Lecture 14:

Real Time Ray Tracing Workload

Visual Computing Systems
Stanford CS348K, Spring 2021

So far in class

- Computational photography algorithms and their mapping to efficient systems (plus abstractions for expressing and scheduling these algorithms

- Deep learning workloads and their mapping to efficient systems
  - And the design of specialized hardware for DNN workloads
  - And discussions of where abstractions/system support might be lacking

- Video compression and video conferencing workloads and systems

This image was rendered in real-time on a single high-end GPU

So was this

Real-time ray tracing

- **Exciting example of co-design of algorithms, specialized hardware, and software abstractions**

- **It's becoming increasingly clear that the immediate future of real-time graphics will involve large amounts of ray tracing**

NVIDIA GeForce RTX 3080 GPU

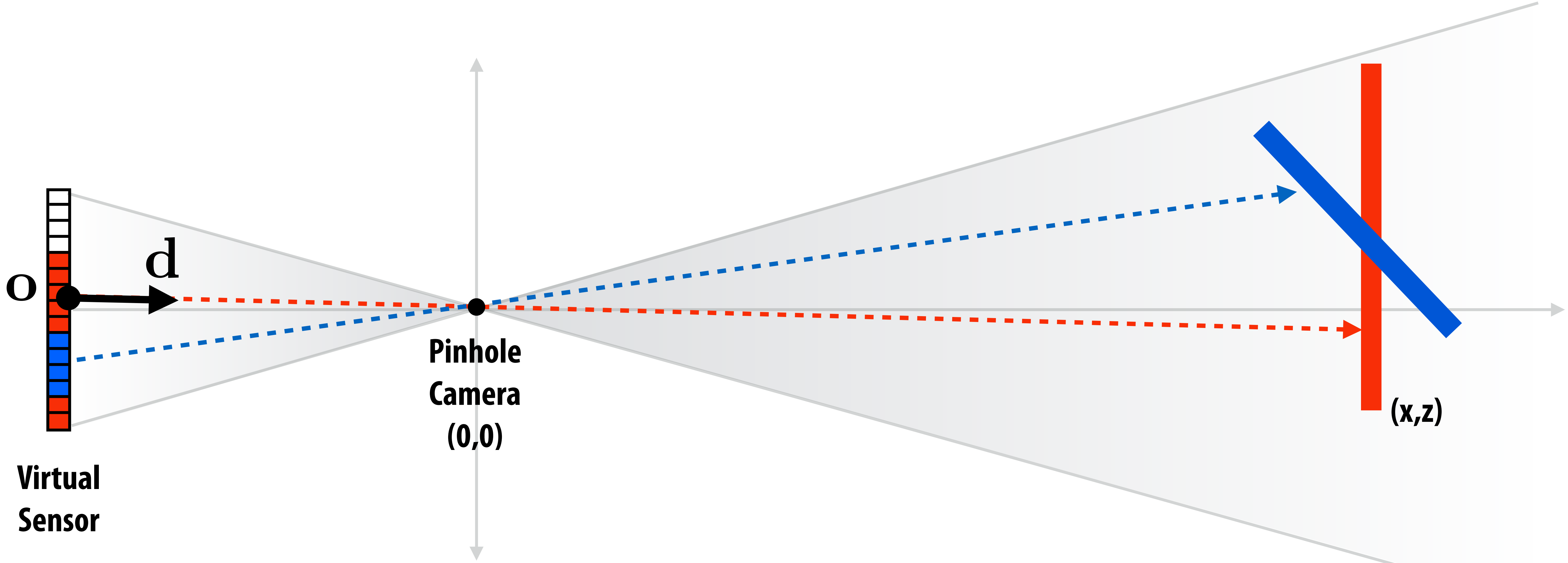
Ray tracing in one class


RAY TRACING
IN ONE WEEKEND
PETER SHIRLEY

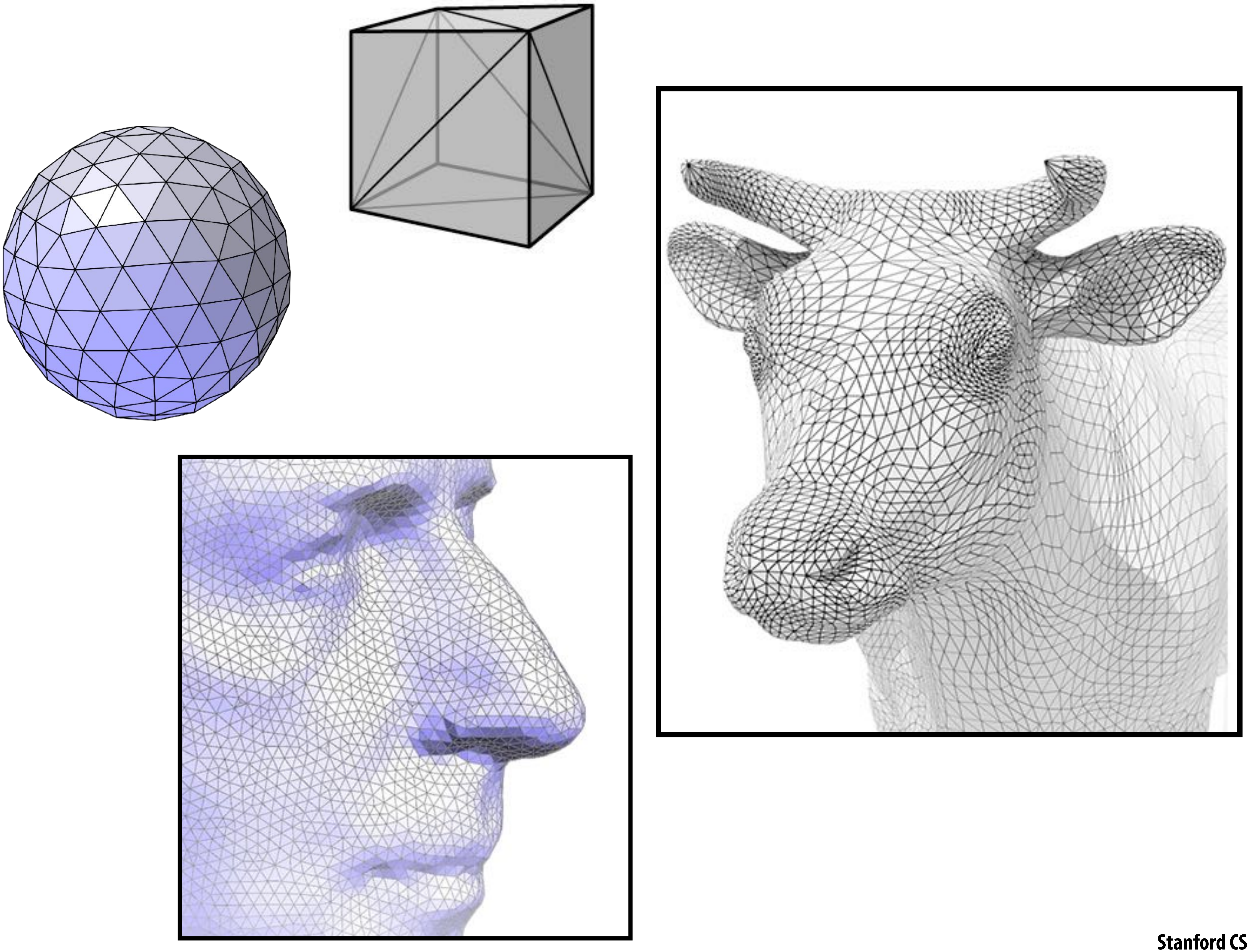
Take that Pete Shirley!

The “visibility problem” in computer graphics

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



In this class: scene geometry = triangles



Basic "ray casting" algorithm to render a picture

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."

Does a ray (in 3D) hit a triangle (in 3D)?

Ray equation

- **Can express ray as:**

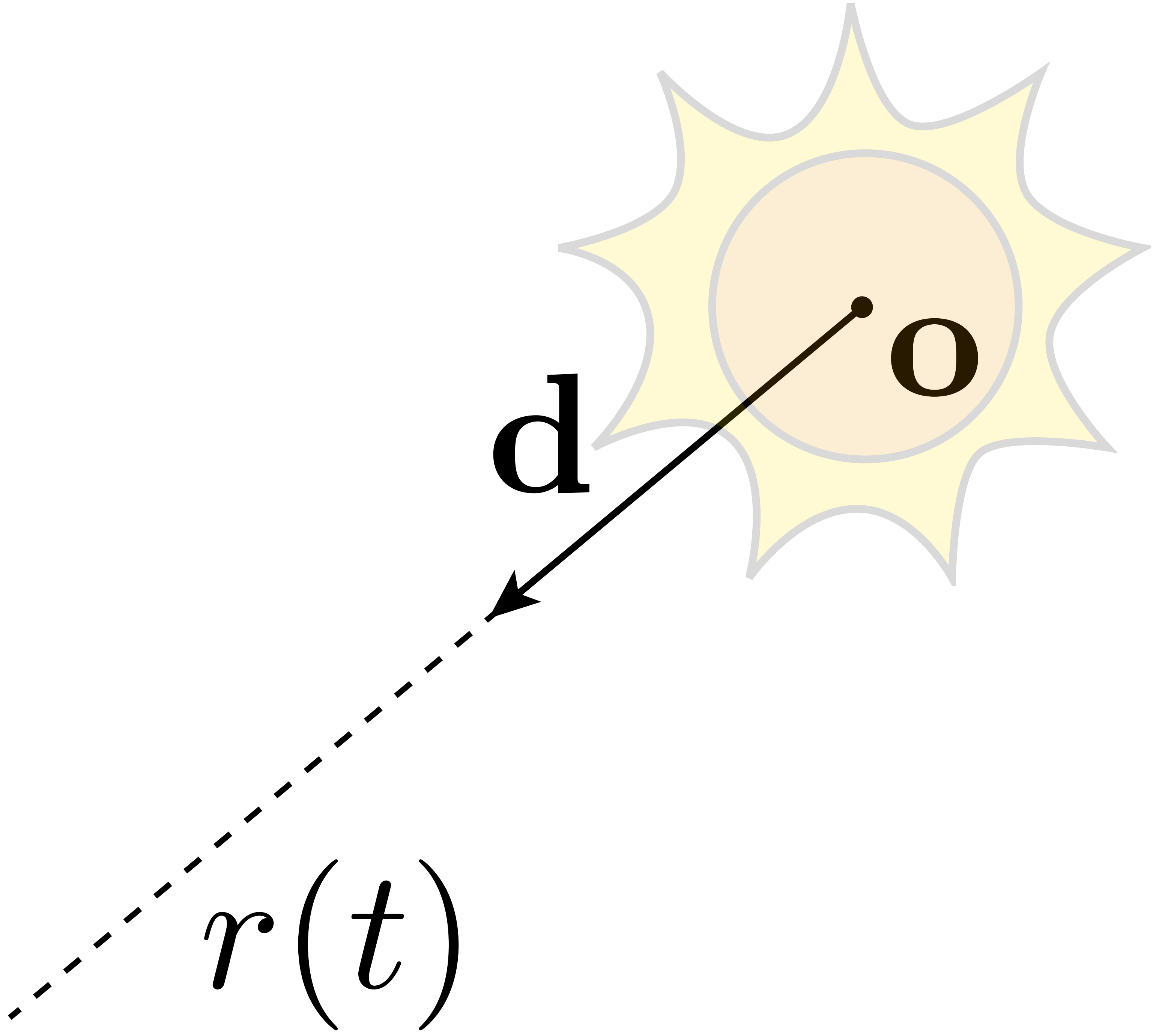
origin

unit direction

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

point along ray

"time"



Review: matrix form of a line (and a plane)

Line is defined by:

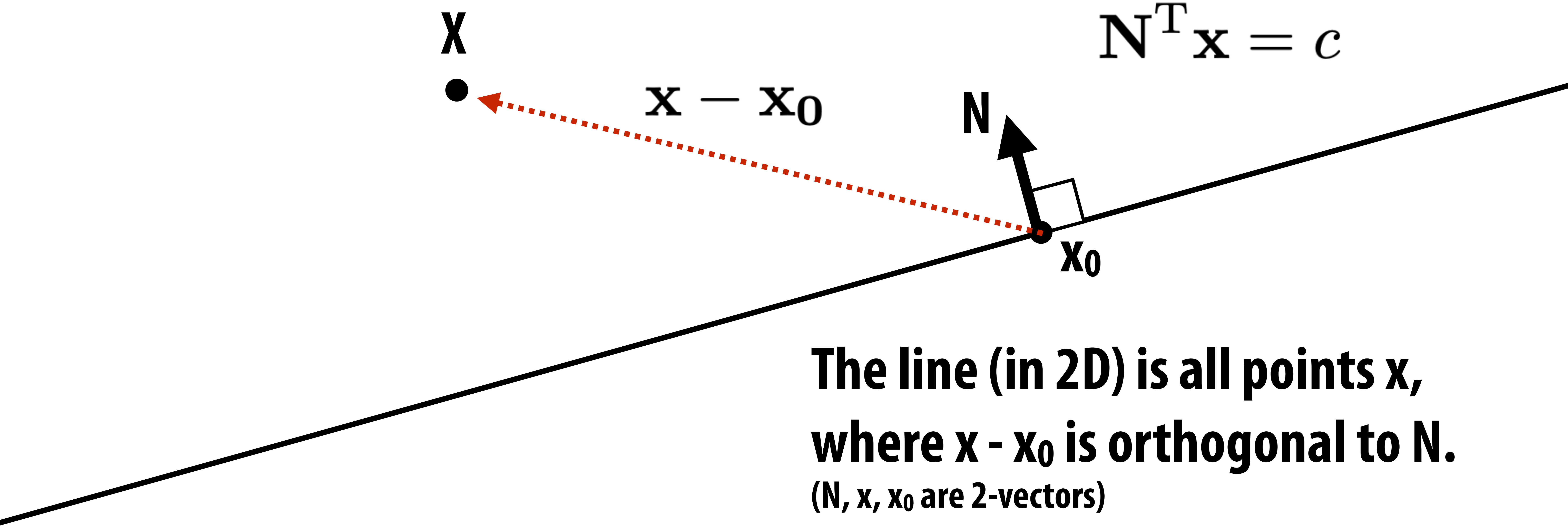
- Its normal: N
- A point x_0 on the line

$$\mathbf{N} \cdot (\mathbf{x} - \mathbf{x_0}) = 0$$

$$\mathbf{N}^{\mathrm{T}}(\mathbf{x} - \mathbf{x_0}) = 0$$

$$\mathbf{N}^{\mathrm{T}}\mathbf{x} = \mathbf{N}^{\mathrm{T}}\mathbf{x_0}$$

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



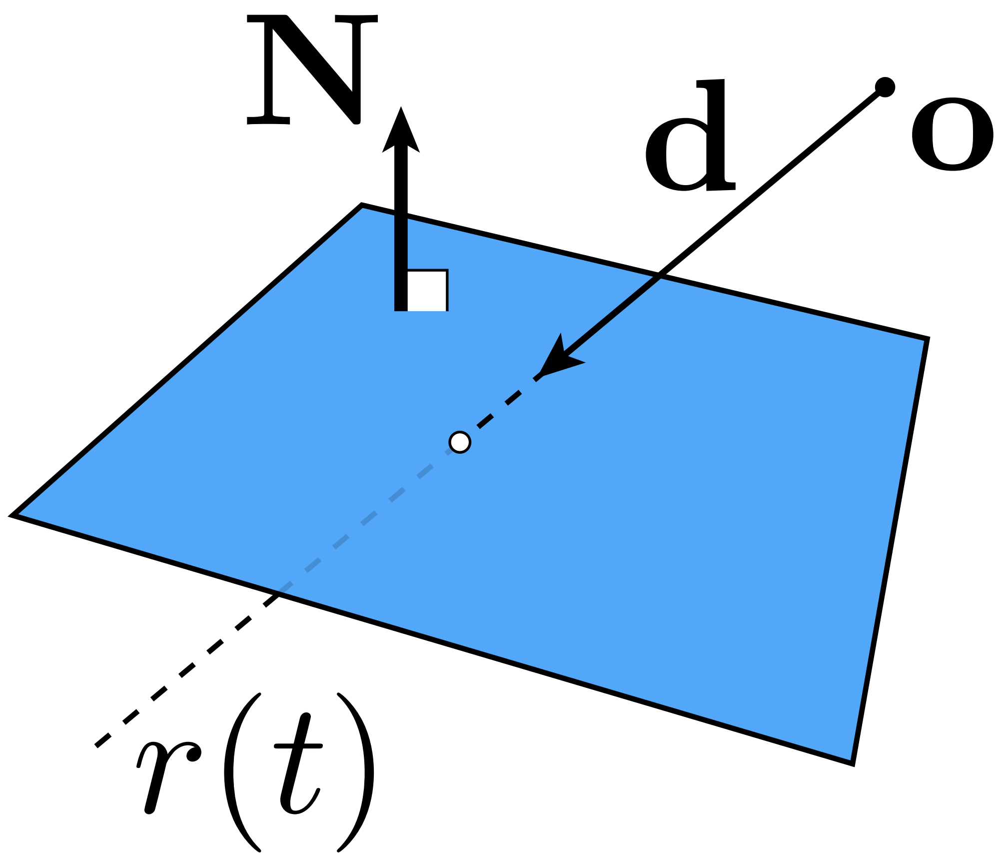
The line (in 2D) is all points x, where x - x_0 is orthogonal to N.

(N, x, x_0 are 2-vectors)

(And a plane (in 3D) is all points x where x - x_0 is orthogonal to N.)

(N, x, x_0 are 3-vectors)

Ray-plane intersection



- Suppose we have a plane $N^Tx = c$
  - N - unit normal
  - c - offset
- How do we find intersection with ray $r(t) = o + td$?
- Replace the point x with the ray equation t:

$$\mathbf{N}^\mathsf{T}\mathbf{r}(t) = c$$

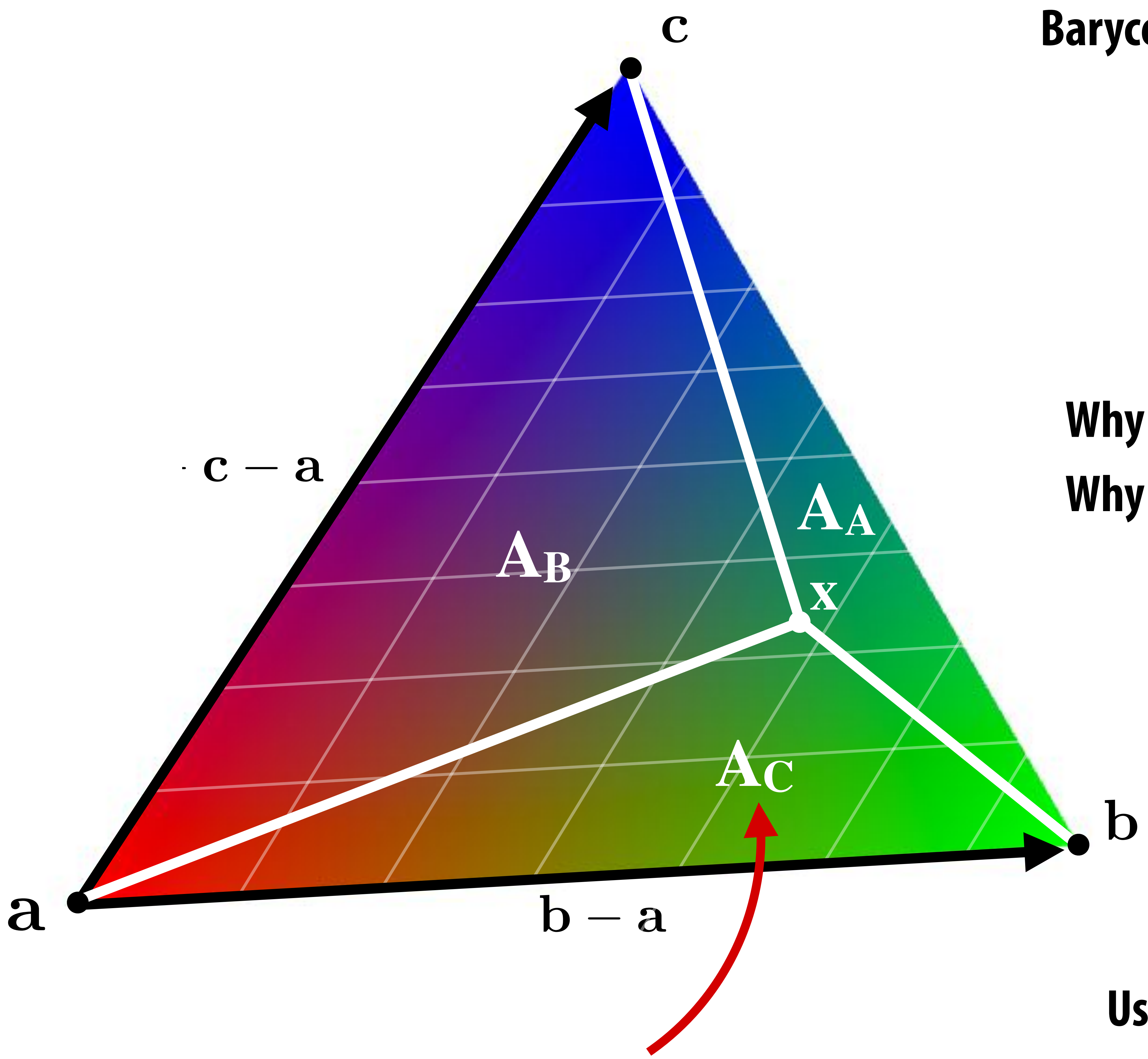
- Now solve for t:

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

- And plug t back into ray equation:

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

Barycentric coordinates (as ratio of areas)



Area of triangle formed by points: a, b, x

Barycentric coords are *signed* areas:

$$\alpha = A_A/A$$

$$\beta = A_B/A$$

$$\gamma = A_C/A$$

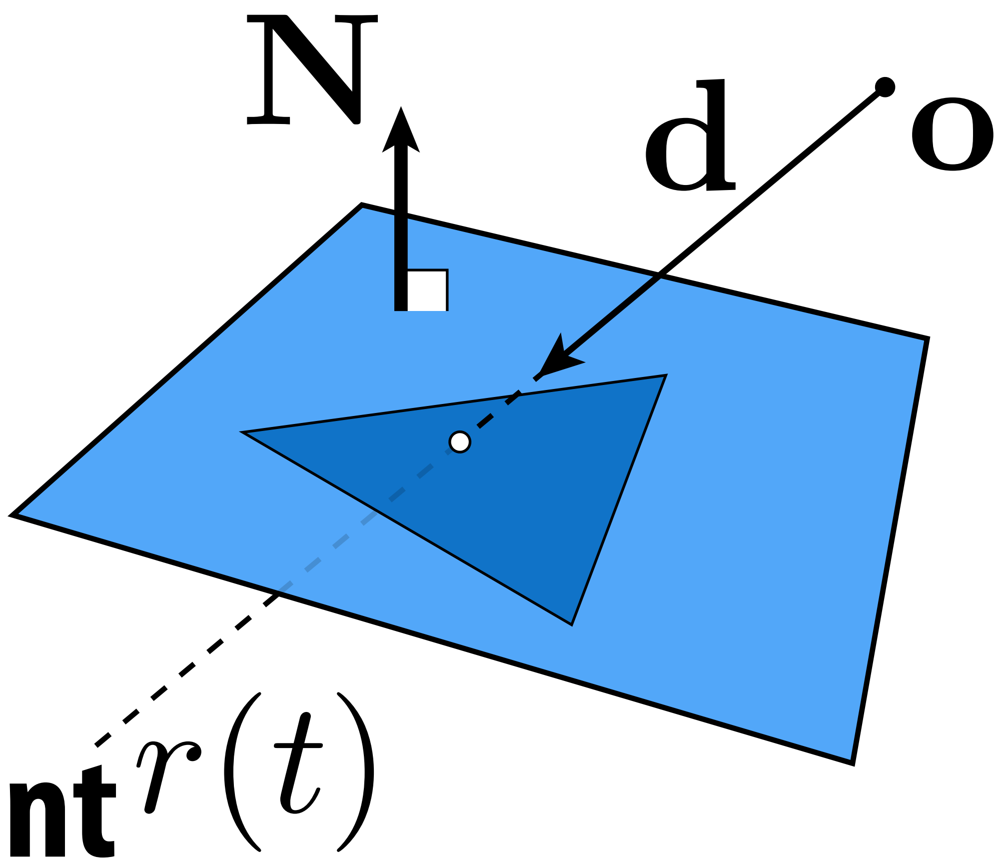
Why must coordinates sum to one?

Why must coordinates be between 0 and 1?

Useful: Heron's formula:

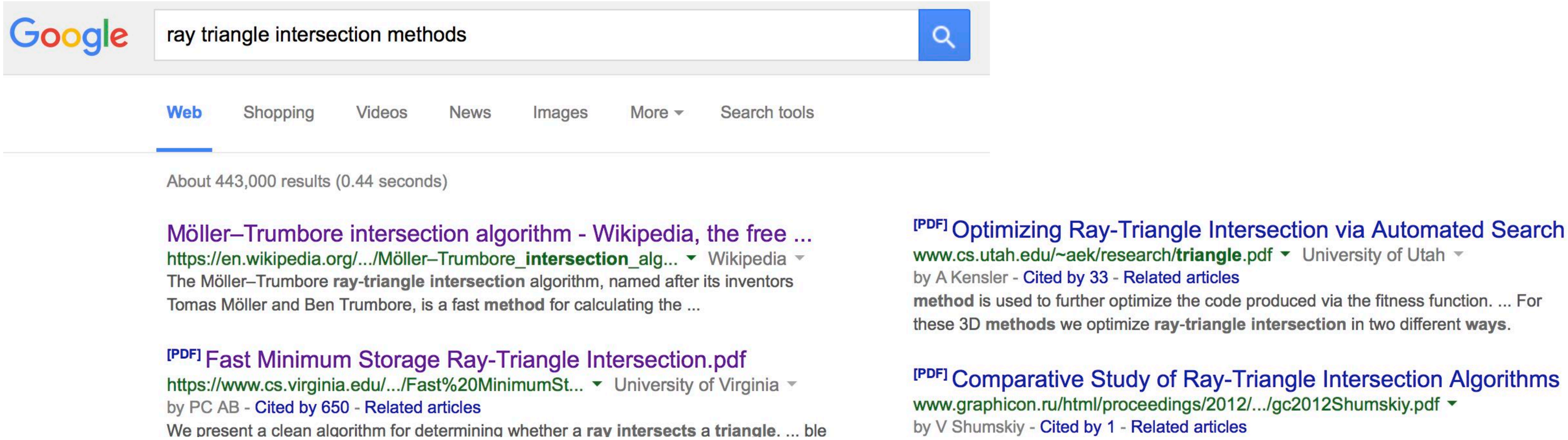
$$A_C = \frac{1}{2}(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a})$$

Ray-triangle intersection



- **Algorithm:**
  - Compute ray-plane intersection
  - Compute barycentric coordinates of hit point
  - If barycentric coordinates are all positive, point is in triangle

- **Many different techniques if you care about efficiency**



Google | ray triangle intersection methods

Web | Shopping | Videos | News | Images | More | Search tools

About 443,000 results (0.44 seconds)

Möller–Trumbore intersection algorithm - Wikipedia, the free ...
https://en.wikipedia.org/.../Möller–Trumbore_**intersection**_alg... Wikipedia
The Möller–Trumbore **ray-triangle intersection** algorithm, named after its inventors Tomas Möller and Ben Trumbore, is a fast **method** for calculating the ...

[PDF] Fast Minimum Storage Ray-Triangle Intersection.pdf
https://www.cs.virginia.edu/.../Fast%20MinimumSt... University of Virginia
by PC AB - Cited by 650 - Related articles
We present a clean algorithm for determining whether a **ray intersects a triangle**. ... ble

[PDF] Optimizing Ray-Triangle Intersection via Automated Search
www.cs.utah.edu/~aek/research/**triangle**.pdf University of Utah
by A Kensler - Cited by 33 - Related articles
method is used to further optimize the code produced via the fitness function. ... For these 3D **methods** we optimize **ray-triangle intersection** in two different **ways**.

