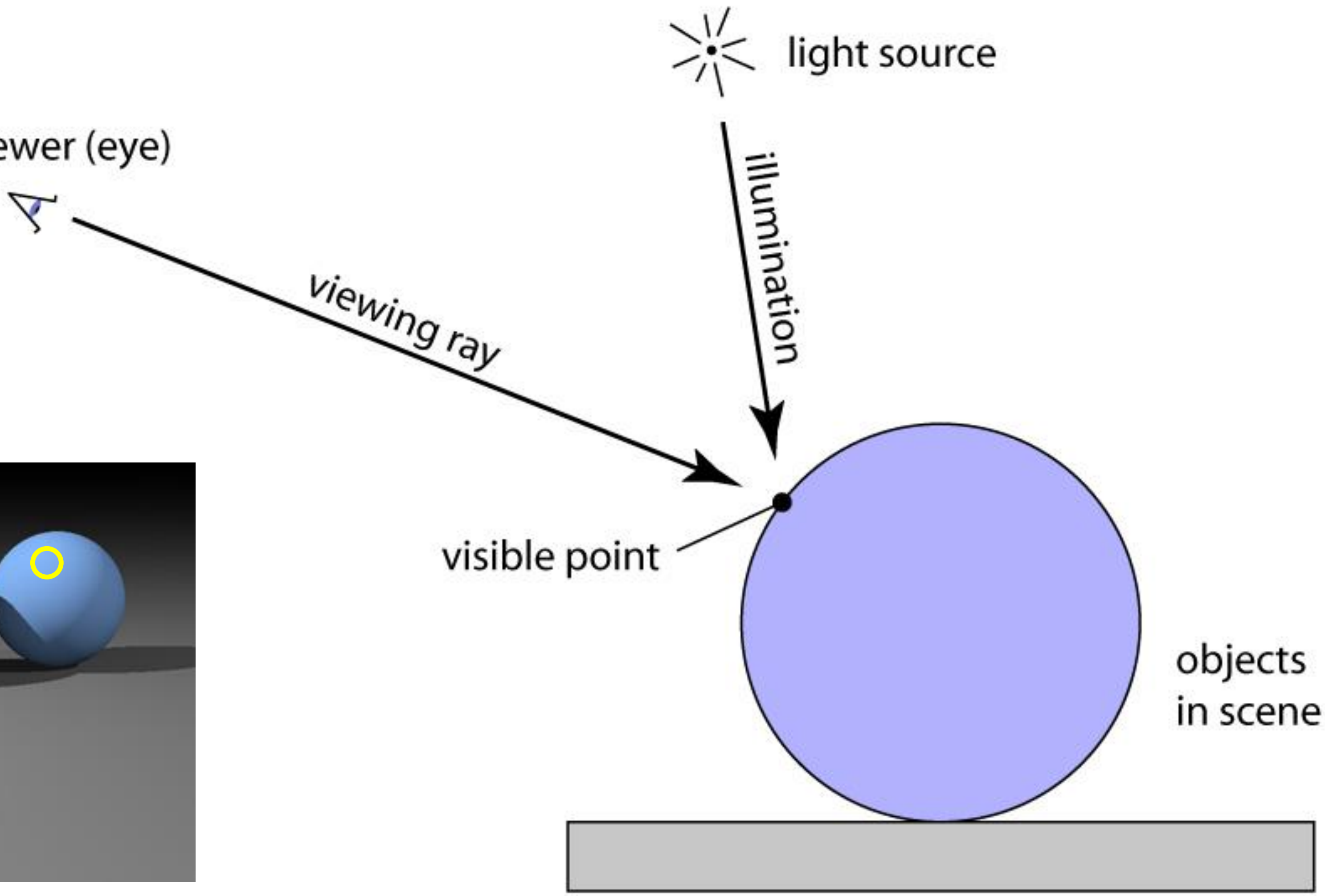
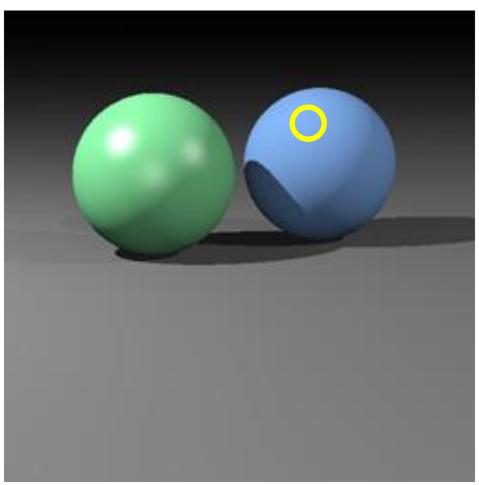
Ray Tracing

