

Lecture 22:

High-Performance Ray Tracing

**Visual Computing Systems
CMU 15-769, Fall 2016**

Rasterization and ray casting are two algorithms for solving the same problem: determining “visibility from a camera”

Recall triangle visibility:

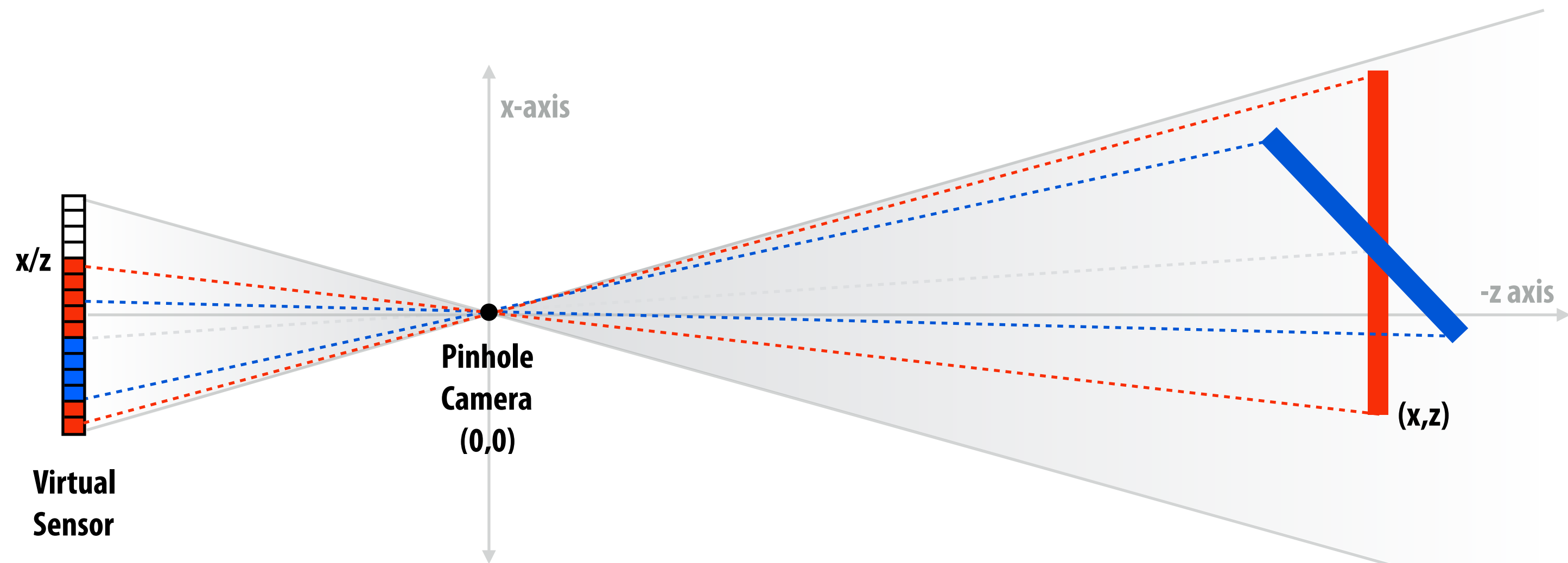
**Question 1: what samples does the triangle overlap?
("coverage")**

Sample

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

The visibility problem

- **What scene geometry is visible at each screen sample?**
 - What scene geometry projects into a screen pixel? (coverage)
 - Which geometry is visible from the camera at that pixel? (occlusion)



Basic rasterization algorithm

Sample = 2D point

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

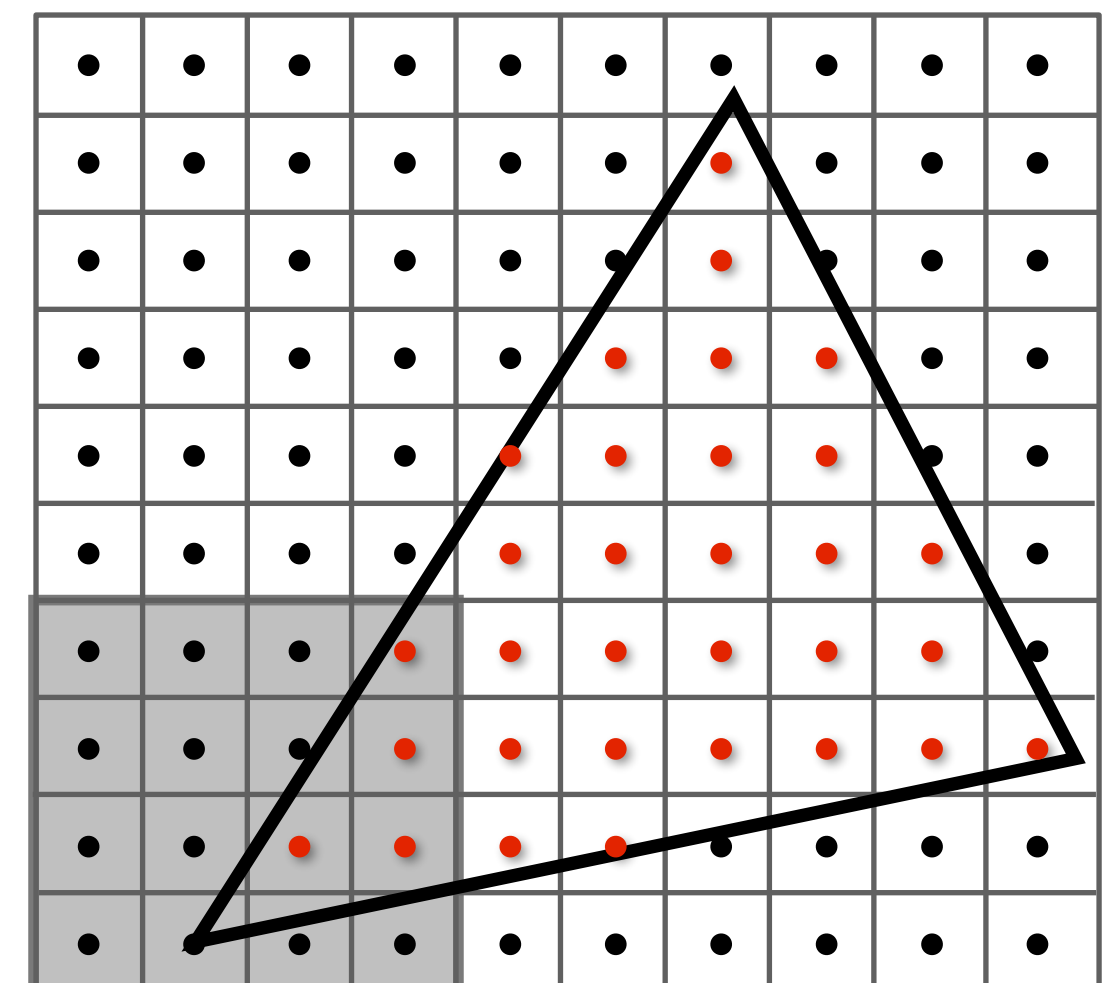
Occlusion: depth buffer

```
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: triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:    // loop 2: visibility samples
        if (t_proj covers s)
            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)

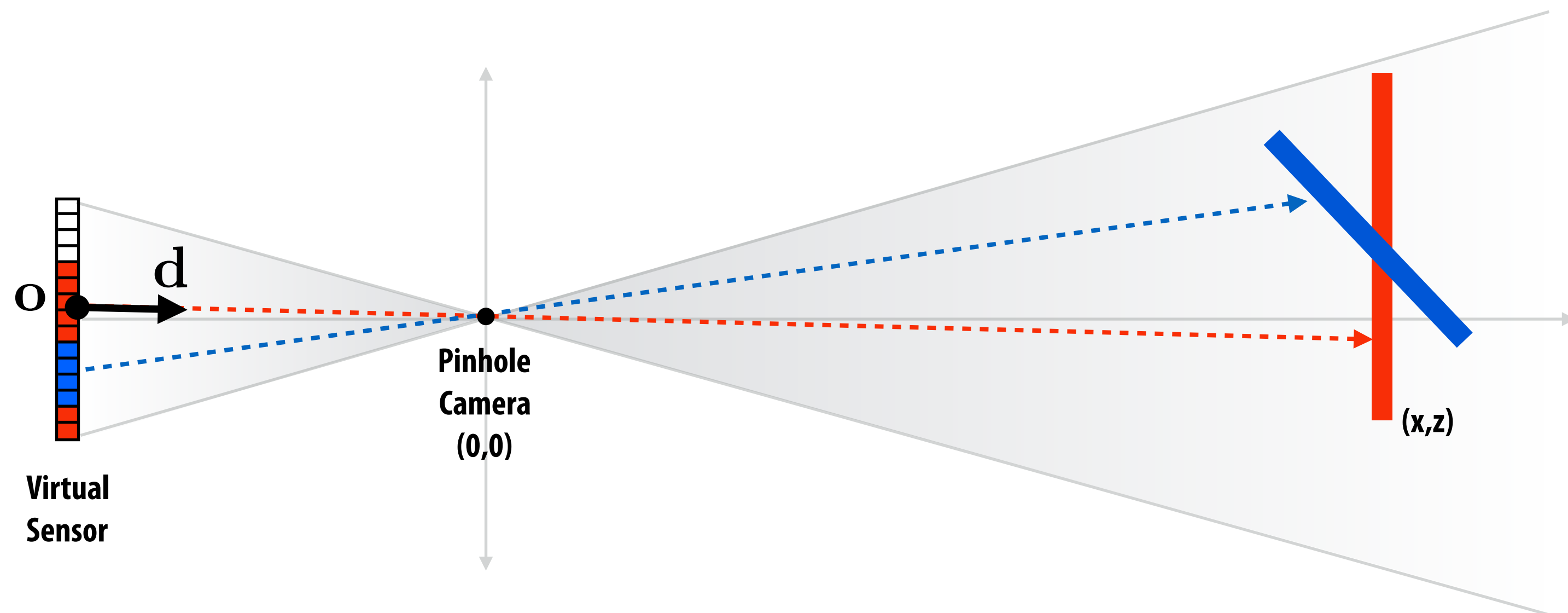
Recall: modern rasterization algorithms are hierarchical.
(for each tile of image, if triangle overlaps tile, check all samples in tile)



The visibility problem (described differently)

■ In terms of casting rays from the 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 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: 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: 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! (+ math in 3D)

“Given a ray, find the closest triangle it hits”

The brute force “for each triangle” loop is typically implemented using a search acceleration structure. (A rasterizer’s “for each sample” inner loop is not just a loop over all screen samples either.)

Recall: rendering as a triple for-loop

Naive “rasterizer”:

```
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: triangles
    t_proj = project_triangle(t)
    for each sample s in frame buffer:      // loop 2: visibility samples
        if (t_proj covers s)
            for each light l in scene:      // loop 3: lights
                accumulate contribution of light l to surface appearance
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```

Naive “ray caster”:

```
initialize color[]                          // store scene color for all samples
for each sample s in frame buffer:          // loop 1: visibility samples (rays)
    ray r = ray from s through pinhole aperture out into scene
    r.closest = INFINITY                    // only store closest-so-far for current ray
    r.triangleId = NULL;
    for each triangle t in scene:           // loop 2: triangles
        if (intersects(r, t)) {             // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.closest)
                update r.closest and r.triangleId = t;
        }
    for each light l in scene:              // loop 3: lights
        accumulate contribution of light l to appearance of intersected surface r.triangleId
    color[s] = surface color of r.triangleId at hit point;
```

Basic rasterization vs. basic ray casting

■ Basic rasterization:

- Stream over triangles in 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:

- Stream over screen samples
 - Never have to store closest depth so far for the entire screen (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)

■ Modern high-performance implementations of rasterization and ray-casting embody very similar techniques

- Hierarchies of rays/samples
- Hierarchies of geometry

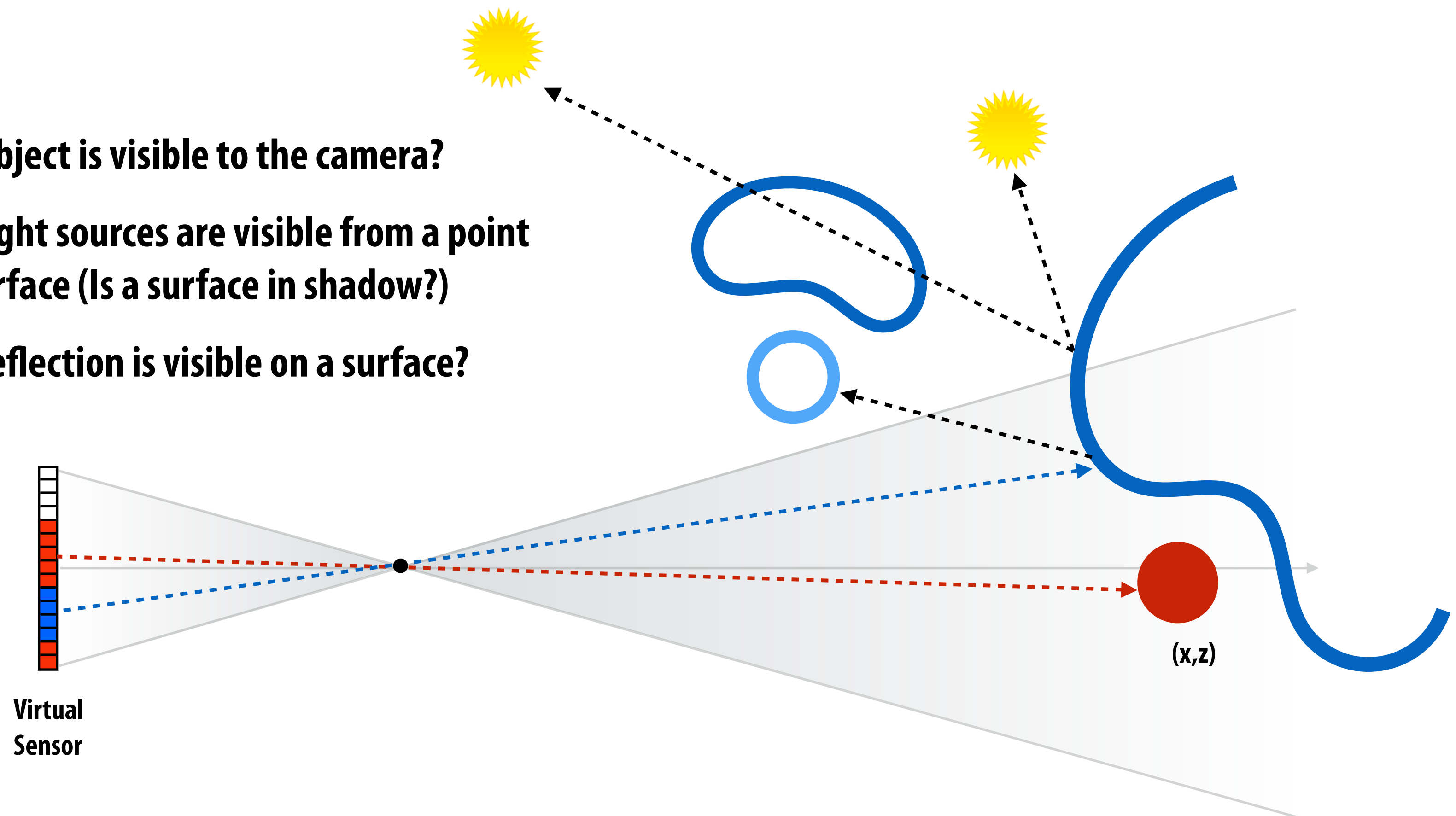
Ray-scene intersection is a general visibility primitive

What object is visible along this ray?

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?

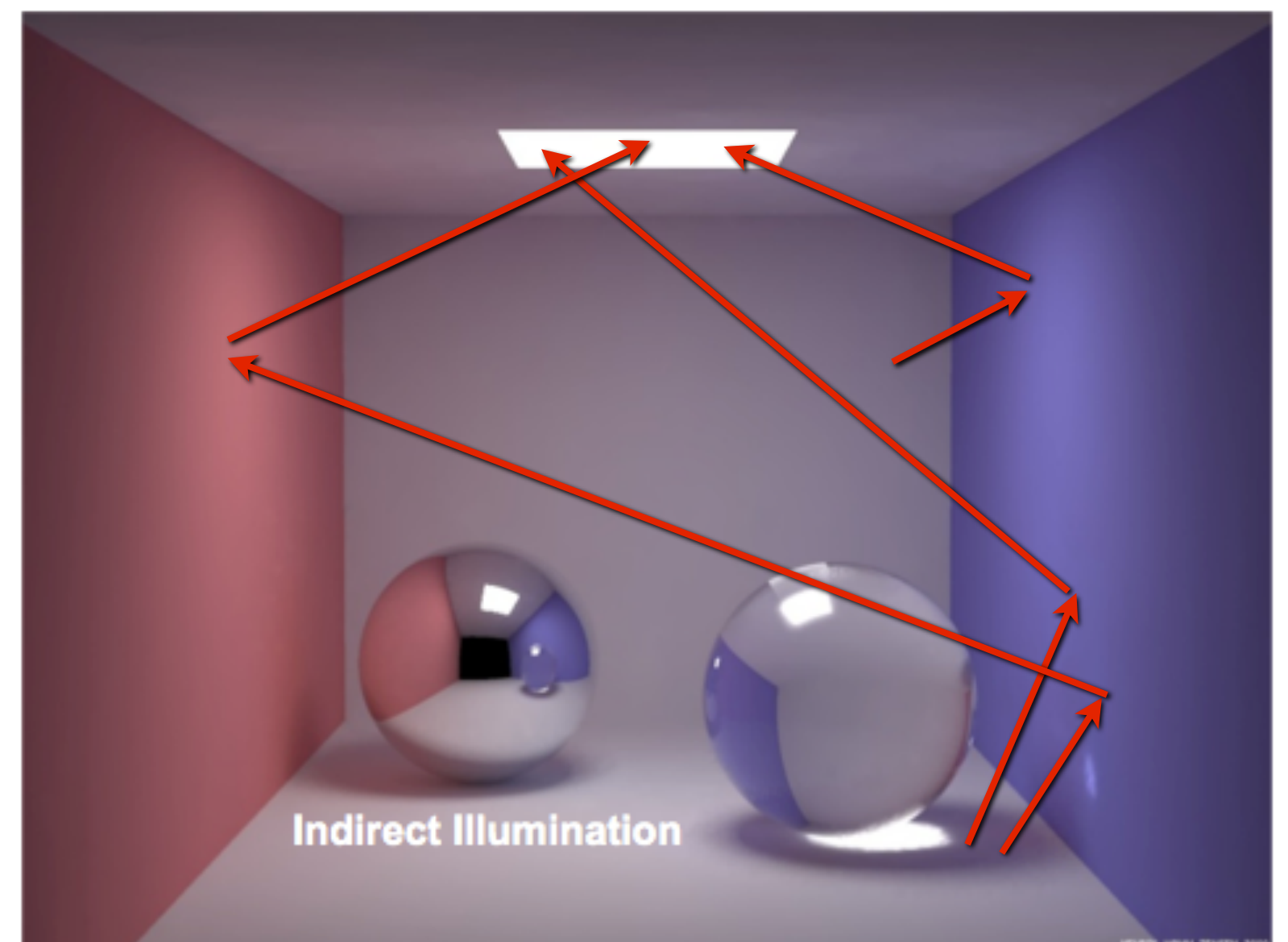
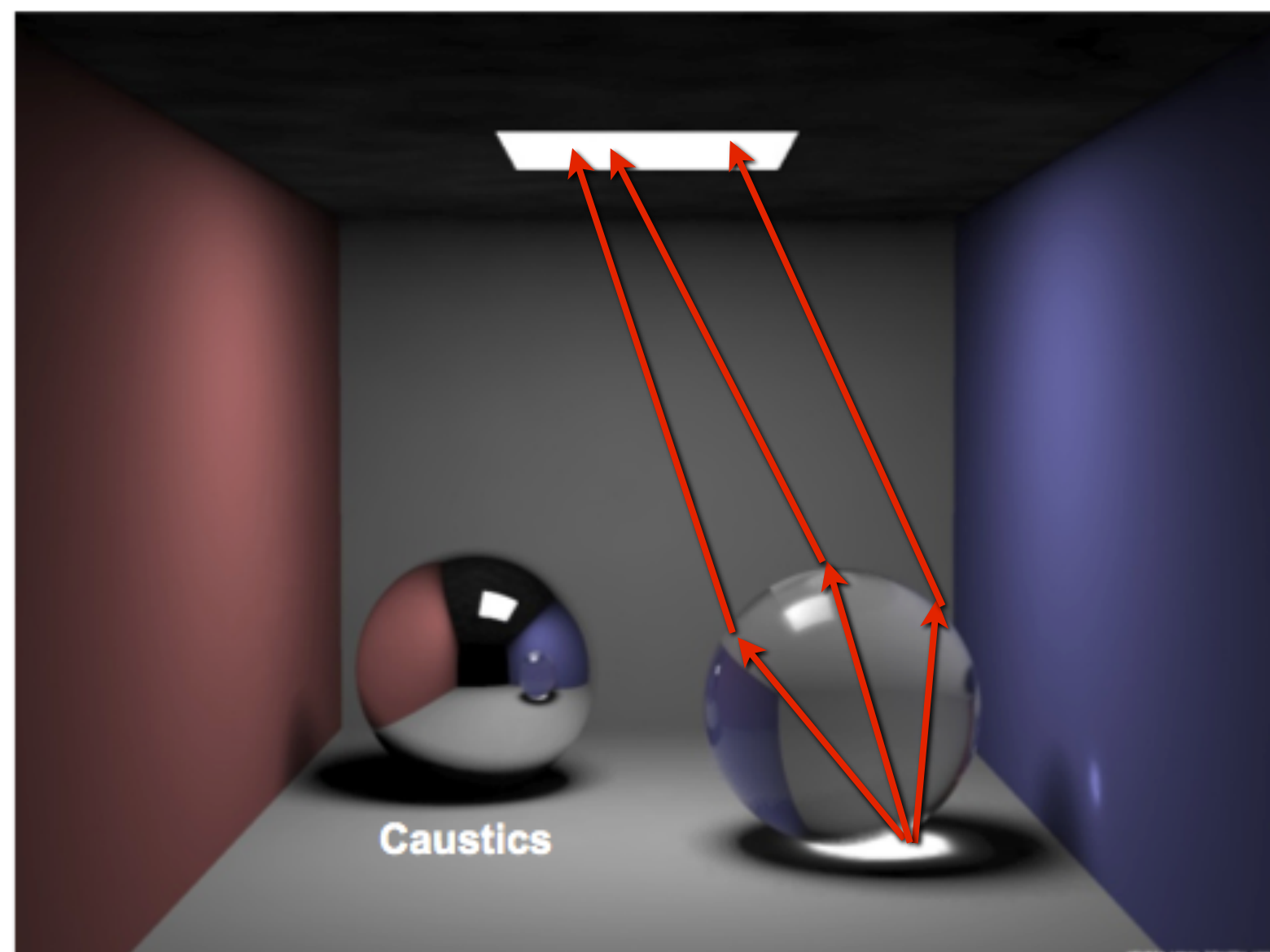
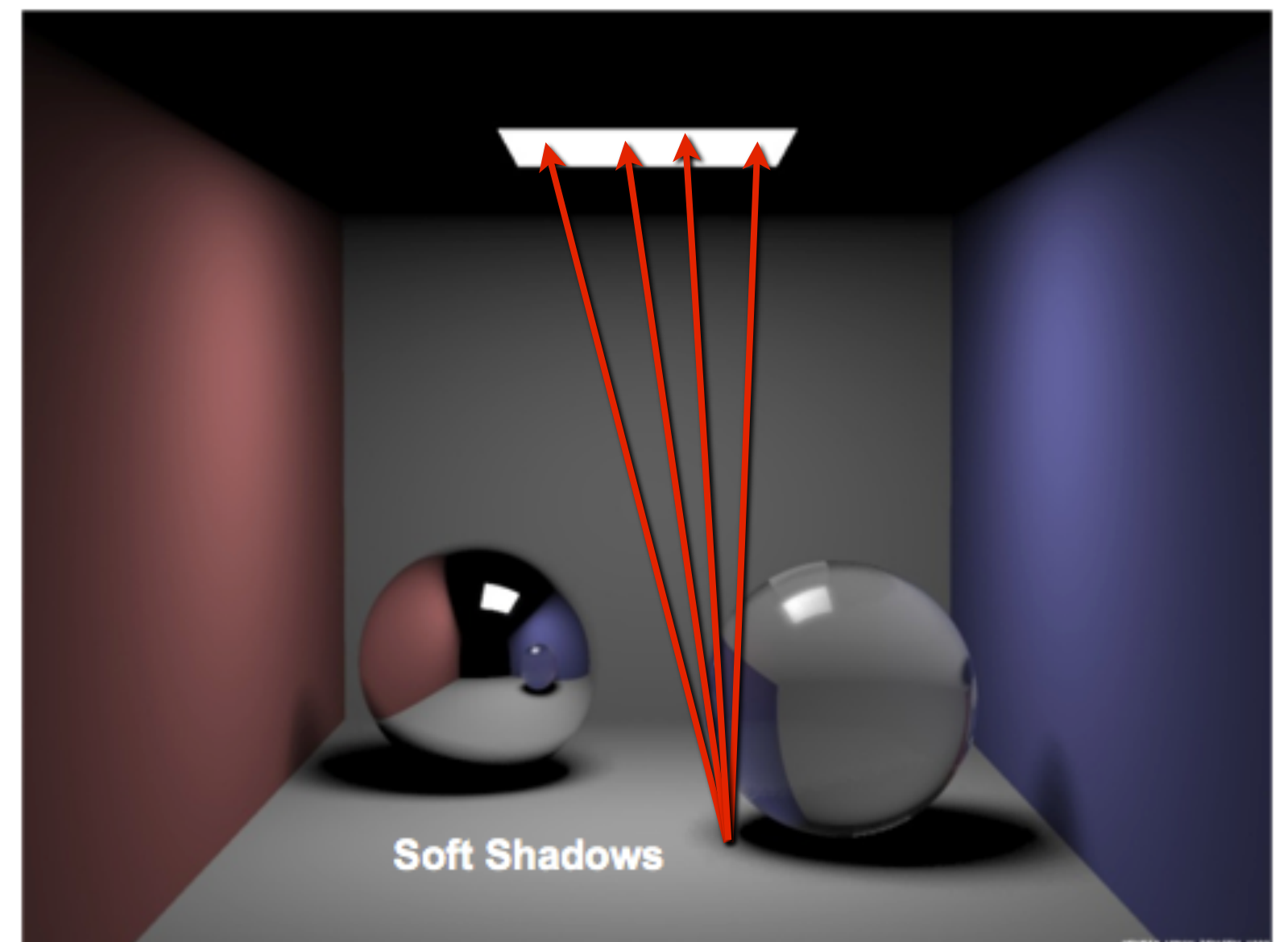
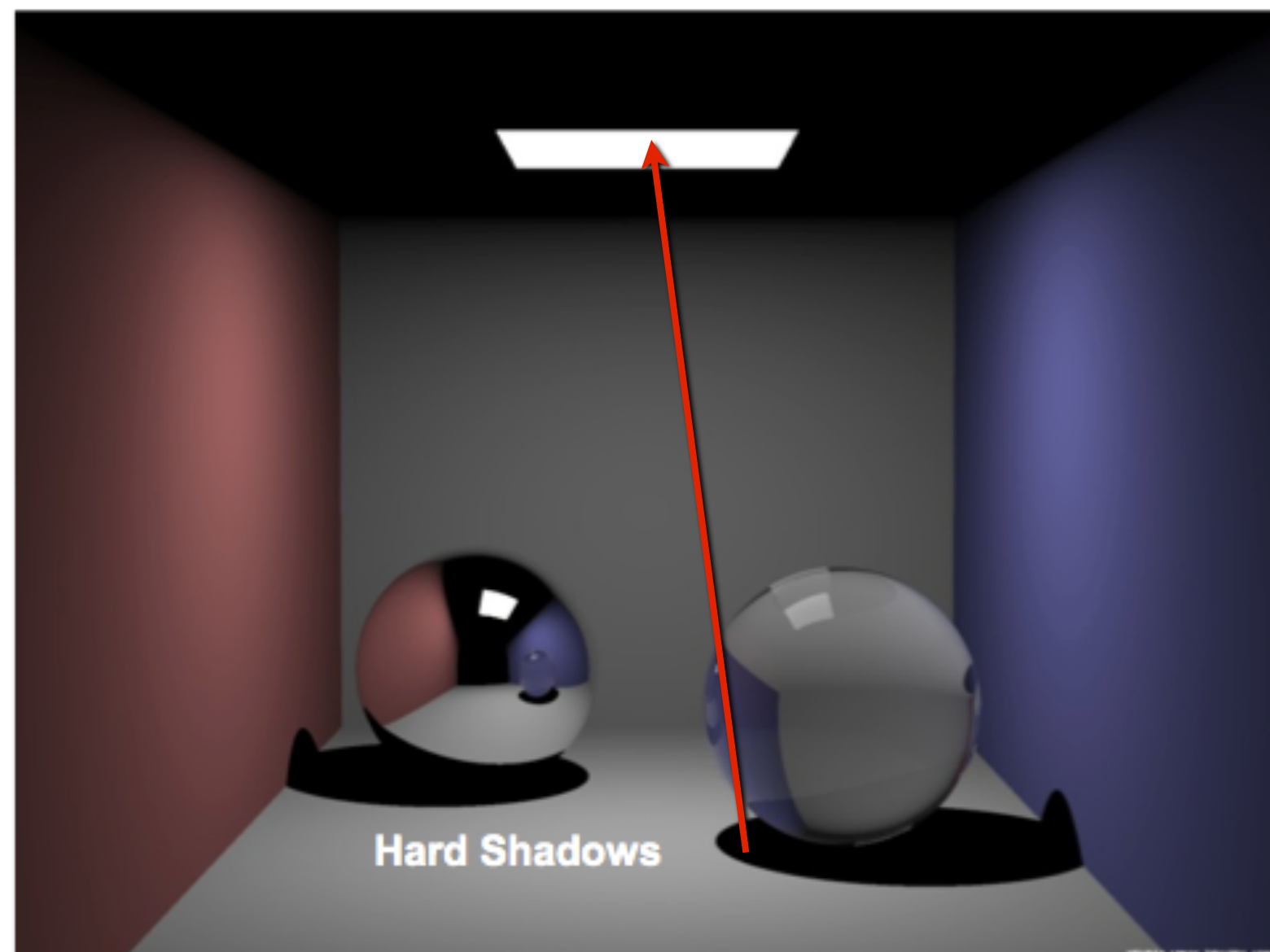