[PDF] Comparative Study of Ray-Triangle Intersection Algorithms
www.graphicon.ru/html/proceedings/2012/.../gc2012Shumskiy.pdf
by V Shumskiy - Cited by 1 - Related articles

Basic ray casting algorithm

Sample = a ray in 3D

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

Occlusion: closest intersection along ray

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

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

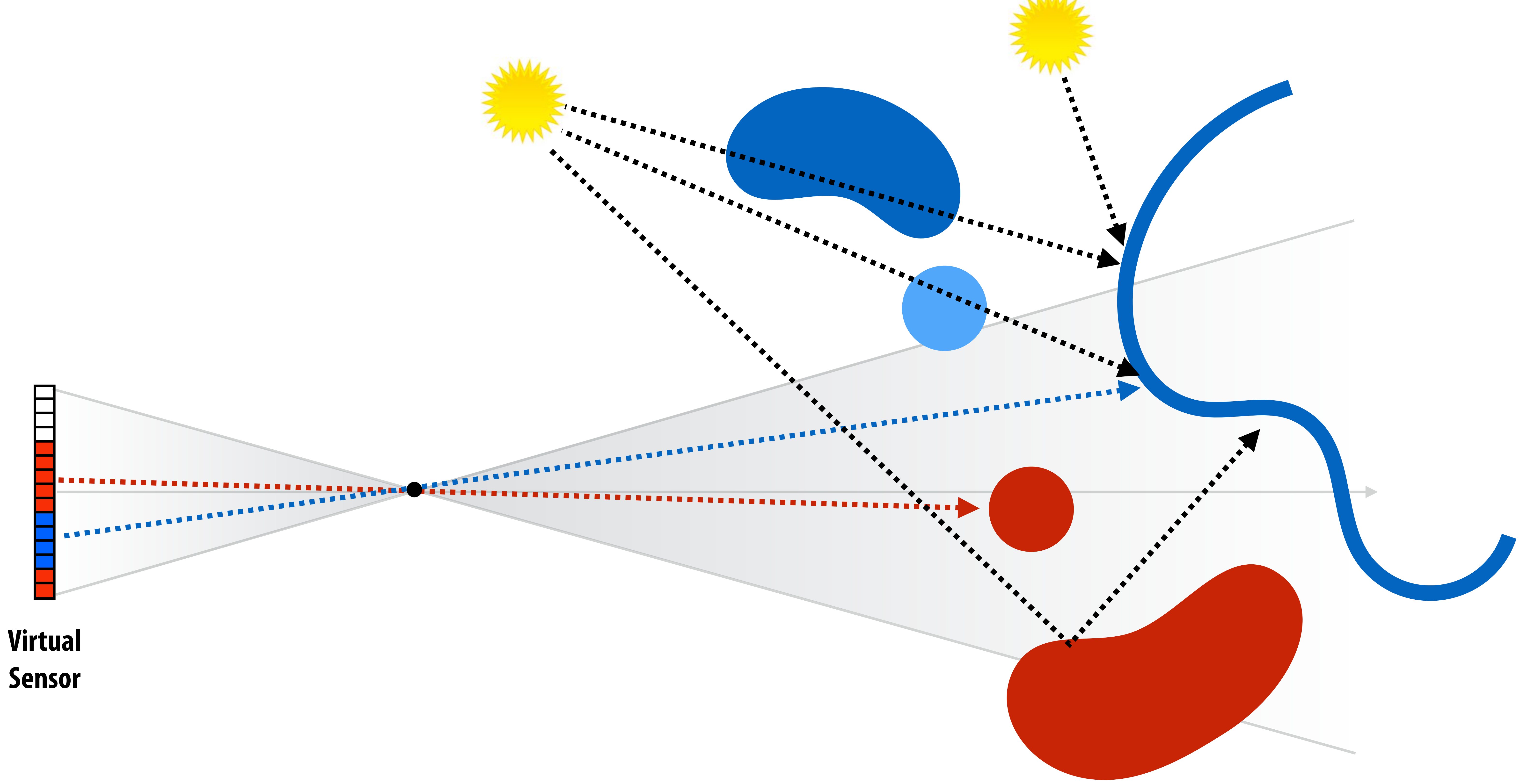
We'll replace this brute force O(N) loop: "for each triangle, see if it's the closest" soon with an acceleration structure in a few slides…

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?



Takeaway:
ray-triangle intersection is an arithmetically rich operation

Ray-scene intersection

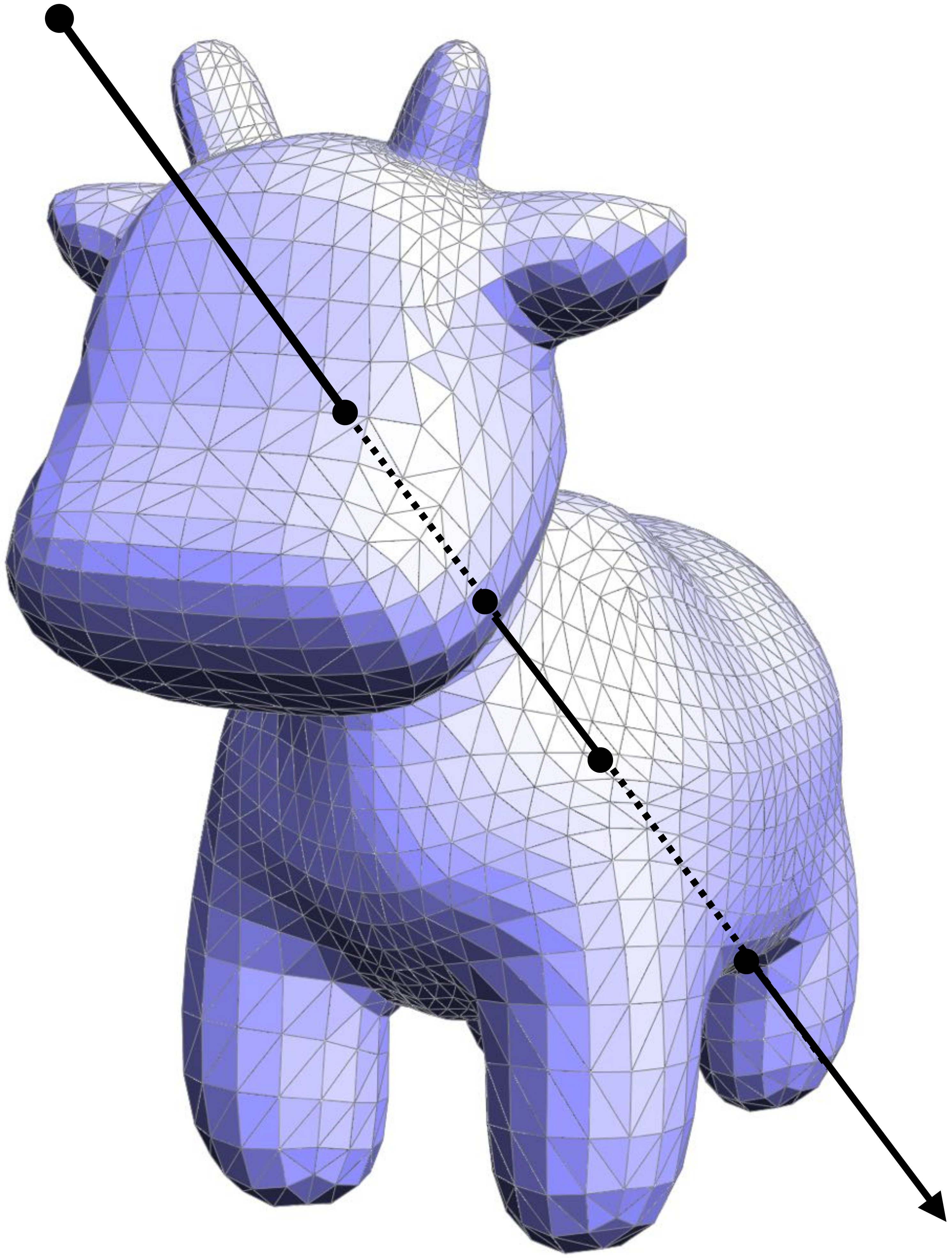
Given a scene defined by a set of *N* primitives and a ray *r*, find the closest point of intersection of *r* with the scene

