Lecture 11:

Modern Rendering Techniques Using the Graphics Pipeline

Interactive Computer Graphics
Stanford CS248, Winter 2022

Screenshot: Red Dead Redemption

Screenshot: Far Cry 5

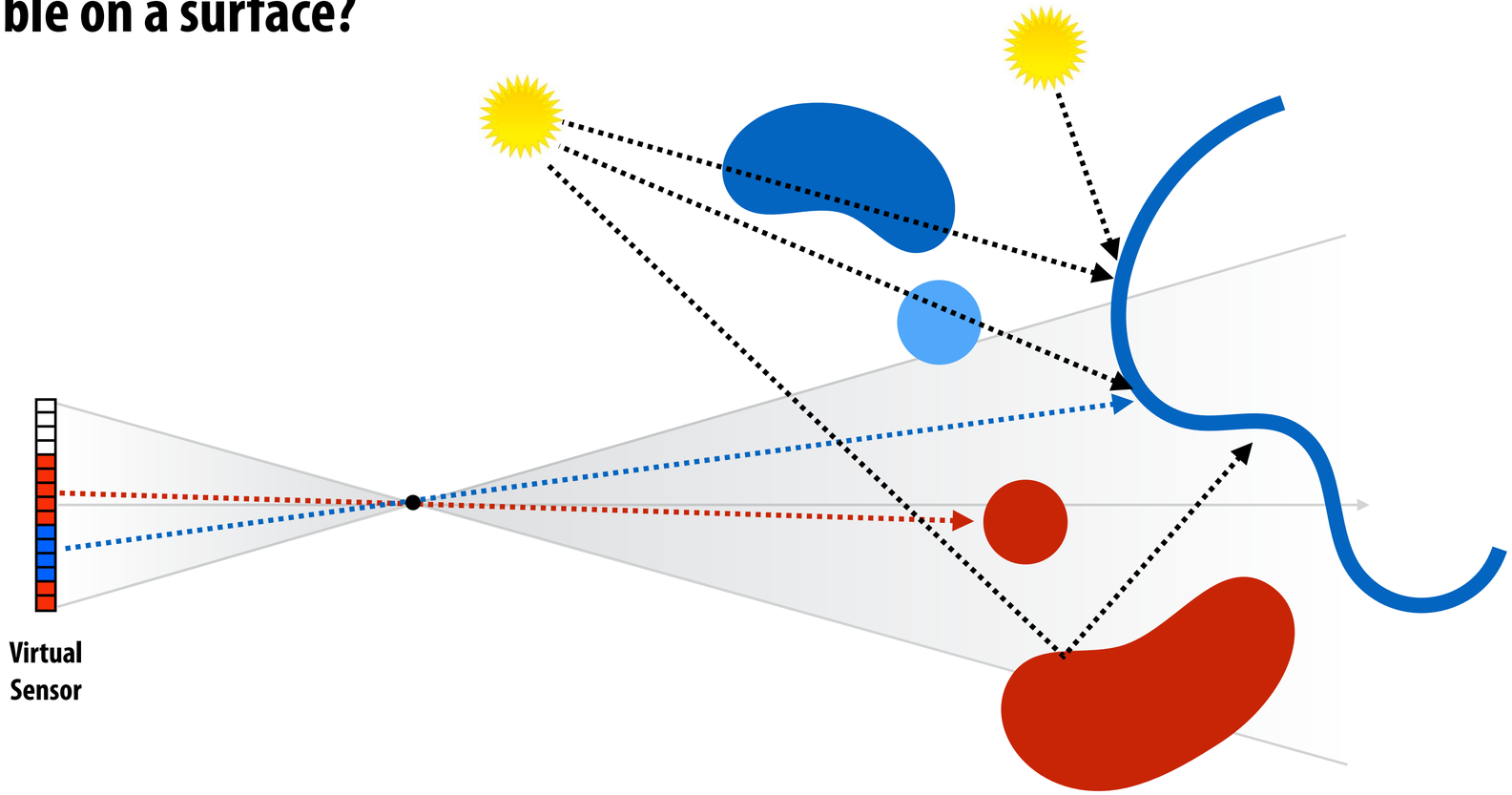
Screenshot: Battlefield V

Last week: 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?)

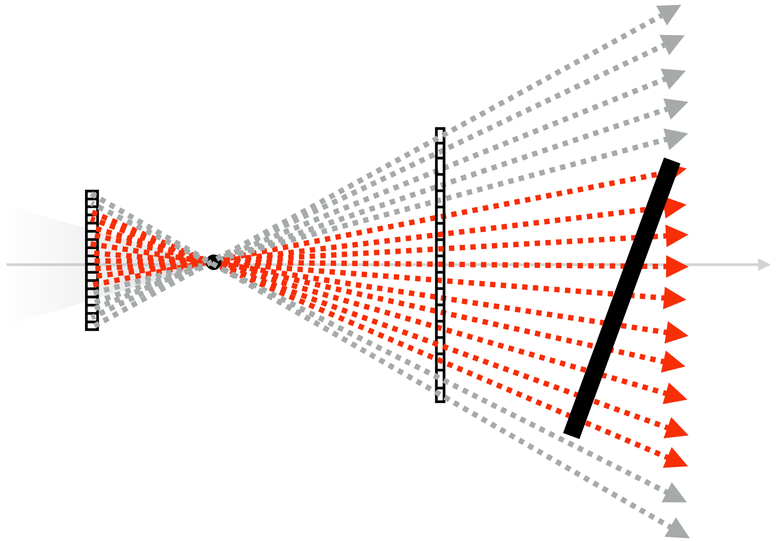
What reflection is visible on a surface?

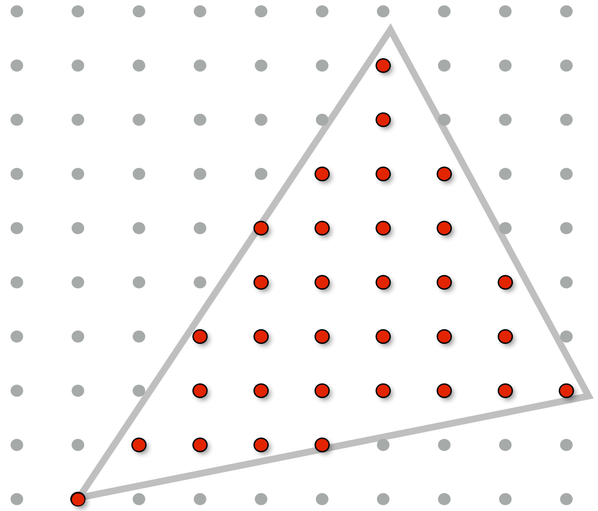


Rasterization algorithm for "camera ray"- scene queries

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





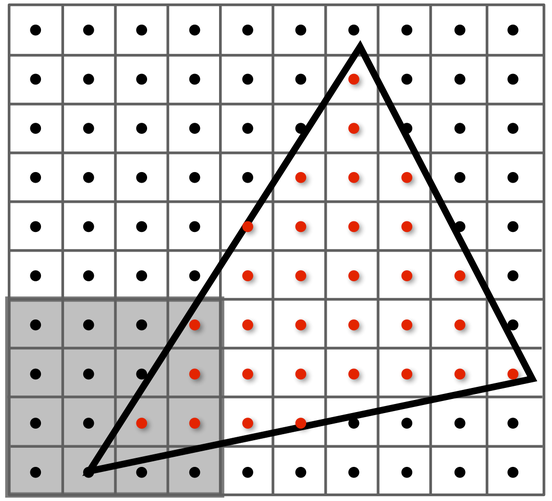
Review: basic rasterization algorithm

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)

More efficient hierarchical rasterization:

For each TILE of image

If triangle overlaps tile, check all samples in tile

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 surface color of triangle r.tri at hit point
```

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

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

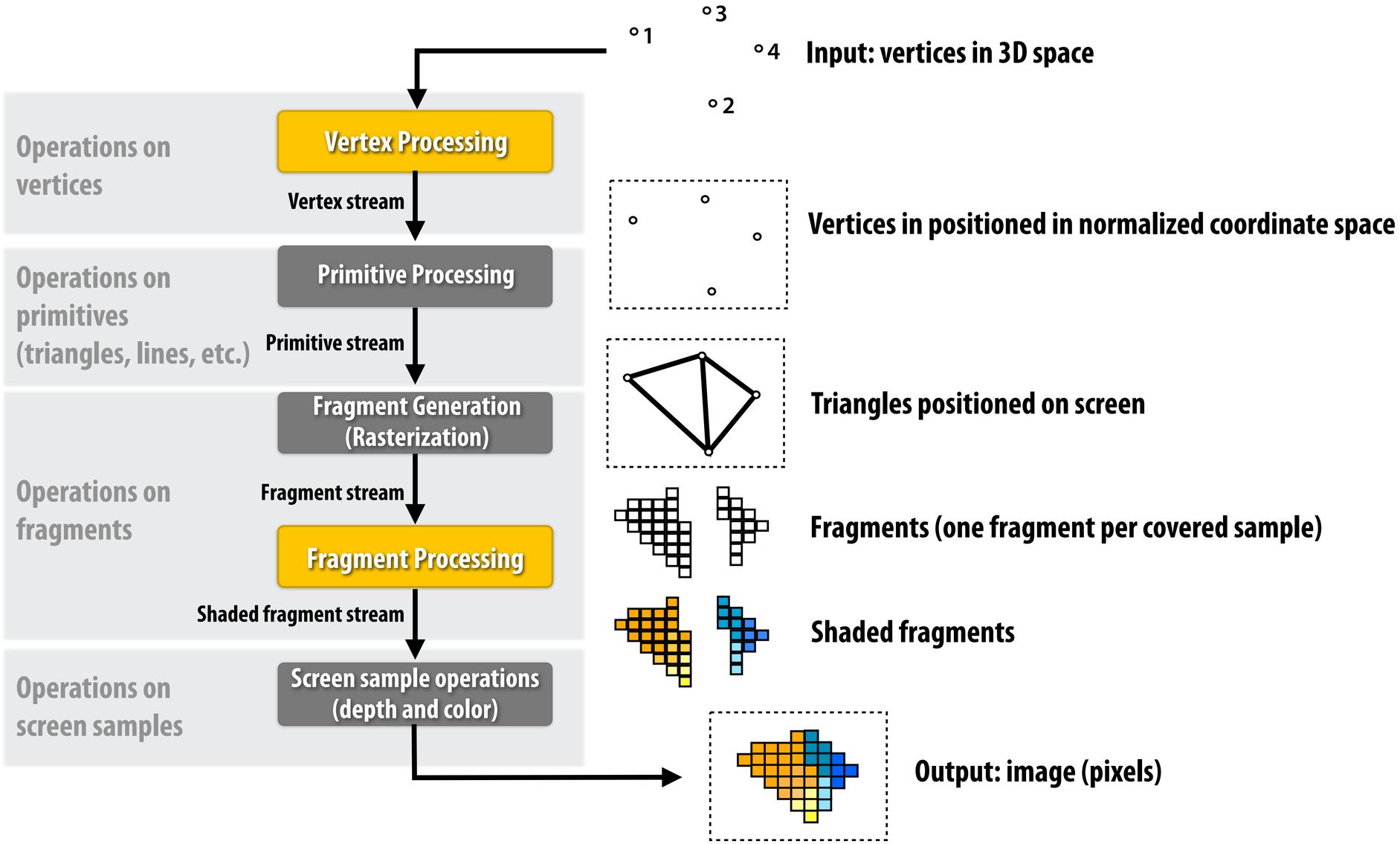
Theme of this part of the lecture:

A surprising number of advanced lighting effects can be *efficiently approximated* using the basic primitives of the rasterization pipeline, without the need to actually ray trace the scene geometry:

- **Rasterization**
- **Texture mapping**
- **Depth buffer for occlusion**

OpenGL/Direct3D graphics pipeline

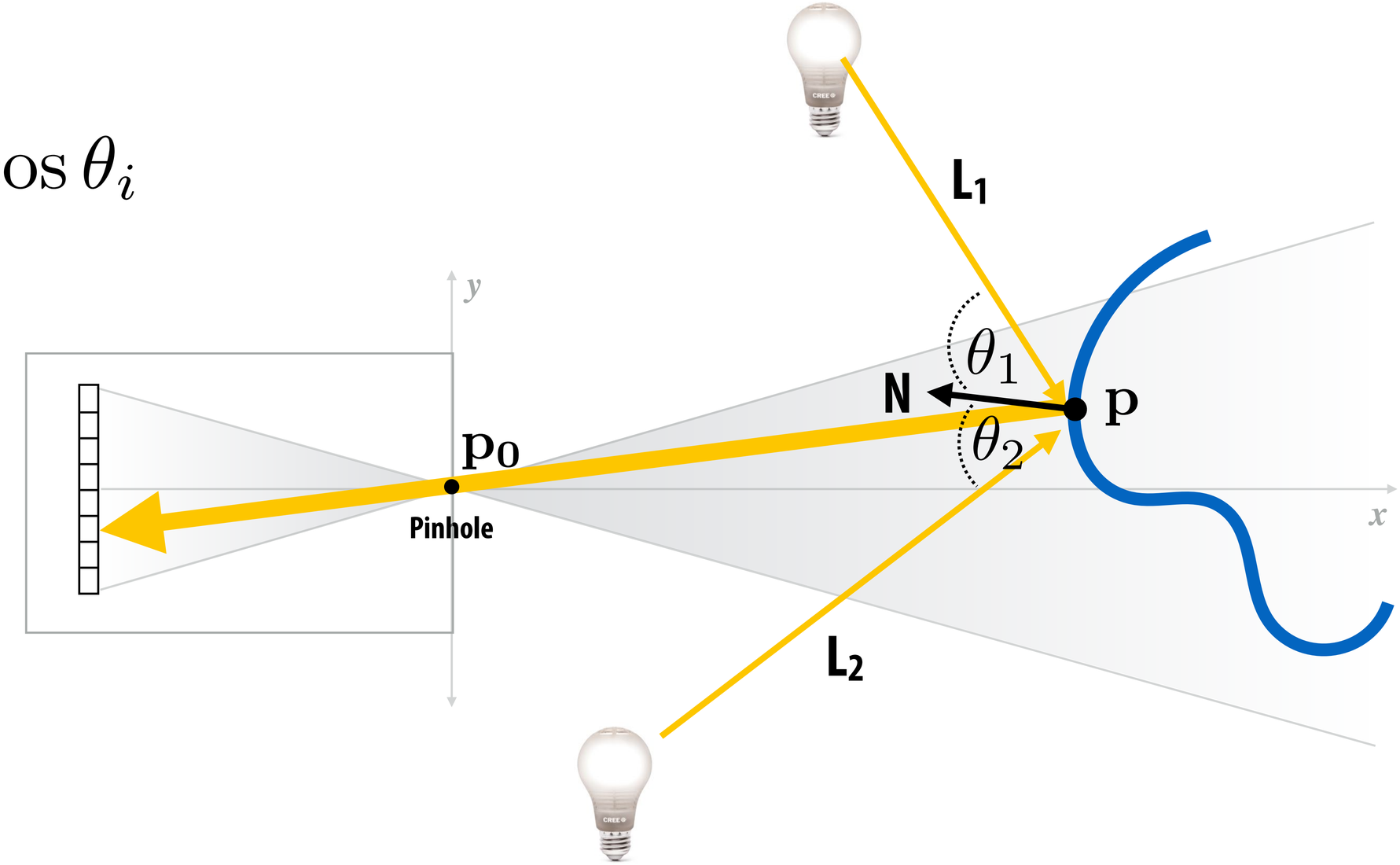
* Several stages of the modern OpenGL pipeline are omitted



Review: how much light hits the surface at point p

(This is light per unit surface area at point P… irradiance at point P)

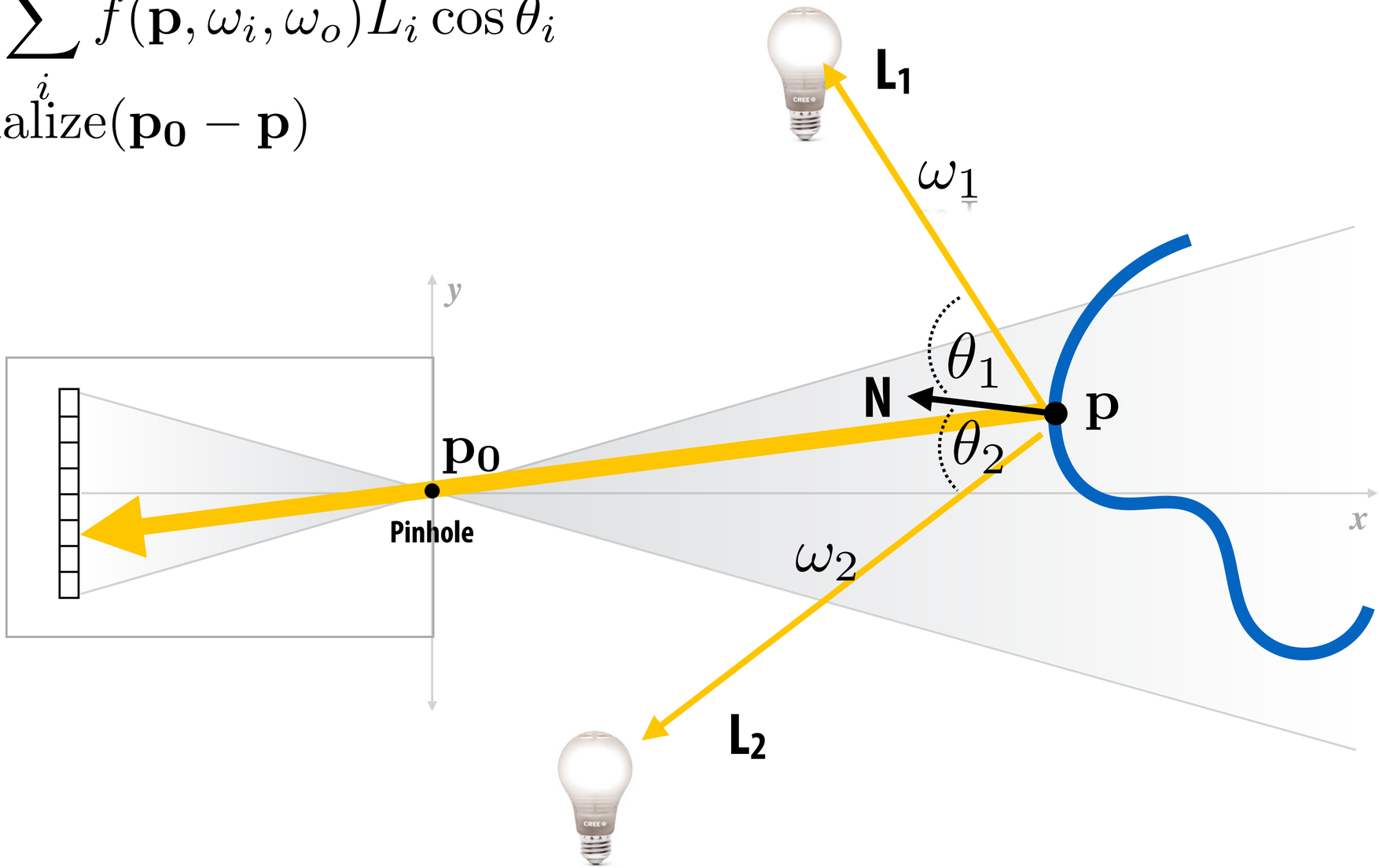
$$\sum_i L_i \cos\theta_i$$



How much light is REFLECTED from p toward p_0

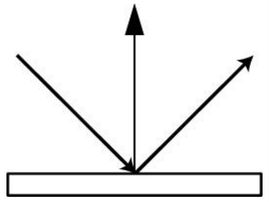
$$L(\mathbf{p}, \omega_o) = \sum_i f(\mathbf{p}, \omega_i, \omega_o) L_i \cos\theta_i$$

$$\omega_o = \text{normalize}(\mathbf{p_0} - \mathbf{p})$$

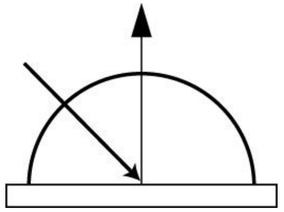


Some basic reflection functions $f(\mathbf{p}, \omega_i, \omega_o)$

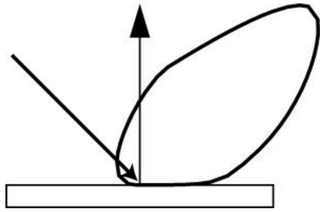
- **Ideal specular**
  Perfect mirror



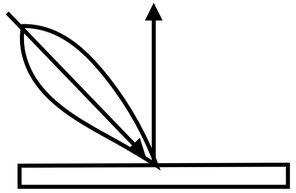
- **Ideal diffuse**
  Uniform reflection in all directions




- **Glossy specular**
  Majority of light distributed in reflection direction



- **Retro-reflective**
  Reflects light back toward source




Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

Shadows

Shadows

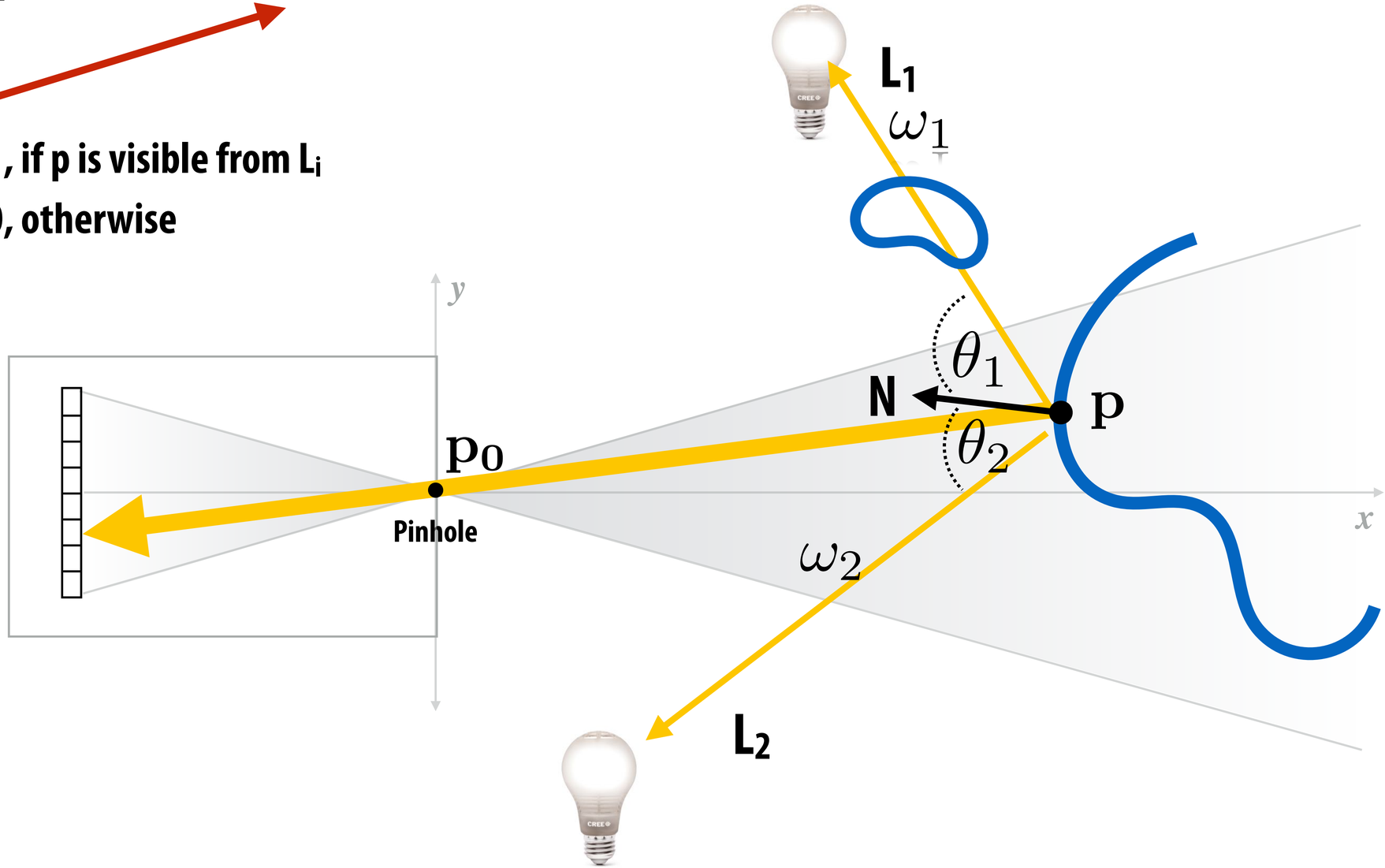
Image credit: Grand Theft Auto V

How much light is REFLECTED from p toward p_0

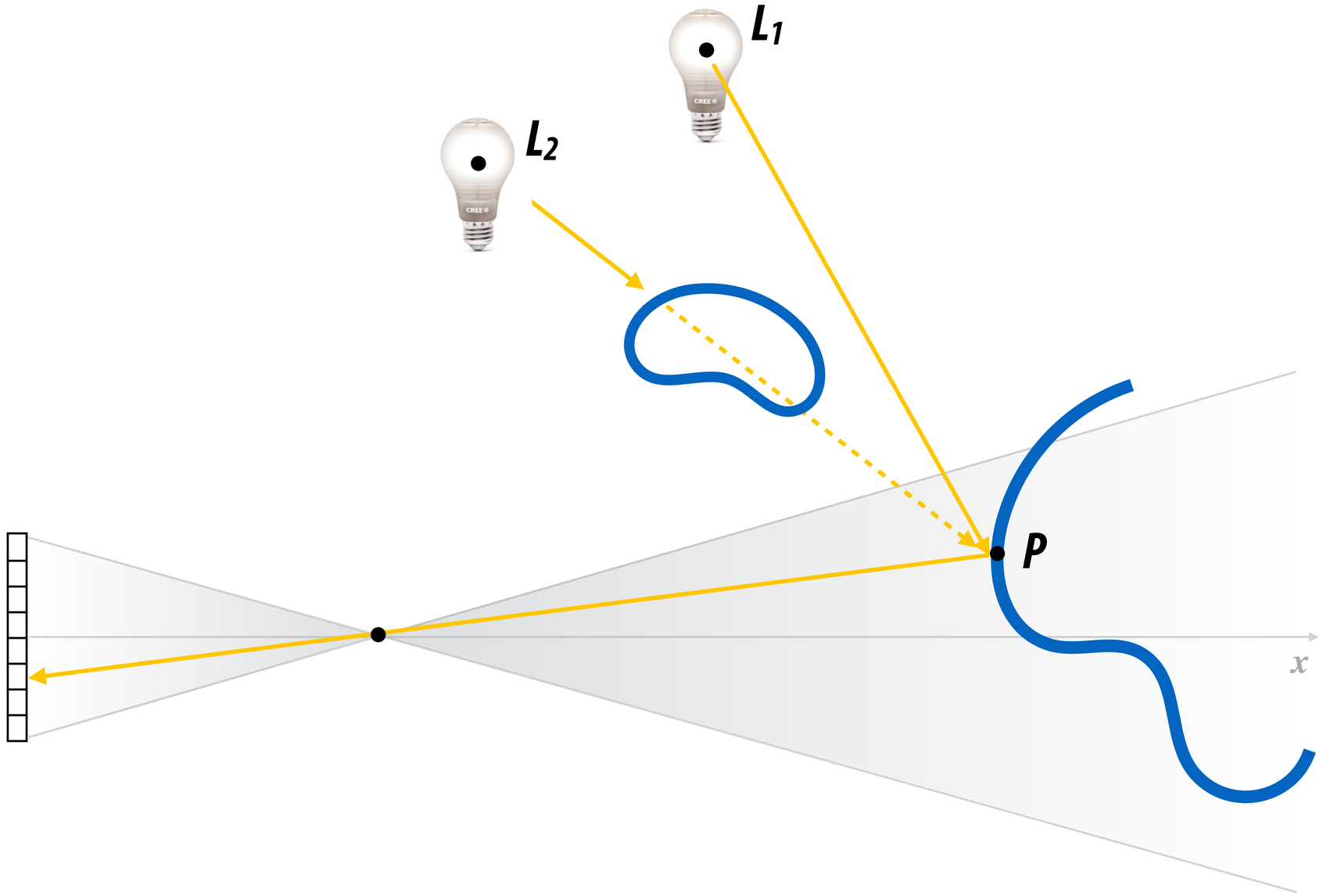
$$L(\mathbf{p}, \omega_o) = \sum_i f(\mathbf{p}, \omega_i, \omega_o) V(\mathbf{p}, \mathbf{L_i}) L_i \cos\theta_i$$

Visibility term:

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

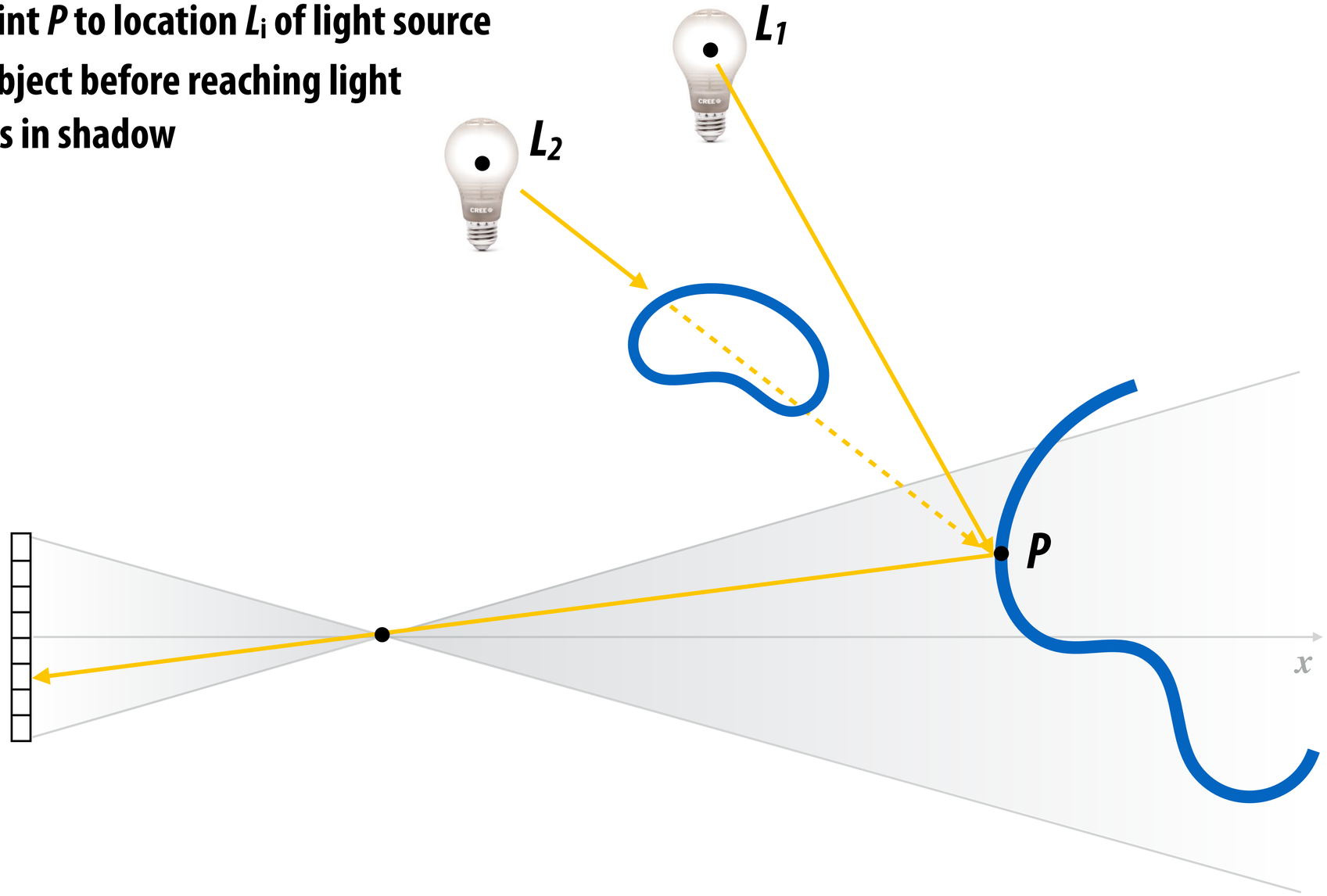


Review: How to compute if a surface is in shadow?



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

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



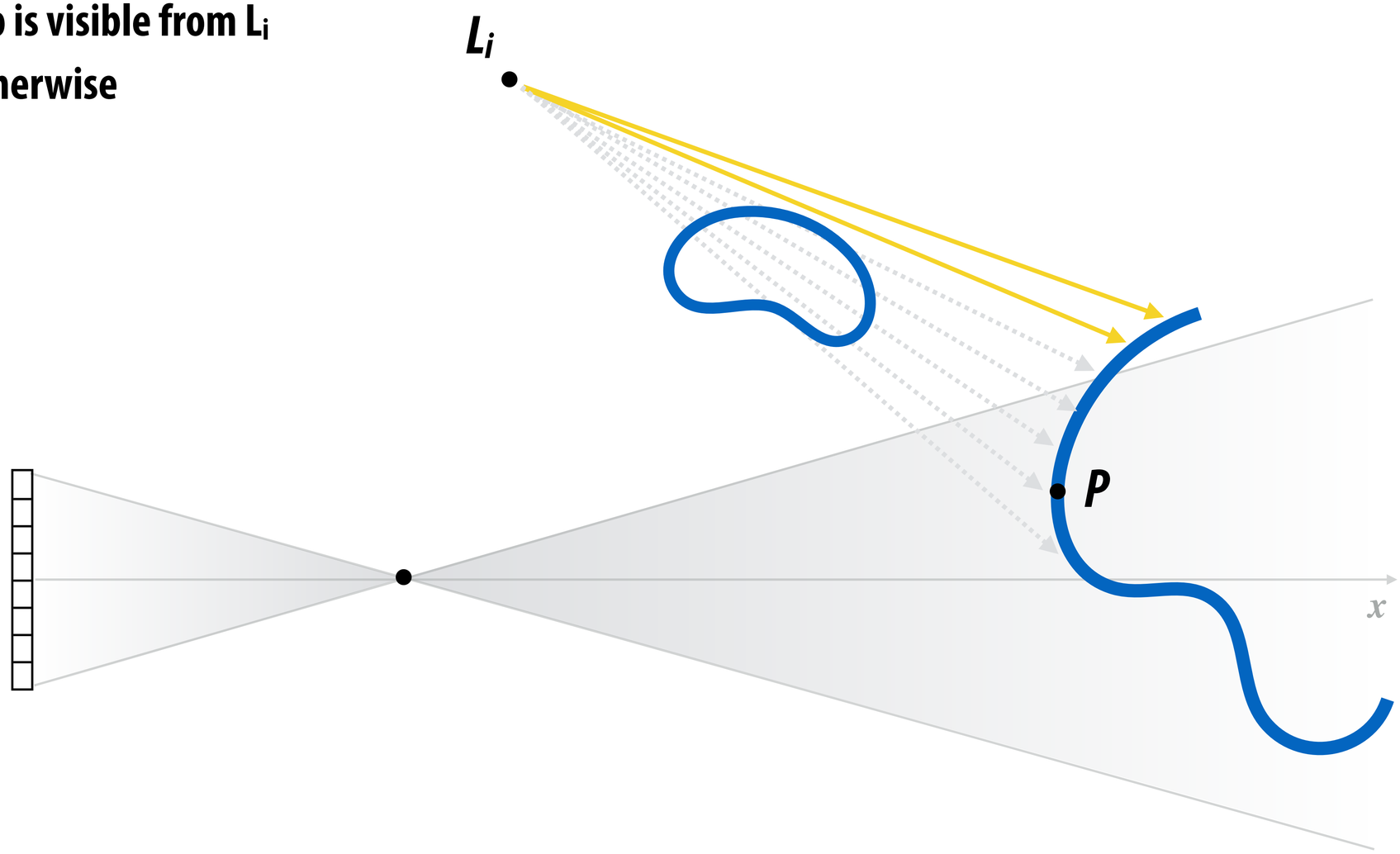
Convince yourself this algorithm produces "hard shadows" like these (what you'd see on a sunny day)

Image credit: Grand Theft Auto V

Or this…

Point lights generate “hard shadows” (Either a point is in shadow or it’s not)

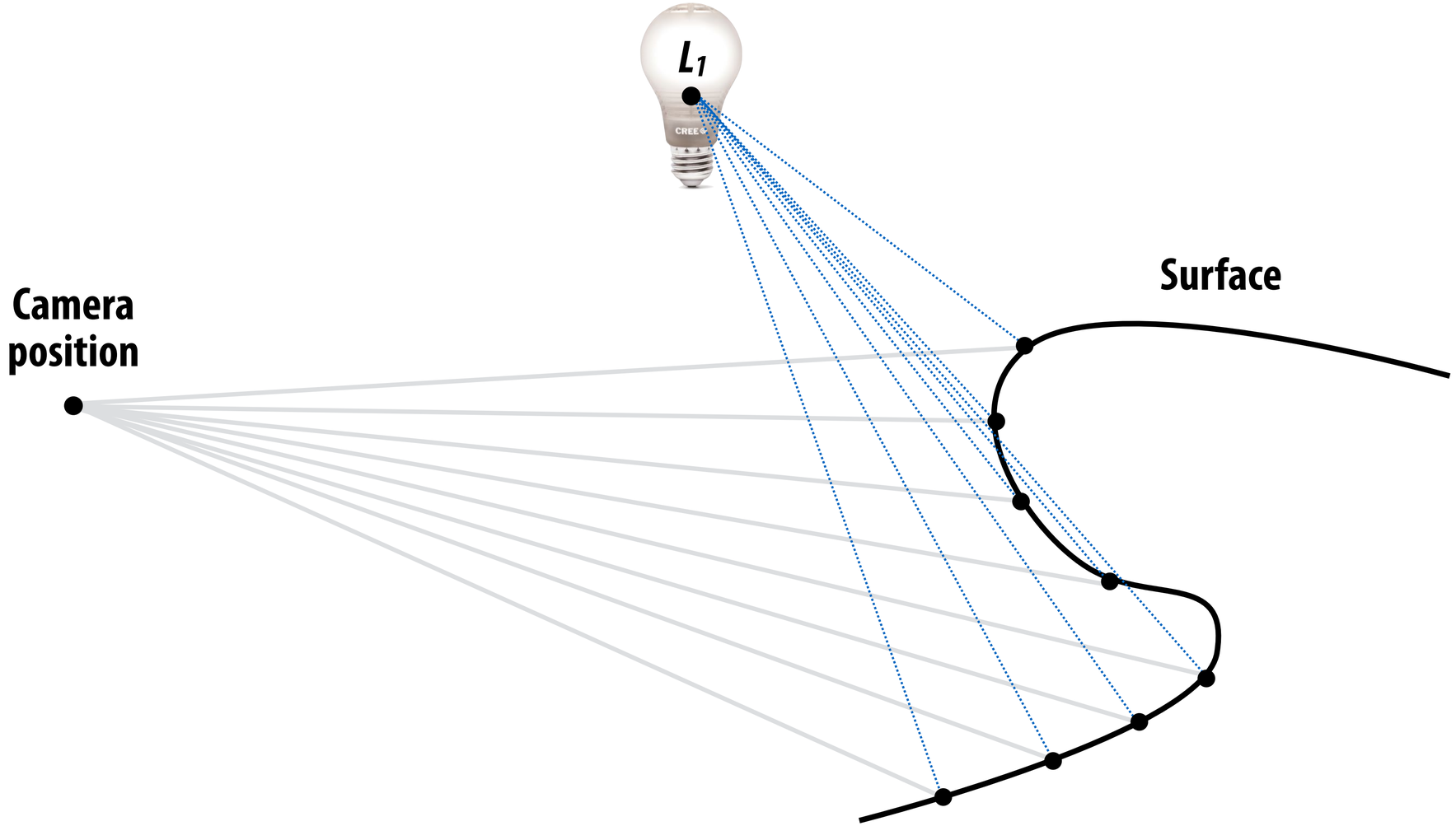
$$V(\mathbf{p}, \mathbf{L_i}) = \begin{cases} \text{1, if p is visible from } L_i \\ \text{0, otherwise} \end{cases}$$



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

We want to shade these points (aka fragments)

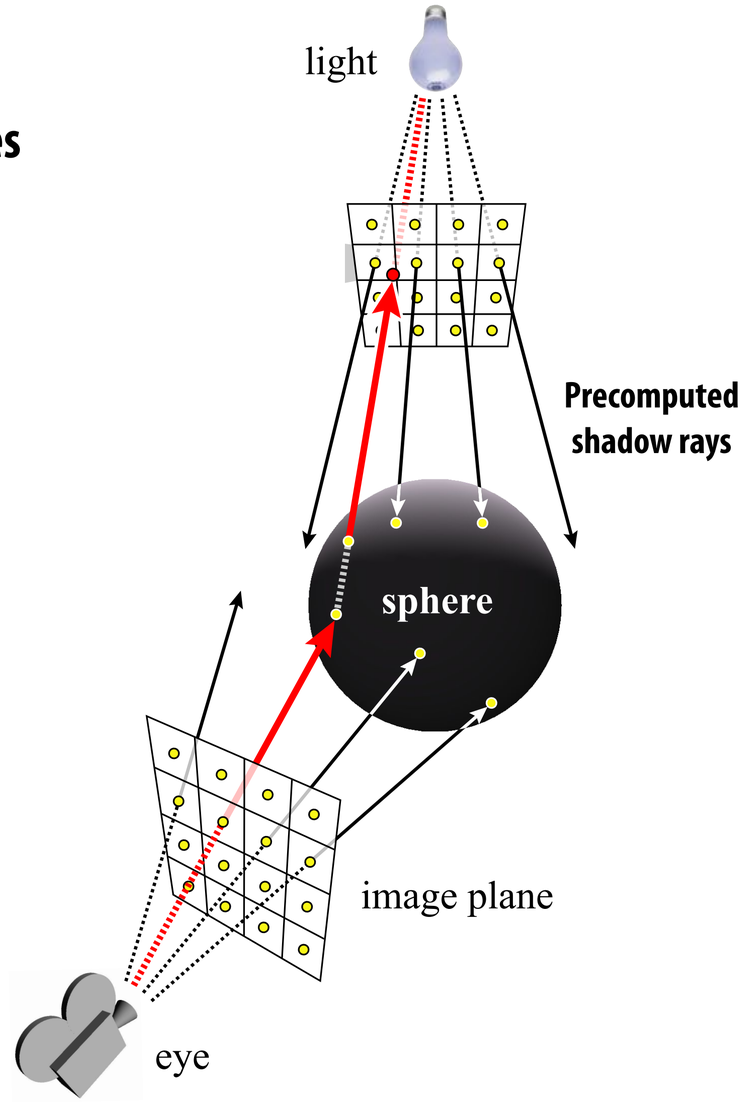
What "shadow rays" do you need to compute shading for this scene?



Shadow mapping (part of assignment 3)

[Williams 78]

1. *Place camera at position of the scene's point light source*
2. Render scene to compute depth to closest object to light along a uniformly spaces 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.
(Stores closest intersection along each shadow ray in a texture)

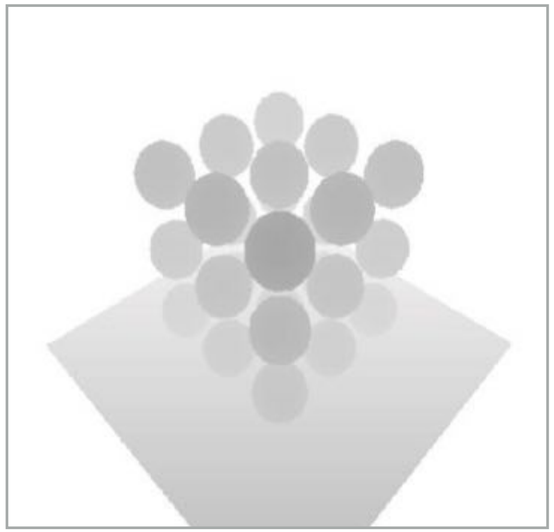
