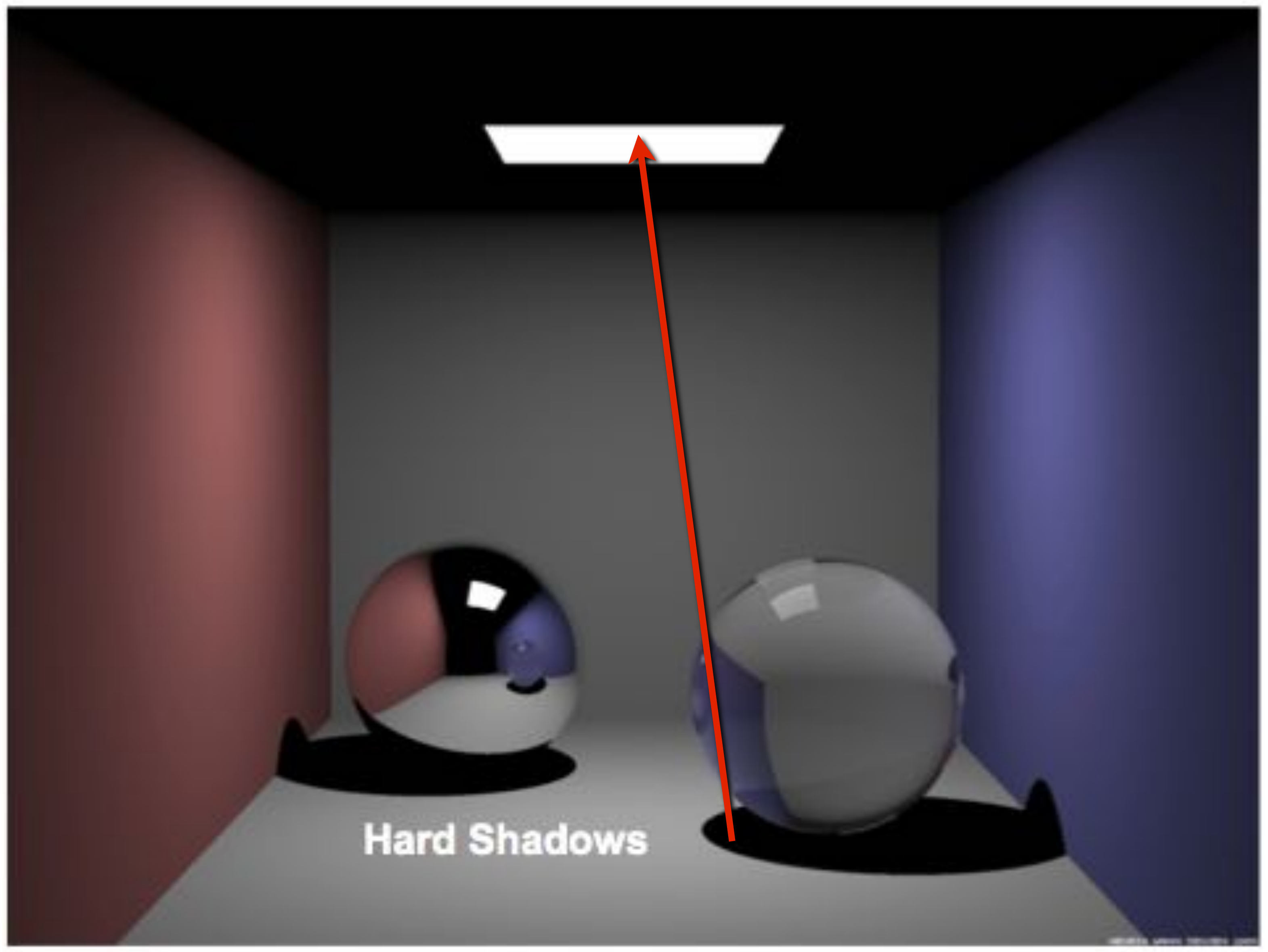
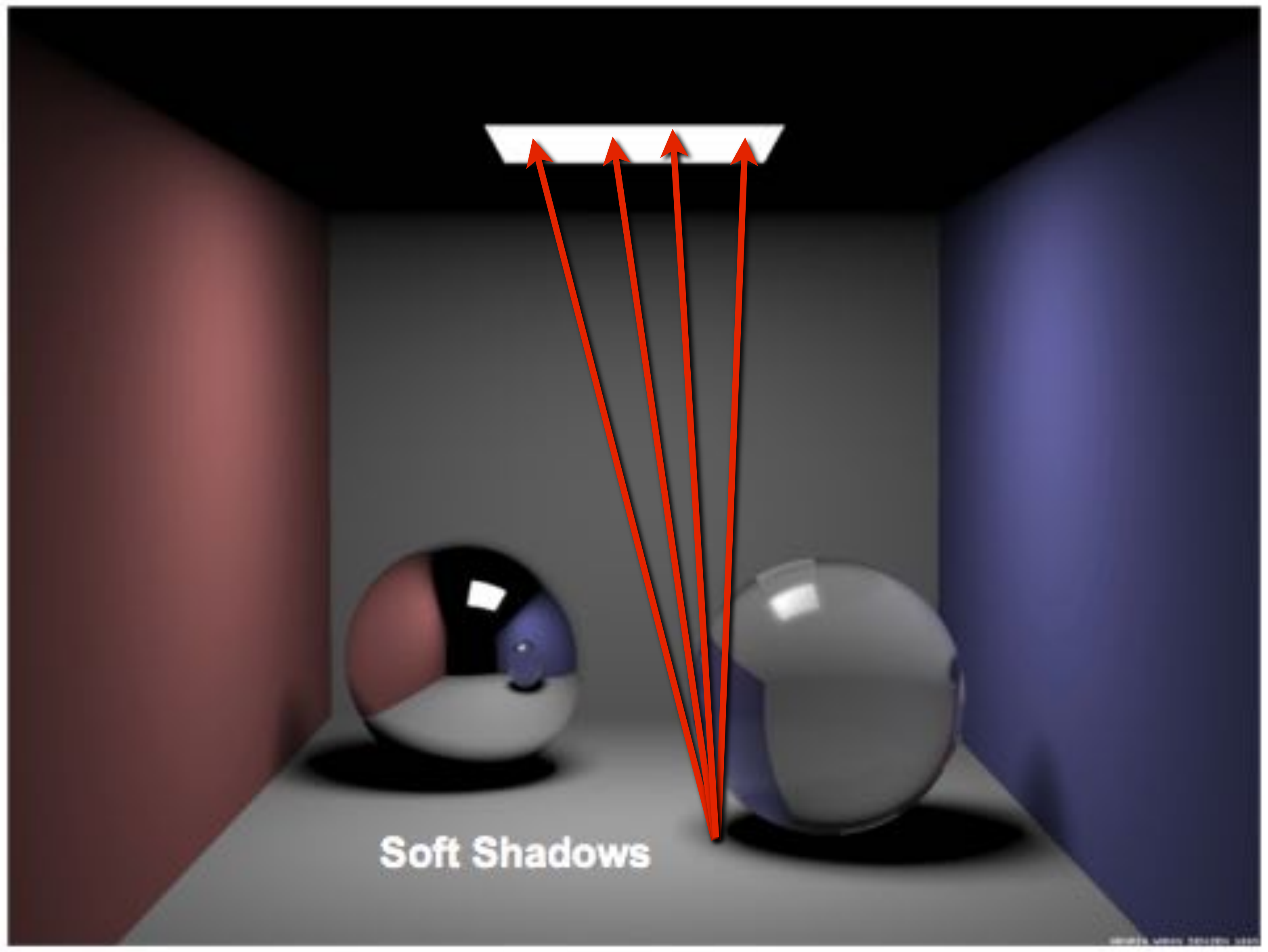
Lecture 15:

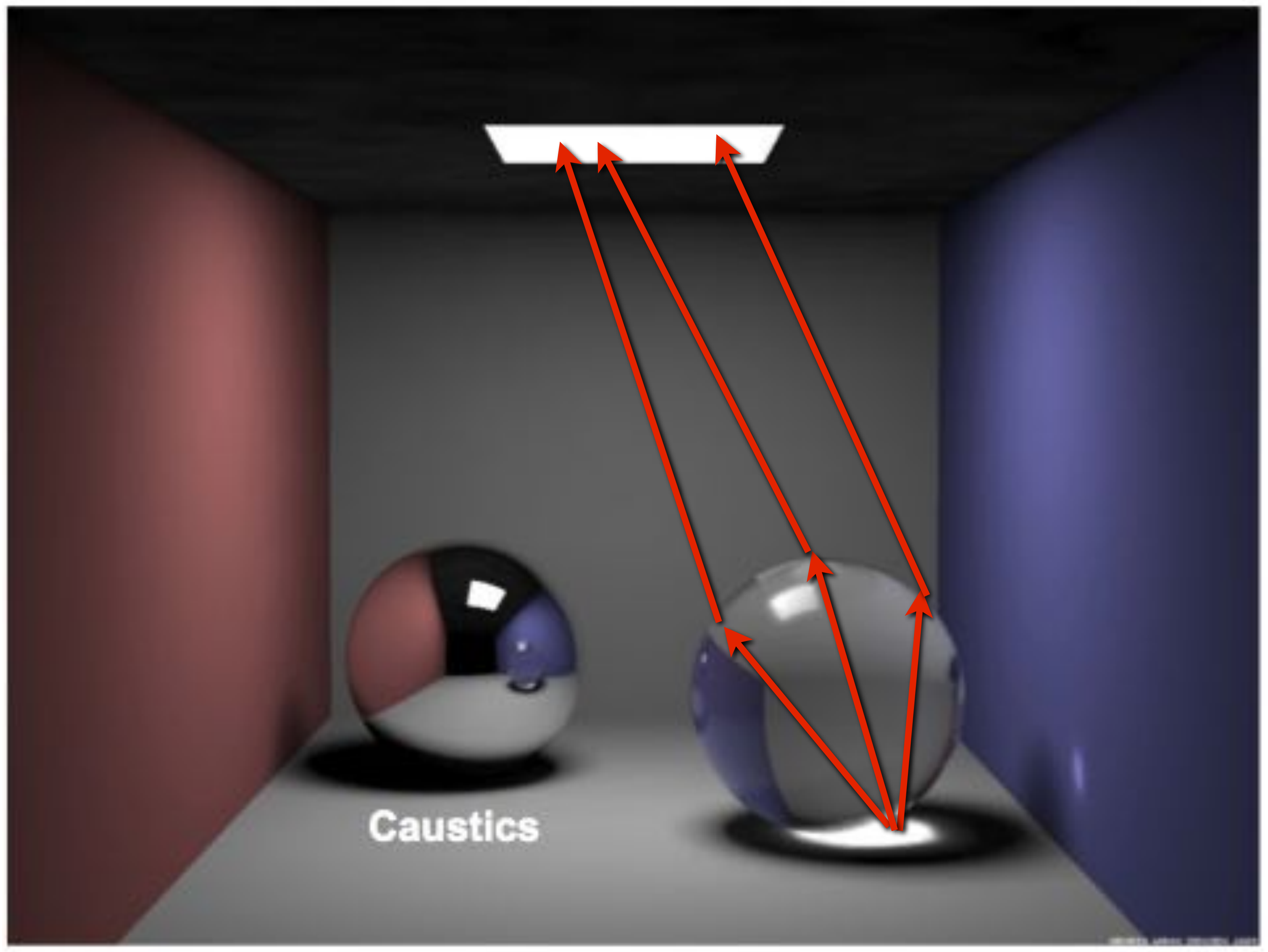
Optimizing Ray Tracing

Visual Computing Systems
Stanford CS348K, Spring 2021

Last time: a ray tracer samples light paths







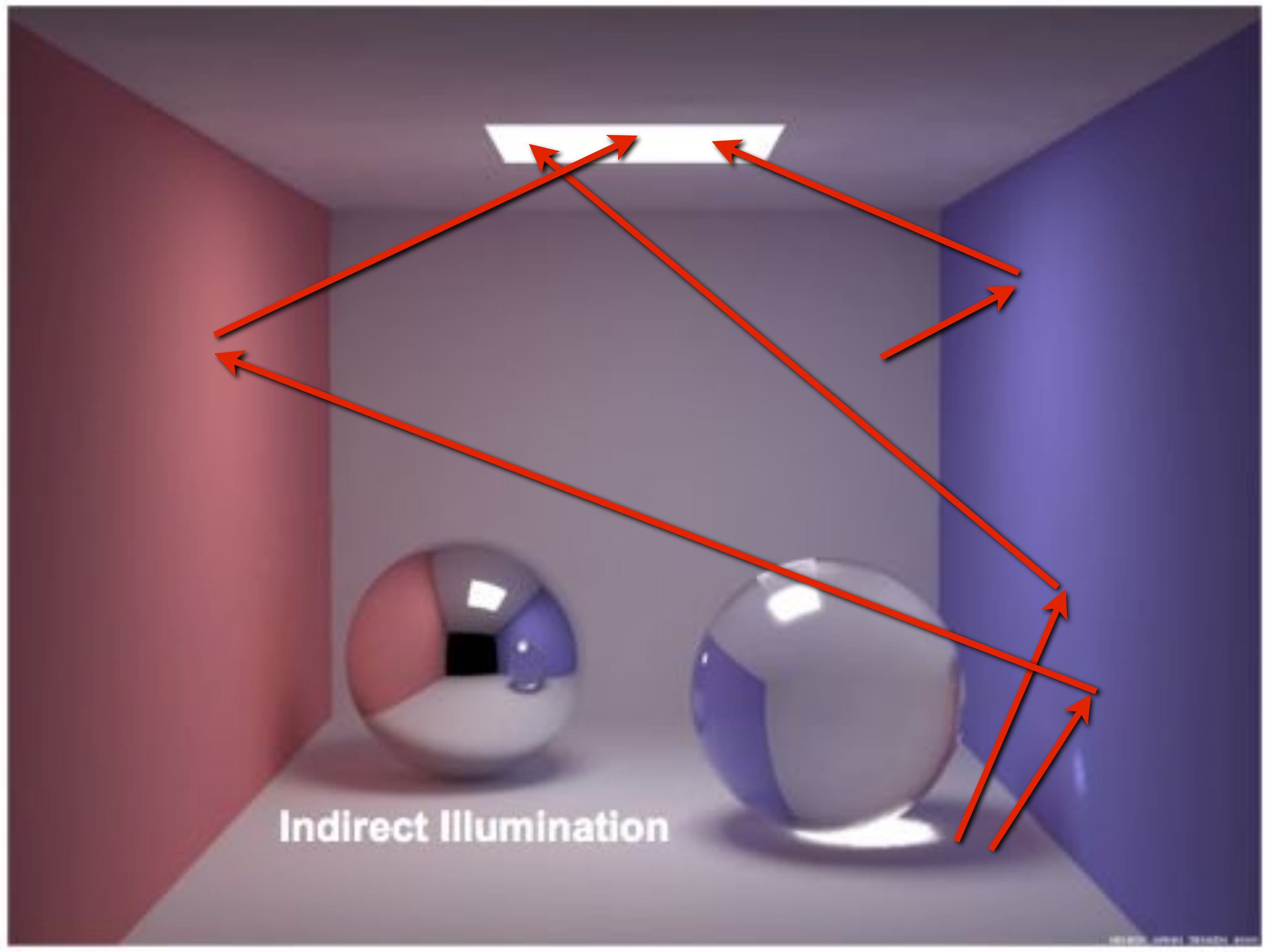
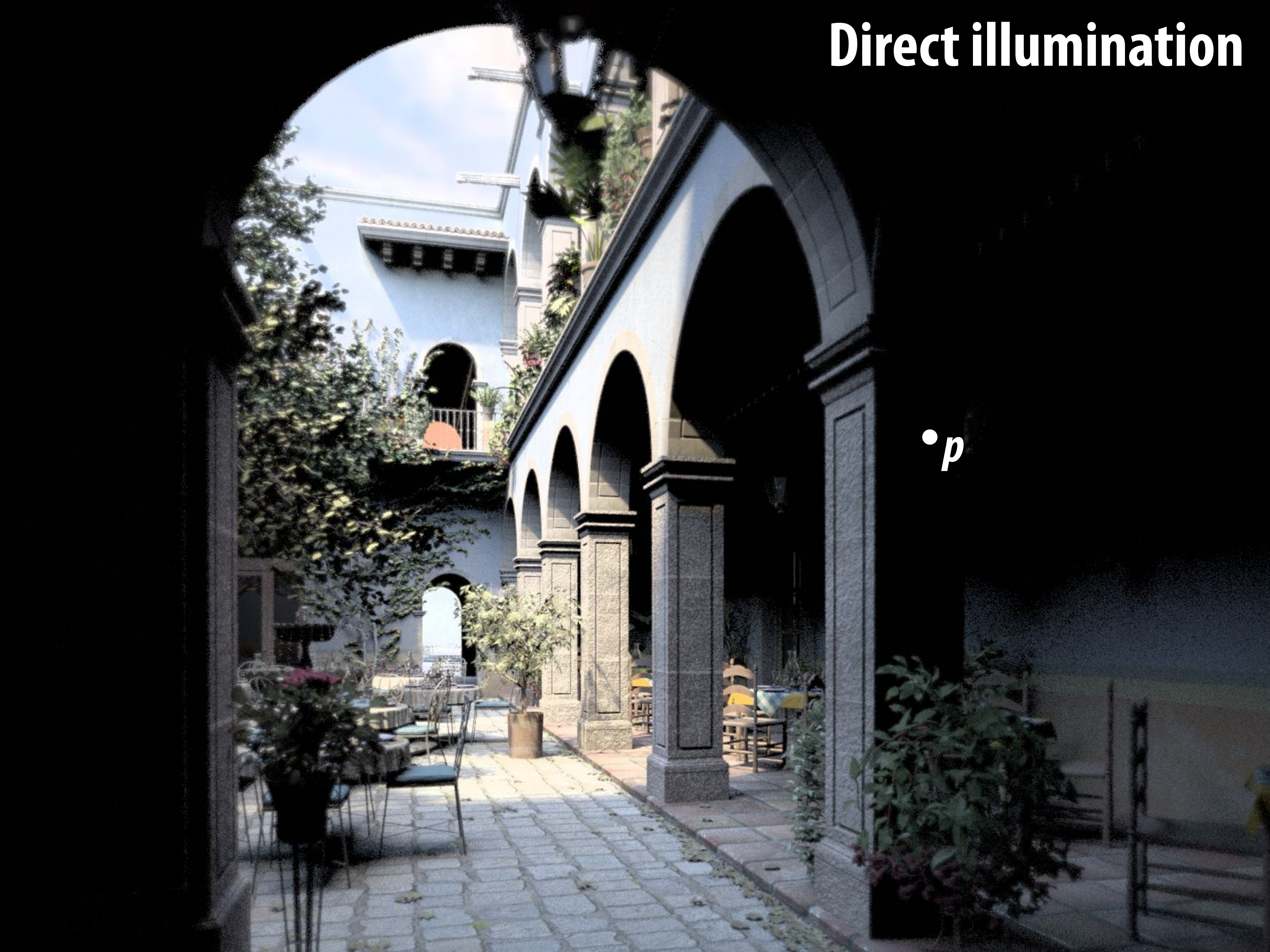
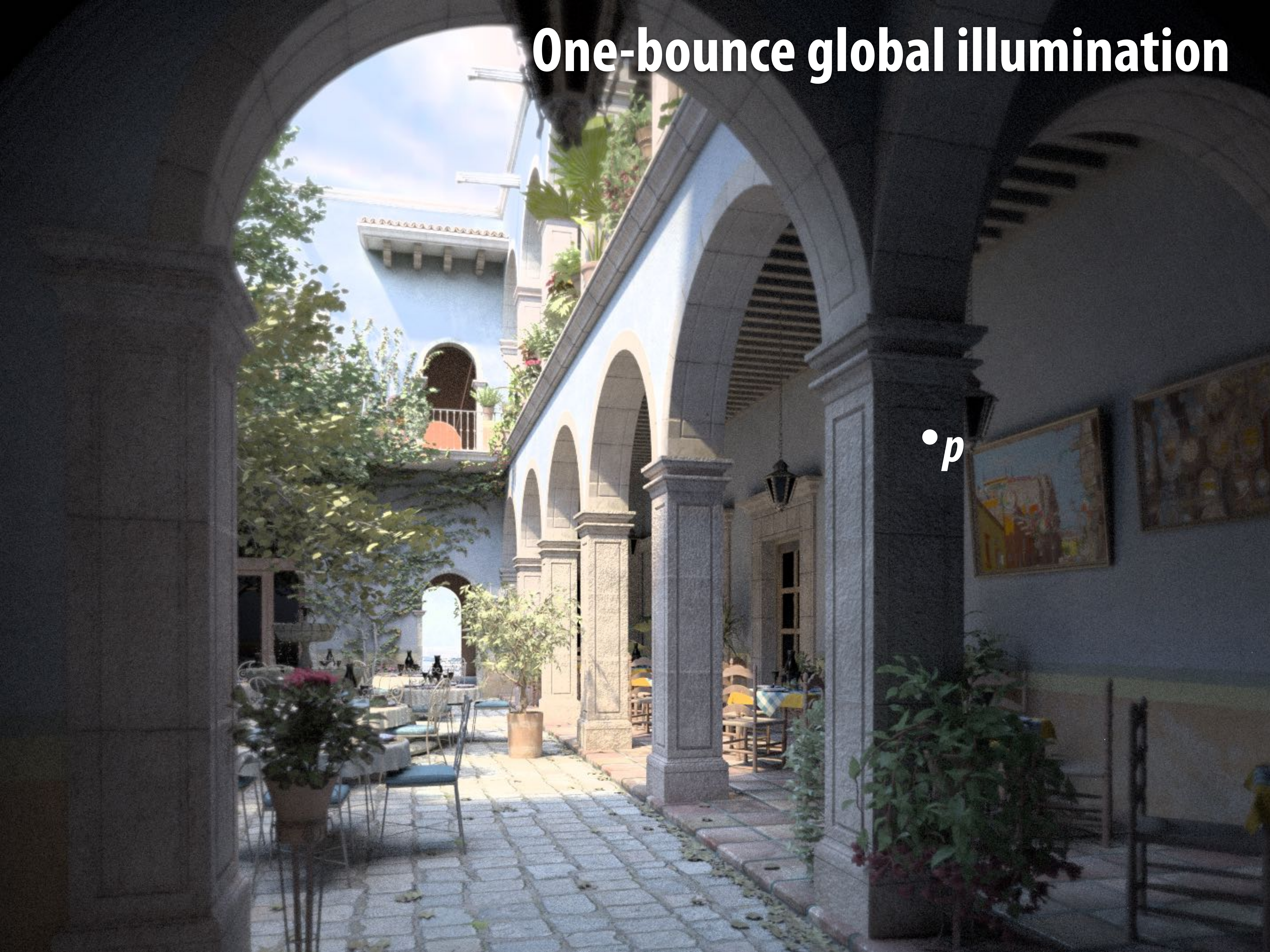


