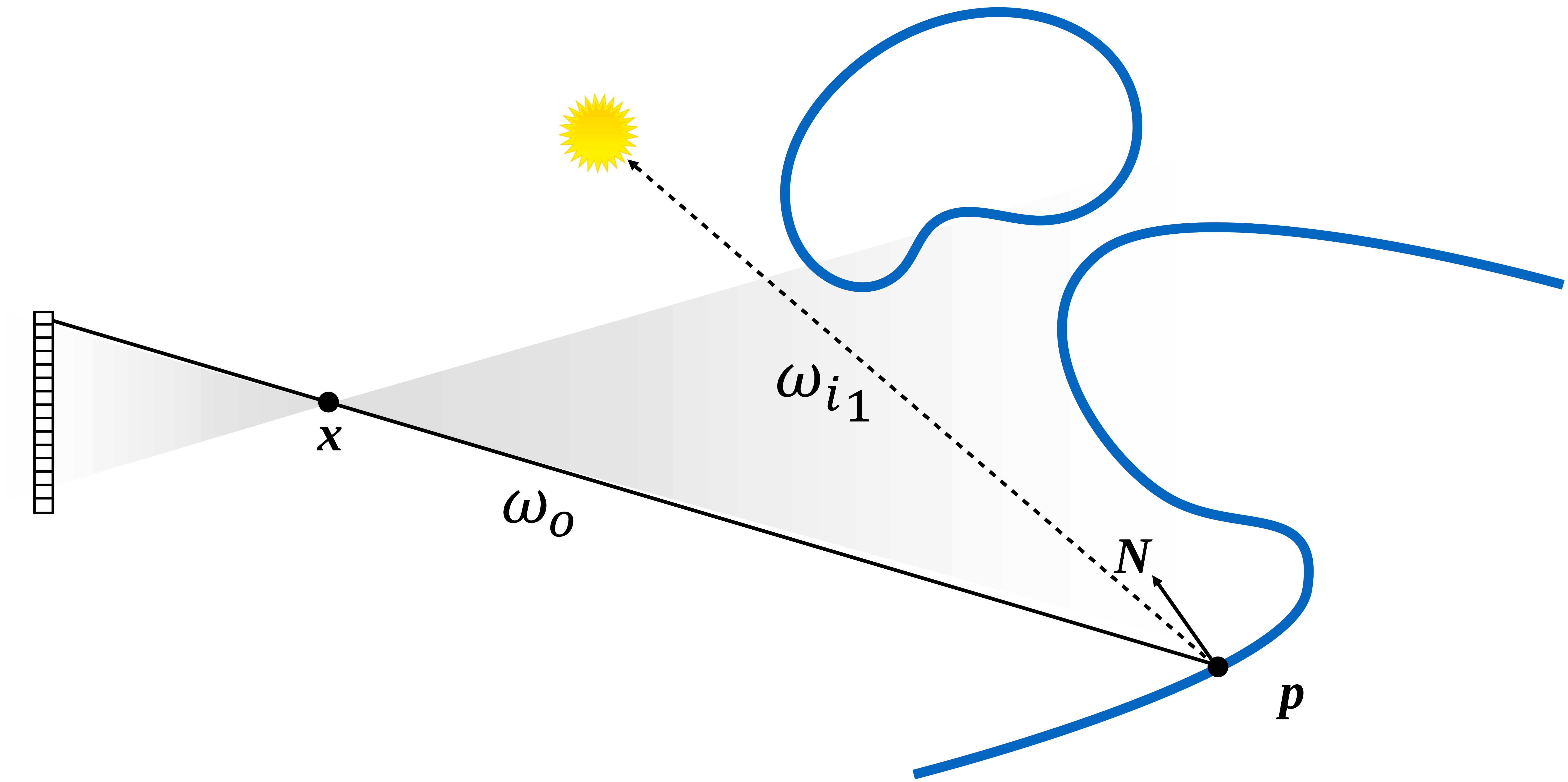
Ray Tracing

From last class...



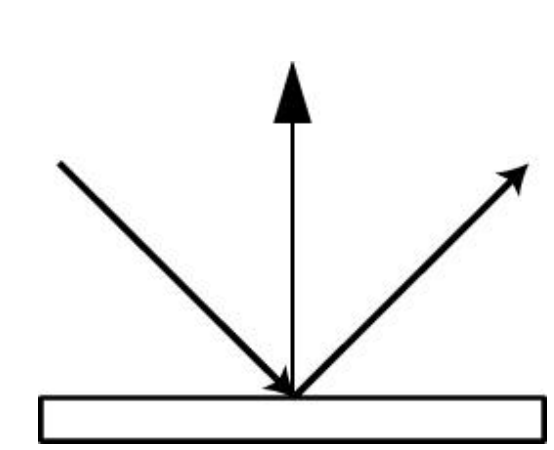
What we really want is to solve the rendering equation:

$$L_o(\mathrm{p},\omega_o) = L_e(\mathrm{p},\omega_o) + \int_{H^2} f_r(\mathrm{p},\omega_i \rightarrow \omega_o)\, L_i(\mathrm{p},\omega_i)\, \cos\theta_i\, \mathrm{d}\omega_i$$

Some basic reflection functions

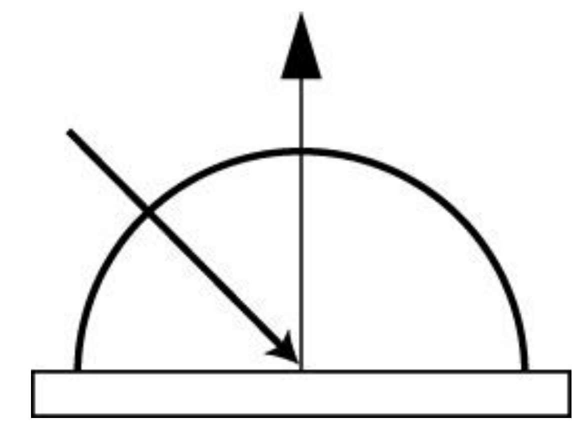
- **Ideal specular**

  Perfect mirror



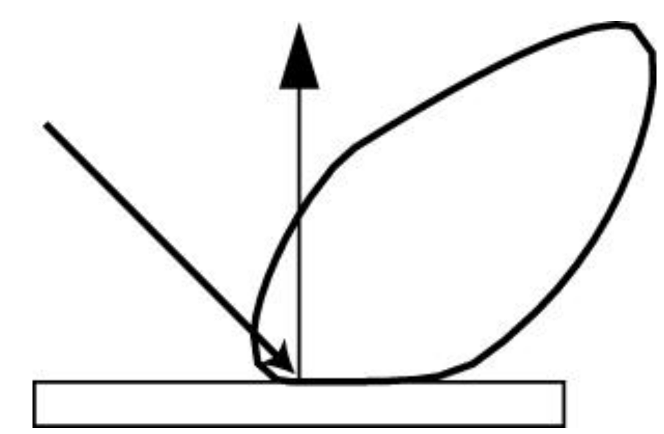
- **Ideal diffuse**

  Uniform reflection in all directions



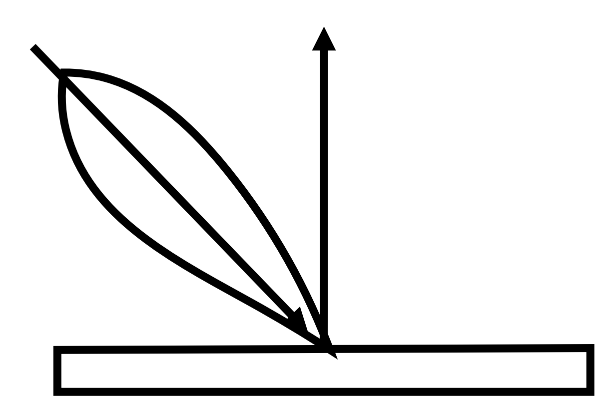
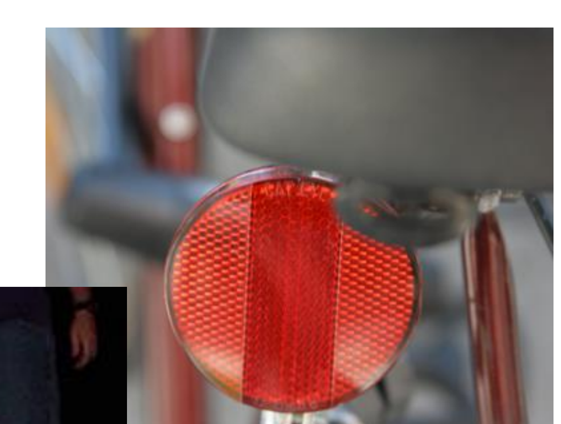

- **Glossy specular**

  Majority of light distributed in reflection direction



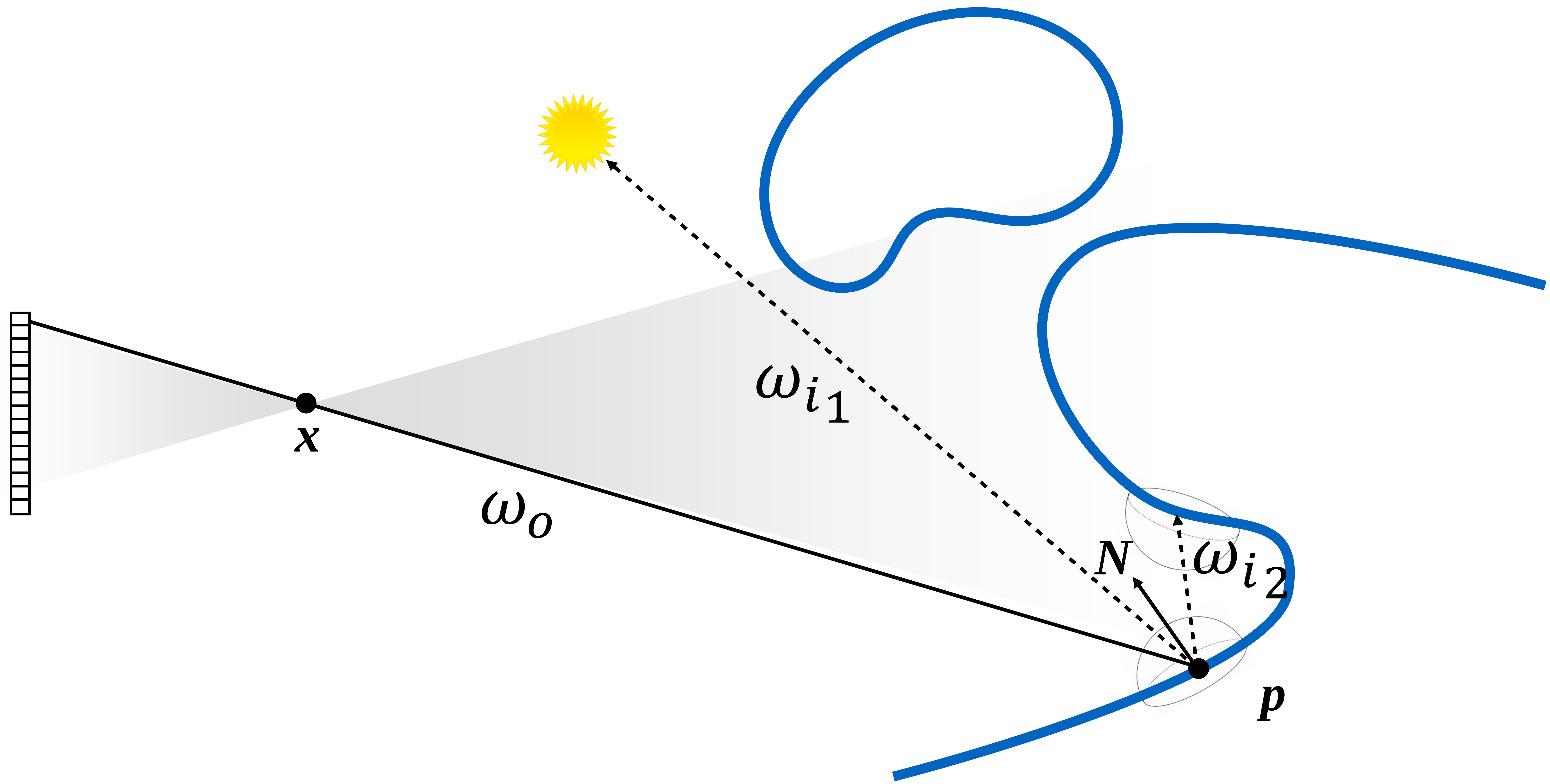
- **Retro-reflective**

  Reflects light back toward source

Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

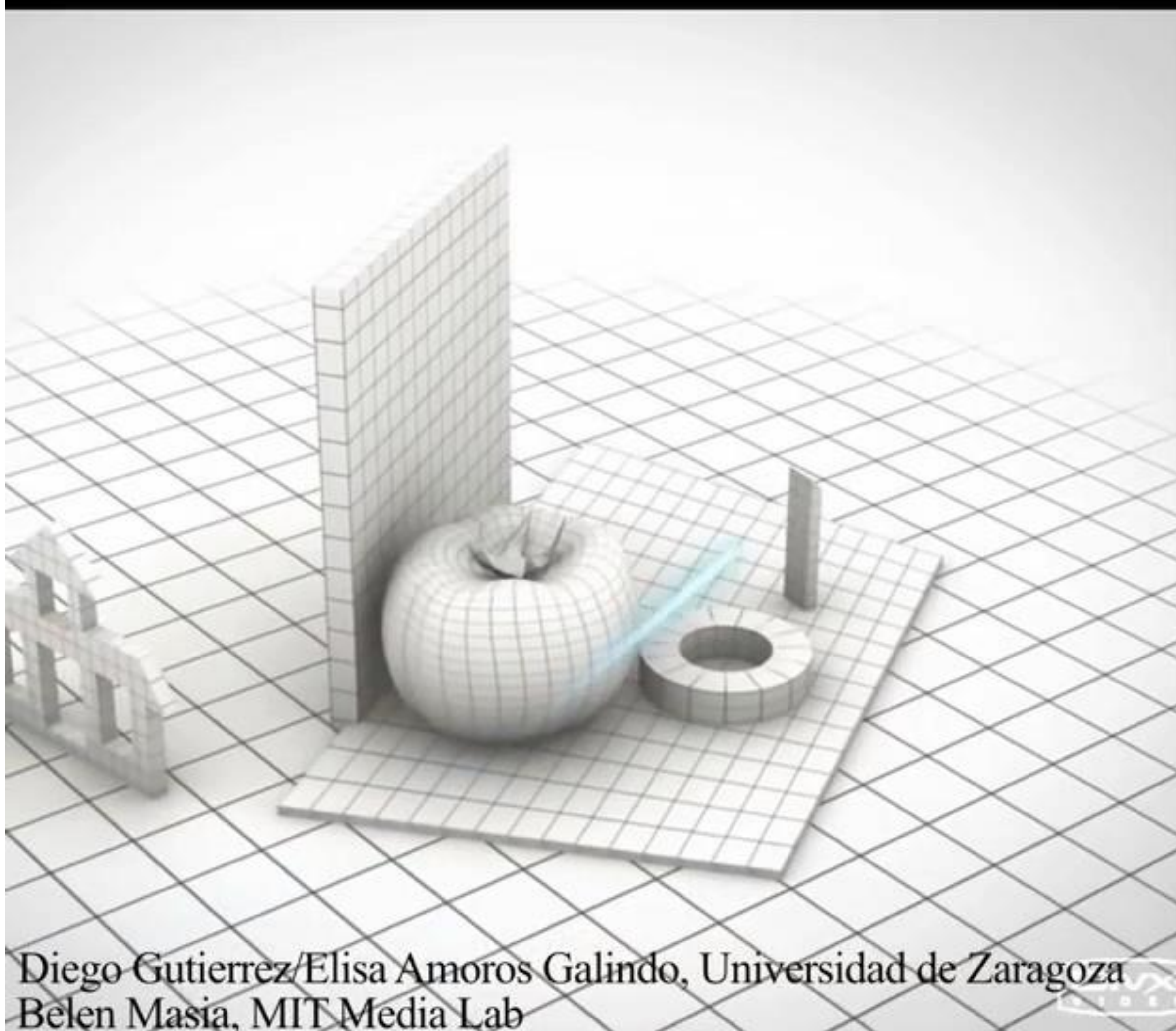
From last class...



What we really want is to solve the rendering equation:

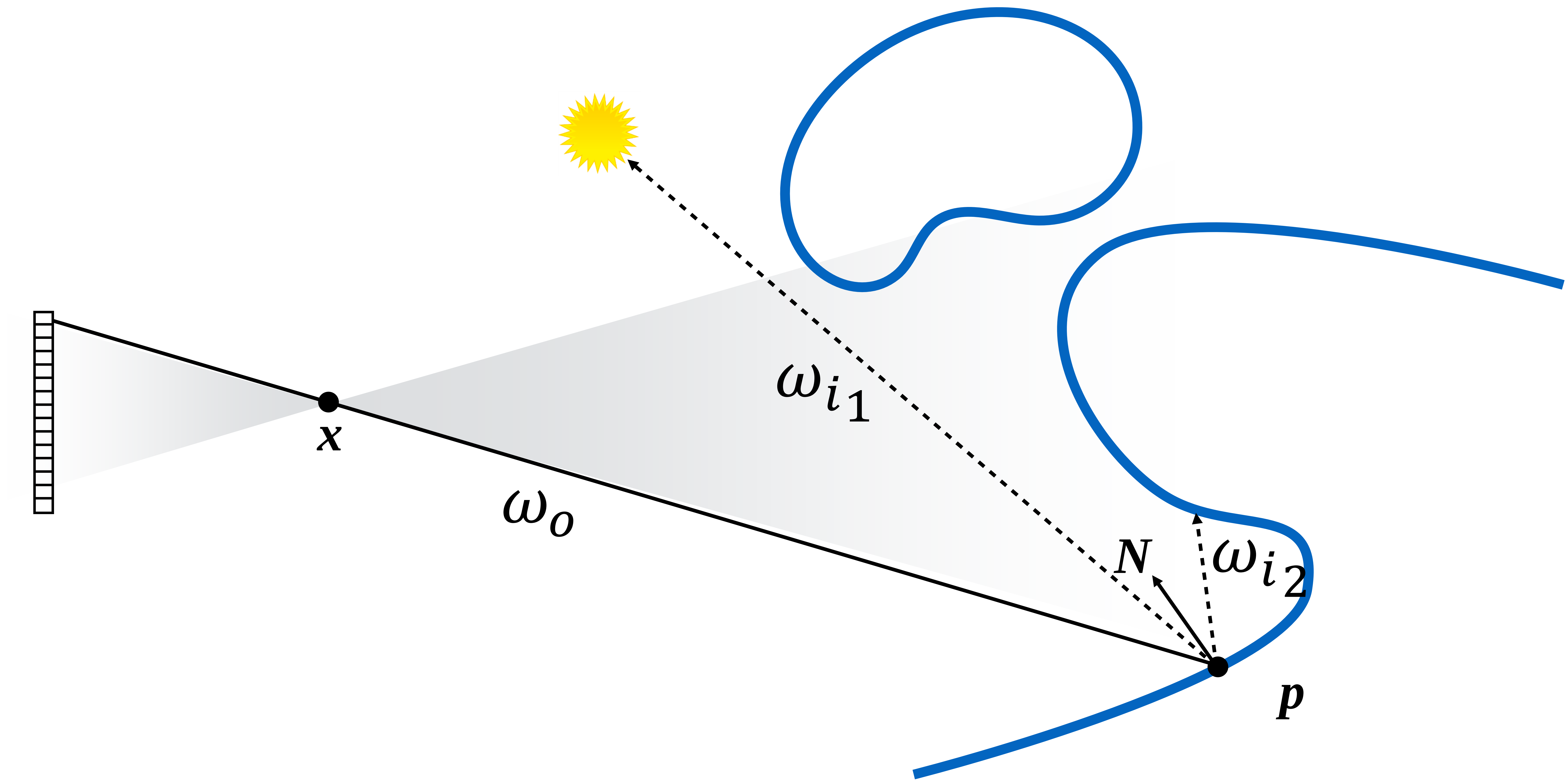
$$L_o(\mathrm{p},\omega_o) = L_e(\mathrm{p},\omega_o) + \int_{H^2} f_r(\mathrm{p},\omega_i \to \omega_o)\, L_i(\mathrm{p},\omega_i)\, \cos\theta_i \,\mathrm{d}\omega_i$$

Animation of the scene



Actual scene

Rays, rays and more rays...



Now we have an idea of what the "right answer" should be...

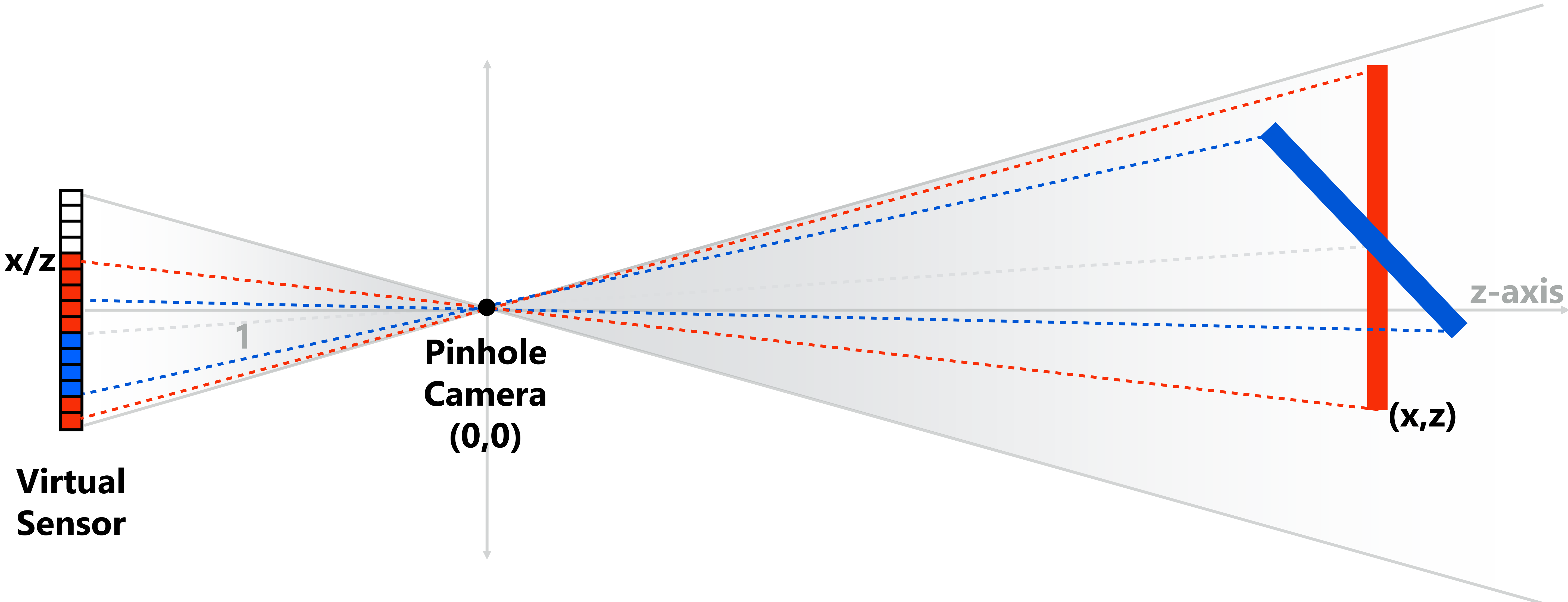
Now we have an idea of what the "right answer" should be…

But things get very complicated very quickly…

You know "everything" you need to create this image.

But it looks so "flat" ☹

Rasterization



Q: How are occlusions handled?

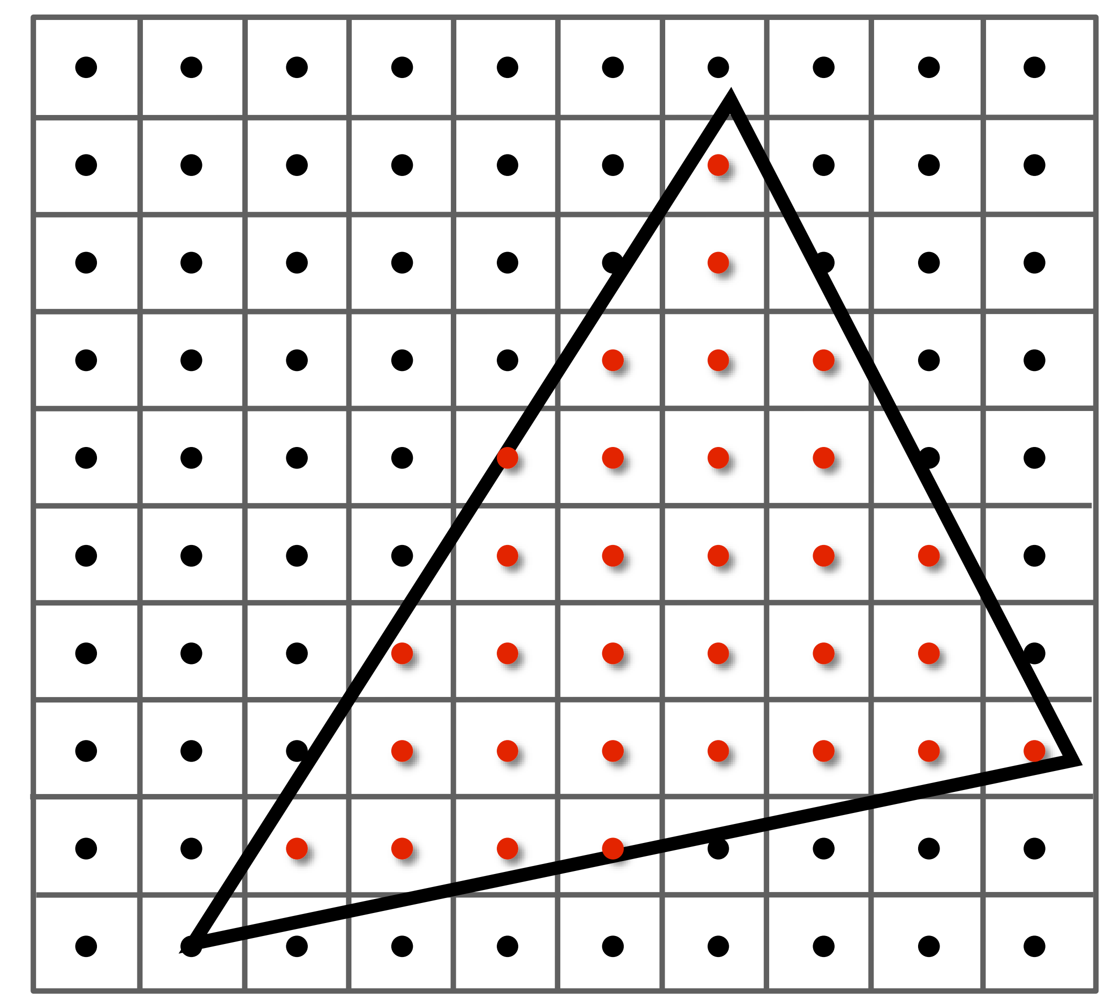
Basic rasterization algorithm

Sample = 2D point

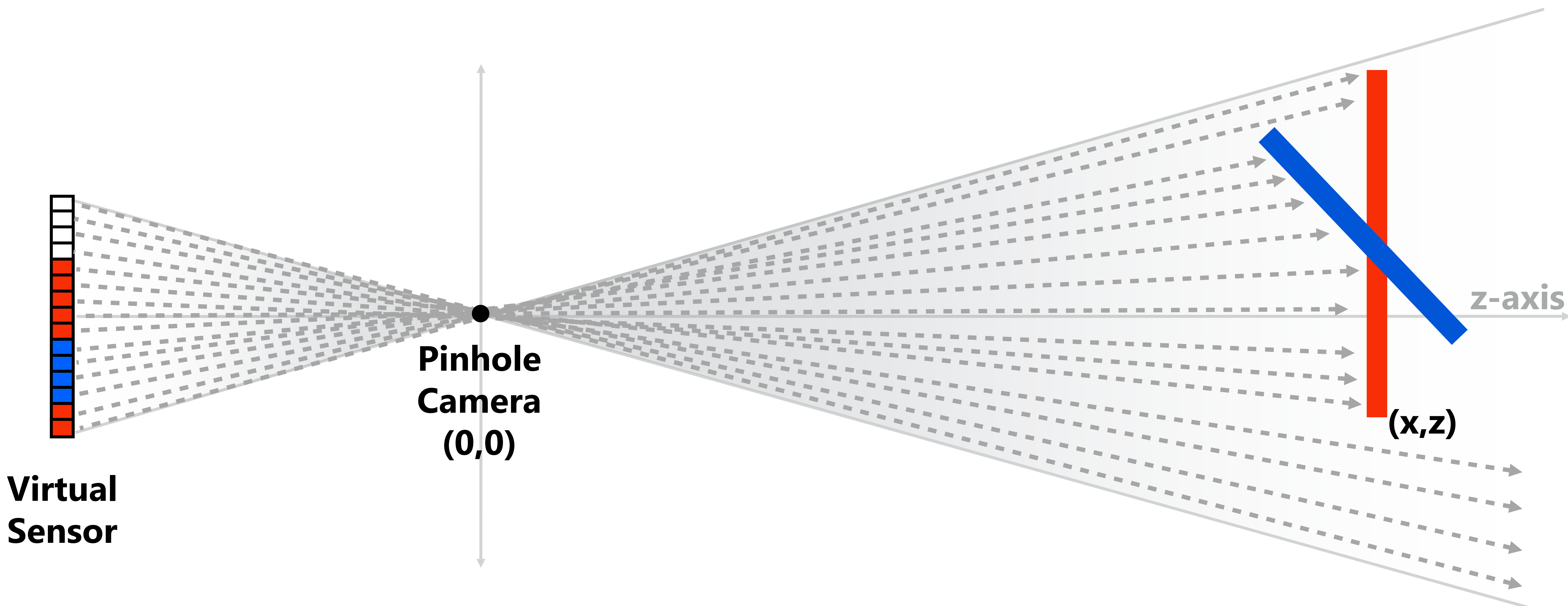
Coverage: does a projected triangle cover 2D sample point?

Occlusion: depth buffer

Finding samples is easy since they are distributed uniformly on screen.



Rendering via ray-casting



We need to compute intersections between rays and the scene

Q: How should occlusions be handled?

Basic ray casting algorithm

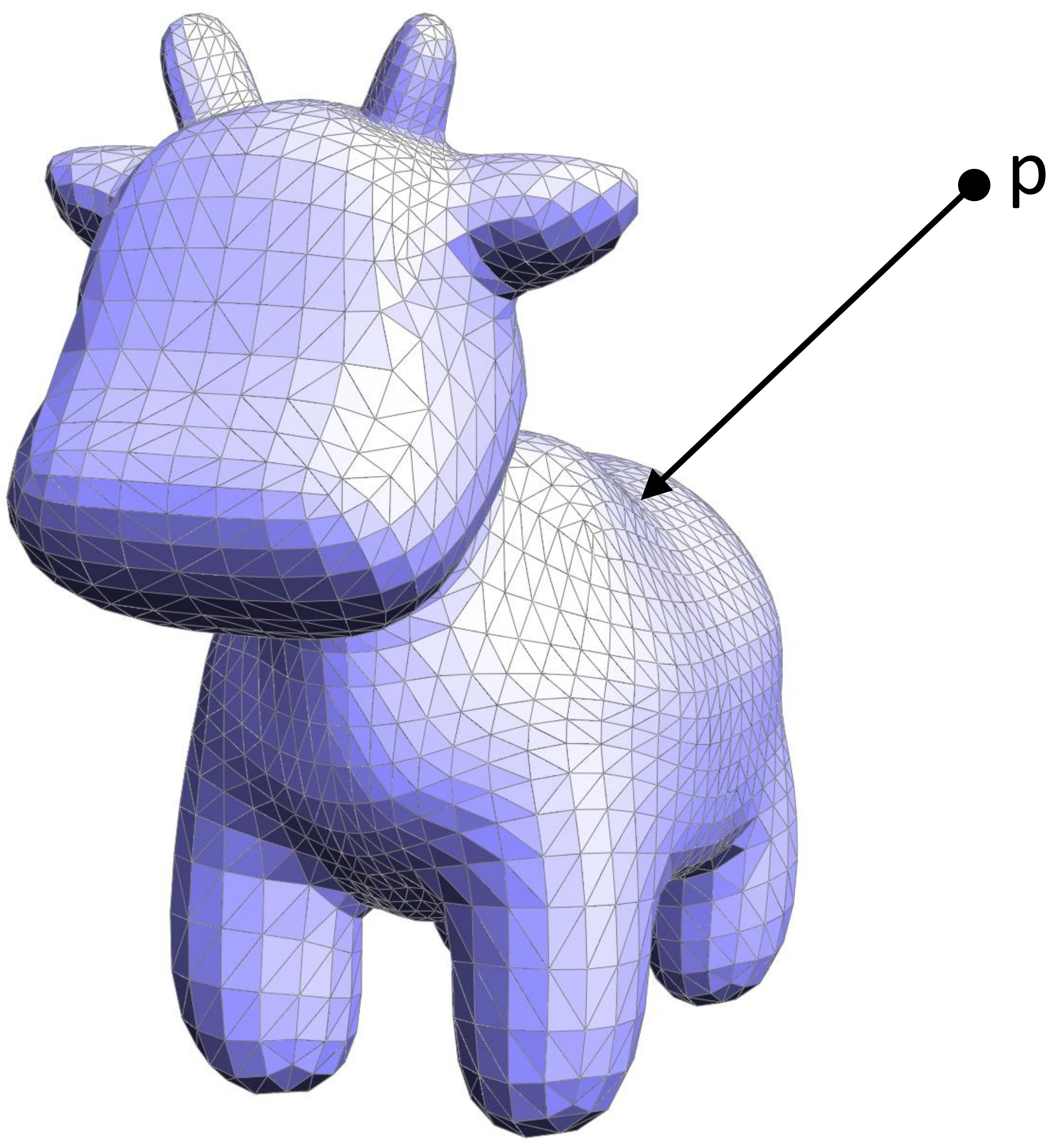
Sample = a ray in 3D

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

Occlusion: closest intersection along ray

Q: What should happen once the point hit by a ray is found?

Q: What are the main differences between rasterization and ray casting?



Rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order
  - Most processing is based on 2D primitives (3d geometry projected into screen space)
  - Store depth buffer (random access to regular structure of fixed size)

- **Ray casting:**
  - Proceeds in screen sample order
    - Never have to store depth buffer (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 (random access to irregular structure of variable size: depends on complexity and distribution of scene)

- **Conceptually, compared to rasterization approach, ray casting is just a reordering of loops + math in 3D**

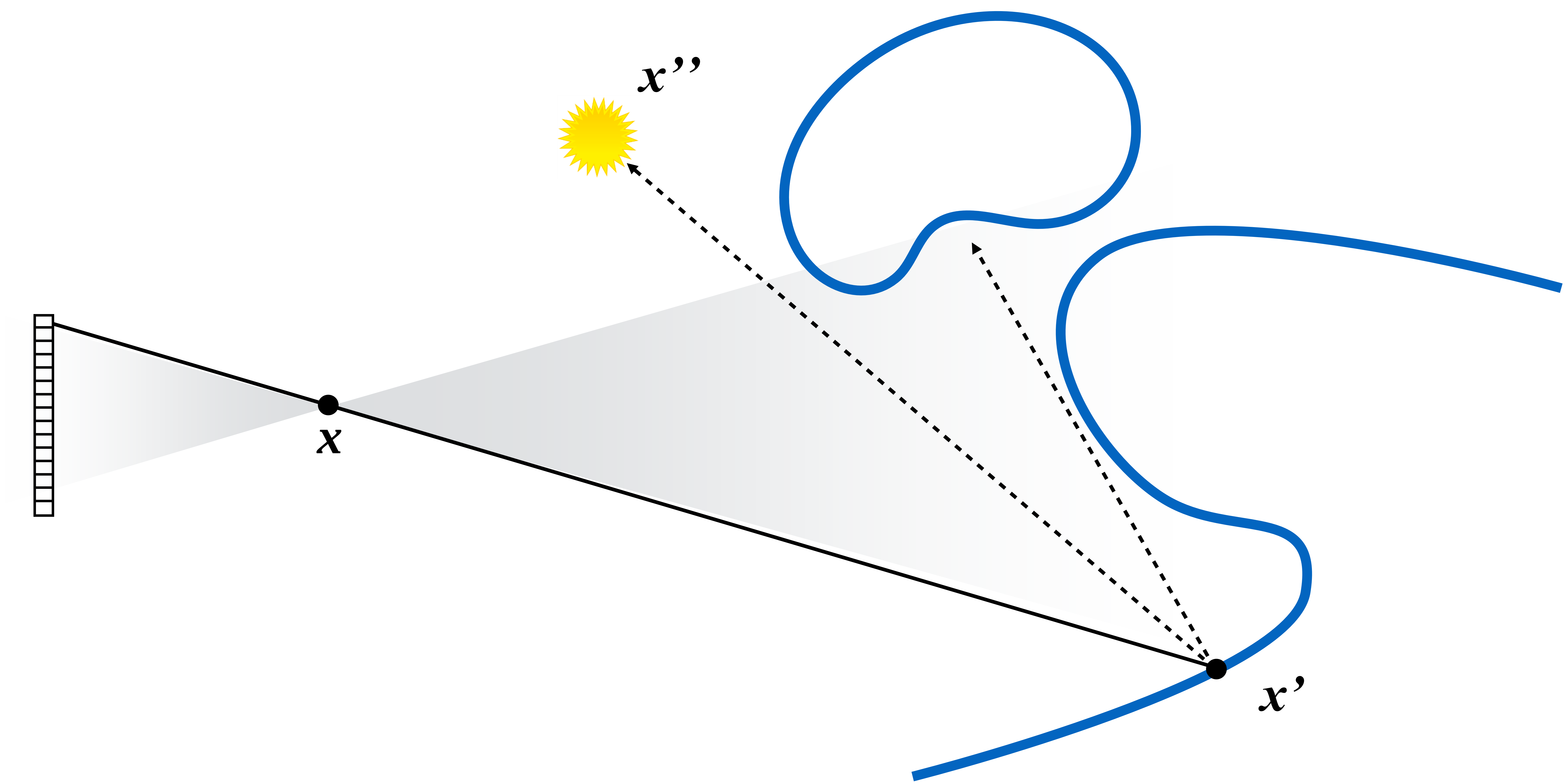
Rasterization and ray casting are two approaches for solving the same problem: determining "visibility"

Another way to think about rasterization

- **An efficient, highly-specialized algorithm for visibility queries, given rays with specific properties**
  - **Assumption 1: Rays have the same origin**
  - **Assumption 2: Rays are uniformly distributed over plane of projection (within specified field of view)**
- **Assumptions lead to significant optimization opportunities**
  - **Project triangles: reduce ray-triangle intersection to 2D point-in-polygon test**
  - **Projection to canonical view volume enables use of efficient fixed-point math, custom GPU hardware for rasterization**
- **But they also make life hard in other ways…**

Ray tracing: a more general mechanism for answering "visibility" queries

$v(x_1, x_2) = 1$ if x_1 is visible from x_2, 0 otherwise



Rasterization vs Ray tracing

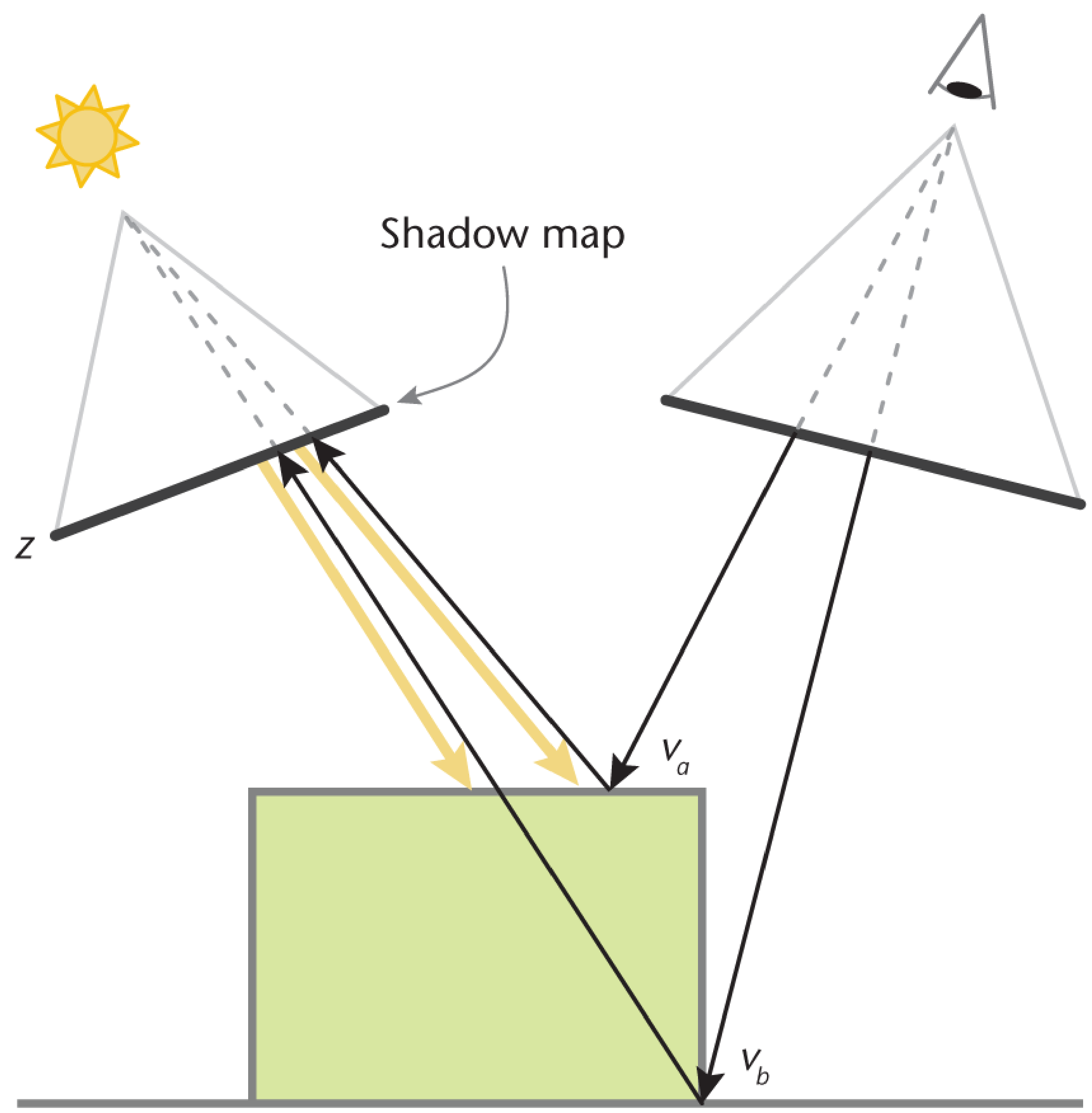
"loop over primitives"

"loop over screen pixels"

Shadows: rasterization

Shadow mapping

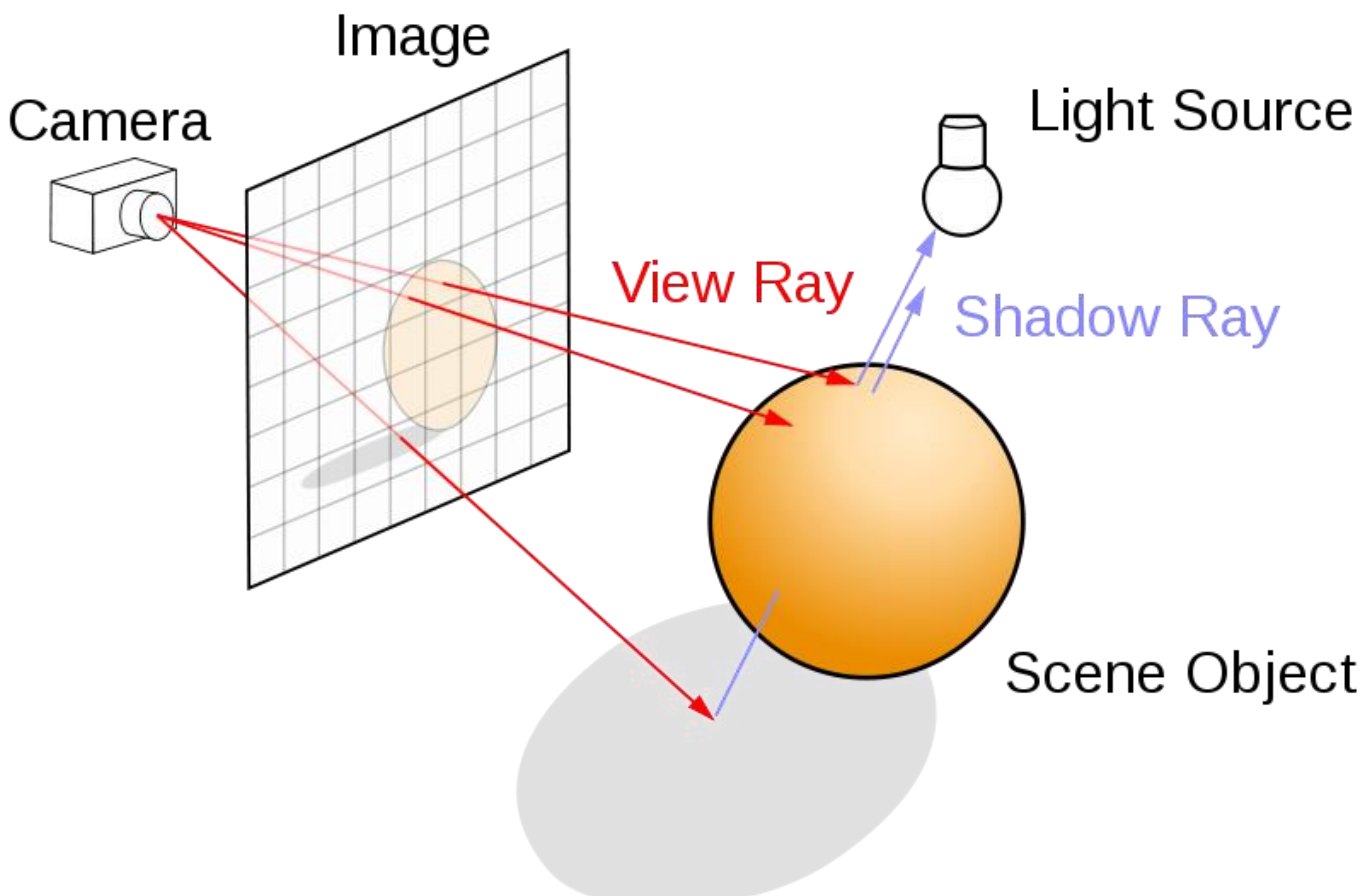
- Render scene (depth buffer only) from location of light
  - Everything "seen" from this point of view is directly lit
- Render scene from location of camera
  - Transform every screen sample to light coordinate frame and perform a depth test (fail = in shadow)



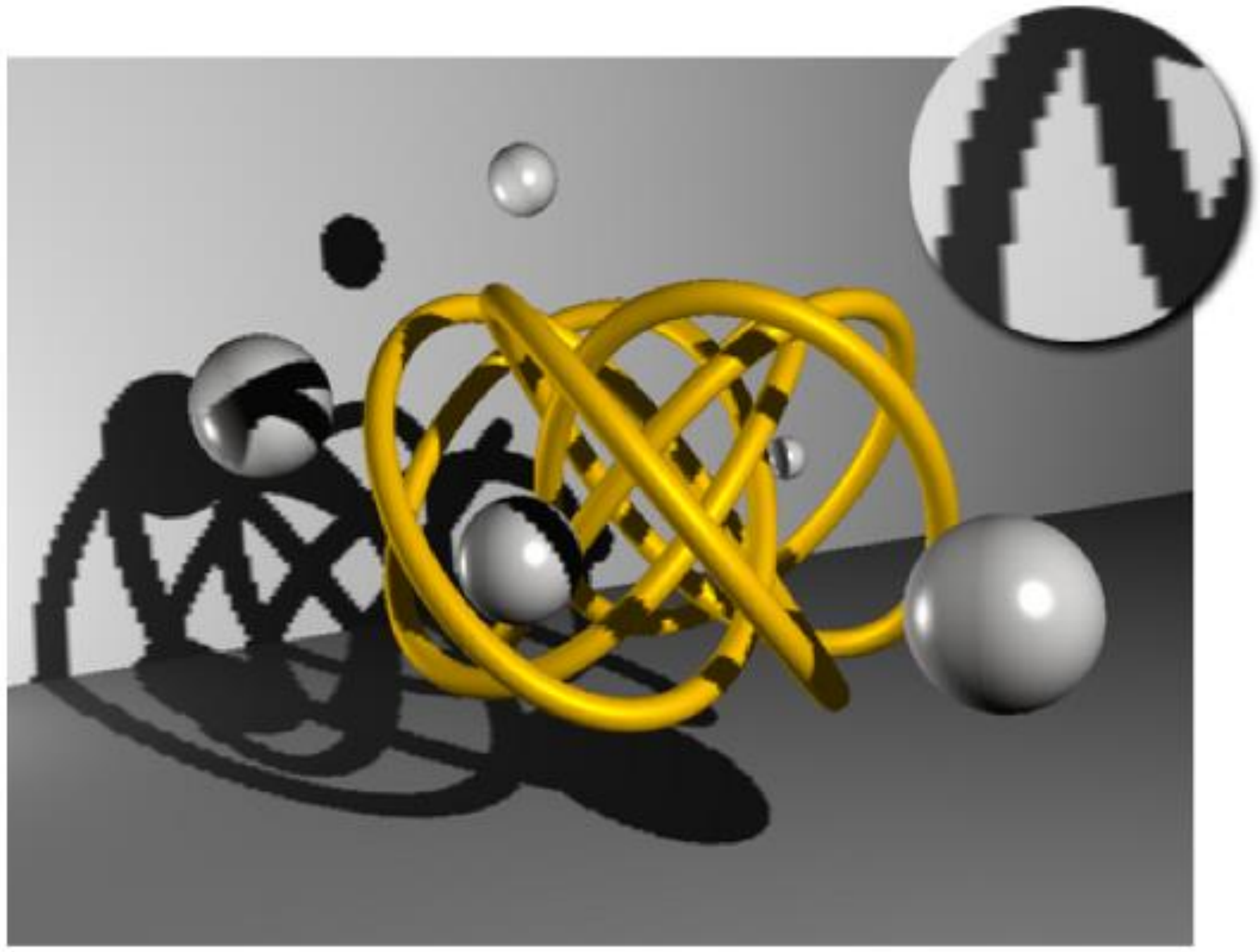
Shadows: ray tracing

Recursive ray tracing

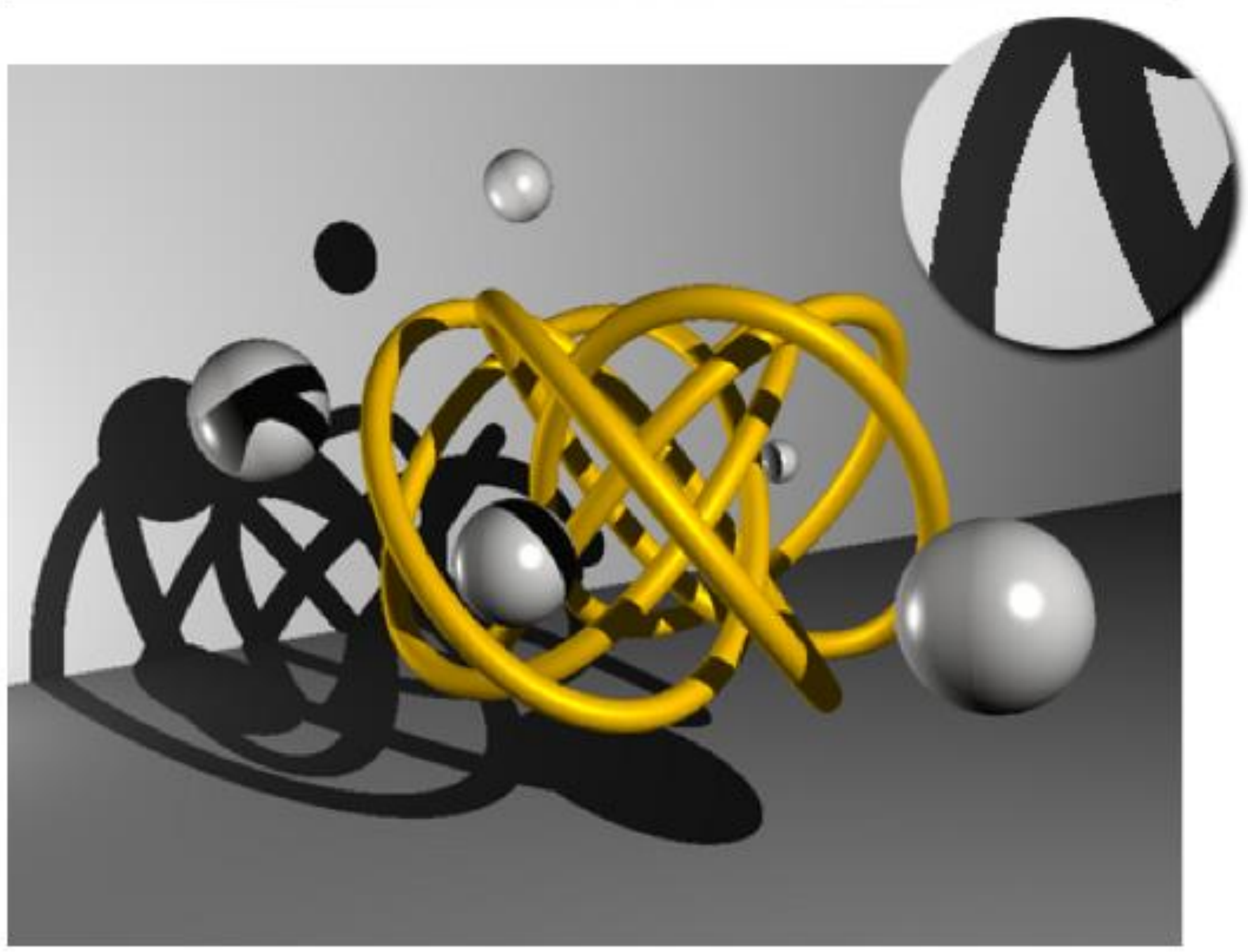
- shoot "shadow" rays towards light source from points where camera rays intersect scene
  - If unconcluded, point is directly lit by light source



Shadows: rasterization vs ray tracing



Shadows computed using shadow map (shadow map texture can lead to aliasing)



Correct hard shadows with raytracing

Q: Hard shadows? What are soft shadows?

Image credit: Johnson et al. TOG 2005

Reflections

Reflections: rasterization

Environment mapping

Place ray origin at location of reflective object, render six views.

Use camera ray reflected about surface normal to determine which texel in cube map is “hit”

Approximates appearance of reflective surface

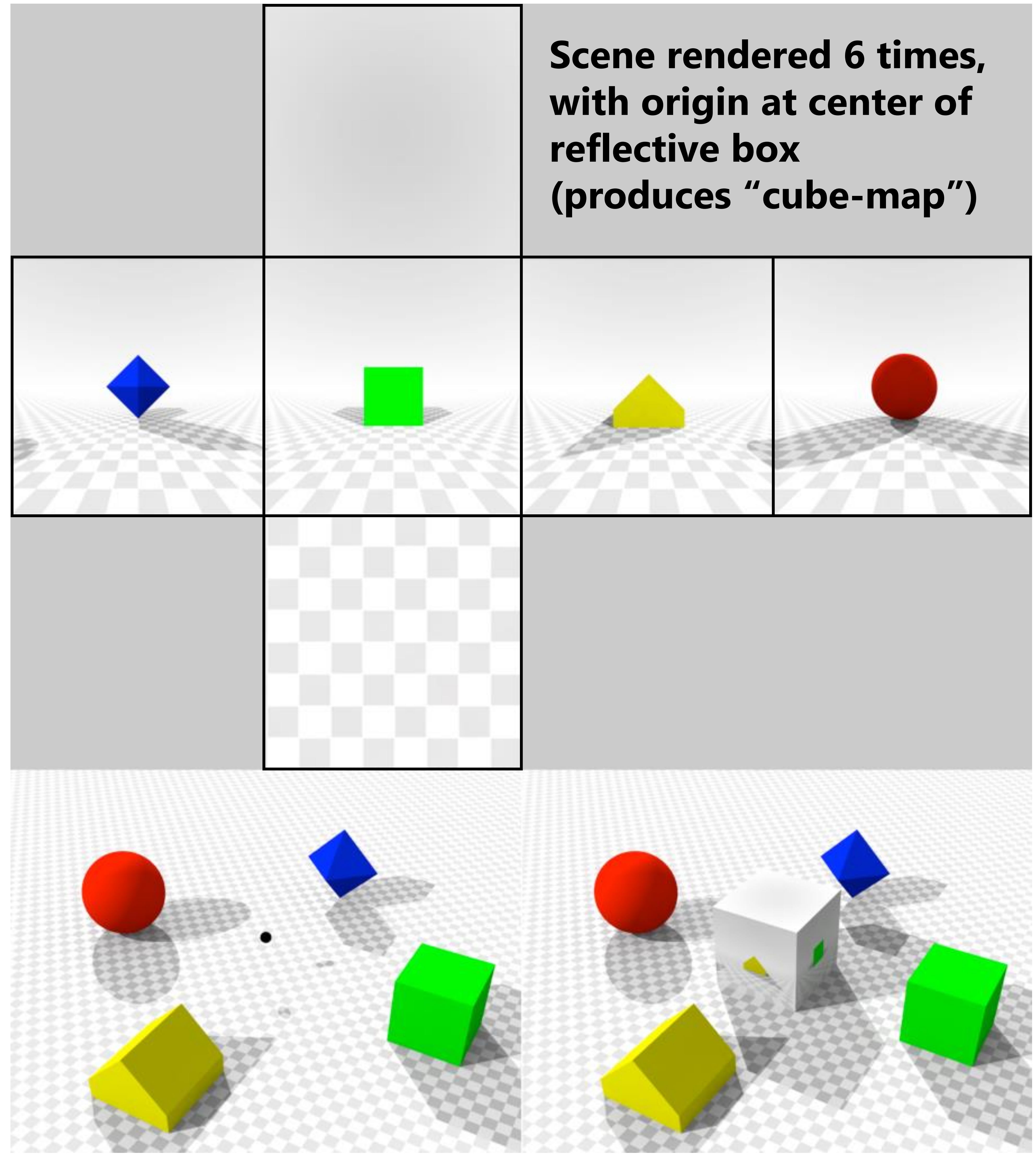
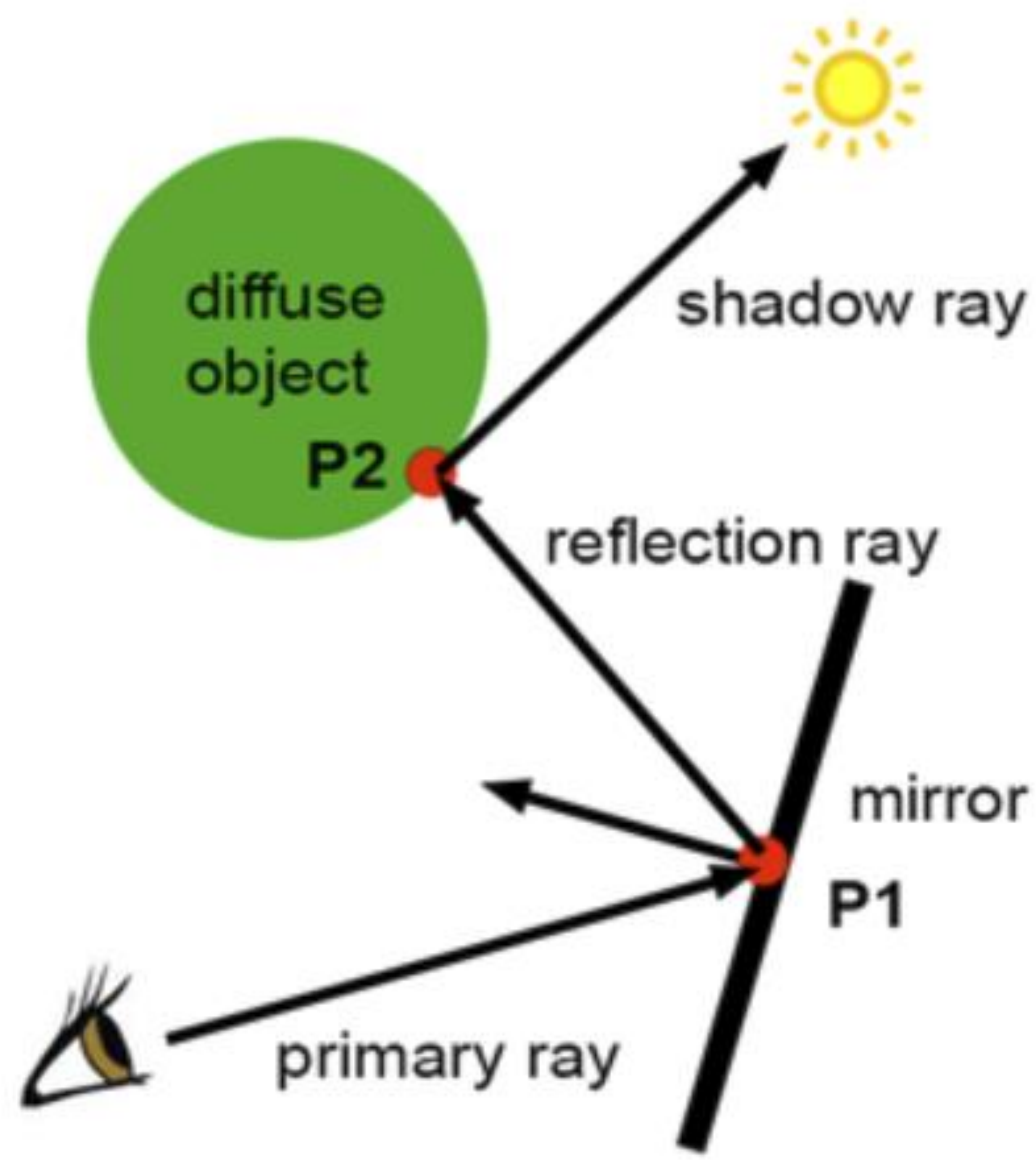


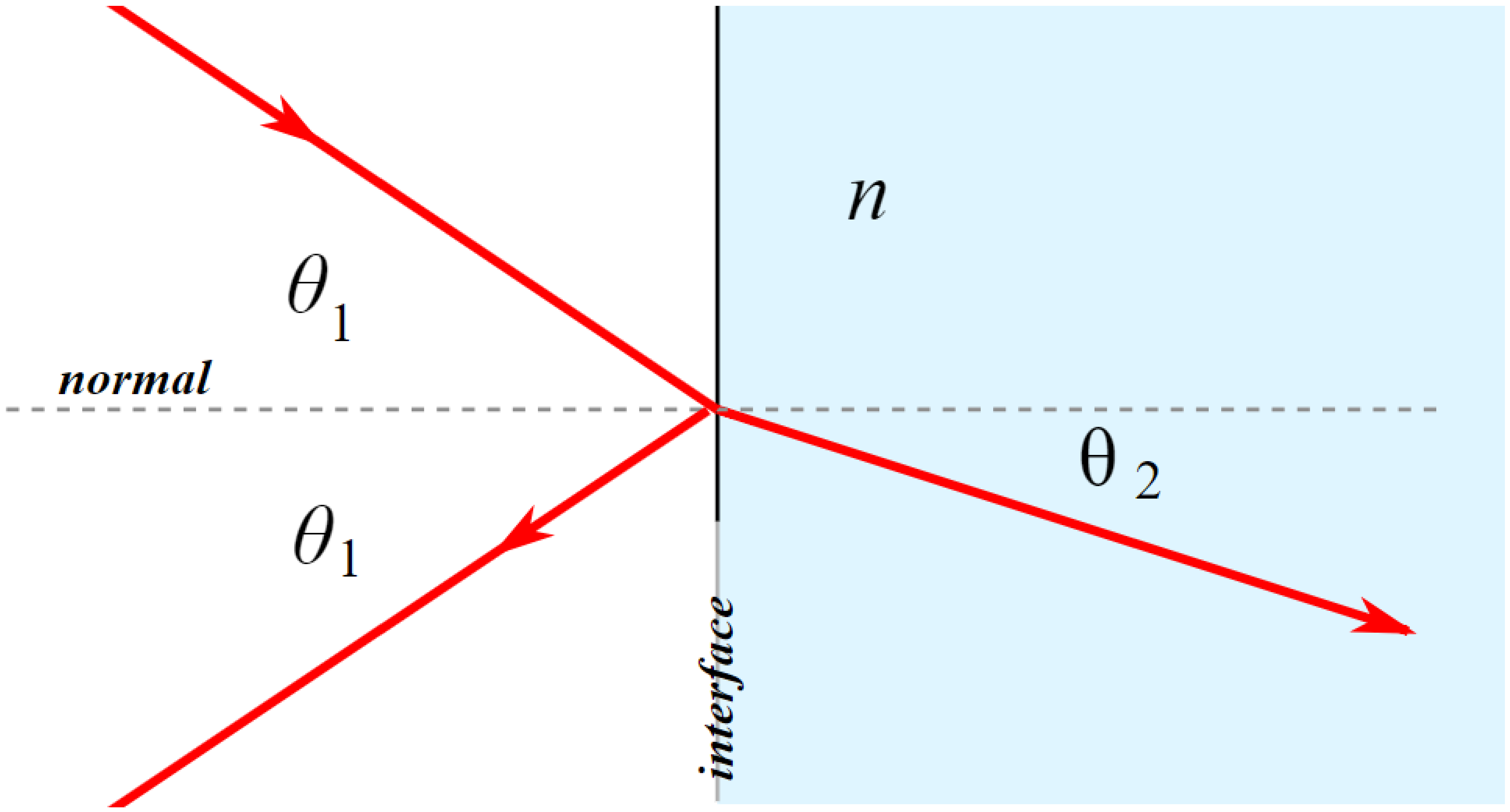
Image credit: http://en.wikipedia.org/wiki/Cube_mapping

Reflections: ray tracing

Recursive ray tracing – more secondary rays

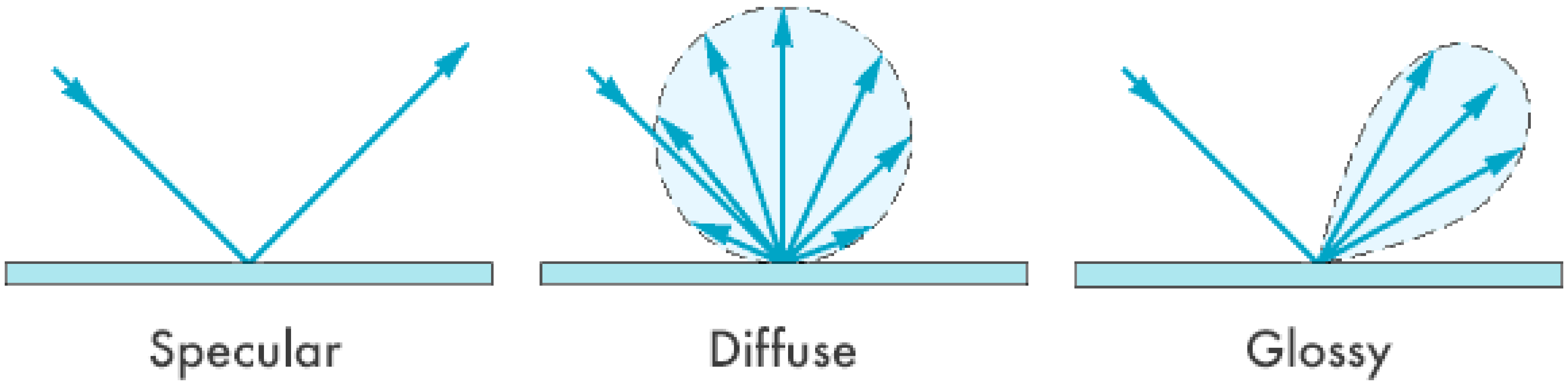


How do we reflect rays?

