

Lecture 8:

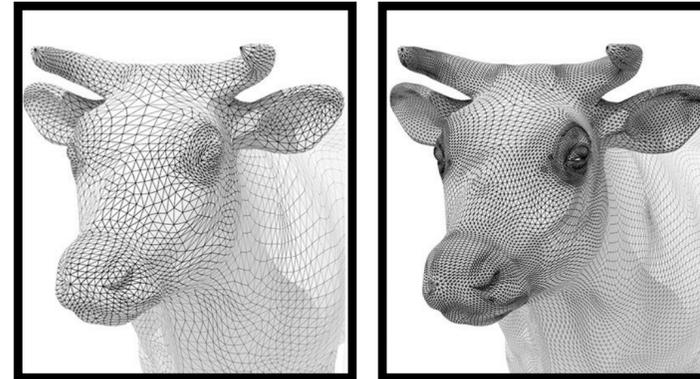
Geometric Queries

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2023

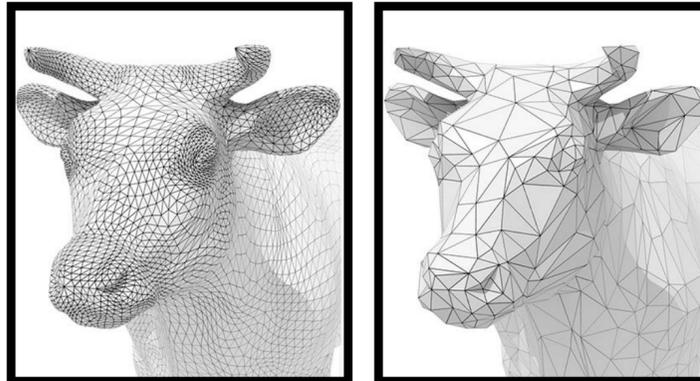
Last time

How to perform a number of basic mesh processing operations

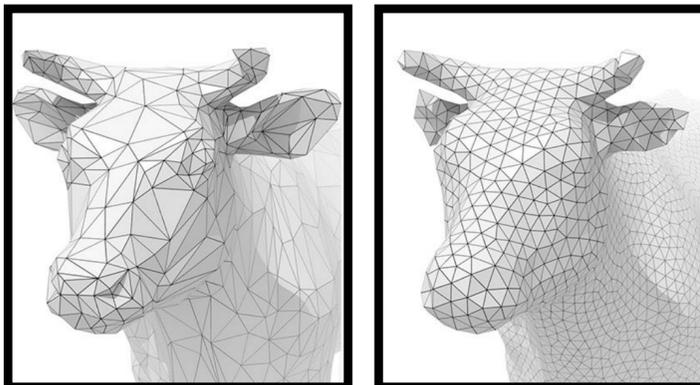
- **Subdivision (upsampling)**



- **Mesh simplification (downsampling)**



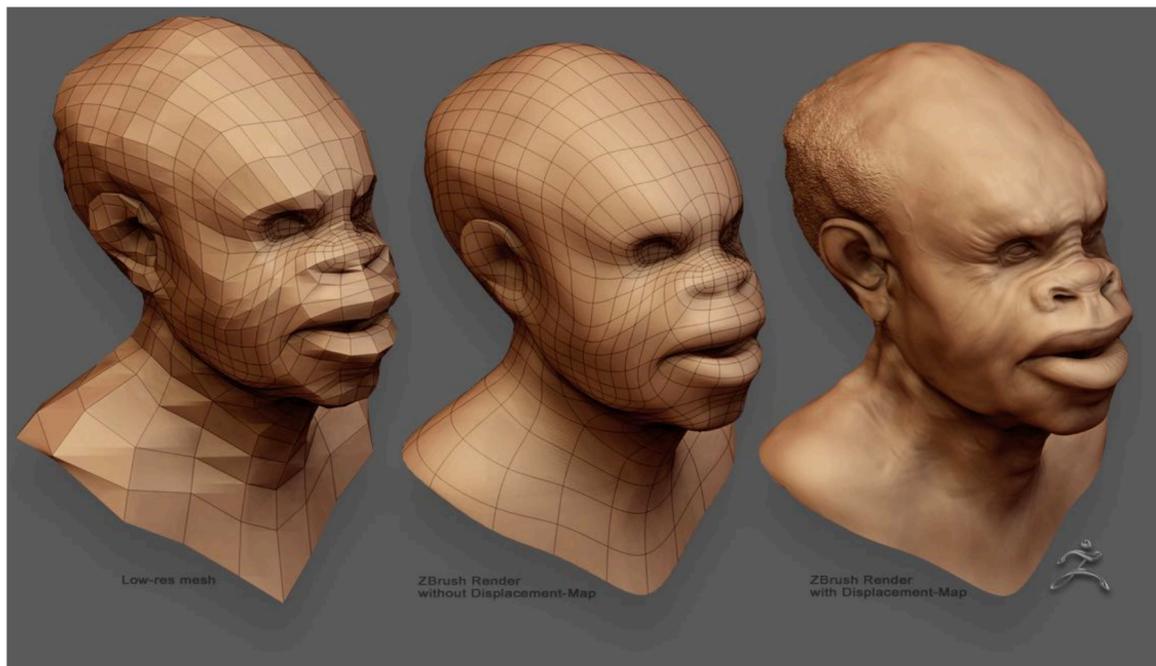
- **Mesh resampling**



Geometric queries — motivation



Intersecting rays and triangles (ray tracing)



Closest point on surface queries

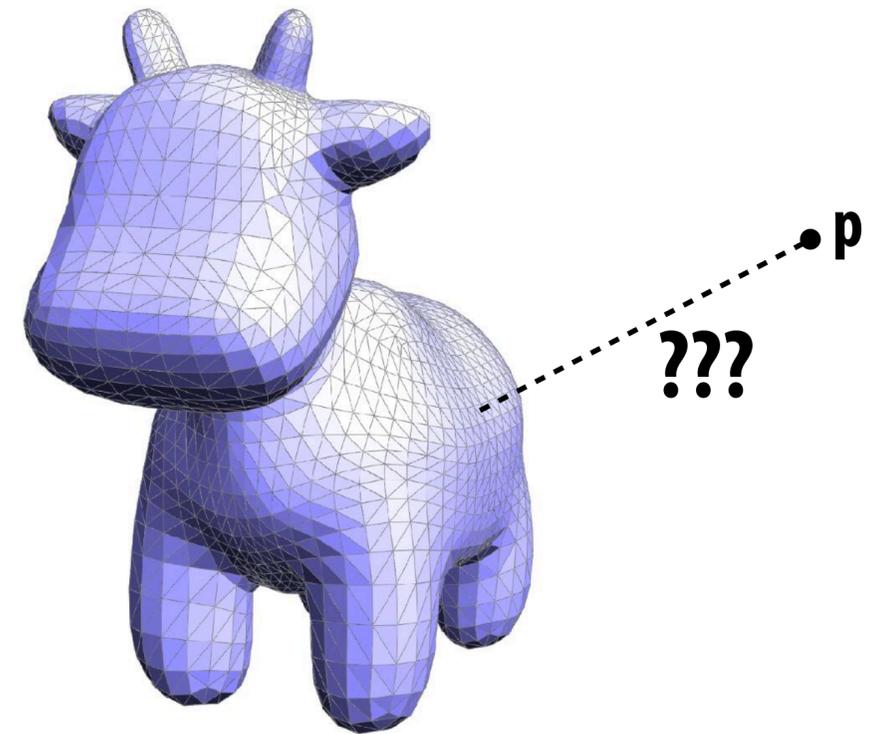


Intersecting triangles (collisions)

Closest point queries

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

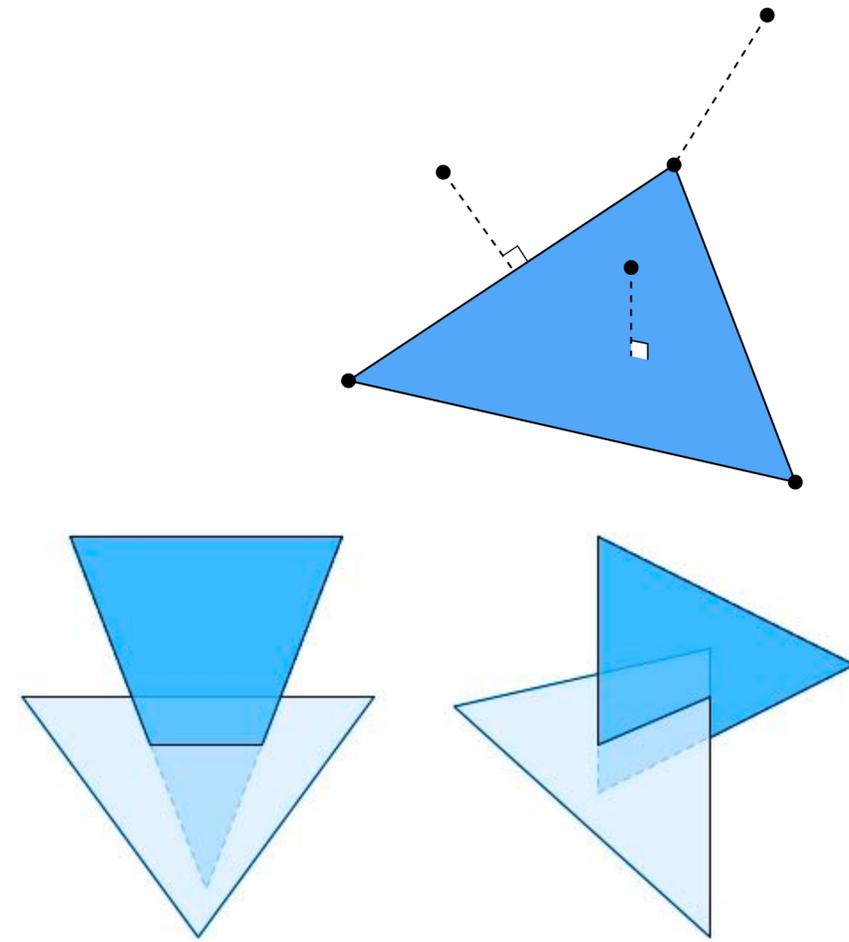
- **Q: Does implicit/explicit representation make this easier?**
- **Q: Does our half-edge data structure help?**
- **Q: What's the cost of the naïve algorithm?**
- **Q: How do we find the distance to a single triangle anyway?**



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...

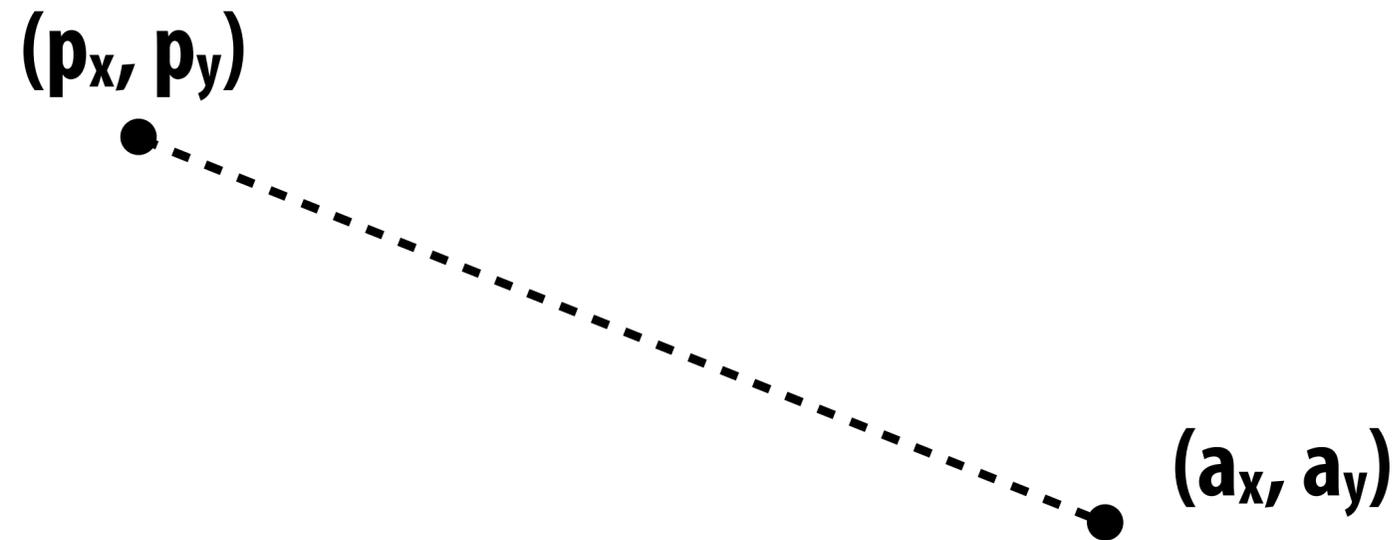
- Need some new ideas!

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

- NEXT TIME: intelligent ways to accelerate geometric queries

Warm up: closest point on point

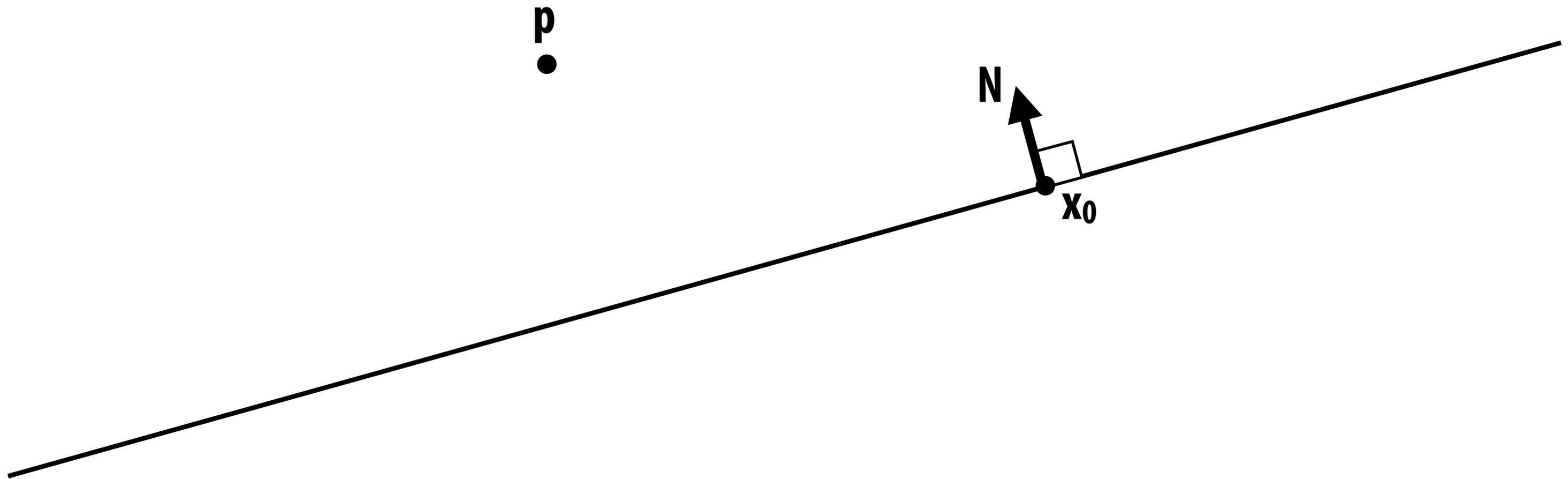
Given a query point (p_x, p_y) , how do we find the closest point on the point (a_x, a_y) ?



