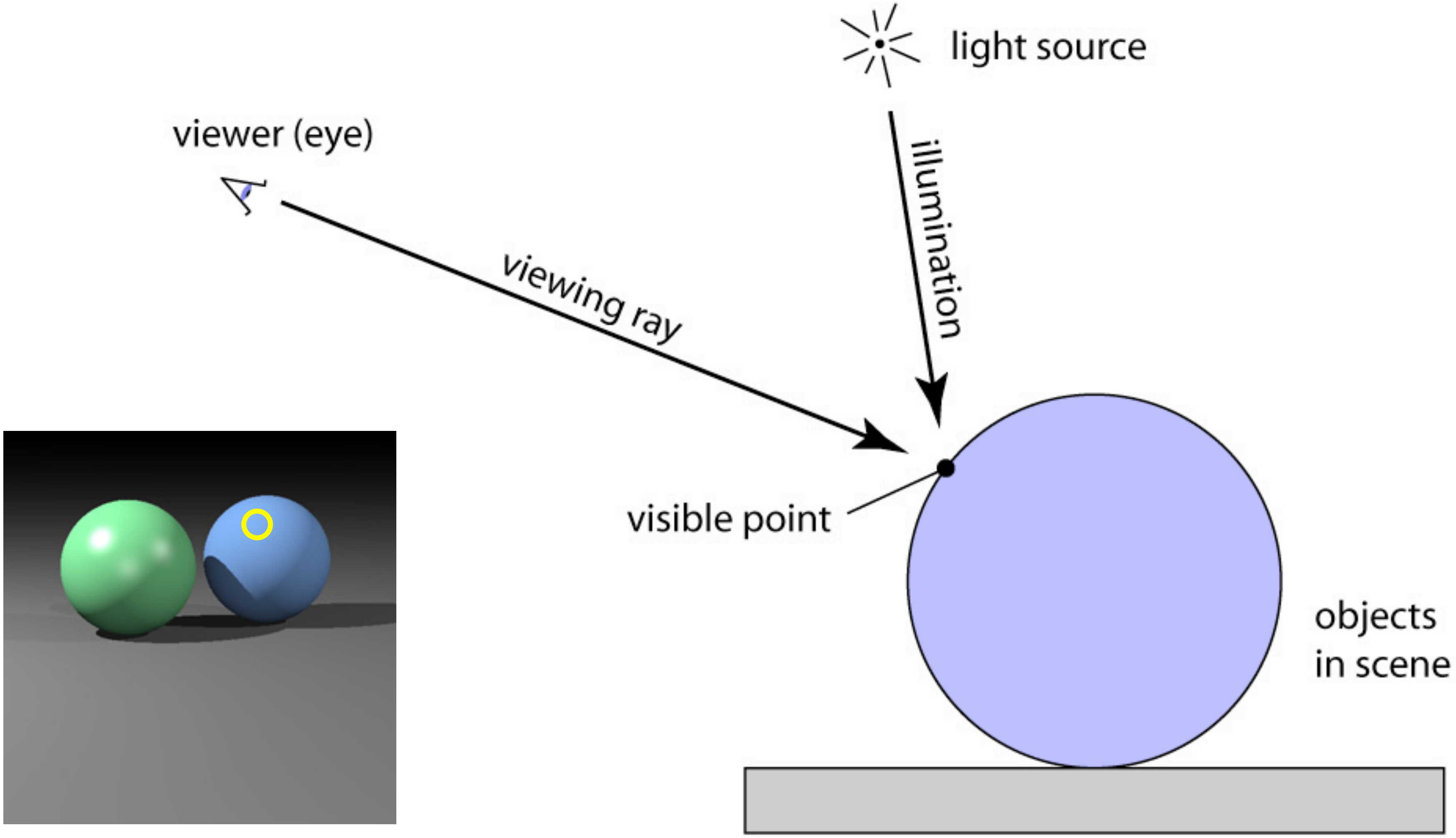
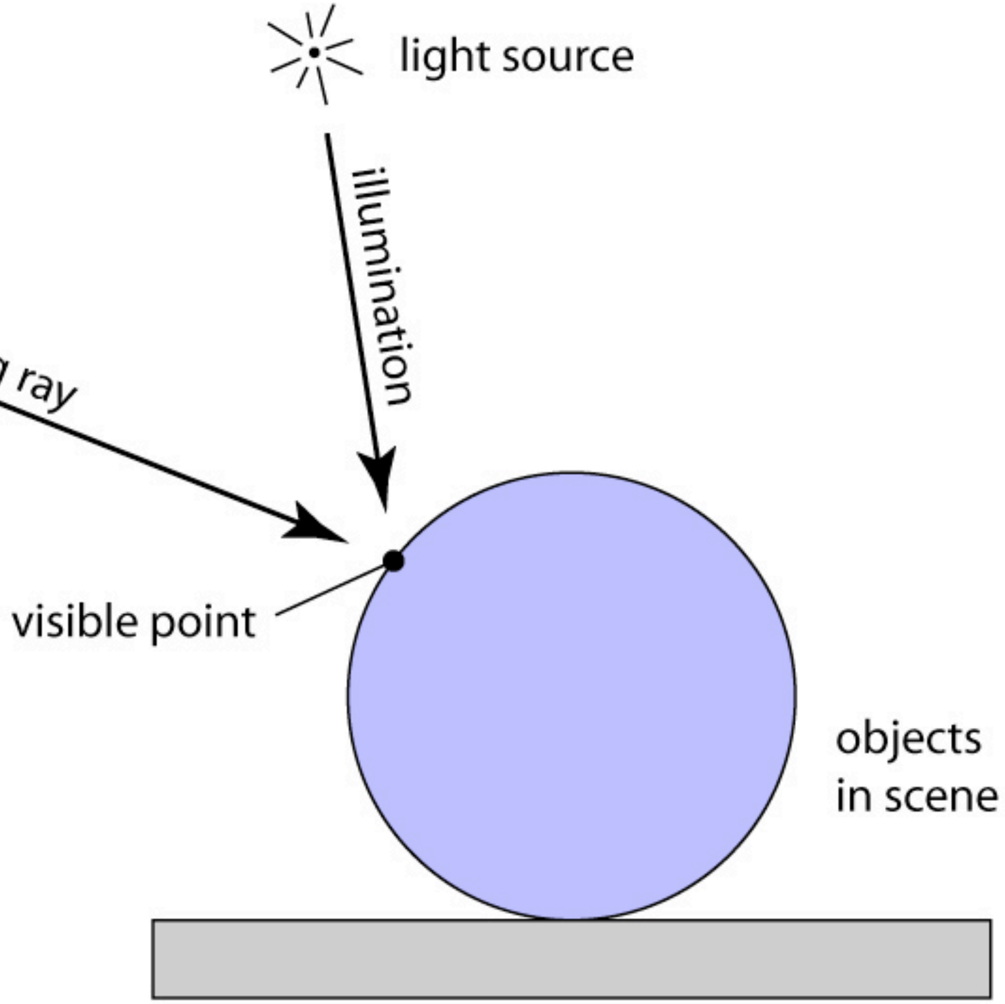
Ray Tracing

CS 465 Lecture 3

Ray tracing idea



Ray tracing algorithm

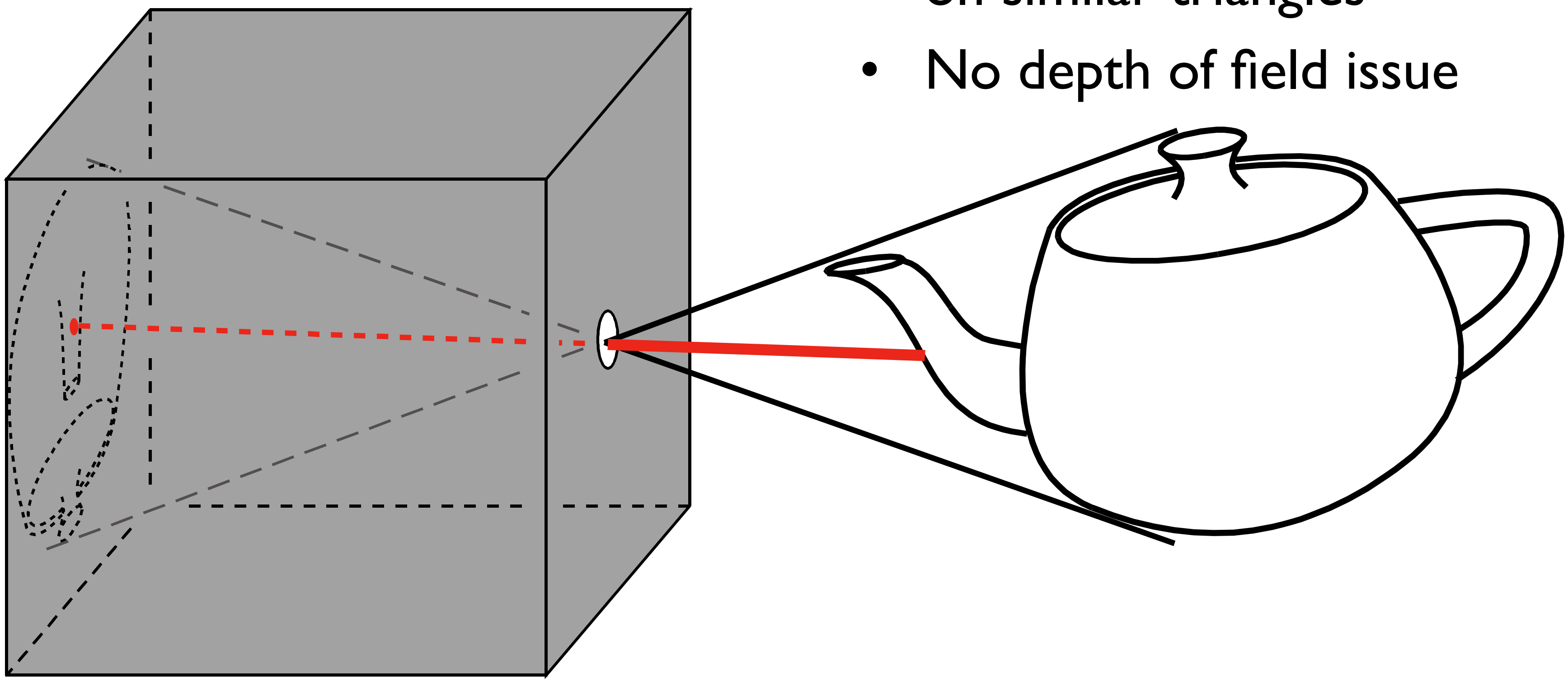


```
for each pixel {
    compute viewing ray
    intersect ray with scene
    compute illumination at visible point
    put result into image
    }
```

