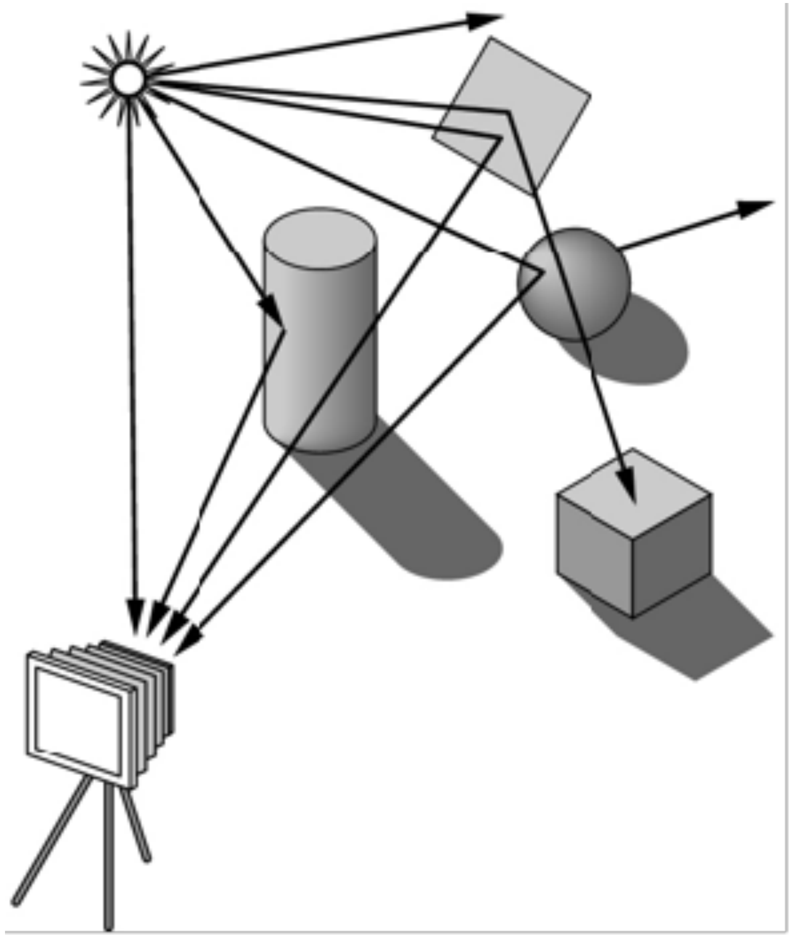
Computer Graphics -Ray tracing

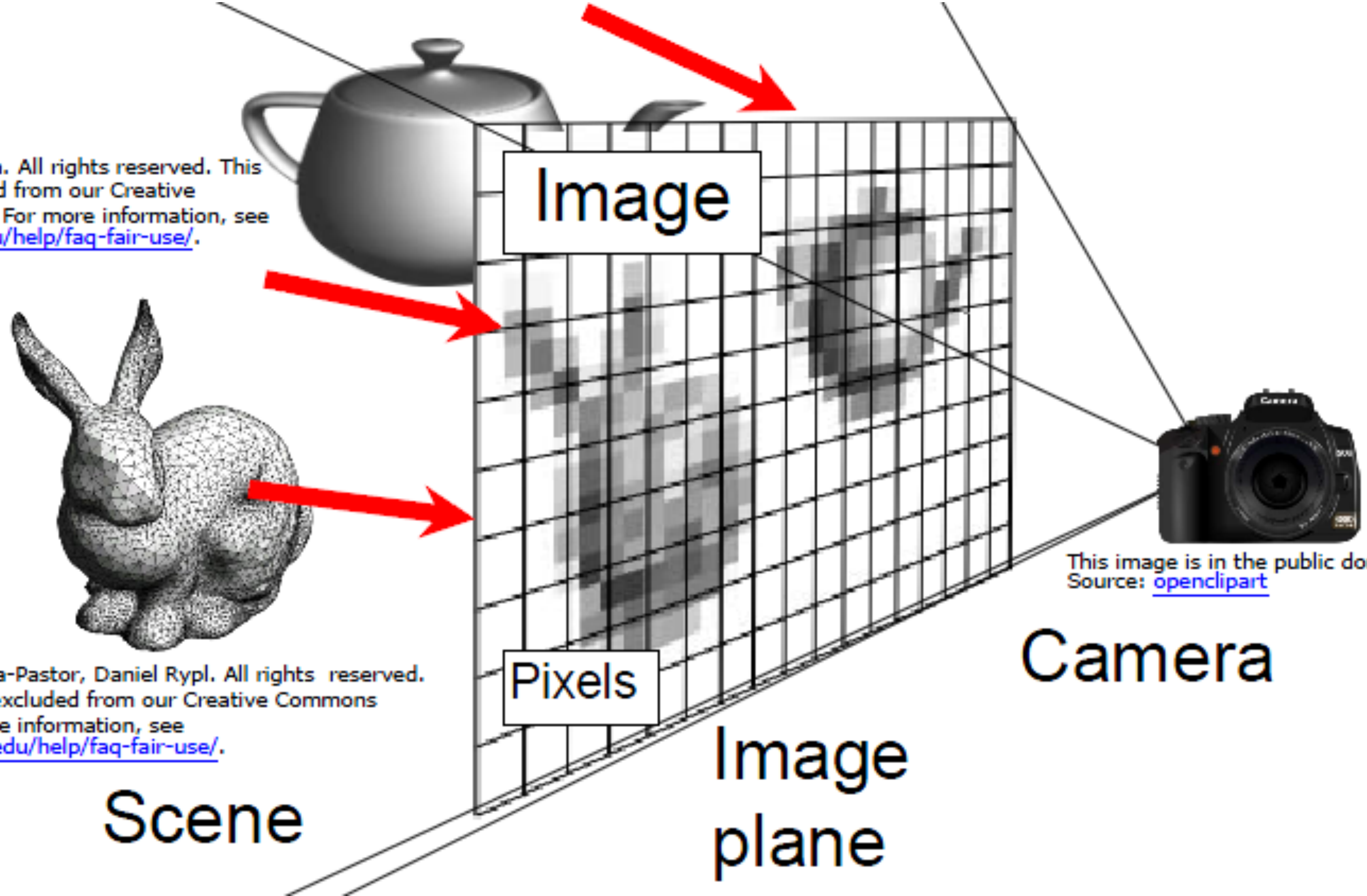
Junjie Cao @ DLUT

Spring 2019

http://jjcao.github.io/ComputerGraphics/



Rendering = Scene to Image



Two approaches to rendering

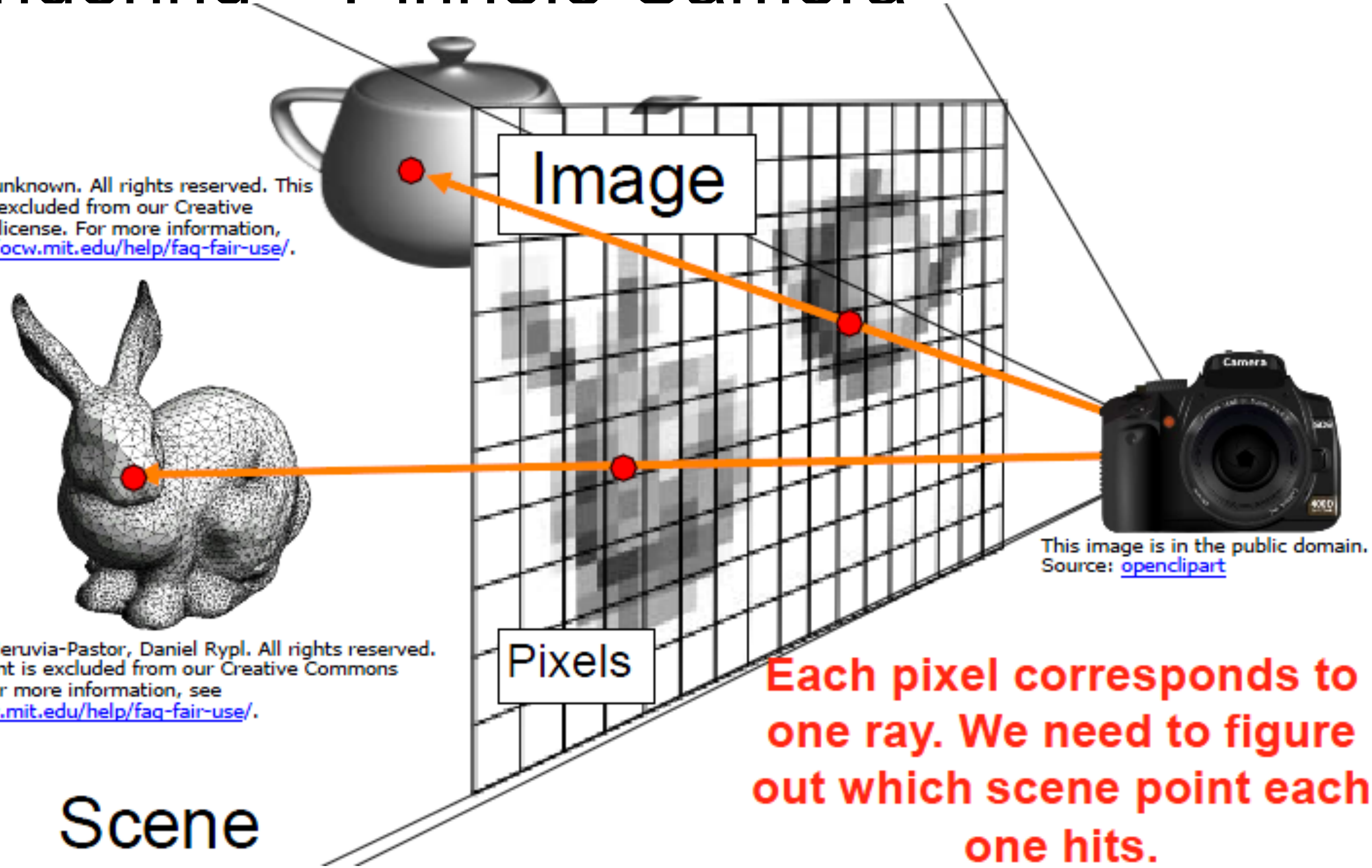
```
for each object in the scene {
  for each pixel in the image {
    if (object affects pixel) {
      do something
    }
  }
}
```

object order
or
rasterization

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

image order
or
ray tracing

Rendering – Pinhole Camera



Each pixel corresponds to one ray. We need to figure out which scene point each one hits.

