Lecture 13:

Modern Real-Time Rendering Techniques

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2026

This image is rendered in real-time on a modern GPU

Image credit: DigitalDreamsyt. (Cyberpunk 2077)

So was this…

Supercomputing for games

NVIDIA Titan X GPU

(~ 7 TFLOPs fp32)Tesla generation NV chip ~ ASCI Red

NVIDIA Founder's Edition RTX 4090 GPU

~ 82 TFLOPs fp32 *

* Doesn't include additional 190 TFLOPS of ray tracing compute and 165 TFLOPS of fp15 DNN compute

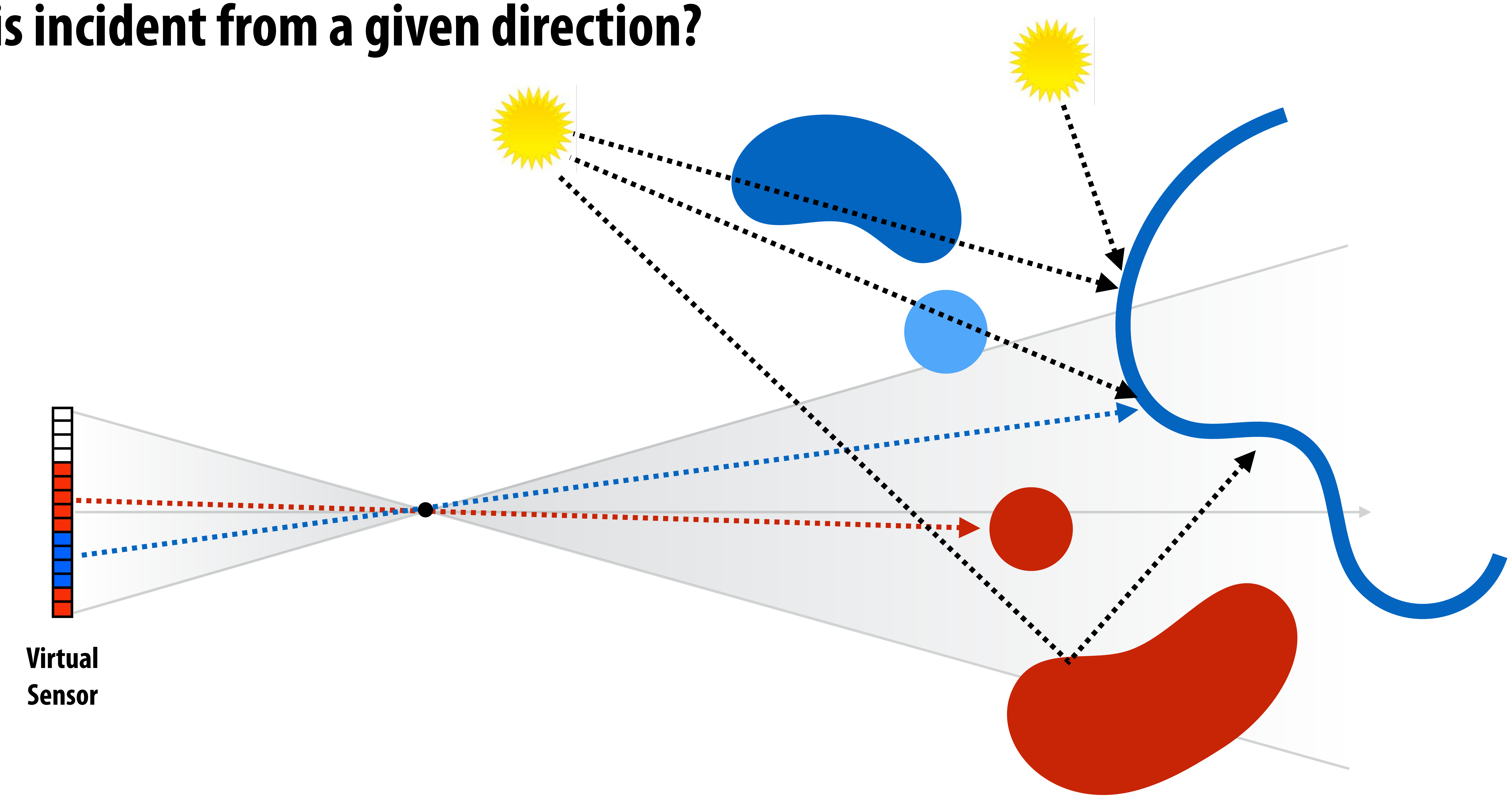
Specialized processors for performing graphics computations.

Last couple of lectures: ray-scene queries

What object is visible to the camera?

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

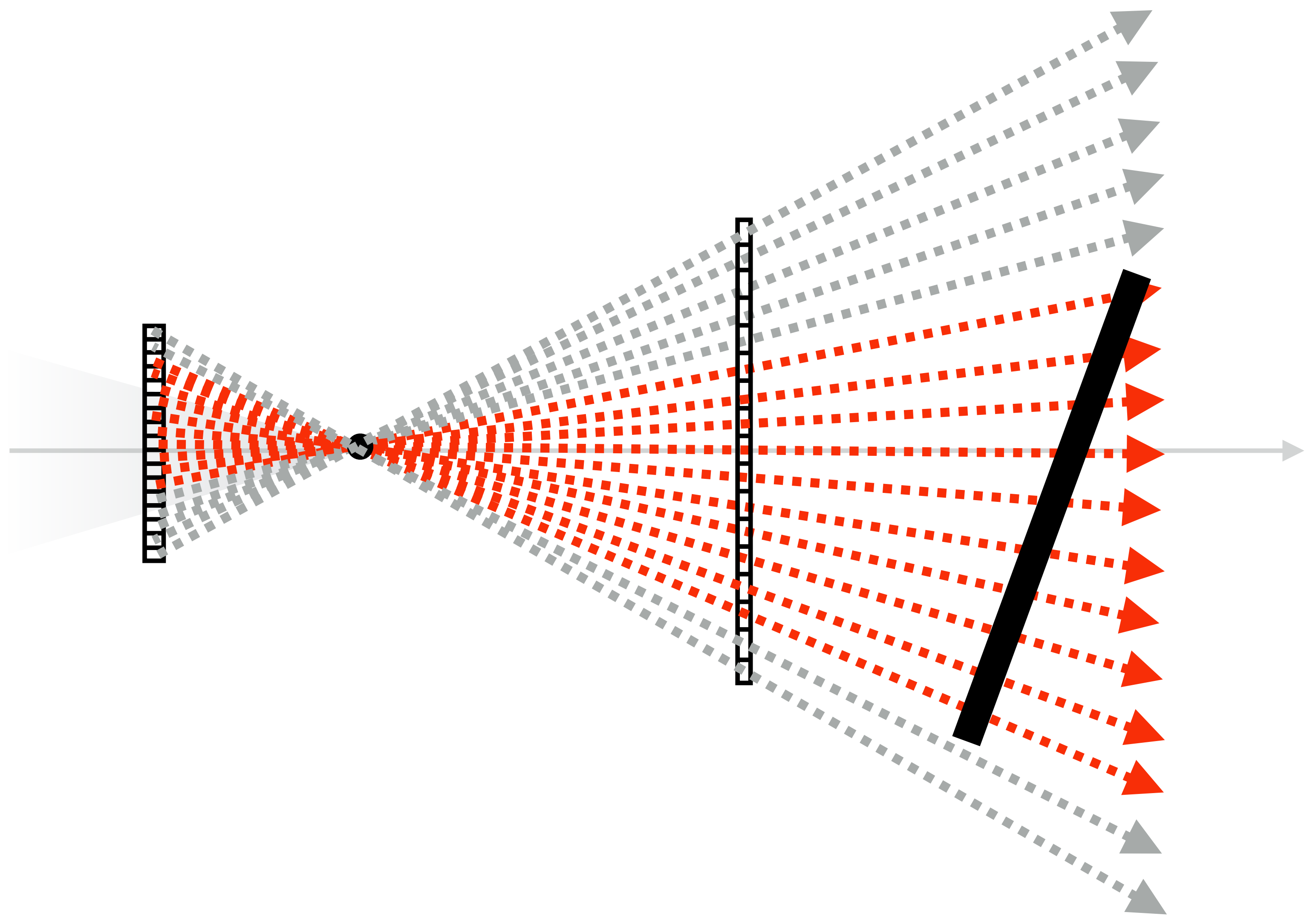
How much radiance is incident from a given direction?

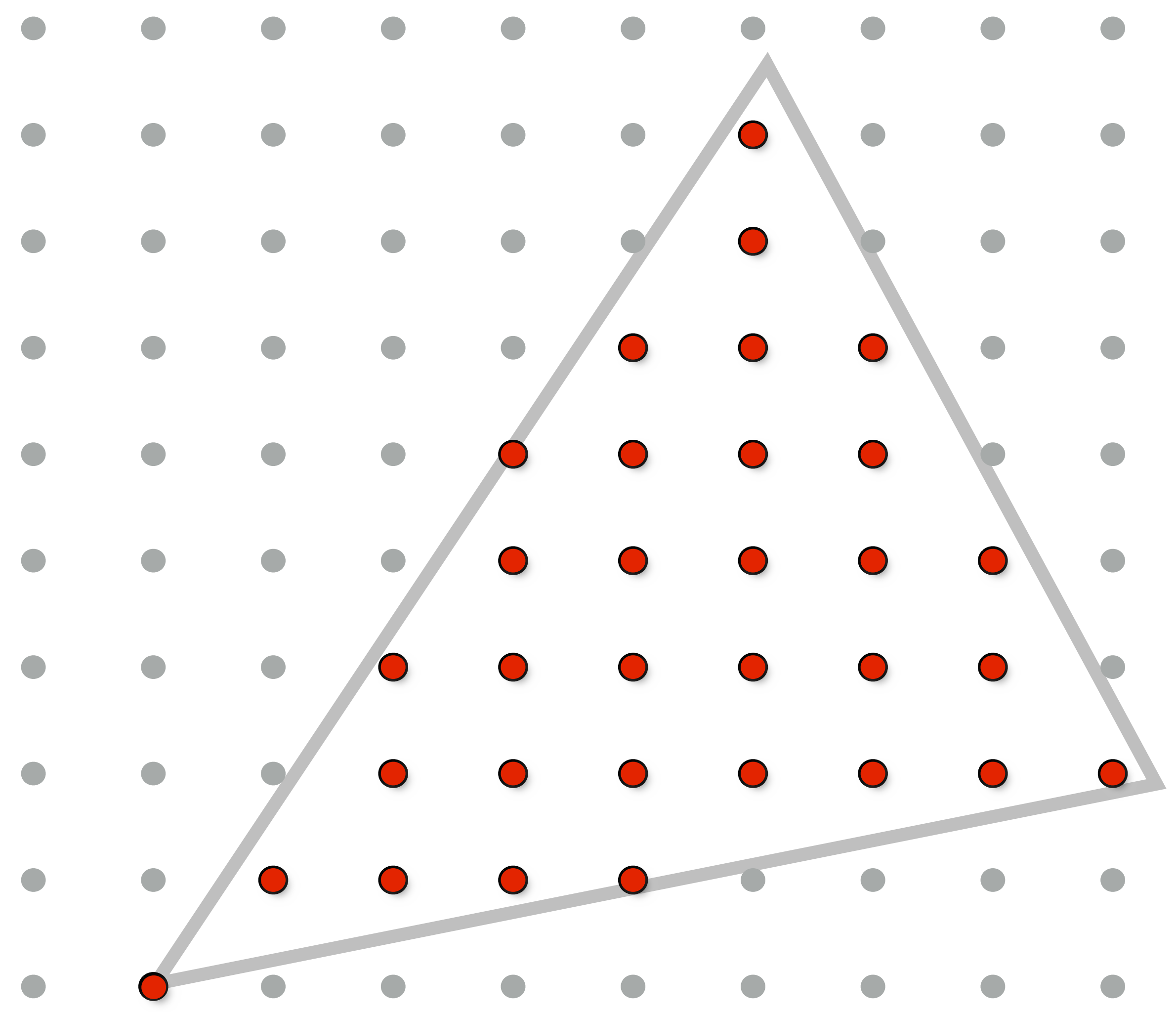


Rasterization: algorithm for "camera ray"- scene queries

- Rasterization is a efficient implementation of ray casting where:
  - Ray-scene intersection is computed for a batch of rays
  - All rays in the batch originate from same origin
  - Rays are distributed uniformly in plane of projection

  Note: rasterization does not yield uniform distribution in angle… angle between rays is smaller away from view direction than it is in the center of the view because equal steps in Y are not equal steps in angle.





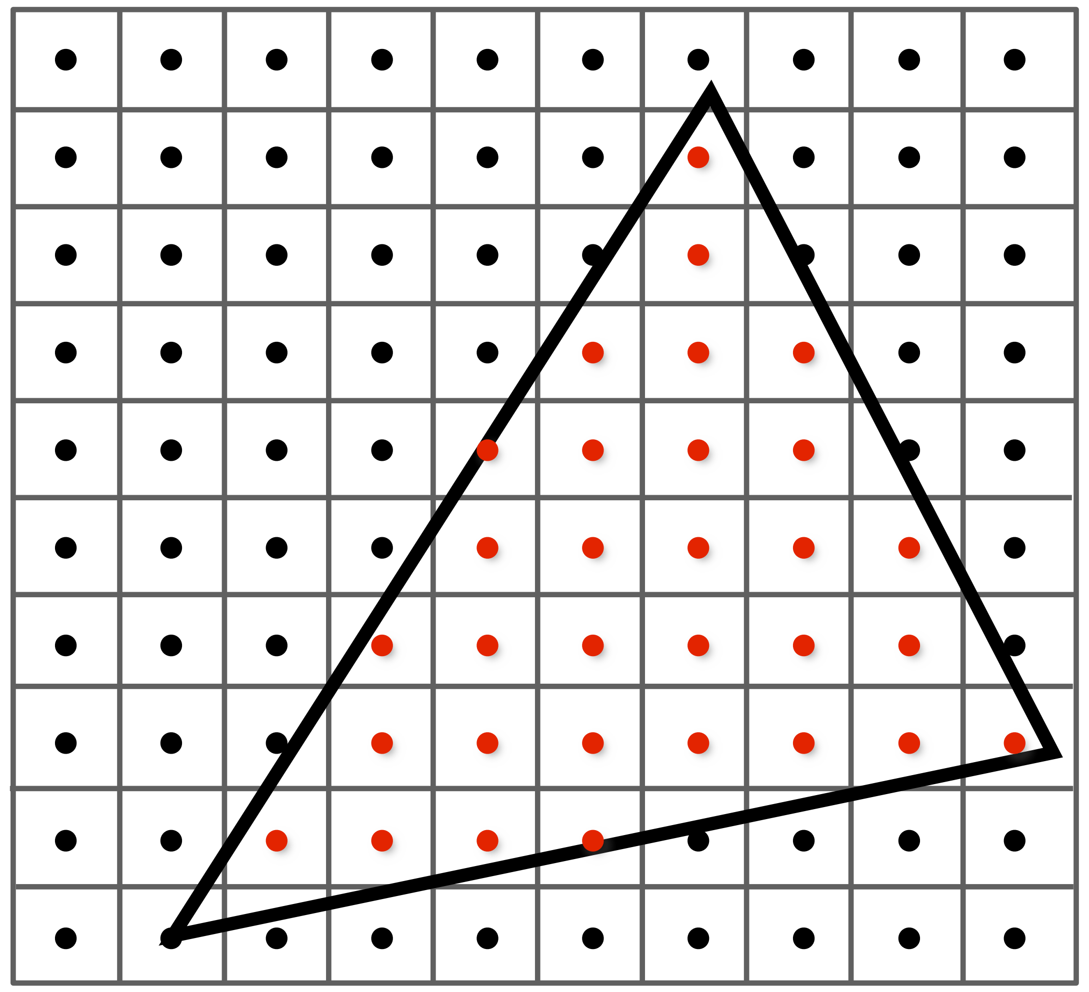
Review: basic rasterization algorithm

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: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:         // loop 2: over 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)

