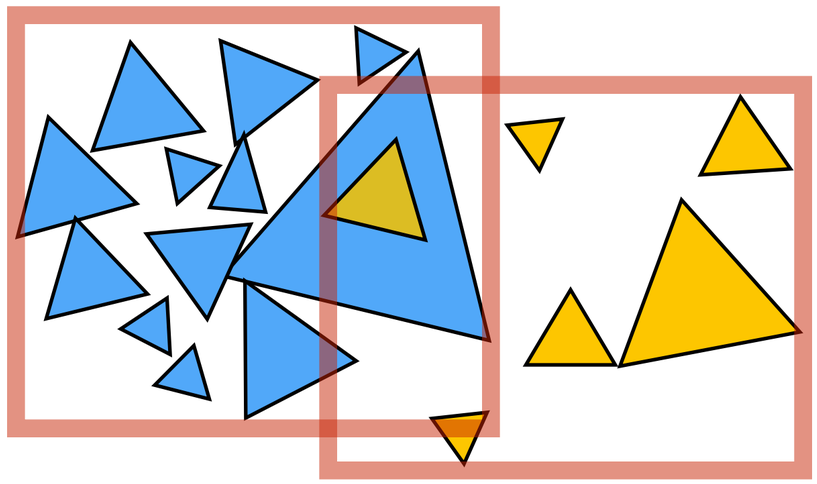
Ray Tracing

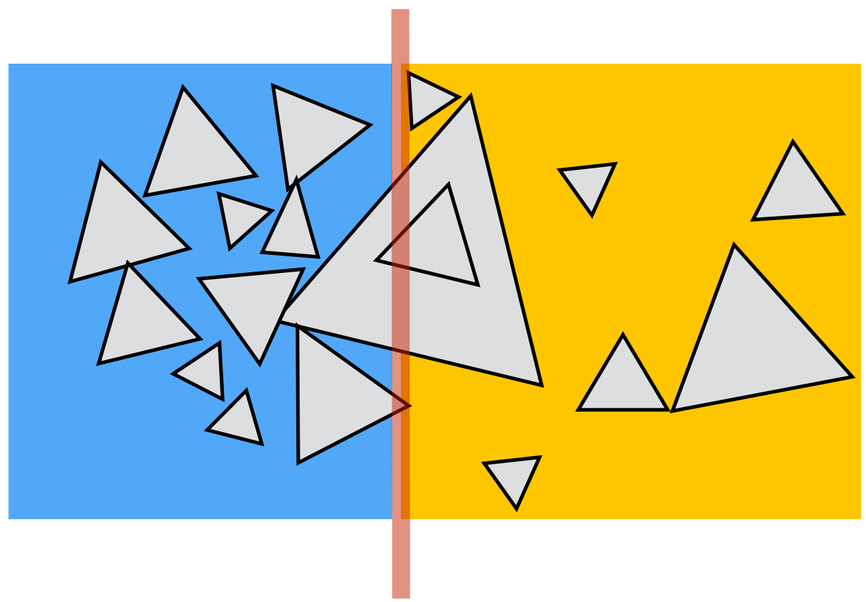
Computer Graphics
CMU 15-462/15-662, Fall 2016

Primitive-partitioning vs. space-partitioning acceleration structures

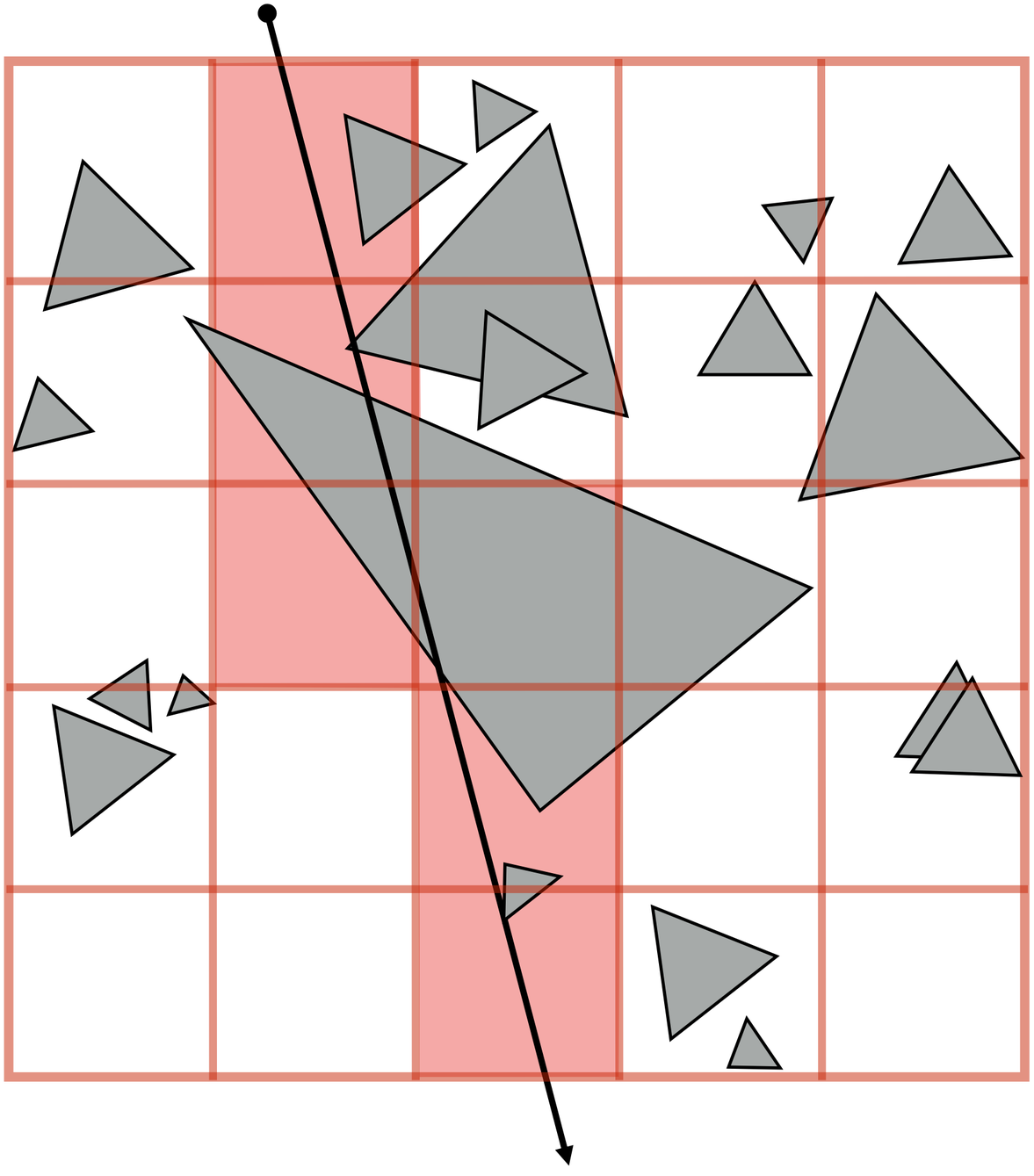
- **Primitive partitioning (bounding volume hierarchy): partitions node's primitives into disjoint sets (but sets may overlap in space)**



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

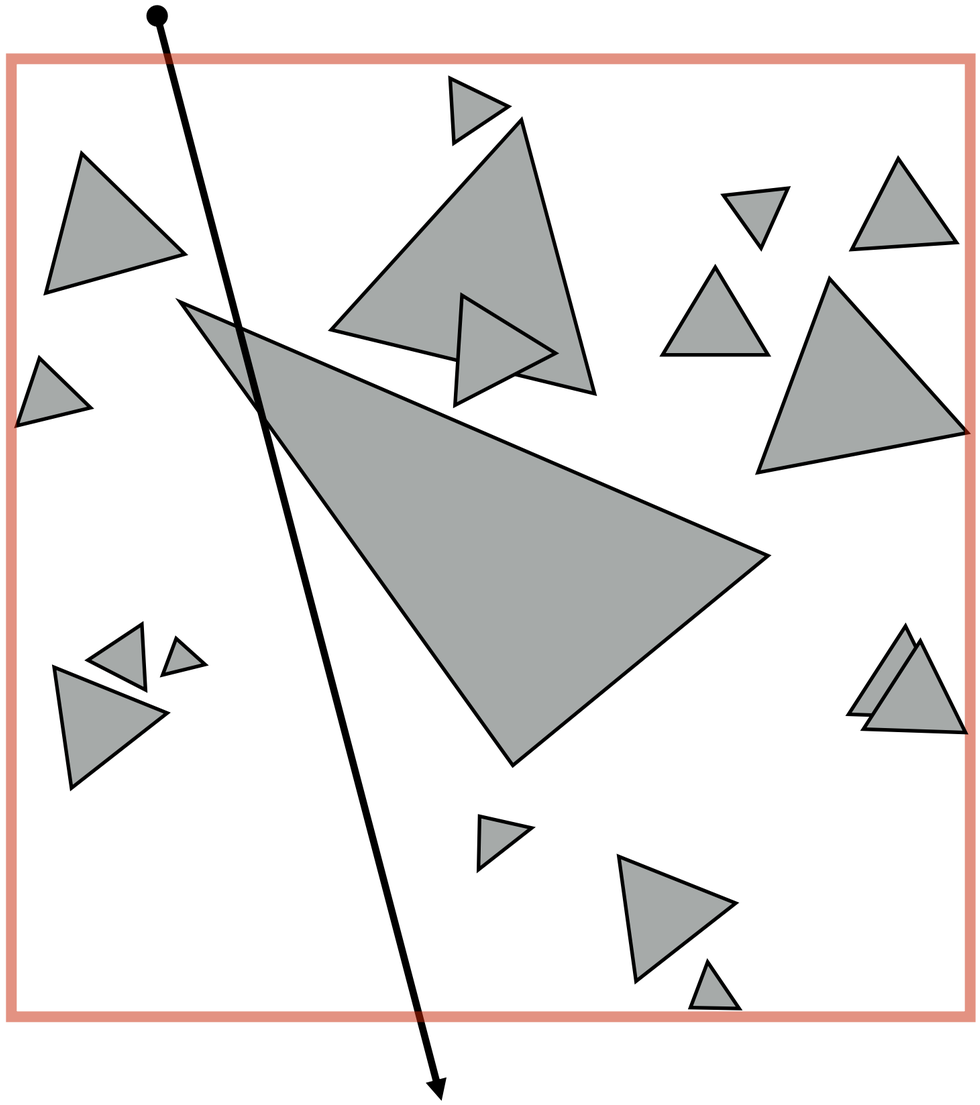


Uniform grid

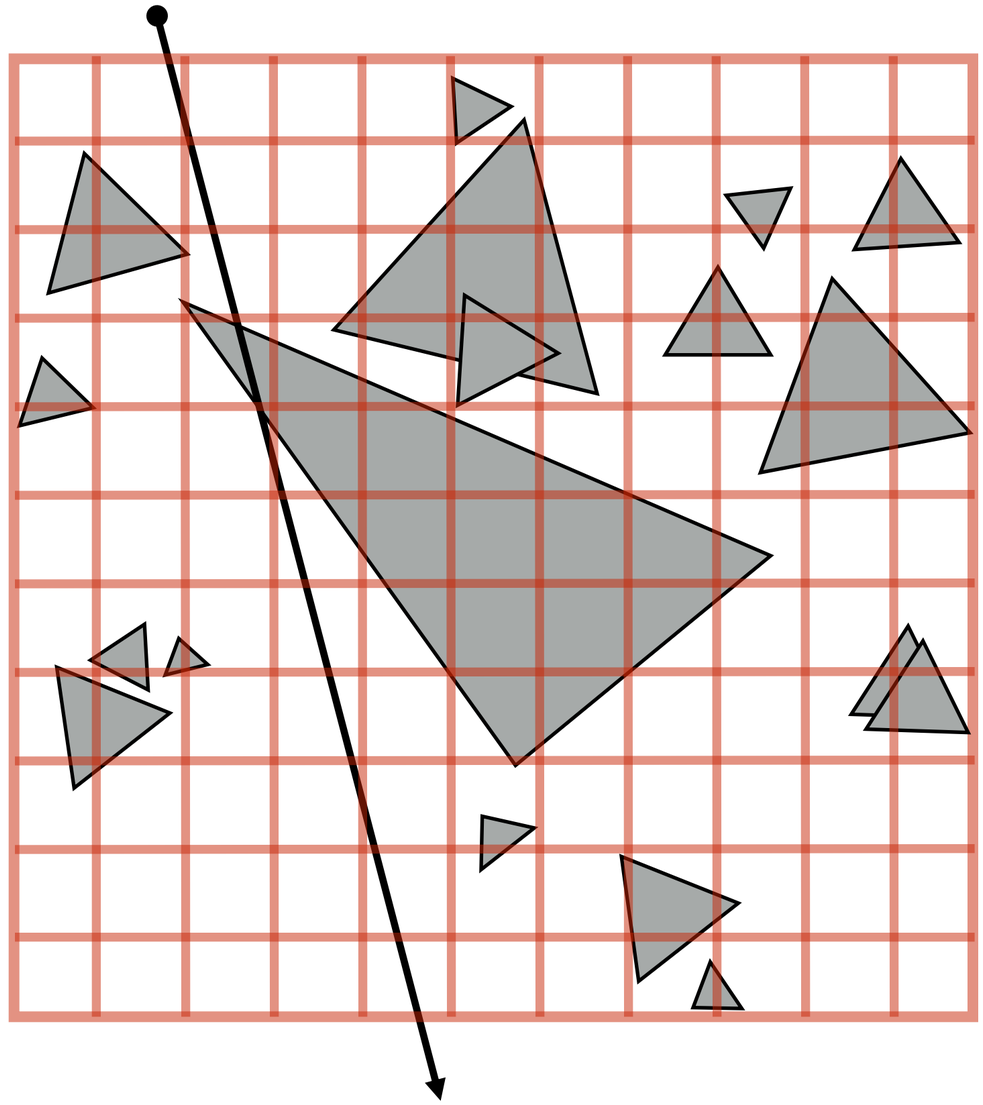


- Partition space into equal sized volumes (“voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
  - Very efficient implementation possible (think: 3D line rasterization)
  - Only consider intersection with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

What should the grid resolution be?



Too few grids cell: degenerates to brute-force approach



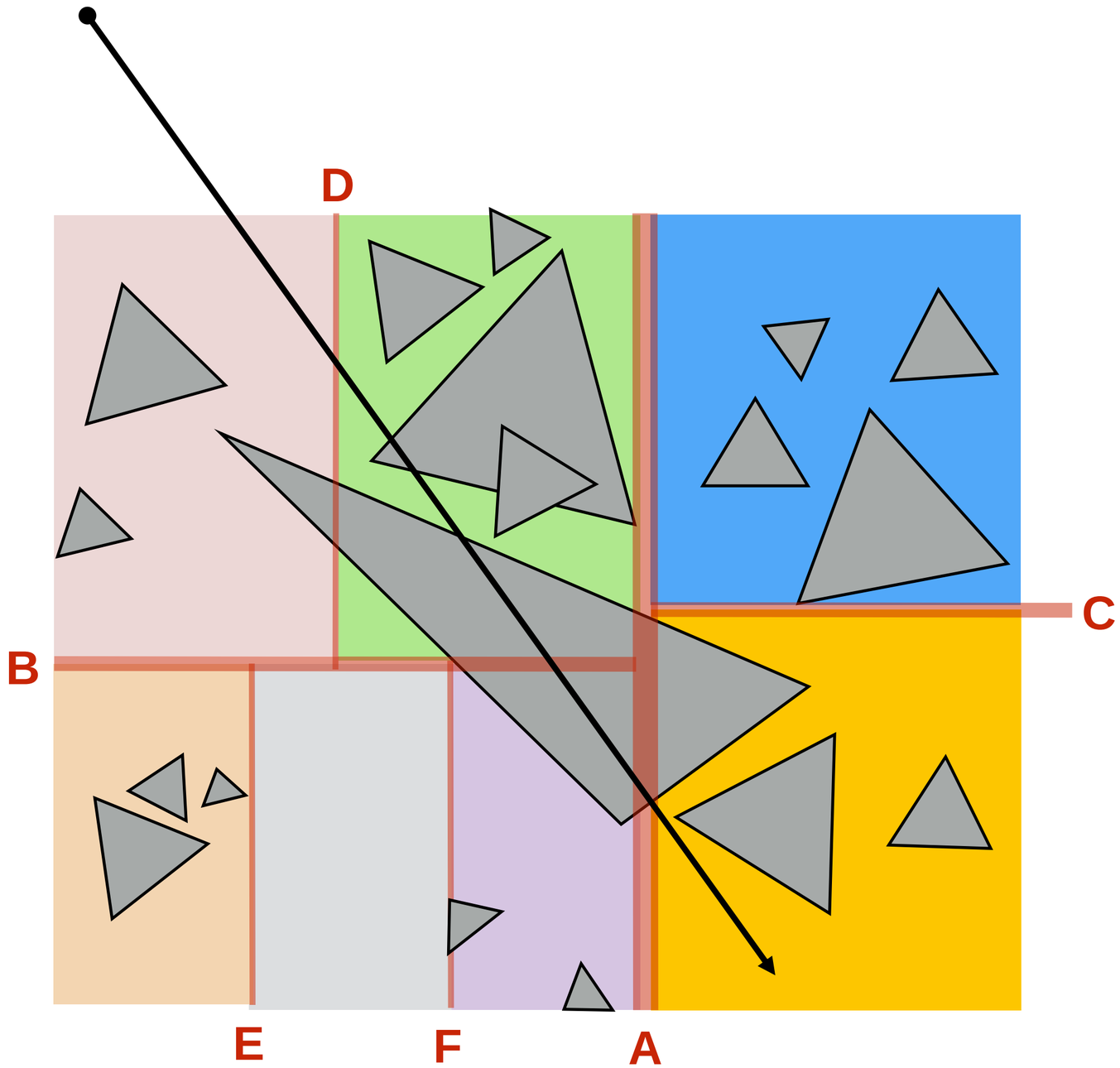
Too many grid cells: incur significant cost traversing through cells with empty space

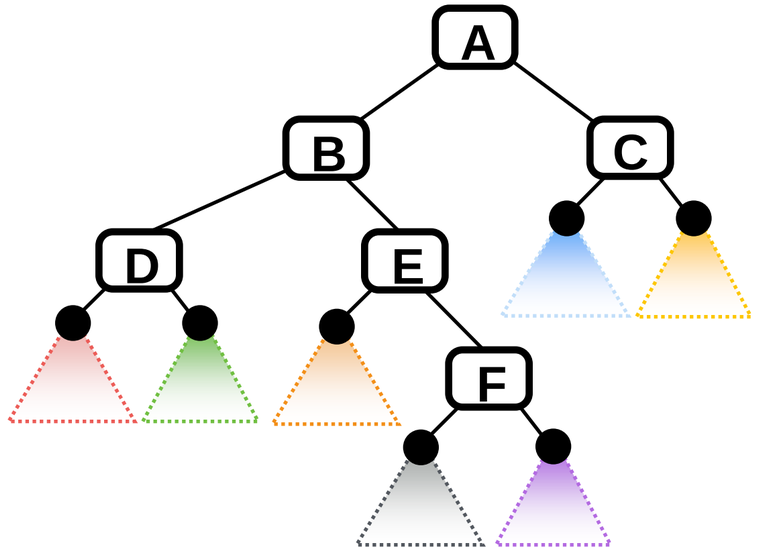
Non-uniform distribution of geometric detail requires adaptive grids

[Image credit: Pixar]