Recent push towards real-time ray tracing



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

Sampling light paths



Ray tracing primitive is used in many contexts

■ Camera rays (a.k.a., eye rays, primary rays)

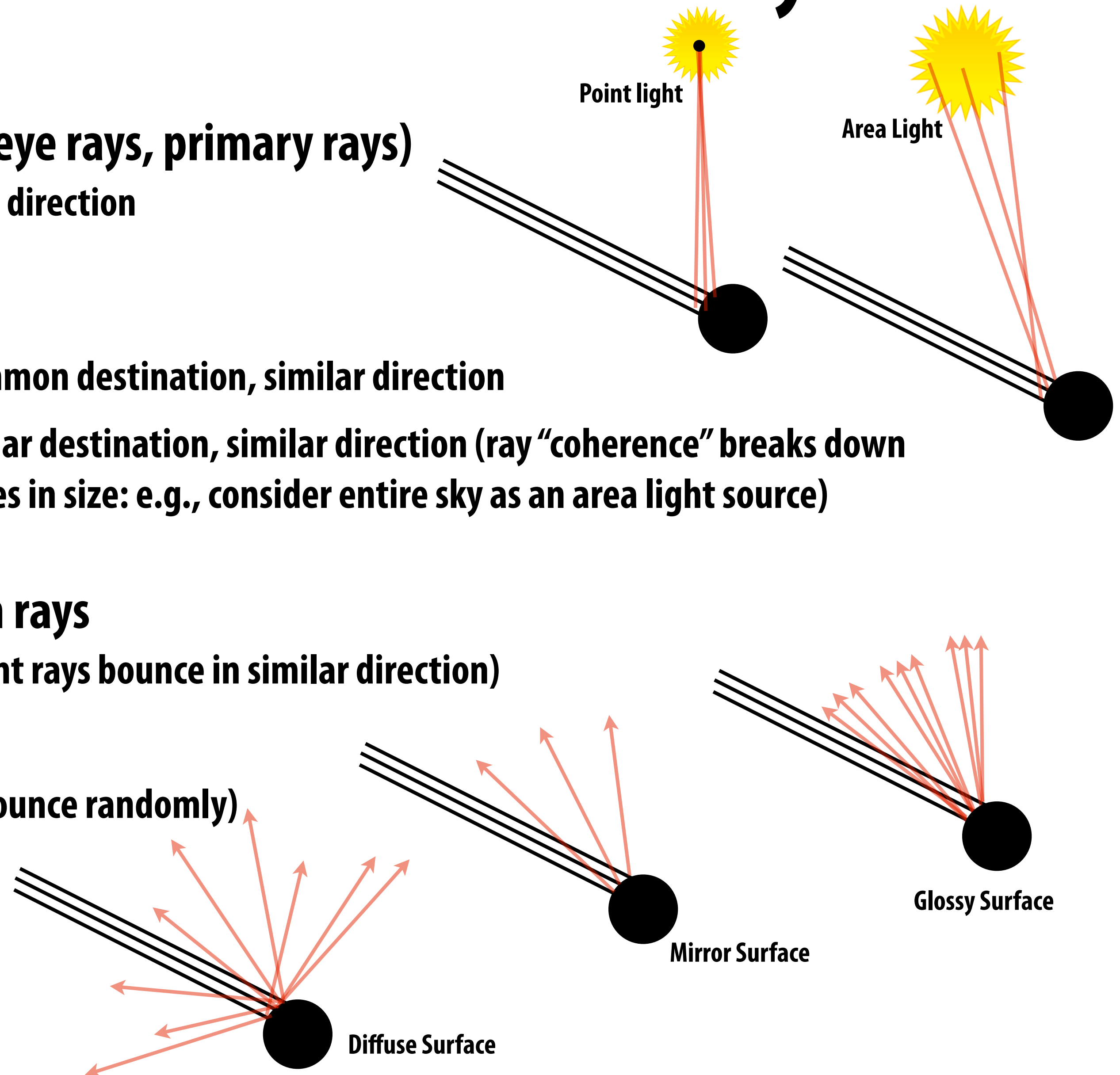
- Common origin, similar direction

■ Shadow rays

- Point light source: common destination, similar direction
- Area light source: similar destination, similar direction (ray “coherence” breaks down as light source increases in size: e.g., consider entire sky as an area light source)

■ Indirect illumination rays

- Mirror surface (coherent rays bounce in similar direction)
- Glossy surface
- Diffuse surface (rays bounce randomly)



Another way to think about rasterization

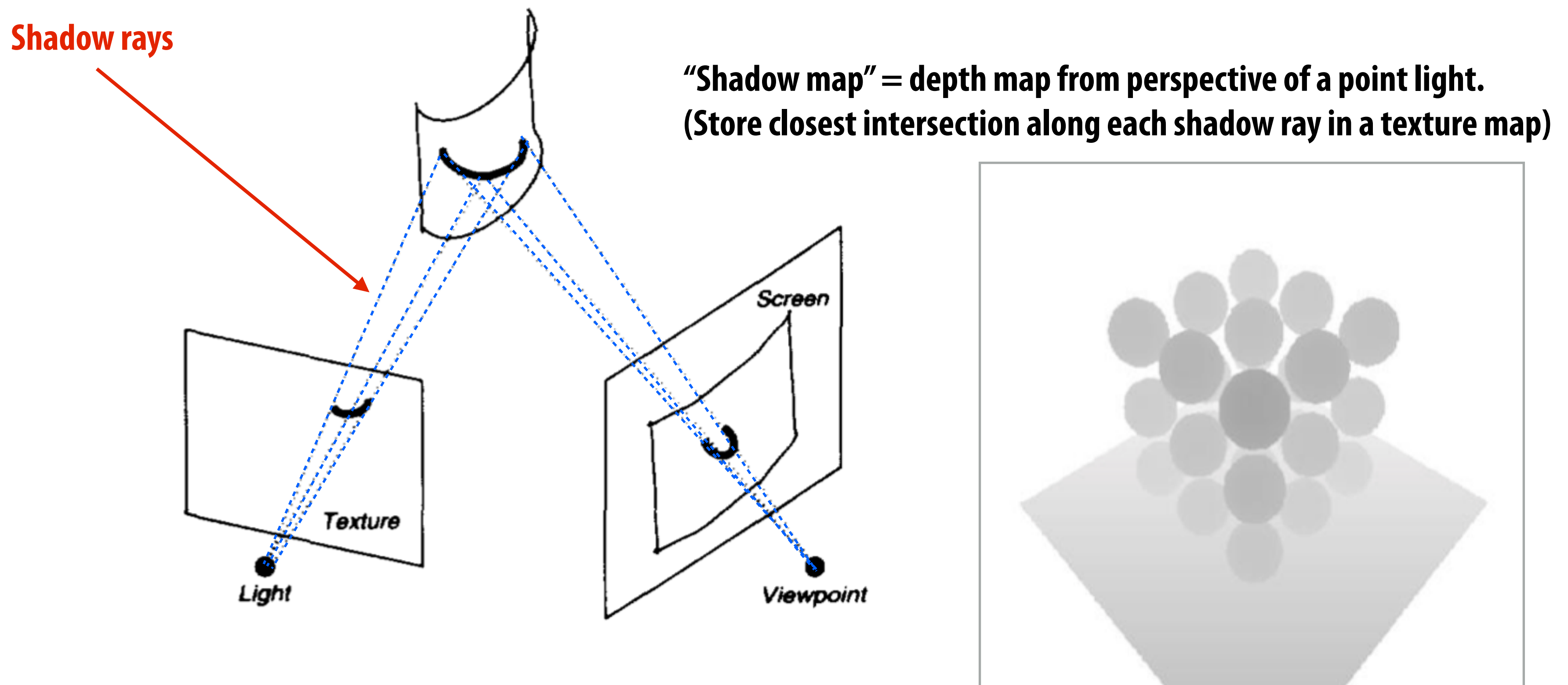
- **Rasterization is an optimized visibility algorithm for batches of rays with specific properties**
 - **Assumption 1: Rays have the same origin**
 - **Assumption 2: Rays are uniformly distributed (across image plane... not uniformly distributed in angle)**

1. Rays have same origin:

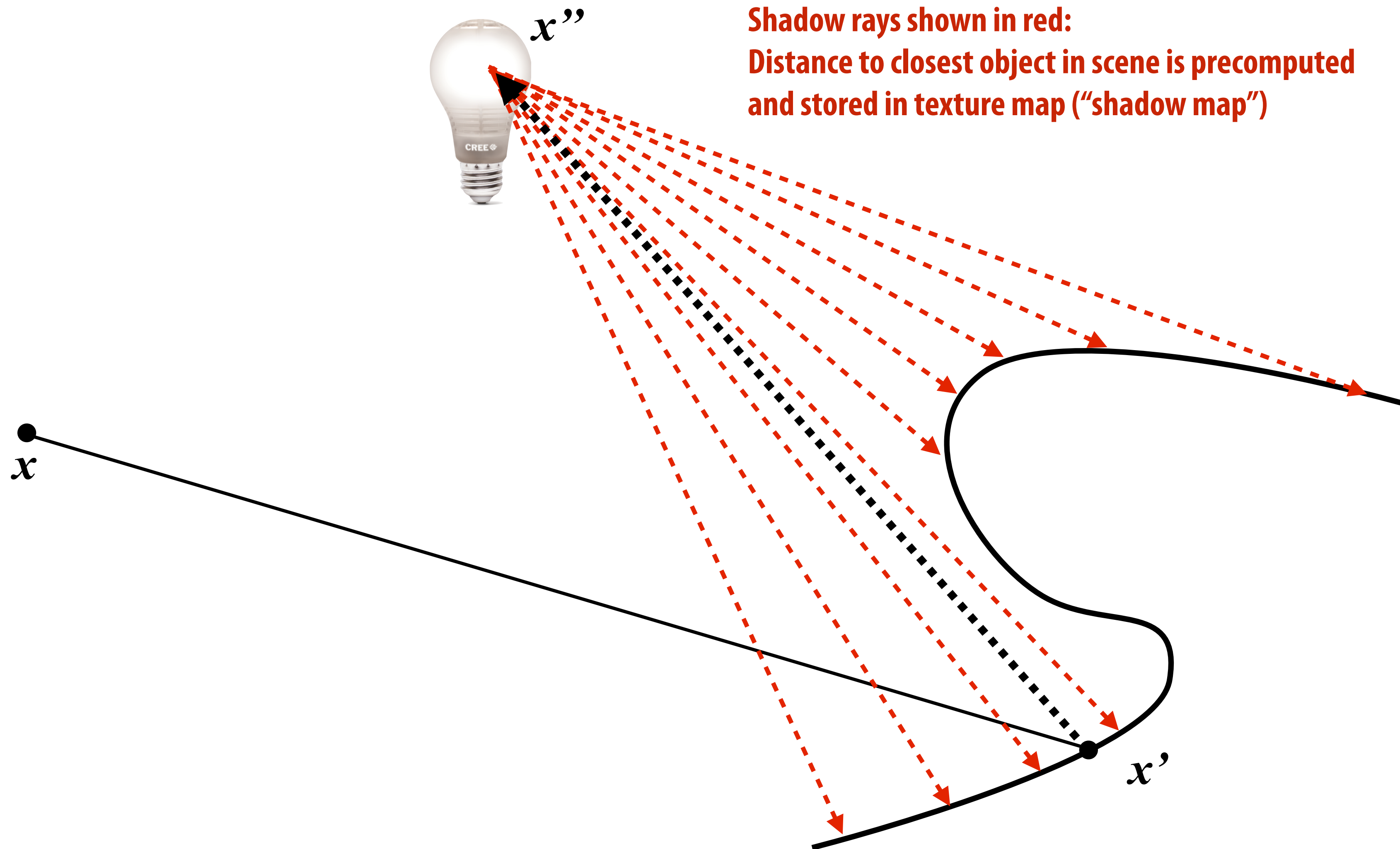
- **Optimization: project triangles to reduce ray-triangle intersection to 2D point-in-polygon tests**
- **Simplifies math (2D point-in-triangle test rather than 3D intersection)**
- **Allows use of fixed-point math (clipping establishes precision bounds)**

Shadow mapping: ray origin need not be the scene's camera position [Williams 78]

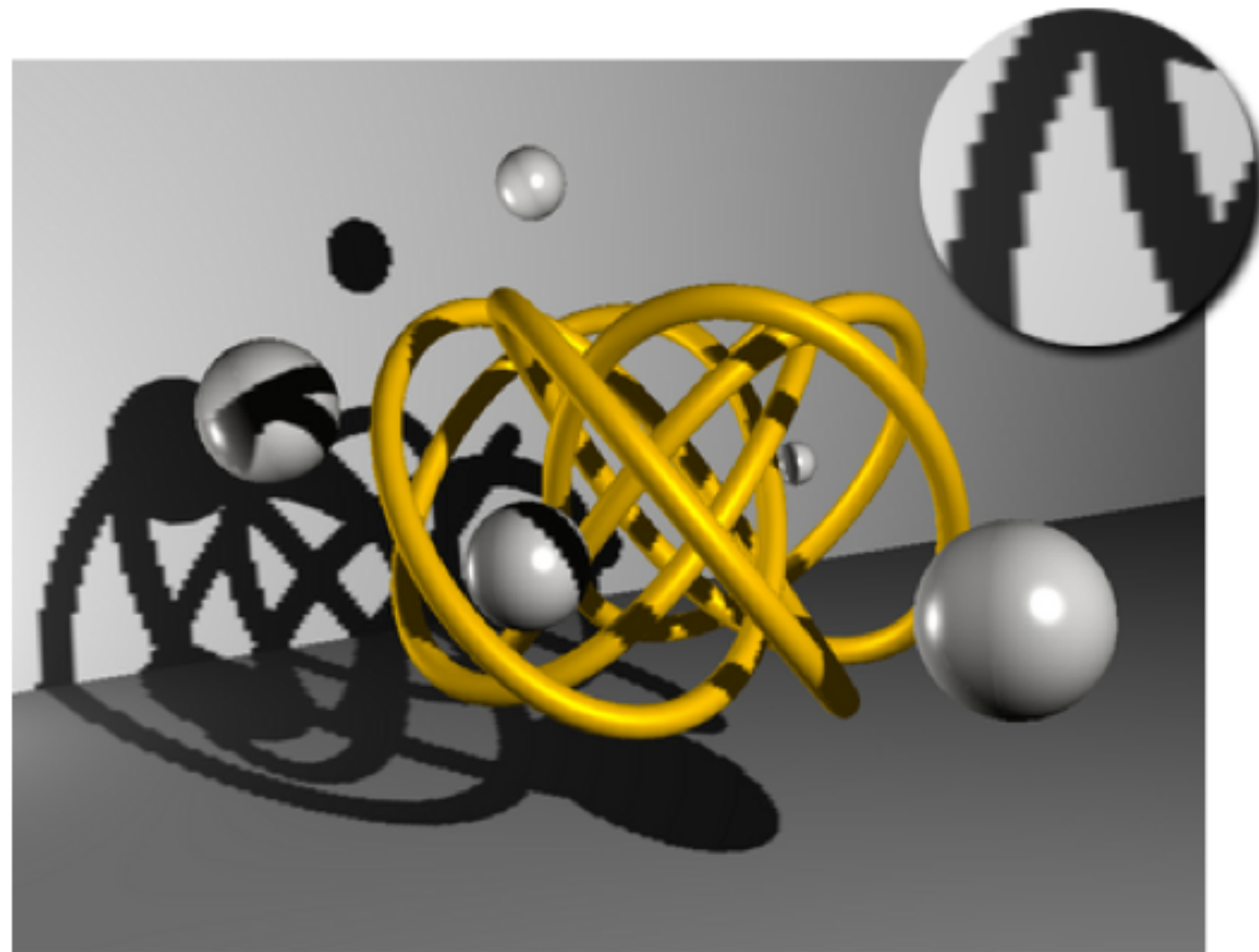
- Place ray origin at position of a point light source
- Render scene to compute depth to closest object to light along uniformly distributed “shadow rays” (answer stored in depth buffer)
- Store precomputed shadow ray intersection results in a texture



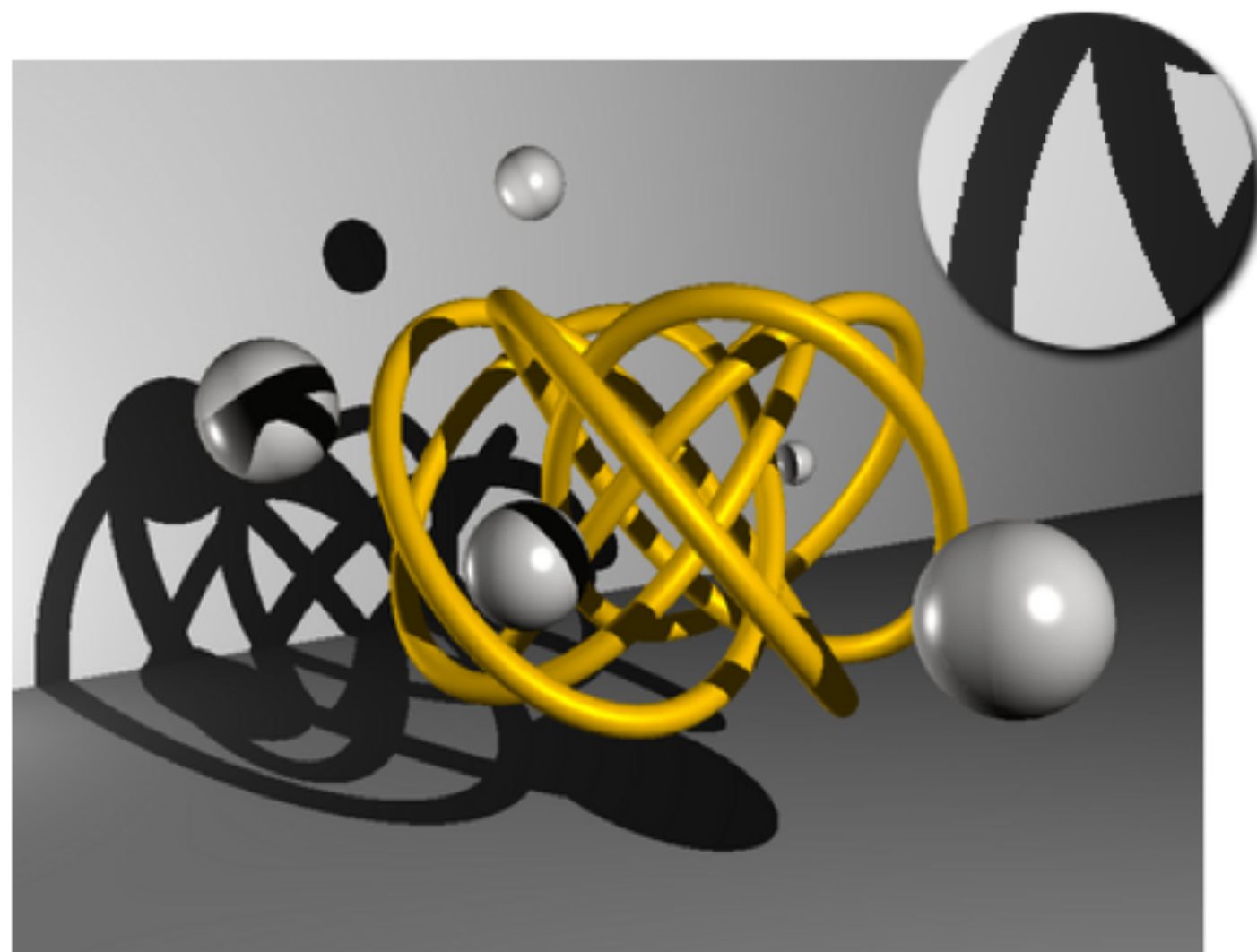
Result of shadow texture lookup approximates visibility result when shading fragment at x'



Shadow aliasing due to shadow map undersampling



Shadows computed using shadow map



**Correct hard shadows
(result from computing $v(x',x'')$ directly using ray tracing)**

Rasterization: ray origin need not be camera position

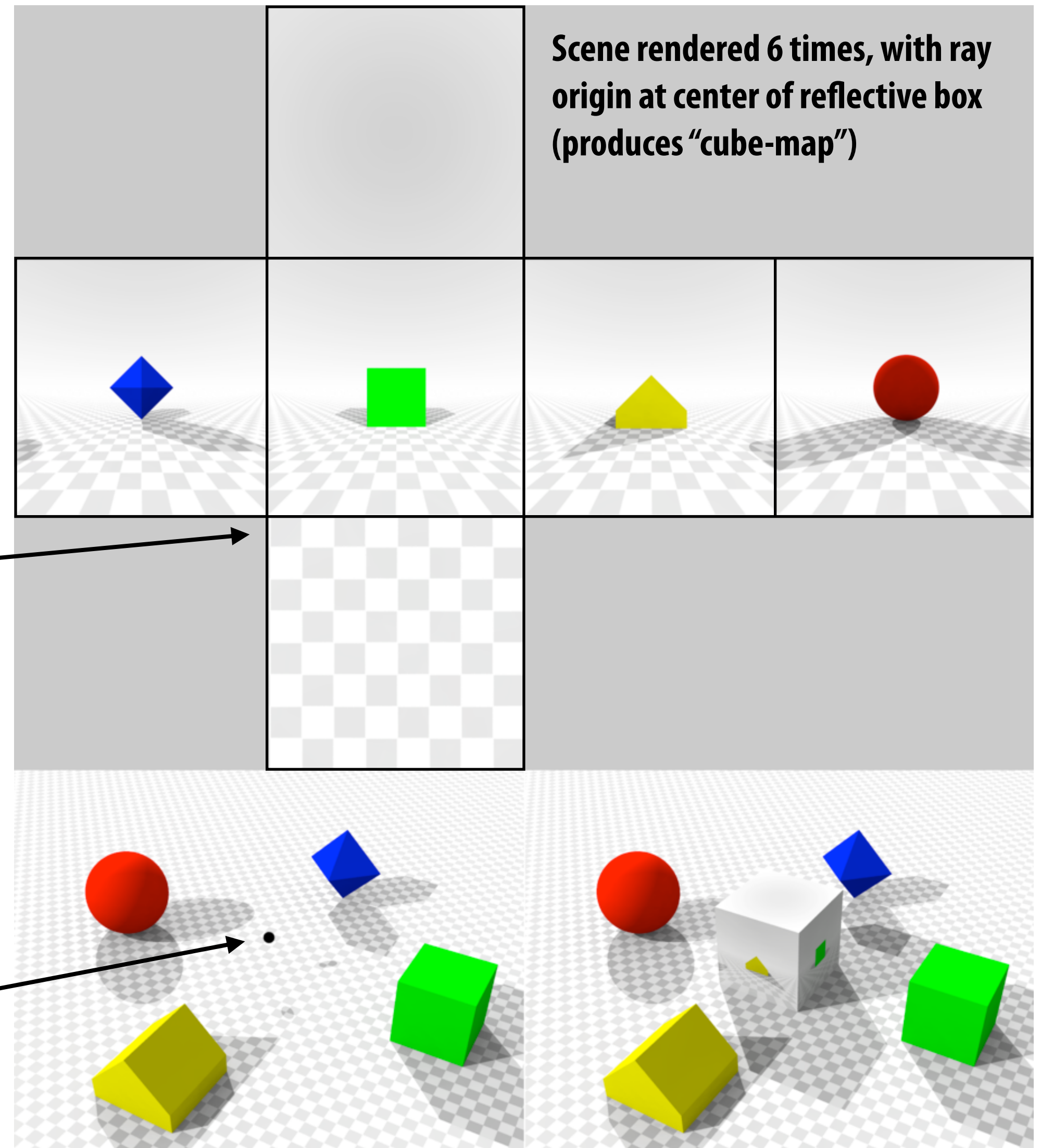
Environment mapping:
place ray origin at reflective object

Yields approximation to true reflection results. Why?

