Lecture 15:

Real Time Ray Tracing Workload (Ray-scene intersection)

Visual Computing Systems
Stanford CS348K, Spring 2022

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

So was this


Ray Tracing: Off
Ray Tracing: On
RTX
ON

Modern real-time ray tracing

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

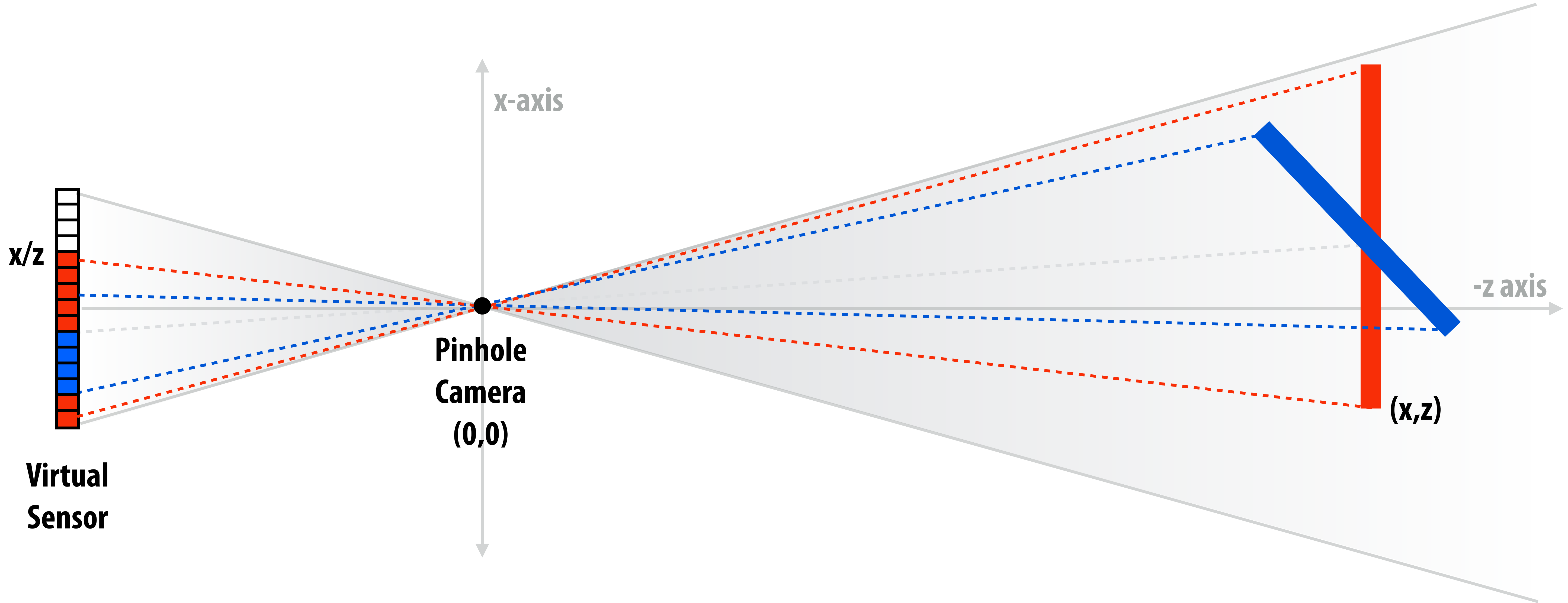
- **It is clear that the near future of real-time graphics will involve large amounts of ray tracing**

NVIDIA GeForce RTX 3080 GPU

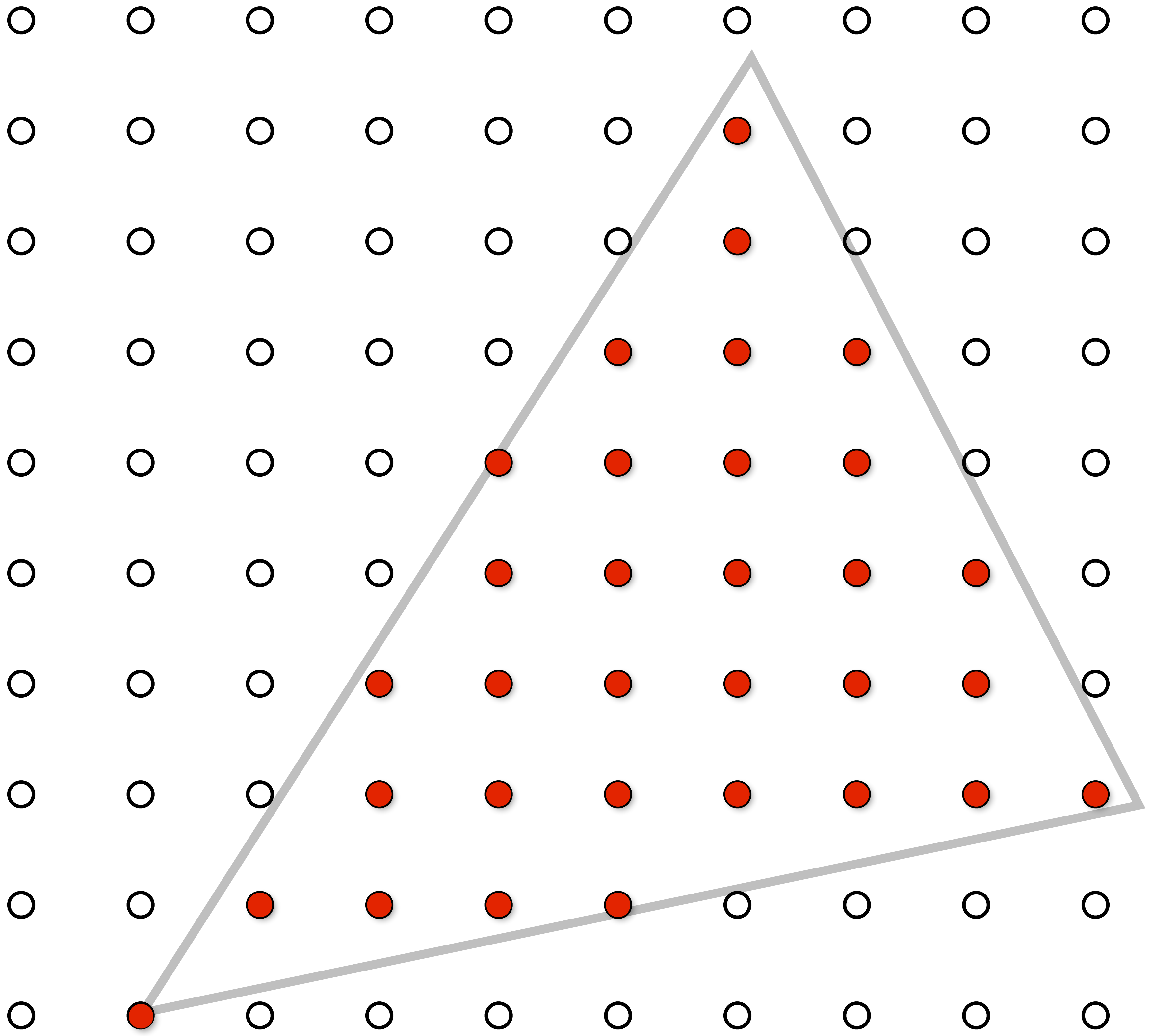
But first… a few positive things to say about rasterization

The visibility problem (as rasterization)

- What scene geometry is visible at each screen sample?
  - What scene geometry *projects* onto screen sample points? (coverage)
  - Which geometry is visible from the camera at each sample? (occlusion)

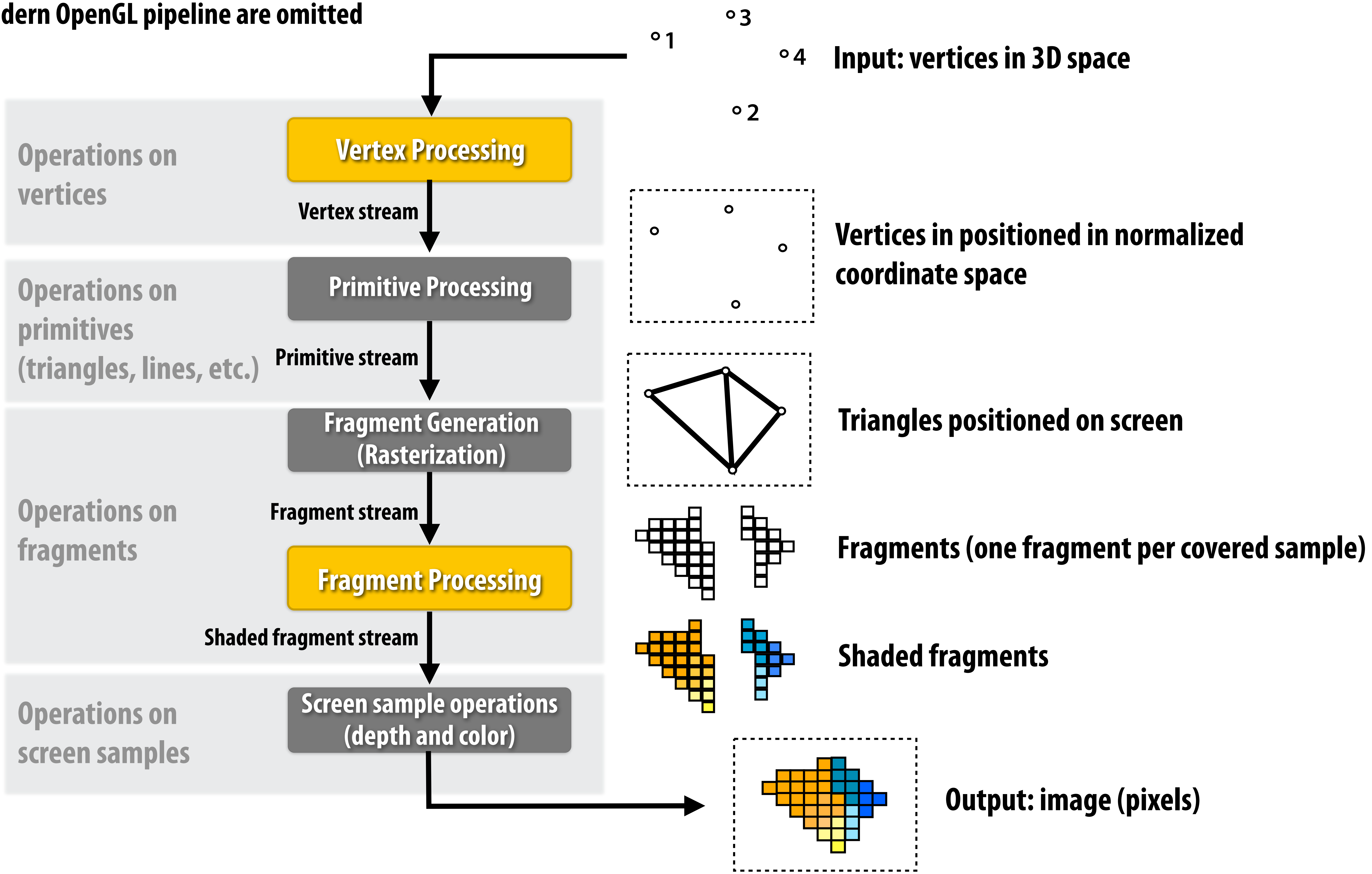


Sample coverage at pixel centers



Simple OpenGL/Direct3D graphics pipeline

* Several stages of the modern OpenGL pipeline are omitted



Basic rasterization algorithm

Sample = 2D point

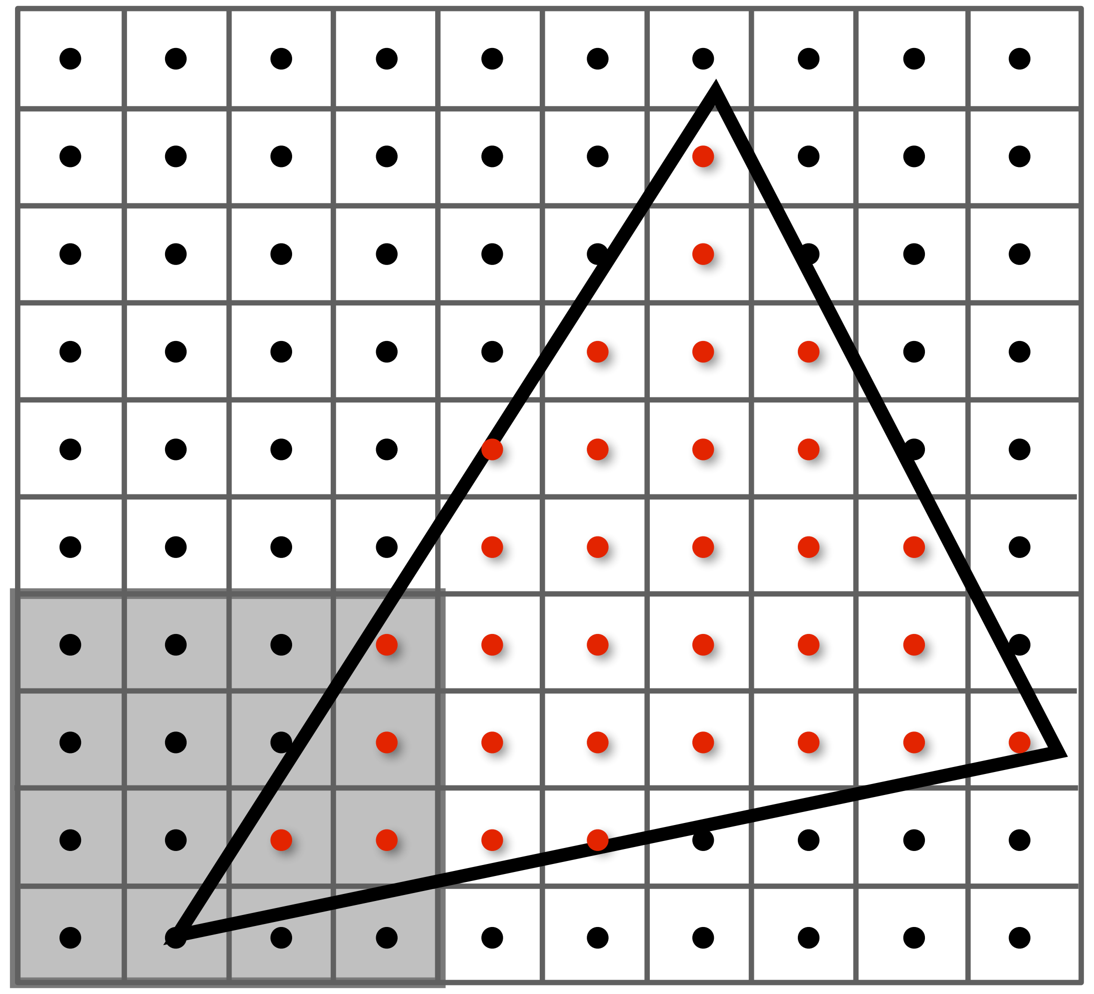
Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point)