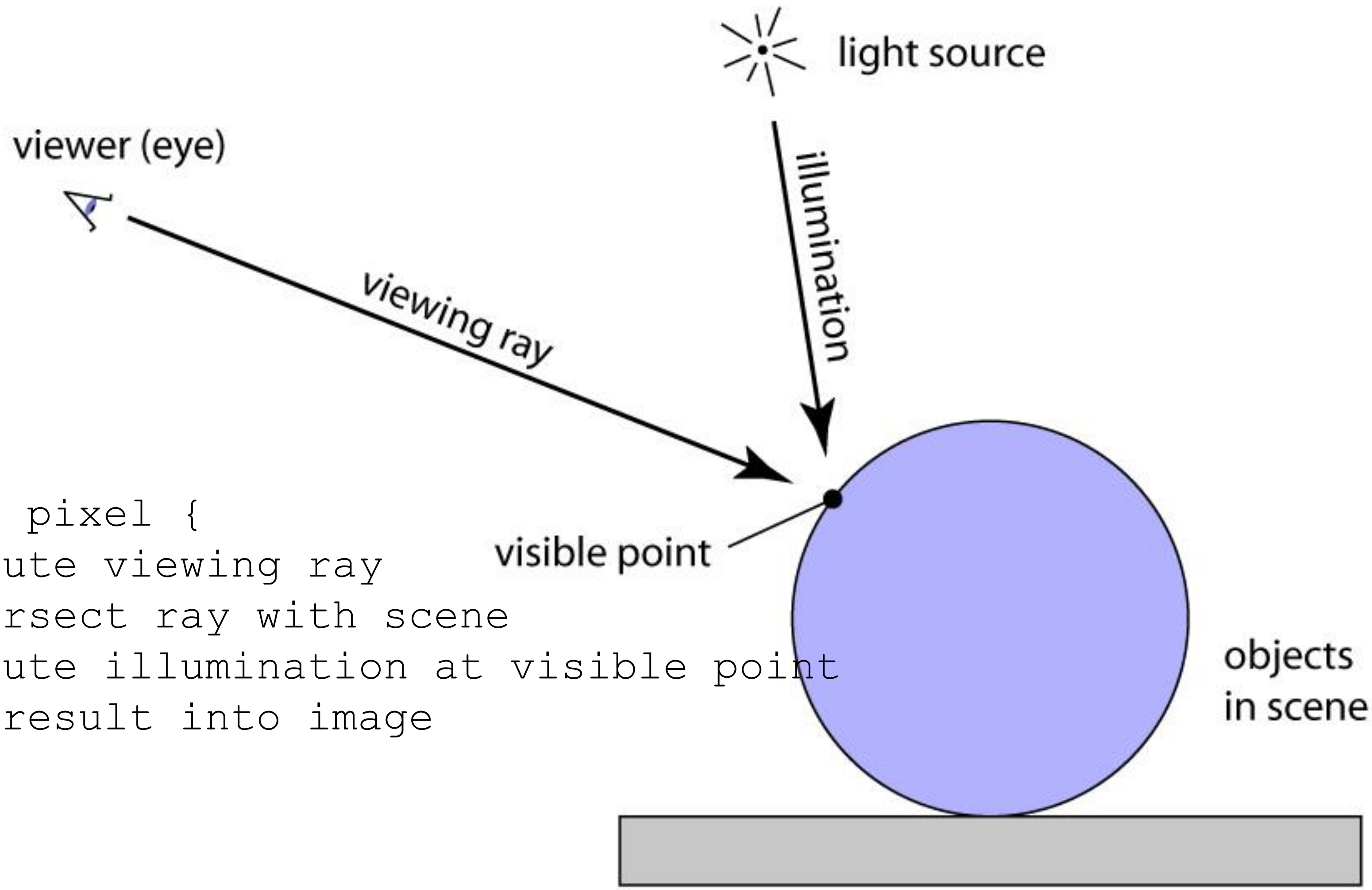
COMP557
Paul Kry

Ray tracing idea

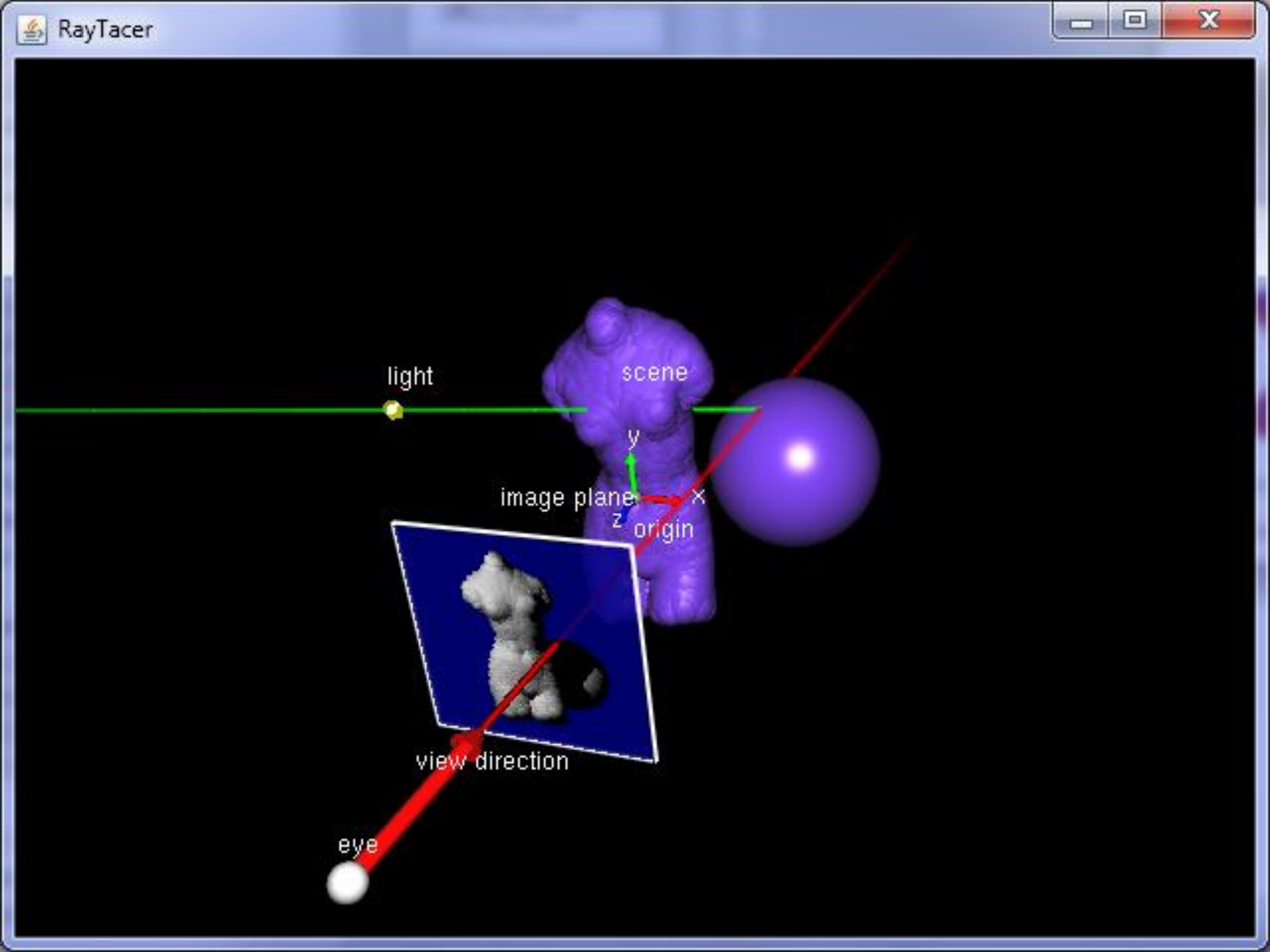




Ray tracing algorithm

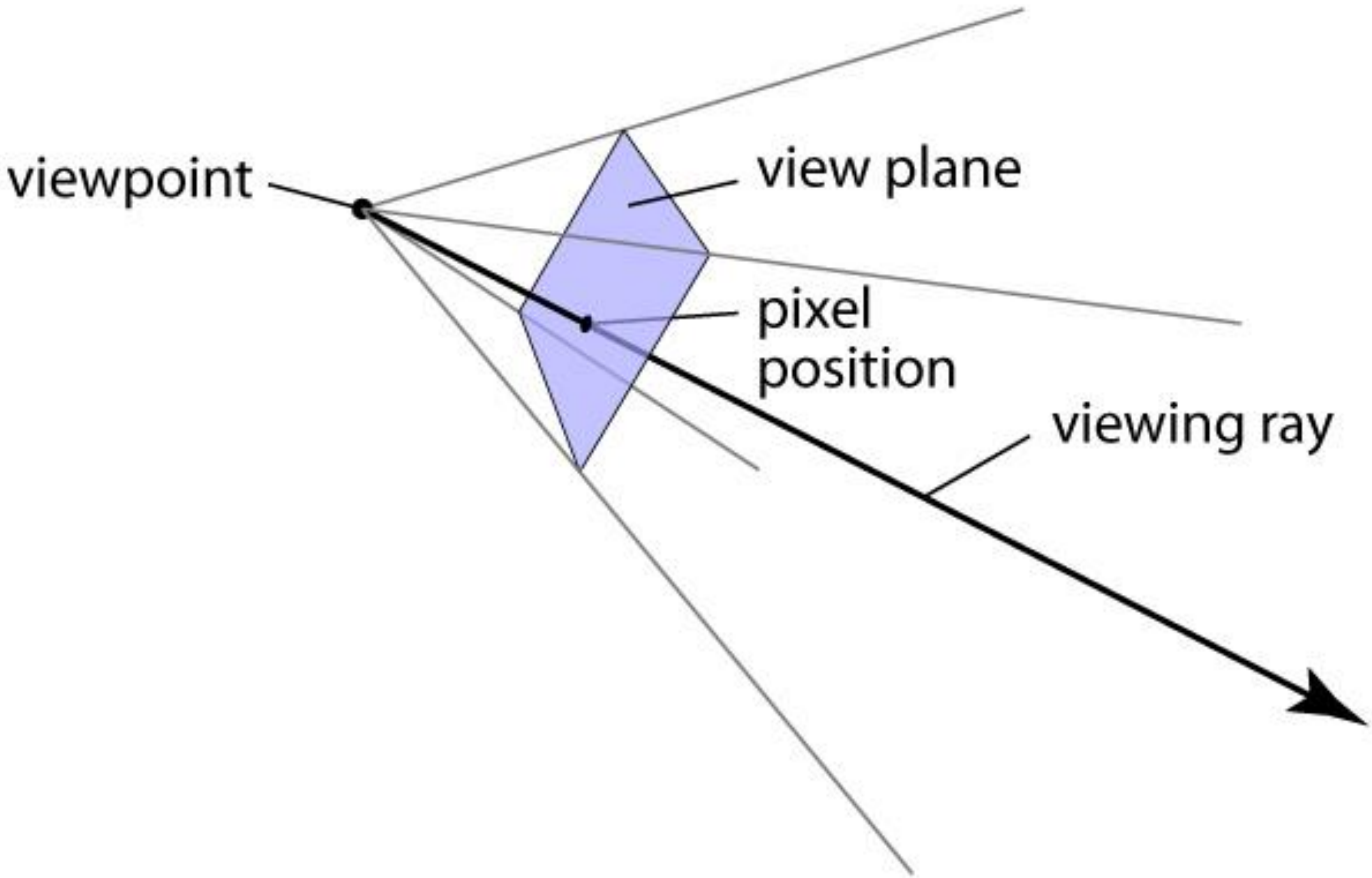



```
for each pixel {
    compute viewing ray
    intersect ray with scene
    compute illumination at visible point
    put result into image
}
```