Cube map: stores results of approximate mirror reflection rays

(Question: how can a glossy surface be rendered using the cube-map)

Center of projection



Summary: rasterization as a visibility algorithm

- Rasterization is an optimized visibility algorithm for specific batches of rays

- Assumption 1: Rays have the same origin
- Assumption 2: Rays are uniformly distributed on image plane

1. Same origin: projection of triangles reduces ray-triangle intersection to cheap/efficient 2D point-in-polygon test

- GPUs have specialized fixed-function hardware for this computation. It's called the rasterizer.

2. Uniform sample distribution: given polygon, it is easy (a.k.a. efficient) to “find” samples covered by polygon

- Frame buffer: constant time sample lookup, update, edit
- Sample search leverages 2D screen coherence
 - Amortize operations over tile of samples (can think of tiled frame buffer as a two-level hierarchy on samples)
- No need for complex acceleration structures to accelerate a search over samples (a basic tiled rasterizer requires no acceleration structures for coverage testing) *

* One could make an argument that hi-Z uses an acceleration structure (precomputed min/max Z)

Rasterization: performance

- **Stream over scene geometry (regular/predictable data access), but arbitrarily access frame-buffer data (per-sample data)**
 - Unpredictable access to sample data is manageable since it's a regular, fixed-size data structure (color/Z-buffer caching, compression, etc.)
- **Z-buffered occlusion**
 - Fixed number of samples (determined by screen resolution, sampling rate)
 - Known sample data structure
 - Implication: efficient to find samples covered by polygon (highly optimized fixed-function implementations of both coverage computation and frame-buffer update)
- **Scales to high scene complexity**
 - Stream over geometry: so required memory footprint in graphics pipeline is independent of the number of triangles in the scene

Why real-time ray tracing?

Potential real-time ray tracing



VR may demand more flexible control over what pixels are drawn. (e.g., row-based display rather than frame-based, higher resolution where eye is looking, correct for distortion of optics)

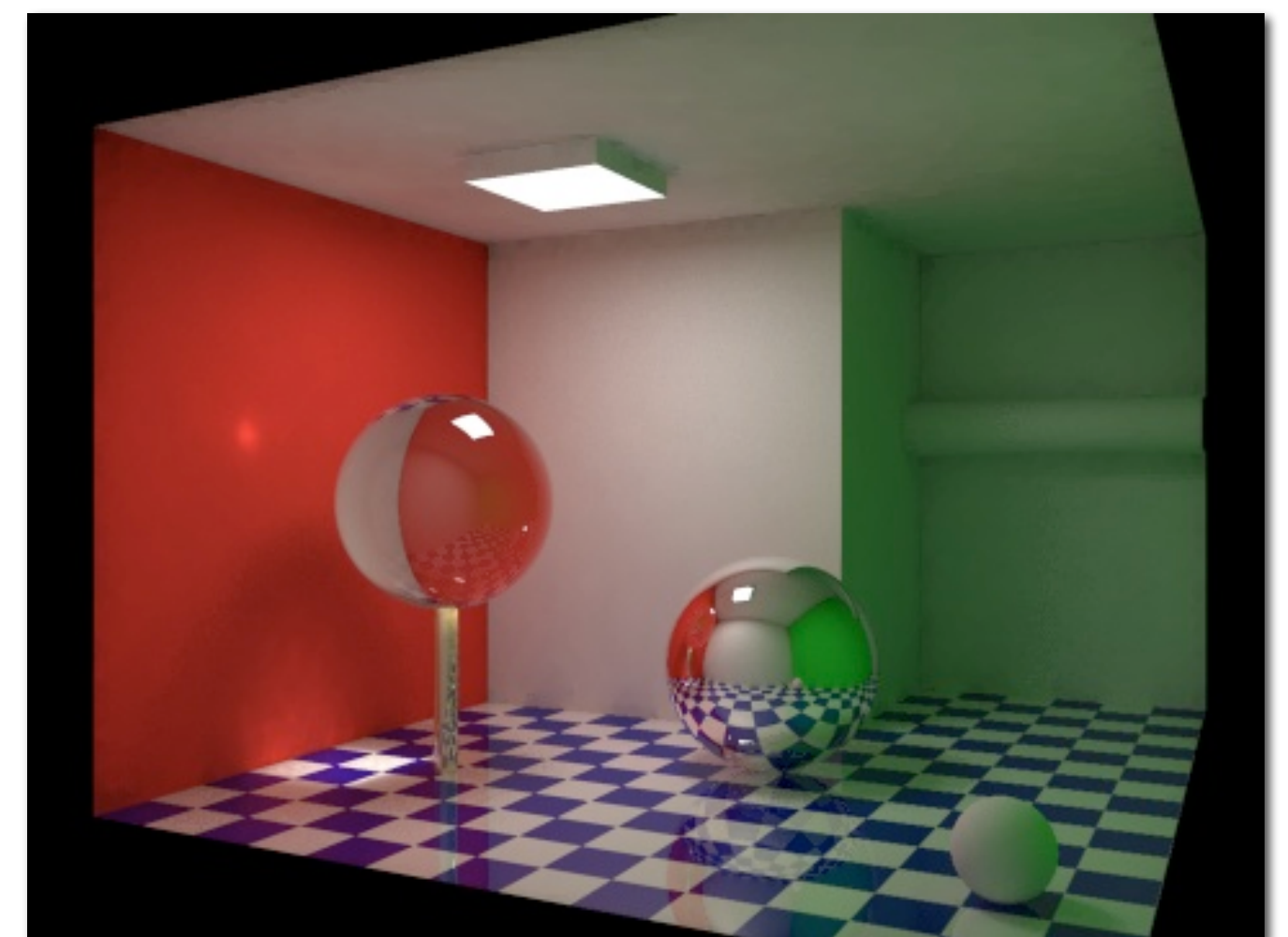
Reduce content creation and game engine development time: single general solution rather than a specialized technique for each effect.

Other indirect illumination effects?
(unclear if ray tracing is best real-time solution for low frequency effects)



Many shadowed lights (pain to manage hundreds of shadow maps)

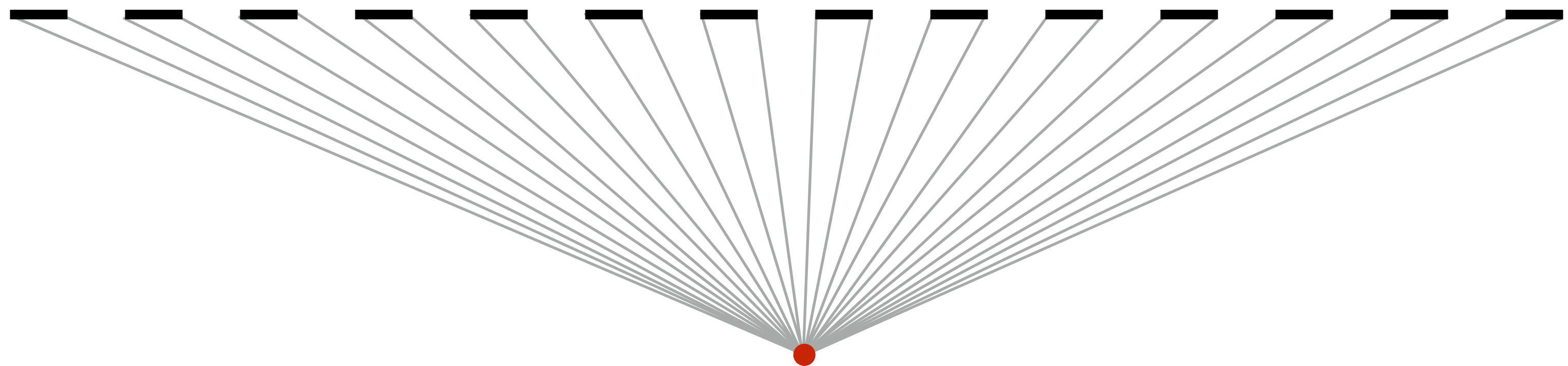
Accurate reflections from curved surfaces



Challenge: rendering via non-planar projection

Recall: rasterization-based graphics is based on perspective projection to plane

- Reasonable for modest FOV, but distorts image under high FOV
- VR rendering spans wide FOV



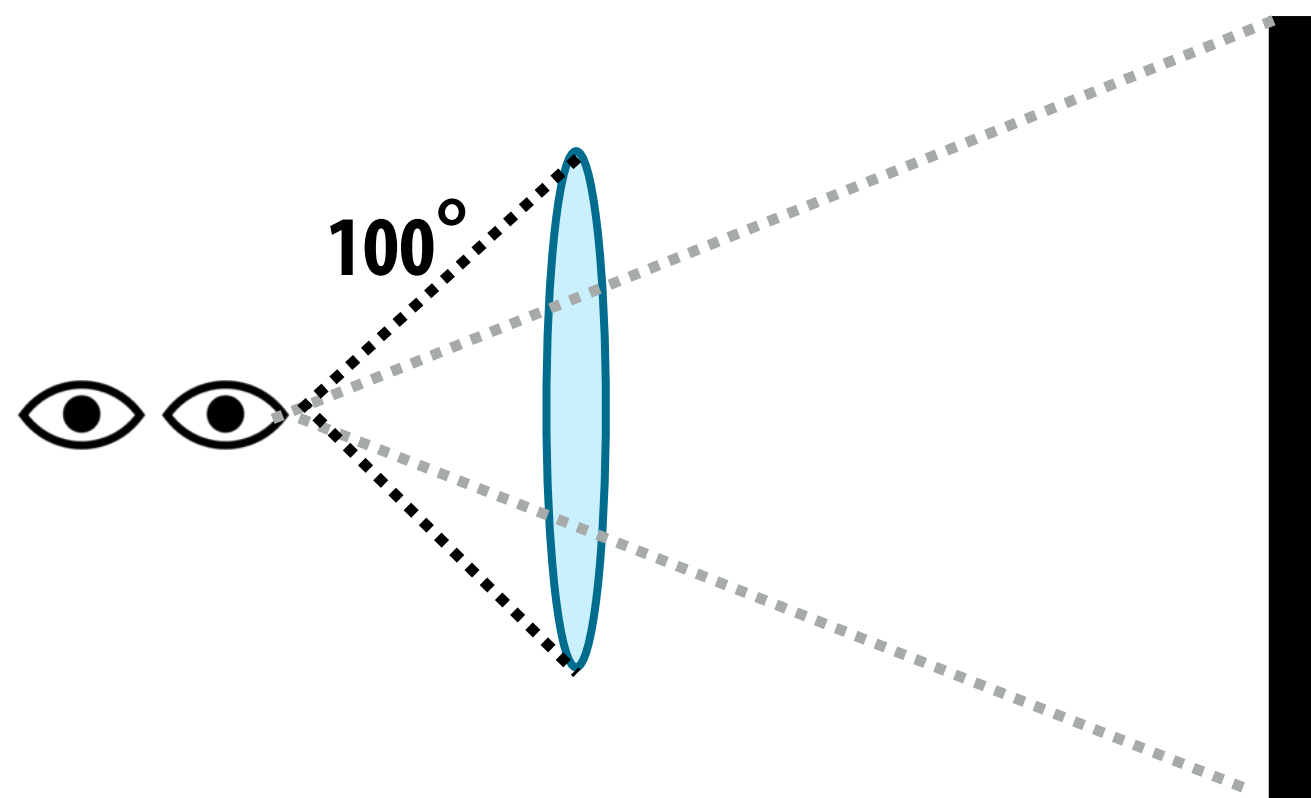
**Pixels span larger angle in center of image
(lowest angular resolution in center)**

Future investigations may consider: curved displays, ray casting to achieve uniform angular resolution, rendering with piecewise linear projection plane (different plane per tile of screen)

Oculus Rift DK2 headset



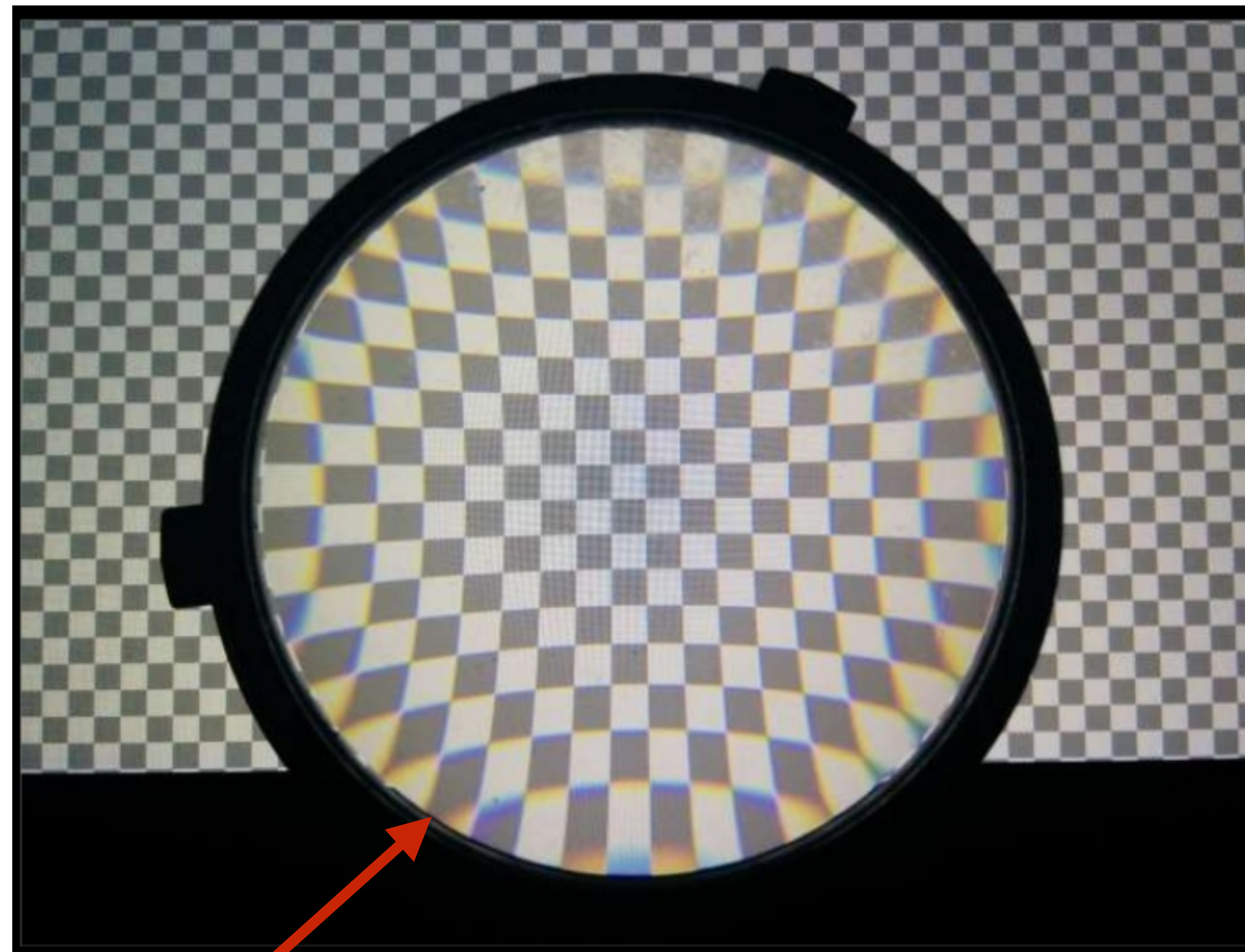
Requirement: wide field of view



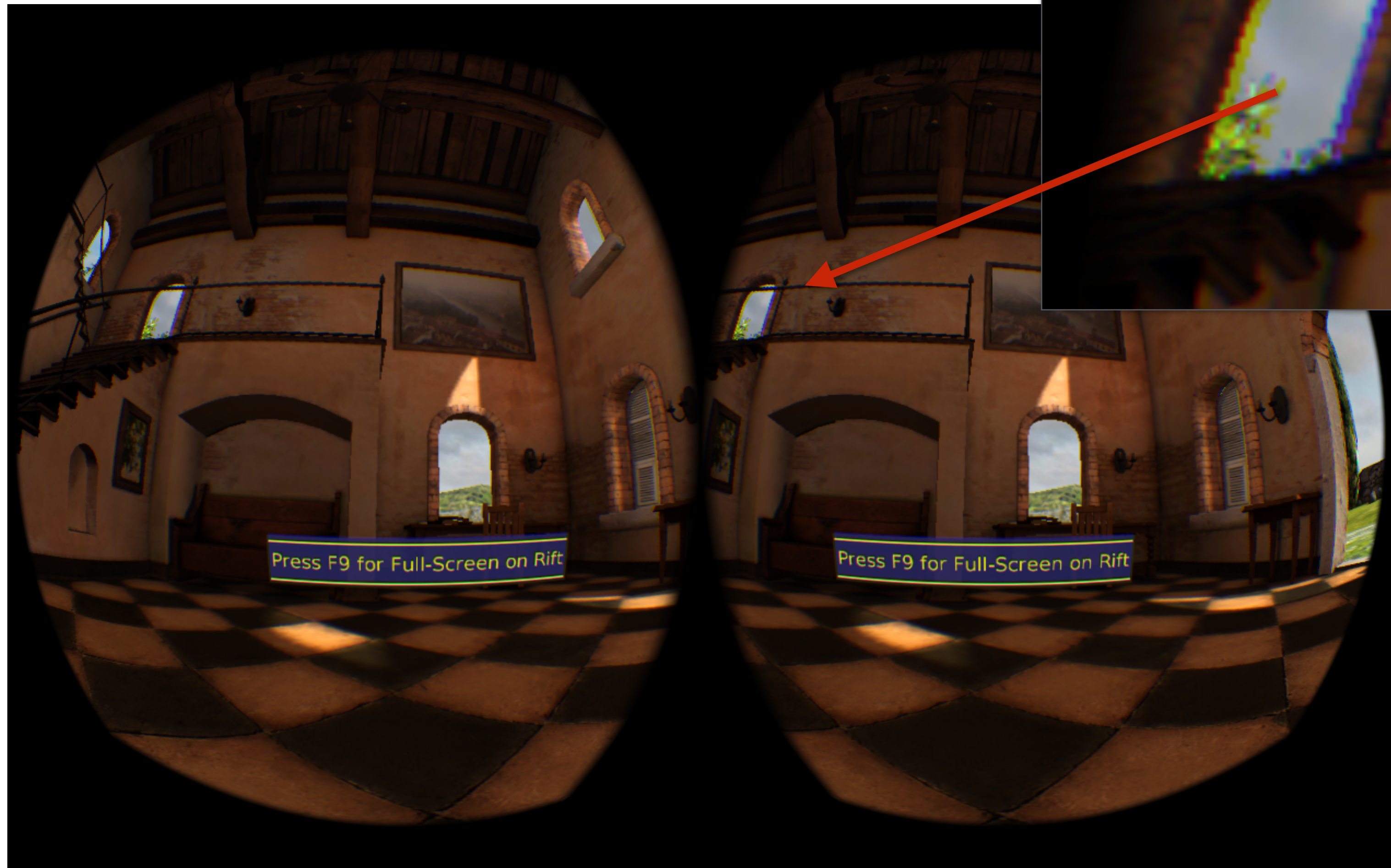
Lens introduces distortion

- Pincushion distortion
- Chromatic aberration
(different wavelengths of light refract by different amount)

View of checkerboard through Oculus Rift lens



Rendered output must compensate for distortion of lens in front of display



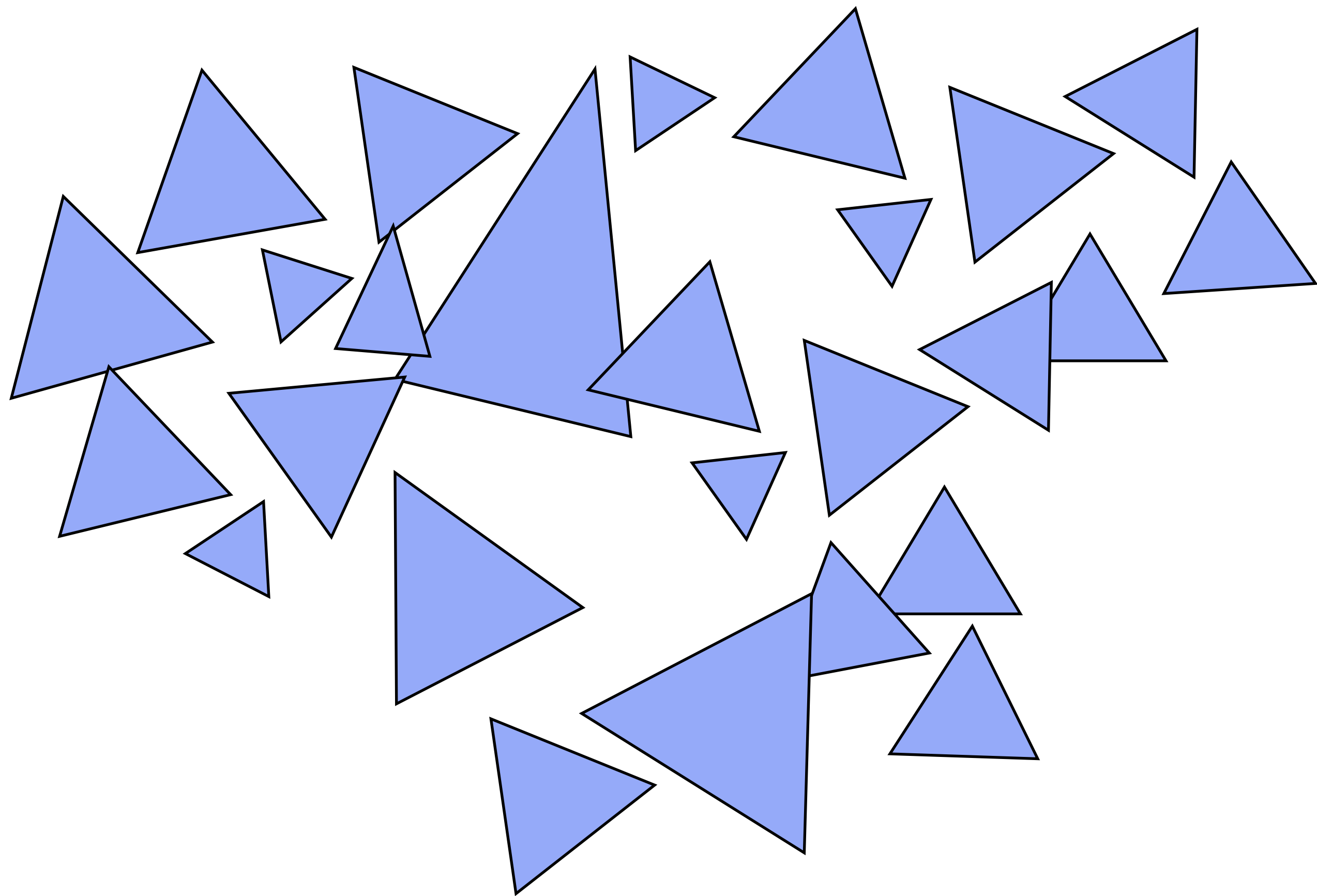
Step 1: render scene using traditional graphics pipeline at full resolution for each eye