Occlusion: depth buffer

```
initialize z_closest[] to INFINITY              // store closest-surface-so-far for all samples
initialize color[]                              // store scene color for all samples
for each triangle t in scene:                   // loop 1: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:       // loop 2: over visibility samples
        if (t_proj covers s)
            Evaluate shader to compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```

"Given a triangle, find the samples it covers"

(finding the samples is relatively easy since they are distributed uniformly on screen)

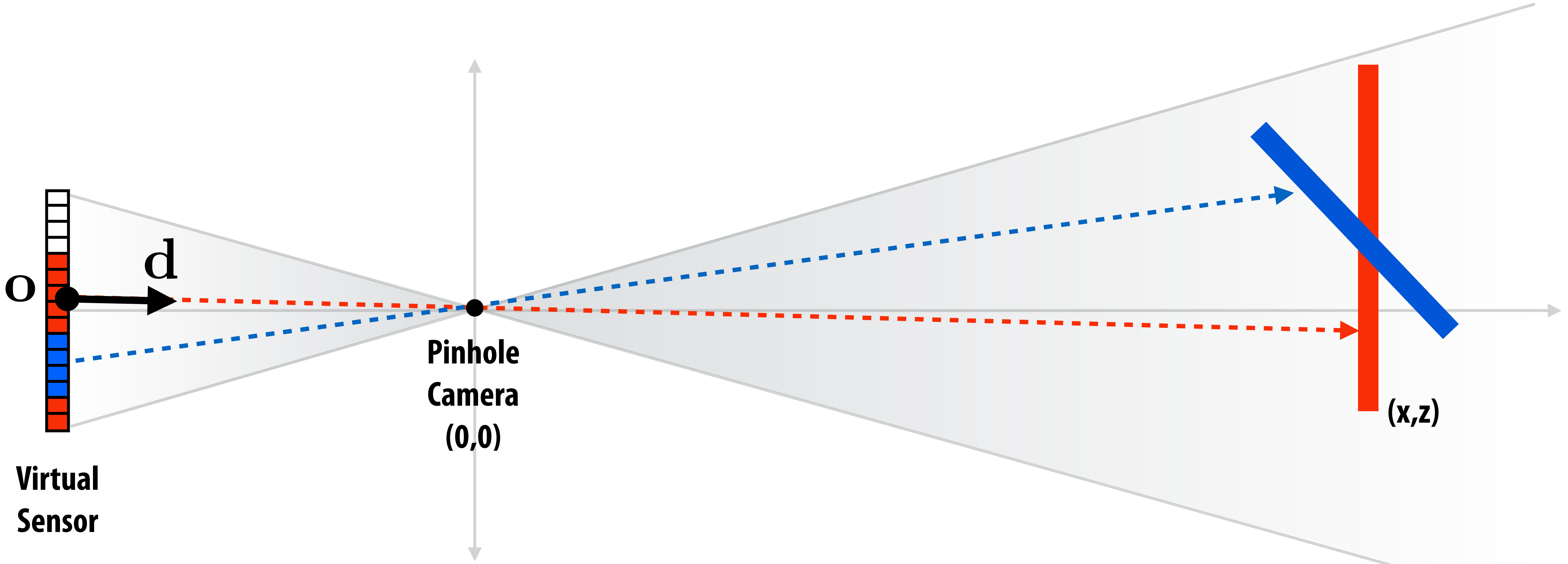


More efficient hierarchical rasterization:

- **For each TILE of image**
  - **If triangle overlaps tile, check all samples in tile**

The visibility problem (as ray tracing)

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



Basic ray casting algorithm

Sample = a ray in 3D

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

Occlusion: closest intersection along ray

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

Compared to rasterization approach: just a reordering of the loops!

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

Basic rasterization vs. ray casting

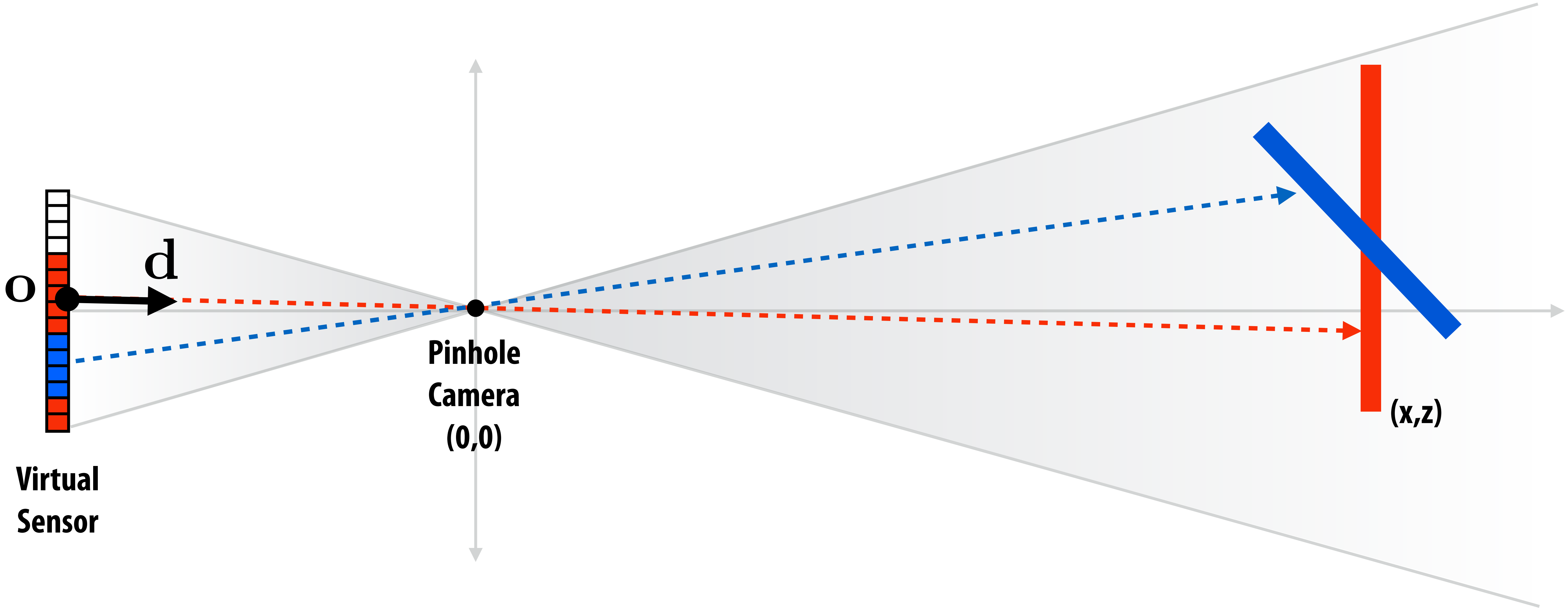
- **Rasterization:**
    - Proceeds in triangle order (for all triangles)
    - Store entire depth buffer (requires access to 2D array of fixed size)
        - Given triangle, “find” samples it covers in 2D buffer
    - Do not have to store entire scene geometry in memory
        - Naturally supports unbounded size scenes
- **Ray casting:**
    - Proceeds in screen sample order (for all rays)
        - Do not have to store closest depth so far for the entire screen (just the current ray)
    - Must store entire scene geometry for fast access (find the hit)
        - Given ray, “find” closest triangle it intersects
