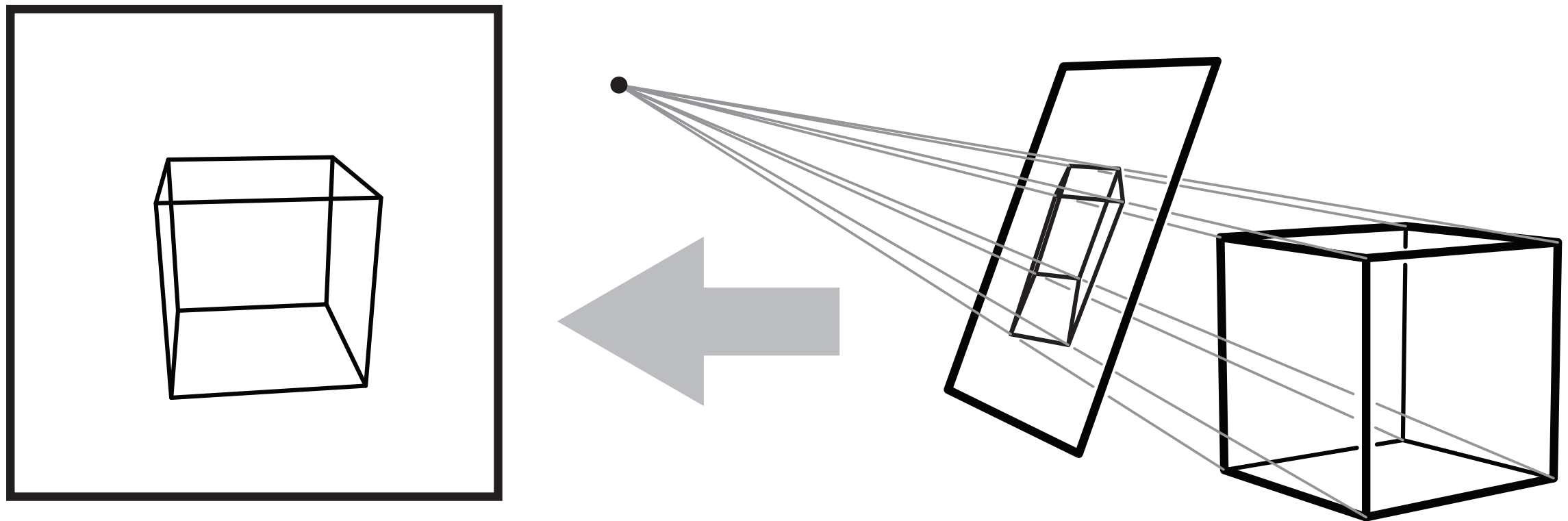


# Viewing and Ray Tracing

## **CS 4620 Lecture 4**

# Projection

- To render an image of a 3D scene, we *project* it onto a plane
- Most common projection type is *perspective projection*



# Two approaches to rendering

```
for each object in the scene {  
  for each pixel in the image {  
    if (object affects pixel) {  
      do something  
    }  
  }  
}
```

**object order**  
or  
**rasterization**

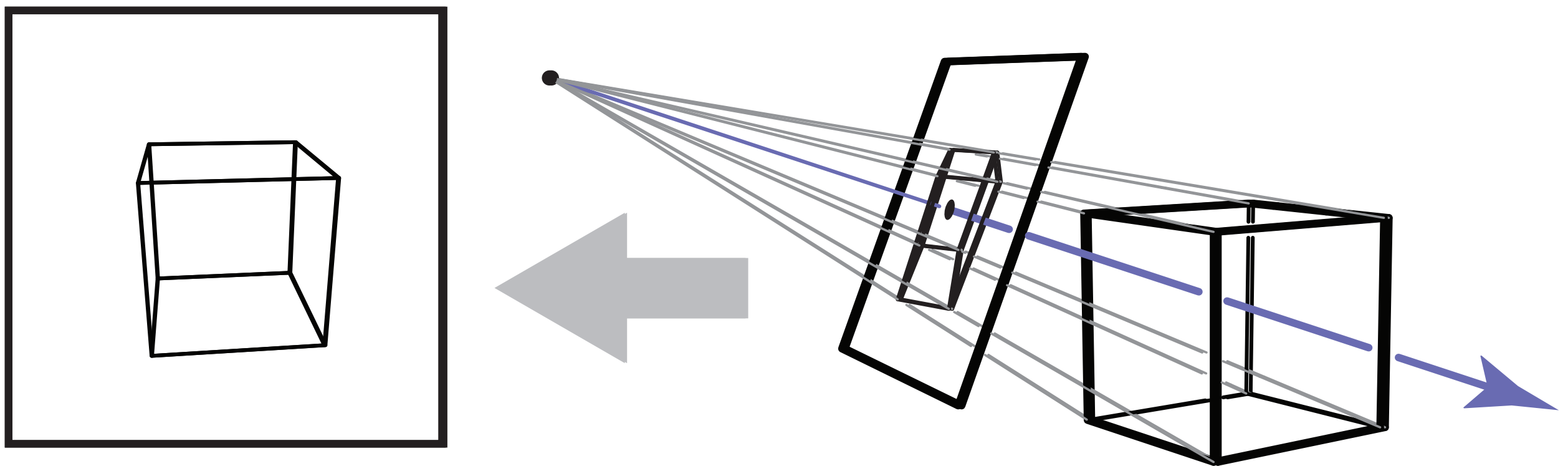
```
for each pixel in the image {  
  for each object in the scene {  
    if (object affects pixel) {  
      do something  
    }  
  }  
}
```

**We will do this first**

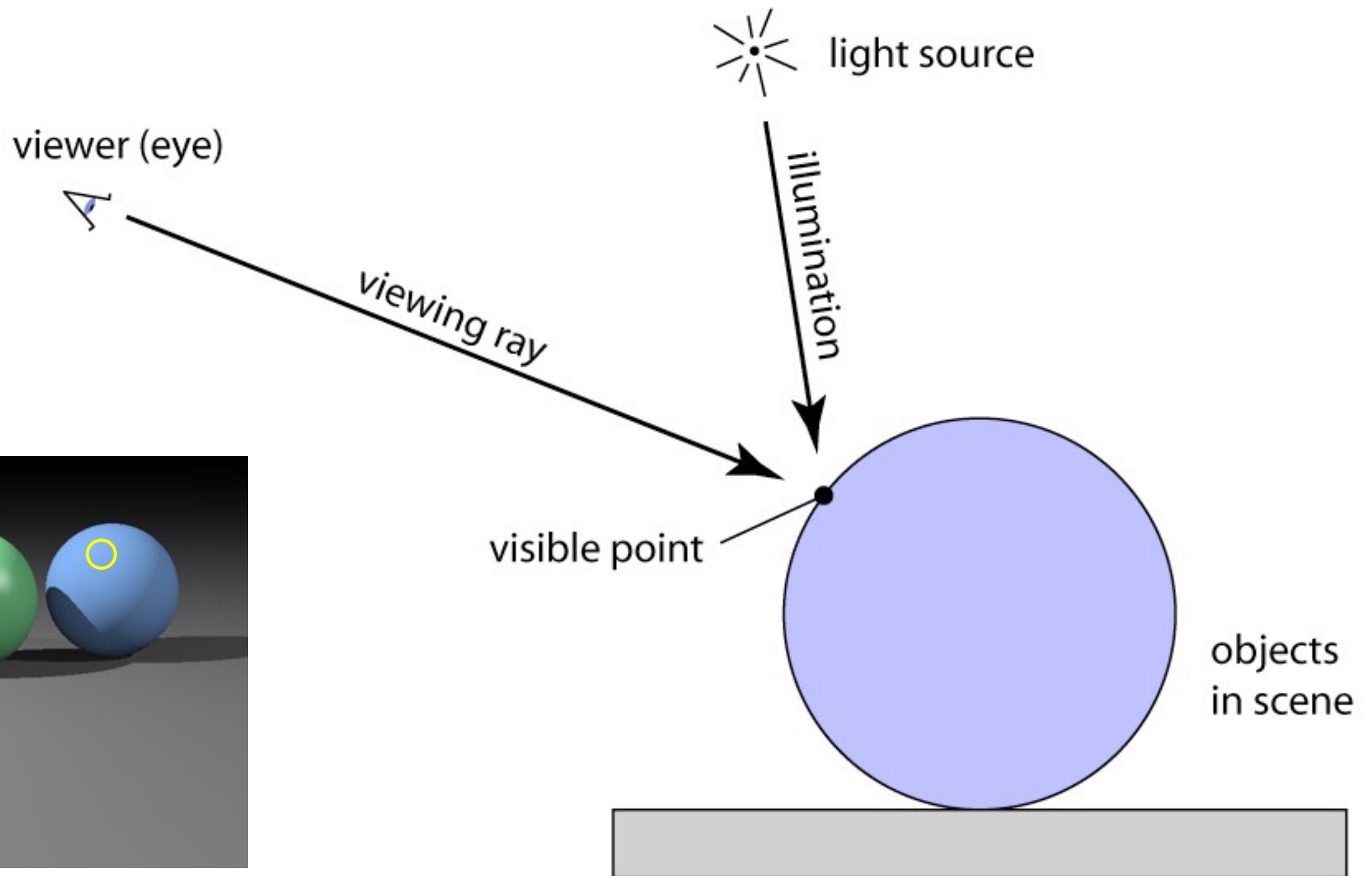
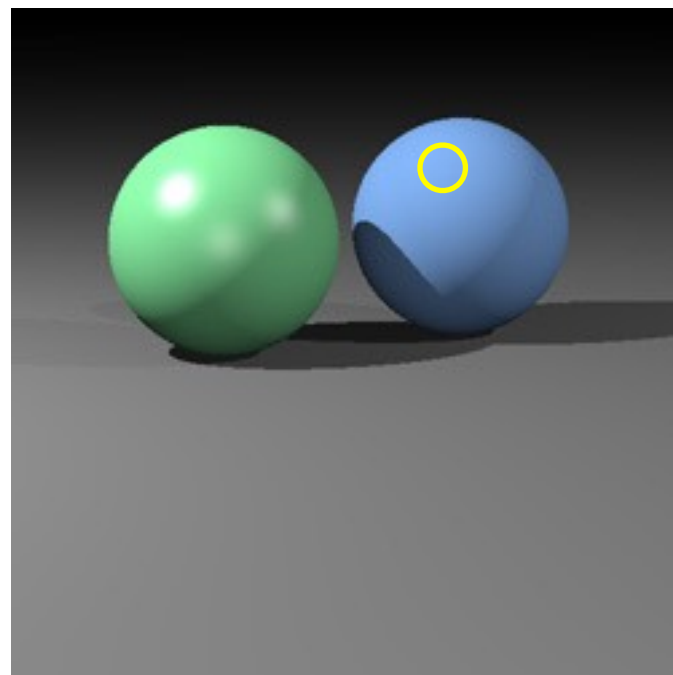
**image order**  
or  
**ray tracing**