Image credit: Wann Jensen, Hanrahan

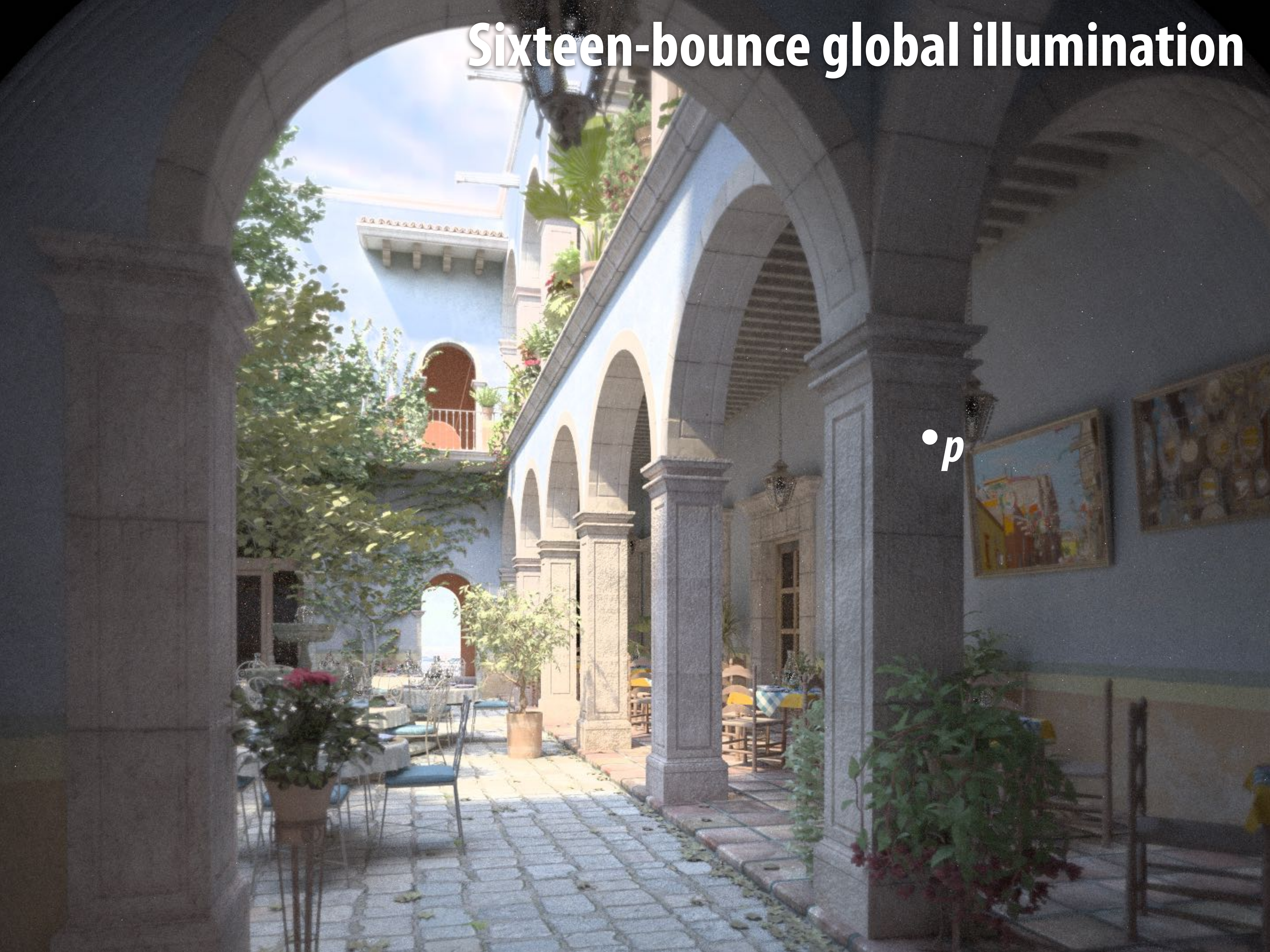
Direct illumination

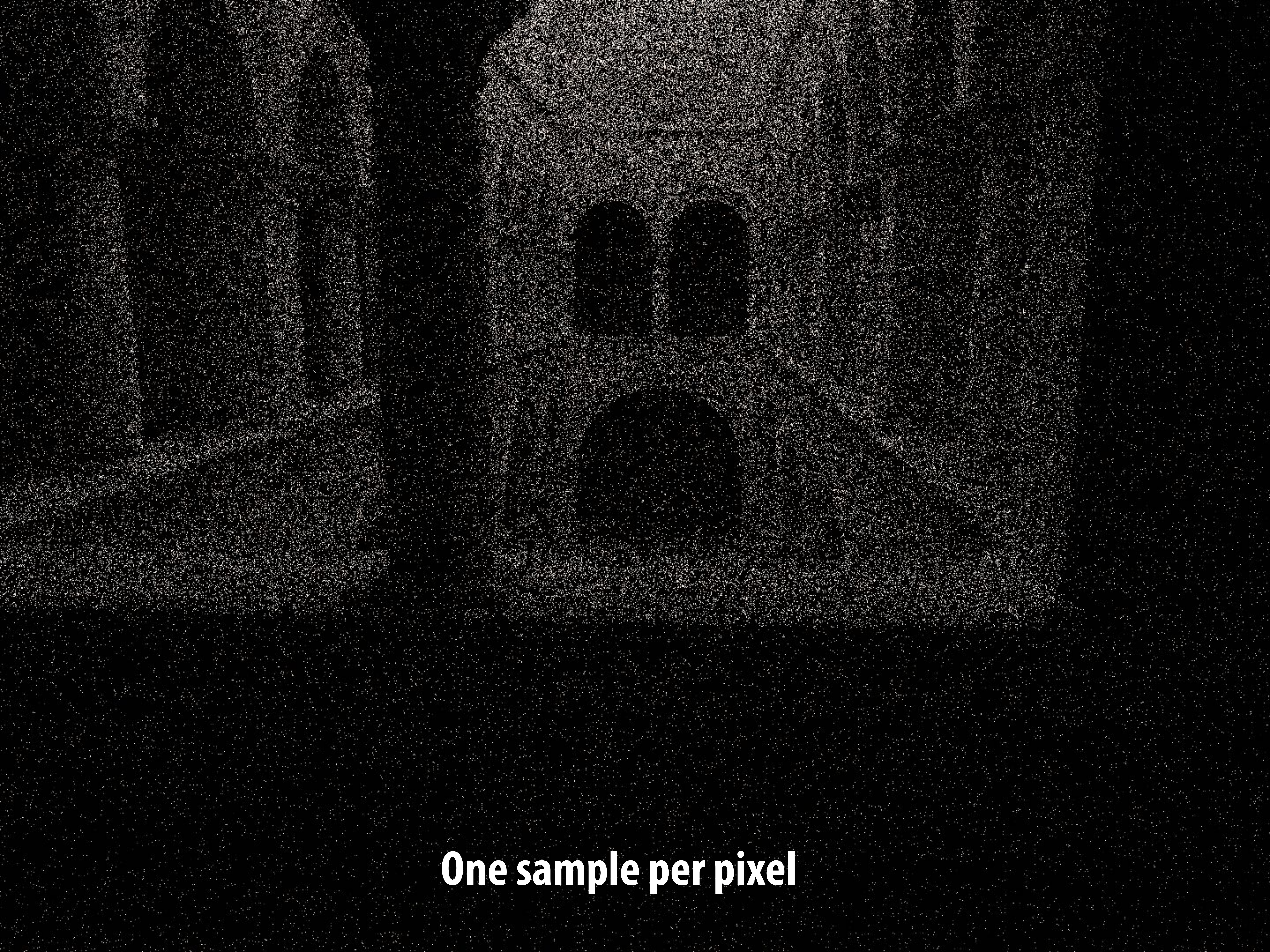


One-bounce global illumination

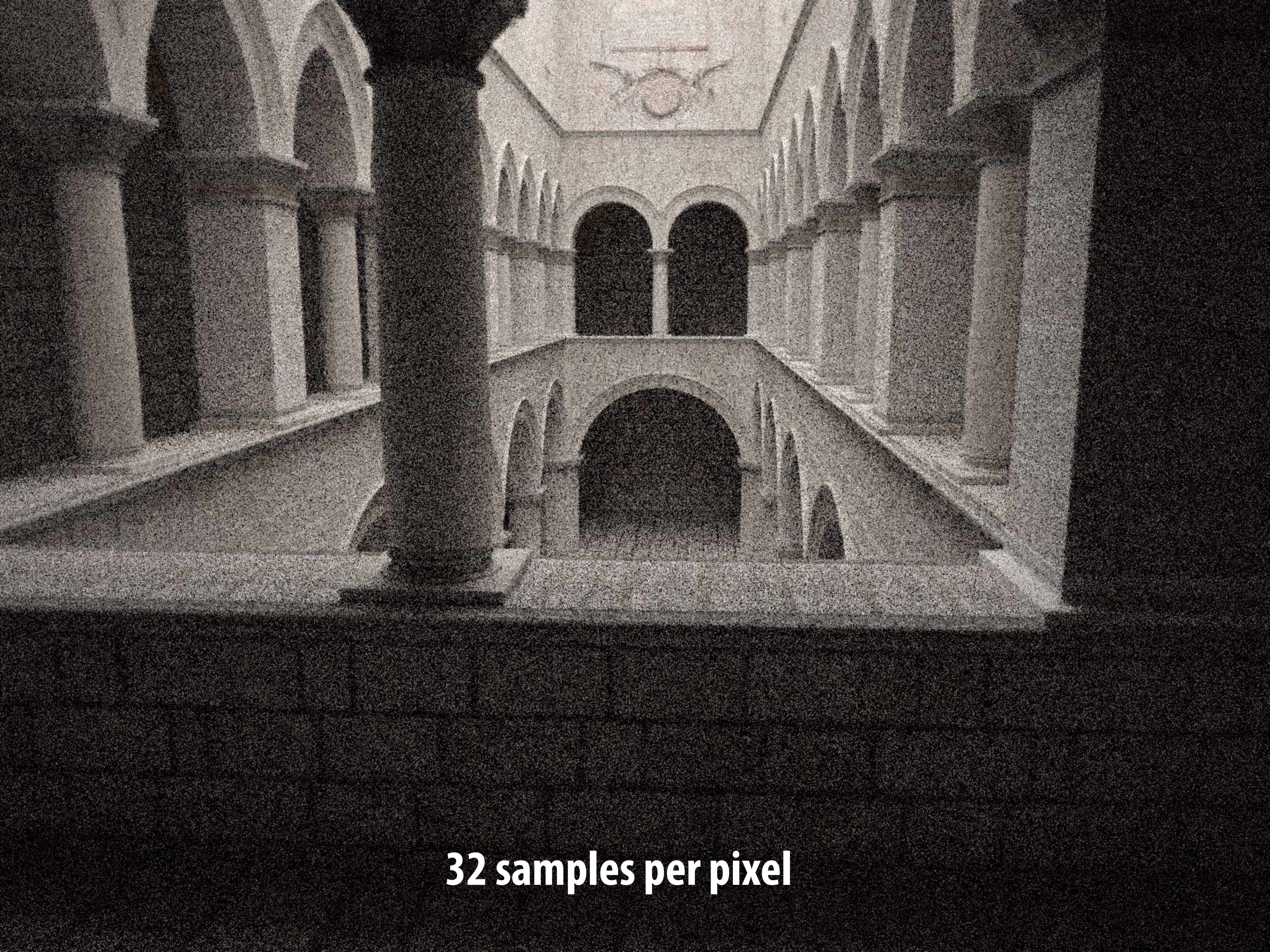


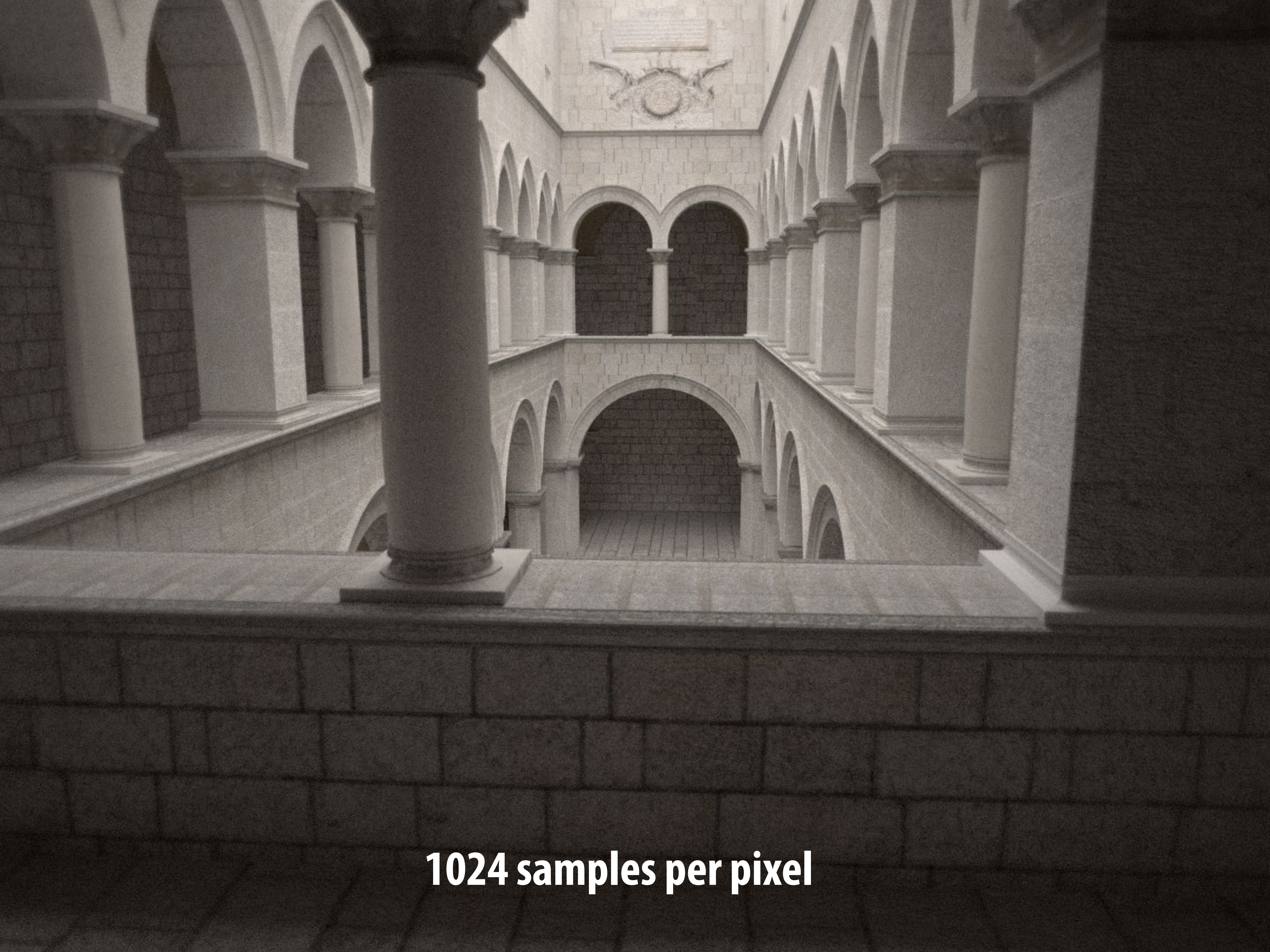
Sixteen-bounce global illumination





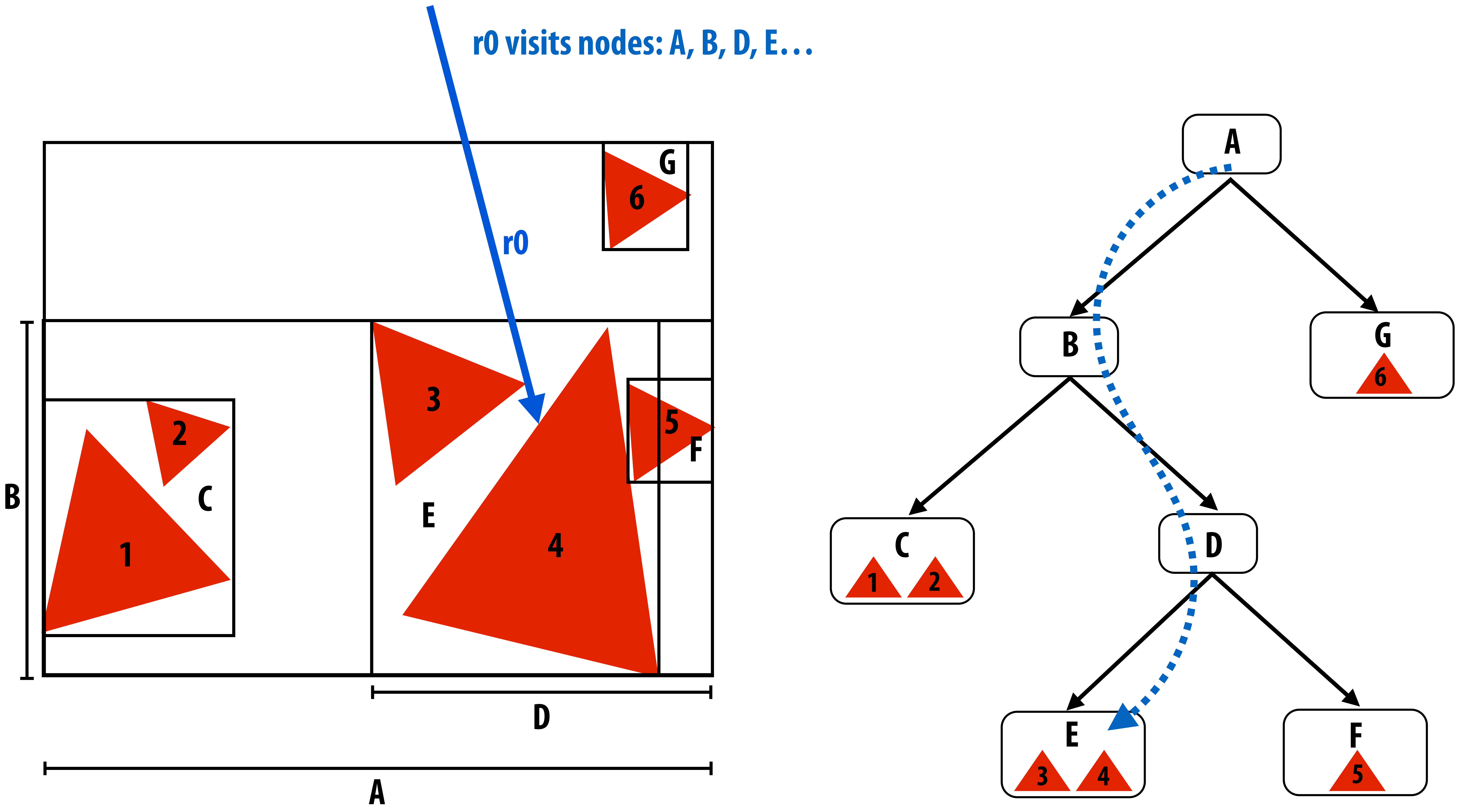
One sample per pixel


32 samples per pixel


1024 samples per pixel

Recall: BVH acceleration structure

r0 visits nodes: A, B, D, E…



Ray tracing performance challenges

1. 3D ray-triangle intersection math is expensive

2. Ray-scene intersection requires traversal through bounding volume hierarchy acceleration structure
   - Unpredictable data access
   - Rays are essentially randomly oriented after enough bounces

