Lecture 18:

Real Time, GPU-Accelerated Ray Tracing

Visual Computing Systems
Stanford CS348K, Spring 2023

Realistic illumination

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


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

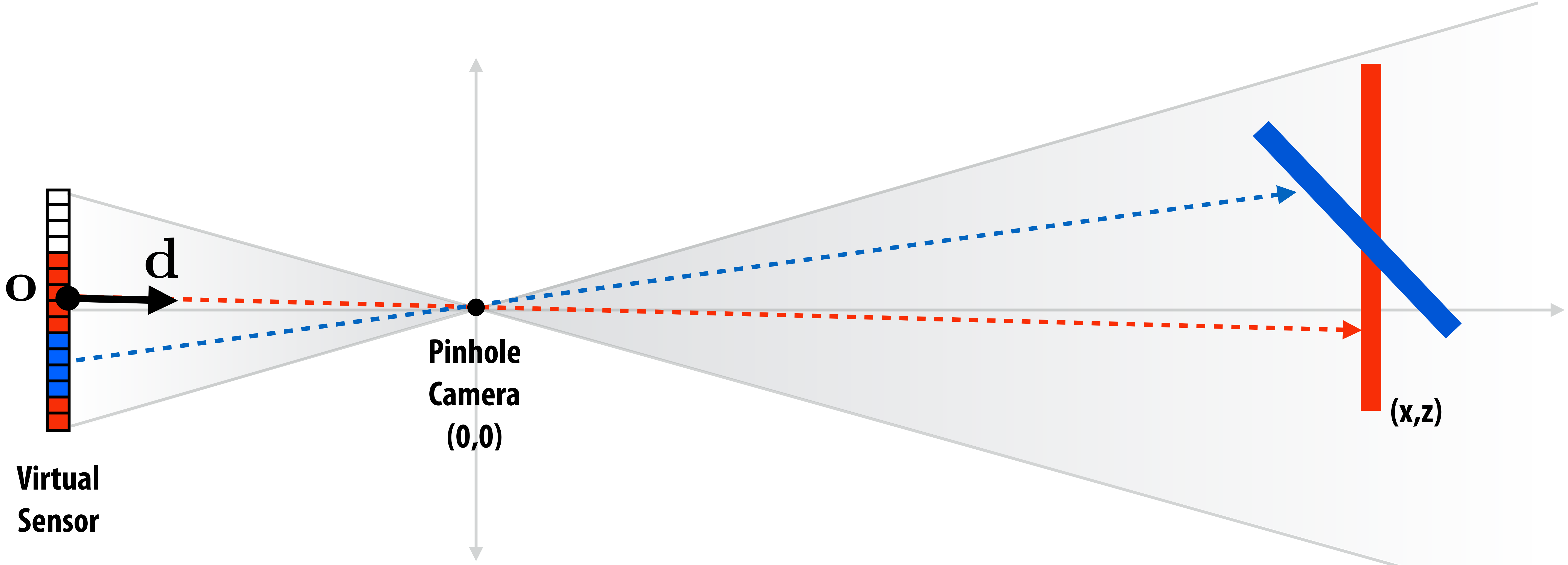
Background/review:
ray tracing in < 10 minutes

Take that Pete Shirley!

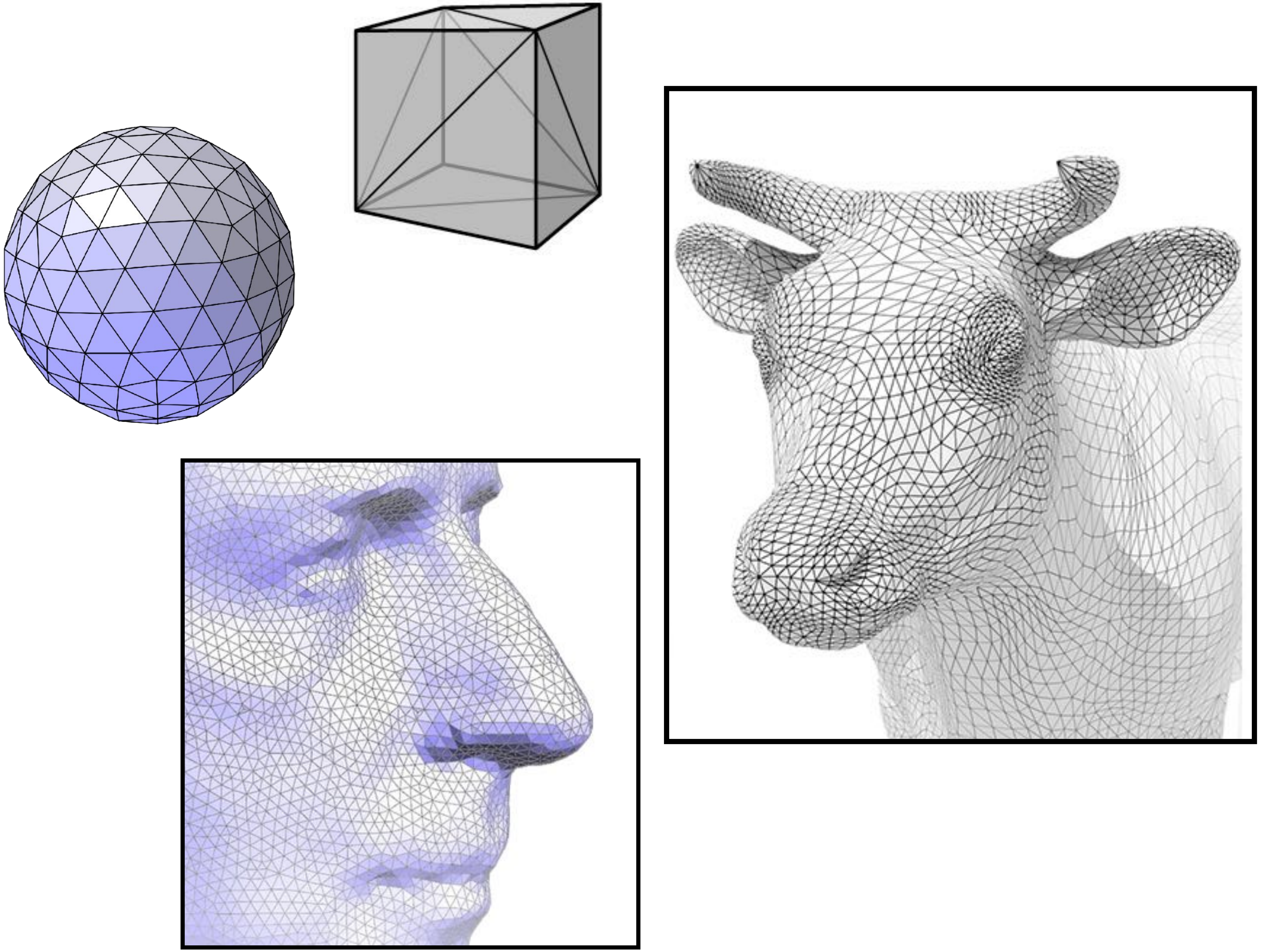
Why do we trace rays?

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)



Today: scene geometry = triangles

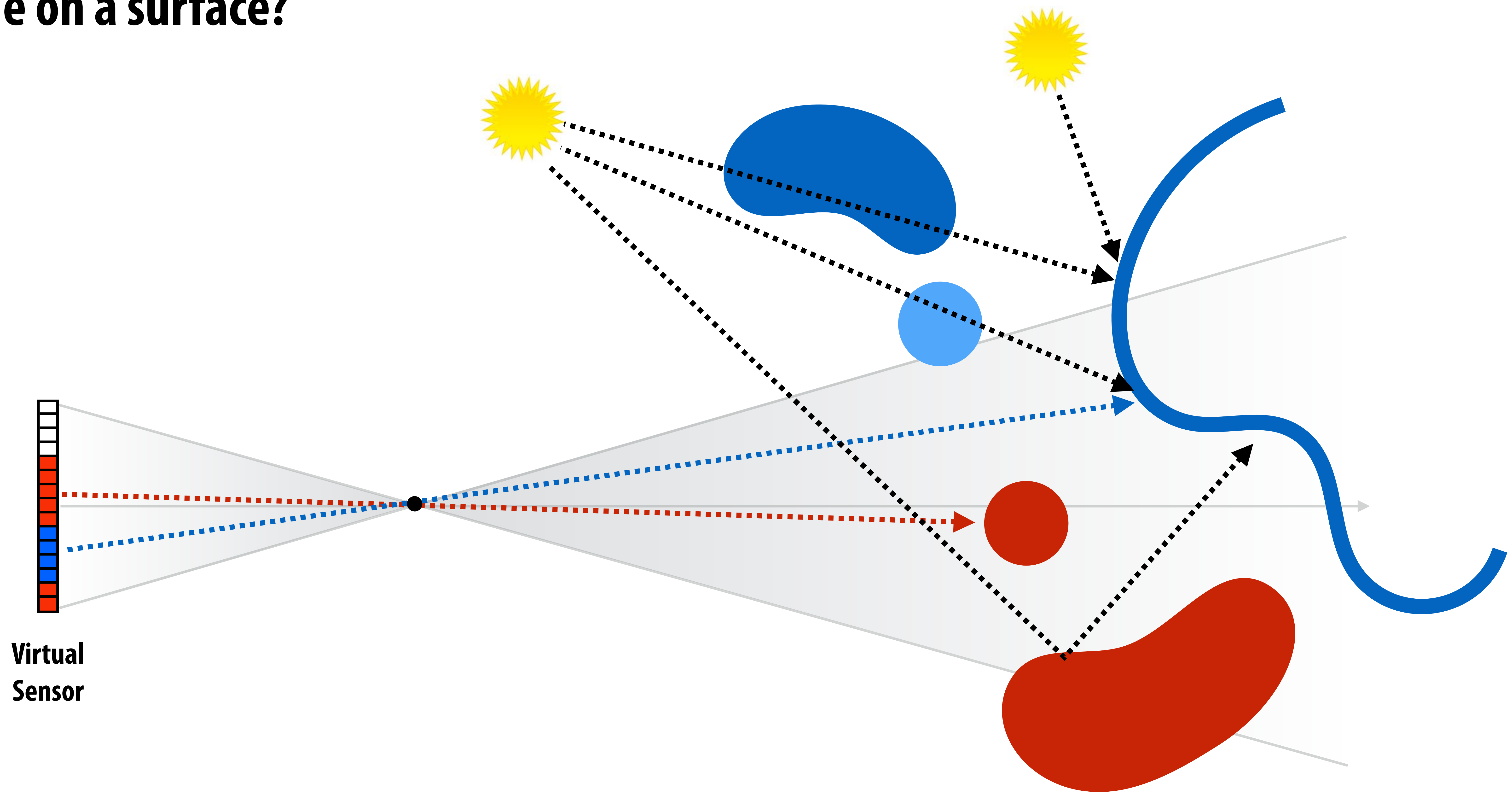


Generality of ray-scene queries

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?

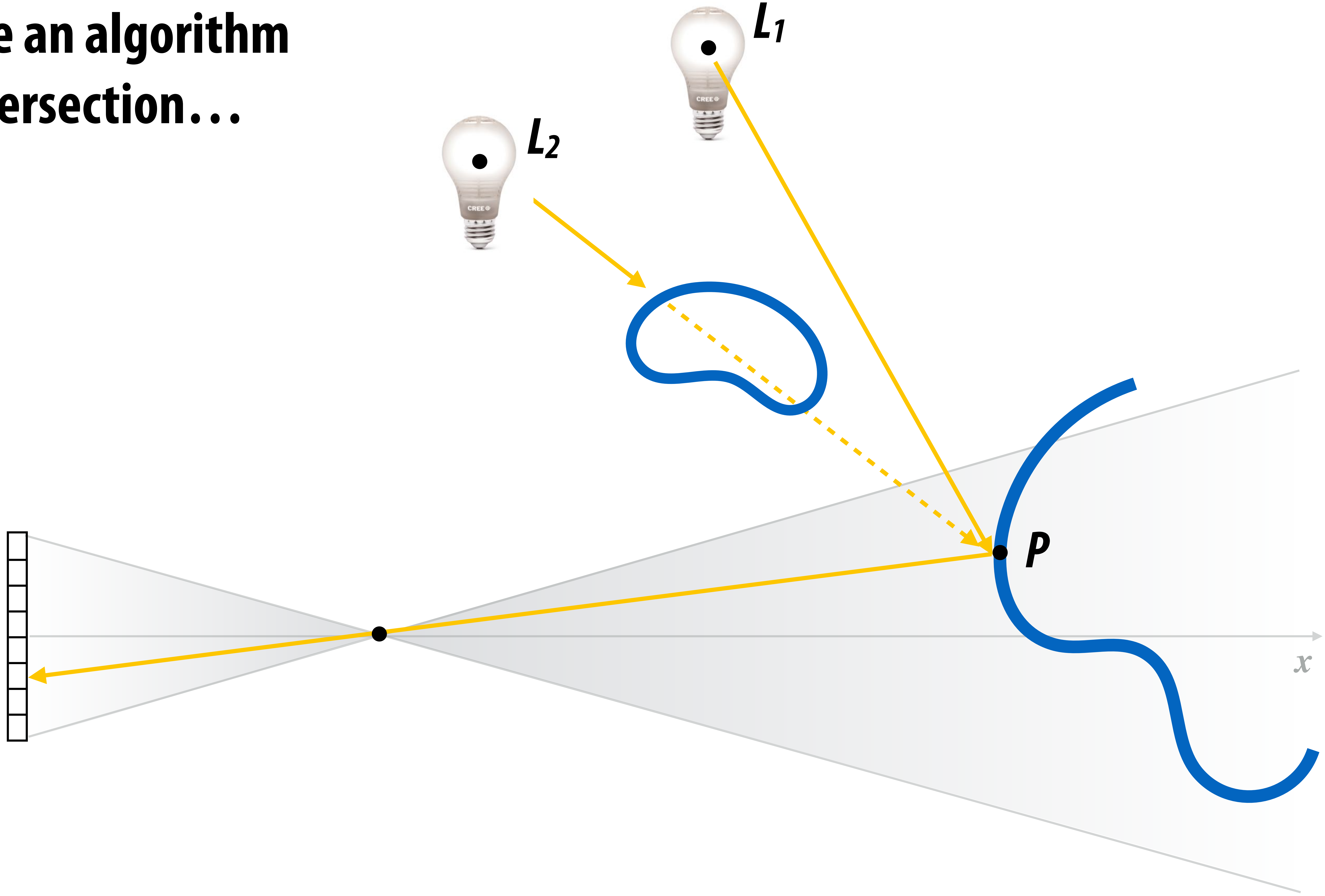


Shadows

Image credit: Grand Theft Auto V

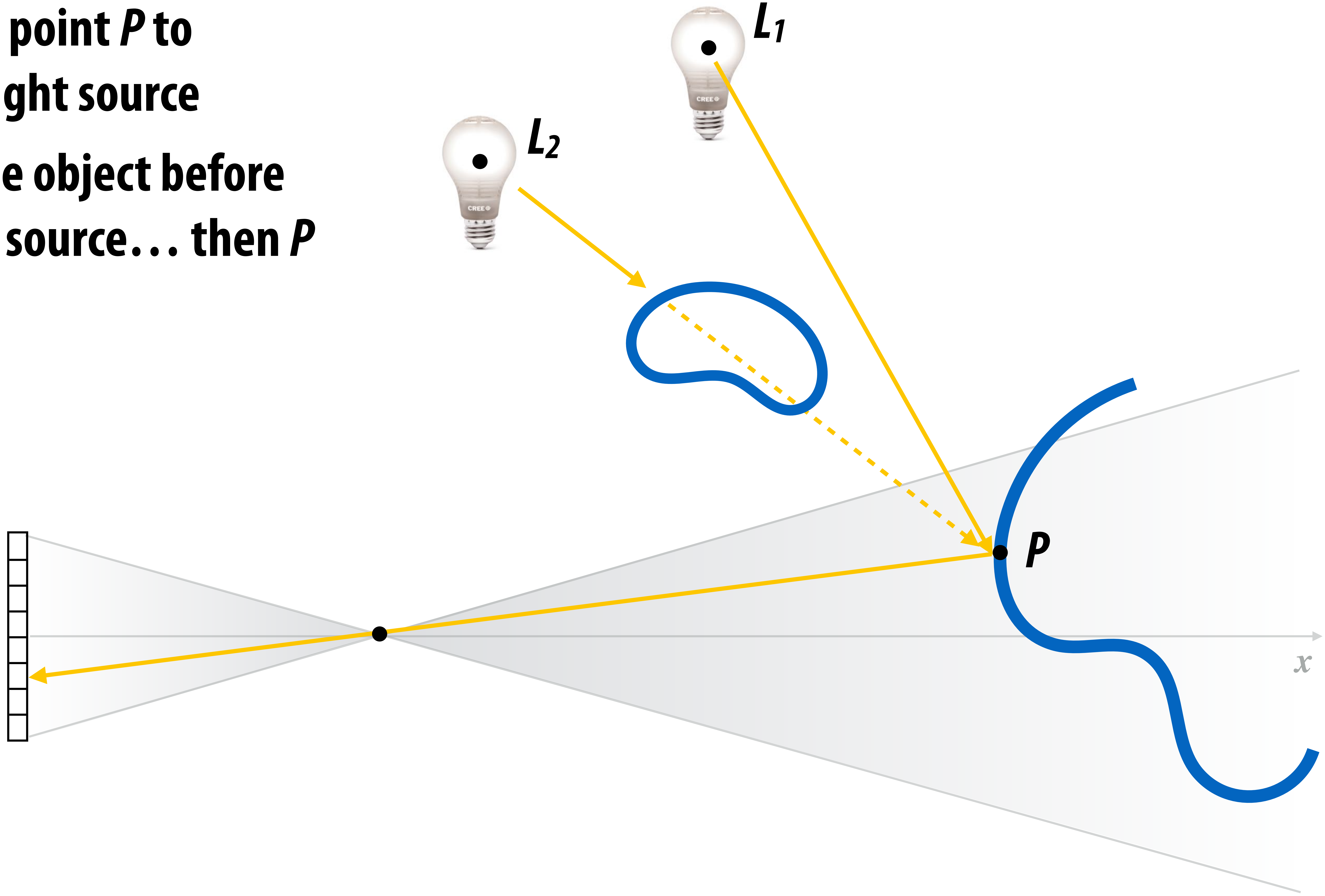
How to compute if a surface point is in shadow?

Assume you have an algorithm for ray-scene intersection…



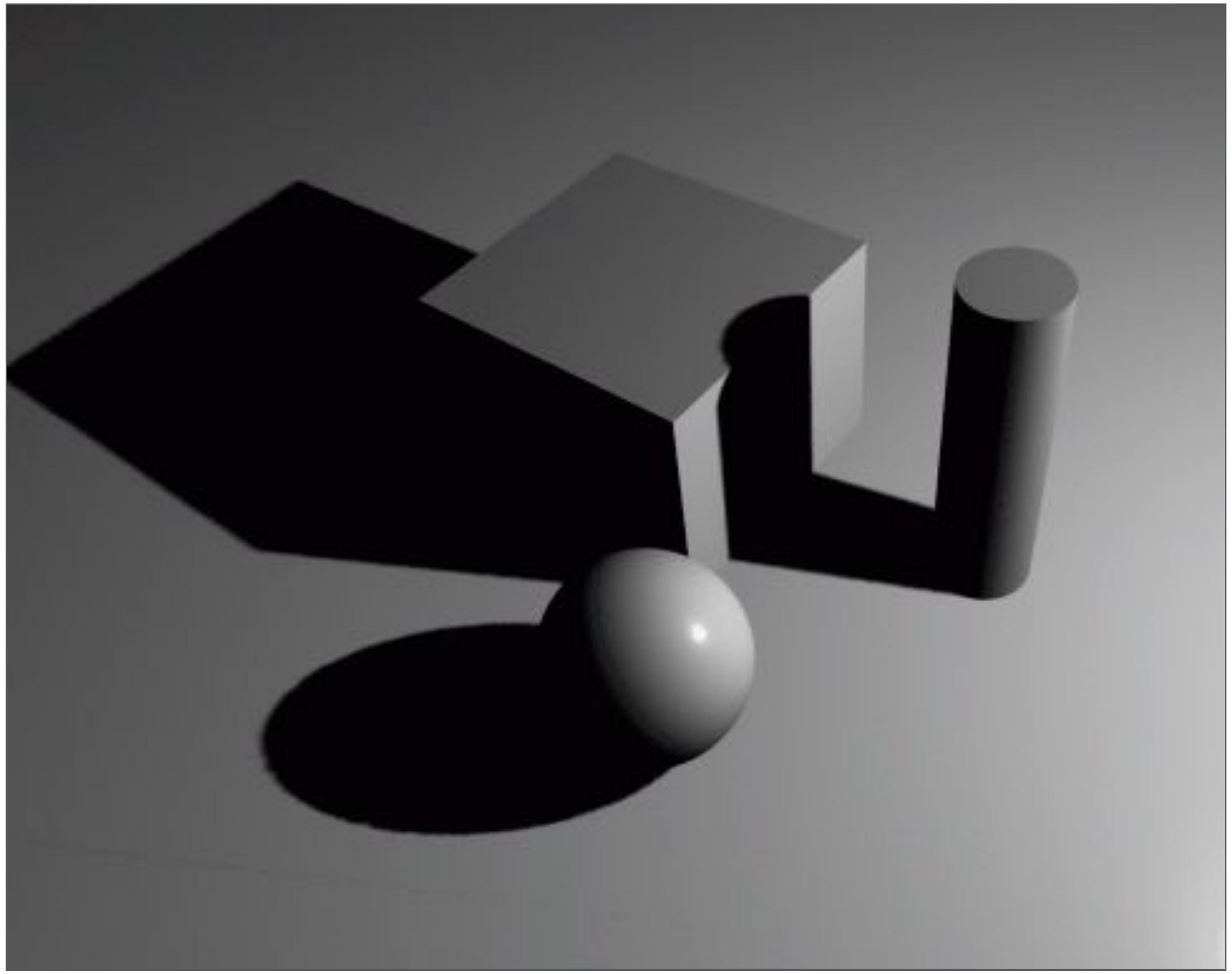
A simple shadow computation algorithm

- 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

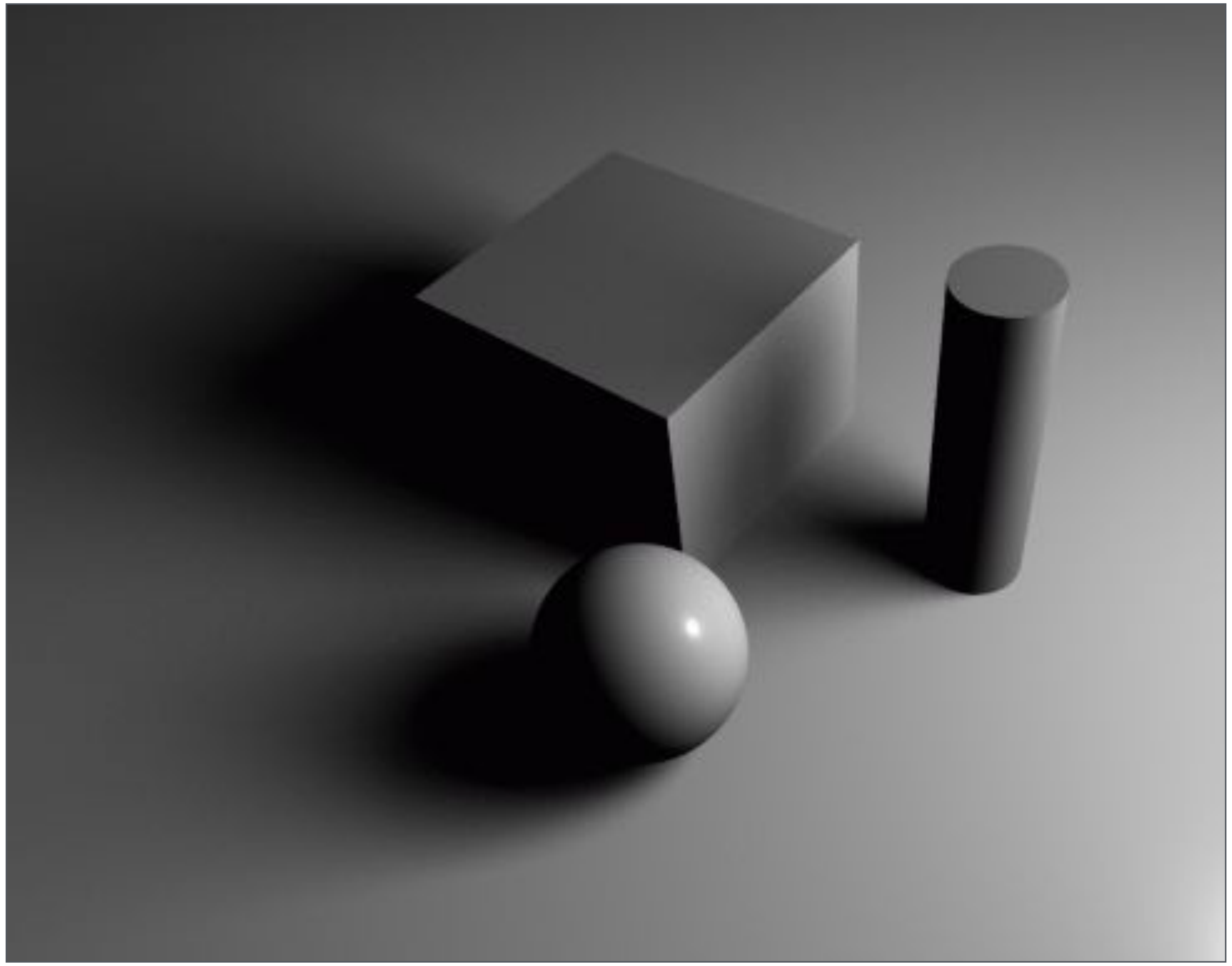


Scene with many light sources

Soft shadows



Hard shadows
(created by point light source)



