Lecture 8:

Geometric Queries

**Interactive Computer Graphics
Stanford CS248, Winter 2019**
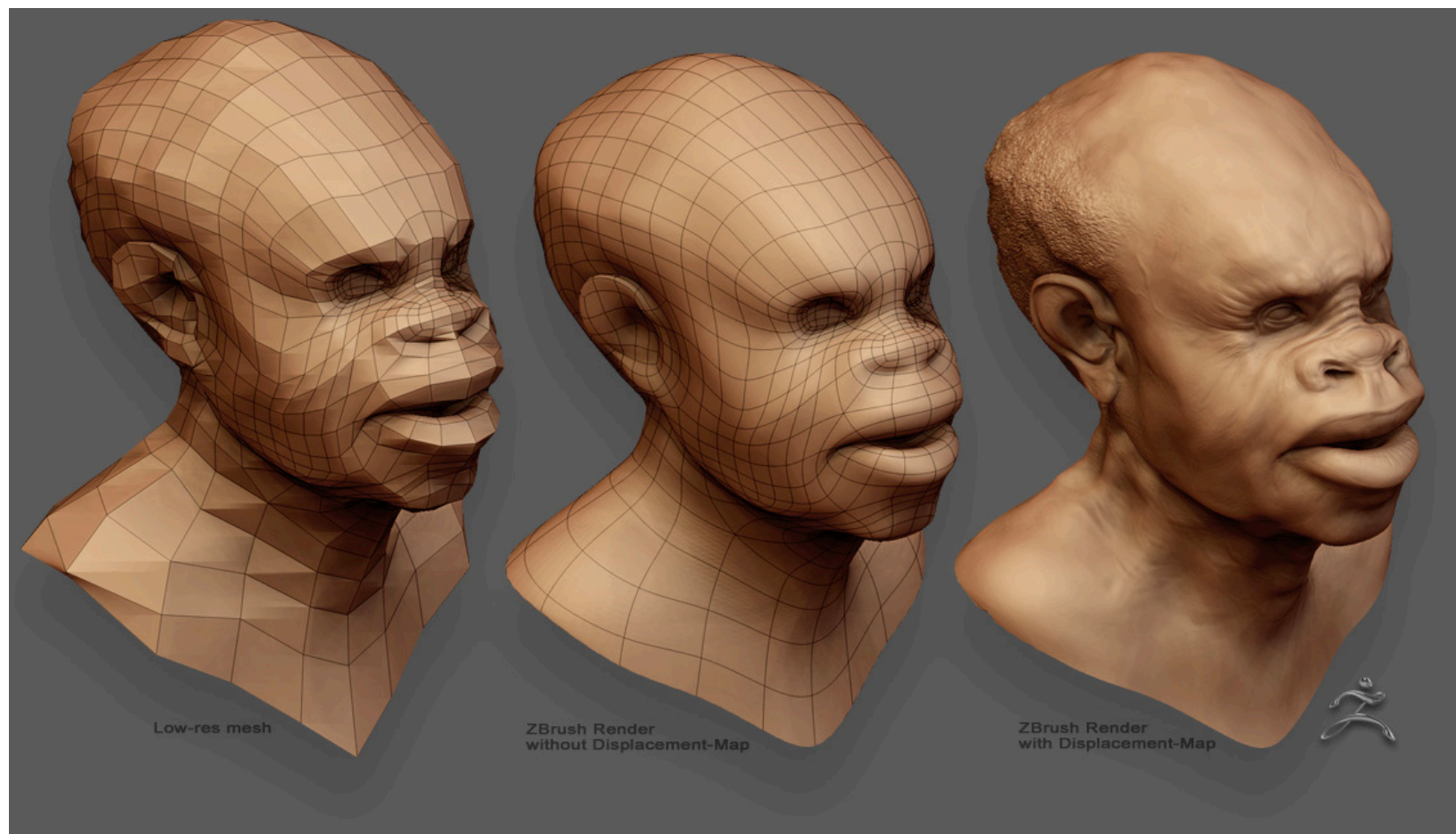
Geometric queries — motivation

**Intersecting rays and triangles (ray tracing)**

Closest point on surface queries

Intersecting triangles (collisions)

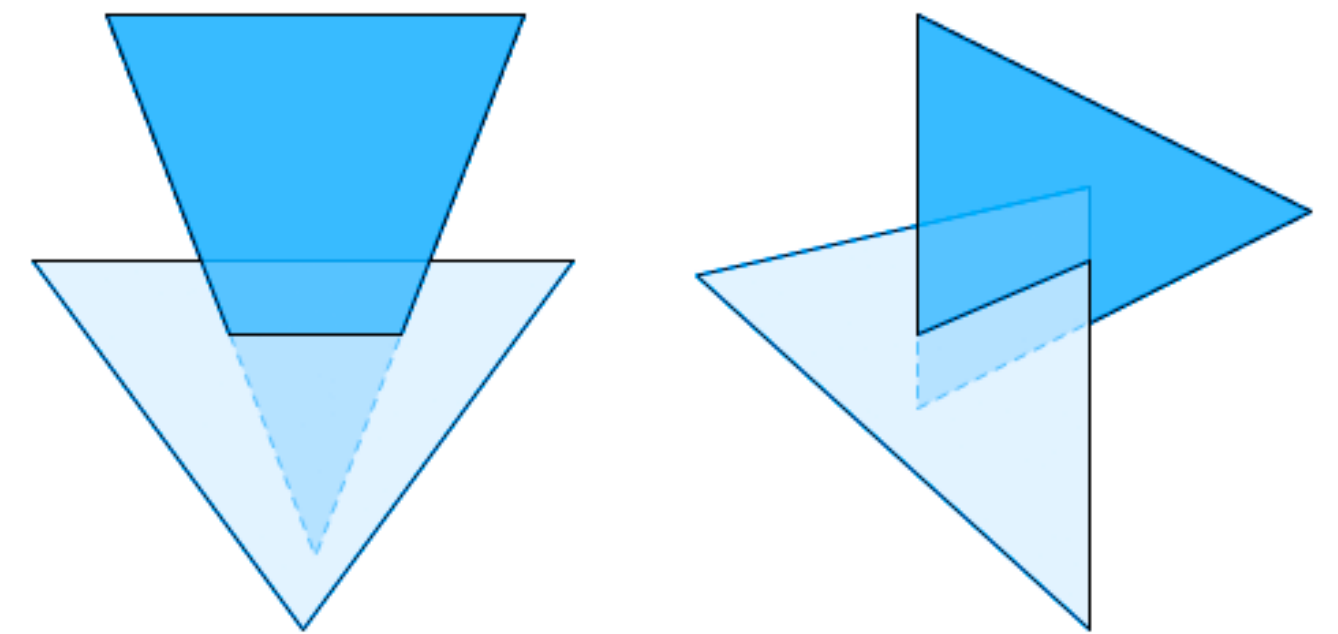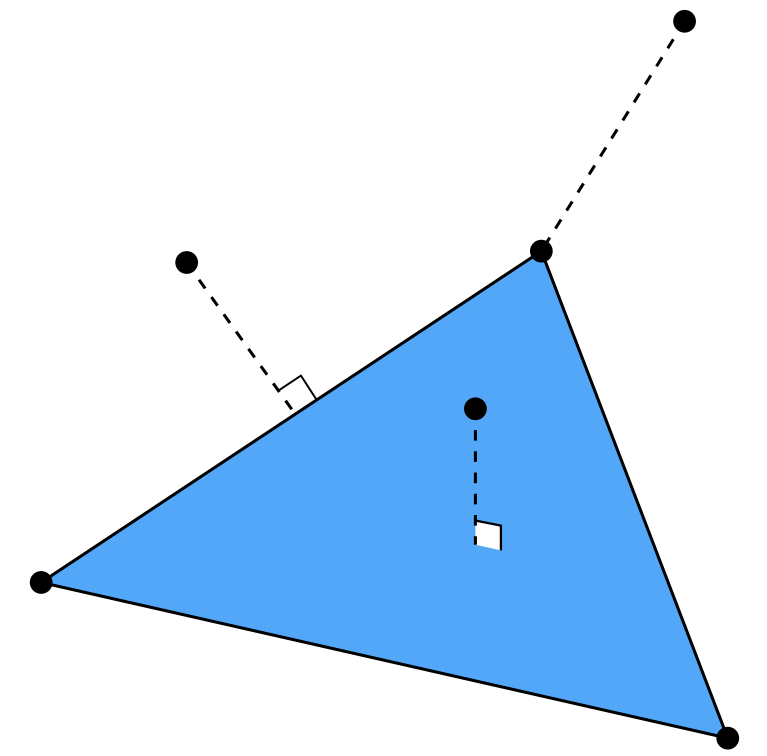# Example: closest point queries

- **Q: Given a point, in space (e.g., a new sample point), how do we find the closest point on a given surface?**

- **Q: Does implicit/explicit representation make this easier?**

- **Q: Does our half-edge data structure help?**

- **Q: What's the cost of the naïve algorithm?**

- **Q: How do we find the distance to a single triangle anyway?**
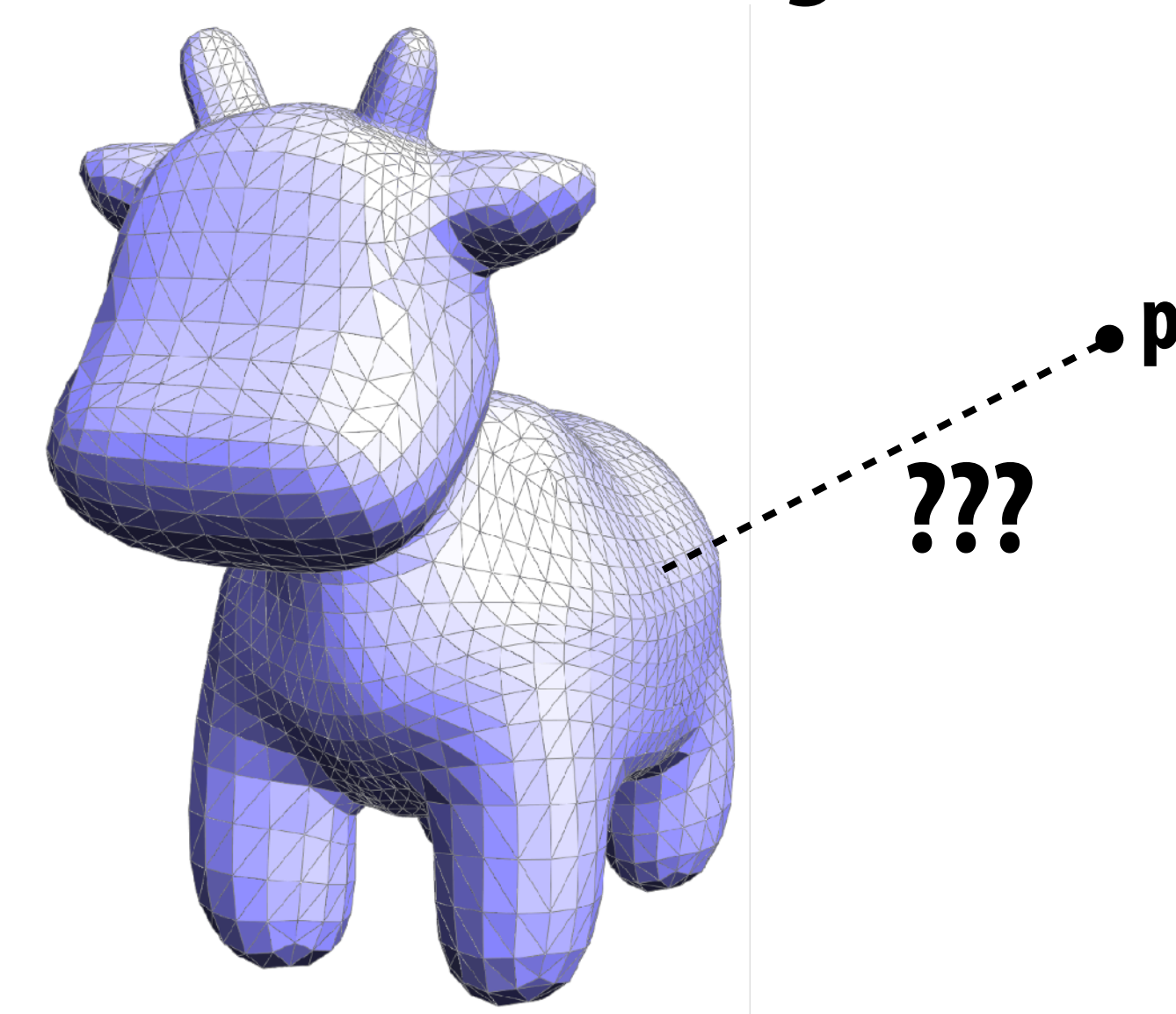
Many types of geometric queries

- **Plenty of other things we might like to know:**
  - Do two triangles intersect?
  - Are we inside or outside an object?
  - Does one object contain another?
  - ...

- **Data structures we've seen so far not really designed for this...**
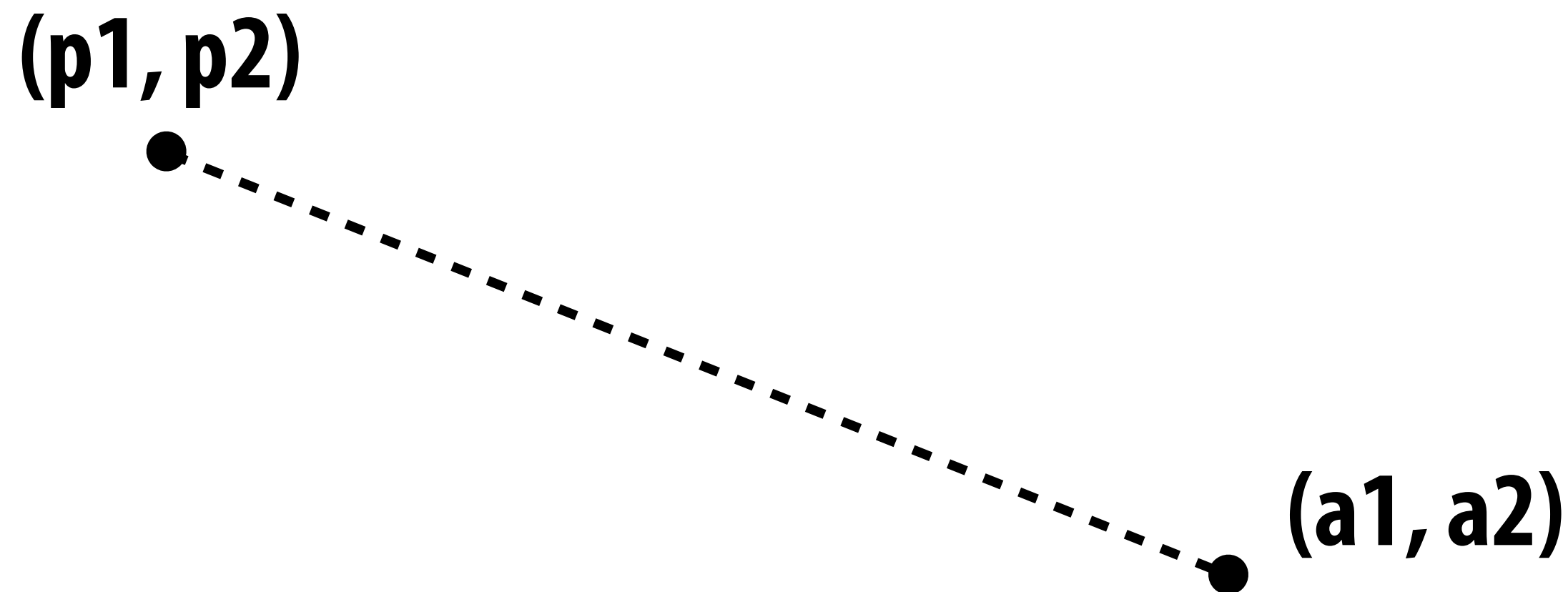
- **Need some new ideas!**

- **TODAY: come up with simple (aka: slow) algorithms**

- **NEXT TIME: intelligent ways to accelerate geometric queries**
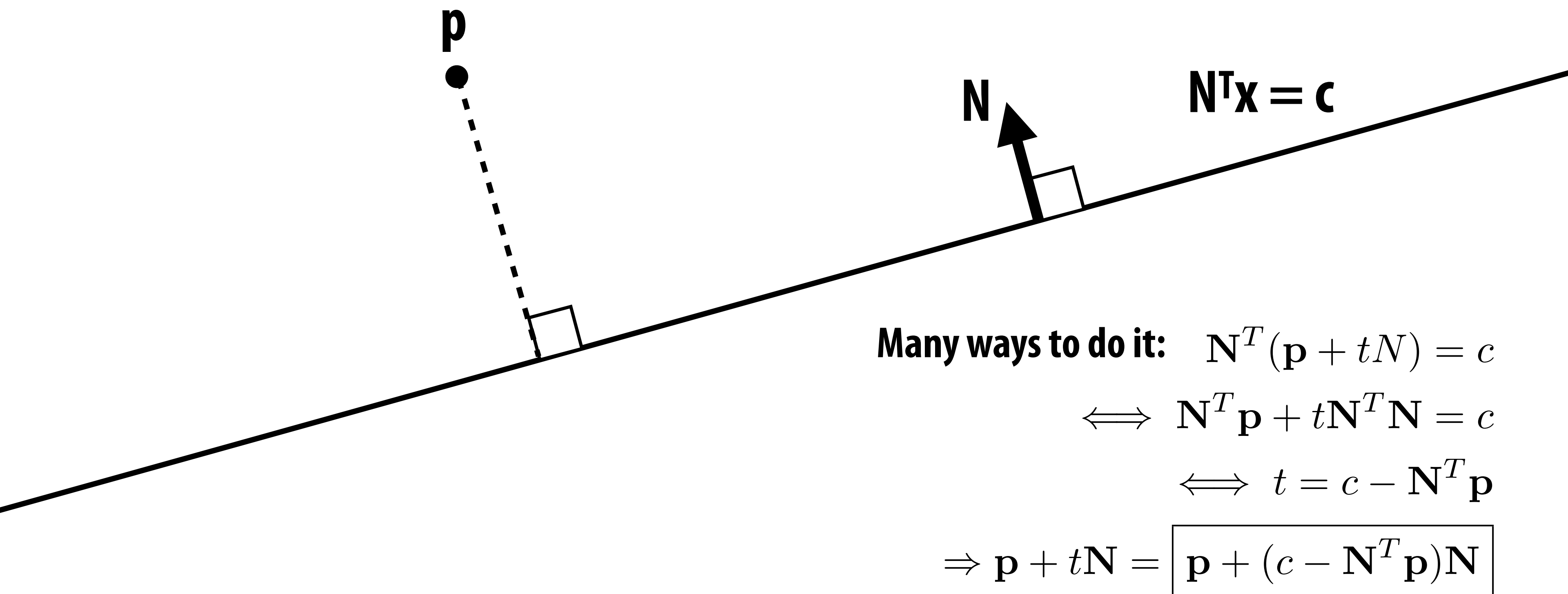
Warm up: closest point on point

- *G*iven a query point (p1,p2), how do we find the closest point on the point (a1,a2)?

**(p1, p2)**

(a1, a2)

Bonus question: what's the distance?

