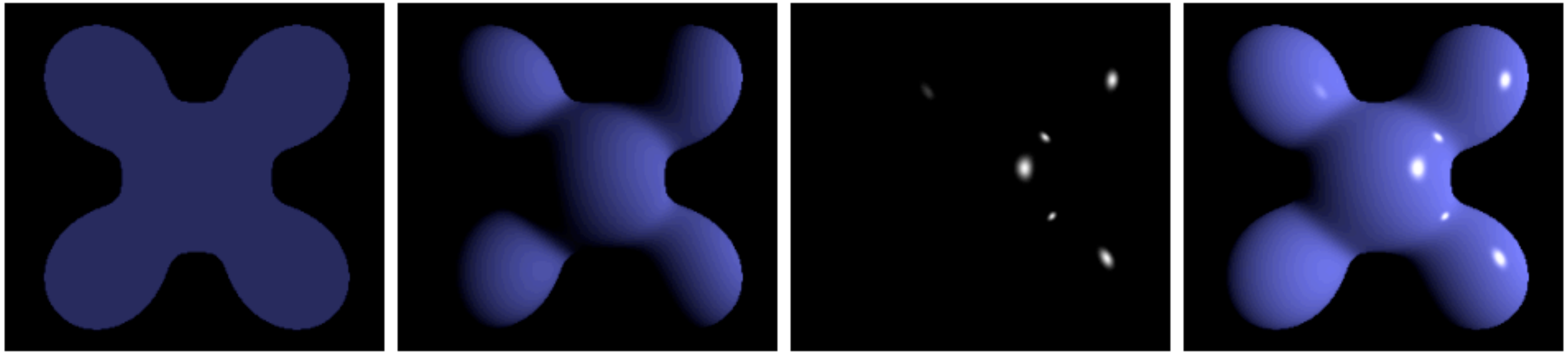
COL781: Computer Graphics

13. Ray Tracing

Turner Whitted

Last class: Local illumination

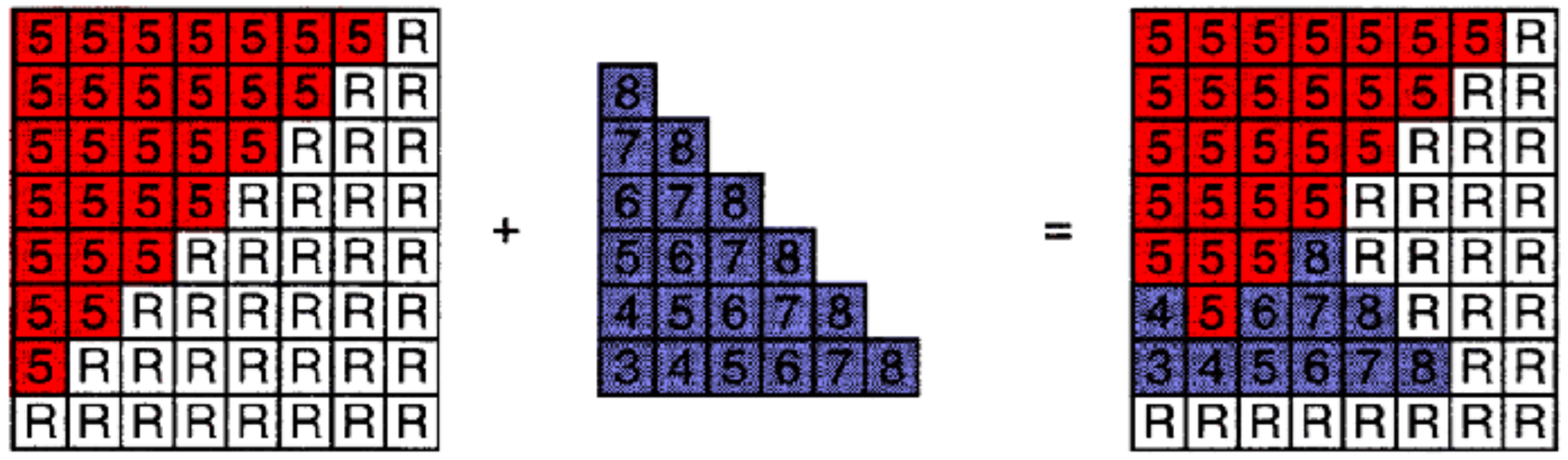


Ambient + Diffuse + Specular

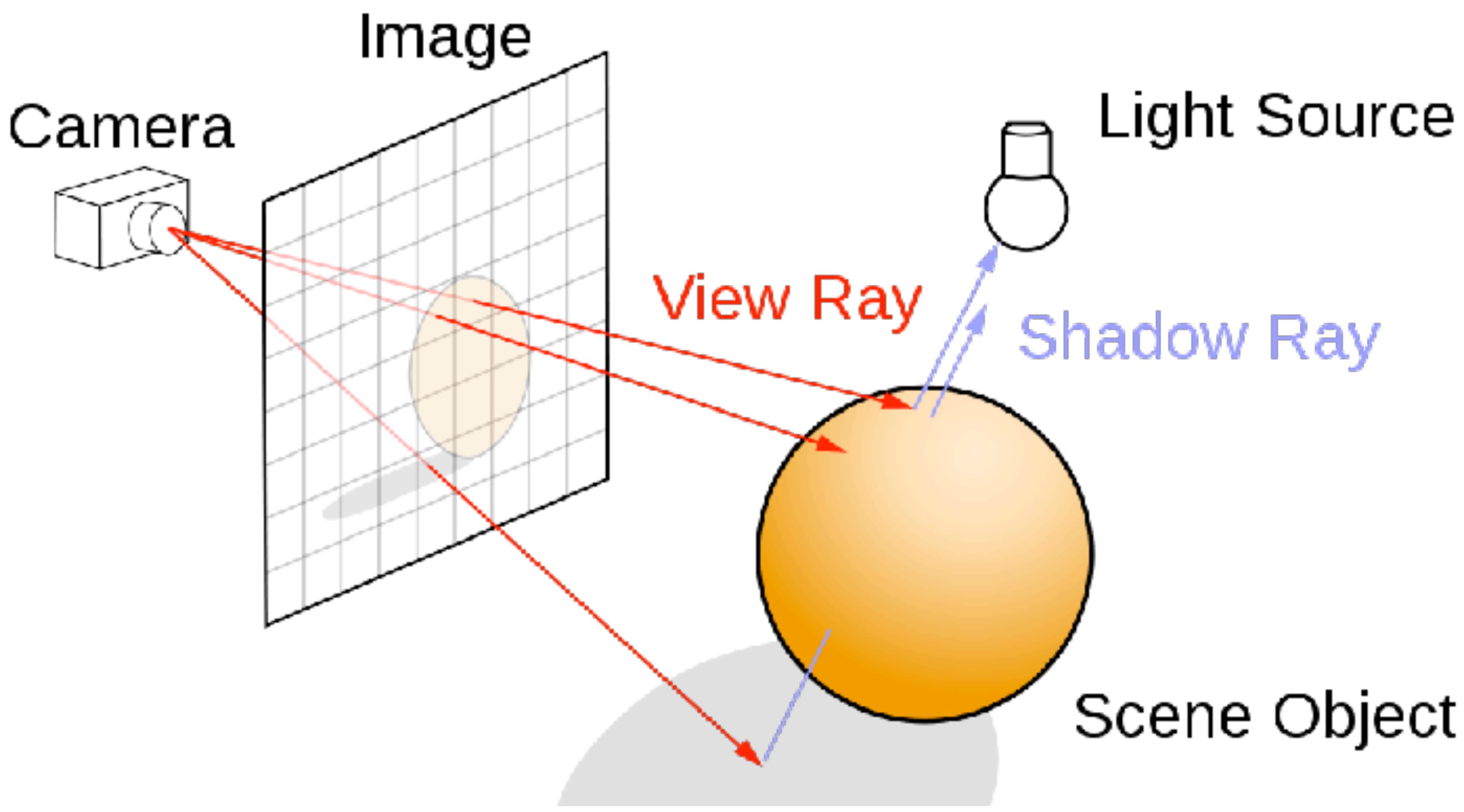
$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + \sum k_d\, I_i \max(0, \mathbf{n} \cdot \boldsymbol{\ell}_i) + k_s\, I_i \max(0, \mathbf{n} \cdot \mathbf{h}_i)^p
\end{aligned}
$$

Rasterization vs. Ray tracing

for each *shape*:
    for each *sample*:
        get *point* where *shape* covers *sample*
        if *point* is closest point seen by *sample*:
            sample.colour = shade(*point*)



for each *sample*:
    for each *shape*:
        get *point* where *shape* covers *sample*
        if *point* is closest point seen by *sample*:
            sample.colour = shade(*point*)



Ray tracing

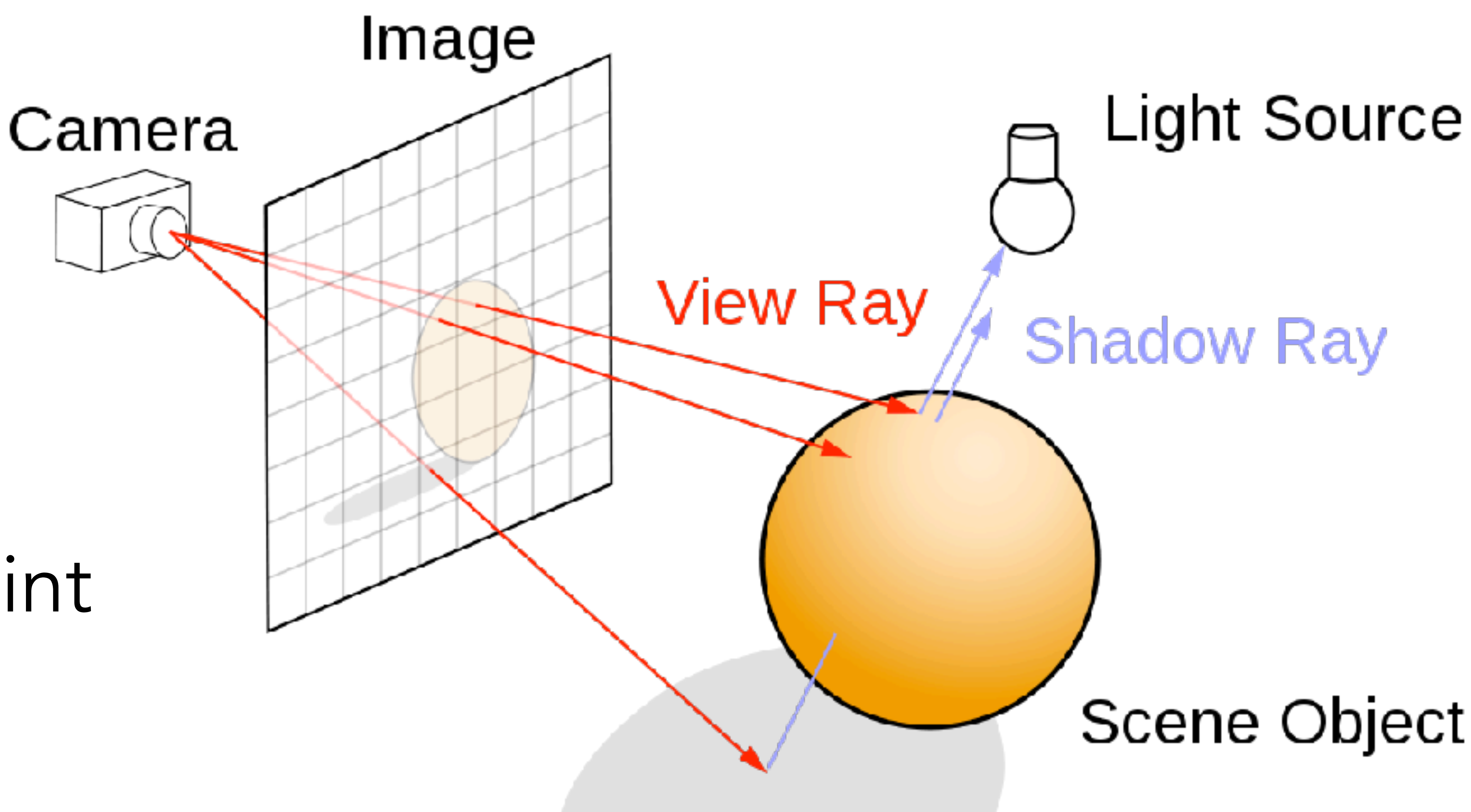
For each sample:

