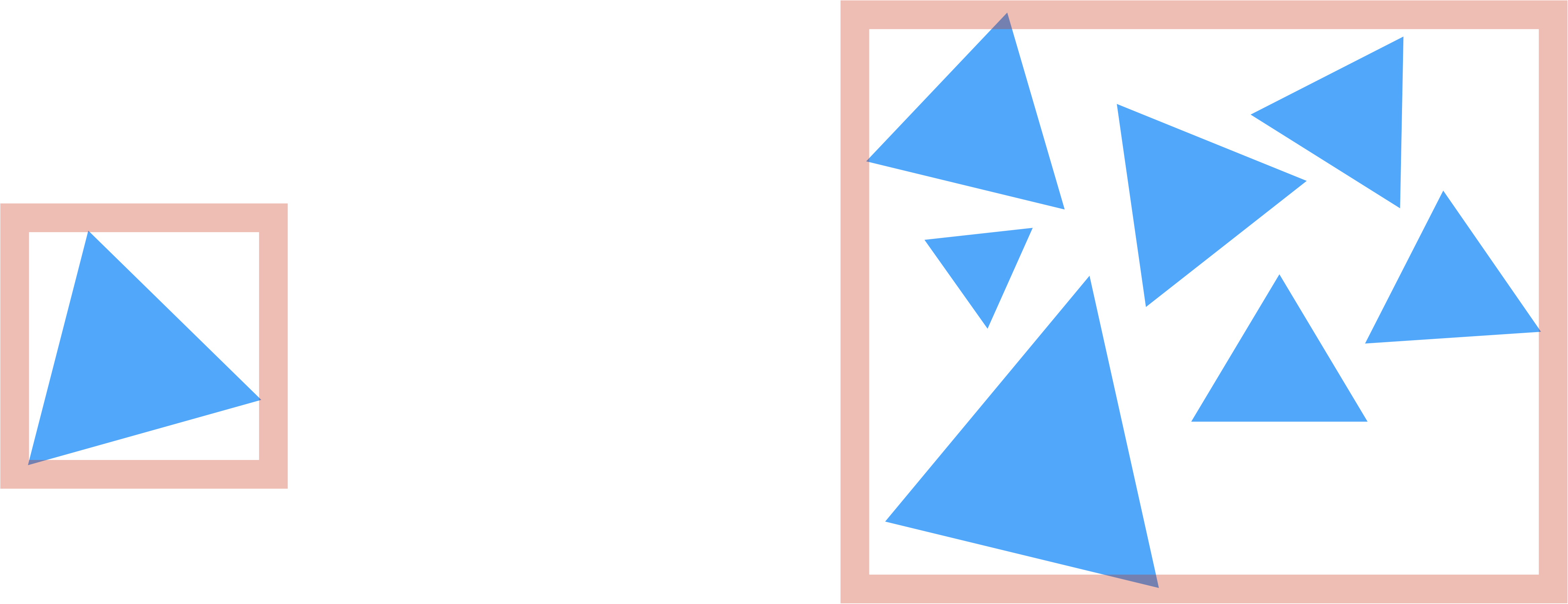
# What about these?



# 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 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.**

$$= NC_{\text{isect}}$$

**(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:**

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

$C_{\text{trav}}$ **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 resultant child subtrees**

$p_A$ **and** $p_B$ **are the probability a ray intersects the bbox of the child nodes A and B**

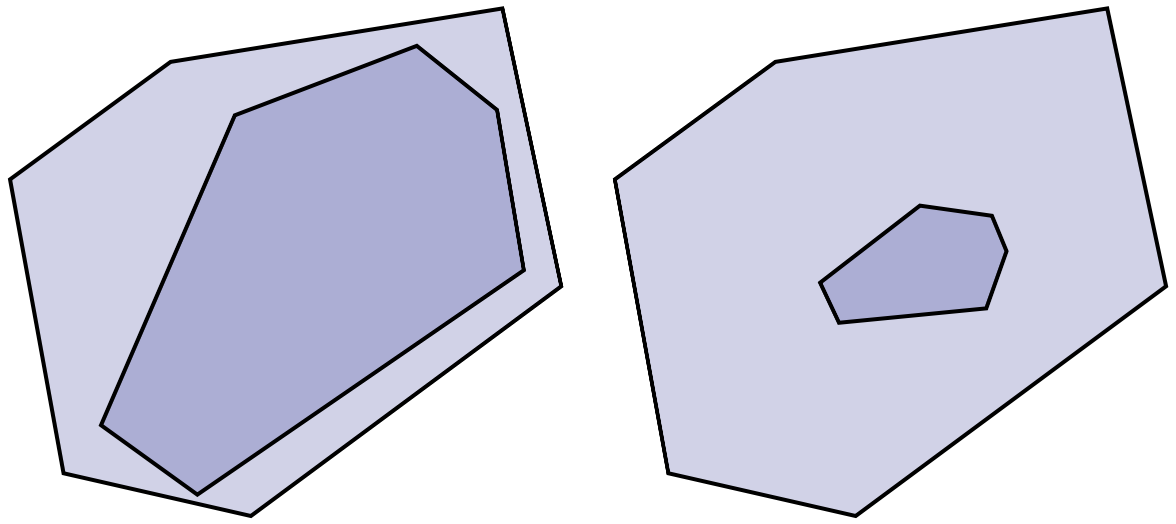
**Primitive count is common approximation for child node costs:**

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

**Remaining question: how do we get the probabilities $p_A$, $p_B$?**

# 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.**



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

**Leads to surface area heuristic (SAH):**

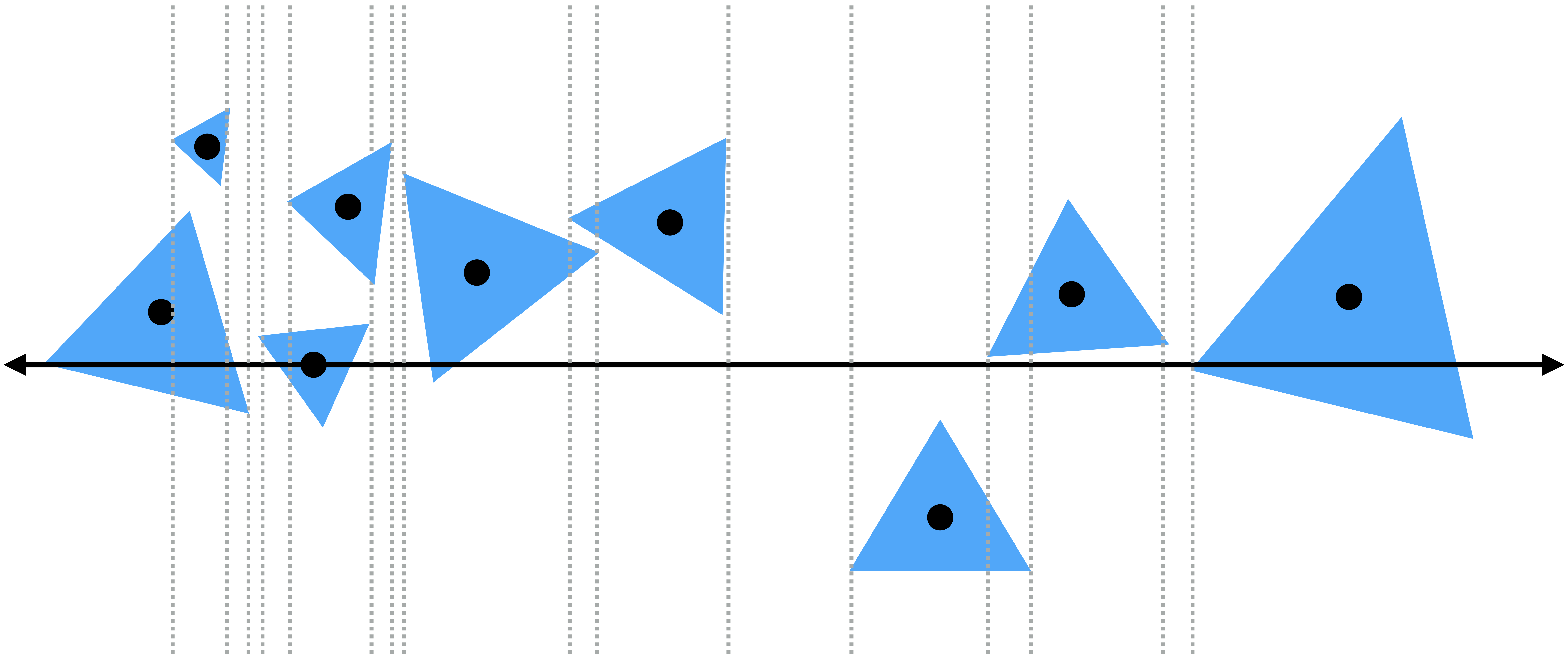
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

