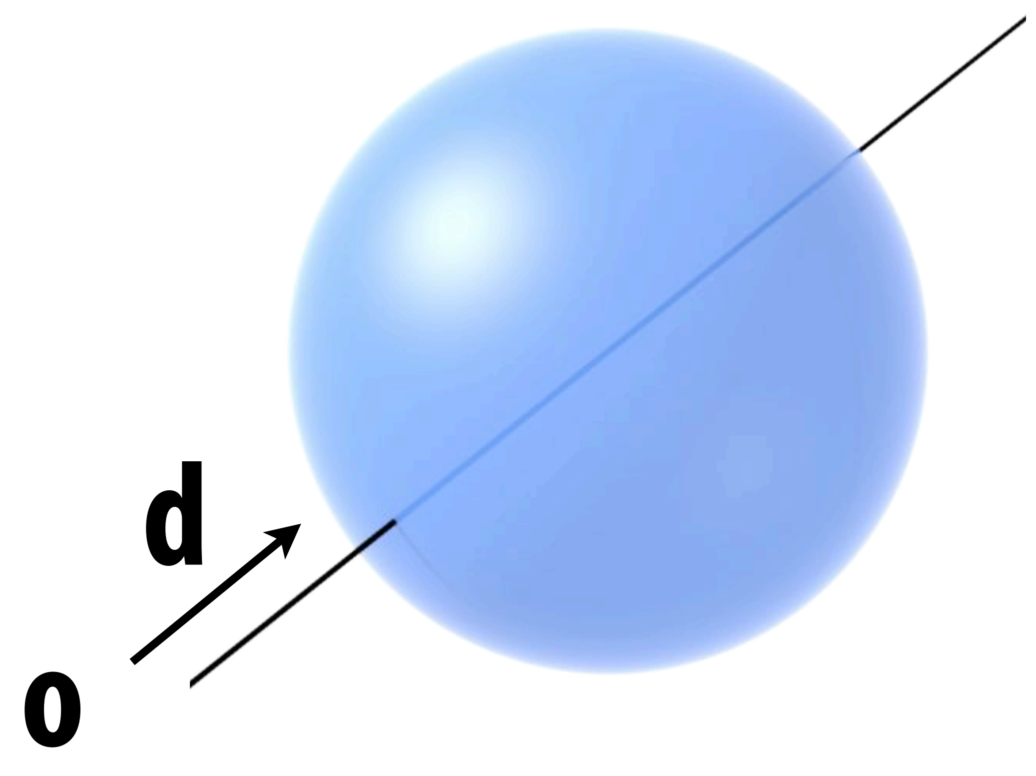
Lecture 9:

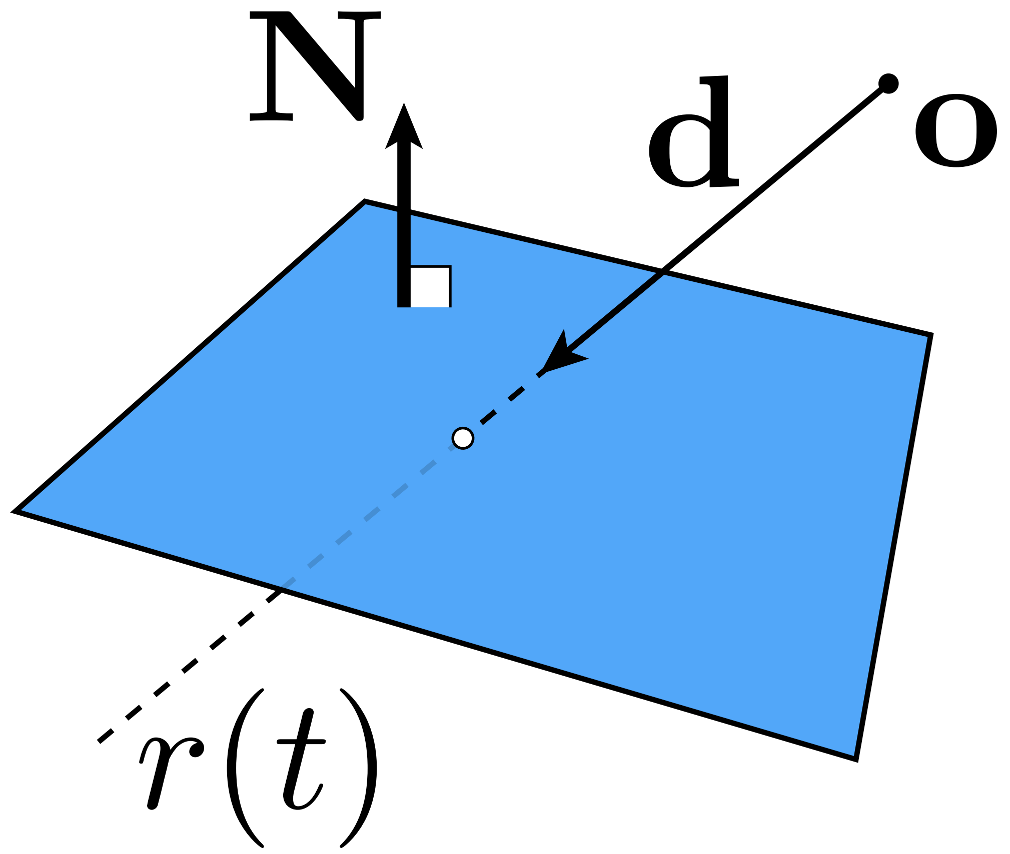
Accelerating Geometric Queries

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2024

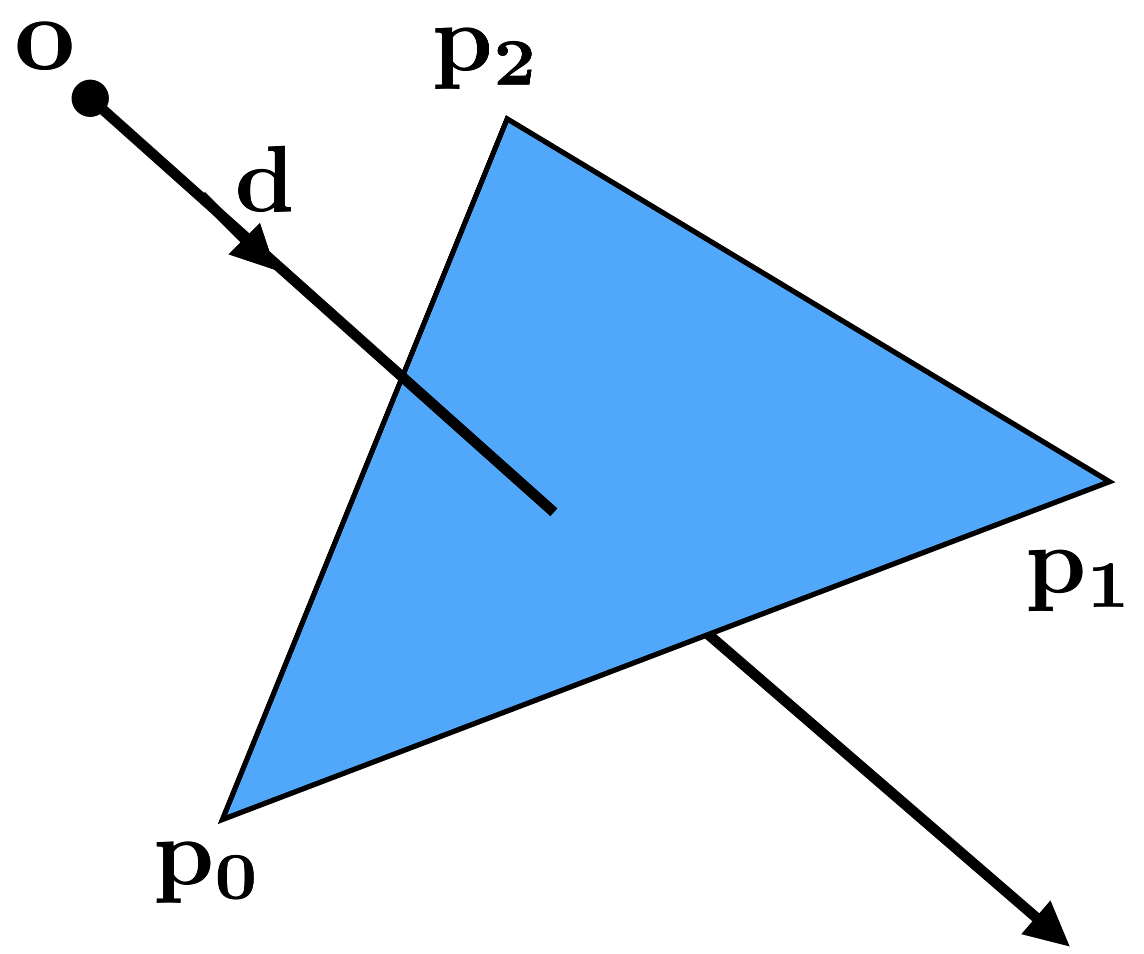
Last time: intersecting a ray with individual primitives



Ray-sphere



Ray-plane



Ray-triangle

Applying what you learned last time

Consider interesting a ray with a cylinder with radius R and length L! (centered at the origin)

I'll give you: the implicit form of a circle in 2D

$$x^2 + y^2 = R^2$$

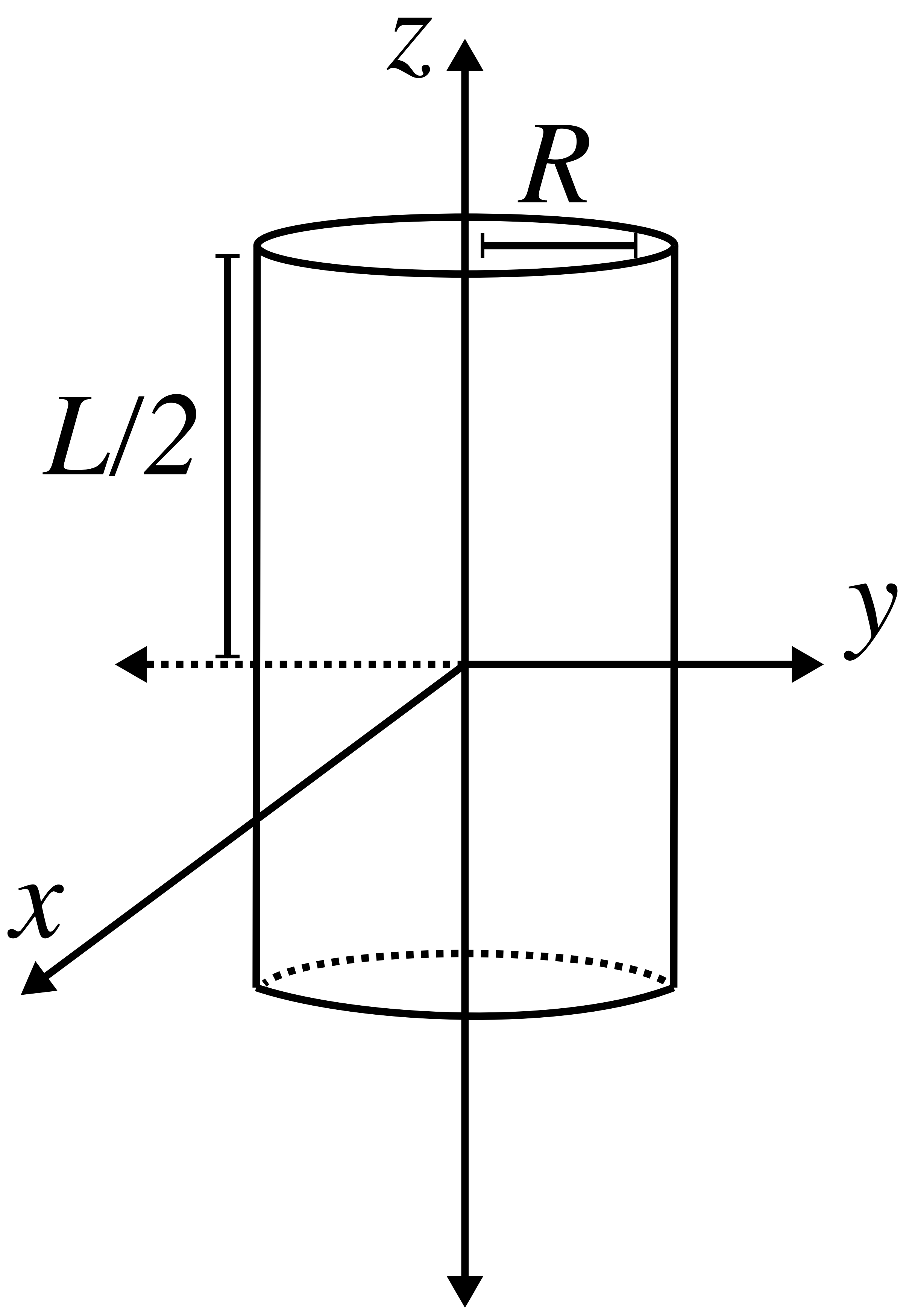
From last class you know:

Explicit form for a ray:

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

Implicit form for a plane:

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

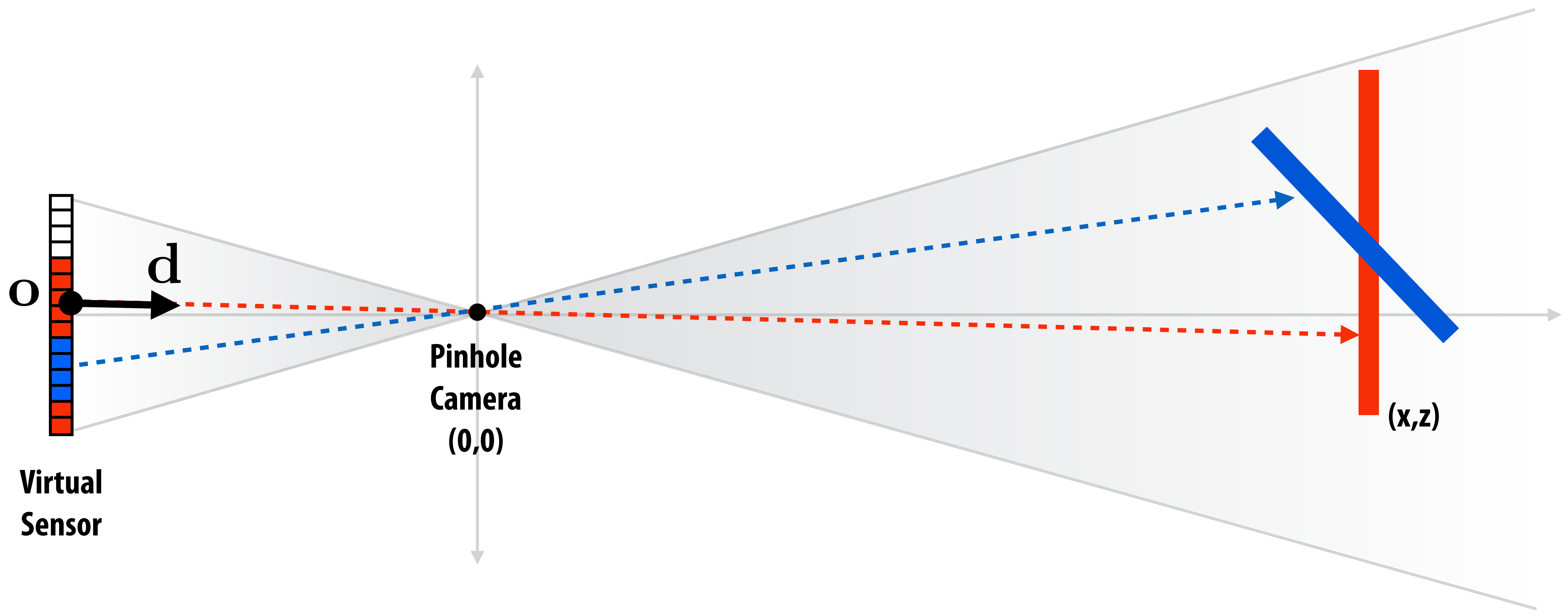


Q. What if the cylinder is centered at (x_0, y_0, z_0) instead of the origin?

Motivation (ray tracing)

The visibility problem: as ray casting

- 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)



Review: basic rasterization algorithm

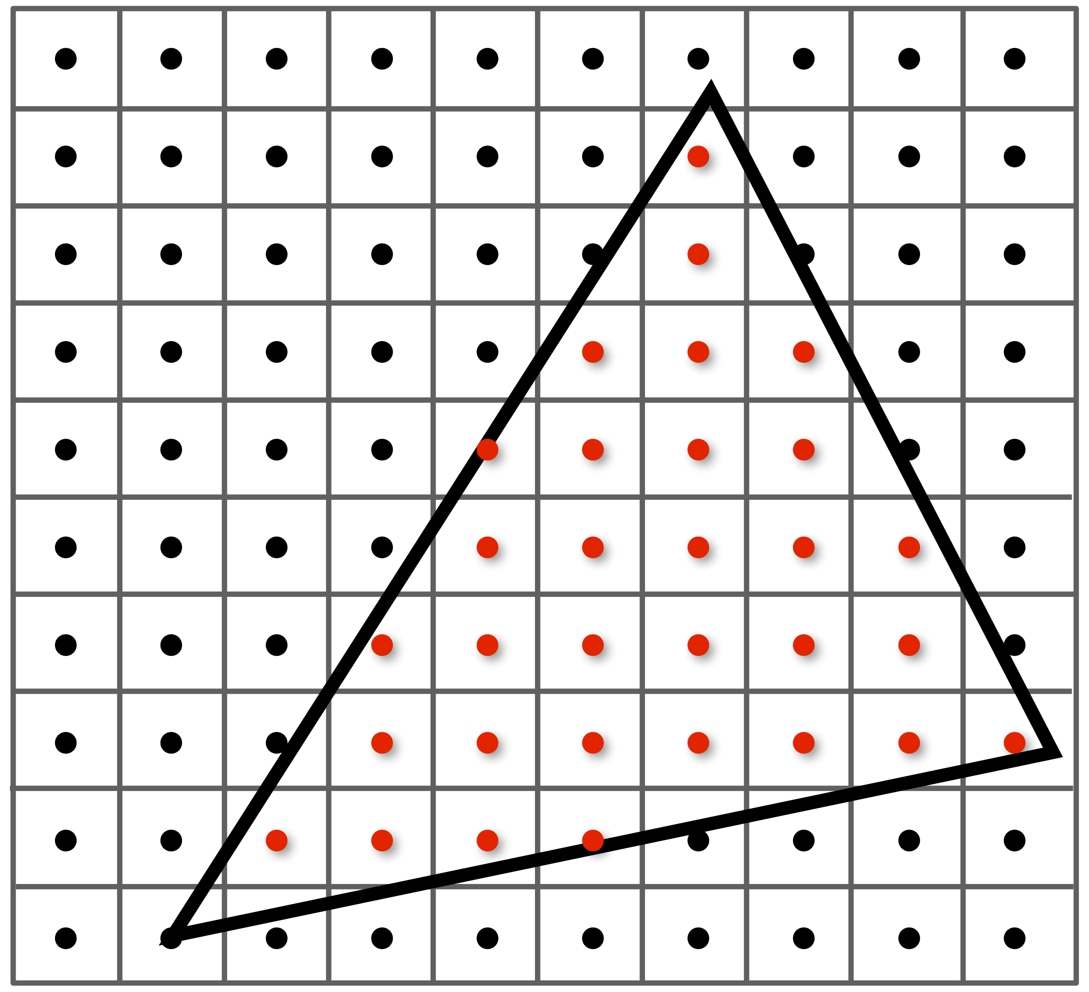
Sample = 2D point

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: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:       // loop 2: over 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"



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: over 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: over 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!

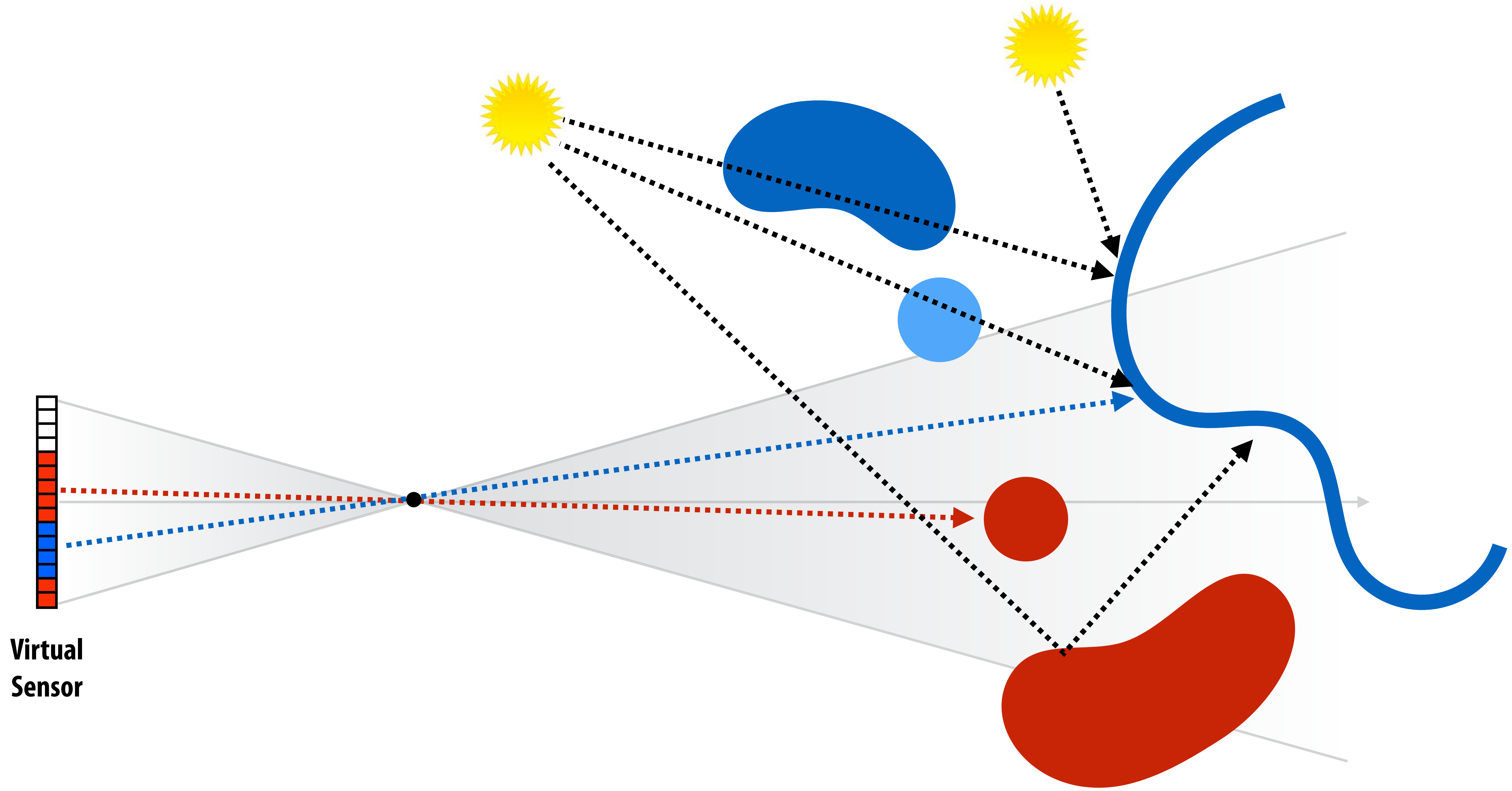
"Given a ray, find the closest triangle it hits."

Generality of ray-scene queries

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?