Generate eye ray

Find the closest intersection

Get shaded colour at intersection point

Set sample colour to it



Ray tracing

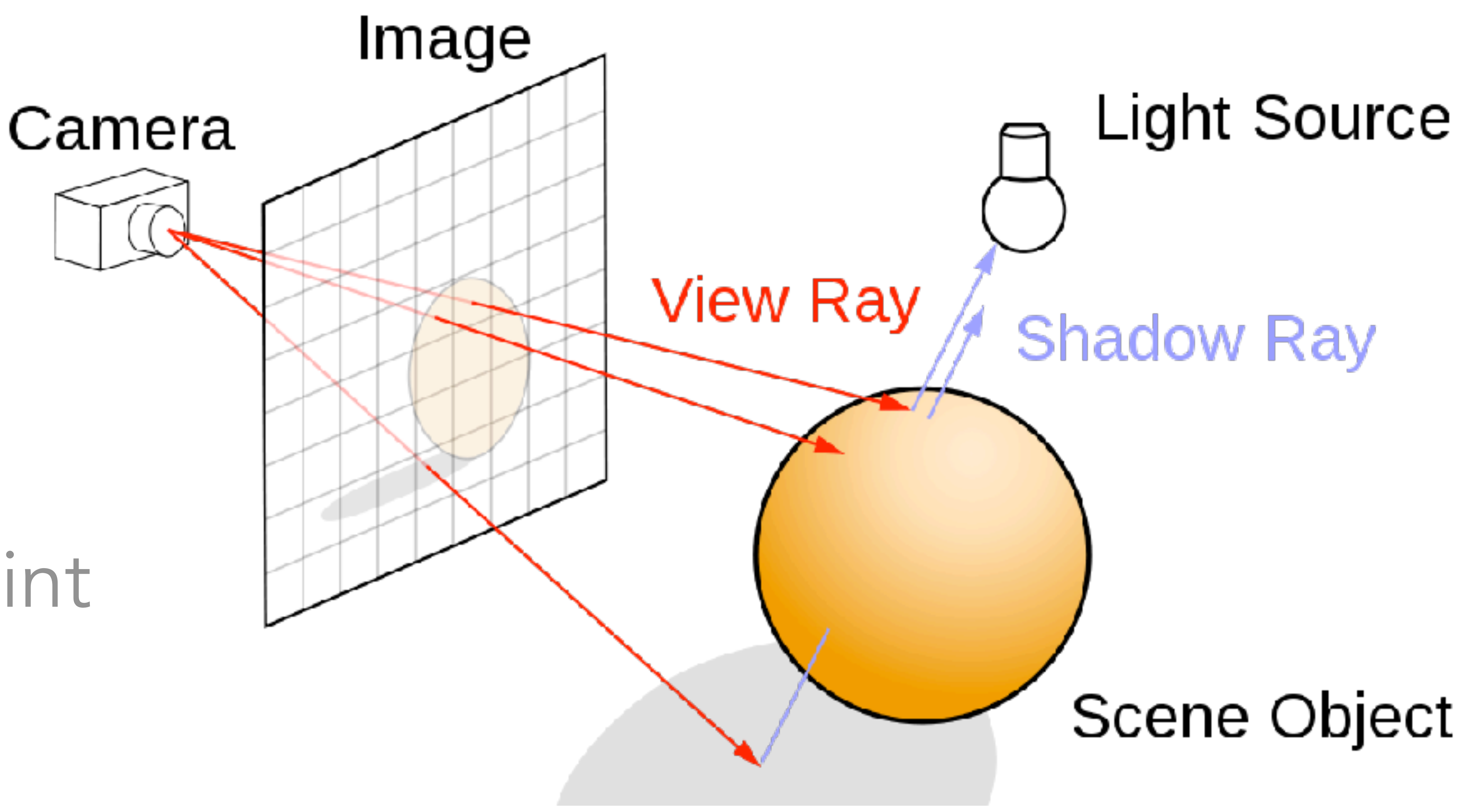
For each sample:

Generate eye ray

Find the closest intersection

Get shaded colour at intersection point

Set sample colour to it

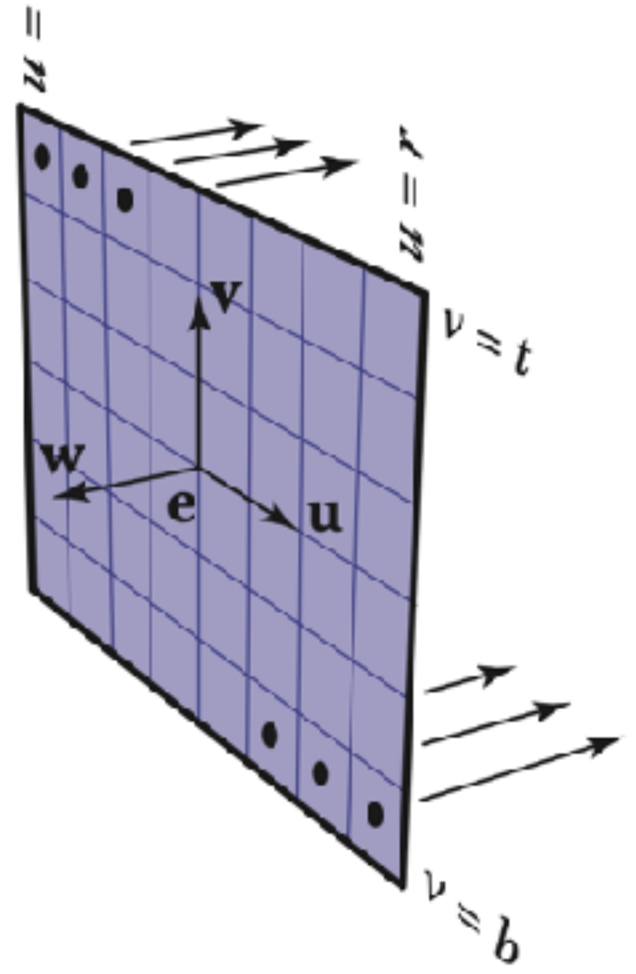


Ray generation

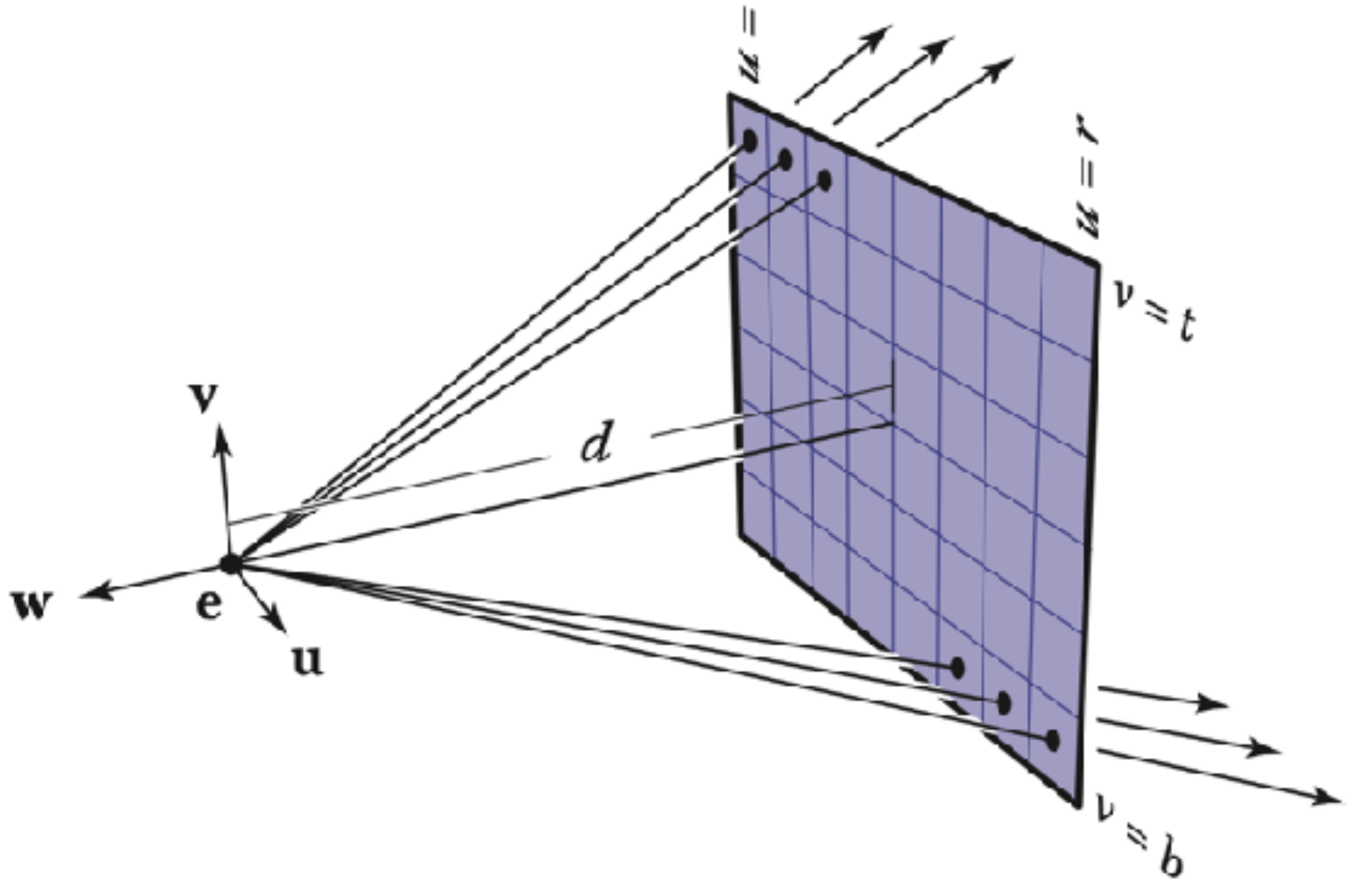
A ray is determined by an origin **o** and a direction **d**.
Any point on the ray is $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ for $t \geq 0$

Each image pixel corresponds to a ray going into the world

- Vertex shader:
  world point → image point
- Ray generation:
  image point → world ray



