Spatial Data Structures

Computer Graphics
CMU 15-462/662

Complexity of geometry

How can we <u>efficiently</u> perform a geometric query on a scene of this complexity?

Important use case: ray tracing

Review: ray-triangle intersection

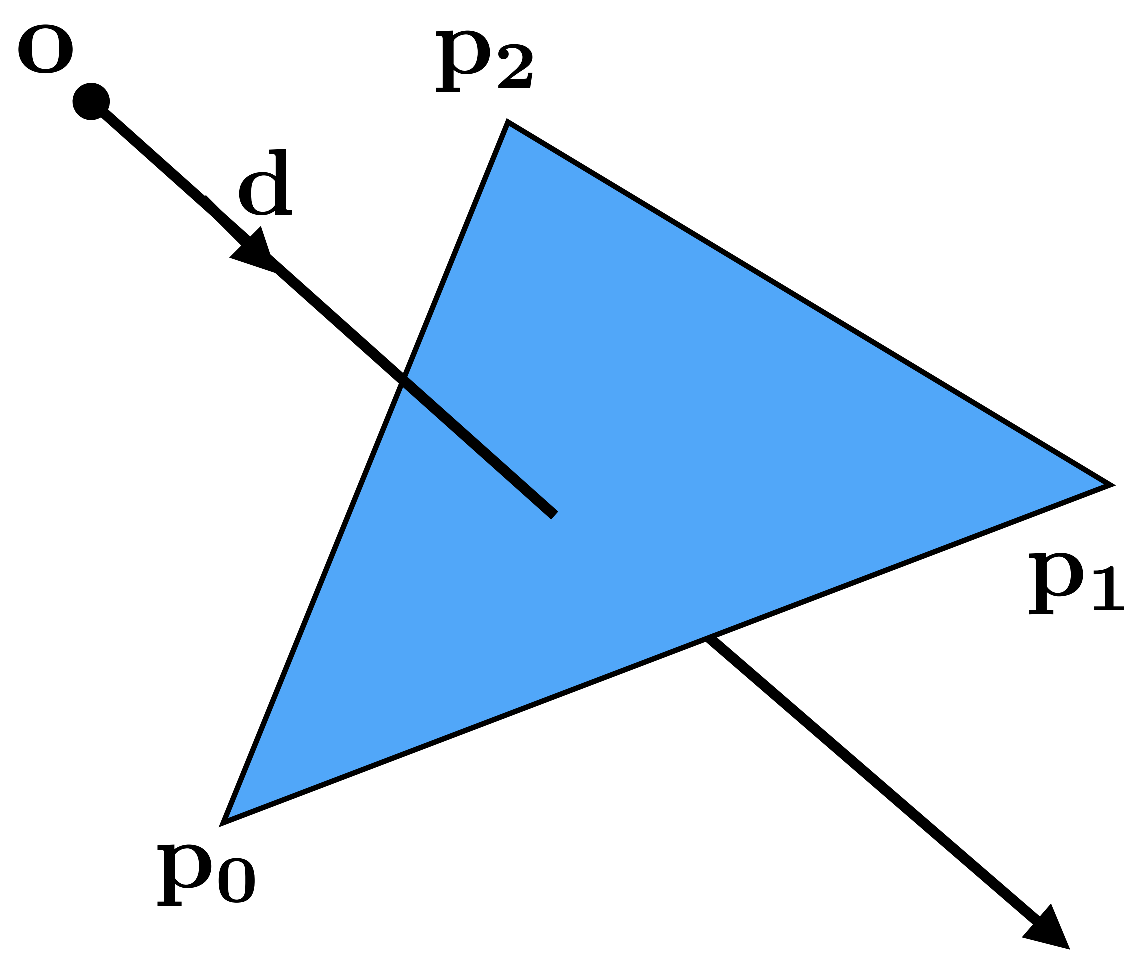
- **Find ray-plane intersection**

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**

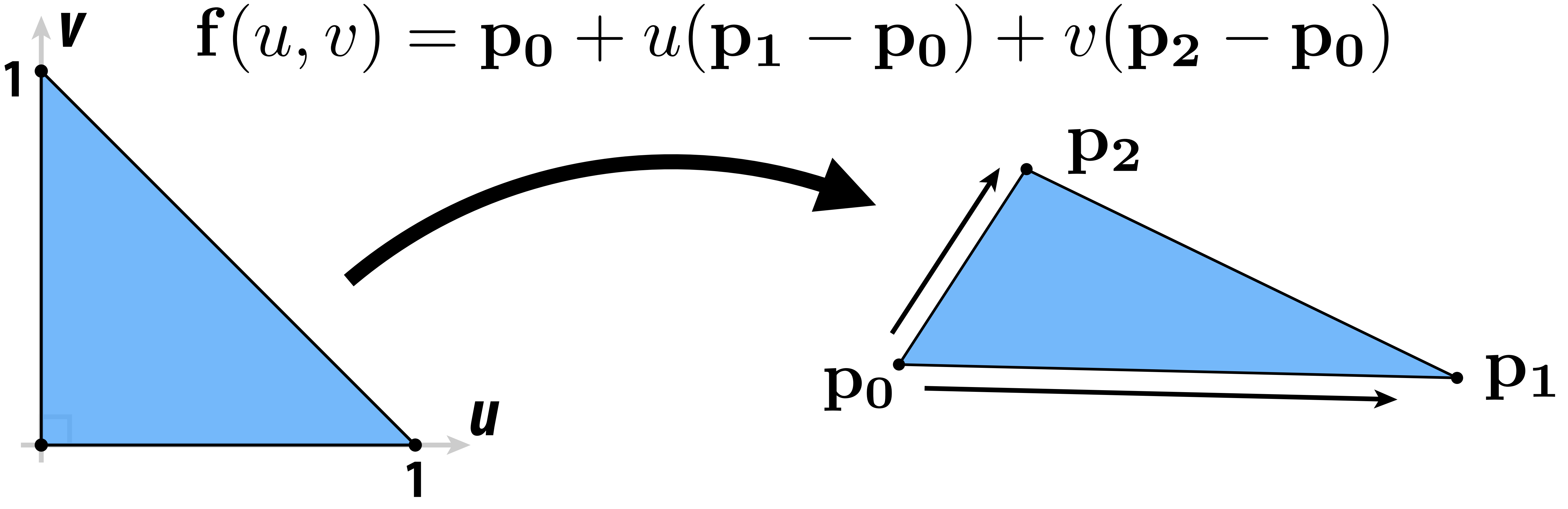
Ray-triangle intersection—a different way

- **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—a different way

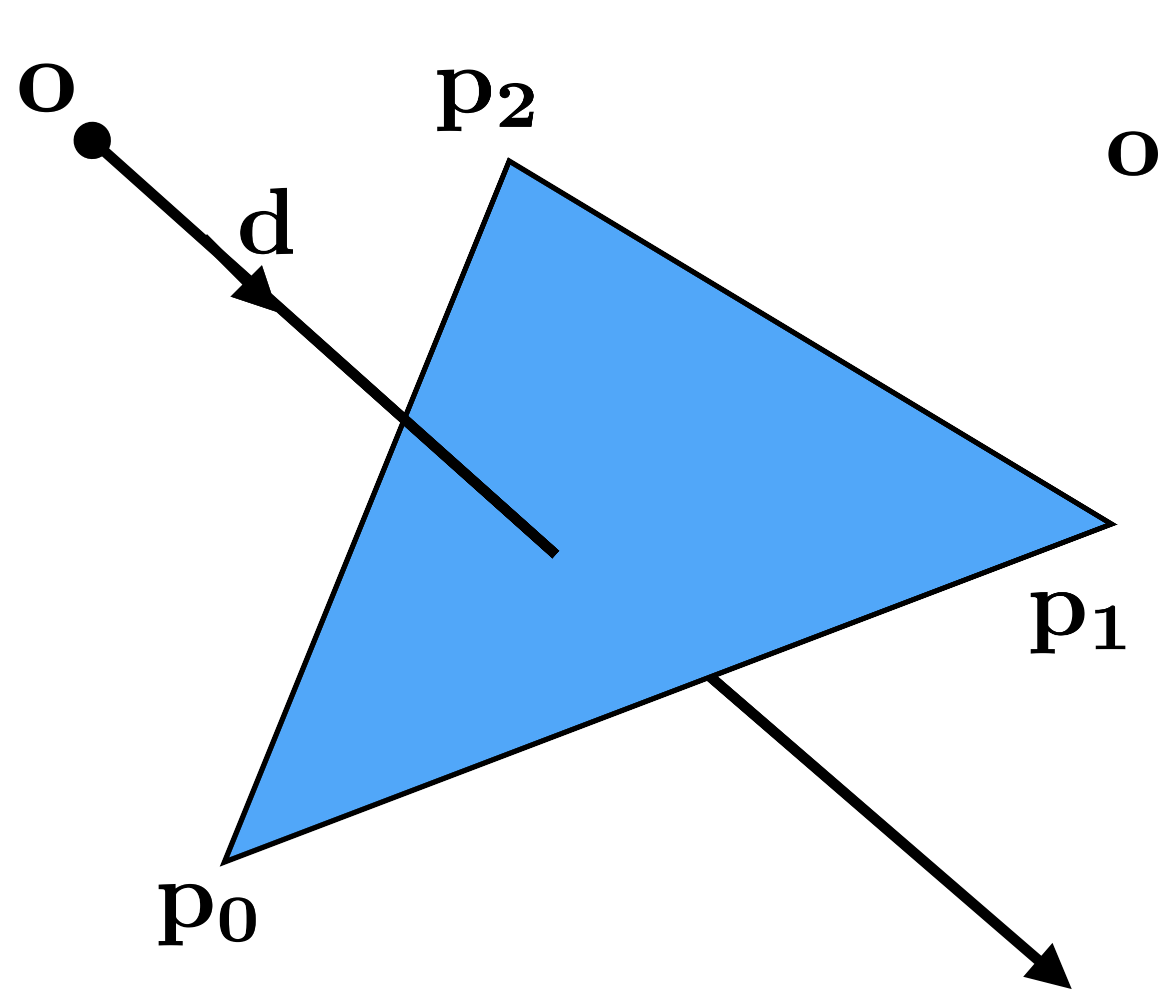
Plug parametric ray equation directly into equation for points on triangle:

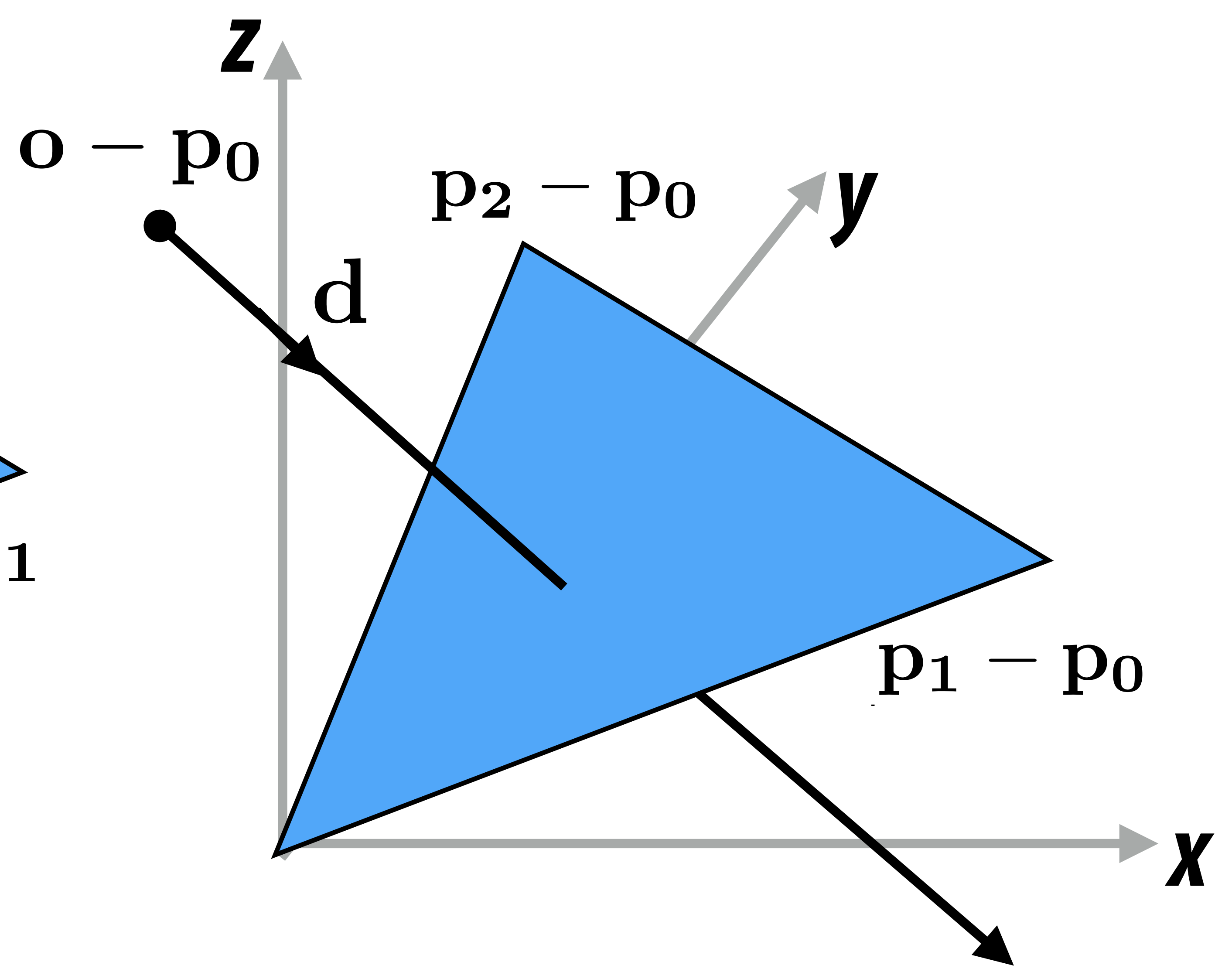
$$\mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0}) = \mathbf{o} + t\mathbf{d}$$

Solve for u, v, t:

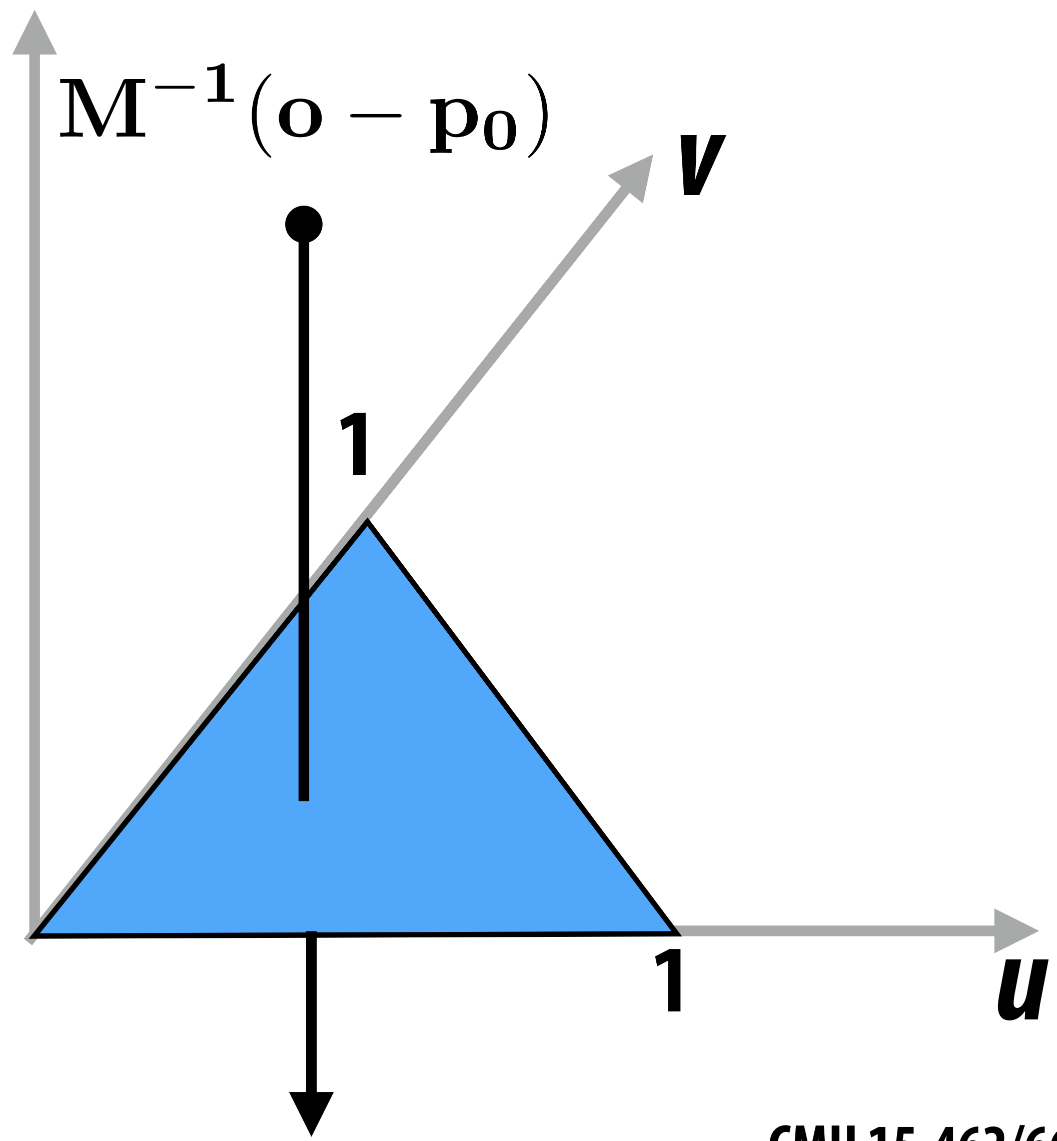
$$\underbrace{\begin{bmatrix} \mathbf{p_1} - \mathbf{p_0} & \mathbf{p_2} - \mathbf{p_0} & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p_0}$$

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





z
$\mathbf{o} - \mathbf{p_0}$
$\mathbf{d}$
$\mathbf{p_2} - \mathbf{p_0}$
y
$\mathbf{p_1} - \mathbf{p_0}$
x



First Hit Problem

Given a scene defined by a set of *N* primitives and a ray *r*, find the <u>closest</u> point of intersection of *r* with the scene

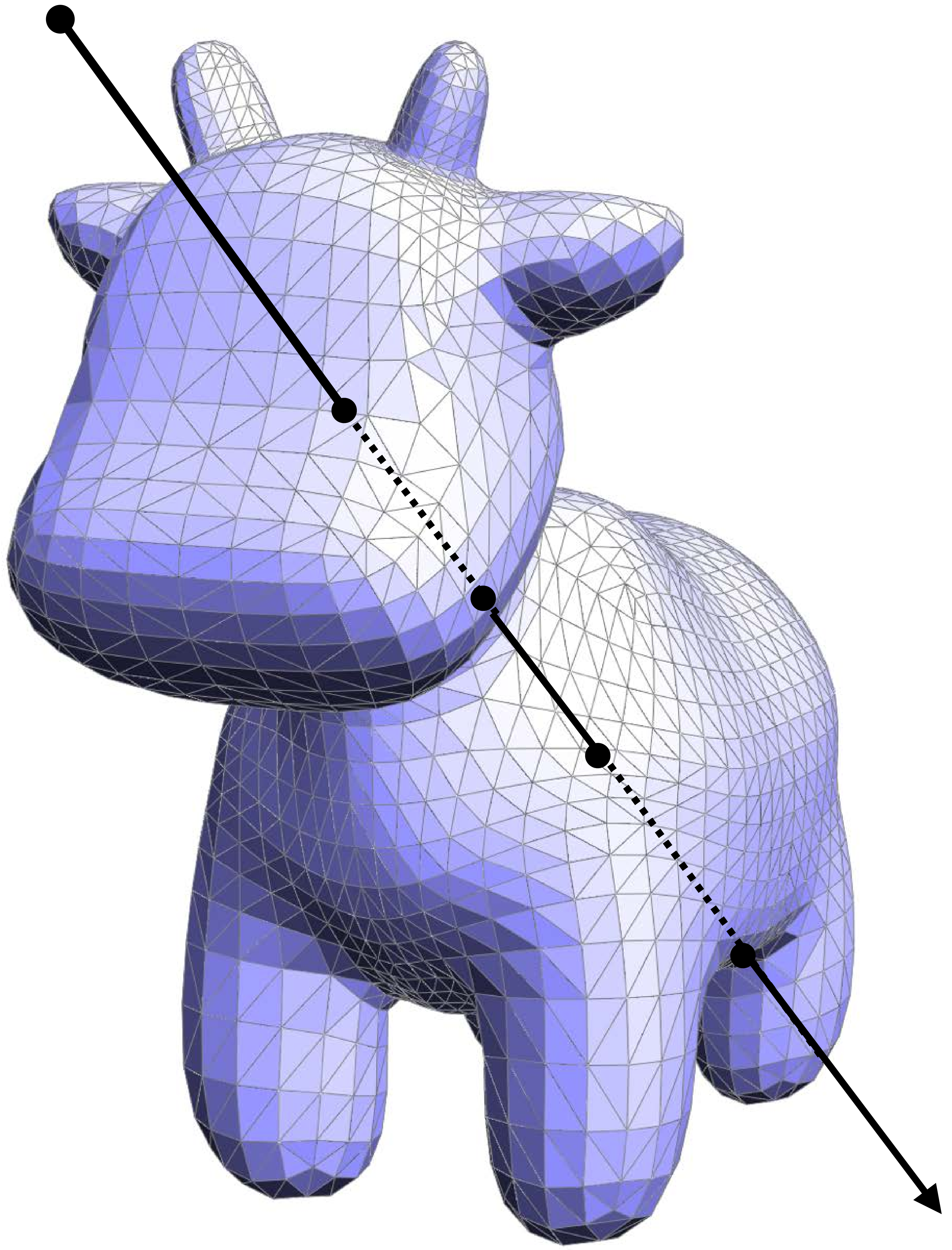
"Find the first primitive the ray hits"

Naïve algorithm?