K-D trees

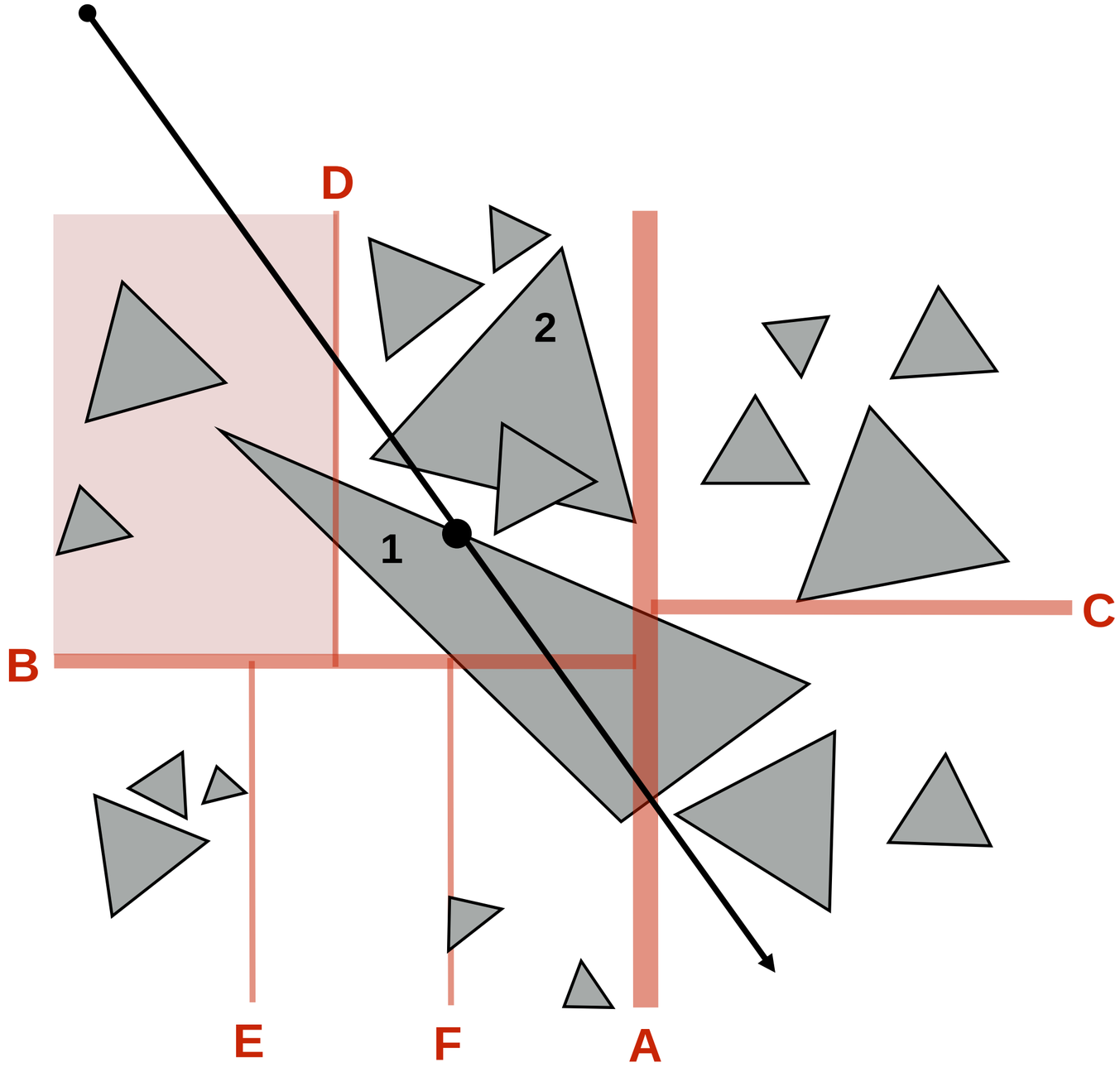
- **Recursively partition <u>space</u> via axis-aligned planes**
  - **Interior nodes correspond to spatial splits (still correspond to spatial volume)**
  - **Node traversal can proceed in front-to-back order (unlike BVH, can terminate search after first hit is found*).**

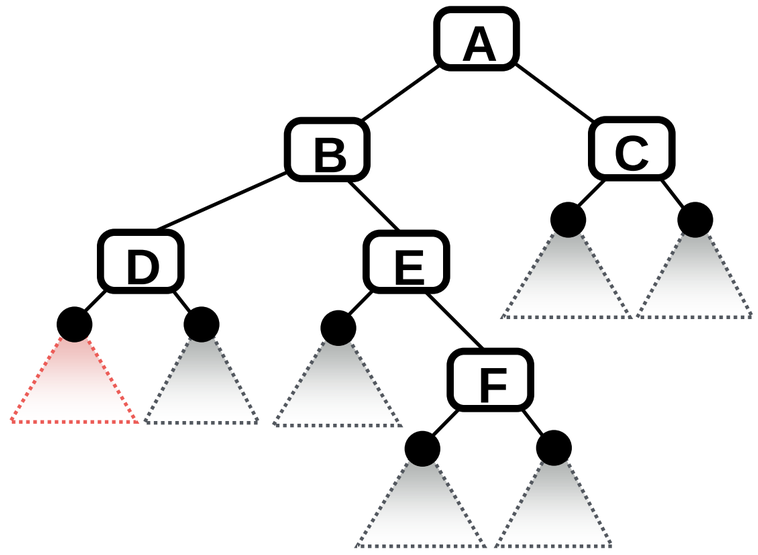




Challenge: objects overlap multiple nodes

- Want node traversal to proceed in front-to-back order so traversal can terminate search after first hit found





Triangle 1 overlaps multiple nodes.

Ray hits triangle 1 when in highlighted leaf cell.

But intersection with triangle 2 is closer! (Haven't traversed to that node yet)

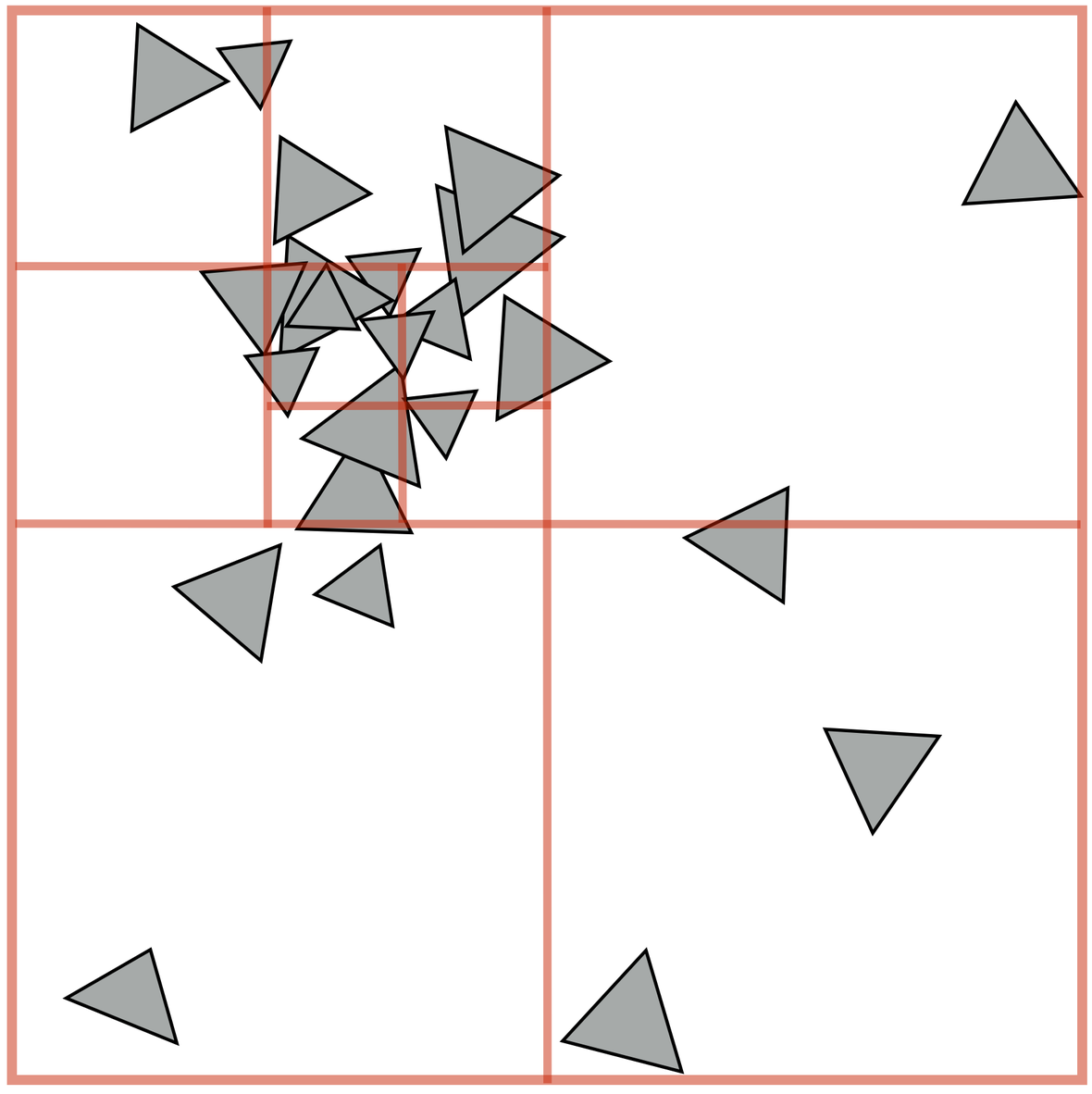
Solution: require primitive intersection point to be within current leaf node. (primitives may be intersected multiple times by same ray)

Quad-tree / octree

Like uniform grid: easy to build (don’t have to choose partition planes)

Has greater ability to adapt to location of scene geometry than uniform grid.

But lower intersection performance than K-D tree (only limited ability to adapt)



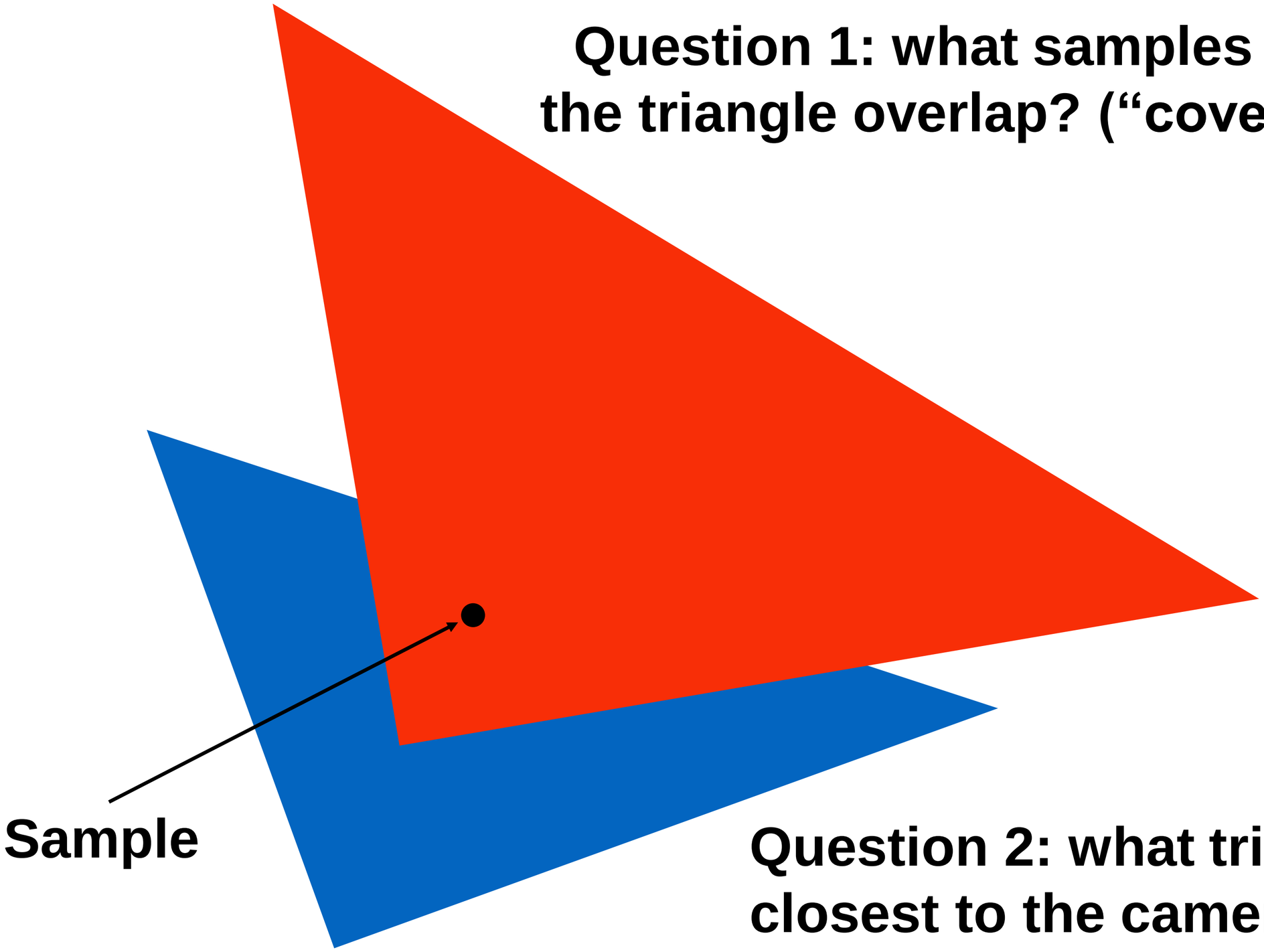
Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)

Summary of acceleration data structures: choose the right method for the job

- Primitive vs. spatial partitioning:
  - Primitive partitioning: partition sets of objects
    - Bounded number of BVH nodes, simpler to update if primitives in scene change position
  - Spatial partitioning: partition space
    - Traverse space in order (first intersection is closest intersection), may intersect primitive multiple times
- Adaptive structures (BVH, K-D tree)
  - More costly to construct (must be able to amortize construction over many geometric queries)
  - Better intersection performance under non-uniform distribution of primitives
- Non-adaptive accelerations structures (uniform grids)
  - Simple, cheap to construct
  - Good intersection performance if scene primitives are uniformly distributed
- Many, many combinations thereof

Back to drawing things: recall the visibility problem



Question 1: what samples does the triangle overlap? (“coverage”)

Question 2: what triangle is closest to the camera in each sample? (“occlusion”)

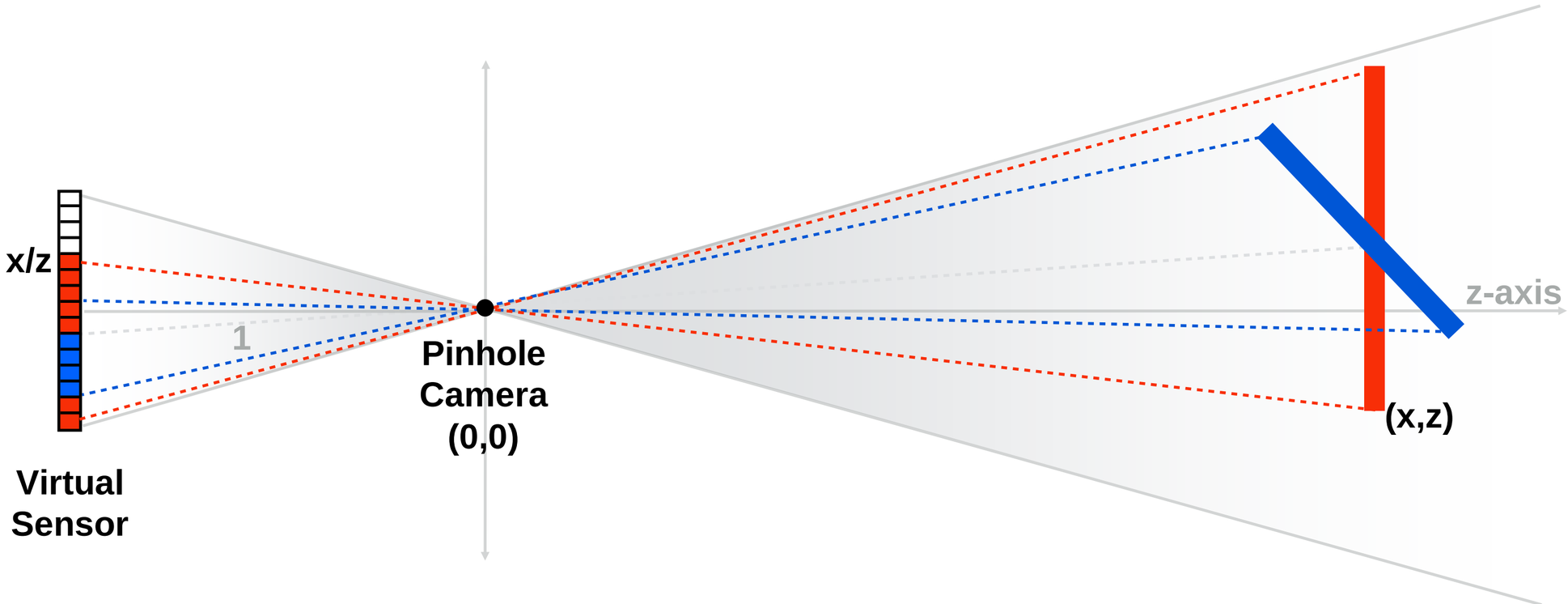
Creating realistic images

You know almost everything
you need to create this image.

But it looks so “flat” ☹

Why is it more difficult to create
the image below?

Rasterization



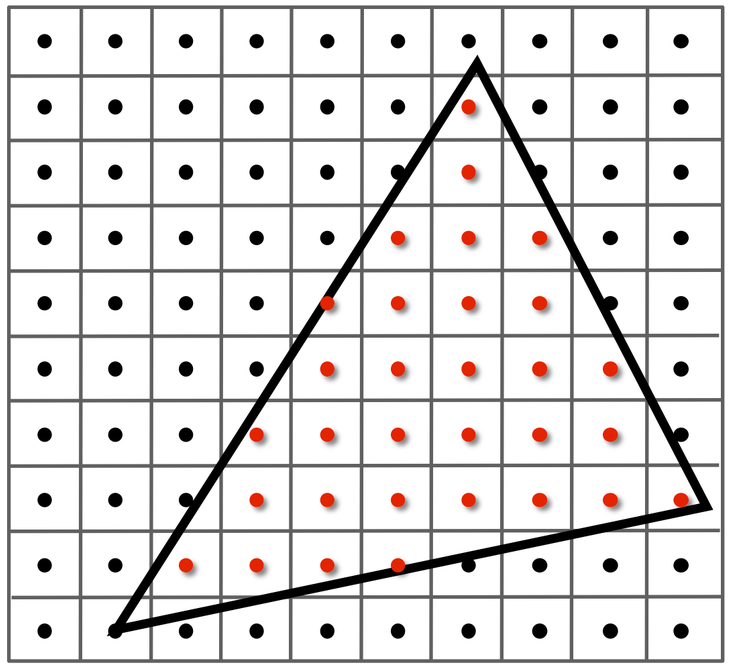
Q: How are occlusions handled?

