Lecture 19:

Systems Trends in Real-Time Ray Tracing + Course Review

Visual Computing Systems
Stanford CS348K, Fall 2018

Presentations: next Tuesday

- 10-minute slots per project group
- Aim for eight minutes of speaking + 2 minutes discussion
- Key goal of the presentation:
  - Tell the class:
    - What the problem was (goals and constraints)
    - What the most interesting part of the project was ("The challenging part was how we solved…")
    - Provide a clear piece of evidence that your goals were achieved ("here is our graph of performance vs…")

A few clear talk tips

For a full treatment see:

http://graphics.stanford.edu/~kayvonf/misc/cleartalktips.pdf

1.

Establish inputs, outputs, and constraints (goals and assumptions)

Establish goals and assumptions early

- Given these inputs, we wish to generate these outputs
- We are working under the following constraints
  - Example: the outputs should have these properties
  - Example: the algorithm...
    - Should be real-time
    - Should be parallelizable
    - Cannot require artist intervention
    - Must be backward compatible with this content creation pipeline

Your contribution is typically a system or algorithm that meets the stated goals under the stated constraints.

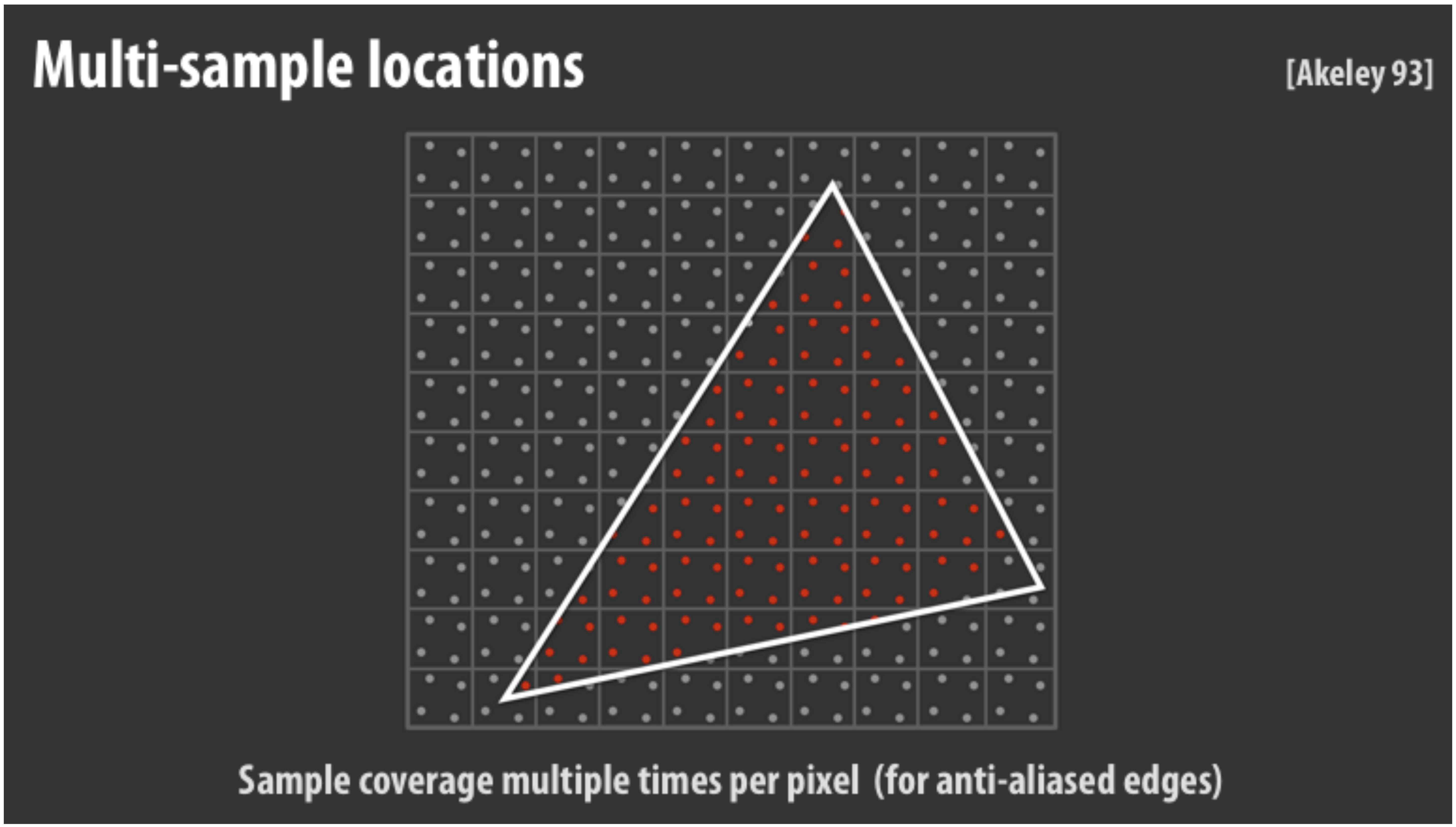
2.

Always, always, always explain any figure or graph

(the audience does not want to think about things you can tell them)

Explain every figure

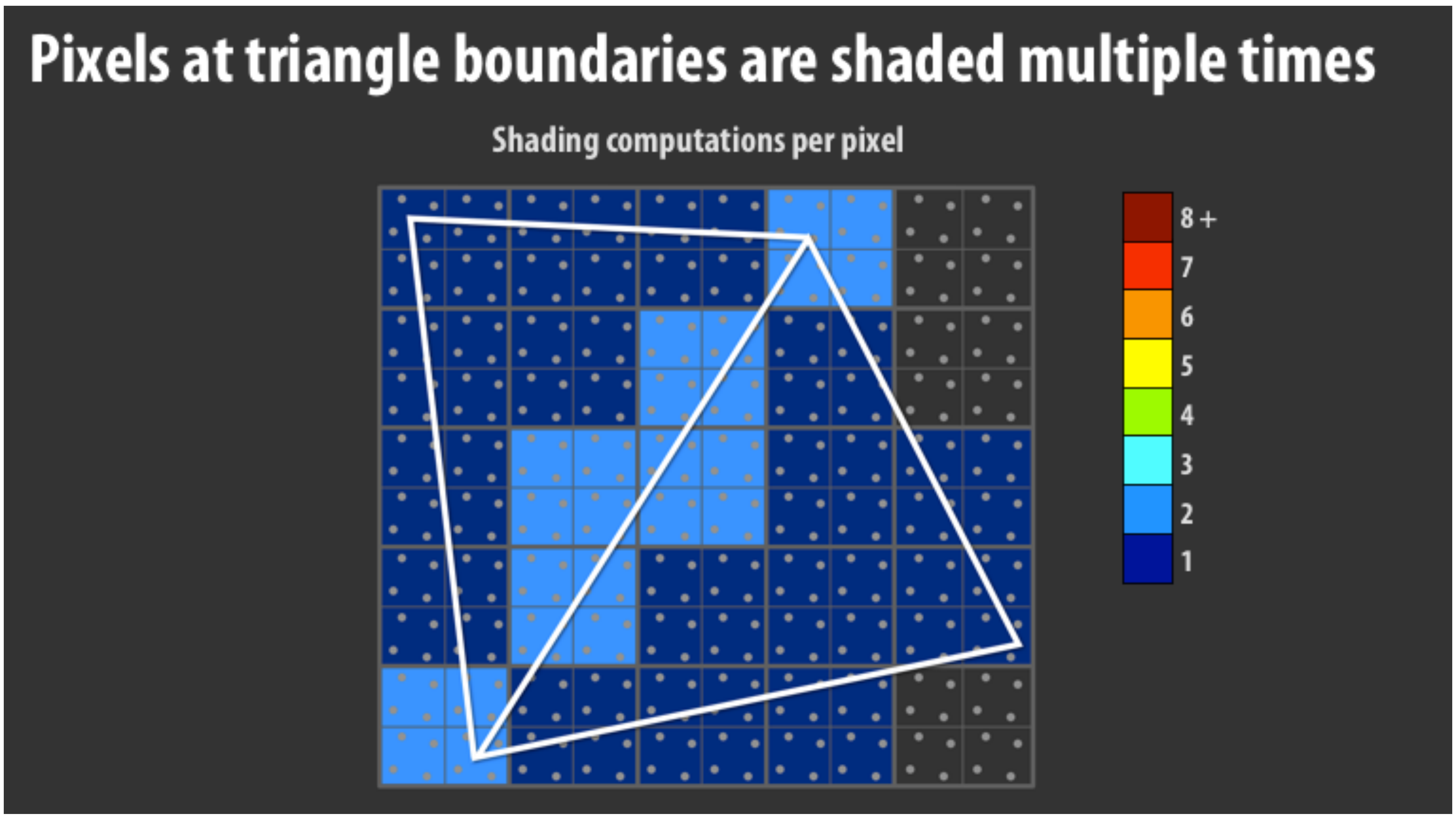
- Explain every visual element used in the figure (don’t make the audience decode a figure)
- Refer to highlight colors explicitly (explain why the visual element is highlighted)



Example voice over: “Here I’m showing you a pixel grid, a projected triangle, and the location of four sample points at each pixel. Sample points falling within the triangle are colored red.

Explain every figure

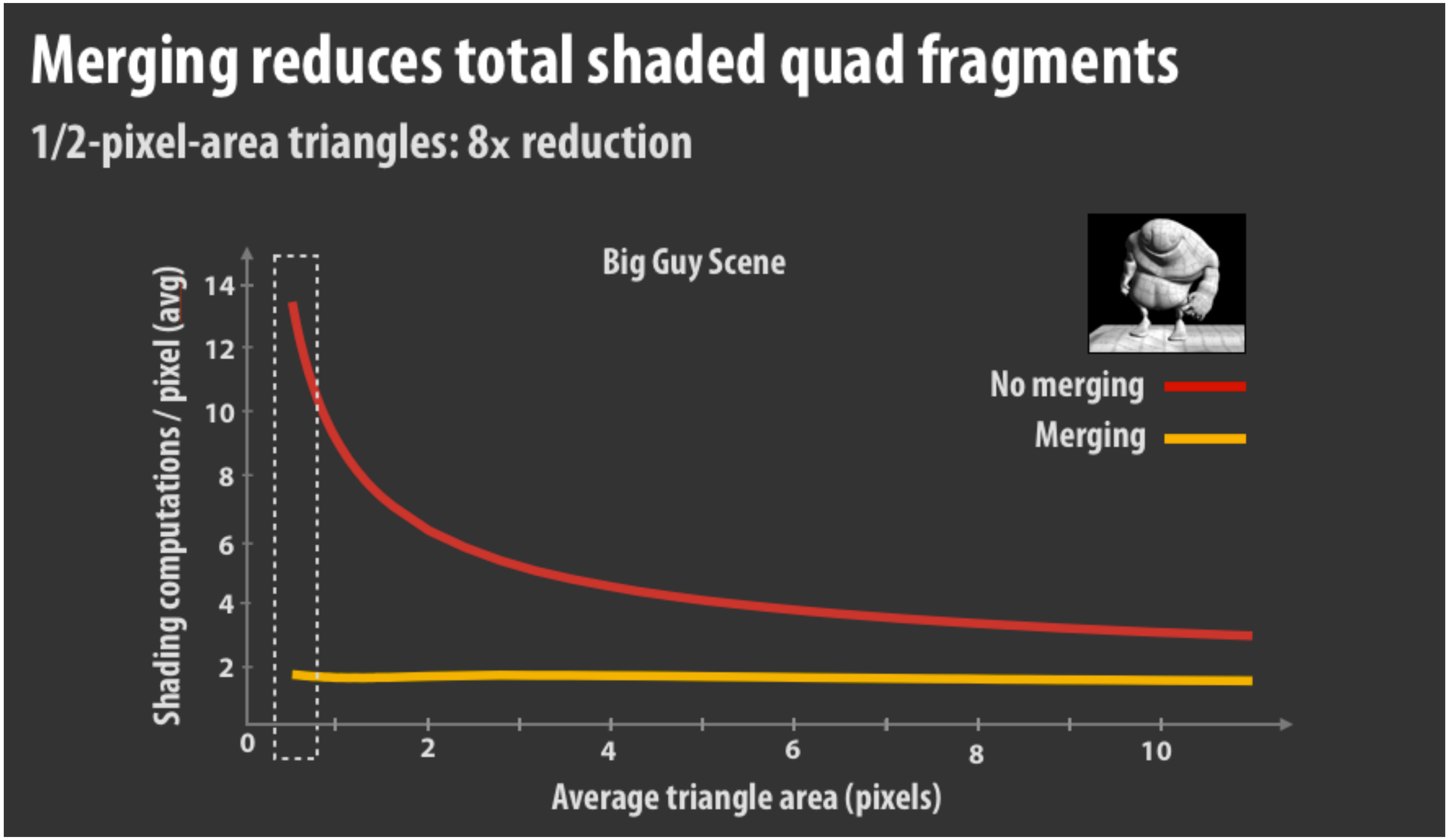
- **Lead the listener through the key points of the figure**
- **Useful phrase: "As you can see..."**
  - It's like verbal eye contact. It keeps the listener engaged and makes the listener happy... "Oh yeah, I can see that! I am following this talk!"



Example voice over: "Now I'm showing you two adjacent triangles, and I'm coloring pixels according to the number of shading computations that occur at each pixel as a result of rendering these two triangles. As you can see from the light blue region, pixels near the boundary of the two triangles get shaded twice.

Explain every results graph

- May start with a general intro of what the graph will address (anticipate result)
- Then describe the axes (and your axes better have labels!)
- Then describe the <u>one point</u> that you wish to make with this results slide (more on this later!)



Example voice over: "Our first questions were about performance: how much did merging reduce the number of the shaded quad fragments? And we found out that the answer is a lot. This figure plots the number of shading computations per pixel when rendering different tessellations of the big guy scene. X-axis gives triangle size. If you look at the left side of the graph, which corresponds to a high-resolution micropolygon mesh, <u>you can see</u> that merging, shown by yellow line, shades over eight times less than the convention pipeline.

3.

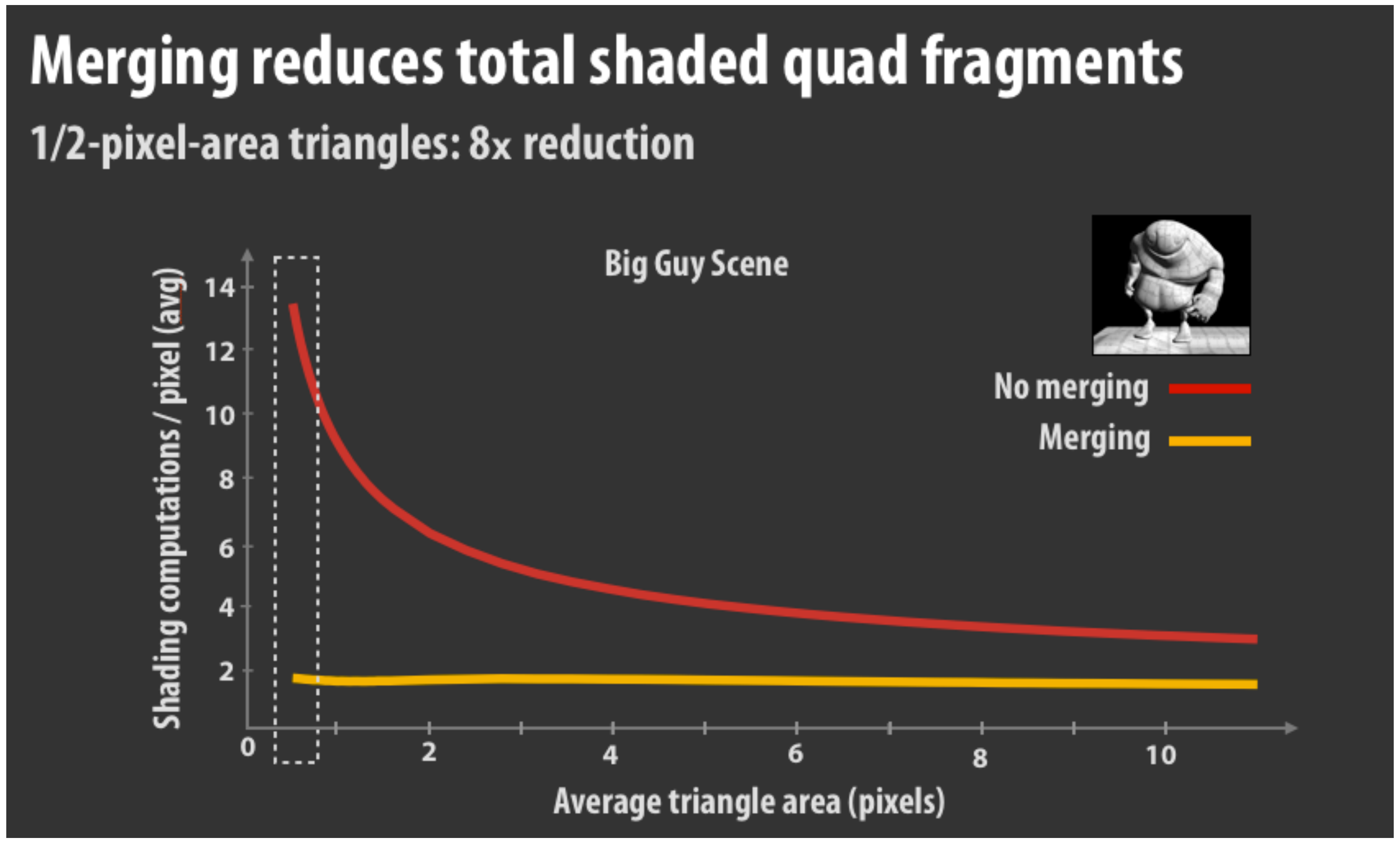
In the results section:

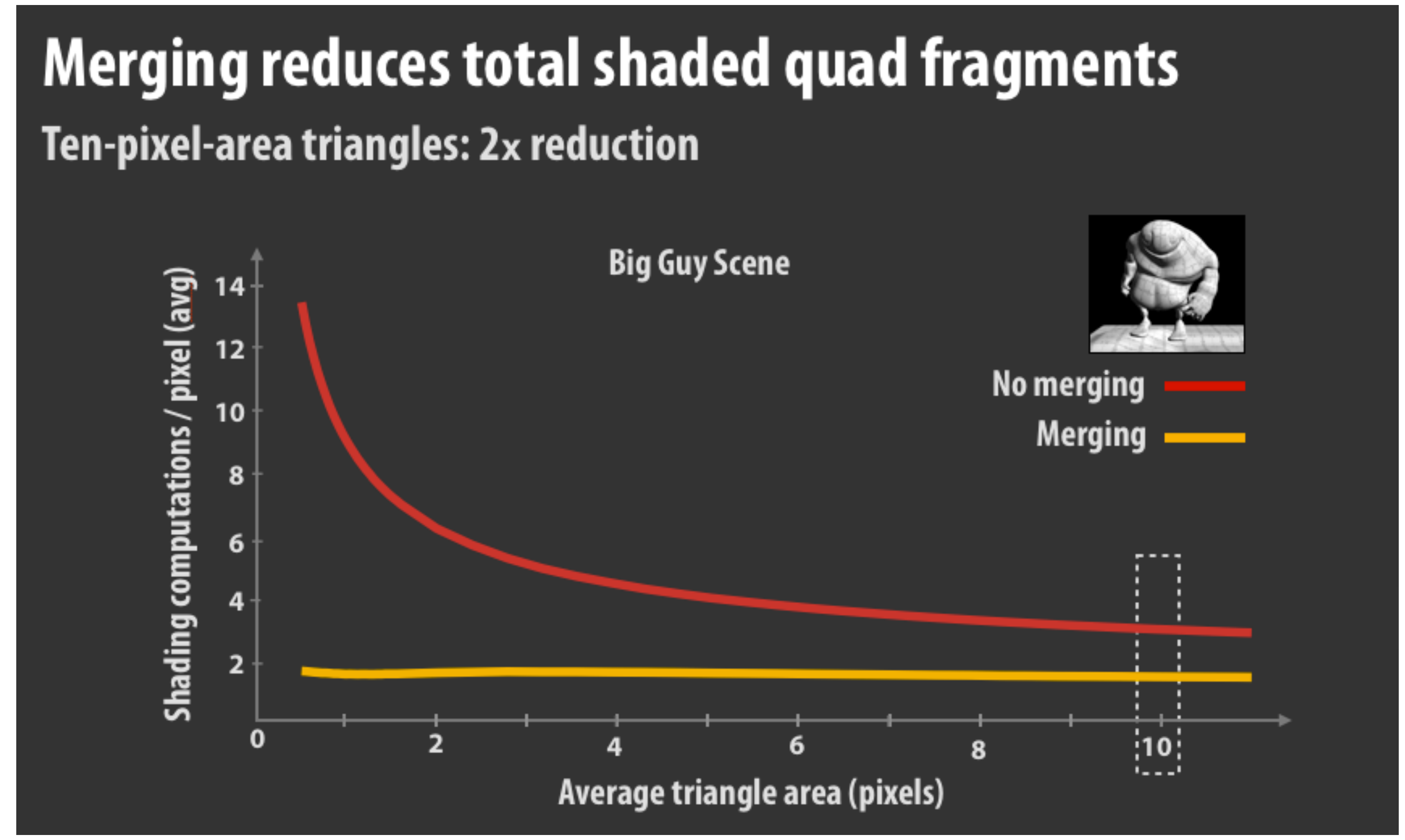
One point per slide!