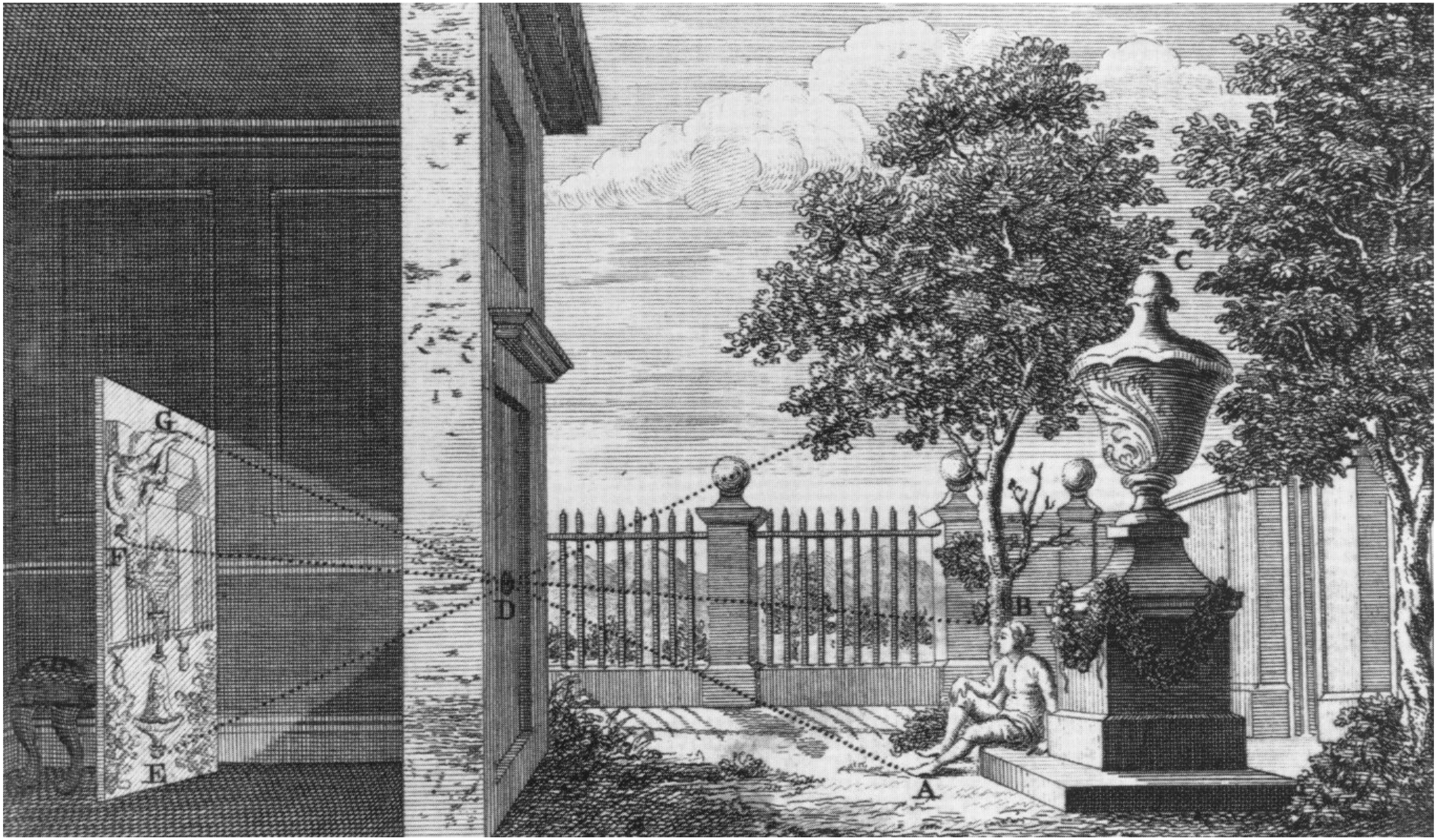
Pinhole camera

- Box with a tiny hole
- Image is inverted

- Perfect image if hole infinitely small
- Pure geometric optics based on similar triangles
- No depth of field issue



Camera Obscura

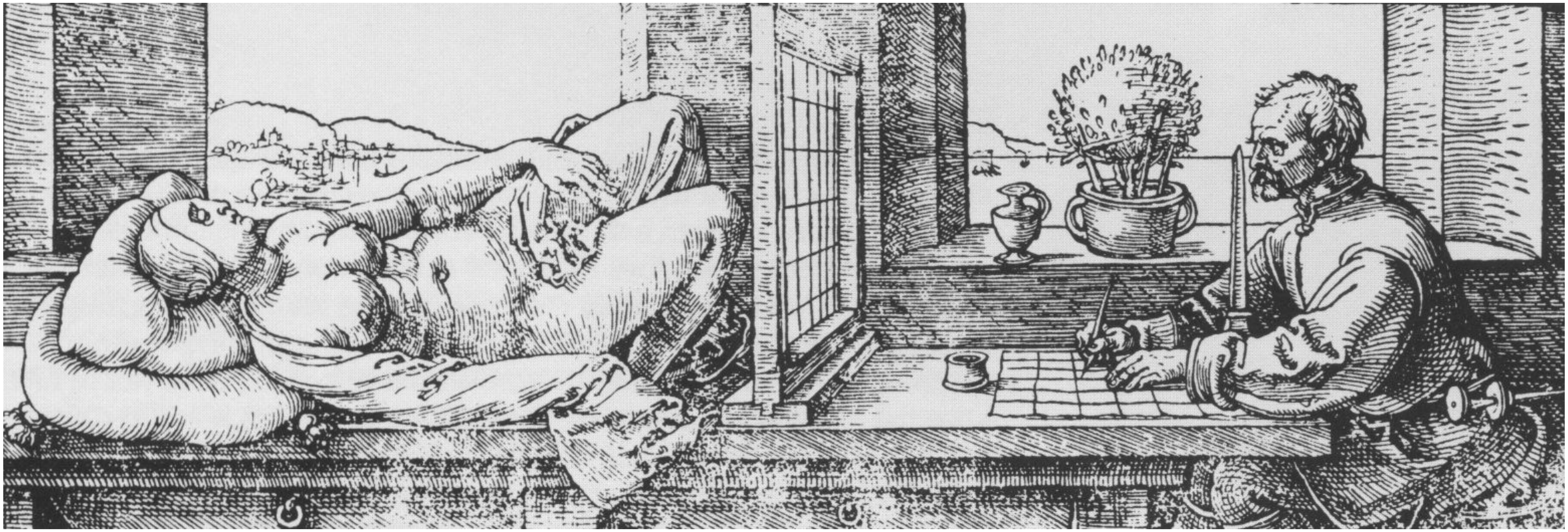


Abelardo Morell

- Photographer who turns hotel room into a camera obscura (pinhole optics)

Durer's Ray casting machine

- Albrecht Durer, 16th century

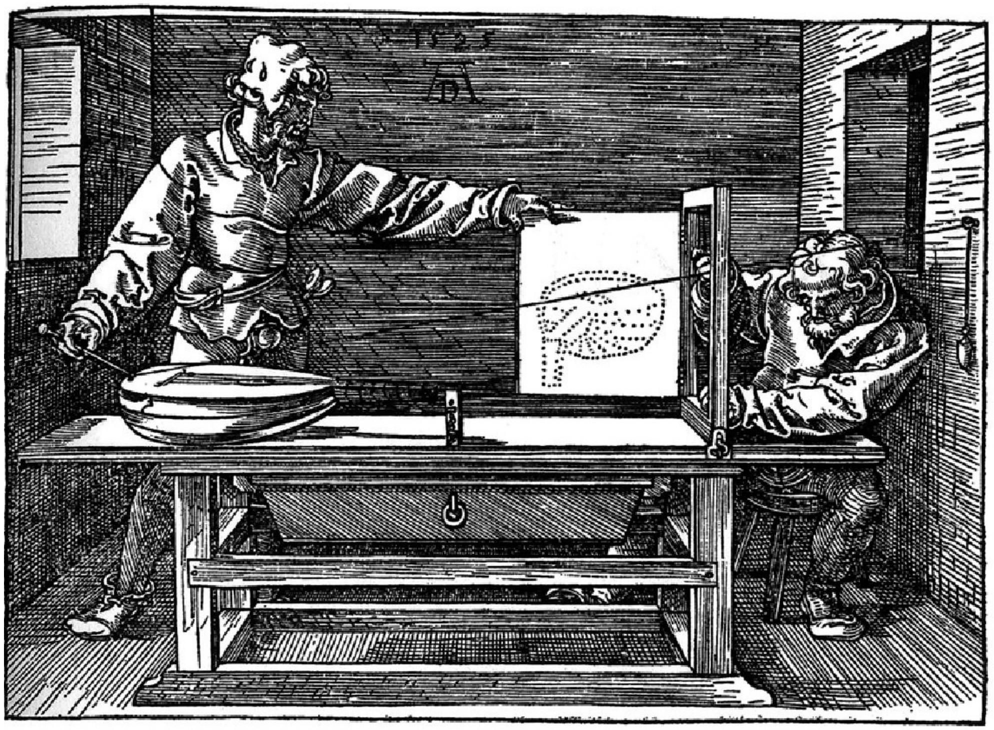


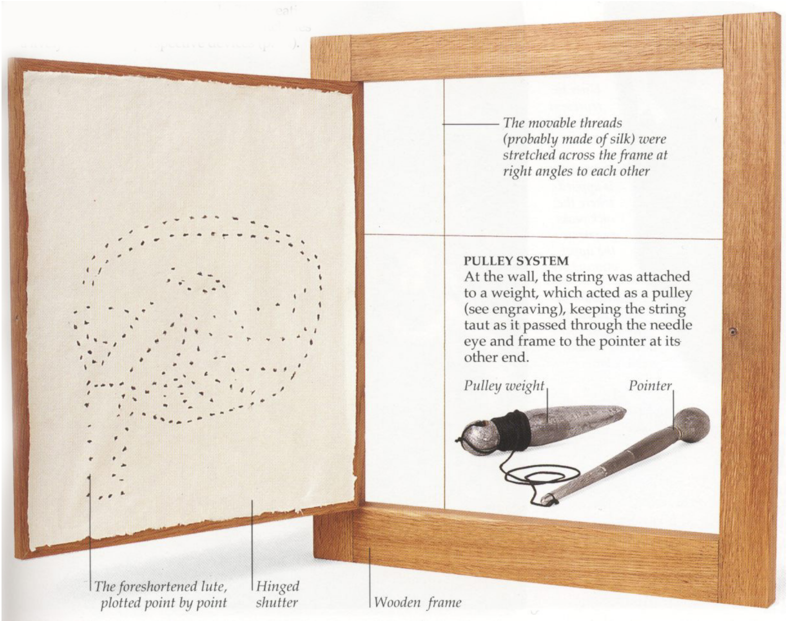
Durer’s Ray casting machine

- Albrecht Durer, 16th century

Durer’s Ray casting machine

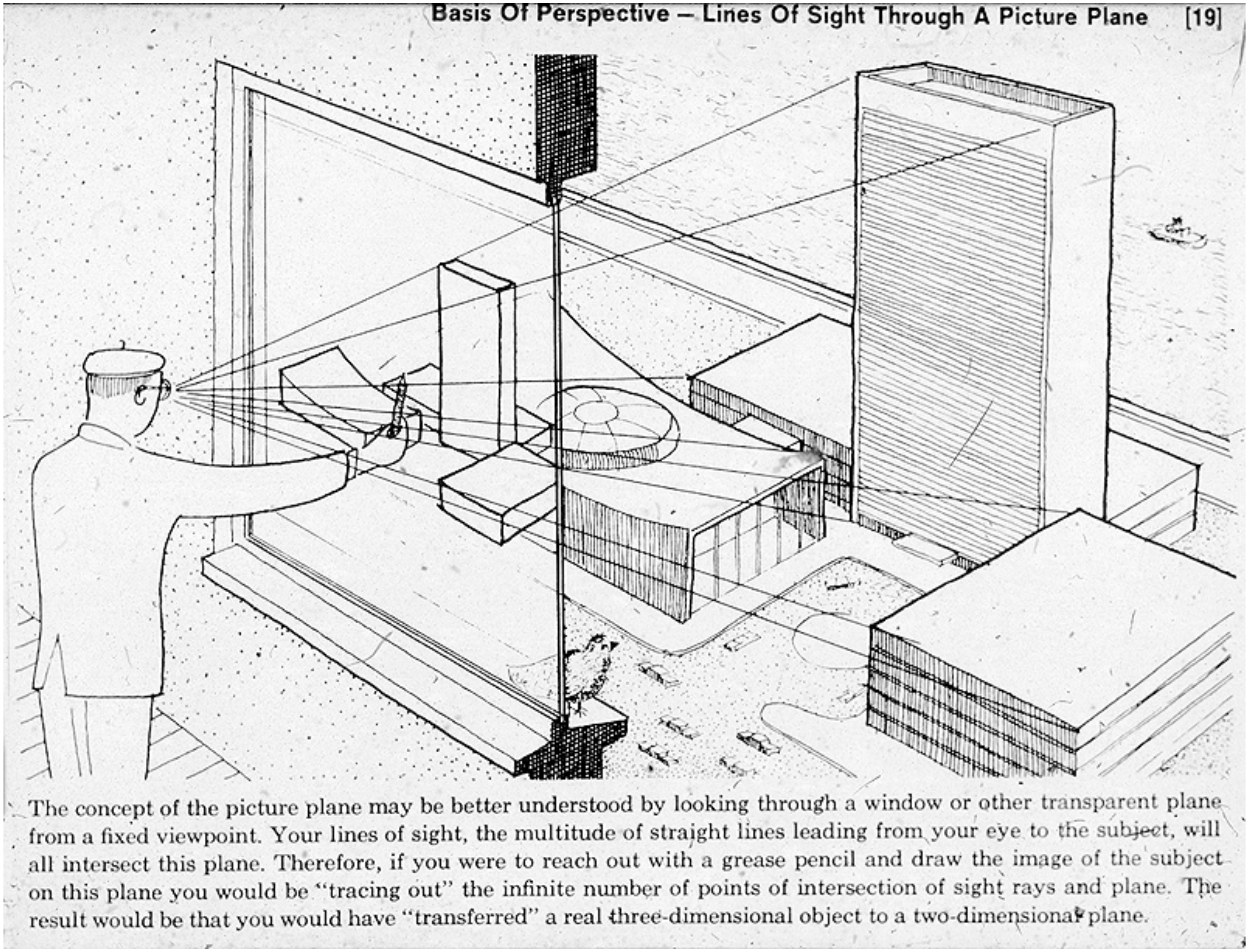
- Albrecht Durer, 16th century




The movable threads (probably made of silk) were stretched across the frame at right angles to each other

PULLEY SYSTEM
At the wall, the string was attached to a weight, which acted as a pulley (see engraving), keeping the string taut as it passed through the needle eye and frame to the pointer at its other end.