Accelerating ray-scene queries

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

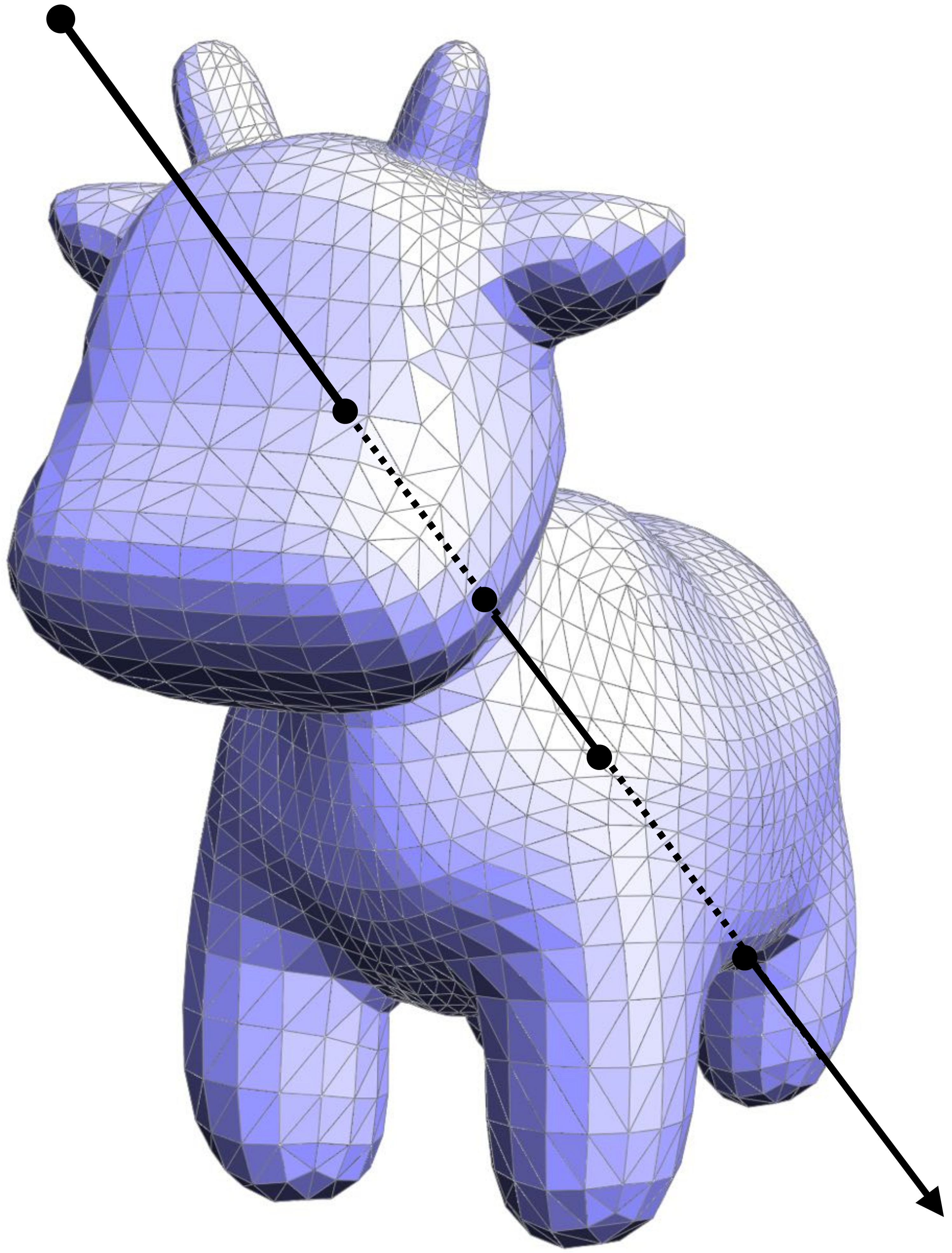
"Find the first primitive the ray hits"

```
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?

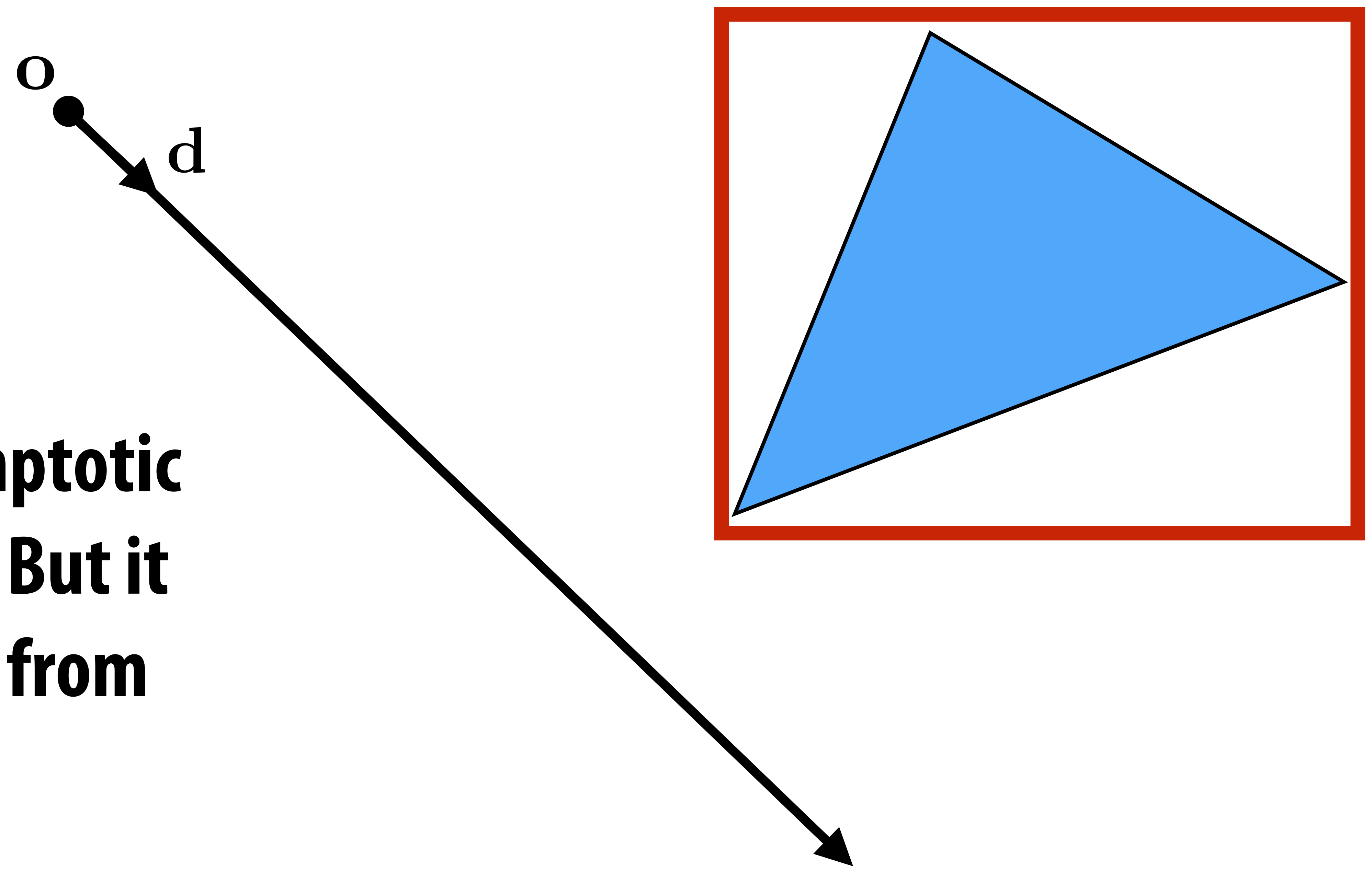
(Assume p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r*)



One simple idea

- "Early out" — Skip ray-primitive test if it's computationally easy to determine that ray does not intersect primitives

- E.g., A ray cannot intersect a primitive if it doesn't intersect the bbox containing it!



Note: early out does not change asymptotic complexity of ray-scene intersection. But it reduces cost by a constant if ray is far from most triangles.

Ray-axis-aligned-box intersection

What is ray's closest/farthest intersection with axis-aligned box?

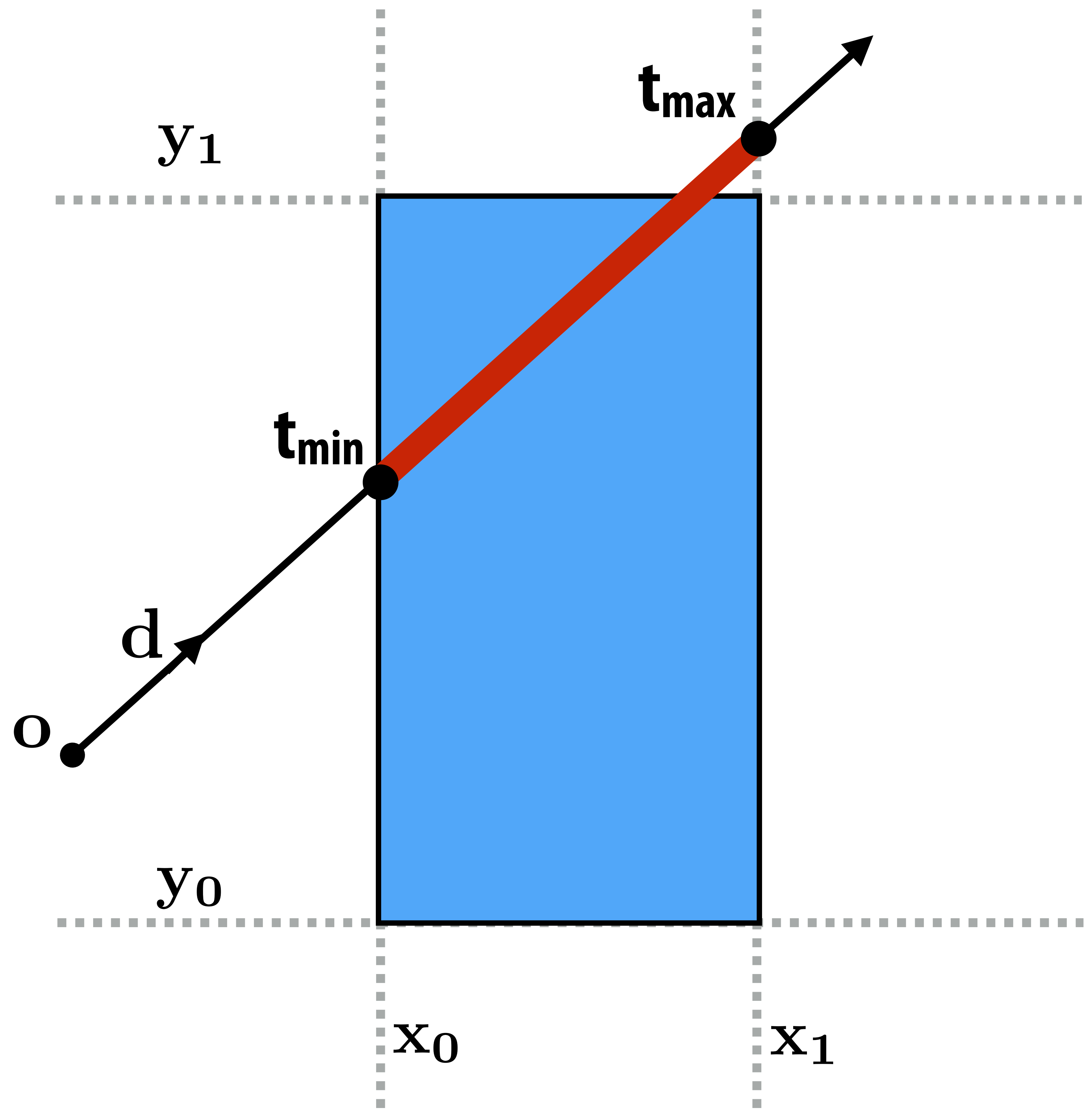


Figure shows intersections with $x=x_0$ and $x=x_1$ planes.

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):

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

$$c = x_0$$

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

Performance note: it is possible to precompute terms that only depend on the ray, so computing *t* is cheap

$$a = \frac{1}{\mathbf{d_x}} \qquad b = -\frac{\mathbf{o_x}}{\mathbf{d_x}}$$

So… $t = ax_0 + b$

So how do we find the closest hit for a 3D box?

1. How do you know there is a hit at all?
2. What is the t value for that hit?

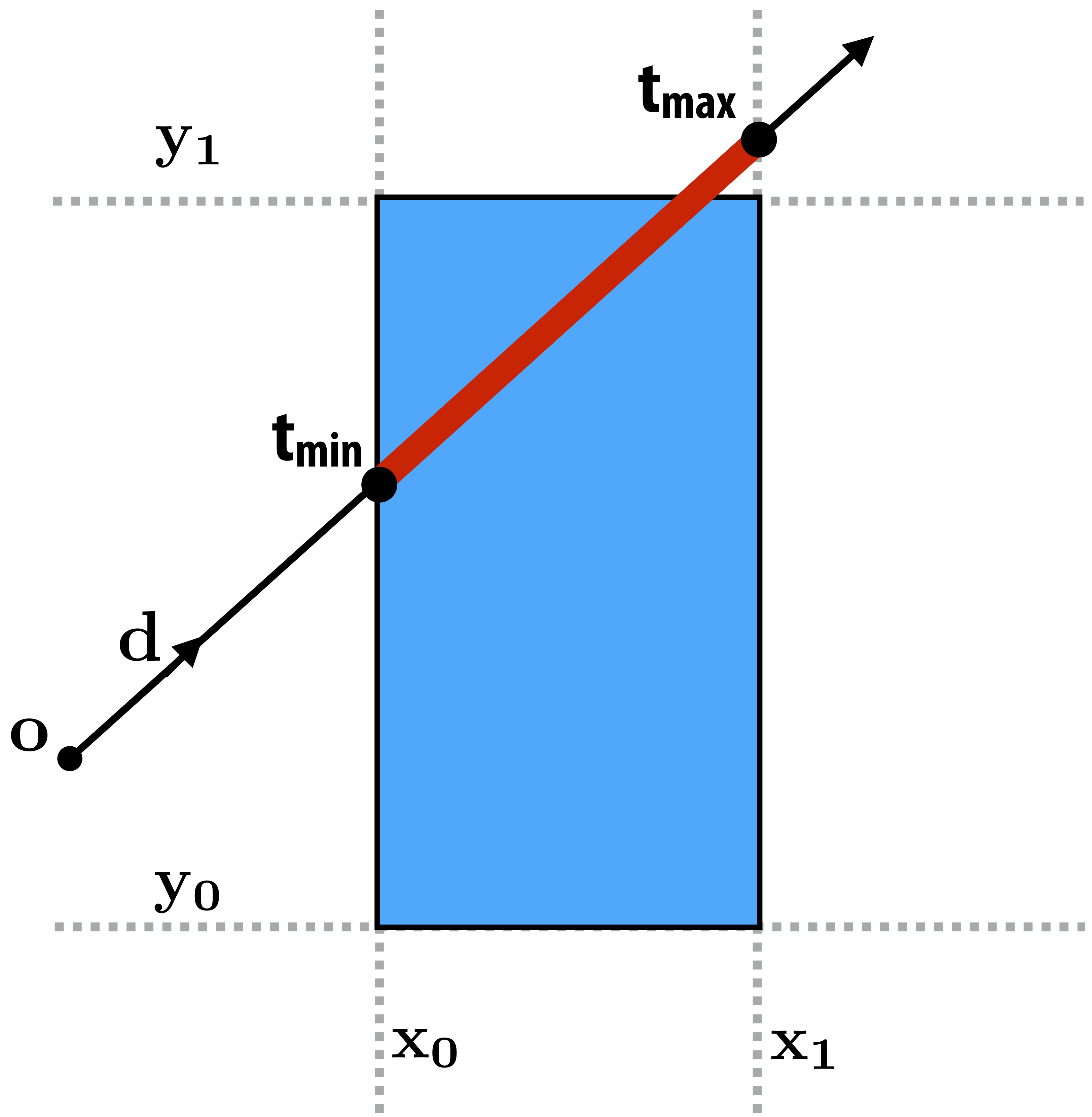
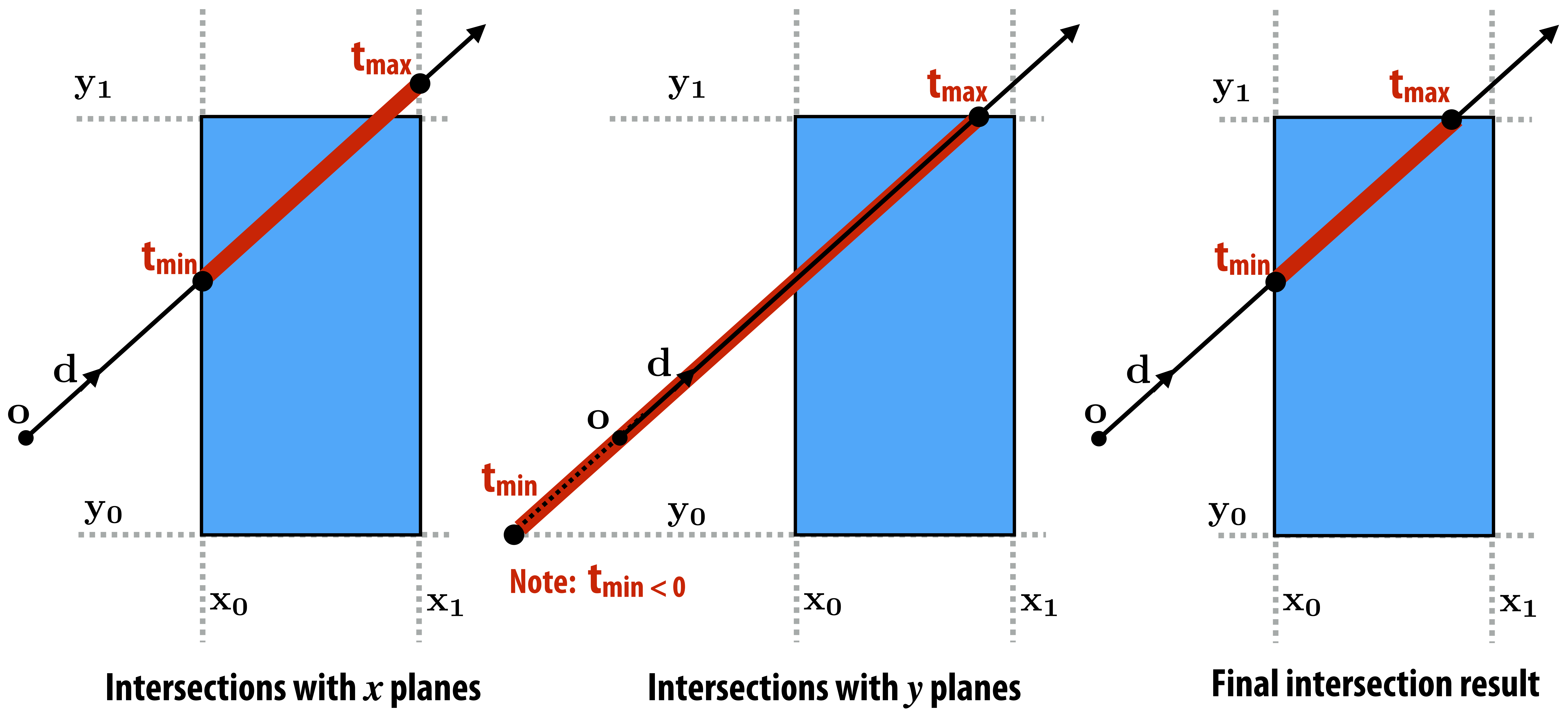


Figure shows intersections with $x=x_0$ and $x=x_1$ planes.

Ray-axis-aligned-box intersection

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



How do we know if the ray hits the box?

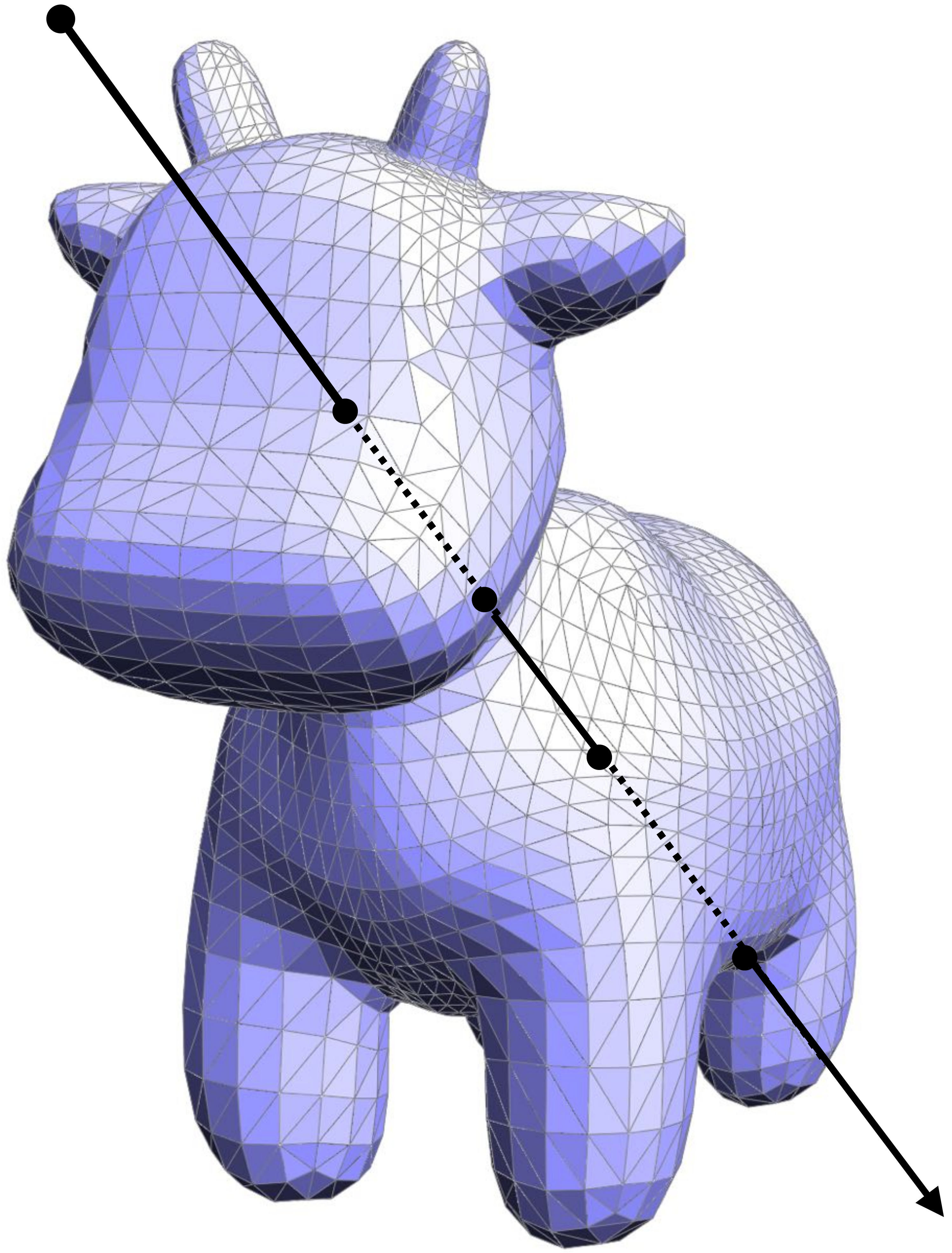
If there's a t-range where the ray is within the X planes, Y planes, AND Z planes, then we are in the box (ray hits it)

Ray-scene intersection with early out

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

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

Still $O(N)$ **complexity.**



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

Recall this optimization in a simple rasterizer

Your assignment 1 rasterizer skipped sample-in-triangle tests for samples not contained in the bounding box of the triangle.
(It's the 2D equivalent of skipping ray-triangle test in 3D if the ray does not hit 3D bbox of a triangle!)

```
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: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:          // loop 2: over 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]
```