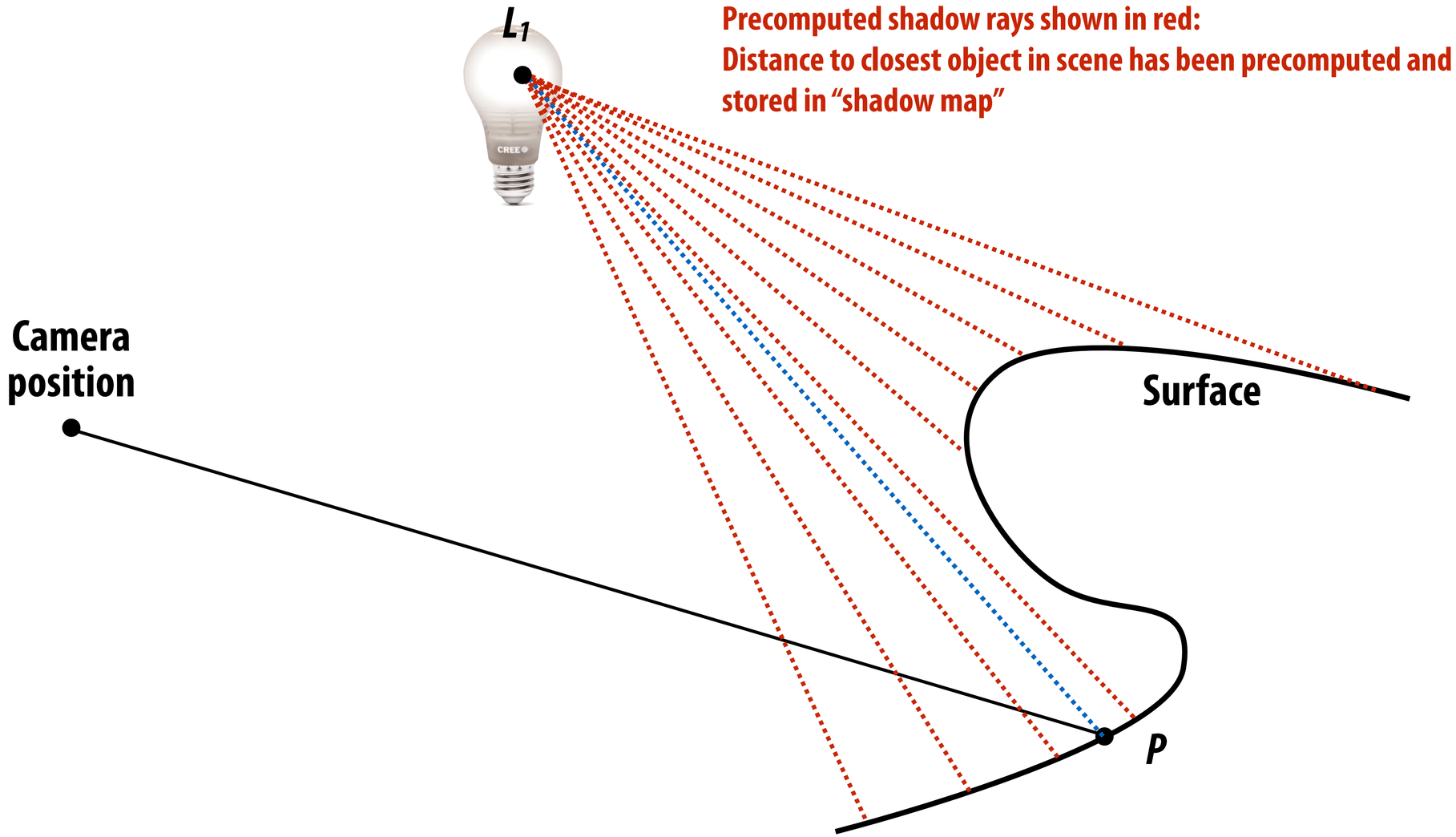


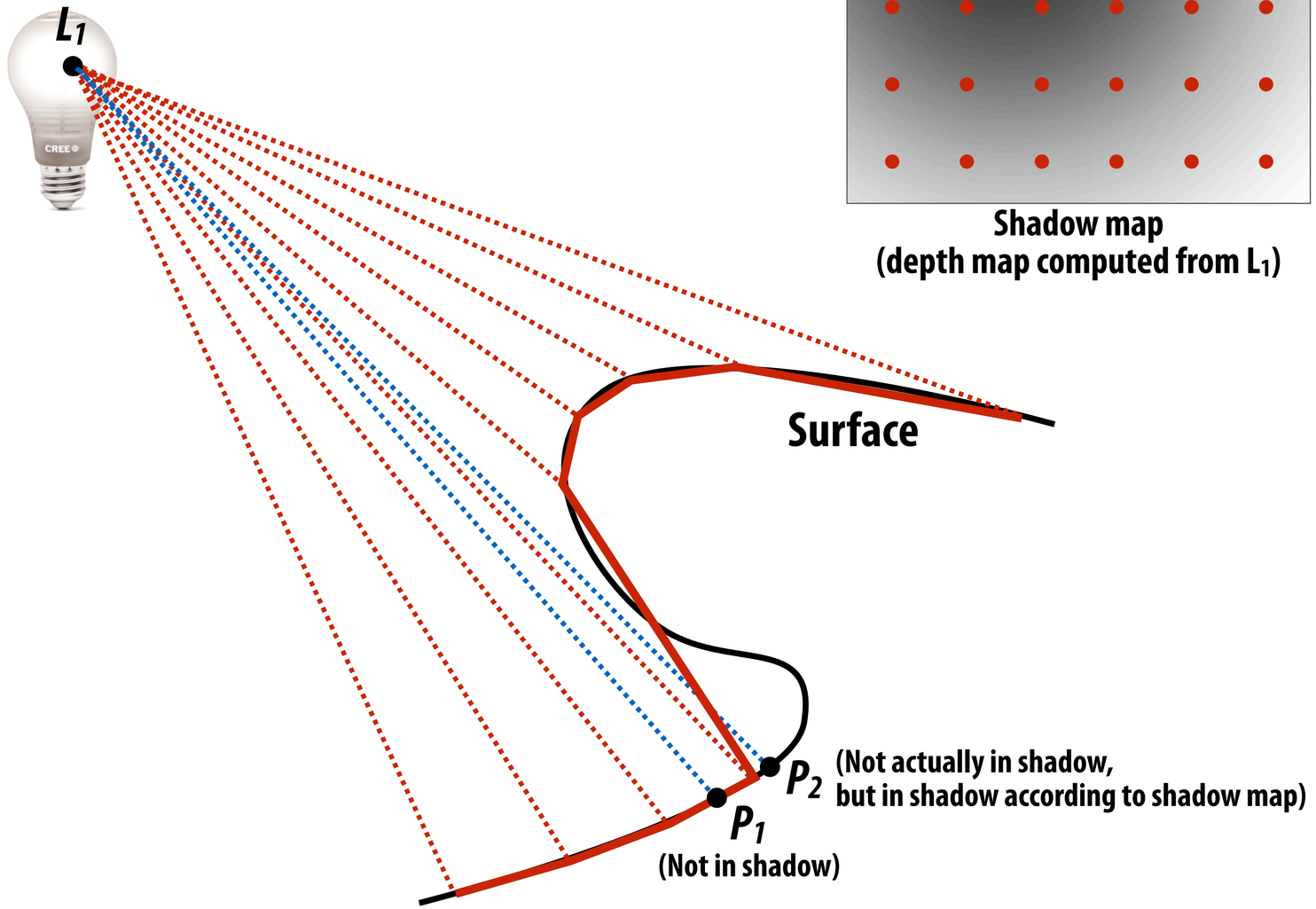
Image credits: Segal et al. 92, NVIDIA

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

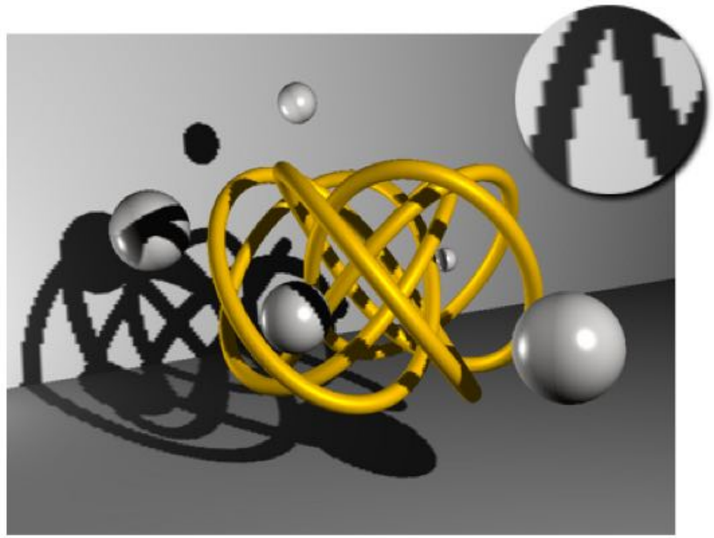


Interpolation error

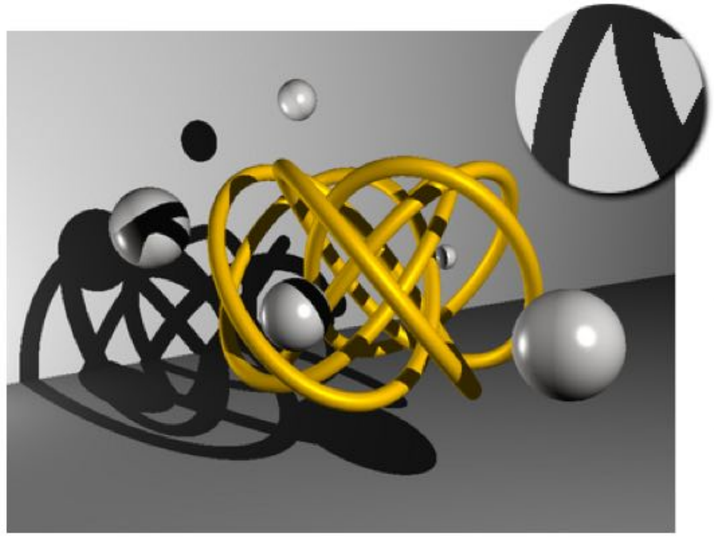
Bilinear interpolation of shadow map values (red line) only approximates distance to closest surface point in all directions from the camera



Shadow aliasing due to shadow map undersampling



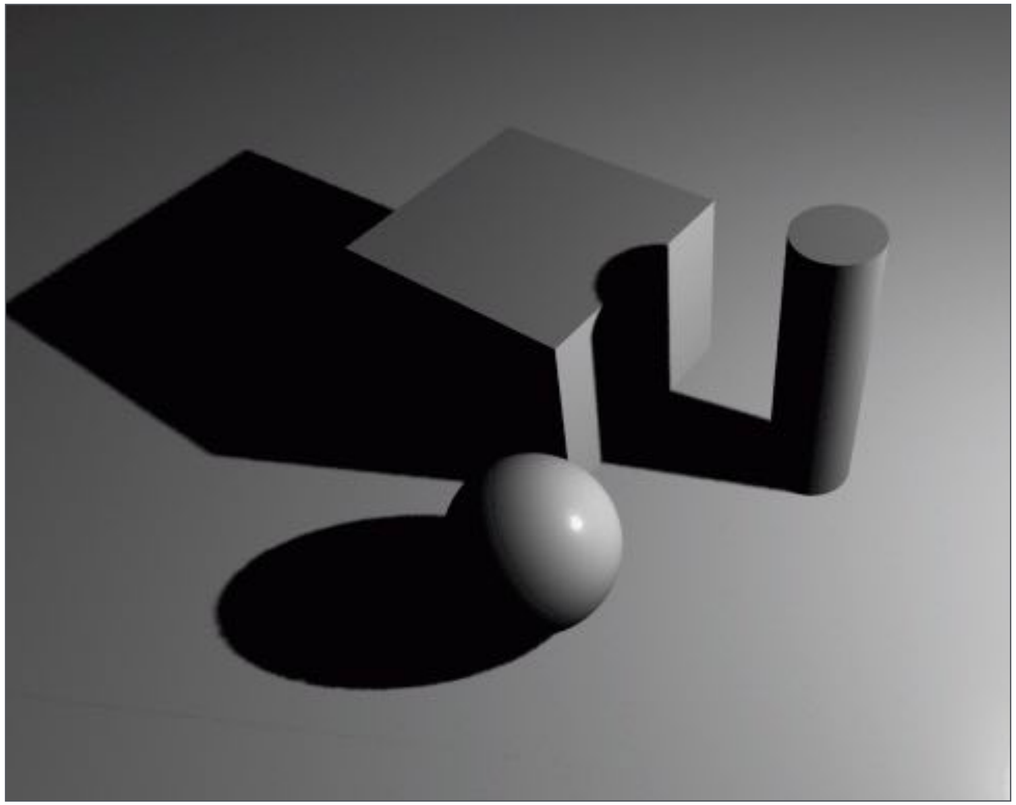
Shadows computed using shadow map



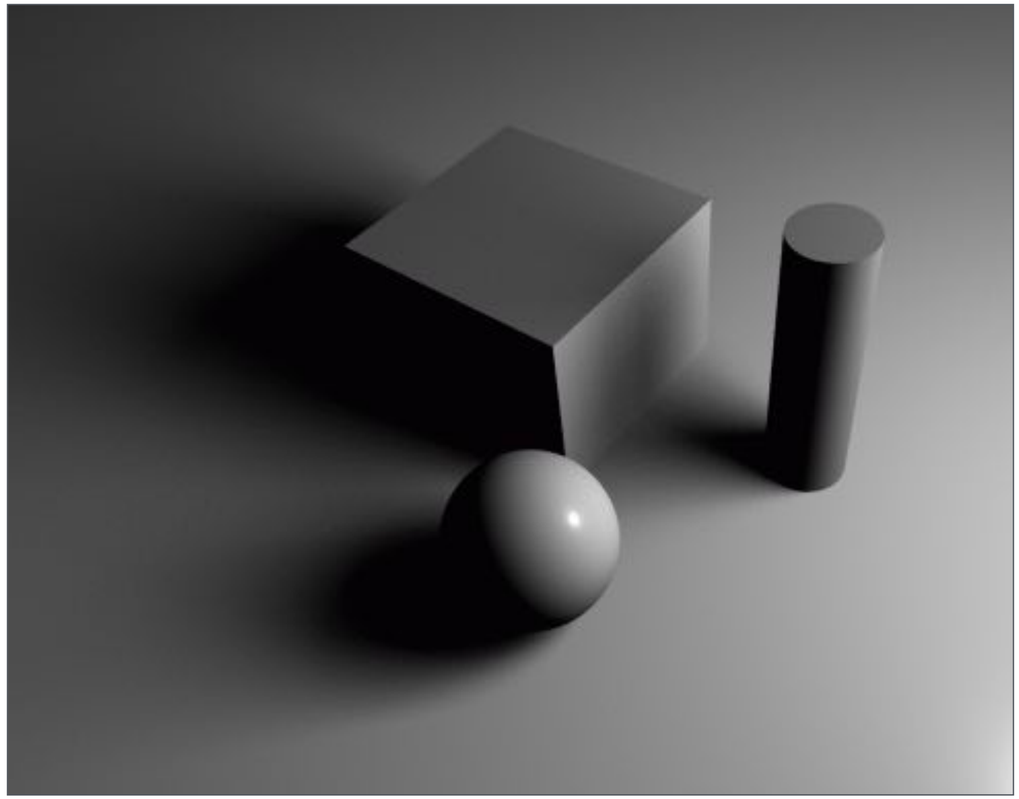
Correct hard shadows
(result from computing visibility along ray between surface point and light directly using ray tracing)

Image credit: Johnson et al. TOG 2005

Soft shadows



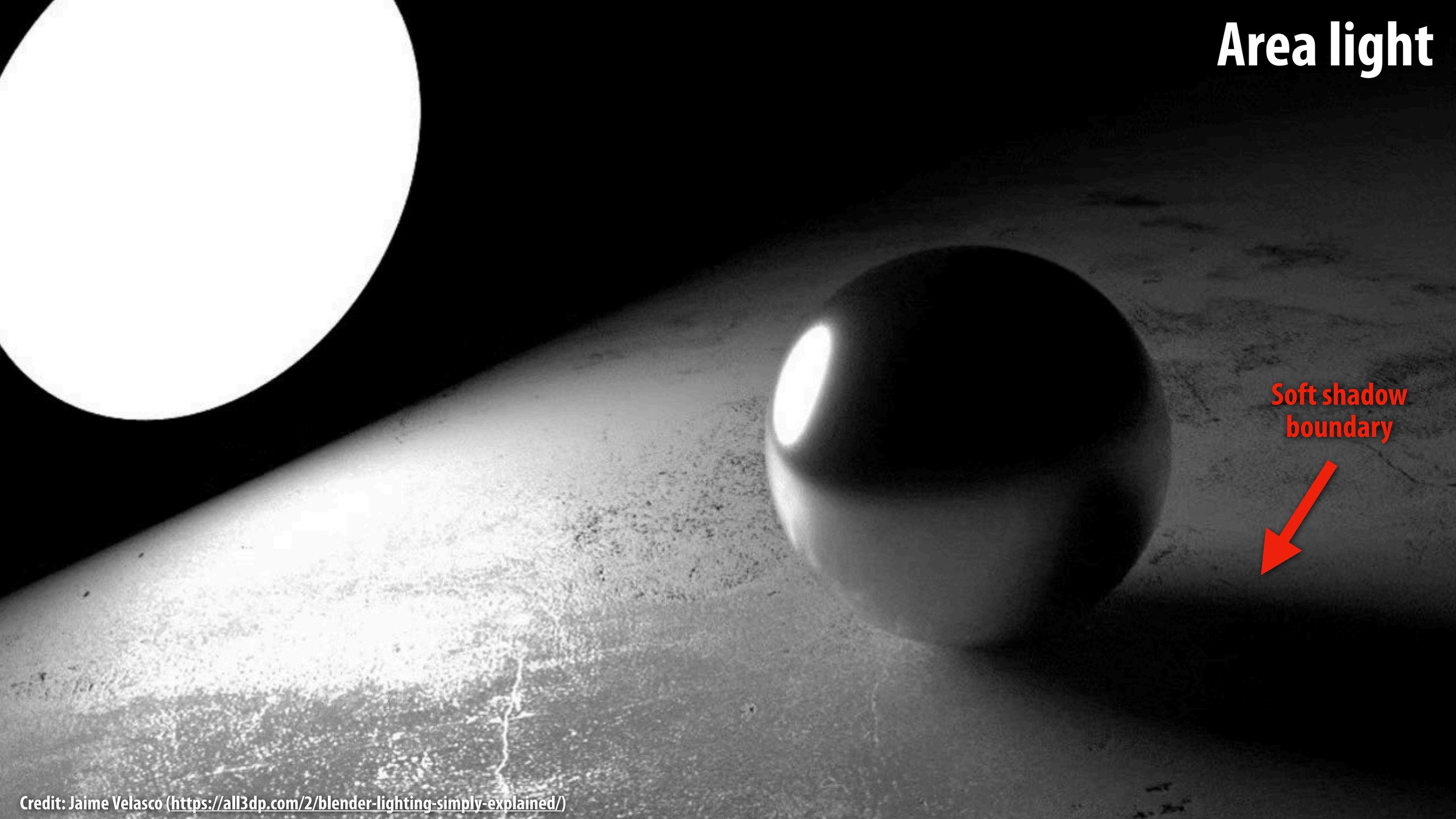
Hard shadows
(created by point light source)



Soft shadows
(created by ???)

Image credit: Pixar

Area light



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

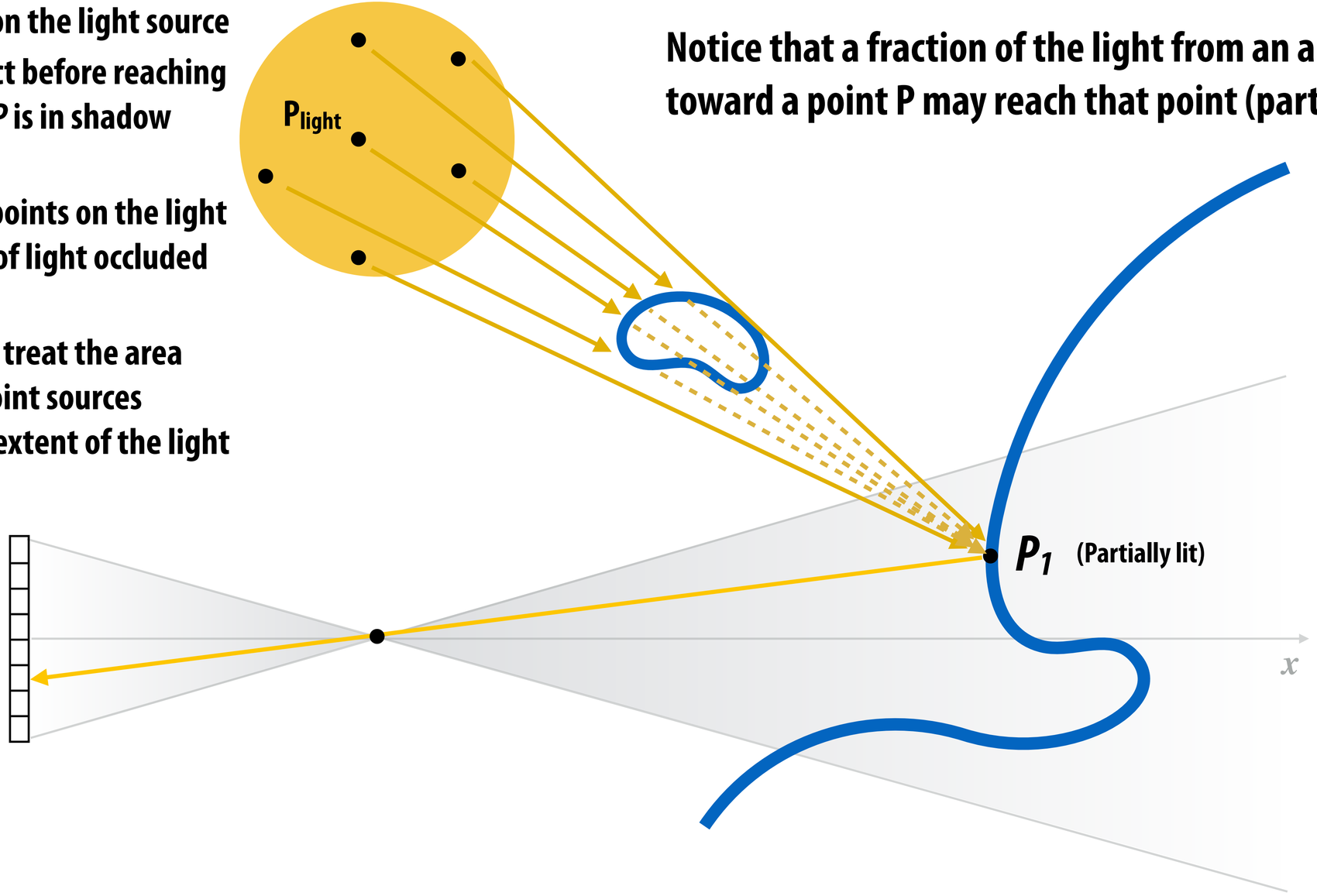
Area light

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

Shadow cast by an area light (via ray tracing…)

- Choose a point P_{light} on the light source
- If ray hits scene object before reaching light source… then *P* is in shadow from P_{light}
- Repeat for multiple points on the light to estimate fraction of light occluded from P.
- In other words… we treat the area light as a bunch of point sources distributed over the extent of the light source

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

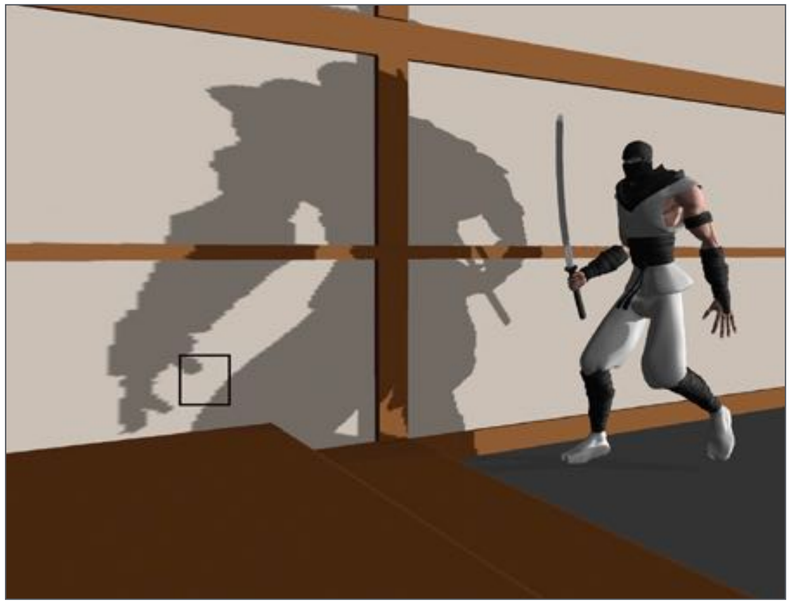


Percentage closer filtering (PCF) — hack!