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

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

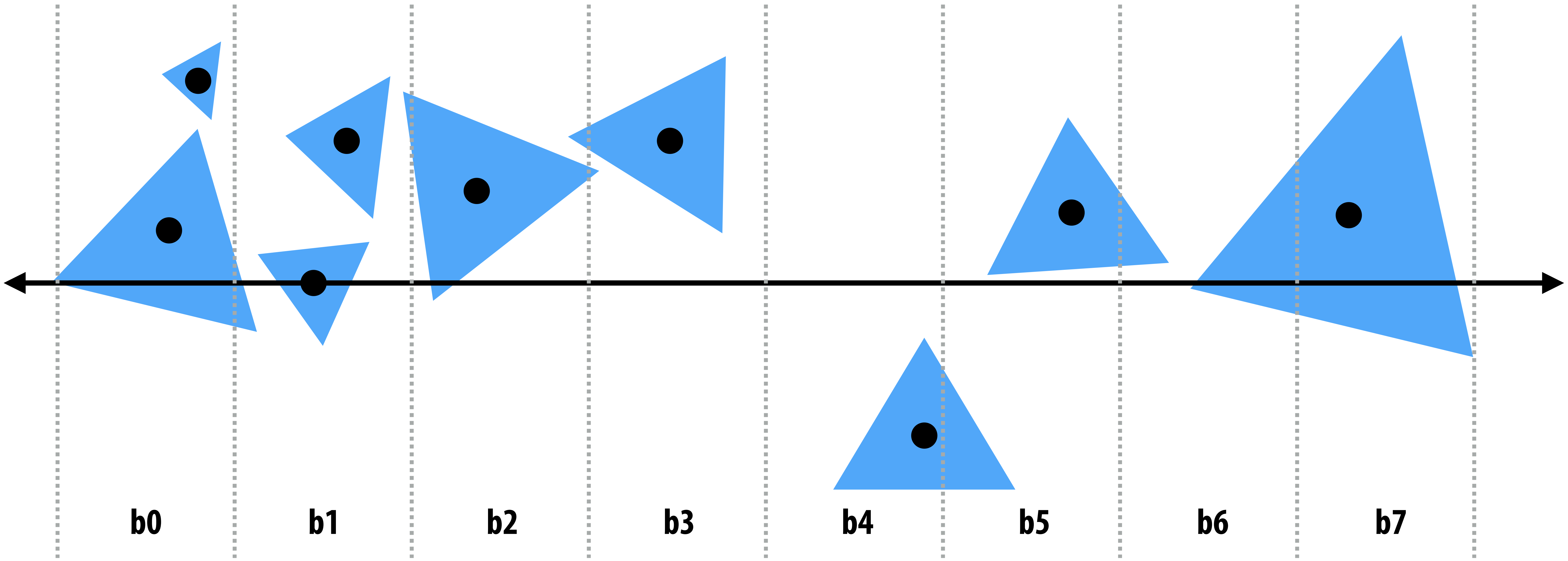
# Implementing partitions

- 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
  - SAH changes only when split plane moves past triangle boundary
  - Have to consider rather large number of possible split planes…



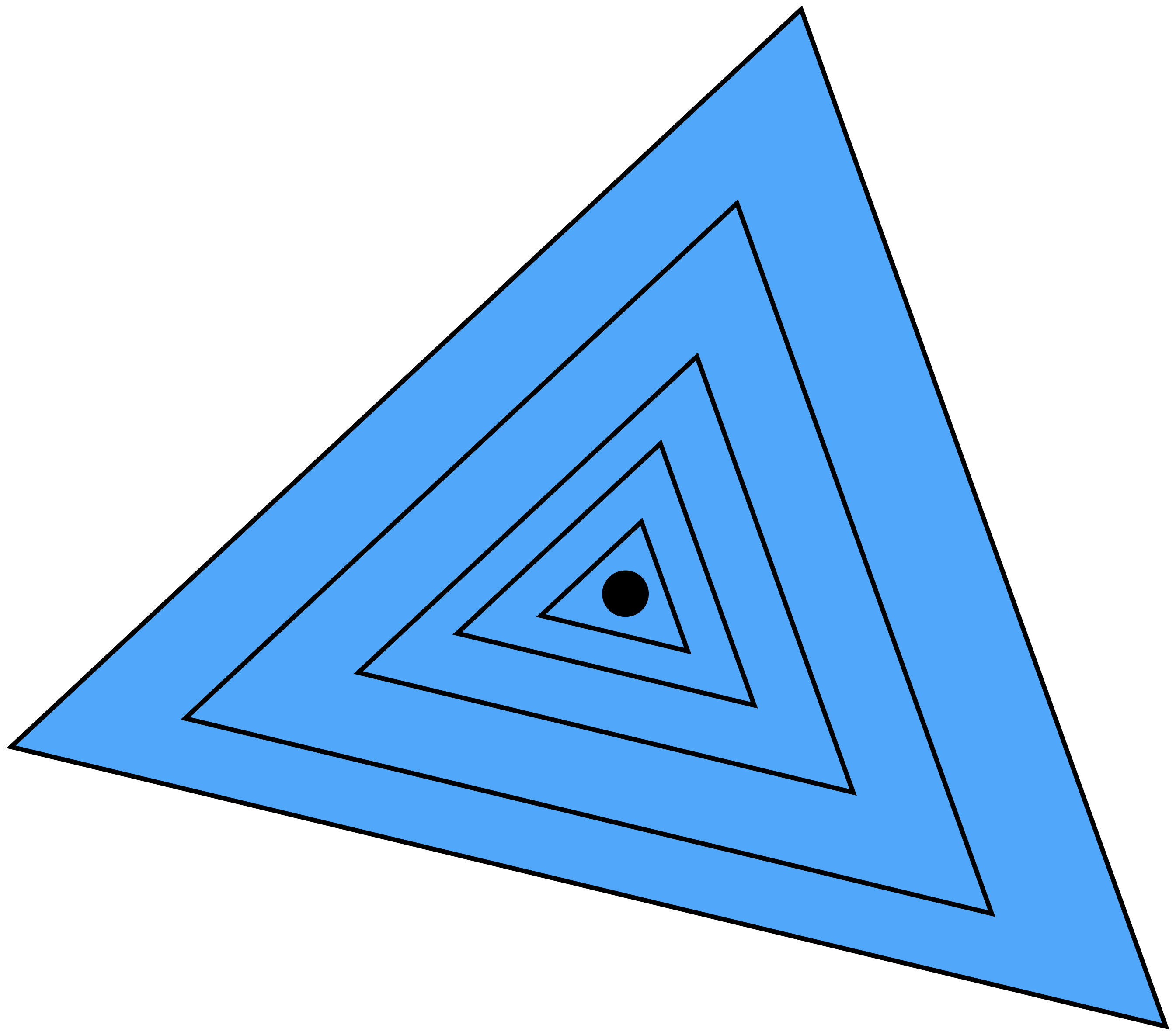
# Efficiently implementing partitioning

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

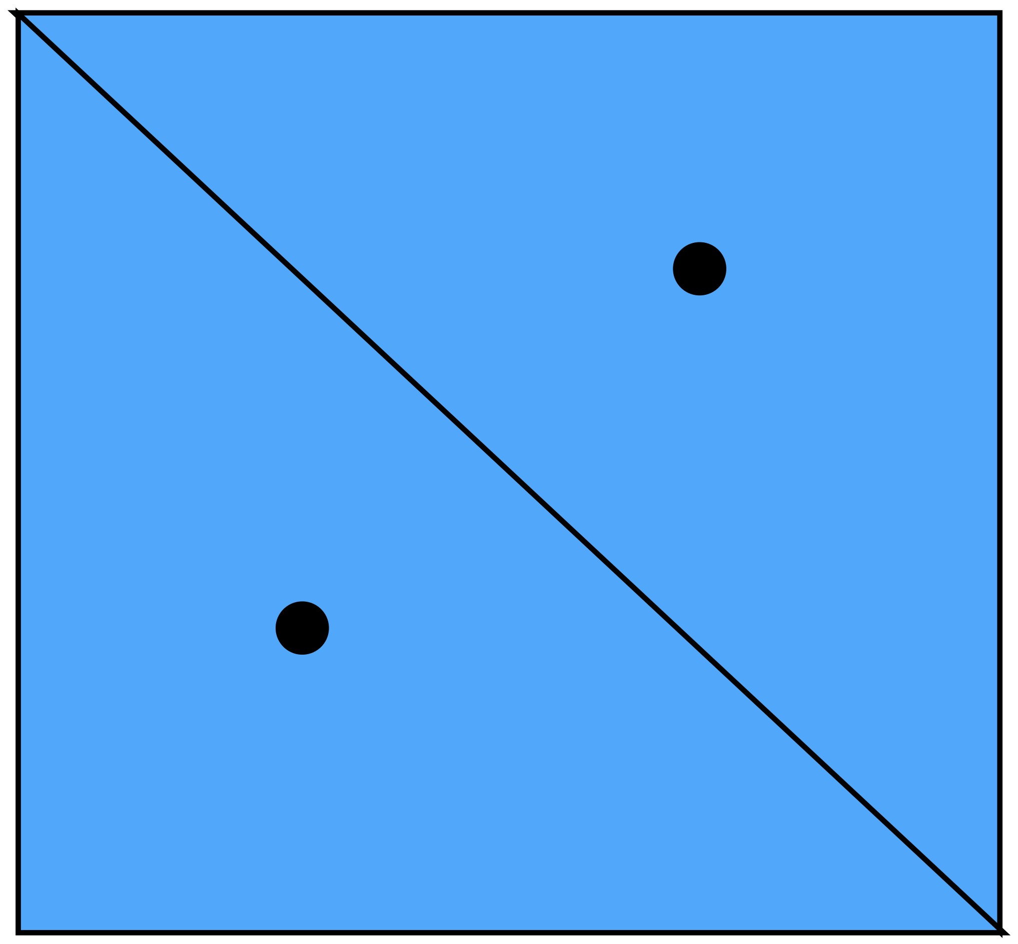


```
For each axis: x,y,z:
   initialize buckets
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Recurse on lowest cost partition found (or make node a leaf)
```

# Troublesome cases



All primitives with same centroid (all primitives end up in same partition)

