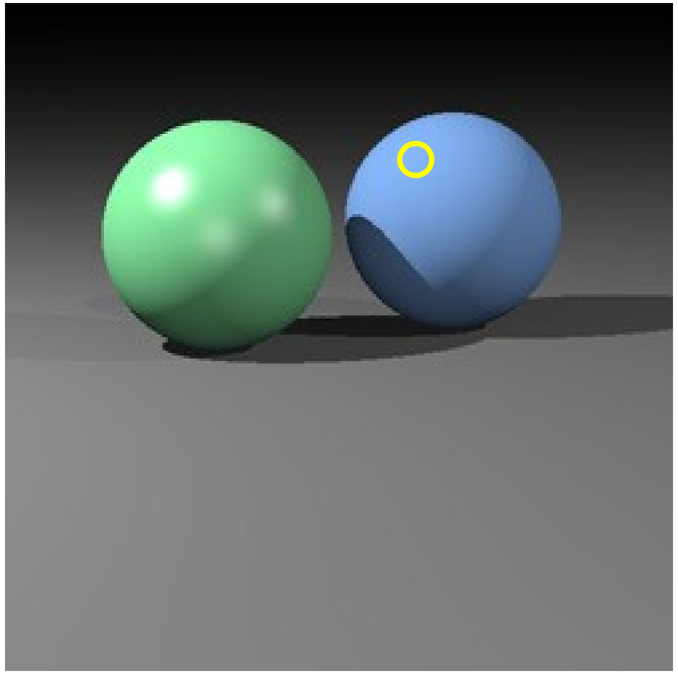
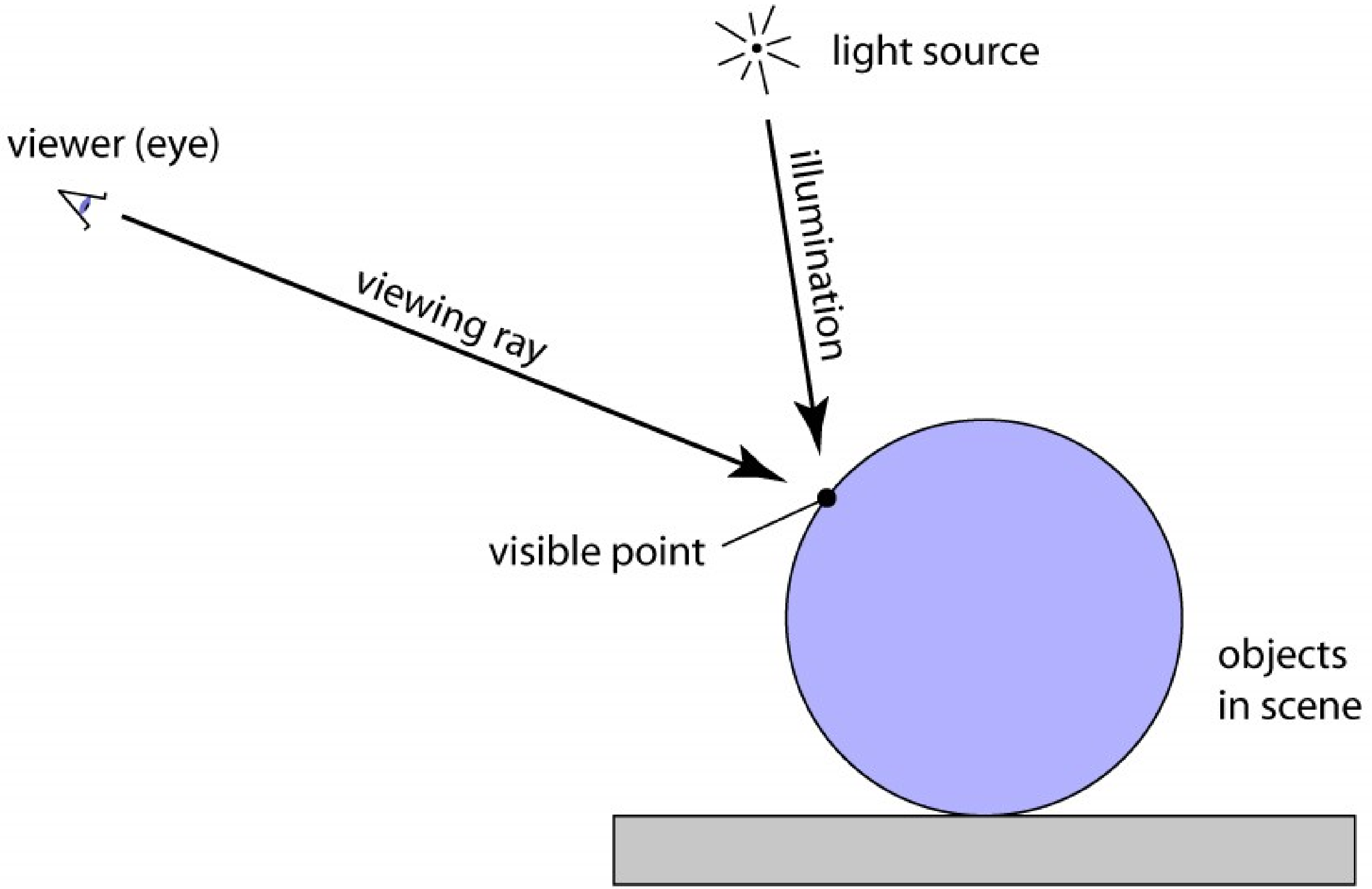
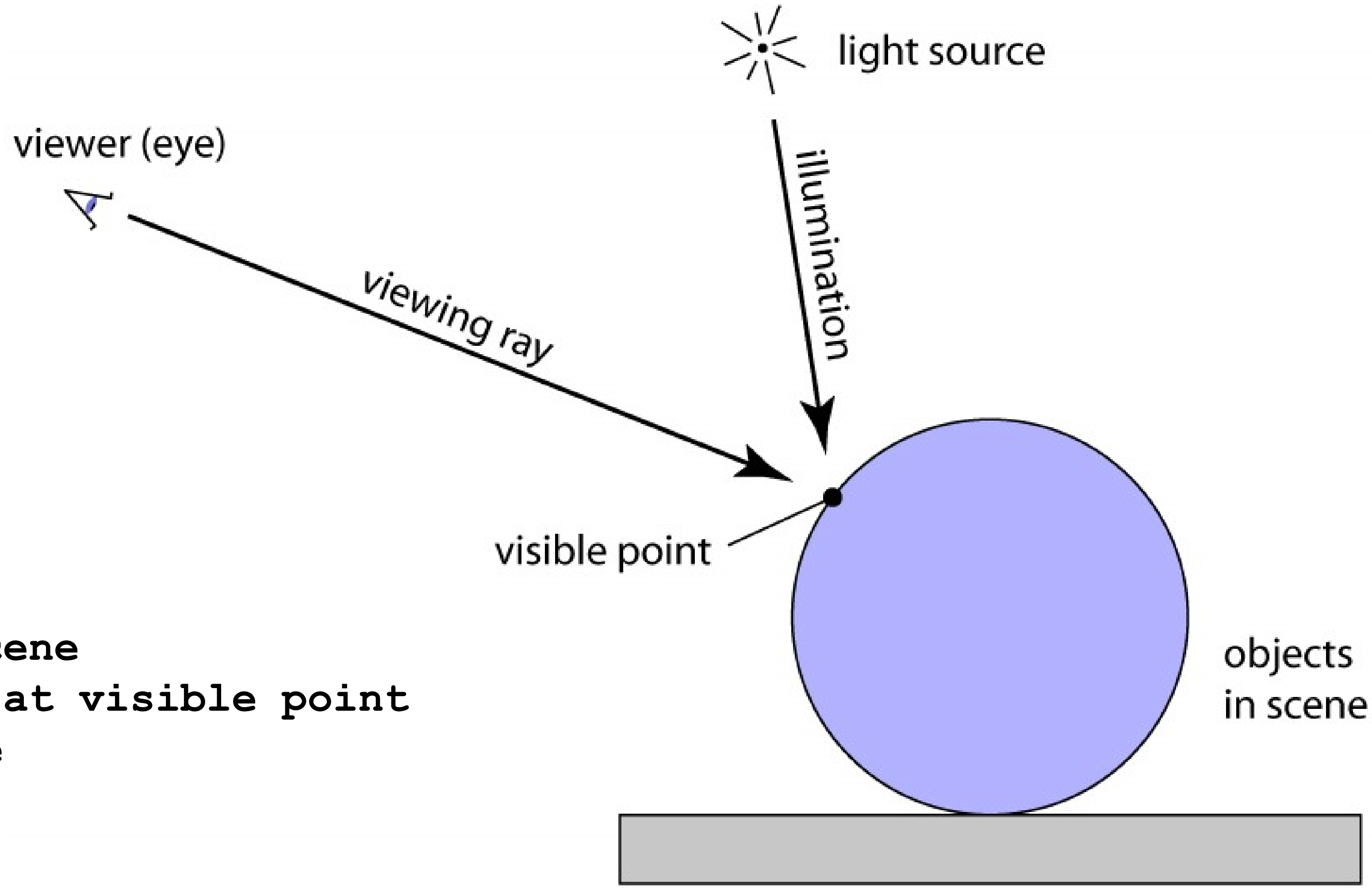
Ray Tracing