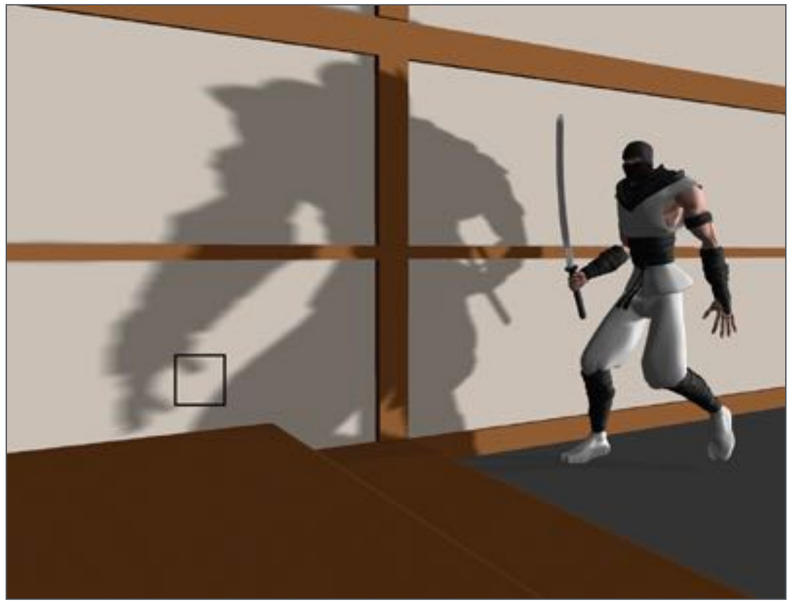
- Instead of sample shadow map once, perform multiple lookups around desired texture coordinate
- Tabulate fraction of lookups that are in shadow, modulate light intensity accordingly

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



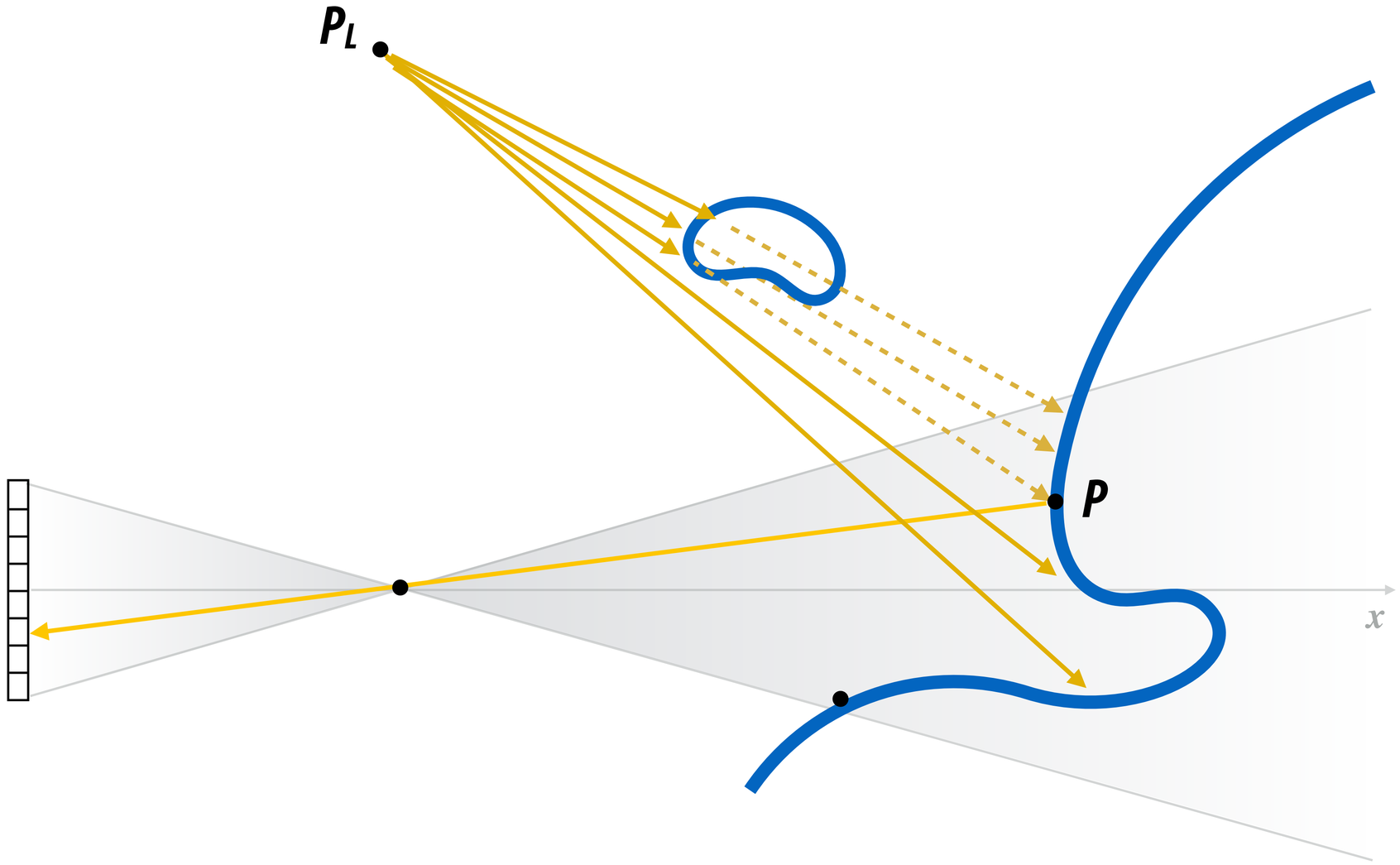
Hard shadows
(one lookup per fragment)



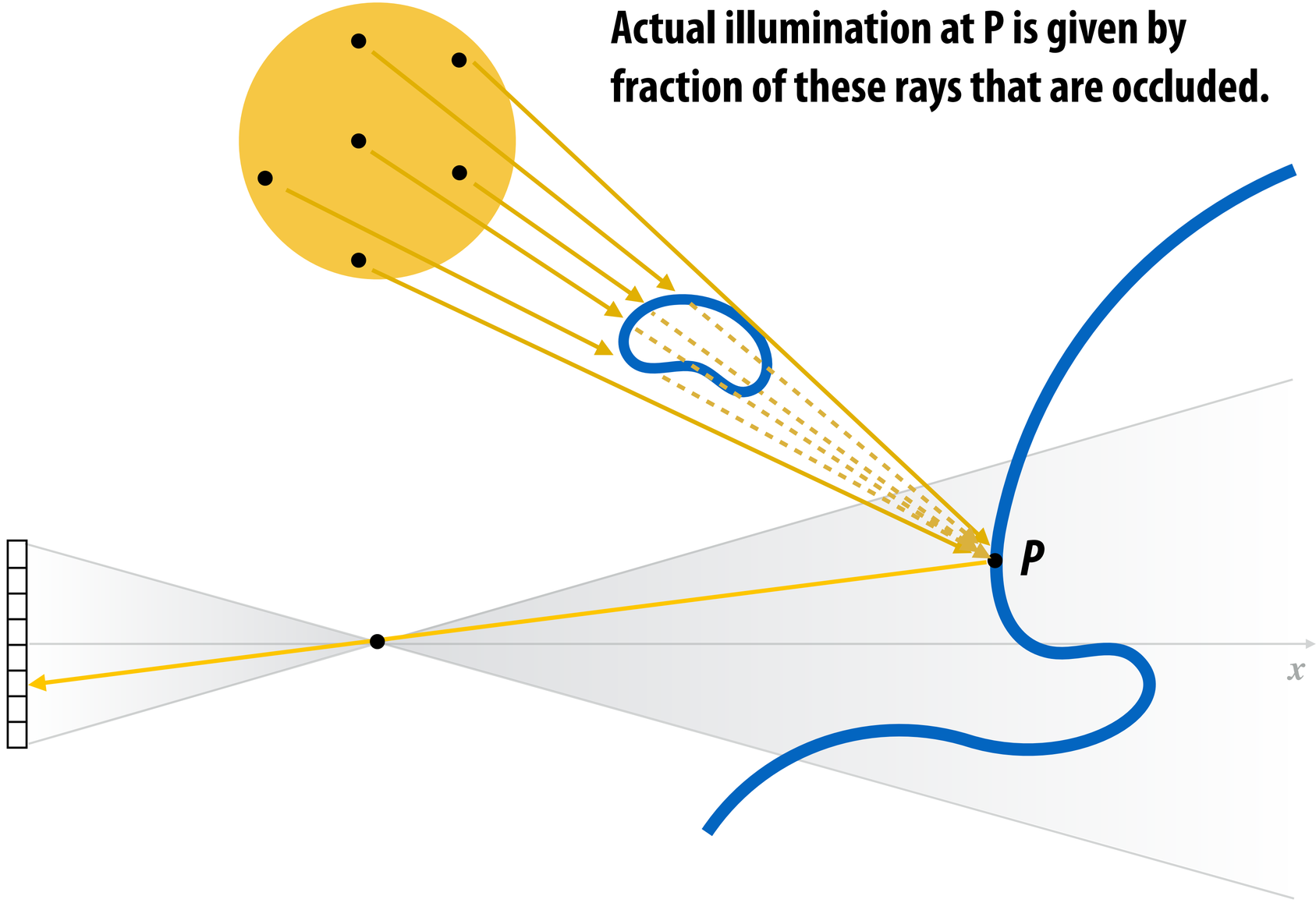
PCF shadows
(16 lookups per fragment)

What PCF computes

The fraction of these rays that are shorter than $|P\text{-}P_L|$



Shadow cast by an area light



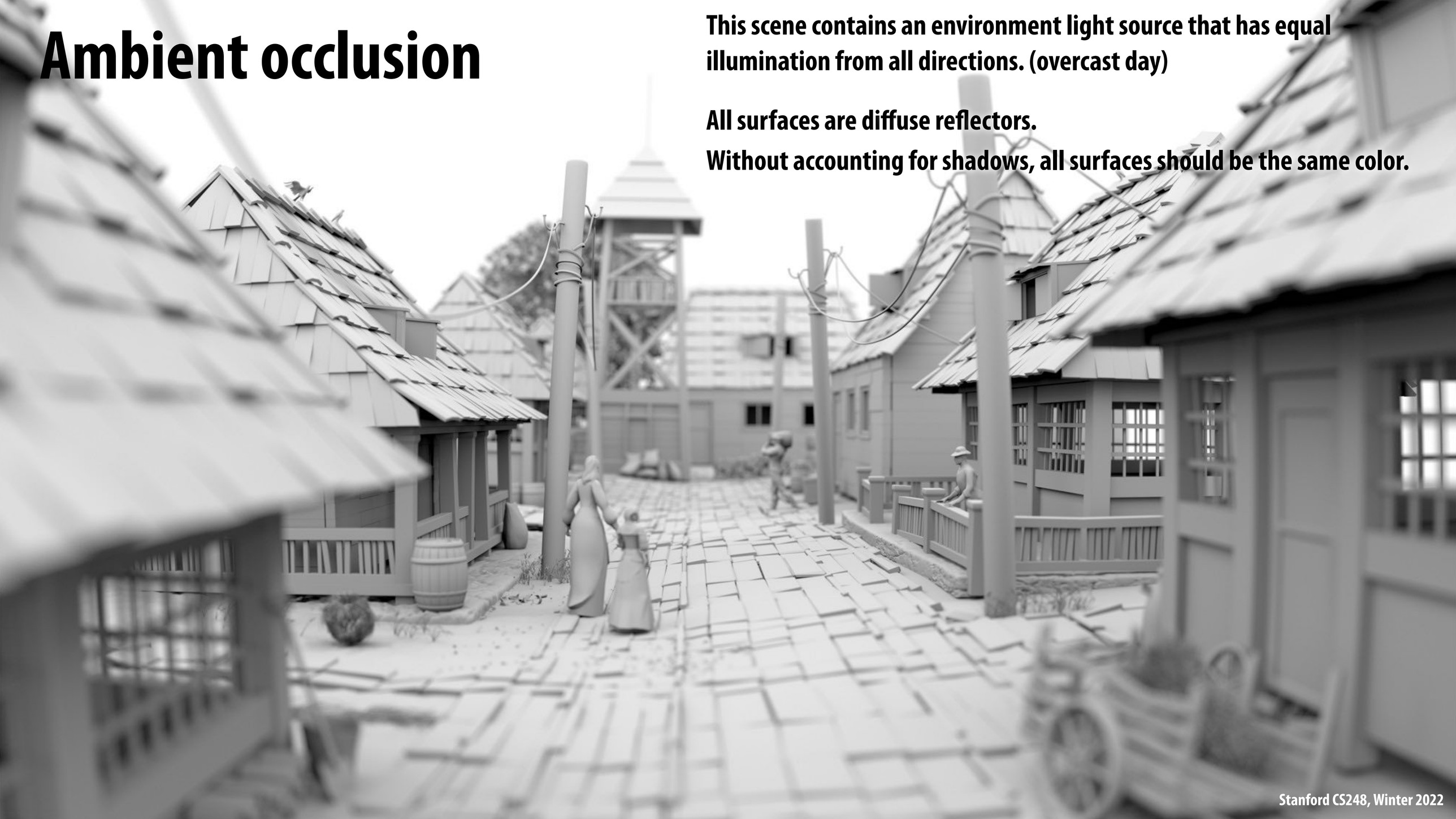
Q. Why isn't the surface in shadow completely black?

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. (overcast day)

All surfaces are diffuse reflectors.

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

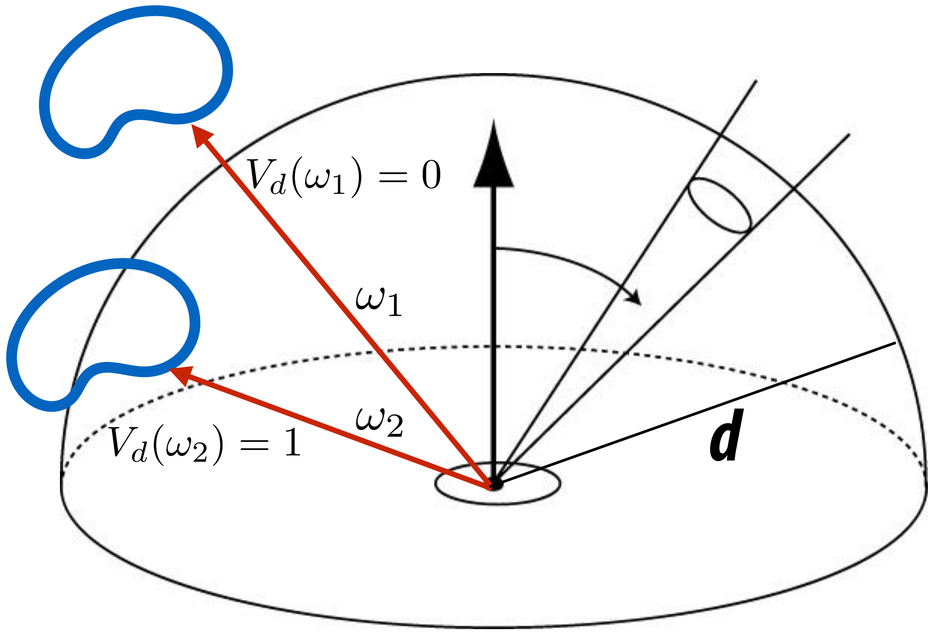
Hack: ambient occlusion

Idea:

Precompute "fraction of hemisphere" that is occluded within distance *d* from a point.

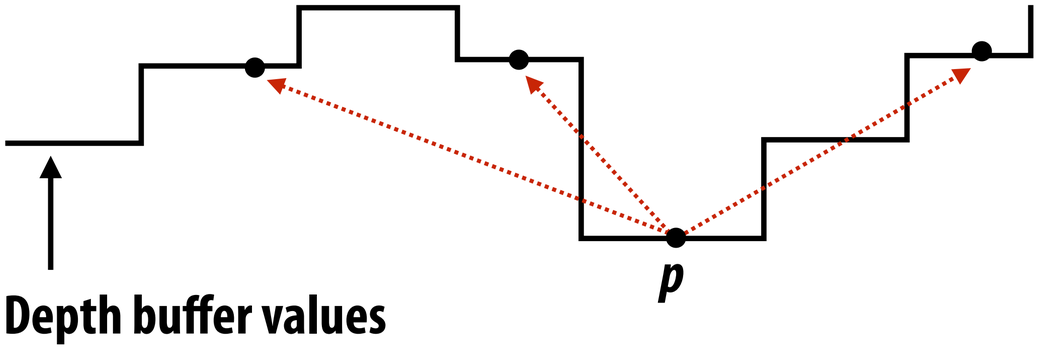
Store this fraction in a texture map

When shading, attenuate environment lighting by this fraction



"Screen-space" ambient occlusion in games

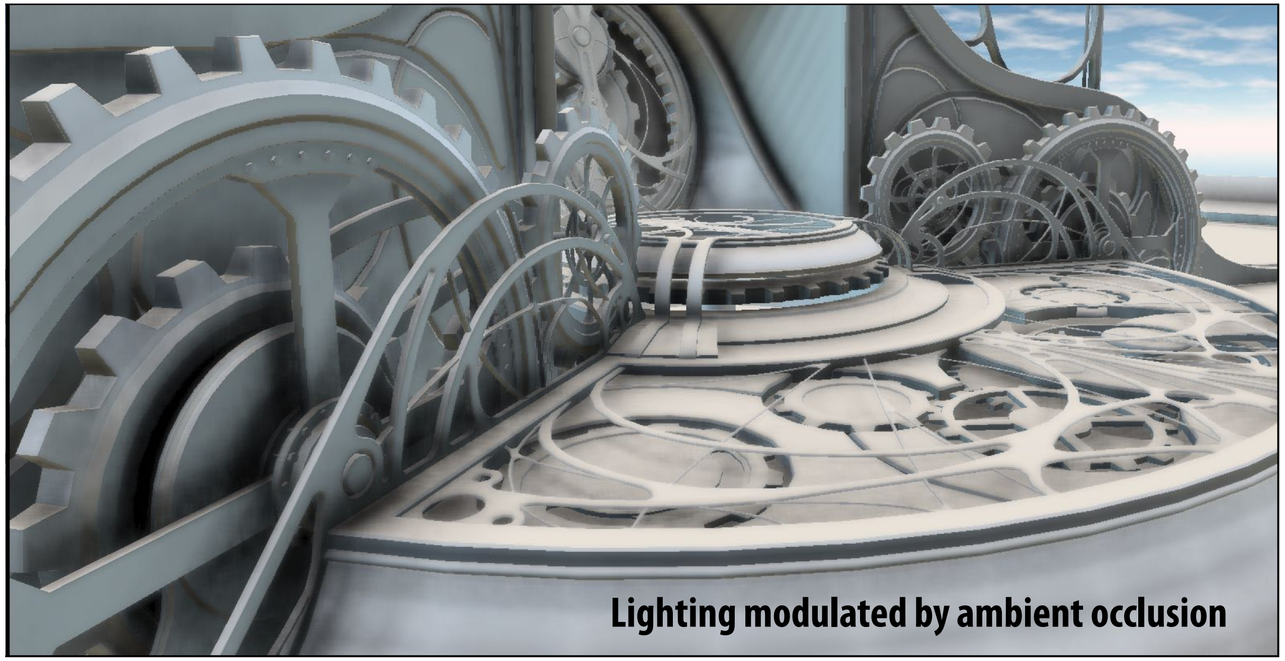
1. Render scene to depth buffer
2. For each pixel *p* ("ray trace" the depth buffer to estimate local occlusion of hemisphere - use a few samples per pixel)
3. Blur the the occlusion map to reduce noise
4. When shading pixels, darken direct environment lighting by occlusion amount



without ambient occlusion

with ambient occlusion

Ambient occlusion



Reflections

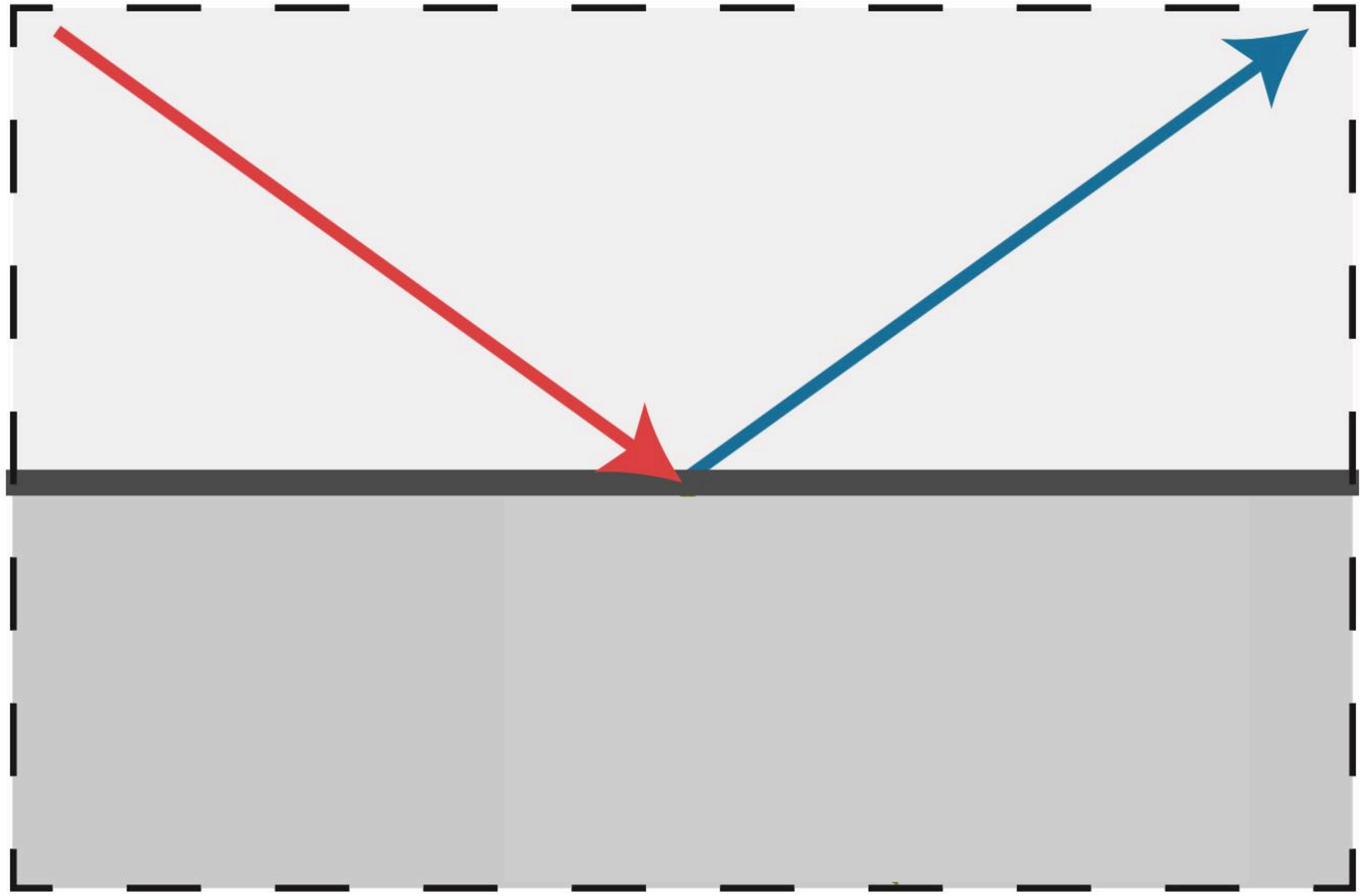
What is wrong with this picture?