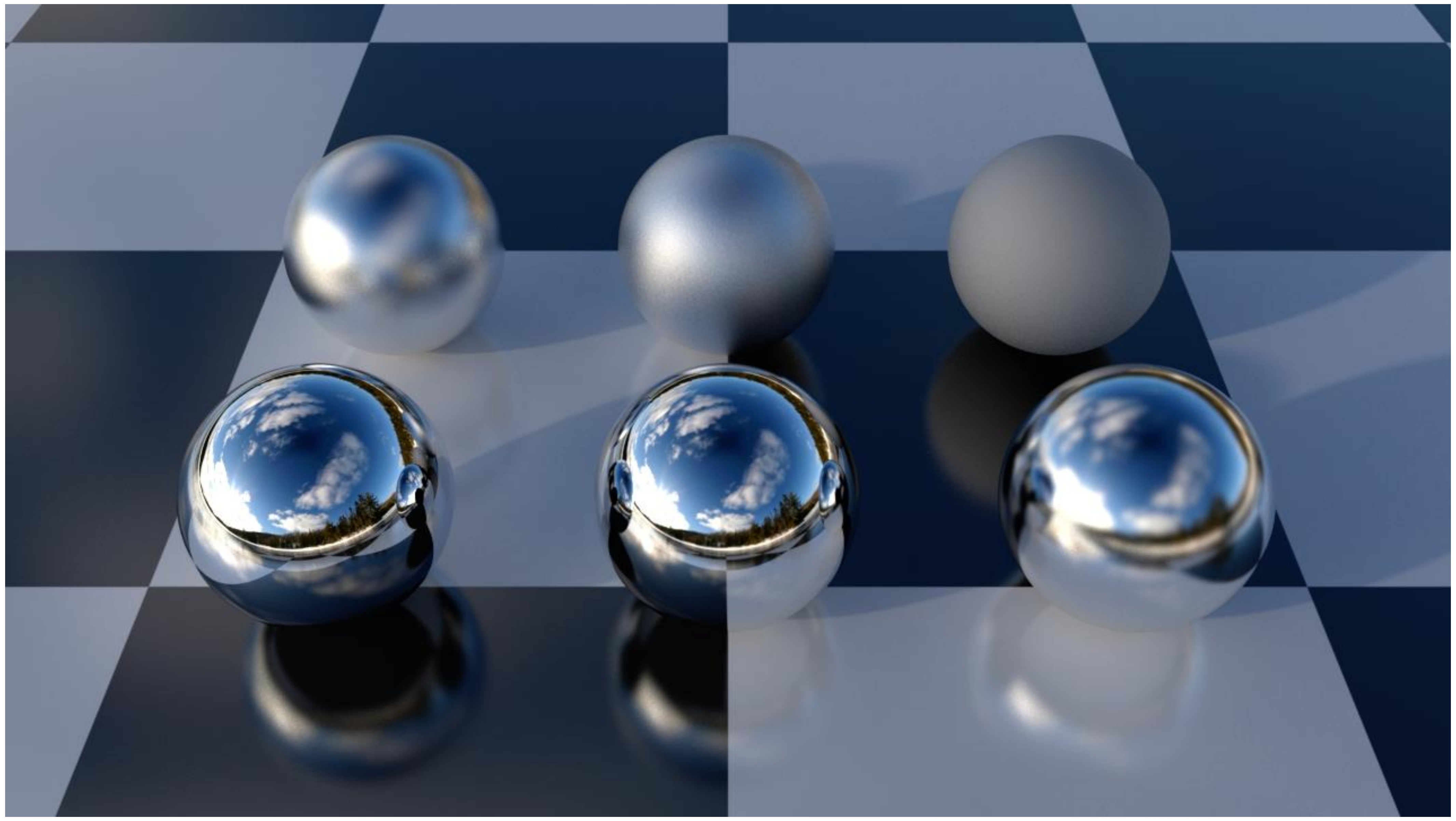


Snell's law

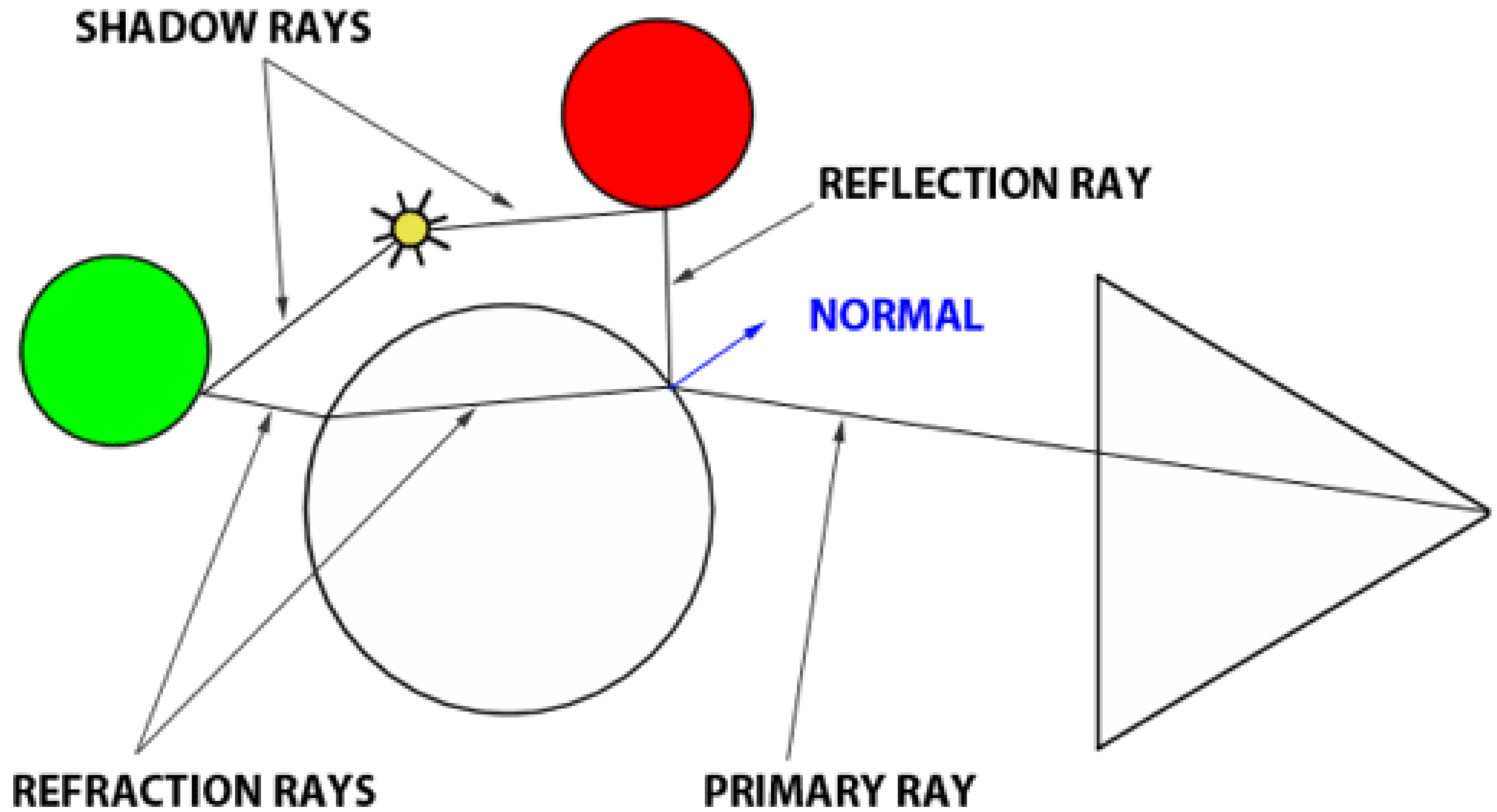
Reflections: ray tracing



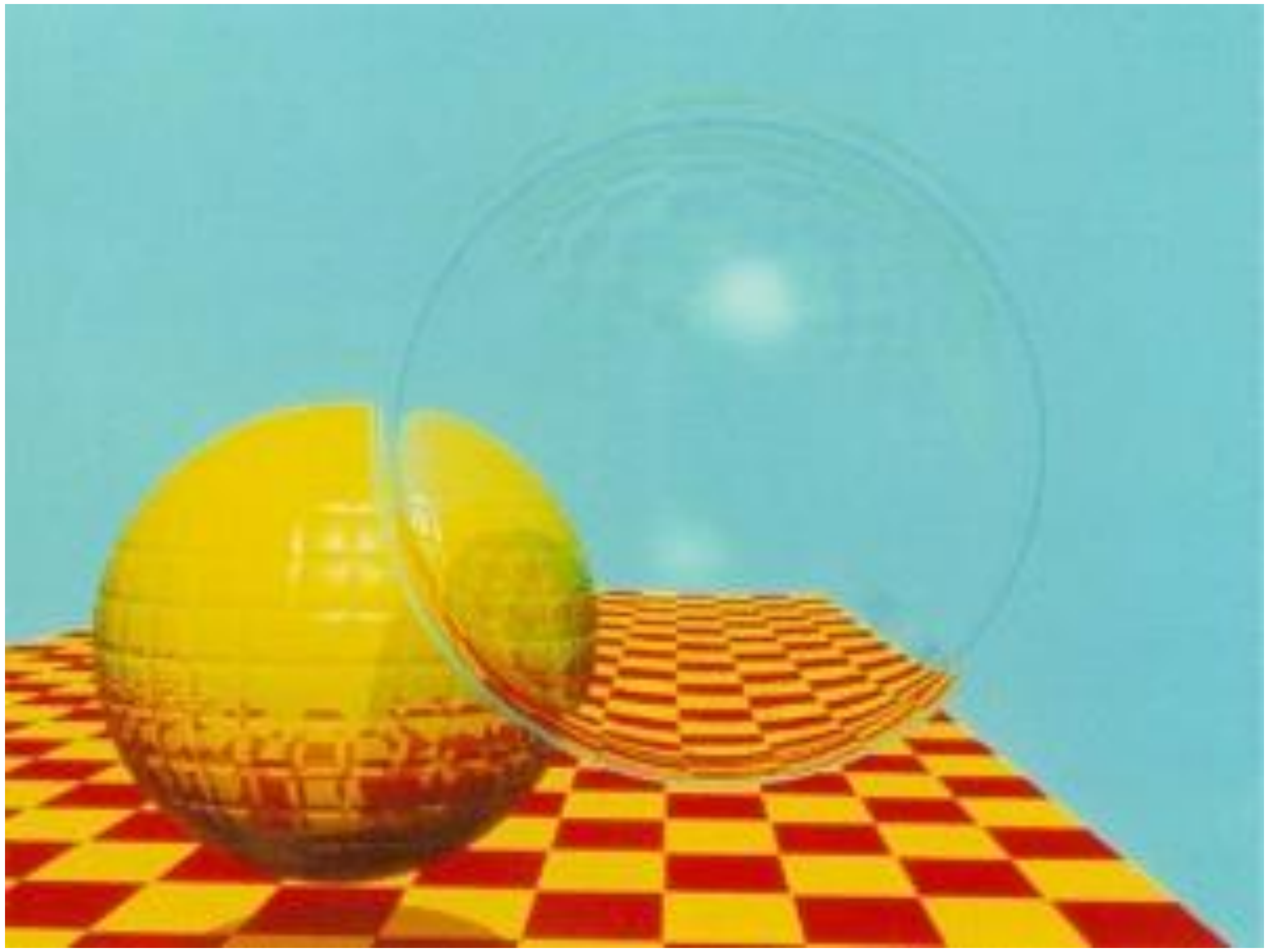
Q: How would you model appearance of a rougher glossy object?



Shadows, Reflections, Refractions: recursive ray tracing



Ray tracing history

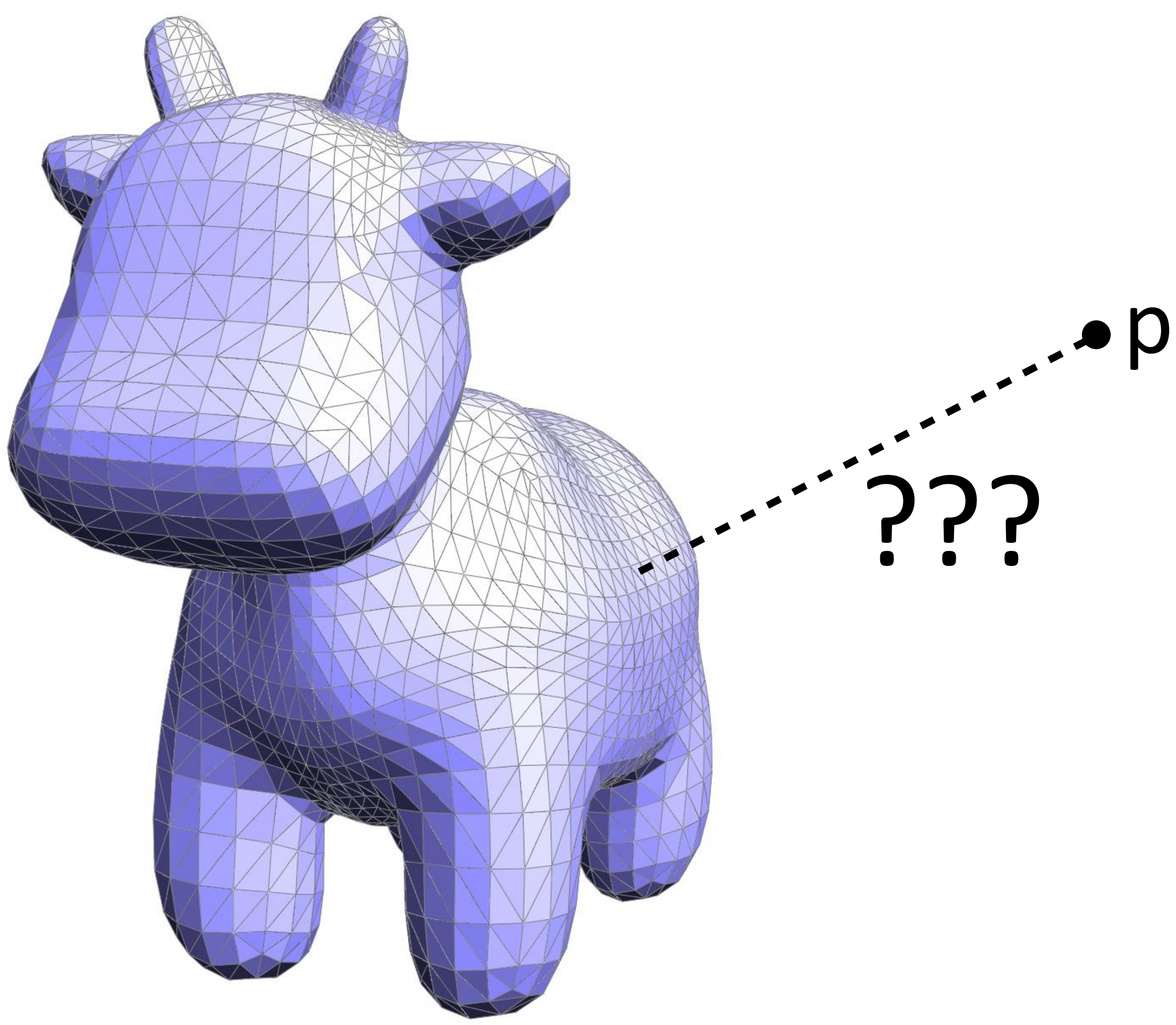


"An improved illumination model for shaded display" by T. Whitted, CACM 1980

It’s all about ray-scene intersections

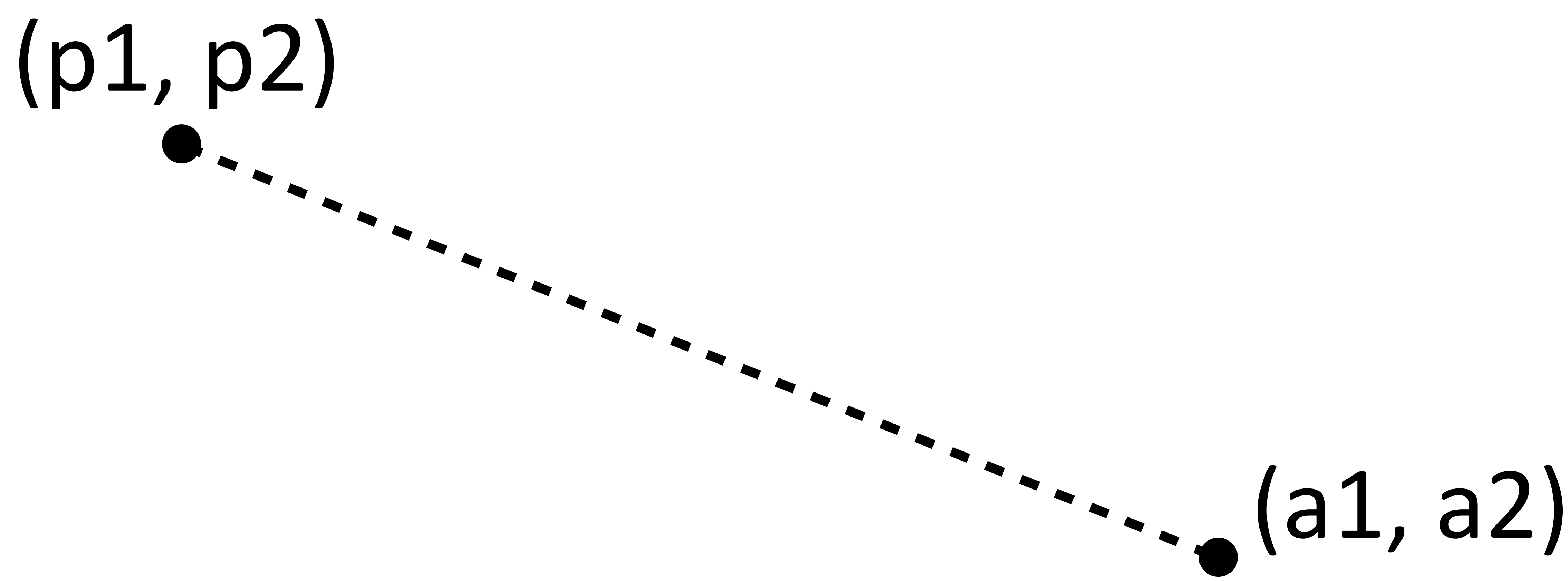
Geometric Queries

- Q: Given a point, in space (e.g., a new sample point), how do we find the closest point on a given surface?
- Q: Does implicit/explicit representation of geometry make this easier?
- Q: How do we find the distance to a single triangle? Or the point on the triangle that is hit by the ray? How about an entire 3D object? Or an entire virtual world?
- So many questions!



Warm up: closest point on point

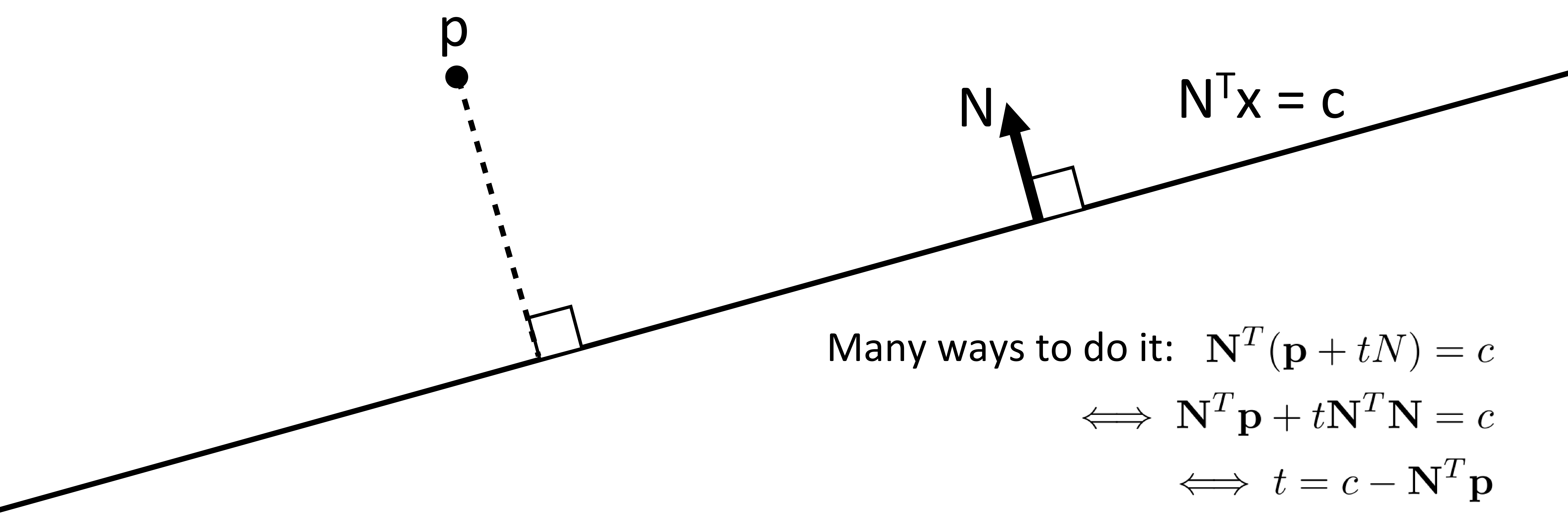
- **Goal is to find the point on a mesh closest to a given point.**
- ***Much* simpler question: given a query point (p1,p2), how do we find the closest point on the point (a1,a2)?**



Bonus question: what's the distance?

Slightly harder: closest point on 2D line

- **Now suppose I have a 2D line $N^T x = c$, where N is the unit normal**
- **How do I find the point closest to my query point p?**



Many ways to do it: $\mathbf{N}^T(\mathbf{p} + tN) = c$

$$\iff \mathbf{N}^T\mathbf{p} + t\mathbf{N}^T\mathbf{N} = c$$

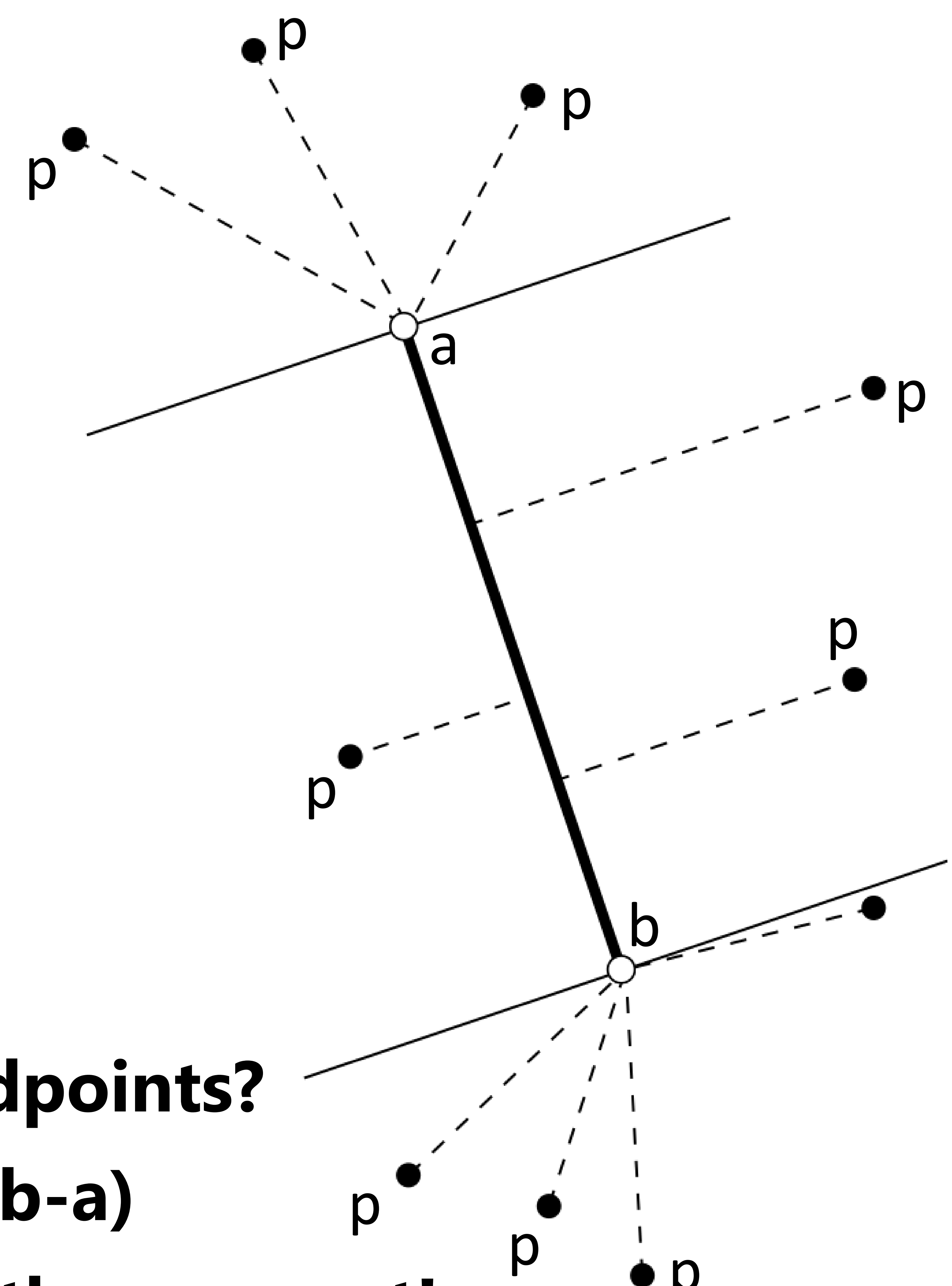
$$\iff t = c - \mathbf{N}^T\mathbf{p}$$

$$\Rightarrow \mathbf{p} + t\mathbf{N} = \boxed{\mathbf{p} + (c - \mathbf{N}^T\mathbf{p})\mathbf{N}}$$

Q: how about closest point to a line in 3D?

Harder: closest point on line segment

- **Two cases: endpoint or interior**
- **Already have basic components:**
  - **point-to-point**
  - **point-to-line**
- **Algorithm?**
  - **find closest point on line**
  - **check if it's between endpoints**
  - **if not, take closest endpoint**
- **How do we know if it's between endpoints?**
  - **write closest point on line as a+t(b-a)**
  - **if t is between 0 and 1, it's inside the segment!**



Great…

- **But for ray casting we want to know if a ray <u>hits</u> objects in the scene…**

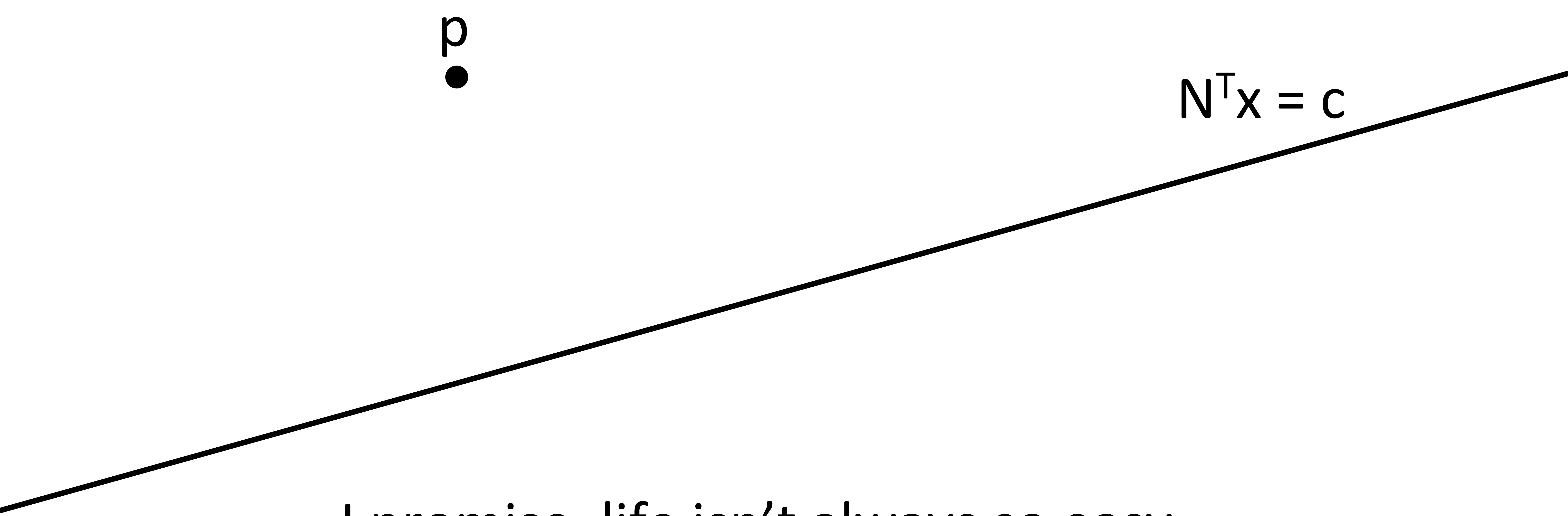
Warm up: point-point intersection

- **Q: How do we know if p intersects a?**
- **A: …check if they're the same point!**

Sadly, life is not always so easy.

Slightly harder: point-line intersection

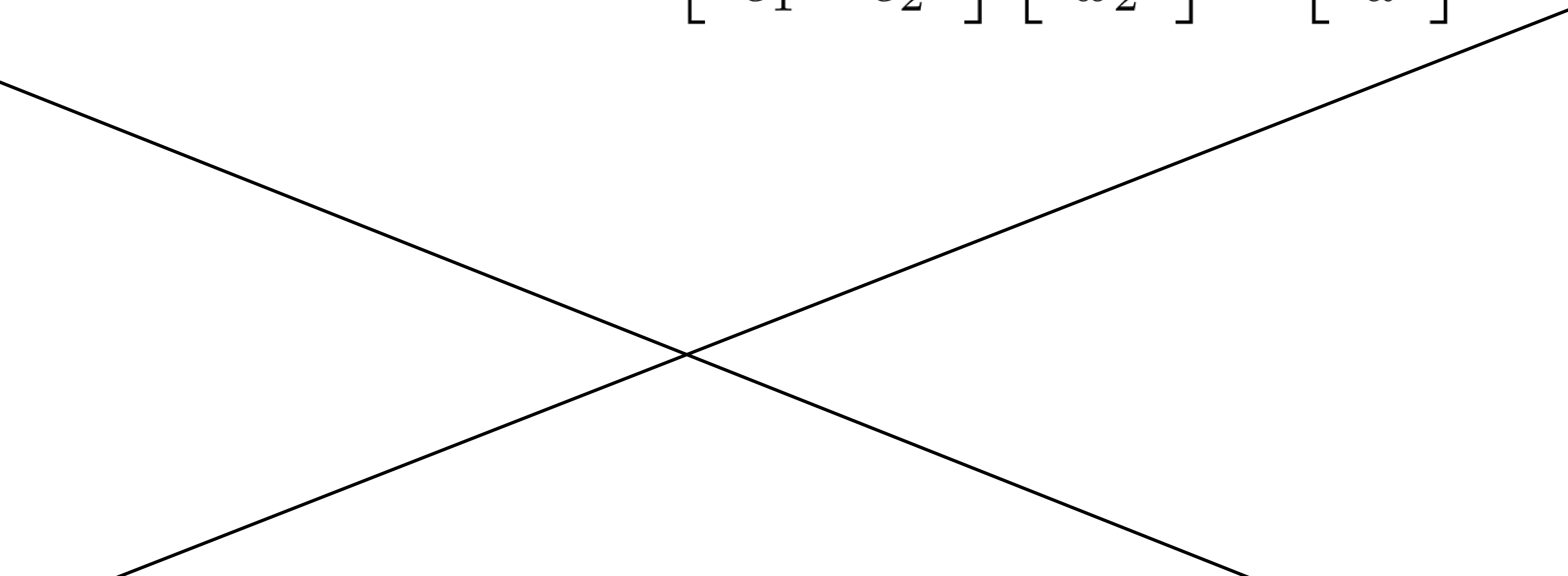
- **Q: How do we know if a point intersects a given line?**
- **A: ...plug it into the line equation!**



I promise, life isn't always so easy.

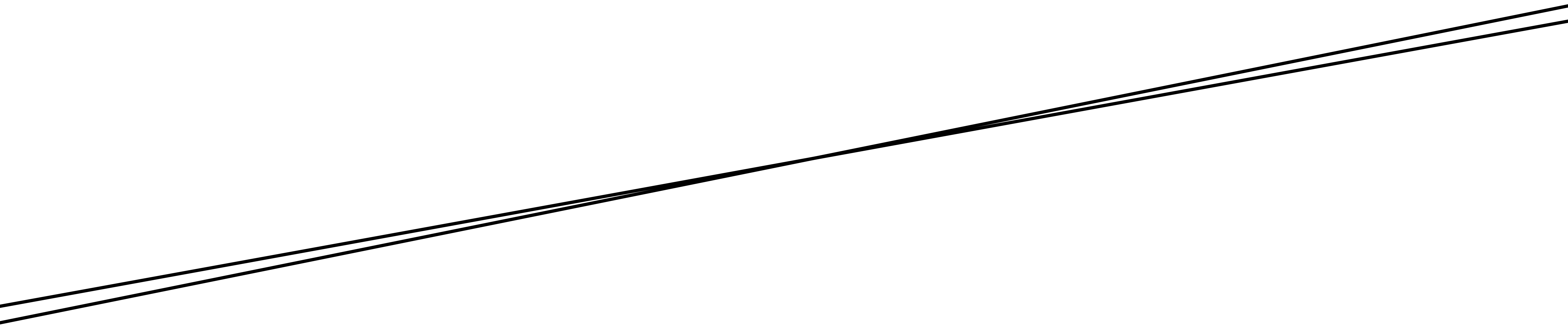
Finally a bit more interesting: line-line intersection

- **Two lines: ax=b and cx=d**
- **Q: How do we find the intersection?**
- **A: See if there is a simultaneous solution**
- **Leads to linear system:** $\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$



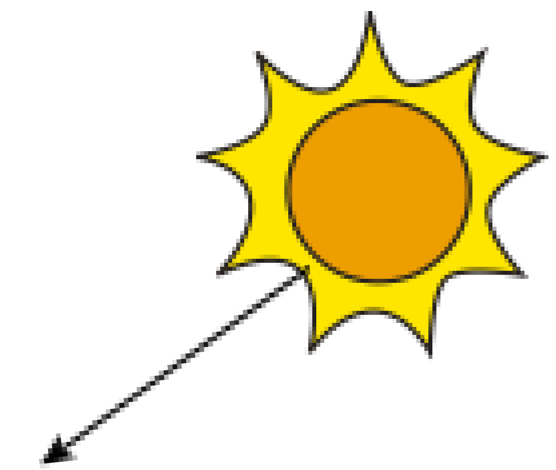
Degenerate line-line intersection?

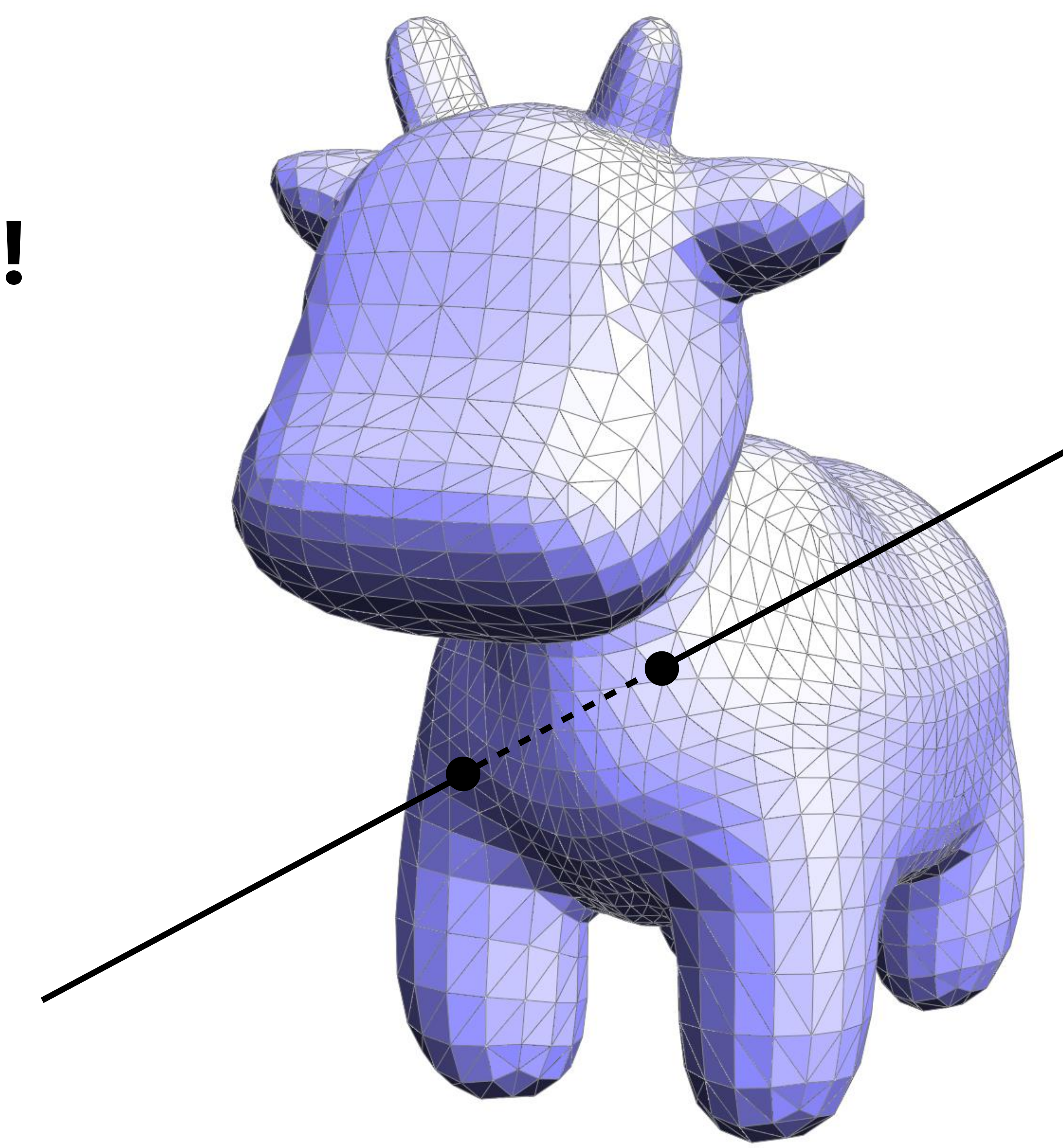
- What if lines are almost parallel?
- Small change in normal can lead to big change in intersection!
- Instability very common, very important in dealing with geometric queries. Demands special care (e.g., analysis of matrix).