1. **Intersect ray with <u>every</u> triangle**
2. **Keep the closest hit point**

Complexity? $O(N)$

Can we do better?



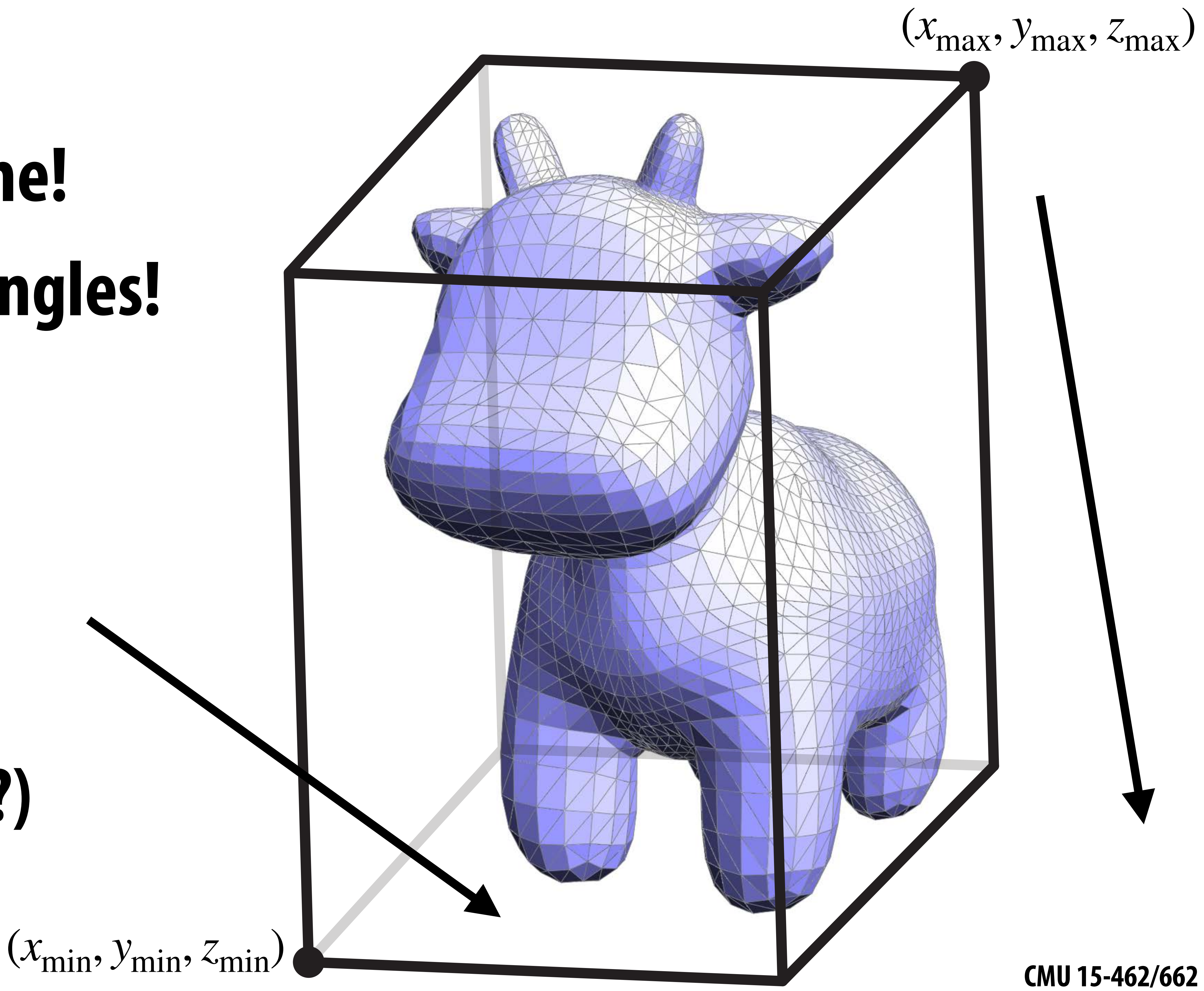
Bounding Box

- Precompute smallest "bounding box" around all primitives
  - Q: How?
  - A: Loop over vertices; keep max/min (x,y,z) coordinates
- Intersect ray with box
  - If it misses, we're done!
  - If it hits…try all triangles!

Did we actually do better?

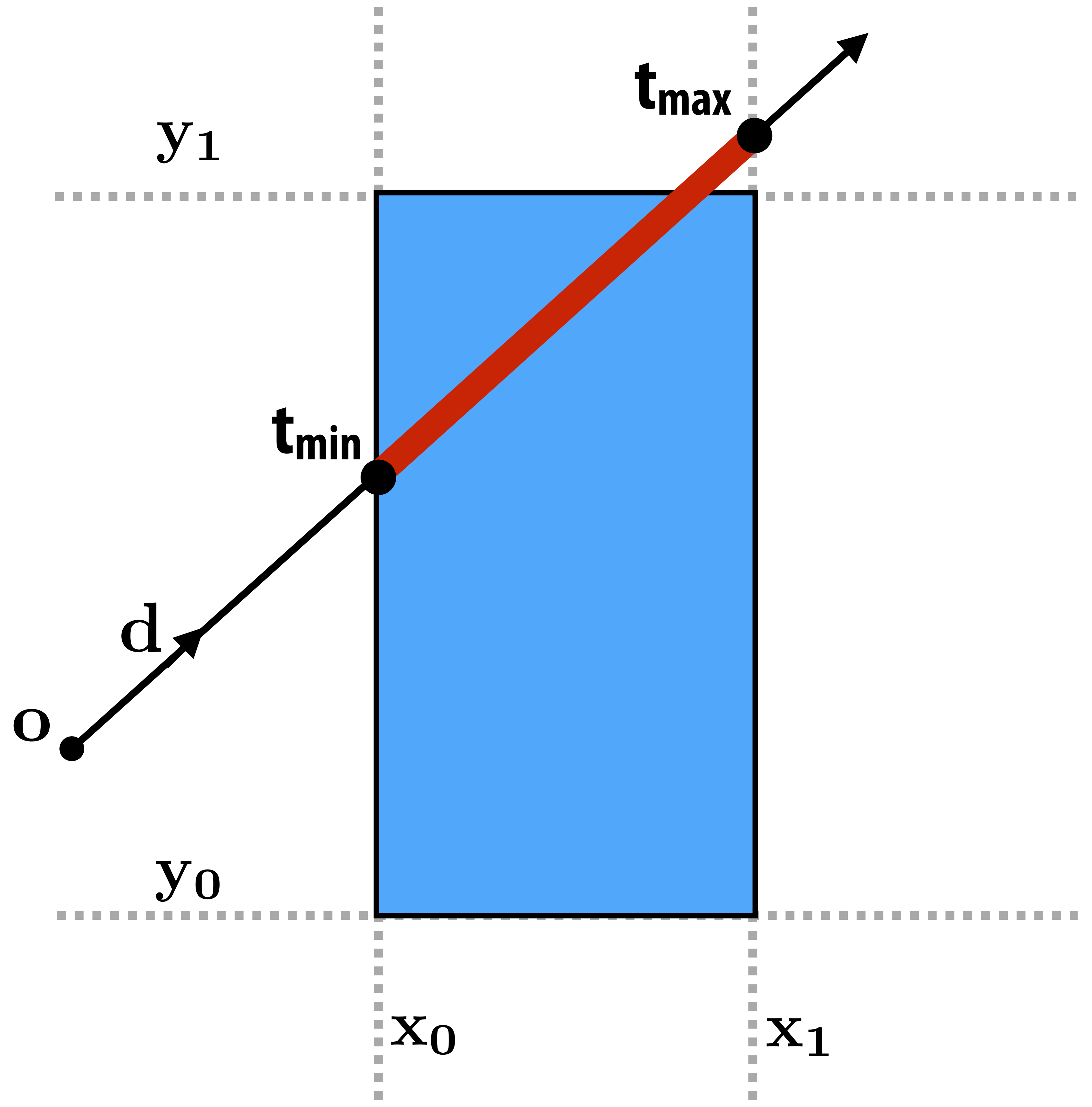
No! Worst case is still O(N)

(Also: ray-box intersection?)


$(x_{\max}, y_{\max}, z_{\max})$
$(x_{\min}, y_{\min}, z_{\min})$

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):

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

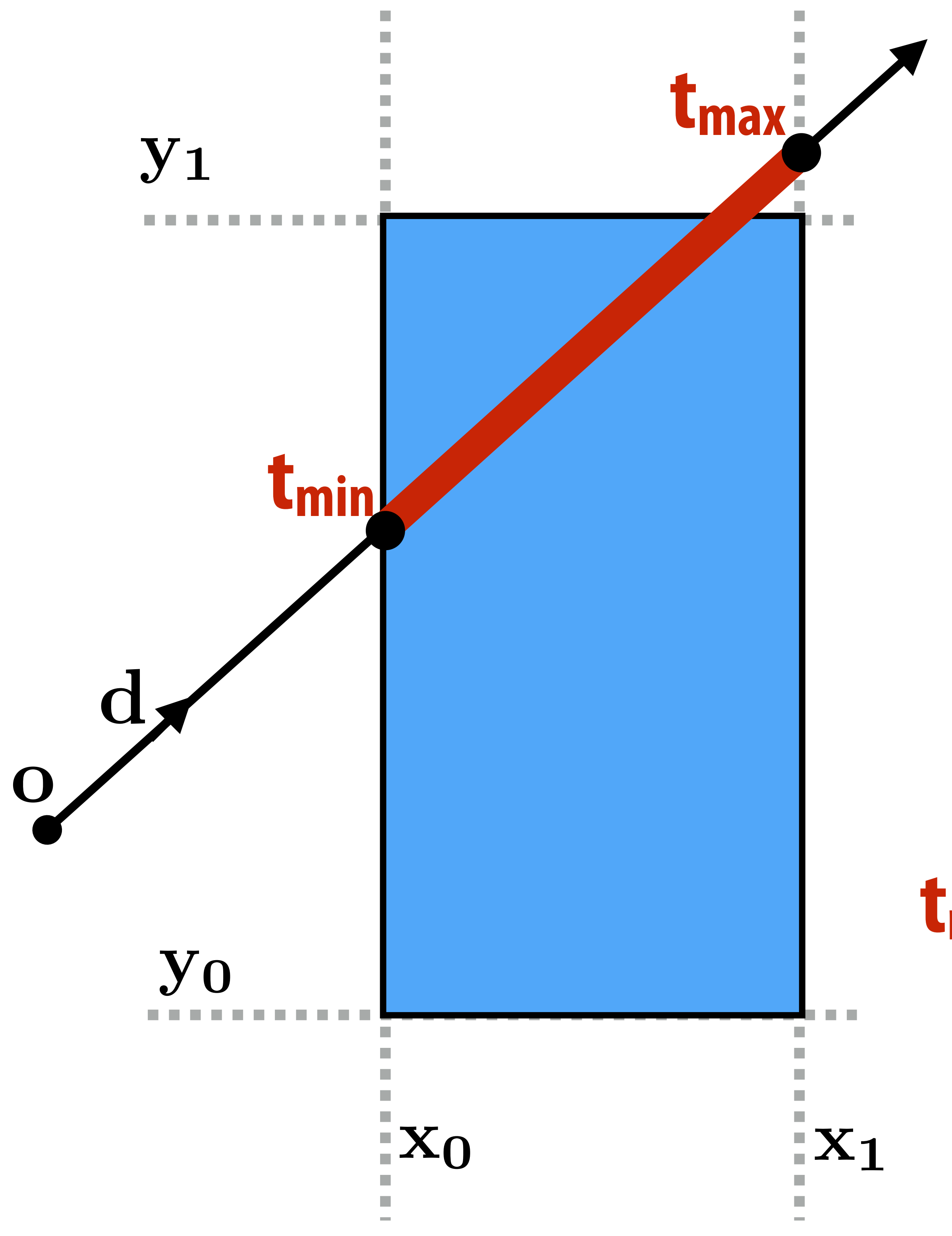
$$c = x_0$$

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

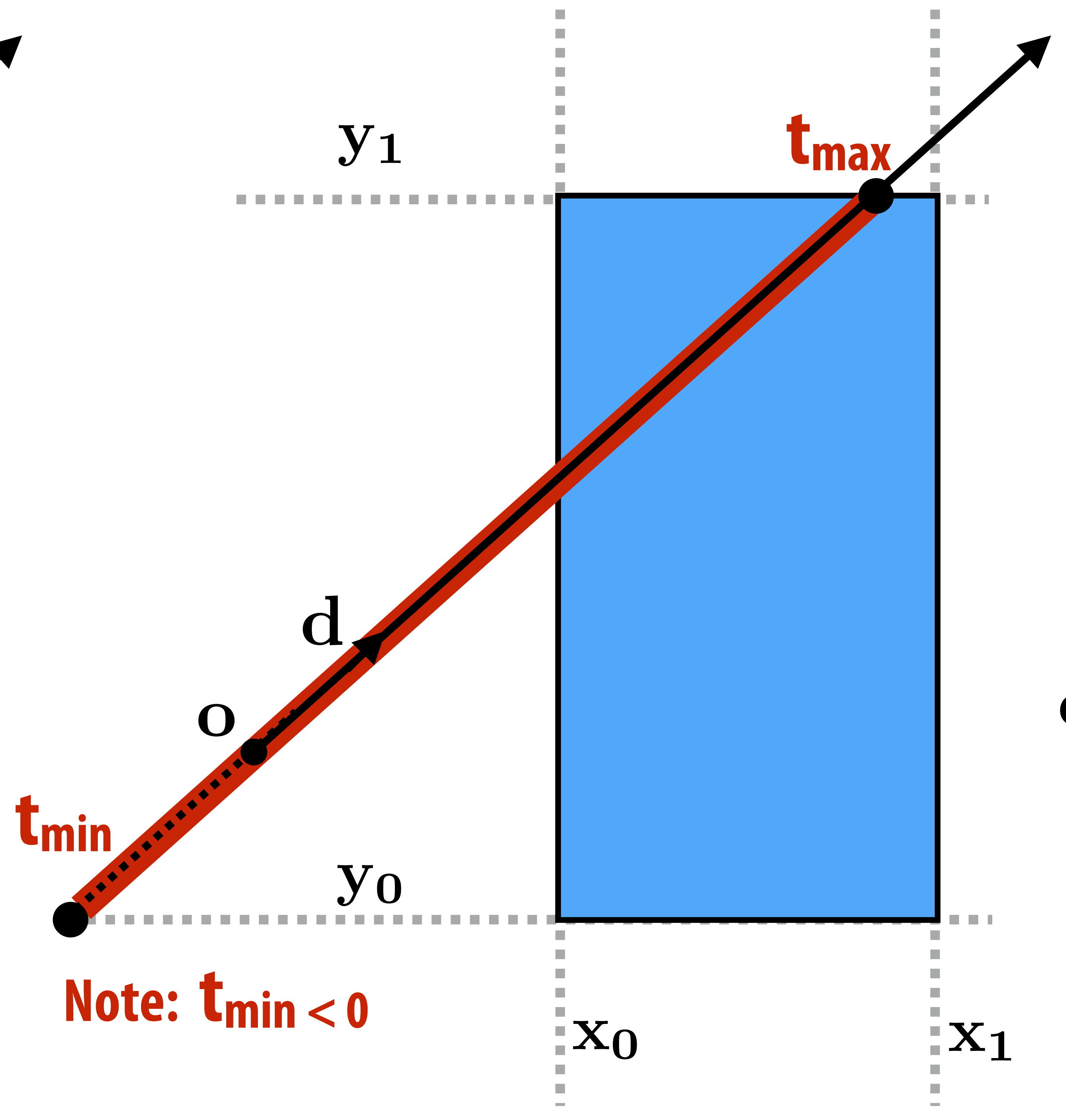
Figure shows intersections with x=x_0 and x=x_1 planes.

Ray-axis-aligned-box intersection

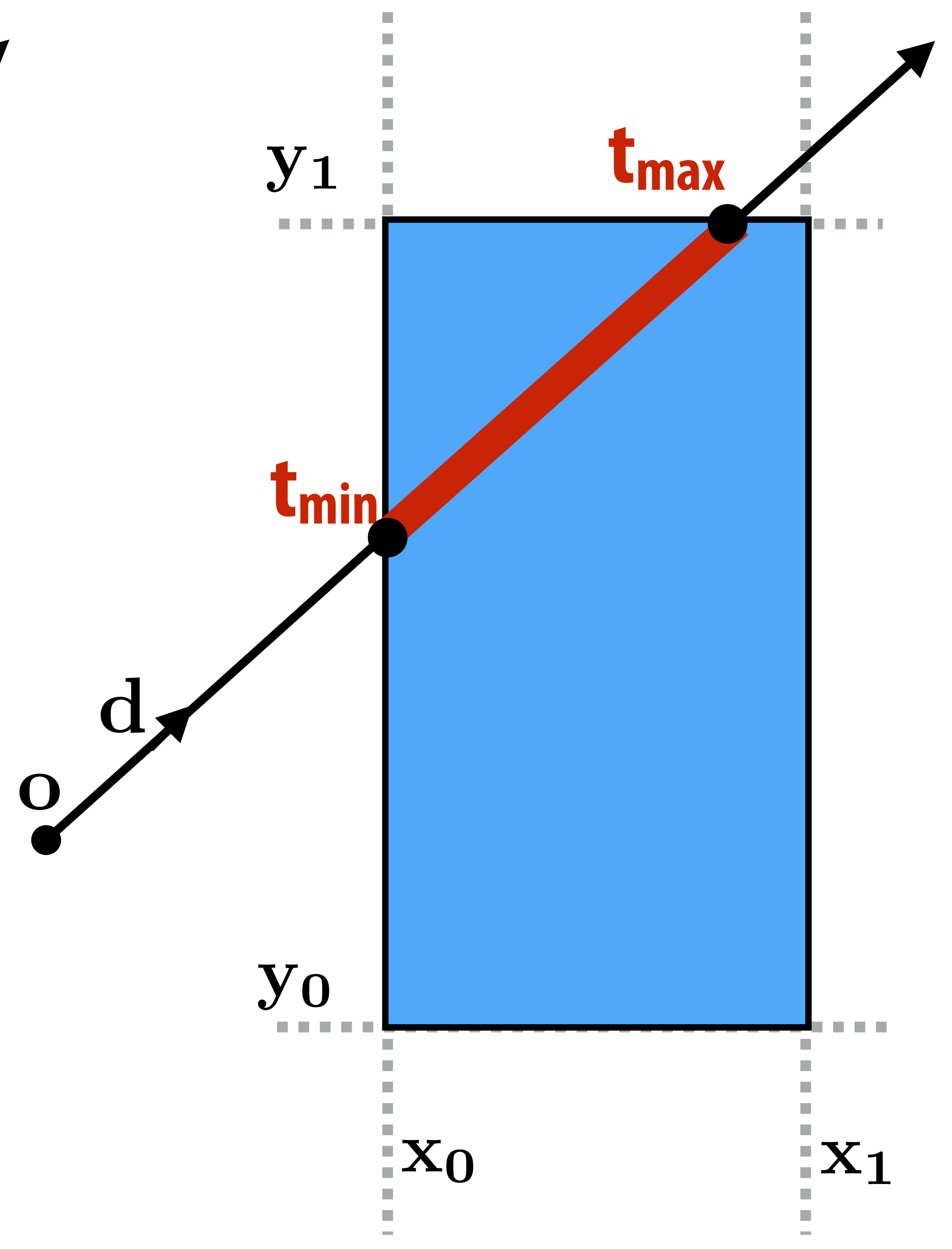
Compute intersections with all planes, take intersection of t_{min}/t_{max} intervals



Intersections with x planes



Intersections with y planes



Final intersection result

How do we know when the ray misses the box?

Ok, but we still didn't make it any faster!

How do we speed things up?