```
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: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in 2D BOUNDING BOX OF TRIANGLE:  // loop 2: over 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]
```

Cull samples not within bbox
(if sample not in bbox don't attempt
more expensive point in triangle test)

Disney Moana scene

Released for rendering research purposes in 2018.
15 billion primitives in scene (more than 90M unique geometric primitives)

Disney Moana scene

Released for rendering research purposes in 2018.

15 billion primitives in scene (more than 90M unique geometric primitives)

Disney Moana scene

Released for rendering research purposes in 2018.
15 billion primitives in scene (more than 90M unique geometric primitives)

Disney Moana scene

Released for rendering research purposes in 2018.
15 billion primitives in scene (more than 90M unique geometric primitives)

Data structures for reducing O(N) complexity of ray-scene intersection

*Given ray, find closest intersection with set of scene triangles.**

** We are also interested in: Given ray, find if there is <u>any</u> intersection with scene triangles*

A simpler problem

- Imagine I have a set of integers
- Given an integer, say *k*=18, find the element in the set that is 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:

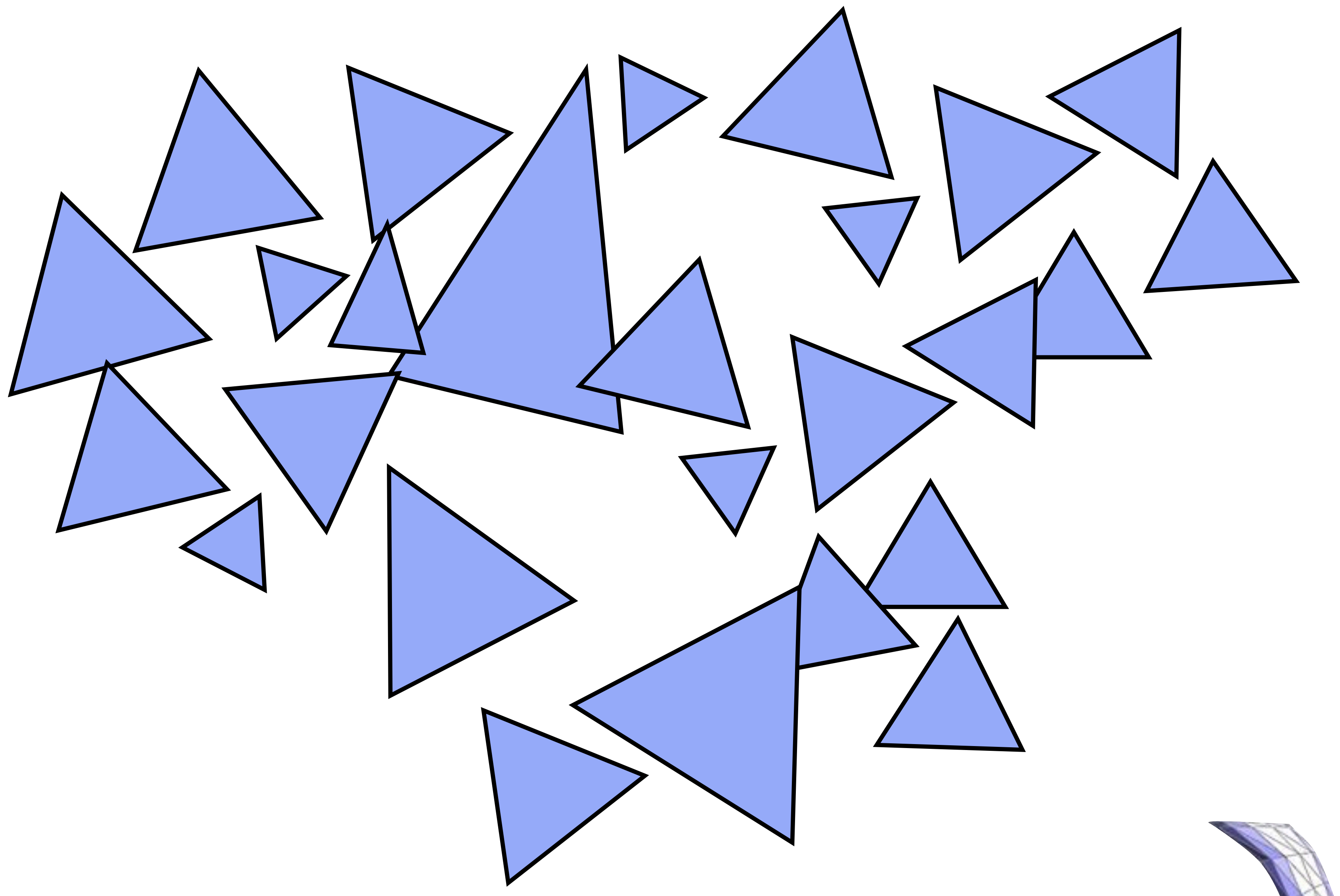
1 2 6 8 10 11 $\boxed{20}$ 25 30 64 80 100 111 123 200 950

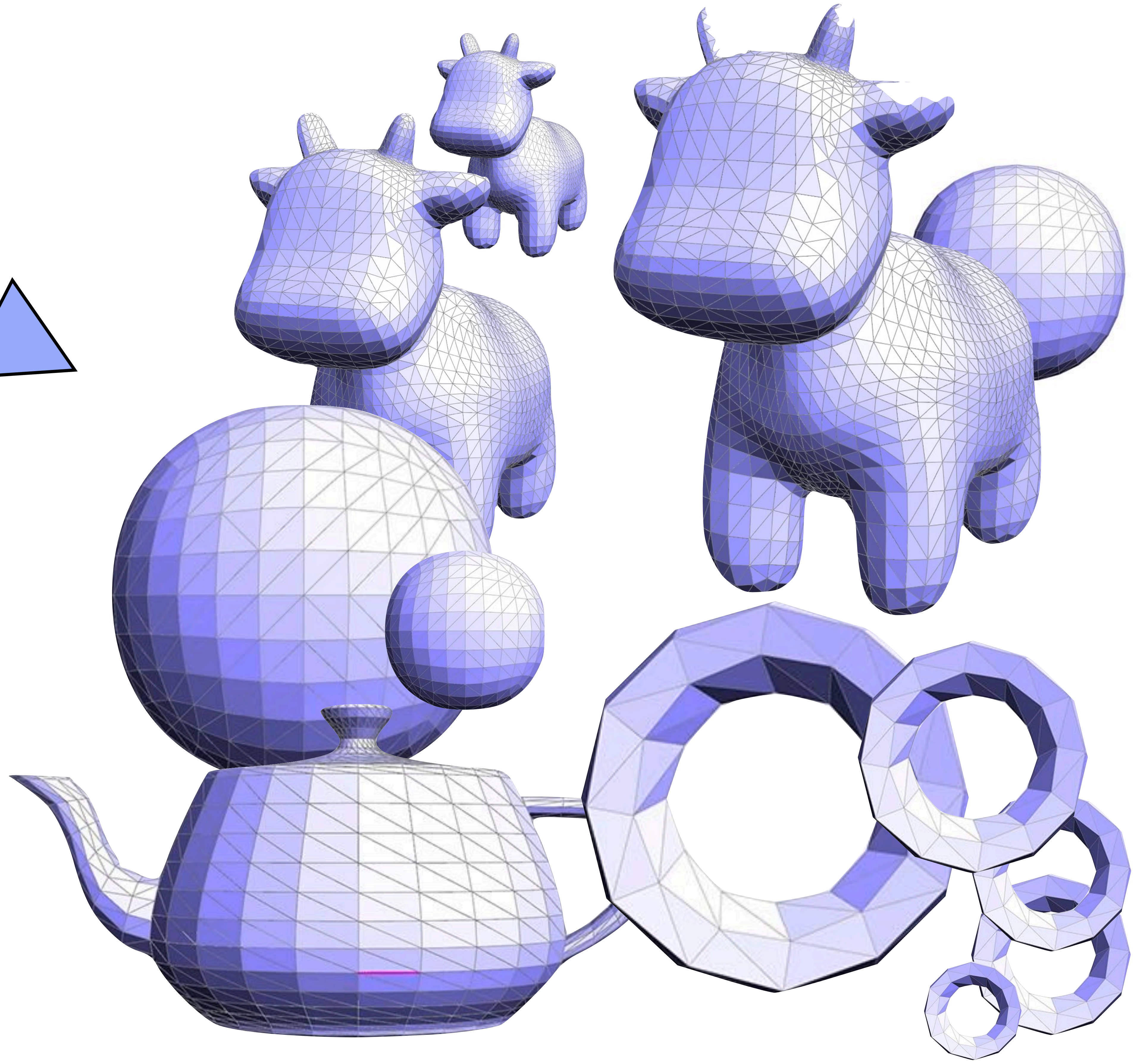
How much does it now cost to find k (*including sorting*)?

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

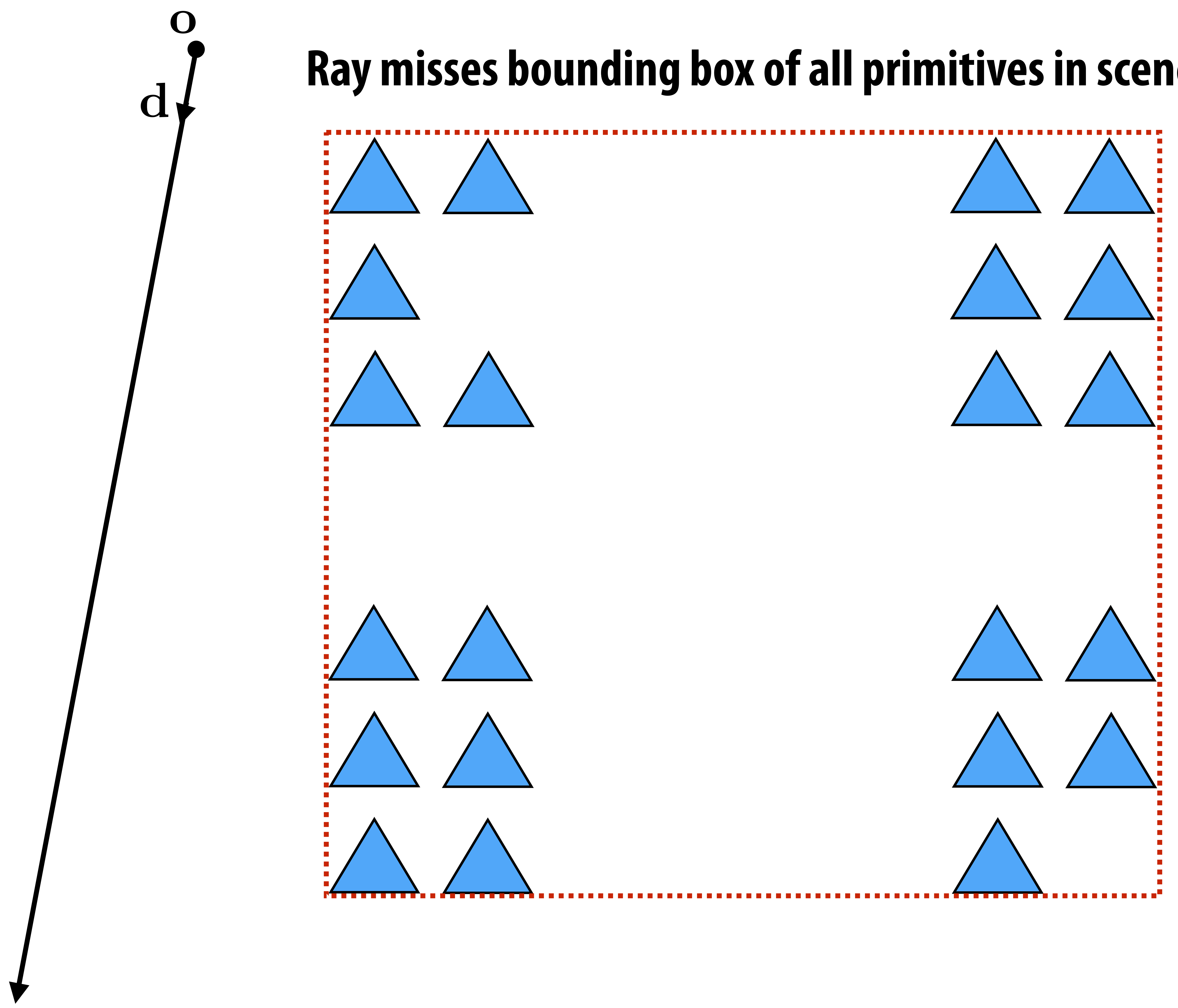
Amortized cost over many queries: O(log n) *much* better!

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





Simple case (rays miss bounding box of scene)



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

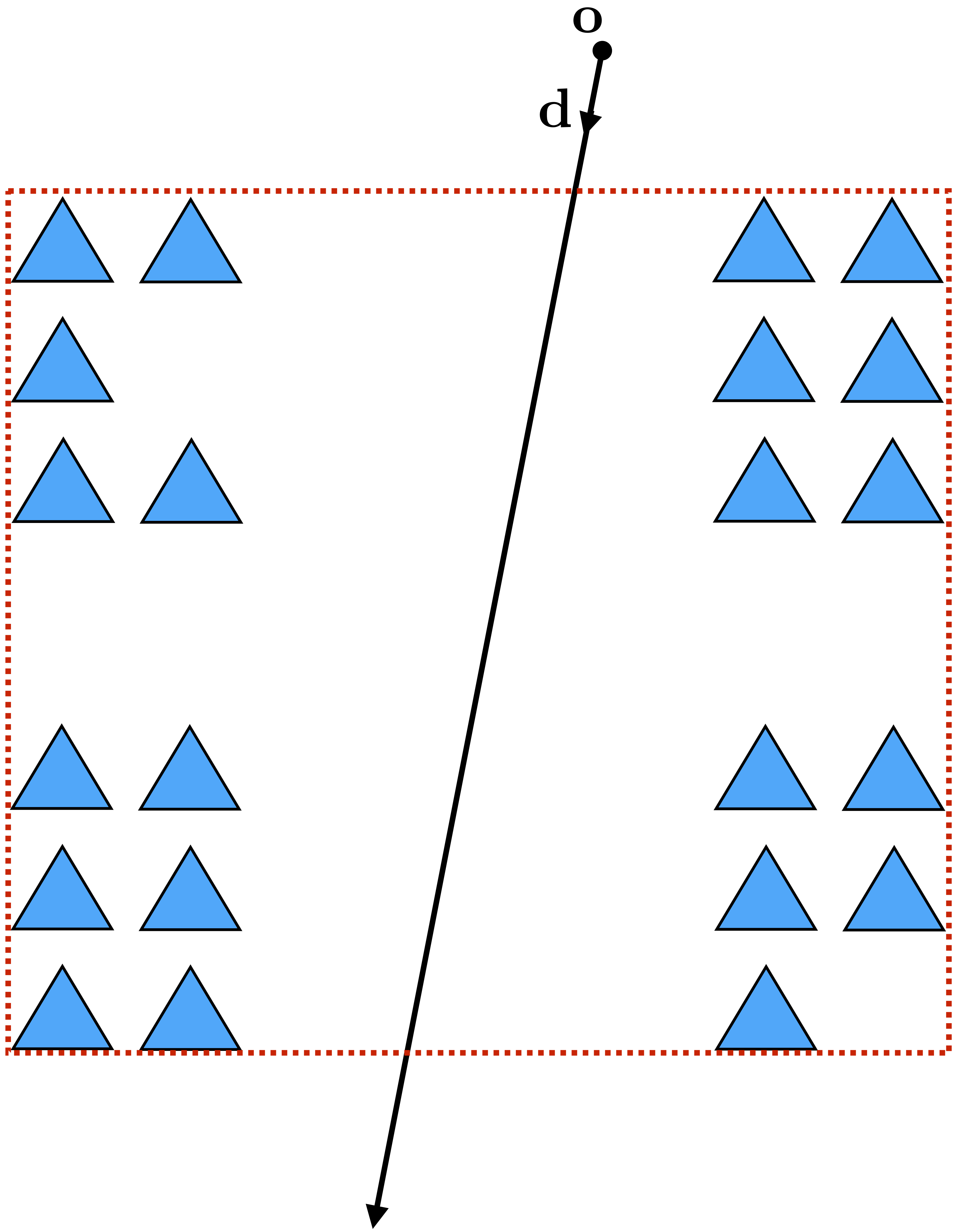
preprocessing: O(n)

ray-box test: O(1)

amortized cost*: O(1)

*amortized 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)

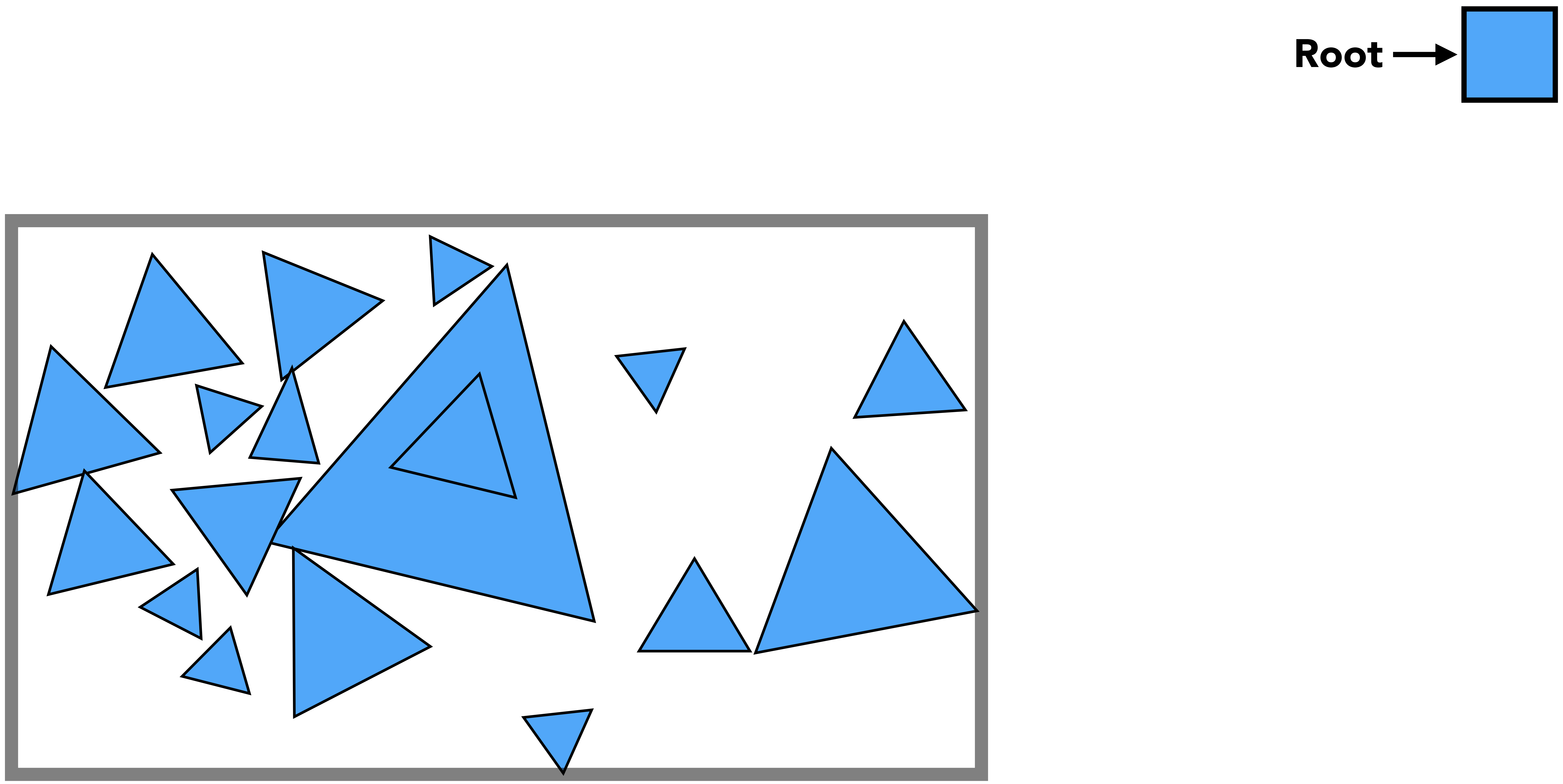
Still no better than naïve algorithm (must test all triangles)!

*amortized over *many* ray-scene intersection tests

Q: How can we do better?

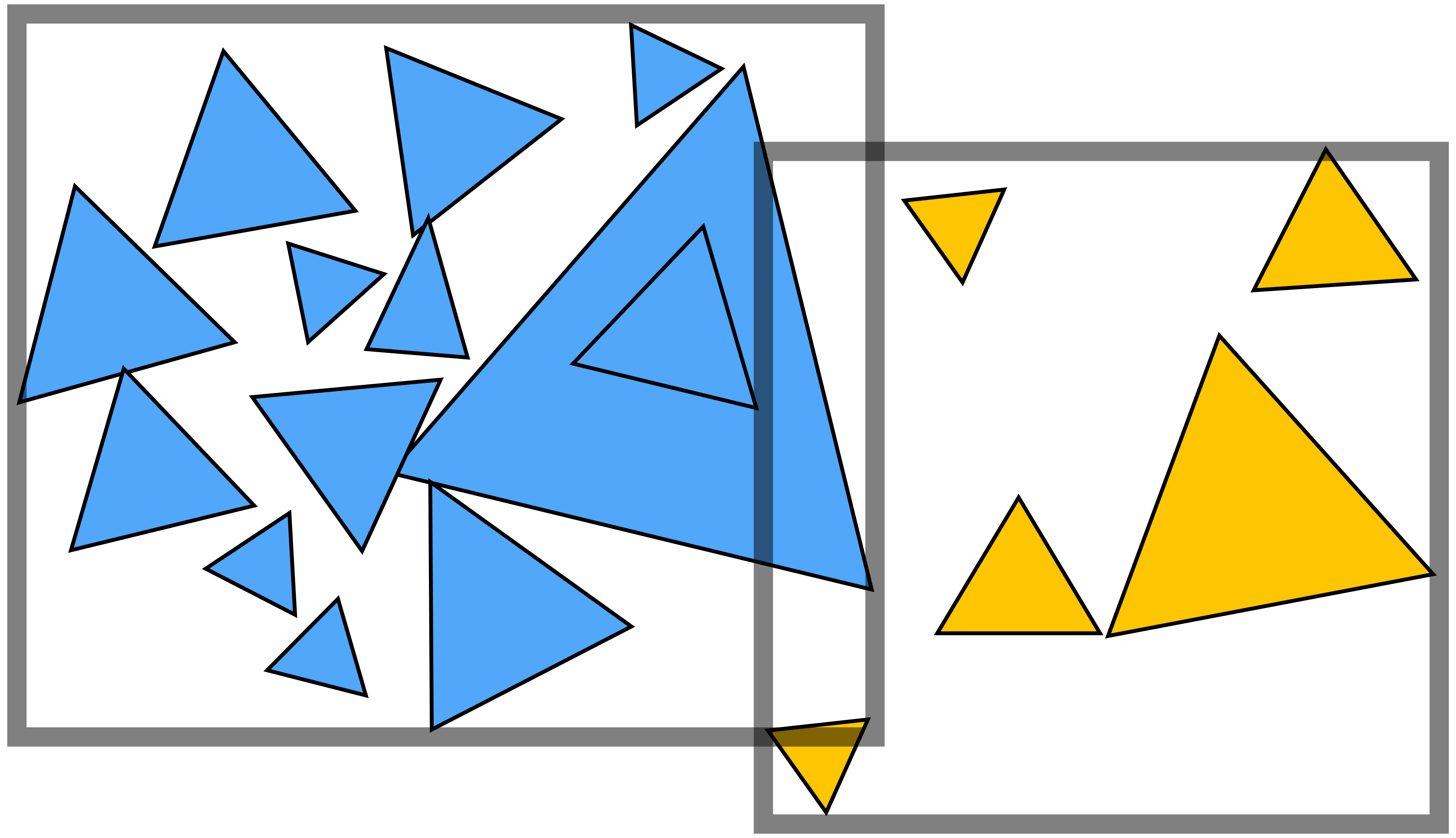
A: Apply this strategy hierarchically

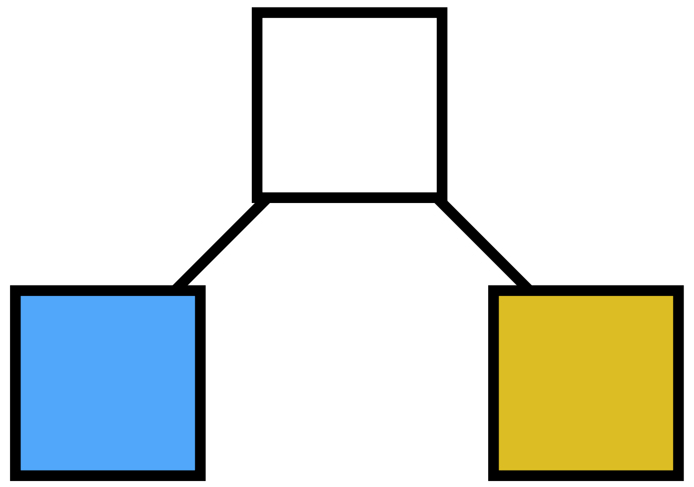
Bounding volume hierarchy (BVH)



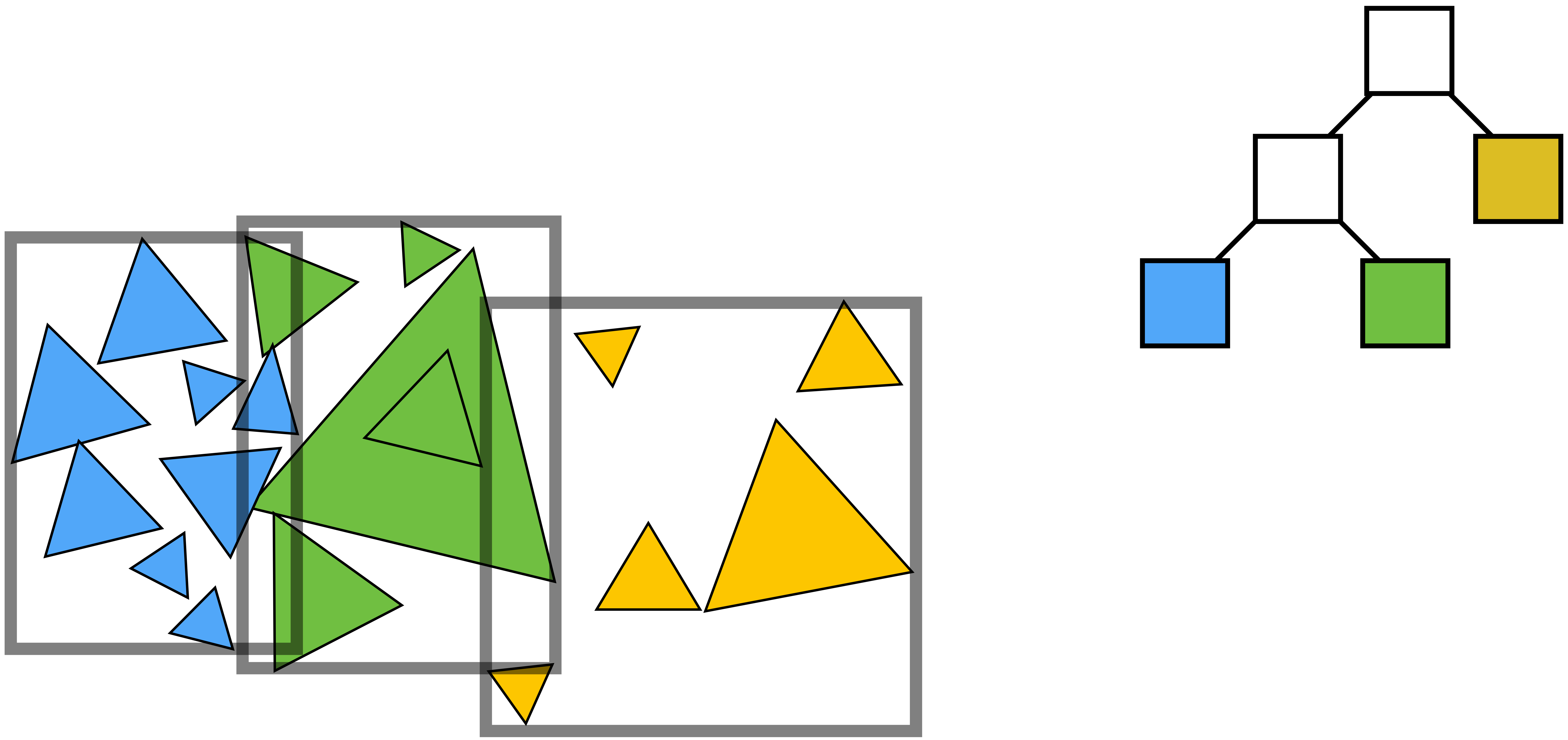
Bounding volume hierarchy (BVH)

- BVH partitions each node's primitives into disjoints sets
  - Note: the sets can overlap in space (see example below)



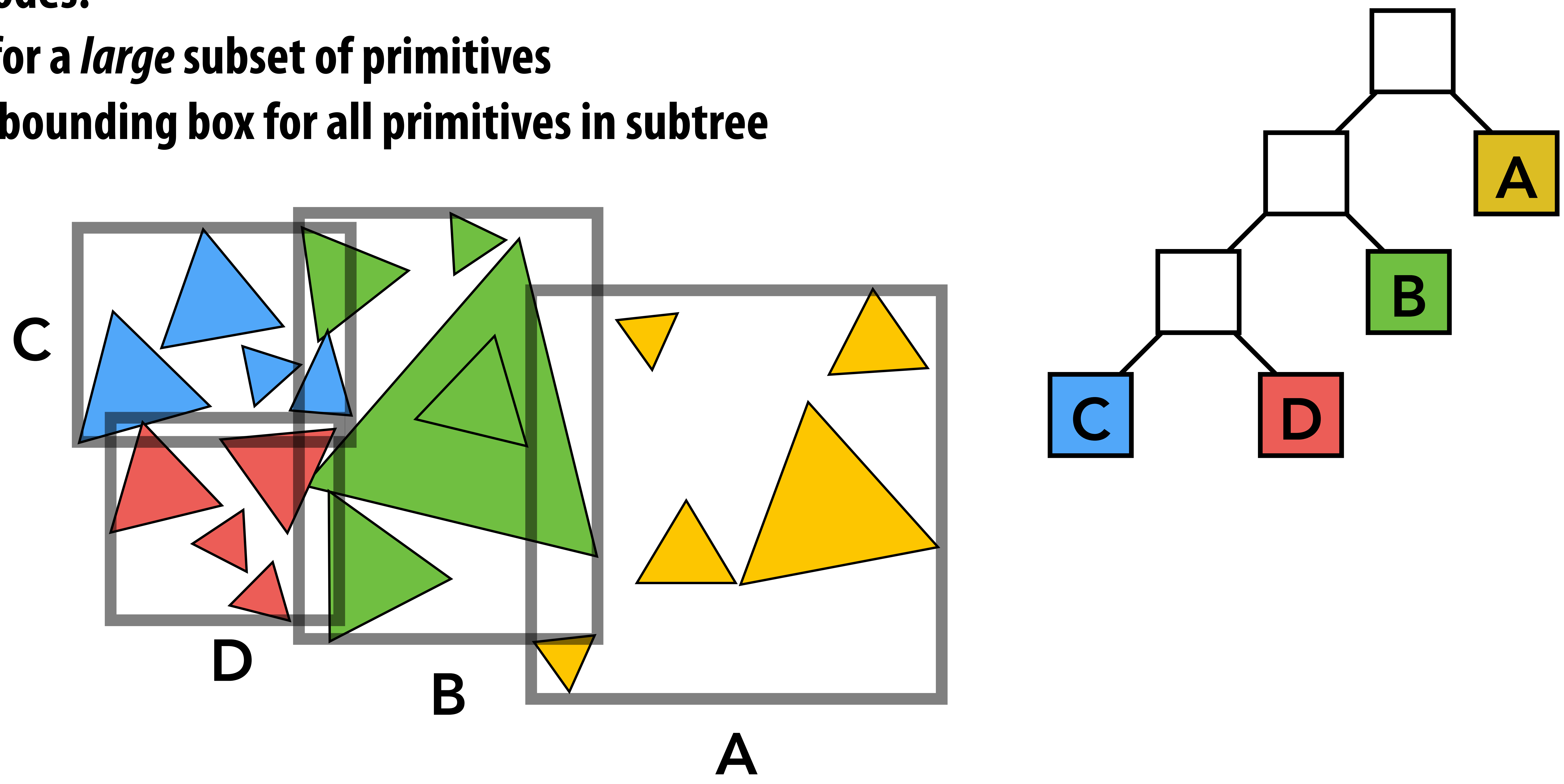


Bounding volume hierarchy (BVH)

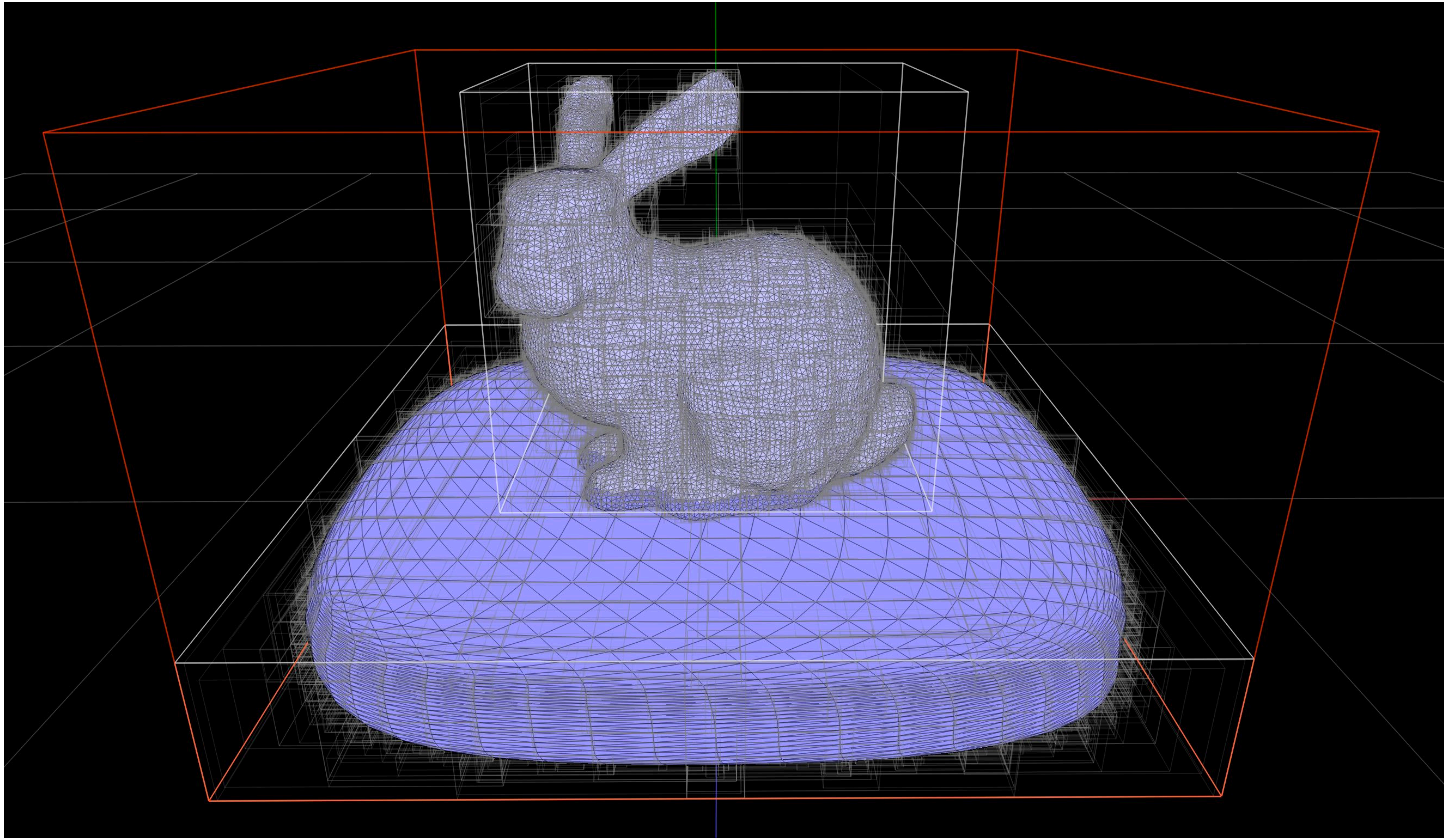


Bounding volume hierarchy (BVH)

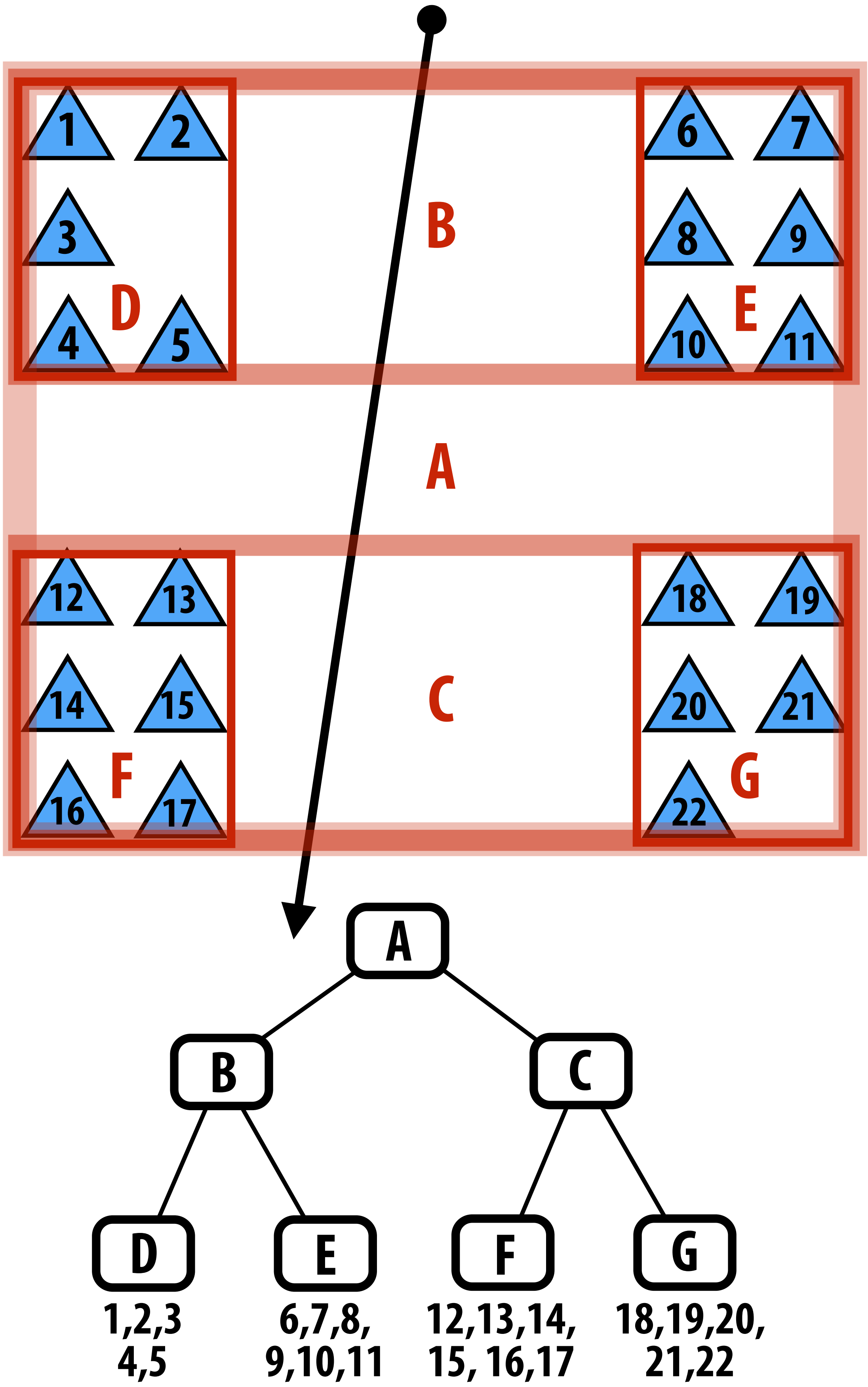
- **Leaf nodes:**
  - Contain *small* list of primitives
- **Interior nodes:**
  - Proxy for a *large* subset of primitives
  - Stores bounding box for all primitives in subtree

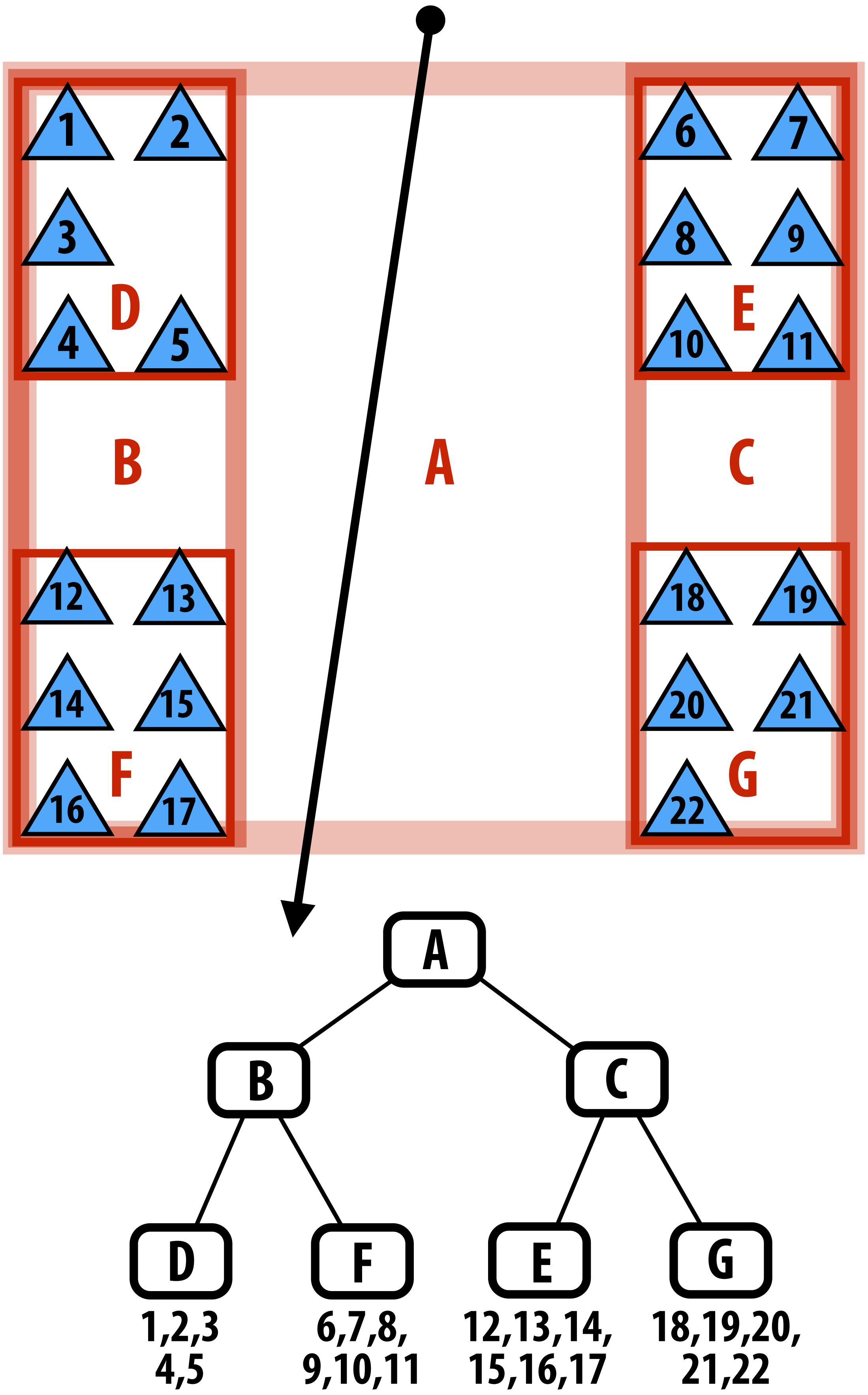


Demo!



Bounding volume hierarchy (BVH)



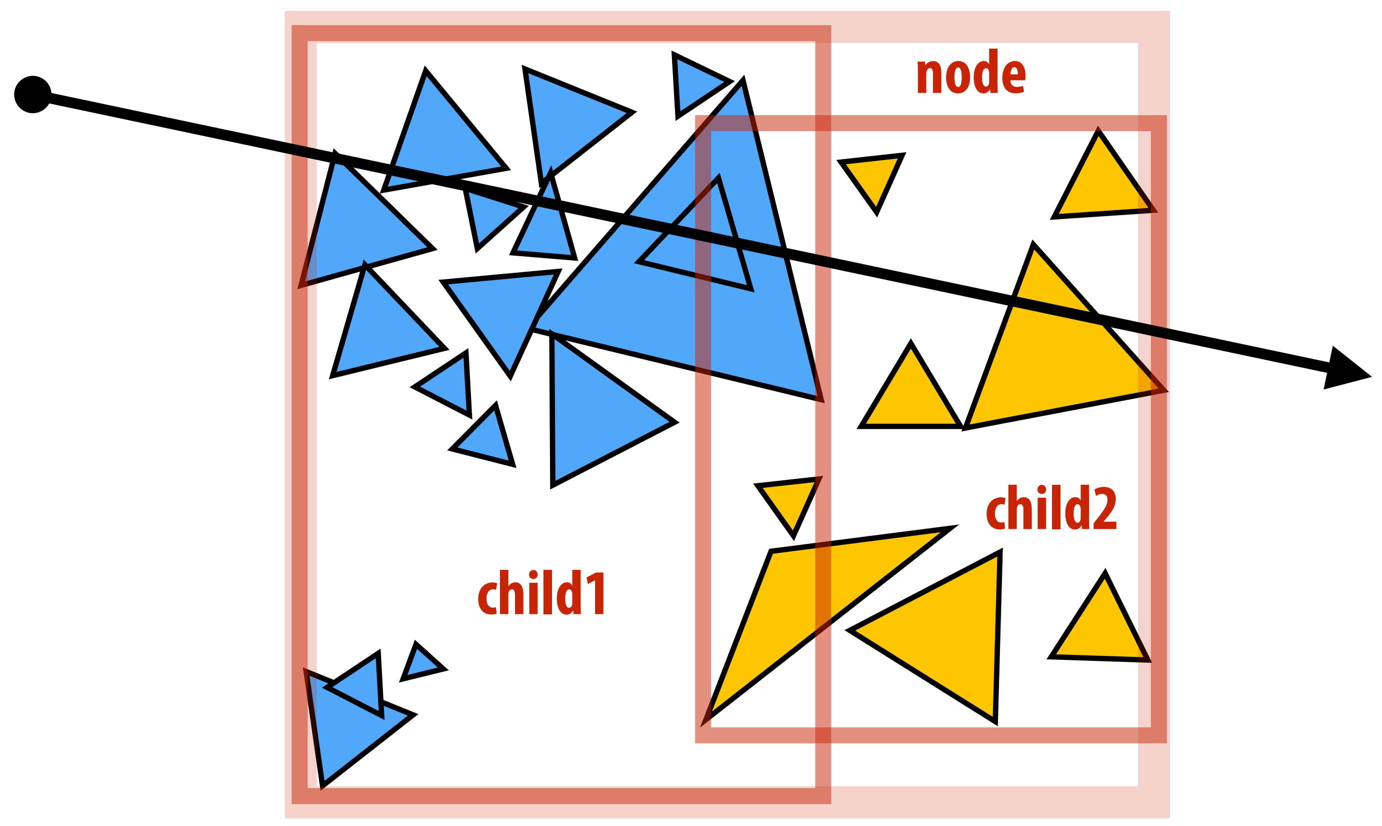


Two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Ray-scene intersection using a BVH