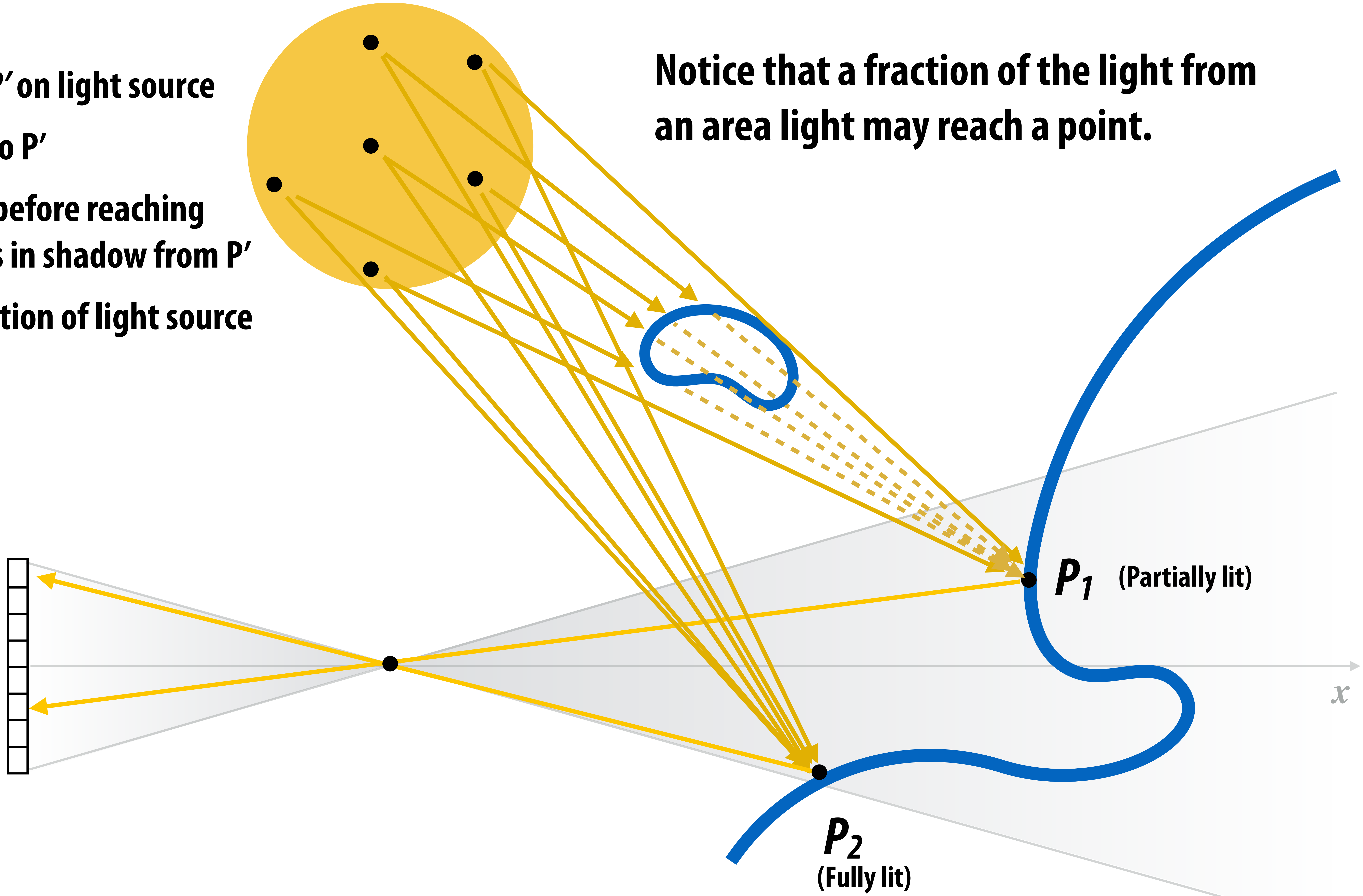
Soft shadows
(created by ???)

Image credit: Pixar

Soft shadow cast by an area light

- Based on ray tracing…
- Sample random point *P′* on light source
- Trace ray from point *P* to P′
- If ray hits scene object before reaching light source… then *P* is in shadow from P′
- Illumination at P is fraction of light source that is visible.

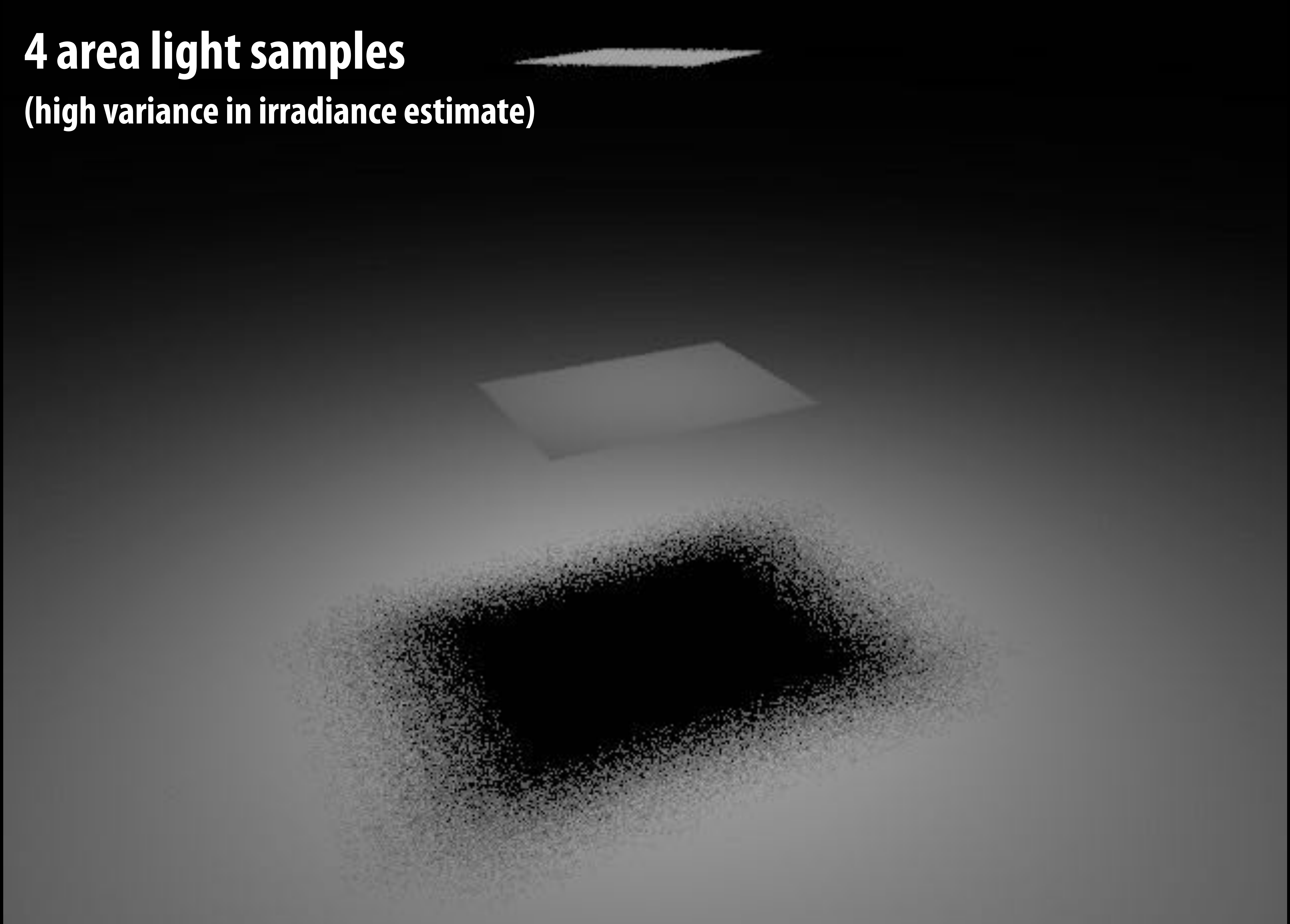
Notice that a fraction of the light from an area light may reach a point.



Implication: must trace many rays per pixel!