Rendering

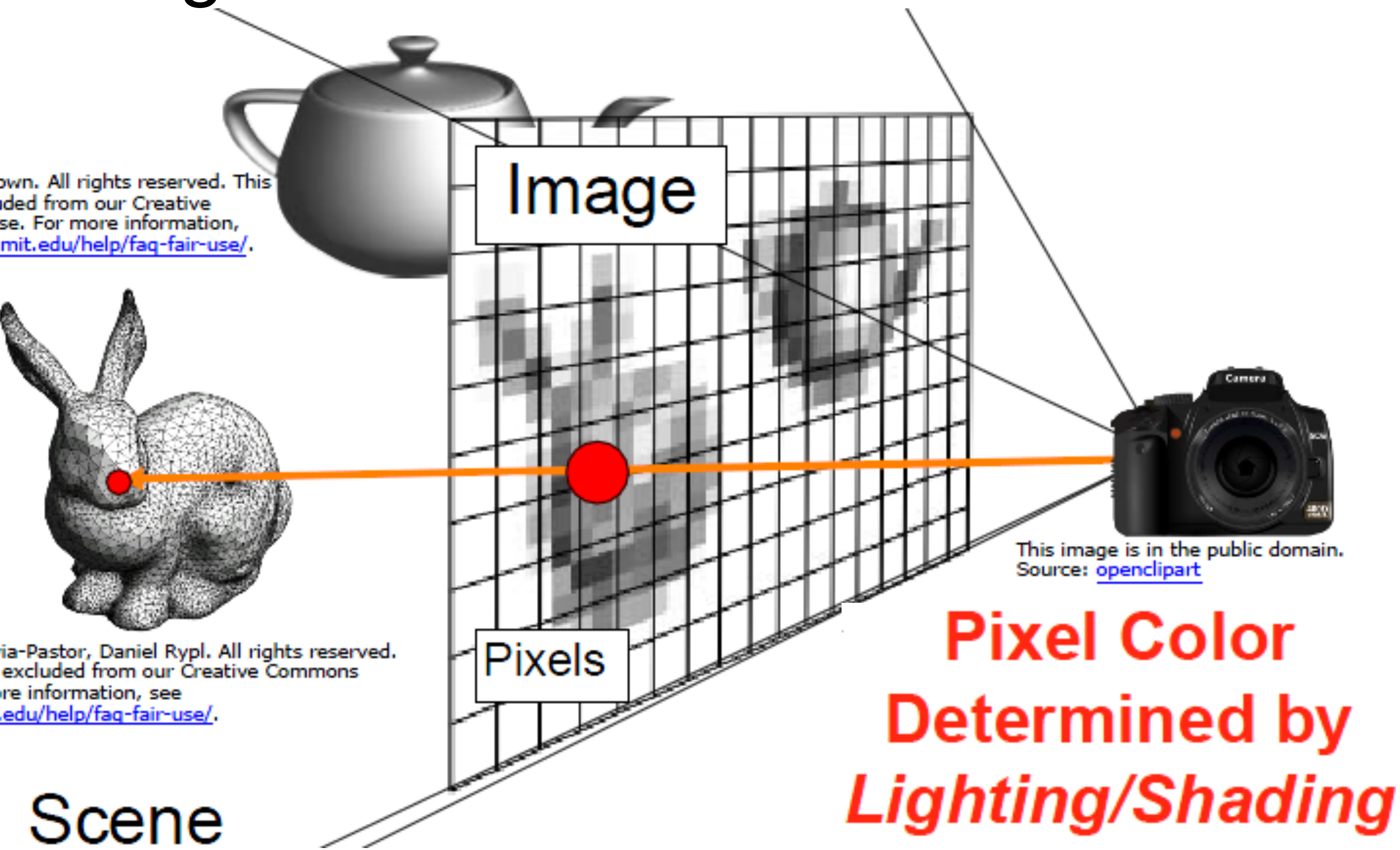


Image
Pixels
Scene
Pixel Color Determined by *Lighting/Shading*

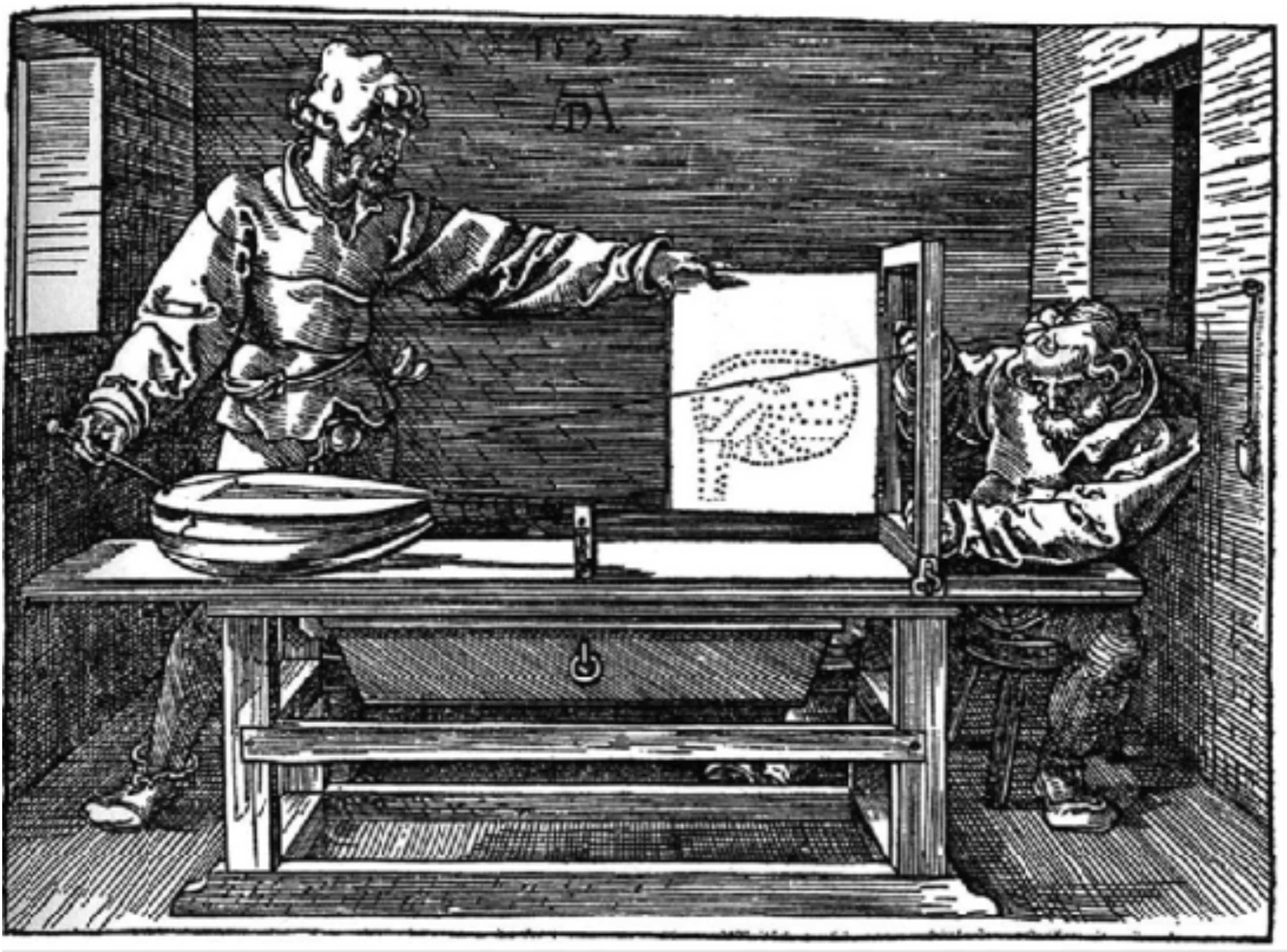
Dürer’s Ray Casting Machine

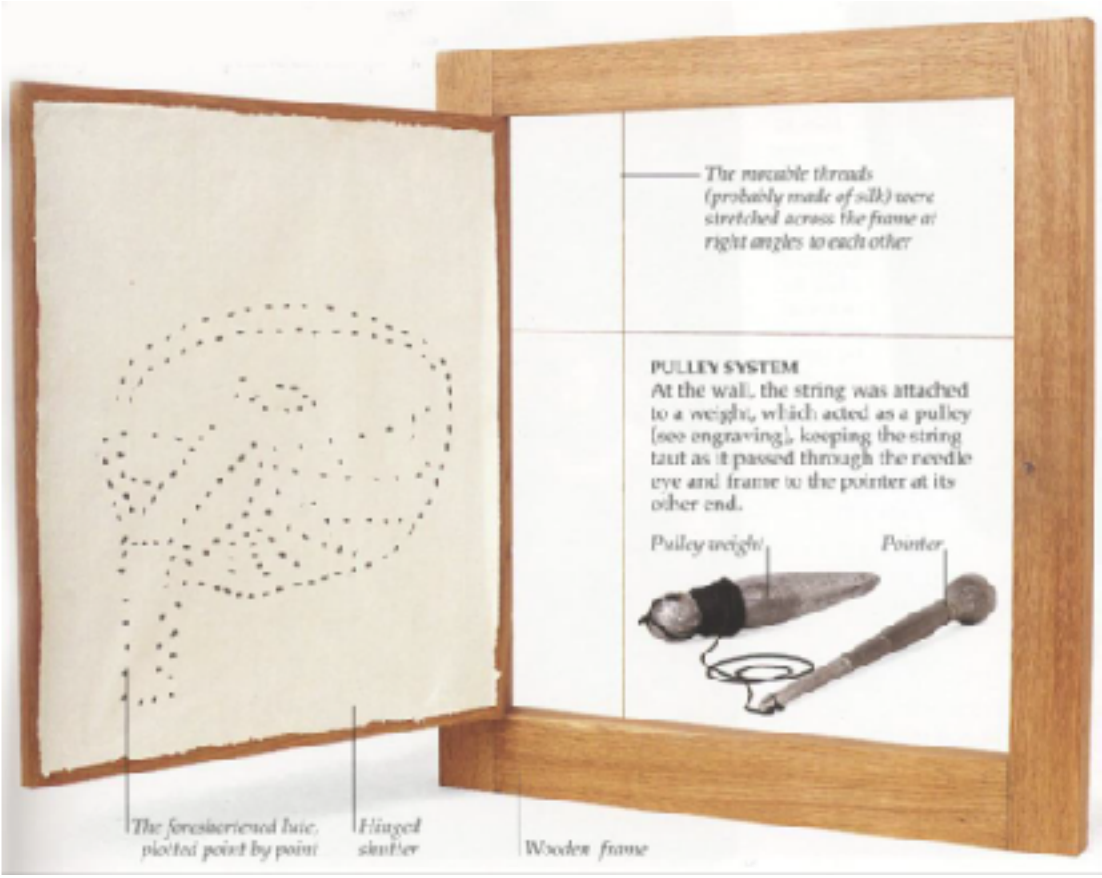
- Albrecht Dürer, 16th century

Dürer's Ray Casting Machine

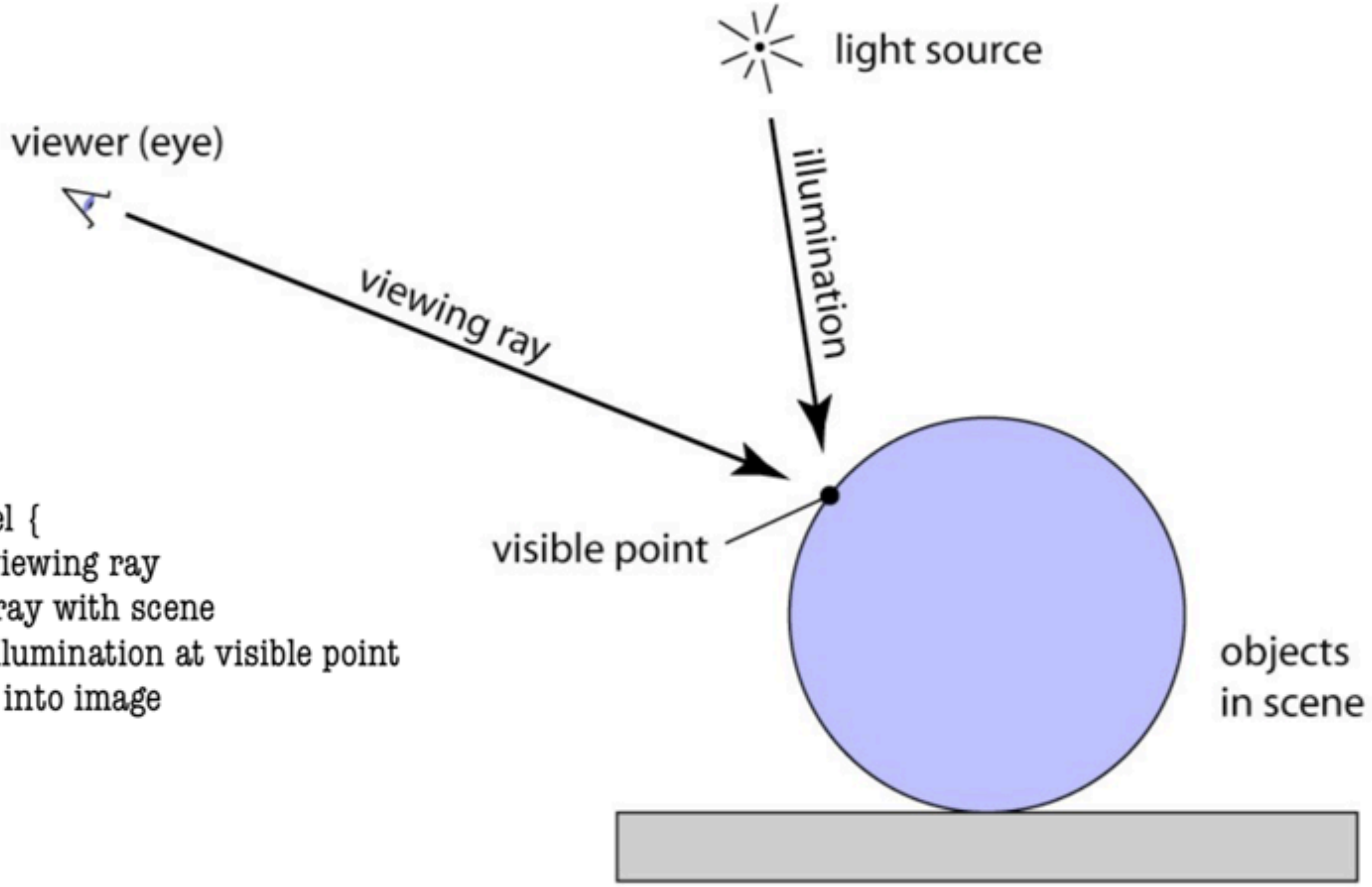
- Albrecht Dürer, 16th century

Dürer’s Ray Casting Machine



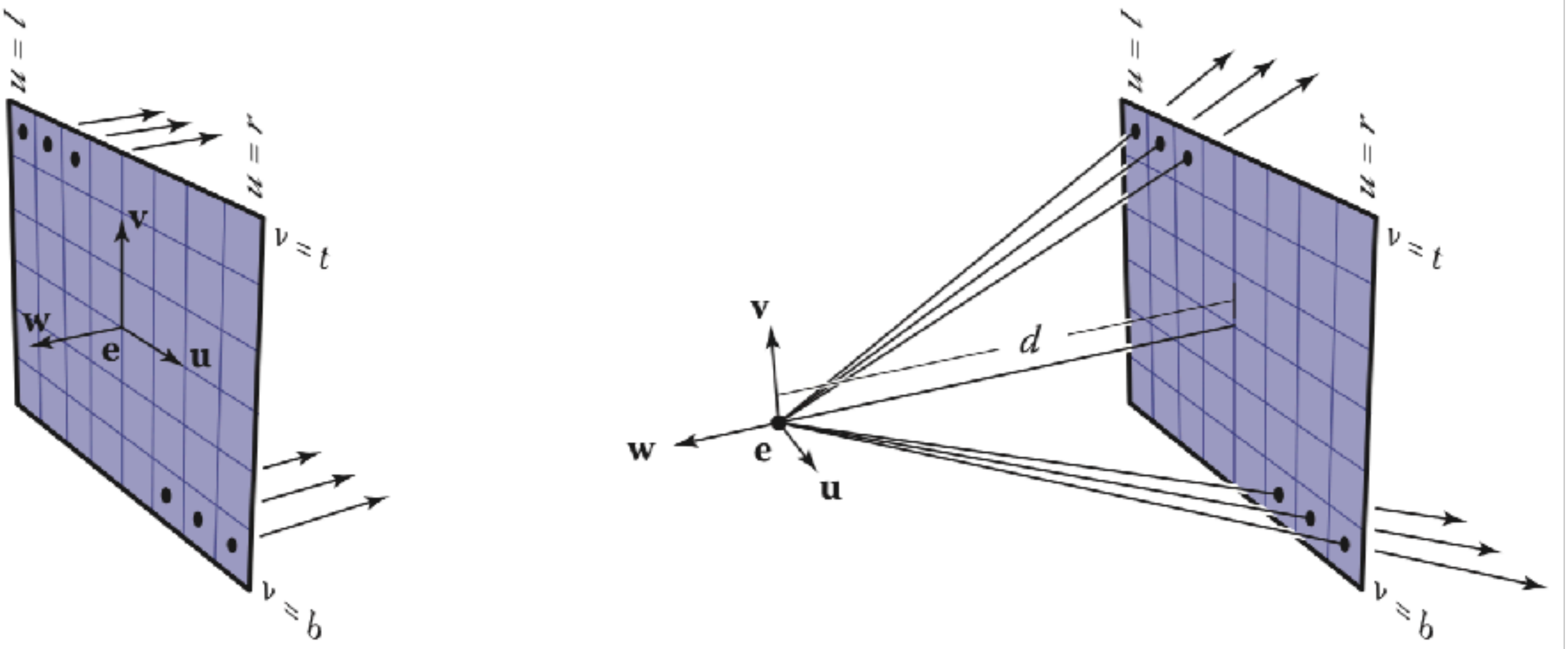


Ray tracing algorithm