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

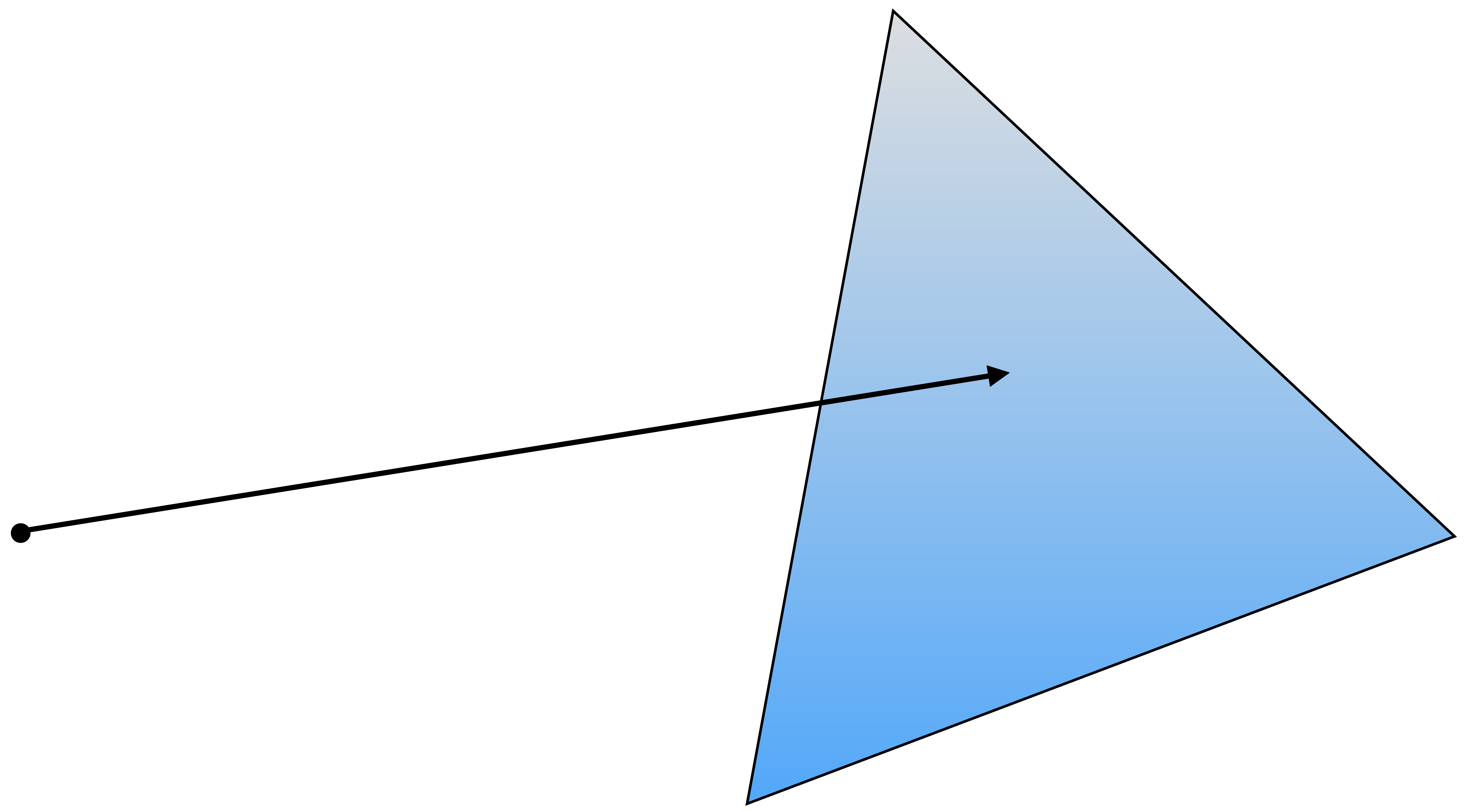
Ray tracing parallelism

Parallelizing ray-scene intersection

- **Parallelize ray tracing across cores (naive data parallel)**
  - **Simultaneously intersect multiple rays with scene**
  - **Enables wide multi-core execution**

Parallelizing single ray-scene queries

(Intra-ray parallelism)



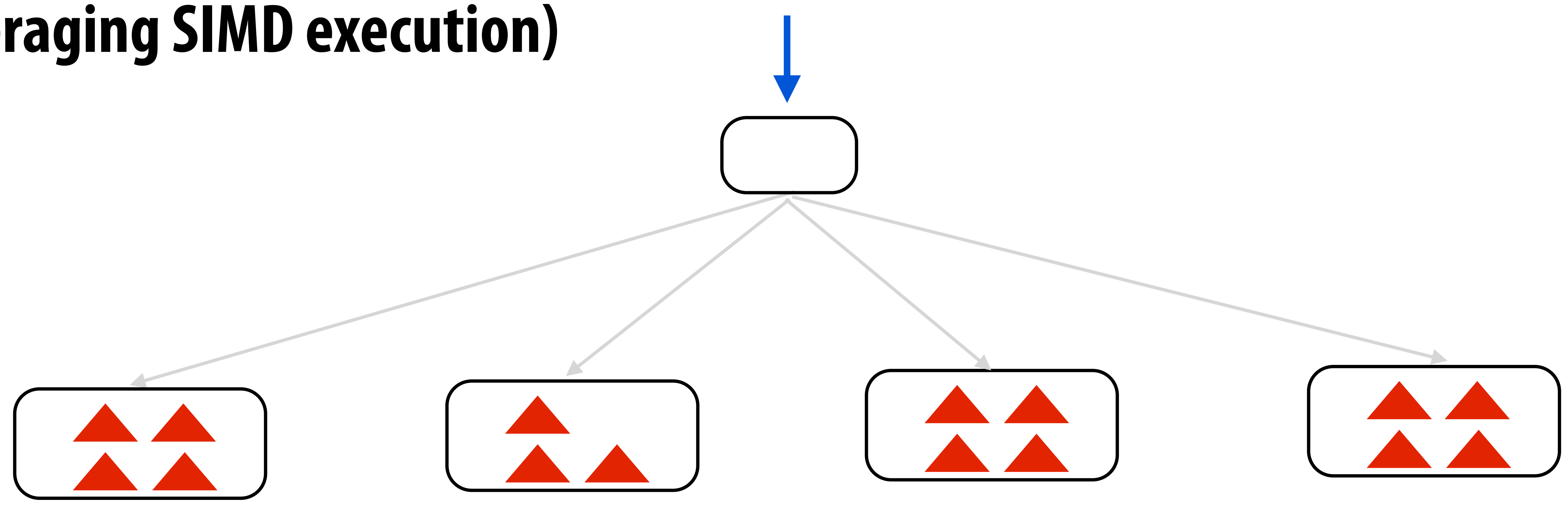
Parallelize ray-box, ray-triangle intersection

- 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]

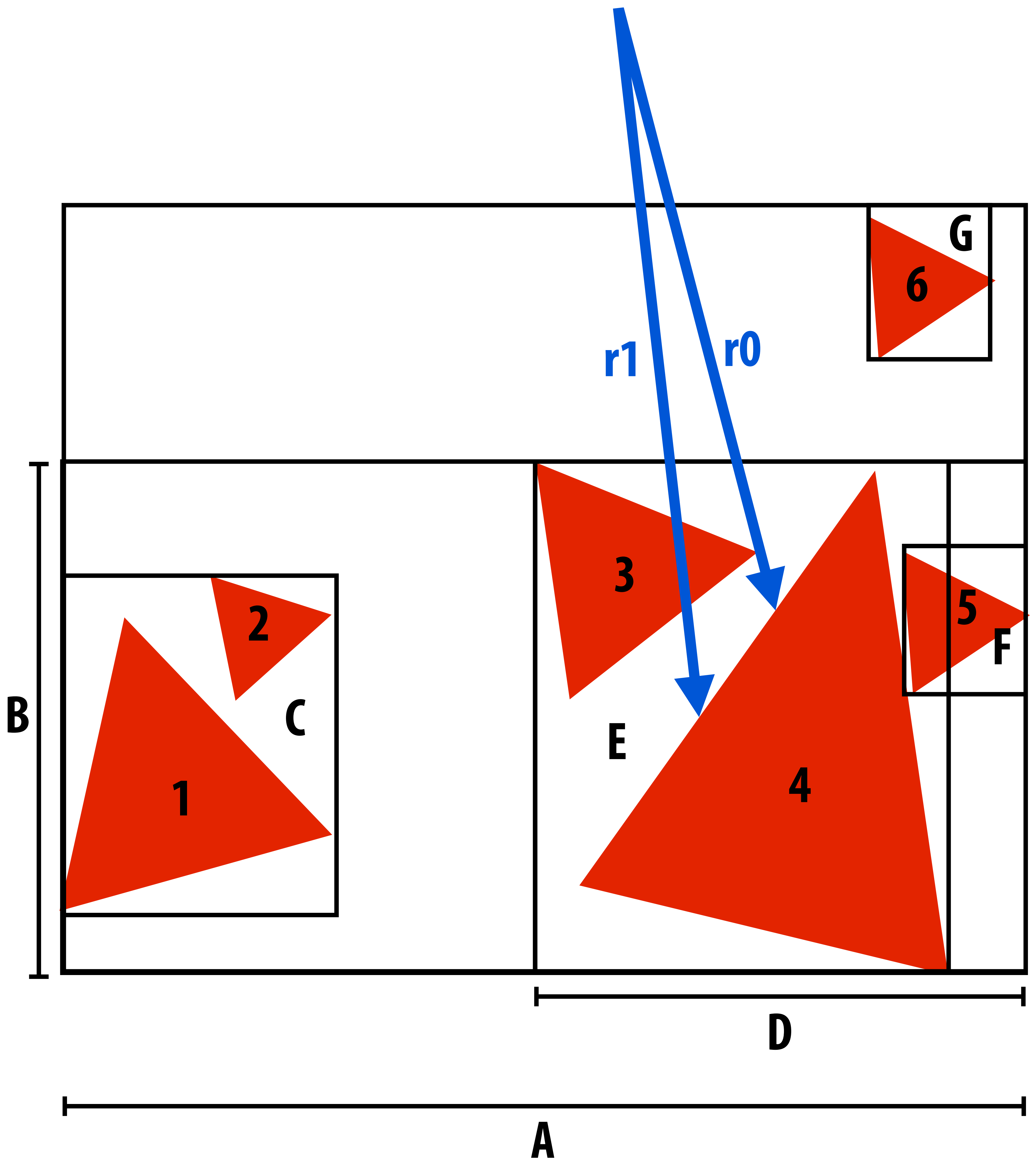
- Idea: change the BVH data structure
- Use wider-branching BVH (test single ray against multiple child node bboxes in parallel)
  - Empirical result: BVH with branching factor four has similar work efficiency as BVH with branching factor two
  - BVH with branching factor 8 or 16 is less work efficient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

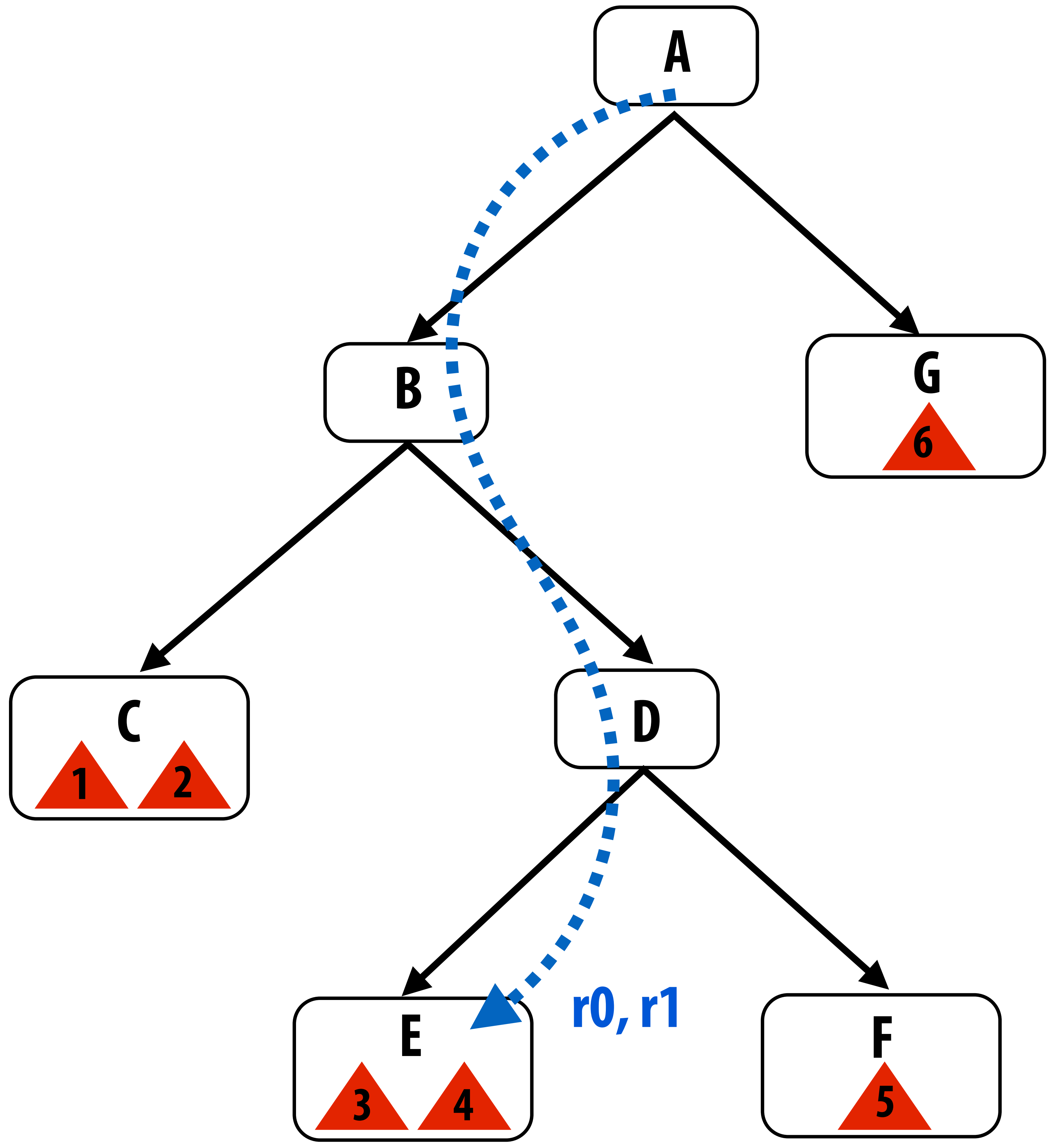


Understanding ray coherence

Ray traversal "coherence"

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

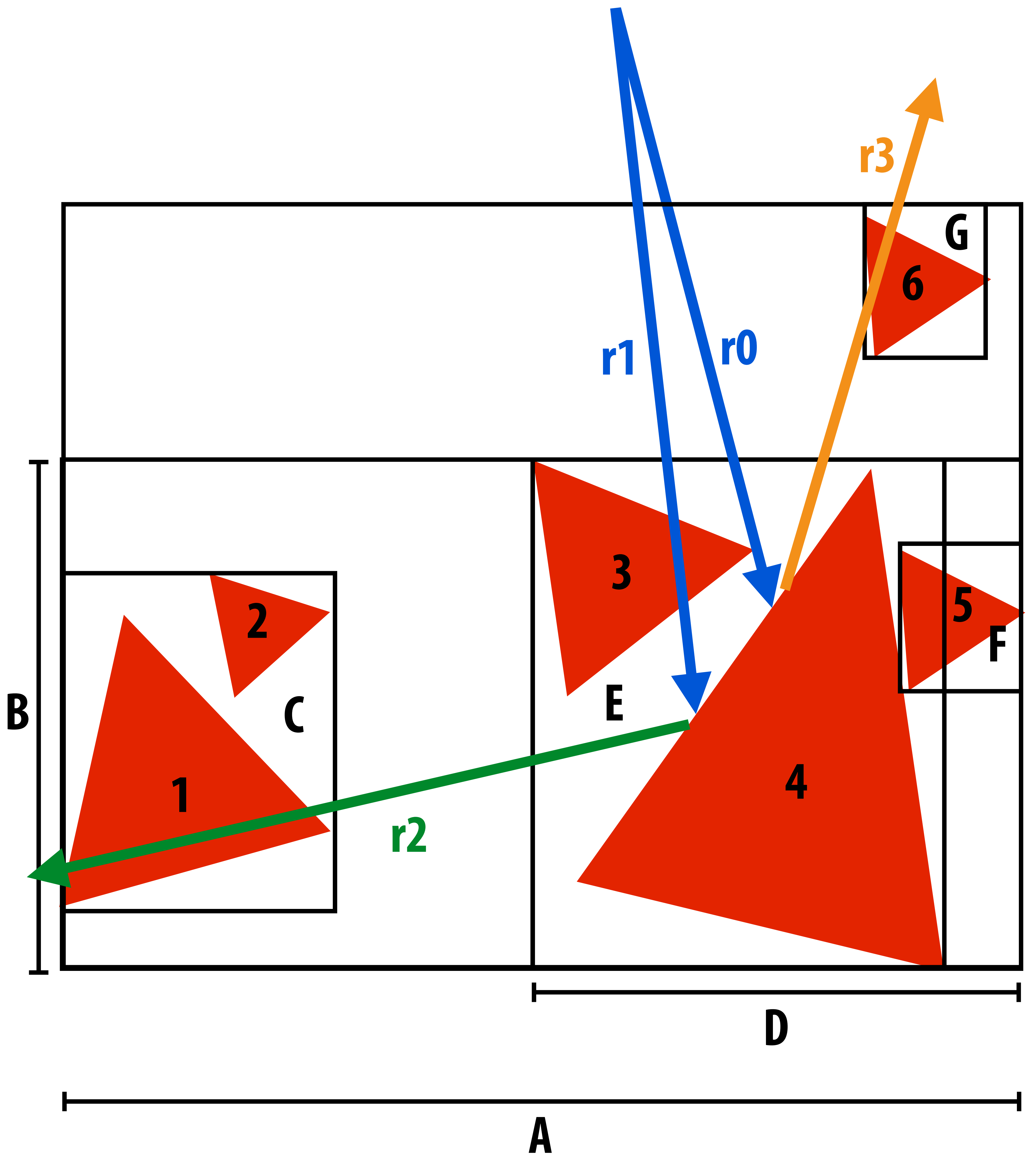


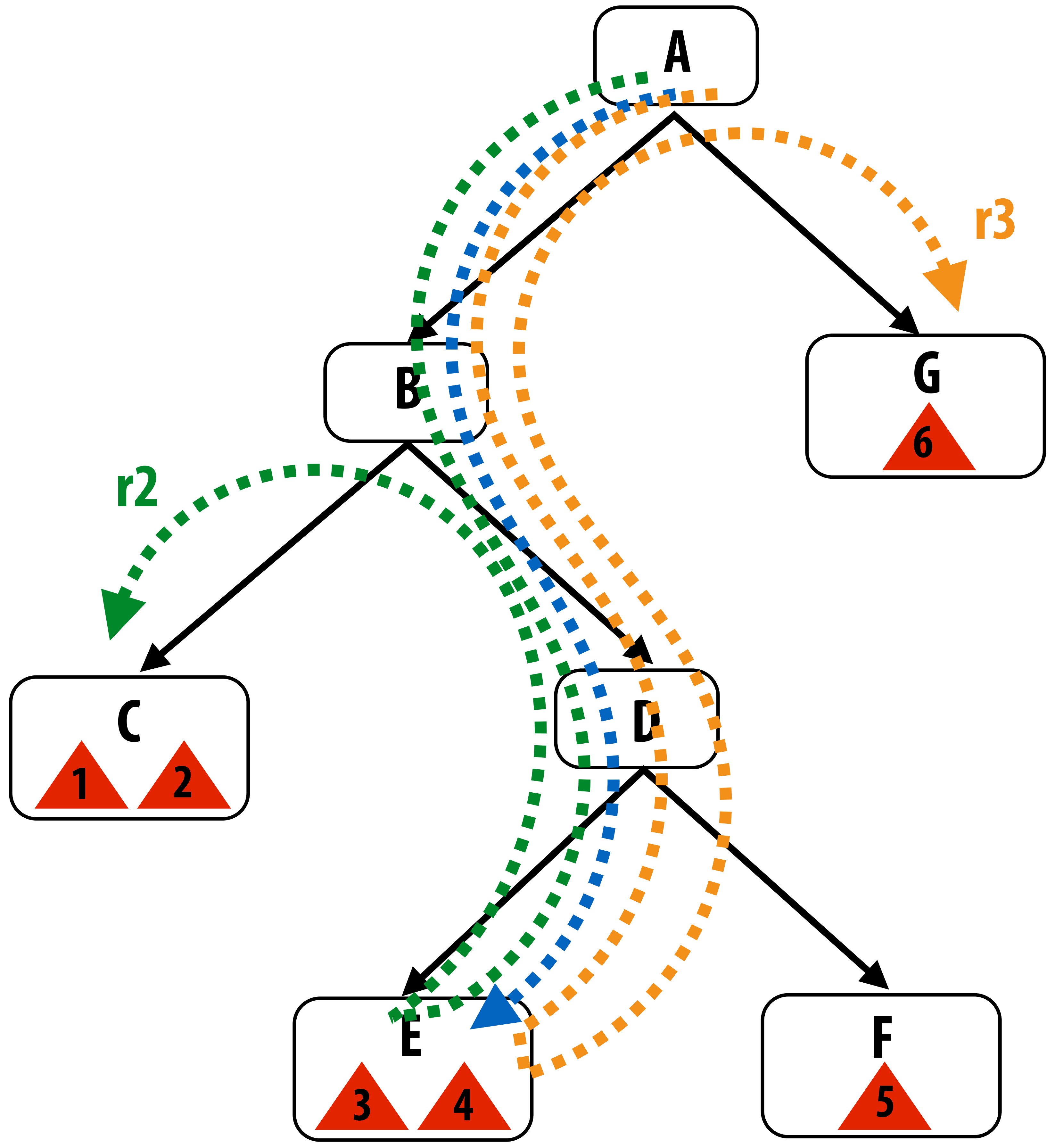


Bandwidth reduction: BVH nodes (and triangles) loaded into cache for computing scene intersection with r0 are cache hits for r1

