Lecture 9:

Accelerating Geometric Queries

Interactive Computer Graphics

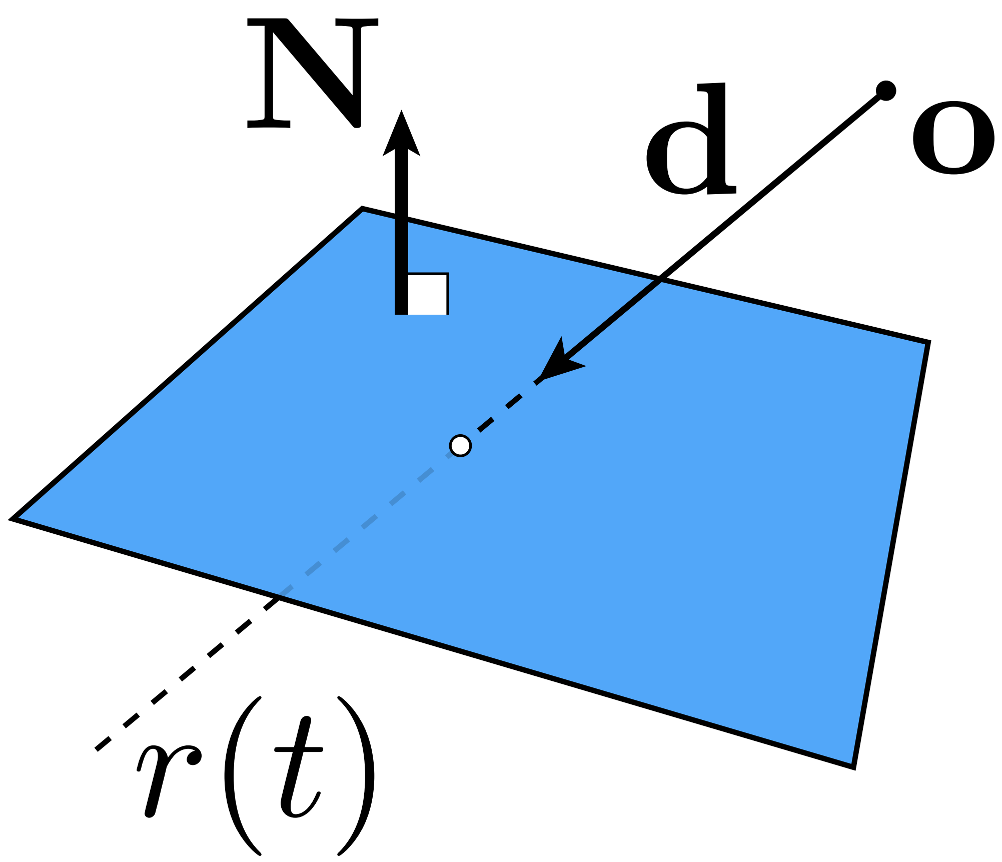
Stanford CS248, Winter 2020

Tunes

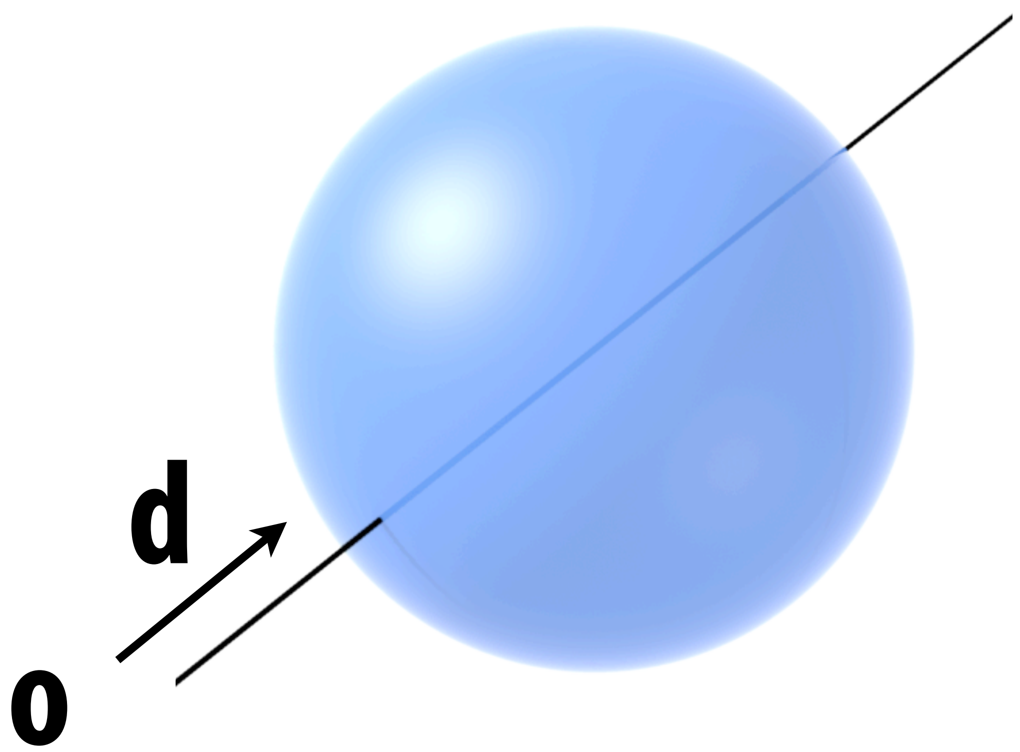
M.I.A.
“Paper Planes”
(Kala)

“B.V.H. was taken by the graphics folks, so I went with M.I.A.”
- Mathangi Arulpragasam

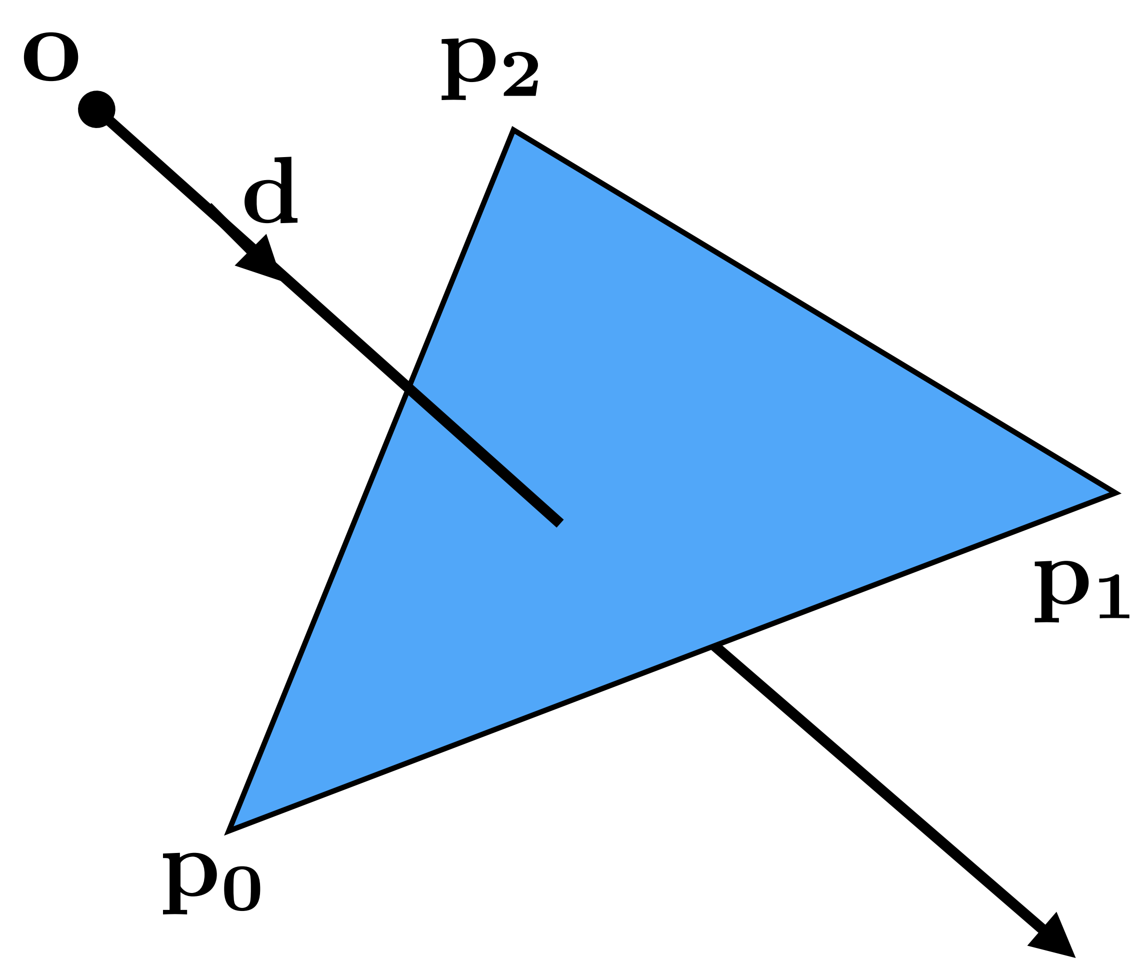
Last time: intersecting a ray with individual primitives



Ray-plane



Ray-sphere



Ray-triangle

Applying what you learned

- **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

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

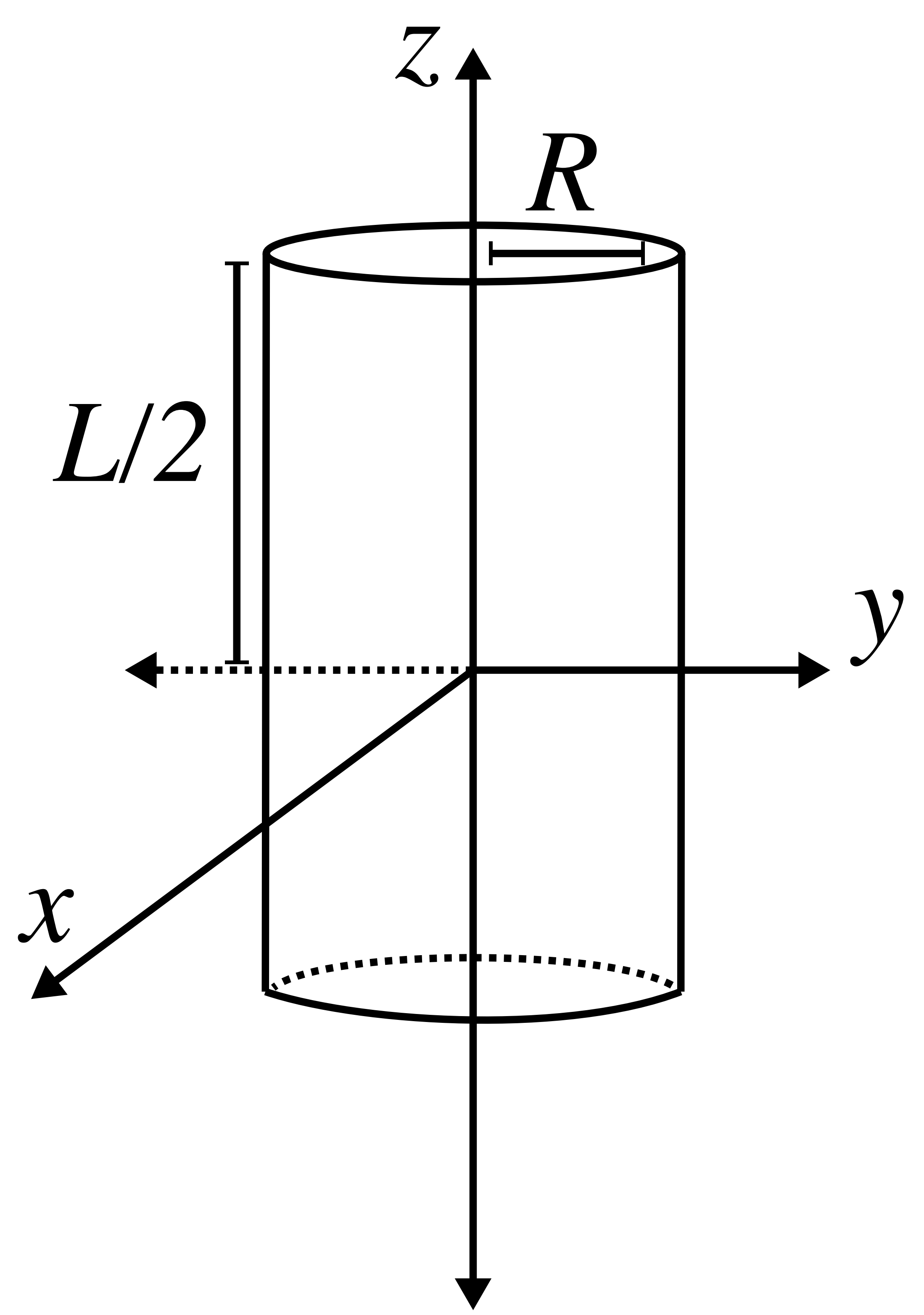
From last class you know:

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?