# Ray tracing idea

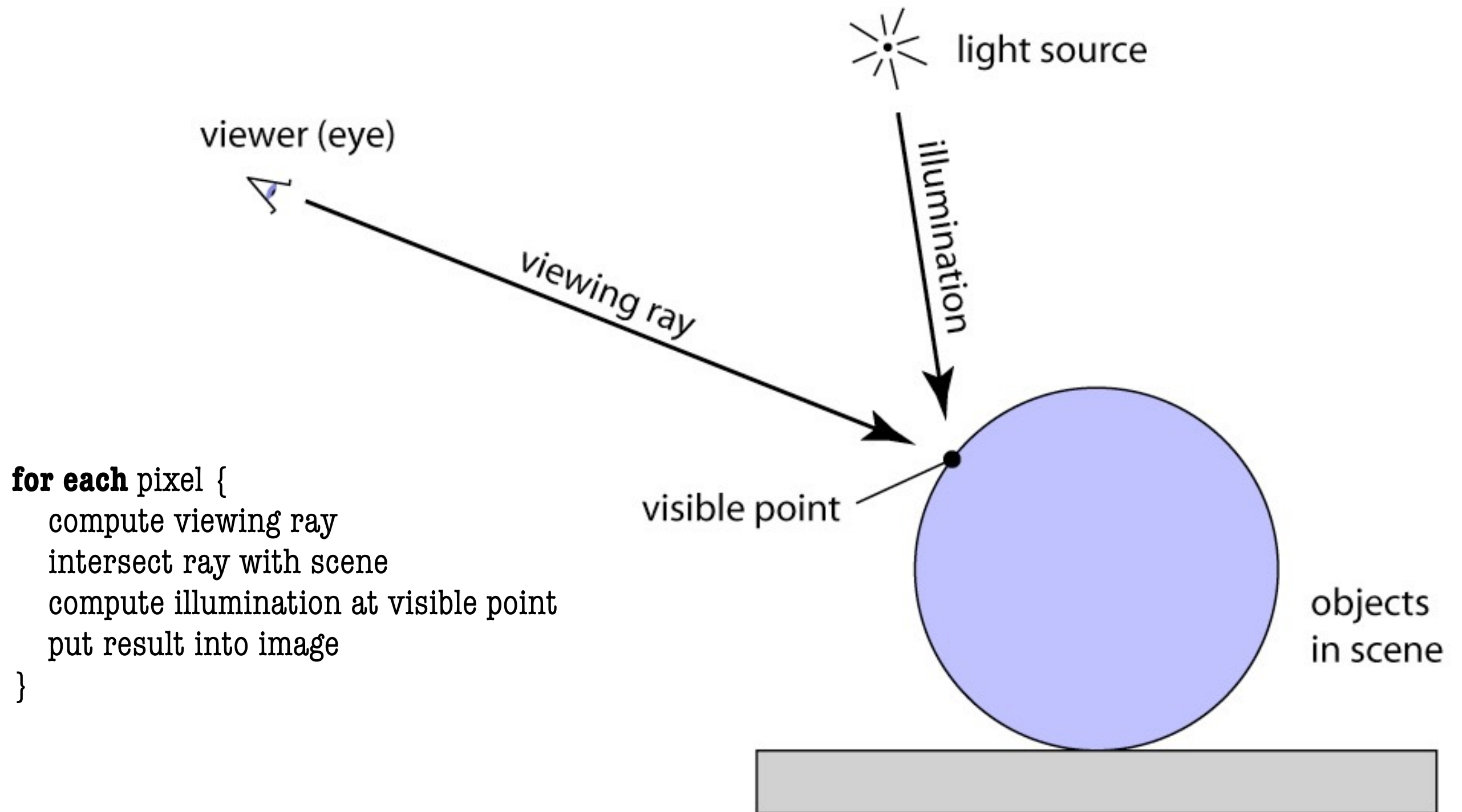
- **Start with a pixel—what belongs at that pixel?**
- **Set of points that project to a point in the image: a ray**