Slightly harder: closest point on line

- **Now suppose I have a line N$^{\mathsf{T}}$x = c, where N is the unit normal**
- **How do I find the point on line closest to my query point p?**

p

N

N$^{\mathsf{T}}$x = c

Many ways to do it: $\mathbf{N}^T(\mathbf{p} + tN) = c$

$$\Longleftrightarrow \mathbf{N}^T\mathbf{p} + t\mathbf{N}^T\mathbf{N} = c$$

$$\Longleftrightarrow t = c - \mathbf{N}^T\mathbf{p}$$

$$\Rightarrow \mathbf{p} + t\mathbf{N} = \boxed{\mathbf{p} + (c - \mathbf{N}^T\mathbf{p})\mathbf{N}}$$

Harder: closest point on line segment

- **Two cases: endpoint or interior**

- **Already have basic components:**

  - **point-to-point**

  - **point-to-line**

- **Algorithm?**

  - **find closest point on line**

  - **check if it is between endpoints**

  - **if not, take closest endpoint**

- **How do we know if it's between endpoints?**

  - **write closest point on line as a+t(b-a)**

  - **if t is between 0 and 1, it's inside the segment!**
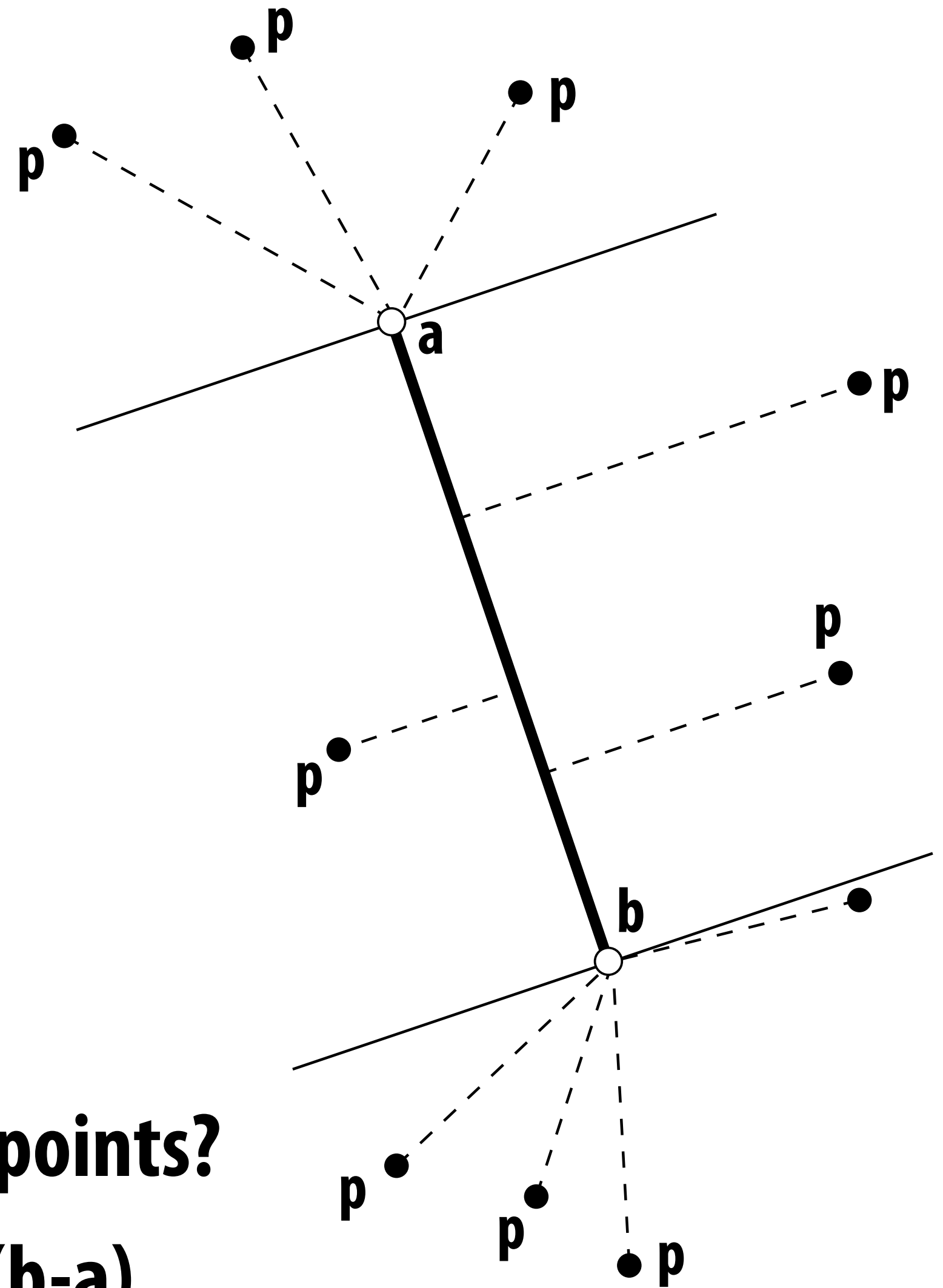
Even harder: closest point on triangle in 2D

- **What are all the possibilities for the closest point?**

- **Almost just minimum distance to three line segments:**
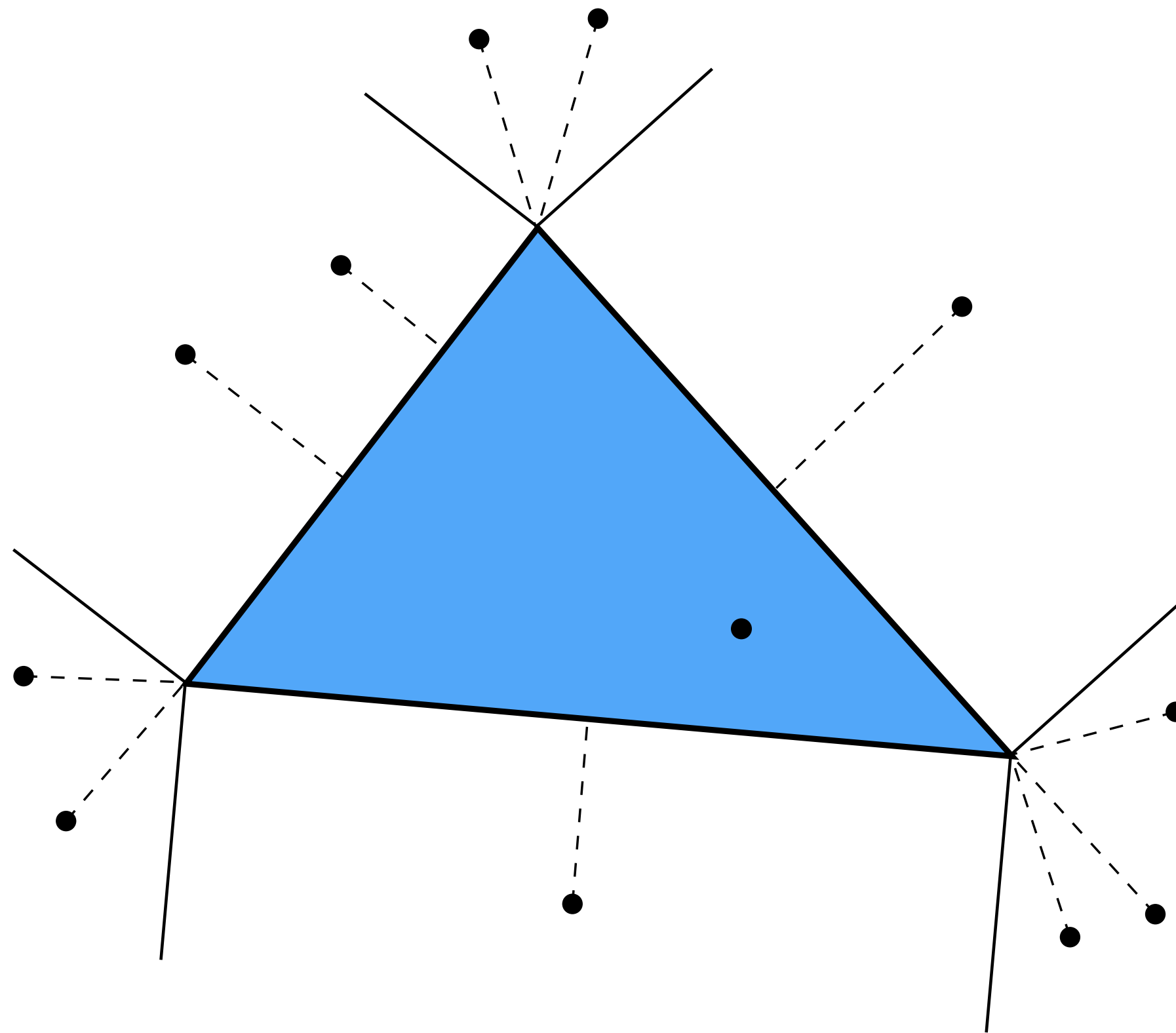
Q: What about a point inside the triangle?

Closest point on triangle in 3D

- **Not so different from 2D case**

- **Algorithm:**

  - **Project point onto plane of triangle**

  - **Use half-*space* tests to classify point (vs. half plane)**

  - **If inside the triangle, we're done!**

  - **Otherwise, find closest point on associated vertex or edge**

- **By the way, how do we find closest point on plane?**

- **Same expression as closest point on a line!** $p + (c - N^T p) N$

Closest point on triangle *mesh* in 3D?

■ **Conceptually easy:**

- **loop over all triangles**

- **compute closest point to current triangle**

- **keep globally closest point**

■ **Q: What's the cost?**

■ **What if we have *billions* of faces?**

■ **NEXT TIME: Better data structures!**

p
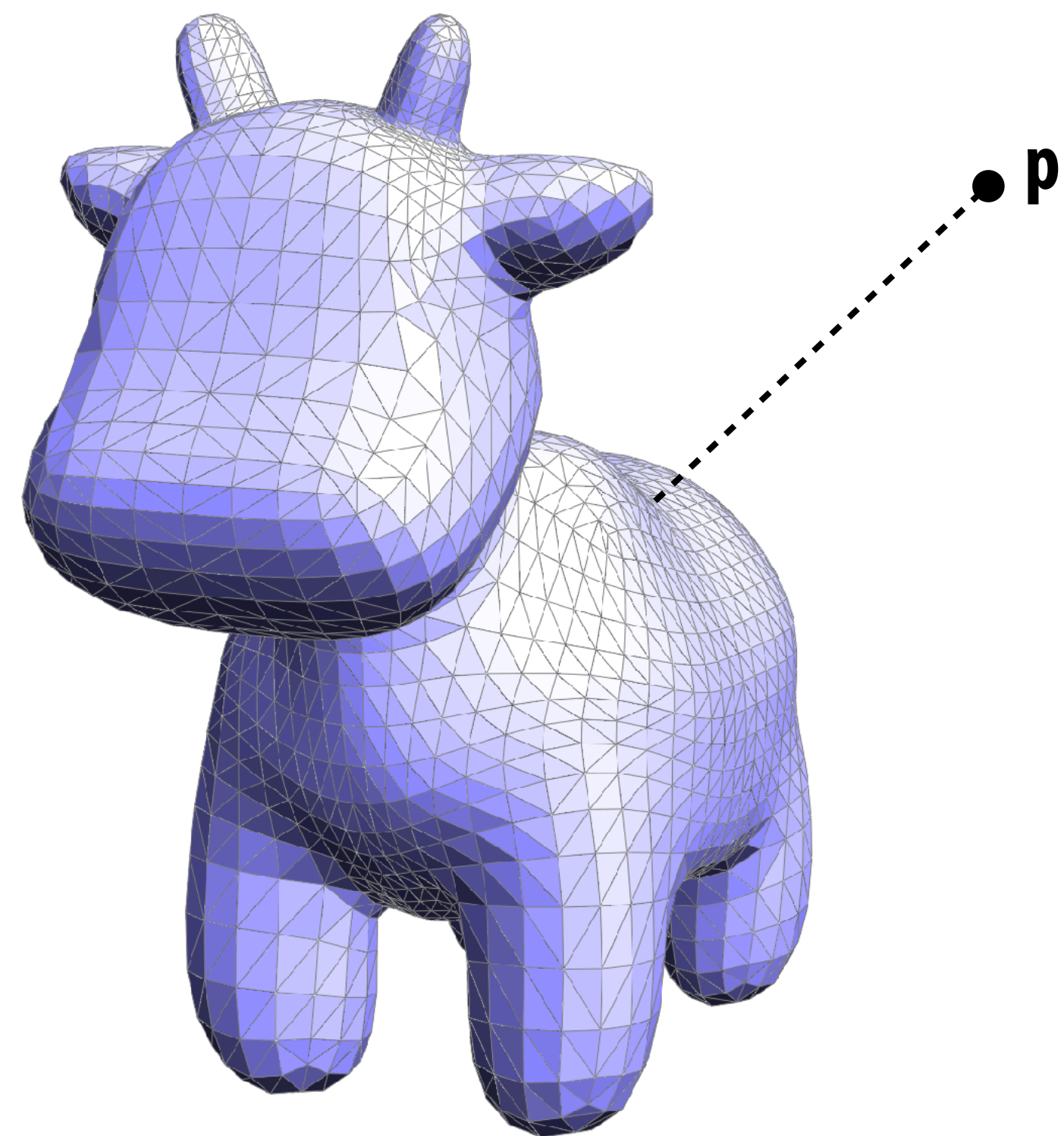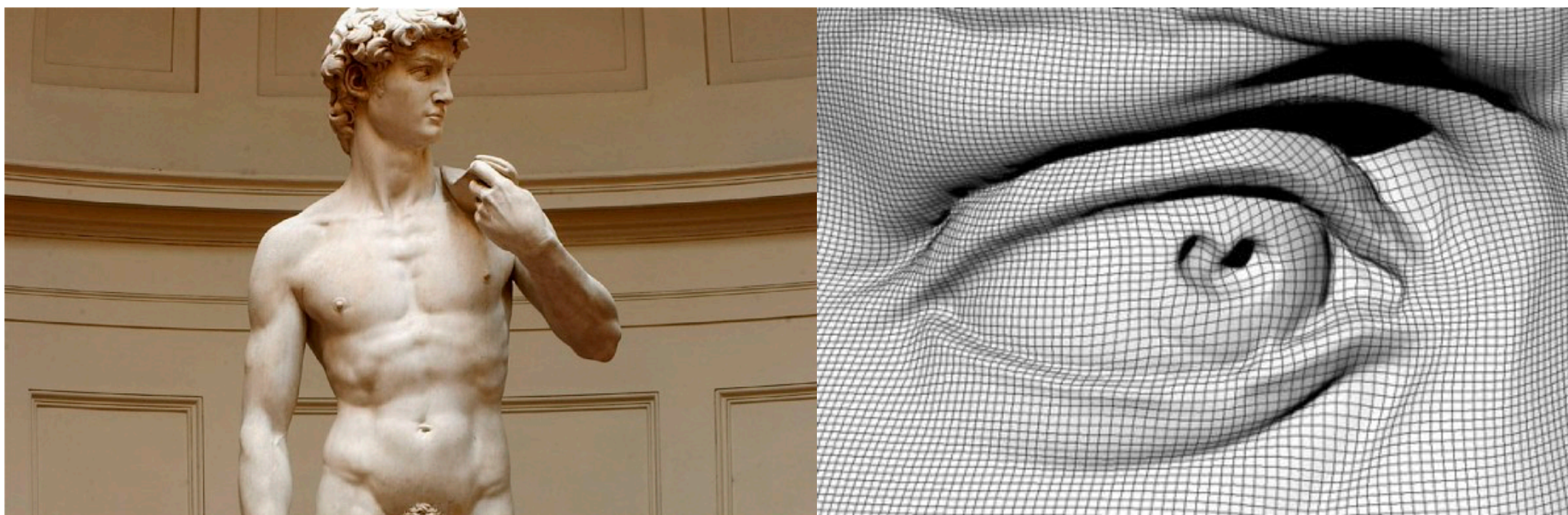
Closest point to *implicit* surface?

- **If we change our representation of geometry, algorithms can change completely**

- **E.g., how might we compute the closest point on an implicit surface described via its distance function?**

- **One idea:**
  - **start at the query point**
  - **compute gradient of distance (using, e.g., finite differences)**
  - **take a little step (decrease distance)**
  - **repeat until we're at the surface (zero distance)**

- **Better yet: just store closest point for each grid cell! (speed/memory trade off)**
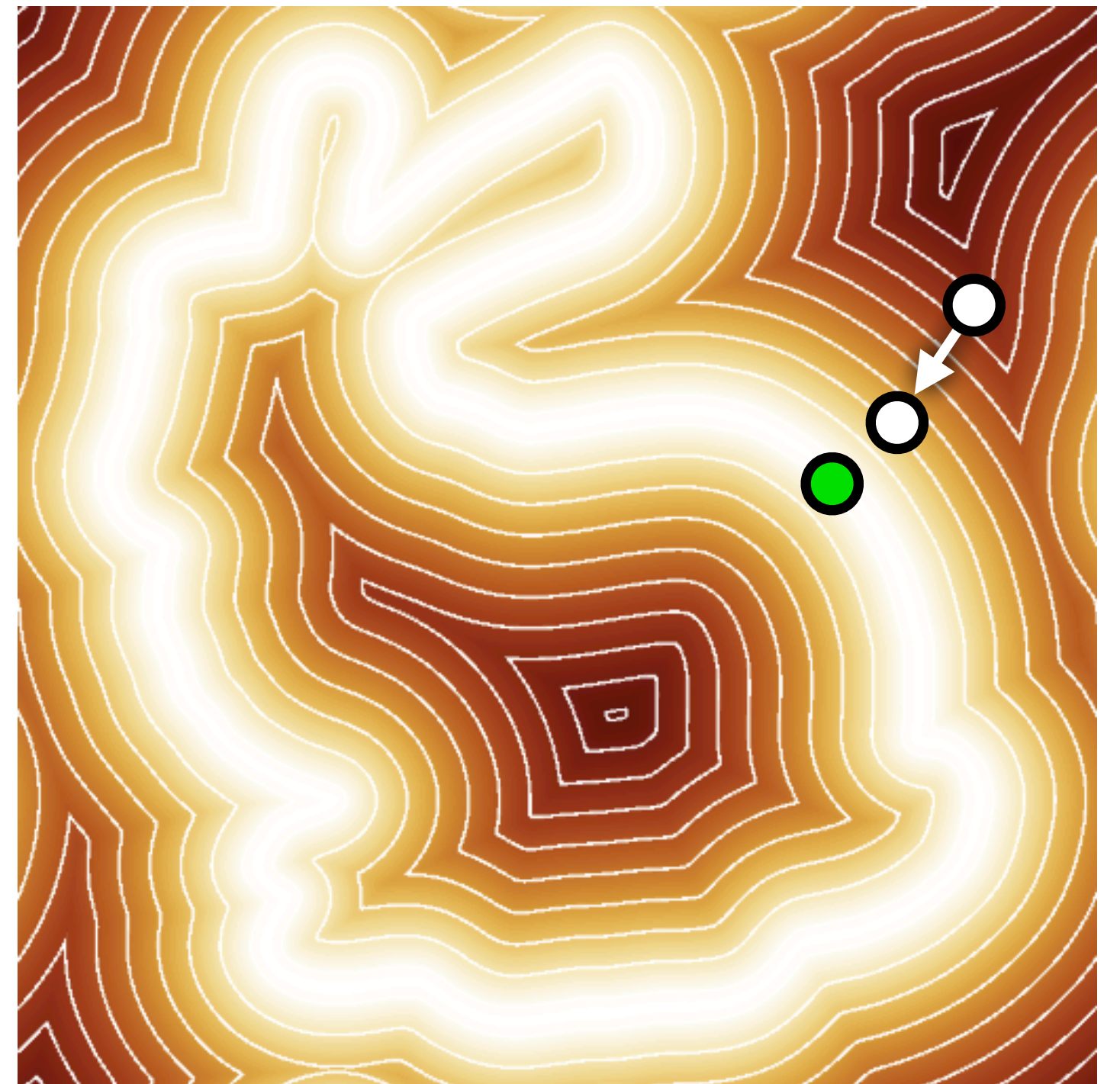
Different query: ray-mesh intersection

- A "ray" is an oriented line starting at a point

- Think about a ray of light traveling from the sun

- Want to know where a ray pierces a surface

- Why?