"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
```

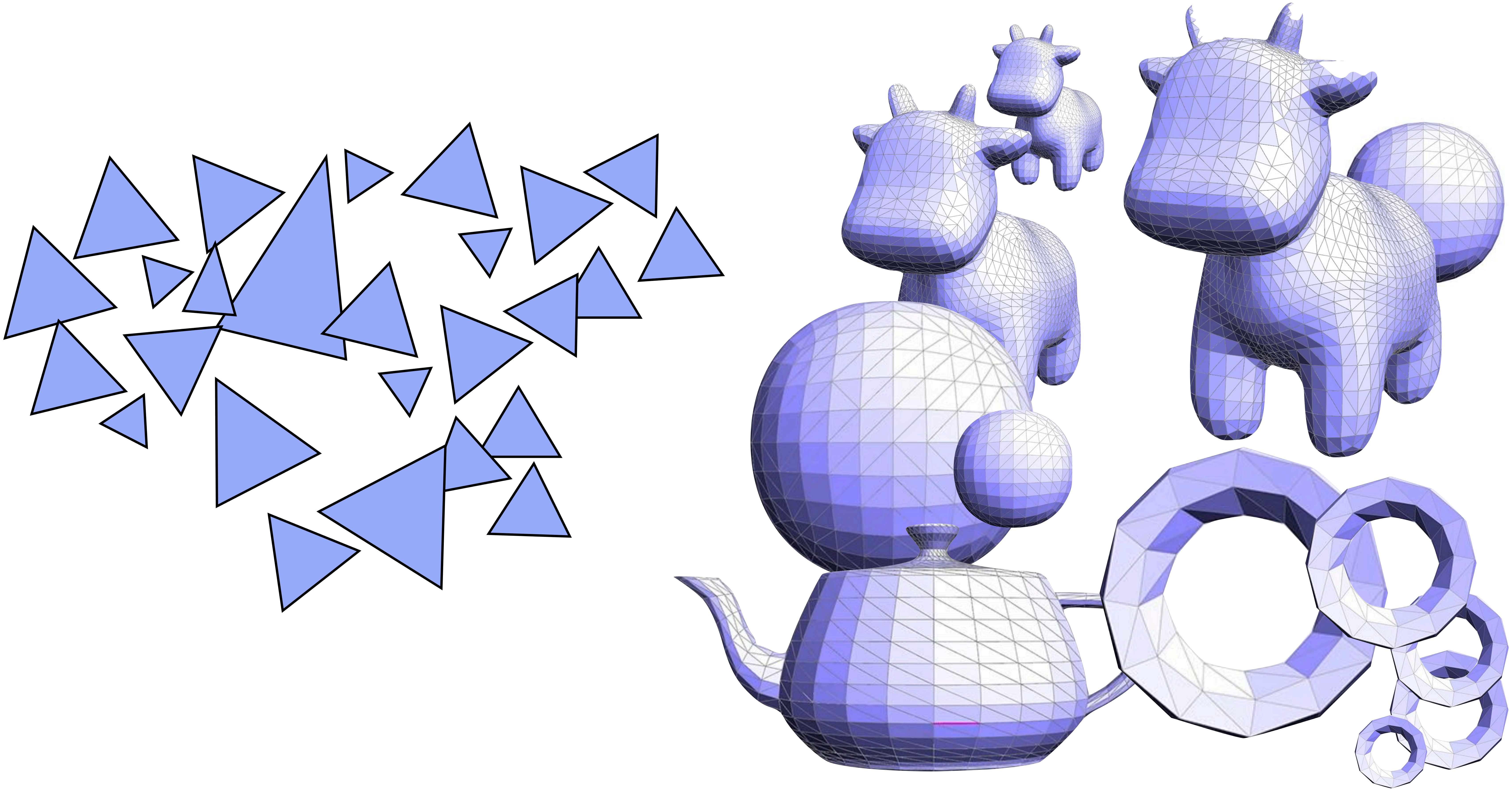
Complexity? $O(N)$

Can we do better?

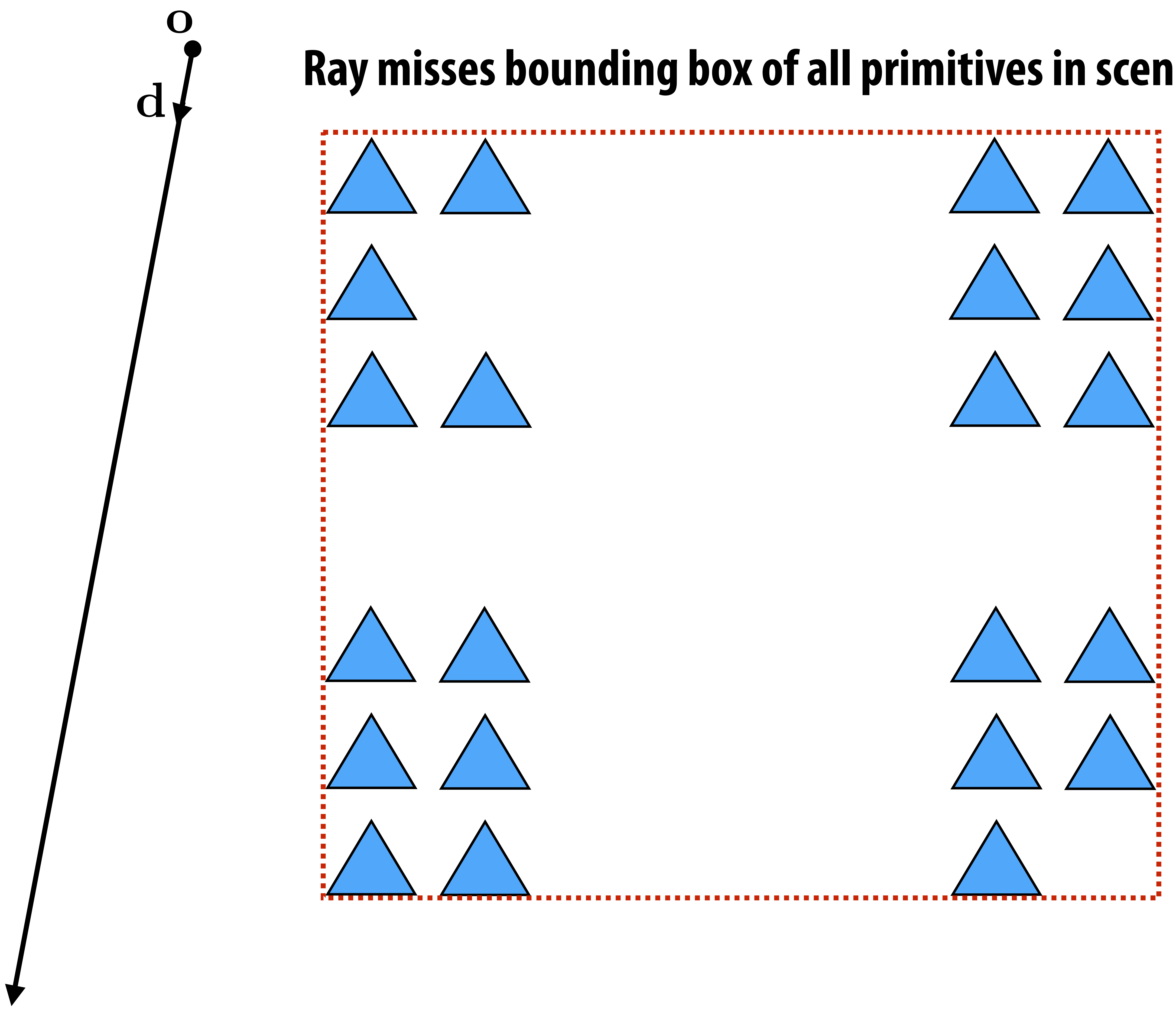


(Assume p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r*)

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>

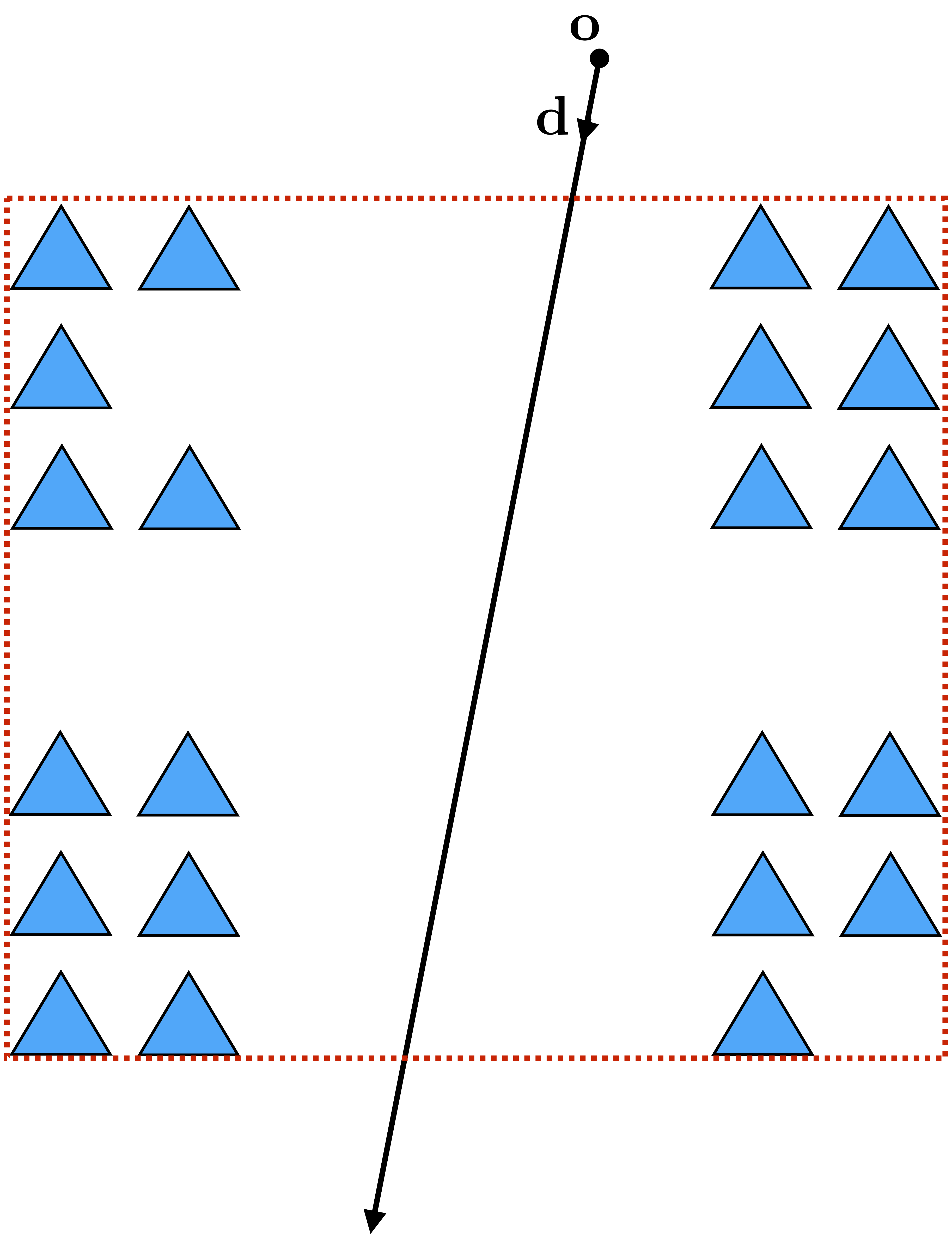
preprocessing: O(n)

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)

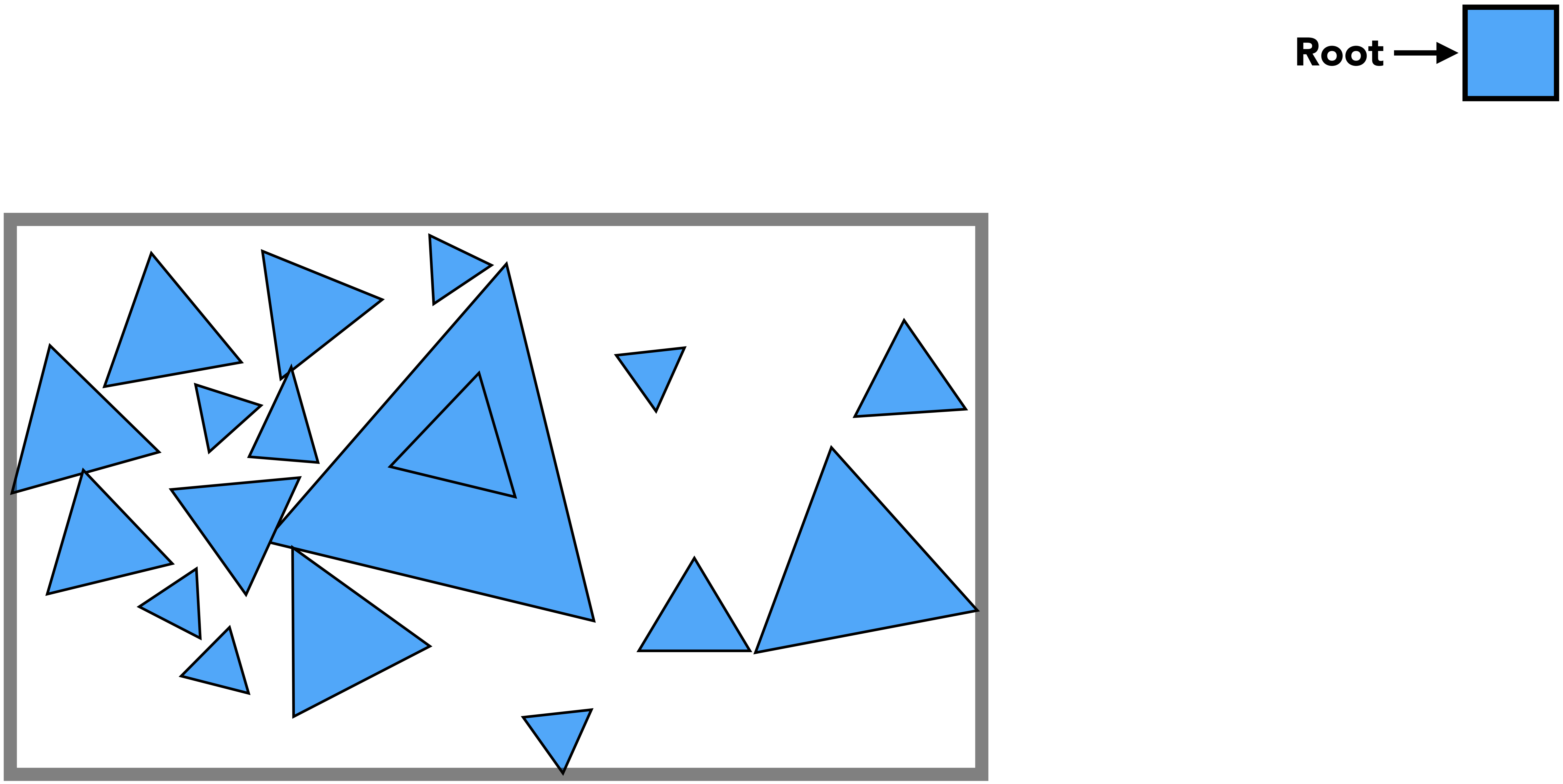
Still no better than naïve algorithm (test all triangles)!

*amortized over *many* ray-scene intersection tests

Q: How can we do better?

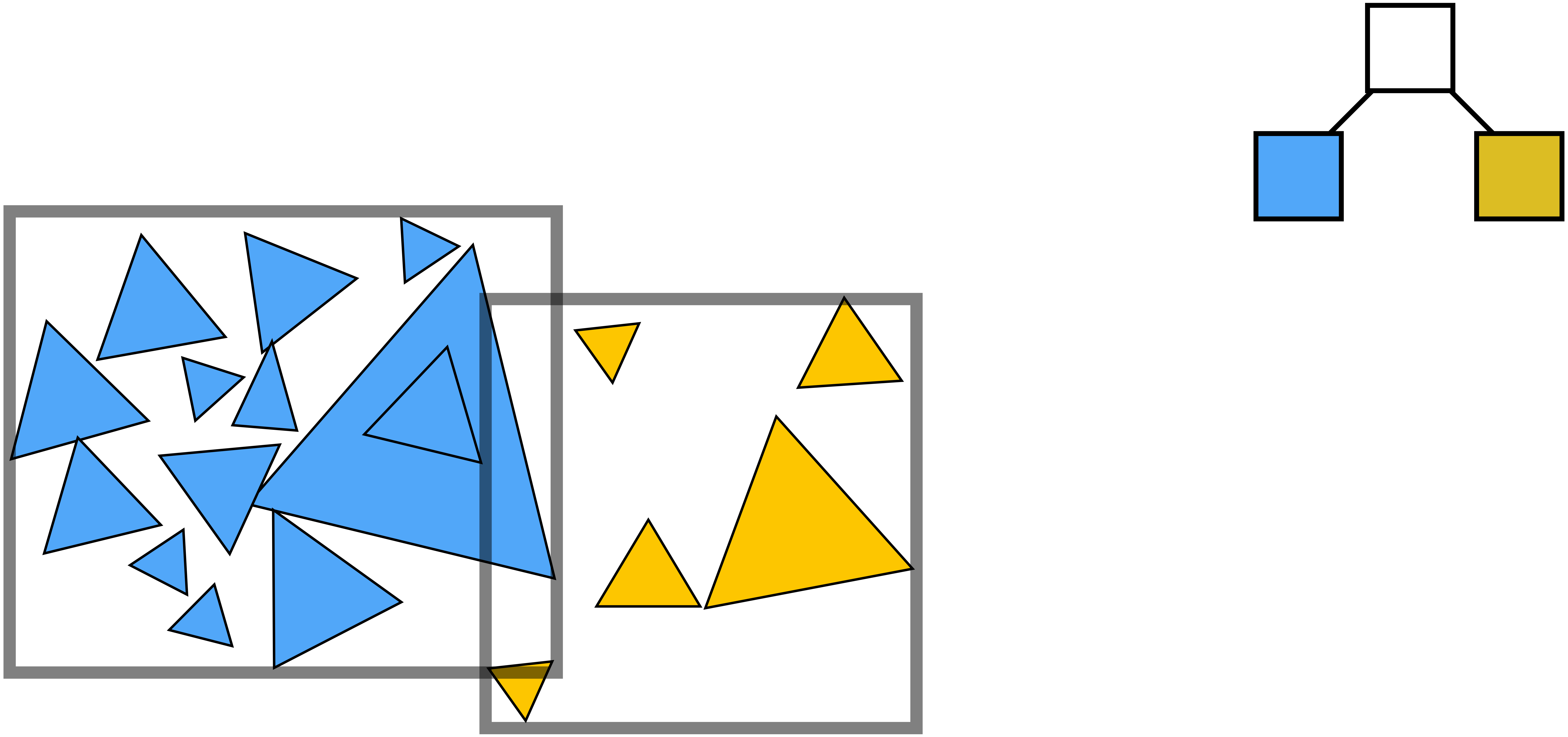
A: Apply this strategy hierarchically

Bounding volume hierarchy (BVH)

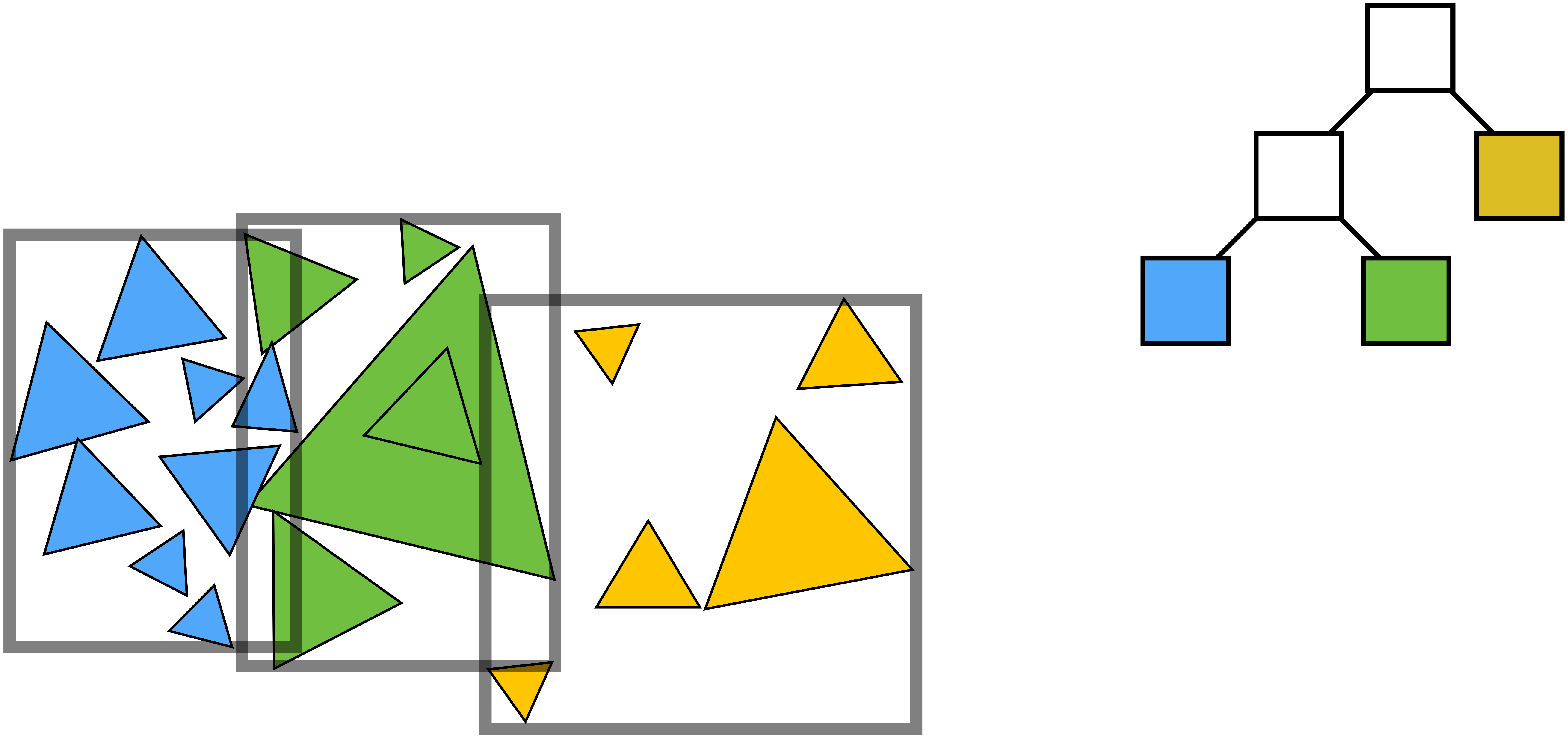


Bounding volume hierarchy (BVH)

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

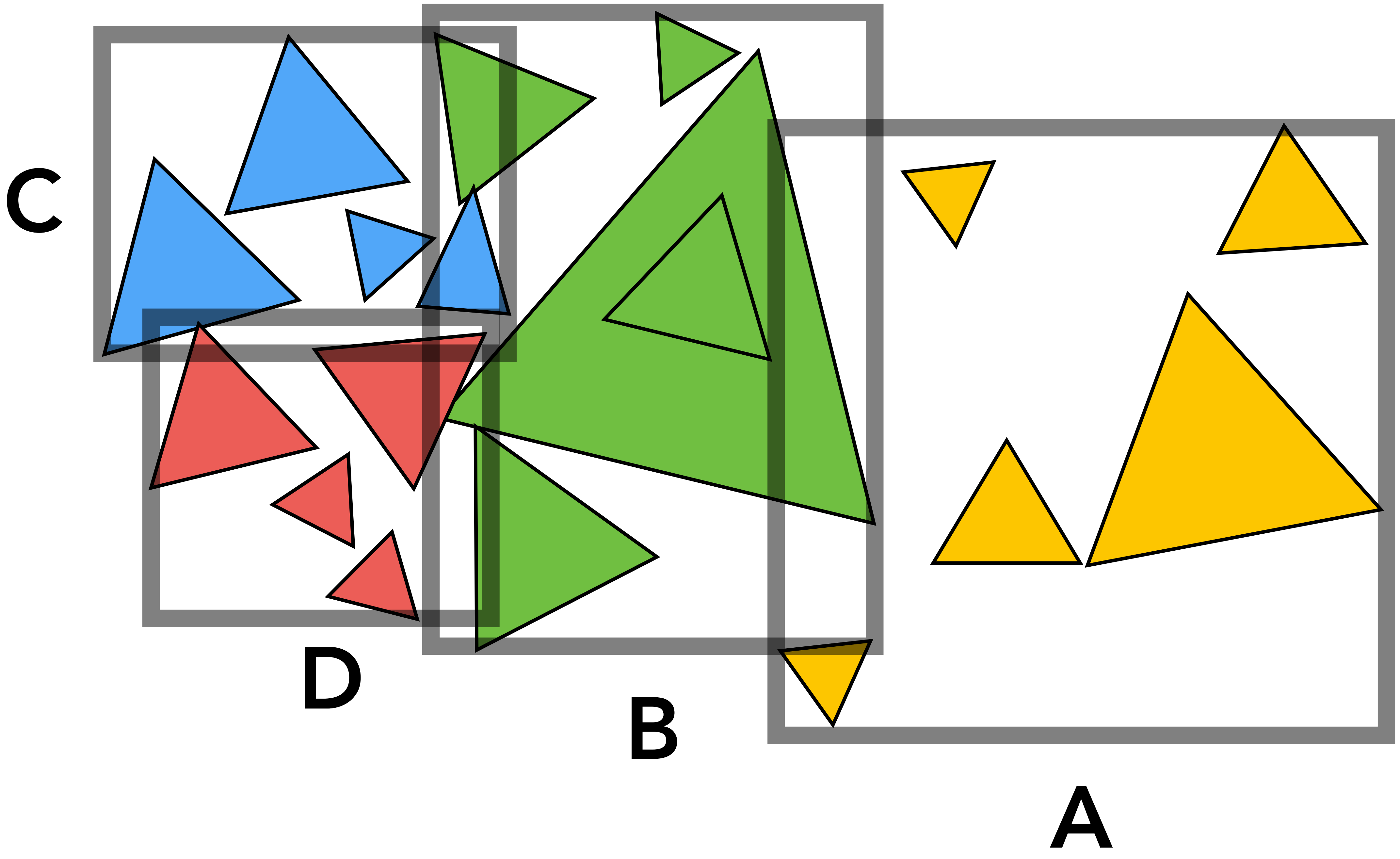


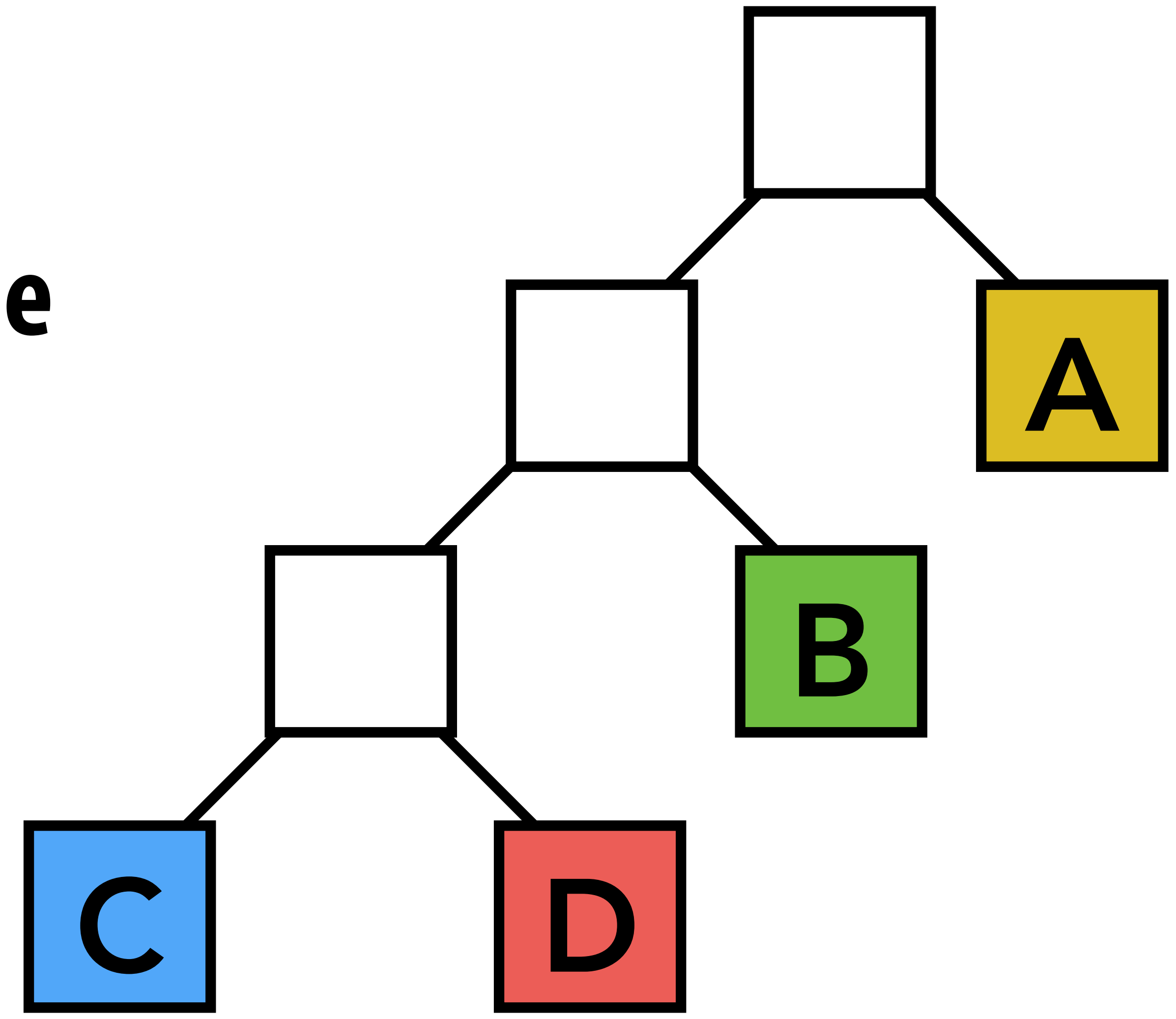
Bounding volume hierarchy (BVH)



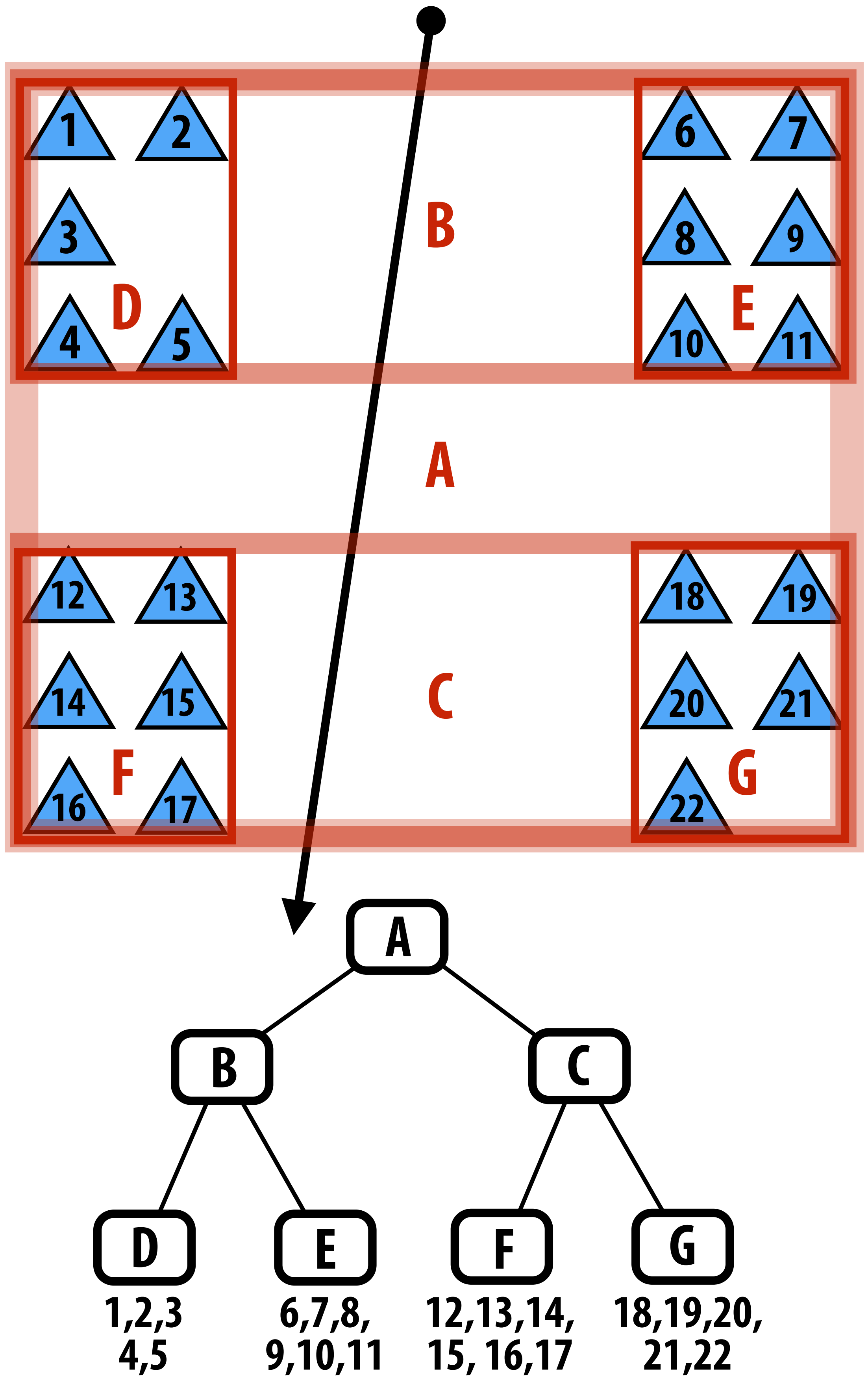
Bounding volume hierarchy (BVH)

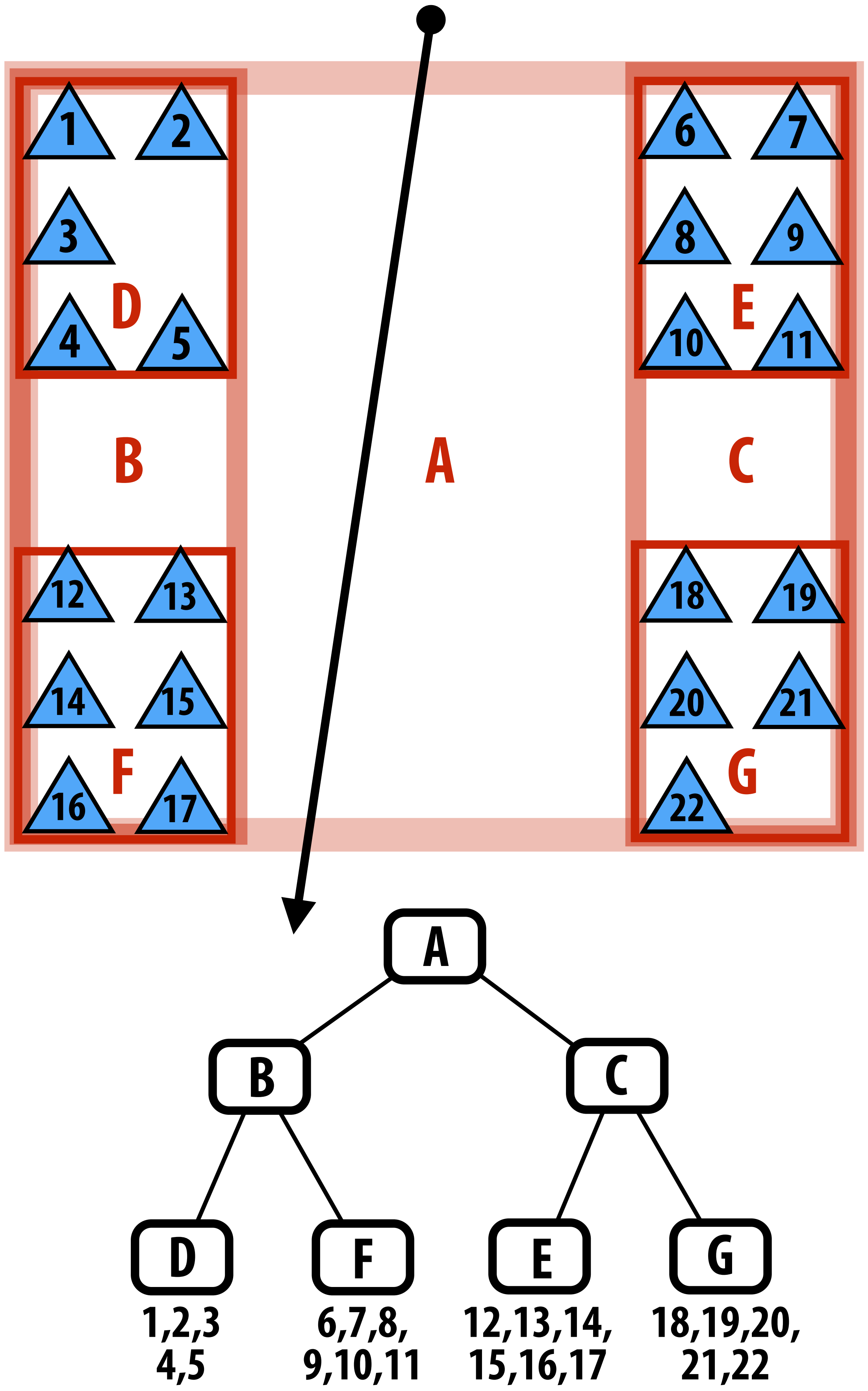
- **Leaf nodes:**
  - Contain *small* list of primitives
- **Interior nodes:**
  - Proxy for a *large* subset of primitives
  - Stores bounding box for all primitives in subtree





Bounding volume hierarchy (BVH)



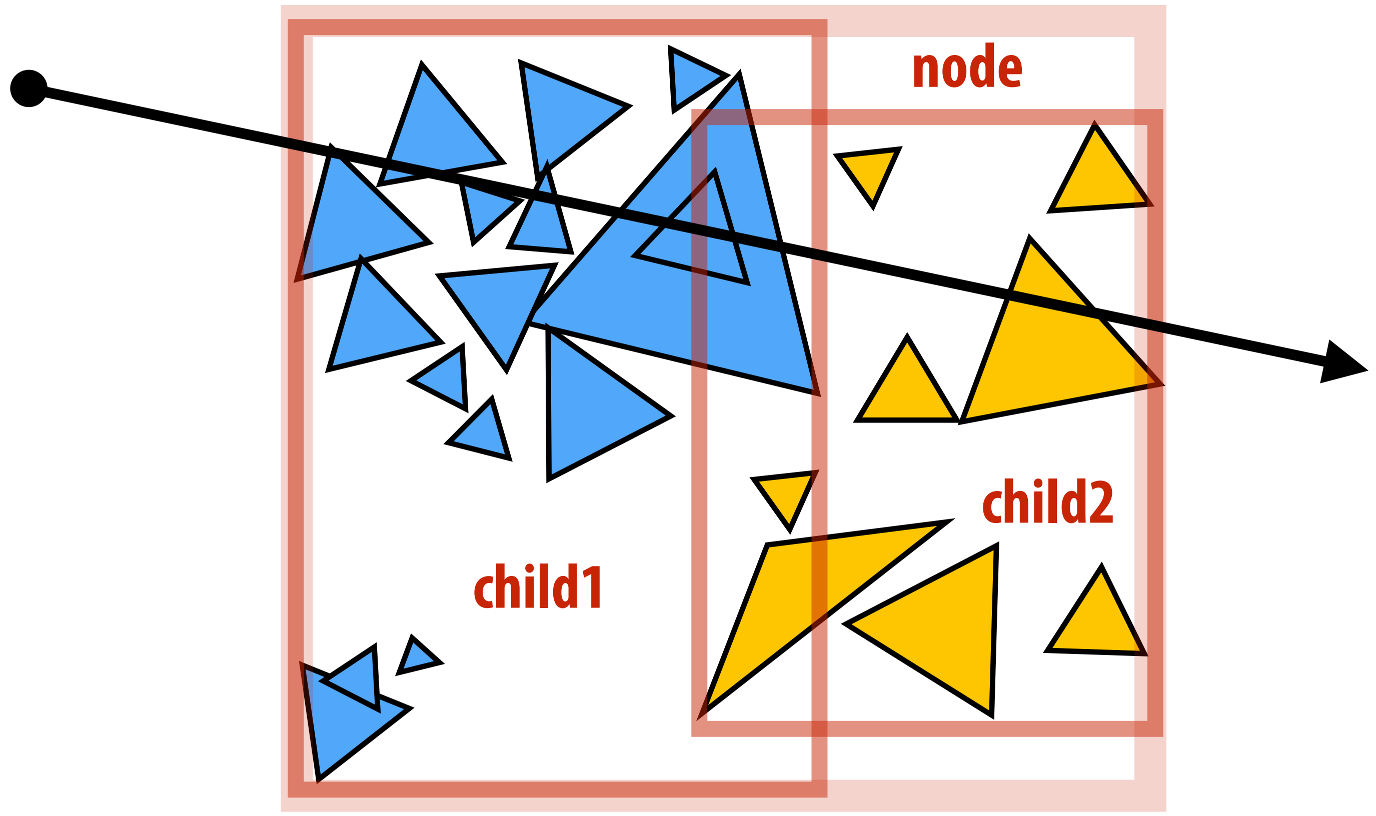


Left: two different BVH organizations of the same scene containing 22 primitives.

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);
   }}
```

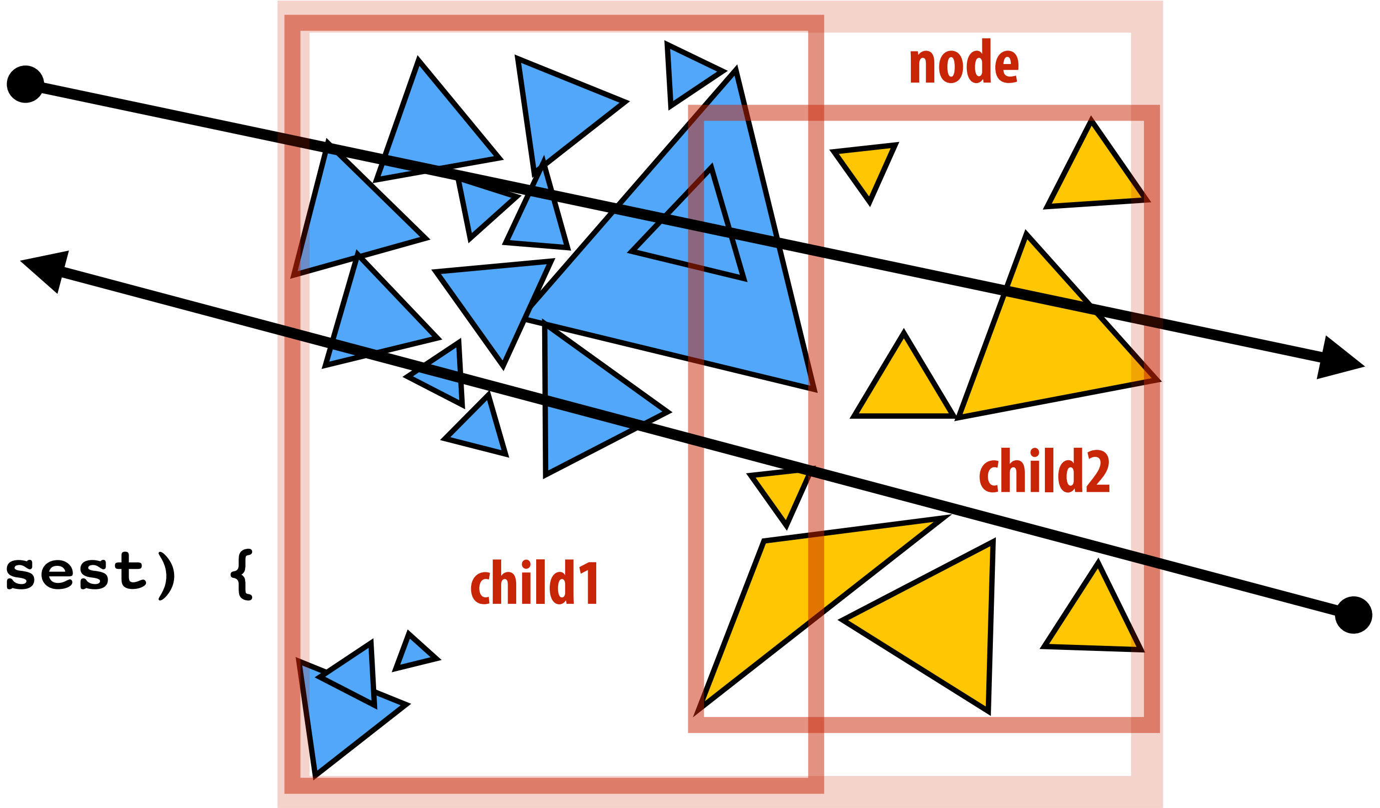


Can this occur if ray hits the box?