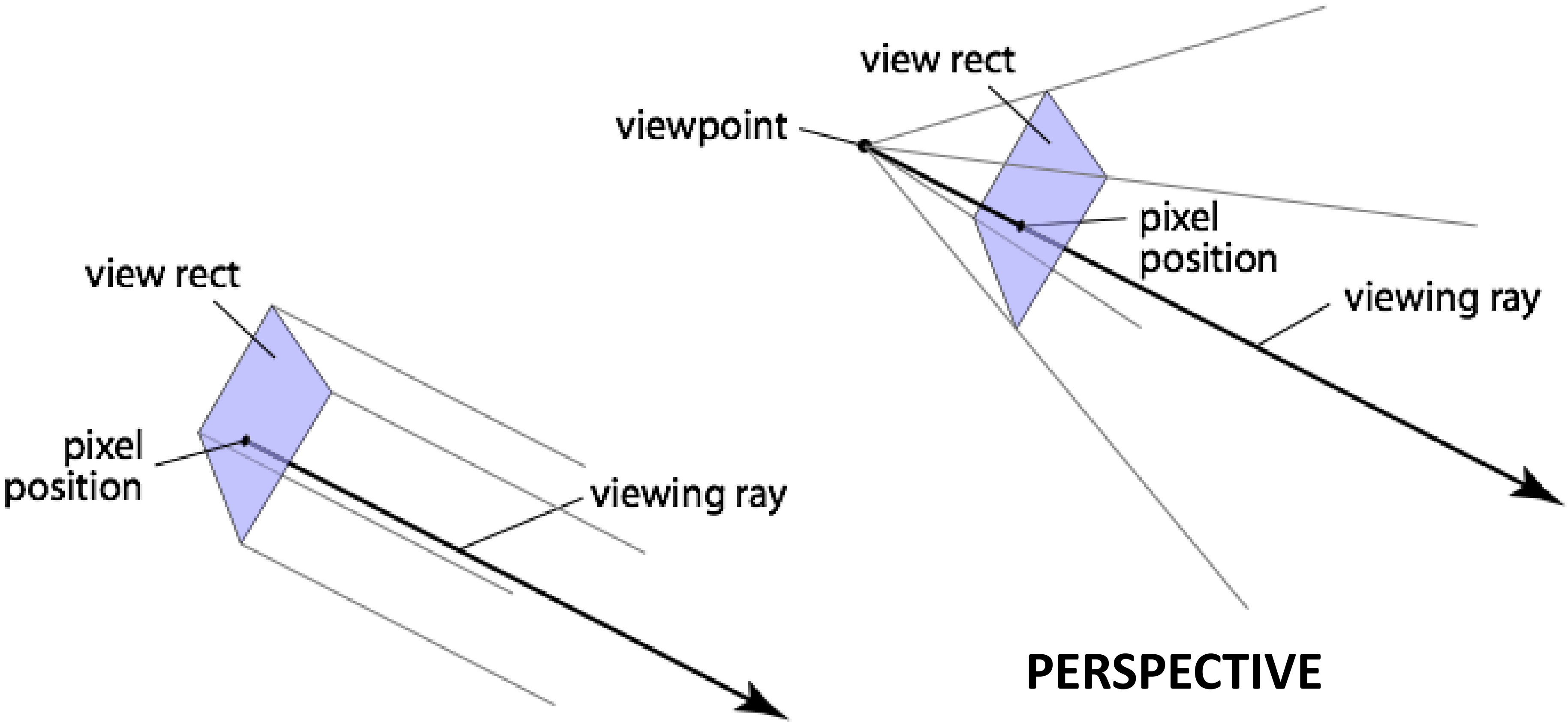

RayTacer
light
scene
y
image plane
x
z
origin
view direction
eye

Generating eye rays

- Use window analogy directly



Generating eye rays

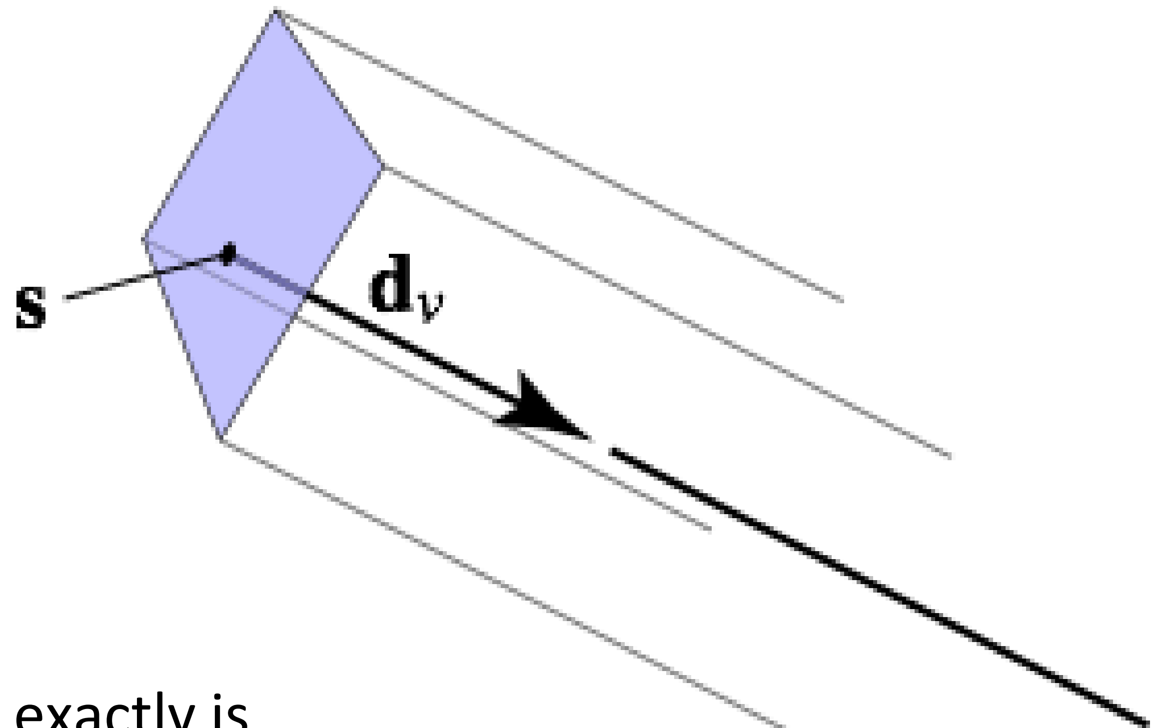


ORTHOGRAPHIC

PERSPECTIVE

Generating eye rays—orthographic

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



  - but where exactly is the view rectangle?

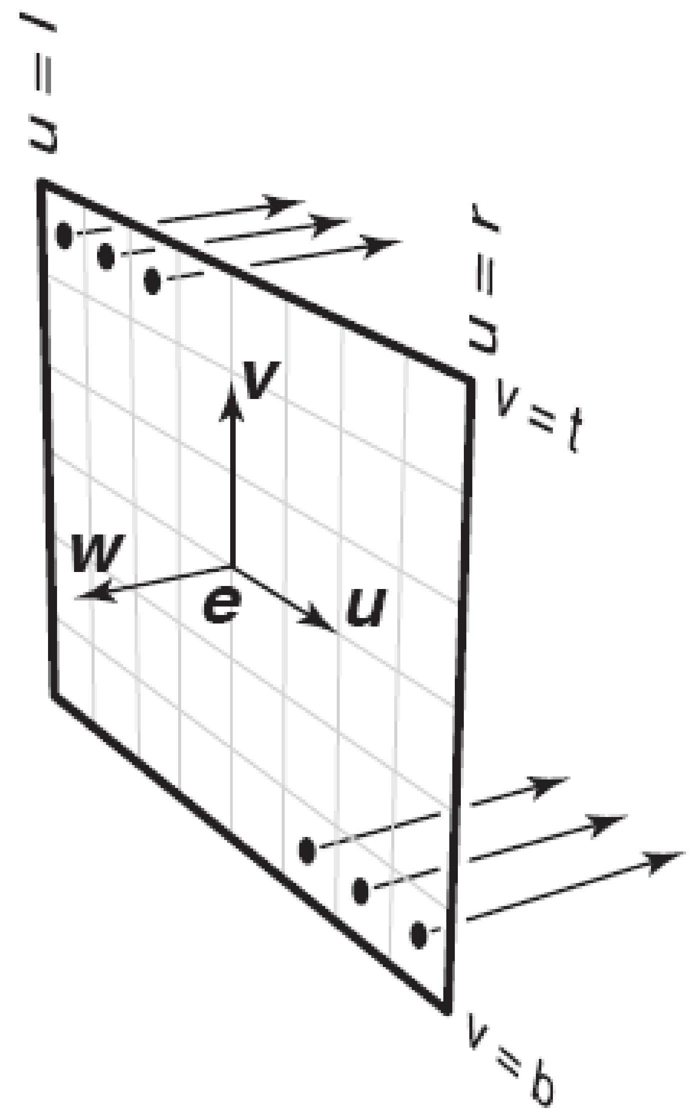
$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = \mathbf{d}_v$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

Generating eye rays—orthographic

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$

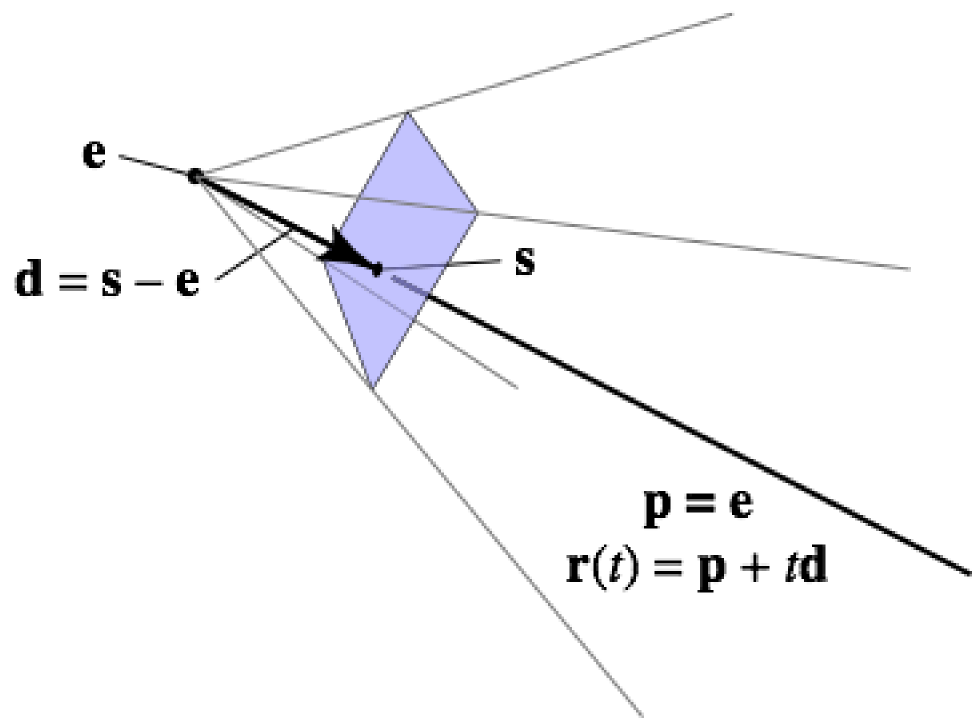
$$\mathbf{p} = \mathbf{s};\ \mathbf{d} = -\mathbf{w}$$

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



Generating eye rays—perspective

- View rectangle needs to be away from viewpoint
- Distance is important: “focal length” of camera
  - still use camera frame but position view rectangle away from viewpoint
  - ray origin always **e**
  - ray direction now controlled by **s**