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

Generating eye rays



Parallel projection
same direction, different origins

Perspective projection
same origin, different directions

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

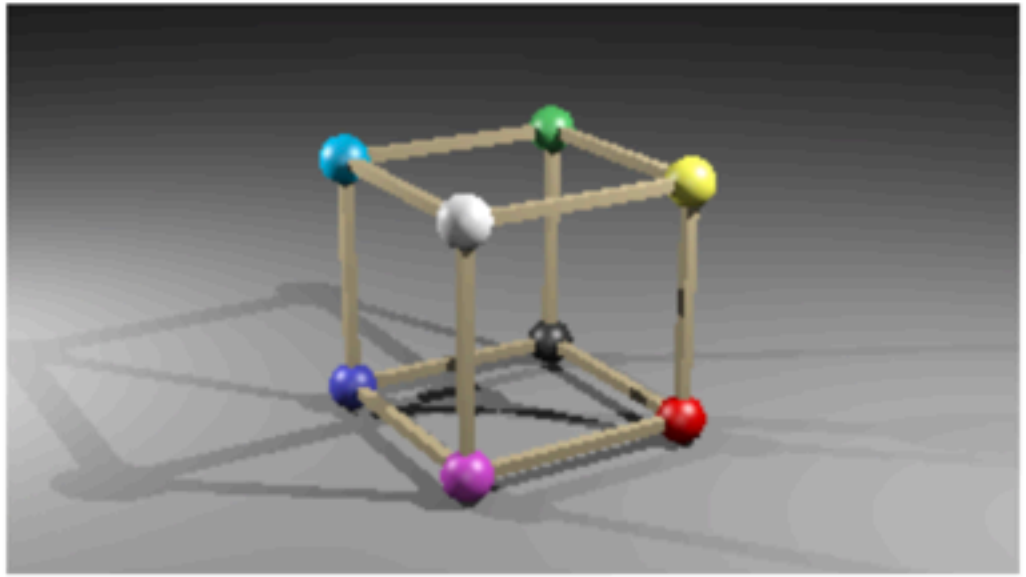
- **Modularity problem: Camera shouldn't have to worry about image resolution**
  - better solution: normalized coordinates

```
class Camera {
  ...
  Ray generateRay(float u, float v);
}
```