Step 2: warp images and composite into frame rendering is viewed correctly after lens distortion

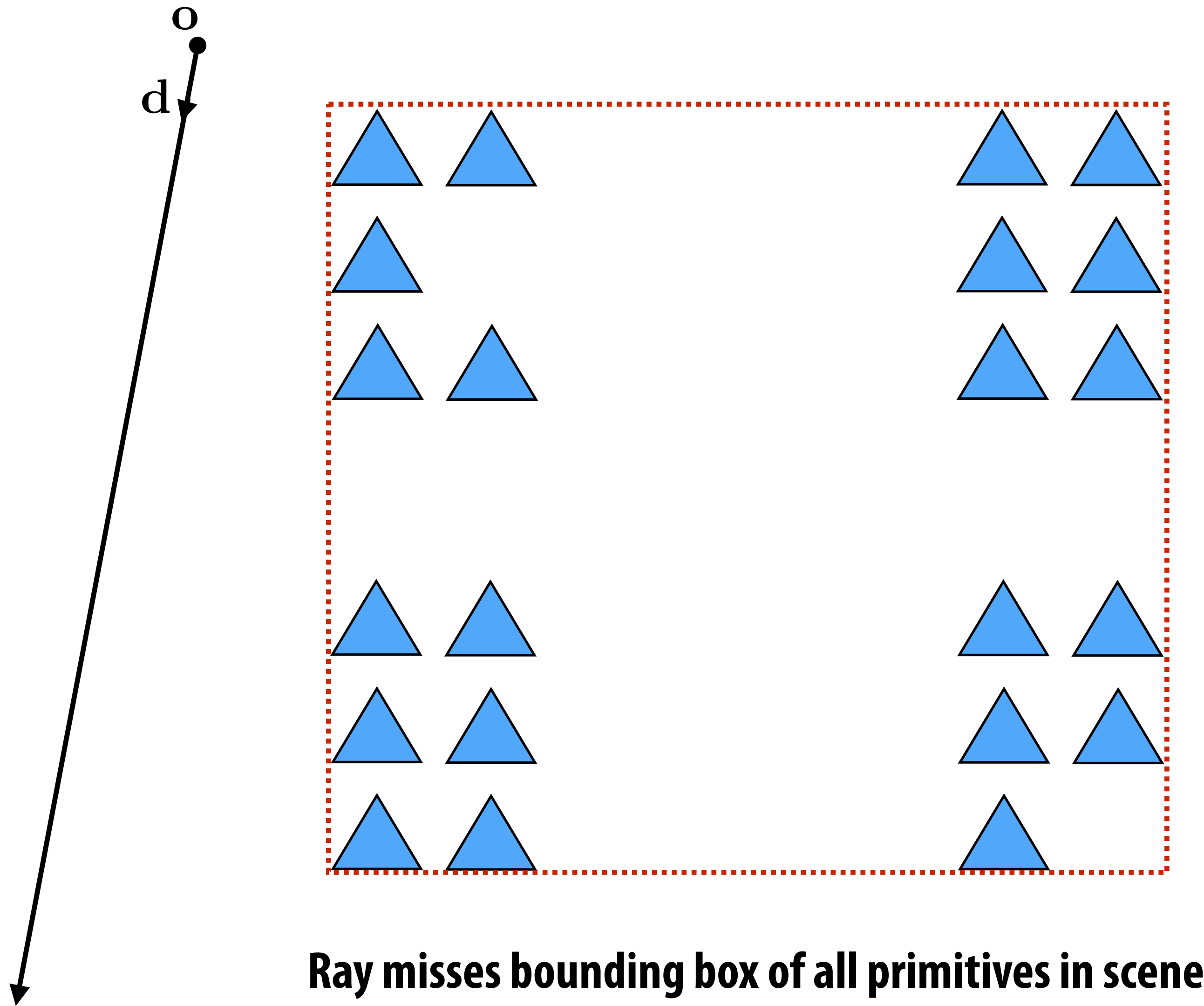
(Can apply unique distortion to R, G, B to approximate correction for chromatic aberration)

Efficient ray traversal algorithms

How do we organize scene primitives to enable fast ray-scene intersection queries?

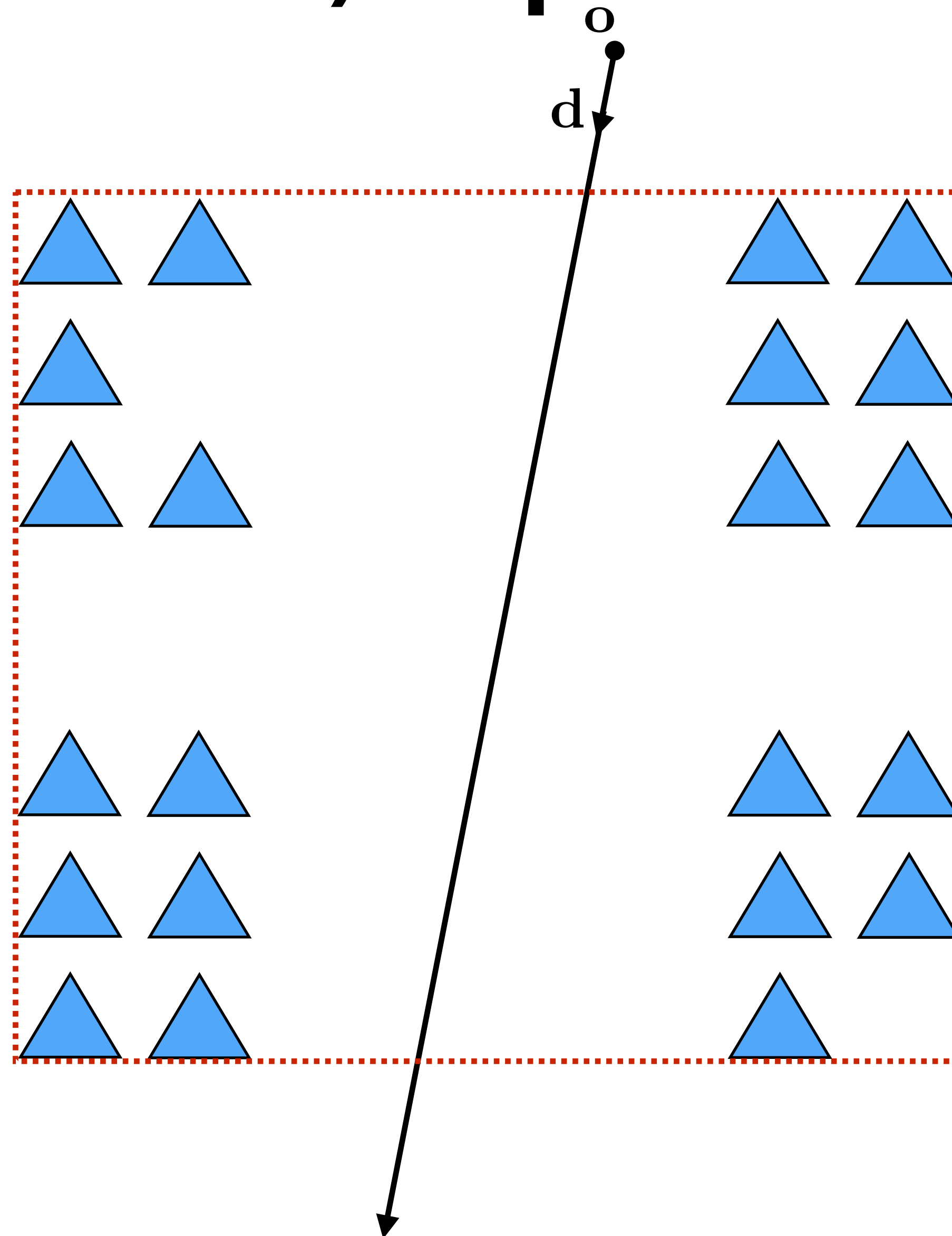


Simple case



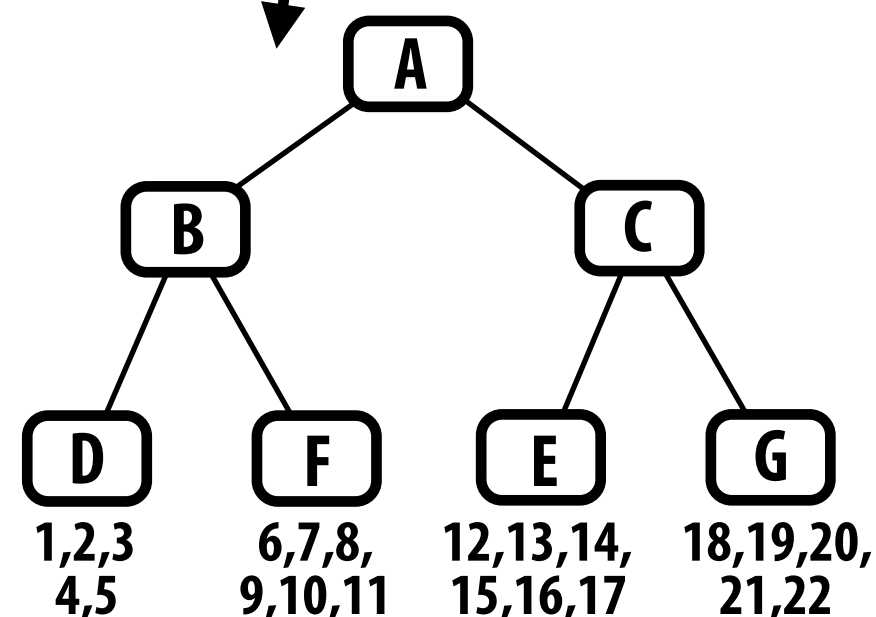
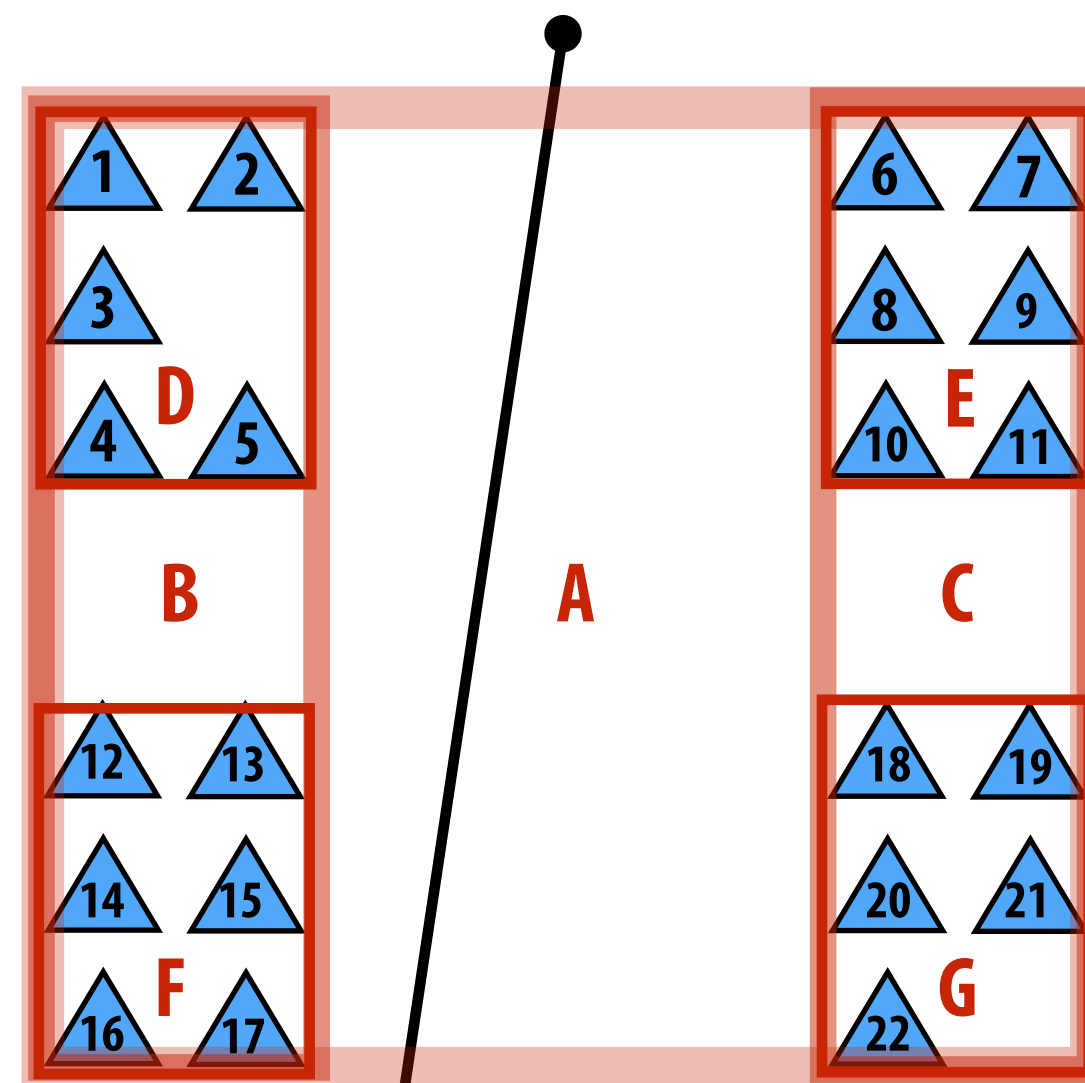
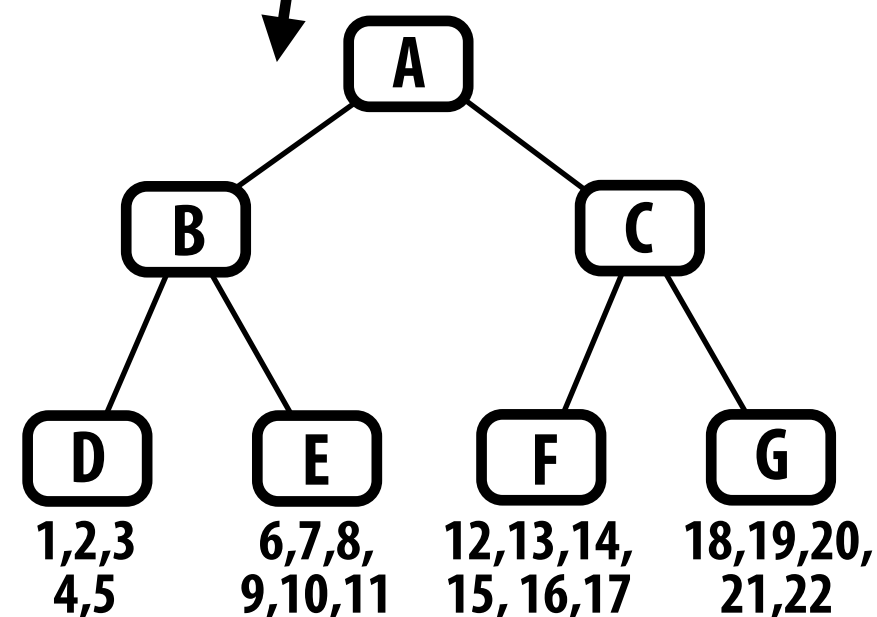
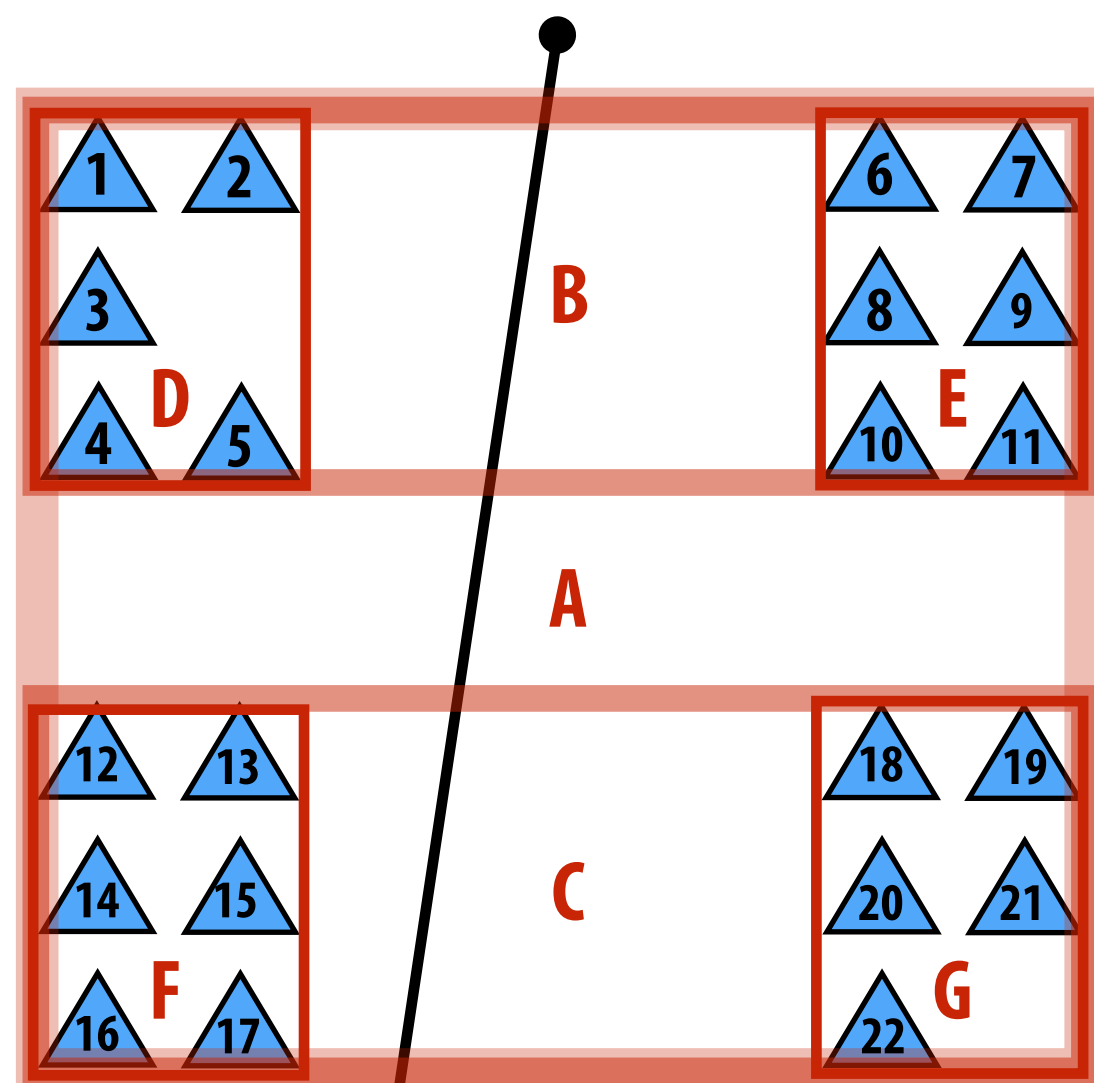
Ray misses bounding box of all primitives in scene
 $O(1)$ cost: requires 1 ray-box test

Another (should be) simple case



Bounding volume hierarchy (BVH)

- Interior nodes:
 - Represents subset of primitives in scene
 - Stores aggregate bounding box for all primitives in subtree
- Leaf nodes:
 - Contain list of primitives

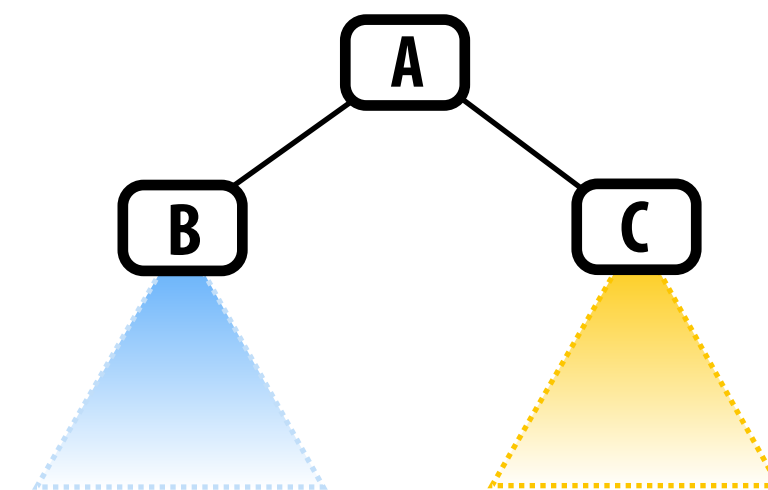
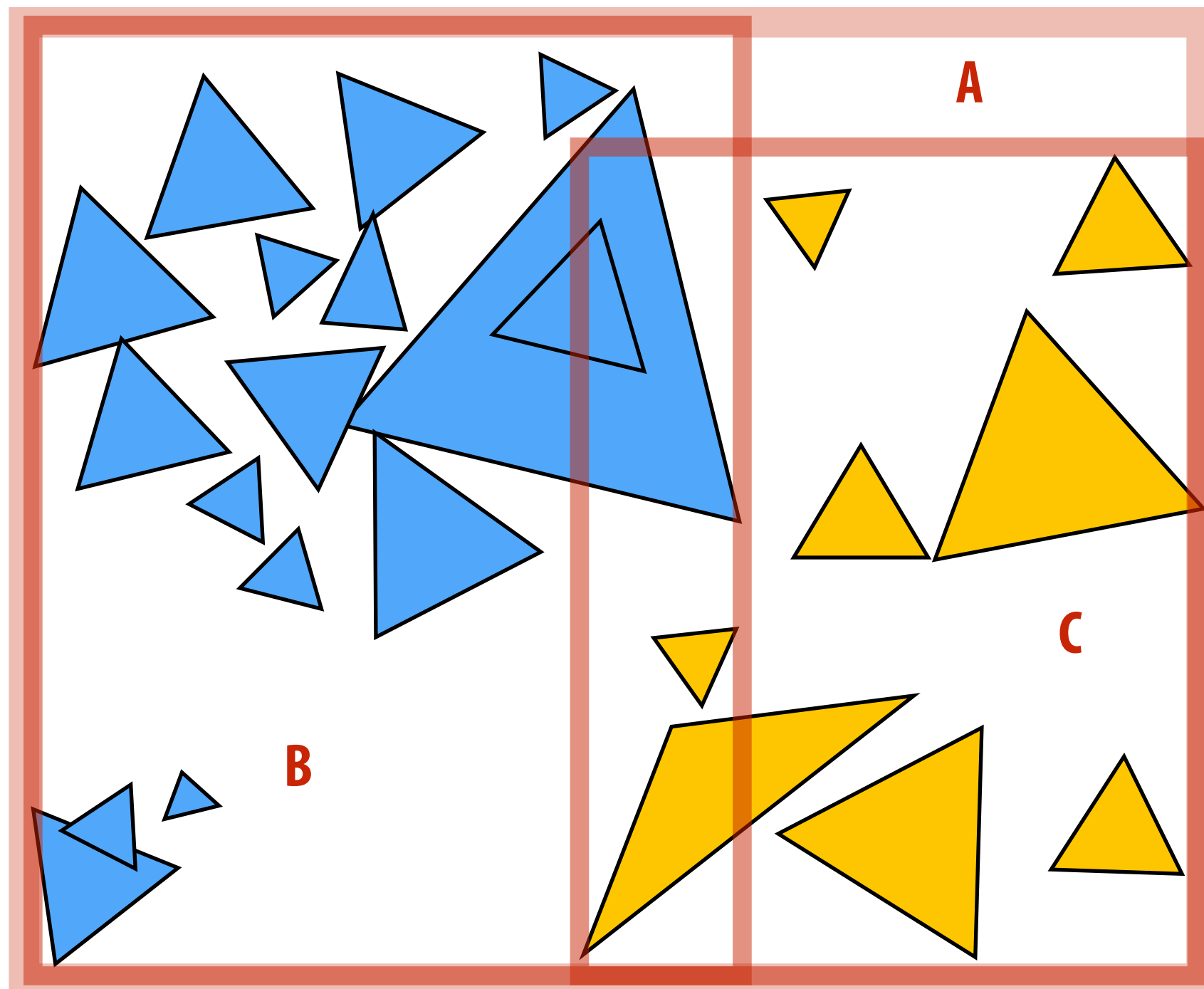


Left: two different BVH organizations of the same scene containing 22 primitives.

Is one BVH better than the other?