```
struct BVHNode {
   bool leaf;  // true if node is a leaf
   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 along ray?
};

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.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);
}}
```

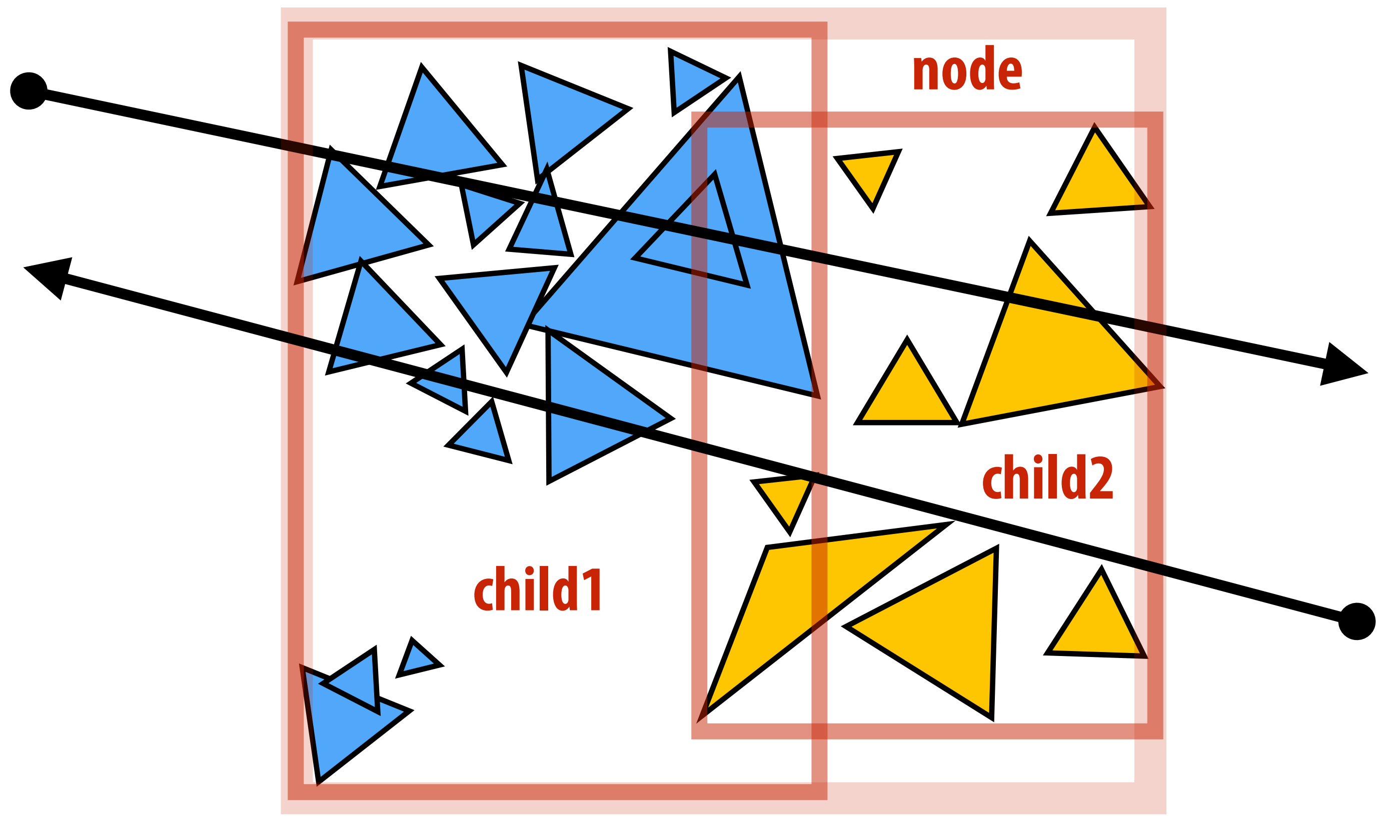


Can this occur if ray hits the box?

(assume hit.t is INF if ray misses box)

Improvement: “front-to-back” traversal

New invariant compared to last slide:
assume find_closest_hit() is only called for nodes where ray intersects bbox.




```
void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && 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 = (hit1.t <= hit2.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);
    }
}
```

“Front to back” traversal.

Traverse to closest child node first.

Why?

Why might we still need to traverse to second child if there was a hit with geometry in the first child?

Aside: another type of query: any hit

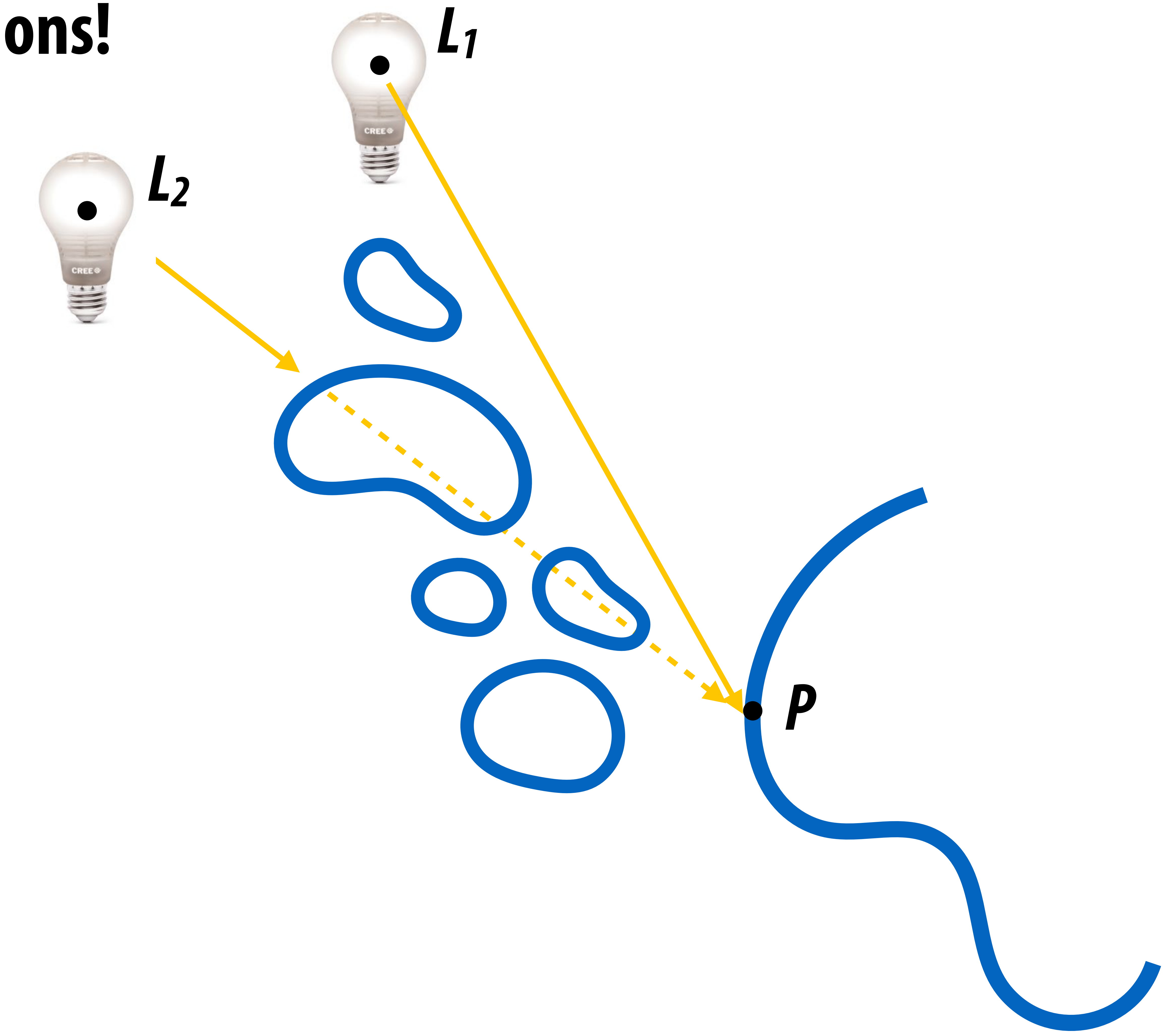
Sometimes it is useful to know if the ray hits ANY primitive in the scene at all (don't care about distance to first hit)

```
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 = intersect(ray, p);
         if (hit.prim)
            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?

Why "any hit" queries?

Shadow computations!

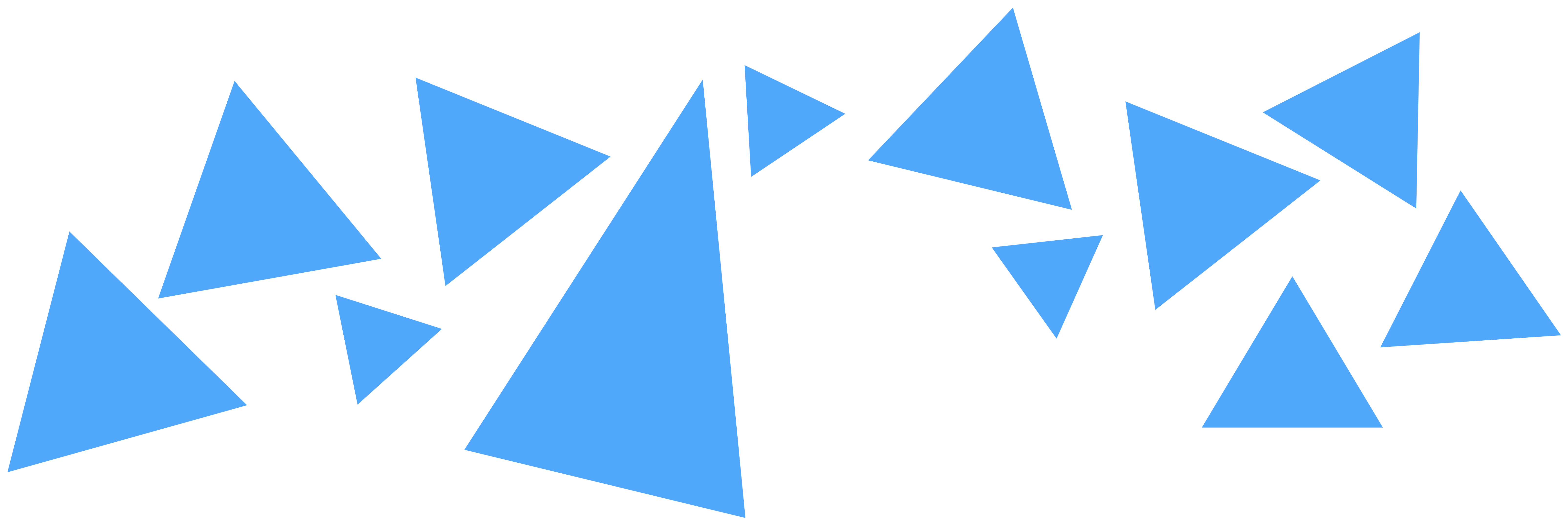


For a given set of primitives,
there are many possible BVHs

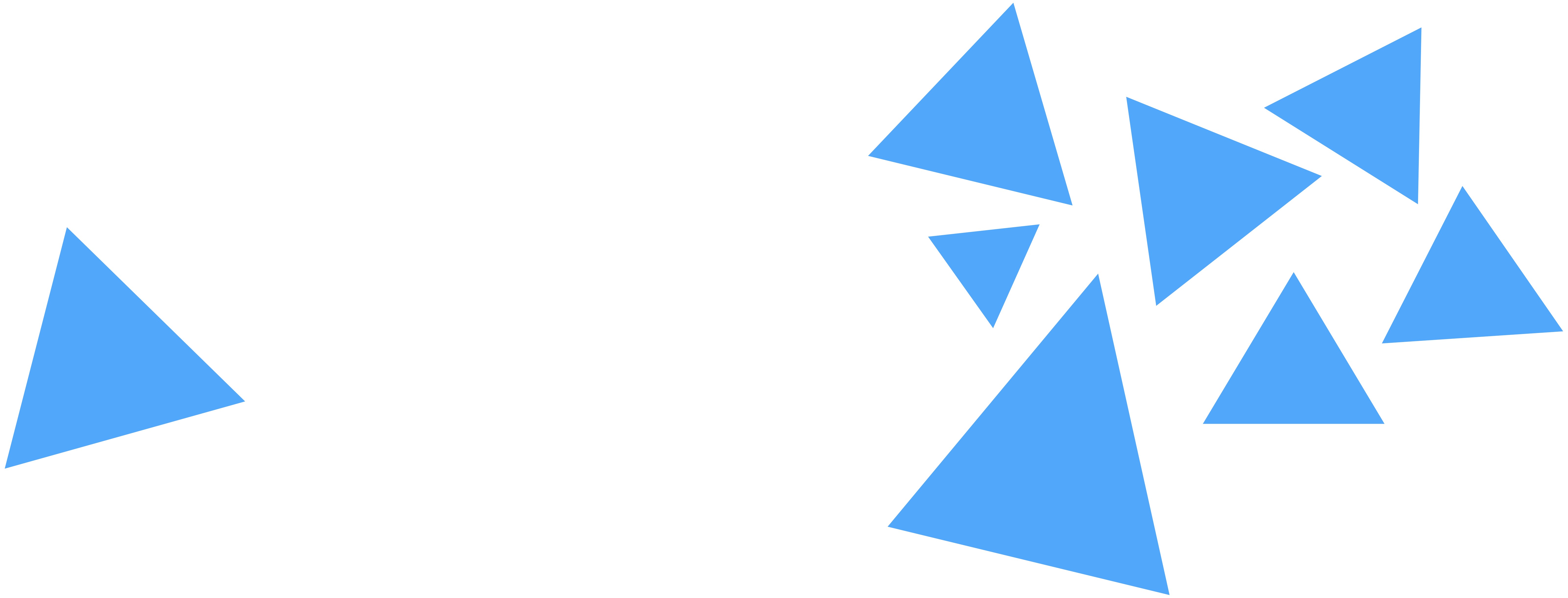
($\sim 2^N$ ways to partition N primitives into two groups)

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

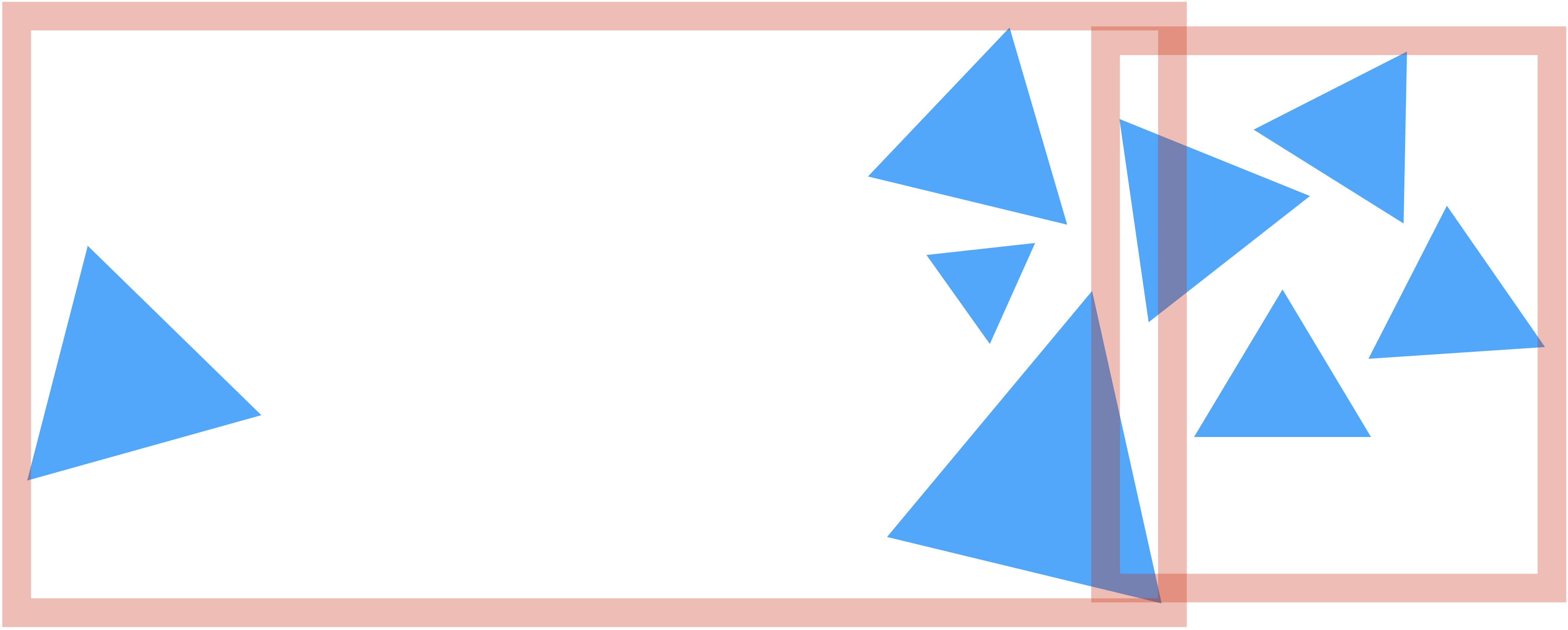
How would you partition these triangles into two groups?



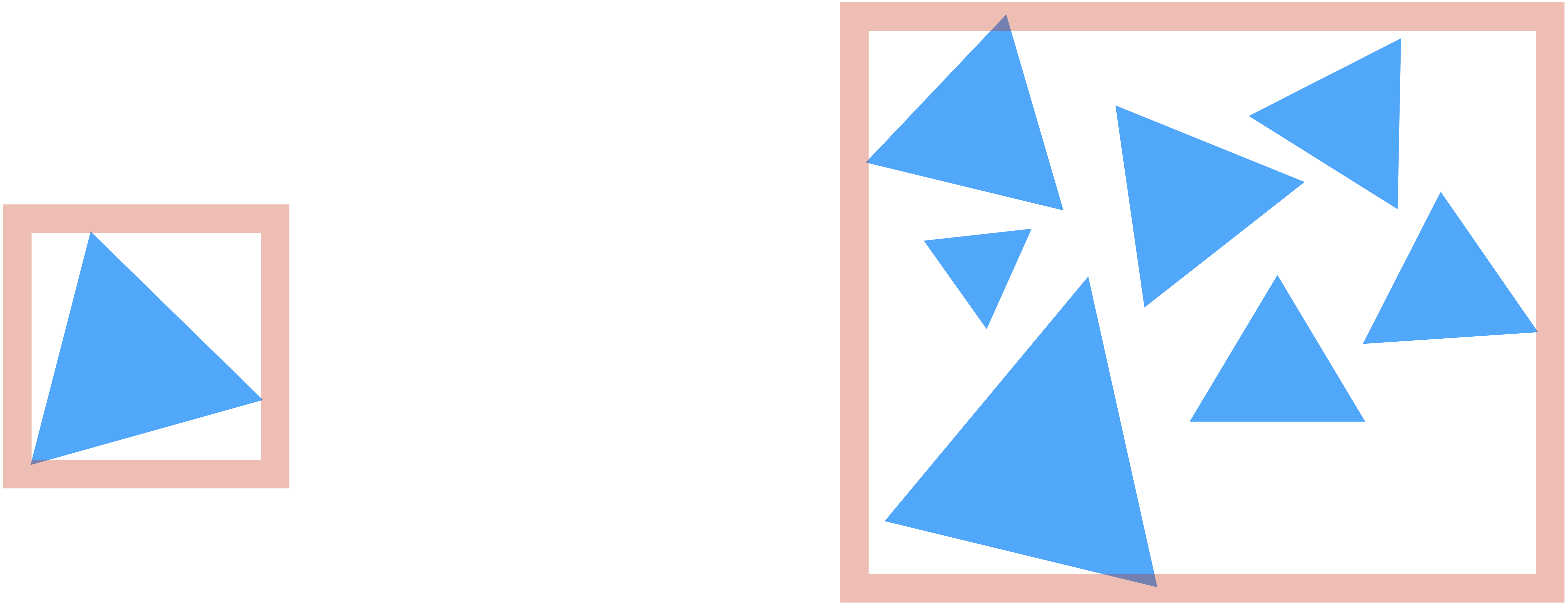
What about these?



Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: want small bounding boxes that minimize overlap between children, avoid bboxes with significant empty space

What are we really trying to do?

A good partitioning minimizes the expected cost of finding the closest intersection of a ray with the scene primitives in the node.

If a node is a leaf node (no partitioning):

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

$$= N C_{\text{isect}}$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive *i* in the node.