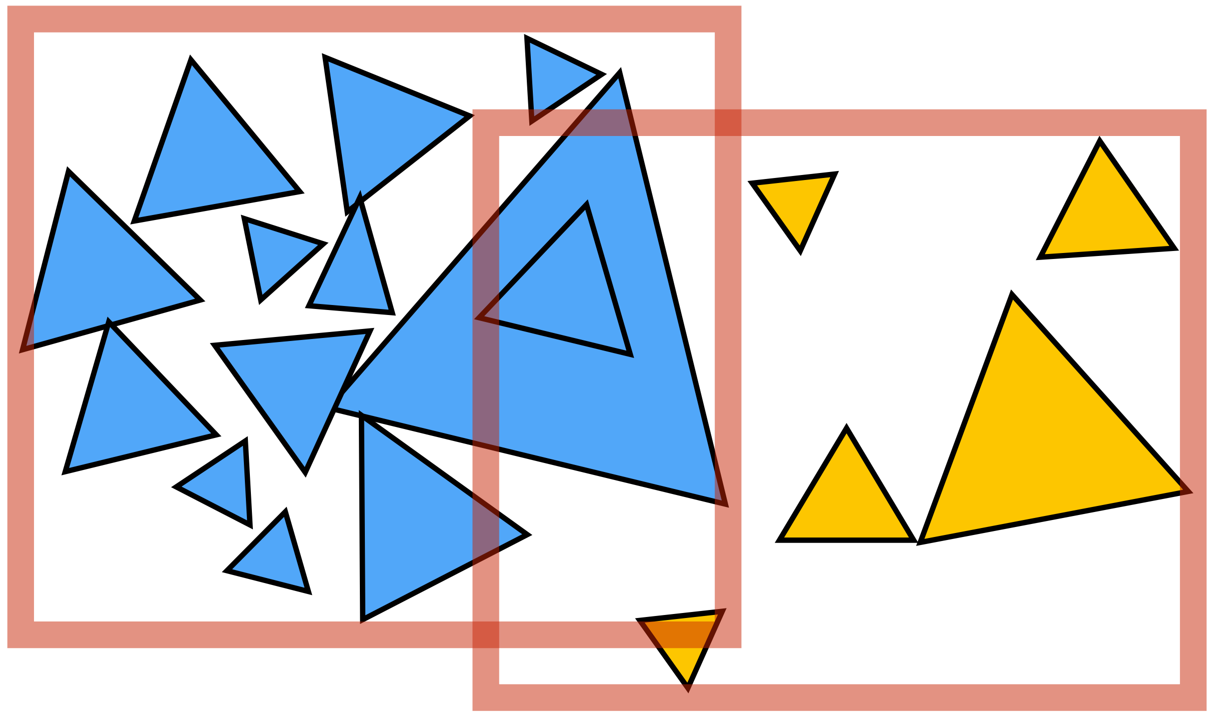


All primitives with same bbox (ray often ends up visiting both partitions)

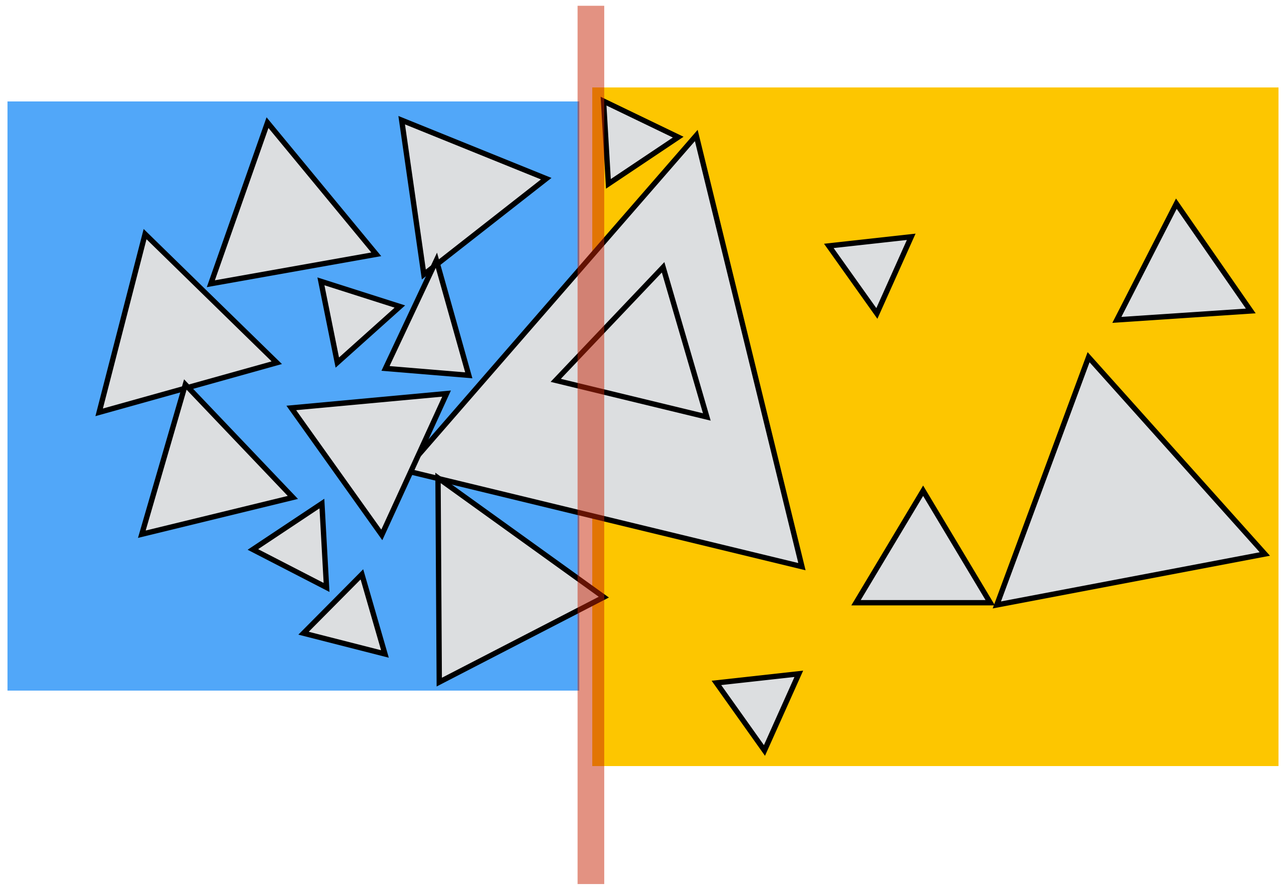
In general, different strategies may work better for different types of geometry / different distributions of primitives…