  - GEOMETRY: inside-outside test

  - RENDERING: visibility, ray tracing

  - ANIMATION: collision detection

- Might pierce surface in many places!
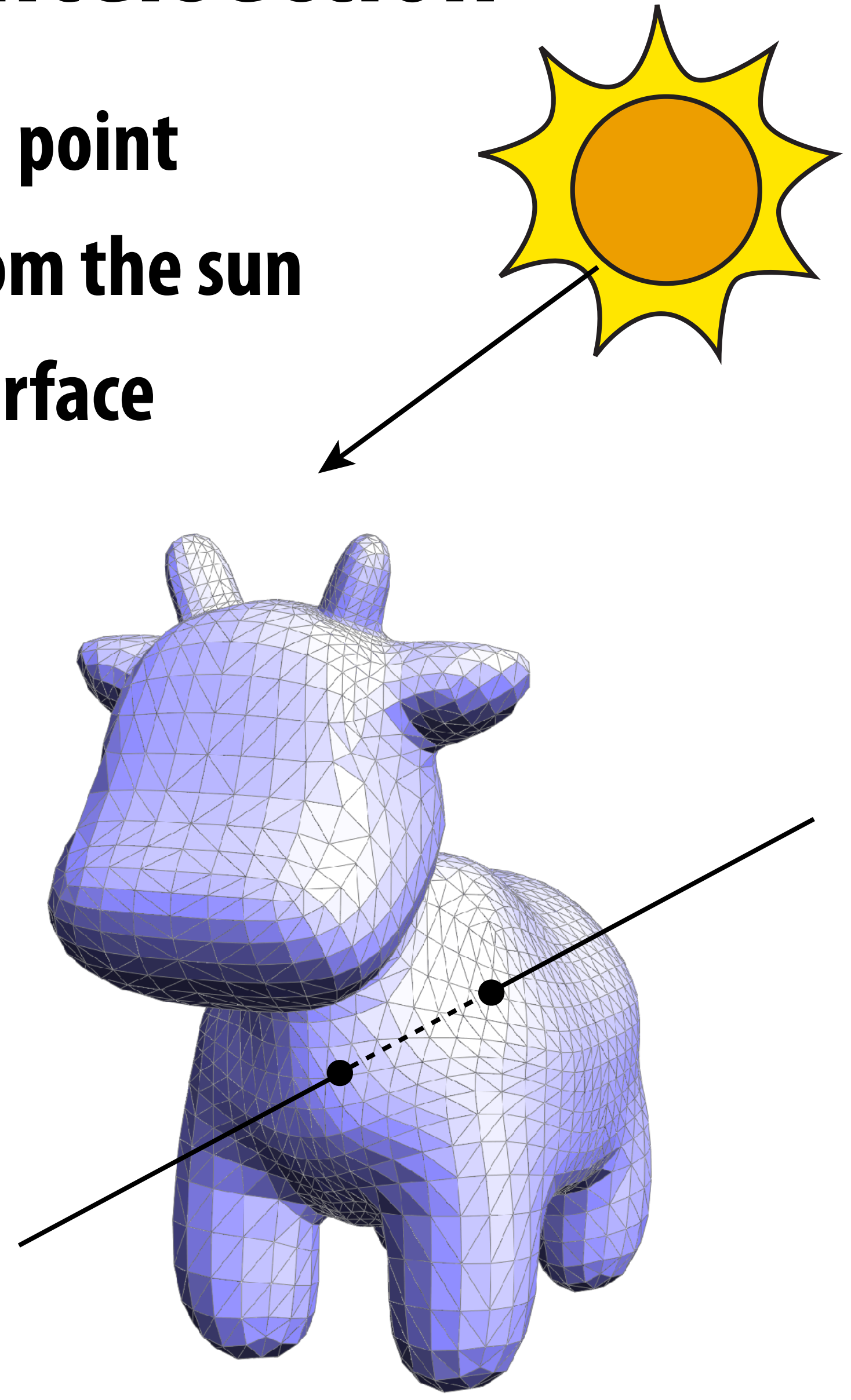
Ray equation

- **Can express ray as**

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

point along ray

"time"

$$\mathbf{d}$$

$$\mathbf{o}$$

$$r(t)$$
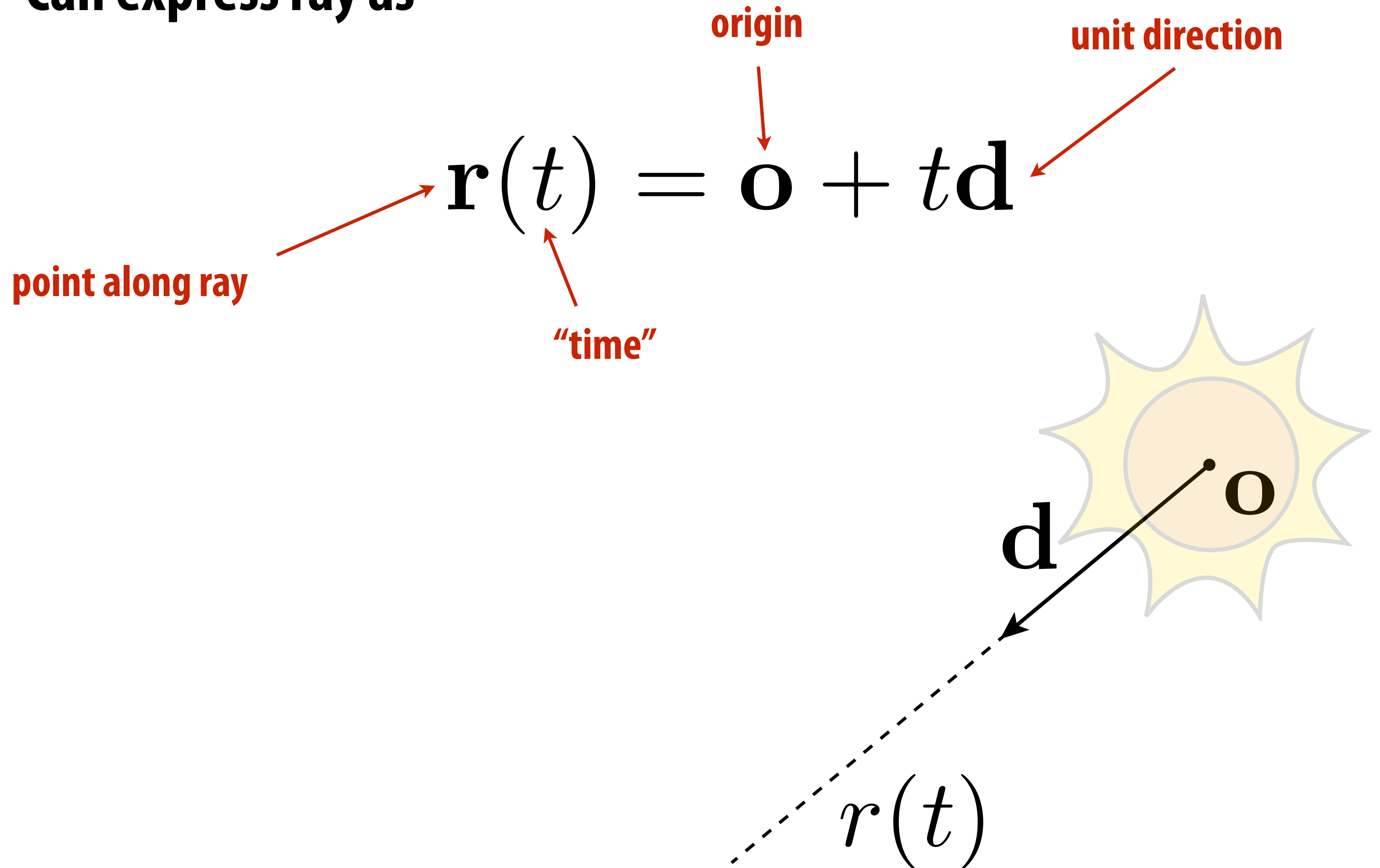
Intersecting a ray with an implicit surface

- **Recall implicit surfaces: all points x such that f(x) = 0**

- **Q: How do we find points where a ray pierces this surface?**

- **Well, we know all points along the ray: r(t) = o + td**

- **Idea: replace "x" with "r" in 1st equation, and solve for t**

- **Example: unit sphere**

$$f(\mathbf{x}) = |\mathbf{x}|^2 - 1$$

$$\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$
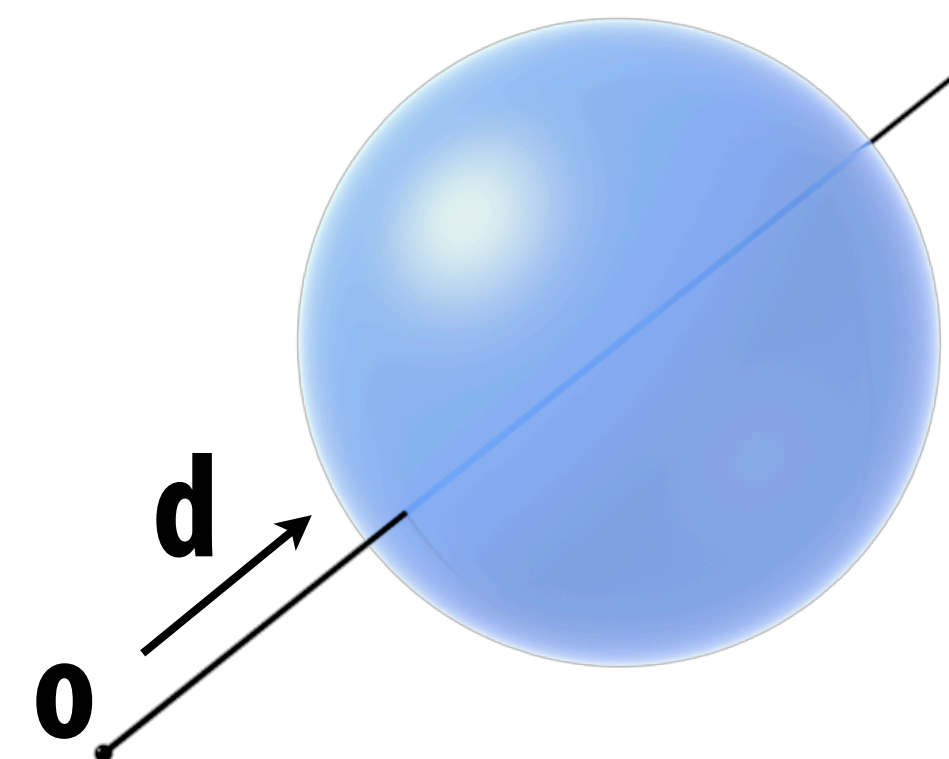
$$\underbrace{|\mathbf{d}|^2}_{a} t^2 + \underbrace{2(\mathbf{o} \cdot \mathbf{d})}_{b} t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

Note: $|\mathbf{d}|^2 = 1$ **since d is a unit vector**

$$t = \boxed{-\mathbf{o} \cdot \mathbf{d} \pm \sqrt{(\mathbf{o} \cdot \mathbf{d})^2 - |\mathbf{o}|^2 + 1}}$$

quadratic formula:
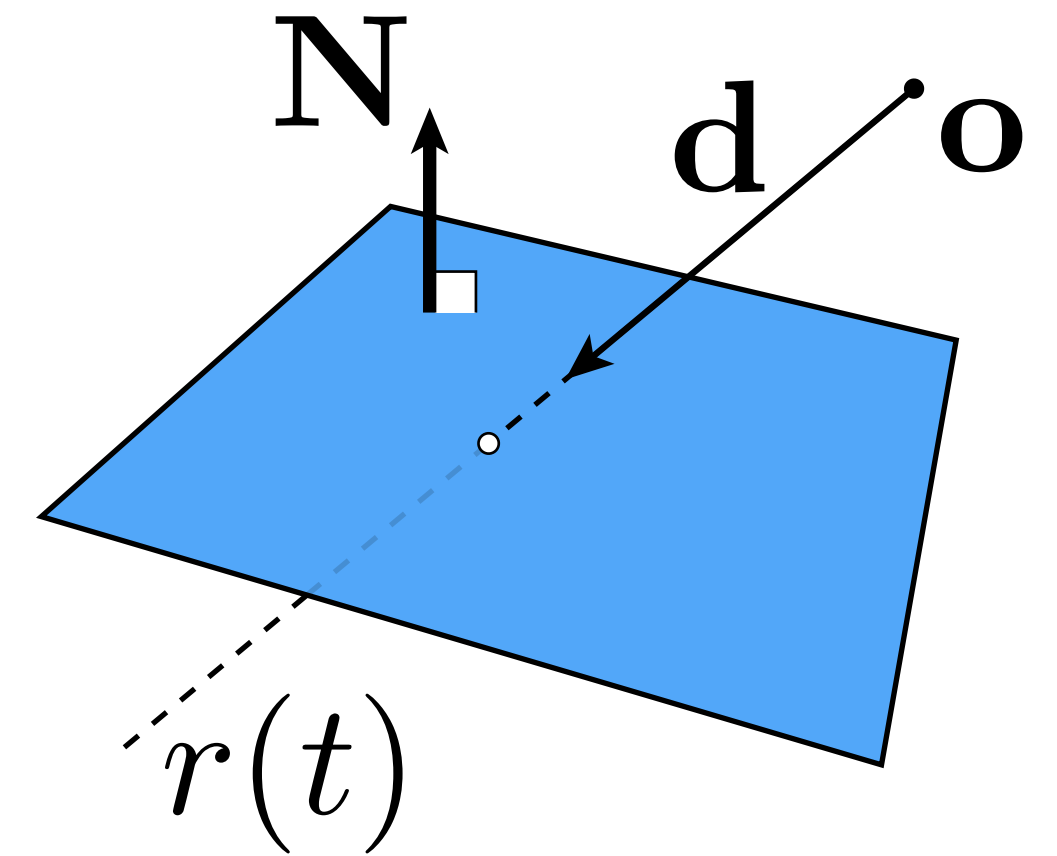
$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

d

o

Why two solutions?

Ray-plane intersection

- **Suppose we have a plane N$^{\mathsf{T}}$x = c**
  - **N - unit normal**
  - **c - offset**

- **How do we find intersection with ray r(t) = o + td?**

- ***Key idea:*** **again, replace the point x with the ray equation t:**

$$\mathbf{N}^{\mathsf{T}}\mathbf{r}(t) = c$$

- **Now solve for t:**

$$\mathbf{N}^{\mathsf{T}}(\mathbf{o} + t\mathbf{d}) = c \qquad \Rightarrow t = \frac{c - \mathbf{N}^{\mathsf{T}}\mathbf{o}}{\mathbf{N}^{\mathsf{T}}\mathbf{d}}$$

- **And plug t back into ray equation:**

$$r(t) = \mathbf{o} + \frac{c - \mathbf{N}^{\mathsf{T}}\mathbf{o}}{\mathbf{N}^{\mathsf{T}}\mathbf{d}}\mathbf{d}$$

Ray-triangle intersection

- **Triangle is in a plane...**

- **Algorithm:**

  - **Compute ray-plane intersection**

  - **Q: What do we do now?**
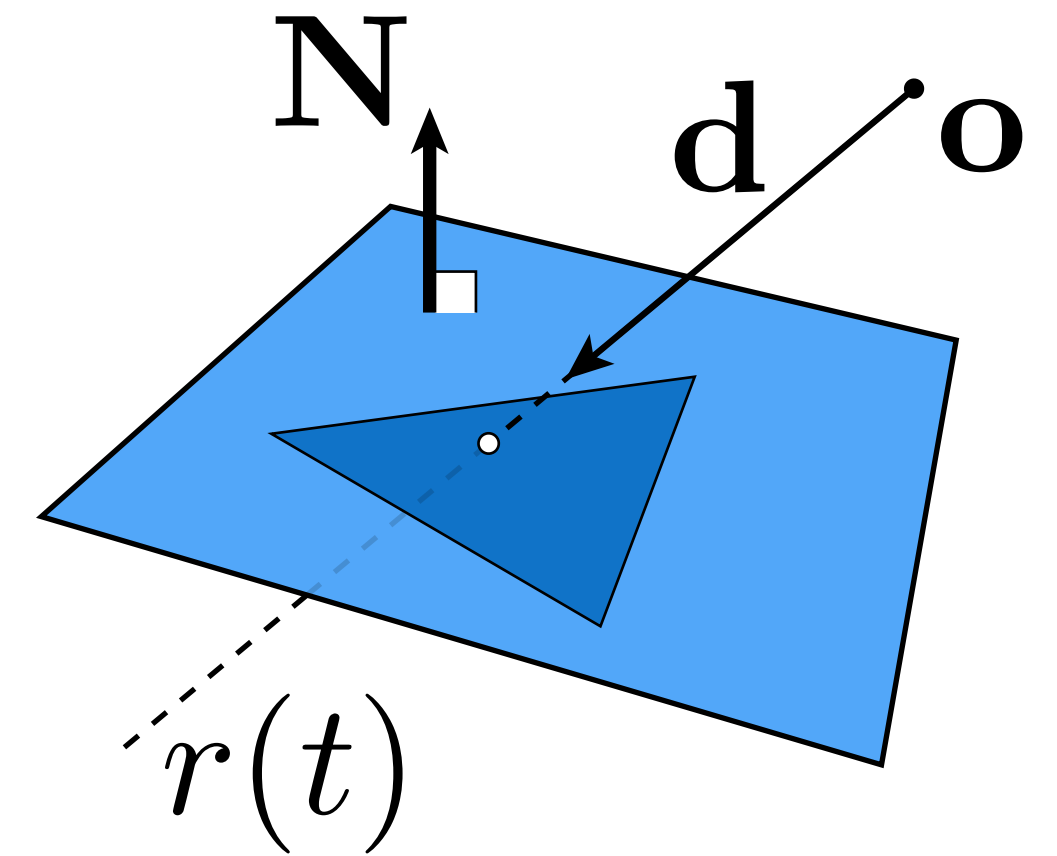
Barycentric coordinates (as ratio of areas)