(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 intersection 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:

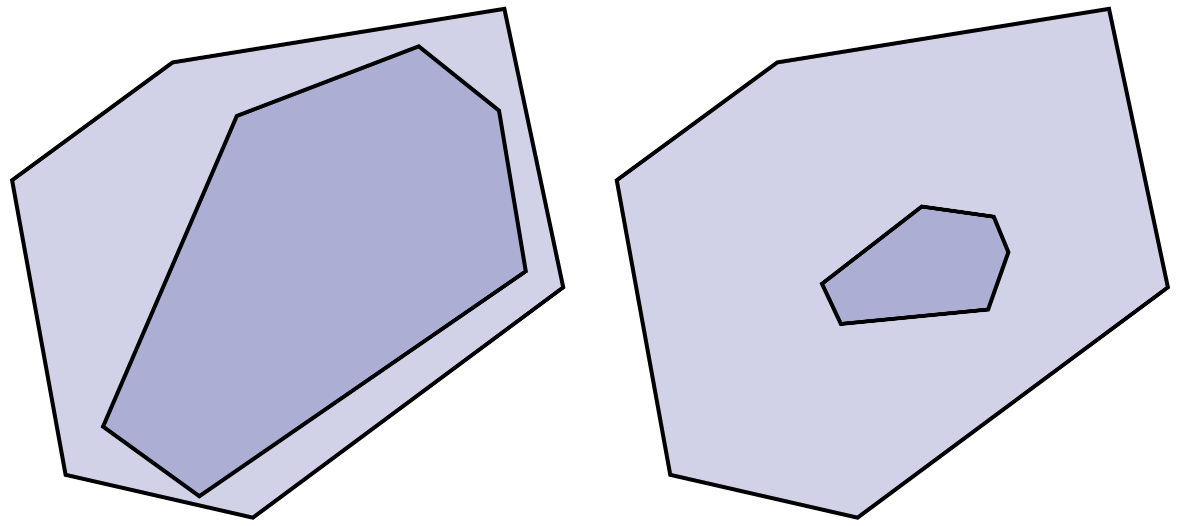
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

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):

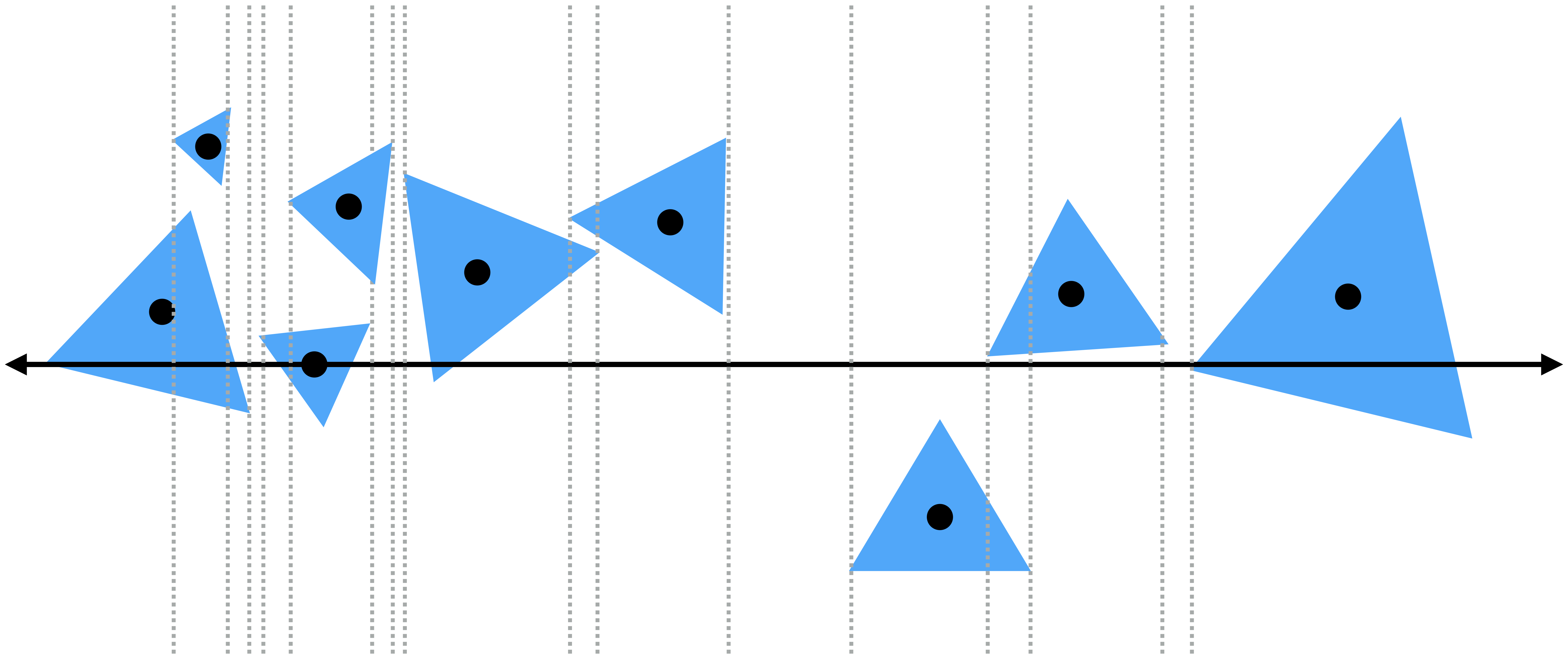
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

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

- Rays are randomly distributed
- Rays are not occluded

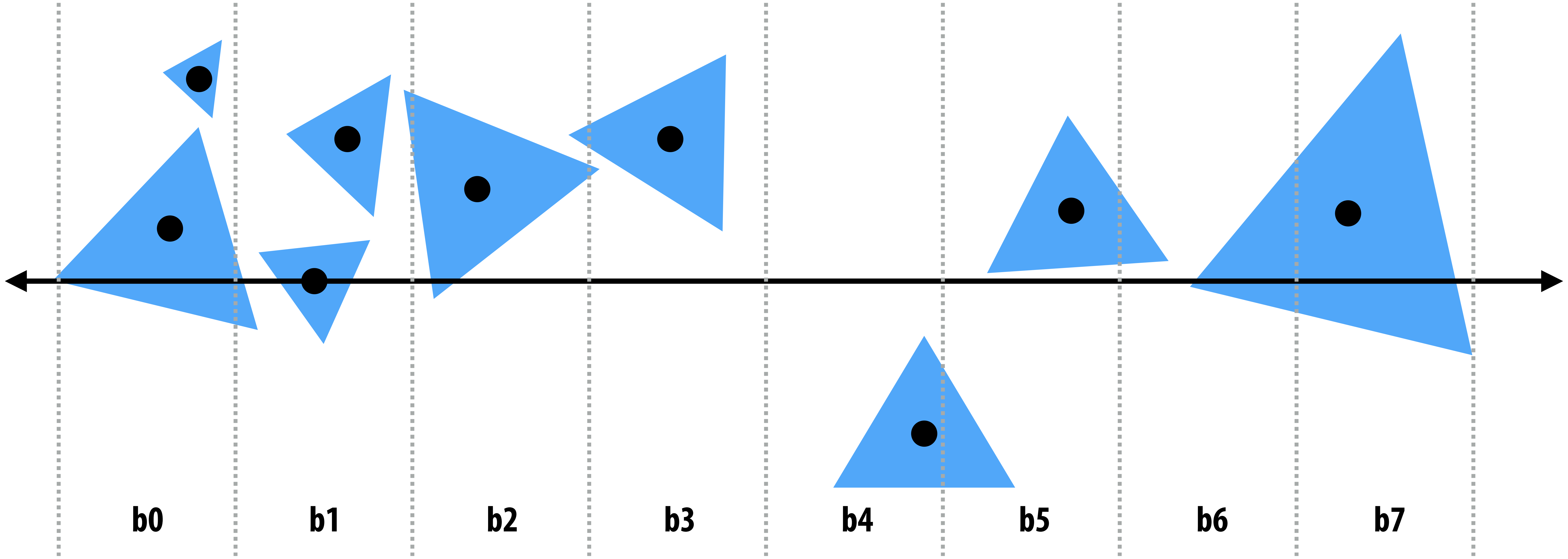
Implementing partitions

- 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 large number of possible split planes… O(# objects)



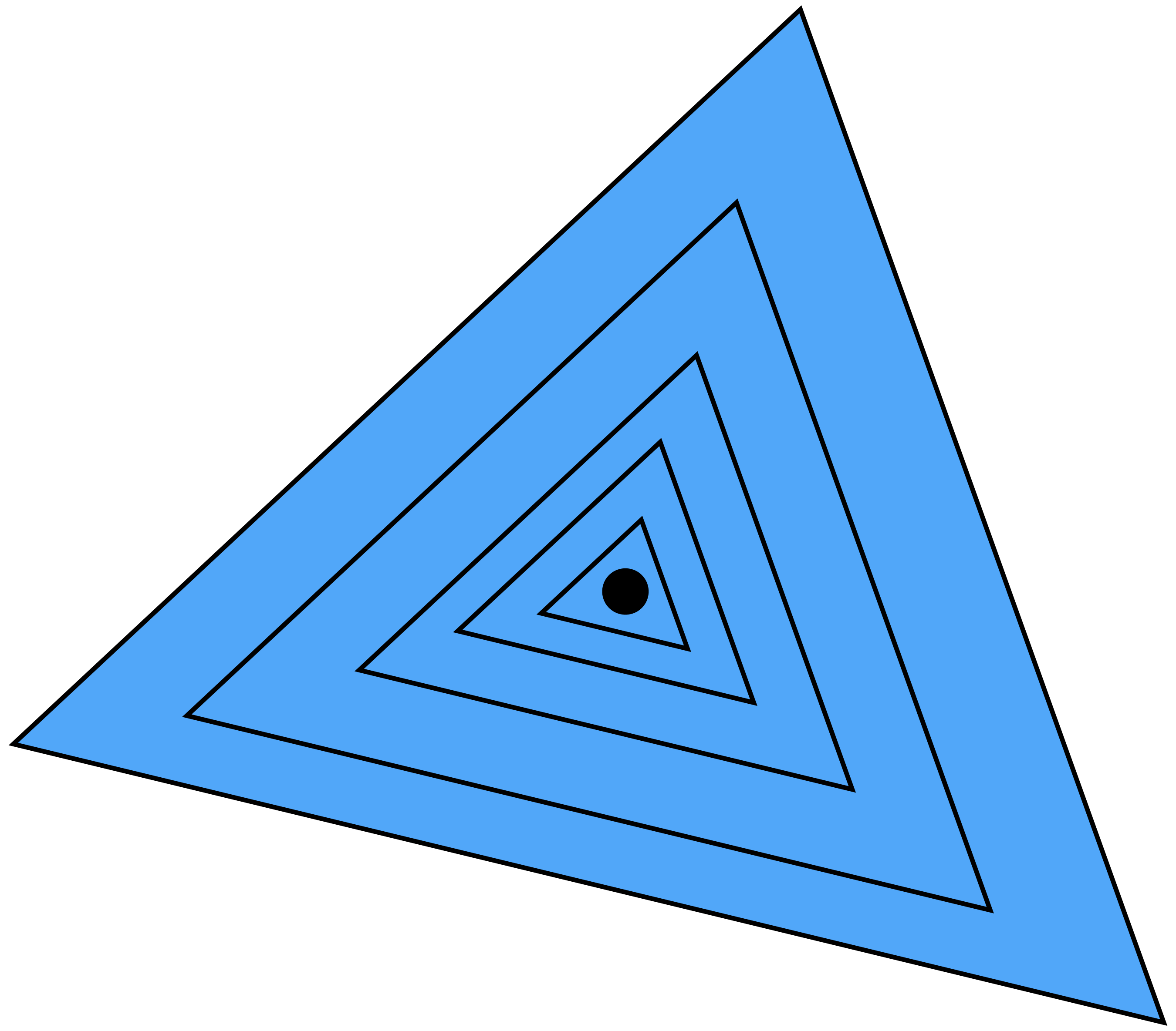
Efficiently implementing partitioning

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

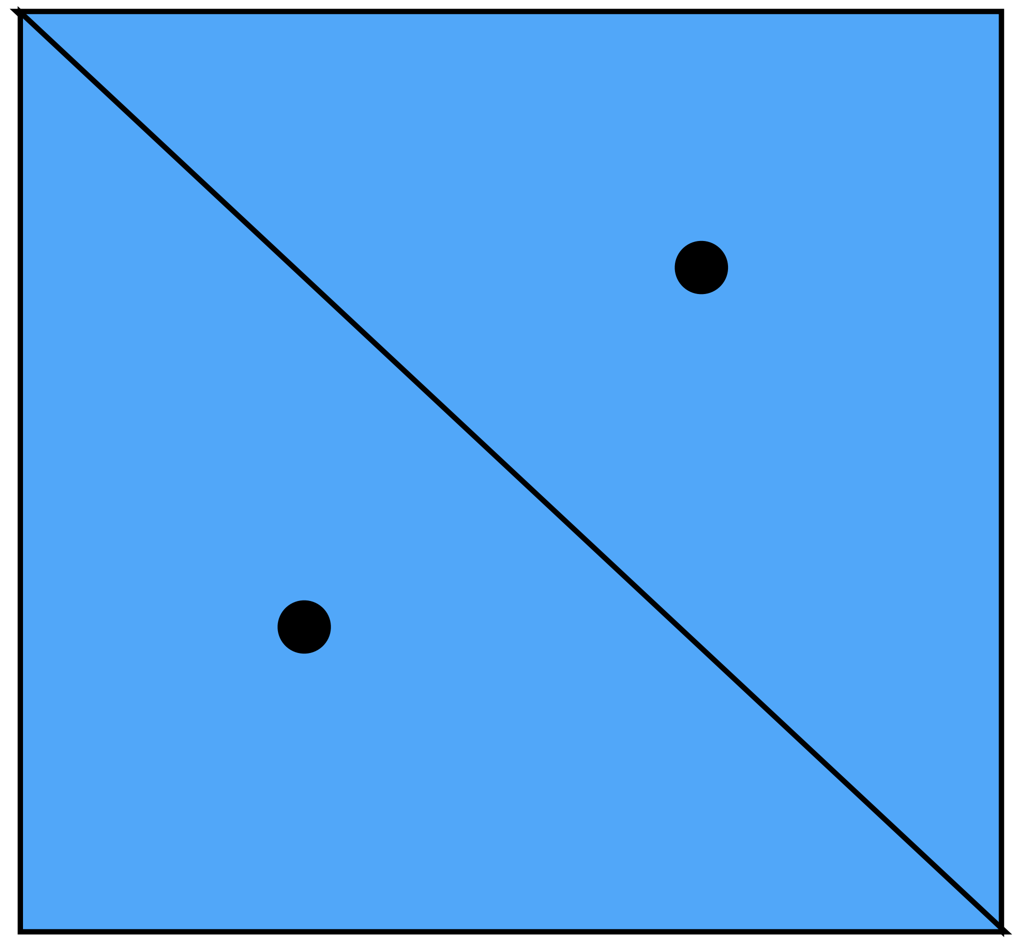



```
For each axis: x,y,z:
   initialize bucket counts to 0, per-bucket bboxes to empty
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Use lowest cost partition found (or make node a leaf)
```

Troublesome cases



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

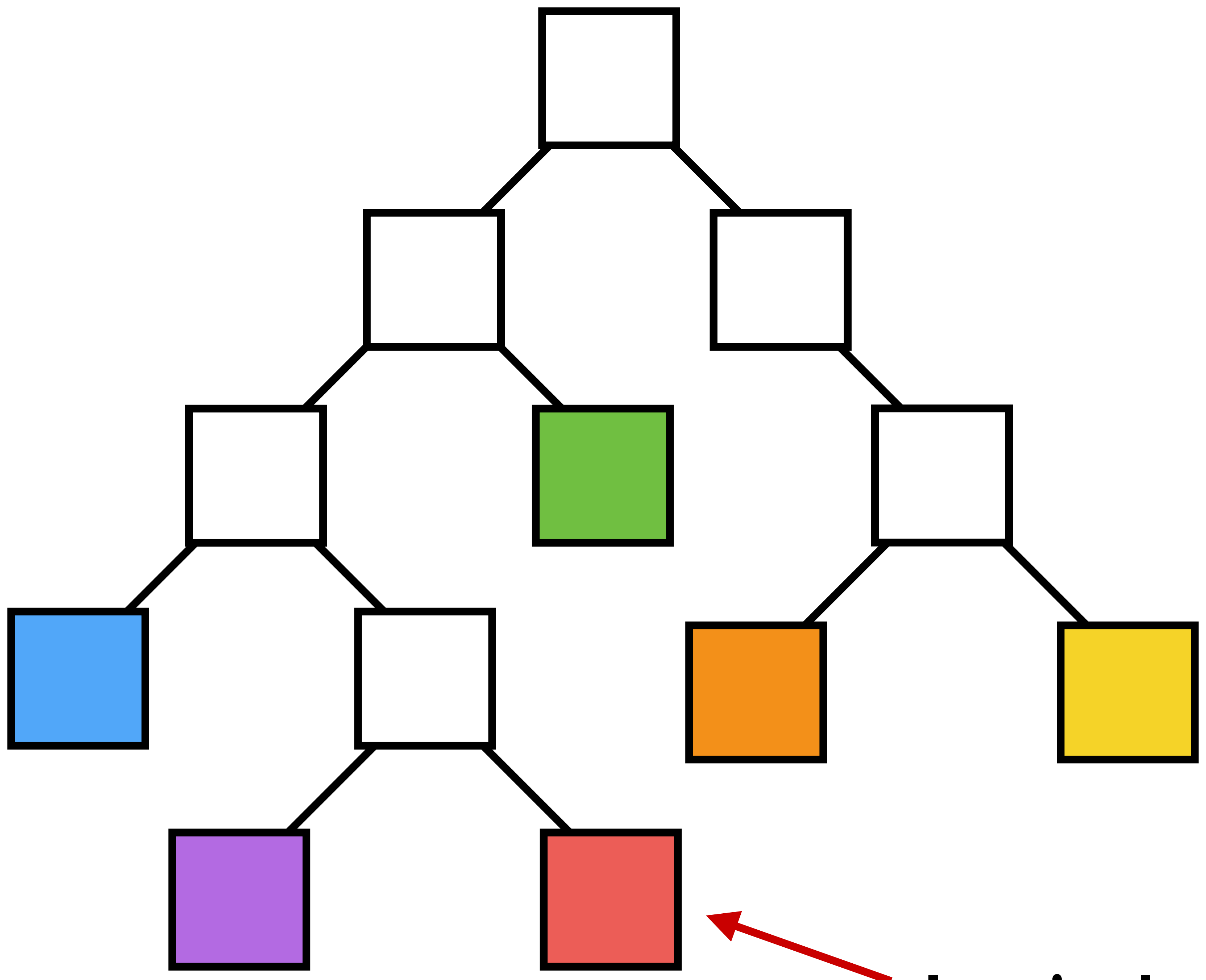


All primitives with same bbox (ray often ends up visiting both partitions)

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

Question

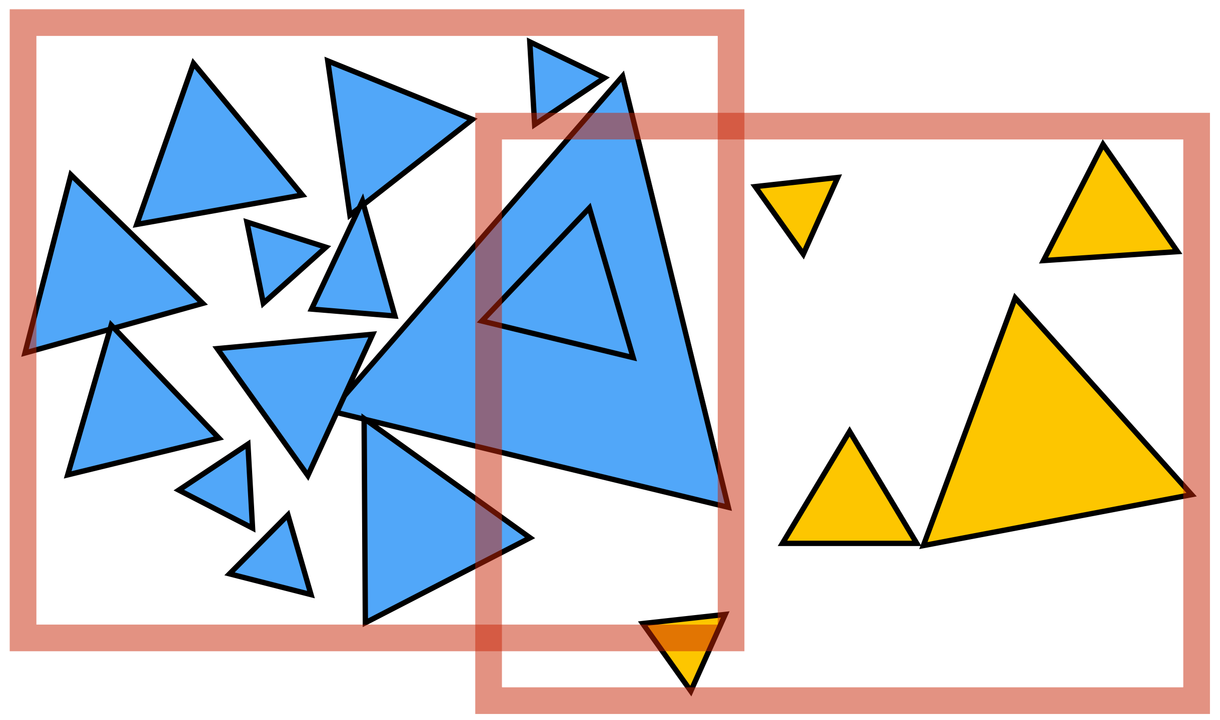
- Imagine you have a valid BVH
- Now I move one of the triangles in the scene to a new location
- How do I "refit" the BVH so it is a valid BVH?



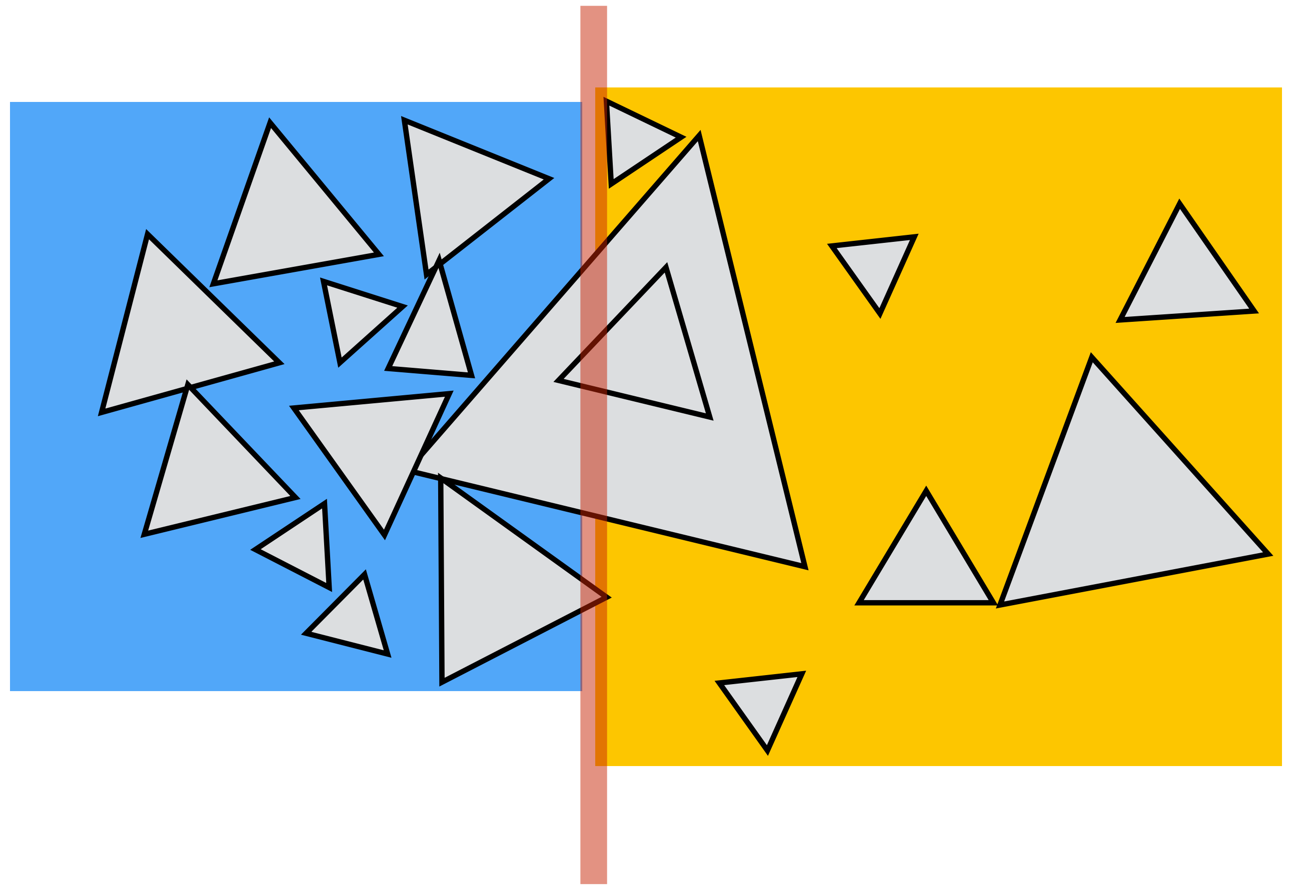
Imagine I moved a triangle
in this red leaf node.

Primitive-partitioning acceleration structures vs. space-partitioning structures

- Primitive partitioning (e.g, bounding volume hierarchy): partitions primitives into disjoint sets (but sets of primitives may overlap in space)



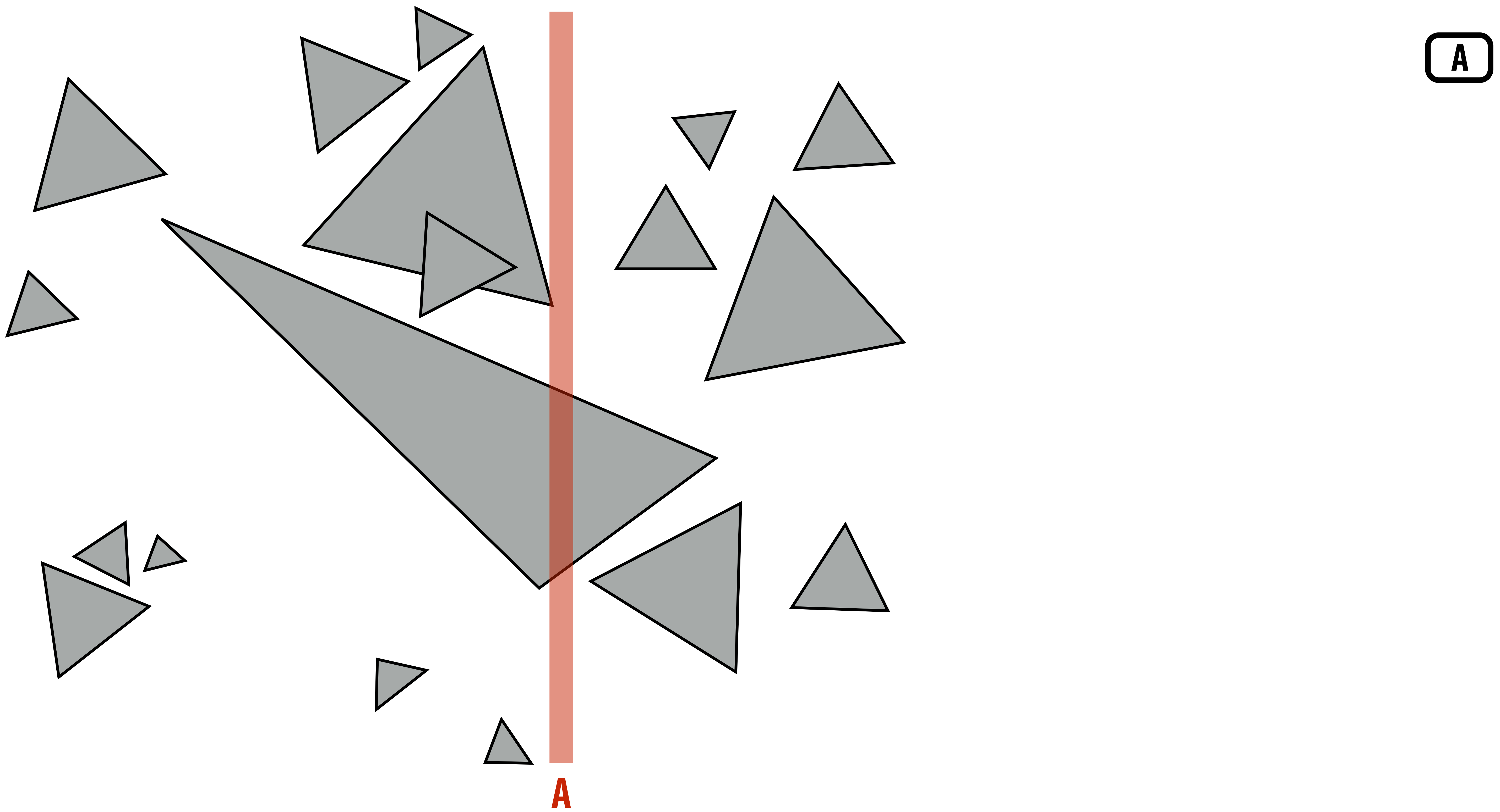
- Space-partitioning (e.g. grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



So far I've only showed you a primitive partitioning structure (a BVH), let's look at two space partitioning structures.