Reflections

Image credit: NVIDIA

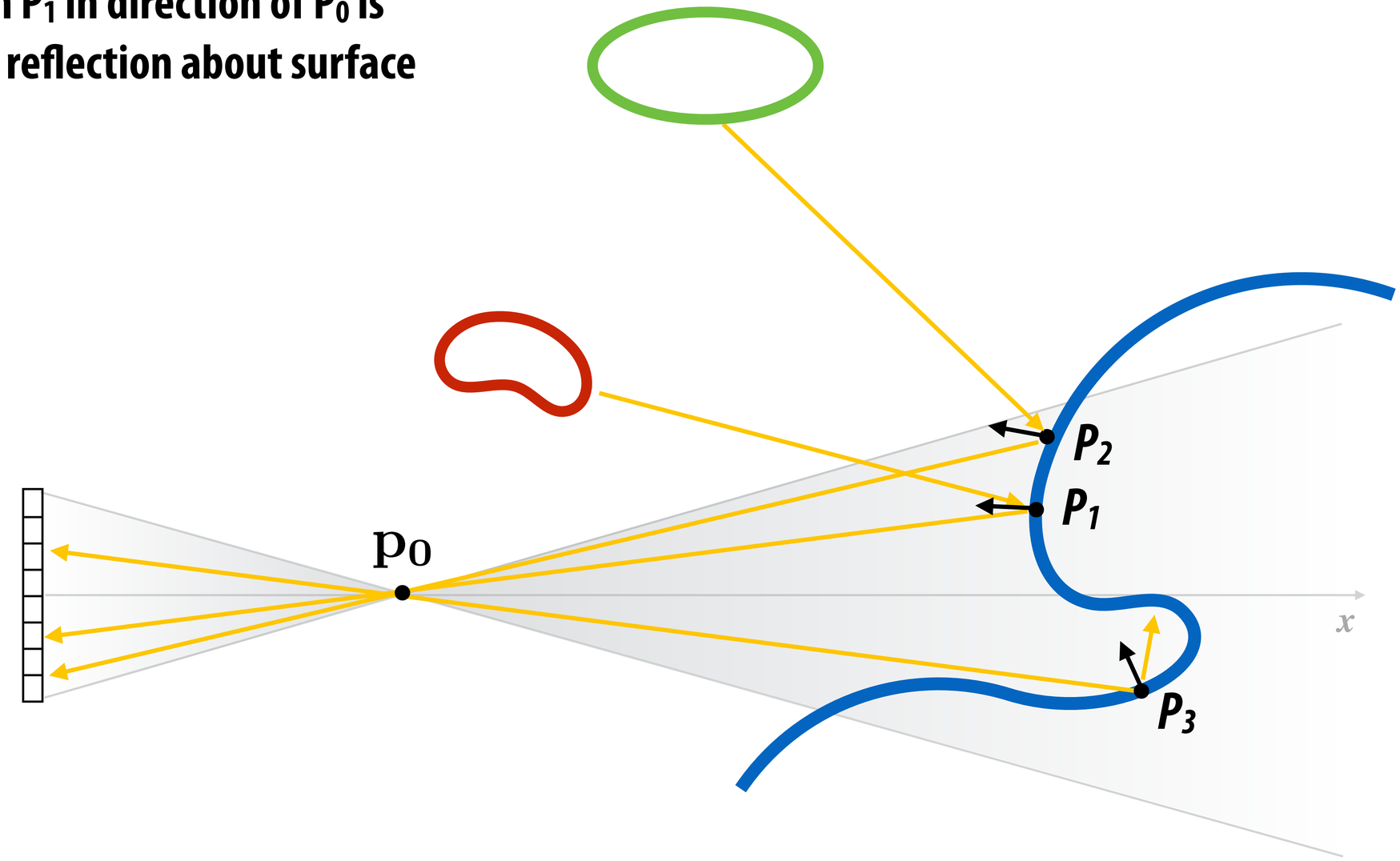
Reflections

Recall: perfect mirror material



Recall: perfect mirror reflection

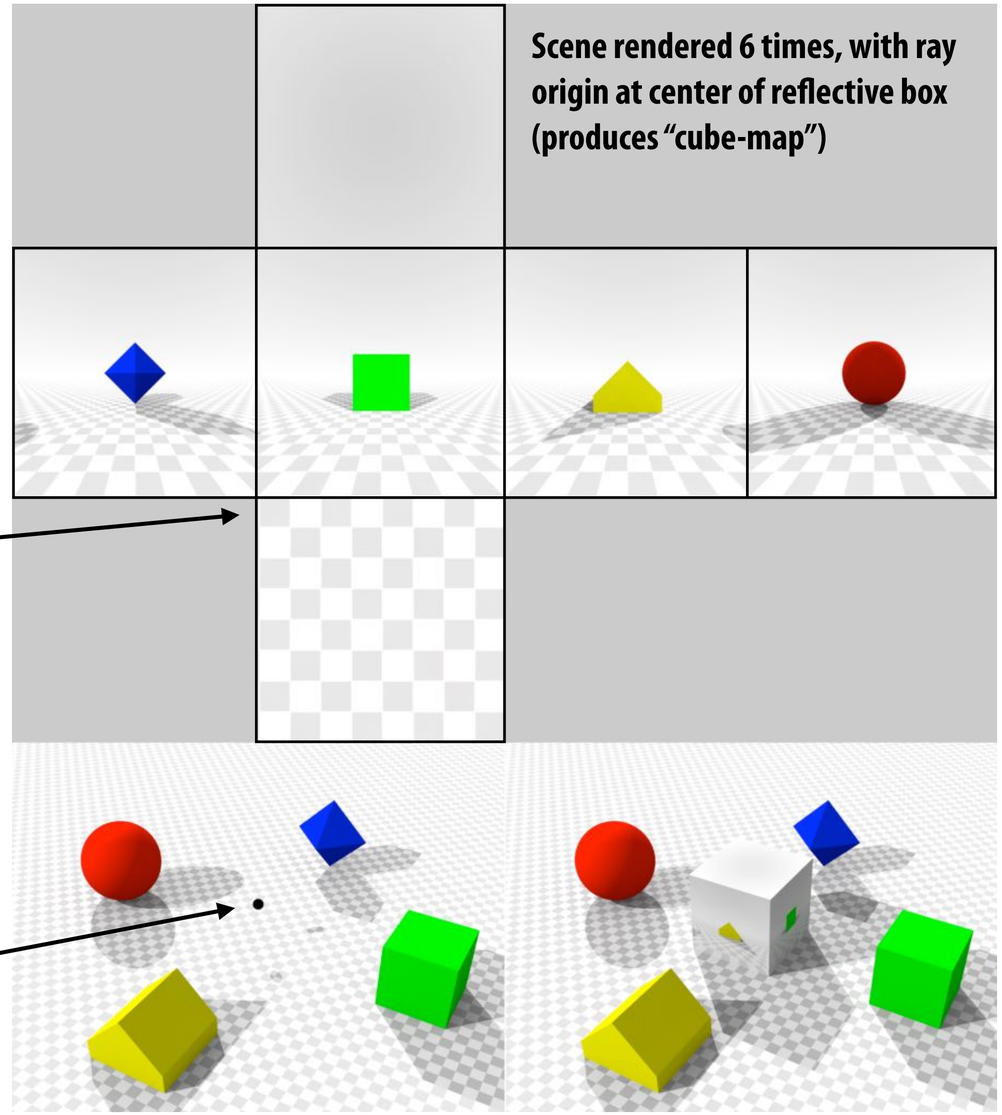
Light reflected from P_1 in direction of P_0 is incident on P_1 from reflection about surface at P_1.



Rasterization: "camera" position can be reflection point

Environment mapping:
place ray origin at reflective object

Yields 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

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

Environment map vs. ray traced reflections

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

Image credit: Control

Environment map vs. ray traced reflections

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

Image credit: Control

Interreflections

Diffuse interreflections

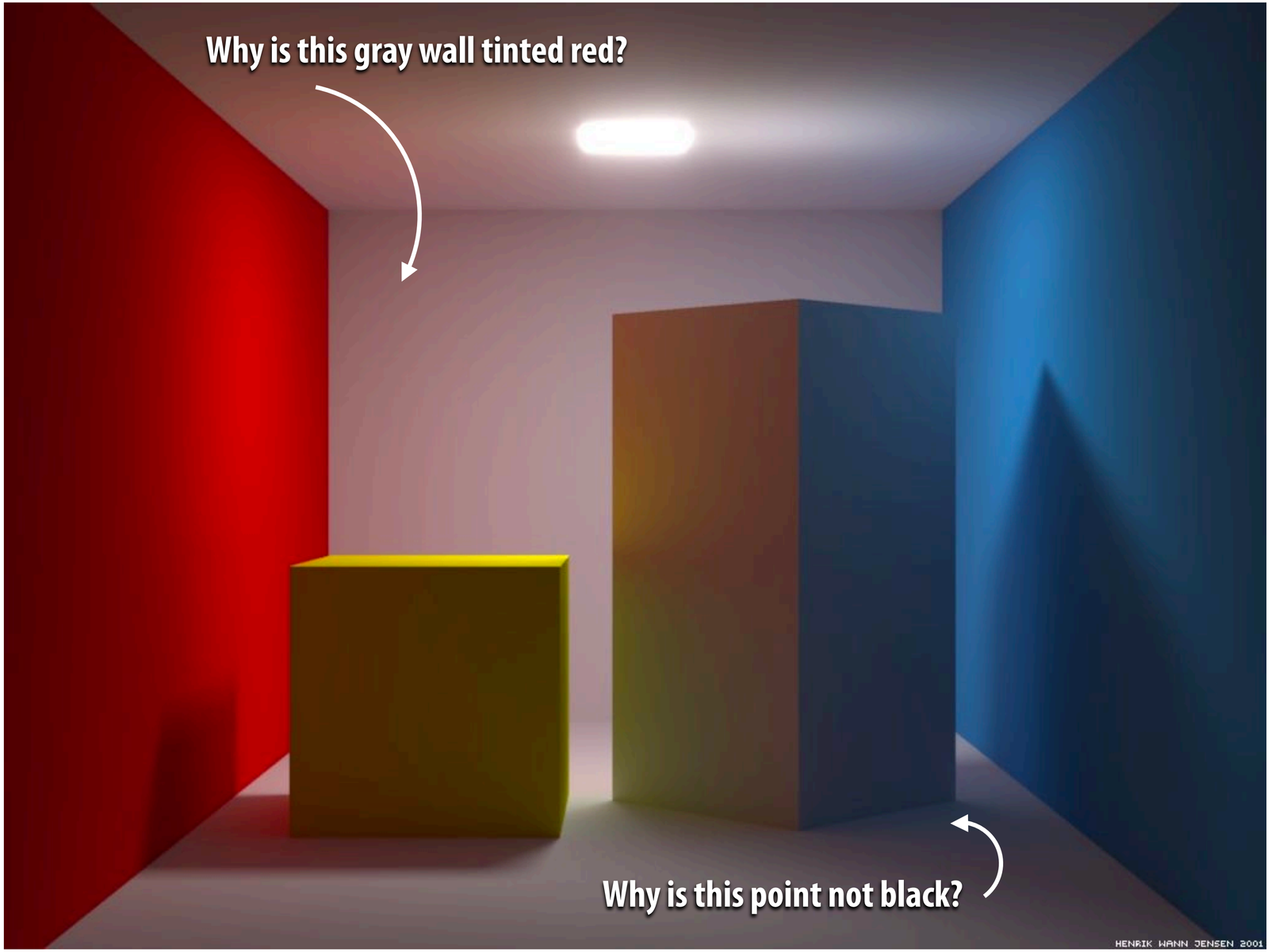
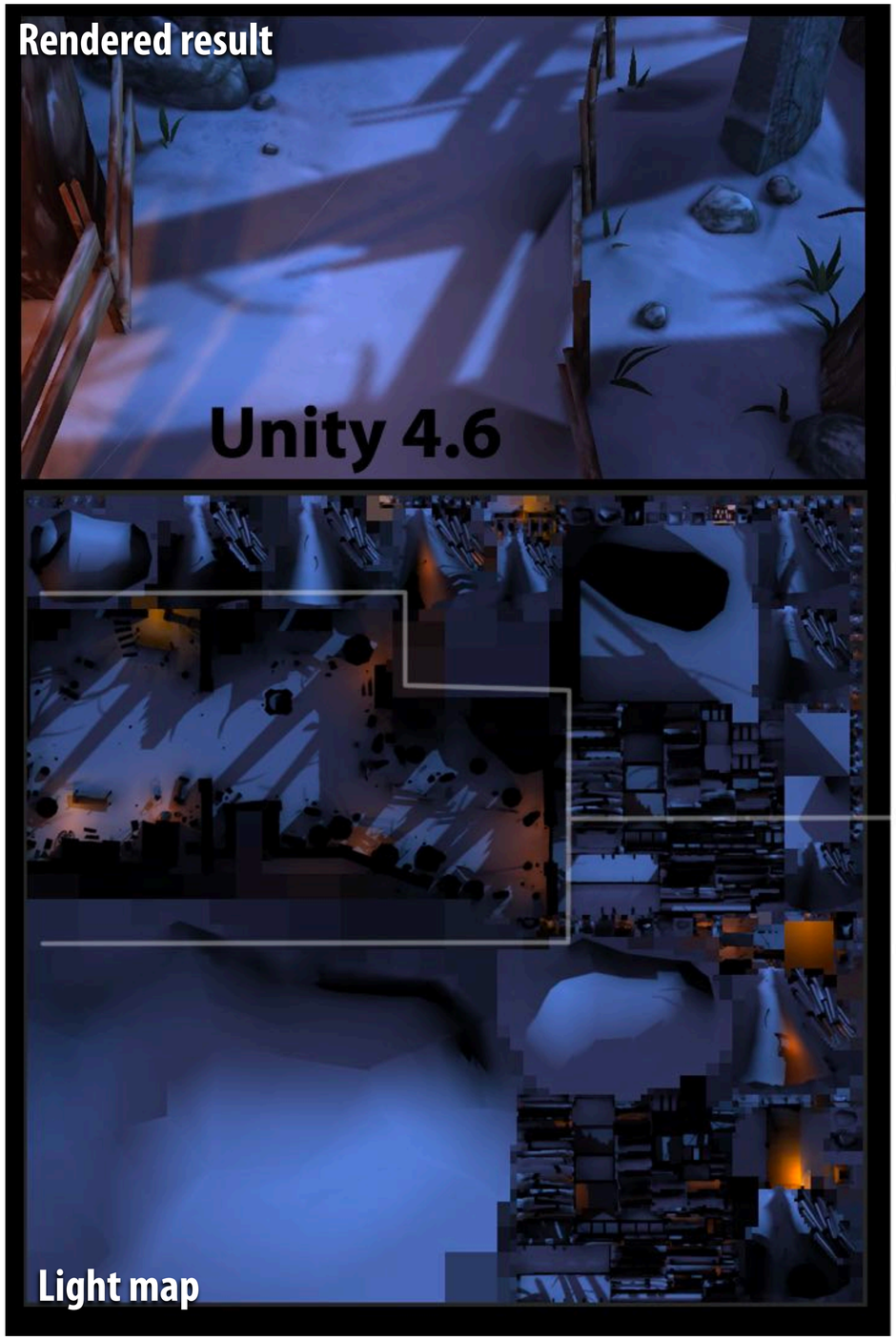


Image credit: Henrik Wann Jensen

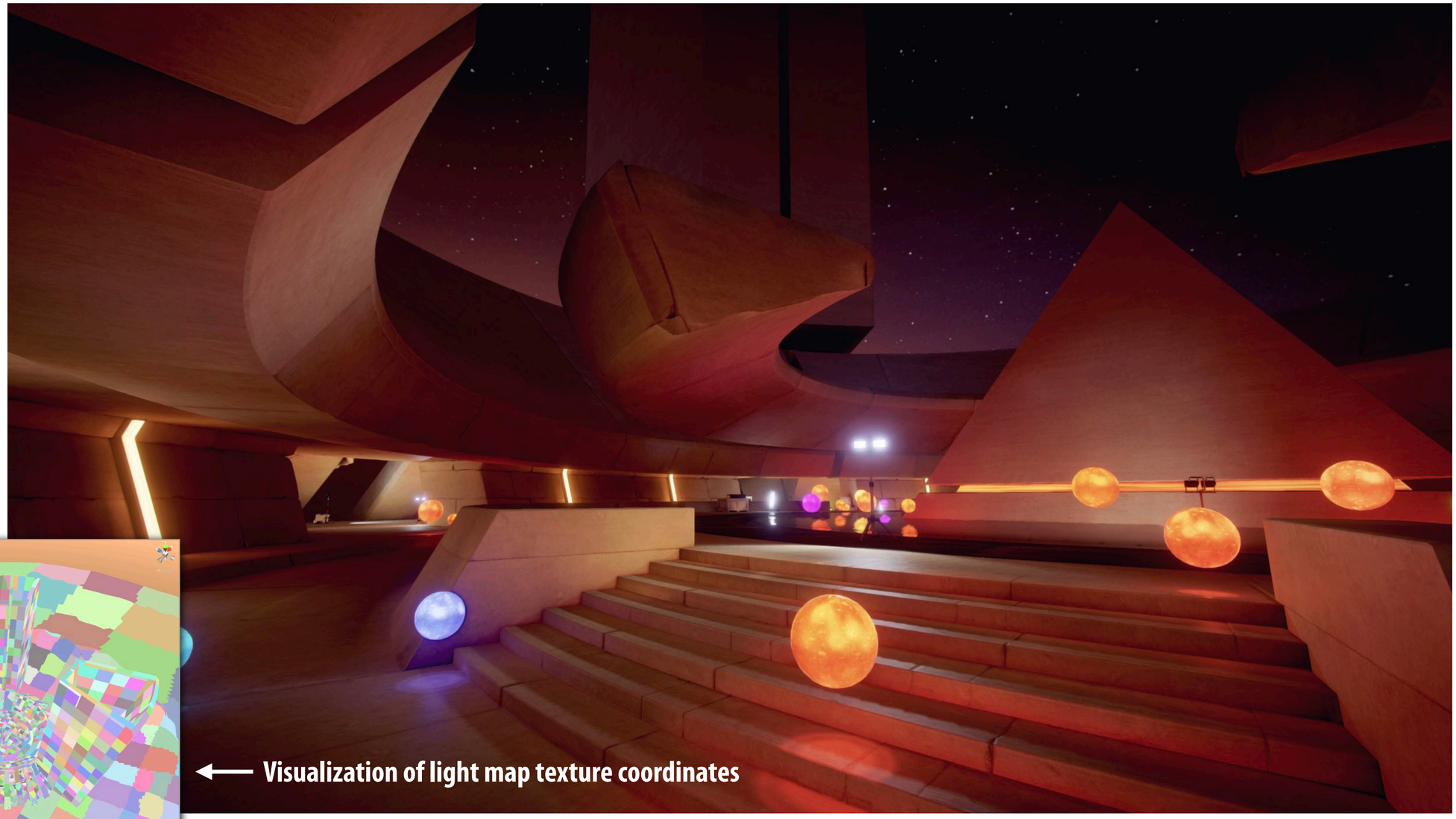
Precomputed lighting

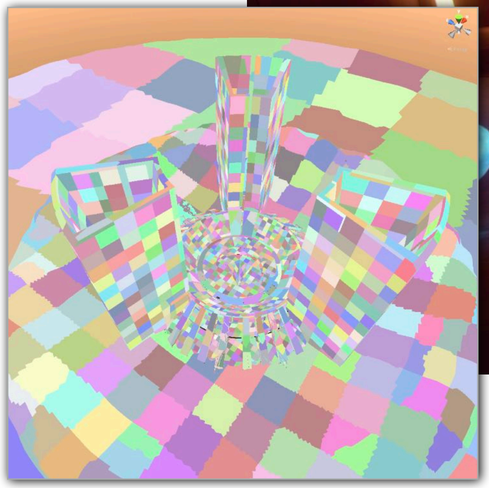
- Precompute lighting for a scene offline (possible for static lights)
  - Offline computations can perform advanced shadowing, inter reflection computations

- "Bake" results of lighting into texture map



Precomputed lighting in Unity Engine



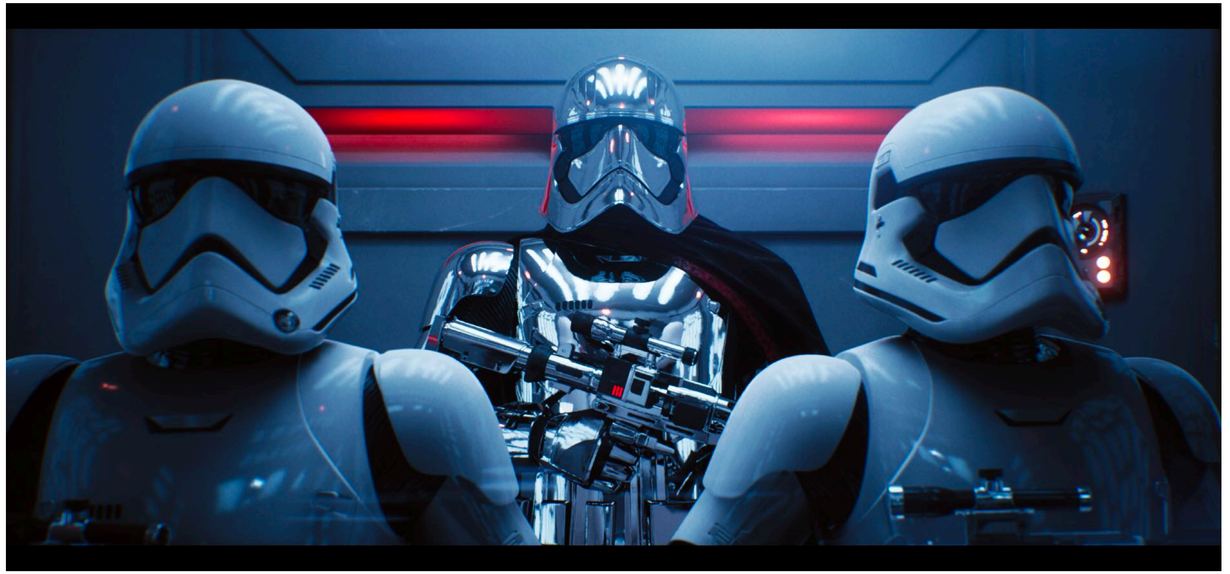


← Visualization of light map texture coordinates

Image credit: Unity / Alex Lovett

Growing interest in 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!")
- Traditionally, tracing rays to solve these problems was too costly for real-time use
  - That may be changing soon…