Plane projection in drawing



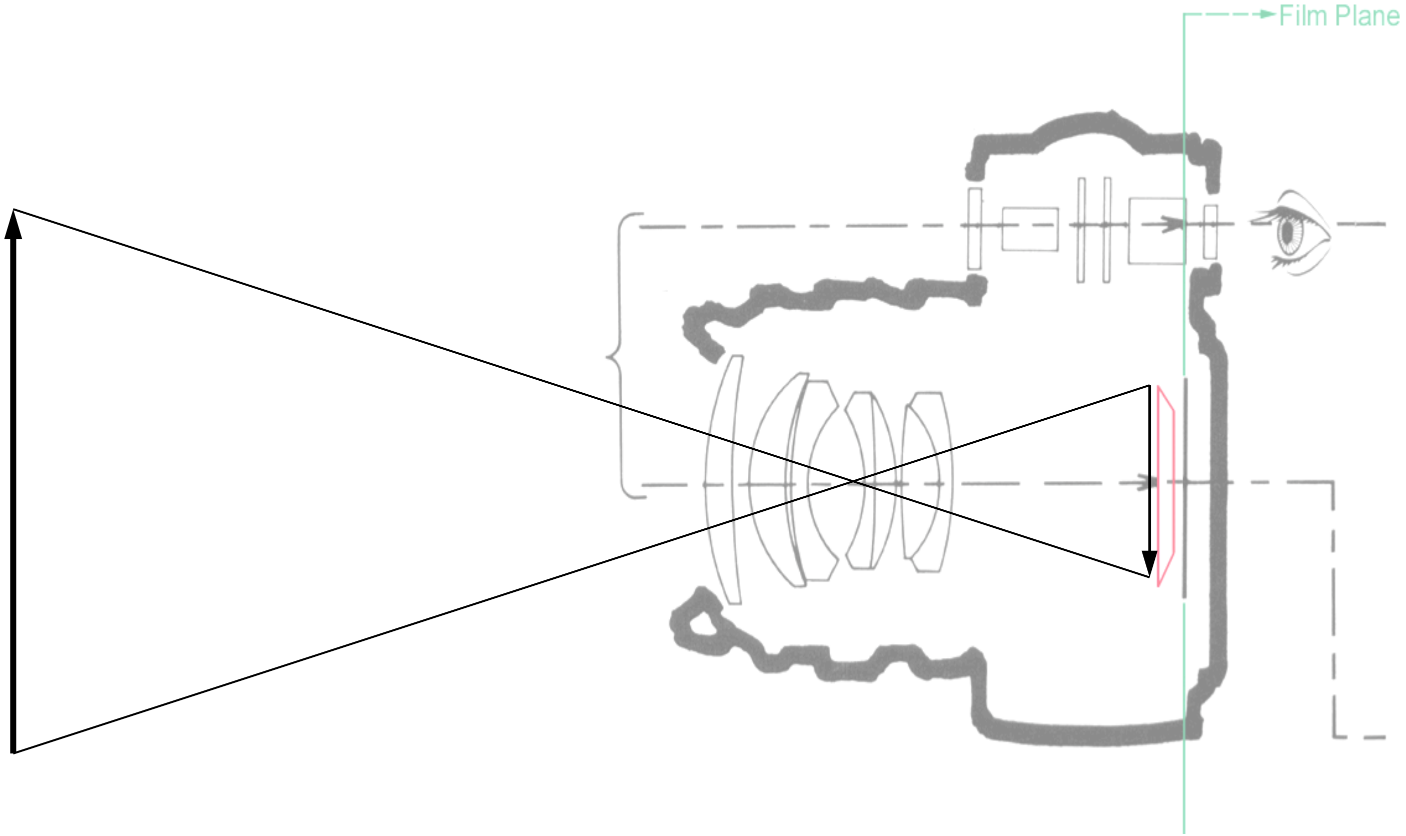
Basis Of Perspective — Lines Of Sight Through A Picture Plane [19]

The concept of the picture plane may be better understood by looking through a window or other transparent plane from a fixed viewpoint. Your lines of sight, the multitude of straight lines leading from your eye to the subject, will all intersect this plane. Therefore, if you were to reach out with a grease pencil and draw the image of the subject on this plane you would be "tracing out" the infinite number of points of intersection of sight rays and plane. The result would be that you would have "transferred" a real three-dimensional object to a two-dimensional plane.

[CS 417 Spring 2002]

Plane projection in photography

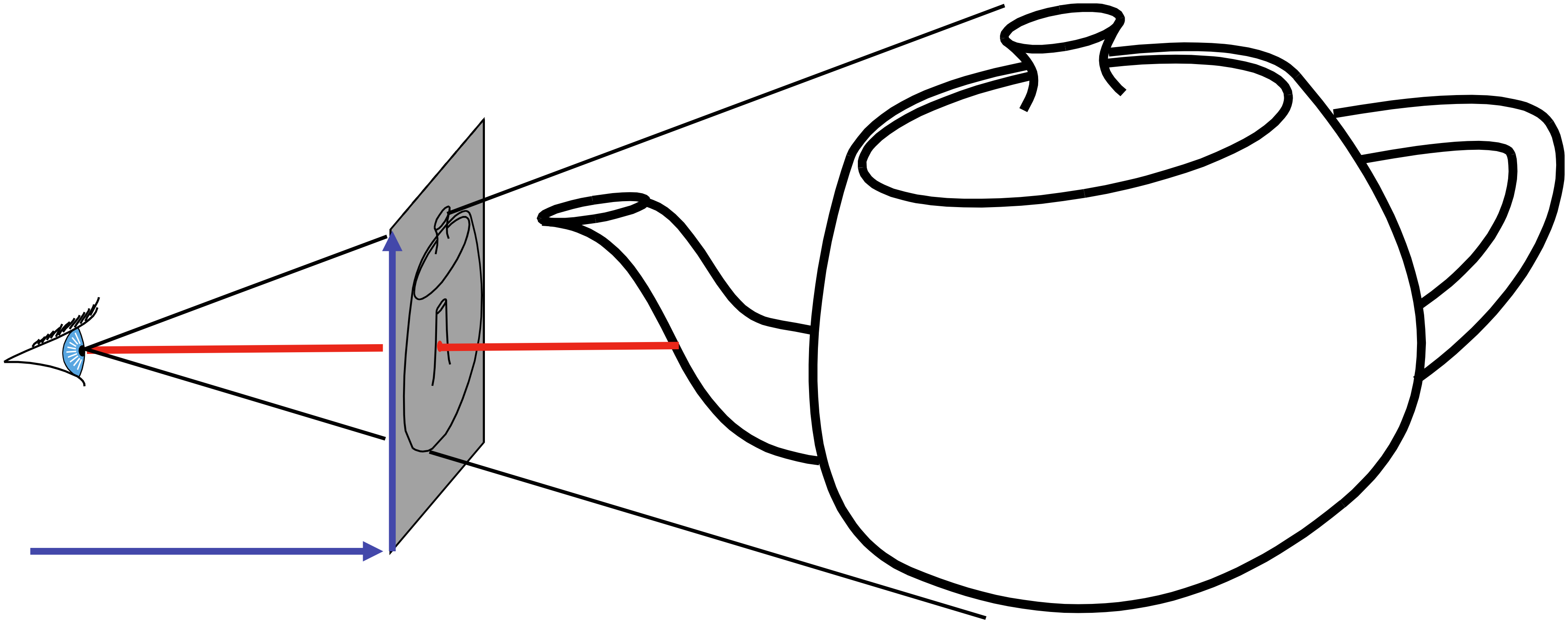
- This is another model for what we are doing
  - applies more directly in realistic rendering



[CS 417 Spring 2002]

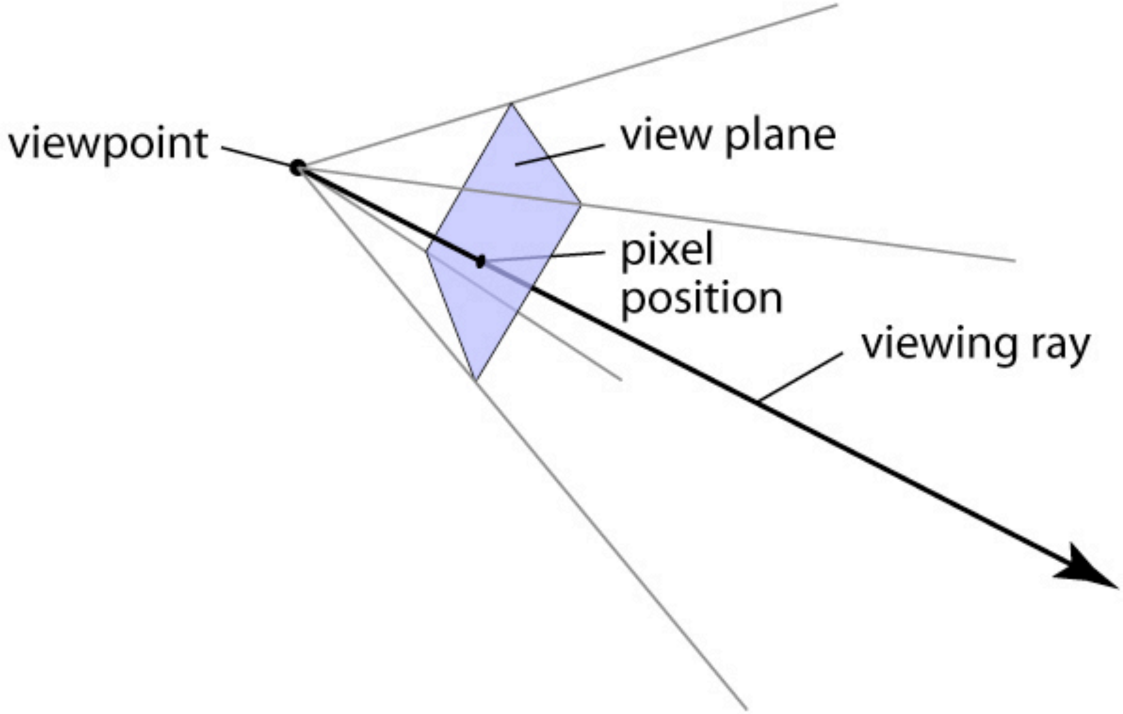
Simplified pinhole camera

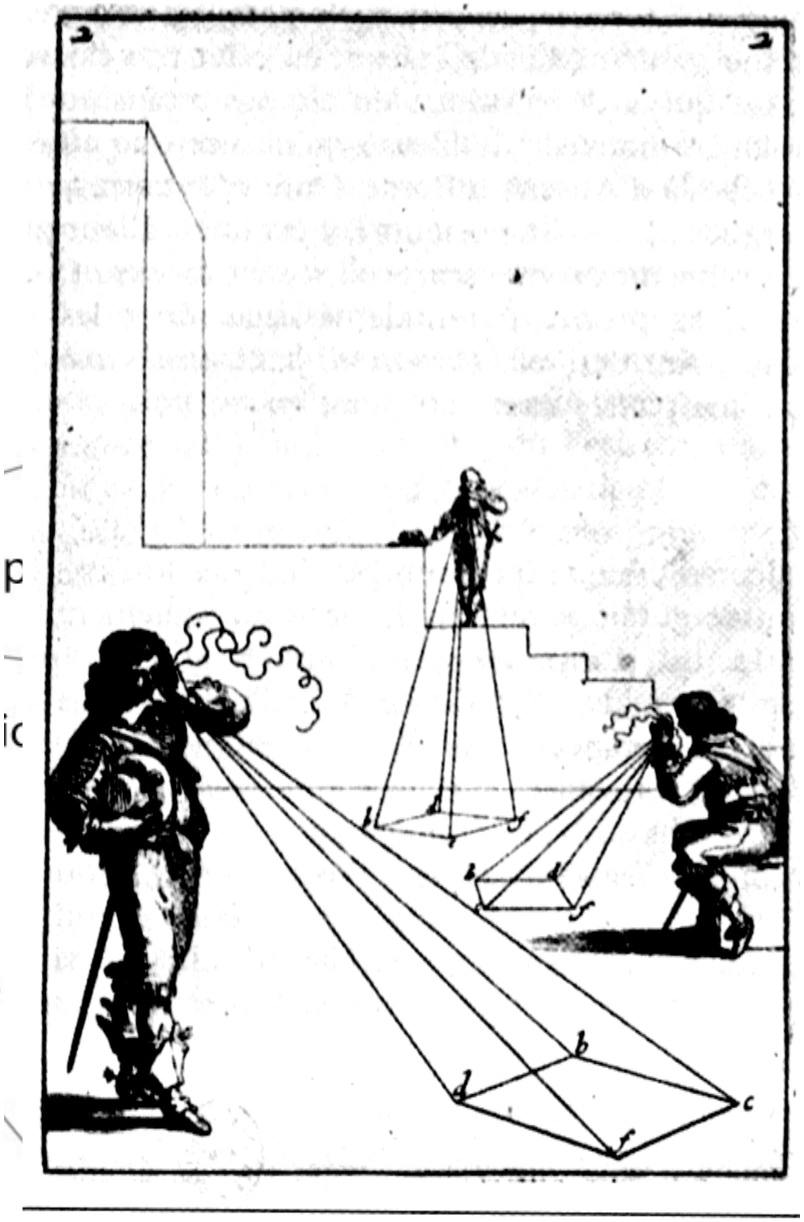
- Eye-image pyramid (frustum)
- Note that the distance/size of image are arbitrary