Basic rasterization algorithm

Sample = 2D point

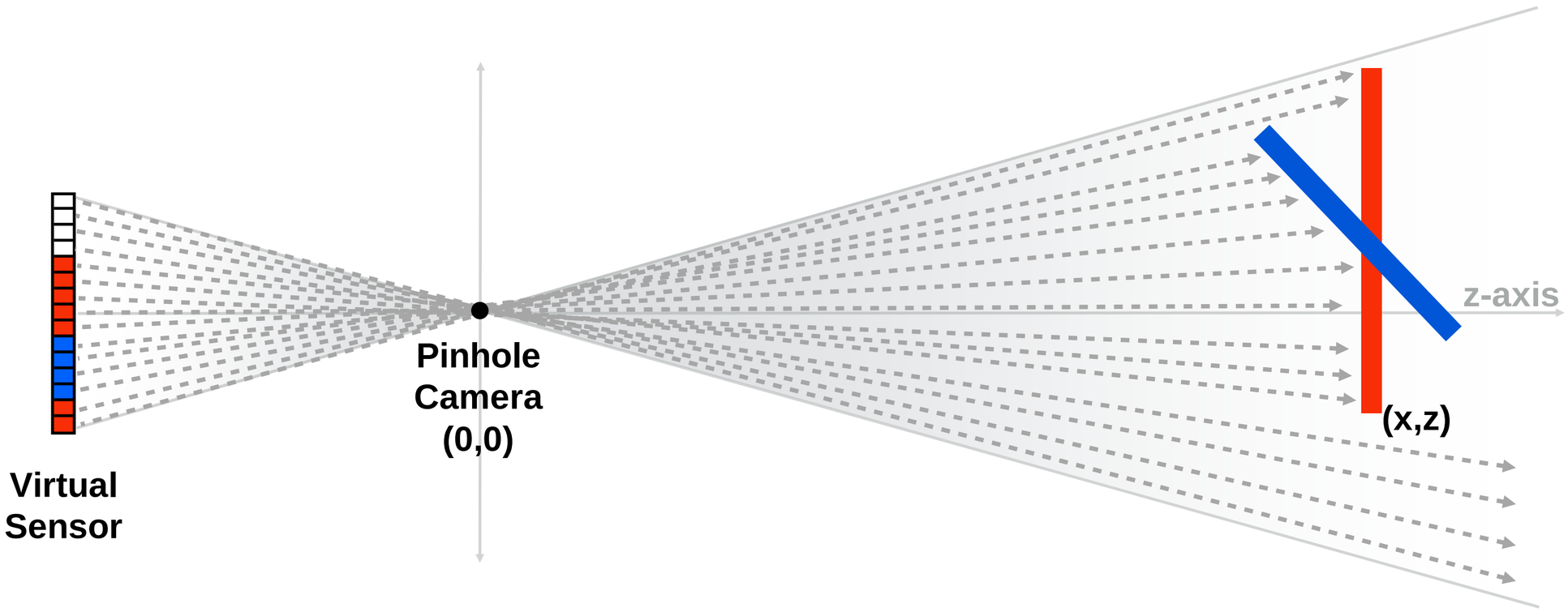
Coverage: does a projected triangle cover 2D sample point?

Occlusion: depth buffer



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

Rendering via ray-casting



A very common use of ray-scene intersection tests!

Q: How are occlusions handled?

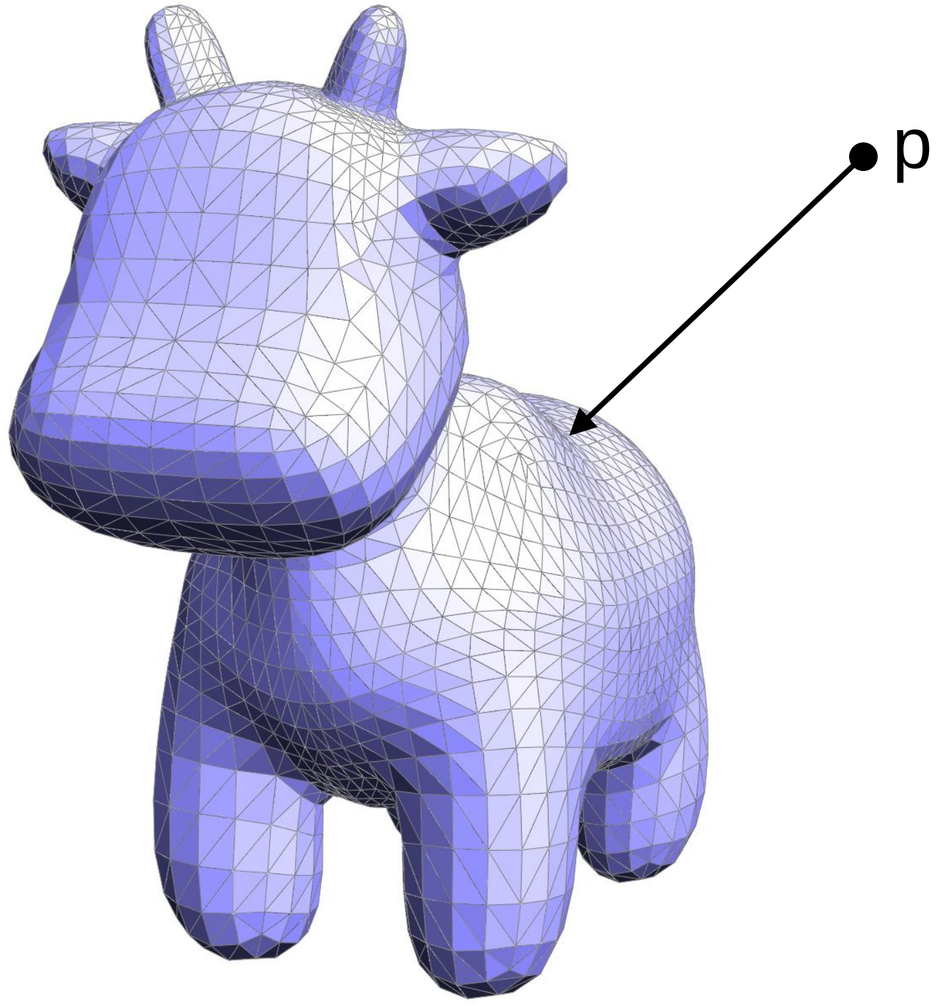
Basic ray casting algorithm

Sample = a ray in 3D

Coverage: does ray “hit” triangle (ray-triangle intersection tests)

Occlusion: closest intersection along ray

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



Rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order (never have to store in entire scene, naturally supports unbounded size scenes)
  - Store depth buffer (random access to regular structure of fixed size)

- **Ray casting:**
  - Proceeds in screen sample order
    - Never have to store depth buffer (just current ray)
    - Natural order for rendering transparent surfaces (process surfaces in the order the are encountered along the ray: front-to-back or back-to-front)
  - Must store entire scene (random access to irregular structure of variable size: depends on complexity and distribution of scene)

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

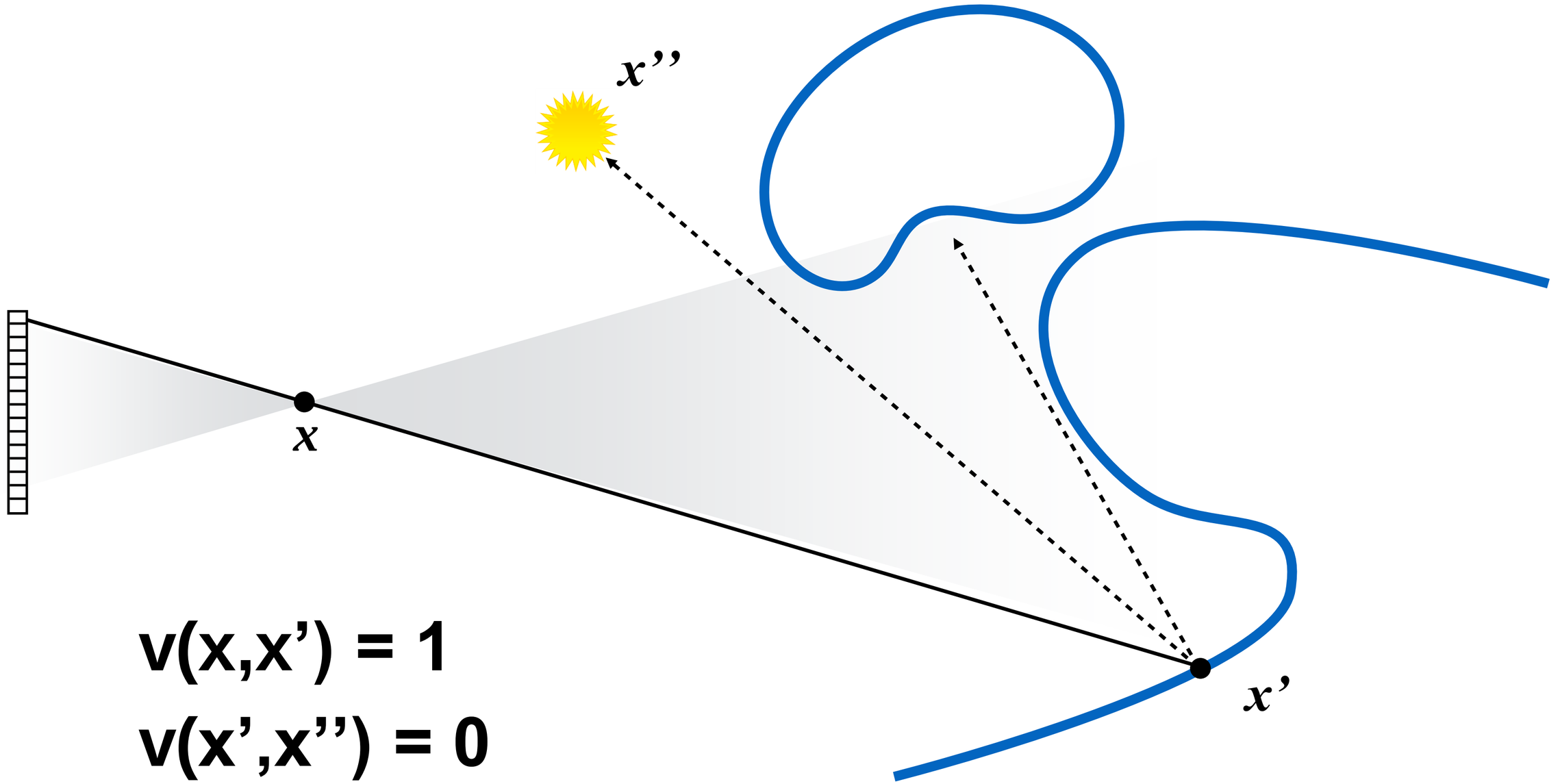
Rasterization and ray casting are two approaches for solving the same problem: determining “visibility”

Another way to think about rasterization

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

Ray tracing: a more general mechanism for answering “visibility” queries

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



$v(x,x') = 1$

$v(x',x'') = 0$

Rasterization vs Ray tracing

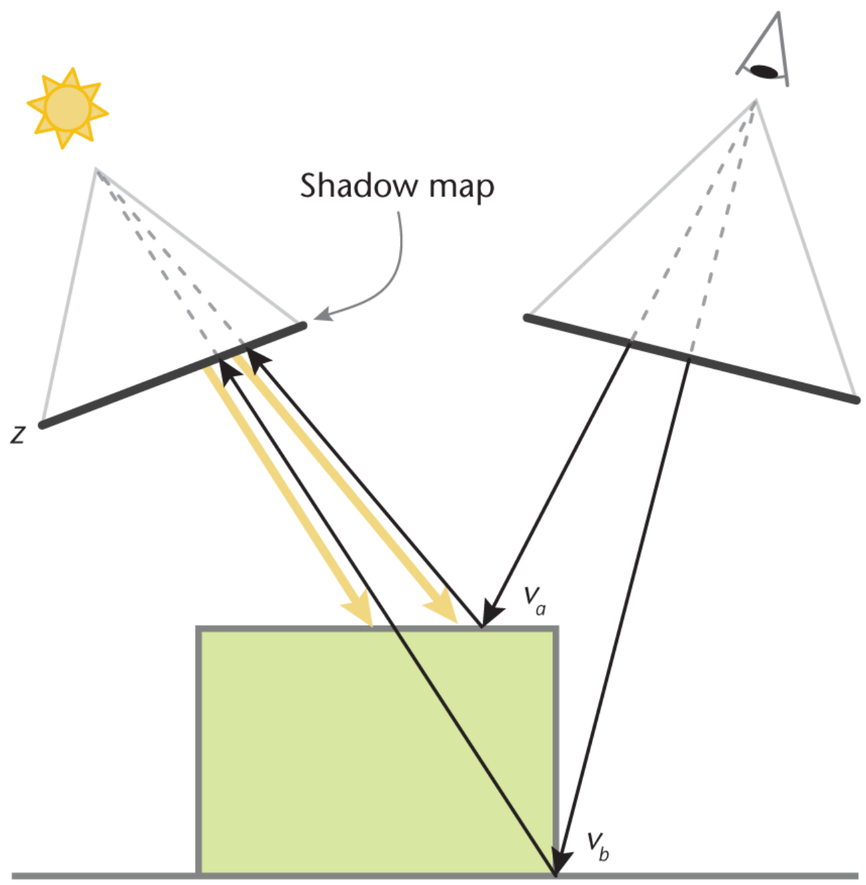
“loop over primitives”

“loop over screen pixels”

Shadows: rasterization

Shadow mapping

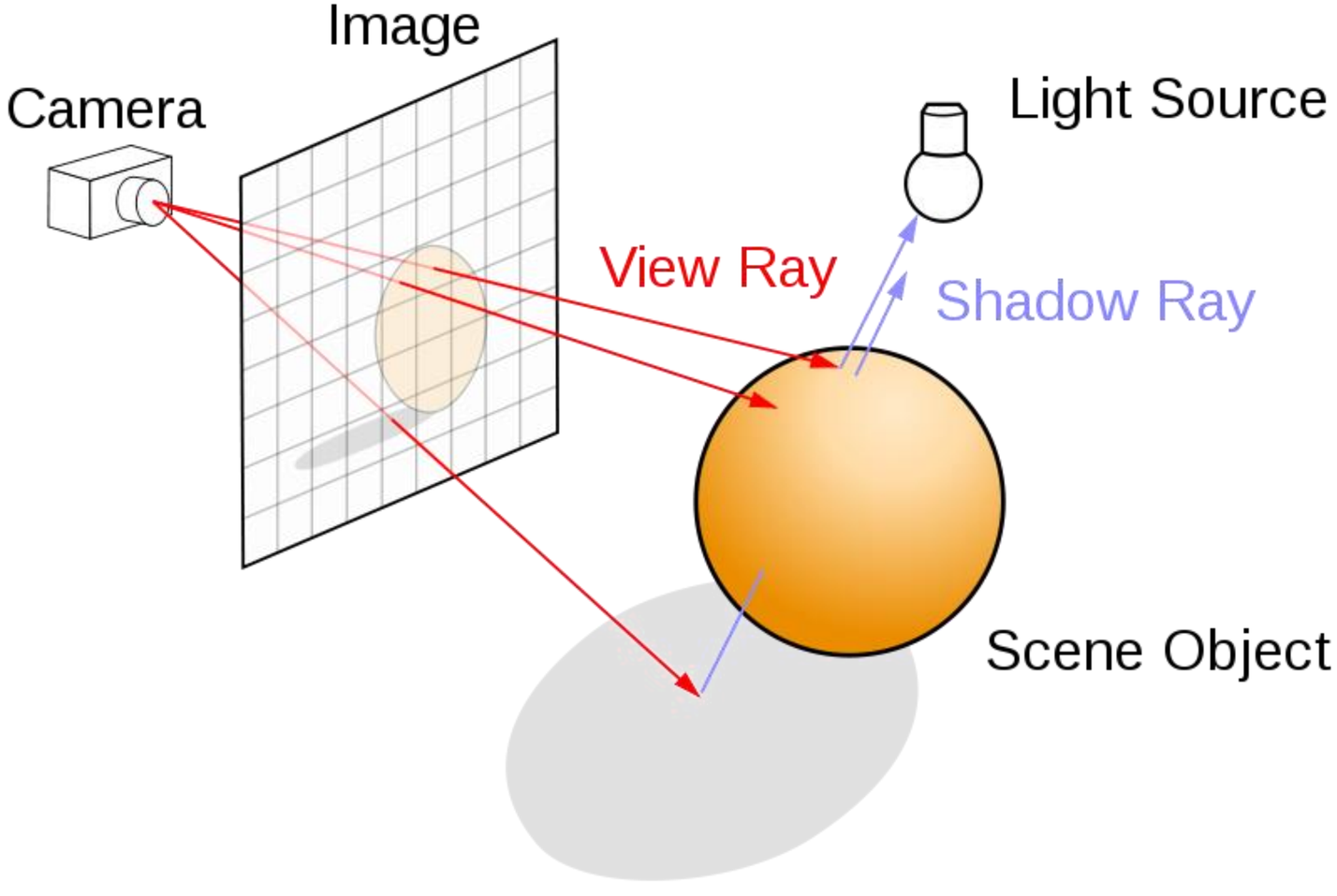
- Render scene (depth buffer only) from location of light
  - Everything “seen” from this point of view is directly lit
- Render scene from location of camera
  - Transform every screen sample to light coordinate frame and perform a depth test (fail = in shadow)



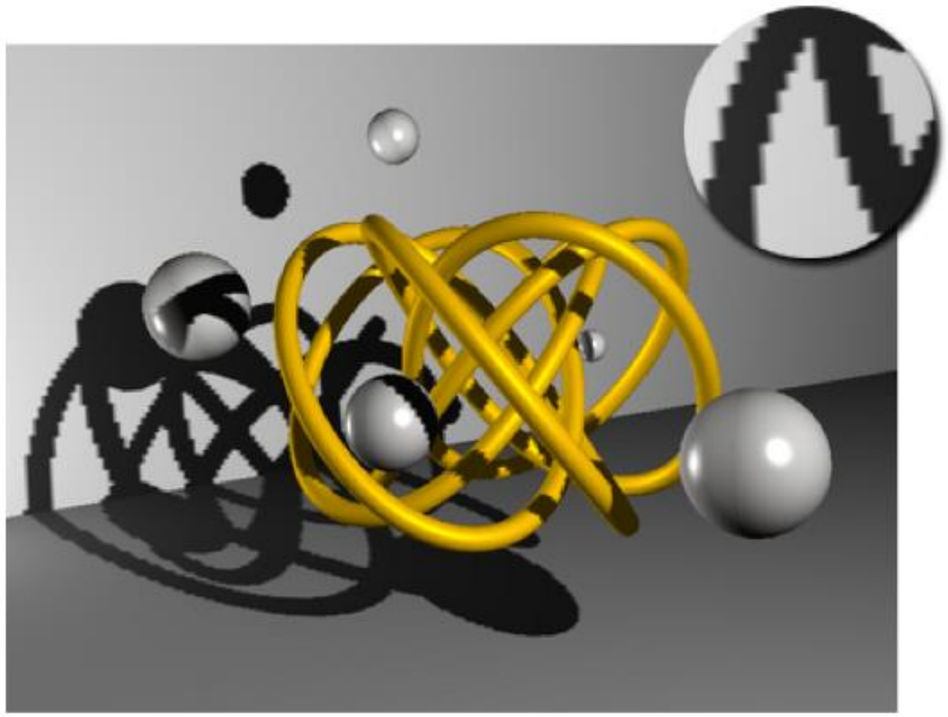
Shadows: ray tracing

Recursive ray tracing

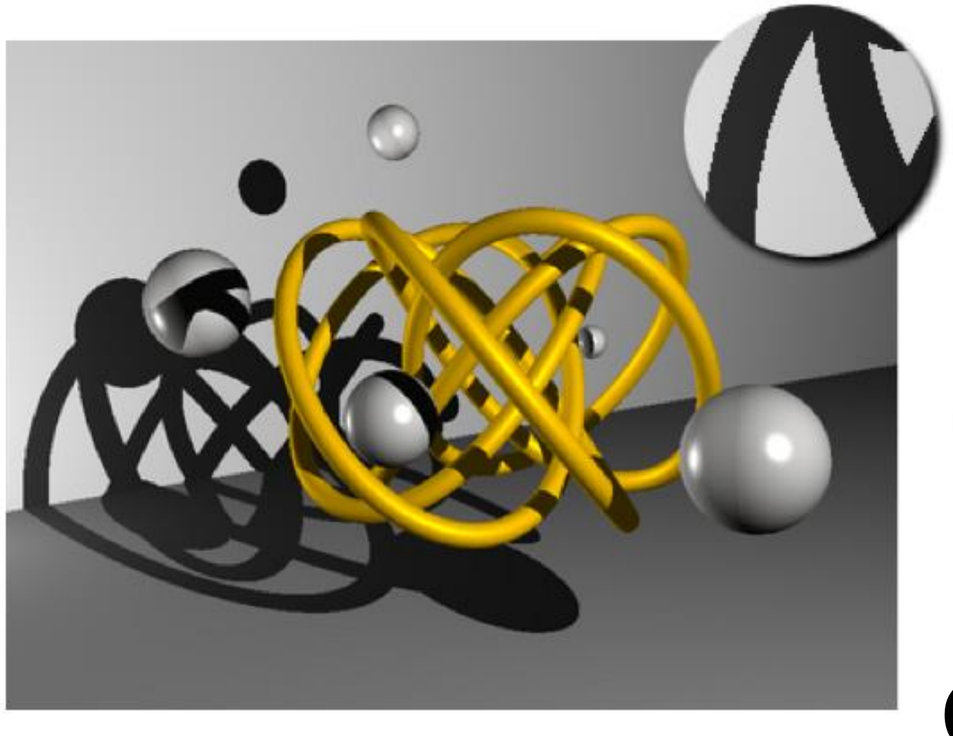
- shoot “shadow” rays towards light source from points where camera rays intersect scene
  - If unconcluded, point is directly lit by light source



Shadows: rasterization vs ray tracing



Shadows computed using shadow map (shadow map resolution is too low → aliasing)



Correct hard shadows with raytracing

Q: Hard shadows? What are soft shadows?