Ray tracing idea





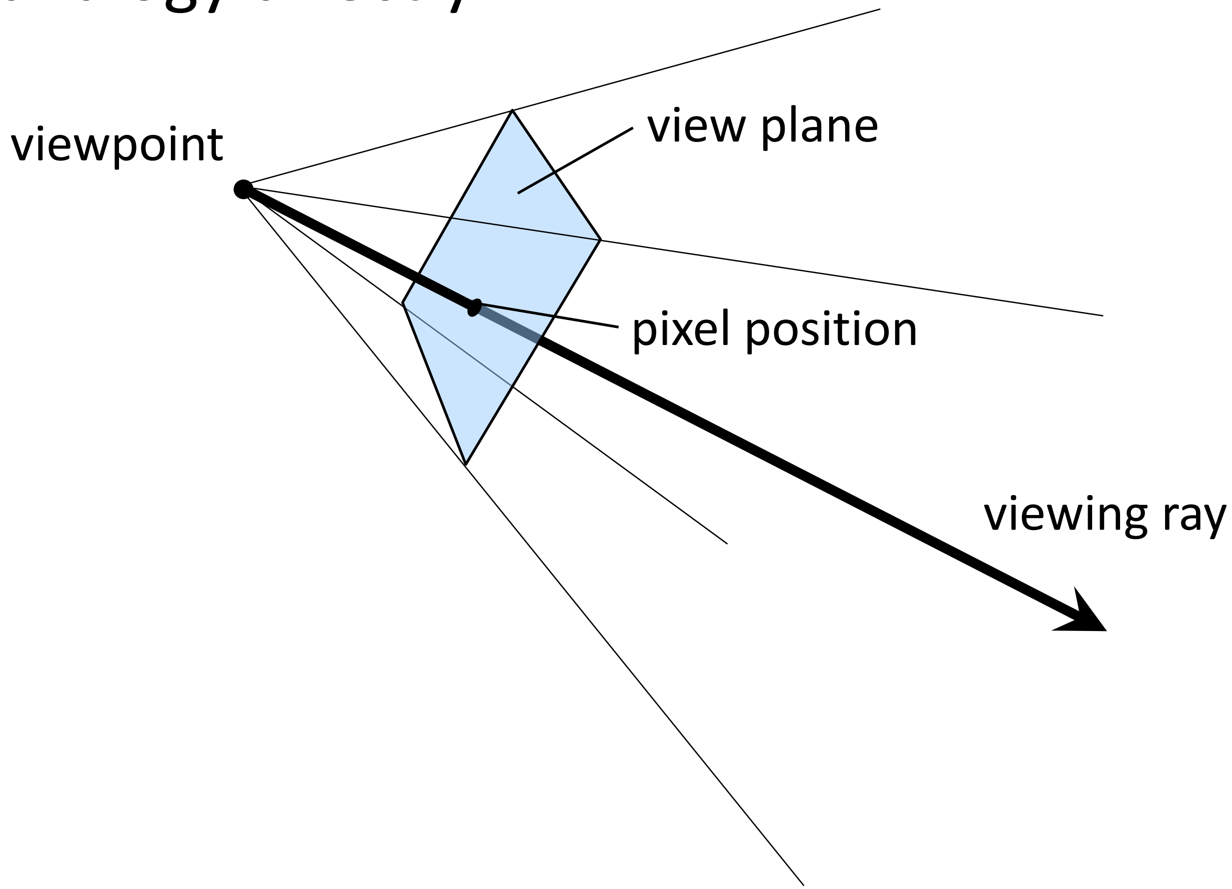
Ray tracing algorithm




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

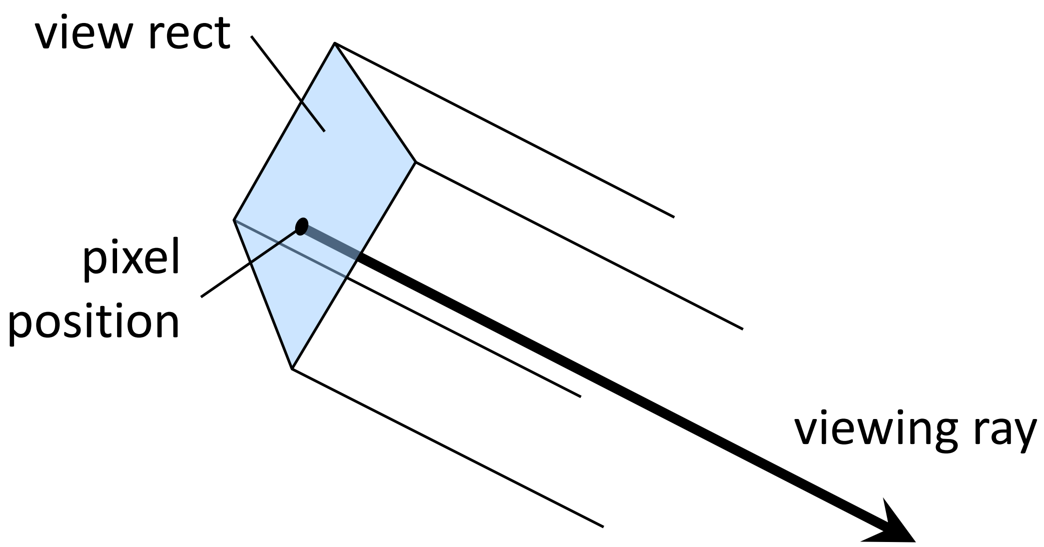
Generating eye rays

- Use window analogy directly

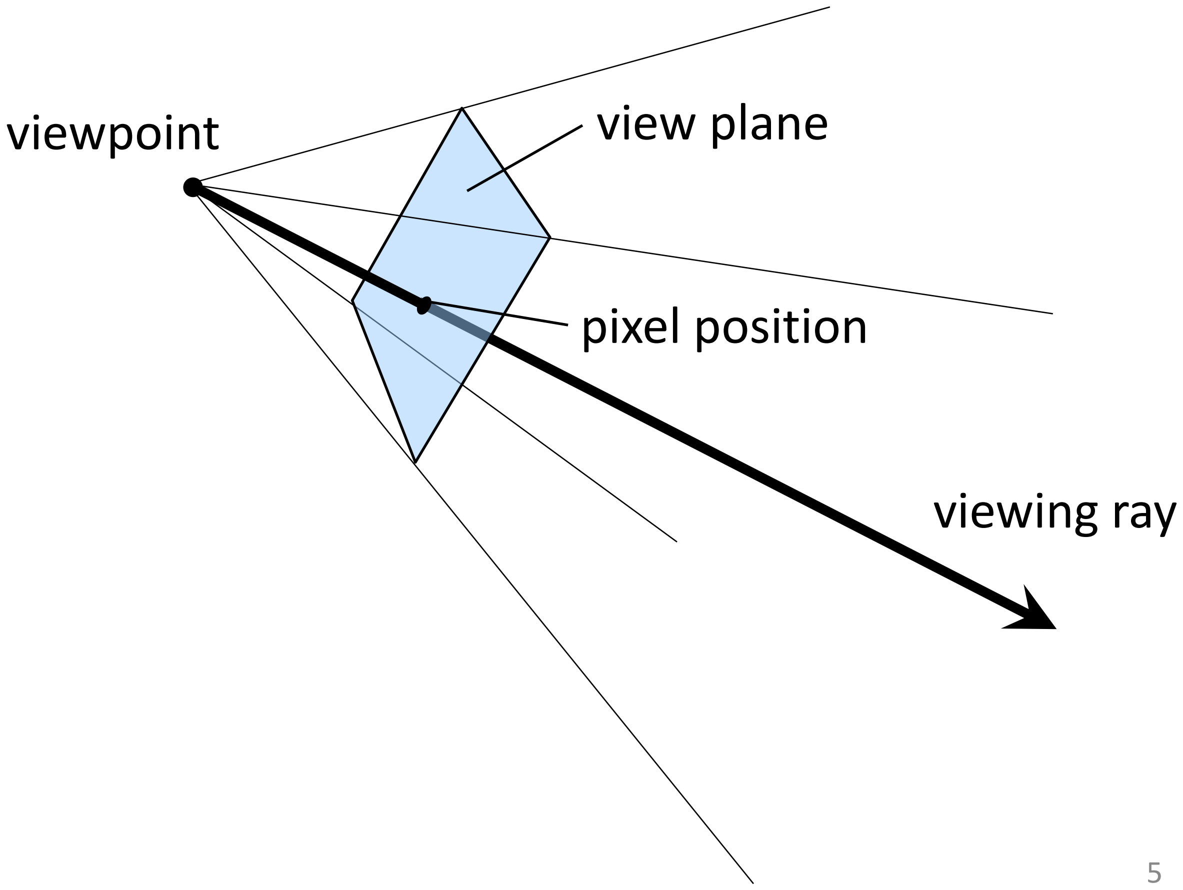


Generating eye rays

ORTHOGRAPHIC

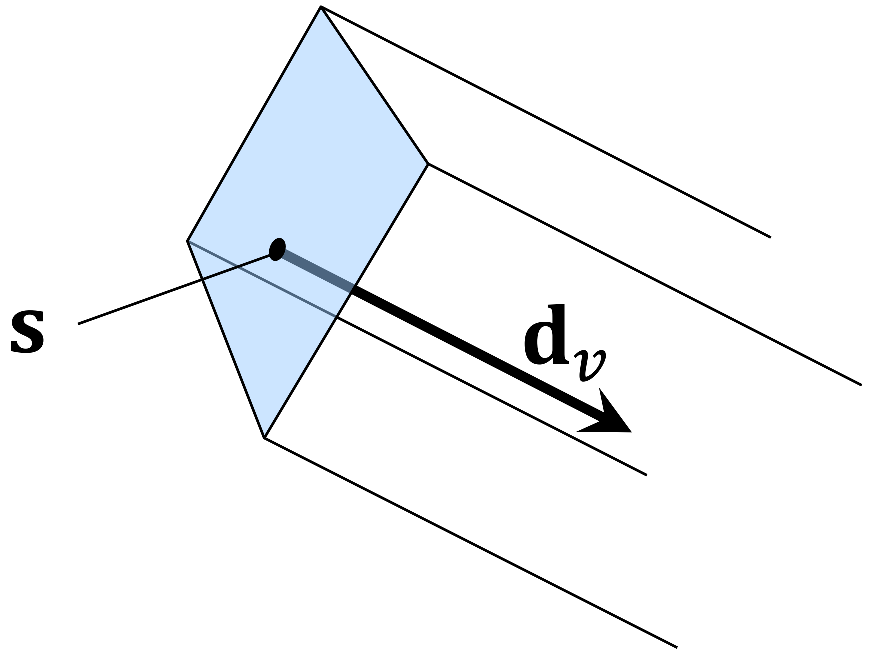


PERSPECTIVE



Generating eye rays—orthographic

- Just need to compute the view plane point $\boldsymbol{s}$



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

But where exactly is the view rectangle?

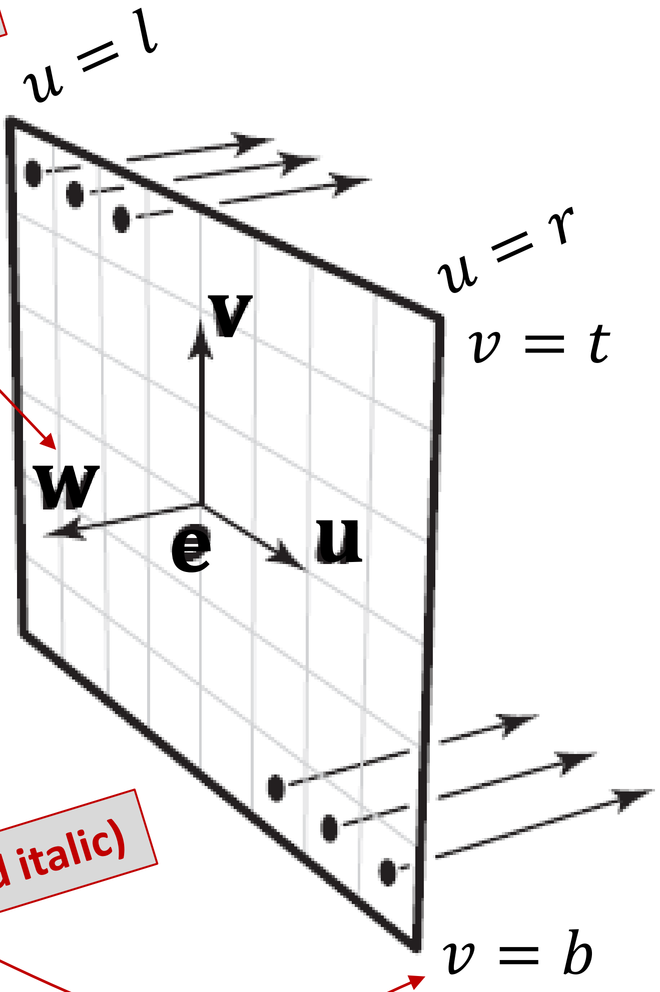
Generating eye rays—orthographic

3D point (bold non-italic)

3D vector (bold non-italic)

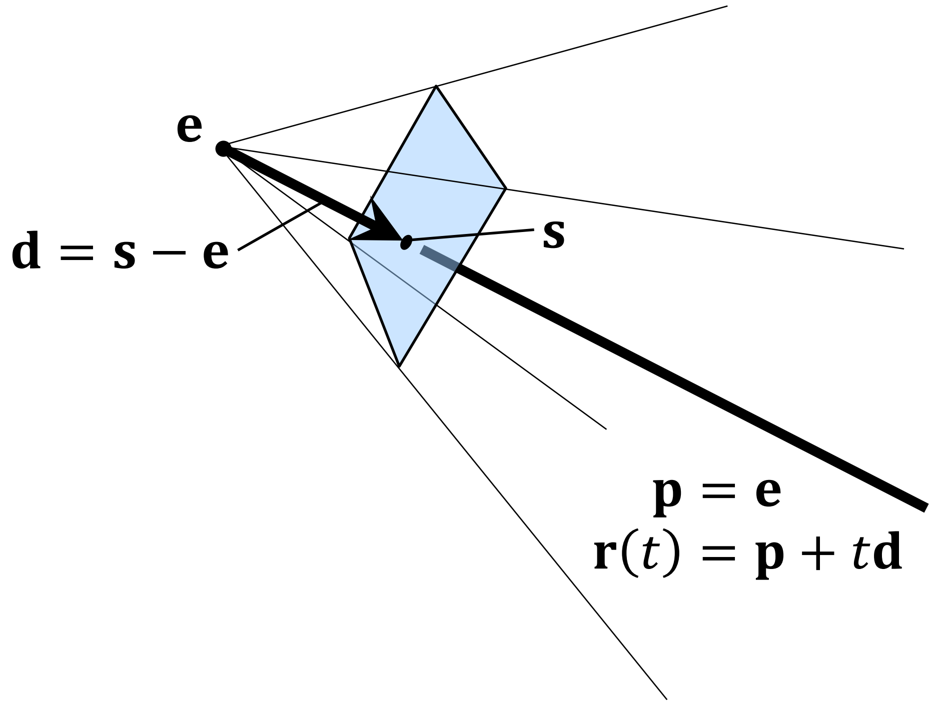
$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v}$$
$$\mathbf{p} = \mathbf{s}$$
$$\mathbf{d} = -\mathbf{w}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