(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 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); // 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

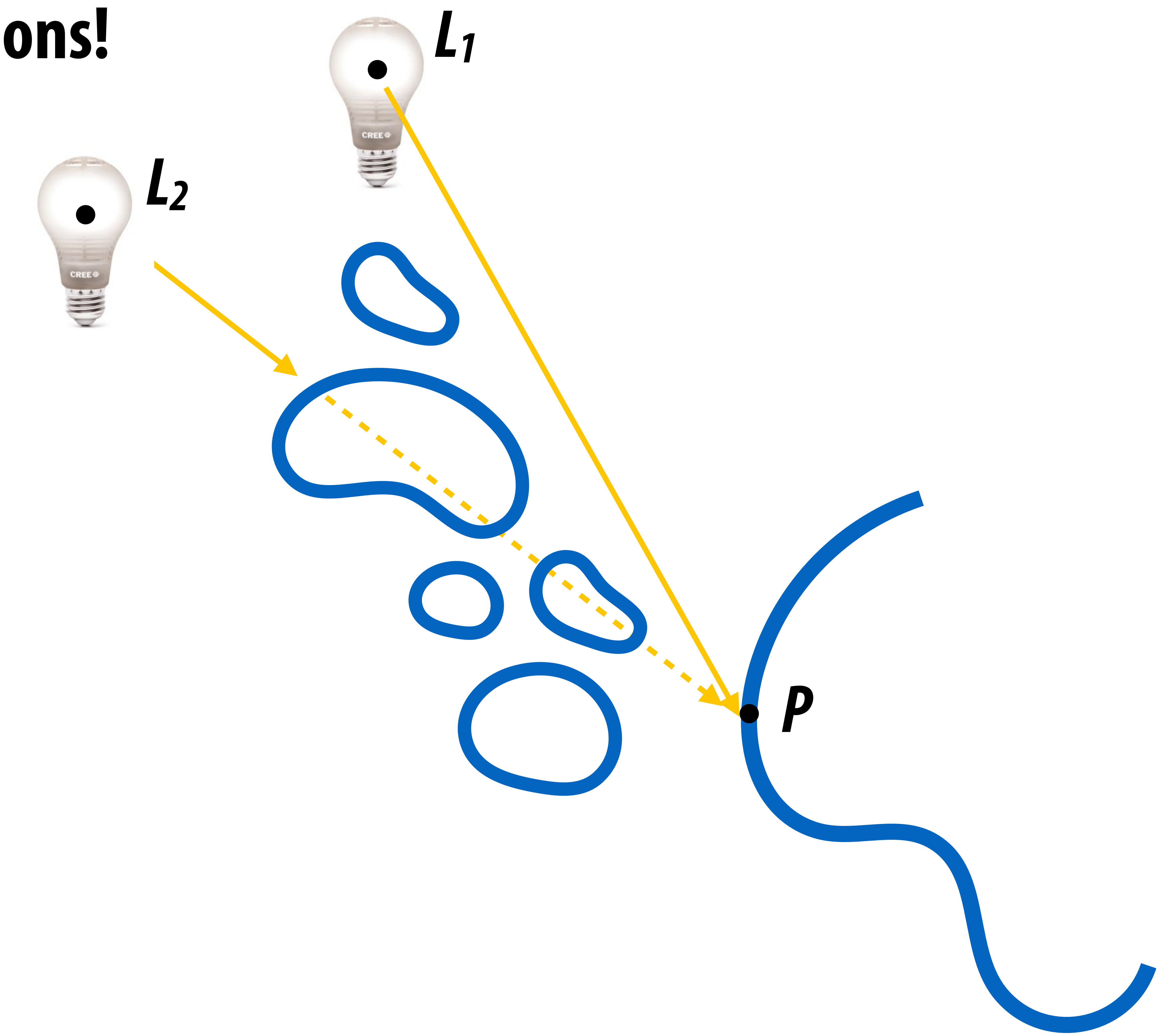
Sometimes it is useful to know if the ray hits ANY primitive in the scene at all (don't care about distance to first hit)

```
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!



Takeaway:
Ray-BVH traversal generates unpredictable (data-dependent) access to an irregular data structure

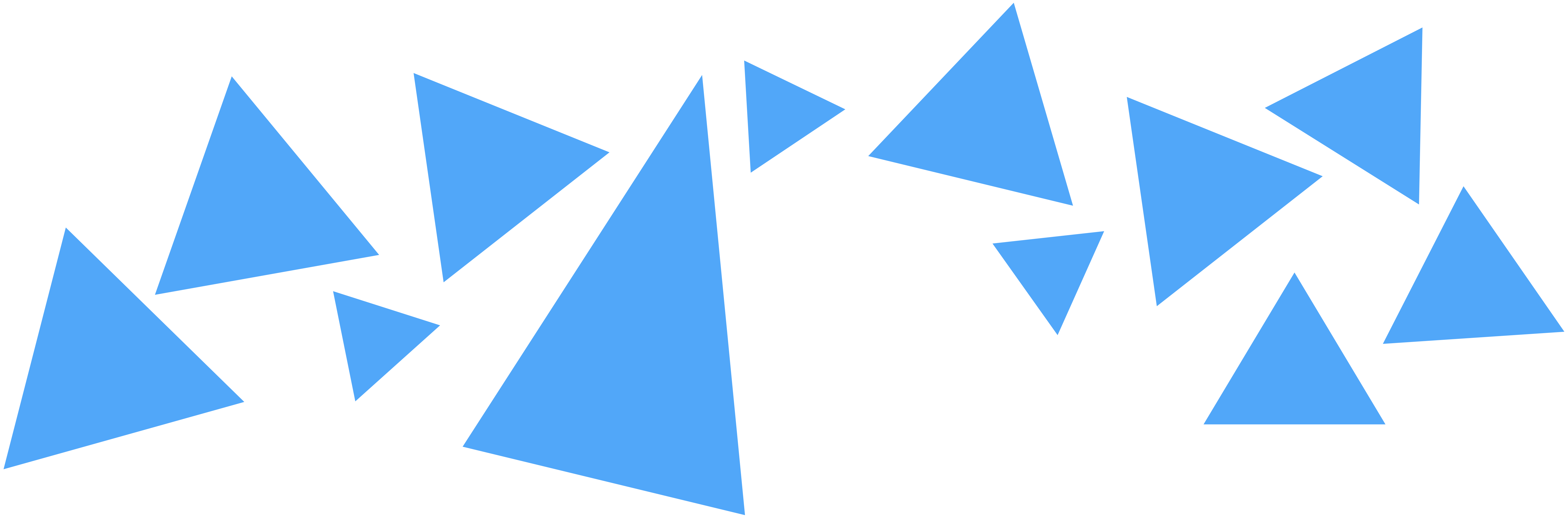
(Later we'll talk about why this can create situations where traversal is bandwidth-limited)

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

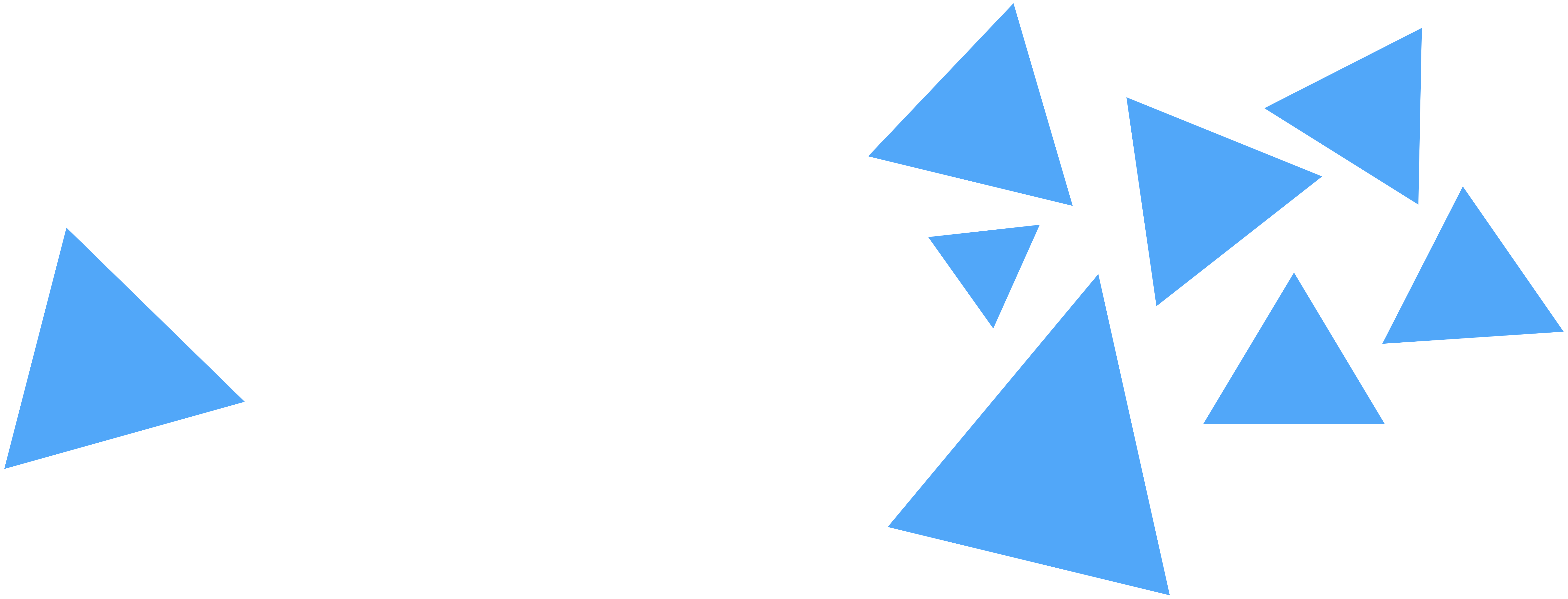
($\sim 2^N$ ways to partition N primitives into two groups)

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

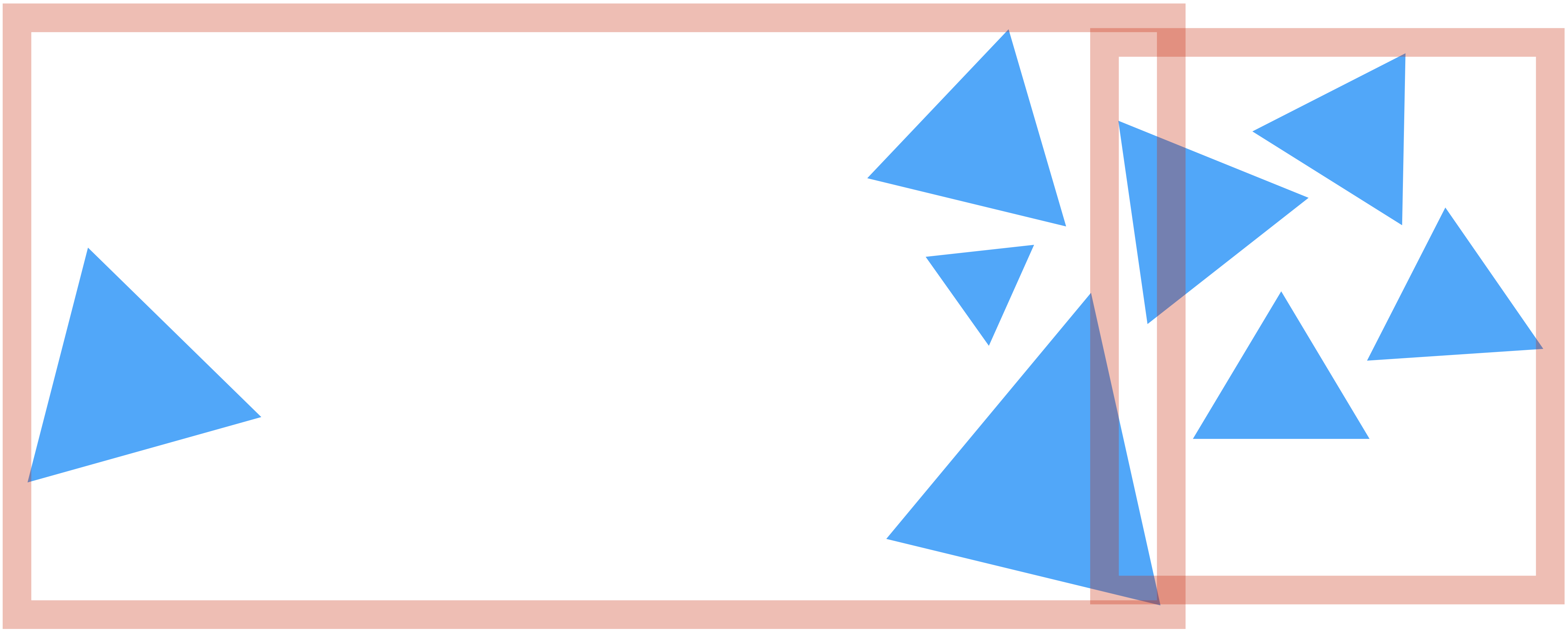
How would you partition these triangles into two groups?



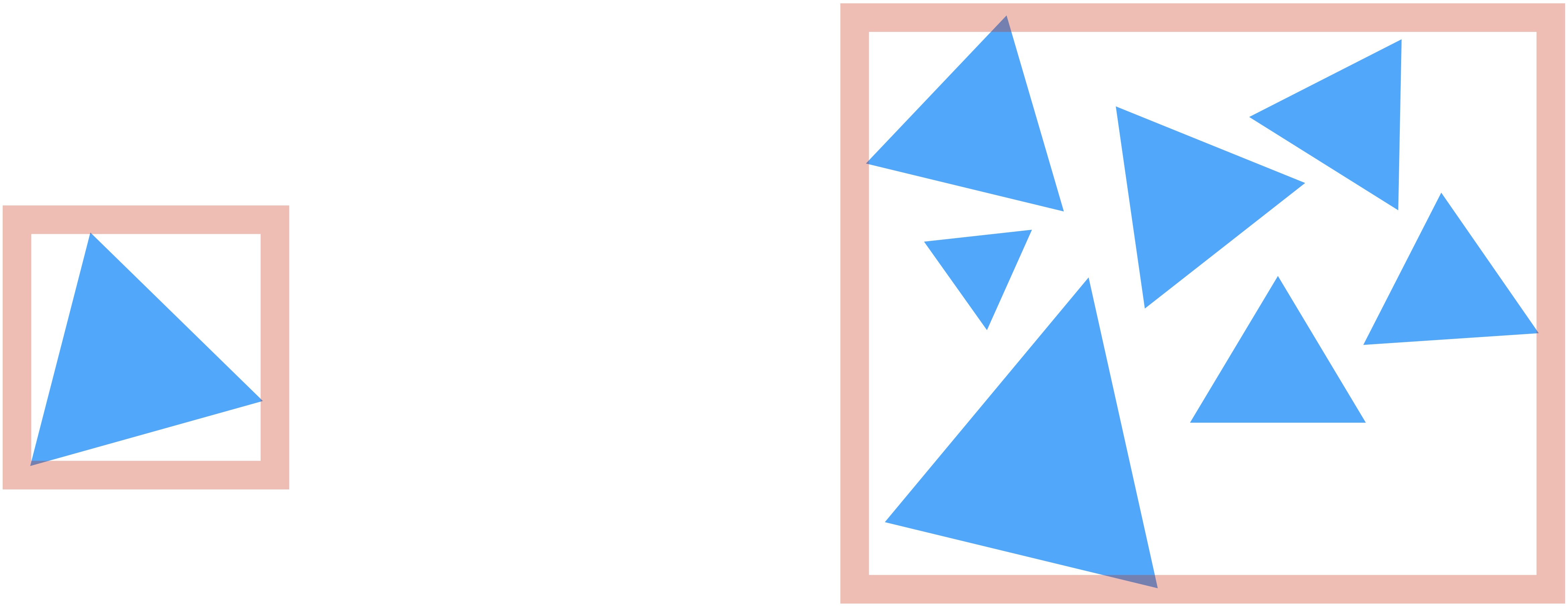
What about these?



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.

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

(Common to assume all primitives have the same cost)

Cost of making a partition

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

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

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:

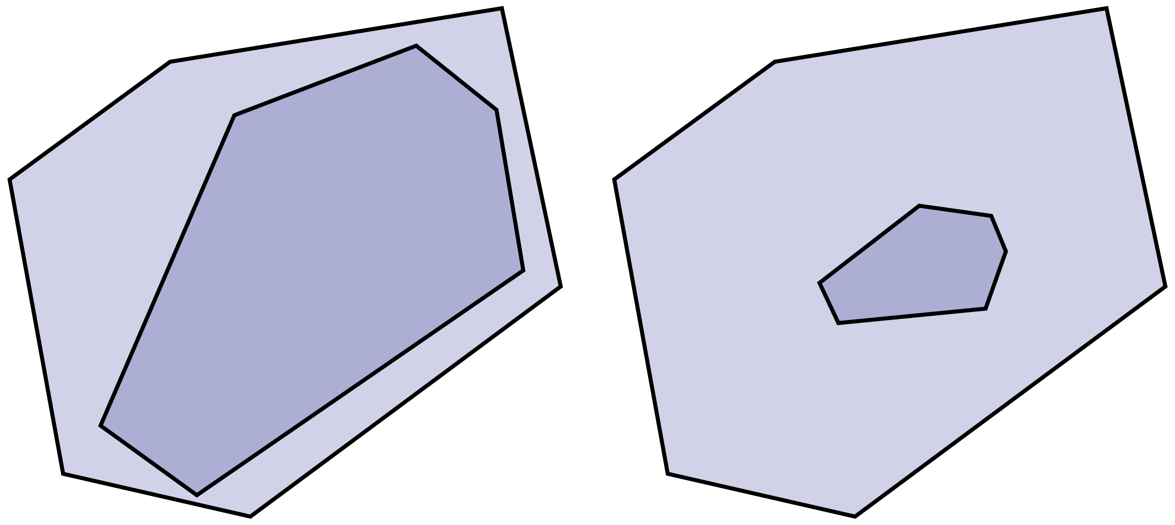
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

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):

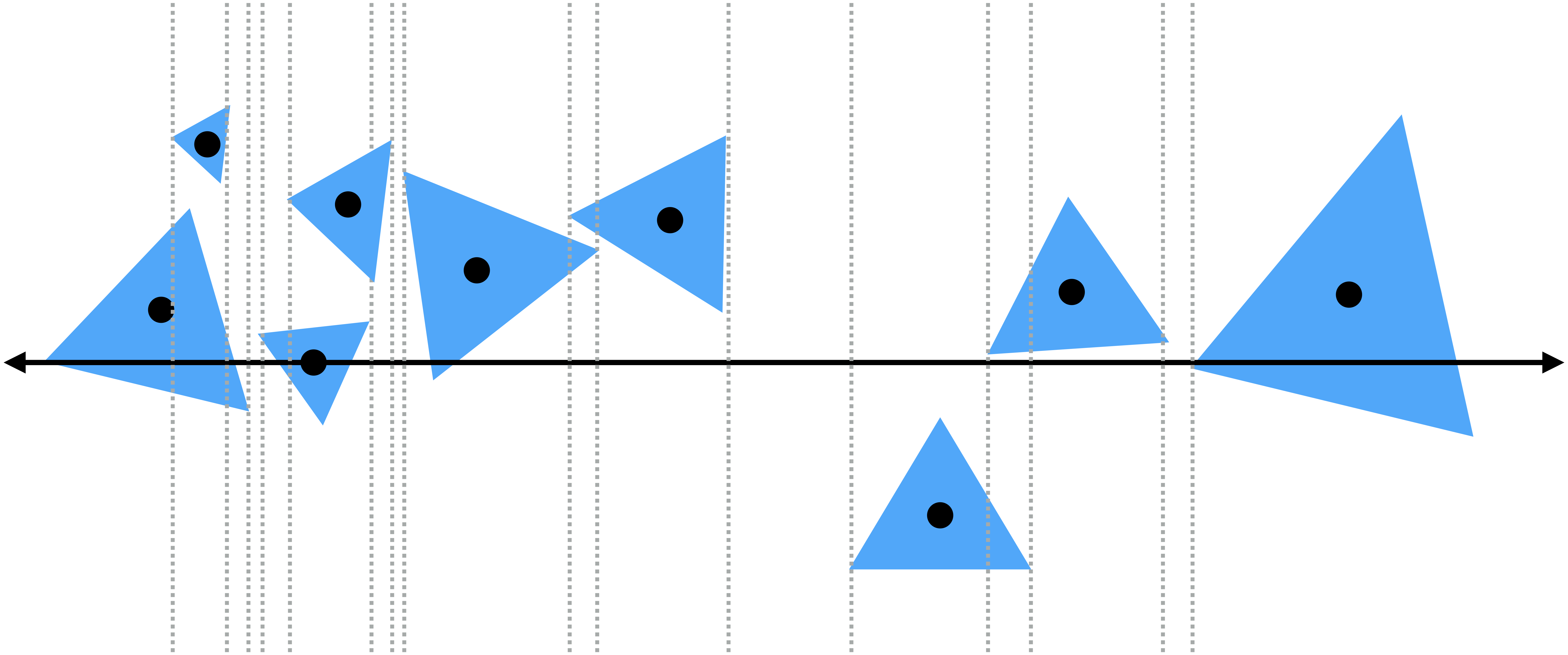
$$C = C_{\mathrm{trav}} + \frac{S_A}{S_N} N_A C_{\mathrm{isect}} + \frac{S_B}{S_N} N_B C_{\mathrm{isect}}$$

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

- Rays are randomly distributed
- Rays are not occluded

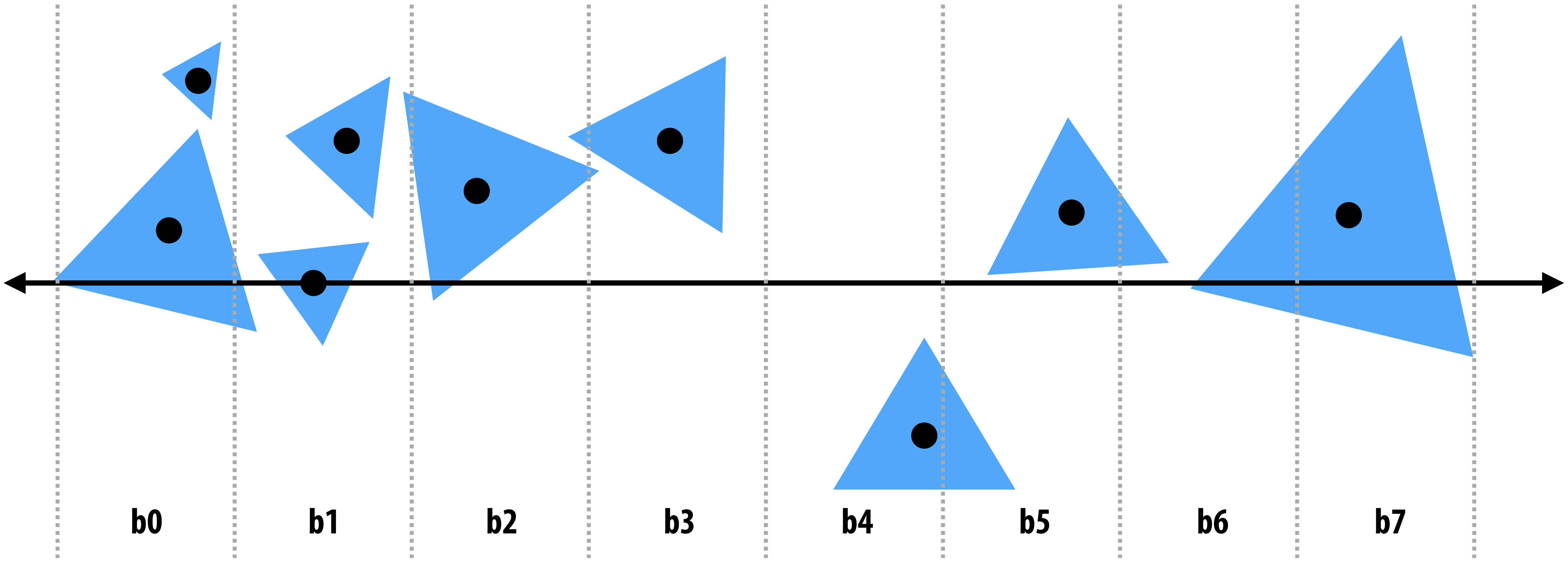
Implementing partitions

- 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, 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)
```

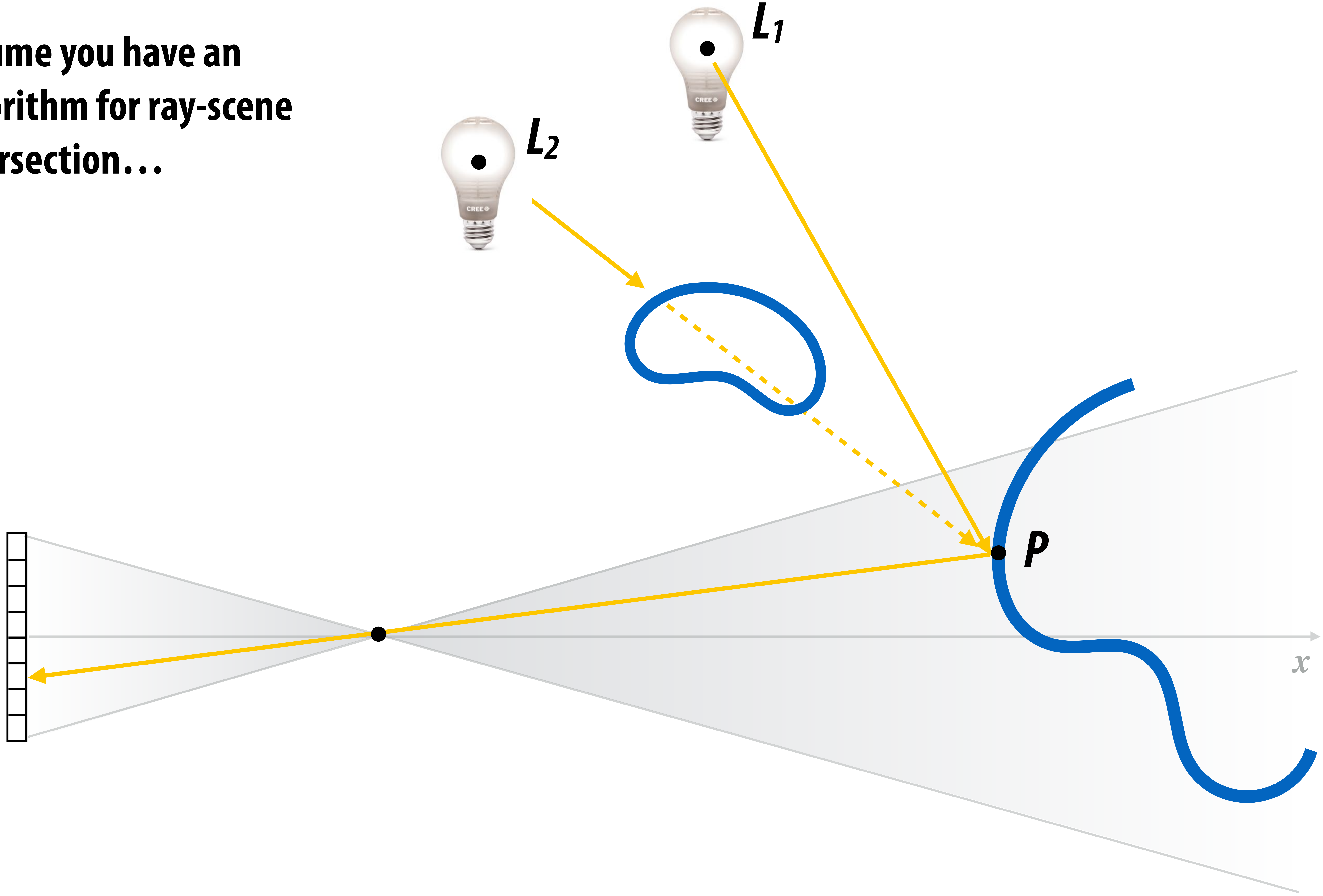
Why do we trace rays?

Shadows

Image credit: Grand Theft Auto V

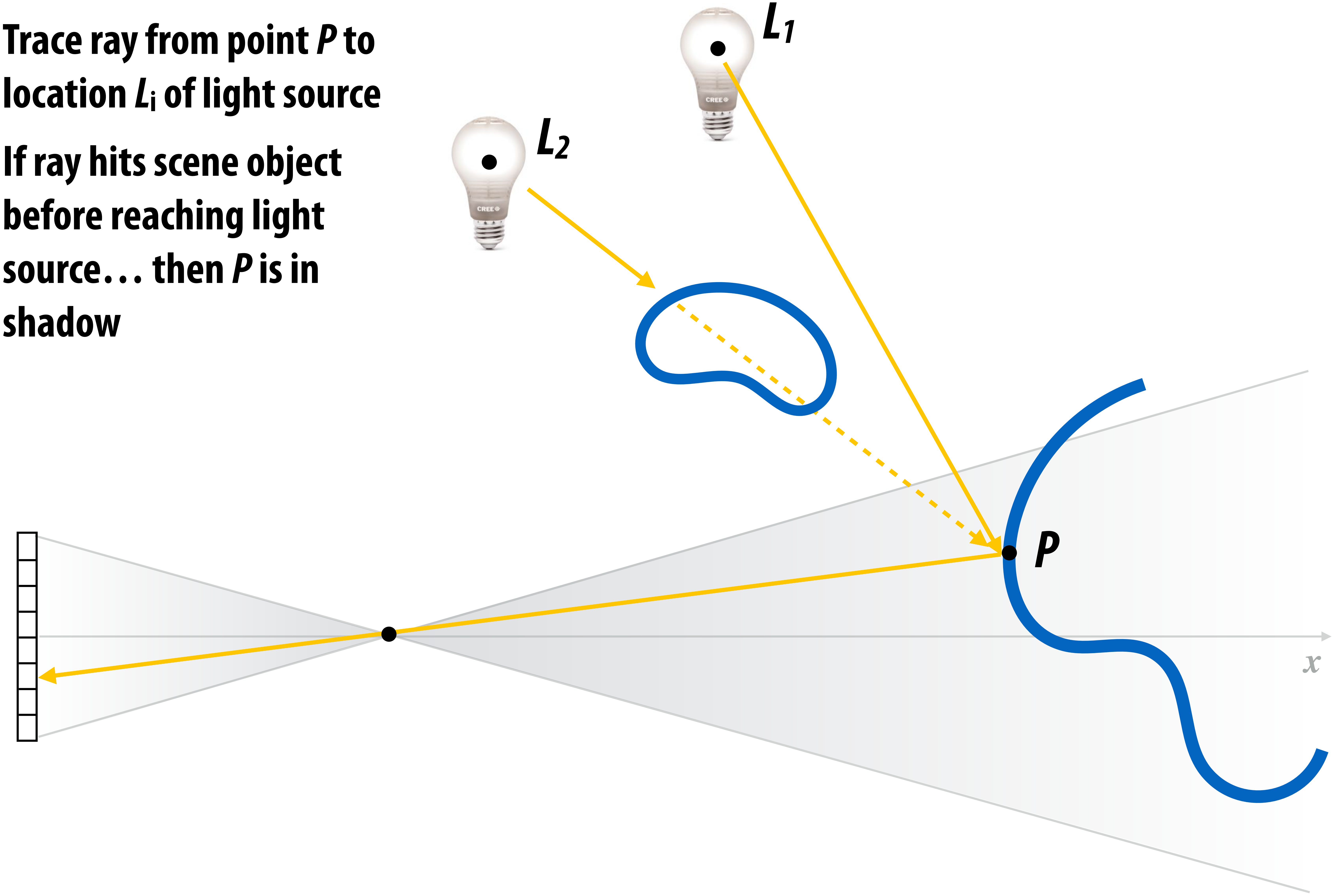
How to compute if a surface point is in shadow?