Image credit: Johnson et al. TOG 2005

Reflections

Reflections: rasterization

Environment mapping

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

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

Approximates appearance of reflective surface

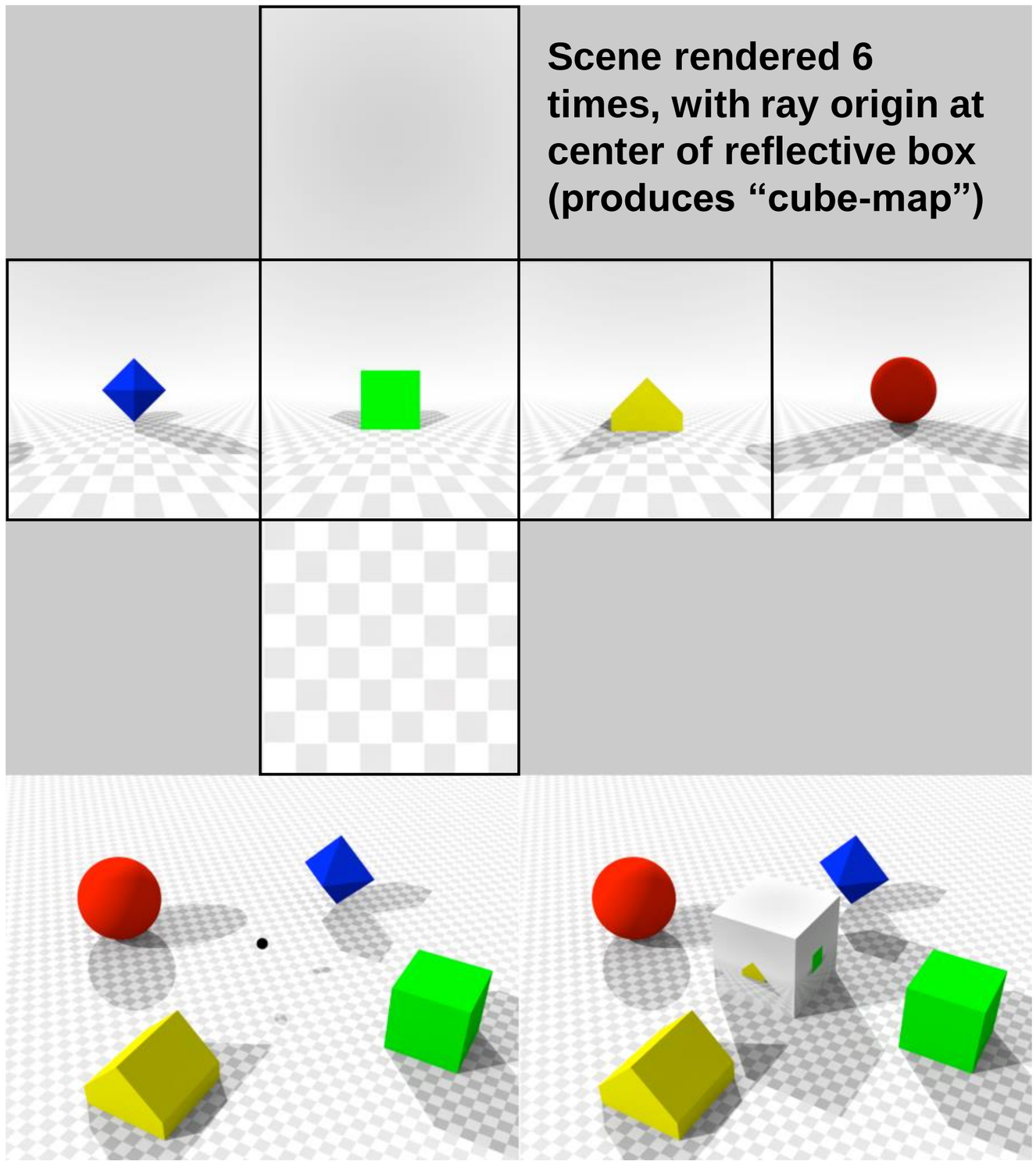
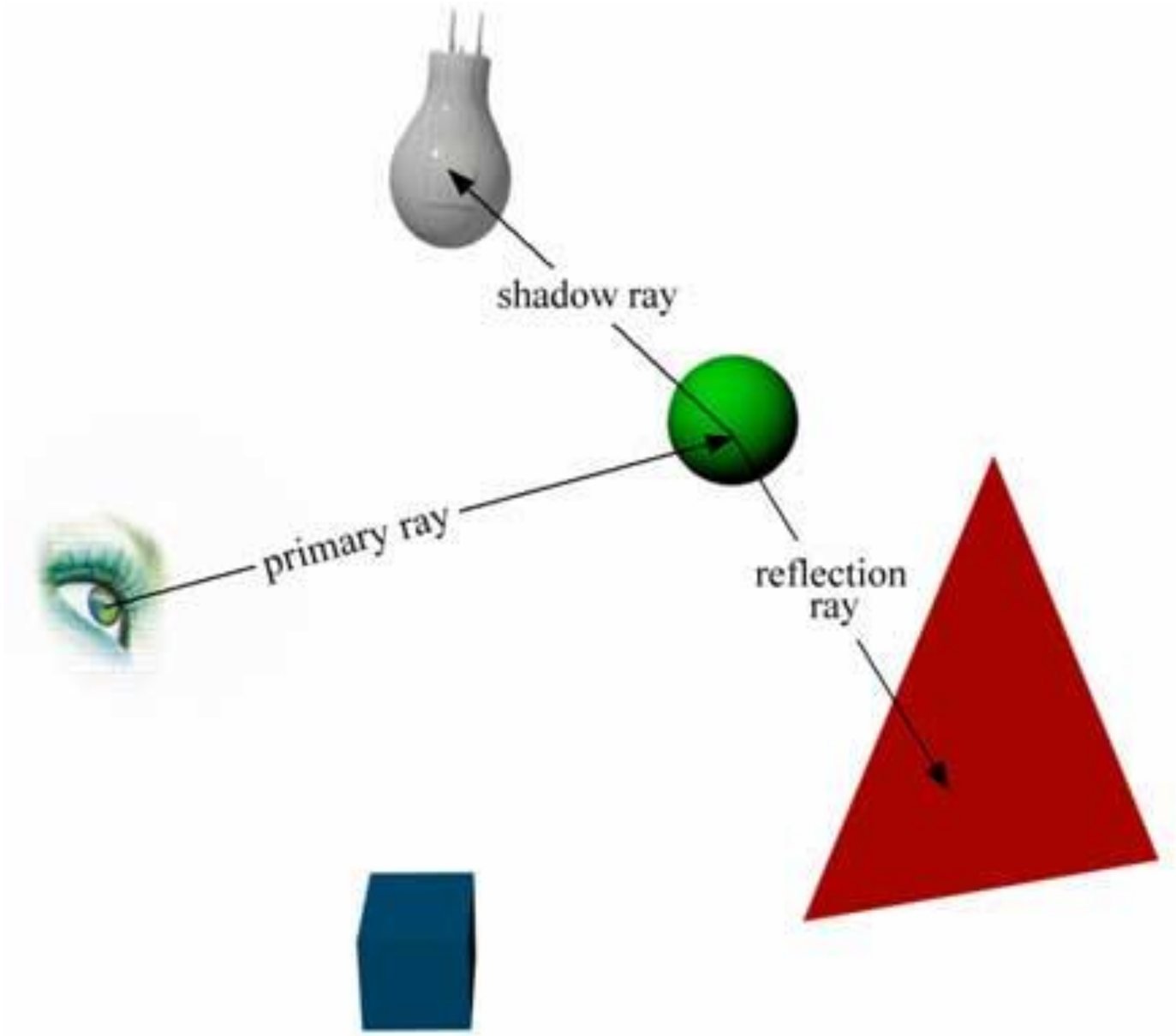


Image credit: http://en.wikipedia.org/wiki/Cube_mapping

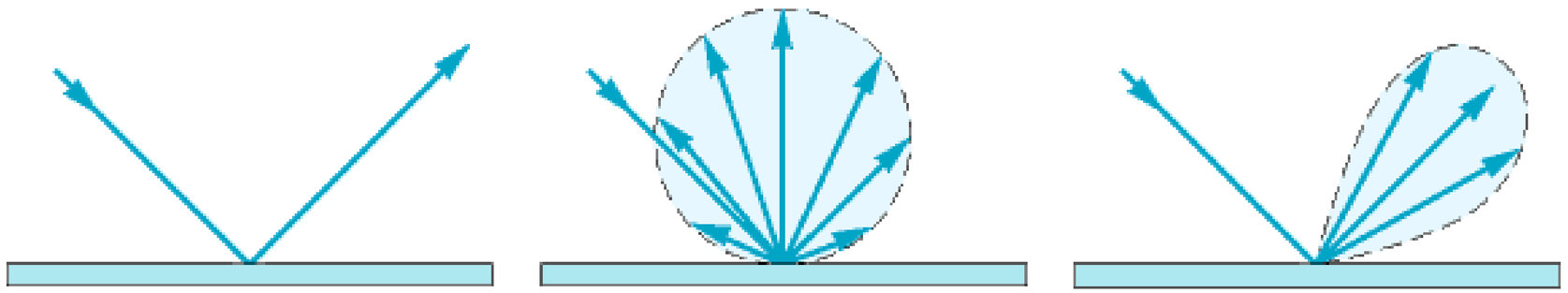
Reflections: ray tracing

Recursive ray tracing



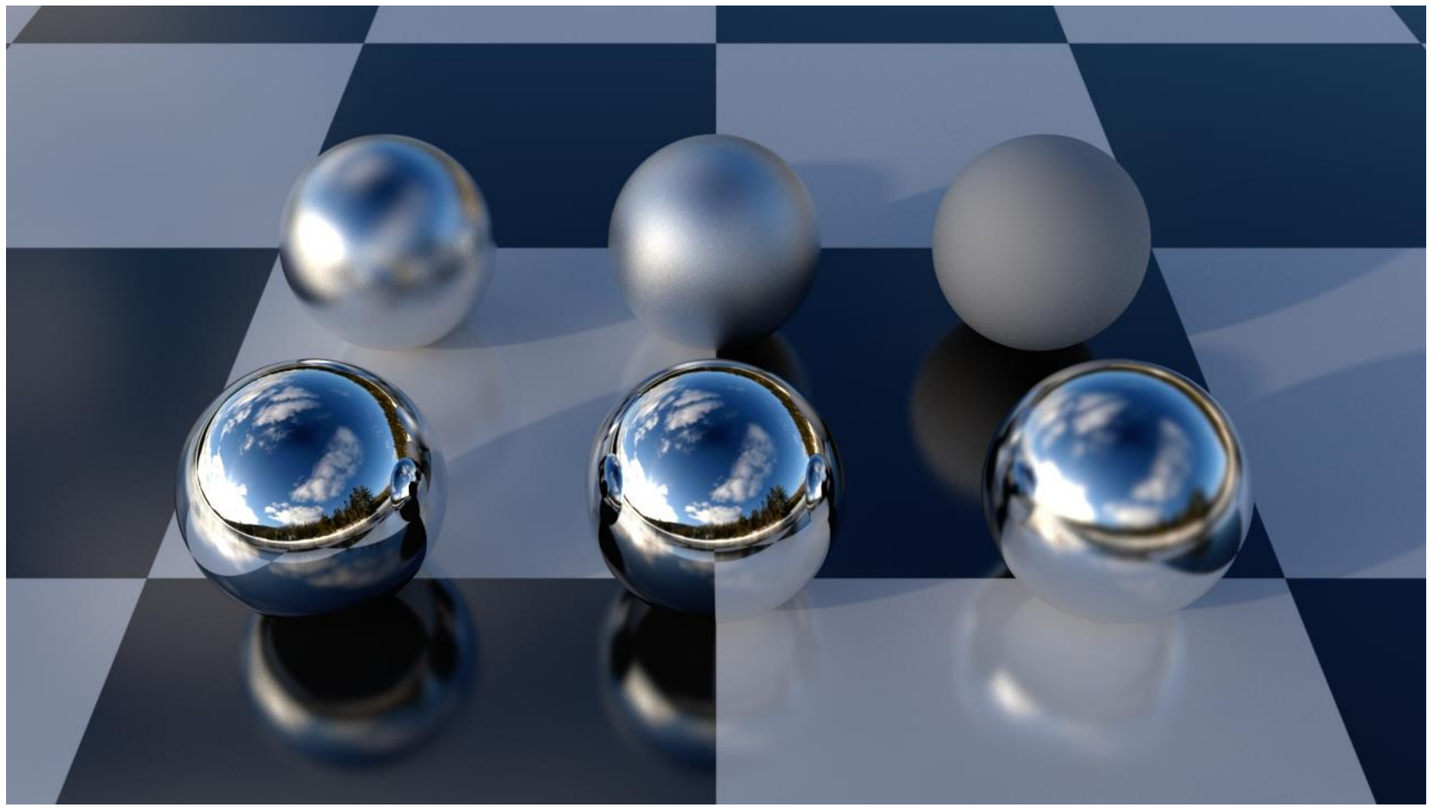
Reflections: ray tracing

Note: this reflection model assumes a very shiny surface.
Light interacts with real-life objects in very interesting ways!

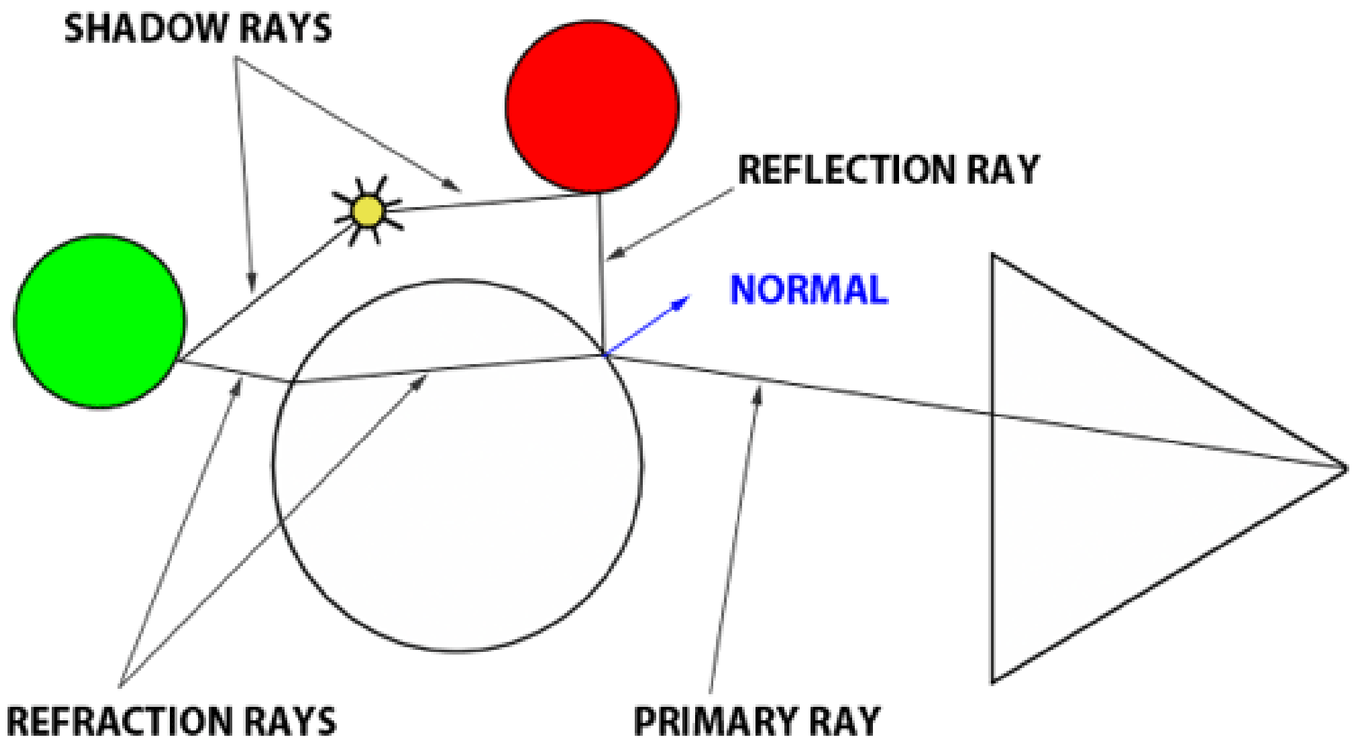


Specular Diffuse Glossy

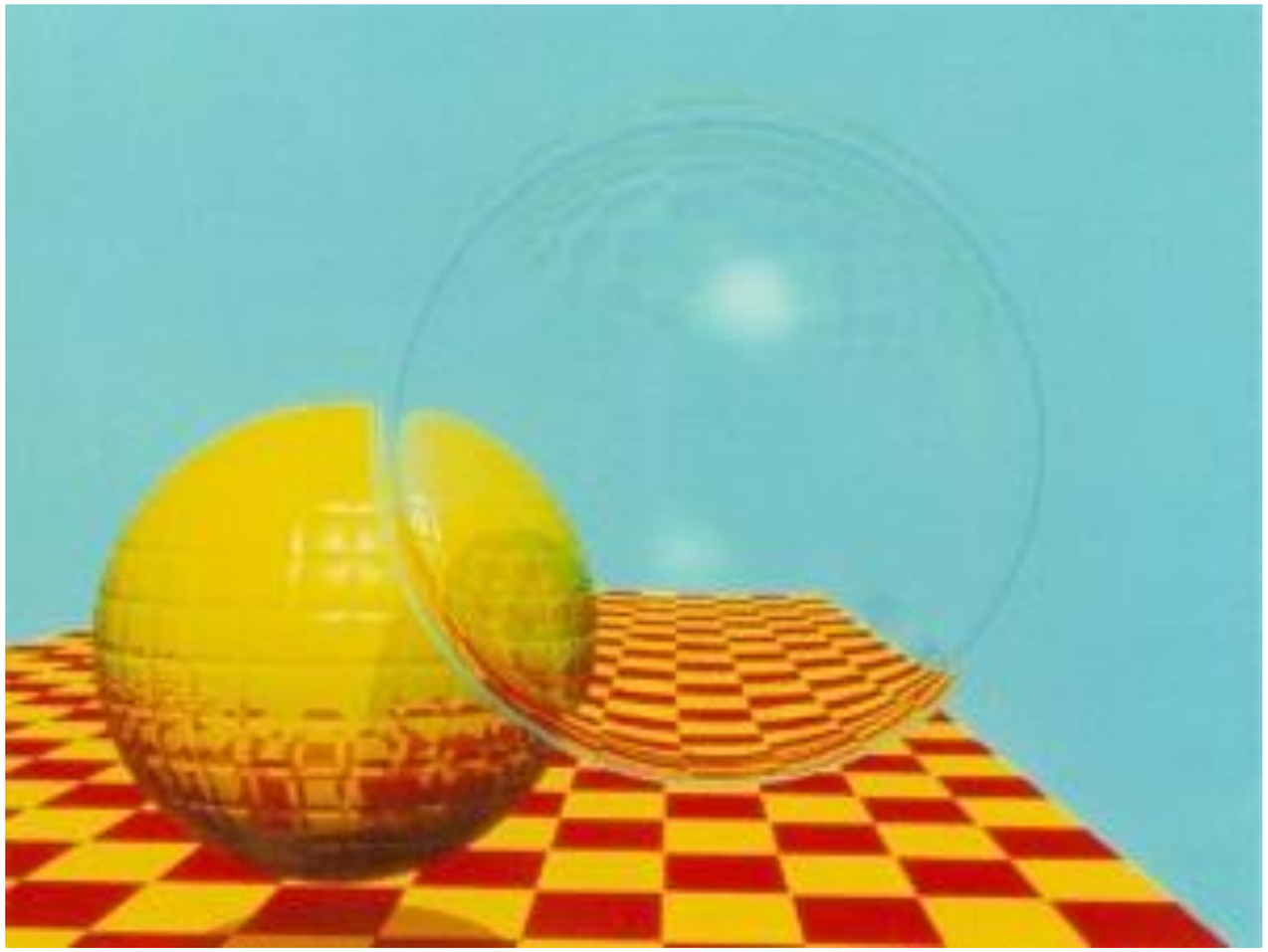
Q: How would you model appearance of a rougher glossy object?