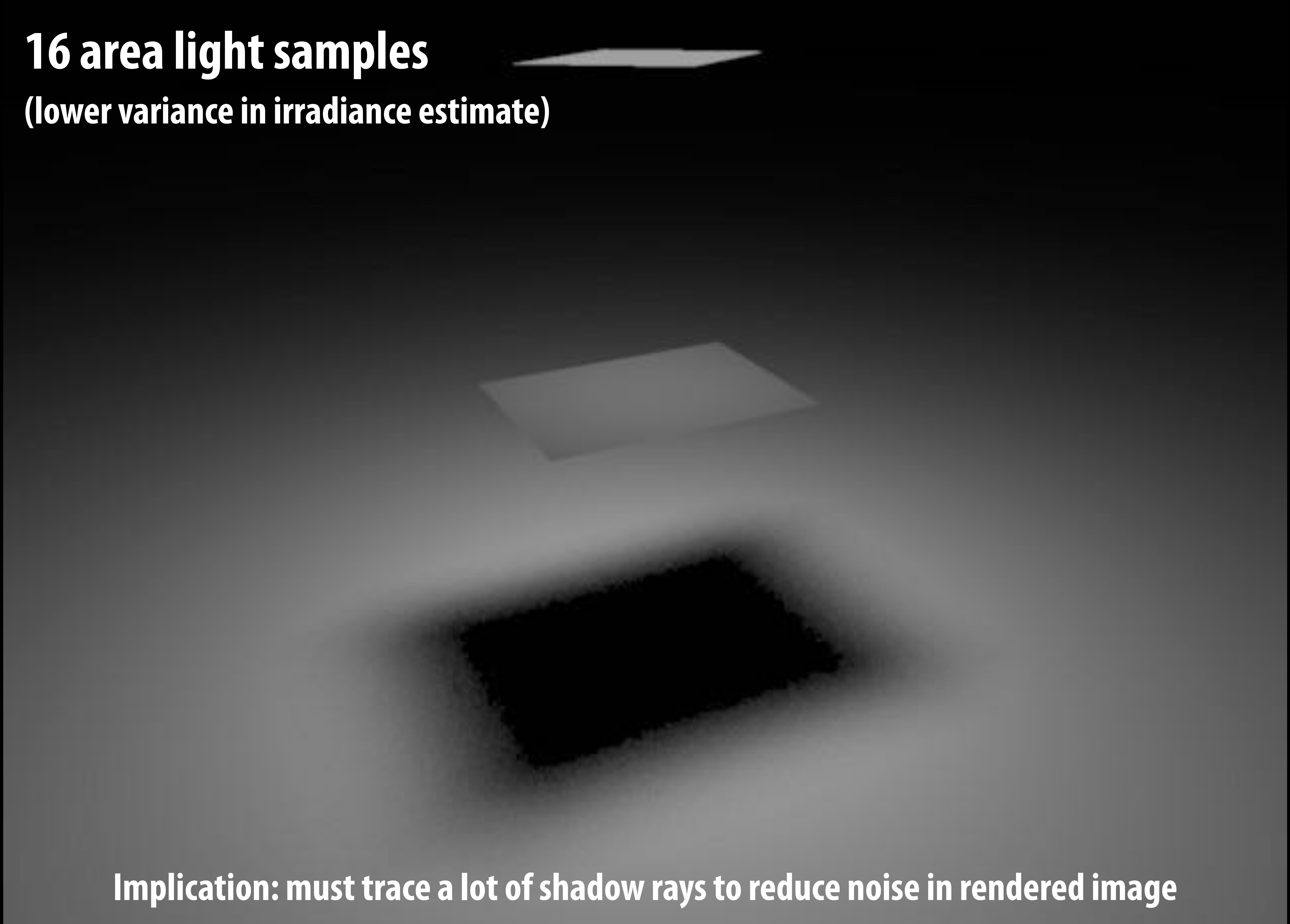
4 area light samples

(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

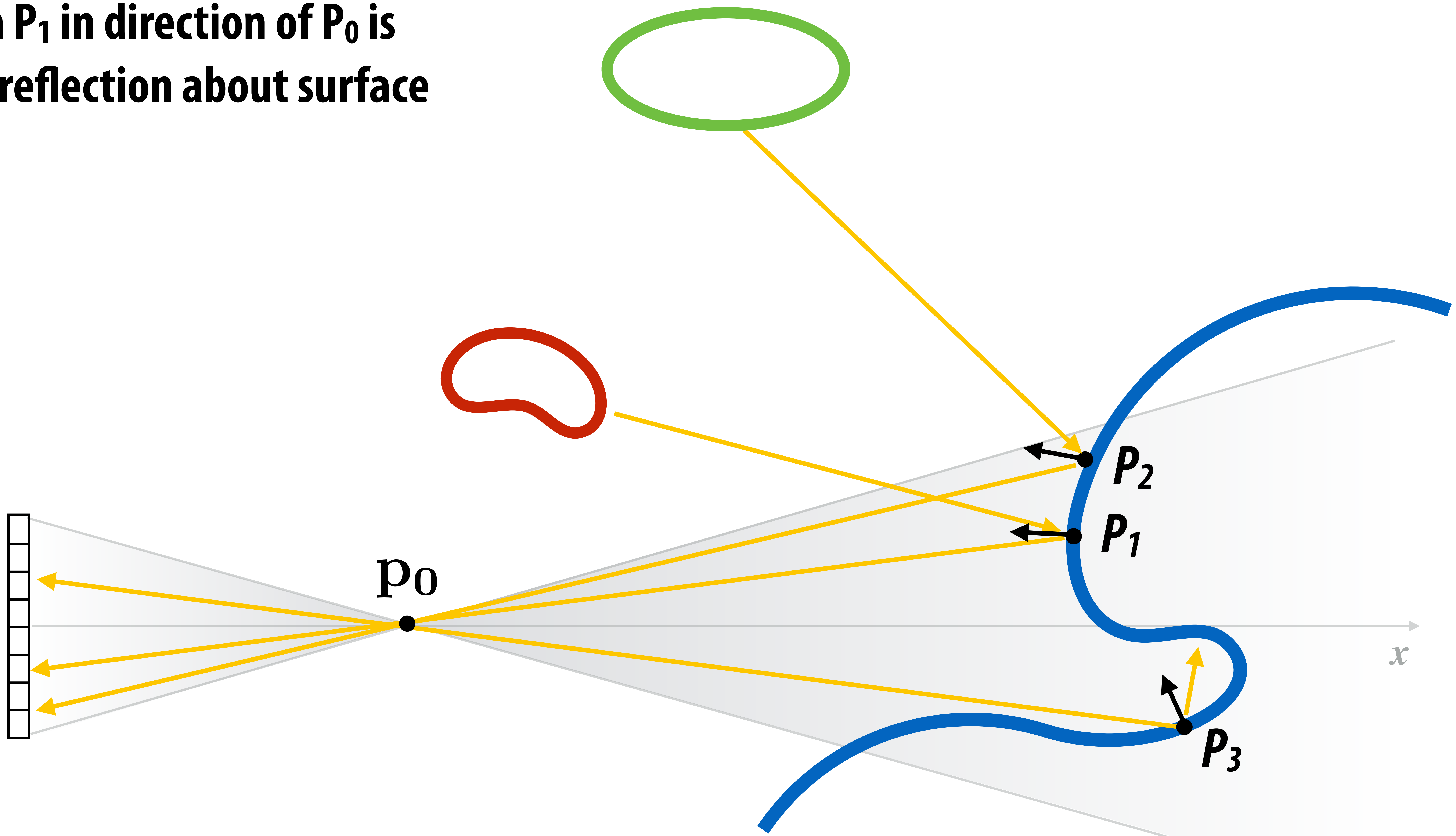
Reflections

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

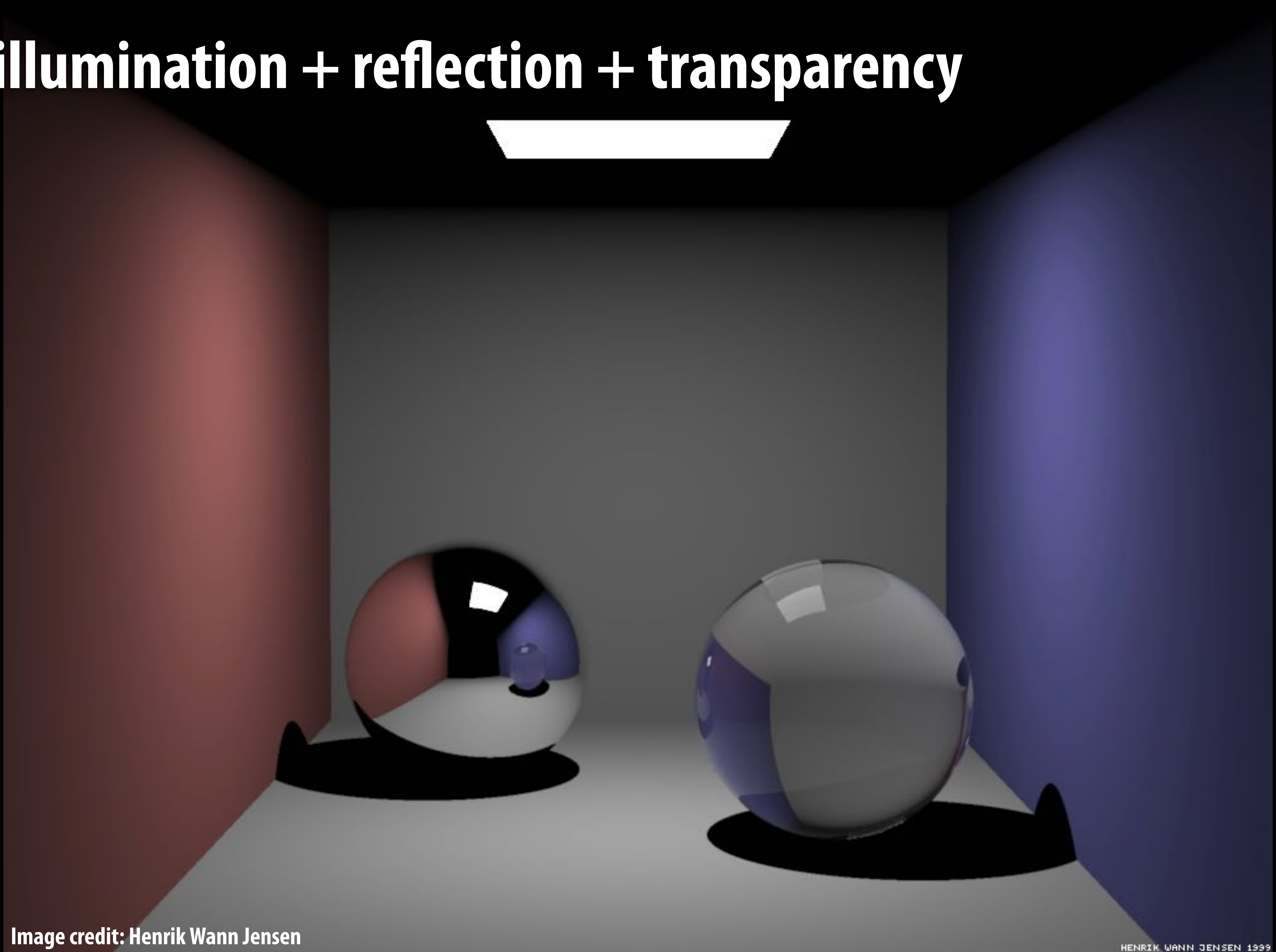


Image credit: Henrik Wann Jensen

Global illumination solution

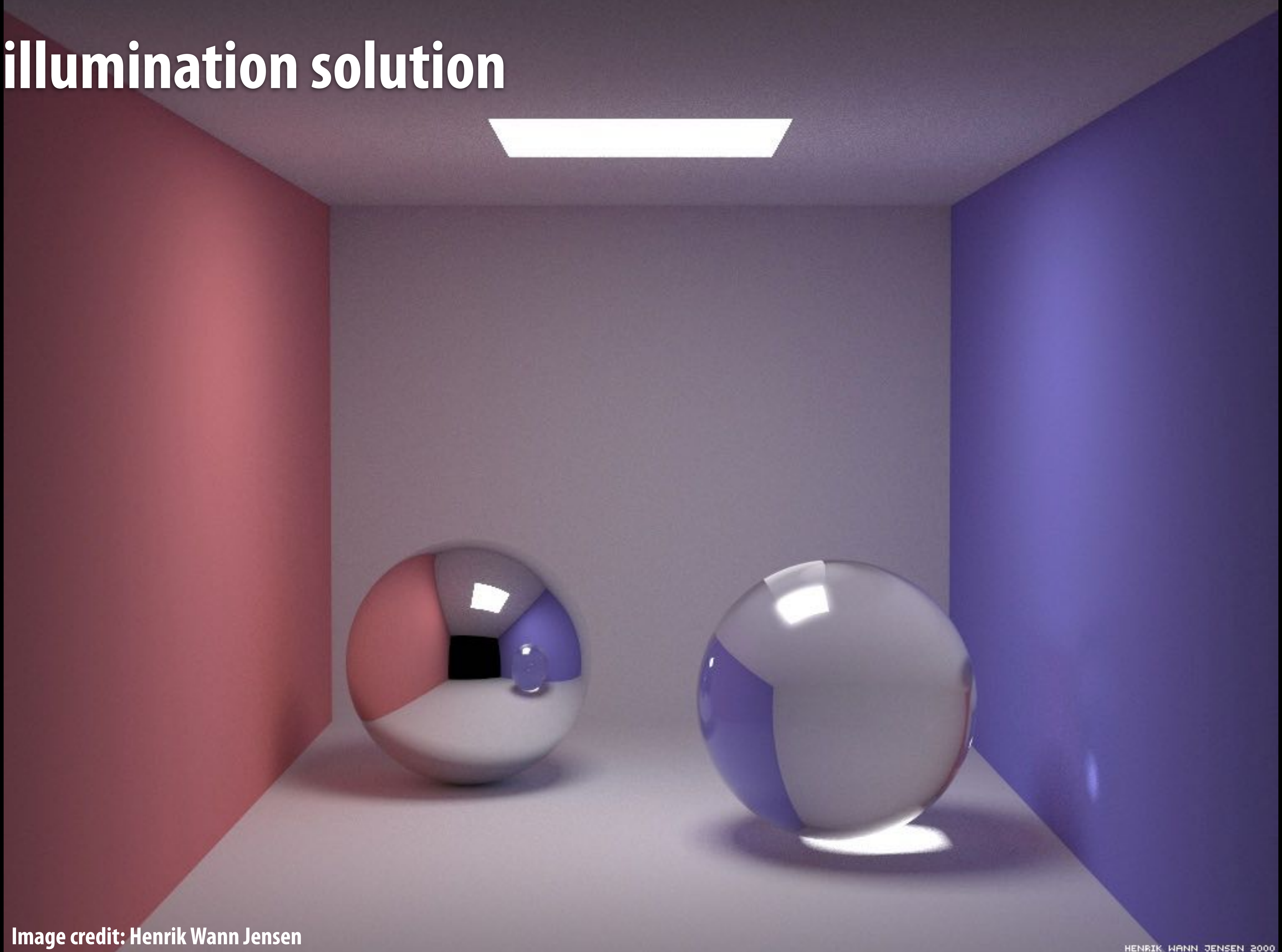
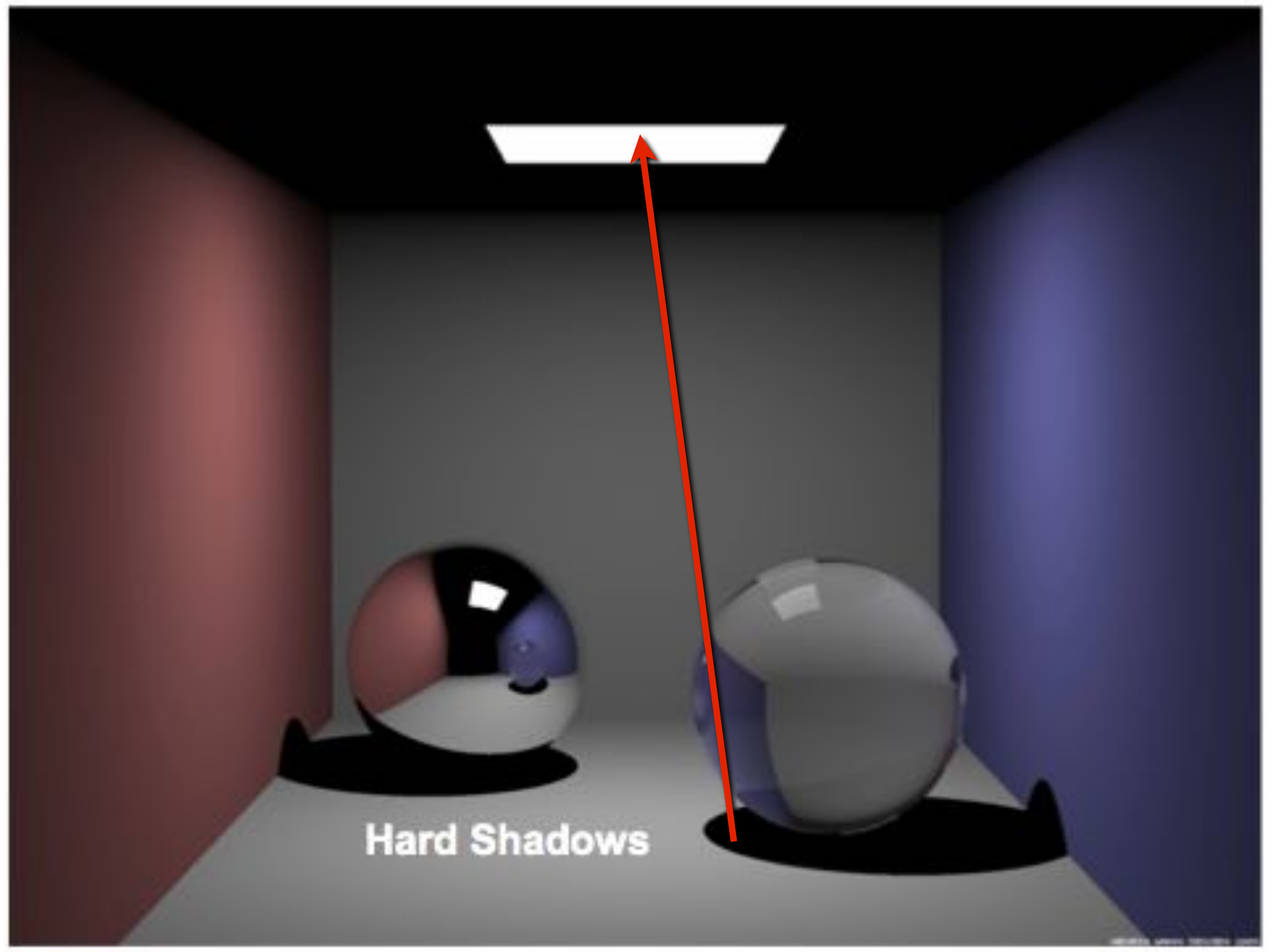
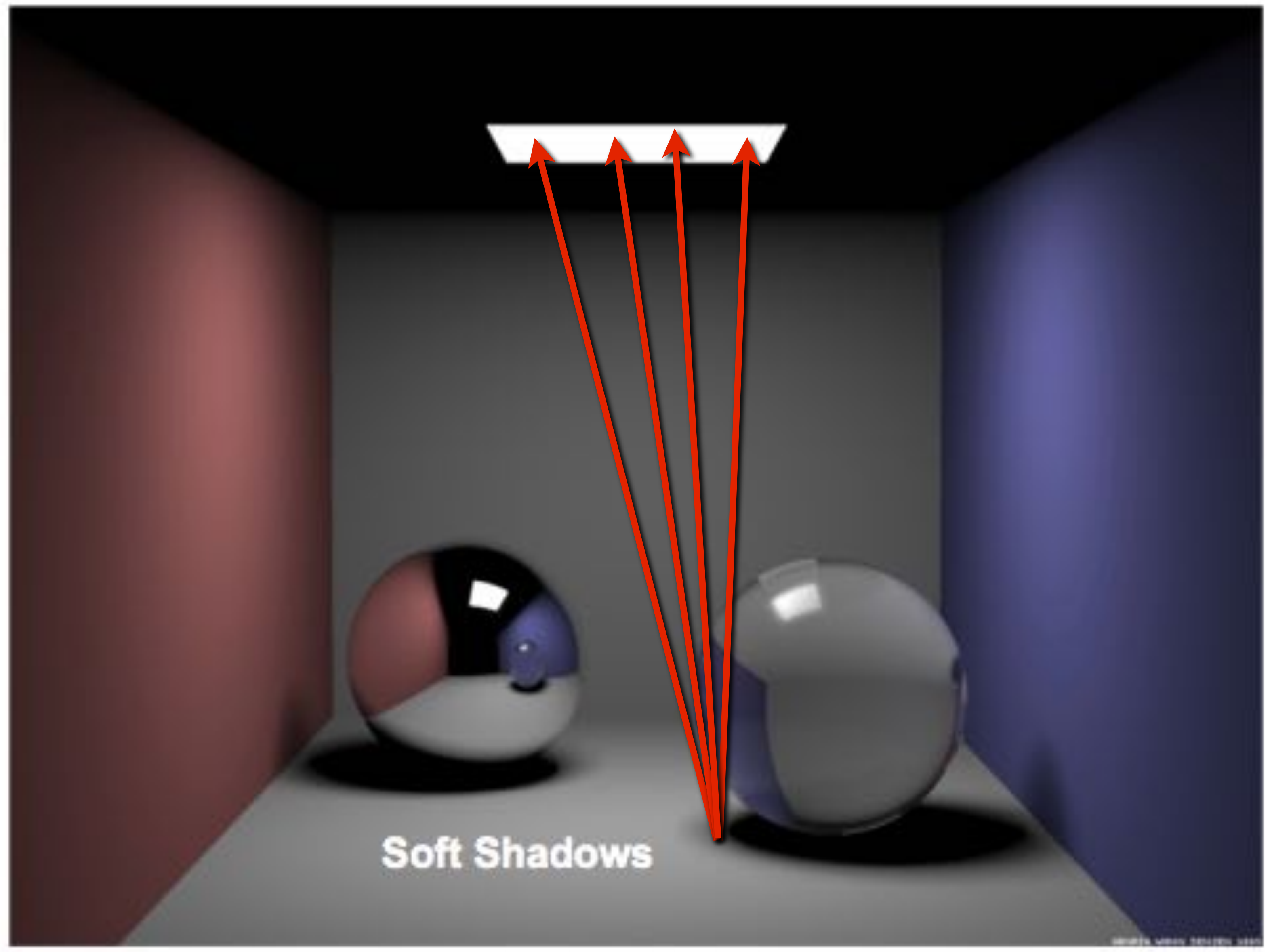
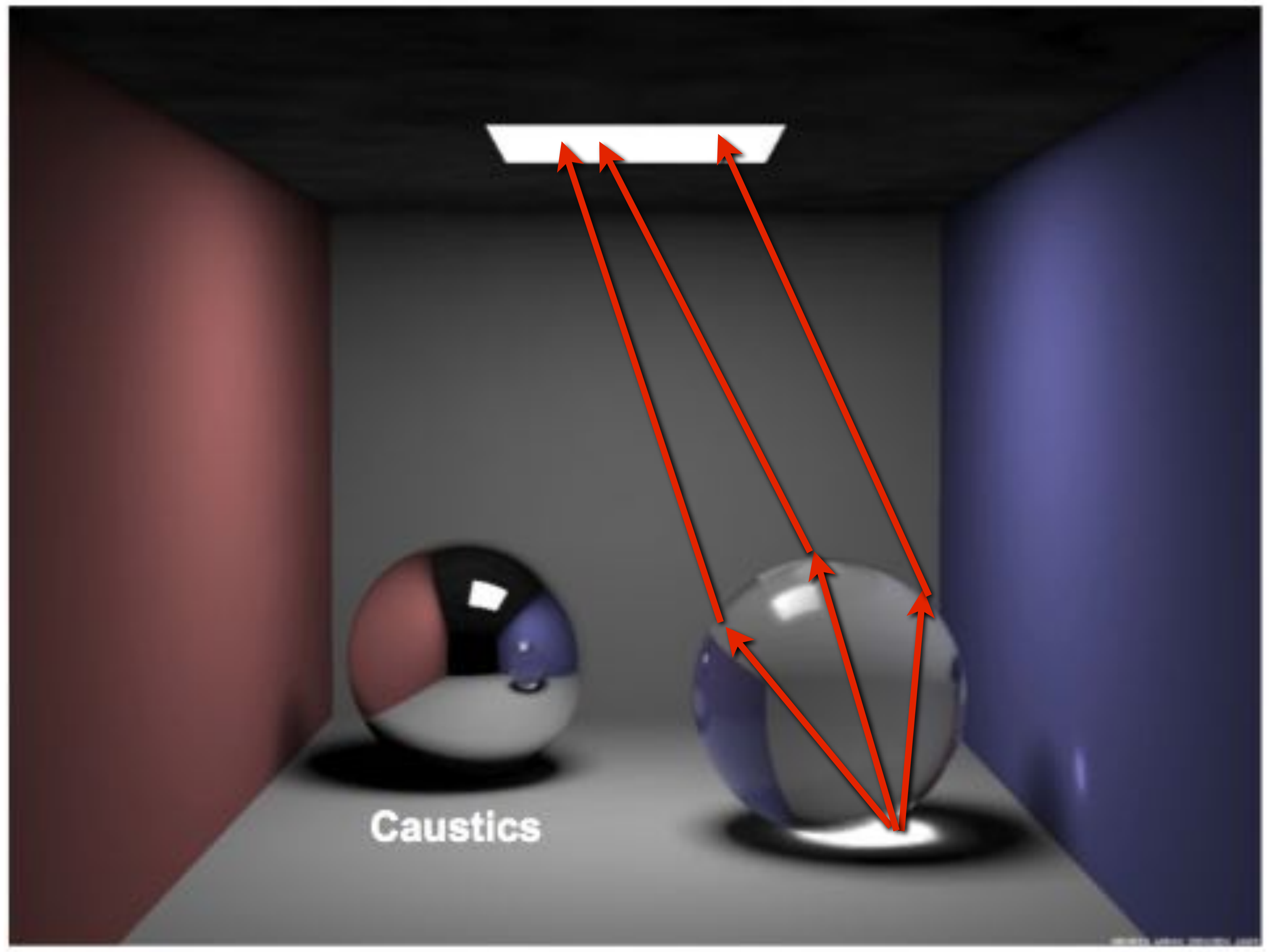


Image credit: Henrik Wann Jensen

Sampling light paths







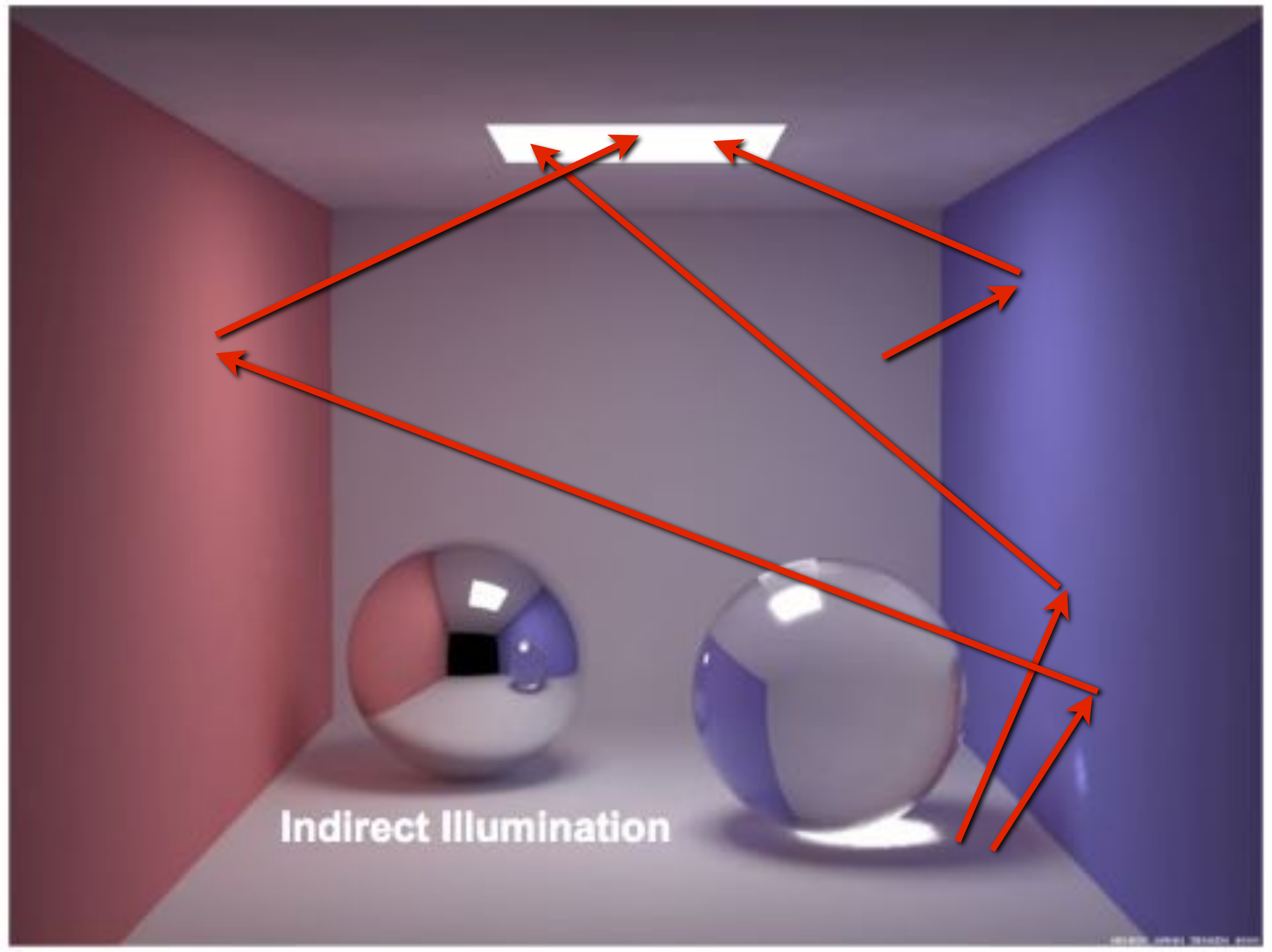
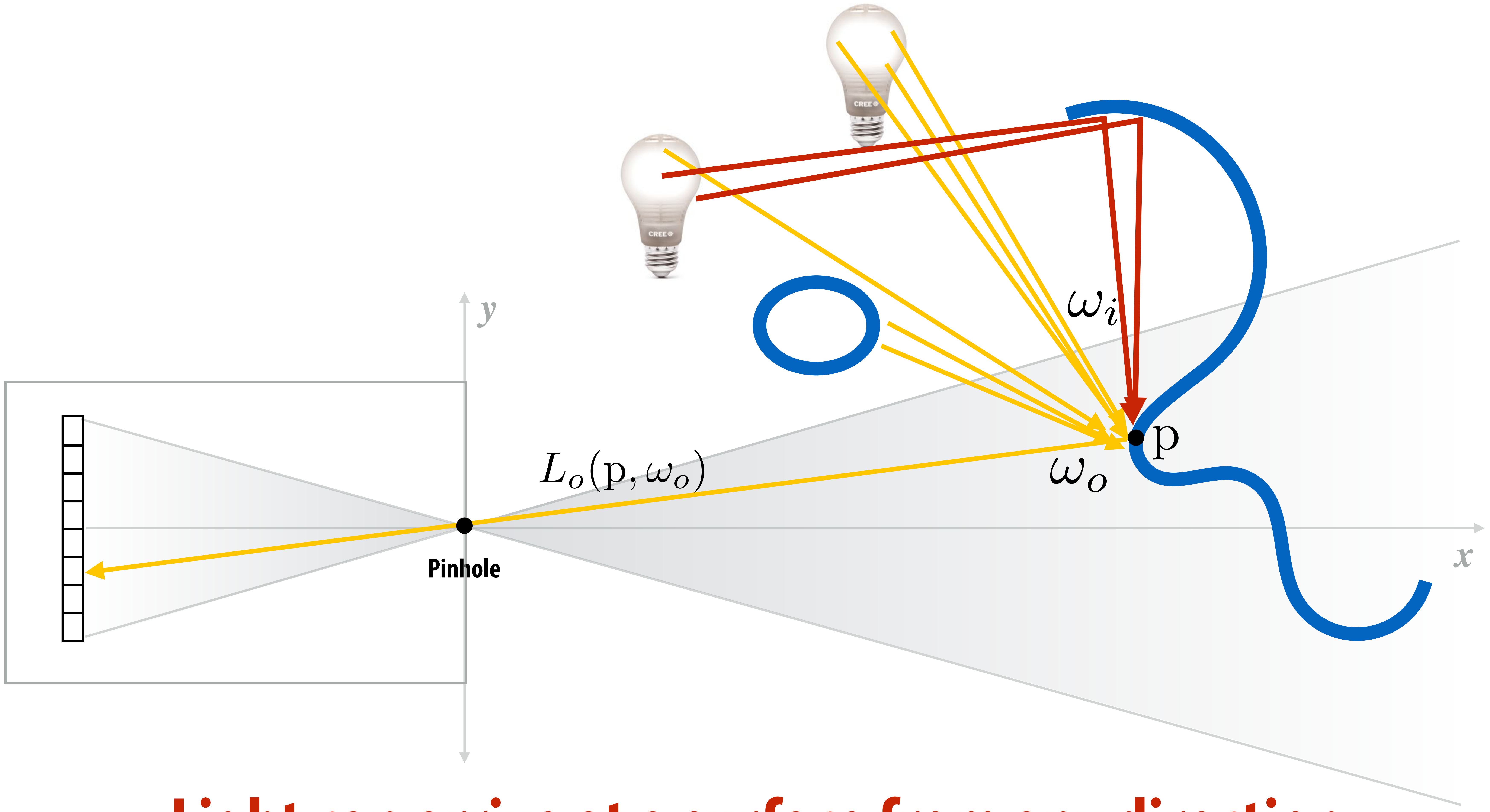