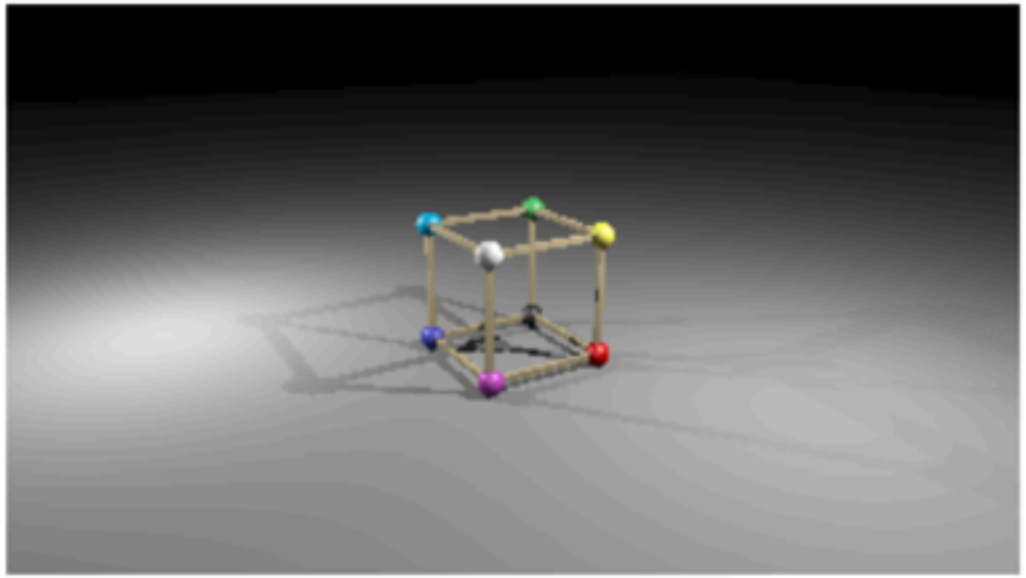
← args go from 0, 0 to 1, 1

Specifying views in Ray 1

```
<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>
```

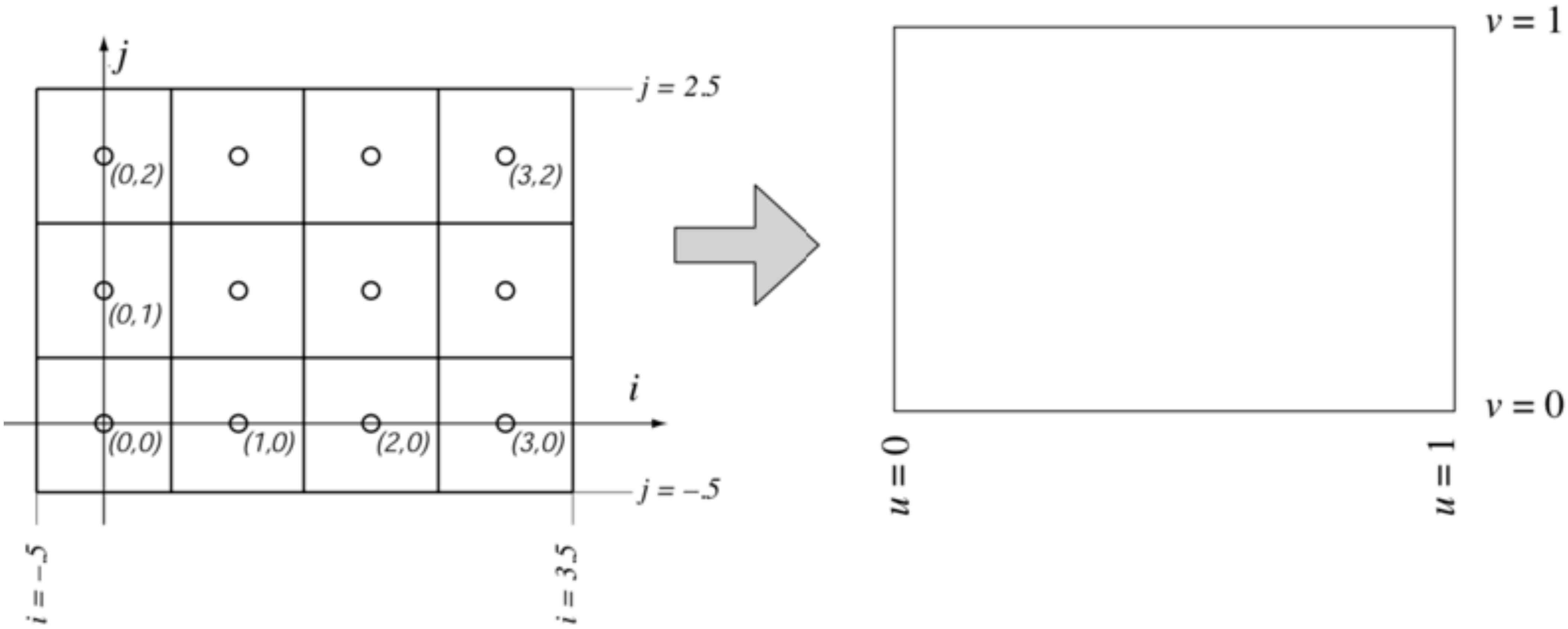


```
<camera type="PerspectiveCamera">
  <viewPoint>10 4.2 6</viewPoint>
  <viewDir>-5 -2.1 -3</viewDir>
  <viewUp>0 1 0</viewUp>
  <projDistance>3</projDistance>
  <viewWidth>4</viewWidth>
  <viewHeight>2.25</viewHeight>
</camera>
```



Pixel-to-image mapping

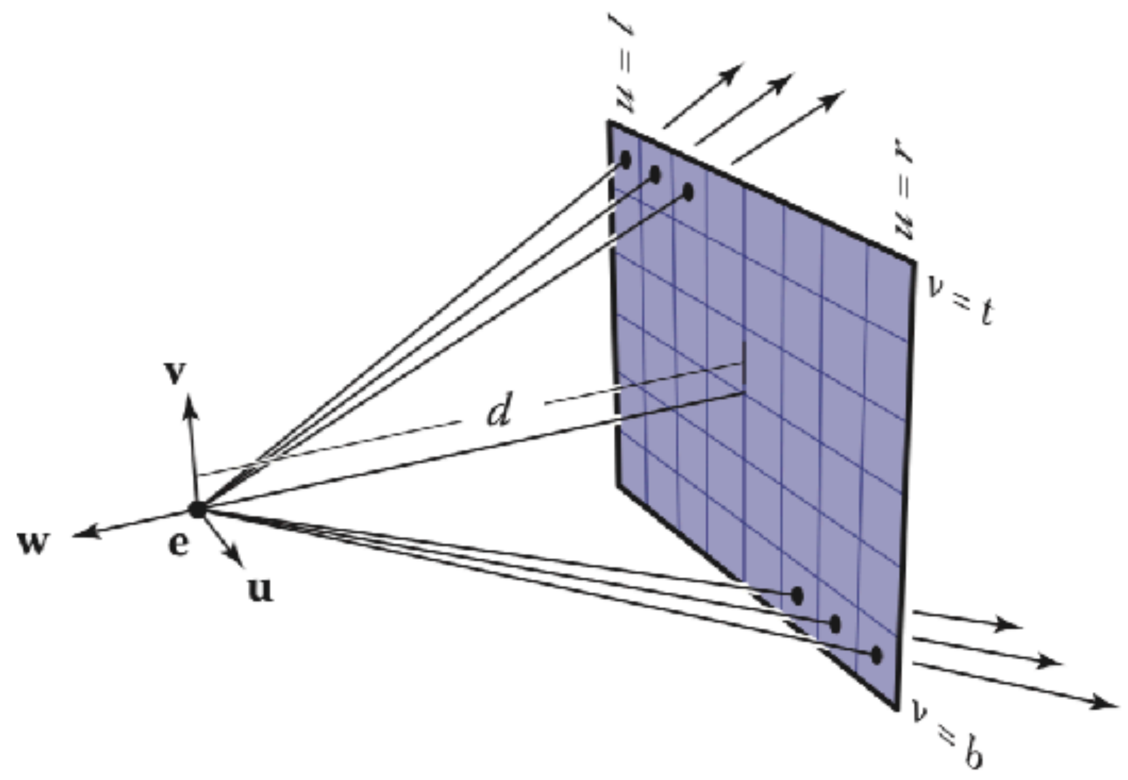
- **One last detail: exactly where are pixels located?**