Review: 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: over 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: over 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 rejected radiance from triangle r.tri at hit point
```

And as you know now, a performant raytracer will use an acceleration structure like a BVH.

Compared to rasterization approach: just a reordering of the loops!

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

Theme of this part of the lecture

A surprising number of advanced lighting effects can be *approximated* using the basic primitives of the rasterization pipeline, without the need to actually ray trace the scene geometry. We are going to approximate the use of ray tracing with:

- Rasterization
- Texture mapping
- Depth buffer for occlusion

These techniques have been the basis of high quality real-time rendering for decades.

Since ray tracing performance is not fast enough to be used in real-time applications.

Although this is changing…

Shadows

How much light is REFLECTED from p toward p_0

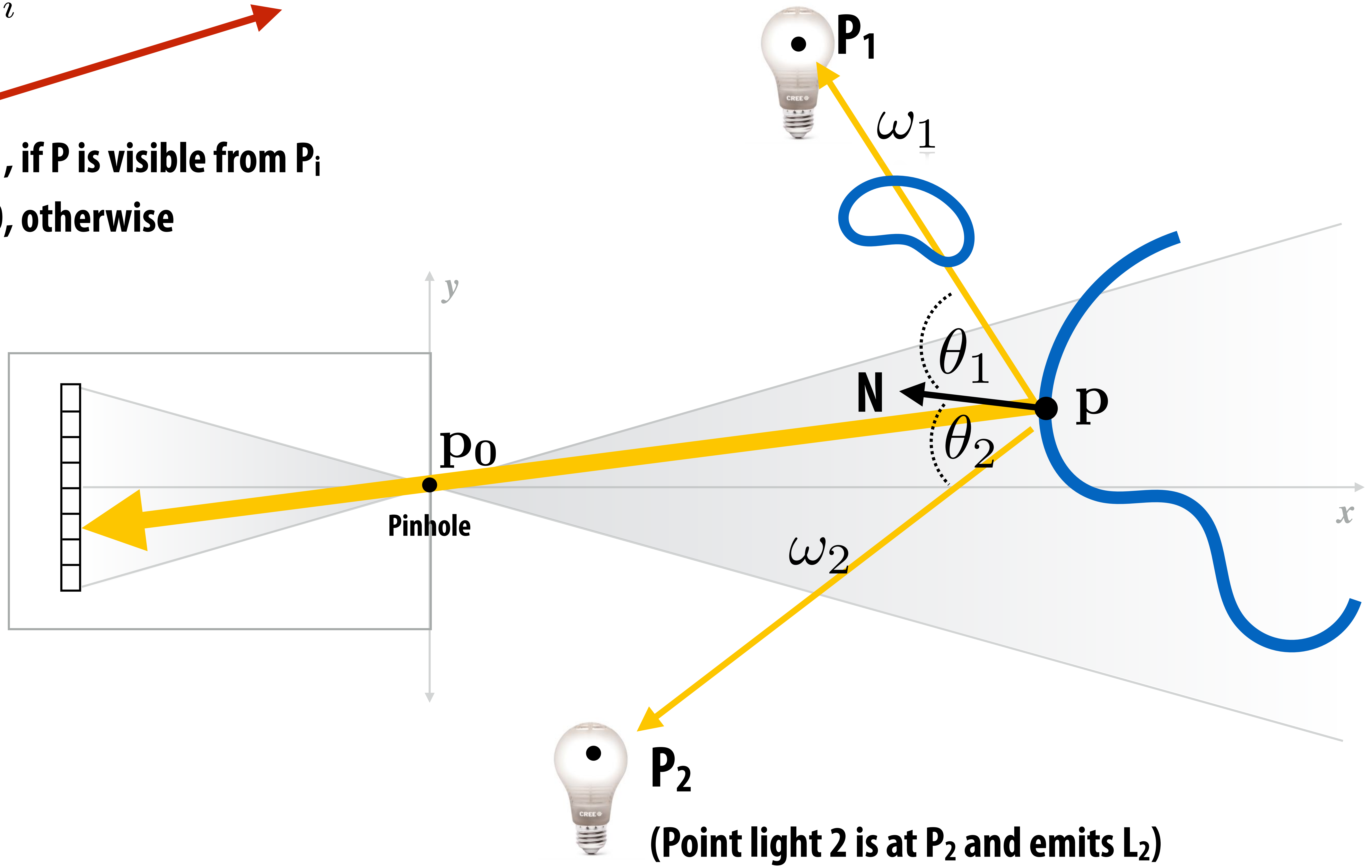
$$L(\mathbf{p},\omega_o) = \sum_i f(\mathbf{p},\omega_i,\omega_o) V(\mathbf{p},\mathbf{p_i})\, L_i\, \cos\theta_i$$

Visibility term:

$V(\mathbf{p},\mathbf{p_i})$ 1, if P is visible from P_i

0, otherwise

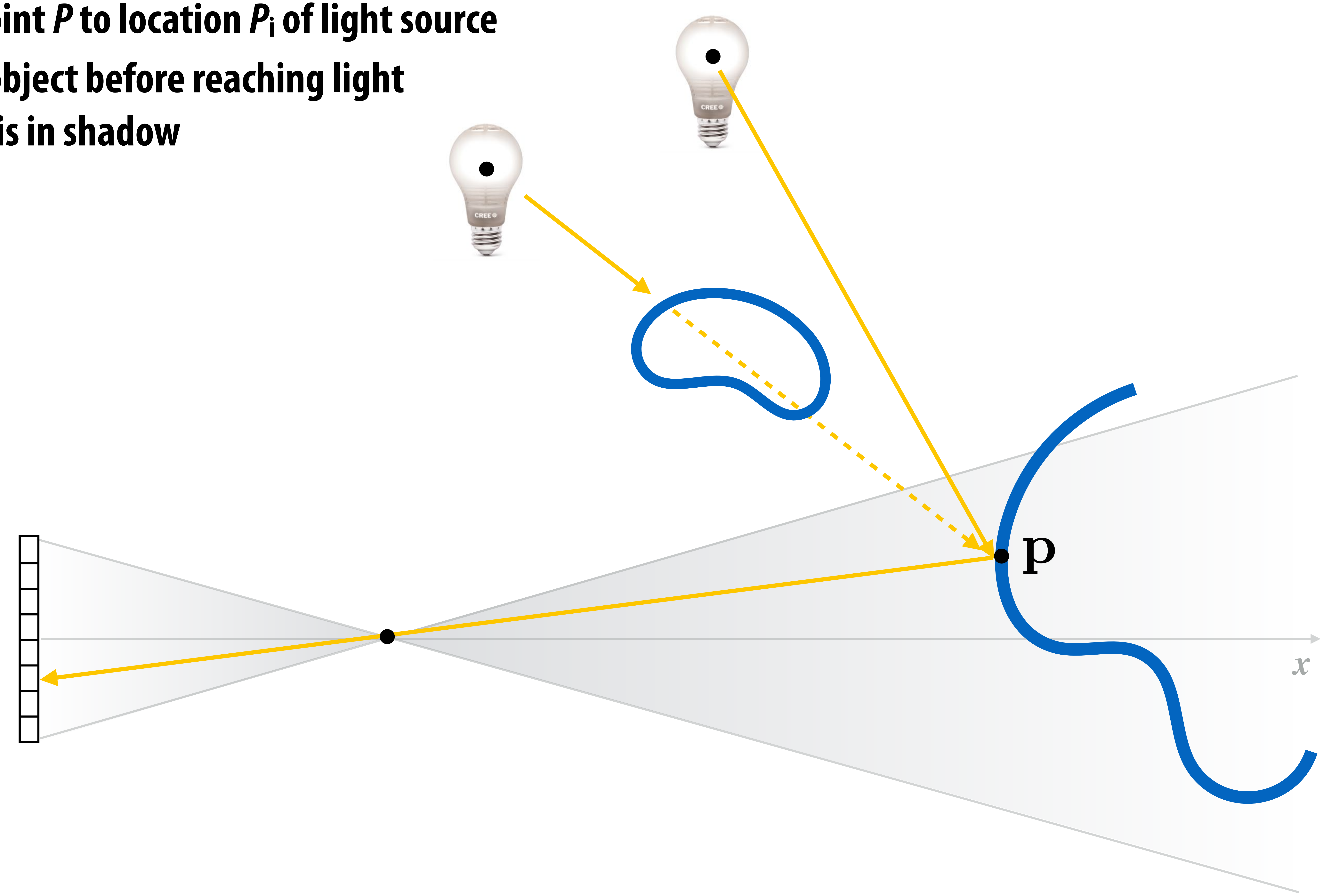
(Point light 1 is at P_1 and emits L_1)



(Point light 2 is at P_2 and emits L_2)

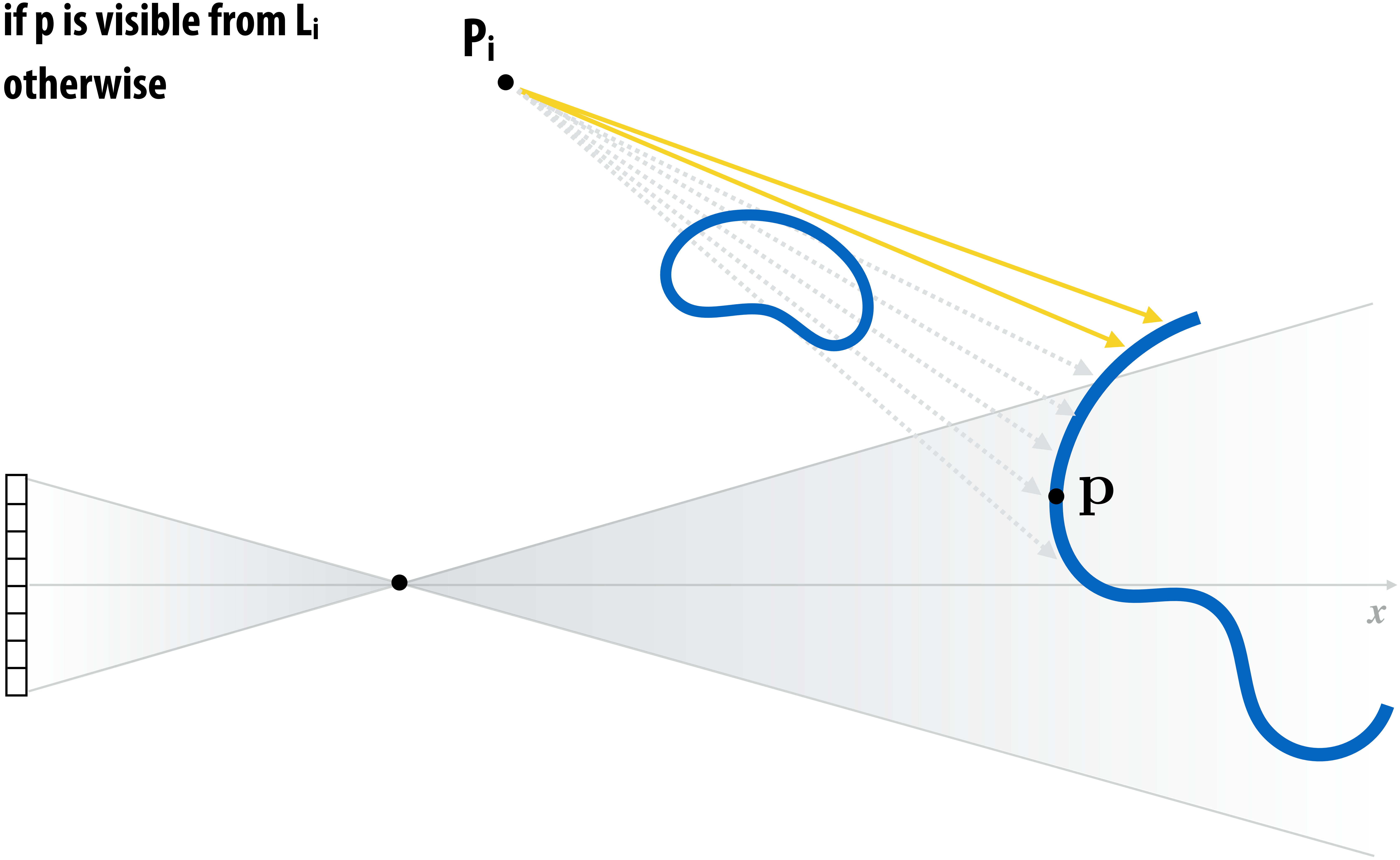
Review: How to compute $V(\mathrm{p}, \mathrm{p_i})$ using ray tracing

- Trace ray from point P to location P_i of light source
- If ray hits scene object before reaching light source… then P is in shadow



Point lights generate "hard shadows"
(Either a point is in shadow or it's not)

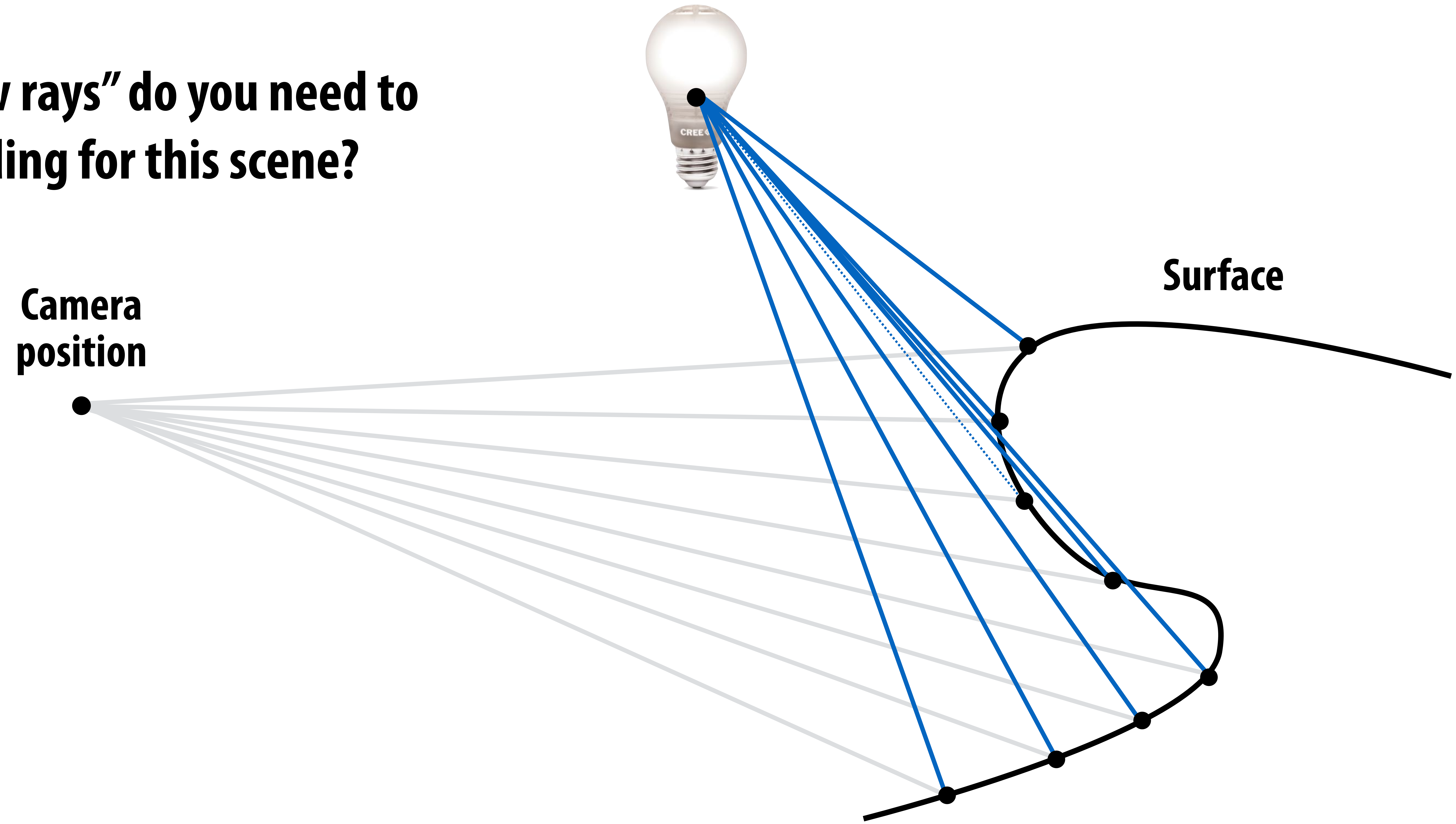