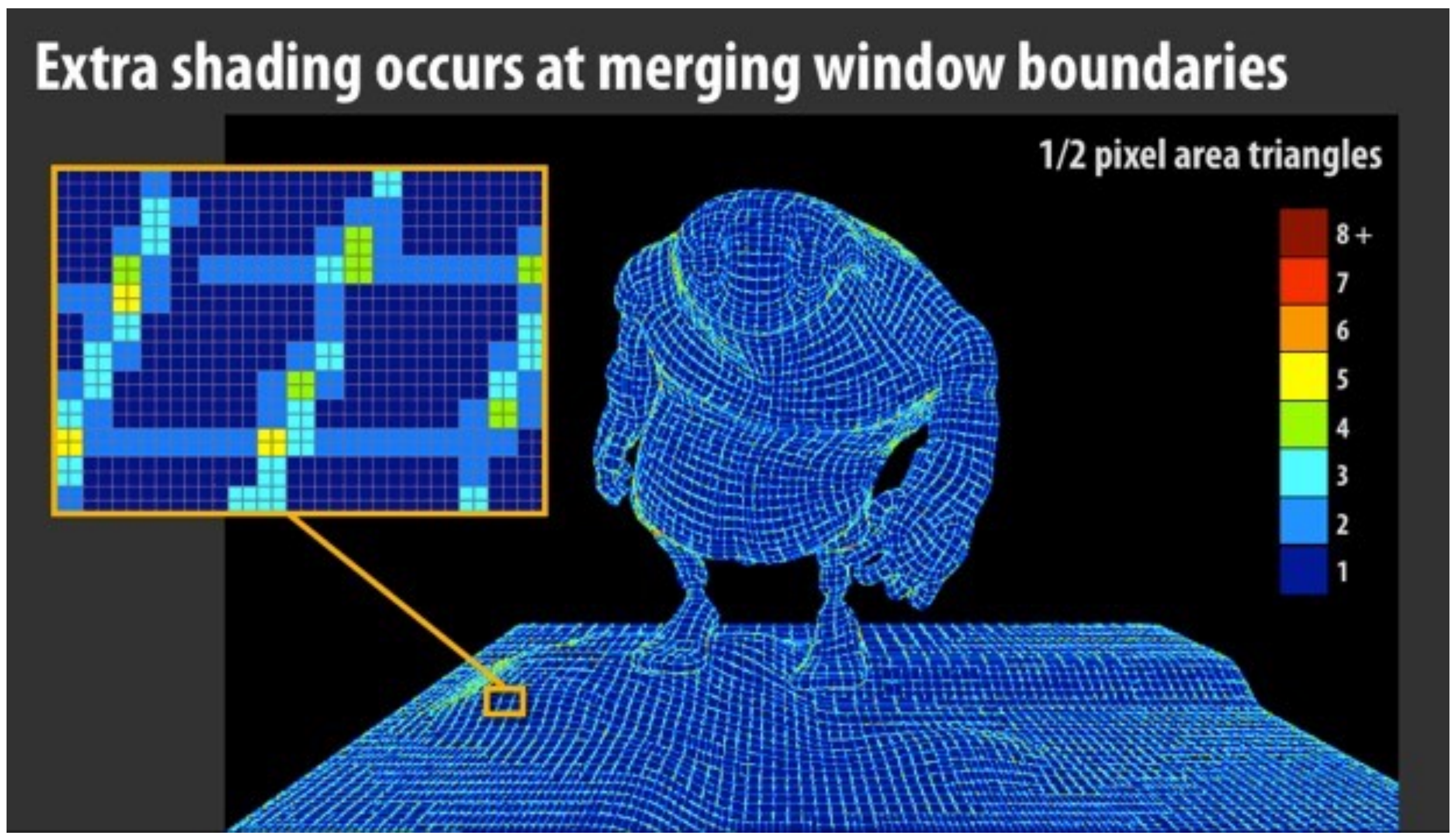
One point per slide!

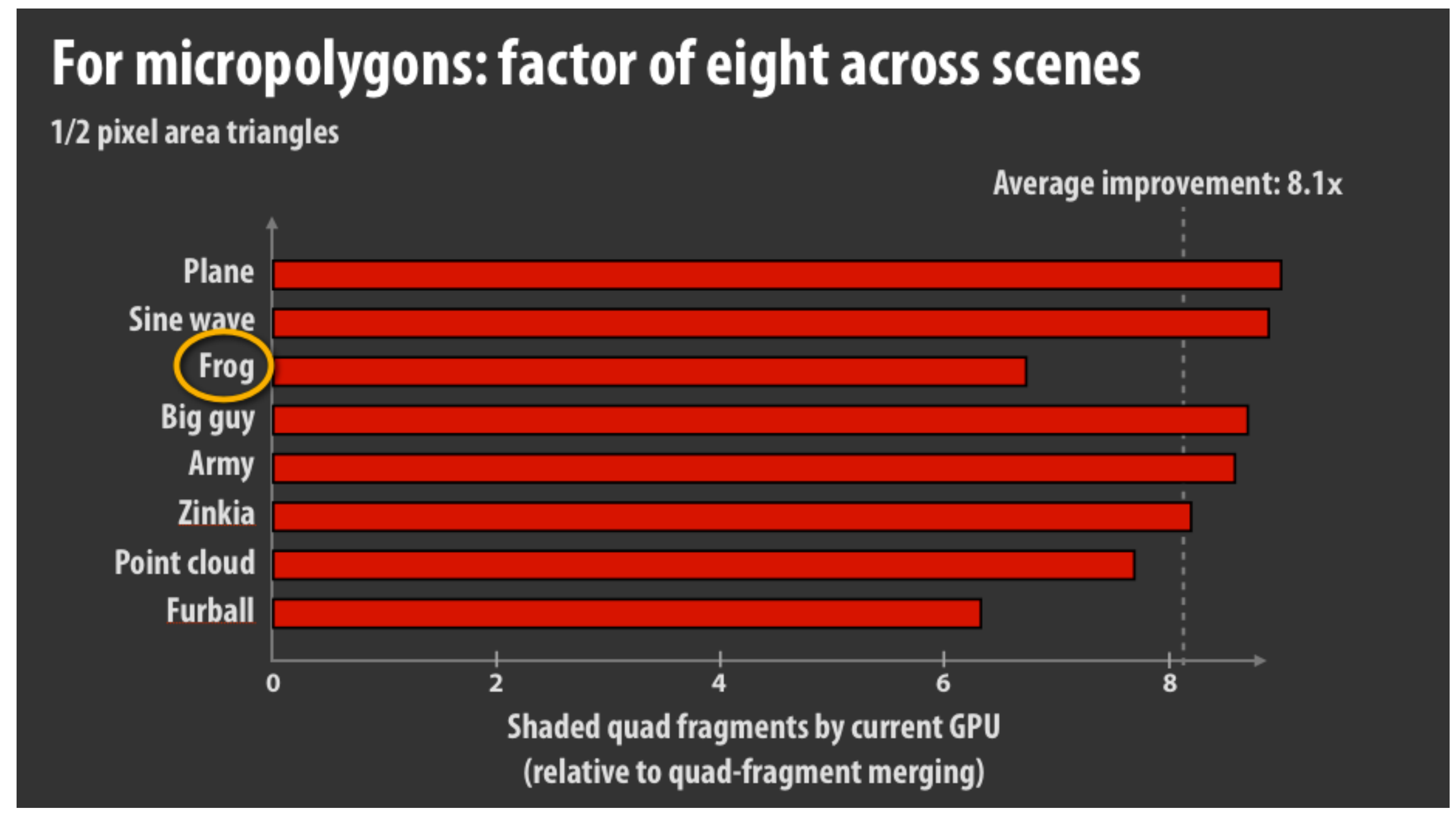
One point per slide!

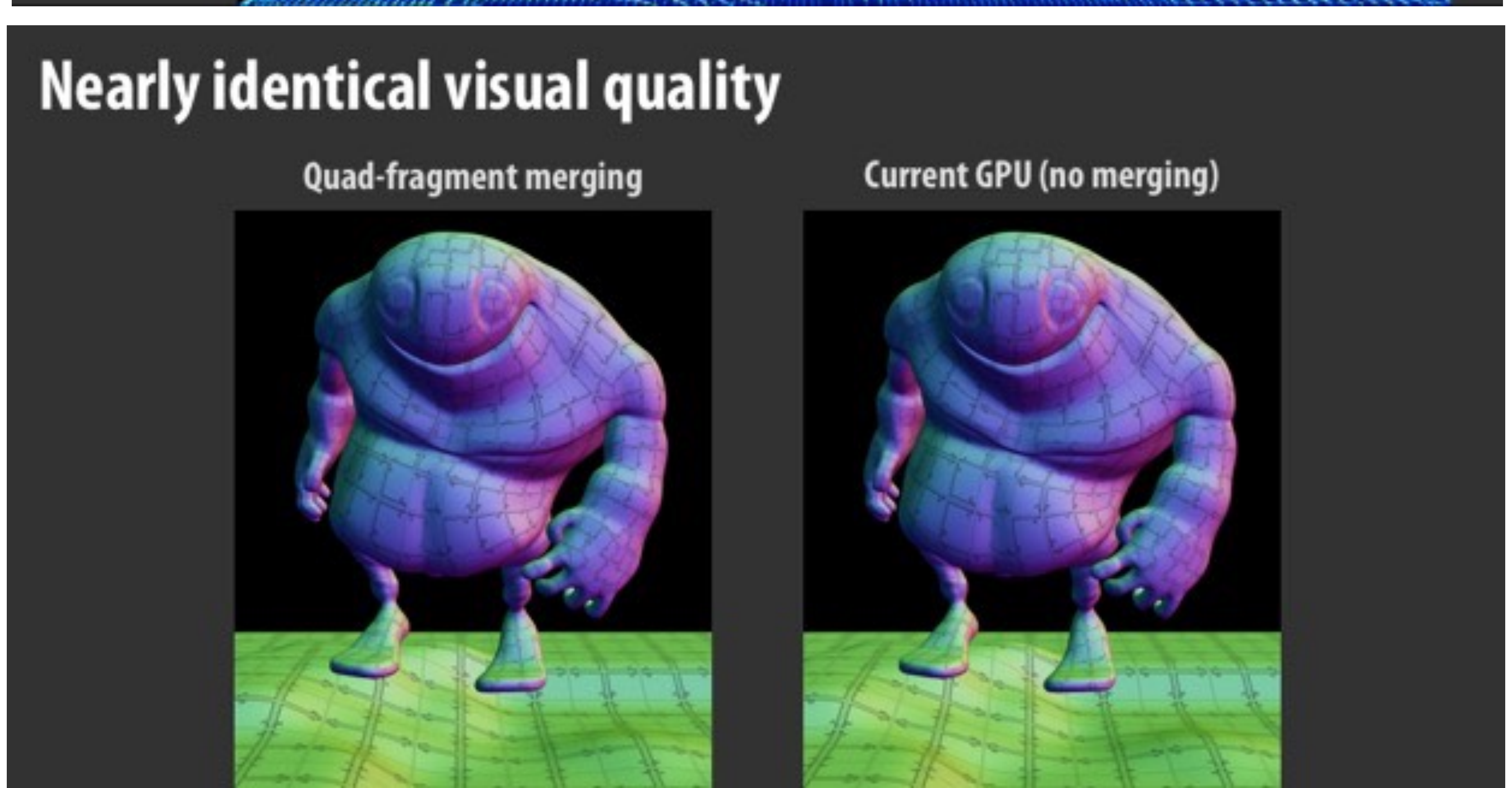
(and the point is the title of the slide!!!)

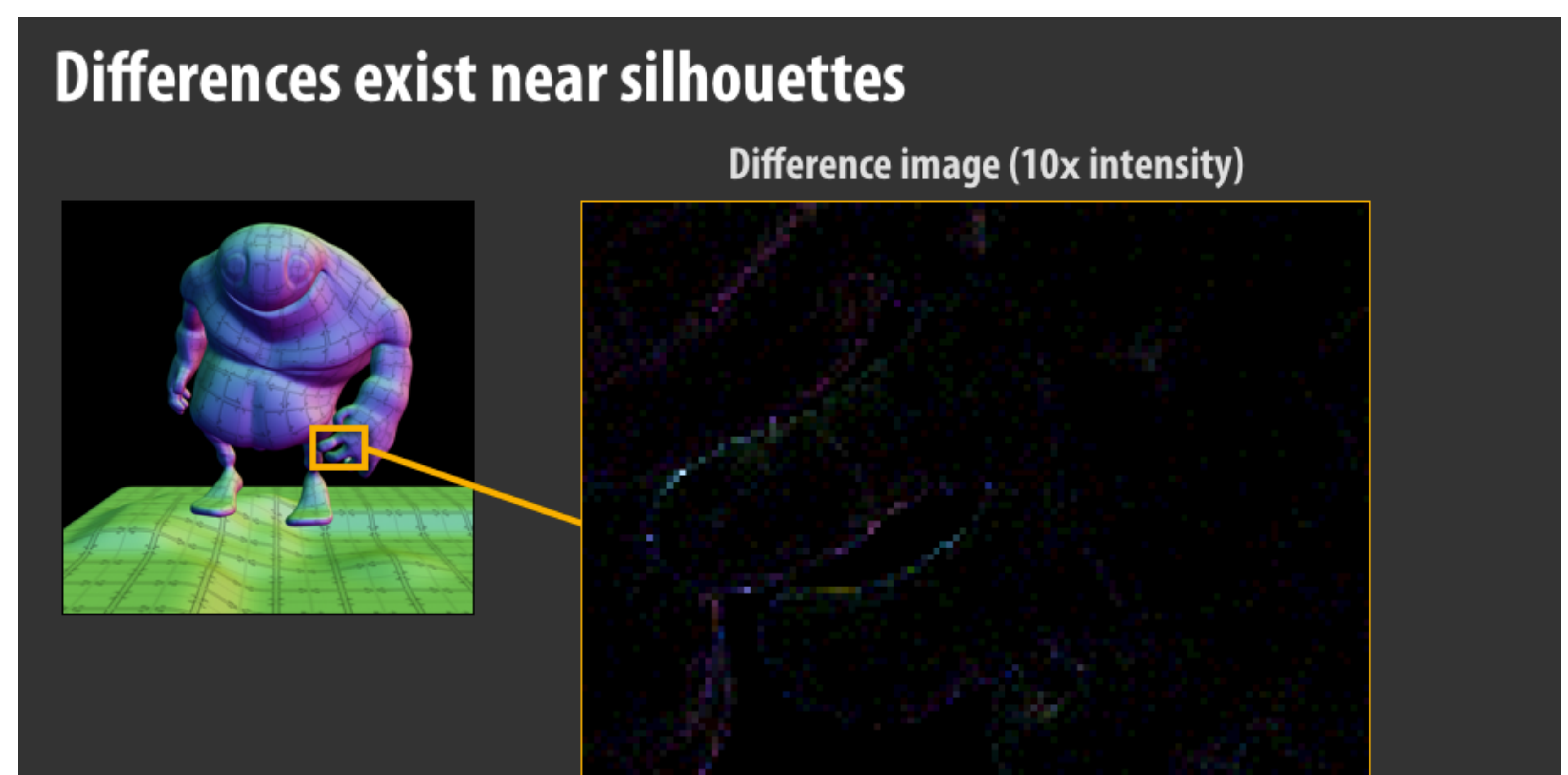

Merging reduces total shaded quad fragments
1/2-pixel-area triangles: 8x reduction
Big Guy Scene
No merging
Merging
Shading computations / pixel (avg)
0
2
4
6
8
10
12
14
Average triangle area (pixels)

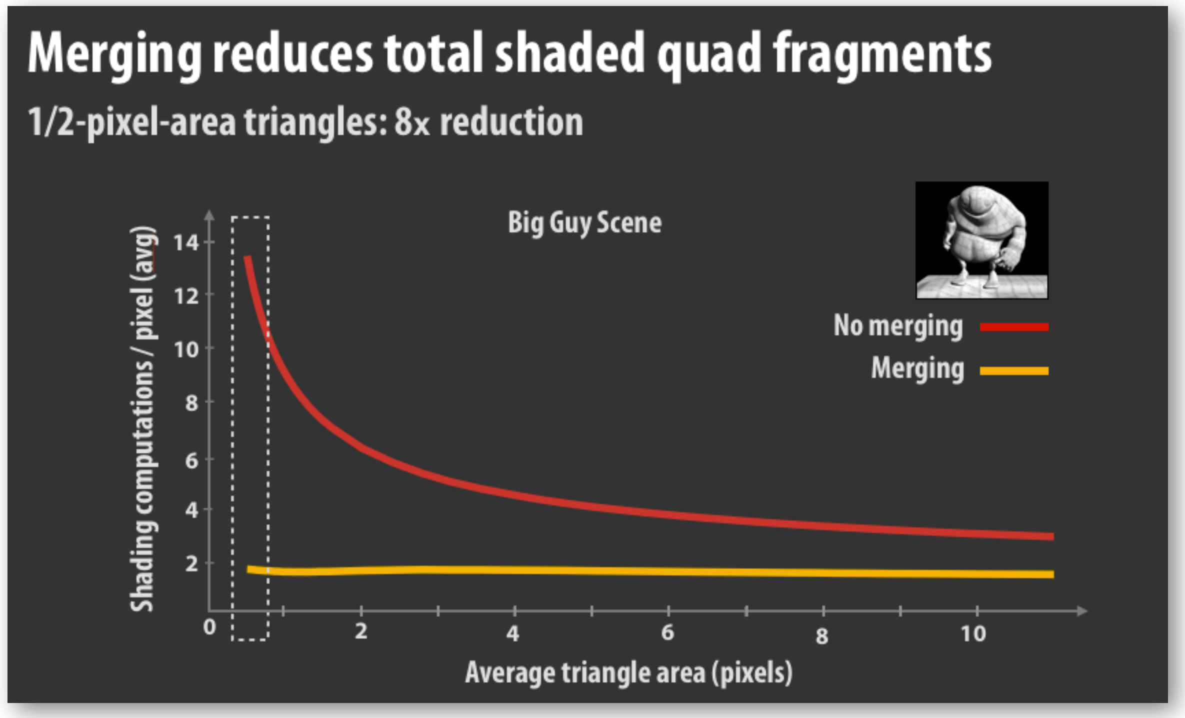

Merging reduces total shaded quad fragments
Ten-pixel-area triangles: 2x reduction
Big Guy Scene
No merging
Merging
Shading computations / pixel (avg)
0
2
4
6
8
10
12
14
Average triangle area (pixels)

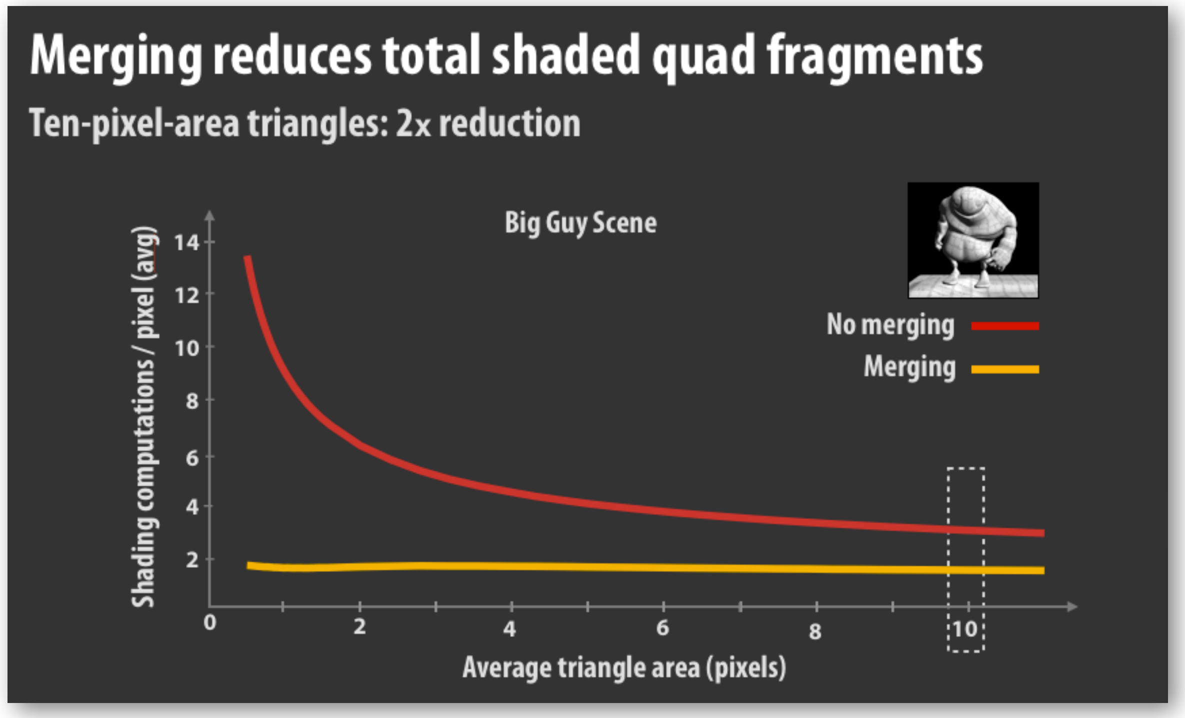

Extra shading occurs at merging window boundaries
1/2 pixel area triangles
8+
7
6
5
4
3
2
1

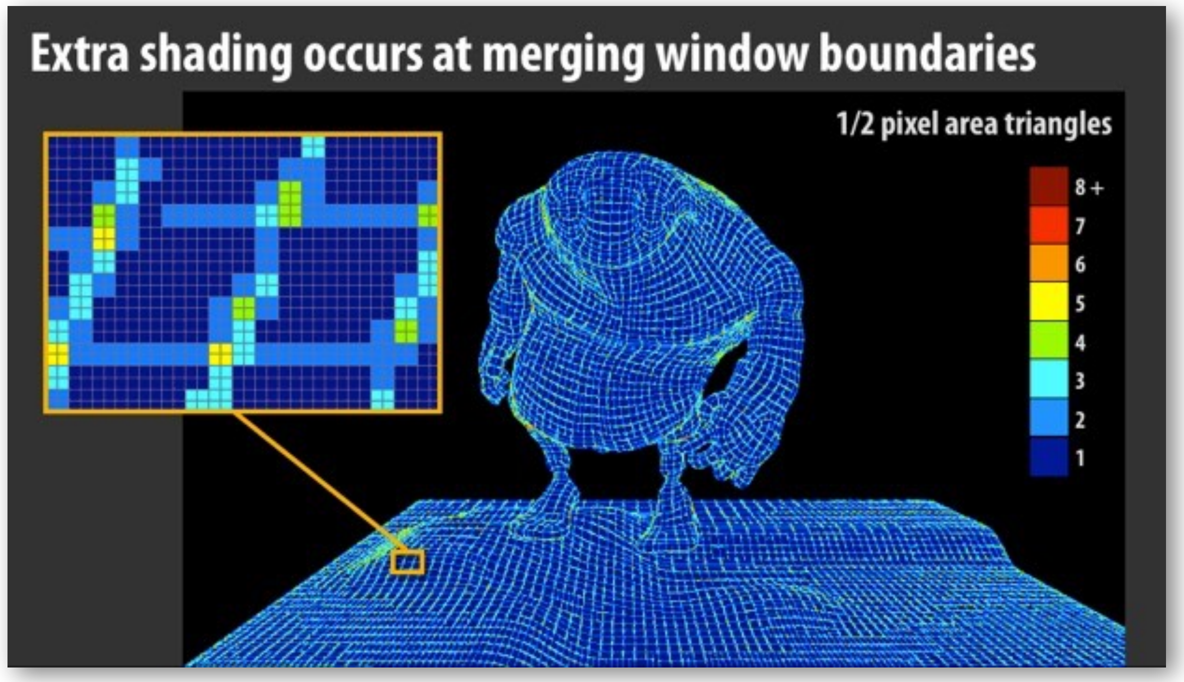

For micropolygons: factor of eight across scenes
1/2 pixel area triangles
Average improvement: 8.1x
Plane
Sine wave
Frog
Big guy
Army
Zinkia
Point cloud
Furball
0
2
4
6
8
Shaded quad fragments by current GPU
(relative to quad-fragment merging)