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

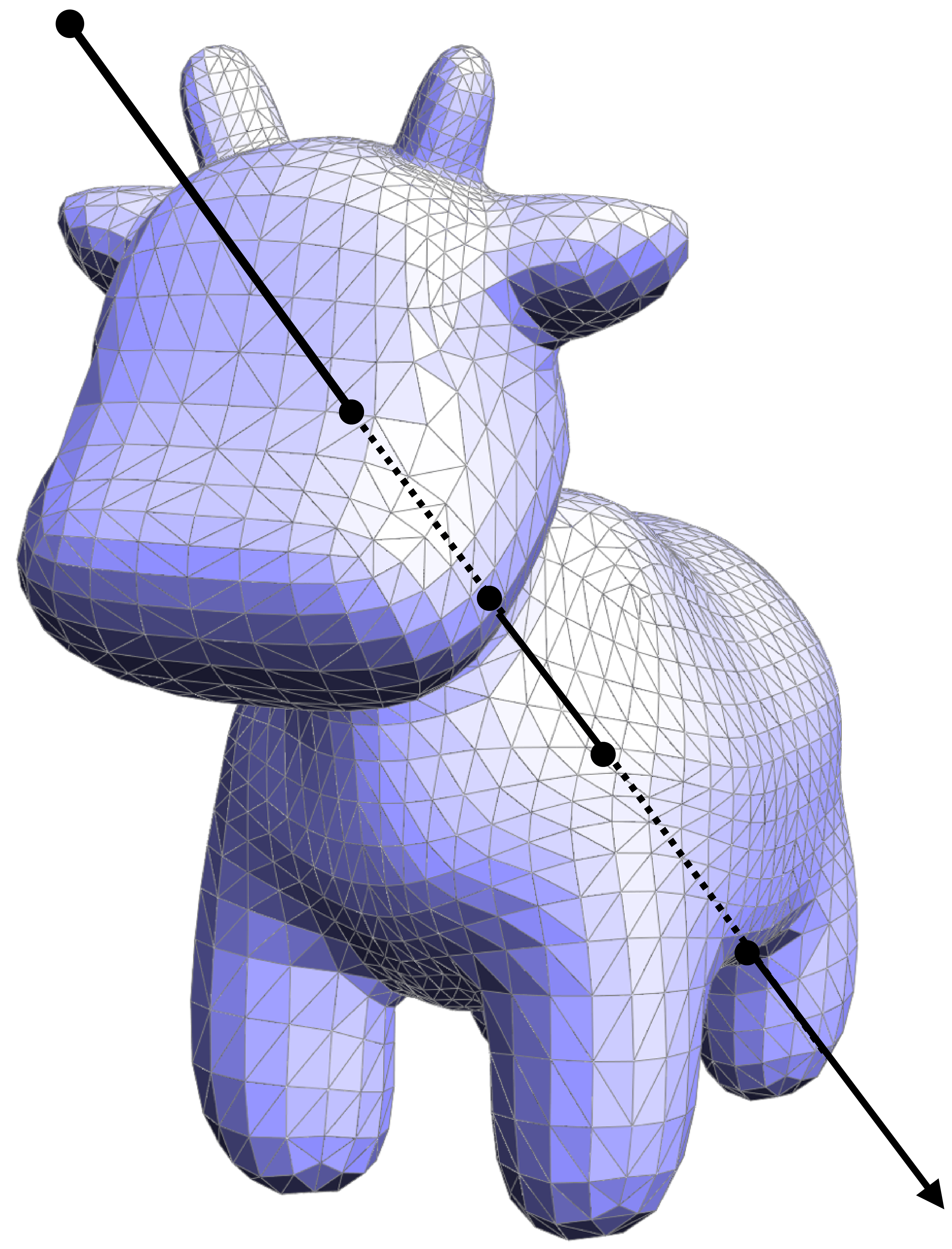
"Find the first primitive the ray hits"

```
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?

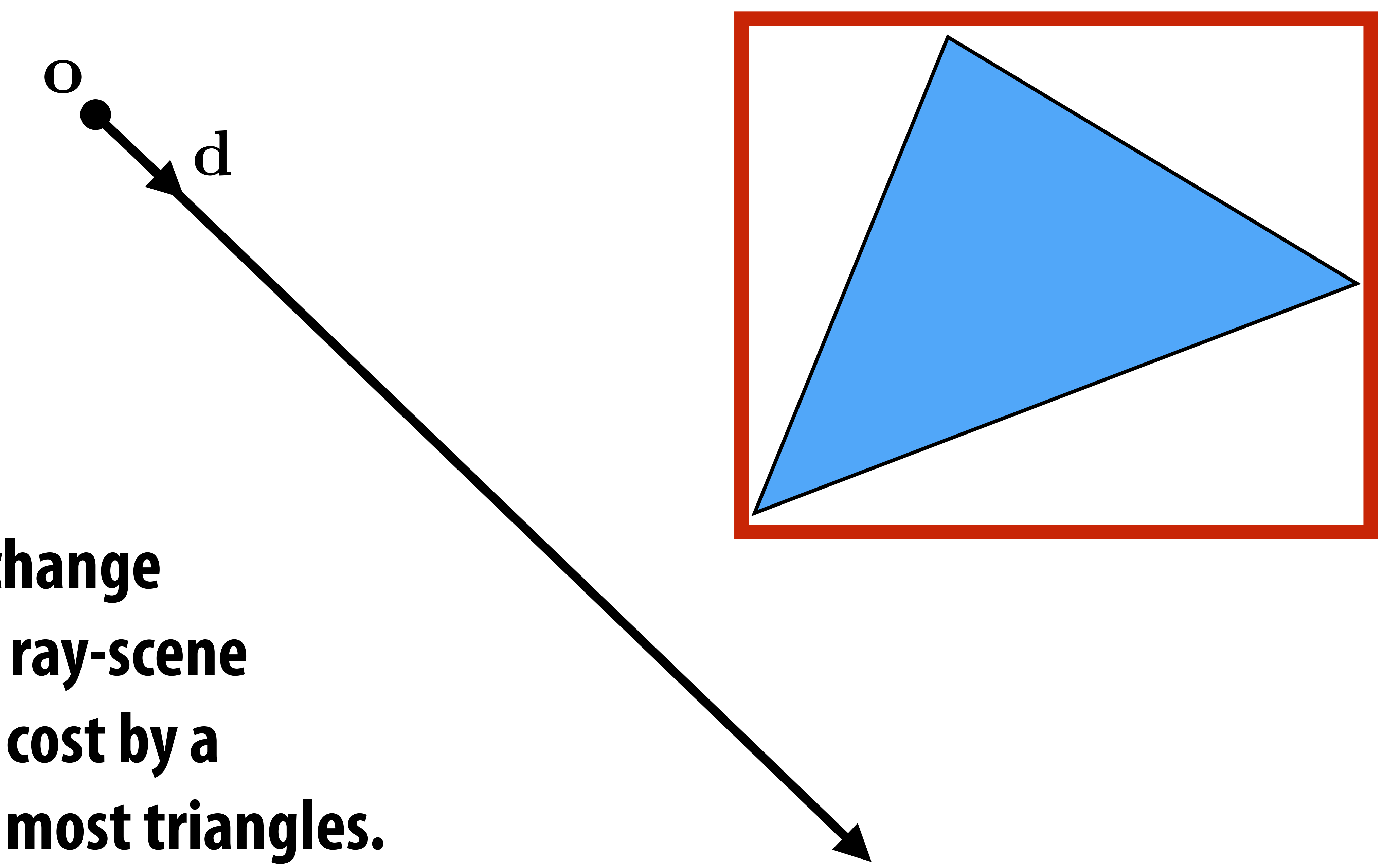
(Assume p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r*)



One simple idea

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

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



Note: early out does not change asymptotic complexity of ray-scene intersection. But 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?

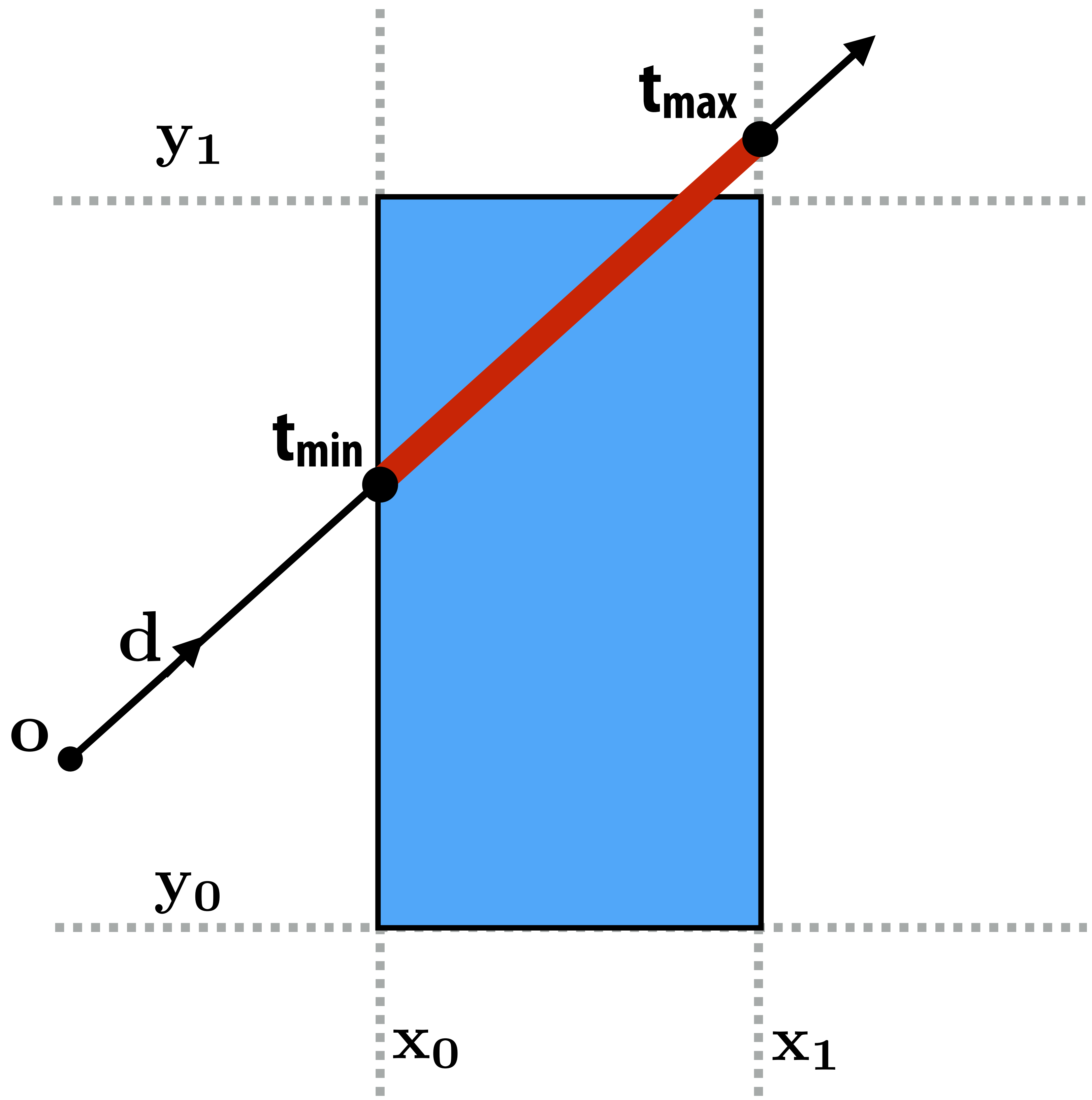


Figure shows intersections with x=x_0 and x=x_1 planes.

Find intersection of ray with all planes of box:

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

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

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

$$c = x_0$$

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

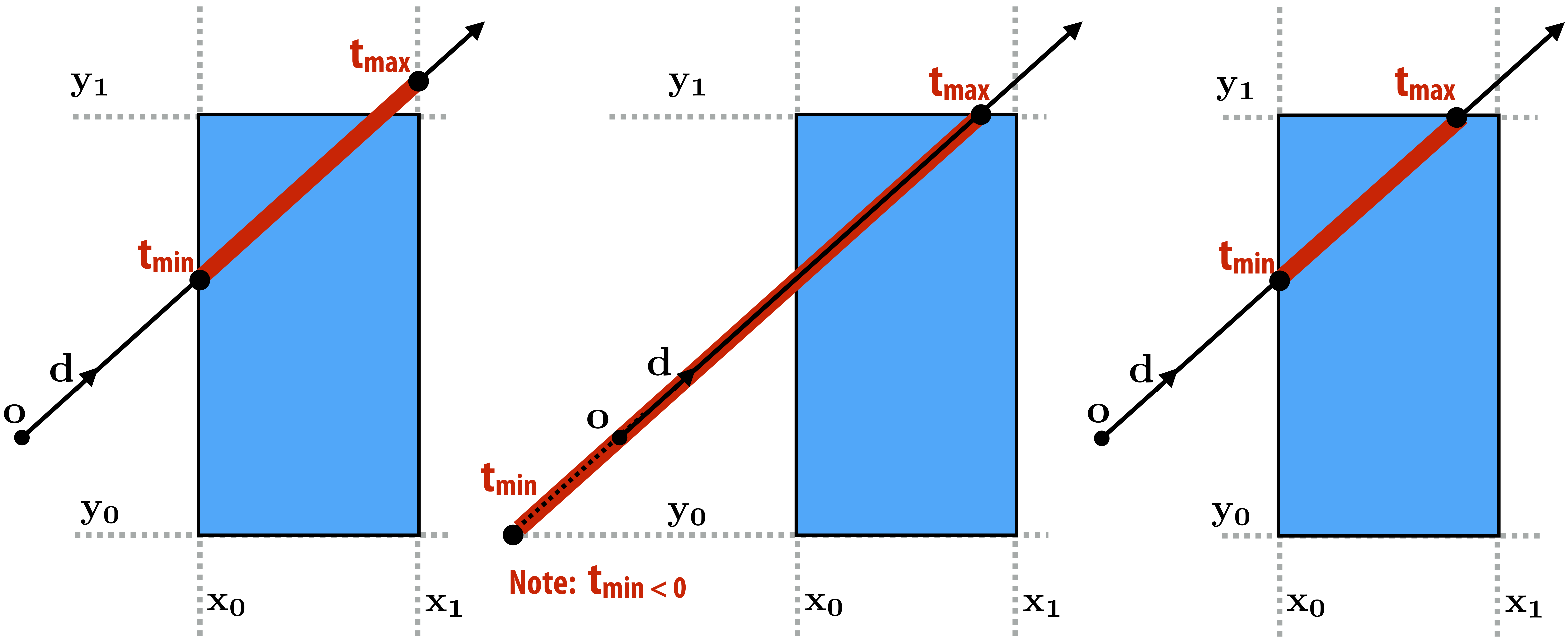
Performance note: it is possible to precompute box independent terms, so computing *t* is cheap

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

So… $t = ax + b$

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

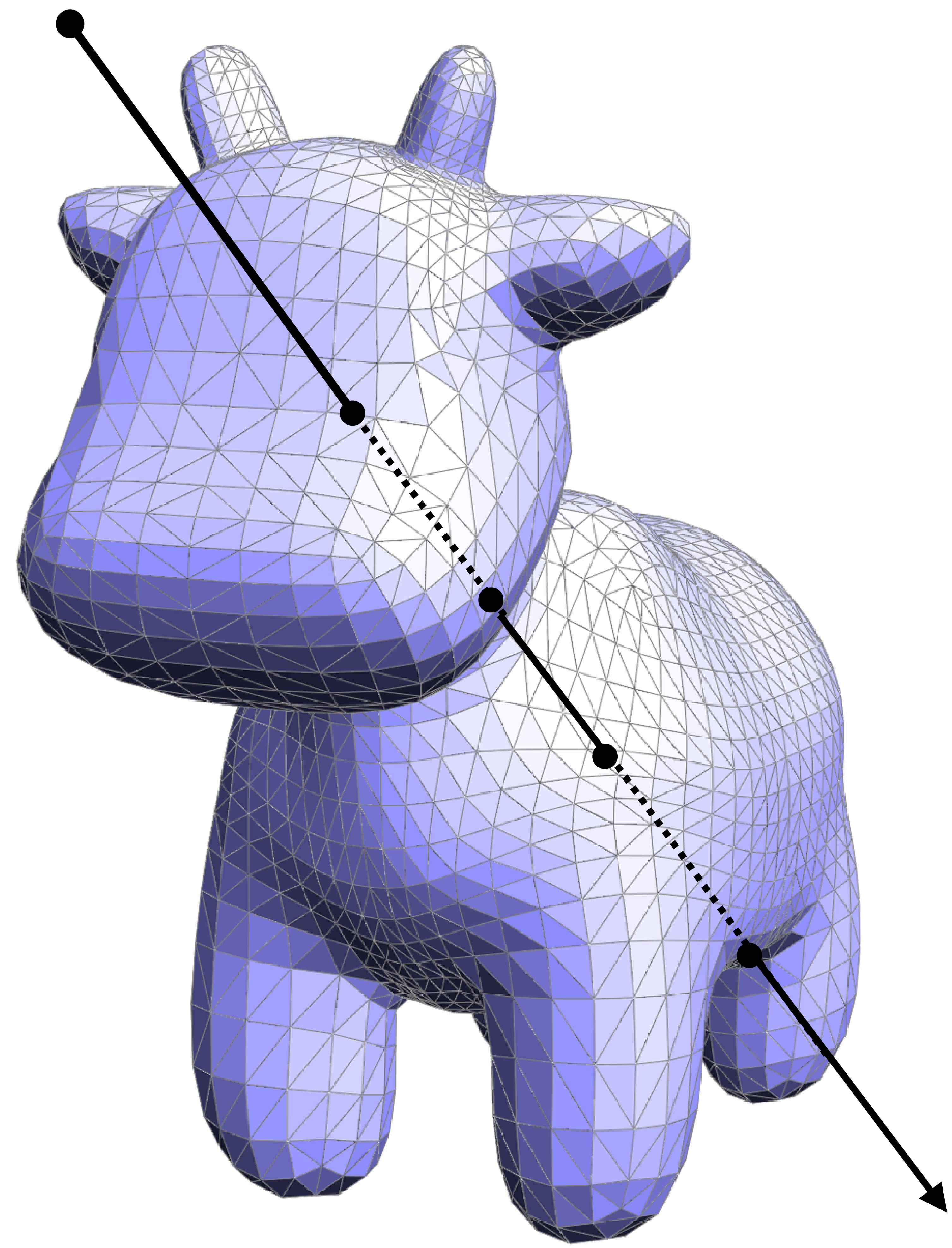
How do we know when the ray misses the box?

Ray-scene intersection 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*)

Review: recall optimization in simple rasterizer

Sample = 2D point
Coverage: 2D point in triangle tests
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]
```

```
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)

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 any intersection with scene triangles*