$$V(\mathbf{p}, \mathbf{p_i}) = \begin{cases} 1, \text{ if p is visible from } L_i \\ 0, \text{ otherwise} \end{cases}$$



What if you didn't have a ray tracer, just a rasterizer?

We want to shade these points
(aka "fragments" in rasterization pipeline)

What "shadow rays" do you need to compute shading for this scene?



Shadow mapping

[Williams 78]

1. *Place camera at position of the scene's point light source*
2. Render scene to compute depth of closest object to light along a uniformly spaced set of "shadow rays" (note: answer is stored in depth buffer after rendering)
3. Store precomputed shadow ray intersection results in a texture map

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

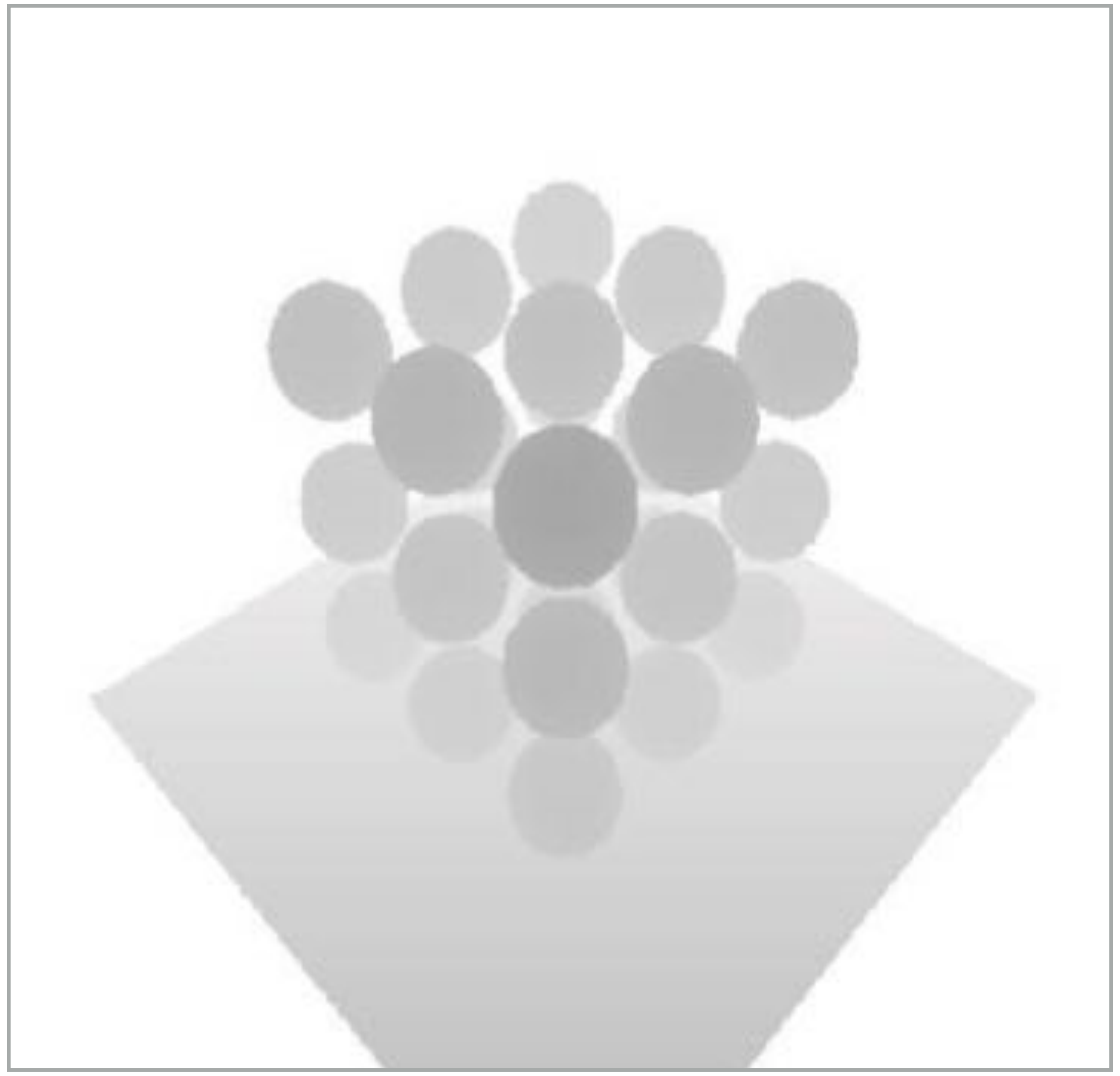
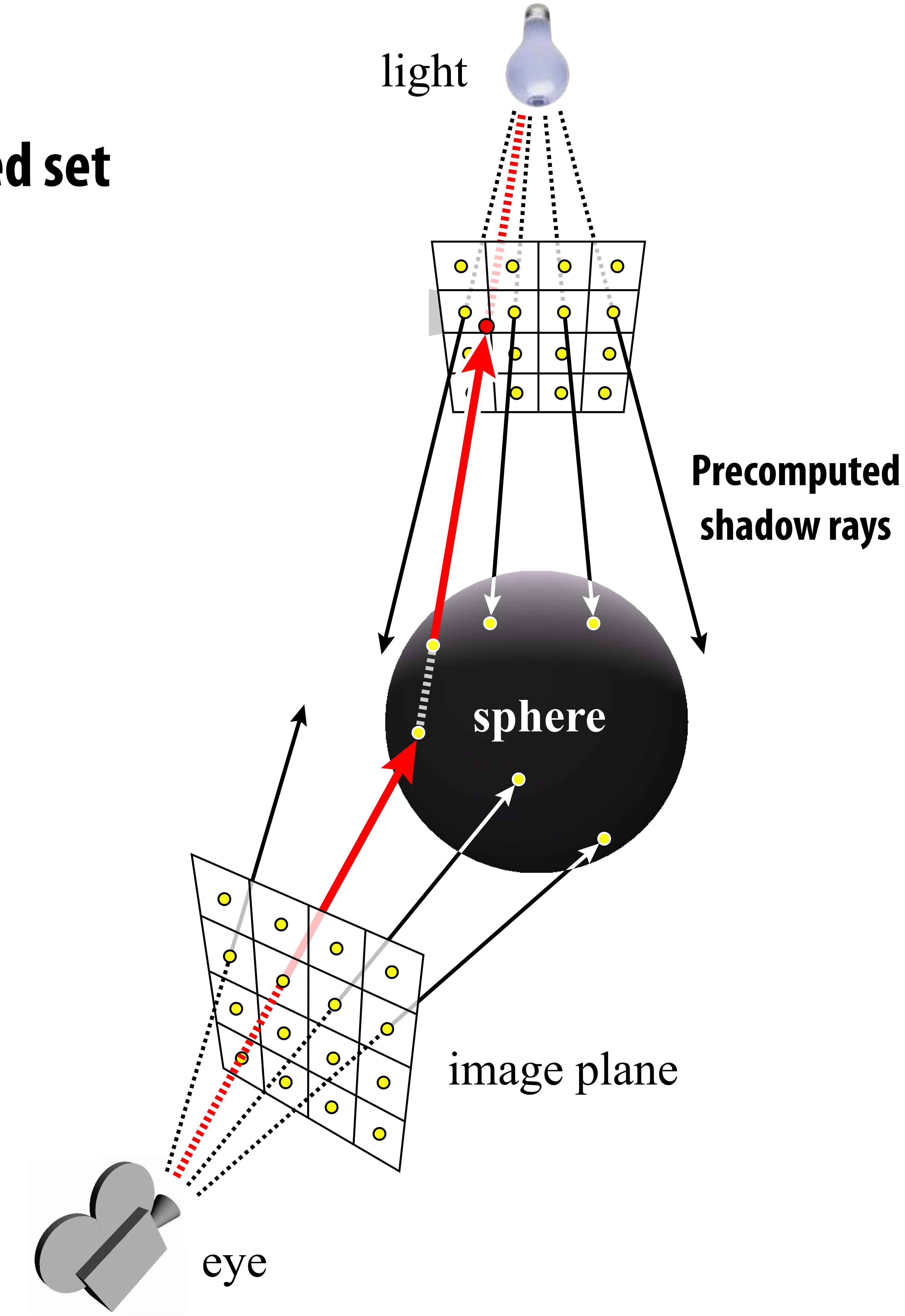
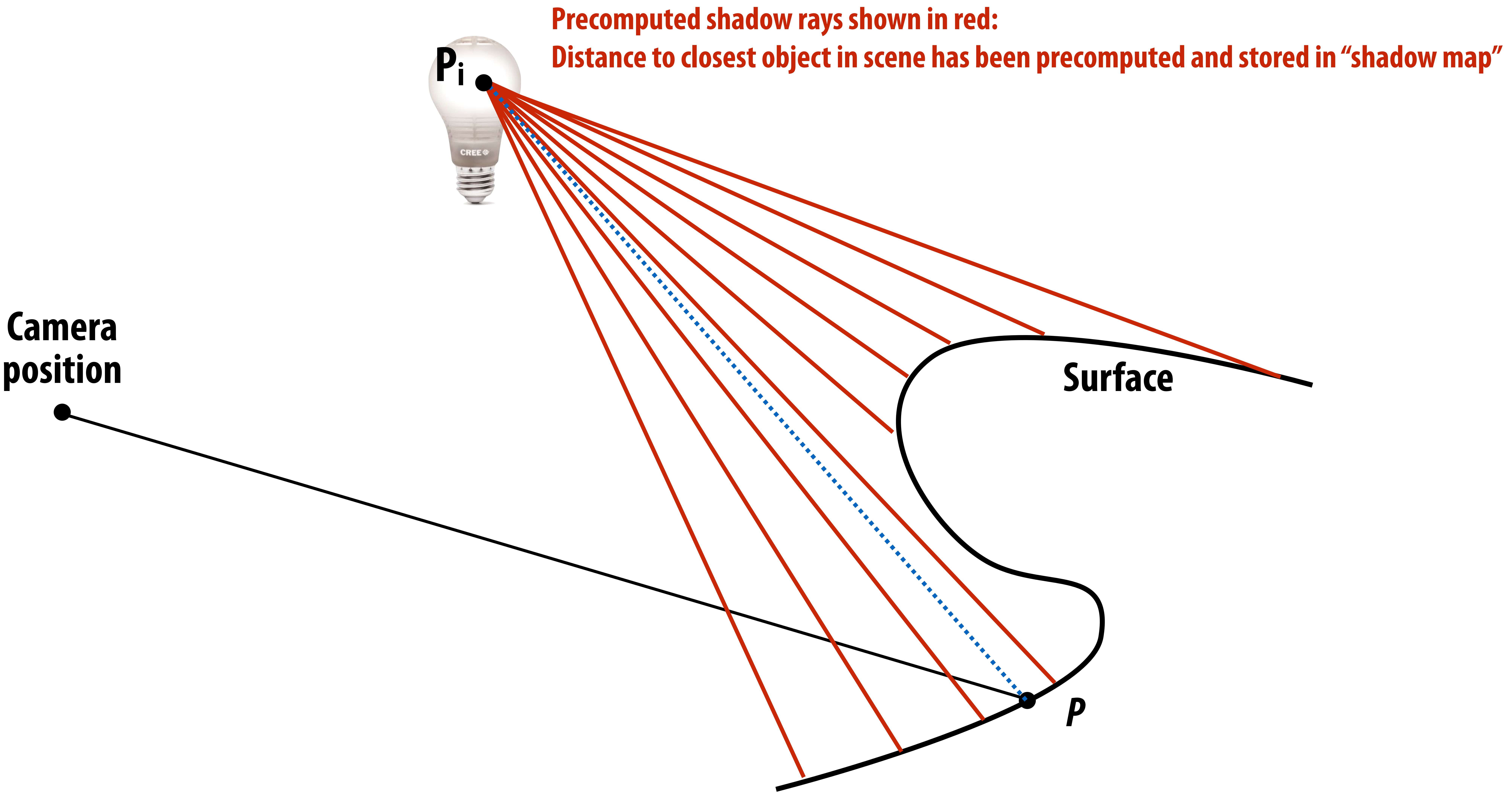


Image credits: Segal et al. 92, NVIDIA

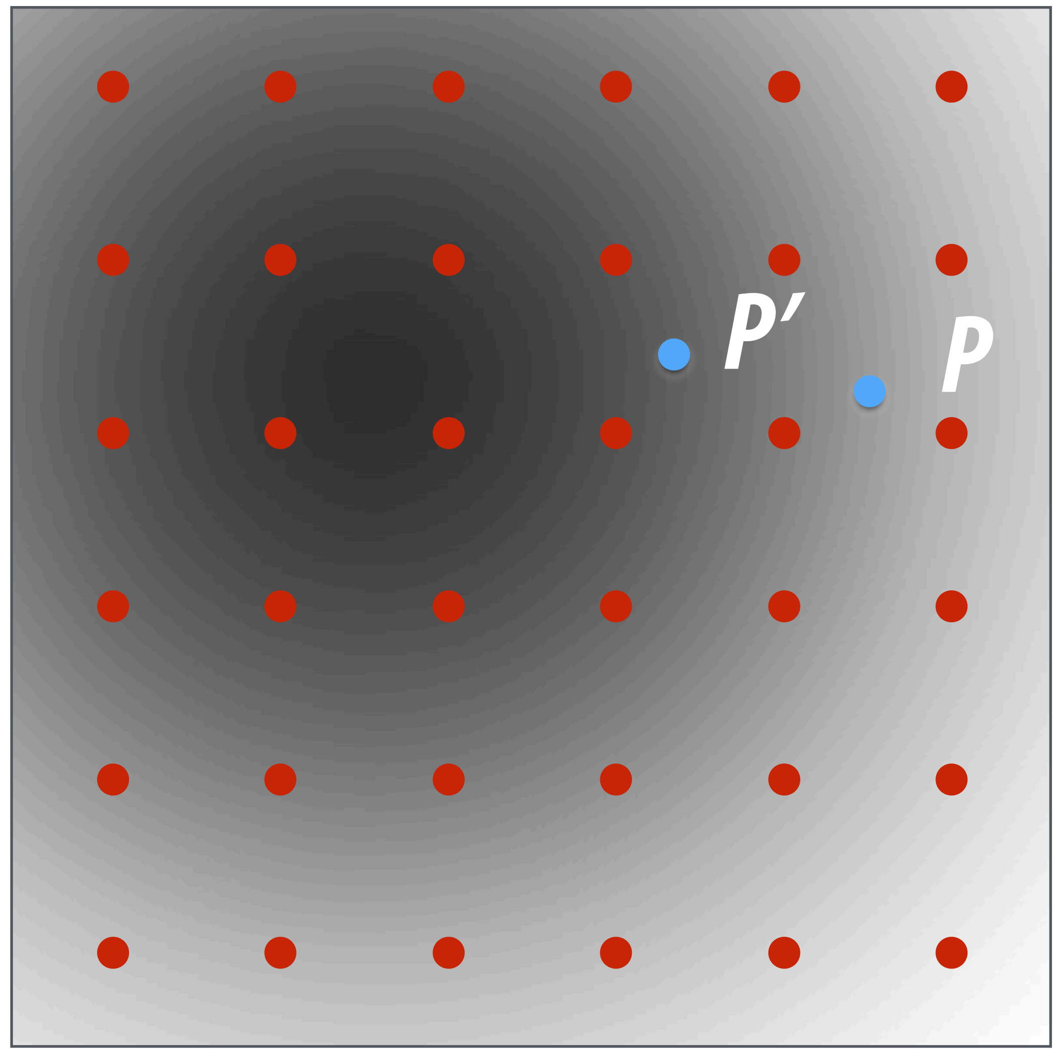


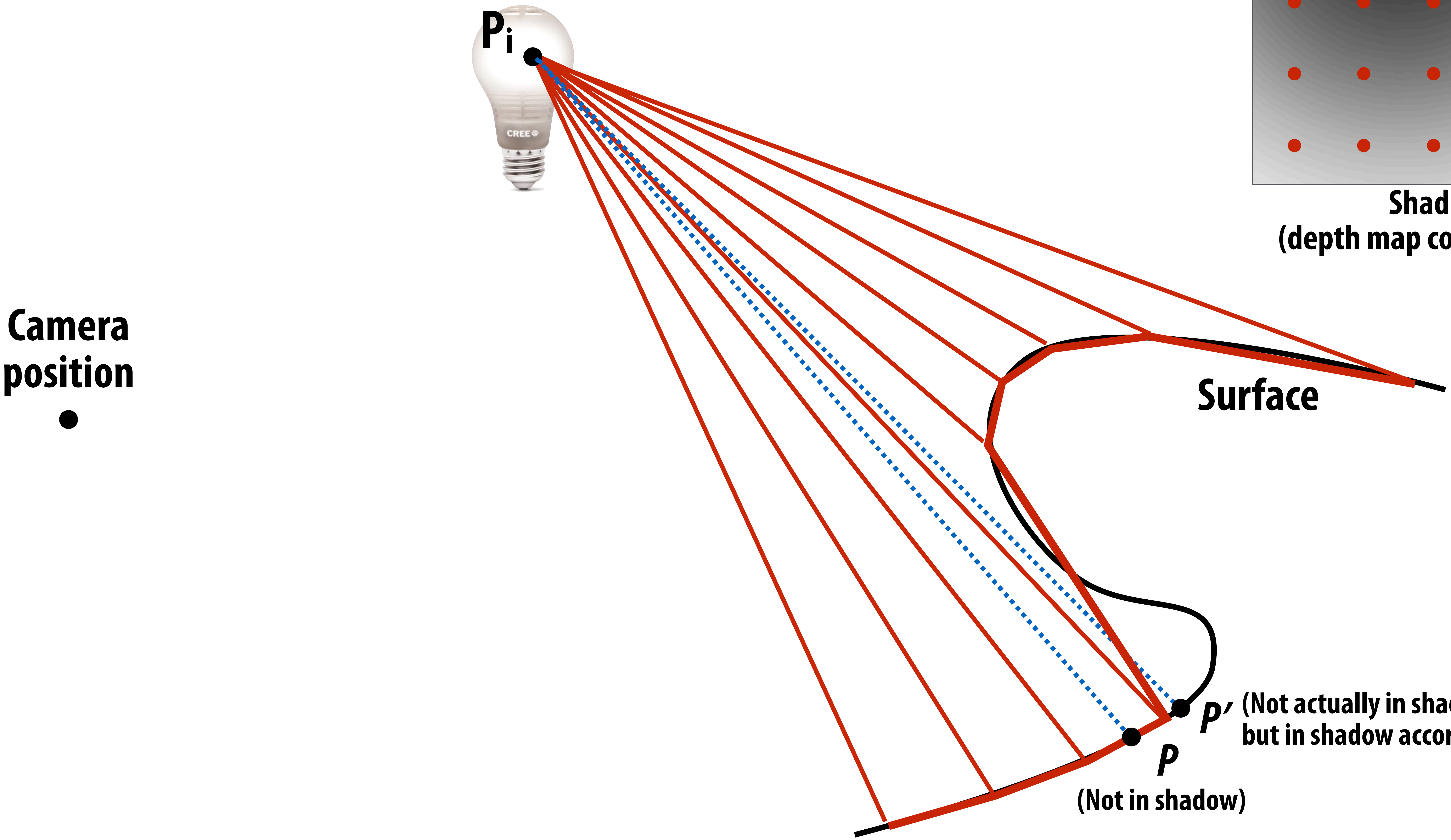
Result of shadow texture lookup approximates visibility result when shading fragment at *P*



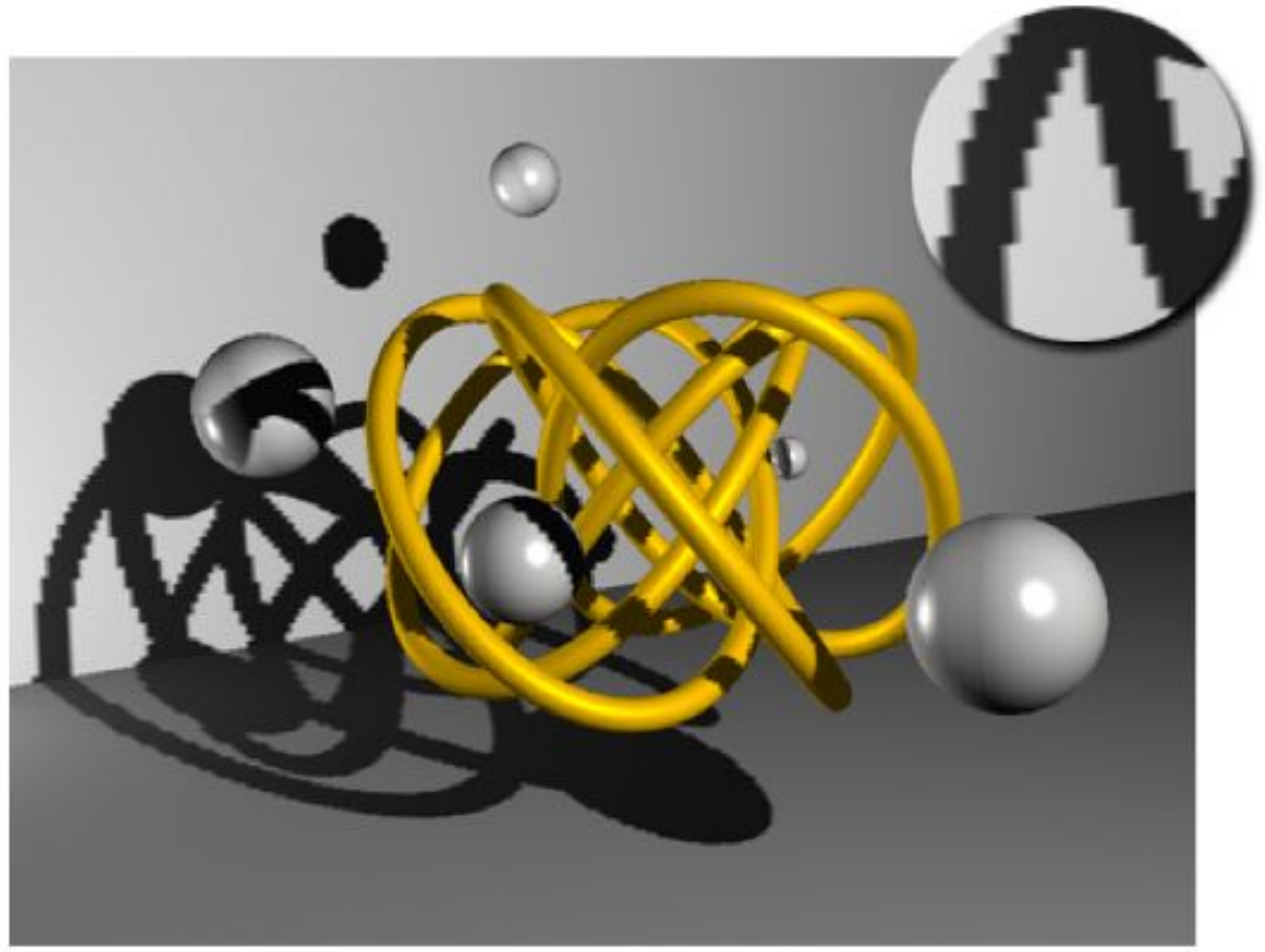
Interpolation error

Bilinear interpolation of shadow map values (red line) only approximates distance to closest surface point in all directions from the camera

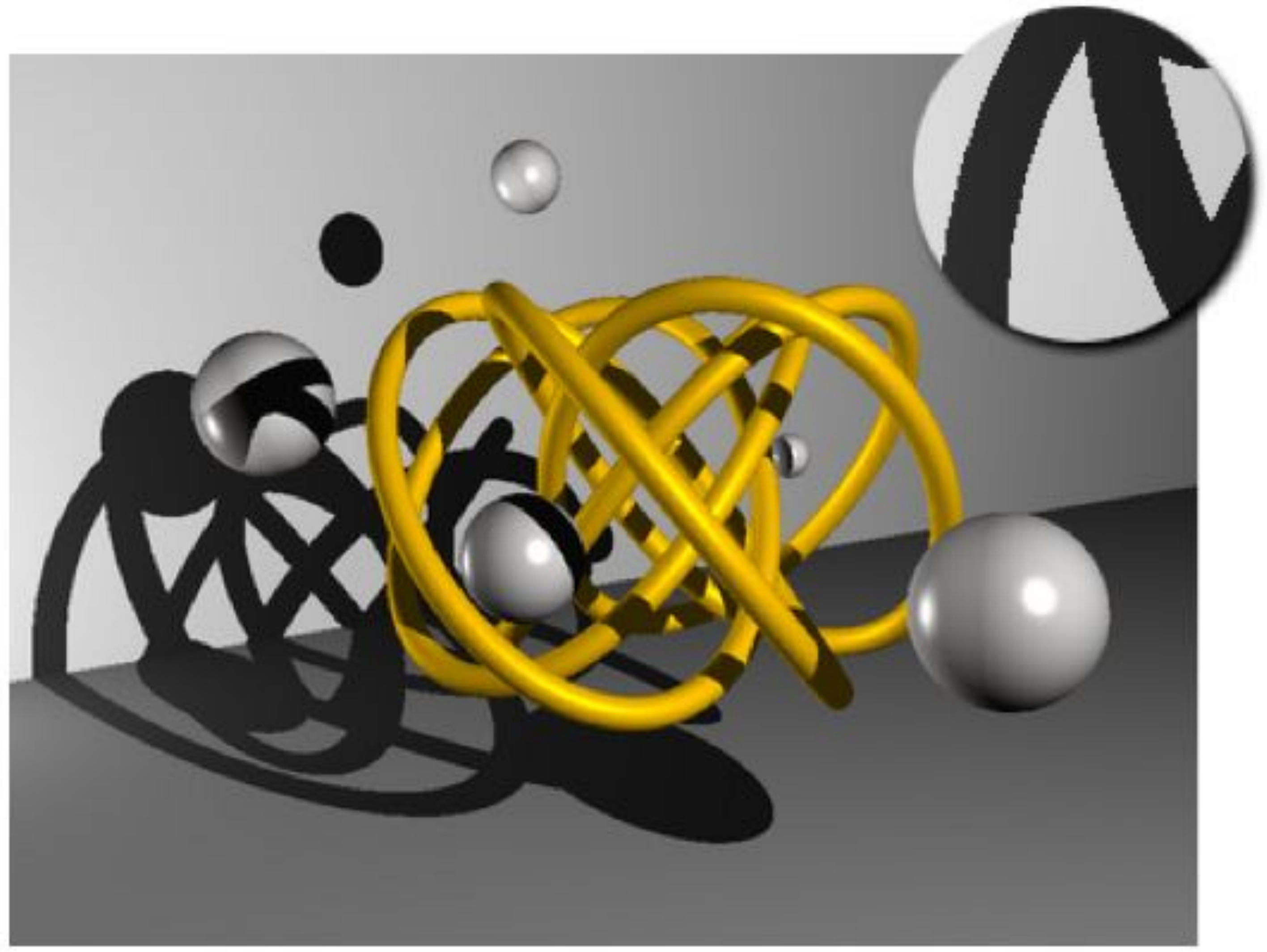




Shadow aliasing due to shadow map undersampling



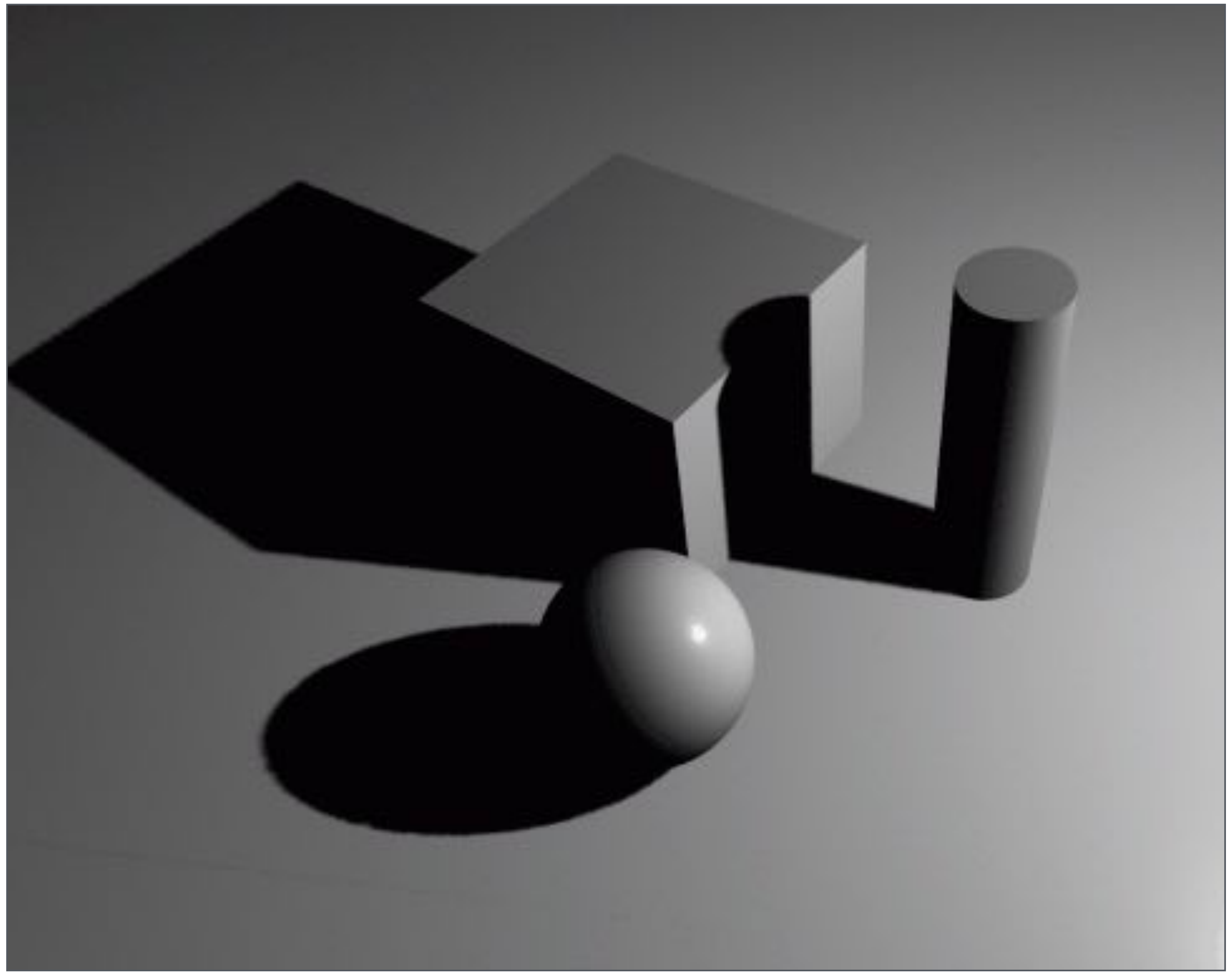
Shadows computed using shadow map



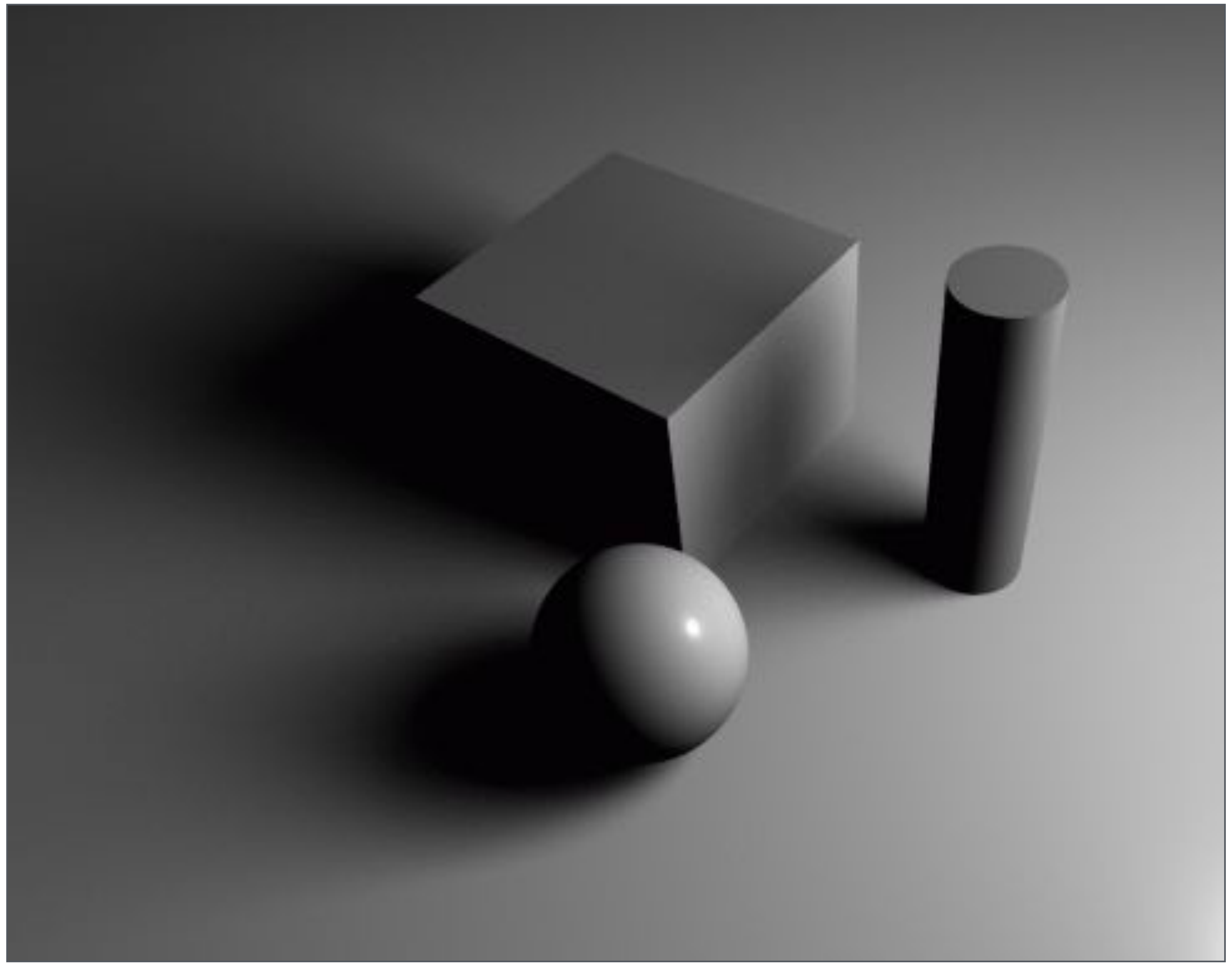
Correct hard shadows
(result from computing visibility along ray between surface point and light directly using ray tracing)

Image credit: Johnson et al. TOG 2005

Soft shadows



Hard shadows
(created by point light source)



Soft shadows
(created by ???)

Area light

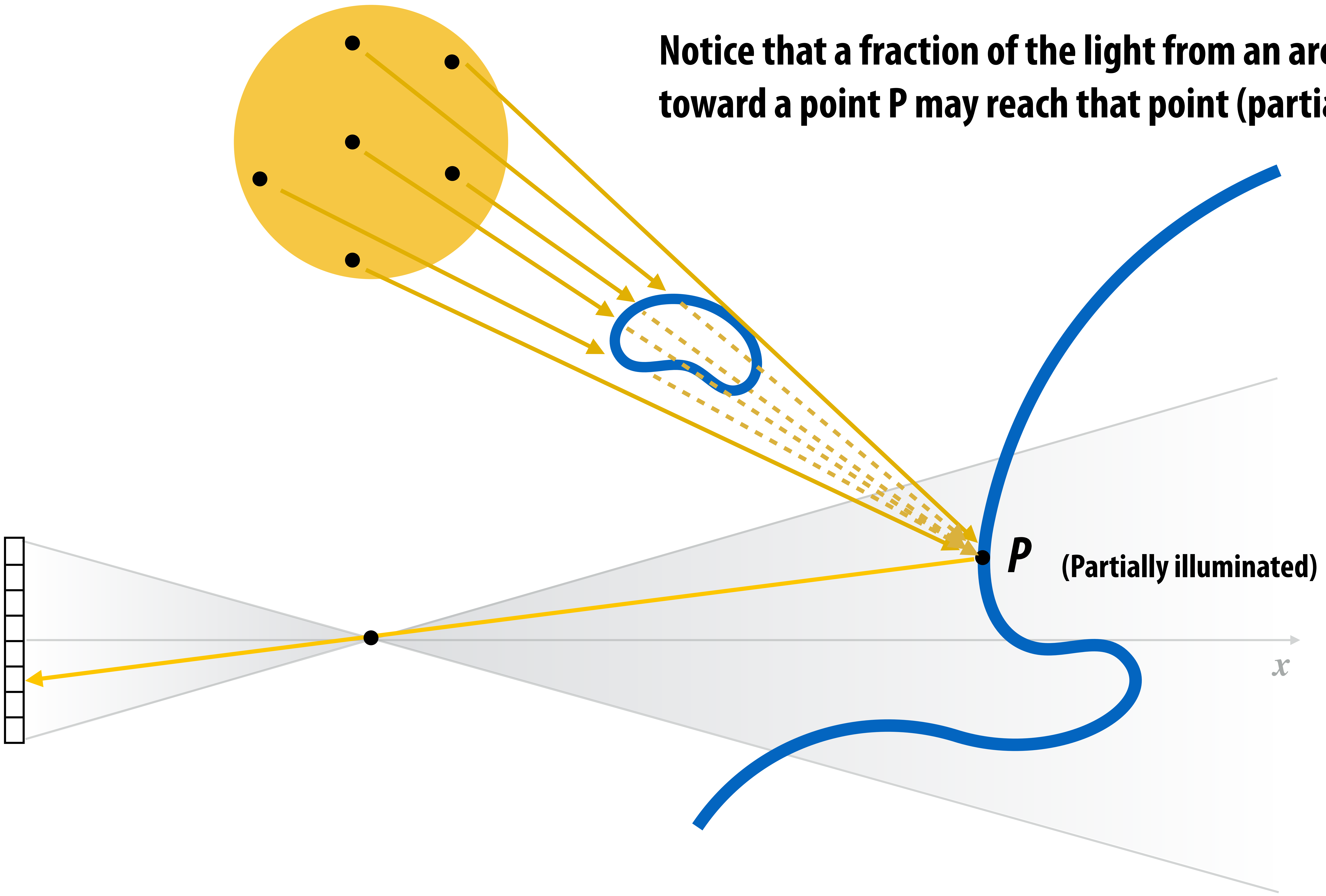
Credit: Jaime Velasco (https://all3dp.com/2/blender-lighting-simply-explained/)