Scalar (non-bold italic)



Generating eye rays—perspective

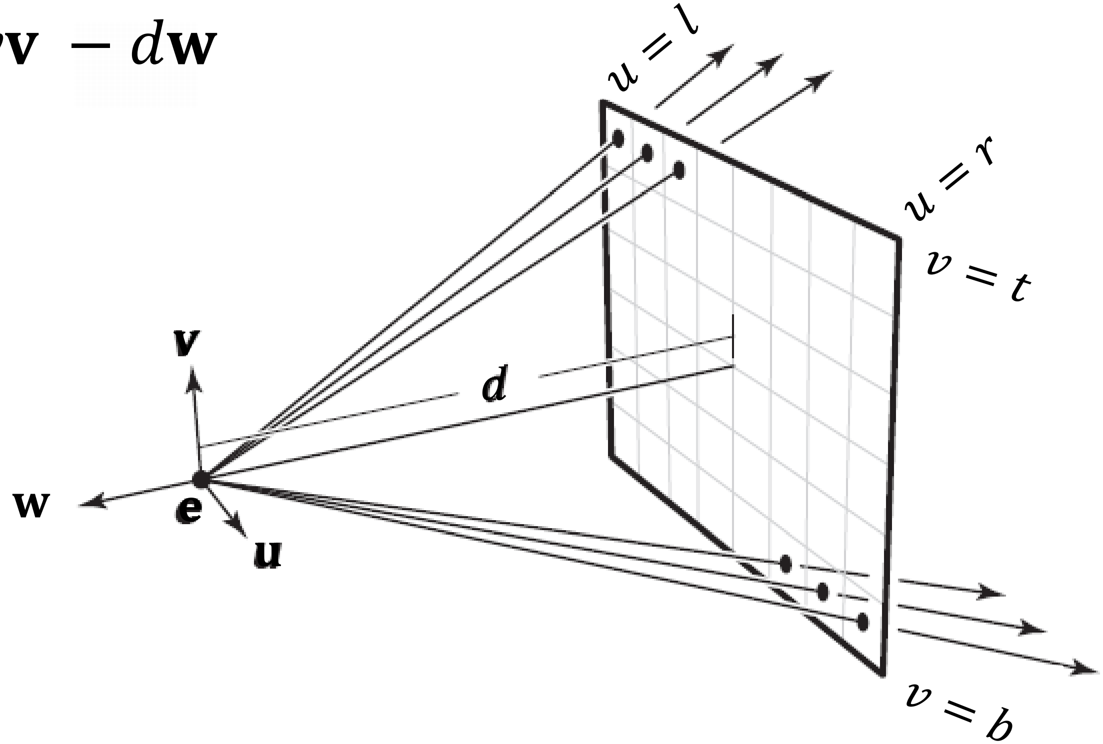
- View rectangle needs to be away from viewpoint
- Distance is important: “focal length” of camera
  - still use camera frame but position view rectangle away from viewpoint
  - ray origin always $\mathbf{e}$
  - ray direction now controlled by $\boldsymbol{s}$



Generating eye rays—perspective

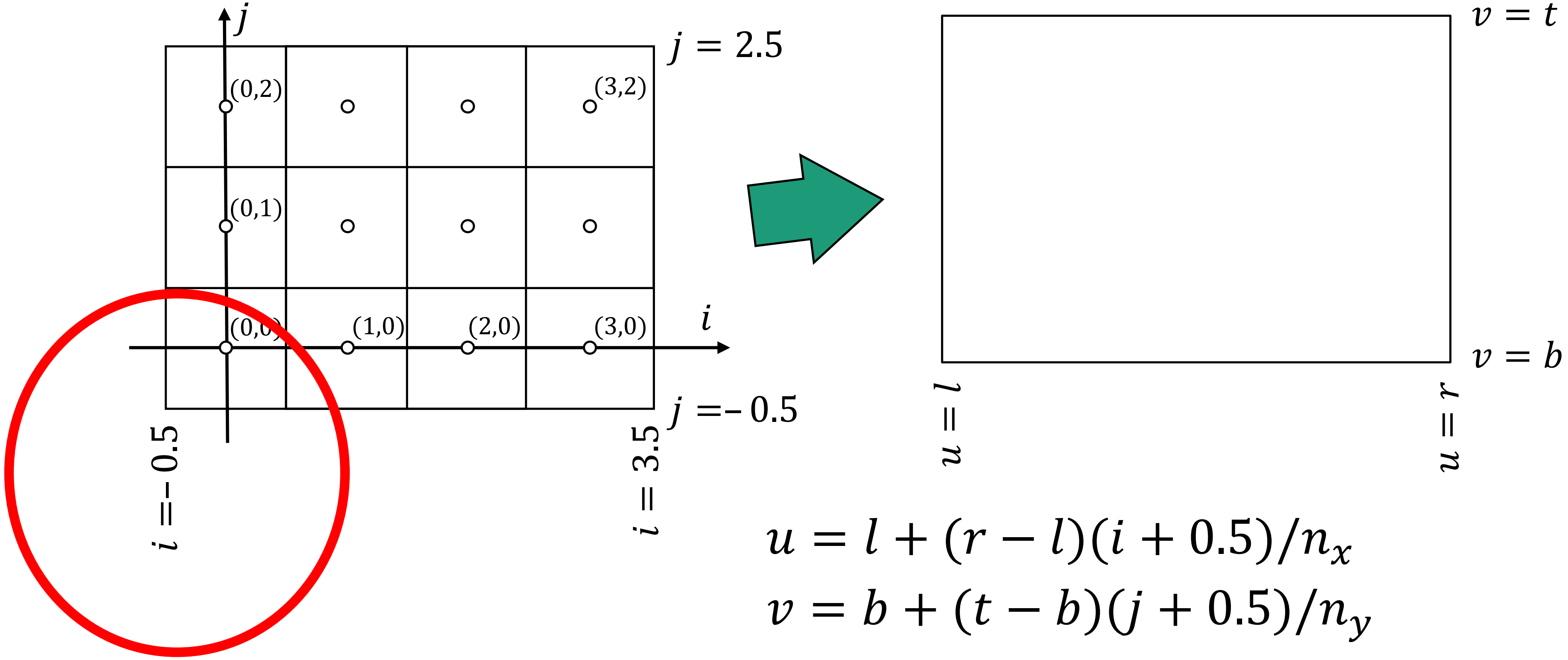
- Compute $\mathbf{s}$ in the same way; just subtract $d\boldsymbol{w}$
  - coordinates of $\mathbf{s}$ are $(u, v, -d)$

$$\mathbf{s} = \mathbf{e} + u\mathbf{u} + v\mathbf{v} - d\mathbf{w}$$
$$\mathbf{p} = \mathbf{e}$$
$$\mathbf{d} = \mathbf{s} - \mathbf{e}$$
$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$



Pixel-to-image mapping

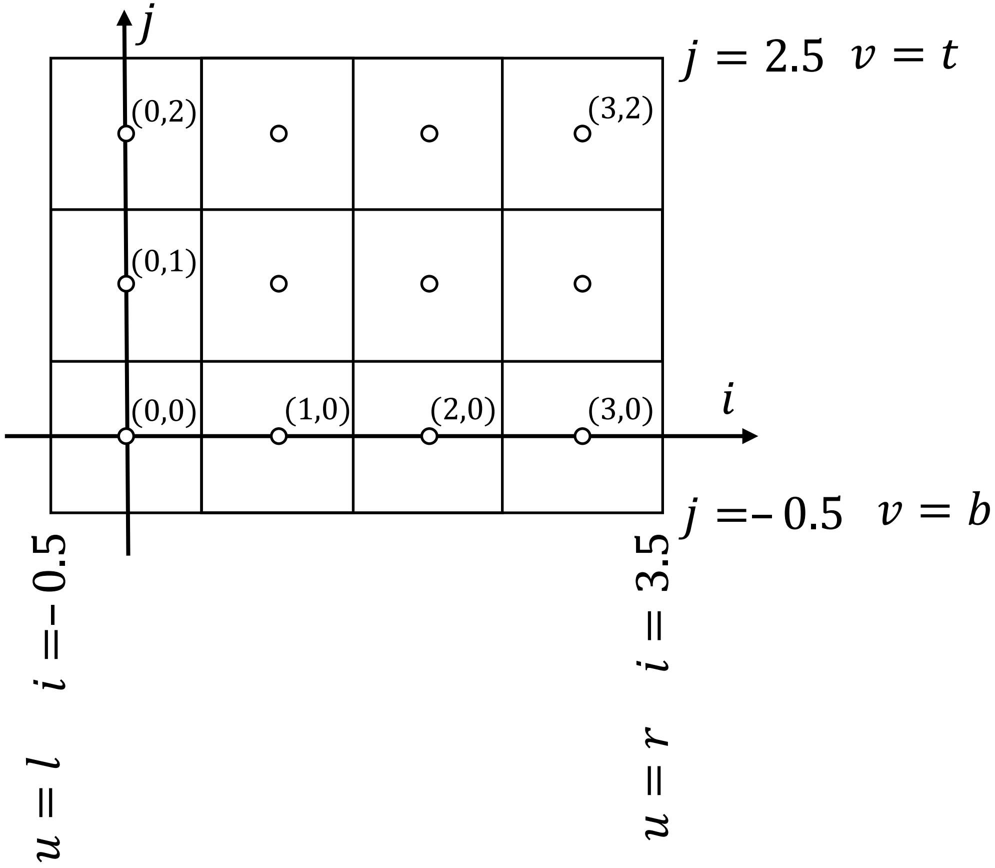
- One last detail: (u, v) coordinates of a pixel