Image credit: Wann Jensen, Hanrahan

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

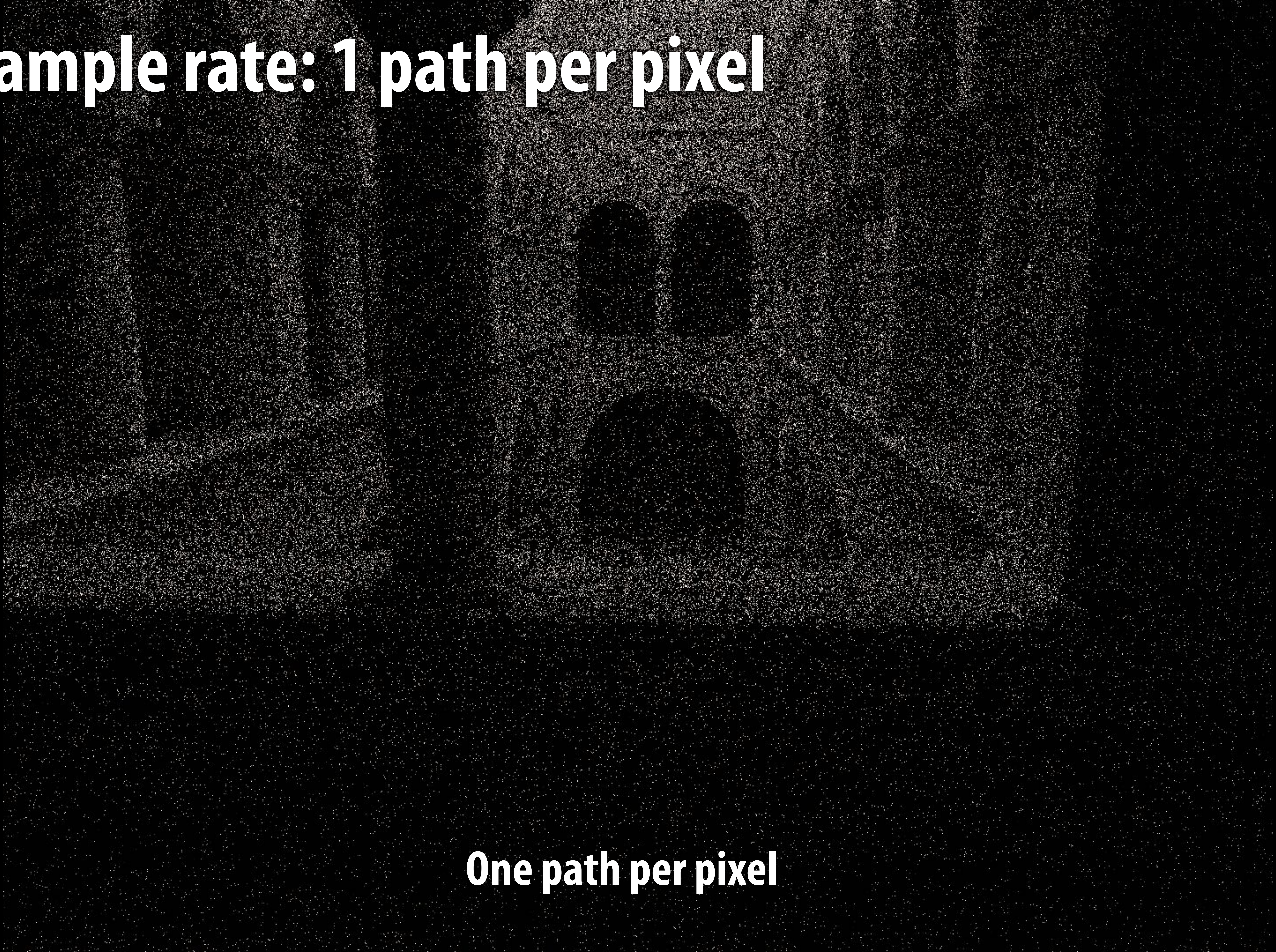
Direct illumination

Global Illumination

Importance of indirect illumination

1024 samples per pixel

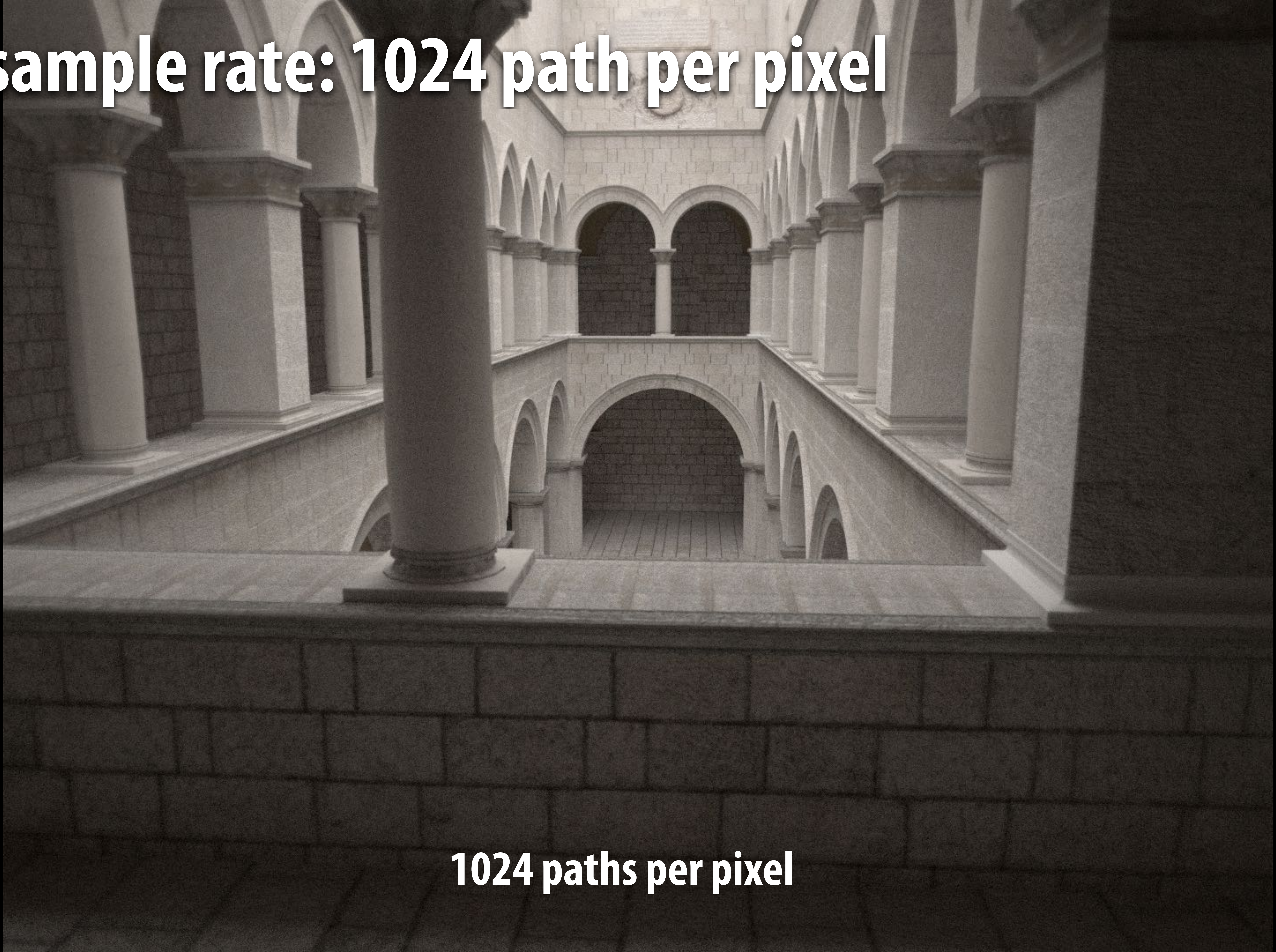
Low sample rate: 1 path per pixel



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

But wait, how to we efficiently perform ray-scene intersection?

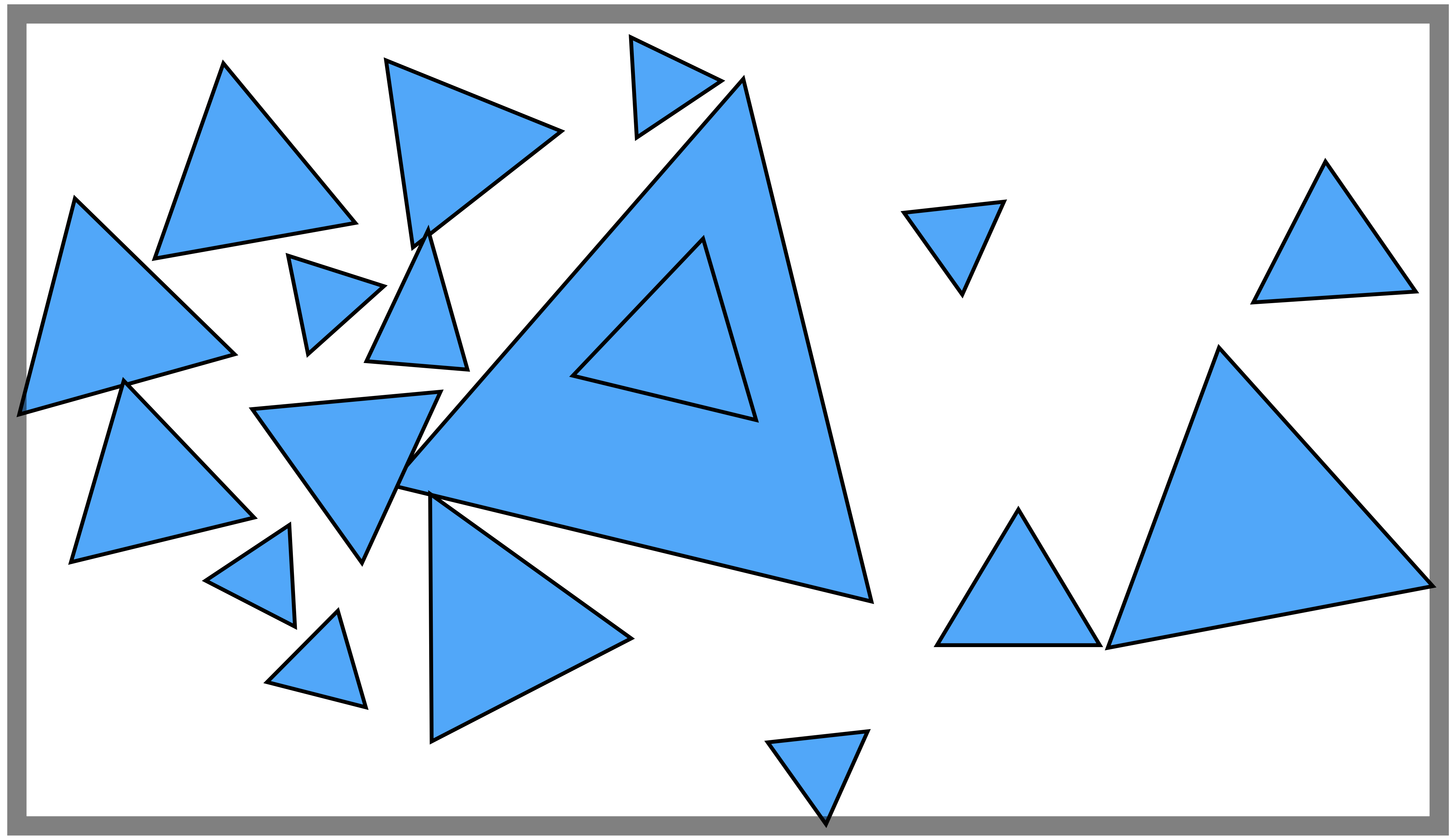
Disney Moana scene

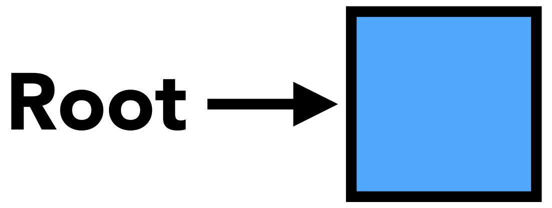
Released for rendering research purposes in 2018.

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

How to efficiently find the closest hit using BVH acceleration structures

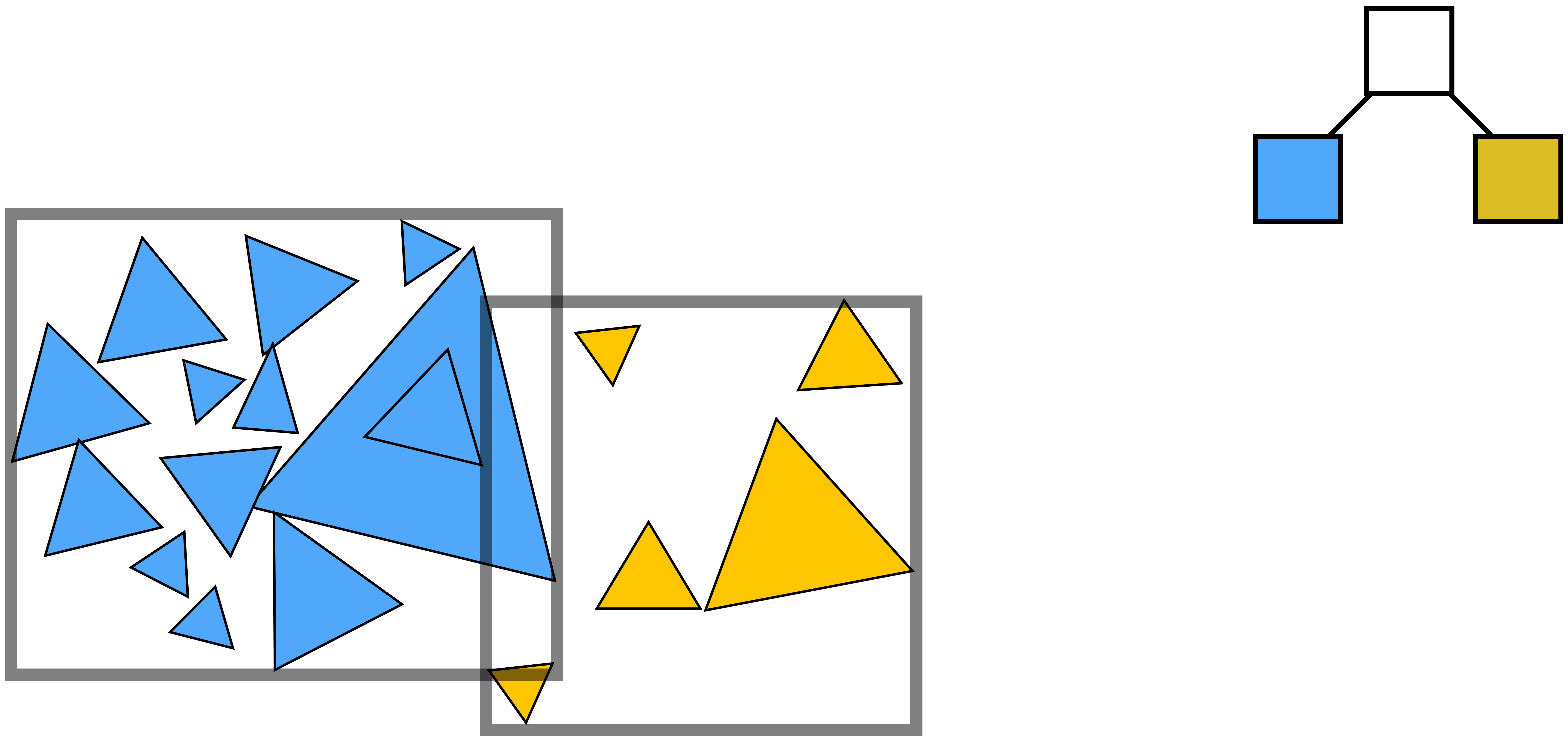
Bounding volume hierarchy (BVH)



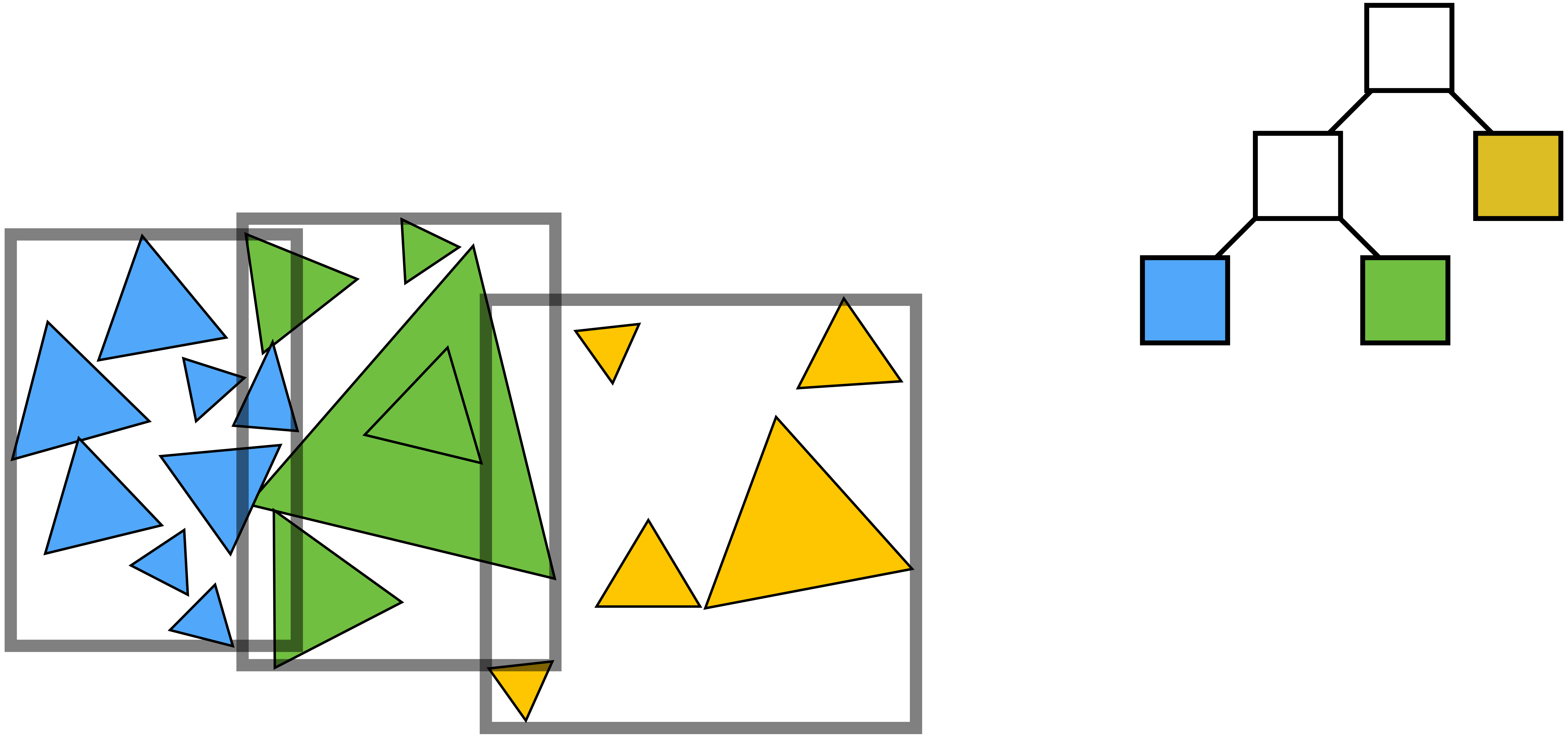


Bounding volume hierarchy (BVH)

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

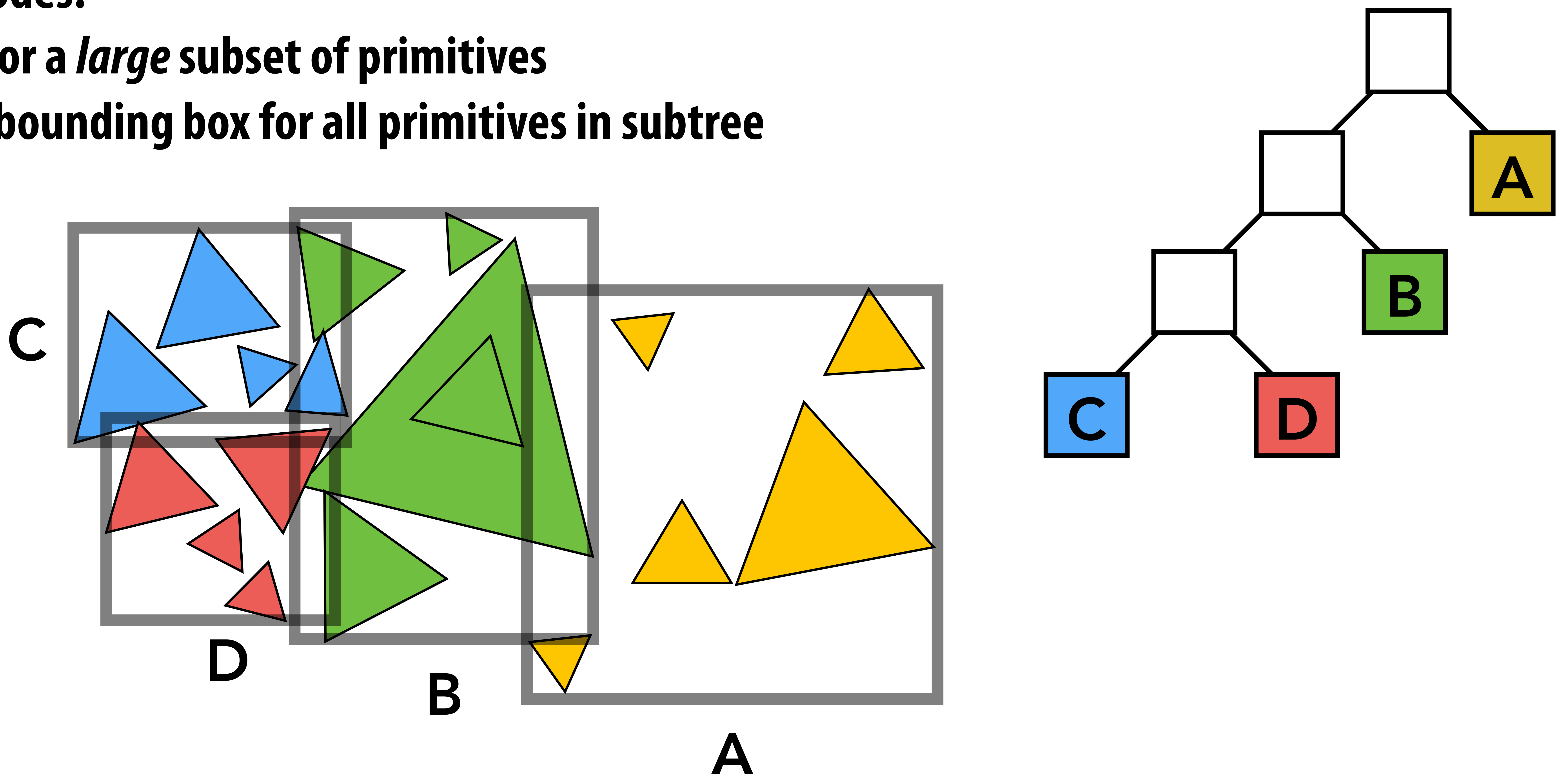


Bounding volume hierarchy (BVH)