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 $\boxed{20}$ 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:

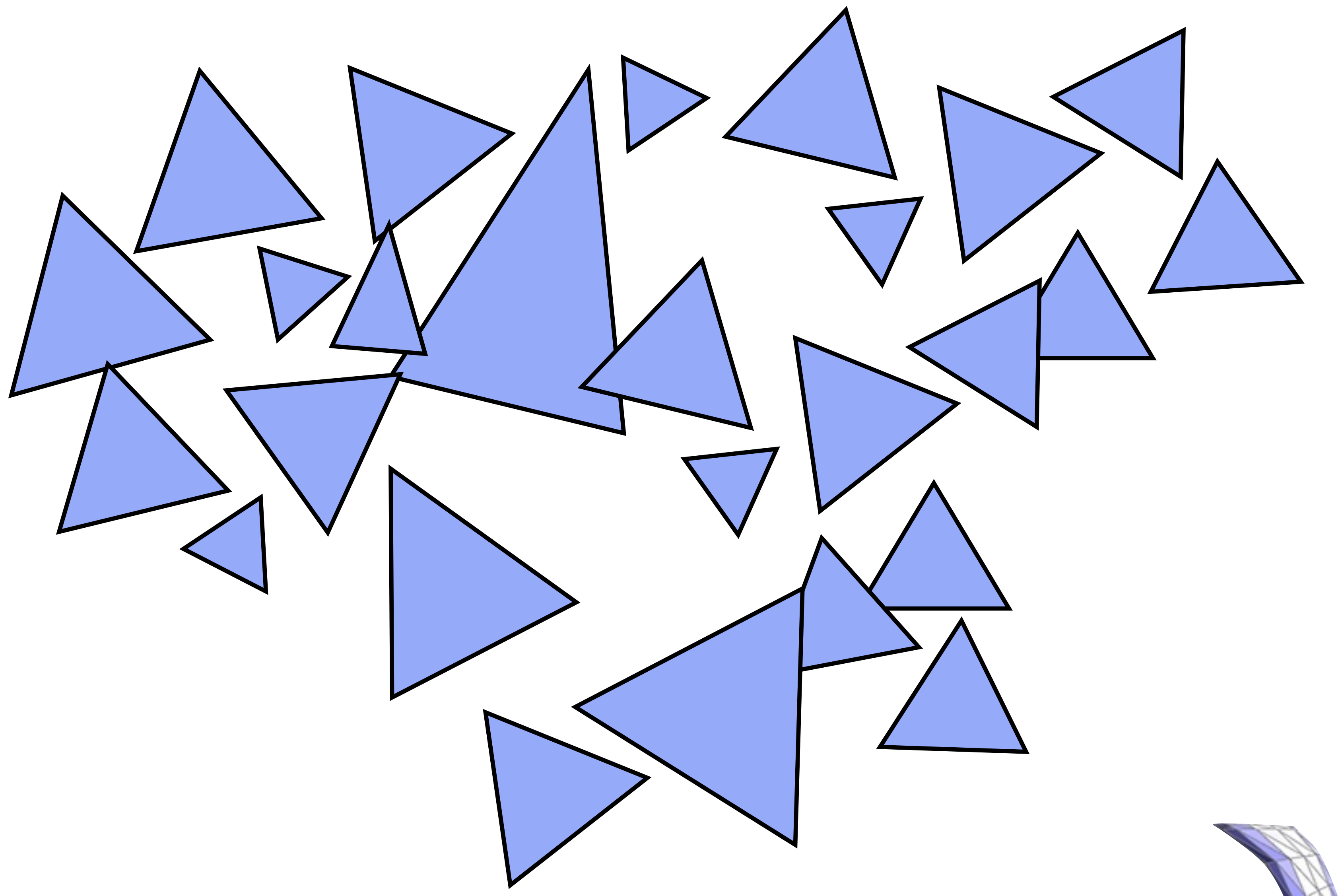
$\boxed{1\quad 2\quad 6\quad 8\quad \boxed{10\quad 11\quad \boxed{\boxed{20}\quad 25}}}$ 30 64 80 100 111 123 200 950

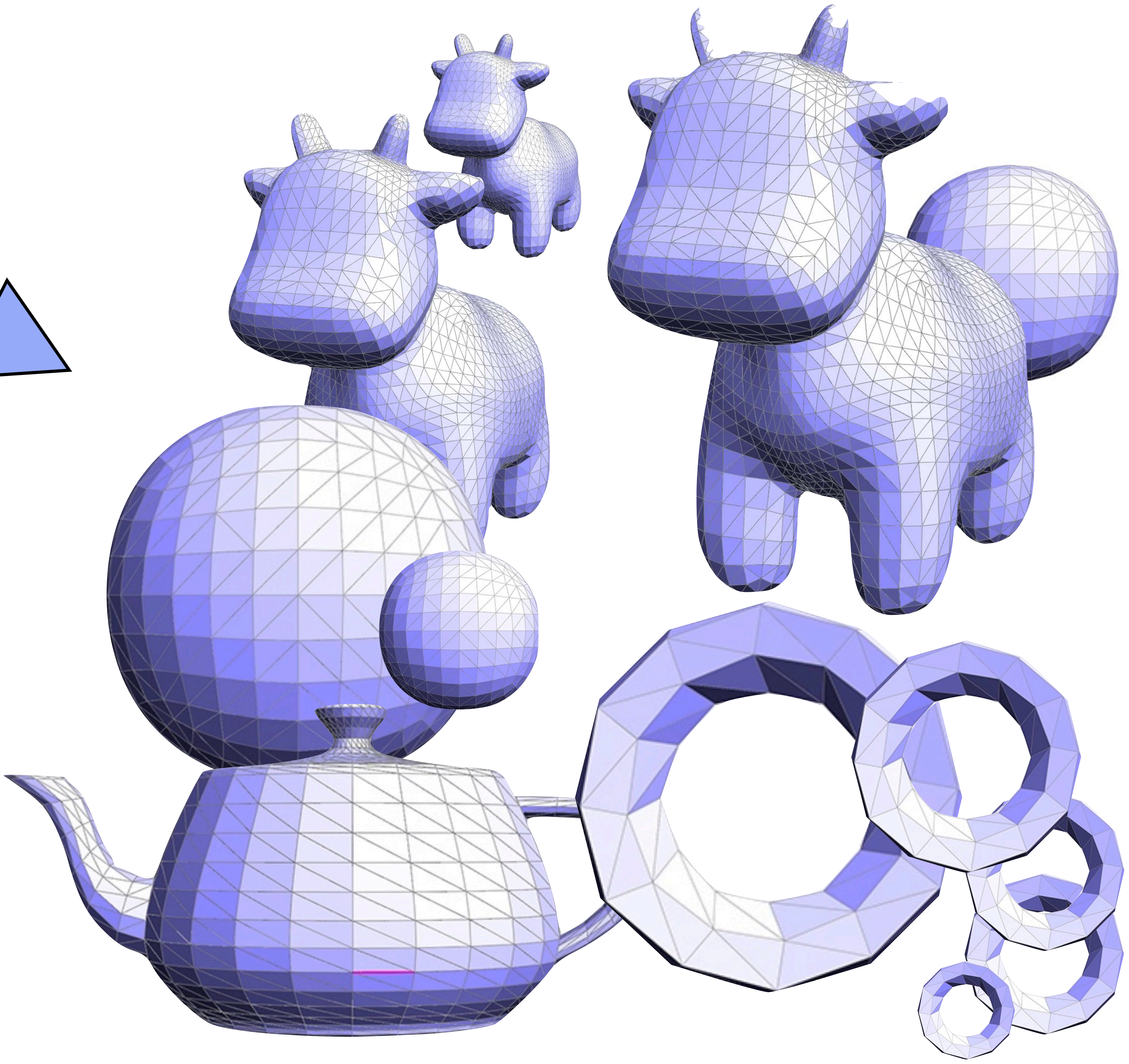
How much does it now cost to find k (*including sorting*)?

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

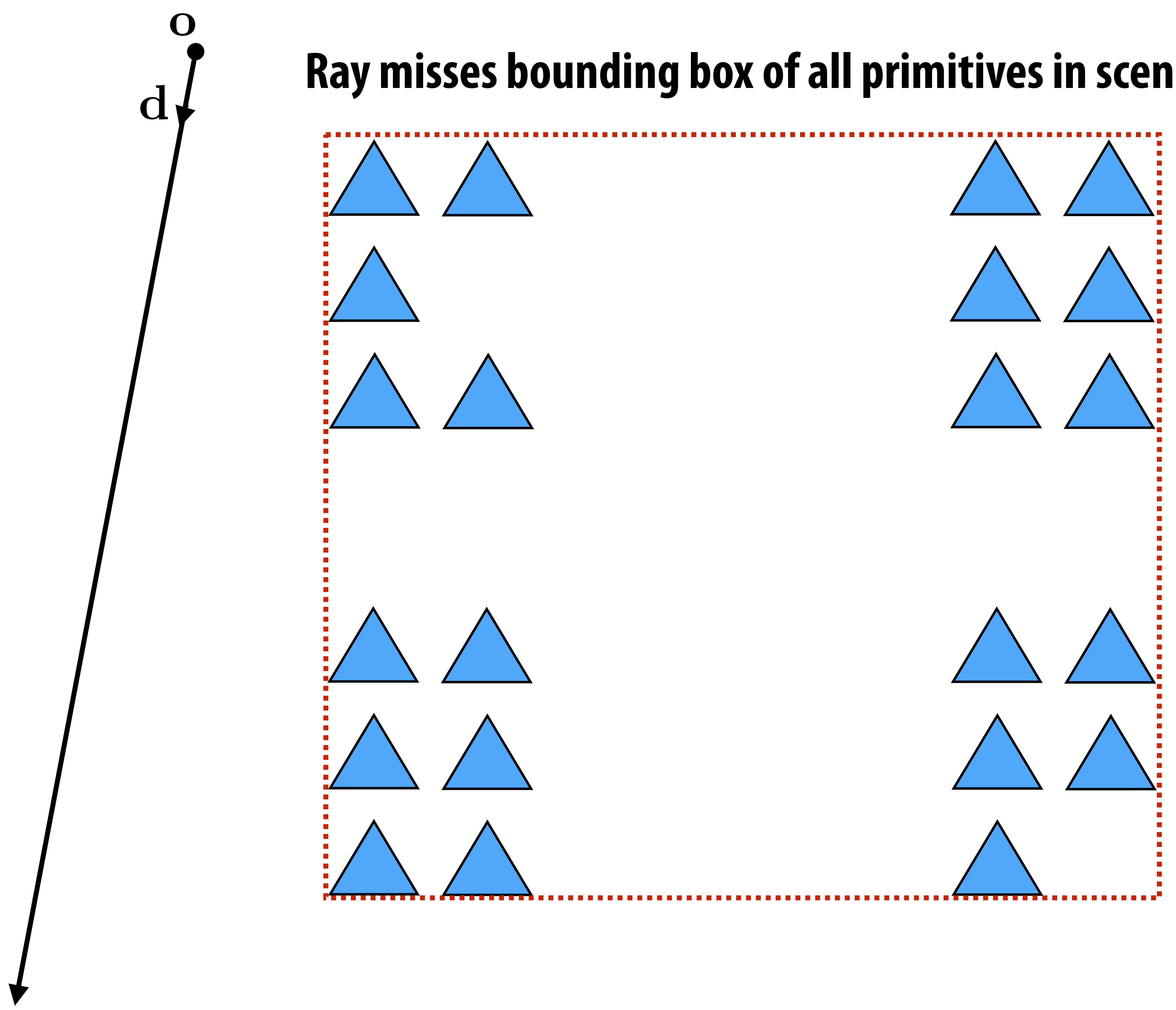
Amortized cost: O(log n) …*much* better!

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





Simple case

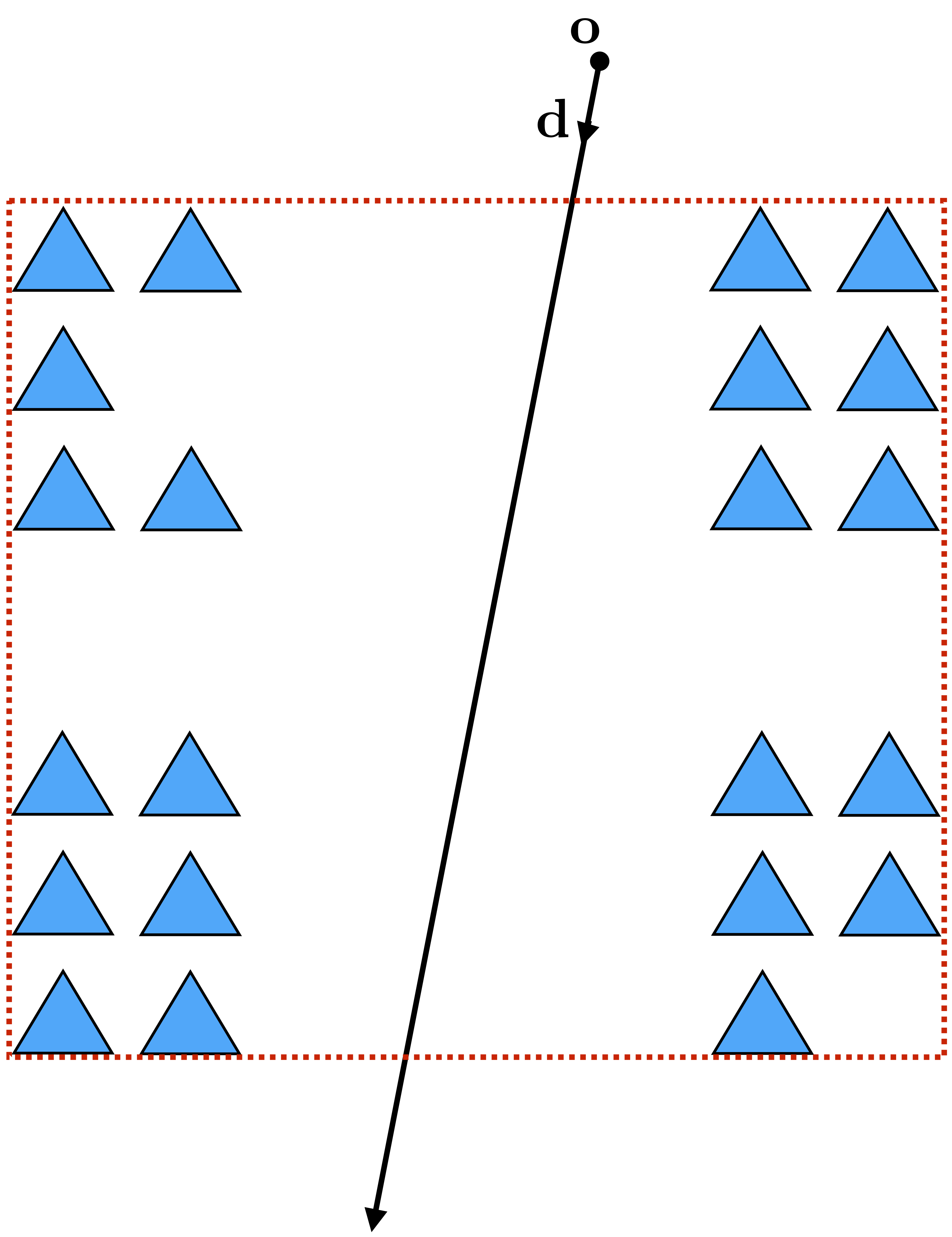


Cost (misses box):

- preprocessing: O(n)
- ray-box test: O(1)
- amortized cost*: O(1)

*over *many* ray-scene intersection tests

Another (should be) simple case



<u>Cost (hits box):</u>

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)!

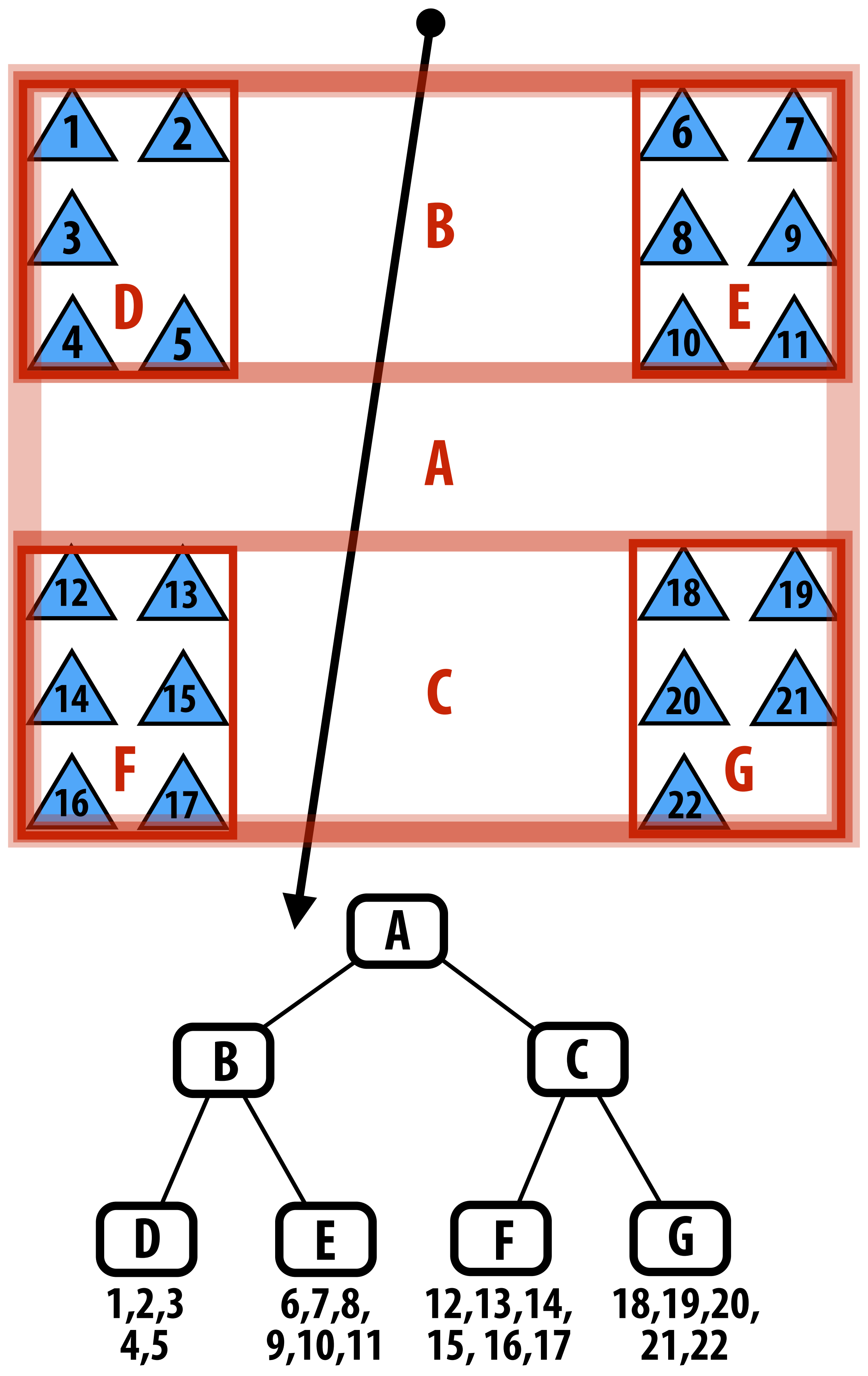
***over *many* ray-scene intersection tests**

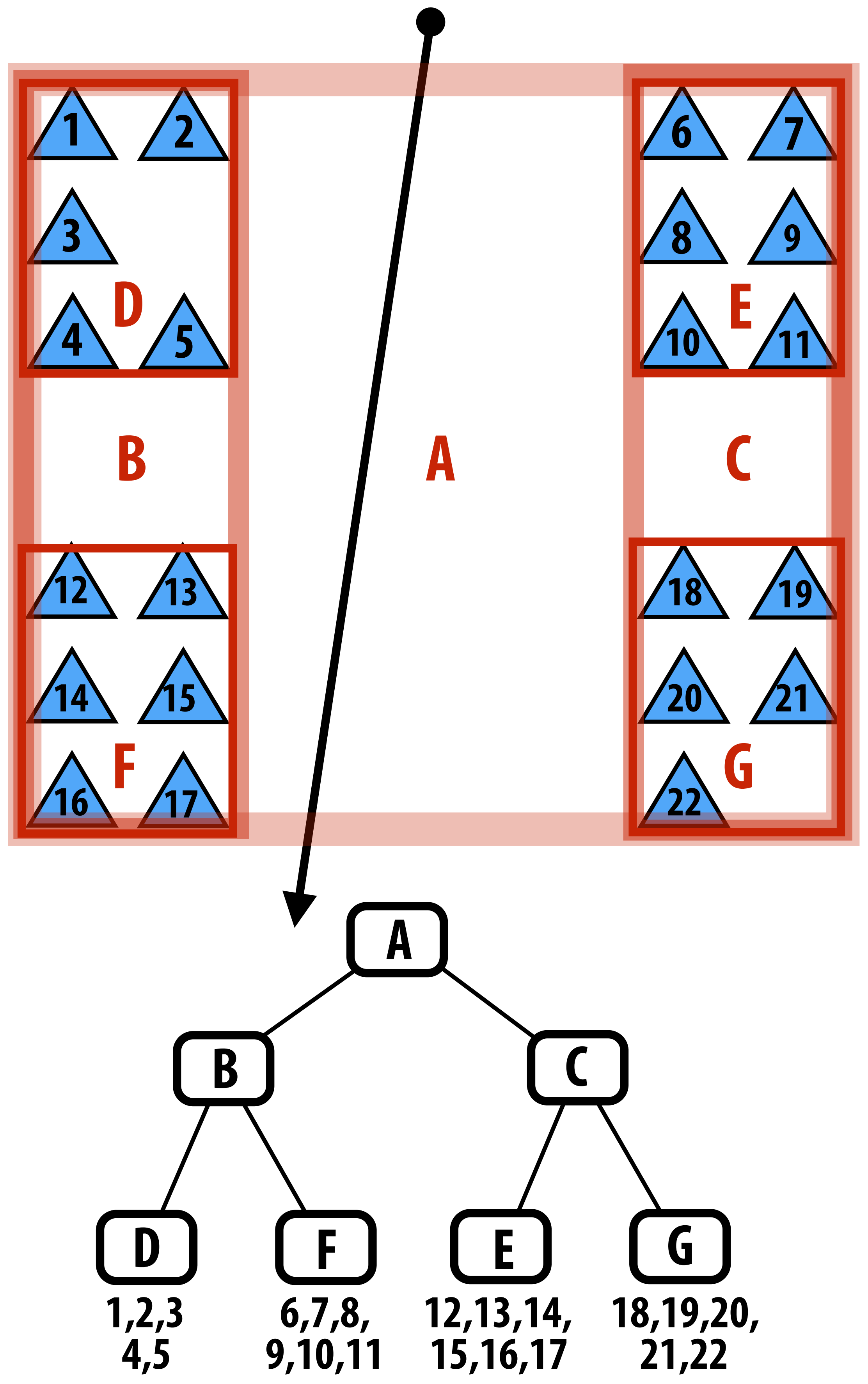
Q: How can we do better?