# Ray tracing idea

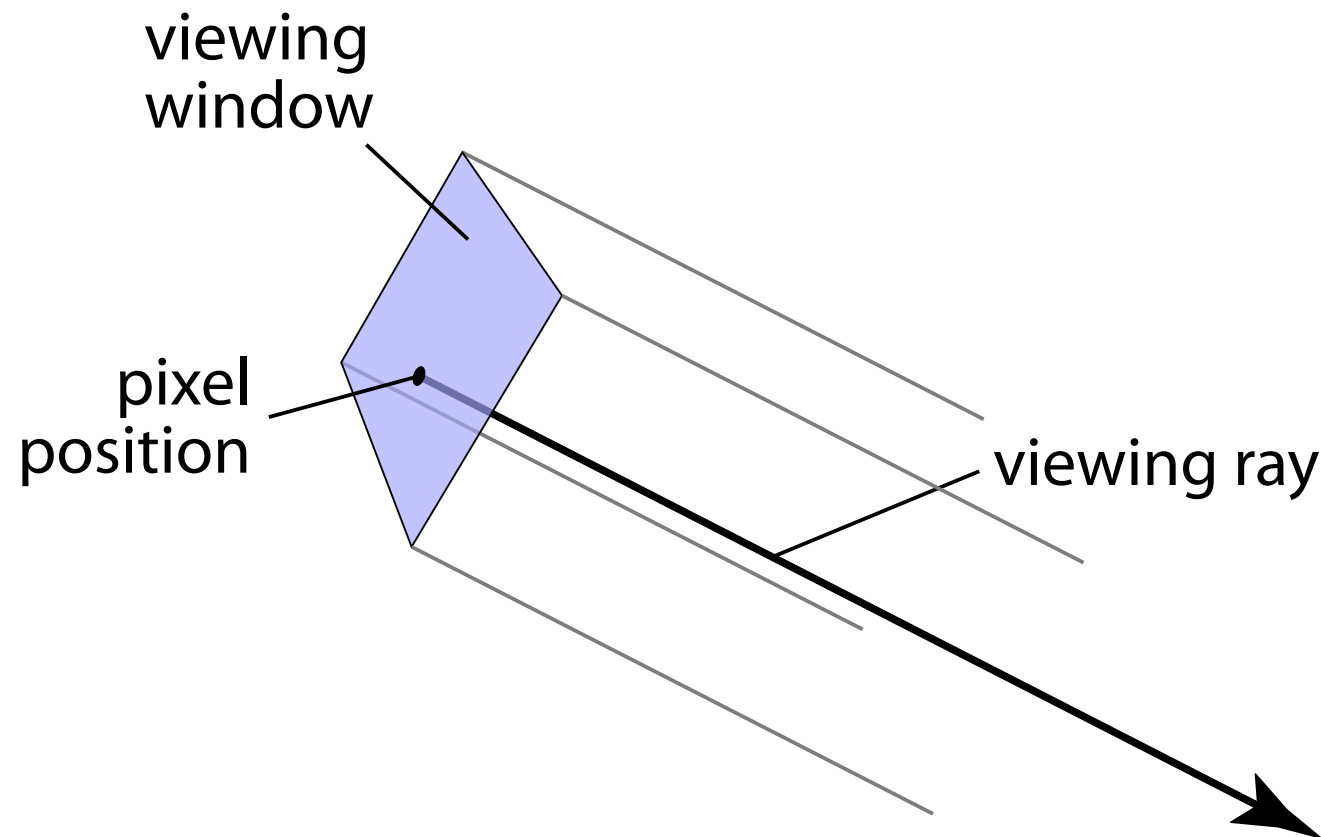


# Ray tracing algorithm



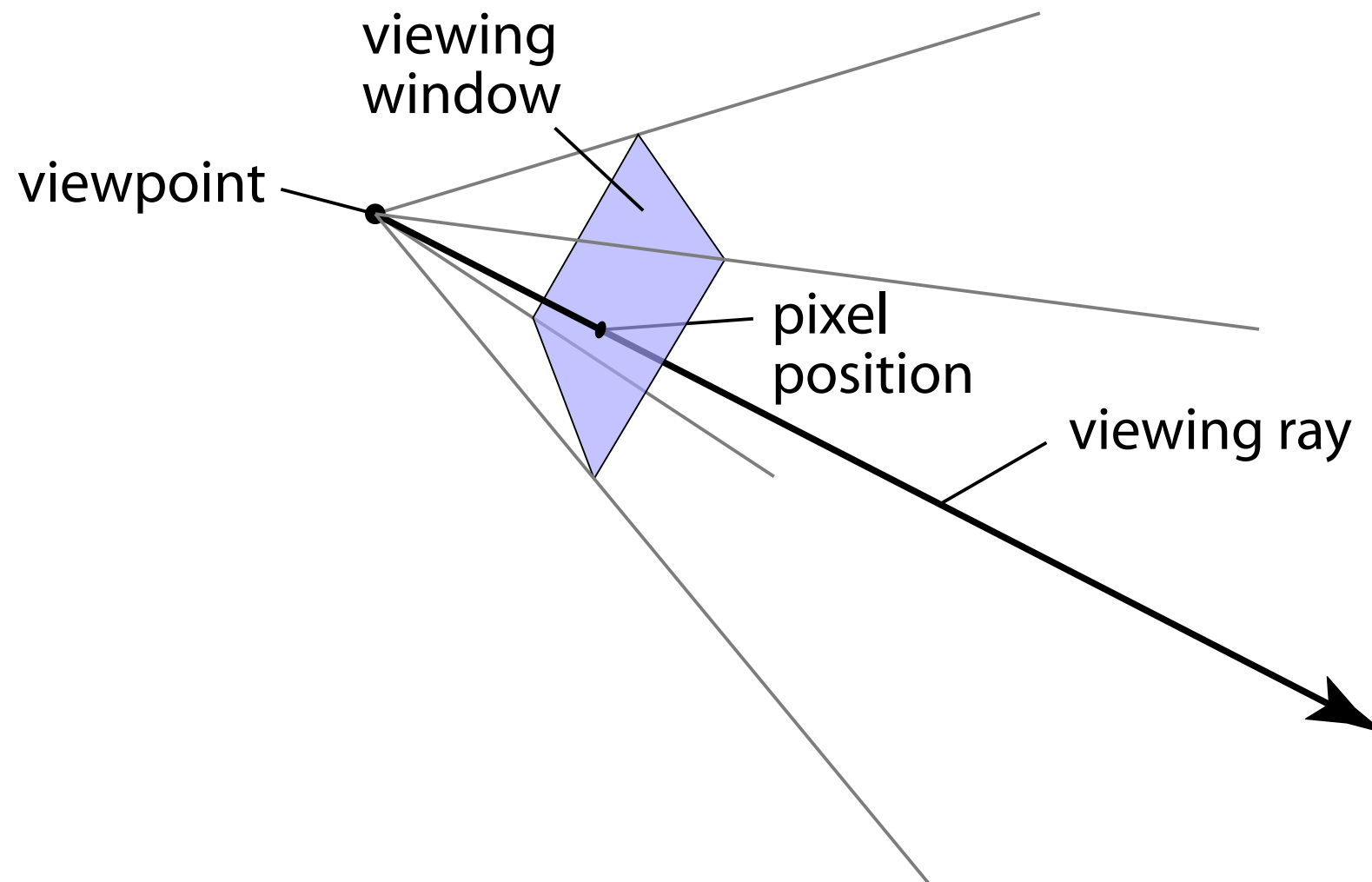
# Generating eye rays—planar projection

- **Ray origin (varying):** pixel position on viewing window
- **Ray direction (constant):** view direction



# Generating eye rays—perspective

- **Ray origin (constant):** viewpoint
- **Ray direction (varying):** toward pixel position on viewing window





# Software interface for cameras

- **Key operation: generate ray for image position**

```
class Camera {
```

```
    ...
```

```
    Ray generateRay(int col, int row);  
}
```

← args go from 0, 0  
to width - 1, height - 1

- **Modularity problem: Camera shouldn't have to worry about image resolution**
  - better solution: normalized coordinates

```
class Camera {
```

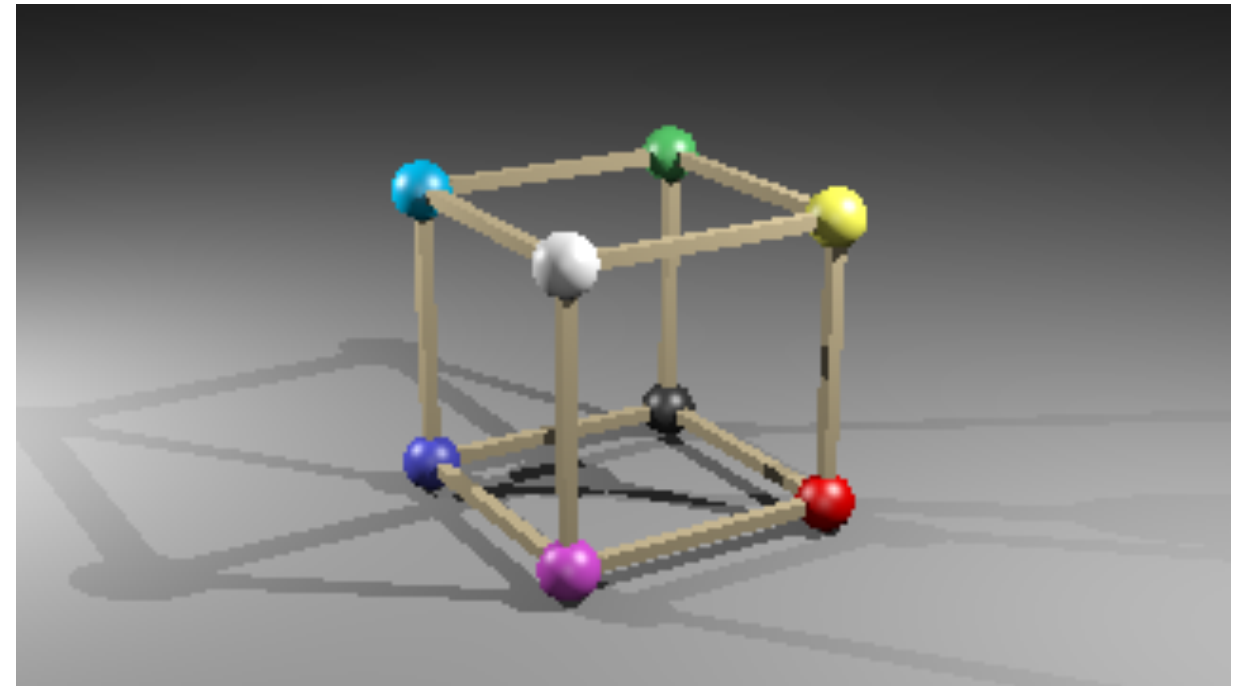
```
    ...
```

```
    Ray generateRay(float u, float v);  
}
```