Barycentric coords are *signed* areas:

$$\alpha = A_A/A$$

$$\beta = A_B/A$$

$$\gamma = A_C/A$$

Why must coordinates sum to one?
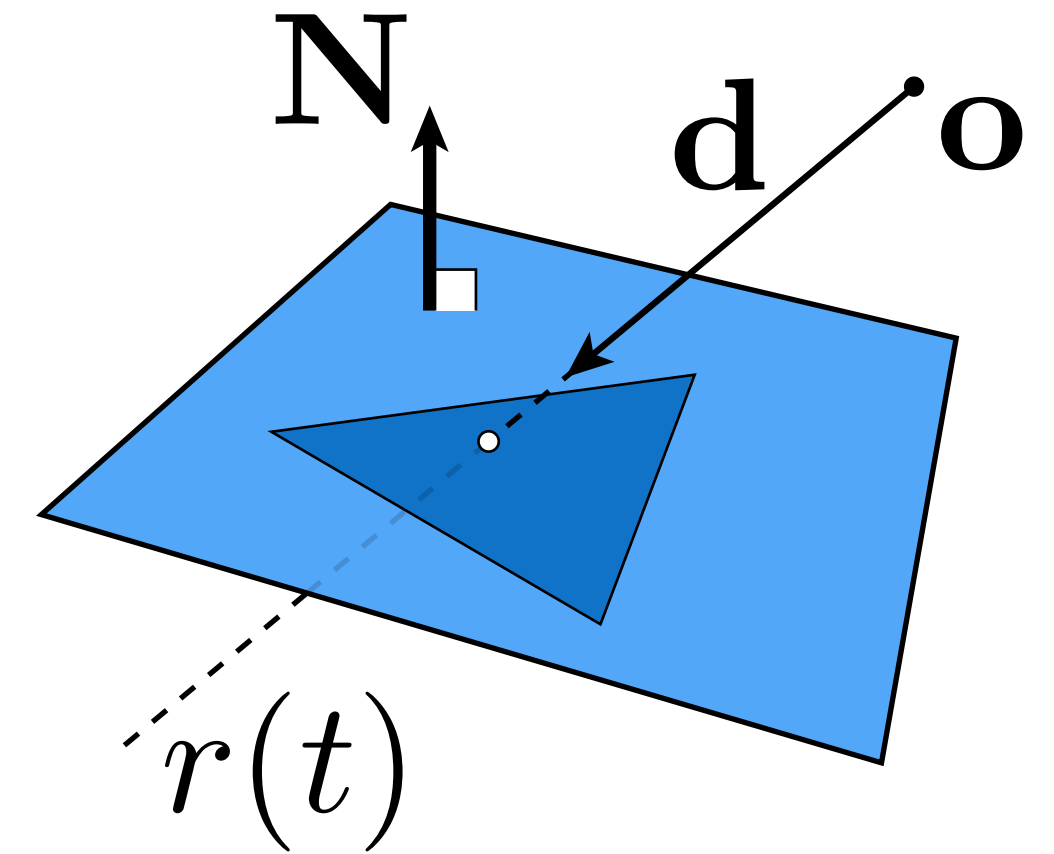
Why must coordinates be between 0 and 1?
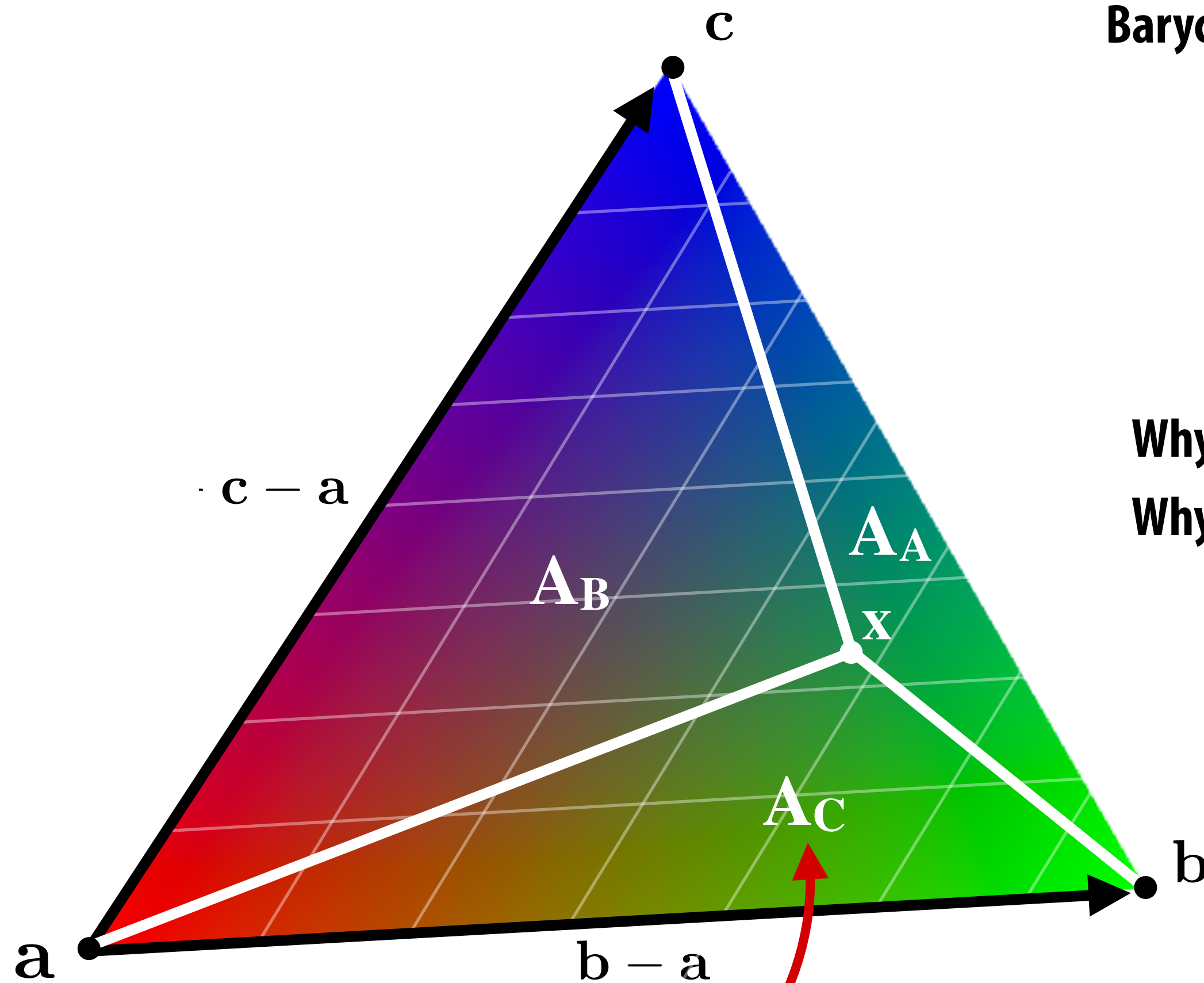
Area of triangle formed by points: a, b, x

Useful: Heron's formula:

$$A_C = \frac{1}{2}(\mathbf{b} - \mathbf{a}) \times (\mathbf{x} - \mathbf{a})$$
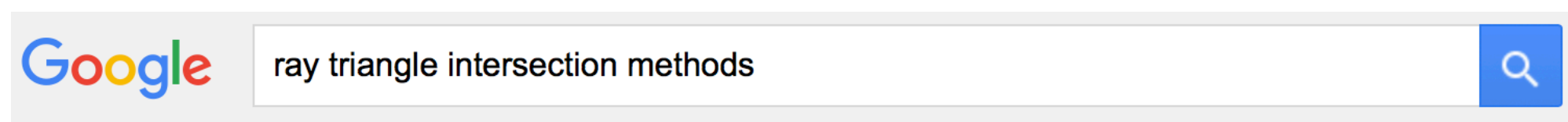
Ray-triangle intersection

- **Algorithm:**

  - **Compute ray-plane intersection**

  - **Q: What do we do now?**

  - **A: Compute barycentric coordinates of hit point?**

  - **If barycentric coordinates are all positive, point is in triangle**

- **Many different techniques if you care about efficiency**

Google   ray triangle intersection methods

Web   Shopping   Videos   News   Images   More ▾   Search tools

About 443,000 results (0.44 seconds)

Möller–Trumbore intersection algorithm - Wikipedia, the free ...
https://en.wikipedia.org/.../Möller–Trumbore_intersection_alg... ▾ Wikipedia ▾
The Möller–Trumbore **ray-triangle intersection** algorithm, named after its inventors
Tomas Möller and Ben Trumbore, is a fast **method** for calculating the ...

[PDF] Fast Minimum Storage Ray-Triangle Intersection.pdf
https://www.cs.virginia.edu/.../Fast%20MinimumSt... ▾ University of Virginia ▾
by PC AB - Cited by 650 - Related articles
We present a clean algorithm for determining whether a **ray intersects a triangle**. ... ble

[PDF] Optimizing Ray-Triangle Intersection via Automated Search
www.cs.utah.edu/~aek/research/**triangle**.pdf ▾ University of Utah ▾
by A Kensler - Cited by 33 - Related articles
method is used to further optimize the code produced via the fitness function. ... For
these 3D **methods** we optimize **ray-triangle intersection** in two different **ways**.

[PDF] Comparative Study of Ray-Triangle Intersection Algorithms
www.graphicon.ru/html/proceedings/2012/.../gc2012Shumskiy.pdf ▾
by V Shumskiy - Cited by 1 - Related articles

Another way: ray-triangle intersection

- **Parameterize triangle given by vertices $\mathrm{p_0}, \mathrm{p_1}, \mathrm{p_2}$ using barycentric coordinates**

$$f(u, v) = (1 - u - v)\mathbf{p_0} + u\mathbf{p_1} + v\mathbf{p_2}$$

- **Can think of a triangle as an affine map of the unit triangle**
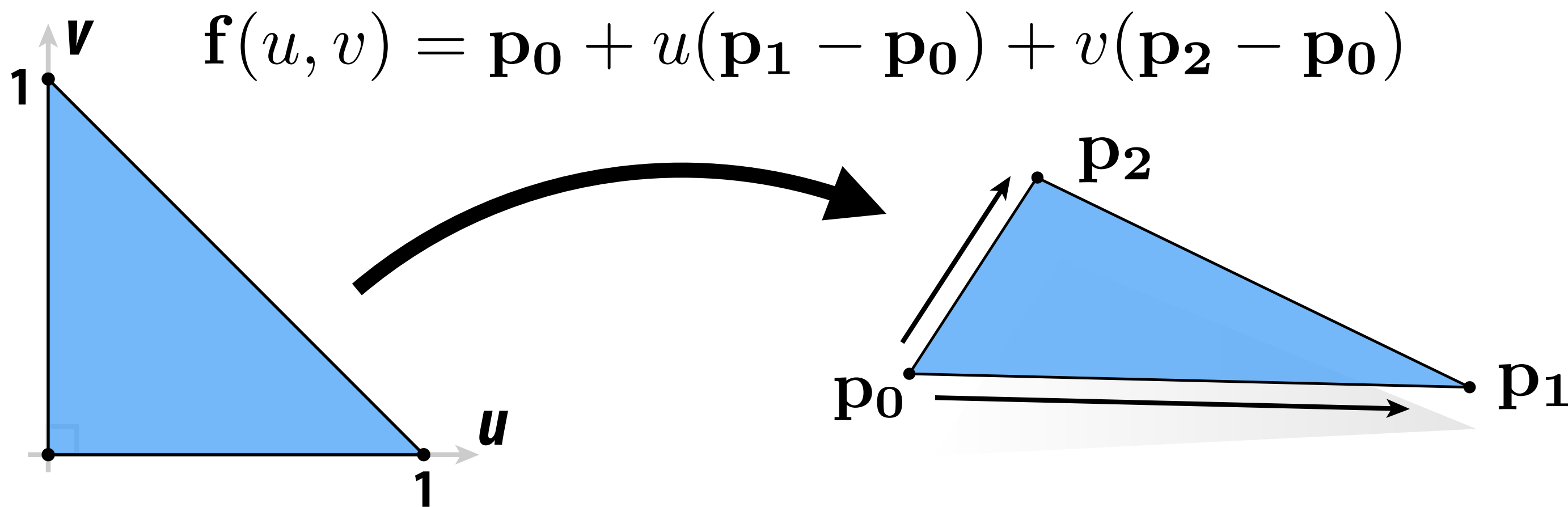
$$\mathbf{f}(u, v) = \mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0})$$
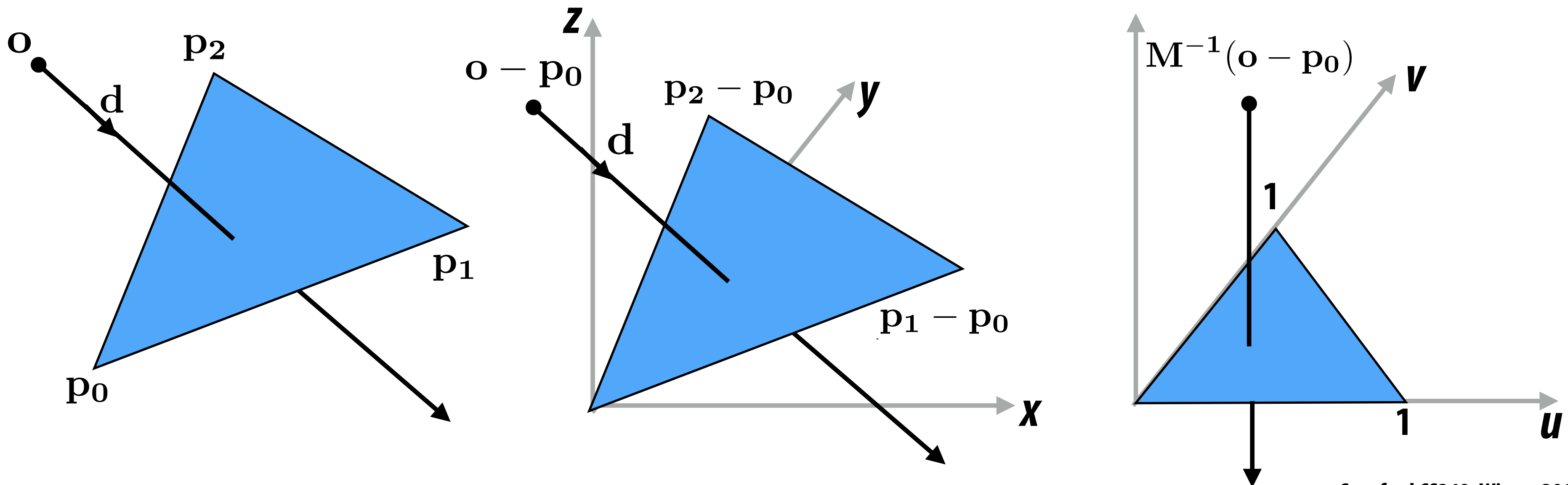
Another way: ray-triangle intersection

Plug parametric ray equation directly into equation for points on triangle:

$$\mathbf{p_0} + u(\mathbf{p_1} - \mathbf{p_0}) + v(\mathbf{p_2} - \mathbf{p_0}) = \mathbf{o} + t\mathbf{d}$$
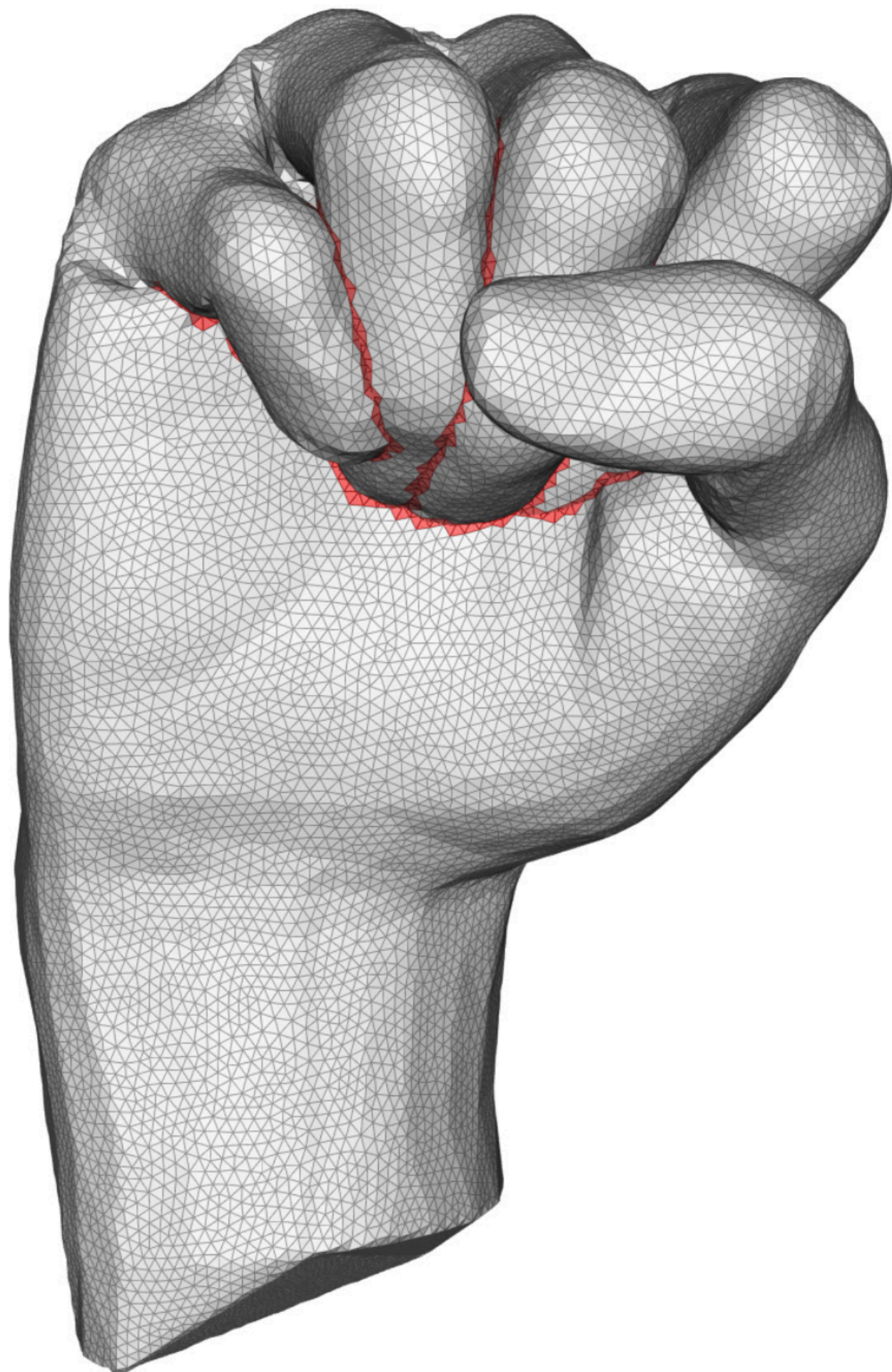
Solve for u, v, t:

$$\underbrace{\begin{bmatrix} \mathbf{p_1} - \mathbf{p_0} & \mathbf{p_2} - \mathbf{p_0} & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p_0}$$

$\mathbf{M^{-1}}$ **transforms triangle back to unit triangle in u,v plane, and transforms ray's direction to be orthogonal to plane. It's a point in 2D triangle test now!**

One more query: mesh-mesh intersection

- **GEOMETRY: How do we know if a mesh intersects itself?**

- **ANIMATION: How do we know if a collision occurred?**

Warm up: point-point intersection

- **Q: How do we know if p intersects a?**

- **A: …check if they're the same point!**

(p1, p2)

●

● **(a1, a2)**

Slightly harder: point-line intersection

- **Q: How do we know if a point intersects a given line?**

- **A: ...plug it into the line equation!**

p

$N^T x = c$

Line-line intersection

- **Two lines: ax=b and cx=d**

- **Q: How do we find the intersection?**

- **A: See if there is a simultaneous solution**

- **Leads to linear system:** $\begin{bmatrix} a_1 & a_2 \\ c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$

Degenerate line-line intersection?

- **What if lines are almost parallel?**

- **Small change in normal can lead to big change in intersection!**

- **Instability very common, very important with geometric predicates. Demands special care (e.g., analysis of matrix).**

See for example Shewchuk, "*Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*"

Triangle-triangle intersection?