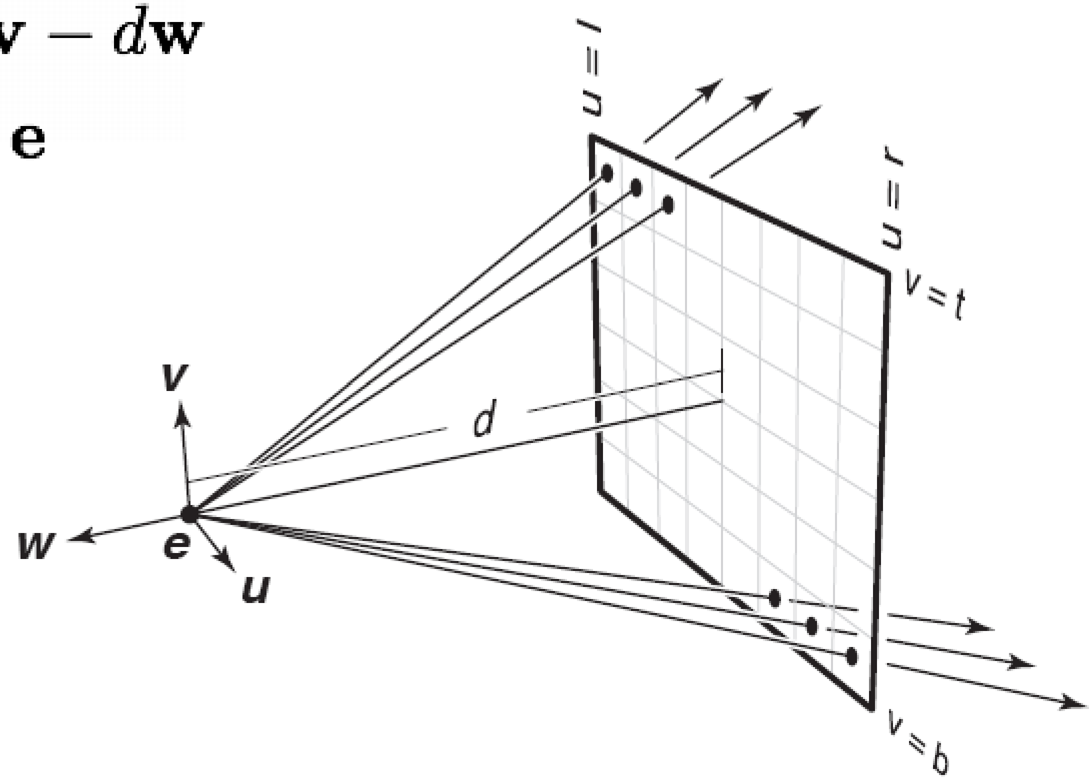
Generating eye rays—perspective

- Compute **s** in the same way; just subtract *d***w**
  - coordinates of **s** are (u, v, $-d$)

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$

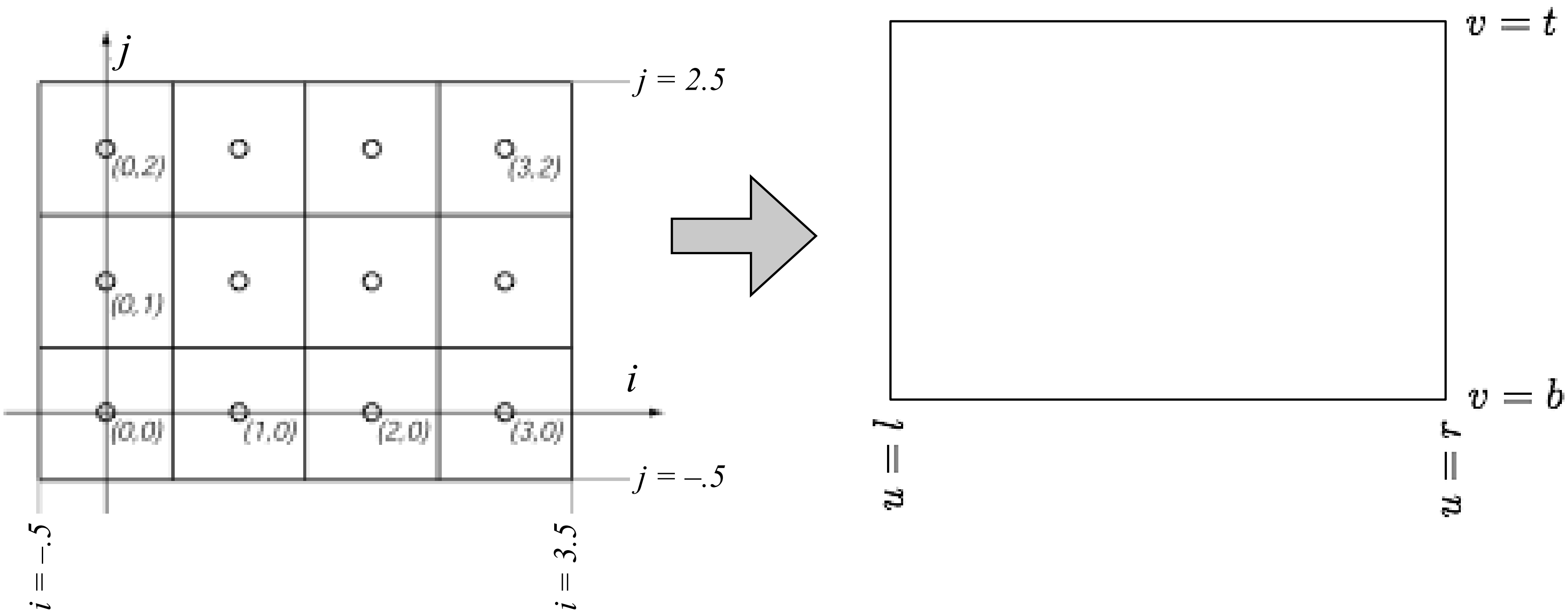
$$\mathbf{p} = \mathbf{e};\ \mathbf{d} = \mathbf{s} - \mathbf{e}$$

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



Pixel-to-image mapping

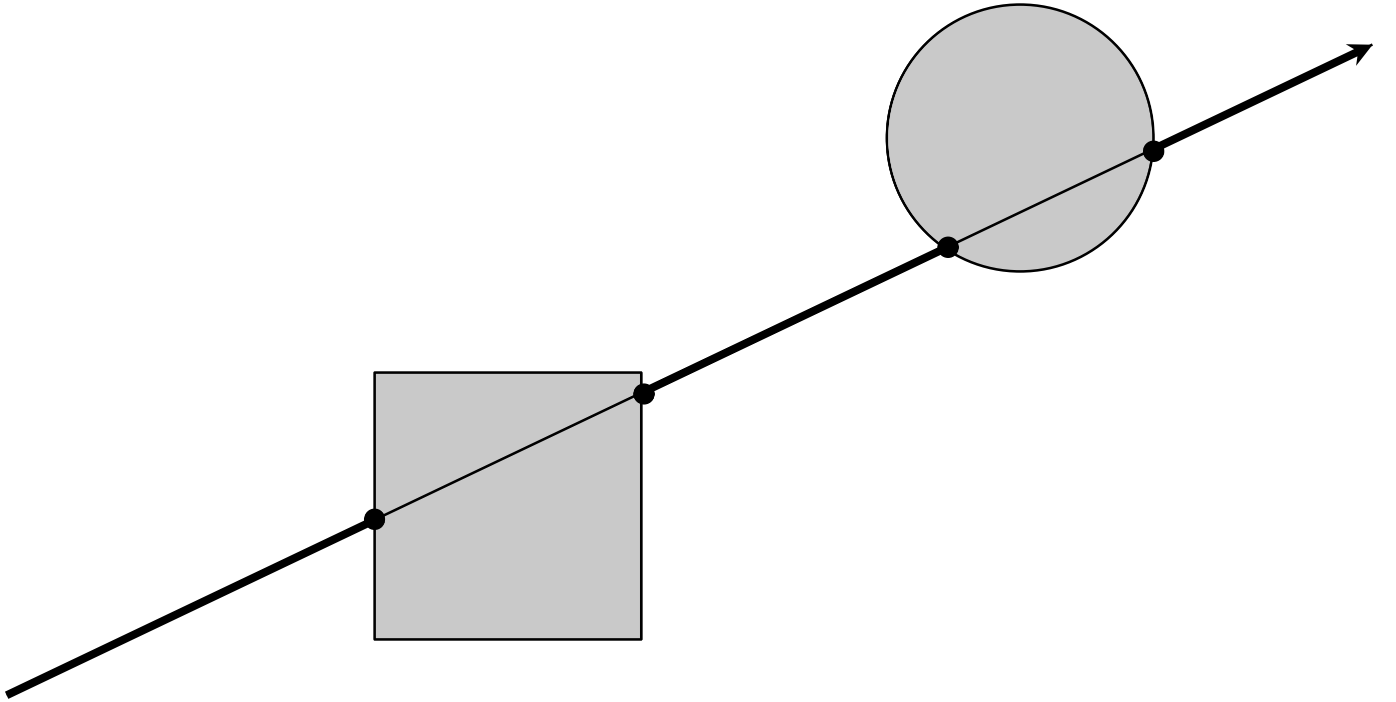
- One last detail: (*u*, *v*) coordinates of a pixel



$$u = l + (r - l)(i + 0.5)/n_x$$

$$v = b + (t - b)(j + 0.5)/n_y$$

Ray intersection

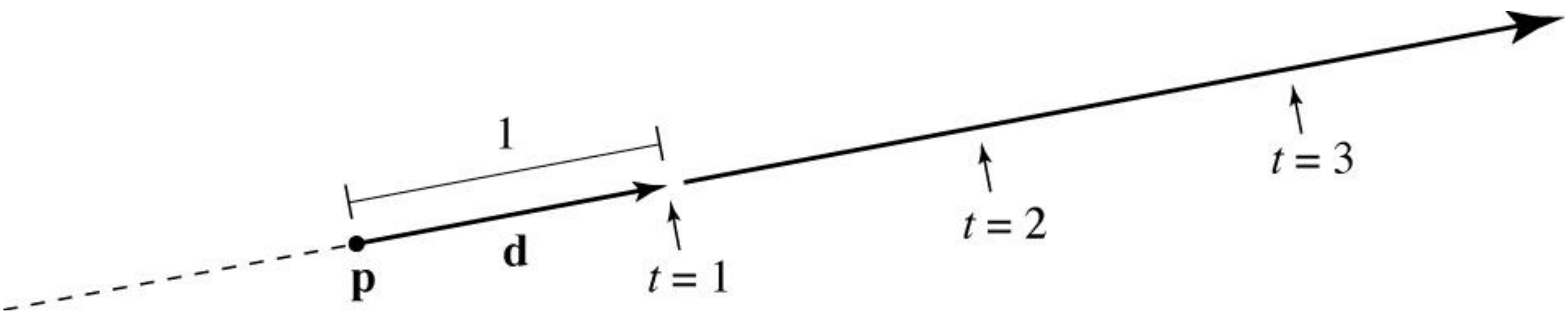


Ray: a half line

- 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 ($a > 0$)