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:

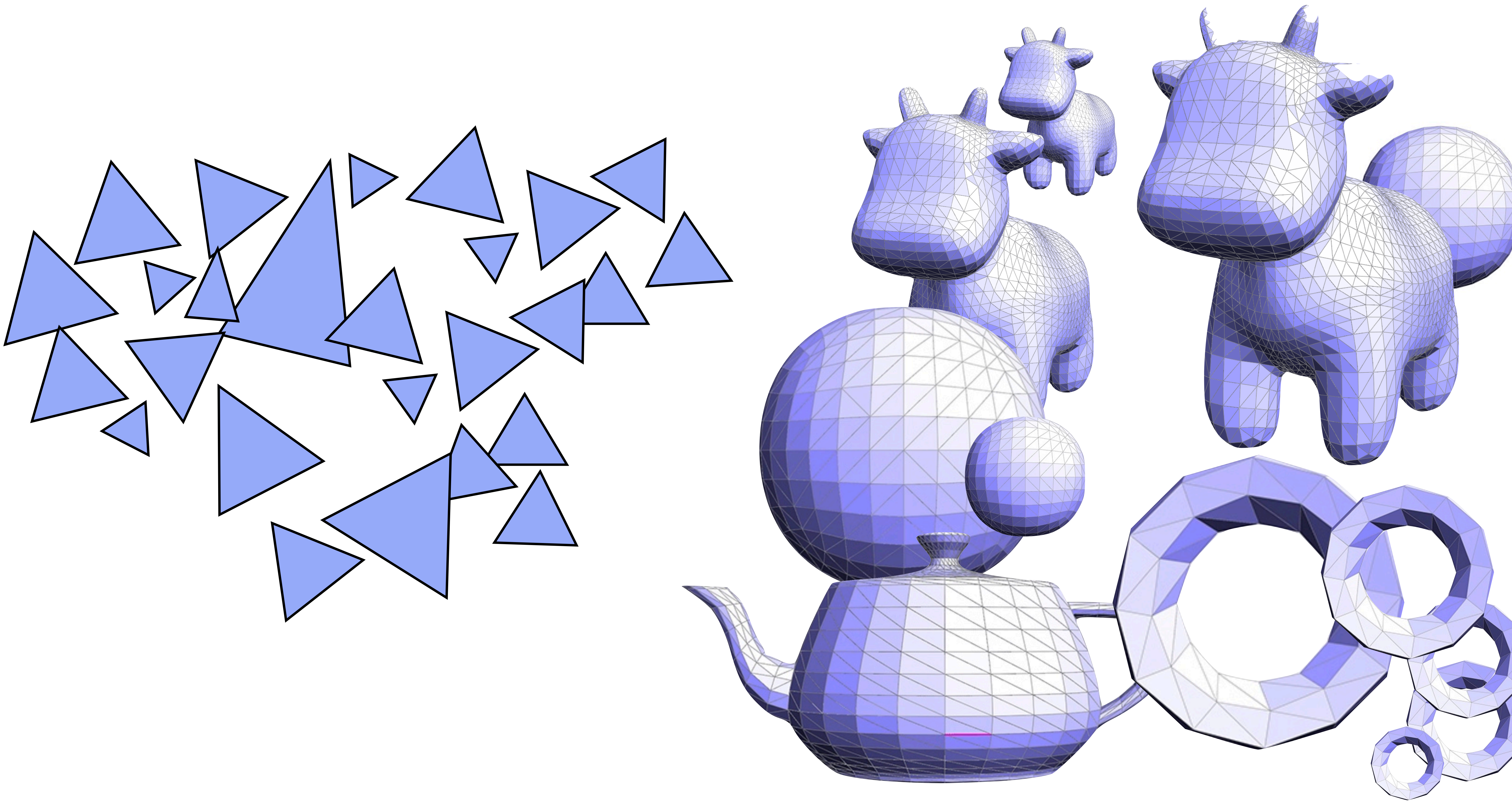
1 2 6 8 10 11 20 25 30 64 80 100 111 123 200 950

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

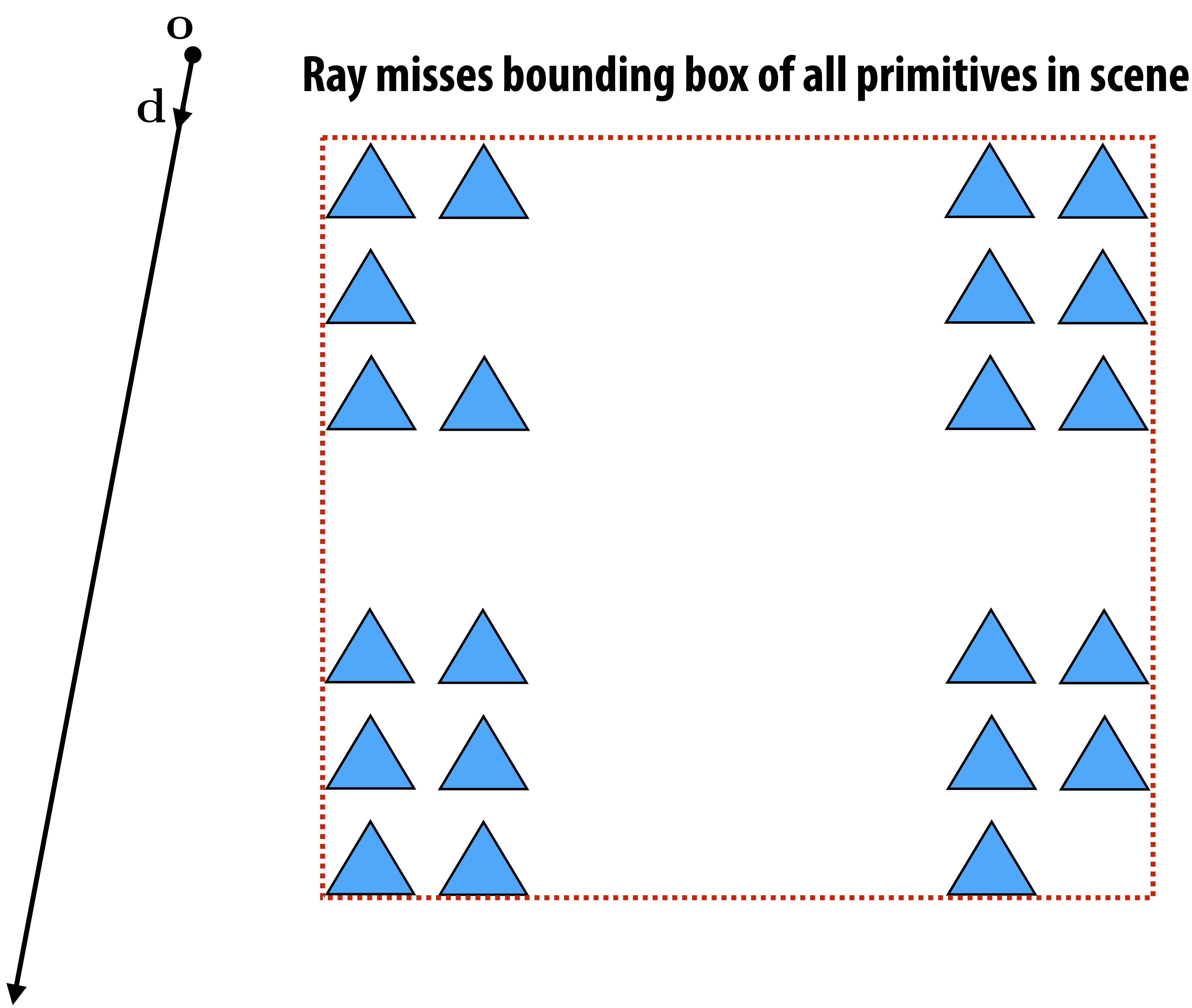
Cost for just ONE query: O(n log n) *worse* than before! :-(

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

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



Simple case



Cost (misses box):

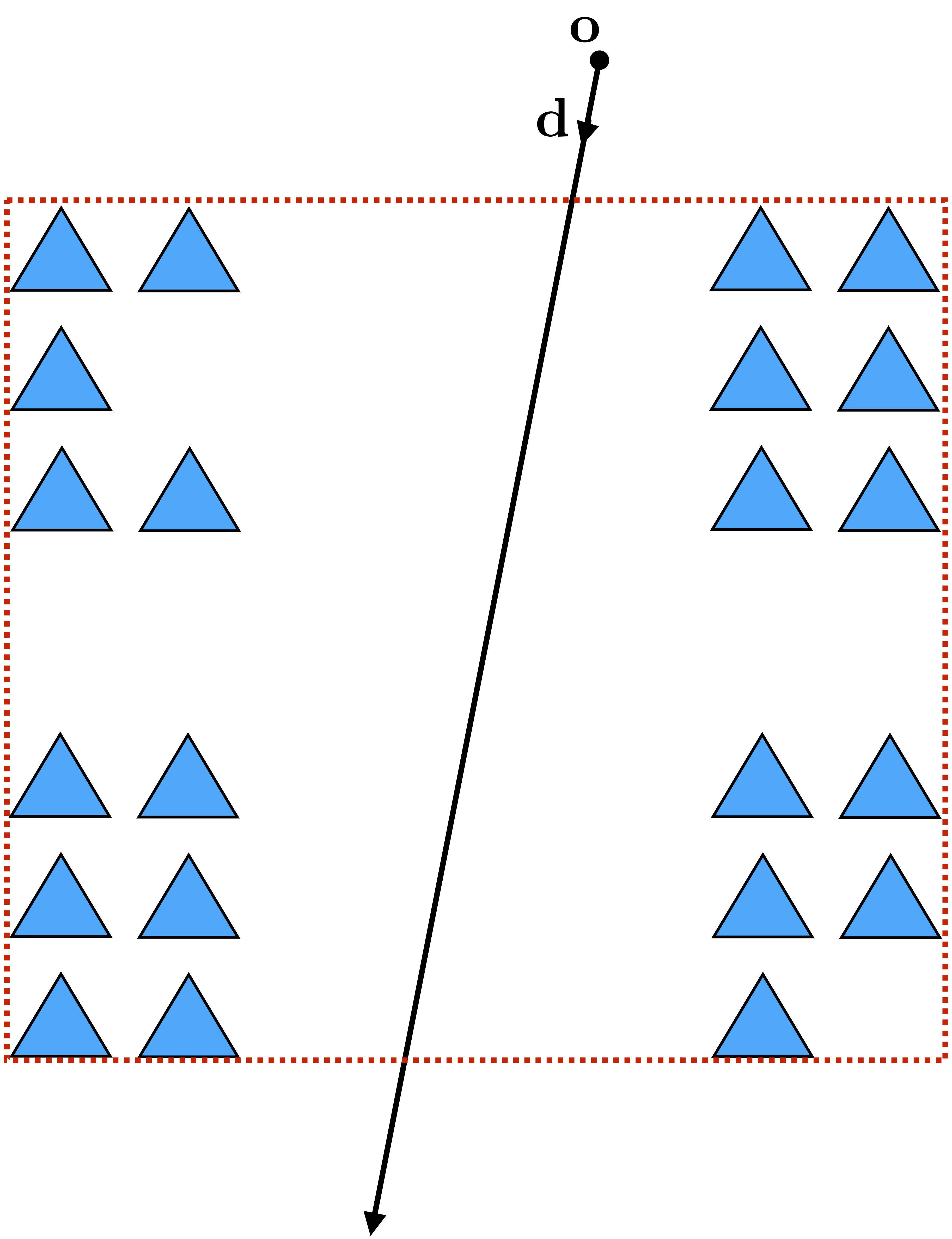
preprocessing: O(n)

ray-box test: O(1)

amortized cost*: O(1)

*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)

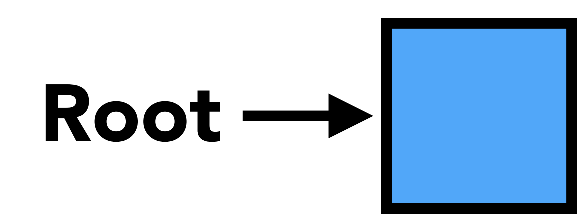
Still no better than naïve algorithm (test all triangles)!

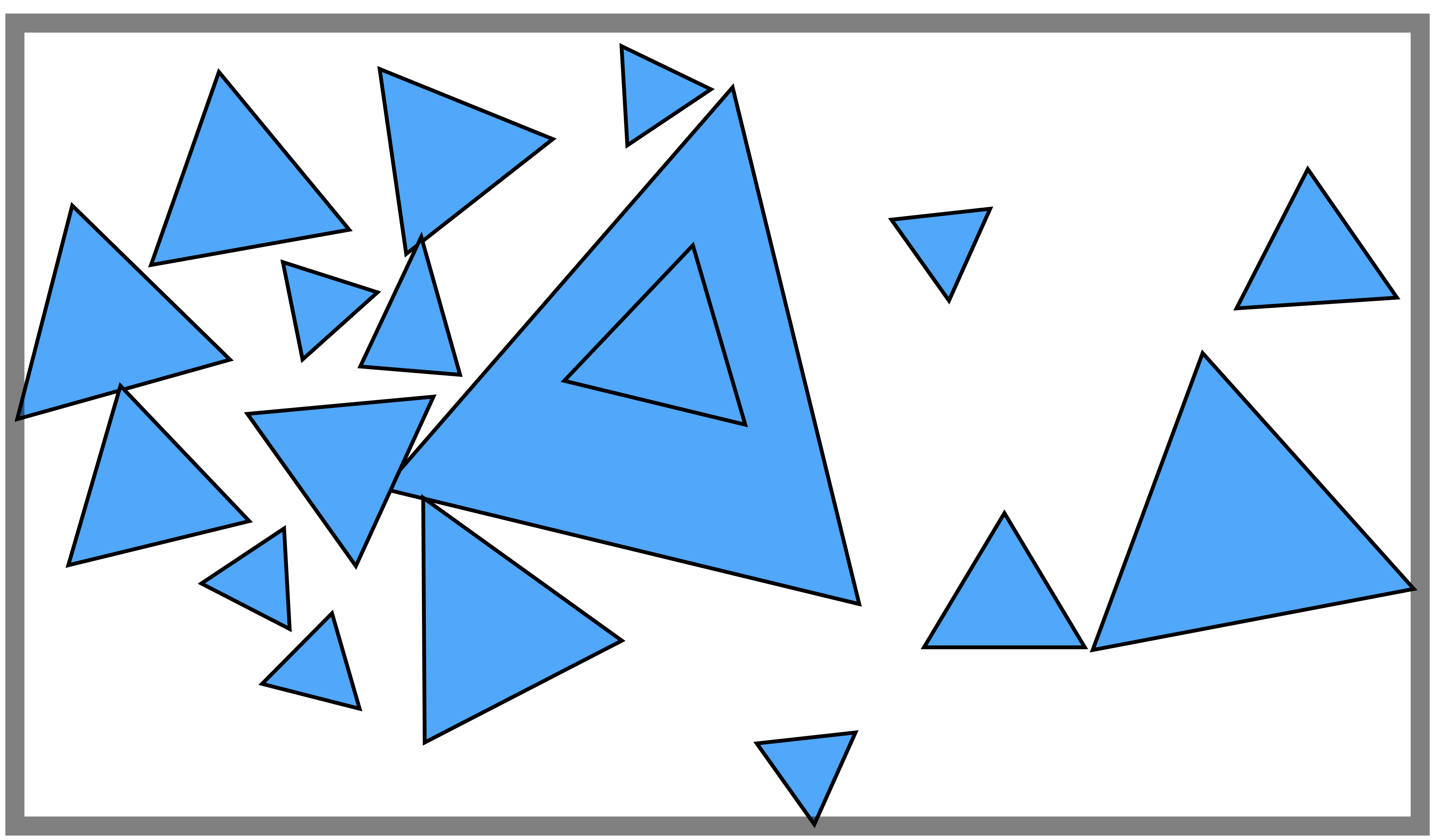
*amortized over *many* ray-scene intersection tests

Q: How can we do better?

A: Apply this strategy hierarchically

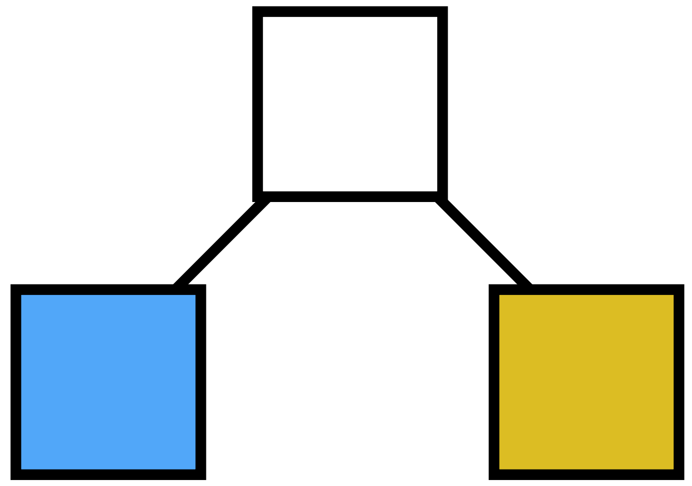
Bounding volume hierarchy (BVH)

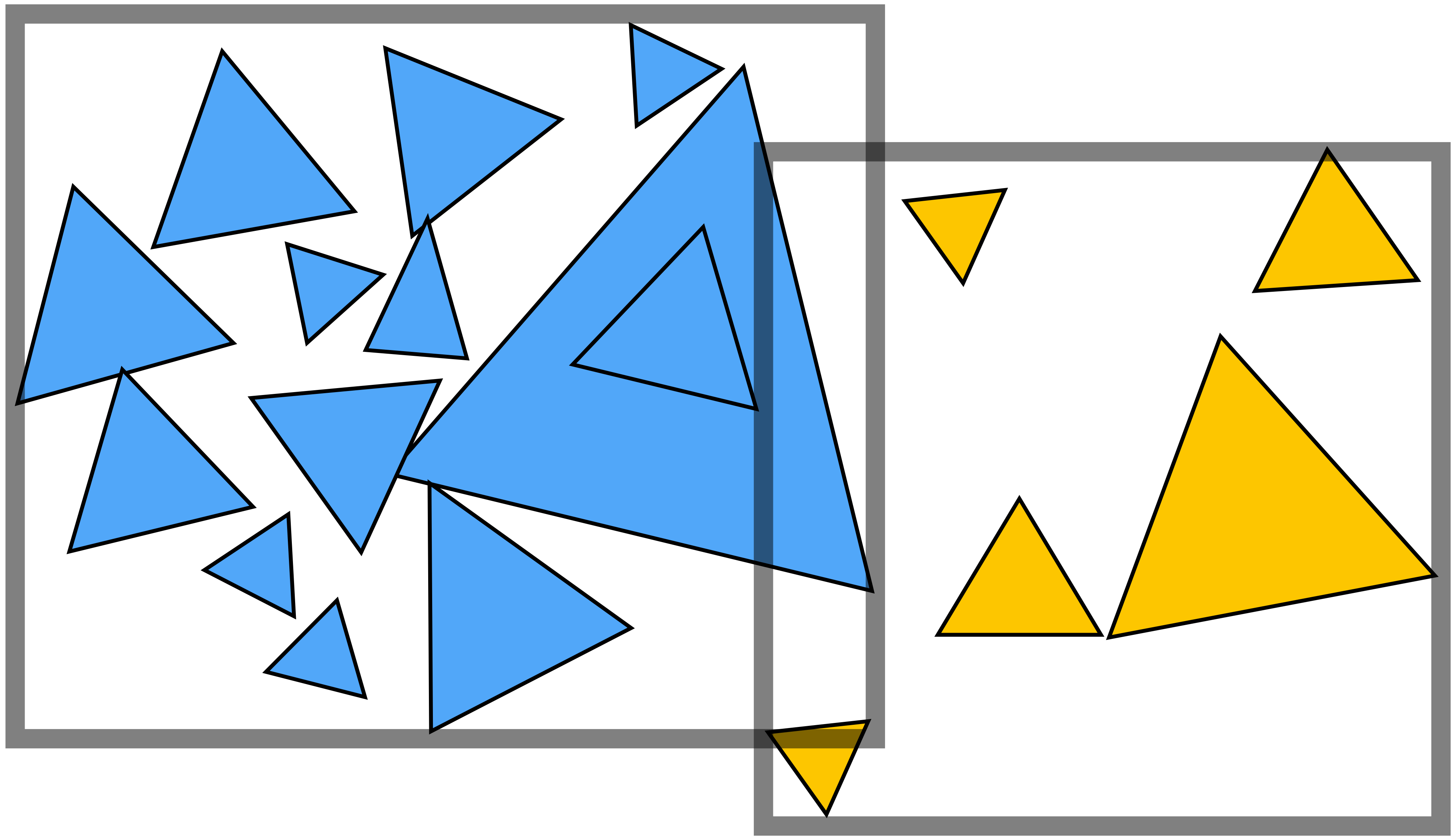




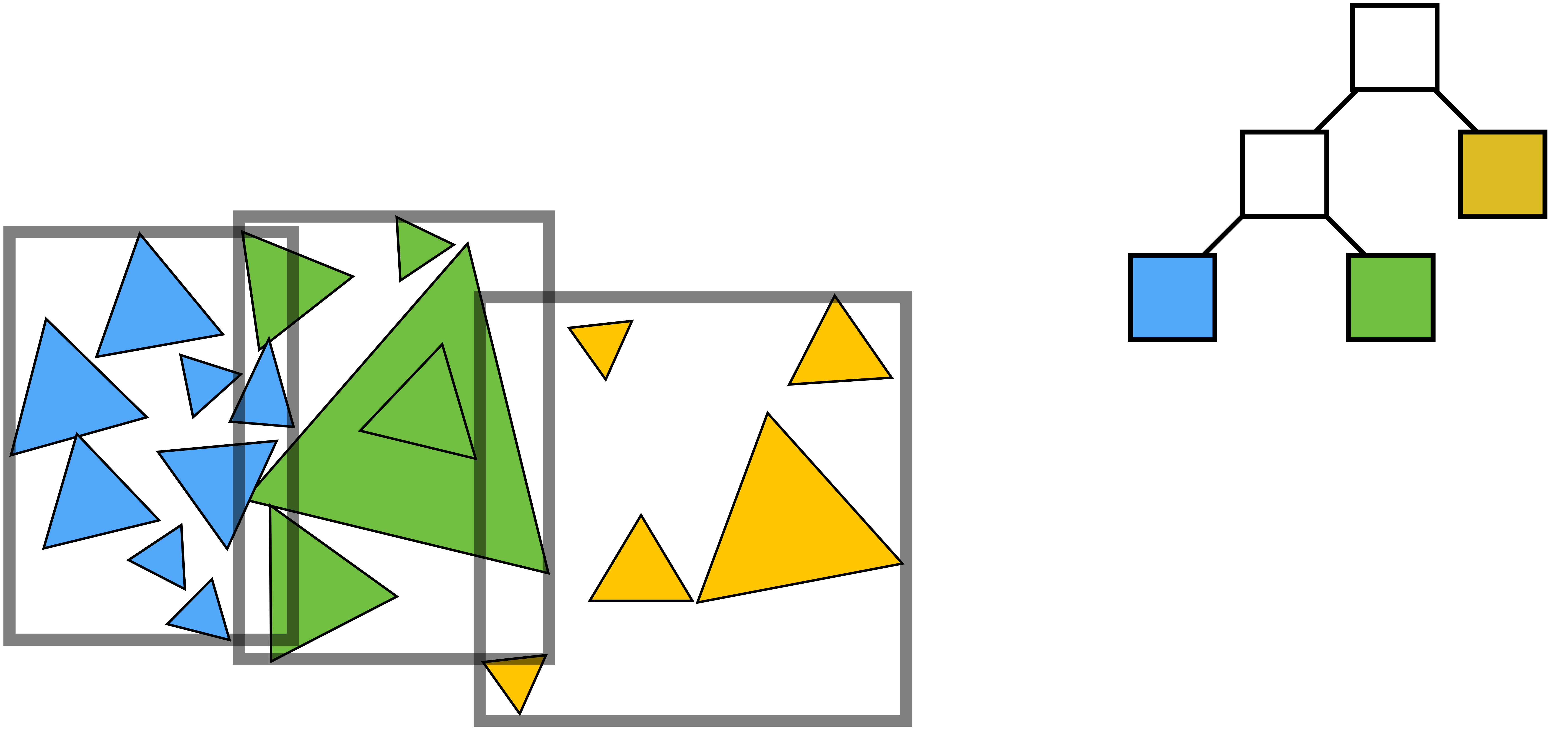
Bounding volume hierarchy (BVH)

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



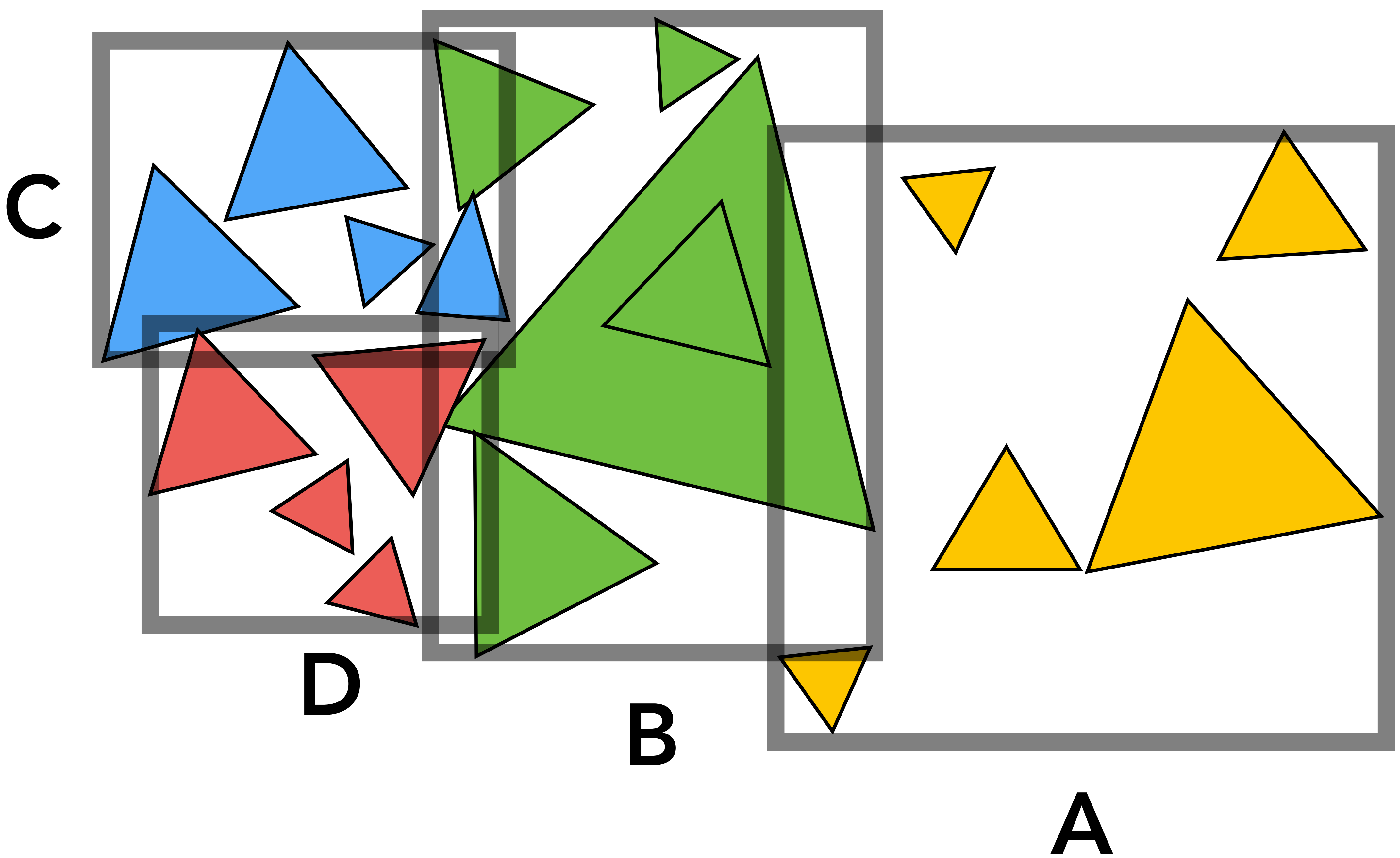


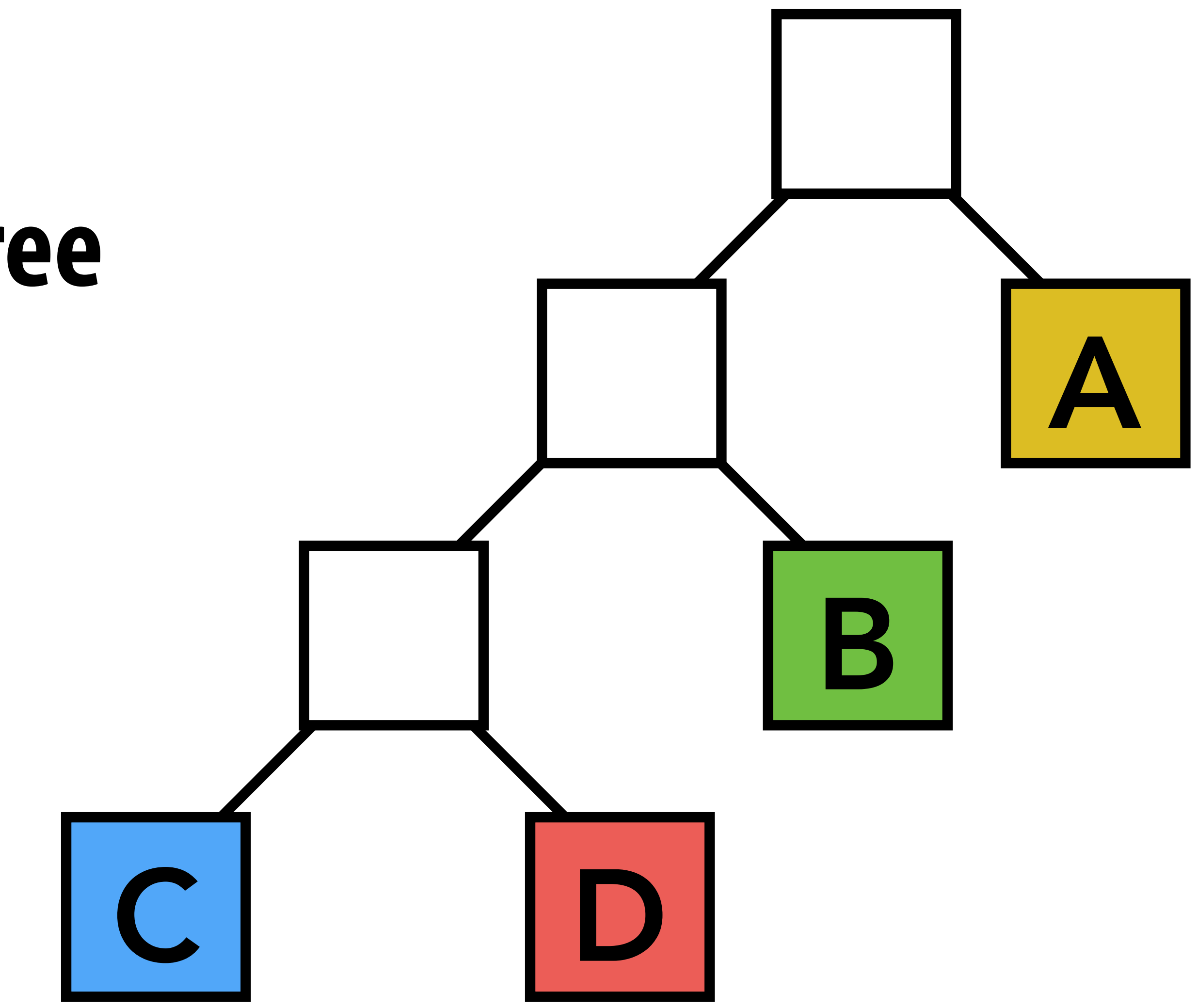
Bounding volume hierarchy (BVH)



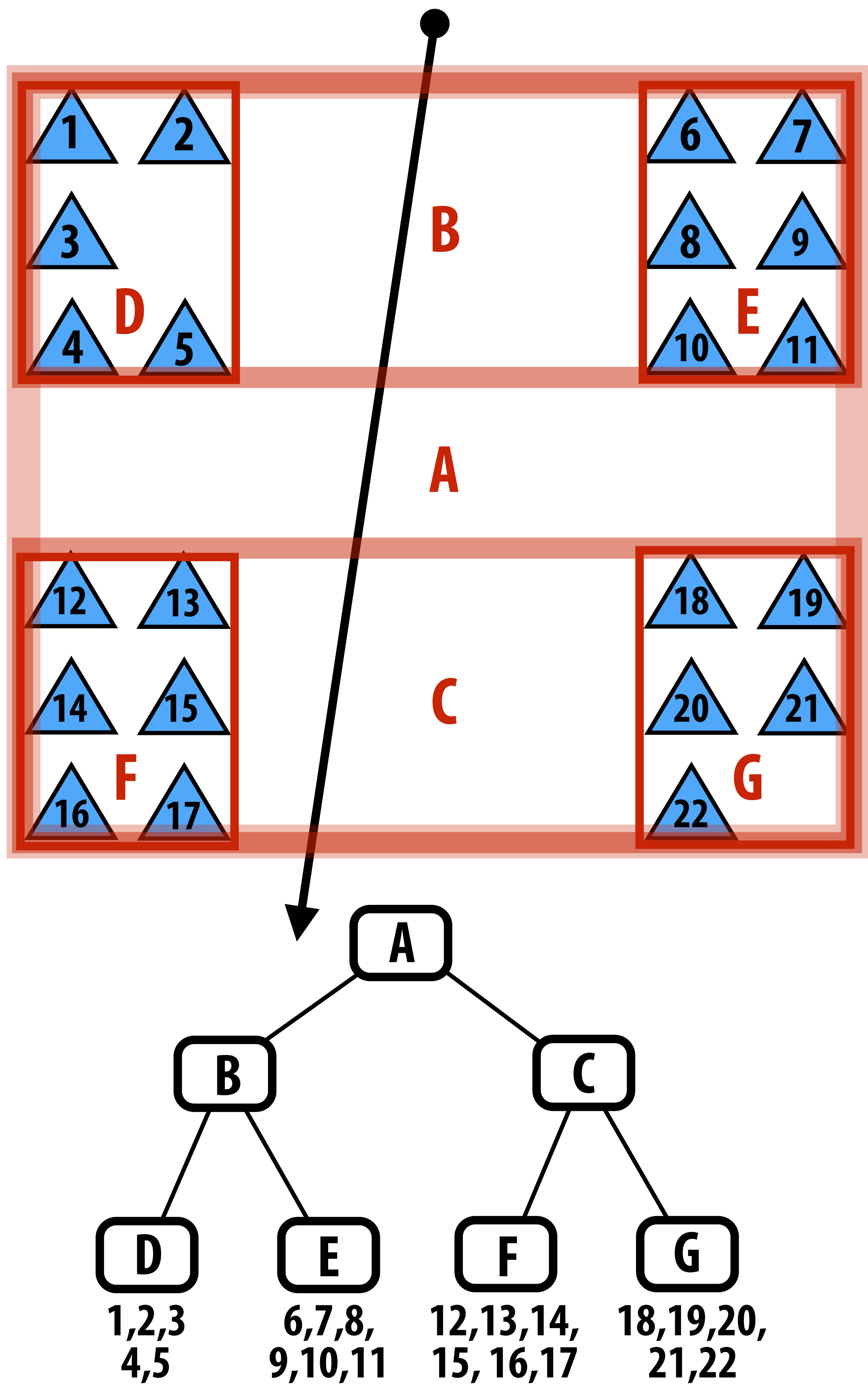
Bounding volume hierarchy (BVH)

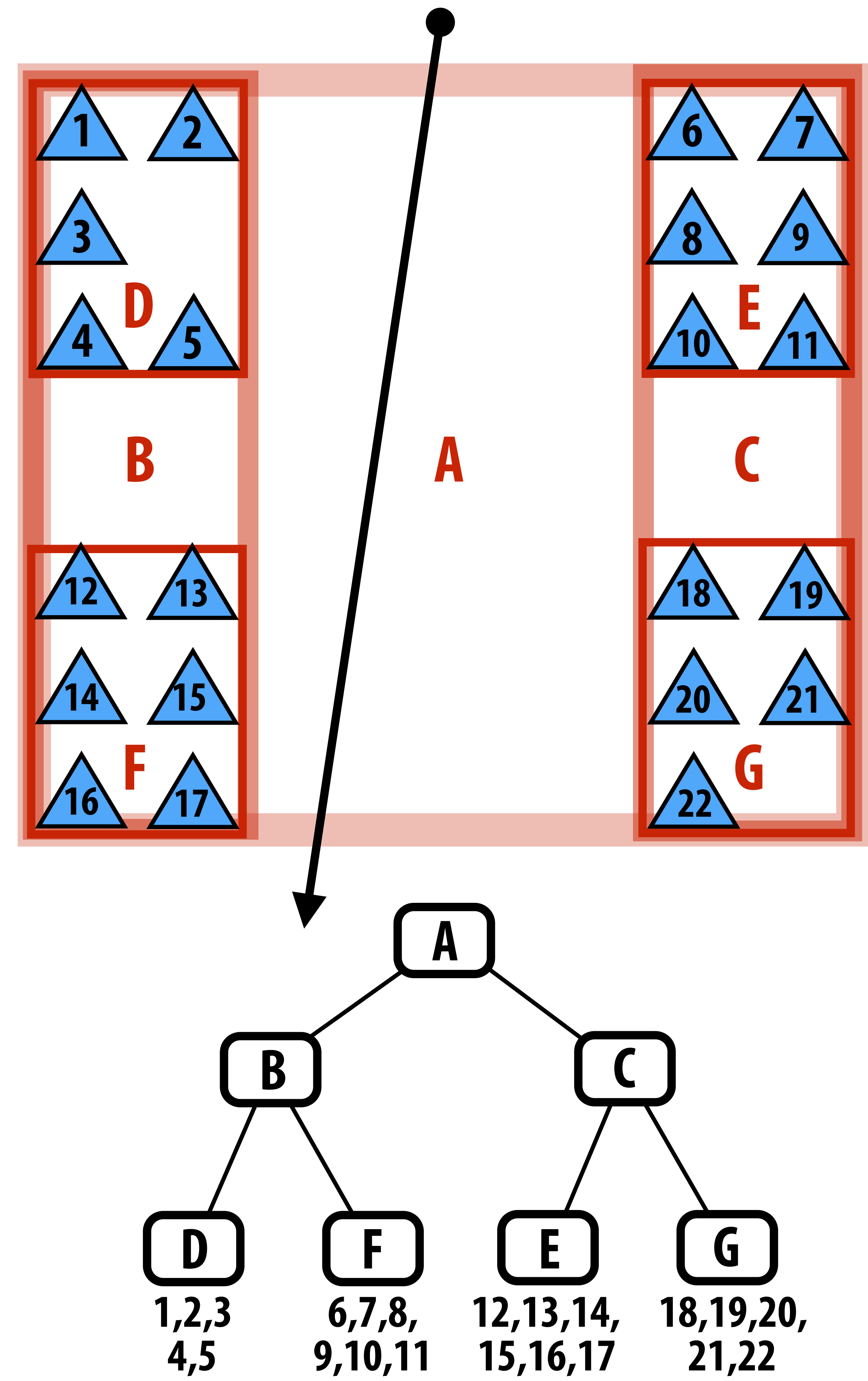
- Leaf nodes:
  - Contain *small* list of primitives
- Interior nodes:
  - Proxy for a *large* subset of primitives
  - Stores bounding box for all primitives in subtree





Bounding volume hierarchy (BVH)



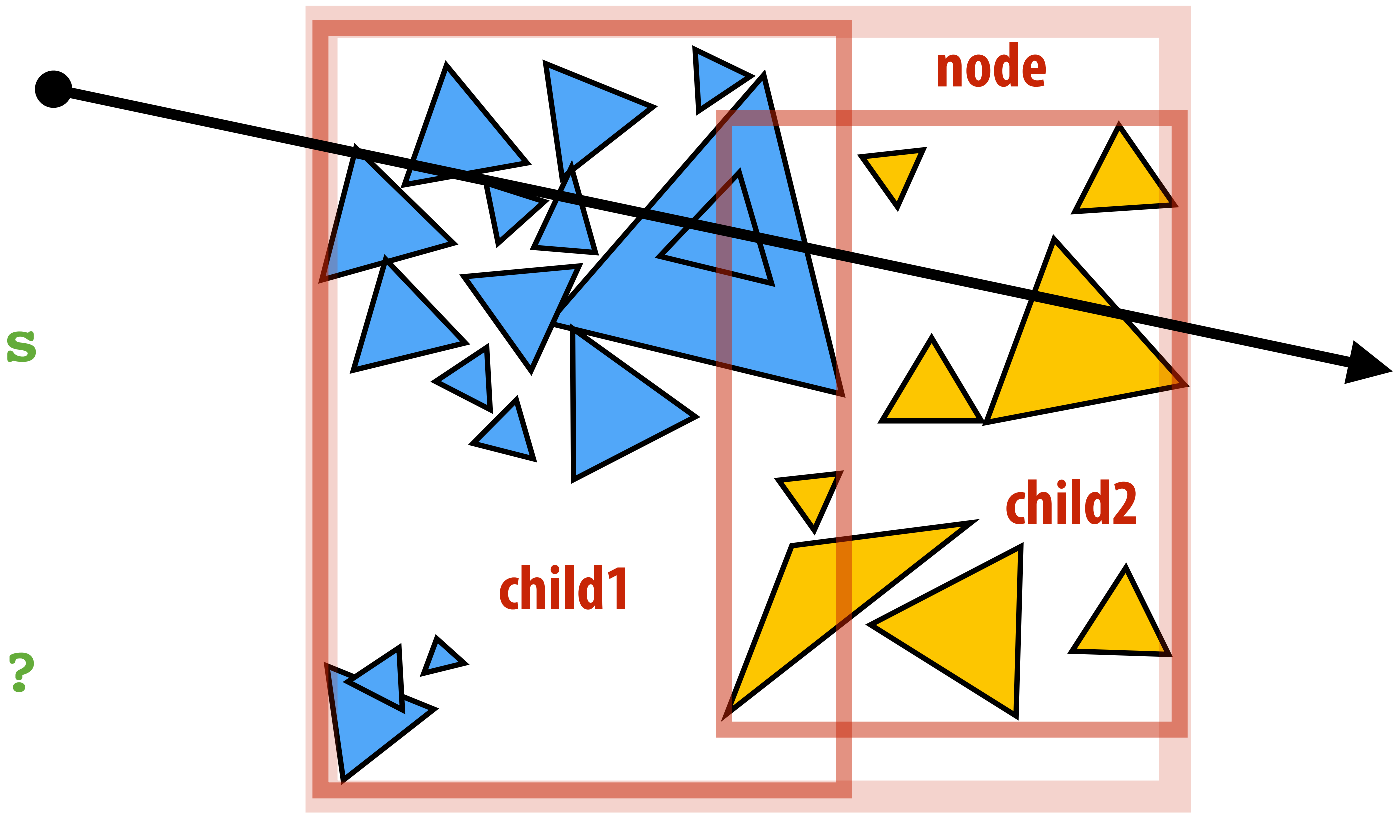


Left: two different BVH organizations of the same scene containing 22 primitives.

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);
}}
```