Bonus question: what's the distance?

Slightly harder: closest point on line

- Now suppose I have a line $N^T x = c$, where N is the unit normal
 - Remember: a line is all points x such that $N^T x = c$
- How do I find the point on the line closest to my query point p ?



Review: matrix form of a line (and a plane)

Line is defined by:

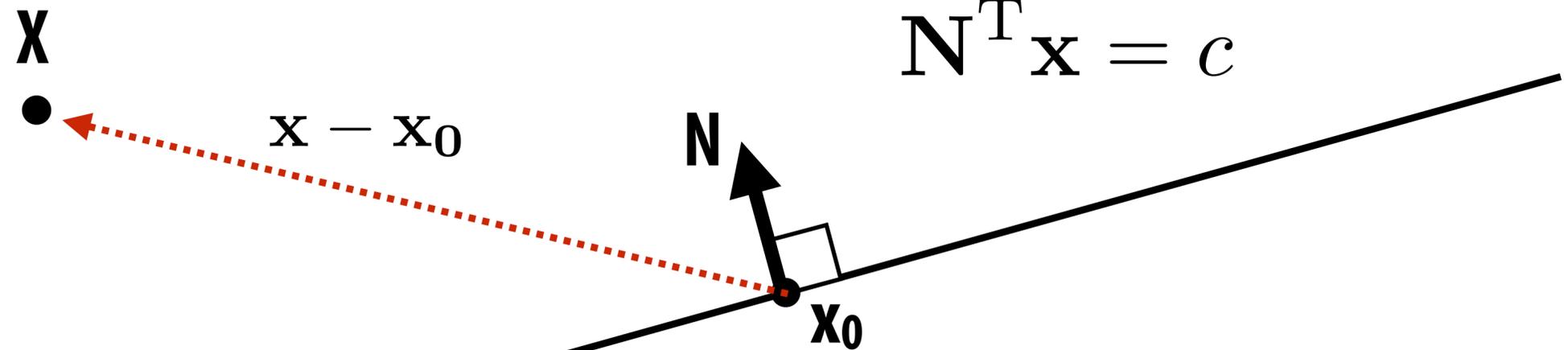
- Its normal: \mathbf{N}
- A point \mathbf{x}_0 on the line

$$\mathbf{N} \cdot (\mathbf{x} - \mathbf{x}_0) = 0$$

$$\mathbf{N}^T (\mathbf{x} - \mathbf{x}_0) = 0$$

$$\mathbf{N}^T \mathbf{x} = \mathbf{N}^T \mathbf{x}_0$$

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

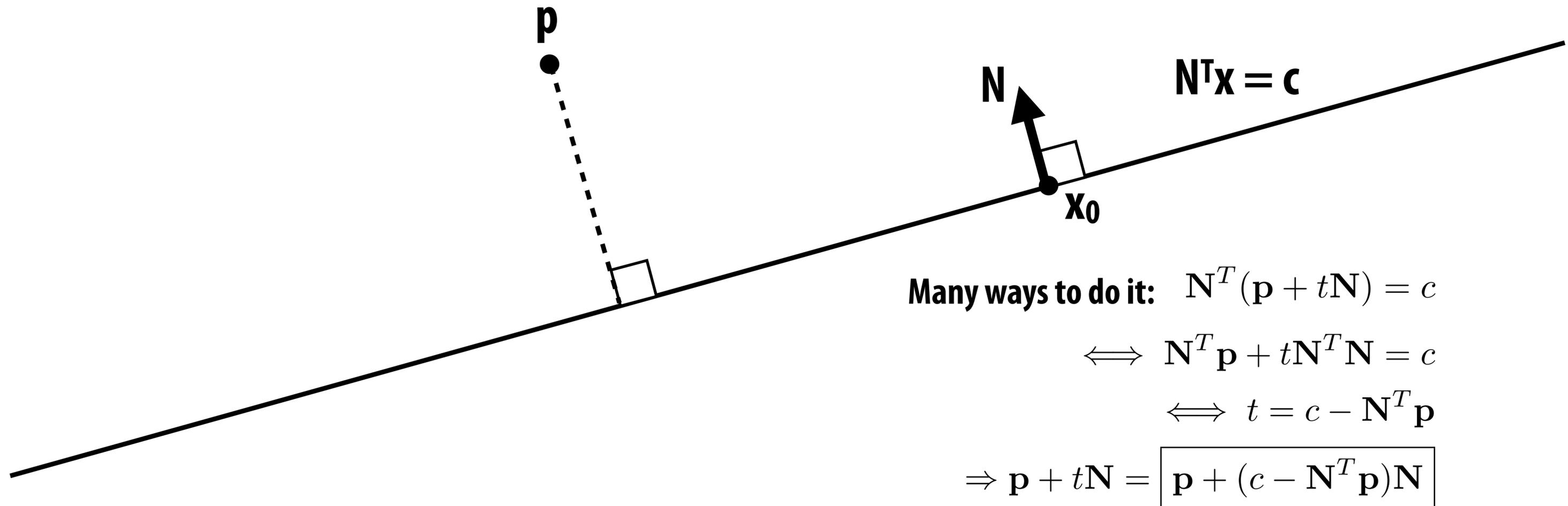


**The line (in 2D) is all points \mathbf{x} , where $\mathbf{x} - \mathbf{x}_0$ is orthogonal to \mathbf{N} .
($\mathbf{N}, \mathbf{x}, \mathbf{x}_0$ on this slide are 2-vectors)**

**(And a plane (in 3D) is all points \mathbf{x} where $\mathbf{x} - \mathbf{x}_0$ is orthogonal to \mathbf{N} .)
($\mathbf{N}, \mathbf{x}, \mathbf{x}_0$ are 3-vectors)**

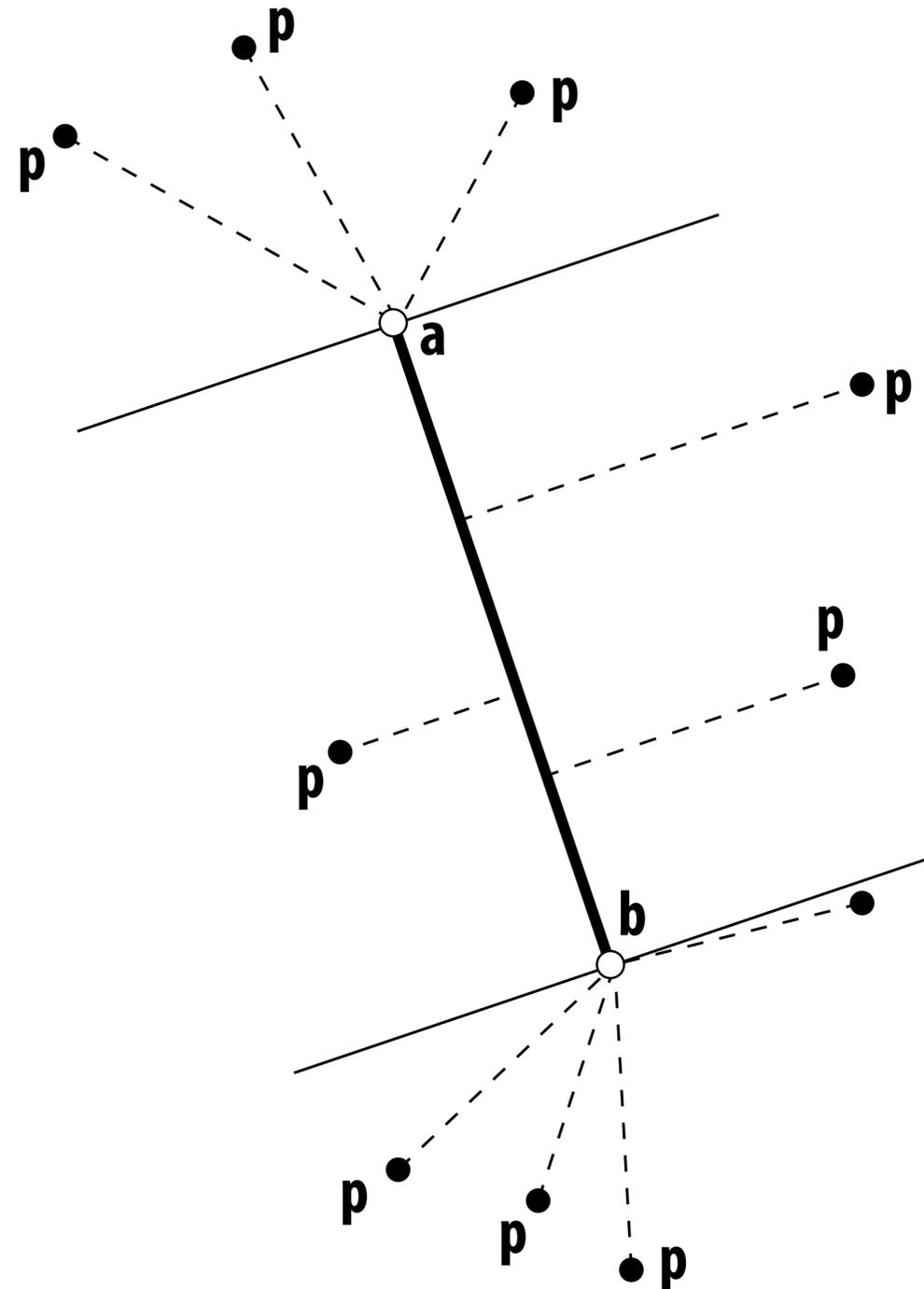
Closest point on line

- Now suppose I have a line $\mathbf{N}^T \mathbf{x} = c$, where \mathbf{N} is the unit normal
 - Remember: a line is all points \mathbf{x} such that $\mathbf{N}^T \mathbf{x} = c$
- How do I find the point on line that is closest to my query point \mathbf{p} ?



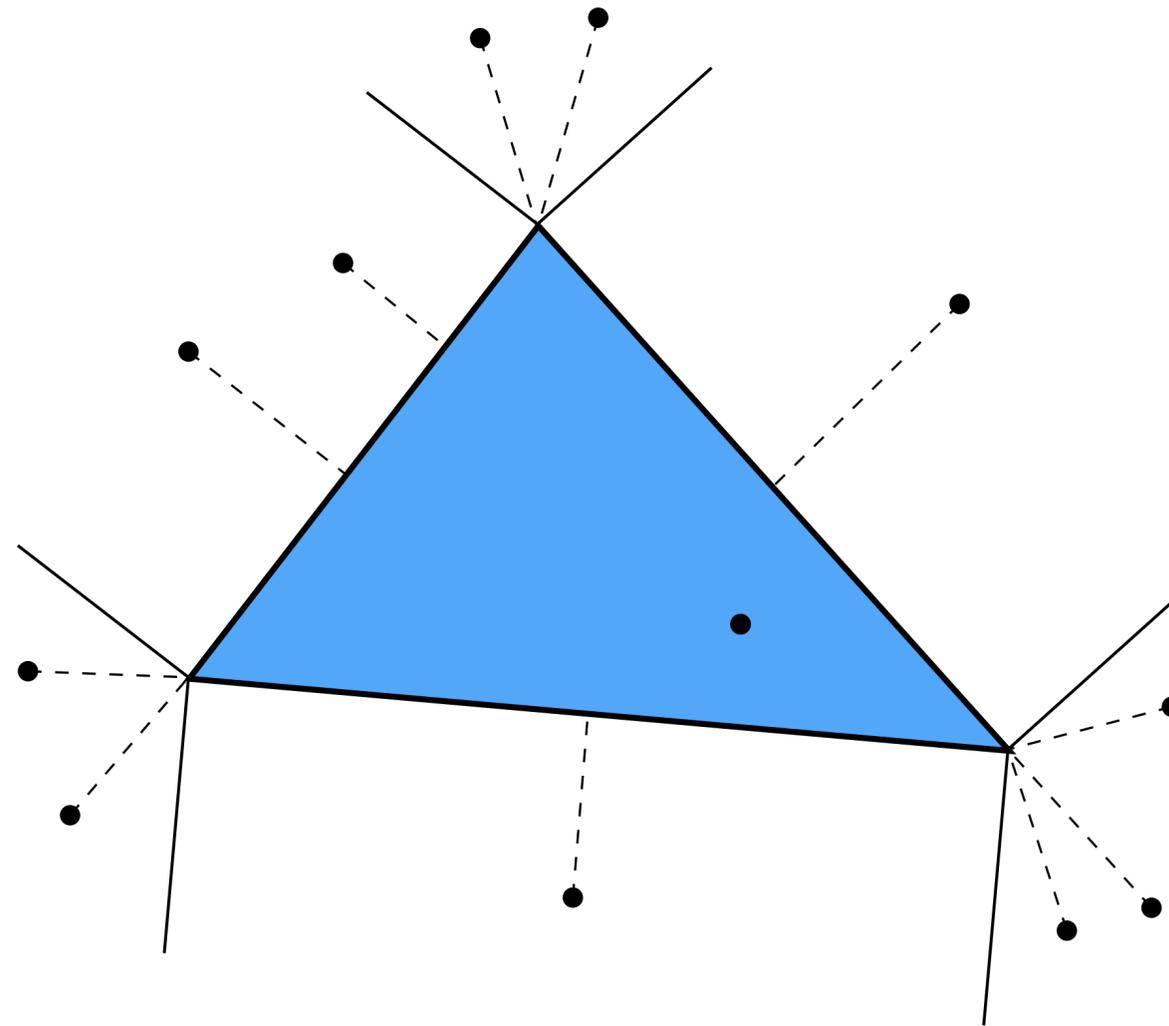
Harder: closest point on line segment

- 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:



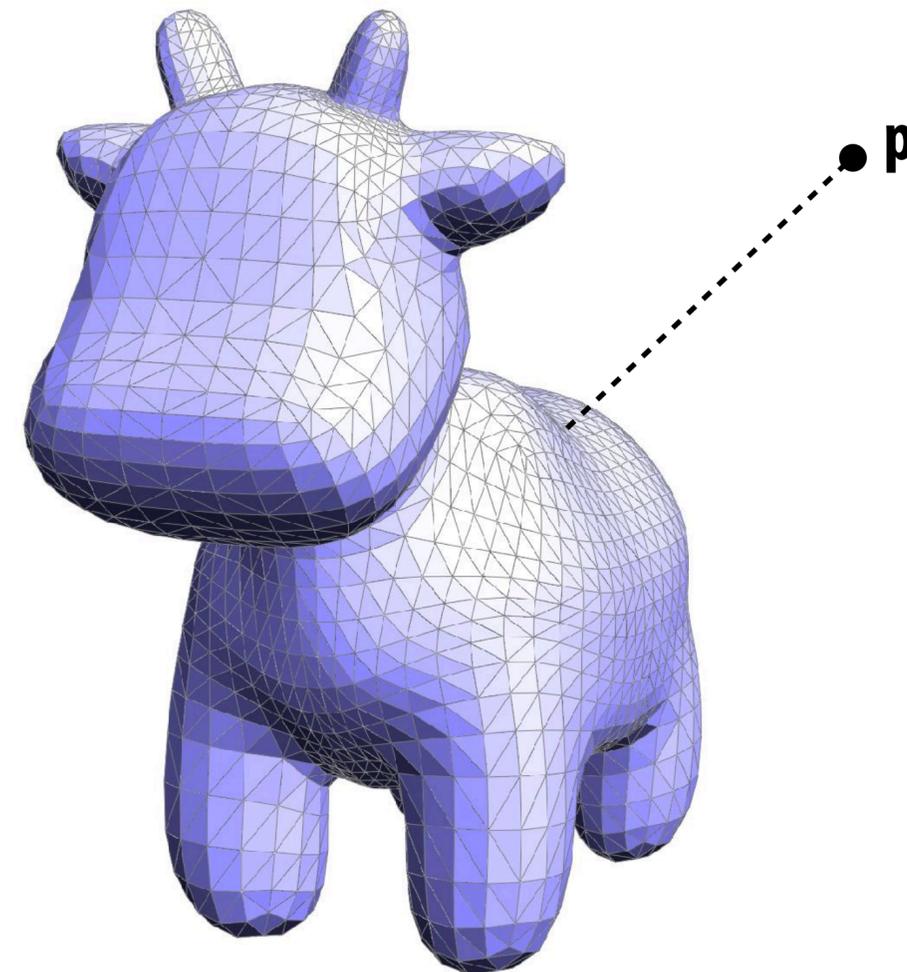
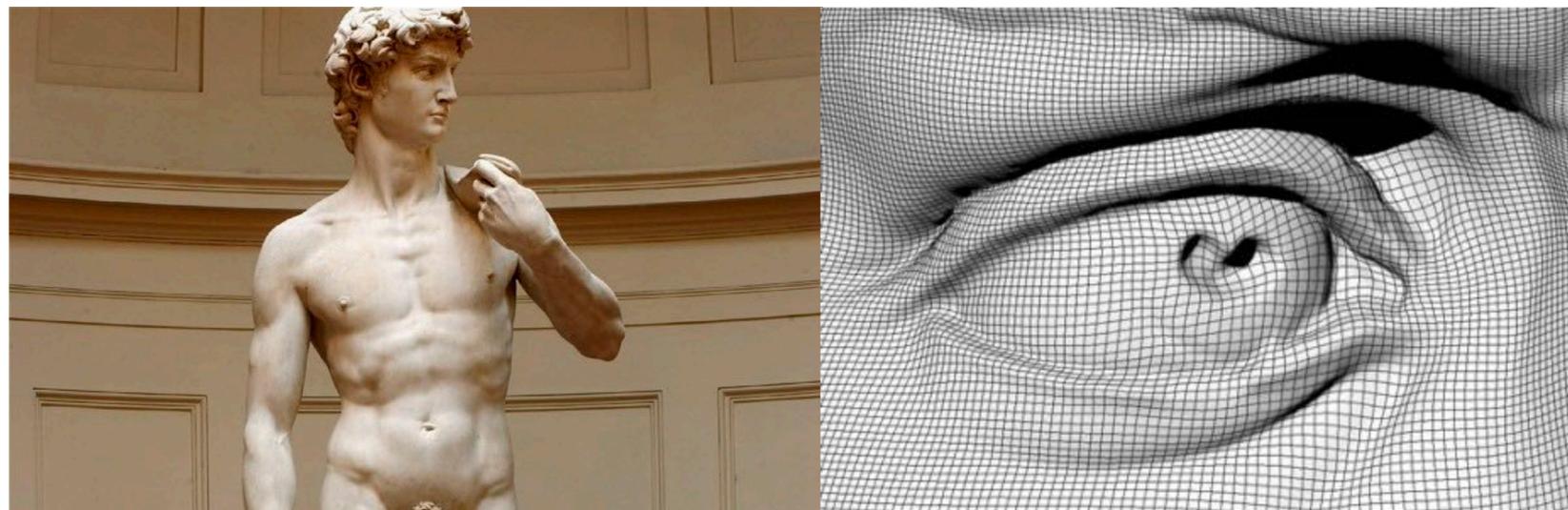
Q: What about a point inside the triangle?

Closest point on triangle in 3D

- Not so different from 2D case
- Algorithm:
 - Project point onto plane of triangle
 - Use three half-*plane* 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! $p + (c - N^T p) N$

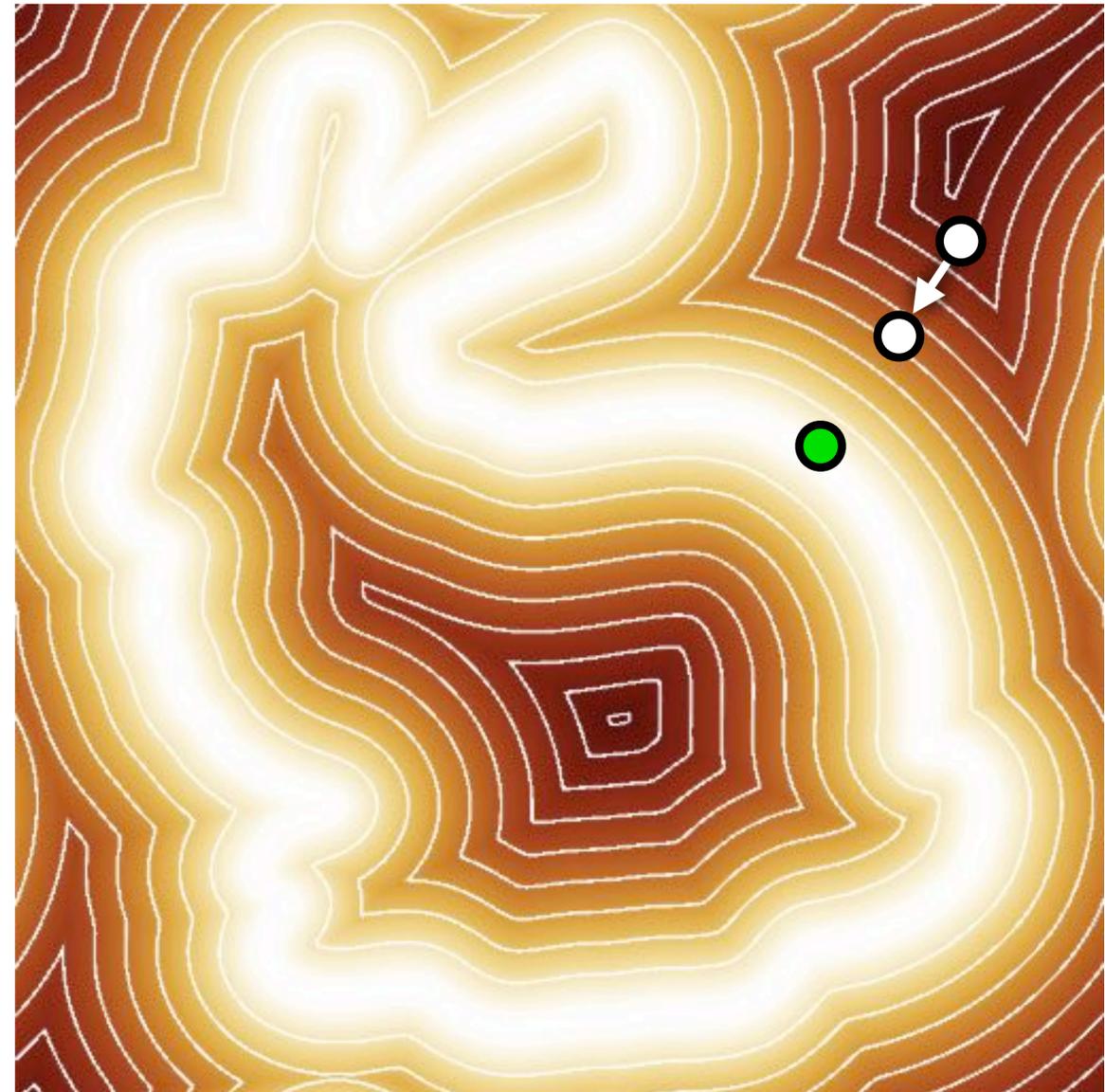
Closest point on triangle *mesh* in 3D?

- **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!**



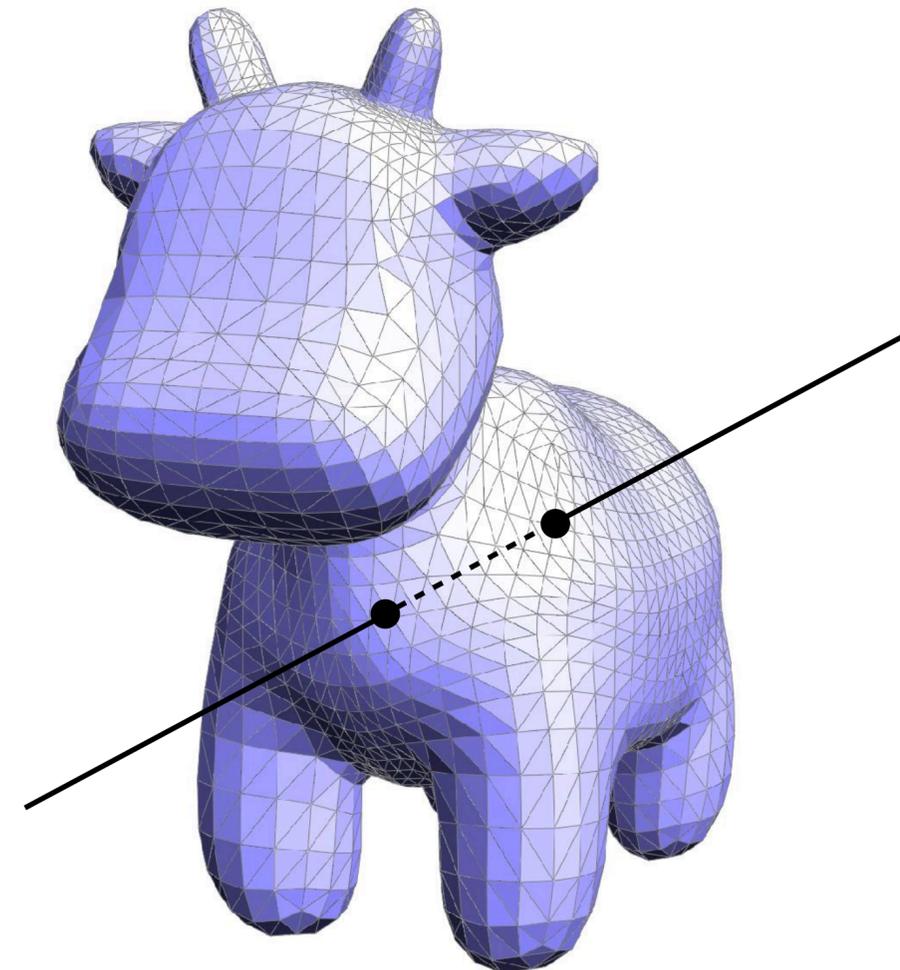
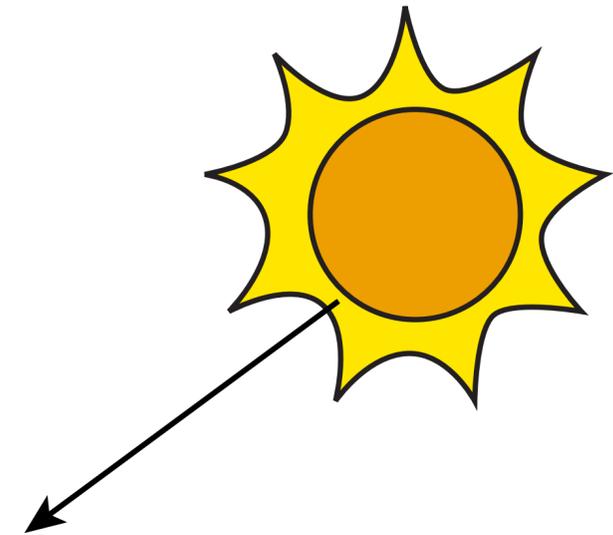
Closest point to *implicit* surface?

- 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)



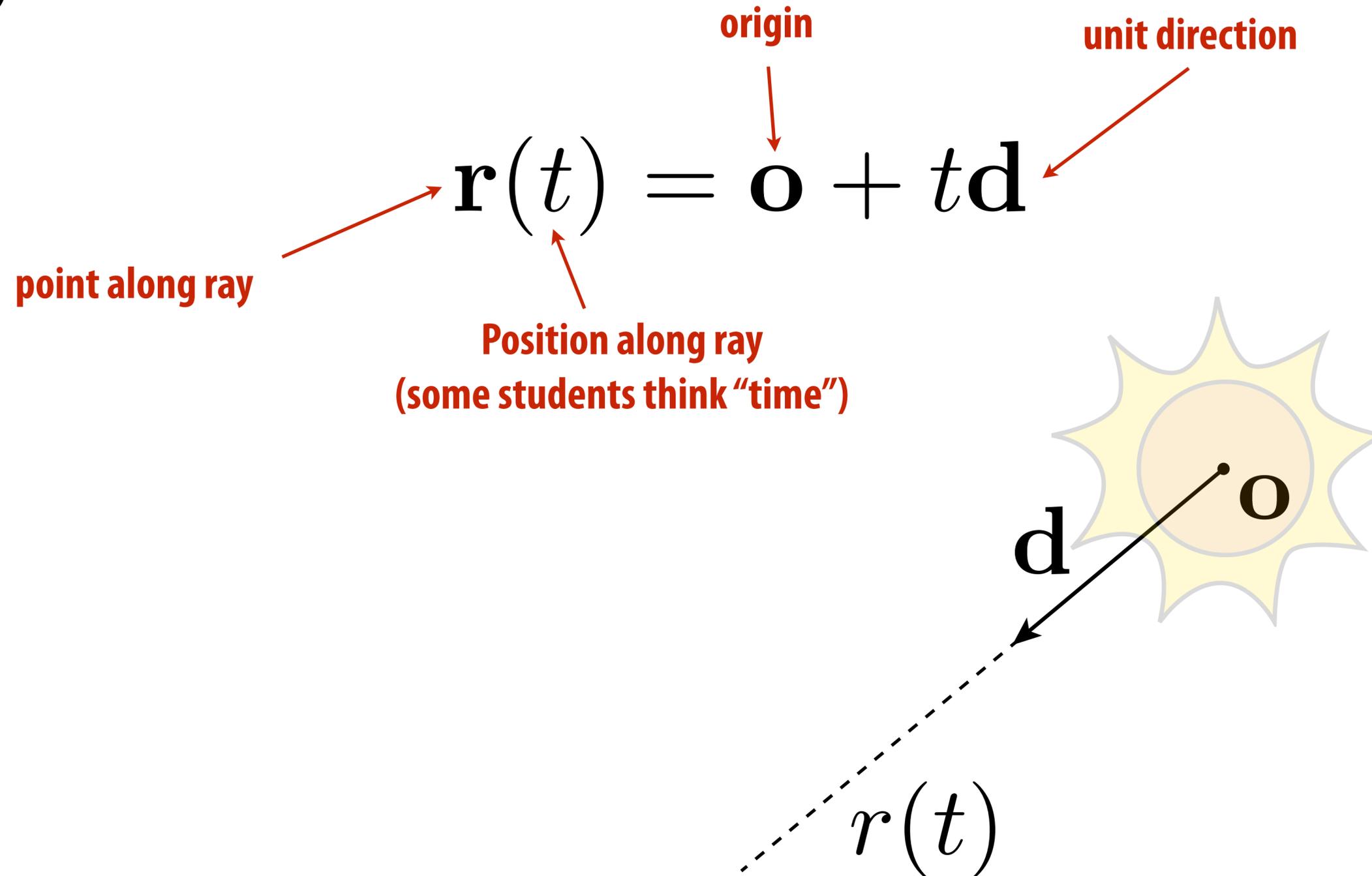
Different query: ray-mesh intersection

- 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
 - Notice: this is a different query than finding the closest point on surface from ray’s origin.
- Applications?
 - GEOMETRY: inside-outside test
 - RENDERING: visibility, ray tracing
 - ANIMATION: collision detection
- Ray might pierce surface in many places!