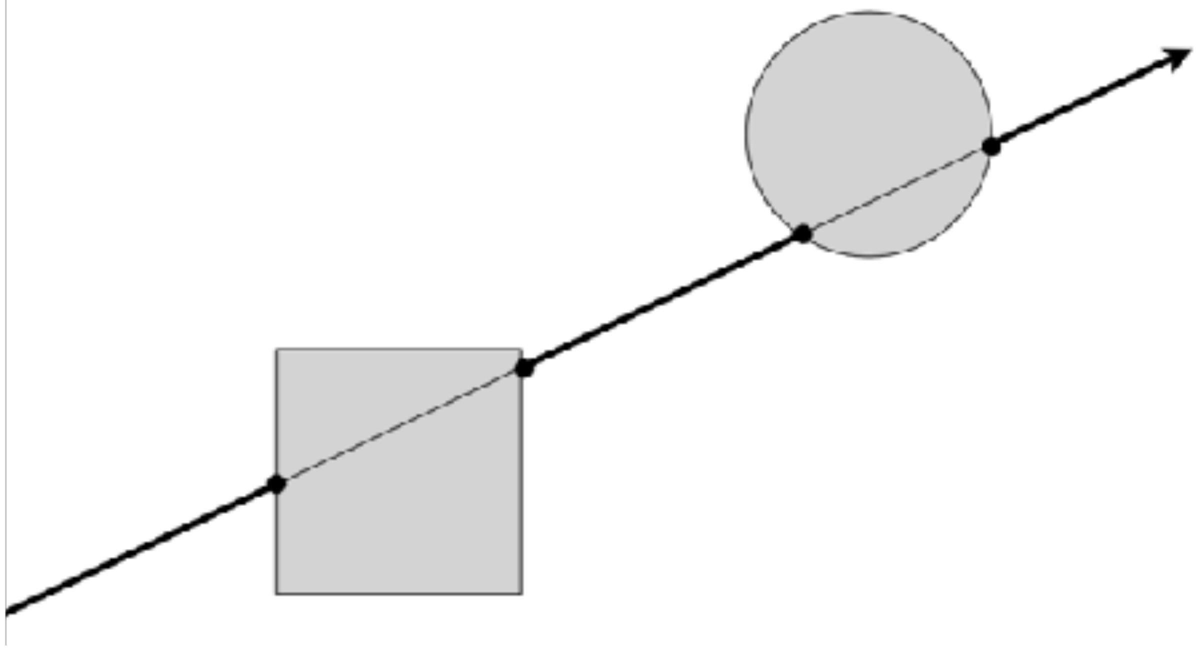


$$u = (i + 0.5)/n_x$$

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

Ray intersection



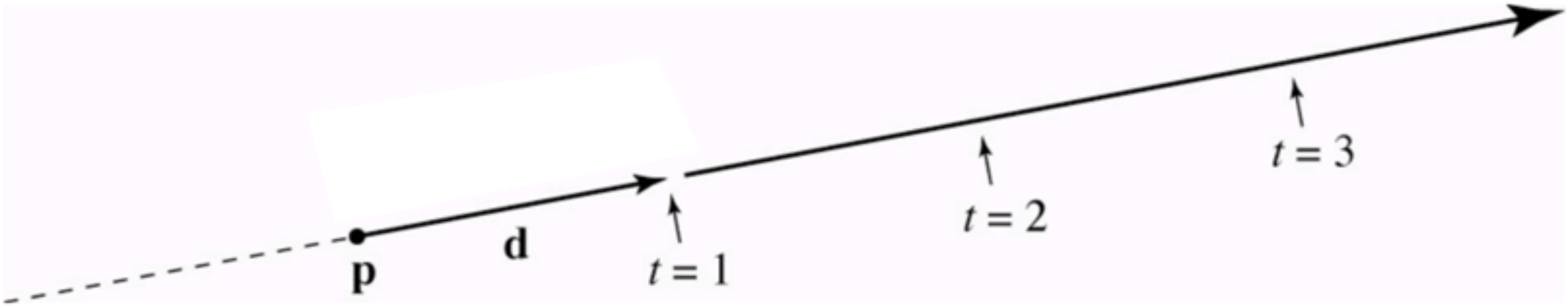


Ray: a half line

- **Standard representation: point 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 **αd** doesn't change ray (α > 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

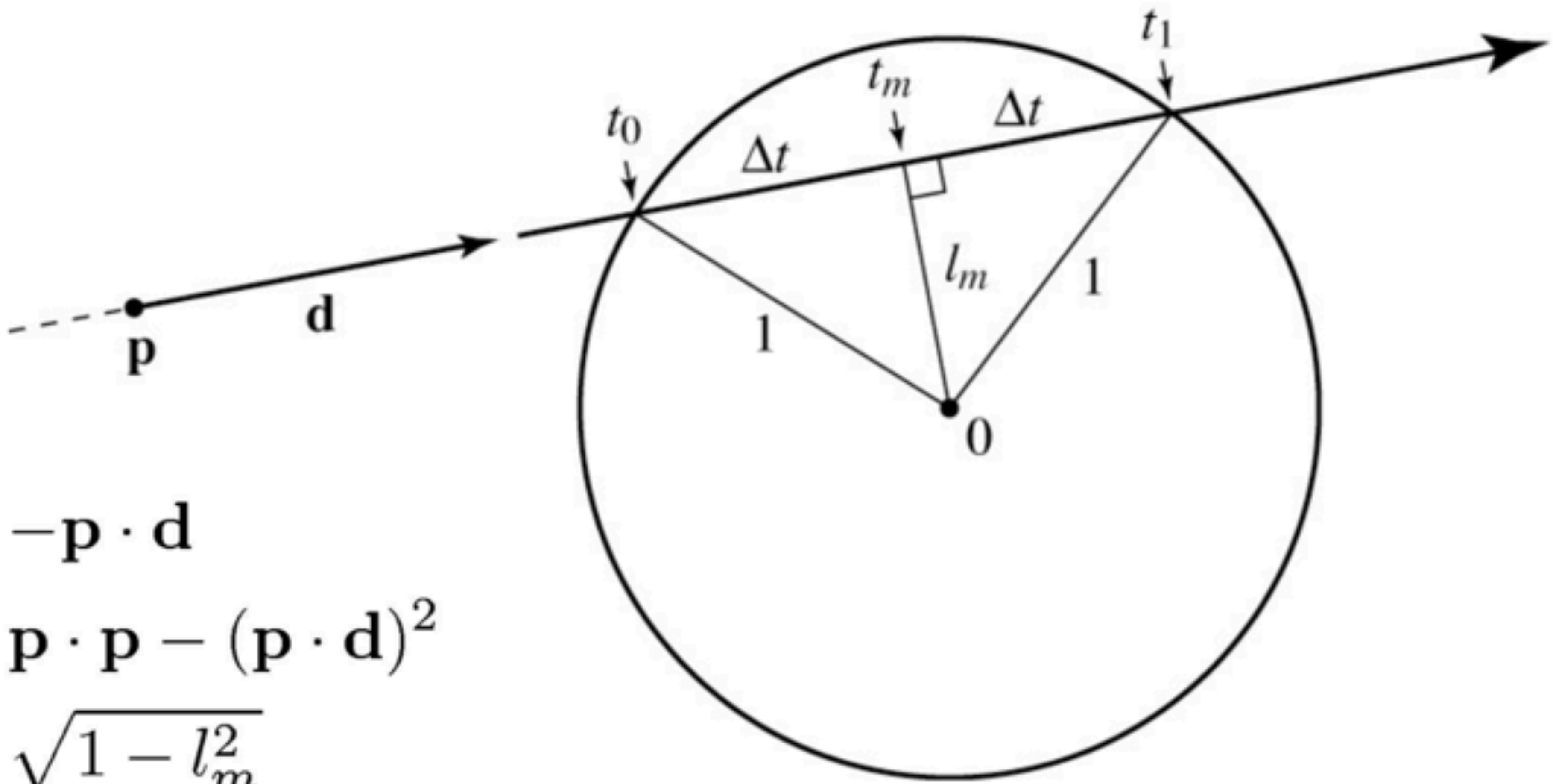
- **Solution for *t* by quadratic formula:**

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

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

  - 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**

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

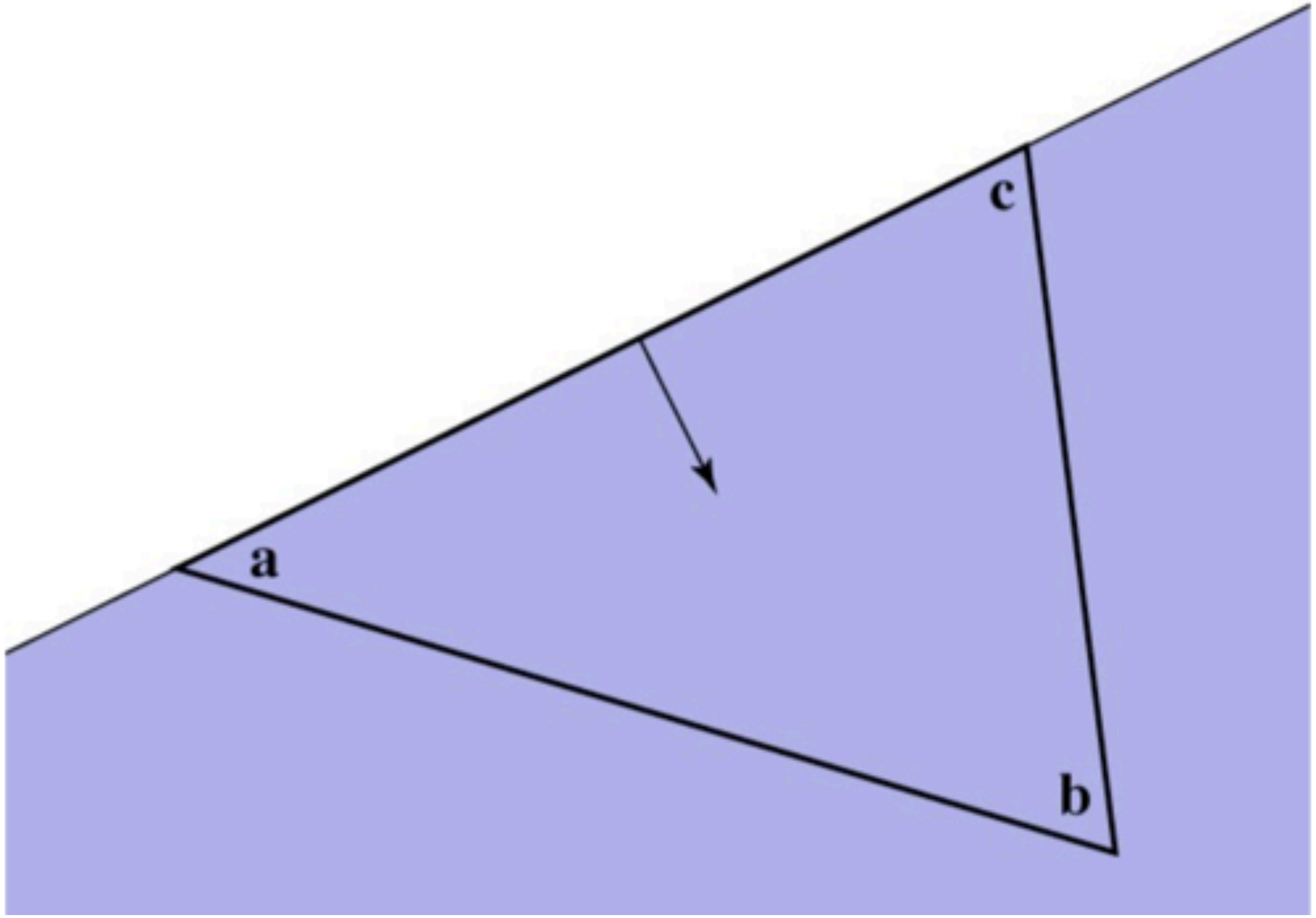
- **Condition 3: point is on the inside of all three edges**
- **First solve 1&2 (ray–plane intersection)**
  - substitute and solve for *t*:

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

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

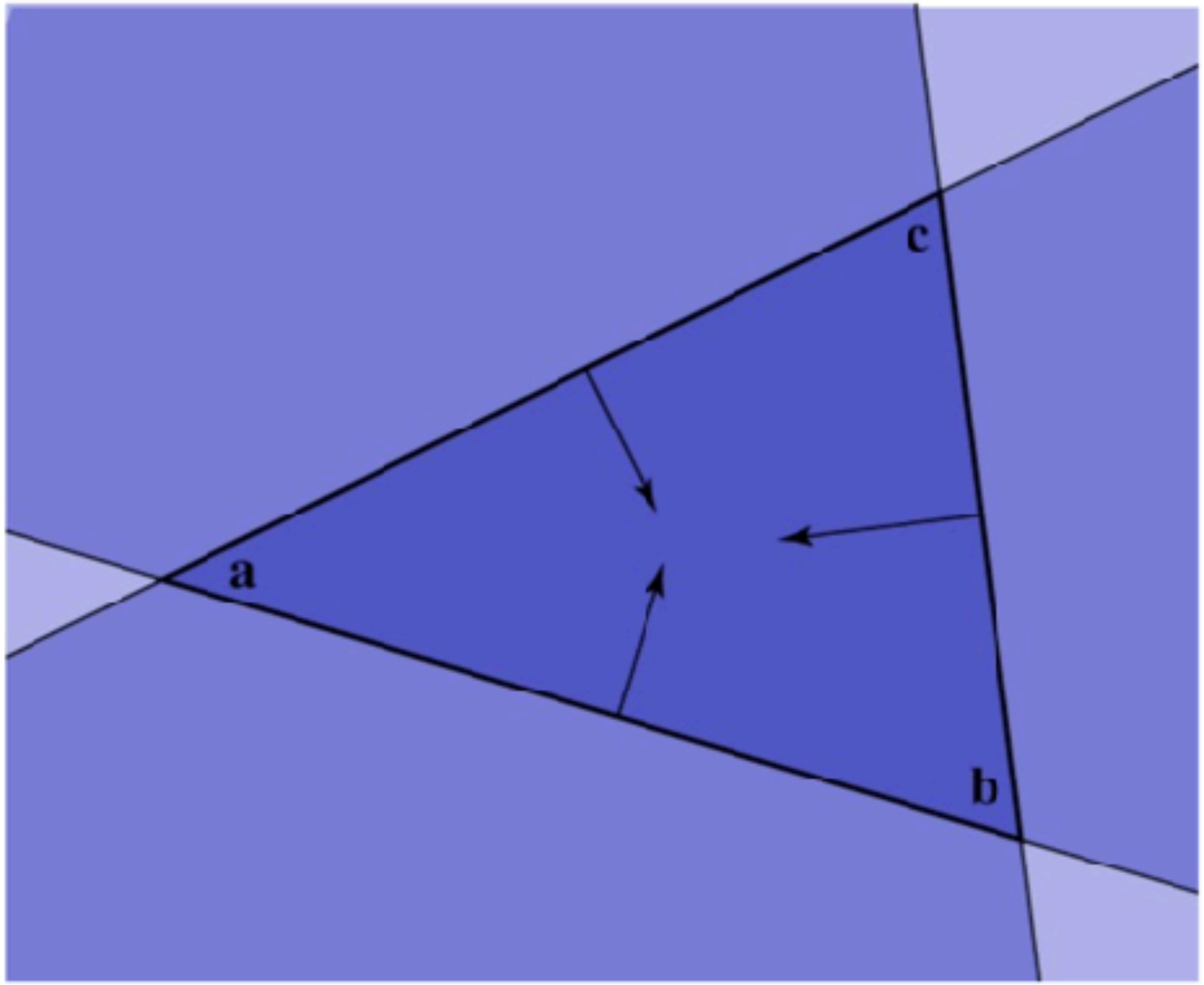
Ray-triangle intersection

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



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**

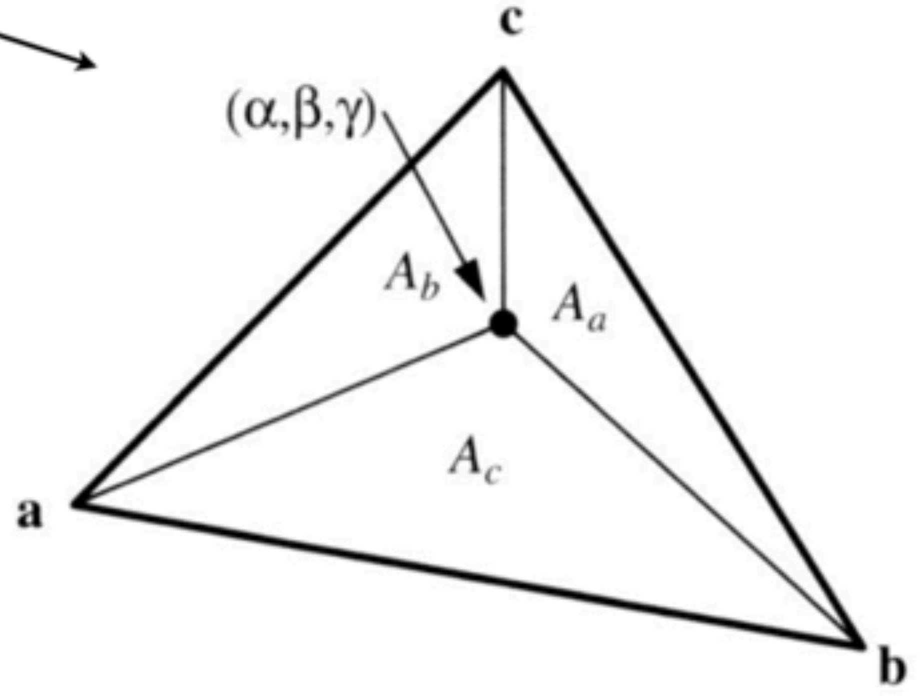
Barycentric coordinates

- **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):

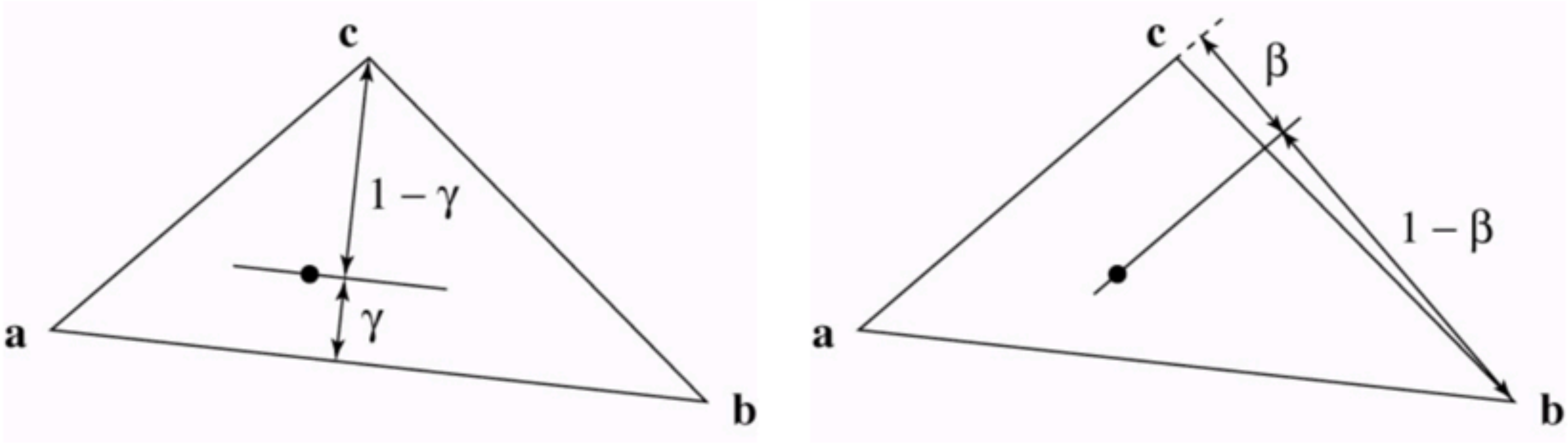


- **Triangle interior test:**

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

Barycentric coordinates

- **A coordinate system for triangles**
  - geometric viewpoint: distances



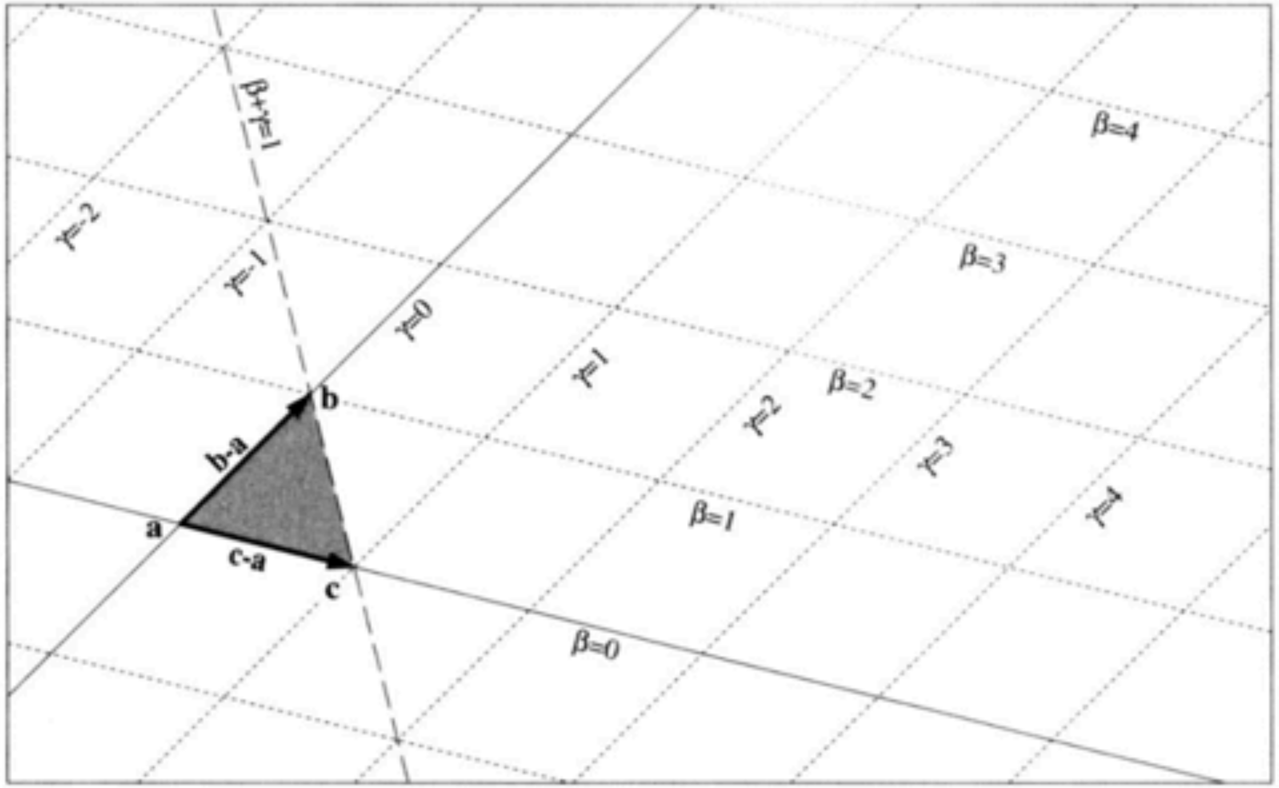
  - linear viewpoint: basis of edges

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

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

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 β and γ.