A: 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



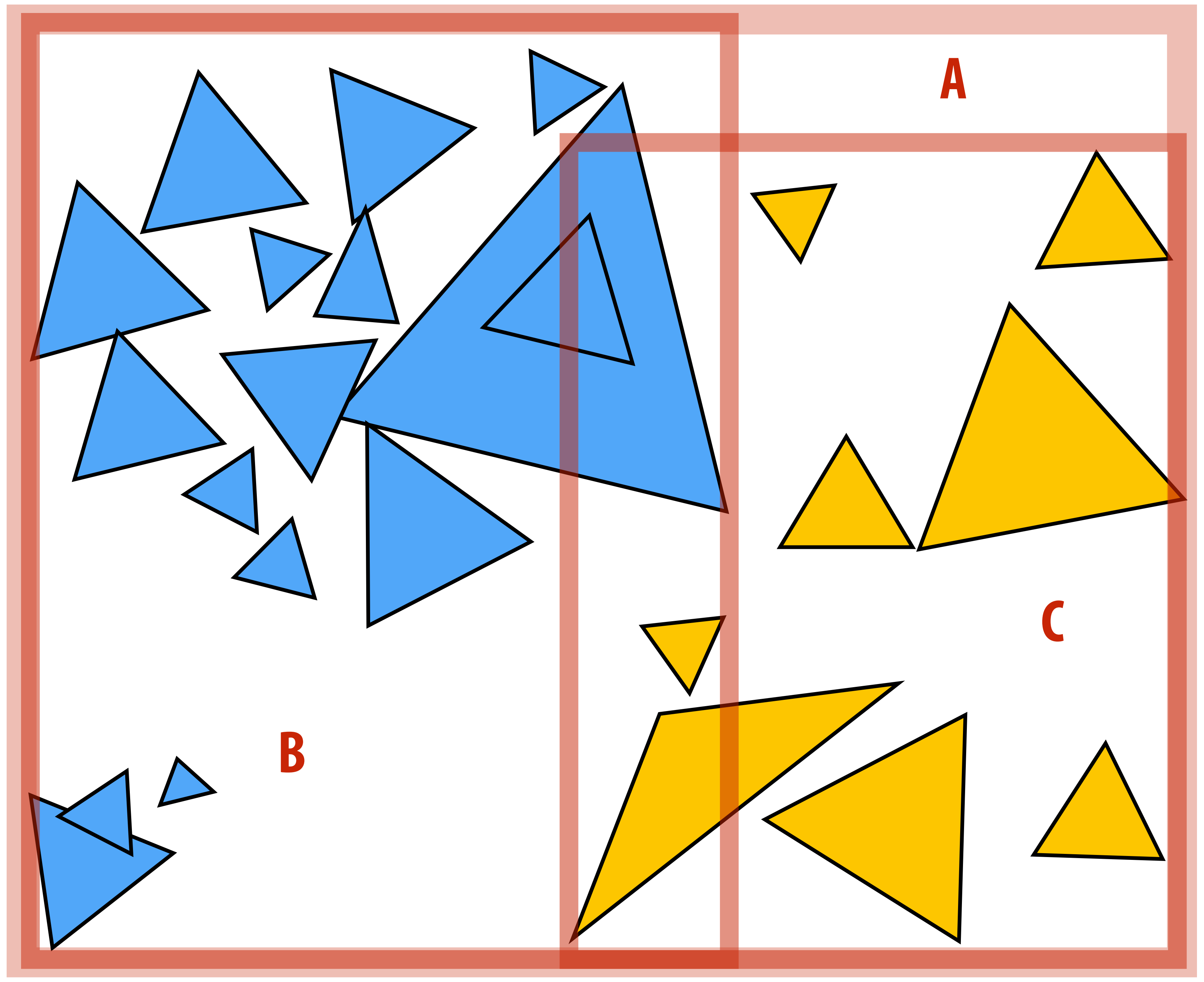


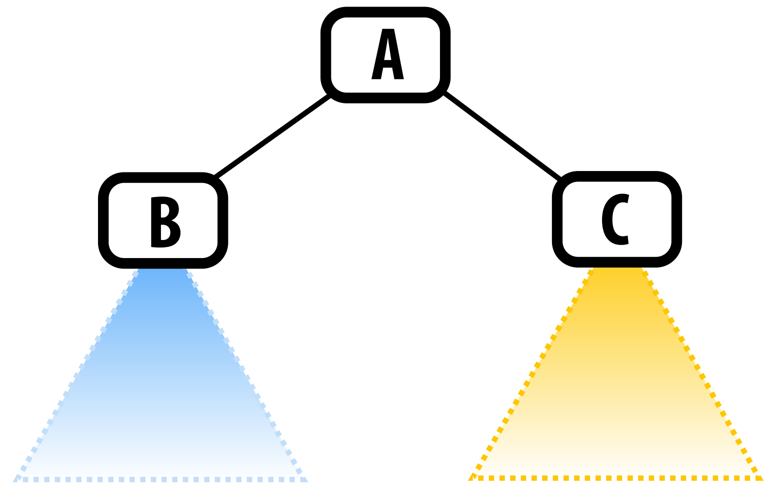
Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Another BVH example

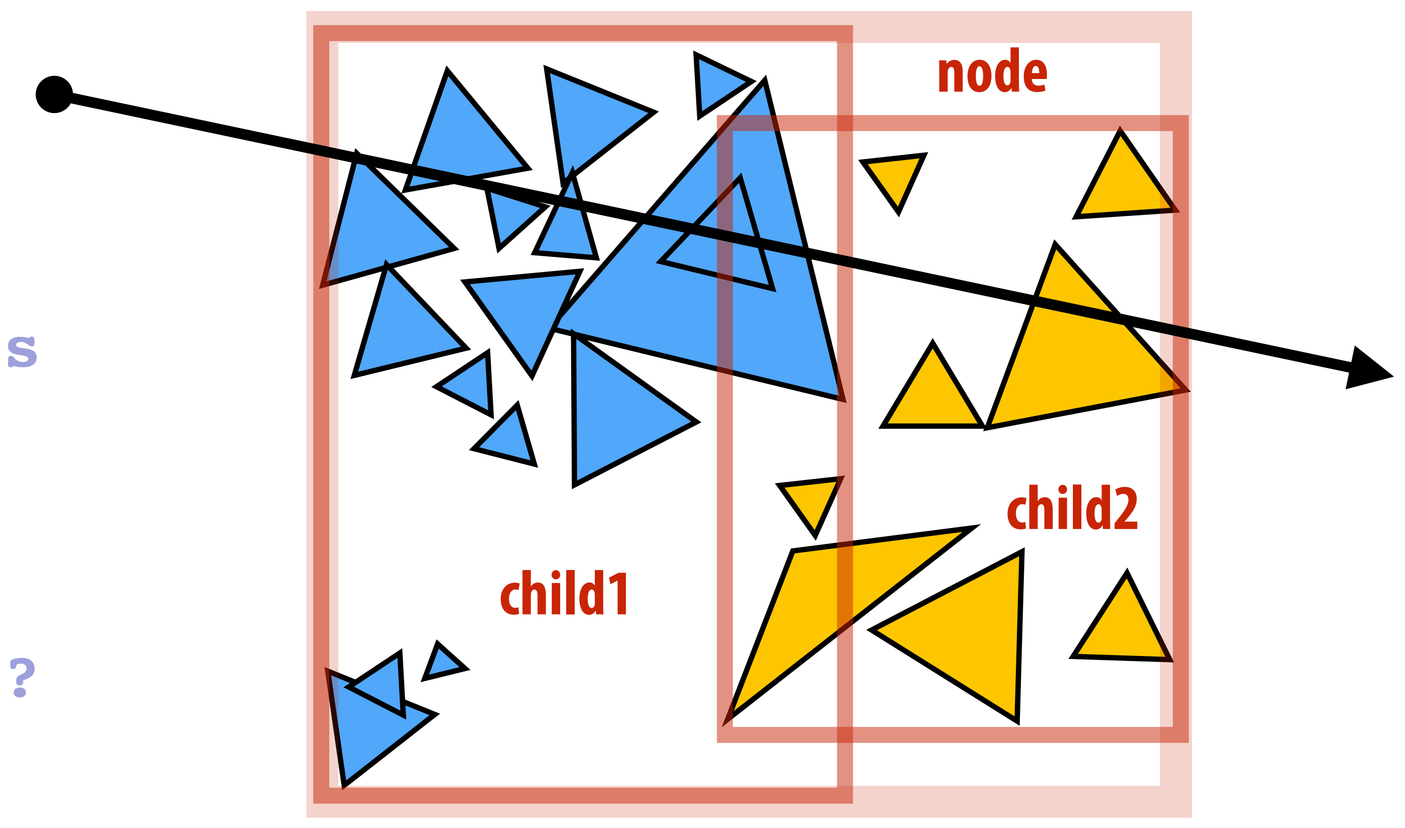
- BVH partitions each node's primitives into disjoints sets
  - Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)





Ray-scene intersection using a BVH

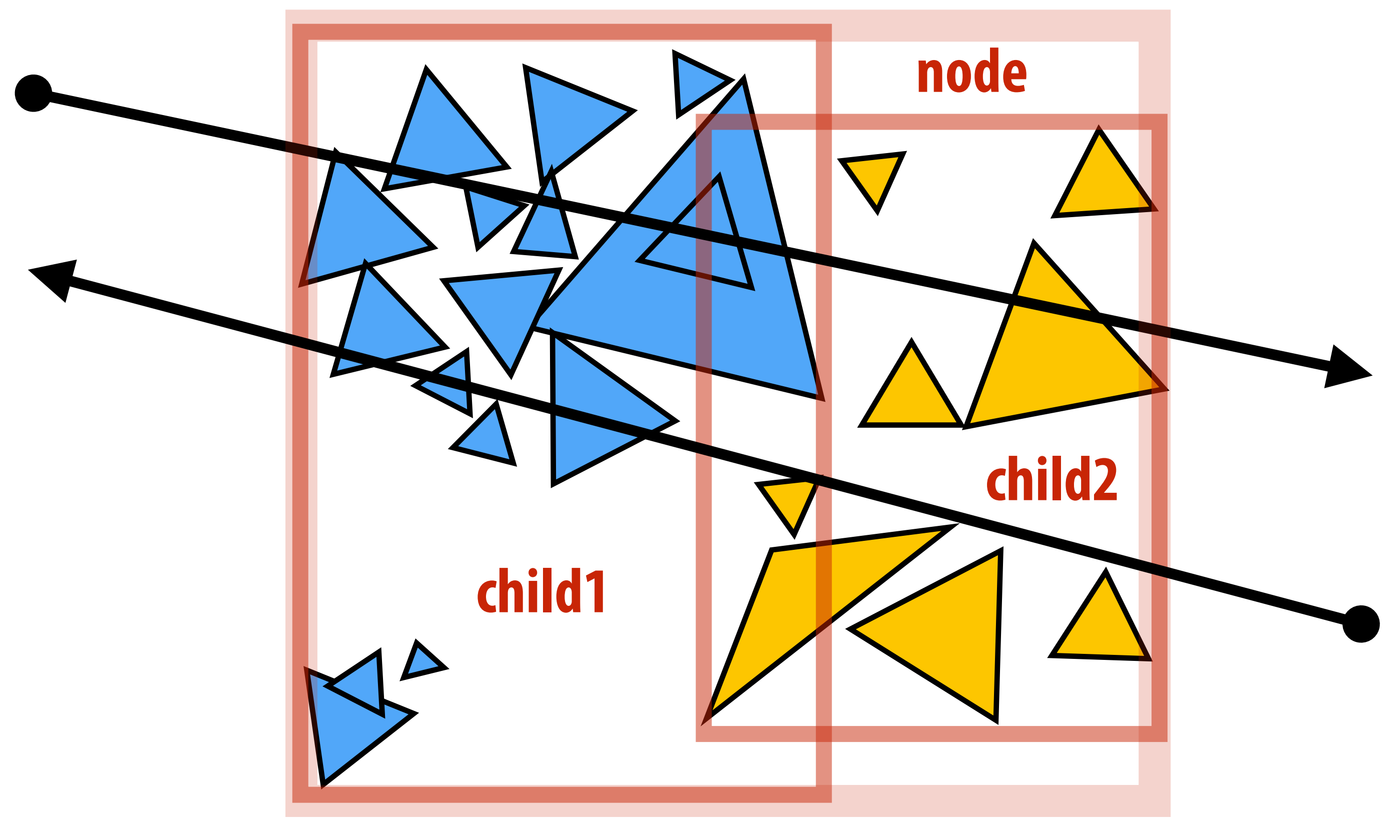
```
struct BVHNode {
   bool leaf;  // am I a leaf node?
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // “left” child (could be NULL)
   BVHNode* child2; // “right” child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

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);
   }}
```



Improvement: “front-to-back” traversal

General strategy for improving performance:

Do traversal in a way that is likely to terminate “early”



```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest)
{
   if (node->leaf) {
      // same as before
   } 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 (hit2.t < 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?

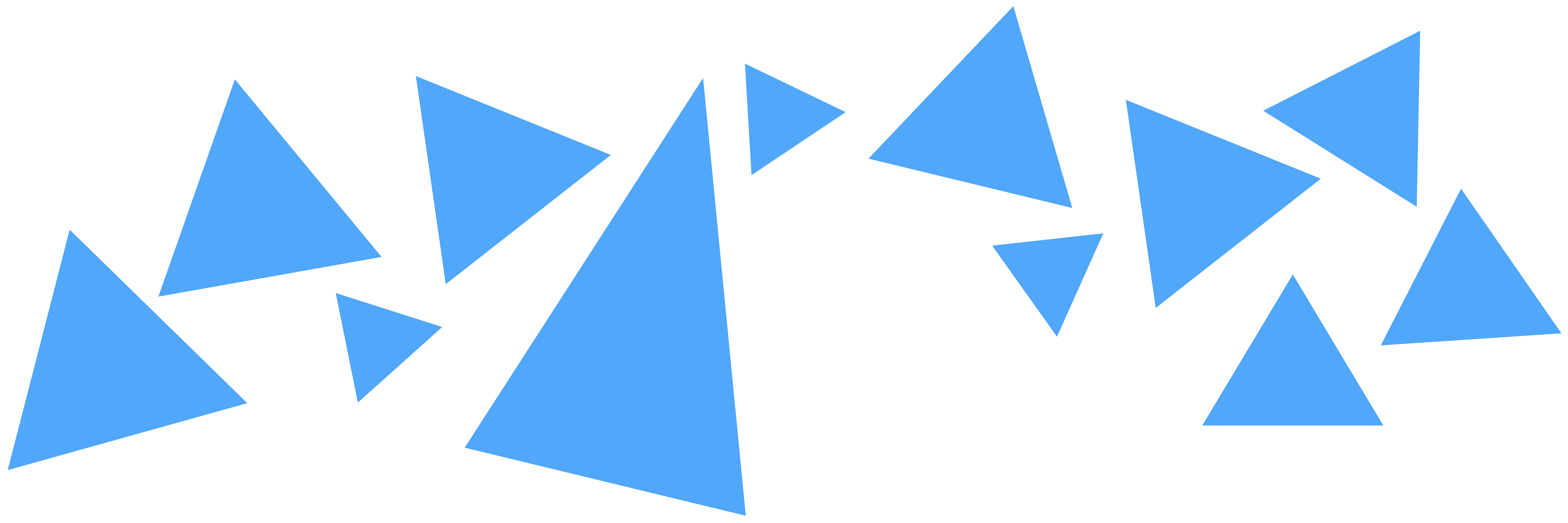
Other strategy for improving performance: Build a “better” BVH!

But for a given set of primitives, there are *many* possible BVHs…

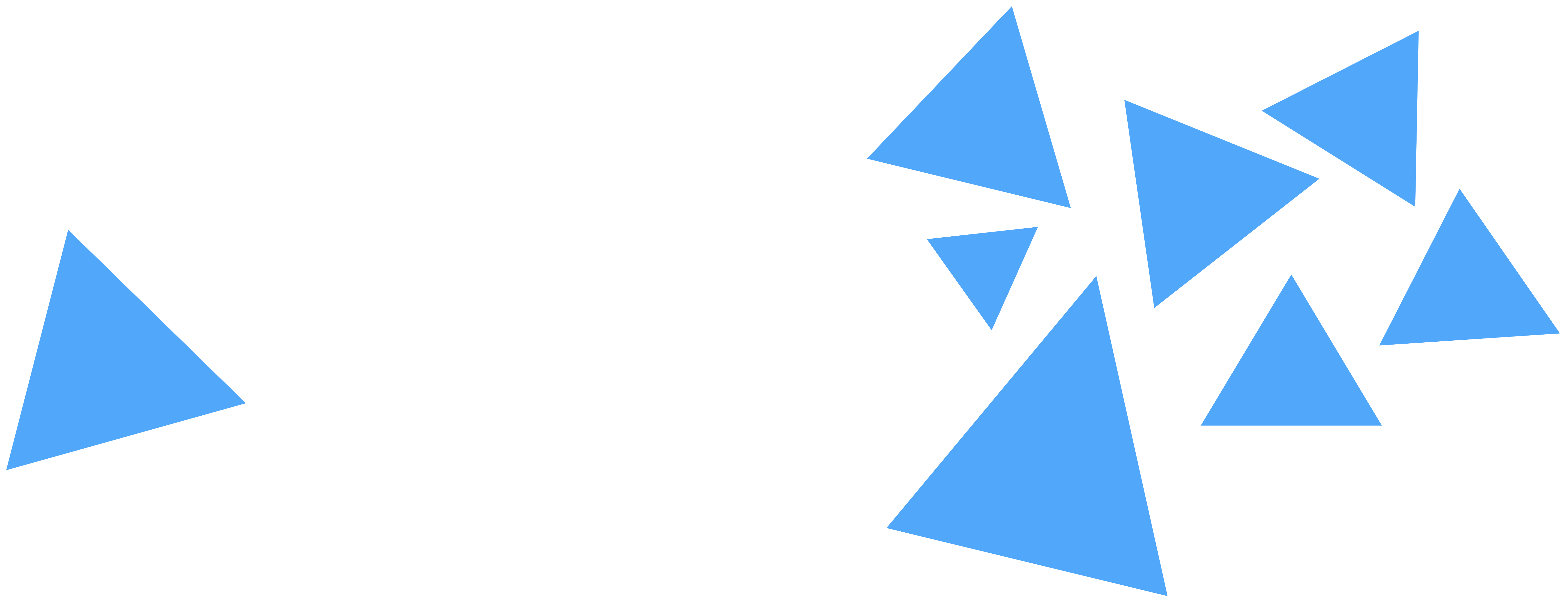
($2^N/2$ ways to partition N primitives into two groups)

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

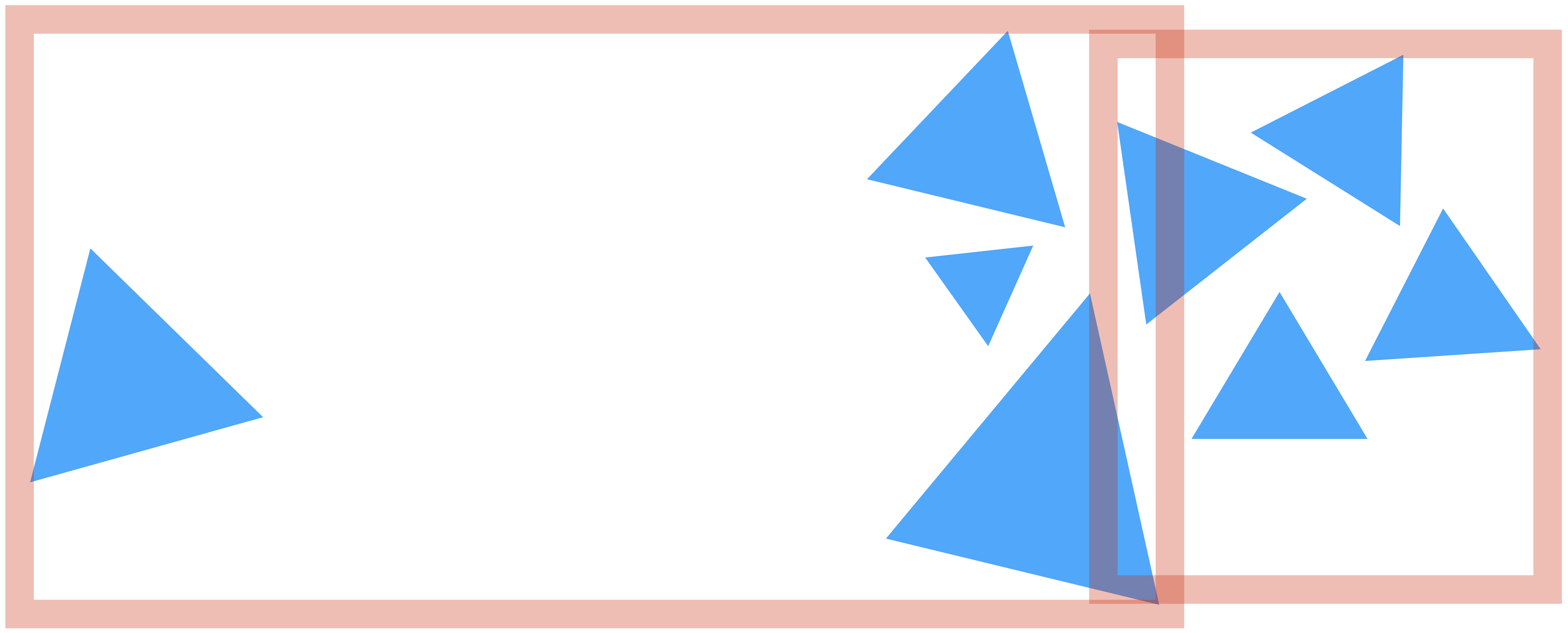
How would you partition these triangles into two groups?



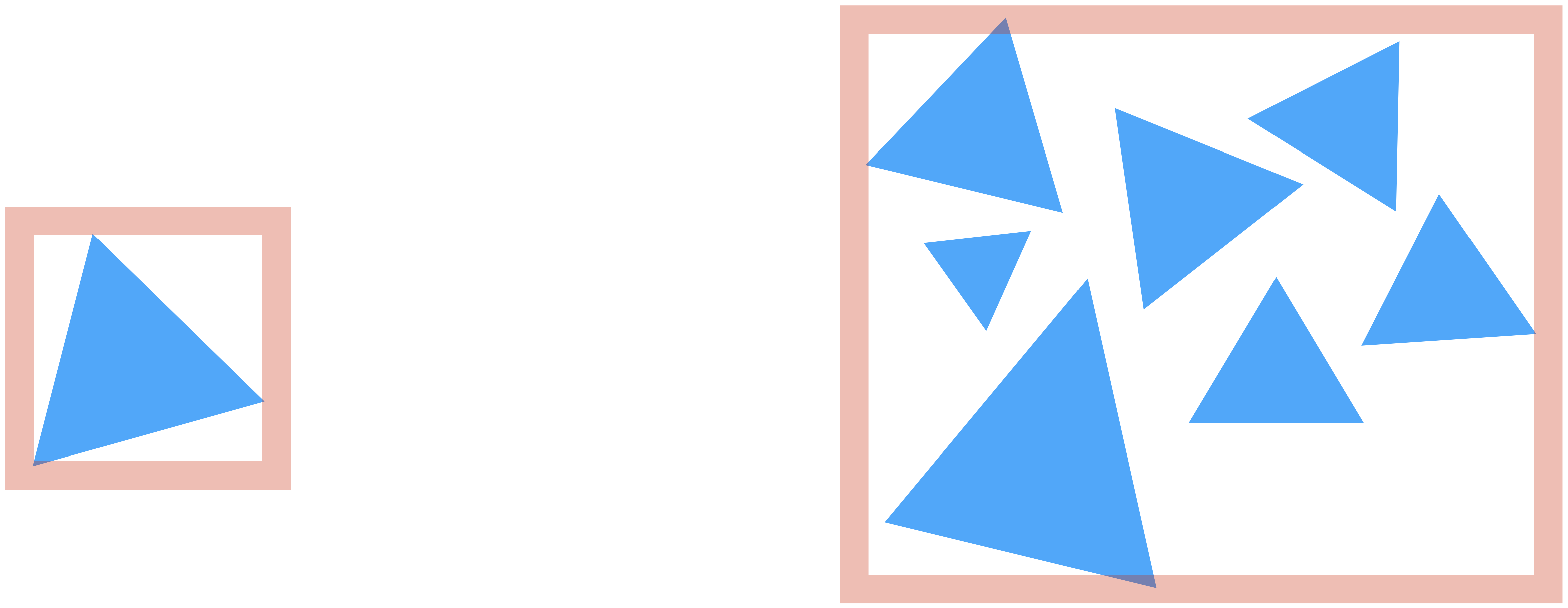
What about these?



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.

EASY CASE—for a leaf node:

$$C = \sum_{i=1}^{N} C_{\text{isect}}(i)$$

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

HARDER CASE—the <u>expected cost</u> of intersecting an interior node, given that the node's primitives are partitioned into child sets A and B:

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

C_{trav} **is the cost of traversing an interior node (e.g., bounding box test)**

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 heuristic for child node costs:

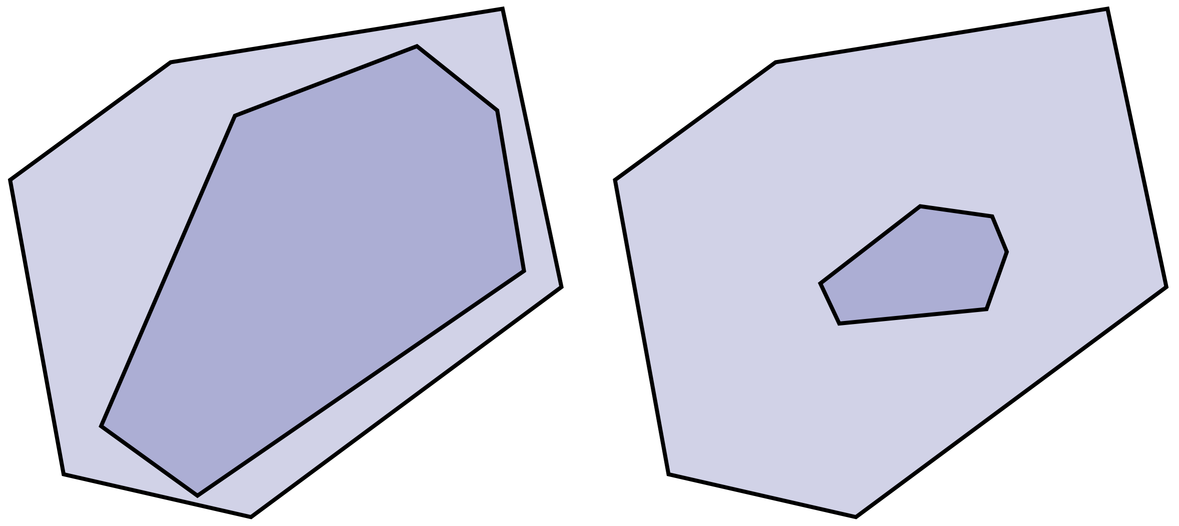
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

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.

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



Leads to surface area heuristic (SAH):

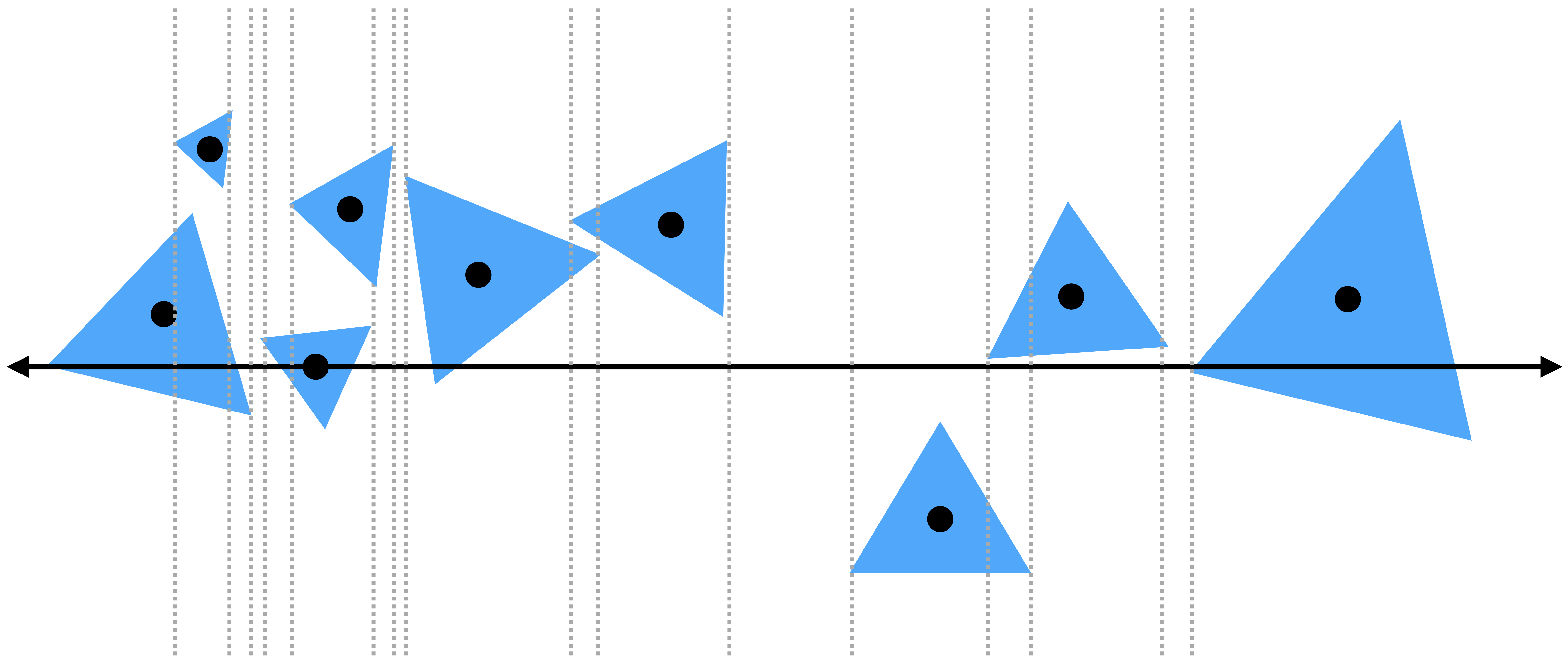
$$C = C_{\mathrm{trav}} + \frac{S_A}{S_N} N_A C_{\mathrm{isect}} + \frac{S_B}{S_N} N_B C_{\mathrm{isect}}$$

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

- Rays are randomly distributed
- No occlusion (i.e., one object blocking another)

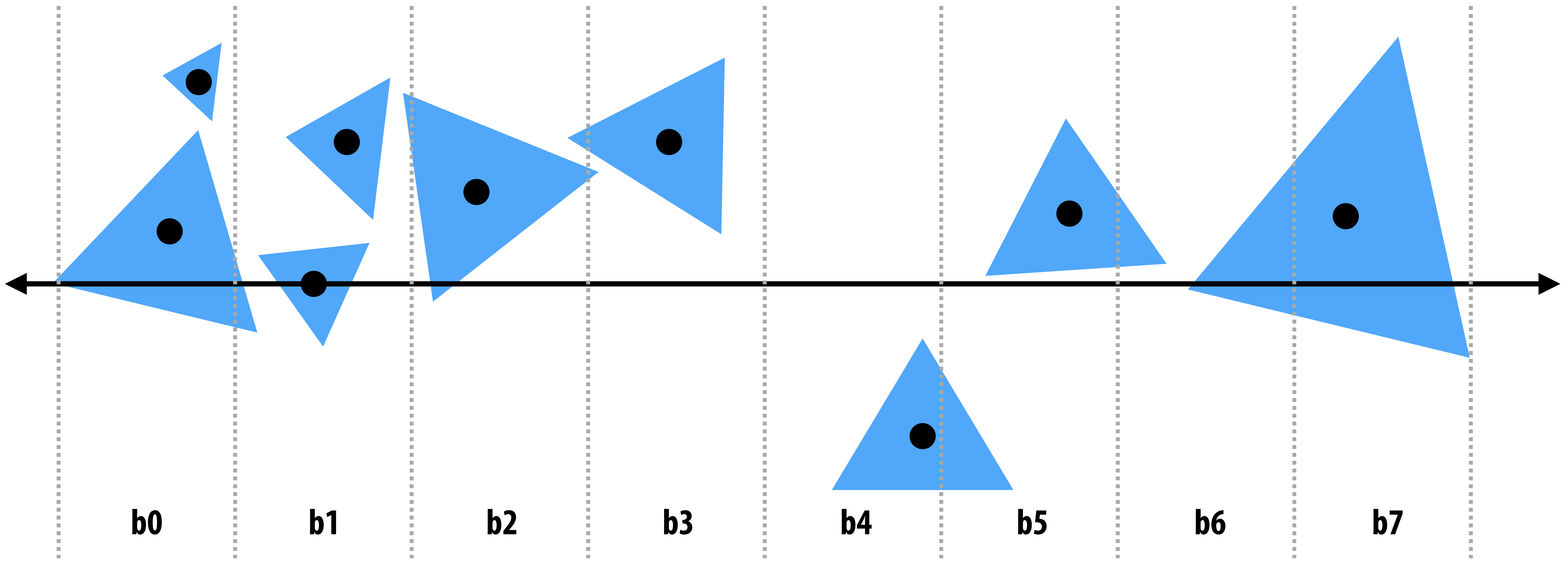
Implementing partitions

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



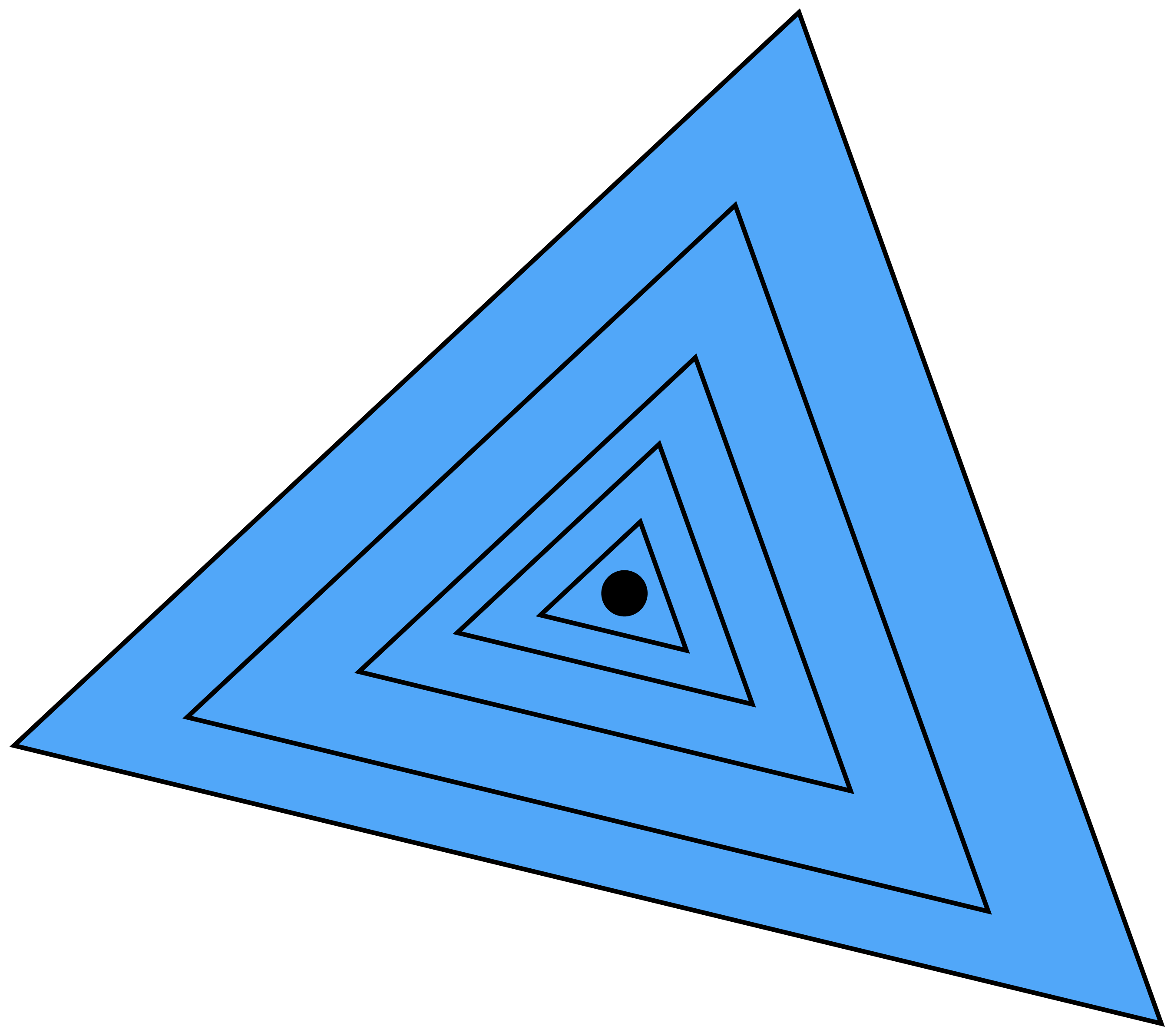
Efficiently implementing partitioning

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

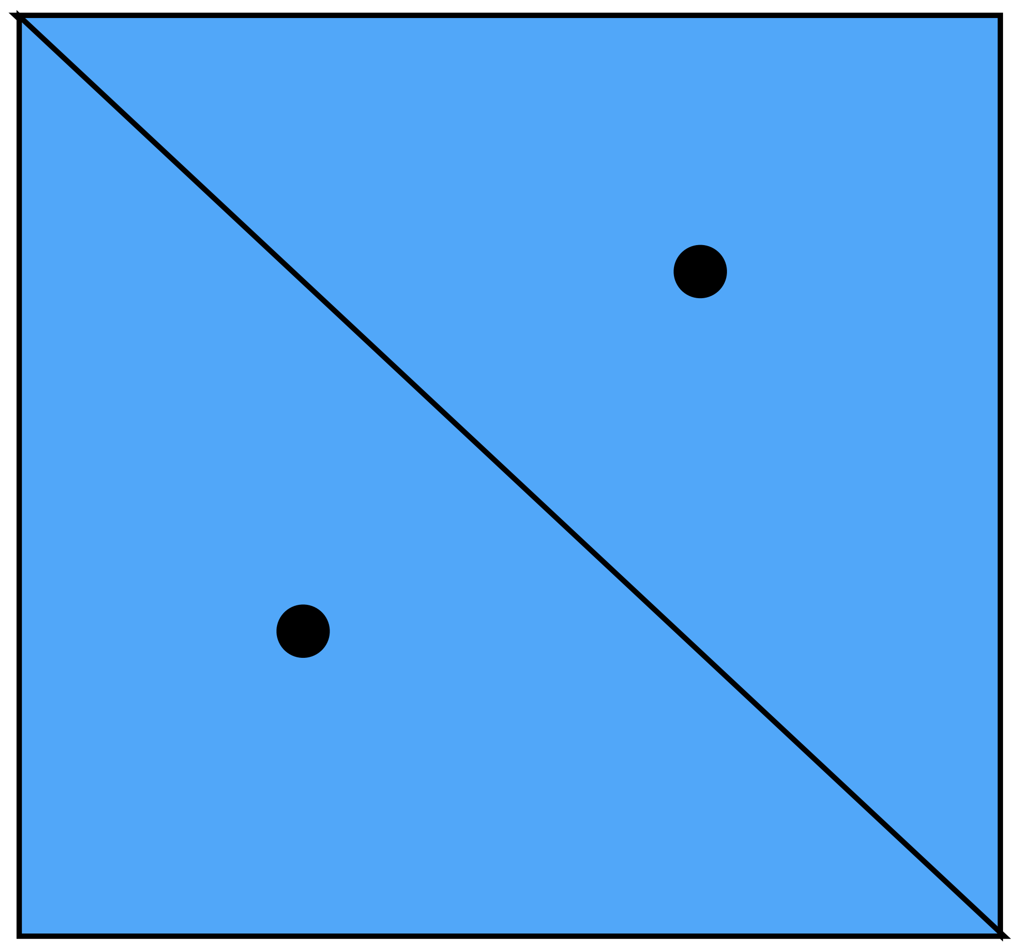


```
For each axis x,y,z:
   initialize buckets
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes
      Evaluate cost, keep track of lowest cost partition
Recurse on lowest cost partition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

Troublesome cases



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

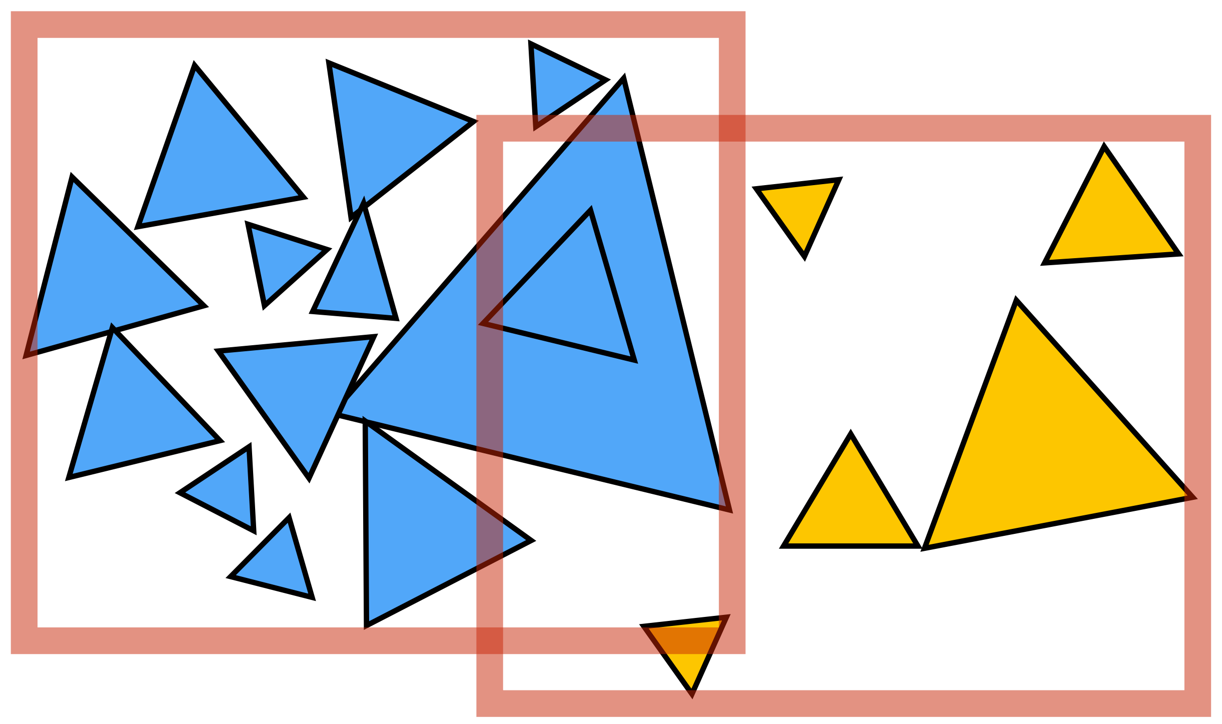


All primitives with same bbox (ray often ends up visiting both partitions)

