

**Lecture 10:**

# **Real-time ray tracing**

**(and opportunities for hardware acceleration)**

---

**Visual Computing Systems**  
**CMU 15-869, Fall 2013**



# Recent push towards real-time ray tracing

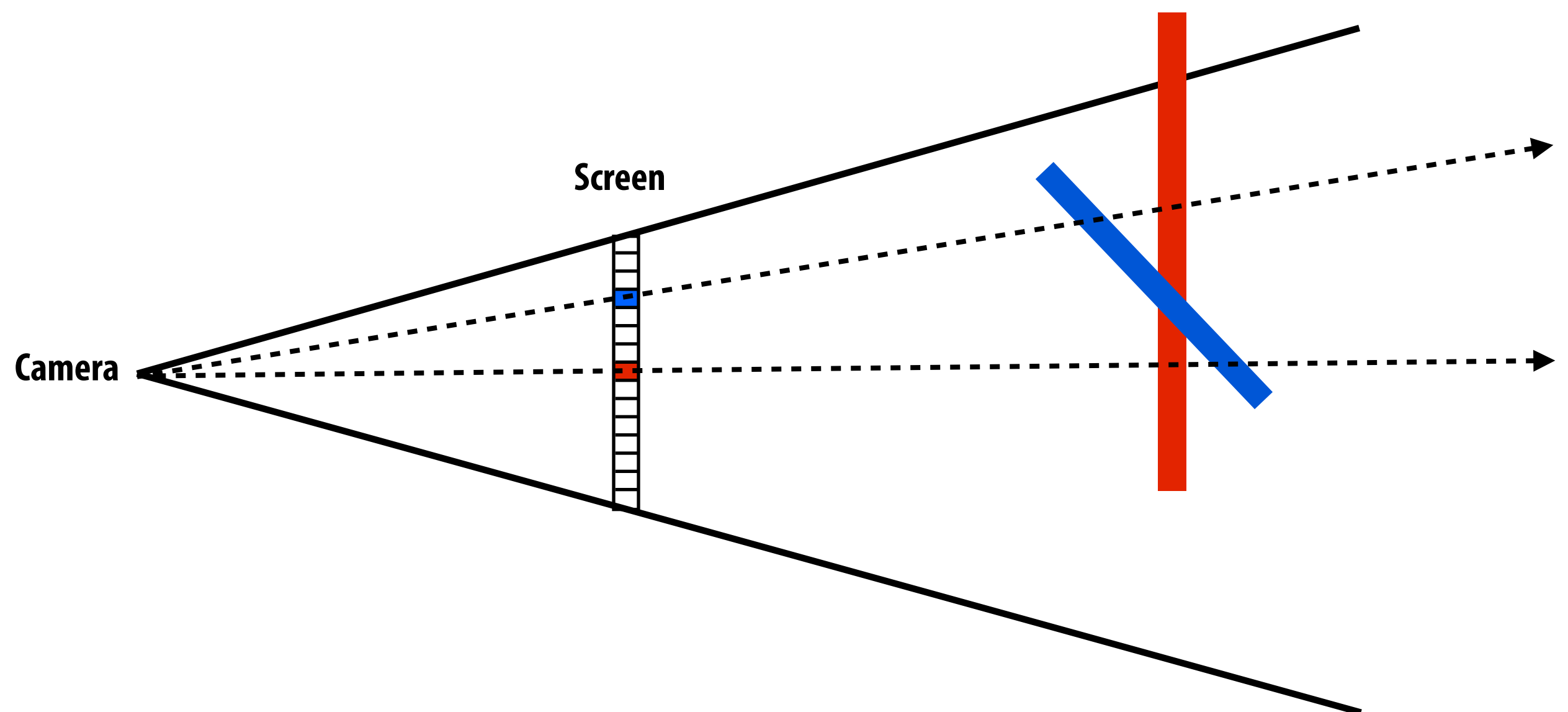


Image credit: NVIDIA (this ray traced image can be rendered at interactive rates on modern GPUs)



# Review: visibility

- **Problem: determine what scene geometry contributes to the appearance of which screen pixels**
- **Visibility can be thought of as a search problem**
  - So far in this course: given triangle, find samples it contributes to
  - Today: given sample, find triangle(s) that contribute to it



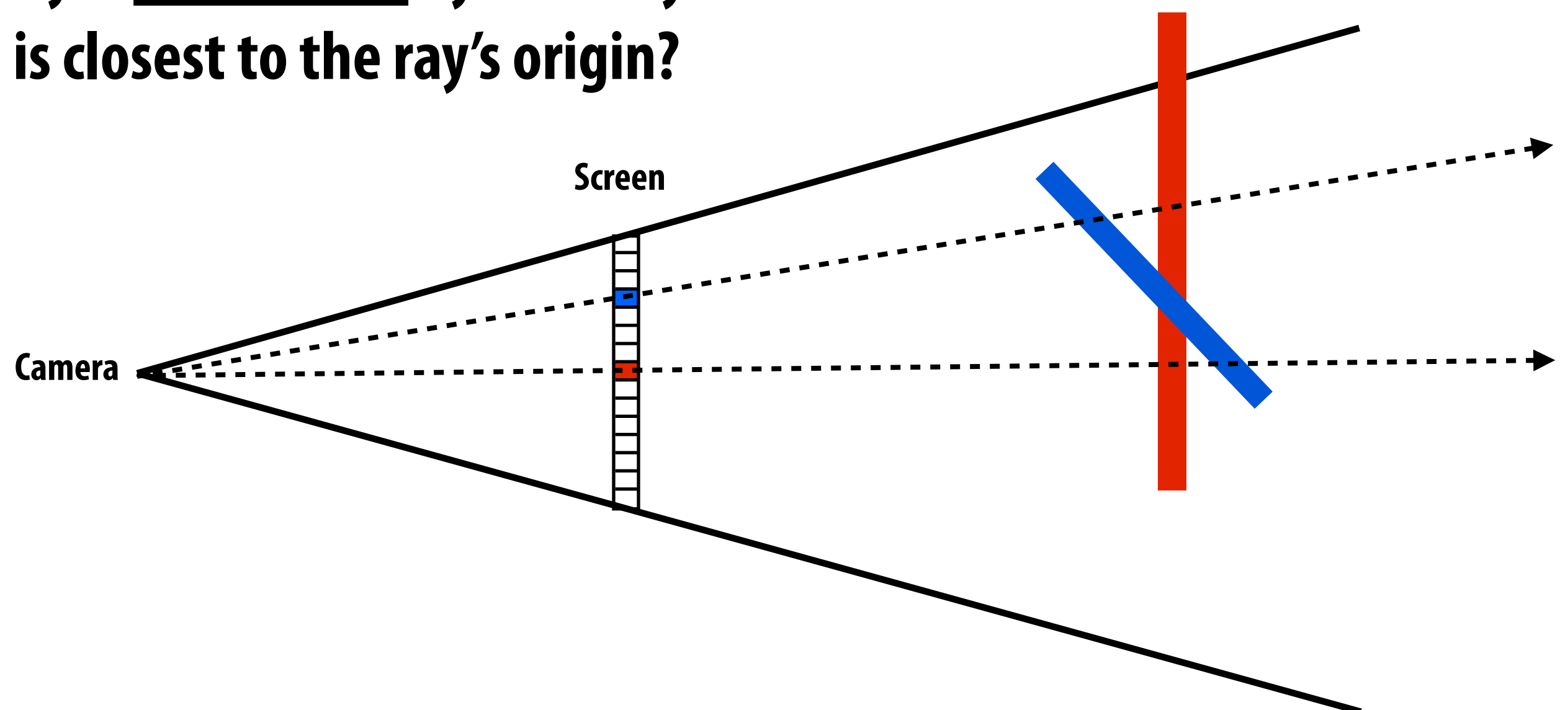
# Visibility as a search problem

## ■ Rasterization formulation:

- Sample = 2D point on screen
- What scene geometry, after projection into 2D, covers each visibility sample?
- Coverage (what triangles cover) + occlusion (closest covering triangle)

## ■ Ray casting formulation:

- Sample = ray in 3D (ray = (origin, direction))
- What scene geometry is intersected by each ray?
- Which intersection is closest to the ray's origin?



# Two naive visibility algorithms \*

## Naive “rasterizer”:

```
initialize z_closest[] to INFINITY           // store closest for all samples
for each triangle t in scene:                // stream over triangles in outer loop
    t_proj = project_triangle(t)
    for each sample s in frame buffer:
        if (t_proj covers s)
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s];
```

## Naive “ray caster”:

```
for each sample s in frame buffer:           // stream over samples (rays) in outer loop
    ray = generate ray from camera through s out into scene
    ray_closest = INFINITY                    // store only closest point for current ray
    closest_tri = NULL;
    for each triangle t in scene:
        if (ray intersects t)
            if (intersection point is closer than ray_closest)
                update ray_closest;
                closest_tri = t;
```

\* As we will discuss, as optimizations get added the difference between these two approaches blurs

# Recall: the rendering equation

[Kajiya 86]

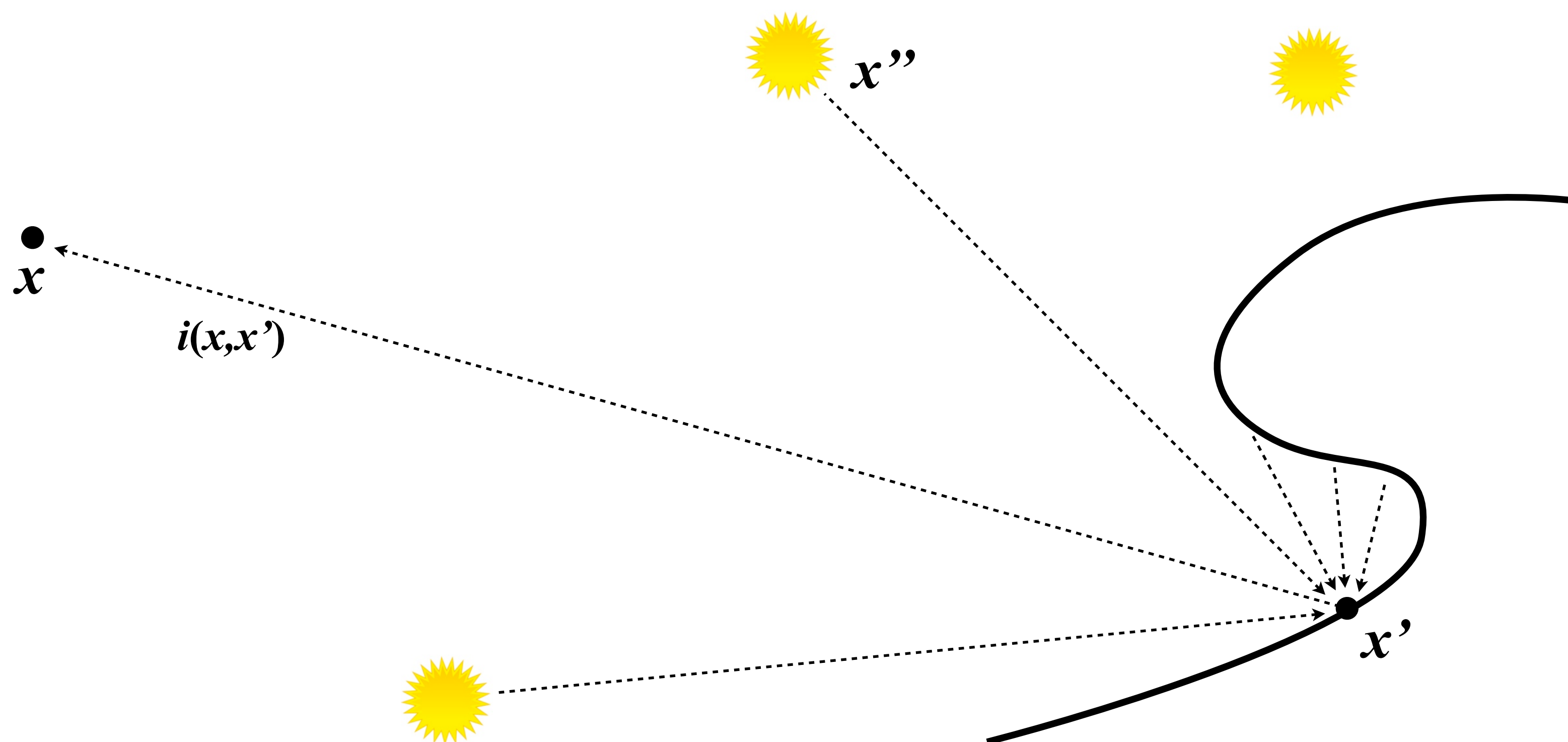
$$i(x, x') = v(x, x') \left[ l(x, x') + \int r(x, x', x'') i(x', x'') dx'' \right]$$

$i(x, x')$  = Radiance (energy along a ray) from point  $x'$  in direction of point  $x$

$v(x, x')$  = Binary visibility function (1 if ray from  $x'$  reaches  $x$ , 0 otherwise)

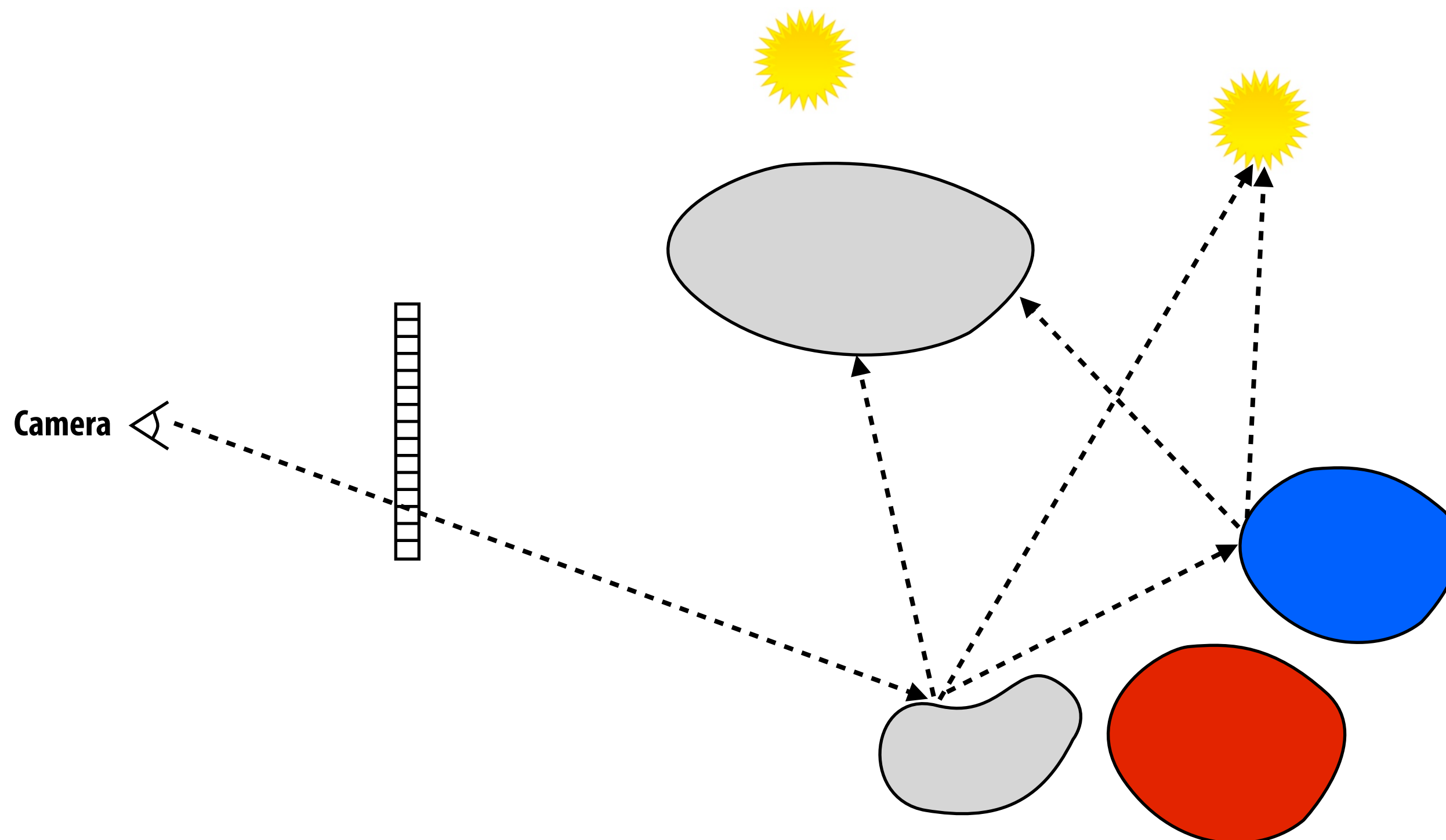
$l(x, x')$  = Radiance emitted from  $x'$  in direction of  $x$  (if  $x'$  is an emitter)