Shadows, Reflections, Refractions: recursive ray tracing



Ray tracing history



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

Why we want real-time ray tracing

Image Credit: Pixar (Cars)

Single general solution rather than a specialized technique for each lighting effect.

Less parameter tweaking (e.g., choosing shadow map resolution)

Scales well (e.g. big pain to manage many shadow maps in scene with many lights)

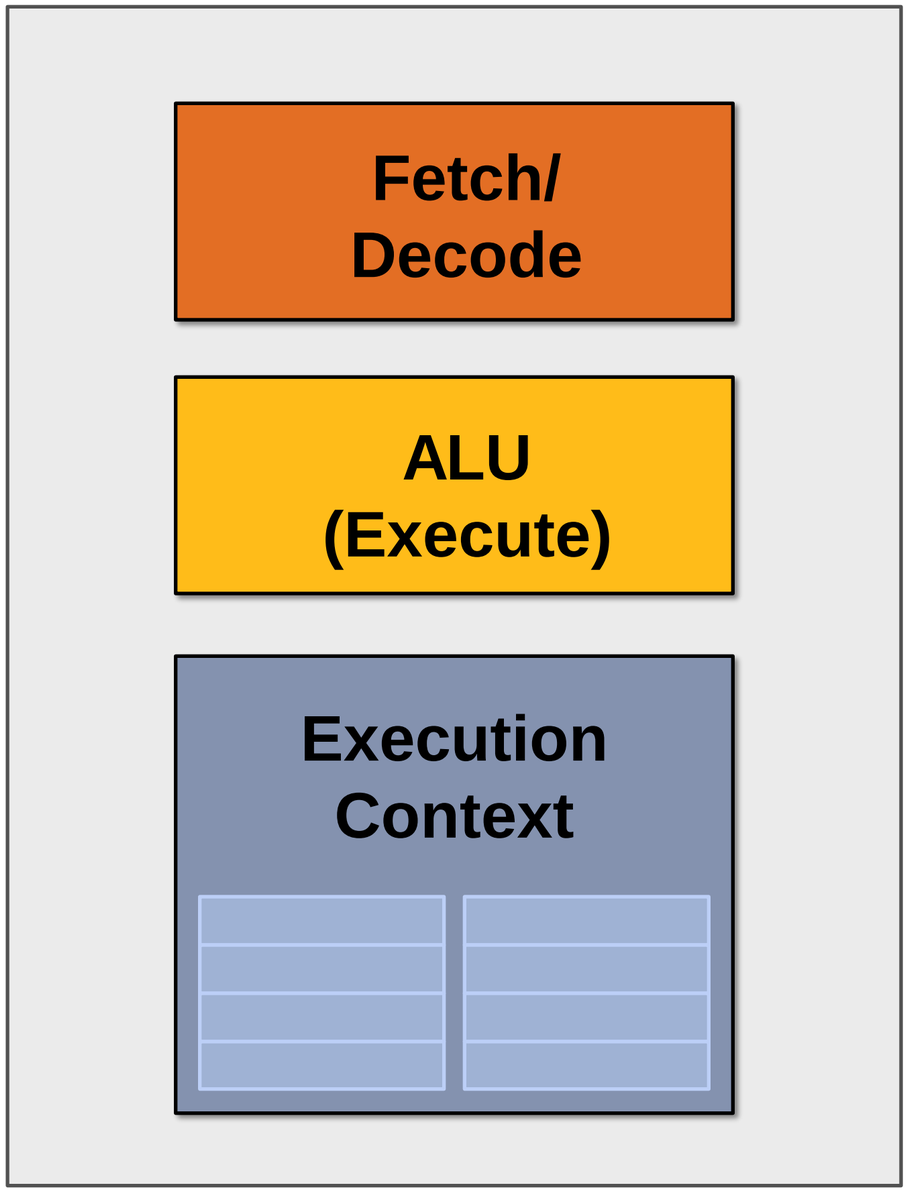
Efficient ray tracing

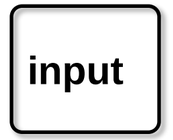
How would you parallelize your ray tracer on a multi-core CPU/GPU?

Image credit: NVIDIA (this ray traced image can be rendered at interactive rates on modern GPUs)

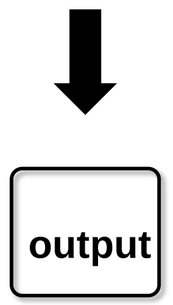
A very high-level look at modern computer architecture

What does a processor do?