$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

Pixel-to-image mapping

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

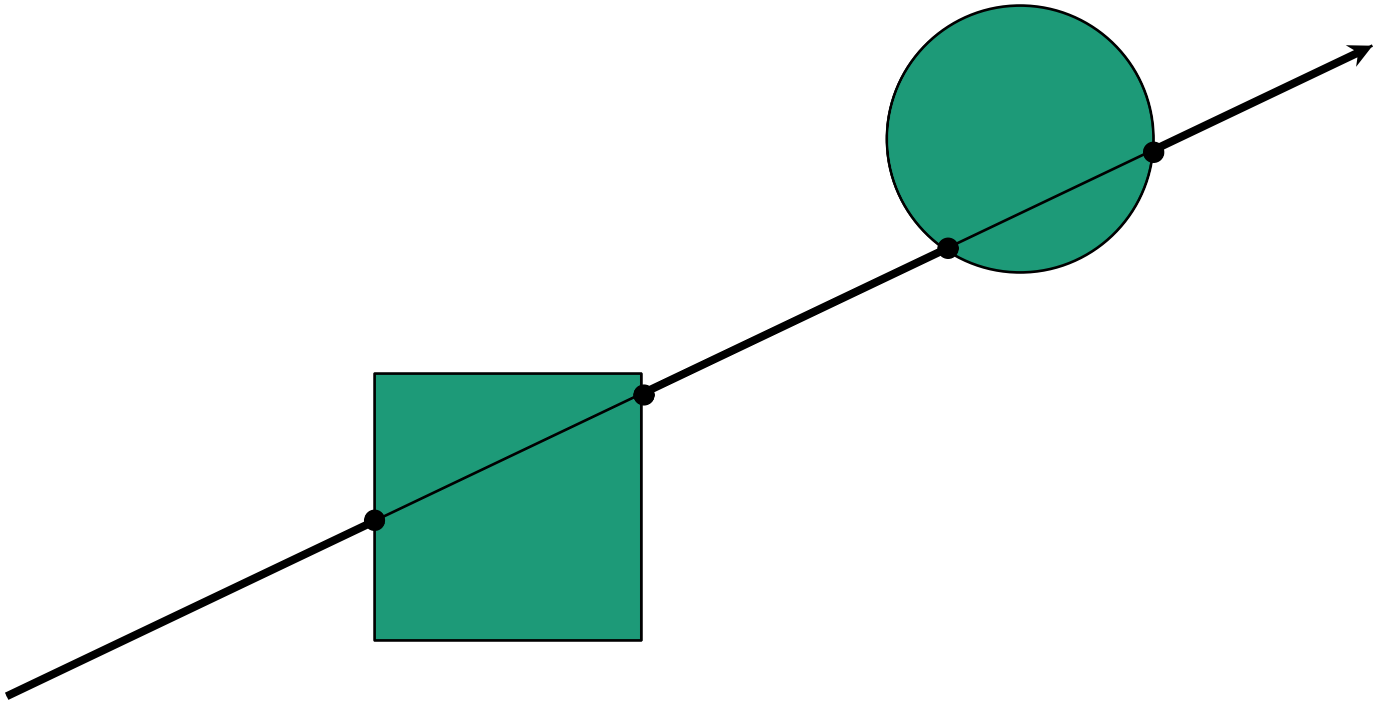


$$u = l + (r - l)(i + 0.5)/n_x$$
$$v = b + (t - b)(j + 0.5)/n_y$$

Generating eye rays – camera specification

- Given
  - Field of view in y direction,
  - eye point, look at, up direction
  - Image resolution $n_x \times n_y$
- How do you generate eye rays?
  - Do you need to compute a viewing transformation matrix?
  - Do you need to invert a matrix?
  - What is the aspect ratio of the image?
  - What is the left right top and bottom?
  - If the focal length is unspecified, what should d be?

Ray intersection

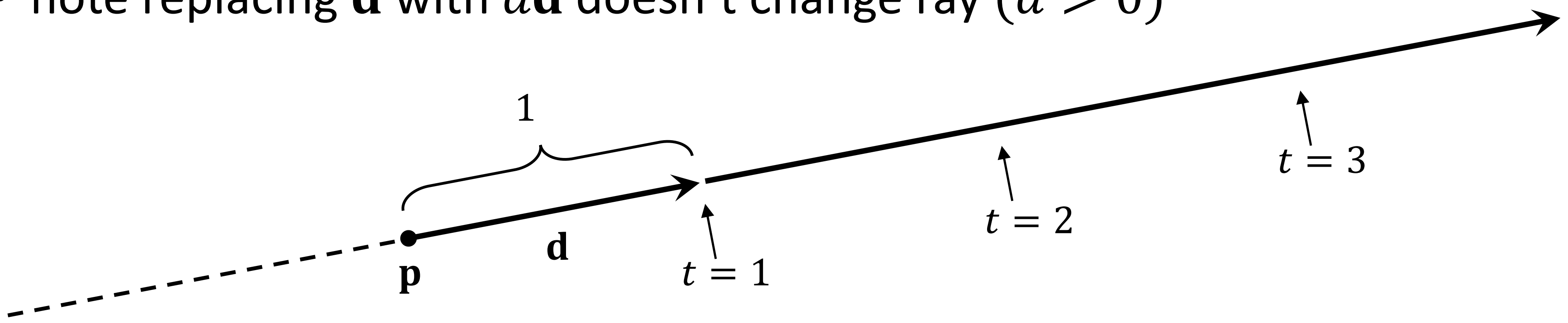


Ray: a half line

- 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 $a\mathbf{d}$ 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

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

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

- Substitute:

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

  - this is a quadratic equation in t

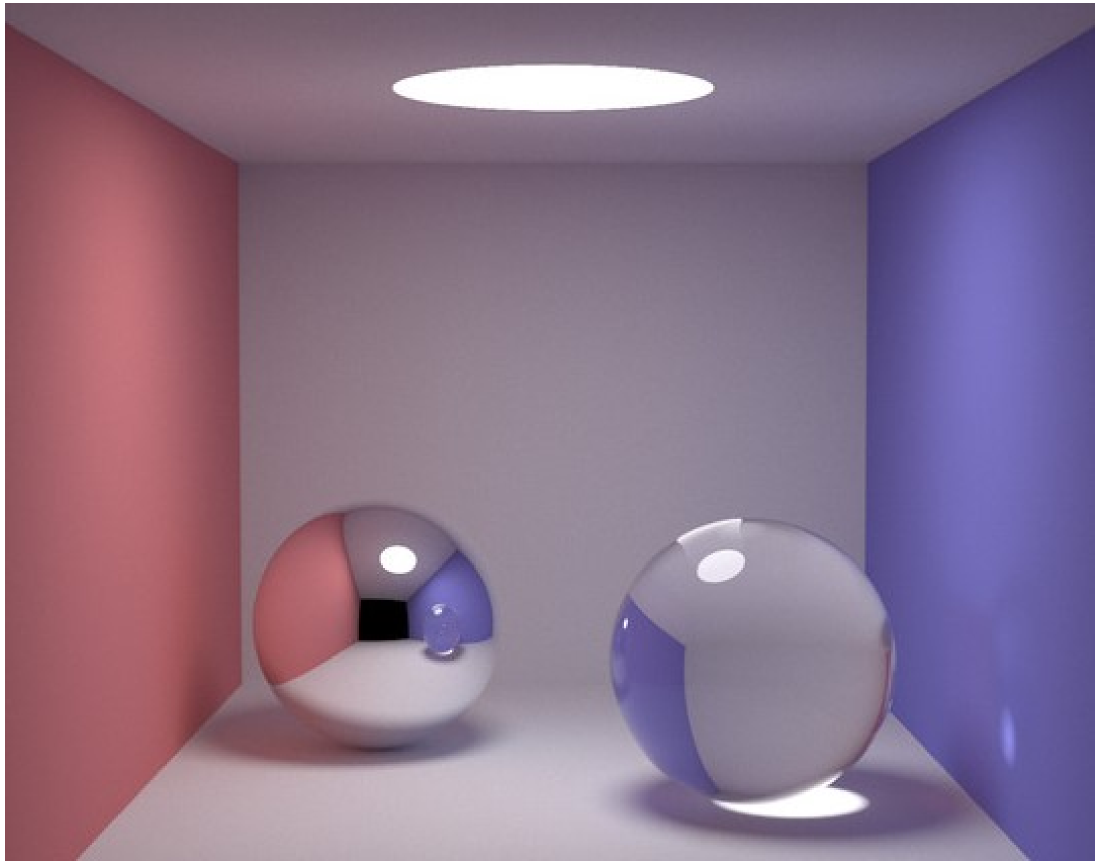
Ray-sphere intersection: algebraic

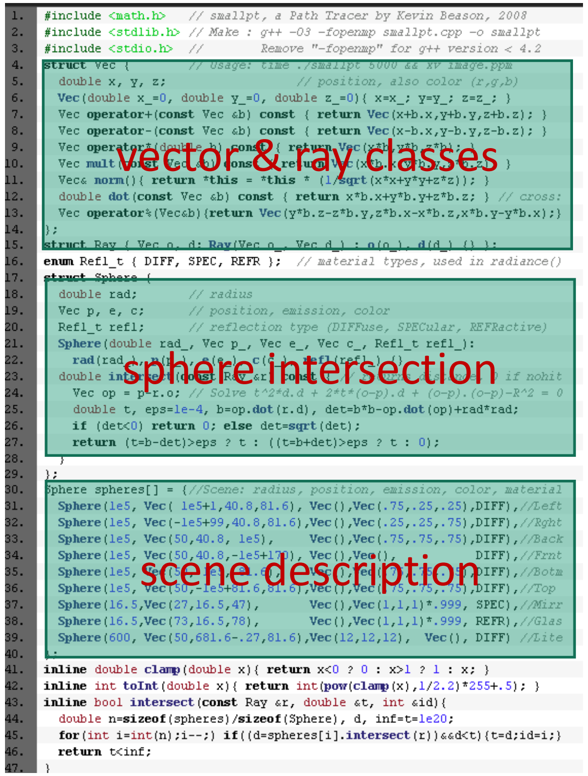
- Solution for t by quadratic formula:

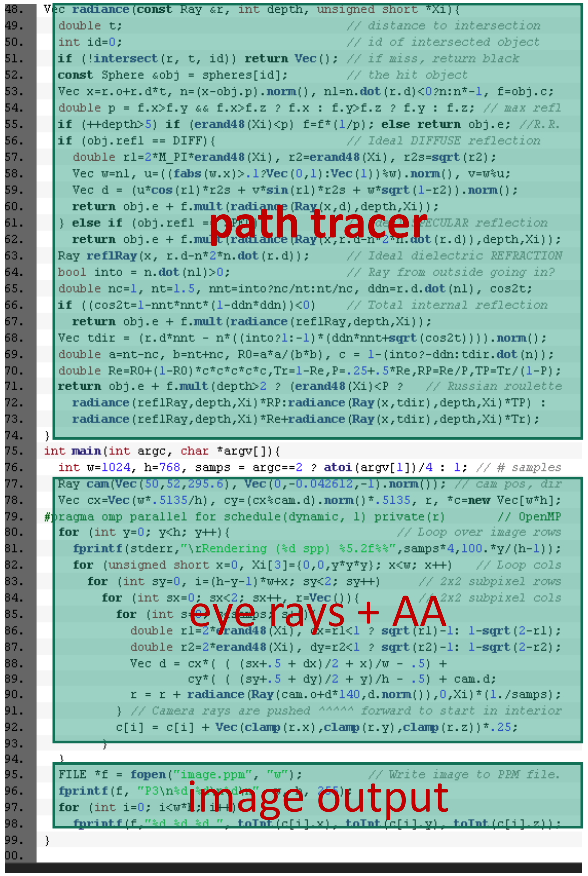
$$t = \frac{-\mathbf{d} \cdot \mathbf{p} \pm \sqrt{(\mathbf{d} \cdot \mathbf{p})^2 - (\mathbf{d} \cdot \mathbf{d})(\mathbf{p} \cdot \mathbf{p} - 1)}}{\mathbf{d} \cdot \mathbf{d}}$$

What can you do with Spheres? (ASIDE)