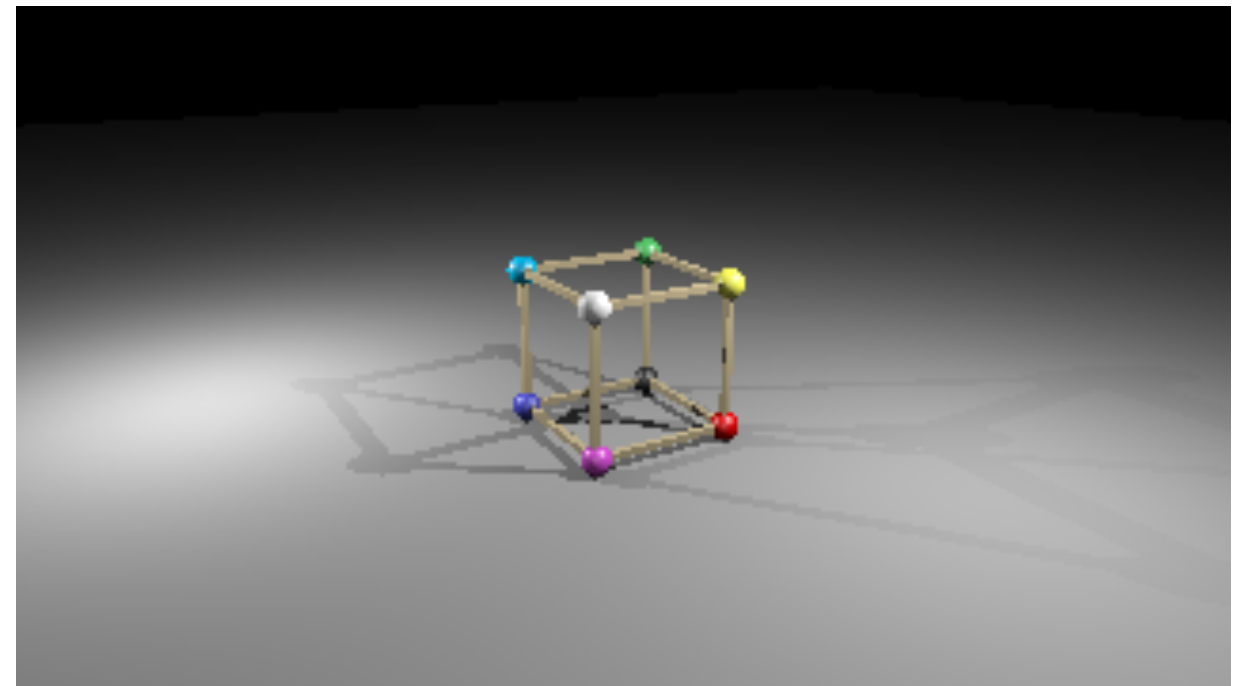
← args go from 0, 0 to 1, 1

# Specifying views in Ray I

```
<camera type="PerspectiveCamera">  
  <viewPoint>10 4.2 6</viewPoint>  
  <viewDir>-5 -2.1 -3</viewDir>  
  <viewUp>0 1 0</viewUp>  
  <projDistance>6</projDistance>  
  <viewWidth>4</viewWidth>  
  <viewHeight>2.25</viewHeight>  
</camera>
```

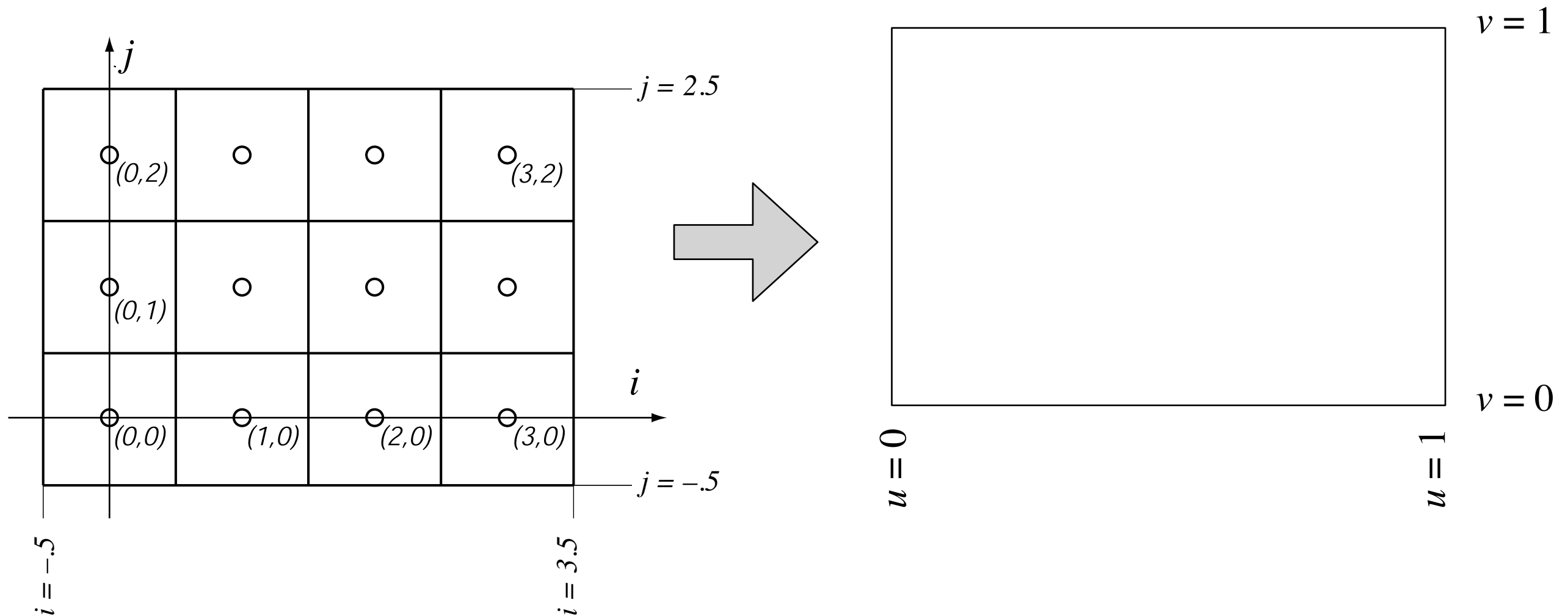


```
<camera type="PerspectiveCamera">  
  <viewPoint>10 4.2 6</viewPoint>  
  <viewDir>-5 -2.1 -3</viewDir>  
  <viewUp>0 1 0</viewUp>  
  <projDistance>3</projDistance>  
  <viewWidth>4</viewWidth>  
  <viewHeight>2.25</viewHeight>  
</camera>
```



# Pixel-to-image mapping

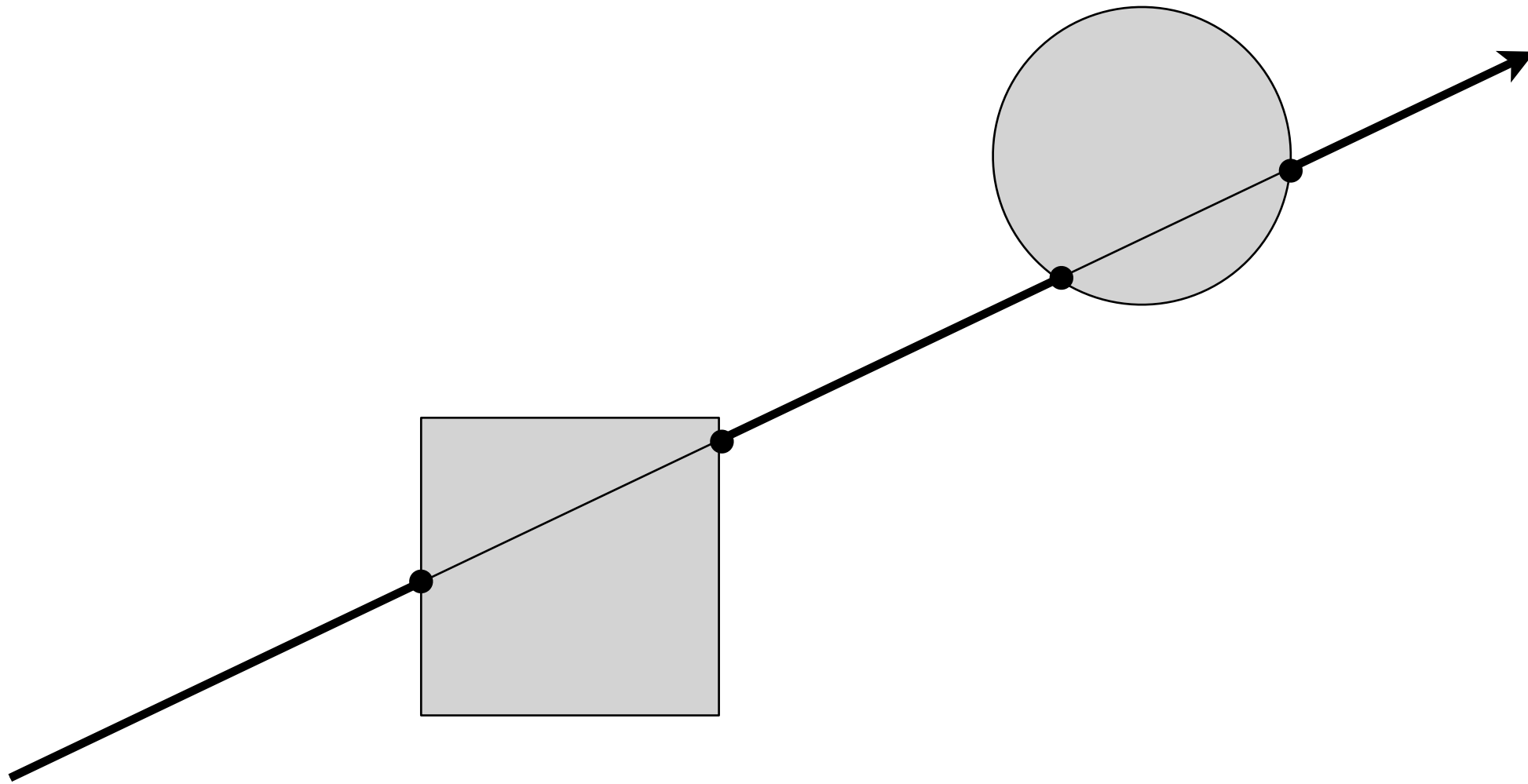
- **One last detail: exactly where are pixels located?**