$r(x, x', x'')$  = BRDF: fraction of energy arriving at  $x'$  from  $x''$  that is reflected in direction of  $x$



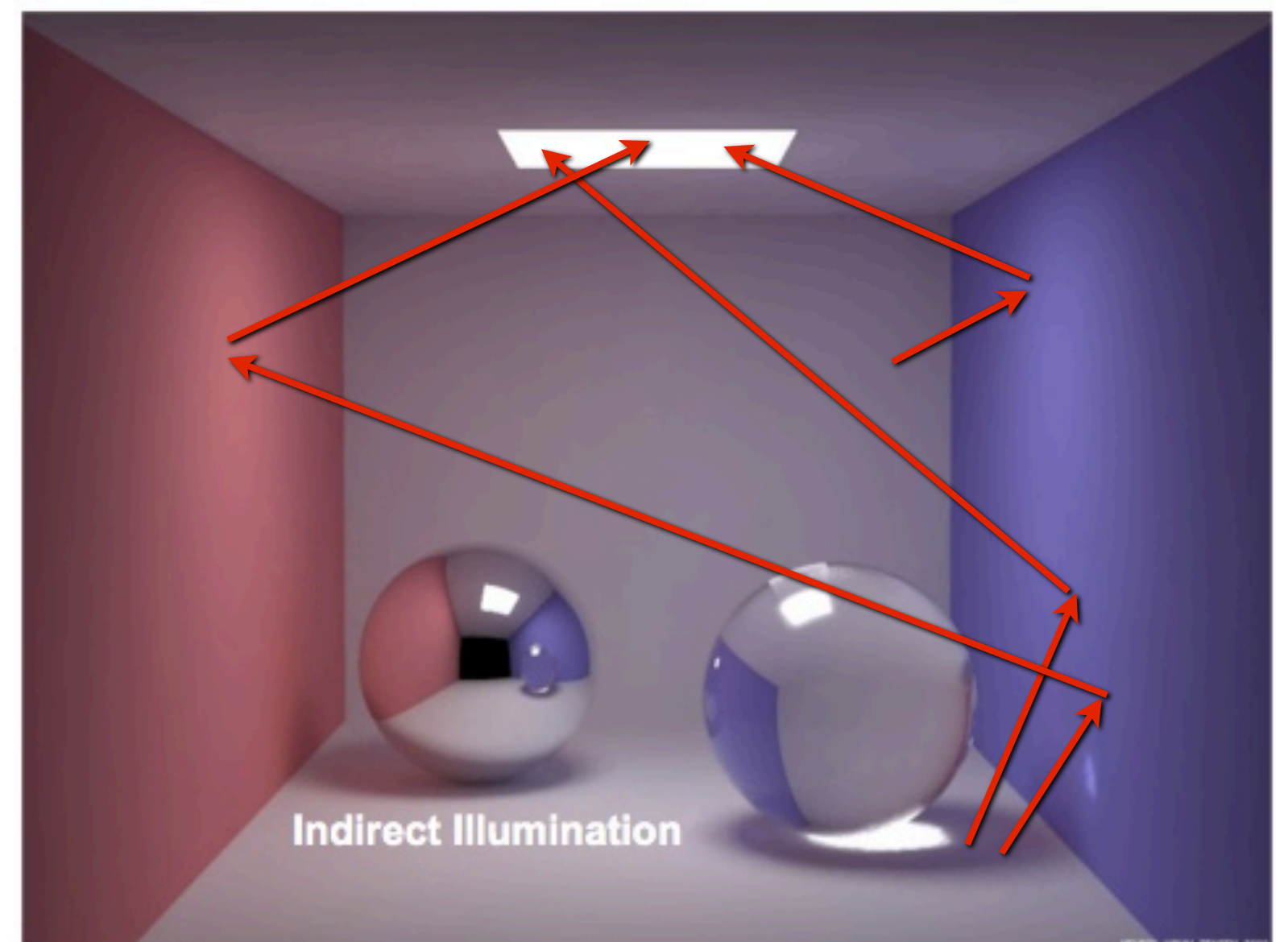
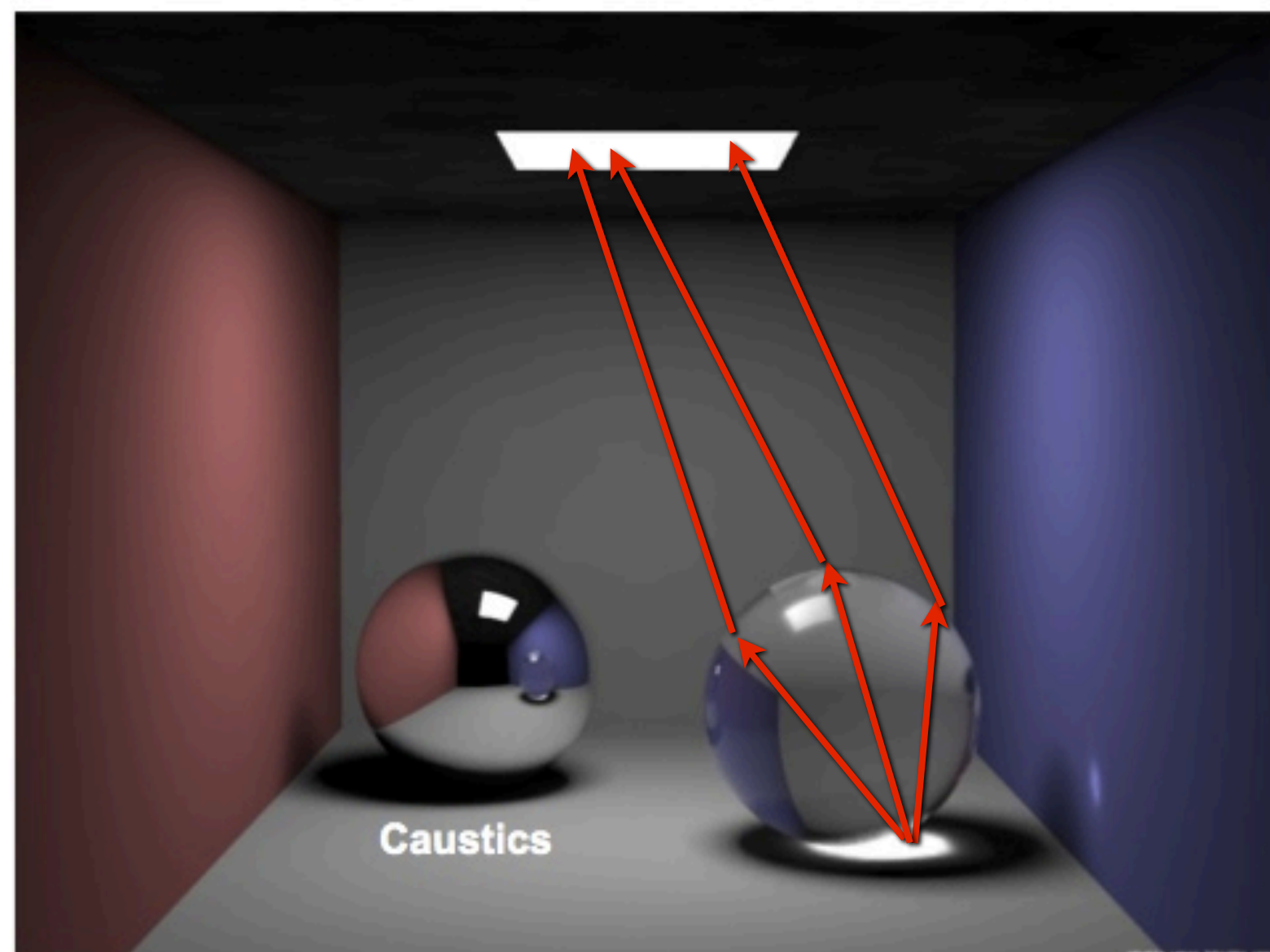
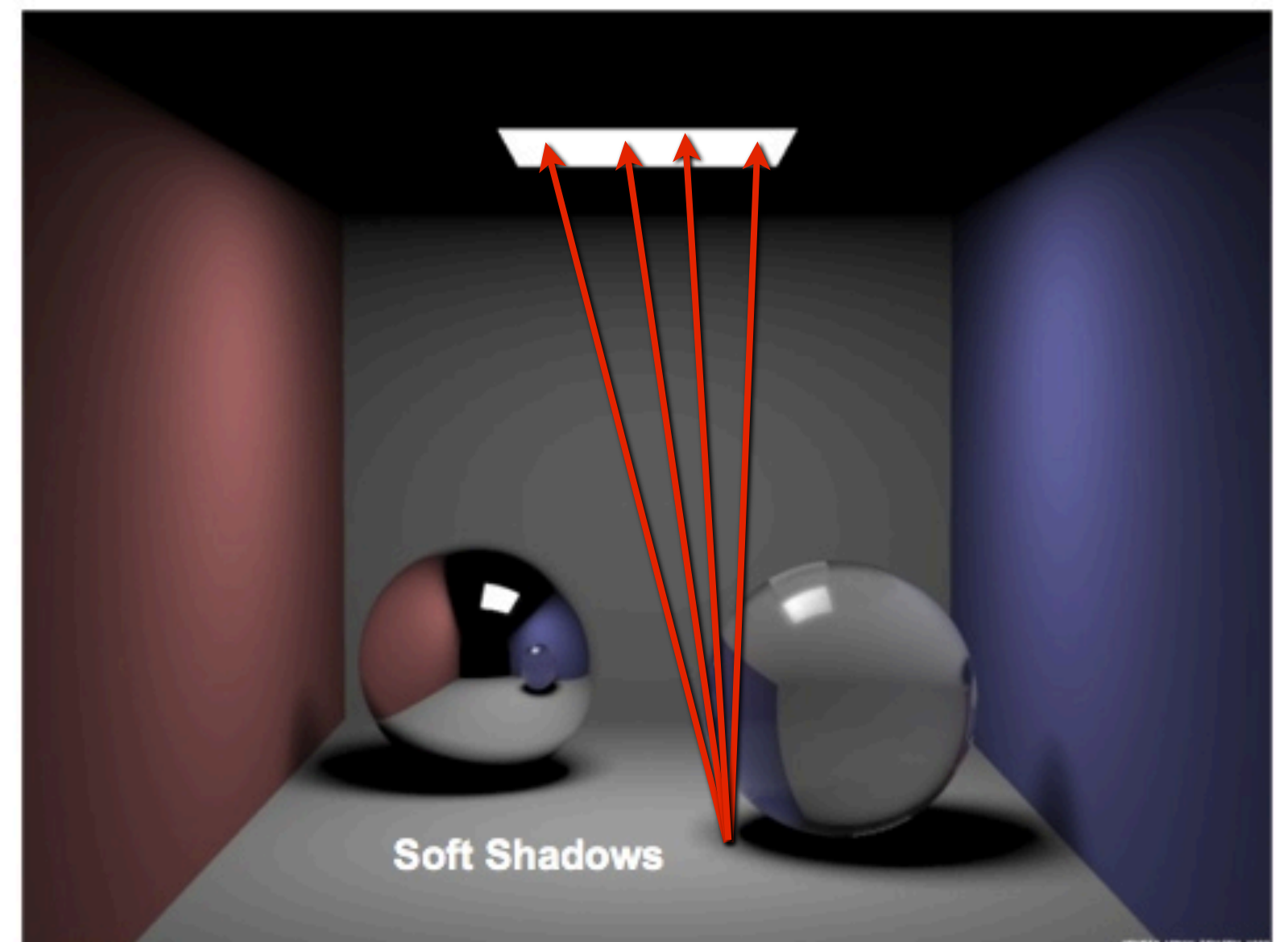
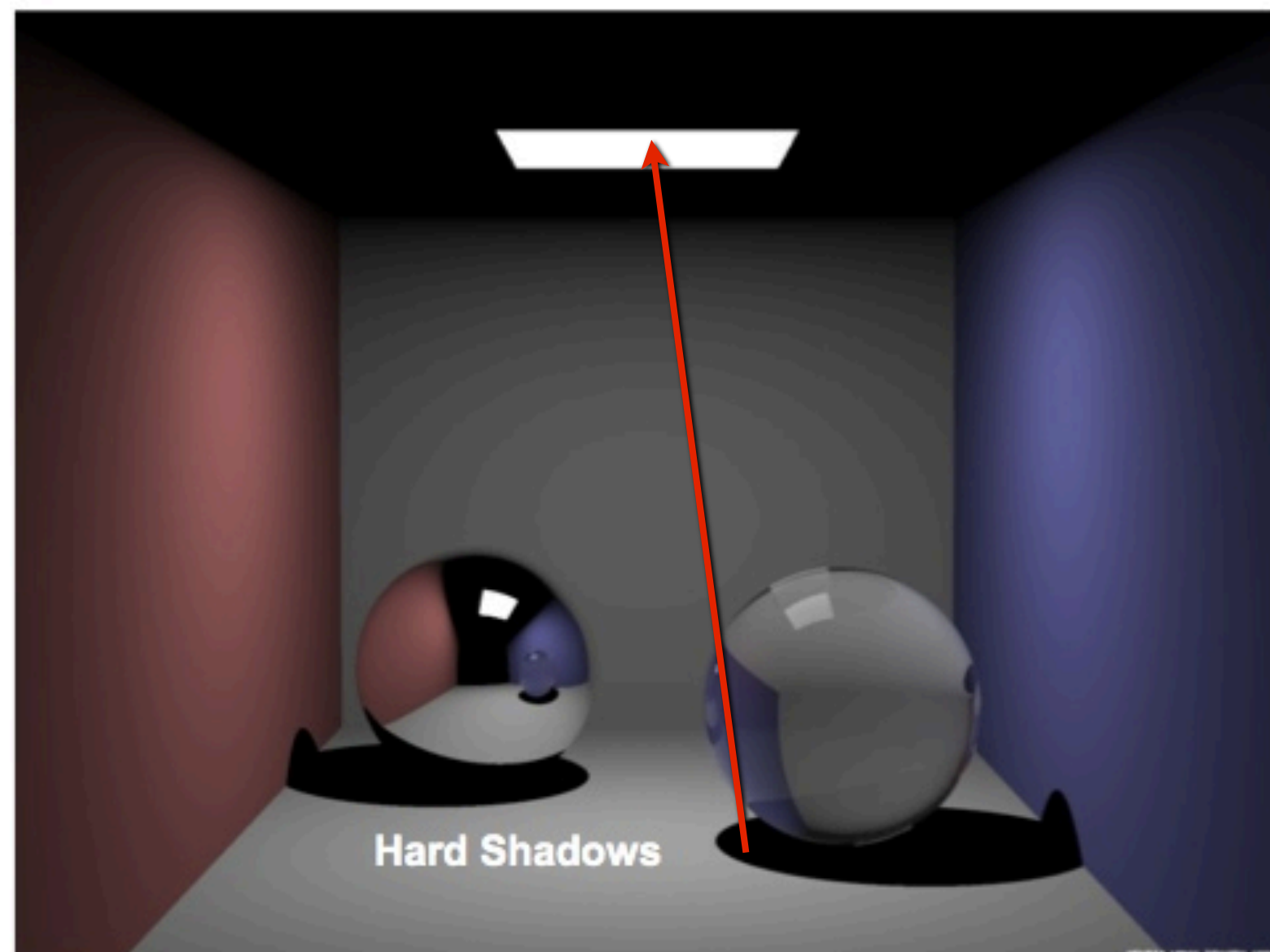
# Trace a ray to compute visibility between two scene points (key component of evaluating rendering equation)