Generating eye rays

- Use window analogy directly





braham Bosse, *Les Perspecteurs*. Gravure extraite de la M

Vector math review

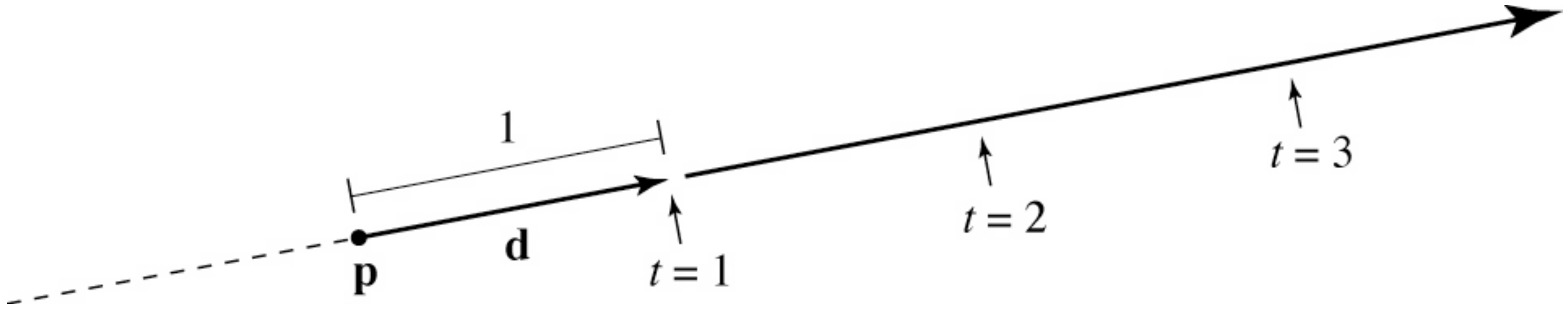
- Vectors and points
- Vector operations
  - addition
  - scalar product
- More products
  - dot product
  - cross product

Ray: a half line

- Standard representation: point **p** and direction **d**

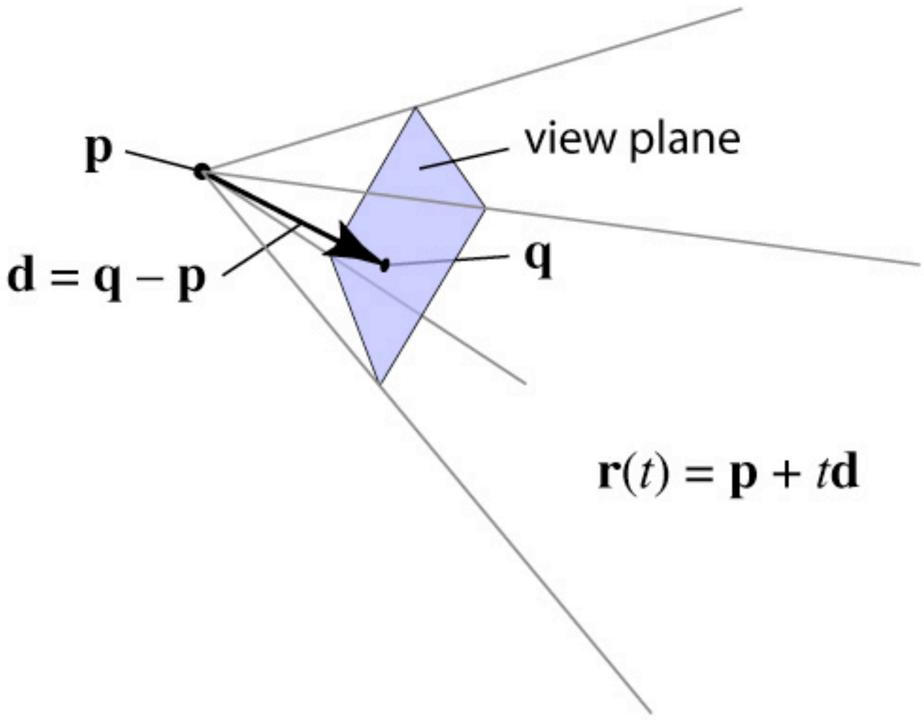
  $\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$

  - this is a *parametric equation* for the line
  - lets us directly generate the points on the line
  - if we restrict to *t* > 0 then we have a ray
  - note replacing **d** with *a***d** doesn't change ray (*a* > 0)



Generating eye rays

- Just need to compute the view plane point **q**:



  - we won't worry about the details for now

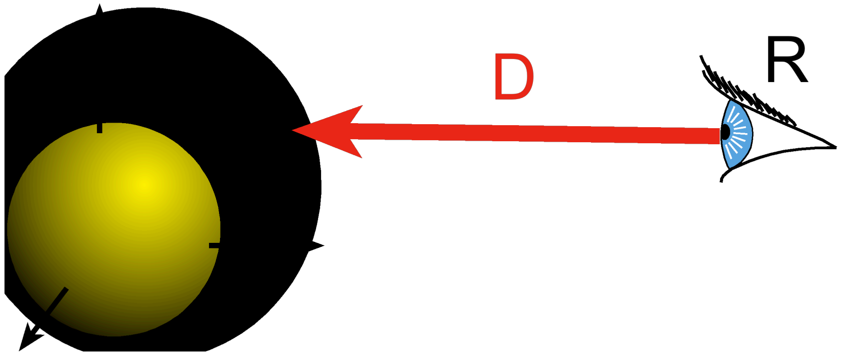
Sphere equation

- Sphere equation (implicit):

$$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Assume unit dimensions, centered at origin



Explicit vs. implicit?

- Sphere equation is implicit
  - Solution of an equation
  - Does not tell us how to generate a point on the sphere
  - Tells us how to check that a point is on the sphere
- Ray equation is explicit
  - Parametric
  - How to generate points
  - Harder to verify that a point is on the ray

Ray-sphere intersection: algebraic

- Condition 1: point is on ray

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

- Condition 2: point is on sphere
  - assume unit sphere

$$\|\mathbf{x}\| = 1 \Leftrightarrow \|\mathbf{x}\|^2 = 1$$

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{x} - 1 = 0$$

- Substitute:

$$(\mathbf{p} + t\mathbf{d}) \cdot (\mathbf{p} + t\mathbf{d}) - 1 = 0$$

  - this is a quadratic equation in *t*

Ray-sphere intersection: algebraic

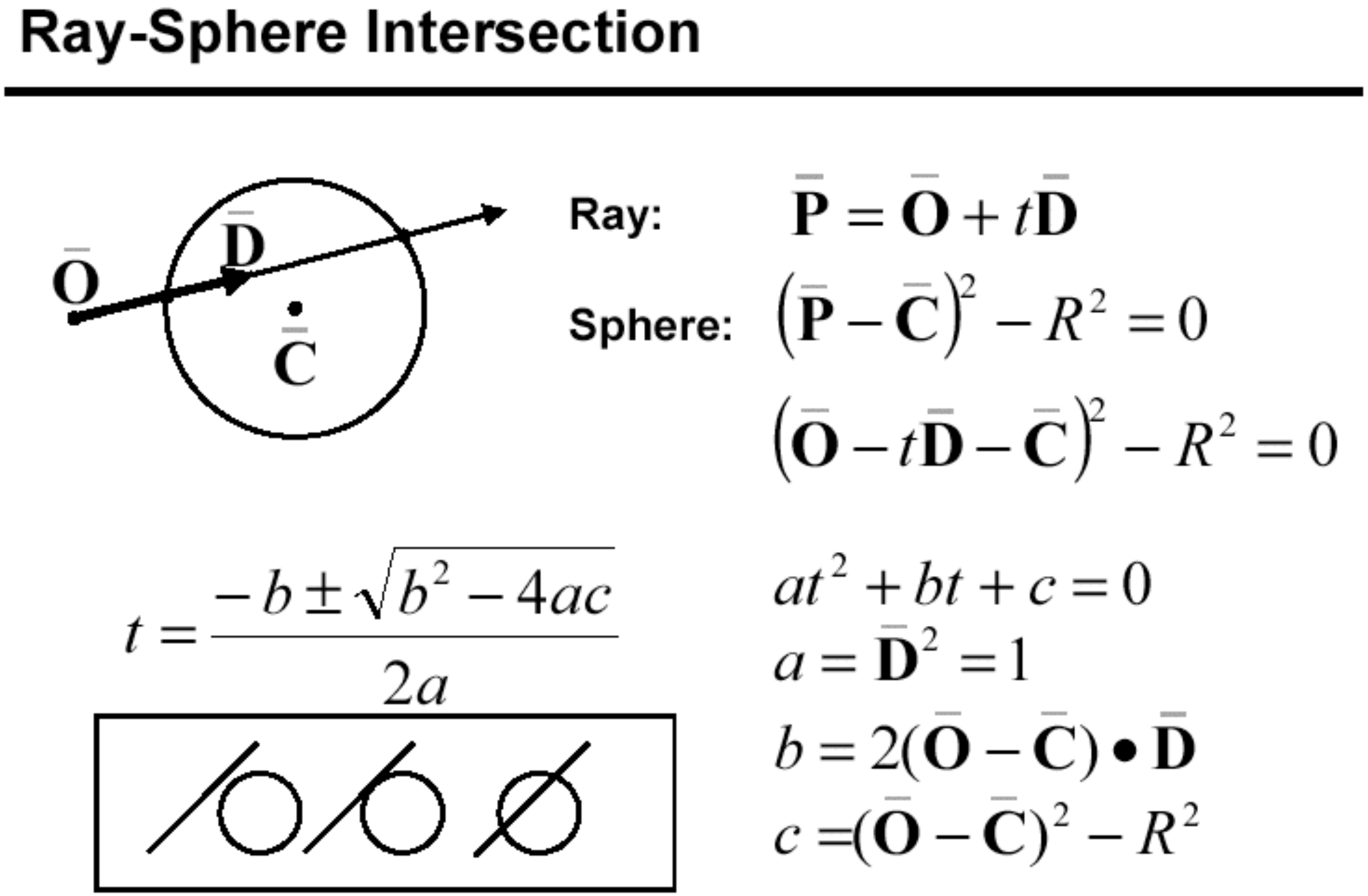
- Solution for *t* by quadratic formula:

$$t = \frac{-\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - (\mathbf{d} \cdot \mathbf{d})(\mathbf{p} \cdot \mathbf{p} - 1)}}{\mathbf{d} \cdot \mathbf{d}}$$

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

  - simpler form holds when **d** is a unit vector
    but we won’t assume this in practice (reason later)
  - I’ll use the unit-vector form to make the geometric interpretation

Ray-sphere intersection: algebraic



Ray-Sphere Intersection

Ray: $\bar{\mathbf{P}} = \bar{\mathbf{O}} + t\bar{\mathbf{D}}$

Sphere: $\left(\bar{\mathbf{P}} - \bar{\mathbf{C}}\right)^2 - R^2 = 0$

$$\left(\bar{\mathbf{O}} - t\bar{\mathbf{D}} - \bar{\mathbf{C}}\right)^2 - R^2 = 0$$