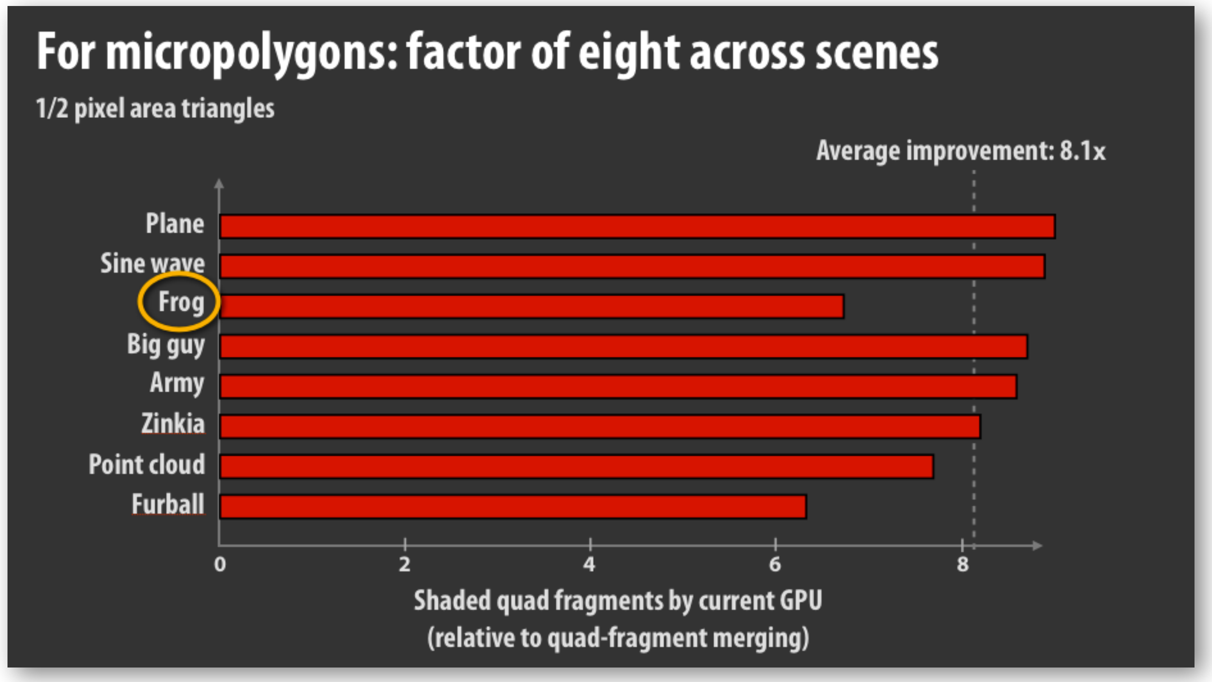

Nearly identical visual quality
Quad-fragment merging
Current GPU (no merging)

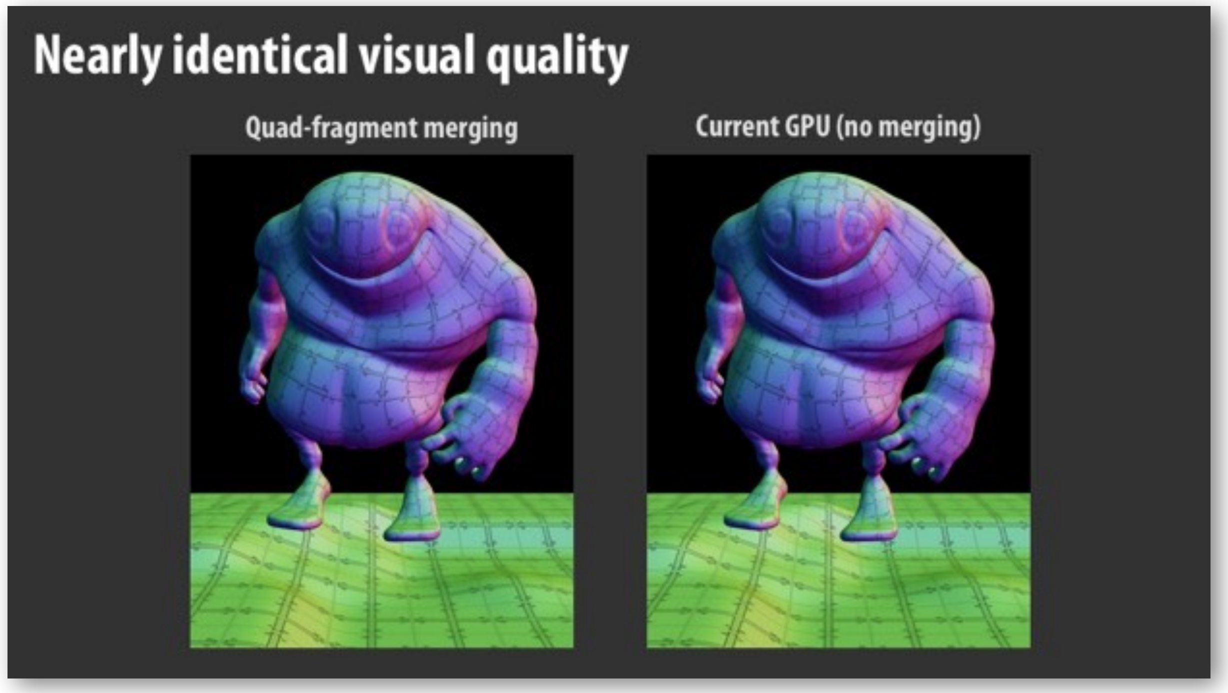

Differences exist near silhouettes
Difference image (10x intensity)

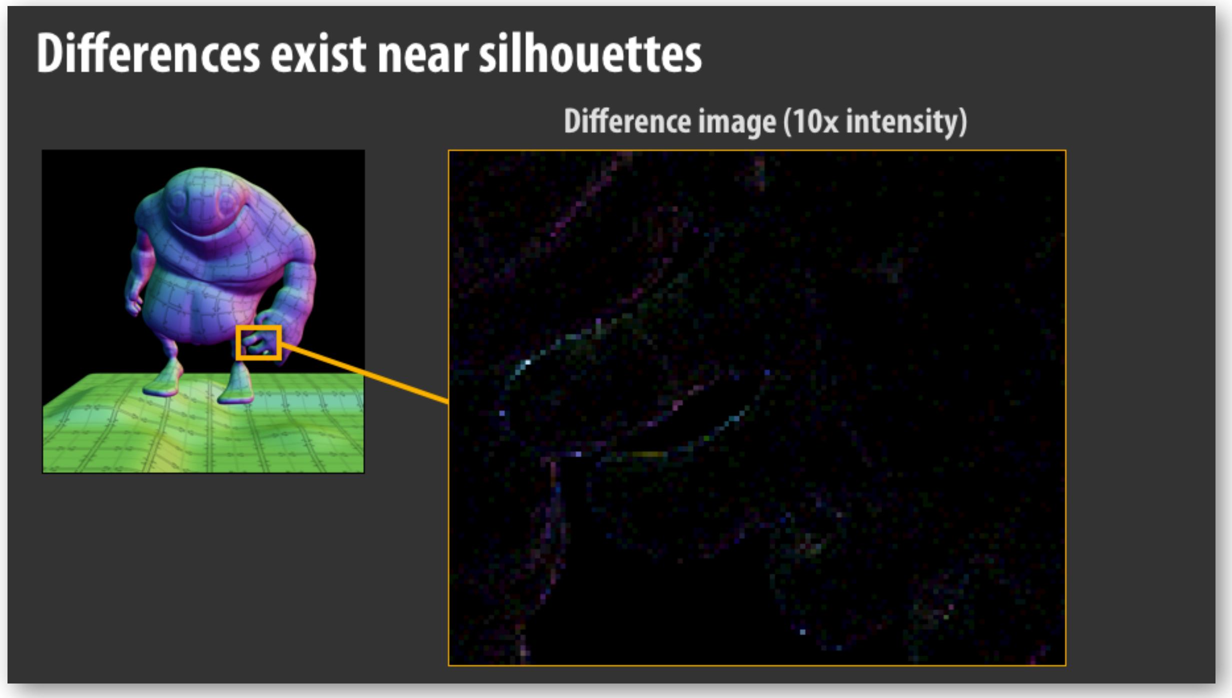












- **Place the point of the slide in the title:**
  - **Provide audience context for interpreting the graph ("Let me see if I can verify that point in the graph to check my understanding")**

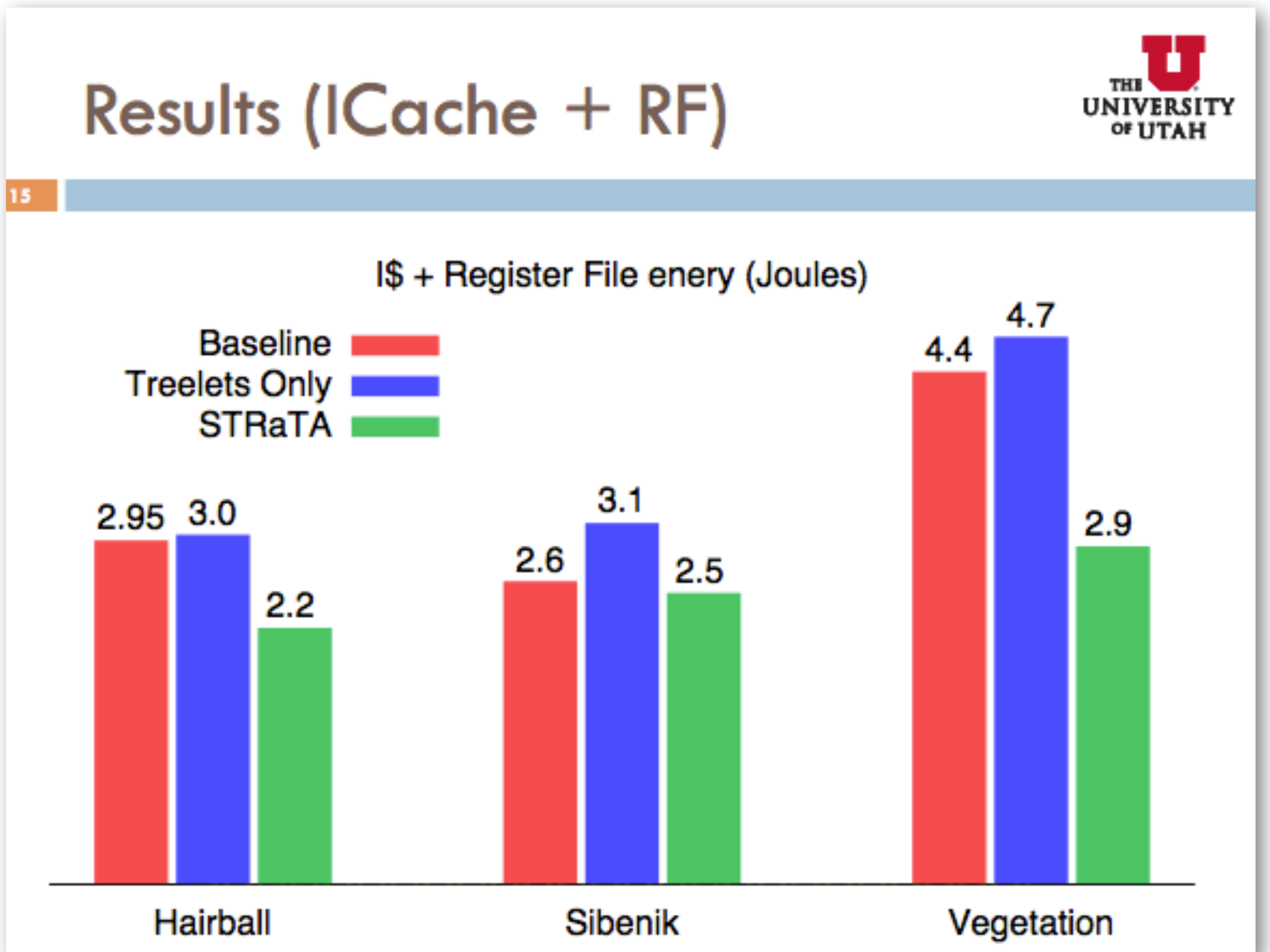
Corollary to the one point per slide rule

- In general, you don't want to show data on a results slide that is unrelated to the point of the slide

- This usually means you need to remake the graphs from your paper (it's a pain, but sorry, it's important) *

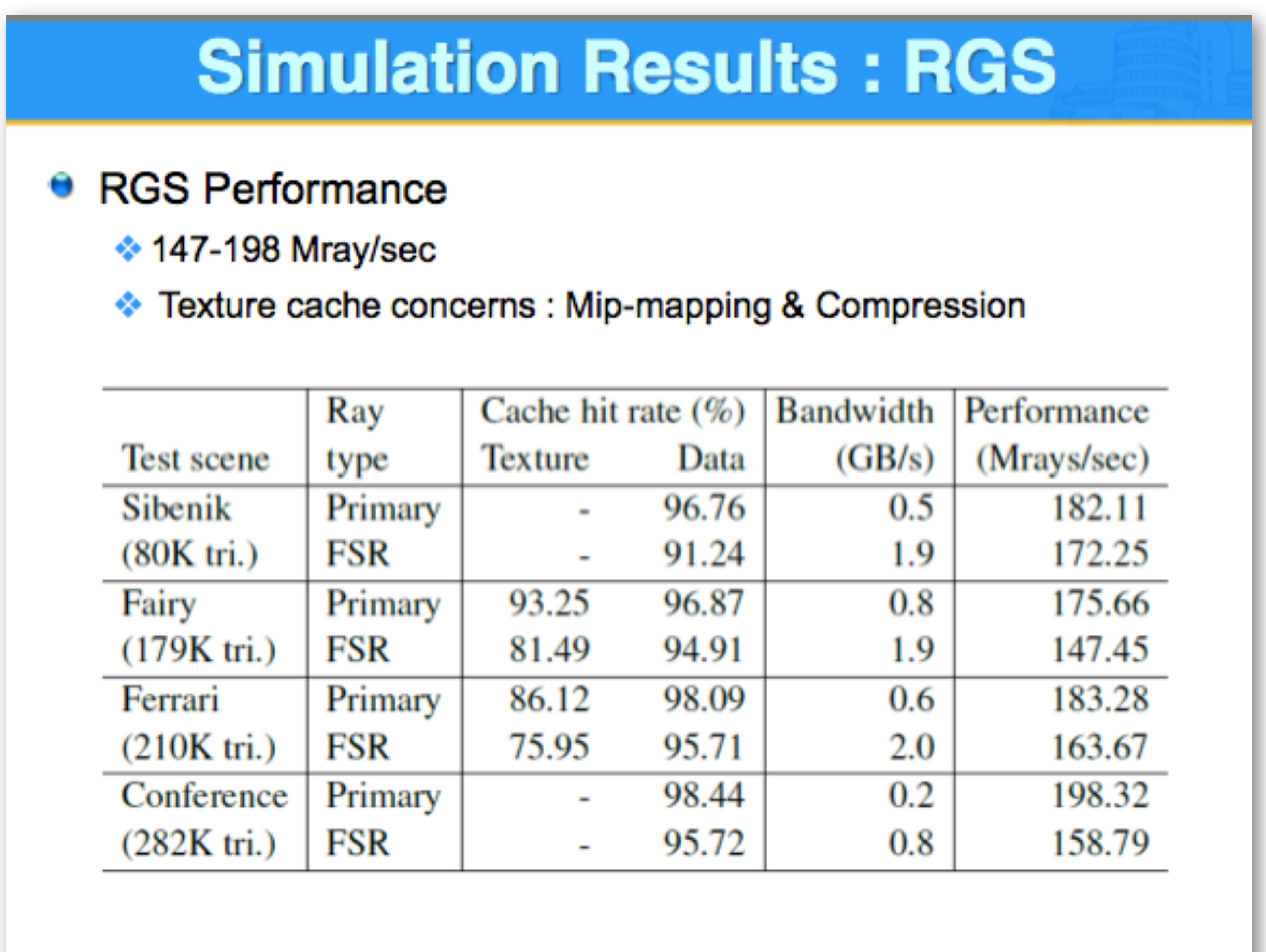
* This is an example of a tip for conference talk polish: not necessary for class talks

Bad examples of results slides

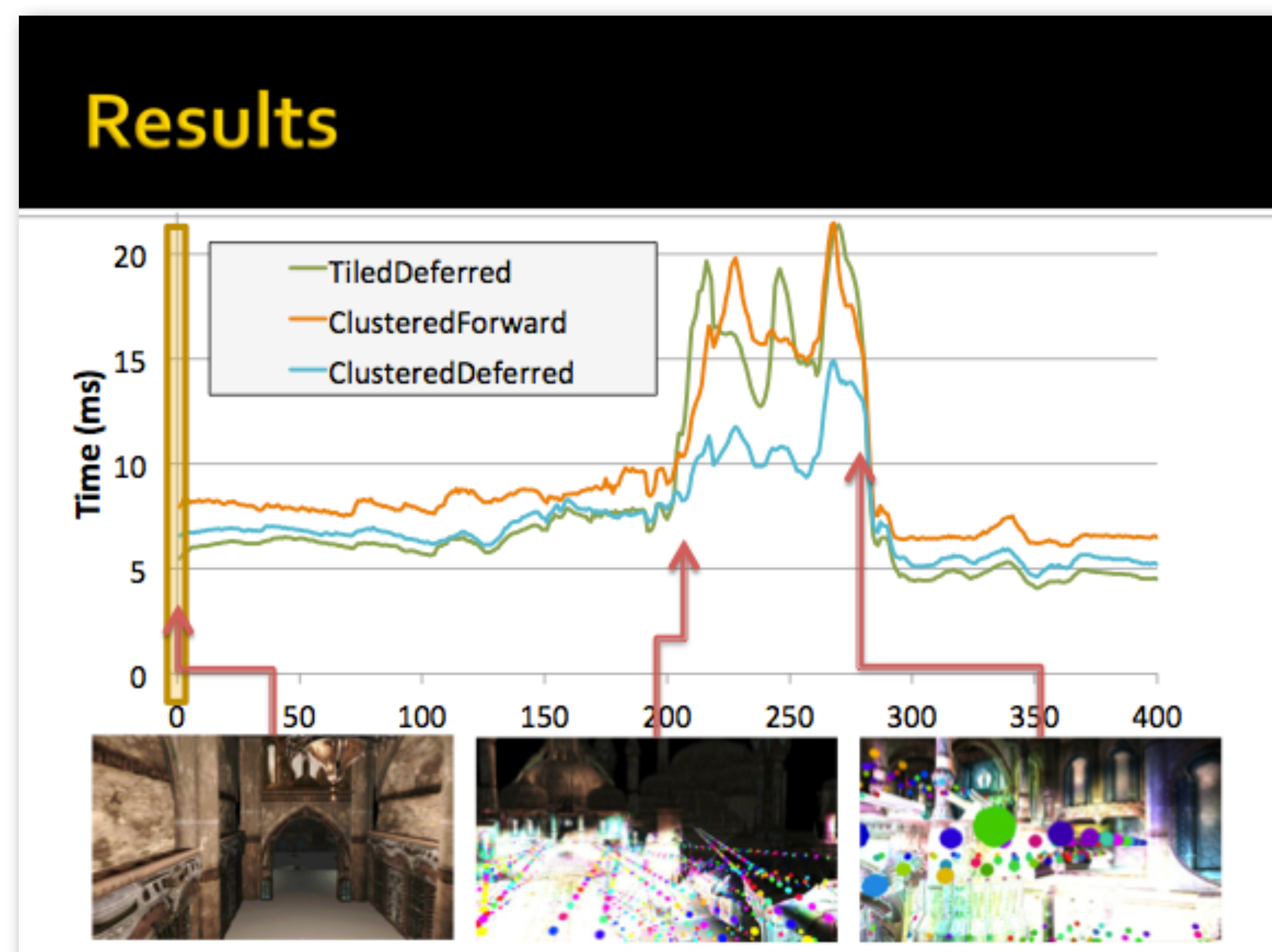


- Notice how you (as an audience member) are working hard to interpret the trends in these graphs
  - You are asking: what do these results say?

- You just want to be told what to look for



Test scene	Ray type	Cache hit rate (%) Texture	Data	Bandwidth (GB/s)	Performance (Mrays/sec)
Sibenik (80K tri.)	Primary	-	96.76	0.5	182.11
	FSR	-	91.24	1.9	172.25
Fairy (179K tri.)	Primary	93.25	96.87	0.8	175.66
	FSR	81.49	94.91	1.9	147.45
Ferrari (210K tri.)	Primary	86.12	98.09	0.6	183.28
	FSR	75.95	95.71	2.0	163.67
Conference (282K tri.)	Primary	-	98.44	0.2	198.32
	FSR	-	95.72	0.8	158.79



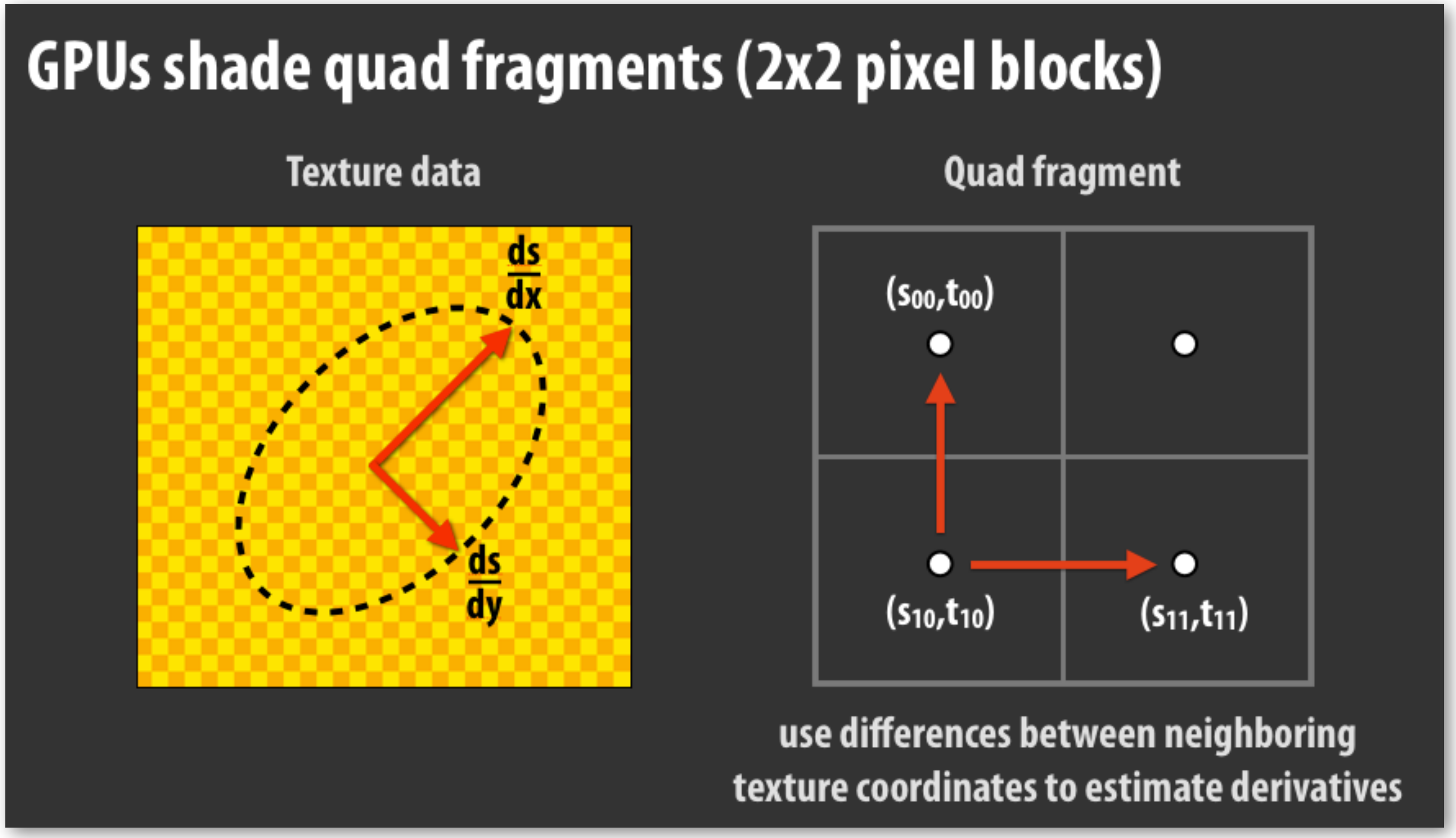
4.