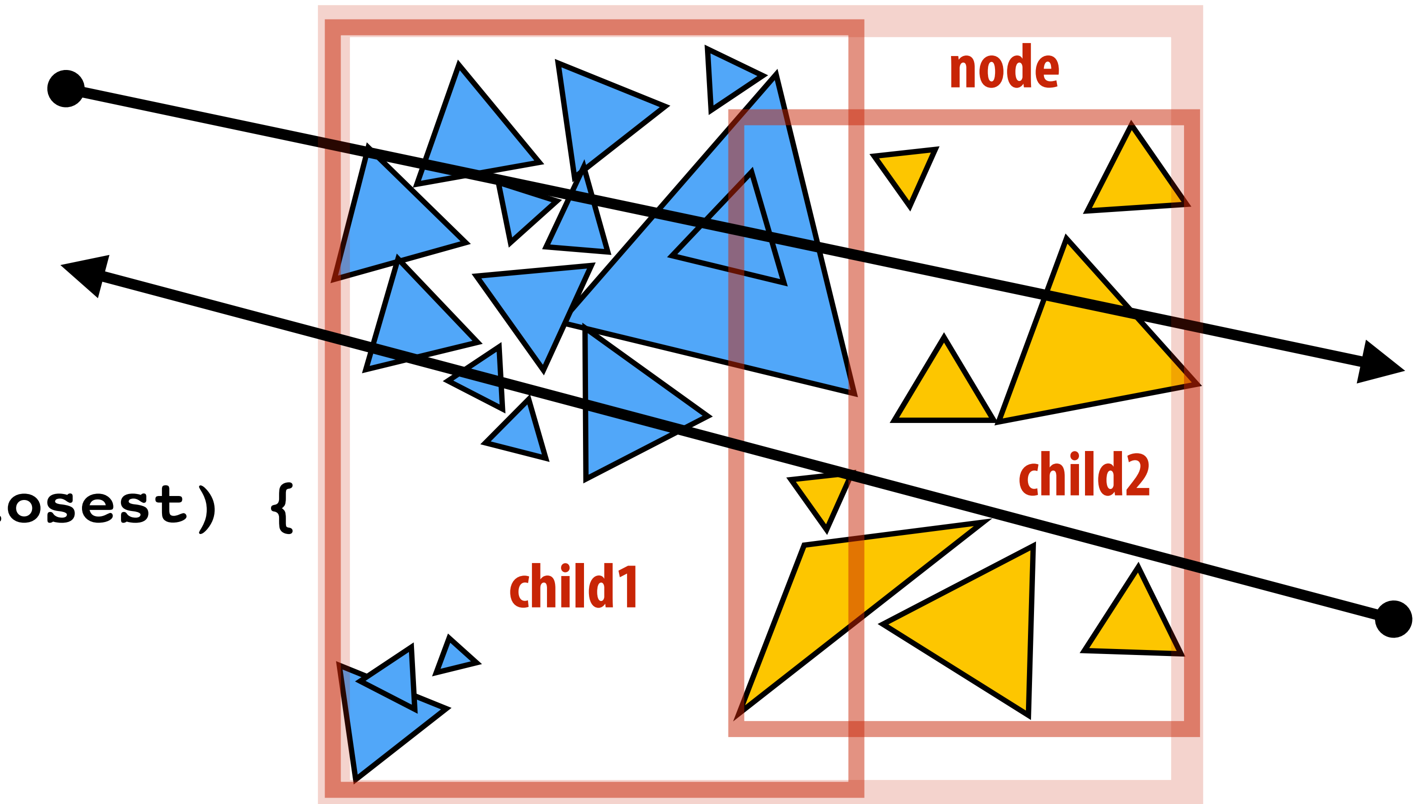


Can this occur if ray hits the box?

(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 on node 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 = 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); // why might we still need to do this?
   }
}
```

“Front to back” traversal.

Traverse to closest child node first. Why?

Aside: another type of query: any hit

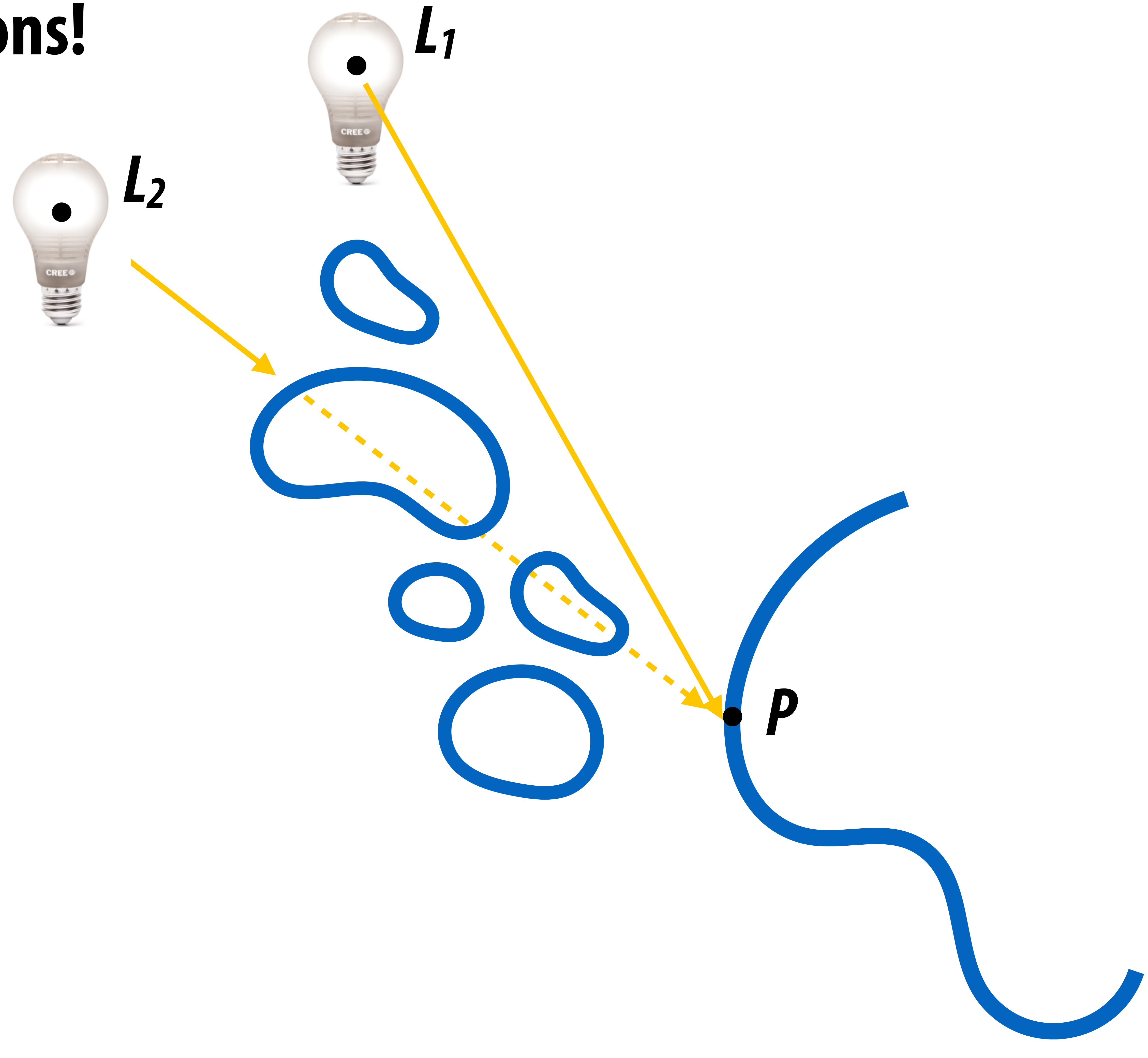
Sometimes it is useful to know if the ray hits ANY primitive in the scene at all (don't care about distance to first hit)

```
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!

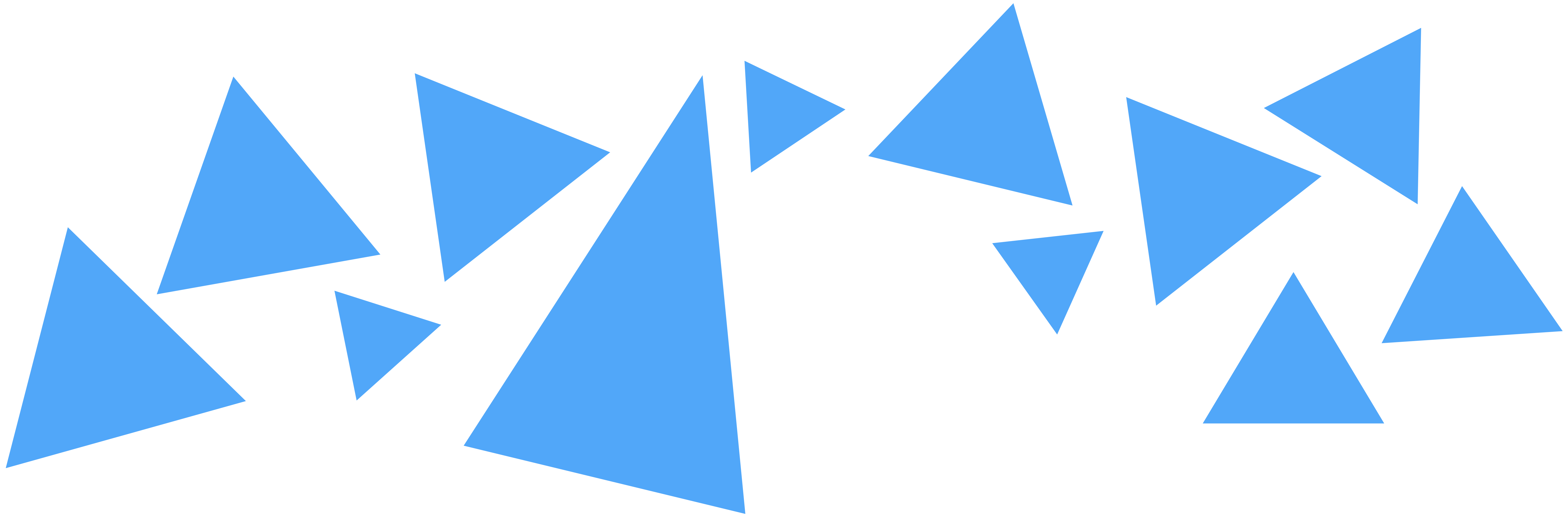


For a given set of primitives, there are many possible BVHs

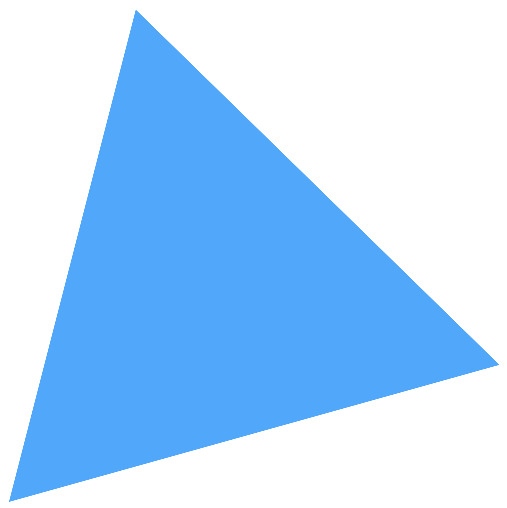
($\sim 2^N$ ways to partition N primitives into two groups)

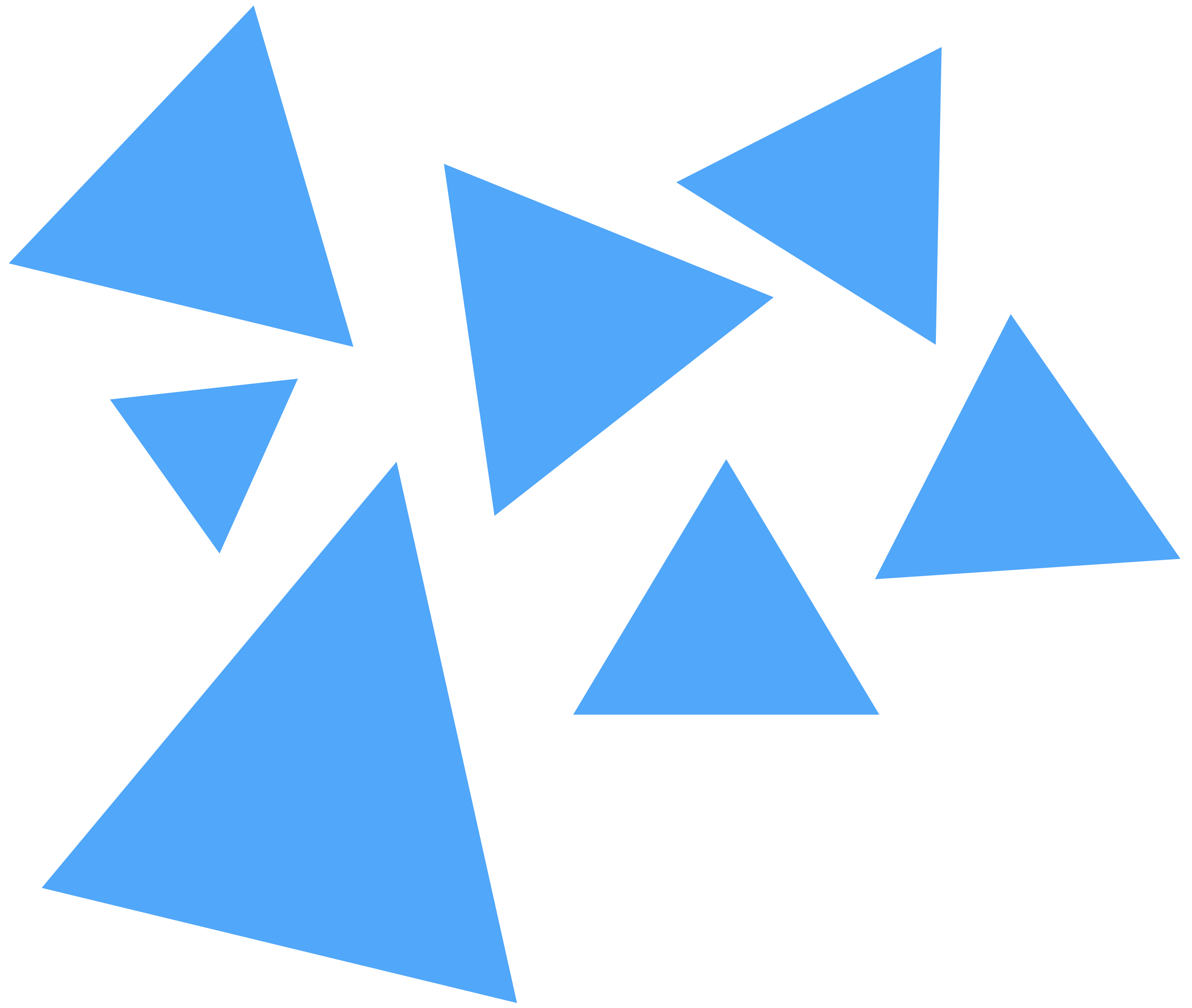
Q: How do we build a high-quality BVH?

How would you partition these triangles into two groups?

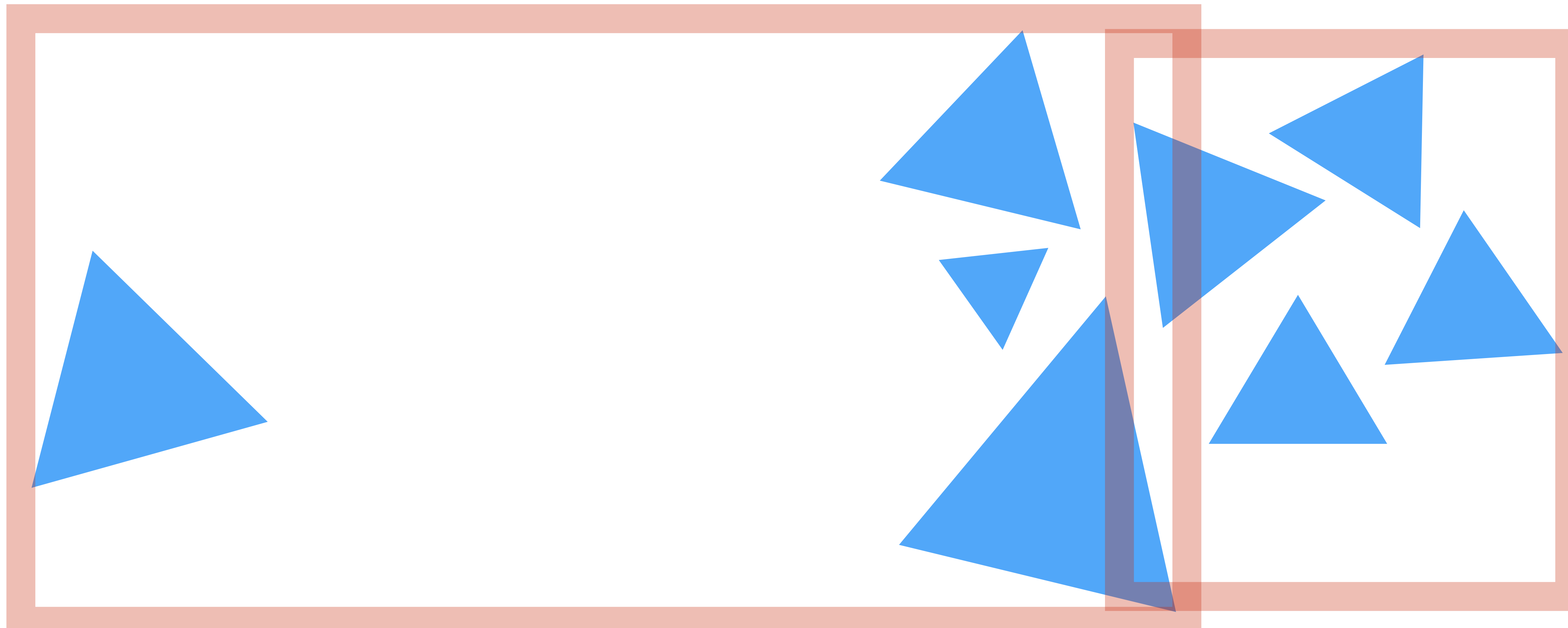


What about these?