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

$$at^2 + bt + c = 0$$

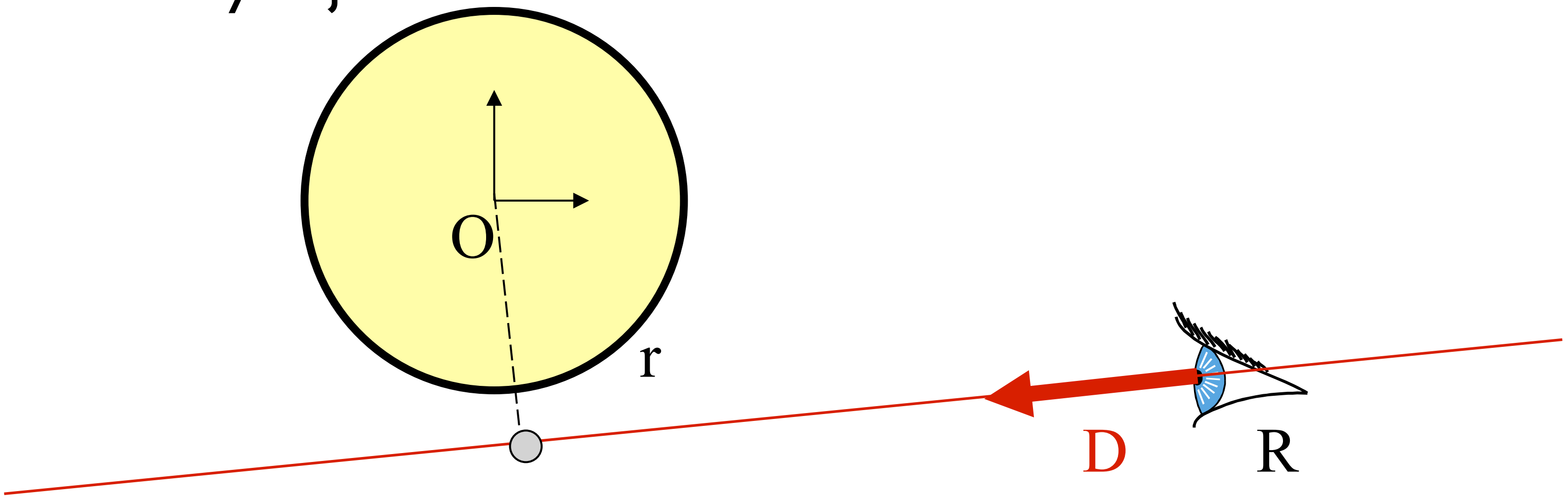
$$a = \bar{\mathbf{D}}^2 = 1$$

$$b = 2(\bar{\mathbf{O}} - \bar{\mathbf{C}}) \bullet \bar{\mathbf{D}}$$

$$c = (\bar{\mathbf{O}} - \bar{\mathbf{C}})^2 - R^2$$

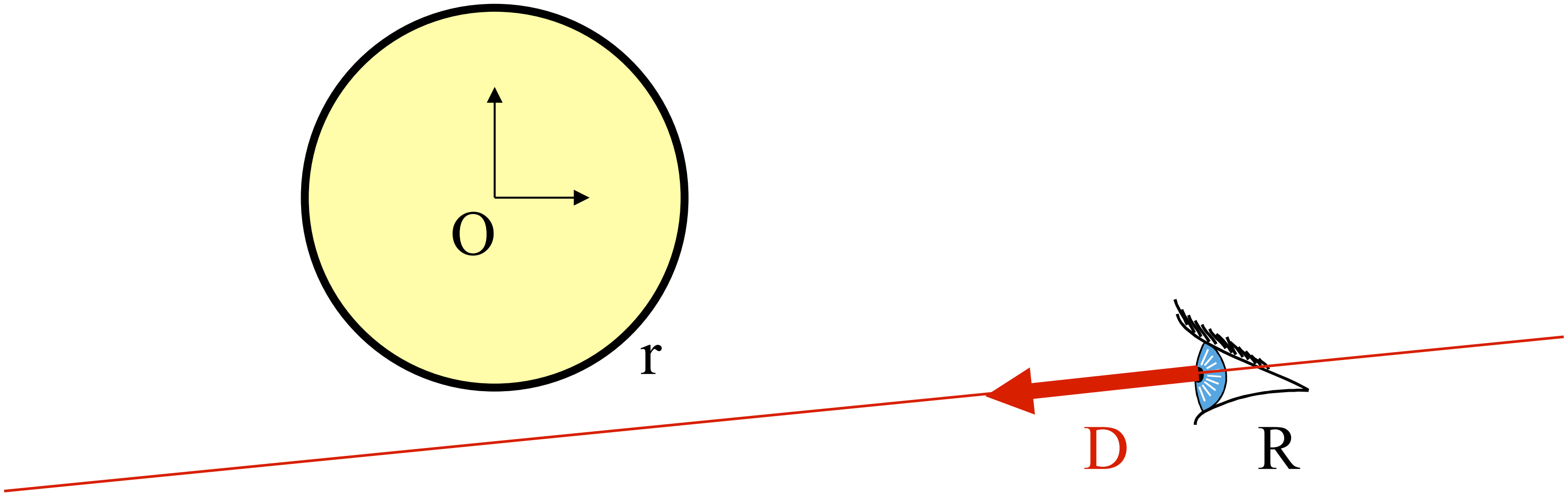
Ray-sphere intersection: geometric

- What geometric information is important?
  - Inside/outside
  - Closest point
  - Direction
- Geometric considerations can help shortcut calculations
  - easy reject



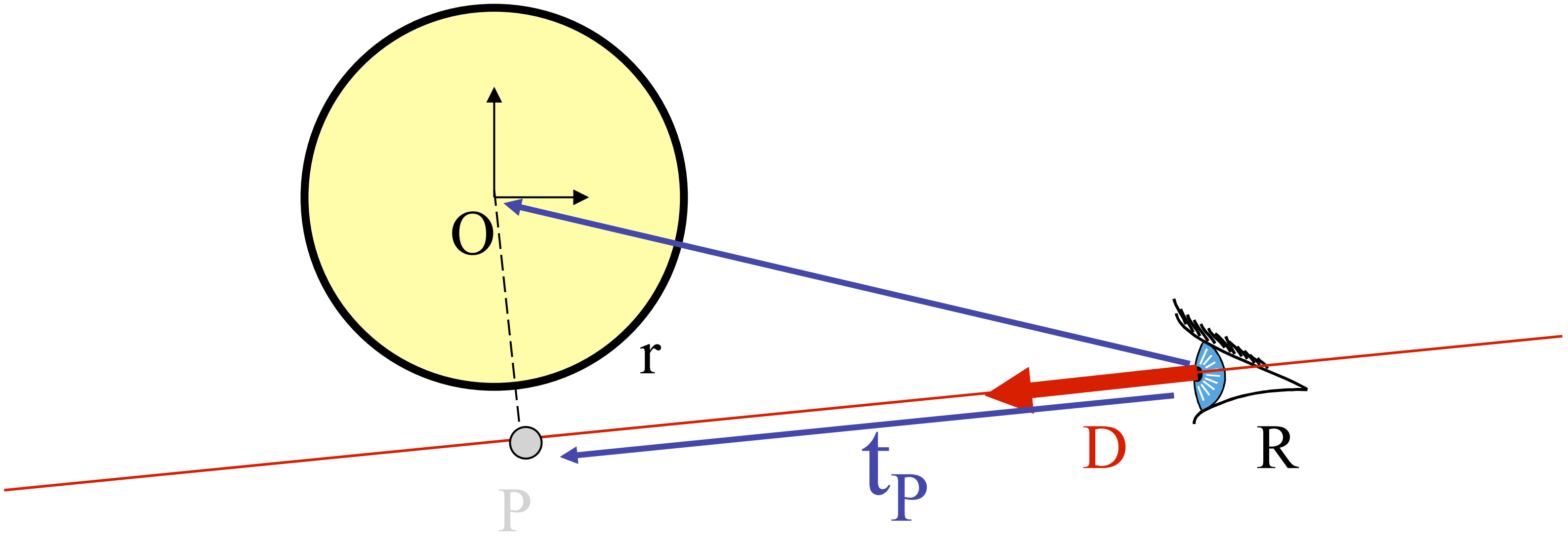
Ray-sphere intersection: geometric

- Find if the ray's origin is outside the sphere
  - $R^2>r^2$
  - If inside, it intersects
  - If on the sphere, it does not intersect (avoid degeneracy)



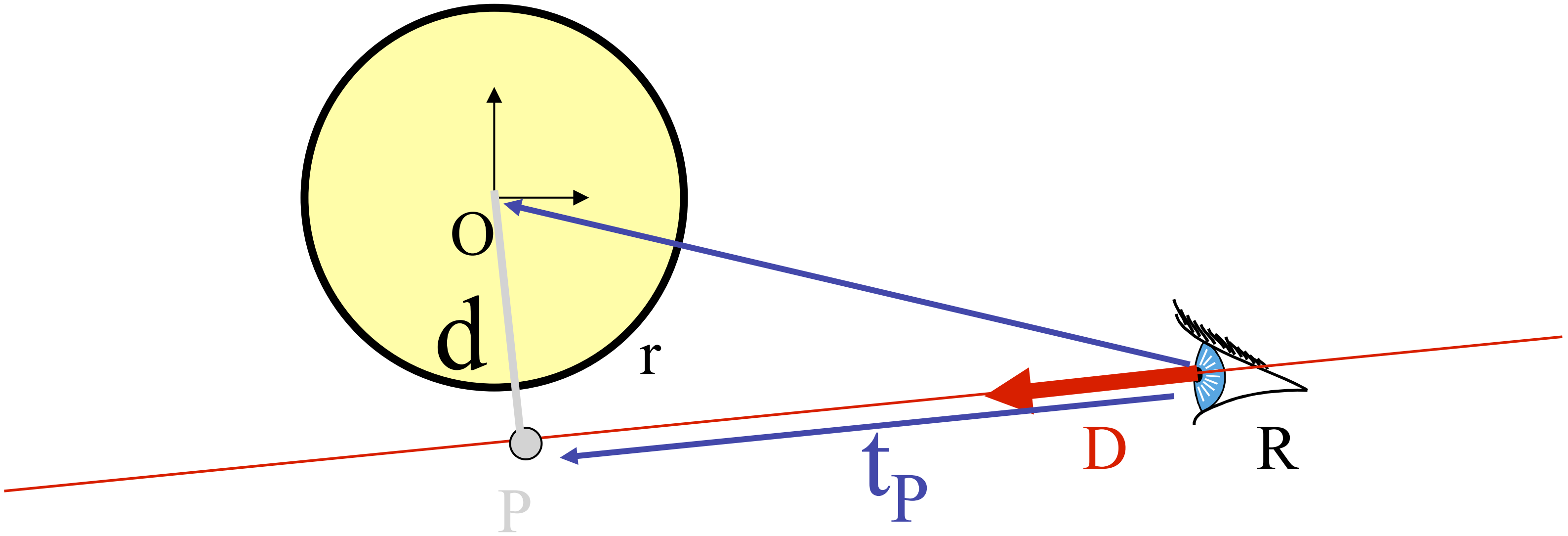
Ray-sphere intersection: geometric

- Find if the ray's origin is outside the sphere
- Find the closest point to the sphere center
  - t_P=RO.D
  - If t_P<0, no hit