Another BVH example

- **BVH partitions each node's primitives into disjoint sets**
 - **Note: The sets can still be overlapping in space (below: child bounding boxes may overlap in space)**



Ray-scene intersection using a BVH

```
struct BVHNode {
    bool leaf;
    BBox bbox;
    BVHNode* child1;
    BVHNode* child2;
    Primitive* primList;
};
```

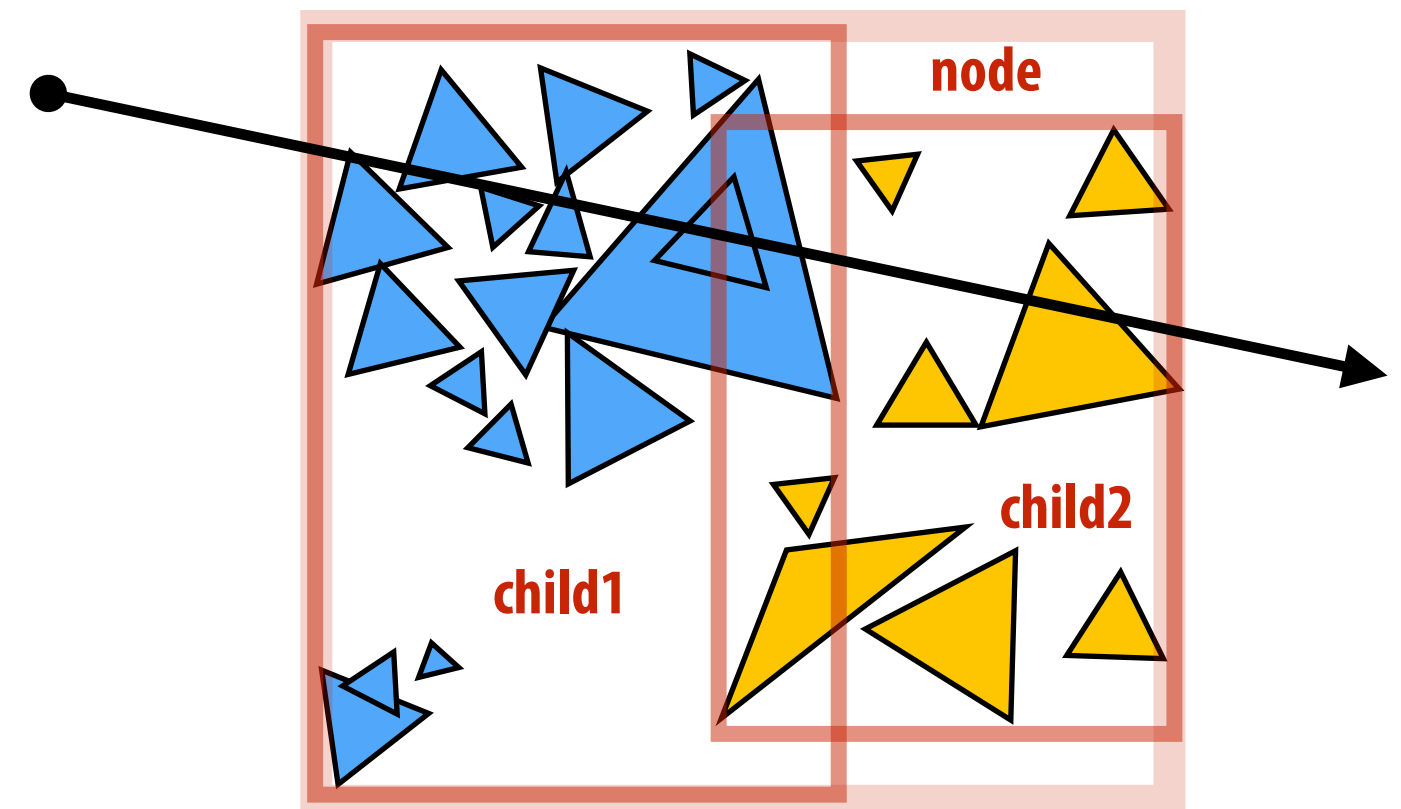
```
struct ClosestHitInfo {
    Primitive prim;
    float min_t;
};
```

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

```
    if (!intersect(ray, node->bbox) || (closest point on box is farther than closest.min_t))
        return;
```

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

```
    } else {
        find_closest_hit(ray, node->child1, closest);
        find_closest_hit(ray, node->child2, closest);
    }
}
```



How could this occur?

Improvement: “front-to-back” traversal

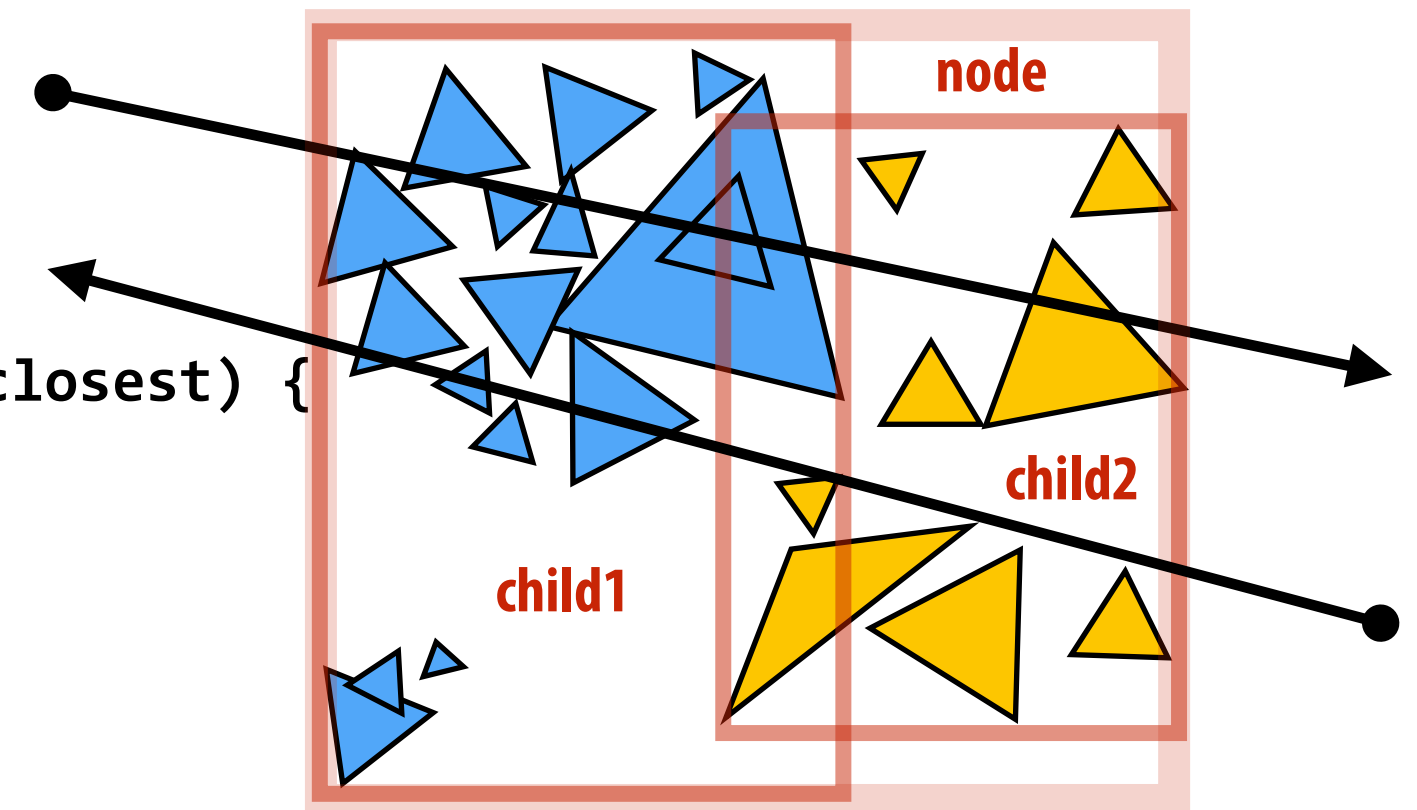
Invariant: only call `find_closest_hit()` if ray intersects bbox of node.

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

```
    if (node->leaf) {
        for (each primitive p in node->primList) {
            (hit, t) = intersect(ray, p);
            if (hit && t < closest.min_t) {
                closest.prim = p;
                closest.min_t = t;
            }
        }
    } else {
        (hit1, min_t1) = intersect(ray, node->child1->bbox);
        (hit2, min_t2) = intersect(ray, node->child2->bbox);

        NVHNode* first = (min_t1 <= min_t2) ? child1 : child2;
        NVHNode* second = (min_t1 <= min_t2) ? child2 : child1;

        find_closest_hit(ray, first, closest);
        if (second child's min_t is closer than closest.min_t)
            find_closest_hit(ray, second, closest);
    }
}
```

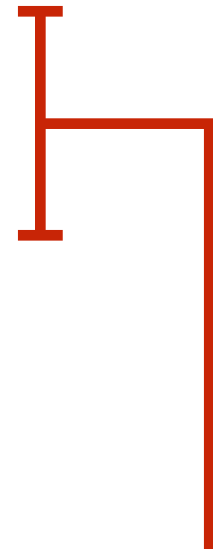


“Front to back” traversal. Traverse to closest child node first. Why?

Another type of query: any hit

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

```
bool find_any_hit(Ray* ray, BVHNode* node) {  
  
    if (!intersect(ray, node->bbox))  
        return false;  
  
    if (node->leaf) {  
        for (each primitive p in node->primList) {  
            (hit, t) = intersect(ray, p);  
            if (hit)  
                return true;  
        }  
    } else {  
        return ( find_closest_hit(ray, node->child1, closest) ||  
                find_closest_hit(ray, node->child2, closest) );  
    }  
}
```



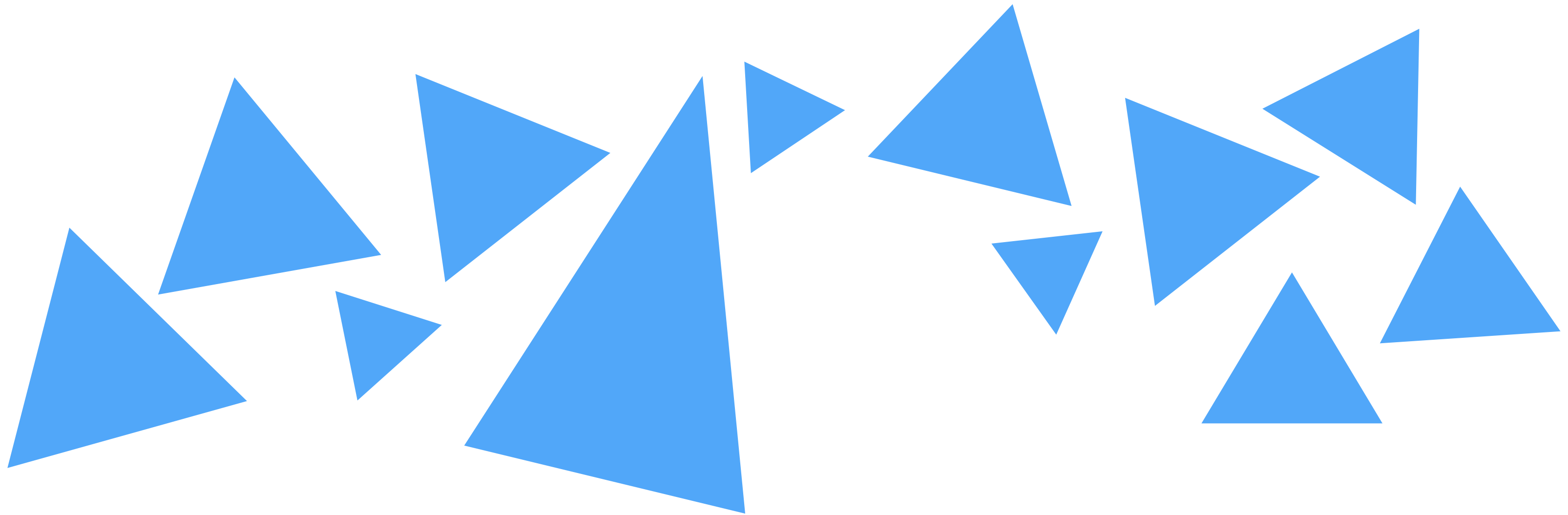
Interesting question of which child to enter first. How might you make a good decision?

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

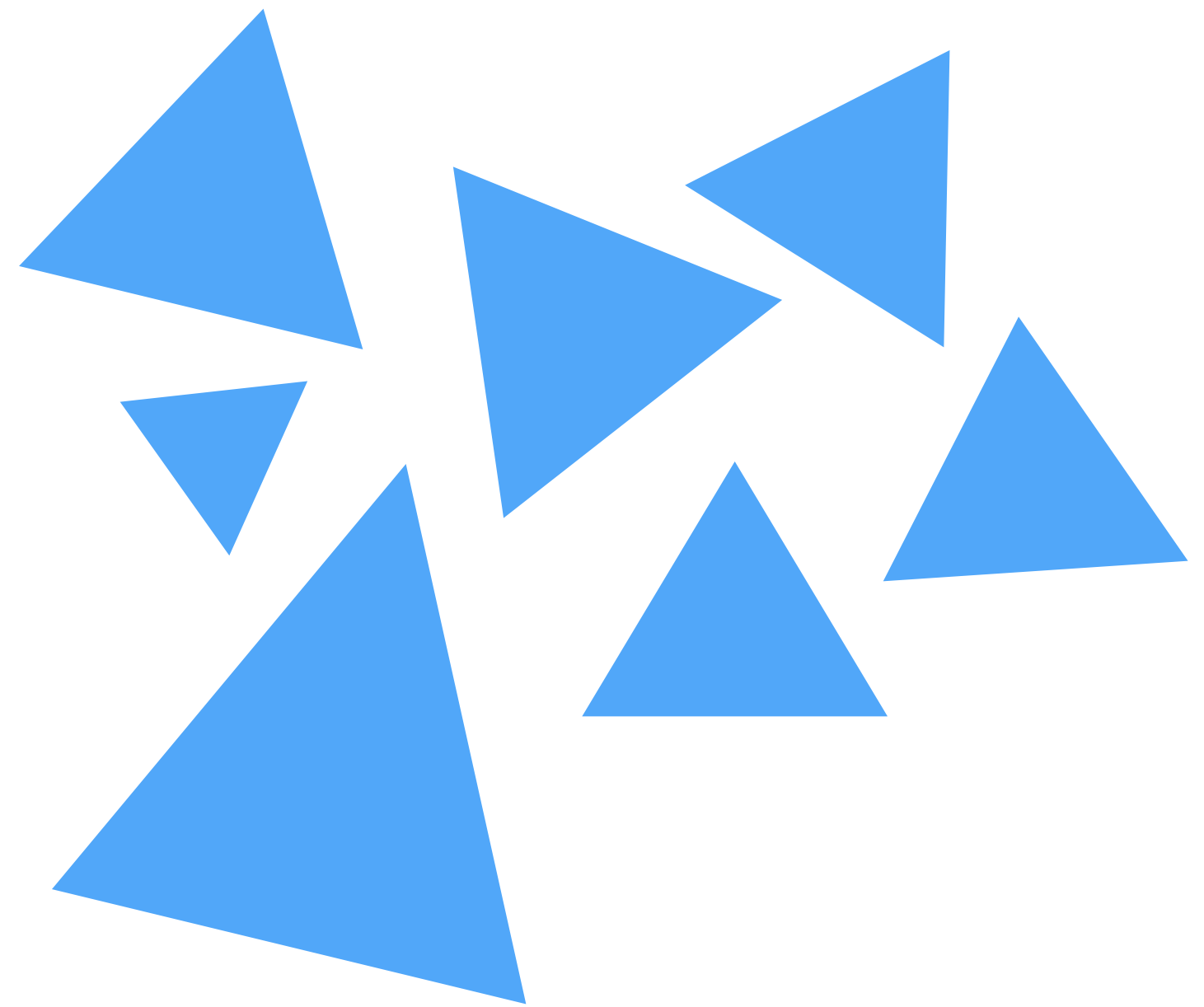
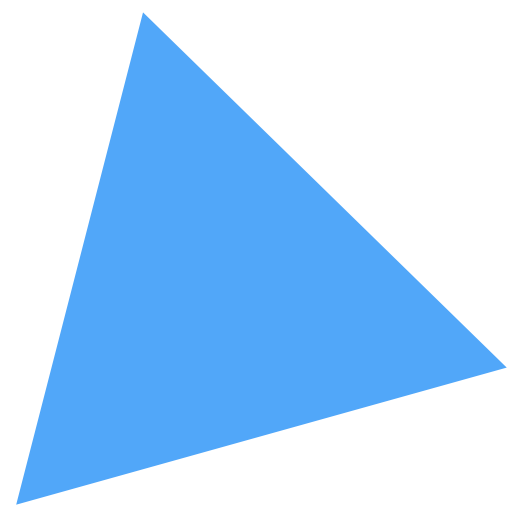
($\sim 2^N$ ways to partition N primitives into two groups)

How do we build a high-quality BVH?

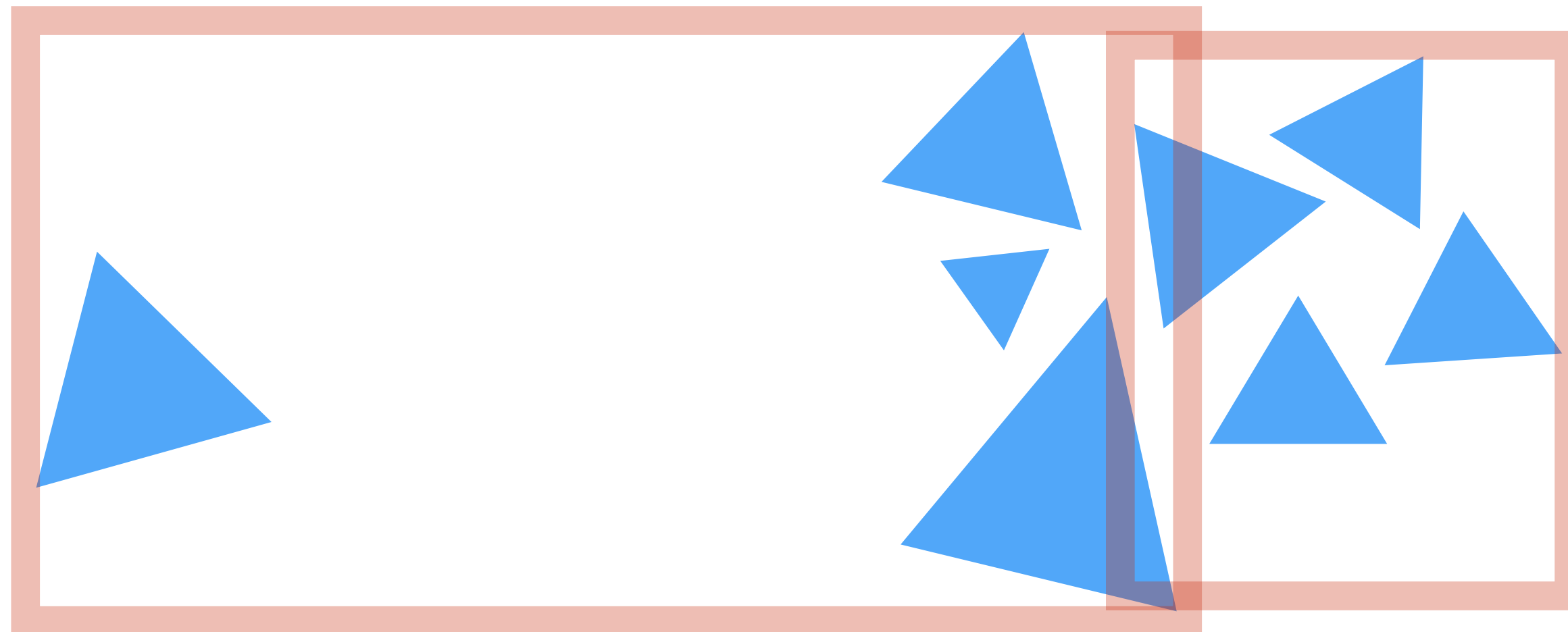
How would you partition these triangles into two groups?



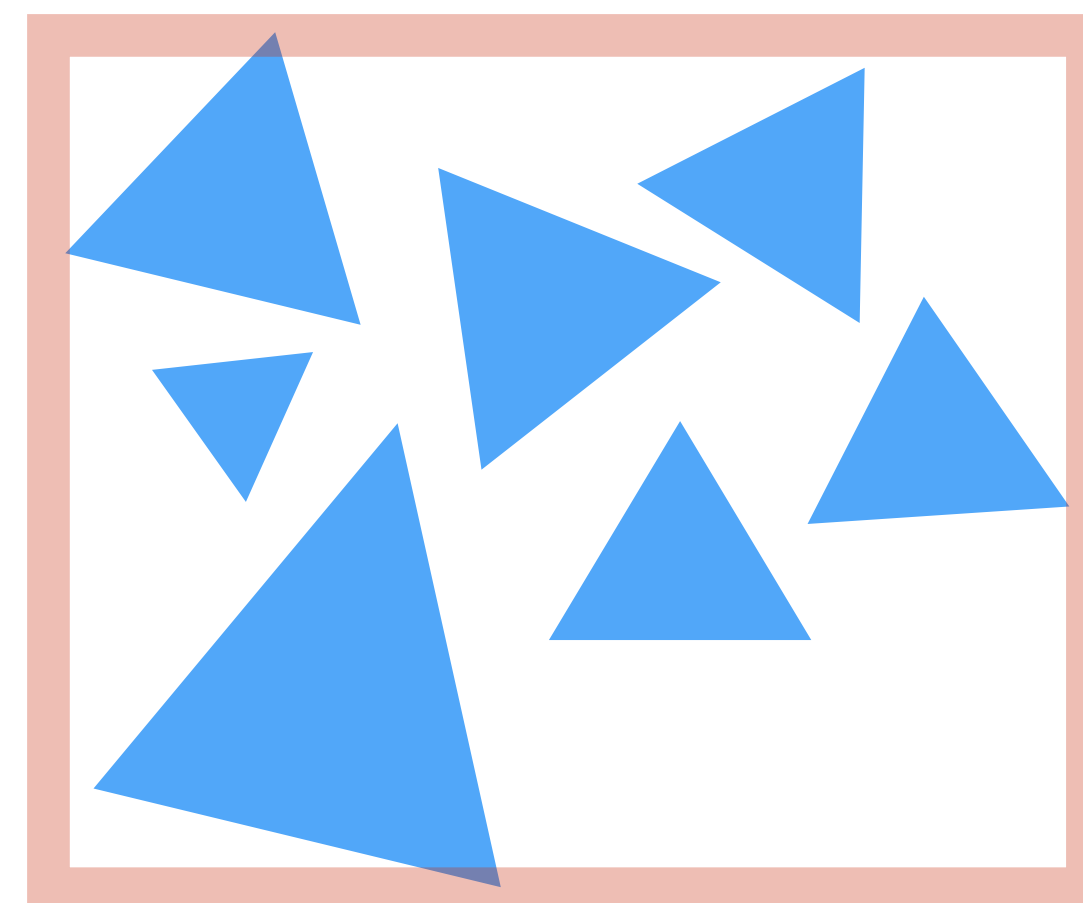
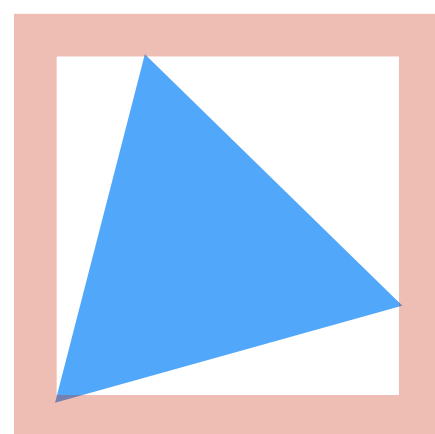
What about these?



Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: want small bounding boxes (minimize overlap between children, avoid empty space)

What are we really trying to do?

A good partitioning minimizes the cost of finding the closest intersection of a ray with primitives in the node.

If a node is a leaf node (no partitioning):

$$\begin{aligned} C &= \sum_{i=1}^N C_{\text{isect}}(i) \\ &= N C_{\text{isect}} \end{aligned}$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive i in the node.