Ray-mesh intersection

- A "ray" is an oriented line starting at a point
- Want to know where a ray pierces a surface
- Why?
  - RENDERING: visibility, ray tracing
  - SIMULATION: collision detection
  - INTERACTION: mouse picking
- Might pierce surface in many places!





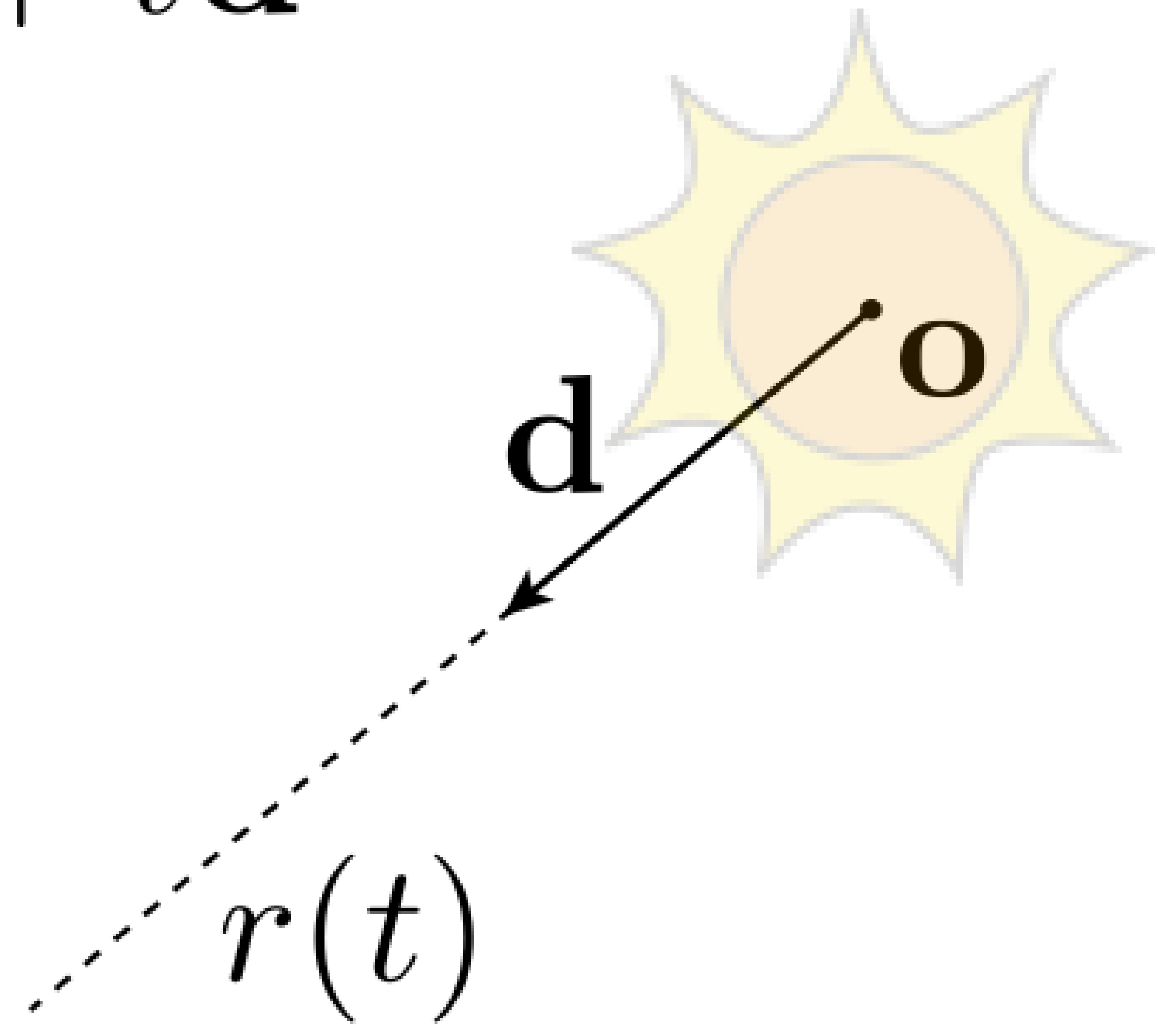
Ray: parametric equation

origin

unit direction

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

point along ray



Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points x such that f(x) = 0
- How do we find points where a ray intersects this surface?
  - we know all points along the ray: r(t) = o + td
  - replace “x” with “r”, solve for t
- Example: unit sphere

quadratic formula:

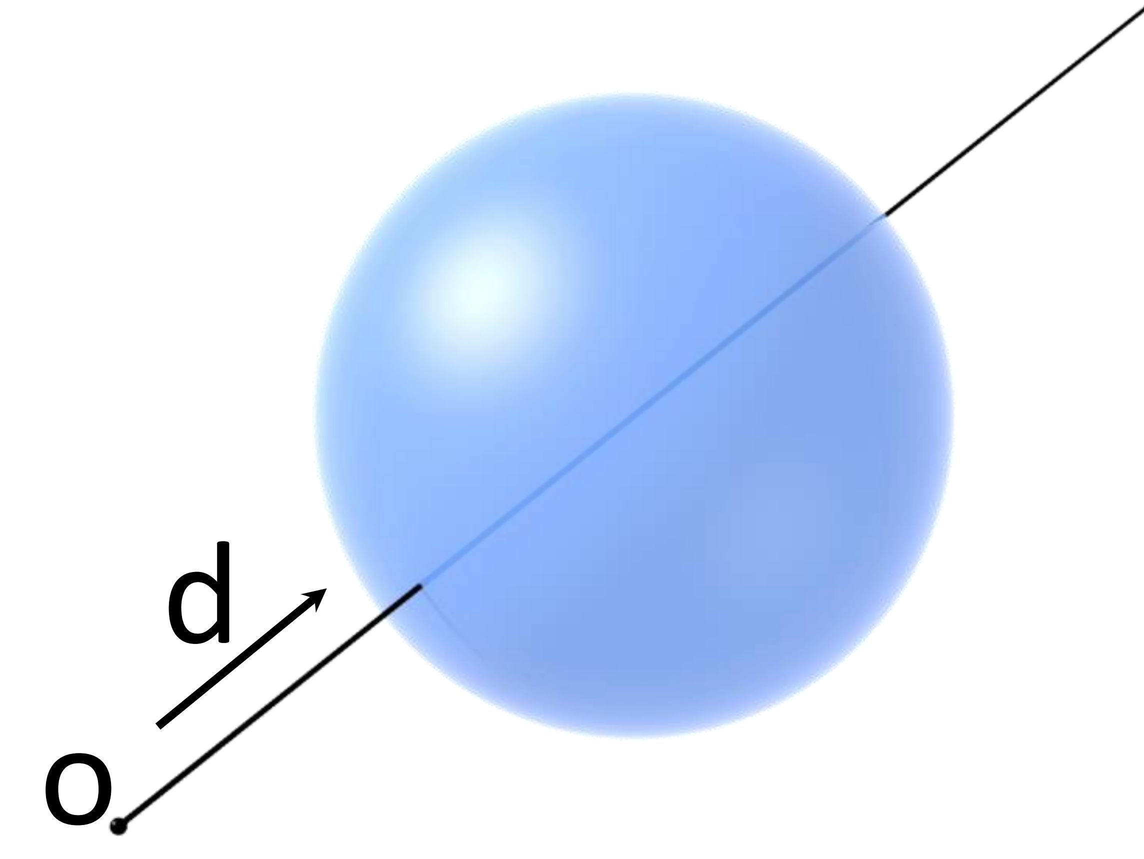
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$f(\mathbf{x}) = |\mathbf{x}|^2 - 1$$

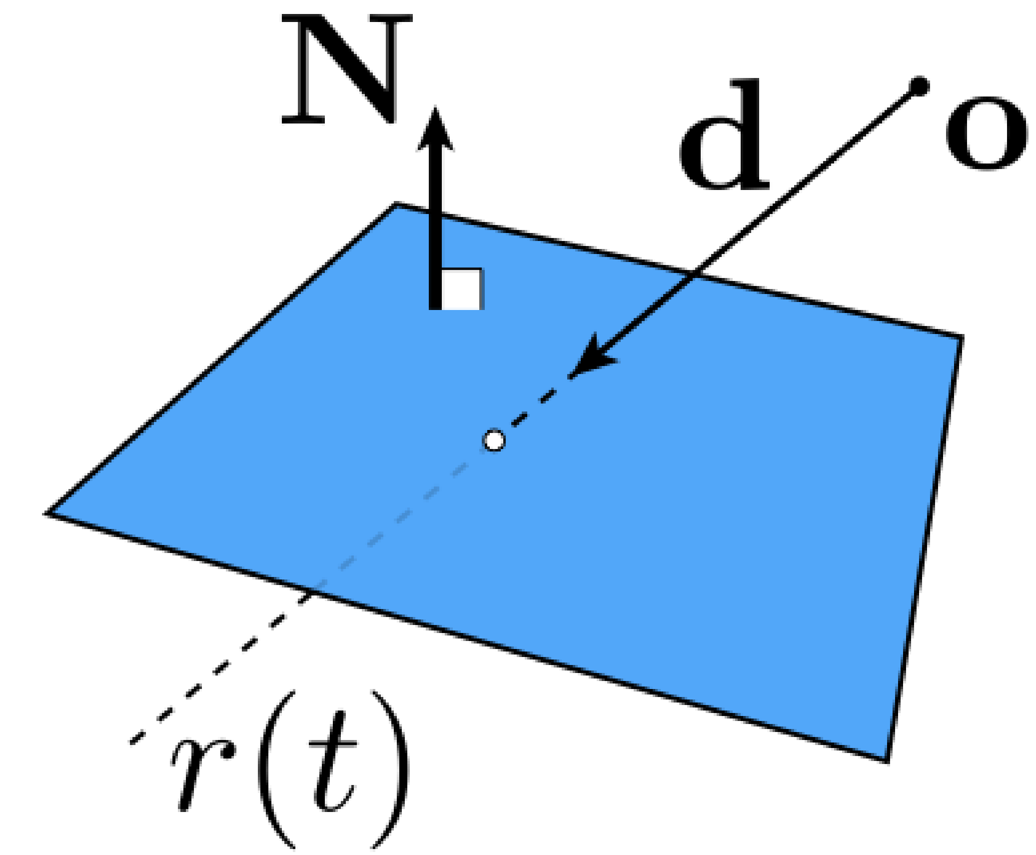
$$\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

$$\underbrace{|\mathbf{d}|^2}_{a} t^2 + \underbrace{2(\mathbf{o}\cdot\mathbf{d})}_{b} t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

$$t = \boxed{-\mathbf{o}\cdot\mathbf{d} \pm \sqrt{(\mathbf{o}\cdot\mathbf{d})^2 - |\mathbf{o}|^2 + 1}}$$



Ray-plane intersection



- Suppose we have a plane $N^T x = c$
- How do we find intersection with ray $r(t) = o + td$?
- *A*gain, replace point x with the ray equation:

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

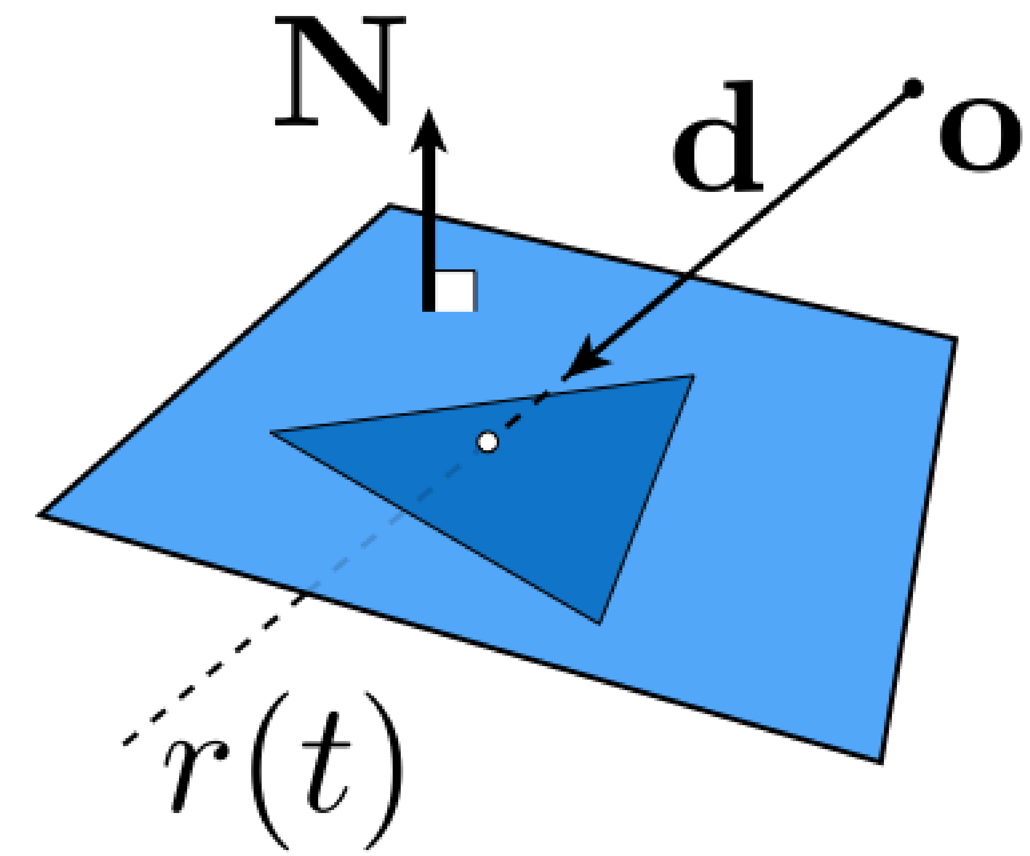
- Solve for t:

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

- And plug t back into ray equation:

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

Ray-triangle intersection



- Triangle is in a plane...
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Why not compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point in triangle

Ray-triangle intersection

Parameterize triangle given by vertices $\mathbf{p_0}, \mathbf{p_1}, \mathbf{p_2}$ using barycentric coordinates

$$f(u,v) = (1-u-v)\mathbf{p_0} + u\mathbf{p_1} + v\mathbf{p_2}$$

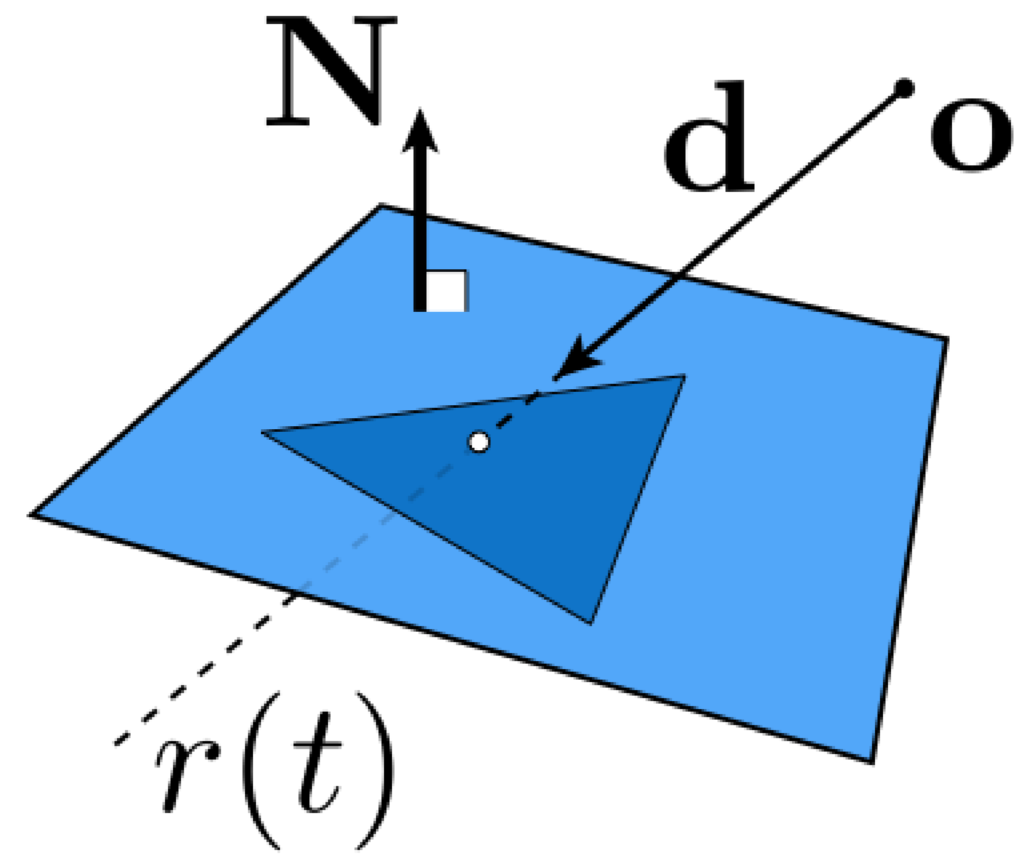
Plug parametric ray equation directly into equation for points on triangle:

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

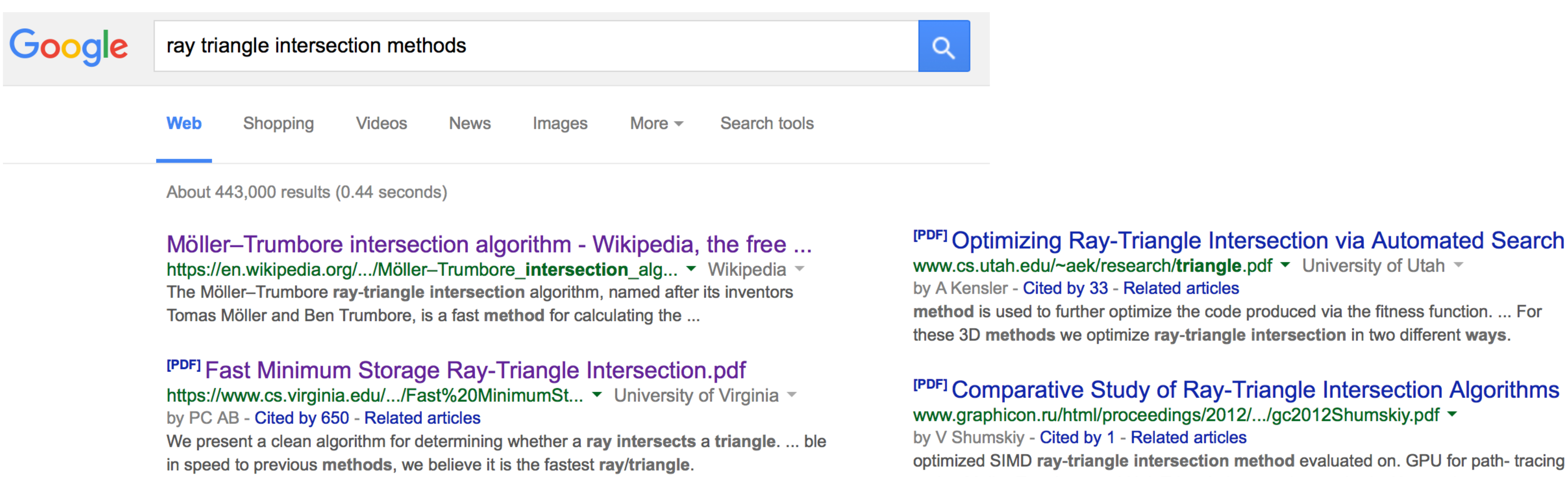
Solve for u, v, t:

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

Ray-triangle intersection



- Triangle is in a plane…
  - Compute ray-plane intersection
  - Q: What do we do now?
  - A: Why not compute barycentric coordinates of hit point?
  - If barycentric coordinates are all positive, point in triangle
  - Not much more to say!
- Actually, a *lot* more to say…



Core methods for ray-primitive queries

Given primitive p:

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

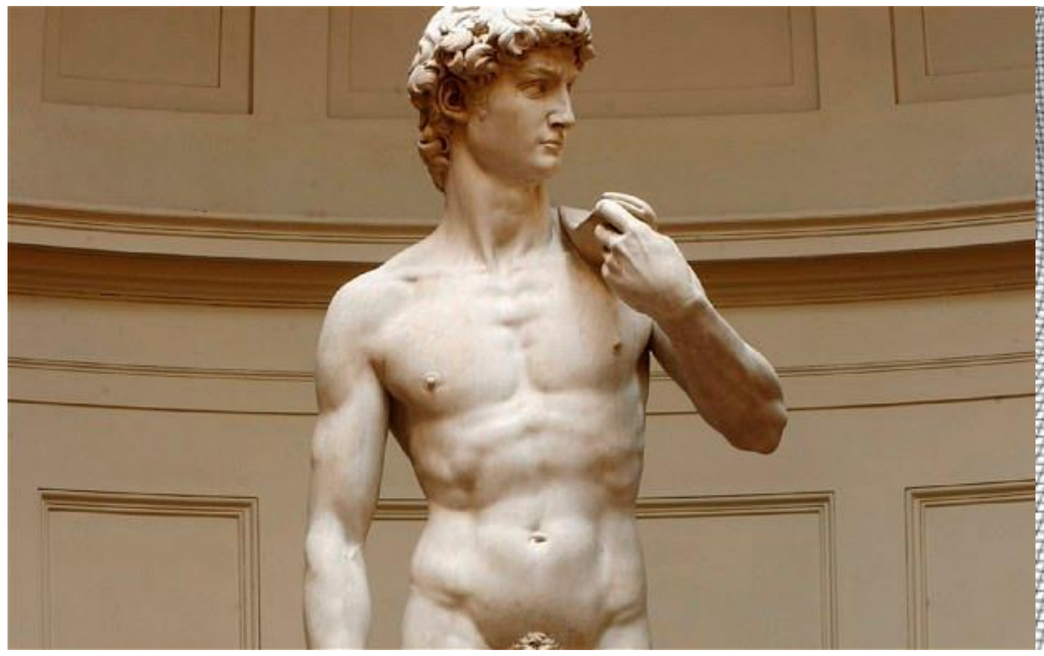
Ray-scene intersection

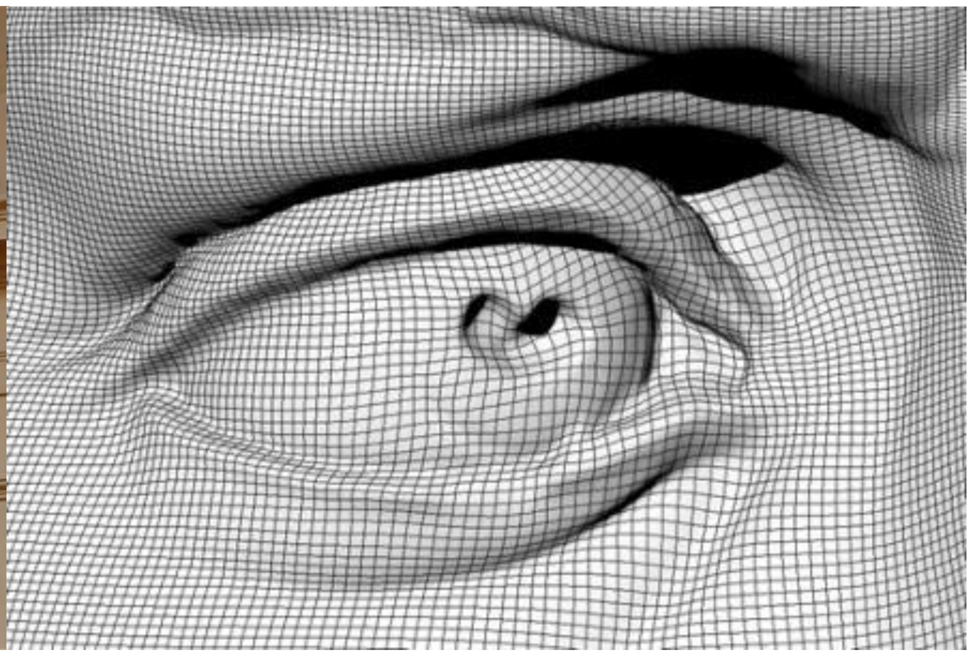
Given a scene defined by a set of *N* primitives and a ray *r*, find the closest point of intersection of *r* with the scene

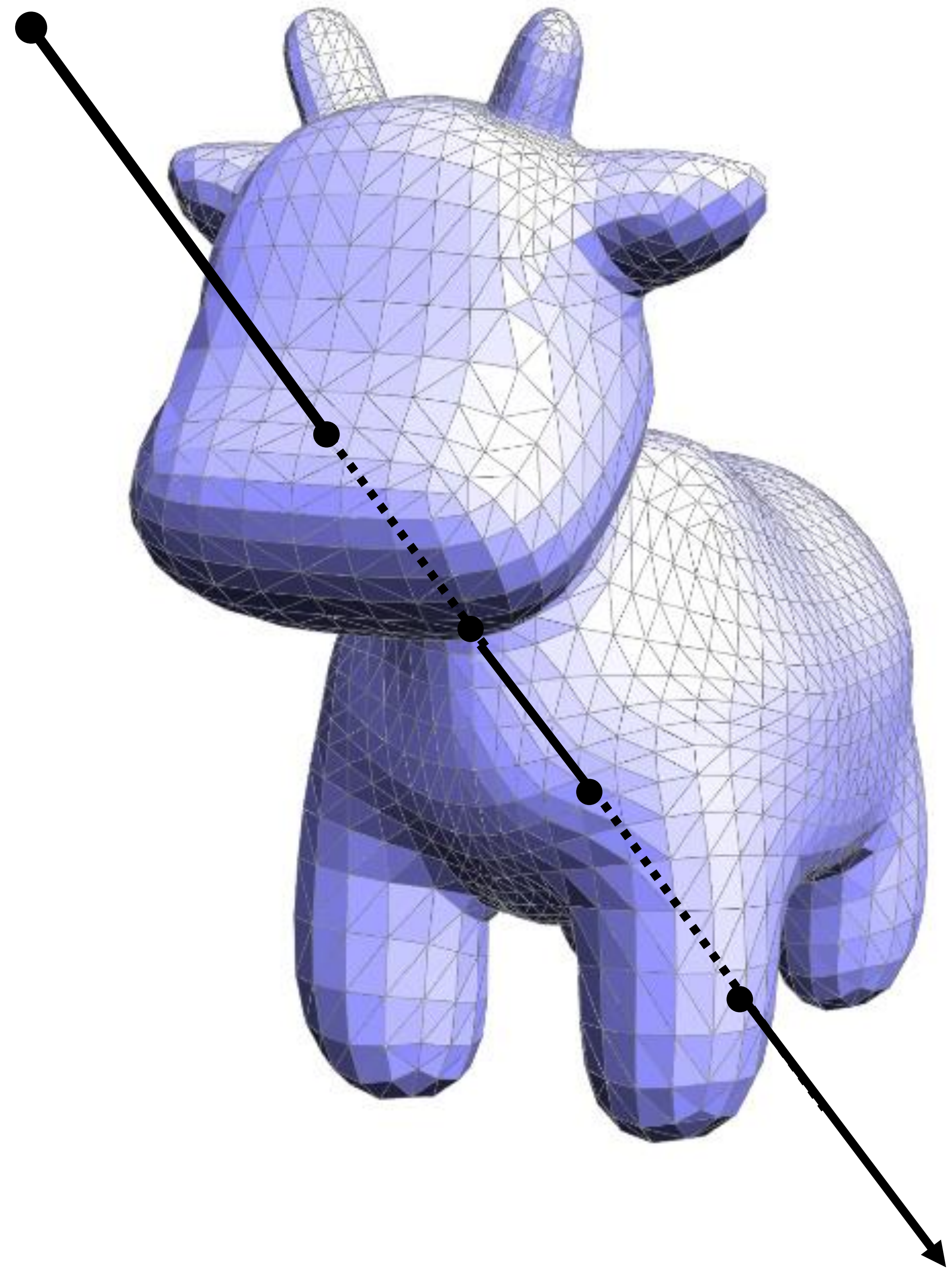
"Find the first primitive the ray hits"

```
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
```