Ray-sphere intersection: geometric

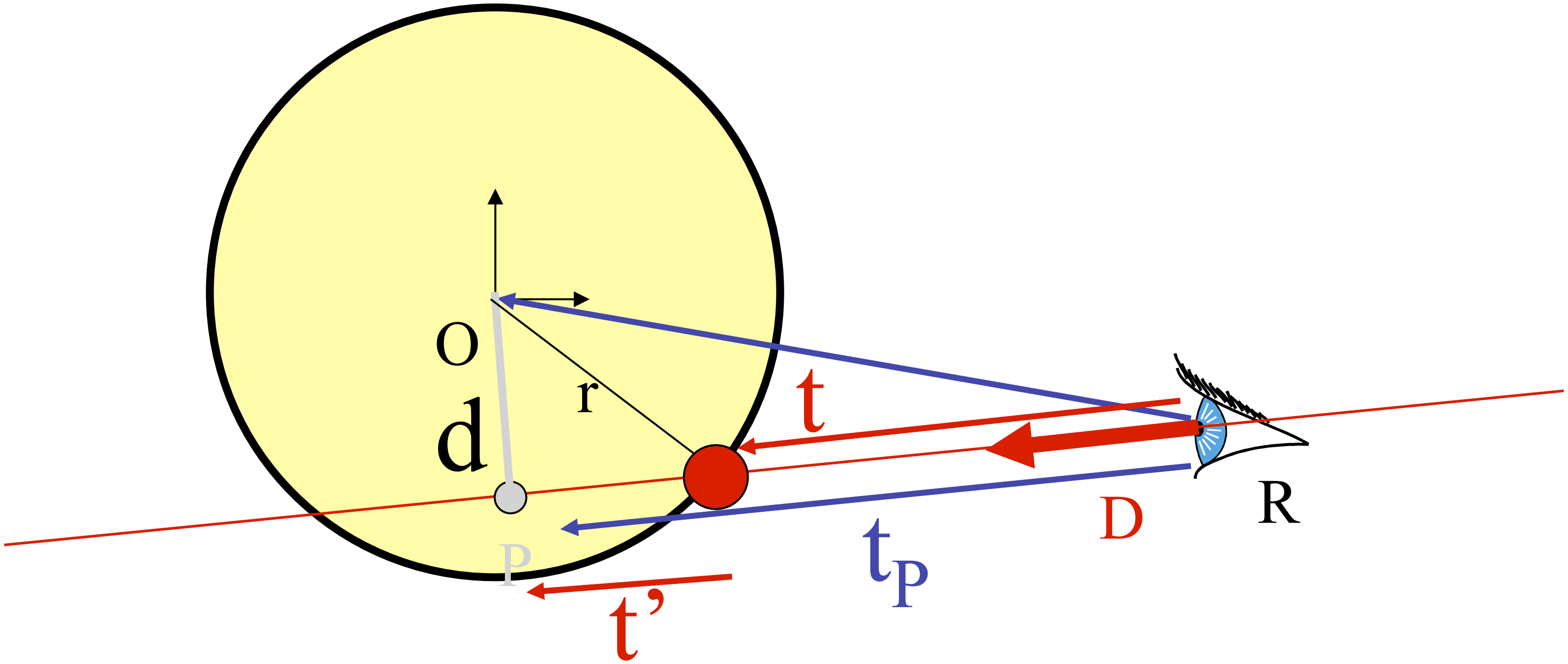
- Find if the ray's origin is outside the sphere
- Find the closest point to the sphere center
  - If $t_P<0$, no hit
- Else find squared distance d^2
  - Pythagoras: $d^2=R^2-t_P^2$
  - … if $d^2> r^2$ no hit



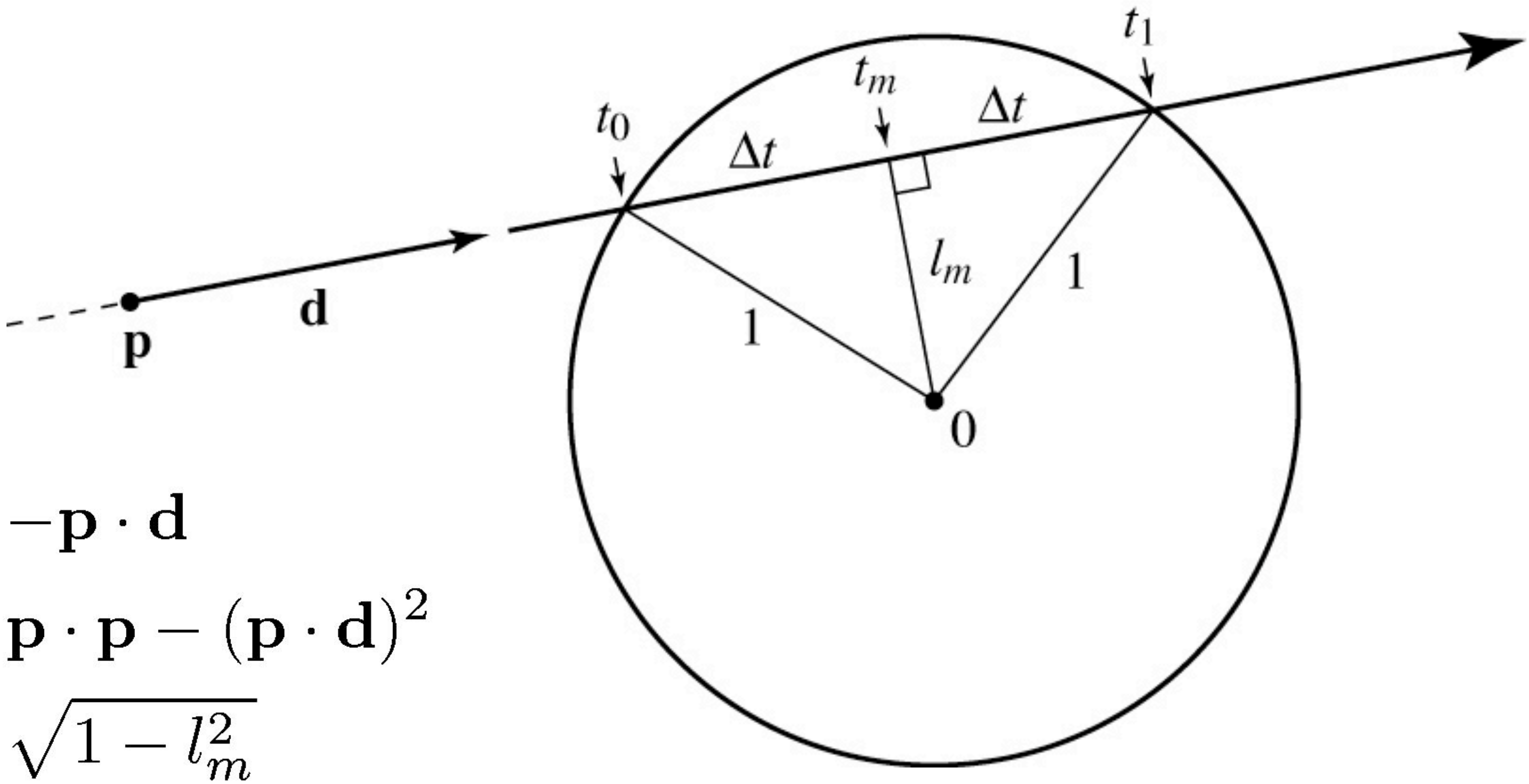
Ray-sphere intersection: geometric

- Find if the ray's origin is outside the sphere
- Find the closest point to the sphere center
  - If $t_P<0$, no hit
- Else find squared distance d^2
  - if $d^2 > r^2$ no hit

- If outside $t = t_P - t'$
  - $t'^2+d^2=r^2$
- If inside $t = t_P + t'$



Ray-sphere intersection: geometric



$$t_m = -\mathbf{p} \cdot \mathbf{d}$$

$$l_m^2 = \mathbf{p} \cdot \mathbf{p} - (\mathbf{p} \cdot \mathbf{d})^2$$

$$\Delta t = \sqrt{1 - l_m^2}$$

$$= \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

$$t_{0,1} = t_m \pm \Delta t = -\mathbf{p} \cdot \mathbf{d} \pm \sqrt{(\mathbf{p} \cdot \mathbf{d})^2 - \mathbf{p} \cdot \mathbf{p} + 1}$$

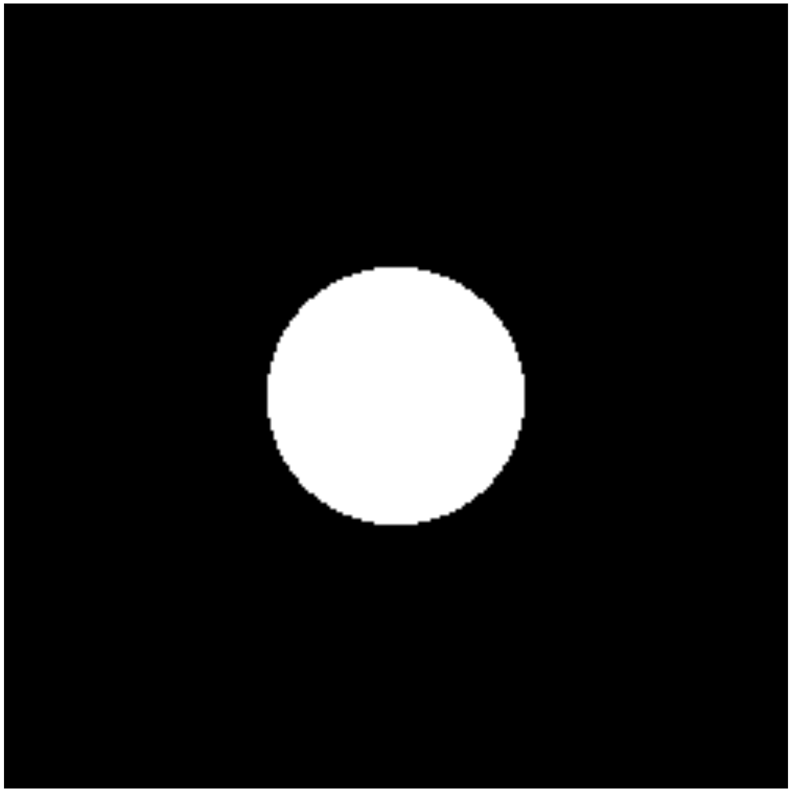
Geometric vs. algebraic

- Algebraic was more simple
  (and more generic)
- Geometric is more efficient
  - Timely tests
  - In particular for outside and pointing away

Image so far

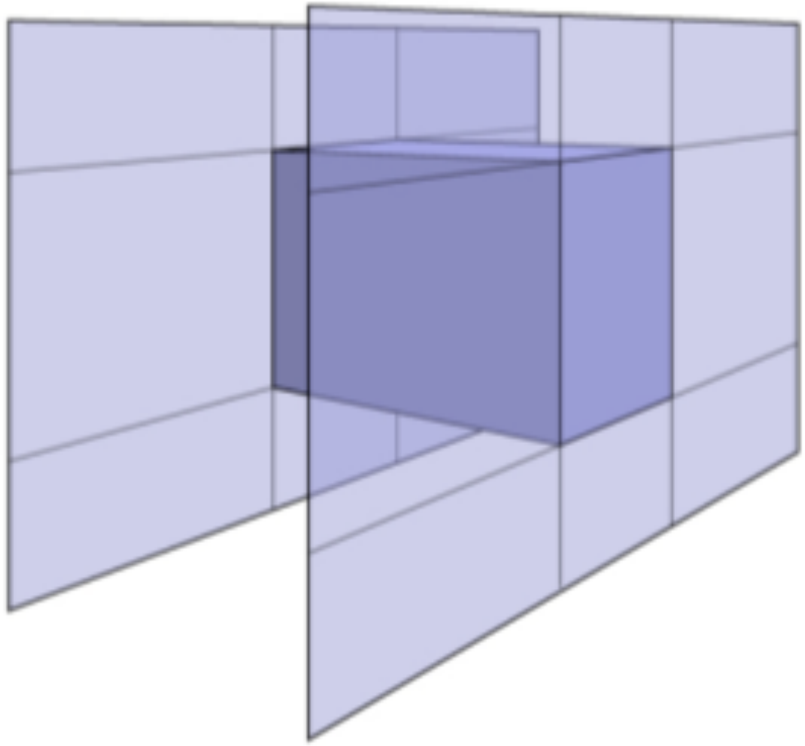
- With eye ray generation and sphere intersection

```
Surface s = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        if (s.intersect(ray, 0, +inf) < +inf)
            image.set(ix, iy, white);
    }
```