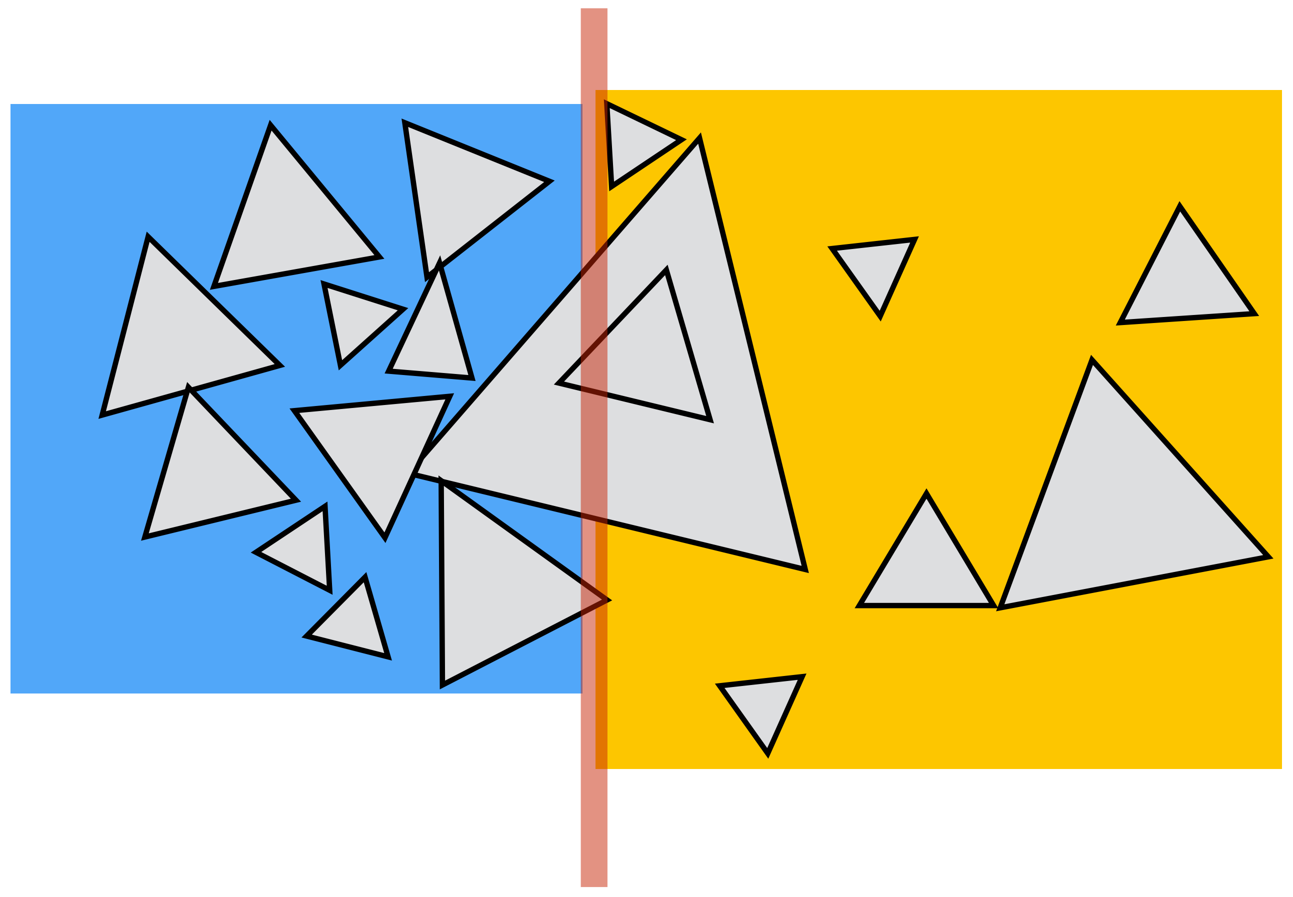
In general, different strategies may work better for different types of geometry / different distributions of primitives…

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)

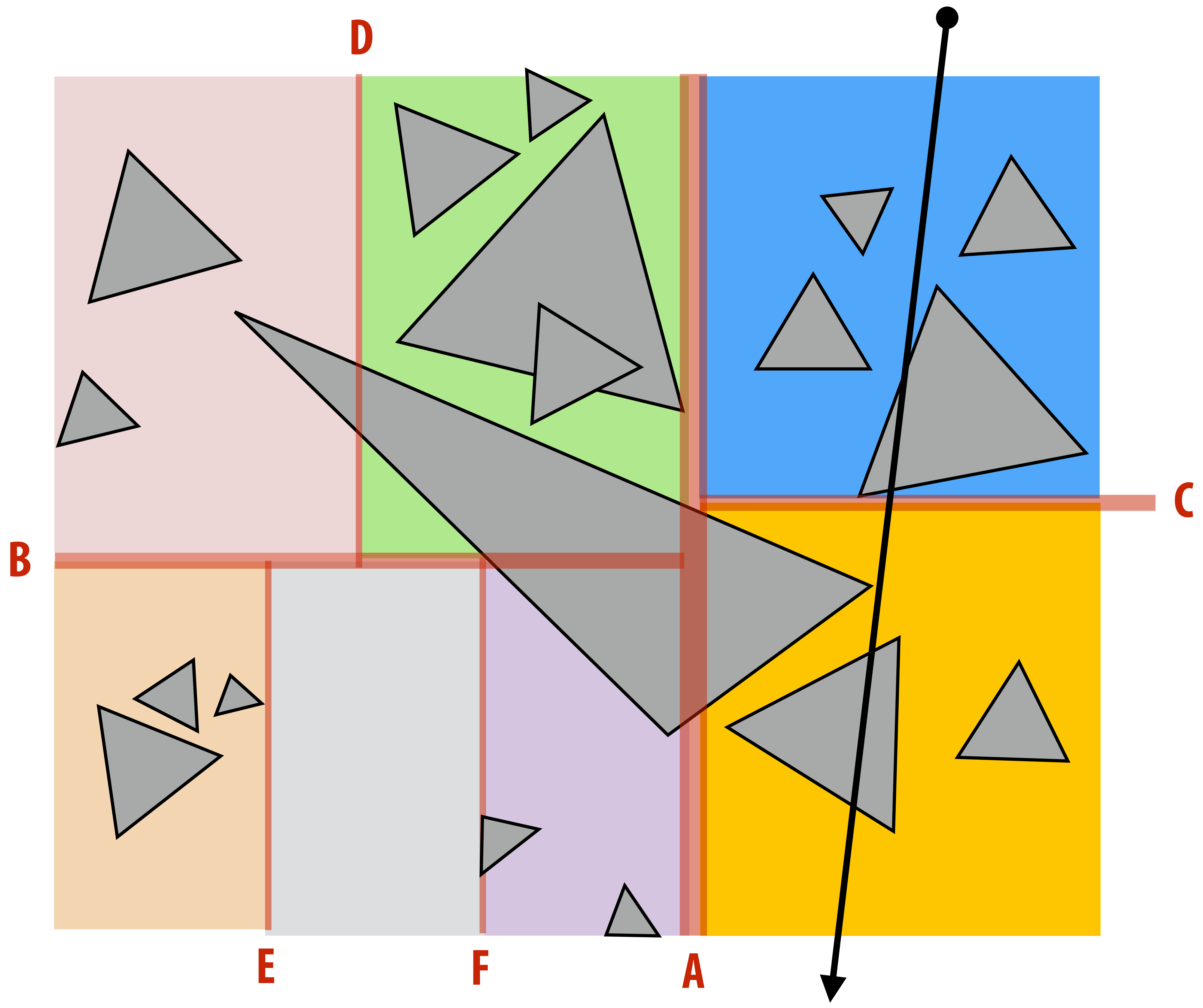


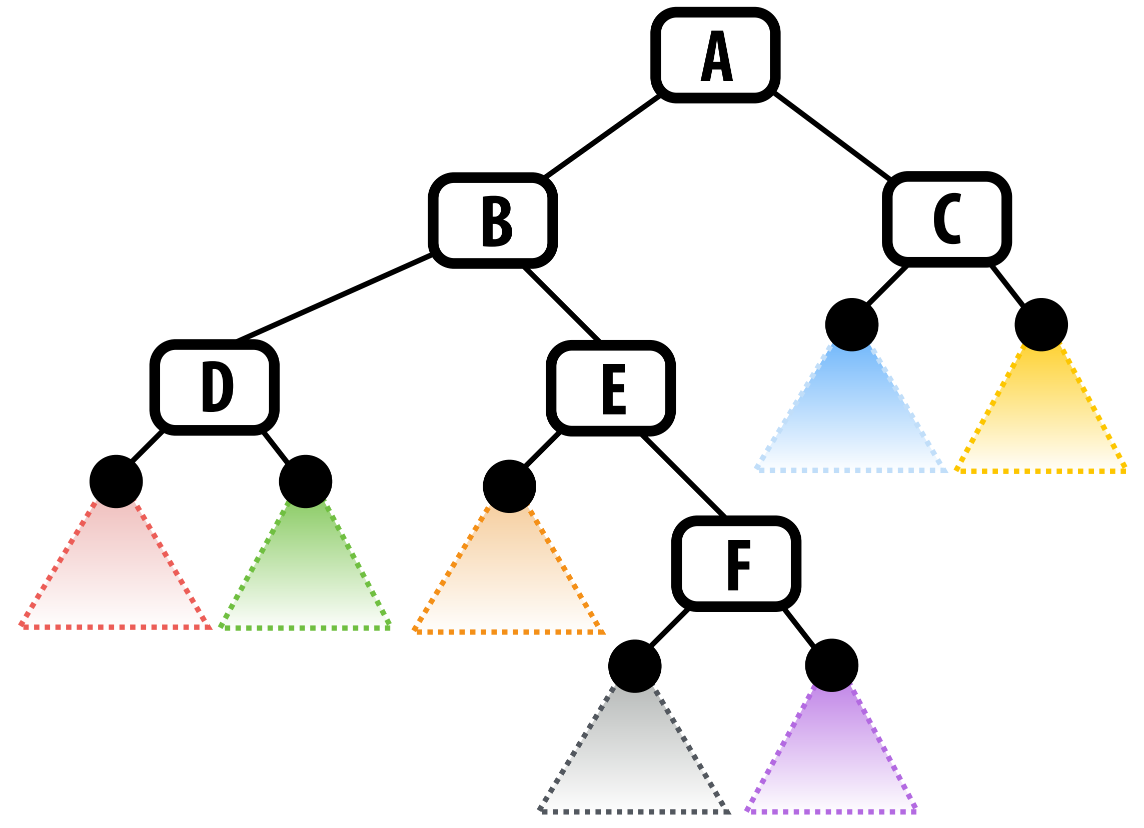
- Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



K-D tree

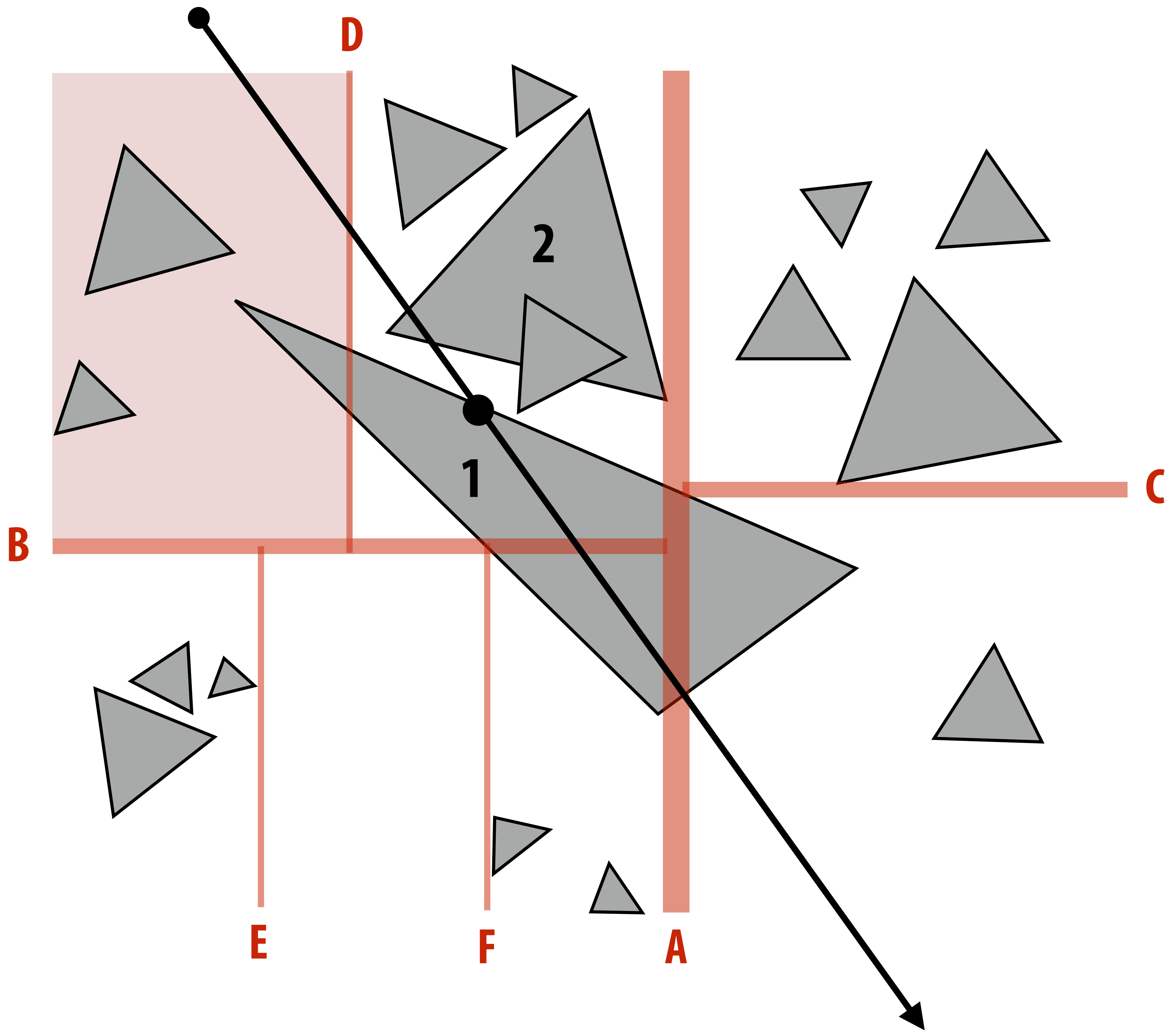
- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in front-to-back order
  - Unlike BVH, can terminate search after first hit is found.

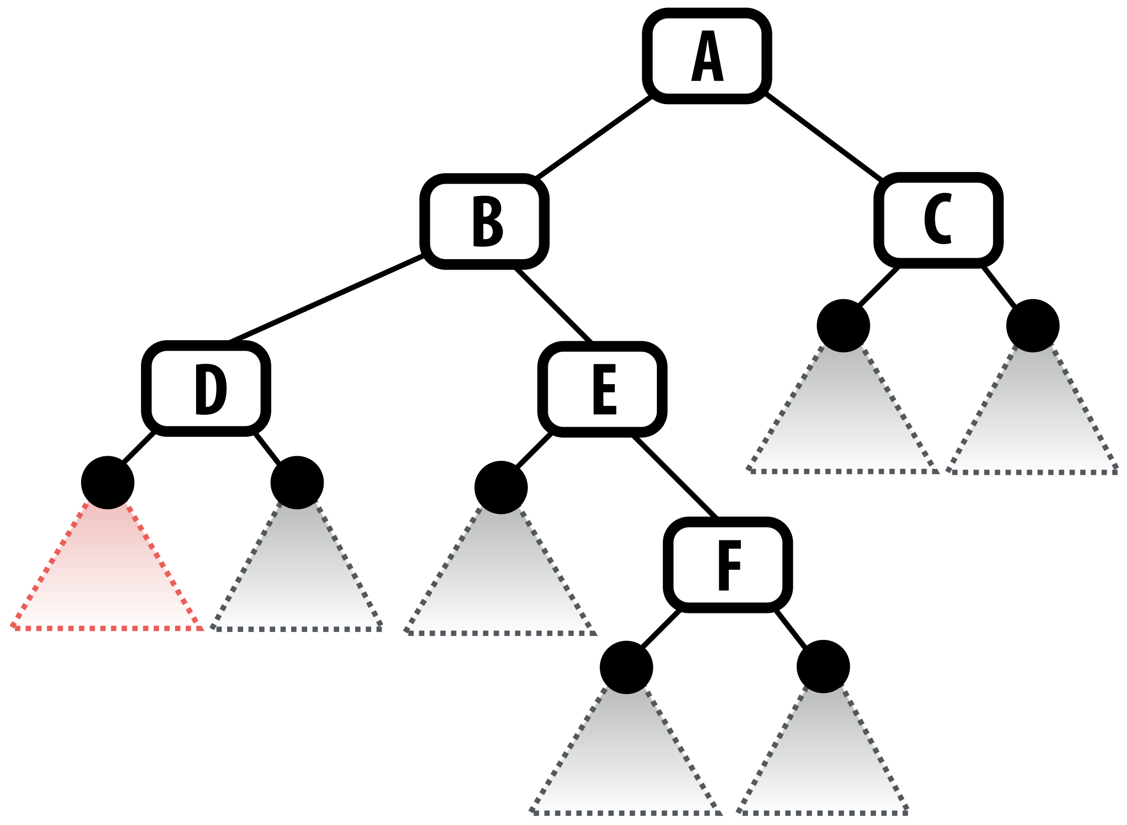




Challenge: objects overlap multiple nodes

- 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.