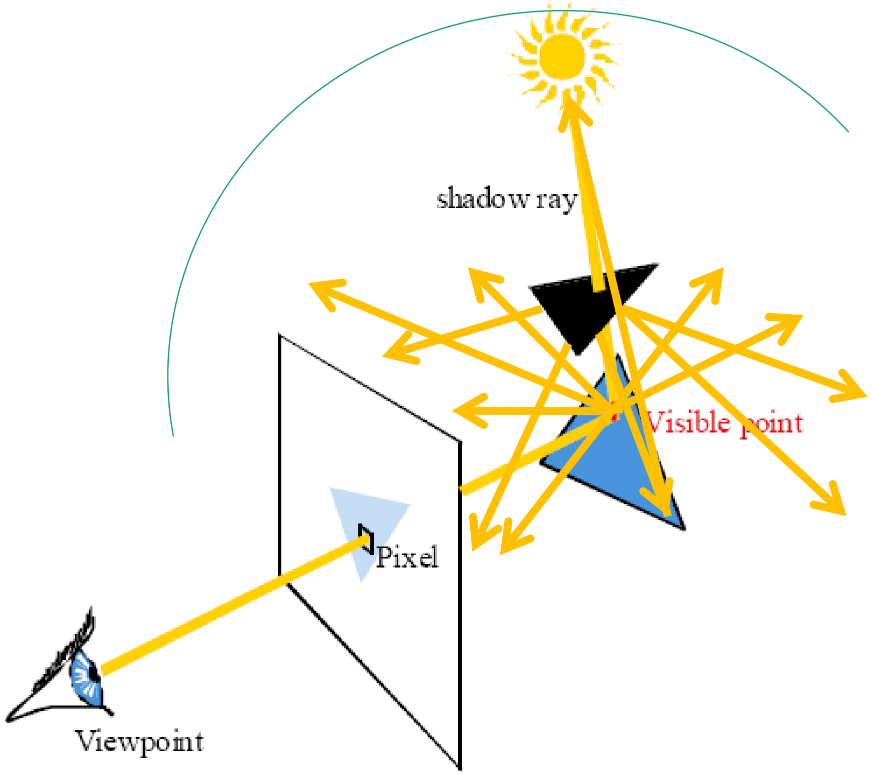
Physically based rendering with 100 lines of code
http://www.kevinbeason.com/smallpt/







Global Illumination (ASIDE)

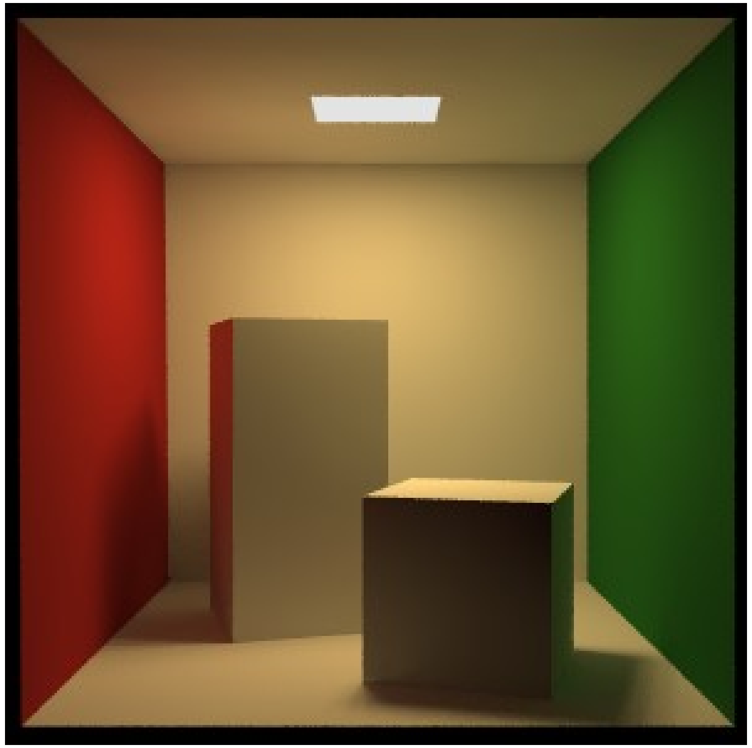


- Where does the light come from?
- Can still solve this problem for special cases, or with careful sampling, in both object-order and image-order implementations

Global Illumination (ASIDE)

- Light can bounce off walls to illuminate different surfaces
  - Red reflected light illuminates left side of left object
  - Green reflected light illuminates right side of cube on the right

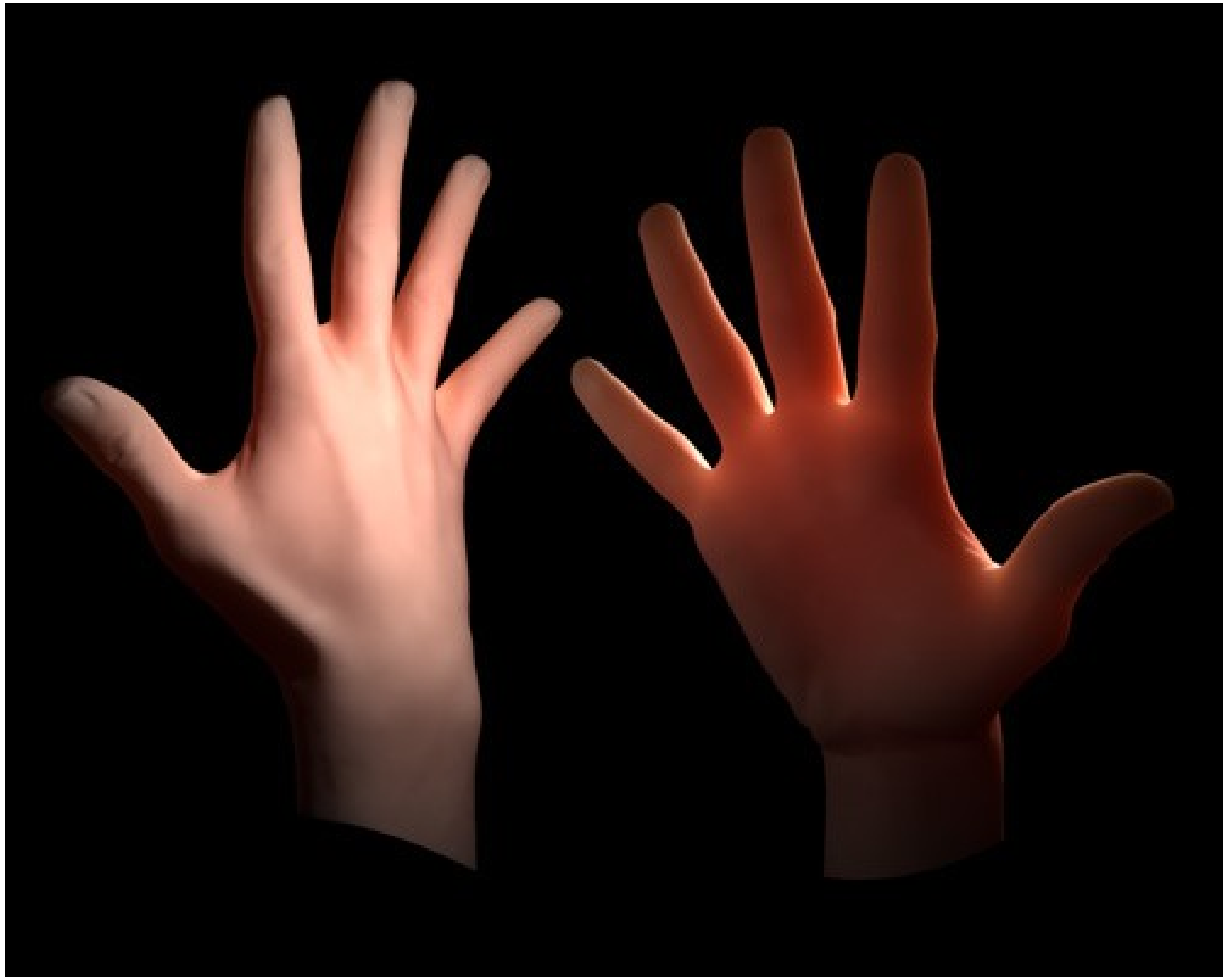
- At right: which is a photo, and which is a rendering?



Cornell PCG

Subsurface Scattering (ASIDE)

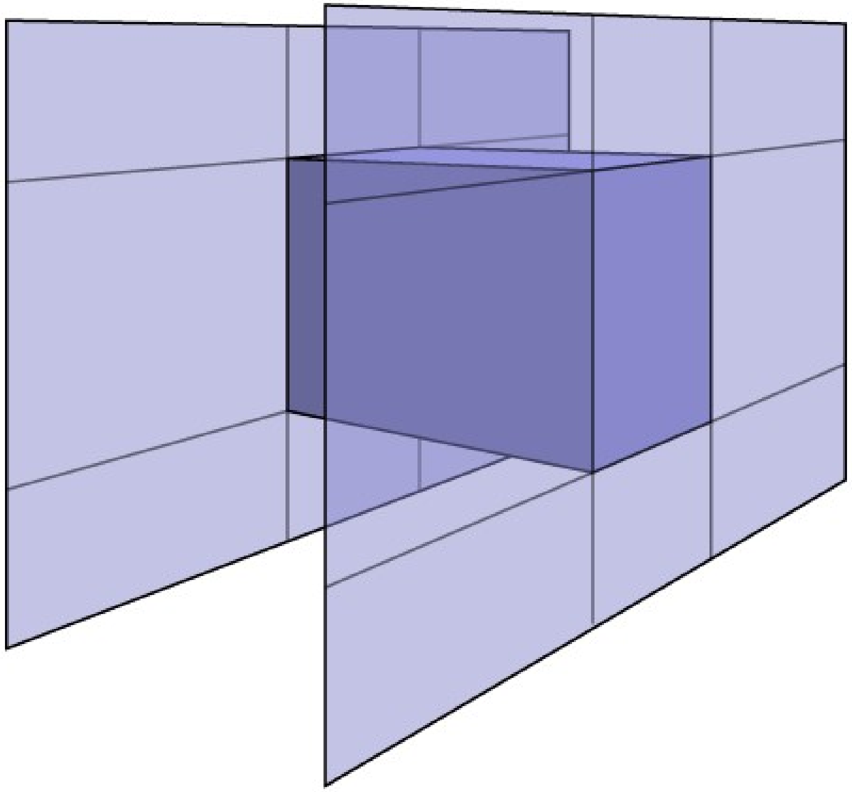
- The full problem is even more complex!!!
  - Light is ***everywhere*** and can even be ***scattered beneath*** the surface of materials
  - Devising a global illumination method that is fast and that correctly models all ***light transport*** is the ultimate problem in rendering.



[spin99, GELATO]

Ray-box intersection

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



Ray-slab intersection

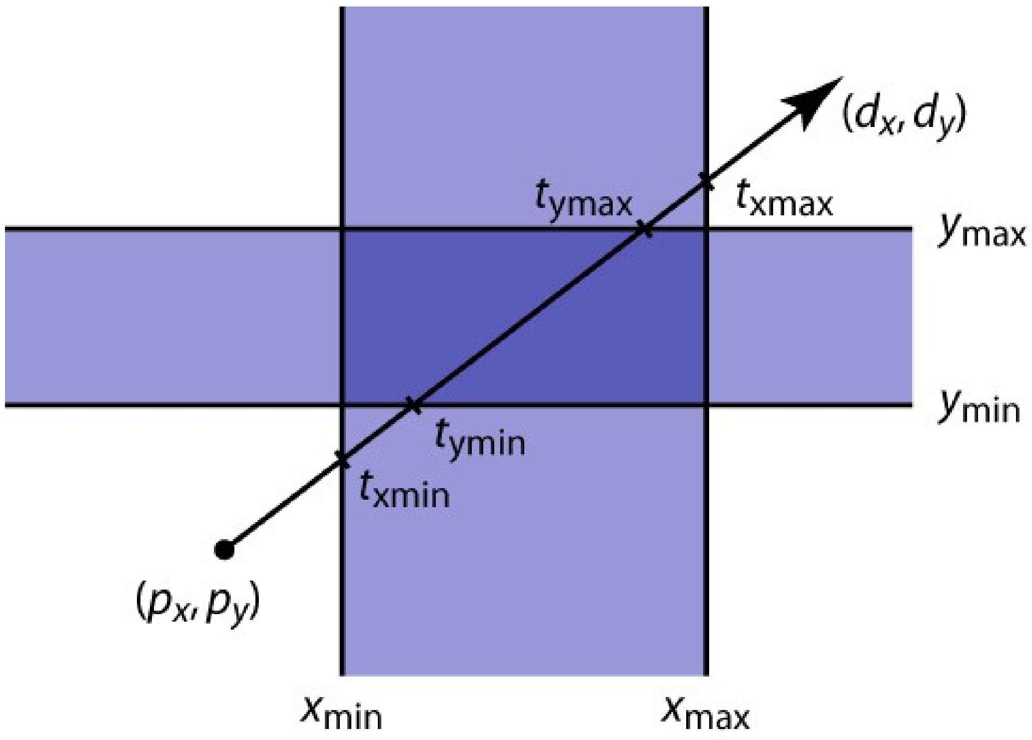
What if $d_x = 0$?
Intersection everywhere if $p_x \in [x_{\min}, x_{\max}]$,
otherwise no intersection

- 2D example
- 3D is the same!