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…

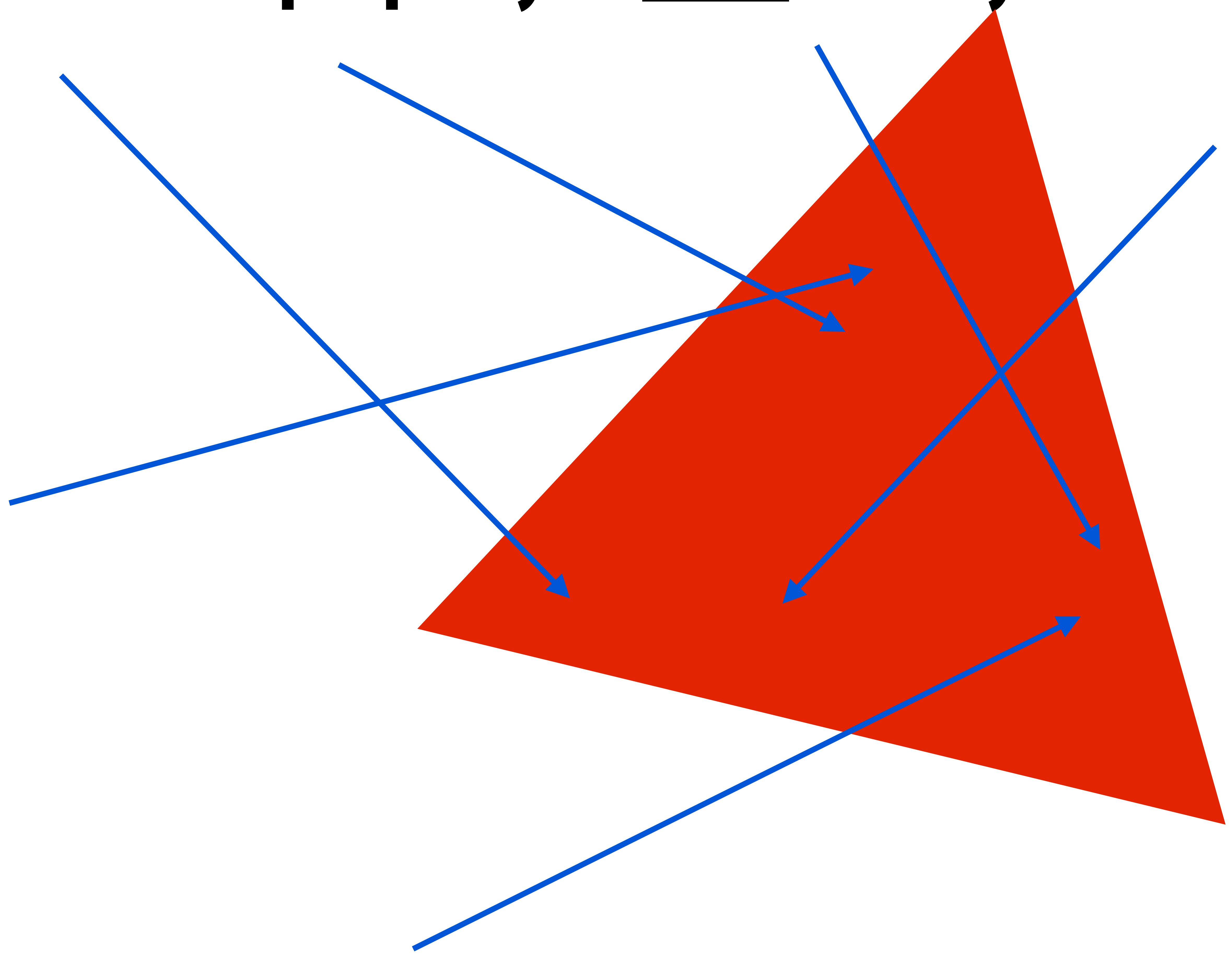




R2 and R3 require different BVH nodes and triangles

Incoherent rays

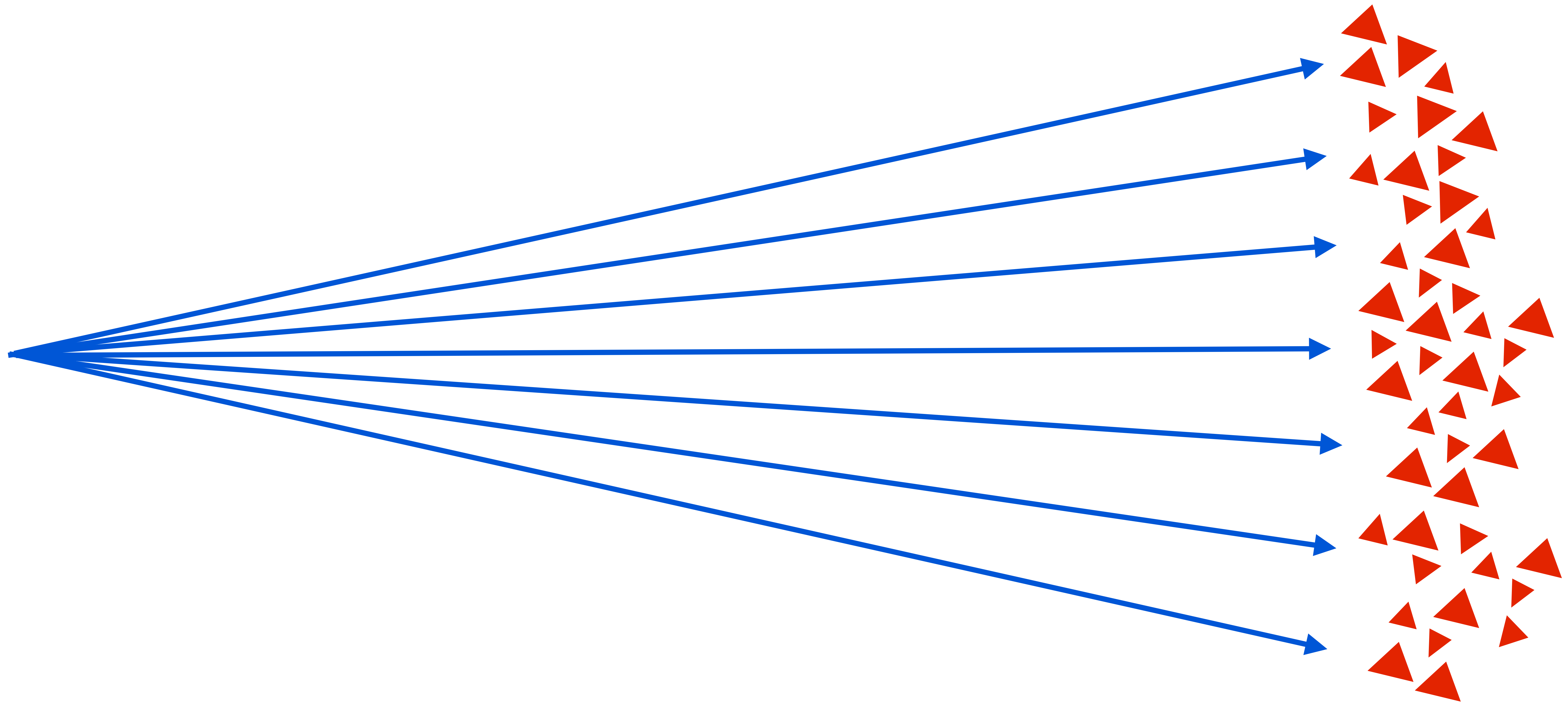
Incoherence is a property of <u>both</u> the rays and the scene



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)

Wide SIMD ray tracing

Ray packet tracing (SIMD)

[Wald et al. 2001]

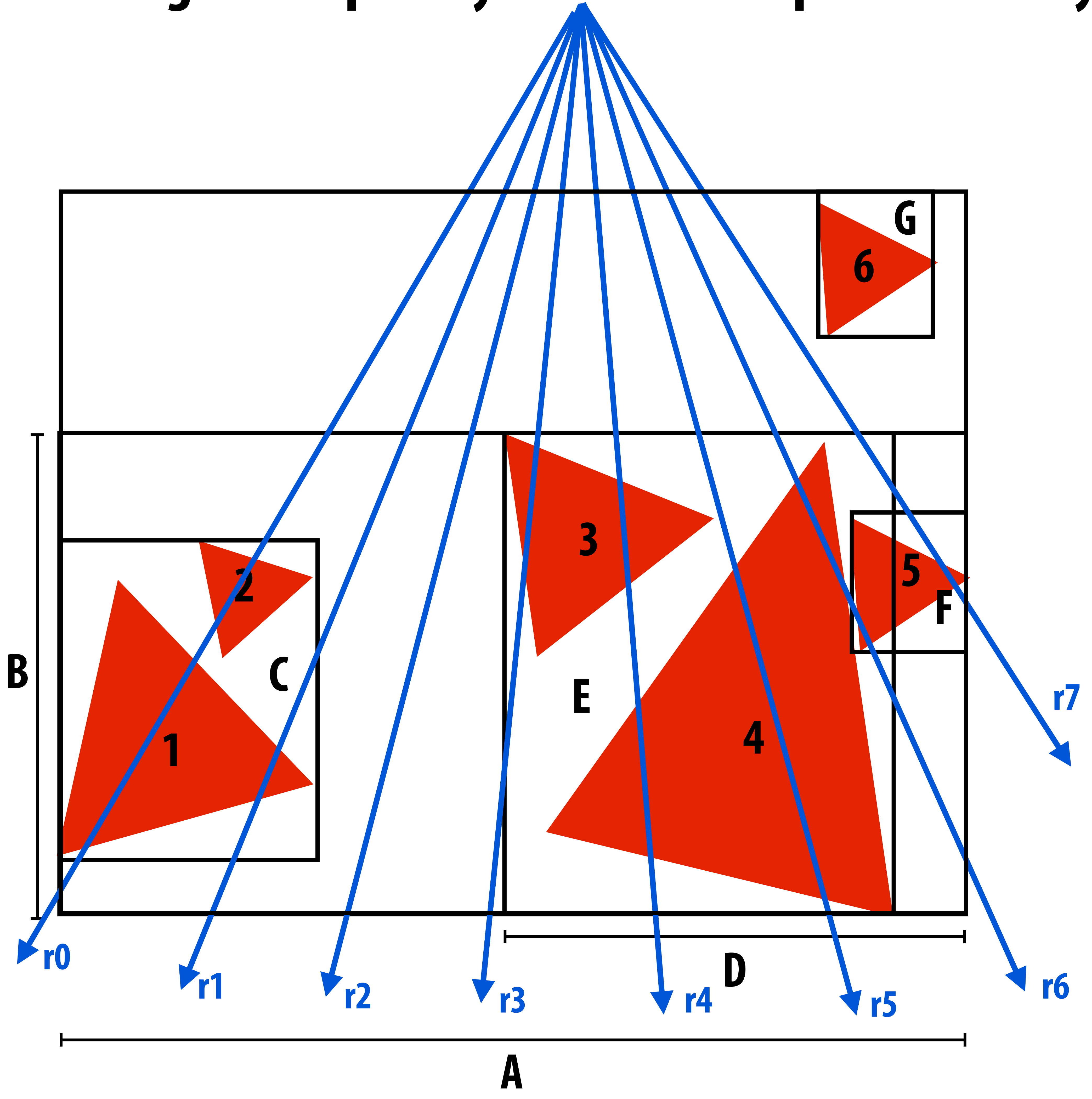
Program explicitly intersects a collection of rays against BVH at once