Ray equation

Can express ray as...



Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points \mathbf{x} such that $f(\mathbf{x}) = 0$
- Q: How do we find points where a ray pierces this surface?
- Well, we know all points along the ray: $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- Idea: replace “ \mathbf{x} ” with “ $\mathbf{r}(t)$ ” in 1st equation, and solve for t
- Example: unit sphere

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

$$\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$$

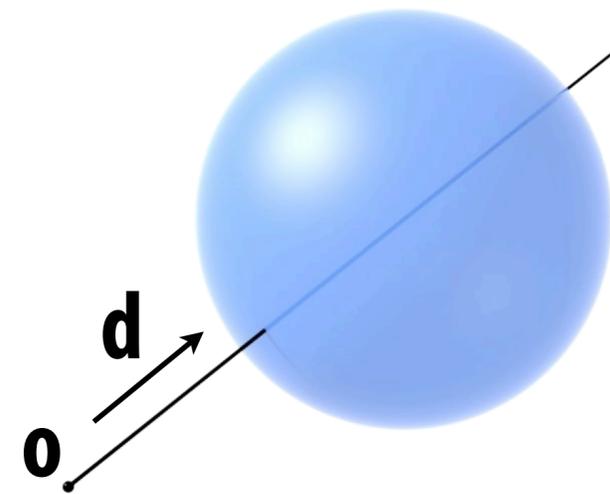
$$\underbrace{|\mathbf{d}|^2}_{a} t^2 + \underbrace{2(\mathbf{o} \cdot \mathbf{d})}_{b} t + \underbrace{|\mathbf{o}|^2 - 1}_{c} = 0$$

Note: $|\mathbf{d}|^2 = 1$ since \mathbf{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:

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



Why two solutions?

Ray-plane intersection

- Suppose we have a plane $\mathbf{N}^T \mathbf{x} = c$

- \mathbf{N} - unit normal
- c - offset

- How do we find intersection with ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$?

- *Key idea:* again, replace the point \mathbf{x} with the ray equation t :

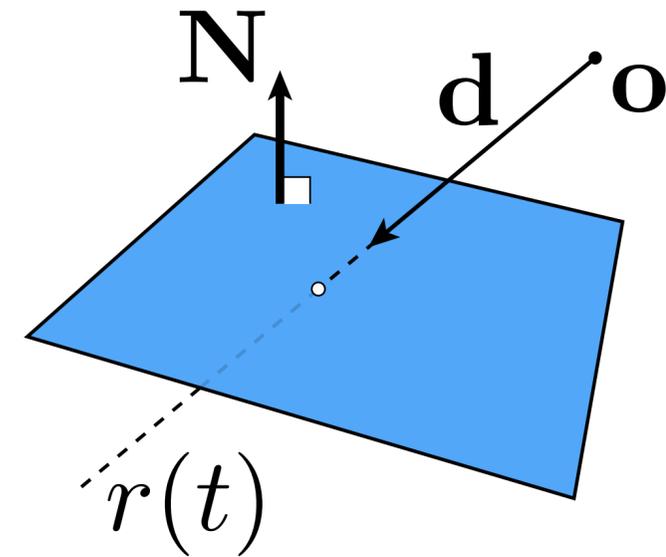
$$\mathbf{N}^T \mathbf{r}(t) = c$$

- Now solve for t :

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

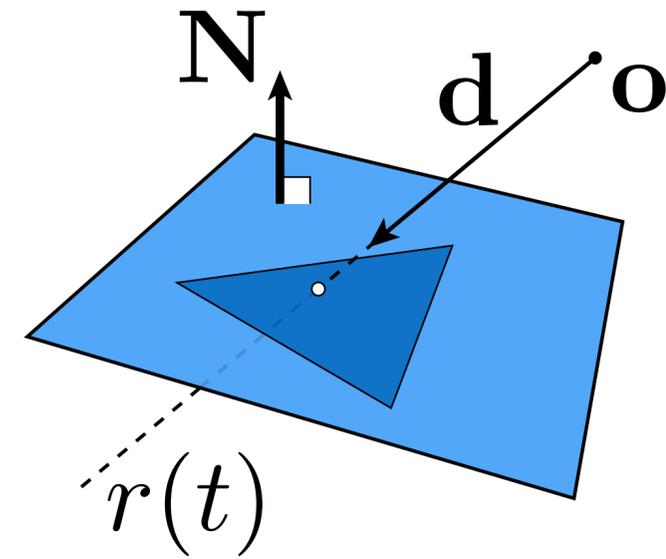
- And plug t back into ray equation:

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

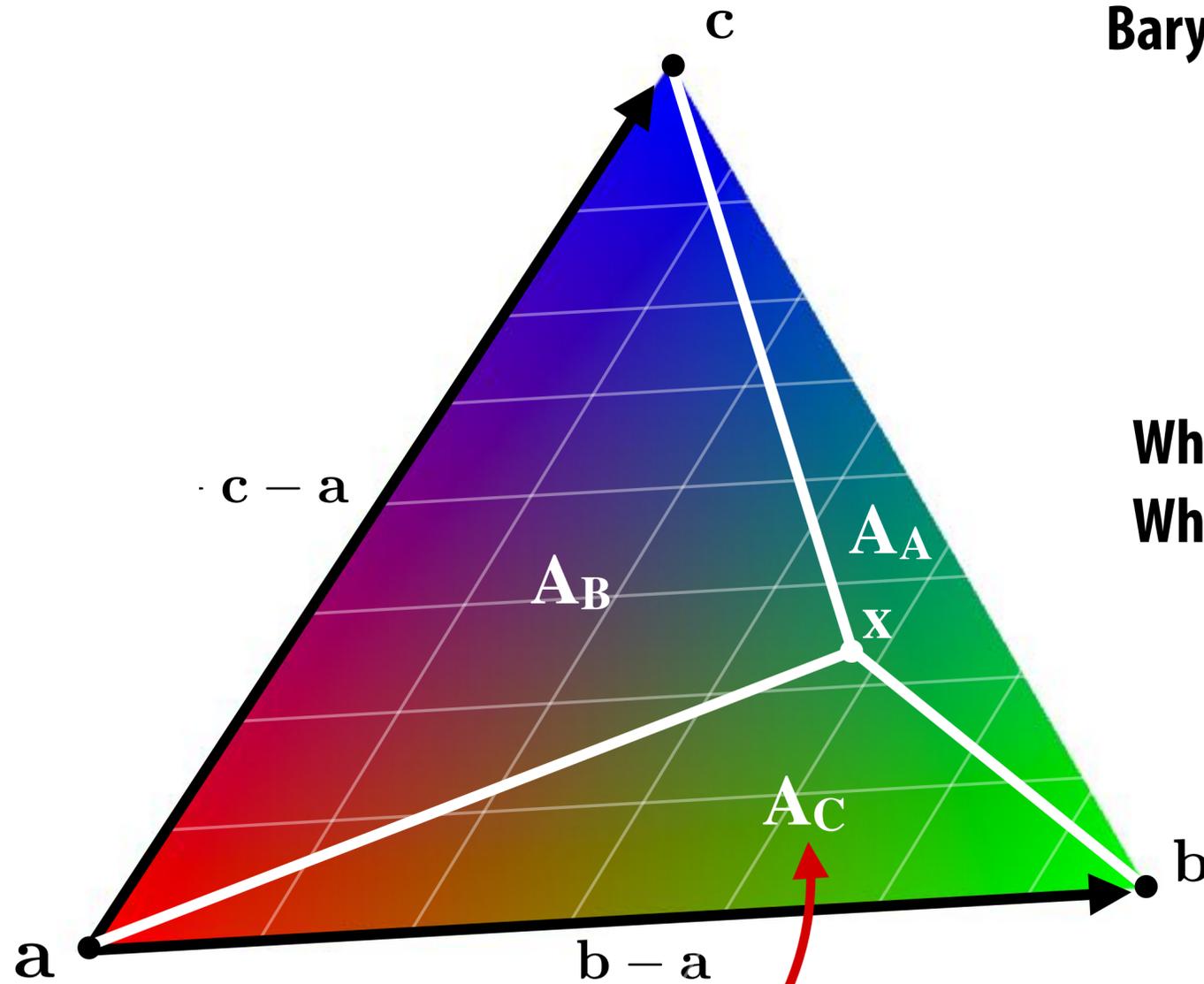


Ray-triangle intersection

- Triangle is in a plane...
- Algorithm:
 - Compute ray-plane intersection
 - Q: What do we do now?



Barycentric coordinates (as ratio of areas)



Area of triangle formed
by points: a, b, x

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?

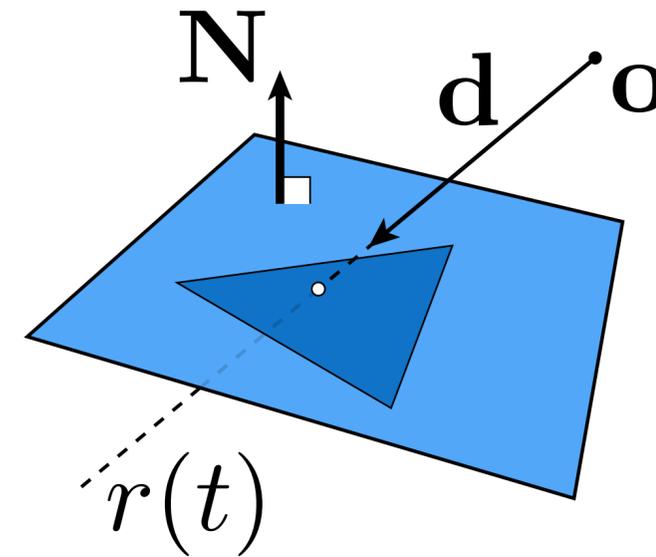
Useful: Heron's formula:

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

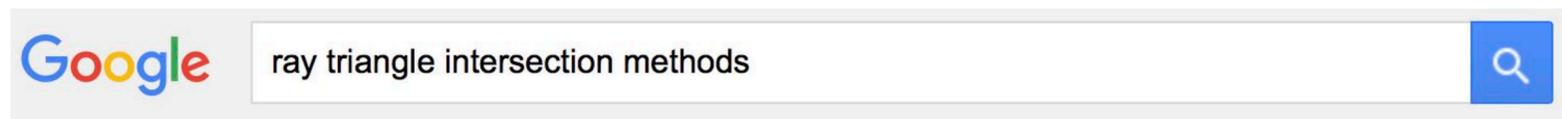
Ray-triangle intersection

■ Algorithm:

- Compute ray-plane intersection
- Compute barycentric coordinates of hit point
- If barycentric coordinates are all positive, point is in triangle



■ Many different techniques if you care about efficiency



About 443,000 results (0.44 seconds)

[Möller–Trumbore intersection algorithm - Wikipedia, the free ...](https://en.wikipedia.org/.../Möller-Trumbore_intersection_alg...)
[https://en.wikipedia.org/.../Möller–Trumbore_intersection_alg...](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...)
<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](http://www.cs.utah.edu/~aek/research/triangle.pdf)
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.

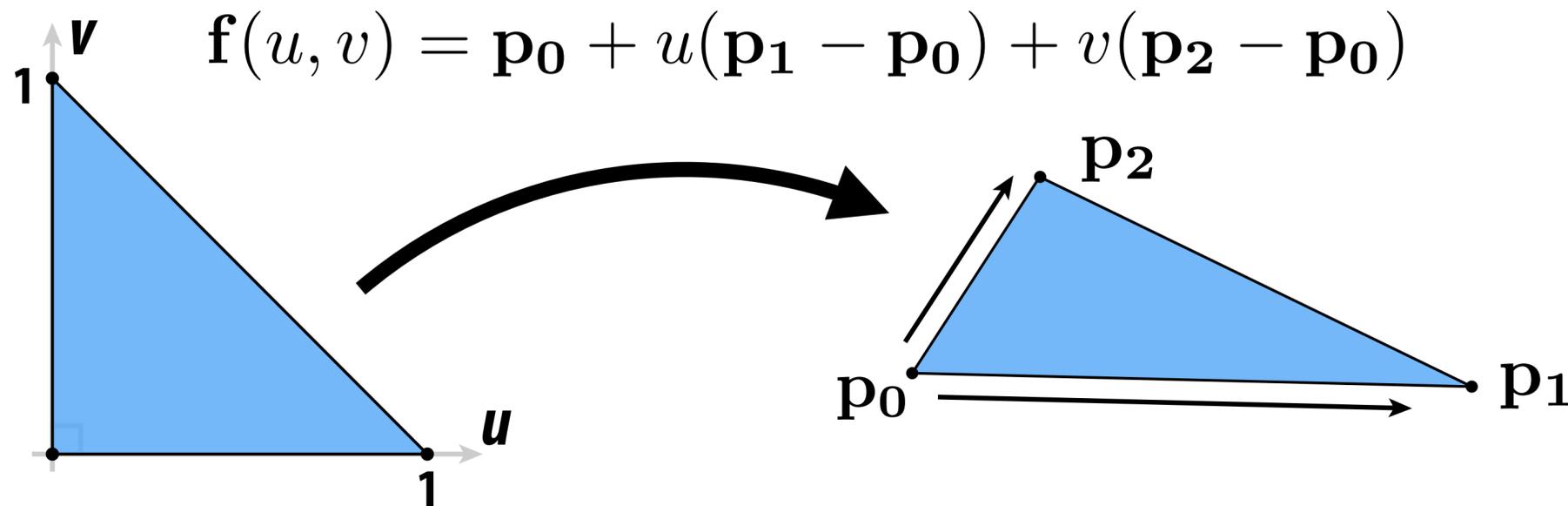
[\[PDF\] Comparative Study of Ray-Triangle Intersection Algorithms](http://www.graphicon.ru/html/proceedings/2012/.../gc2012Shumskiy.pdf)
www.graphicon.ru/html/proceedings/2012/.../gc2012Shumskiy.pdf ▾
by V Shumskiy - Cited by 1 - Related articles

Ray-triangle intersection (another way)

- Parameterize triangle with vertices p_0, p_1, p_2 using barycentric coordinates *

$$f(u, v) = (1 - u - v)p_0 + up_1 + vp_2$$

- Can think of a triangle as an affine map of the unit triangle



* I'm writing u, v instead of beta, gamma to make explicit representation of triangle very clear.