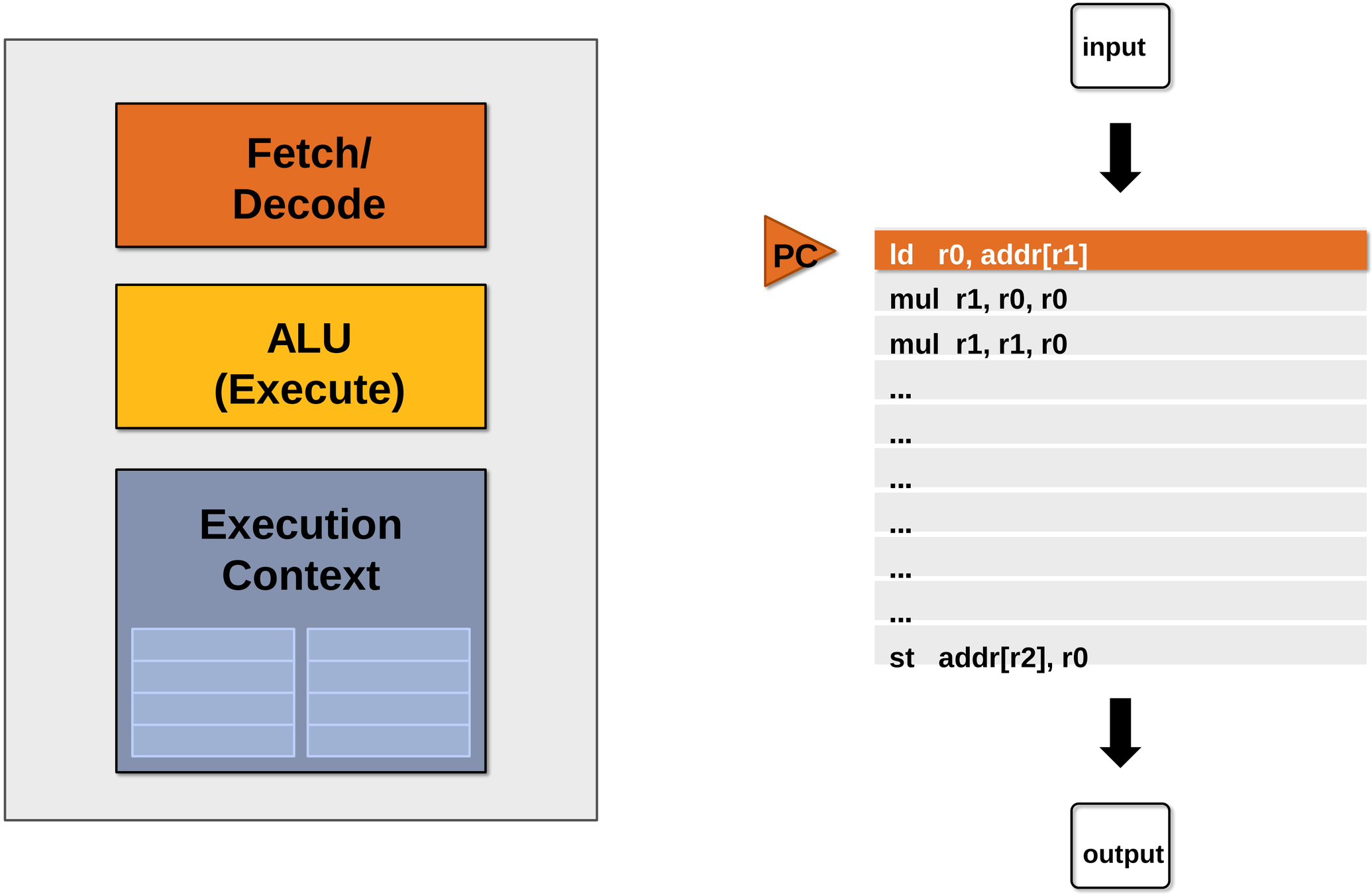



```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```



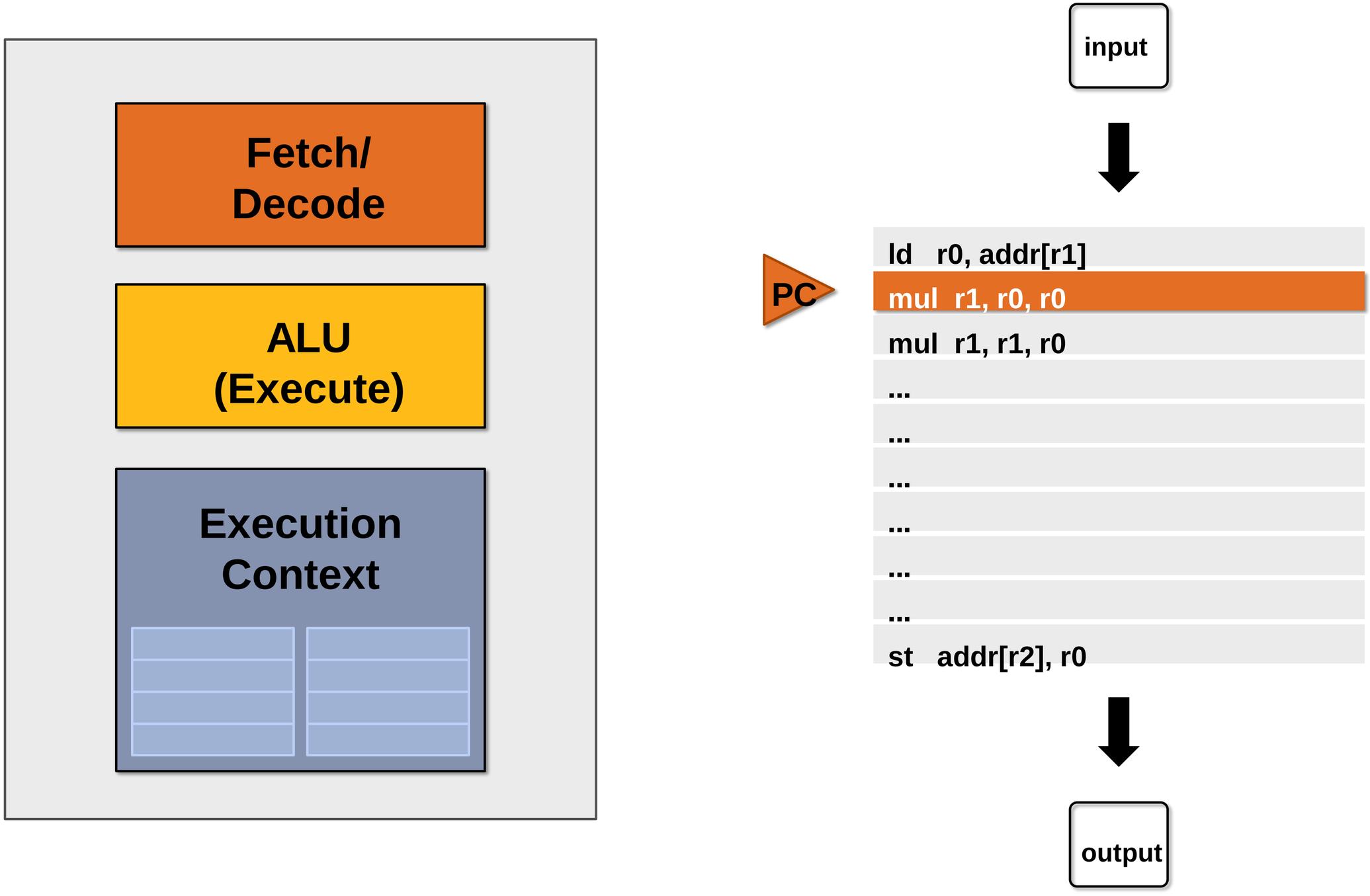
A processor executes an instruction stream

executes one instruction per clock

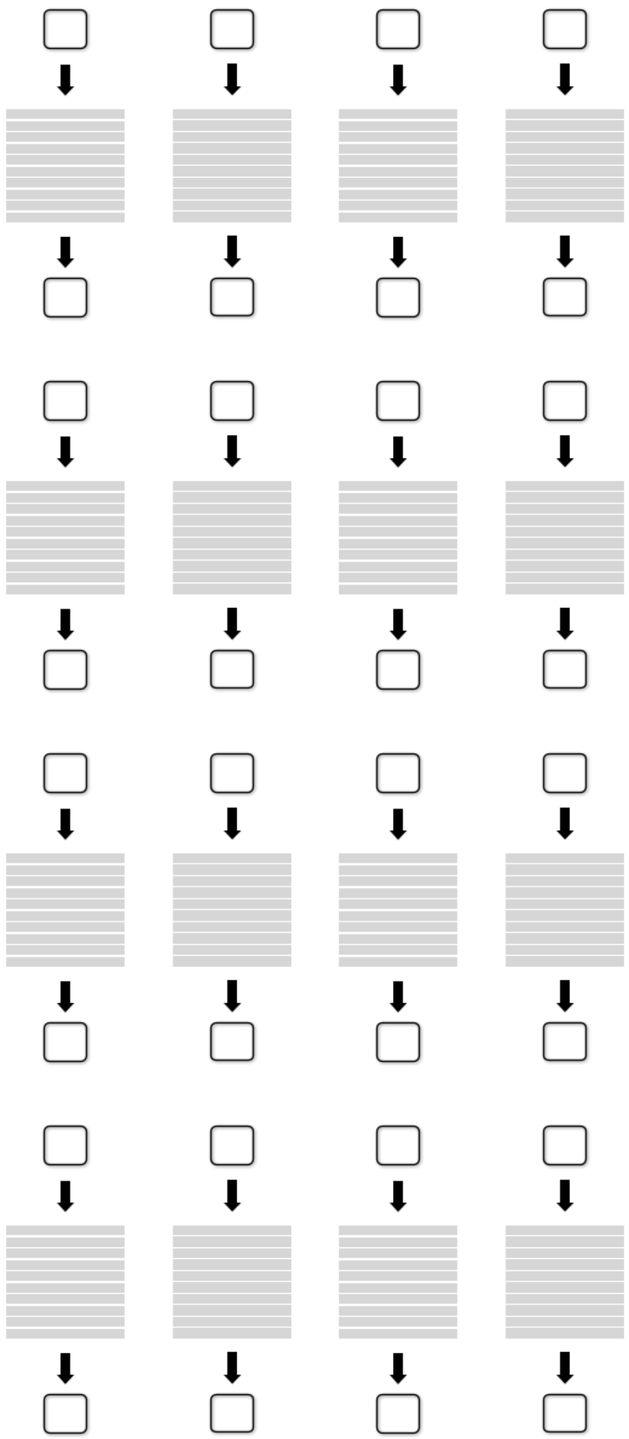


Execute program

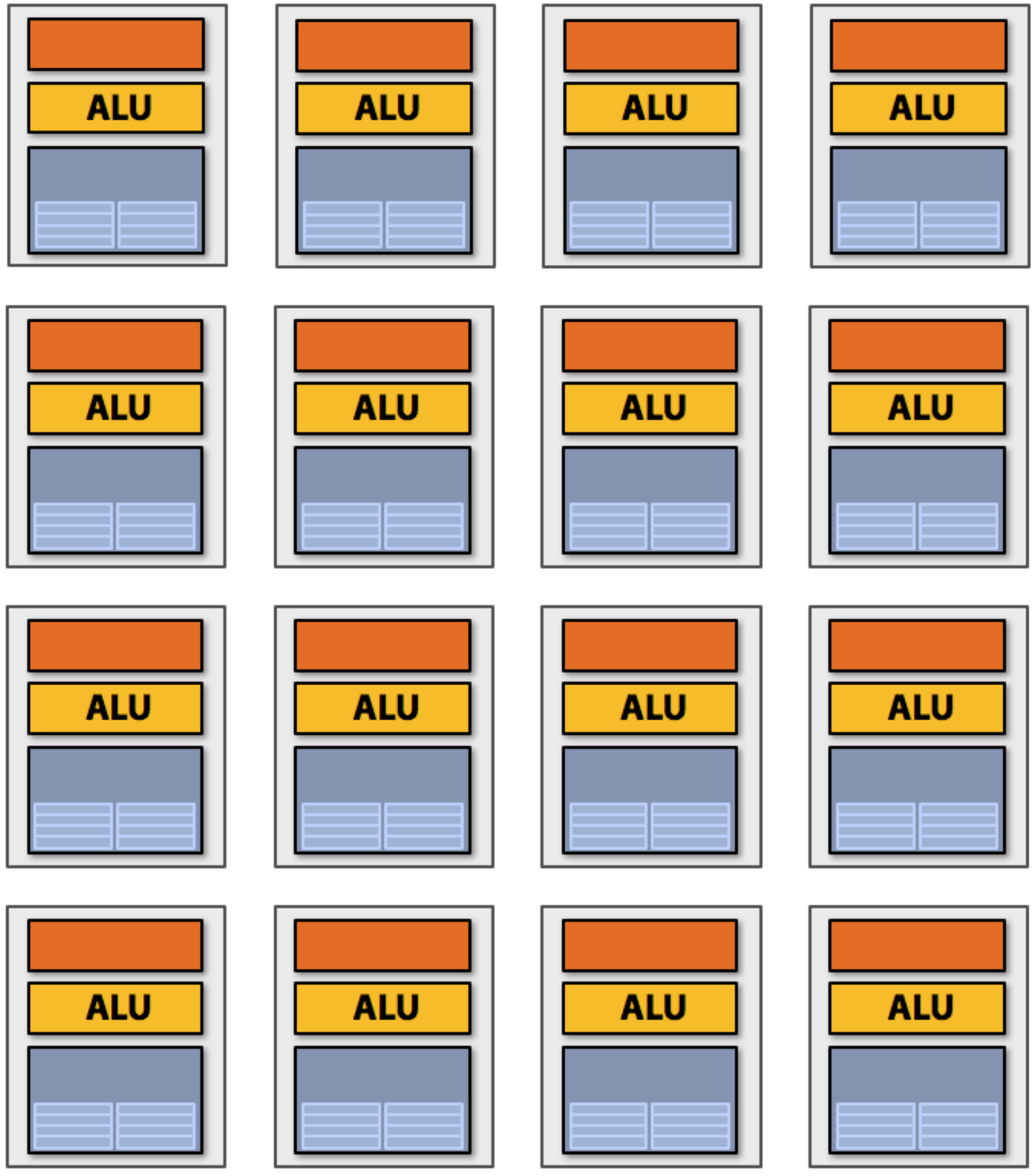
executes one instruction per clock



Sixteen cores: process 16 tasks in parallel



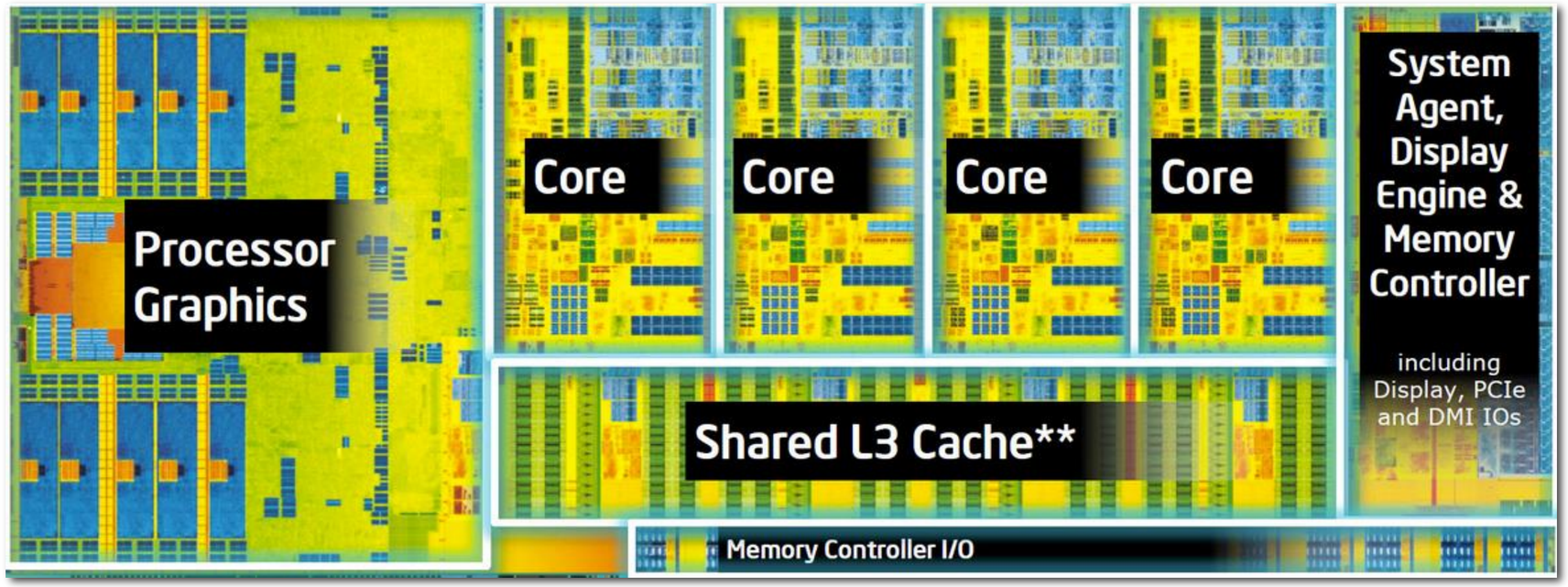
Sixteen tasks processed at once



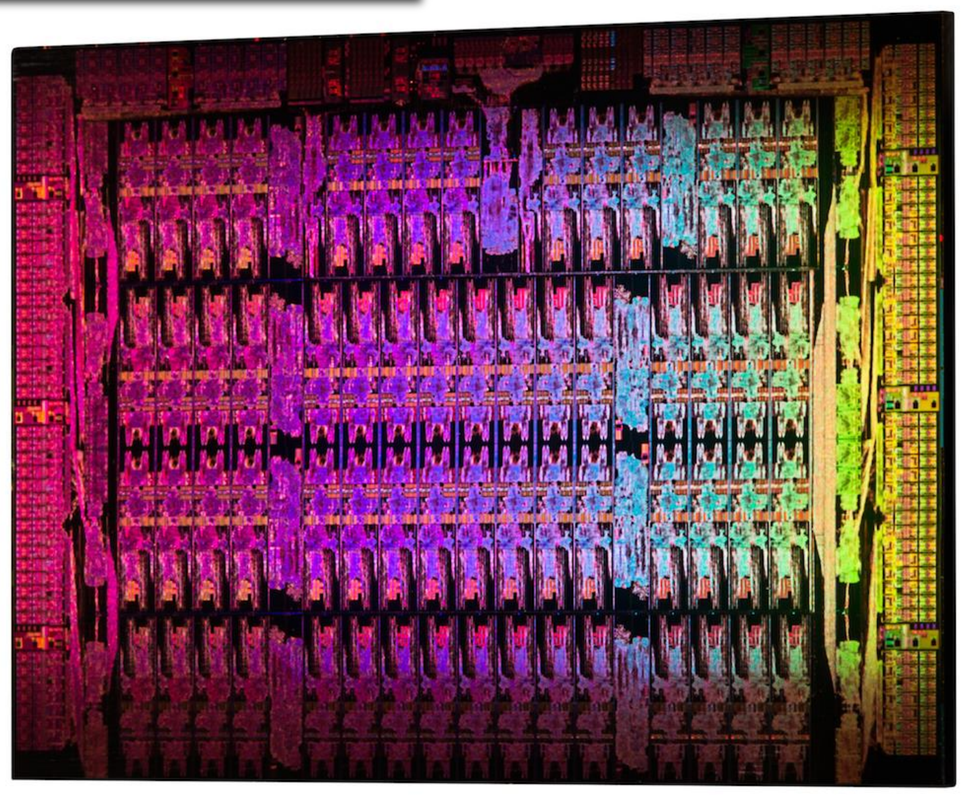
Sixteen cores

Multi-core processors

Intel Core i7 (Haswell): quad-core CPU

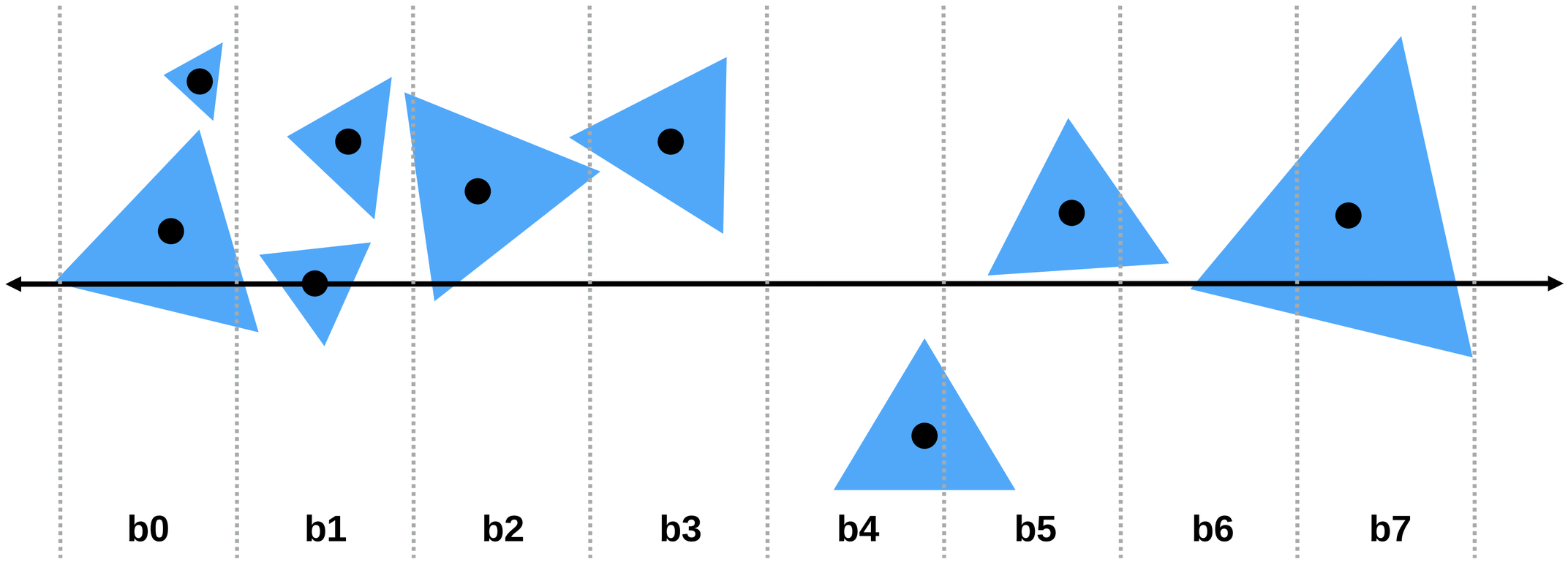


Intel Xeon Phi: 60 core CPU



An efficient ray tracer implementation must use all the cores on a modern processor (this is quite easy)

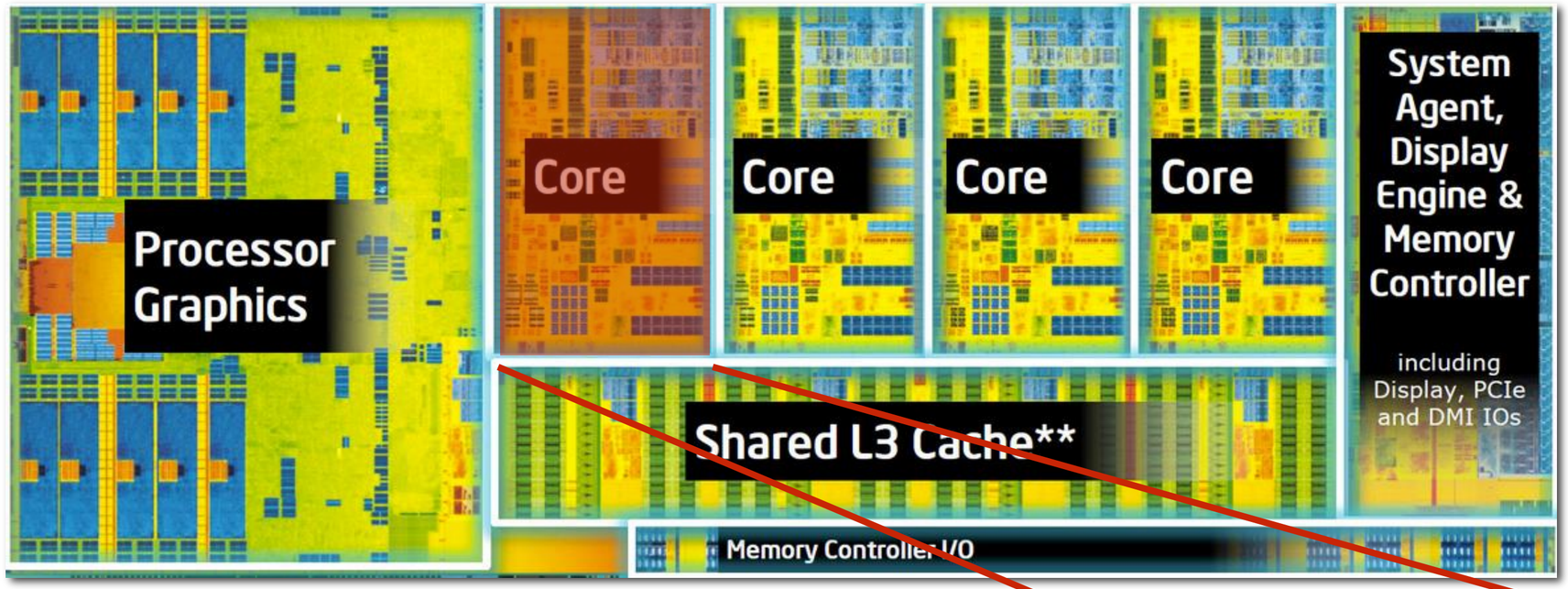
What about building a BVH in parallel?

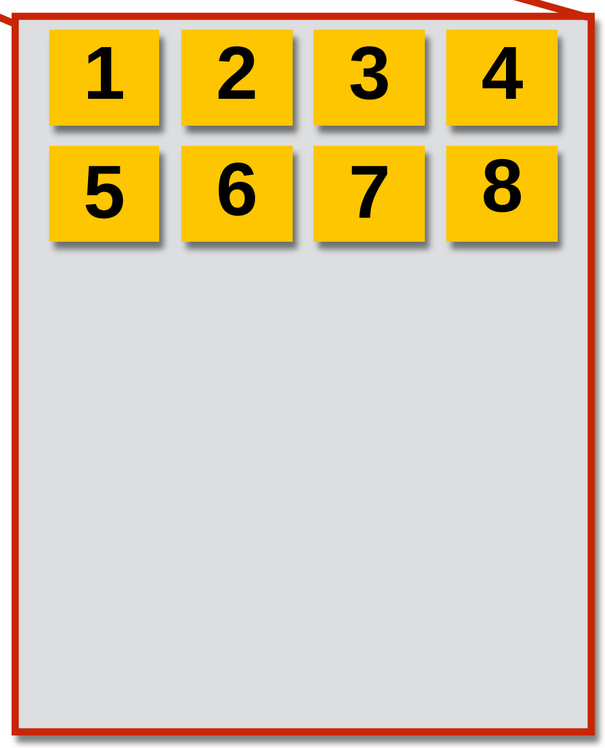



```
Partition(node)
  For each axis: x,y,z:
    initialize buckets
    For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
    For each of the B-1 possible partitioning planes evaluate SAH
  Execute lowest cost partitioning found (or make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

SIMD processing

Single instruction, multiple data





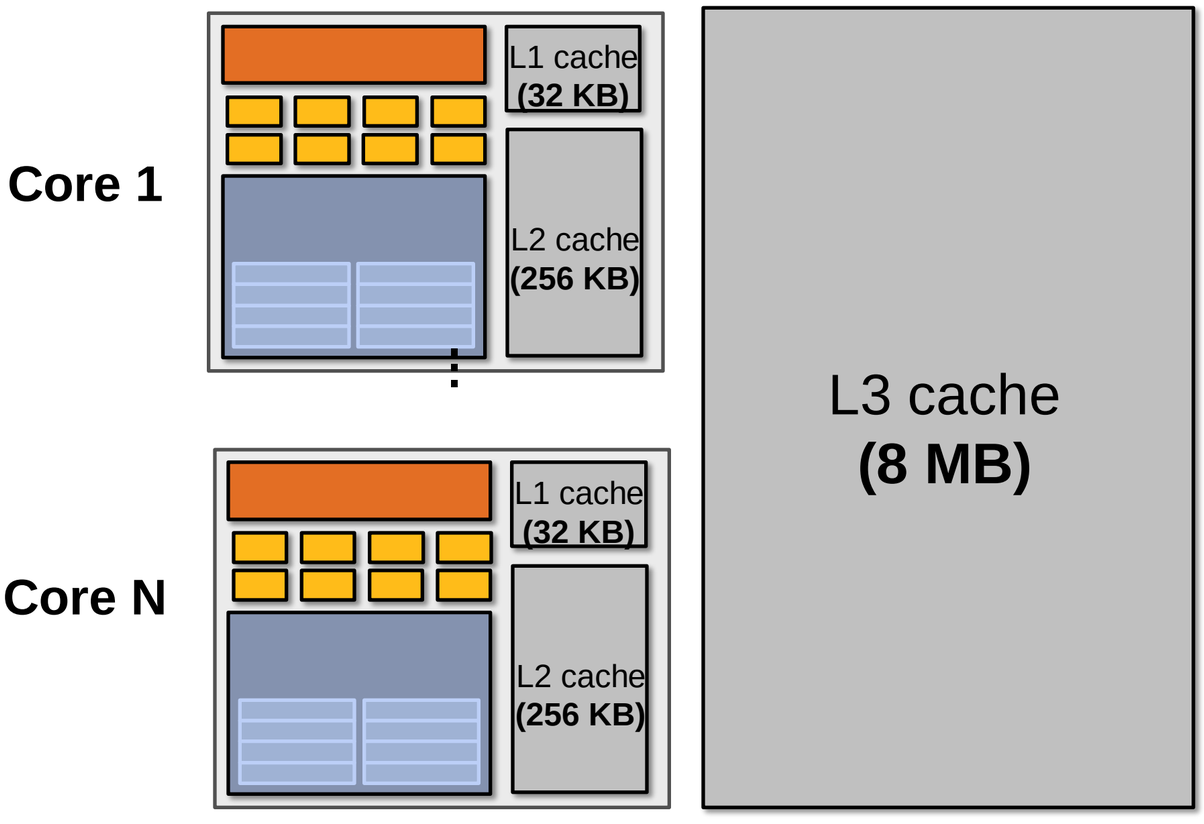
Each core can execute the same instruction simultaneously on multiple pieces of data:

e.g., add vector A to vector B
32-bit addition performed in parallel for each vector element.

An efficient ray tracer implementation must also utilize the SIMD execution capabilities of modern processors

CPUs: up to a factor of 8
GPUs: up to a factor of 32

Accessing memory has high cost



- **High latency: 100's of cycles**
- **Too little bandwidth: modern processors can perform arithmetic much faster than memory can provide data.**

An efficient ray tracer implementation must be careful to reduce memory access costs as much as possible.