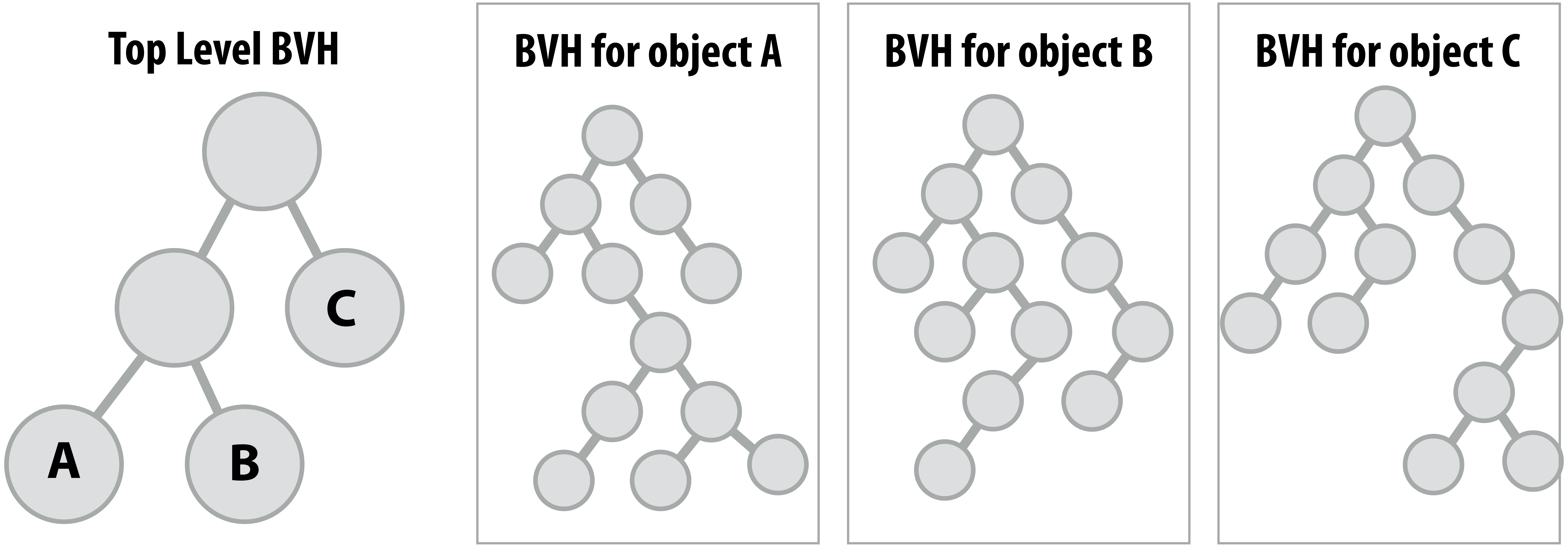
Bounding volume hierarchy (BVH)

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



Two-level BVHs

- Many scene objects do not move from frame-to-frame, or only move rigidly
- Approach: two-level BVH: build a BVH over per-object BVHs
  - Only rebuild this top level BVH each frame as objects move



Contains hundreds
of scene objects

Each per-object BVH might contains tens's of thousands of triangles.
If object's geometry does not undergo relative change
(other than rotation/translation in world)
the BVH can be built once and remain applicable.

SPMD ray tracing (GPU-style)

Each CUDA thread carries out processing for one ray.