$$u = (i + 0.5)/n_x$$

$$v = (j + 0.5)/n_y$$

# Ray intersection

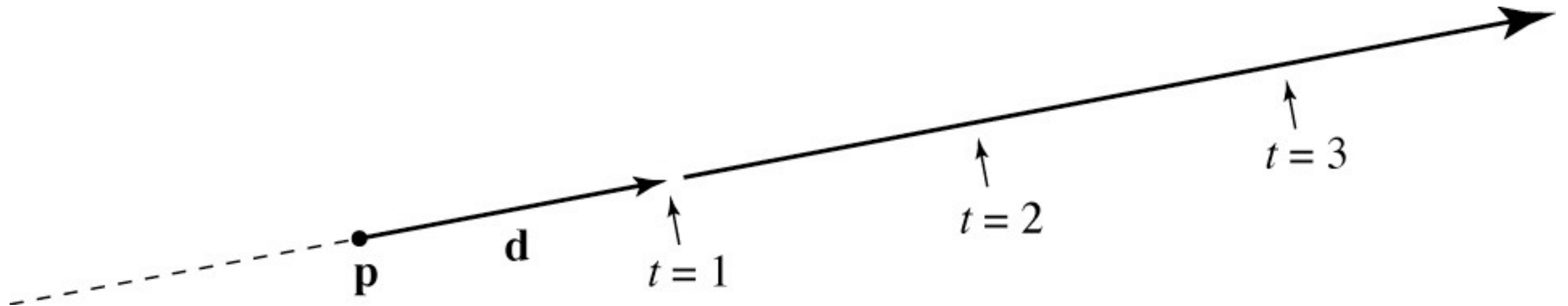


# Ray: a half line

- **Standard representation: point  $\mathbf{p}$  and direction  $\mathbf{d}$**

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

- 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  $\mathbf{d}$  with  $\alpha\mathbf{d}$  doesn't change ray ( $\alpha > 0$ )



# 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; see book or notes for general

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

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

- **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

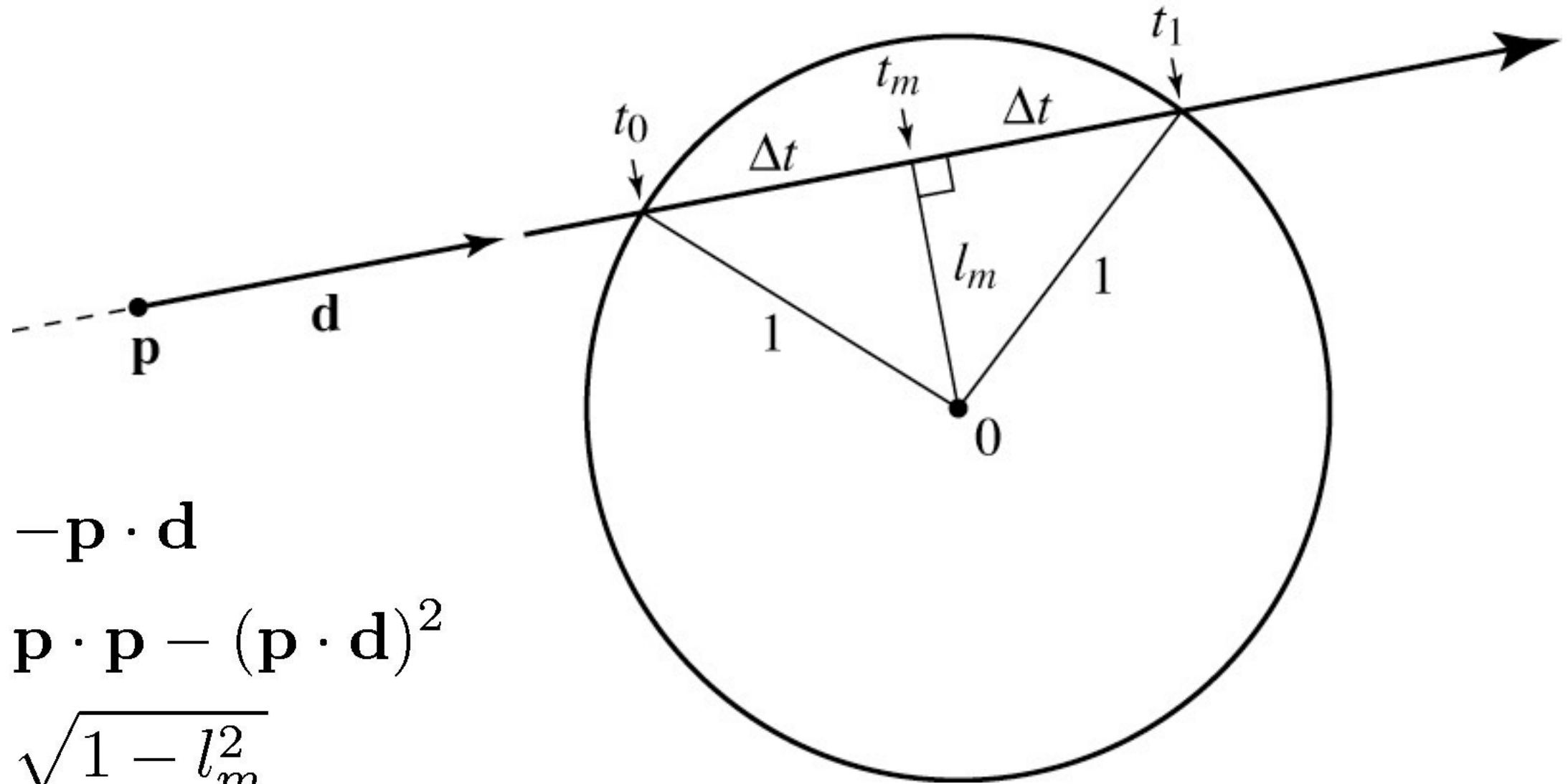
- **Solution for  $t$  by quadratic formula:**