Assume you have an algorithm for ray-scene intersection…

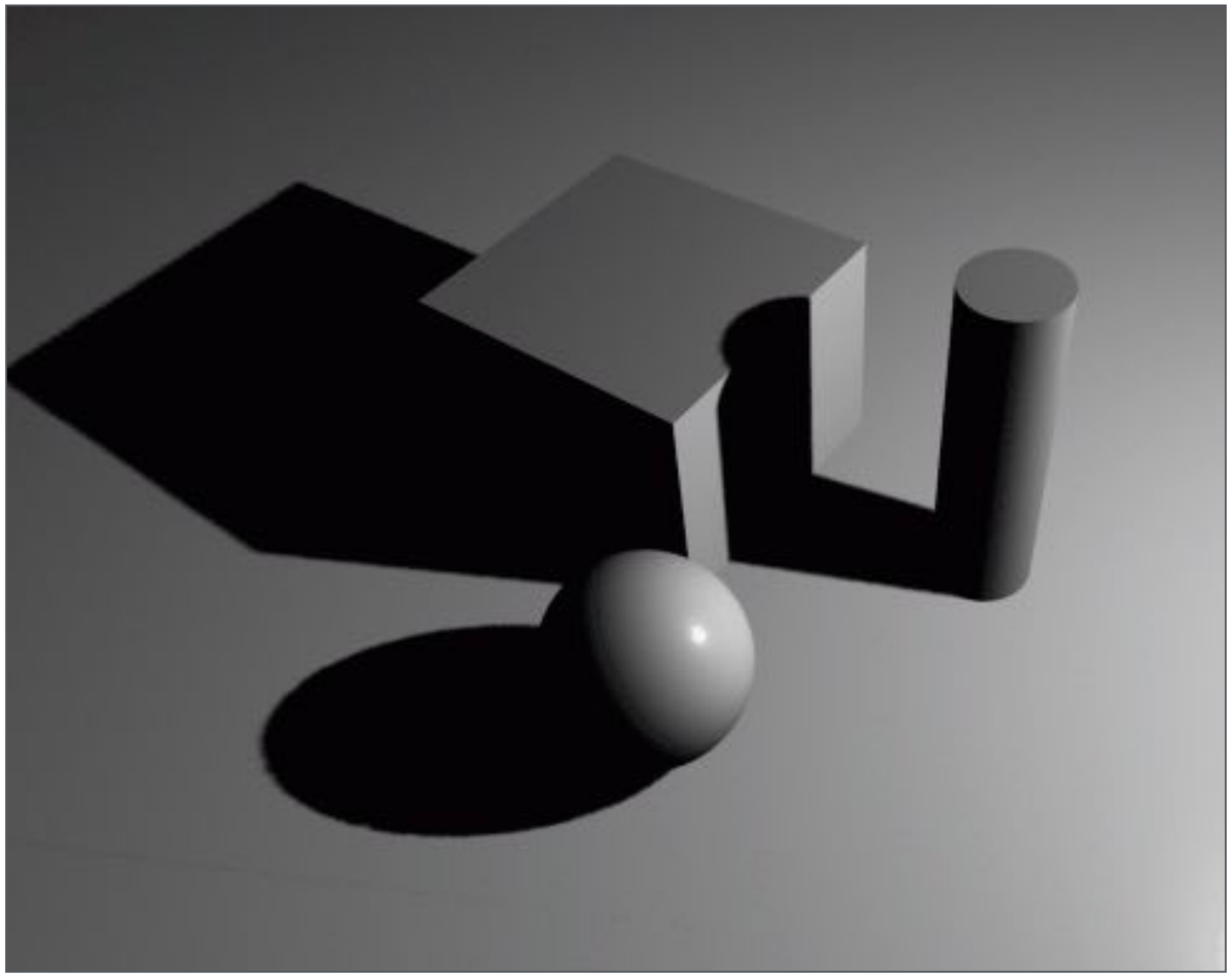


A simple shadow computation algorithm

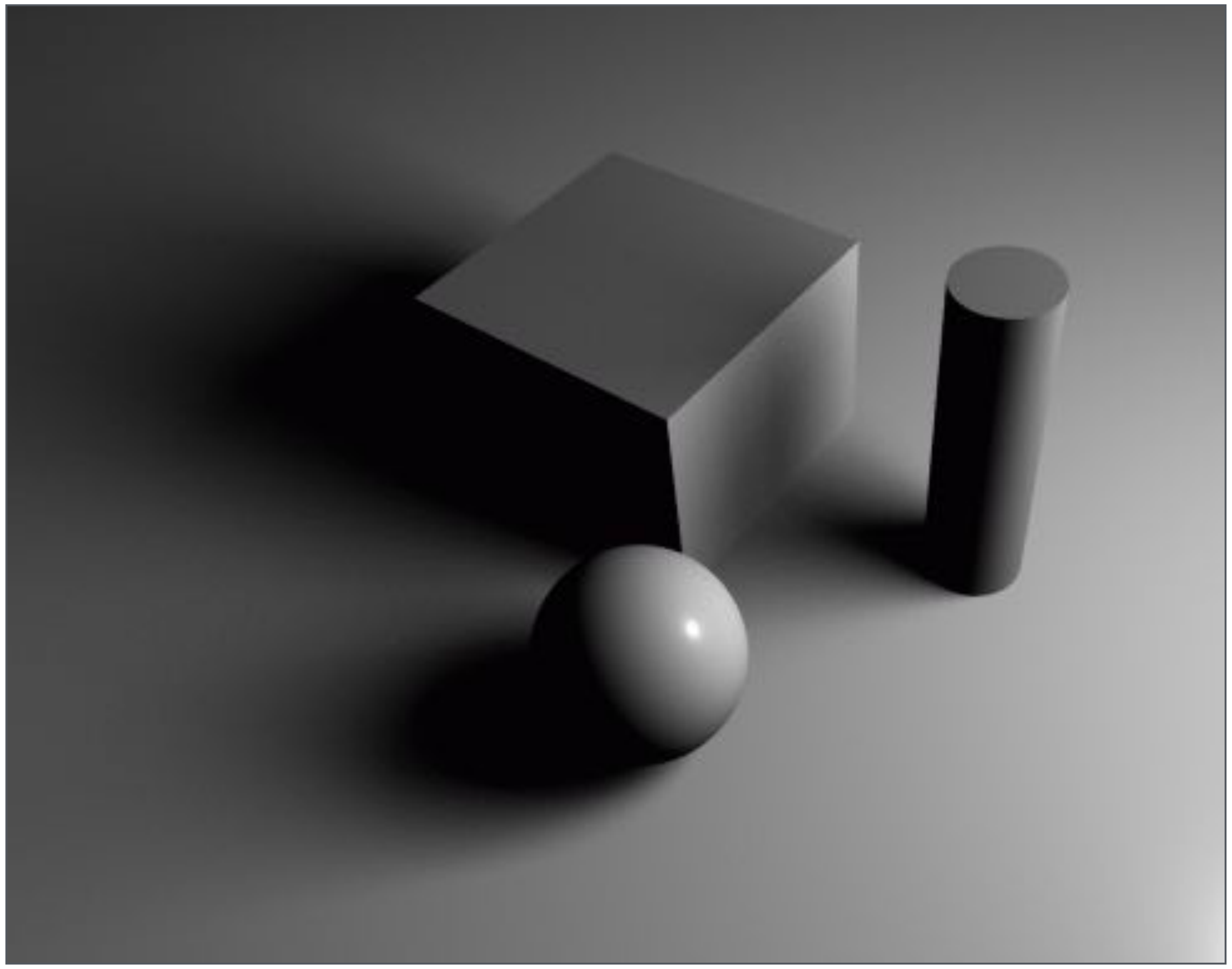
- Trace ray from point *P* to location L_i of light source
- If ray hits scene object before reaching light source… then *P* is in shadow



Soft shadows



Hard shadows
(created by point light source)



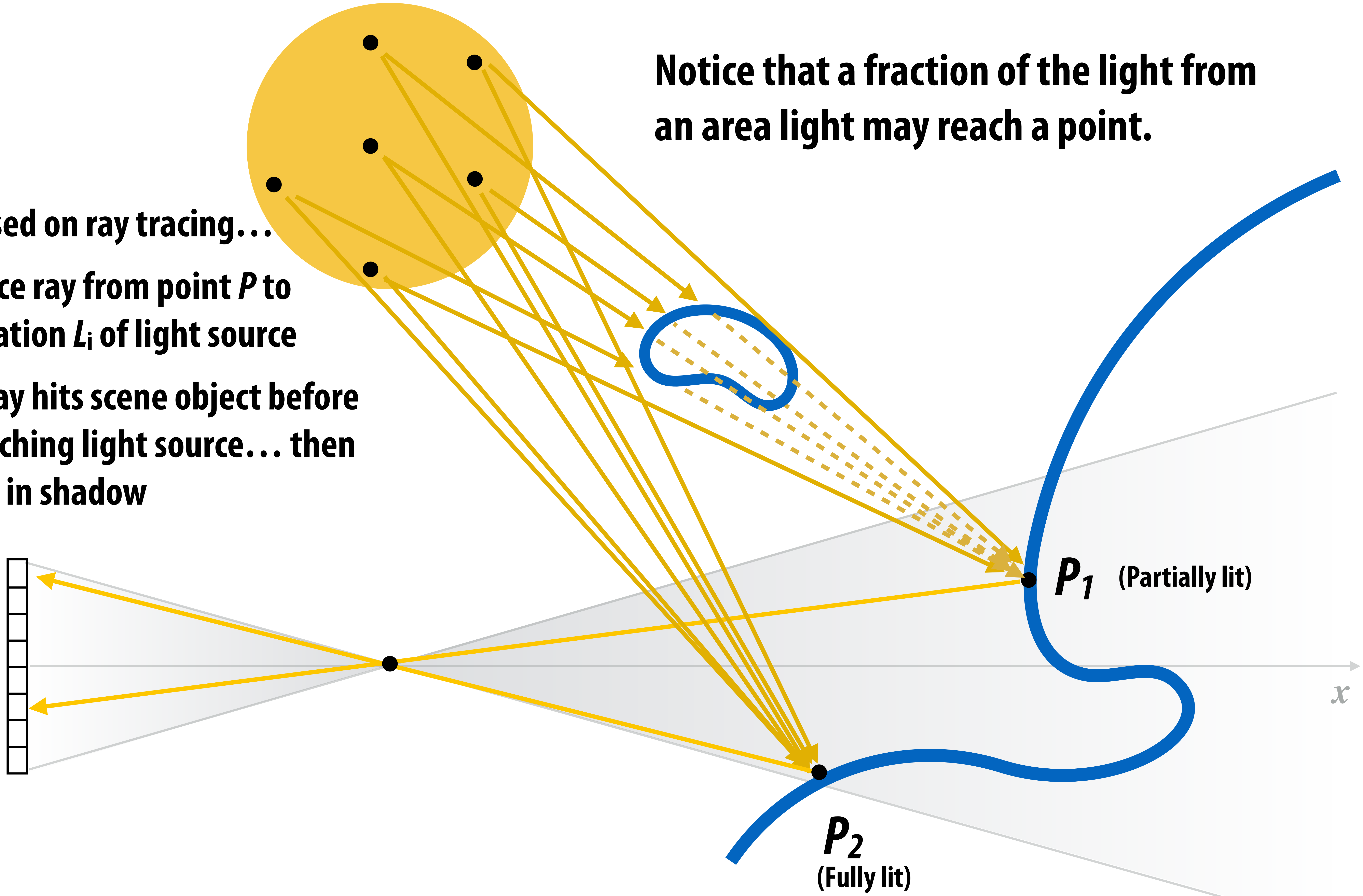
Soft shadows
(created by ???)

Image credit: Pixar

Shadow cast by an area light

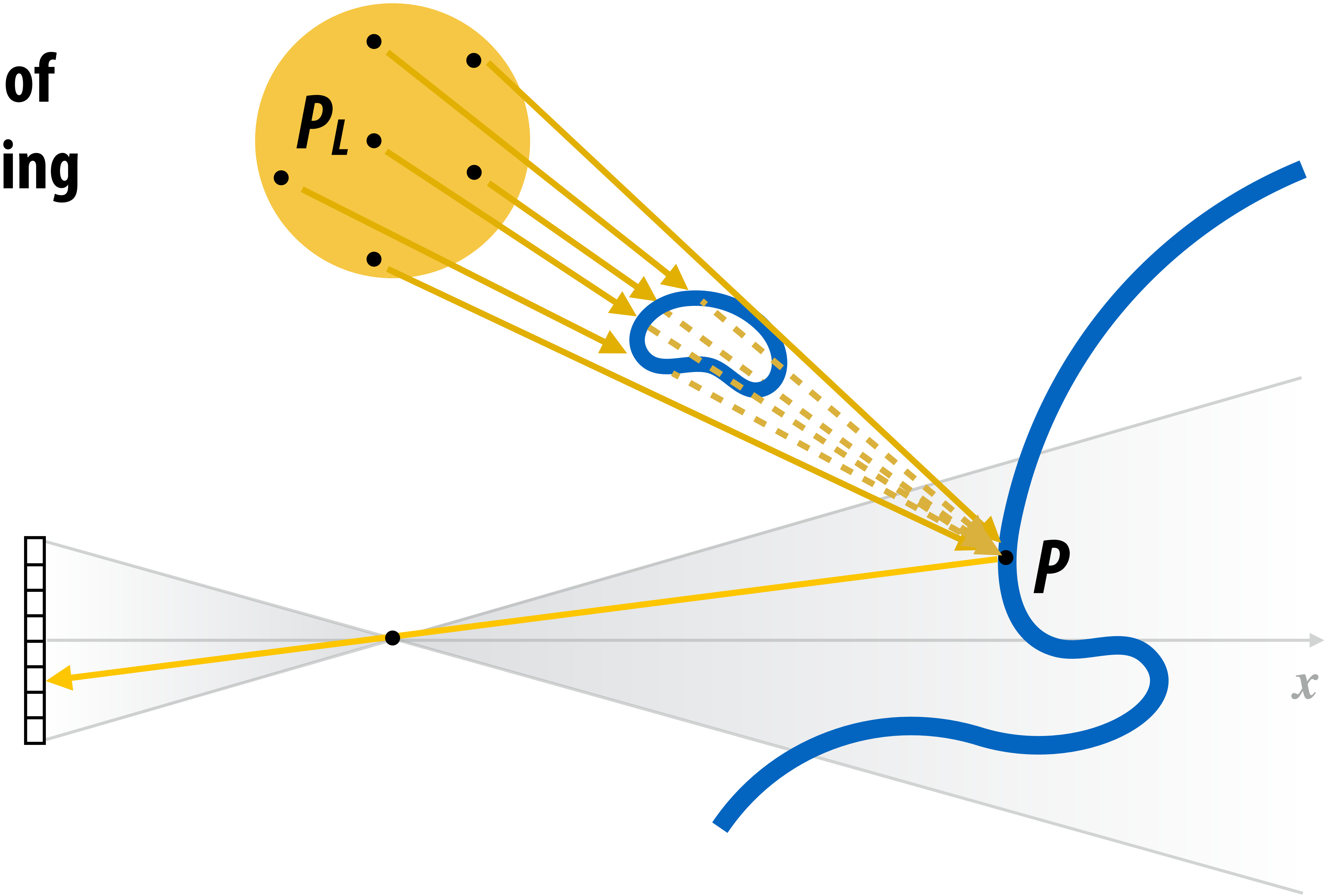
Notice that a fraction of the light from an area light may reach a point.

- Based on ray tracing…
- Trace ray from point *P* to location L_i of light source
- If ray hits scene object before reaching light source… then *P* is in shadow



Sampling based algorithm

Goal: estimate the amount of light from area source arriving at a surface point P

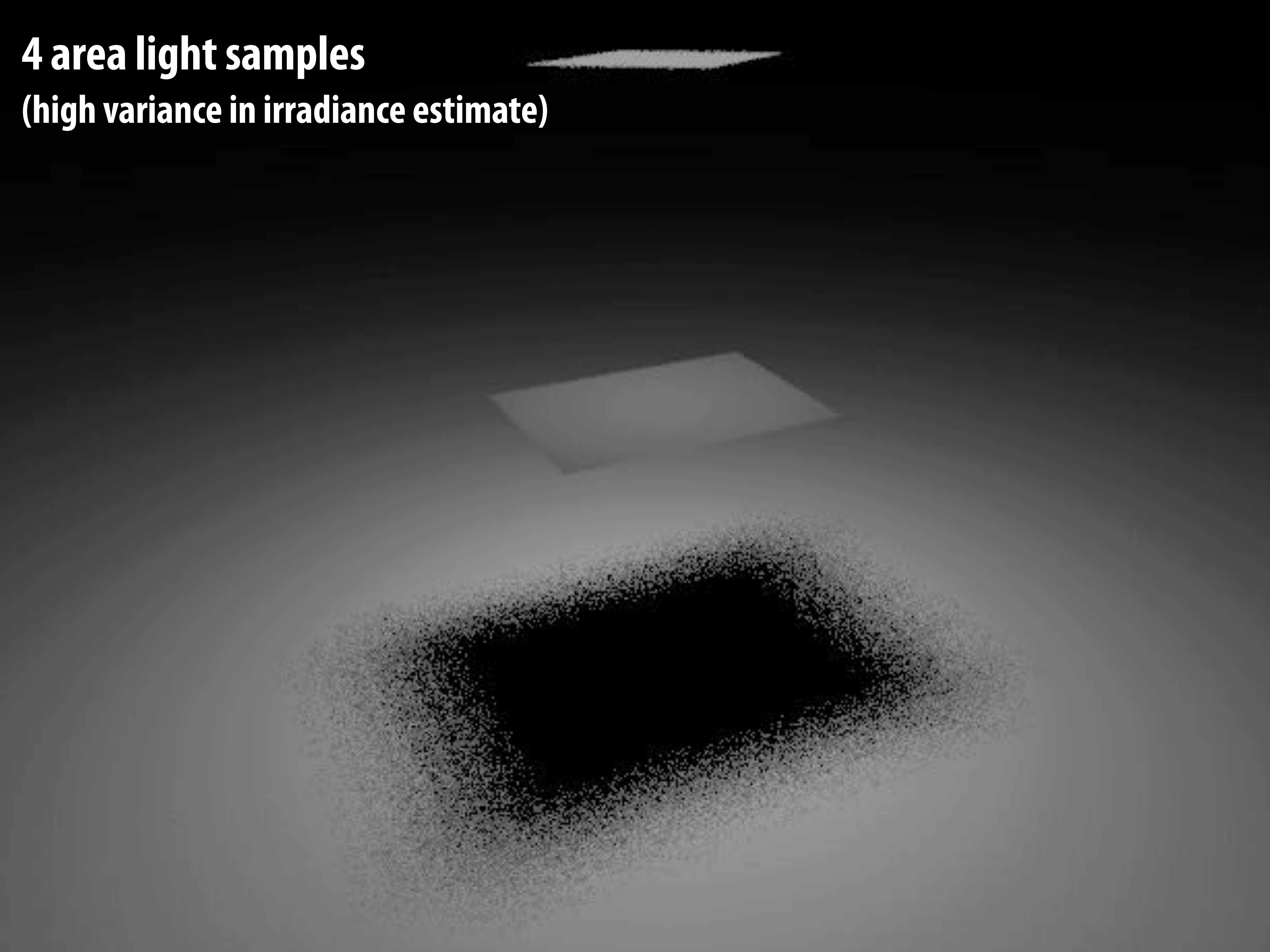


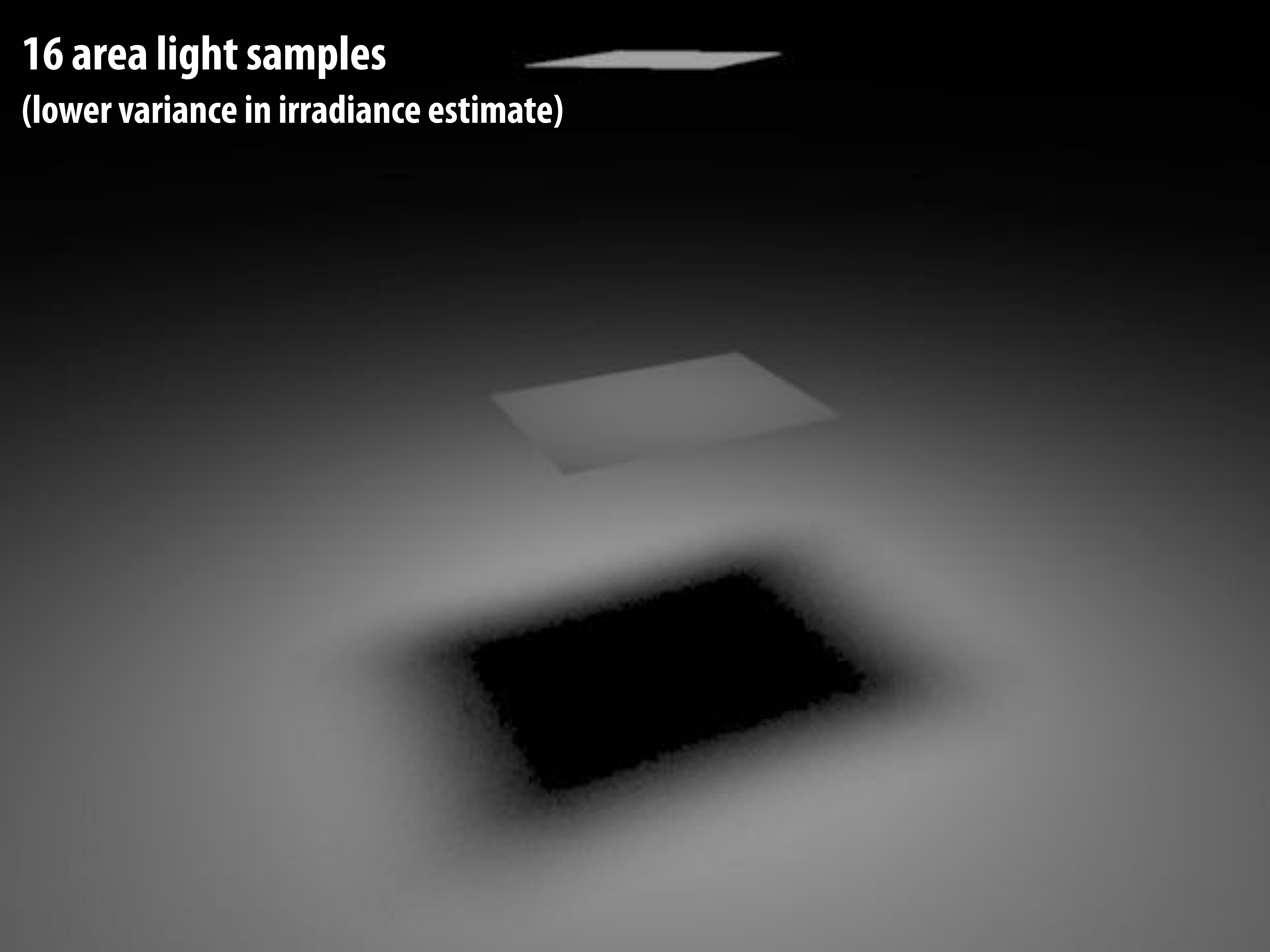
- **For all samples:**
  - Randomly pick a point P_L on the area light:
  - Determine if surface point P is in shadow with respect to P_L
  - Compute contribution to illumination from P_L

Implication: must trace many rays per pixel!

4 area light samples

(high variance in irradiance estimate)





16 area light samples

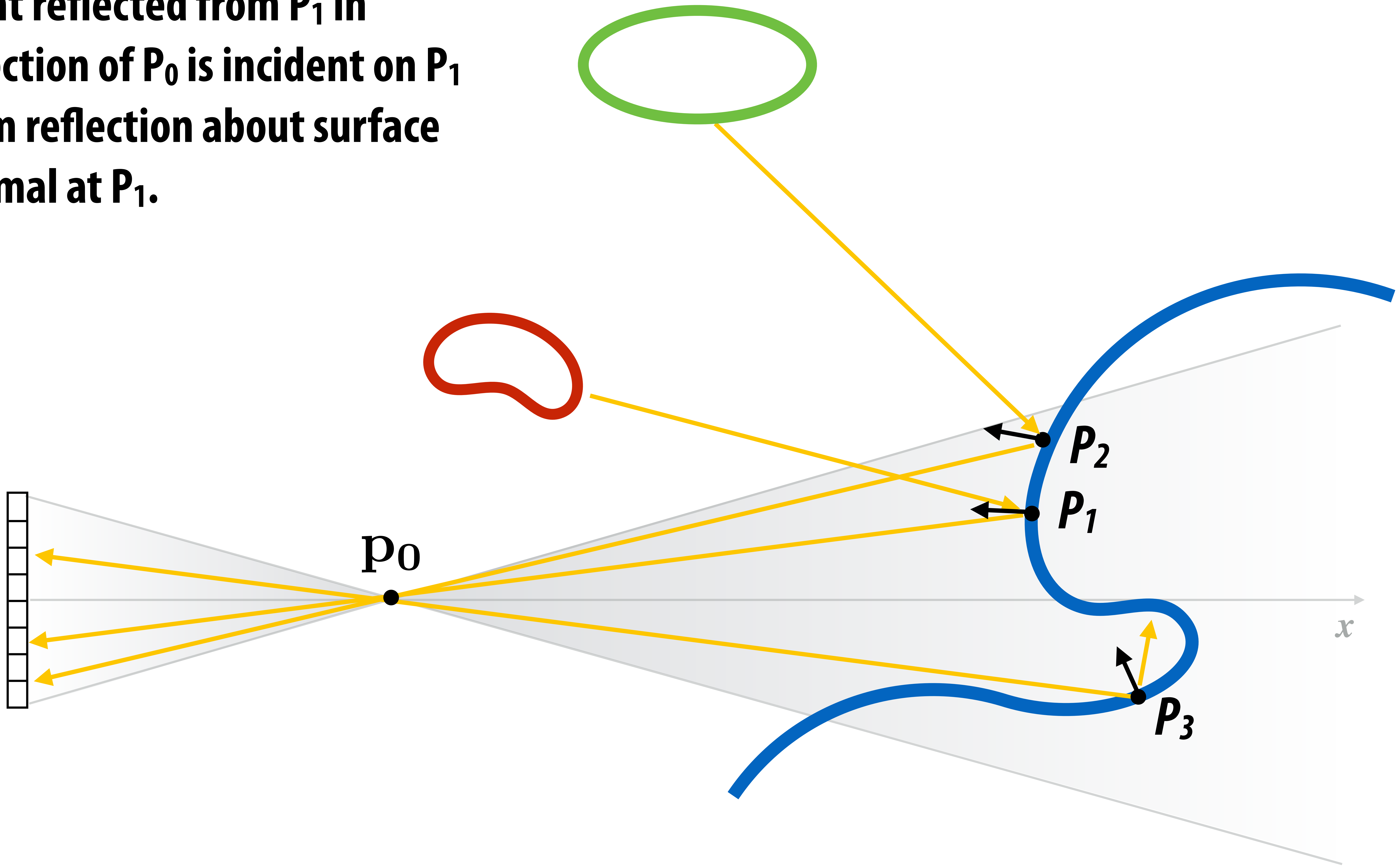
(lower variance in irradiance estimate)

Reflections

Image credit: NVIDIA

Recall: perfect mirror reflection

Light reflected from P_1 in direction of P_0 is incident on P_1 from reflection about surface normal at P_1.