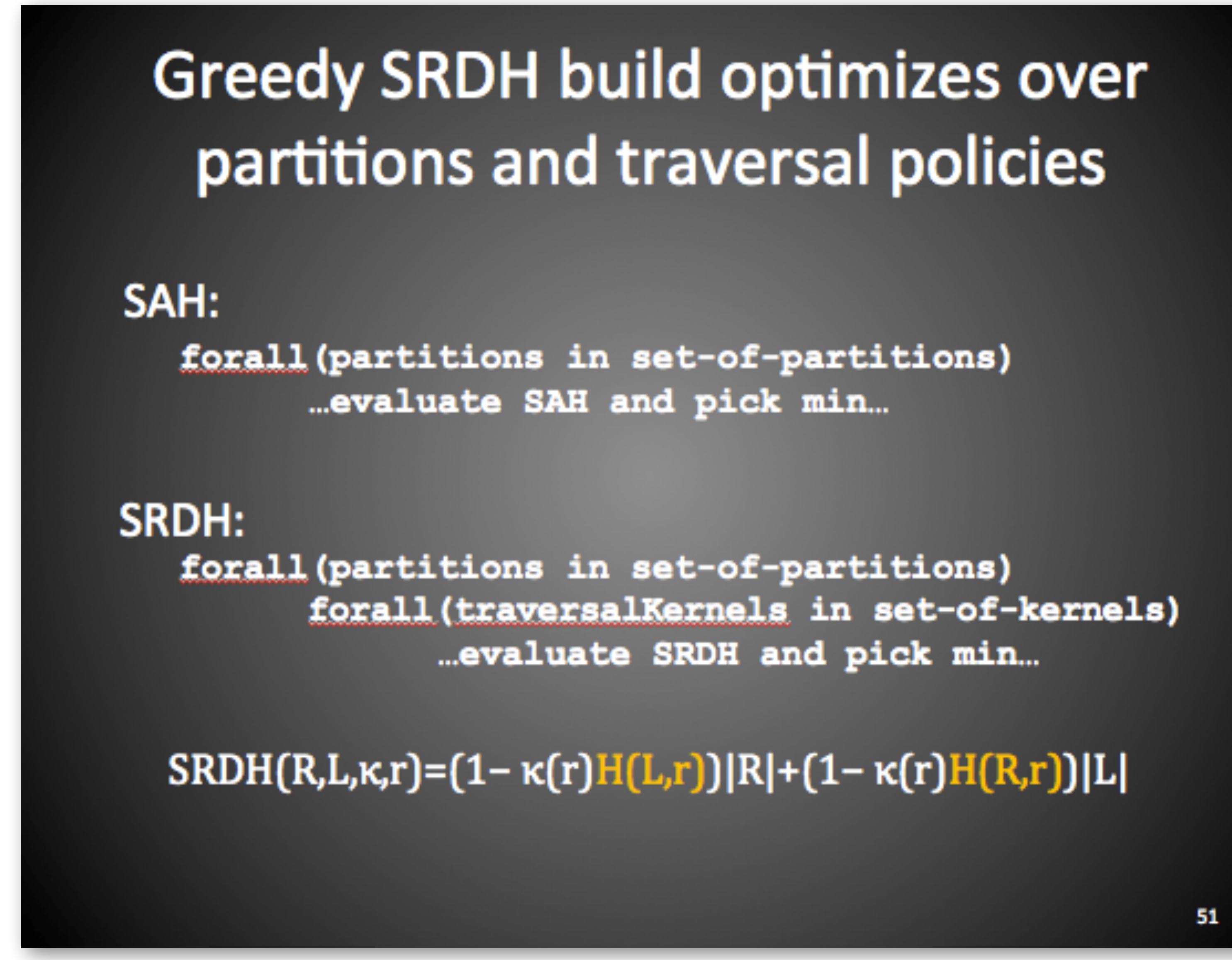
Titles matter.

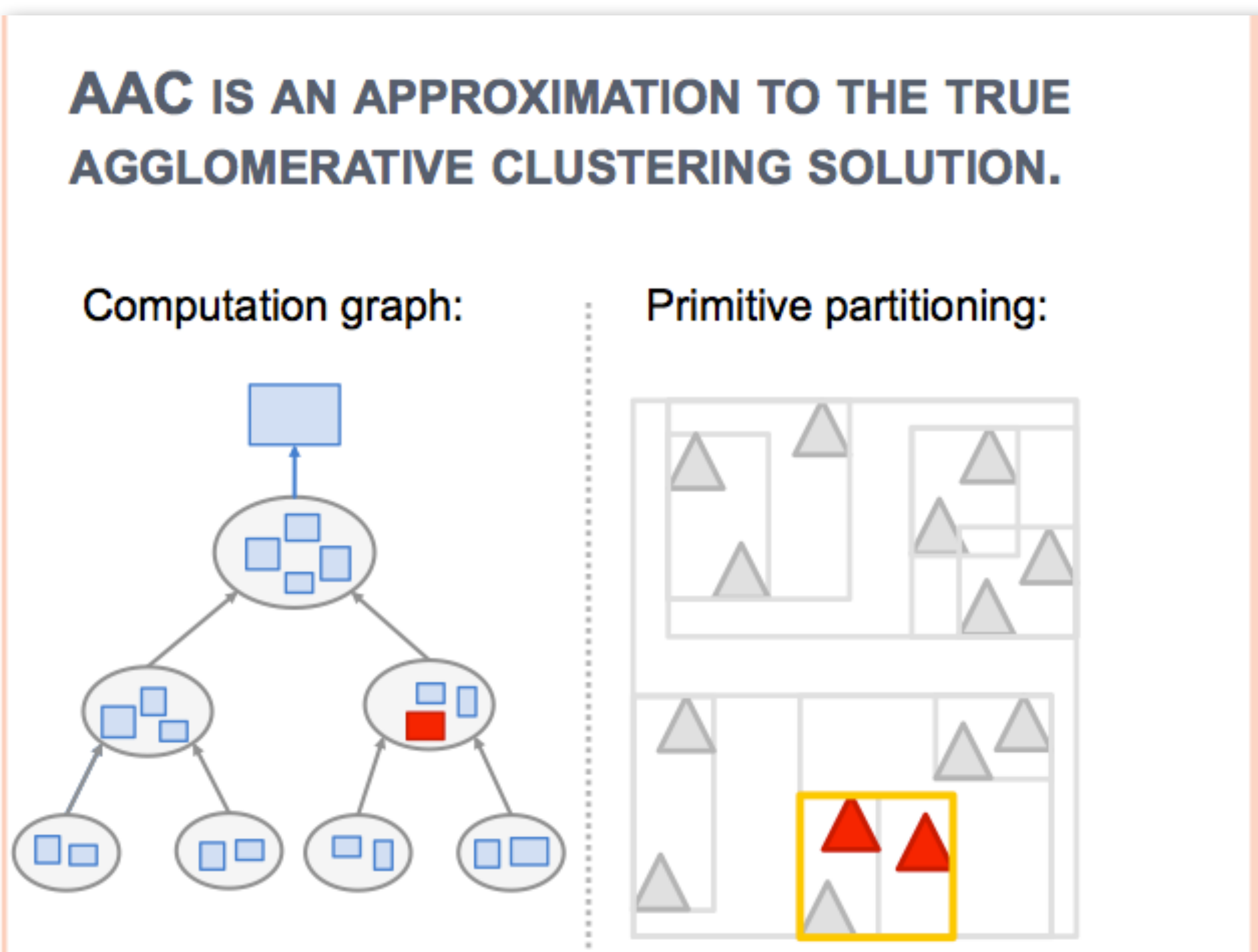
If you read the titles of your talk all the way through, it should be a great summary of the talk.

(basically, this is “one-point-per-slide” for the whole talk)

Examples of good slide titles







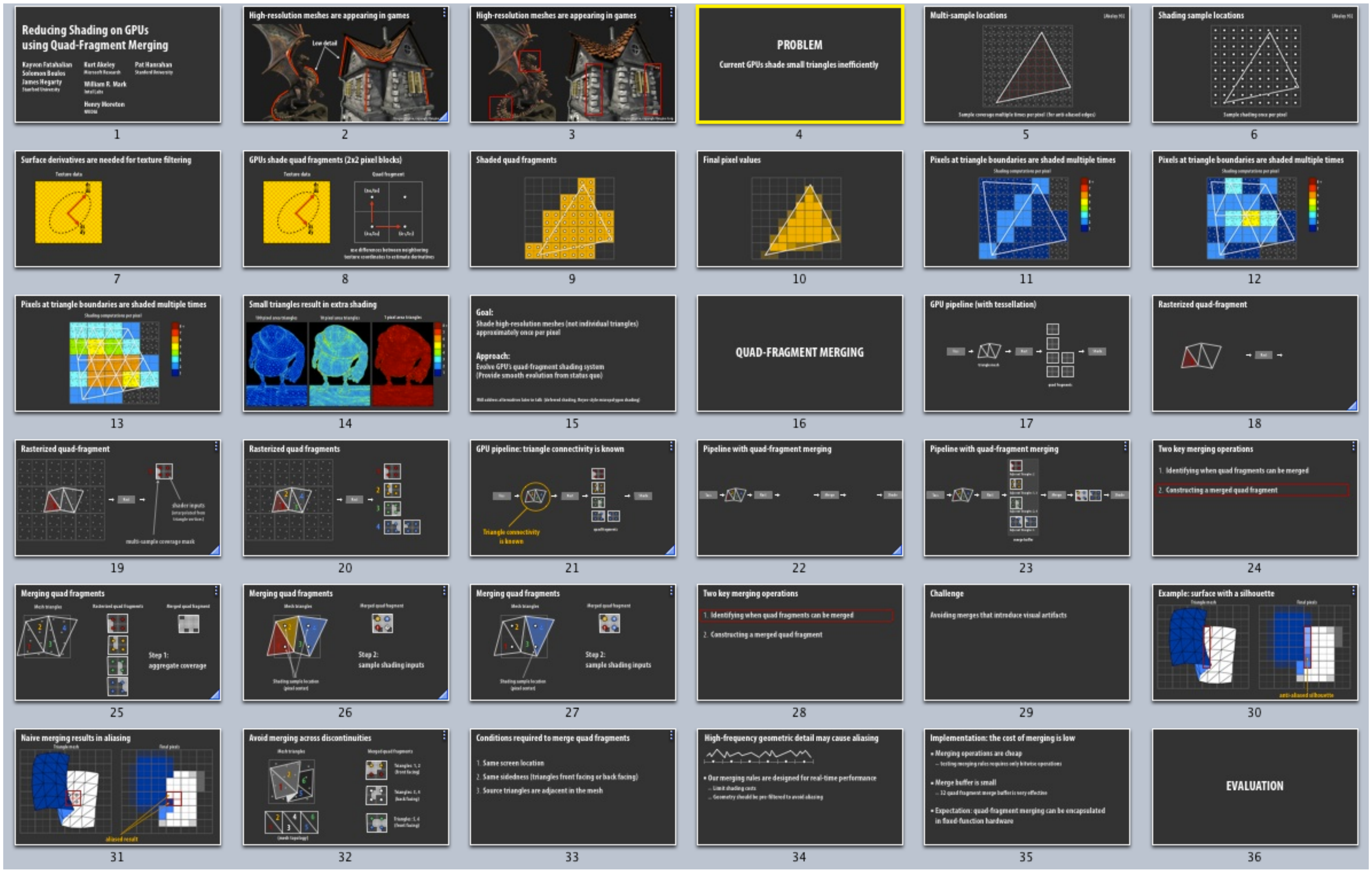
The reason for meaningful slide titles is convenience and clarity for the audience

“Why is the speaker telling me this again?”

(Why before what.)

Read your slide titles in thumbnail view

Do they make all the points of the story you are trying to tell?



5.

Practice.

Even for a 10 minute class talk, practicing the talk <u>out loud</u> the night before goes a lot way

Trends in real-time ray tracing

D3D12 Ray Tracing Support

Examples

- https://www.youtube.com/watch?v=LXo0WdlELJk

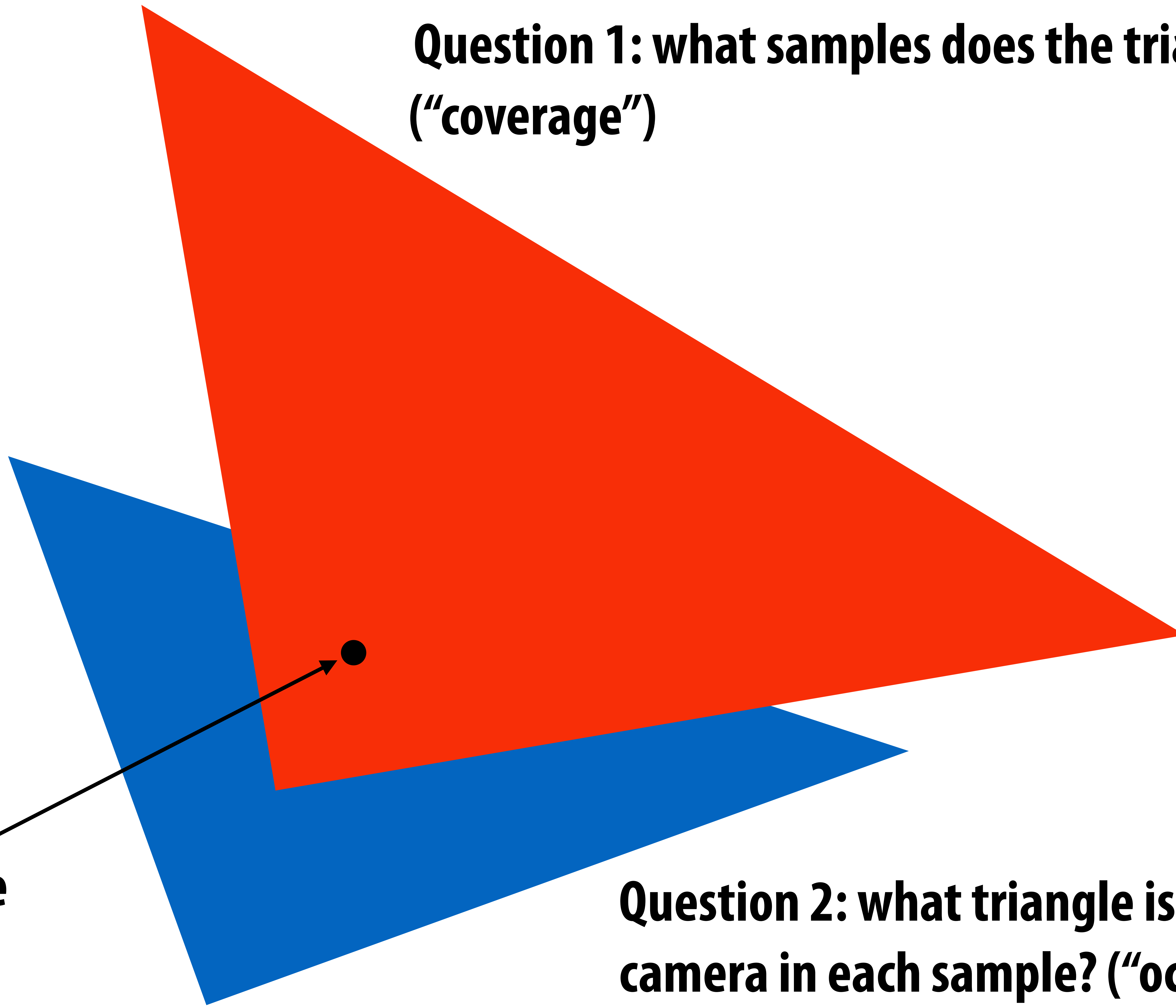
- UE4 Reflections
  - https://www.youtube.com/watch?v=lMSuGoYcT3s

- AtomicHeart Demo
  - https://www.youtube.com/watch?v=1IIiQZw_p_E

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

Visibility problem

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

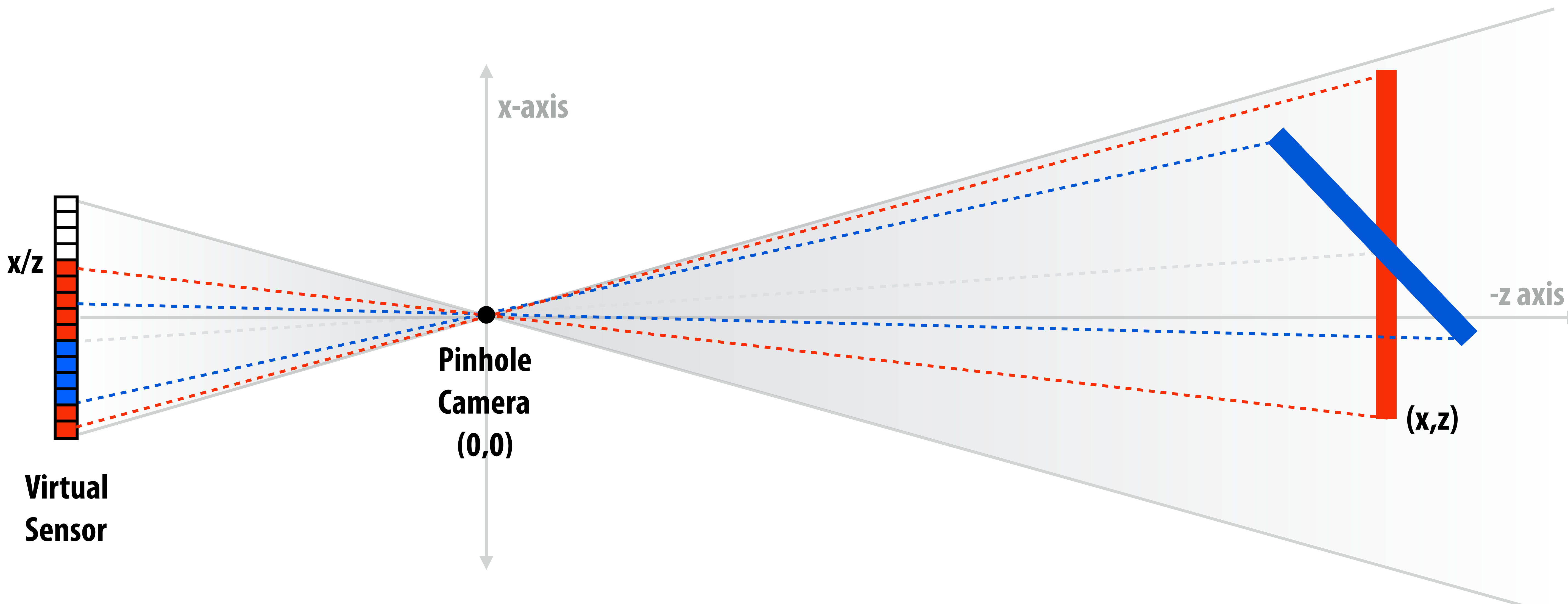


Sample

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

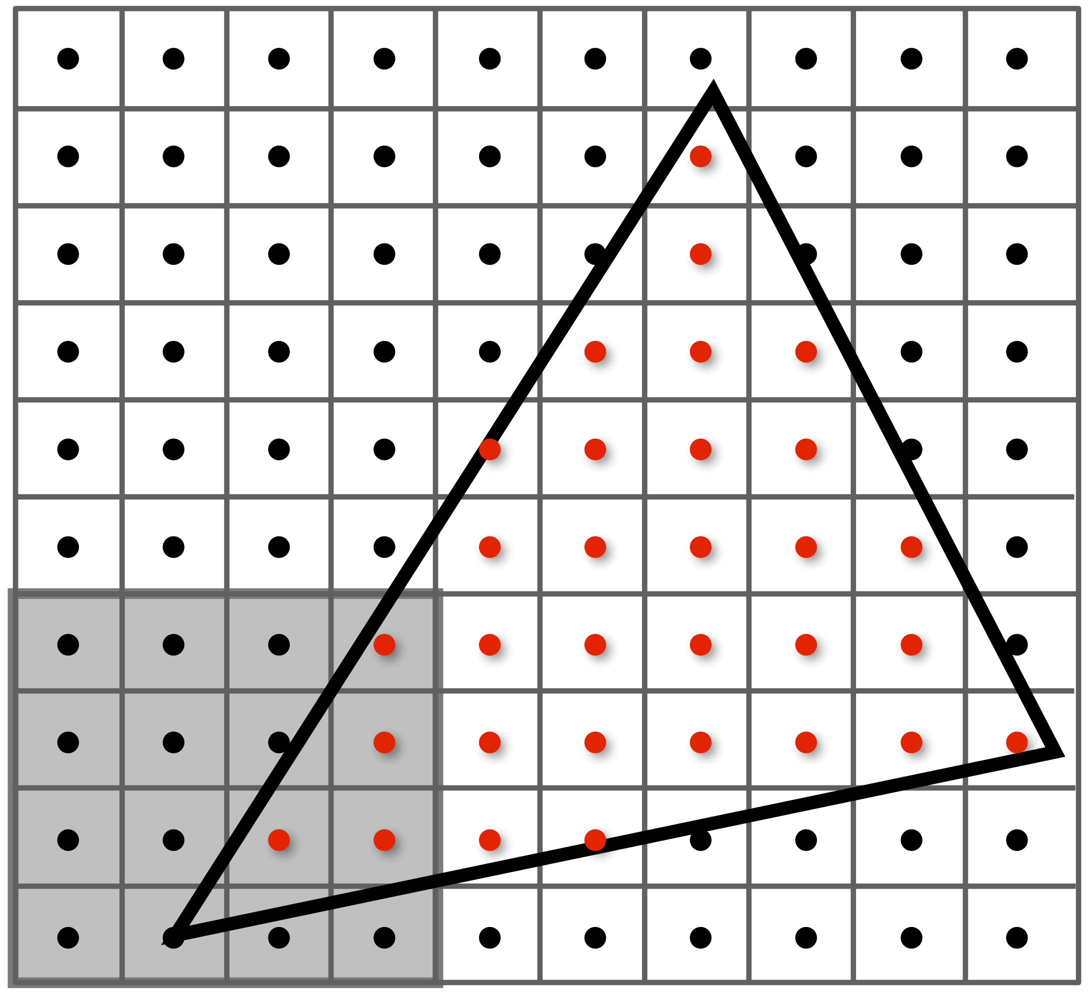
Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point)