(Common to assume all primitives have the same cost)

Cost of making a partition

The expected cost of ray-node intersection, given that the node's primitives are partitioned into child sets A and B is:

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

C_{trav} is the cost of traversing an interior node (e.g., load data, bbox check)

C_A and C_B are the costs of intersection with the resultant child subtrees

p_A and p_B are the probability a ray intersects the bbox of the child nodes A and B

Primitive count is common approximation for child node costs:

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

Where: $N_A = |A|$, $N_B = |B|$

Estimating probabilities

- For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas S_A and S_B of these objects.

$$P(\text{hit } A | \text{hit } B) = \frac{S_A}{S_B}$$

Surface area heuristic (SAH):

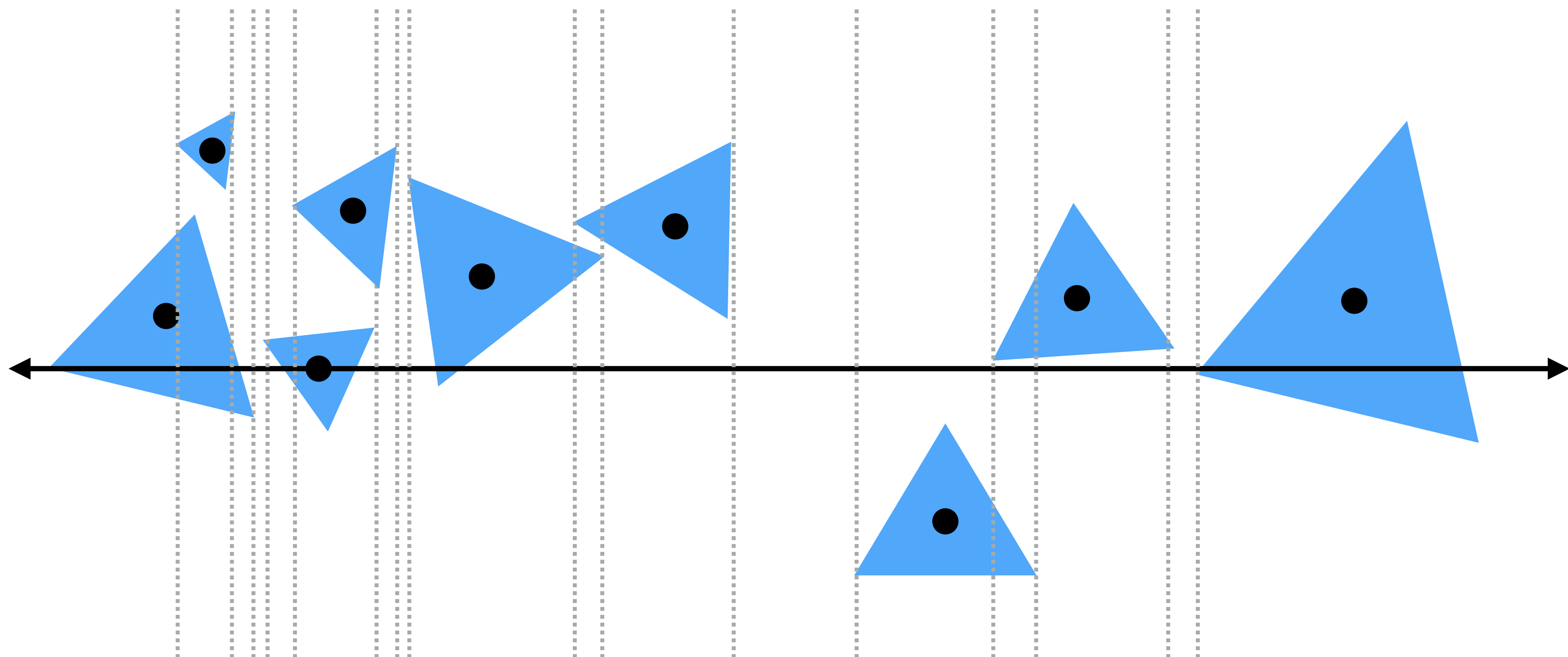
$$C = C_{\text{trav}} + \frac{S_A}{S_N} N_A C_{\text{isect}} + \frac{S_B}{S_N} N_B C_{\text{isect}}$$

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

- Rays are randomly distributed
- Rays are not occluded

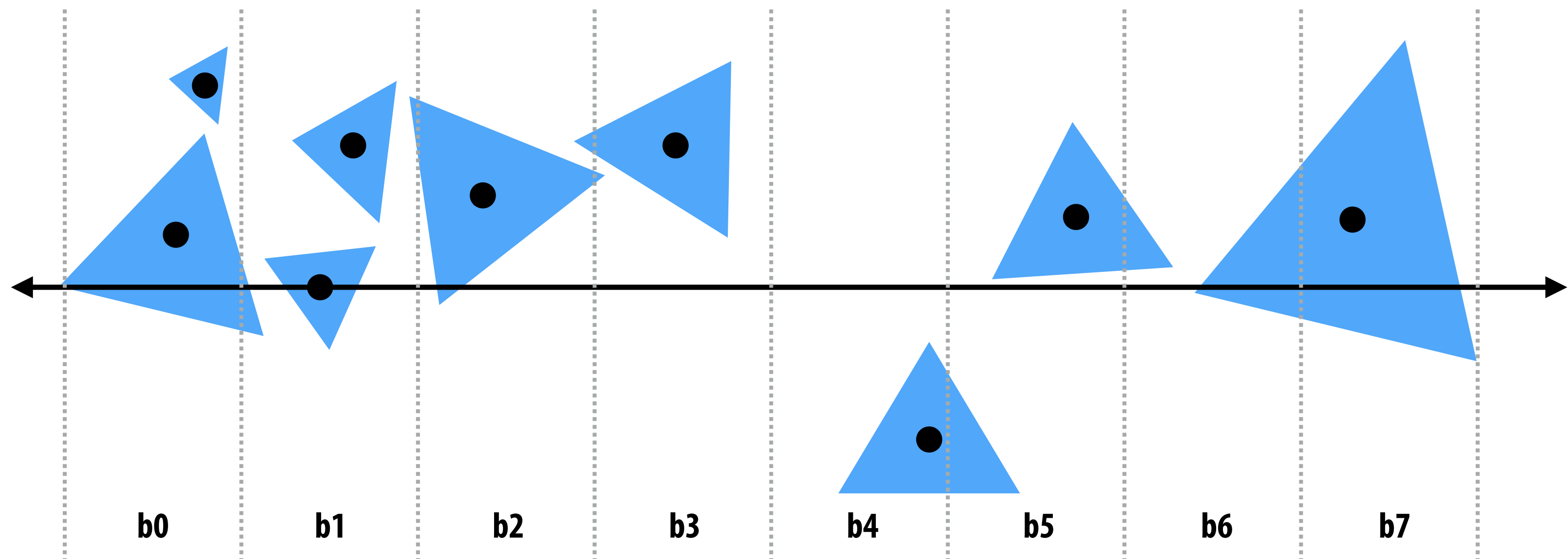
Implementing partitions

- **Constrain search for good partitions to axis-aligned spatial partitions**
 - **Choose an axis**
 - **Choose a split plane on that axis**
 - **Partition primitives by the side of splitting plane their centroid lies**
 - **$2N-2$ possible splitting positions for node with N primitives. (Why?)**



Efficiently implementing partitioning

- Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: $B < 32$)



For each axis: x, y, z :
initialize buckets

For each primitive p in node:

$b = \text{compute_bucket}(p.\text{centroid})$

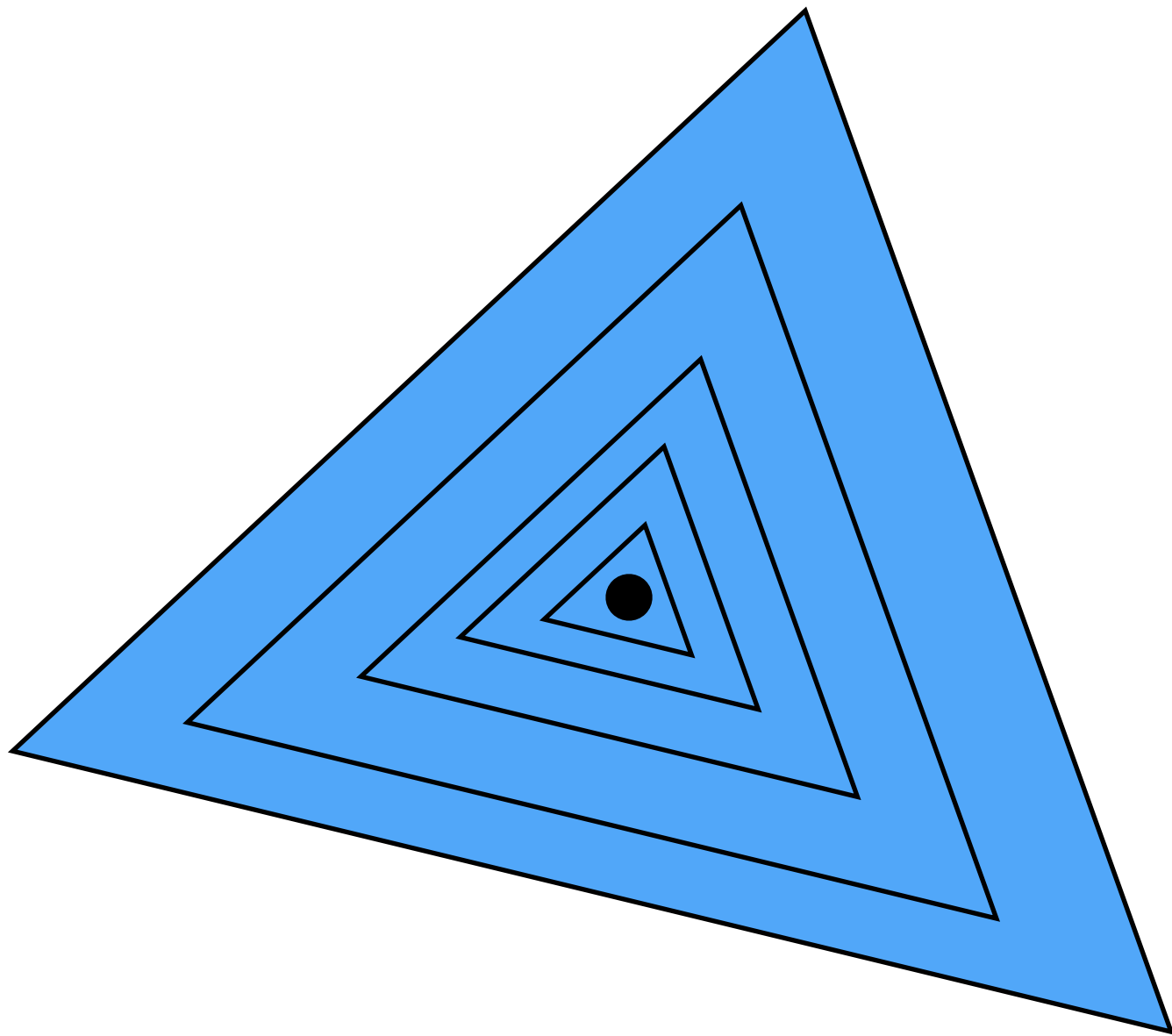
$b.\text{bbox.union}(p.\text{bbox});$

$b.\text{prim_count}++;$

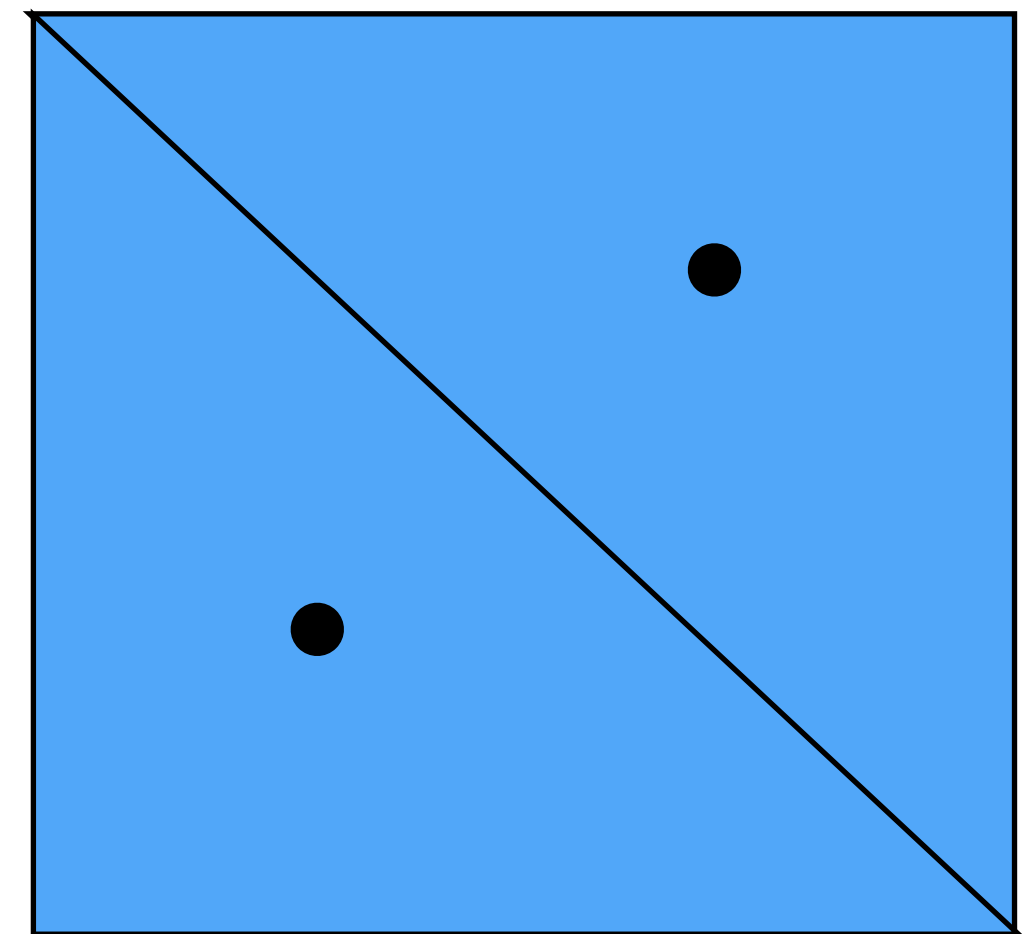
For each of the $B-1$ possible partitioning planes evaluate SAH

Execute lowest cost partitioning found (or make node a leaf)

Troublesome cases



All primitives with same centroid (all primitives end up in same partition)



All primitives with same bbox (ray often ends up visiting both partitions)

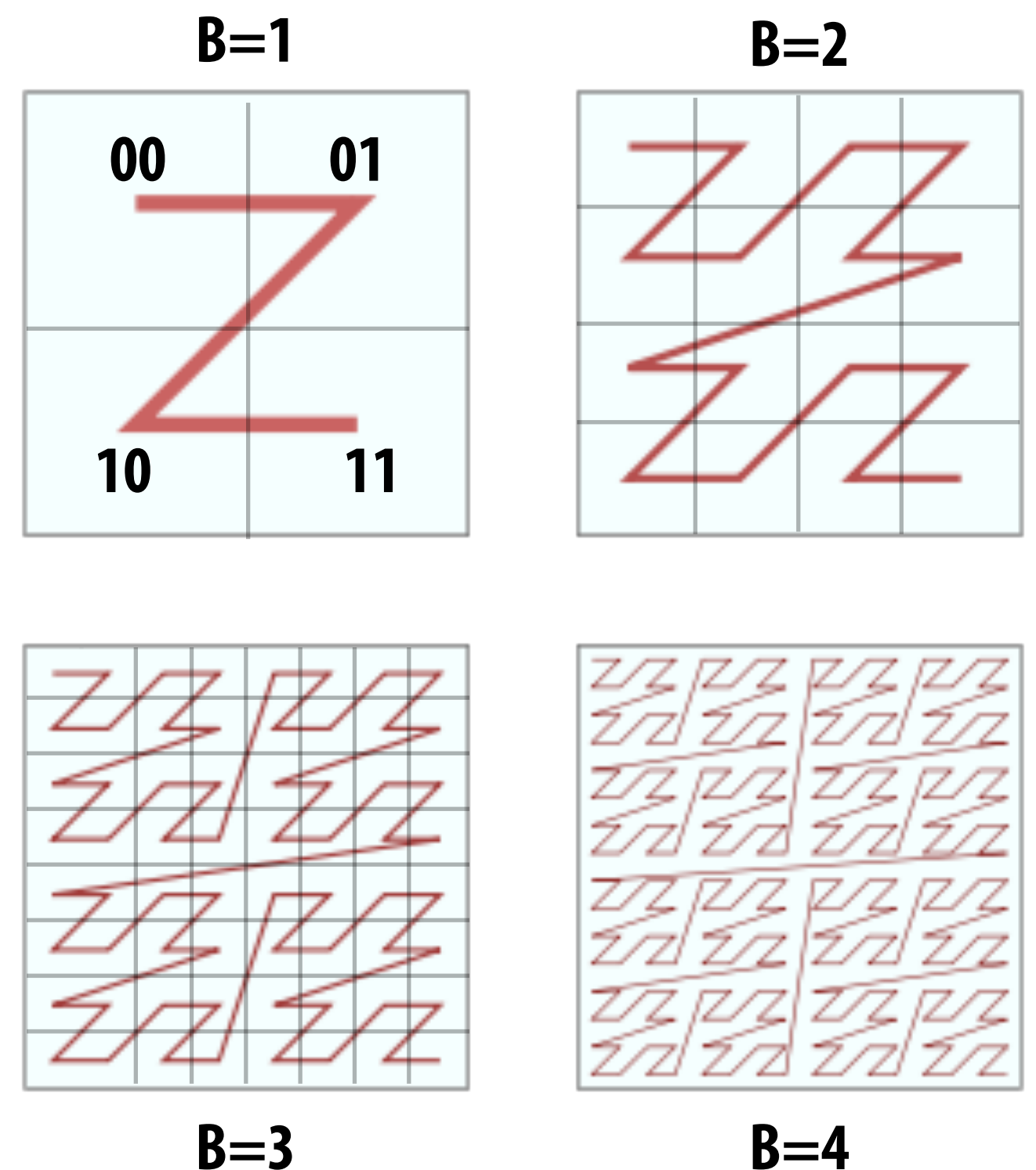
Building a low-quality BVH quickly

1. Discretize each dimension of scene into 2^B cells
2. Compute index of centroid of bounding box of each primitive:
(c_i, c_j, c_k)
3. Interleave bits of c_i, c_j, c_k to get $3B$ bit-Morton code
4. Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
 - $O(N)$ radix sort

Simple, highly parallelizable BVH build:

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

2D Morton Order

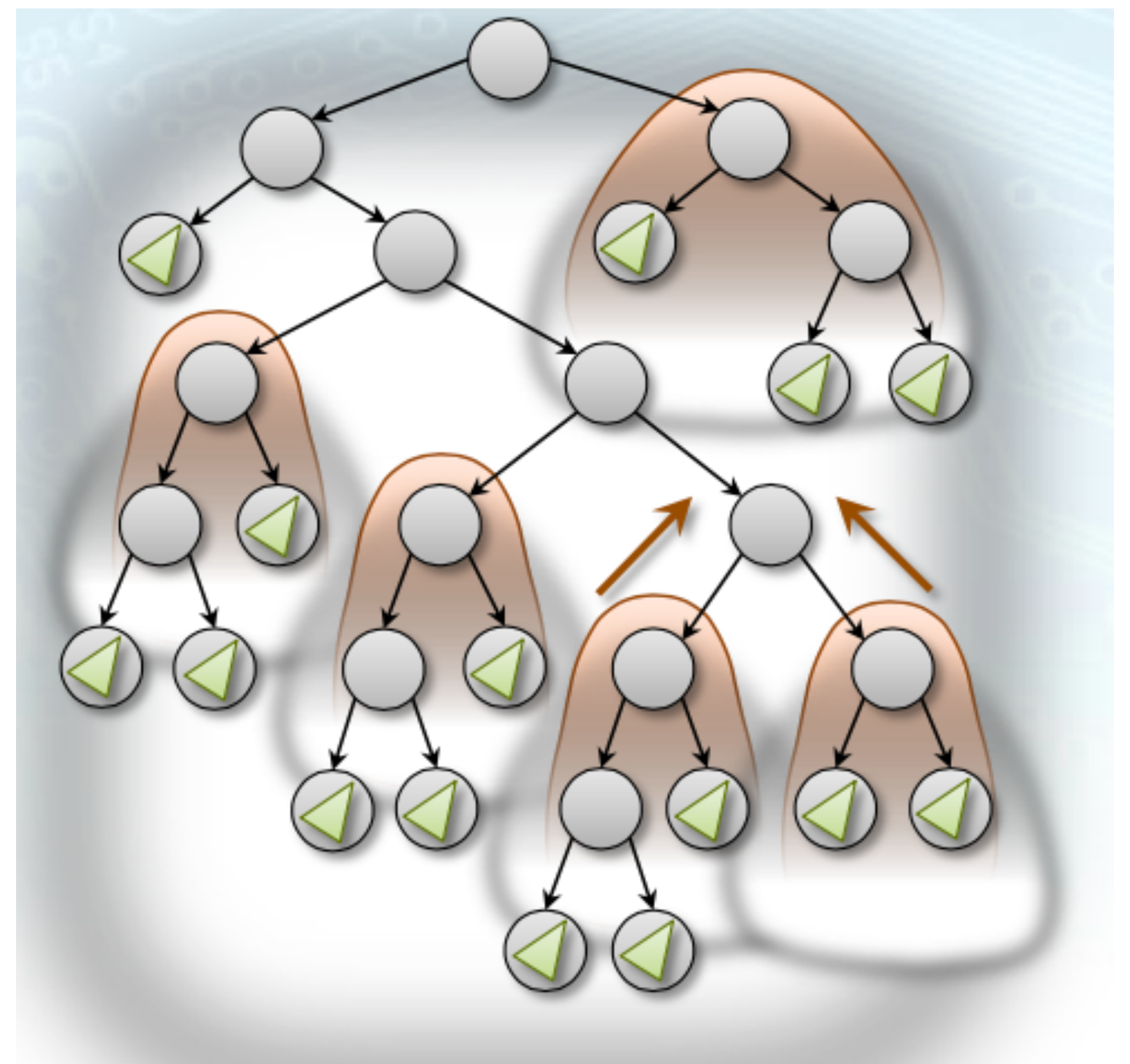


Modern, fast BVH construction schemes

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

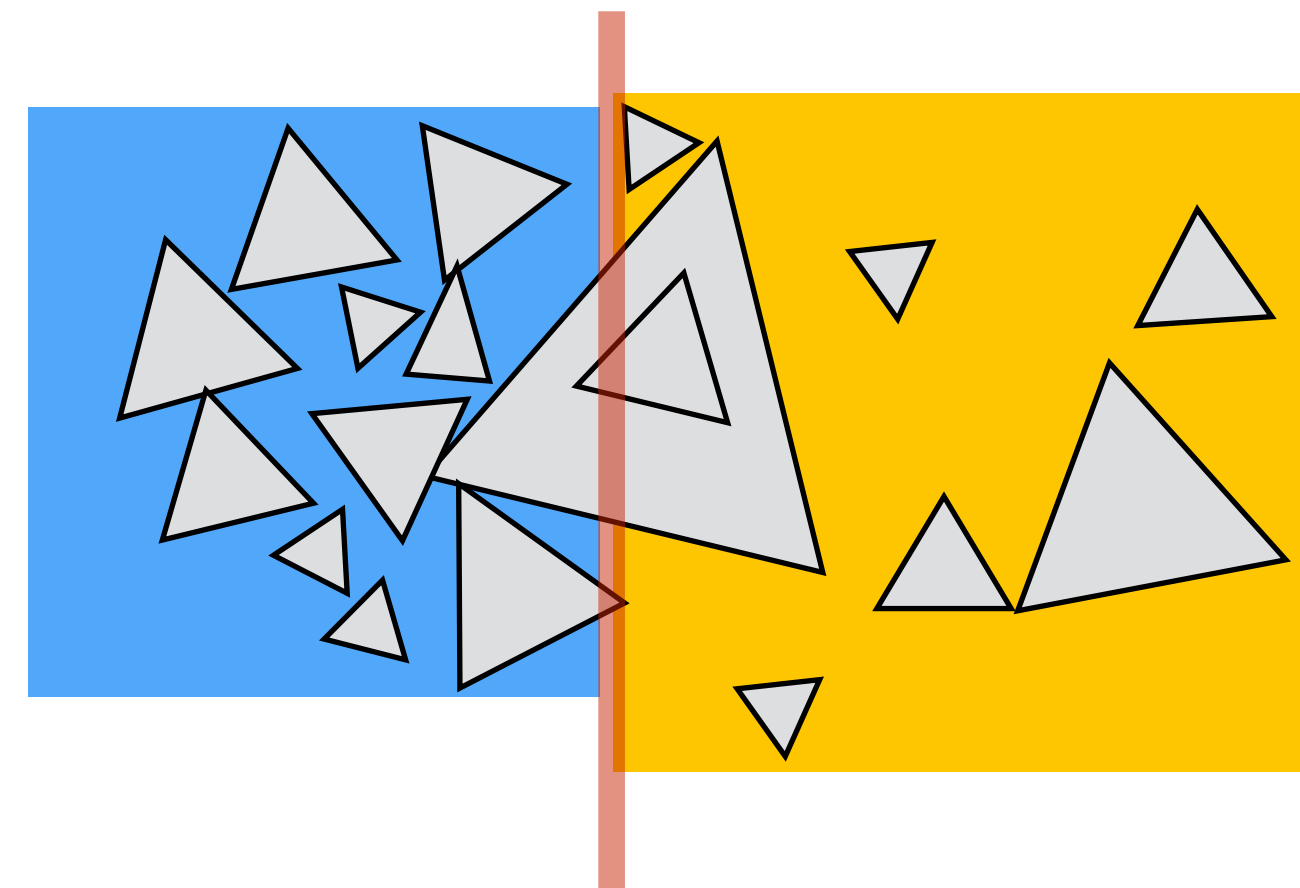
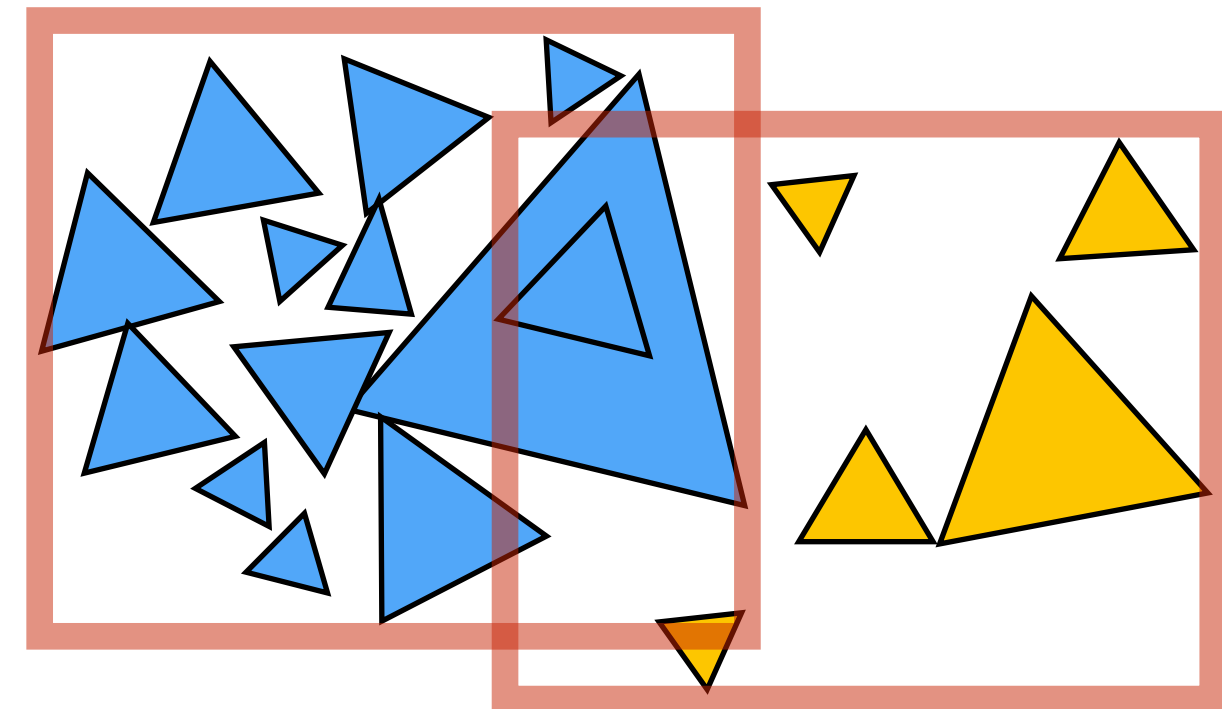
For all treelets of size $< N$ in original “low quality” BVH: (in parallel)

try all possible trees, keeping “optimal” topology that minimizes SAH for treelet