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

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

Compute t values where
ray crosses each plane



Intersecting intersections

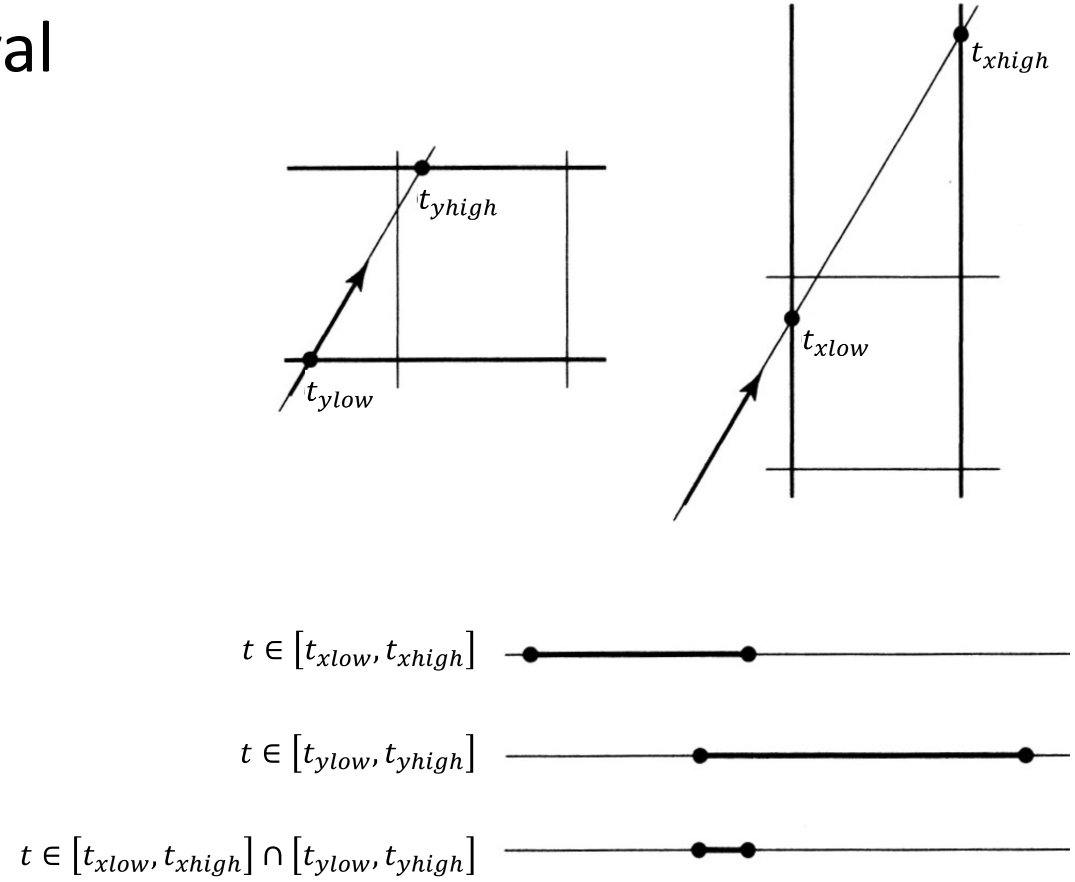
- Each intersection is an interval

  $t_{xlow} = \min(t_{xmin}, t_{xmax})$
  $t_{xhigh} = \max(t_{xmin}, t_{xmax})$

- Want last entry point and first exit point

  $t_{min} = \max(t_{xlow}, t_{ylow})$
  $t_{max} = \min(t_{xhigh}, t_{yhigh})$

- If $t_{max} < t_{min}$ then empty interval, i.e., no intersection



Ray-triangle intersection

- Condition 1: point is on ray

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

- Condition 2: point is on plane

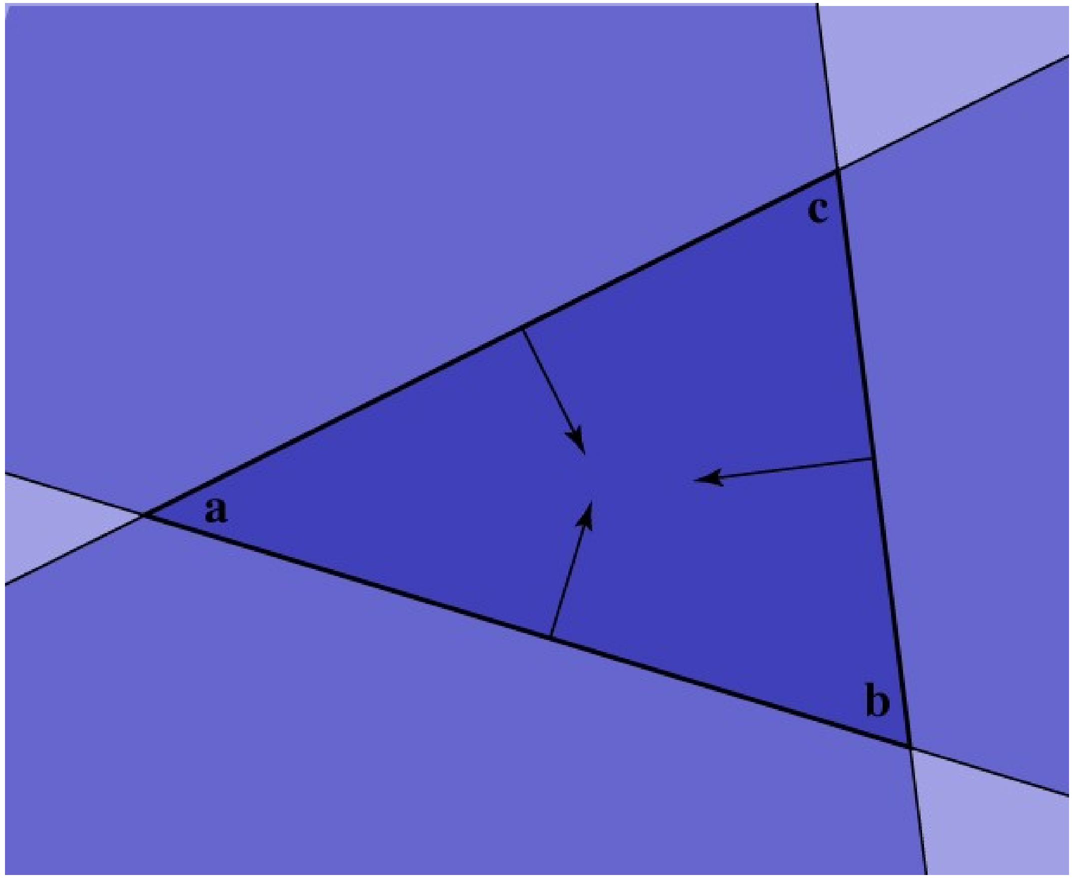
$$(\mathbf{x} - \mathbf{a}) \cdot \mathbf{n} = 0$$

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

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

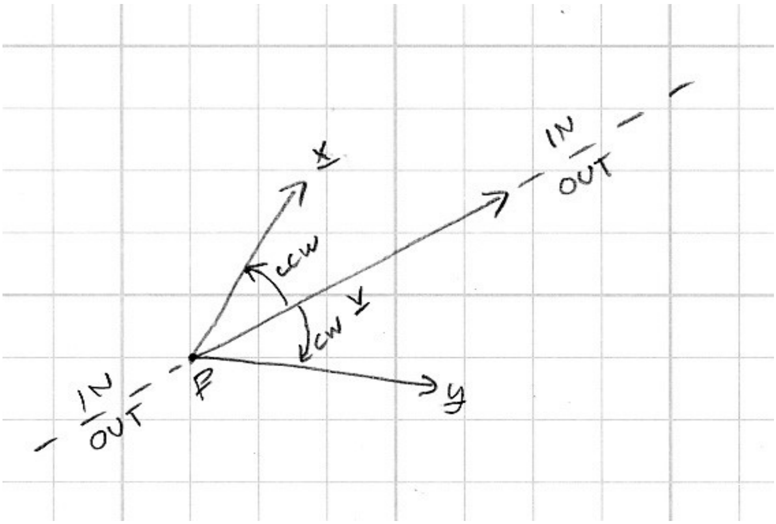
Ray-triangle intersection

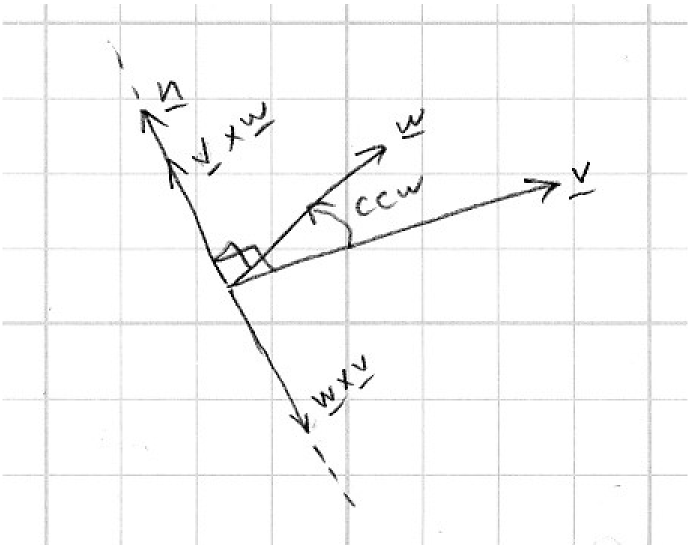
- In plane, the triangle is the intersection of 3 half spaces



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

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

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

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

- Can see this as a step toward computing barycentric coordinates (useful as texture coordinates)
- Can also use a similar inside/outside test on non-convex n-gons, by computing the winding number (see CGPP 7.10)