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"

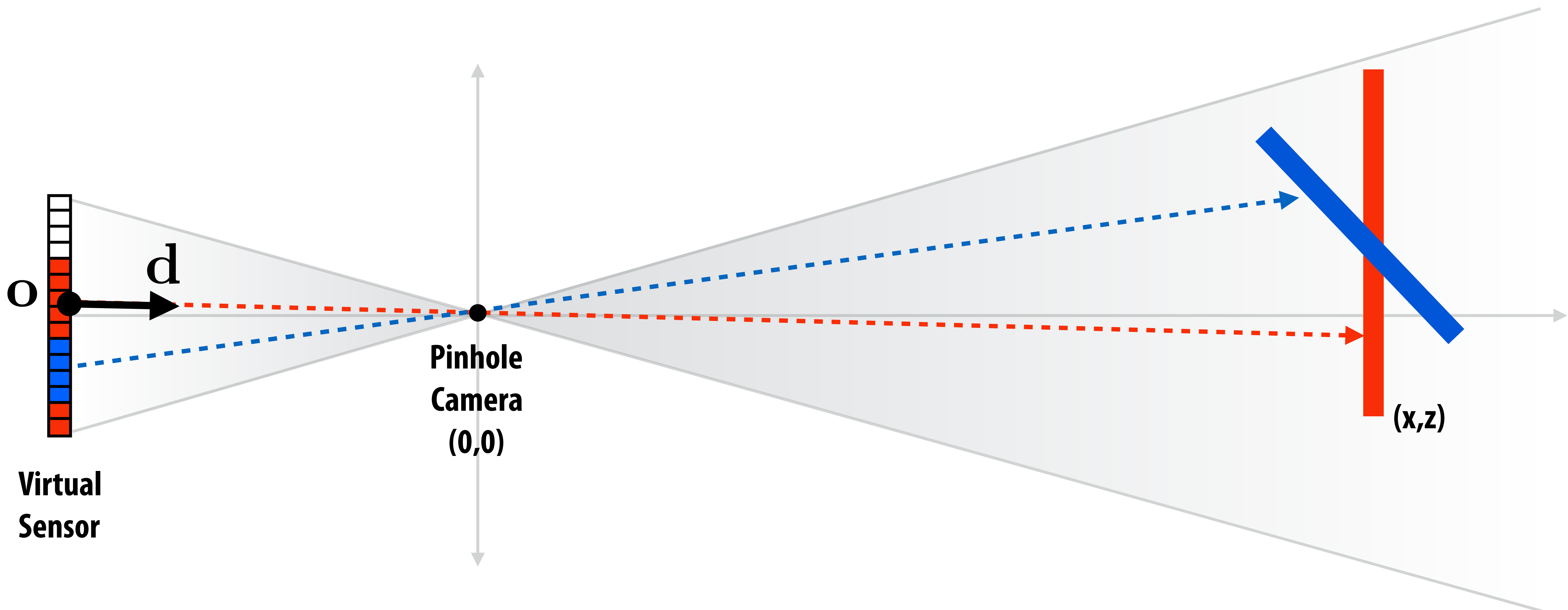
(finding the samples is relatively easy since they are distributed uniformly on screen)



Depth buffer example

The visibility problem (described differently)

- In terms of casting rays from a simulated camera:
  - What scene primitive is “hit” by a ray originating from a point on the virtual sensor and traveling through the aperture of the pinhole camera? (coverage)
  - What 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
```

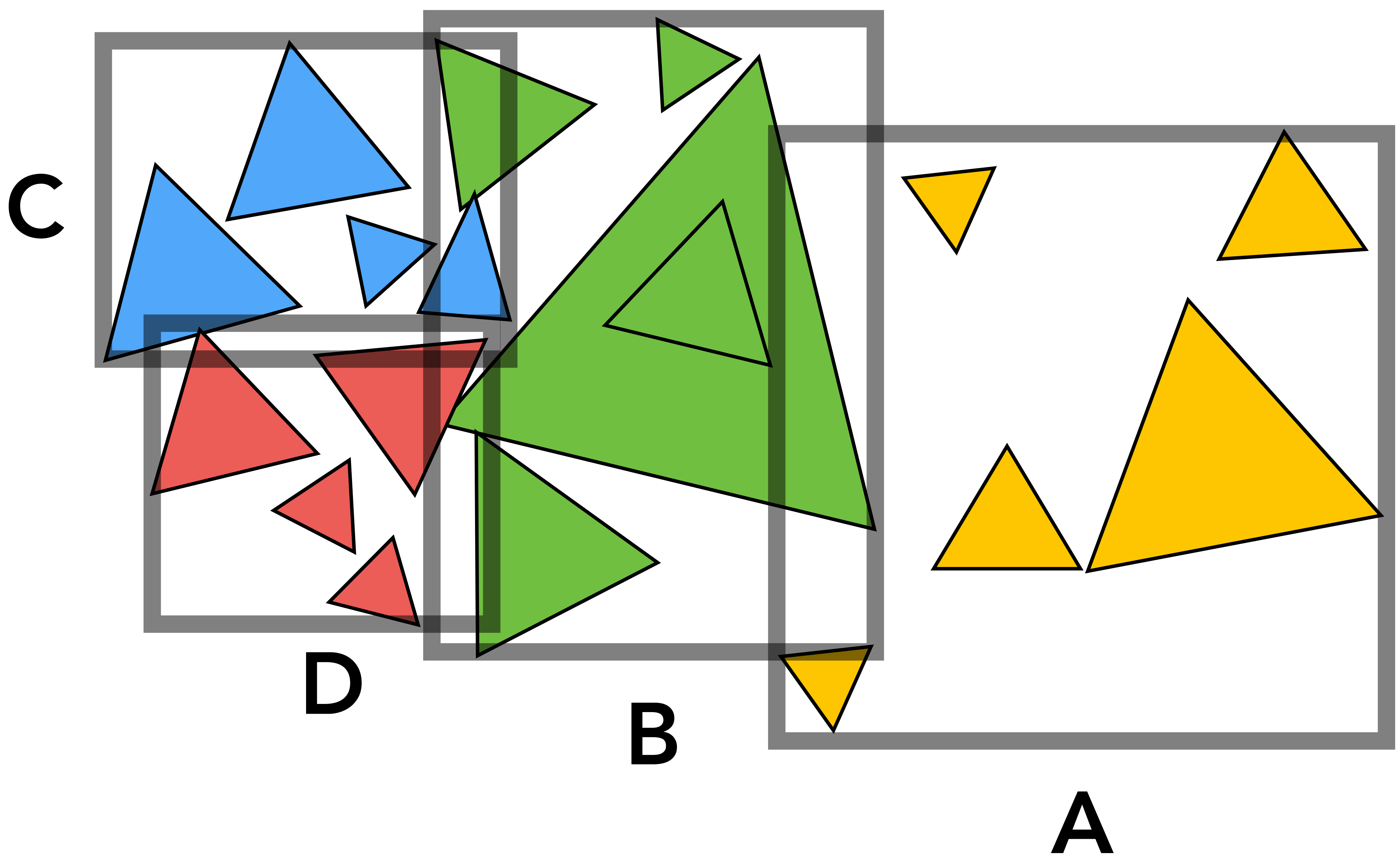
Compared to rasterization approach: just a reordering of the loops! (+ math in 3D)

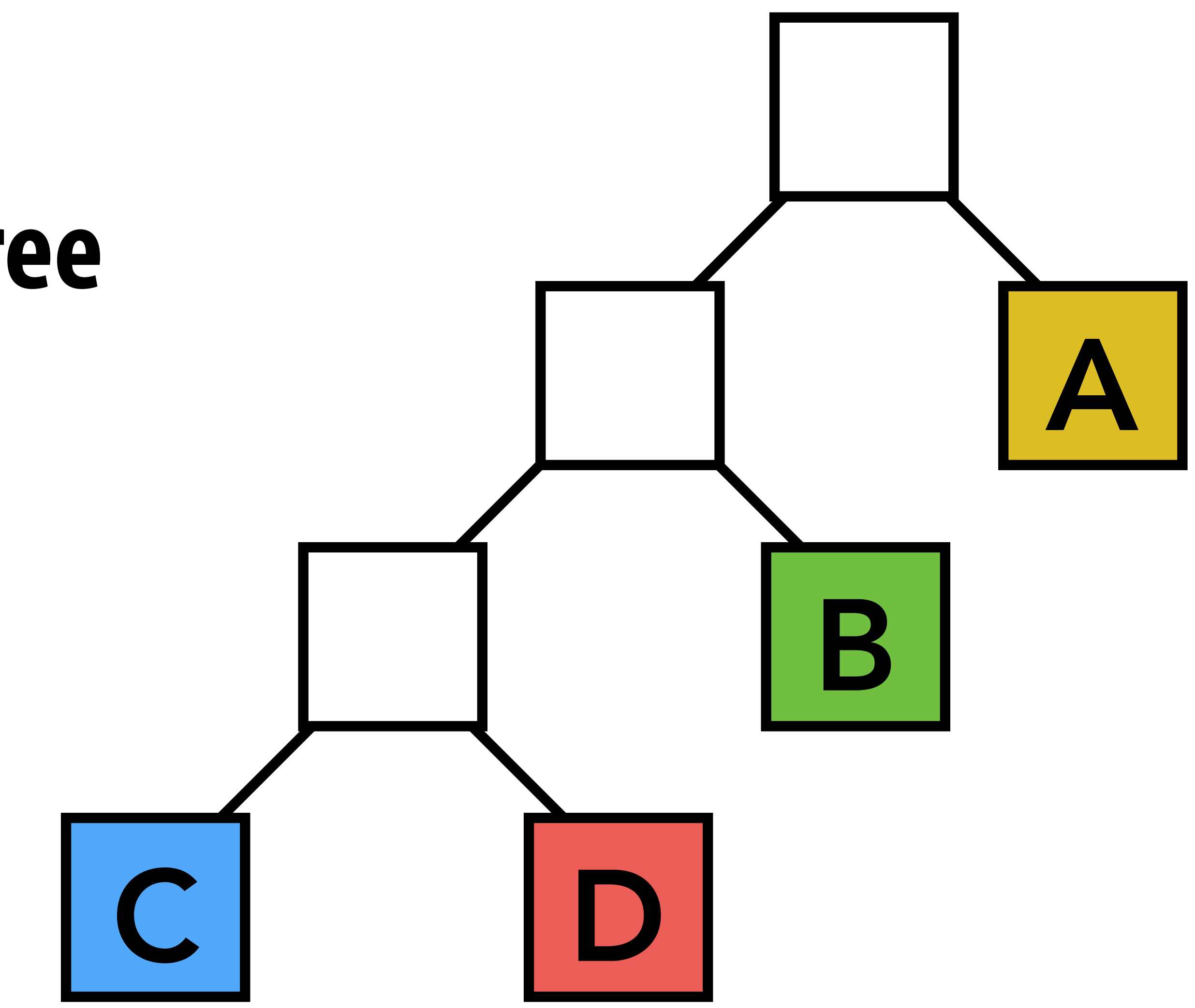
"Given a ray, find the closest triangle it hits"

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.)

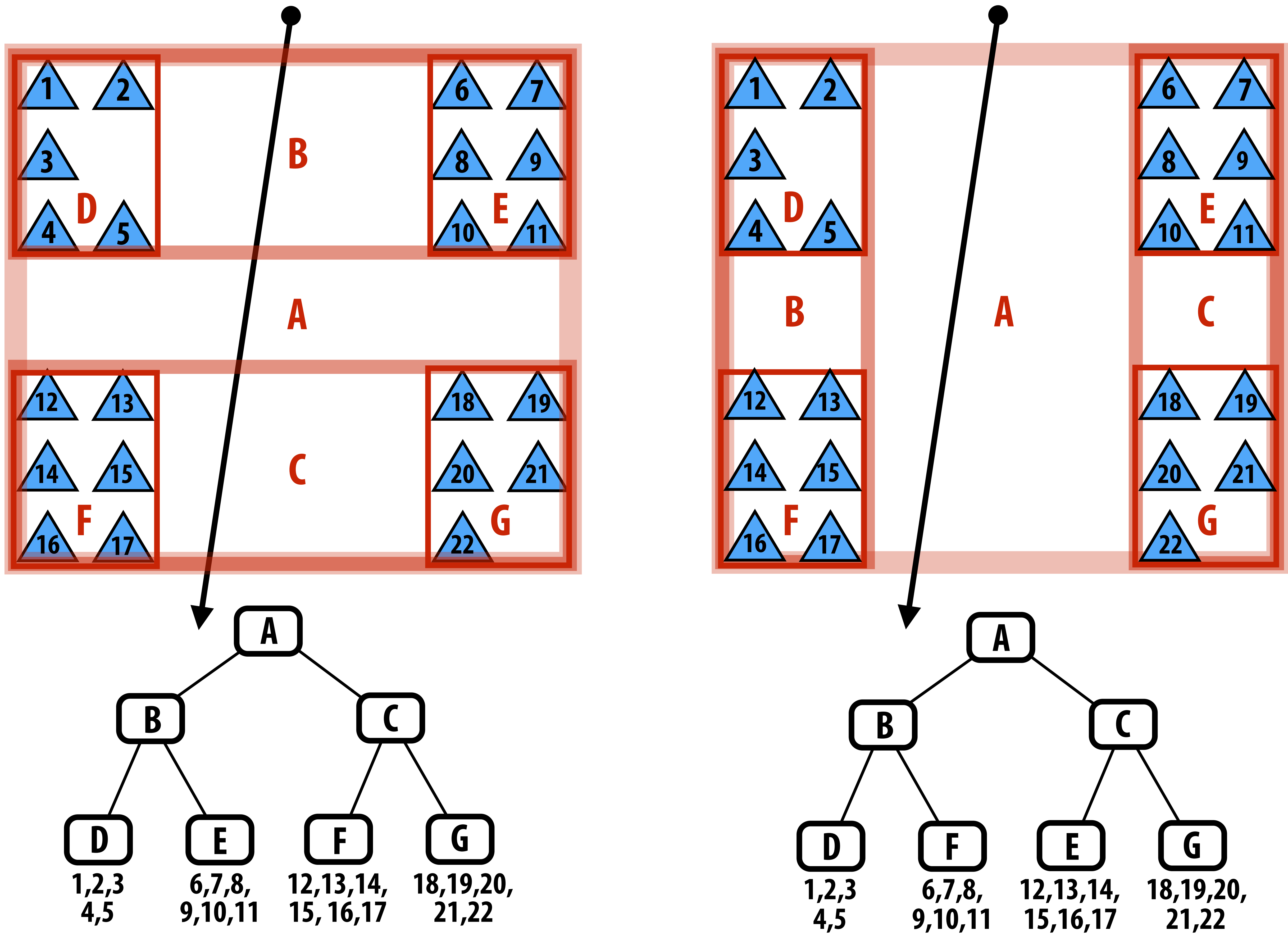
Bounding volume hierarchy (BVH)

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





Bounding volume hierarchy (BVH)



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

Is one BVH better than the other?

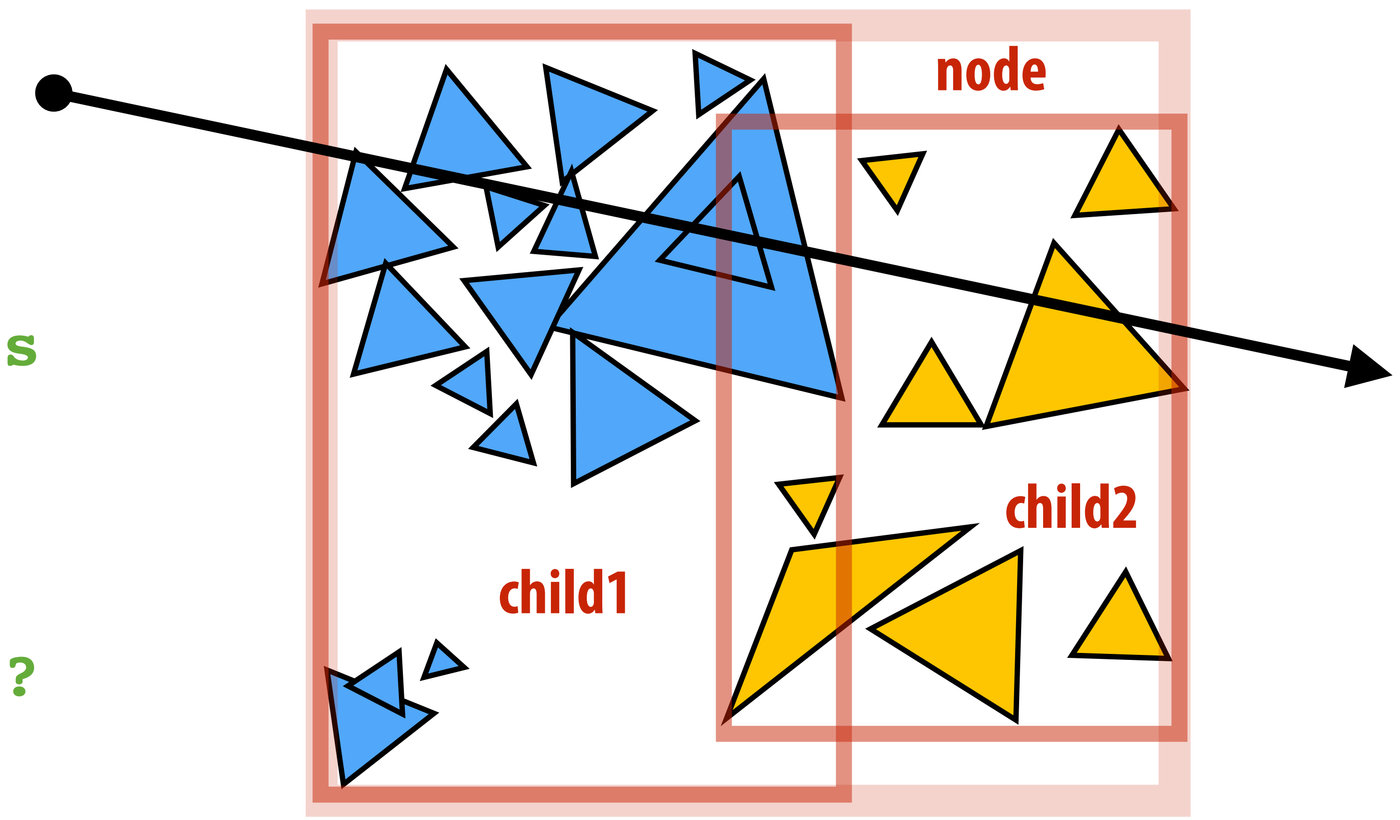
Ray-scene intersection using a BVH