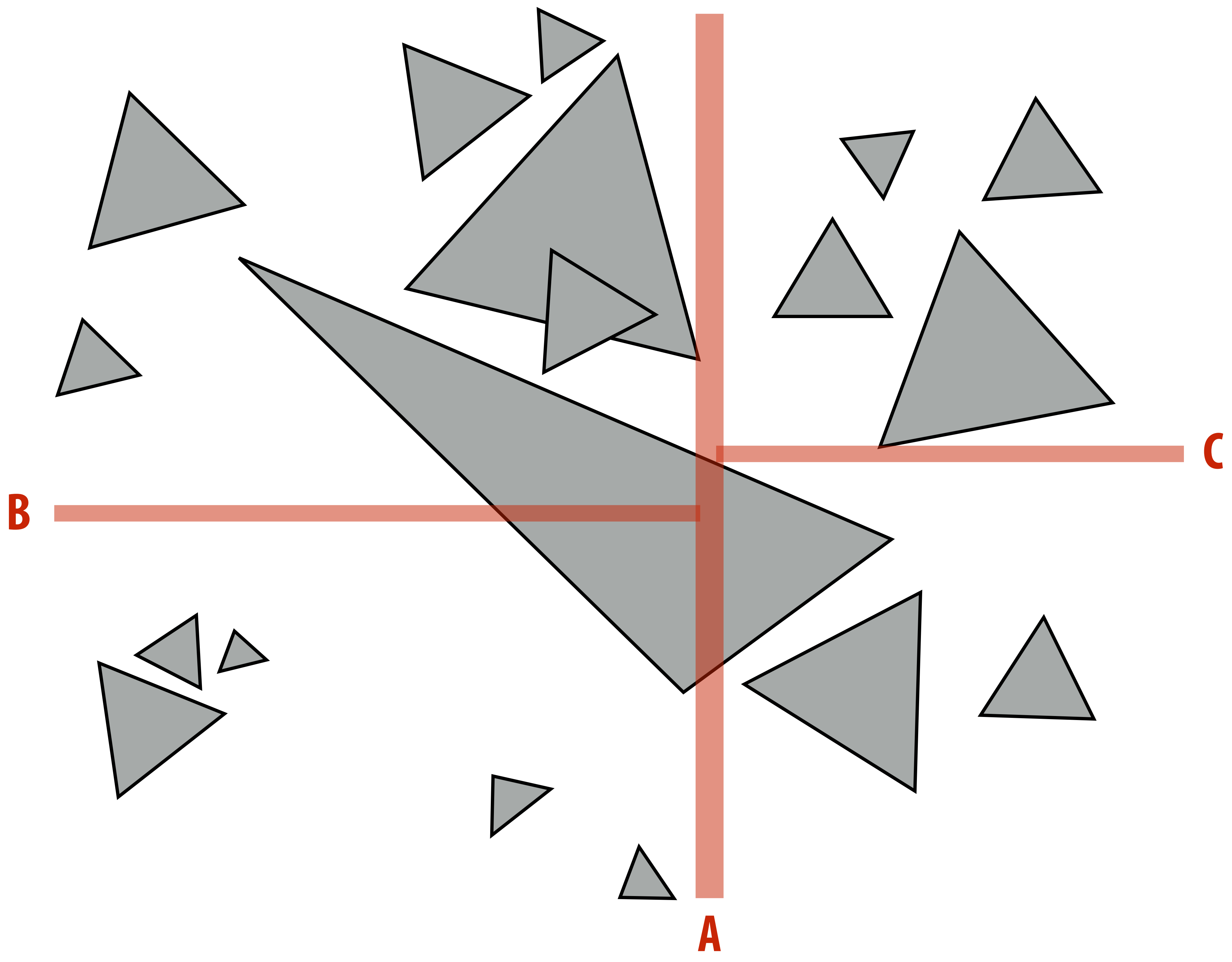
K-D tree

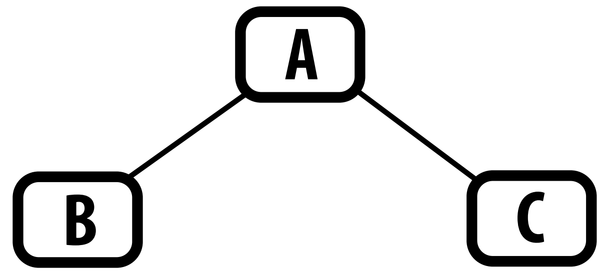
- Recursively partition <u>space</u> via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits



K-D tree

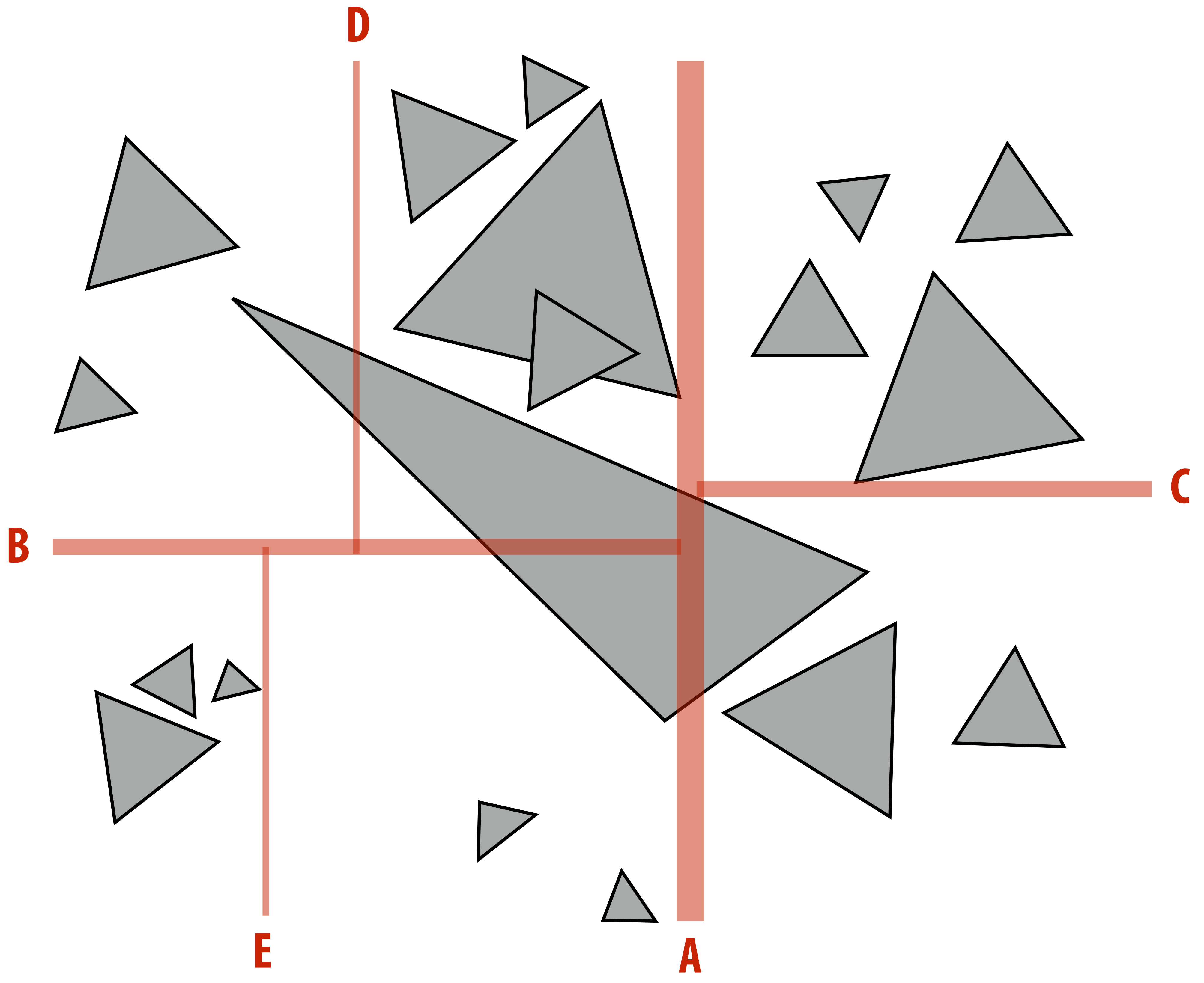
- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits

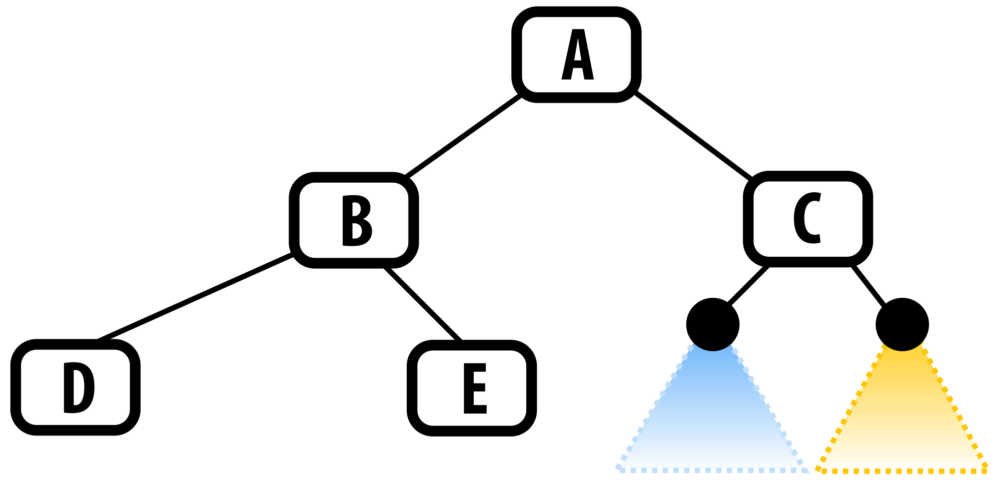




K-D tree

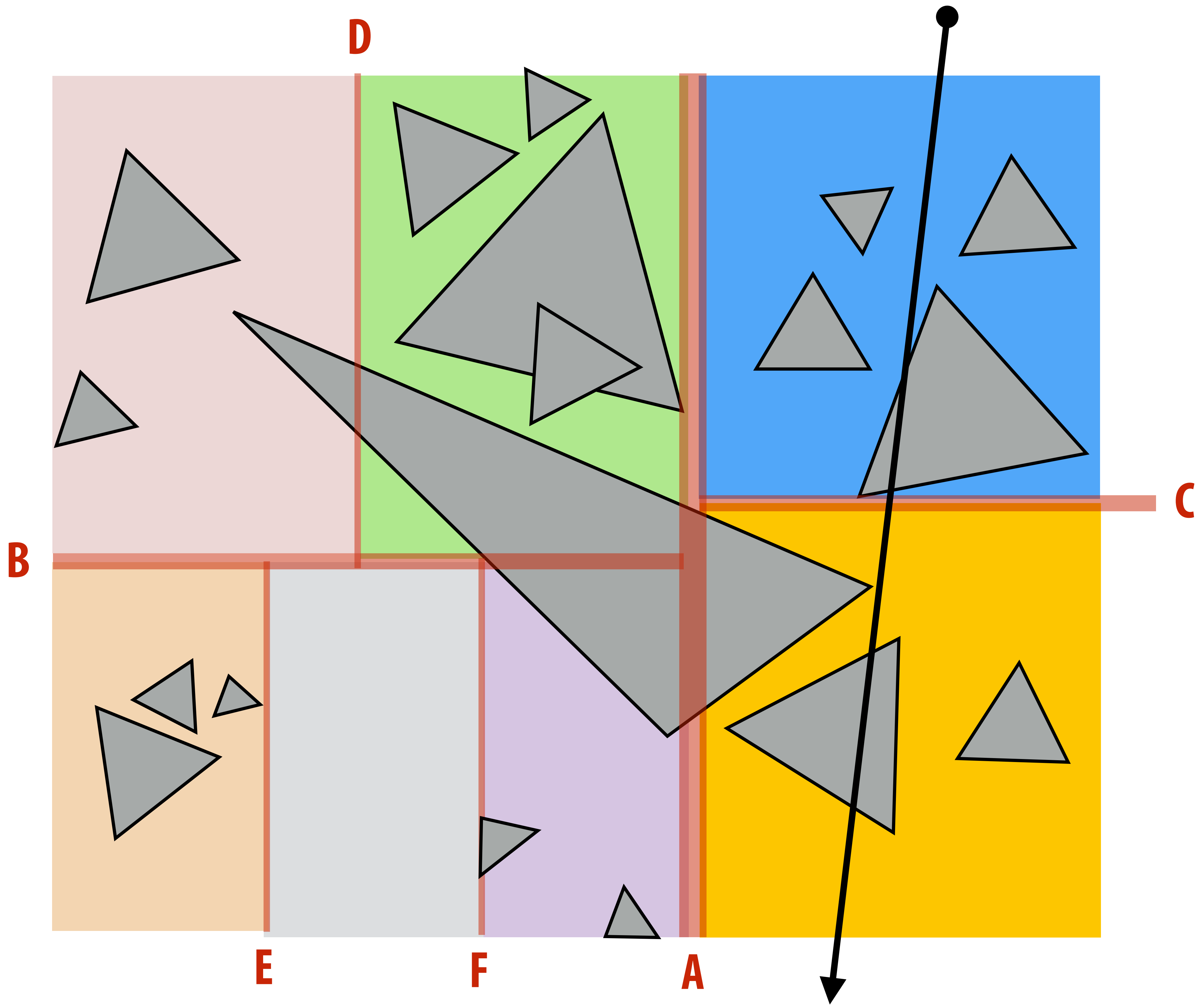
- Recursively partition <u>space</u> via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits

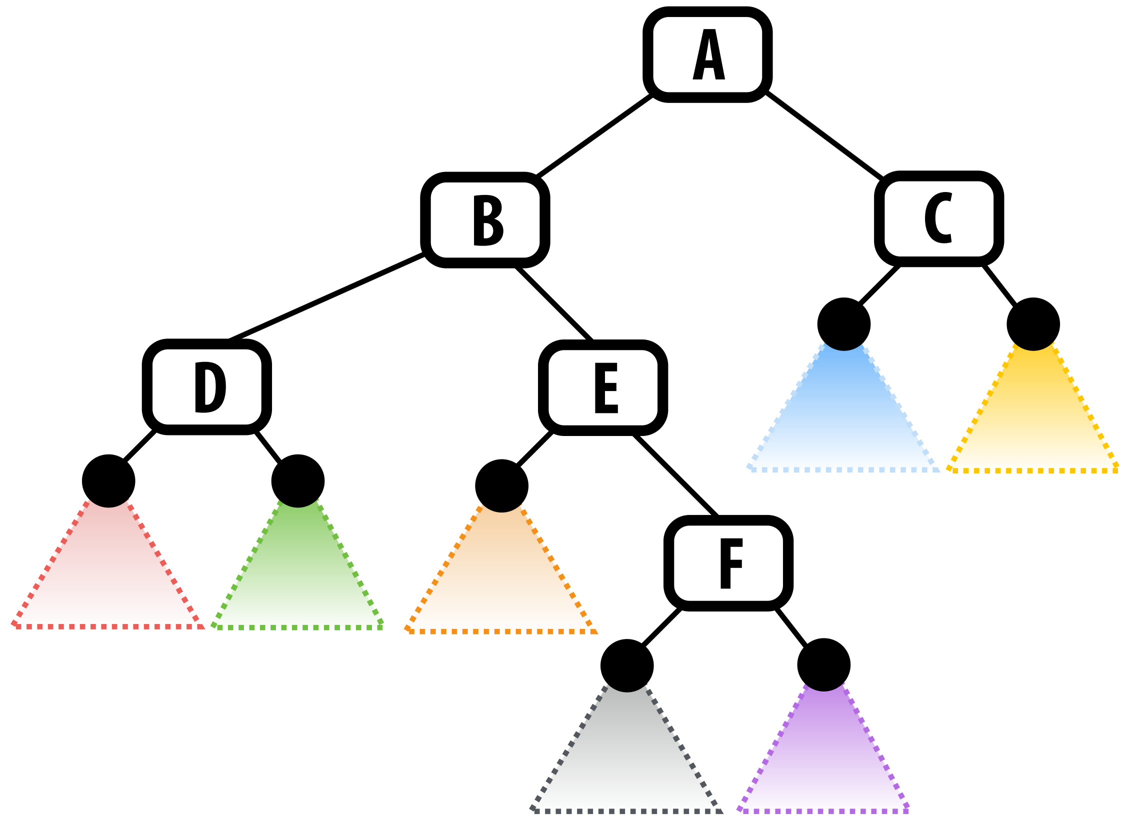




K-D tree

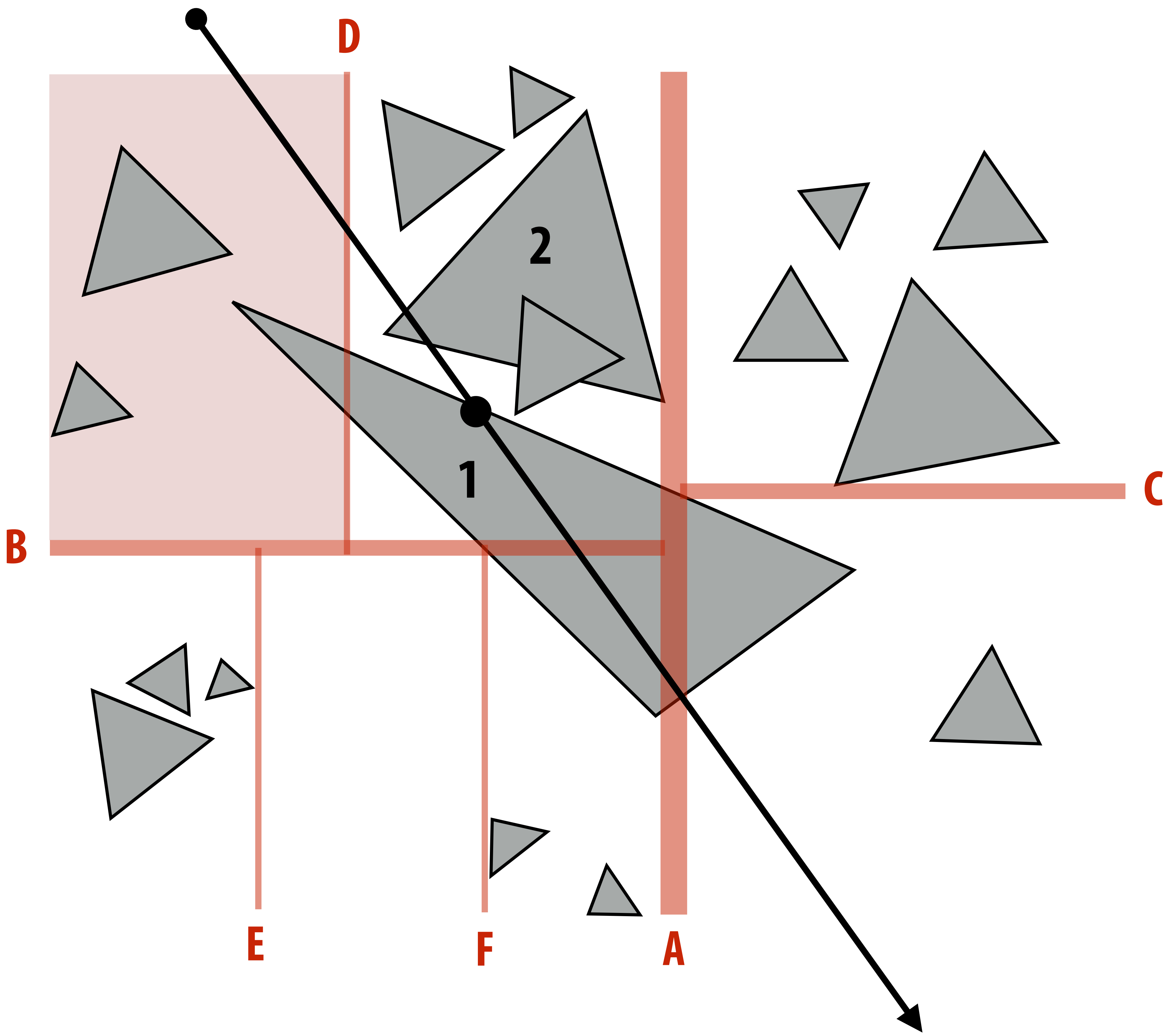
- Recursively partition space via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in strict front-to-back order
  - So unlike BVH, can terminate search after first hit is found

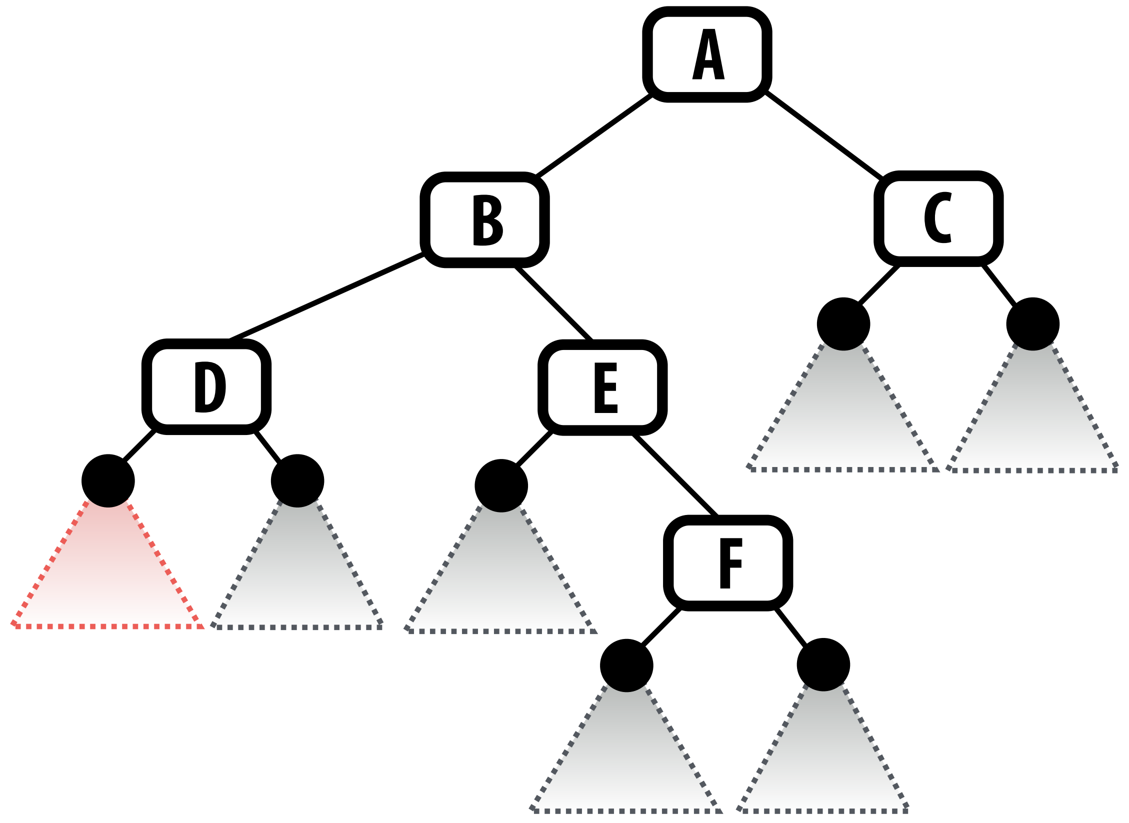




Challenge: objects overlap multiple tree nodes

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)

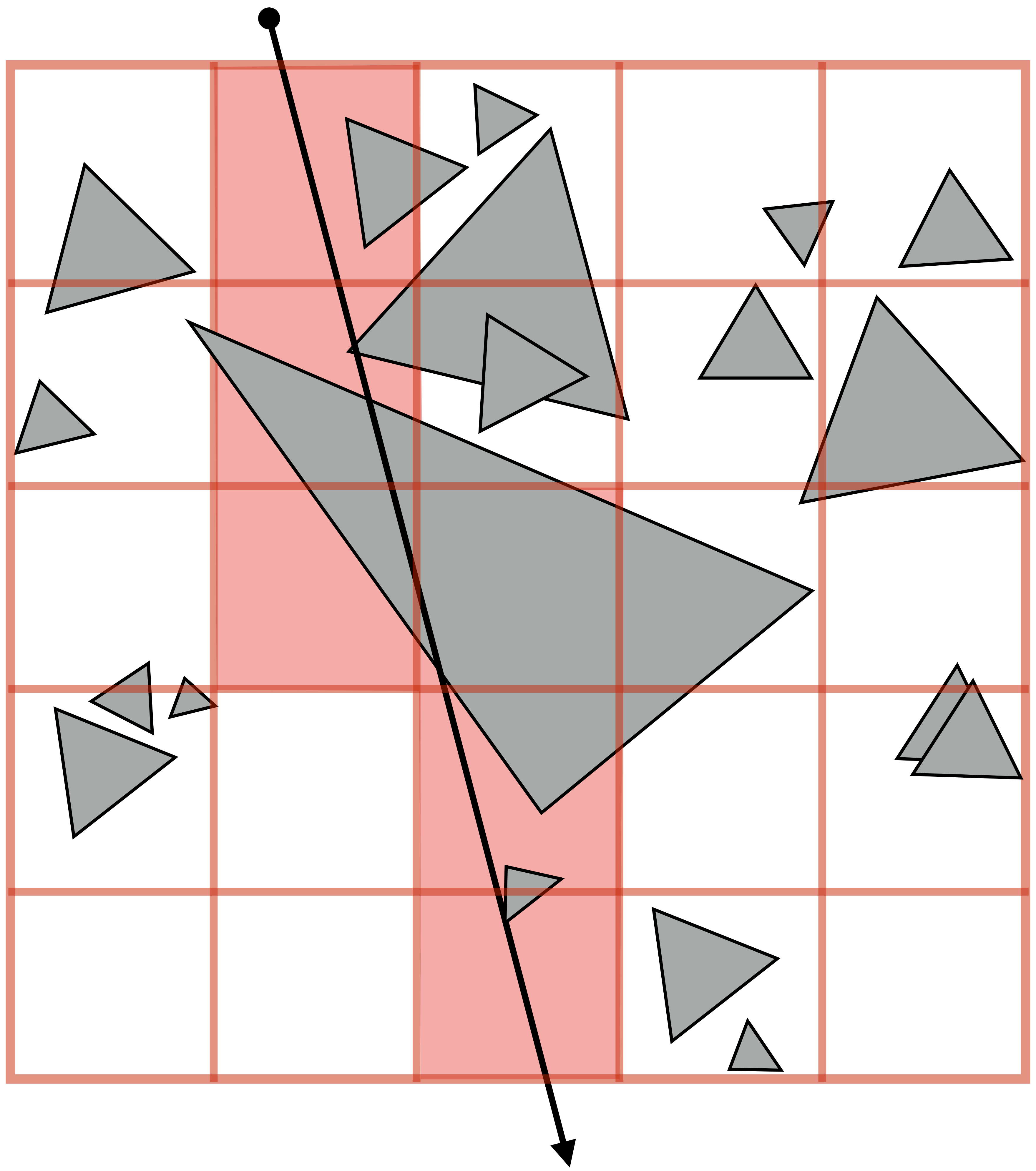
Solution: require primitive intersection point to be within spatial volume current leaf node.

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

* Caching hit info ("mailboxing") can be used to avoid repeated intersections

Uniform grid
(a very simple space partitioning hierarchy)

Uniform grid



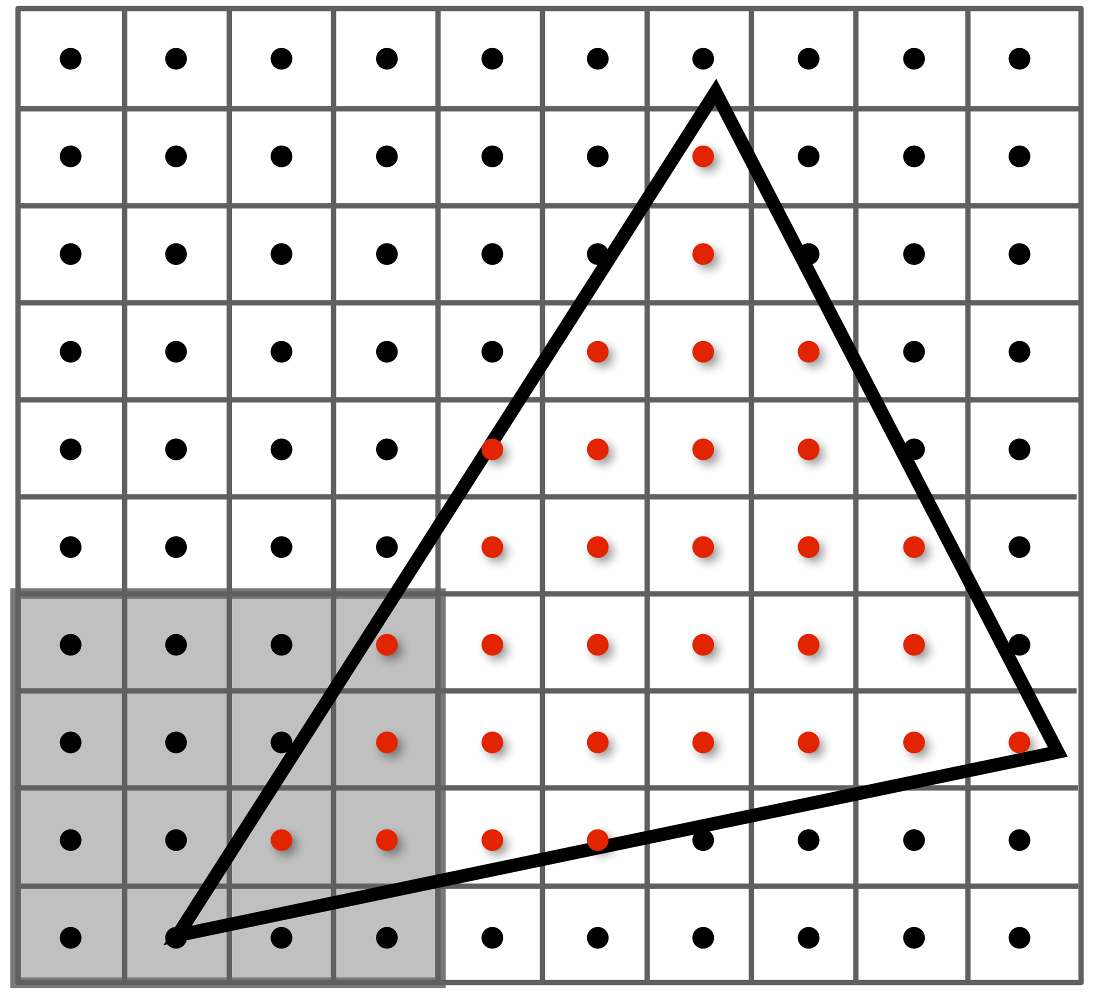
- Partition space into equal sized volumes (volume-elements or "voxels")
- Each grid cell contains primitives that overlap the voxel.
  - Cheap to construct acceleration structure
- Walk ray through volume in order
  - Efficient implementation possible (think: *3D line rasterization*)
  - Only consider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

Consider tiled triangle rasterization

```
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 tile of screen samples touching bbox of triangle:   // loop 2: tiles
       if (triangle does not overlap tile)
          continue;
       for each 2D sample s in tile:            // loop 3: 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]
```

For each TILE of image

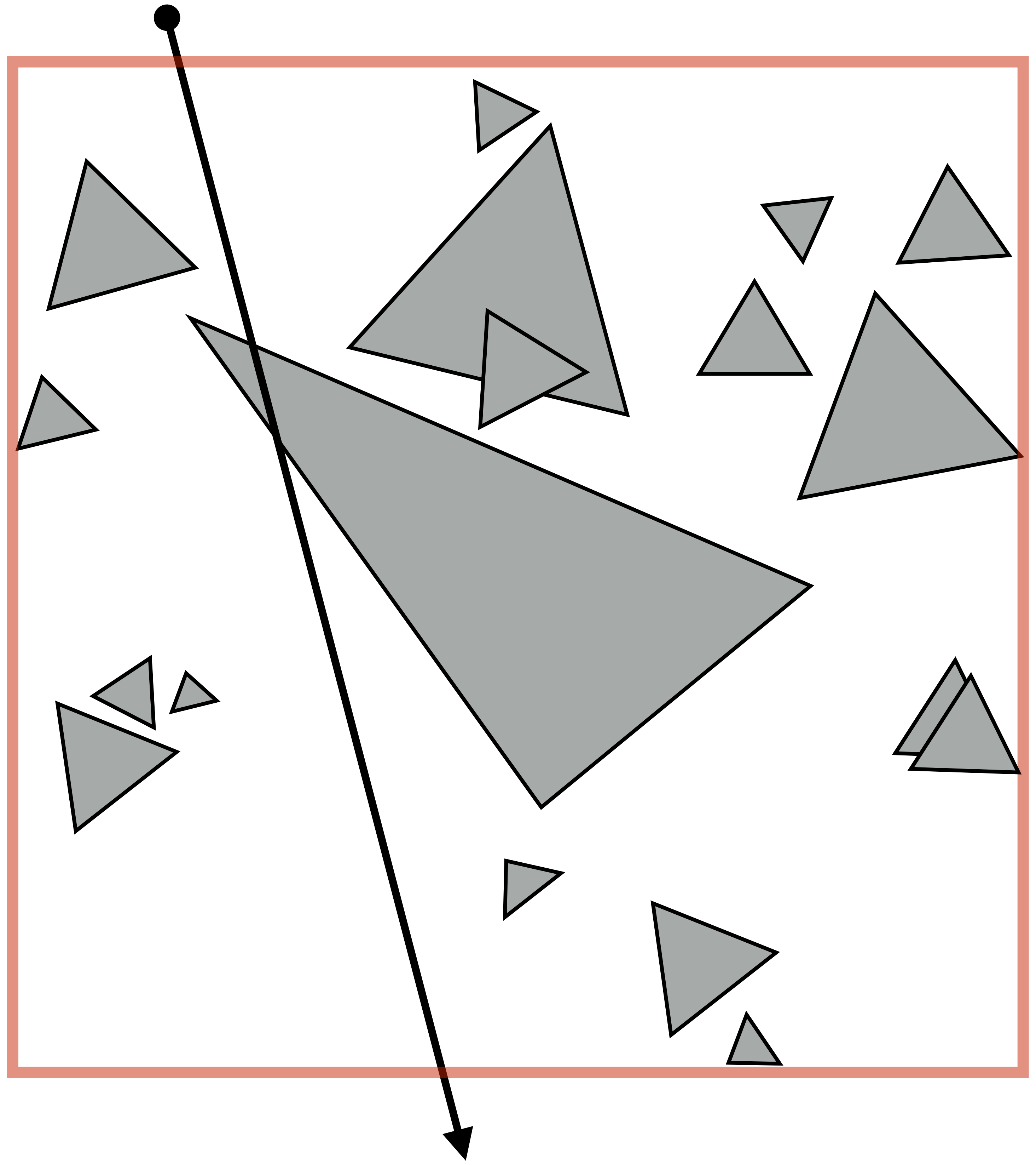
If triangle overlaps tile, check all samples in tile