- Compute  $v(x, x')$  (is there visibility along ray from  $x$  to  $x'$ )
- Compute  $\text{hit} = \text{trace}(x, x')$  (what surface was the first hit by ray from  $x$  to  $x'$ )





# Sampling light paths





# Tracing rays used in many contexts

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

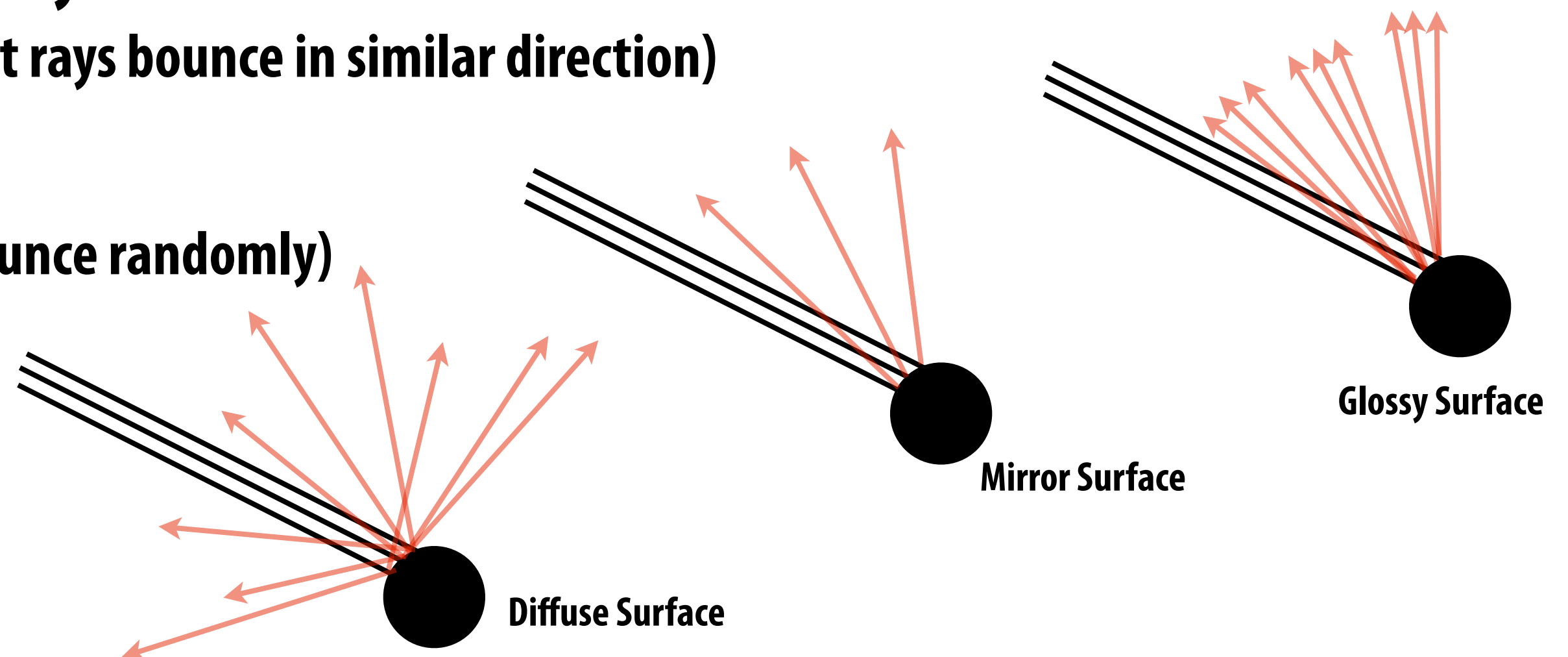
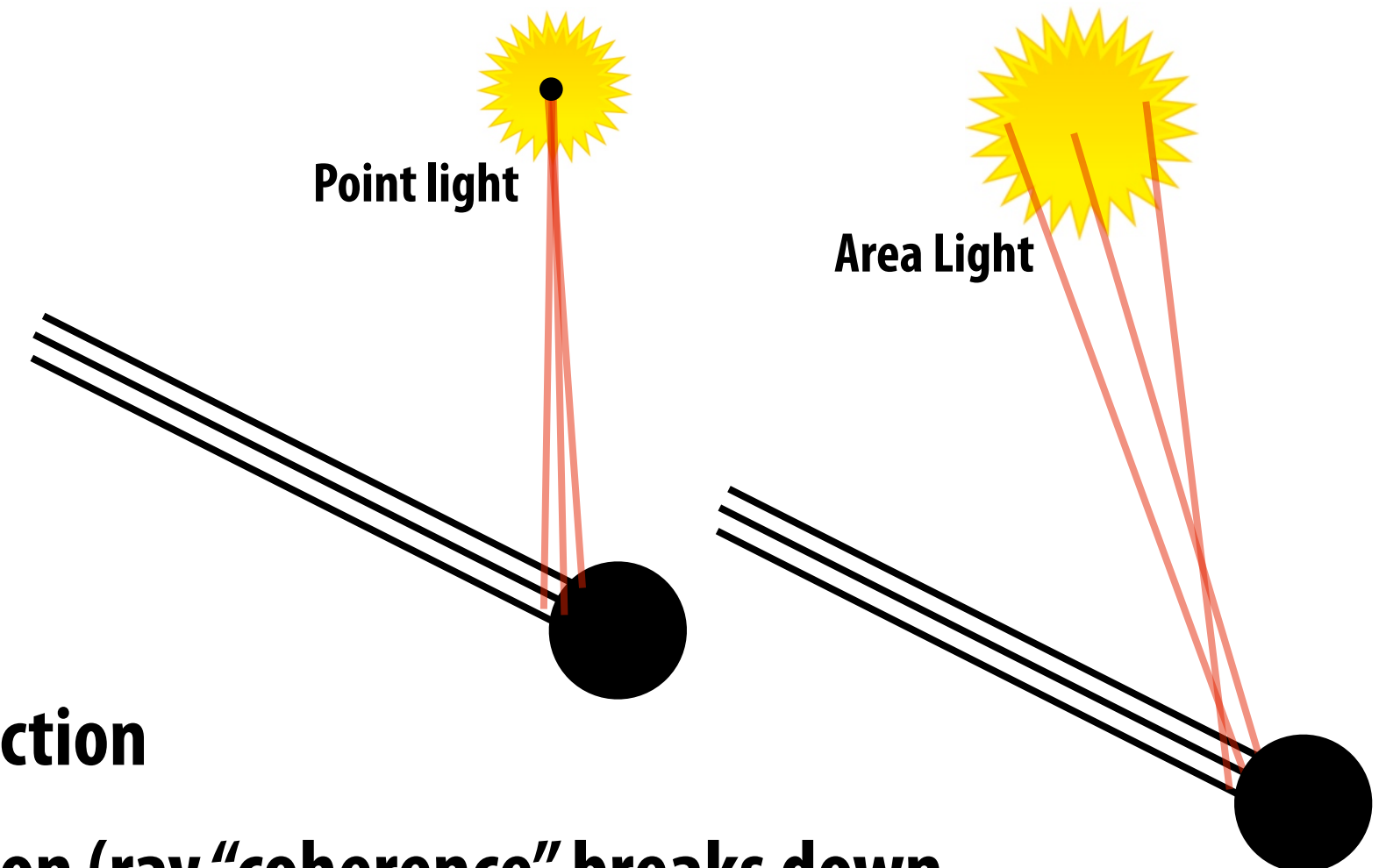
- Common origin, similar direction

## ■ 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 (within field of view)**
  
- 1. Same origin/known field-of-view: project triangles to reduce ray-triangle intersection to 2D point-in-polygon test**
  - **Simplifies math (2D coverage computations rather than 3D interaction)**
  - **Fixed-point math (clipping used to ensures precision bounds)**

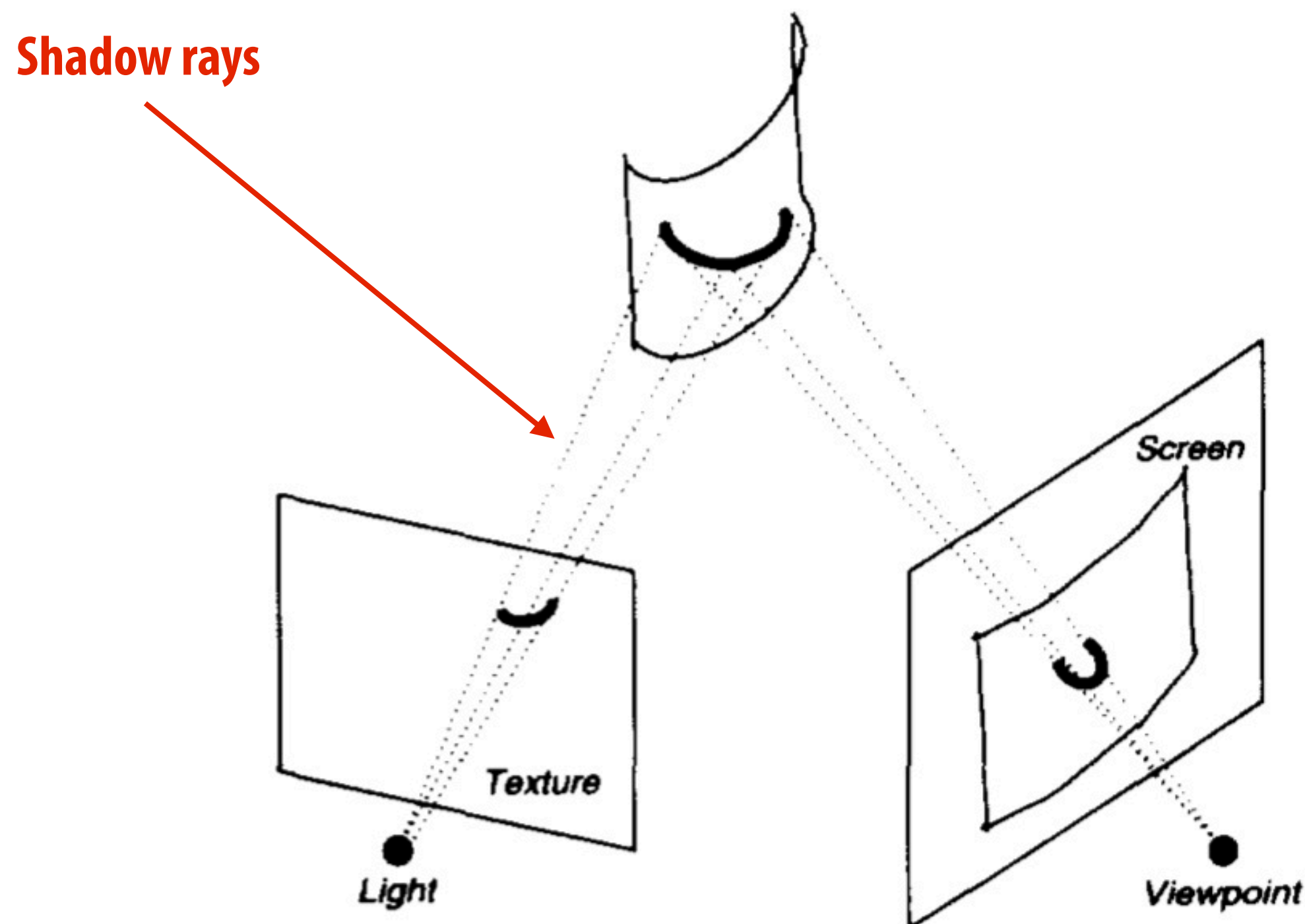
# Rasterization: ray origin need not be camera position