```
struct BVHNode {
   bool leaf;  // true if node is a leaf
   BBox bbox;  // min/max coords of enclosed primitives
   BVHNode* child1; // "left" child (could be NULL)
   BVHNode* child2; // "right" child (could be NULL)
   Primitive* primList; // for leaves, stores primitives
};

struct HitInfo {
   Primitive* prim;  // which primitive did the ray hit?
   float t;          // at what t value along ray?
};

void find_closest_hit(Ray* ray, BVHNode* node, HitInfo* closest) {
   HitInfo hit = intersect(ray, node->bbox);  // test ray against node's bounding box
   if (hit.prim == NULL || hit.t > closest.t))
      return; // don't update the hit record

   if (node->leaf) {
      for (each primitive p in node->primList) {
         hit = intersect(ray, p);
         if (hit.prim != NULL && hit.t < closest.t) {
            closest.prim = p;
            closest.t = t;
         }
      }
   } else {
      find_closest_hit(ray, node->child1, closest);
      find_closest_hit(ray, node->child2, closest);
}}
```



How could this occur?

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 (need *random access to regular structure of fixed size*)**

- **Ray casting:**
  - ***Stream over screen samples*** **(rays)**
    - **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)**

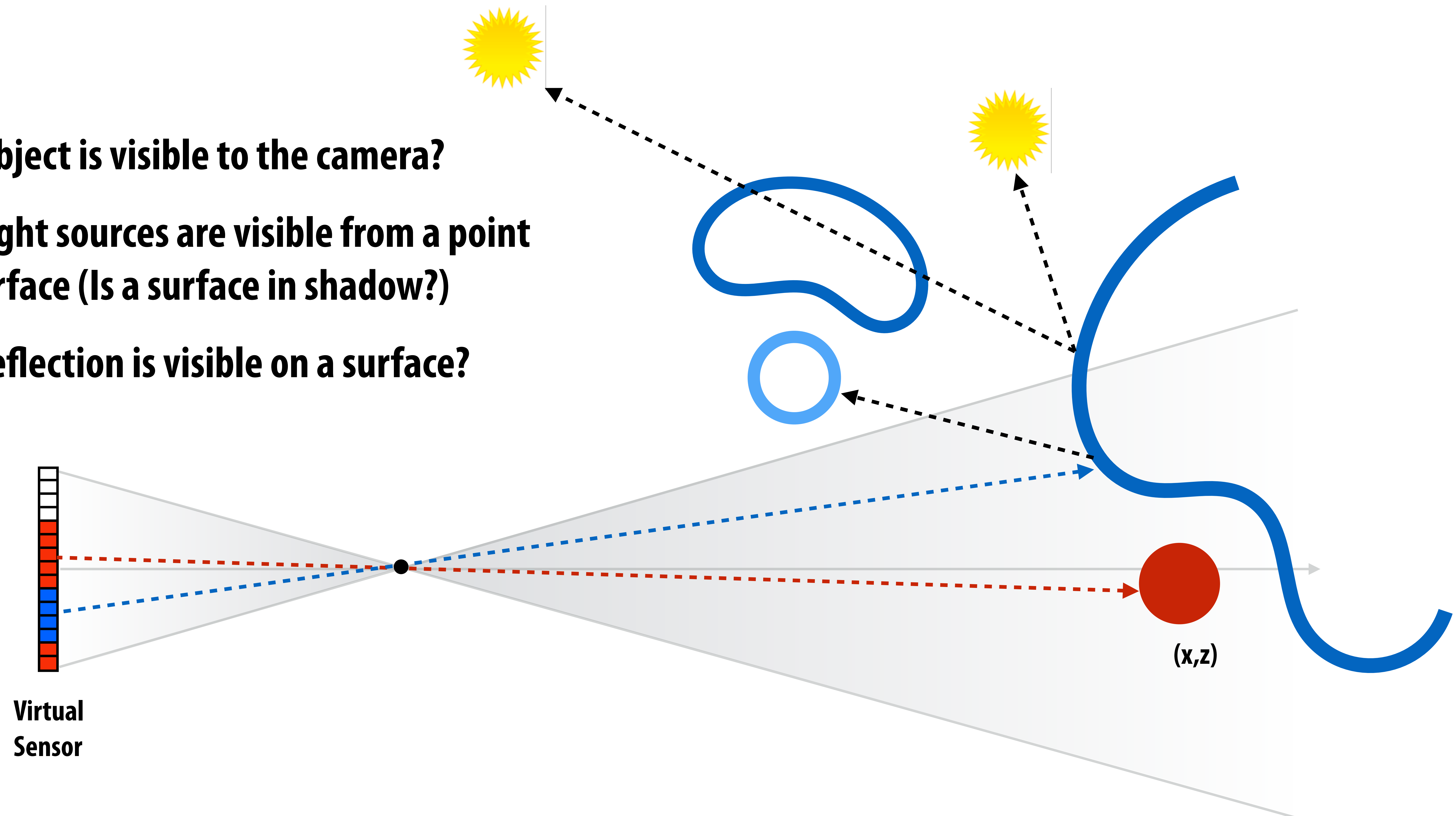
Ray-scene intersection is a general visibility primitive

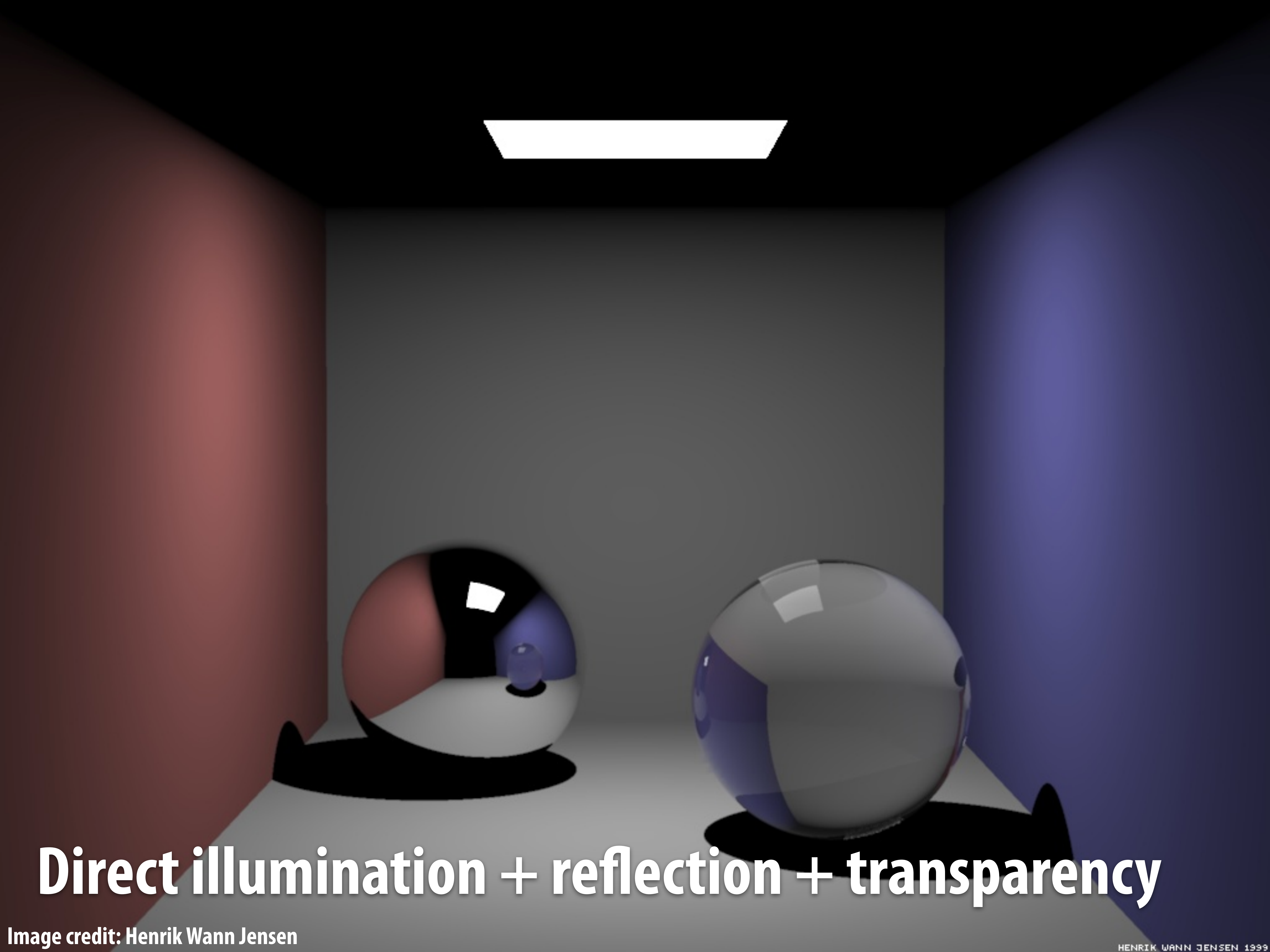
What object is visible along this ray?

What object is visible to the camera?

What light sources are visible from a point on a surface (Is a surface in shadow?)

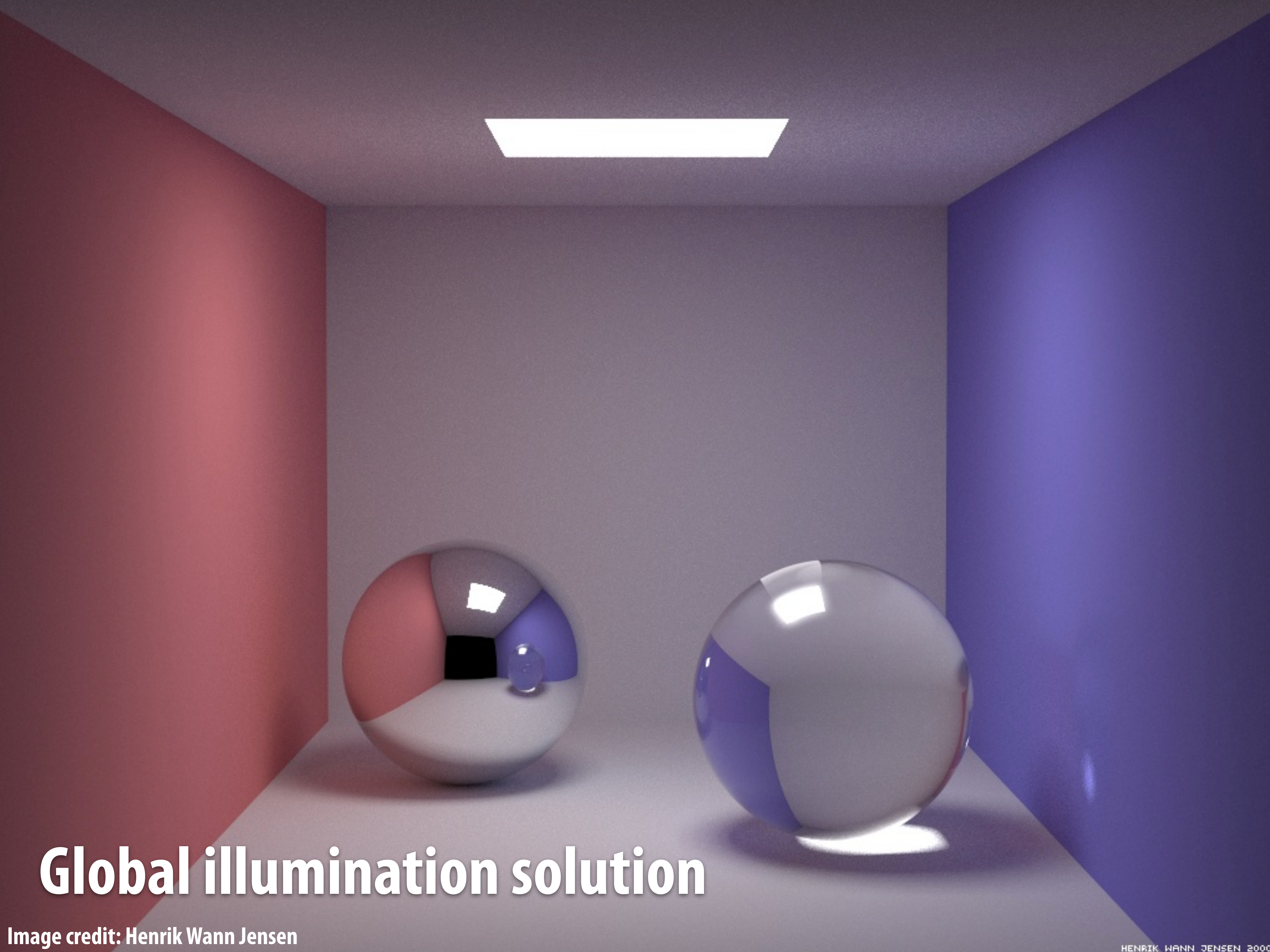
What reflection is visible on a surface?





Direct illumination + reflection + transparency

Image credit: Henrik Wann Jensen



Global illumination solution

Image credit: Henrik Wann Jensen


p
Direct illumination


p
Sixteen-bounce global illumination

Sampling light paths

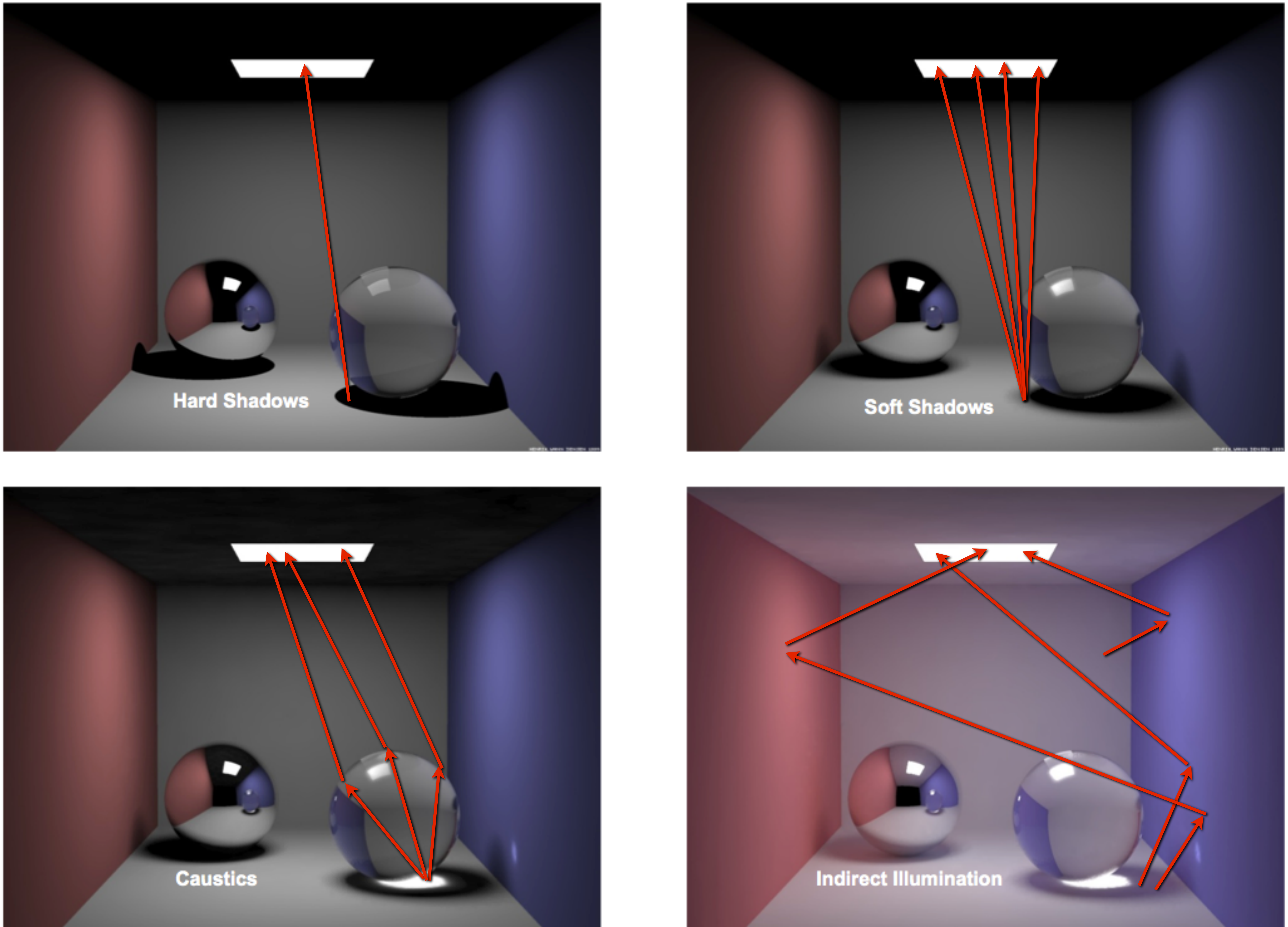


Image credit: Wann Jensen, Hanrahan