Area light

Credit: Jaime Velasco (https://all3dp.com/2/blender-lighting-simply-explained/)

Shadow cast by an area light (via ray tracing)

Notice that a fraction of the light from an area light toward a point P may reach that point (partial occlusion)

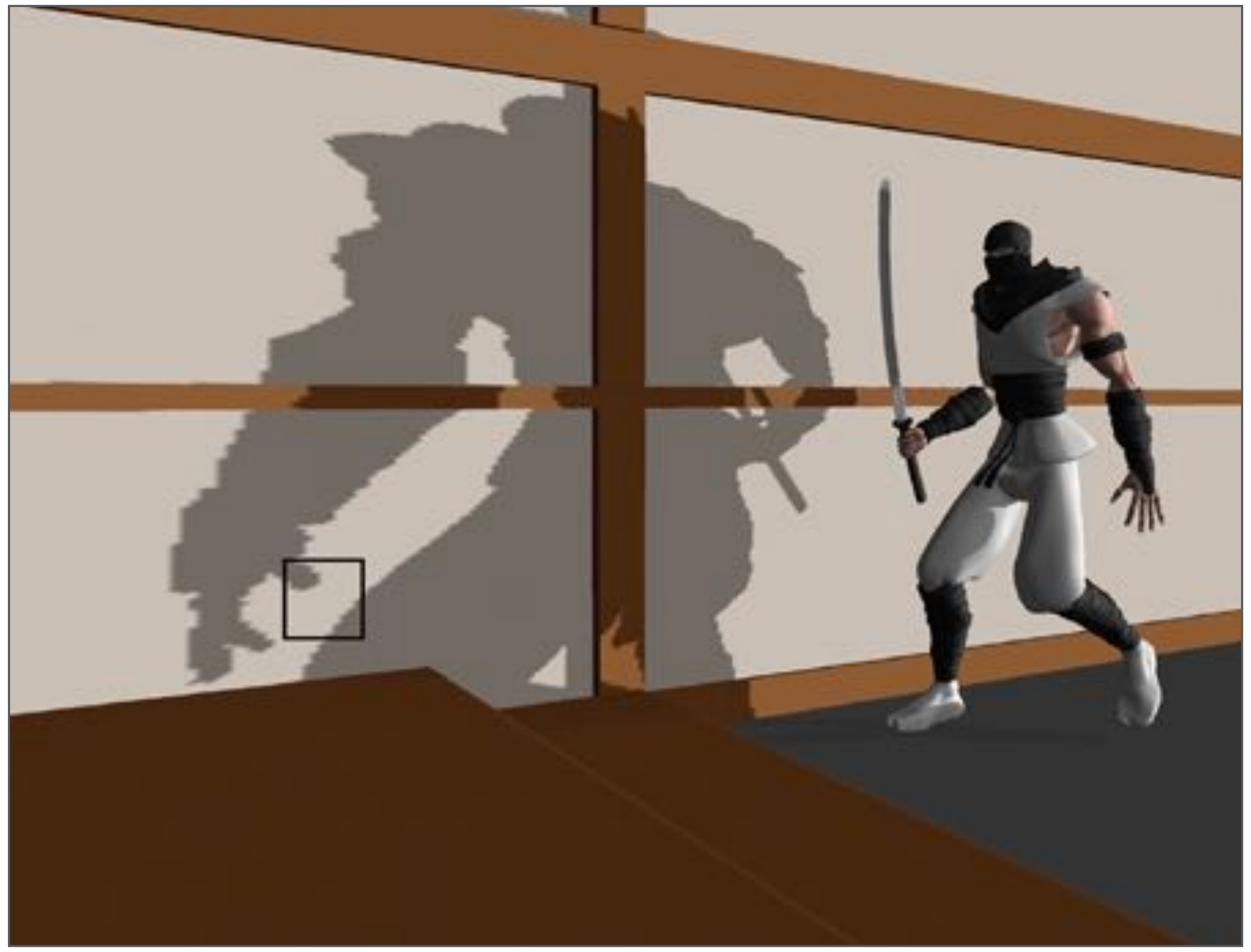


Percentage closer filtering (PCF) — hack!

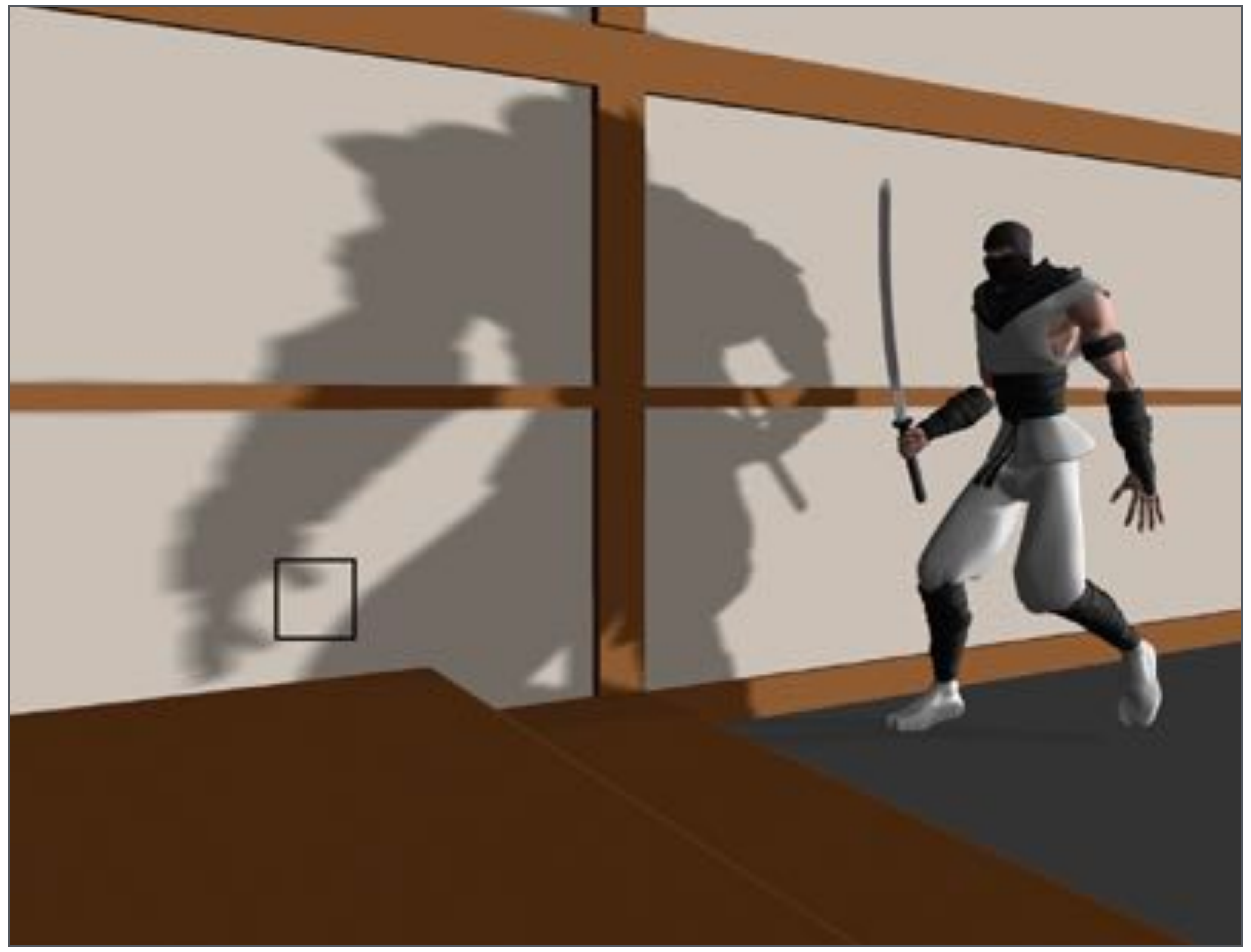
- Instead of sampling shadow map once, perform multiple lookups around desired texture coordinate
- Tabulate fraction of lookups that are in shadow, modulate light intensity accordingly

shadow map values (consider case where distance from light to surface is 0.5)

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1



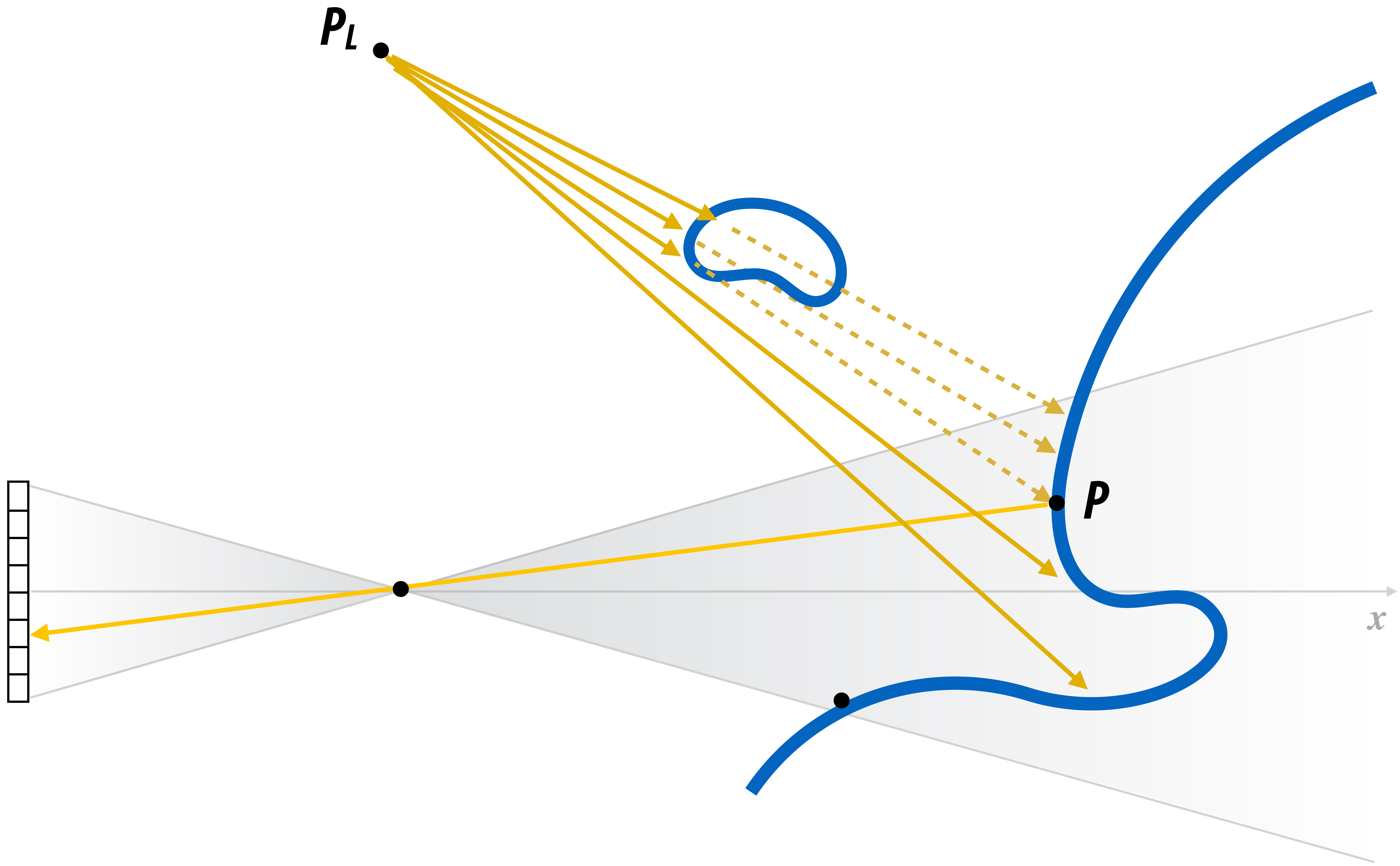
Hard shadows (one lookup per fragment)



PCF shadows (16 lookups per fragment)

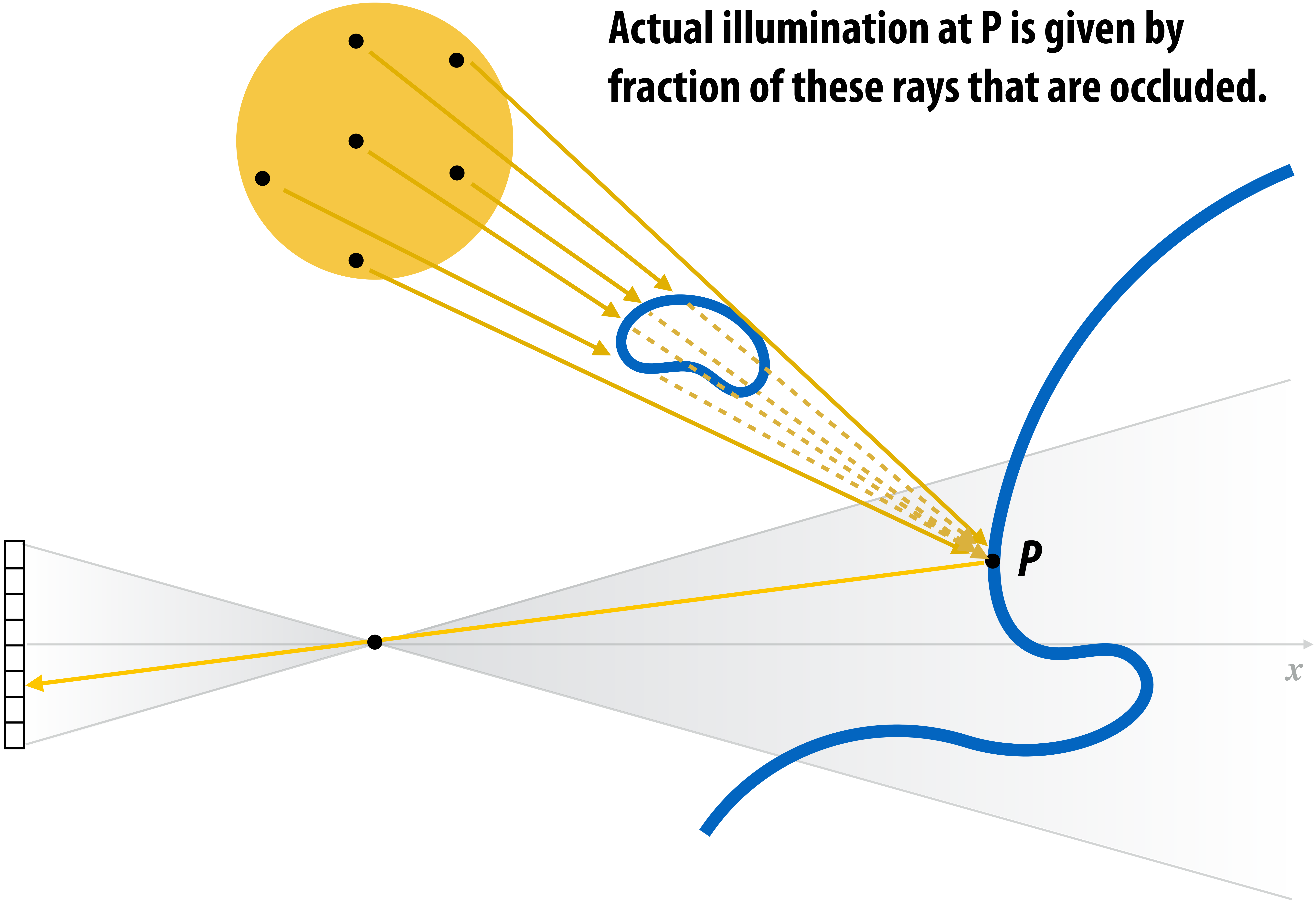
What PCF computes

The fraction of these rays that are shorter than $|P-P_L|$



Shadow cast by an area light

Actual illumination at P is given by fraction of these rays that are occluded.



Q. Why isn't the surface in shadow completely black?

Answer: Assumption that some amount of "ambient light" (light scattered from off surfaces) hits every surface. Here… ambient light is just a constant.

Image credit: Grand Theft Auto V

Image credit: Brennan Shacklett

Ambient occlusion

This scene contains an environment light source that has equal illumination from all directions. (e.g., an overcast day)

All surfaces are diffuse reflectors.

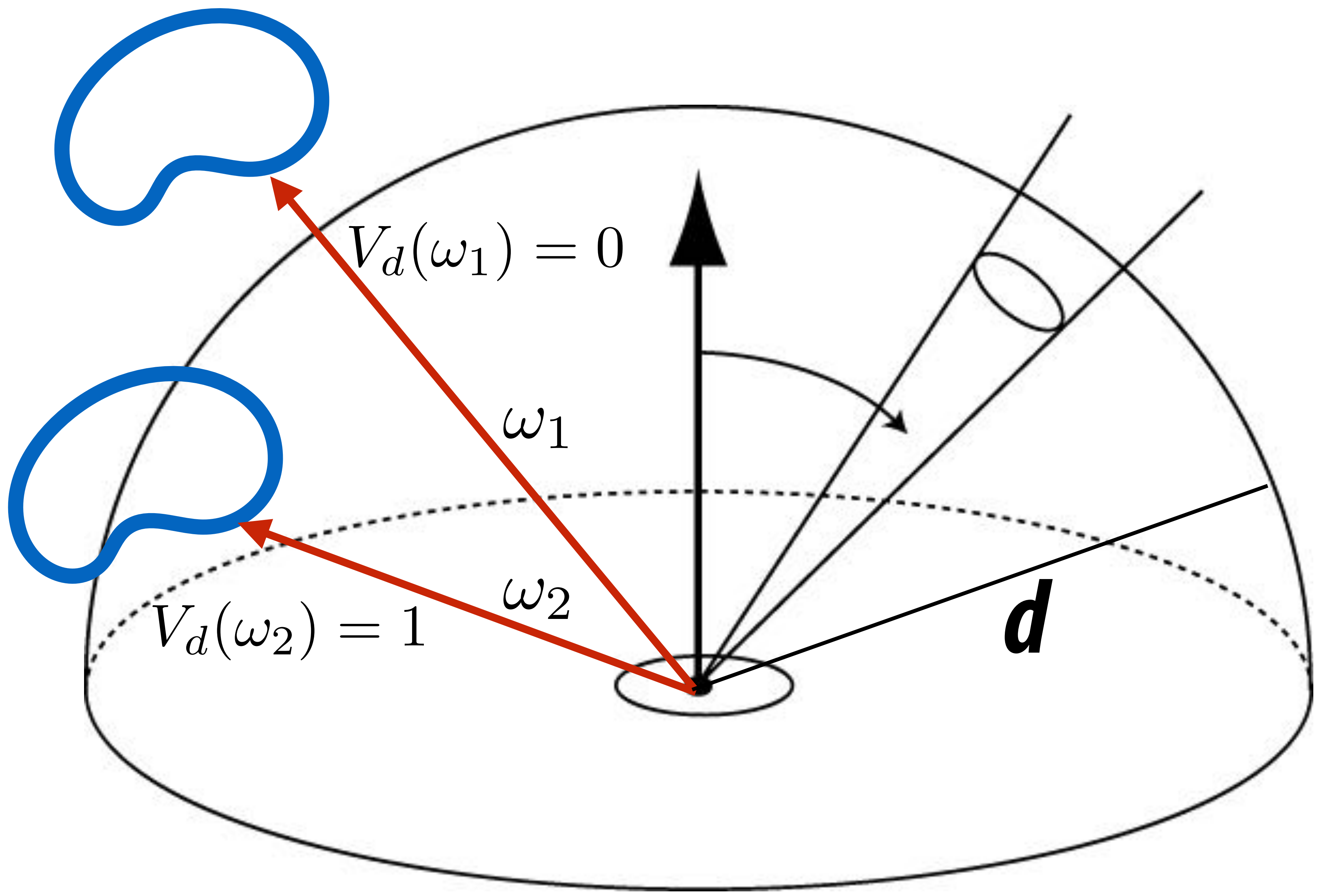
Without accounting for shadows, all surfaces should be the same color.

Hack: ambient obscurance

Idea: Offline, precompute "fraction of hemisphere" that is occluded within distance *d* from a point (e.g., via a ray tracer)

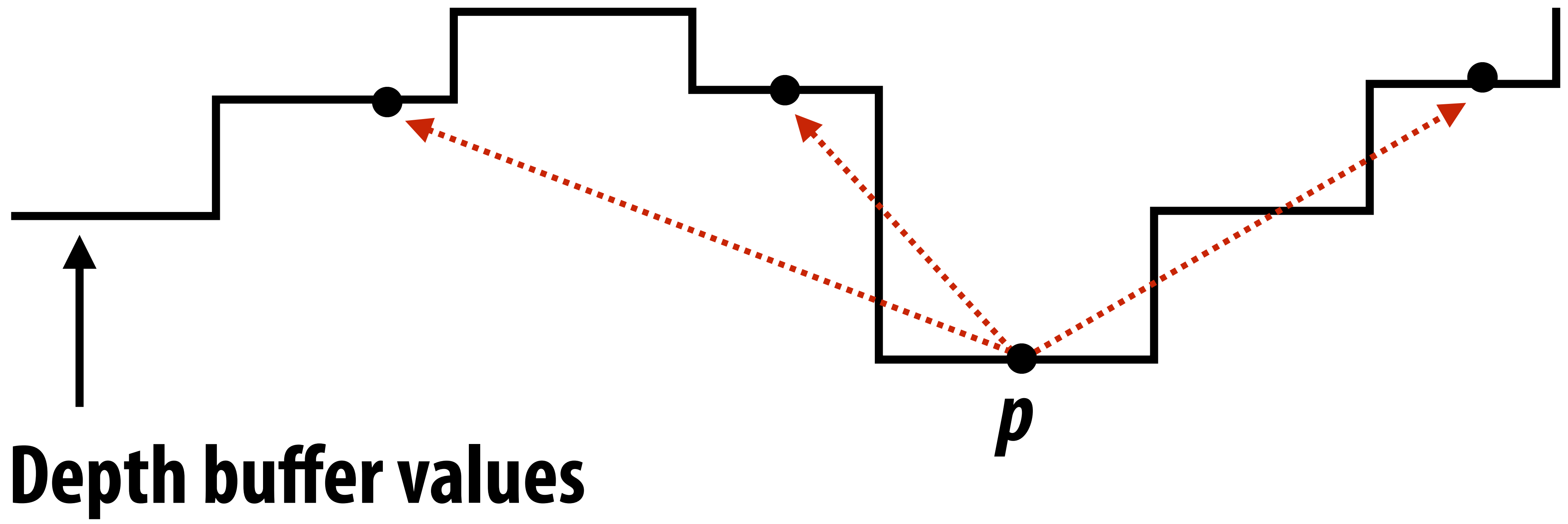