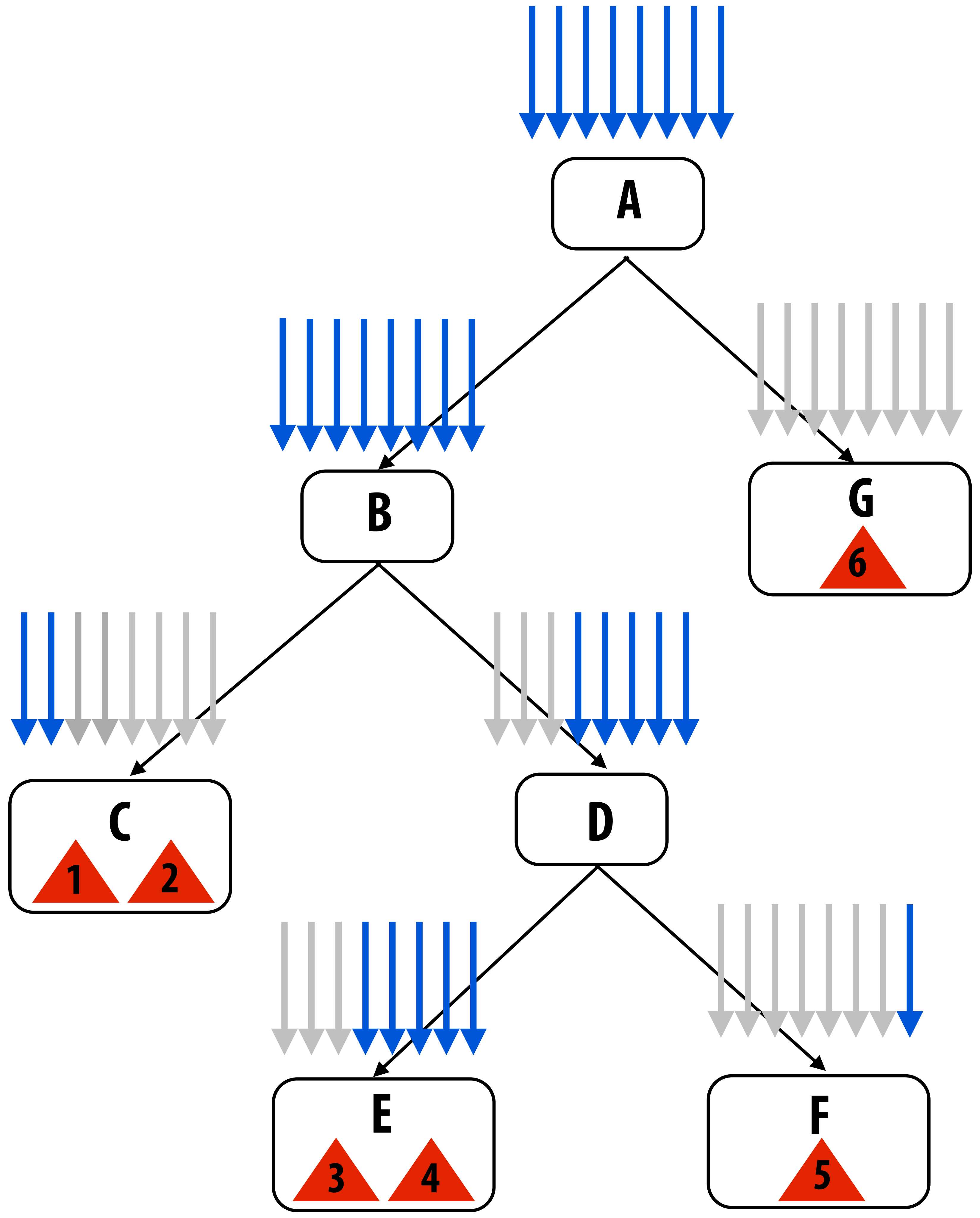
```
RayPacket
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

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

Program explicitly intersects a "packet" of rays against BVH at once





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

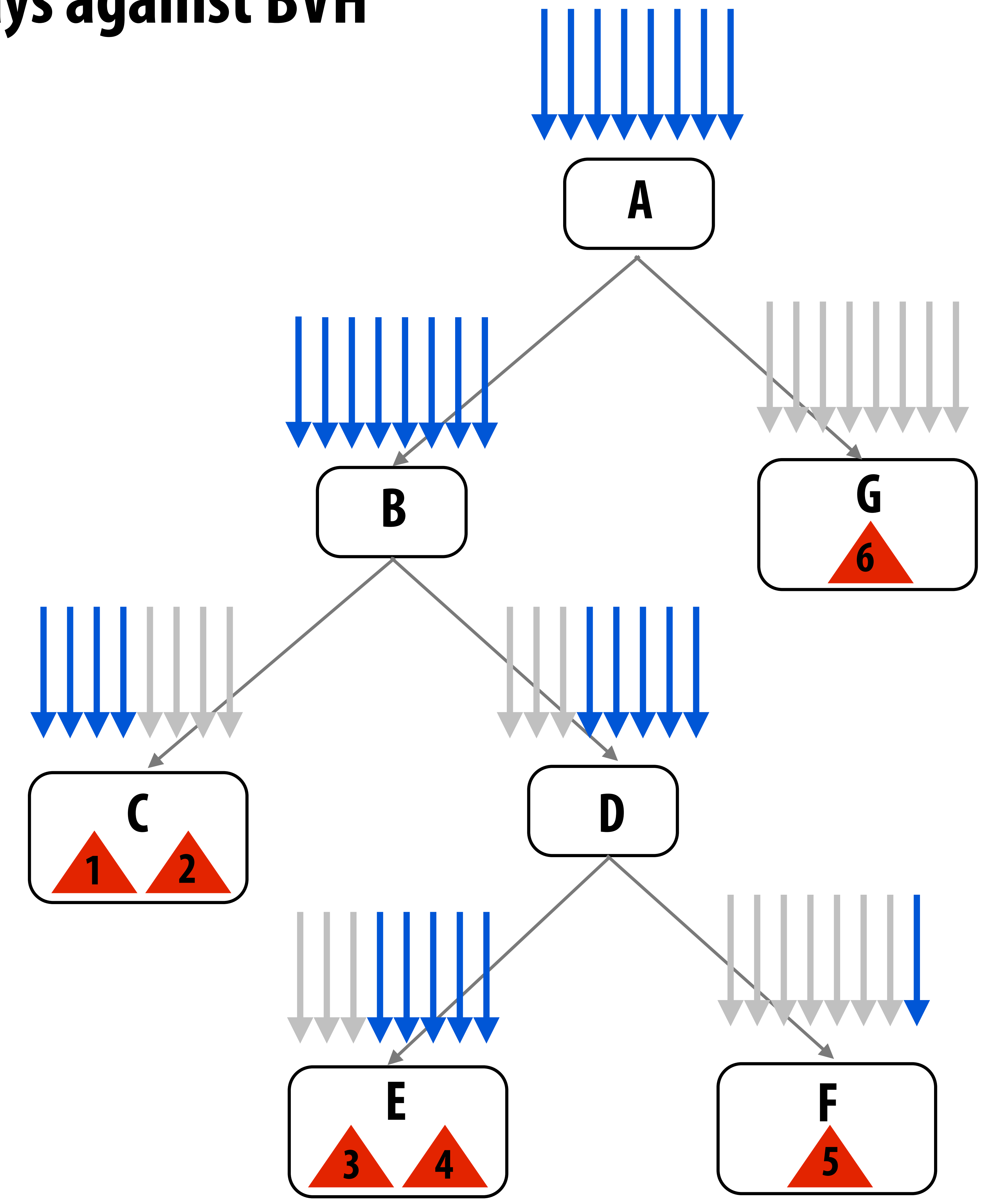
- Enables wide SIMD execution
  - One vector lane per ray
- Amortize BVH node data fetch: all rays in packet visit BVH node at same time
  - Load BVH node once for all rays in packet (not once per ray)
  - Note: because of this, there is value to making packets bigger than SIMD width! (e.g., size = 64)
- 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

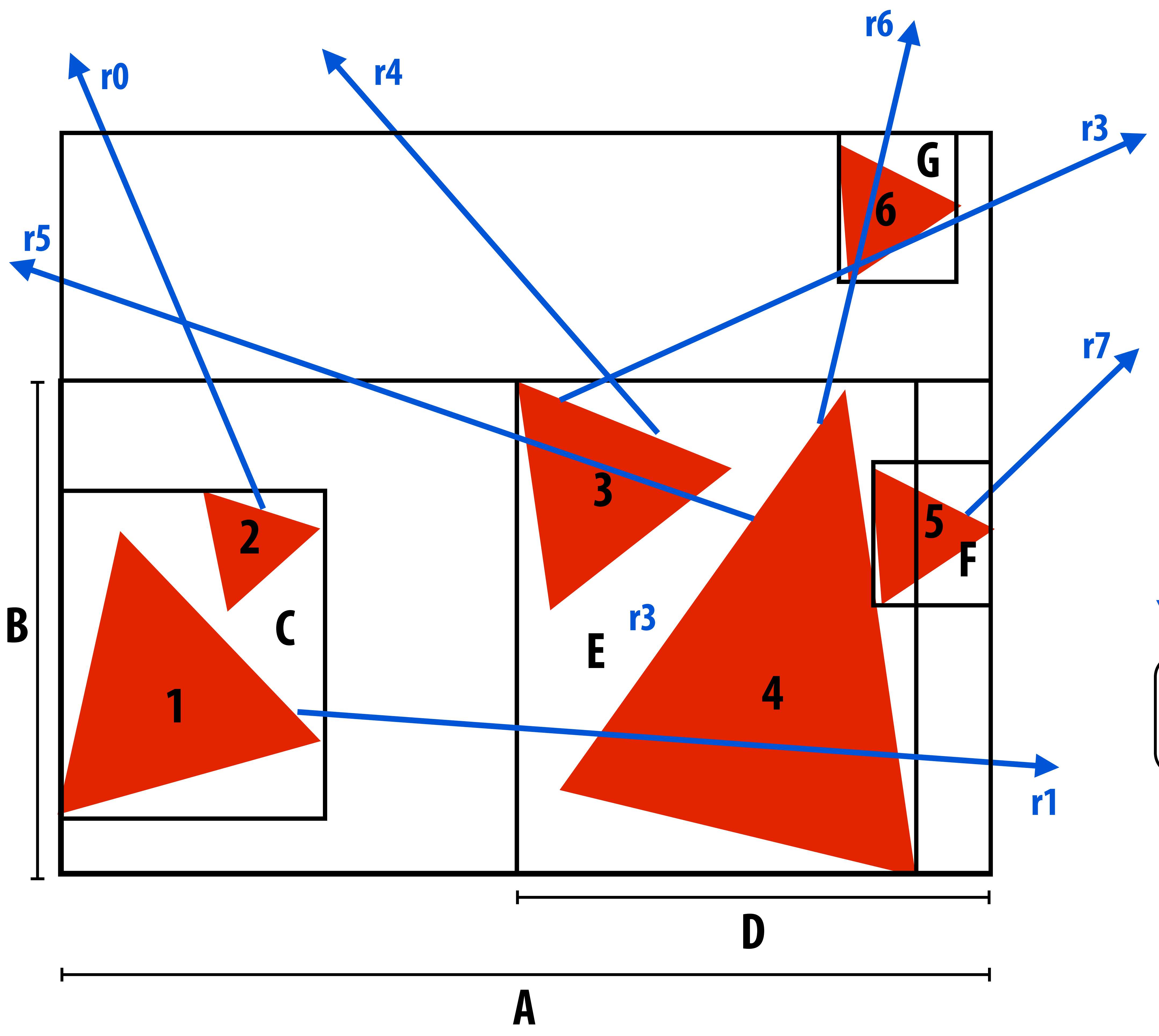
Blue = active ray after node box test

Program explicitly intersects a collection of rays against BVH at once

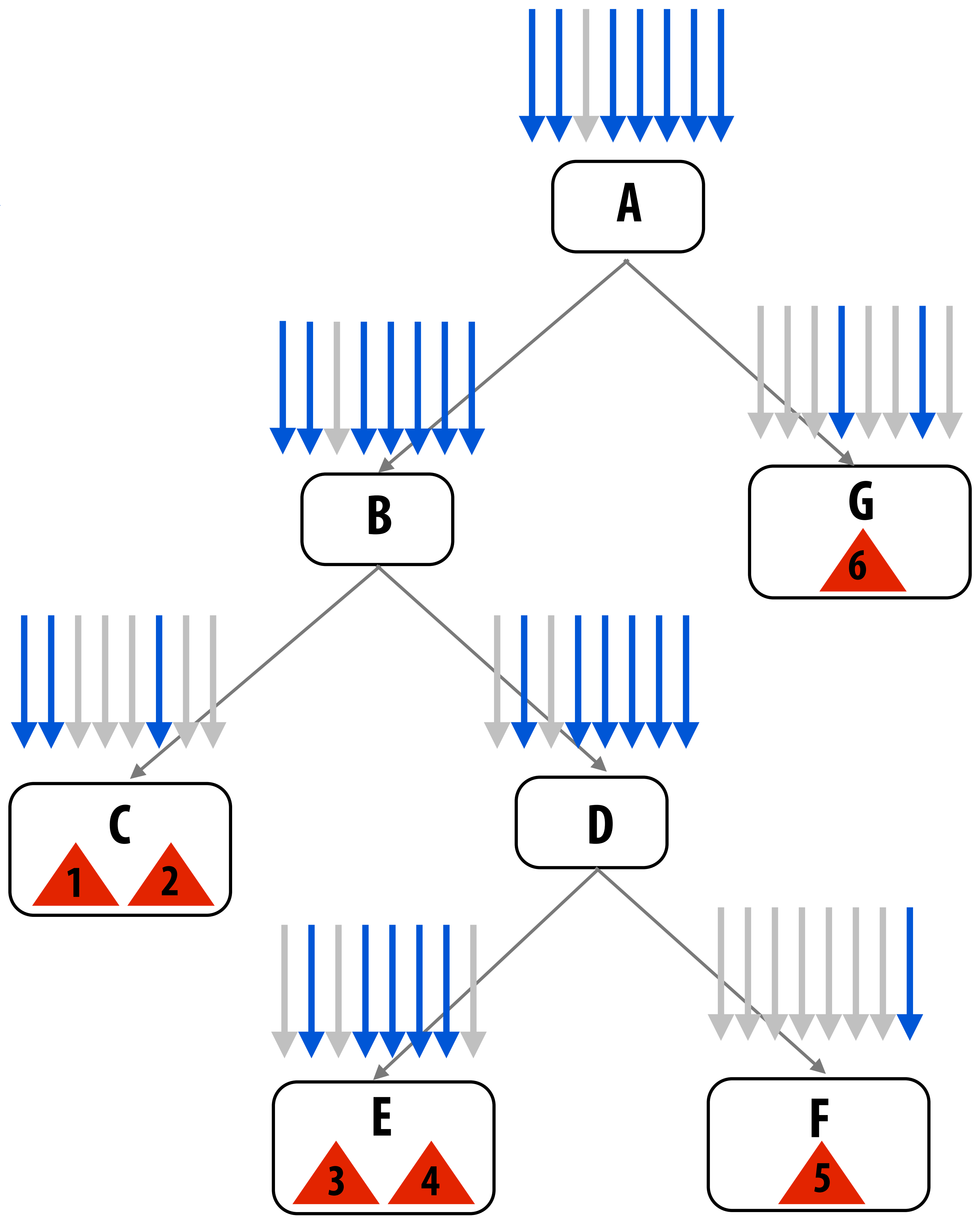
- If any ray must visit a node, it drags all rays in the packet along with it)
- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays
- Not all SIMD lanes doing useful work (SIMD divergence)



Ray packet tracing: incoherent rays



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!)

SPMD ray tracing (GPU style)

No packets!

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

Each worker does no "extra traversal". Each accesses only BVH nodes it needs

Is there SIMD divergence? Where is it?

Algorithm 1