Parallel projection
same direction, different origins



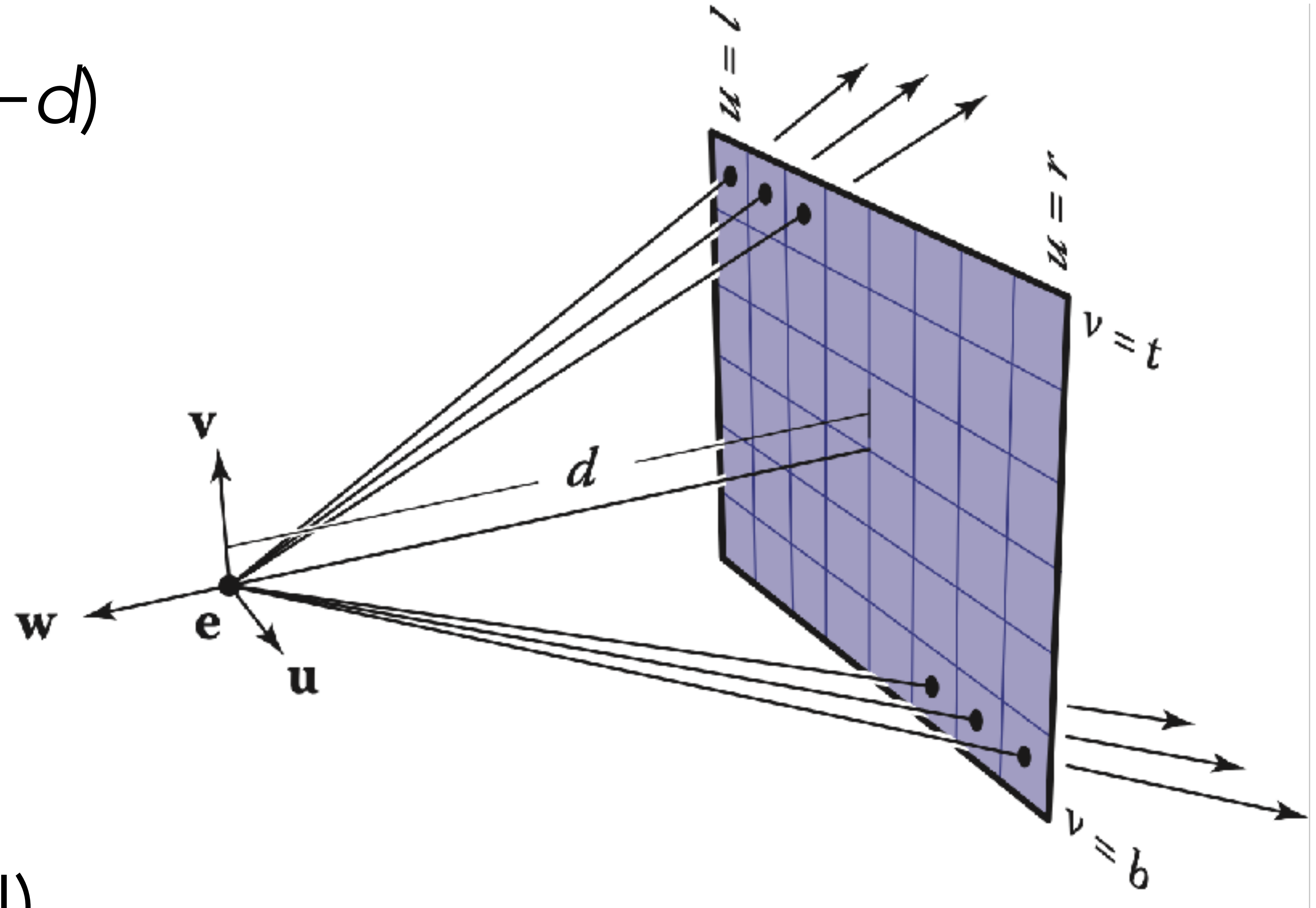
Perspective projection
same origin, different directions

Perspective camera:

- Pixel $(i, j) \to$ image plane (u, v)
- In camera space, $\mathbf{o} = (0, 0, 0)$, $\mathbf{d} = (u, v, -d)$
- Transform to world space using

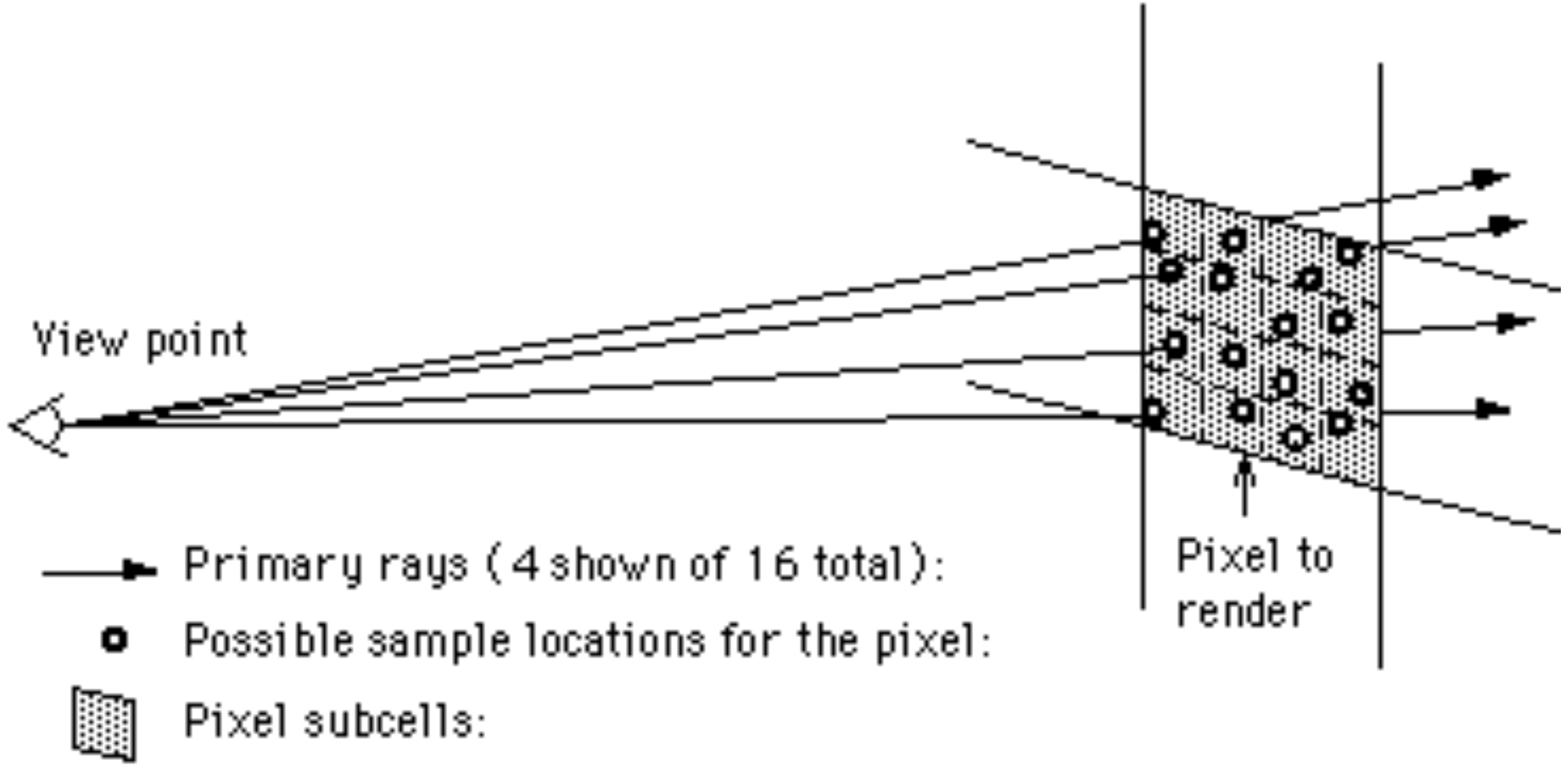
$$\mathbf{M}_{\text{view}} = \begin{bmatrix} | & | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ | & | & | & | \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(**Note:** We will not assume **d** is normalized!)



Perspective projection
same origin, different directions

Supersampling is trivial: shoot multiple rays per pixel and average the results



Paul Bourke

A camera is just a device for mapping image locations (i, j) to rays $\mathbf{o} + t\mathbf{d}$

Arbitrary pixel/ray mappings are possible:

Seitz and Kim 2003

Ray tracing

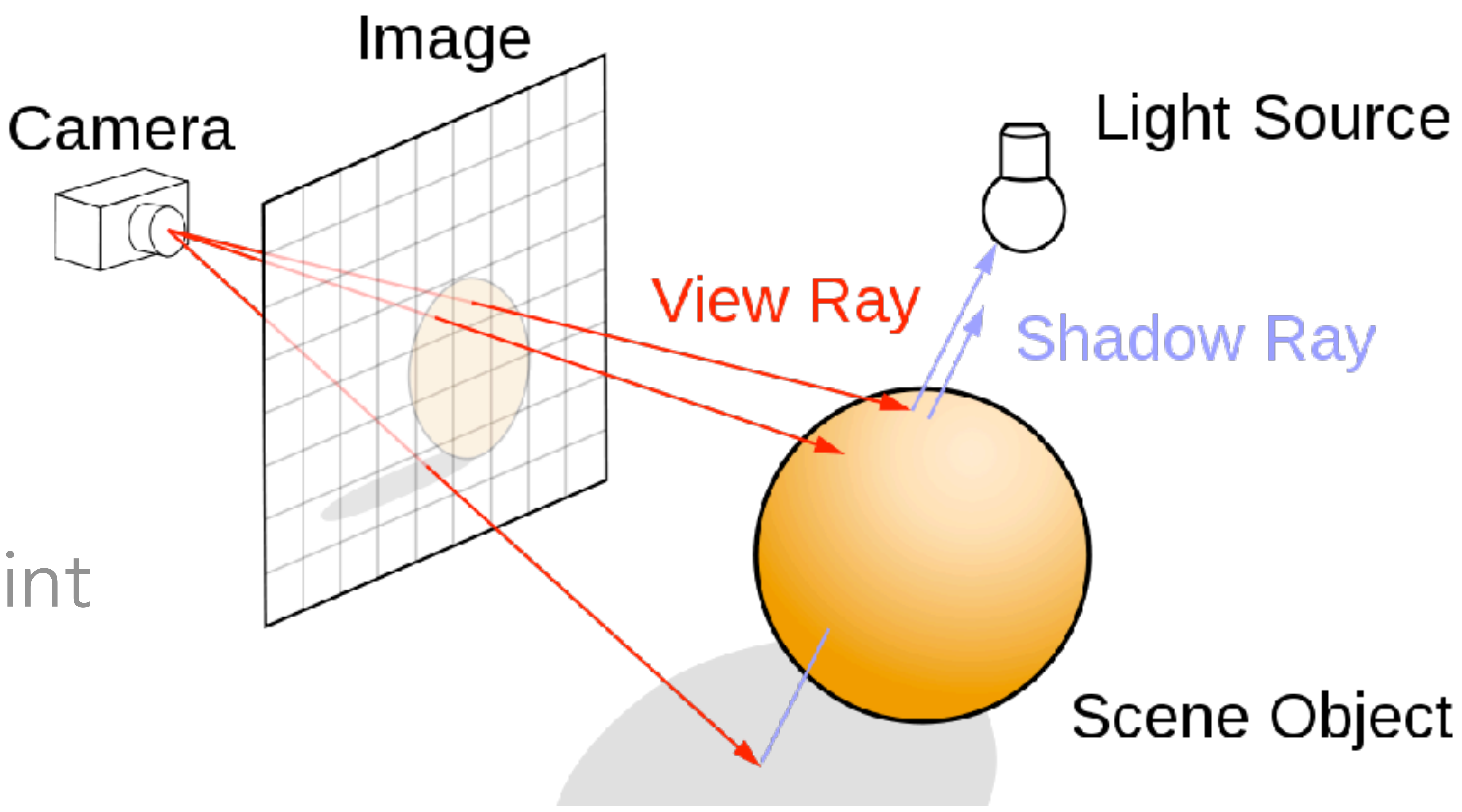
For each sample (x, y):