# 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)

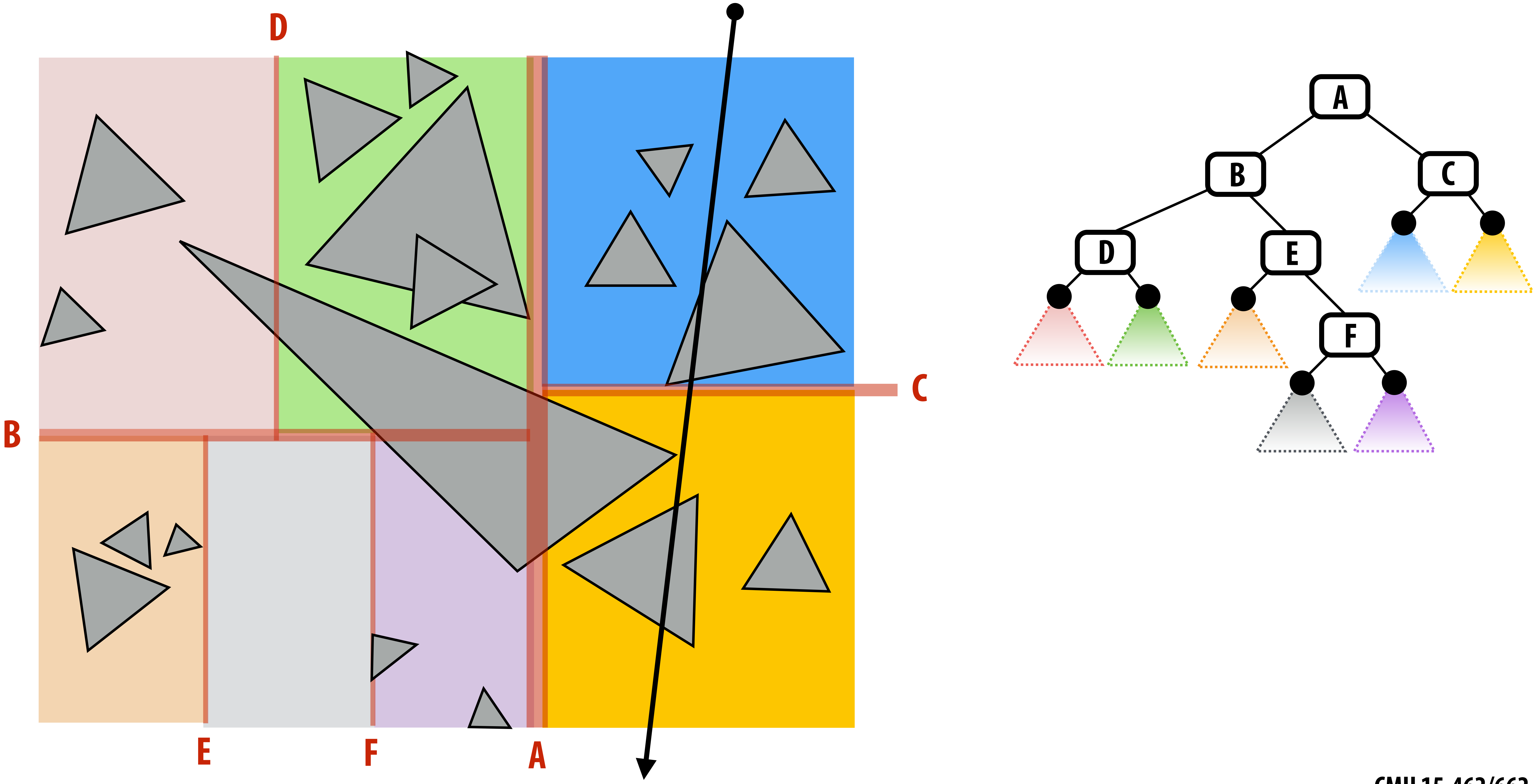


- Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



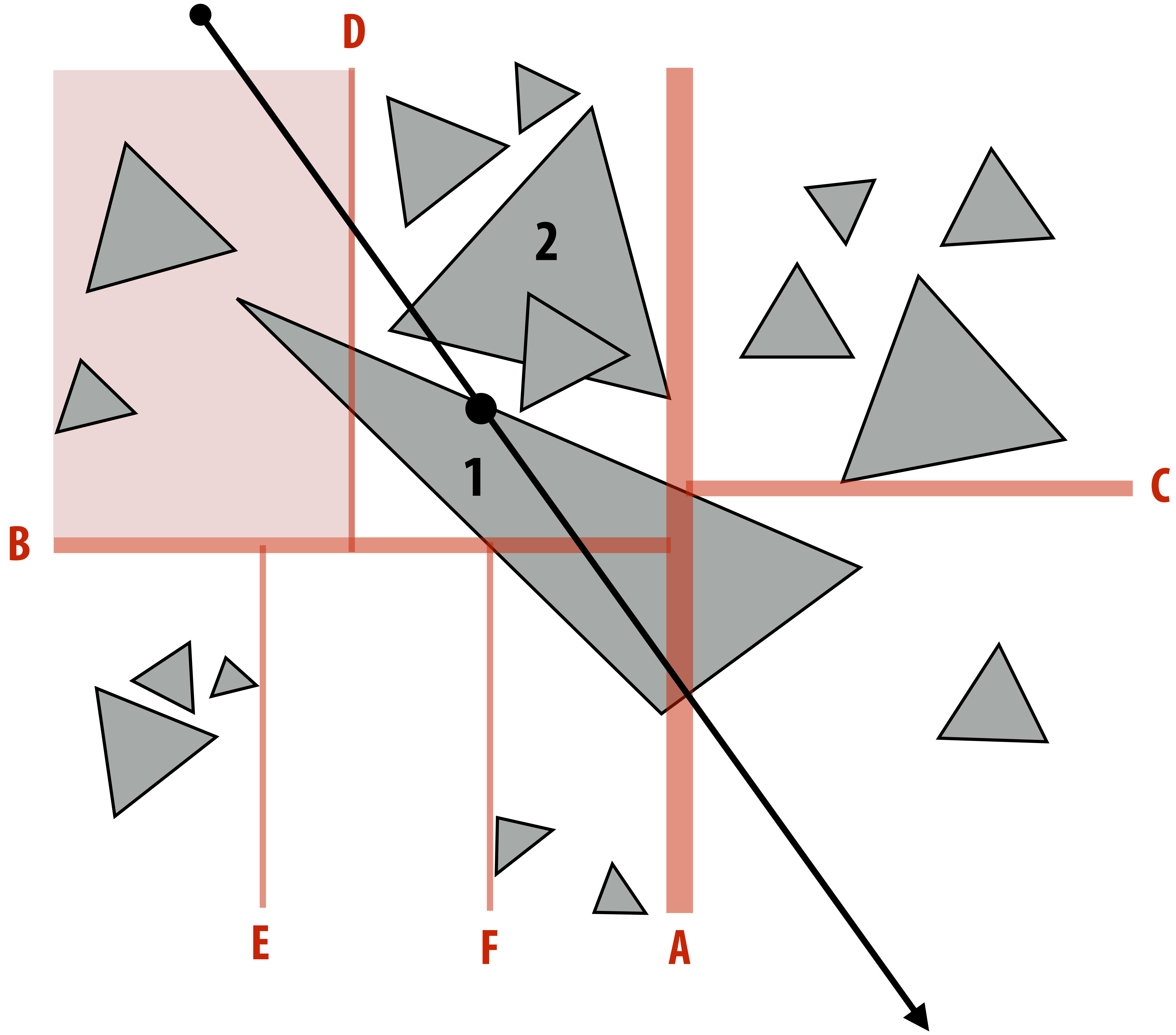
# K-D tree

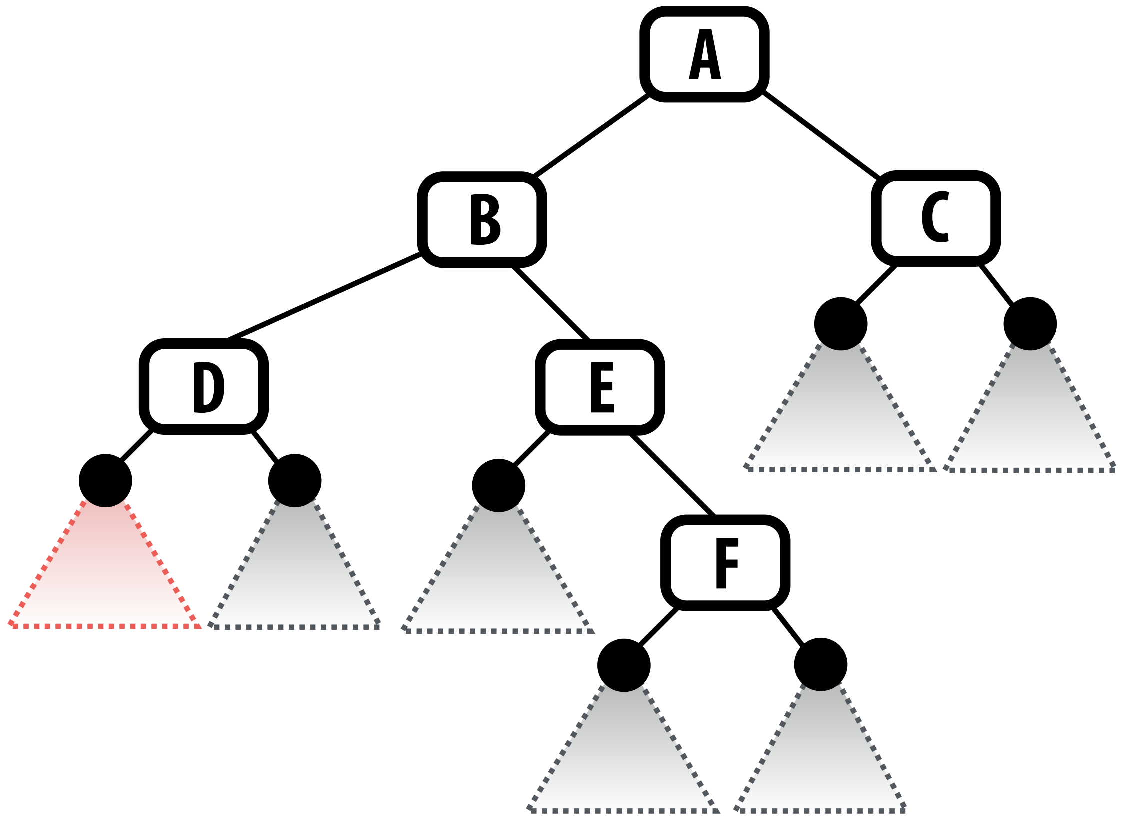
- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in front-to-back order
  - Unlike BVH, can terminate search after first hit is found.



# 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.

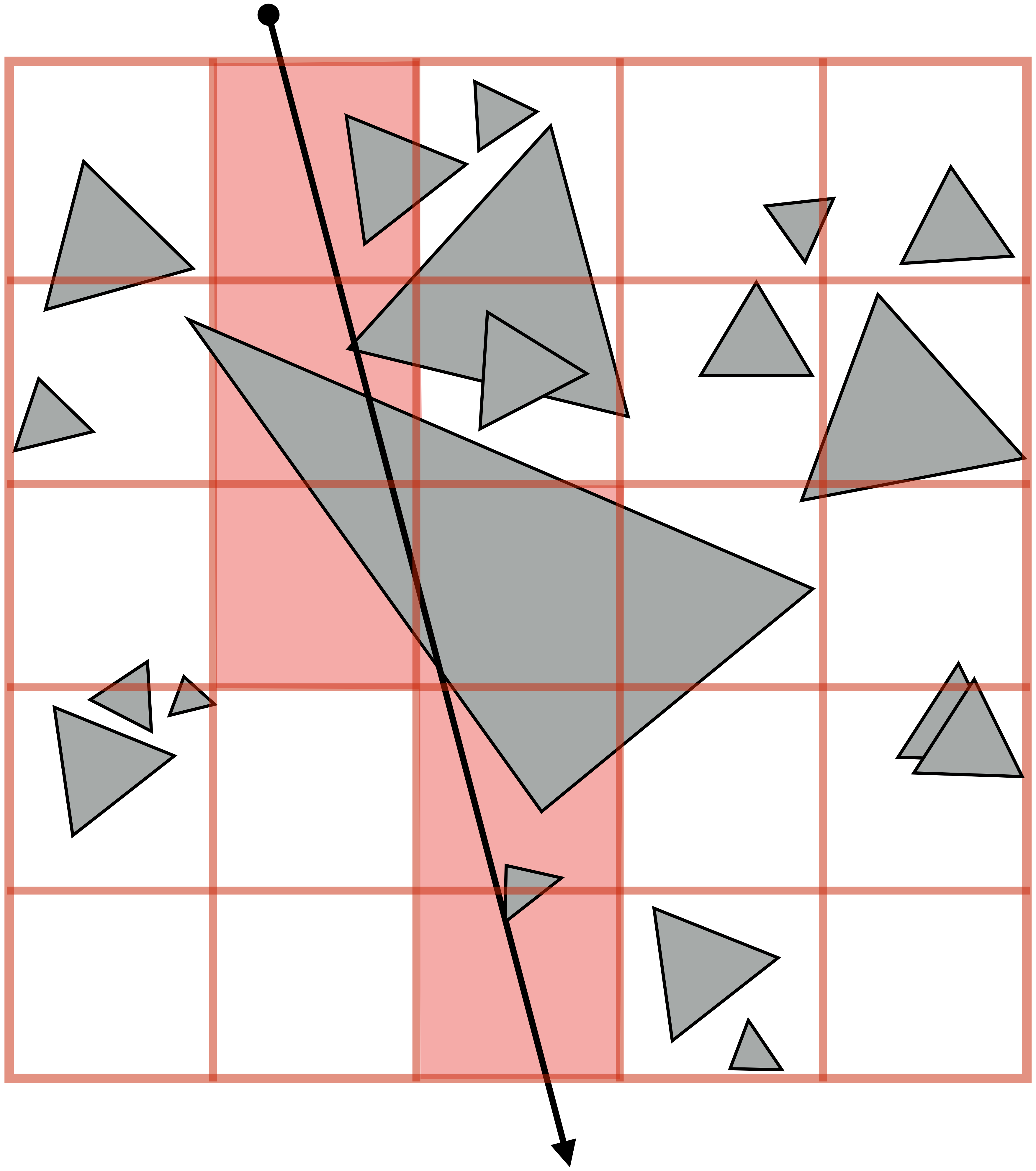
But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

Solution: require primitive intersection point to be within current leaf node.