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

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

- 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: geometric



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

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

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

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

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



# Ray-triangle 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 three edges**

- **First solve 1&2 (ray-plane intersection)**

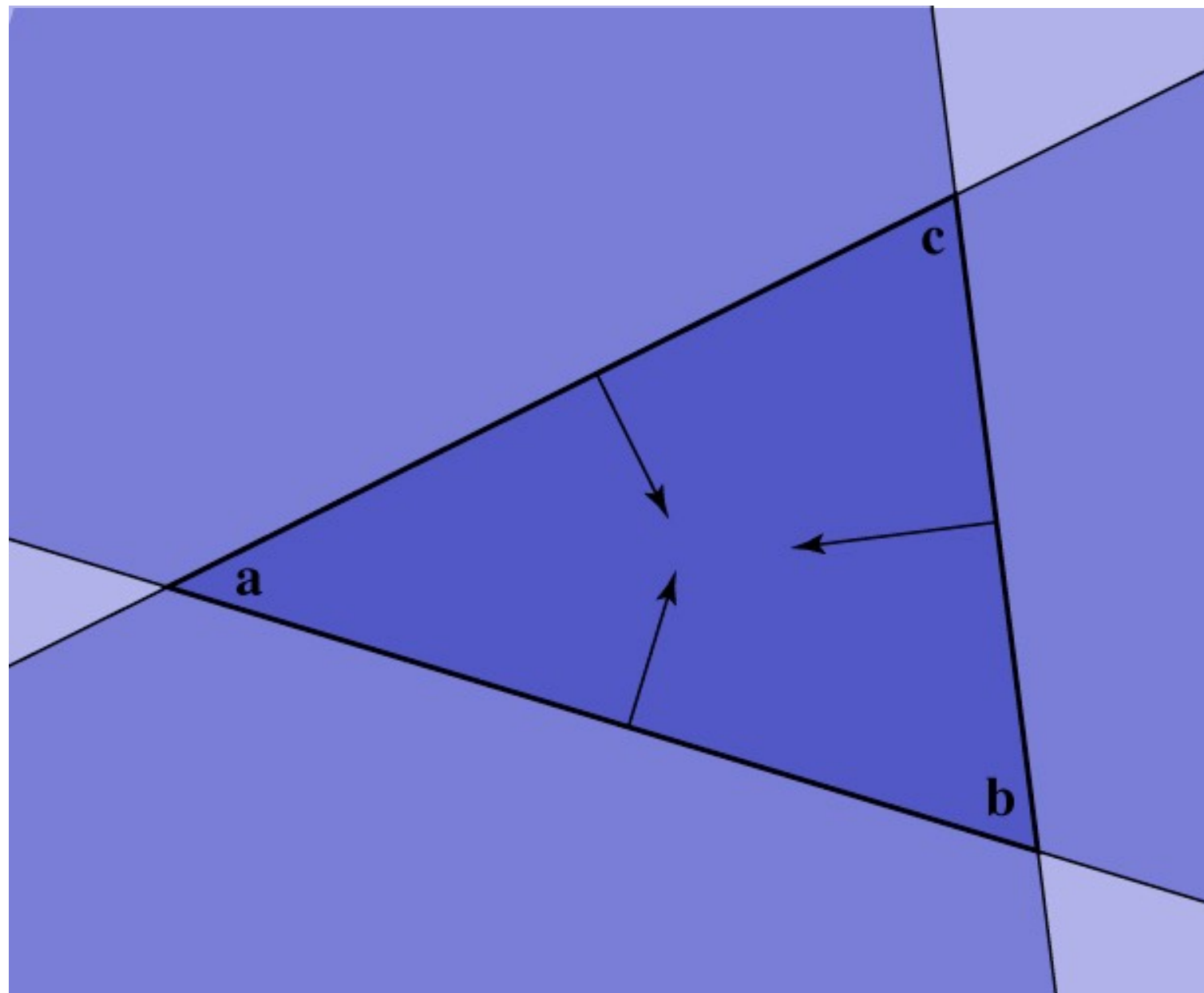
– substitute and solve for  $t$ :

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

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

# Ray-triangle intersection

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



# Deciding about insideness

- **Need to check whether hit point is inside 3 edges**
  - easiest to do in 2D coordinates on the plane
- **Will also need to know where we are in the triangle**
  - for textures, shading, etc. ... next couple of lectures
- **Efficient solution: transform to coordinates aligned to the triangle**

# Barycentric coordinates

- **A coordinate system for triangles**

- algebraic viewpoint:

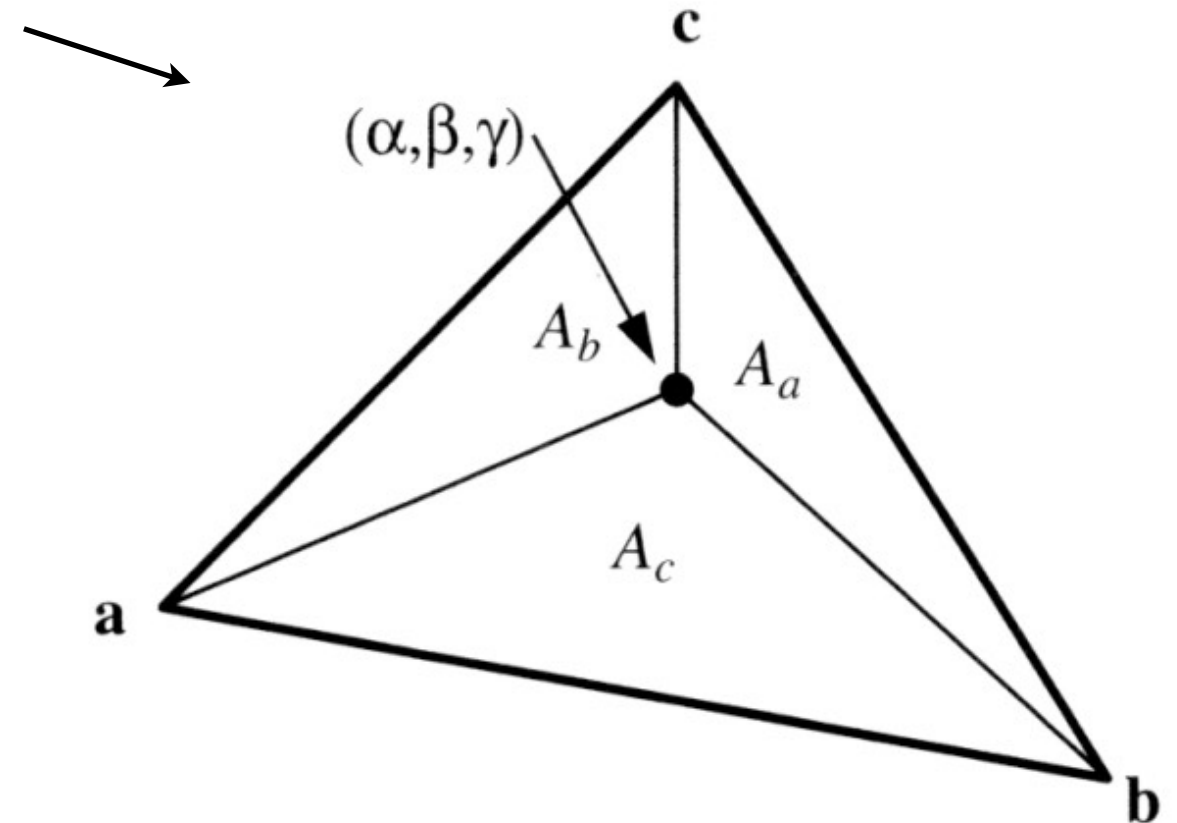
$$\mathbf{p} = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}$$

$$\alpha + \beta + \gamma = 1$$

- geometric viewpoint (areas):

- **Triangle interior test:**

$$\alpha > 0; \quad \beta > 0; \quad \gamma > 0$$

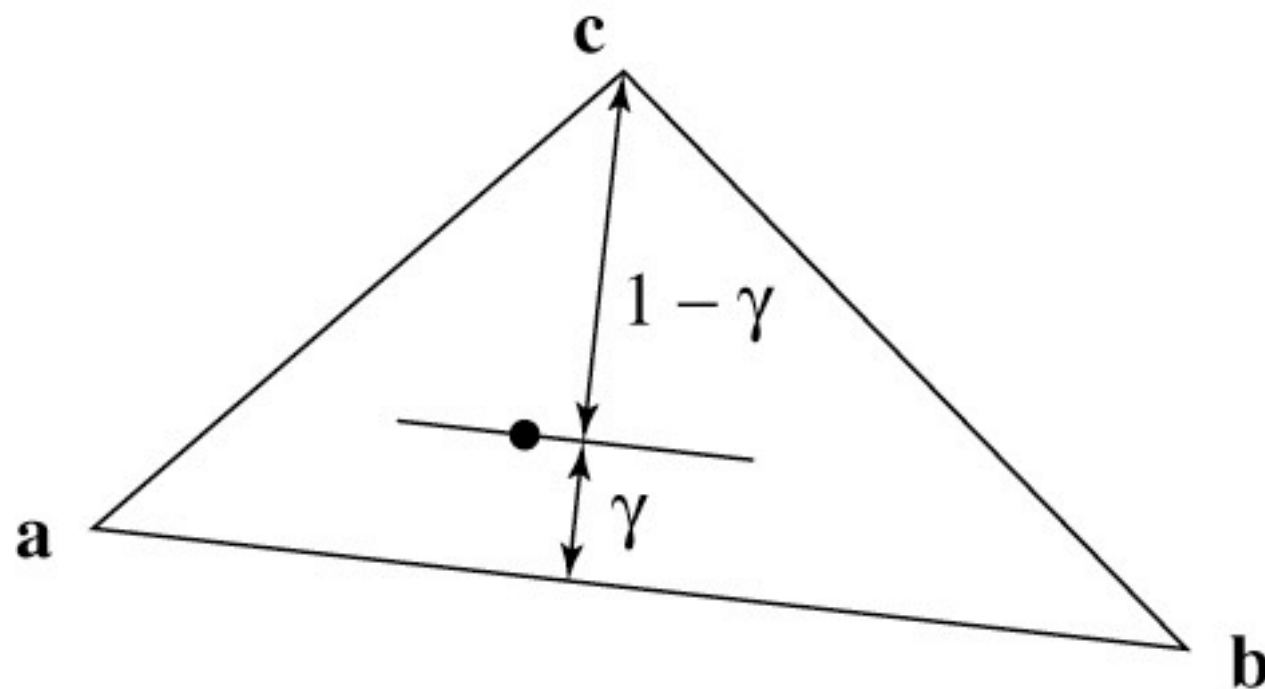


[Shirley 2000]