```
stack<BVHNode> to_visit;
to_visit.push(BVH_root_node);
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
   }
}
```

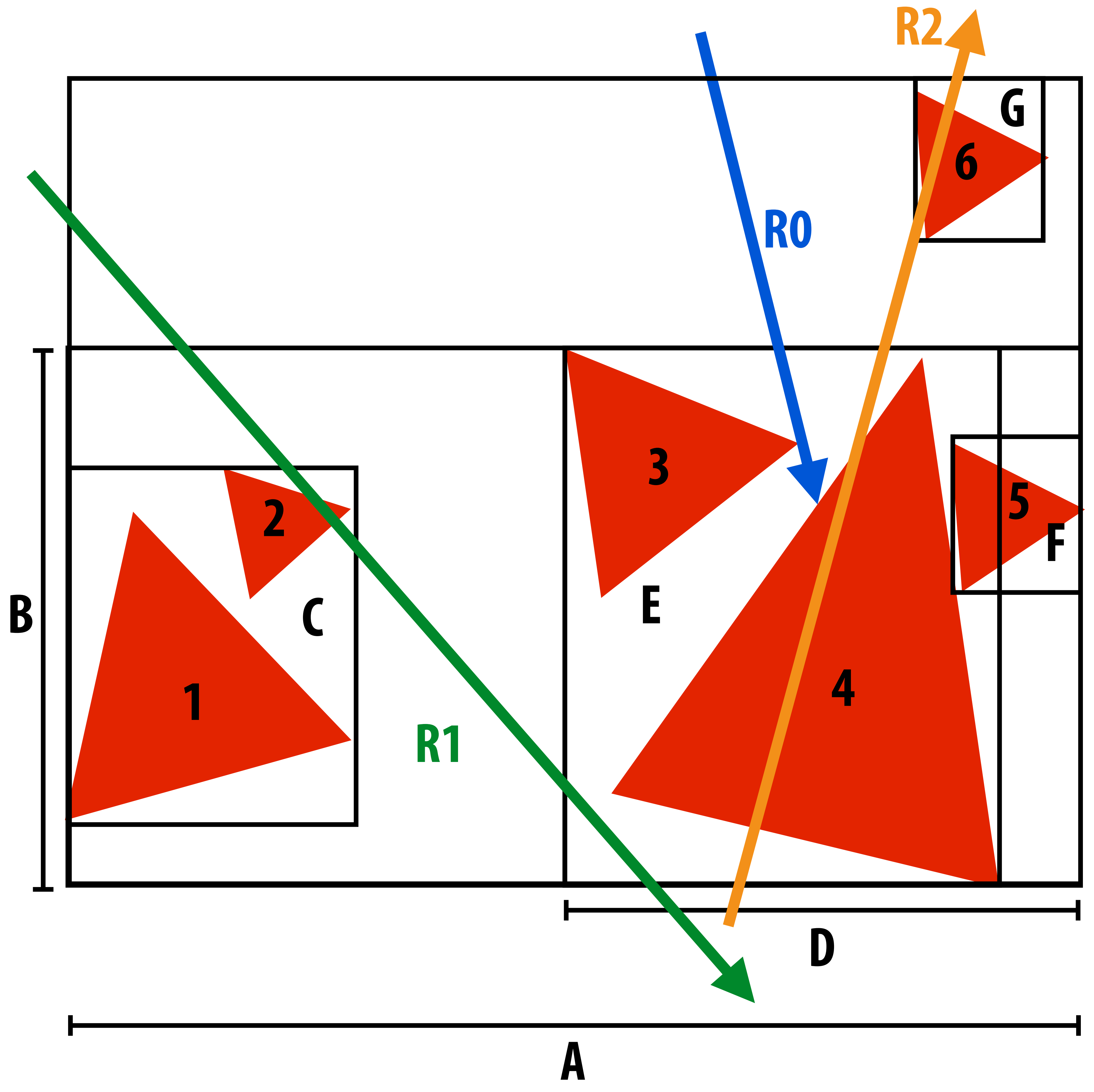
Algorithm 2

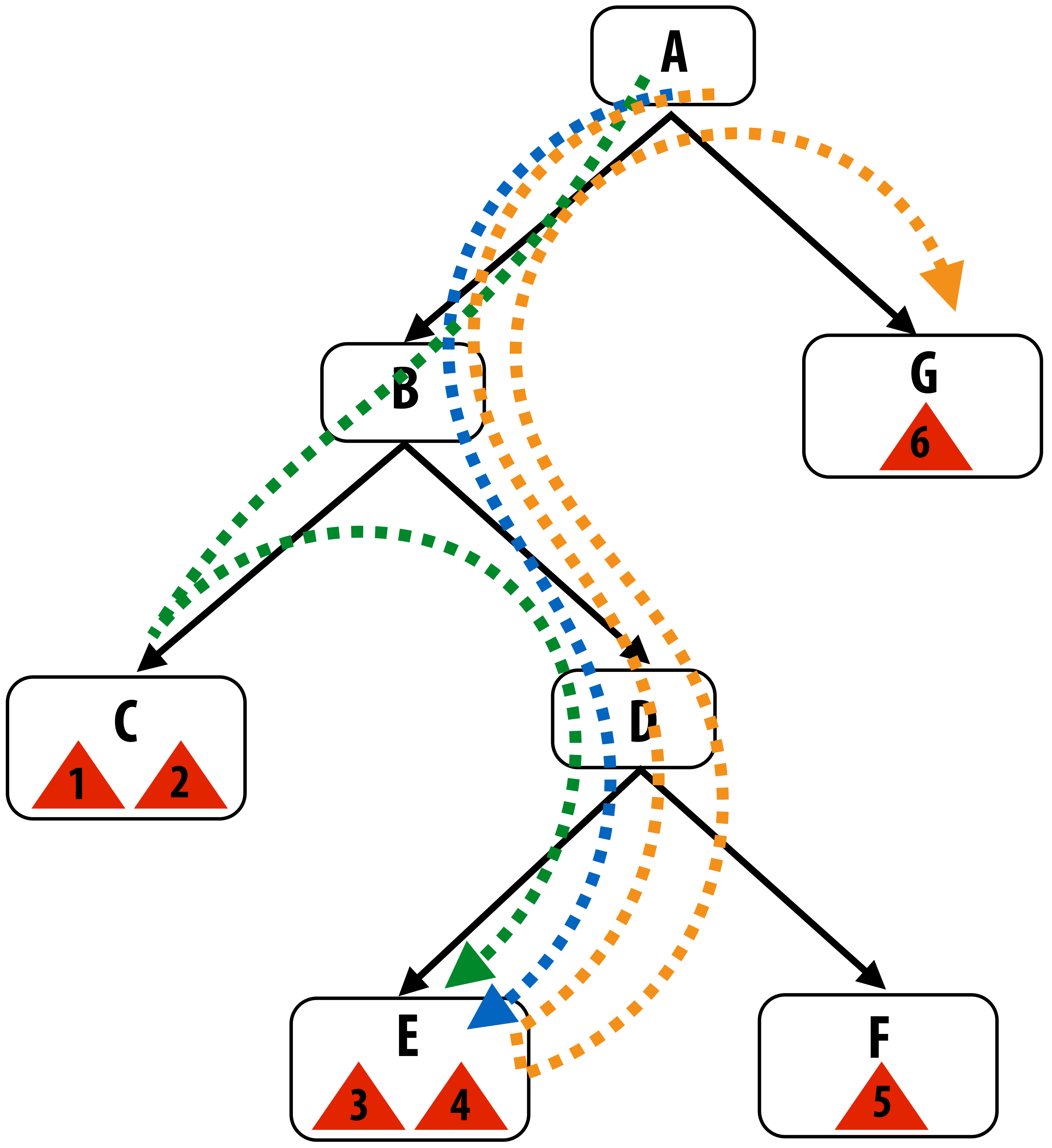
```
stack<BVHNode> to_visit;
to_visit.push(BVH_root_node);
while (ray not terminated) {
   node = to_visit.pop();
   if (node is not a leaf) {
      traverse node // perform ray-box test,
                    // push children to stack
   }
   else (not done testing tris in leaf) {
      ray-triangle test
   }
}
```

Incoherent rays = bandwidth bound

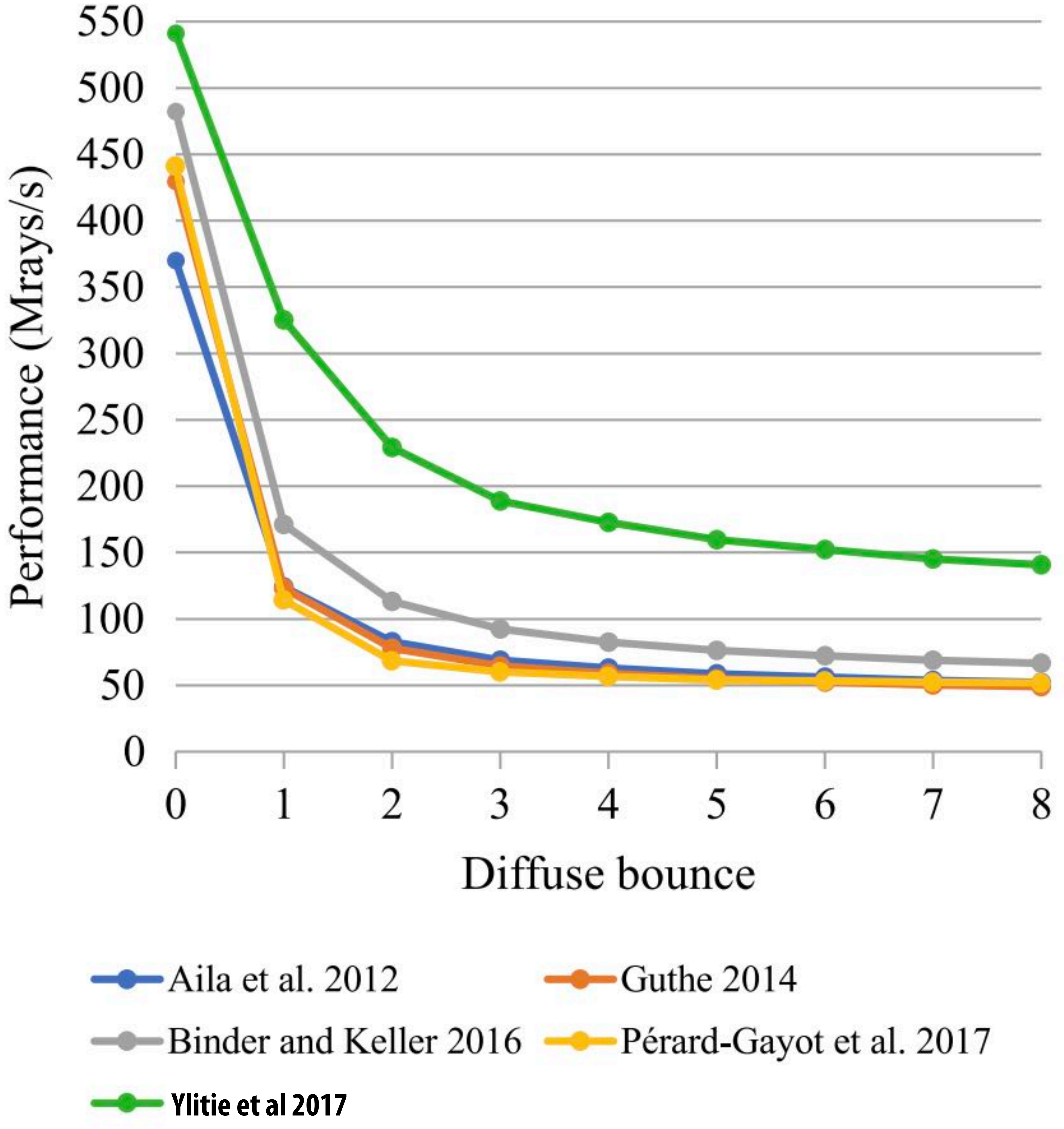
Different threads may access different BVH nodes at the same time:

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



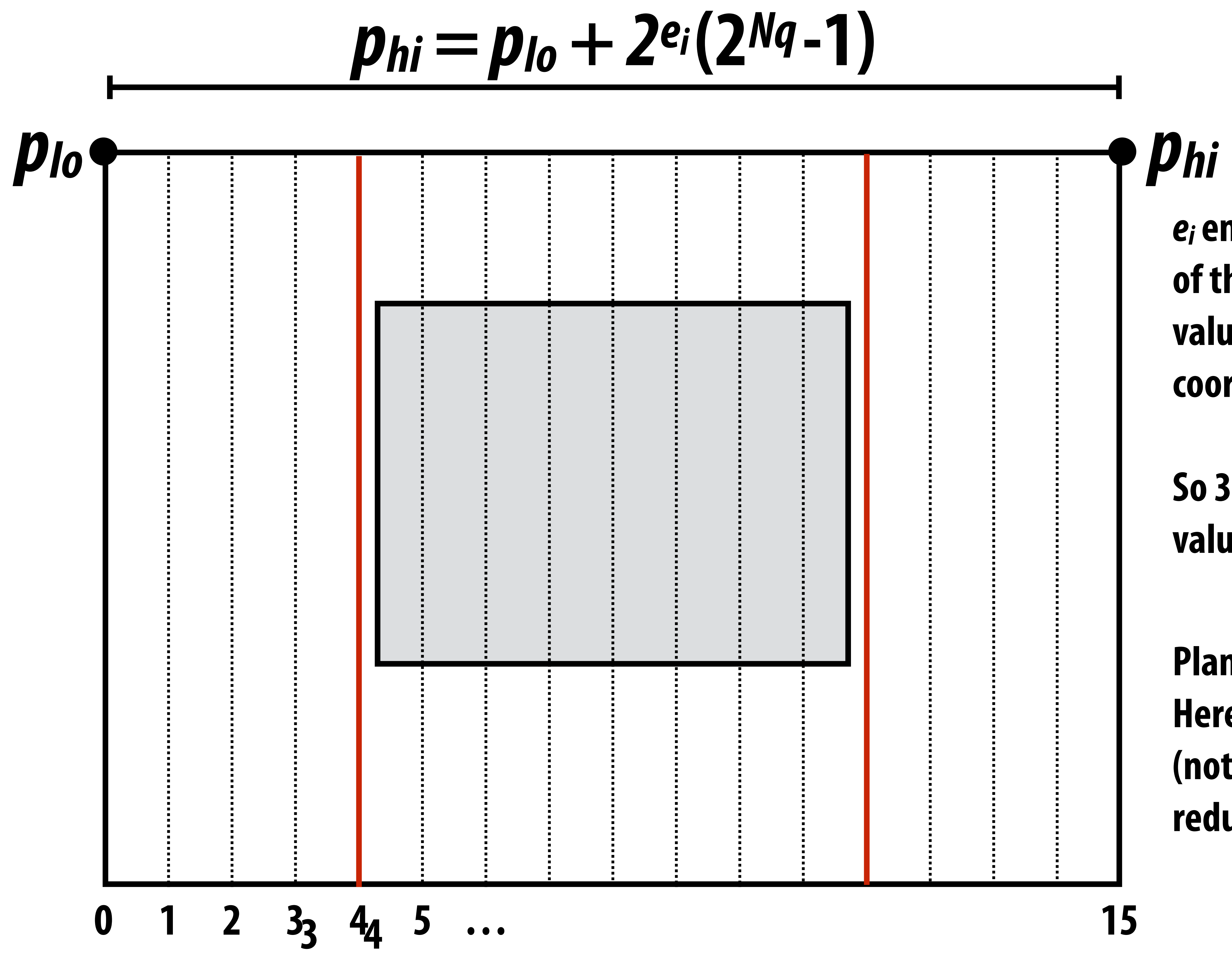


Ray throughput decreases with increasing numbers of bounces (aka increasing ray incoherence)



BVH compression

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



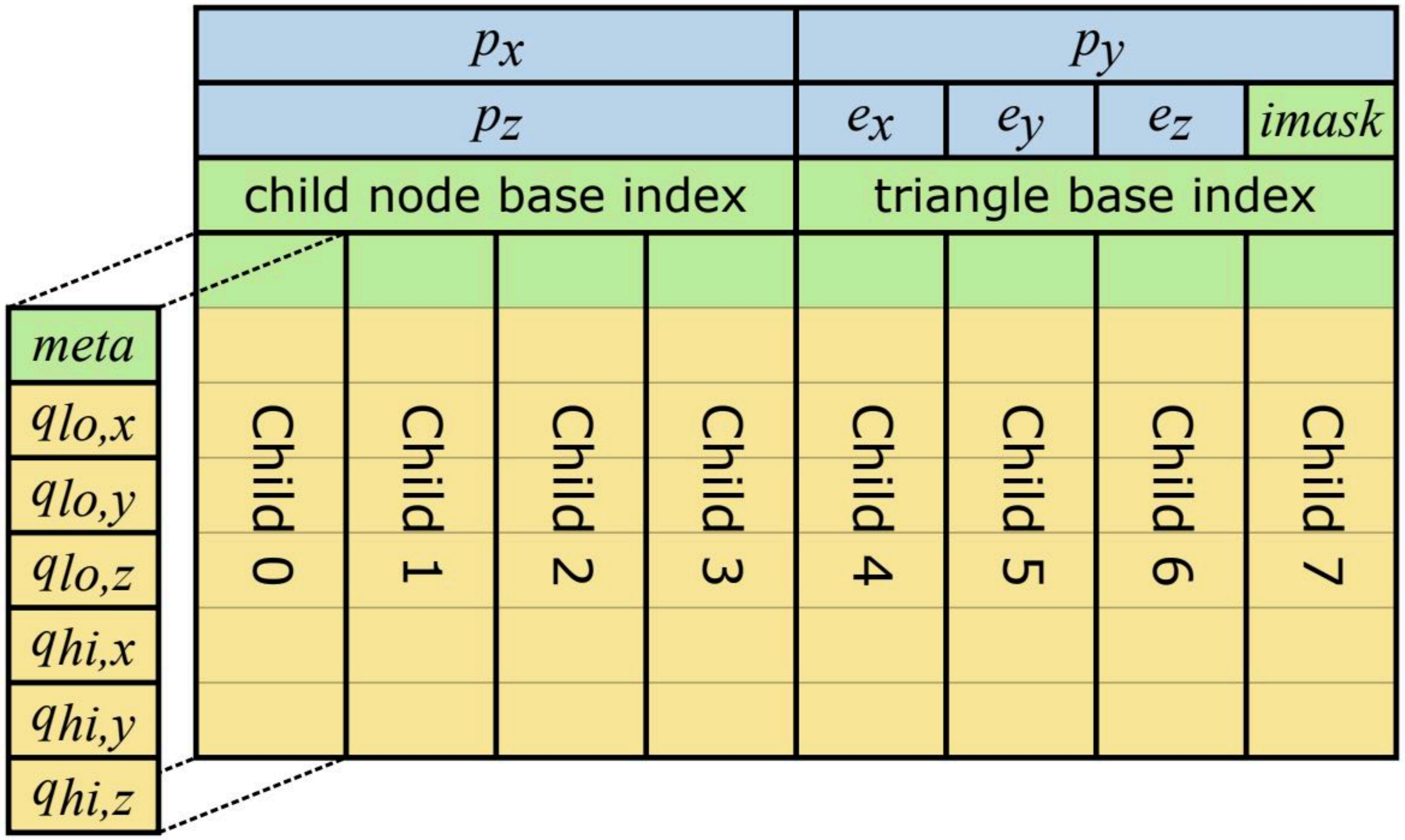
e_i encodes 8 bit exponent that defines "scale" of the parent bbox so that quantized N_q-bit values can be used to represent points in local coordinate frame

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

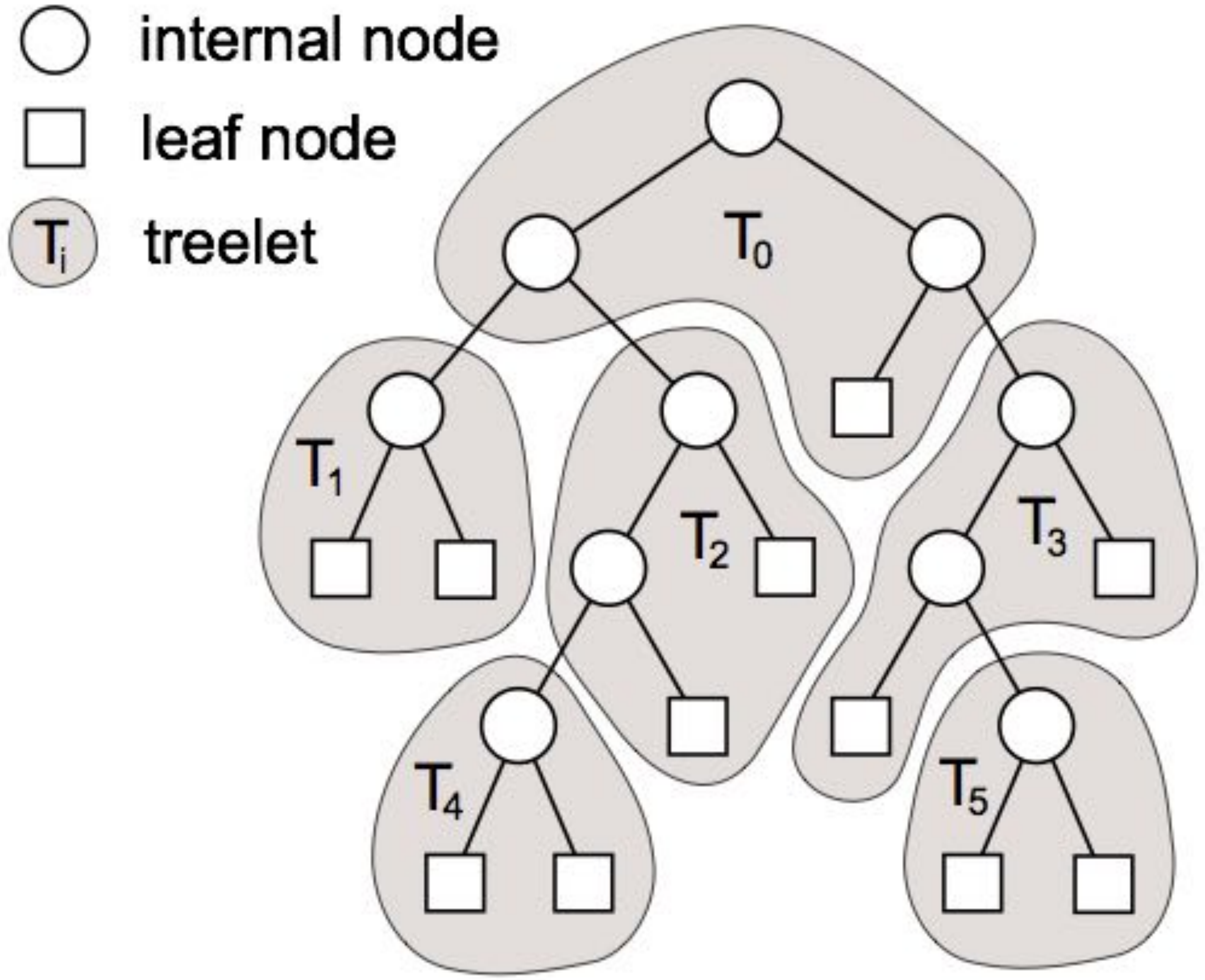


Amortized 10 bytes per child
(3.2x compression over standard BVH formats)

Queue-based global ray reordering

[Pharr 1997, Navratil 07, Alia 10]

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)

Understanding ray coherence

Ray incoherence impacts shading