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 Shirley 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:

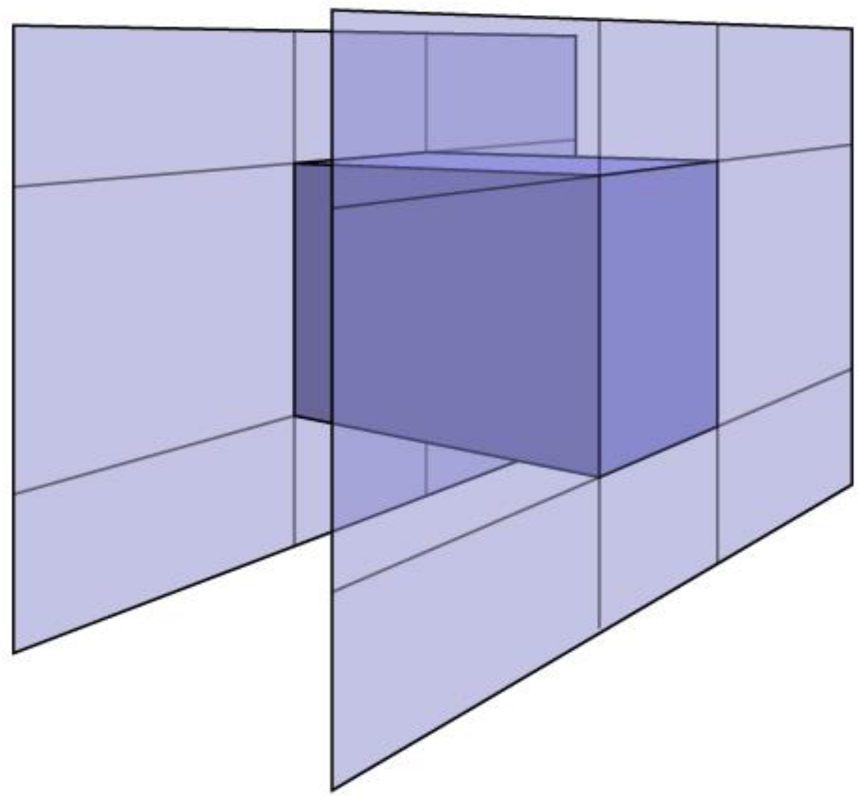
$$t = \frac{-\mathbf{d}\cdot\mathbf{p} \pm \sqrt{(\mathbf{d}\cdot\mathbf{p})^2 - (\mathbf{d}\cdot\mathbf{d})(\mathbf{p}\cdot\mathbf{p}-1)}}{\mathbf{d}\cdot\mathbf{d}}$$

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

  - 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-box intersection

- Could intersect with 6 faces individually
- Better way: box is the intersection of 3 slabs



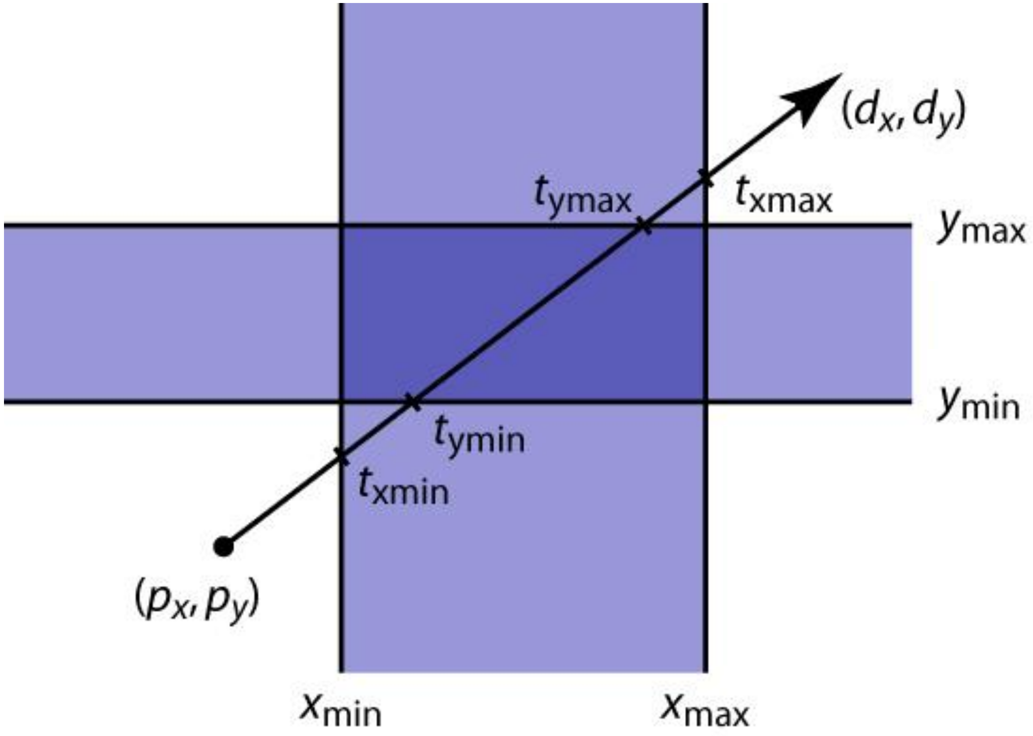
Note: min and max errors now fixed

Ray-slab intersection

- 2D example
- 3D is the same!

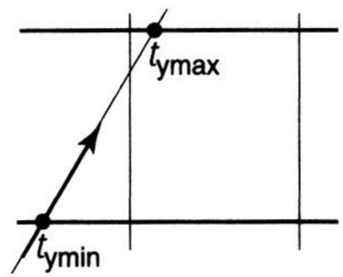
$$p_x + t_{x\mathrm{min}}\, d_x = x_{\mathrm{min}}$$
$$t_{x\mathrm{min}} = (x_{\mathrm{min}} - p_x)/d_x$$

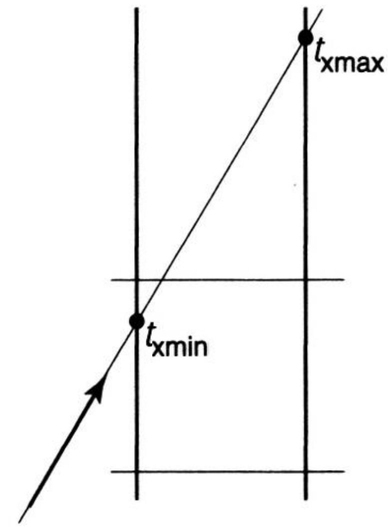
$$p_y + t_{y\mathrm{min}}\, d_y = y_{\mathrm{min}}$$
$$t_{y\mathrm{min}} = (y_{\mathrm{min}} - p_y)/d_y$$