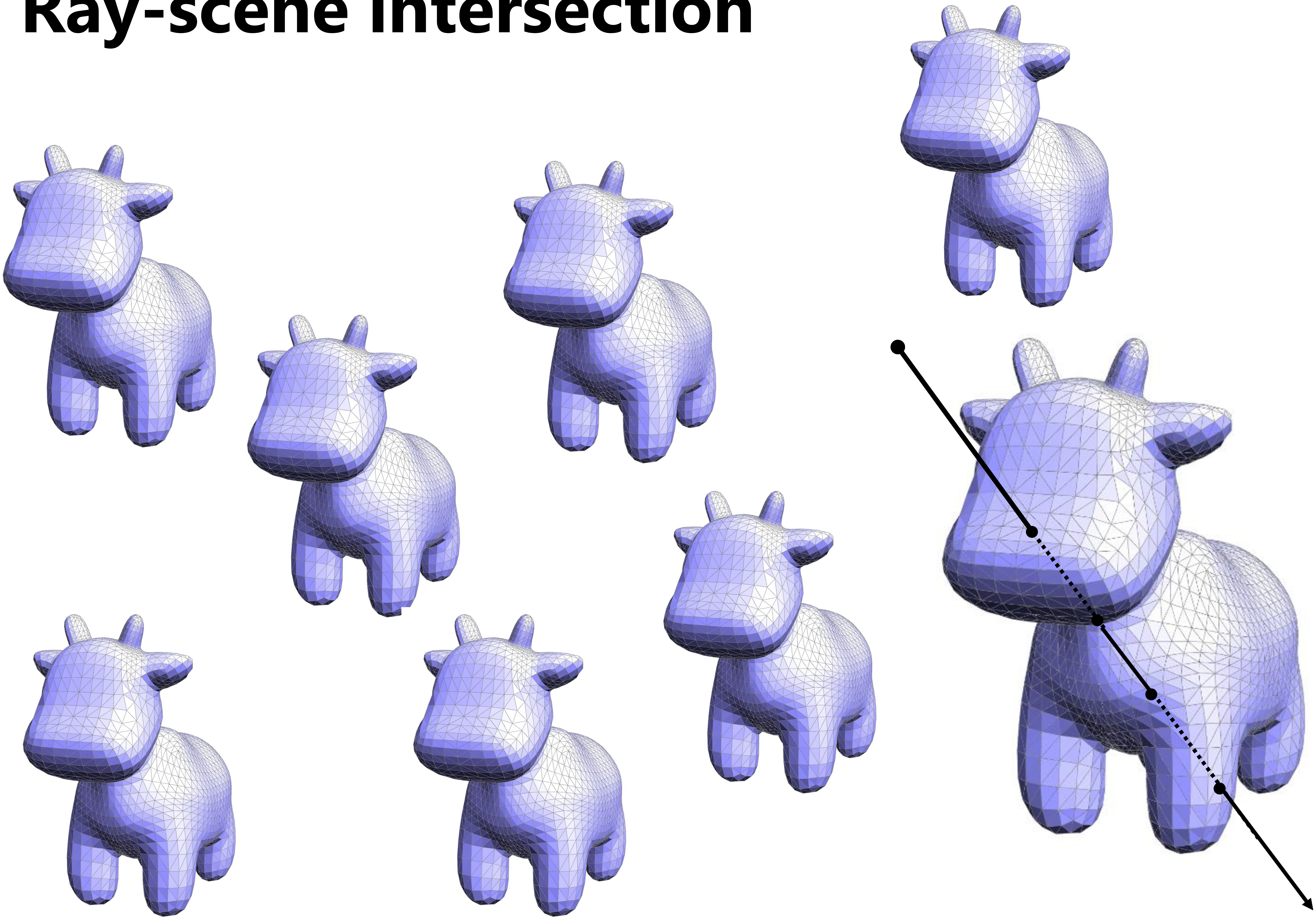
Complexity: $O(N)$



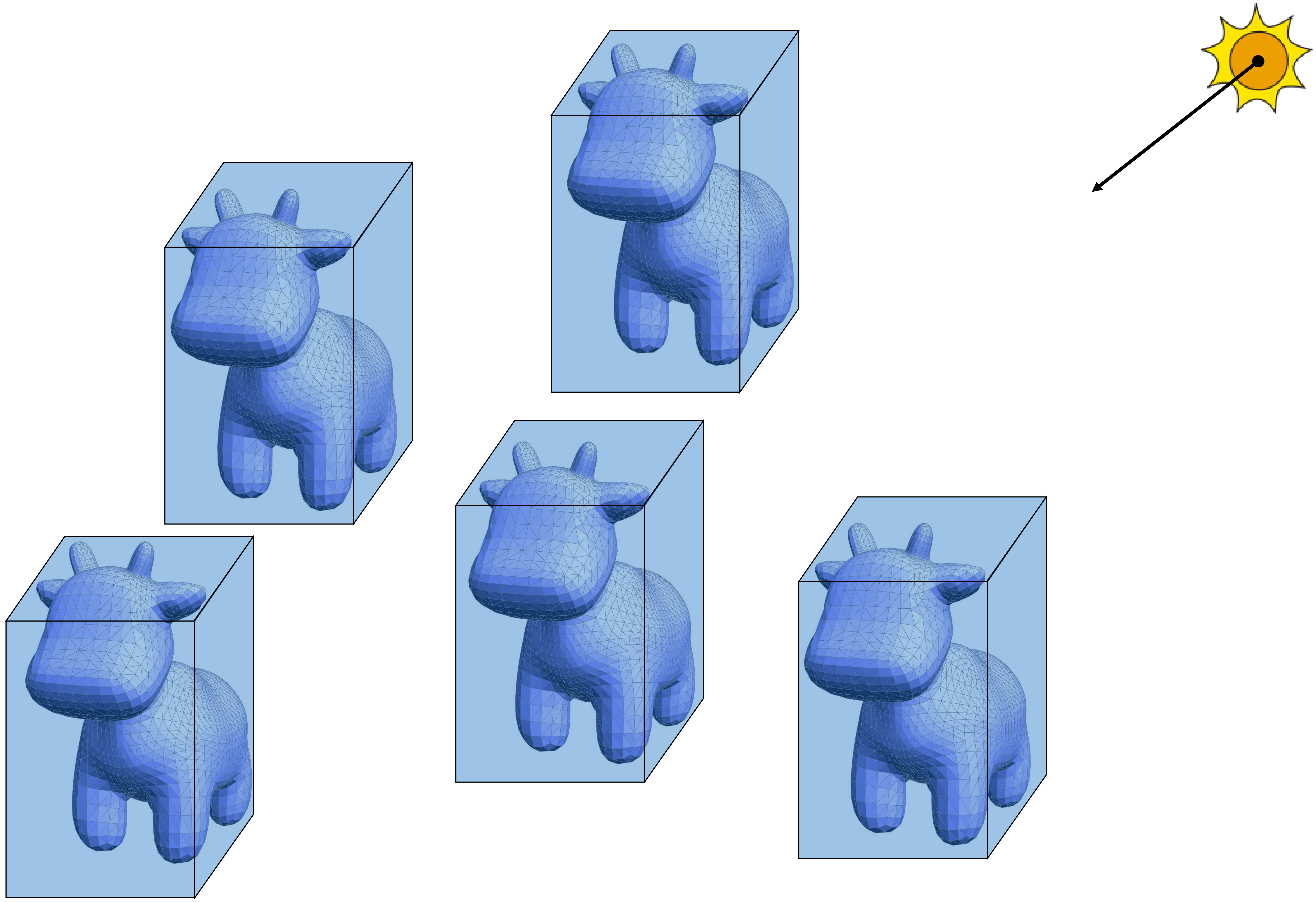




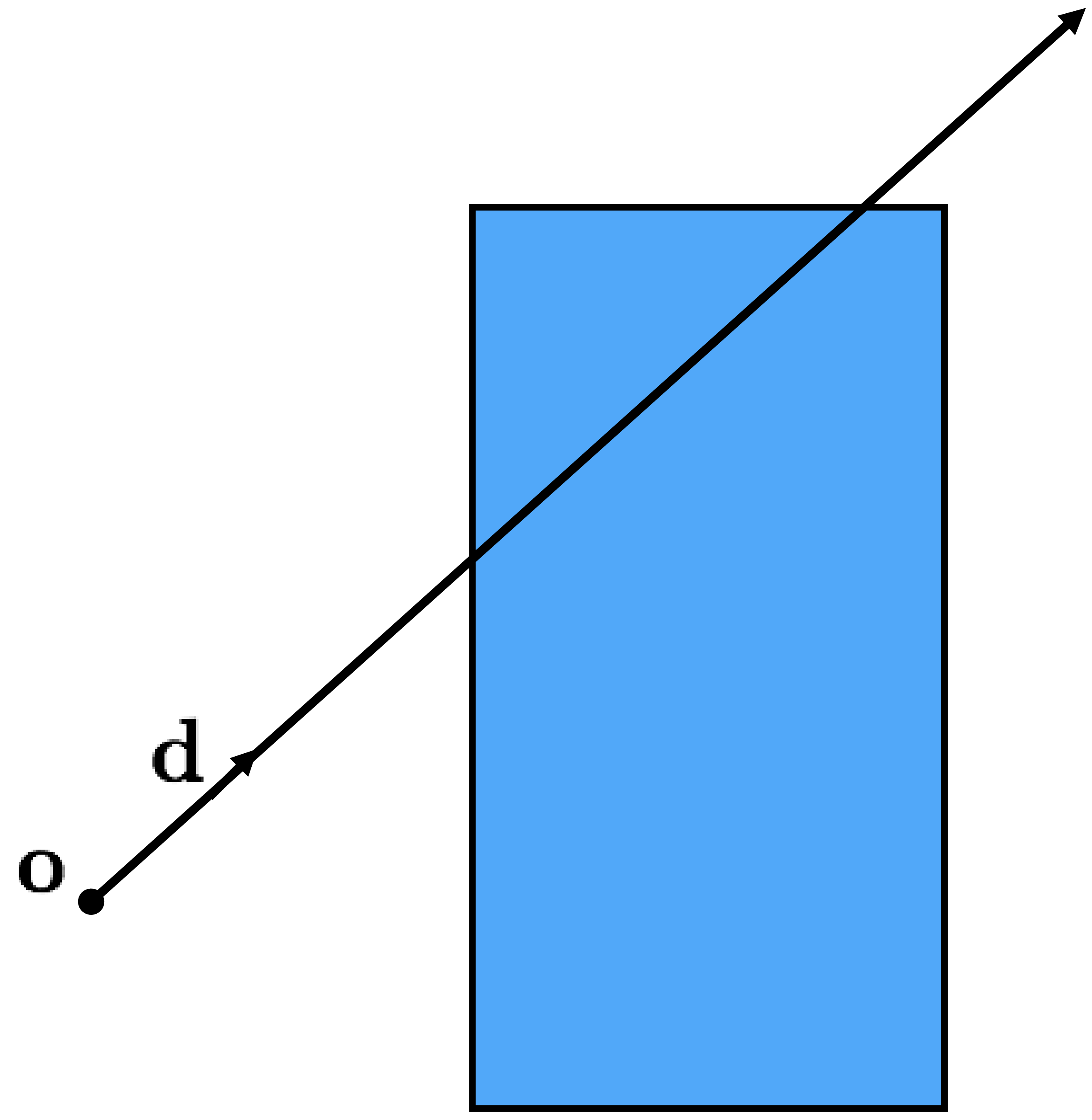
Ray-scene intersection



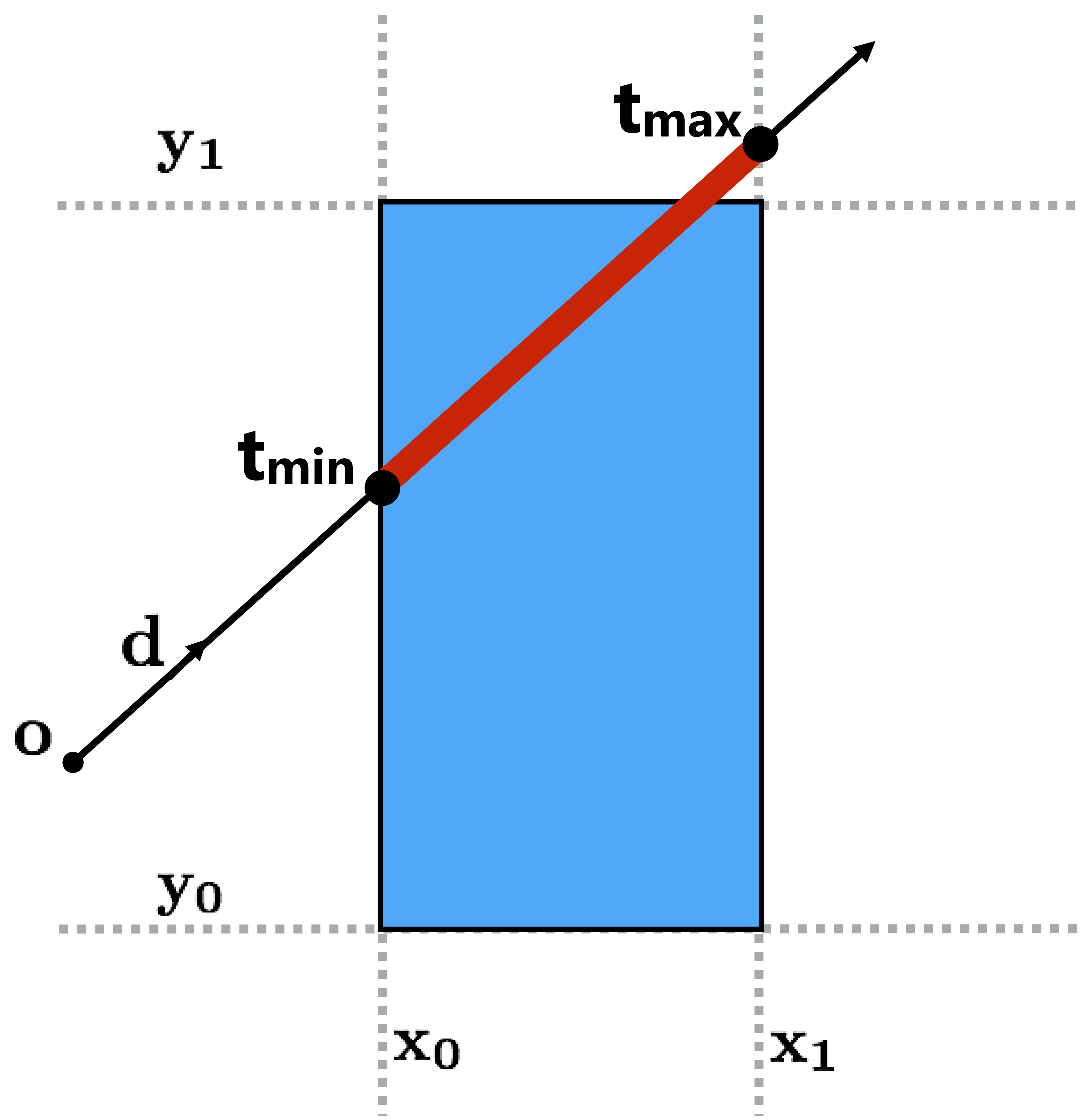
Ray-scene intersection – a first optimization



Ray-axis-aligned-box intersection



Ray-axis-aligned-box intersection



Find intersection of ray with all planes of box:

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

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

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

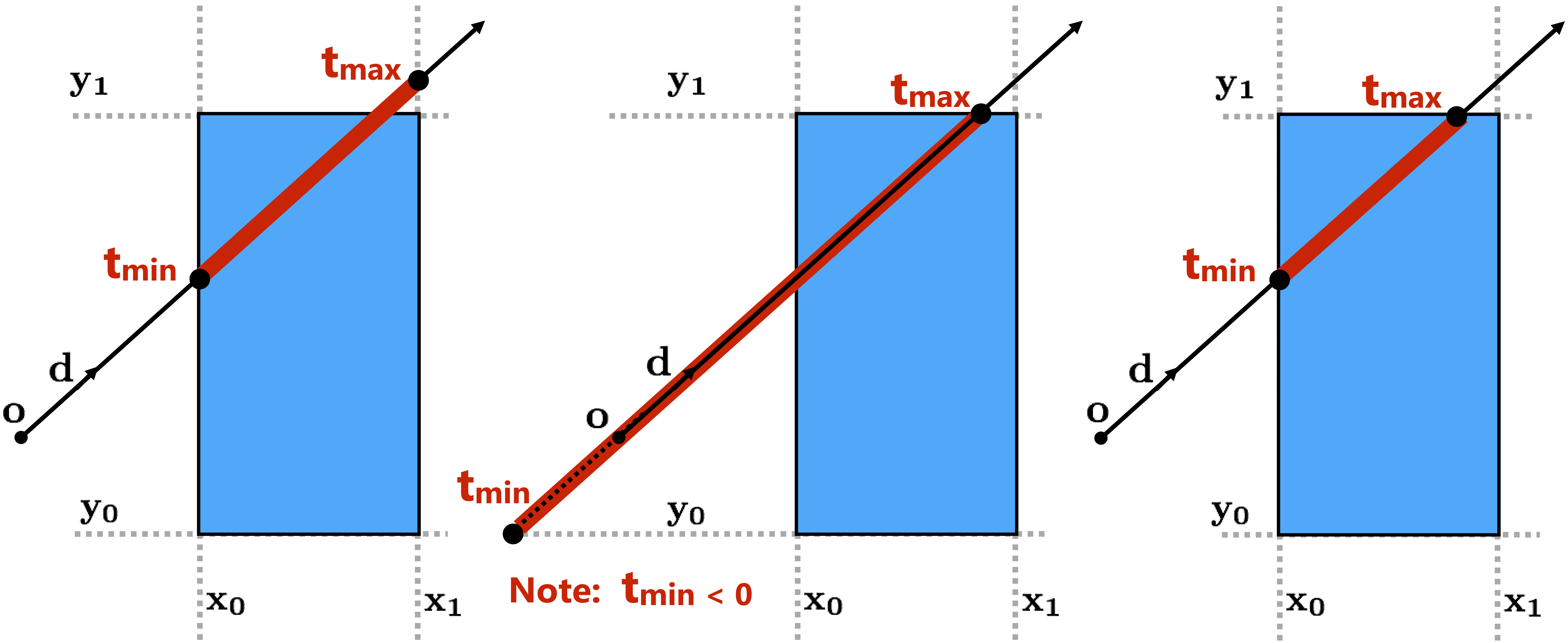
$$c = x_0$$

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

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

Ray-axis-aligned-box intersection

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

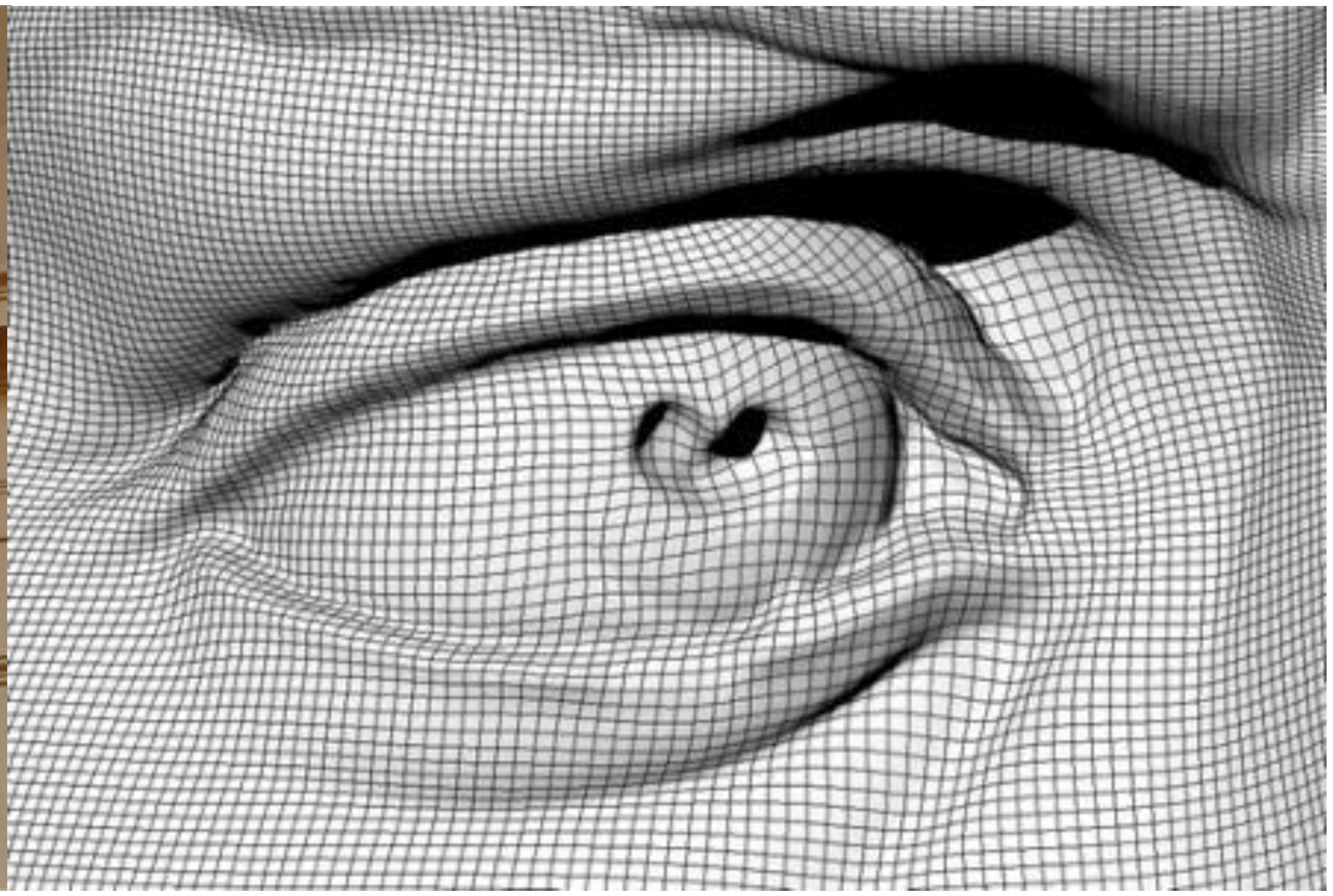


Intersections with x planes | Intersections with y planes | Final intersection result

How do we know when the ray misses the box?

Ray-scene intersection

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

Let's look at a simpler problem

- **Take a set of integers S**

10 123 20 100 6 25 64 11 200 30

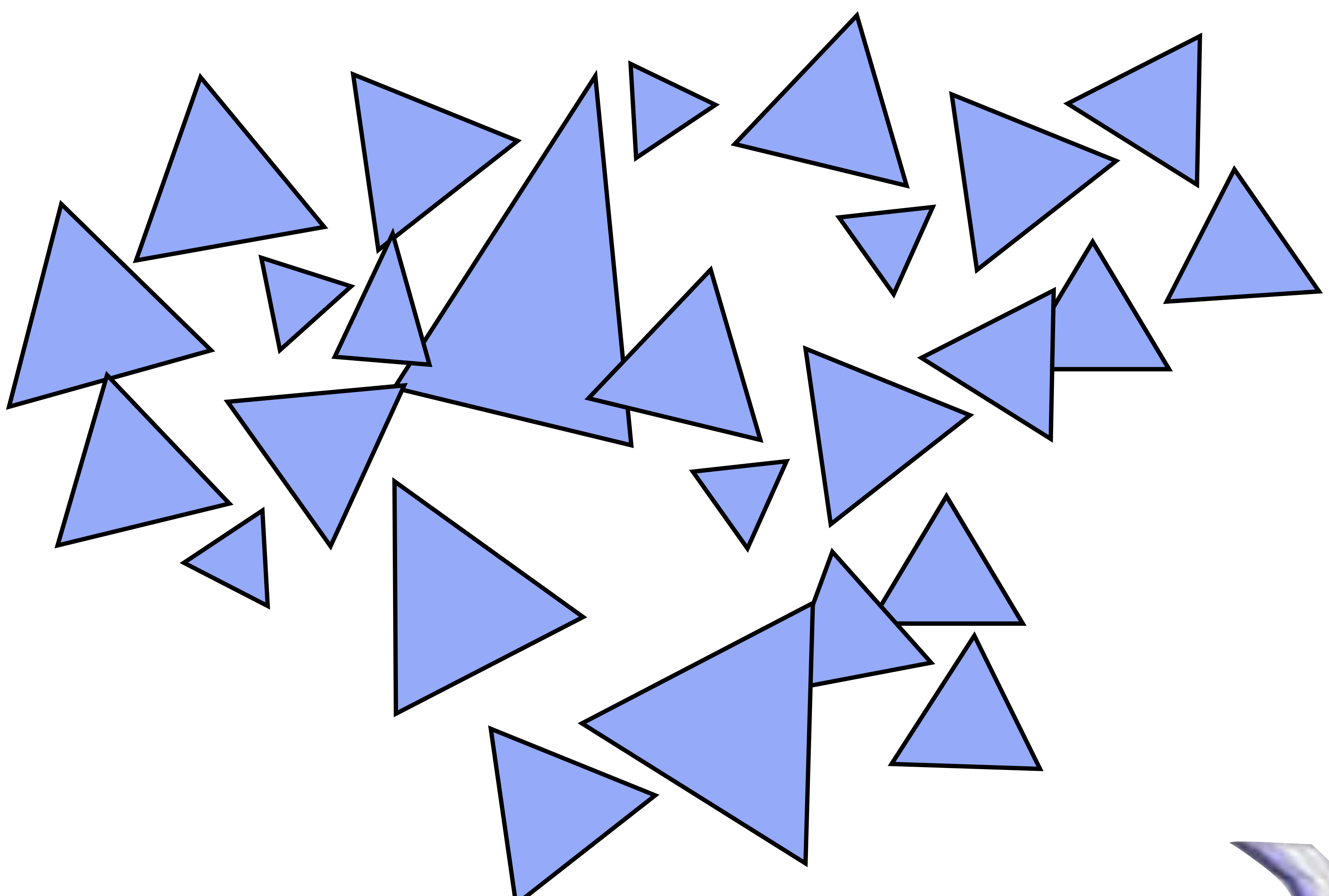
- **Given a new integer *k=18*, find the element in S that is closest**

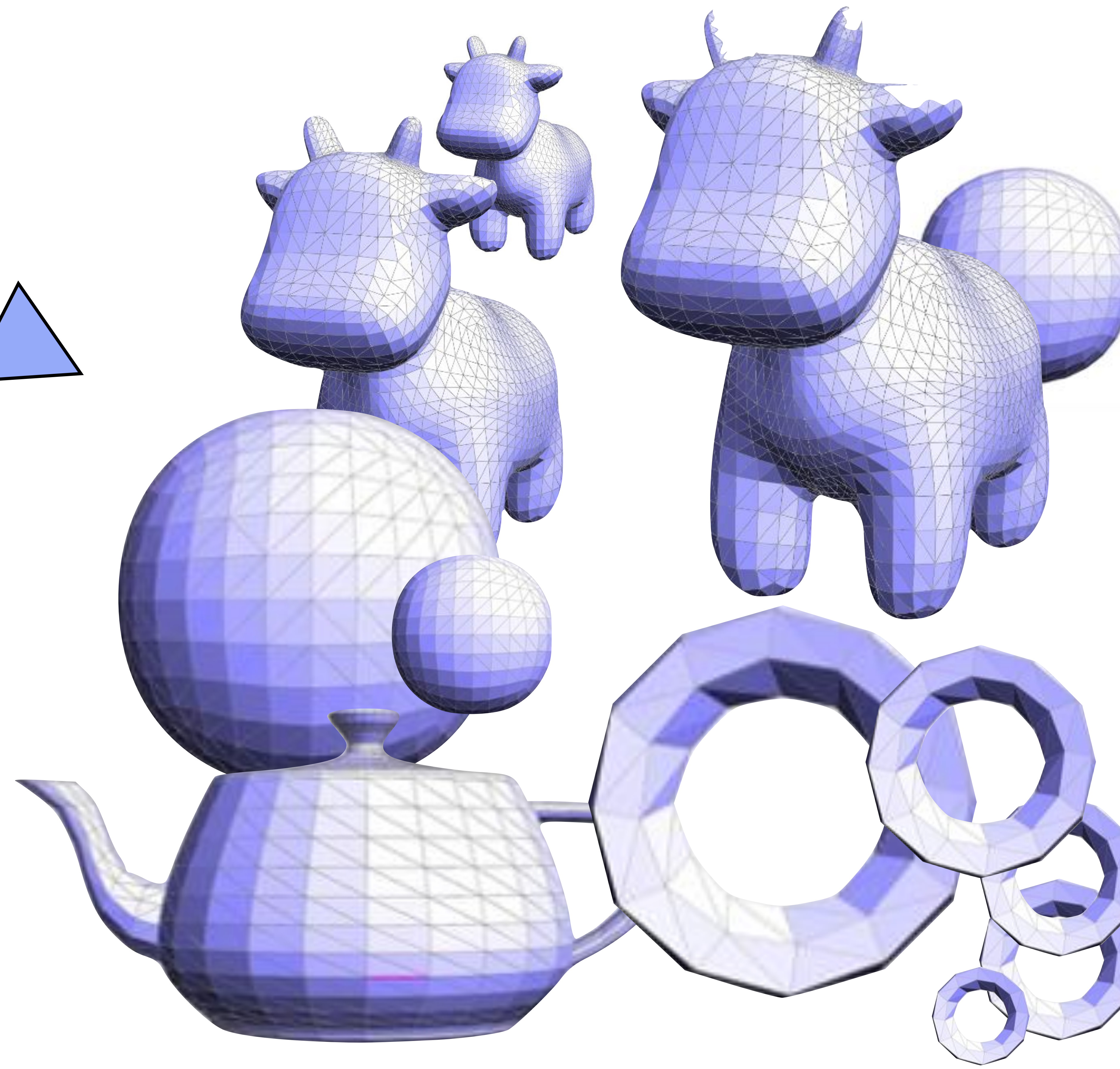
Sort first:

6 10 11 20 25 30 64 100 123 200

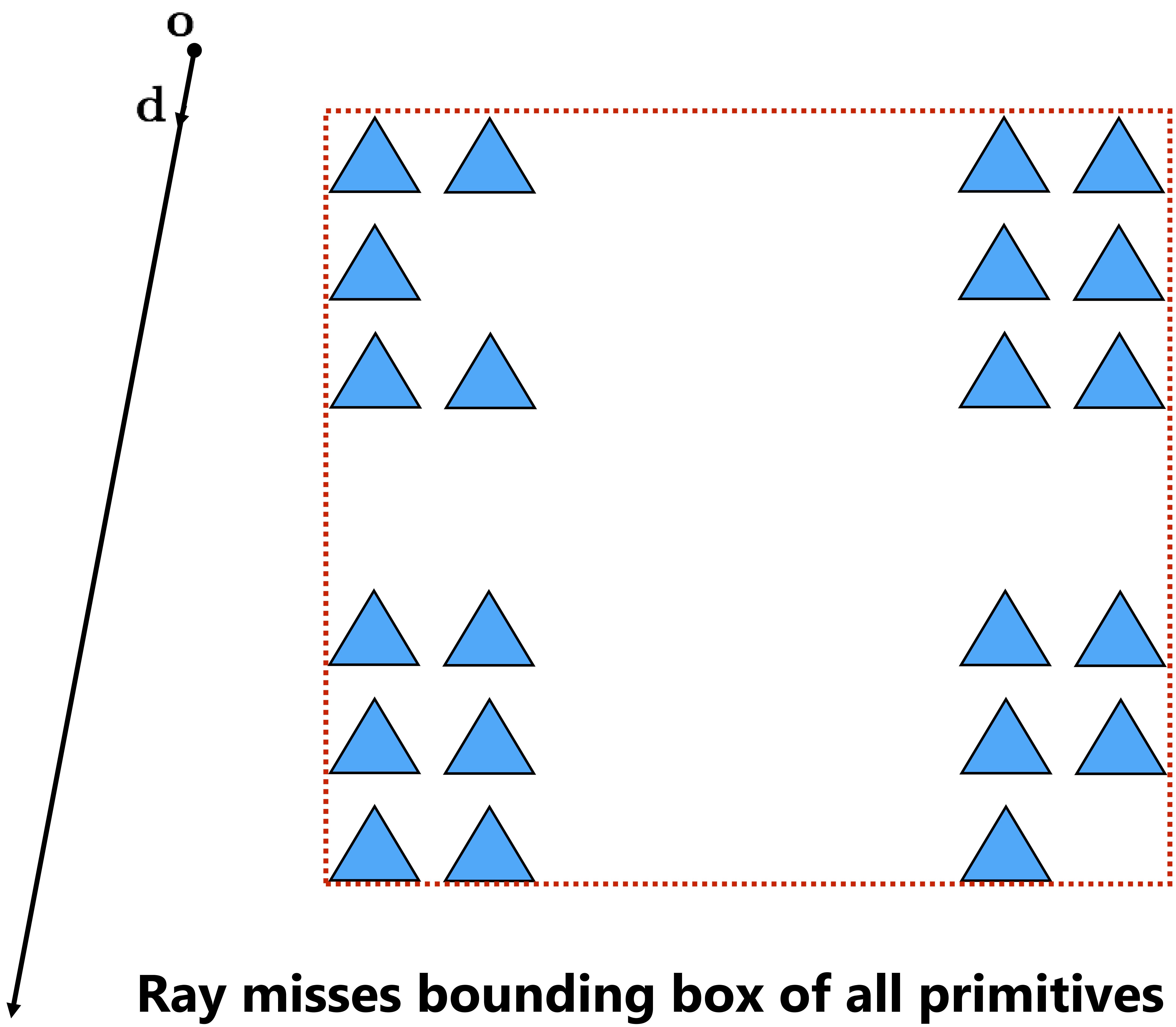
Then what?

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



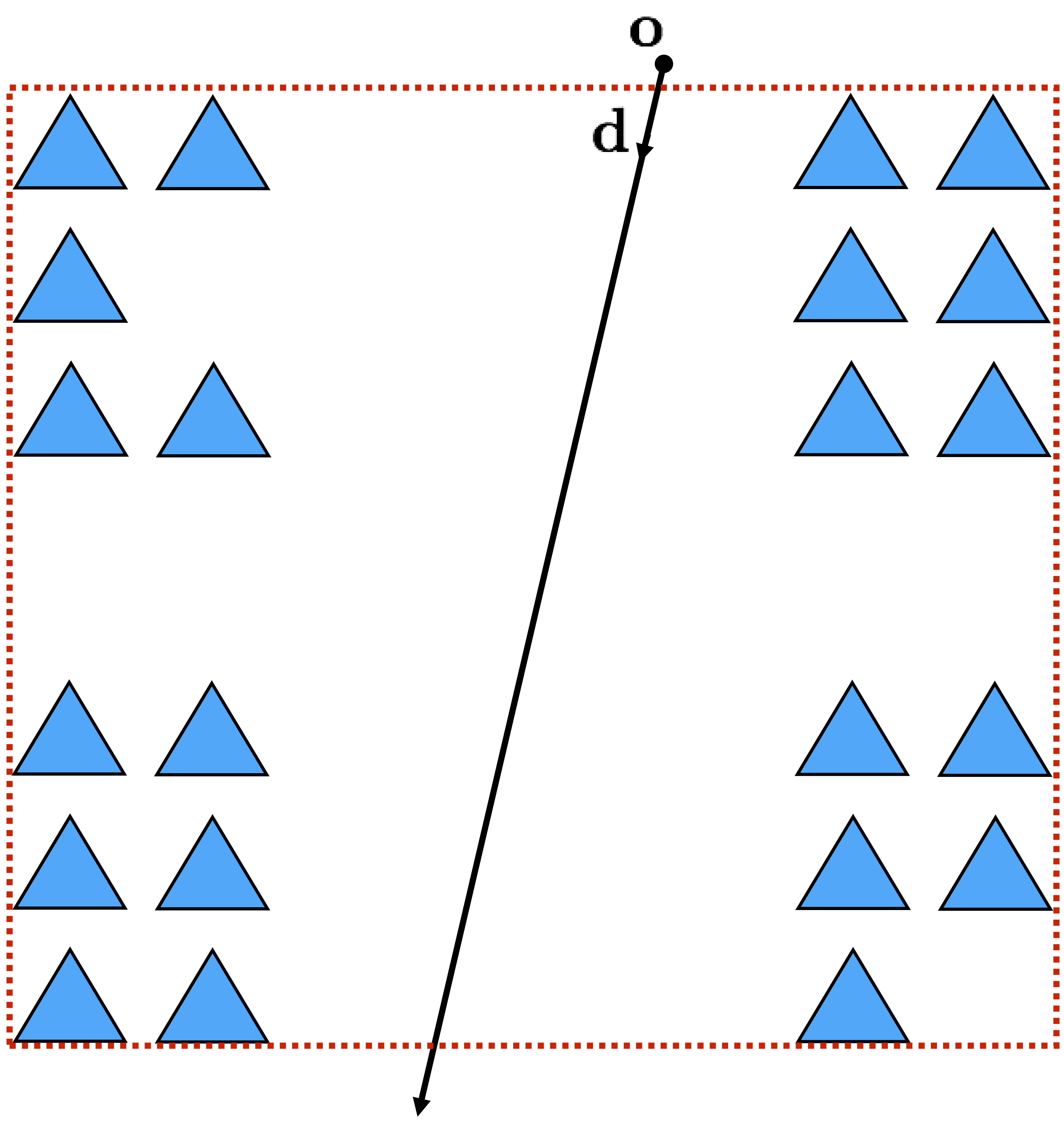


Simple case (we've seen it already)



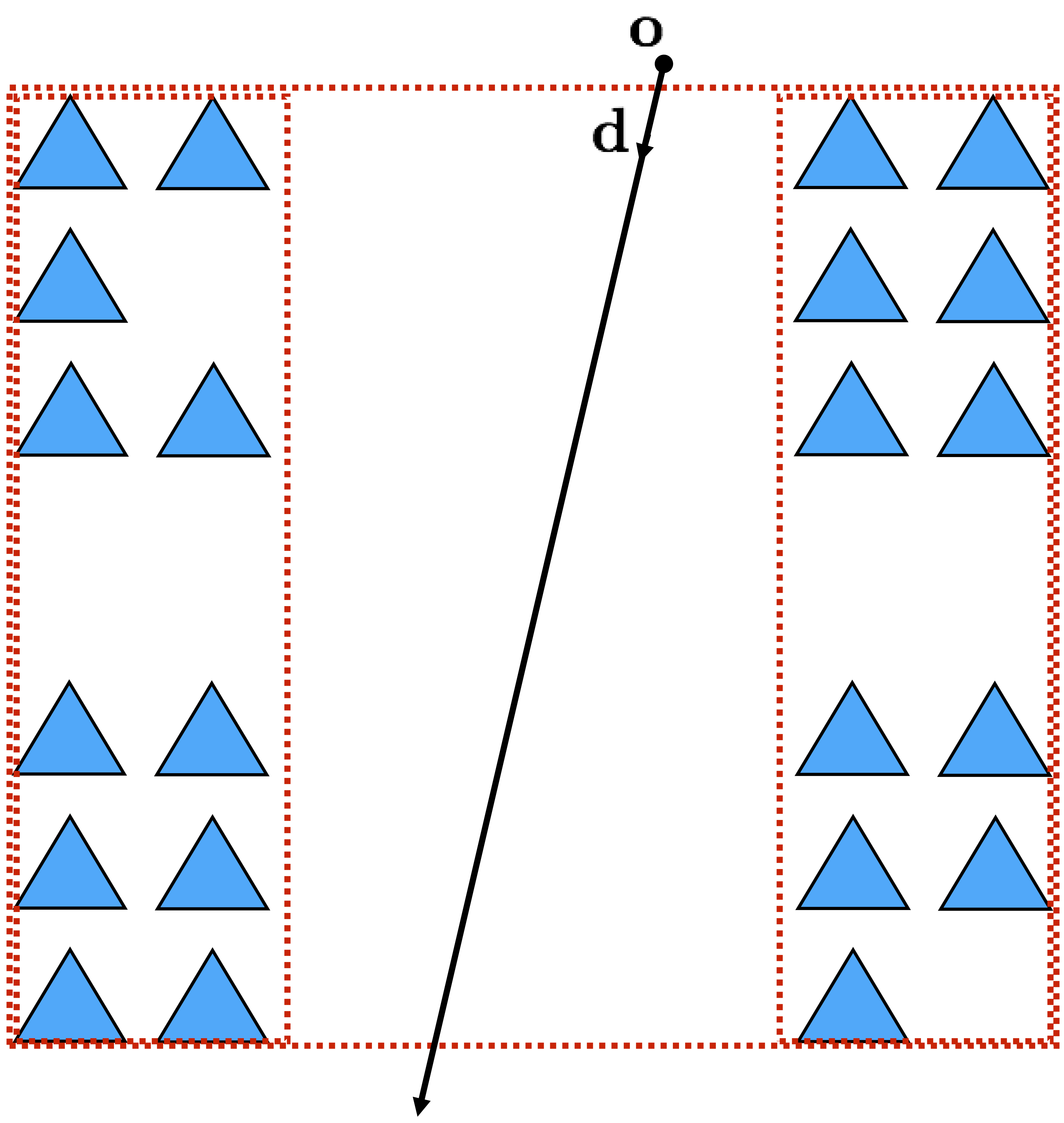
Ray misses bounding box of all primitives in scene
O(1) cost: requires 1 ray-box test

Another (should be) simple case



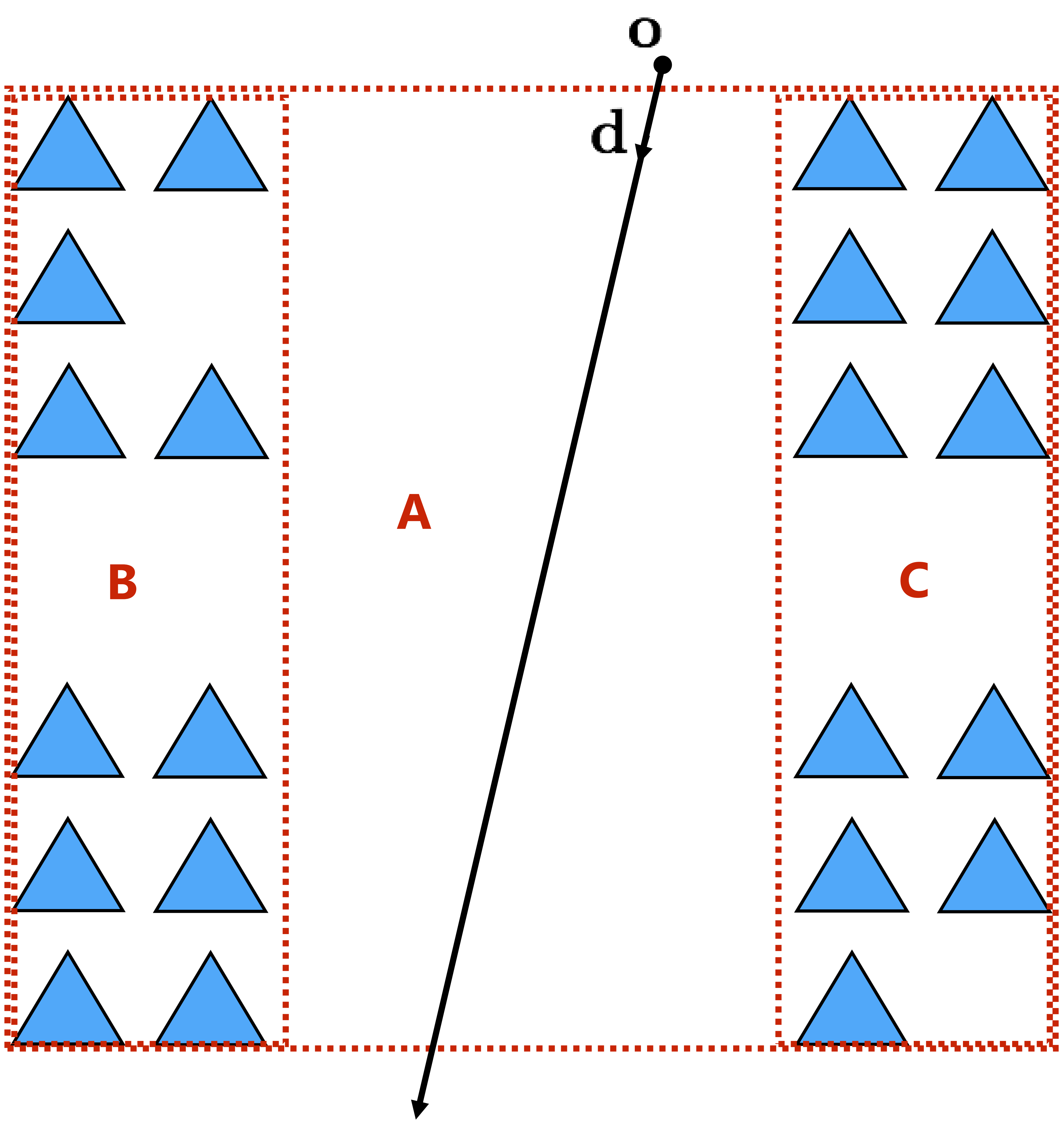
Ray hits bounding box, check all primitives
O(N) cost ☹

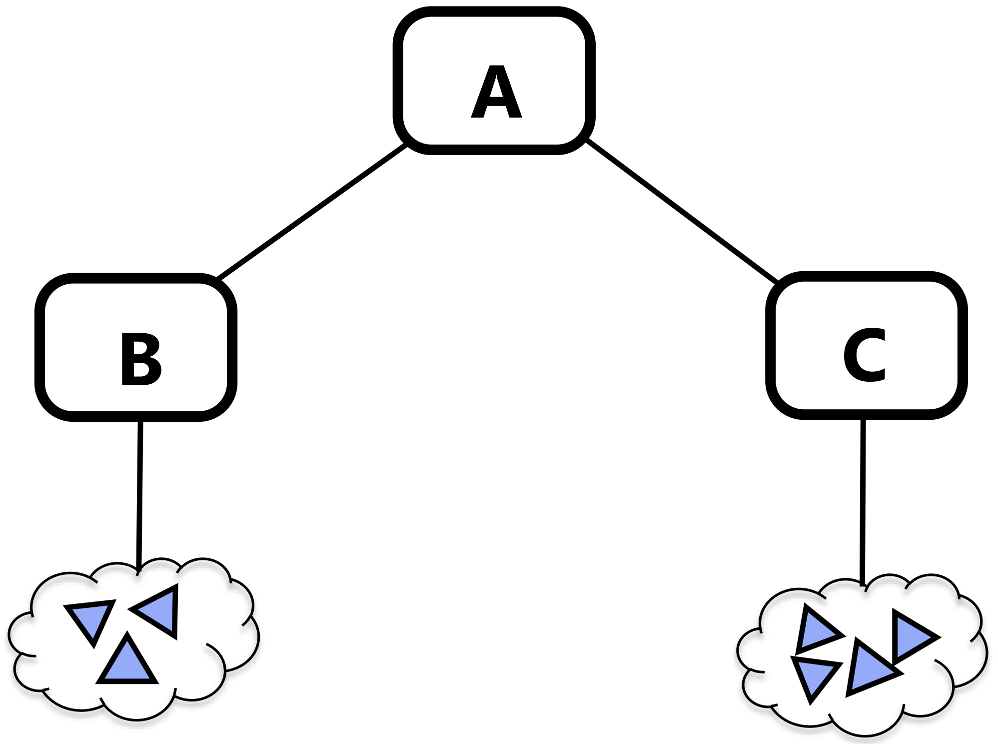
Another (should be) simple case



A bounding box of bounding boxes!