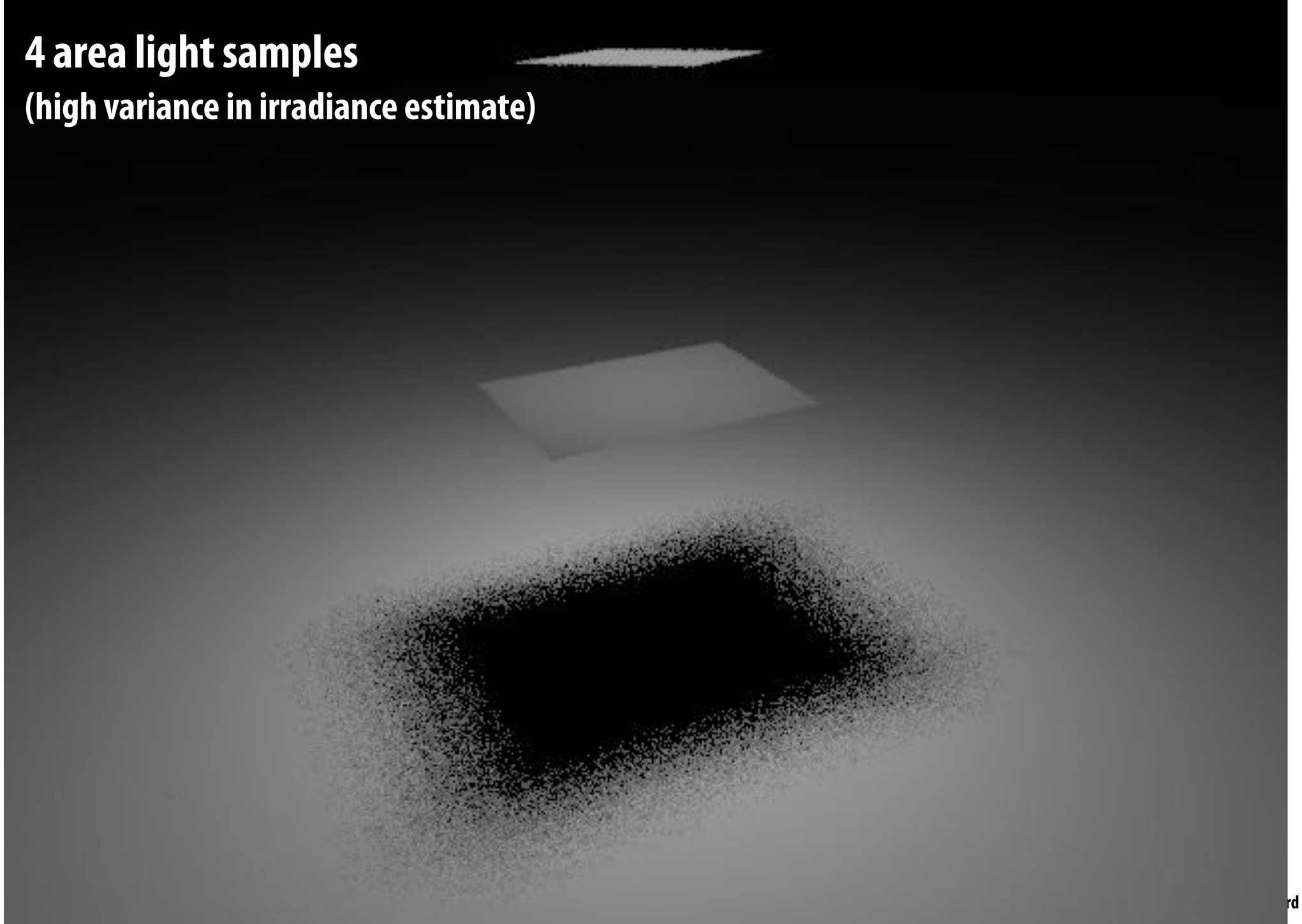


← This image was ray traced in real-time on a (very high end) GPU

Learn more in CS348B!

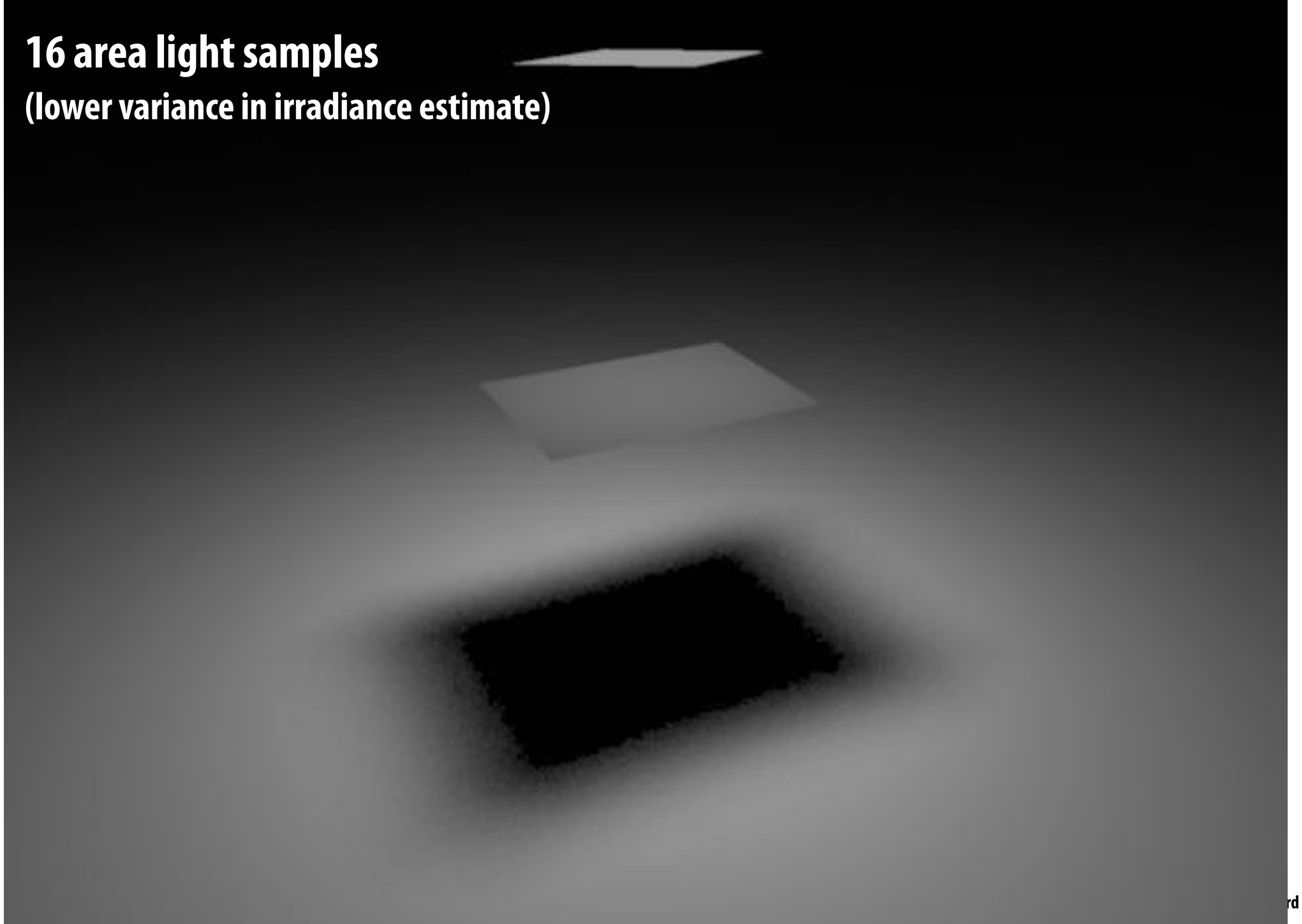
Why ray tracing is expensive

4 area light samples
(high variance in irradiance estimate)

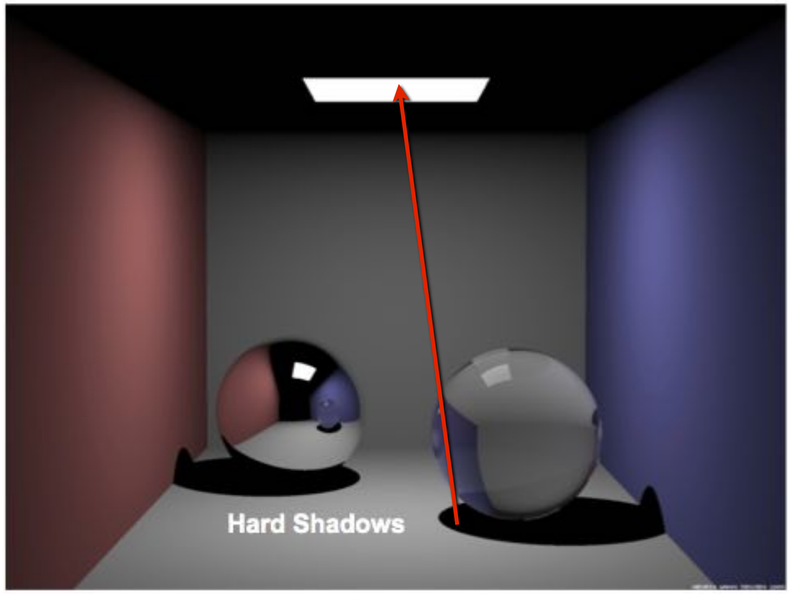


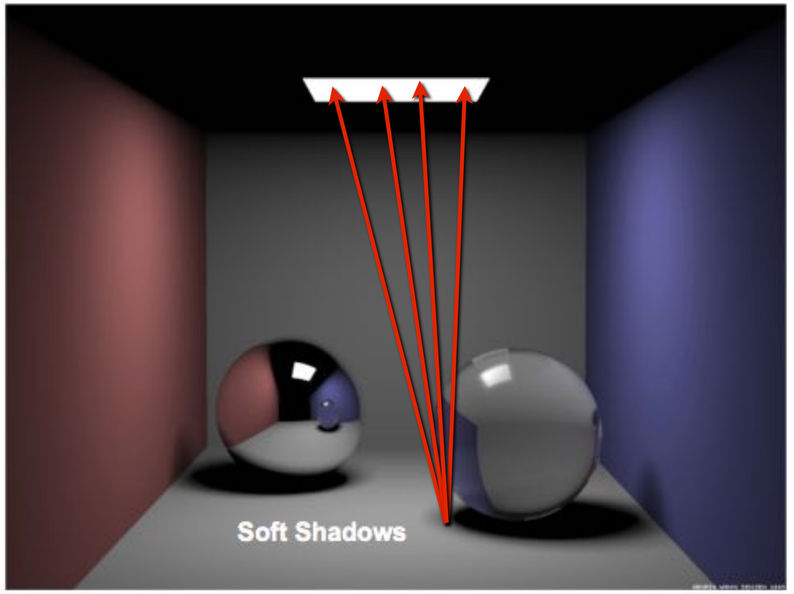
16 area light samples

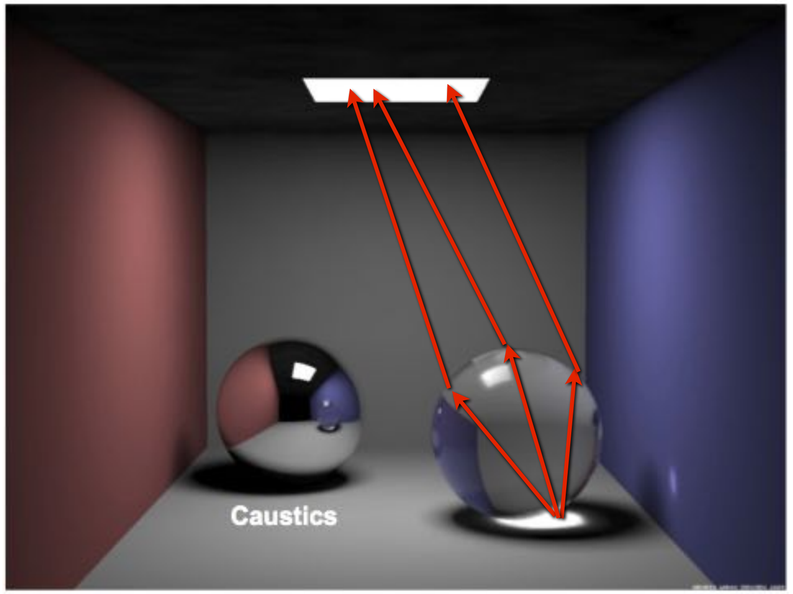
(lower variance in irradiance estimate)



Sampling light paths







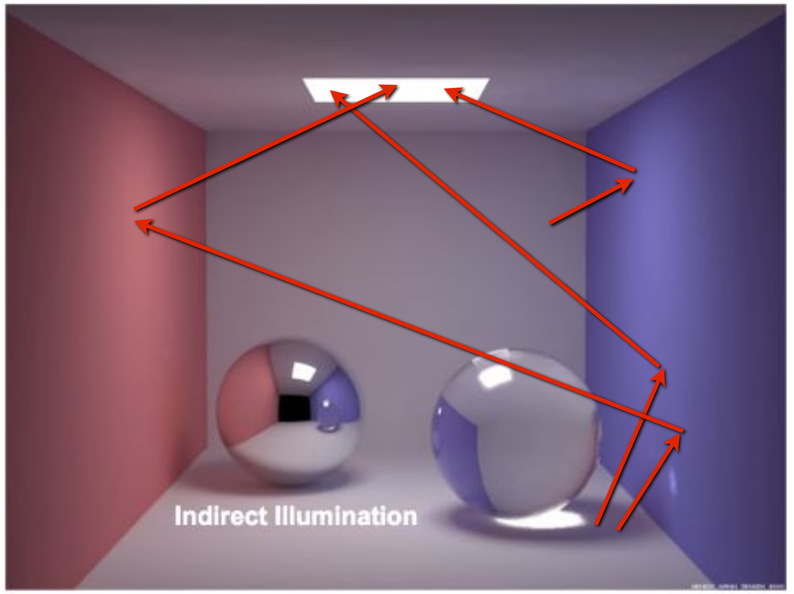
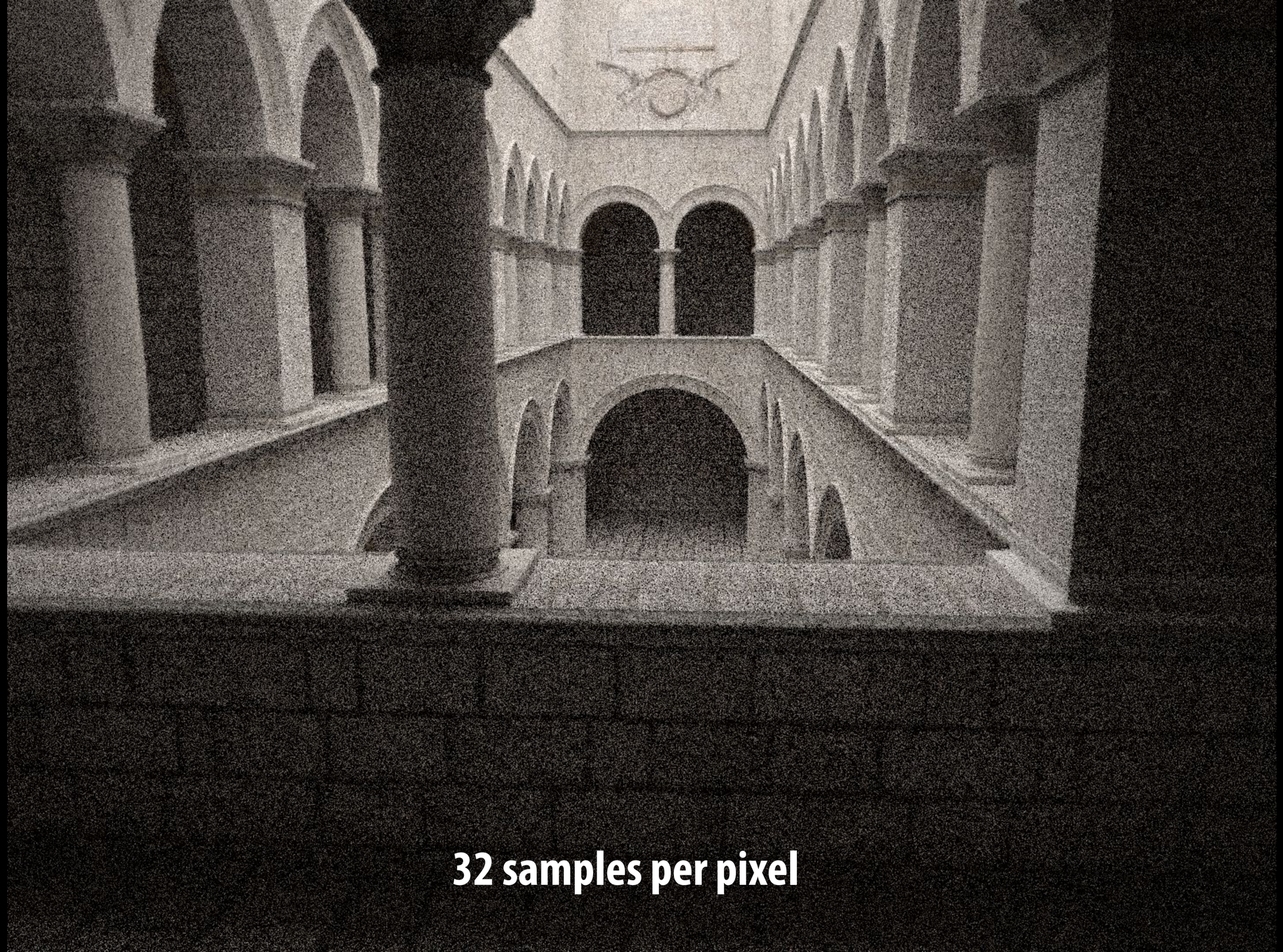


Image credit: Wann Jensen, Hanrahan

One sample per pixel



32 samples per pixel

1024 samples per pixel

Need to shoot many rays per pixel to accurately simulate advanced lighting effects

Want to preserve interactive rendering

Denoising examples

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

Denoising examples

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

Denoising examples

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

Denoising examples

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

Custom GPU hardware for RT + better RL algorithms + DNN-based denoting

NVIDIA GeForce RTX 3080 GPU

This image was rendered in real-time on a single high-end GPU

So was this

Supersampling in a deferred shading system

- In assignment 1, you anti-aliased rendering via supersampling
  - Stored N color samples and N depth samples per pixel

- Deferred shading makes supersampling challenging due to large amount of information that must be stored per pixel
  - 3840 x 2160 (4K display)
  - 8 samples per pixel
  - 20 bytes per G-buffer sample