## Example: shadow mapping

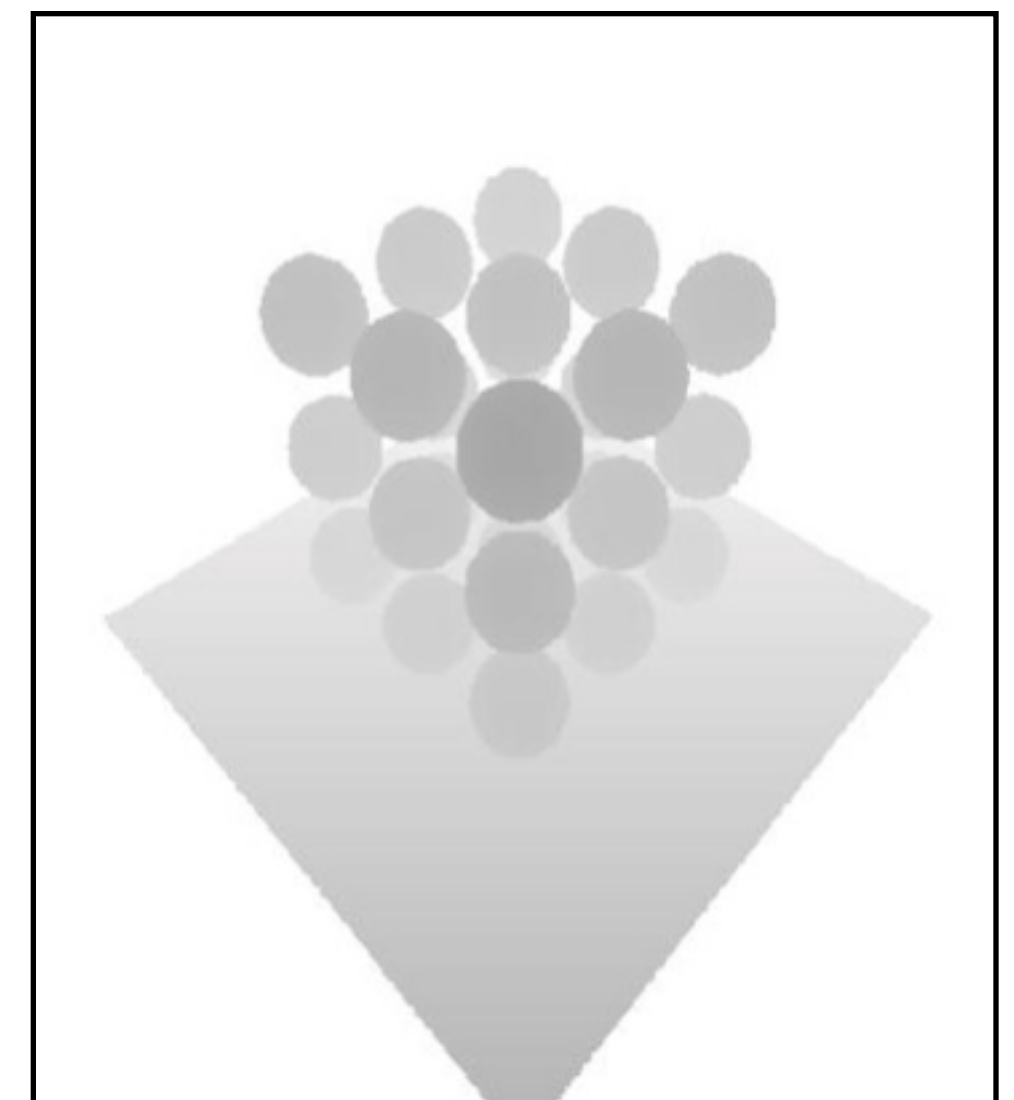
- Place ray origin at position of light source

Shadow map: Render scene with camera at light position to compute depth along uniformly distributed “shadow rays”

- Store precomputed shadow rays in texture map

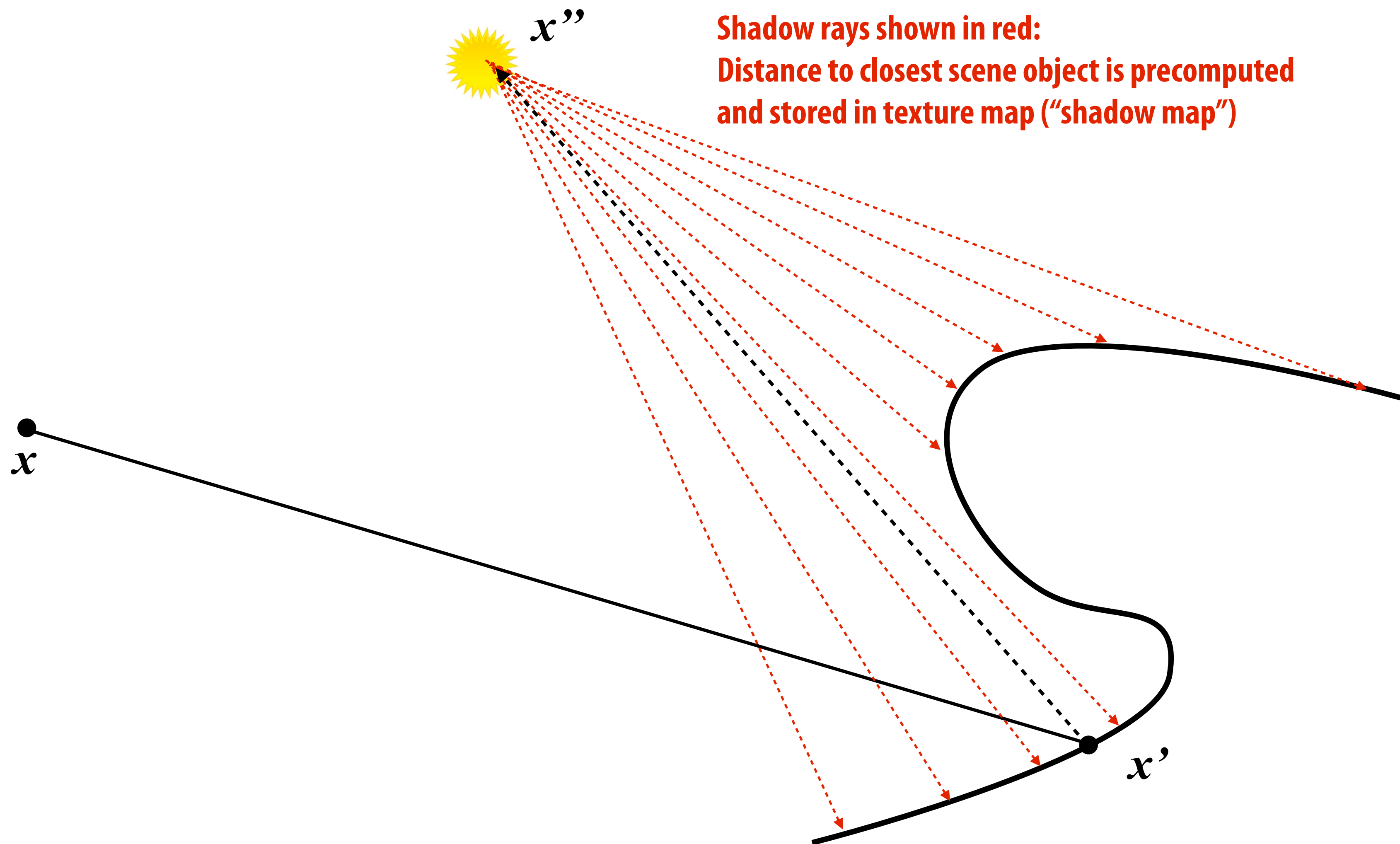


Shadow map: texture stores closest intersection along each shadow ray

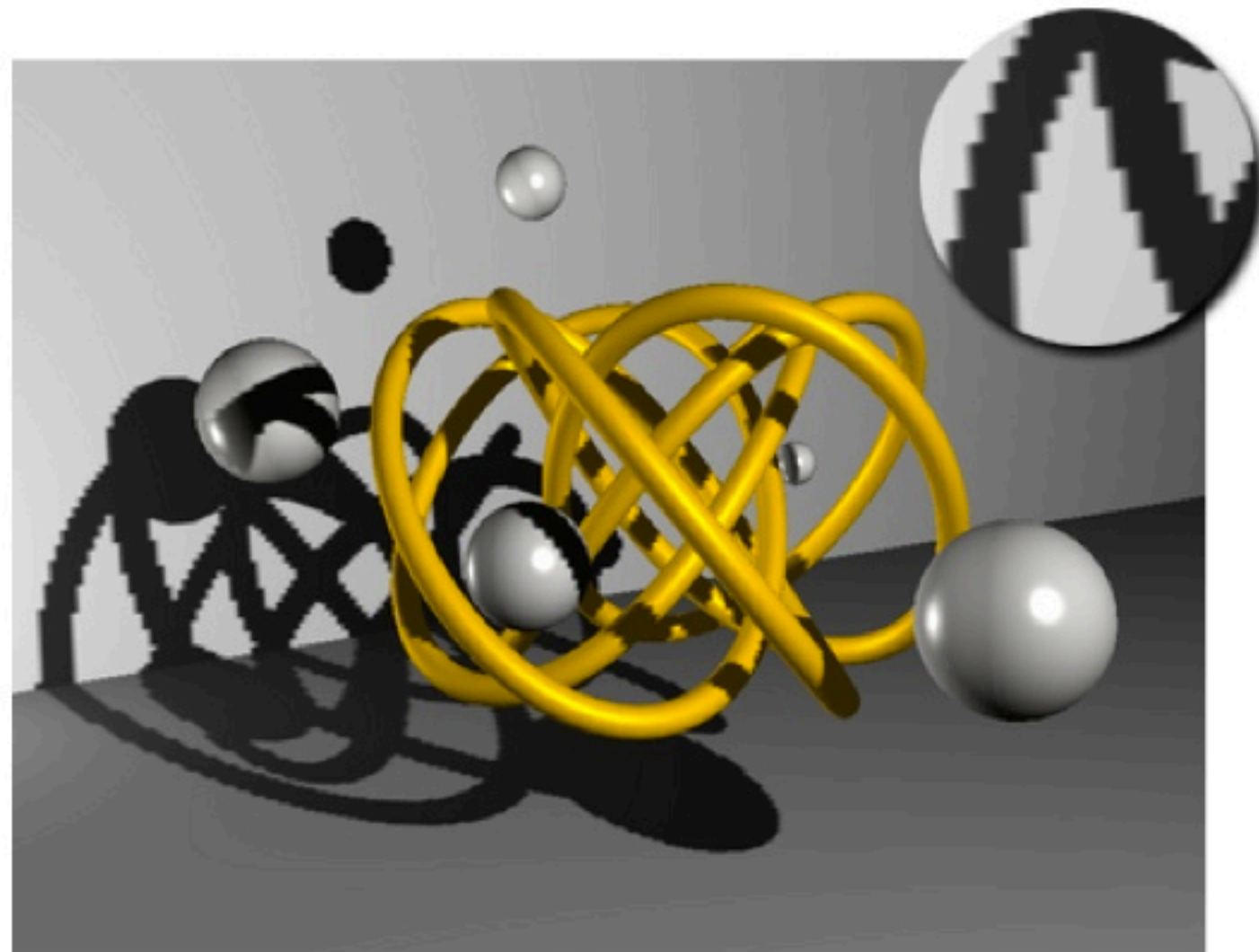




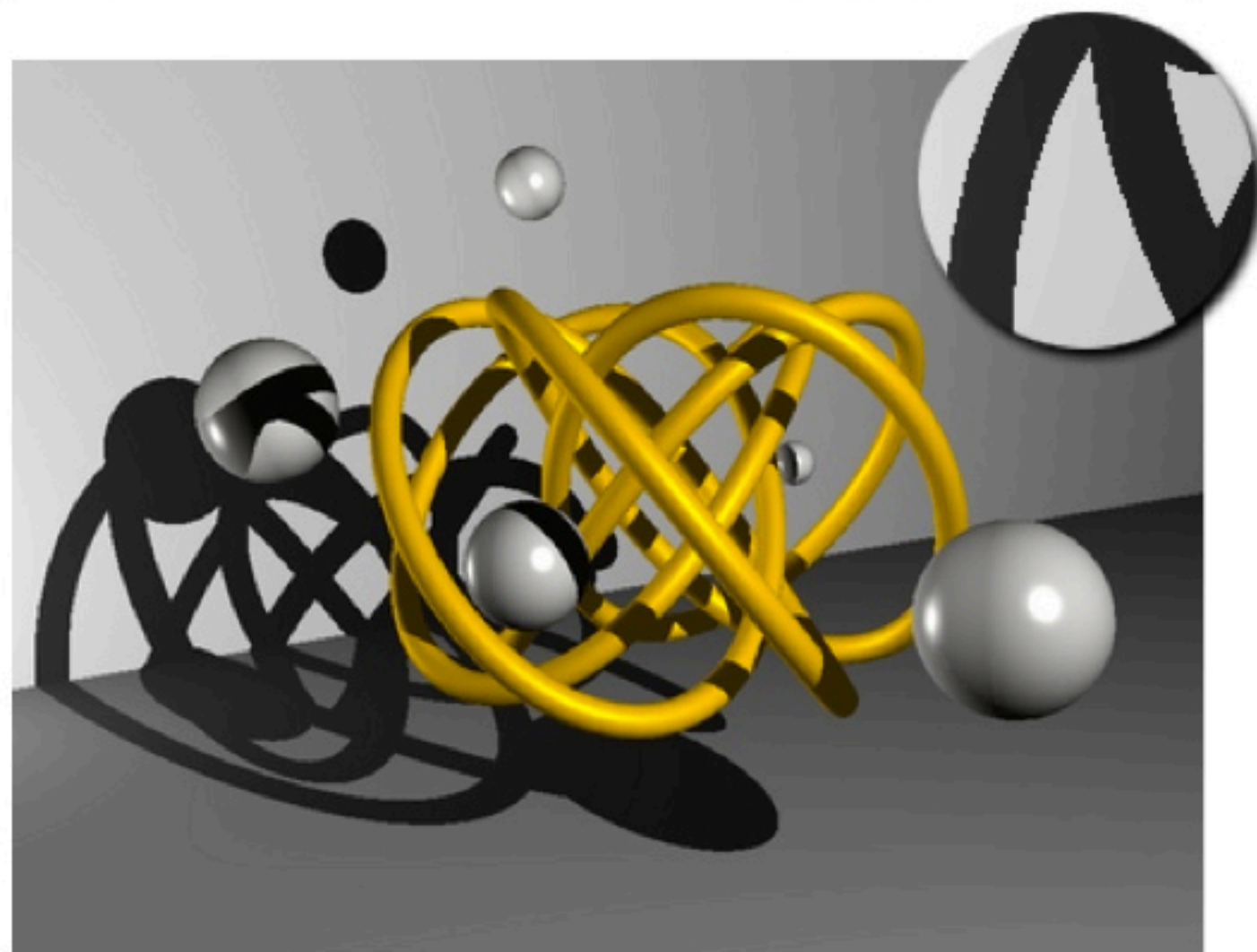
# Shadow map used to approximate $v(x', x'')$ when shading fragment at $x'$