        - Challenging, since a ray may go anywhere in the scene

Ray tracing in one class

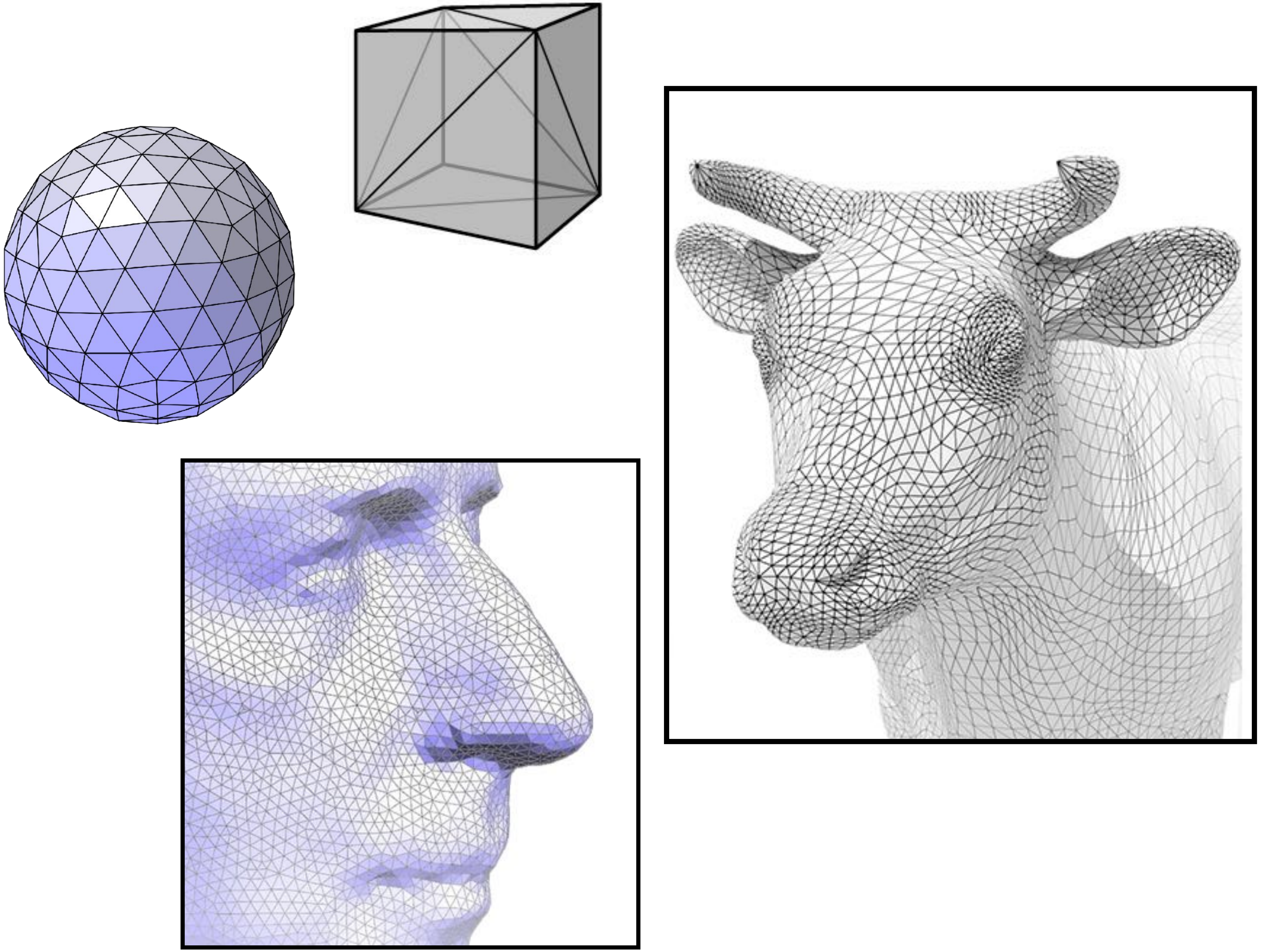
Take that Pete Shirley!

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



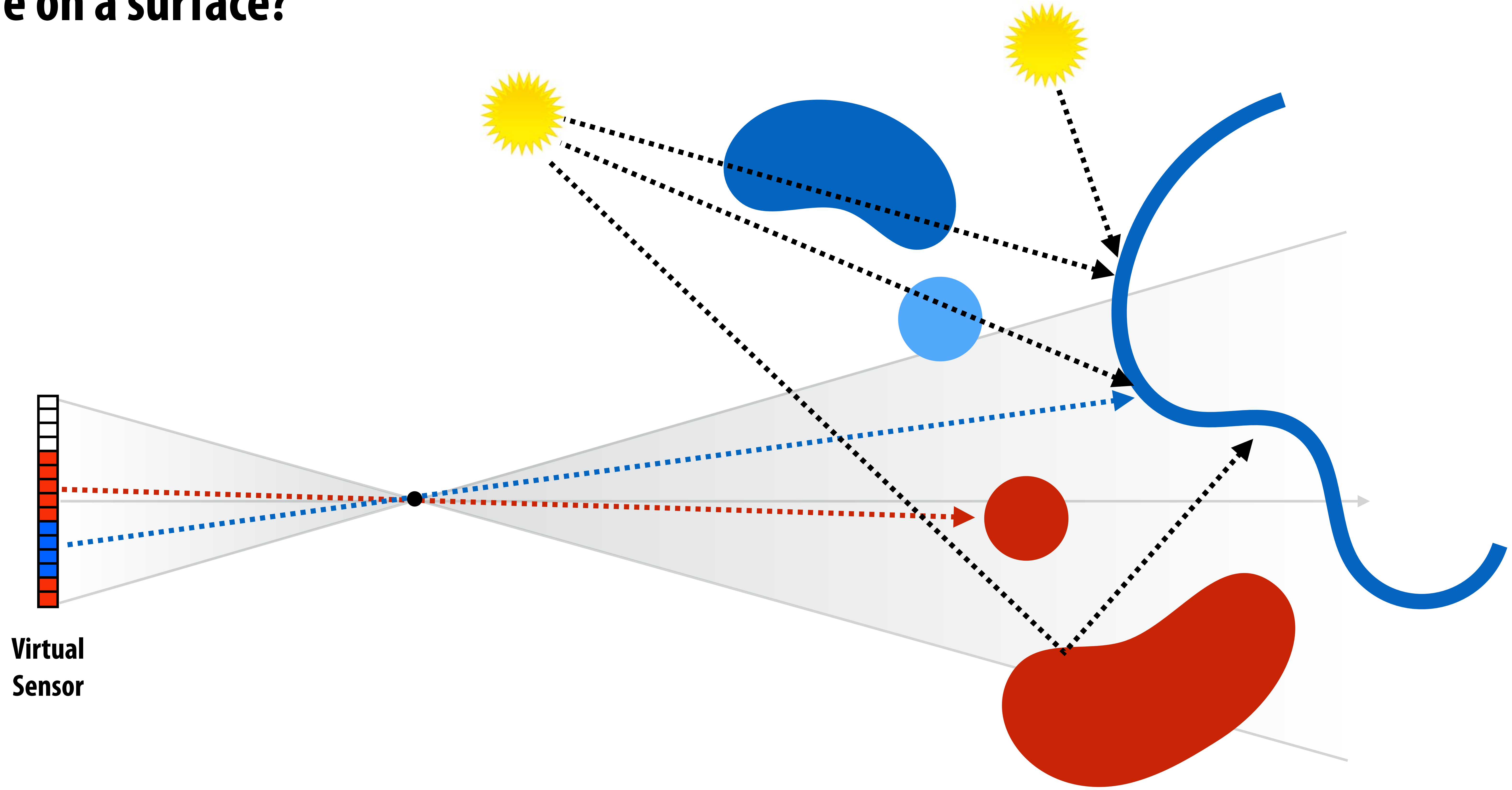
Why do we trace rays?

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?

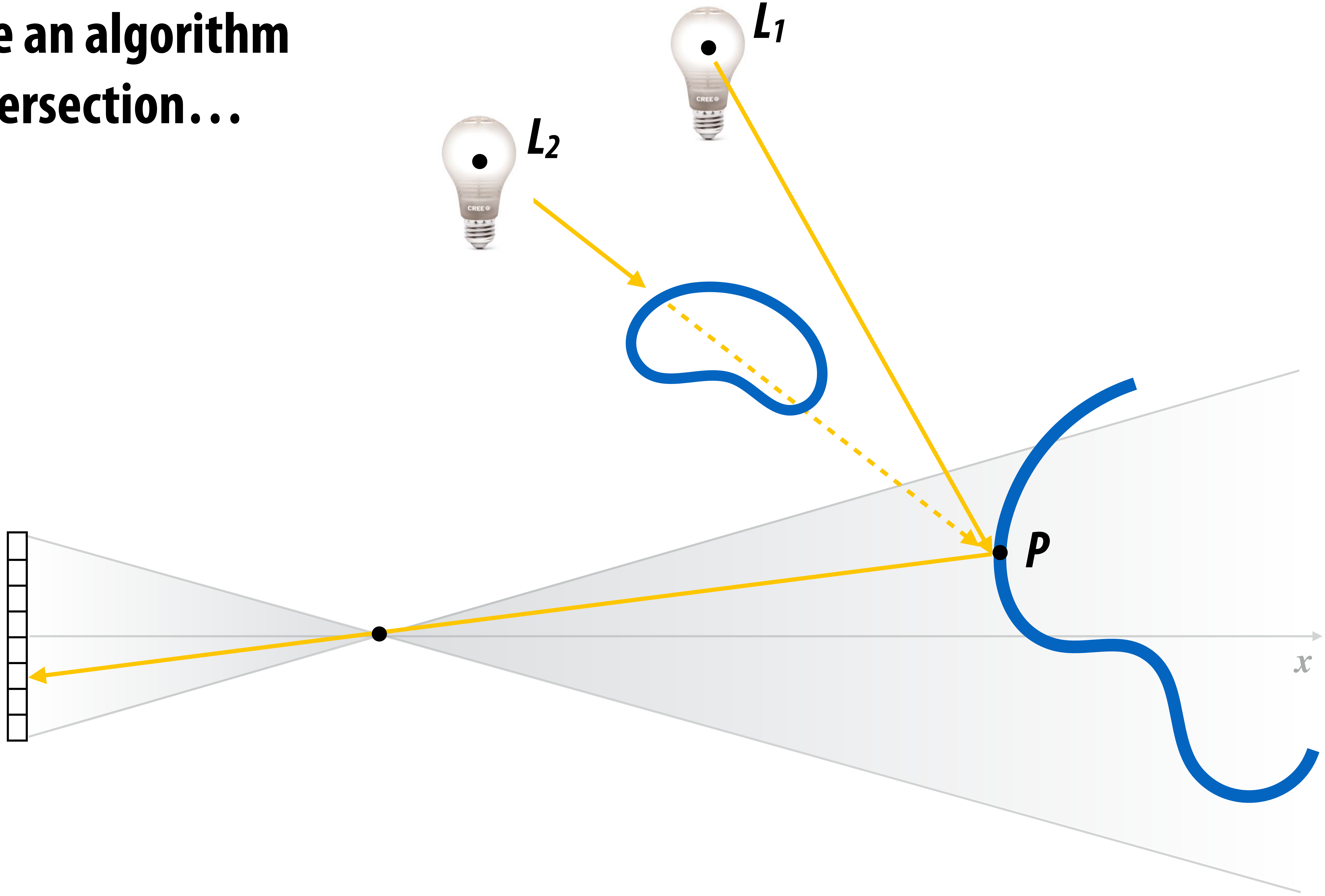


Shadows

Image credit: Grand Theft Auto V

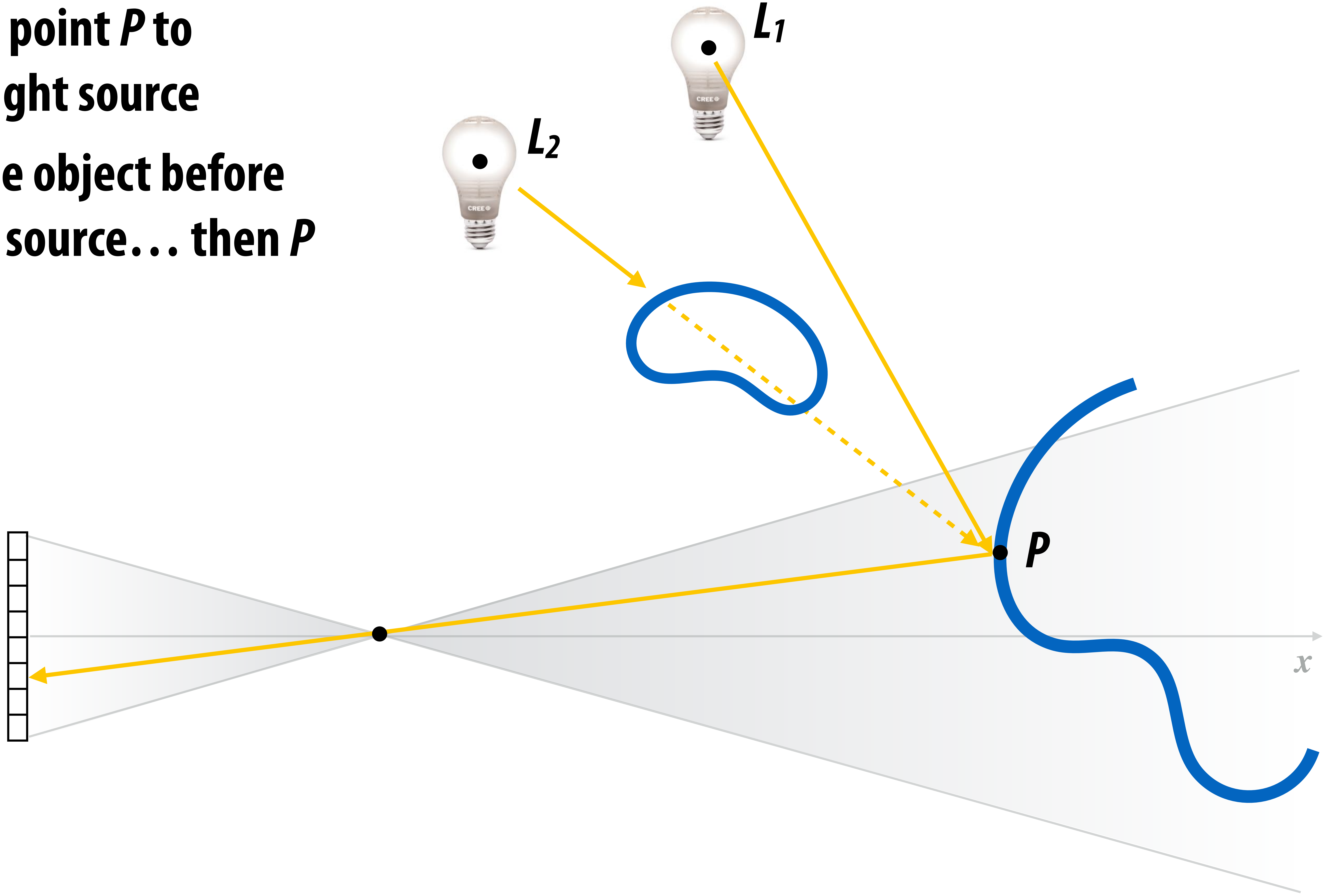
How to compute if a surface point is in shadow?

Assume you have an algorithm for ray-scene intersection…



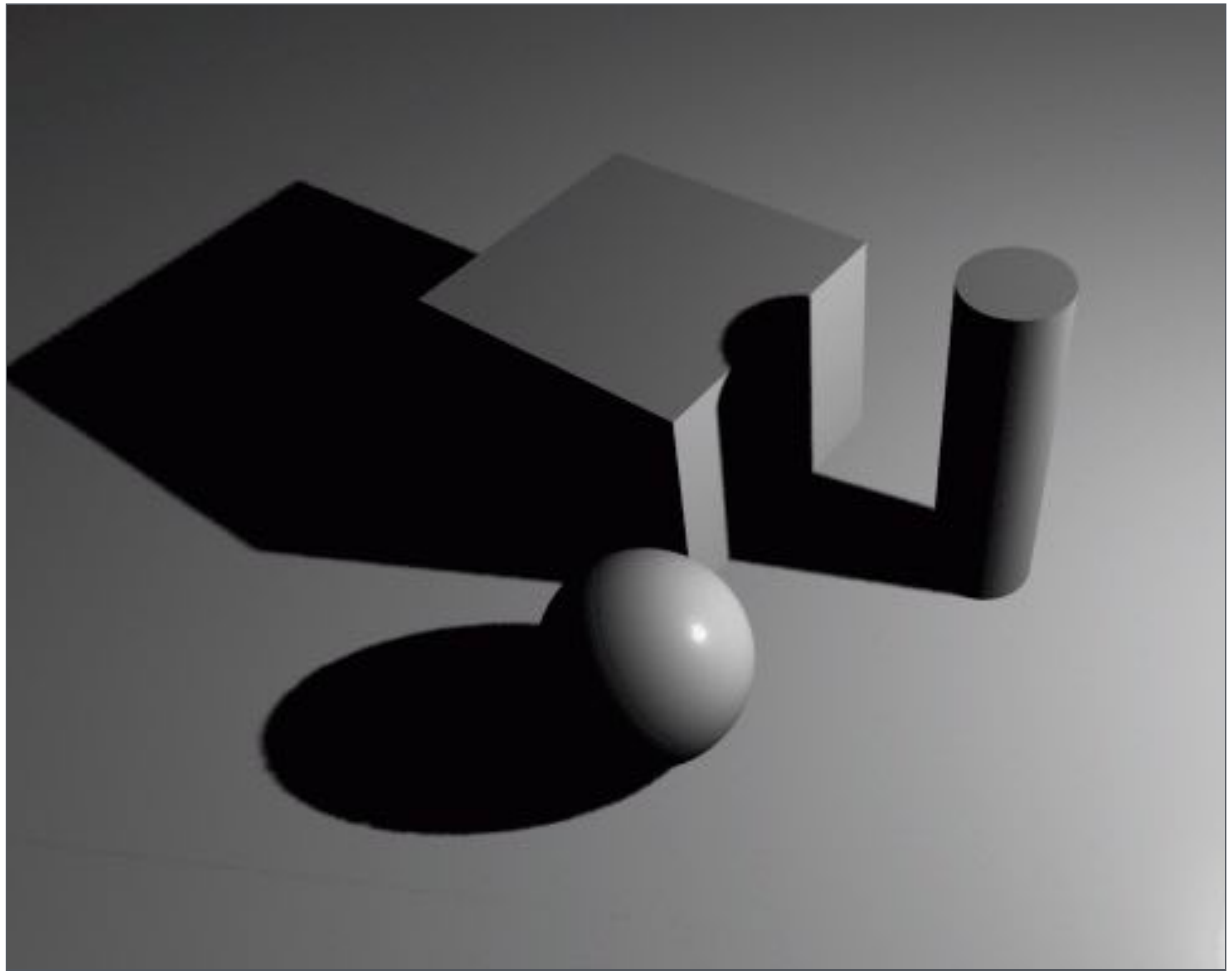
A simple shadow computation algorithm

- 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

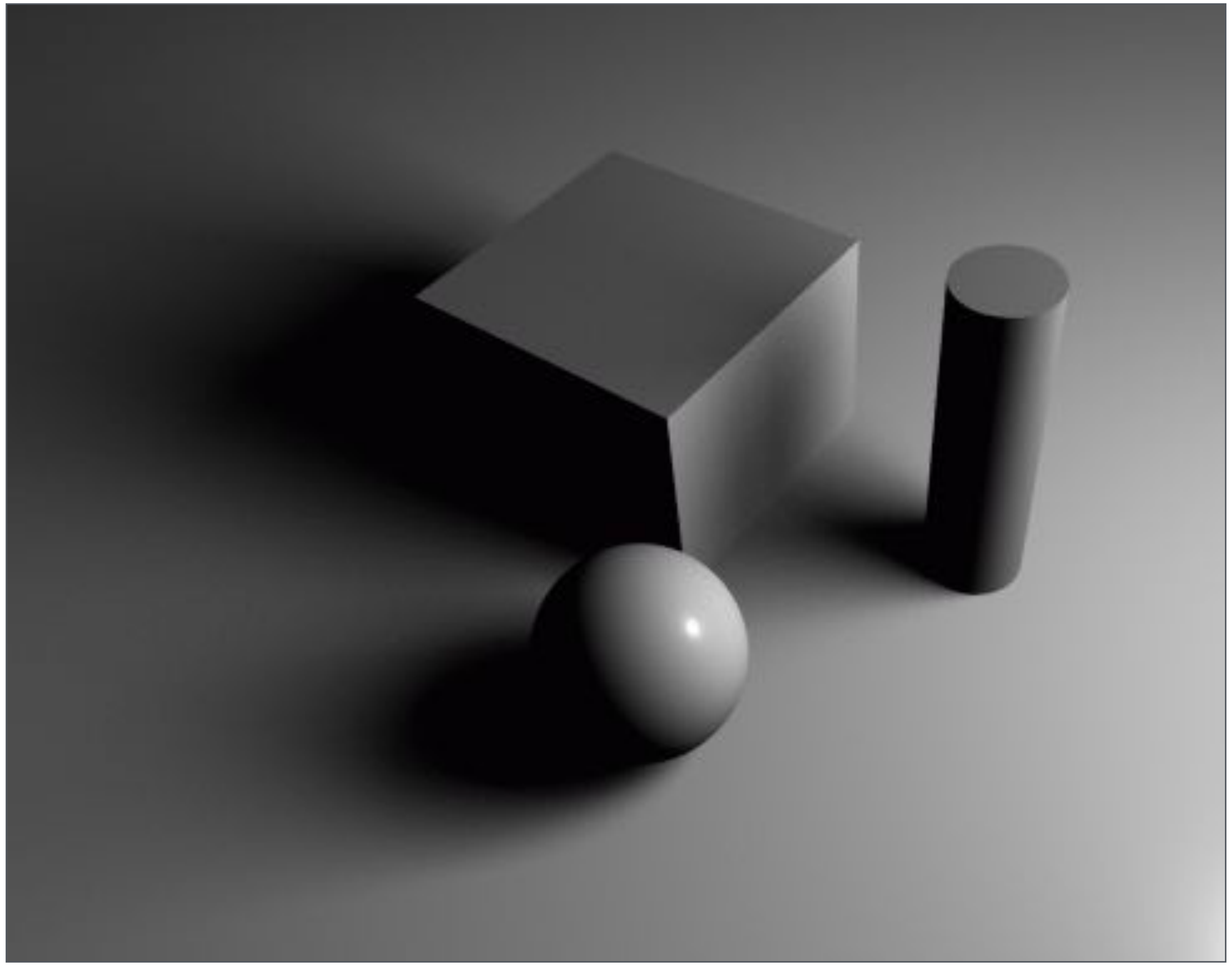


Scene with many light sources

Soft shadows



Hard shadows
(created by point light source)

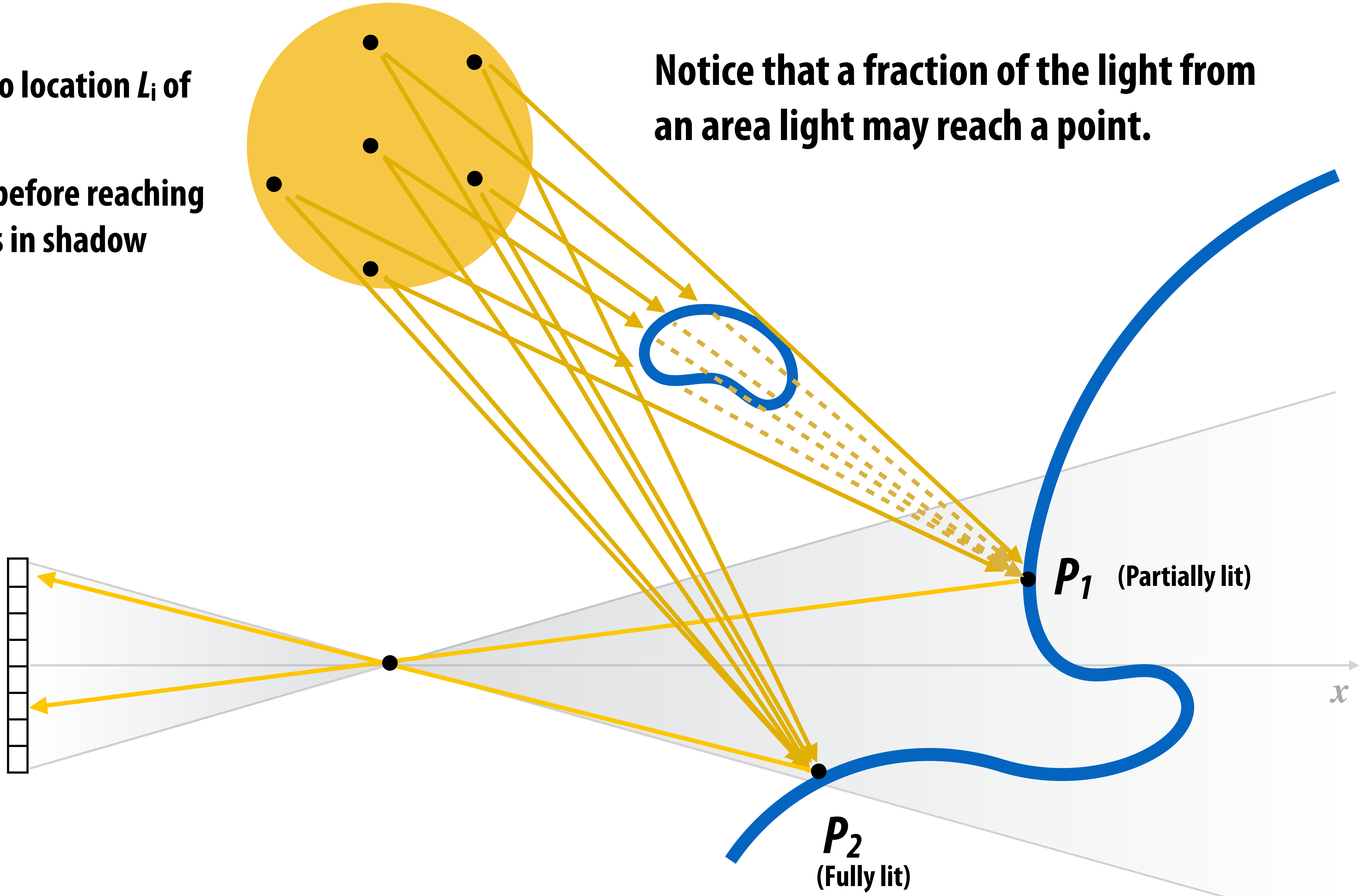


Soft shadows
(created by ???)

Image credit: Pixar

Shadow cast by an area light

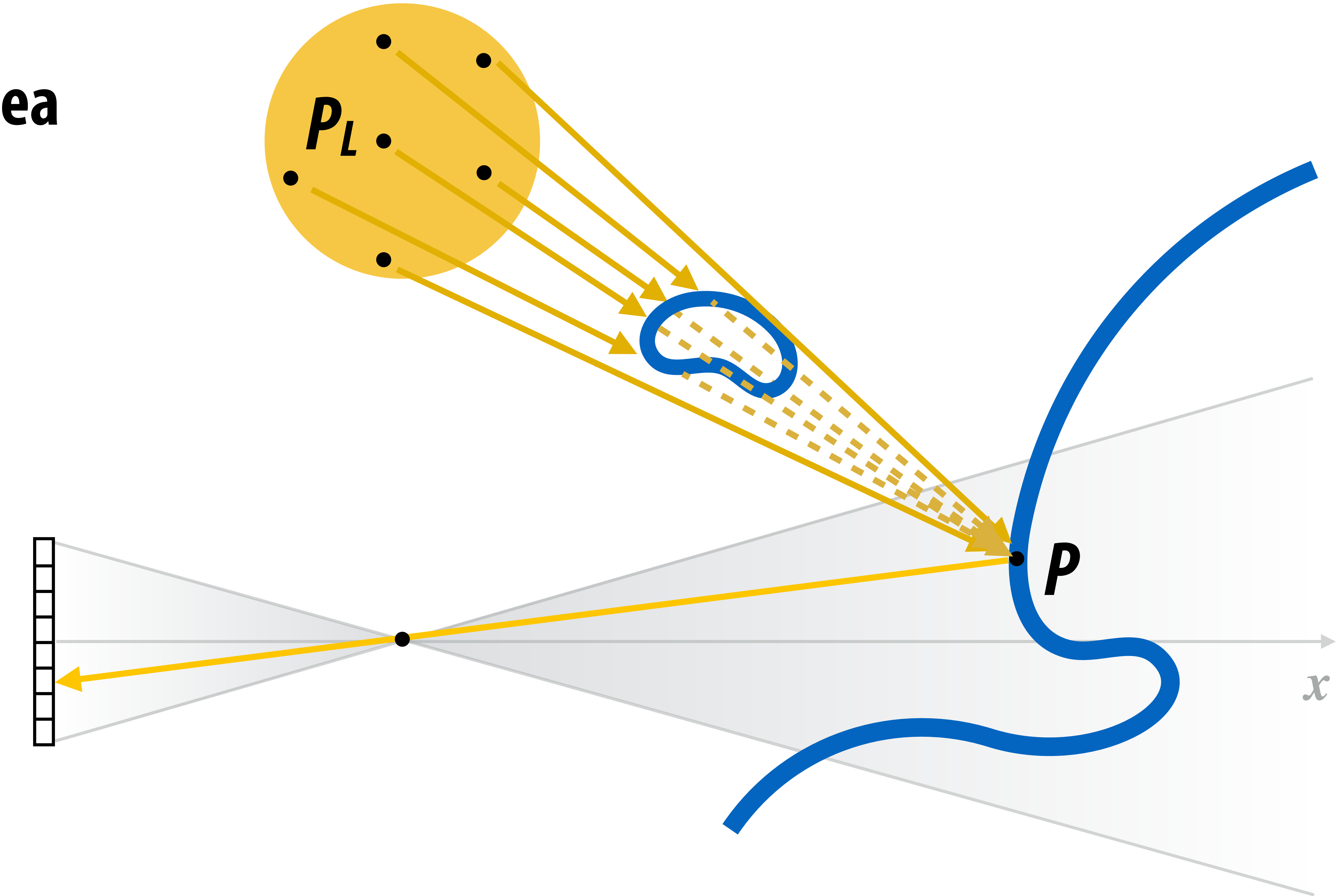
- Based on ray tracing…
- Trace ray from point *P* to location L_i of light source
- If ray hits scene object before reaching light source… then *P* is in shadow



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

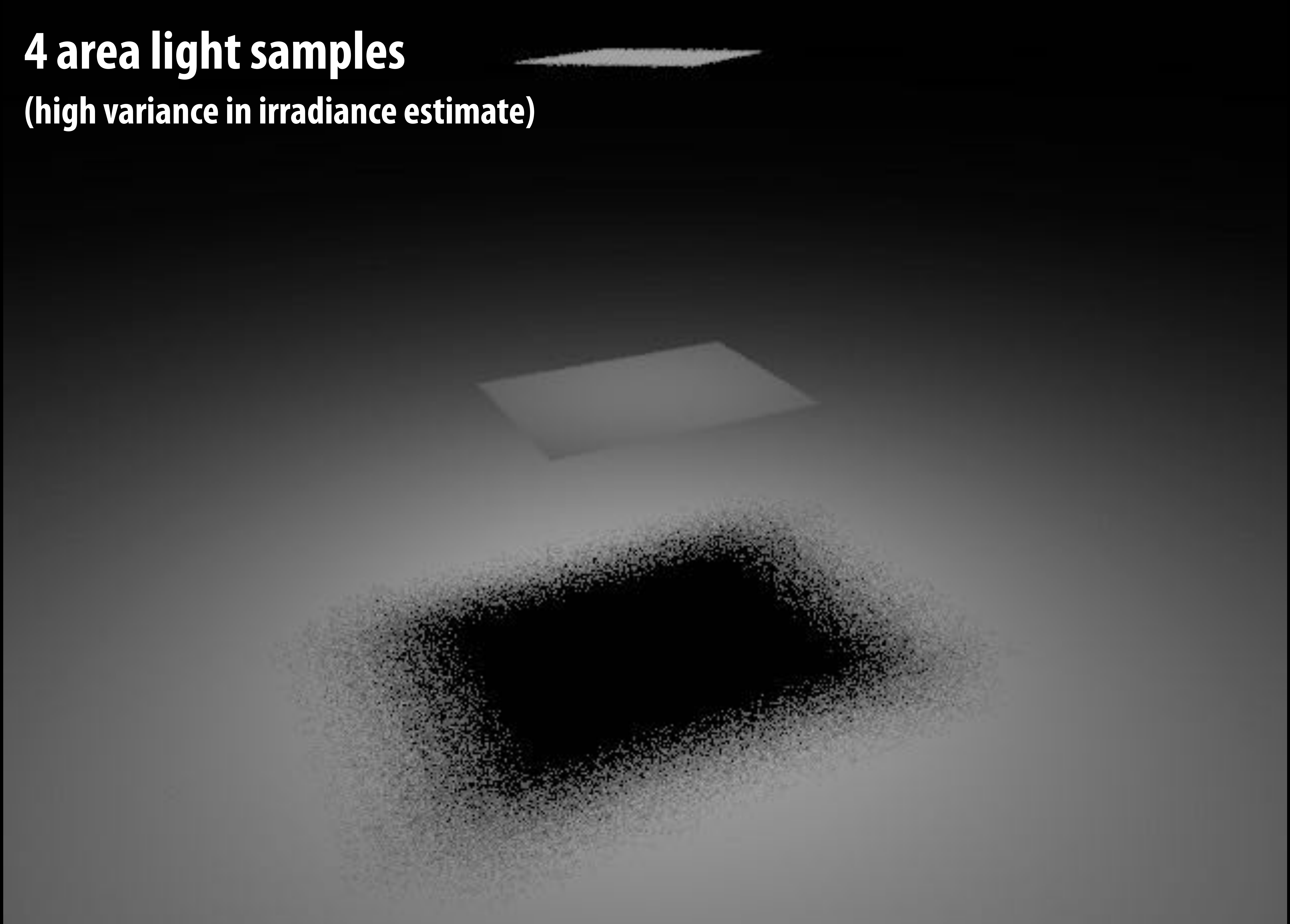
Sampling based algorithm

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!

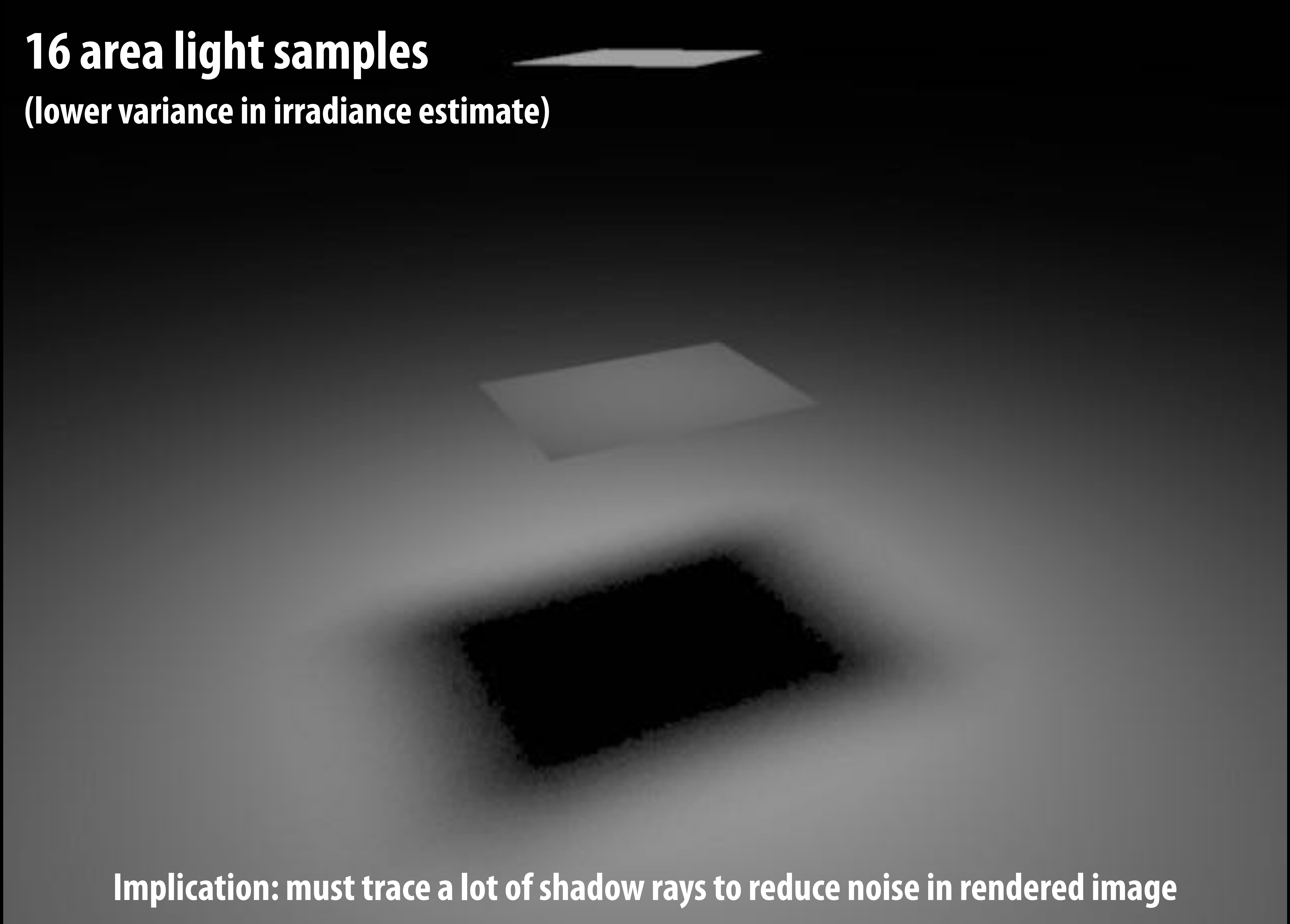


4 area light samples

(high variance in irradiance estimate)

16 area light samples

(lower variance in irradiance estimate)



Implication: must trace a lot of shadow rays to reduce noise in rendered image

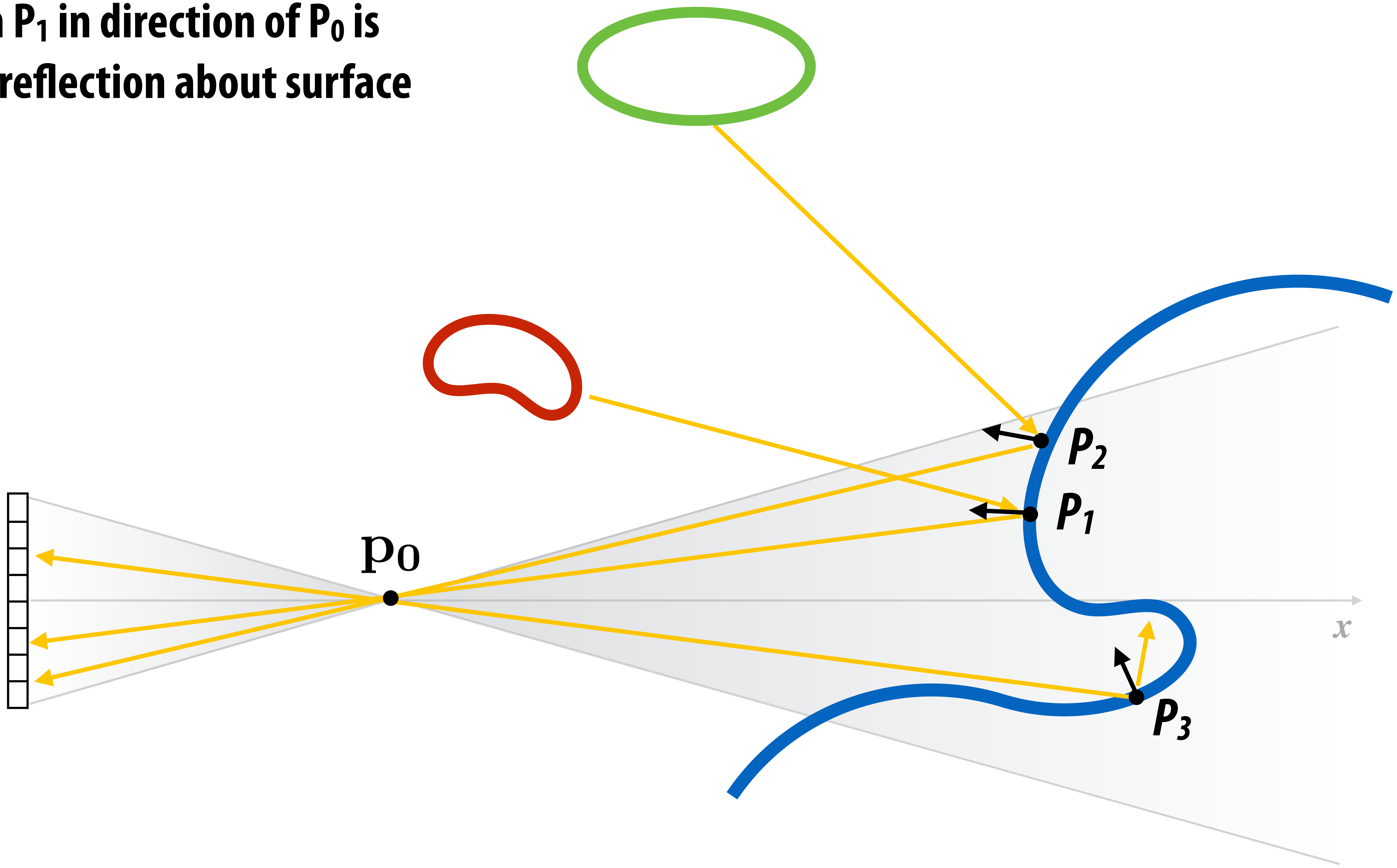
Reflections

Image credit: NVIDIA

Reflections

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

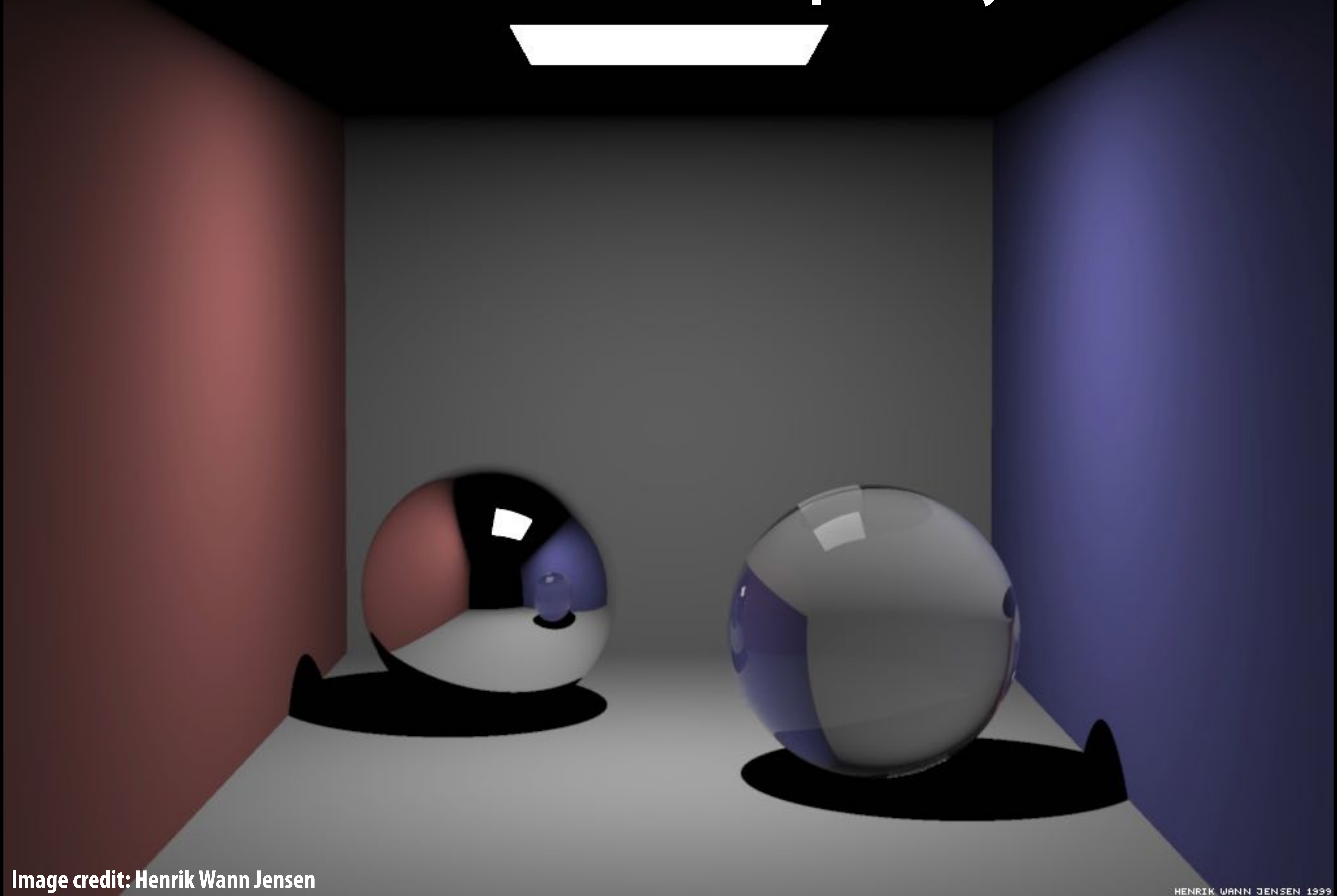


Image credit: Henrik Wann Jensen

Global illumination solution

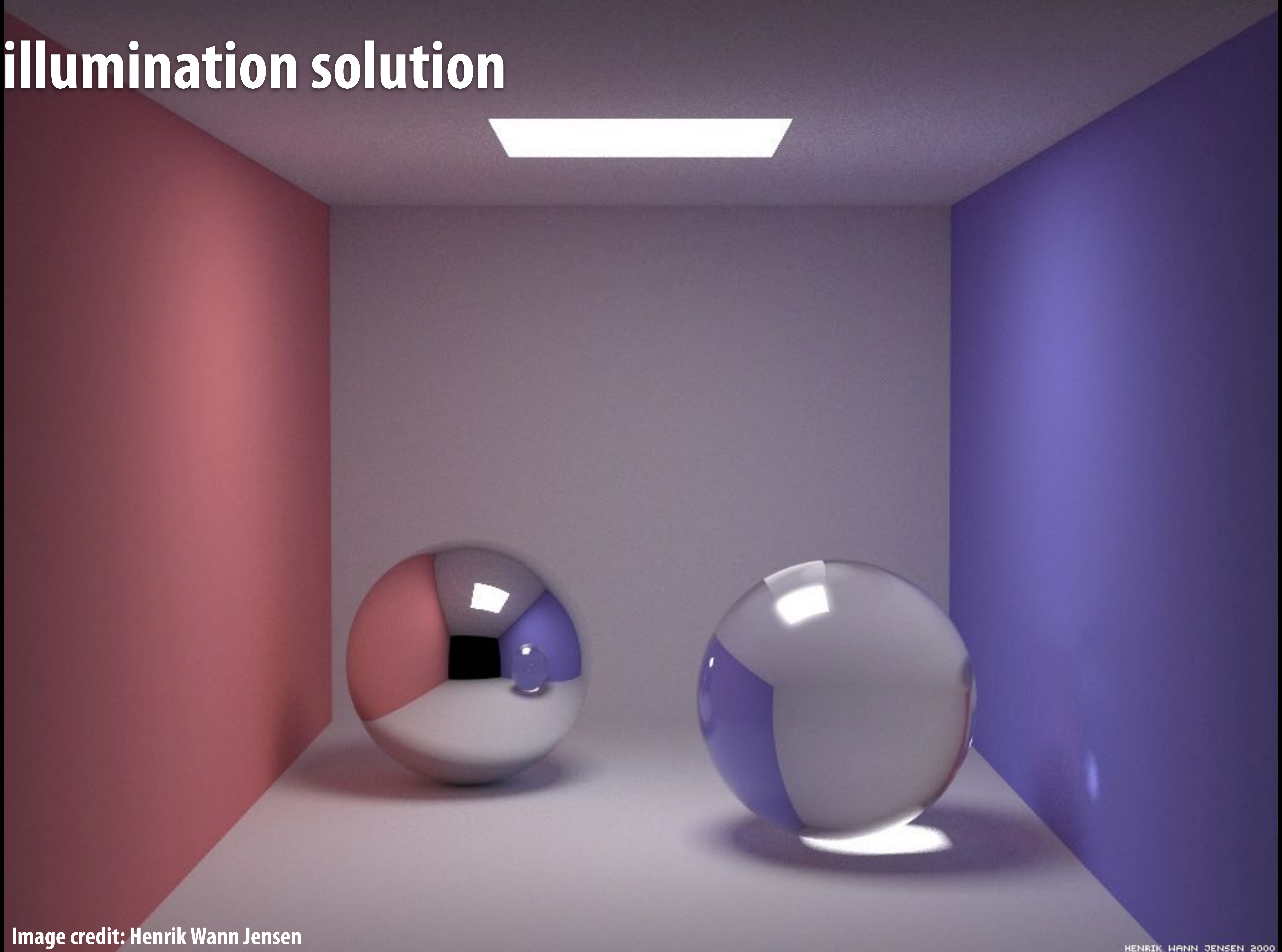
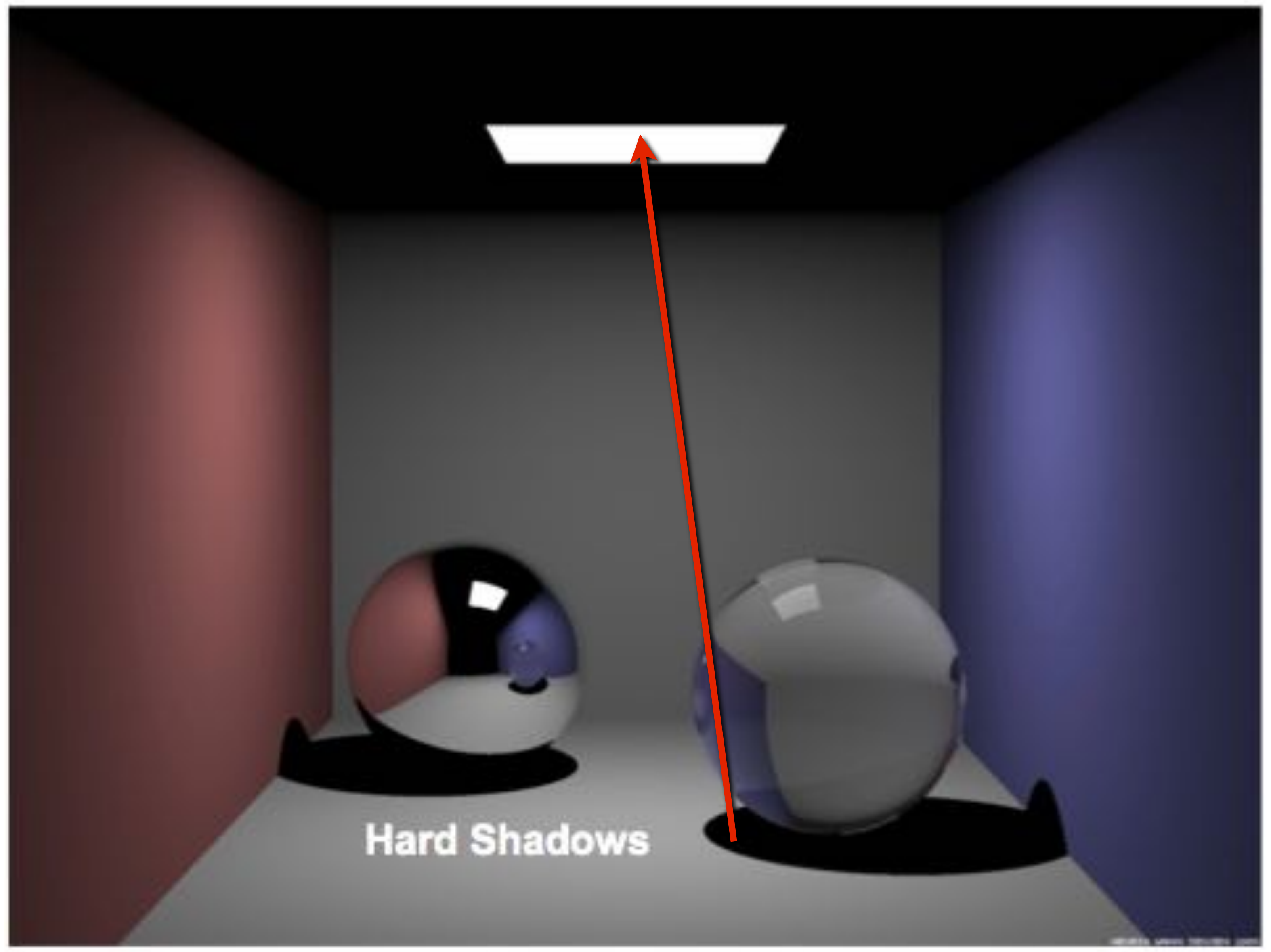