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

Consider implications for SIMD processing

Perfect specular reflection material

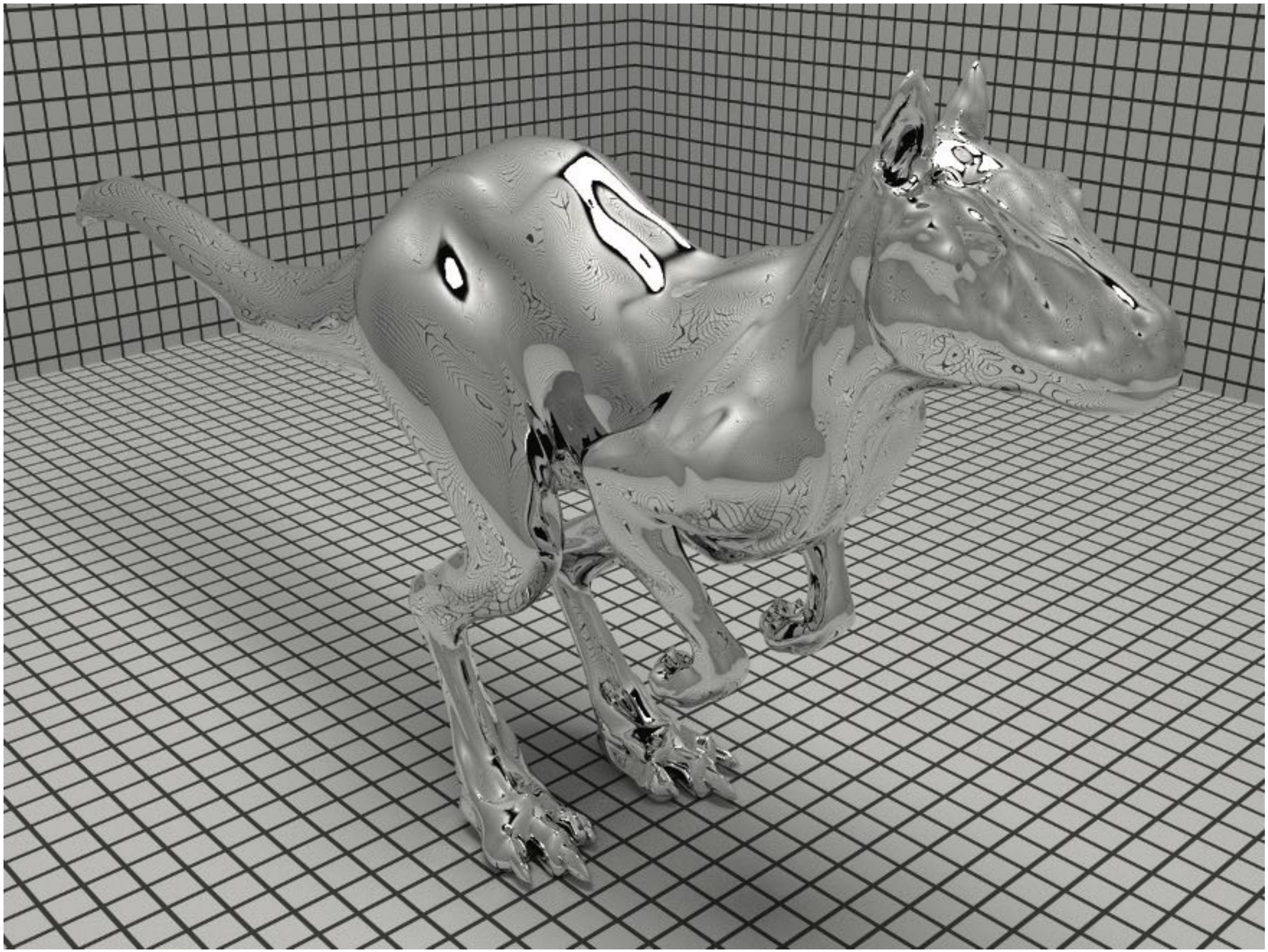
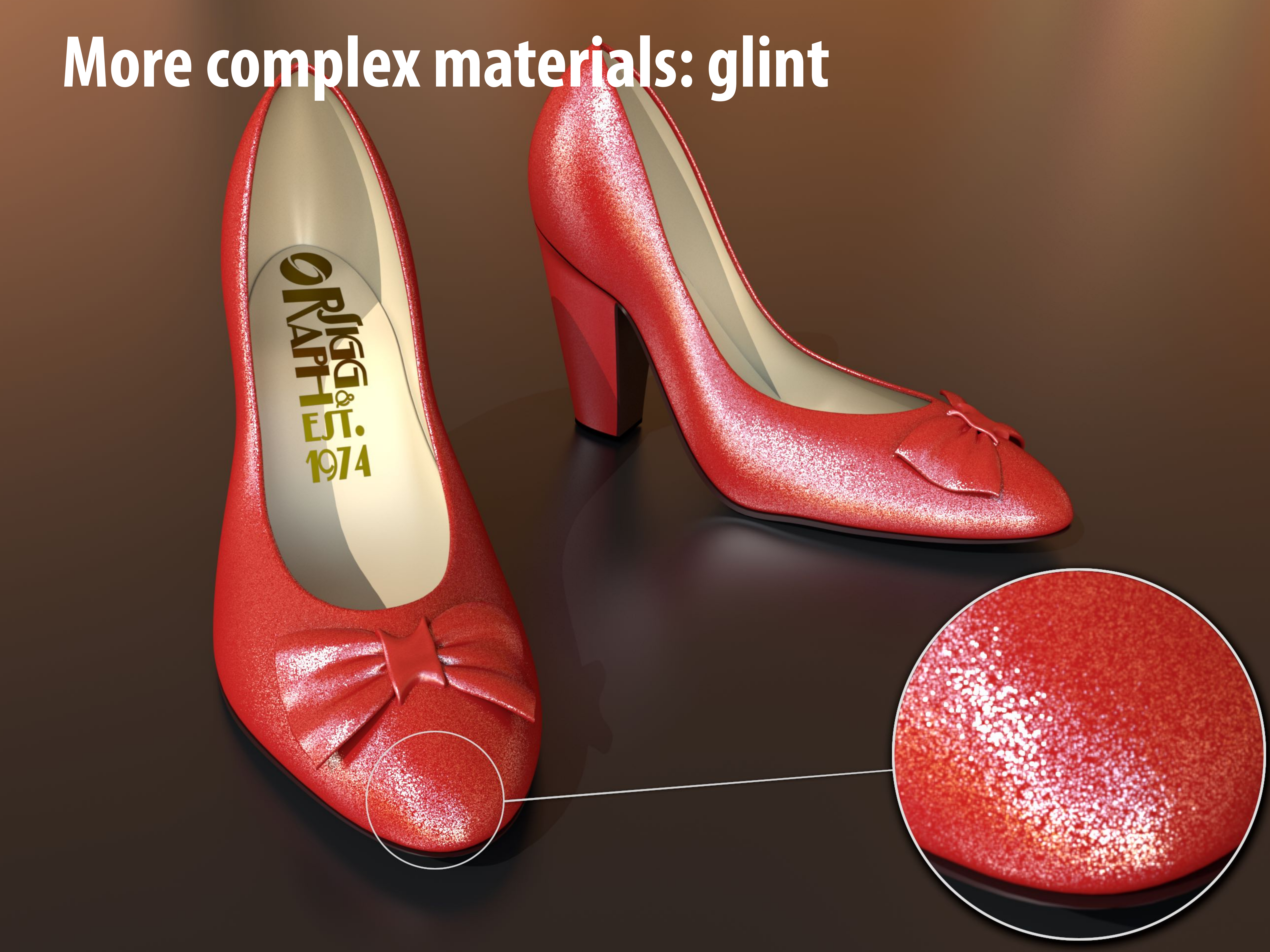
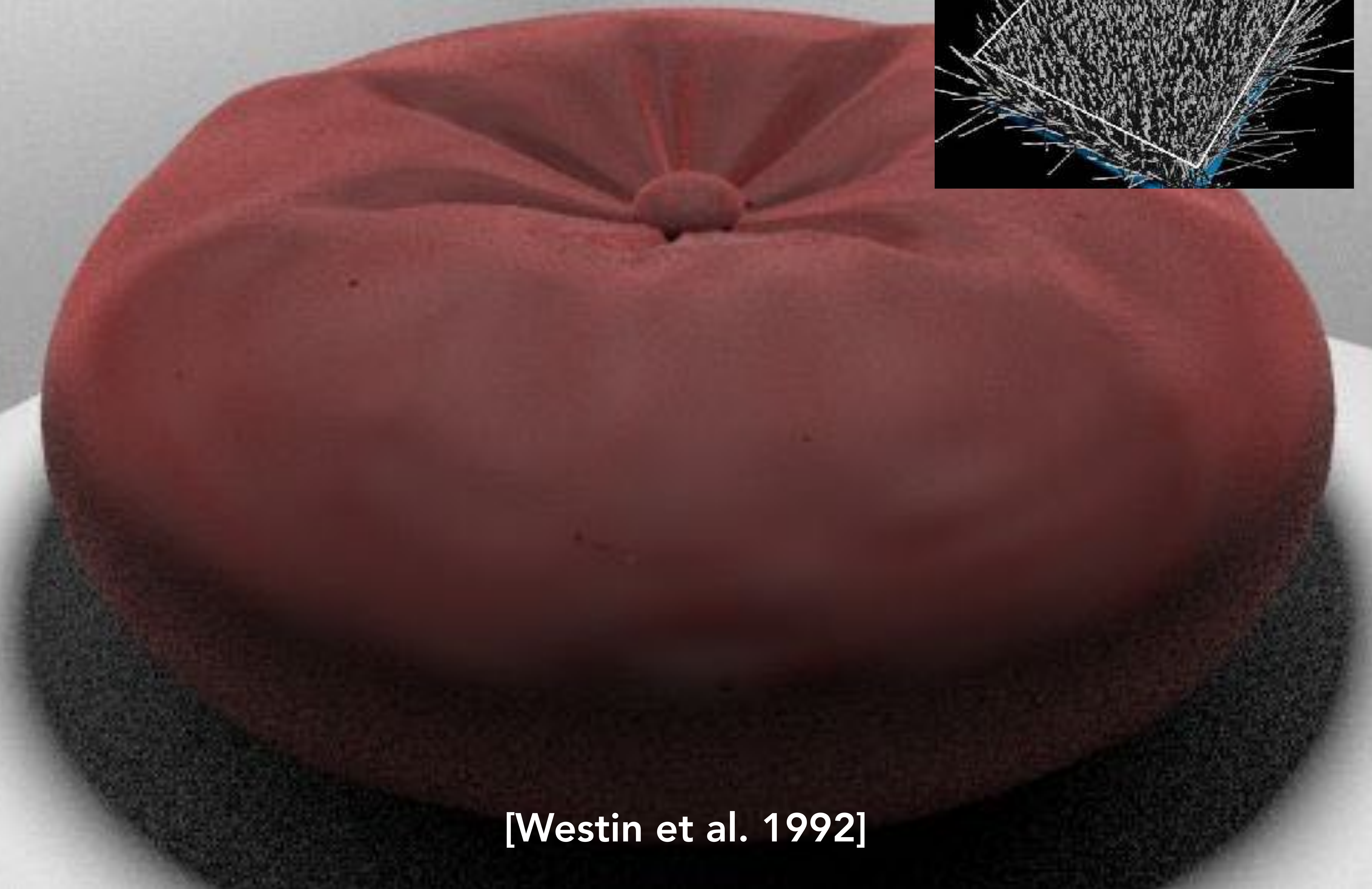


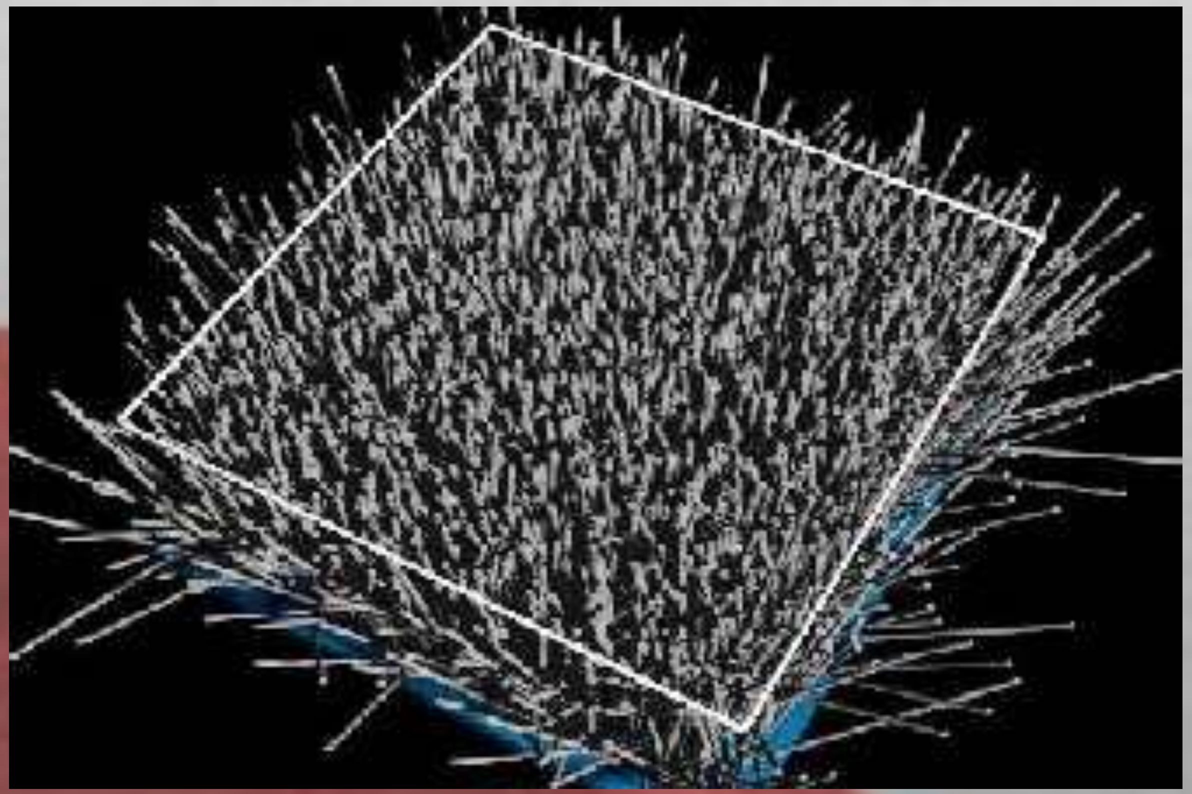
Image credit: PBRT

More complex materials: glint



Velvet





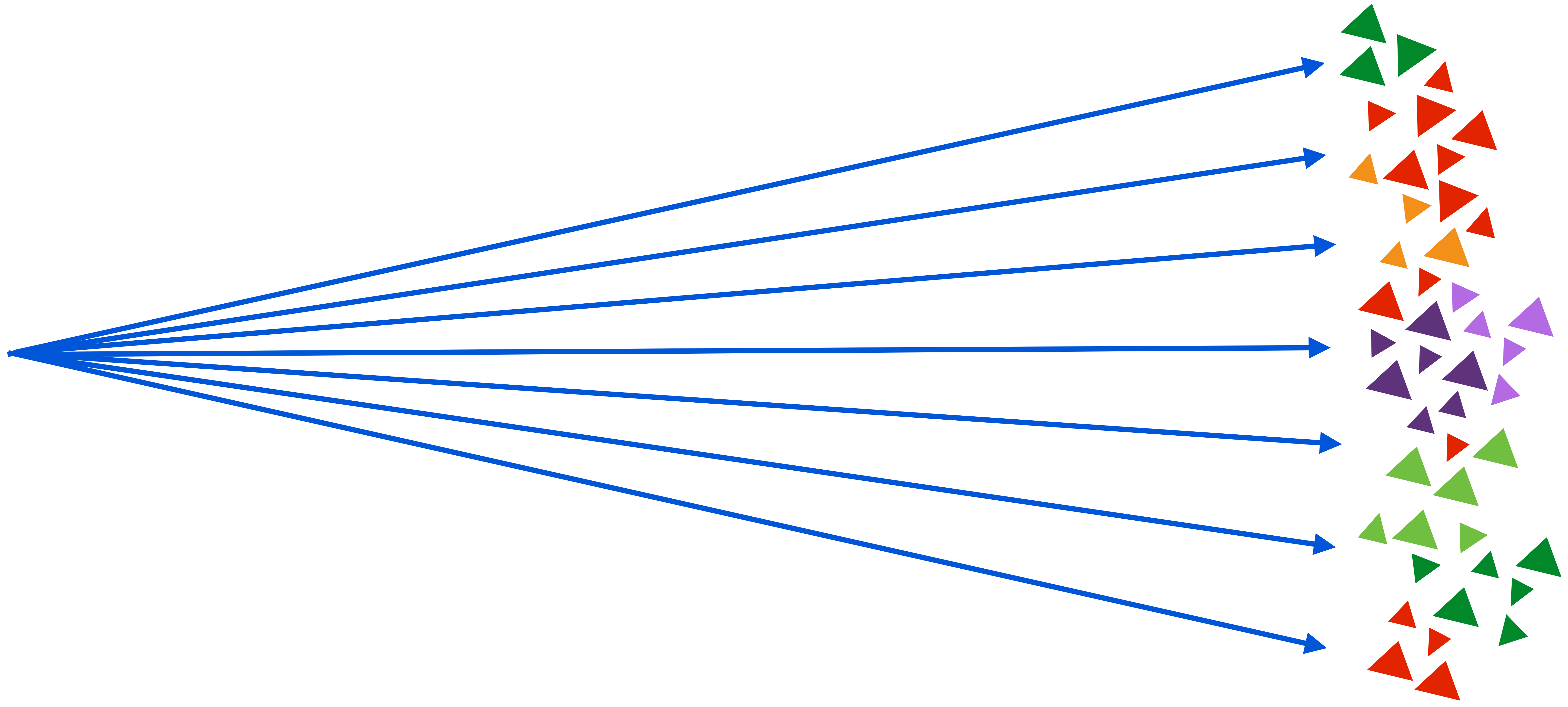
[Westin et al. 1992]

Subsurface scattering

When rays hit different surfaces…

Surface shading incoherence:

Different code paths needed to compute the reflectance of different materials

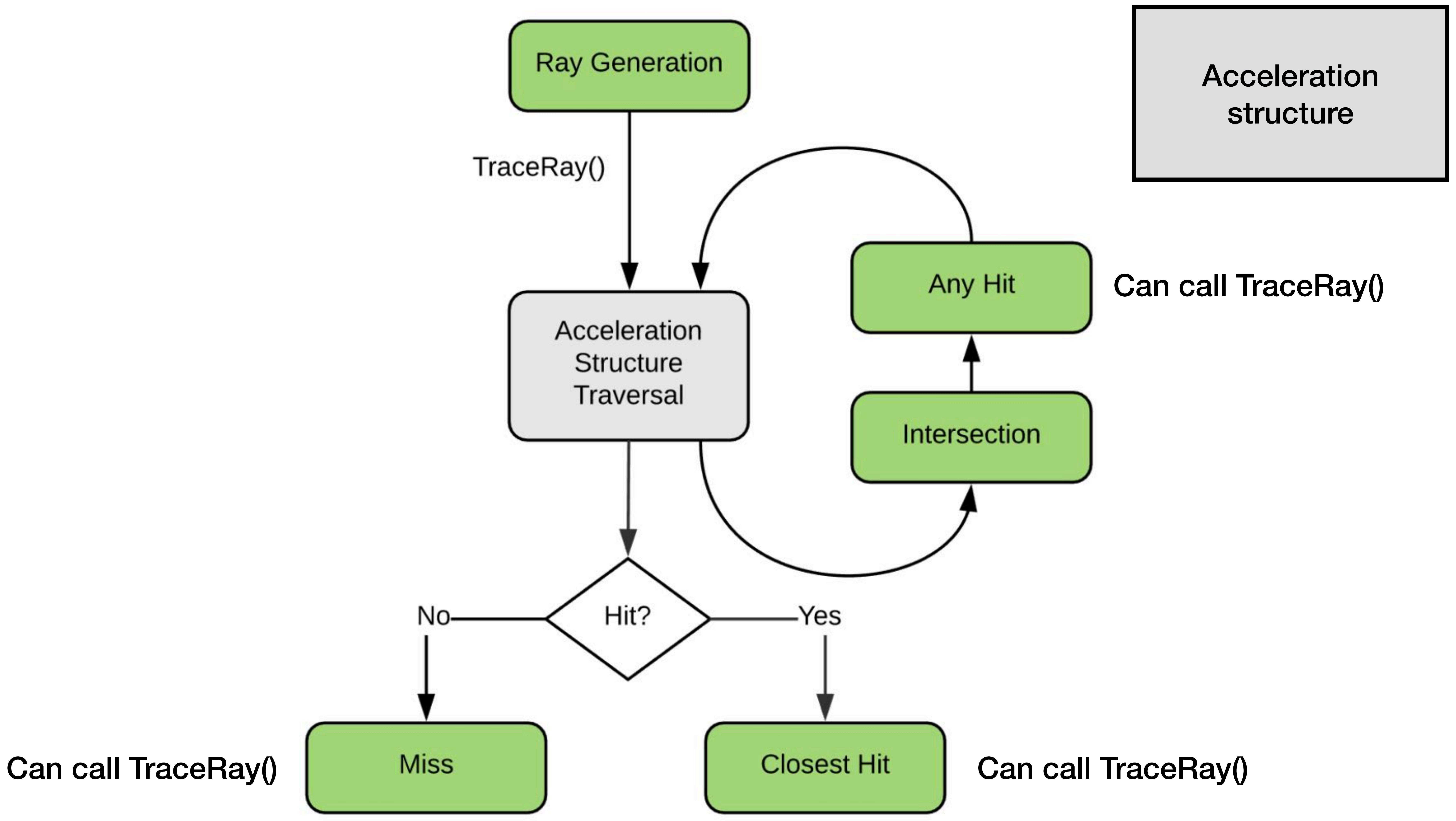


Real-time ray tracing APIs

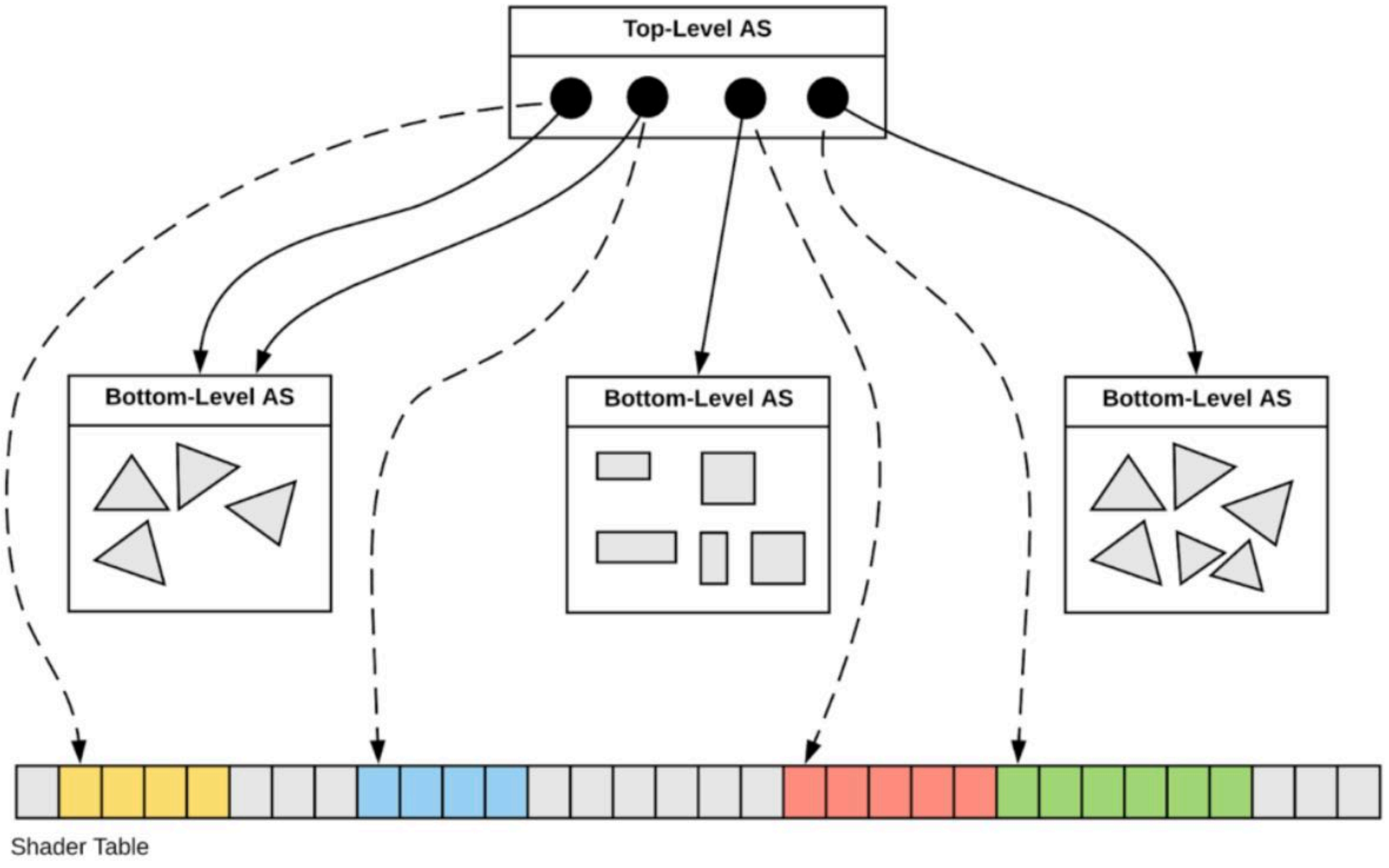
(Recurring theme in this course: increase level of abstraction to enable optimized implementations)

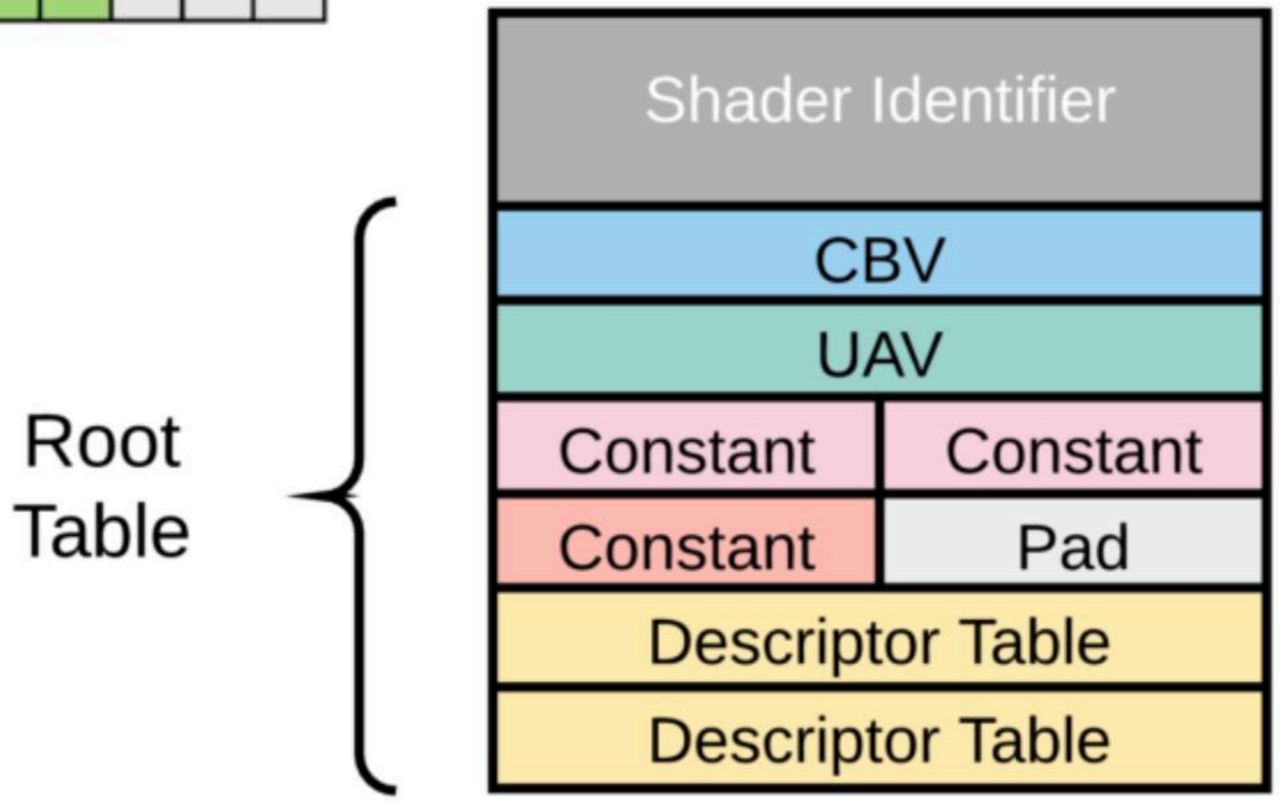
D3D12's DXR ray tracing "stages"

- 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

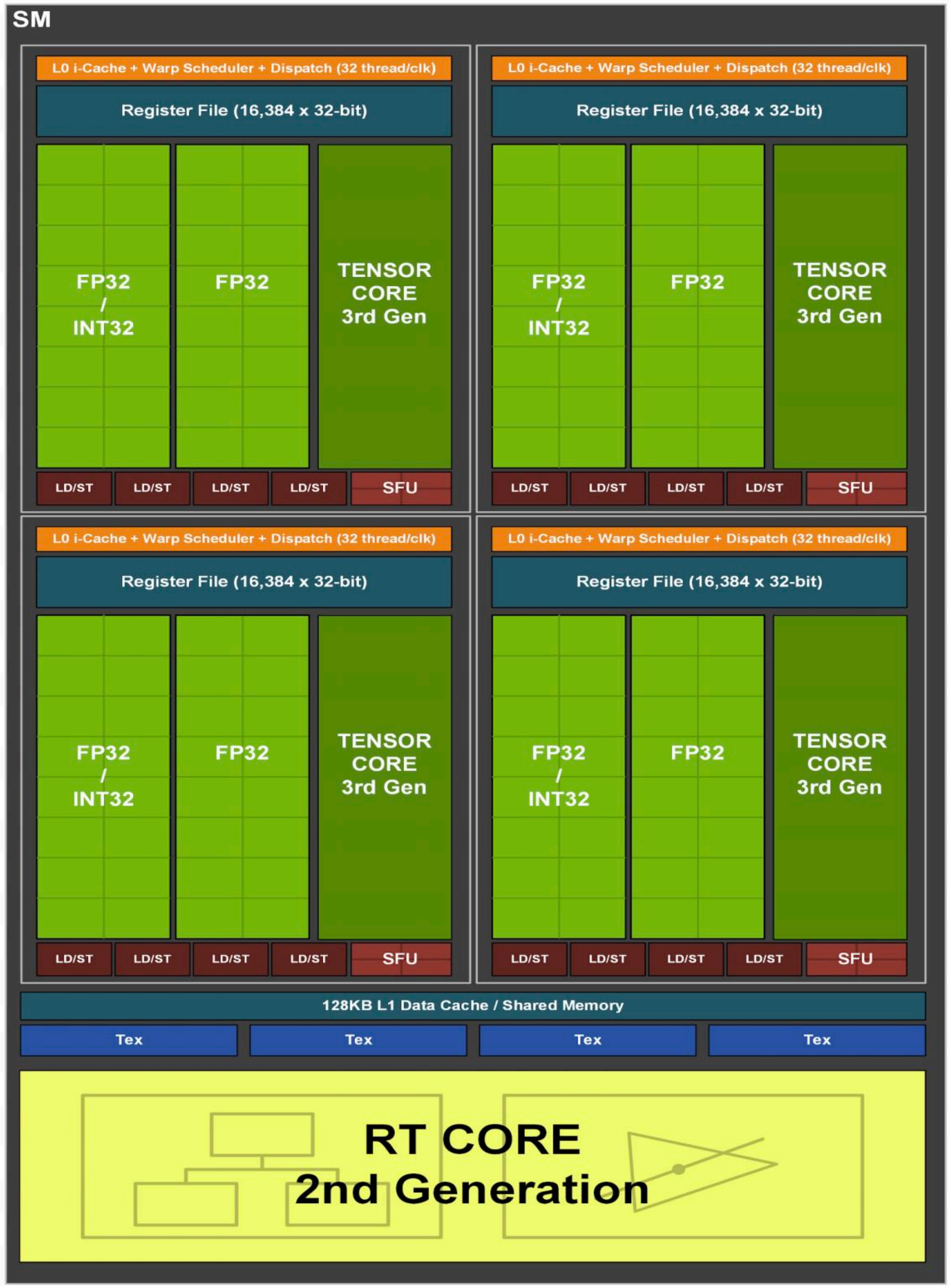
Custom hardware for RT

NVIDIA GeForce RTX 3080 GPU

NVIDIA Ampere SM (RTX 3xxx series)

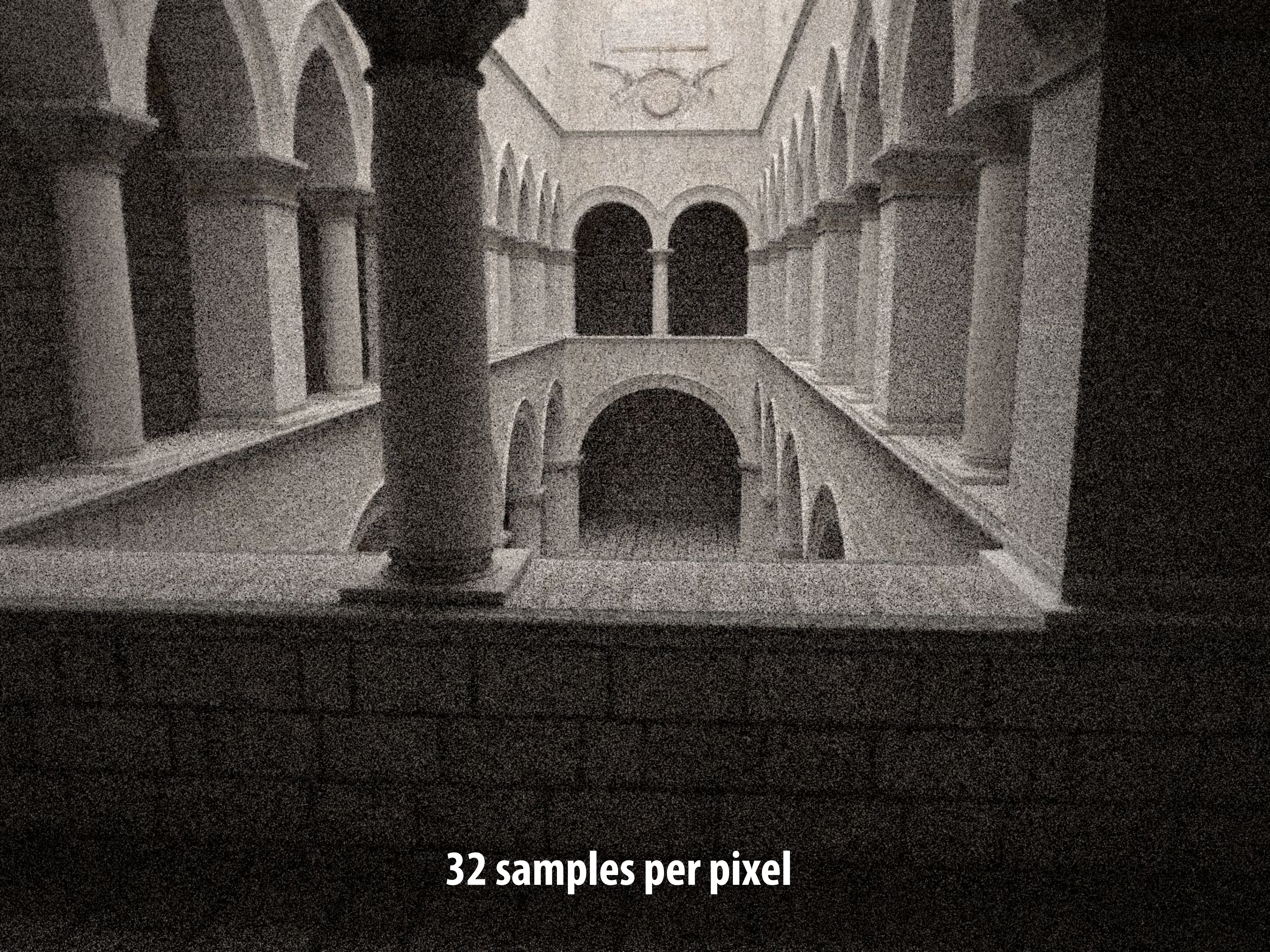
- Hardware support for ray-triangle intersection and ray-BVH intersection ("RT core")

- Very little public documentation of architectural details at this time



SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Denoising ray traced images


32 samples per pixel

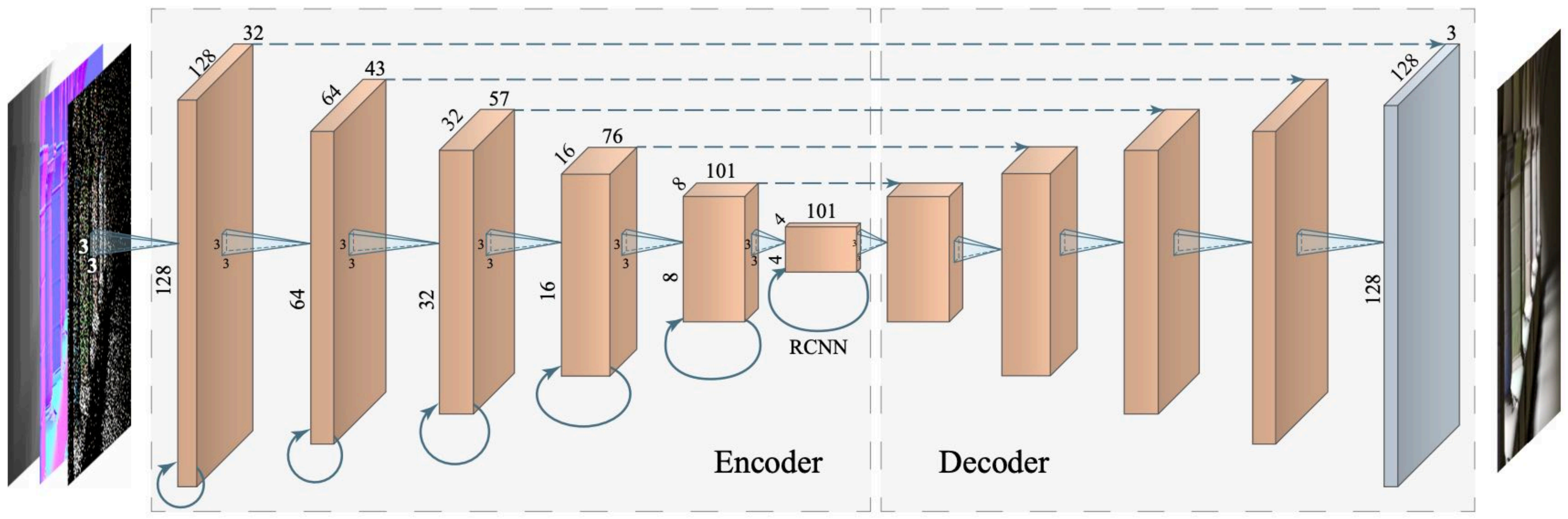
Deep learning-based denoising

- Can we “learn” to turn noisy images into clean ones?