Direct illumination + reflection + transparency

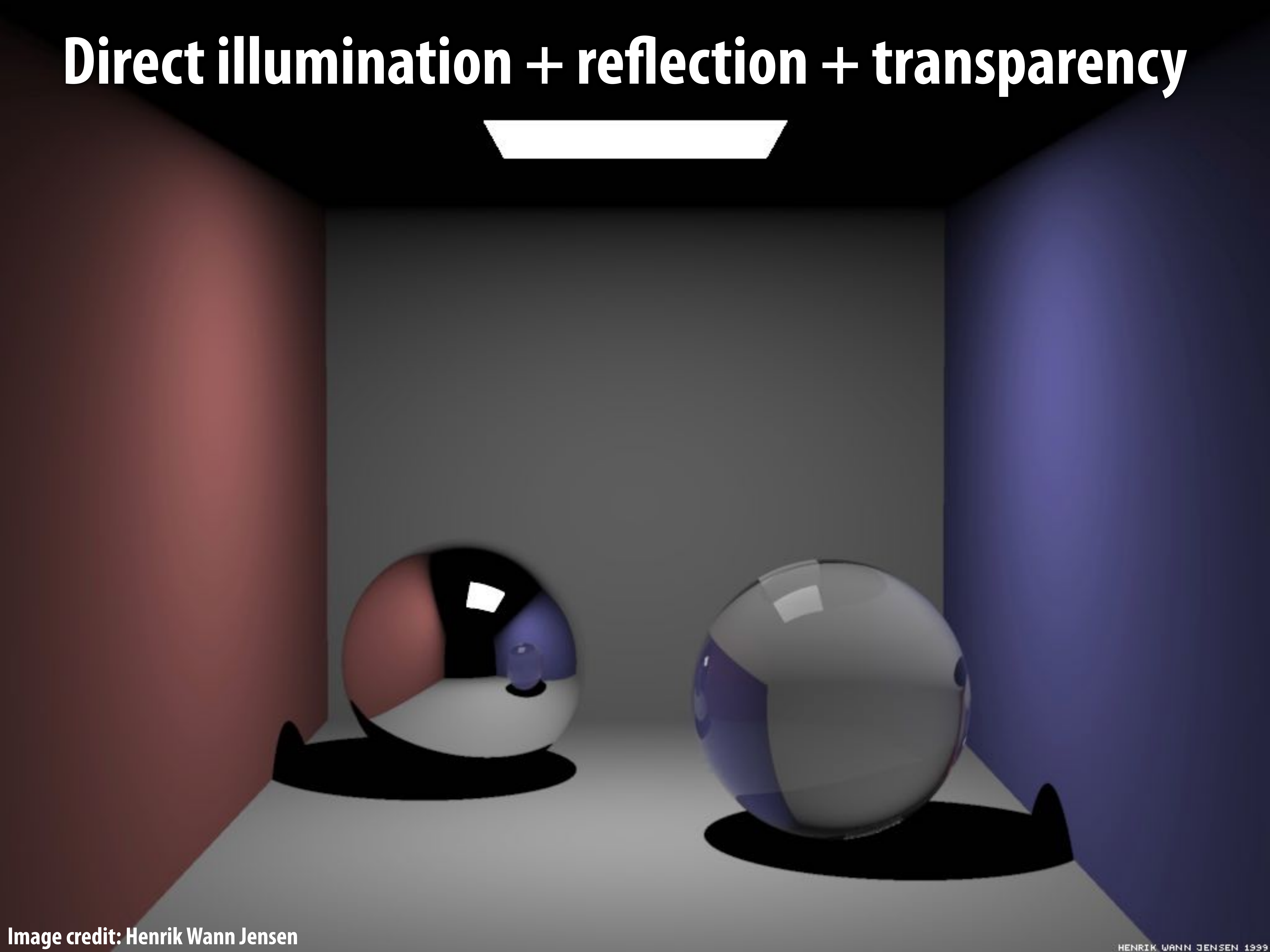


Image credit: Henrik Wann Jensen

Global illumination solution

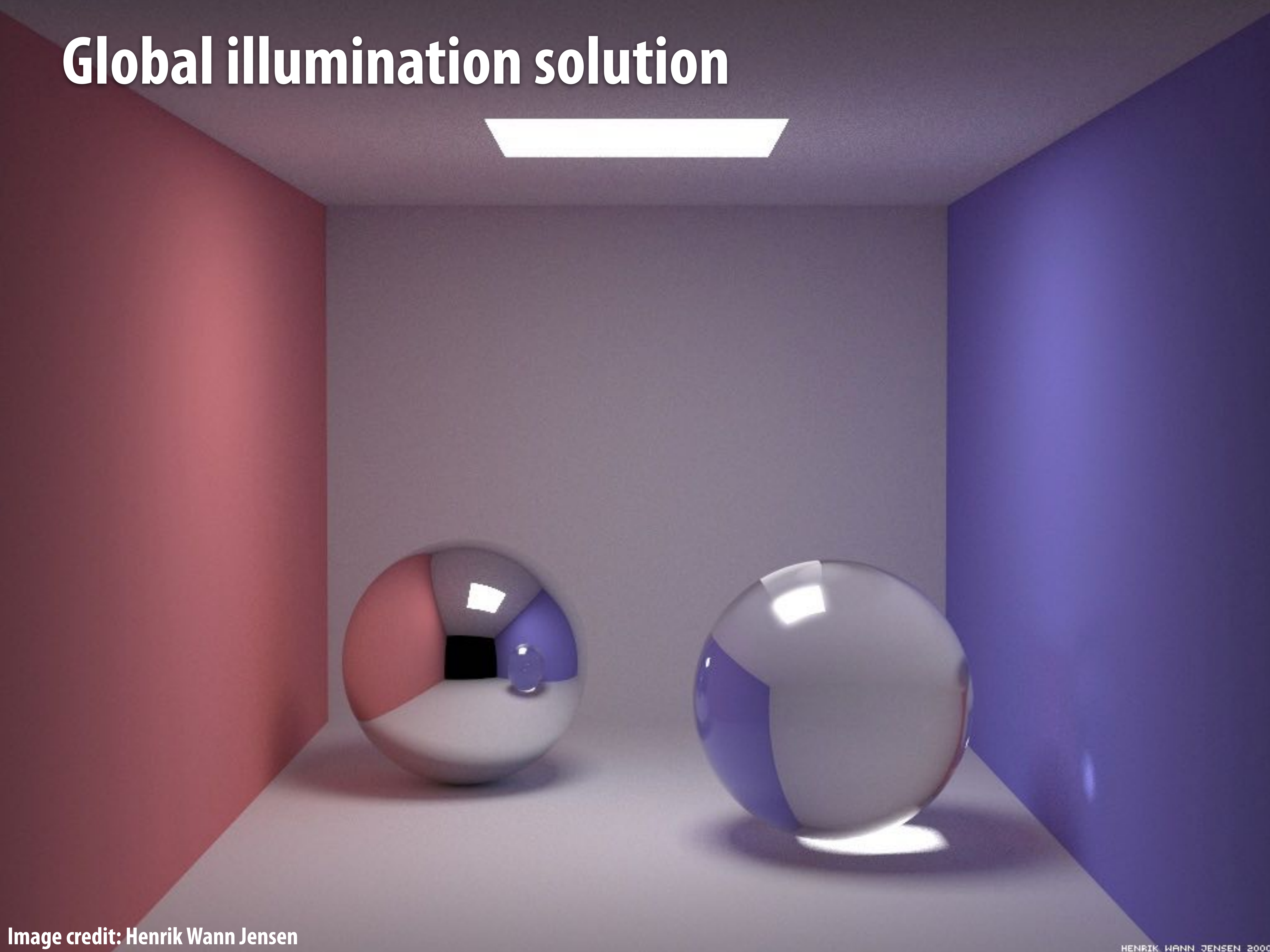
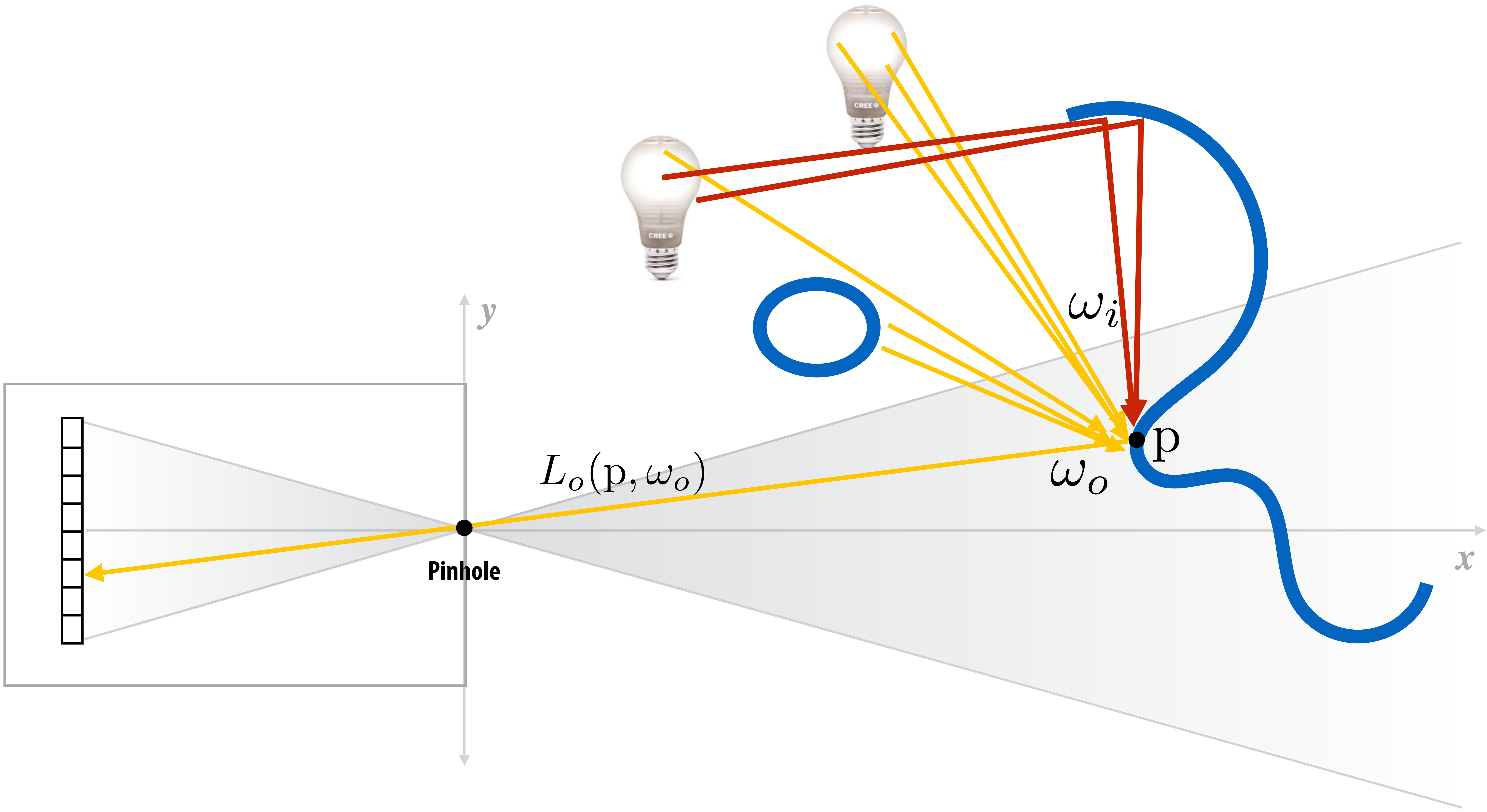


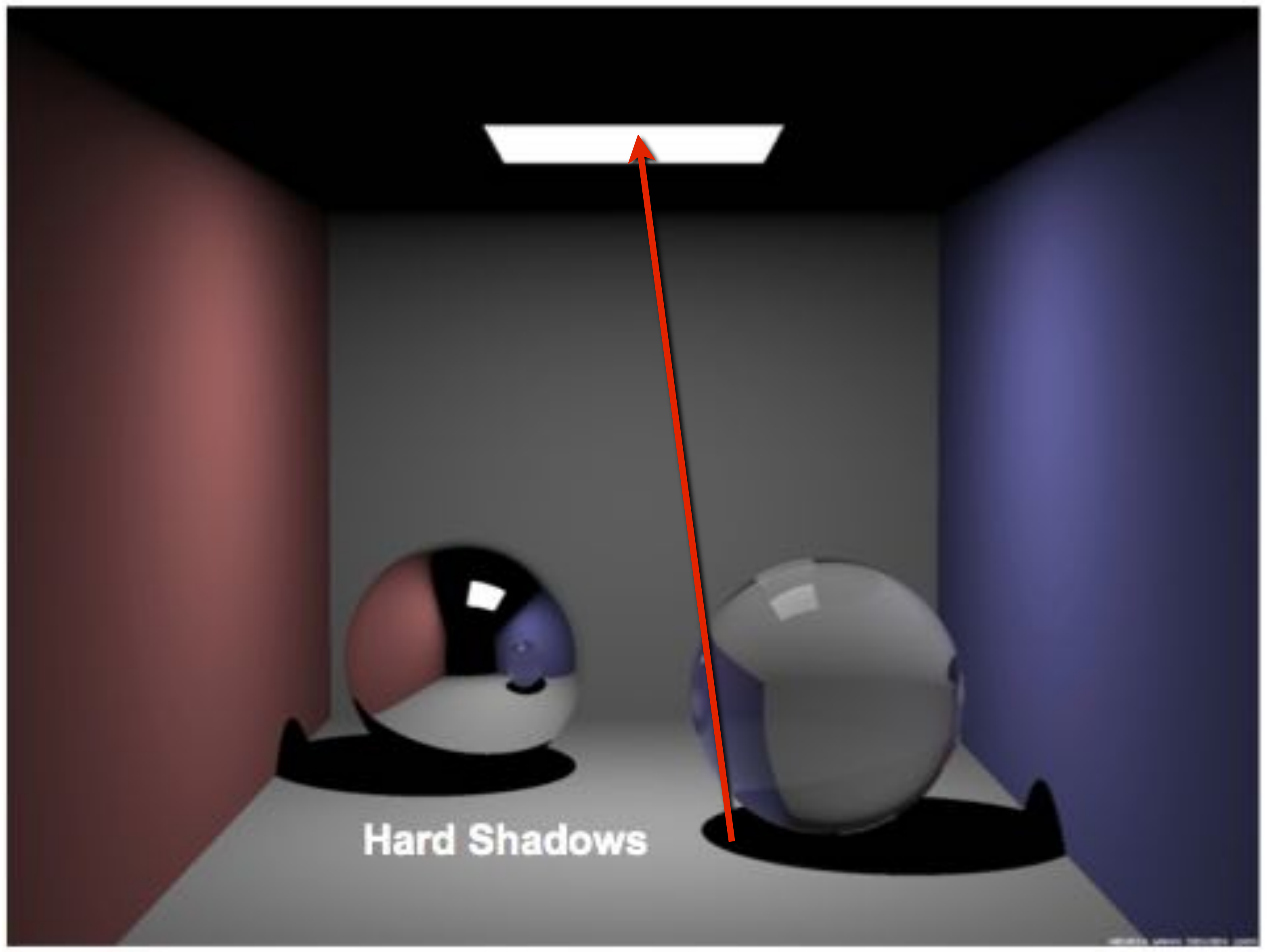
Image credit: Henrik Wann Jensen

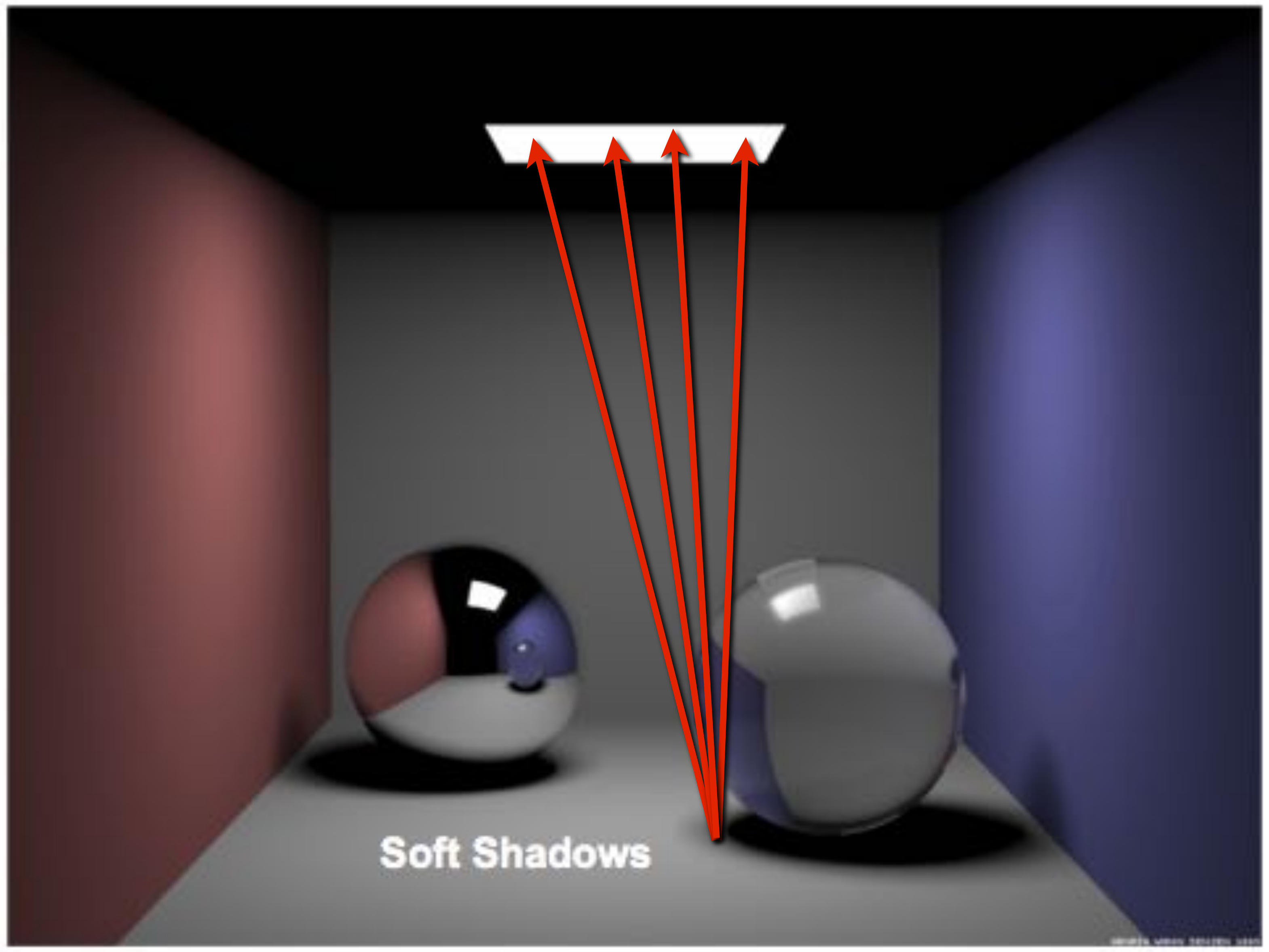
Accounting for indirect illumination

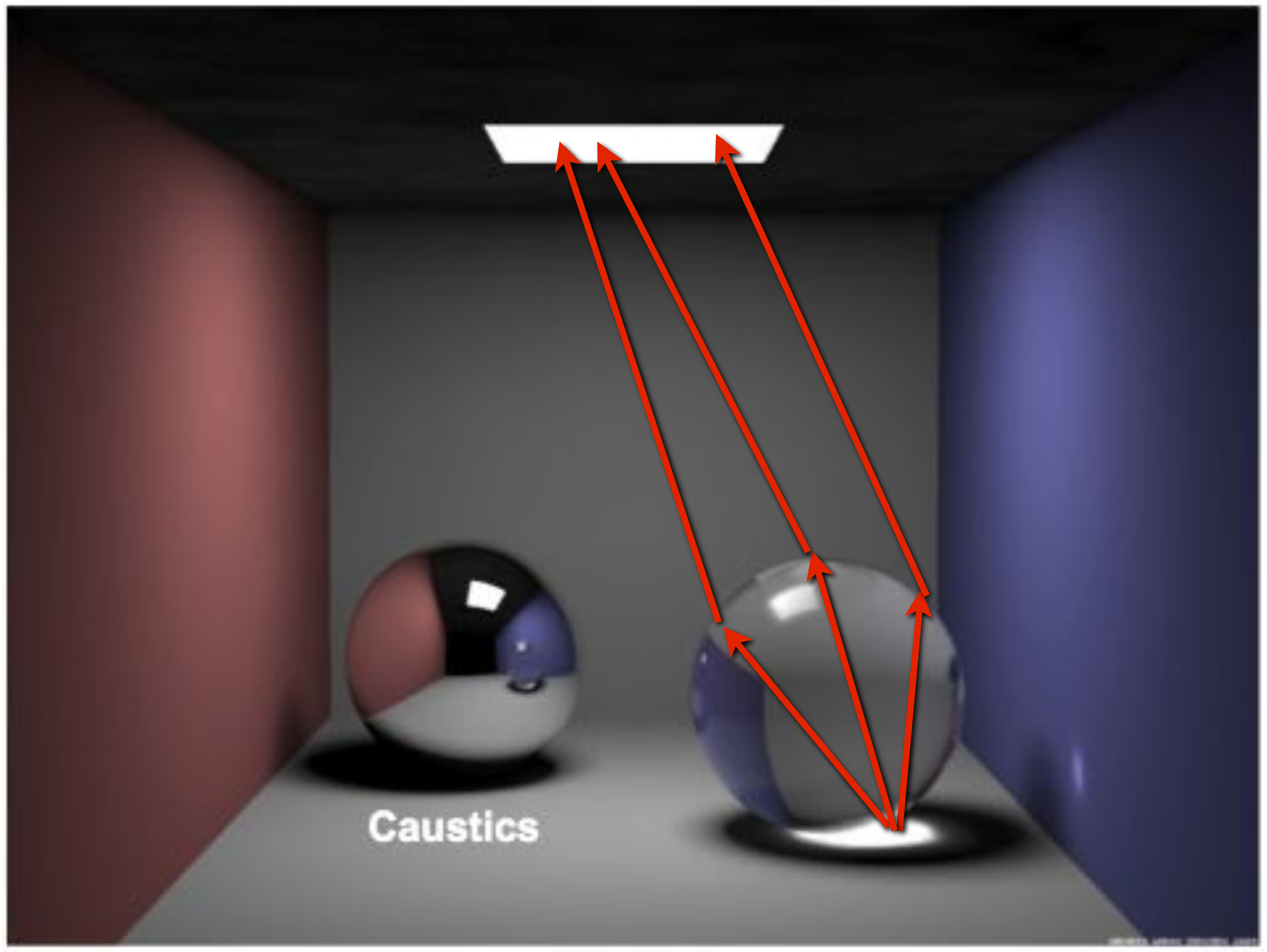


Implication: even more ray tracing per pixel!