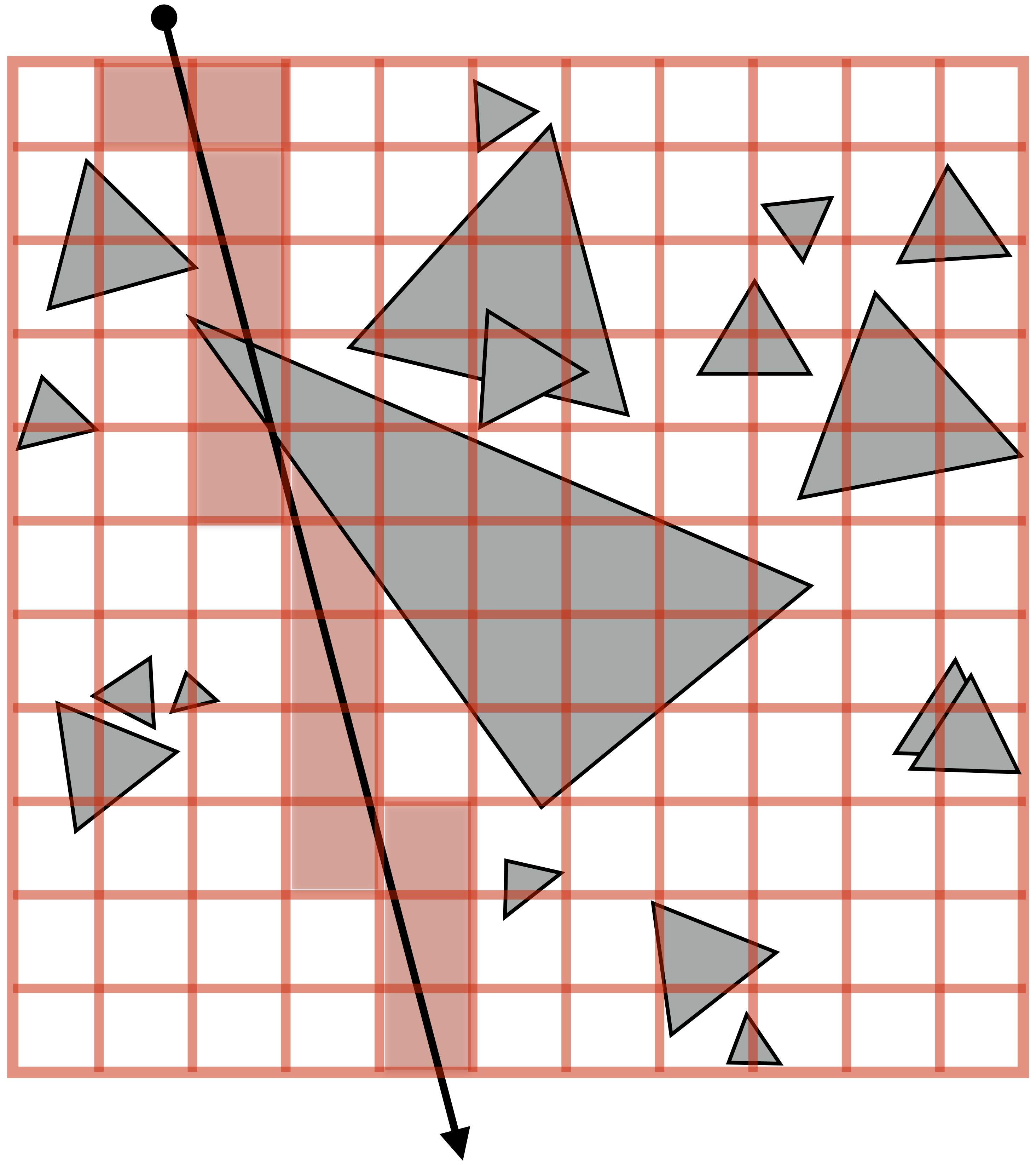
What does this rasterization strategy remind you of? :-)

Think about ray-casting using a uniform grid.

What should the grid resolution be?



Too few grids cell: degenerates to brute-force approach

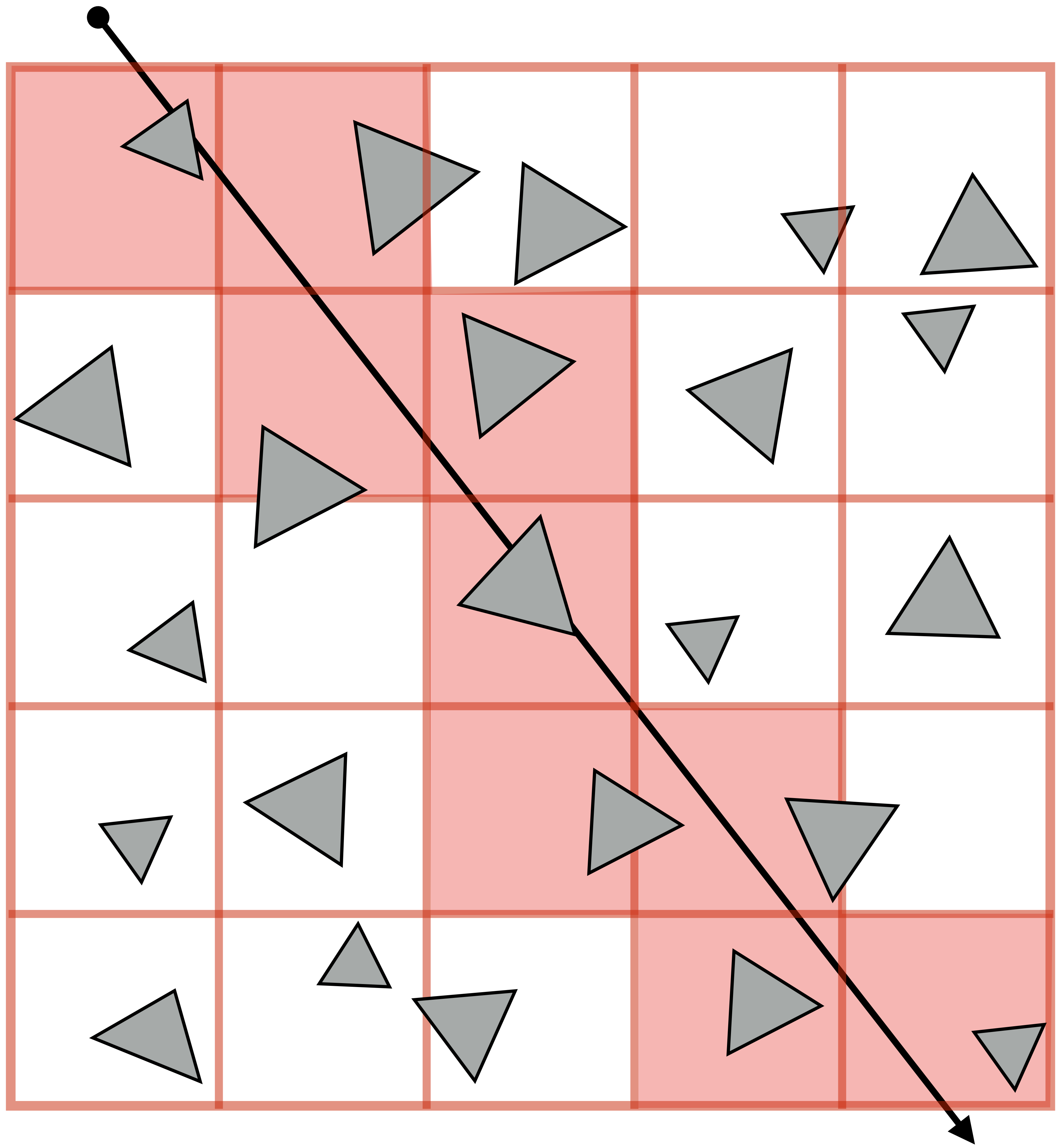


Too many grid cells: incur significant cost traversing through cells with empty space

Grid size heuristic

Choose number of cells ~ total number of primitives

(yields constant prims per cell for any scene size — assuming uniform distribution of primitives)



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

(assuming 3D grid)

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

When uniform grids work well: uniform distribution of primitives in scene

Terrain / height fields:

[Image credit: Misuba Renderer]

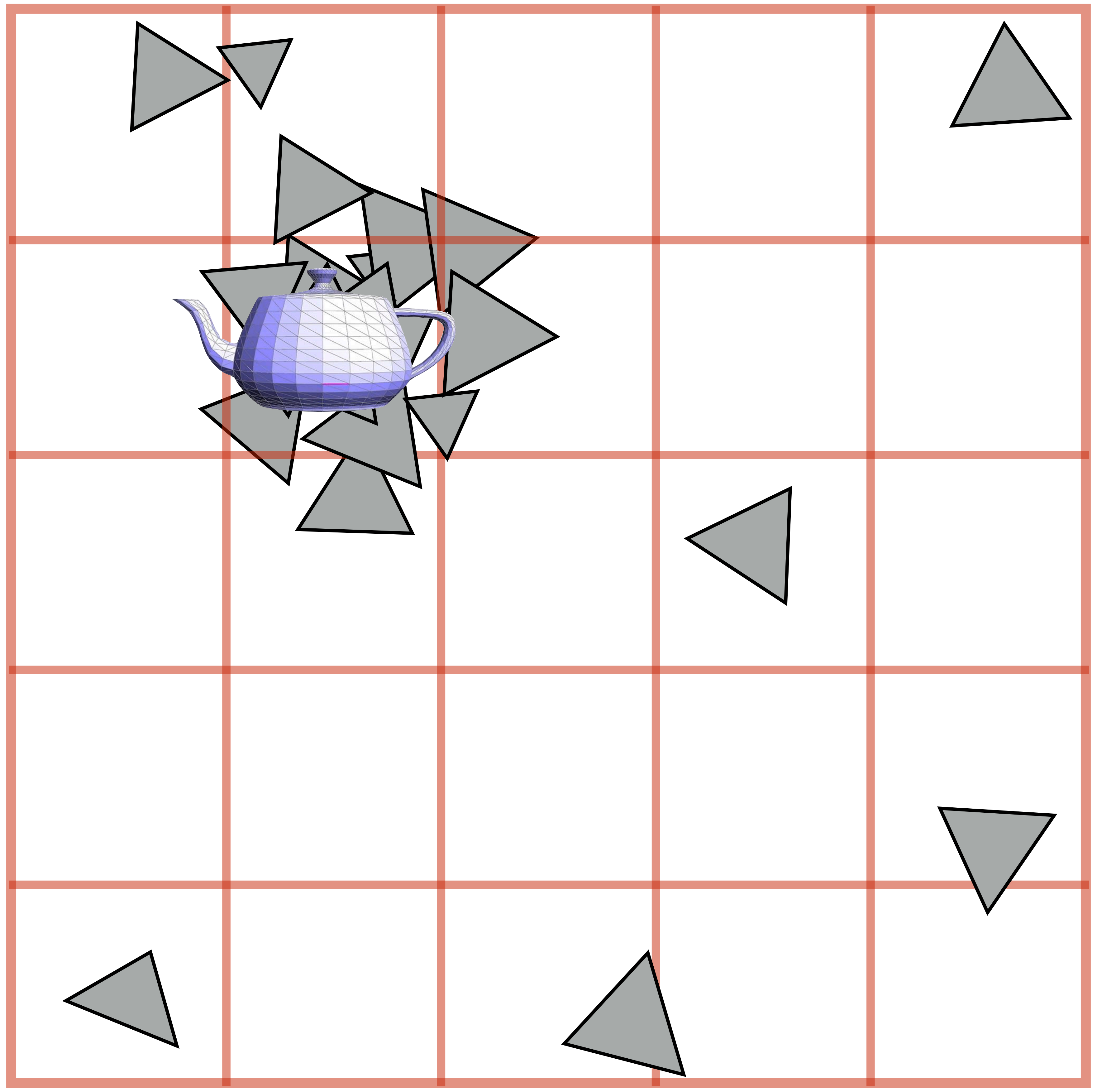
Field of grass

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

Slide credit: Pat Hanrahan

Uniform grids 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)

When uniform grids do not work well: non-uniform distribution of geometric detail

Jun Yan, Tracy Renderer

When uniform grids do not work well: non-uniform distribution of geometric detail

[Image credit: Pixar]

Quad-tree / octree

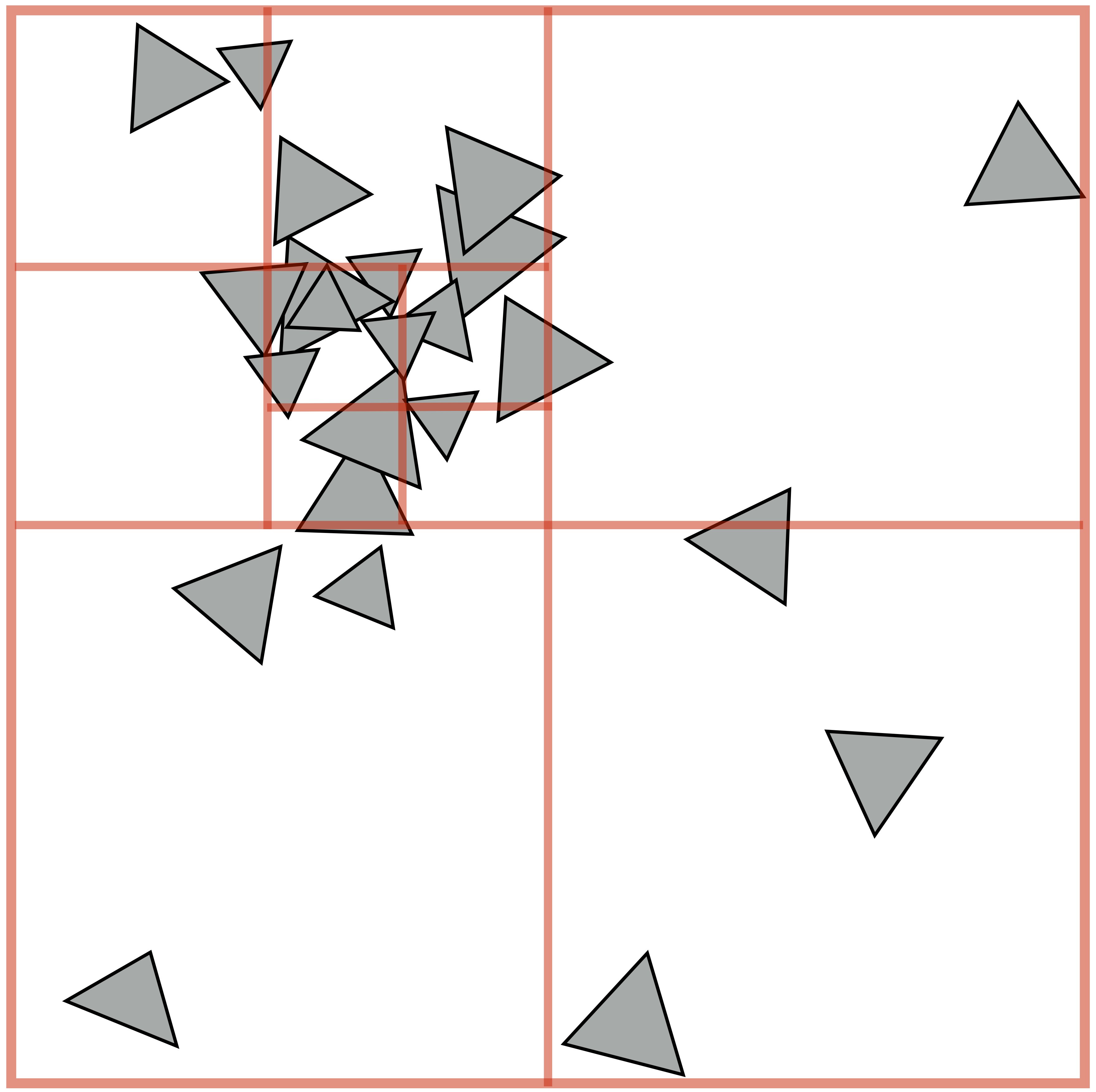
Like uniform grid: easy to build
(don't have to choose partition planes)

Has greater ability to adapt to location of scene geometry than a uniform grid.

But lower intersection performance than a hierarchical structure like a BVH or K-D tree
(the structure only has limited ability to adapt to distribution of scene geometry)

Quad-tree: nodes have 4 children
(partitions 2D space, like the example to the right)

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 into non-overlapping regions
    - 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

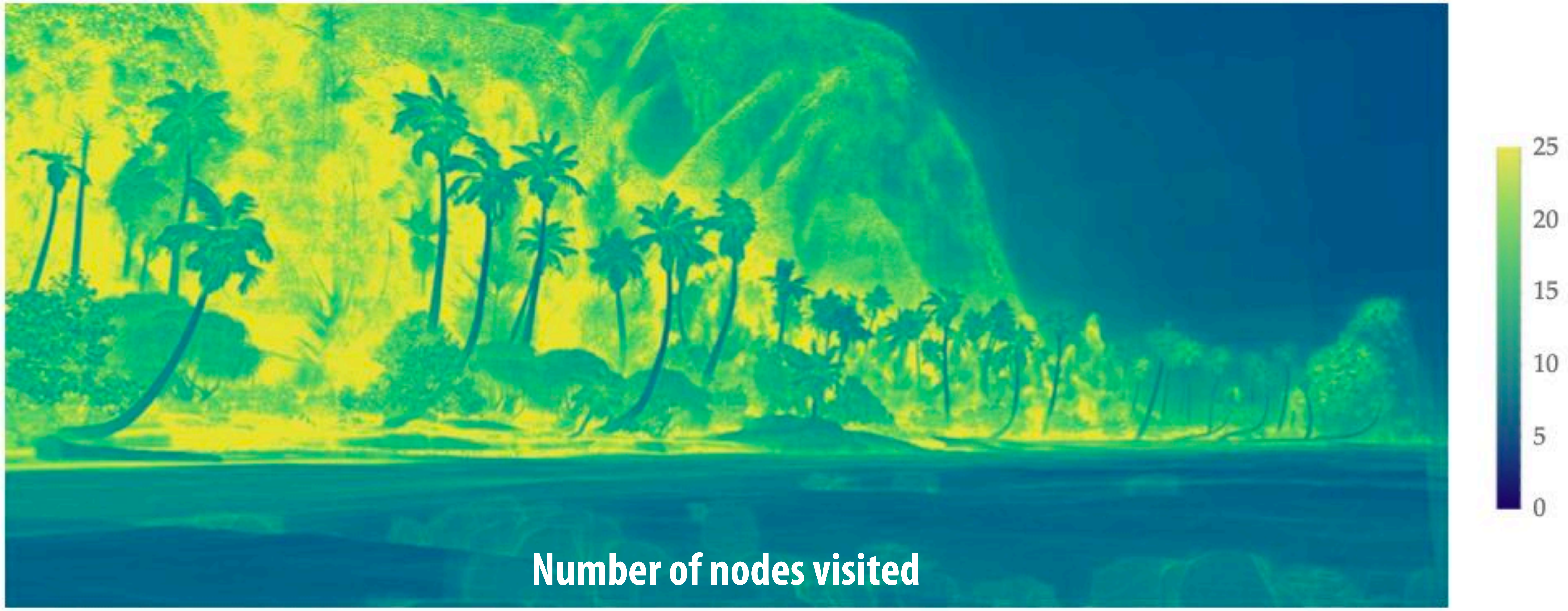
- Non-adaptive accelerations structures (uniform grids)
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed

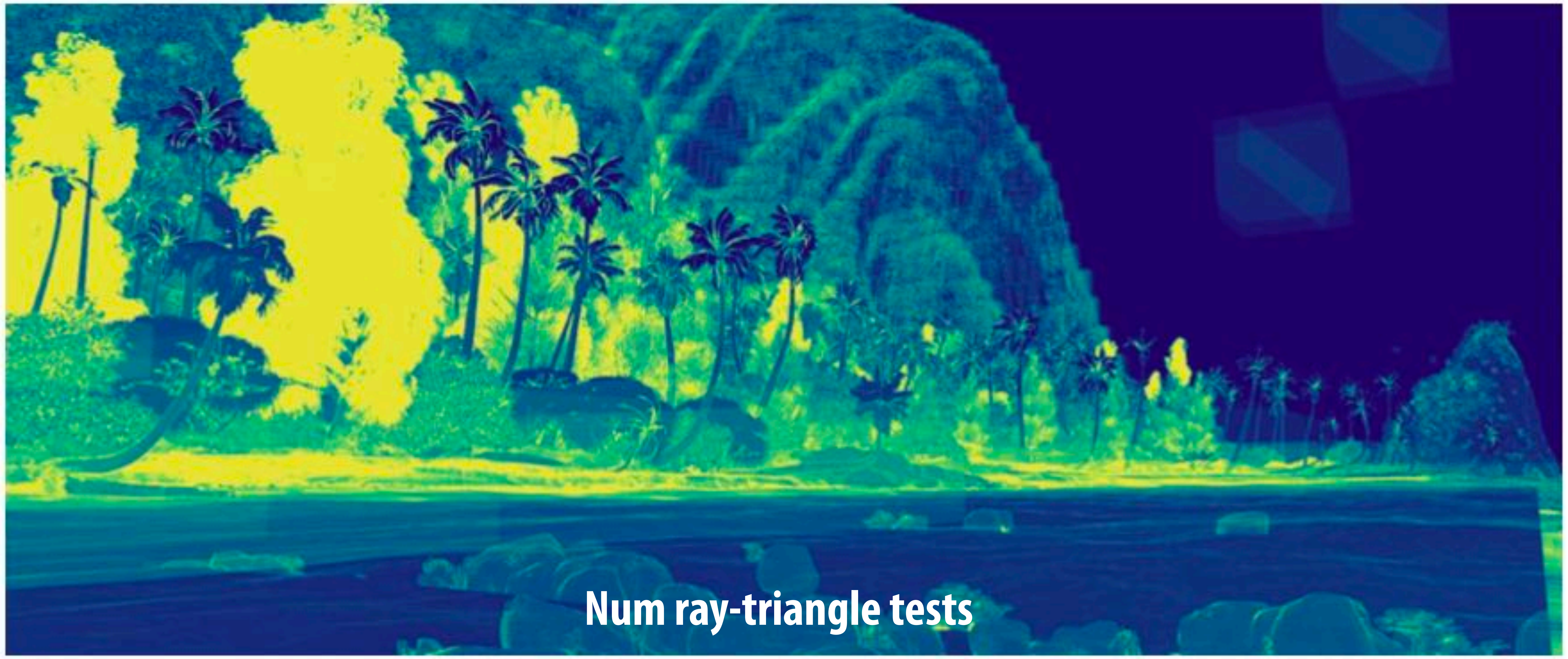
- Many, many combinations thereof…

Understanding BVH Performance

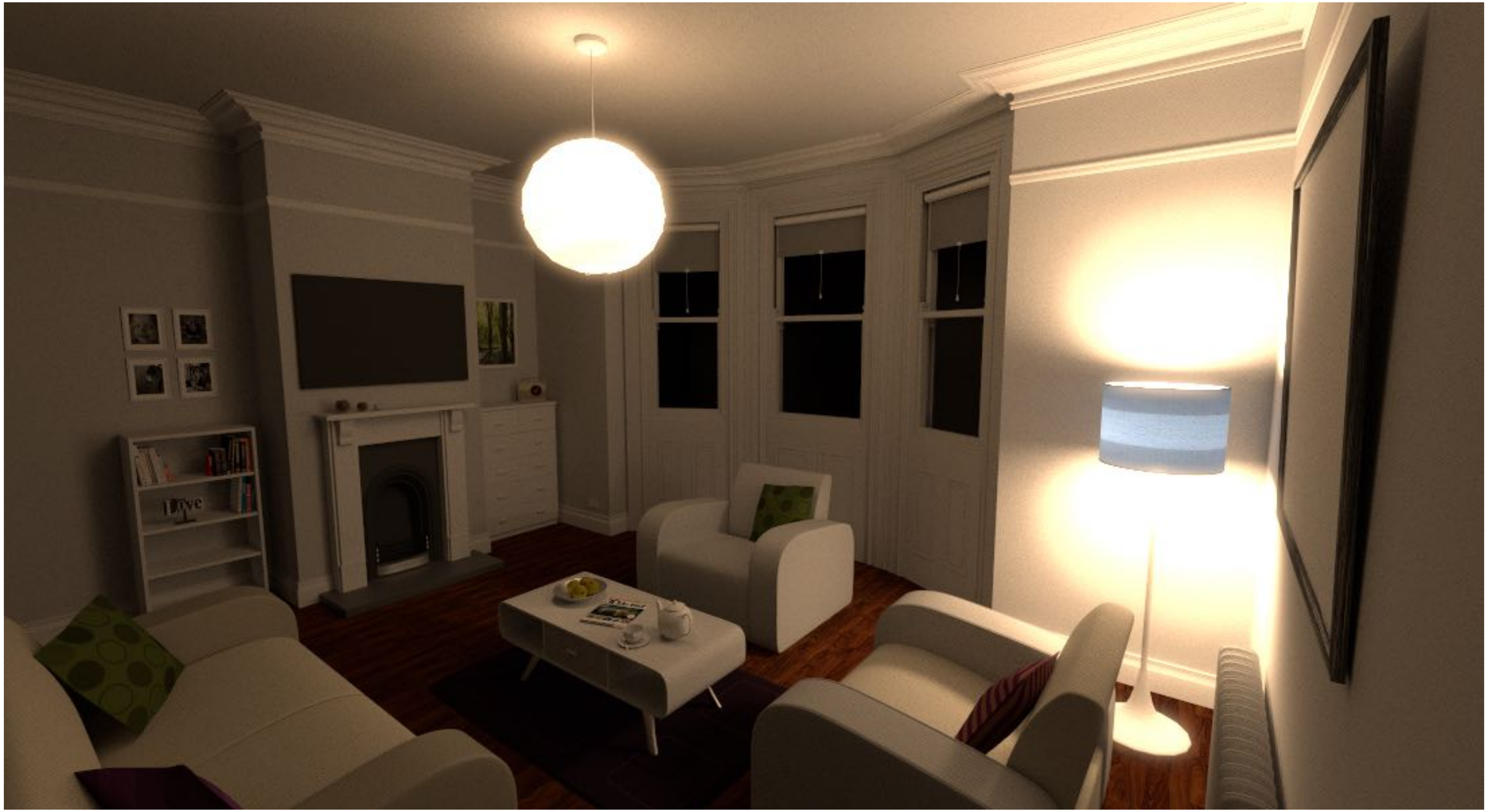
Recall: Moana scene

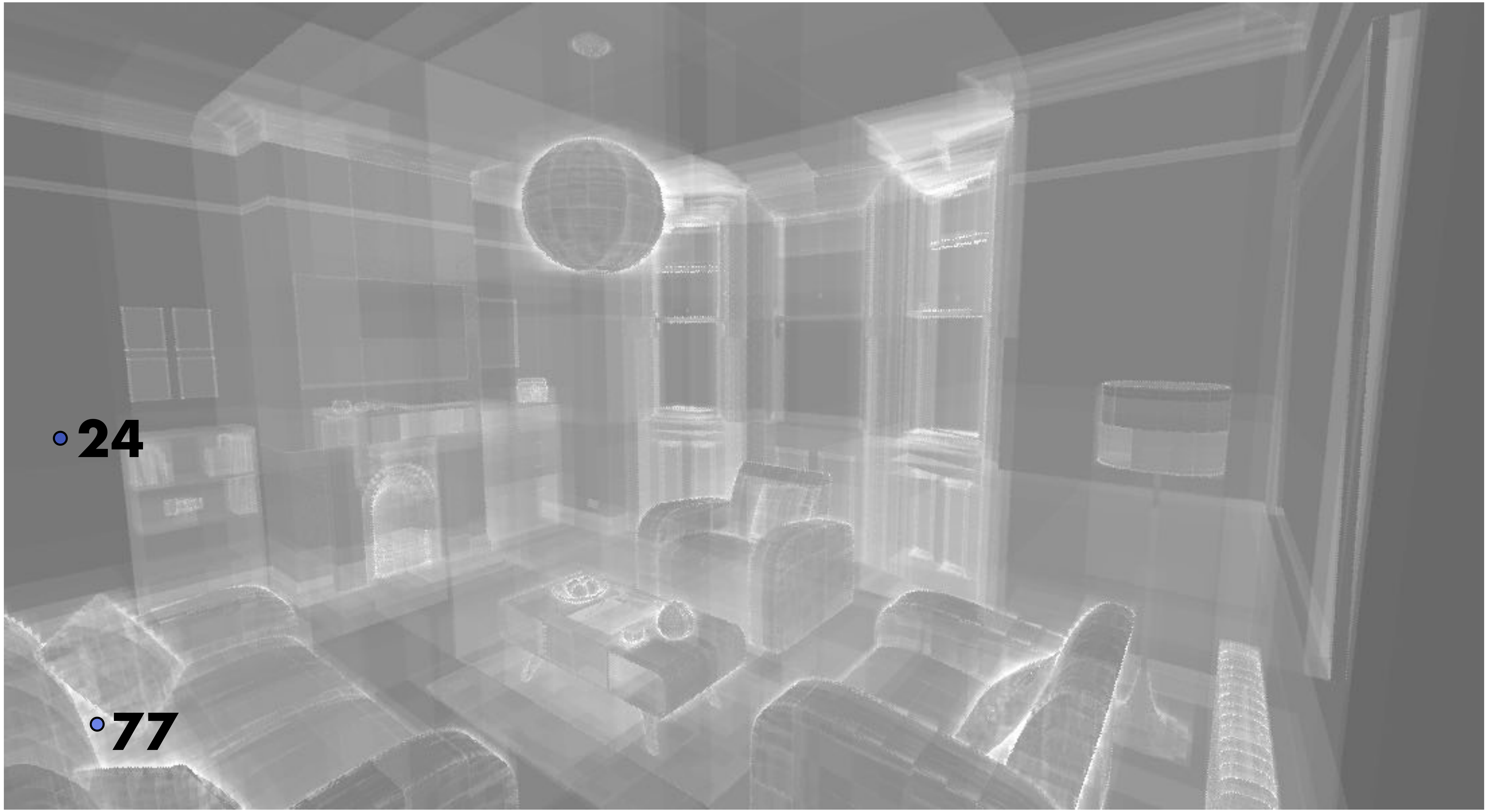
Moana costs





Another example

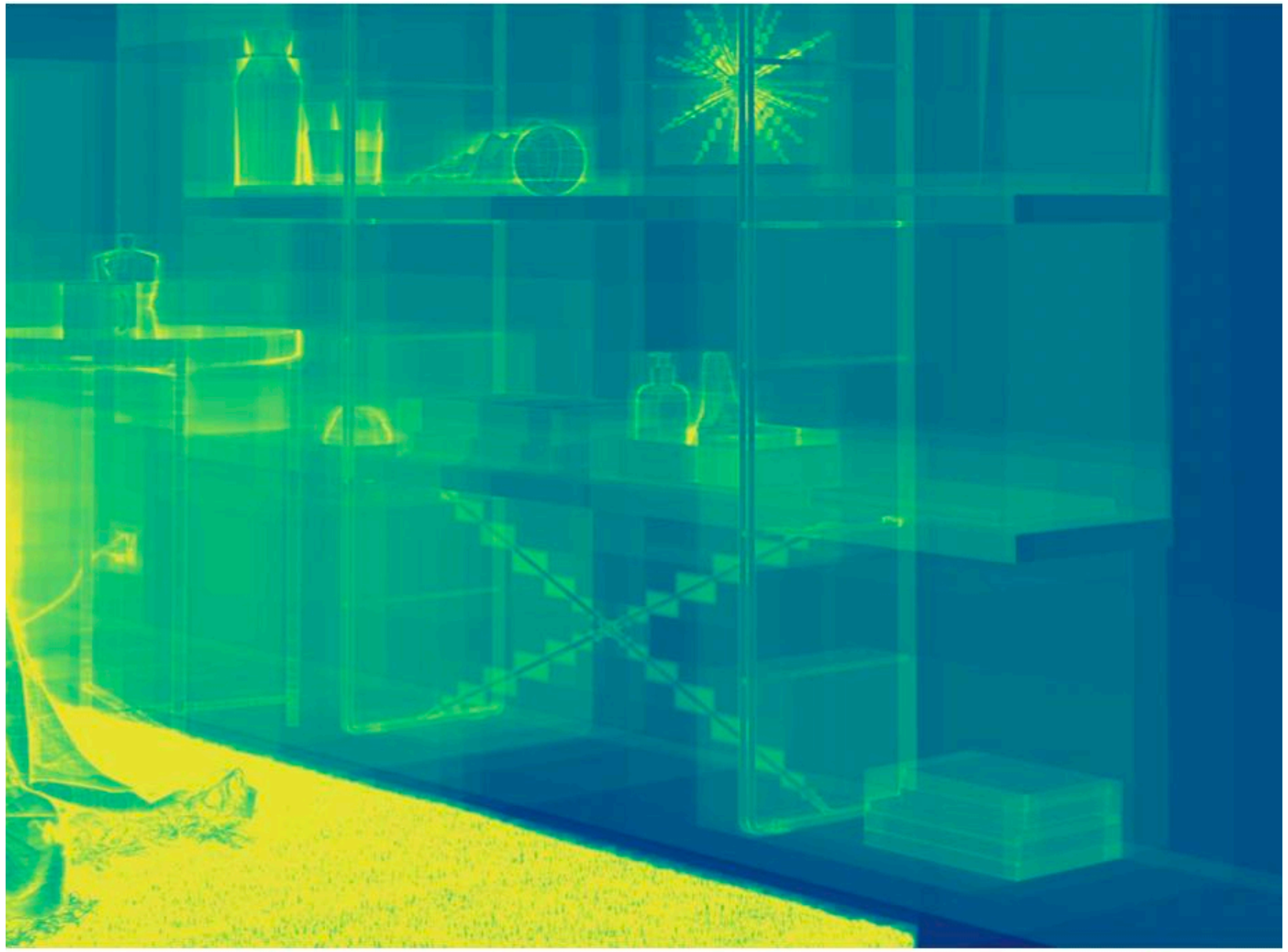




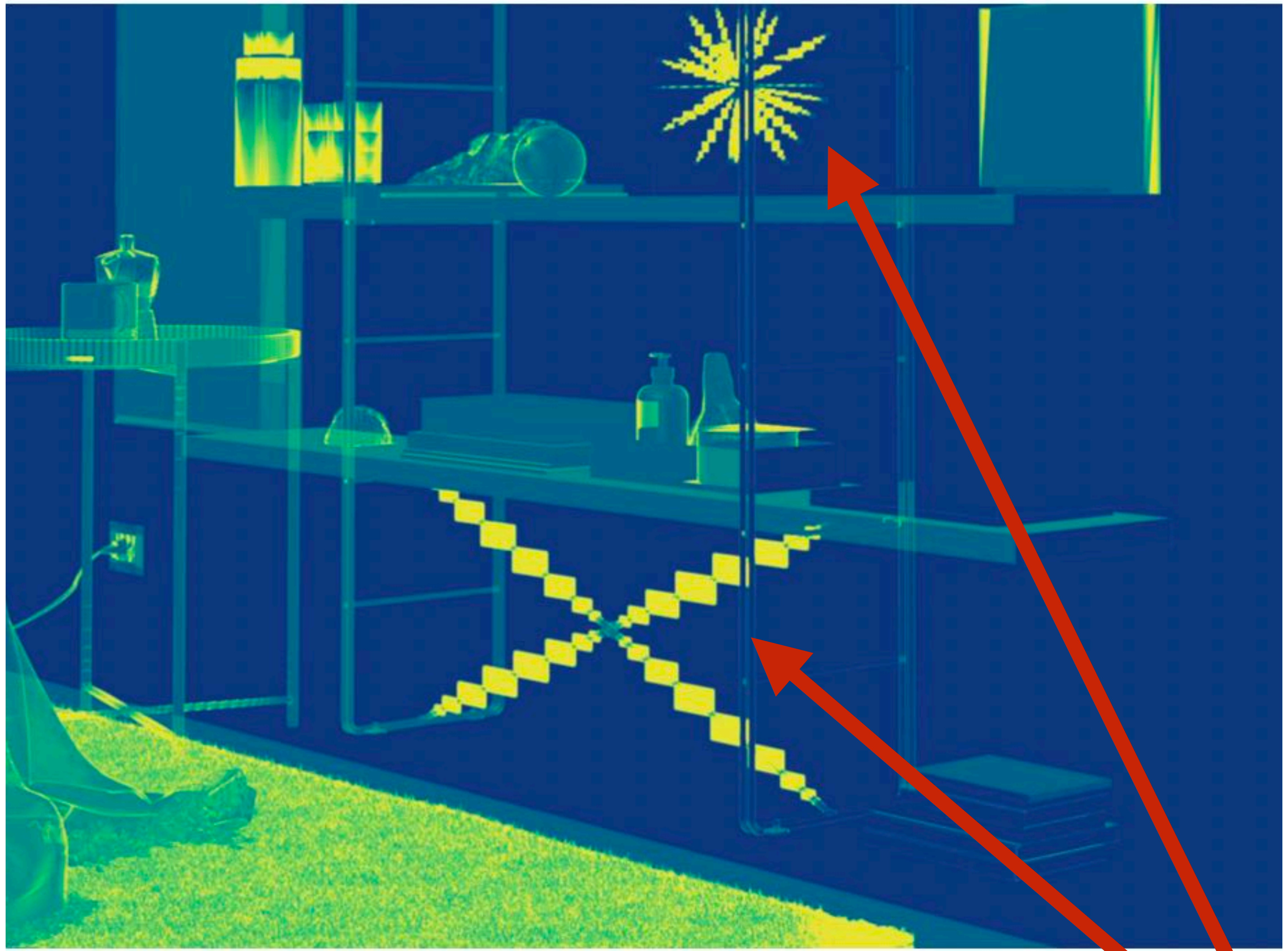
Number of BVH Nodes Visited

Number of Ray-Triangle Tests (when using BVH)

Another example: diagonal geometry (not axis aligned)



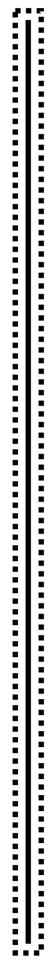
Number of nodes visited



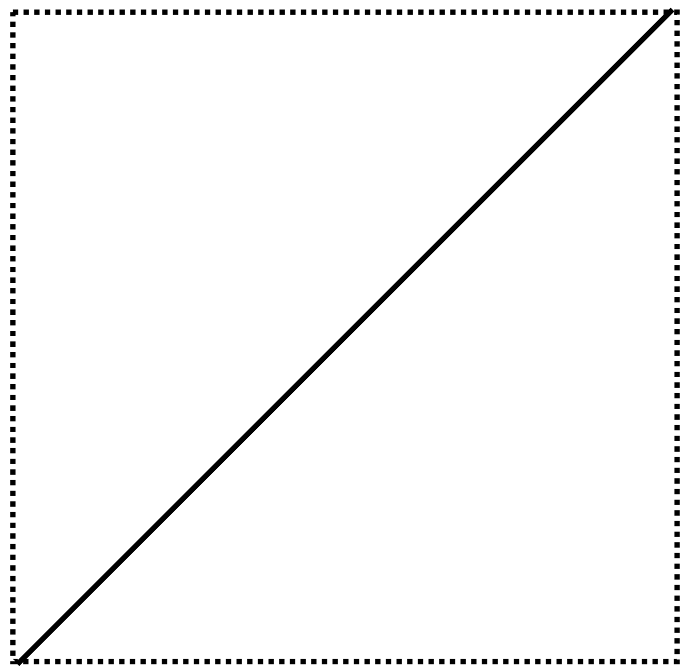
Num ray-triangle tests

??

Axis-alignment and performance

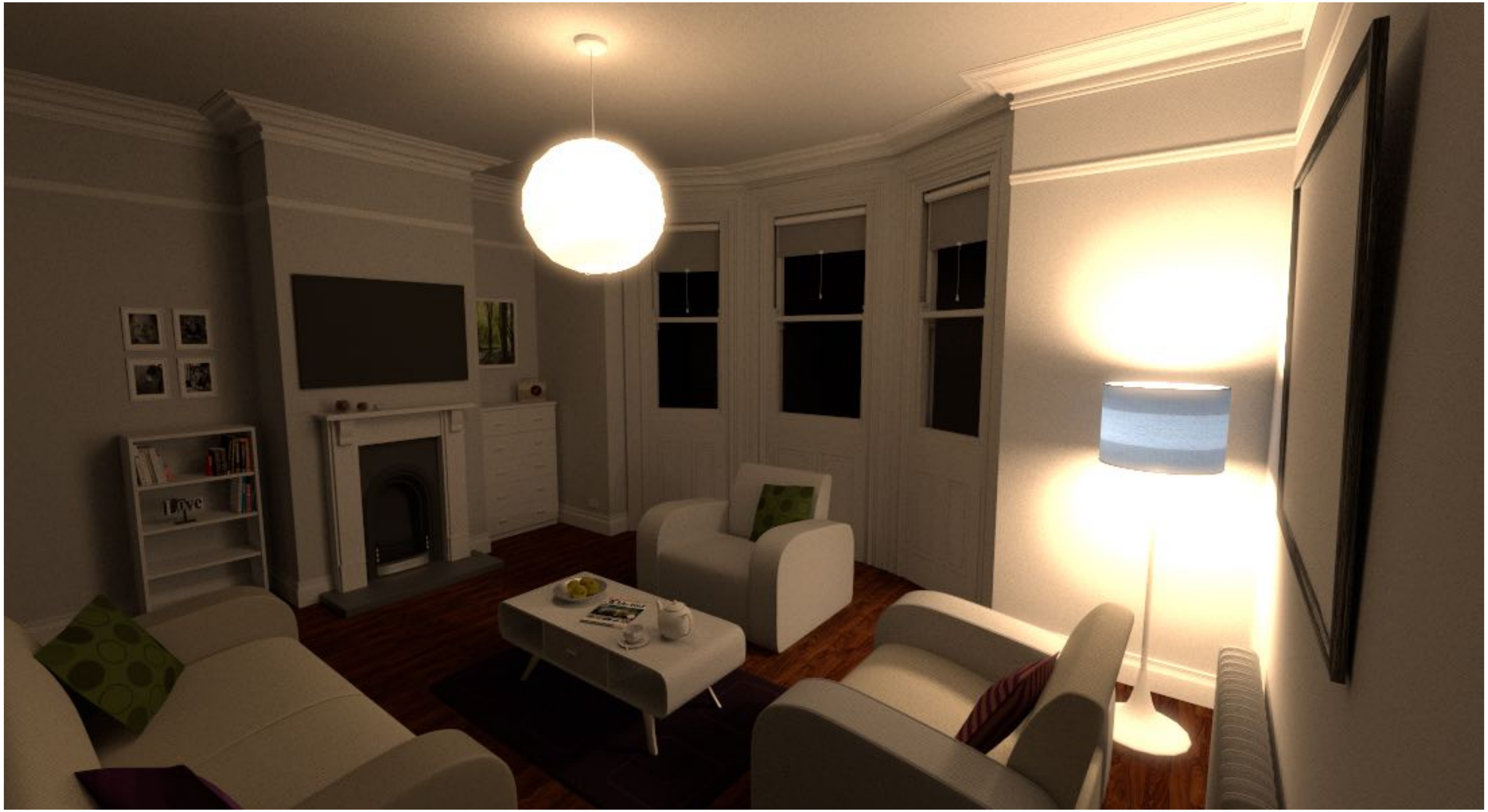


Wall and its
bounding box



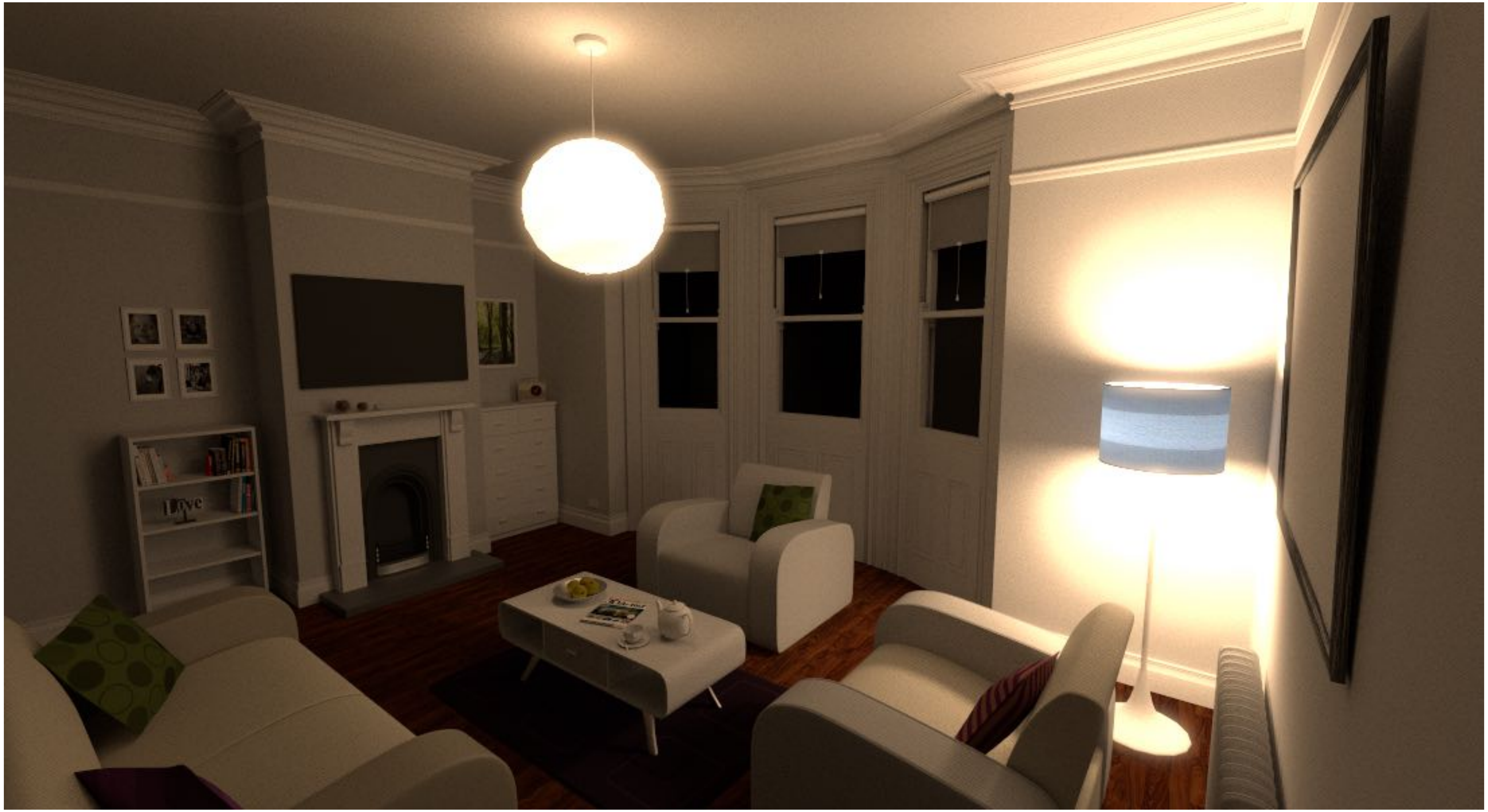
Rotated wall and its
bounding box

Original scene



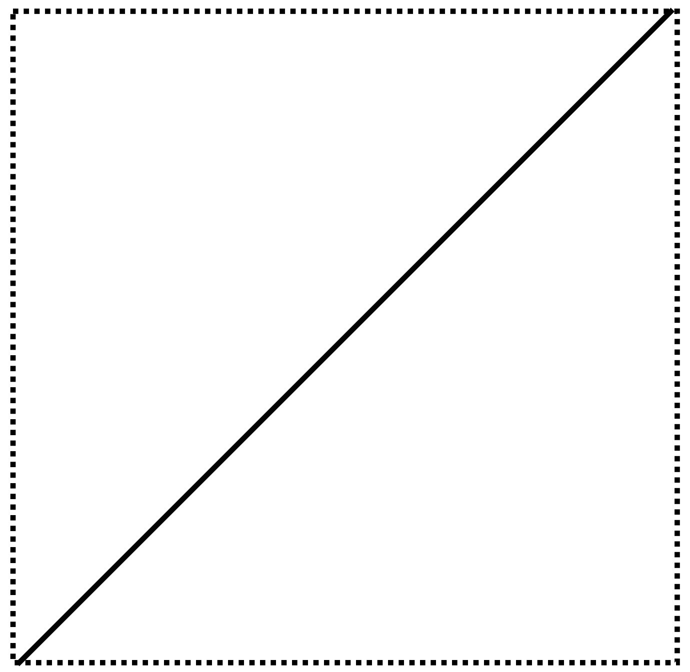
Rendering time: 27m 38s

Same scene (But now rotated in world space, so walls are less axis aligned)

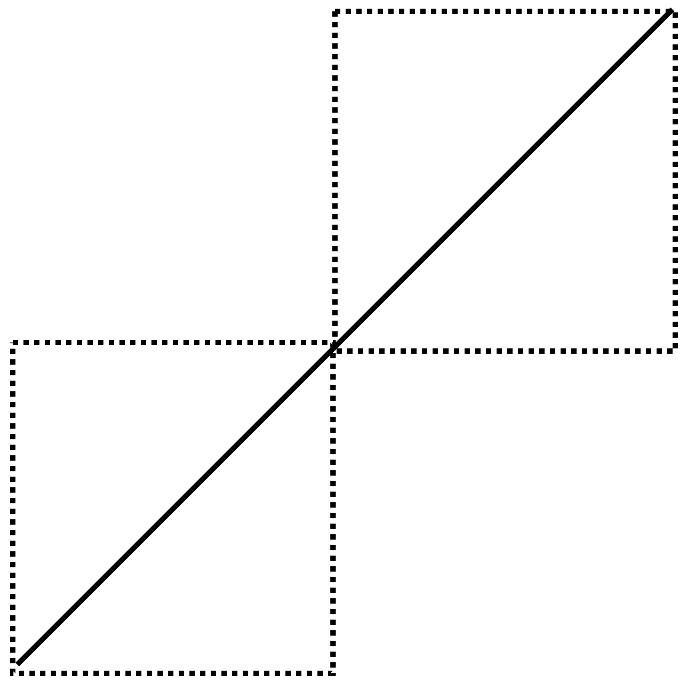


Rendering time: 1h 55m 45s

Axis-alignment and performance



Rotated wall and its bounding box



Work-around: refine bounding boxes

Note: this introduces back the idea of partitioning space! (Recall octree, KD-tree)

Immense interest in real time ray tracing

Image credit: Unreal Engine 4

Ray tracing dynamic scenes

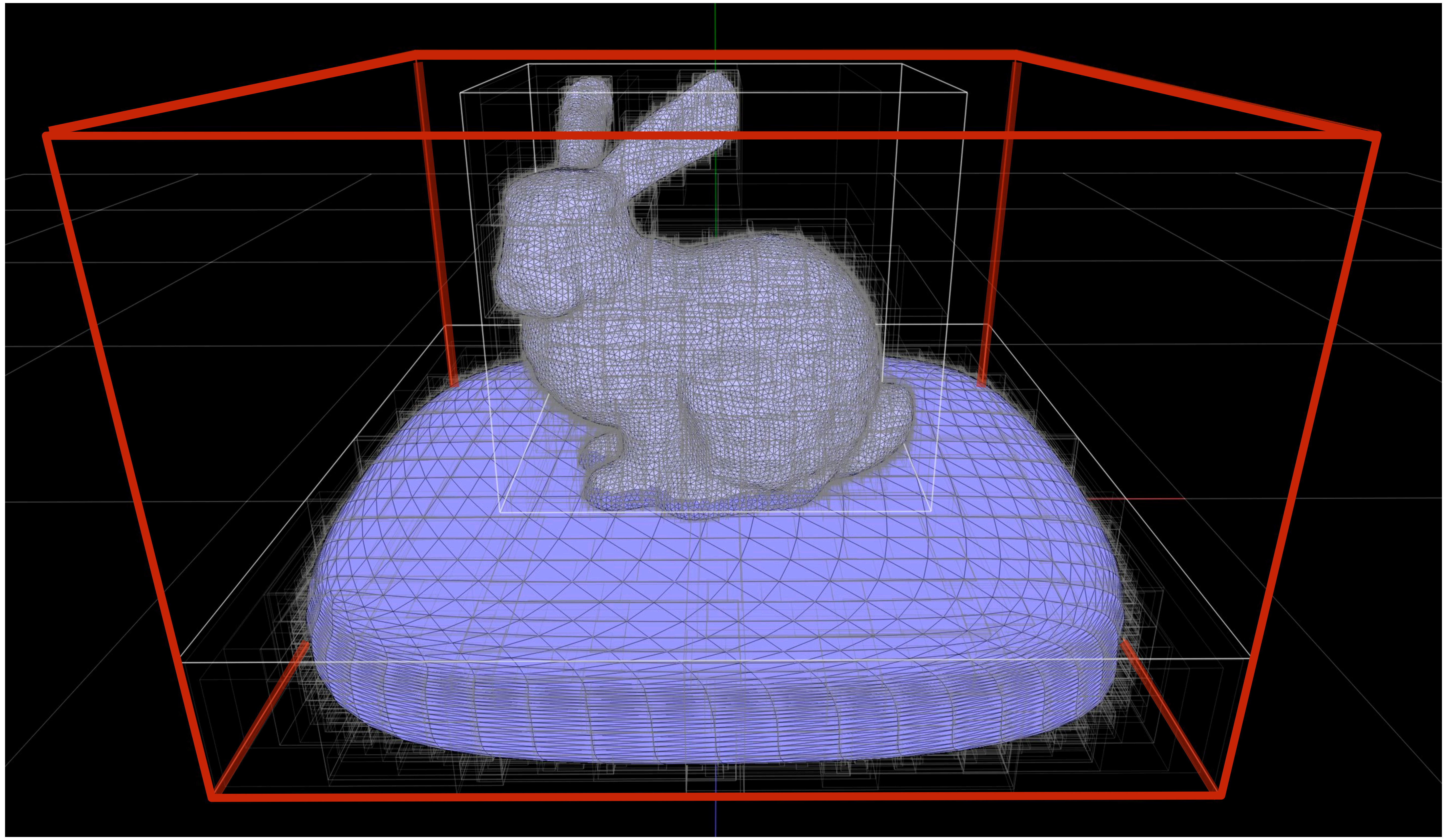
- Scenes have millions of triangles, many objects are in motion

- For real time applications, can allow a few ms / frame for BVH build
  - e.g. @10M tris, 60fps, need to build BVH at 600M tris / second

➡ Hierarchy construction efficiency really matters!

A BVH itself is an intersectable primitive!

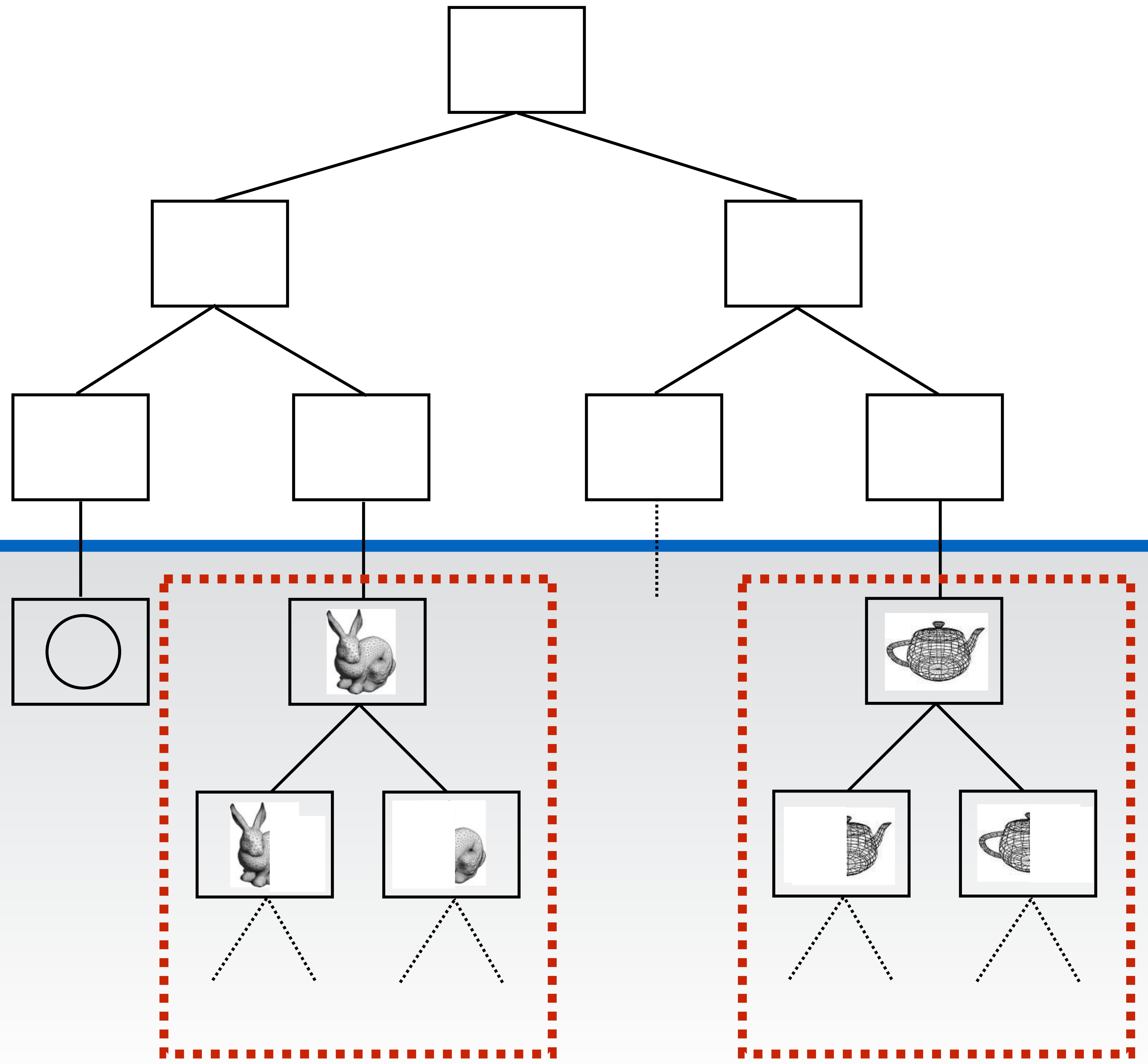
- It has a bounding box
- It supports ray-primitive intersection
- So it can be used as a primitive in another BVH



Two-level acceleration structures

2-level hierarchy

"Top-level" acceleration structure
(build every frame)



"Bottom-level" acceleration structures are primitives in top-level BVH

(Built once upon scene load)

Hierarchical BVH build

At scene load…

Build one BVH for each object

Each frame…

Build top-level BVH of BVH's based on current object positions.

(Scene may contain millions of triangles, but only hundreds of objects.)

Image credit: Brennan Shacklett

Acknowledgements

- Thanks to Keenan Crane, Ren Ng, and Matt Pharr for presentation resources