(primitives may be intersected multiple times by same ray *)

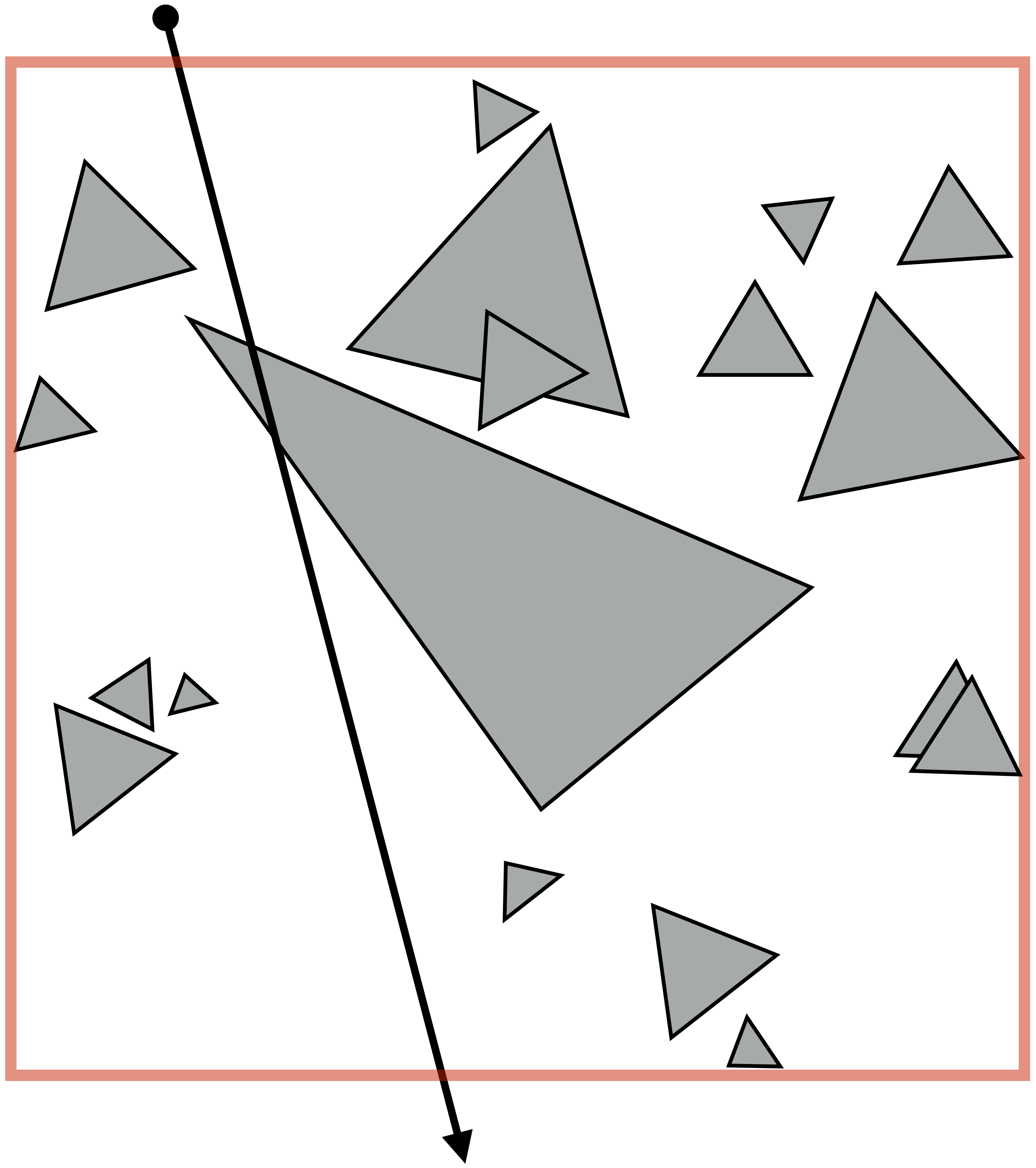
* Caching or "mailboxing" can be used to avoid repeated intersections

# Uniform grid

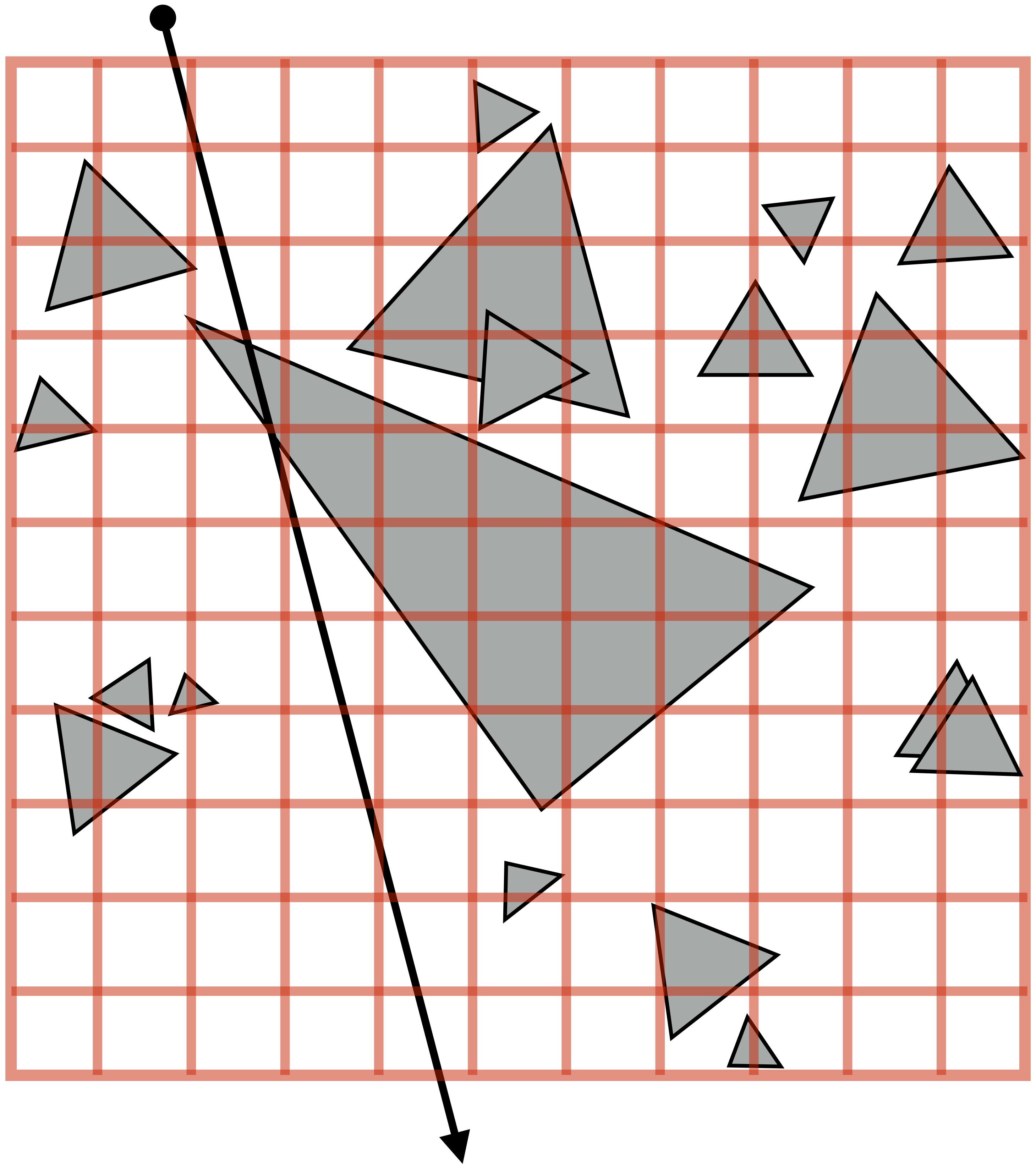


- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider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

# What should the grid resolution be?



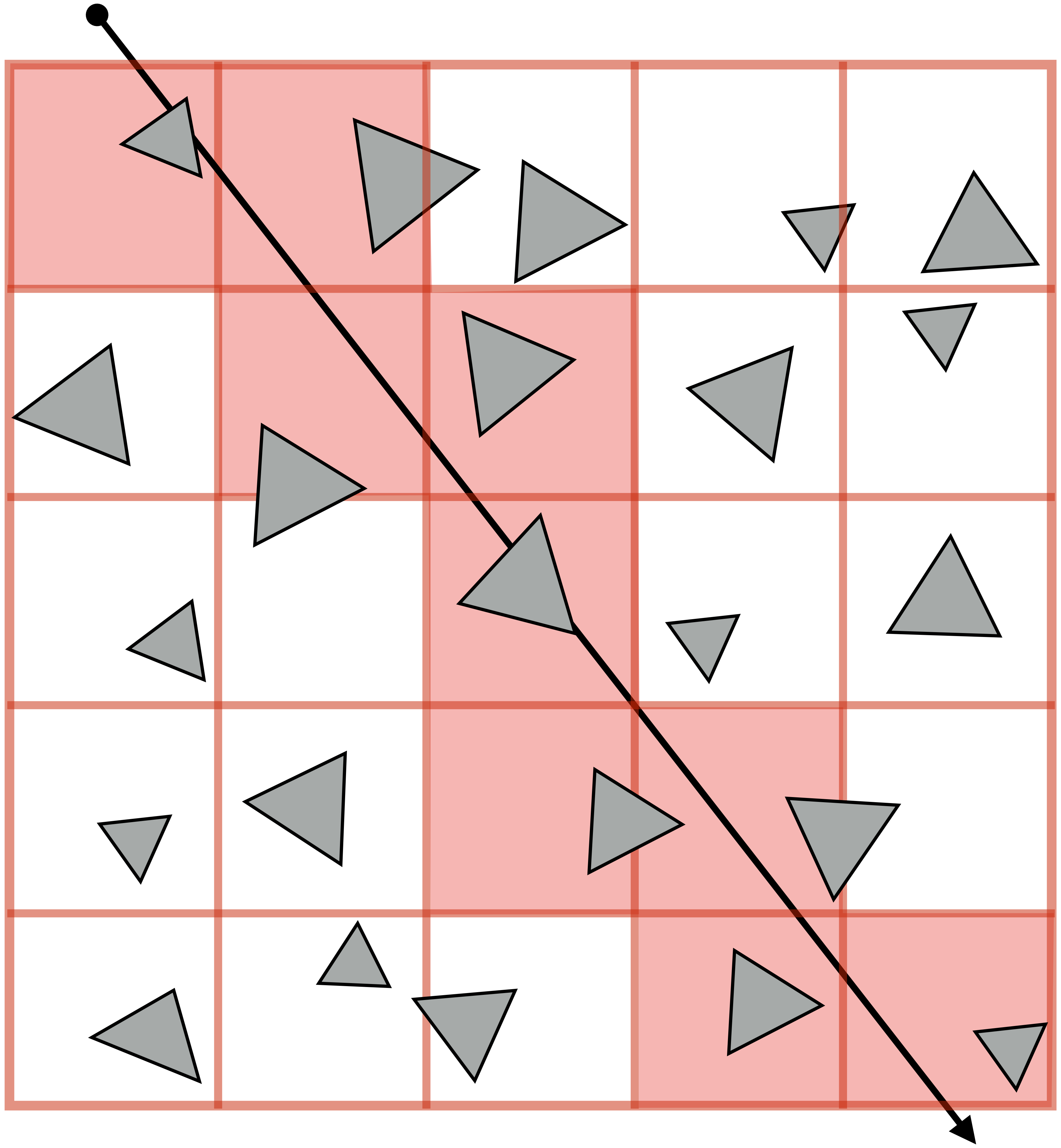
**Too few grids cell: degenerates to brute-force approach**