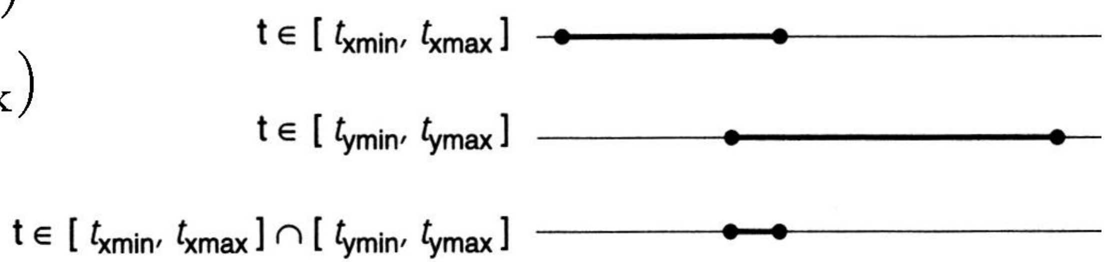
Intersecting intersections

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





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



Ray-triangle intersection

- Condition 1: point is on ray

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

- Condition 2: point is on plane

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

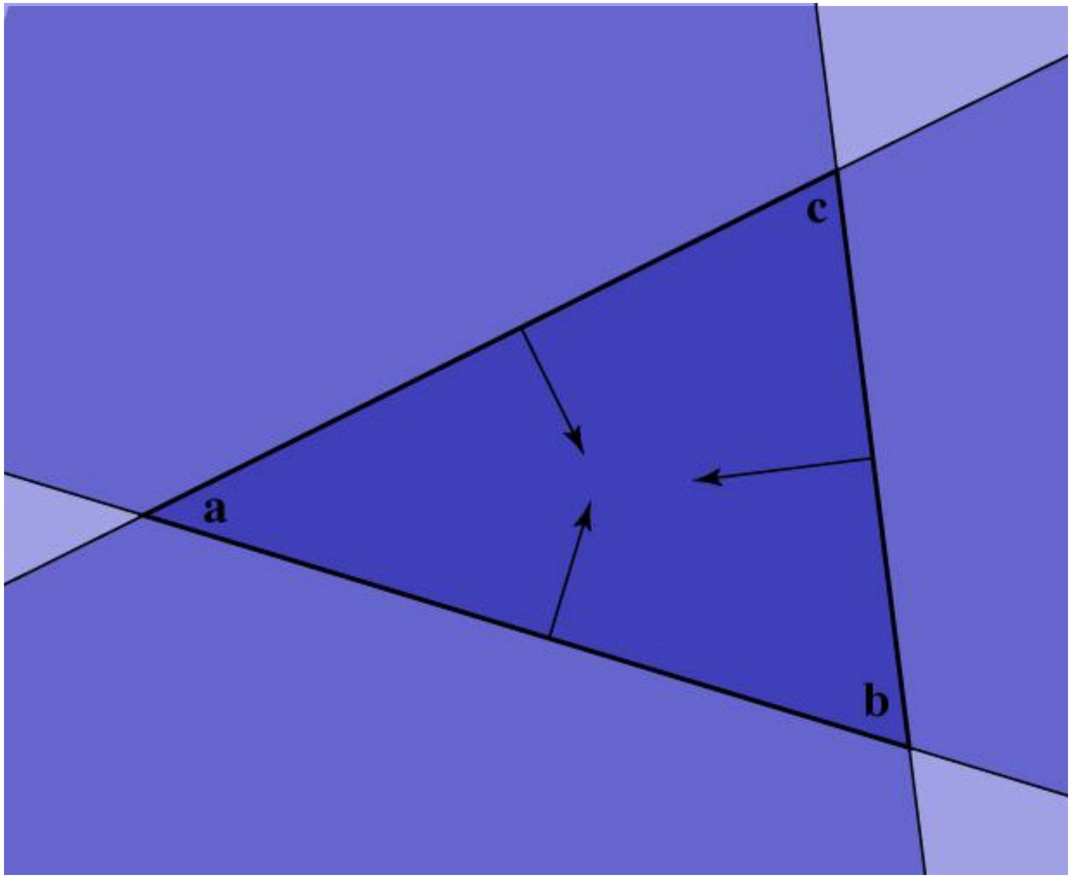
- Condition 3: point is on the inside of all three edges
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

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

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

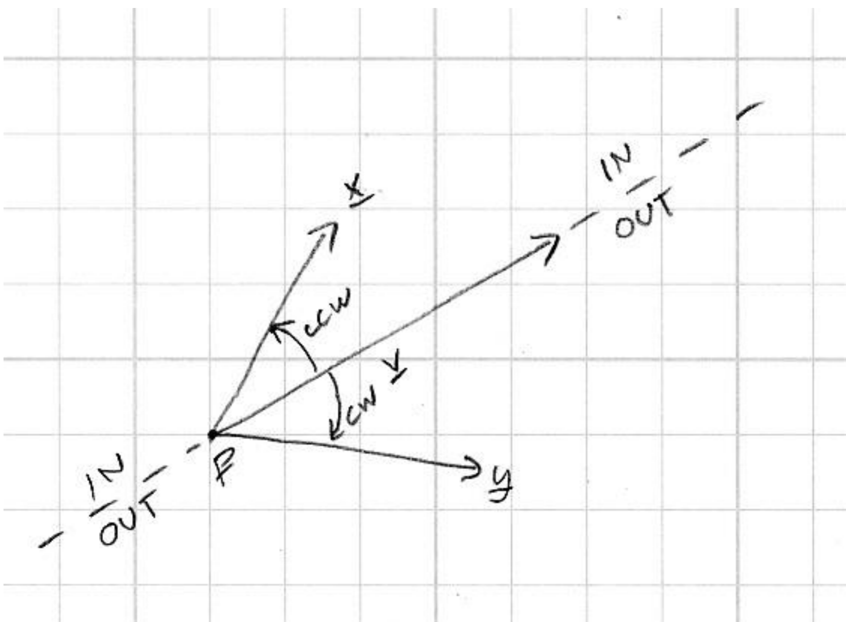
Ray-triangle intersection

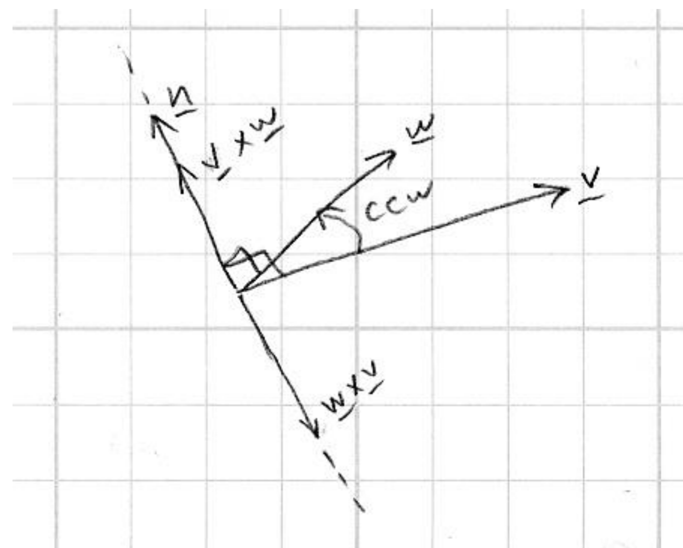
- In plane, triangle is the intersection of 3 half spaces



Inside-edge test

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



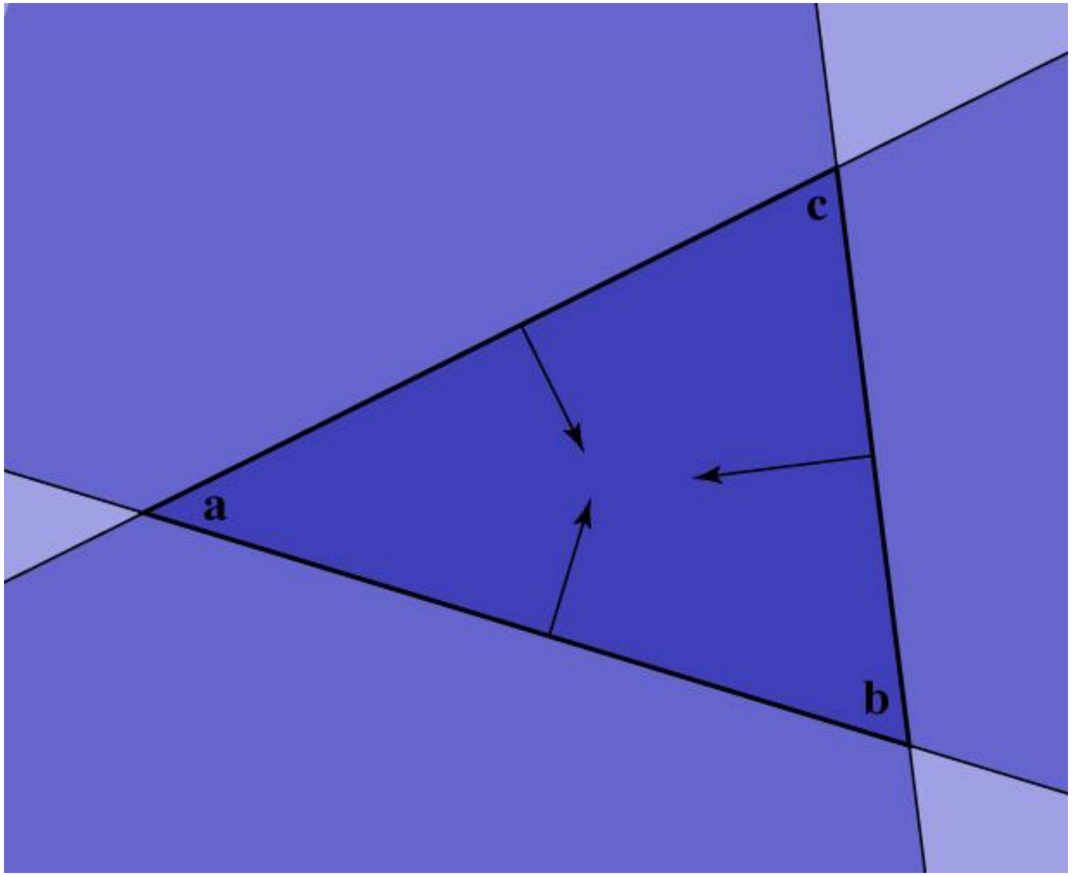


Ray-triangle intersection

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

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

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



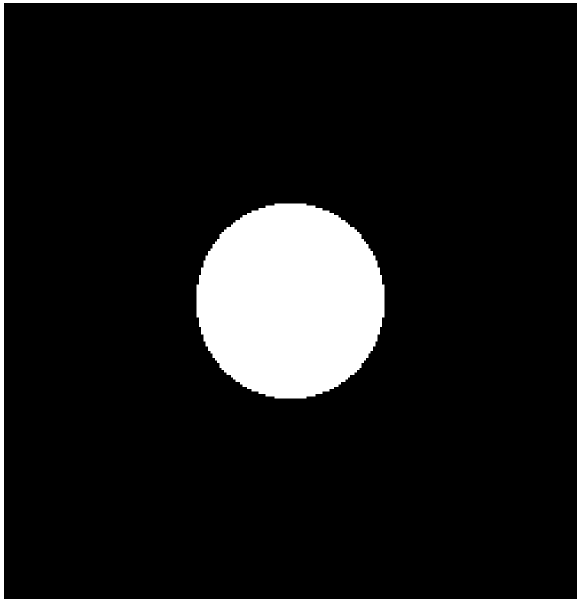
Ray-triangle intersection

- See book too…
  - See Section 4.4.2 for method based on linear systems and Cramer's rule
  - See also Section 2.7 with respect to barycentric coordinates