  = 670MB G-buffer

  (80 GB/sec of memory bandwidth just to read and write the G-buffer at 30 fps)

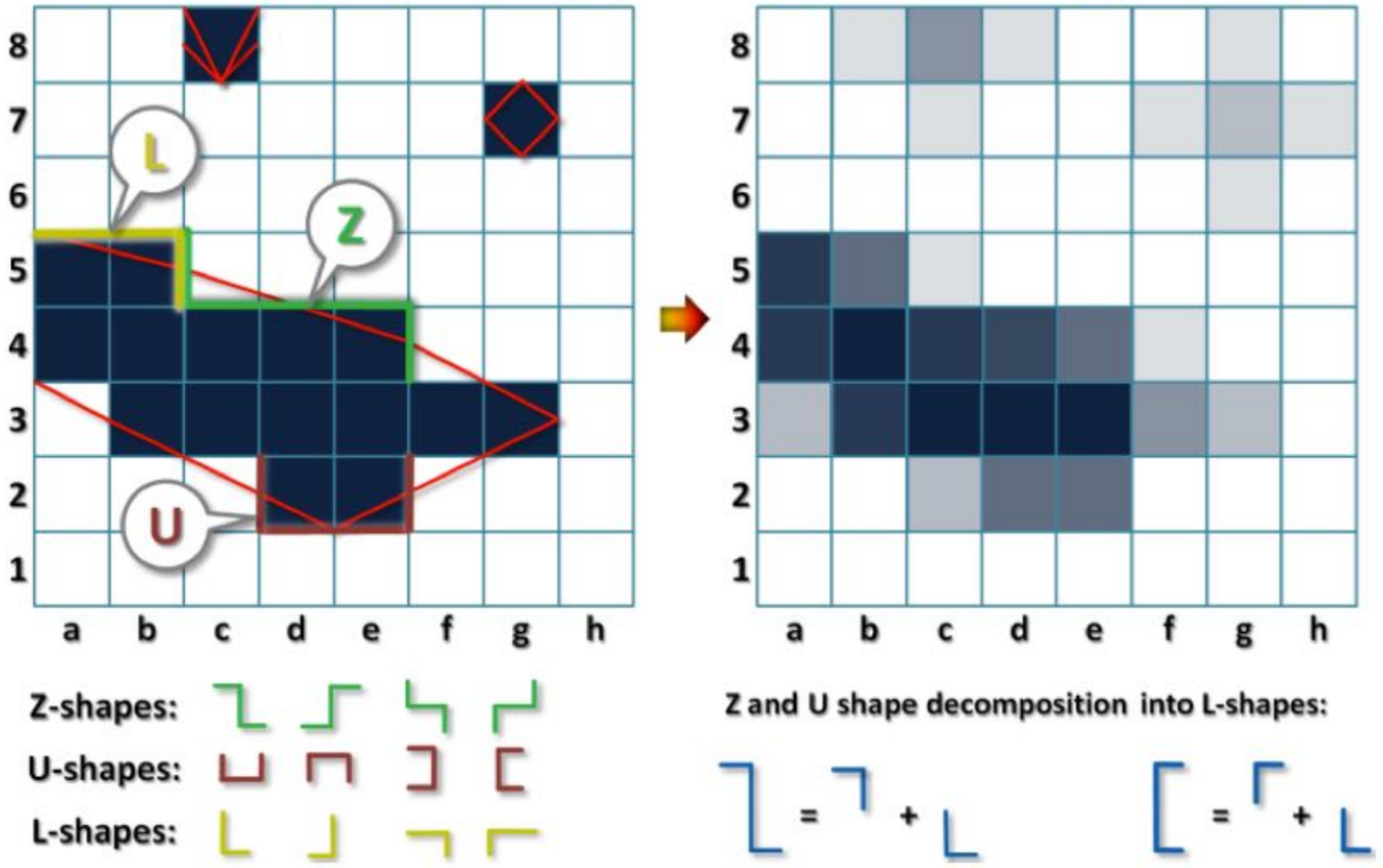
[Reshetov 09]

Morphological anti-aliasing (MLAA)

Detect careful designed patterns in rendered image

For detected patterns, blend neighboring pixels according to a few simple rules

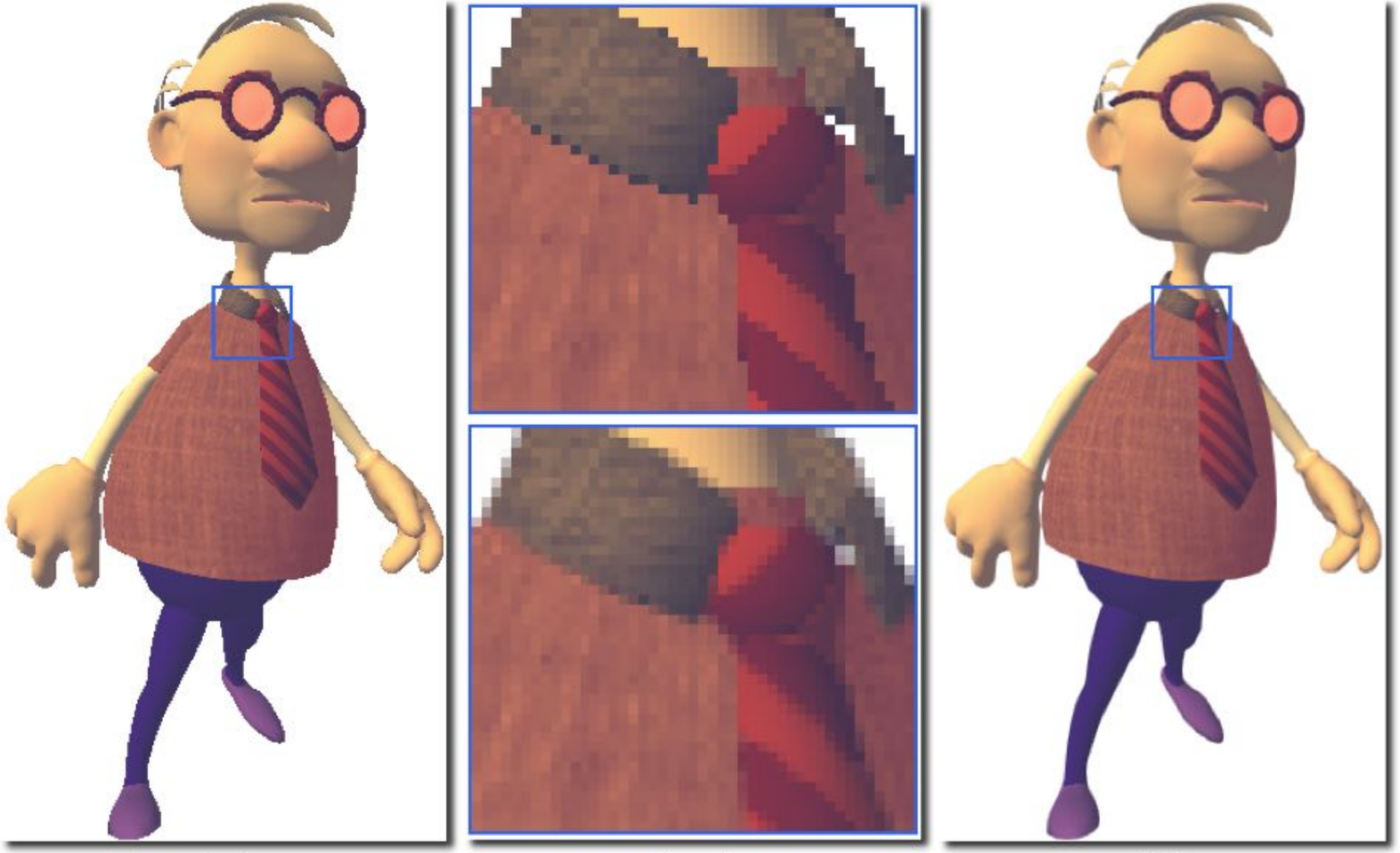
("hallucinate" a smooth edge.. it's a hack!)



Note: modern interest in replacing MLAA patterns with DNN-based anti-aliasing.

Morphological anti-aliasing (MLAA)

[Reshetov 09]



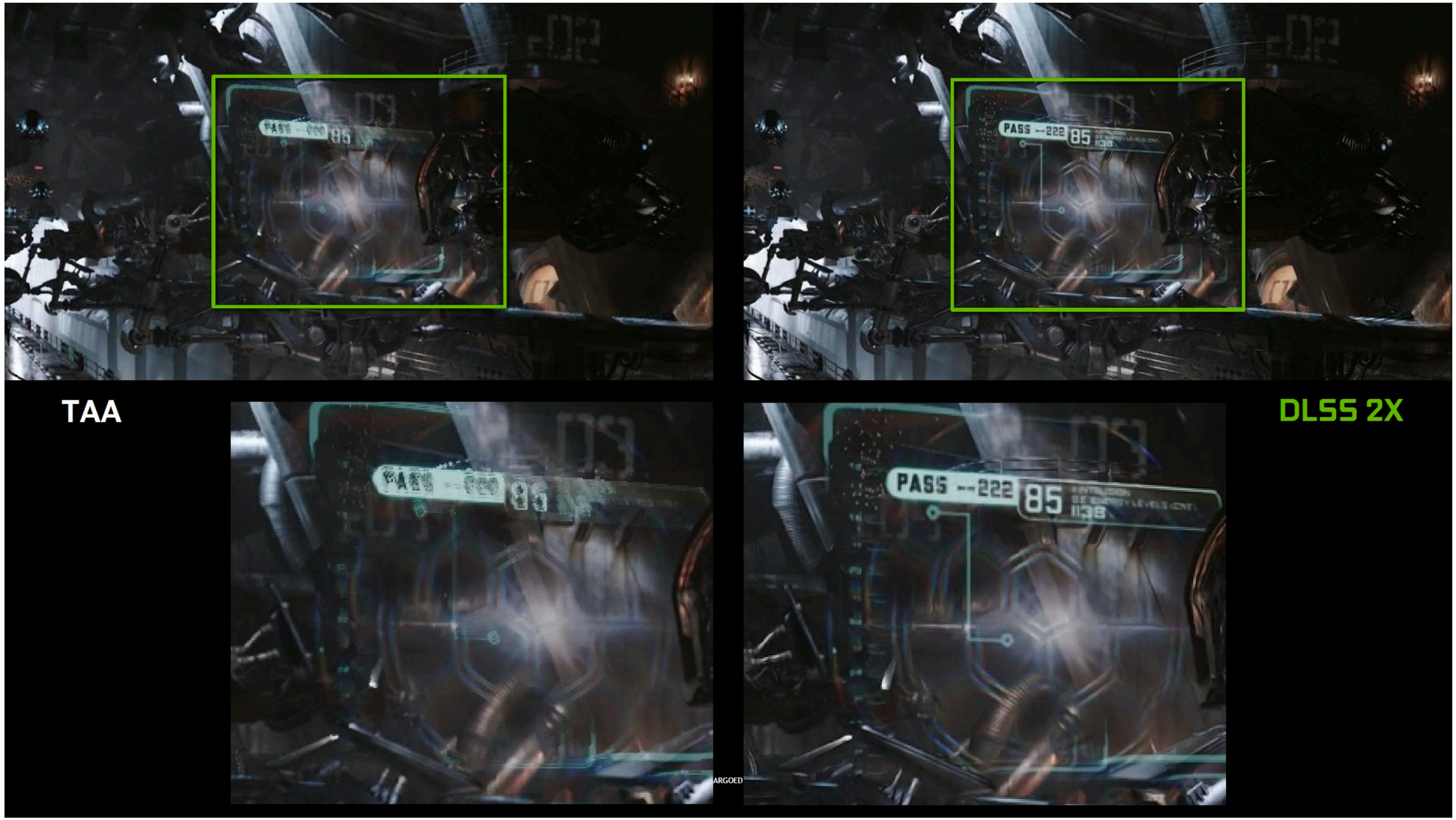
Aliased image
(one shading sample per pixel)

Zoomed views
(top: aliased, bottom: after MLAA)

After filtering using MLAA

Modern trend: learn anti-aliasing functions

Use modern image processing deep networks to reduce aliasing artifacts from rendered images.



https://wccftech.com/nvidia-dlss-explained-nvidia-ngx/

Learn anti-aliasing functions

Use modern image processing deep networks to reduce aliasing artifacts from rendered images.

Traditional Heuristic (TXAA)

Learned AA (DLSS)

https://wccftech.com/nvidia-dlss-explained-nvidia-ngx/

Summary: deferred shading

- Very popular technique in modern games
- Creative use of graphics pipeline
  - Create a G-buffer, not a final image
- Two major motivations
  - Convenience and simplicity of separating geometry processing logic/costs from shading costs
  - Potential for high performance under complex lighting and shading conditions
    - Shade only once per sample despite triangle overlap
    - Often more amenable to "screen-space shading techniques"
      - e.g., screen space ambient occlusion