**Too many grid cells: incur significant cost traversing through cells with empty space**

# Heuristic

- **Choose number of voxels ~ total number of primitives**
  **(constant prims per voxel — assuming uniform distribution of primitives)**



**Intersection cost:** $O(\sqrt[3]{N})$

**(Q: Which grows faster, cube root of N or log(N)?**

# Uniform distribution of primitives

**Terrain / height fields:**

[Image credit: Misuba Renderer]

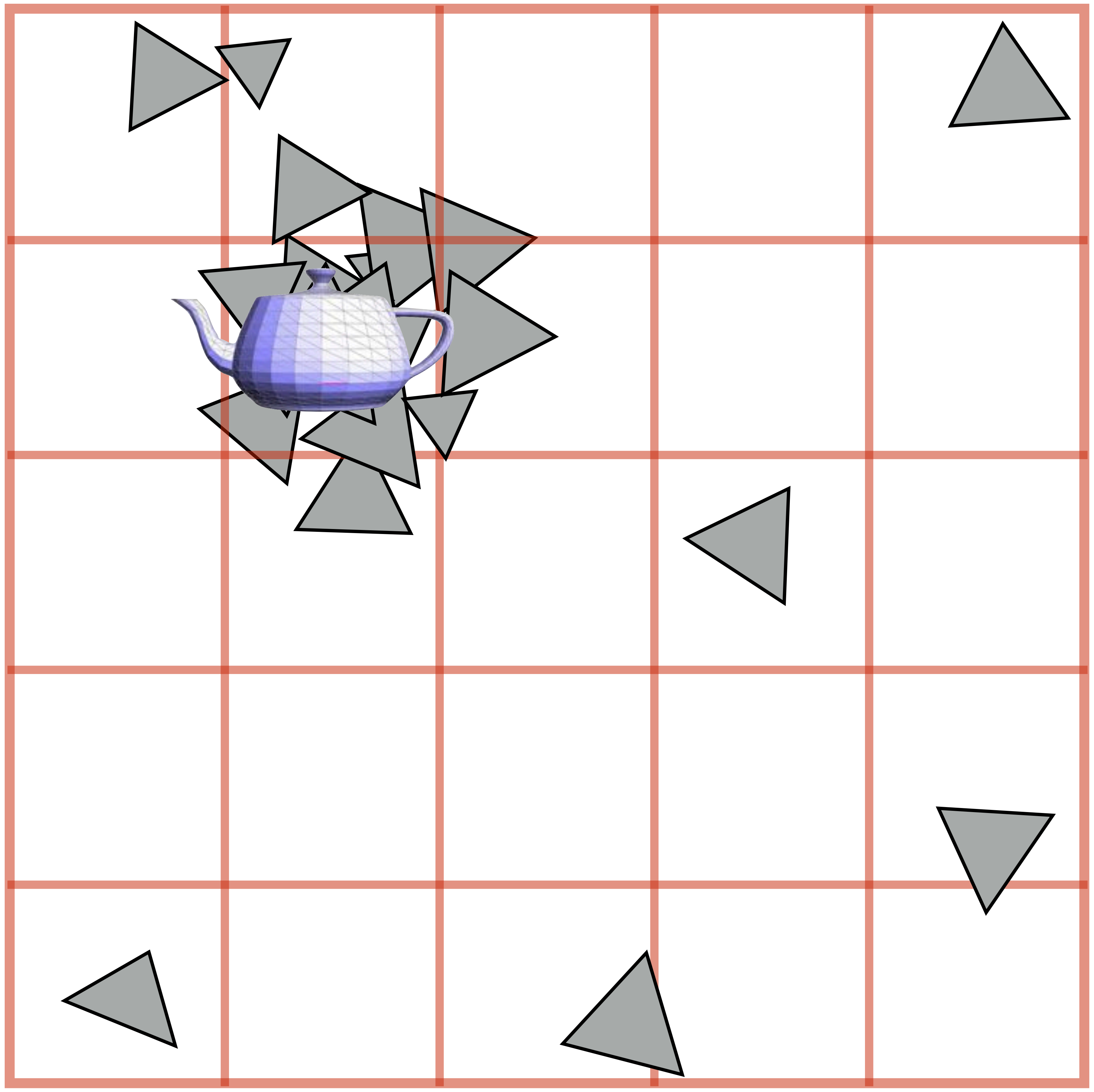
**Grass:**

[Image credit: www.kevinboulanger.net/grass.html]

Example credit: Pat Hanrahan

# 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)



**"Teapot in a stadium problem"**

Scene has large spatial extent.

Contains a high-resolution object that has small spatial extent (ends up in one grid cell)

# Non-uniform distribution of geometric detail

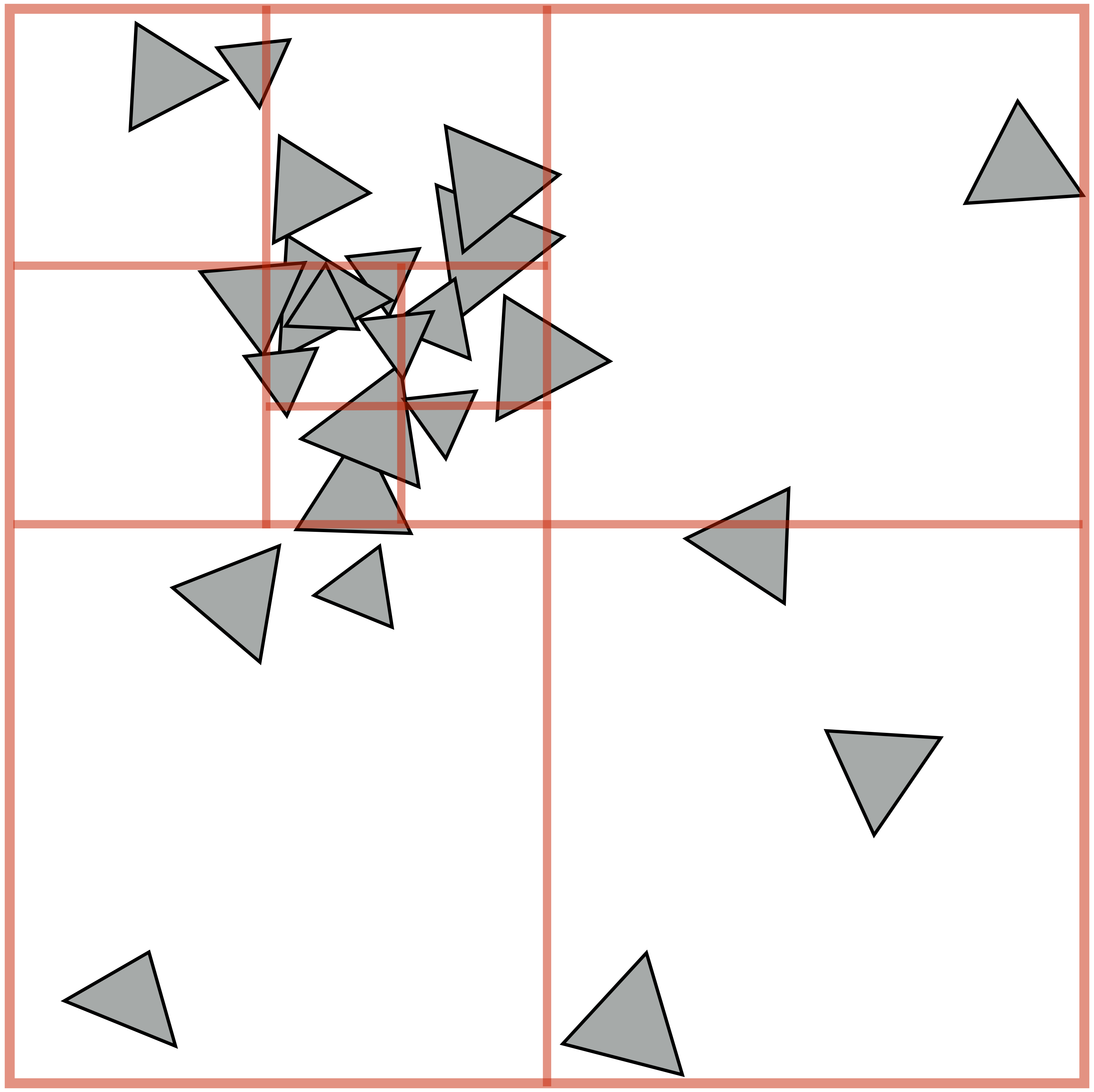
[Image credit: Pixar]

# 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)



Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)

# Summary of spatial acceleration structures:
# 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 cost 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