Sampling light paths







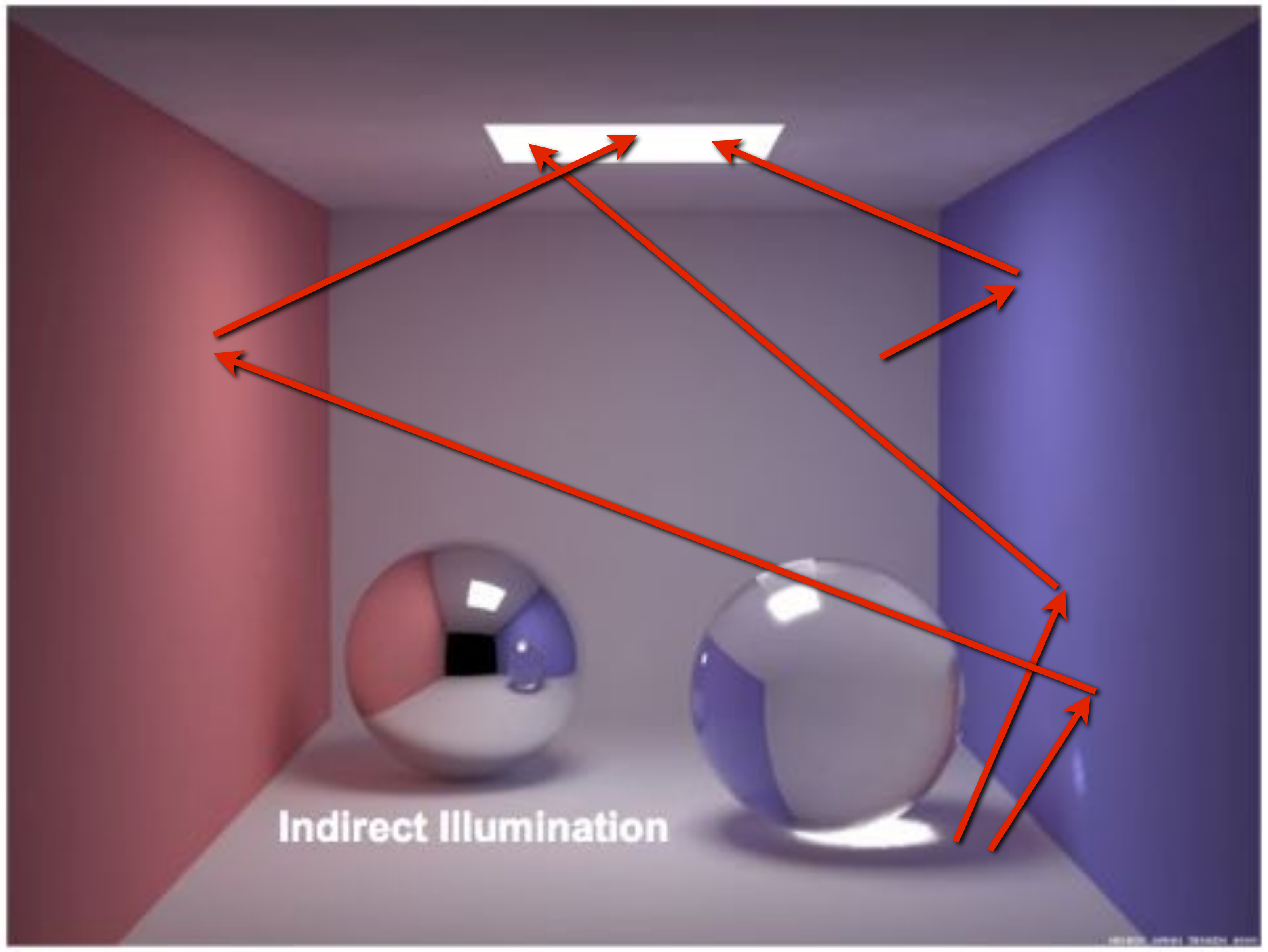


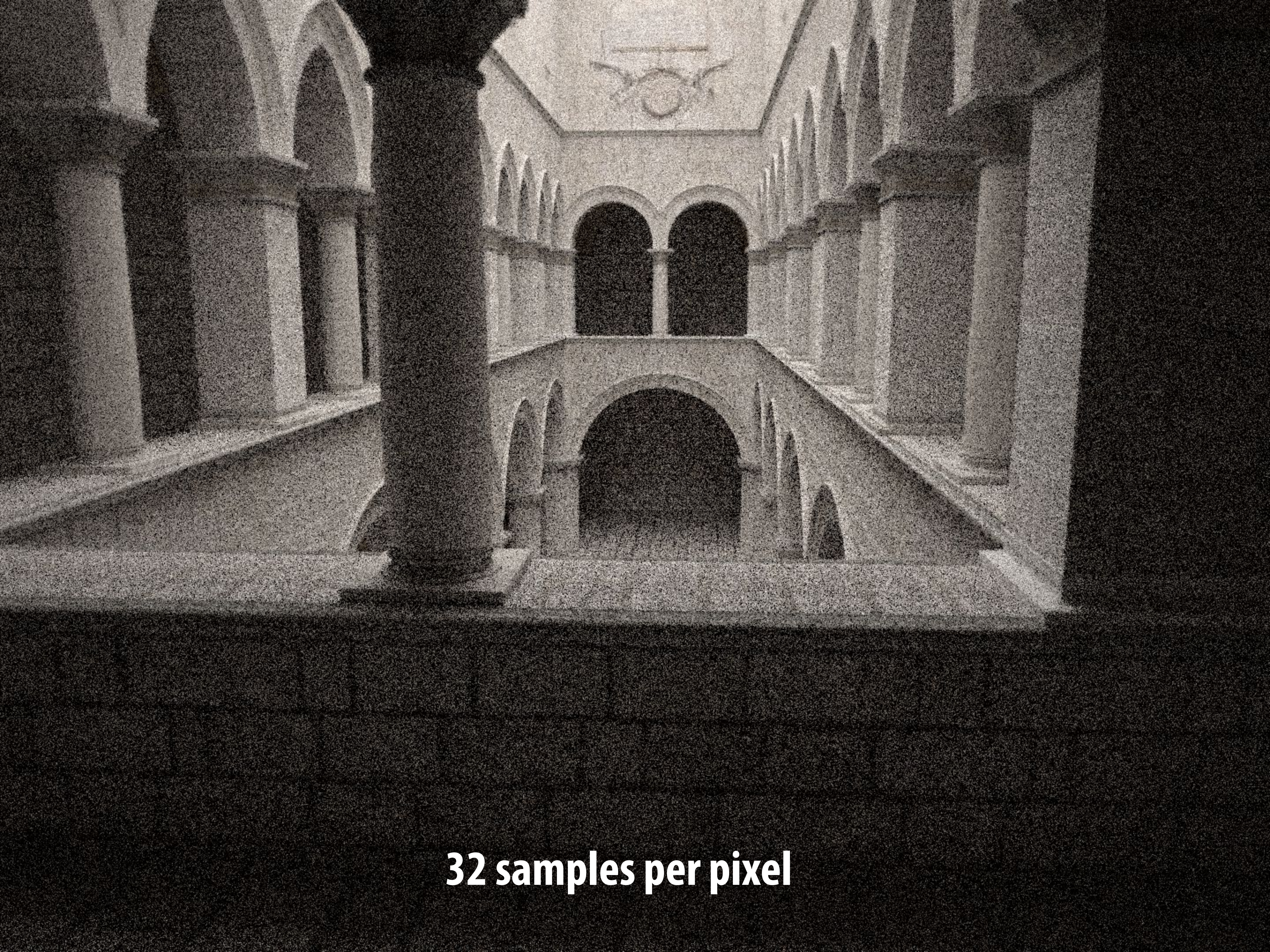
Image credit: Wann Jensen, Hanrahan

Direct illumination

One-bounce global illumination

Sixteen-bounce global illumination

One sample per pixel



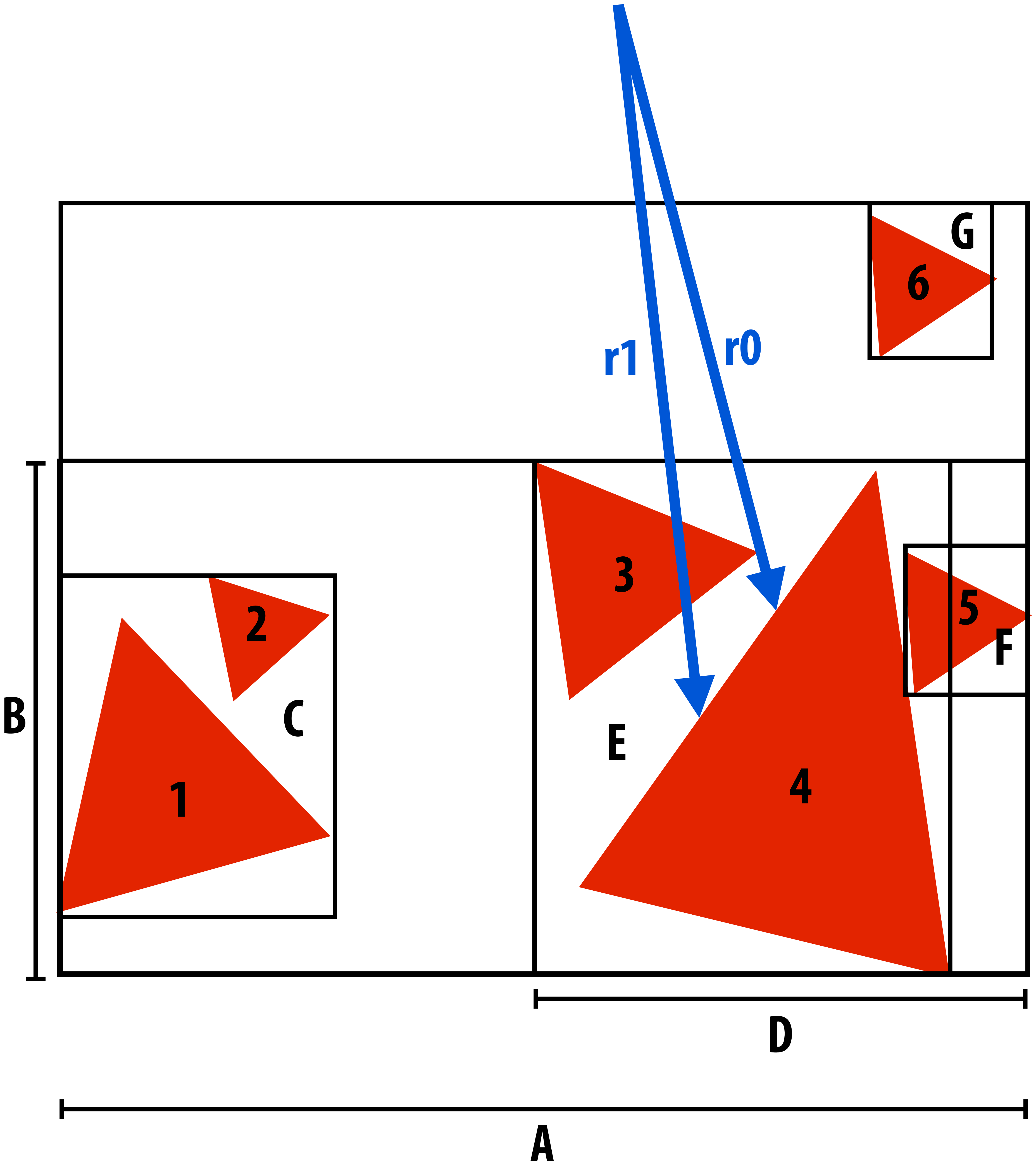
32 samples per pixel

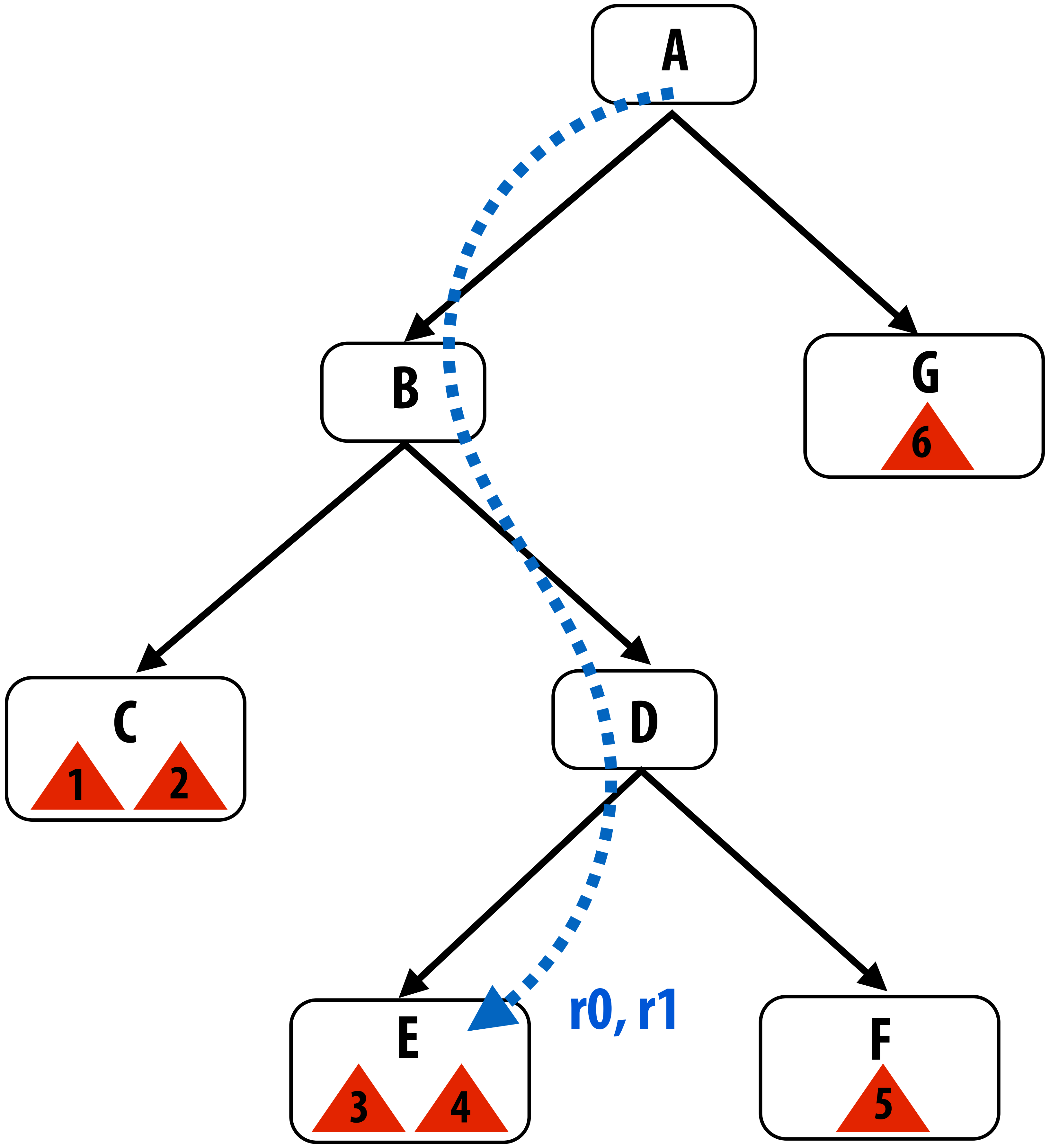
1024 samples per pixel

Understanding ray coherence

Ray traversal “coherence”

r0 visits nodes: A, B, D, E…
r1 visits nodes: A, B, D, E…

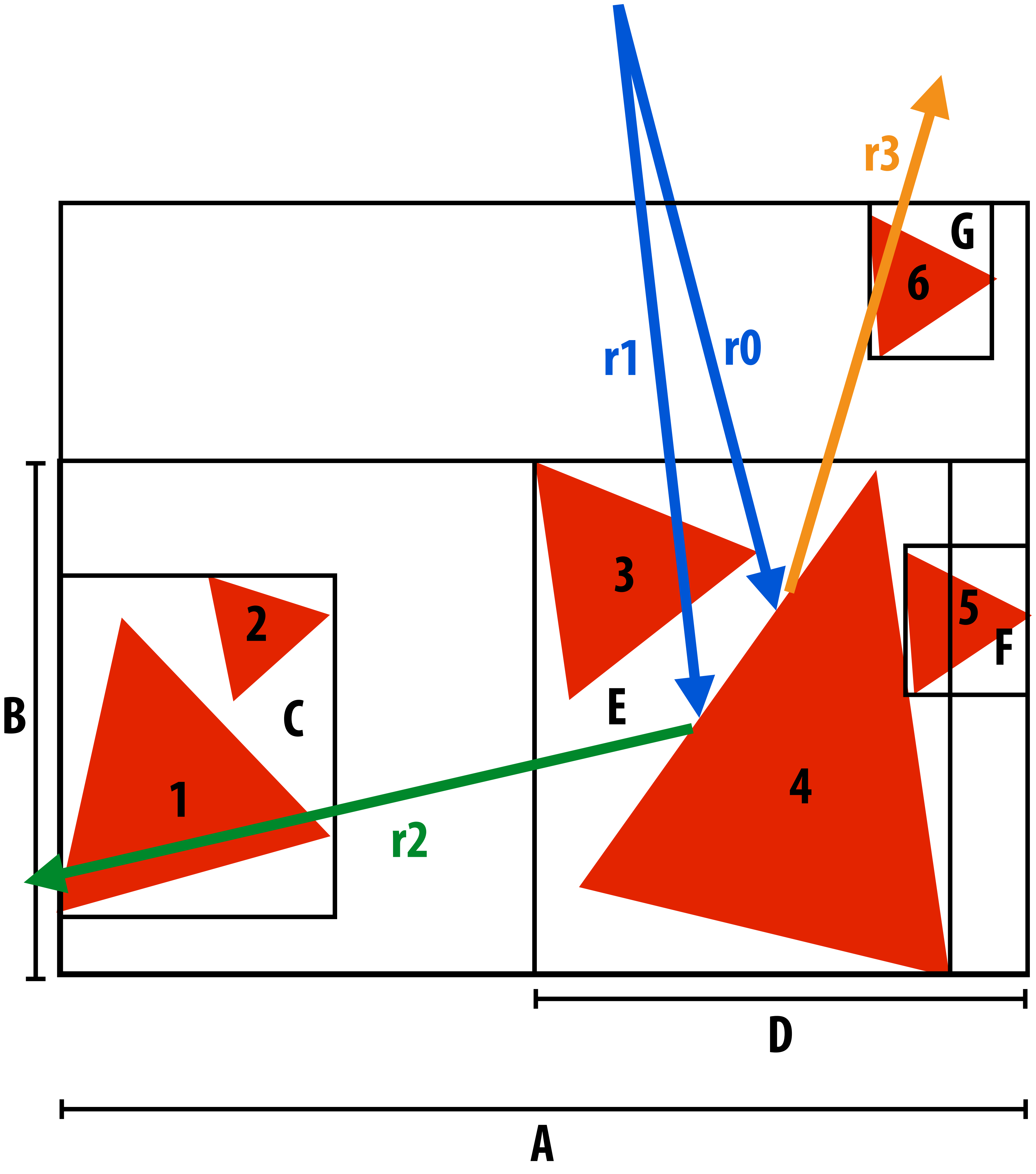


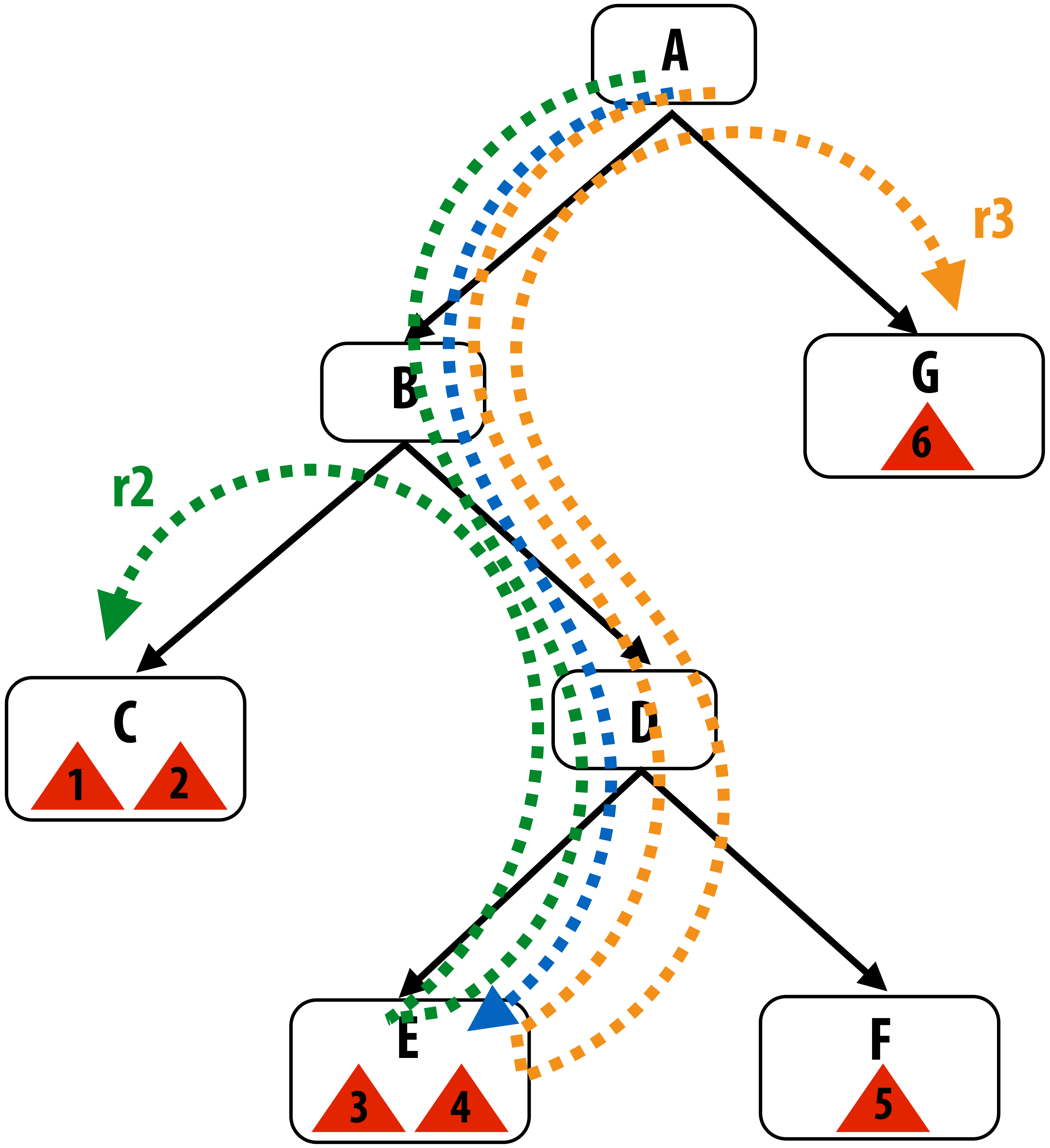


Bandwidth reduction: BVH nodes (and triangles) loaded into cache for computing scene intersection with r0 are cache hits for r1

Ray traversal “divergence”

r0 visits nodes: A, B, D, E…
r1 visits nodes: A, B, D, E…
r2 visits nodes: A, B, D, E, C…
r3 visits nodes: A, B, D, E, G…

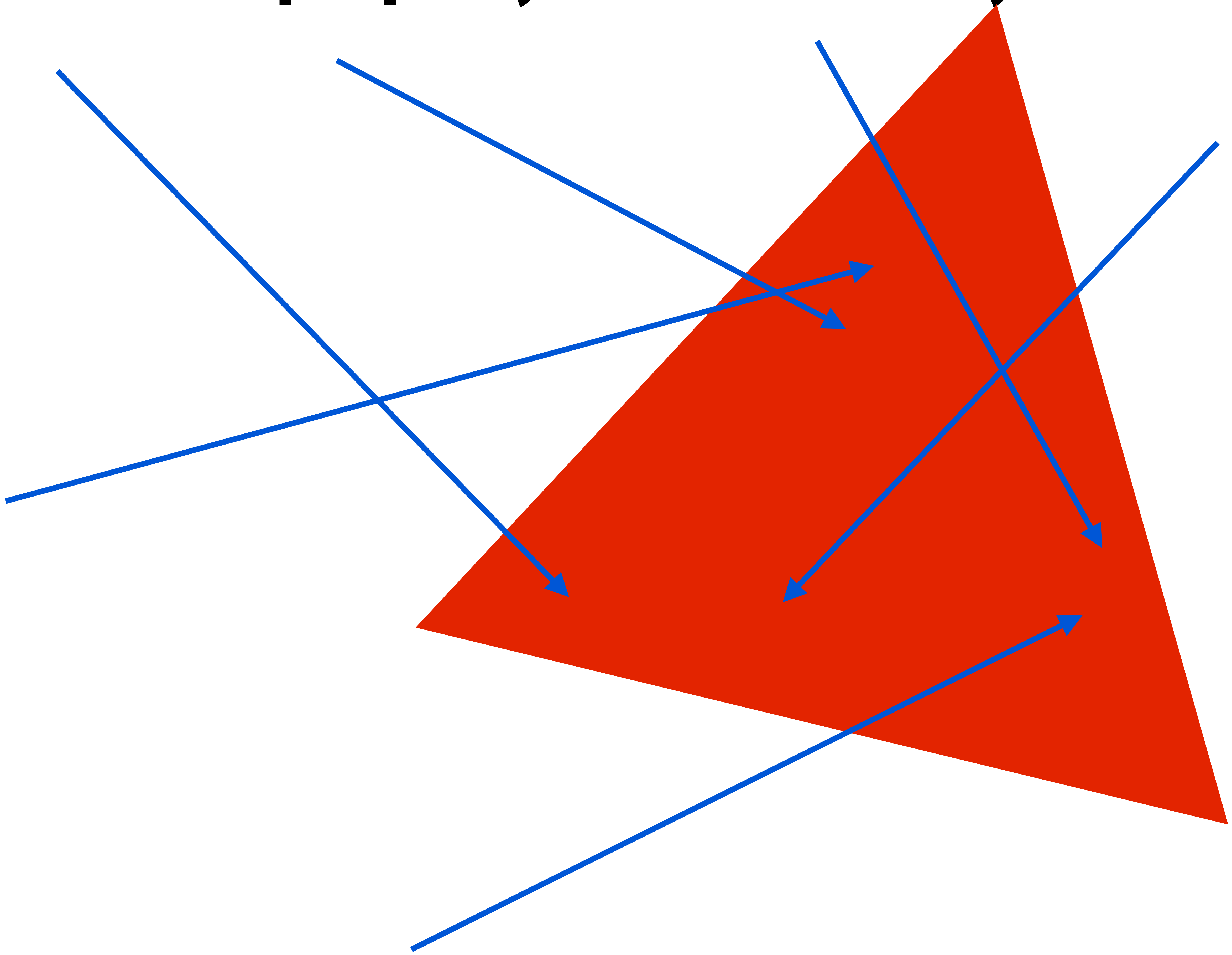




R2 and R3 require different BVH nodes and triangles

Incoherent rays

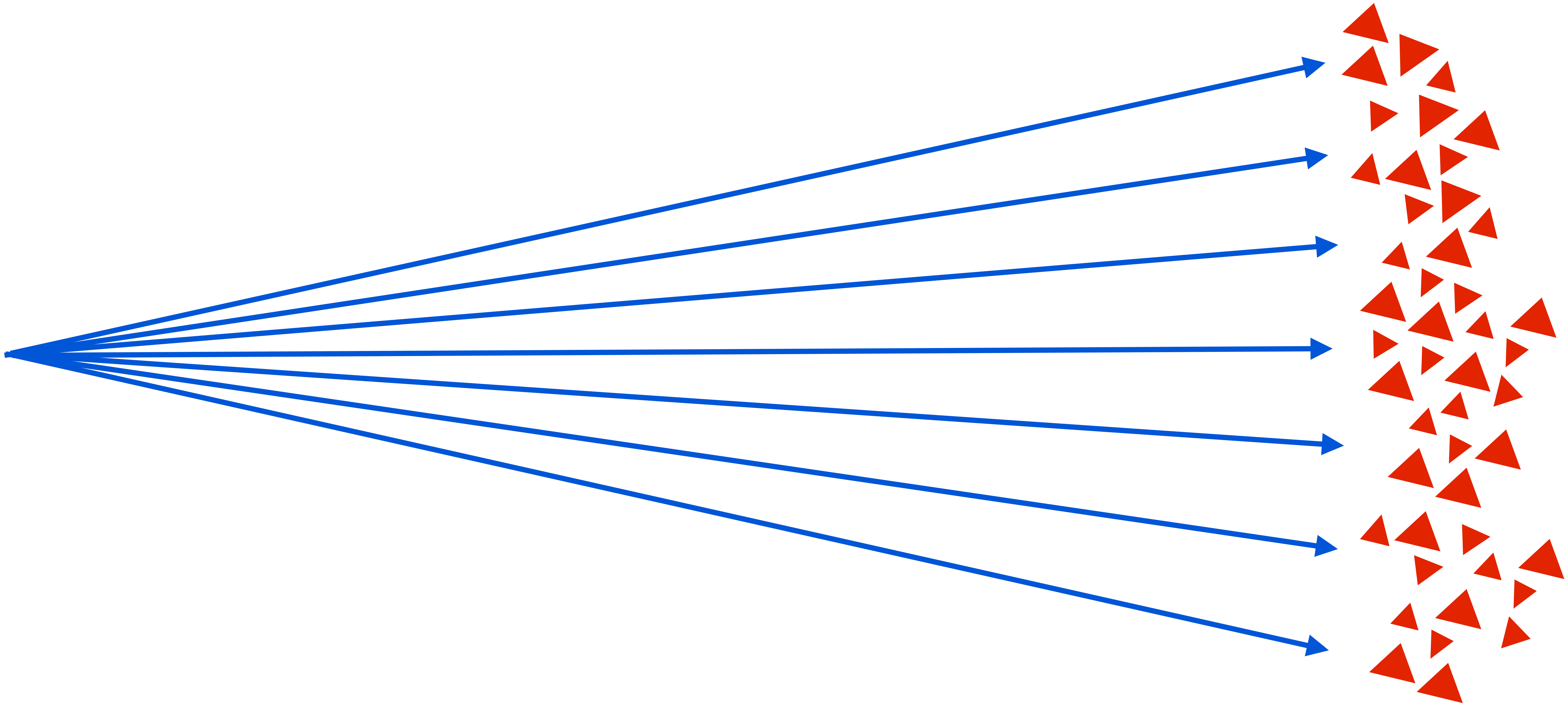
Incoherence is a property of both the rays and the scene



Example: random rays are "coherent" with respect to the BVH if the scene is one big triangle!