SIMD parallelism comes from executing multiple threads in a WARP

```
stack<BVHNode> nodesToVisit;

if ray hits root.bbox:
  nodesToVisit.push(root);

while (nodesToVisit is not empty):

   // ray is “traversing” interior nodes
   while (not reached leaf node)
     node = nodesToVisit.pop(); // pop stack
     Perform ray-box tests for node.left.bbox and node.right.bbox
     if (ray hits both children):
        nodesToVisit.push(farther of left and right children)
        nodesToVisit.push(closer of left and right children)
     else if (ray only hits left child):
        nodesToVisit.push(left child);
     else if (ray only hits right child)
        nodesToVisit.push(right child);

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

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.

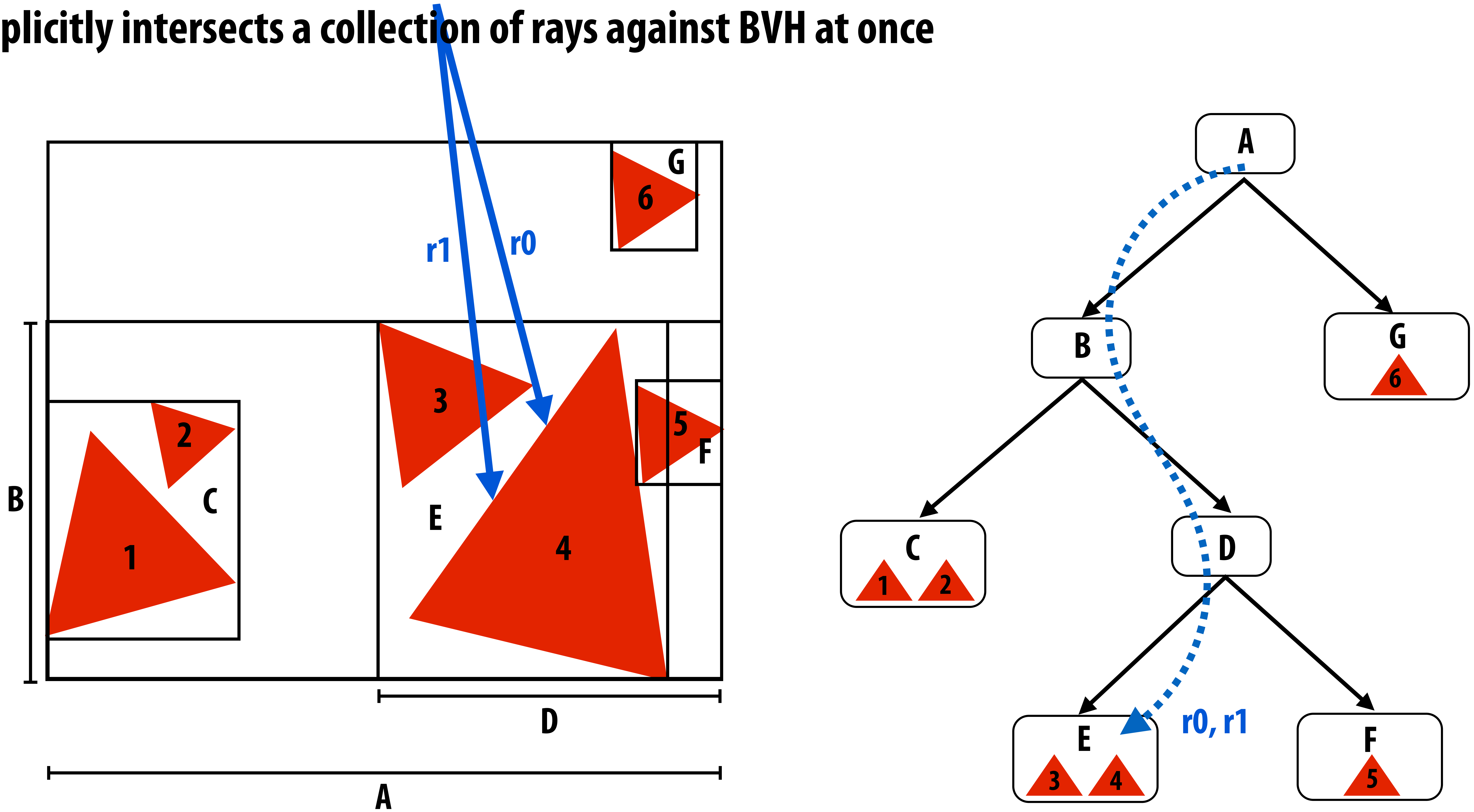
Takeaway:
Ray-BVH traversal generates unpredictable (data-dependent) access to an irregular data structure

Understanding ray coherence during BVH traversal

Ray traversal "coherence"

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

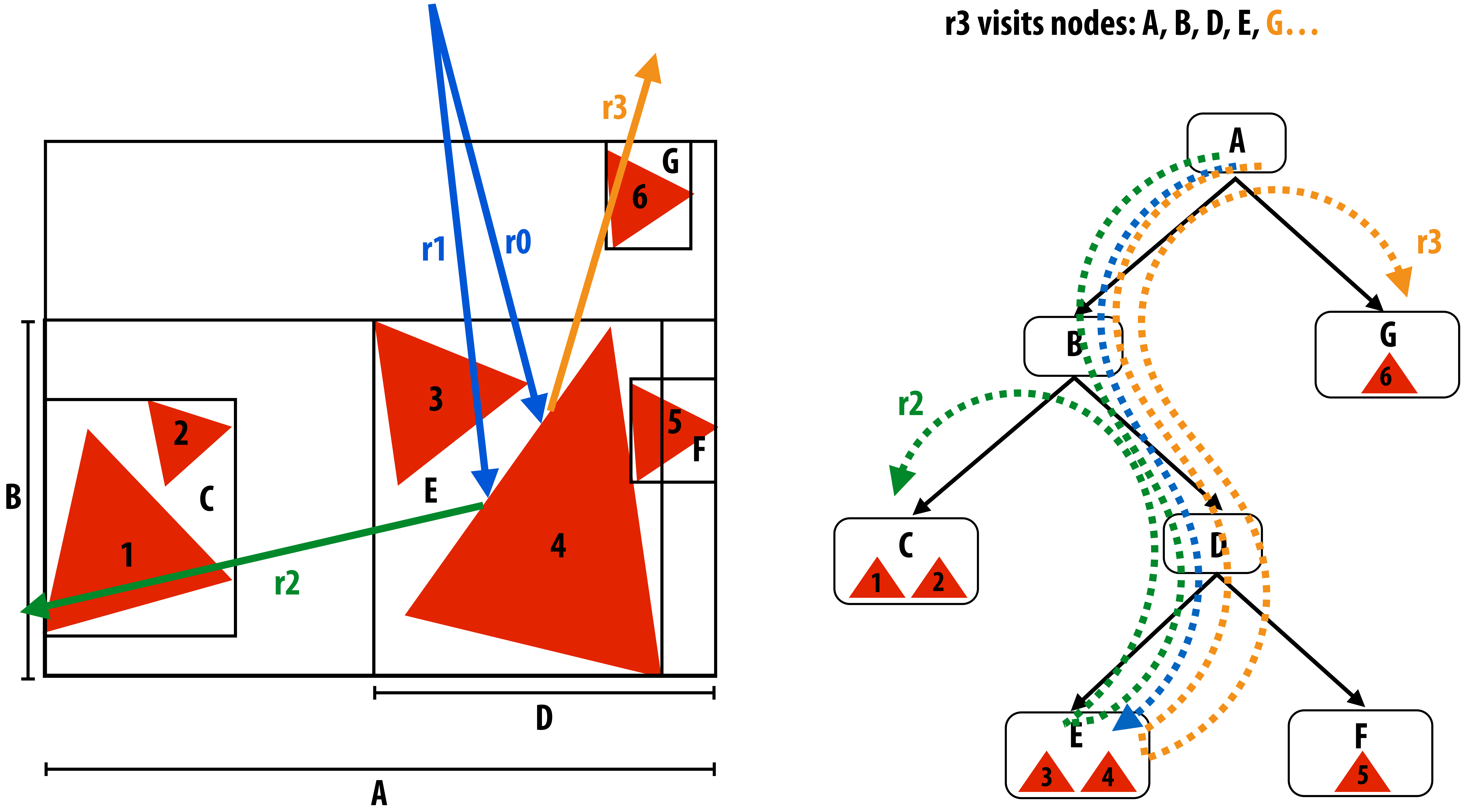
Program explicitly intersects a collection of rays against BVH at once



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

Ray traversal "divergence"

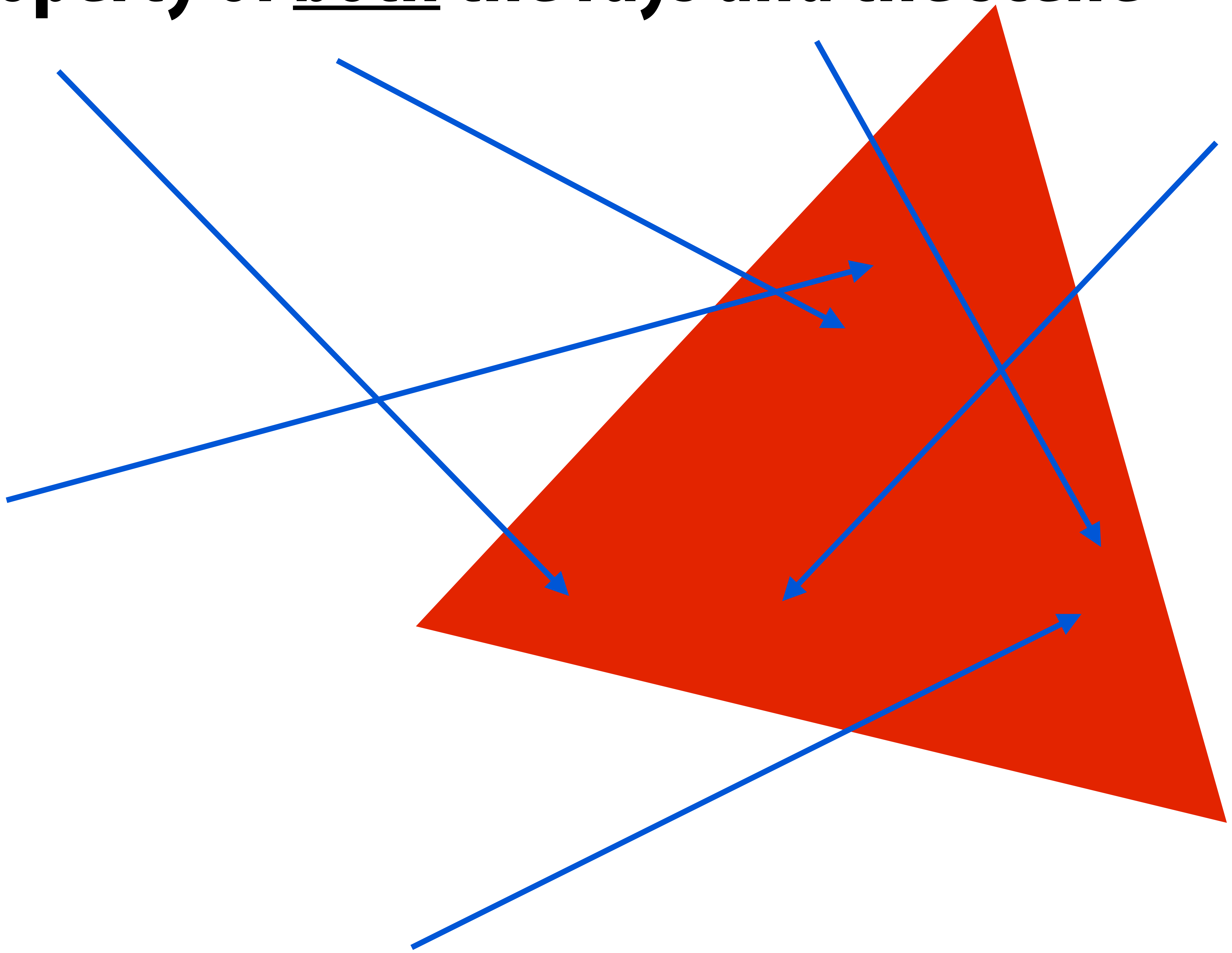
r0 visits nodes: A, B, D, E…
r1 visits nodes: A, B, D, E…
r2 visits nodes: A, B, D, E, C…
r3 visits nodes: A, B, D, E, G…



R2 and R3 require different BVH nodes and triangles

Incoherent rays

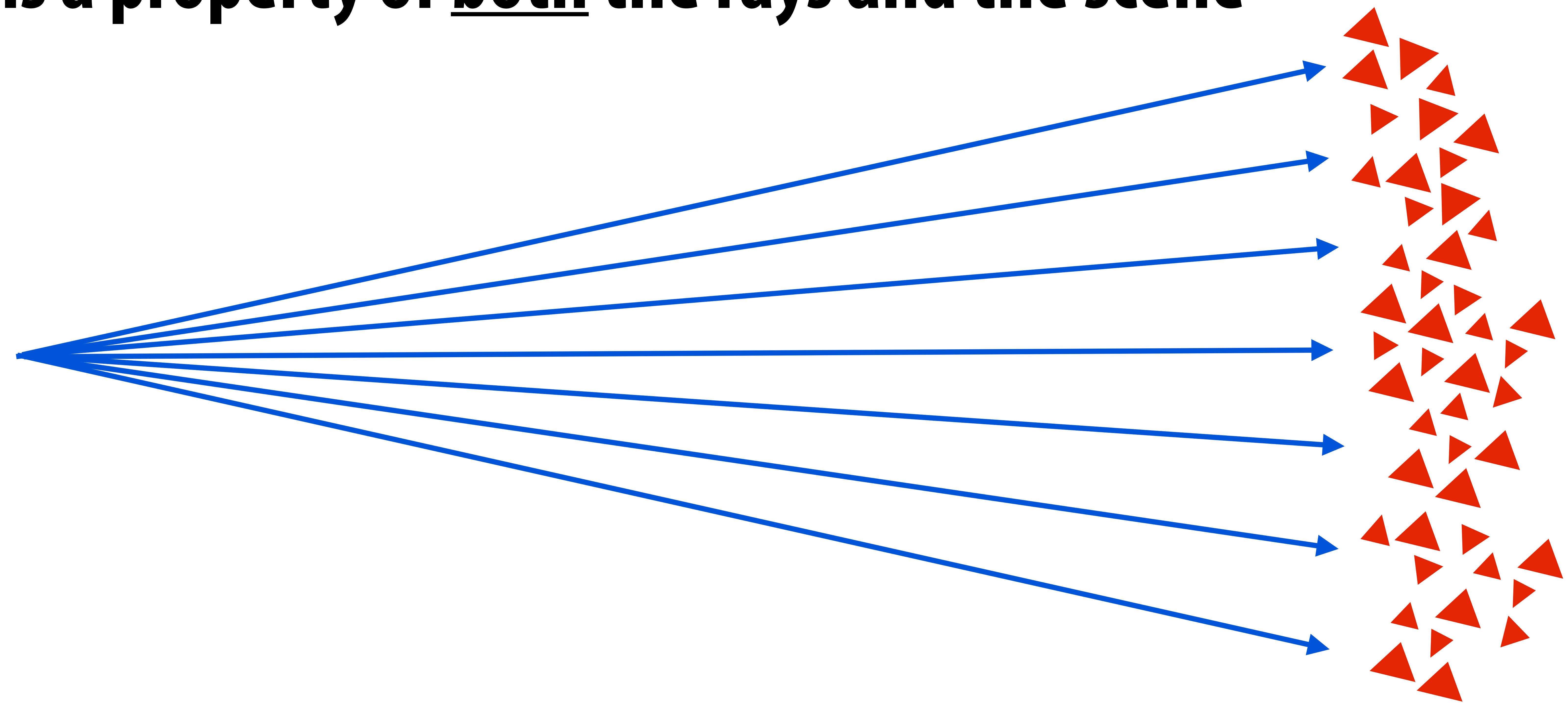
Incoherence is a property of both the rays and the scene



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

Incoherent rays

Incoherence is a property of both the rays and the scene



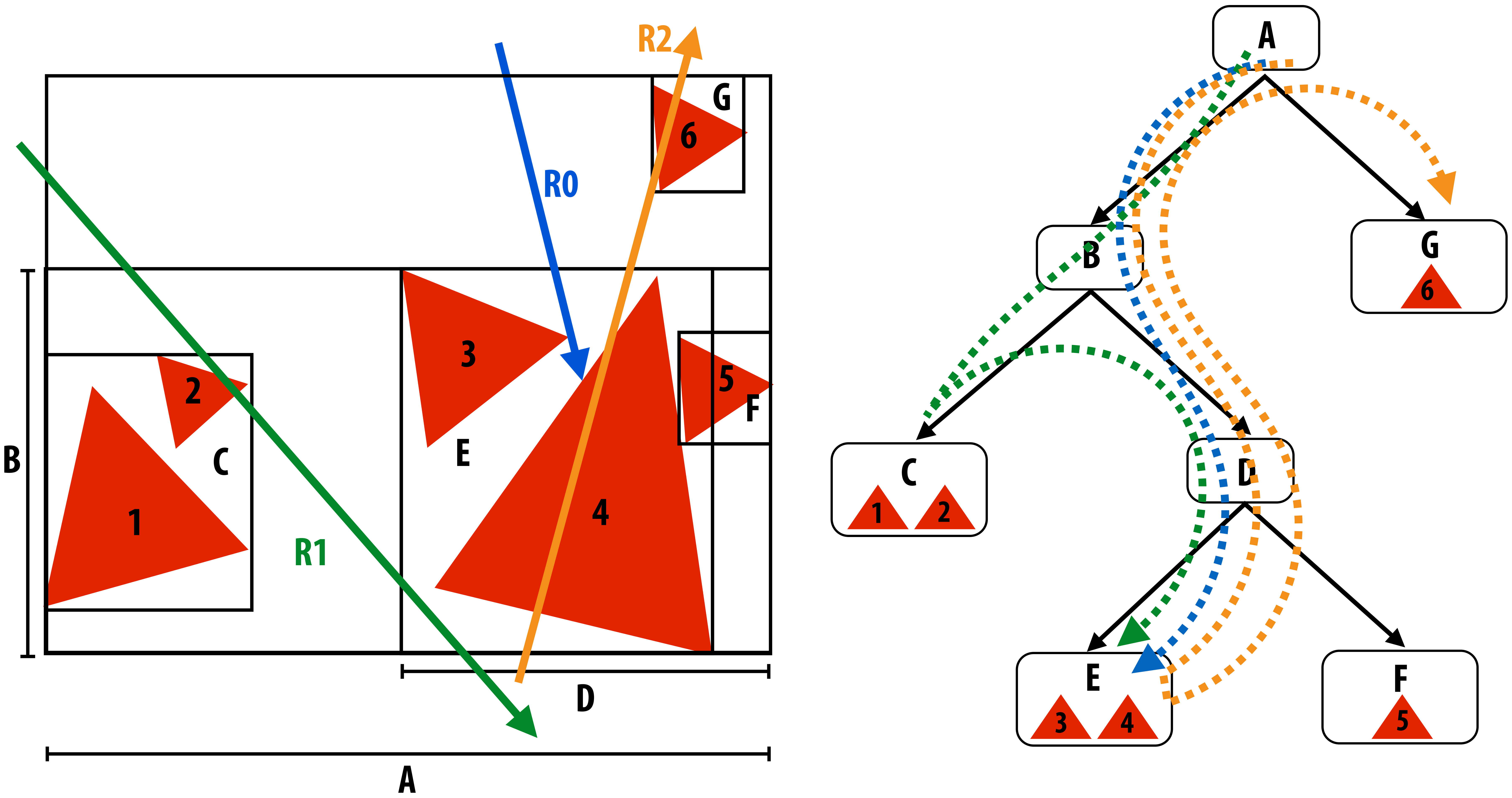
Similarly oriented rays from the same point become “incoherent” with respect to lower nodes in the BVH if a scene is overly detailed

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

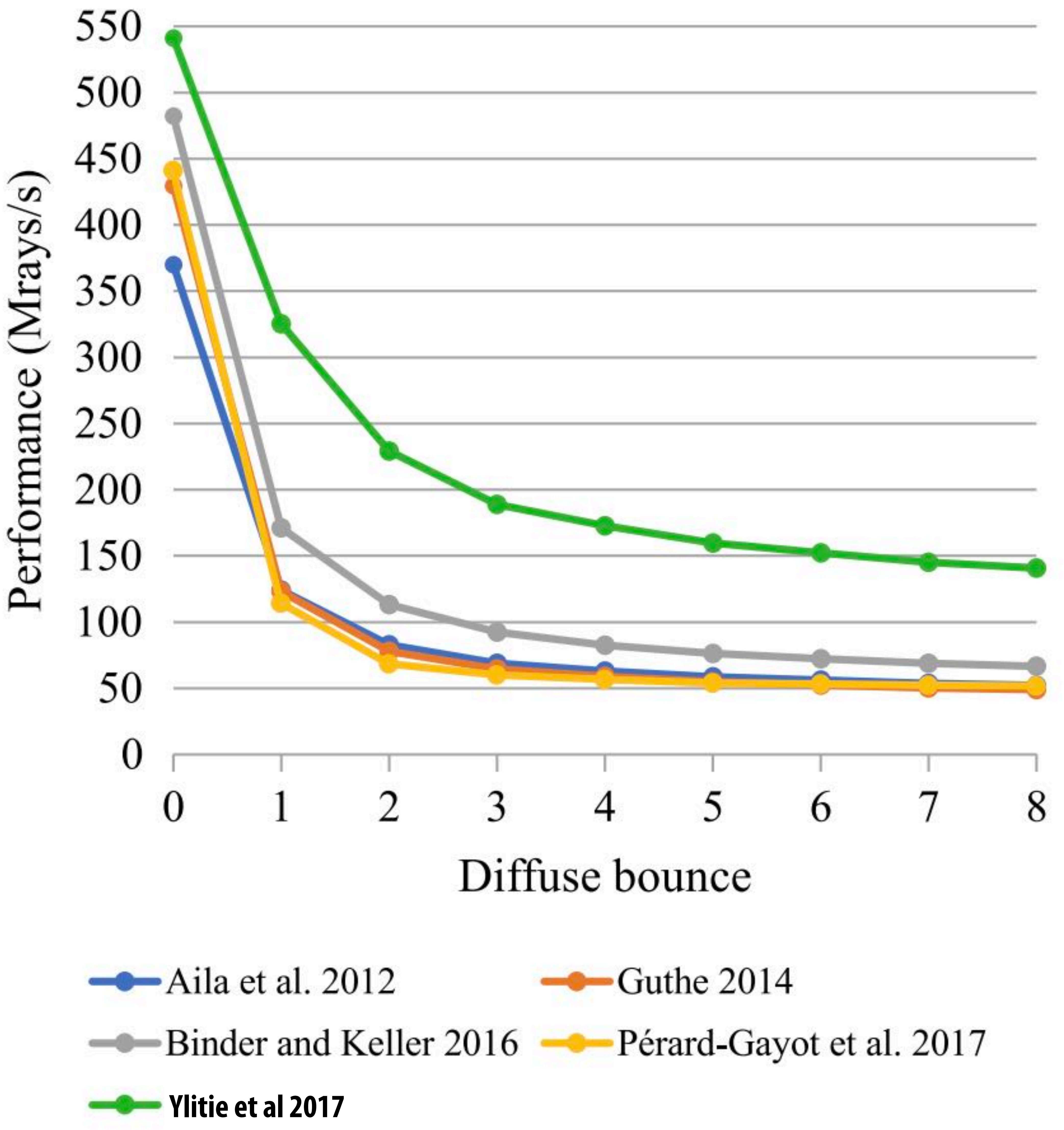
Incoherent rays = bandwidth bound

Different threads may access different BVH nodes at the same time:

Note how R0/R2 are accessing D while R1 is accessing C



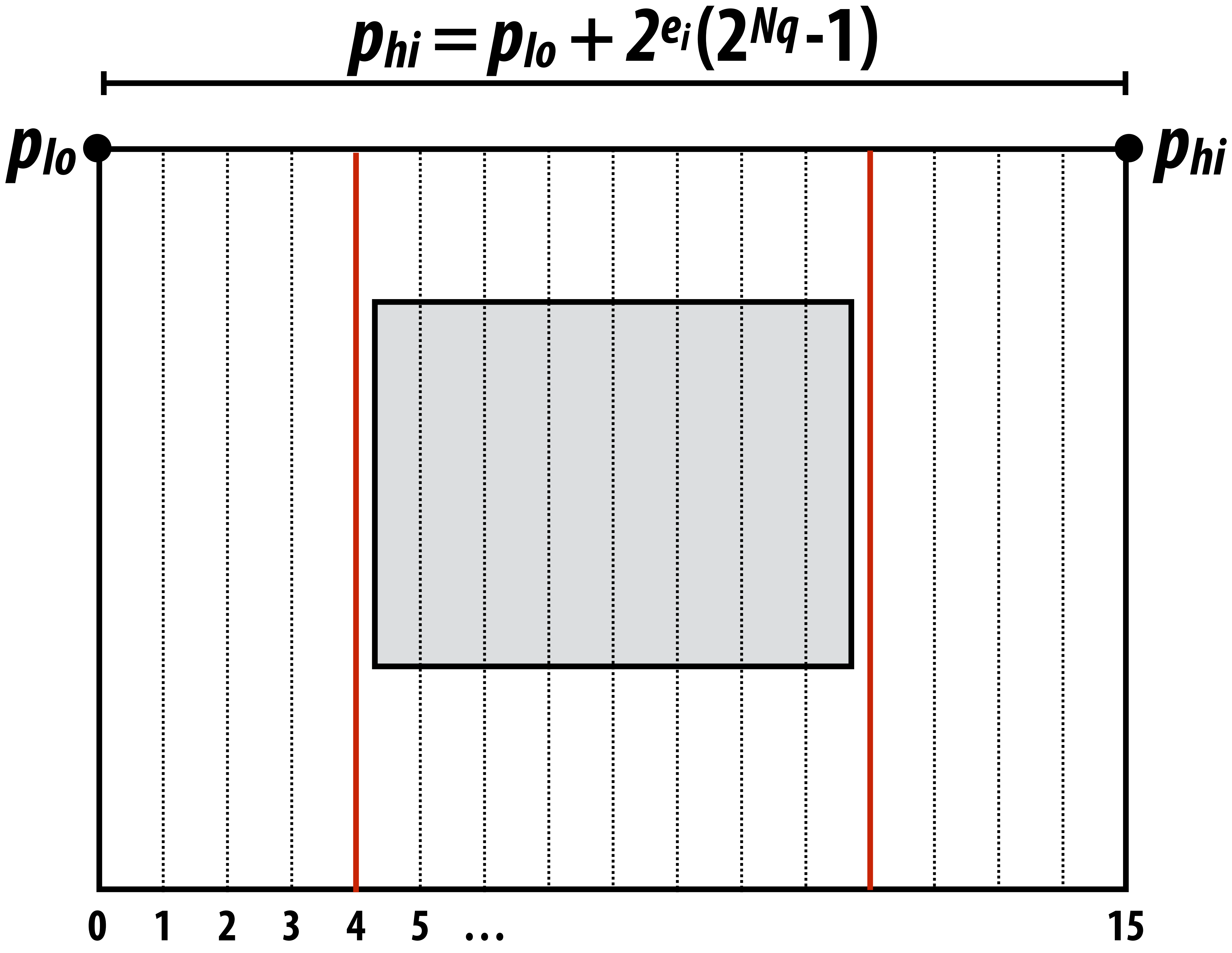
Ray throughput decreases with increasing numbers of bounces
(The more light bounces around a scene, the greater the ray divergence)



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



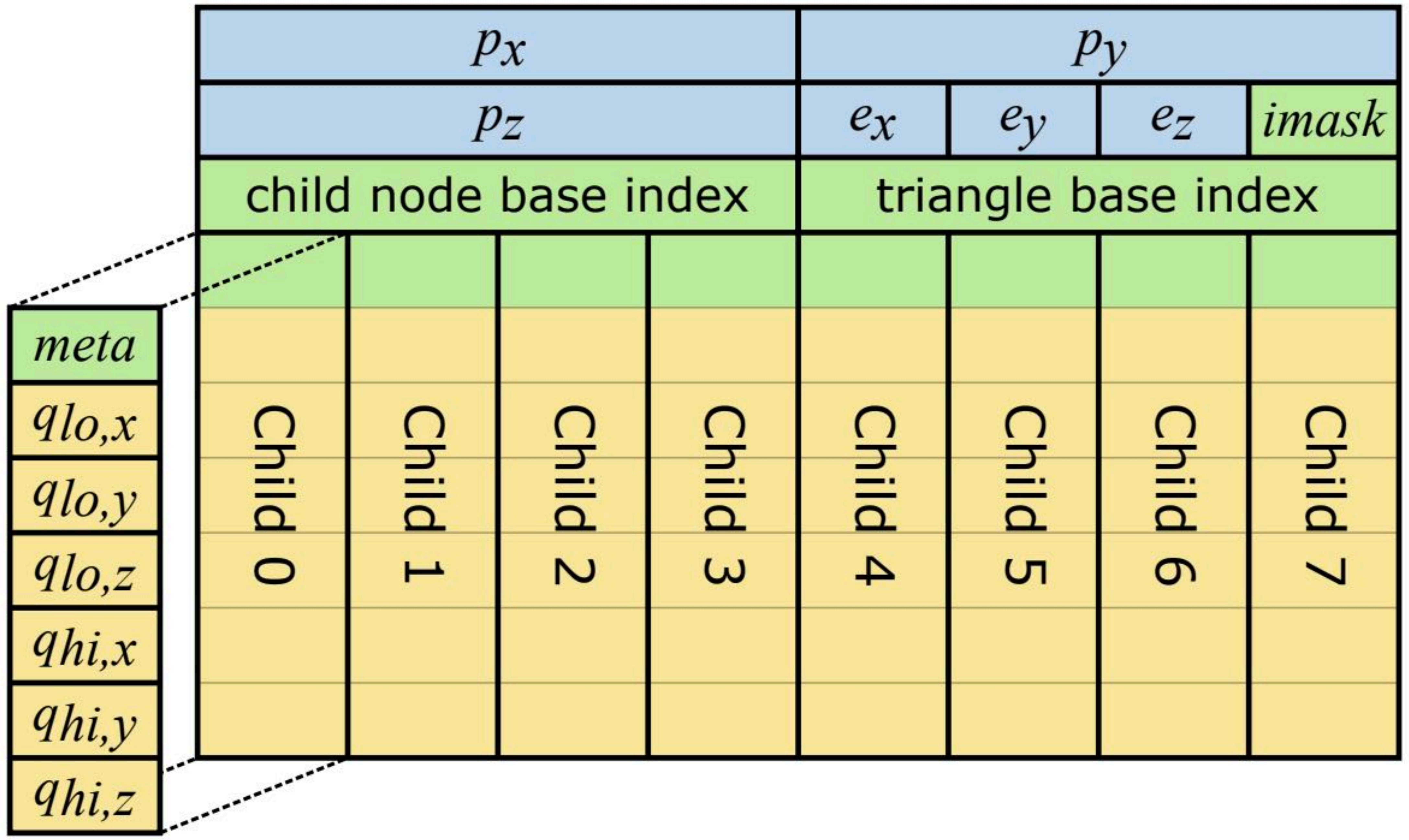
e_i encodes 8 bit exponent that defines "scale" of the parent bbox so that quantized N_q-bit values can be used to represent points in local coordinate frame

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 (4 children, 8 children) to:**
  - **Amortize storage of local coordinate frame definition across multiple child nodes**
  - **Reduce number of BVH node requests during traversal**



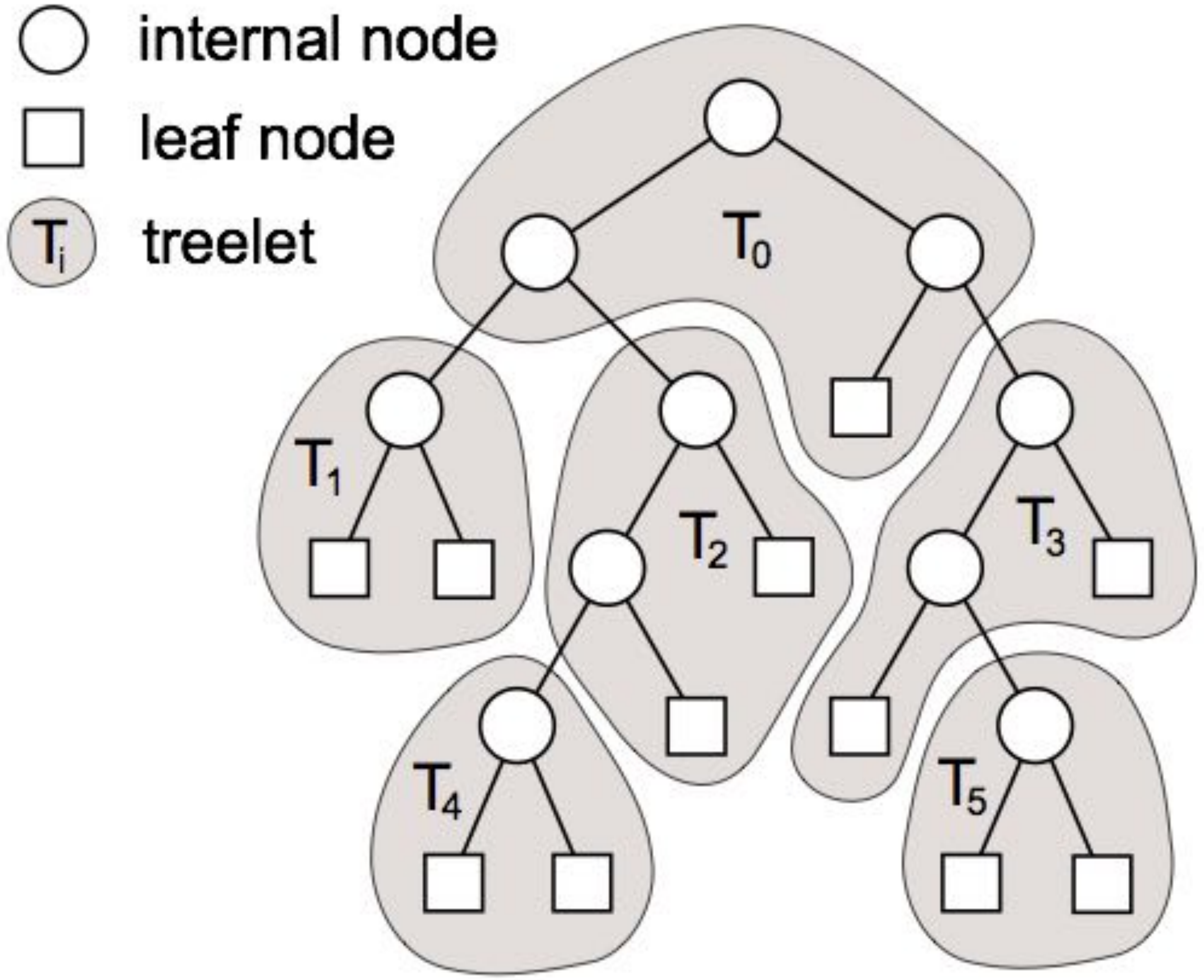
Amortized 10 bytes per child
(3.2x compression over standard BVH formats)

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

Incurs overhead of buffering: must store per-ray "stack" for many rays.

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

Hardware acceleration for ray tracing

NVIDIA Ampere SM (RTX 3xxx series)

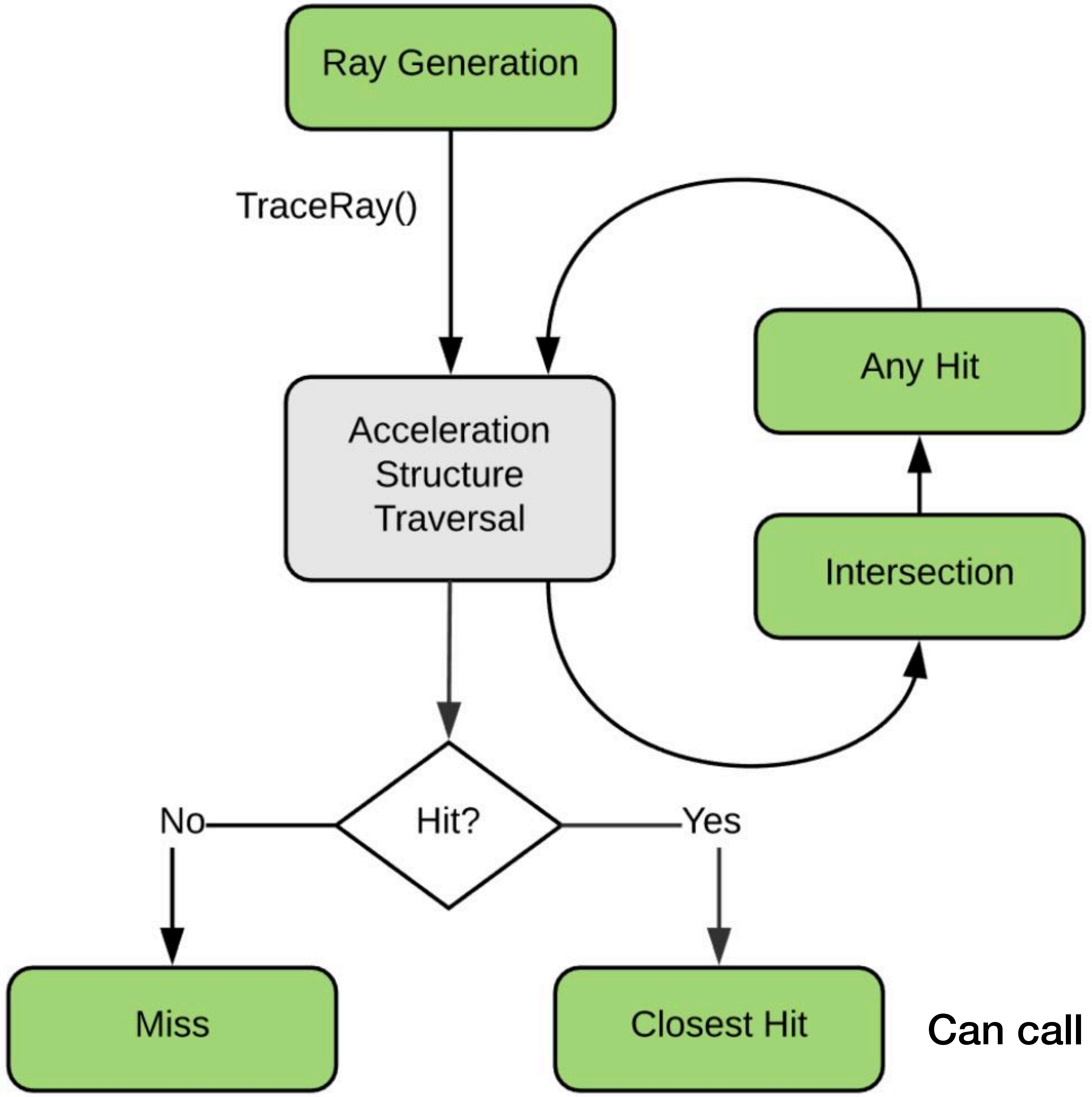
- Hardware support for ray-triangle intersection and ray-BVH intersection (“RT core”)
- Very little public documentation of architectural details at this time

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



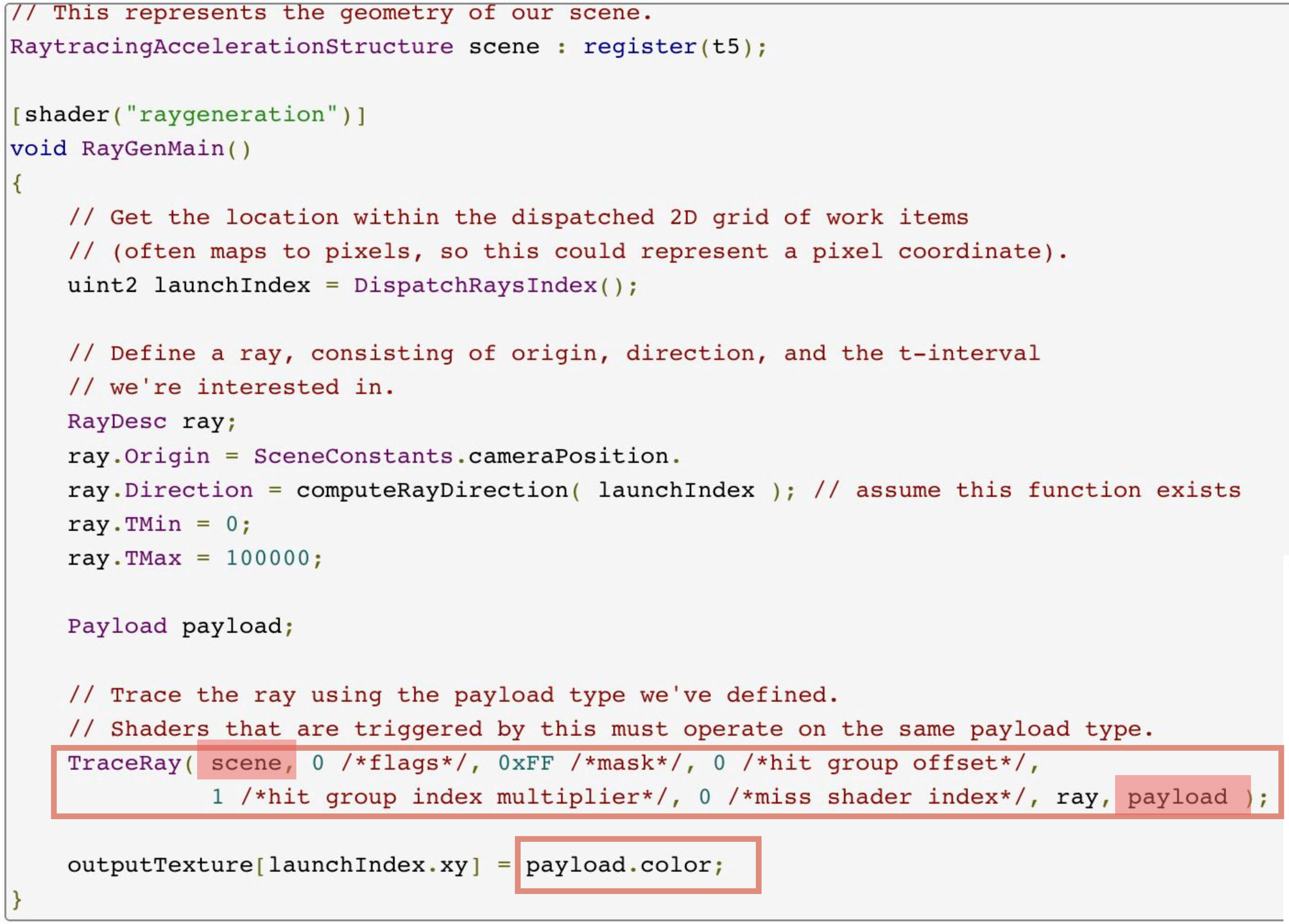
Acceleration structure

Can call TraceRay()

Can call TraceRay()

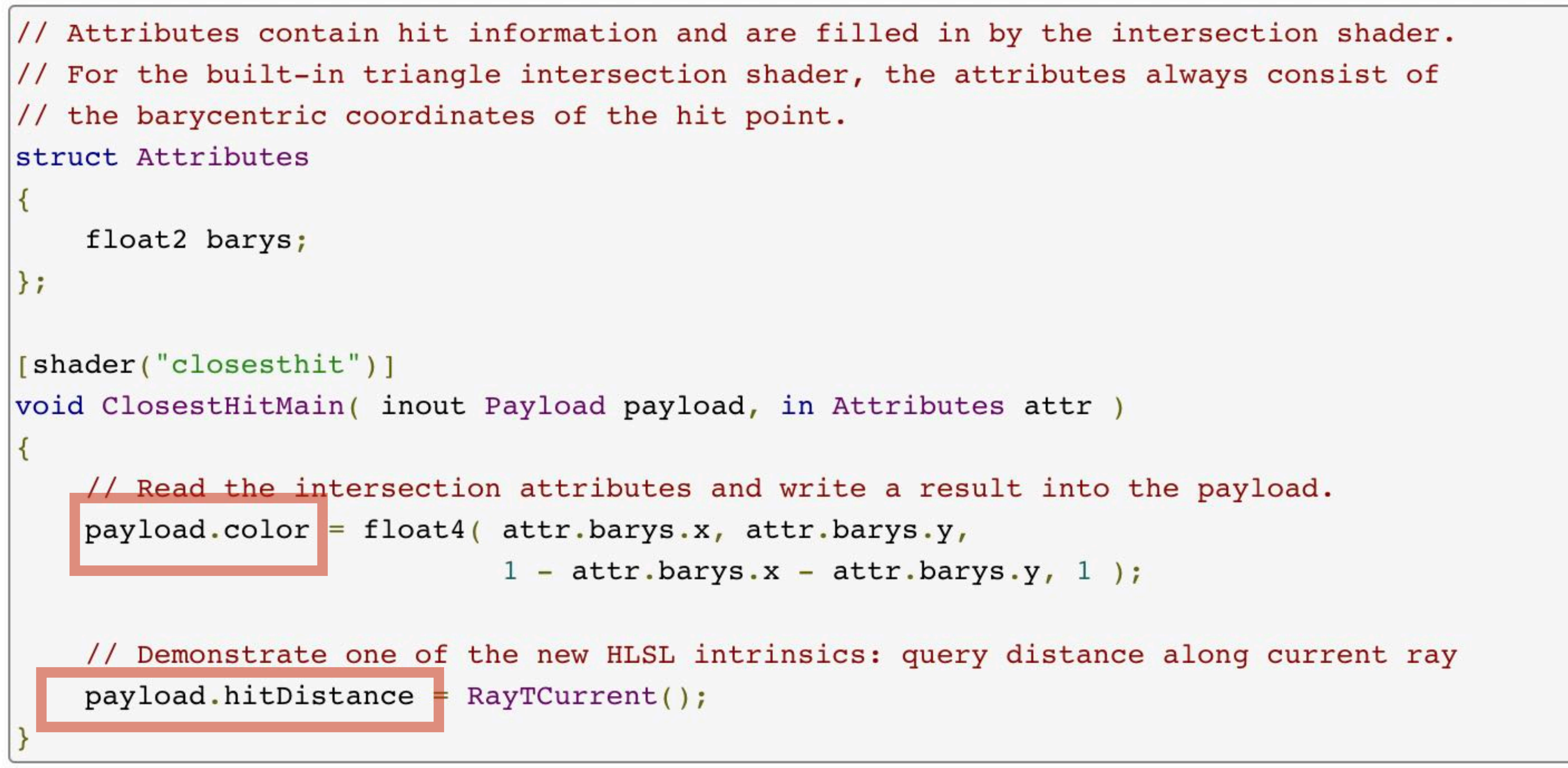
Can call TraceRay()

Example: ray generation shader (camera rays)



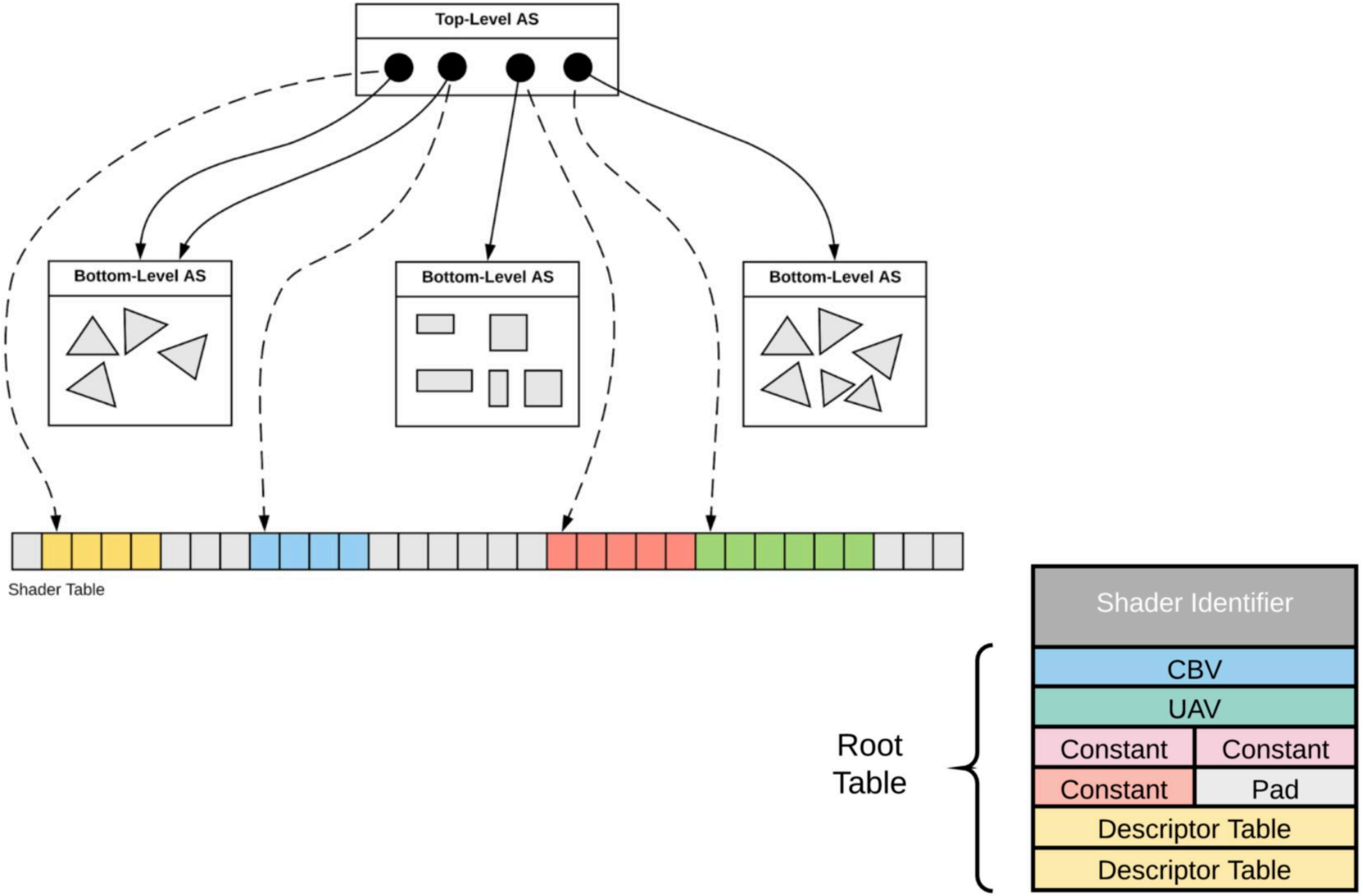
```
// This represents the geometry of our scene.
RaytracingAccelerationStructure scene : register(t5);