- **Lots of ways to do it**
- **Basic idea:**
  - **Q: Any ideas?**
  - **One way: reduce to edge-triangle intersection**
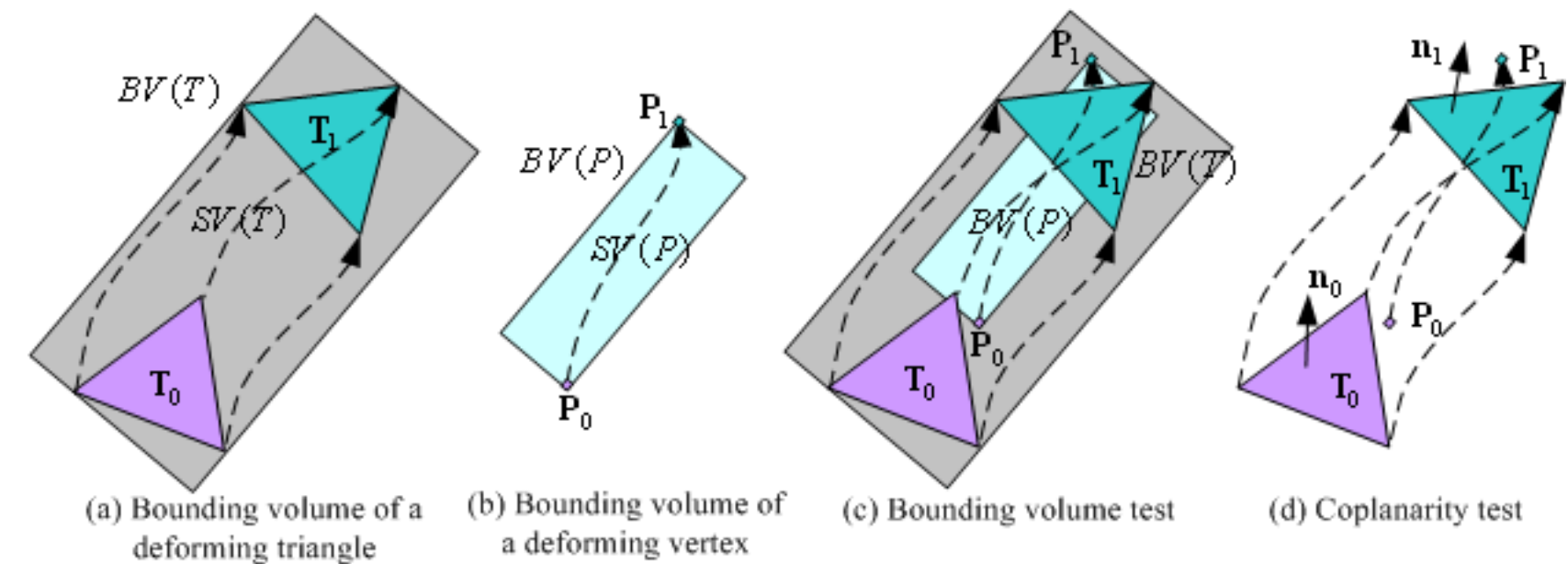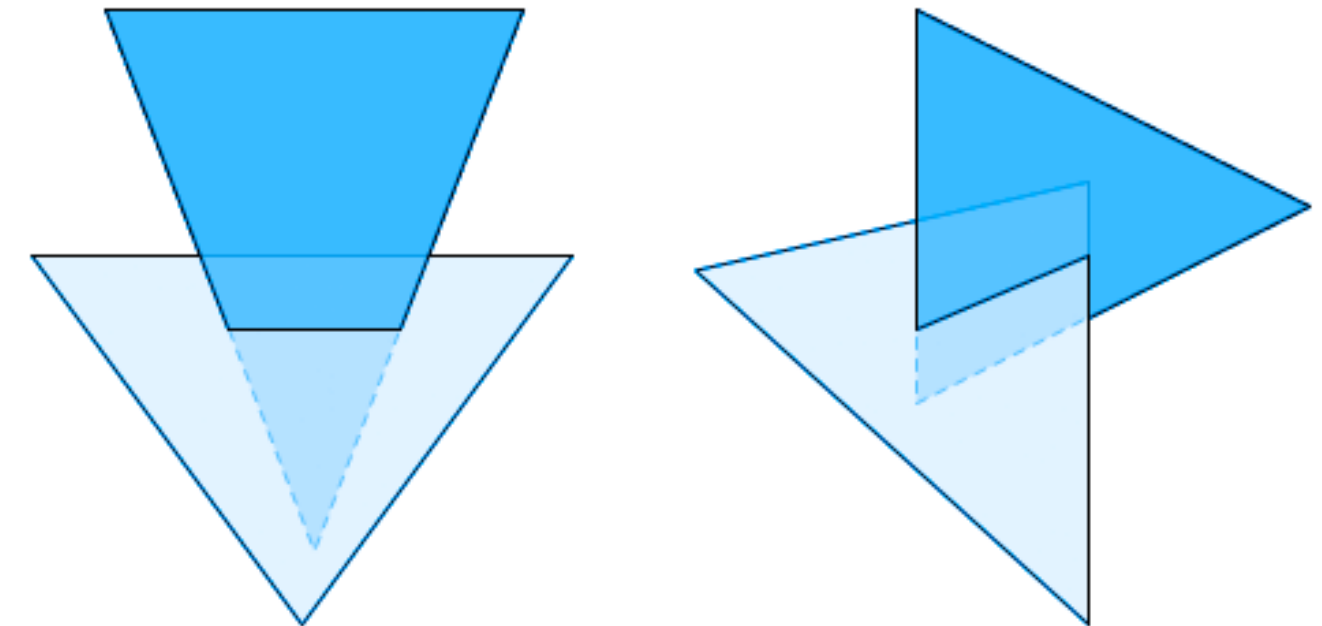 - **Check if each line passes through plane (ray-triangle)**
  - **Then do interval test**
- **What if triangle is *moving*?**
  - **Important case for animation**
  - **Can think of triangles as *prisms* in time**
  - **Turns dynamic problem (in nD + time) into purely geometric problem in (n+1)-dimensions**

(a) Bounding volume of a deforming triangle (b) Bounding volume of a deforming vertex (c) Bounding volume test (d) Coplanarity test

Ray-scene intersection

Given a scene defined by a set of *N* primitives and a ray *r*, find the closest point of intersection of *r* with the scene
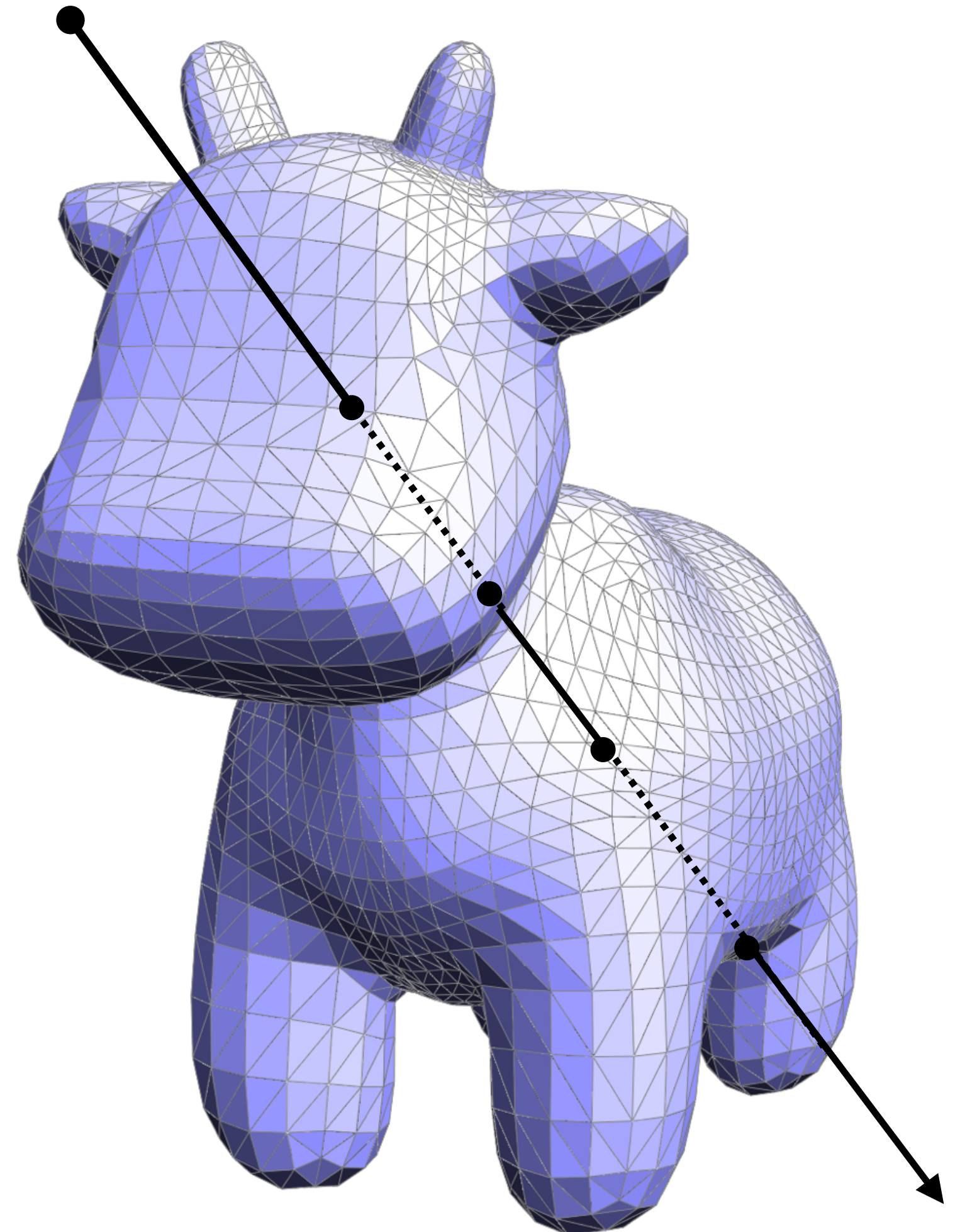
"Find the first primitive the ray hits"

```
p_closest = NULL
t_closest = inf
for each primitive p in scene:
    t = p.intersect(r)
    if t >= 0 && t < t_closest:
        t_closest = t
        p_closest = p
```

(Assume p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r*)

Complexity? $O(N)$

Can we do better? Of course… but you'll have to wait until next class

Rendering via ray casting:
(one common use of ray-scene intersection tests)

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

Recall triangle visibility:

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

Sample

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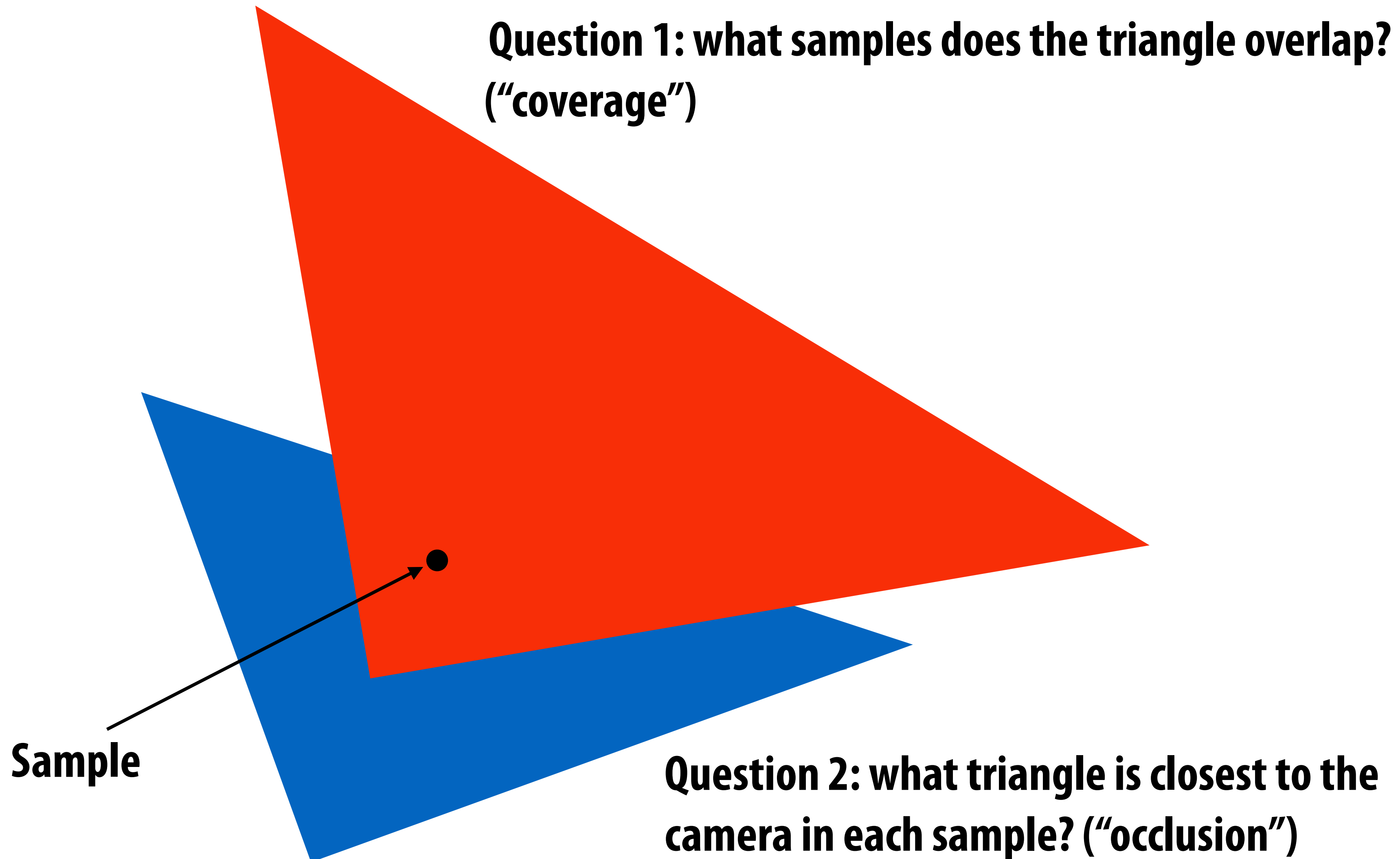
The visibility problem

- **What scene geometry is visible at each screen sample?**
  - What scene geometry *projects* onto screen sample points? (coverage)
  - Which geometry is visible from the camera at each sample? (occlusion)

x-axis

x/z

Pinhole
Camera
(0,0)