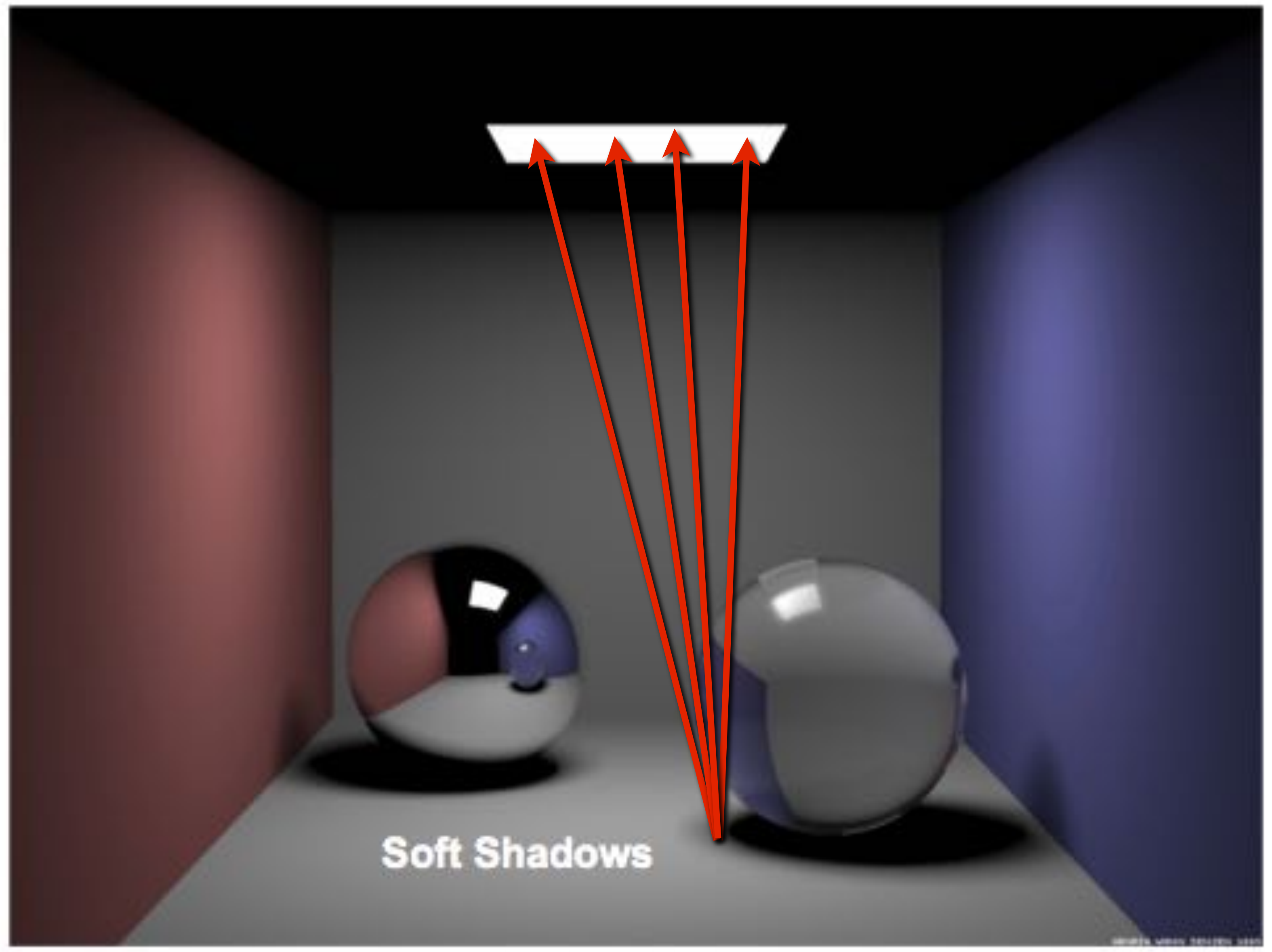
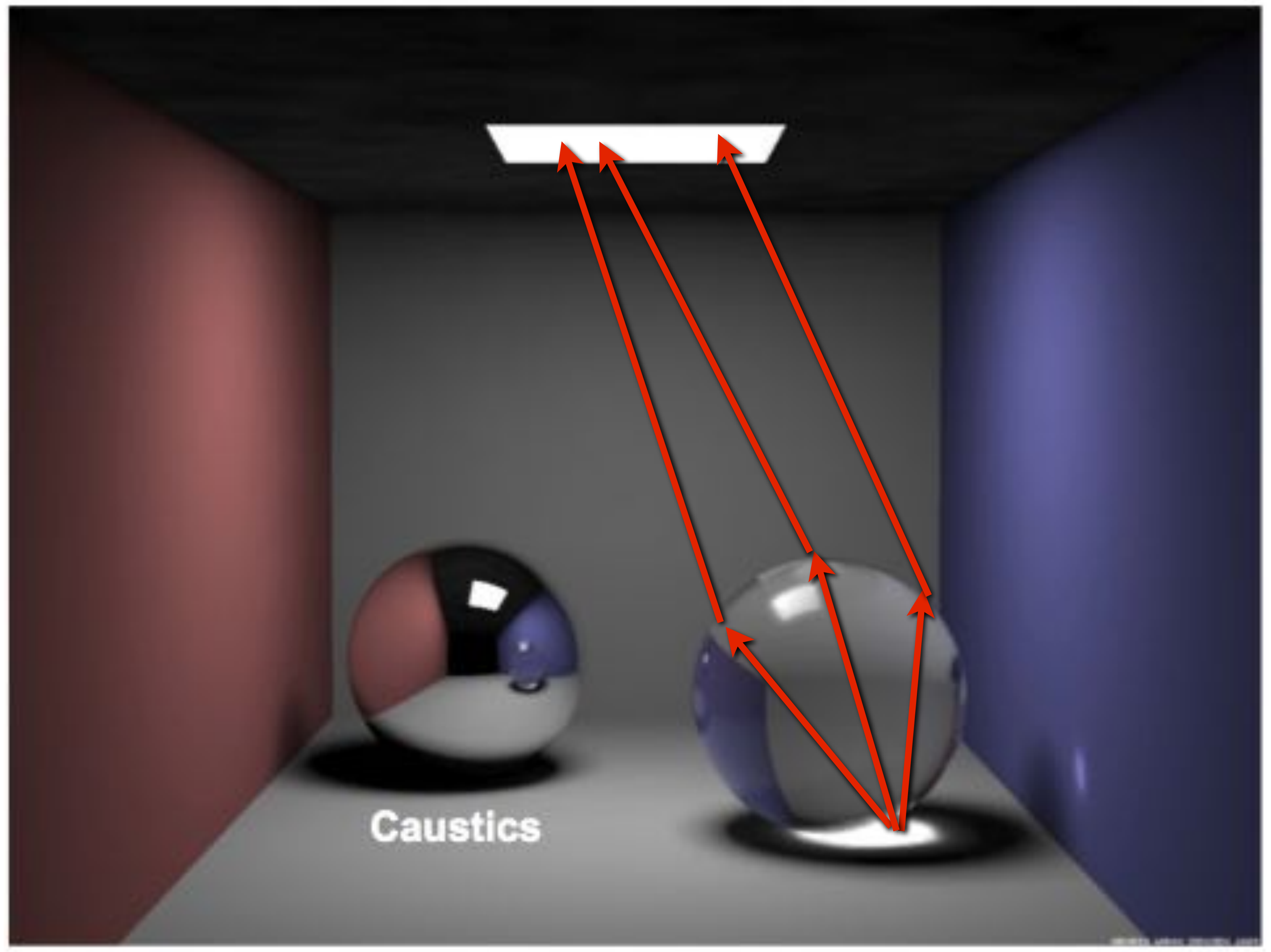


Image credit: Henrik Wann Jensen

Sampling light paths







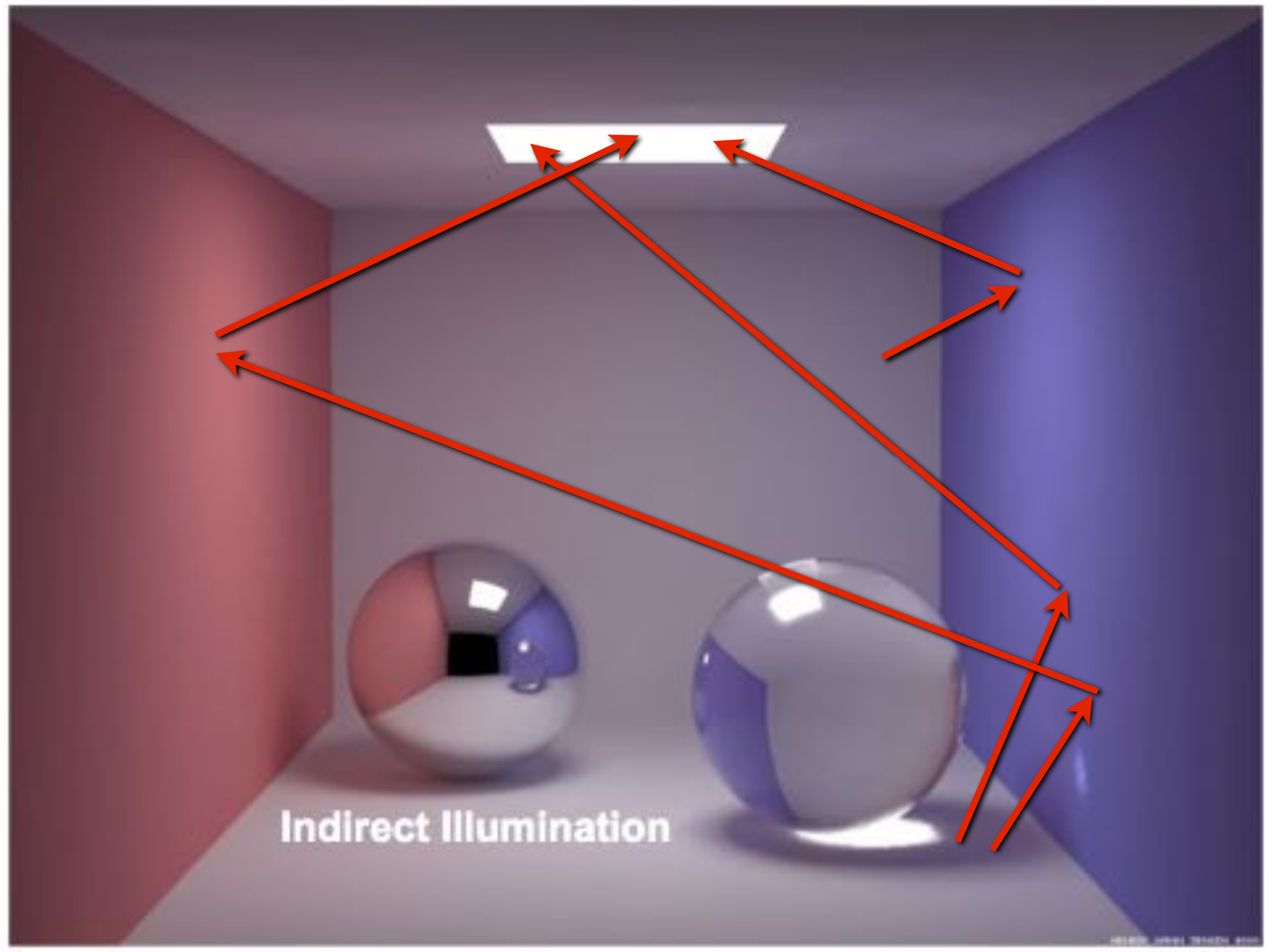
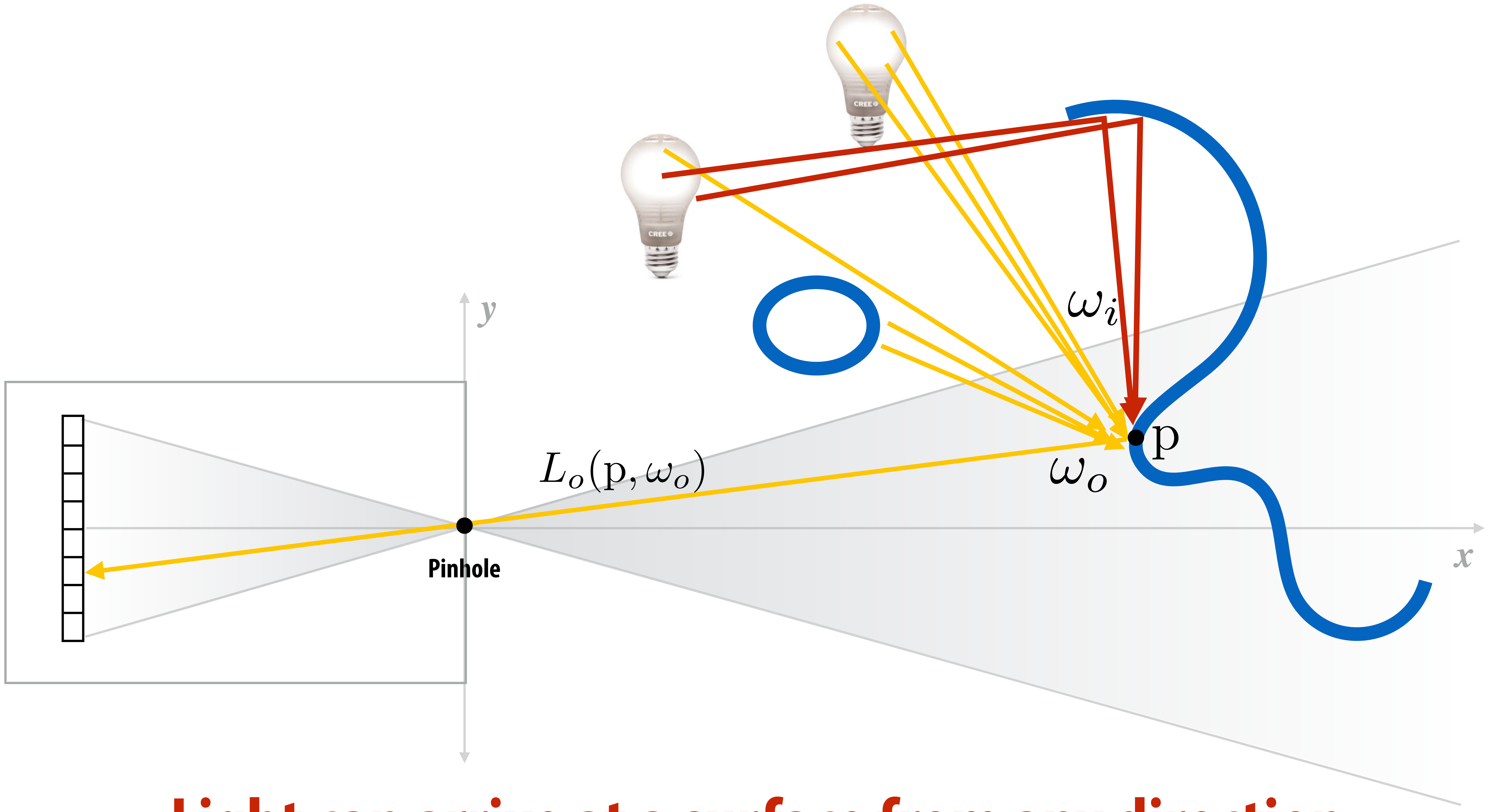


Image credit: Wann Jensen, Hanrahan

Indirect illumination



Light can arrive at a surface from any direction.

Implication: even more ray tracing per pixel!

Direct illumination

One-bounce global illumination

Sixteen-bounce global illumination

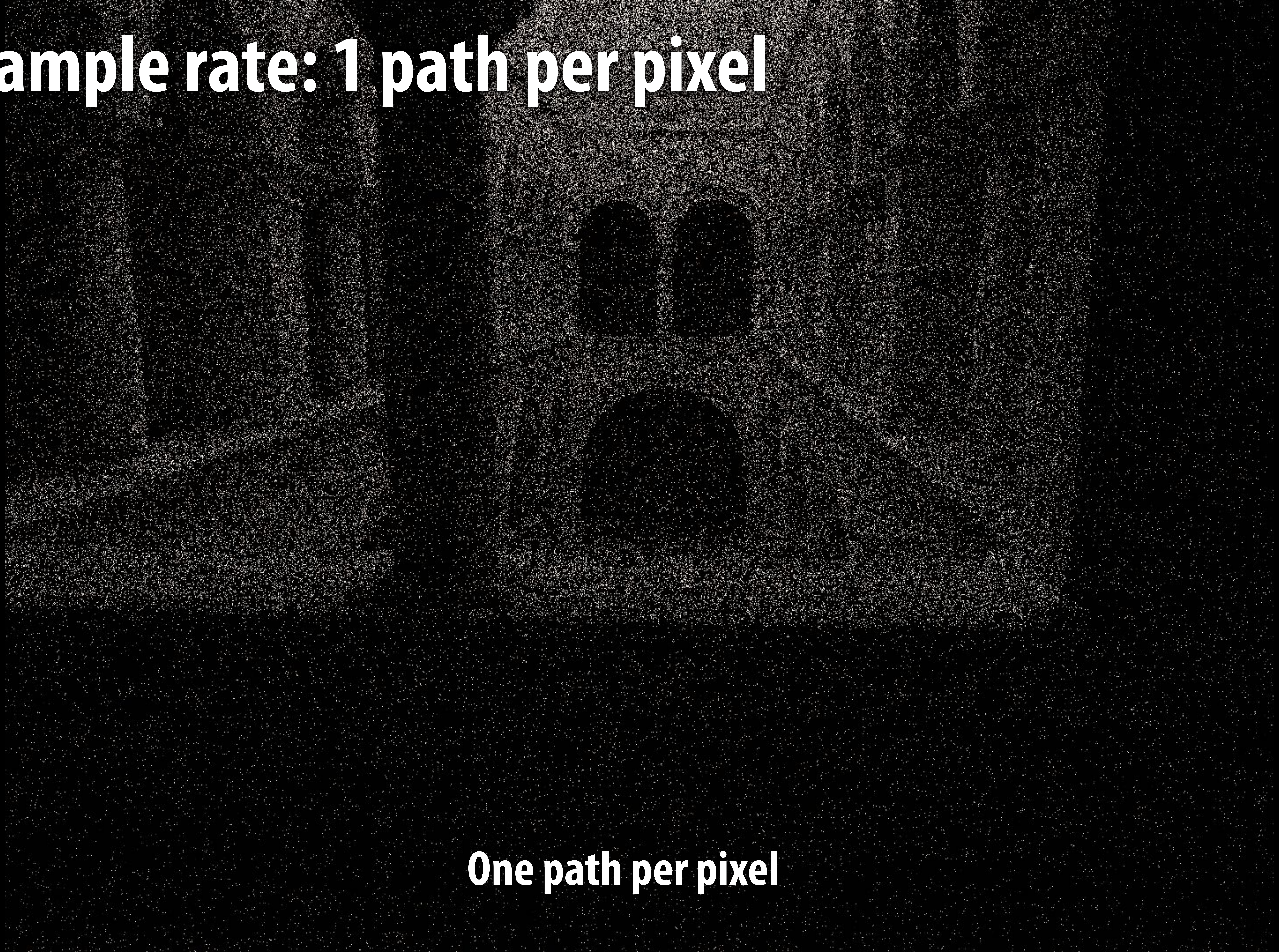
Direct illumination

Global Illumination

Importance of indirect illumination

1024 samples per pixel

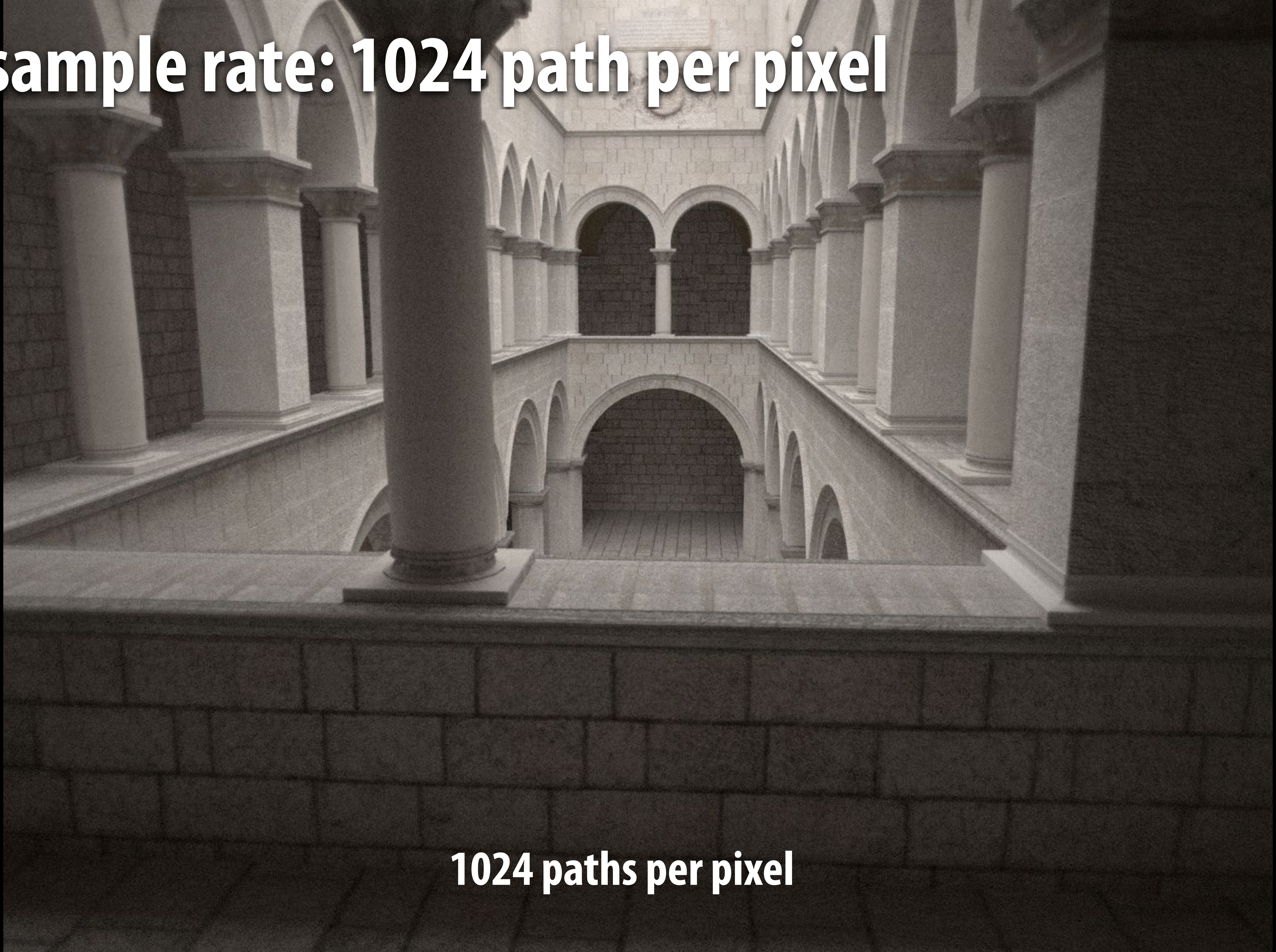
Low sample rate: 1 path per pixel



One path per pixel

32 paths per pixel

High sample rate: 1024 path per pixel



1024 paths per pixel

Takeaway:
Must trace many rays per pixel through complex scenes to render realistic images in real time

Ray-scene intersection preliminaries:

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

Ray equation

- Recall, can express ray as:

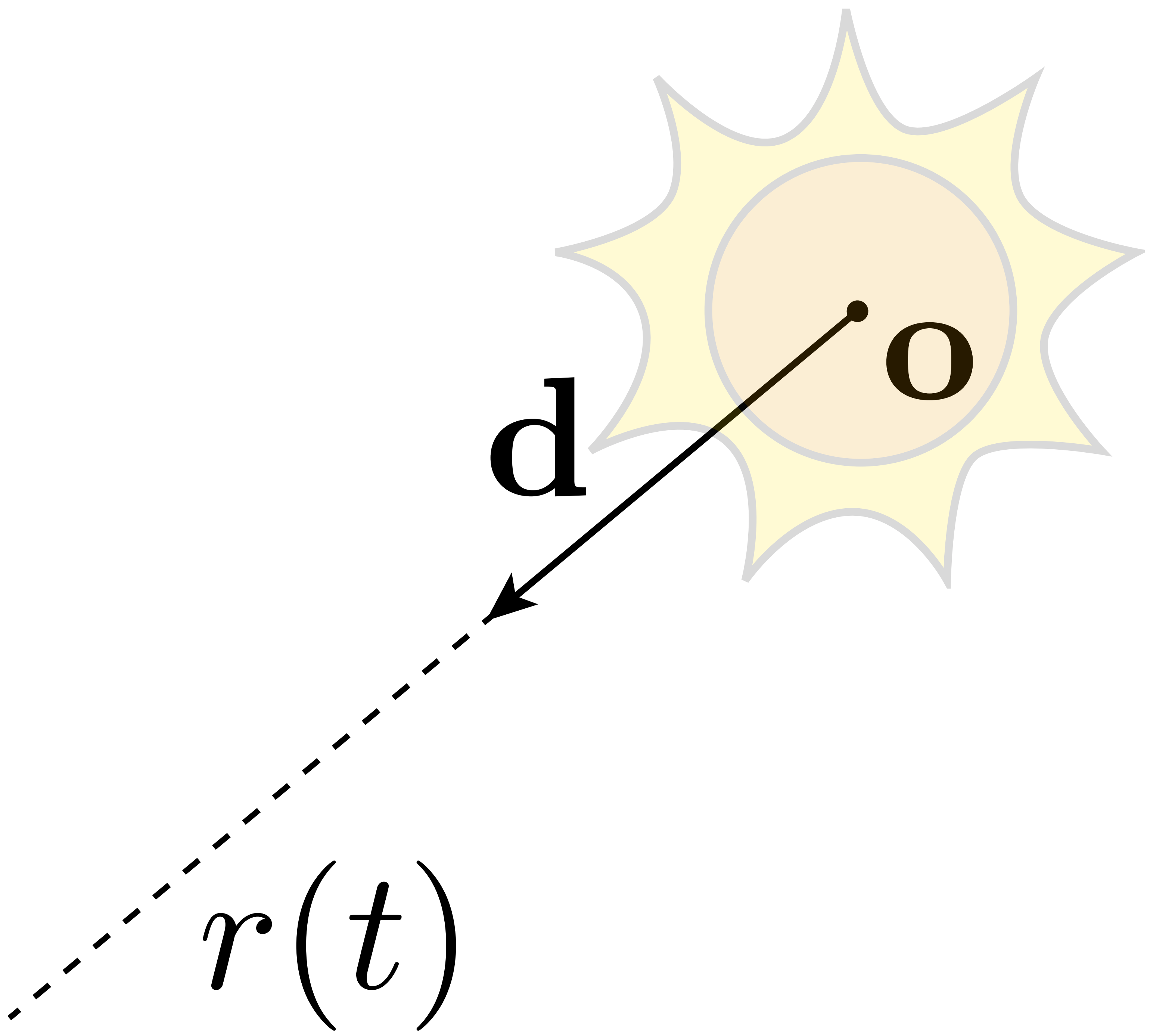
origin

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:

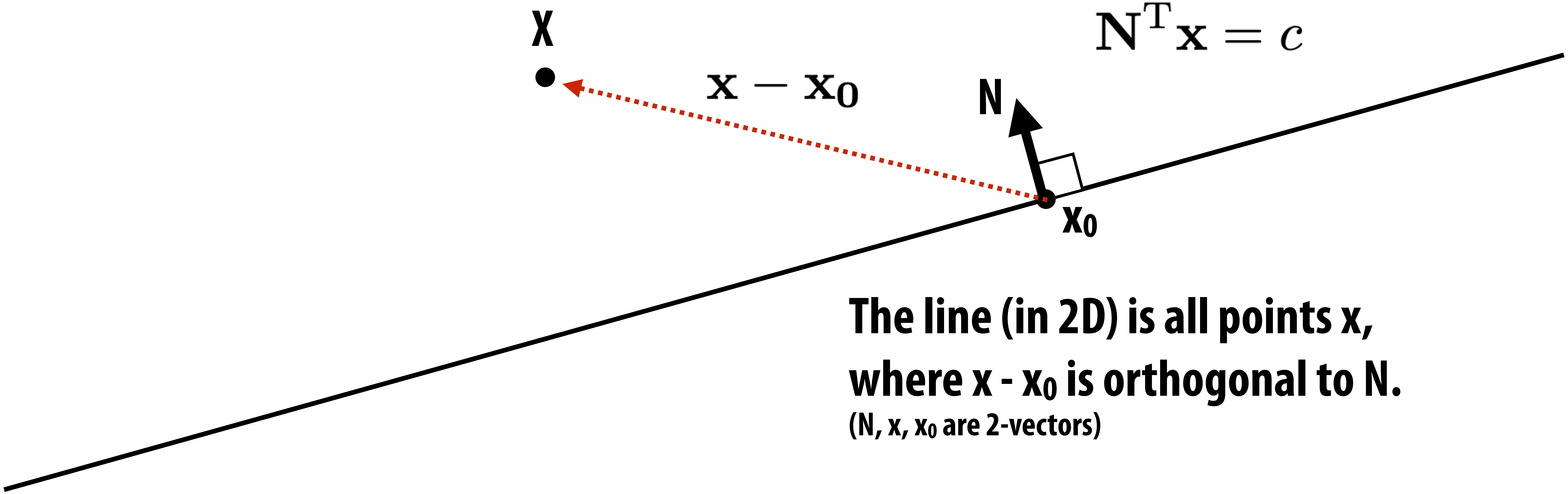
- Its normal: N
- A point x_0 on the line

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

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

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

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



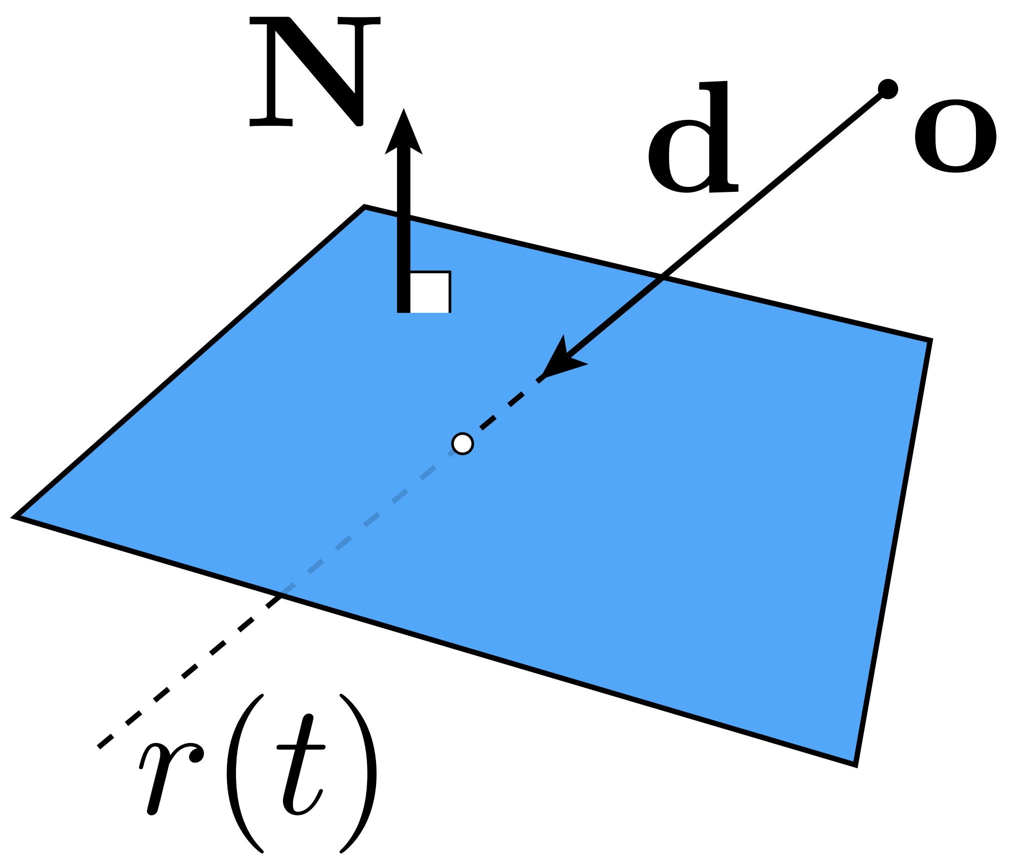
The line (in 2D) is all points x, where x - x_0 is orthogonal to N.

(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 $N^T x = c$**
  - **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:**

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

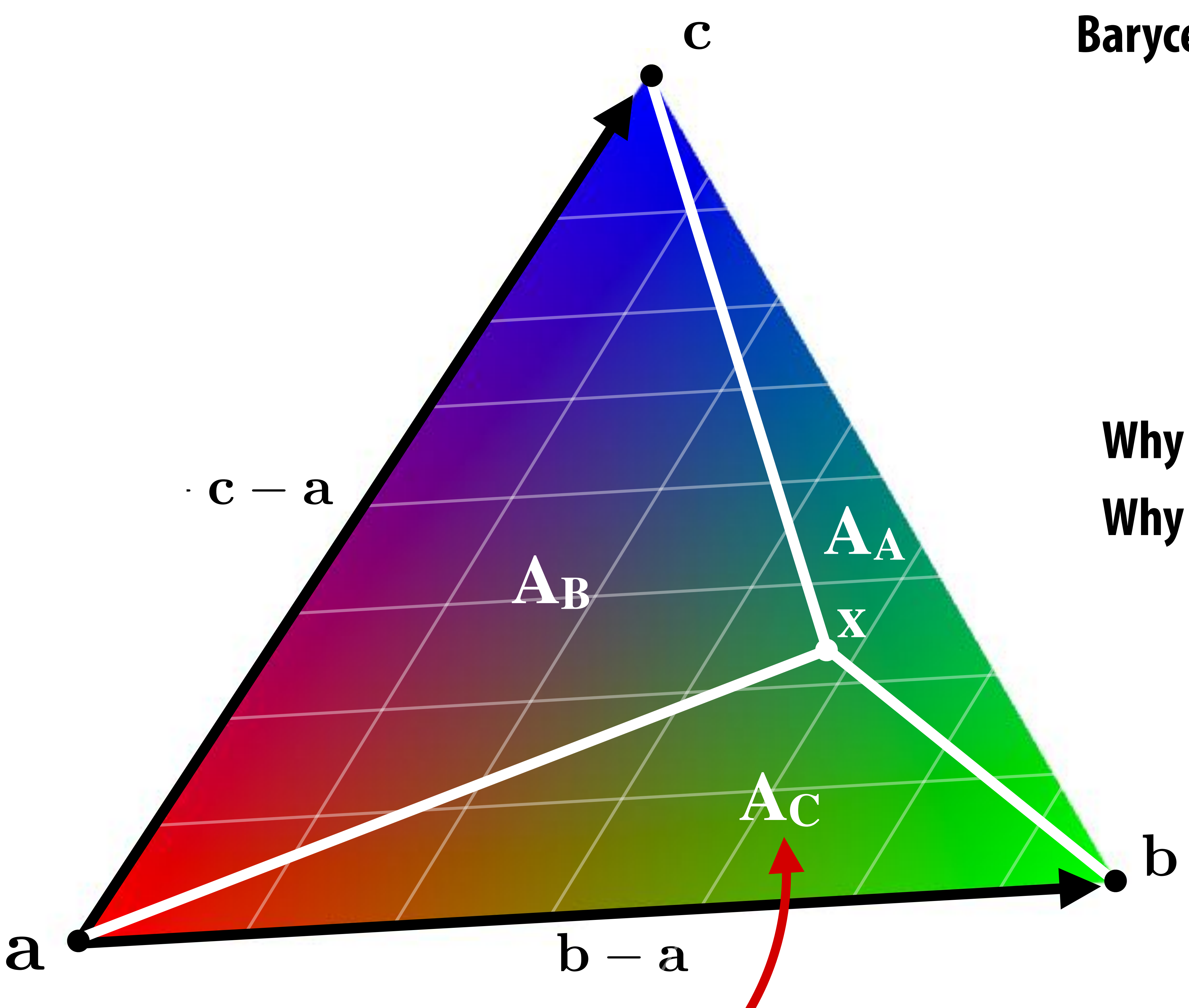
- **Now solve for t:**

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

- **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:

$$\alpha = A_A/A$$

$$\beta = A_B/A$$

$$\gamma = A_C/A$$

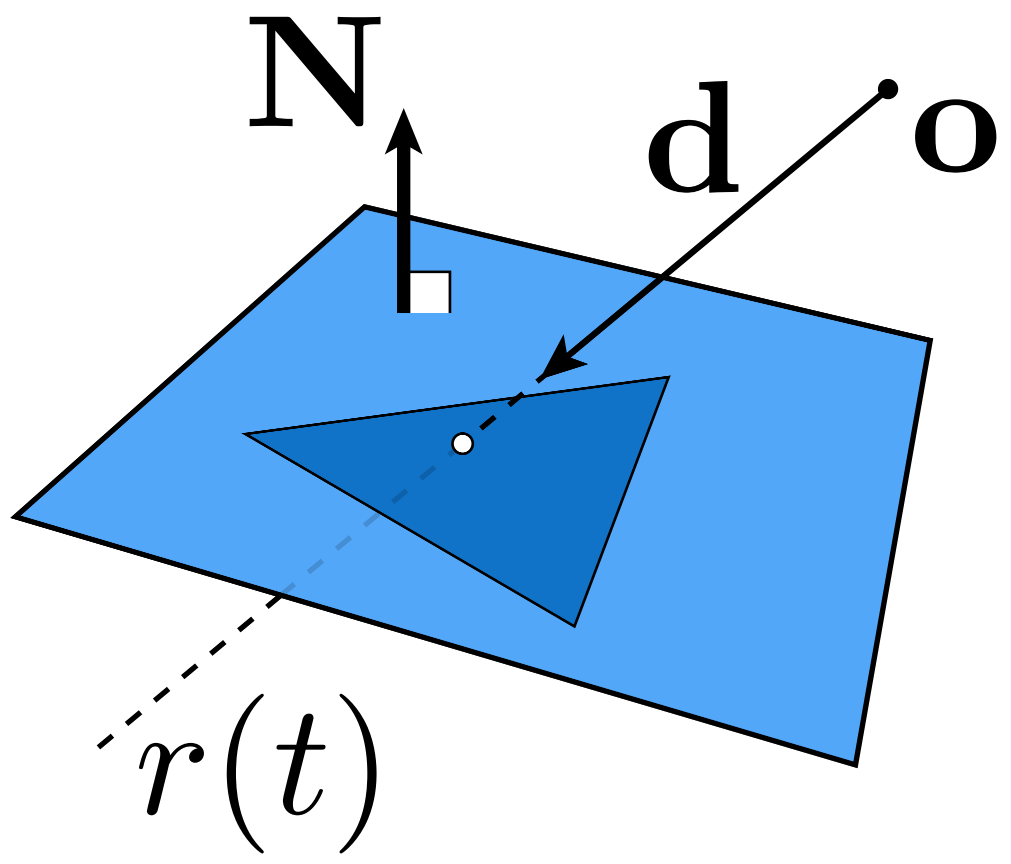
Why must coordinates sum to one?

Why must coordinates be between 0 and 1?

Useful: Heron's formula:

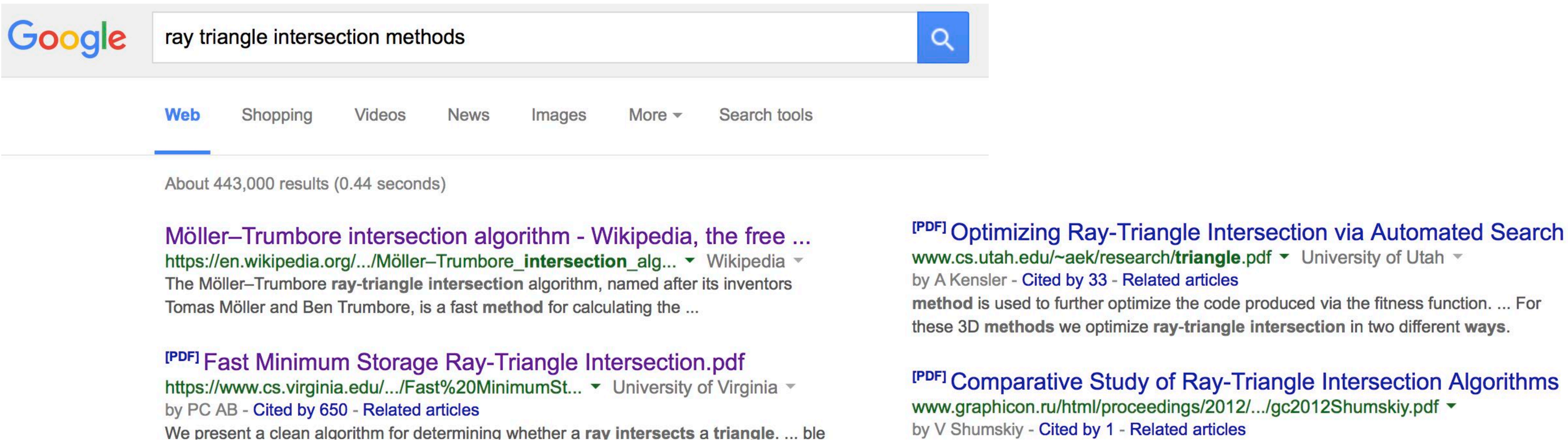
$$A_C = \frac{1}{2}(\mathbf{b}-\mathbf{a}) \times (\mathbf{x}-\mathbf{a})$$

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



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**.

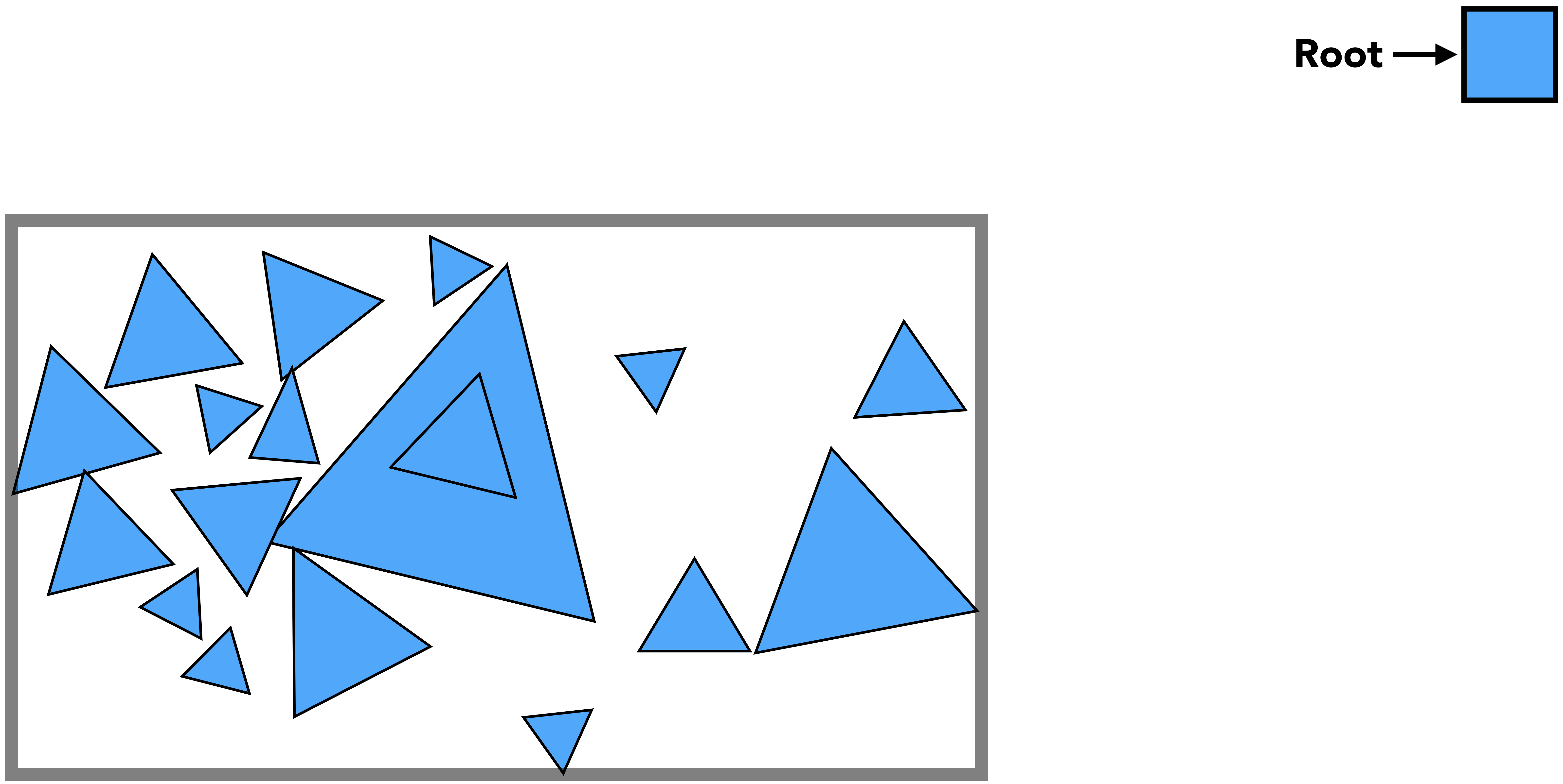
[PDF] Comparative Study of Ray-Triangle Intersection Algorithms
www.graphicon.ru/html/proceedings/2012/.../gc2012Shumskiy.pdf
by V Shumskiy - Cited by 1 - Related articles

Takeaway:
Ray-triangle intersection is an arithmetically rich operation

Ray-scene intersection preliminaries:

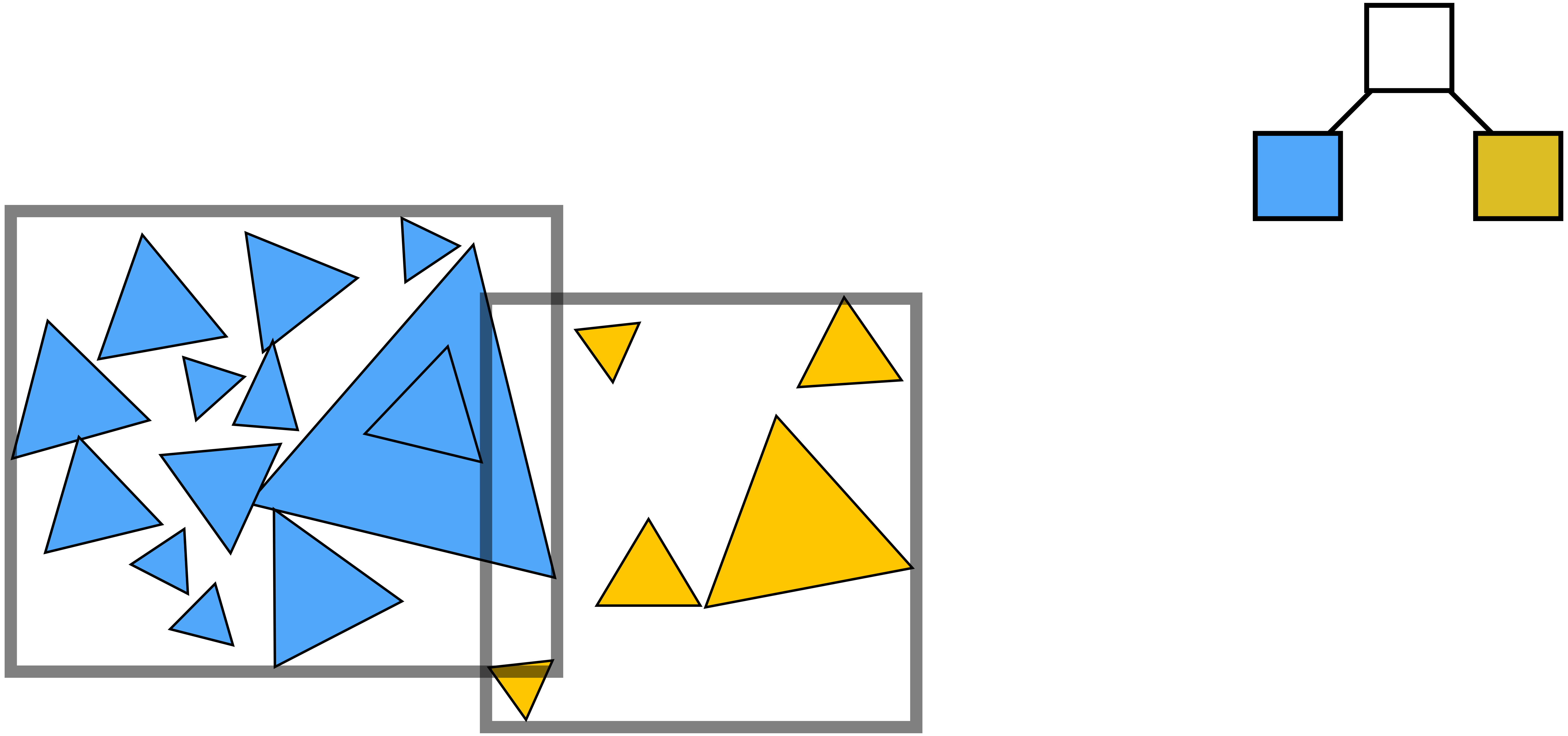
How to efficiently find the closest hit using BVH acceleration structures ("indices")

Bounding volume hierarchy (BVH)

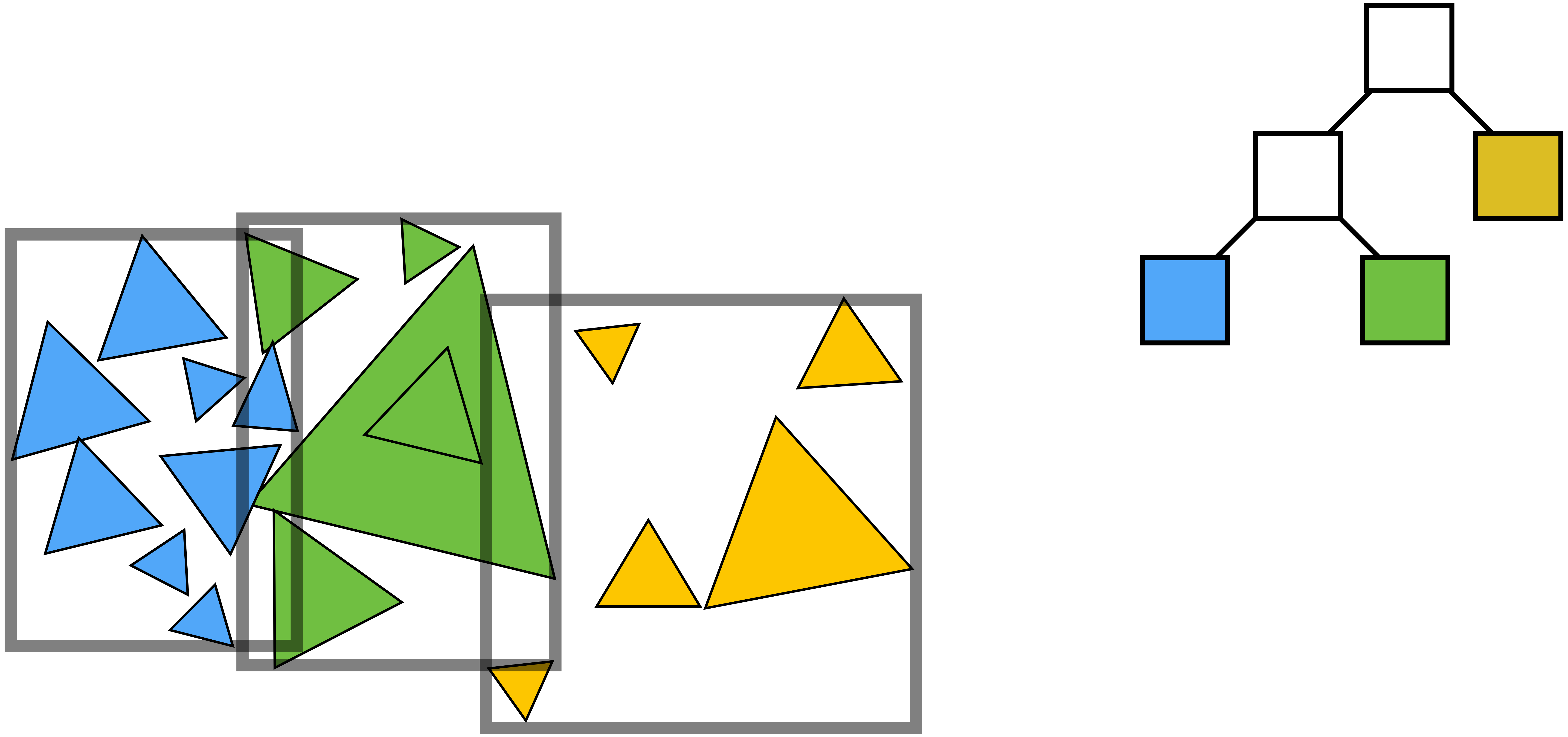


Bounding volume hierarchy (BVH)

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

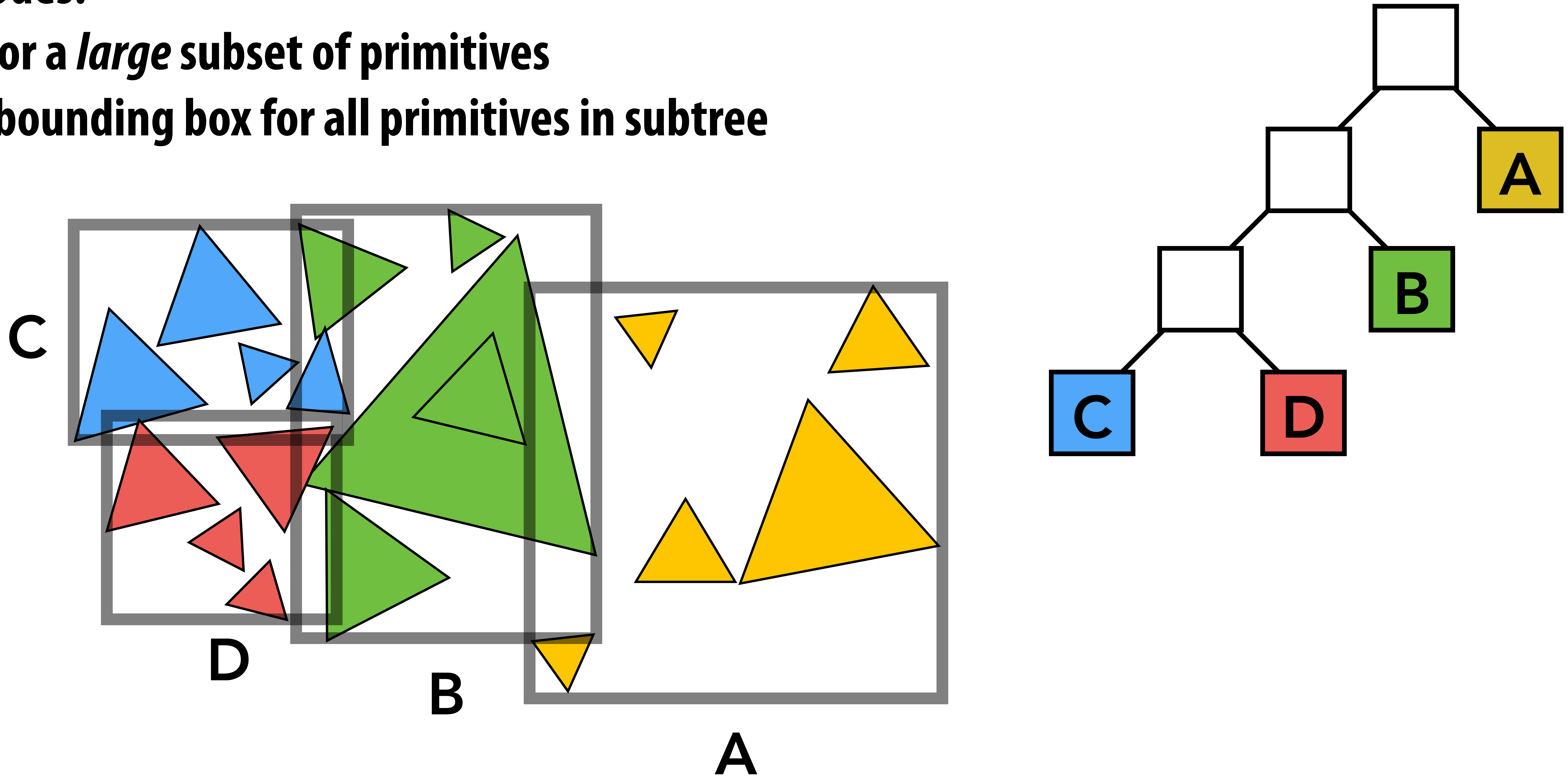


Bounding volume hierarchy (BVH)

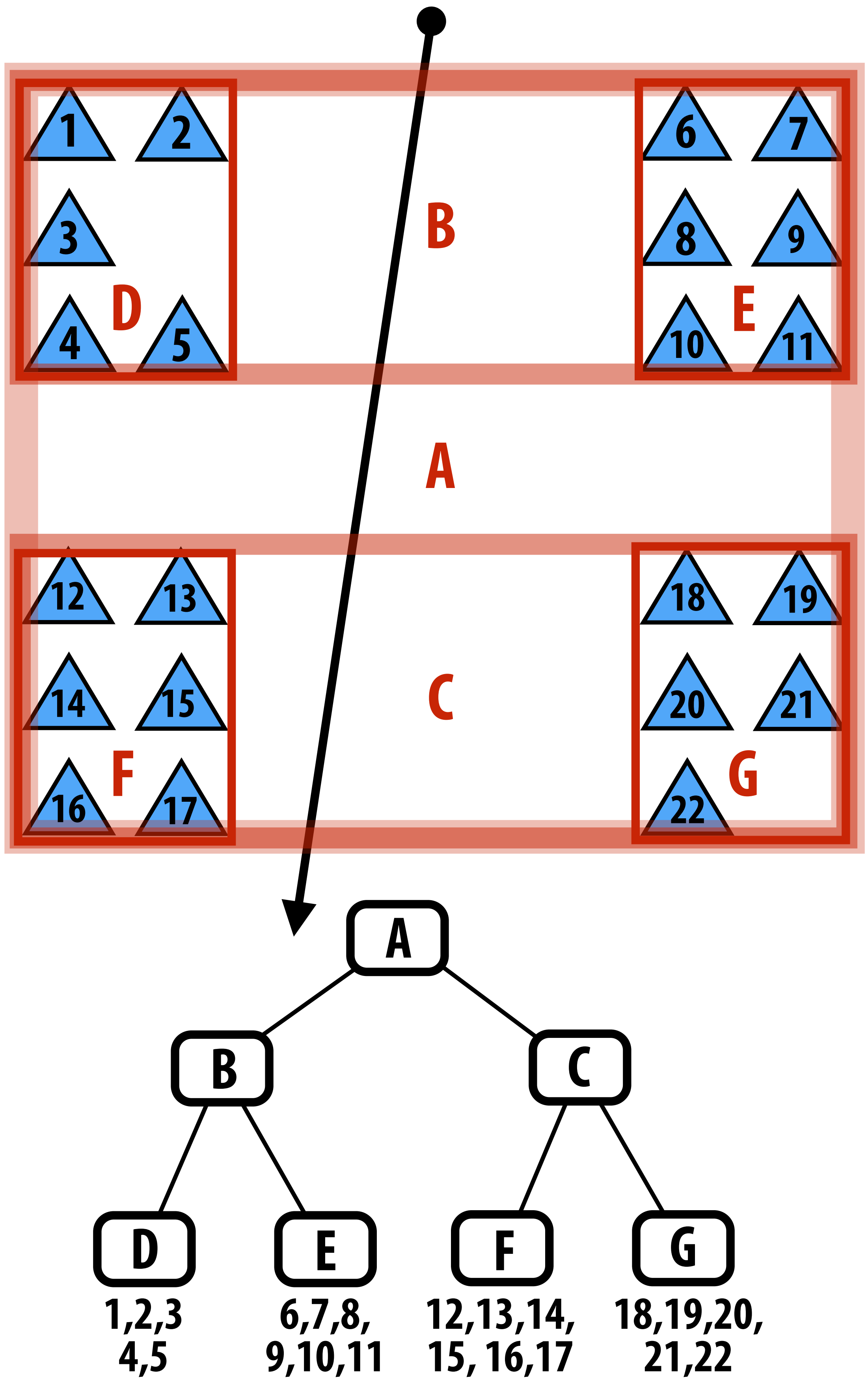


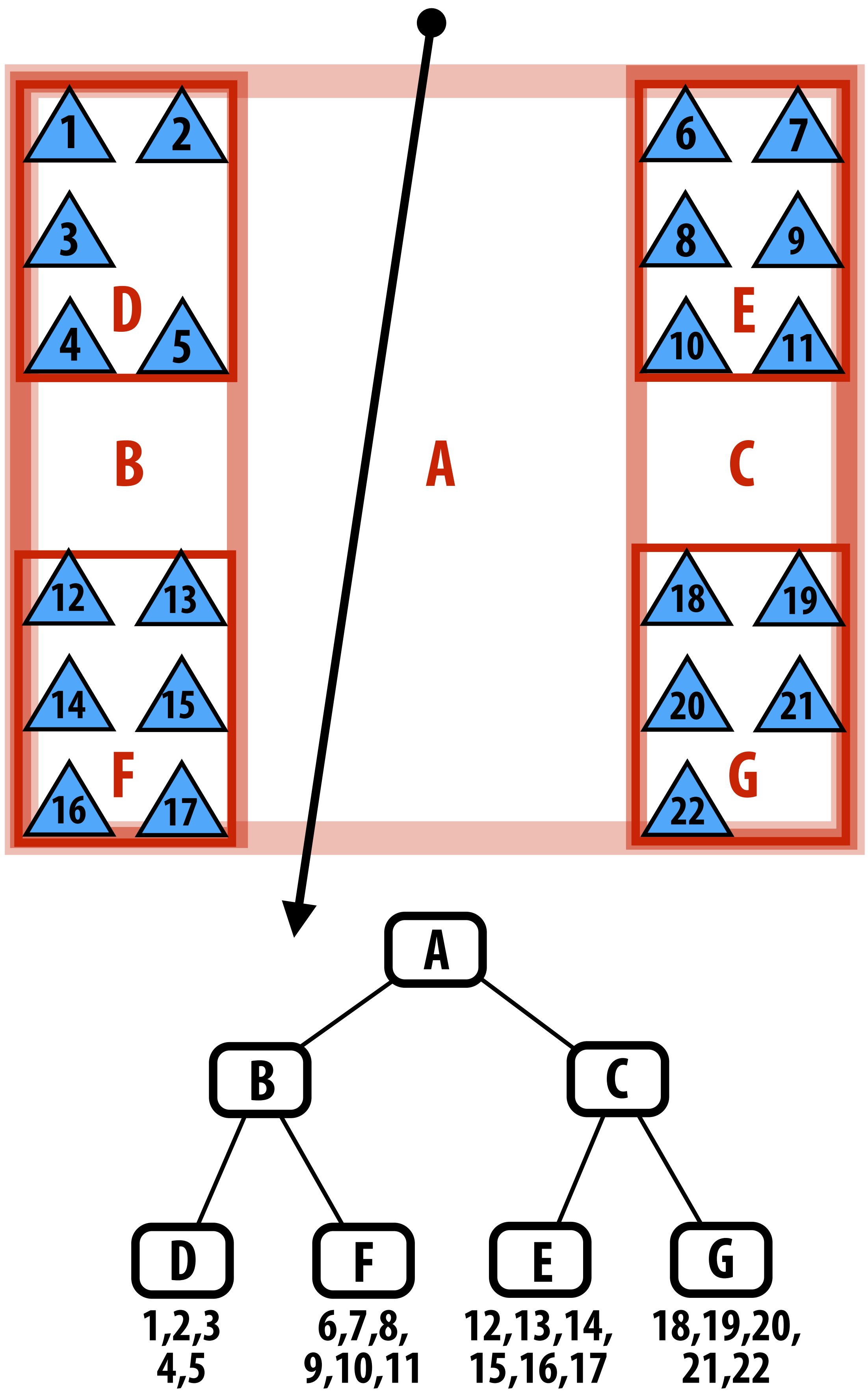
Bounding volume hierarchy (BVH)

- **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)

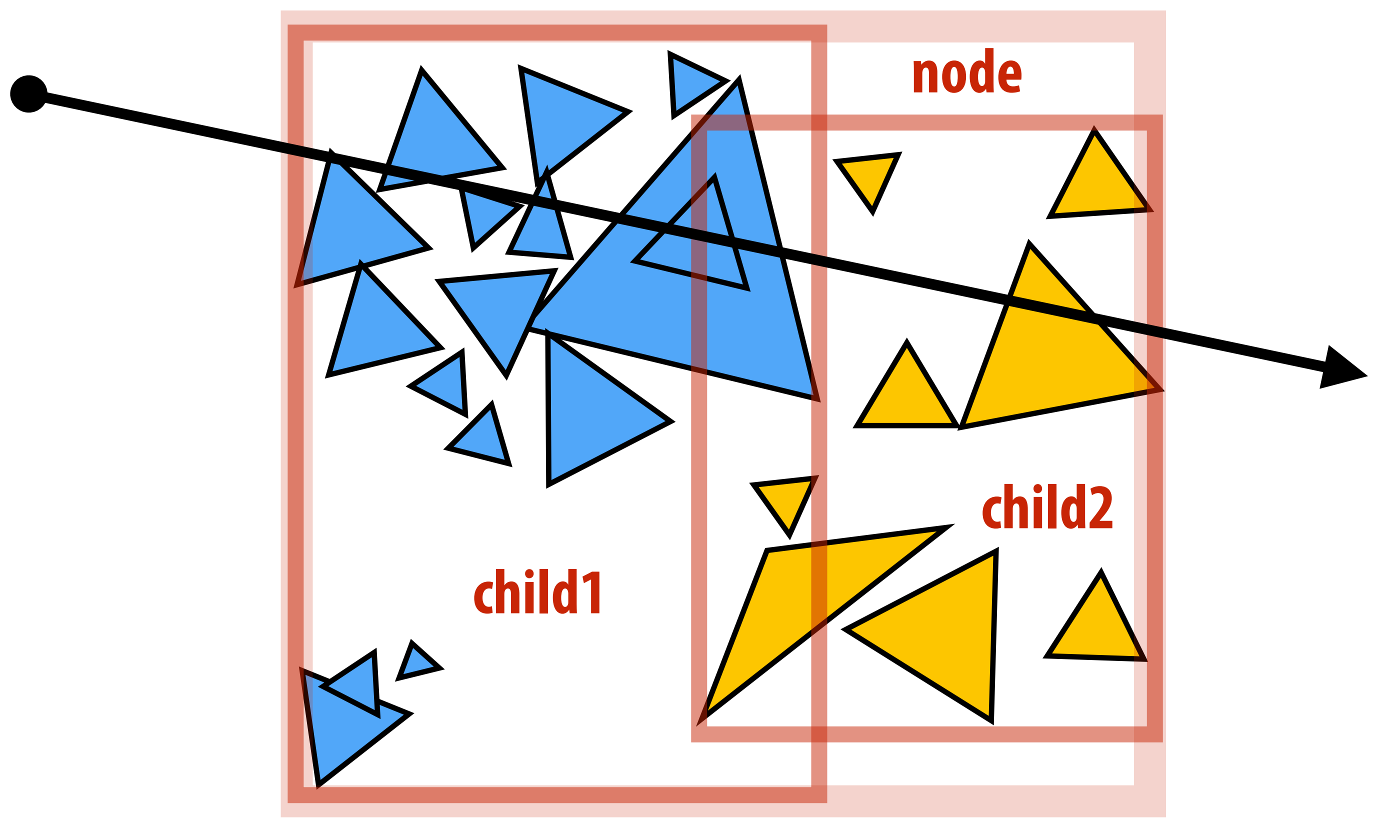




Two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Ray-scene intersection using a BVH



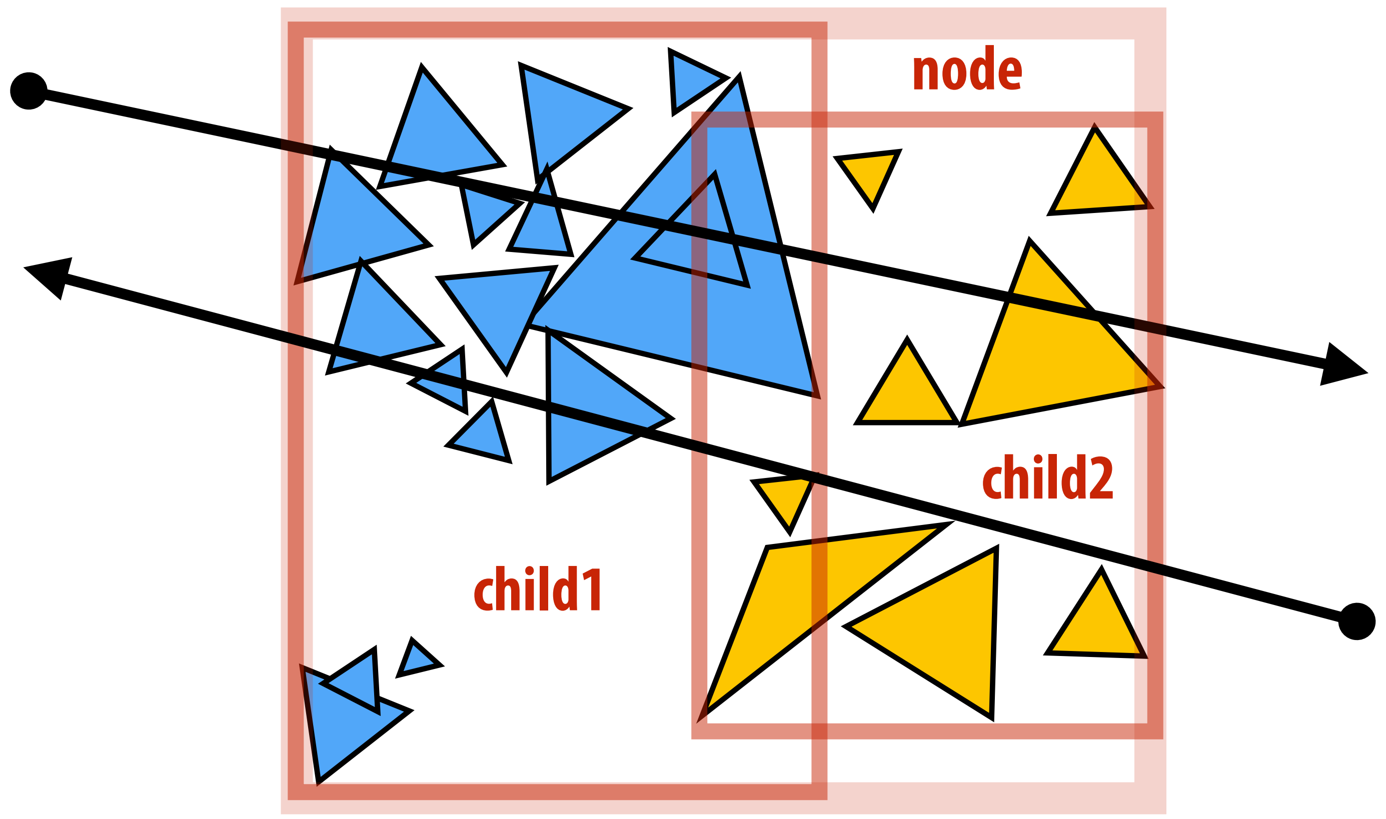
```
struct BVHNode {
   bool leaf;  // true if node is a leaf
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // “left” child (could be NULL)
   BVHNode* child2; // “right” child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;          // at what t value along ray?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node’s bounding box
   if (hit.t > closest.t)
      return; // don’t update the hit record