# Shadow aliasing due to shadow map undersampling



**Shadows computed using shadow map**



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

# Rasterization: ray origin need not be camera position

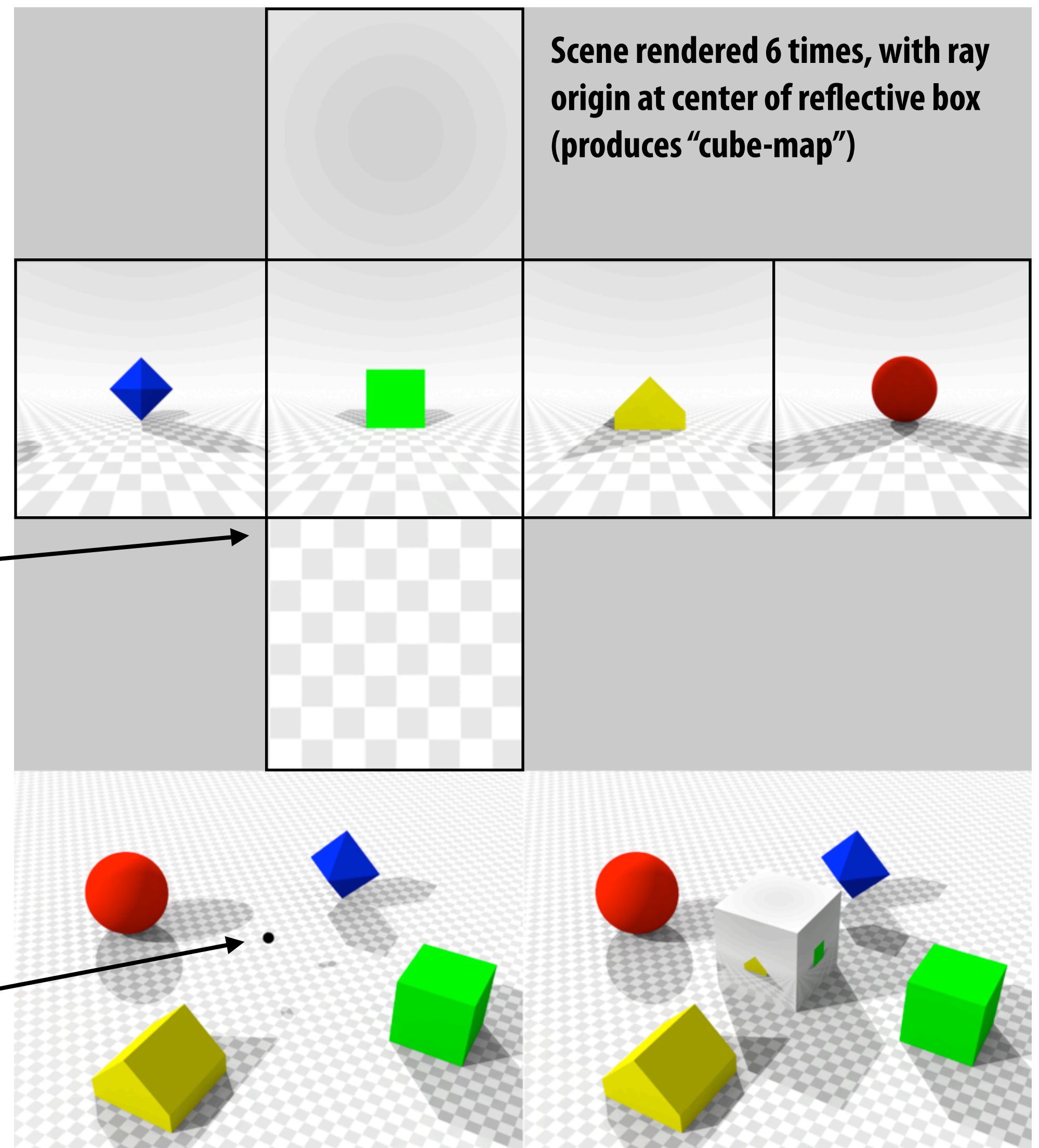
**Environment mapping:**  
place ray origin at reflective object

**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 within field of view
- 1. Same origin/known field of field: project triangles to reduce ray-triangle intersection to cheap/efficient 2D point in polygon test
- 2. Uniform sample distribution: given polygon, easy (a.k.a. fast/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 (hierarchy implicit in the samples)

# Rasterization: performance

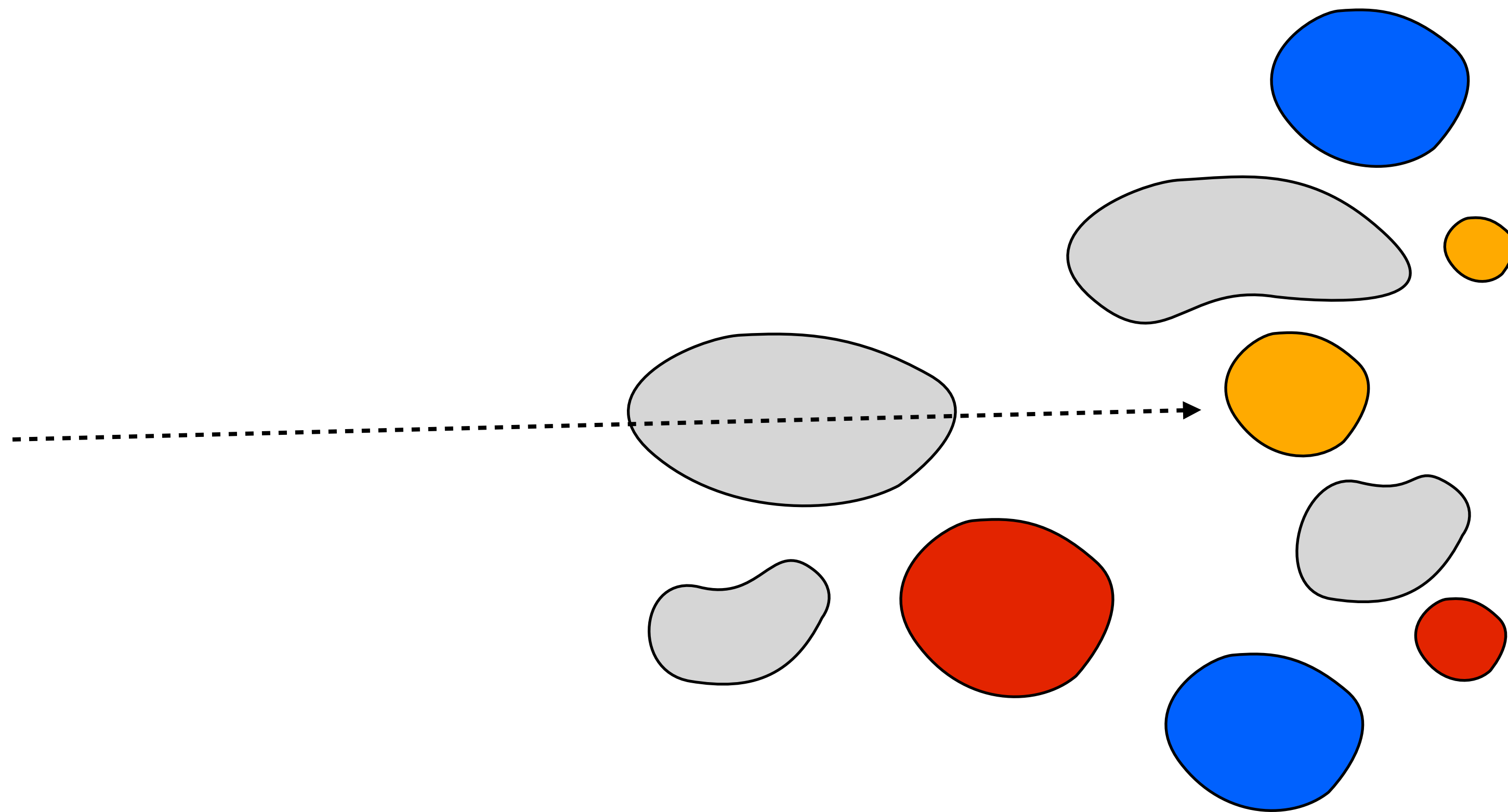
- **Stream over scene geometry (regular/predictable data access), but arbitrarily access frame-buffer sample data**
  - Unpredictable access to sample data is manageable
- **Main idea: Z-buffered occlusion**
  - Fixed number of samples (determined by screen resolution, sampling rate)
  - Known sample 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 scene size

# **Review: ray tracing 101**



# Problem

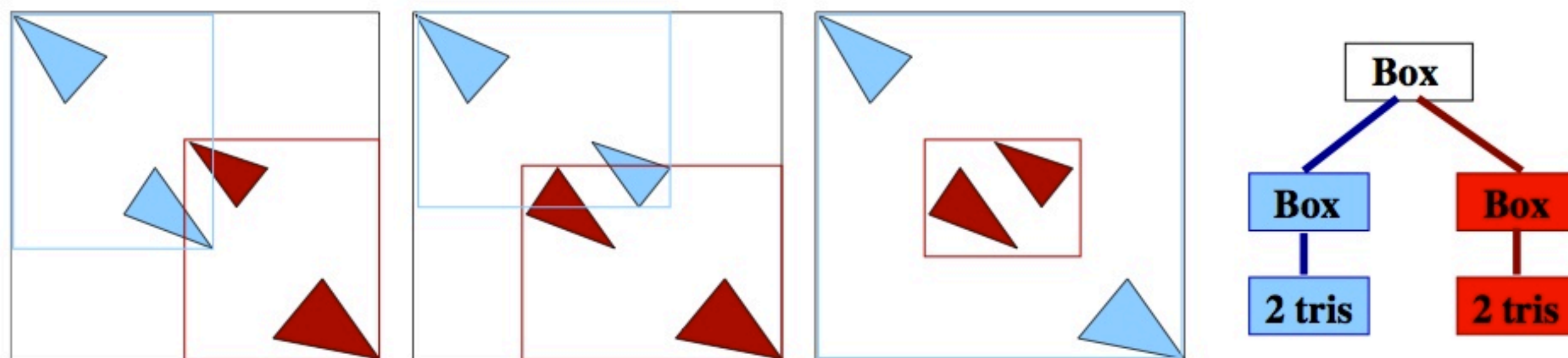
**Given ray, find first intersection with scene geometry \*\***



**\*\* A simpler, but common, query is to determine only if any intersection exists**

# Accelerating ray-scene intersection

- Preprocess scene to accelerate ray-scene visibility queries
  - 1D analogy: sort integers in a list to enable efficient binary search
  - Database analogy: build an index (e.g., B-tree)
- Popular acceleration structure for ray tracing: bounding volume hierarchy (BVH)
  - Group objects with spatial proximity into tree nodes
  - Adapts to non-uniform density of scene objects
  - Note: many other acceleration structures: K-D trees, octrees, uniform grids