Virtual
Sensor

-z axis

(x,z)
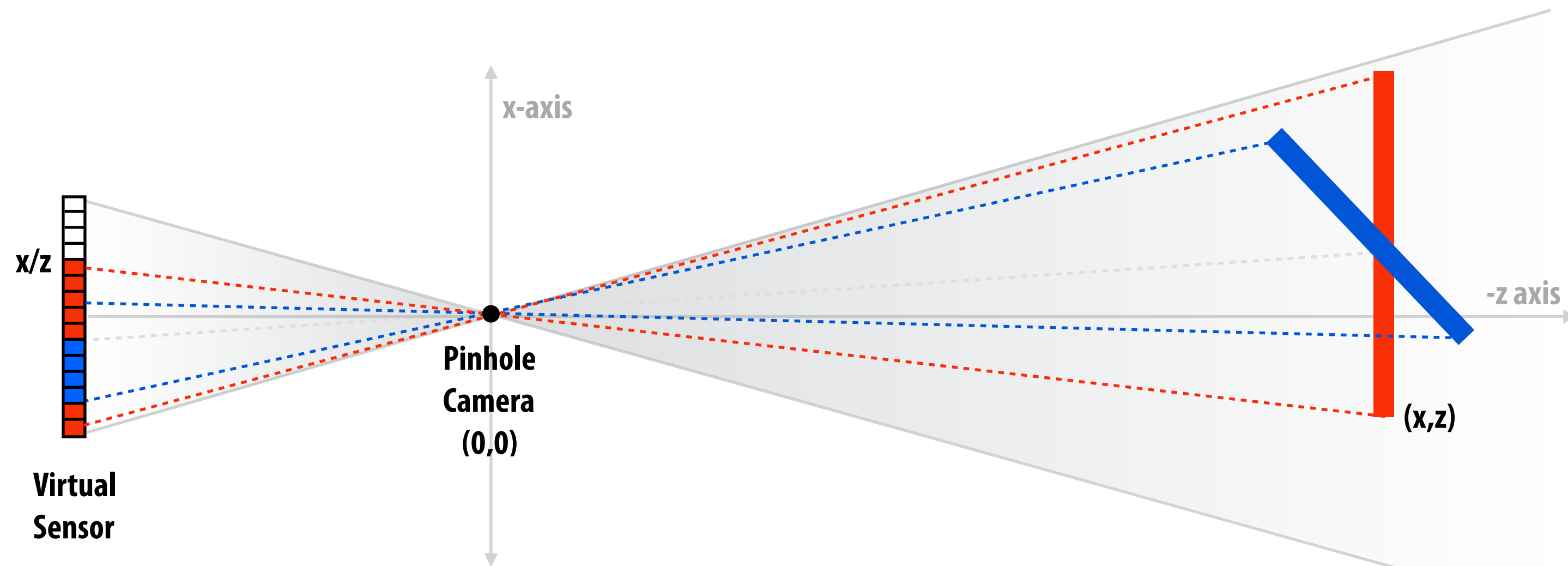
Basic rasterization algorithm

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]
```
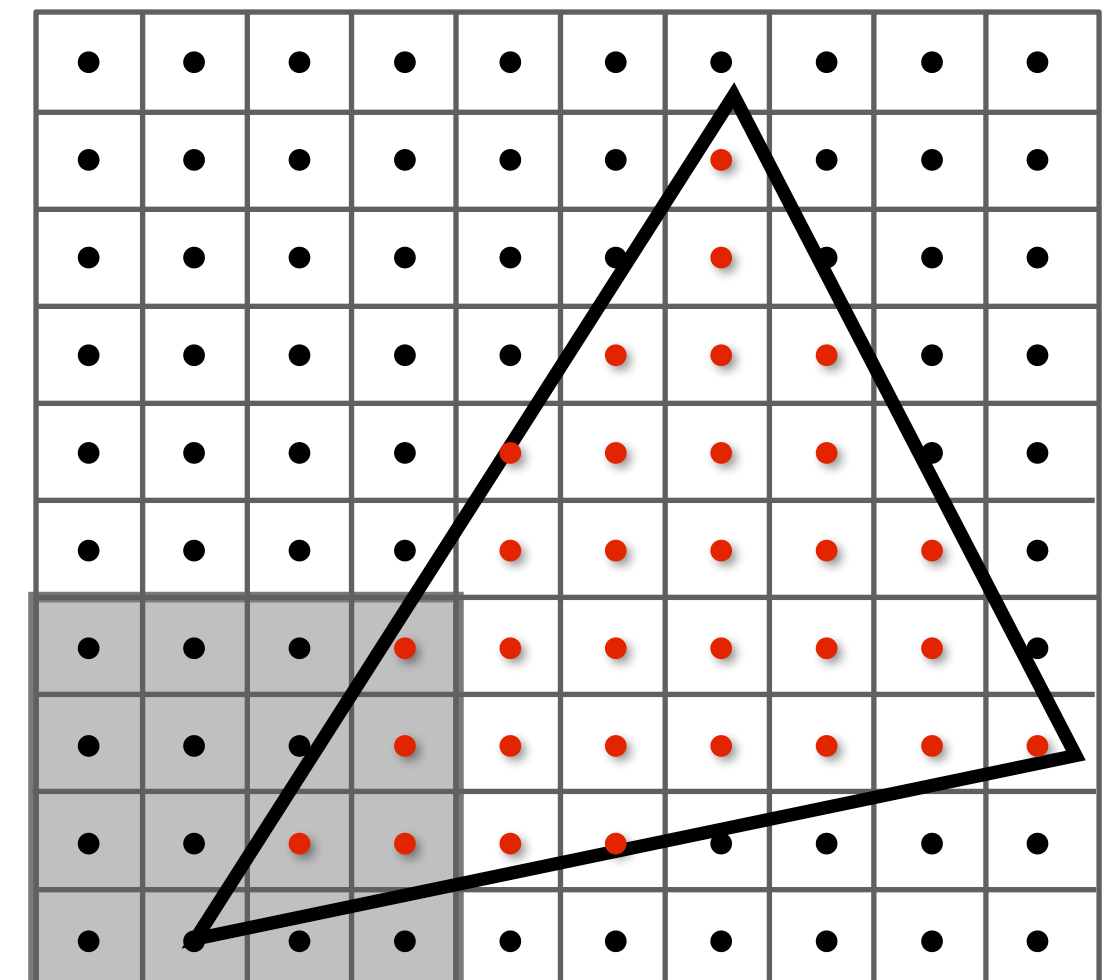
"Given a triangle, __find__ the samples it covers"
**(finding the samples is relatively easy since they are
distributed uniformly on screen)**

**More efficient __hierarchical__ rasterization:**

For each TILE of image

If triangle overlaps tile, check all samples in tile

The visibility problem (described differently)

- **In terms of casting rays from the camera:**

  - Is a scene primitive 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)
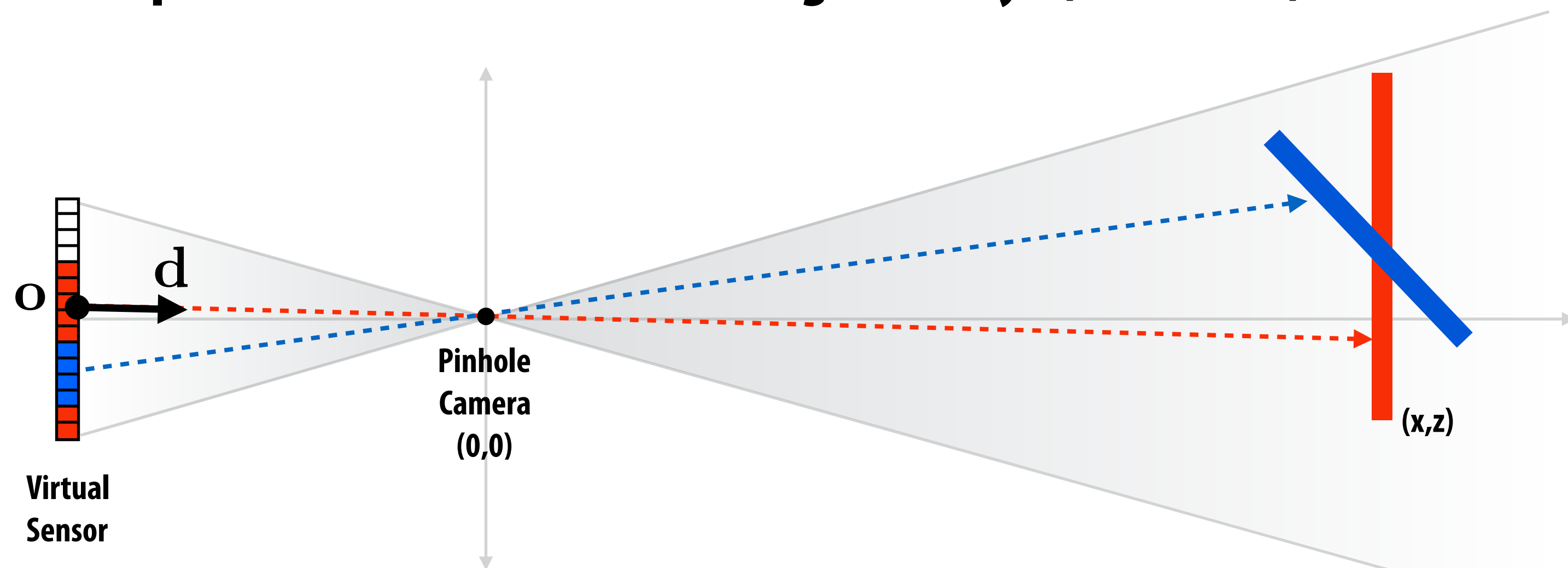
Basic ray casting algorithm

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: 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."

# Basic rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order (for all triangles)
  - Store entire depth buffer (requires access to 2D array of fixed size)
  - Do not have to store entire scene geometry in memory
    - Naturally supports unbounded size scenes

- **Ray casting:**
  - Proceeds in screen sample order (for all rays)
    - Do not have to store closest depth so far for the entire screen (just the current ray)
    - This is the natural order for rendering transparent surfaces (process surfaces in the order the are encountered along the ray: front-to-back)
  - Must store entire scene geometry for fast access

In other words…

- **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 uniform distribution in angle… angle between rays is smaller away from view direction)**