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

Can this occur if ray hits the box?

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

Improvement: “front-to-back” traversal

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



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

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

      NVHNode* first = (hit1.t <= hit2.t) ? child1 : child2;
      NVHNode* second = (hit1.t <= hit2.t) ? child2 : child1;

      find_closest_hit(ray, first, closest);
      if (second child’s t is closer than closest.t)
         find_closest_hit(ray, second, closest);
    }
}
```

“Front to back” traversal.

Traverse to closest child node first.
Why?

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

BVH traversal workload in a nutshell

- Fetch left/right node bbox data from memory (data loads)
- Ray-bbox intersection (computation)
- Depending on results, move to left or right child node
  - Unpredictable what to load next (depends on ray)
- Repeat…

As always, let's focus here on the data access part of the algorithm.

Takeaway:

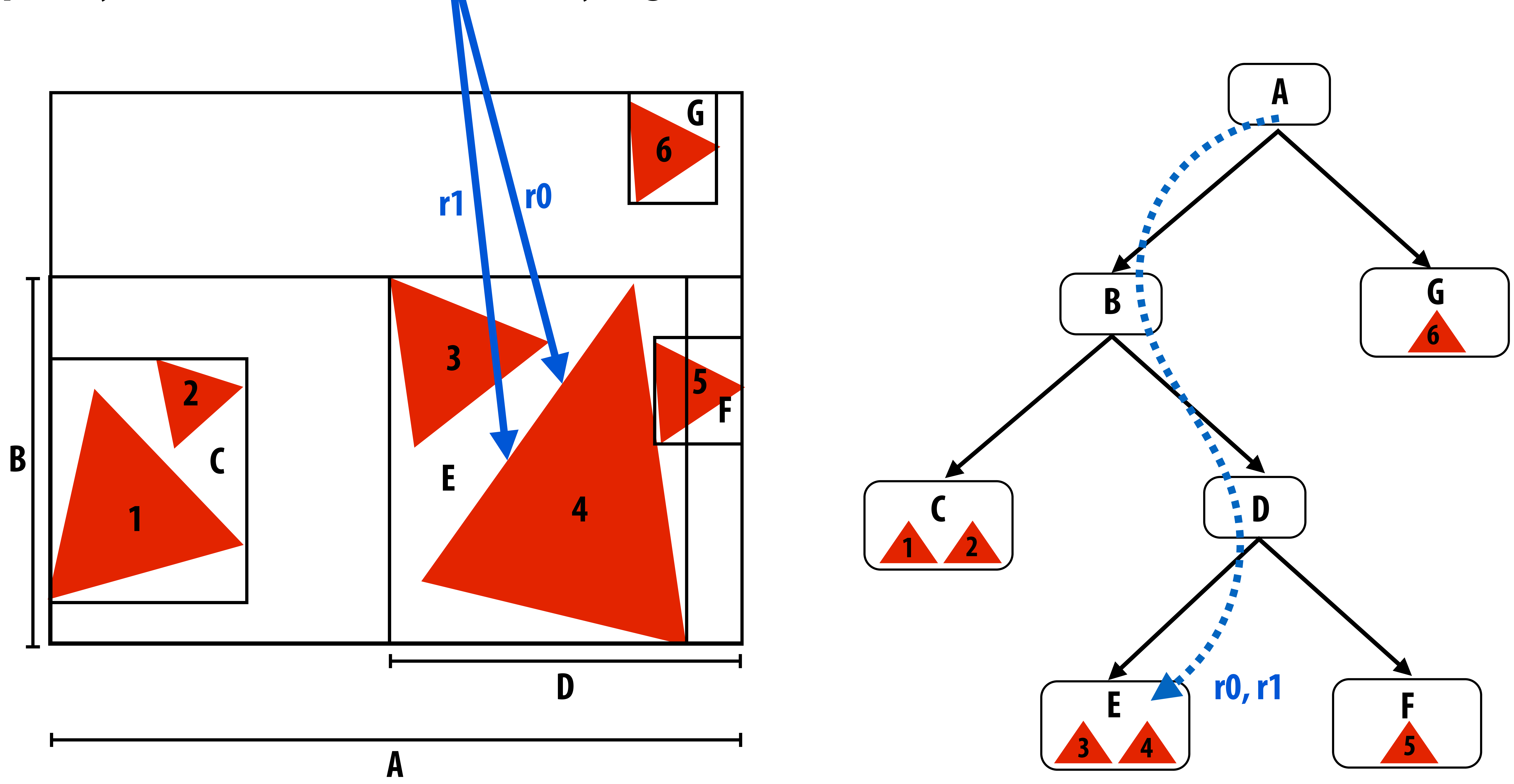
Ray-BVH traversal generates unpredictable (data-dependent) access to an irregular data structure

Understanding ray coherence during BVH traversal

Ray traversal "coherence"

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

Program explicitly intersects a collection of rays against BVH at once

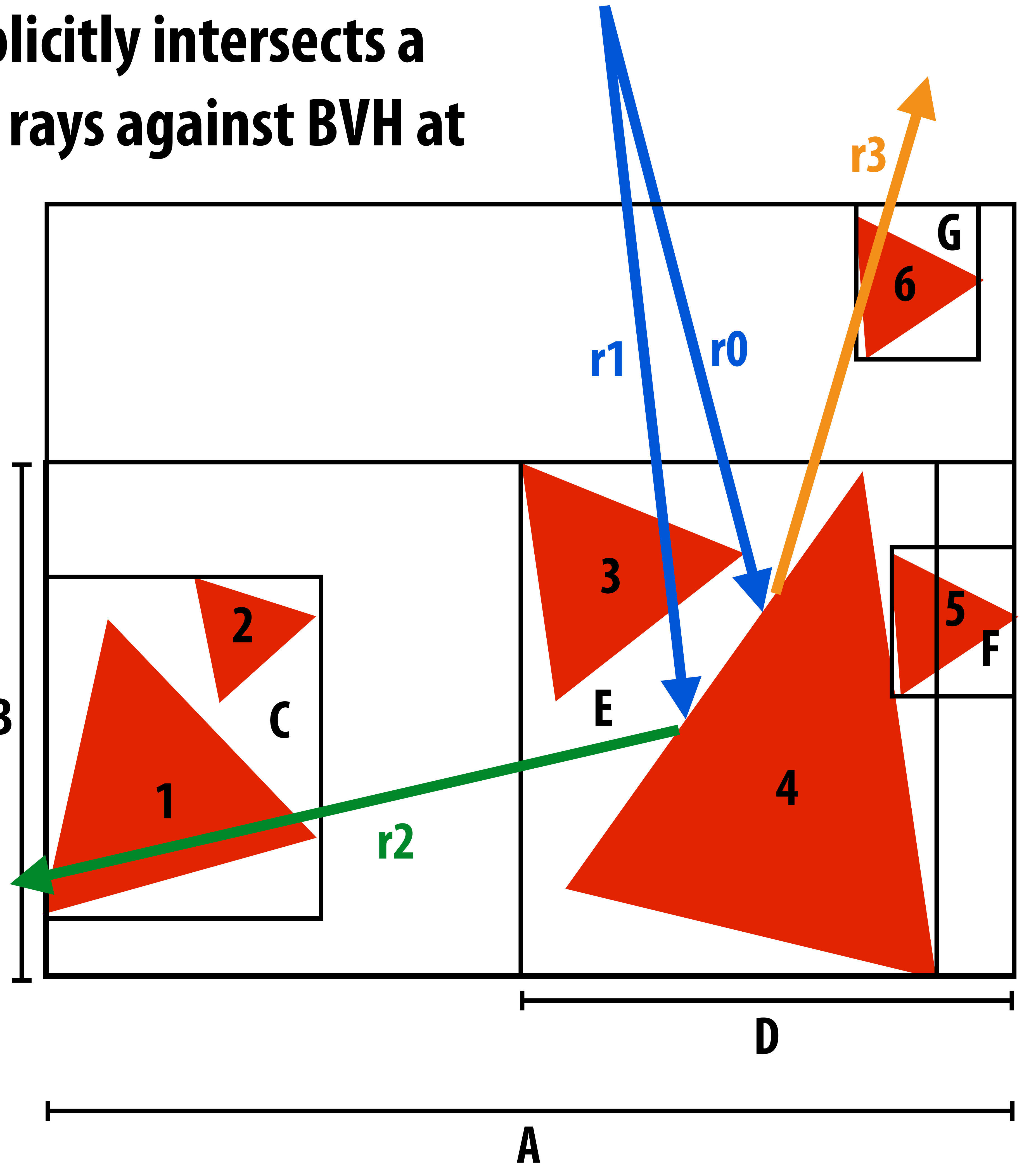


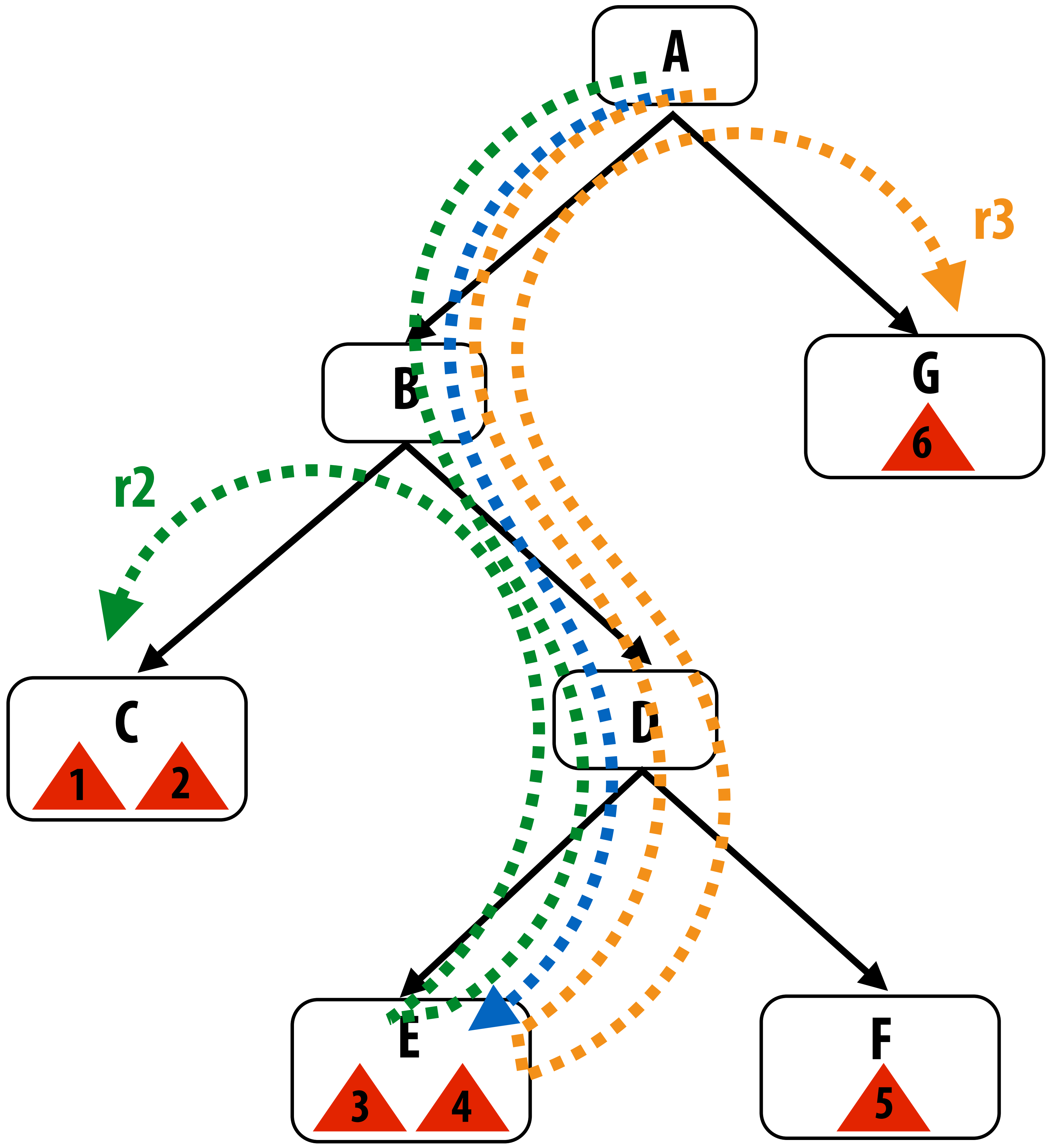
Bandwidth reduction: BVH nodes (and triangles) loaded into cache for computing scene intersection with r0 are cache hits for r1

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…

Program explicitly intersects a collection of rays against BVH at once

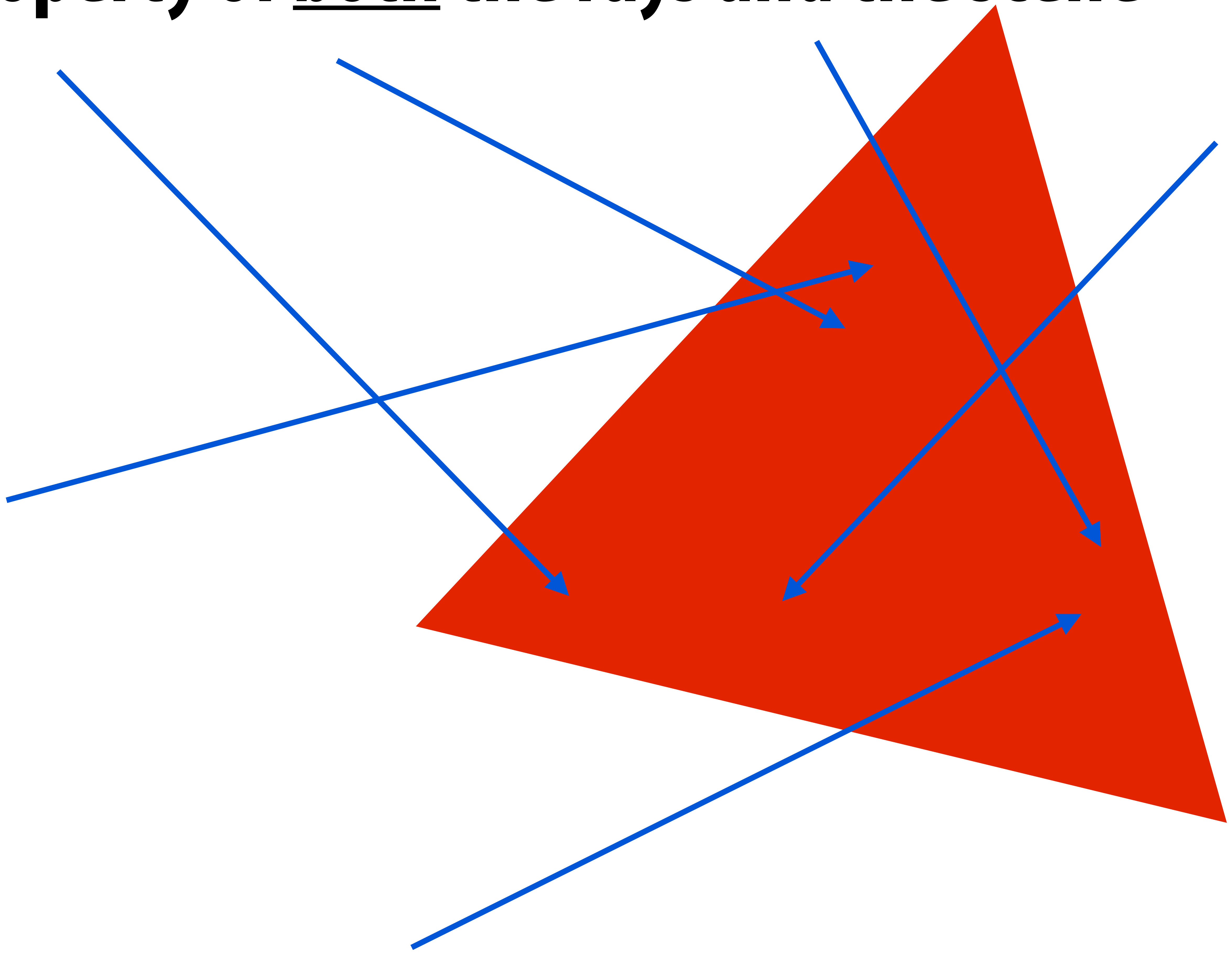




R2 and R3 require different BVH nodes and triangles

Incoherent rays

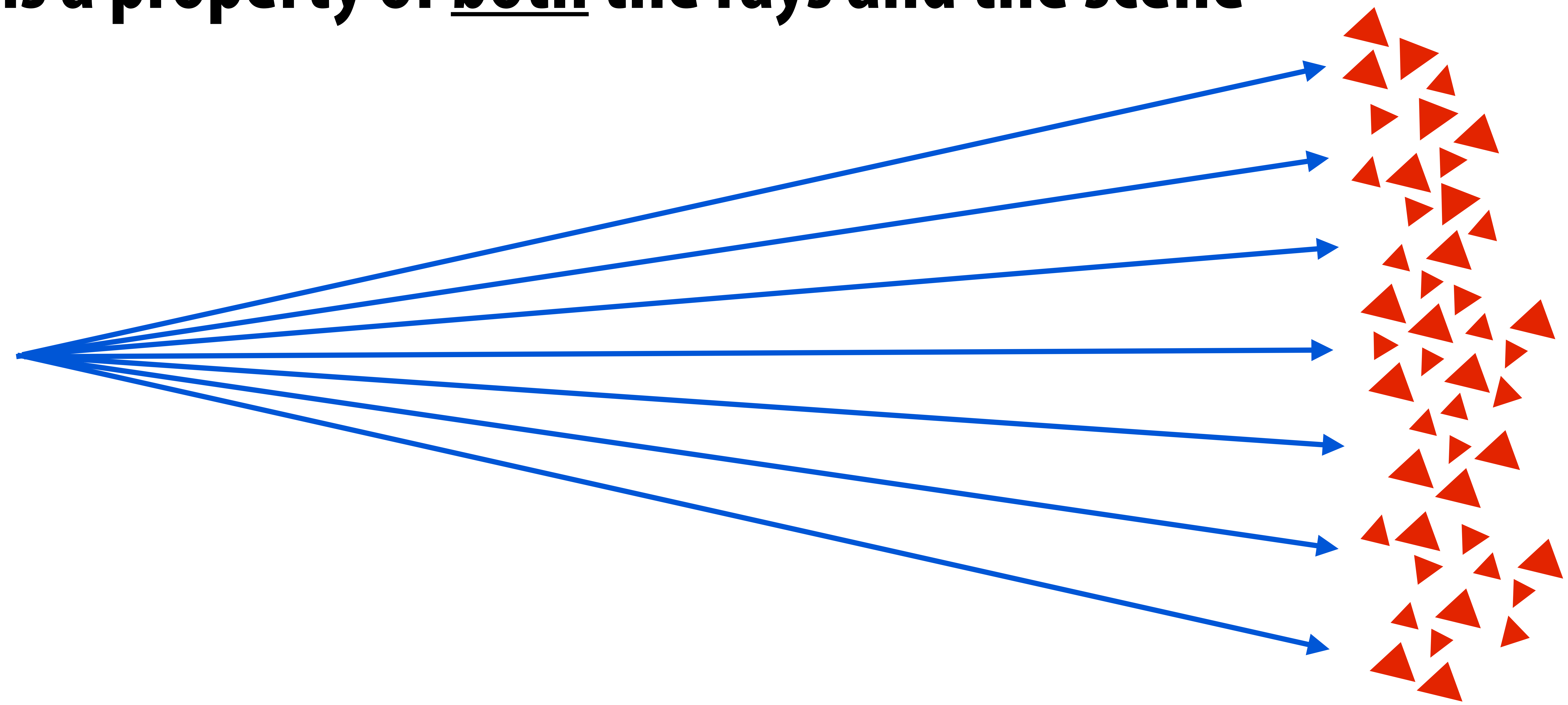
Incoherence is a property of <u>both</u> the rays and the scene



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

Incoherent rays

Incoherence is a property of both the rays and the scene

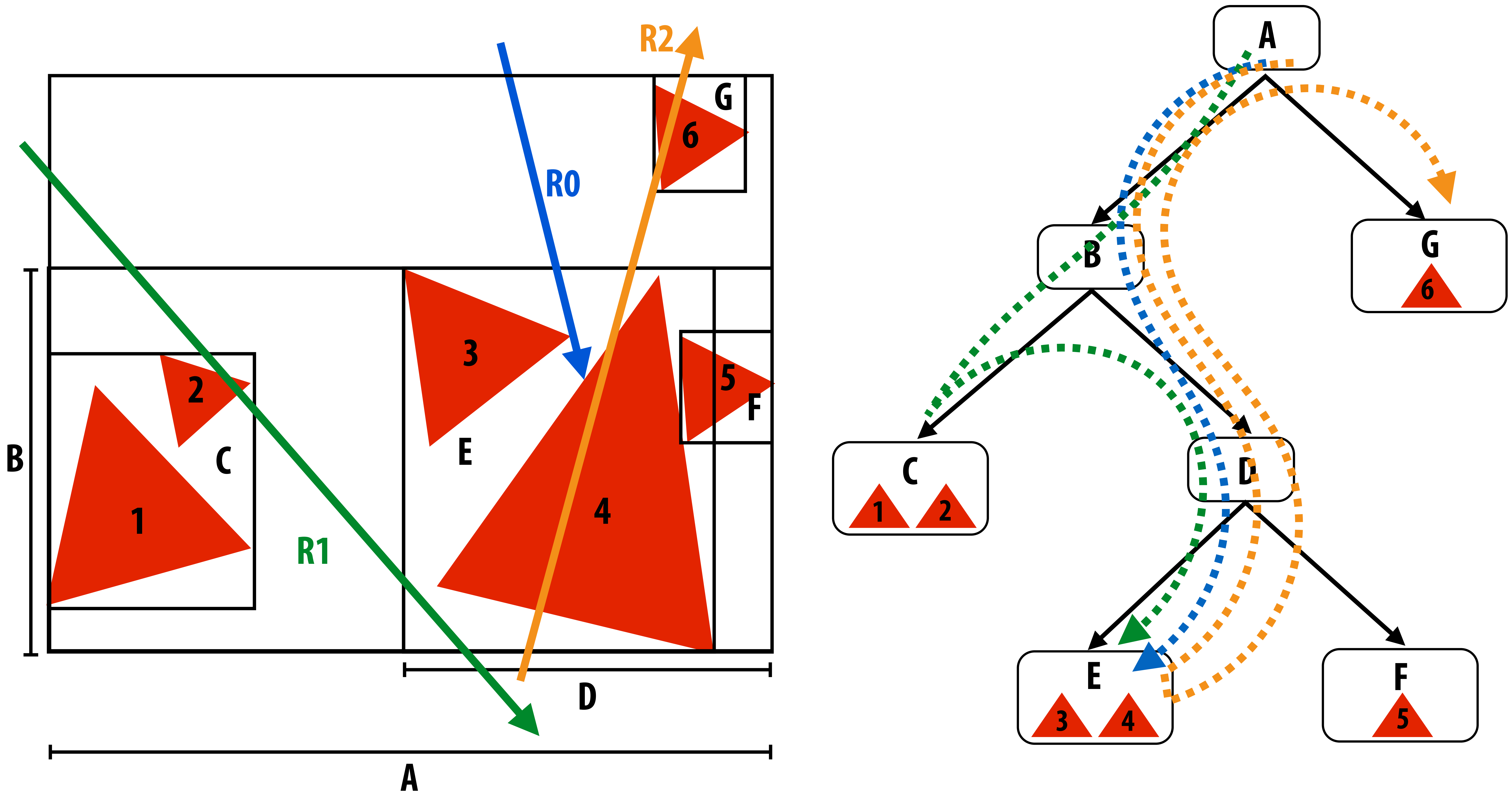


Similarly oriented rays from the same point become "incoherent" with respect to lower nodes in the BVH if a scene is overly detailed

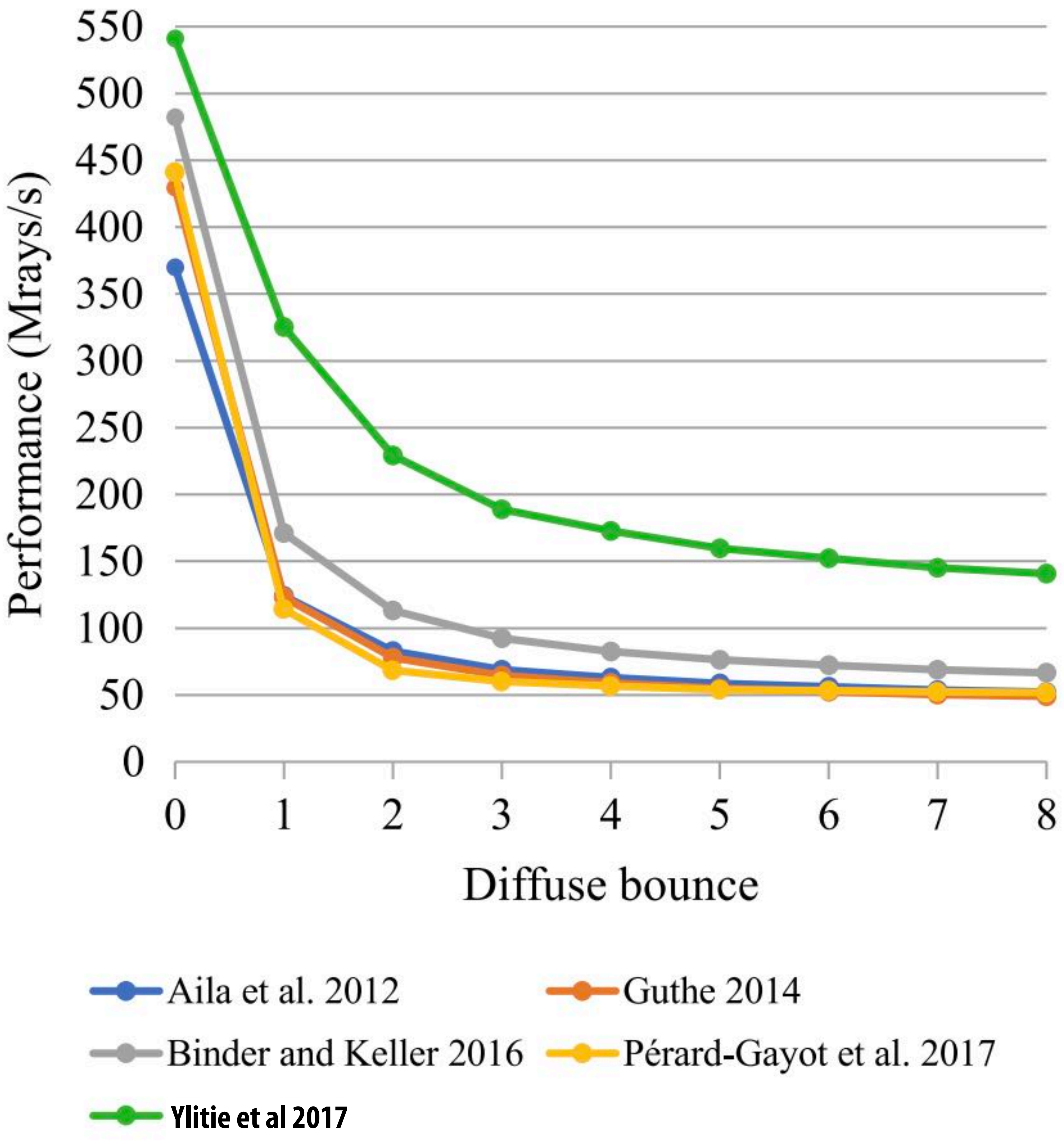
(Side note: this suggests the importance of choosing the right geometric level of detail)

Incoherent rays = bandwidth bound

Different threads may access different BVH nodes at the same time:
Note how R0/R2 are accessing D while R1 is accessing C



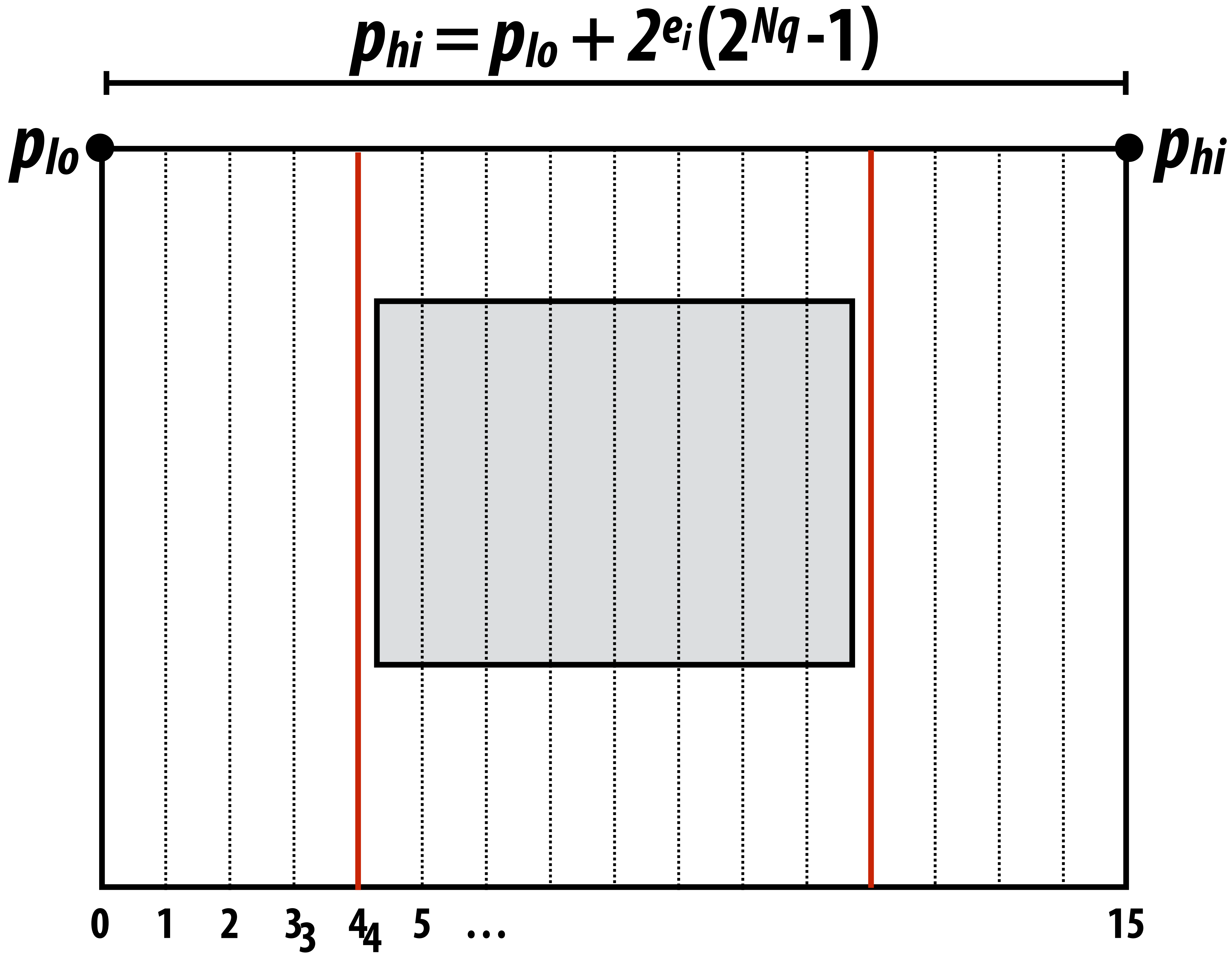
Ray throughput decreases with increasing numbers of bounces (aka increasing ray incoherence)



Idea 1: use compression to reduce data transfer

Reduce bandwidth requirements with BVH compression

Example: store child bboxes as quantized values in local coordinate frame defined by parent node's bbox



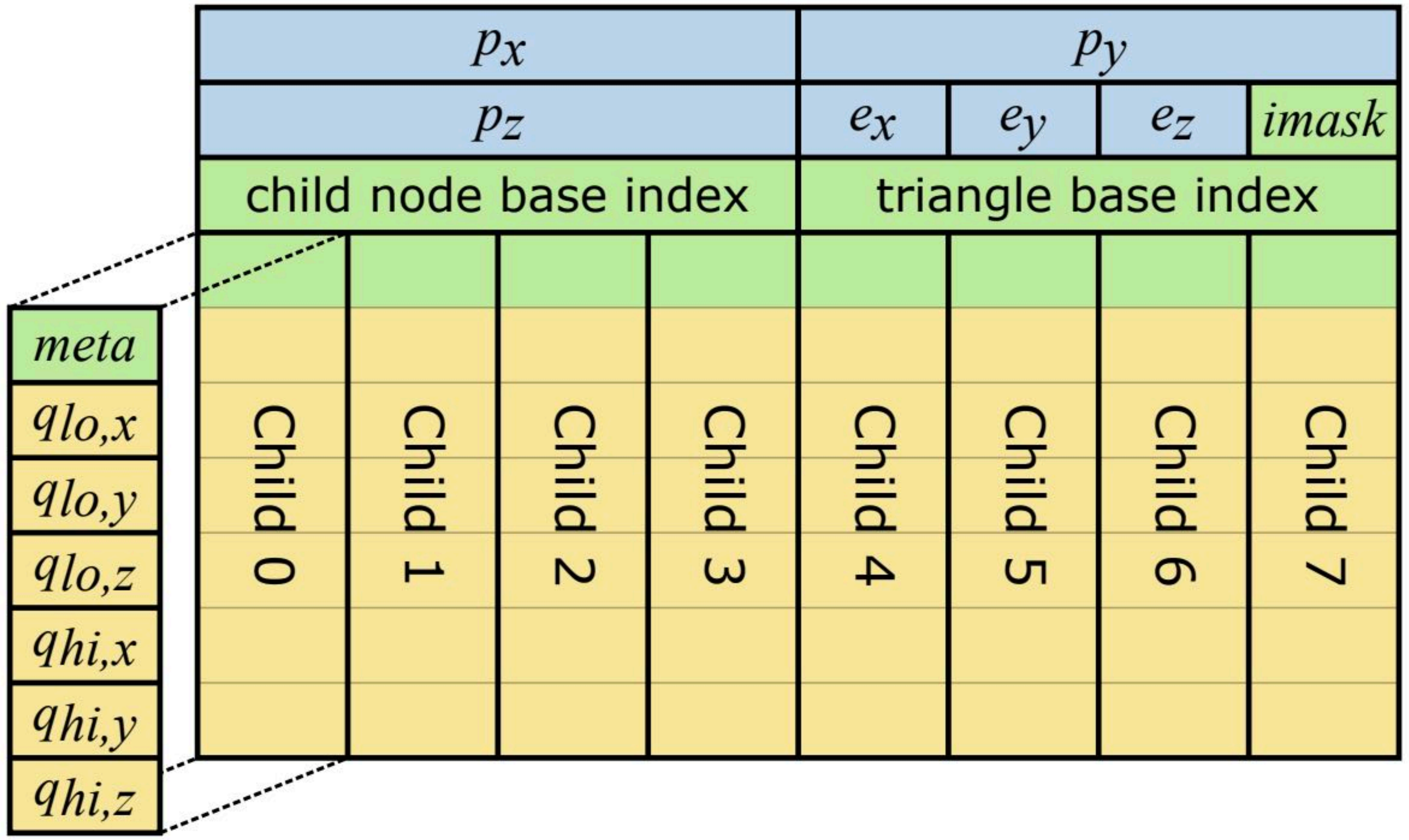
e_i encodes 8 bit exponent that defines "scale" of the parent bbox so that quantized N_q-bit values can be used to represent points in local coordinate frame

So 3D coordinate frame is defined by 3 fp32 values (p_{lo}) and 3 8-bit extent exponents e_i

Planes of child bboxes stored as N_q bit values. Here $N_q = 4$ for illustration, in practice $N_q = 8$
(note quantization expands actual box, reducing efficiency of BVH structure)

BVH compression

- Example: store child bboxes as quantized values in local coordinate frame defined by parent node's bbox
- Use wider BVHs to:
  - Amortize storage of local coordinate frame definition across multiple child nodes
  - Reduce number of BVH node requests during traversal



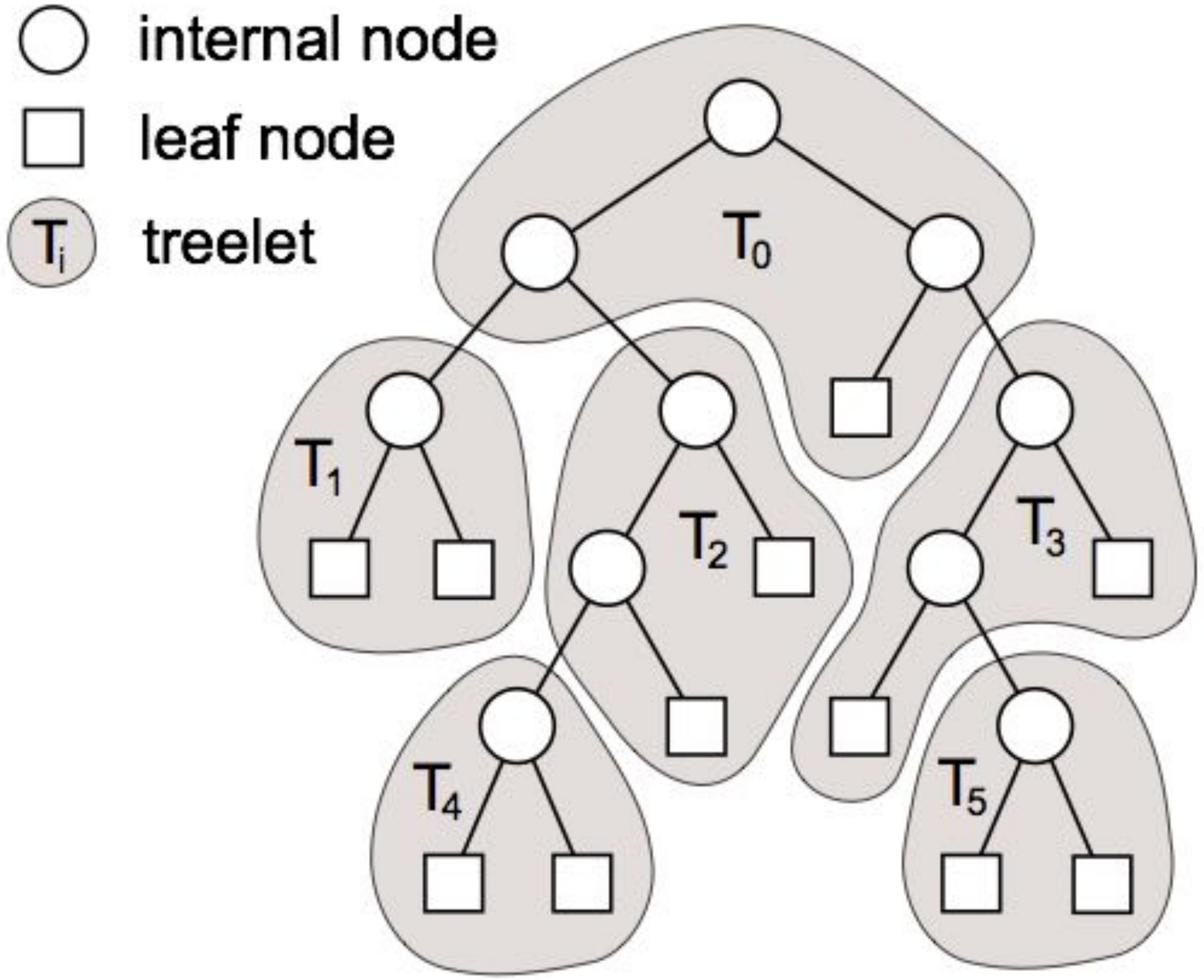
Amortized 10 bytes per child
(3.2x compression over standard BVH formats)

Idea 2: reorder computation to increase locality

[Pharr 1997, Navratil 07, Alia 10]

Queue-based global ray reordering

Idea: dynamically batch up rays that must traverse the same part of the scene. Process these rays together to increase locality in BVH access



Partition BVH into treelets
(treelets sized for L1 or L2 cache)

1. **When ray (or packet) enters treelet, add rays to treelet queue**
2. **When treelet queue is sufficiently large, intersect enqueued rays with treelet**
   (amortize treelet load over all enqueued rays)

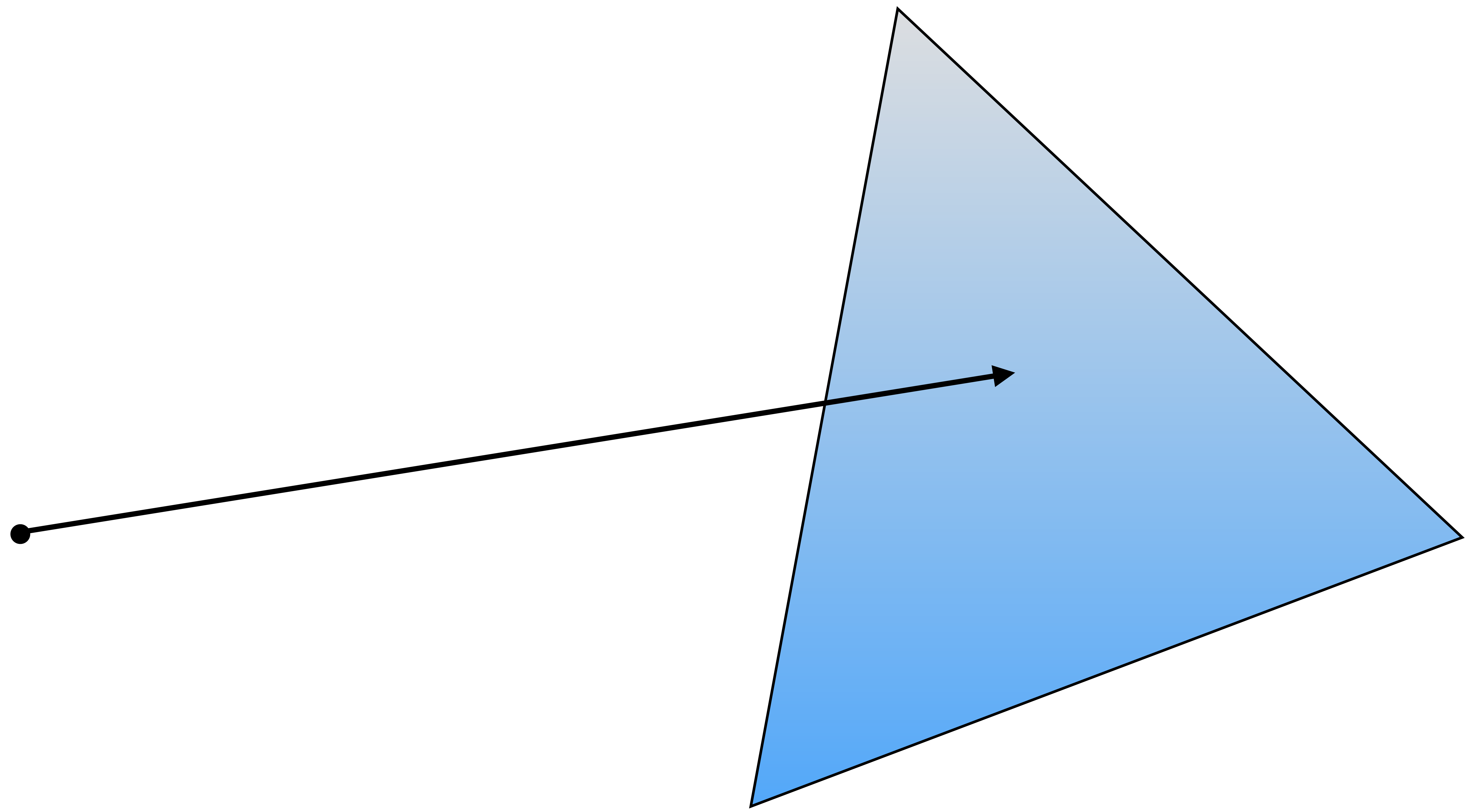
Buffering overhead to global ray reordering: must store per-ray "stack" (need not be entire call stack, but must contain traversal history) for many rays.

Per-treelet ray queues sized to fit in caches
(or in dedicated ray buffer SRAM)

SIMD implications of ray tracing

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.)

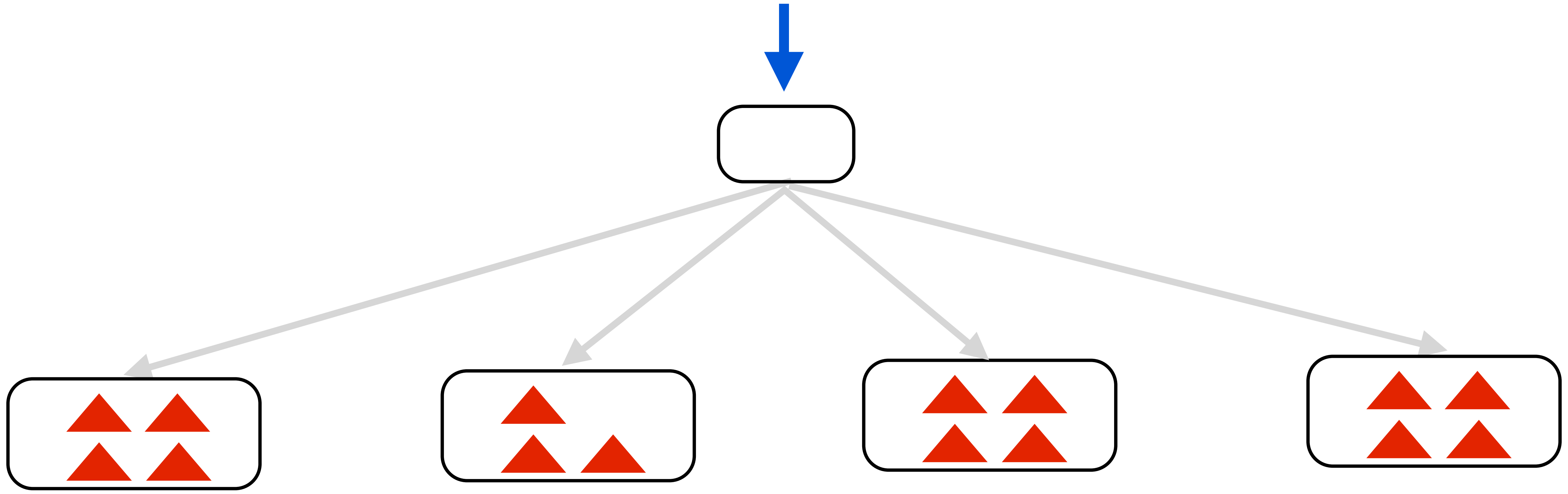
- Similar SIMD parallelism in ray-triangle test at BVH leaf

- 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)



- Note: wider branching factor also reduces height of tree.
  - Reduced number of requests out to memory
  - Wider memory transactions

SPMD ray tracing (GPU-style)

Each work item (e.g., CUDA thread) carried out processing for one ray.

SIMD parallelism comes from executing multiple threads in a WARP

Algorithm 1