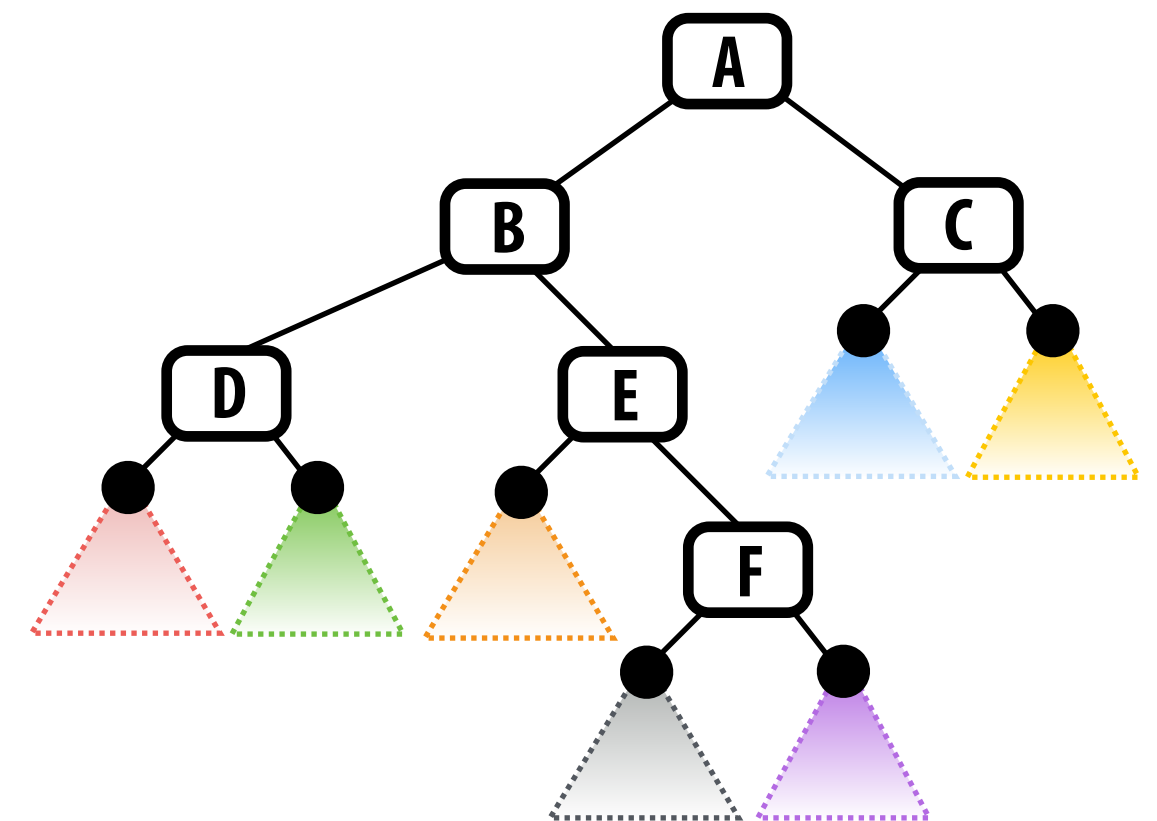
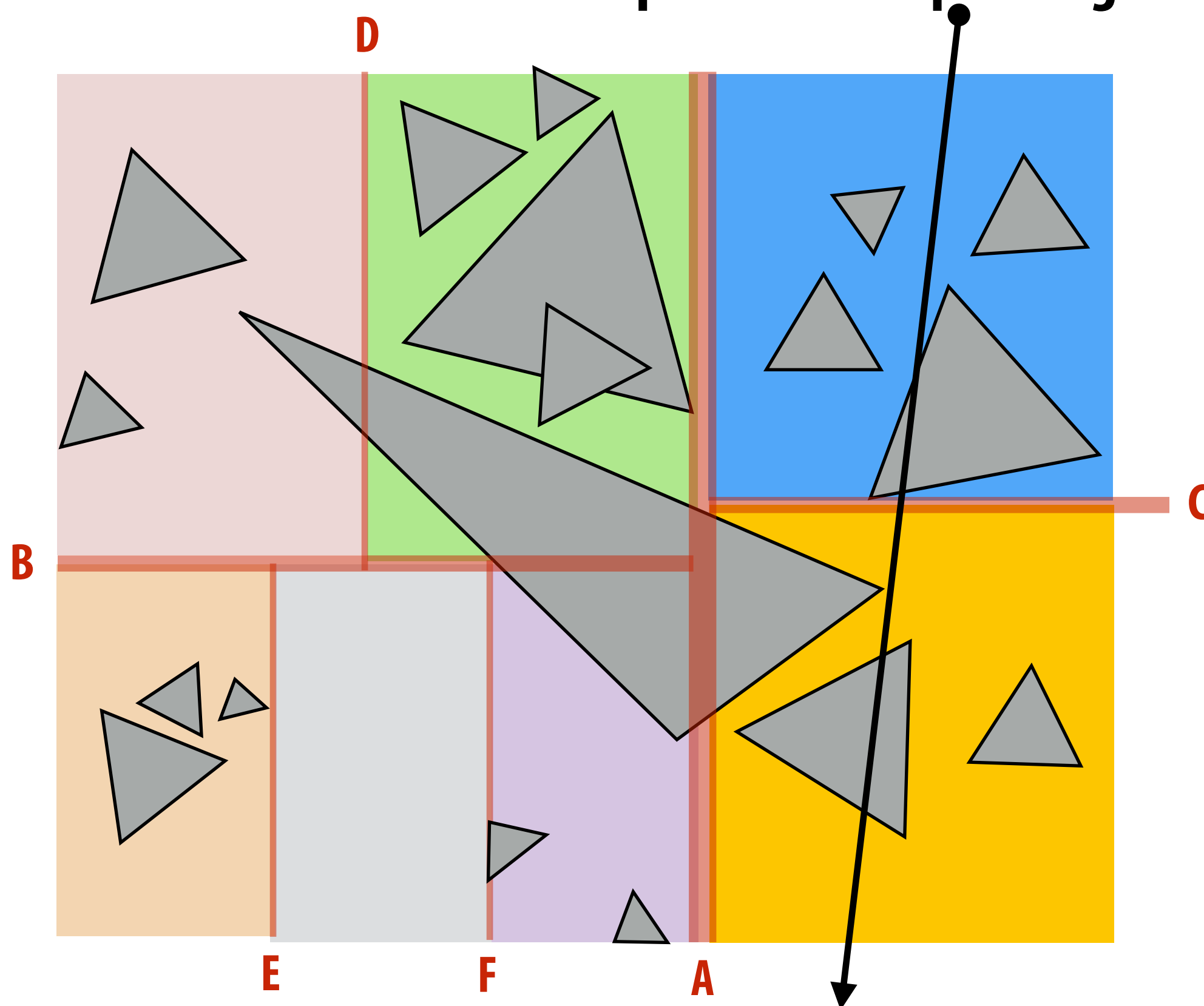
Primitive-partitioning acceleration structures vs. space-partitioning structures

- **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)**



K-D tree

- Recursively partition space via axis-aligned partitioning 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).
 - Intuition: partitions carve out empty space (construction of K-D tree may produce more tree nodes than primitives depending on ratio of C_{trav} and C_{isect})



Accelerating ray-scene queries using a BVH

Simplifications in today's discussion:

Will not discuss how to make BVH construction fast (we assume acceleration structure is given)

Assume scene acceleration structure is read-only: (no on-demand build, no on-demand tessellation)

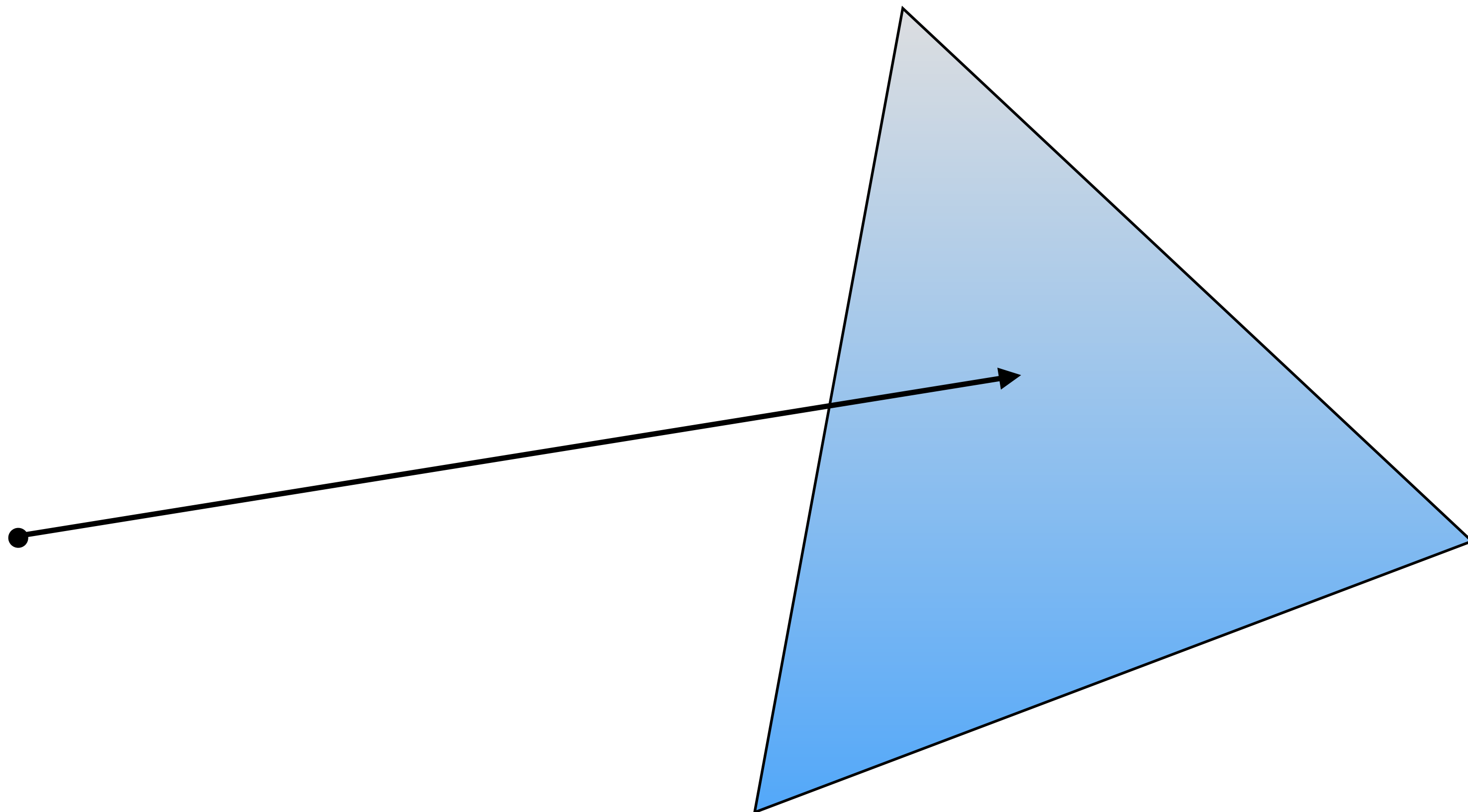
High-throughput ray tracing

- **Want work-efficient algorithms (do less)**
 - High-quality acceleration structures (minimize ray-box, ray-primitive tests)
 - Smart traversal algorithms (early termination, etc.)
- **Implementations for existing parallel hardware (CPUs/GPUs):**
 - High multi-core, SIMD execution efficiency
 - Help from fixed-function processing?
- **Bandwidth-efficient implementations:**
 - How to minimize bandwidth requirements?

Same issues we've talked about all class!

Tension between employing most work-efficient algorithms, and using available execution and bandwidth resources well.

Parallelizing ray-triangle tests?



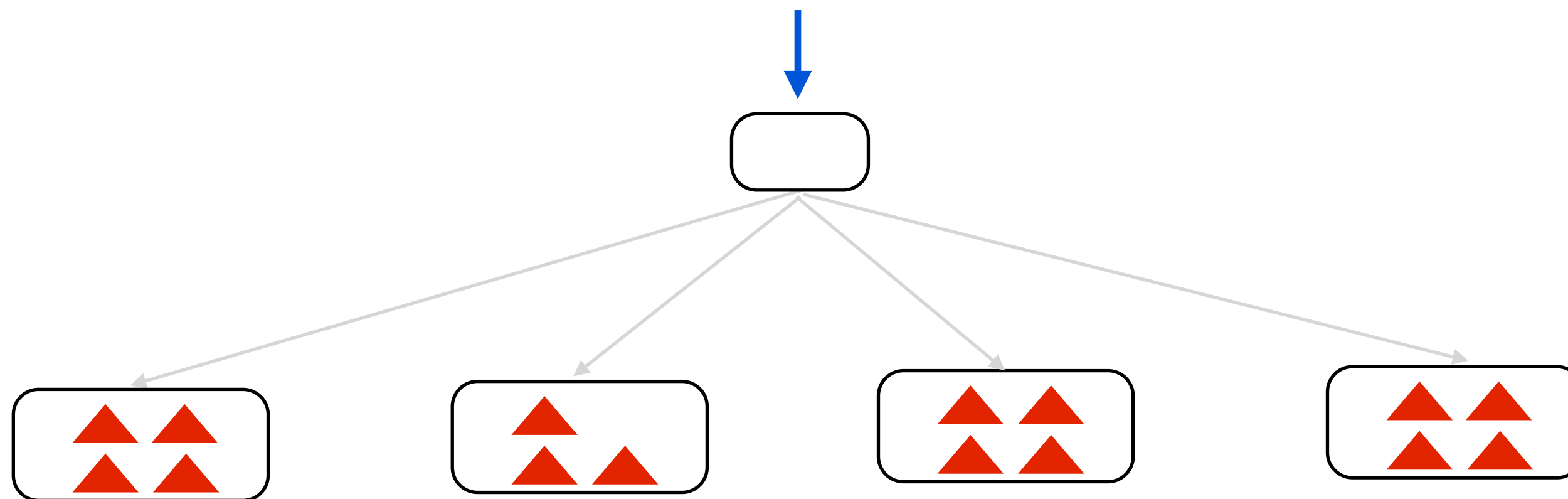
Parallelize ray-box, ray-triangle intersection

- **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 4 has similar work efficiency to branching factor 2**
 - **BVH with branching factor 8 or 16 is less work efficient (diminished benefit of leveraging SIMD execution)**



Parallelize across rays

- **Simultaneously intersect multiple rays with scene**

Simple ray tracer (using a BVH)

```
// stores information about closest hit found so far
struct ClosestHitInfo {
    Primitive primitive;
    float distance;
};

trace(Ray ray, BVHNode node, ClosestHitInfo hitInfo)
{
    if (!intersect(ray, node.bbox) || (closest point on box is farther than hitInfo.distance))
        return;

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


Ray packet tracing

[Wald et al. 2001]

Program explicitly intersects a collection of rays against BVH at once

RayPacket

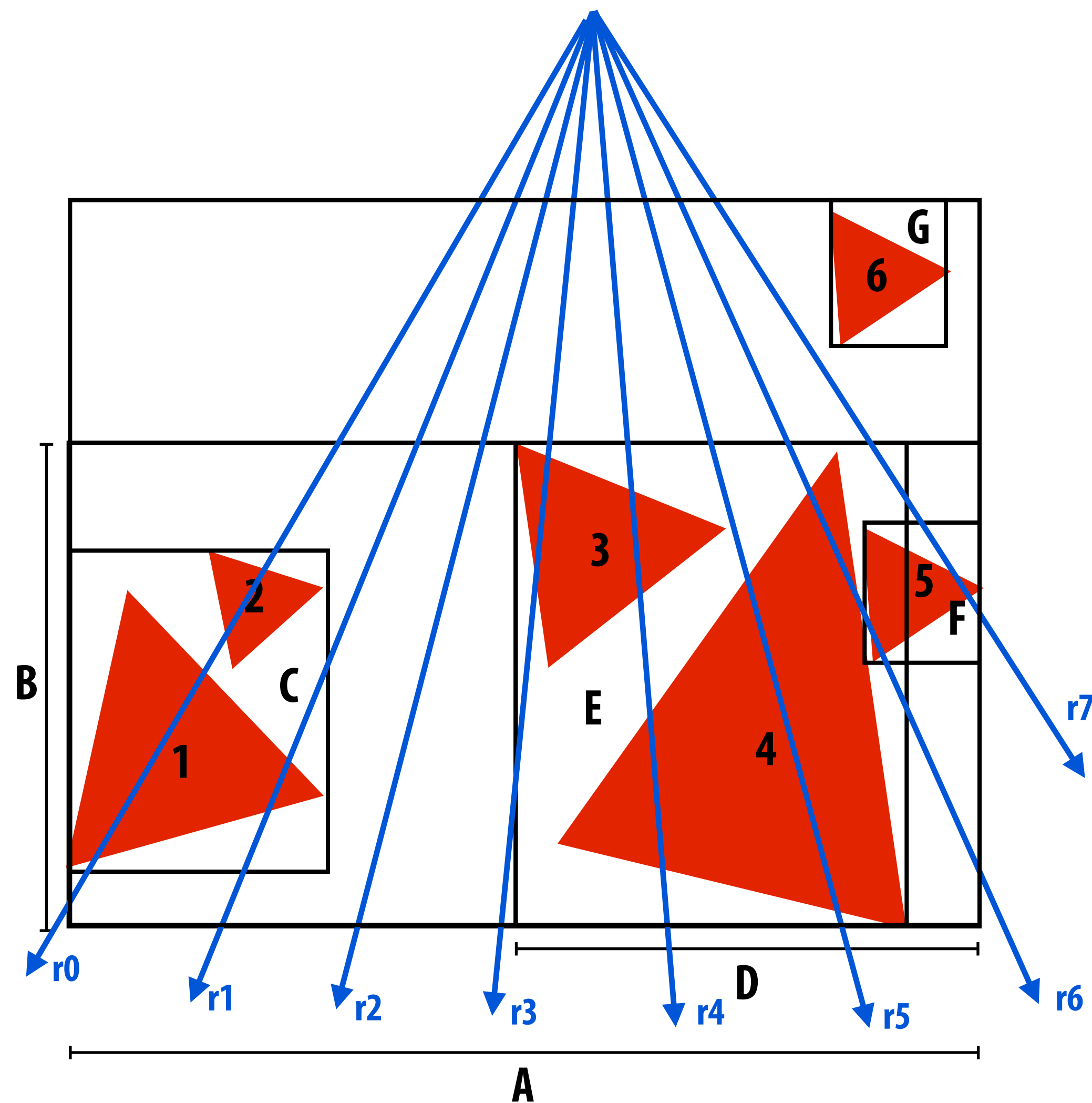
```
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

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;

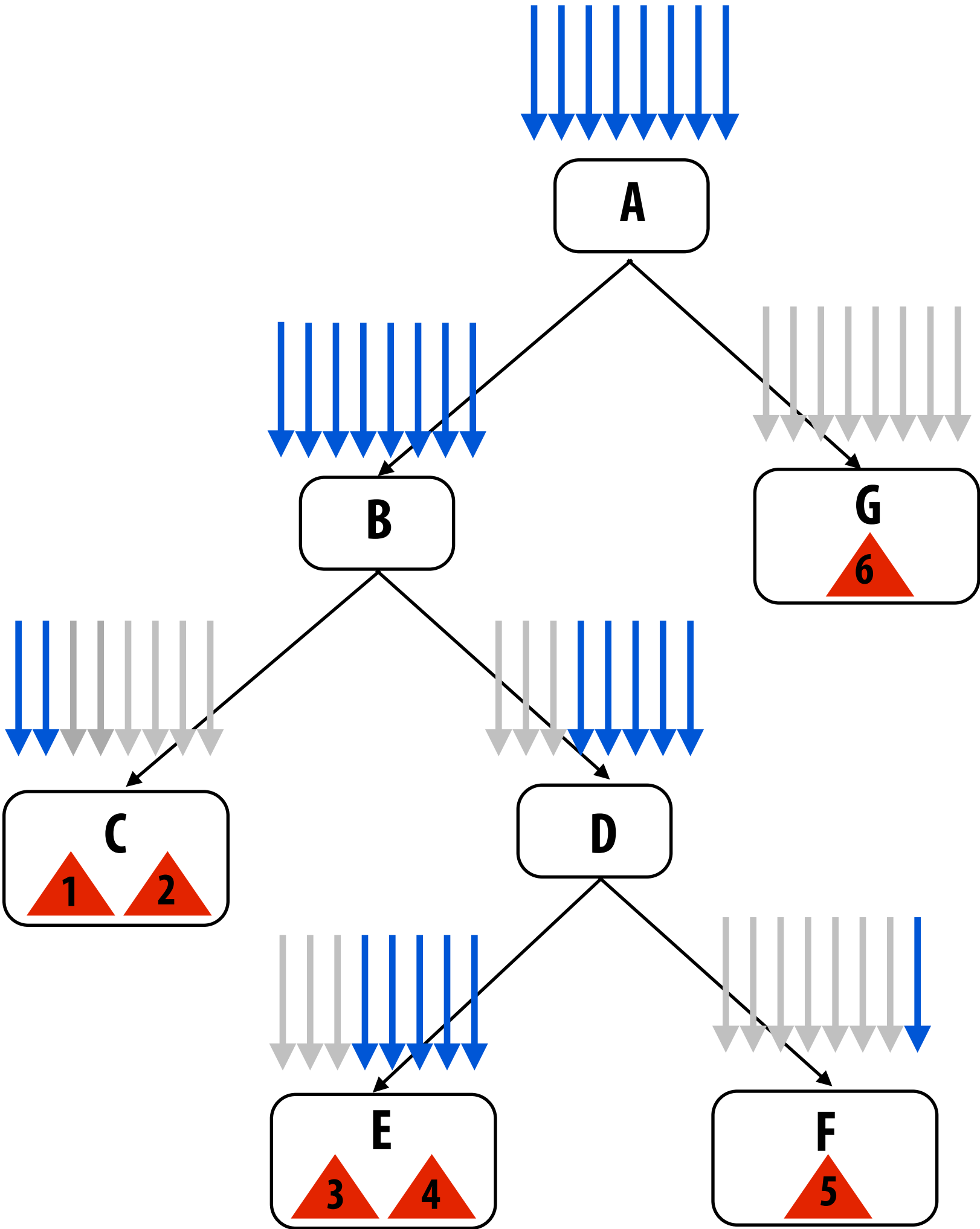
    update packet active mask

    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

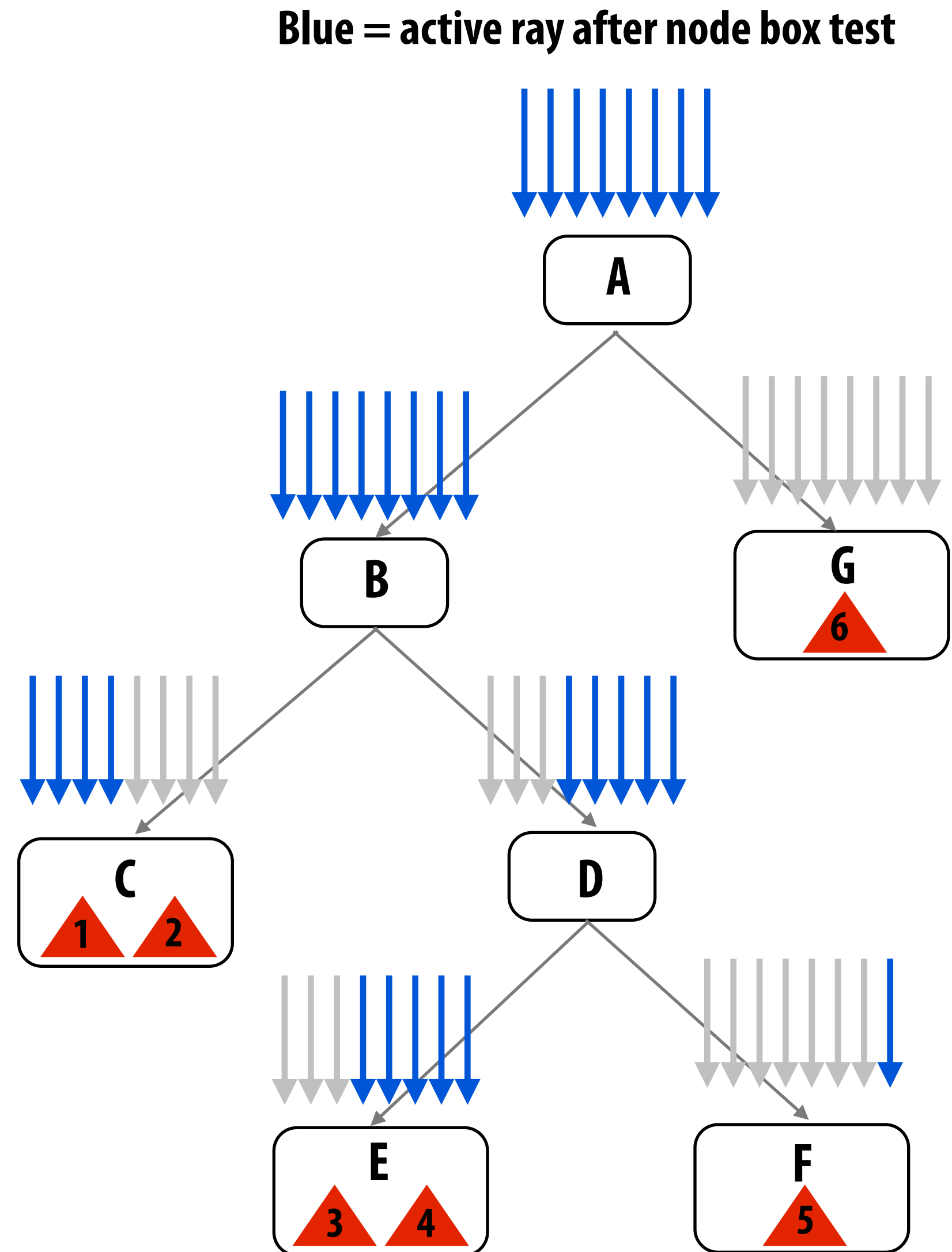
- Load BVH node once for all rays in packet (not once per ray)
- **Note: there is value to making packets bigger than SIMD width! (e.g., size = 64)**