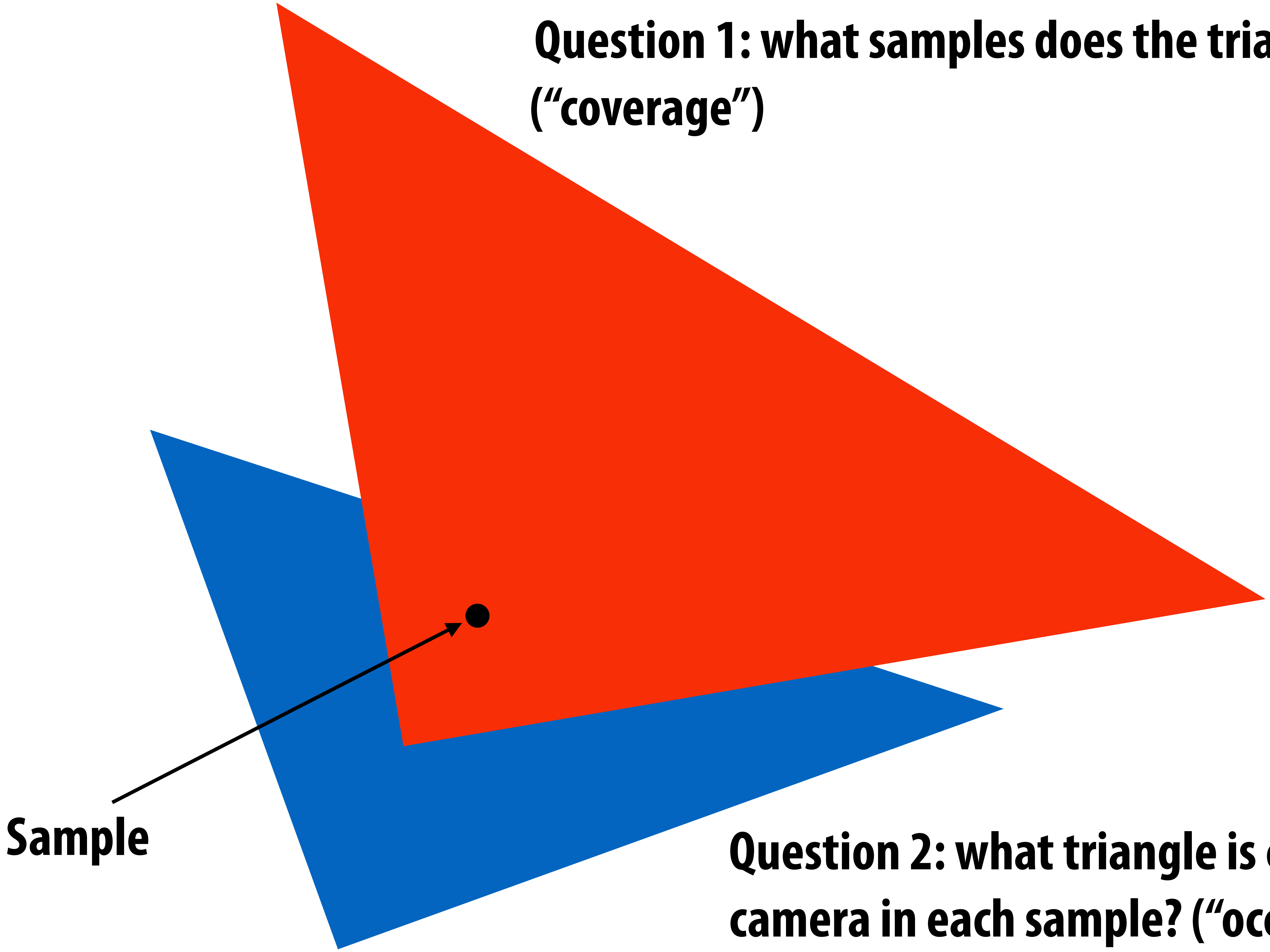
- Many, many combinations thereof…

# Rendering via ray casting: one common use of ray-scene intersection tests *

* Last lecture we briefly discussed the use of ray-scene queries for applications in geometry processing (e.g., inside-outside tests) and simulation (e.g., collision detection)

**Rasterization and ray casting are two algorithms for solving the same problem: determining “visibility from a camera”**

# Recall triangle visibility:



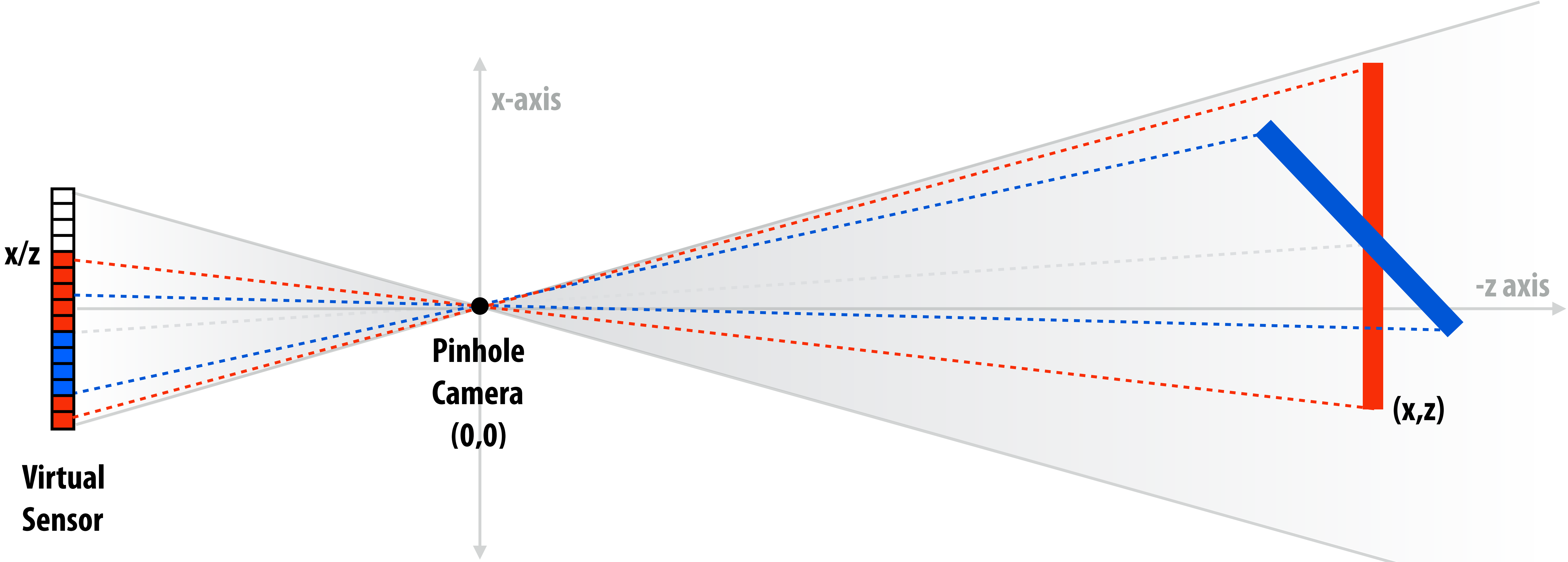
Question 1: what samples does the triangle overlap? ("coverage")

Sample

Question 2: what triangle is closest to the camera in each sample? ("occlusion")

# The visibility problem

- What scene geometry is visible at each screen sample?
  - What scene geometry projects into a screen pixel? (coverage)
  - Which geometry is visible from the camera at that pixel? (occlusion)