Another way: ray-triangle intersection

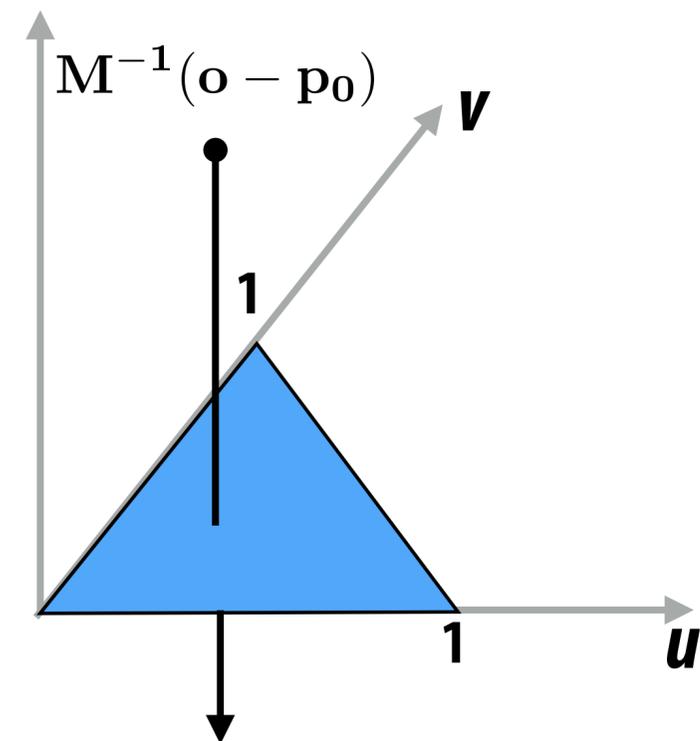
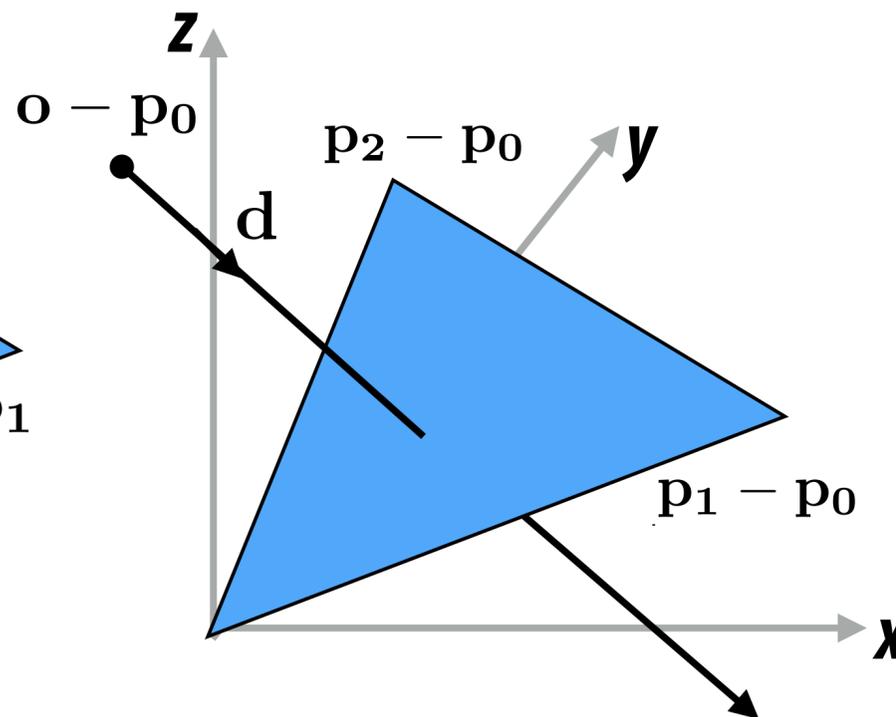
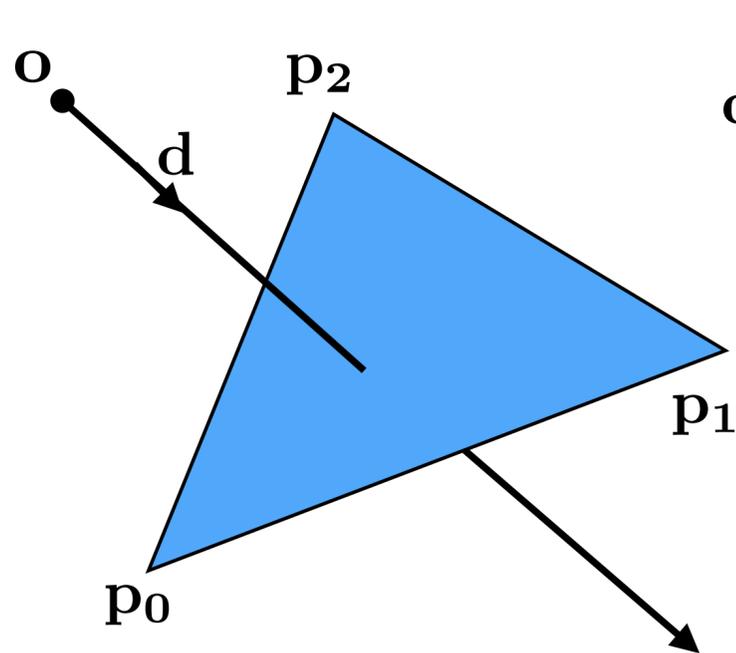
Plug parametric ray equation directly into equation for points on triangle:

$$\mathbf{p}_0 + u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0) = \mathbf{o} + t\mathbf{d}$$

Solve for u, v, t :

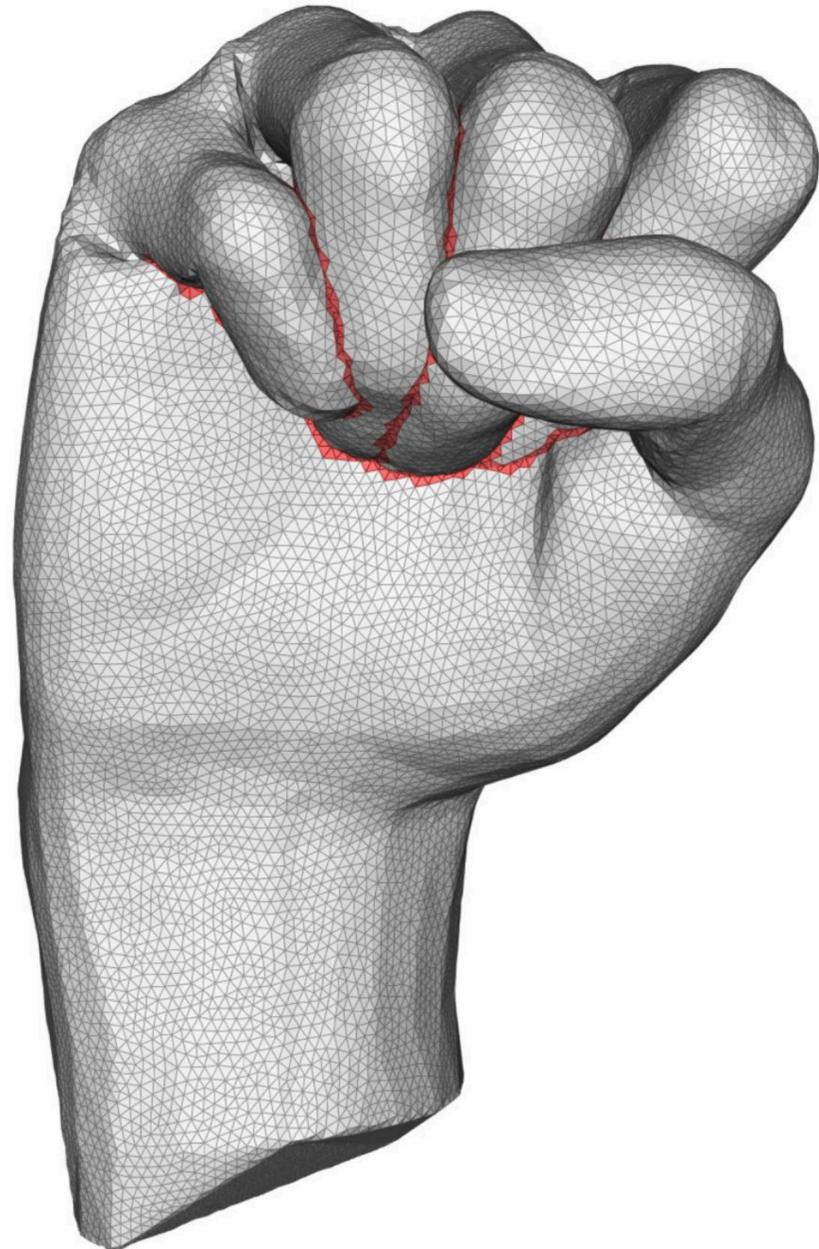
$$\underbrace{\begin{bmatrix} \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 & -\mathbf{d} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} u \\ v \\ t \end{bmatrix} = \mathbf{o} - \mathbf{p}_0$$

\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!

(p_x, p_y)
●

● (a_1, a_2)

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
●

$$N^T x = c$$

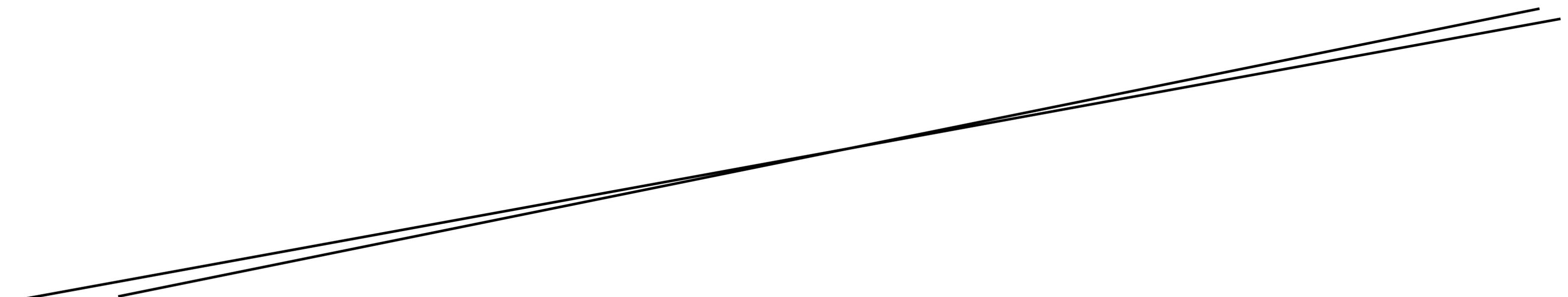
Line-line intersection

- 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).**



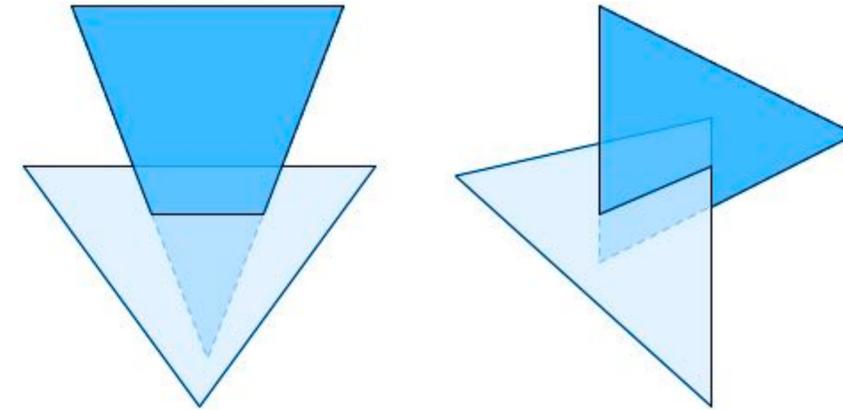
See for example Shewchuk, “*Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*”

Triangle-triangle intersection?