Another (should be) simple case

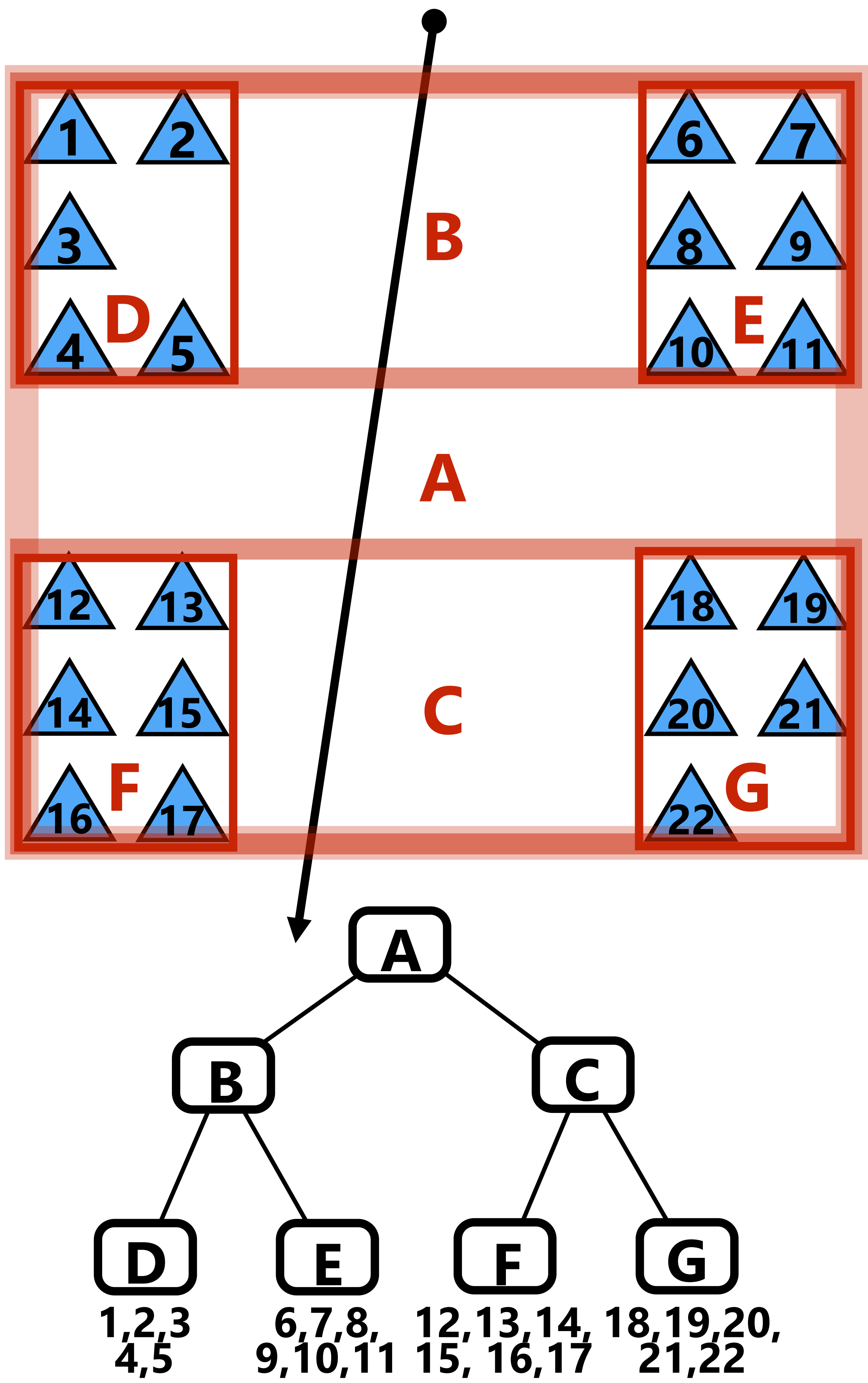


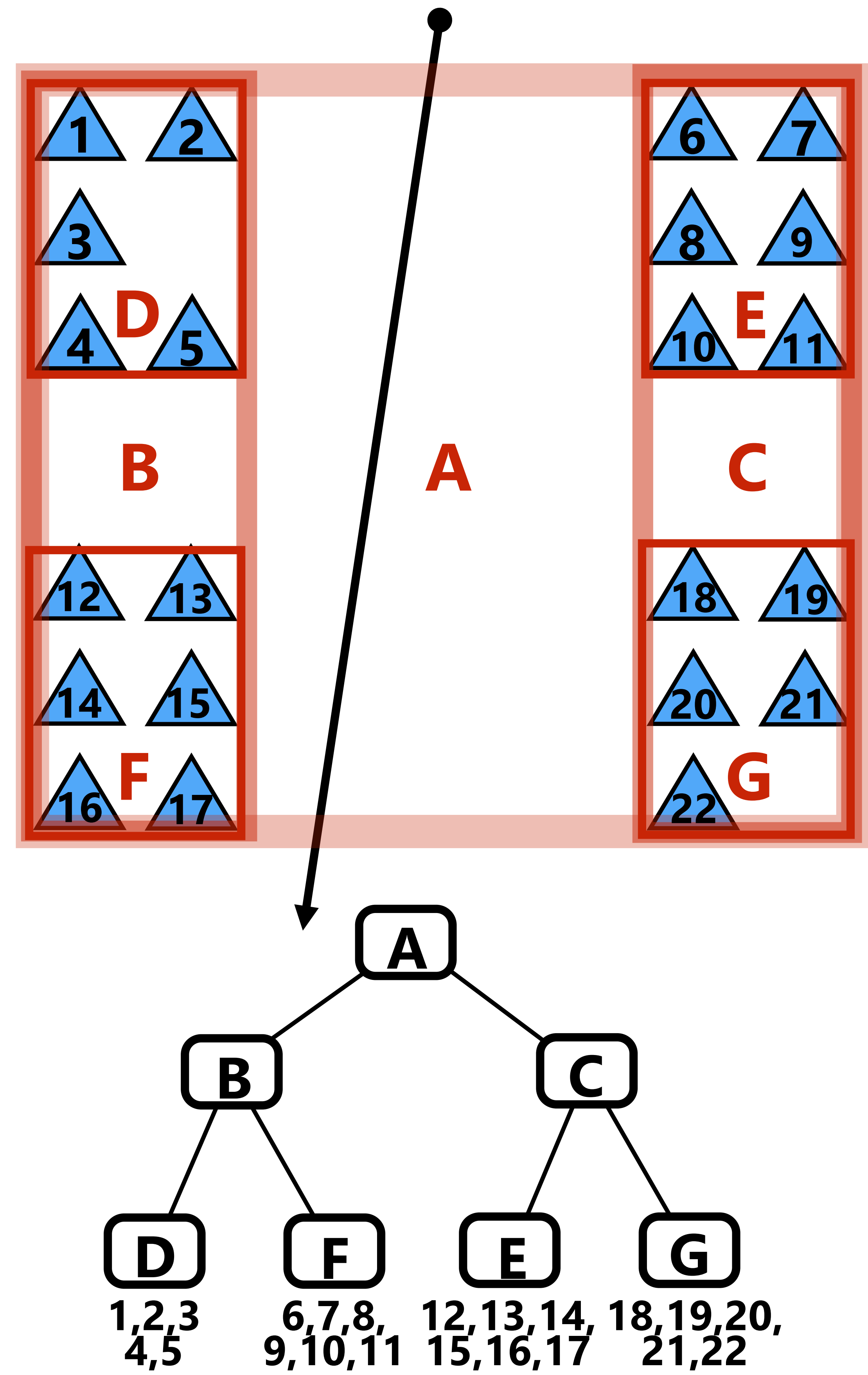


There is no reason to stop there!

Bounding volume hierarchy (BVH)

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



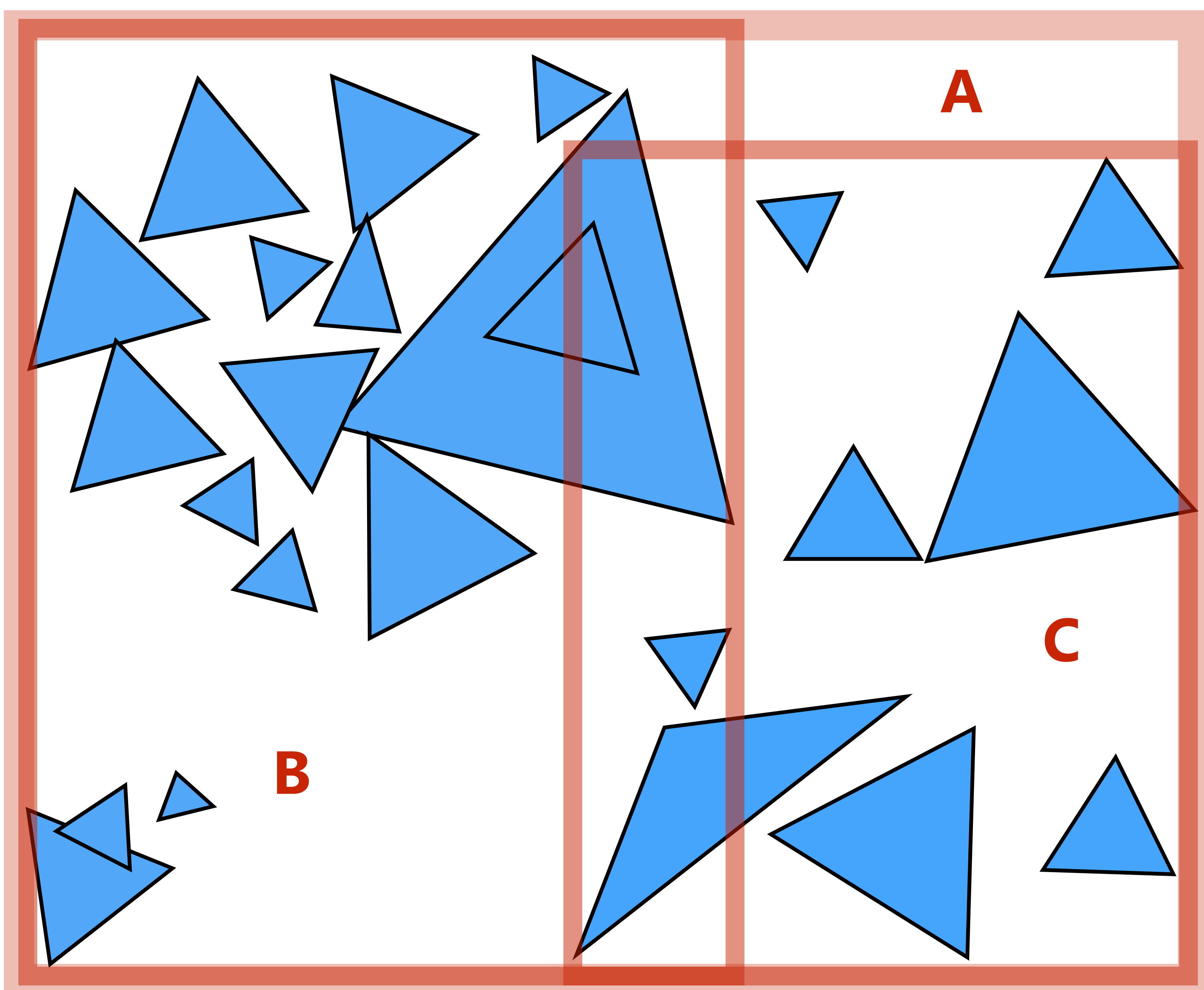


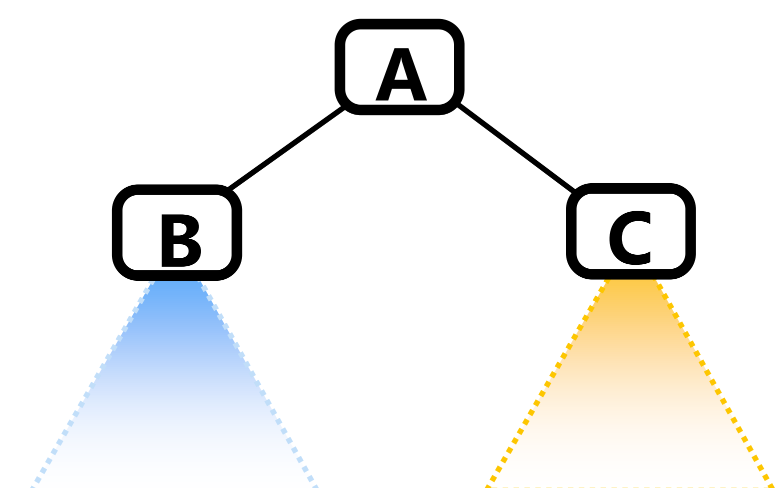
Two different BVH organizations of the same scene containing 22 primitives. Leaf node are the same.

Q: Which one is better?

A less-structured BVH example

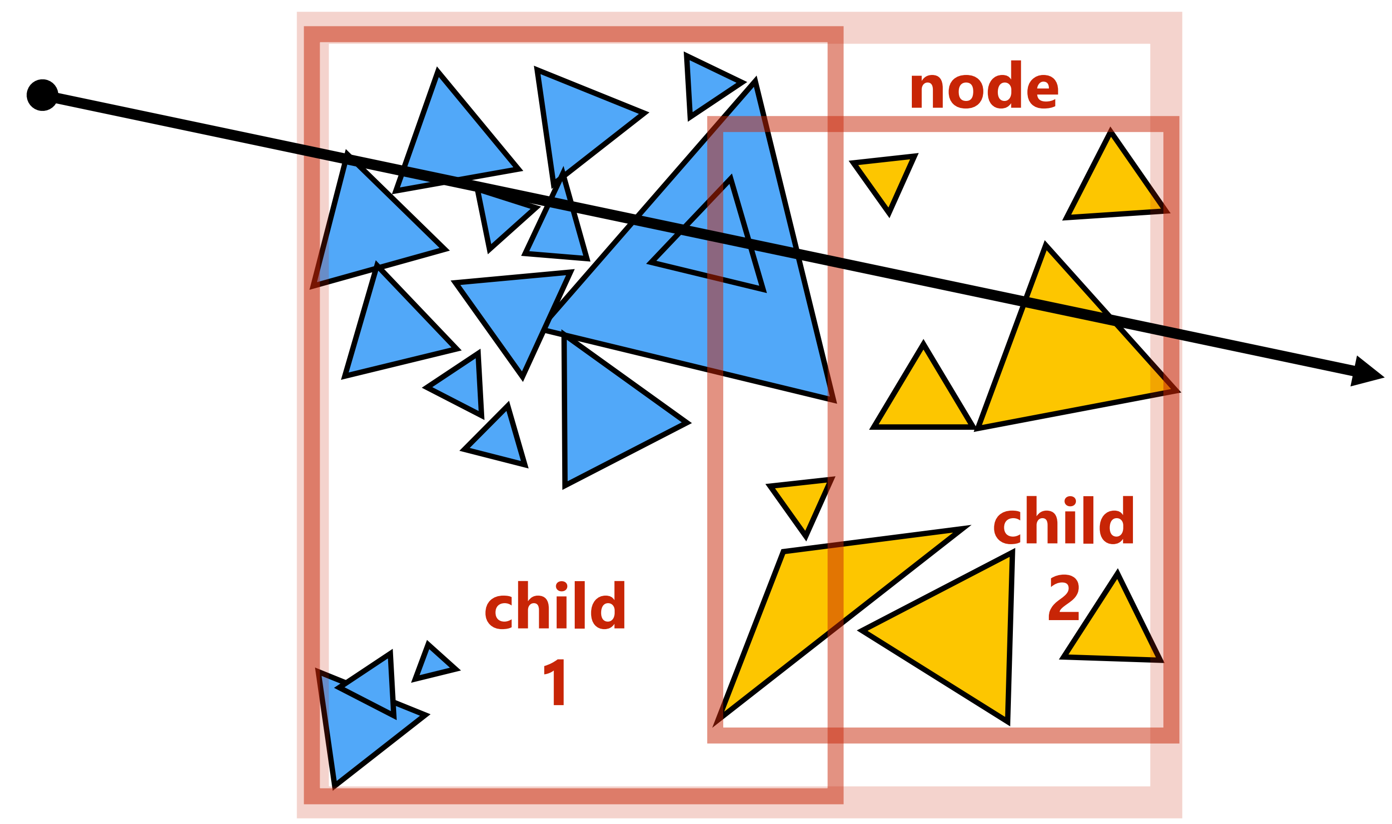
- **BVH partitions each node's primitives into disjoints sets**
  - **Note: The sets can still be overlapping in space!**





Ray-scene intersection using a BVH

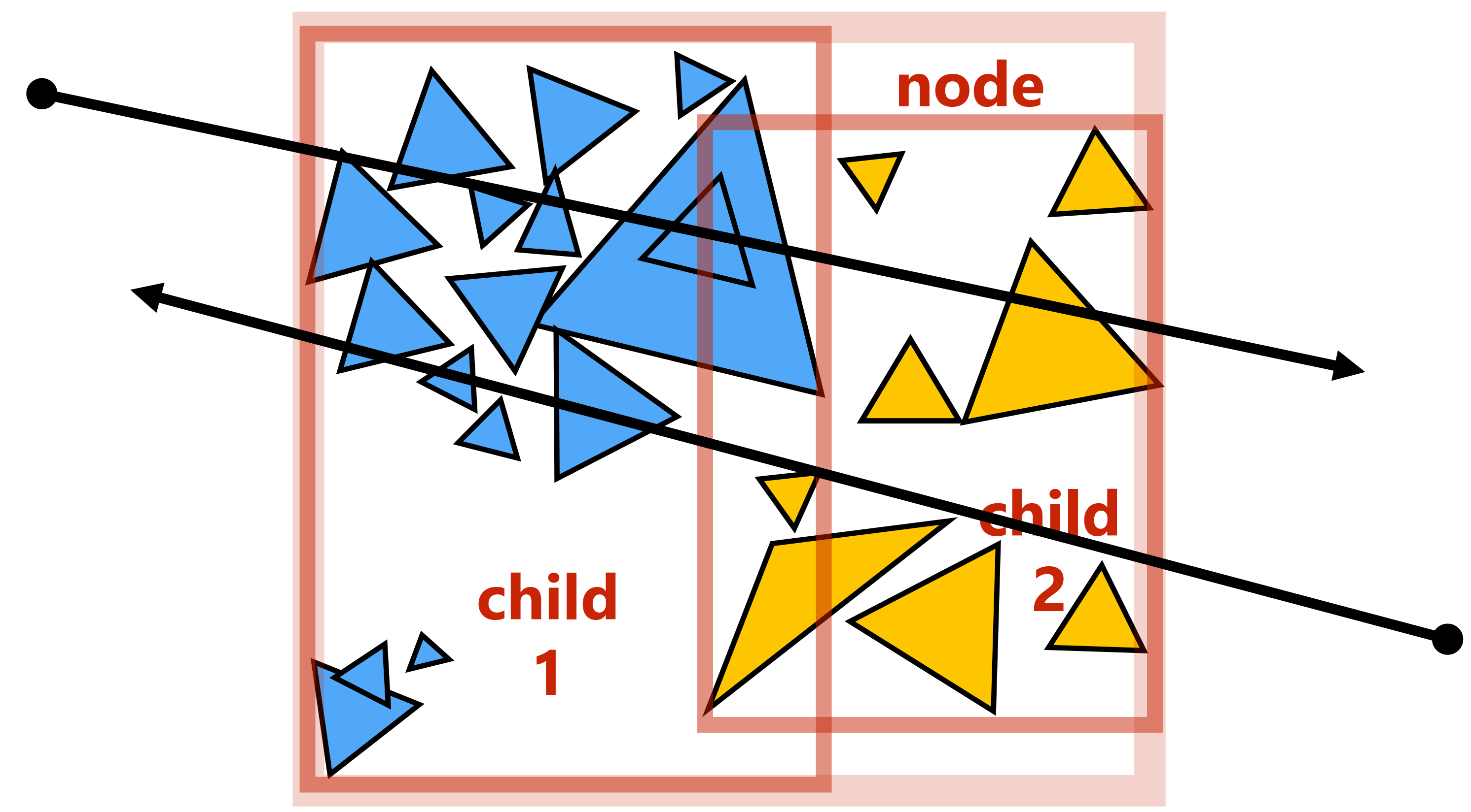
```
struct BVHNode {
   bool leaf;
   BBox bbox;
   BVHNode* child1;
   BVHNode* child2;
   Primitive* primList;
};

struct ClosestHitInfo {
   Primitive prim;
   float min_t;
};

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

   if (!intersect(ray, node->bbox))
      return;

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



Hmmm... this is still checking all the primitives in the scene.

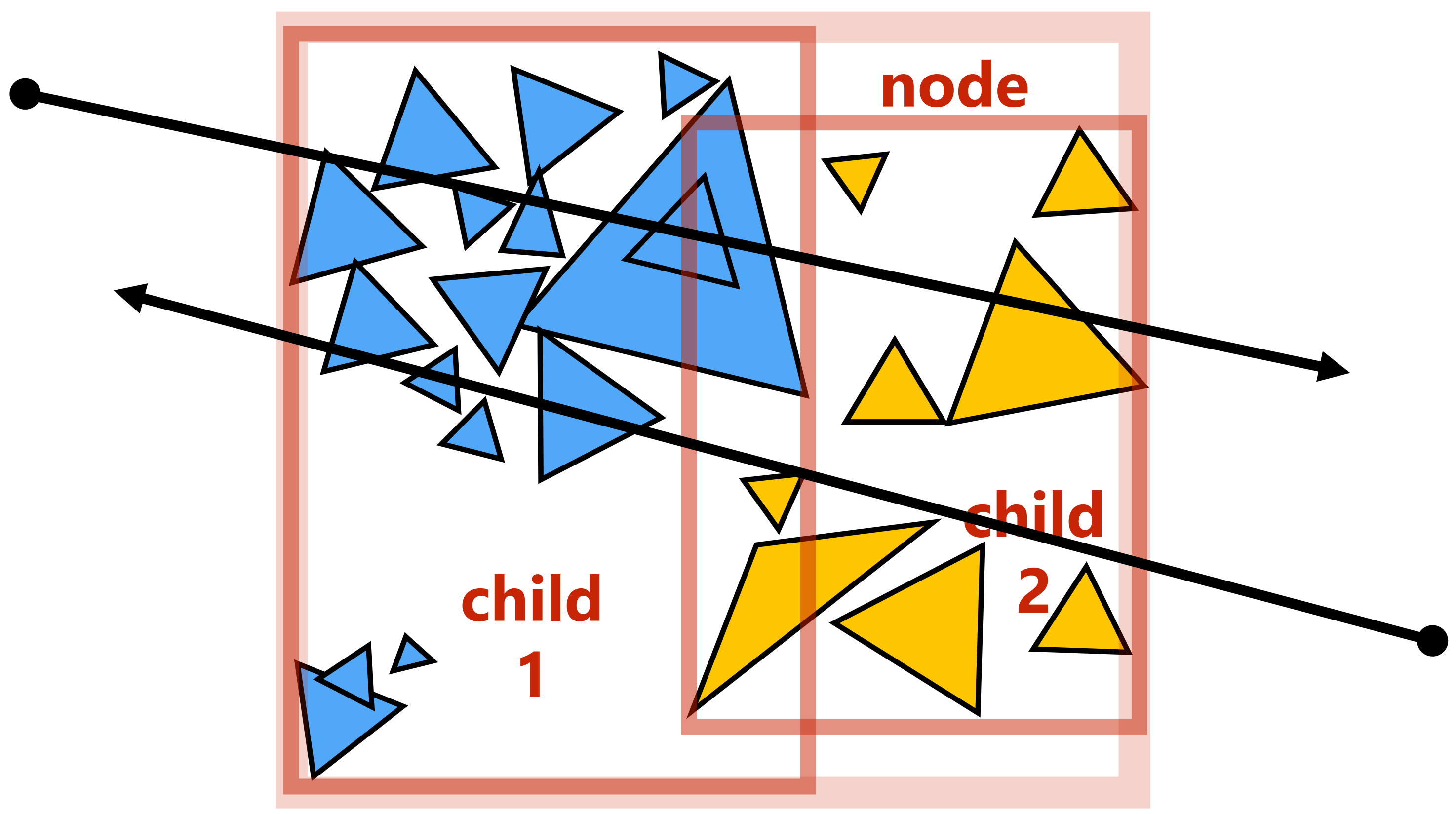
Ray-scene intersection using a BVH

```
struct BVHNode {
   bool leaf;
   BBox bbox;
   BVHNode* child1;
   BVHNode* child2;
   Primitive* primList;
};

struct ClosestHitInfo {
   Primitive prim;
   float min_t;
};

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

   if (!intersect(ray, node->bbox) || (closest point on box is farther than closest.min_t))
      return;

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



What if ray points the other way?

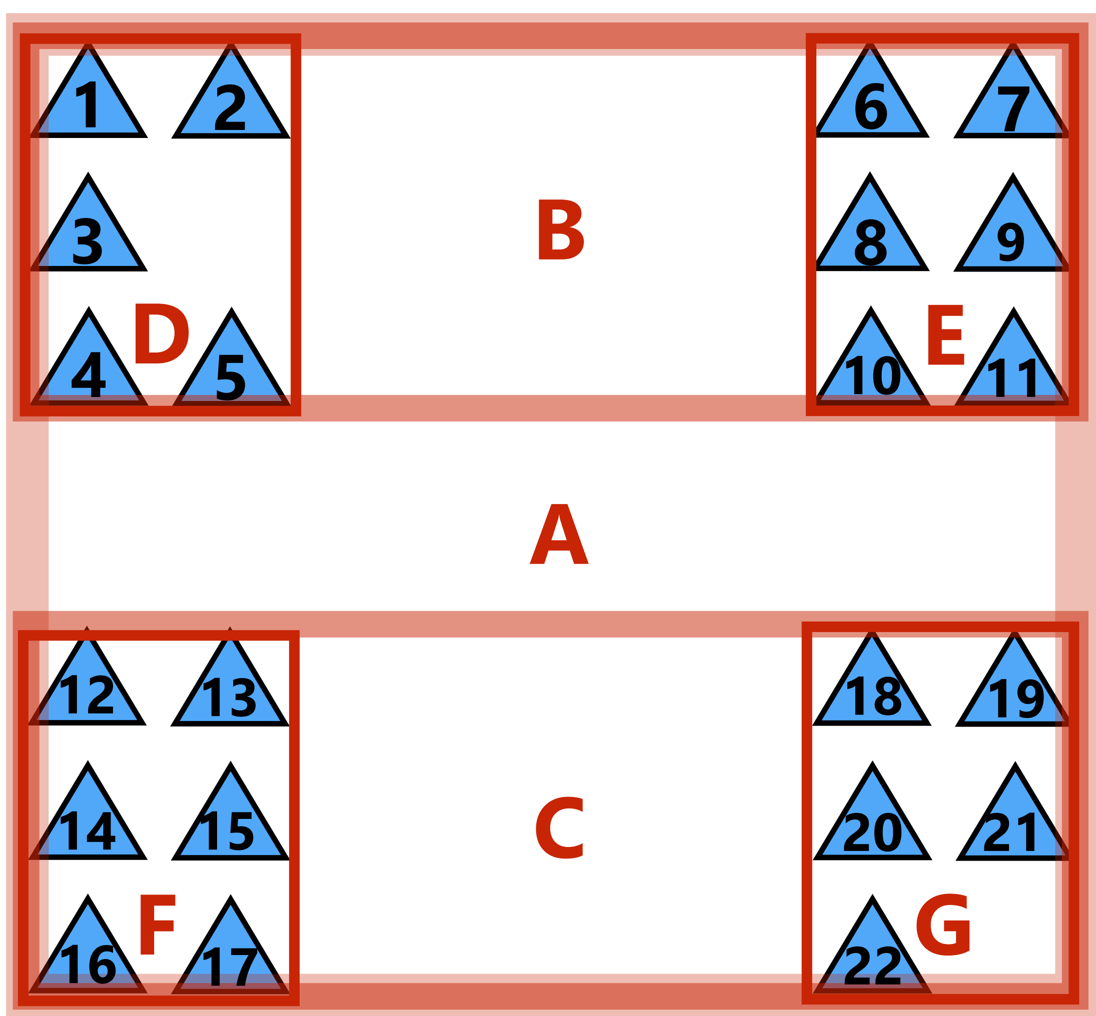
Improvement: “front-to-back” traversal

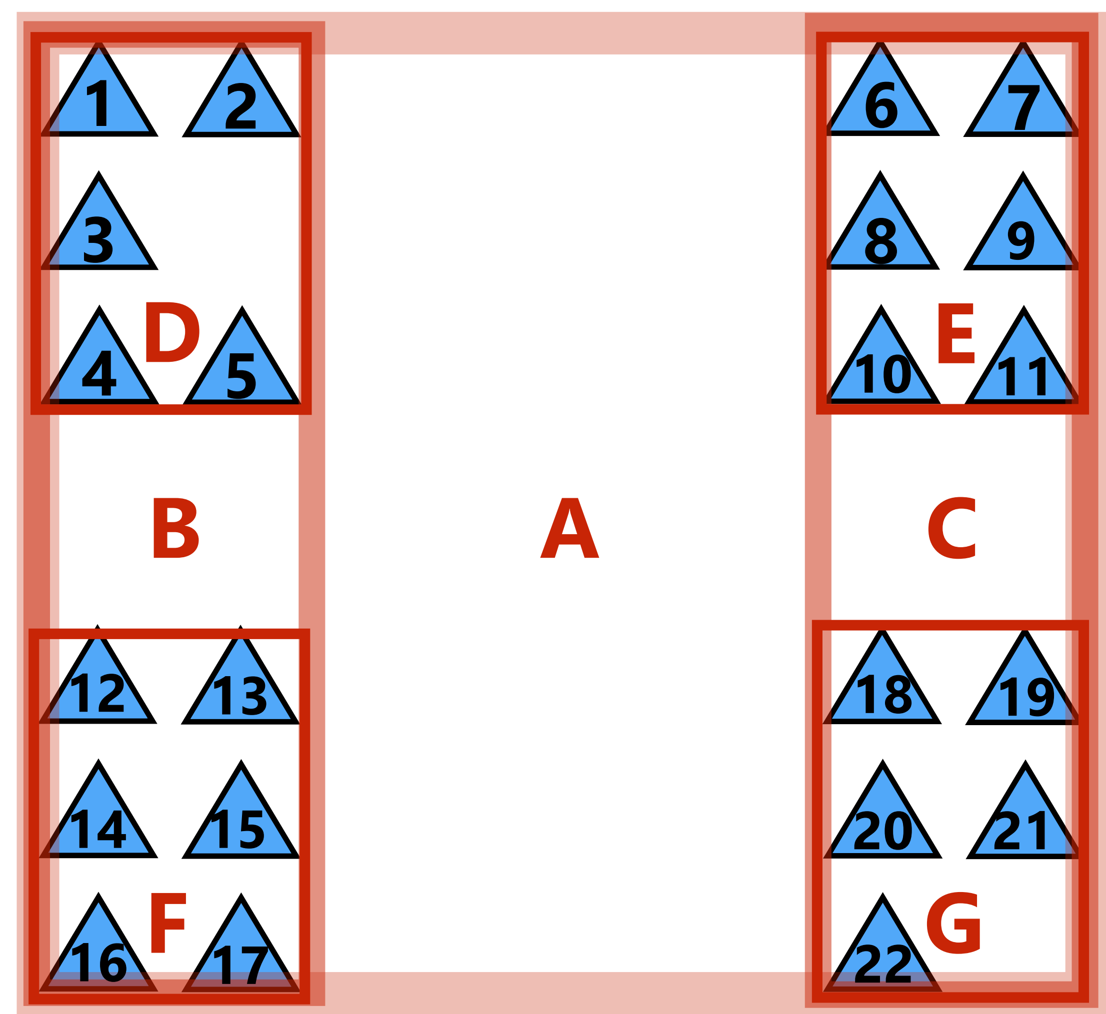



```
void find_closest_hit(Ray* ray, BVHNode* node, ClosestHitInfo* closest) {
 if (!intersect(ray, node->bbox) || (closest point on box is farther than
closest.min_t))
      return;

   if (node->leaf) {
      for (each primitive p in node->primList) {
         (hit, t) = intersect(ray, p);
         if (hit && t < closest.min_t) {
            closest.prim = p;
            closest.min_t = t;
         }
      }
   } else {
      (hit1, min_t1) = intersect(ray, node->child1->bbox);
      (hit2, min_t2) = intersect(ray, node->child2->bbox);

      NVHNode* first = (min_t1 <= min_t2) ? child1 : child2;
      NVHNode* second = (min_t1 <= min_t2) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (second child’s min_t is closer than closest.min_t)
         find_closest_hit(ray, second, closest);
   }
}
```

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

Bounding volume hierarchy (BVH)





Two different BVH trees.
Which one is better?