- **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, β, and γ all at once!

Barycentric ray-triangle intersection

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

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

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

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

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.**

ray to be intersected

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

was there an intersection?

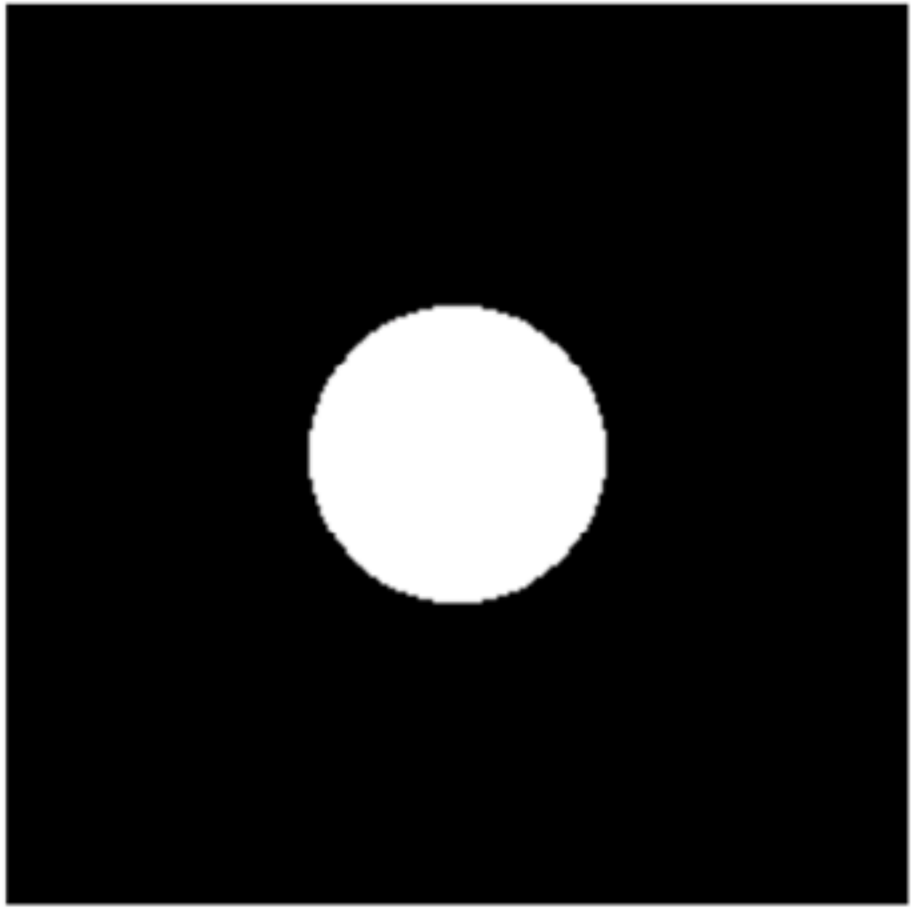
information about first intersection

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

Image so far

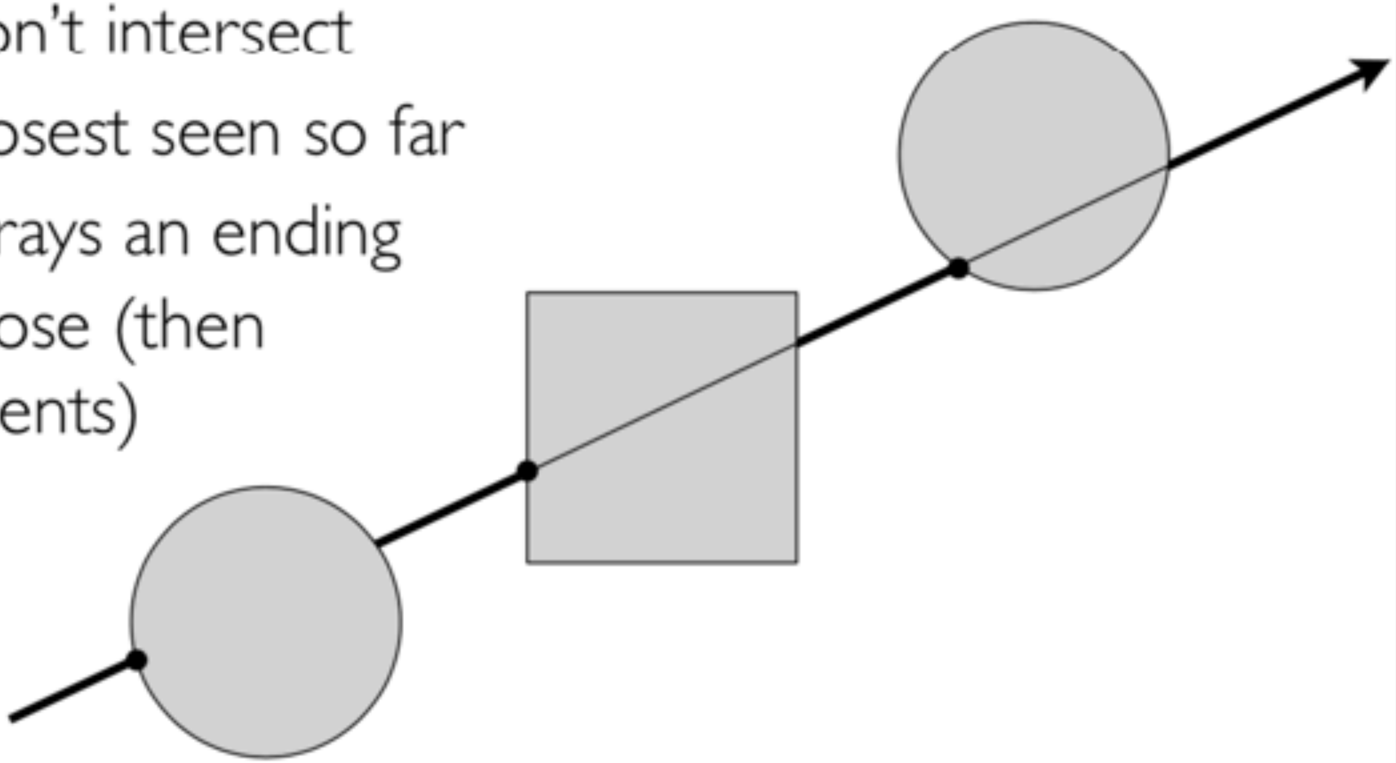
- **With eye ray generation and sphere intersection**

```
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:

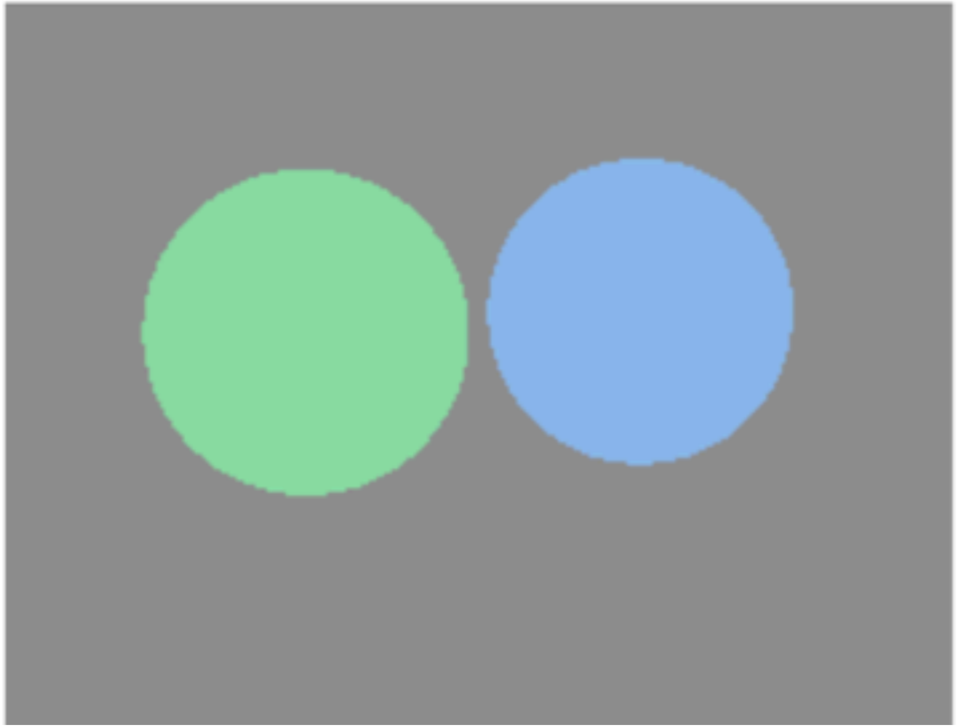
```
intersect (ray, tMin, tMax) {
    tBest = +inf; firstSurface = null;
    for surface in surfaceList {
        hitSurface, t = surface.intersect(ray, tMin, tBest);
        if hitSurface is not null {
            tBest = t;
            firstSurface = hitSurface;
        }
    }
    return hitSurface,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

  - this is linear in the number of shapes
  - real applications use sublinear methods (acceleration structures) which we will see later