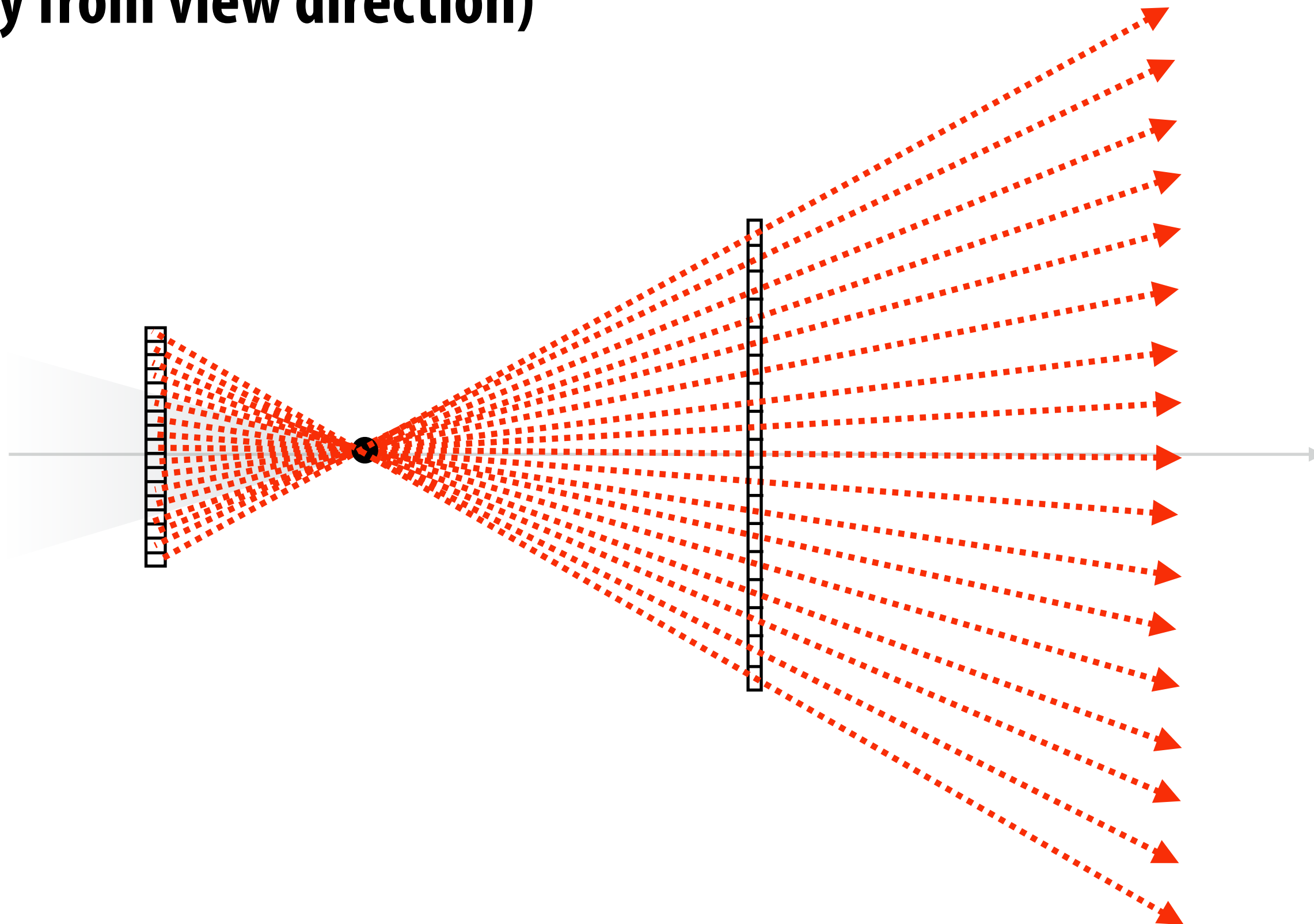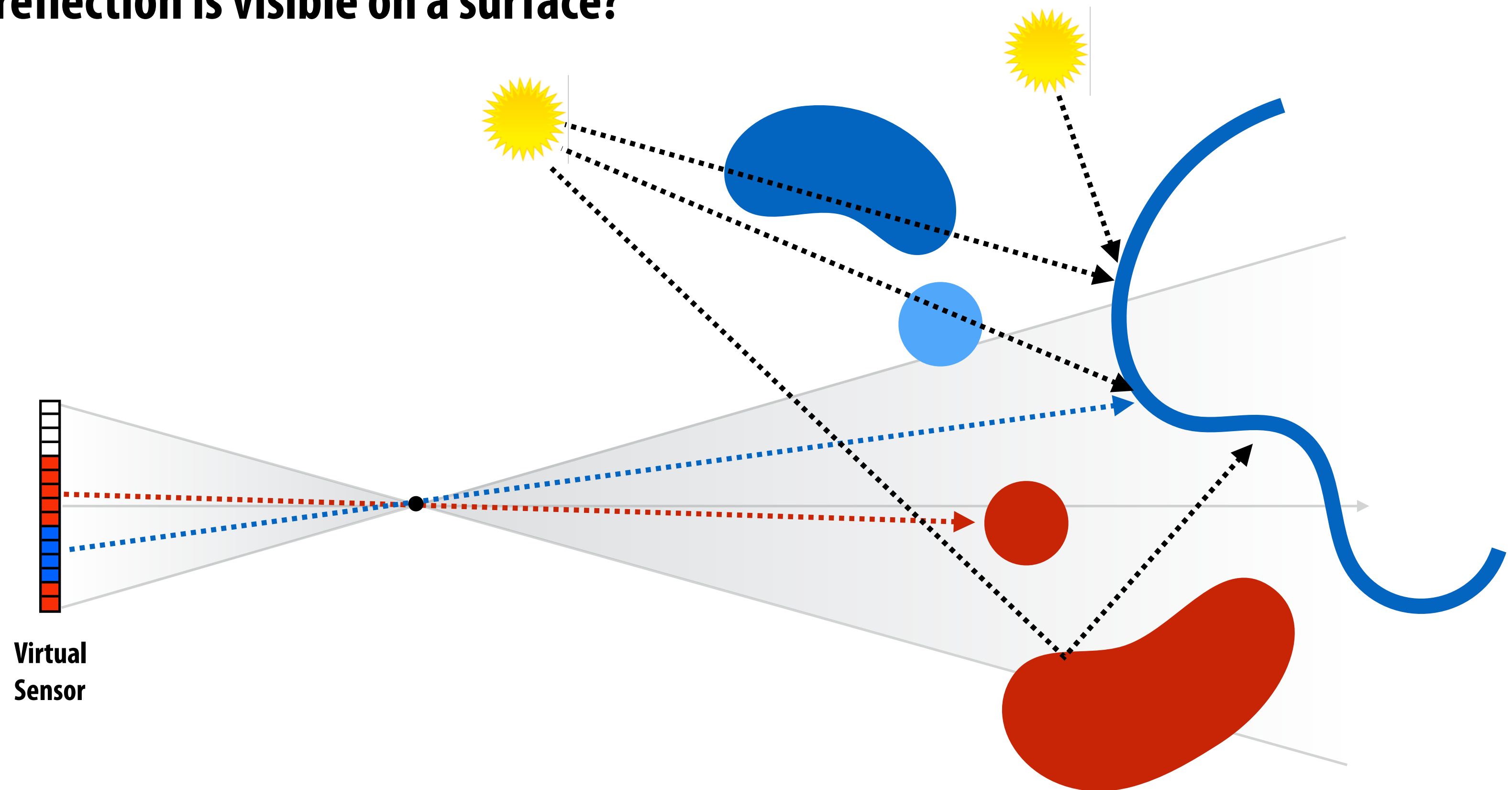
# Generality of ray-scene queries

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?
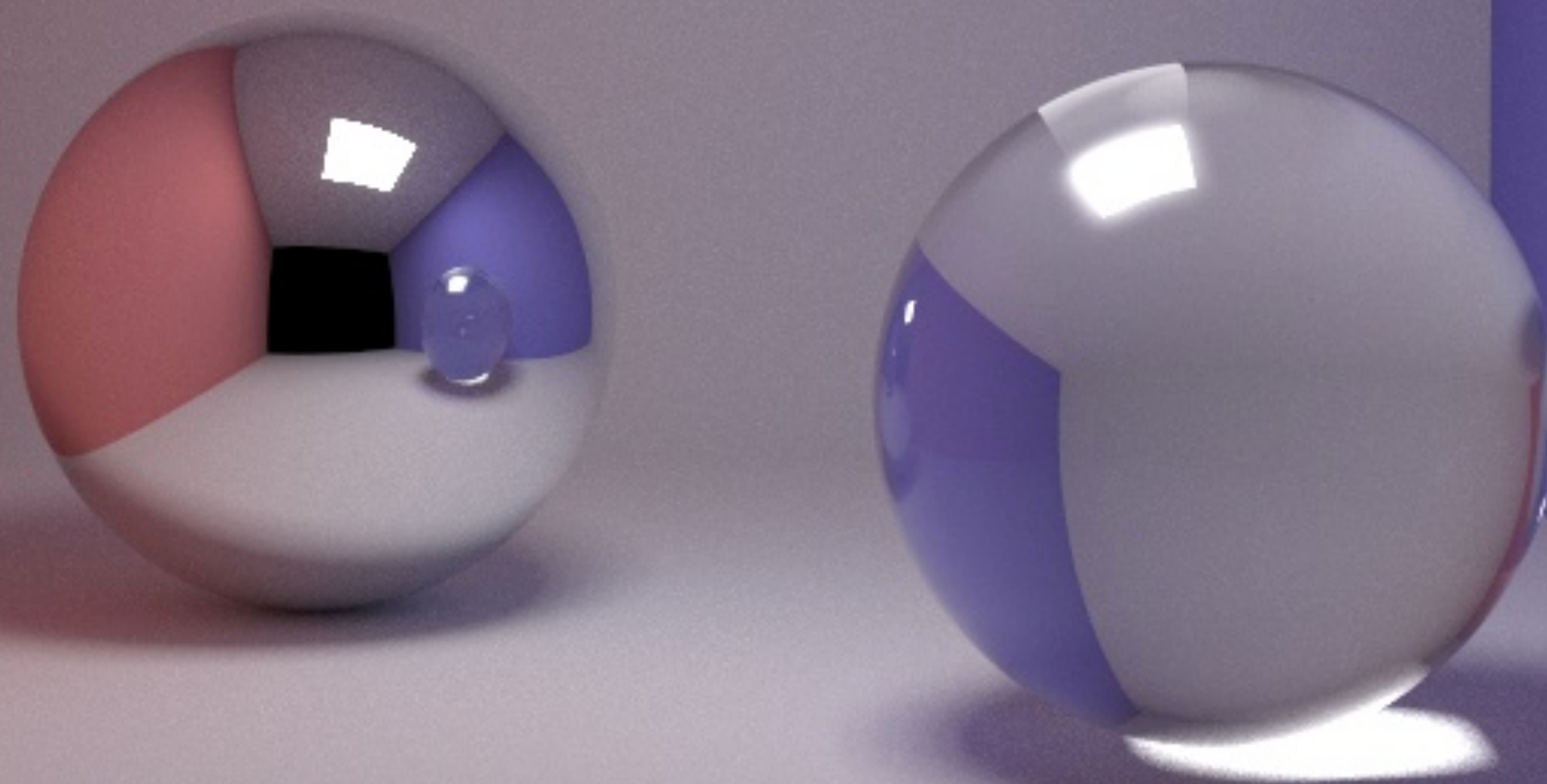

Virtual Sensor

In contrast, rasterization is a highly-specialized solution for computing visibility for a set of uniformly distributed rays originating from the same point (most often: the camera)

Direct illumination + reflection + transparency

HENRIK WANN JENSEN 1999
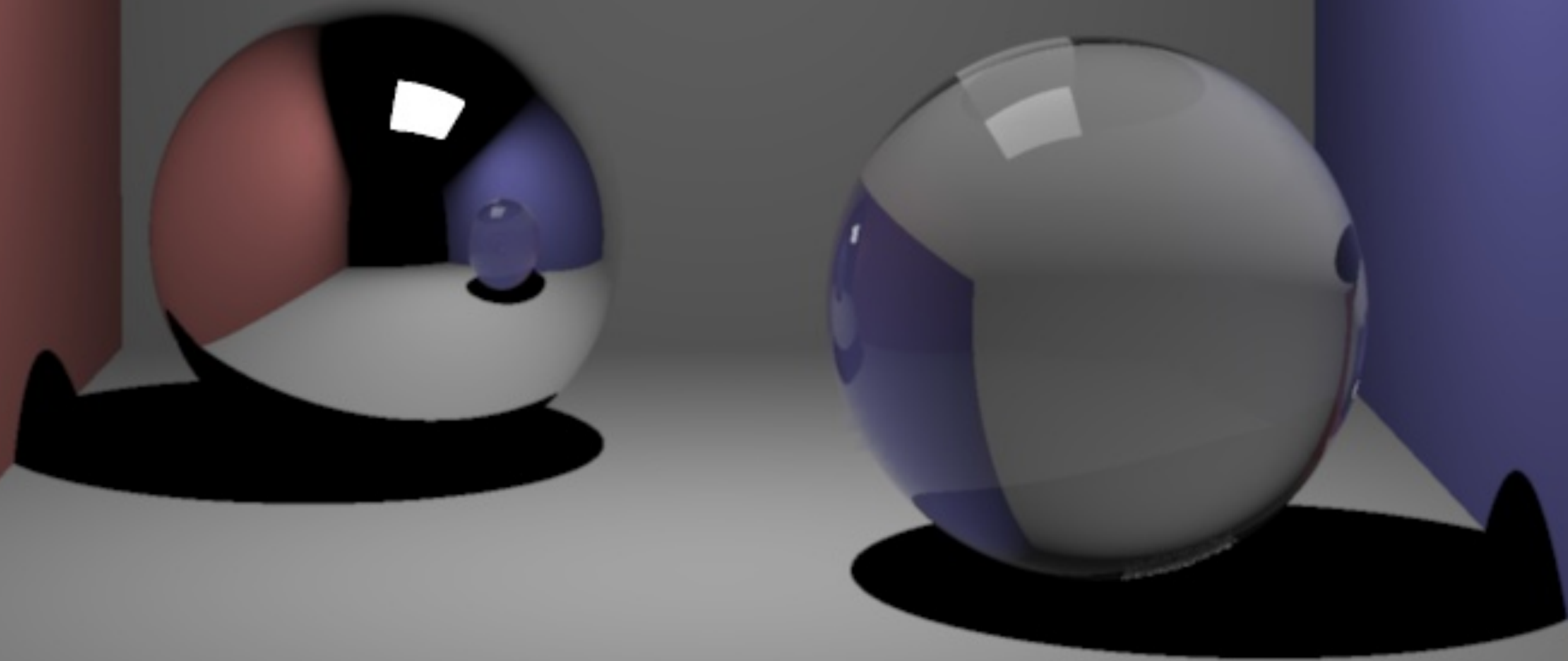
Global illumination solution

Image credit: Henrik Wann Jensen

Sixteen-bounce global illumination

Shadows

How to compute if a surface point is in shadow?

Assume you have an algorithm for ray-scene intersection…

L_1

L_2

P

x
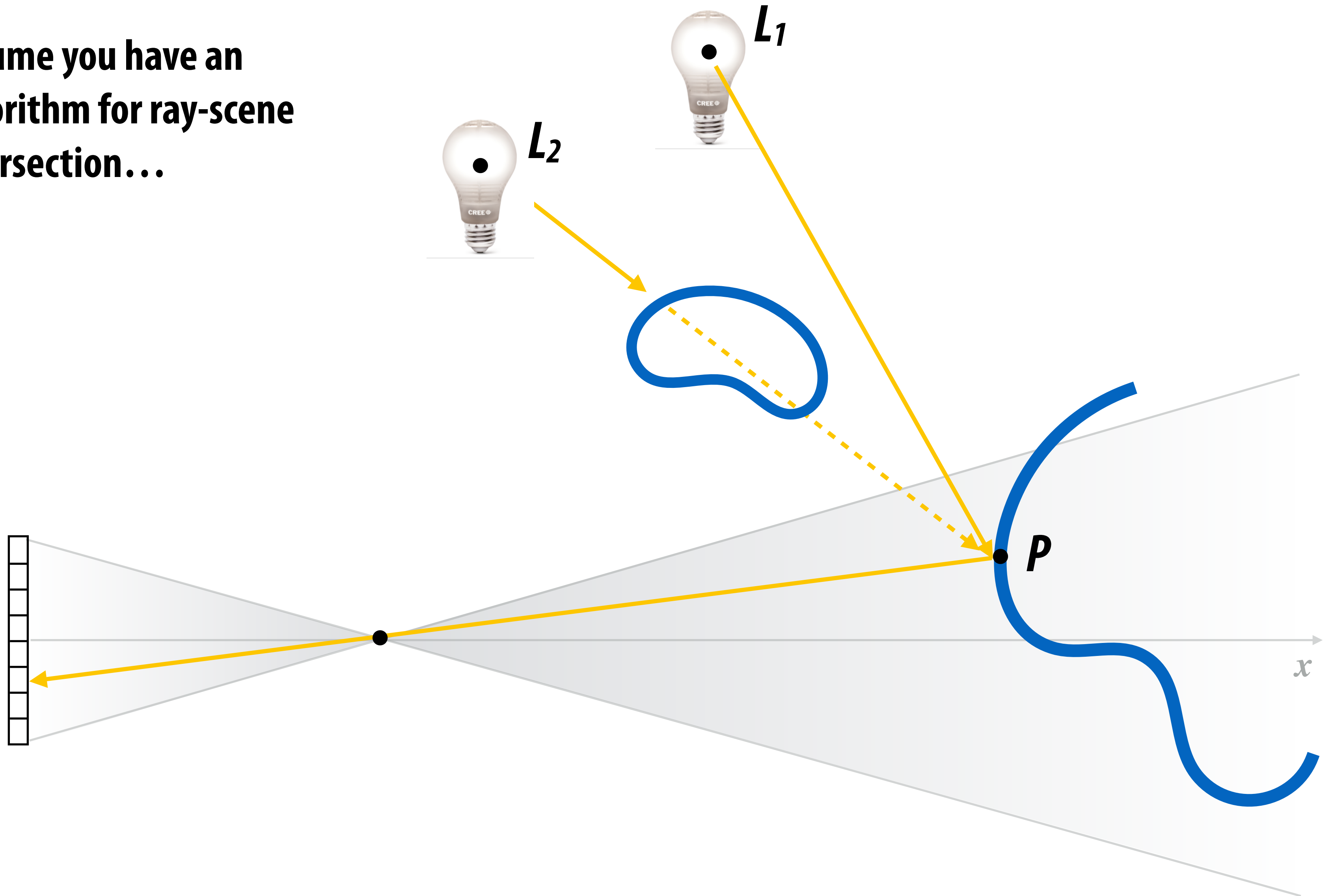
A simple shadow computation algorithm

- 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

L_1

L_2

P

x
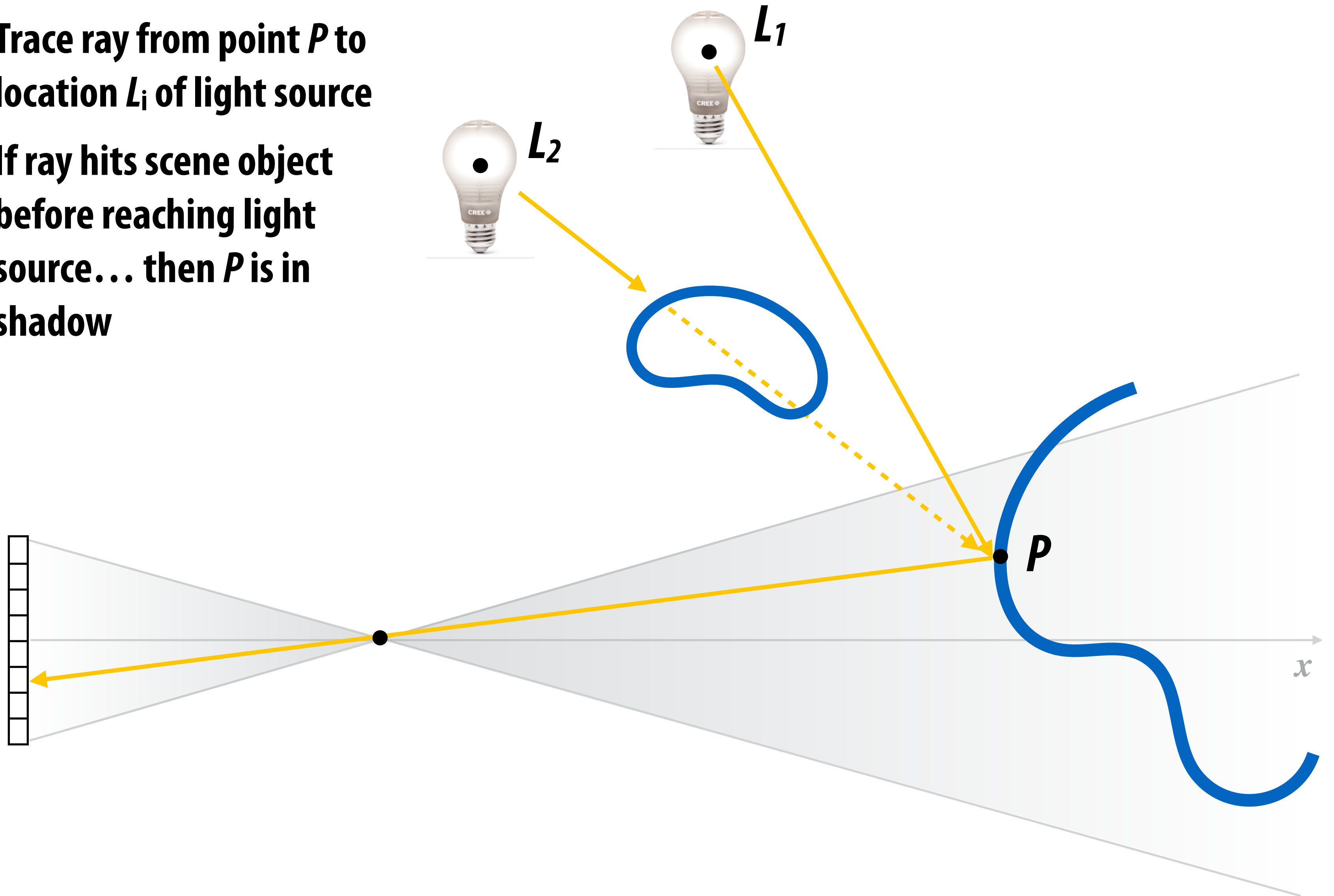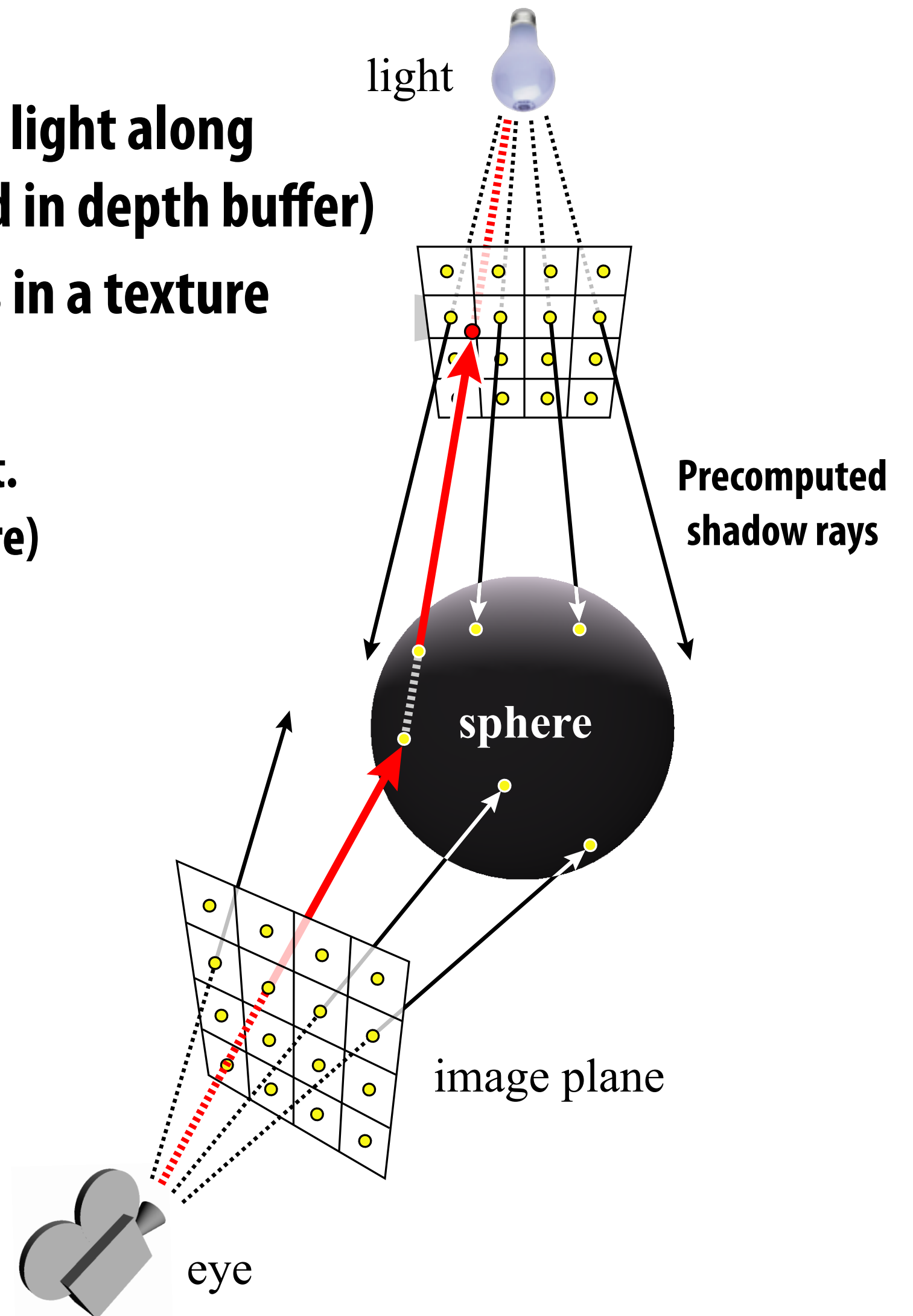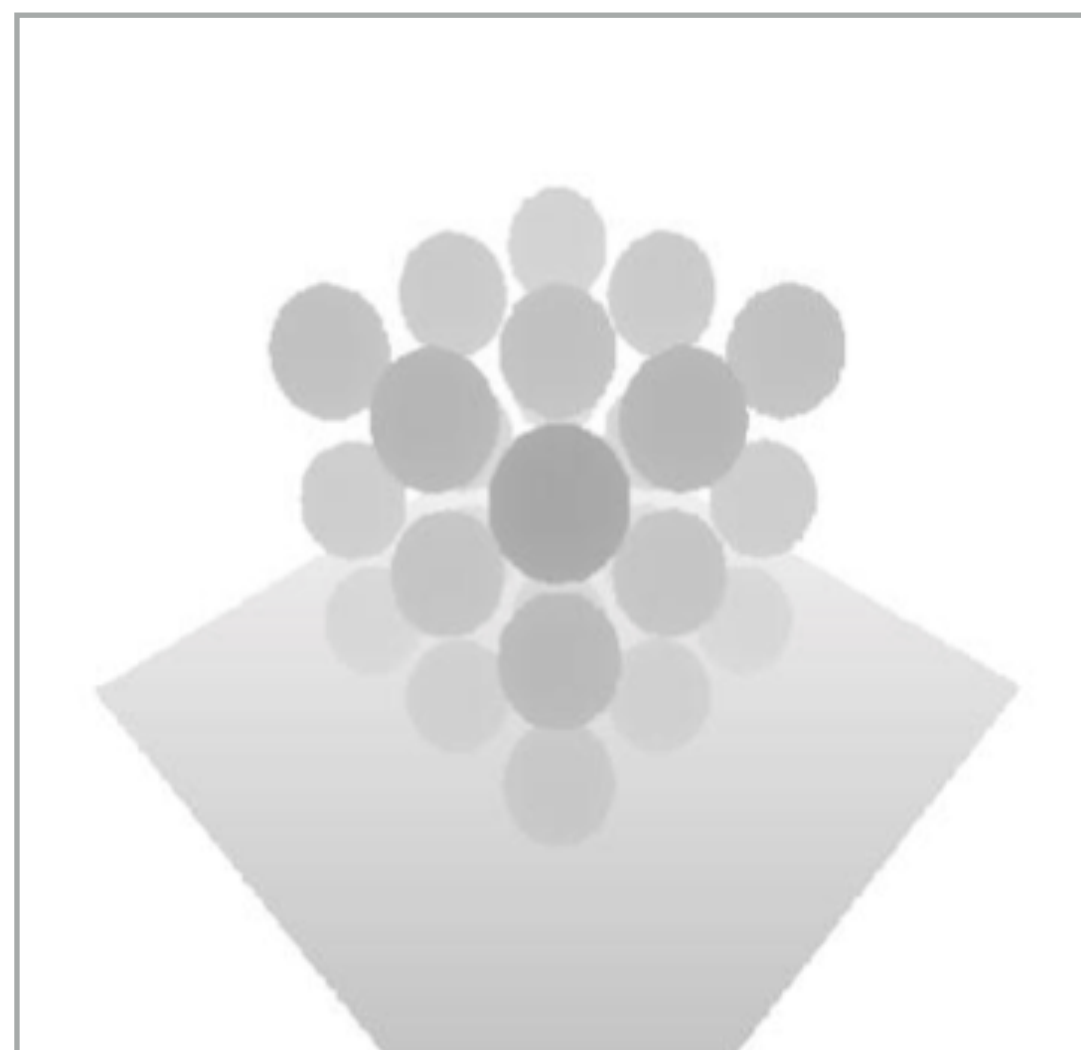
Recall rasterization / ray casting relationship

- **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 uniform distribution in angle… angle between rays is smaller away from view direction)
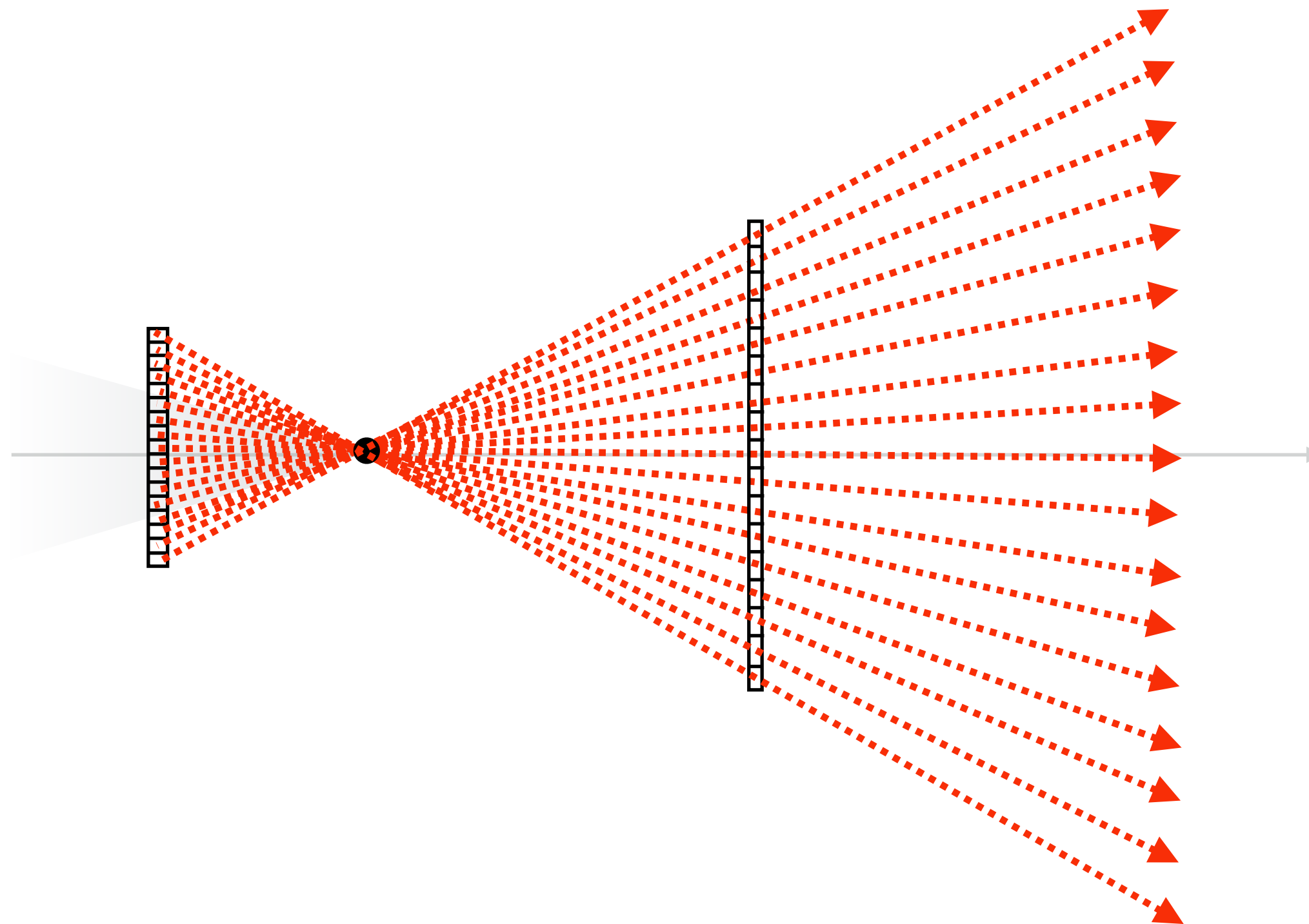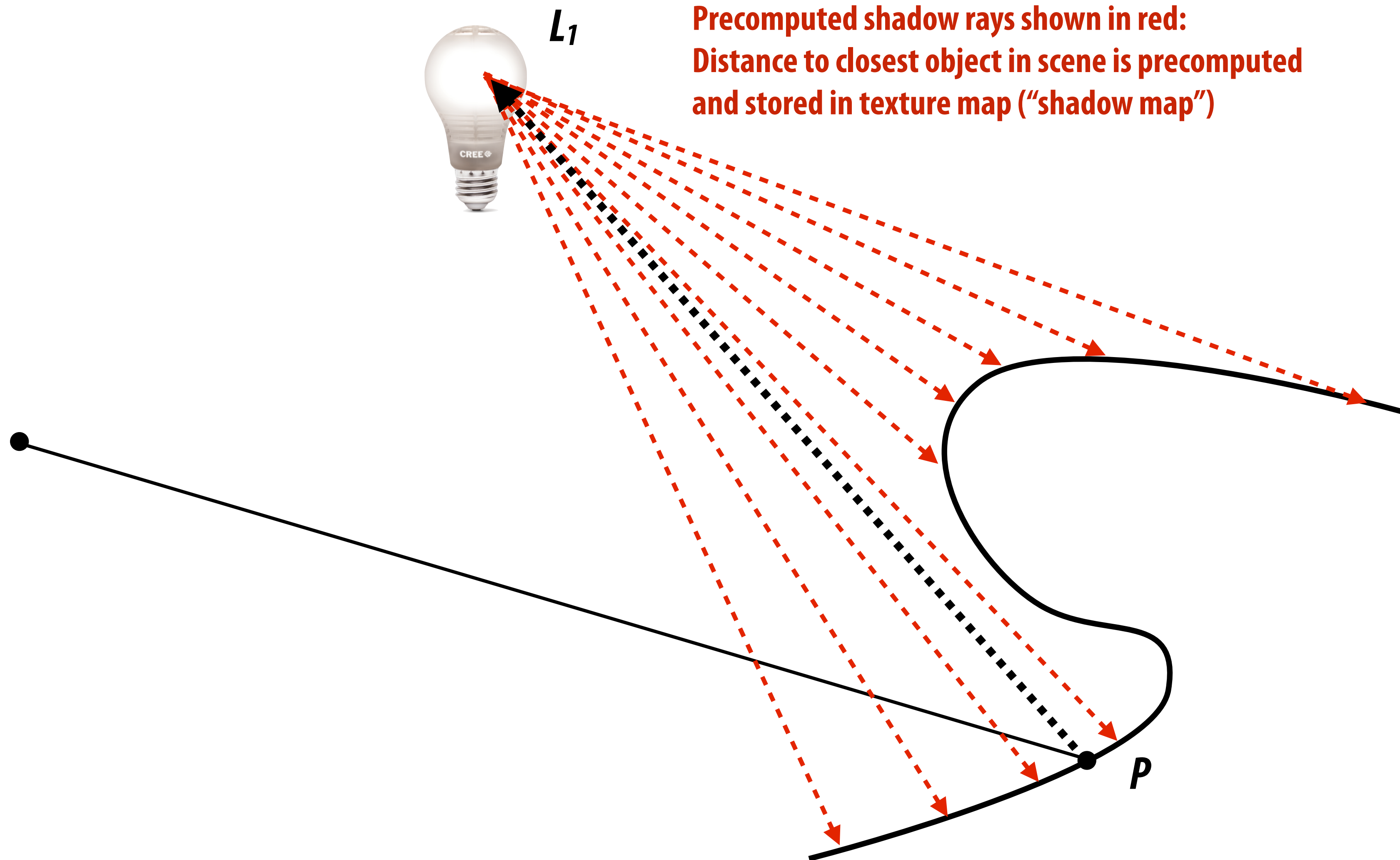
Shadow mapping: ray origin for rasterization need not be the scene's camera position [Williams 78]

1. **Place camera at position of a point light source**

2. **Render scene to compute depth to closest object to light along uniformly distributed "shadow rays" (answer stored in depth buffer)**