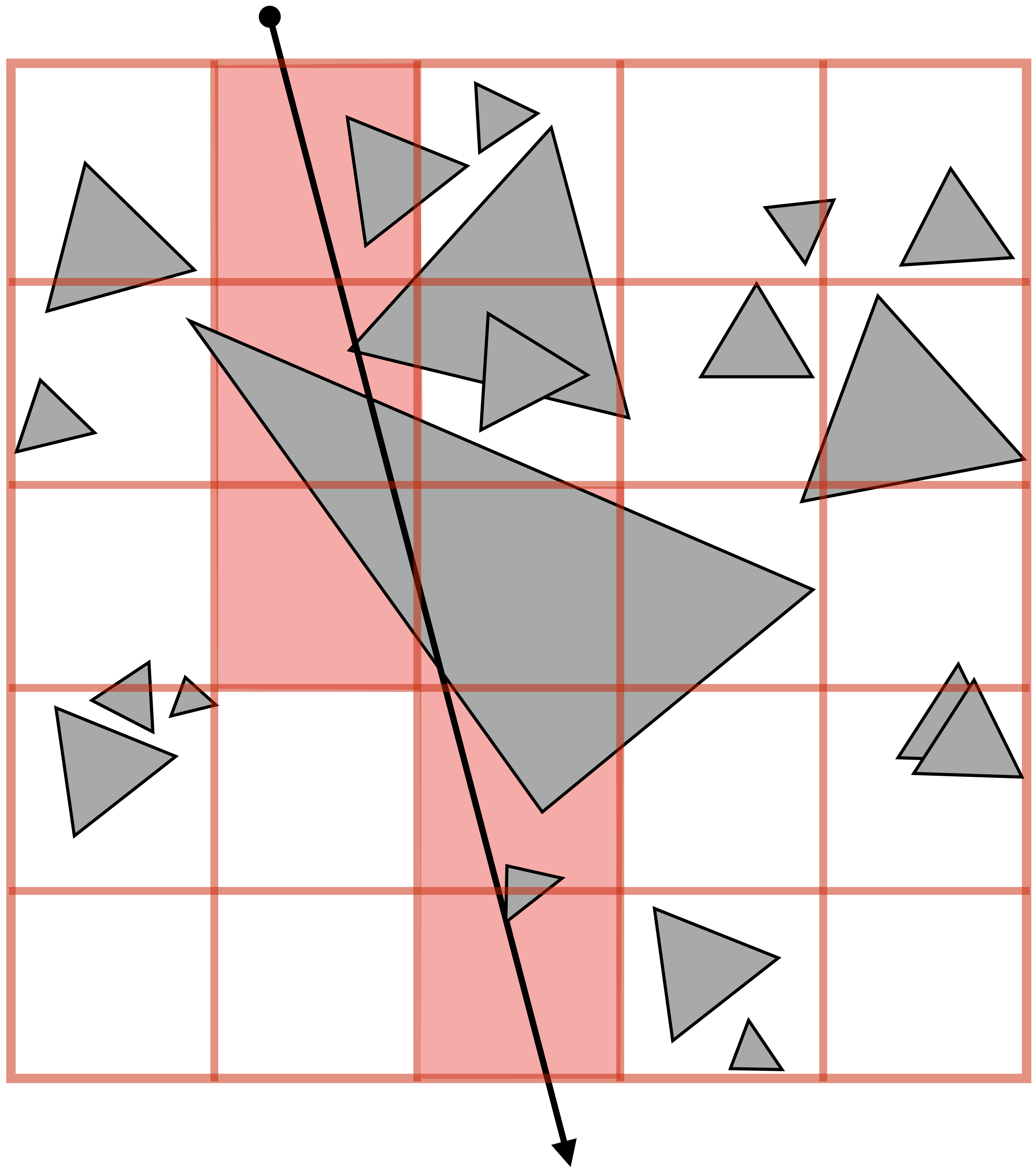
But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

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

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

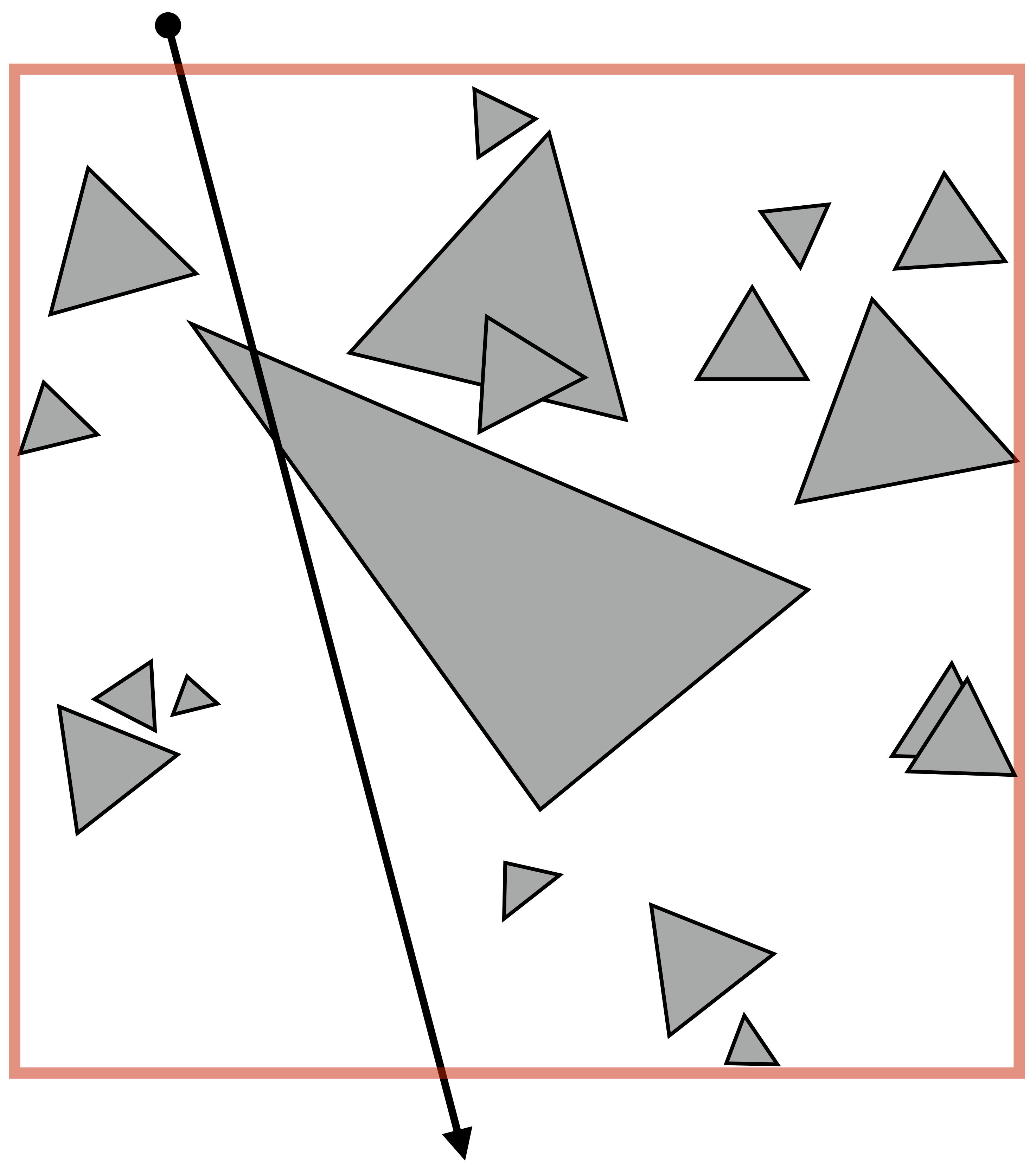
* Caching or "mailboxing" can be used to avoid repeated intersections

Uniform grid

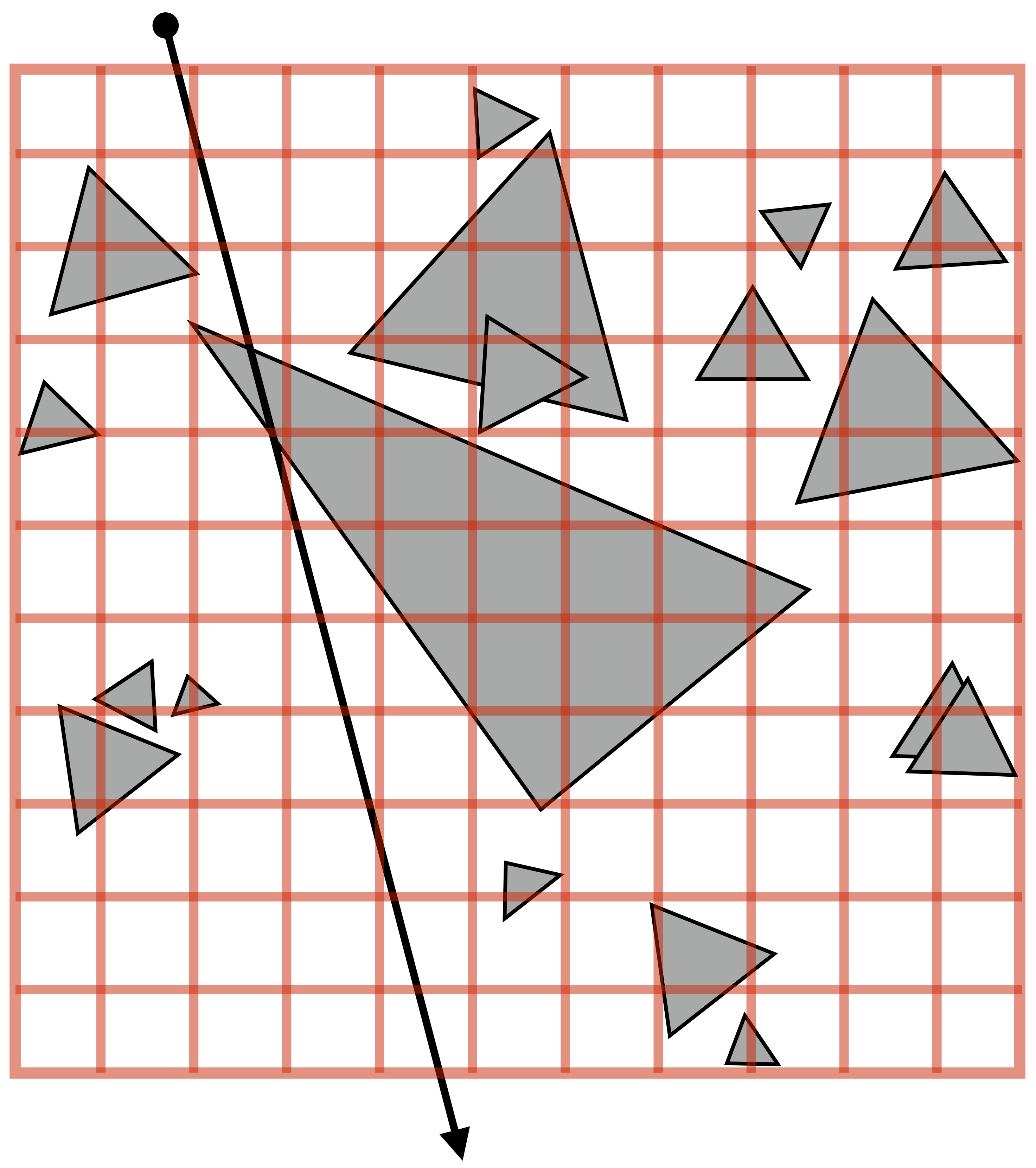


- Partition space into equal sized volumes (volume-elements or "voxels")
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: *3D line rasterization*)
  - Only consider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

What should the grid resolution be?



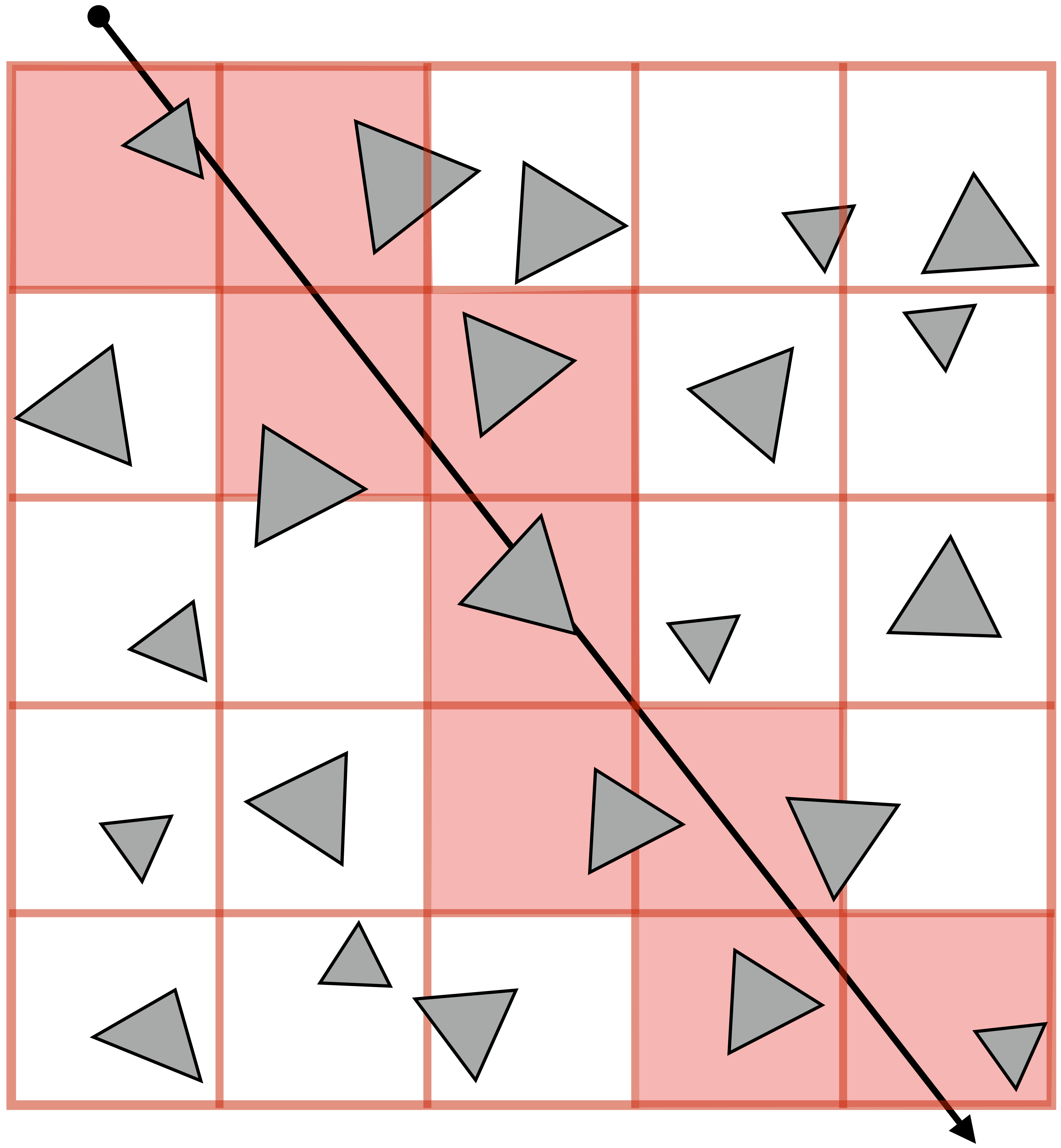
Too few grid cells: degenerates to brute-force approach



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

Heuristic

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



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]

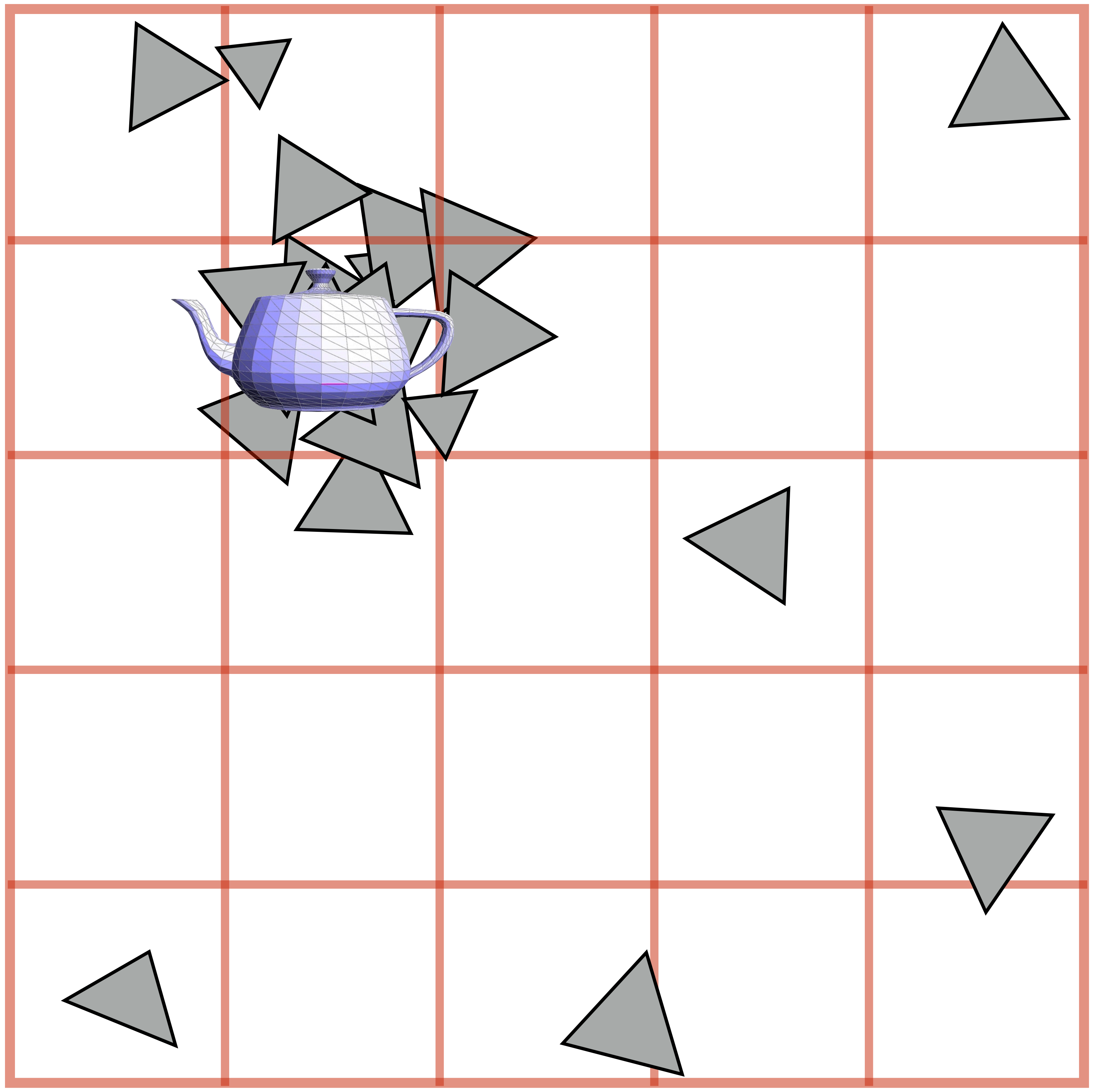
Grass:

[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)

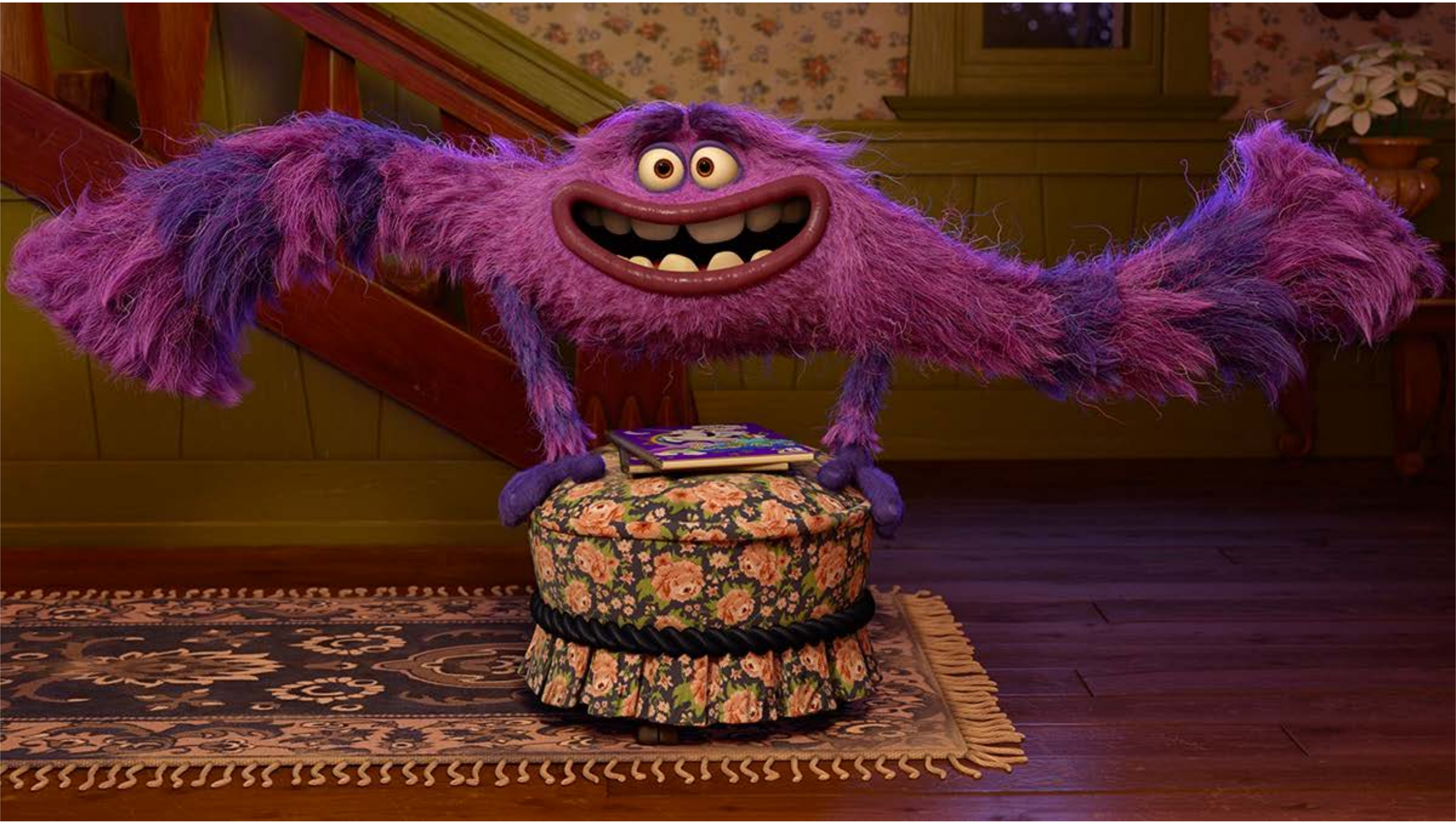


"Teapot in a stadium problem"