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

Find the closest intersection

Get shaded colour at intersection point

Set sample colour to it



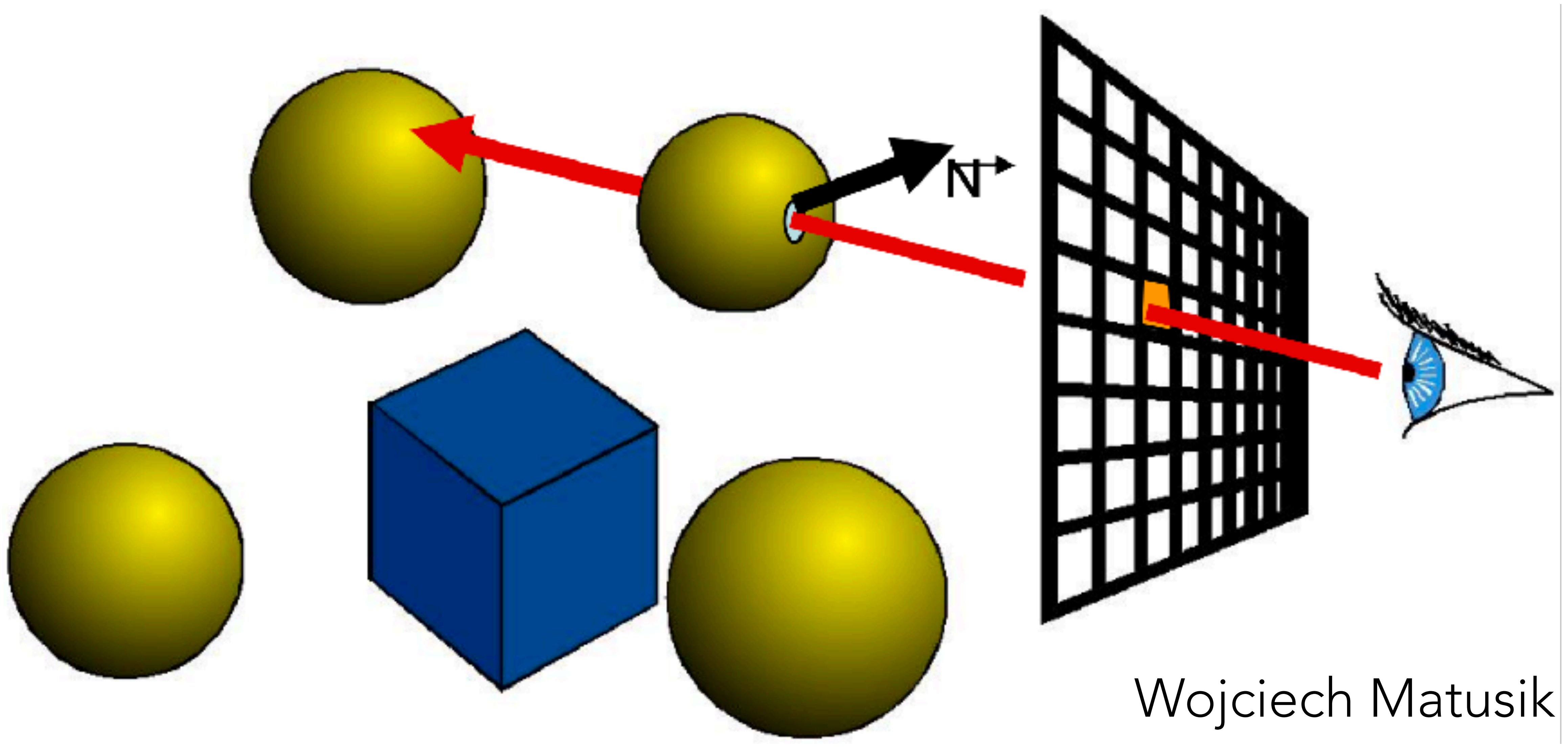
Ray-surface intersection

Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$, find closest intersection i.e. minimum t

Return info needed for shading:

- Position $\mathbf{p}$
- Normal $\mathbf{n}$
- Object ID / material properties

(Roughly the same data you would need in a fragment shader)



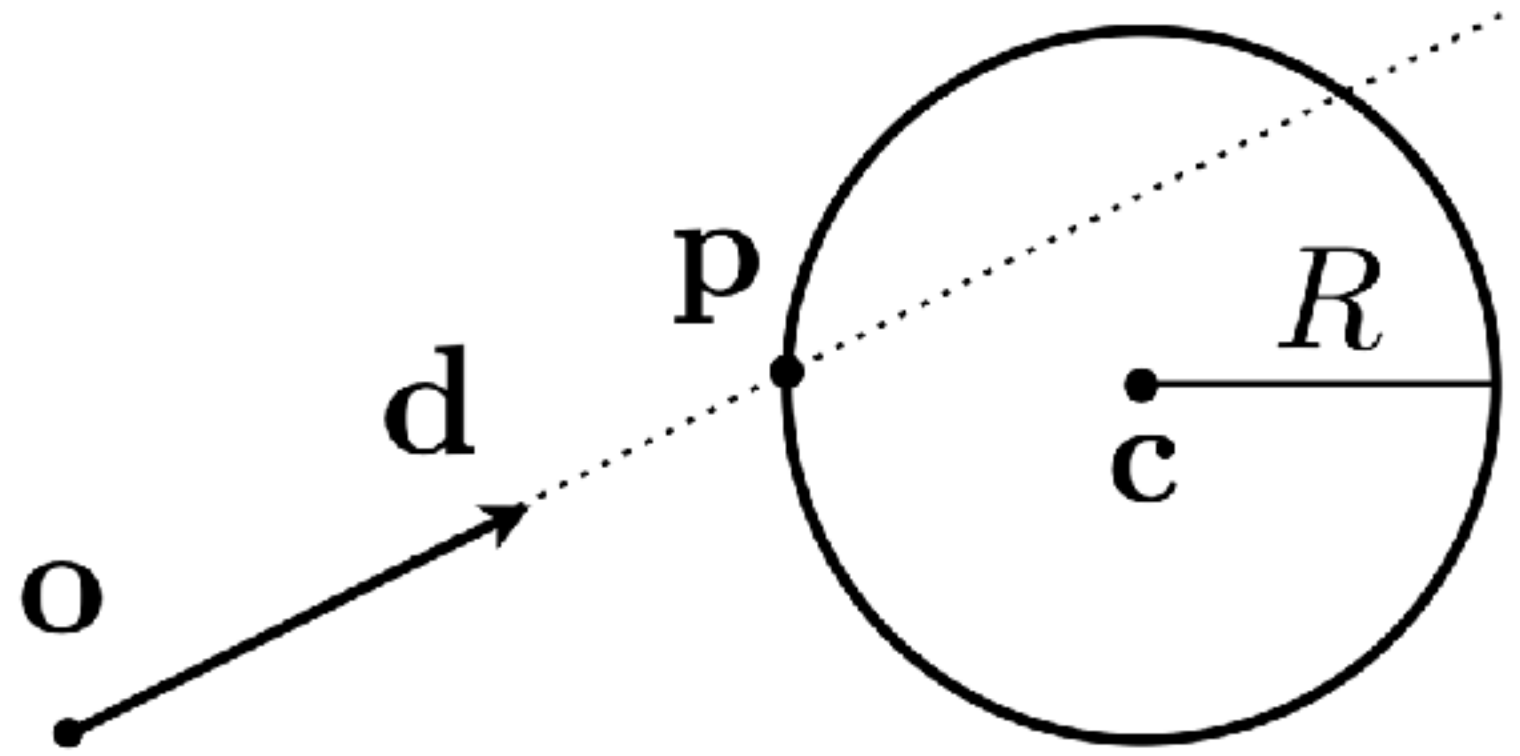
Wojciech Matusik

Ray-sphere intersection

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

Sphere equation: $||\mathbf{p} - \mathbf{c}||^2 = R^2$

Intersection point must satisfy both:



$$||(\mathbf{o} - \mathbf{c}) + t\mathbf{d}||^2 = R^2$$

$$||\mathbf{d}||^2\, t^2 + 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})\, t + ||\mathbf{o} - \mathbf{c}||^2 - R^2 = 0$$

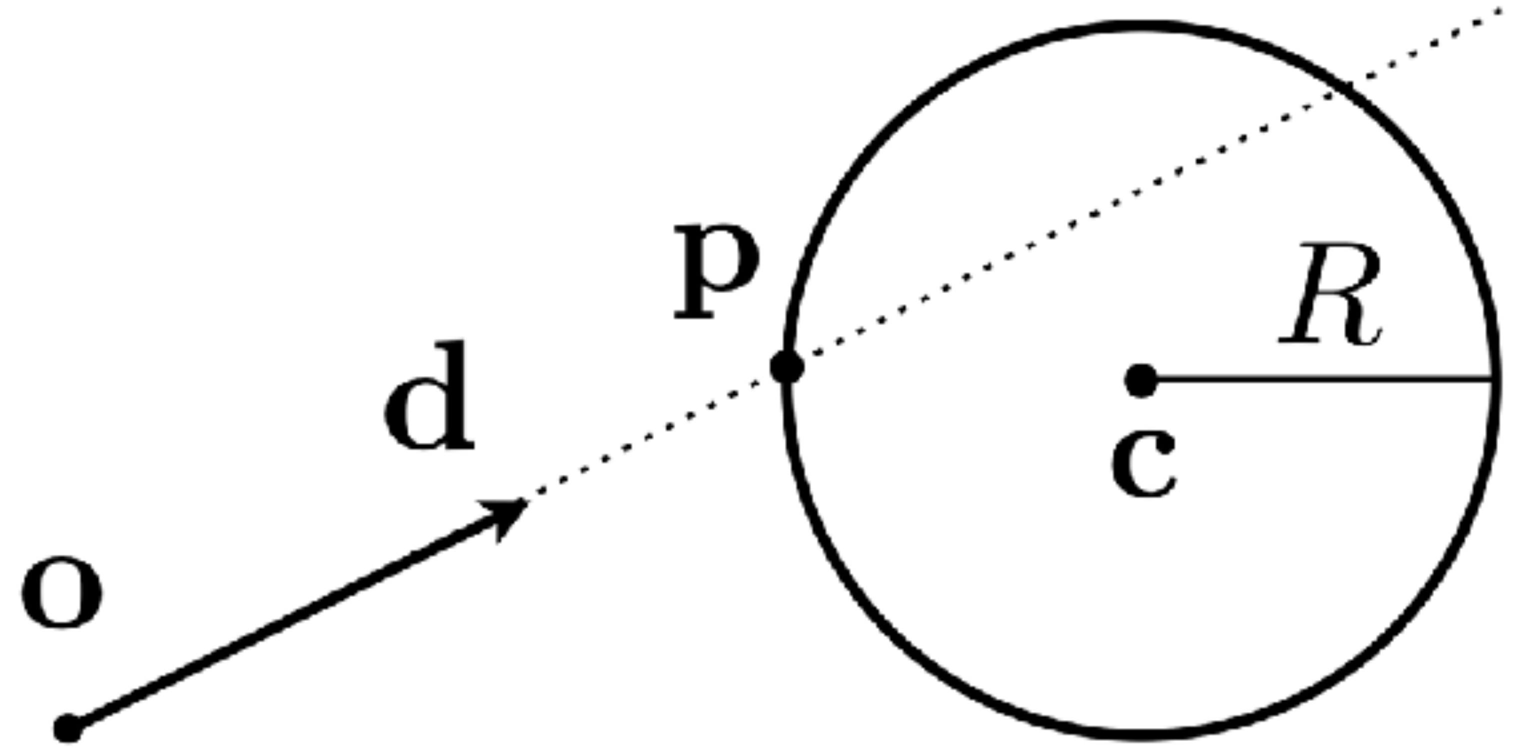
(Recall $||\mathbf{v}||^2 = \mathbf{v} \cdot \mathbf{v}$)