Rules of the game

- Many individual processor cores
  - Run tasks in parallel

- SIMD instruction capability
  - Single instruction carried out on multiple elements of an array in parallel (8-wide on modern GPUs, 16-wide on Xeon Phi, 8-to-32-wide on modern GPUs)

- Accessing memory is expensive
  - Processor must wait for data to arrive
  - Role of CPU caches is to reduce wait time (want good locality)

Parallelize ray-box, ray-triangle intersection

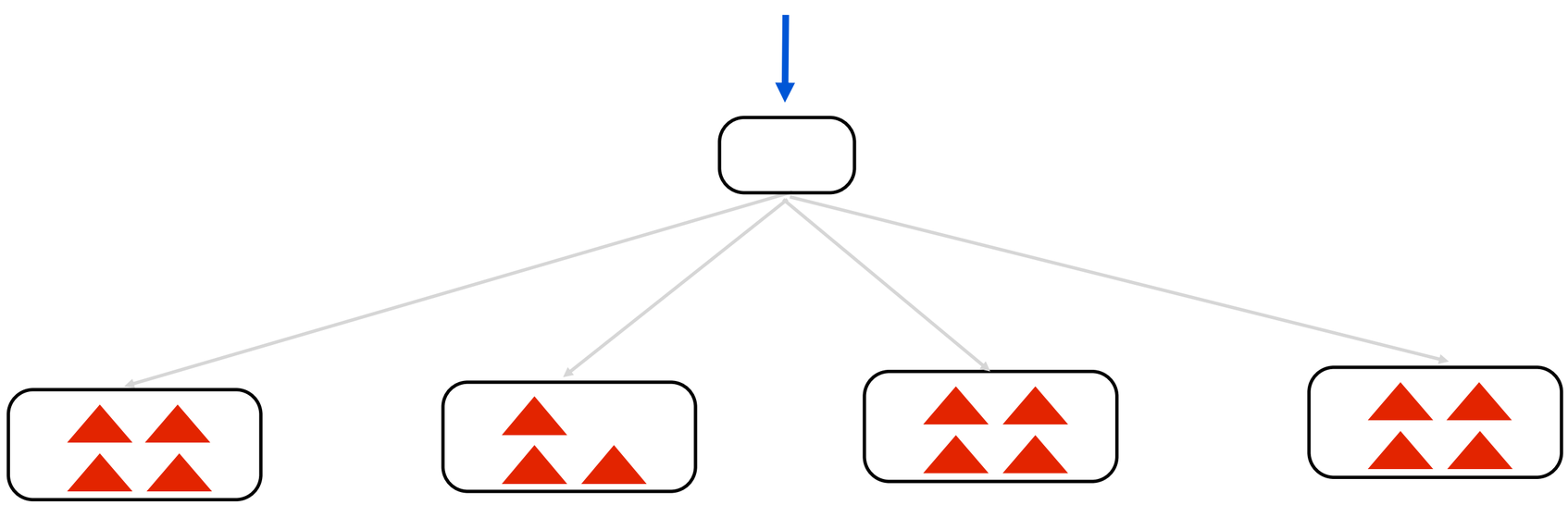
- Given one ray and one bounding box, there are opportunities for SIMD processing
  - (e.g., xyz work, ray-multiple-plane tests, etc.)

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

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

Parallelize over BVH child nodes

- **Idea: use wider-branching BVH (test single ray against multiple child node bboxes in parallel)**
  - **BVH with branching factor 4 has similar work efficiency to branching factor 2**
  - **BVH with branching factor 8 or 16 is significantly less work efficient (diminished benefit of leveraging SIMD execution)**

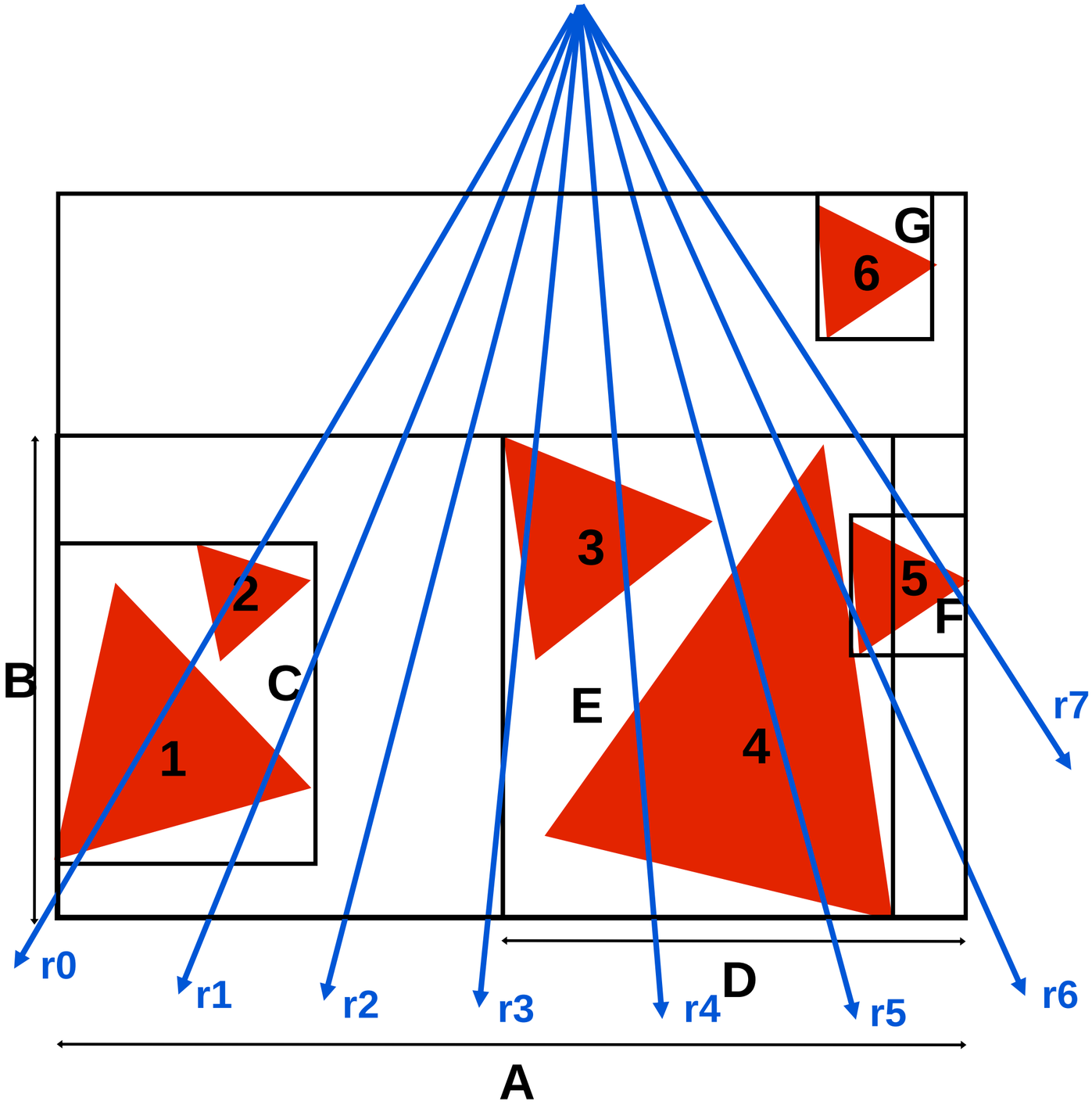


[Wald et al. 2008]

Parallelize across rays

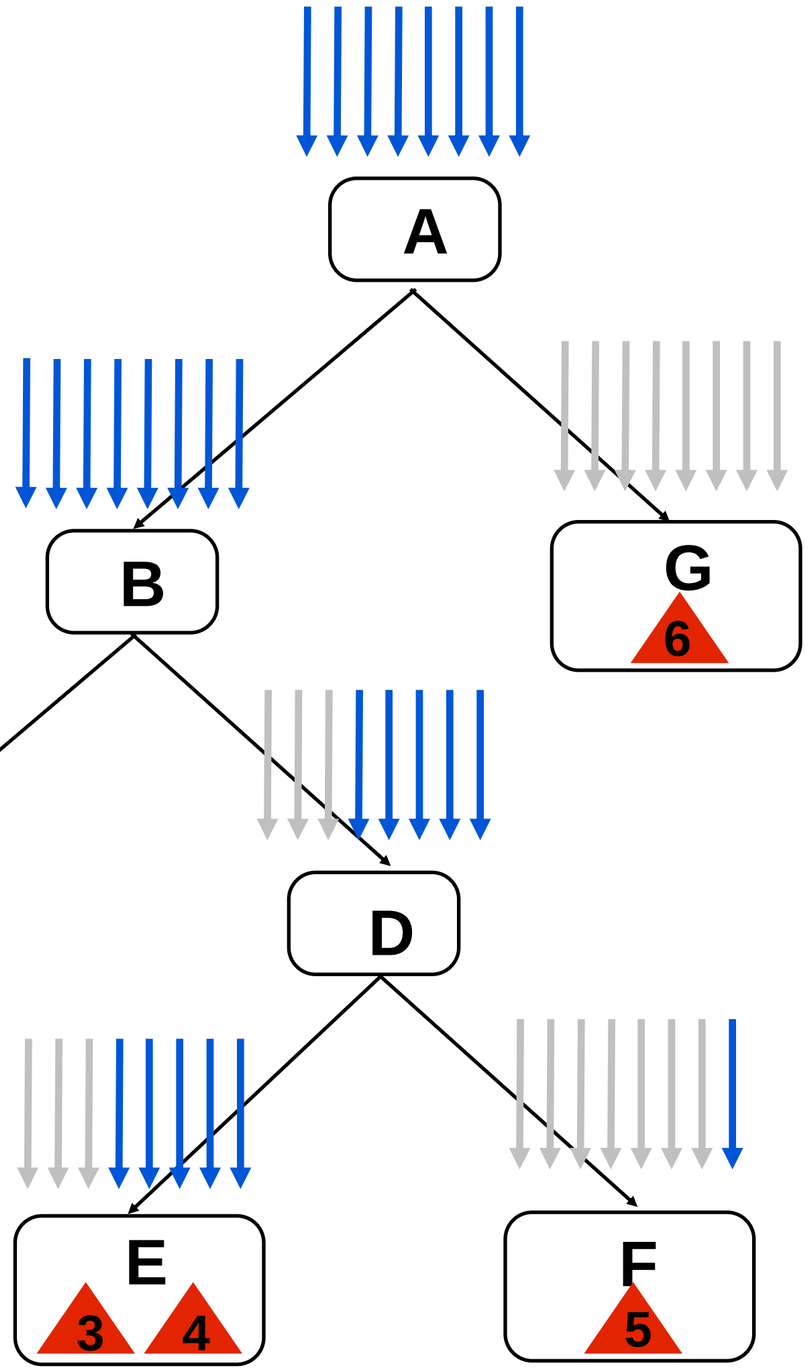
- Simultaneously intersect multiple rays with scene
- One possible approach: ray packets
  - Code is explicitly written to trace N rays at a time, not 1 ray

Ray packet tracing



[Wald et al. 2001]

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

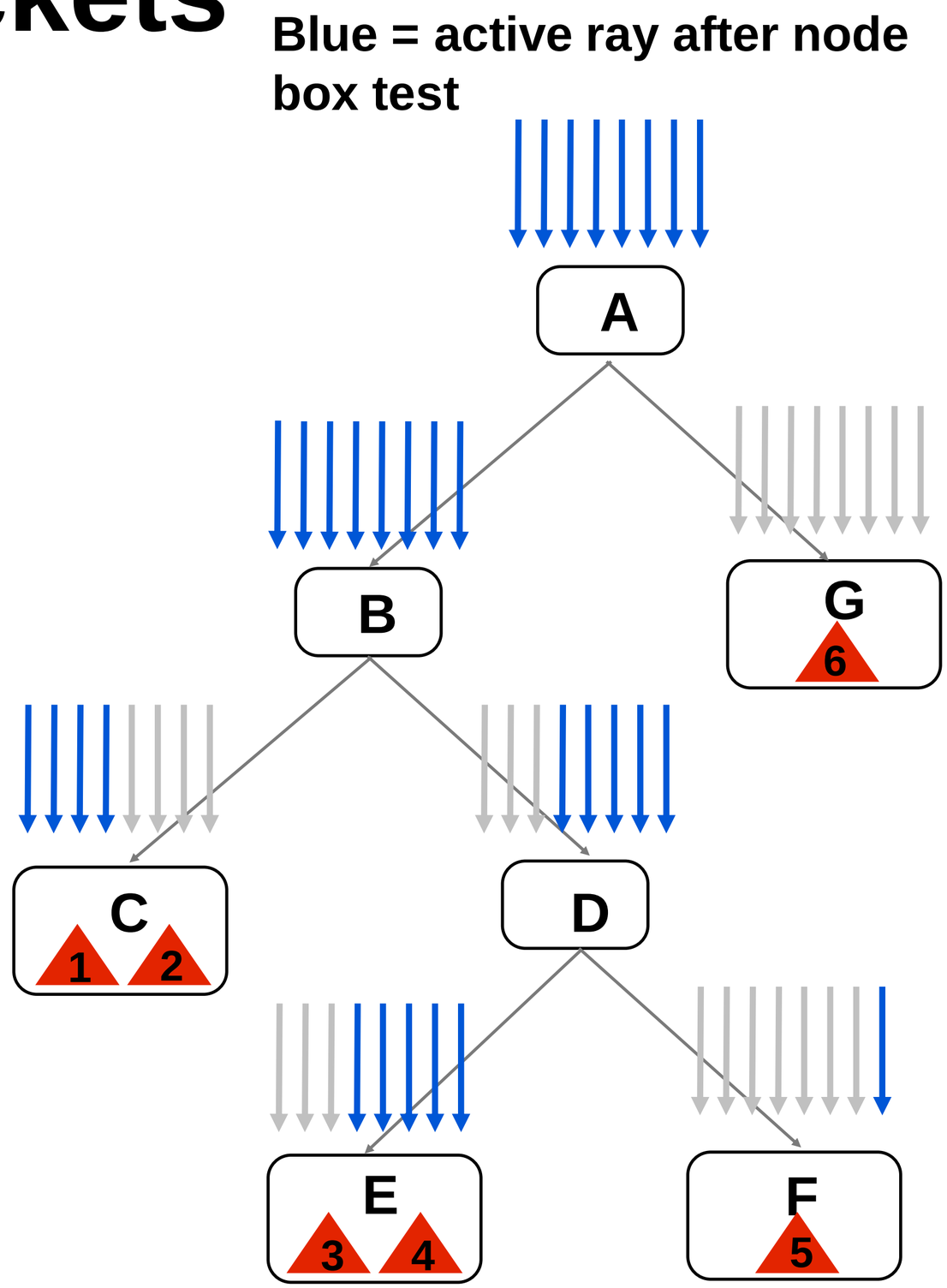
Advantages of packets

- **Enable 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**

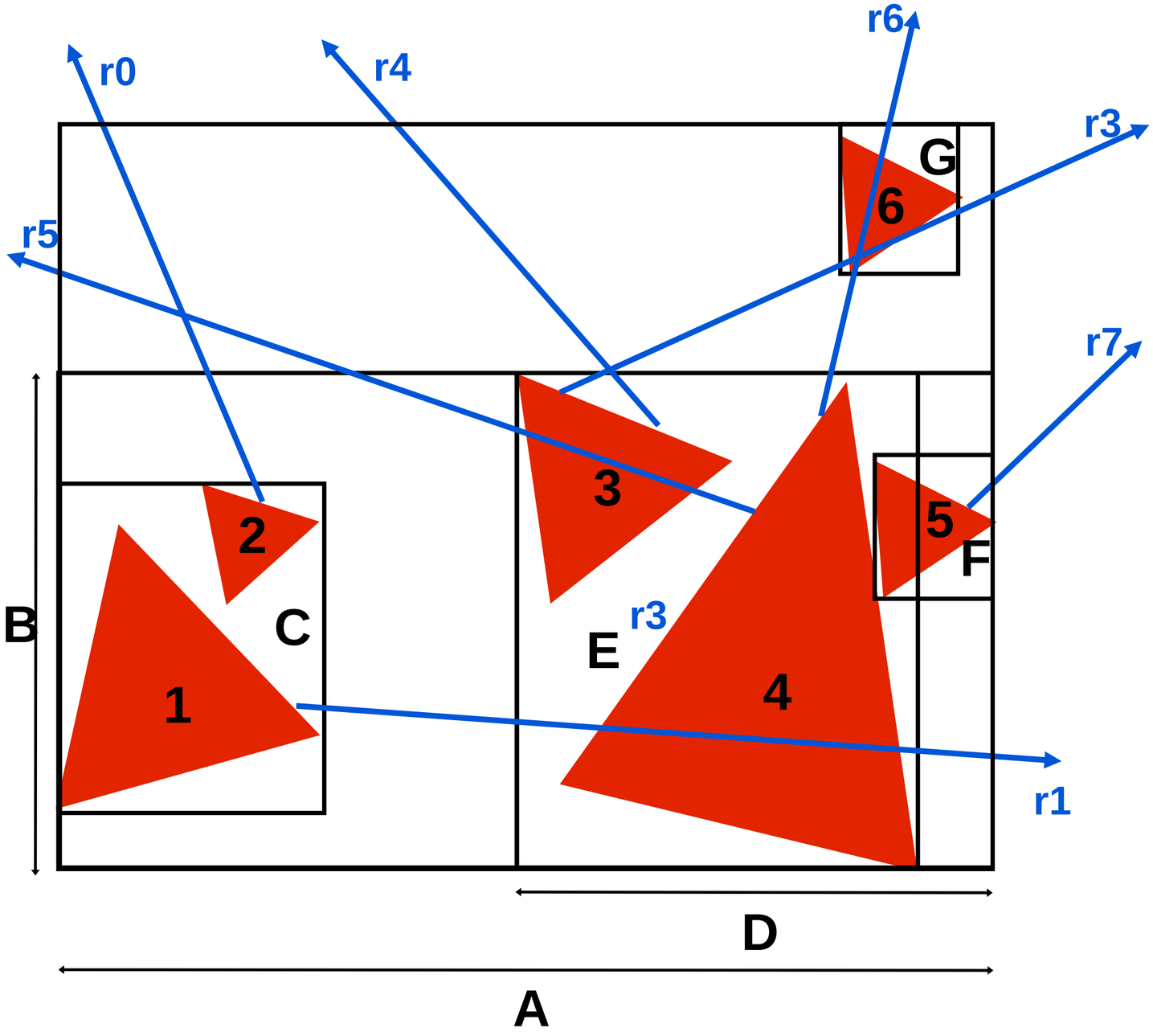
Disadvantages of packets

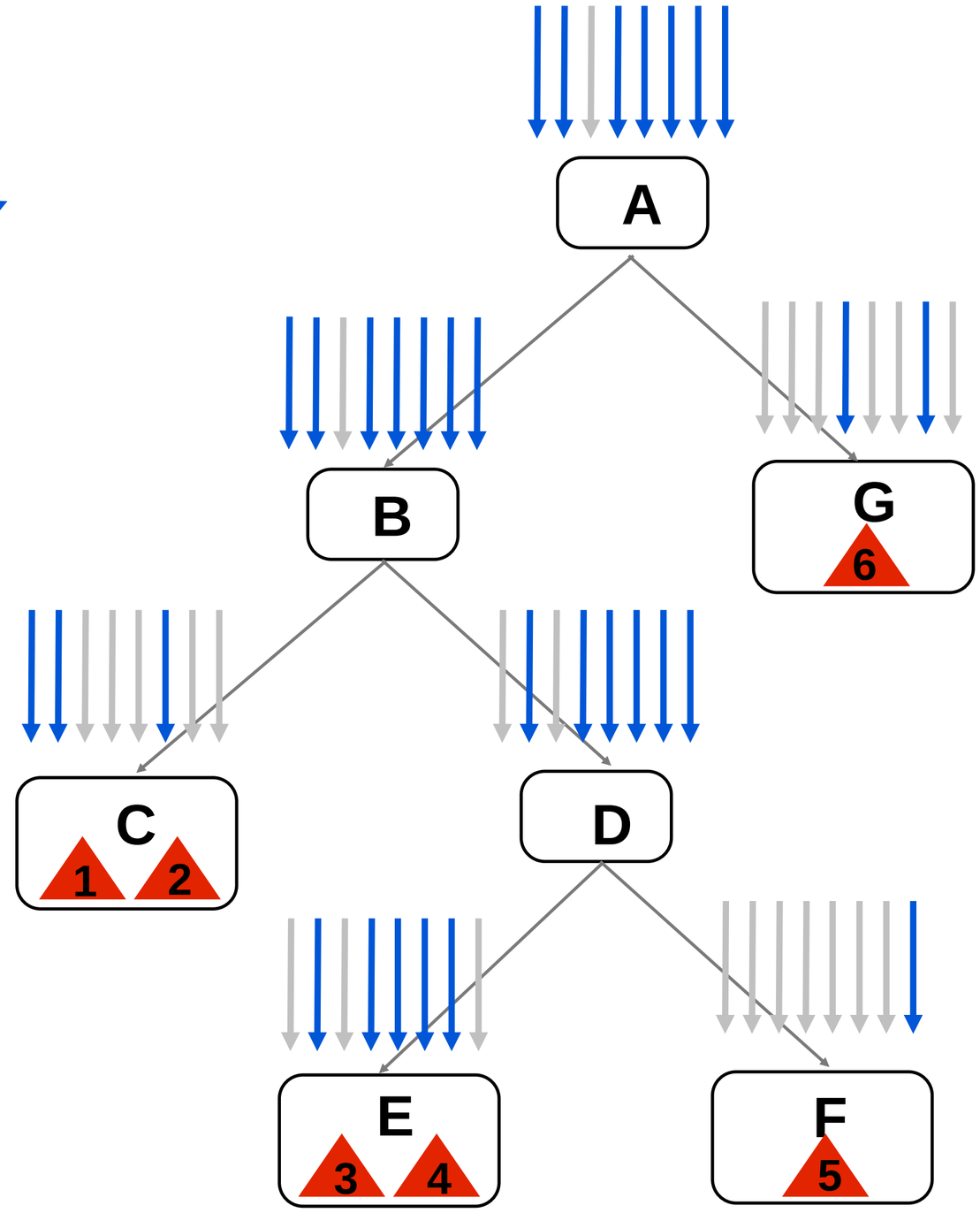
- If any ray must visit a node, it drags all rays in the packet along with it)