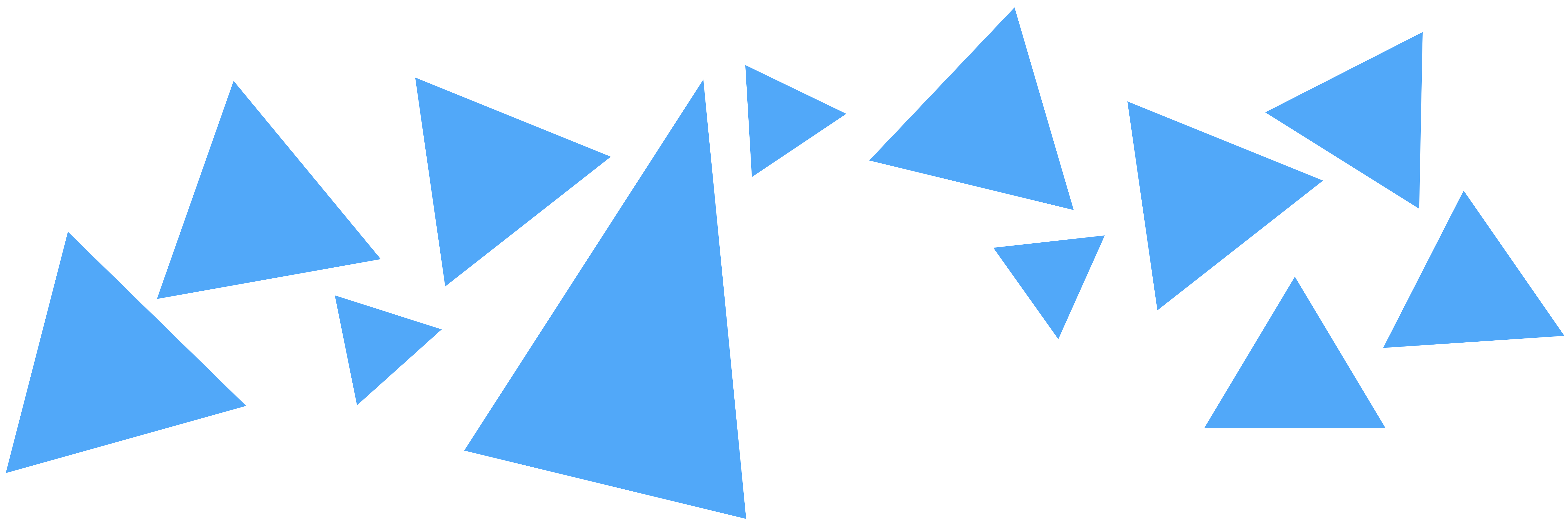
For a given set of primitives, there are many possible BVHs

Q: how many ways are there to partition N primitives into two groups?

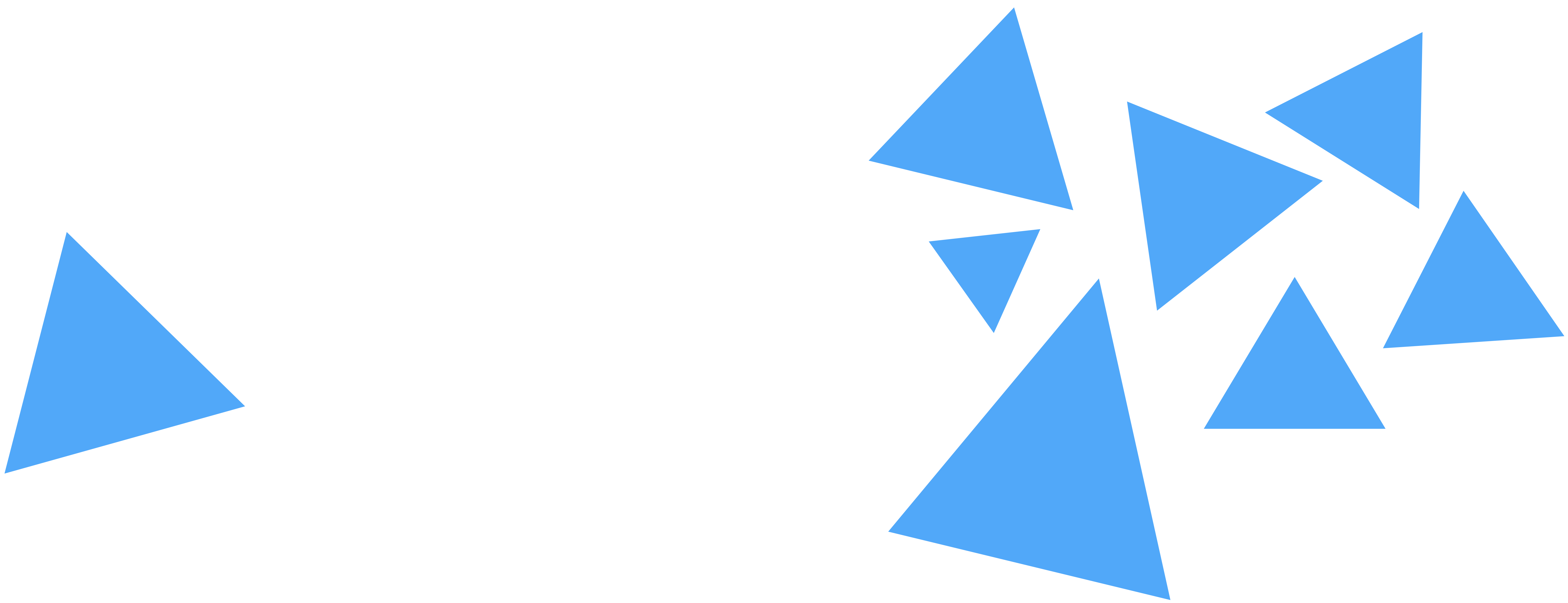
A: 2^N-2

So, how do we build a high-quality BVH tree?

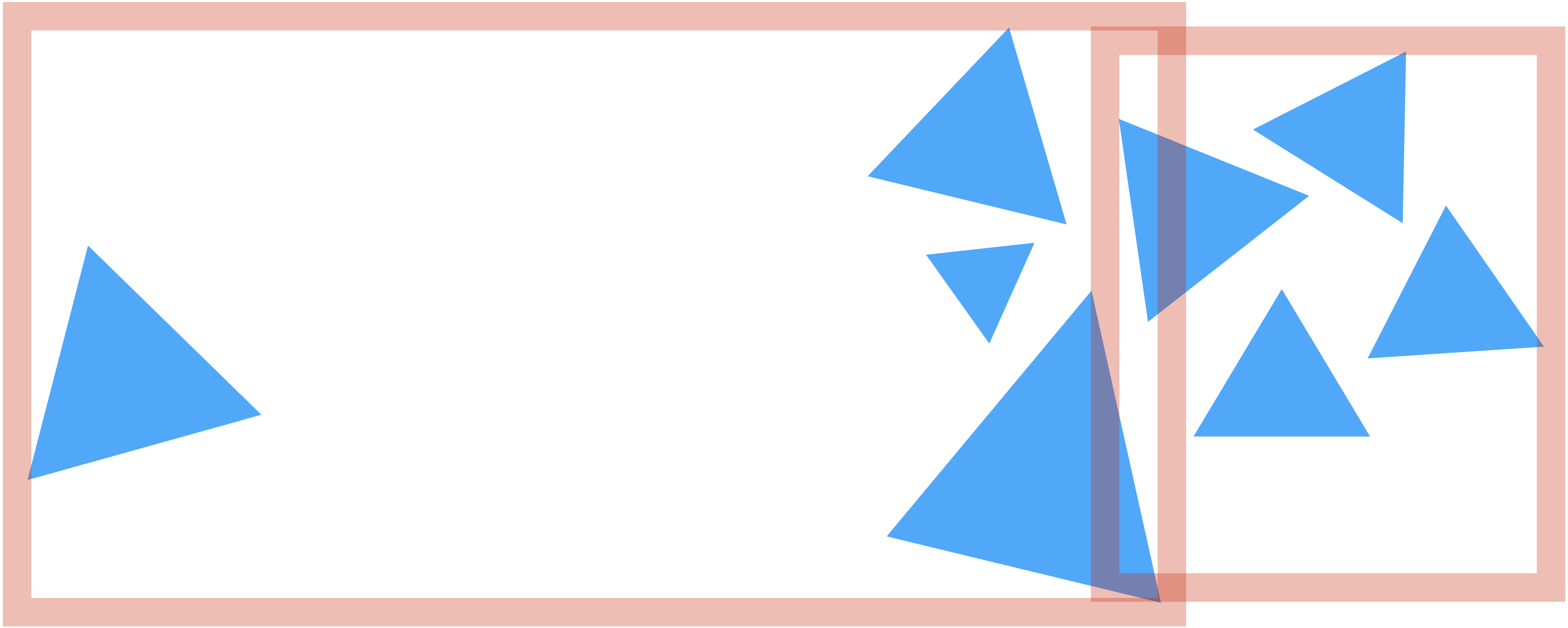
How would you partition these triangles into two groups?



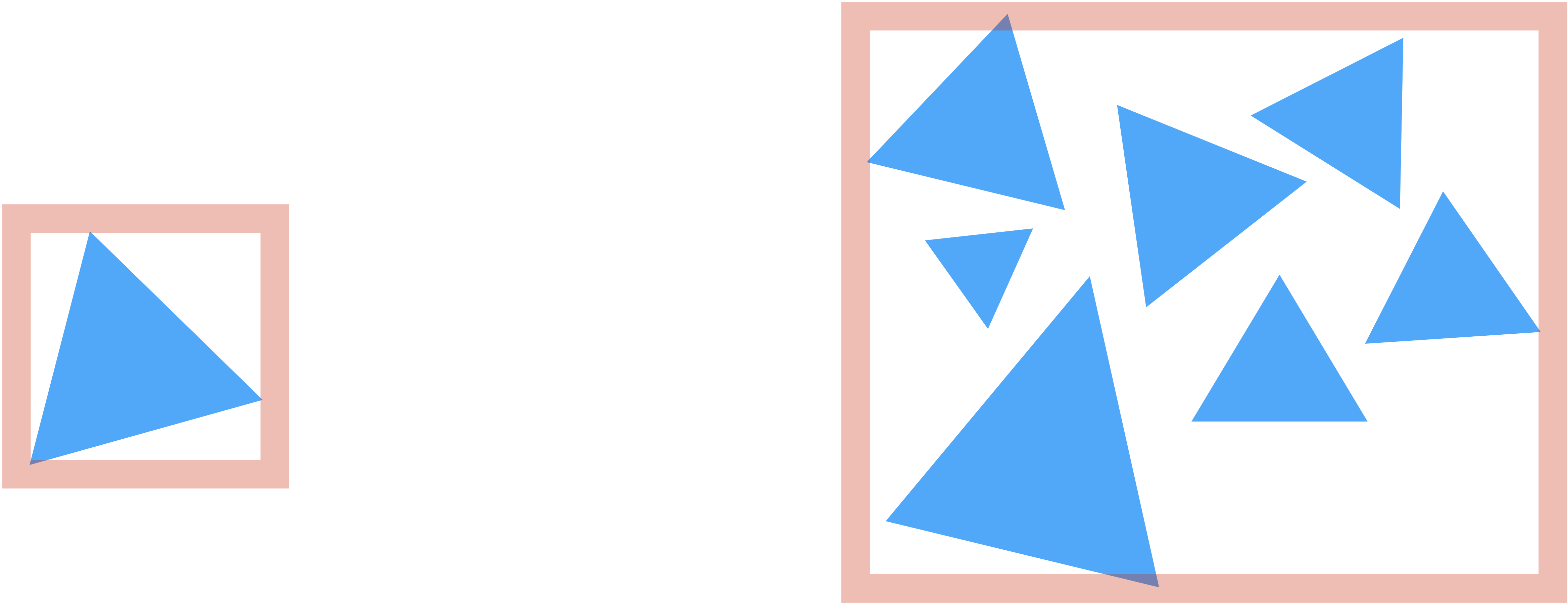
What about these?



Intuition about a "good" partition?



Partition into child nodes with equal numbers of primitives

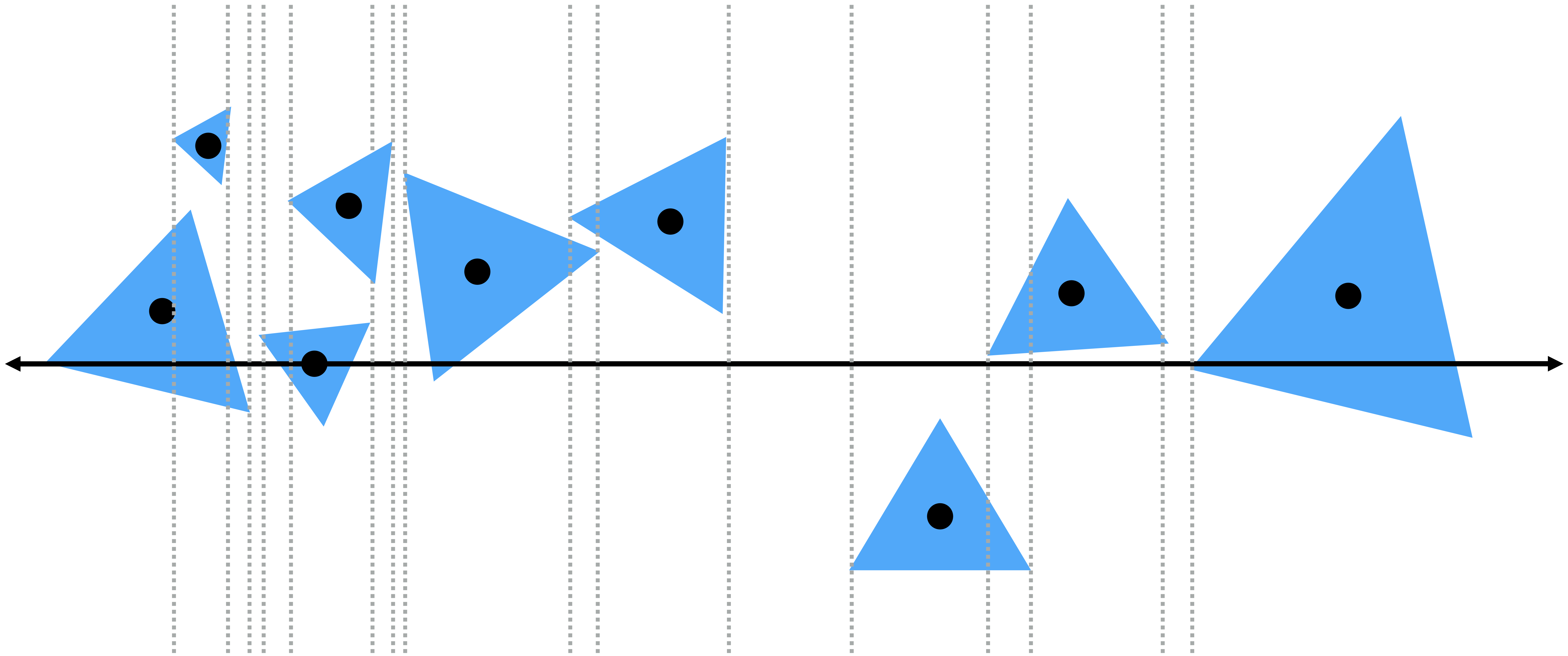


Minimize overlap between children, avoid empty space

Implementing partitions

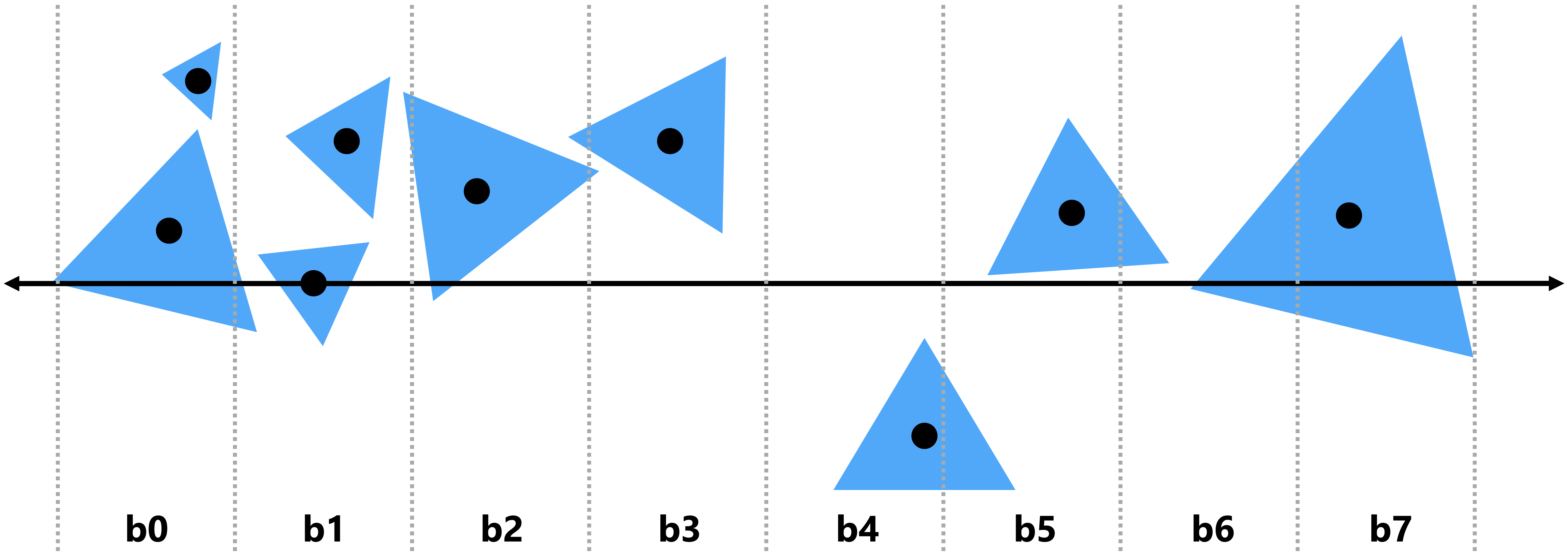
Constrain search for good partitions to axis-aligned spatial partitions

- **Choose an axis**
- **Choose a split plane on that axis**
- **Partition primitives by the side of splitting plane their centroid lies**