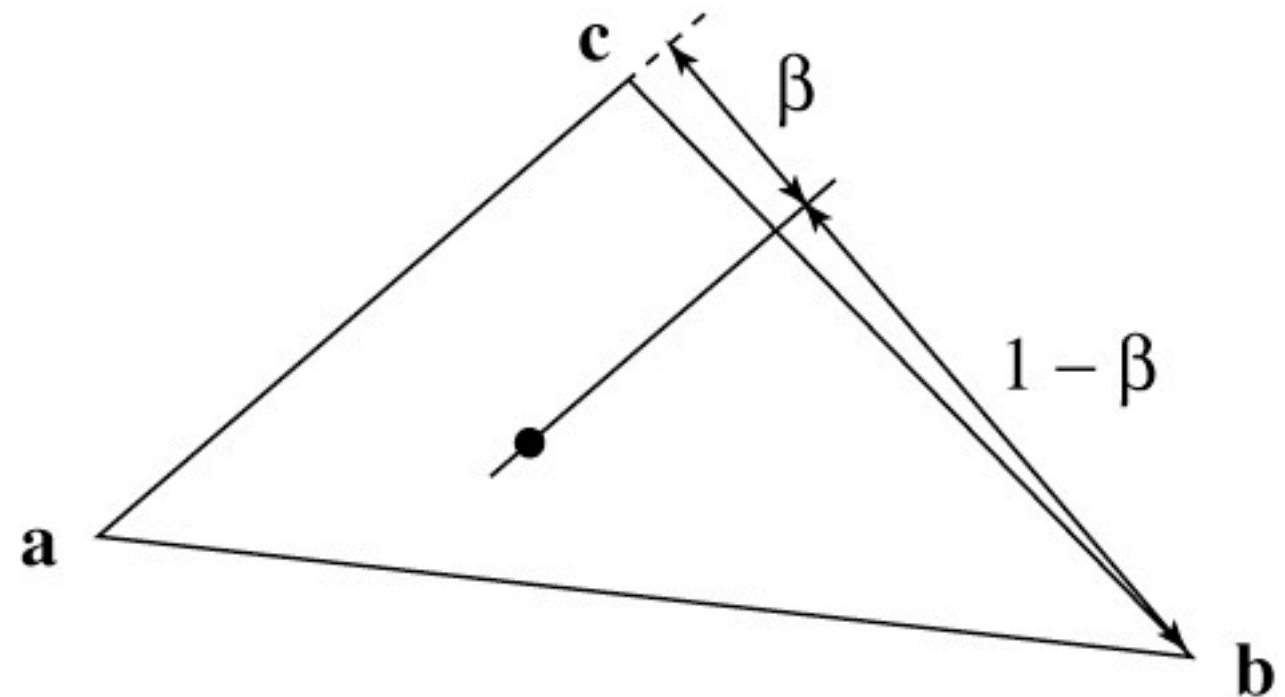
# Barycentric coordinates

- **A coordinate system for triangles**

- geometric viewpoint: distances



- linear viewpoint: basis of edges

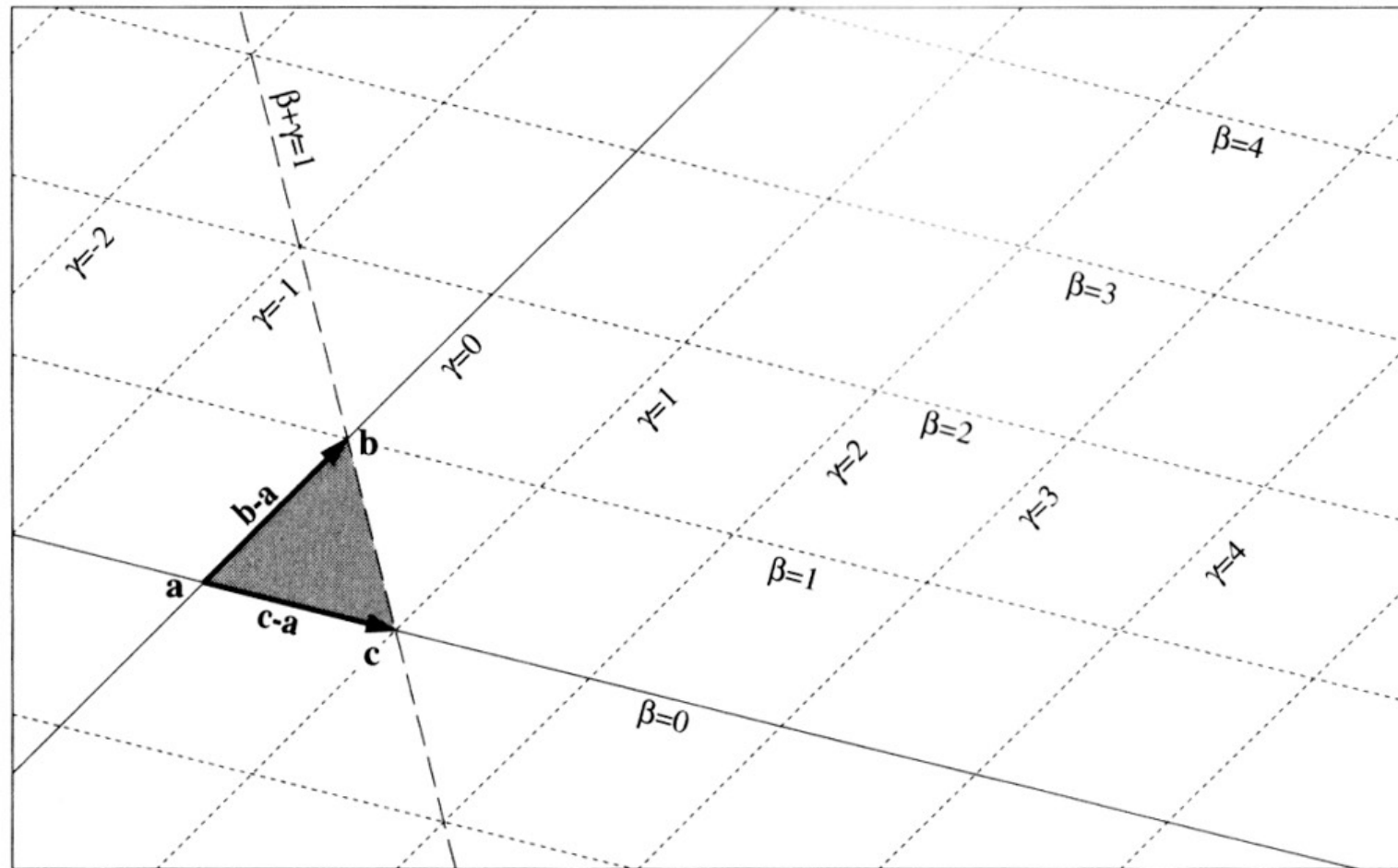


$$\alpha = 1 - \beta - \gamma$$

$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

# Barycentric coordinates

- **Linear viewpoint: basis for the plane**



– in this view, the triangle interior test is just

$$\beta > 0; \quad \gamma > 0; \quad \beta + \gamma < 1$$

# Barycentric ray-triangle intersection

- **Every point on the plane can be written in the form:**

$$\mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

**for some numbers  $\beta$  and  $\gamma$ .**

- **If the point is also on the ray then it is**

$$\mathbf{p} + t\mathbf{d}$$

**for some number  $t$ .**

- **Set them equal: 3 linear equations in 3 variables**

$$\mathbf{p} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

**...solve them to get  $t$ ,  $\beta$ , and  $\gamma$  all at once!**

# Barycentric ray-triangle intersection

$$\mathbf{p} + t\mathbf{d} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

$$\beta(\mathbf{a} - \mathbf{b}) + \gamma(\mathbf{a} - \mathbf{c}) + t\mathbf{d} = \mathbf{a} - \mathbf{p}$$

$$\begin{bmatrix} \mathbf{a} - \mathbf{b} & \mathbf{a} - \mathbf{c} & \mathbf{d} \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \mathbf{a} - \mathbf{p}$$

$$\begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} x_a - x_p \\ y_a - y_p \\ z_a - z_p \end{bmatrix}$$

Cramer's rule is a good fast way to solve this system  
(see text Ch. 2 and Ch. 4 for details)



# Ray intersection in software

- **All surfaces need to be able to intersect rays with themselves.**

```
class Surface {  
    ...  
    abstract boolean intersect(IntersectionRecord result, Ray r);  
}
```

was there an  
intersection?