■ 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!**

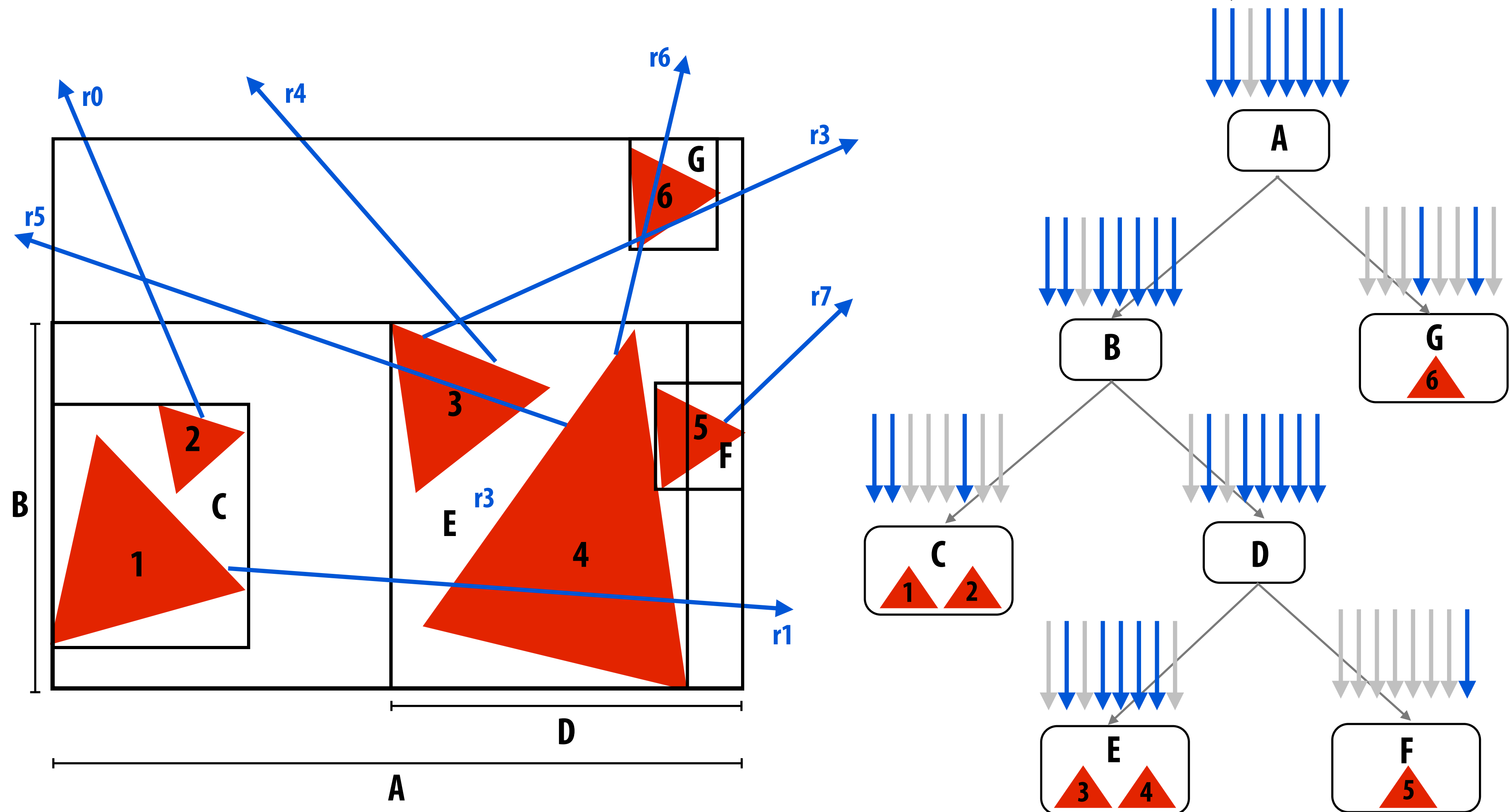
Disadvantages of packets

- 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



Ray packet tracing: incoherent rays

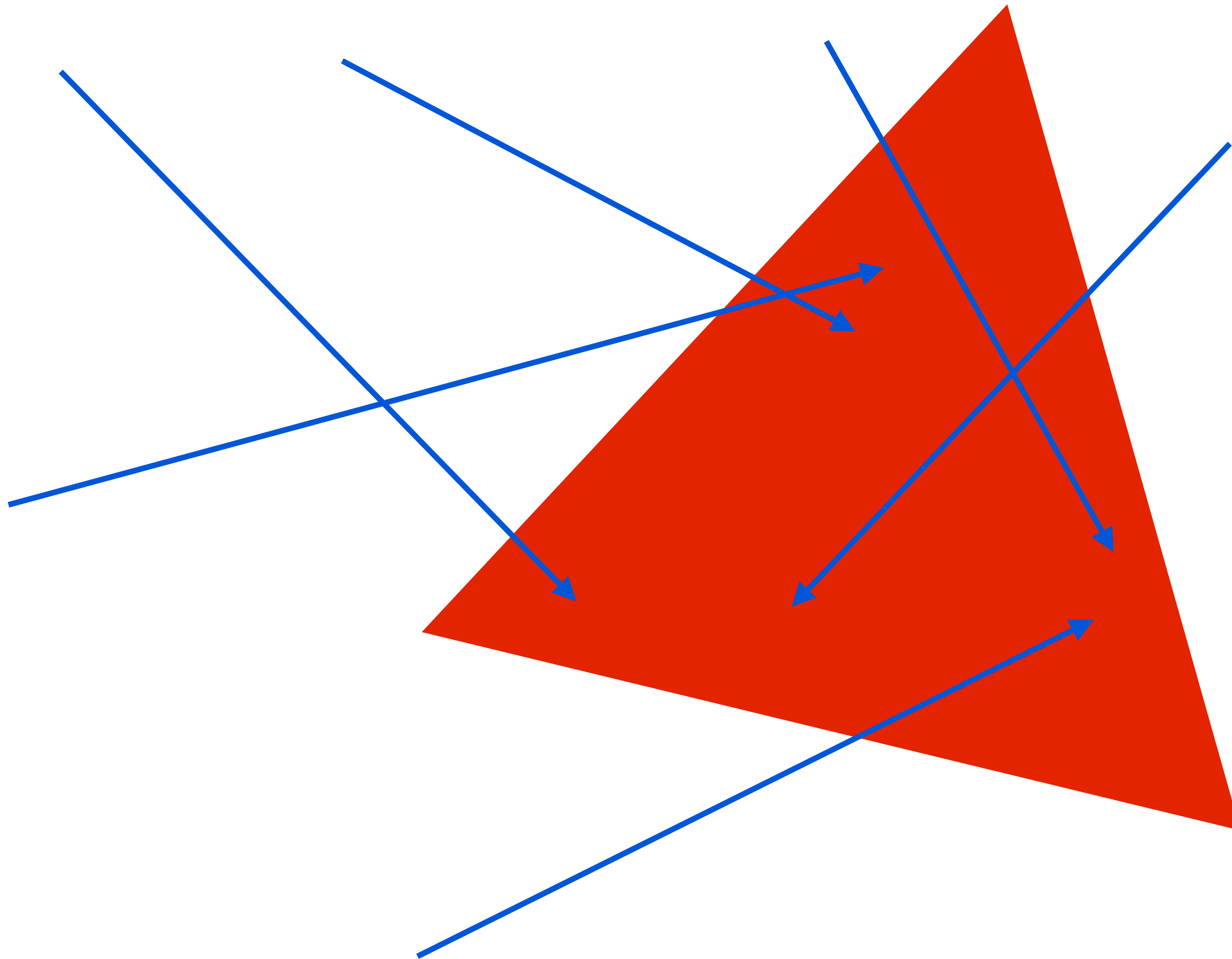
Blue = active ray after node box test



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!)

Incoherent rays

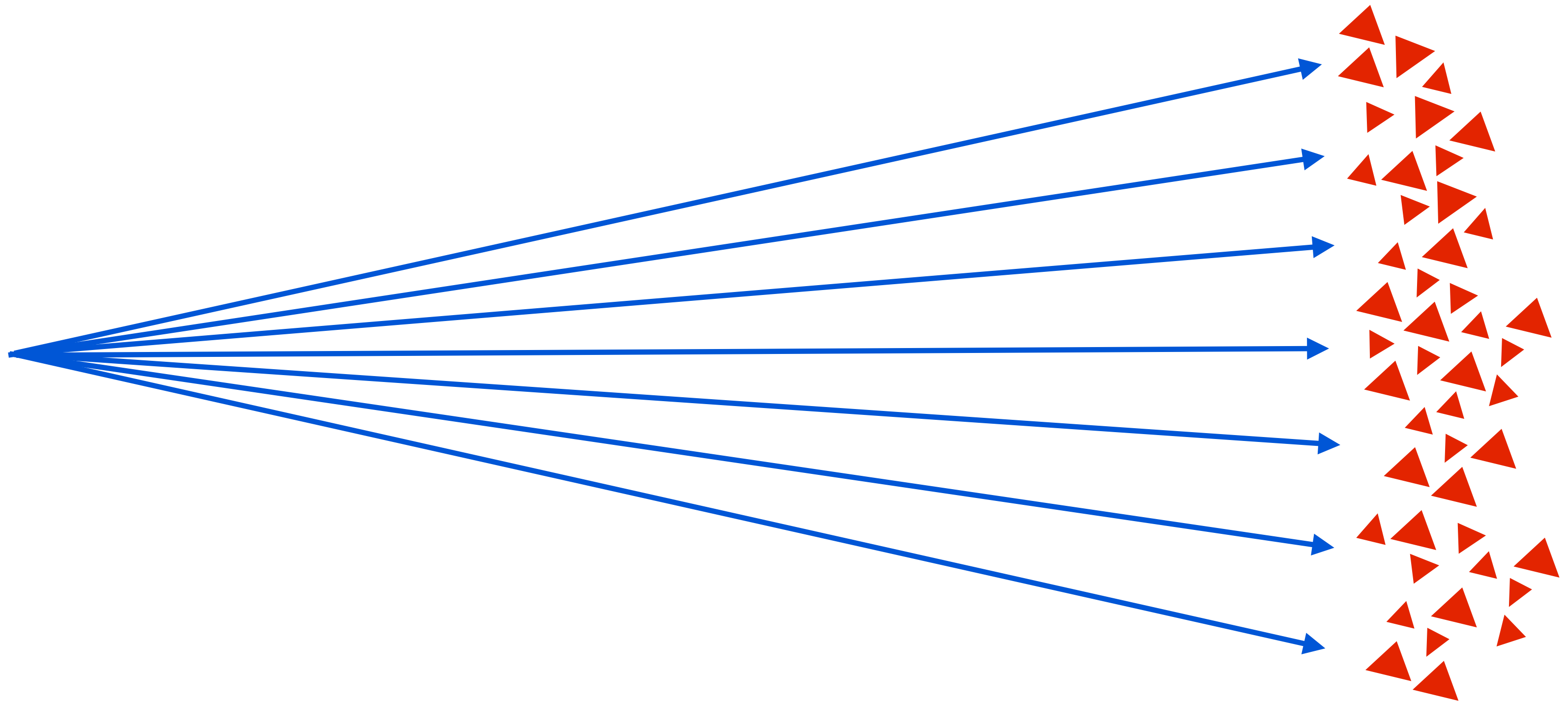
Incoherence is a property of both the rays and the scene



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



**Camera rays 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)

Improving packet tracing with ray reordering

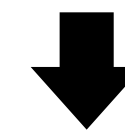
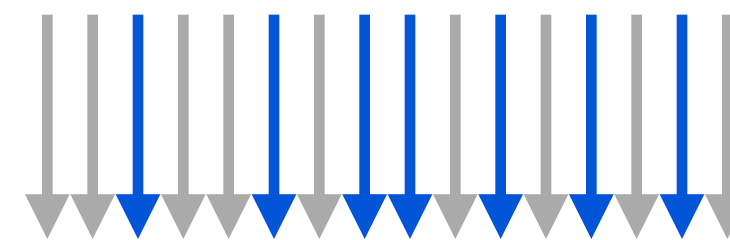
[Boulos et al. 2008]

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

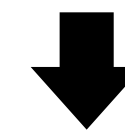
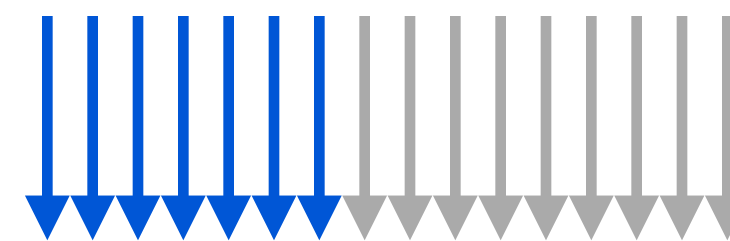
- **Increases SIMD utilization**
- **Amortization benefits of smaller packets, but not large packets**

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

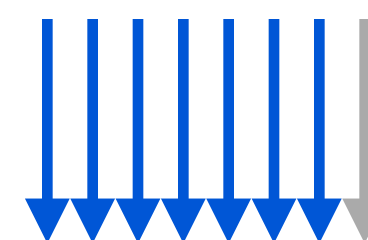
16-ray packet: 7 of 16 rays active



**Reorder rays
Recompute intervals/bounds for active rays**



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



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

SPMD ray tracing

No packets!

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

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
```

Data access challenges

■ Recall data access patterns in rasterization

- Stream through scene geometry
- Arbitrary, direct access to frame-buffer samples (accelerated by specialized GPU implementations)

■ Ray tracer data access patterns

- Frame-buffer access is minimal (once per ray)
- But access to BVH nodes is frequent and unpredictable
 - Not predictable by definition (or the BVH is low quality. Why?)
 - Packets amortize cost of fetching BVH node data, but technique's utility diminishes under divergent conditions.

■ Incoherent ray traversal suffers from poor cache behavior

- Rays require different BVH nodes during traversal
- Ray-scene intersection becomes bandwidth bound for incoherent rays
 - E.g., soft shadows, indirect diffuse bounce rays

Let's stop and think

- **One strong argument for high-performance ray tracing is to produce advanced effects that are difficult or inefficient to compute given the single point of projection and uniform sampling constraints of rasterization**
 - **e.g., soft shadows, diffuse interreflections**
- **But these phenomenon create situations of high ray divergence! (where packet- and SIMD-optimizations are less effective)**

Emerging hardware for ray tracing

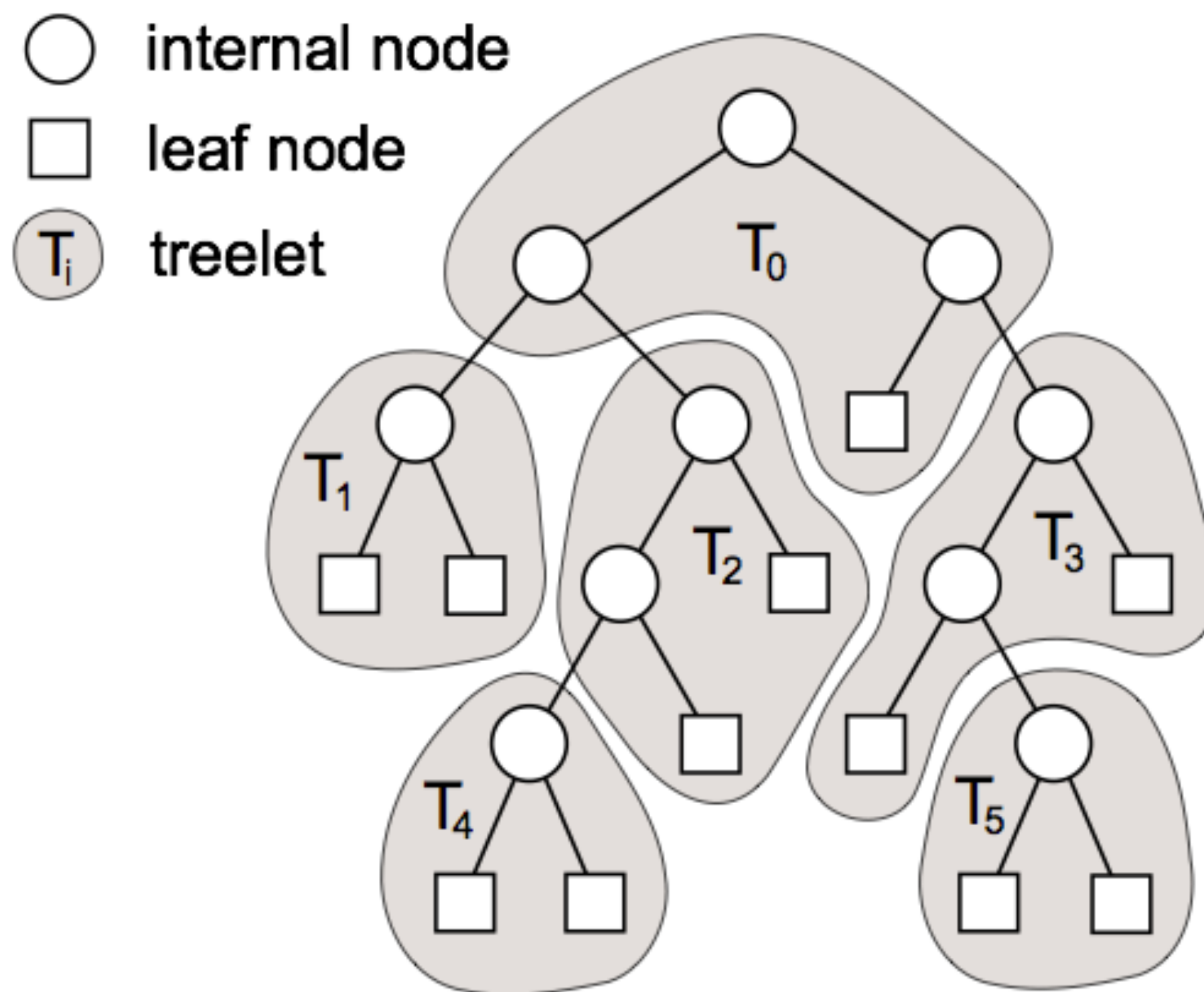
Emerging hardware for ray tracing

- **Modern academic/announced industry 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

[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



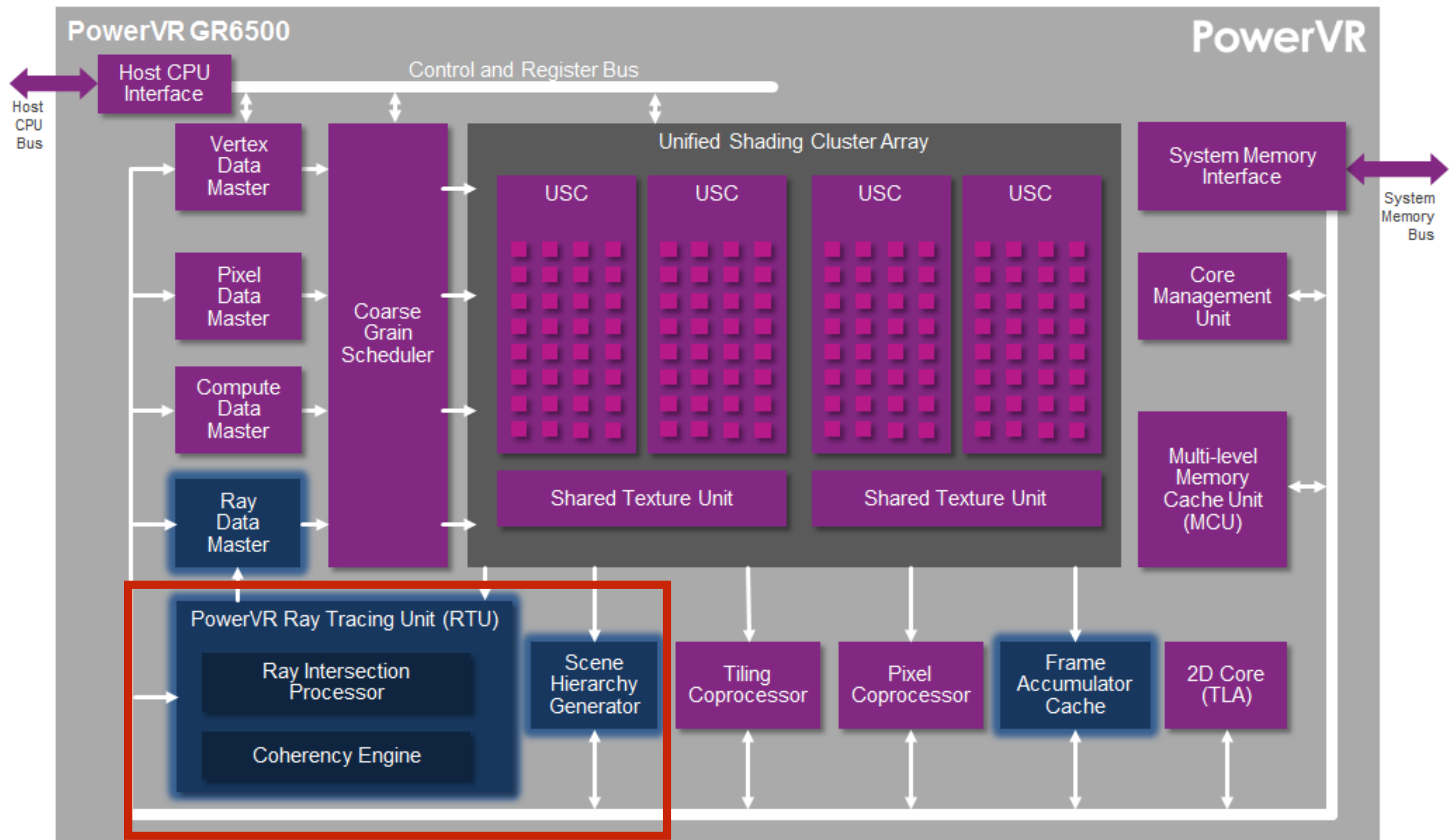
Partition BVH into treelets
(treelets sized for L1 or L2 cache)

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)

PowerVR GR6500 ray tracing GPU



Summary

Visibility summary

- **Visibility problem: determine which scene geometry contributes to the appearance of which screen pixels**
 - “Basic” rasterization: given polygon, find samples(s) it overlaps
 - “Basic” ray tracing: given ray, find triangle(s) that it intersects
- **In practice, optimized versions of both algorithms are not as different as you might think**
- **They are just different ways to solve the problem of finding interacting pairs between two hierarchies**
 - Hierarchy over point samples (tiles, ray packets)
 - Hierarchy over geometry (BVHs)

Consider performant, modern solutions for primary-ray visibility

■ “Rasterizer”

- Hierarchical rasterization (uniform grid over samples)
 - Hierarchical depth culling (quad-tree over samples)
 - Application scene graph, hierarchy over geometry
 - Modern games perform conservative coarse culling, only submit potentially visible geometry to the rendering pipeline
- (in practice, rasterization not linear in amount of geometry in scene)

■ “Ray tracer”

- BVH: hierarchy over geometry
 - Packets form hierarchy over samples (akin to frame buffer tiles). Breaking packets into small packets during traversal adds complexity to the hierarchy
 - Wide packet traversal, high-branching BVH: decrease work efficiency for better machine utilization
- (in practice, significant constants in front of that $\lg(N)$)