Store this fraction in a texture map

When shading, attenuate environment lighting by this fraction



"Screen-space" ambient occlusion in games

1. Render scene to depth buffer
2. For each pixel *p*, "ray trace" the depth buffer to estimate local occlusion of hemisphere - use a few samples per pixel
3. Blur the the per-pixel occlusion results to reduce noise
4. When shading pixels, darken direct environment lighting by occlusion amount computed for the current pixel

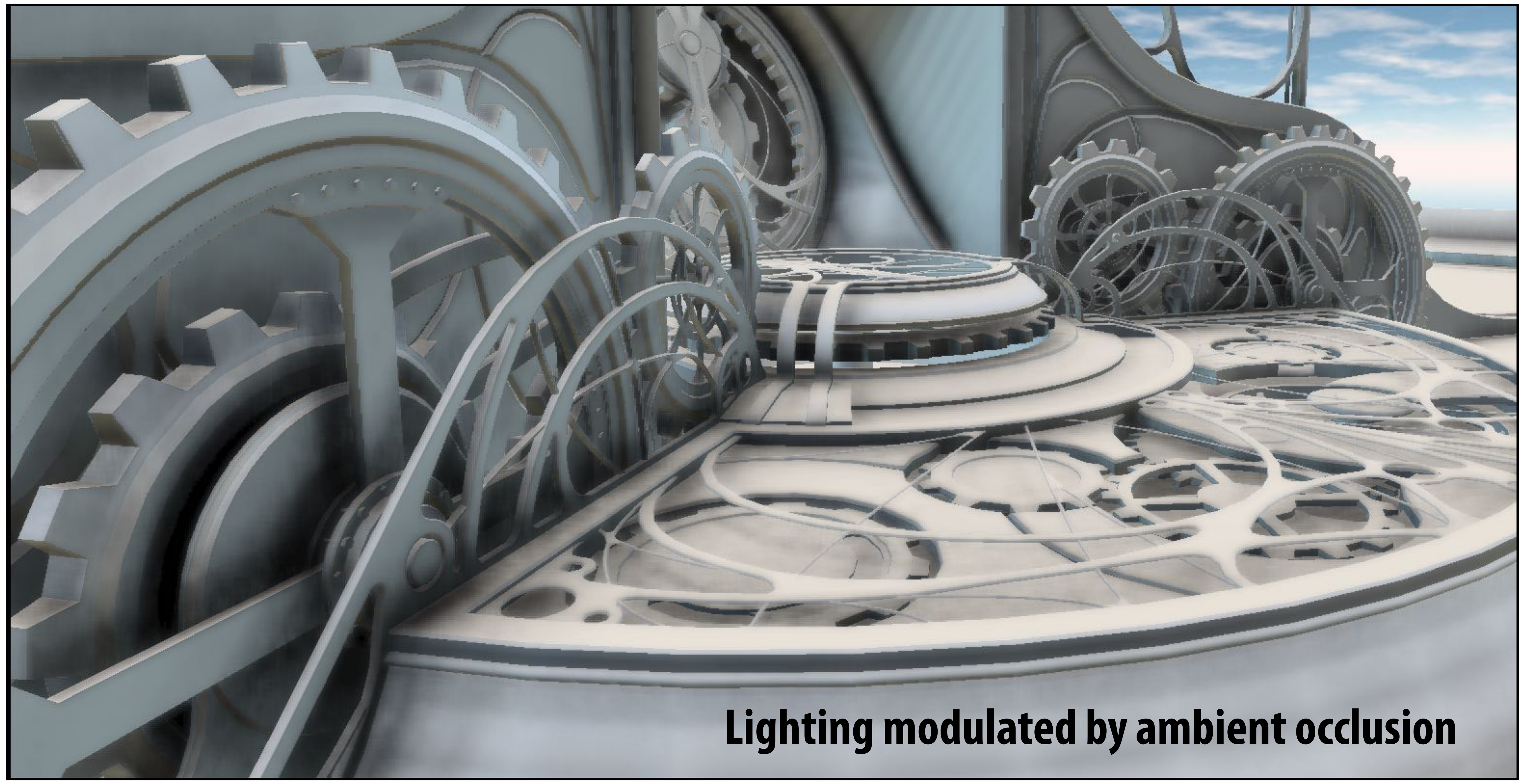


without ambient occlusion

with ambient occlusion

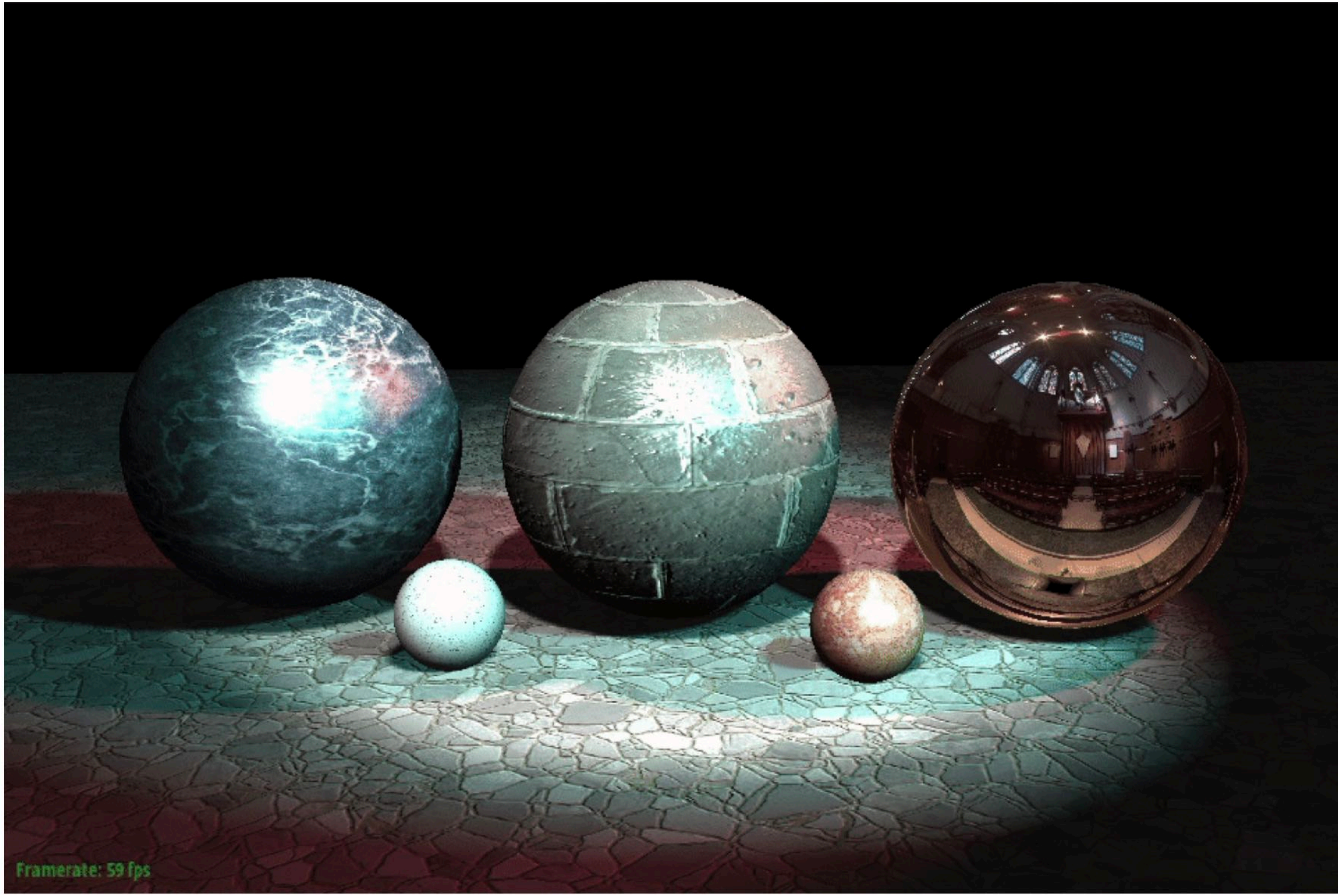
Ambient occlusion


Direct Lighting (no self-shadowing computations)


Lighting modulated by ambient occlusion

Reflections

What is wrong with this picture?

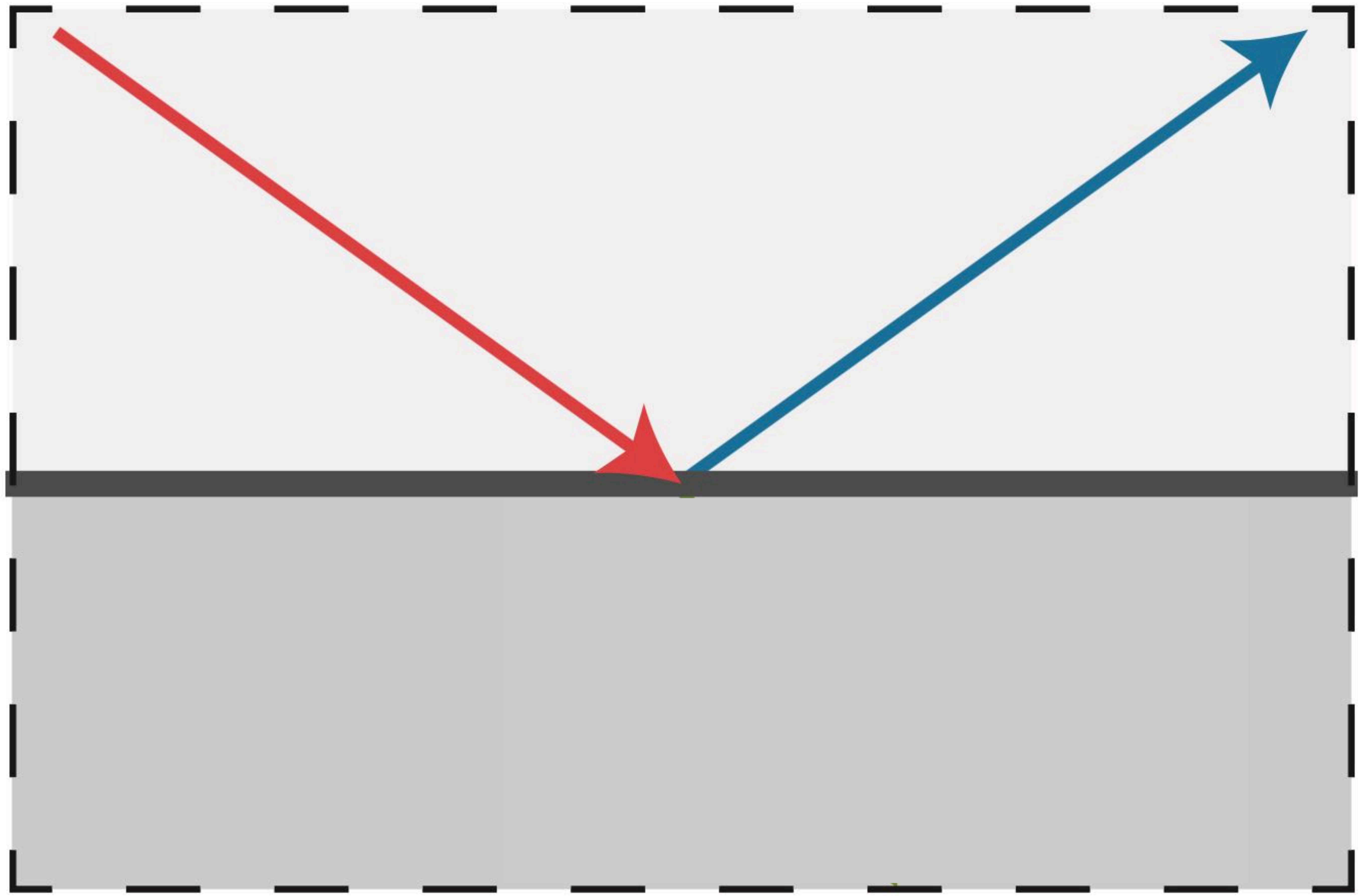


Reflections

Image credit: NVIDIA

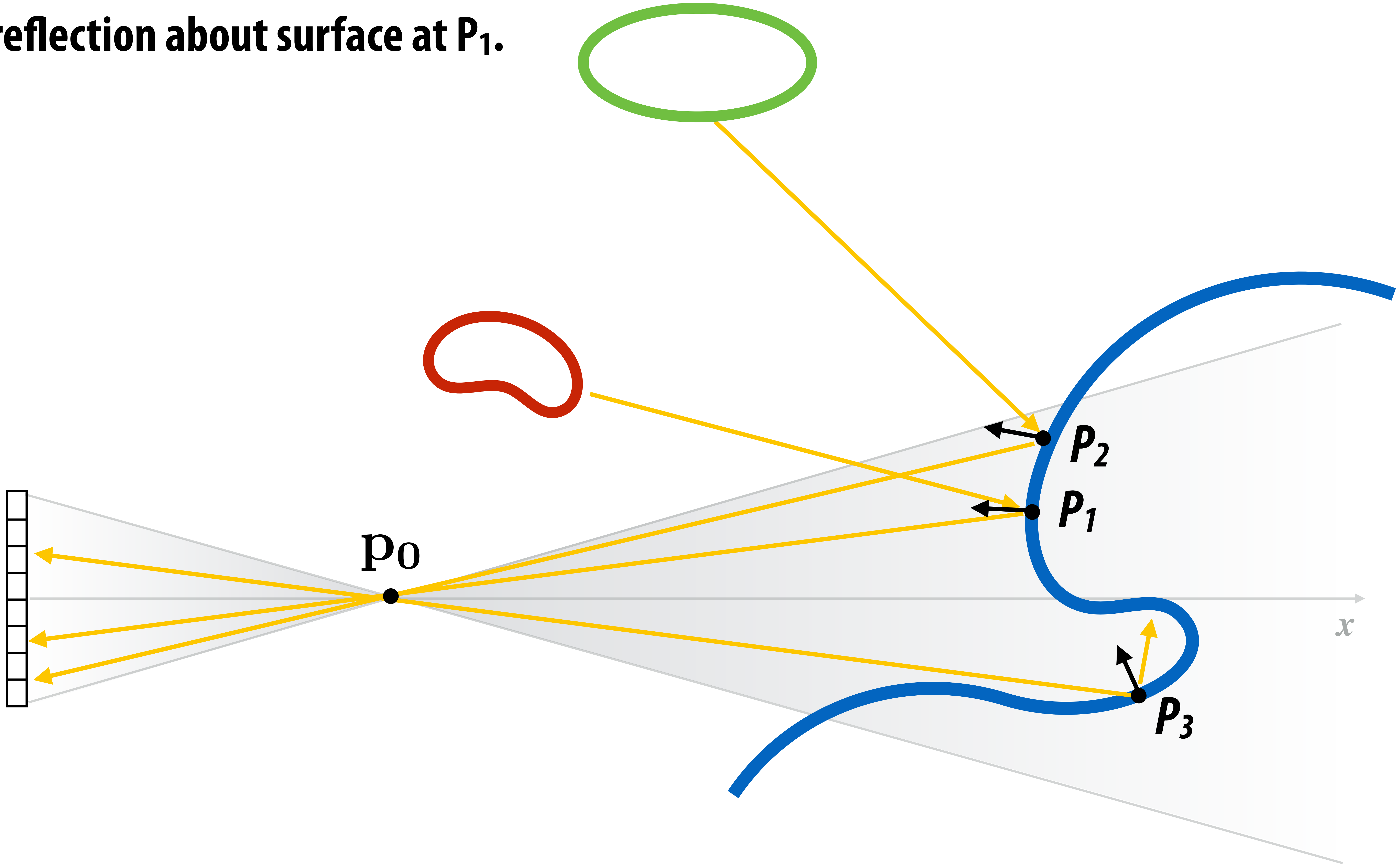
Reflections

Recall: perfect mirror material



Recall: perfect mirror reflection

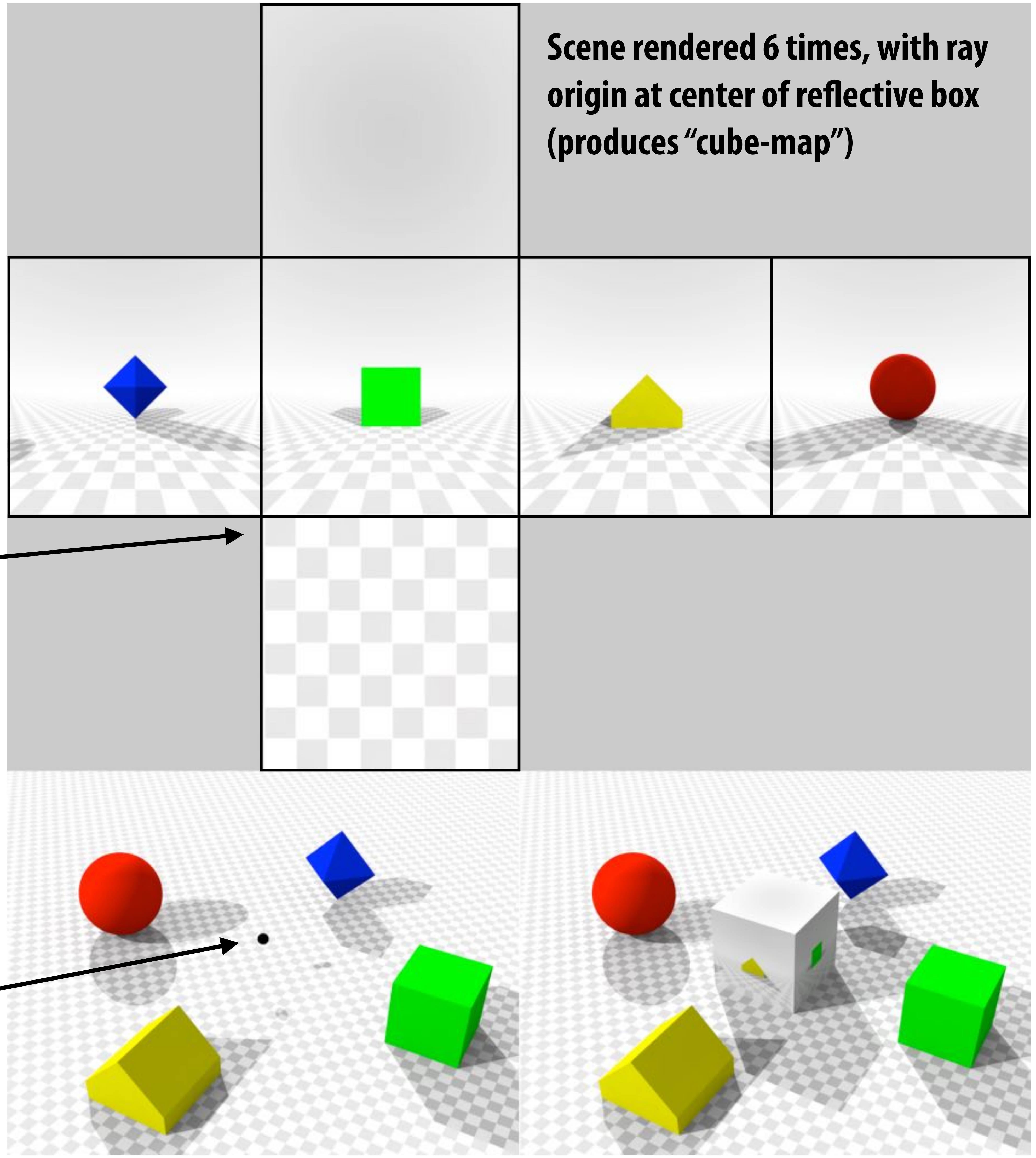
Light reflected from P_1 in direction of P_0 is incident on P_1 from reflection about surface at P_1.



Rasterization: "camera" position can be reflection point

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

Environment map vs. ray traced reflections

Image credit: Control

https://www.techspot.com/article/1934-the-state-of-ray-tracing/

Environment map vs. ray traced reflections

Image credit: Control

https://www.techspot.com/article/1934-the-state-of-ray-tracing/

Indirect lighting

Indirect lighting

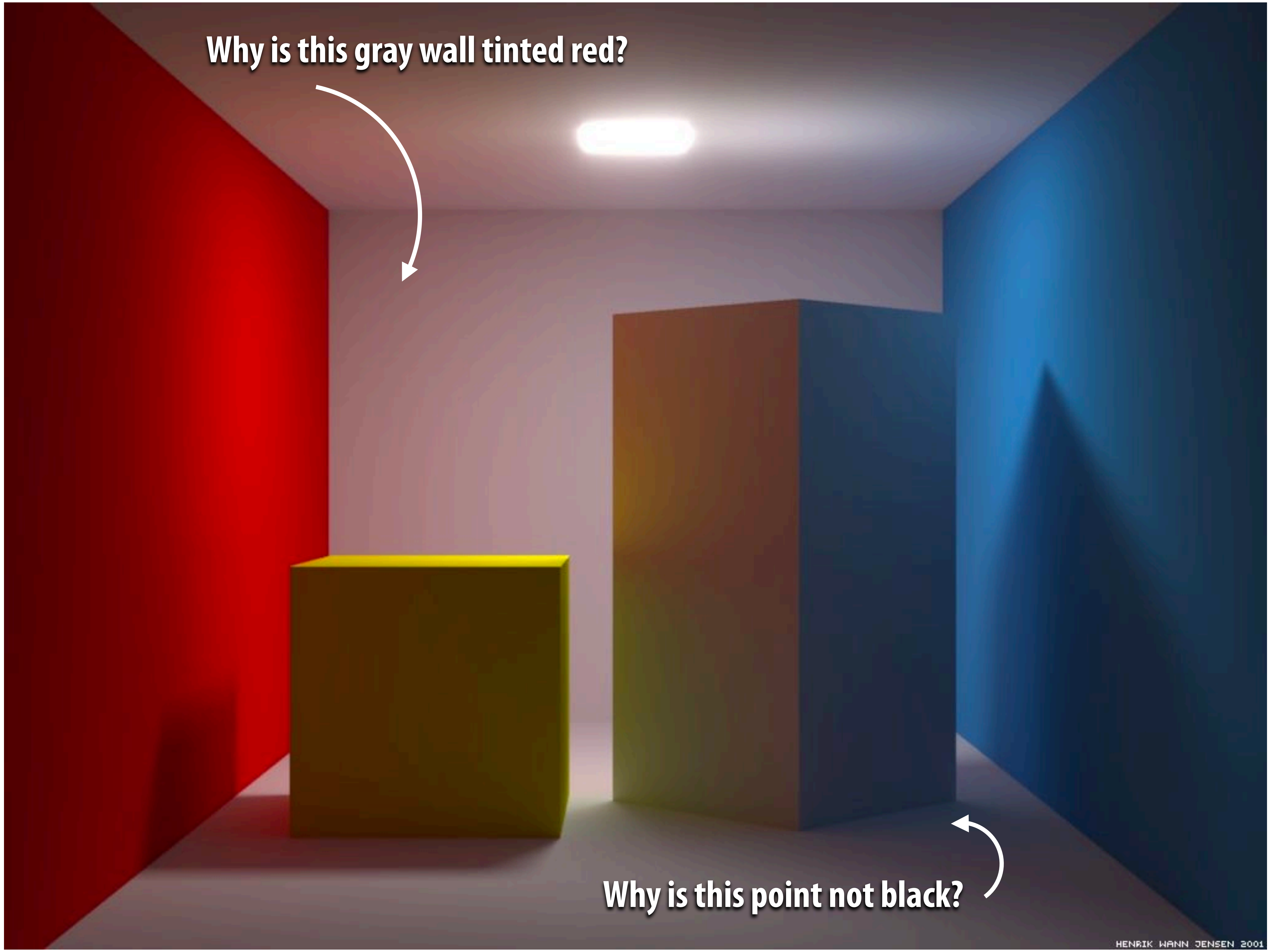
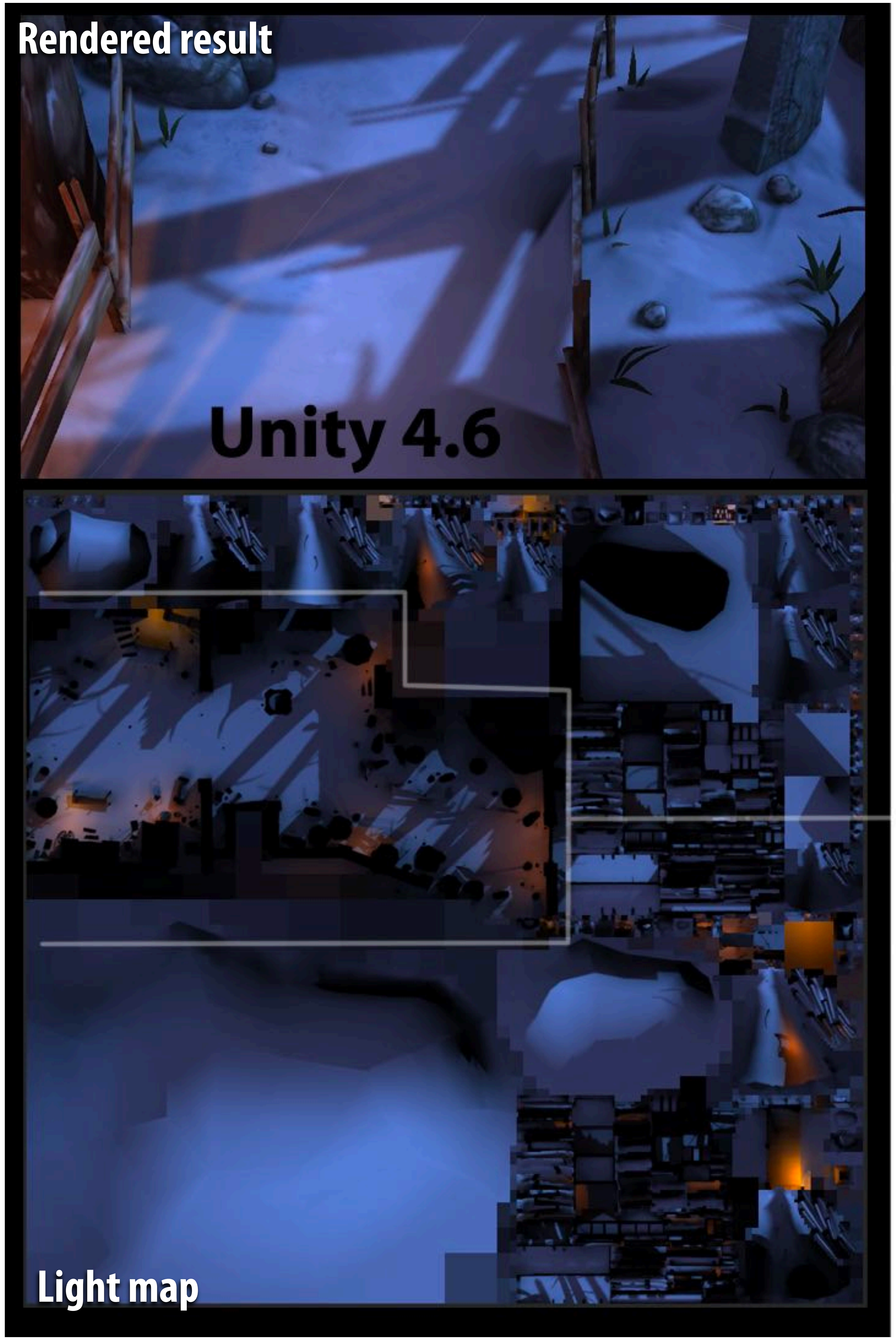


Image credit: Henrik Wann Jensen

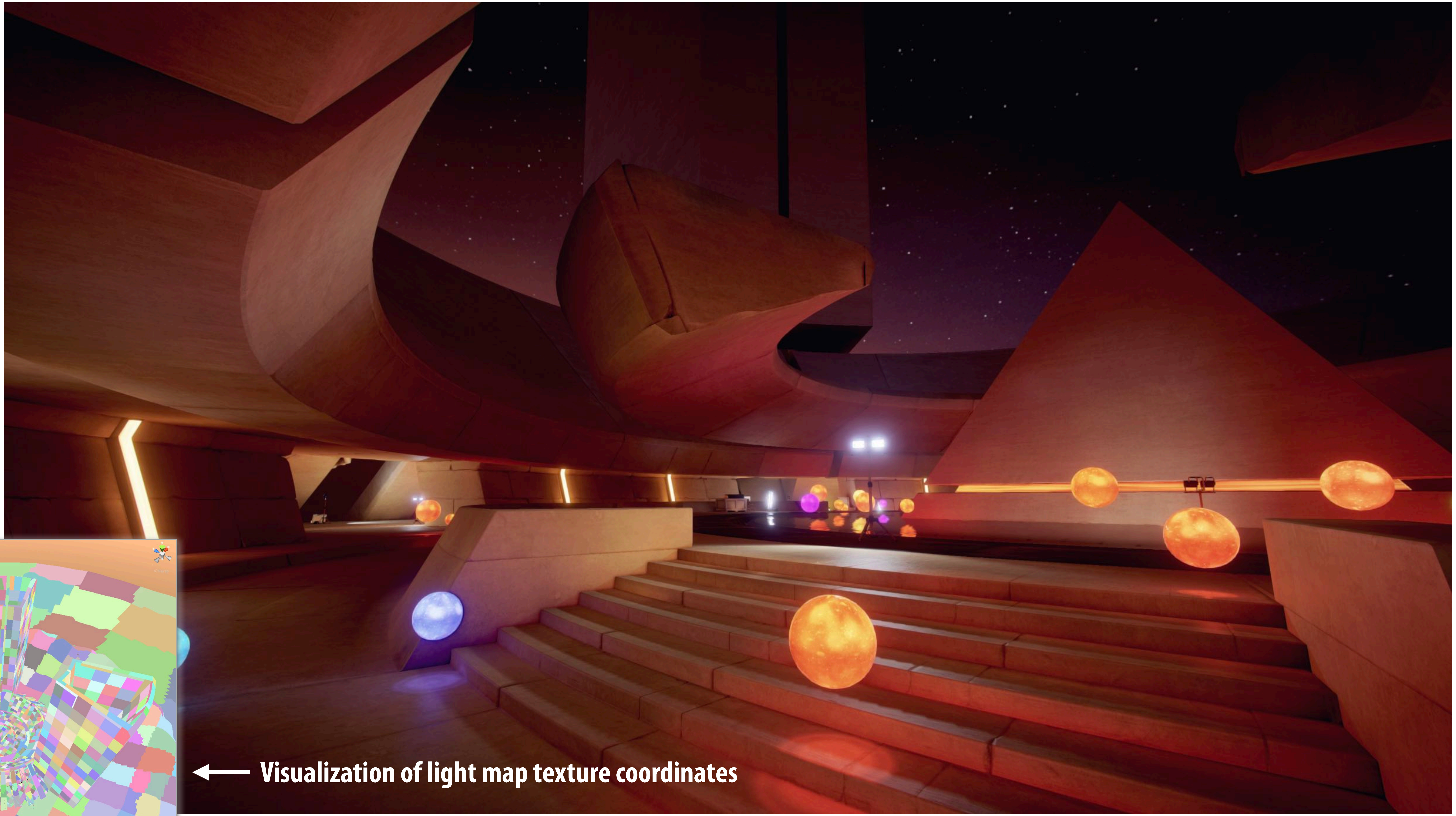
Precomputed lighting

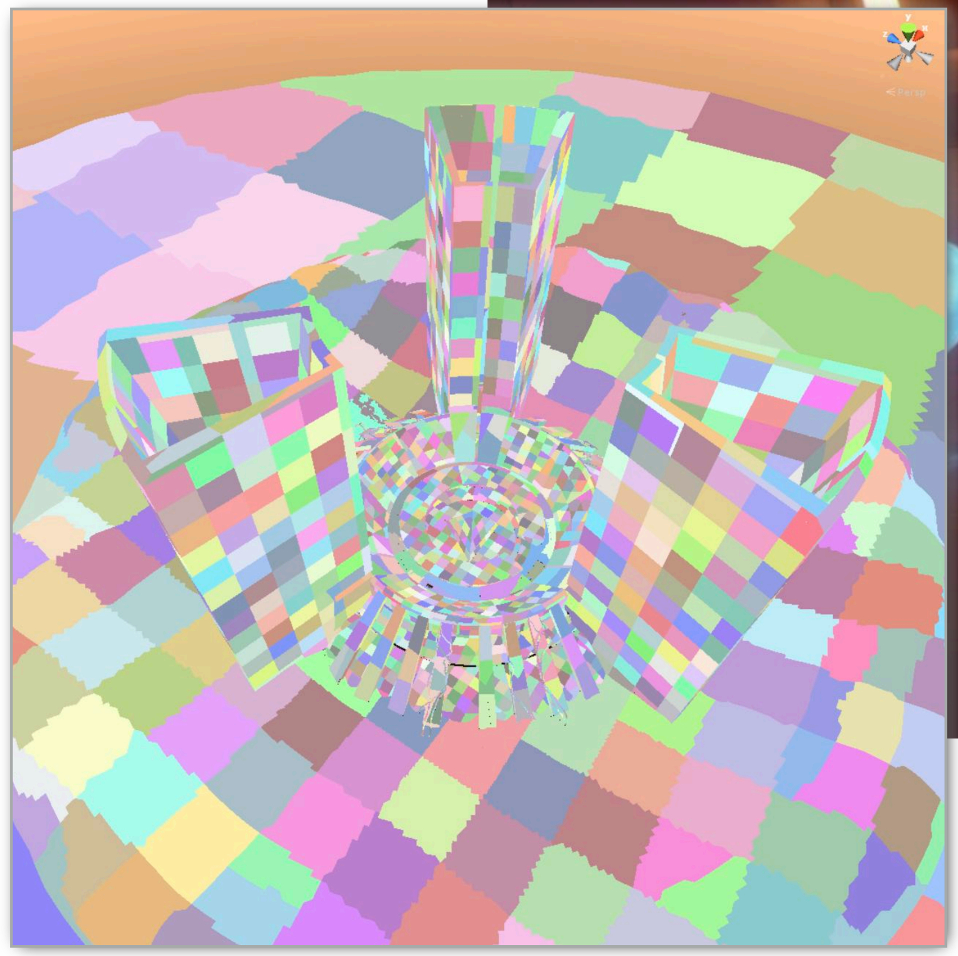
- Precompute accurate lighting for a scene offline using a ray tracer (possible for static lights)