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);
    }
```

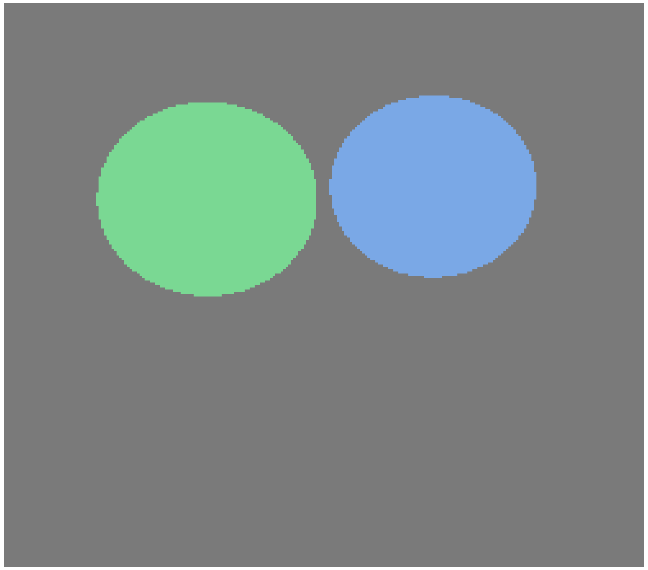


Intersection against many shapes

```
Group.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;
}
```

Image so far

- With eye ray generation and scene intersection



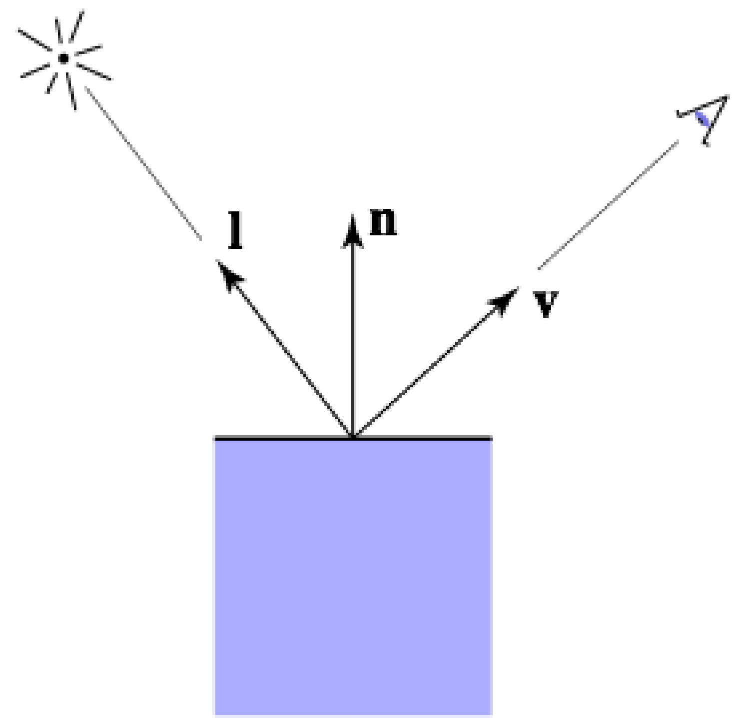
```
for 0 <= iy < ny
    for 0 <= ix < nx {
        ray = camera.getRay(ix, iy);
        c = scene.trace(ray, 0, +inf);
        image.set(ix, iy, c);
    }
```

…

```
Scene.trace(ray, tMin, tMax) {
    surface, t = surfs.intersect(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, …)
- Exact same equations
  as seen previously…

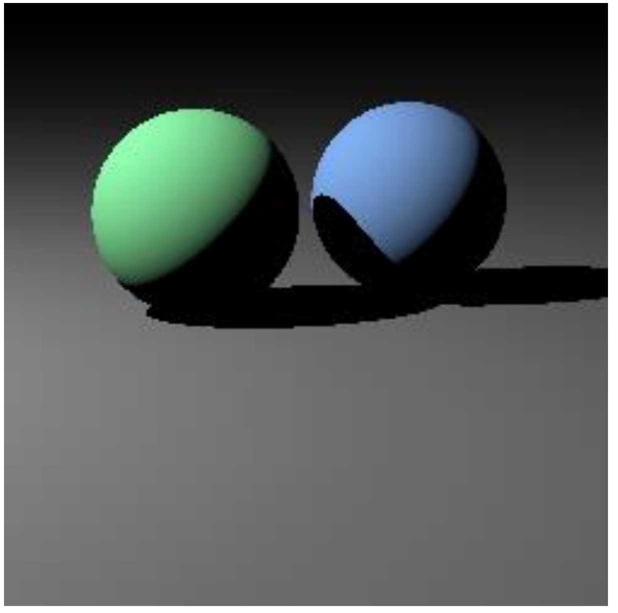


Shadows

- 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!

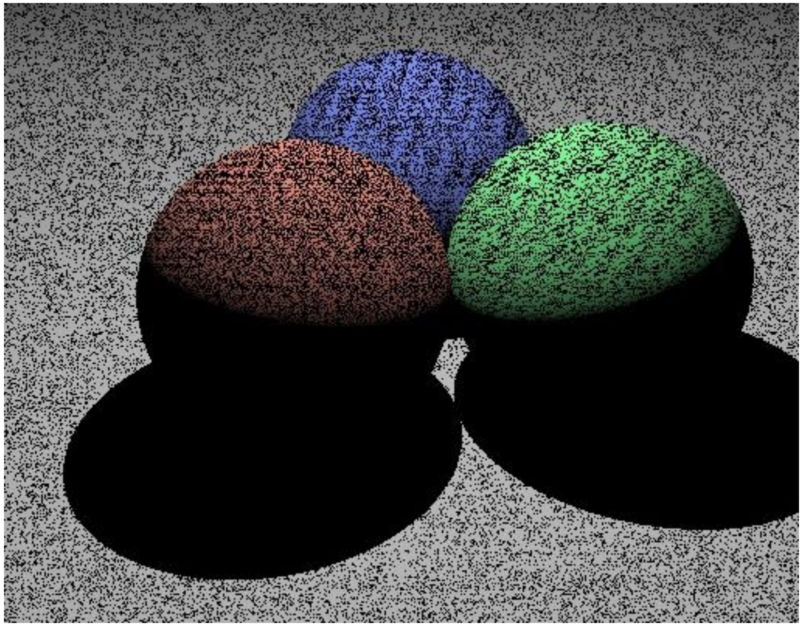
Image so far

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



Shadow rounding errors

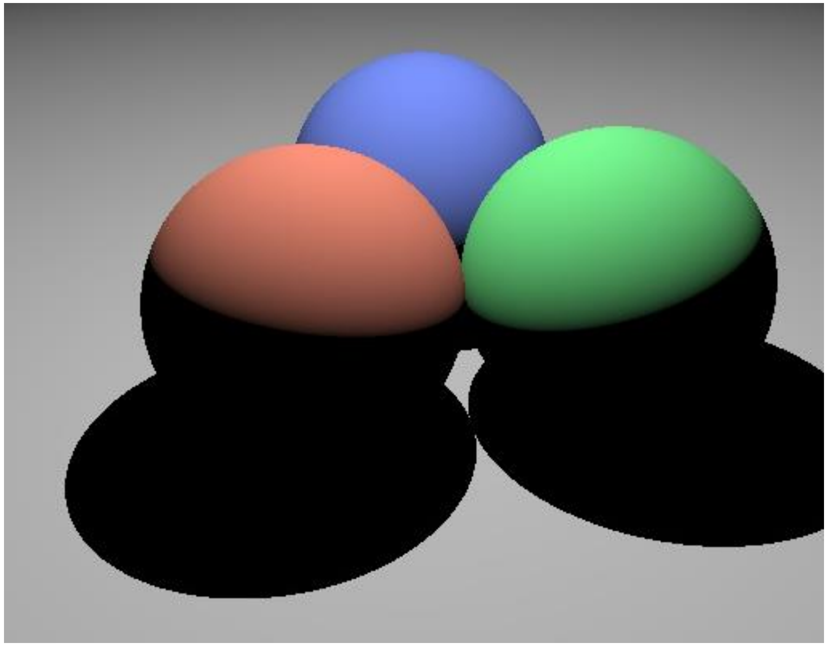
- Don’t fall victim to one of the classic blunders:



- What’s going on?
  - hint: at what *t* does the shadow ray intersect the surface you’re shading?