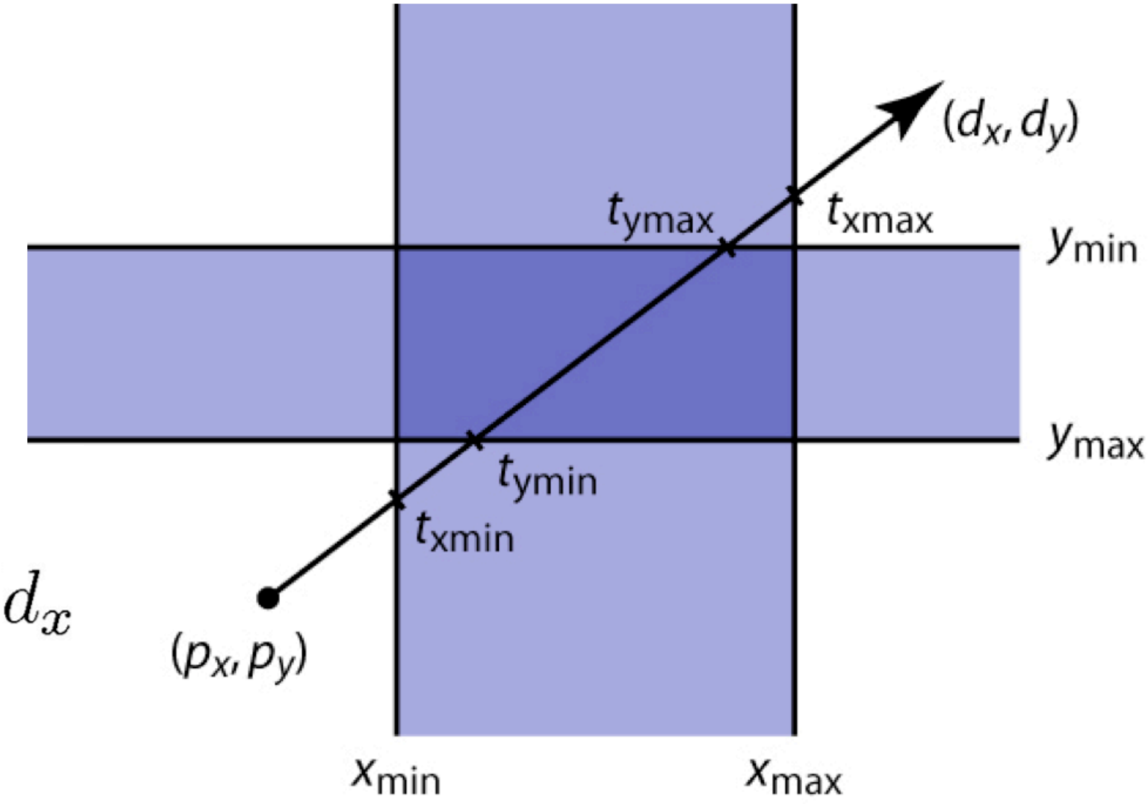
Ray-box intersection

- Could intersect with 6 faces individually
- If axis-aligned, box is
  the intersection of 3 slabs



Ray-slab intersection

- 2D example
- 3D is the same

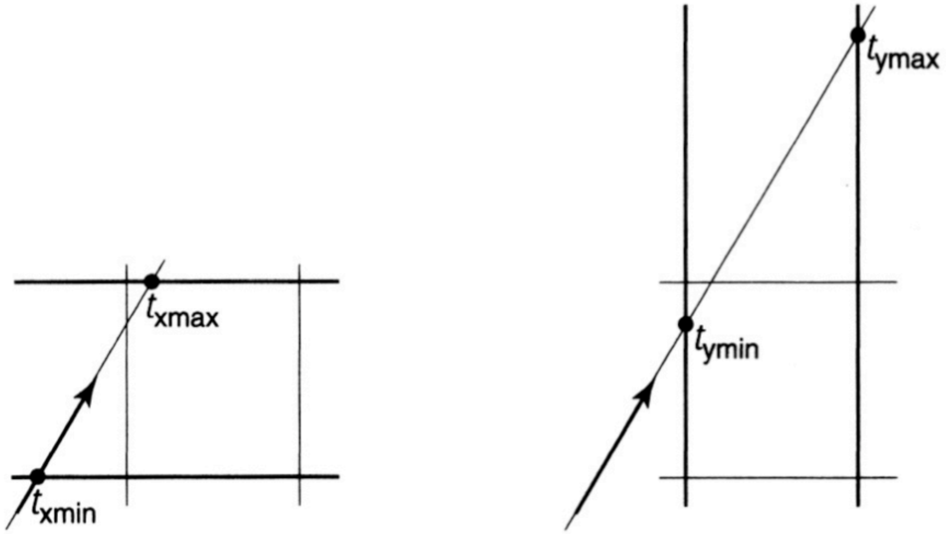


$$p_x + t_{x\min}\, d_x = x_{\min}$$
$$t_{x\min} = (x_{\min} - p_x)/d_x$$

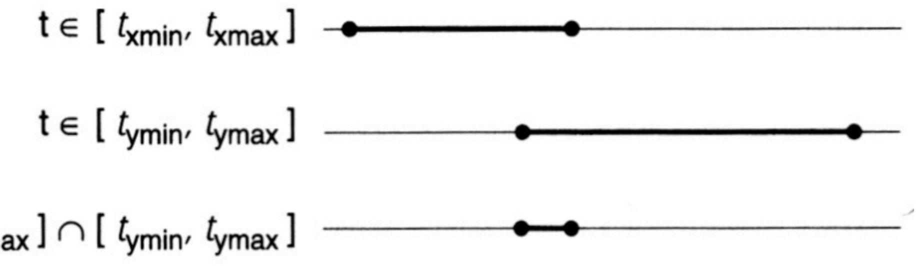
$$p_y + t_{y\min}\, d_y = y_{\min}$$
$$t_{y\min} = (y_{\min} - p_y)/d_y$$

Intersection ranges

- Each intersection is an interval
- Want last entry point and first exit point



$$t_{\min} = \max(t_{x\min}, t_{y\min})$$
$$t_{\max} = \min(t_{x\max}, t_{y\max})$$



Shirley fig. 10.16

General Ray-plane intersection

- Condition 1: point is on ray

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

- Condition 2: point is on plane

$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0$$

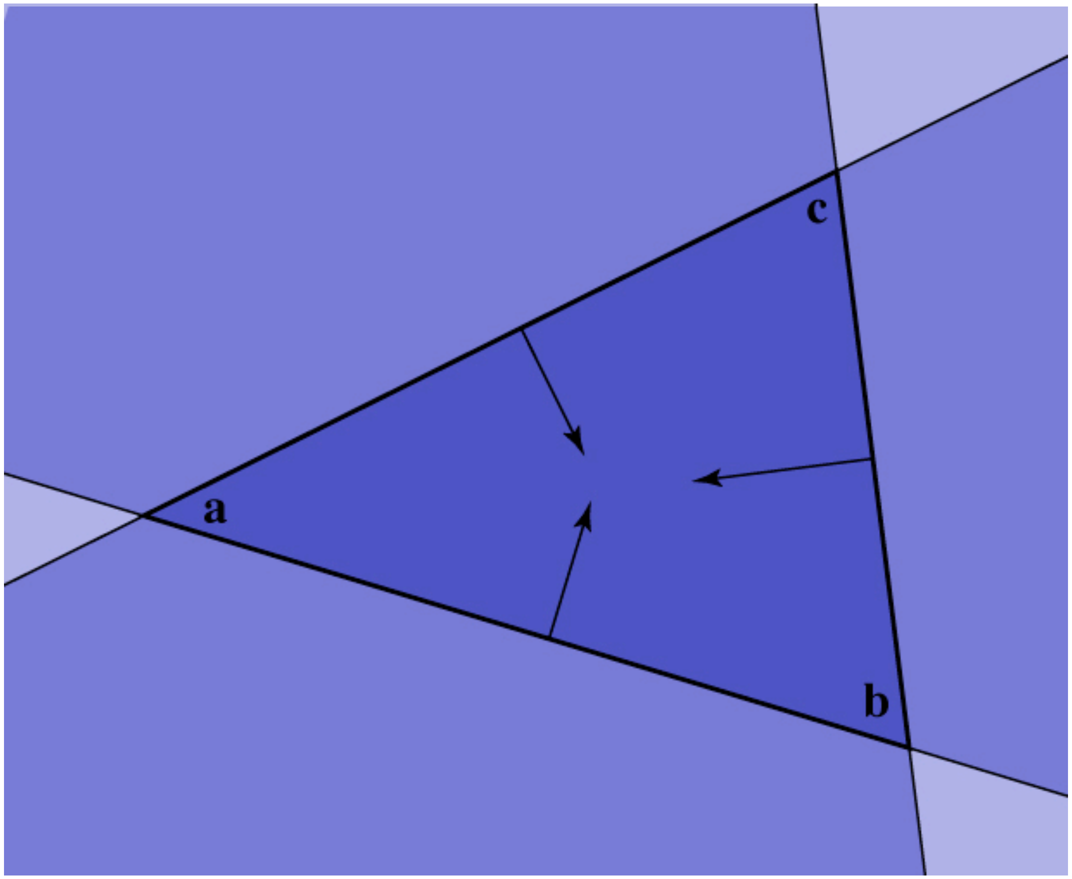
- Condition 3: point is on the inside of all edges
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

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

$$t = \frac{(\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

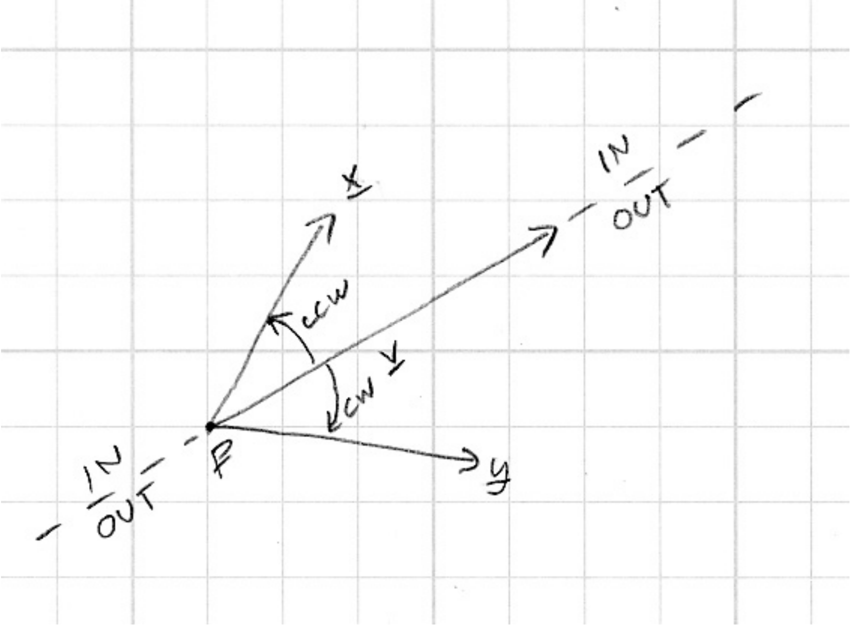
Ray-triangle intersection

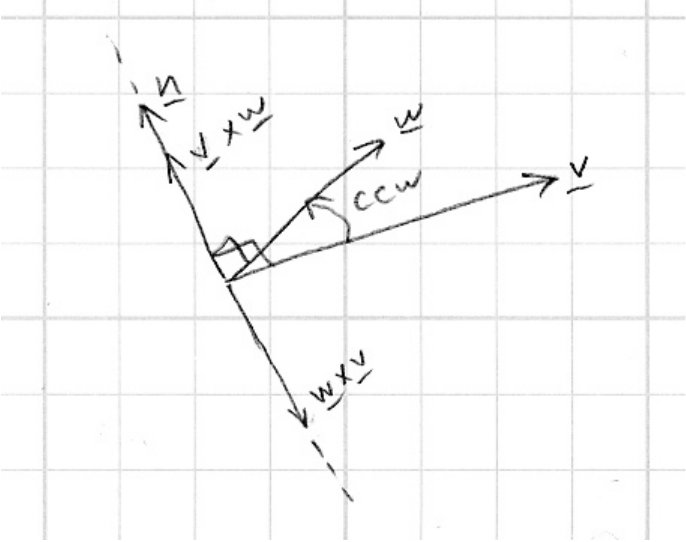
- In plane, triangle is the intersection of 3 half spaces



Inside-edge test

- Need outside vs. inside
- Reduce to clockwise vs. counterclockwise
  - vector of edge to vector to **x**
- Use cross product to decide