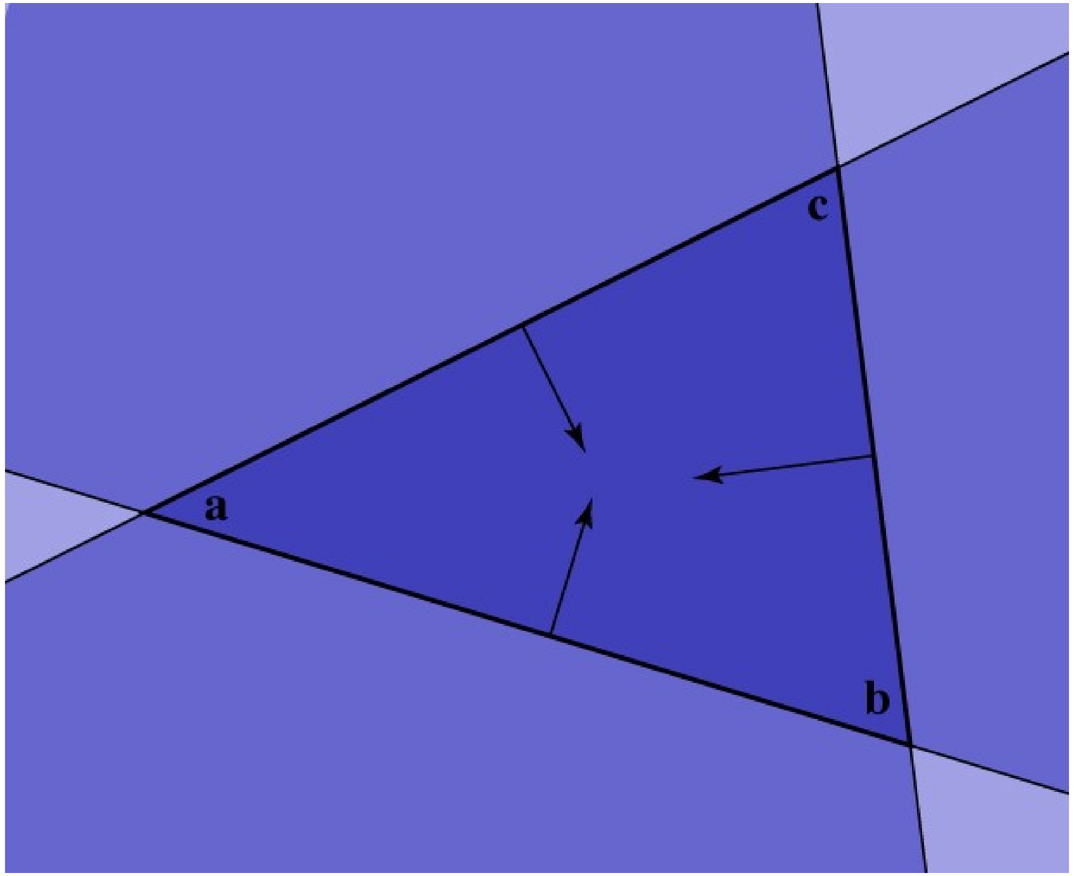
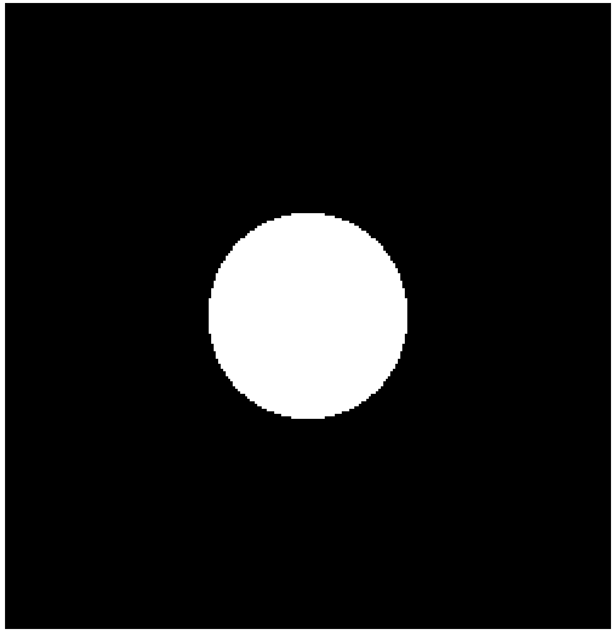


Image so far

- 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);
       }
   }
}
```



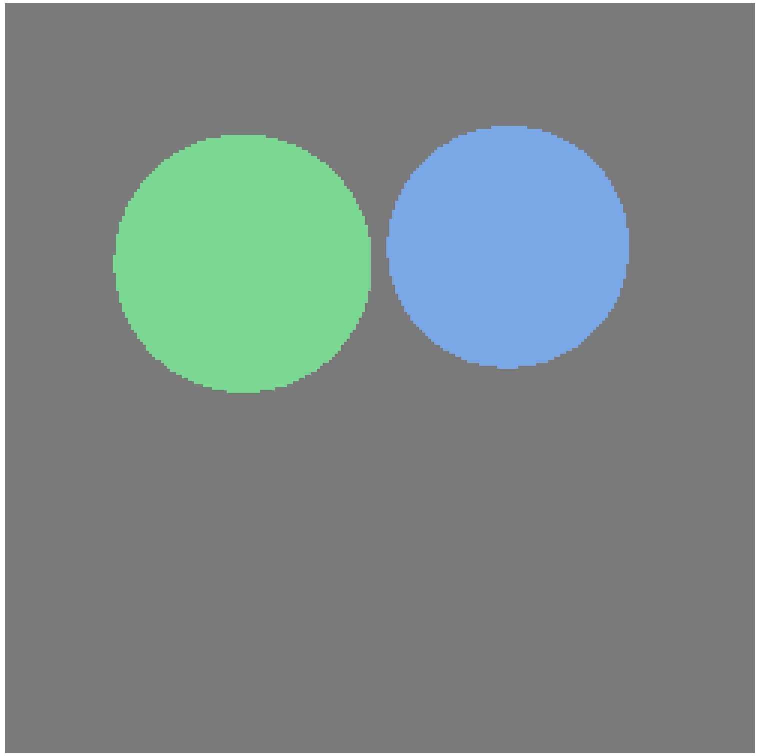
Intersection against many shapes

```
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

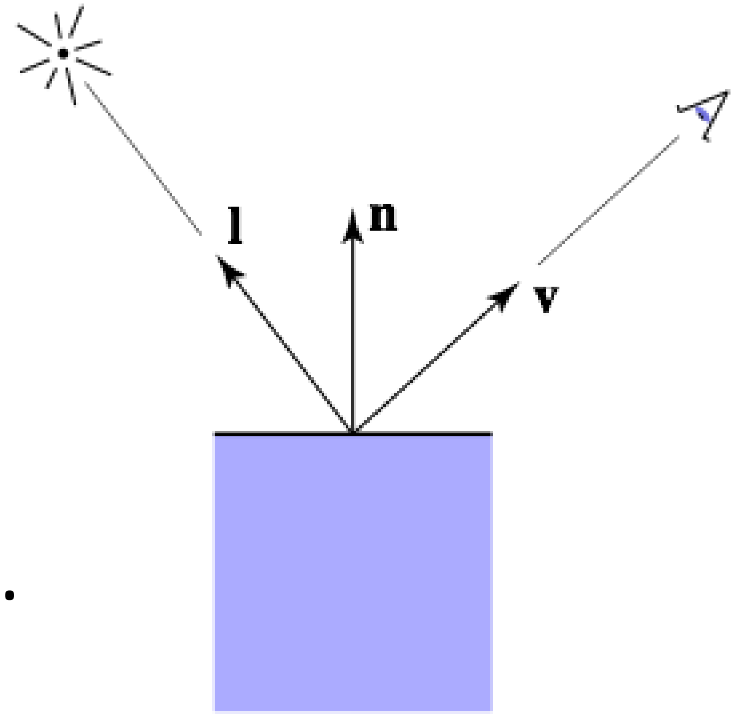
- With eye ray generation and scene intersection

```
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…

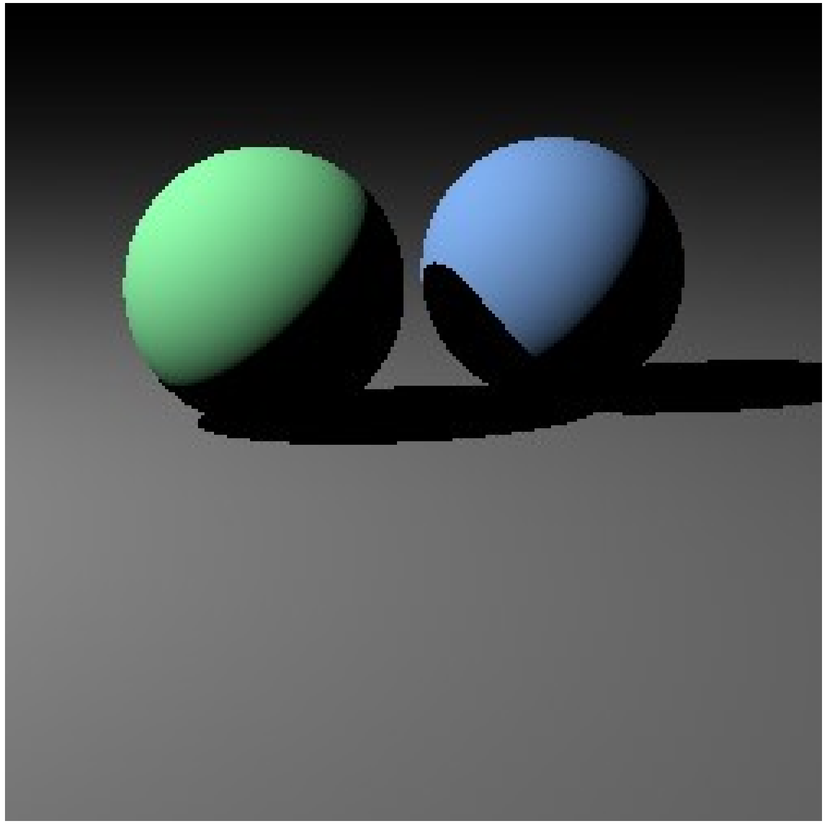


Shadows

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

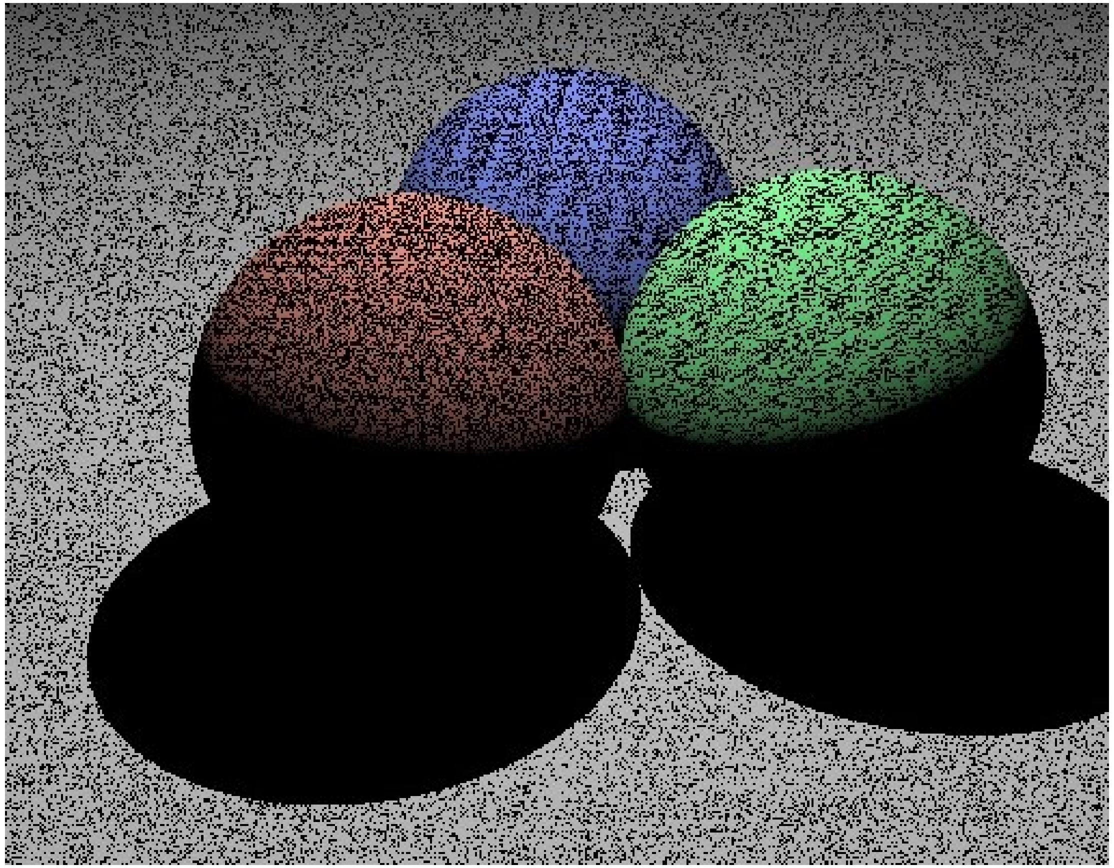
Image so far