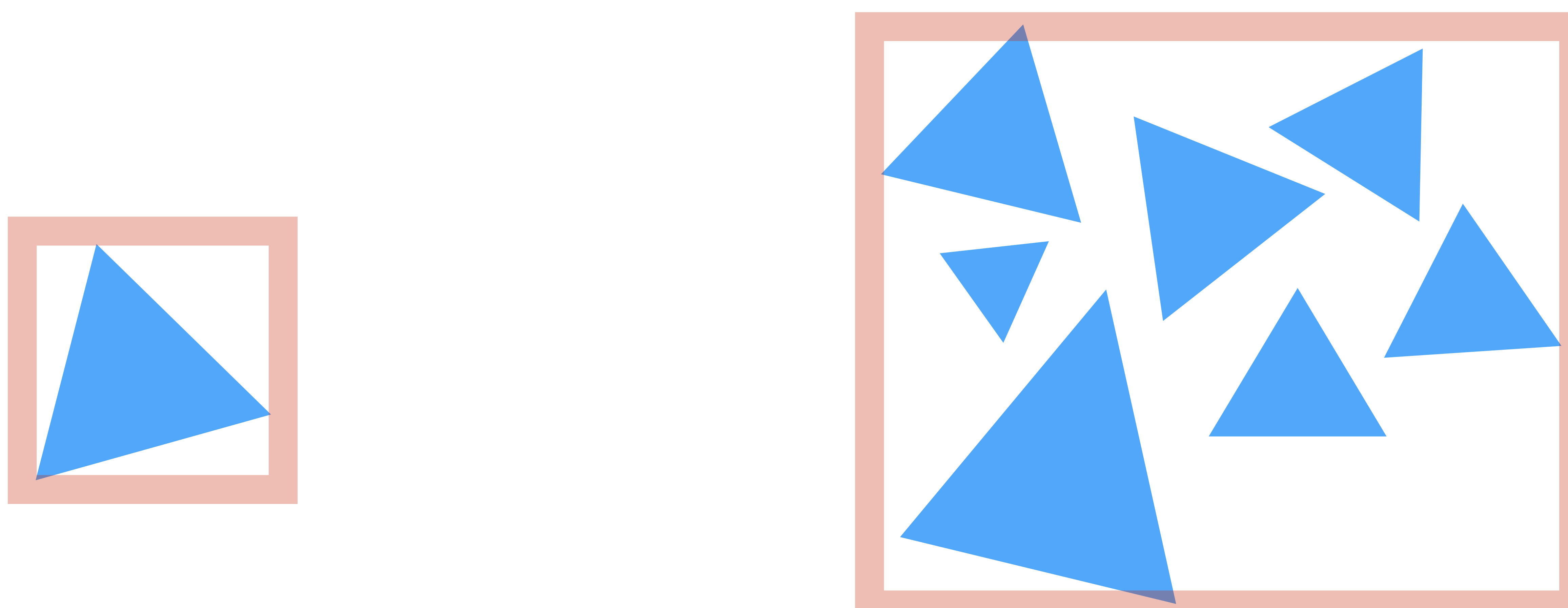




Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives



Better partition

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

What are we really trying to do?

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

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

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

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

$$= NC_{\mathrm{isect}}$$

(Common to assume all primitives have the same cost)

Cost of making a partition

The <u>expected cost</u> of ray-node intersection, given that the node's primitives are partitioned into child sets A and B is:

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

C_{trav} **is the cost of traversing an interior node (e.g., load data + bbox 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:

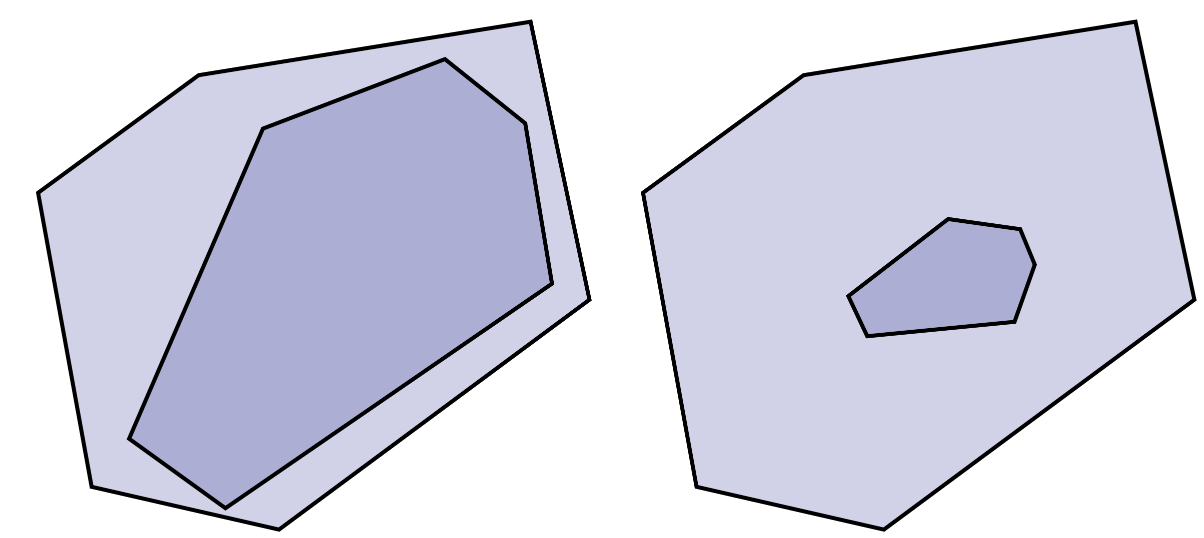
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

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.

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



Leads to surface area heuristic (SAH):

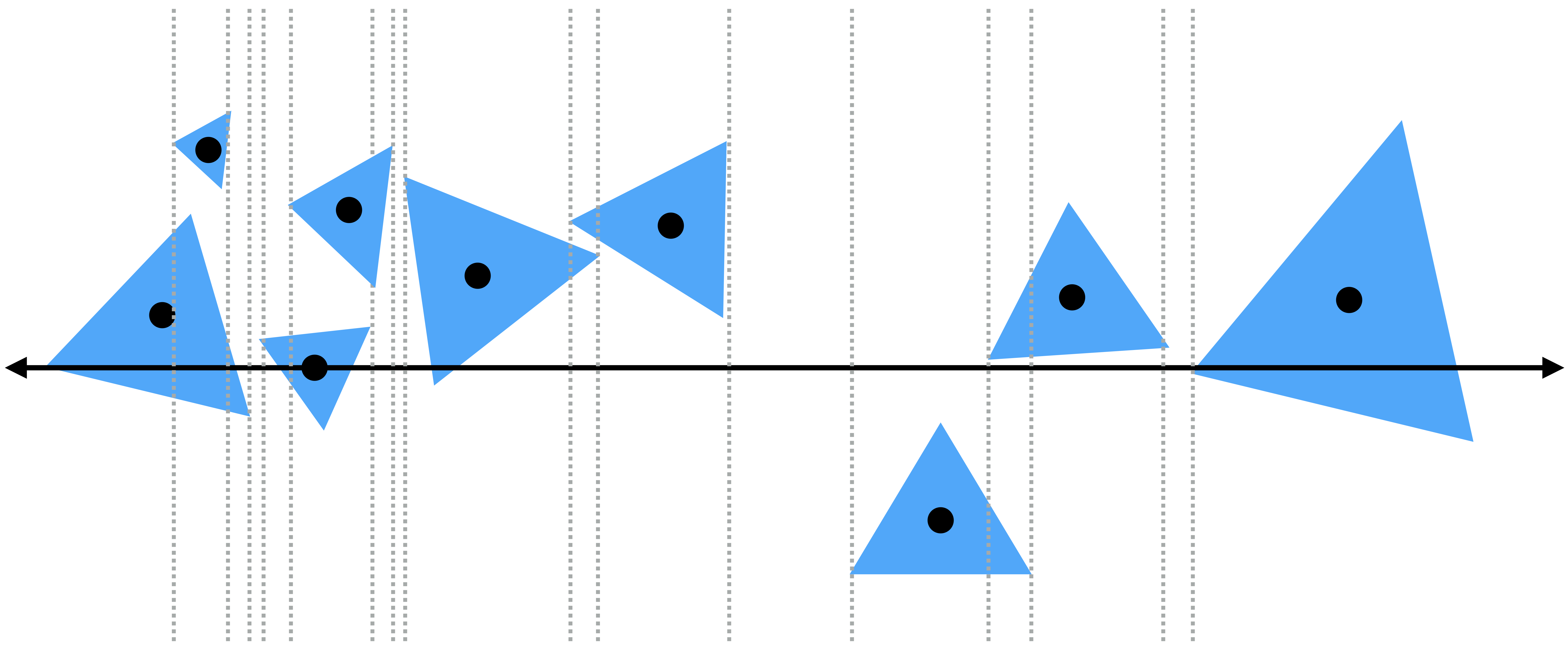
$$C = C_{\mathrm{trav}} + \frac{S_A}{S_N} N_A C_{\mathrm{isect}} + \frac{S_B}{S_N} N_B C_{\mathrm{isect}}$$

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

- Rays are randomly distributed
- Rays are not occluded

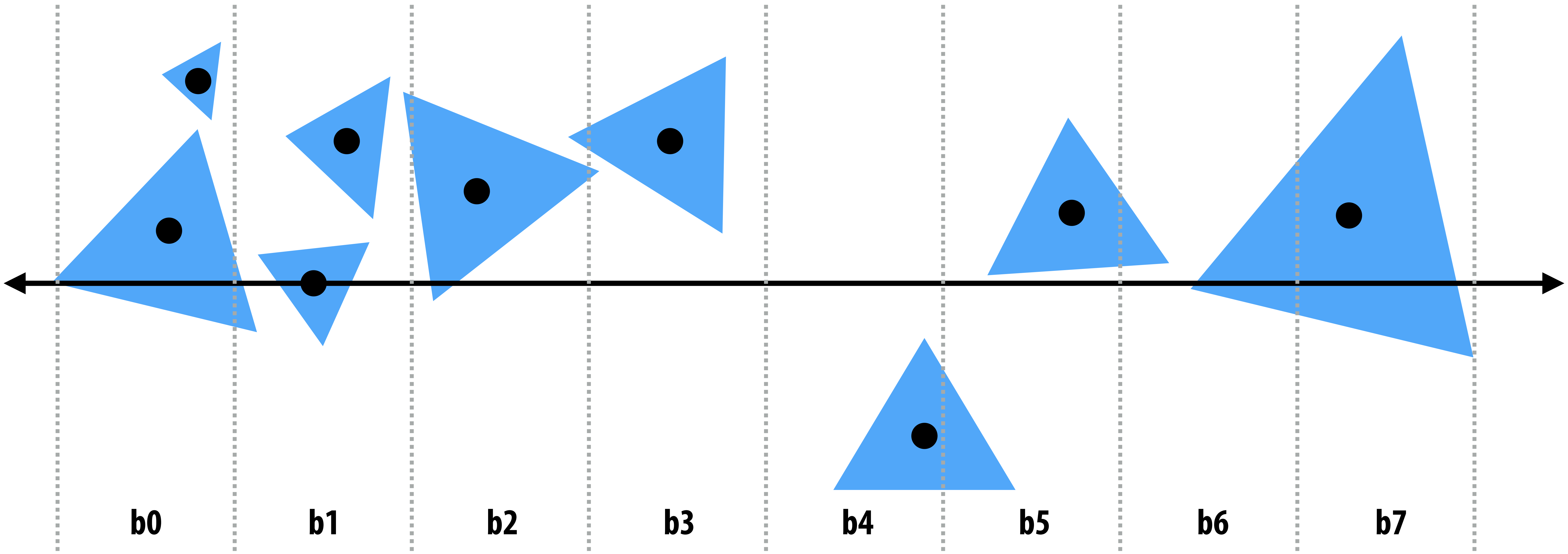
Implementing partitions

- 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)