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

Example: NVIDIA Optix denoiser

- https://developer.nvidia.com/optix-denoiser



Denoising examples

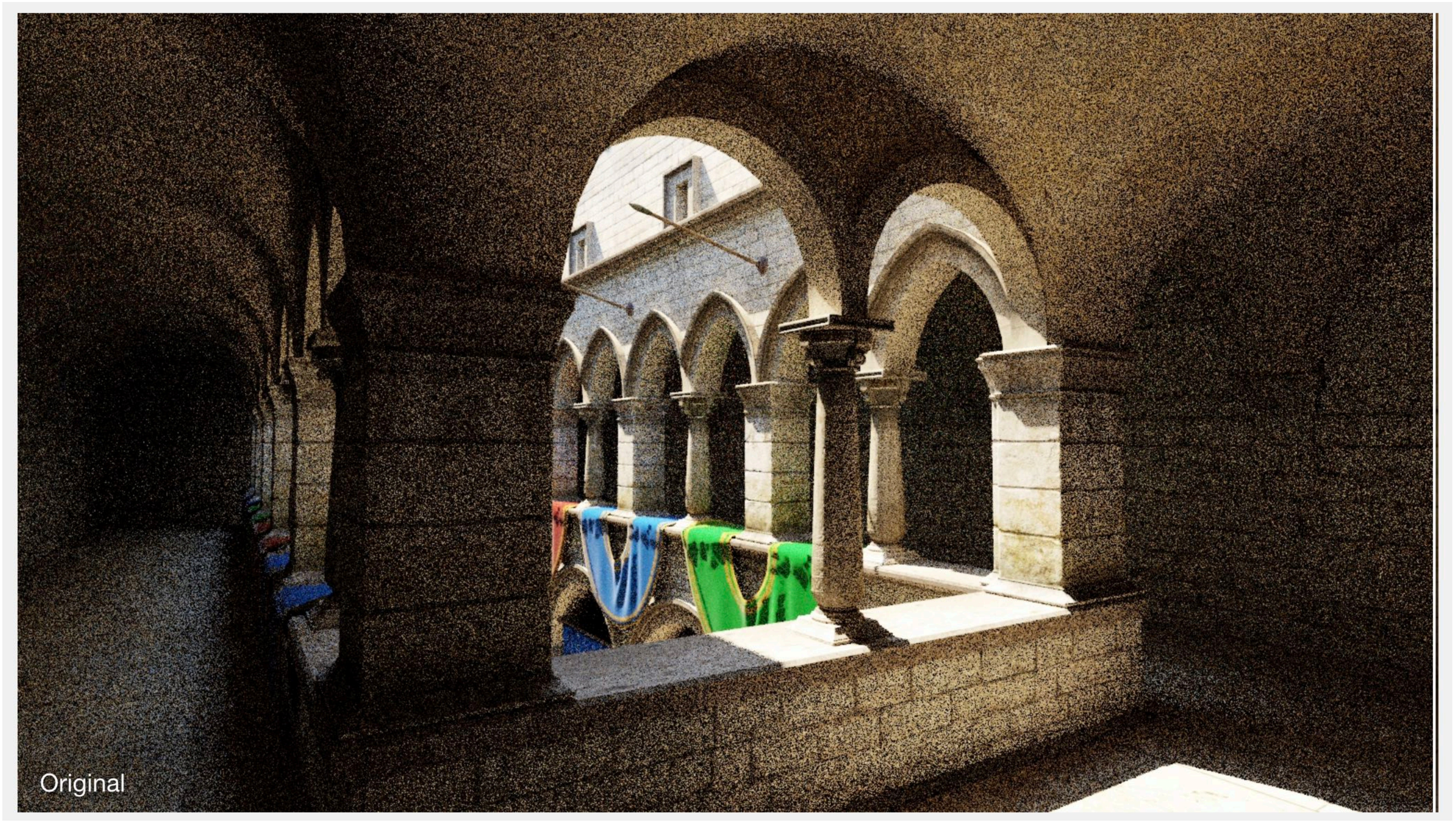


Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

Denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

Denoising examples

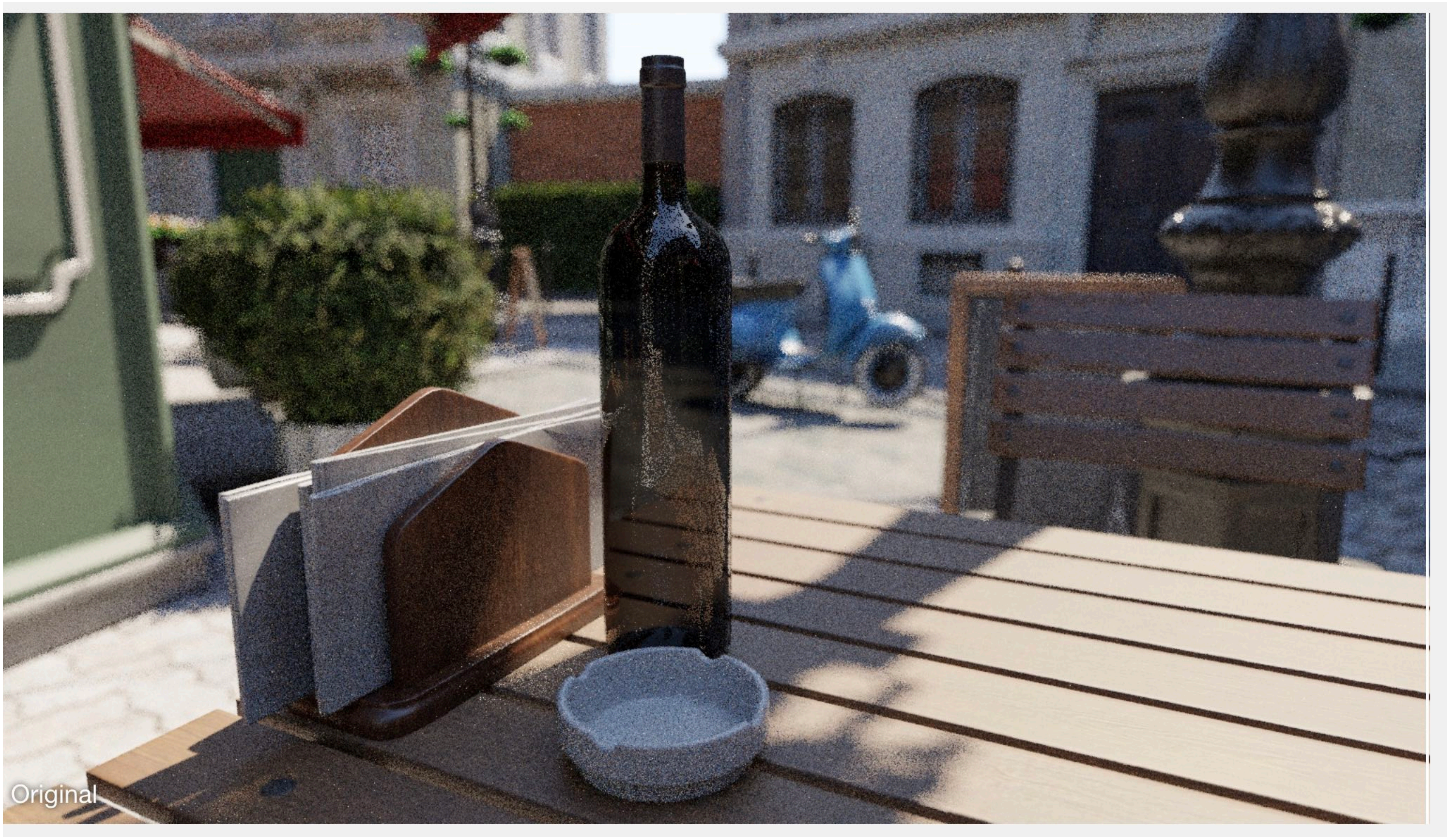


Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

Denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

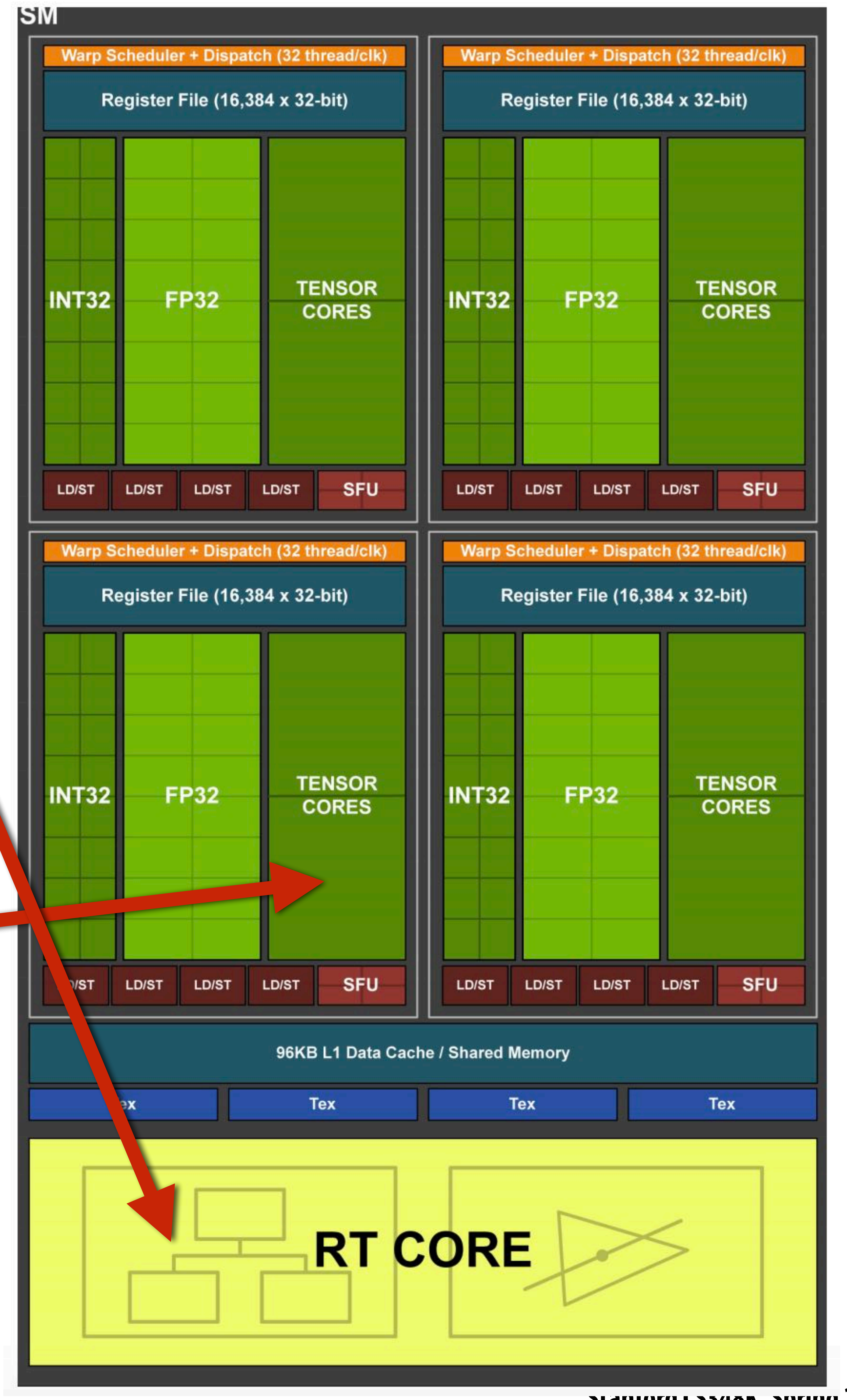
Surprising synergies

- New GPU hardware for ray-tracing operations

- But ray tracing still too expensive for noise-free images in real-time

- Tensor core: specialized hardware for accelerated DNN computations

  (that can be used to perform sophisticated denoising of ray traced images)


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Technologies that are making real-time ray tracing possible

- Better algorithms: fast parallel BVH construction and traversal algorithms (many SIGGRAPH/HPG papers circa 2010-2017)
- GPU hardware evaluation:
  - HW acceleration of ray-triangle intersection, BVH traversal
  - Increasingly flexible aspects of traditional GPU pipeline (bindless textures/resources)
- DNN-based image denoising
  - Can make plausible images using small number of rays per pixel
  - Make use of DNN hardware acceleration

Not discussed today

- Parallelizing BVH construction
- Shading coherence issues/optimizations