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



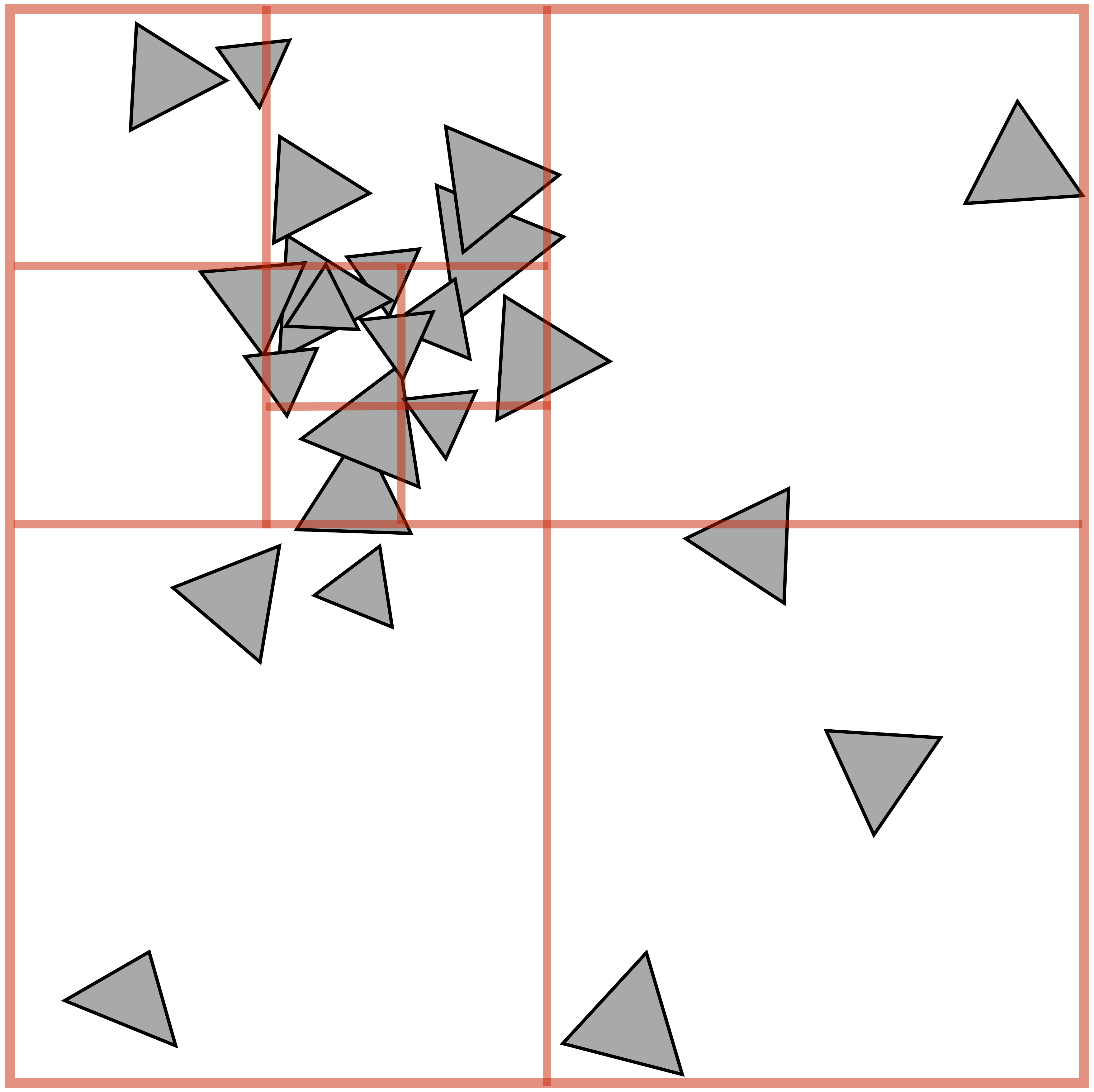
[Image credit: Pixar]

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

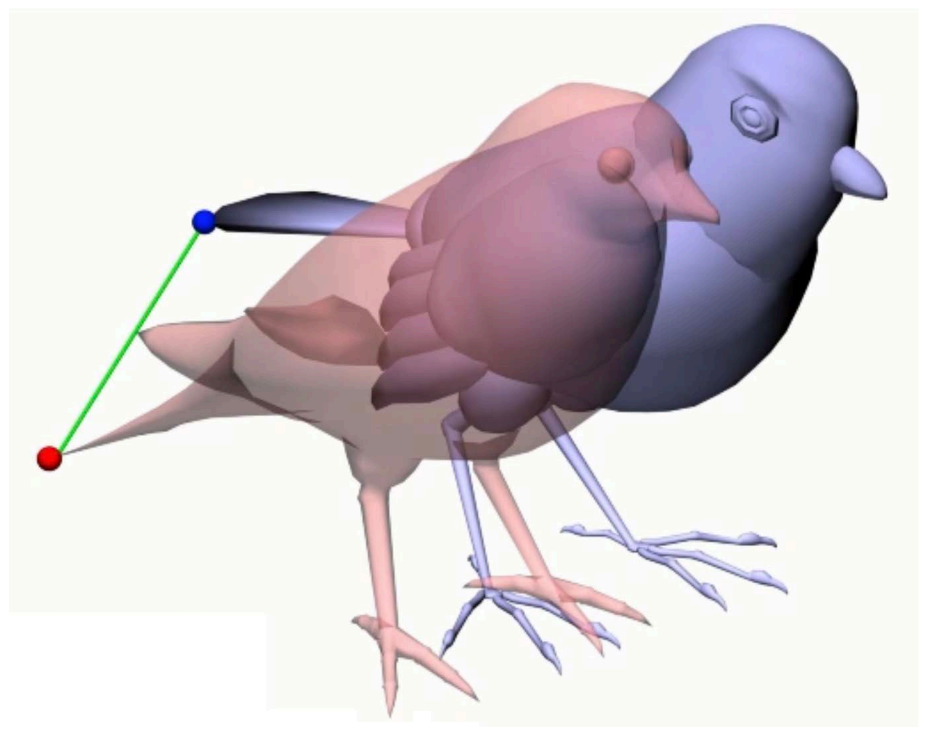
- Adaptive structures (BVH, K-D tree)
  - More costly to construct (must be able to amortize cost over many geometric queries)
  - Better intersection performance under non-uniform distribution of primitives

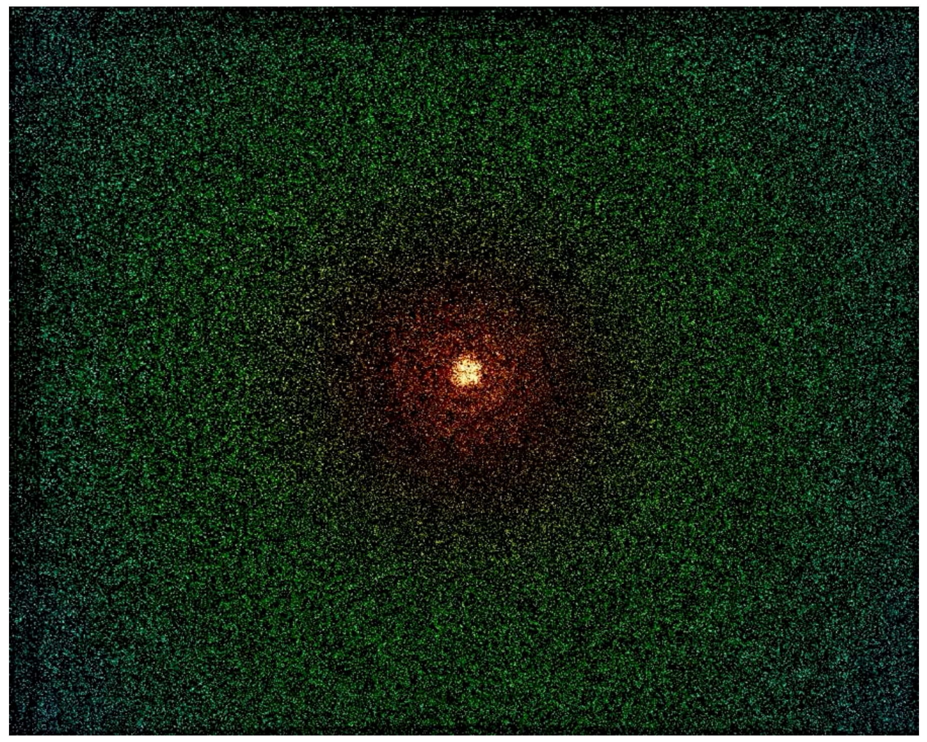
- Non-adaptive accelerations structures (uniform grids)
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed

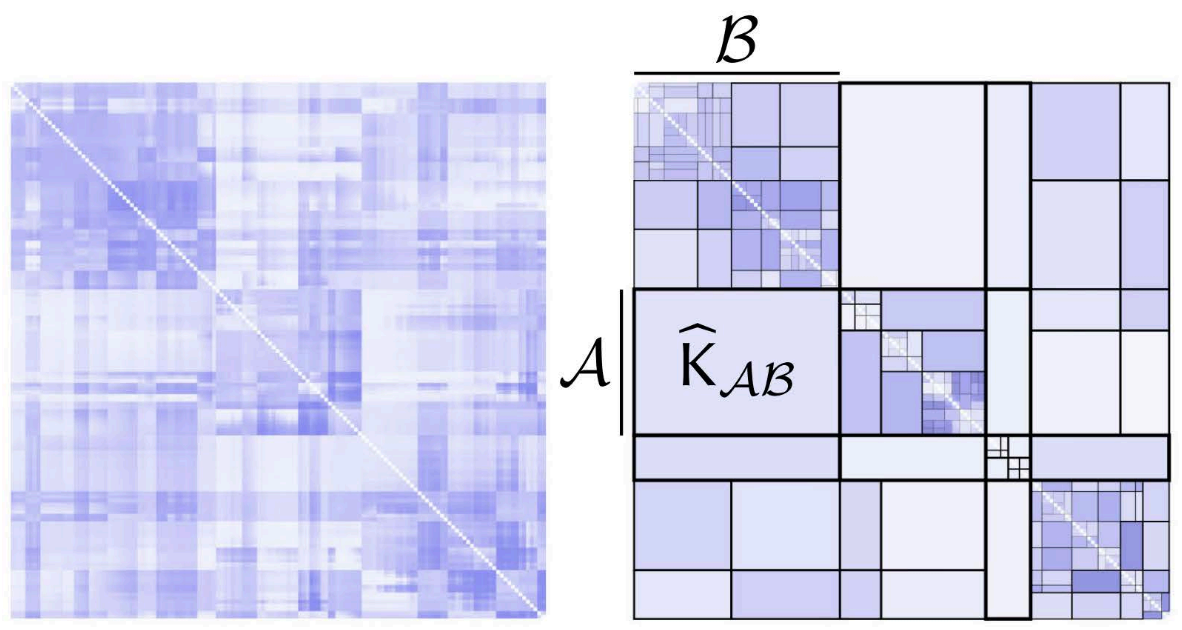
- Many, many combinations thereof…

Hierarchical Acceleration in Graphics

- GEOMETRY
  - Inside-outside tests (e.g., meshing)
  - Closest point tests (e.g., Hausdorff distance)
- ANIMATION/SIMULATION
  - “Particle systems”
  - N-body dynamics, fluid simulation, …
  - Barnes-Hut algorithm
  - fast multipole method
- RENDERING
  - Visibility
  - Physically-based ray tracing

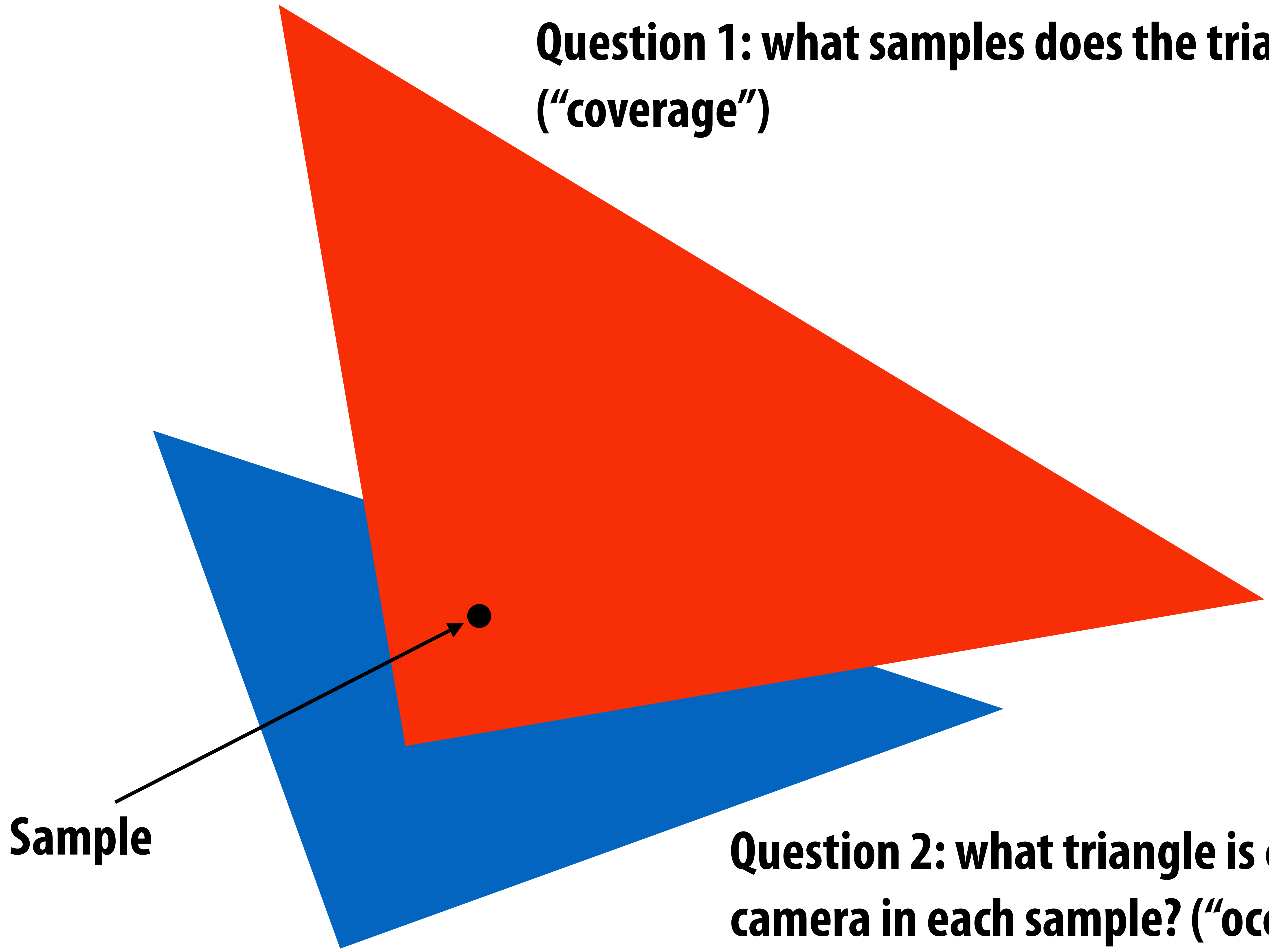






Q: How can we use ray intersection queries to generate an image?

Recall triangle visibility problem:



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

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

Before, we solved this problem using rasterization + depth buffering

But we can also do it via ray queries!

Basic rasterization algorithm

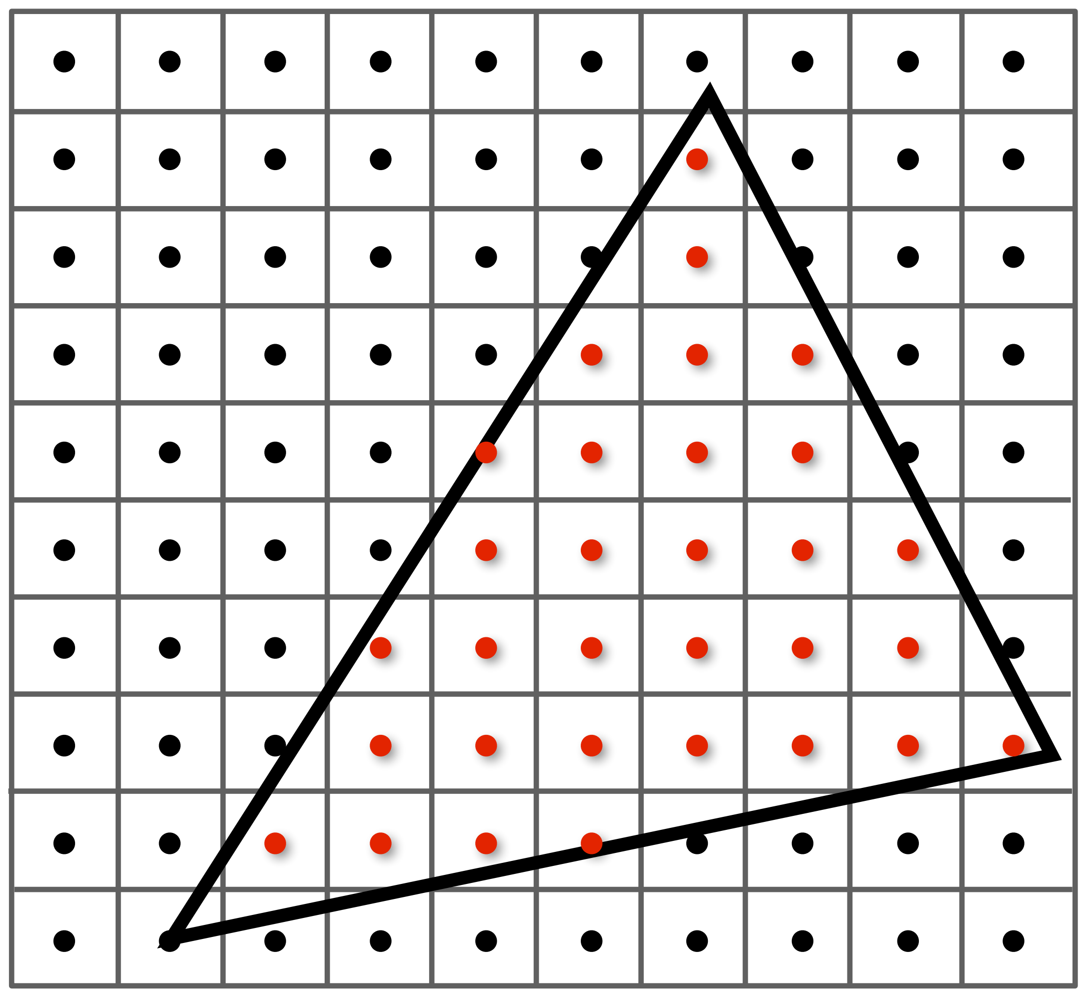
"For each triangle, find the samples it covers"

Sample = 2D point

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

Occlusion: depth buffer

```
initialize z_closest[] to ∞.                 // 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]
```



Basic ray casting algorithm

"For each sample, find the primitives it's covered by"

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 = ∞                                      // 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
```

Both schemes use further acceleration:

RASTERIZATION — limit tests to bounding box of triangle

RAY TRACING — use hierarchical acceleration (as we saw today!)

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
- **High-performance implementations embody similar techniques:**
  - Hierarchies of rays/samples
  - Hierarchies of geometry
  - Deferred shading
  - …

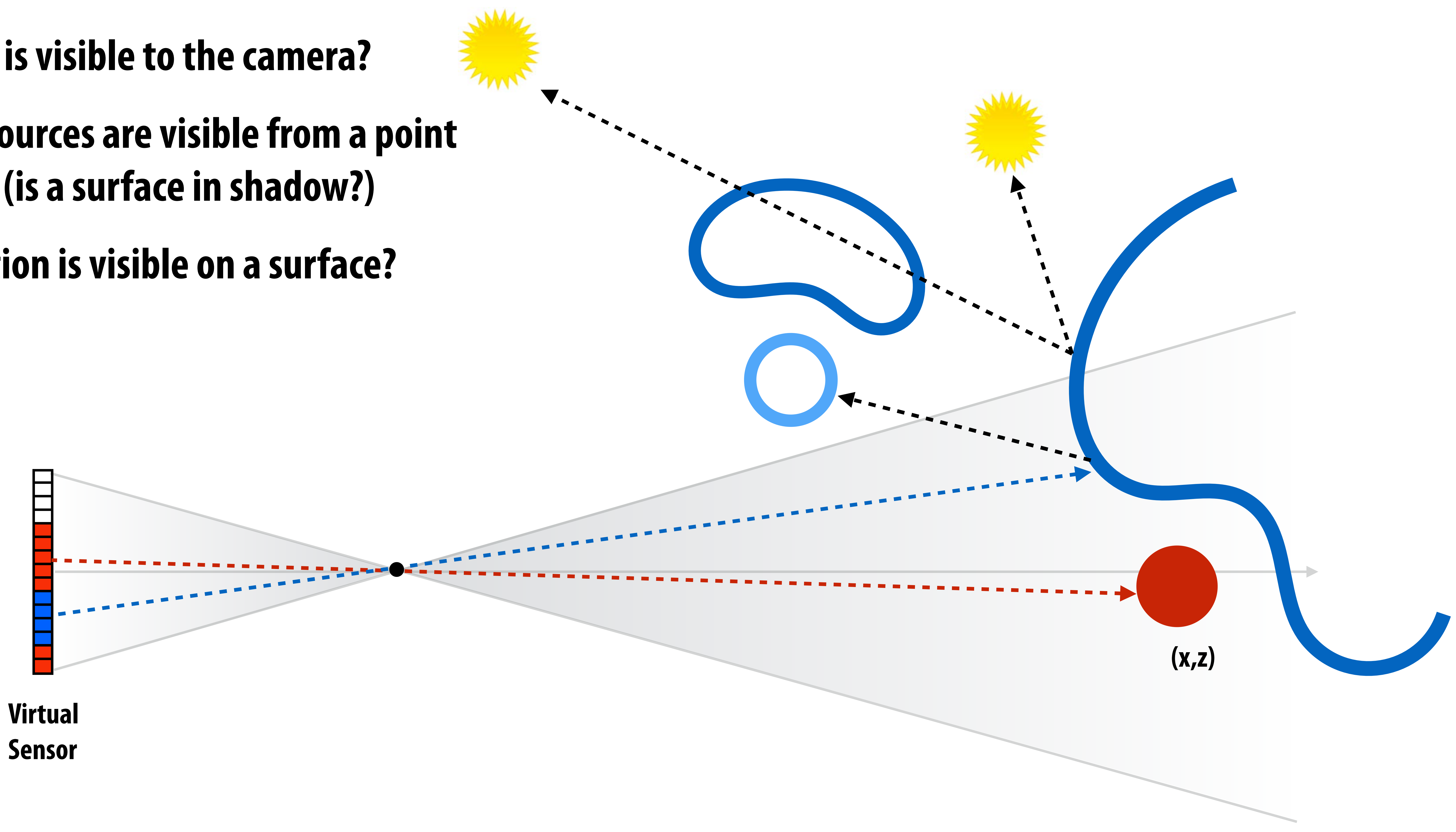
There is an important difference…

Ray casting can be used for many tasks:

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: Color