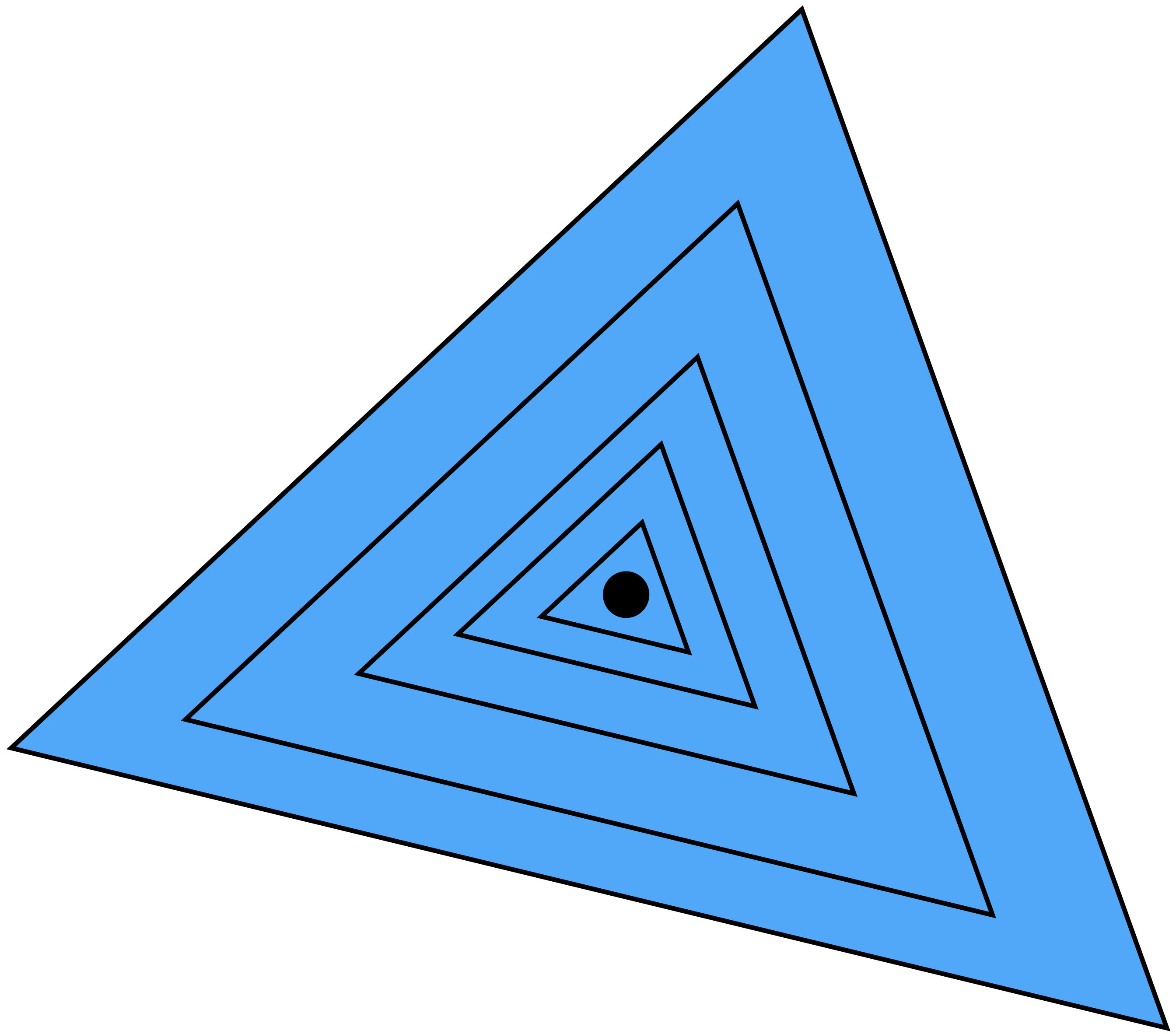
Efficiently implementing partitioning

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

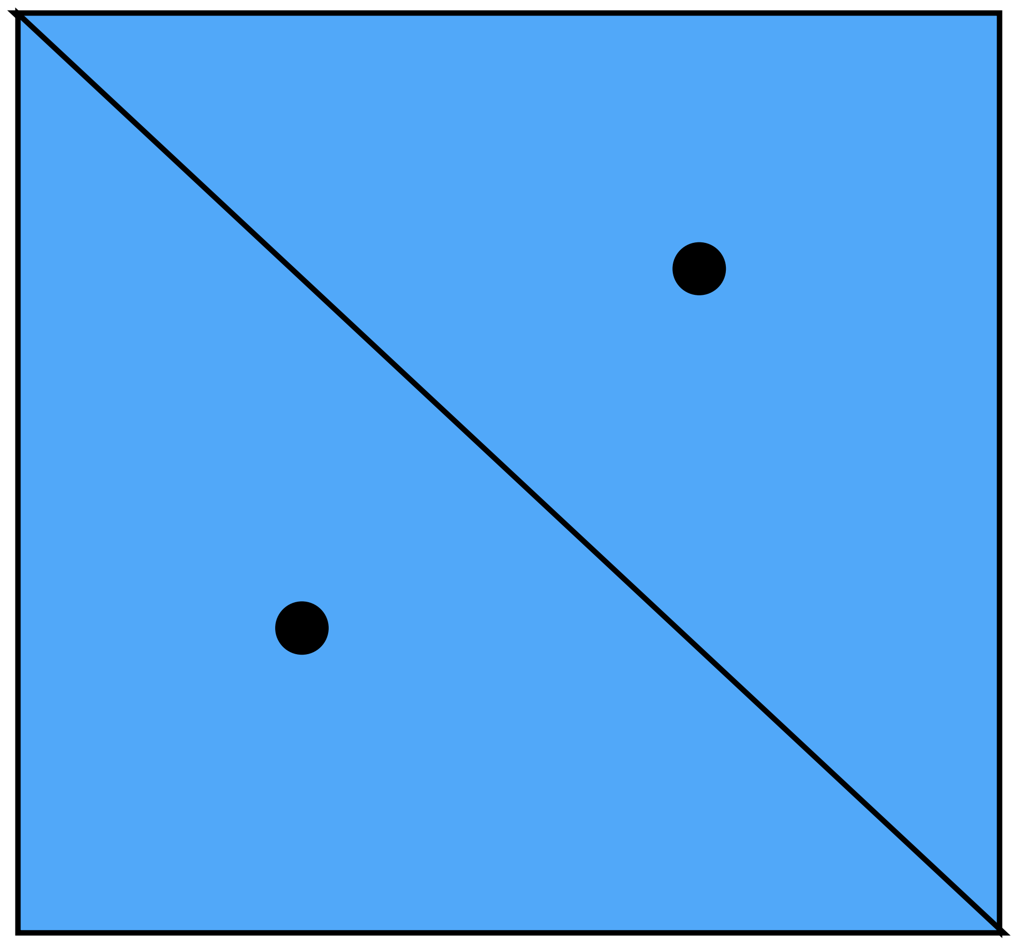



```
For each axis: x,y,z:
  initialize buckets
  For each primitive p in node:
    b = compute_bucket(p.centroid)
    b.bbox.union(p.bbox);
    b.prim_count++;
  For each of the B-1 possible partitioning planes evaluate "goodness" heuristic
Execute lowest cost partitioning found (or make node a leaf)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

Troublesome cases

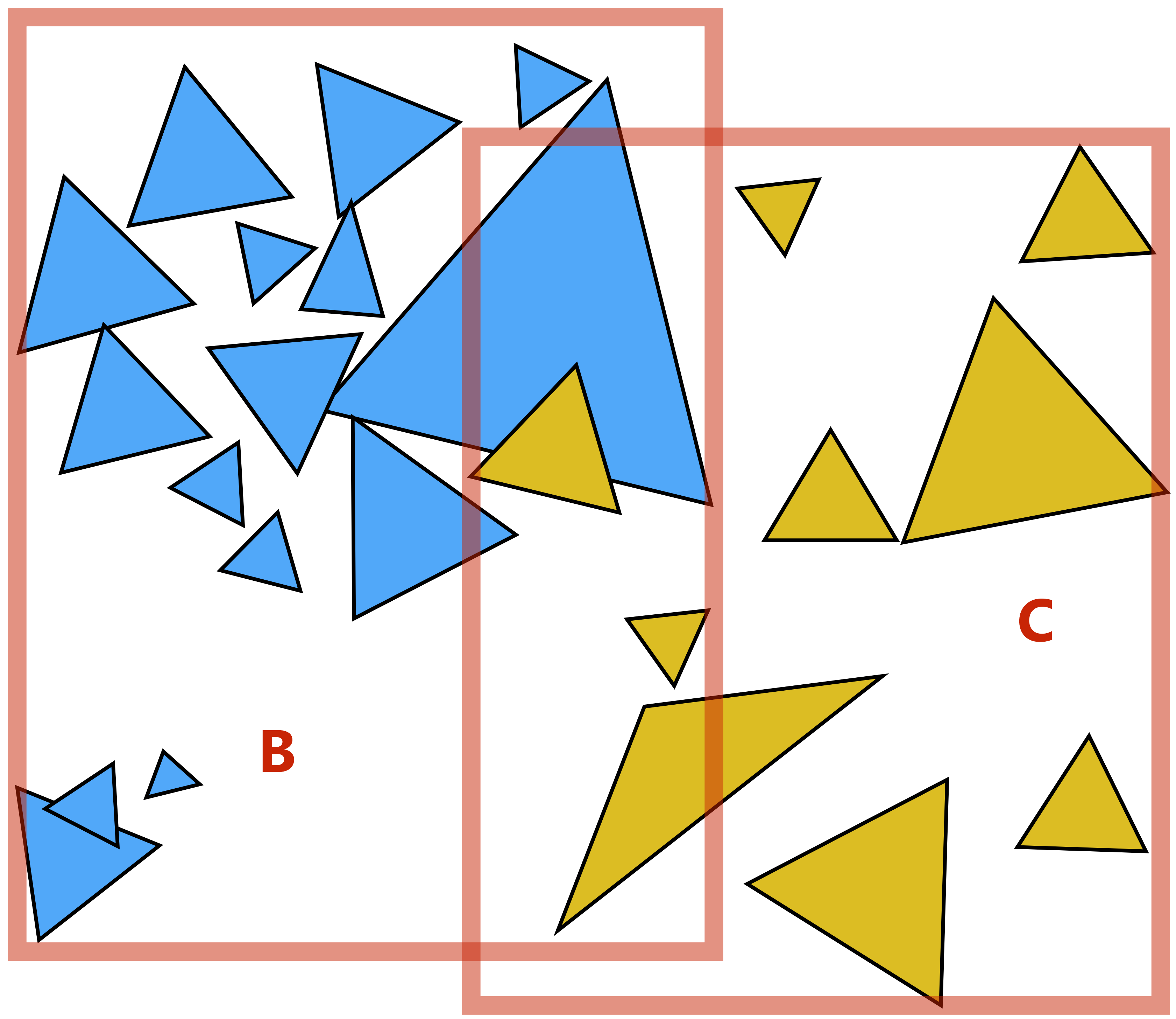


All primitives with same centroid (all primitives end up in same partition)



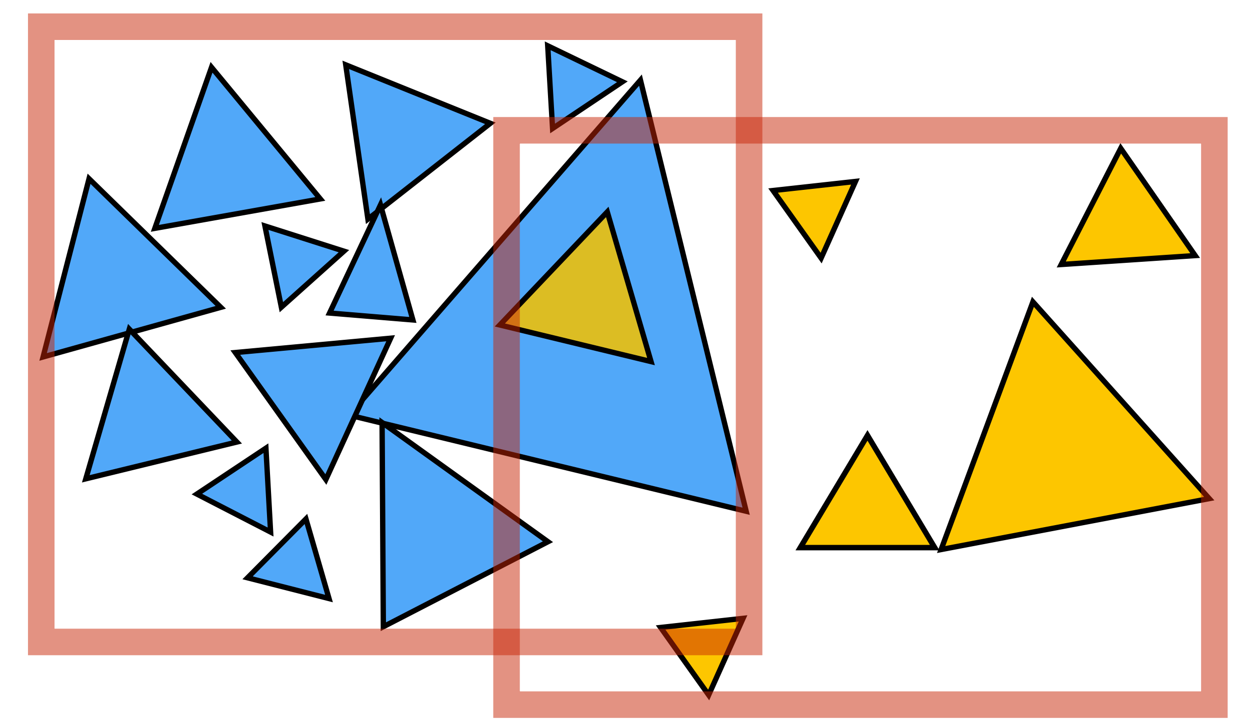
All primitives with same bbox (ray often ends up visiting both partitions)

Primitive-partitioning acceleration

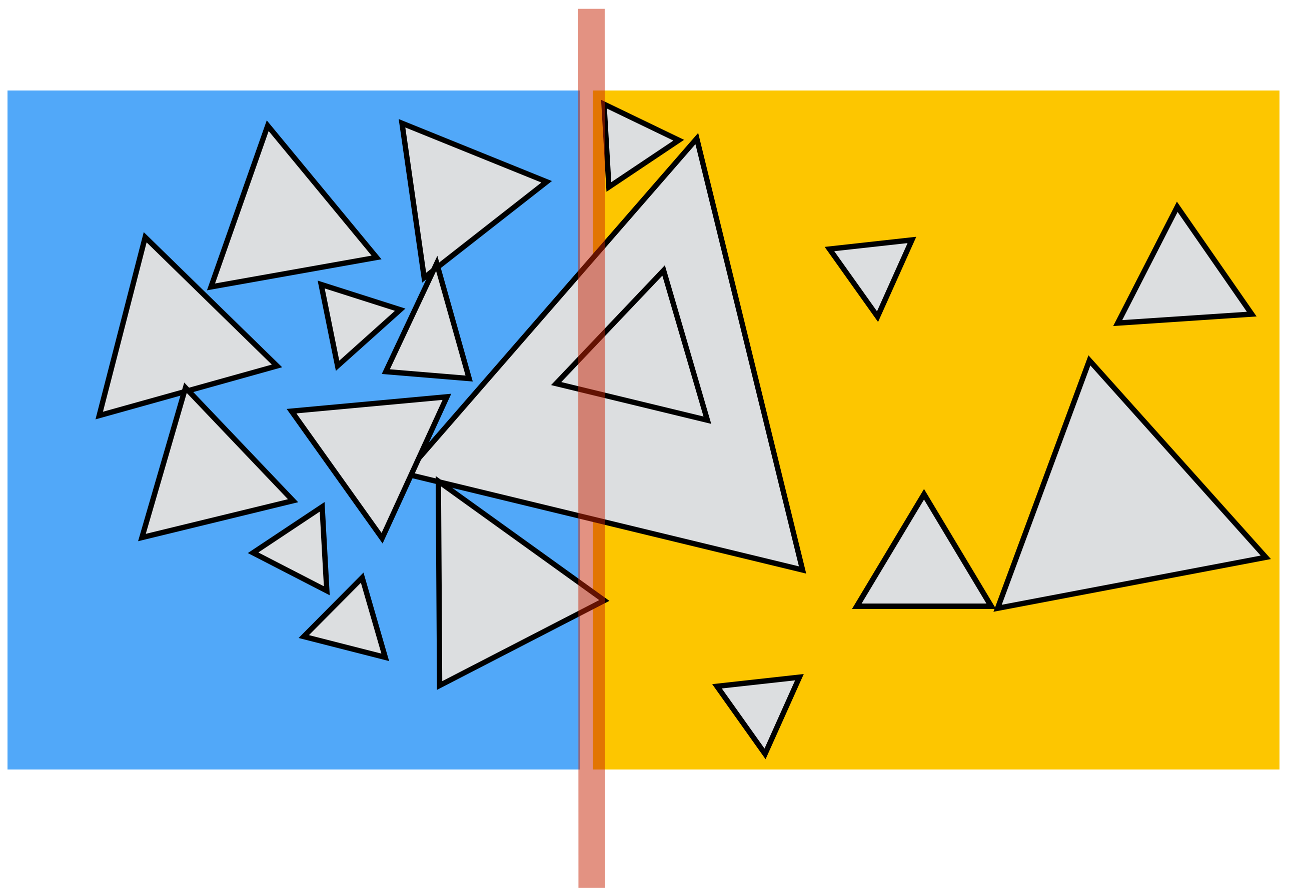


Primitive-partitioning acceleration structures vs. space-partitioning structures

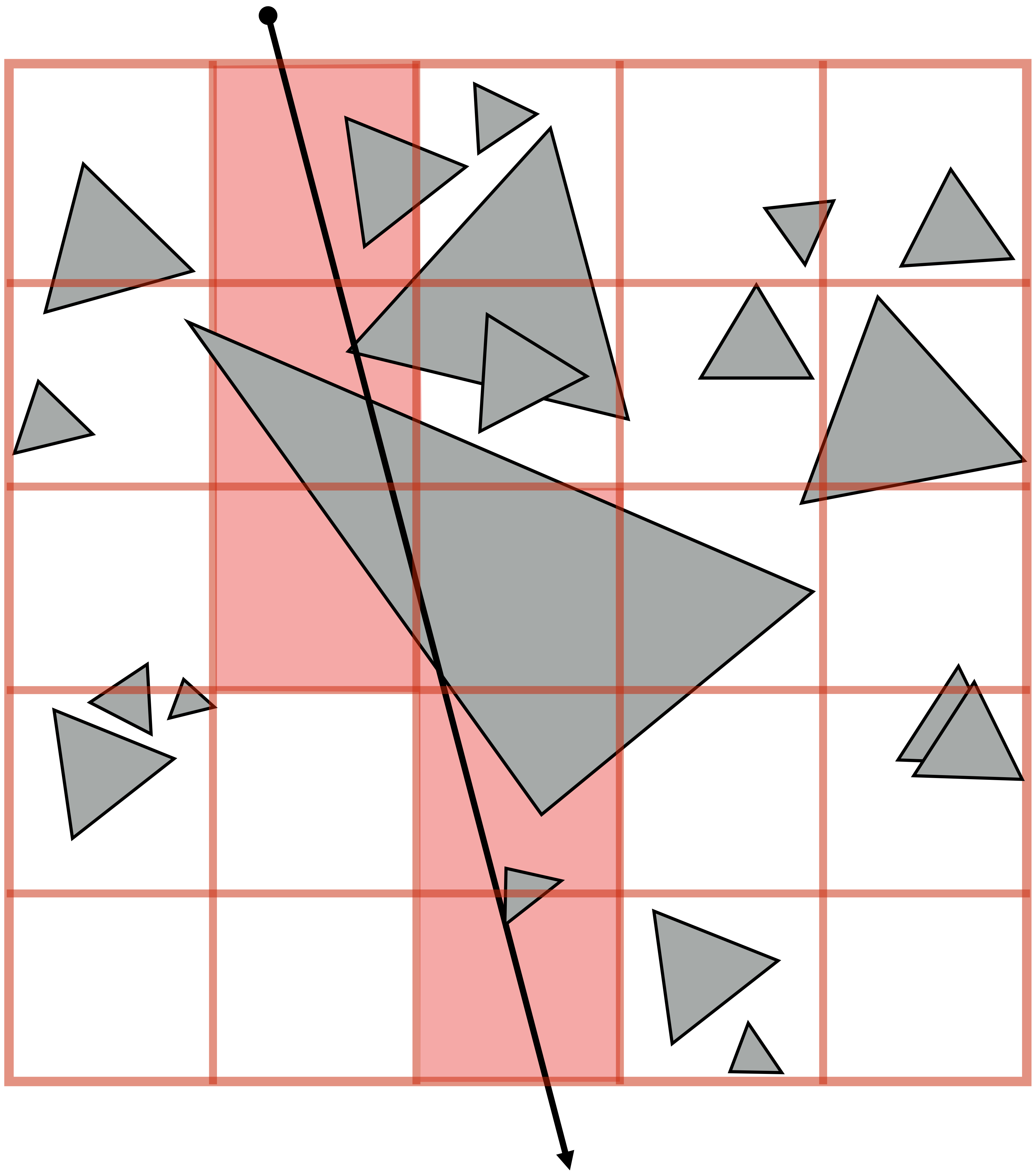
- **Primitive partitioning (bounding volume hierarchy): partitions node's primitives into disjoint sets (but sets may overlap in space)**



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

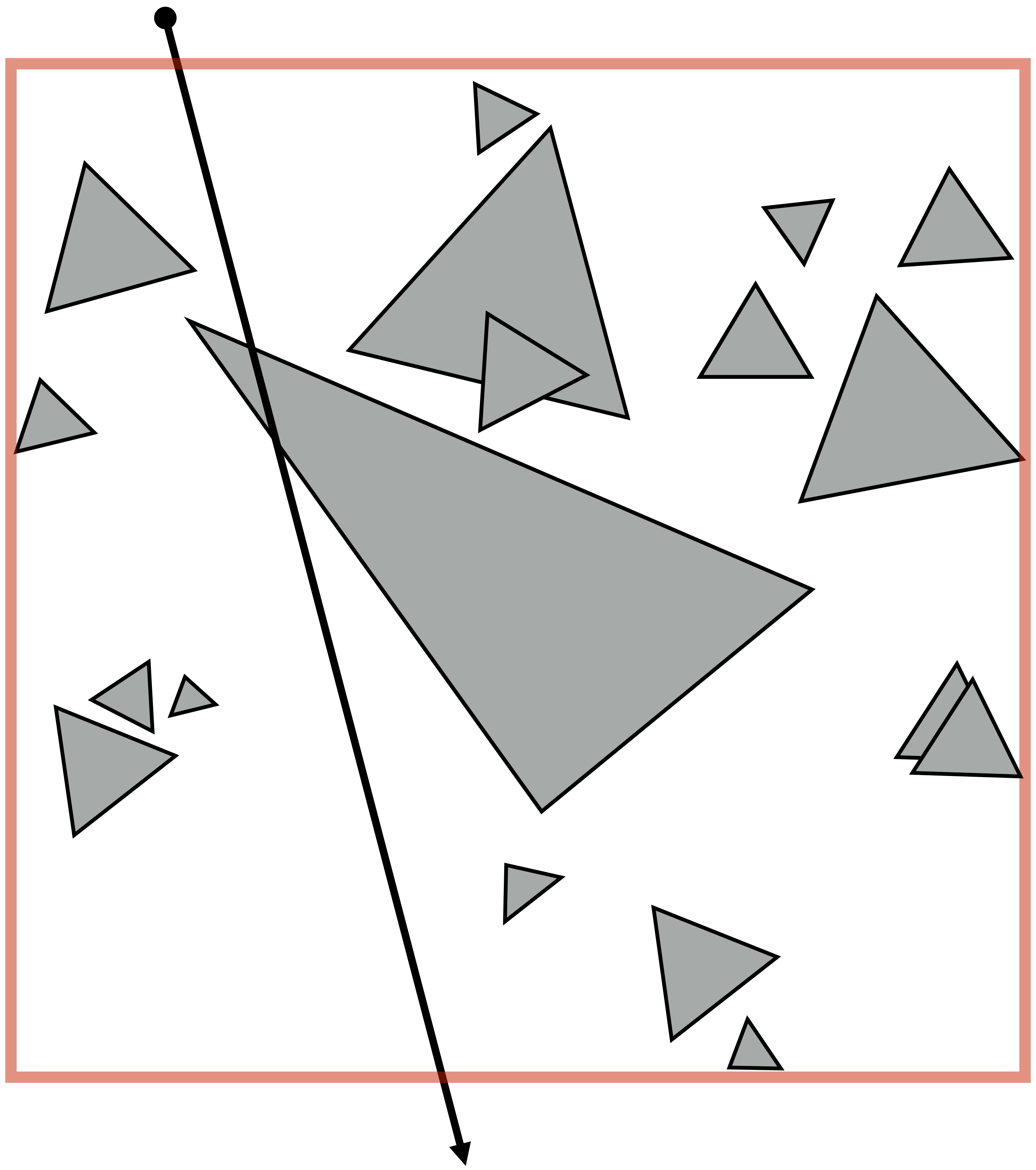


Uniform grid

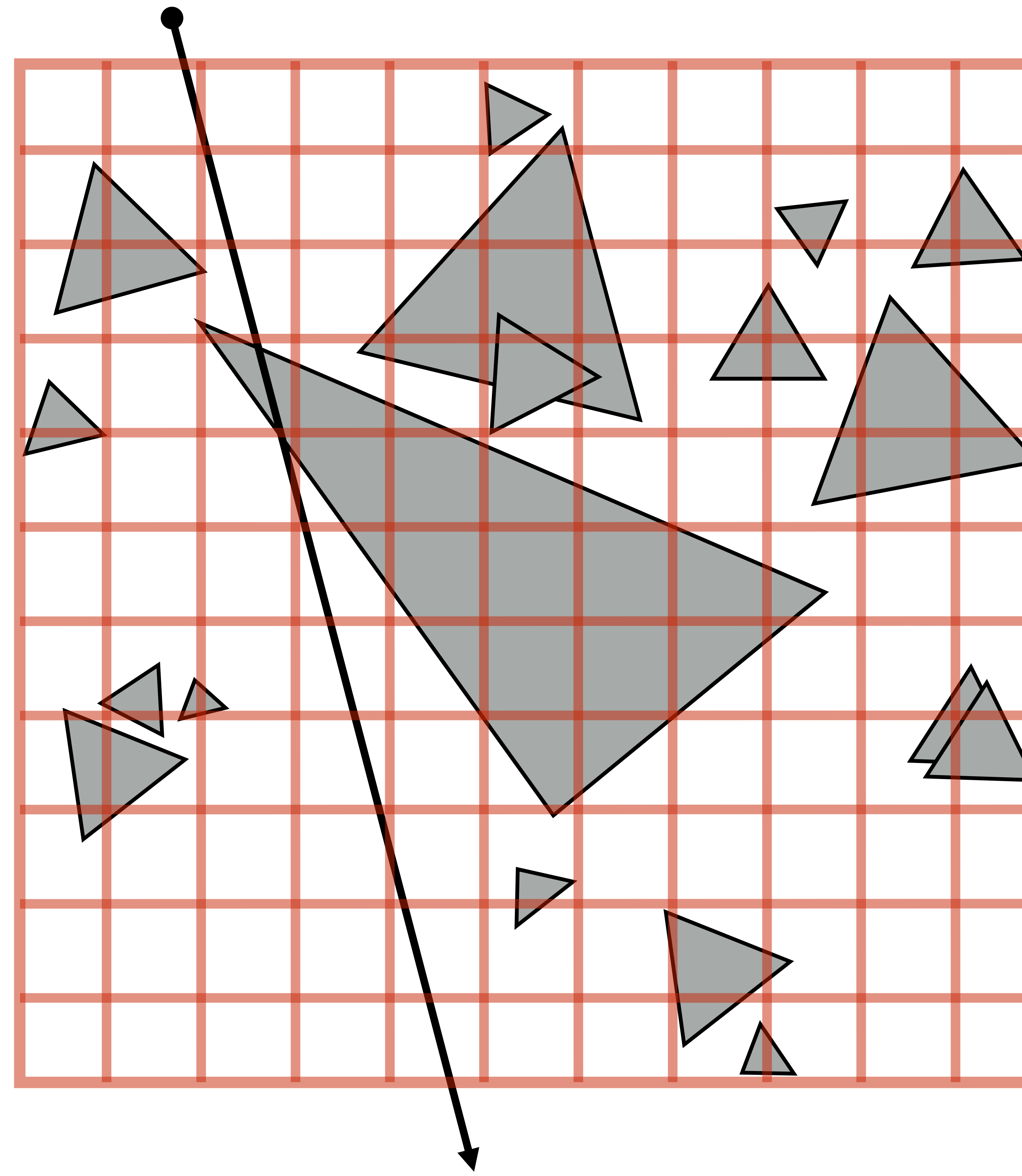


- Partition space into equal sized volumes ("voxels")
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with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

What should the grid resolution be?



Too few grids cell: degenerates to brute-force approach

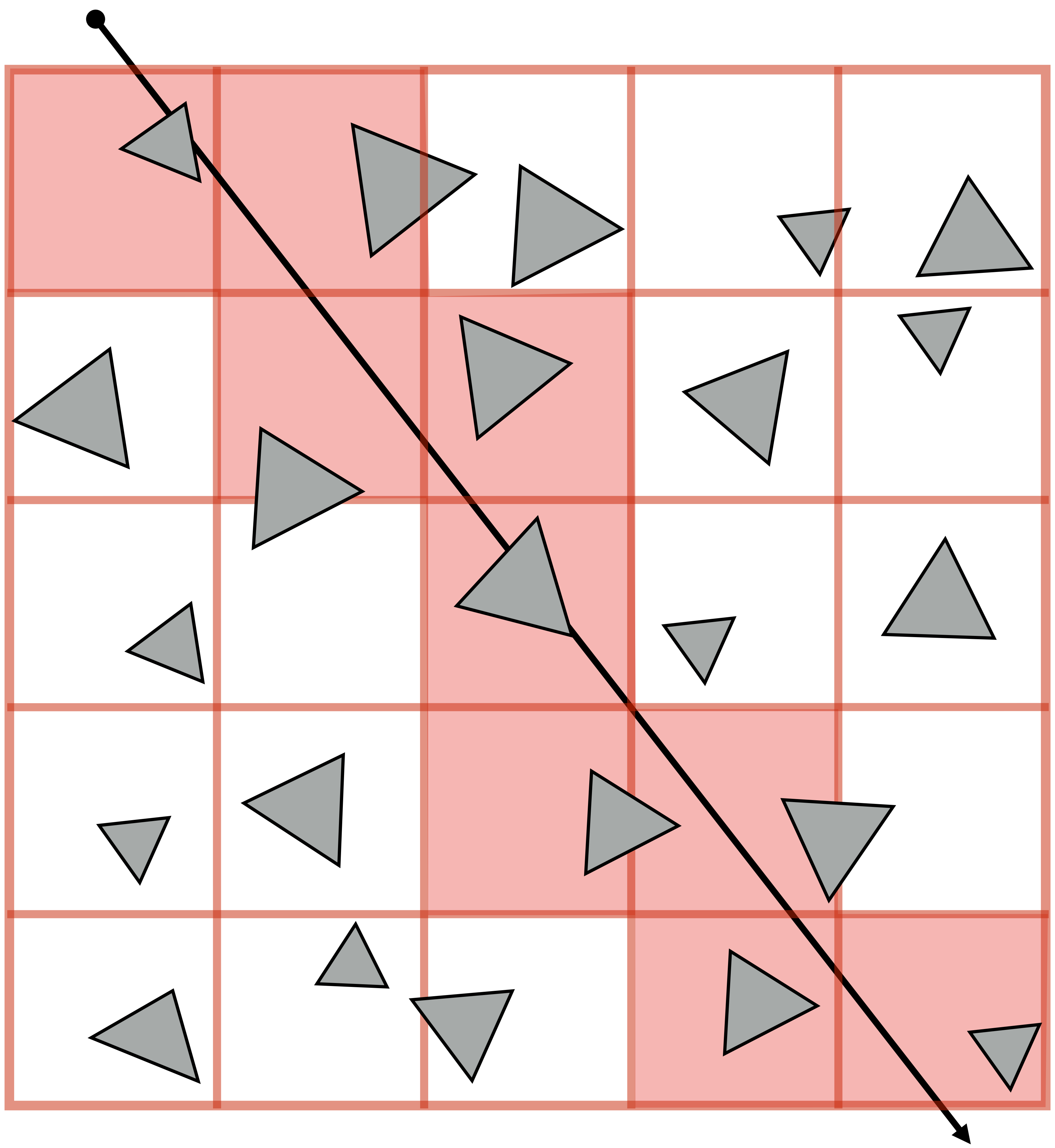


Too many grid cells: incur significant cost traversing through cells with empty space

Heuristic

Choose number of voxels ~ total number of primitives

(constant prims per voxel, assuming uniform distribution)



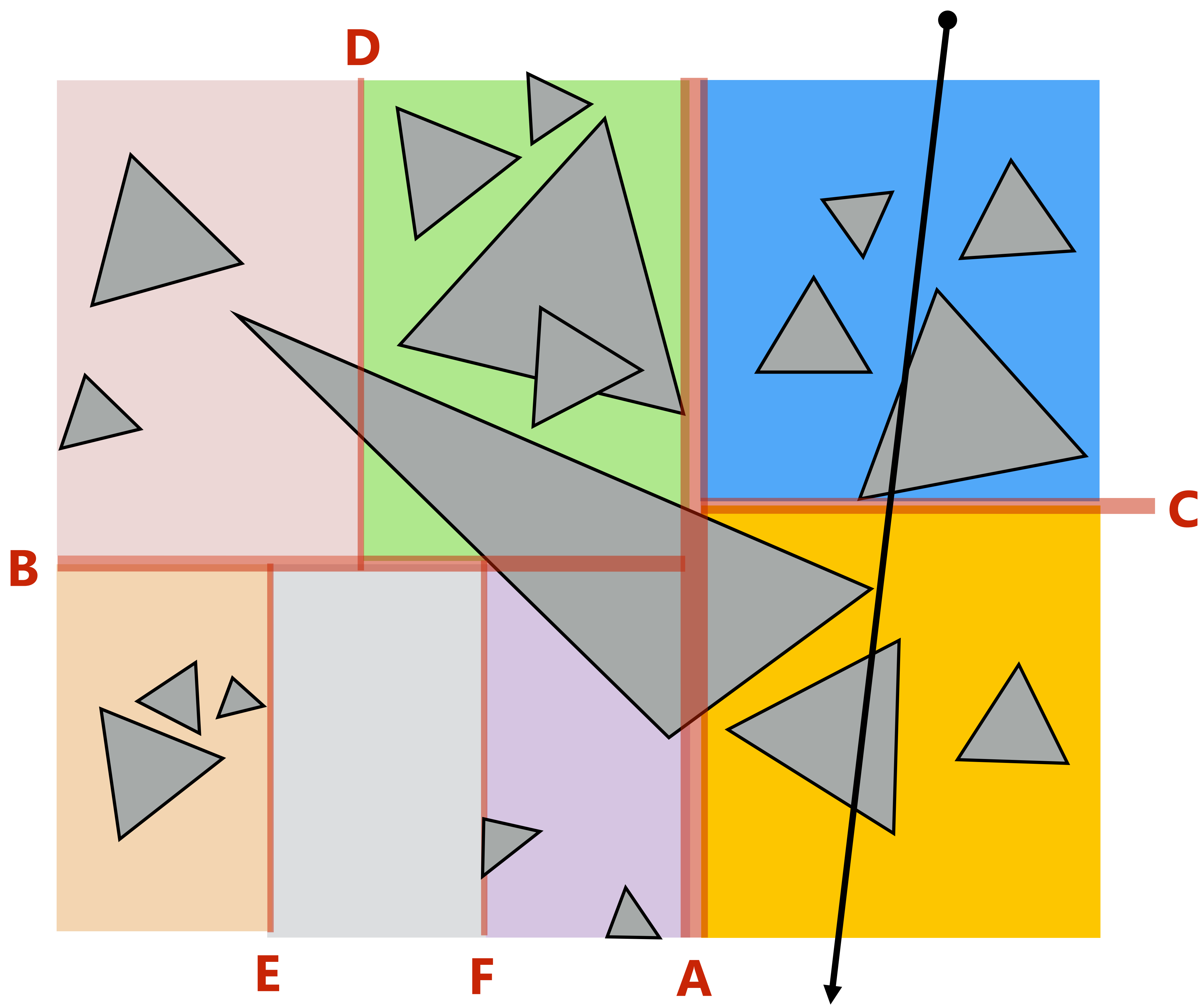
Intersection cost: $O(\sqrt[3]{N})$

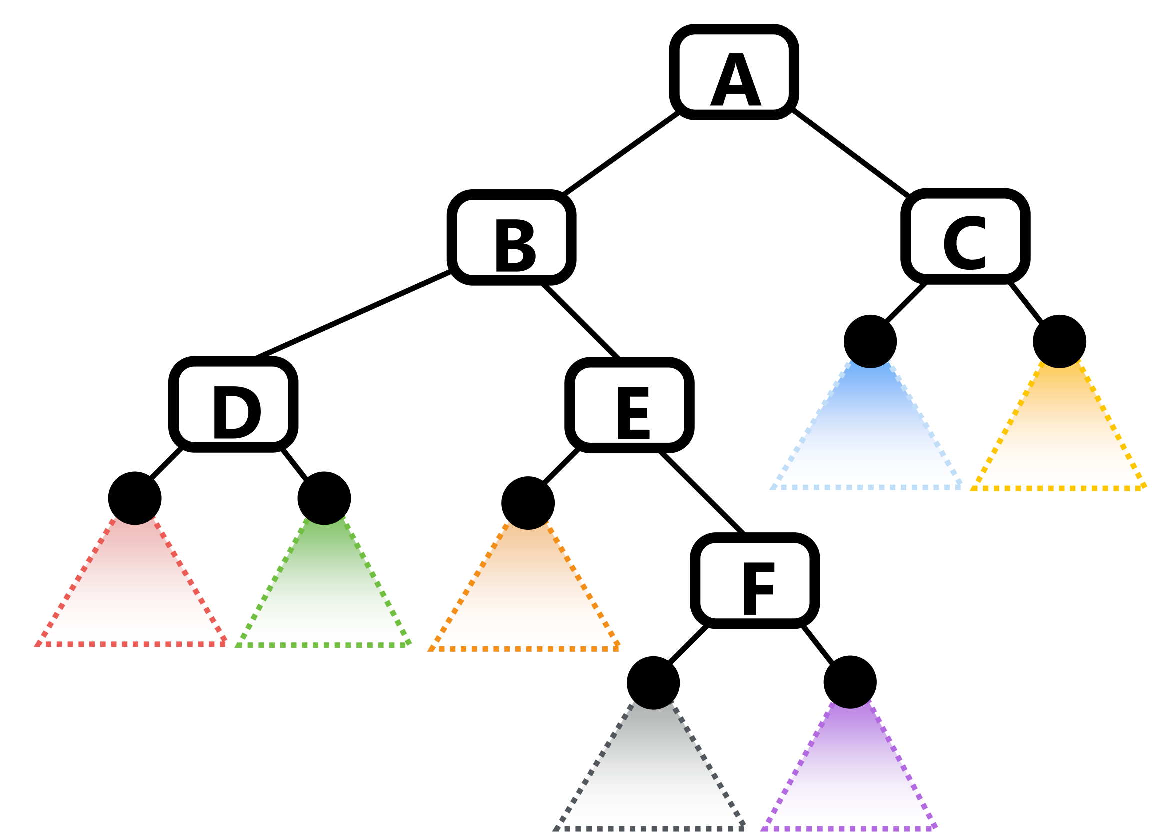
Non-uniform distribution of geometric detail requires adaptive grids

[Image credit: Pixar]

K-D trees

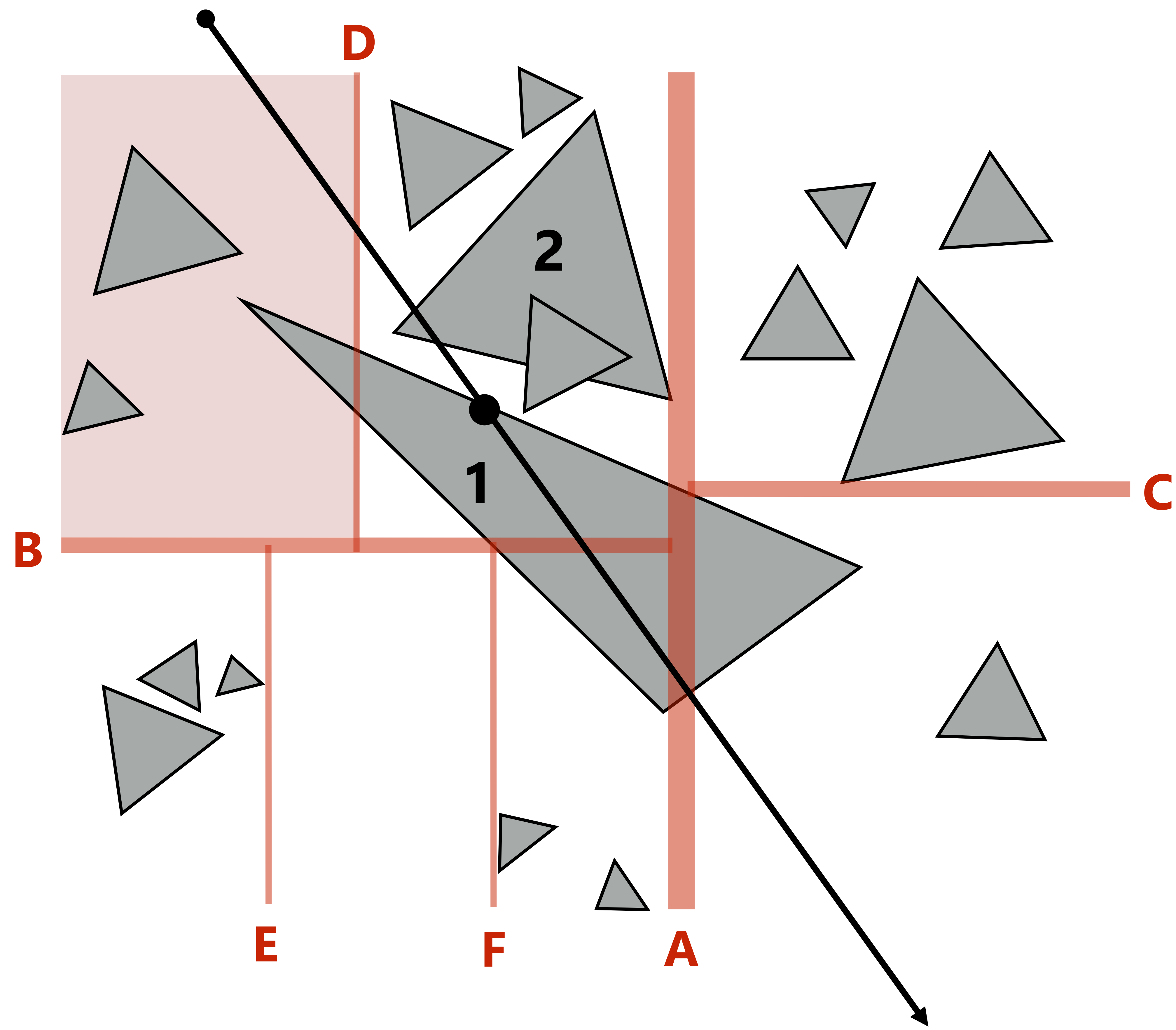
- Recursively partition <u>space</u> via axis-aligned planes
  - Interior nodes correspond to spatial splits (still correspond to spatial volume)
  - Node traversal can proceed in front-to-back order (unlike BVH, can terminate search after first hit is found*).

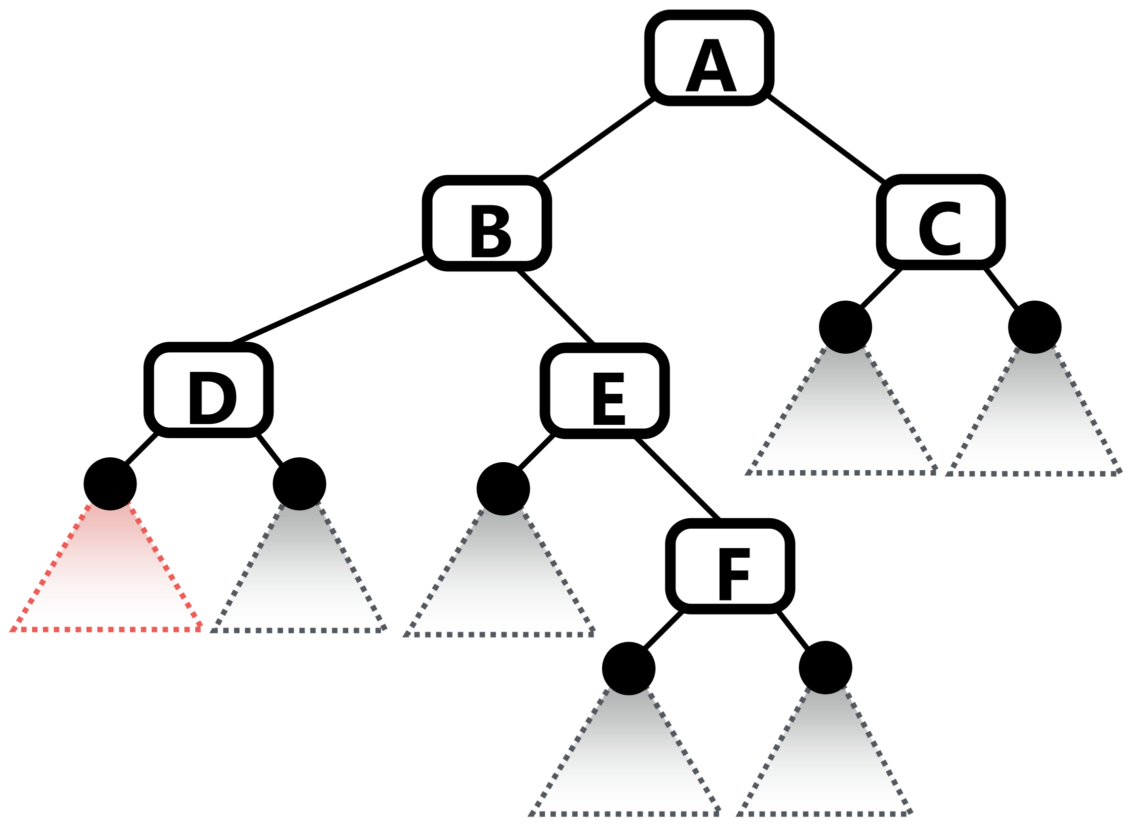




Challenge: objects overlap multiple nodes

- **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)

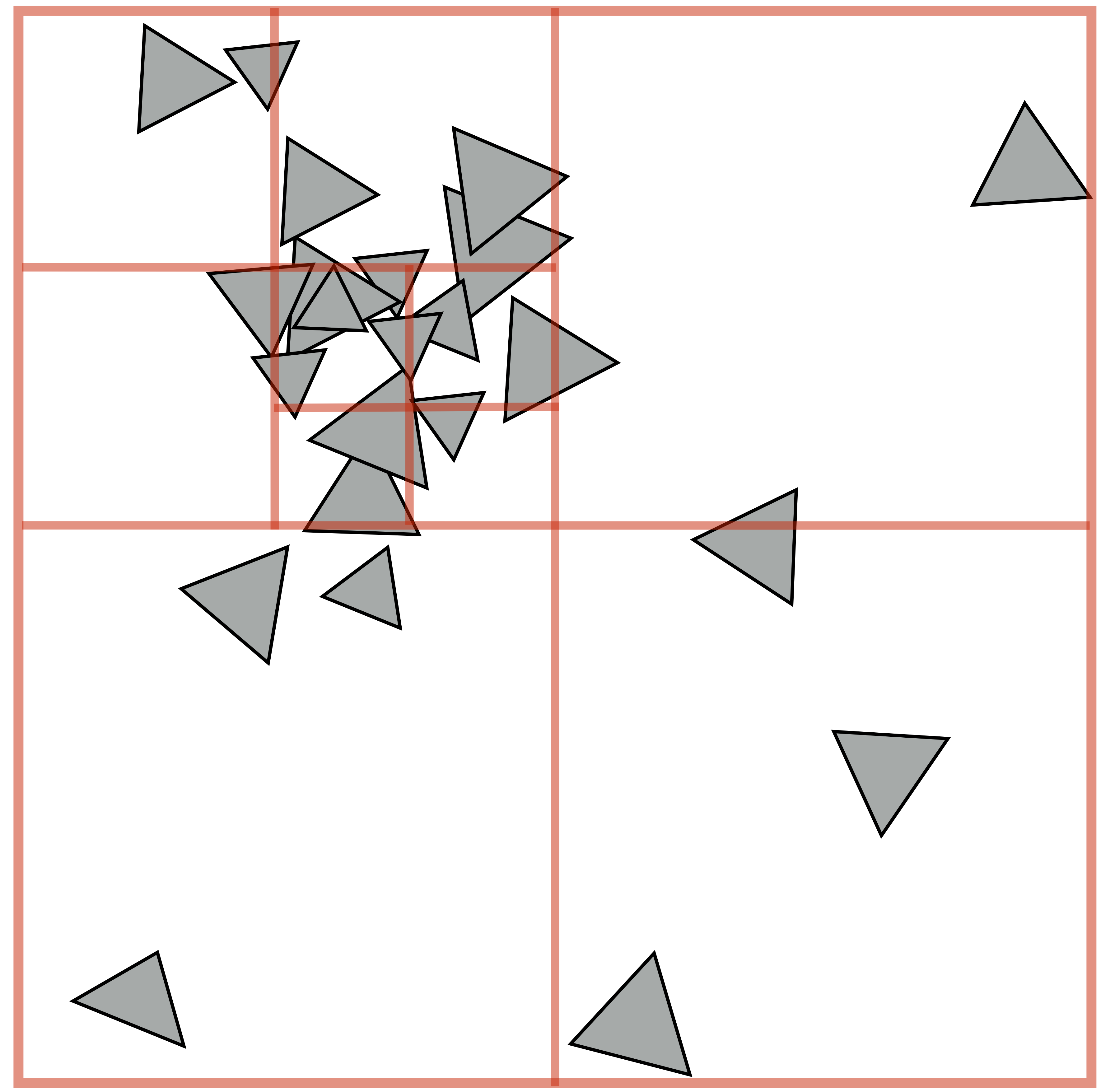
Solution: require primitive intersection point to be within current leaf node. (primitives may be intersected multiple times by same ray *)

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 accelerating geometric queries: 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 construction 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