- "Bake" results of lighting into texture map



Precomputed lighting in Unity Engine





← Visualization of light map texture coordinates

Image credit: Unity / Alex Lovett

Today, there's increasing use of real-time ray tracing

- I've just shown you an array of different techniques for approximating different advanced lighting phenomenon using a rasterizer
- Challenges:
  - Different algorithm for each effect (code complexity)
  - Algorithms may not compose
  - They are only approximations to the physically correct solution ("hacks!")
- These techniques were adopted because historically tracing rays to solve these problems was too costly for real-time us

This image was ray traced in real-time on a GPU

Real-time ray tracing challenge:

Need to shoot many rays per pixel to accurately estimate the value of the rendering equation integral

Want high-performance interactive rendering

😥

Innovation 1: hardware acceleration

Supercomputing for games

- \NVIDIA Titan X GPU

(~ 7 TFLOPs fp32)Tesla generation NV chip ~ ASCI Red

NVIDIA Founder's Edition RTX 4090 GPU

~ 82 TFLOPs fp32 *

* Doesn't include additional 190 TFLOPS of ray tracing compute and 165 TFLOPS of fp15 DNN compute

Specialized processors for performing graphics computations.

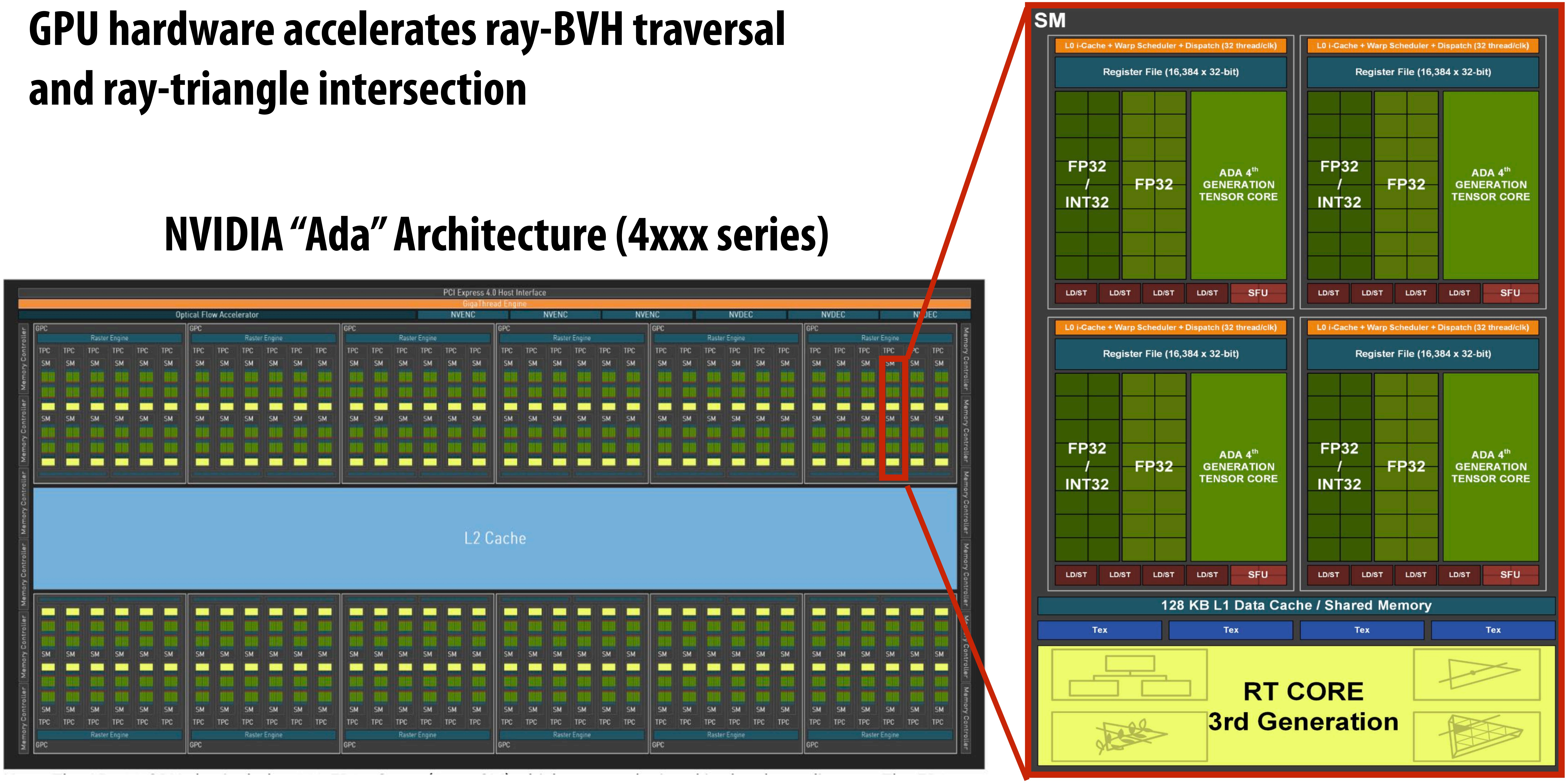
Innovation 1:
Hardware innovation: custom GPU hardware for RT

NVIDIA GeForce RTX 4090 GPU

Fixed-function hardware for ray tracing

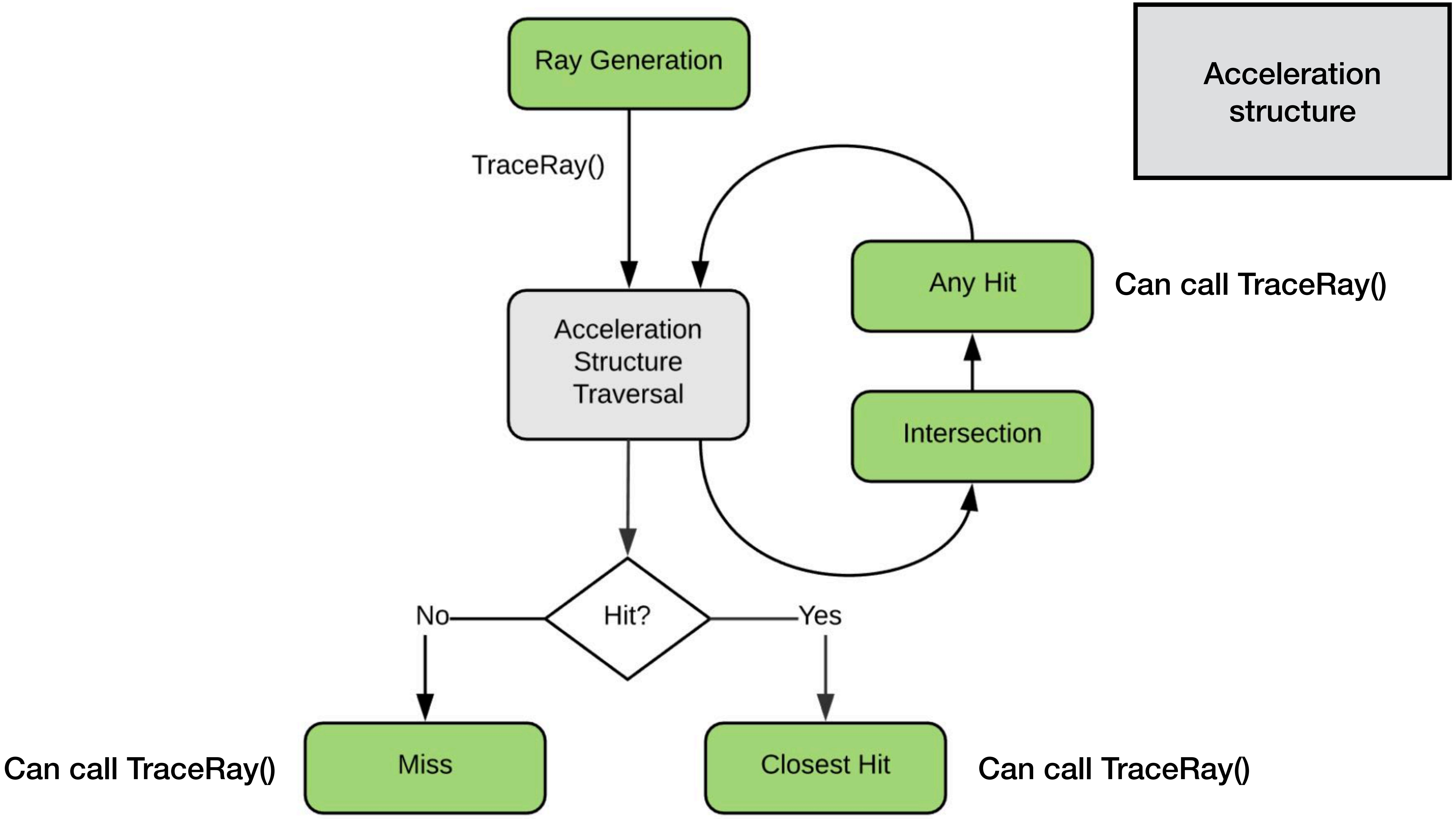
- GPU hardware accelerates ray-BVH traversal and ray-triangle intersection

NVIDIA "Ada" Architecture (4xxx series)



D3D12 API has DXR ray tracing "stages"

- Ray tracing is abstracted as a graph of programmable "stages"
- TraceRay() is a function available in some of those stages



Example: ray generation shader (creates camera rays)