Quadratic equation, solve for t

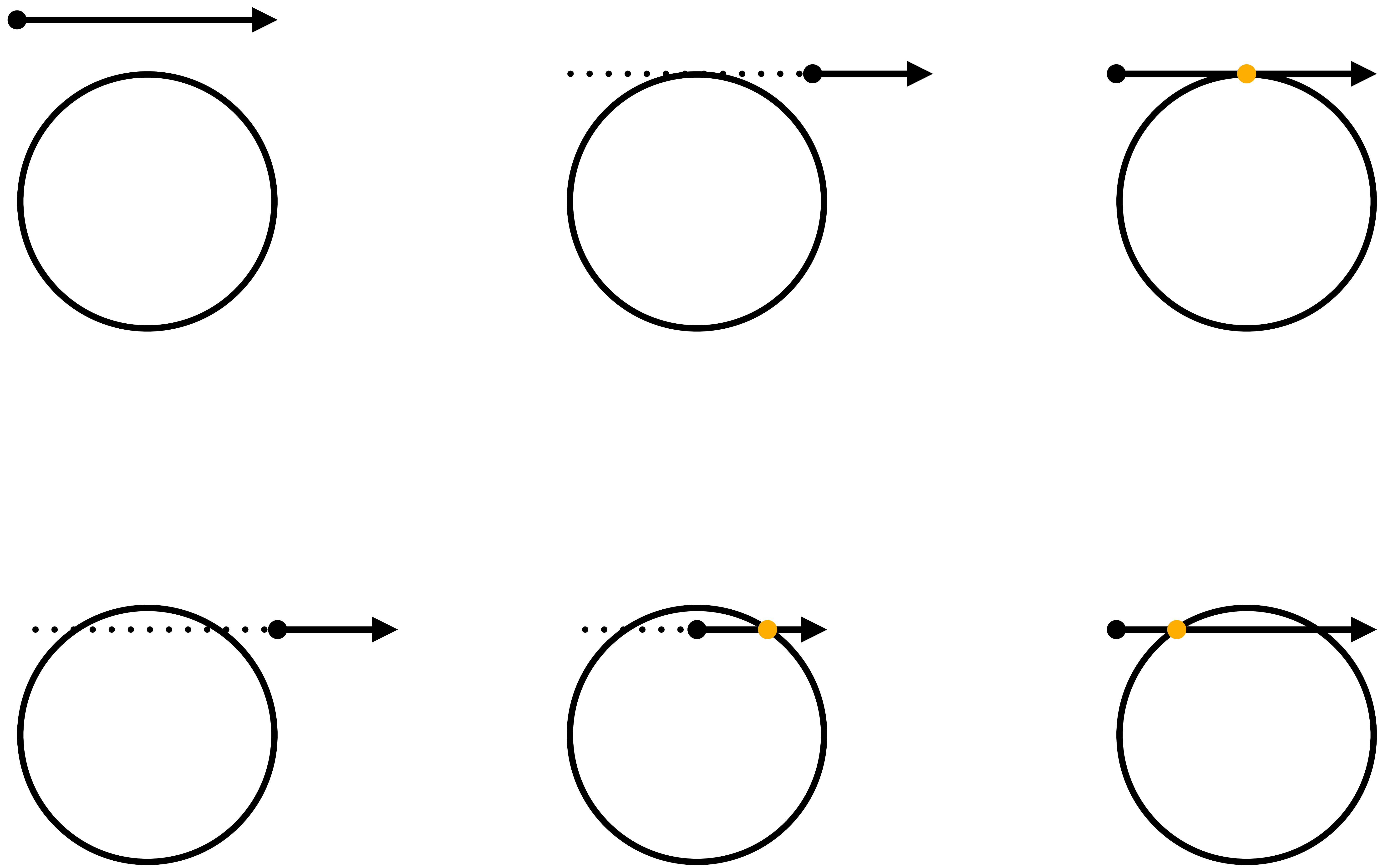
3 cases:

- No solution
- One solution t_1
- Two solutions t_1 and t_2

What do they mean geometrically?



- No solution
- One solution t_1
  - $t_1 < 0$
  - $t_1 > 0$
- Two solutions t_1 and t_2
  - $t_1 < t_2 < 0$
  - $t_1 < 0 < t_2$
  - $0 < t_1 < t_2$



In general: Find all solutions, discard those with $t < 0$, take minimum of remaining

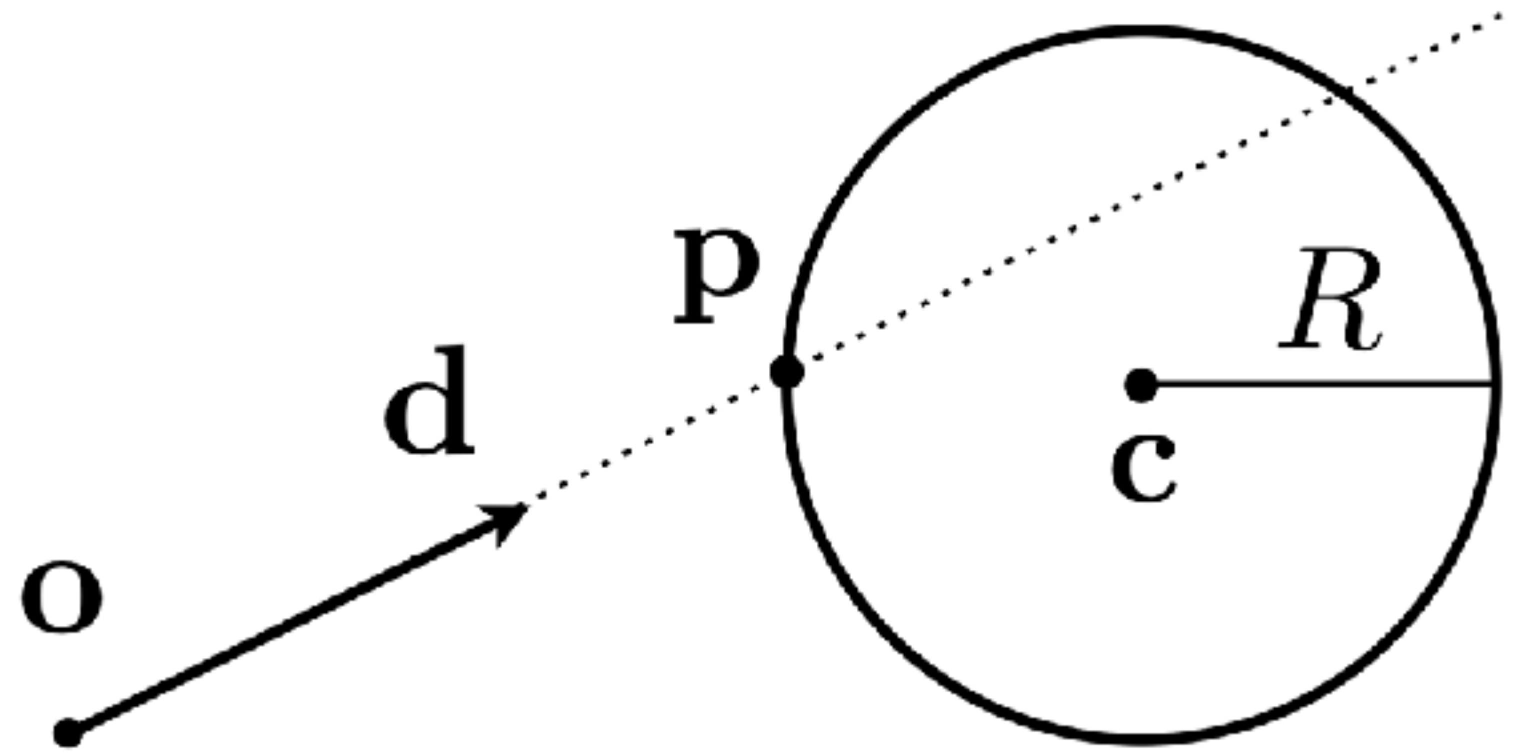
Find t of closest intersection

Then get intersection point from equation of ray:

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

What about the surface normal?

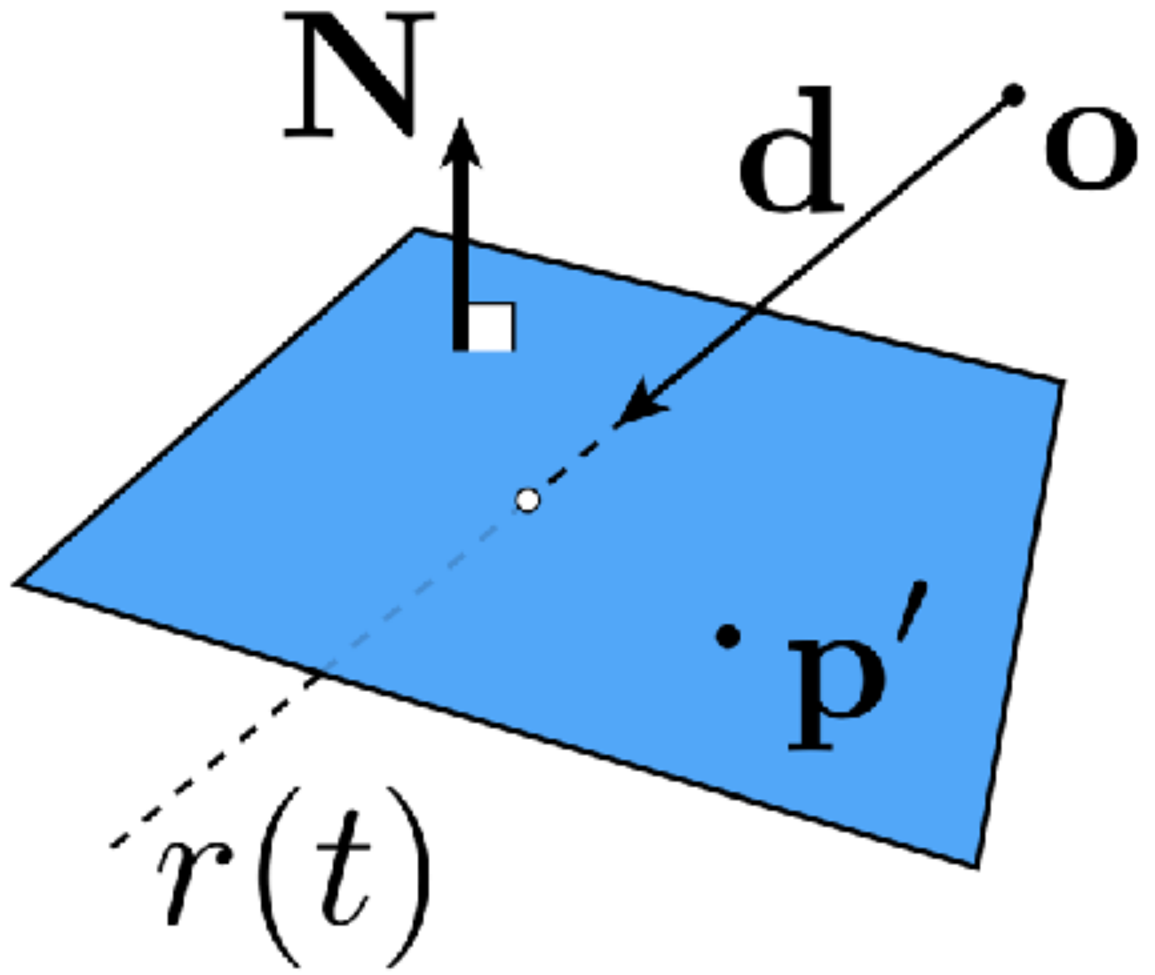
$$\mathbf{n} = (\mathbf{p} - \mathbf{c})/||\mathbf{p} - \mathbf{c}||$$