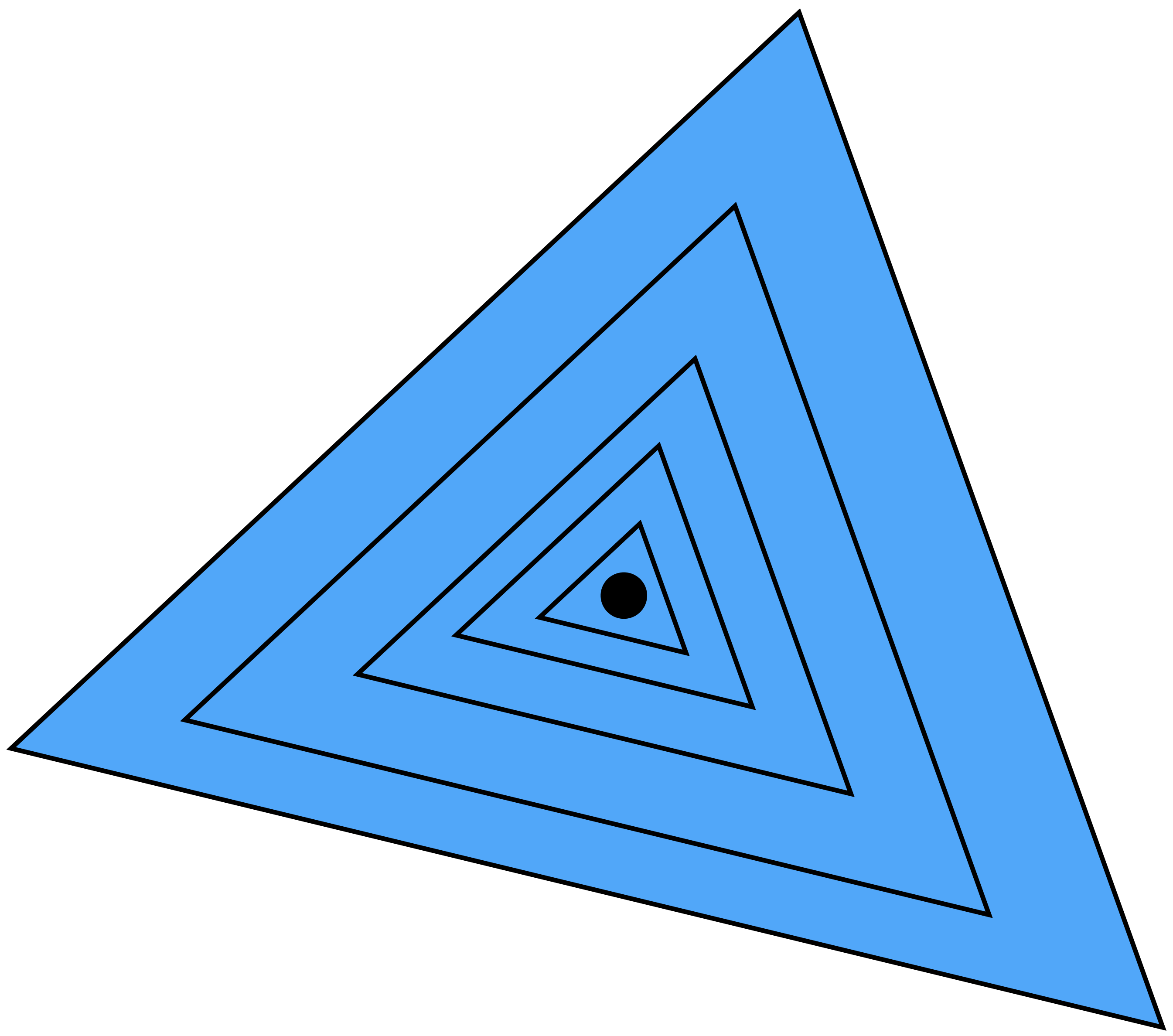
Efficiently implementing partitioning

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

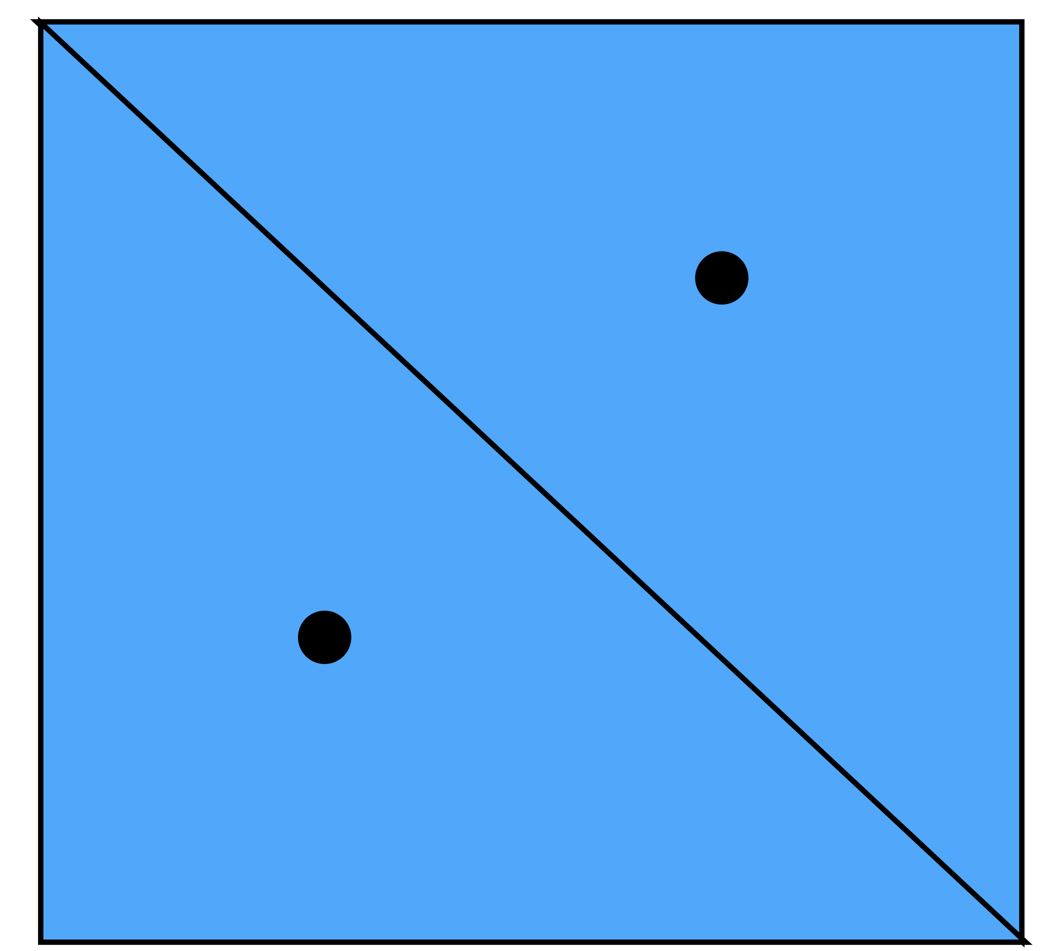



```
For each axis: x,y,z:
   initialize bucket counts to 0, bboxes to empty
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   For each of the B-1 possible partitioning planes evaluate SAH
Use lowest cost partition found (or make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

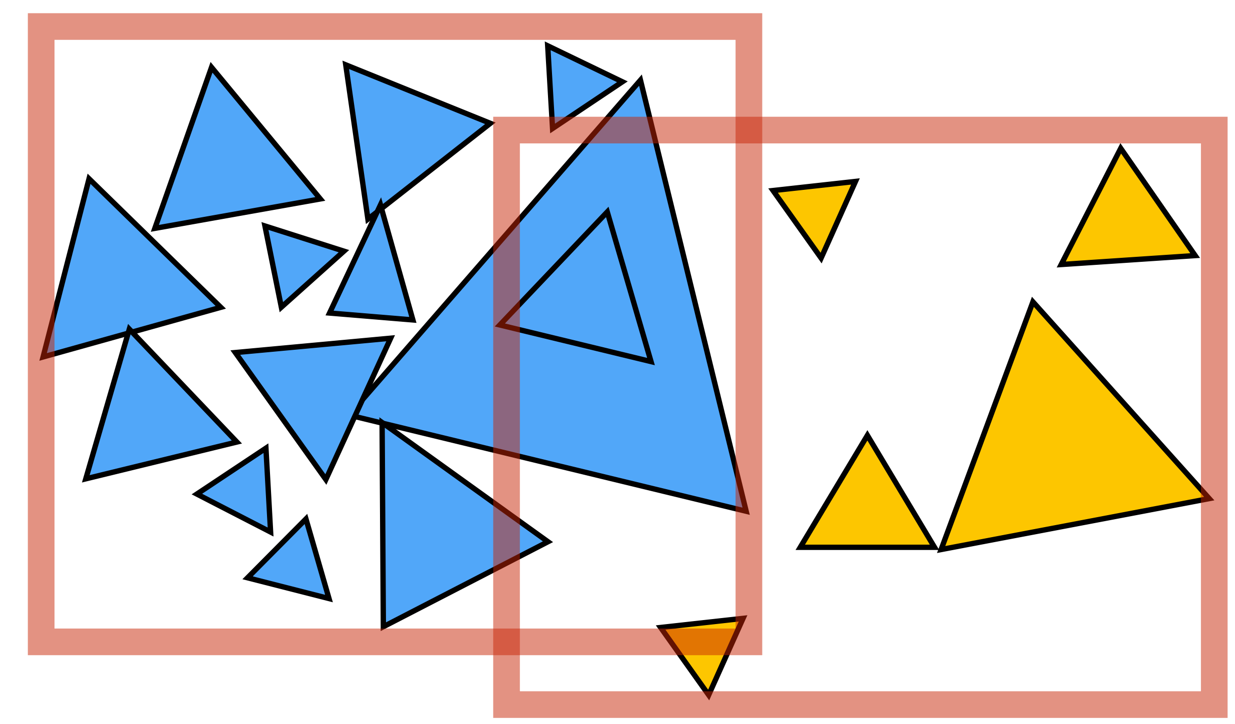
In general, different strategies may work better for different types of geometry / different distributions of primitives…

Question

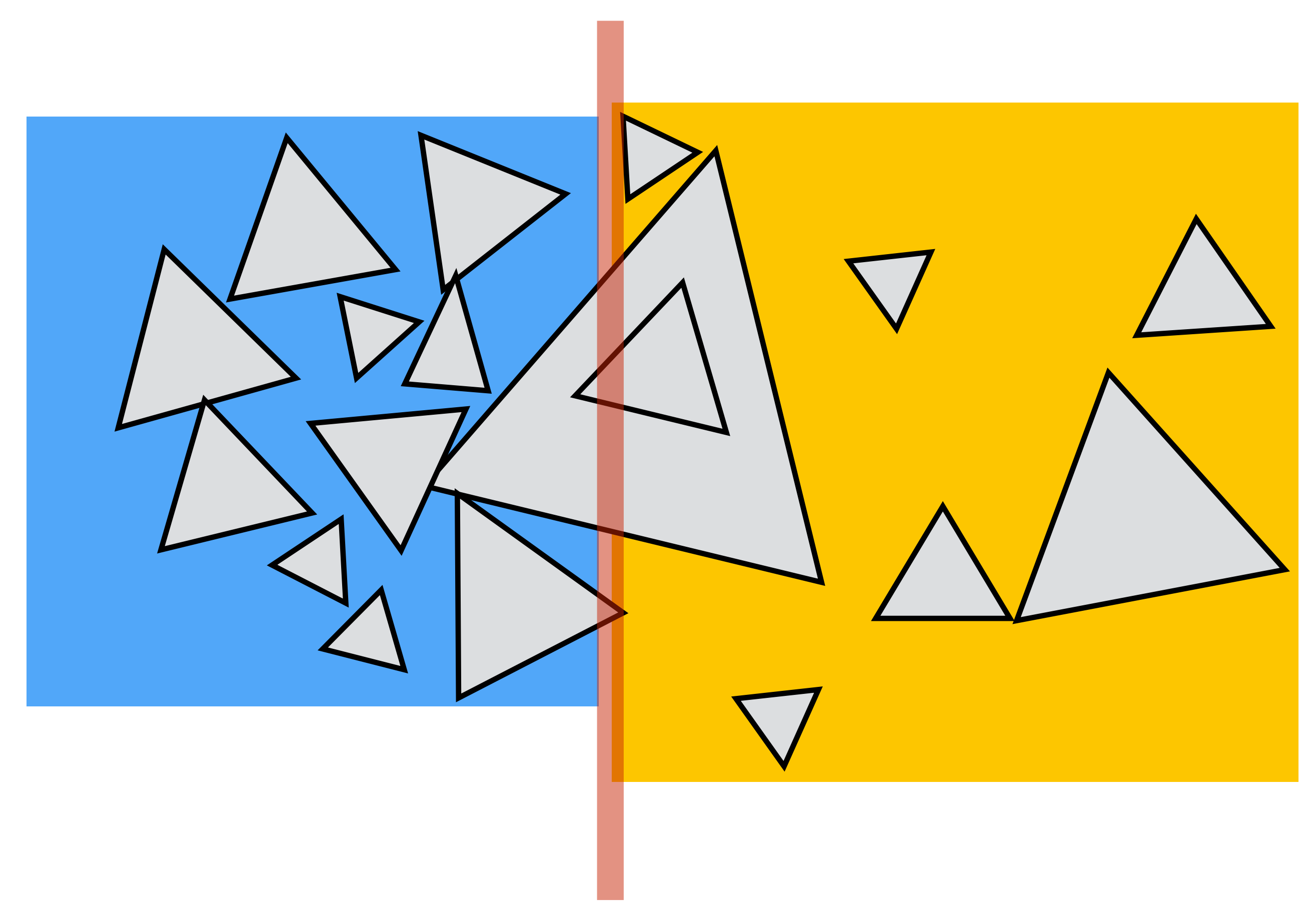
- Imagine you have a valid BVH.
- Now I move one of the triangles in the scene to a new location
- How do I “refit” the BVH so it is a valid BVH?

Primitive-partitioning acceleration structures vs. space-partitioning structures

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

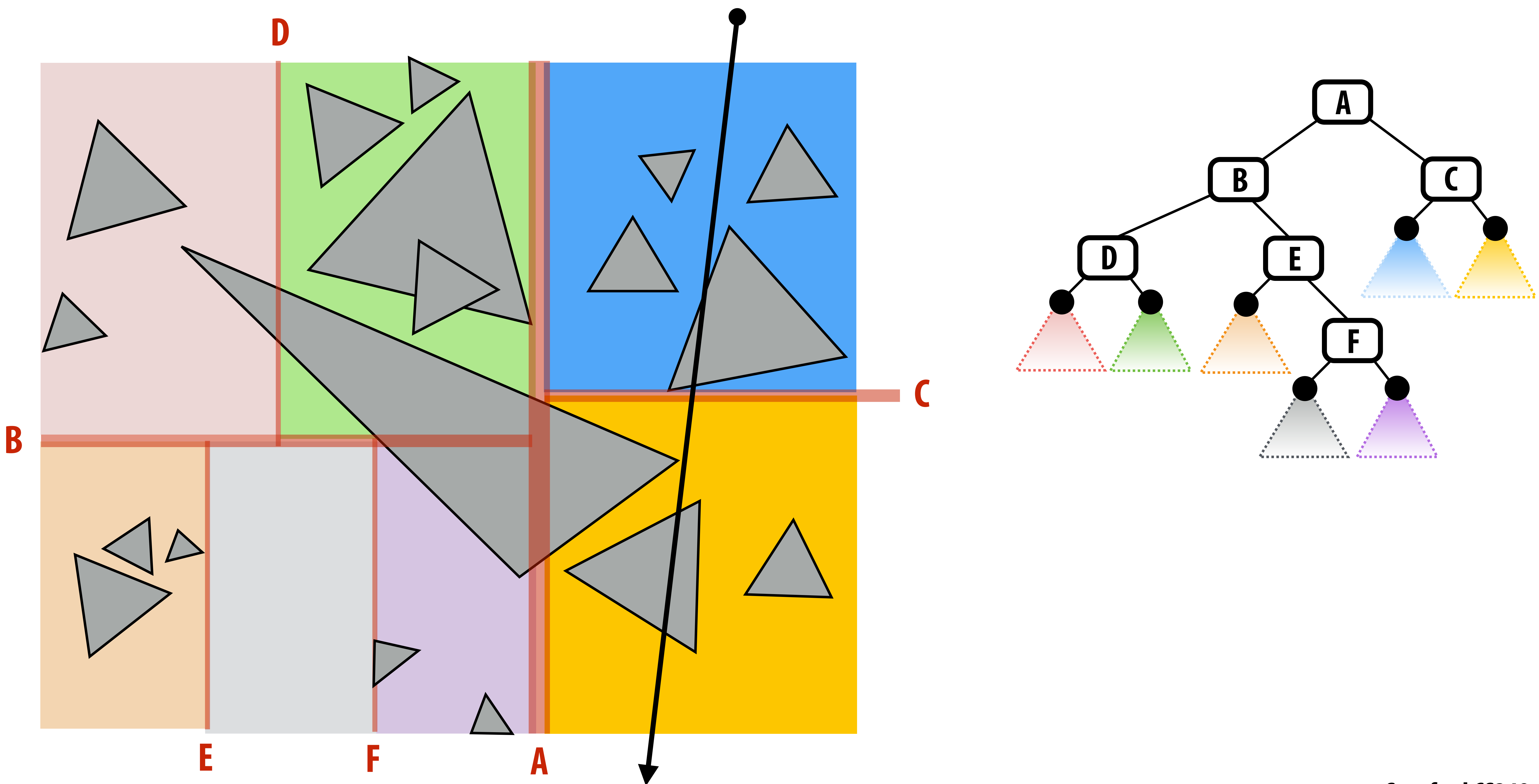


- Space-partitioning (grid, K-D tree) partitions space into disjoint regions (primitives may be contained in multiple regions of space)



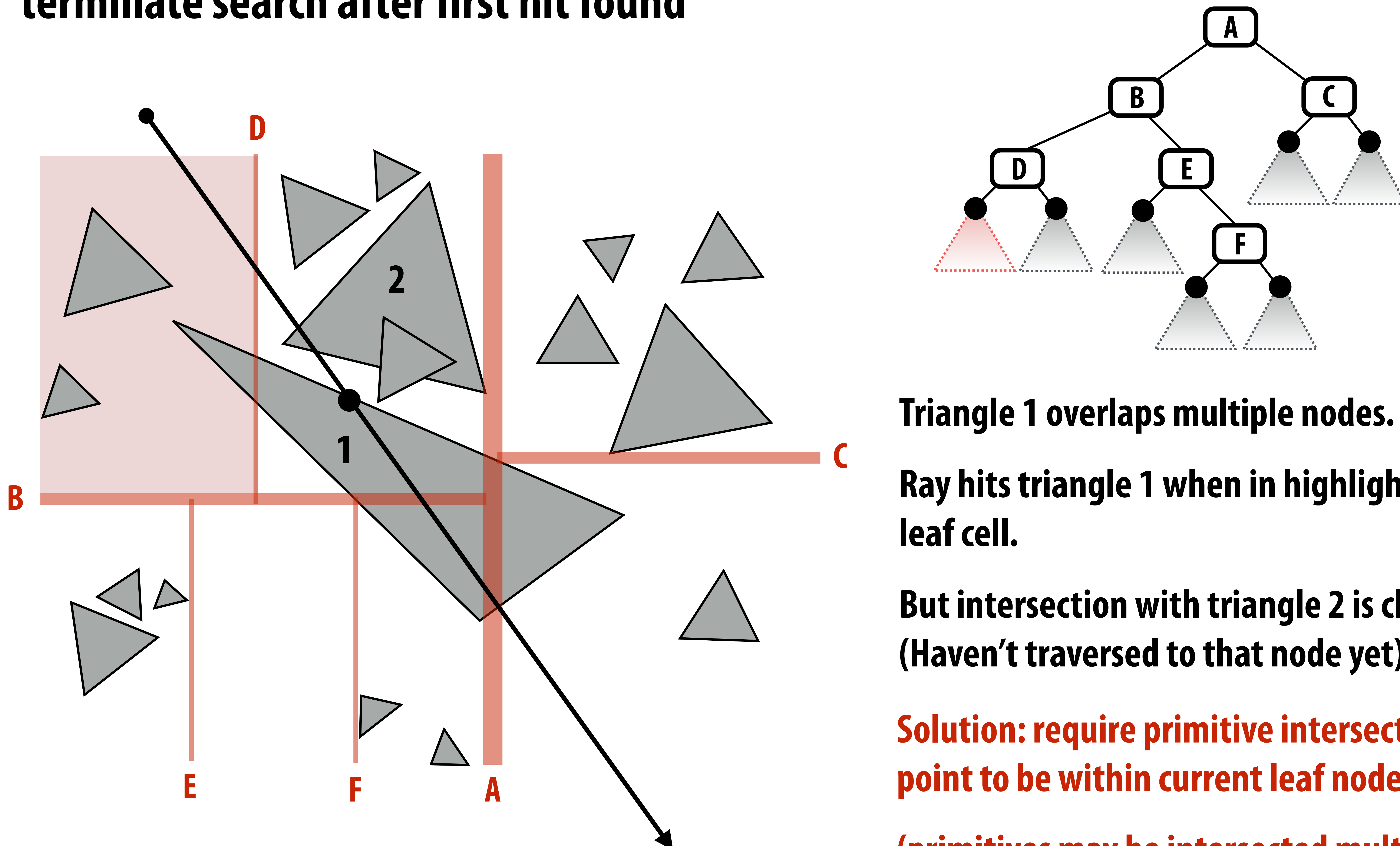
K-D tree

- Recursively partition <u>space</u> via axis-aligned partitioning planes
  - Interior nodes correspond to spatial splits
  - Node traversal can proceed in strict 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)

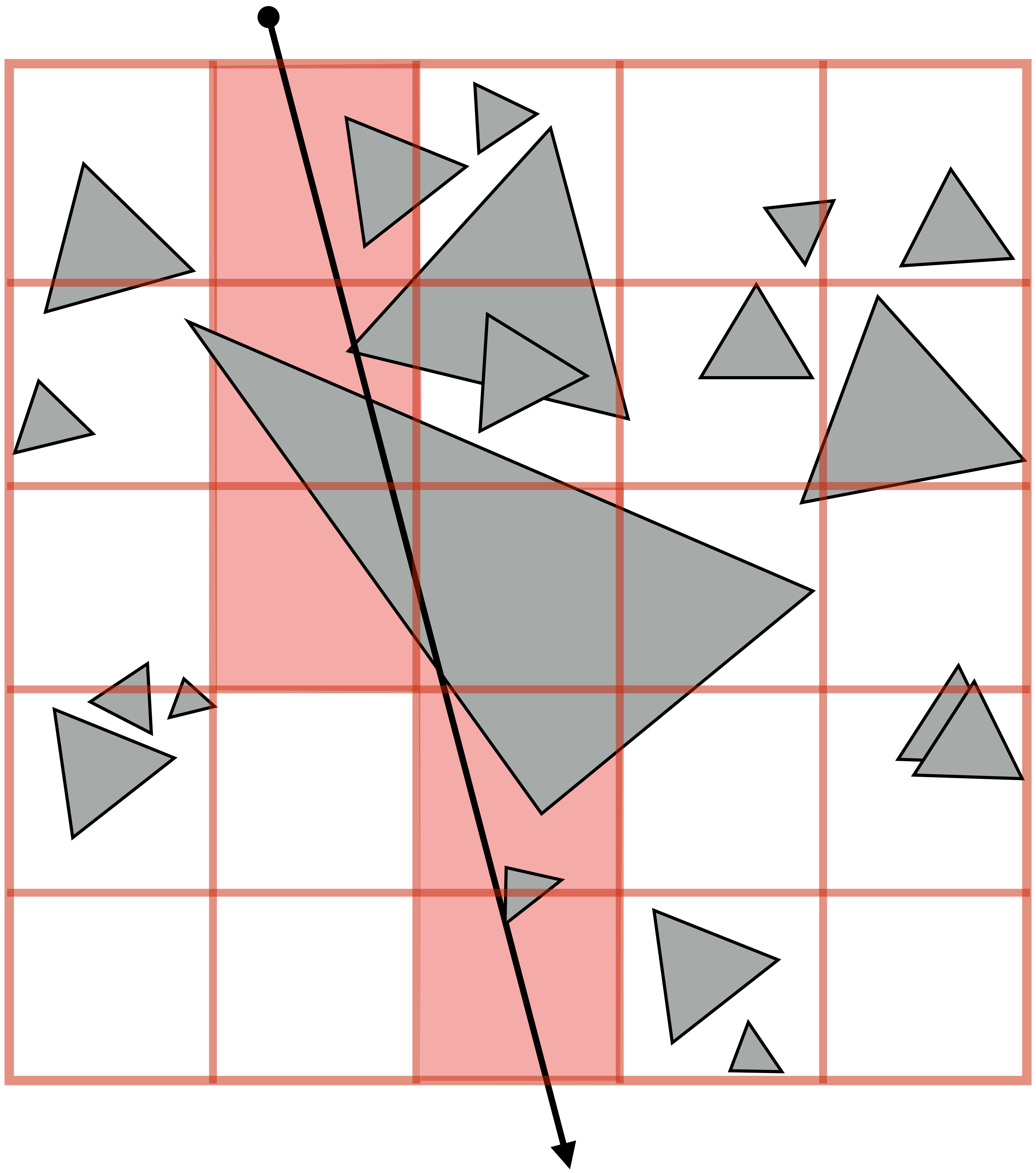
Solution: require primitive intersection point to be within current leaf node.

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

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

Uniform grid (a very simple hierarchy)

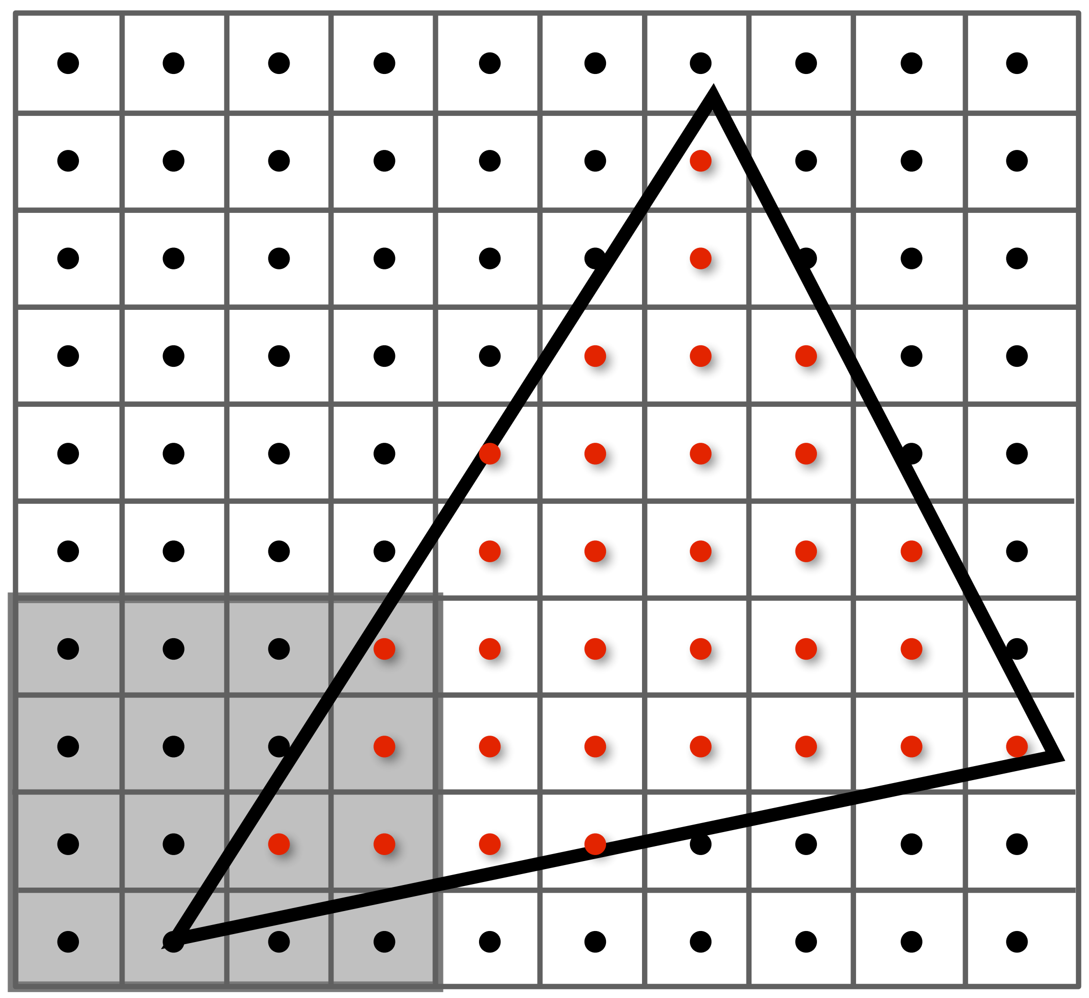
Uniform grid



- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap the 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

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]
```

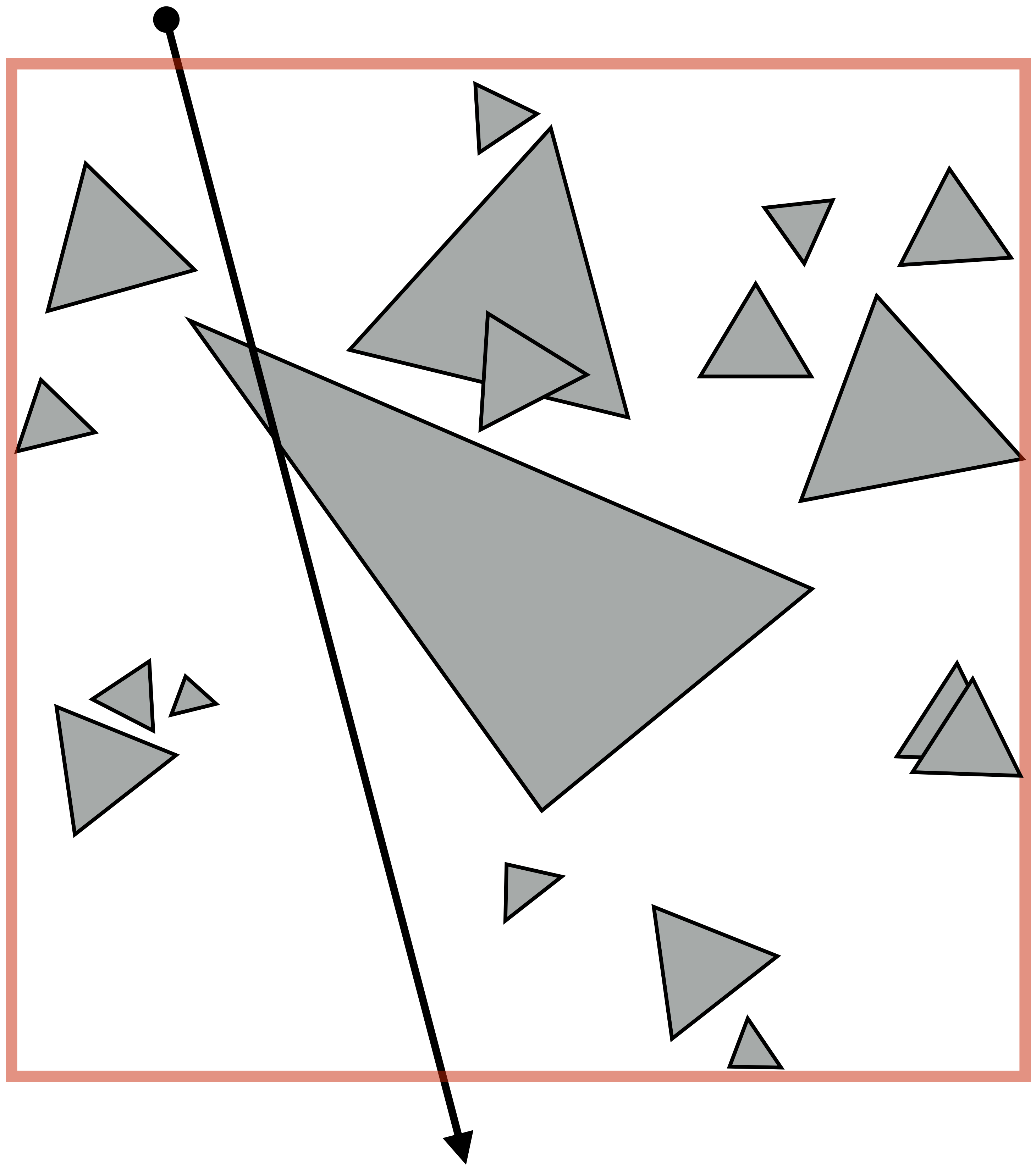


For each TILE of image

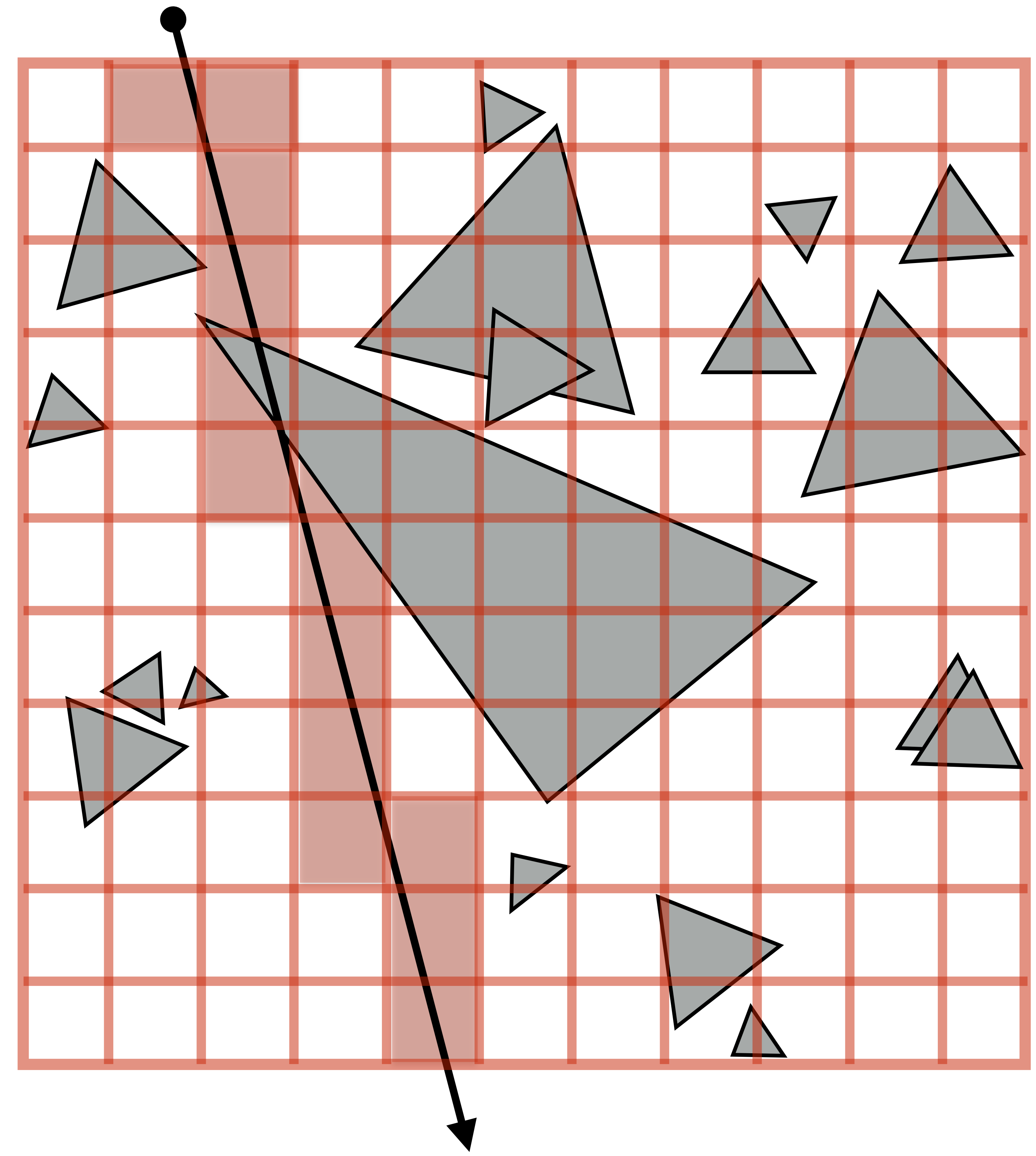
If triangle overlaps tile, check all samples in tile

What does this strategy remind you of? :-)

What should the grid resolution be?