3. **Store precomputed shadow ray intersection results in a texture**

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

light

Precomputed shadow rays

sphere

image plane

eye

Shadow texture lookup approximates visibility result when shading fragment at *P*

L₁

Precomputed shadow rays shown in red:
Distance to closest object in scene is precomputed
and stored in texture map ("shadow map")

P

# Shadow mapping pseudocode

## (this logic would be implemented in fragment shader)

- **Given world-space point P_{world}, light position (L), *and* light direction (D)**

- **Transform P into "light space", defined by light position at origin and -Z aligned with D**

- **Project transformed P into P_{proj}**

- **Lookup value in shadow map at (P_{proj}.x, P_{proj}.y)**

- **If value from shadow map is less than |L-P|, then point P is in shadow**

light

actual

P_{proj}

desired

P_{world}

image plane

eye
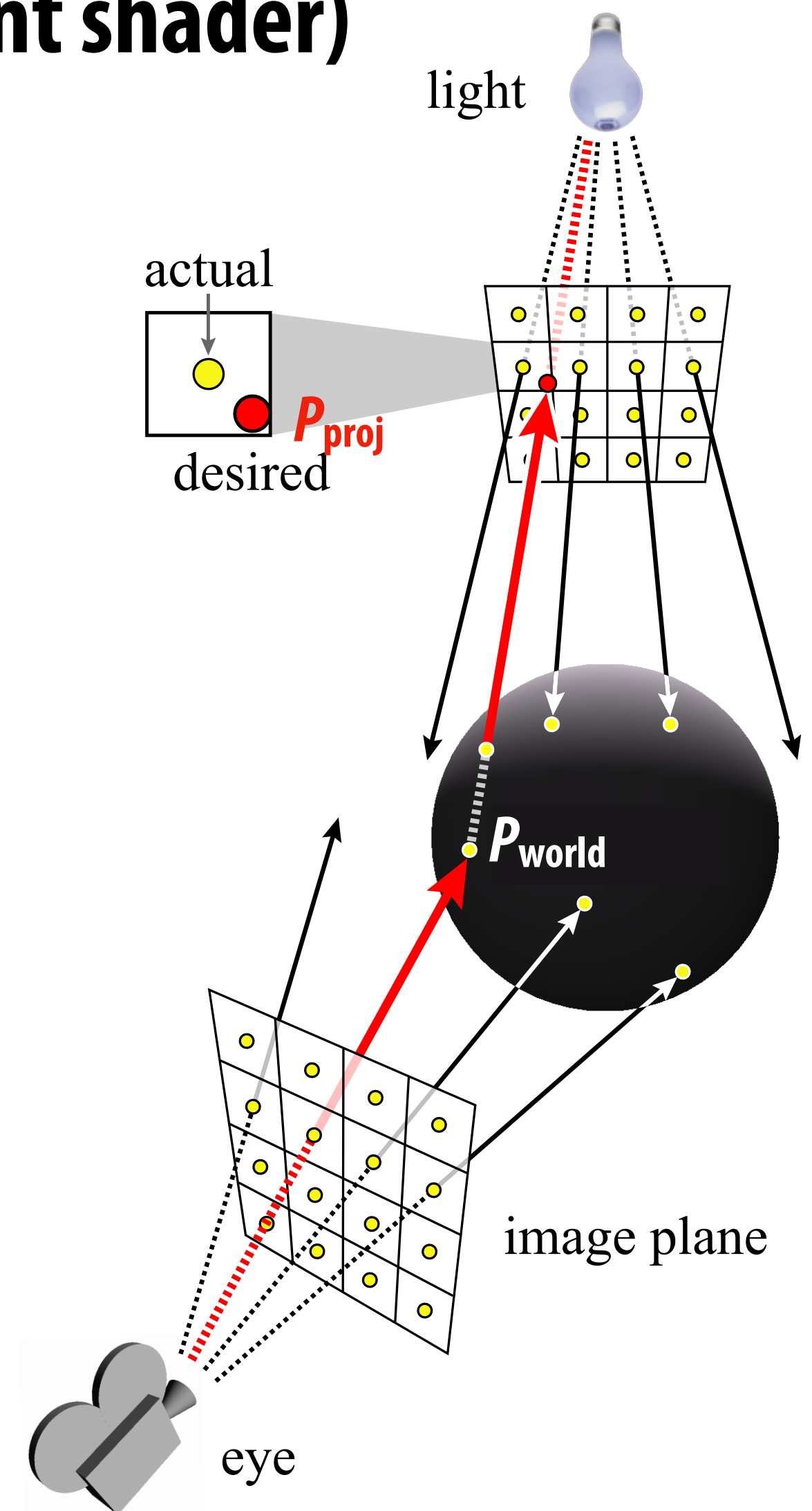
Shadow aliasing due to shadow map undersampling
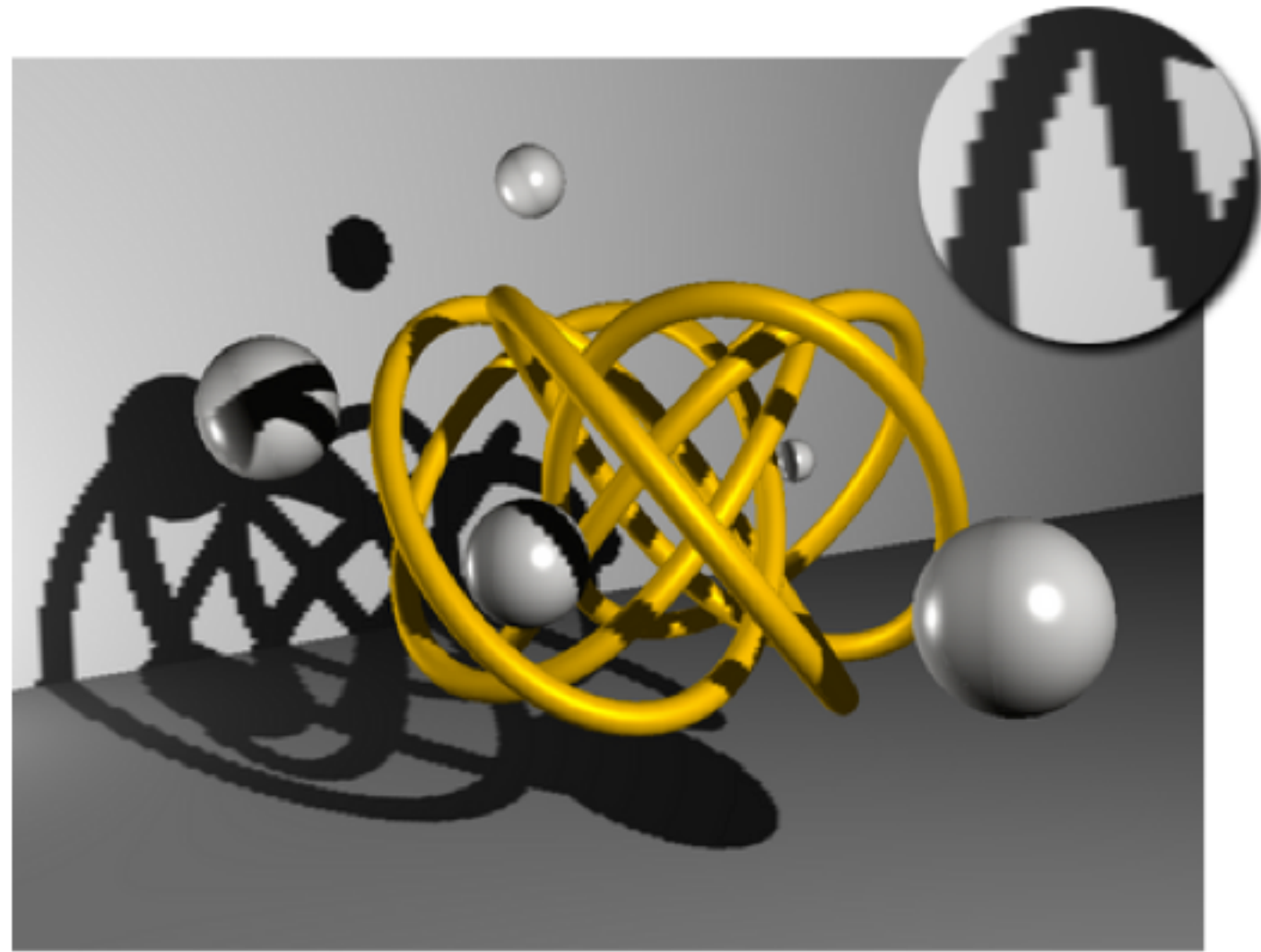
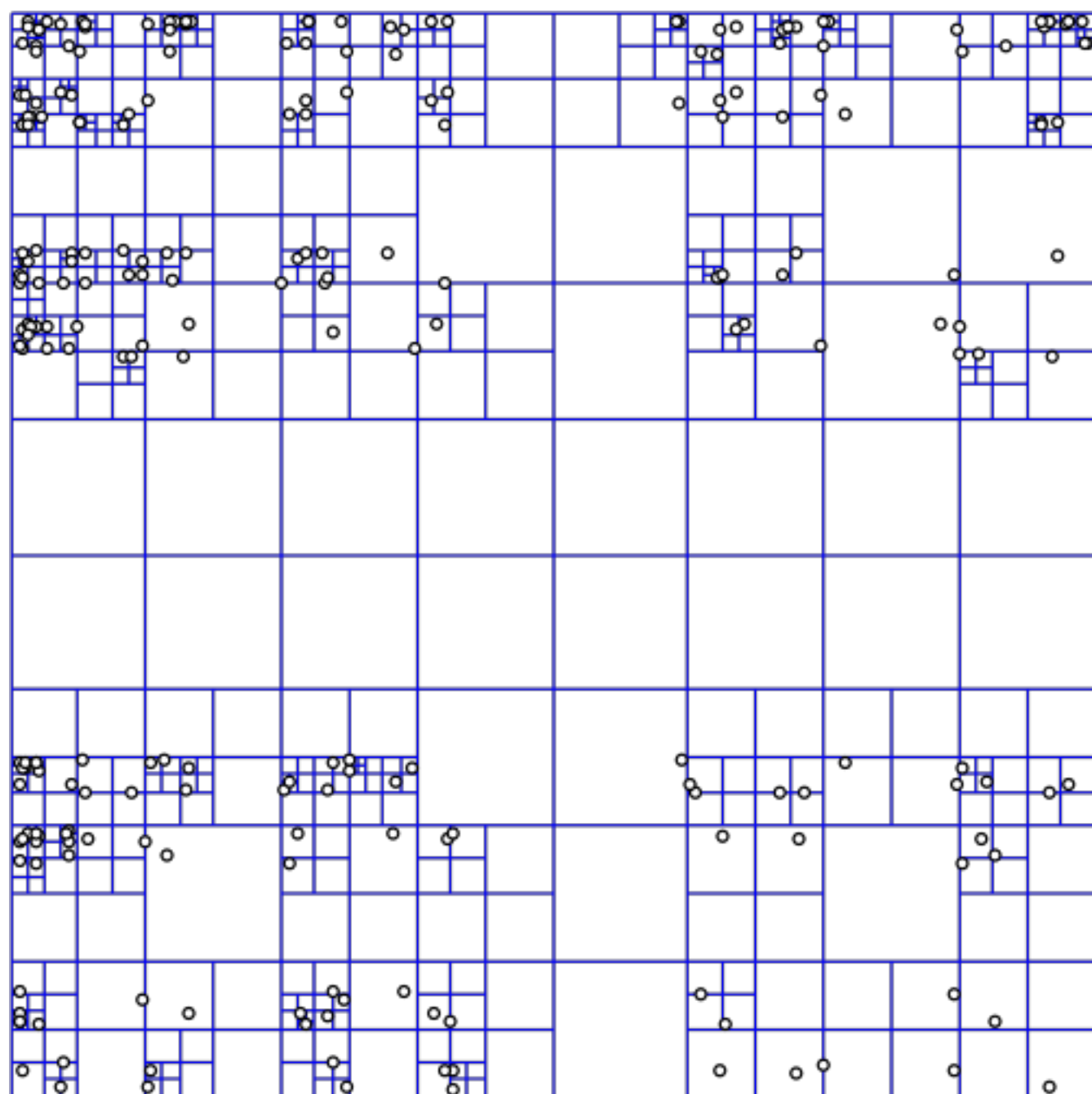

Shadows computed using shadow map
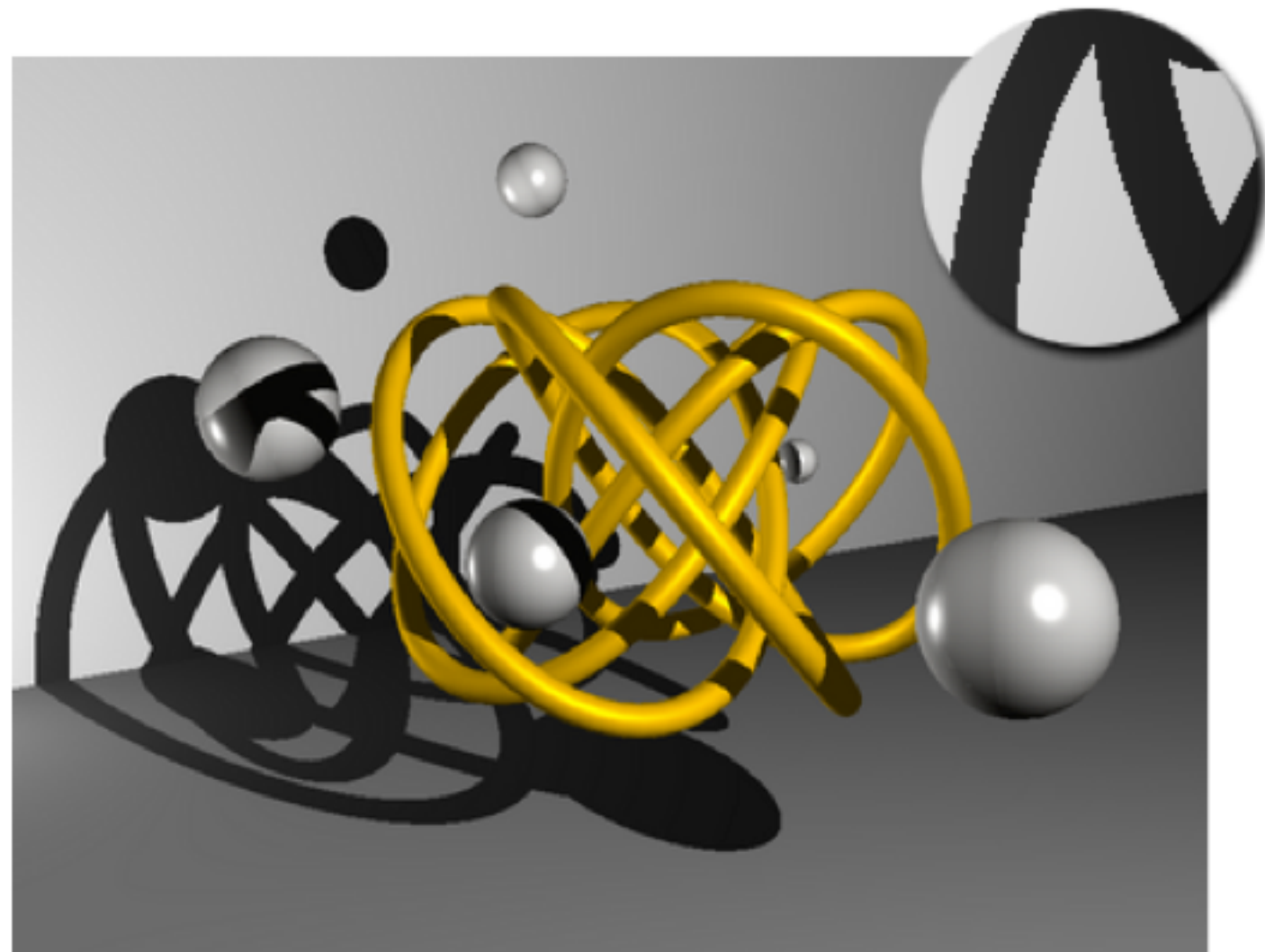
Correct hard shadows
(result from computing shadow directly using ray tracing)

Next time: spatial acceleration data structures

- **Testing every primitive in scene to find ray-scene intersection is *slow!***

- **Consider linearly scanning through a list vs. binary search**
  - **can apply this same kind of thinking to geometric queries**
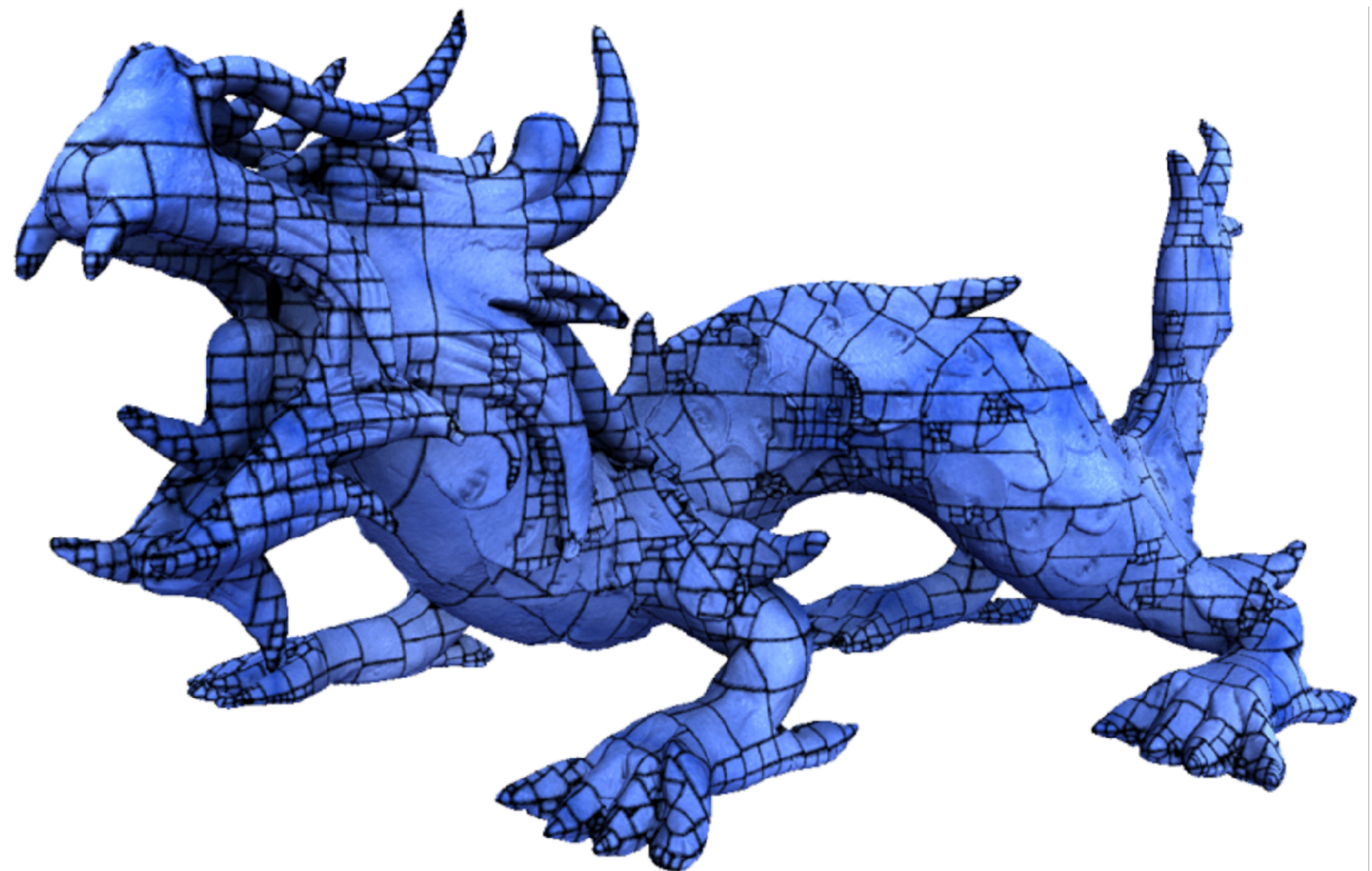
Acknowledgements

- **Thanks to Keenan Crane for presentation resources**