Three different bounding volume hierarchies for the same scene

# Simple ray tracer (using a BVH)

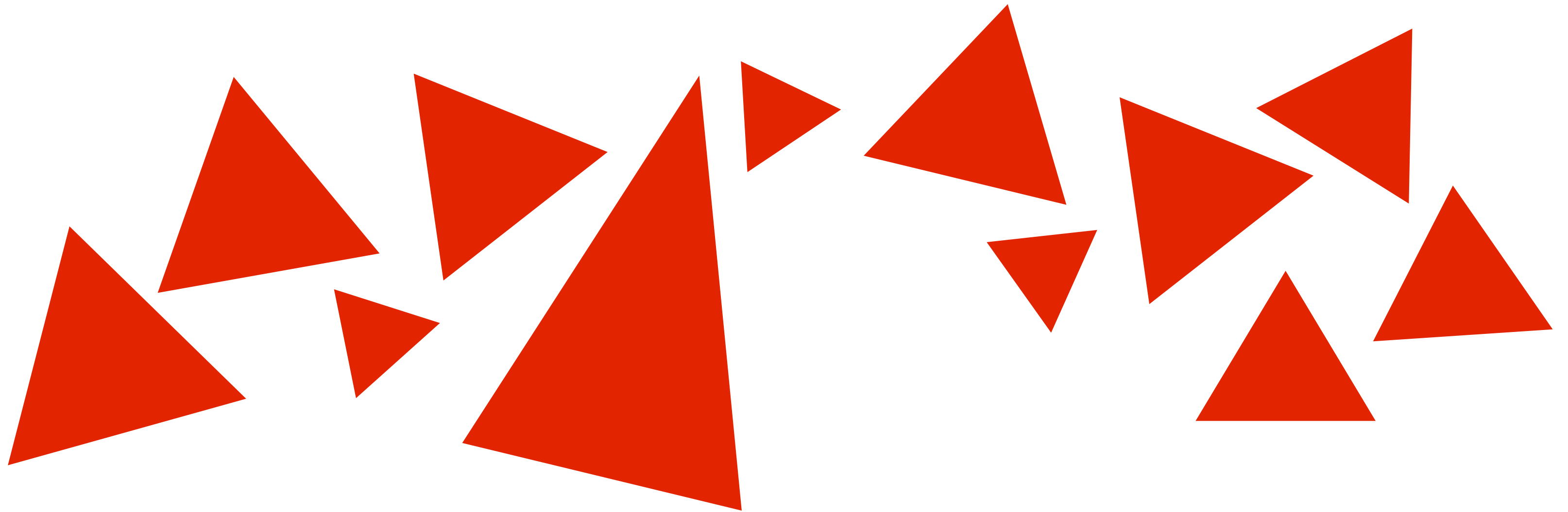
```
// stores information about closest hit found so far
struct ClosestHitInfo {
    Primitive primitive;
    float distance;
};

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

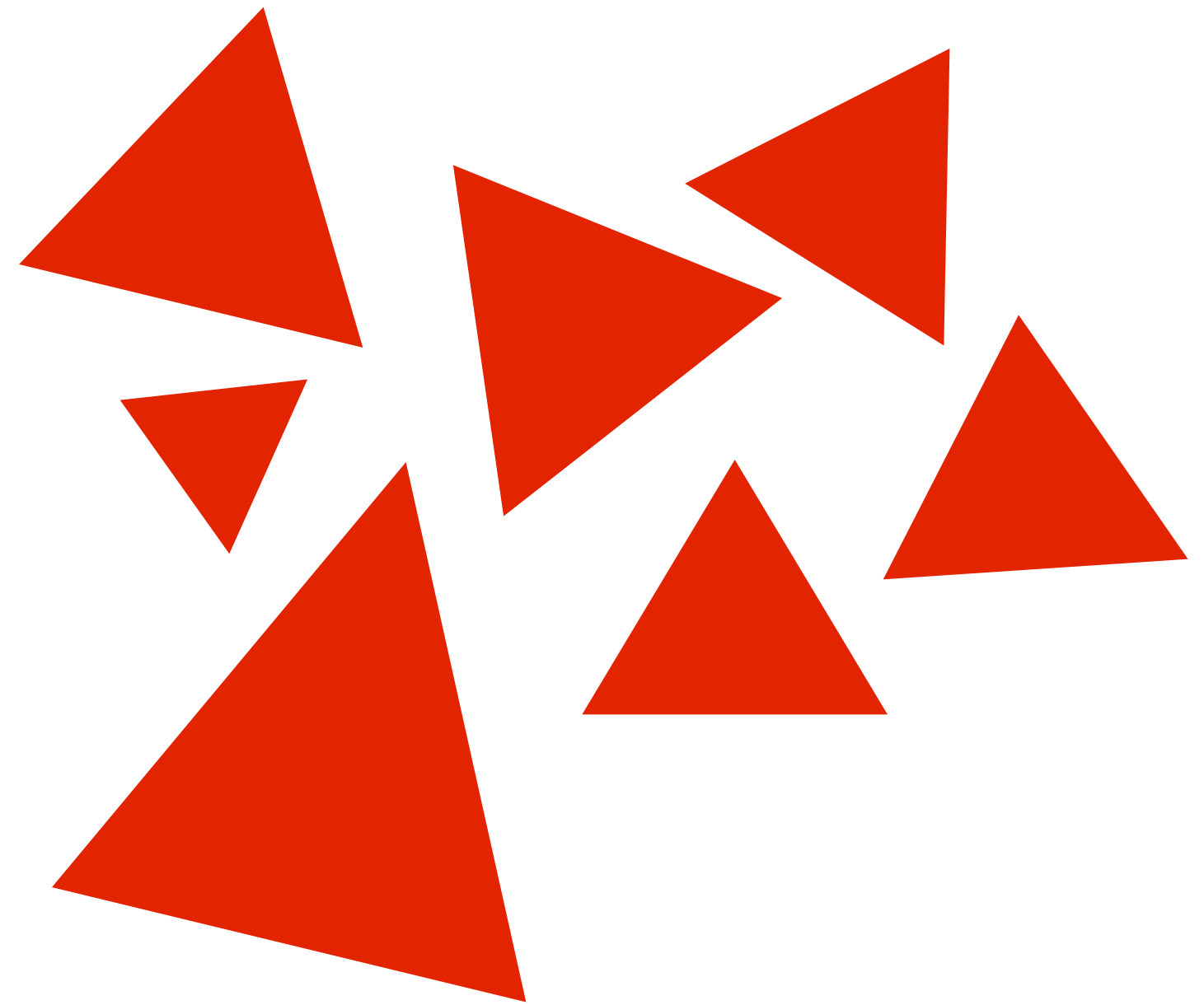
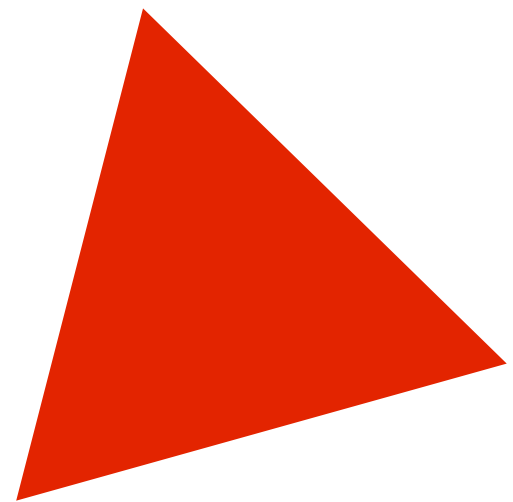
    if (node.leaf) {
        for (each primitive in node) {
            (hit, distance) = intersect(ray, primitive);
            if (hit && distance < hitInfo.distance) {
                hitInfo.primitive = primitive;
                hitInfo.distance = distance;
            }
        }
    } else {
    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



# How to build a BVH?



# How to build a BVH?



# Surface area heuristic

[Goldsmith and Salmon 87]

## ■ Goal: minimize expected cost to trace rays

$$\text{cost} = C_T + (P_L * C_L) + (P_R * C_R)$$

$C_T$  = cost of performing a tree node traversal (ray-box test)

$P_L/P_R$  = probability of ray intersecting left/right child

$C_L/C_R$  = cost of intersecting ray with left/right child

## ■ Assumptions:

- Rays are uniformly distributed (uniform distribution of origin and direction) but originate from outside node bounding box
  - Then  $P_i$  is surface area of child node bbox / surface area of parent node bbox
- Costs of children typically set to be  $C_I * \# \text{ primitives}$



# Accelerating ray-scene queries using a BVH

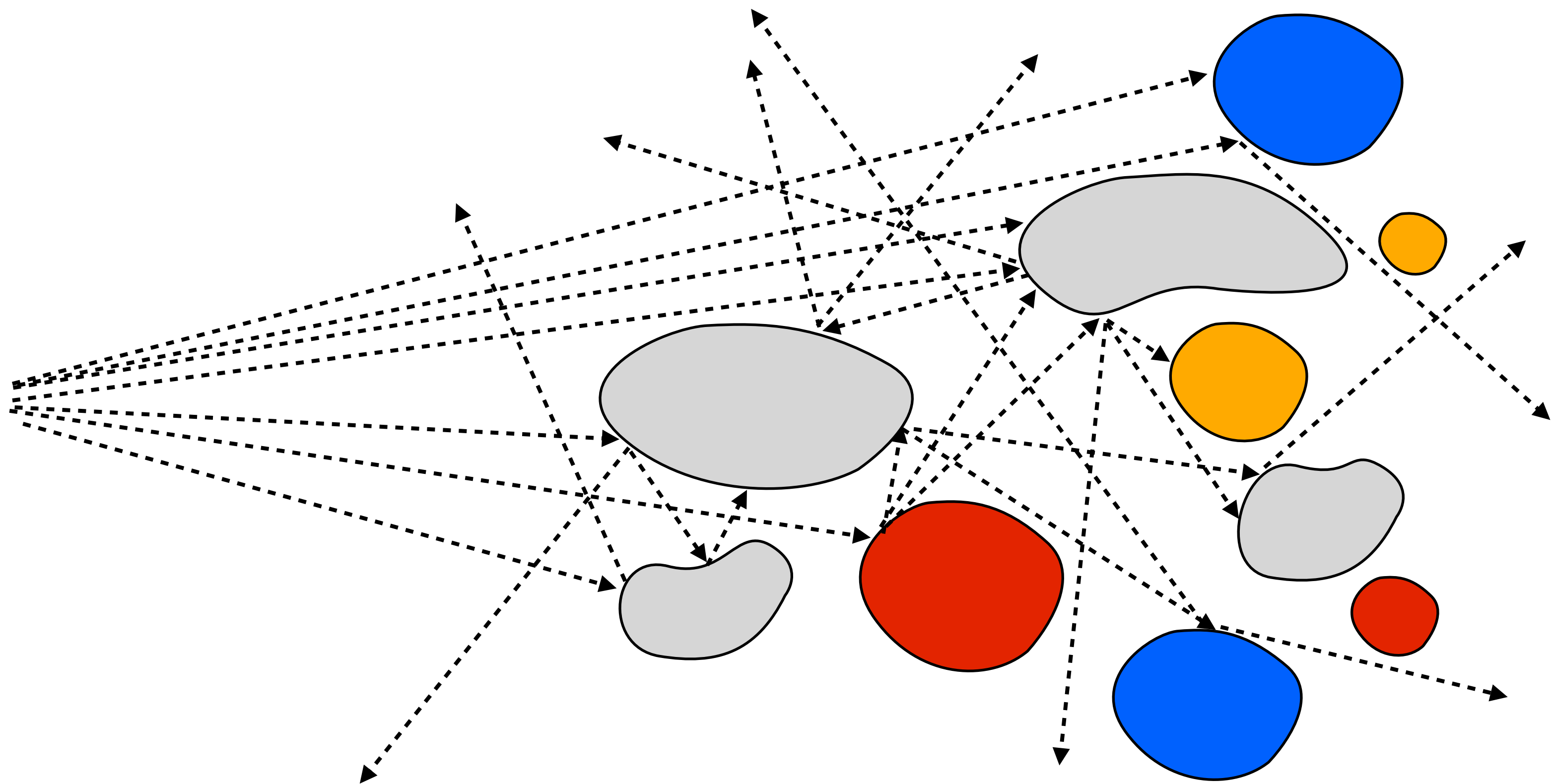
**Simplifications in today's discussion:**

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

**Scene acceleration structure is read-only: no on-demand build, no on-demand tessellation**

# High-throughput ray tracing

Find intersection of millions of rays with scene geometry



# 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.**

# Parallelize ray-box, ray-triangle intersection

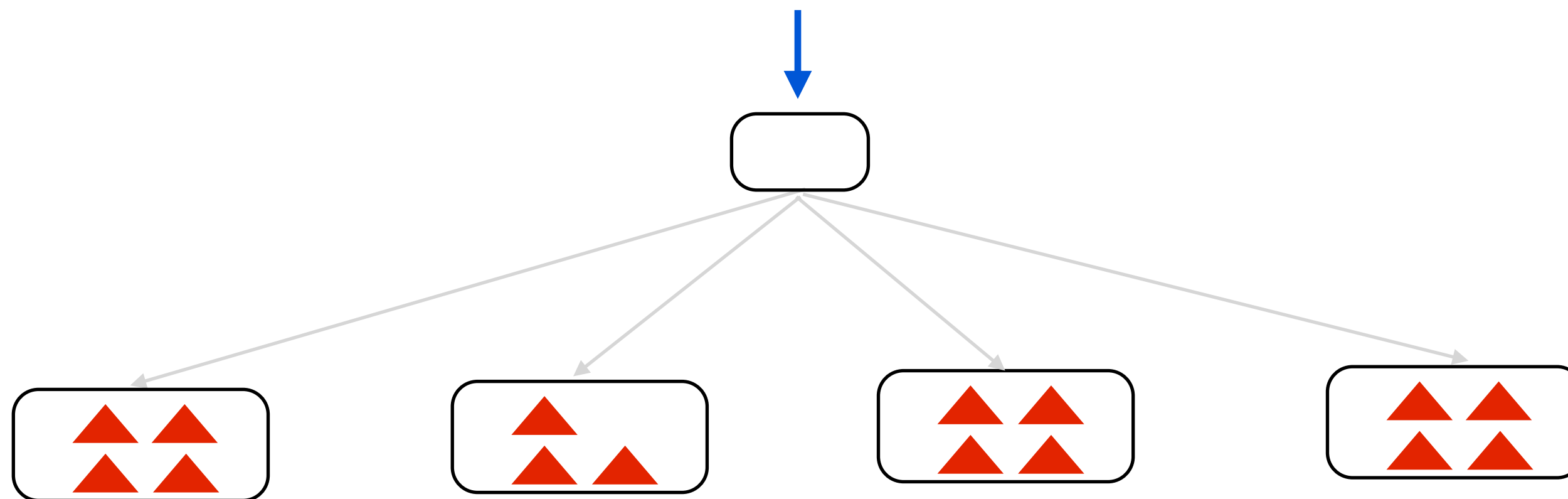
- **Given one ray and one bounding box, there are opportunities for SIMD processing**
  - Can use 3 of 4 SSE vector lanes (e.g., xyz work, point-multiple-plane tests, etc.)
- **Similar short-vector parallelism in ray-triangle test at BVH leaf**
- **If leaf nodes contain multiple triangles, can parallelize ray-triangle intersection across these triangles**



# Parallelize over BVH child nodes

[Wald et al. 2008]

- **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)**



# Parallelize across rays

- **Simultaneously intersect multiple rays with scene**
- **Method 1: SPMD style**
  - Each program instance intersects one ray against scene BVH (programmer writes algorithm for tracing single ray, it is executed simultaneously on a vector-width group of rays)
  - Recall homework assignment (1D ray tracing)
    - High SIMD efficiency when program instances execute same instructions
    - Bandwidth efficient when rays in a SIMD block (“warp”) visit same BVH nodes
- **Method 2: ray packets**
  - Code is explicitly written to trace N rays at a time, not 1 ray

# Ray packet tracing

[Wald et al. 2001]

Program explicitly intersects a collection of rays against BVH at once

RayPacket

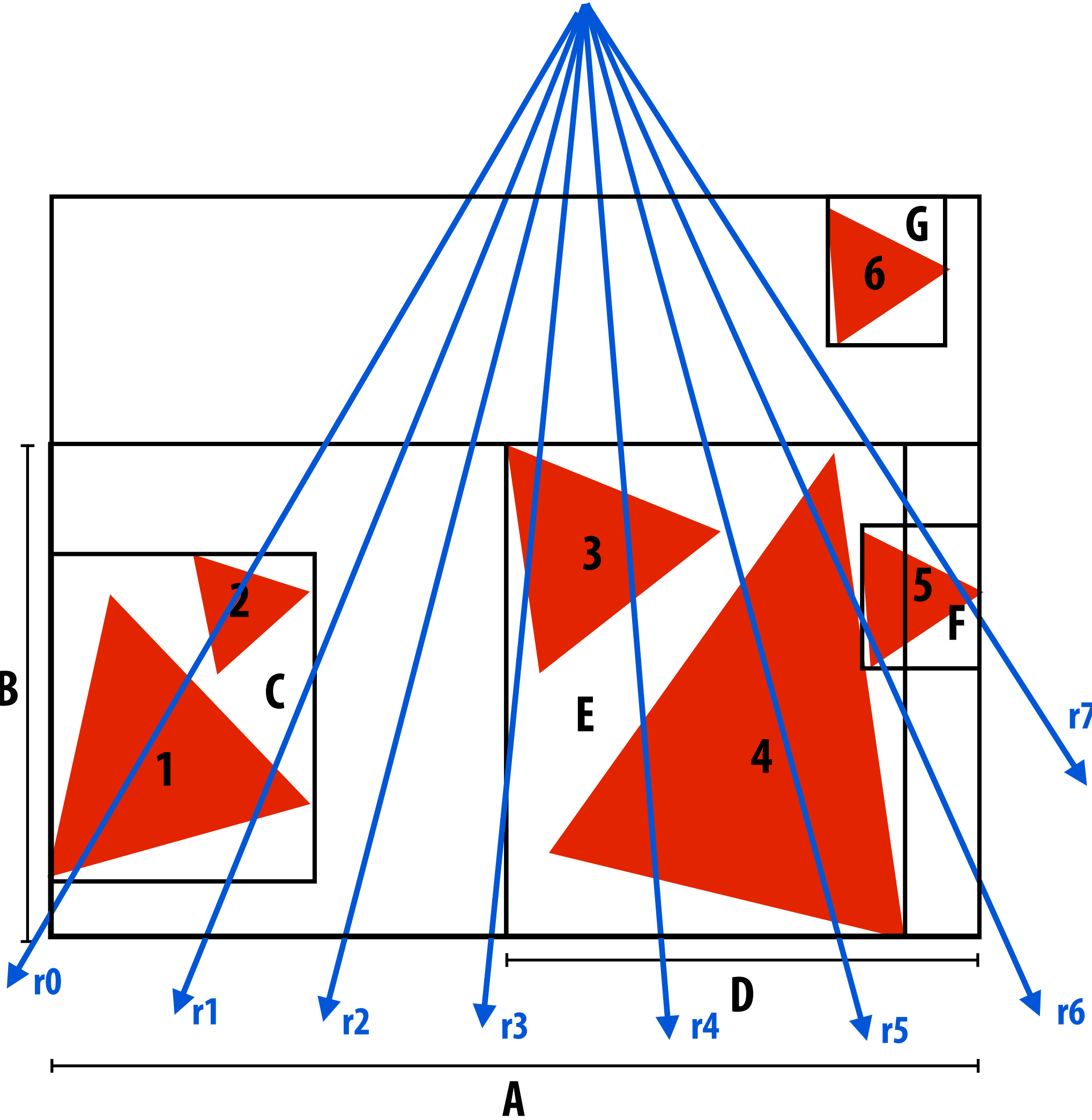
```
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

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;