Incoherent rays

Incoherence is a property of <u>both</u> the rays and the scene



Similarly oriented rays from the same point become "incoherent" with respect to lower nodes in the BVH if a scene is overly detailed

(Side note: this suggests the importance of choosing the right geometric level of detail)

Ray incoherence

Nearby rays may hit different surfaces, with different "shaders"

Consider implications for SIMD processing

Perfect specular reflection material

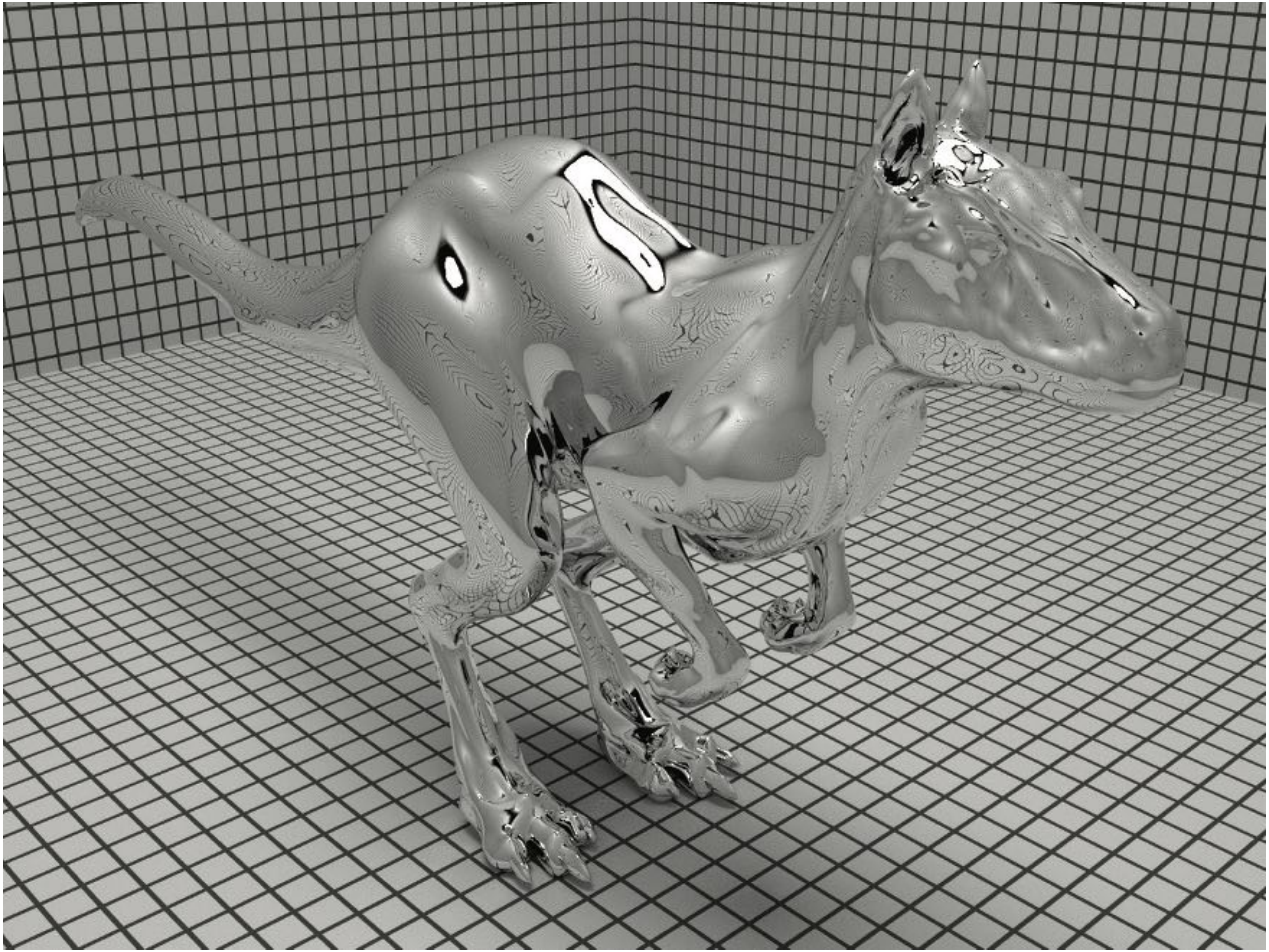
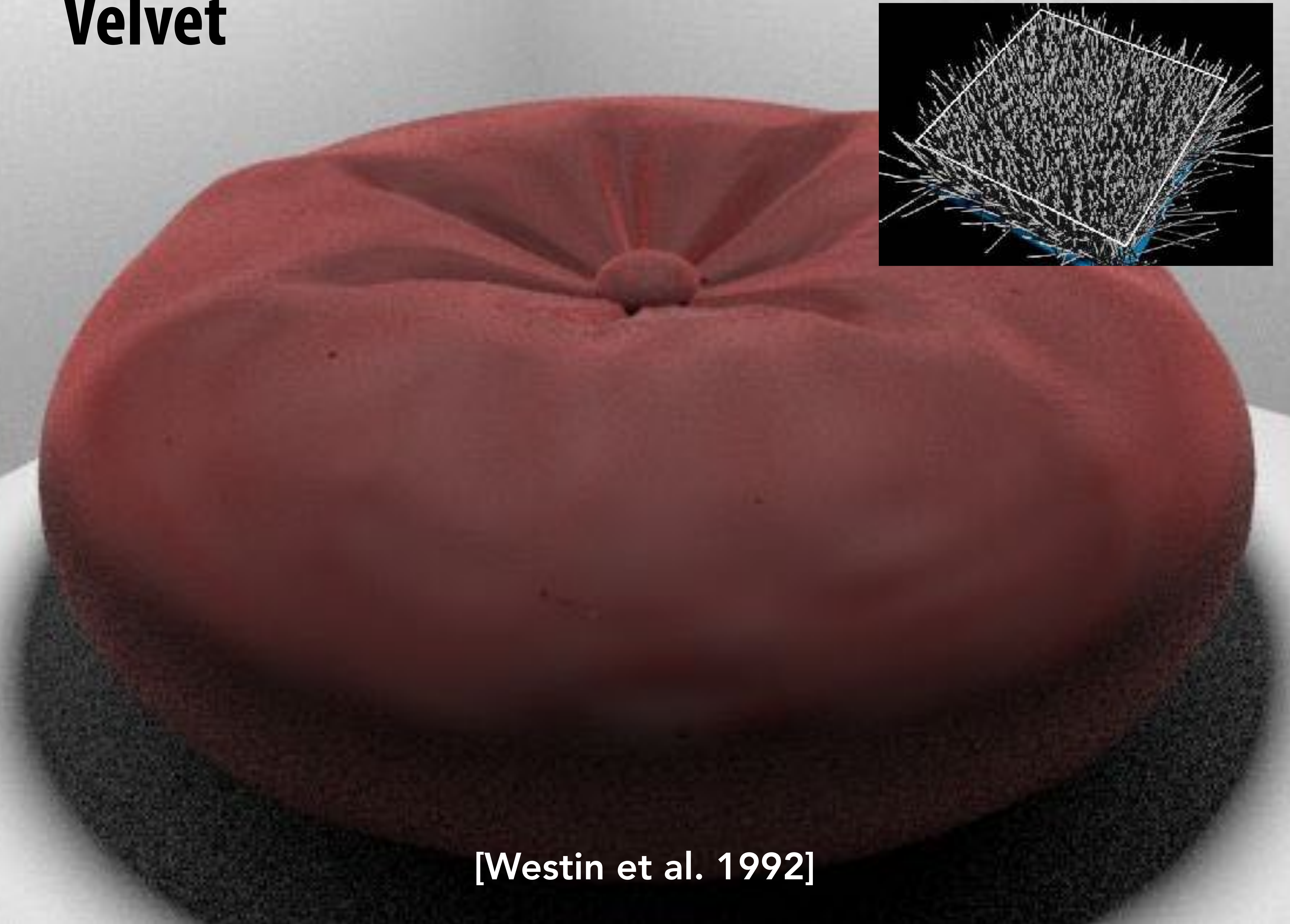


Image credit: PBRT

More complex materials: glint

Velvet



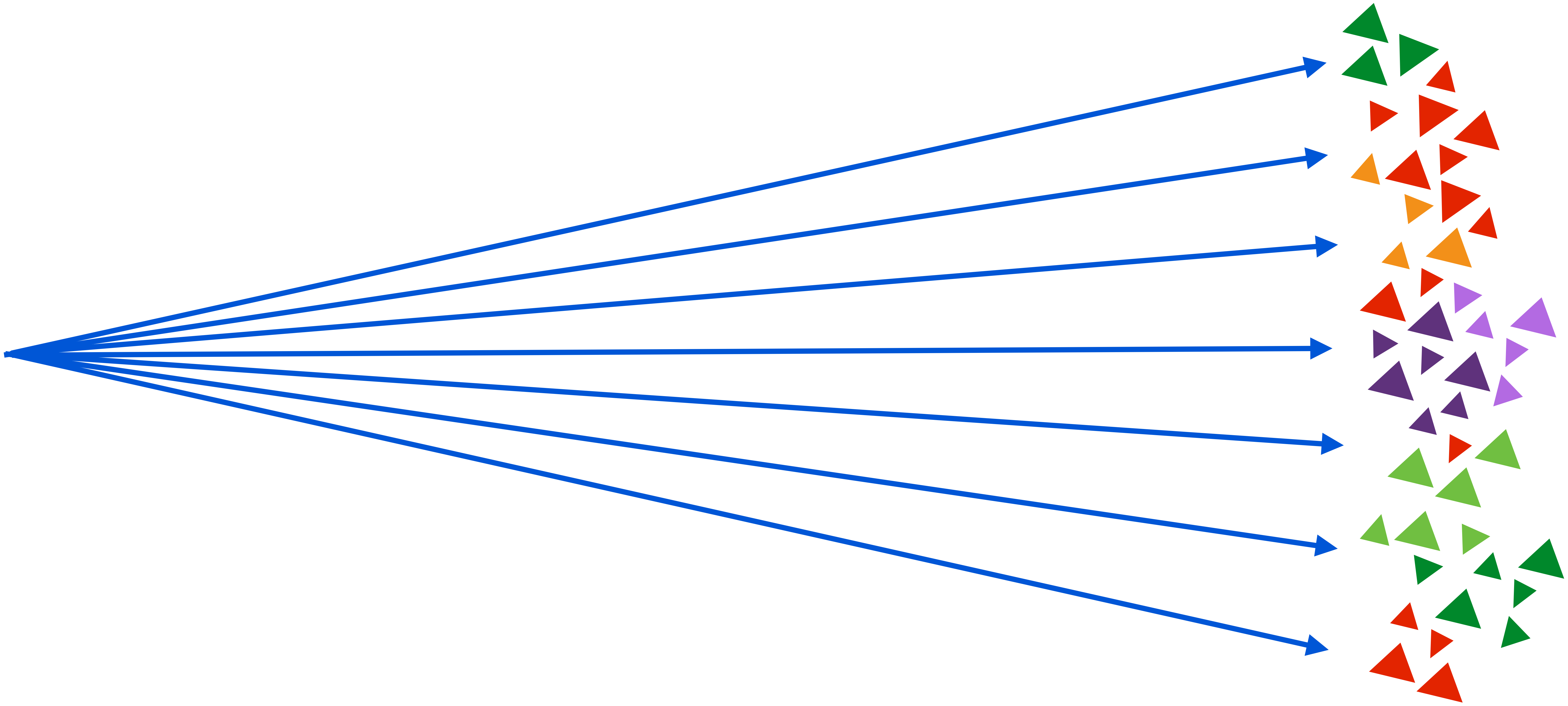
[Westin et al. 1992]

Subsurface scattering

When rays hit different surfaces…

Surface shading incoherence:

Different code paths needed to compute the reflectance of different materials

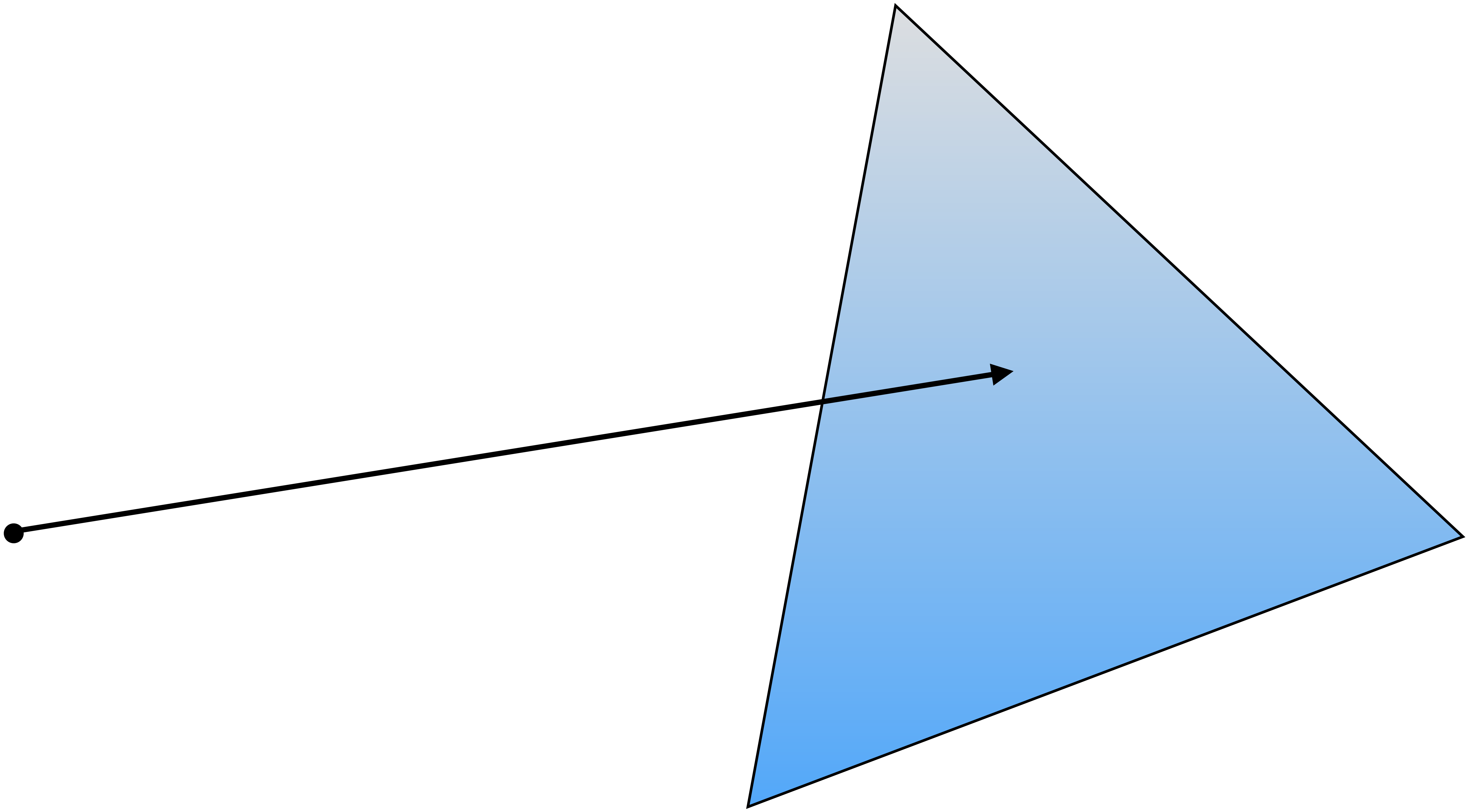


Parallelizing ray-scene intersection

- **Parallelize across rays**
  - **Simultaneously intersect multiple rays with scene**
  - **Enables wide data-parallel execution**

Parallelizing single ray-scene queries

(Intra-ray parallelism)



Parallelize ray-box, ray-triangle intersection

- Given one ray and one bounding box, there are opportunities for SIMD processing
  - Can use 3 of 4 vector lanes (e.g., xyz work, multiple point-plane tests, etc.)

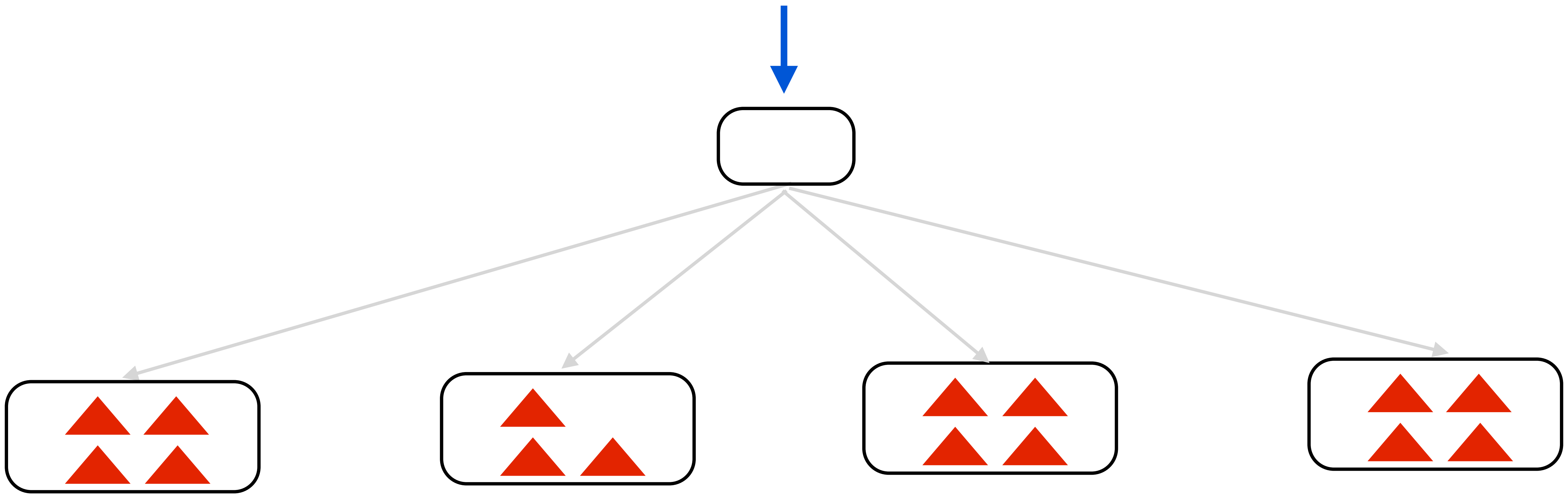
- Similar SIMD parallelism in ray-triangle test at BVH leaf

- If BVH leaf nodes contain multiple triangles, can parallelize ray-triangle intersection across these triangles

Parallelize over BVH child nodes

[Wald et al. 2008]

- Idea: use wider-branching BVH (test single ray against multiple child node bboxes in parallel)
  - Empirical result: BVH with branching factor four has similar work efficiency to branching factor two
  - BVH with branching factor 8 or 16 is less work efficient (diminished benefit of leveraging SIMD execution)



Parallelizing BVH build

- To compute splits, parallelize across primitives
  - Recall binned SAH build is largely generating a histogram

- Divide and conquer parallelism
  - After a split, both subtrees can be processed in parallel

Building a low-quality BVH quickly

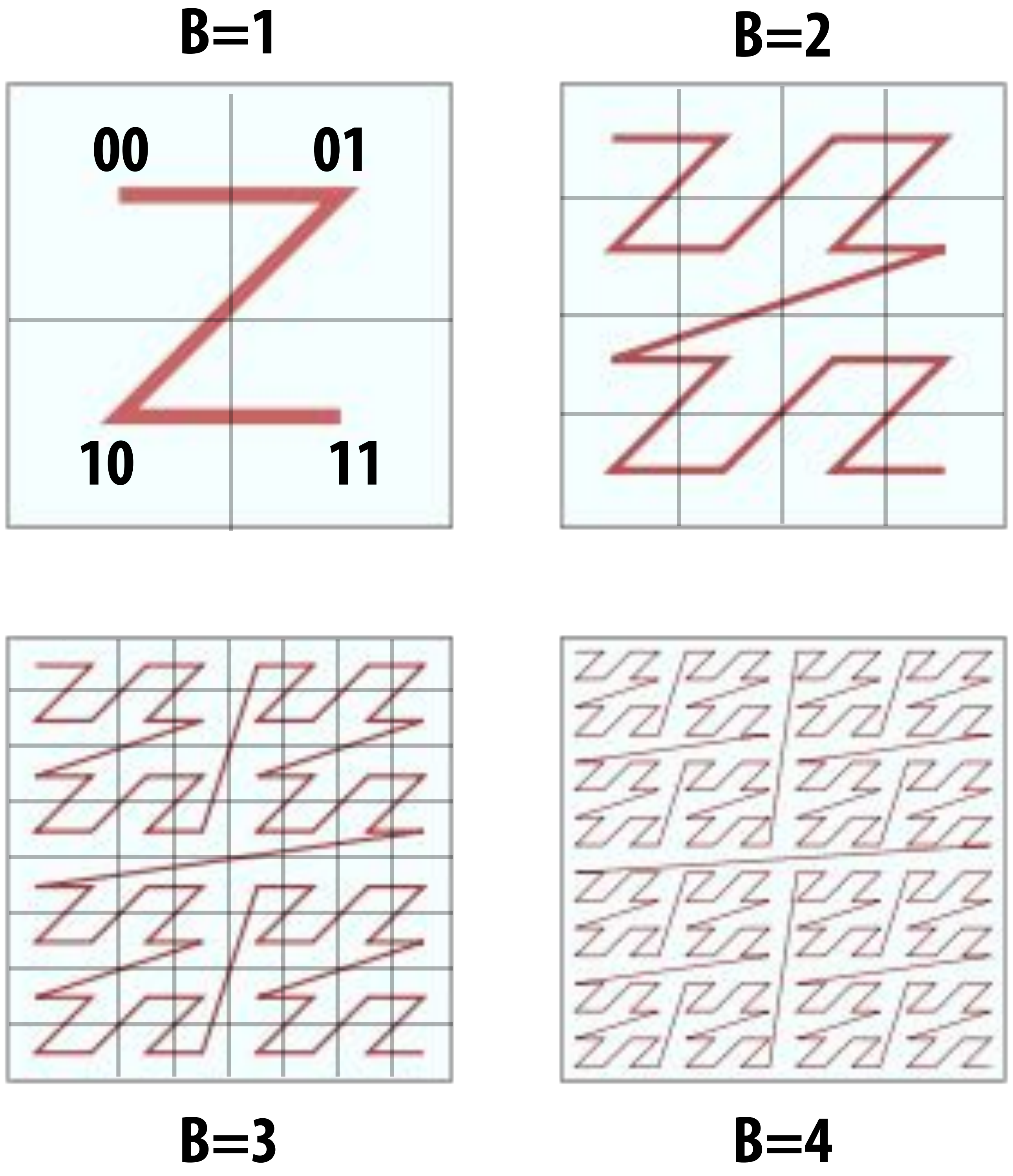
1. Discretize each dimension of scene into 2^B cells
2. Compute index of centroid of bounding box of each primitive:

   (c_i, c_j, c_k)
3. Interleave bits of c_i, c_j, c_k to get 3B bit-Morton code
4. Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
   - O(N) radix sort

Simple, highly parallelizable BVH build:

```
Partition(int i, primitives):
  node.bbox = bbox(primitives)
  (left, right) = partition primitives by bit i
  if there are more bits:
     Partition(left, i+1);
     Partition(right, i+1);
  else:
     make a leaf node
```

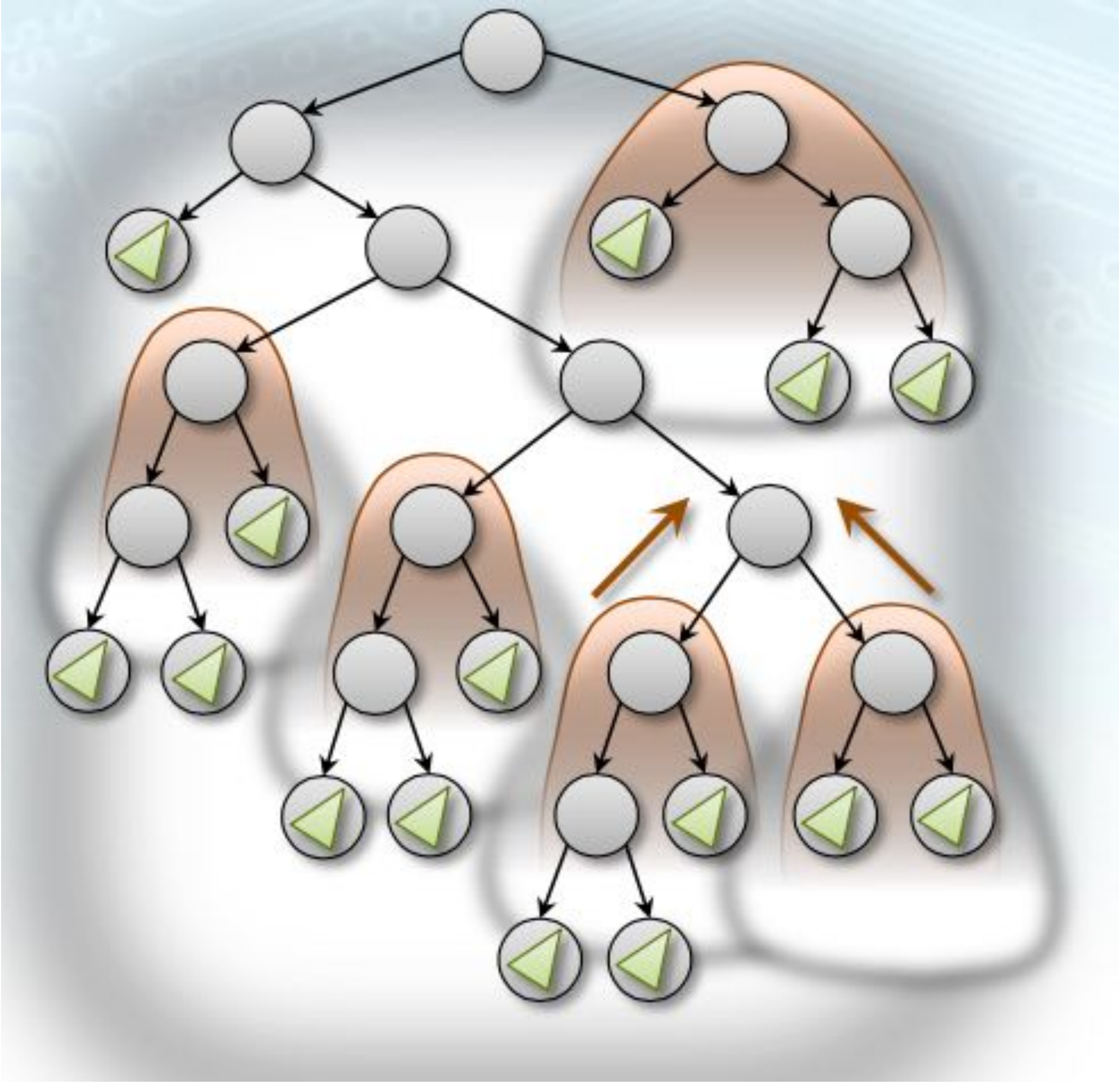
2D Morton Order



Modern, fast BVH construction schemes

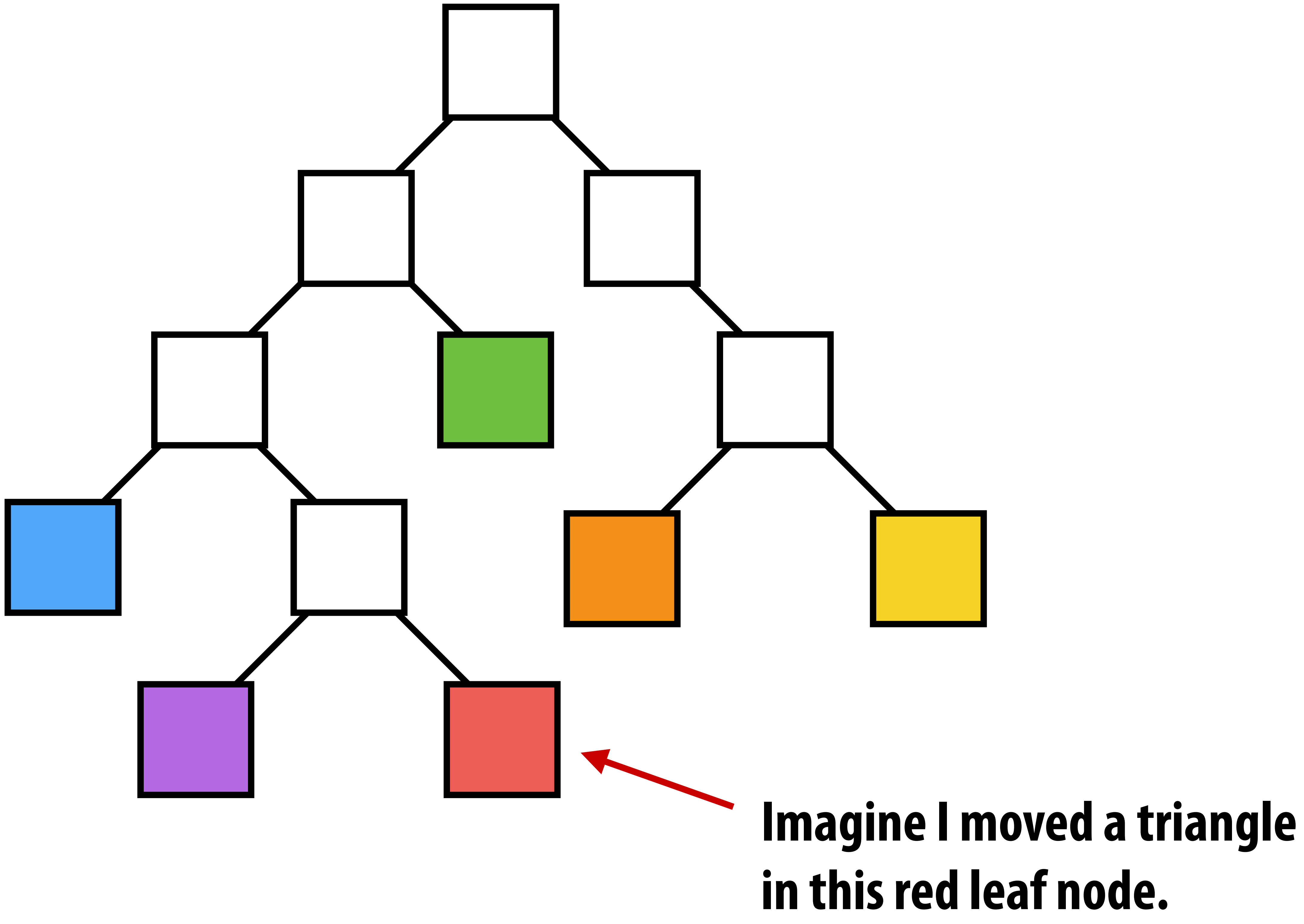
- **Combine greedy "top-down" divide-and-conquer build with "bottom up" construction techniques**
- **Build low-quality BVH quickly using Morton Codes**
- **Use initial BVH to <u>accelerate</u> construction of high-quality BVH**
- **Example: [Kerras 2013]**

```
For all treelets of size < N in original "low
quality" BVH: (in parallel)
    try all possible trees, keeping "optimal"
    topology that minimizes SAH for treelet
```



Refitting (instead of rebuilding) a BVH

- 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.

Ray tracing performance challenges

3D ray-triangle intersection math is expensive

Ray-scene intersection requires traversal through bounding volume hierarchy acceleration structure

- Unpredictable data access
- Rays are essentially randomly oriented after enough bounces

To simulate advanced effects renderer must trace many rays per pixel to reduce variance (noise) that results from numerical integration (via Monte Carlo sampling)