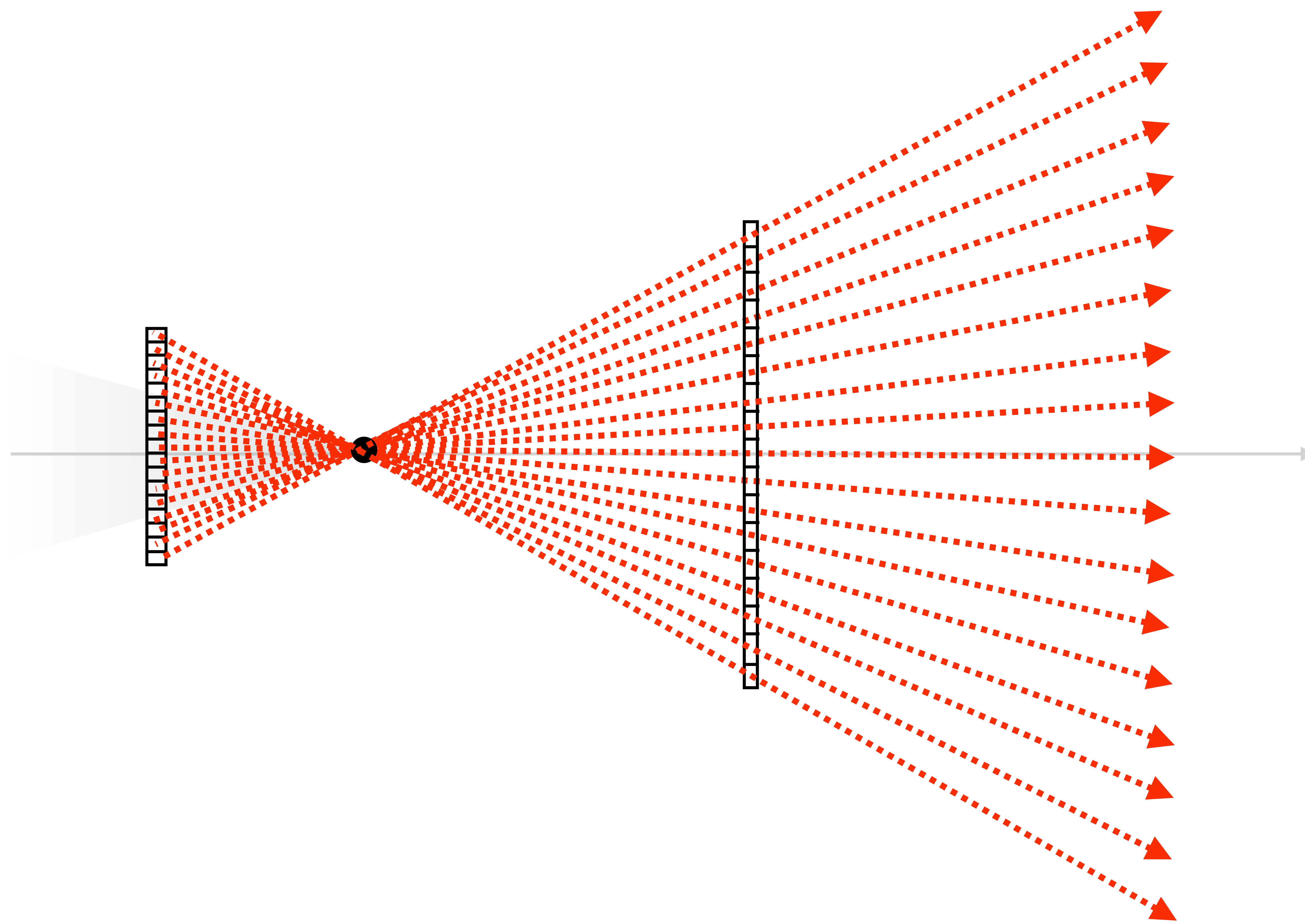
Another way to think about rasterization

- **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)

Another way to think about rasterization

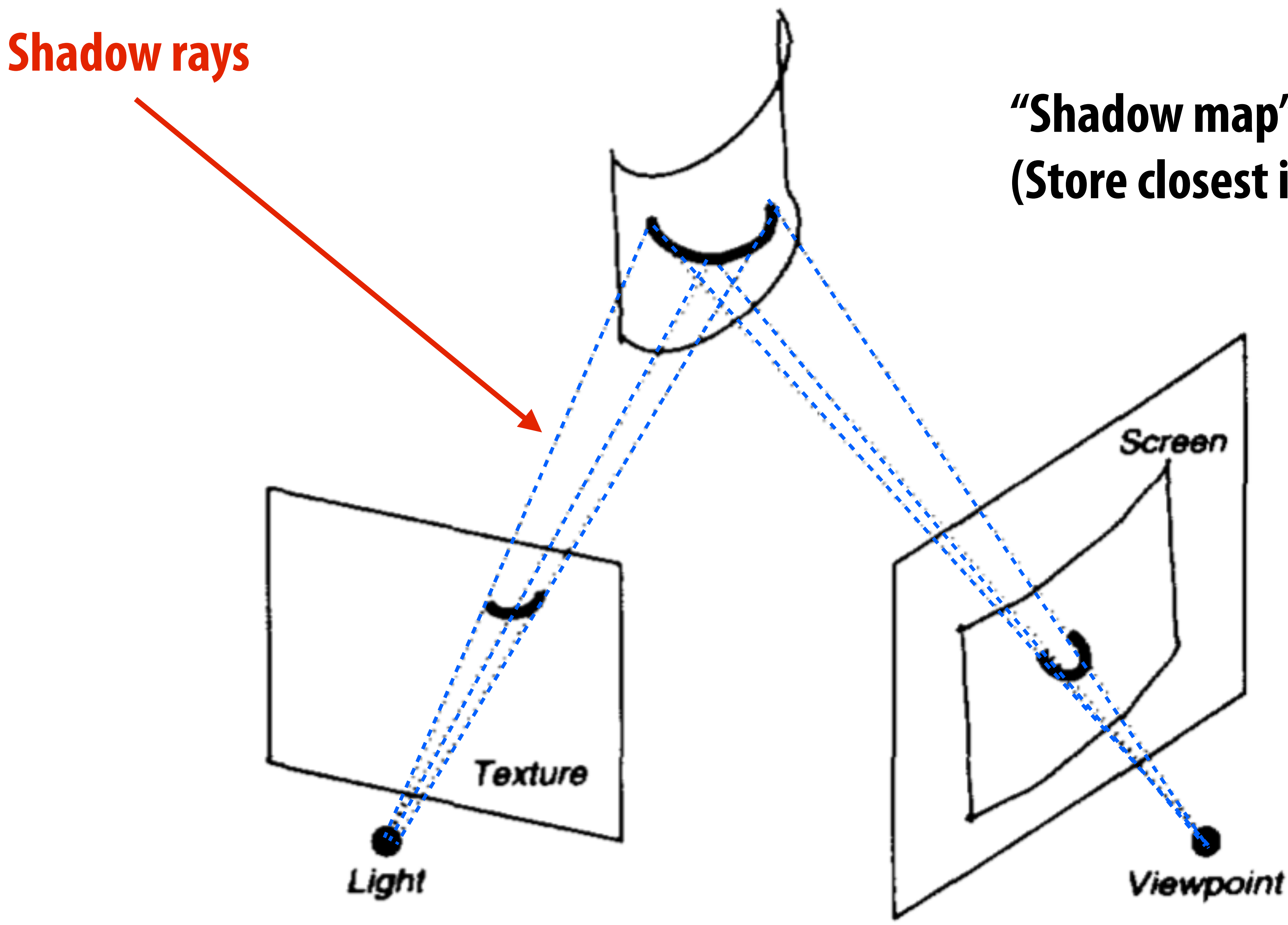
- Rasterization is a efficient implementation of ray casting where:
  - Scene intersection results for a batch of rays are computed at a time
  - All rays originate from same origin
  - Projection of rays distributed uniformly in plane of projection

    (Note: not uniform distribution in angle… angle between rays is smaller away from view direction)



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



"Shadow map" = depth map from perspective of a point light.
(Store closest intersection along each shadow ray in a texture map)

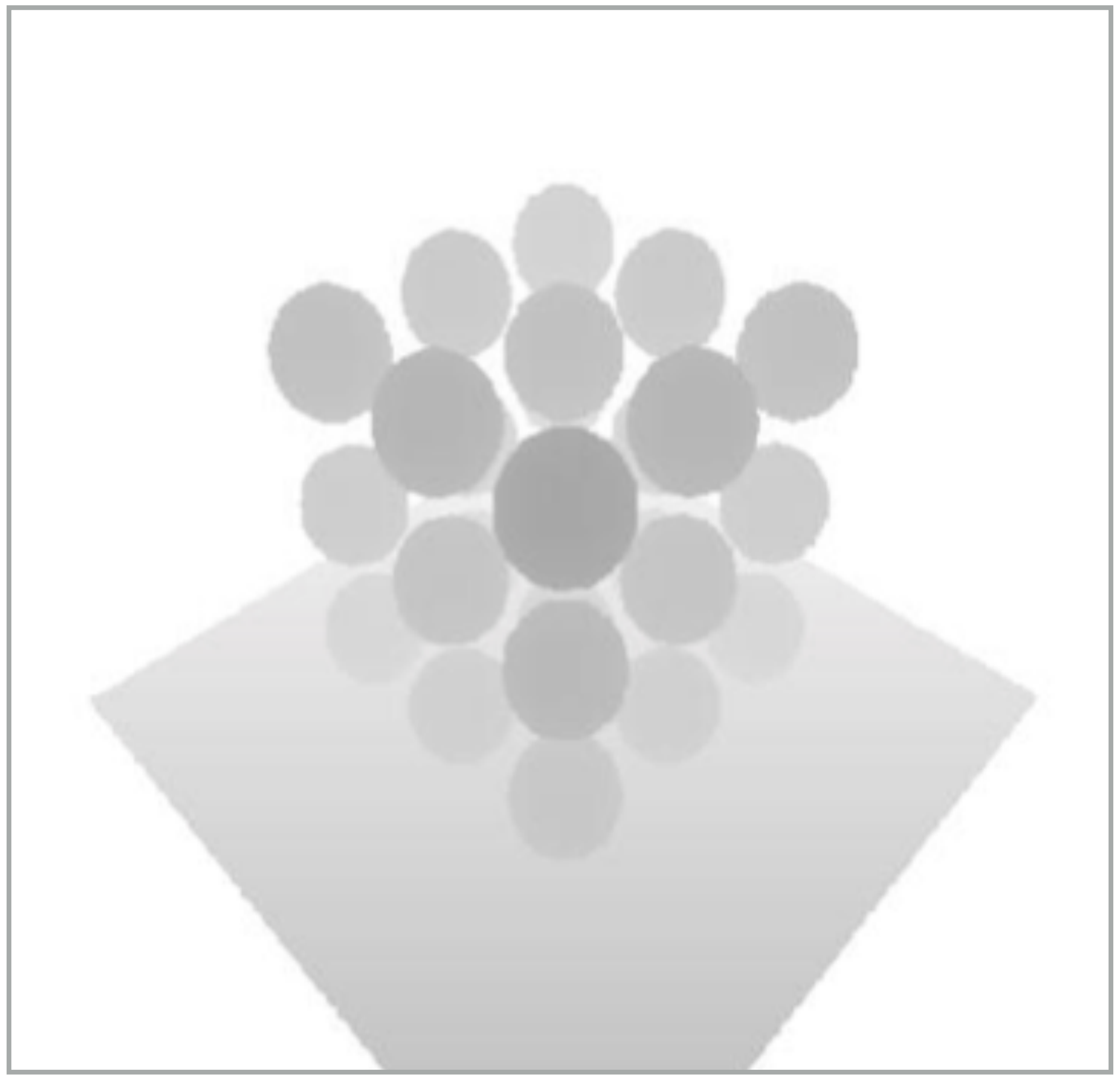
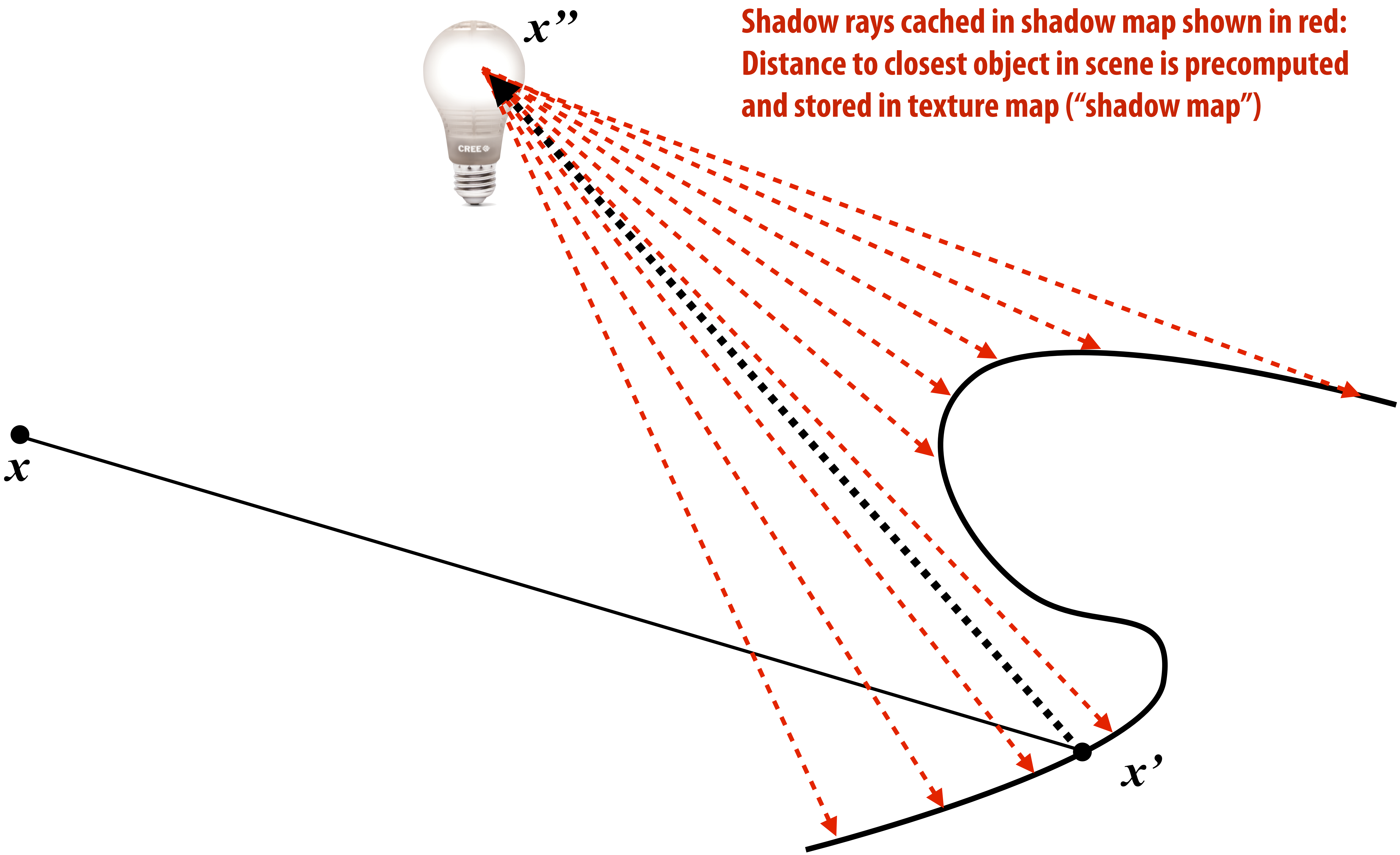
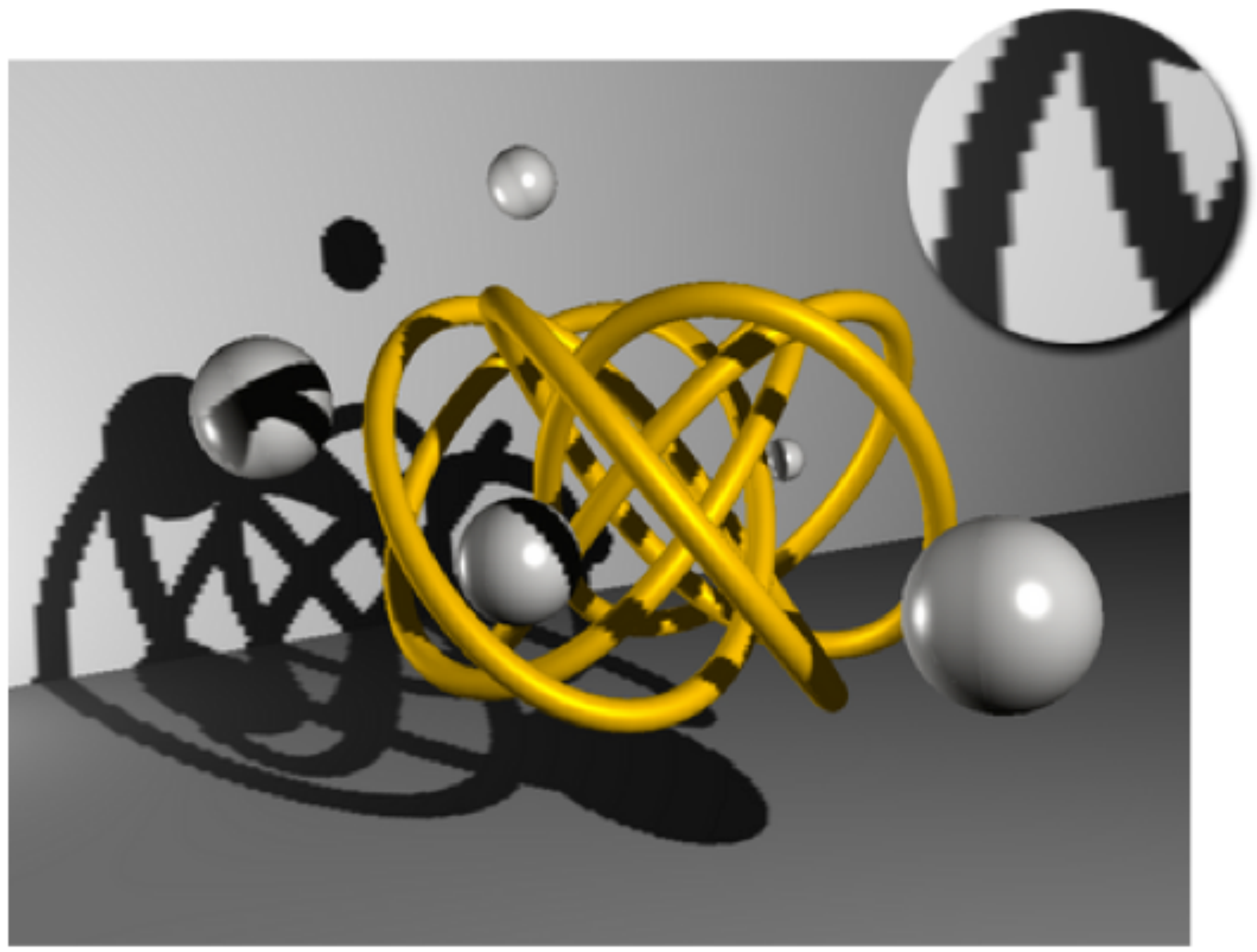


Image credits: Segal et al. 92, Cass Everitt

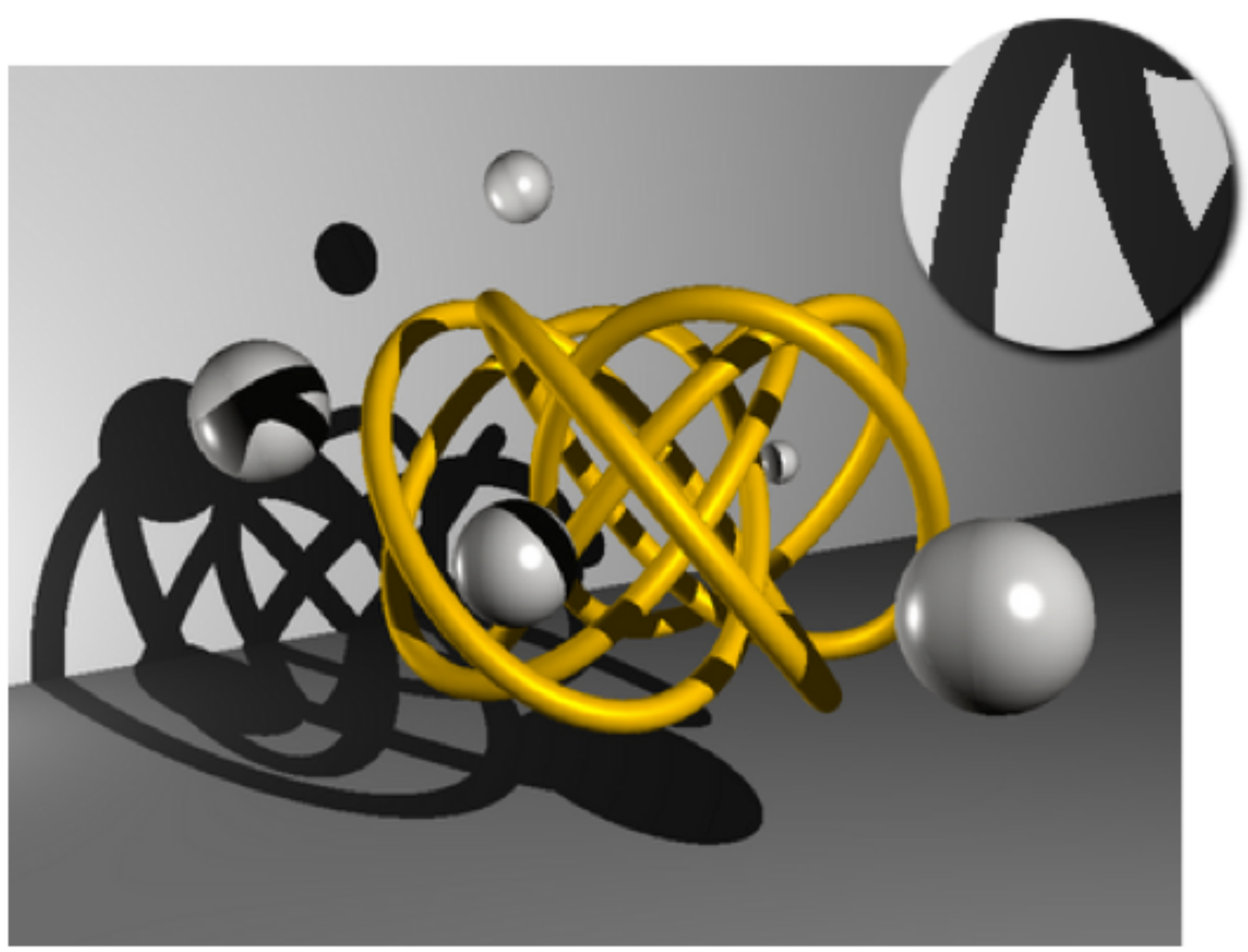
Result of shadow texture lookup approximates visibility result when shading fragment at x’



Shadow aliasing due to shadow map undersampling



Shadows computed using shadow map



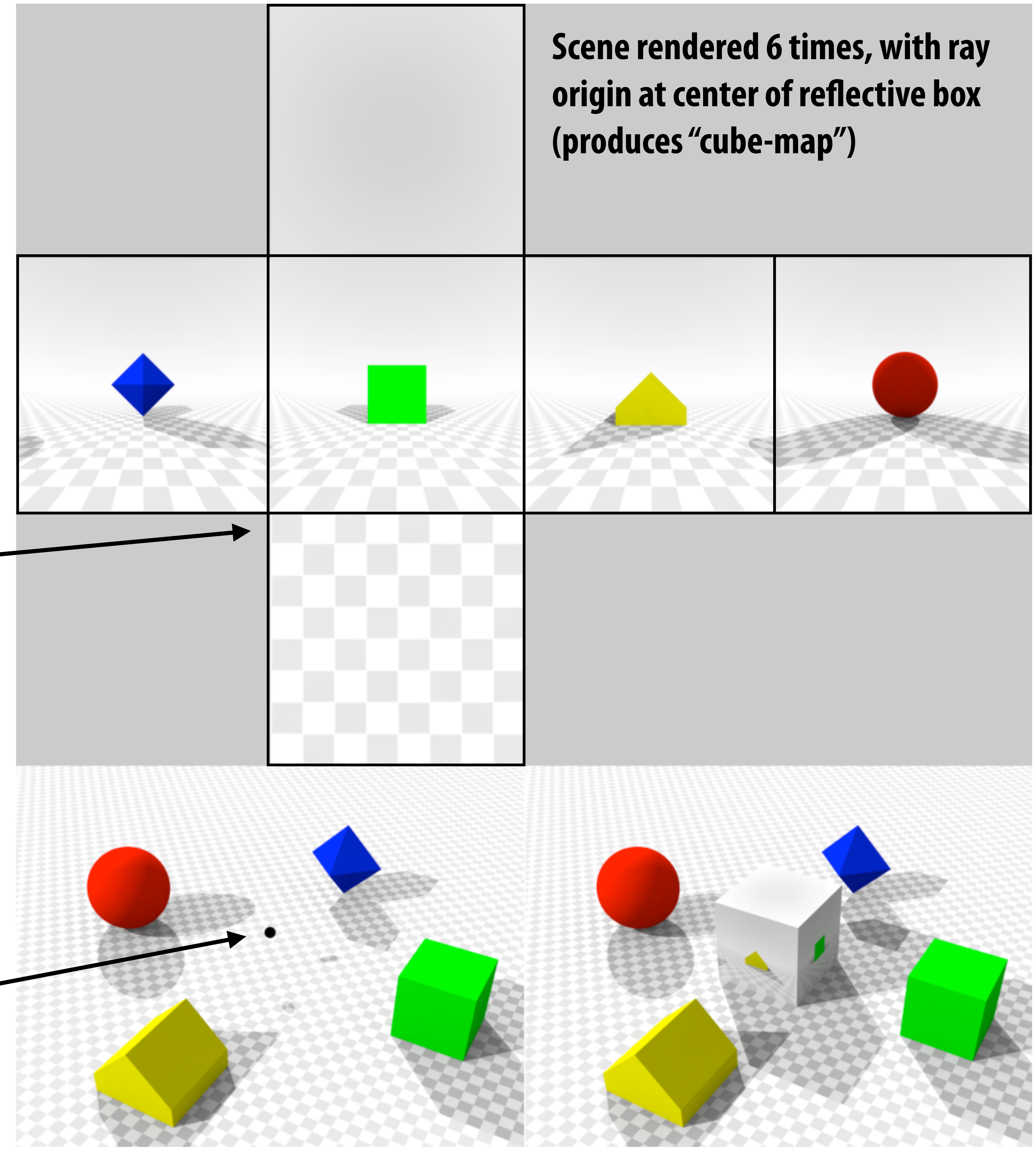
Correct hard shadows
(result from computing v(x′,x″) directly using ray tracing)

Image credit: Johnson et al. TOG 2005

Rasterization: ray origin need not be camera position

Environment mapping:
place ray origin at reflective object

Yields <u>approximation</u> 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

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

Why real-time ray tracing?

Why ray tracing

- Accurate lighting/shading effects
  - Correct reflections from surfaces surfaces
  - Correct shadows (no aliasing)
  - Soft shadows
  - Ambient occlusion
  - "Global illumination" (multiple bounces)

- Software simplicity
  - Many effects created from a single primitive (traceRay())
  - This is was the "killer reason" to move to ray tracing for film rendering