- Not all SIMD lanes doing useful work



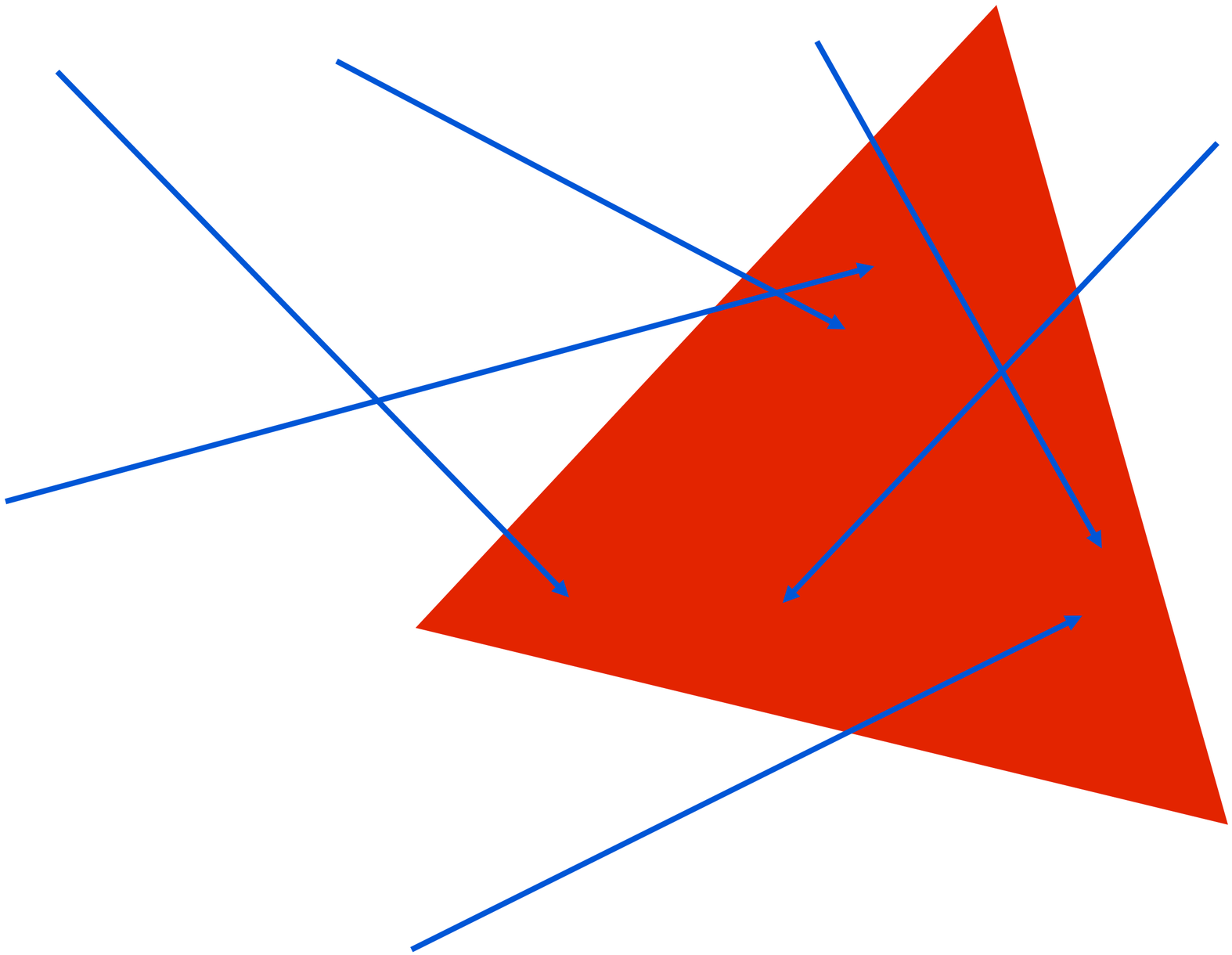
Ray packet tracing: incoherent rays





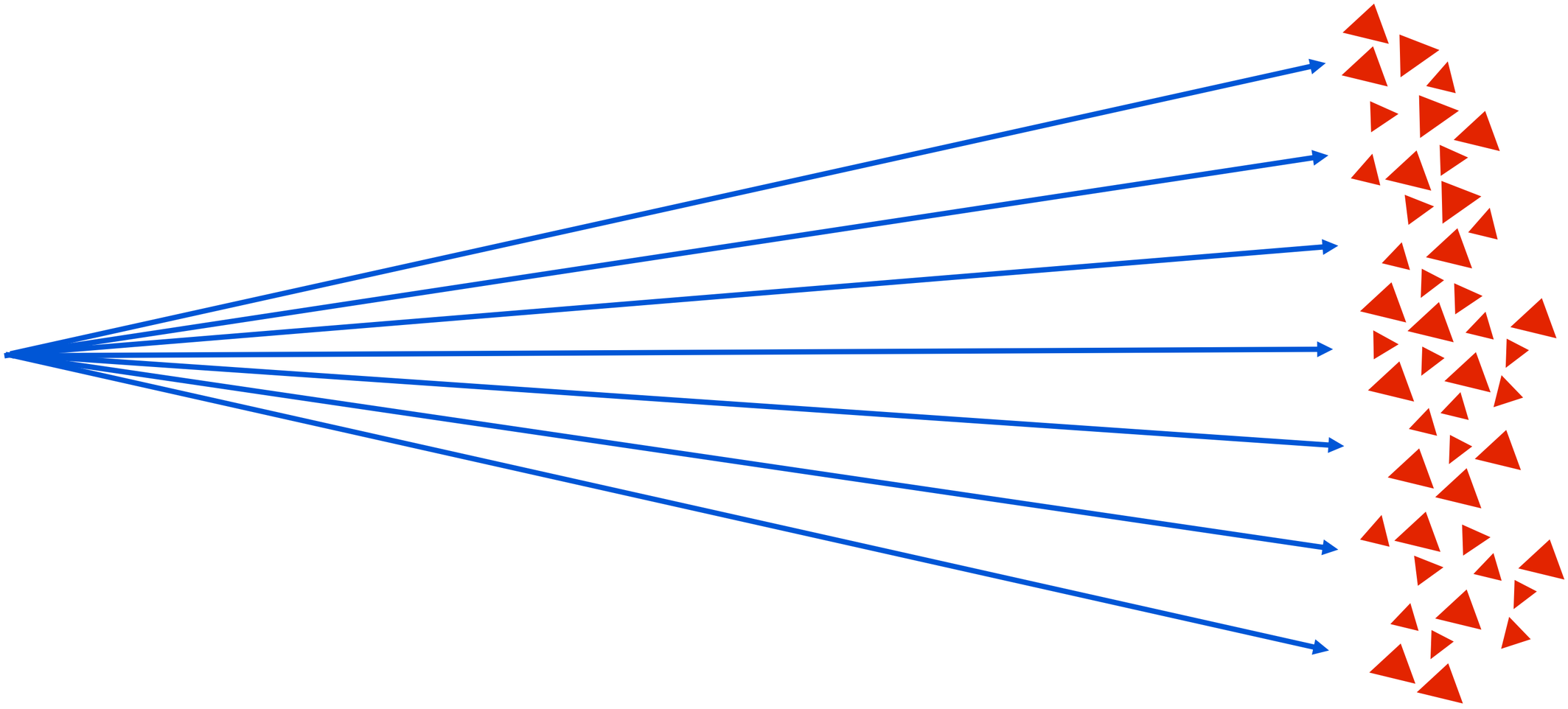
When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (So all eight rays visit all tree nodes! No culling benefit!)

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



Random rays are “coherent” with respect to the BVH if the scene is one big triangle!

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



Camera rays become “incoherent” with respect to lower nodes in the BVH if a scene is overly detailed

Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

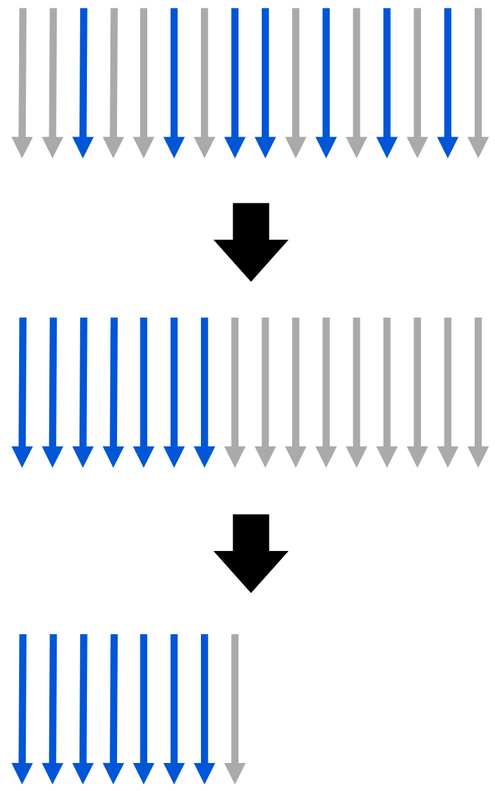
- **Increases SIMD utilization**
- **Amortization benefits of smaller packets, but not large packets**

Example: consider 8-wide SIMD processor and 16-ray packets
(2 SIMD instructions required to perform each operation on all rays in packet)

16-ray packet: 7 of 16 rays active

Reorder rays, recompute intervals/bounds for active rays

Continue tracing with 8-ray packet: 7 of 8 rays active



[Boulos et al. 2008]

Giving up on packets

- **Even with reordering, ray coherence during BVH traversal will diminish**
  - **Diffuse bounces result in essentially random ray distribution**
  - **High-resolution geometry encourages incoherence near leaves of tree**
- **In these situations there is little benefit to packets (can even decrease performance compared to single ray code)**

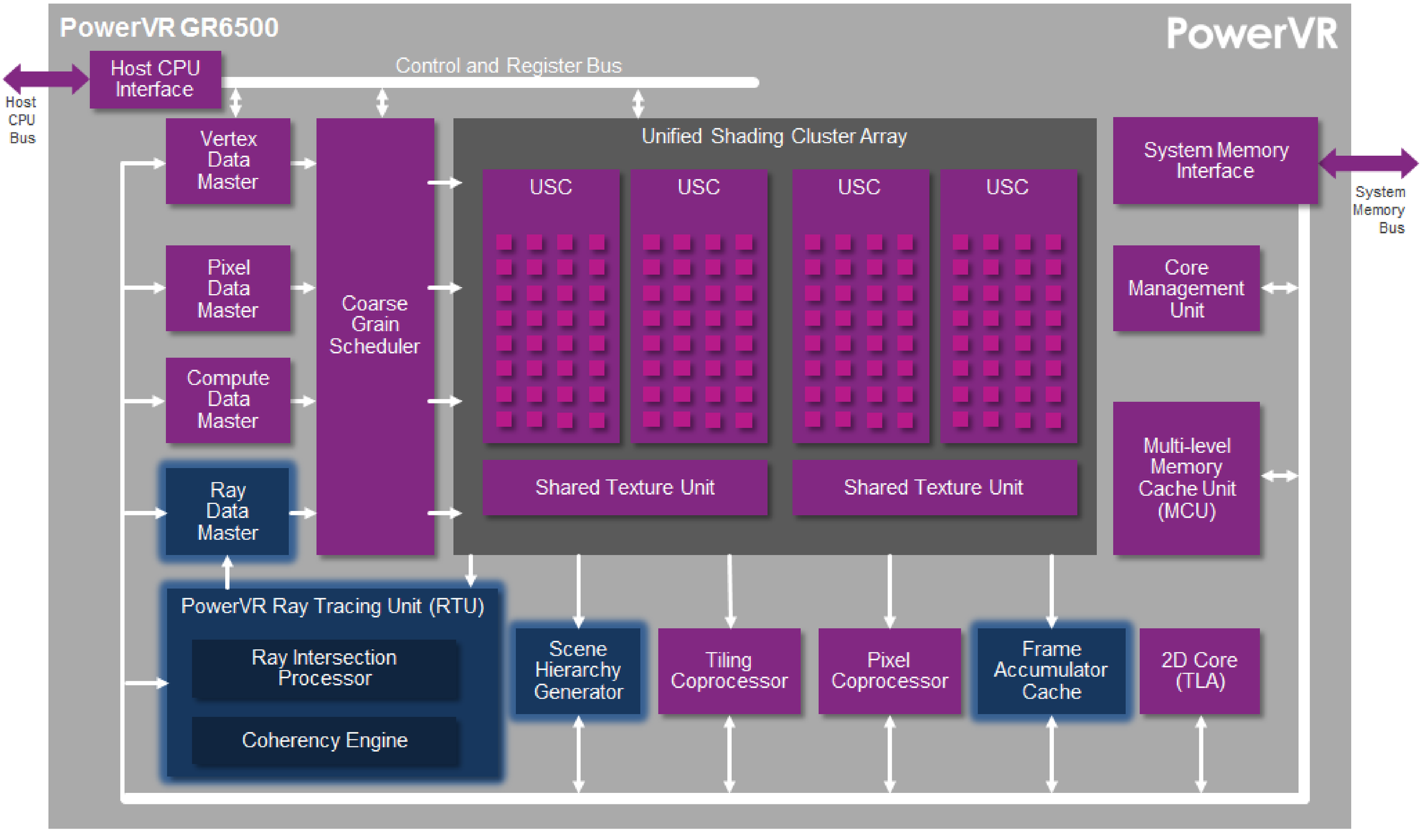
Packet tracing best practices

- **Use large packets for eye/reflection/point light shadow rays or higher levels of BVH**
  - **Ray coherence always high at the top of the tree**

- **Switch to single ray when packet utilization drops below threshold**
  - **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**
  - **Reordering allows packets to provide benefit deeper into tree**
  - **Not often used in practice due to high implementation complexity**

Emerging hardware for ray tracing

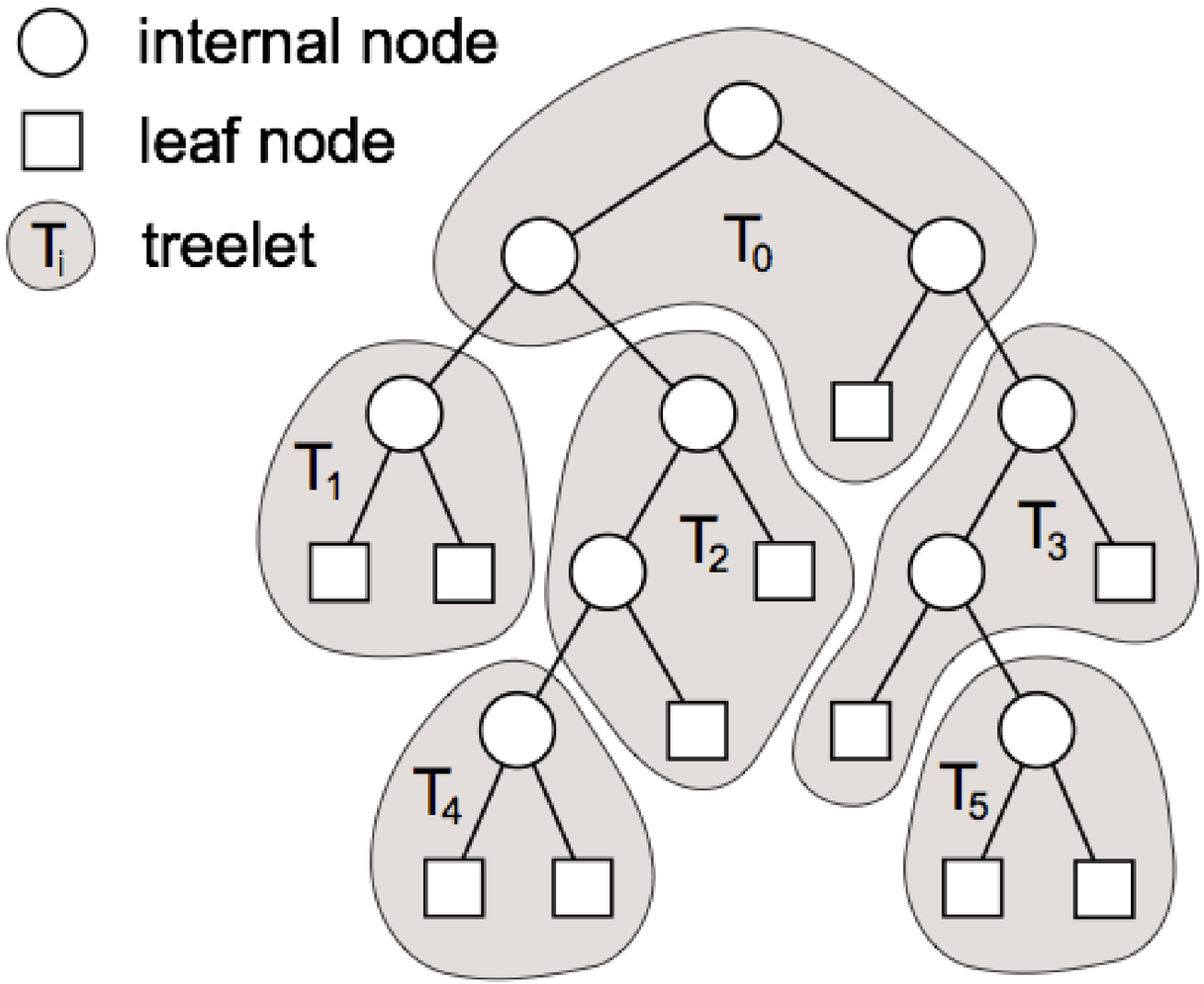


Emerging hardware for ray tracing

- **Modern implementations:**
  - **Trace single rays, not ray packets (assume most rays are incoherent rays…)**
- **Two areas of focus:**
  - **Custom logic for accelerating ray-box and ray-triangle tests**
    - **MIMD designs: wide SIMD execution not beneficial**
  - **Support for efficiently reordering ray-tracing computations to maximize memory locality (ray scheduling)**

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 queued rays with treelet**

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

[Pharr 1997, Navratil 07, Alia 10]

Next up: lighting, materials, so much fun!