- Lots of ways to do it

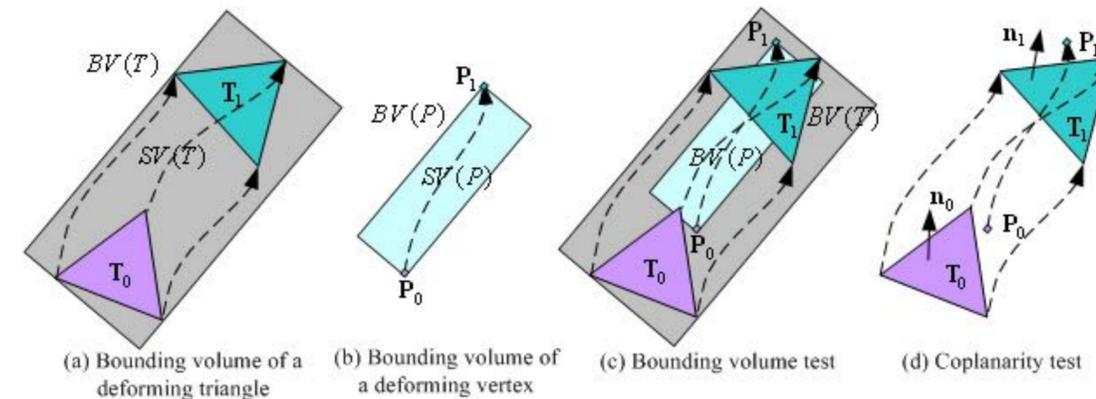
- Basic idea:

- 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 $nD + \text{time}$) into purely geometric problem in $(n+1)$ -dimensions



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

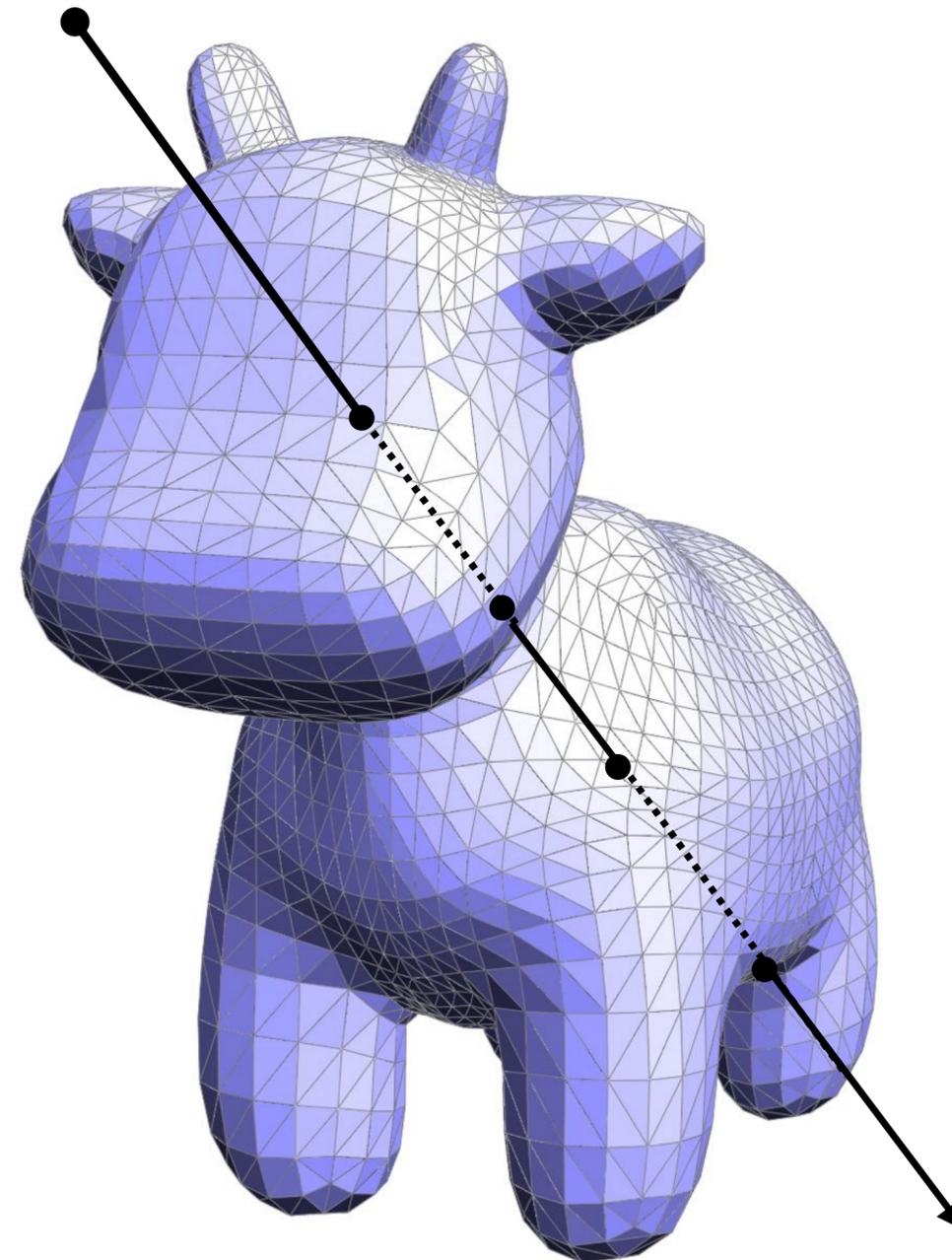
```
t_closest = inf
for each primitive p in scene:
    t = p.intersect(r)
    if t >= 0 && t < t_closest:
        t_closest = t
```

```
// closest hit is:
// r.o + t_closest * r.d
```

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

Complexity? $O(N)$

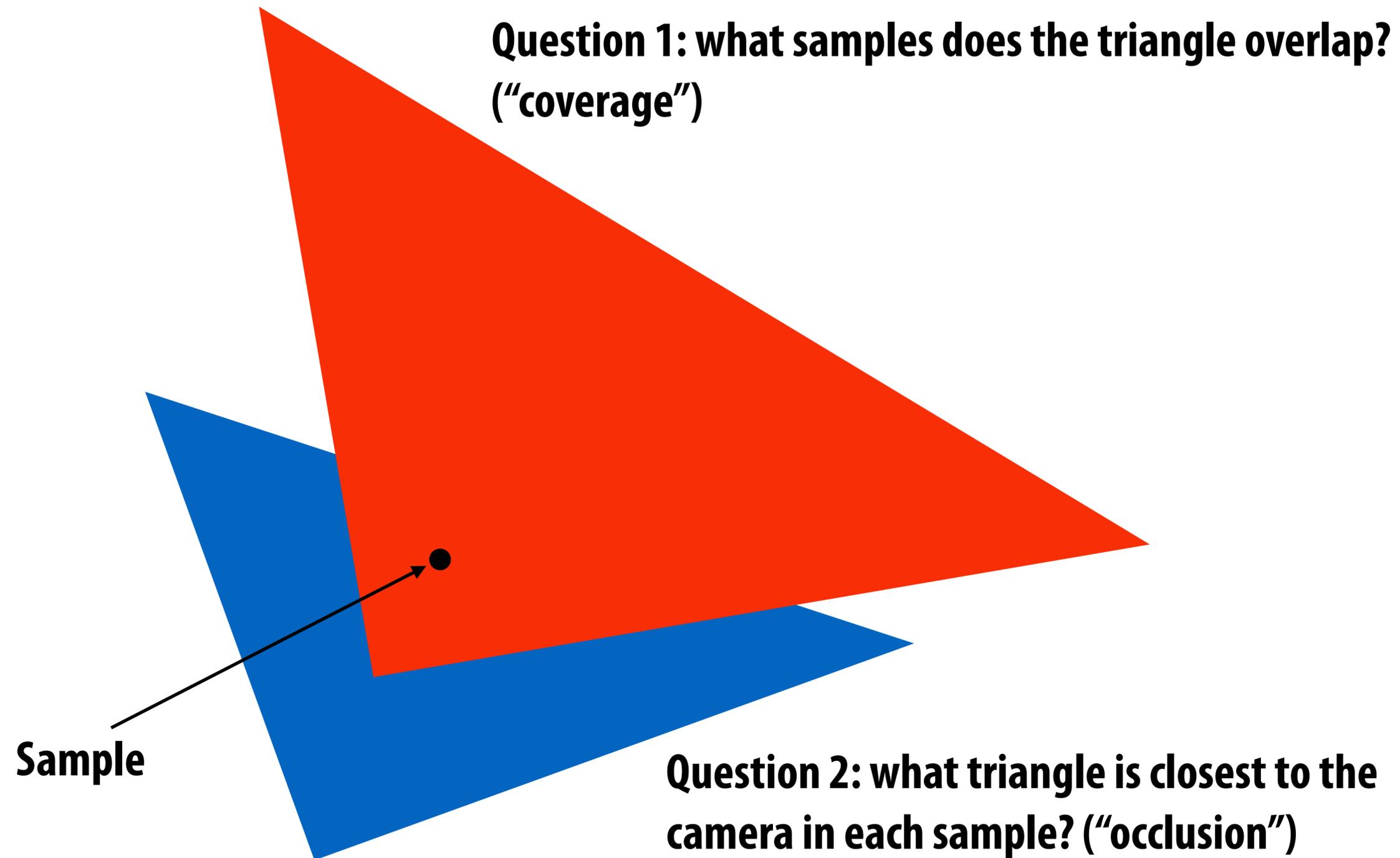
Can we do better? Of course... but you'll have to wait until next class



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

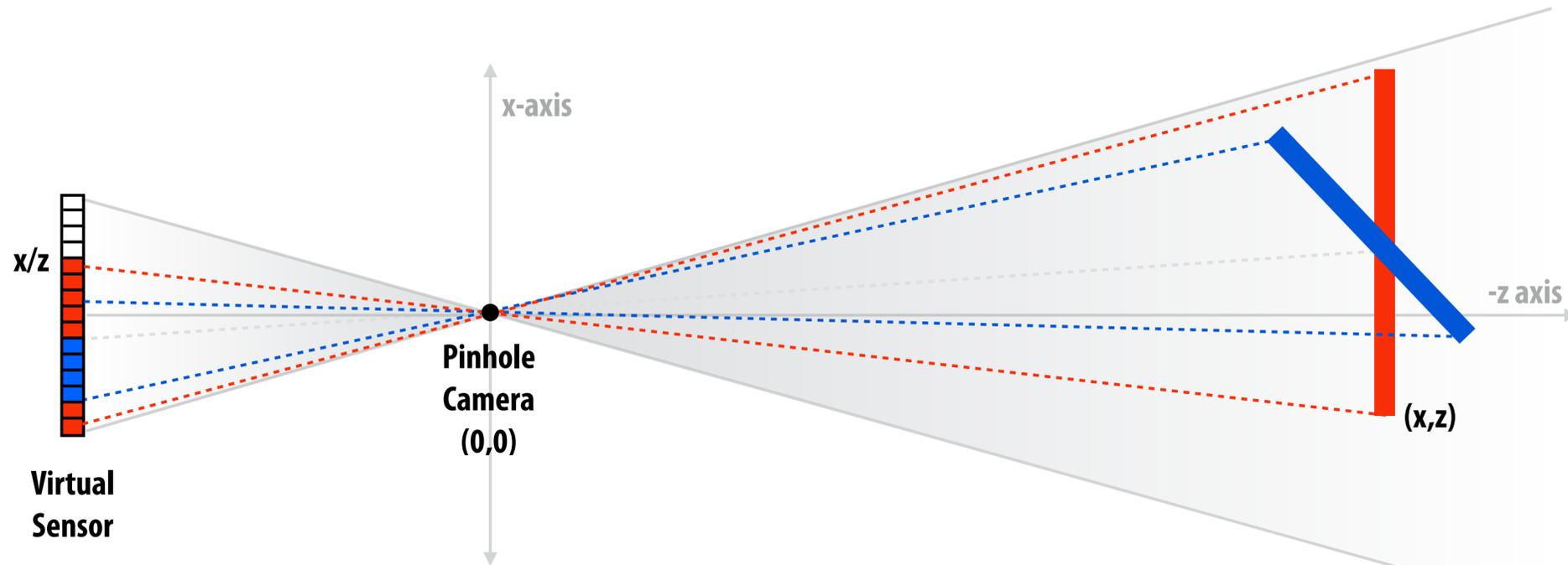
**Rasterization and ray casting are two algorithms for solving the
same problem:
determining “visibility from a camera”**

Recall triangle visibility:



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)