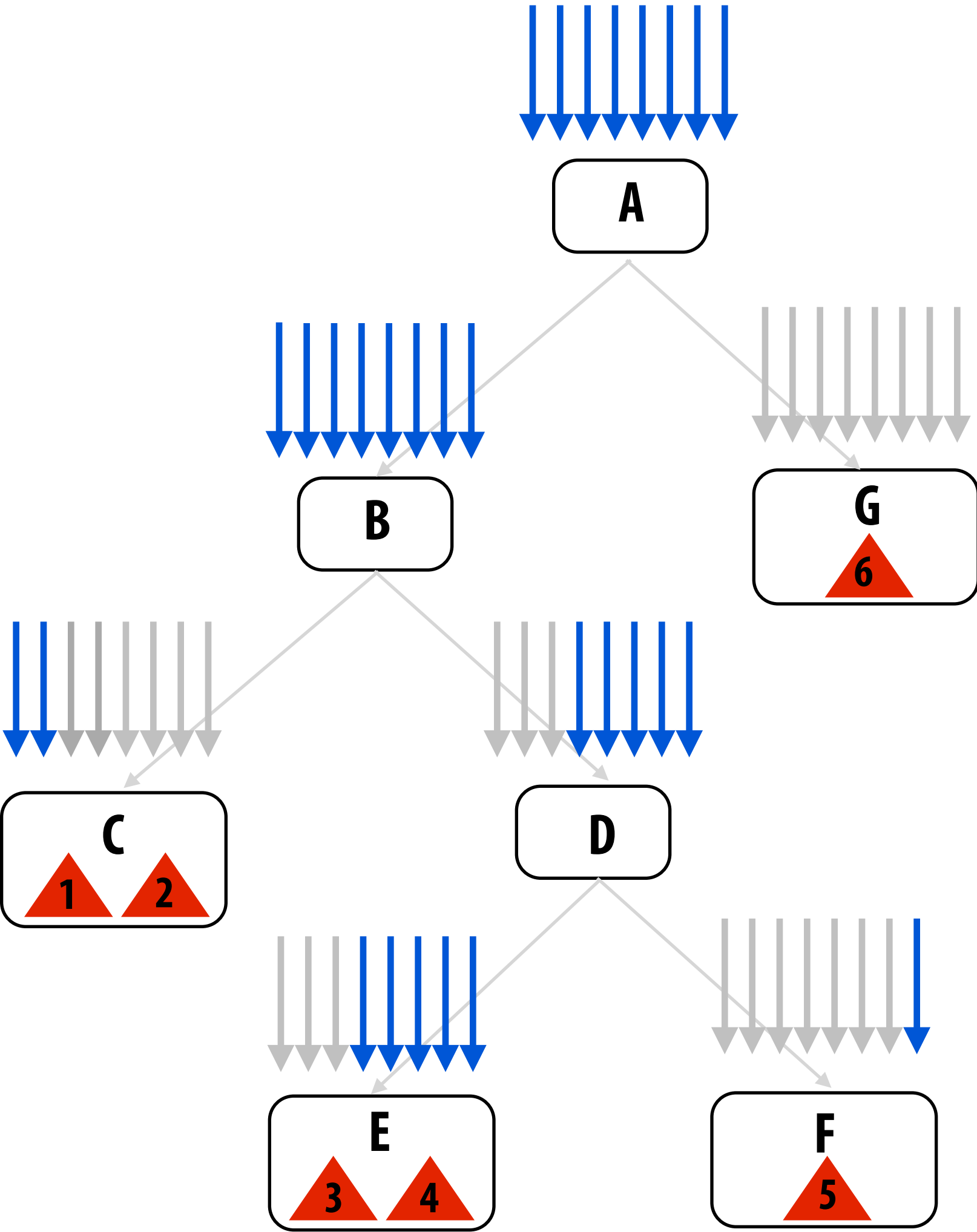
    update packet active mask

    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



# 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!**
  - Contrast with SPMD approach
- **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 math 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

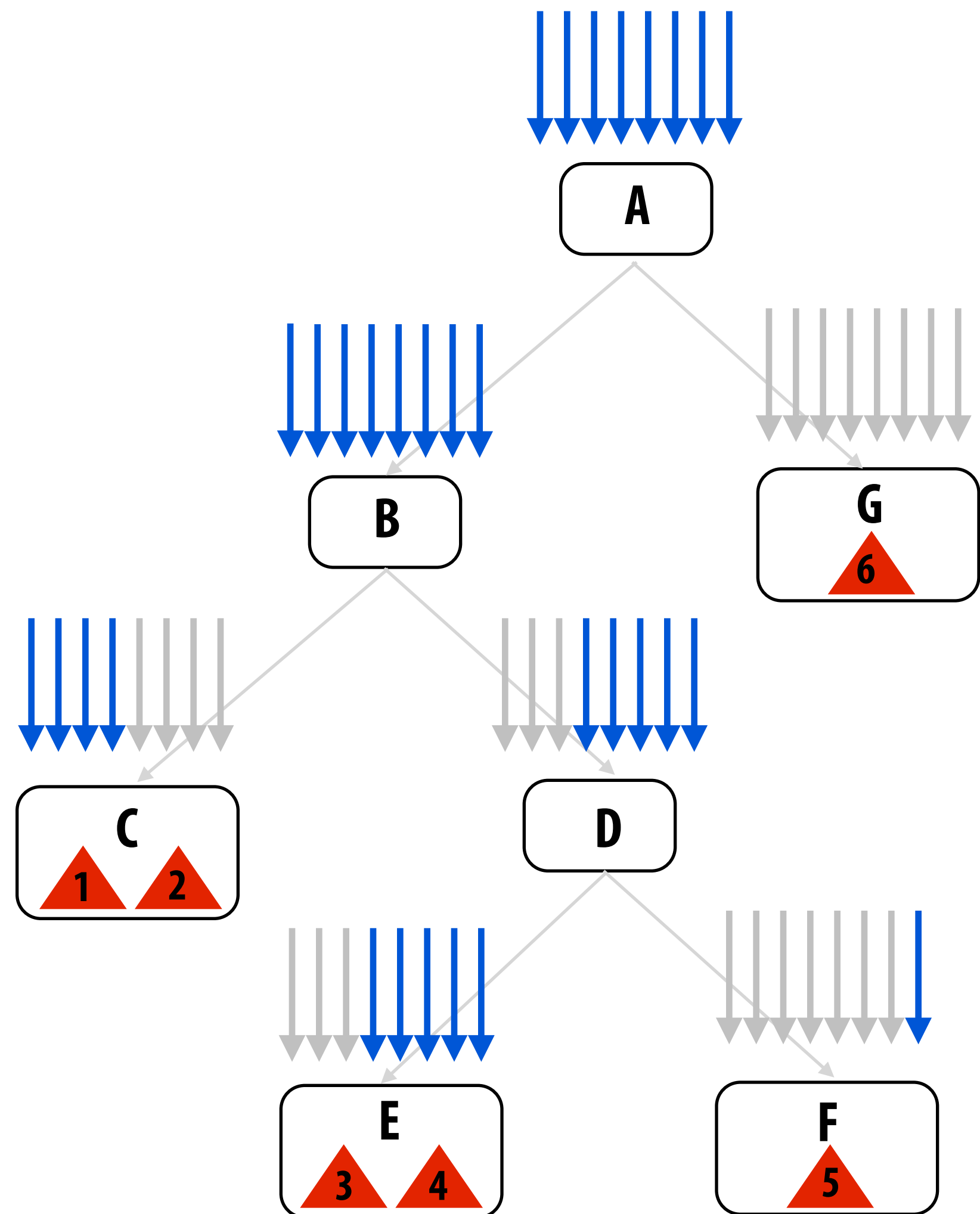
(note contrast with SPMD version: each ray only visits BVH nodes it is required to)

- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays

- Not all SIMD lanes doing useful work

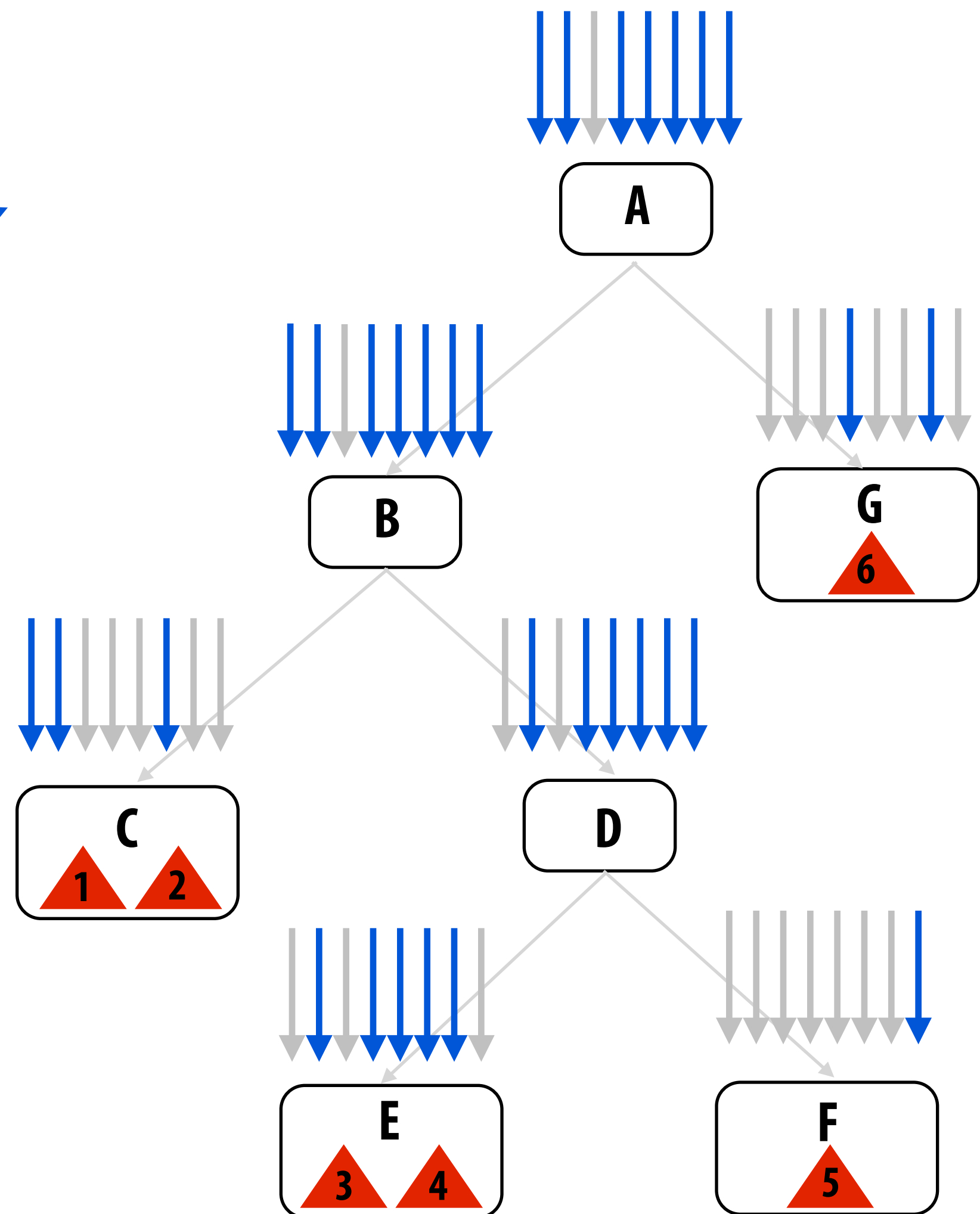
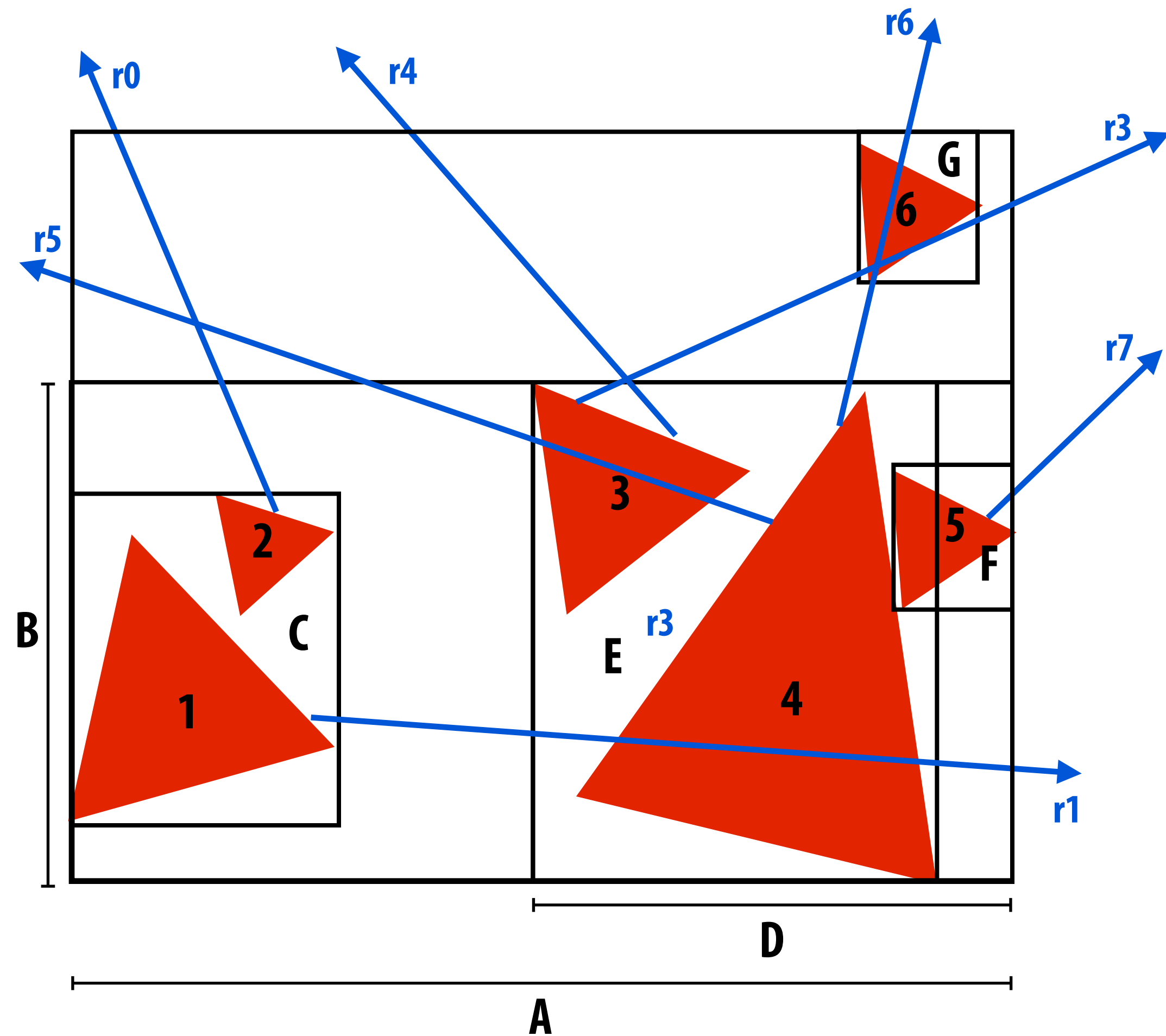
Both packet tracing and SPMD ray tracing suffer from decreased SIMD and cache efficiency when rays traverse the BVH differently... but take a moment to think about why (the reasons are different).