Ray-plane intersection

Plane equation: $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

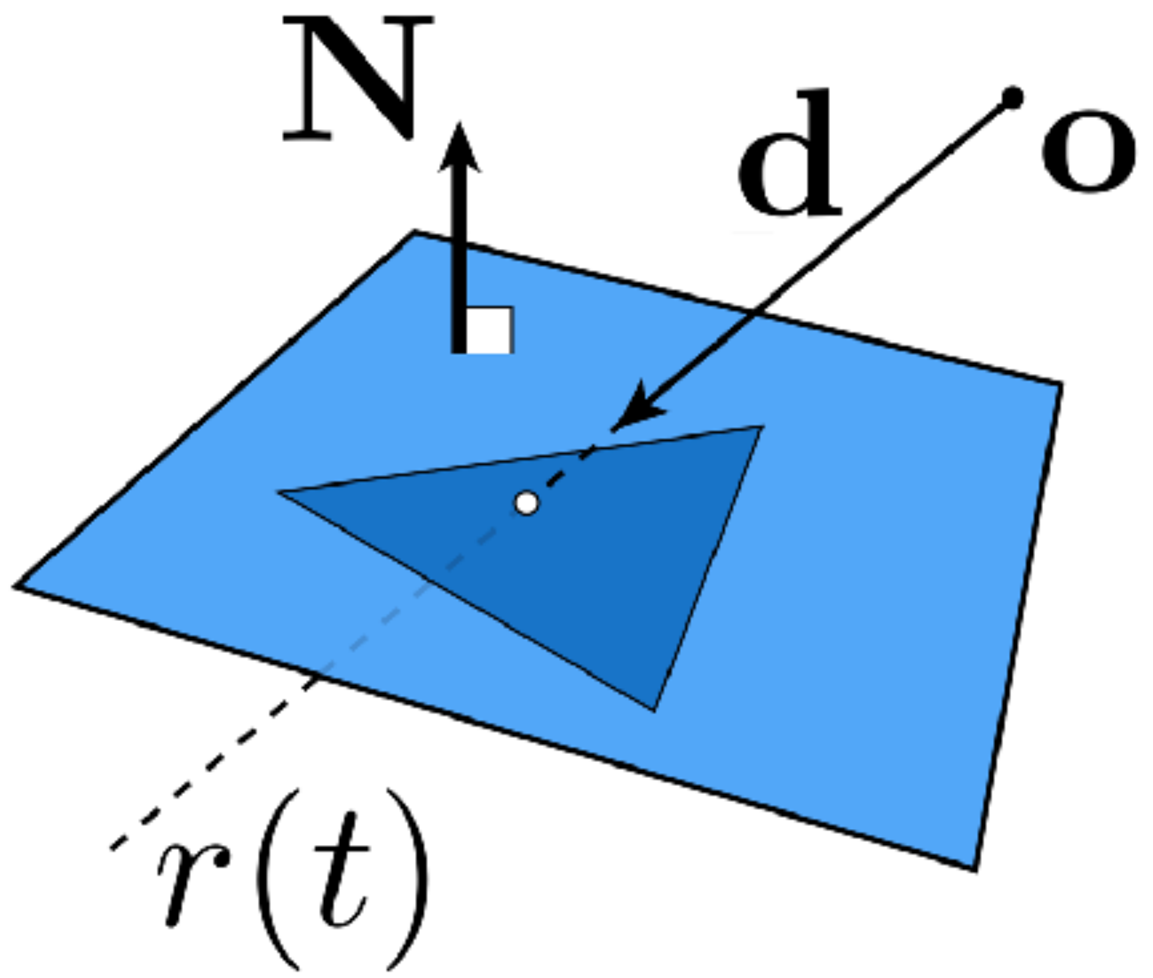
any known point
on the plane



$$\mathbf{n} \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{p}_0) = 0$$

$$t = (\mathbf{n} \cdot (\mathbf{p}_0 - \mathbf{o}))/(\mathbf{n} \cdot \mathbf{d})$$

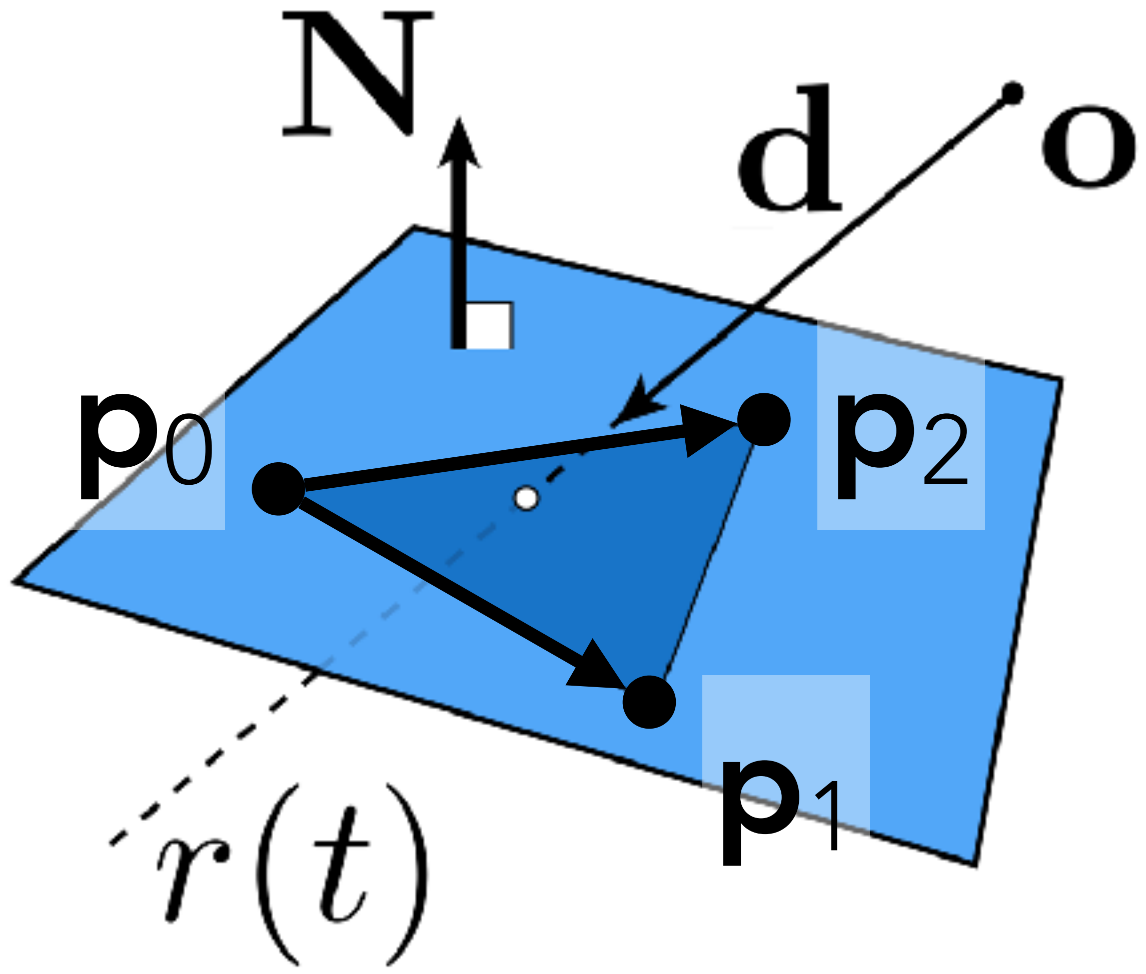
Ray-triangle intersection



Intersect ray with plane, then check if it is inside triangle?

A better way: Any point on the plane is

$$\mathbf{p} = \mathbf{p}_0 + b_1(\mathbf{p}_1 - \mathbf{p}_0) + b_2(\mathbf{p}_2 - \mathbf{p}_0)$$
$$= (1-b_1-b_2)\mathbf{p}_0 + b_1\mathbf{p}_1 + b_2\mathbf{p}_2$$



$$\mathbf{o} + t\mathbf{d} = \mathbf{p}_0 + b_1(\mathbf{p}_1 - \mathbf{p}_0) + b_2(\mathbf{p}_2 - \mathbf{p}_0)$$

3 equations in 3 unknowns:

$$\begin{bmatrix} | & | & | \\ -\mathbf{d} & \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \\ | & | & | \end{bmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} | \\ \mathbf{o} - \mathbf{p}_0 \\ | \end{bmatrix}$$

Solve to get t, b_1, b_2. For what values of b_1, b_2 is the point inside the triangle?

Fastest classic method to solve: Cramer's rule

$$\begin{bmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \mathbf{t}$$

$$\implies \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\det \begin{bmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix}} \begin{bmatrix} \det \begin{bmatrix} \mathbf{t} & \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{d} & \mathbf{t} & \mathbf{e}_2 \end{bmatrix} \\ \det \begin{bmatrix} -\mathbf{d} & \mathbf{e}_1 & \mathbf{t} \end{bmatrix} \end{bmatrix}$$

$$= \frac{1}{\mathbf{p} \cdot \mathbf{e}_1} \begin{bmatrix} \mathbf{q} \cdot \mathbf{e}_2 \\ \mathbf{p} \cdot \mathbf{t} \\ \mathbf{q} \cdot \mathbf{d} \end{bmatrix}$$ where $\mathbf{p} = \mathbf{d} \times \mathbf{e}_2$, $\mathbf{q} = \mathbf{t} \times \mathbf{e}_1$