[shader("raygeneration")]
void RayGenMain()
{
    // Get the location within the dispatched 2D grid of work items
    // (often maps to pixels, so this could represent a pixel coordinate).
    uint2 launchIndex = DispatchRaysIndex();

    // Define a ray, consisting of origin, direction, and the t-interval
    // we're interested in.
    RayDesc ray;
    ray.Origin = SceneConstants.cameraPosition.
    ray.Direction = computeRayDirection( launchIndex ); // assume this function exists
    ray.TMin = 0;
    ray.TMax = 100000;

    Payload payload;

    // Trace the ray using the payload type we've defined.
    // Shaders that are triggered by this must operate on the same payload type.
    TraceRay( scene, 0 /*flags*/, 0xFF /*mask*/, 0 /*hit group offset*/,
              1 /*hit group index multiplier*/, 0 /*miss shader index*/, ray, payload );

    outputTexture[launchIndex.xy] = payload.color;
}
```

Example "hit shader": Runs on ray hit to fill in payload



```
// Attributes contain hit information and are filled in by the intersection shader.
// For the built-in triangle intersection shader, the attributes always consist of
// the barycentric coordinates of the hit point.
struct Attributes
{
    float2 barys;
};

[shader("closesthit")]
void ClosestHitMain( inout Payload payload, in Attributes attr )
{
    // Read the intersection attributes and write a result into the payload.
    payload.color = float4( attr.barys.x, attr.barys.y,
                            1 - attr.barys.x - attr.barys.y, 1 );