Blue = active ray after node box test



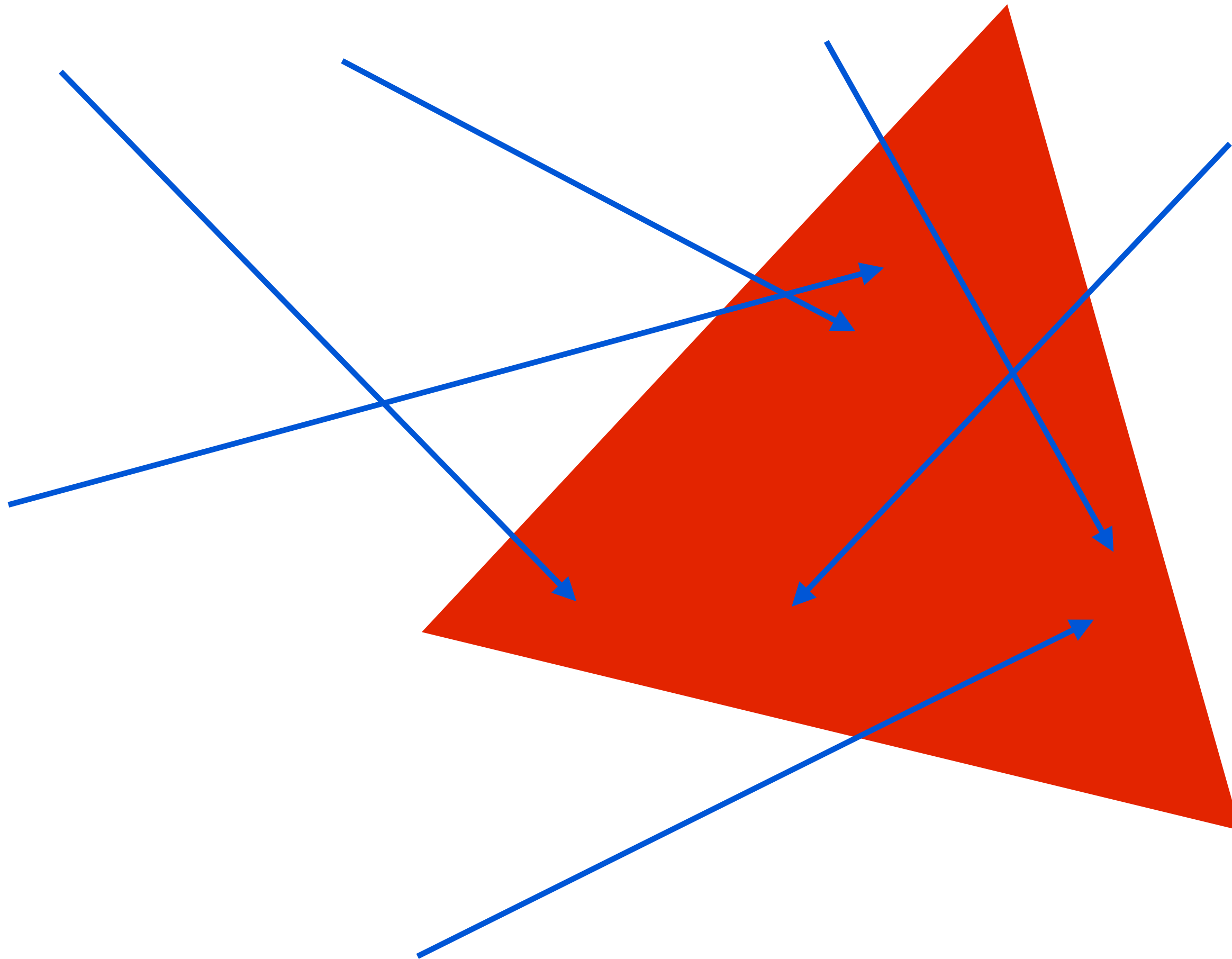
# Ray packet tracing: incoherent rays

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!)

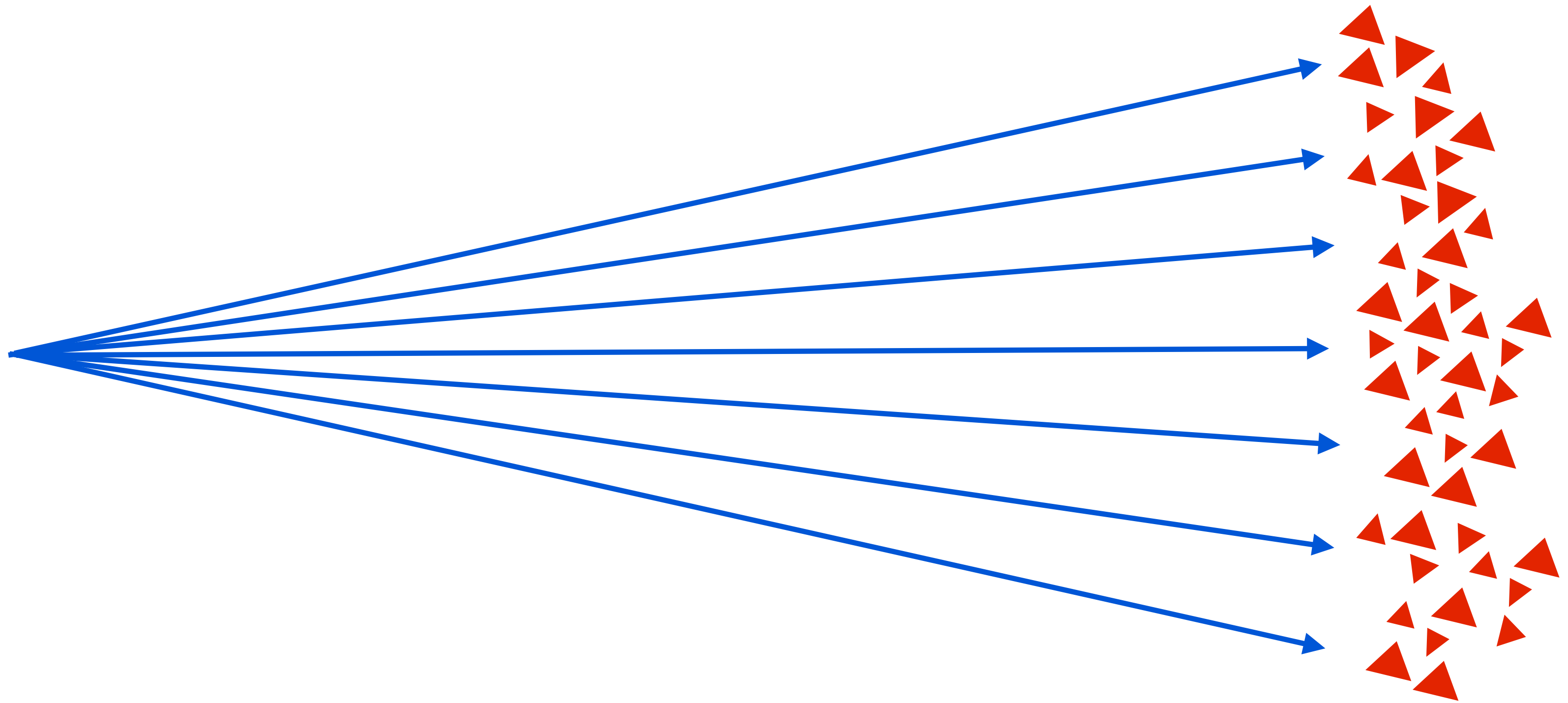
# Incoherence is a property of both the rays and the scene



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



# 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**

**(note importance of geometric level of detail)**

# Improving packet tracing with ray reordering

[Boulos et al. 2008]

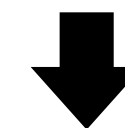
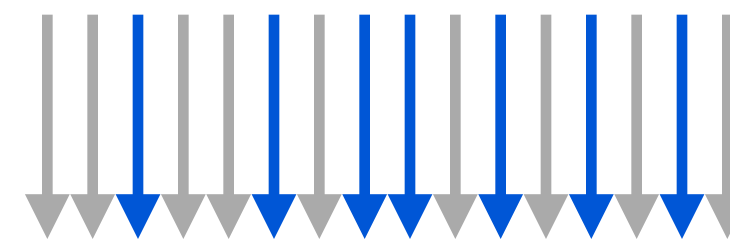
**Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet**

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

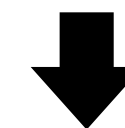
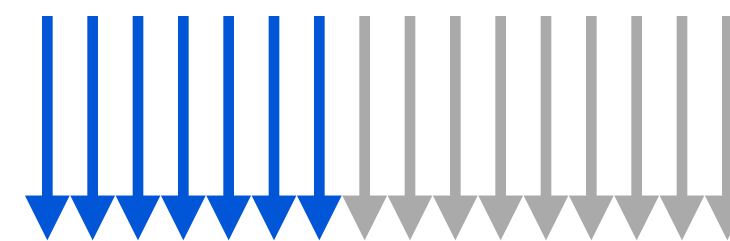
---

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

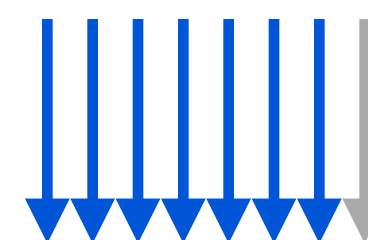
**16-ray packet: 7 of 16 rays active**



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



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



# Improving packet tracing with ray reordering

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

- Increases SIMD utilization**
- Still loses amortization benefits of large packets**

**Benefit of higher utilization/tighter packet bounds must overcome overhead of reordering operation**

---

**10-18% speedup over standard packet tracing for glossy reflection rays** [Boulos et al. 2008]  
**25-50% speedup for 2-bounce diffuse interreflection rays**  
**(4-wide SSE implementation)**

# 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**  
[Wald et al. 2007]
  - Ray coherence always high at the top of the tree  
[Benthin et al. 2011]
- **Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold**
  - For wide SIMD machine, a single branching-factor 4 BVH works well for both packet and single ray traversal
  - Recall: intra-ray SIMD provides no work amortization or bandwidth reduction benefits
- **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

# Data access challenges

## ■ Recall data access in rasterization

- Stream through scene geometry
- Allow arbitrary, direct access to frame-buffer samples (accelerated by highly specialized implementations)

## ■ Ray tracer data access

- 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)
  - Packets amortize cost of node fetches, but are less useful 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, sampling indirect illumination

# 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

## ■ Modern implementations:

- Trace single rays, not ray packets (assume most rays are incoherent rays... if they weren't there problem is a reasonable rasterization-based solution)

## ■ Two areas of focus:

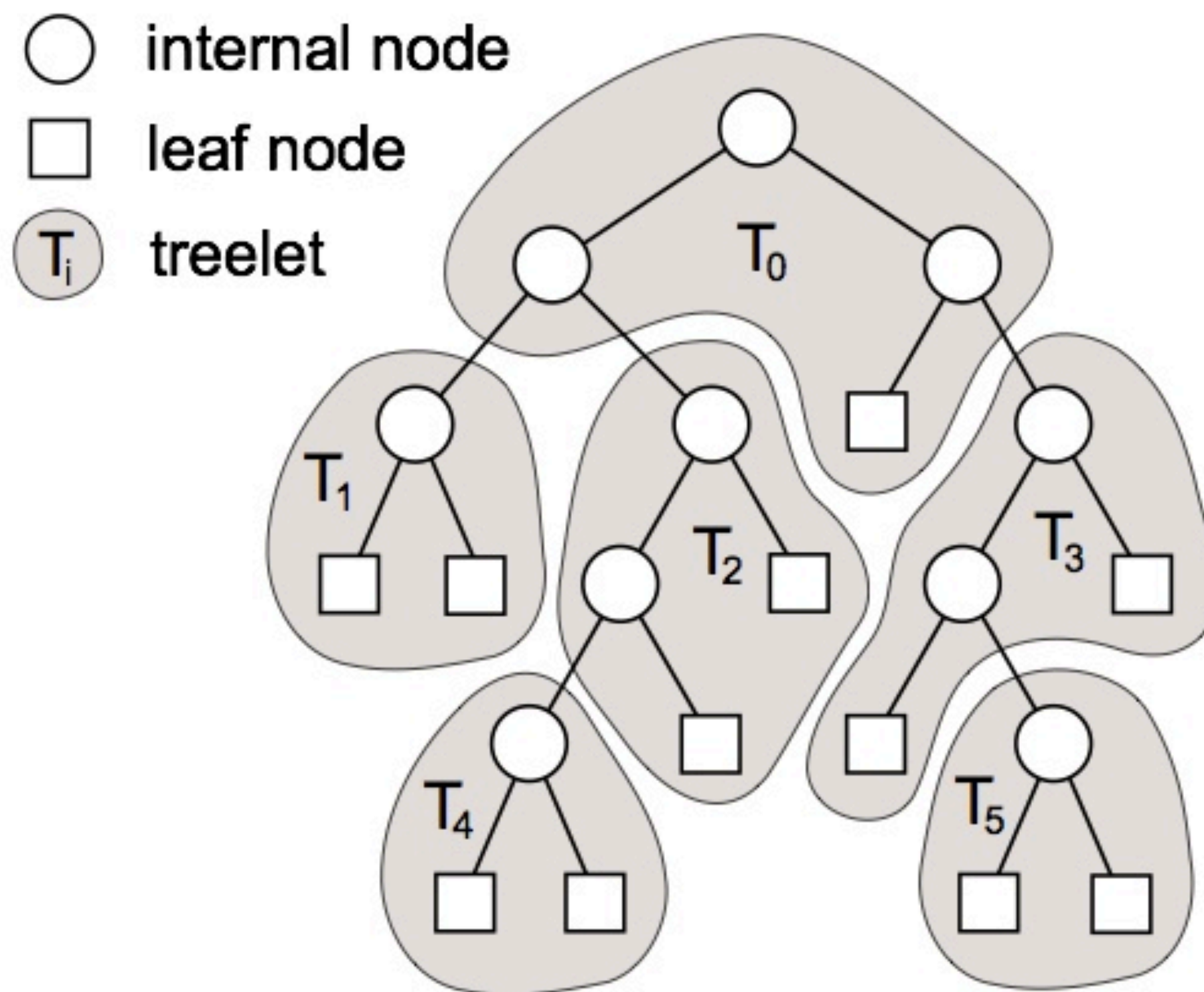
- Custom logic for accelerating ray-box and ray-triangle tests
  - SIMD designs: wide SIMD execution not beneficial
- Support for efficiently reordering ray-tracing computations to maximize memory locality (ray scheduling)

## ■ See “further reading” on web site for a list of references

# Global ray reordering

[Pharr 1997, Navratil 07, Alia 10]

**Idea: 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 constrained to fit in caches (or in dedicated ray buffer SRAM)**

[Pharr 1997, Navratil 07, Alia 10]



# Summary

# Not discussed today

**A practical, efficient real-time ray tracing system will also need to solve these important challenges**

## **1. Building the BVH efficiently**

- **Good recent work on parallel BVH builds, see course web site for references**

## **2. On-demand geometry: tessellation**

- **Tessellate surface first time it is hit by a ray**
- **Intersection modifies BVH (not so embarrassingly parallel anymore)**
- **How to determine level-of-detail?**

## **3. Efficiently shading ray hits**

- **Shading remains at least 50% of execution time in modern ray tracers (making ray tracing infinitely fast yields only a 2X speedup!)**
- **What to do when rays in a packet hits surfaces with different shaders?**

# 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, not as different as you might think**
- **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  $\lg(N)$ )



# Trends: ray tracing used often in film



Image Credit: Blue Sky

- **Reflections, indirect illumination, ambient occlusion, some shadows often computed via ray tracing**
- **Sony Pictures Imageworks now uses only ray tracing for all films**
  - **Arnold renderer has replaced Renderman at Sony**
- **Complex reasons motivate shift to ray tracing**
  - **More than just performance (artist time, production cost, etc.)**



Image Credit: Pixar (Cars)



Image Credit: Sony (Cloudy With a Chance of Meatballs)



# Readings

- **For next time:**
  - **T. Aila and S. Laine, *Understanding the Efficiency of Ray Traversal* on GPUs. High Performance Graphics 2009**
  - **T. Aila and S. Laine, *Architecture Considerations for Tracing Incoherent Rays*. High Performance Graphics 2010**
  
- **Lots of supplemental ray tracing readings posted on the web site**
  - **Best practice ray-tracing algorithms for CPUs/GPUs**
  - **Specialized hardware research prototypes**