```
stack<BVHNode> tovisit;
tovisit.push(root);
while (ray not terminated)

   // ray is traversing interior nodes
   while (not reached leaf node)
     traverse node // pop stack, perform
                   // ray-box test, push
                   // children to stack

   // ray is now at leaf
   while (not done testing tris in leaf)
     ray-triangle test
```

Algorithm 2

```
stack<BVHNode> tovisit;
tovisit.push(root);
while (ray not terminated)
   node = tovisit.pop();
   if (node is not a leaf)
      traverse node // perform ray-box test,
                    // push children to stack

   else (not done testing tris in leaf)
      ray-triangle test
```

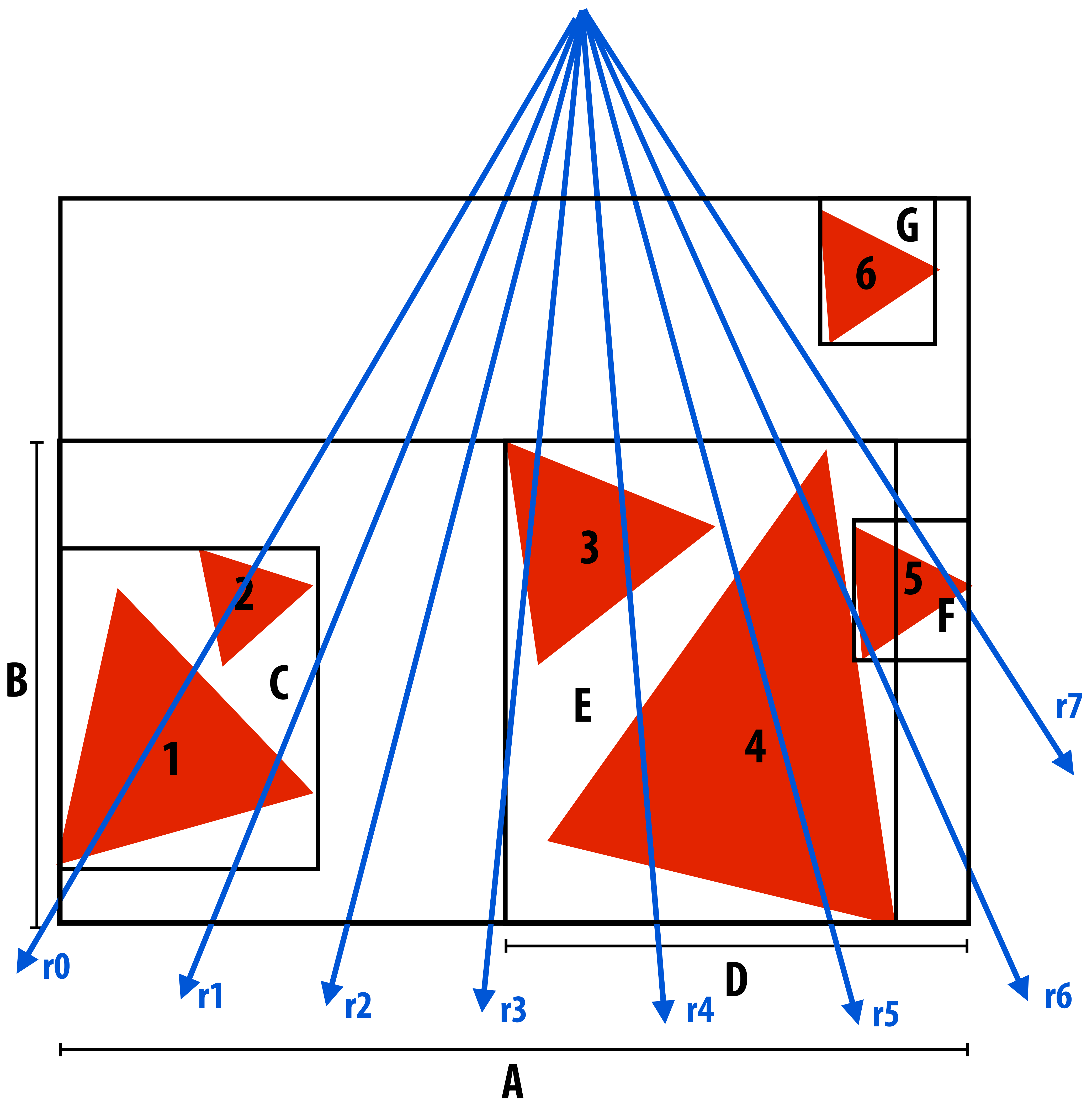
Ray packet tracing (CPU-style SIMD ray tracing)

[Wald et al. 2001]

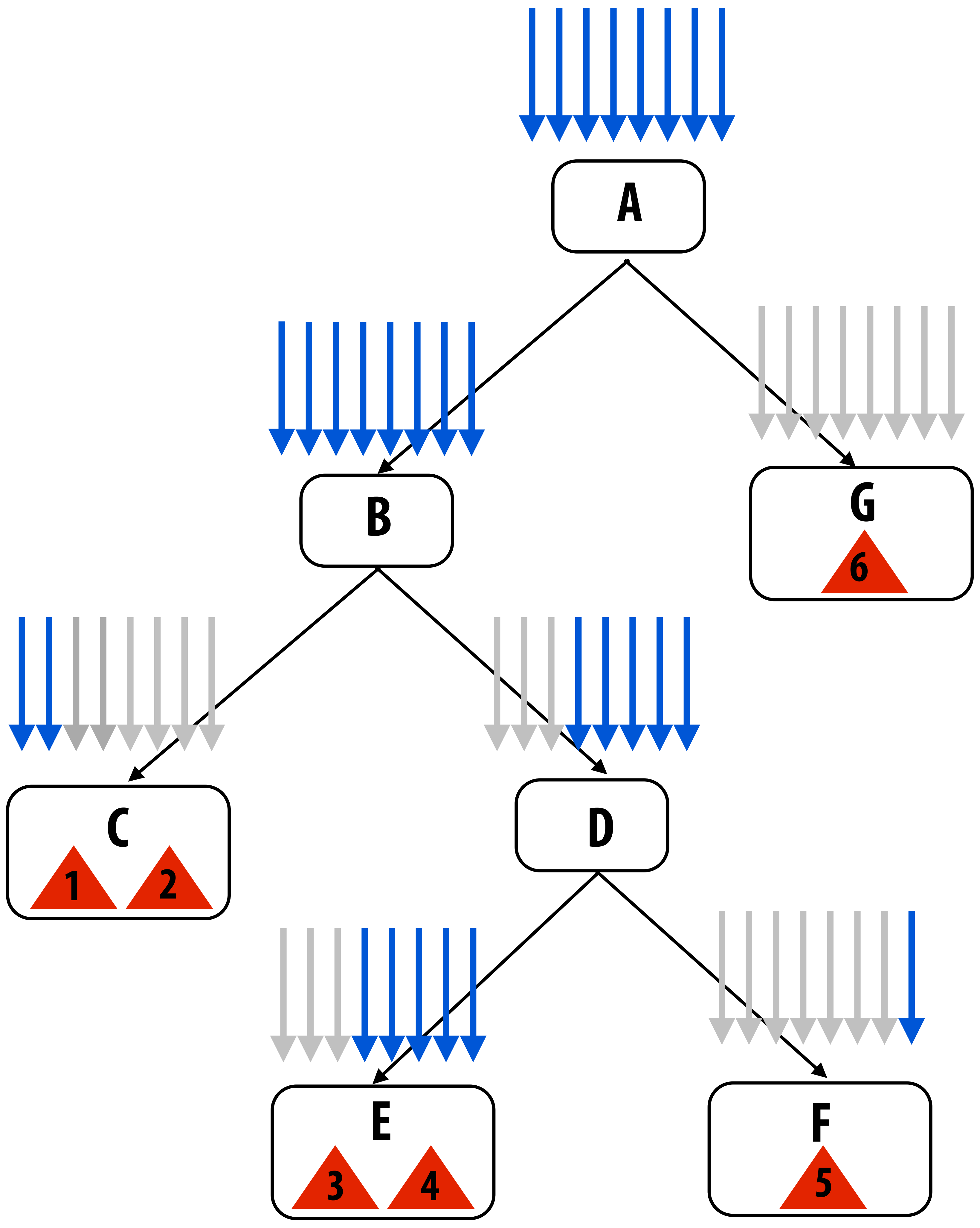
Program explicitly intersects a collection of rays against BVH at once