Too few grids cell: degenerates to brute-force approach

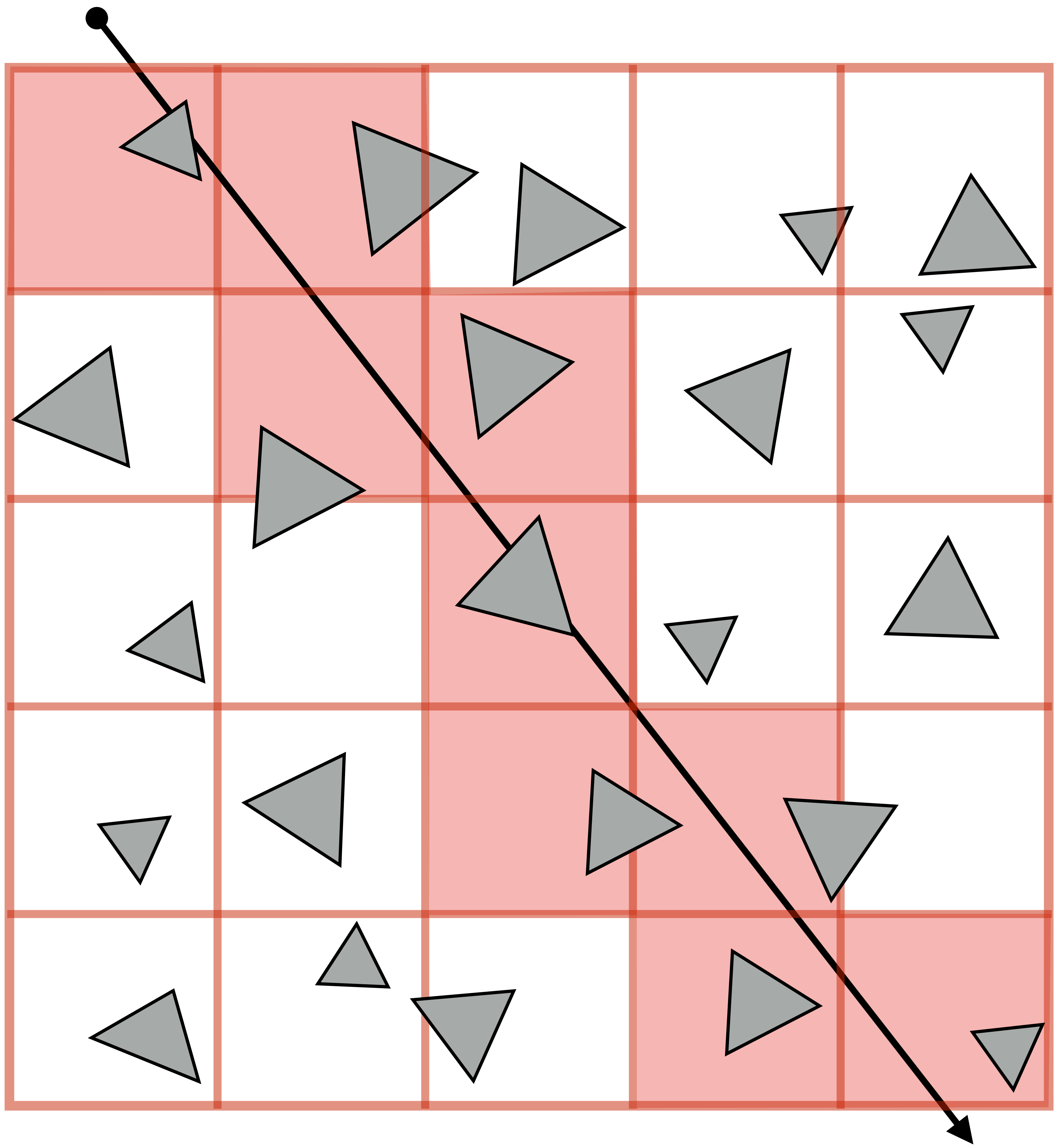


Too many grid cells: incur significant cost traversing through cells with empty space

Heuristic

- **Choose number of cells ~ total number of primitives**

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

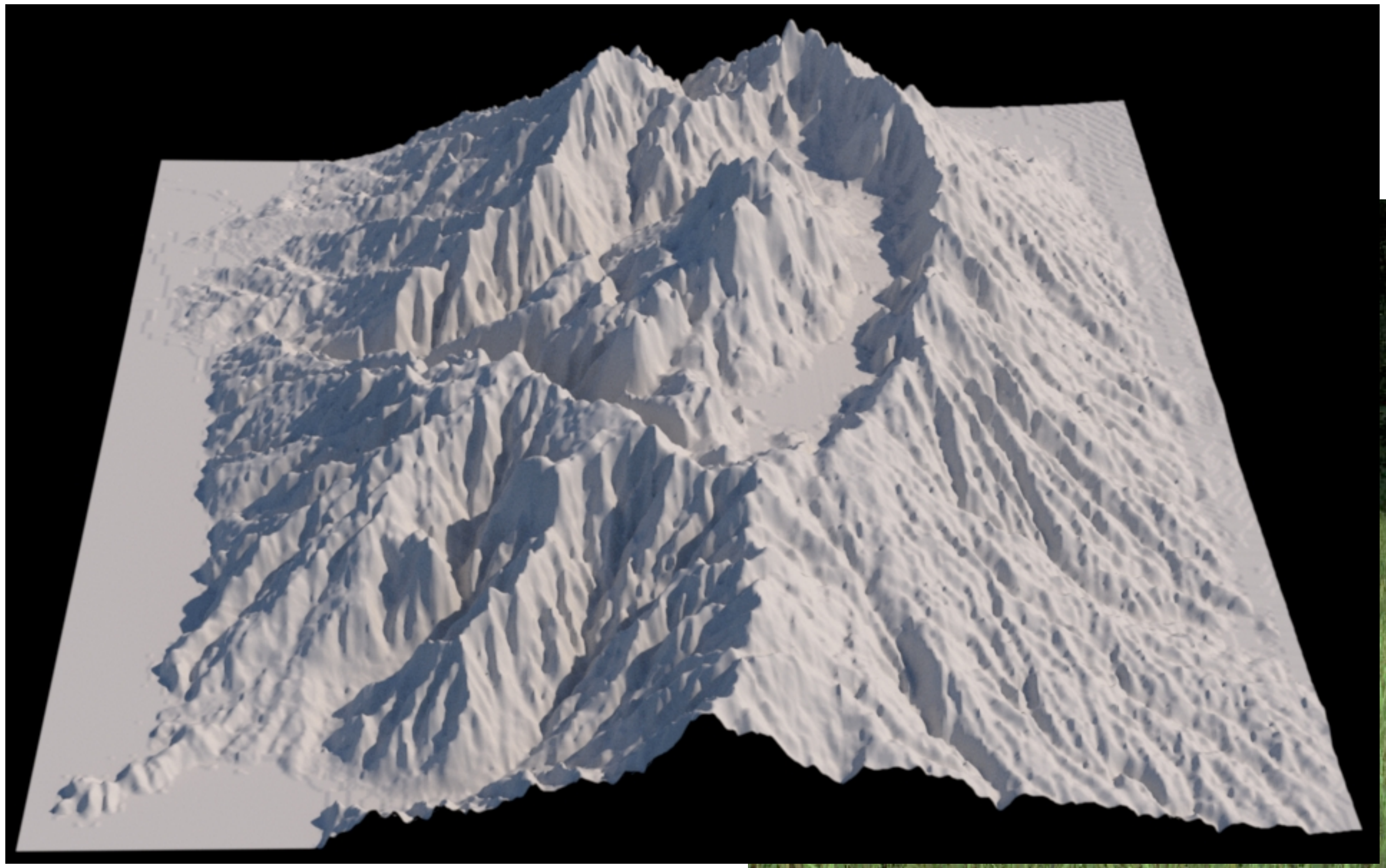


Intersection cost: $O(\sqrt[3]{N})$

(assuming 3D grid)

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

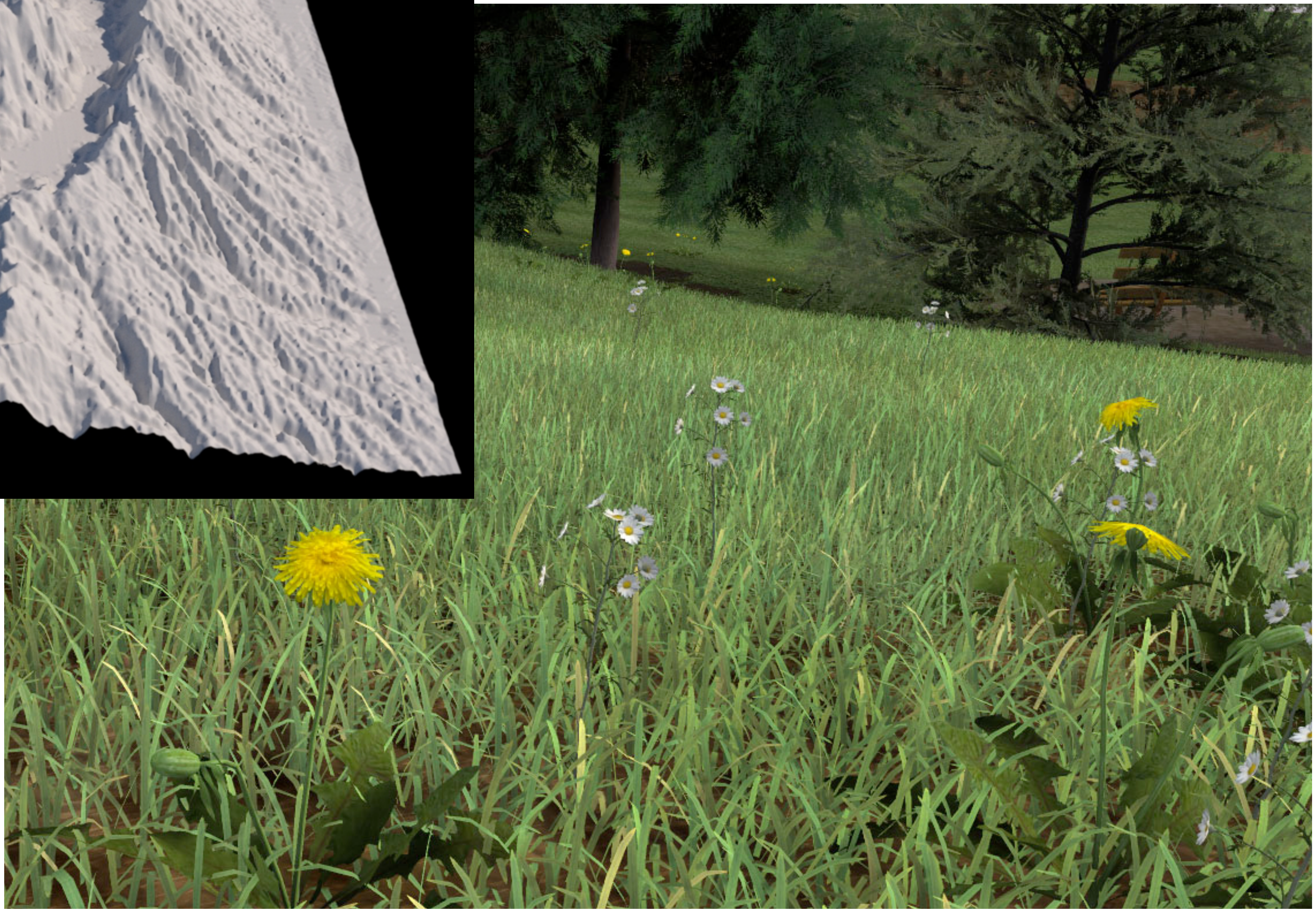
When uniform grids work well: uniform distribution of primitives in scene



Terrain / height fields:

[Image credit: Misuba Renderer]

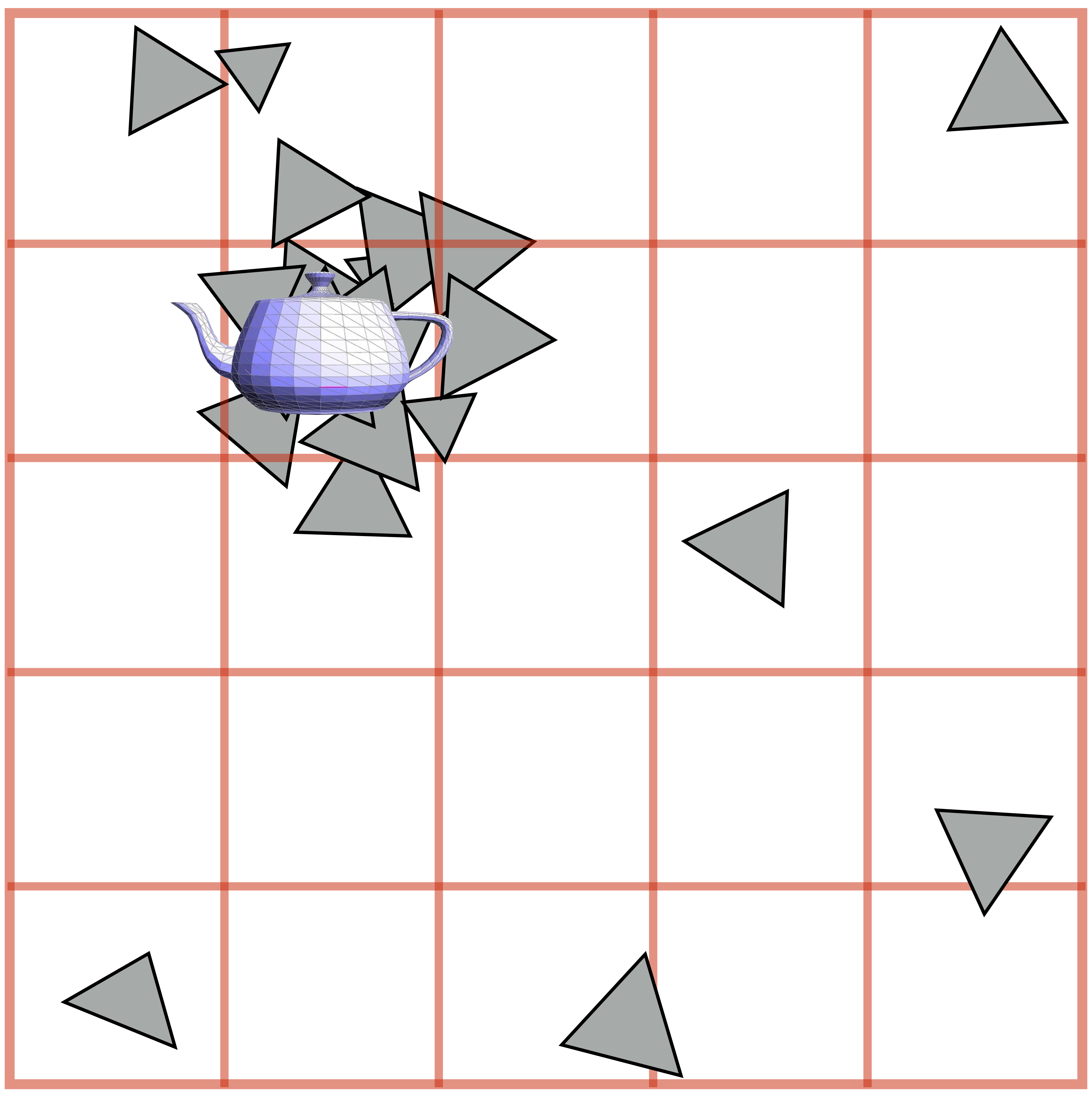
Grass:



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

Example credit: Pat Hanrahan

Uniform grids cannot adapt to non-uniform distribution of geometry in scene



"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

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

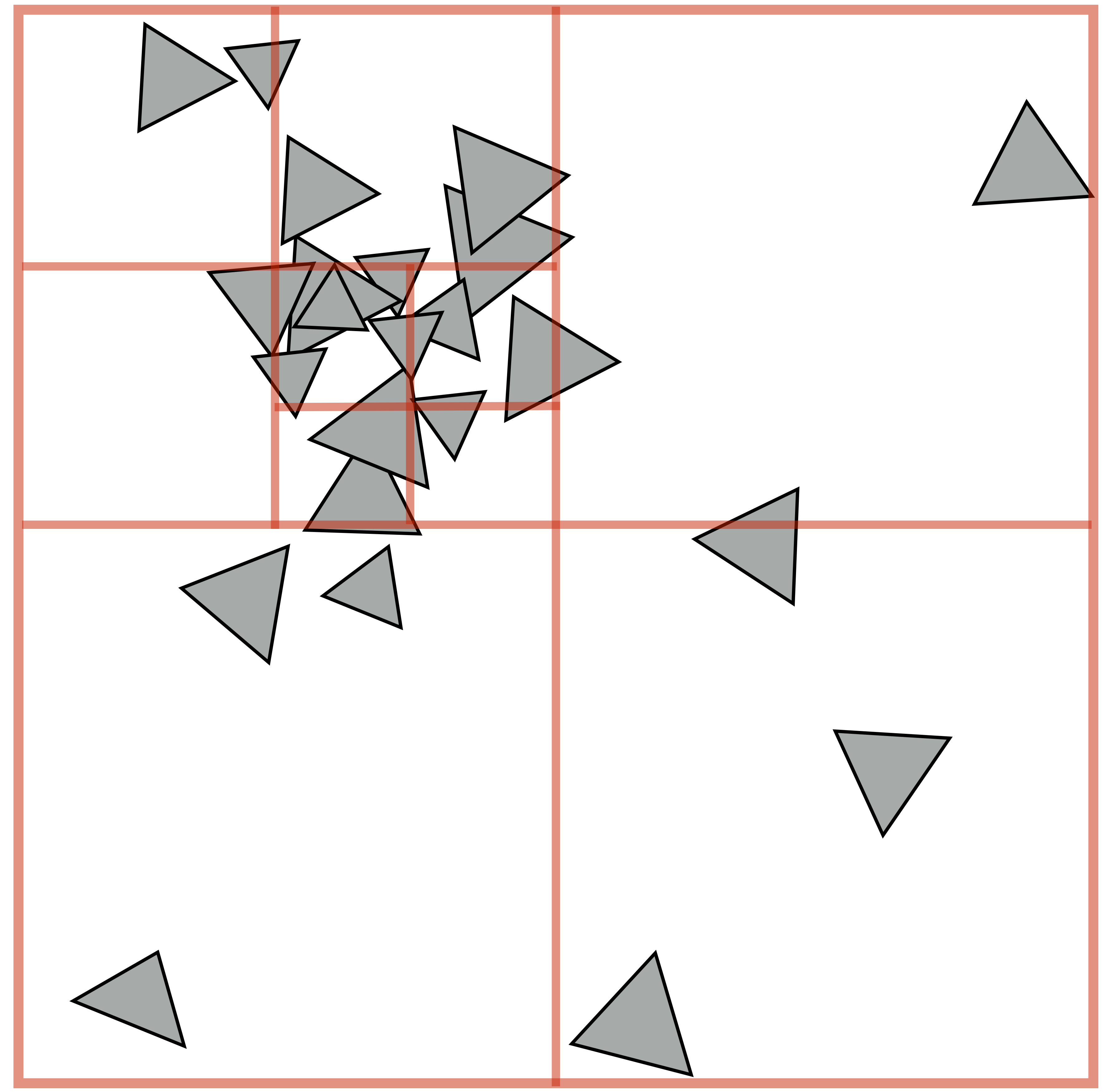
[Image credit: Pixar]

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 uniform grid.

But lower intersection performance than 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)
Octree: nodes have 8 children (partitions 3D space)

Disney Moana scene

Released for rendering research purposes in 2018.