Shadow rounding errors

- Solution: shadow rays start a tiny distance from the surface



- Do this by moving the start point, or by limiting the *t* range

Mirror reflection

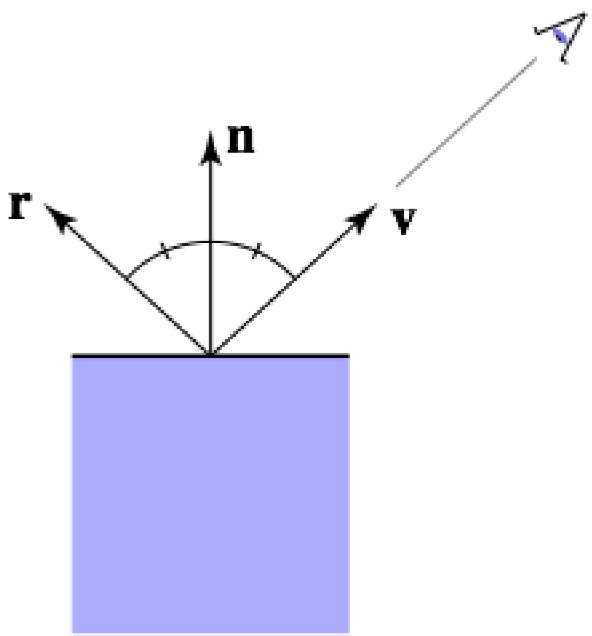
- Consider perfectly shiny surface
  - there isn't a highlight
  - instead there's a reflection of other objects
- Can render this using recursive ray tracing
  - to find out mirror reflection color, ask what color is seen from surface point in reflection direction
  - already computing reflection direction for Phong…
- "Glazed" material has mirror reflection and diffuse

$$L = L_a + L_d + L_m$$

  - where L_m is evaluated by tracing a new ray

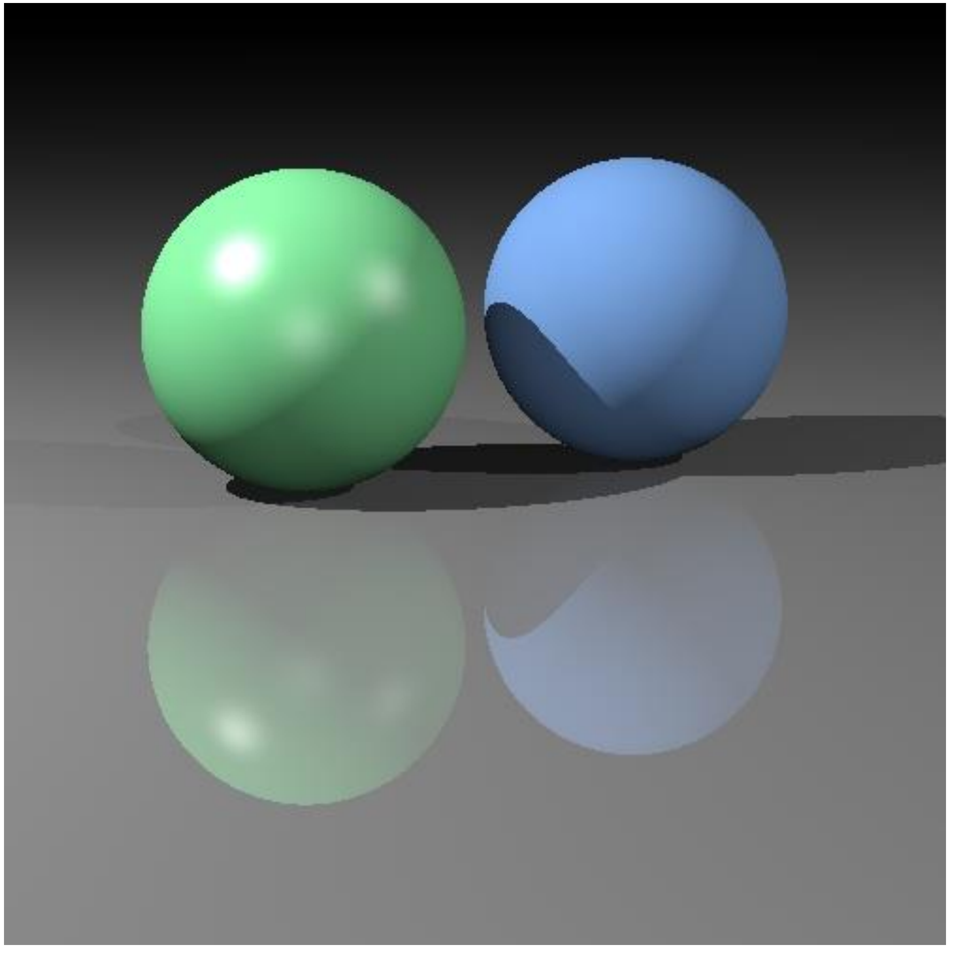
Mirror reflection

- Intensity depends on view direction
  - reflects incident light from mirror direction



$$\begin{aligned}\mathbf{r} &= \mathbf{v} + 2((\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v}) \\ &= 2(\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v}\end{aligned}$$

Diffuse + mirror reflection (glazed)



(glazed material on floor)

Ray tracer architecture 101

- You want a class called Ray
  - point and direction; evaluate(t)
  - possible: tMin, tMax
- Some things can be intersected with rays
  - individual surfaces
  - groups of surfaces (acceleration goes here)
  - the whole scene
  - make these all subclasses of Surface
  - limit the range of valid *t* values (*e.g.* shadow rays)
- Once you have the visible intersection, compute the color
  - may want to separate shading code from geometry
  - separate class: Material (each Surface holds a reference to one)
  - its job is to compute the color

Architectural practicalities

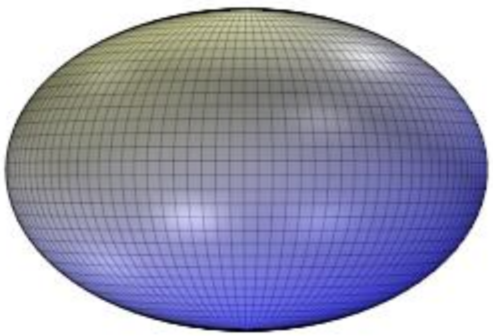
- Return values
  - surface intersection tends to want to return multiple values
    - *t*, surface or shader, normal vector, maybe surface point
  - typical solution: an *intersection record*
    - a class with fields for all these things
    - keep track of the intersection record for the closest intersection
- Efficiency
  - in Java the (or, a) key to being fast is to minimize creation of objects
  - what objects are created for every ray? try to find a place for them where you can reuse them.
  - Shadow rays can be cheaper (any intersection will do, don't need closest)
  - but: "First Get it Right, Then Make it Fast"

Debugging strategies

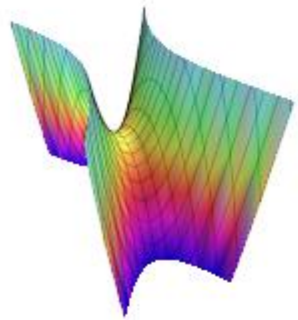
- Test with small images
- Set breakpoints!!!
  - E.g., conditional on a specific pixel
- Make sure your rays are in the correct direction
  - For example, is the ray for the center of the image what you expect it to be?
- Watch out for other common mistakes…

Quadrics

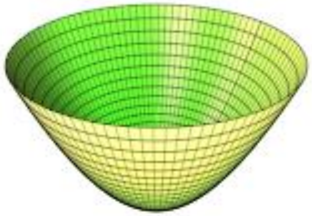
http://en.wikipedia.org/wiki/Quadric



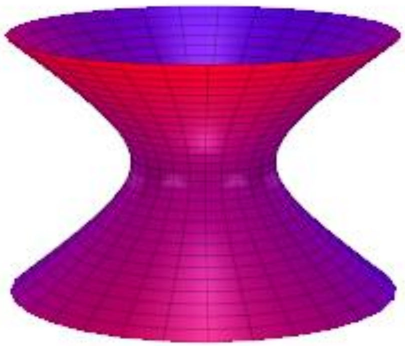
Ellipsoid



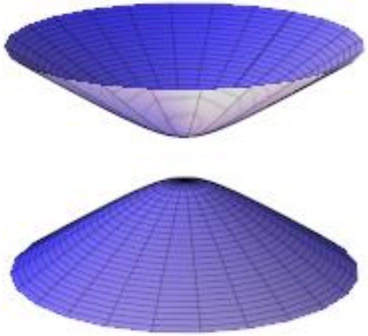
Hyperbolic paraboloid



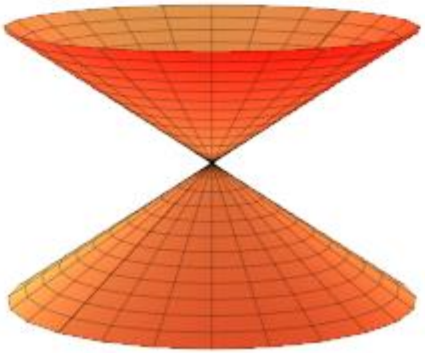
Elliptic paraboloid



Hyperboloid of one sheet



Hyperboloid of two sheets



Cone

Quadrics

- In non-homogeneous coordinates we can write

$$[x\ y\ z]A\begin{bmatrix}x\\y\\z\end{bmatrix} + b^T\begin{bmatrix}x\\y\\z\end{bmatrix} + c = 0 \qquad A \in R^{3\times 3} \quad b \in R^{3\times 3} \quad c \in R$$

- In homogeneous coordinates, use $Q \in R^{4\times 4}$ matrix

$$Q = \begin{bmatrix} A & \frac{1}{2}b \\ \frac{1}{2}b^T & c \end{bmatrix} \qquad \boldsymbol{x}^T Q \boldsymbol{x} = 0 \qquad \boldsymbol{x} = \begin{bmatrix}x\\y\\z\\1\end{bmatrix} = \mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Solution is same as ray sphere intersection.
  - Replace *x* with ray equation,
  - Expand, solve for *t*
  - ***Given intersection point x, what is the normal?***