```
RayPacket
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

trace(RayPacket rays, BVHNode node, ClosestHitInfo packetHitInfo)
{
   if (!ANY_ACTIVE_intersect(rays, node.bbox) ||
       (closest point on box (for all active rays) is farther than hitInfo.distance))
      return;

   update packet active mask

   if (node.leaf) {
      for (each primitive in node) {
         for (each ACTIVE ray r in packet) {
            (hit, distance) = intersect(ray, primitive);
            if (hit && distance < hitInfo.distance) {
               hitInfo[r].primitive = primitive;
               hitInfo[r].distance = distance;
            }
         }
      }
   } else {
     trace(rays, node.leftChild, hitInfo);
     trace(rays, node.rightChild, hitInfo);
   }
}
```

Ray packet tracing



Blue = active rays after node box test



Note: r6 does not pass node F box test due to closest-so-far check, and thus does not visit F

Performance advantages of packets

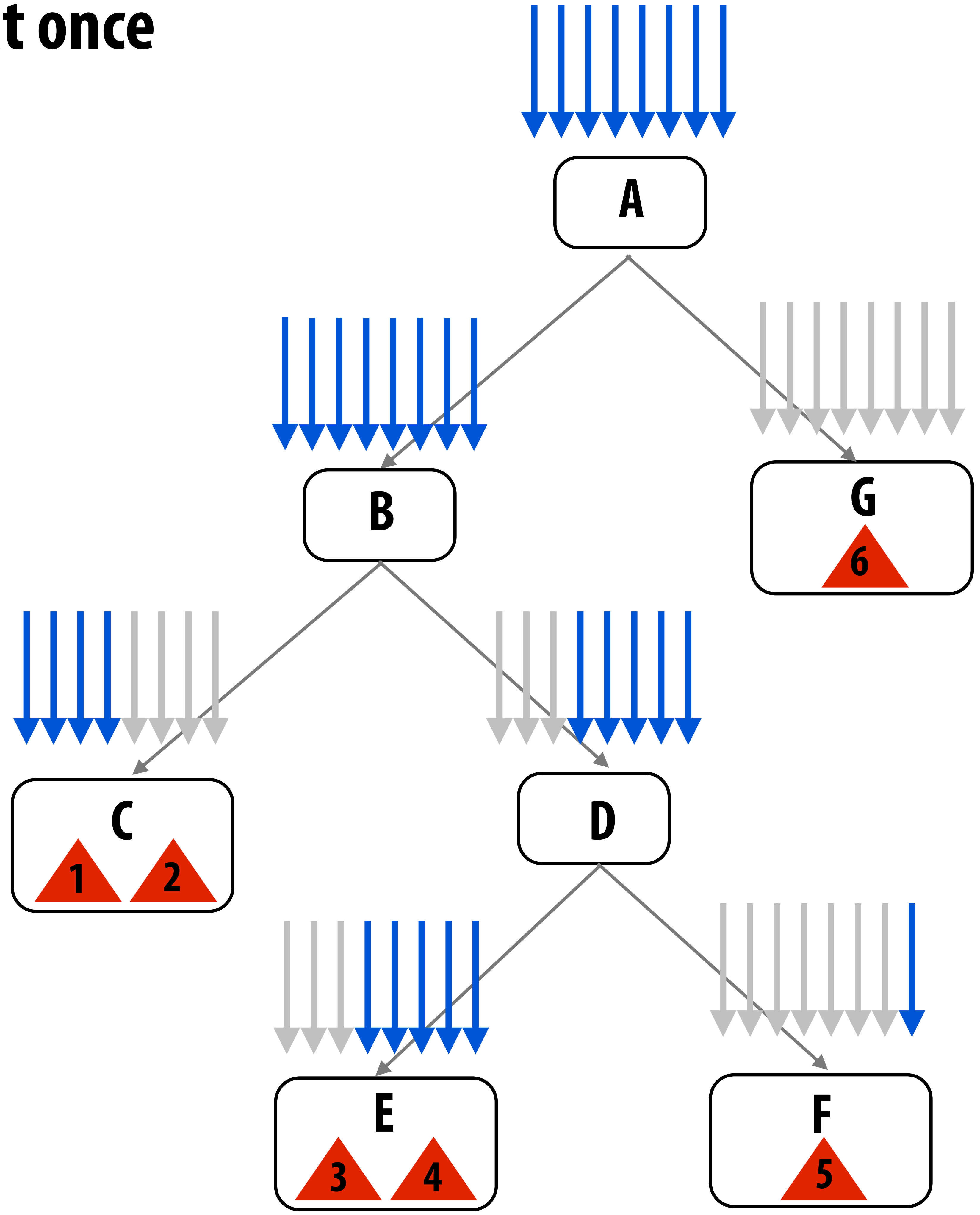
- Wide SIMD execution
  - One vector lane per ray
- Amortize BVH data fetch: all rays in packet visit node at same time
  - Load BVH node once for all rays in packet (not once per ray)
  - Note: there is value to making packets bigger than SIMD width! (e.g., size = 64)
- Amortize work (packets are hierarchies over rays)
  - Use interval arithmetic to conservatively test entire set of rays against node bbox (e.g., think of a packet as a beam)
  - Further arithmetic optimizations possible when all rays share origin
  - Note: there is value to making packets much bigger than SIMD width!

Disadvantages of packets

Program explicitly intersects a collection of rays against BVH at once

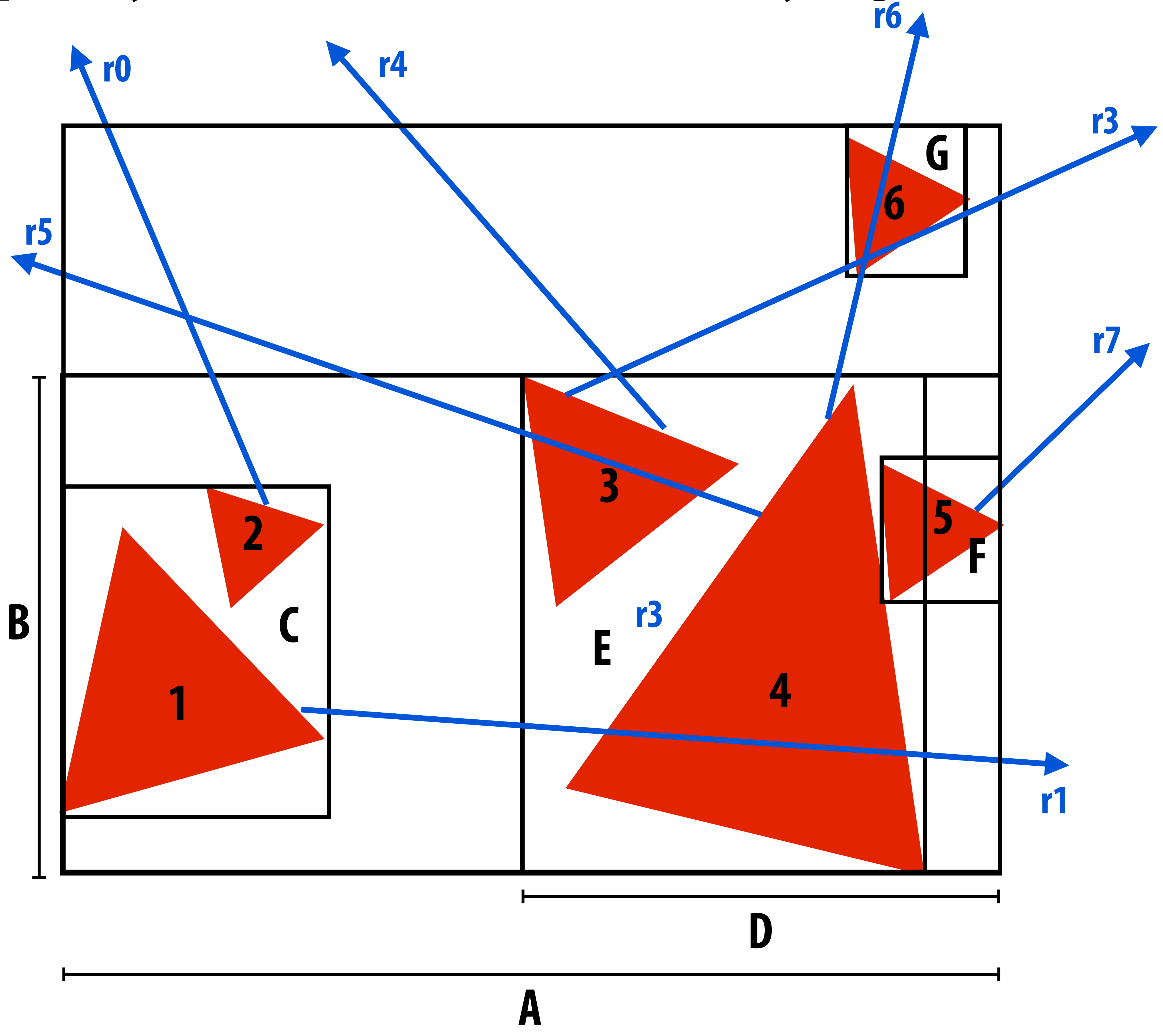
- If any ray must visit a node, it drags all rays in the packet along with it)
- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays
- Not all SIMD lanes doing useful work

Blue = active ray after node box test

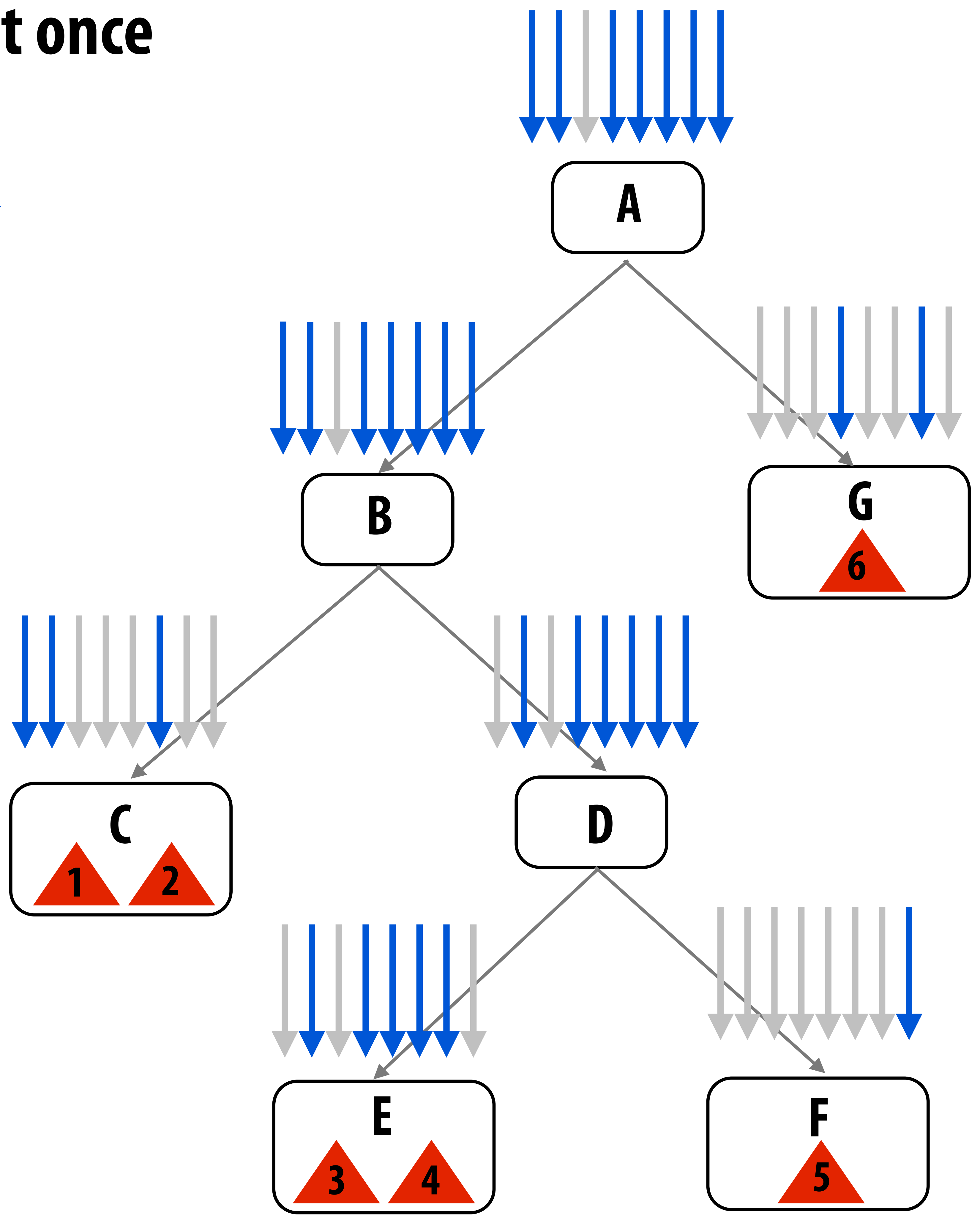


Ray packet tracing: incoherent rays

Program explicitly intersects a collection of rays against BVH at once



Blue = active ray after node box test



When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (So all eight rays visit all tree nodes! No culling benefit!)

Packet tracing best practices

- **Use large packets for eye/reflection/point light shadow rays or higher levels of BVH** [Wald et al. 2007]
  - Ray coherence always high at the top of the tree
- **Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold** [Benthin et al. 2011]
  - For wide SIMD machine, a branching-factor-4 BVH works well for both packet traversal and single ray traversal
- **Can use packet reordering to postpone time of switch** [Boulos et al. 2008]
  - Reordering allows packets to provide benefit deeper into tree
  - Not often used in practice due to high implementation complexity

Ray incoherence impacts efficiency of shading

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

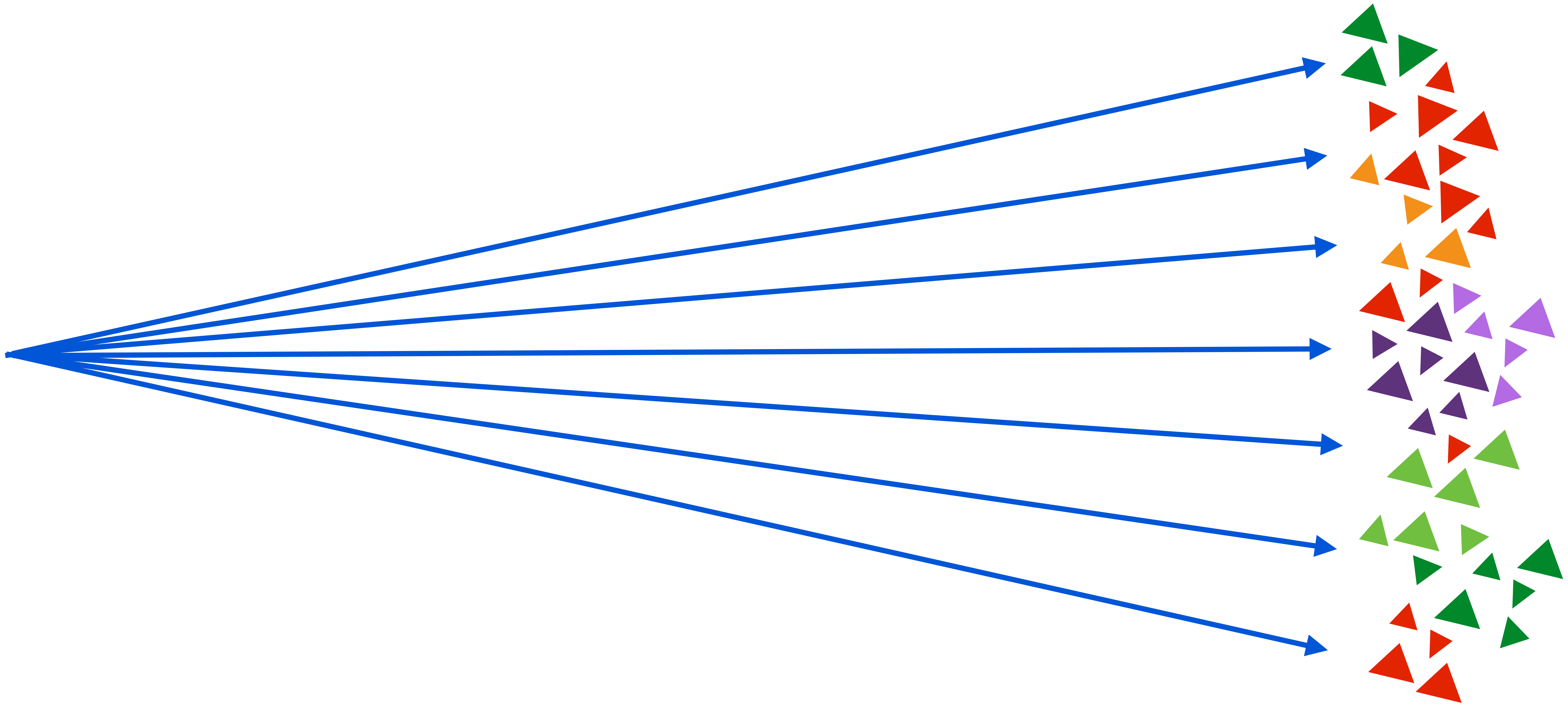
Consider implications for SIMD processing

When rays hit different surfaces…

Surface shading incoherence:

Different code paths needed to compute the reflectance of different materials

[OR] use same highly parameterized “ubershader” (“megakernel”) for all surfaces



Ray tracing performance challenges

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

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

Incoherent shading

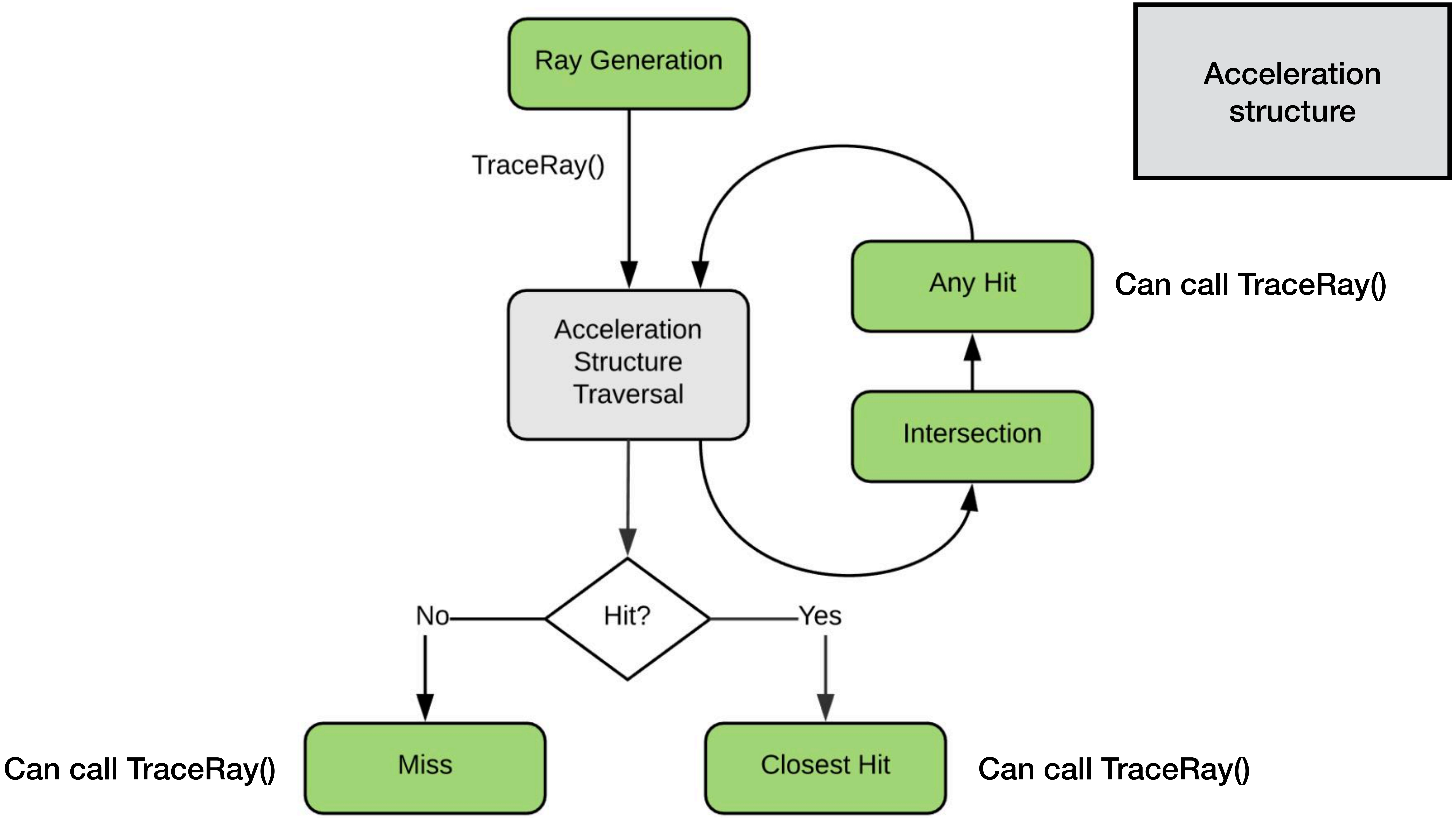
Not discussed today: building the BVH structure each frame

Real-time ray tracing APIs

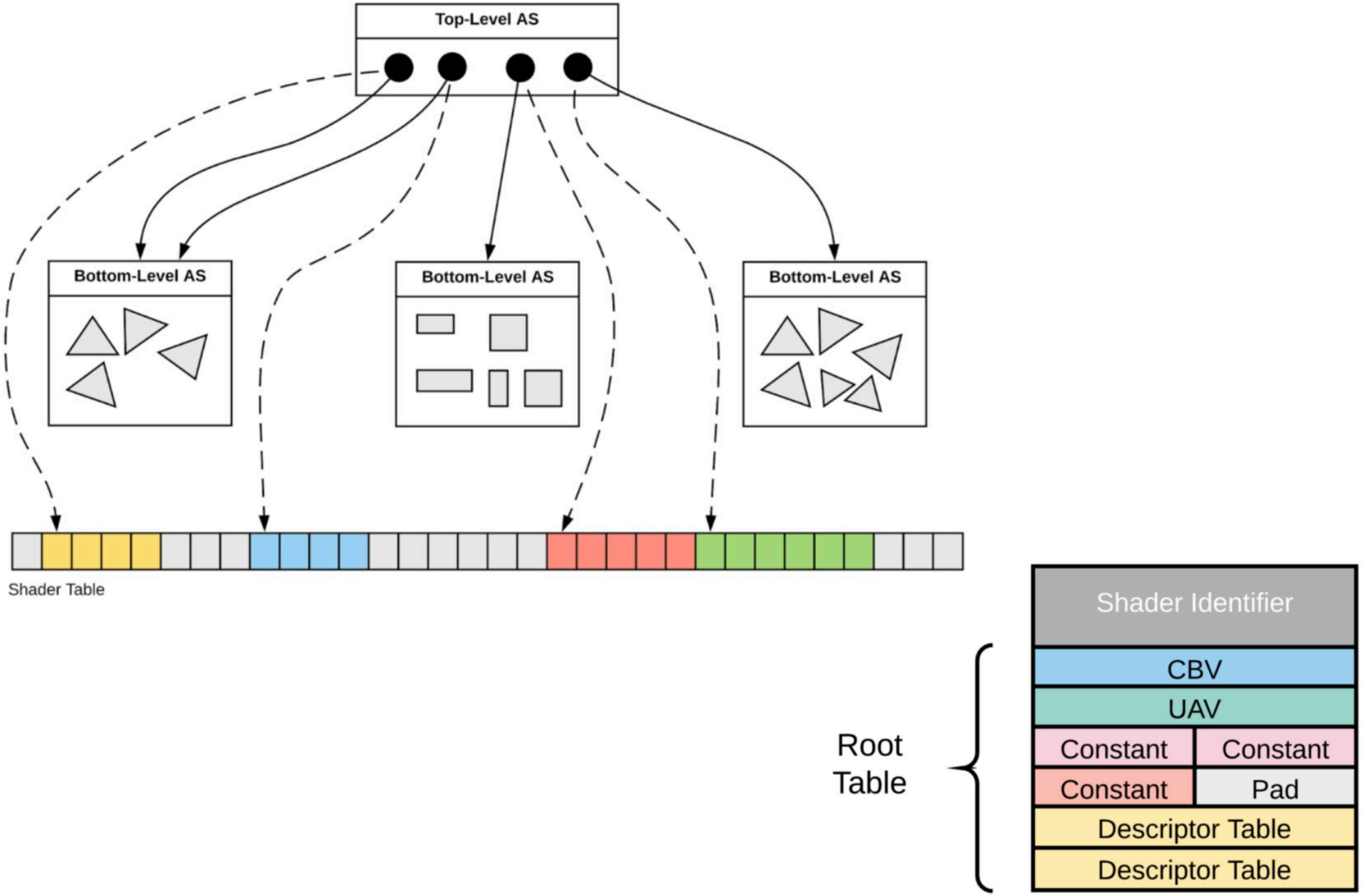
(Recurring theme in this course: increase level of abstraction to enable optimized implementations)

D3D12's DXR ray tracing "stages"

- Ray tracing is abstracted as a graph of programmable "stages"
- TraceRay() is a blocking function in some of those stages



GPU understands format of BVH acceleration structure and "shader table"



Hardware acceleration for ray tracing

NVIDIA Ampere SM (RTX 3xxx series)

- Hardware support for ray-triangle intersection and ray-BVH intersection ("RT core")
- Very little public documentation of architectural details at this time