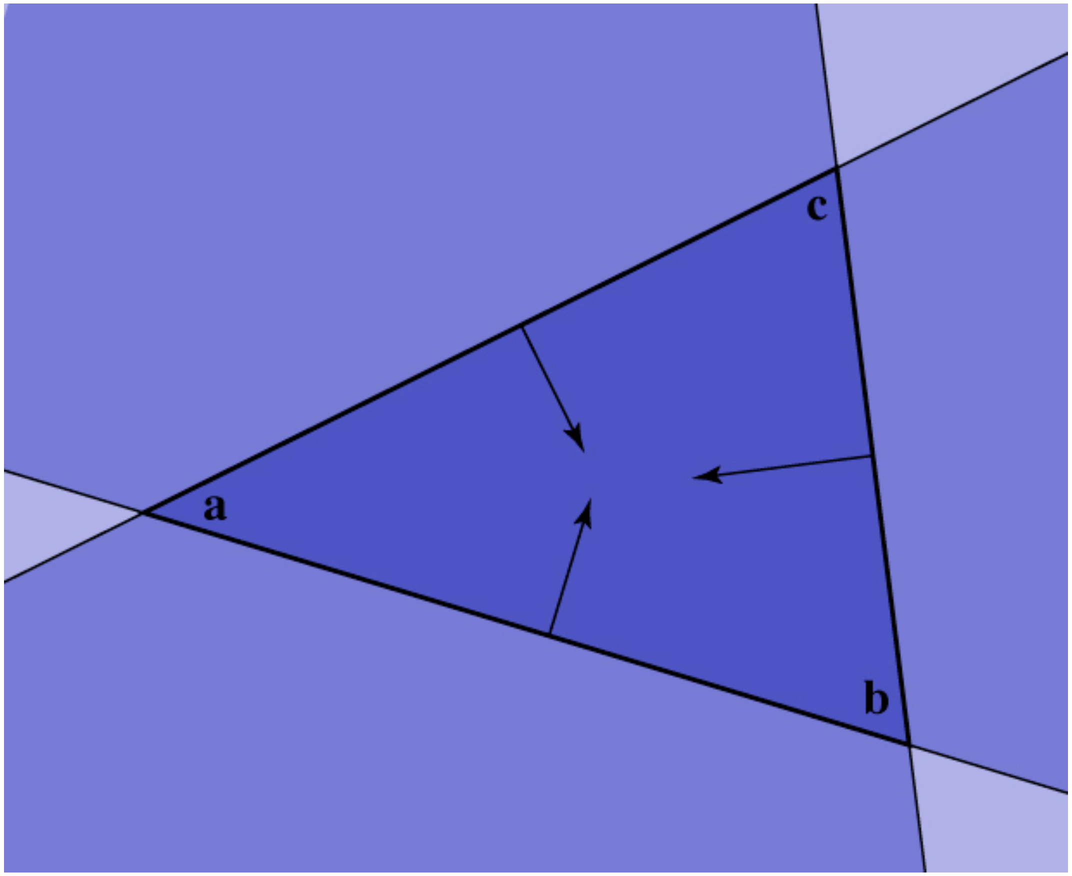


Ray-triangle intersection

$$(\mathbf{b}-\mathbf{a})\times(\mathbf{x}-\mathbf{a})\cdot\mathbf{n}>0$$

$$(\mathbf{c}-\mathbf{b})\times(\mathbf{x}-\mathbf{b})\cdot\mathbf{n}>0$$

$$(\mathbf{a}-\mathbf{c})\times(\mathbf{x}-\mathbf{c})\cdot\mathbf{n}>0$$



Intersection against many shapes

- The basic idea is:

```
hit (ray, tMin, tMax) {
    tBest = +inf; hitSurface = null;
    for surface in surfaceList {
        t = surface.intersect(ray, tMin, tMax);
        if t < tBest {
            tBest = t;
            hitSurface = surface;
        }
    }
    return hitSurface, t;
}
```

  – this is linear in the number of shapes
    but there are sublinear methods (acceleration structures)

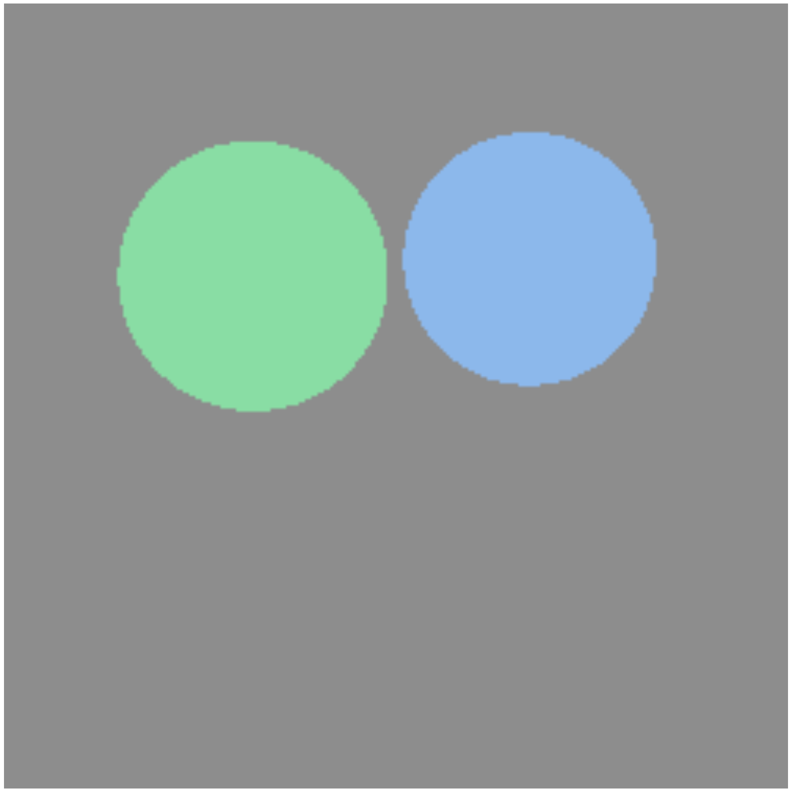
Image so far

- With eye ray generation and scene intersection

```
Geometry g = new Sphere((0.0, 0.0, 0.0), 1.0);
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        c = scene.trace(ray, 0, +inf);
        image.set(ix, iy, c);
    }

...

trace(ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if (surface != null) return surface.color();
    else return black;
}
```



Shading

- Compute light reflected toward camera
- Inputs:
  - eye direction
  - light direction
    (for each of many lights)
  - surface normal
  - surface parameters
    (color, shininess, …)
- More on this in the next lecture

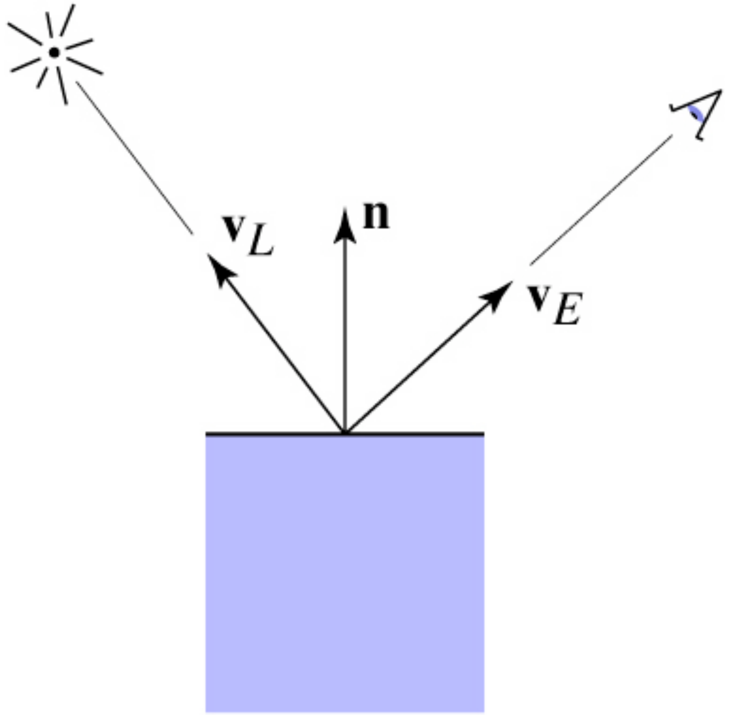
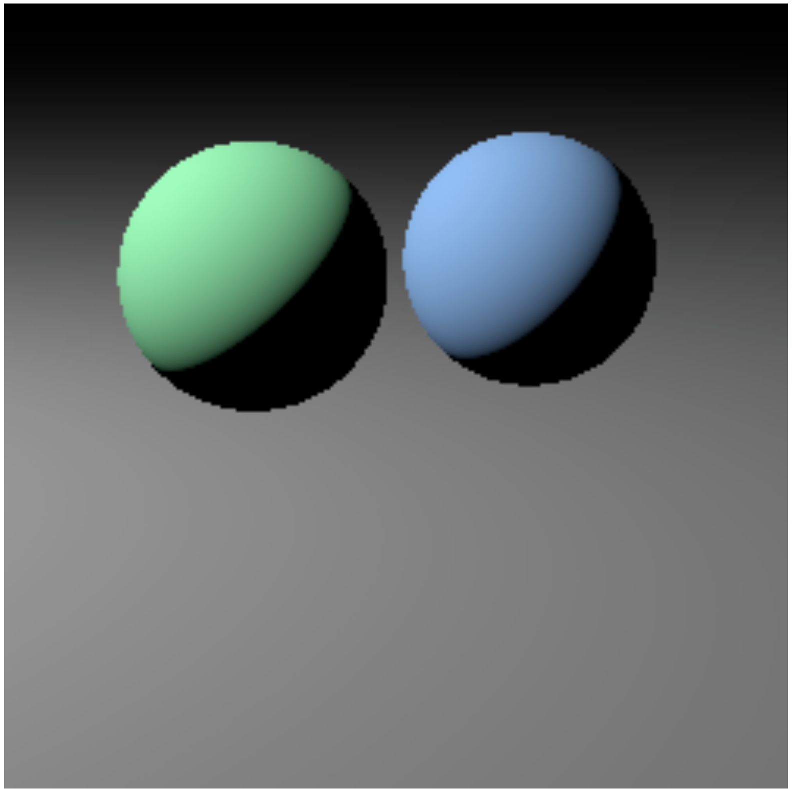


Image so far

```
trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if (surface != null) {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return black;
}

...

shade(ray, point, normal, light) {
    v_E = –normalize(ray.direction);
    v_L = normalize(light.pos - point);
    // compute shading
}
```

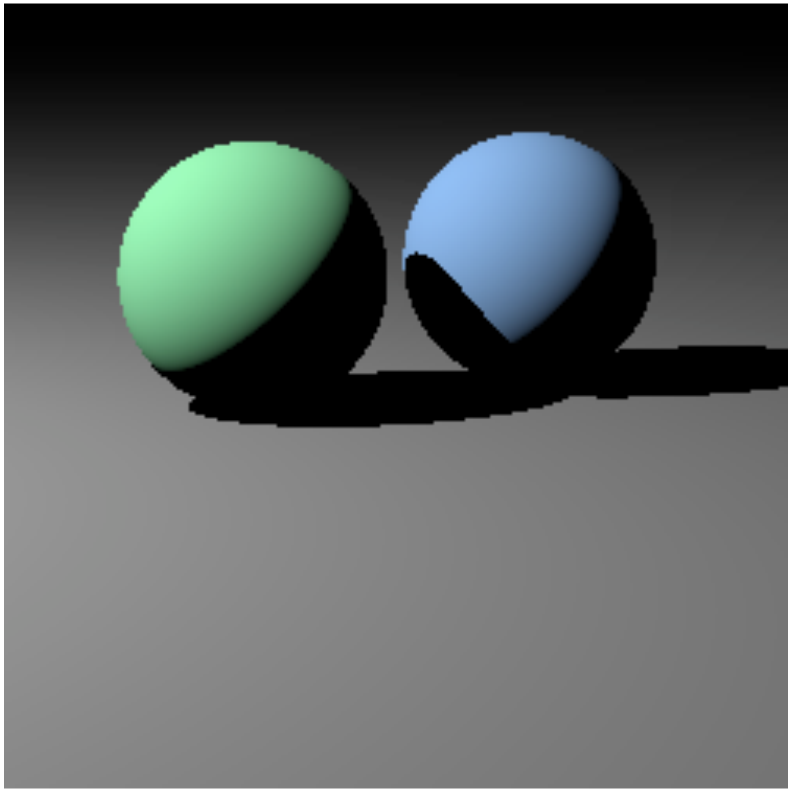


Shadows

- Surface is only illuminated if nothing blocks its view of the light.
- With ray tracing it's easy to check
  - just intersect a ray with the scene!

Image so far

```
shade(ray, point, normal, light) {
    shadRay = (point, light.pos - point);
    if (shadRay not blocked) {
        v_E = −normalize(ray.direction);
        v_L = normalize(light.pos - point);
        // compute shading
    }
    return black;
}
```



Multiple lights

- Important to fill in black shadows
- Just loop over lights, add contributions
- Ambient shading
  - black shadows are not really right
  - one solution: dim light at camera
  - alternative: all surface receive a bit more light
    - just add a constant “ambient” color to the shading…

Image so far

```
shade(ray, point, normal, lights) {
    result = ambient;
    for light in lights {
        if (shadow ray not blocked) {
            result += shading contribution;
        }
    }
    return result;
}
```