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]
```

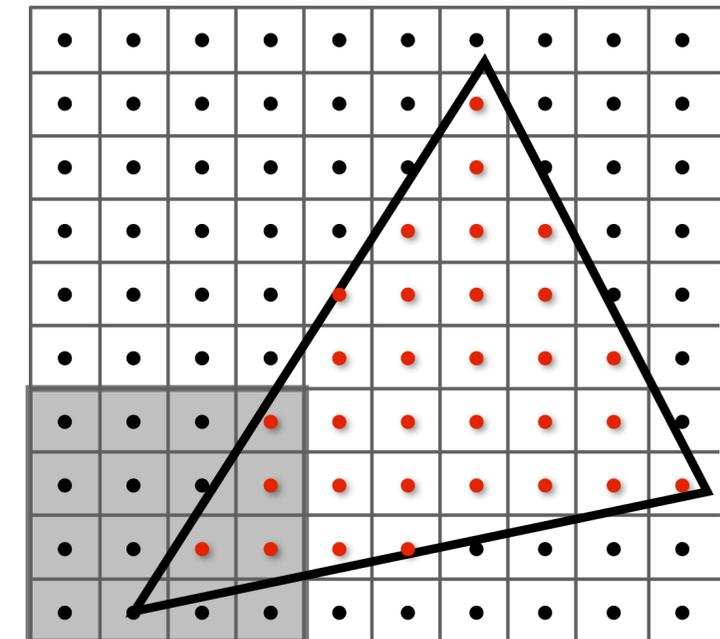
“Given a triangle, find the samples it covers”

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

More efficient hierarchical rasterization:

For each TILE of image

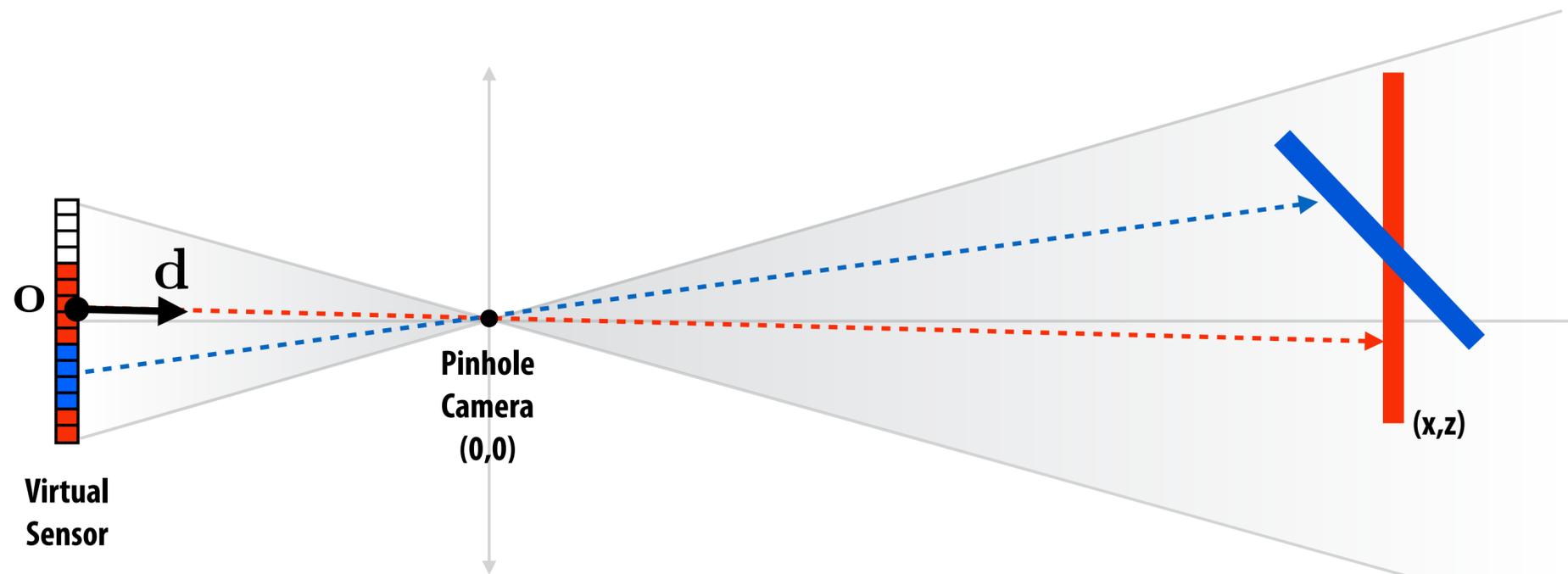
If triangle overlaps tile, check all samples in tile



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:

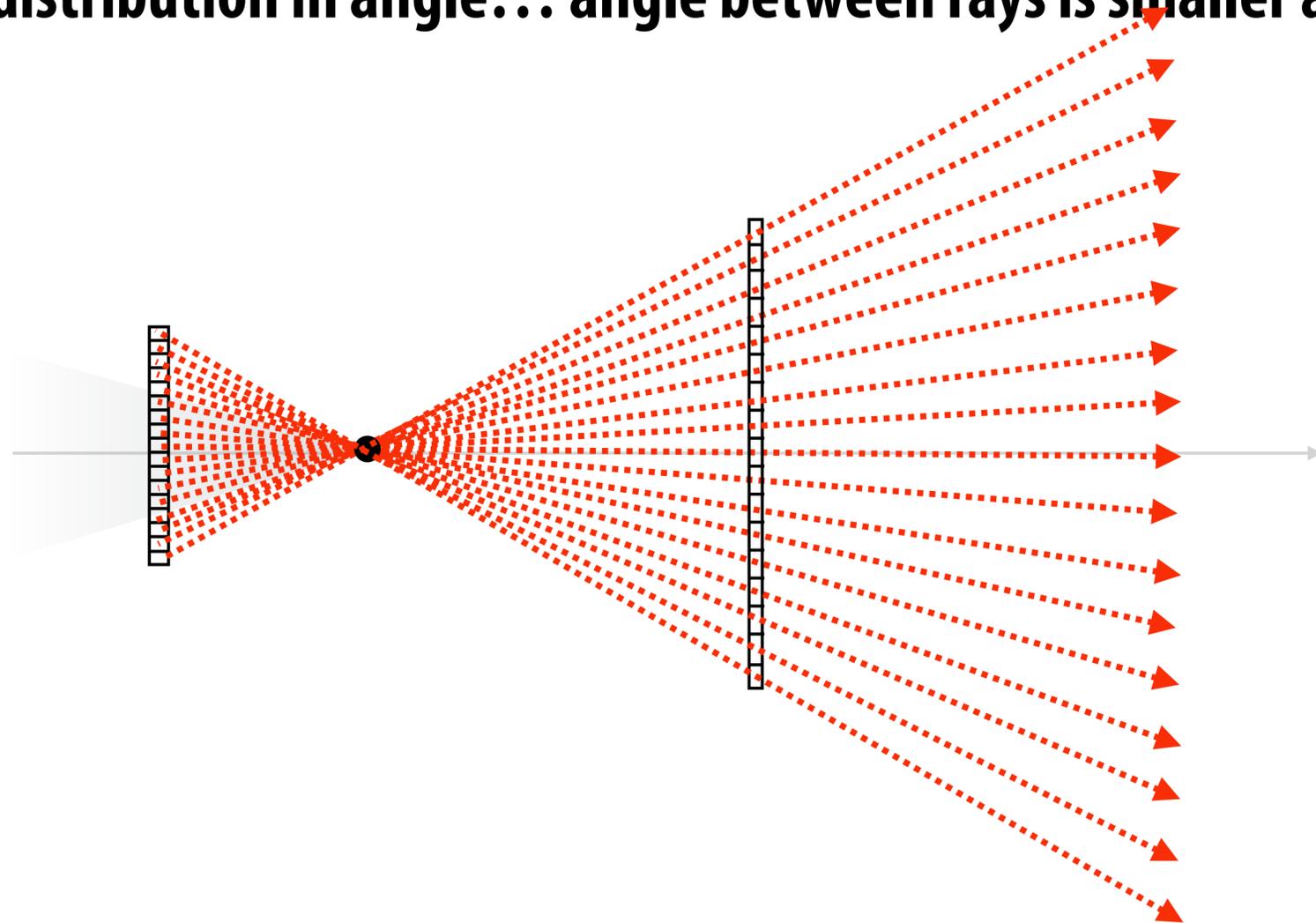
- Proceeds in triangle order (for all triangles)
- Store entire depth buffer (requires access to 2D array of fixed size)
- Do not have to store entire scene geometry in memory
 - Naturally supports unbounded size scenes

■ 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 they are encountered along the ray: front-to-back)
- Must store entire scene geometry for fast access

In other words...

- Rasterization is an 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)

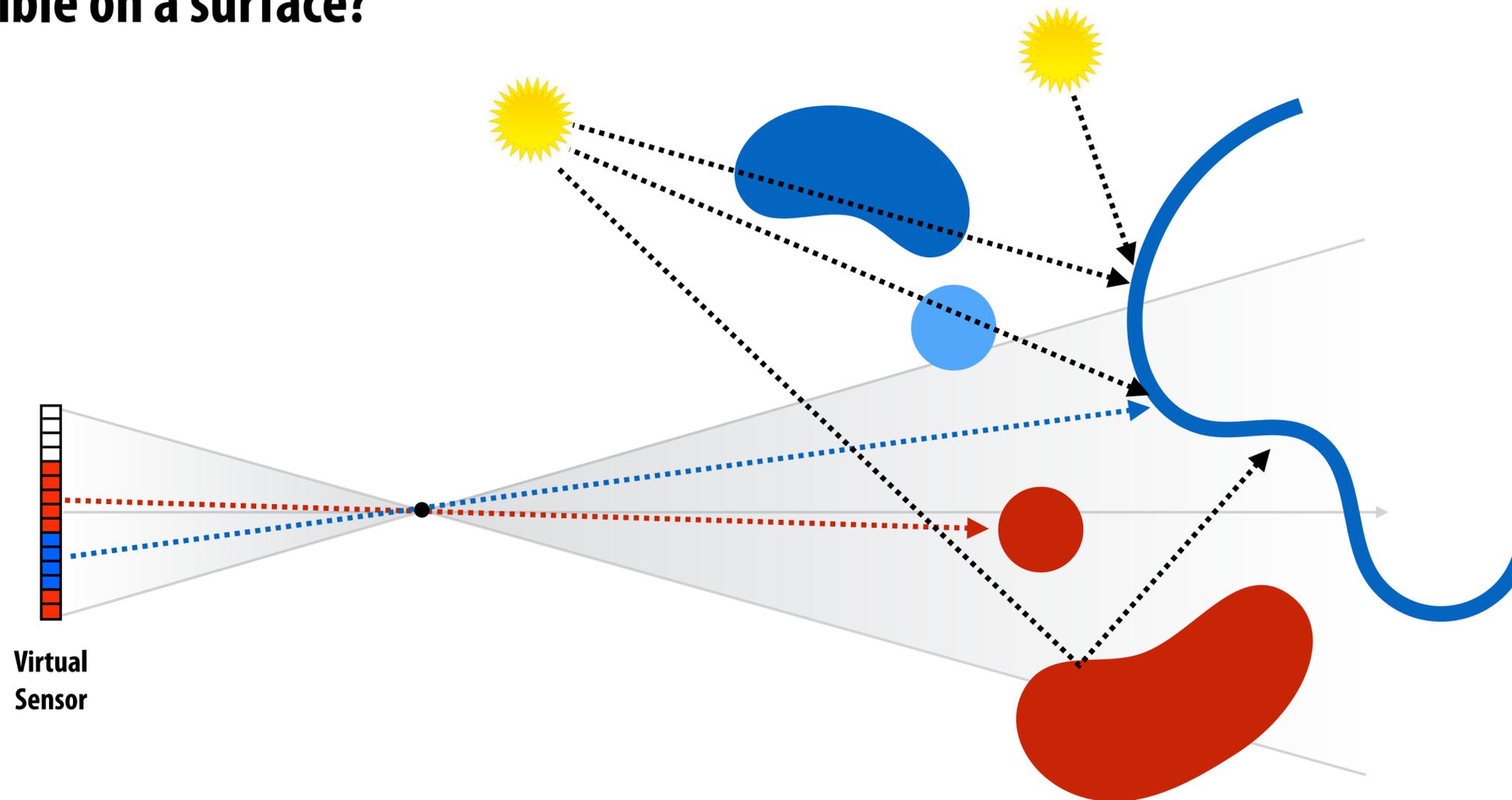


Generality of ray-scene queries