information about  
first intersection

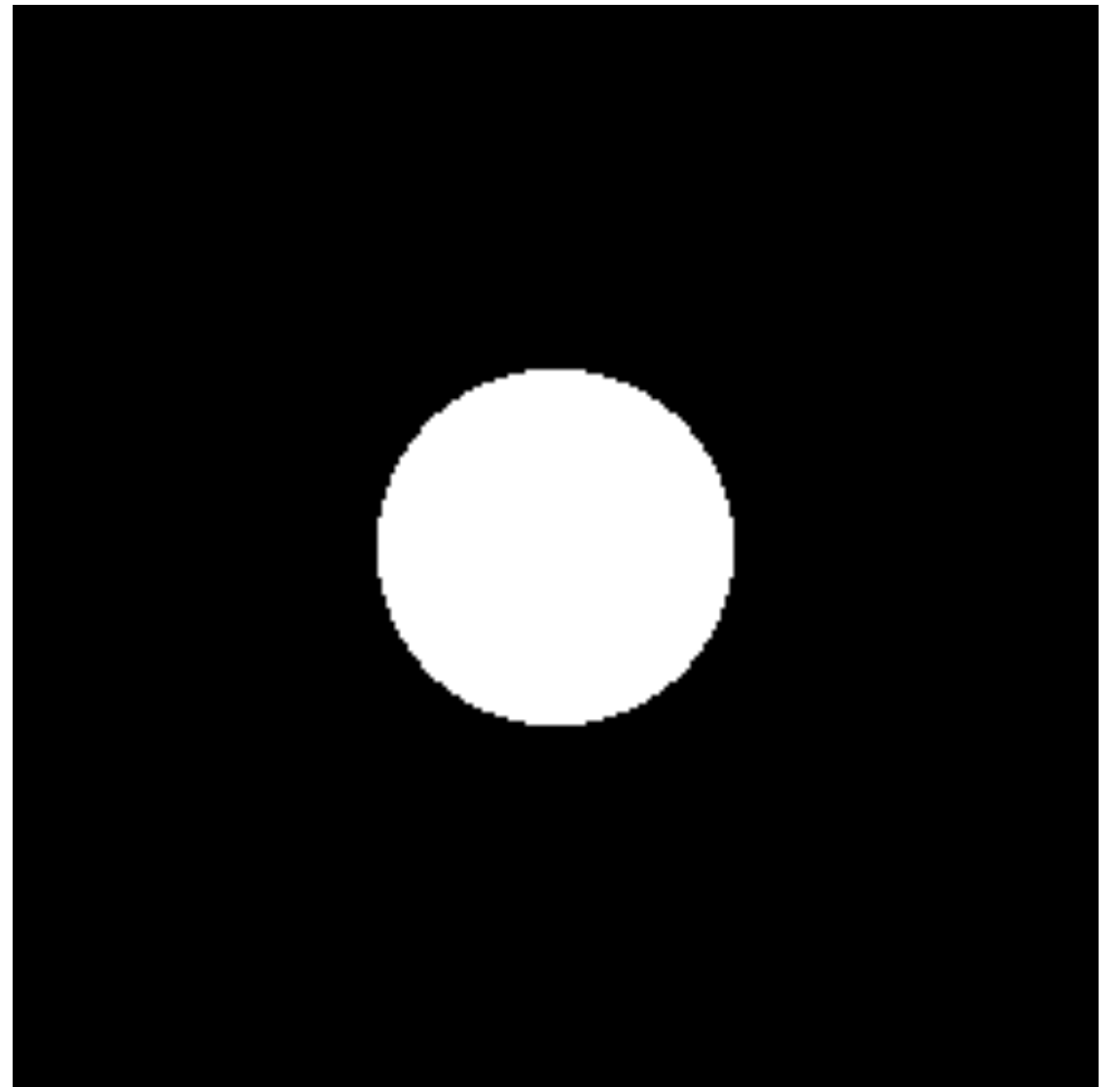
ray to be  
intersected

```
class IntersectionRecord {  
    float t;  
    Vector3 hitLocation;  
    Vector3 normal;  
    ...  
}
```

# Image so far

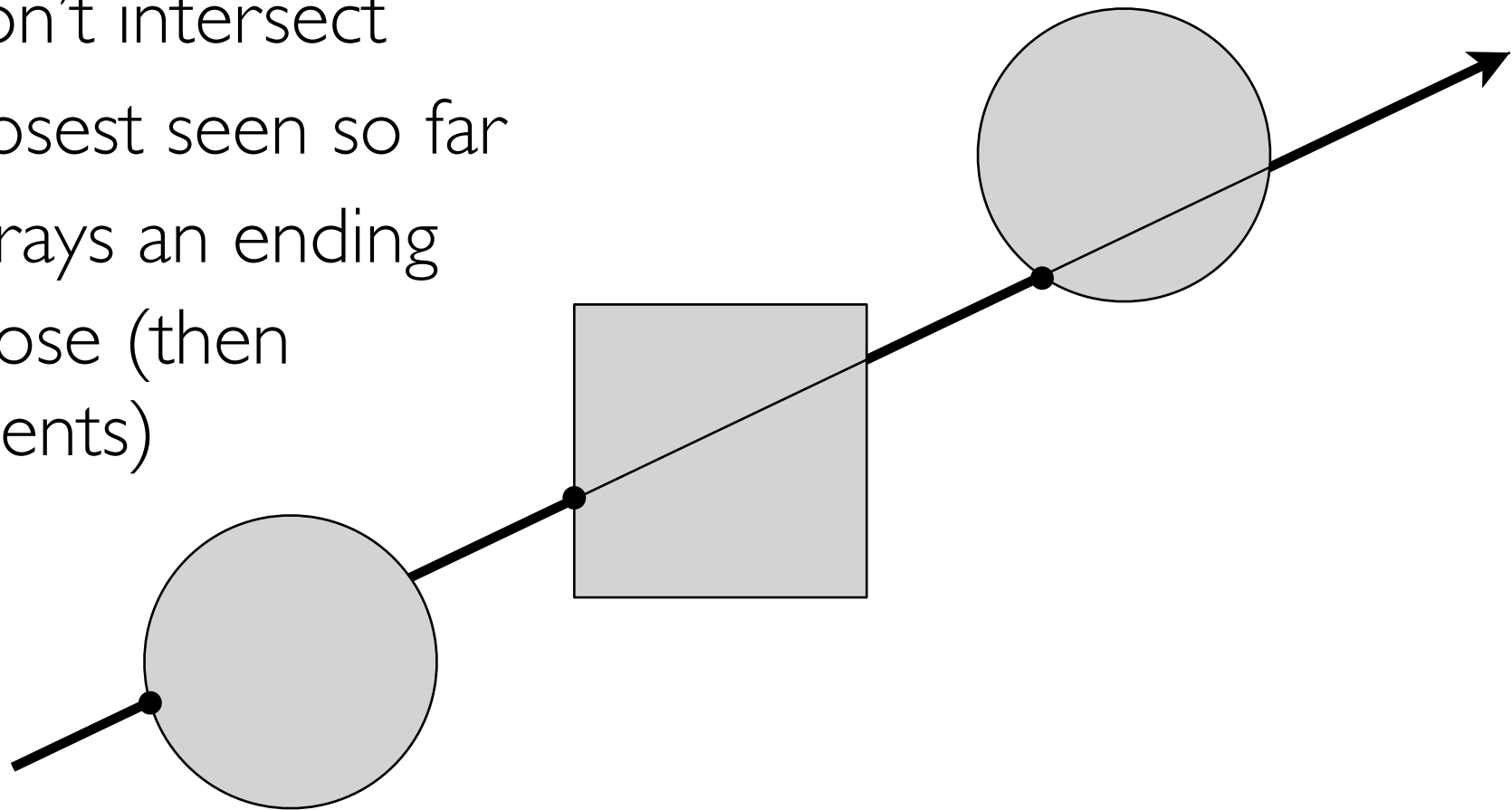
- **With eye ray generation and sphere intersection**

```
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);
    hitSurface, t = s.intersect(ray, 0, +inf)
    if hitSurface is not null
      image.set(ix, iy, white);
  }
```



# Ray intersection in software

- **Scenes usually have many objects**
- **Need to find the first intersection along the ray**
  - that is, the one with the smallest positive  $t$  value
- **Loop over objects**
  - ignore those that don't intersect
  - keep track of the closest seen so far
  - Convenient to give rays an ending  $t$  value for this purpose (then they are really segments)



# Intersection against many shapes

- **The basic idea is:**

```
intersect (ray, tMin, tMax) {  
    tBest = +inf; firstSurface = null;  
    for surface in surfaceList {  
        hitSurface, t = surface.intersect(ray, tMin, tBest);  
        if hitSurface is not null {  
            tBest = t;  
            firstSurface = hitSurface;  
        }  
    }  
    return hitSurface, tBest;  
}
```

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