```
// This represents the geometry of our scene.
RaytracingAccelerationStructure scene : register(t5);

[shader("raygeneration")]
void RayGenMain()
{
    // Get the location within the dispatched 2D grid of work items
    // (often maps to pixels, so this could represent a pixel coordinate).
    uint2 launchIndex = DispatchRaysIndex();

    // Define a ray, consisting of origin, direction, and the t-interval
    // we're interested in.
    RayDesc ray;
    ray.Origin = SceneConstants.cameraPosition.
    ray.Direction = computeRayDirection( launchIndex ); // assume this function exists
    ray.TMin = 0;
    ray.TMax = 100000;

    Payload payload;

    // Trace the ray using the payload type we've defined.
    // Shaders that are triggered by this must operate on the same payload type.
    TraceRay( scene, 0 /*flags*/, 0xFF /*mask*/, 0 /*hit group offset*/,
              1 /*hit group index multiplier*/, 0 /*miss shader index*/, ray, payload );

    outputTexture[launchIndex.xy] = payload.color;
}
```

Example "hit shader": Runs on ray hit to fill in payload

```
// Attributes contain hit information and are filled in by the intersection shader.
// For the built-in triangle intersection shader, the attributes always consist of
// the barycentric coordinates of the hit point.
struct Attributes
{
    float2 barys;
};

[shader("closesthit")]
void ClosestHitMain( inout Payload payload, in Attributes attr )
{
    // Read the intersection attributes and write a result into the payload.
    payload.color = float4( attr.barys.x, attr.barys.y,
                            1 - attr.barys.x - attr.barys.y, 1 );