15 billion primitives in scene

(more than 90M unique geometric primitives, instancing is used to create full scene)

Disney Moana scene

Disney Moana scene

Disney Moana scene

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

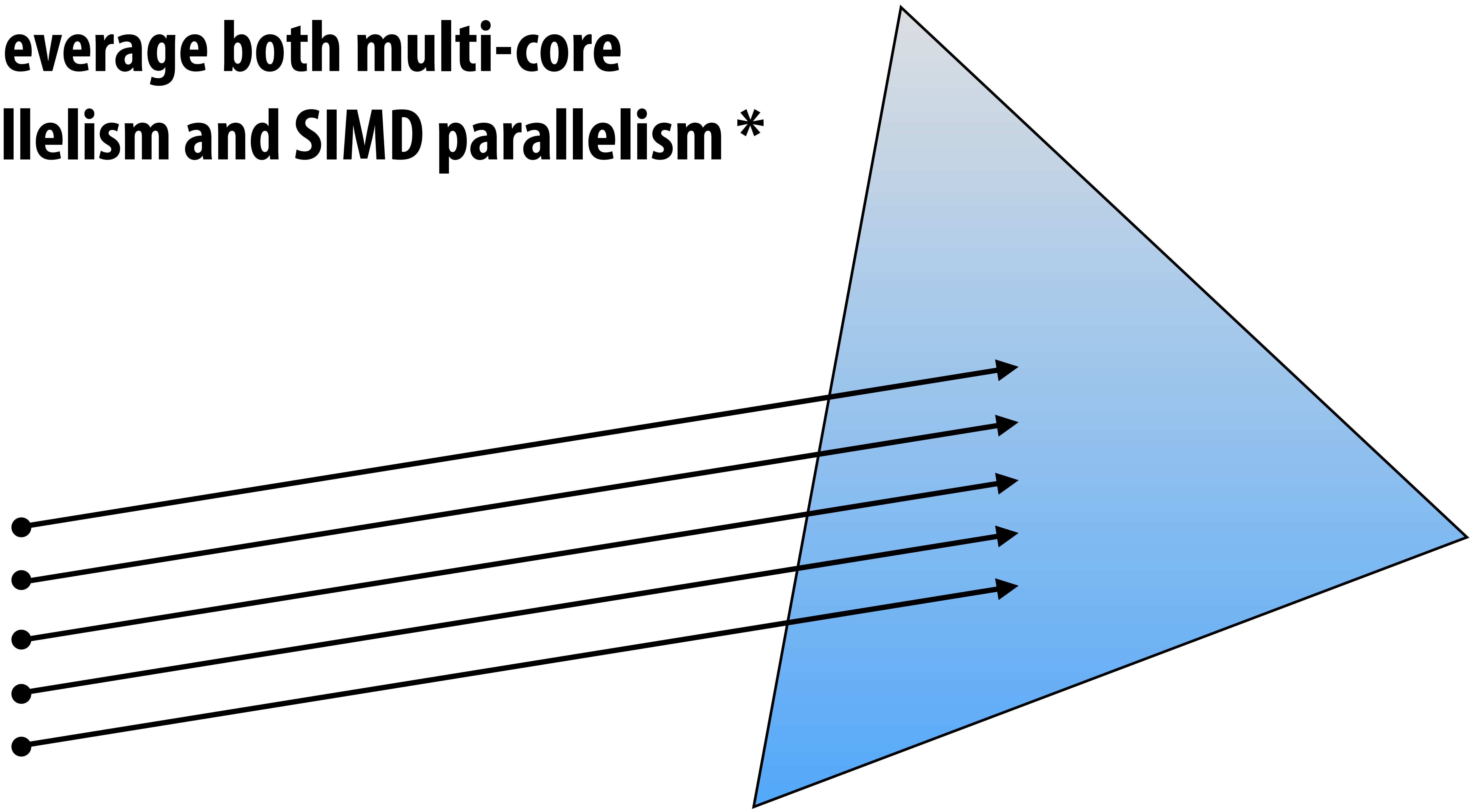
- Non-adaptive accelerations structures (uniform grids)
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed

- Many, many combinations thereof…

A few words on fast ray tracing

A ray tracer is conceptual easy to parallelize

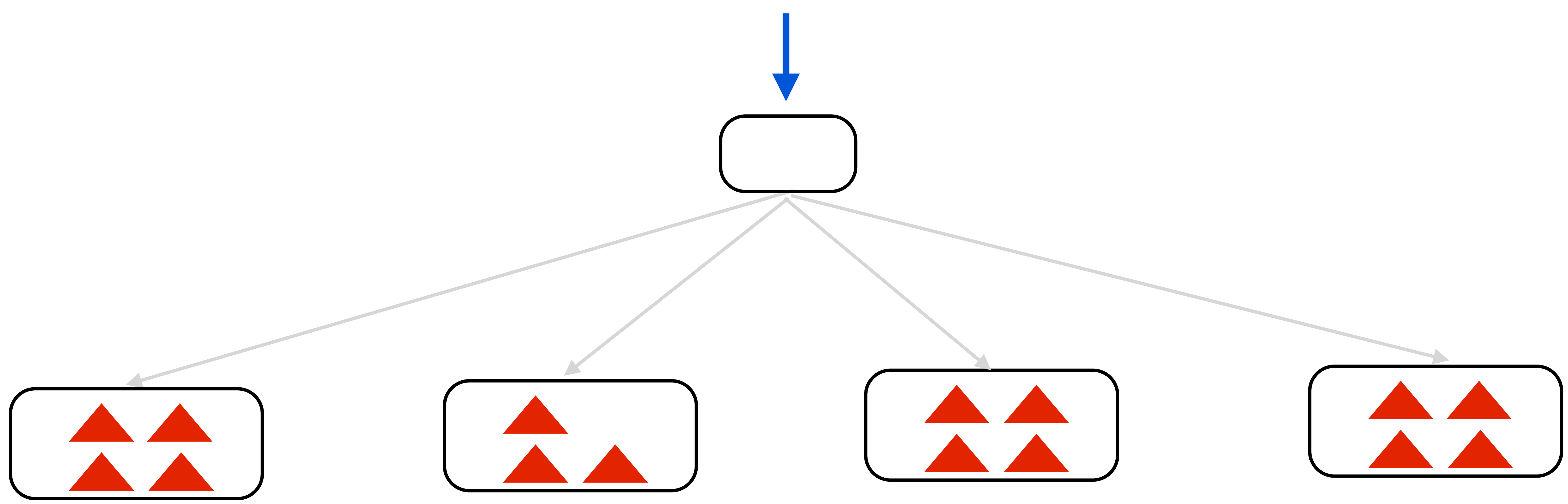
- Trace each ray against scene in parallel
- Use leverage both multi-core parallelism and SIMD parallelism *



* Take CS149 if you want to know what this means!

Wider BVHs enable easier parallelism

- Idea: use wider-branching BVH (test single ray against multiple child node bboxes in parallel)
  - In practice, BVH's with branching factor 4 have similar culling efficiency to branching factor 2
  - Good for SIMD processing architectures



[Wald et al. 2008]

Increasing interest in high performance implementations of real-time ray tracing

Microsoft's DirectX Ray Tracing support / NVIDIA's DXR announced in April 2018

Image credit: Electronic Arts (Project PICA)

Real time ray tracing

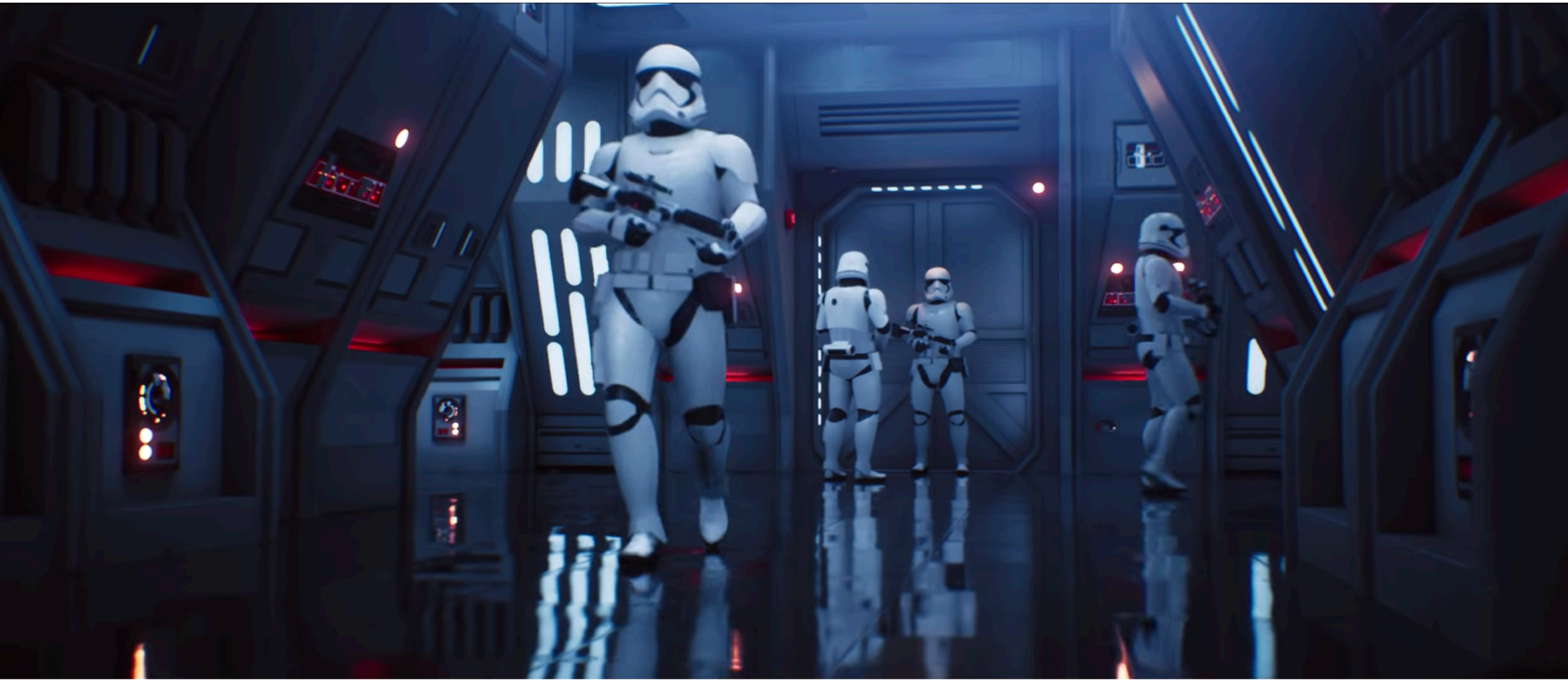


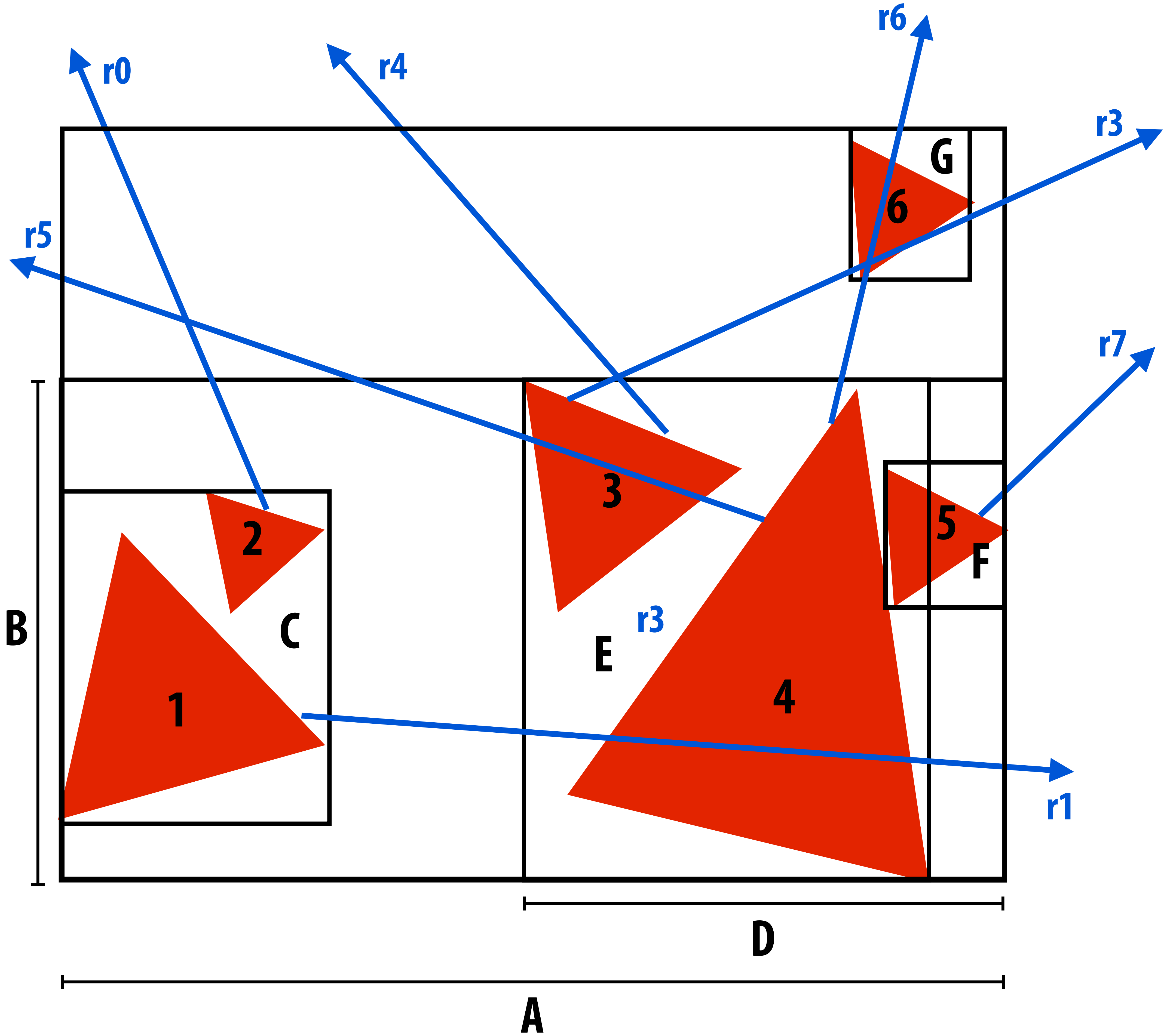
Image credit: Unreal Engine 4

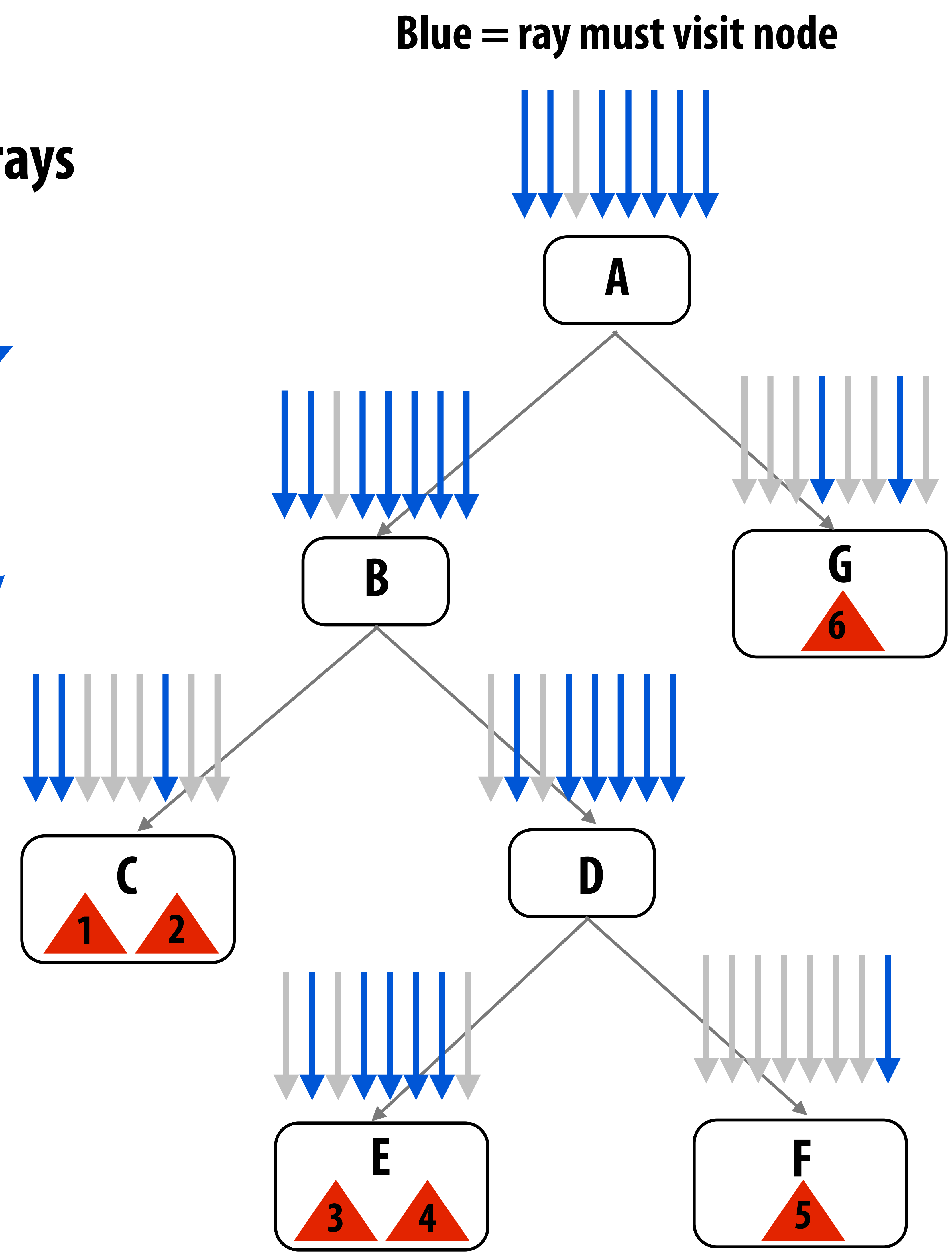
Hardware support for ray tracing

- Accelerate ray tracing by building hardware to perform operations like ray-triangle intersection and ray-BVH intersection
- Long academic history of papers…
- 2018: NVIDIA's RTX GPUs — 10B rays/sec

A key challenge is accessing memory efficiently, not just finding parallel work (again, a core CS149 topic)

- Need large amounts of DRAM for large scenes
  - So scene BVH and primitives fit in memory
- Consider cache behavior of tracing a batch of rays





Building a BVH in parallel is tricker!

- I'll post a few references for the curious

- But I recommend "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies" by Karras and Aila, HPG 2013

Fast Parallel Construction of High-Quality Bounding Volume Hierarchies

Tero Karras Timo Aila

NVIDIA

Abstract

We propose a new massively parallel algorithm for constructing high-quality bounding volume hierarchies (BVHs) for ray tracing. The algorithm is based on modifying an existing BVH to improve its quality, and executes in linear time at a rate of almost 40M triangles/sec on NVIDIA GTX Titan. We also propose an improved approach for parallel splitting of triangles prior to tree construction. Averaged over 20 test scenes, the resulting trees offer over 90% of the ray tracing performance of the best offline construction method (SBVH), while previous fast GPU algorithms offer only about 50%. Compared to state-of-the-art, our method offers a significant improvement in the majority of practical workloads that need to construct the BVH for each frame. On the average, it gives the best overall performance when tracing between 7 million and 60 billion rays per frame. This covers most interactive applications, product and architectural design, and even movie rendering.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: ray tracing, bounding volume hierarchies

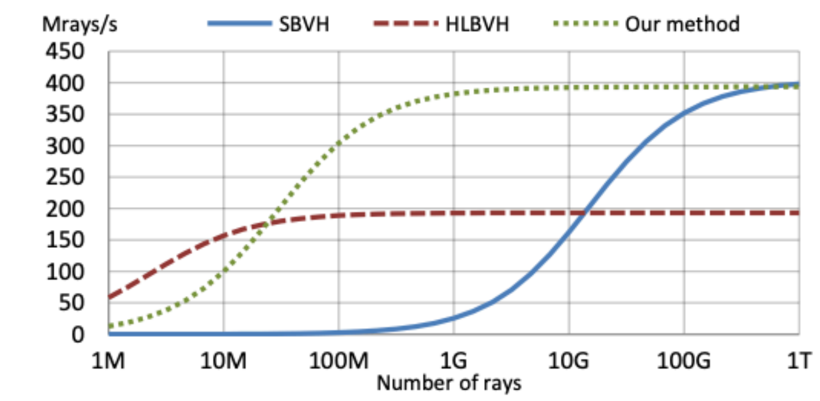


Figure 1: *Performance of constructing a BVH and then casting a number of diffuse rays with NVIDIA GTX Titan in SODA (2.2M triangles). SBVH [Stich et al. 2009] yields excellent ray tracing performance, but suffers from long construction times. HLBVH [Garanzha et al. 2011a] is very fast to construct, but reaches only about 50% of the performance of SBVH. Our method is able to reach 97% while still being fast enough to use in interactive applications. In this particular scene, it offers the best quality–speed tradeoff for workloads ranging from 30M to 500G rays per frame.*

1 Introduction

Ray tracing is the main ingredient in most of the realistic rendering algorithms, ranging from offline image synthesis to interactive visualization. While GPU computing has been successful in accelerating the tracing of rays [Aila and Laine 2009; Aila et al. 2012], the problem of constructing high-quality acceleration structures needed to reach this level of performance remains elusive when precomputation is not an option.

Bounding volume hierarchies (BVHs) are currently the most popular acceleration structures for GPU ray tracing because of their low memory footprint and flexibility in adapting to temporal changes in scene geometry. High-quality BVHs are typically constructed using a greedy top-down sweep [MacDonald and Booth 1990; Stich et al. 2009], commonly considered to be the gold standard in ray tracing performance. Recent methods [Kensler 2008; Bittner et al. 2013] can also provide comparable quality by restructuring an existing, lower quality BVH as a post-process. Still, the construction of high-quality BVHs is computationally intensive and difficult to parallelize, which makes these methods poorly suited for applications where the geometry changes between frames. This includes most interactive applications, product and architectural visualization, and movie production.

Recently, a large body of research has focused on tackling the problem of animated scenes by trading BVH quality for increased construction speed [Wald 2007; Pantaleoni and Luebke 2010; Garanzha et al. 2011a; Garanzha et al. 2011b; Karras 2012; Kopta et al. 2012]. Most of these methods are based on limiting the search space of the top-down sweep algorithm, and they can yield significant increases in construction speed by utilizing the massive parallelism offered by GPUs. However, the BVH quality achieved by these methods falls short of the gold standard, which makes them practical only when the expected number of rays per frame is small.

The practical problem facing many applications is that the gap between the two types of construction methods is too wide (Figure 1). For moderately sized workloads, the high-quality methods are too slow to be practical, whereas the fast ones do not achieve sufficient ray tracing performance. In this paper, we bridge the gap by presenting a novel GPU-based construction method that achieves performance close to the best offline methods, while at the same time executing fast enough to remain competitive with the fast GPU-based ones. Furthermore, our method offers a way to adjust the quality–speed tradeoff in a scene-independent manner to suit the needs of a given application.

Our main contribution is a massively parallel GPU algorithm for restructuring an existing BVH in order to maximize its expected ray tracing performance. The idea is to look at local neighborhoods of nodes, i.e., *treelets*, and solve an NP-hard problem for each treelet to find the optimal topology for its nodes. Even though the optimization itself is exponential with respect to the size of the treelet, the overall algorithm scales linearly with the size of the scene. We show that even very small treelets are powerful enough to transform a low-quality BVH that can be constructed in a matter of milliseconds into a high-quality one that is close to the gold standard in ray tracing performance.

Our second contribution is a novel heuristic for splitting triangles prior to the BVH construction that further improves ray tracing performance to within 10% of the best split-based construction method to date [Stich et al. 2009]. We extend the previous work [Ernst and Greiner 2007; Dammertz and Keller 2008] by providing a more accurate estimate for the expected benefit of splitting a given triangle, and by taking steps to ensure that the chosen split planes agree with each other to reduce node overlap more effectively.

Acknowledgements

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