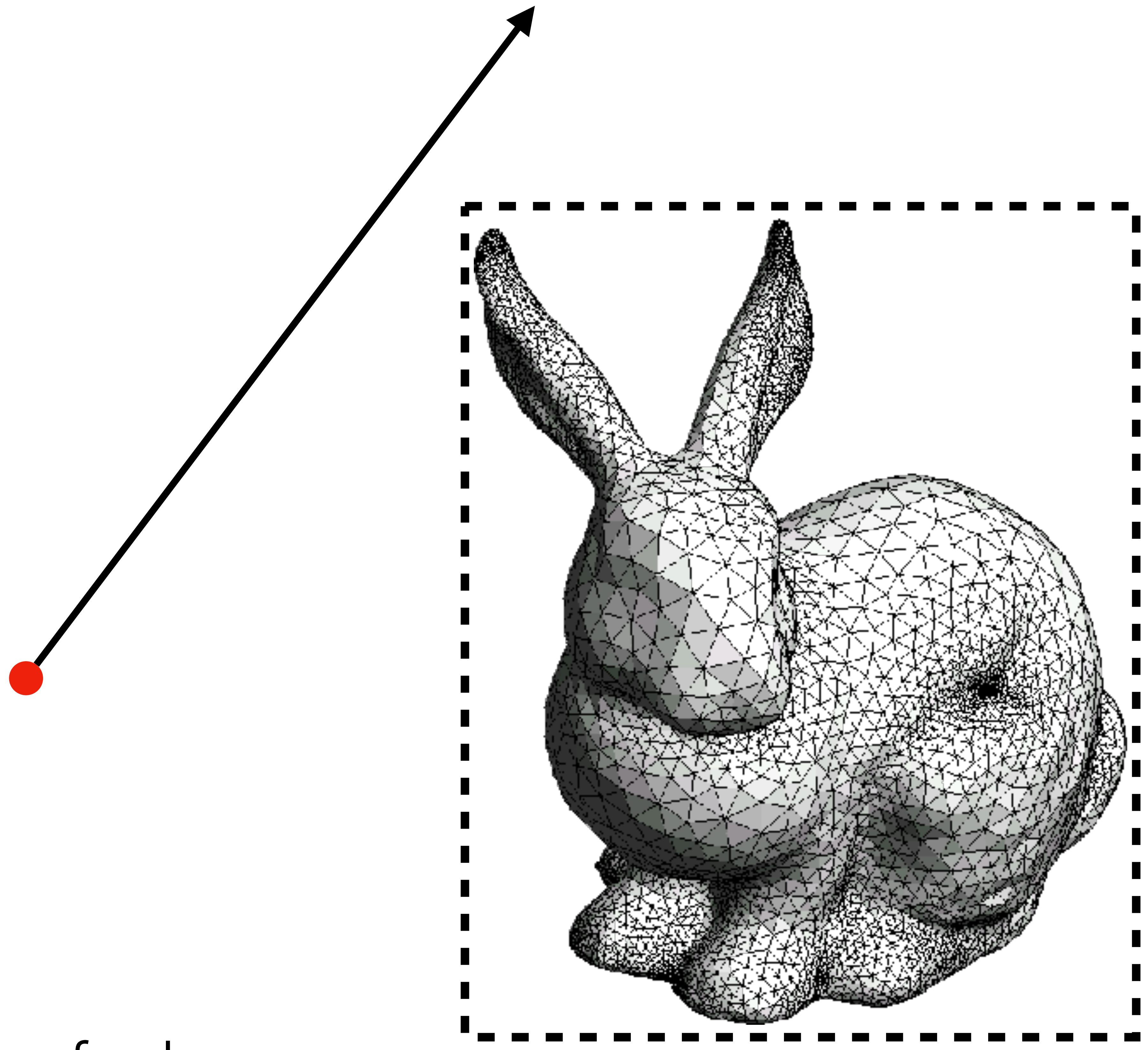
Ray-mesh intersection

Naïve approach: Test ray with all triangles, return the earliest hit.

Cost = O(#triangles)! Can we speed it up?

Construct a conservative bounding volume: all mesh triangles lie inside it

Super easy to reject rays that don't come close to intersecting the mesh.

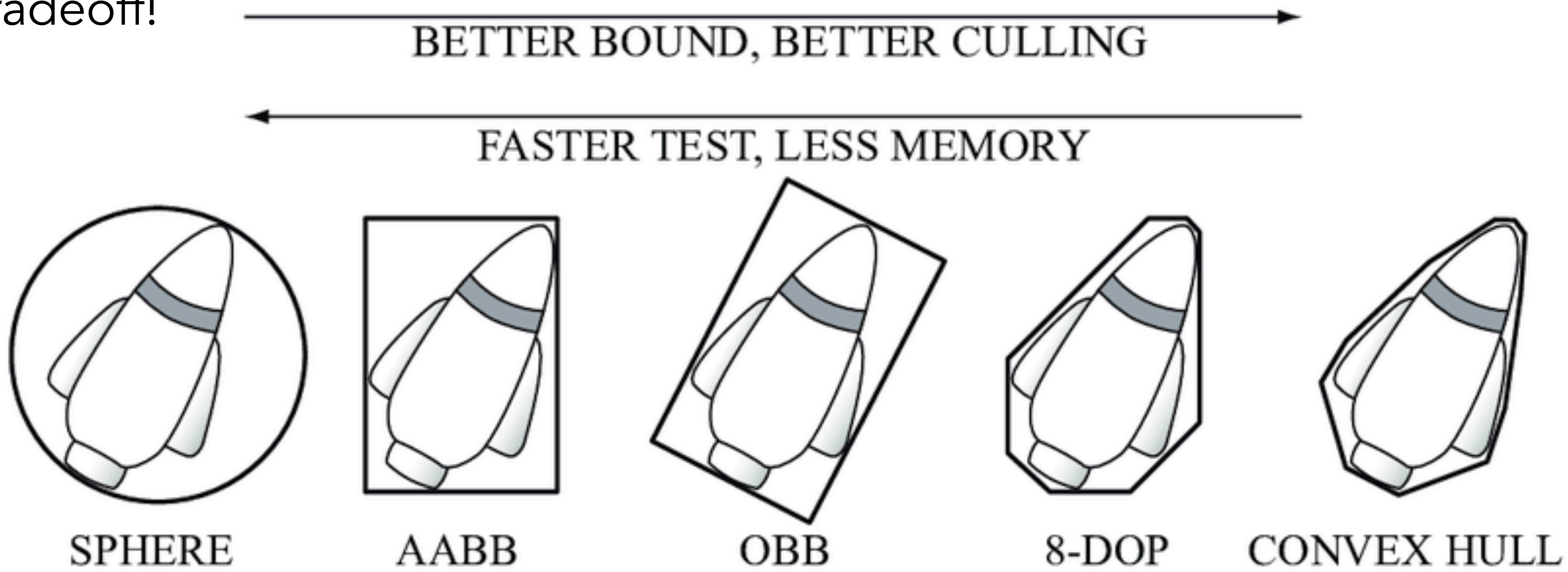


Later, we will study bounding volume hierarchies to speed things up further.

What do we want from a bounding volume?

- Tight (minimize # of false positives)
- Fast to intersect

This is a tradeoff!

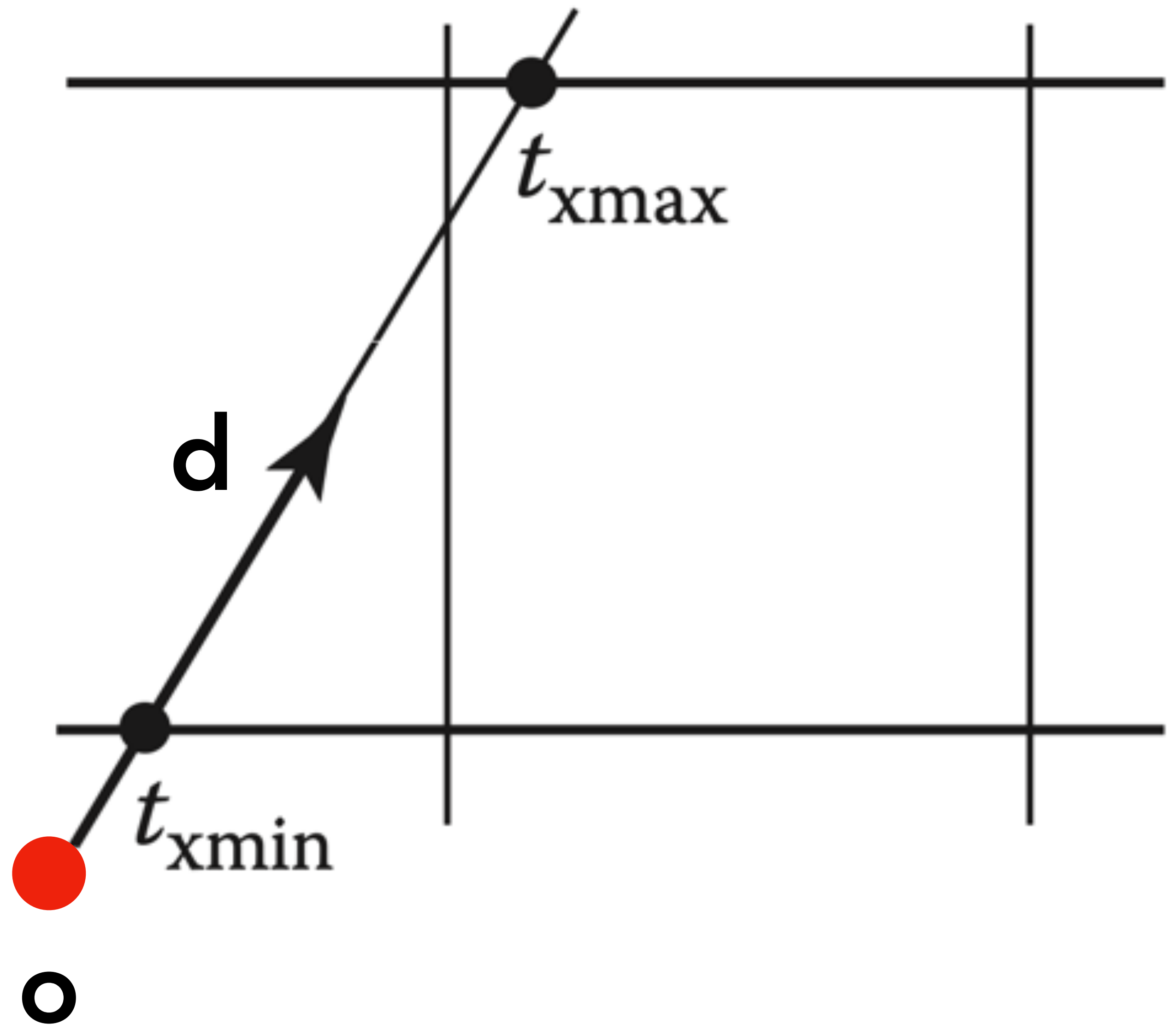


Ericson, *Real-Time Collision Detection*

Let's stick with axis-aligned bounding boxes (AABBs).