```
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;
}
```



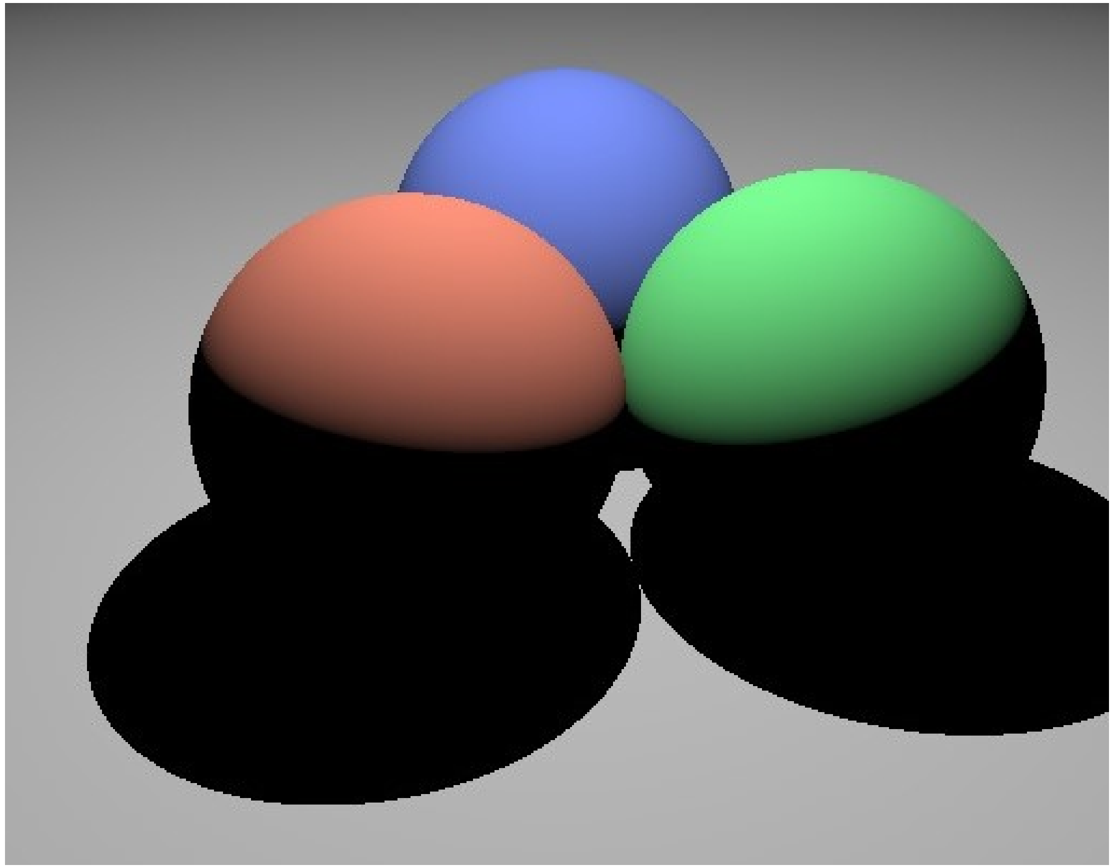
Shadow rounding errors

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



Mirror reflection

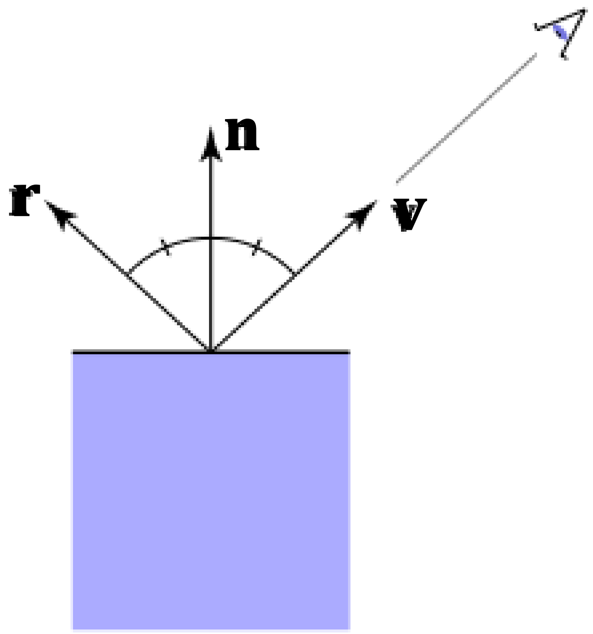
- Consider perfectly shiny surface
  - There isn't a highlight (i.e., no L_s)
  - Instead there's a reflection of other objects
- Can render this using recursive ray tracing
  - To find out mirror reflection color, ask what color is seen from surface point in reflection direction
  - Already computing reflection direction for Phong?
- "Glazed" material has mirror reflection and diffuse

$$L = L_a + L_d + L_m$$

  - Here L_m is evaluated by tracing a new ray

Mirror reflection

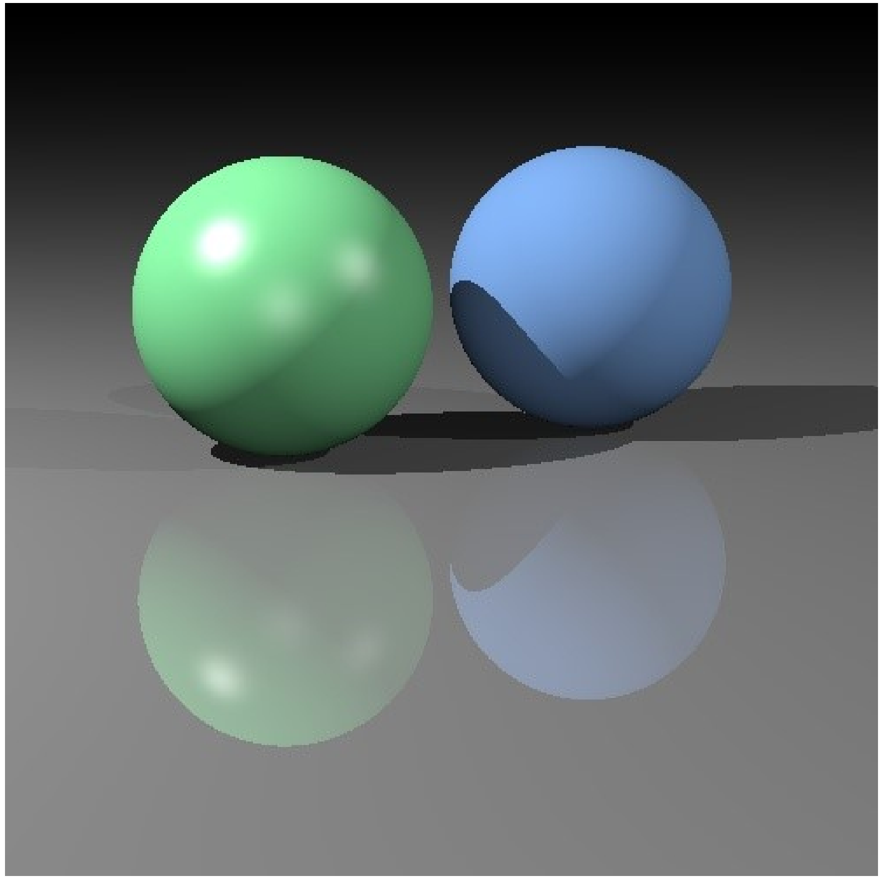
- Intensity depends on view direction
  - Reflects incident light from mirror direction



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

$$\begin{aligned}\mathbf{r} &= \mathbf{v} + 2\big(\mathbf{n}\big(\mathbf{n}^T\mathbf{v}\big) - \mathbf{v}\big)\\ &= 2\big(\mathbf{n}\mathbf{n}^T\big)\mathbf{v} - \mathbf{v}\\ &= \big(2\mathbf{n}\mathbf{n}^T - I\big)\mathbf{v}\end{aligned}$$

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

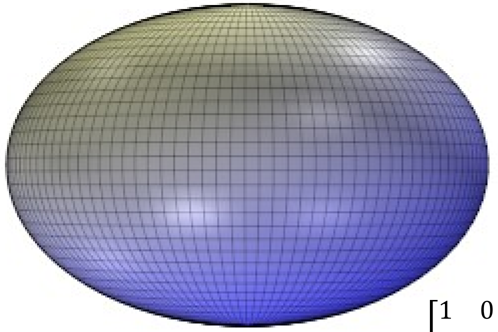
- Return values
  - Surface intersection tends to want to return multiple values
    - Scalar t, surface or shader, normal vector, maybe surface point
  - Typical solution: an *intersection record*
    - A class with fields for all these things
    - Keep track of the intersection record for the closest intersection
- Efficiency
  - In Java to be fast one should 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 that blocks the light 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…

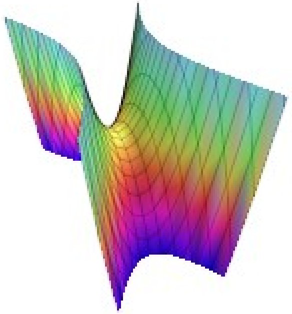
Quadrics

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

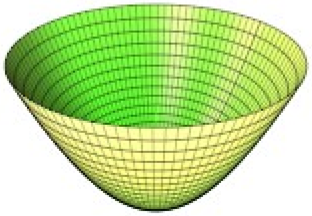


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

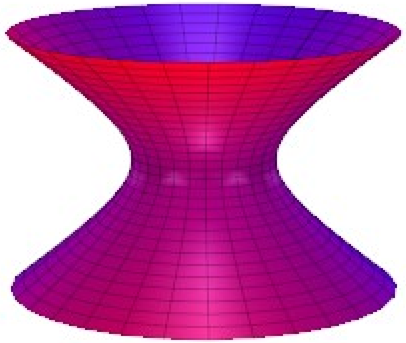
Ellipsoid



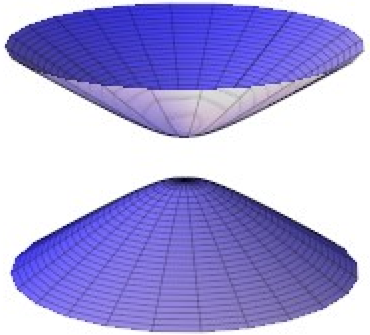
Hyperbolic paraboloid



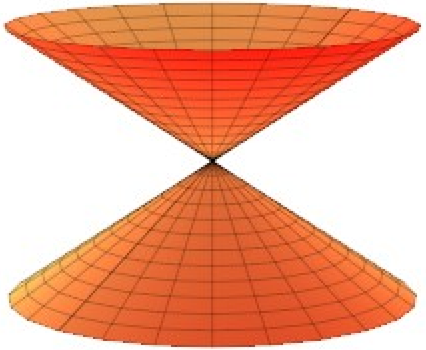
Elliptic paraboloid



Hyperboloid of one sheet



Hyperboloid of two sheets



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Cone

Quadrics

- In non-homogeneous coordinates we can write

$$[x\ y\ z]A\begin{bmatrix}x\\y\\z\end{bmatrix} - 2\mathbf{b}^T\begin{bmatrix}x\\y\\z\end{bmatrix} + c = 0 \qquad A \in \mathbb{R}^{3\times 3} \quad \mathbf{b} \in \mathbb{R}^3 \quad c \in \mathbb{R}$$

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

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

- Solution is same as ray sphere intersection.
  - Replace x with ray equation, expand, solve for t
  - **Given intersection point x, what is the normal?** $\nabla F(x) = 2Ax - 2\mathbf{b}^T$

Review and More Information

- CGPP 15.4 ray tracing in general
- CGPP 14.5.2 ray tracing implicit surfaces
- Ray-Triangle intersection, see also FCG Section 4.4.2 for method based on linear systems and Cramer's rule, but see also Section 2.7 with respect to barycentric coordinates