    // Demonstrate one of the new HLSL intrinsics: query distance along current ray
    payload.hitDistance = RayTCurrent();
}
```

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



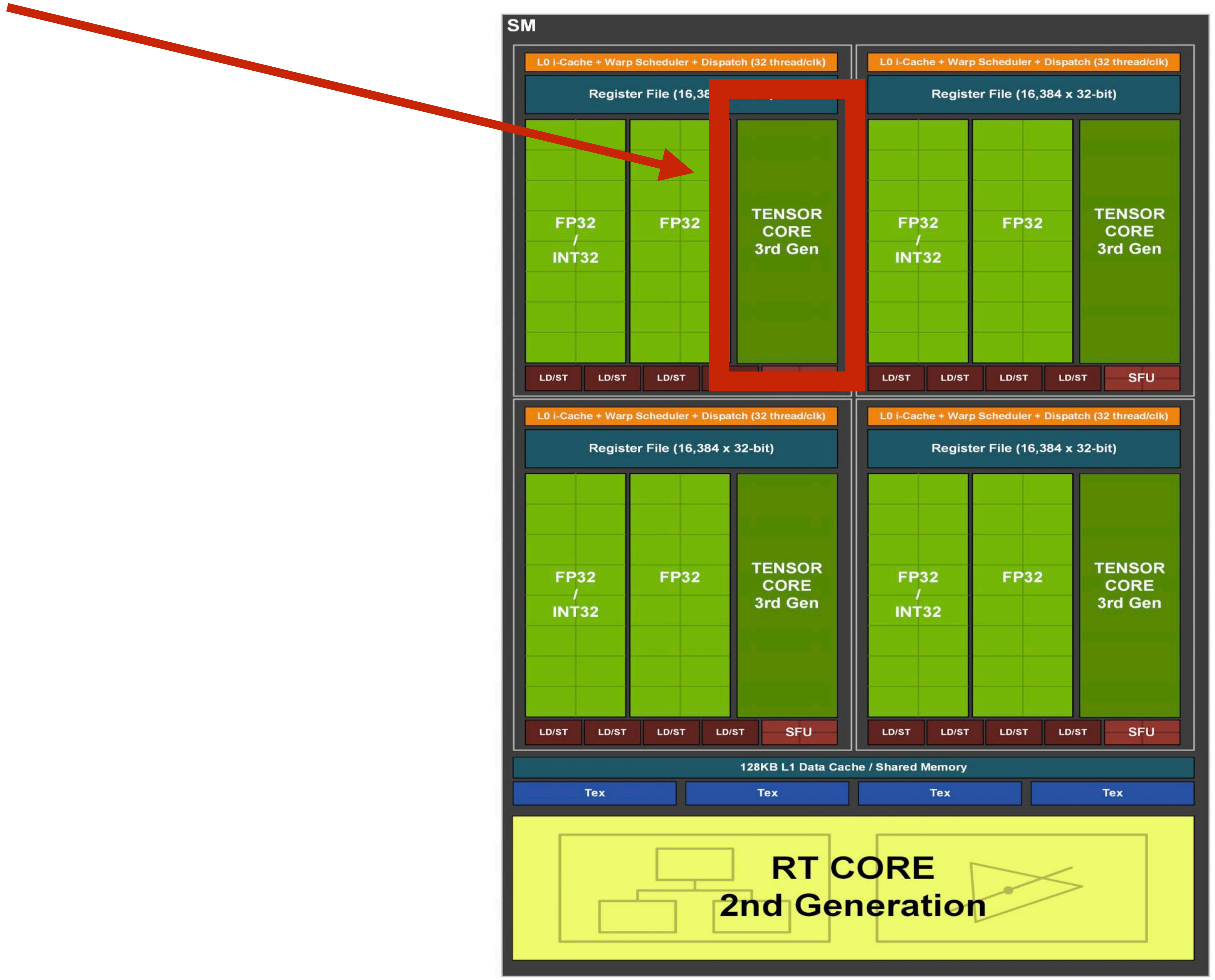
The story so far…

- "High level" raytracing APIs for authoring ray tracing applications
  - High level abstractions allow for extensive optimizations

- Application uses API to "create a BVH"
  - Since API creates BVH, it can make hardware-specific data layout decisions
    - How to compress BVH data structure
    - How wide BVH should be (2 children, 4 children, 8 children?)
  - Knowledge of BVH format about allows use of fixed-function hardware to execute ray-BVH traversal (through compressed structure)

- Application provides functions (shaders) that API calls when certain events happen (ray hits triangle, ray misses all triangles, etc.

- **But the RT hardware is not the only fixed-function hardware on a GPU that is important for real-time raytracing…**



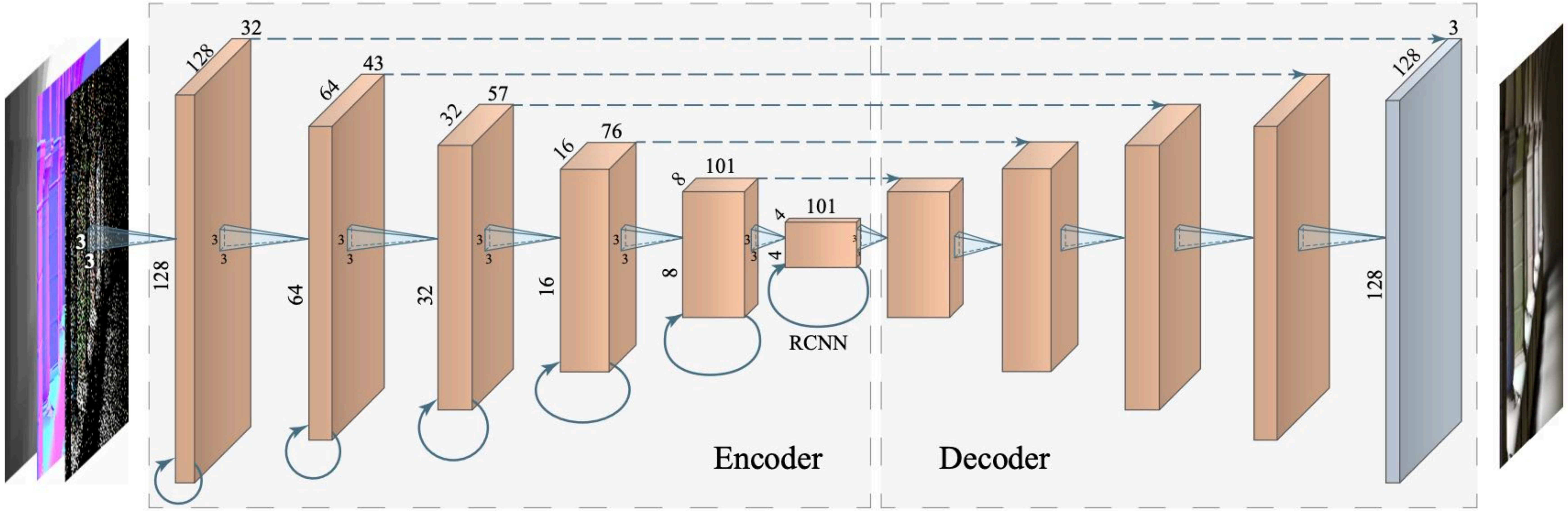
Denoising ray traced images

Deep learning-based denoising

- "Learn" to turn noisy images (computed using only a few light paths per pixel) into noise-free images (that look like images computed using many paths per pixel)?

- Idea: Use neural image-to-image transfer methods to convert cheaper to compute (but noisy) ray traced images into higher quality images that look like they were produced by tracing many rays per pixel

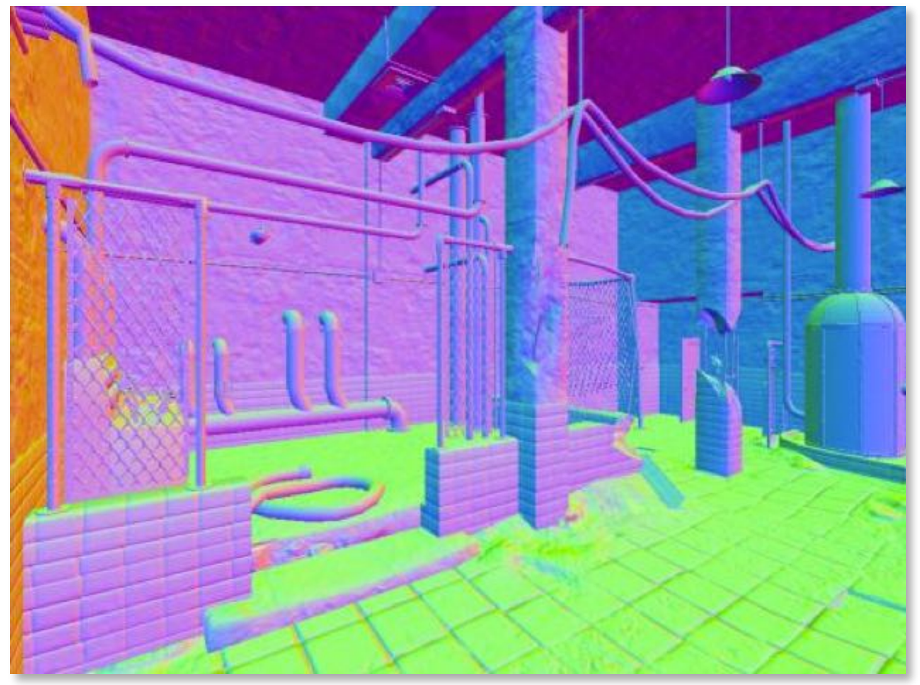
[Chaitanya 17]



Input to network is noisy RGB image * + additional normal, depth, and roughness channels
(These are cheap to compute inputs help network identify silhouettes, sharp structure)

Depth

Normal



Roughness

Albedo

*** Actually the input is RGB demodulated by (divided by) texture albedo (don't force network to learn what texture was)**


VICTROLA

Surface Albedo

Surface normals

16 paths/pixel


64 paths/pixel

256 paths/pixel


1024 paths/pixel


4096 paths/pixel
VICTROLA

Denoised results


16 paths/pixel

16 paths/pixel (denoised)

64 paths/pixel (denoised)

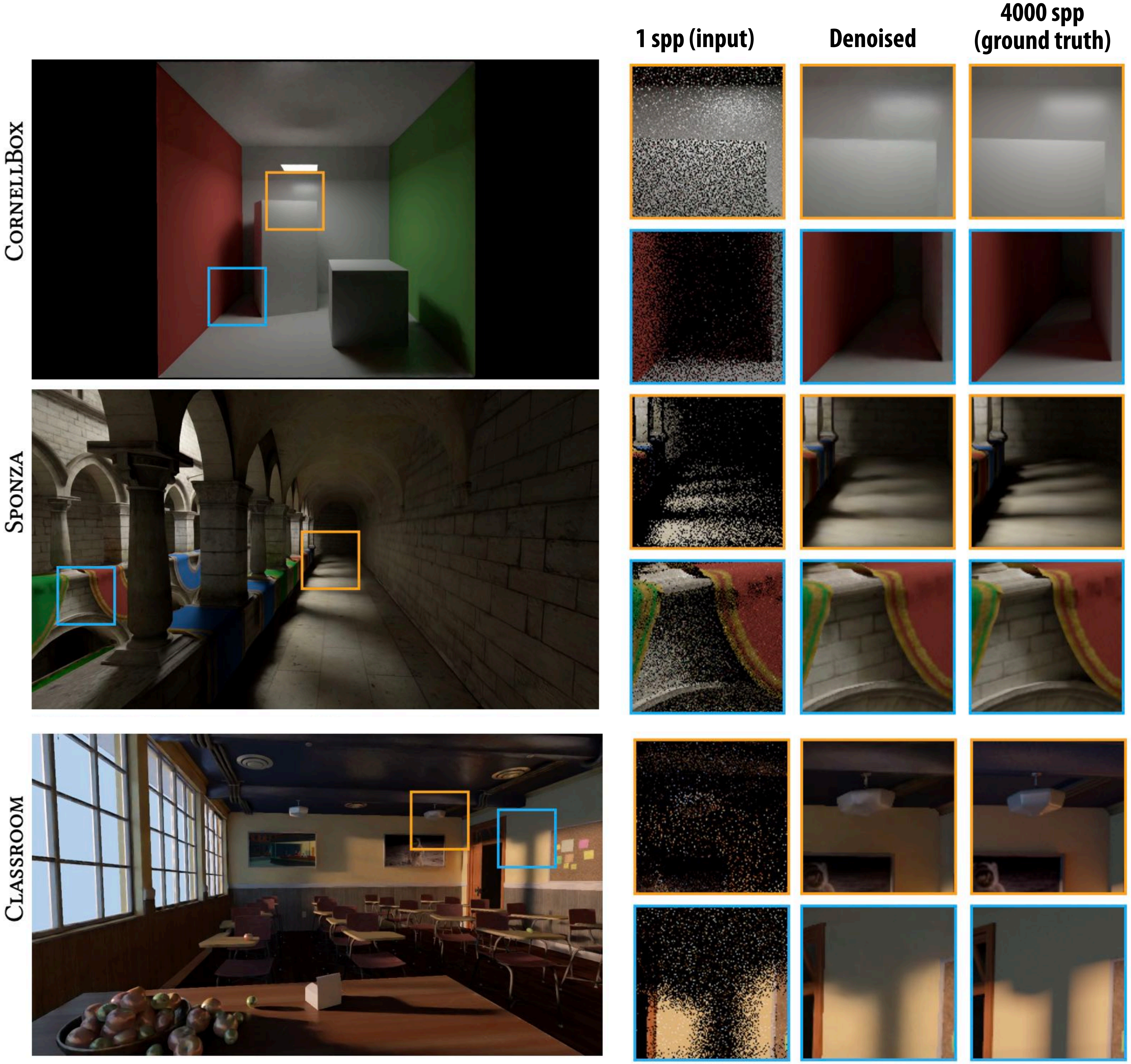
256 paths/pixel (denoised)

1024 paths/pixel (denoised)

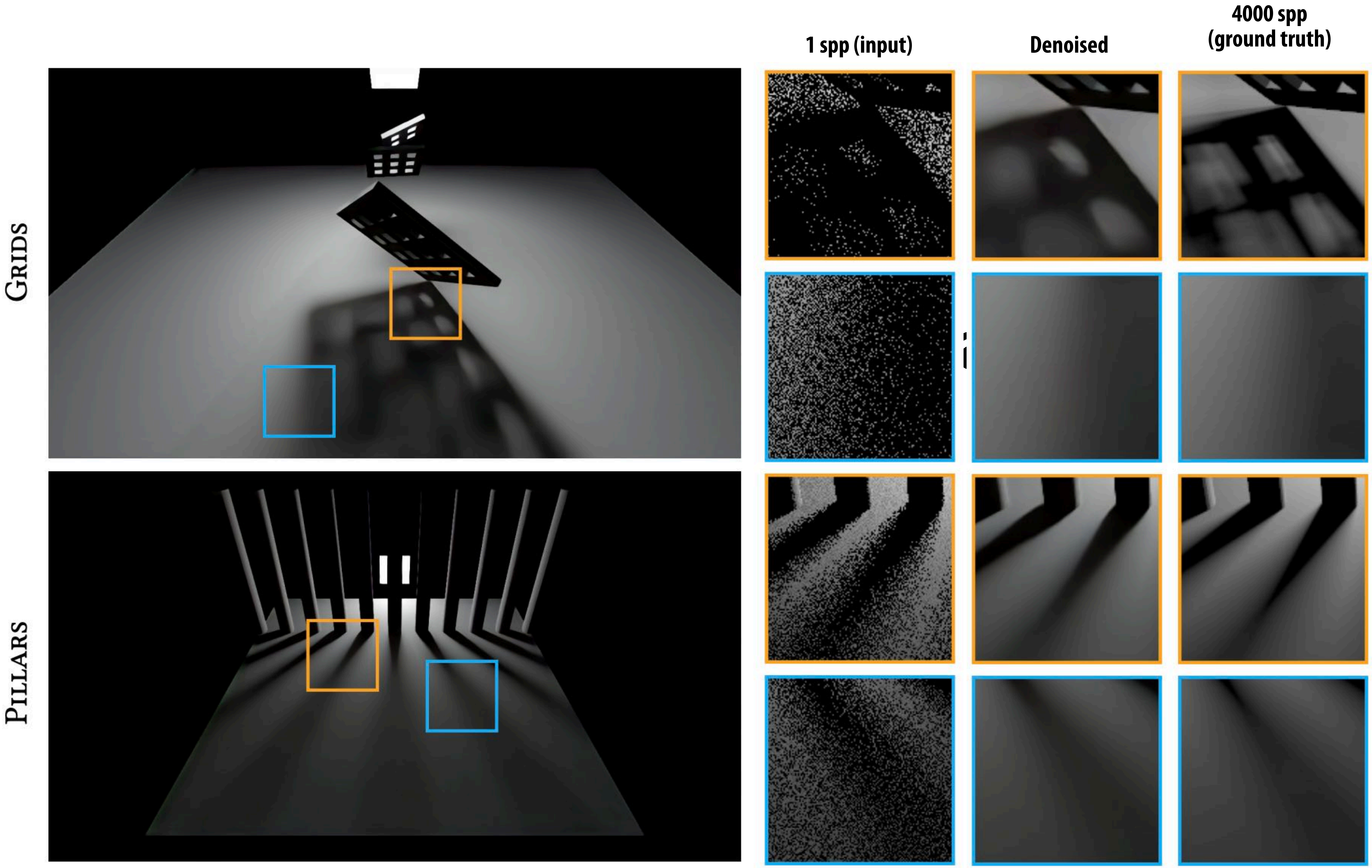
4096 paths/pixel (denoised)


4096 paths/pixel (NOT DENOISED)

[Chaitanya 17]



[Chaitanya 17]



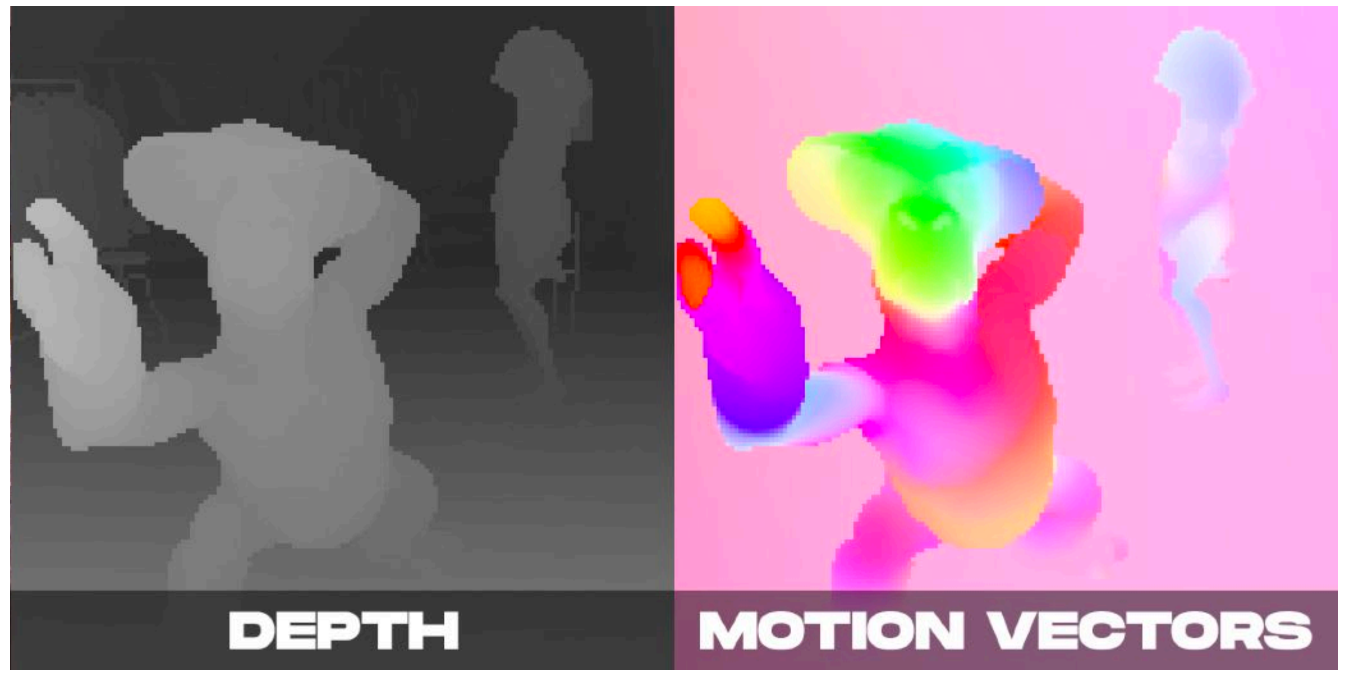
Aside: upsampling low-resolution images to higher resolution images

(This is upsampling, not reducing Monte Carlo noise.)

Examples: NVIDIA's DLSS (performs both anti-aliasing and upsampling)

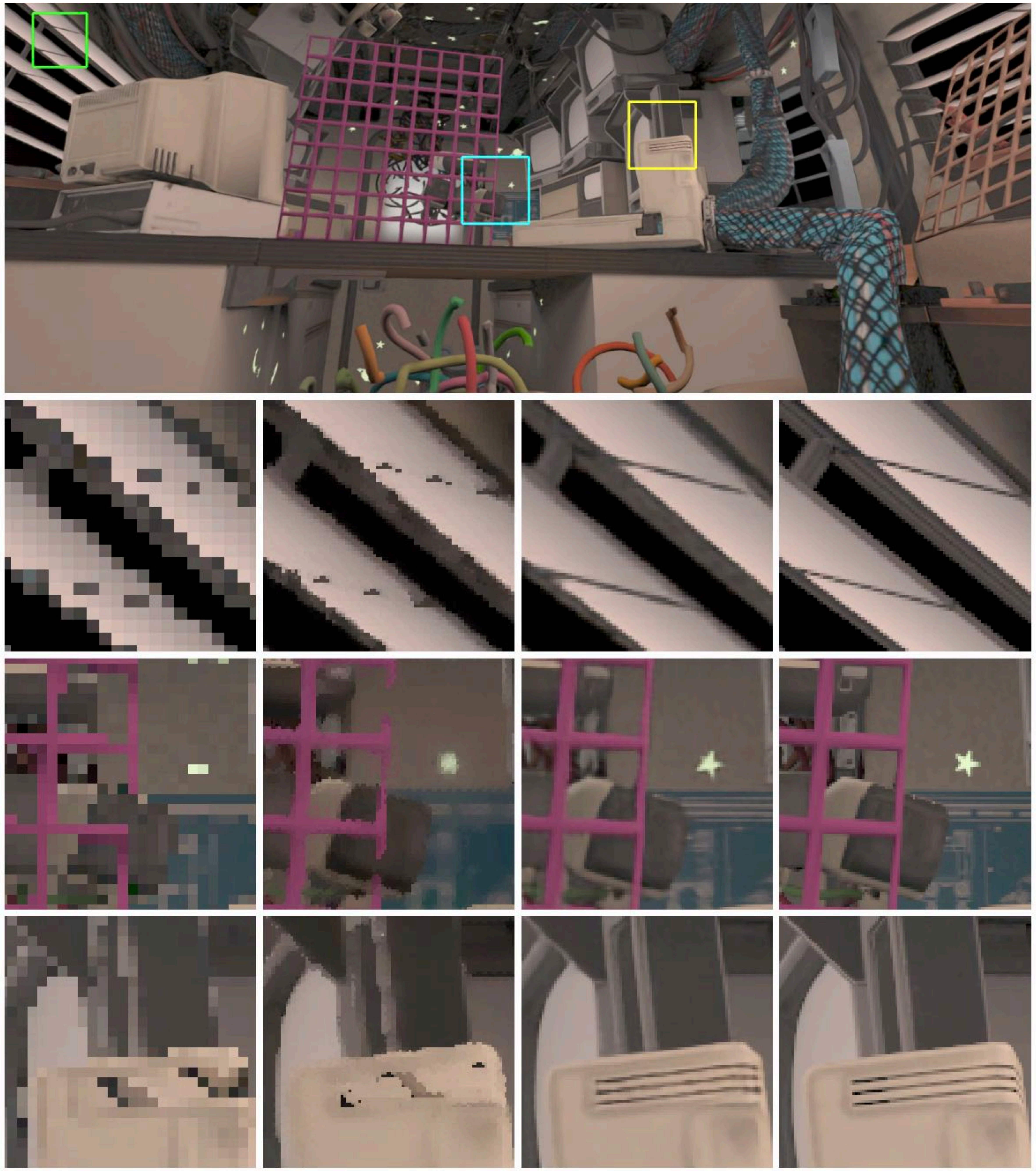
[Xiao 20]

+ auxiliary inputs



4x4 upsampled result (16x more pixels)

[Xiao 20]



Input Unreal TAAU Ours Reference

The story so far

- High-level APIs for real-time ray tracing
  - Enables system to choose efficient data structures
  - Enables use of fixed-function hardware to accelerate ray-BVH traversal and ray-triangle intersection

- Neural post-processing to turn low sample count images into high sample count images (or low resolution images into higher resolution ones)

- Still not enough…
  - Also need to intelligently pick light paths to "get the most information" out of each path.

Tens of thousands of lights…

[Bitterli et al. 2020]

Zero day scene (beeple@)

Very large number of lights

Uniform path sampling (16 spp)

Choosing 16 lights (K=16, uniform probability across lights), tracing one ray to random point on each light (N=1)

Sampling lights proportional to light power (16 spp)

Choosing 16 lights (K=16, light probability proportional to its power), tracing one ray to random point on each light (N=1)

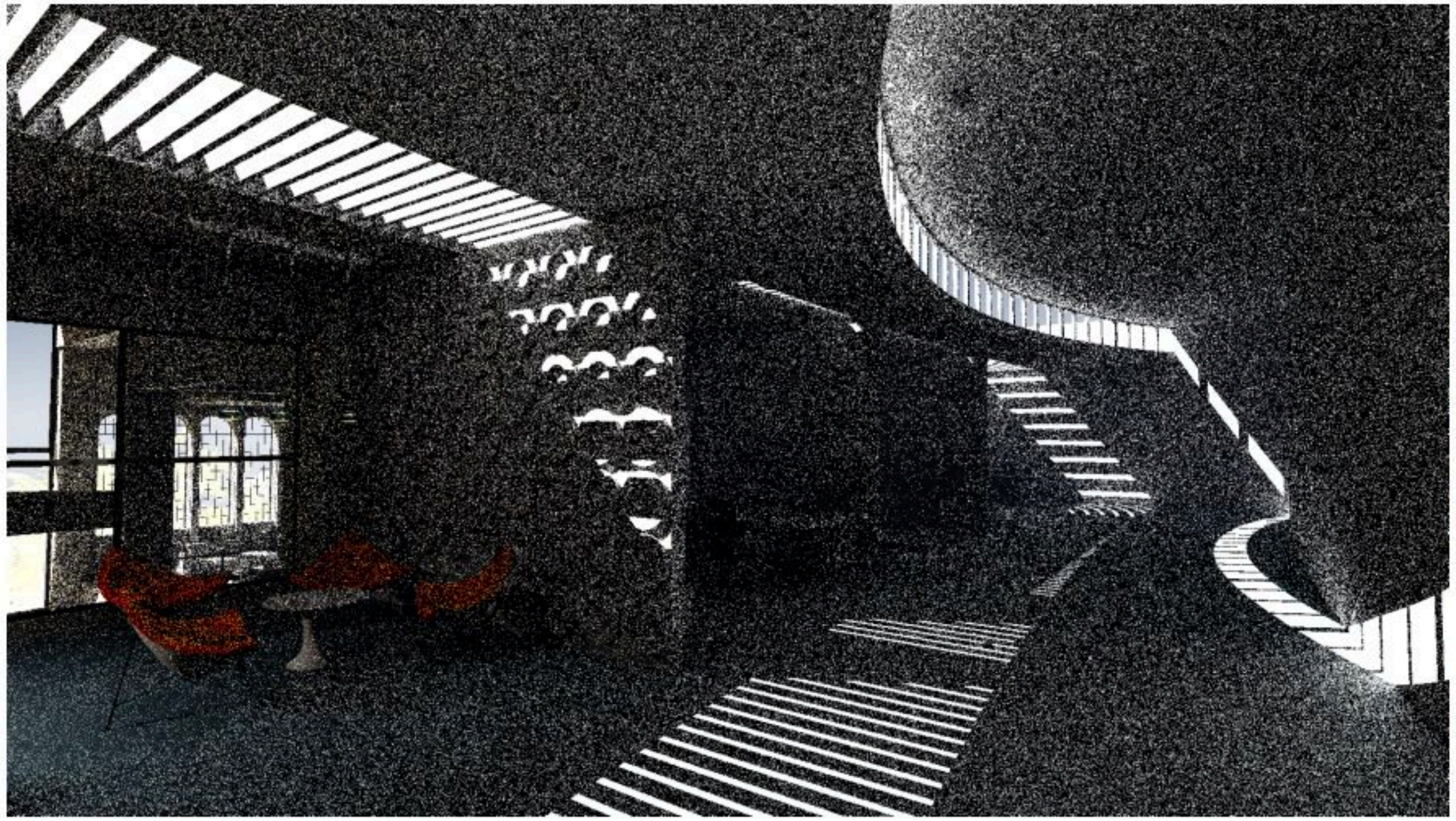
Advanced topic: path guiding

Use results from prior paths to influence choice of future paths.

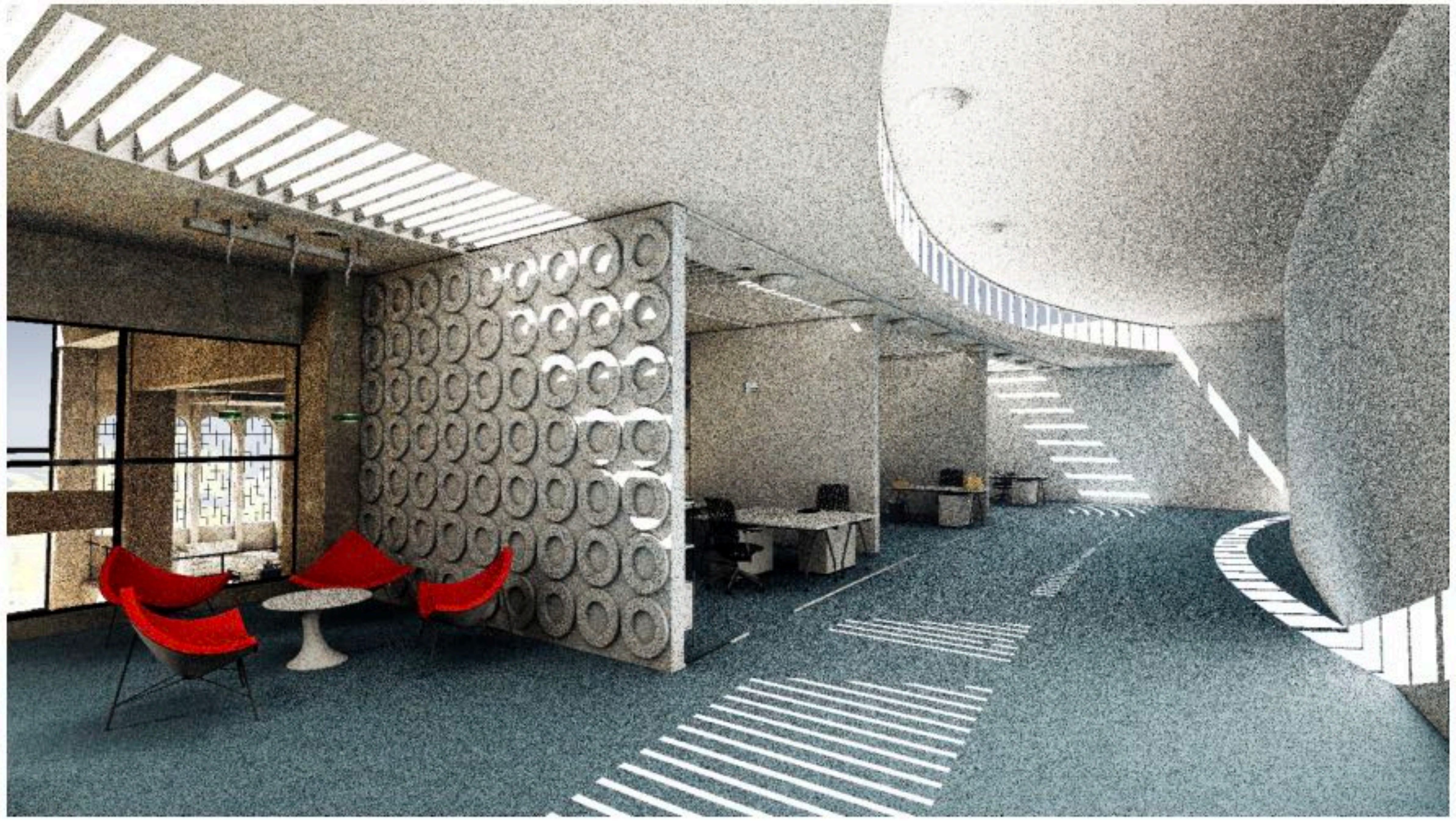
Baseline PPG Neural

[Müller et al. 2018]

Caching/reusing good paths



Path traced: 1 path/pixel (8 ms/frame)



Path traced: 1 path/pixel using ReSTIR GI (8.9 ms/frame)

Key idea: cache good paths, reuse good paths found from from prior frames or for prior pixels in same frame

High sample count path traced "ground truth"

[Ouyang et al. 2021]

Real-time raytracing innovations

- High-level APIs for real-time ray tracing
    - Enables system to choose efficient data structures
    - Enables use of fixed-function hardware to accelerate ray-BVH traversal and ray-triangle intersection

- Better "importance sampling" algorithms to picking the right light paths to trace
    - Improve image quality given a small budget for ray tracing
    - Reduce noise enough so that neural denoising can successfully finish the job

- Neural post-processing to turn low sample count images into high sample count images (or low resolution images into higher resolution ones)