Fast ray-AABB test:

- $t_{xmin} = (x_{min} - o_x)/d_x$
- Similarly for t_{xmax}, t_{ymin}, …
- Final intersection result
  $= [t_{xmin}, t_{xmax}] \cap [t_{ymin}, t_{ymax}] \cap [t_{zmin}, t_{zmax}]$
  $= [\max(t_{xmin}, t_{ymin}, t_{zmin}), \min(t_{xmax}, t_{ymax}, t_{zmax})]$



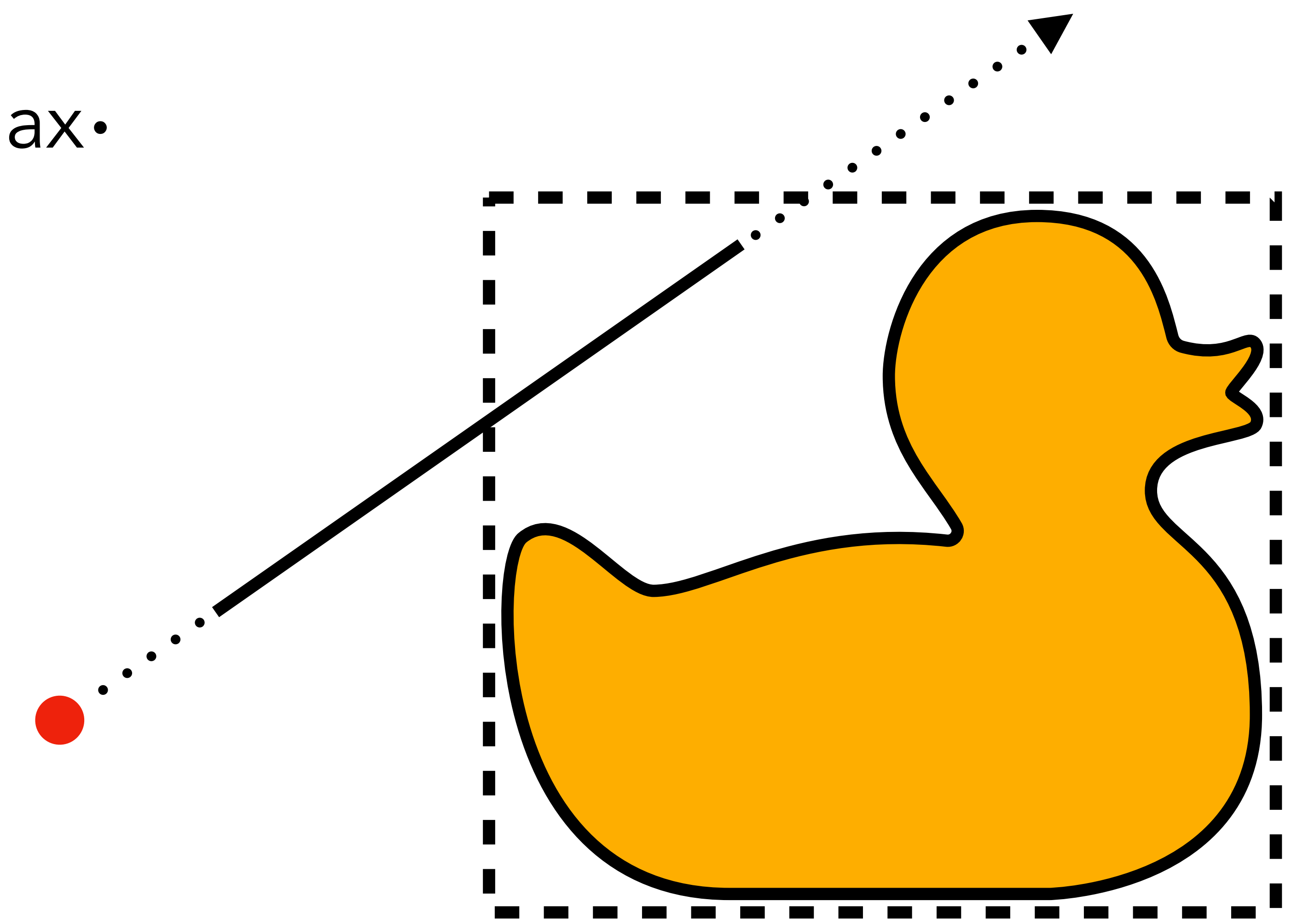
Swap the bounds first if $d_x < 0$! What about if $d_x = 0$?

How do we know if the ray misses the box?

Object-oriented raytracer design

We can ray trace any shape as long as it provides the following methods:

- `bool hit(Ray` $\mathbf{o} + t\mathbf{d}$, `real` t_{min}, `real` t_{max}, `HitRecord &rec)`
  - Only consider intersections in the range $t_{min} \leq t \leq t_{max}$. Usually $[0, \infty]$ for eye rays
  - If hit, write the position, normal, material, etc. into the `HitRecord`
- `Box bounding_box()`
  - For early exit



Transformed objects

How to ray trace a transformed shape?

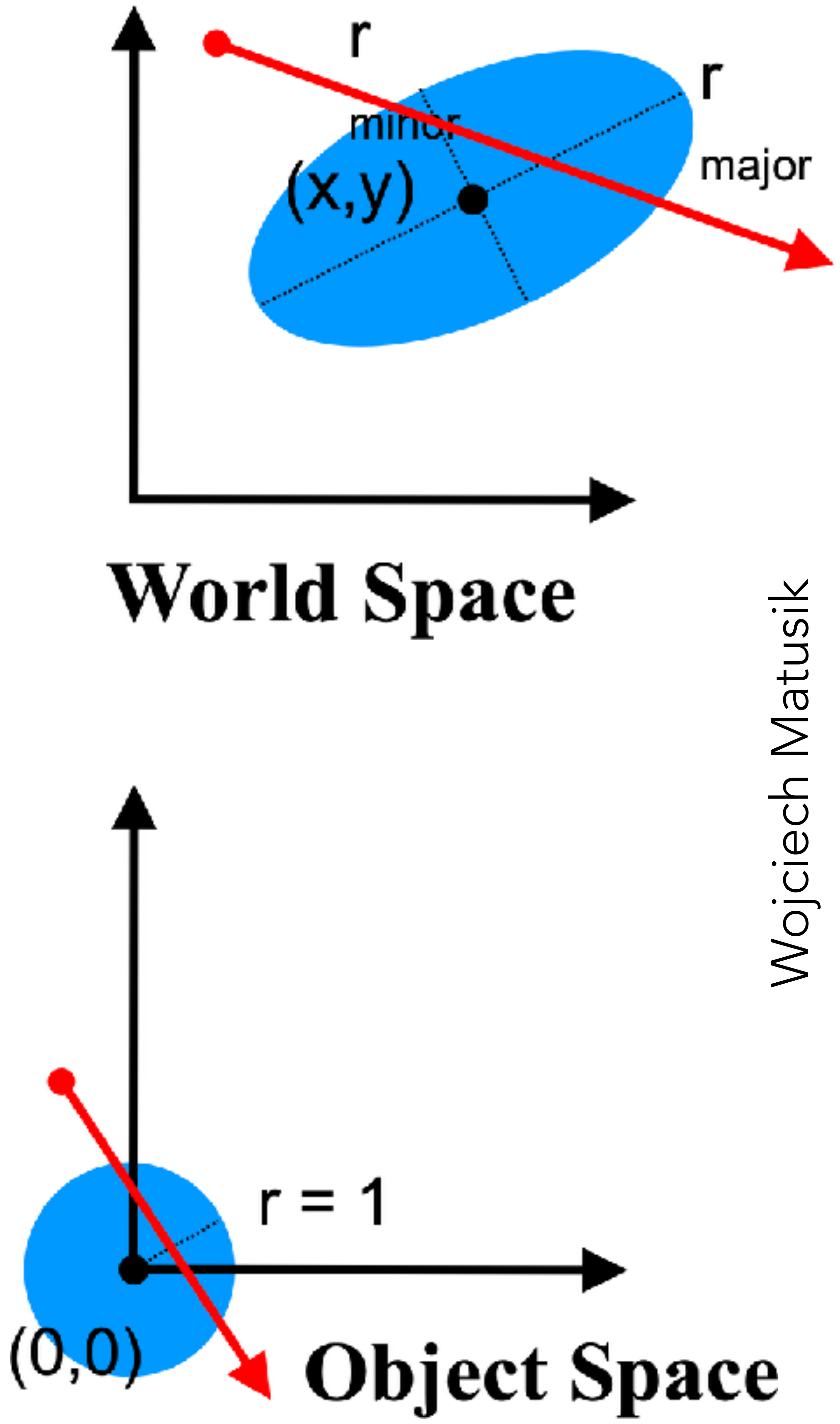
$$\mathbf{p}_{\mathrm{WS}} = \mathbf{M}\ \mathbf{p}_{\mathrm{OS}}$$

Just un-transform the ray into object space…

$$\mathbf{o}_{\mathrm{OS}} = \mathbf{M}^{-1}\ \mathbf{o}_{\mathrm{WS}}$$
$$\mathbf{d}_{\mathrm{OS}} = \mathbf{M}^{-1}\ \mathbf{d}_{\mathrm{WS}}$$

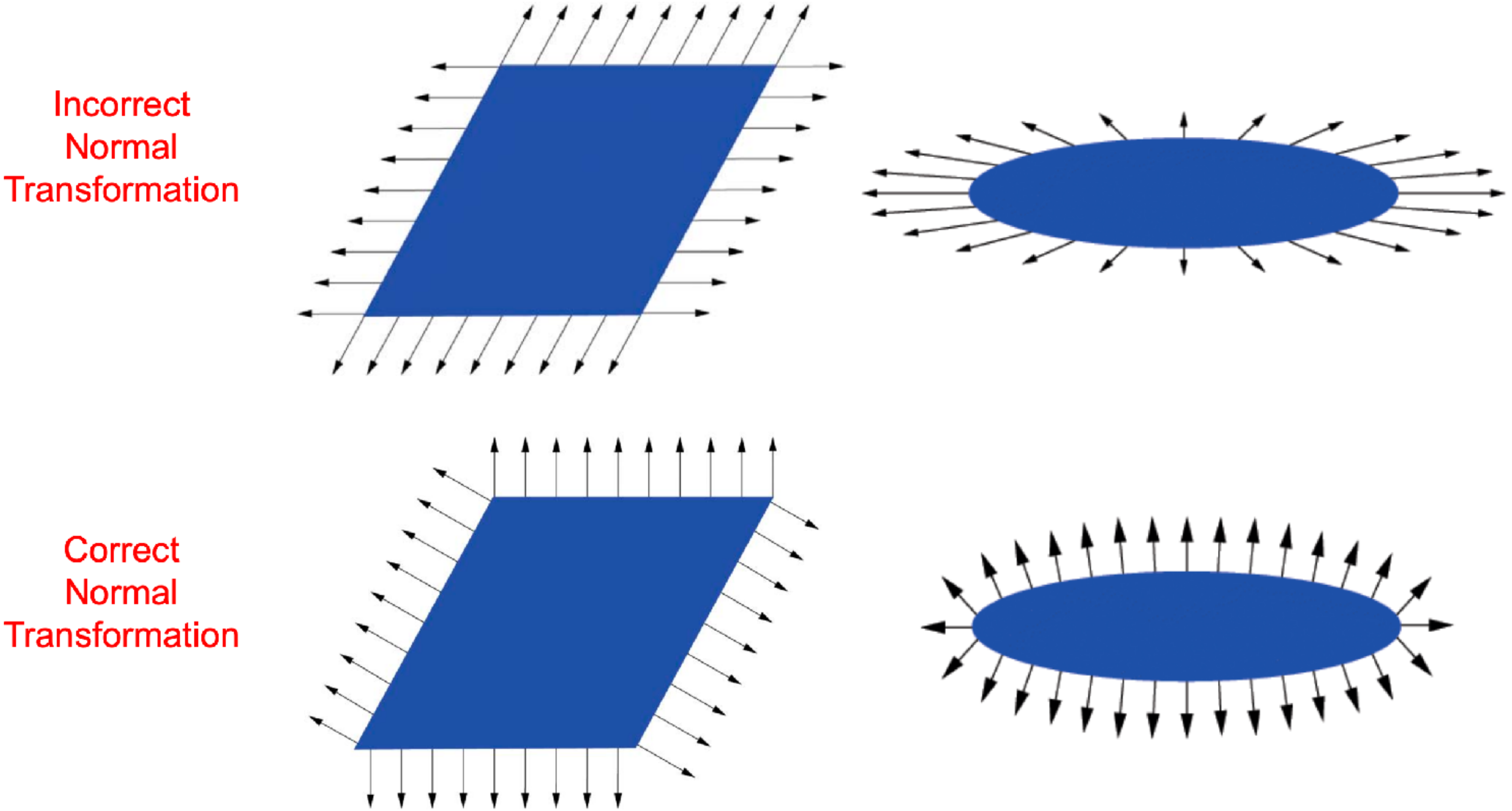
…and do the ray-shape intersection there

(This is why it's better to not assume **d** is normalized)



Wojciech Matusik

Normals don’t transform like other vectors!



Wojciech Matusik