# Basic rasterization algorithm

**Sample = 2D point**

**Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point)**

**Occlusion: depth buffer**

```
initialize z_closest[] to INFINITY                 // store closest-surface-so-far for all samples
initialize color[]                                 // store scene color for all samples
for each triangle t in scene:                      // loop 1: triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:          // loop 2: visibility samples
        if (t_proj covers s)
            compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```

**"Given a triangle, find the samples it covers"**

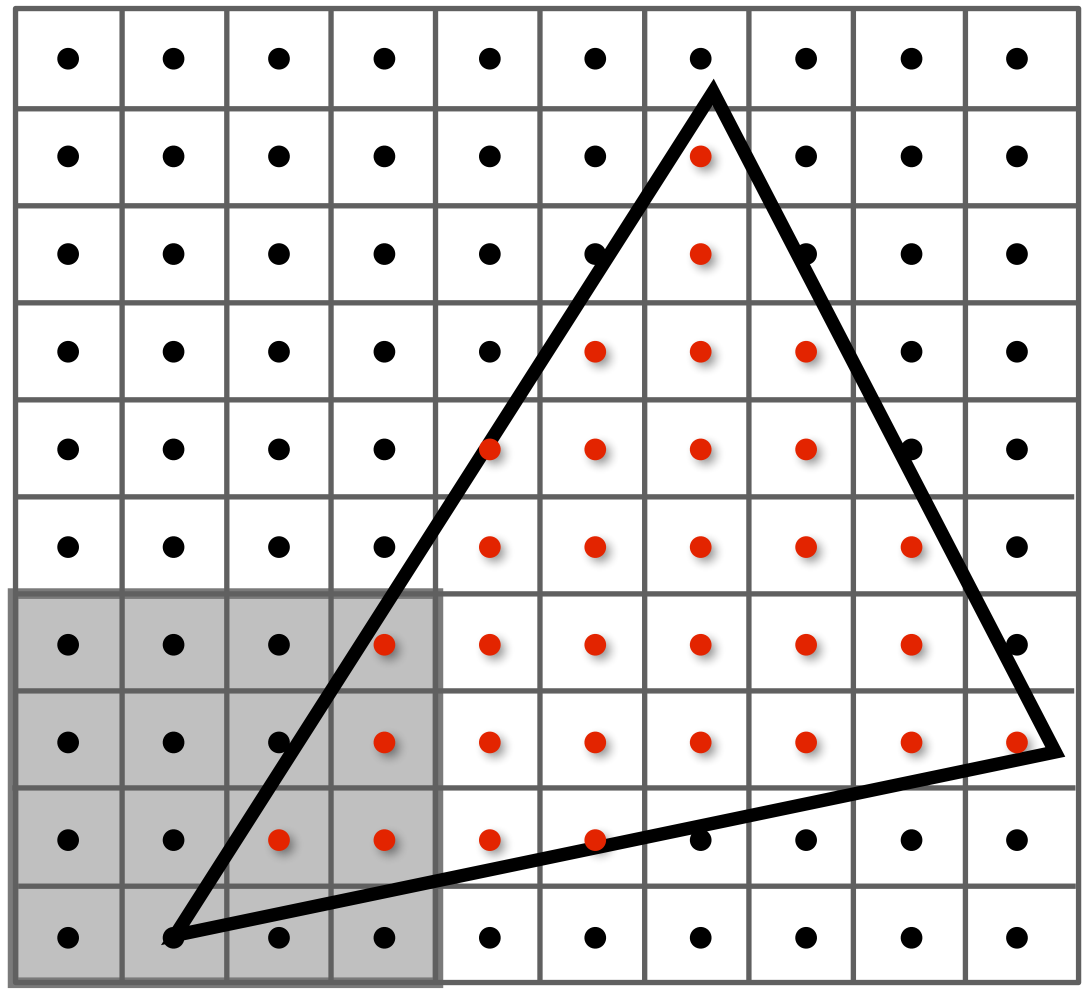
**(finding the samples is relatively easy since they are distributed uniformly on screen)**

**More modern hierarchical rasterization:**

**For each TILE of image**

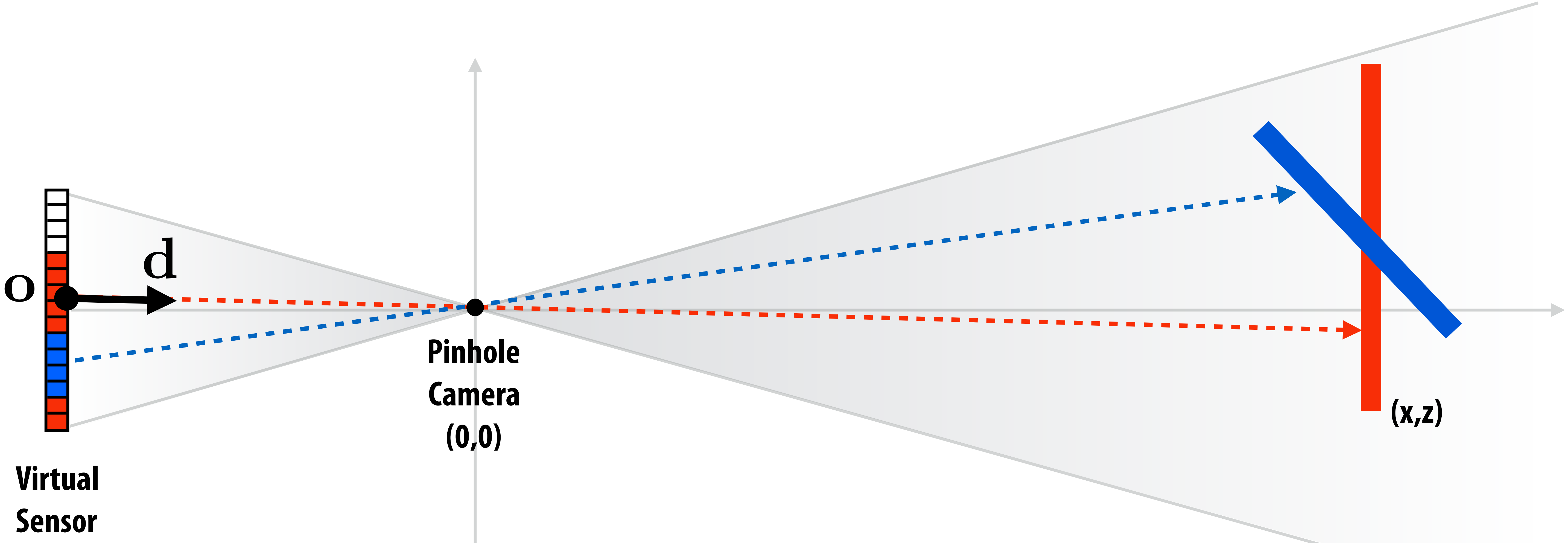
**If triangle overlaps tile, check all samples in tile**

**(What does this strategy remind you of? :-))**



# The visibility problem (described differently)

- In terms of casting rays from the camera:
  - Is a scene primitive hit by a ray originating from a point on the virtual sensor and traveling through the aperture of the pinhole camera? (coverage)
  - What primitive is the first hit along that ray? (occlusion)



# Basic ray casting algorithm

Sample = a ray in 3D

Coverage: 3D ray-triangle intersection tests (does ray "hit" triangle)

Occlusion: closest intersection along ray

```
initialize color[]                                 // store scene color for all samples
for each sample s in frame buffer:                 // loop 1: visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.min_t = INFINITY                             // only store closest-so-far for current ray
    r.tri = NULL;
    for each triangle tri in scene:                  // loop 2: triangles
        if (intersects(r, tri)) {                    // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.min_t)
                update r.min_t and r.tri = tri;
        }
    color[s] = compute surface color of triangle r.tri at hit point
```

Compared to rasterization approach: just a reordering of the loops!

"Given a ray, find the closest triangle it hits"

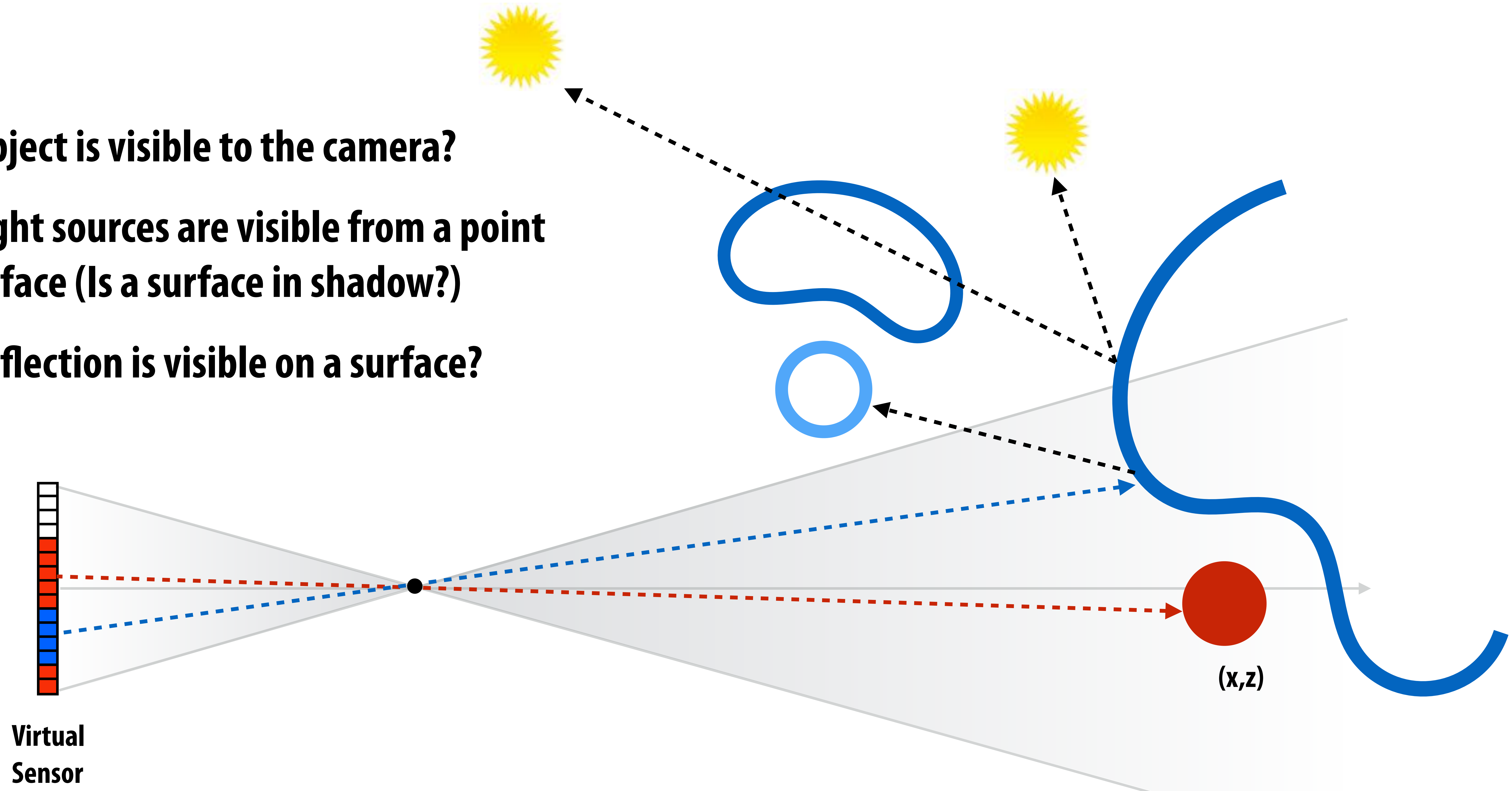
As we saw today, the brute force "for each triangle" loop is typically accelerated using an acceleration structure. (A rasterizer's "for each sample" inner loop is not just a loop over all screen samples either!)

# Basic rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order
  - Store depth buffer (random access to regular structure of fixed size)
  - Don’t have to store entire scene in memory, naturally supports unbounded size scenes
- **Ray casting:**
  - Proceeds in screen sample order
    - Don’t have to store closest depth so far for the entire screen (just current ray)
    - Natural order for rendering transparent surfaces (process surfaces in the order the are encountered along the ray: front-to-back or back-to-front)
  - Must store entire scene
  - Performance more strongly depends on distribution of primitives in scene
- **Modern high-performance implementations of rasterization and ray-casting embody very similar techniques**
  - Hierarchies of rays/samples
  - Hierarchies of geometry
  - Deferred shading
  - …

# Ray-scene intersection is a general visibility primitive:

## What object is visible along this ray?



What object is visible to the camera?

What light sources are visible from a point on a surface (Is a surface in shadow?)

What reflection is visible on a surface?

In contrast, rasterization is a highly-specialized solution for computing visibility for a set of uniformly distributed rays originating from the same point (most often: the camera)

# Next time: light