Technologies that are making RTRT possible

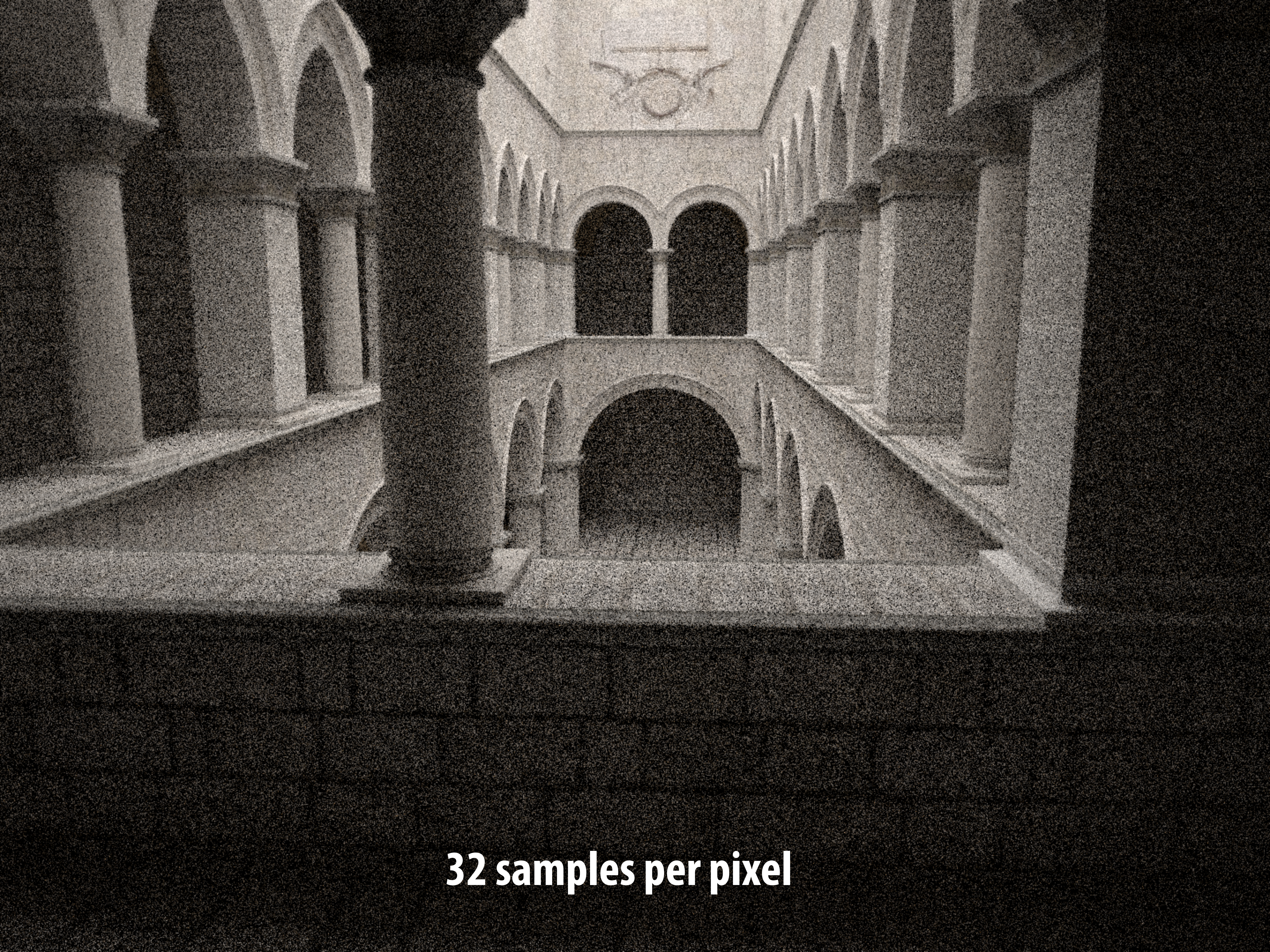
- **Better algorithms: fast parallel BVH construction and traversal algorithms (SIGGRAPH/HPG circa 2010)**

- **GPU hardware evaluation:**
  - **Faster GPUs, sufficient amounts of DRAM**
  - **Increasingly flexible aspects of traditional GPU pipeline (bindless textures/resources)**

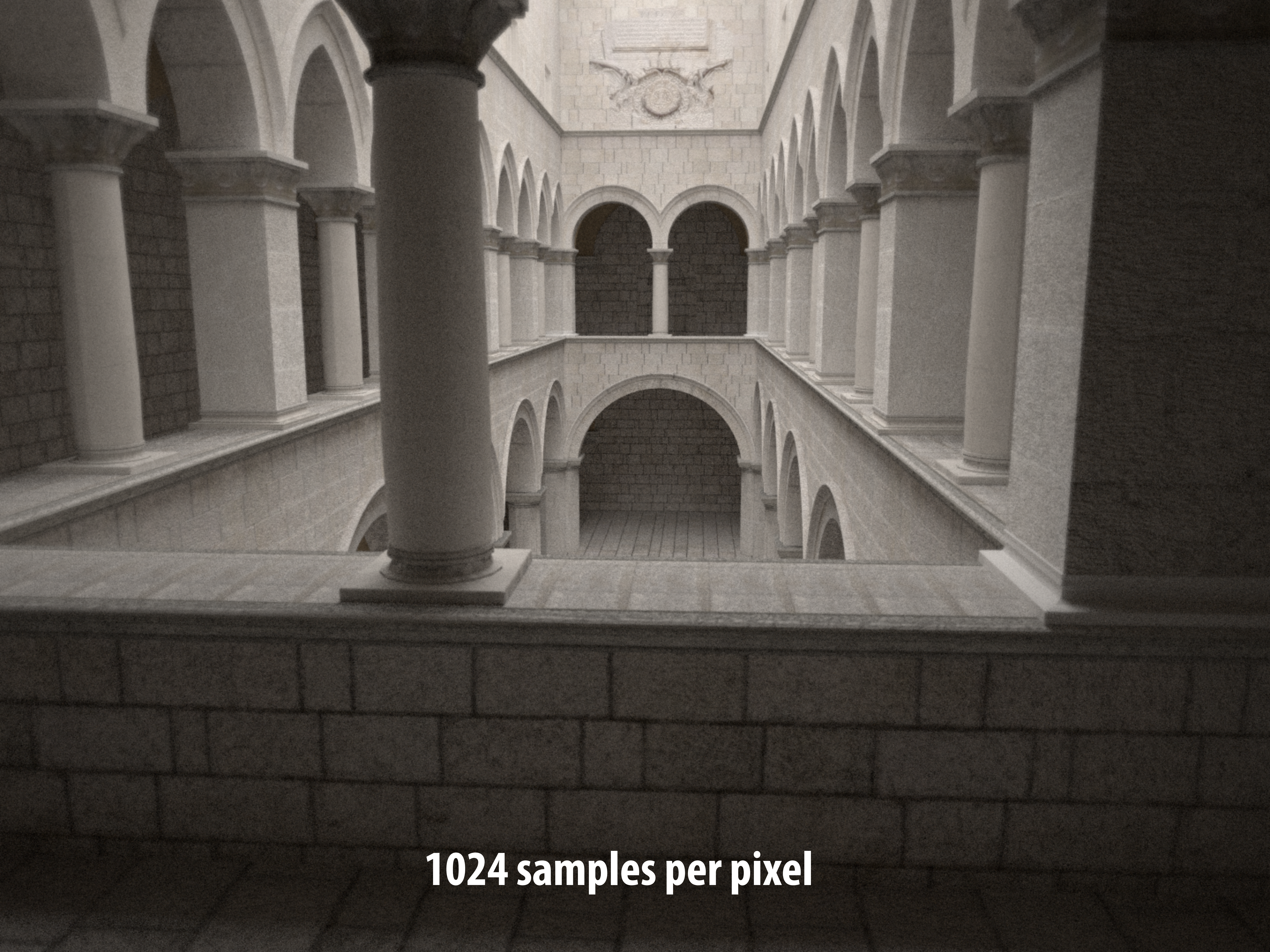
- **DNN-based image denoising**
  - **Can make plausible images using small number of rays per pixel**
  - **Make use of DNN hardware acceleration**

Sampling noise

One sample per pixel

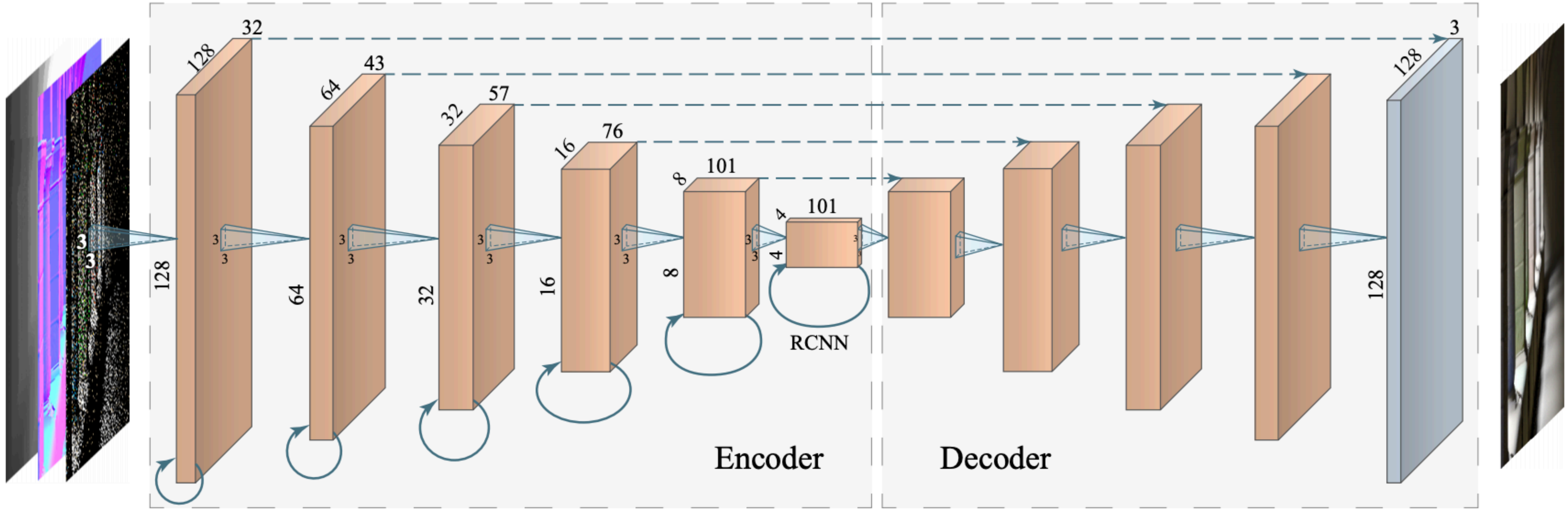


32 samples per pixel

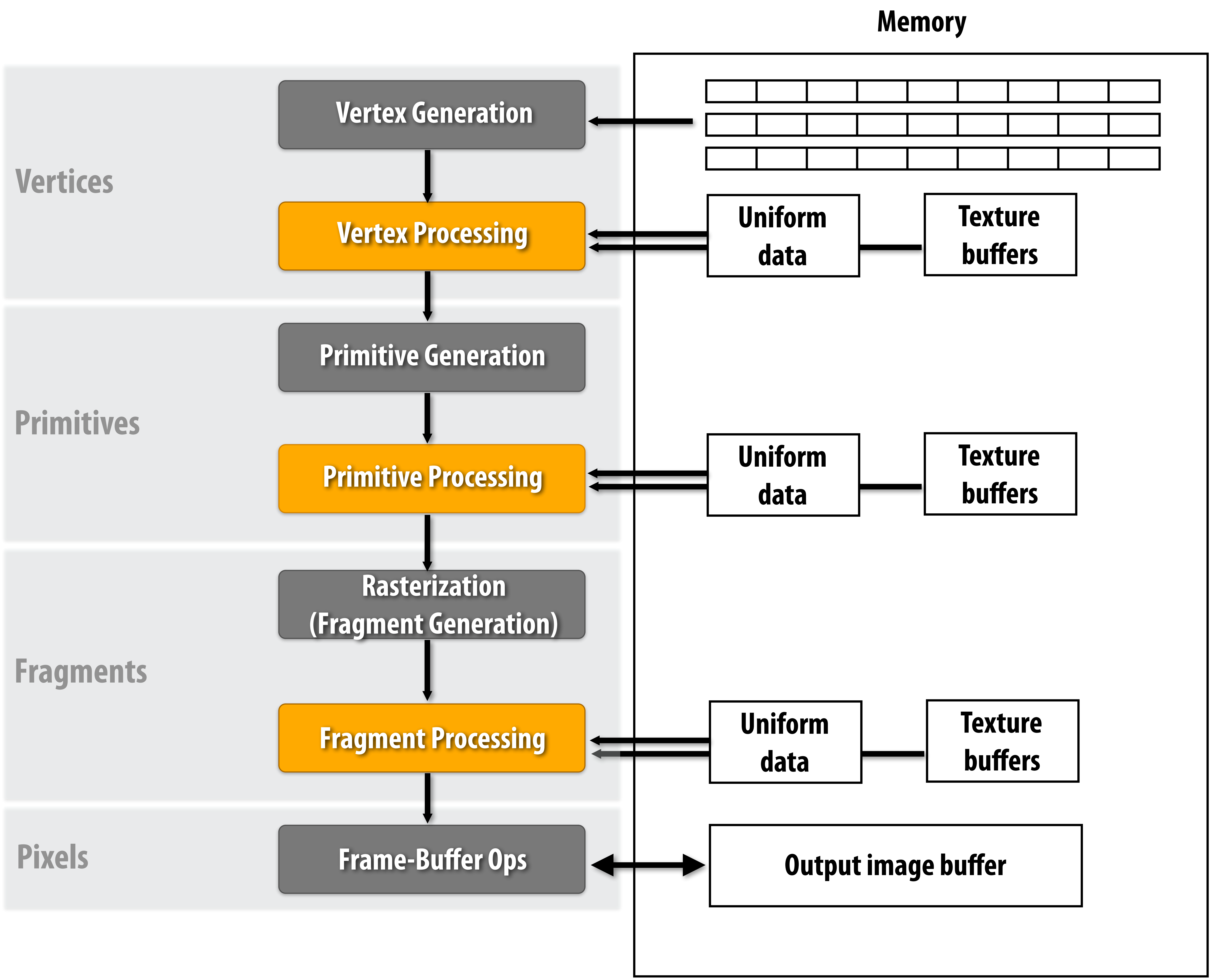

1024 samples per pixel

Example: NVIDIA Optix denoiser

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



Traditional graphics pipeline

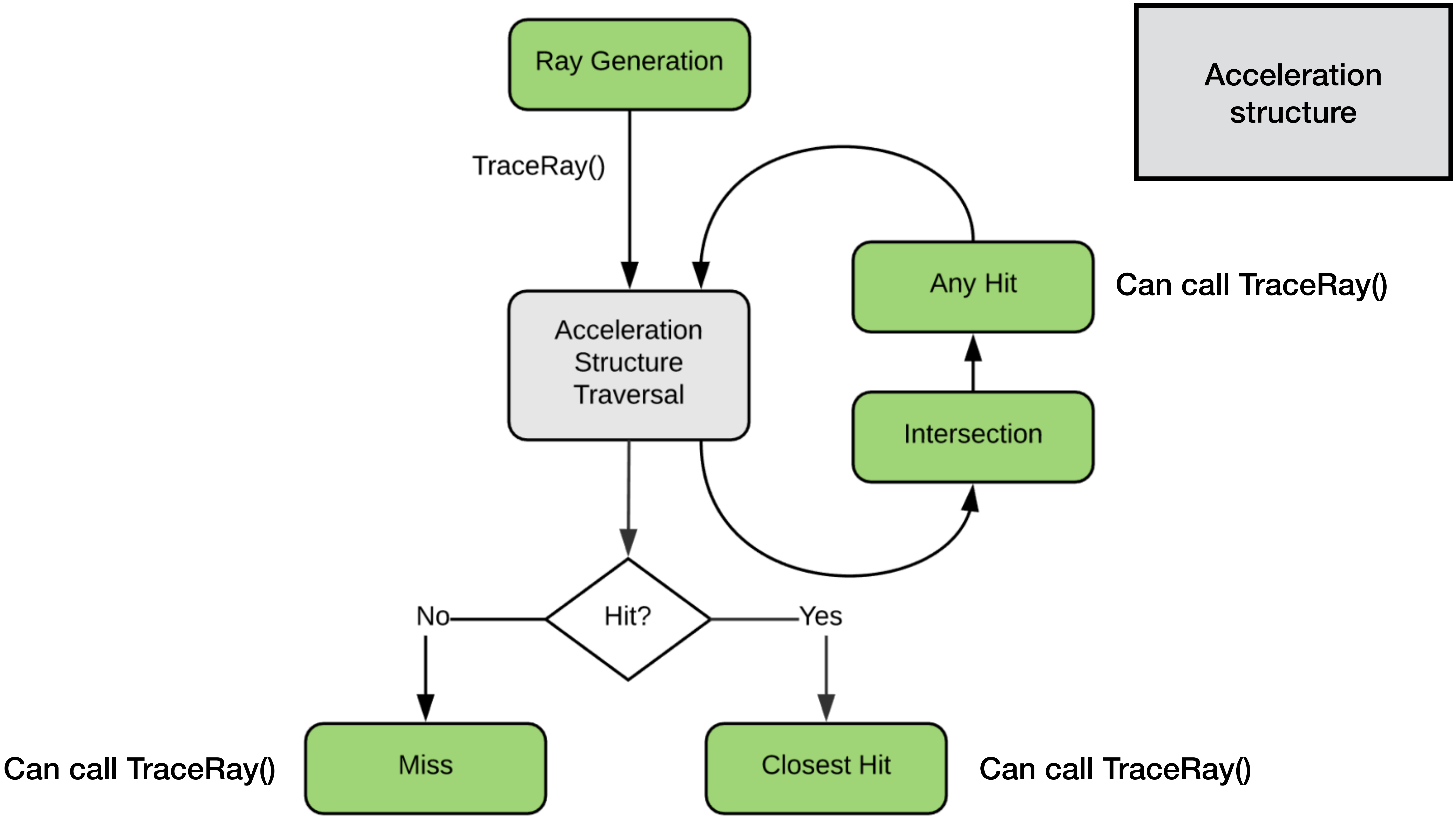


Keep in mind

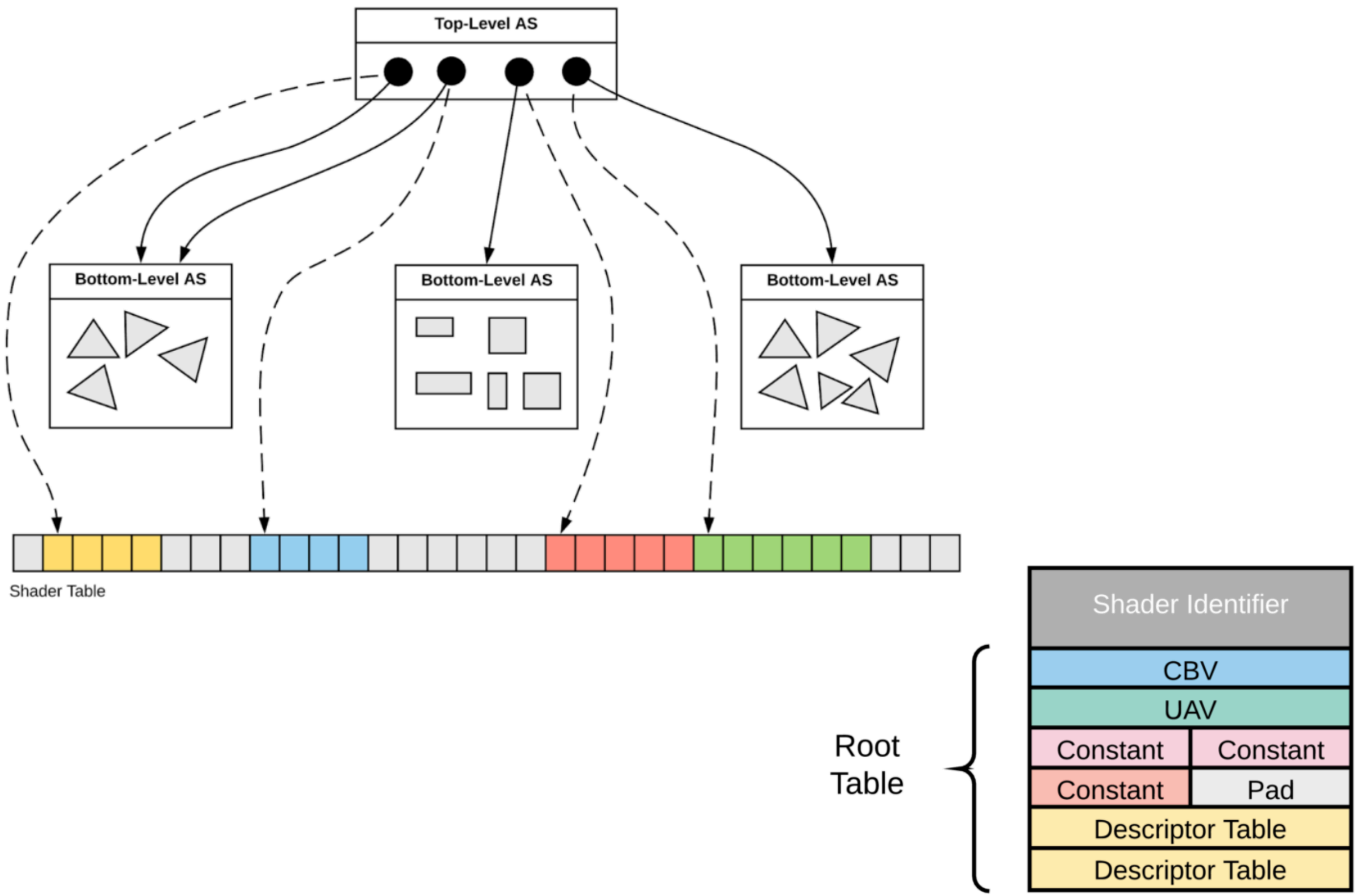
- An application developer has always been able to write a ray tracer in CUDA
- So the ability to use a GPU to perform ray tracing is nothing new
- So why a new API?

D3D12's DXR ray tracing "stages"

- **TraceRay is a blocking function**



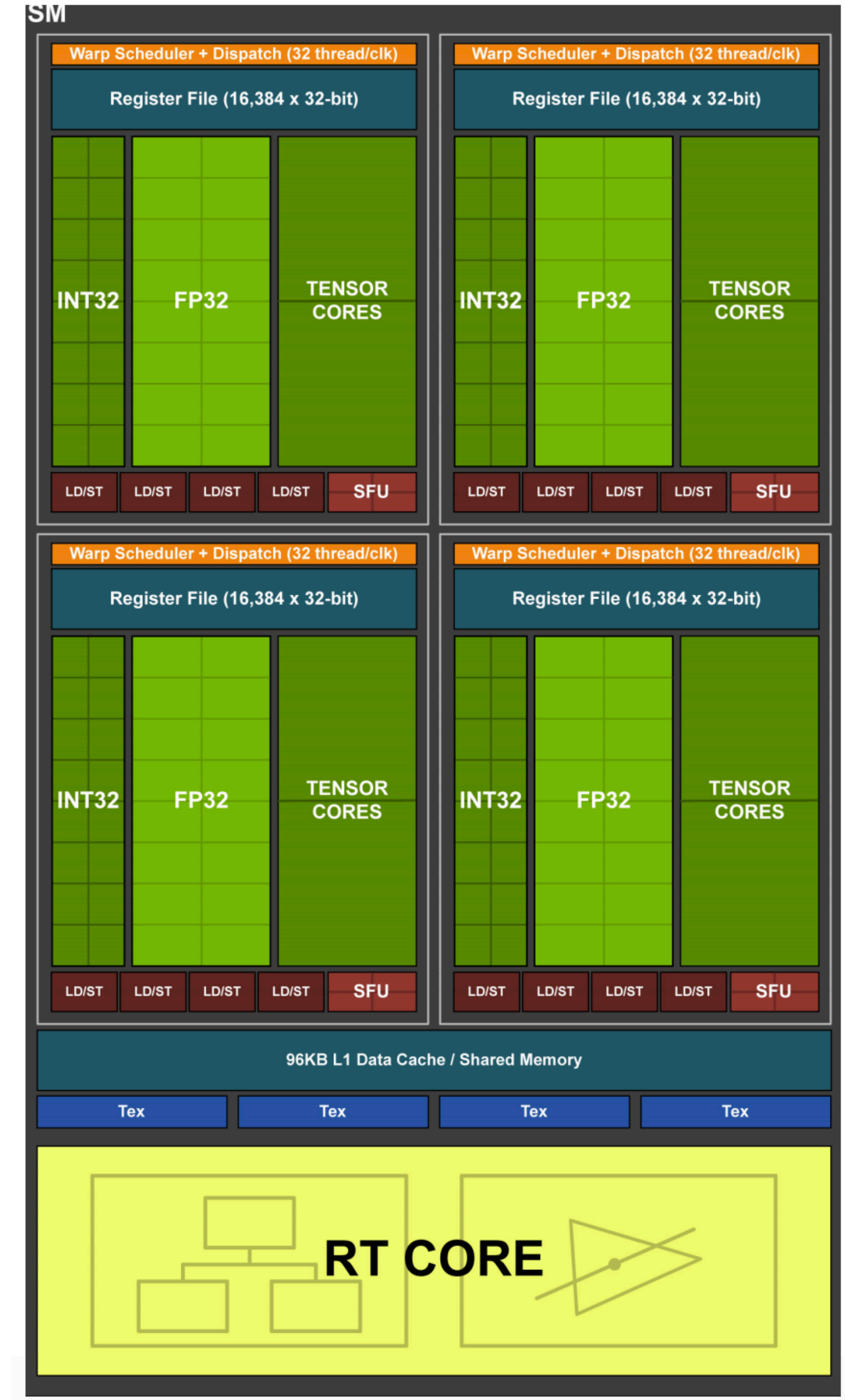
GPU understands format of BVH acceleration structure and "shader table"



Surprising synergies

- New GPU hardware for ray-tracing operations
- But ray tracing still too expensive for noise-free images in real-time
- Tensor core: specialized hardware for accelerated DNN computations

  (that can be used to perform sophisticated denoising)



Summary

- Ray tracing is an elegant, general purpose algorithm for rendering realistic images
  - Simple: single operation for many effects

- Challenge = high cost: must trace large number of rays per pixel to reduce noise in rendered images

- Solutions:
  - Hardware for ray-tracing specific operations
  - Hardware for DNN acceleration used to implement new fast denoising operations