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?



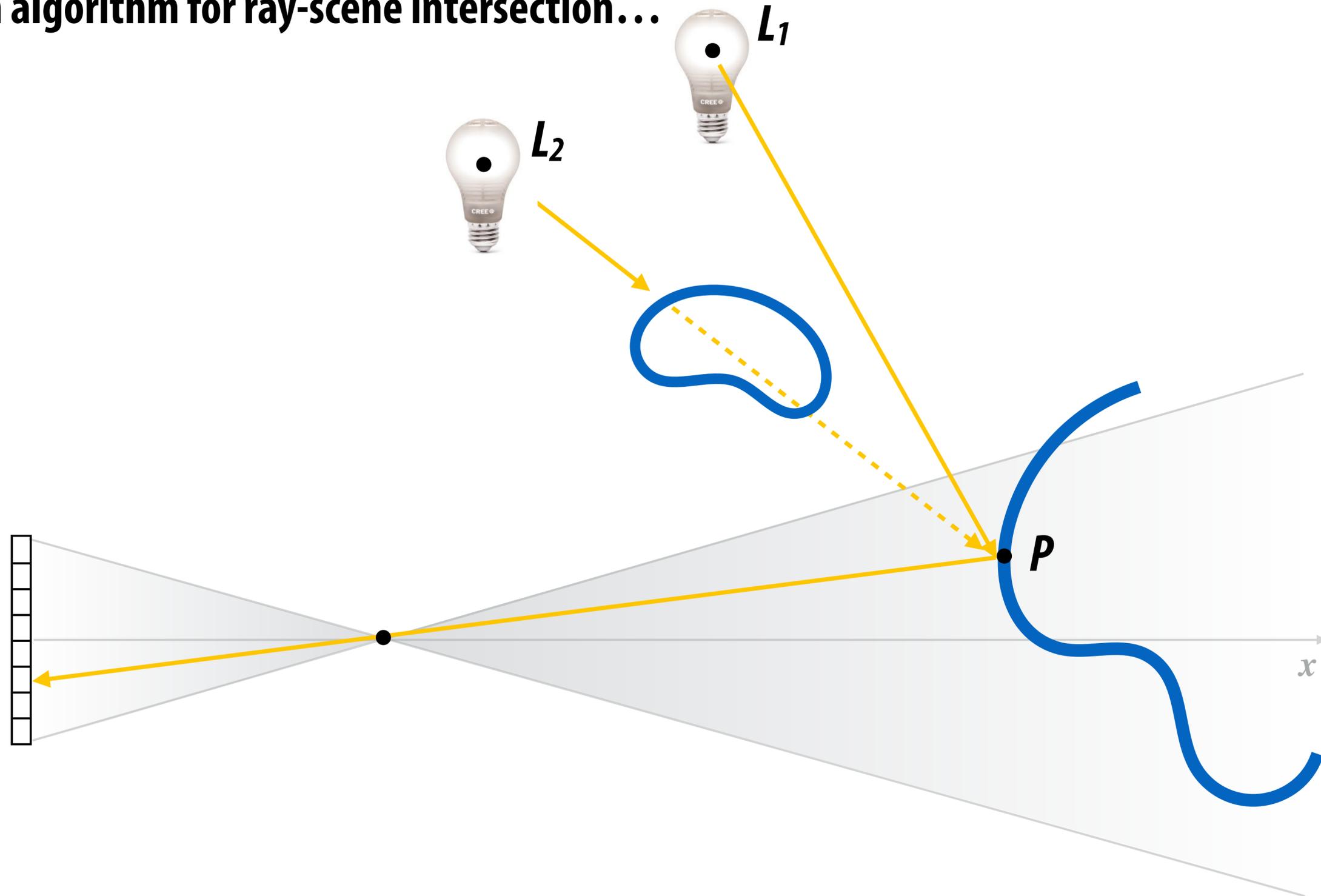
In contrast, rasterization is a highly-specialized solution for computing visibility for a set of uniformly distributed rays originating from the same point (most often: the camera)

Shadows



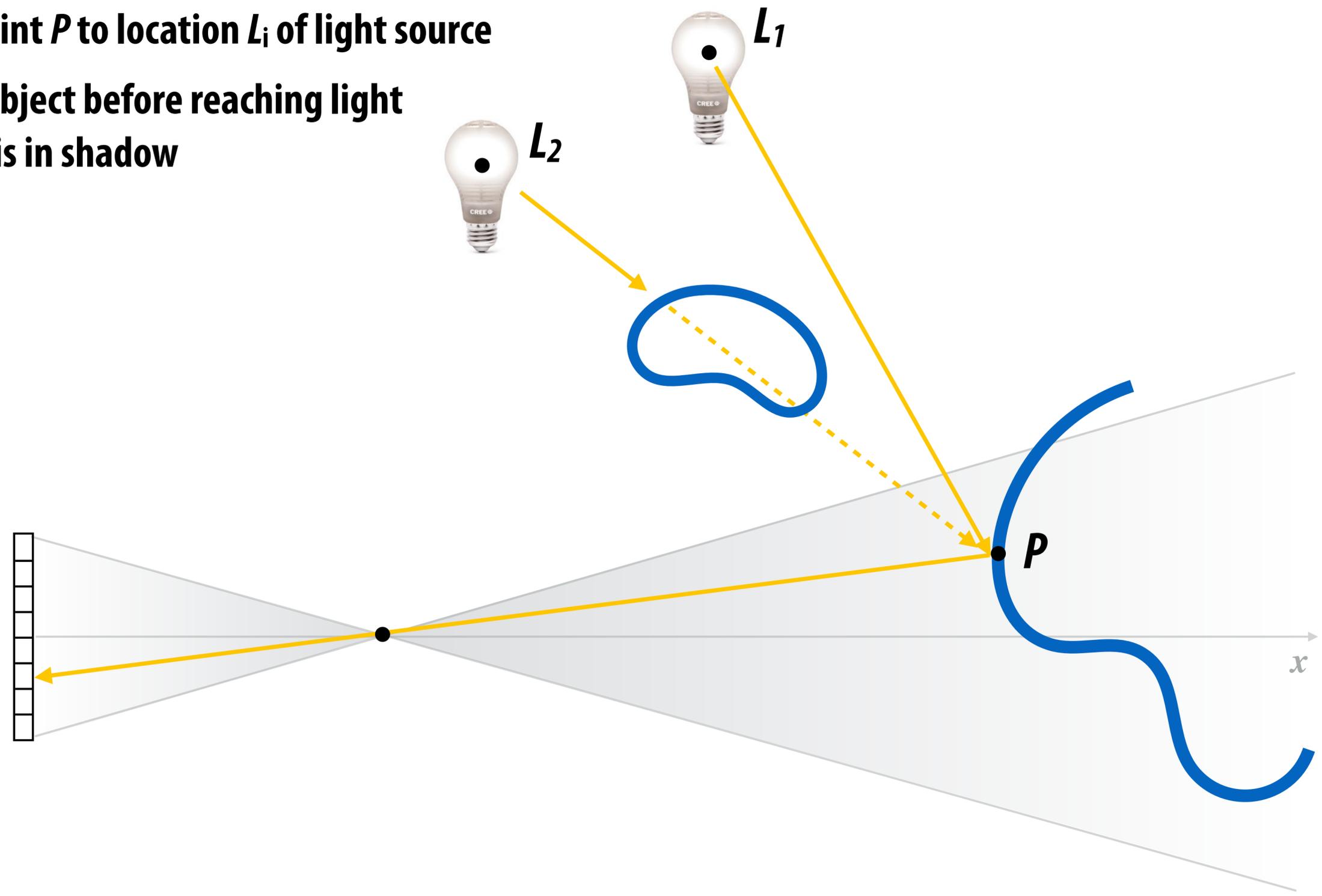
How to compute if a surface point is in shadow?

Assume you have an algorithm for ray-scene intersection...

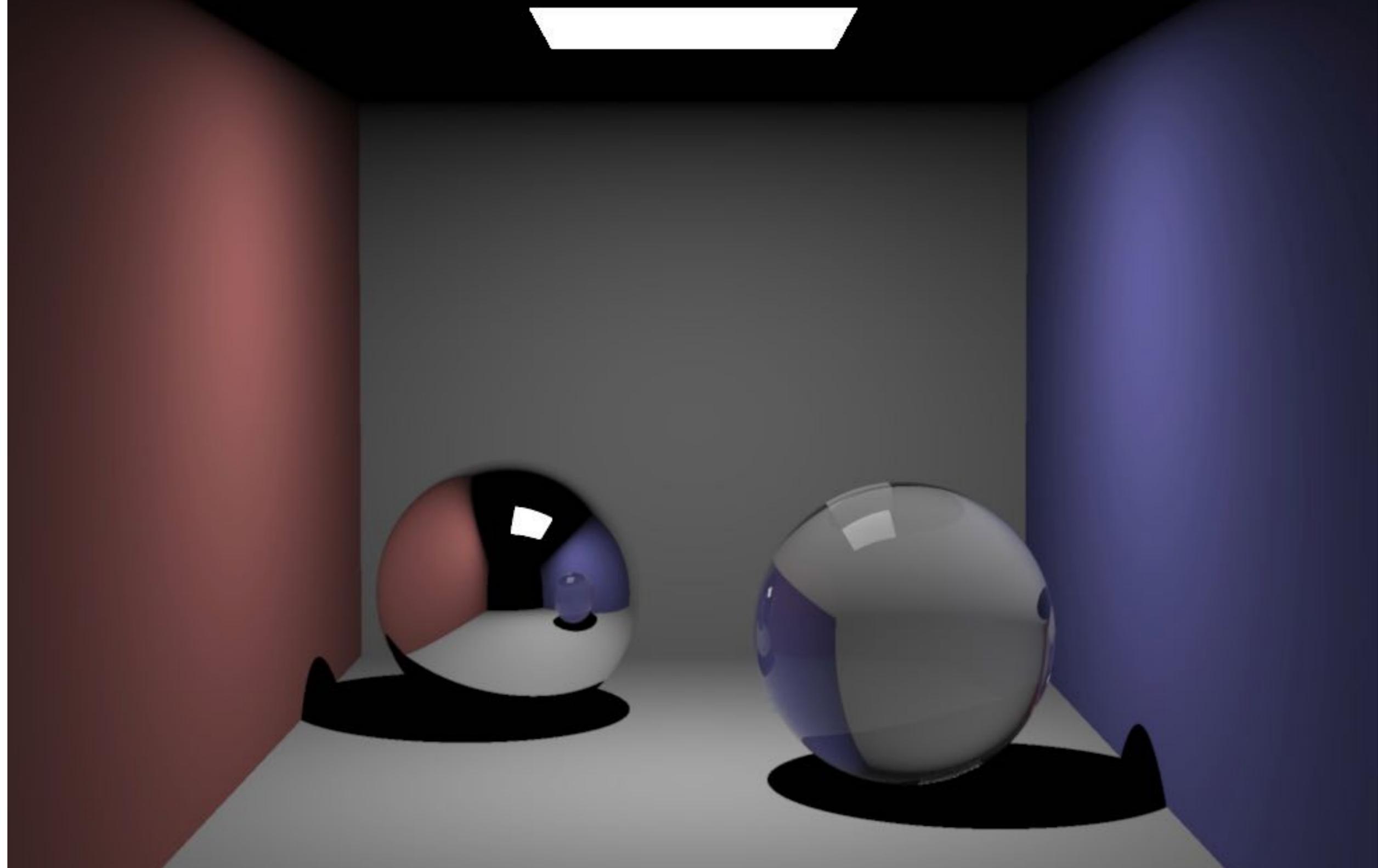


A simple shadow computation algorithm

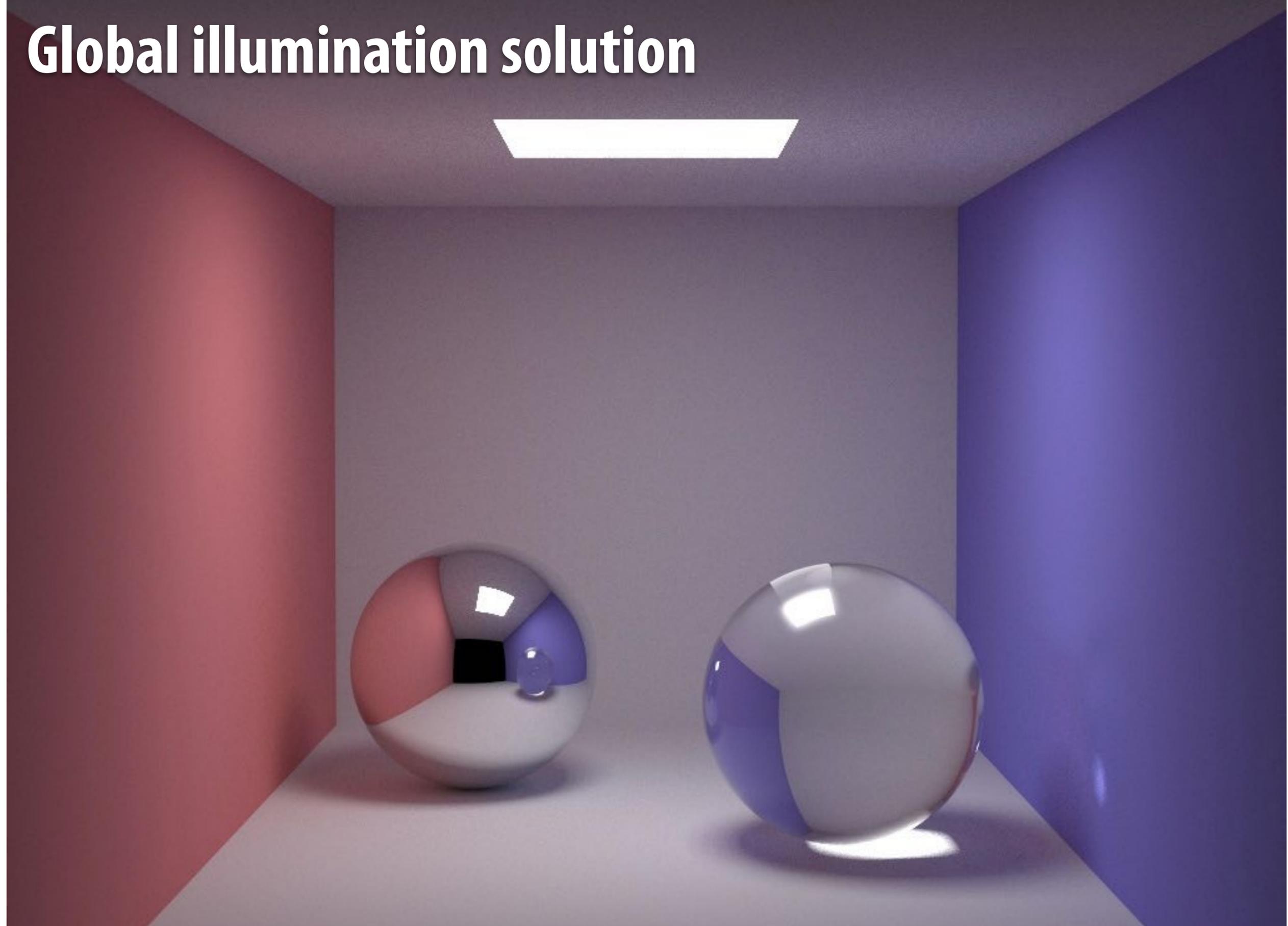
- Trace ray from point P to location L_i of light source
- If ray hits scene object before reaching light source... then P is in shadow



Direct illumination + reflection + transparency



Global illumination solution



Direct illumination



• p

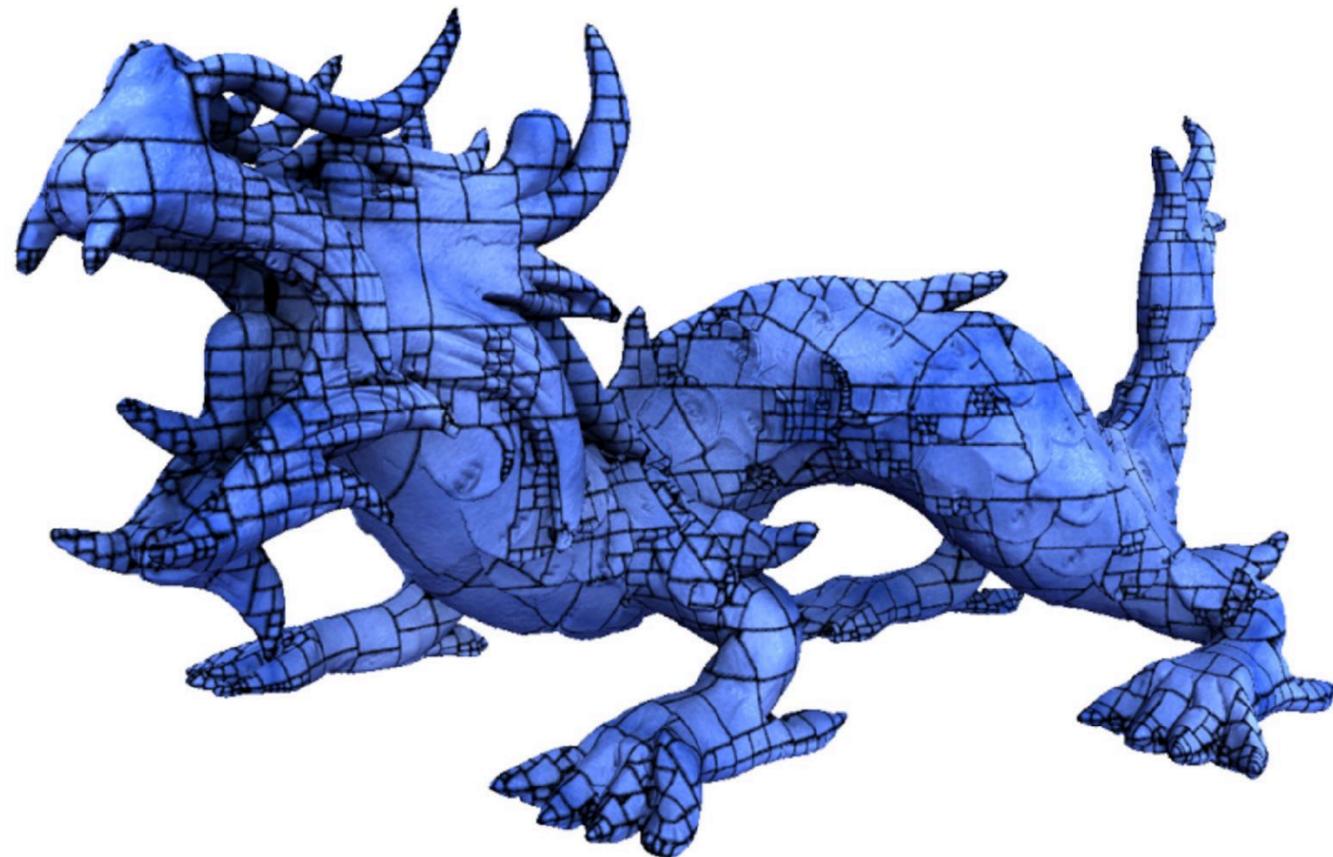
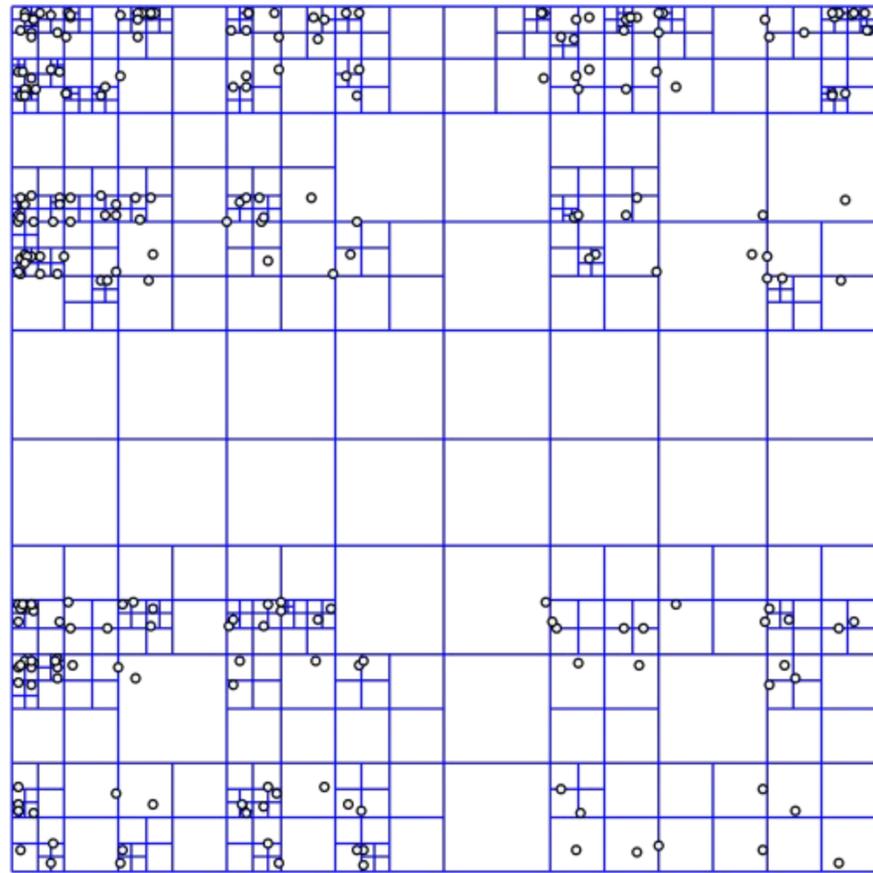
Sixteen-bounce global illumination



• p

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**