    // Demonstrate one of the new HLSL intrinsics: query distance along current ray
    payload.hitDistance = RayTCurrent();
}
```

Innovation 2: more intelligent importance sampling

Recall "perfect" importance sampling

- **Drawing samples from distribution proportion to f(x) yields zero variance estimates (only need a single sample to estimate an integral if you draw that sample according to f(x)**

- **But impractical because to know p(x), you need to know the integral you are trying to estimate!**

$$\tilde{p}(x) = cf(x)$$ ⟵ **Normalization to make a pdf**

$$c = \frac{1}{\int f(x)dx}$$

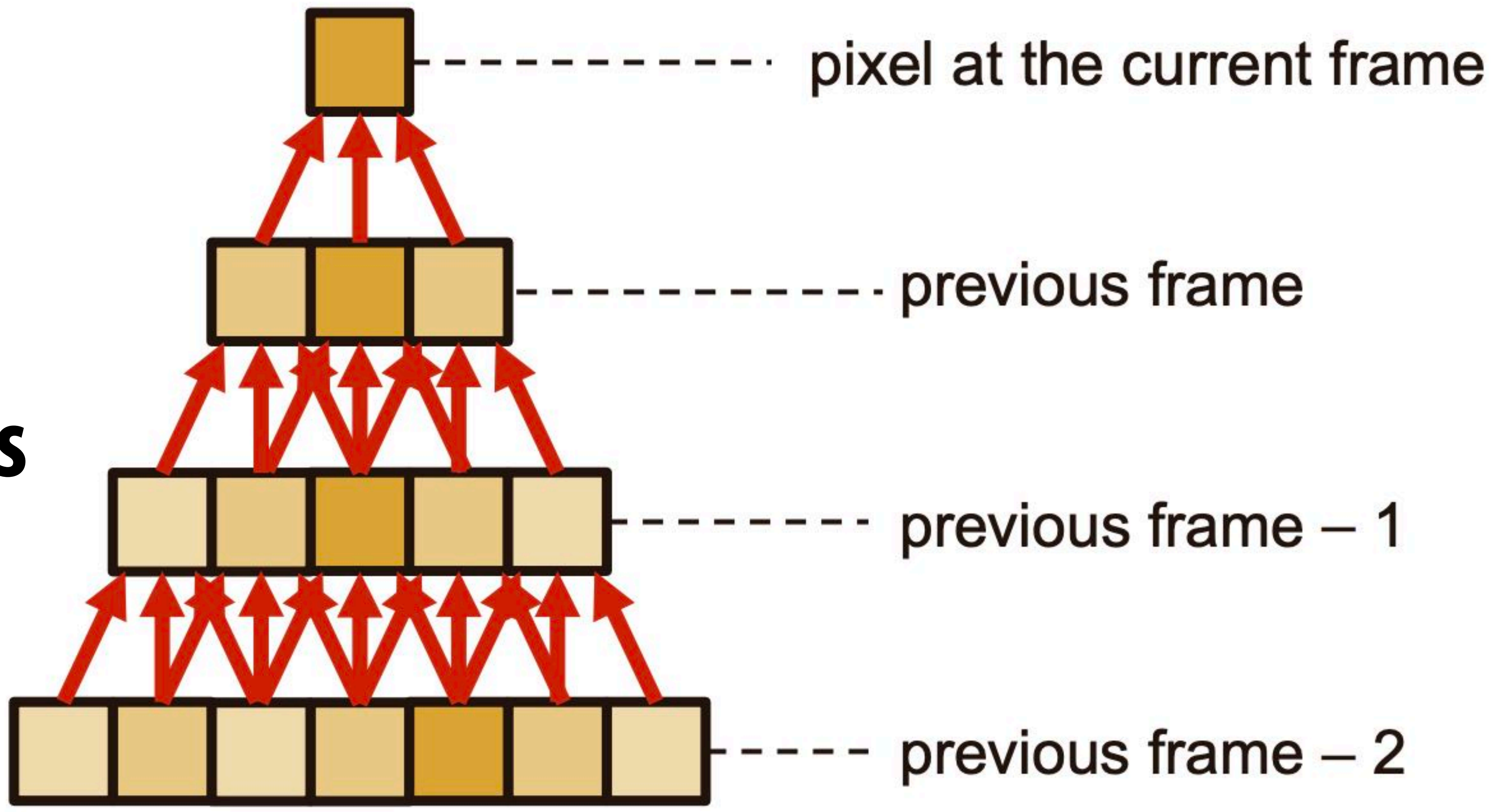
$$\tilde{f}(x) = \frac{f(x)}{p(x)} = \frac{f(x)}{cf(x)} = \frac{1}{c}$$

Generalized MC estimator (regardless of what sample we draw, our estimator is 1/c)
So variance in the estimate after taking N samples is 0.

$$\frac{1}{c} = \int f(x)\,\mathrm{d}x$$

Resampled importance sampling

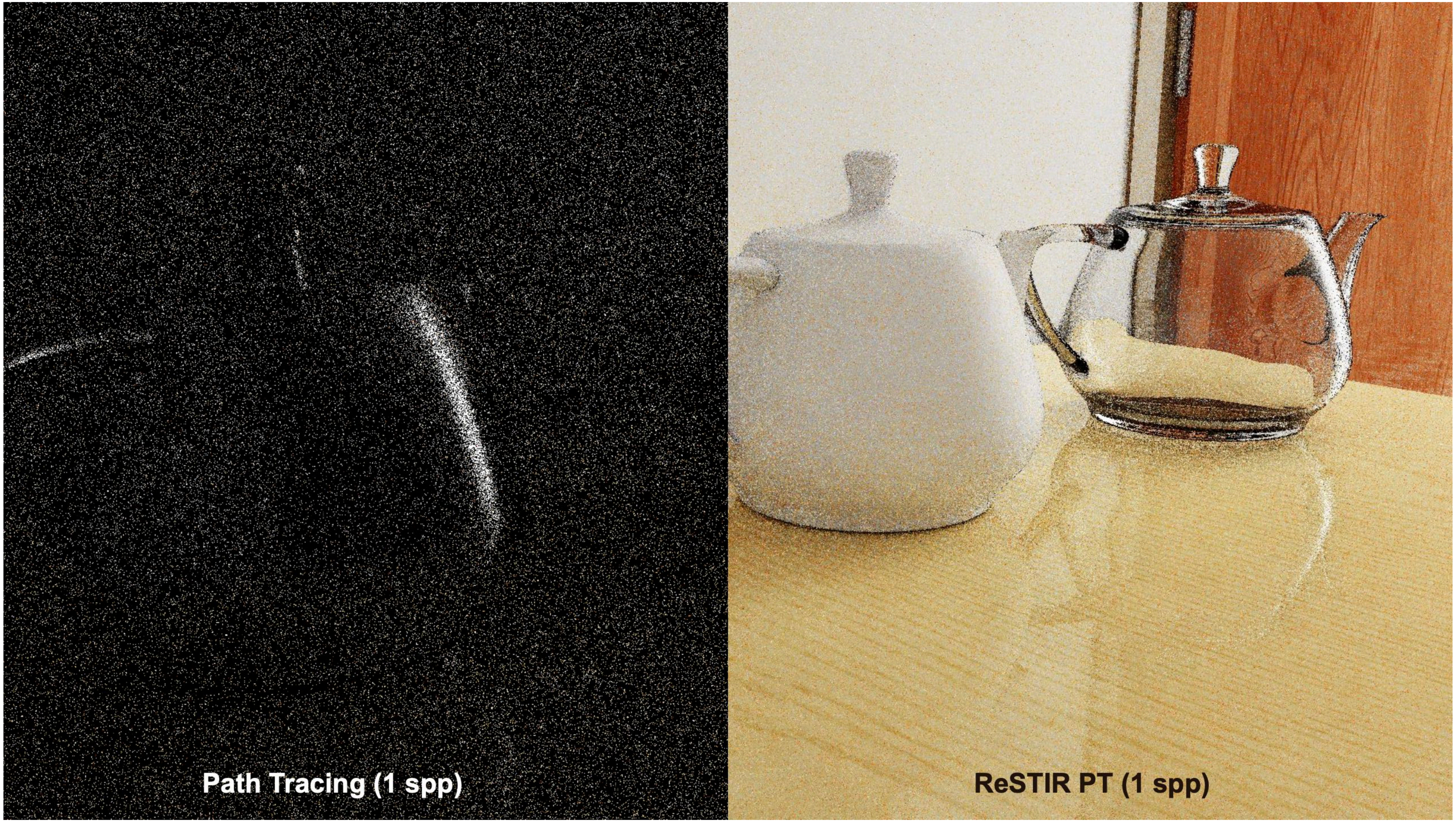
- Modern variance reduction techniques in ray tracing (ReSTIR = "resampled spatiotemporal importance sampling) try to approximate the ideal pdf cf(x) by randomly samples from a set of prior chosen samples ("resampling")
  - Nearby samples in "space" (samples chosen to compute integrals nearby on screen)
  - Nearby samples in "times" (samples chosen at the same screen location in prior frames)



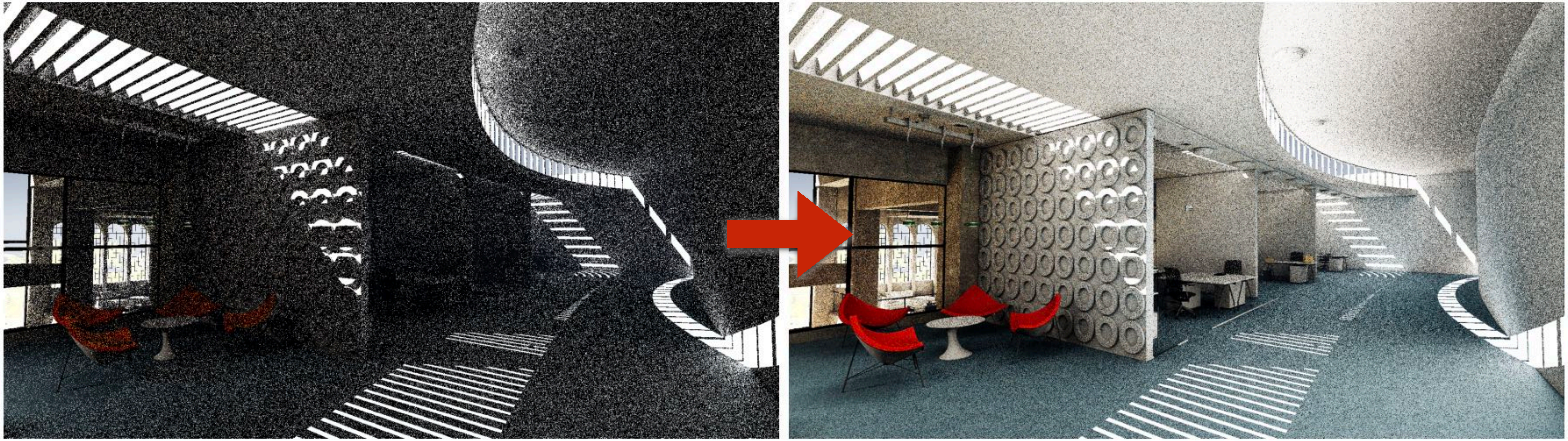
Suggested reference for learning more:
"A Gentle Introduction to ReSTIR: Path Reuse in Real-time", SIGGRAPH 2023 course notes *

* Disclaimer: gentle can be in the eye of the writer

Better importance sampling reduces required ray count



Better importance sampling algorithms



Path traced: 1 path/pixel (8 ms/frame)

Path traced: 1 path/pixel using ReSTIR GI (8.9 ms/frame)

Key idea: cache good paths, reuse good paths found from from prior frames or for prior pixels in same frame

[Ouyang et al. 2021]

Innovation 3: Neural network based denoising

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

This image was rendered using many paths per pixel (expensive)

Recall: numerical integration of light (via Monte Carlo sampling) suffers from high variance, resulting in images with "noise"

16 paths/pixel


64 paths/pixel


256 paths/pixel


1024 paths/pixel


4096 paths/pixel

Rendering of surface albedo ("material color")
(no illumination — very cheap)

Rendering of surface normals
(no illumination — very cheap)

Denoised results

16 paths/pixel


16 paths/pixel (denoised)


64 paths/pixel (denoised)
VICTROLA

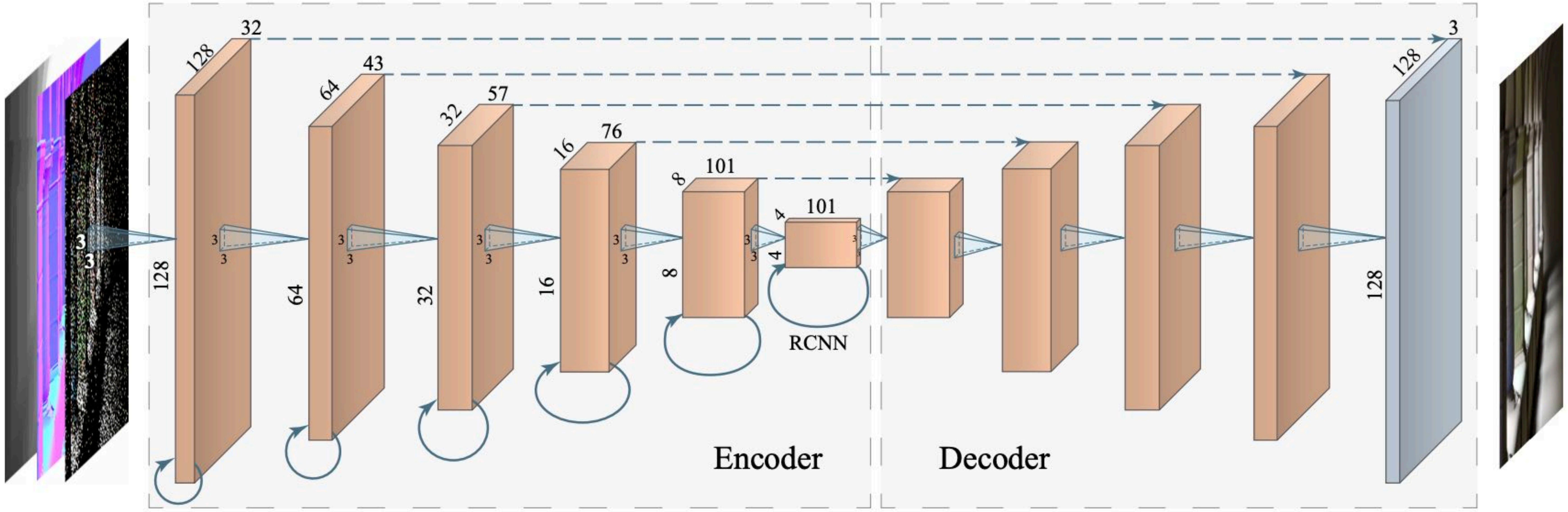
256 paths/pixel (denoised)


1024 paths/pixel (denoised)

4096 paths/pixel (denoised)

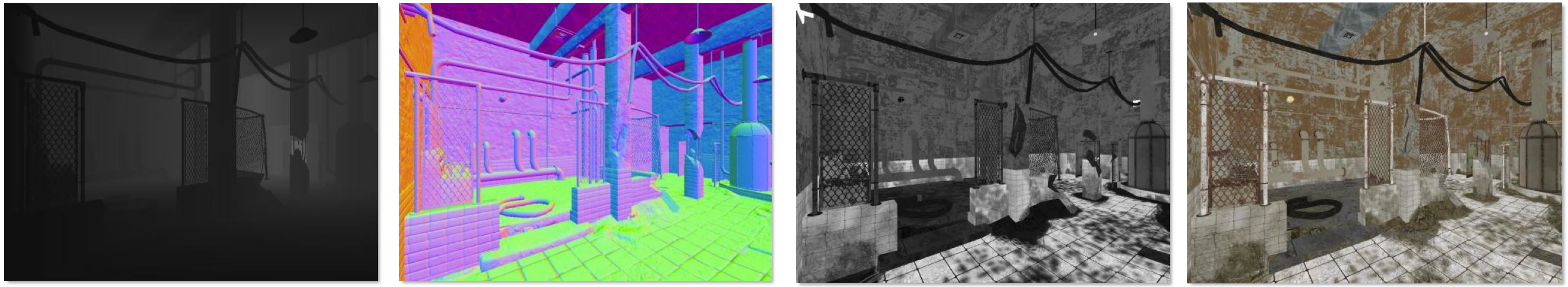
4096 paths/pixel (NOT DENOISED)

Example: neural denoiser DNN



Input to network is noisy RGB image * + additional normal, depth, and roughness channels
(These are cheap to compute inputs help network identify silhouettes, sharp structure)

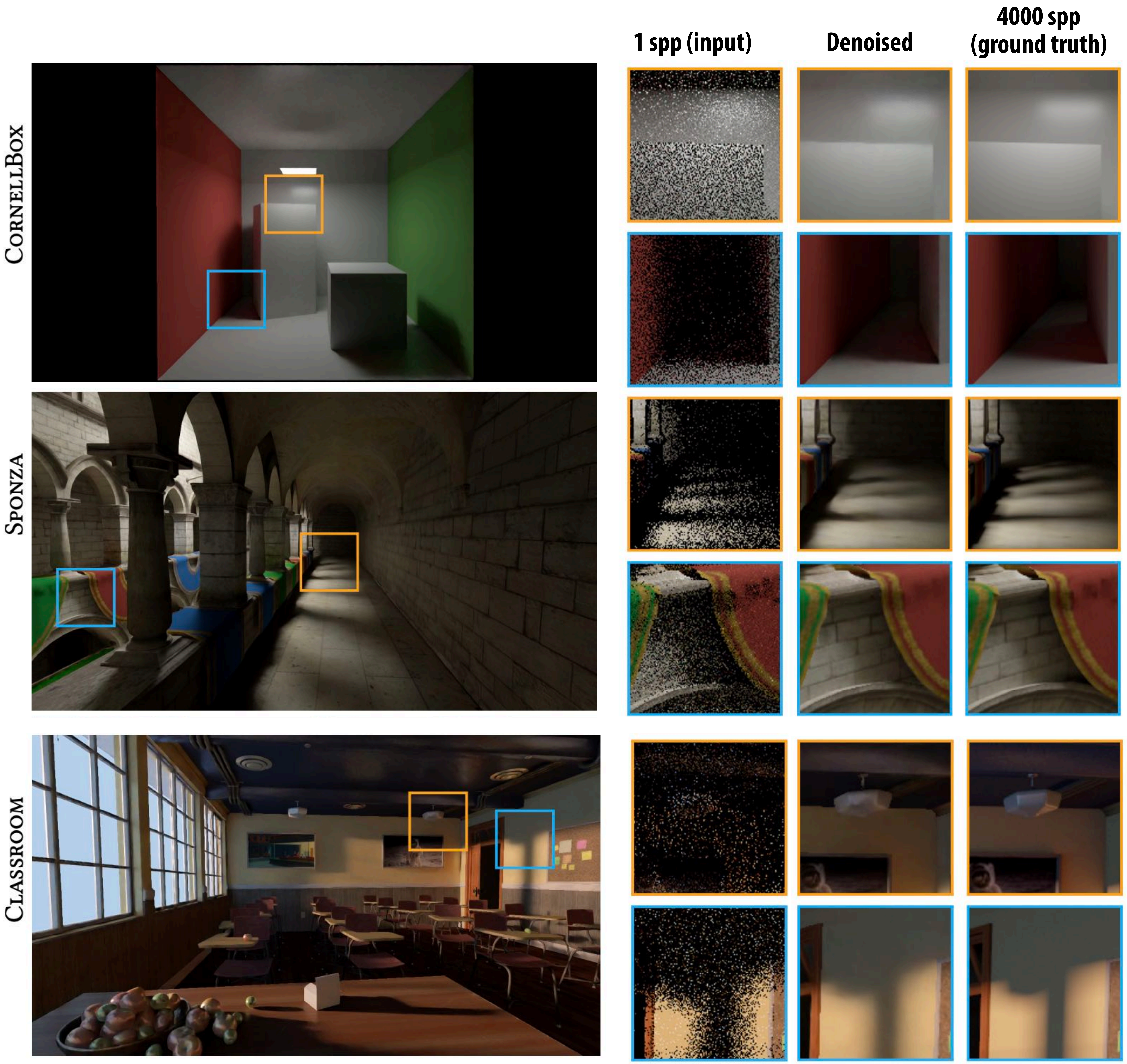
Depth **Normal** **Roughness** **Albedo**



*** Actually the input is RGB demodulated by (divided by) texture albedo (don't force network to learn what texture was)**

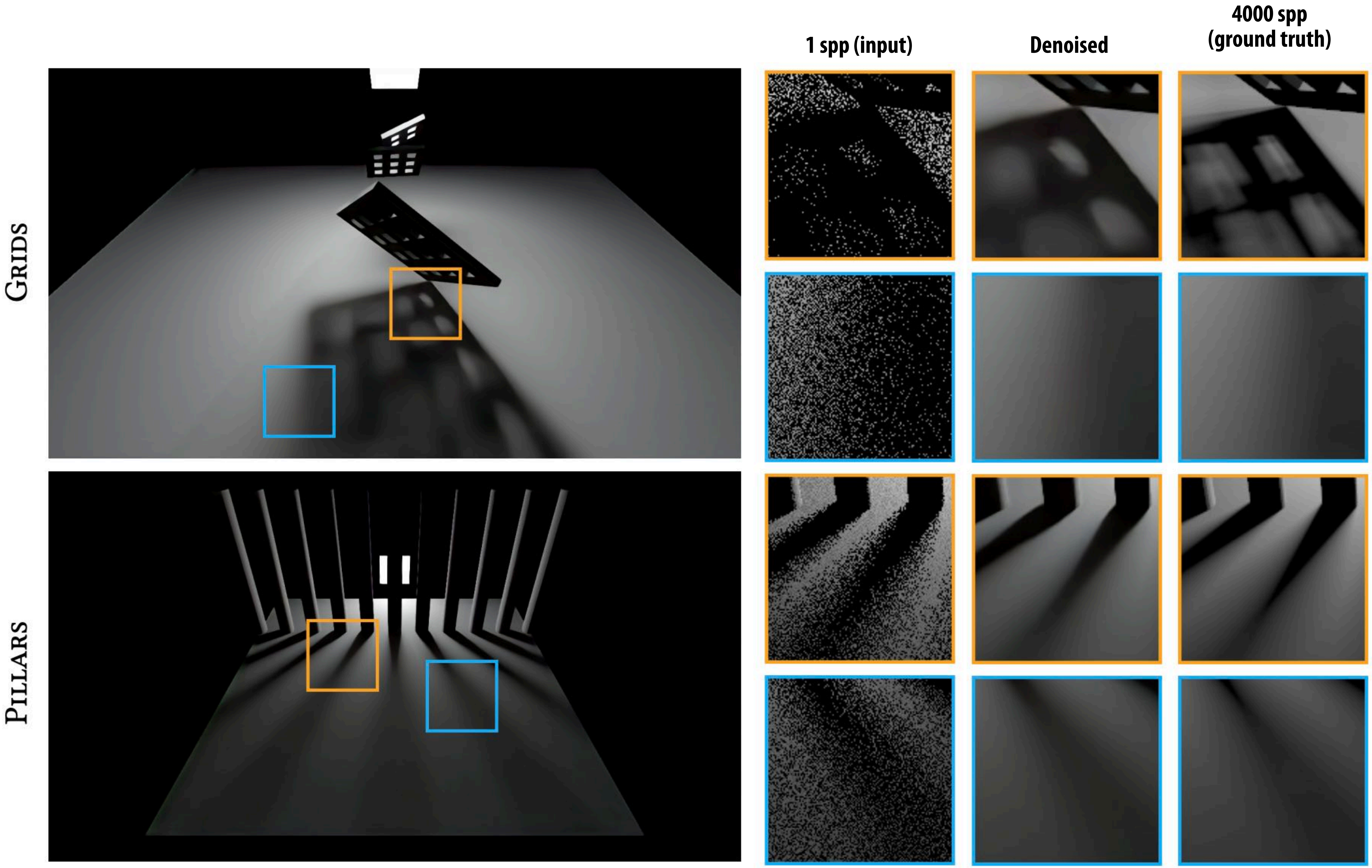
Denoising results

[Chaitanya 17]



[Chaitanya 17]

Denoising results (challenging)



More denoising examples

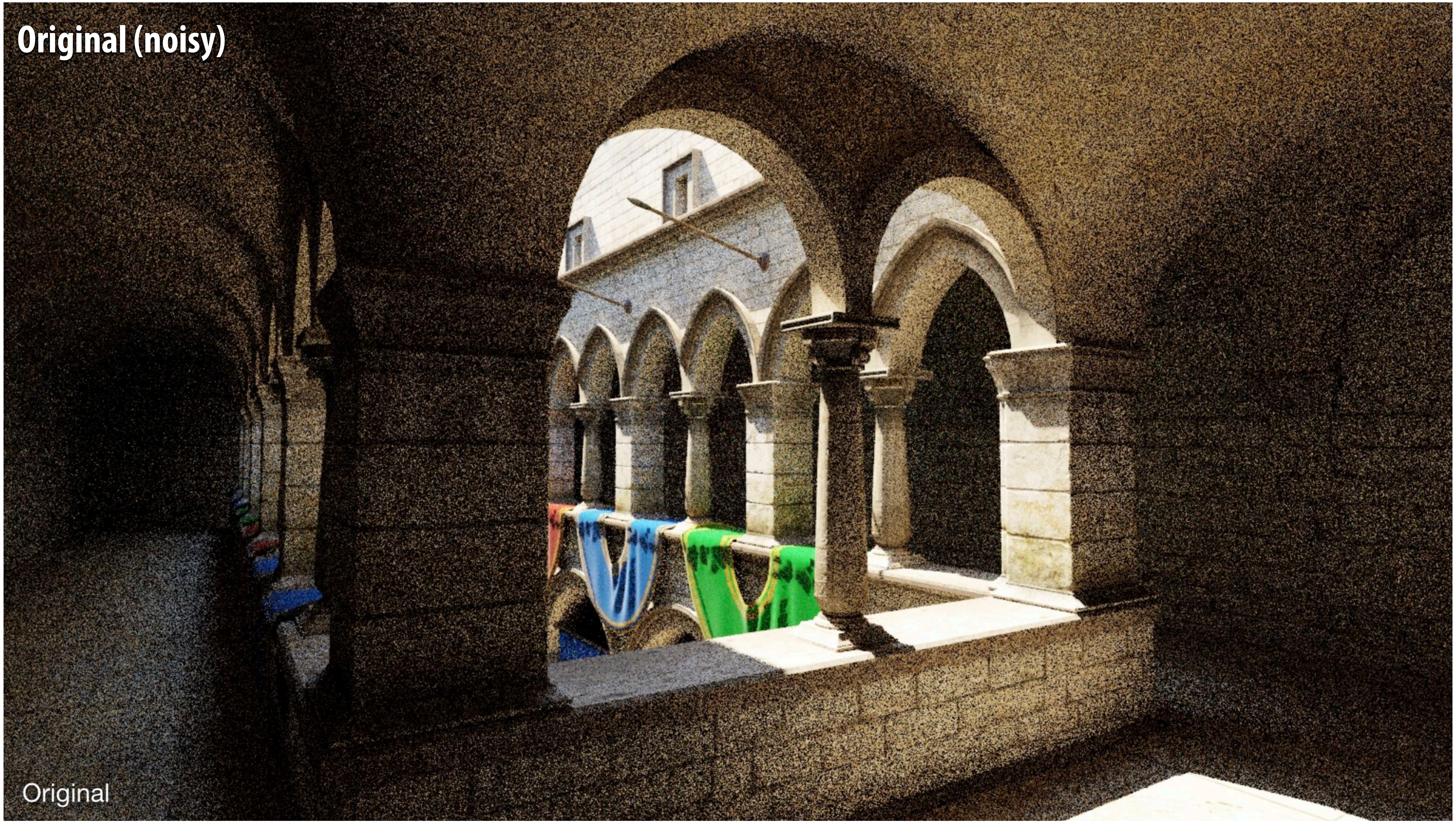


Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

More denoising examples

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

More denoising examples

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

More denoising examples

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

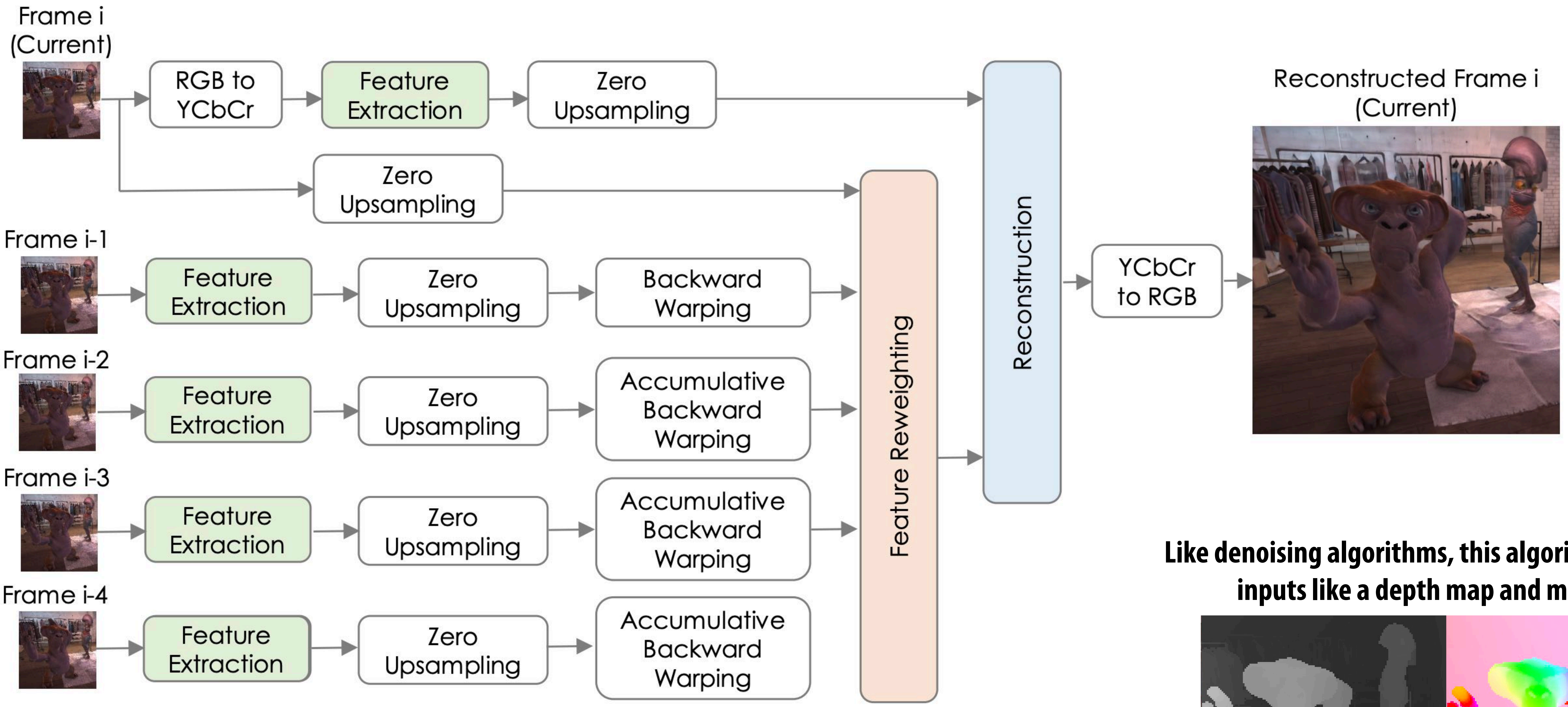
Neural upsampling (hallucinating detail)

Note: now we are talking about upsampling (increasing image resolution), not denoising

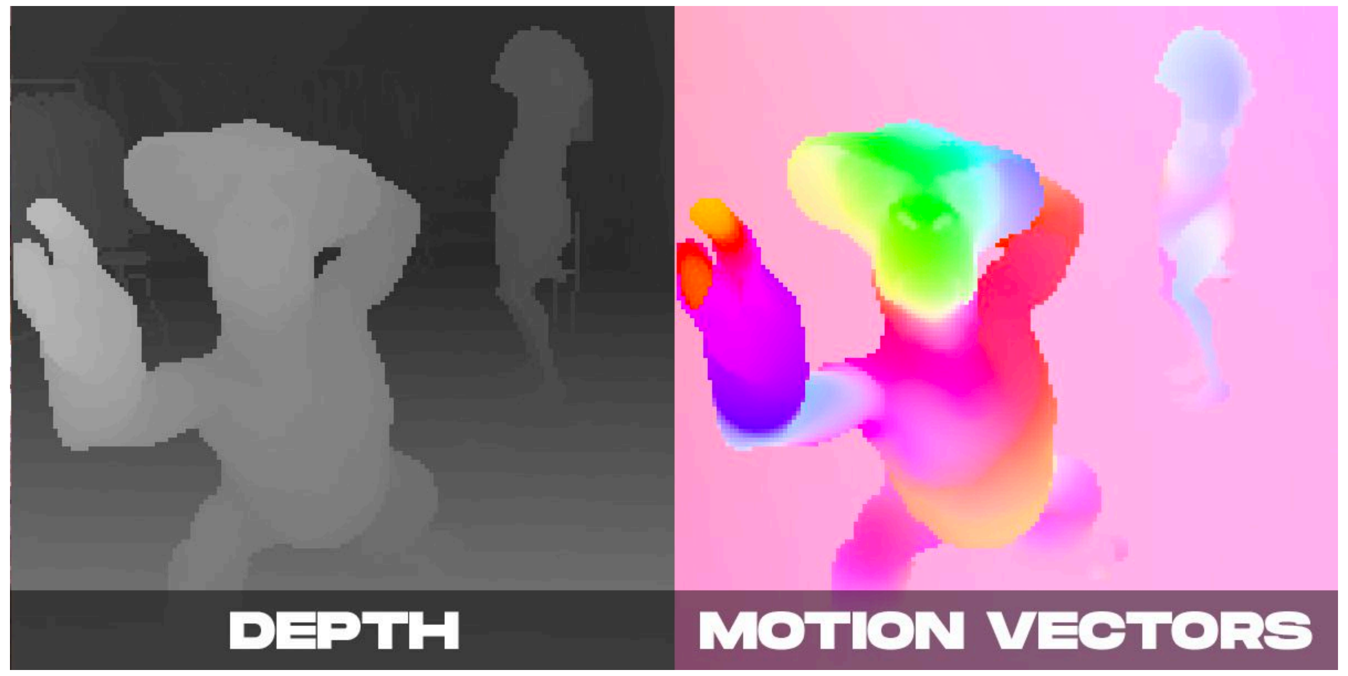
Neural upsampling (hallucinating detail)

4x4 upsampled result (16x more pixels)

Neural upsampling pipeline



Like denoising algorithms, this algorithm using auxiliary inputs like a depth map and motion vectors



Main idea: gain resolution by aligning and merging multiple recent frames

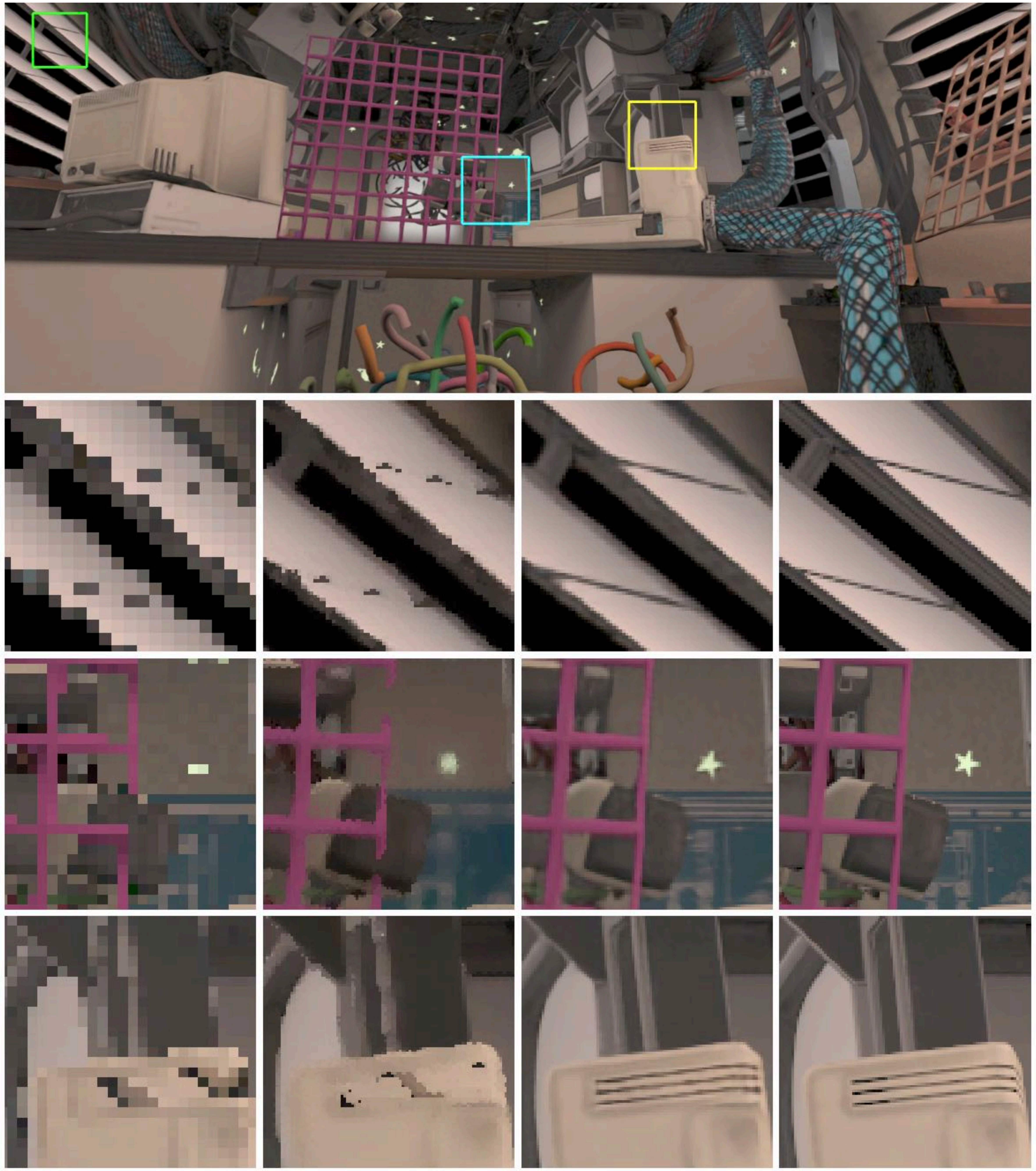
Frame-to-frame alignment vectors provided by renderer

Learn a model that determines weights for combining aligned features ("feature reweighting")

Then decode with neural decoder ("reconstruction")

Closer look

[Xiao 20]



Summary: neural methods + rendering

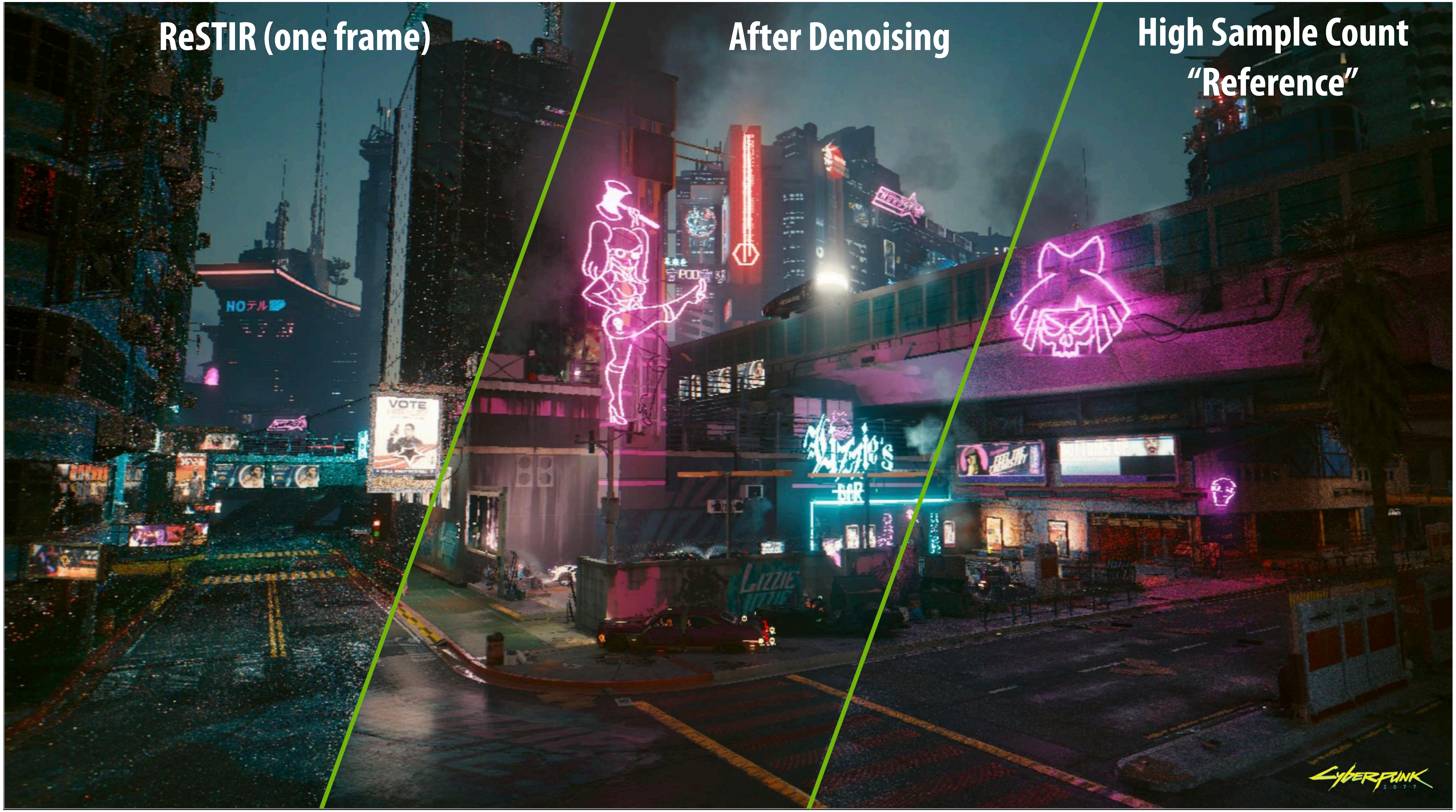
- Neural methods now used to:
  - Denoise images
  - Upsample images
  - Increase frame rate (temporal upsampling = frame interpolation)
  - Anti-alias images

- All of these post-processing techniques serve to reduce the number of rays needed to make a picture

- You can think of the responsibility of a modern ray tracer/renderer as: produce enough samples of the scene so ML can "take it the rest of the way" and robustly hallucinate a high-quality image.

Modern renderers designed in conjunction with denoiser

Image from Cyberpunk 2077



Interactive ray tracing summary

- Until very recently, it was too expensive to perform ray tracing in real-time graphics systems
- So the computer graphics field developed many rasterization-based methods for approximating ray traced effects (shadows, reflections, etc).
- In last decade: a major shift toward using more ray tracing in real-time graphics systems
- Driven by three innovations:
  - Brute force: new ray tracing hardware supported by graphics APIs (D3D12/Vulkan) increases the number of rays that can be traced per second
  - Algorithmic innovation: smarter ways to importance sample paths
  - Introduction of ML into rendering: use ML to convert noisy low sample count images to images that "look like" images that were ray traced at high sample counts, or to increase the resolution of rendered images