Image so far

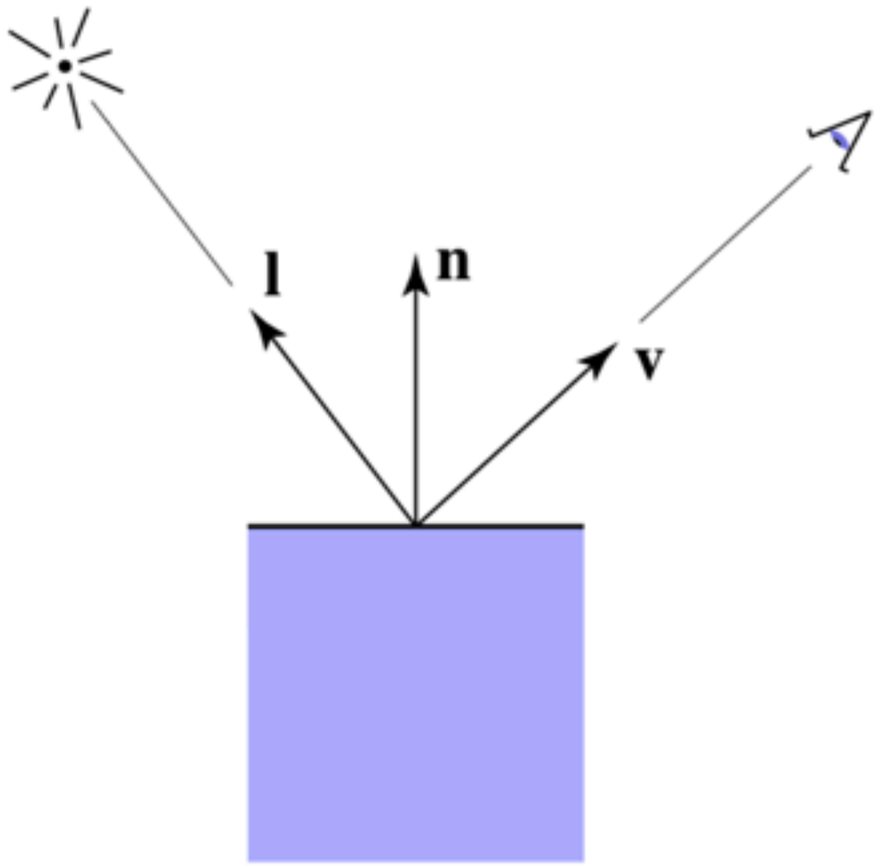
- **With eye ray generation and scene intersection**

```
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, roughness, …)



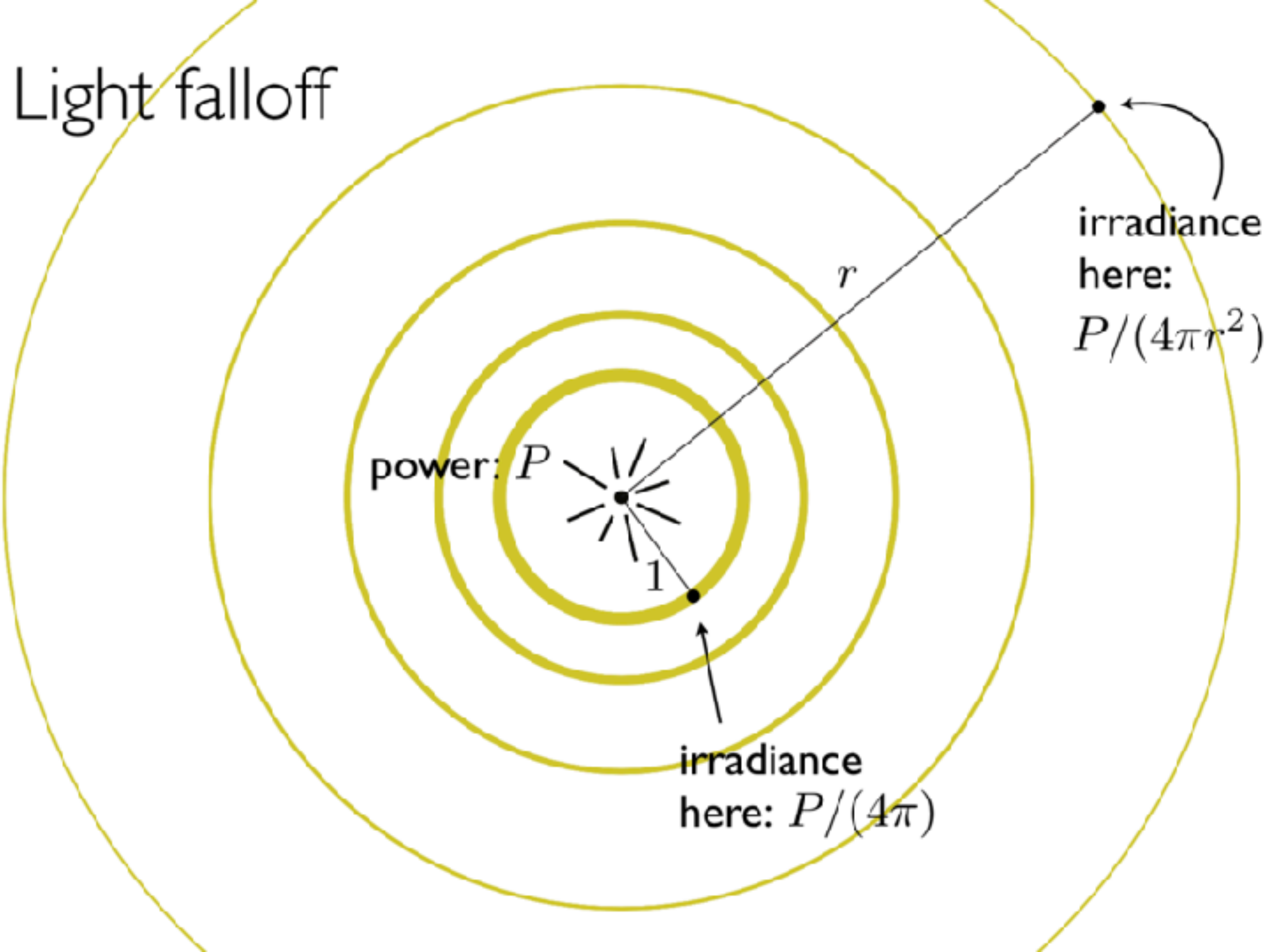
Shading philosophy

- **Goals of shading depend on purpose of image**
  - visualization, CAD: maximize visual clarity
  - visual effects, advertising: maximize resemblance to reality
  - animation, games: somewhere in between
- **Basic starting point: physics of light reflection**
  - a set of useful approximations to real surfaces
  - can remove things for simplicity/clarity
  - can add things for increased accuracy/realism

Light

- **Think of light as a flow of particles through space**
  - disregarding wave nature: polarization, interference, diffraction
  - for now disregarding color: only how much light
- **Sources of light**
  - point sources (a flashlight) ← we will stick to this for now.
  - directional sources (the sun)
  - area sources (a fluorescent tube)
  - environment sources (the sky)

Light falloff



Irradiance from isotropic point source

- **A sphere surrounding the source receives all the power**
- **A small, flat surface of area *A* facing the source receives a fraction (area of surface) / (area of sphere) of that power:**

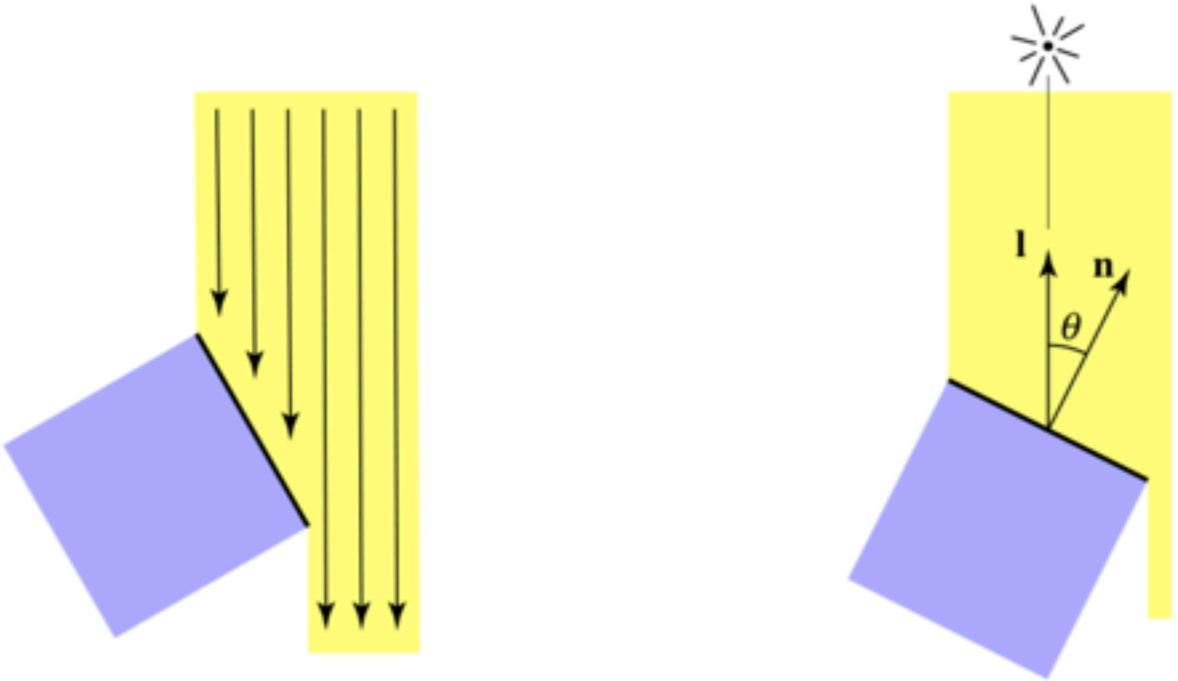
$$P_A = P\frac{A}{4\pi r^2}$$

- **Irradiance is power per unit area:**

$$L = \frac{L_0}{const + lin * d + quad * d^2}$$

$$E = P_A/A = \frac{P}{4\pi r^2} = \underbrace{\frac{P}{4\pi}}_{\text{intensity}}\ \underbrace{\frac{1}{r^2}}_{\text{geometry factor}}$$

Lambert's cosine law



Top face of cube receives a certain amount of light

Top face of 60° rotated cube intercepts half the light

In general, light per unit area is proportional to $\cos\theta = \mathbf{l} \cdot \mathbf{n}$

Irradiance from isotropic point source

- **A surface of area *A* facing at an angle to the source receives a factor of $\cos\theta$ less light:**

$$P_A = P \frac{A\cos\theta}{4\pi r^2}$$

- **Irradiance is power per unit area:**

$$E = P_A / A = \underbrace{\frac{P}{4\pi}}_{\text{intensity}} \underbrace{\frac{\cos\theta}{r^2}}_{\text{geometry factor}}$$

Diffuse reflection

- **Simplest reflection model**
- **Reflected light is independent of view direction**
- **Reflected light is proportional to irradiance**
  - constant of proportionality is the diffuse reflection coefficient

$$L_d = k_d E$$

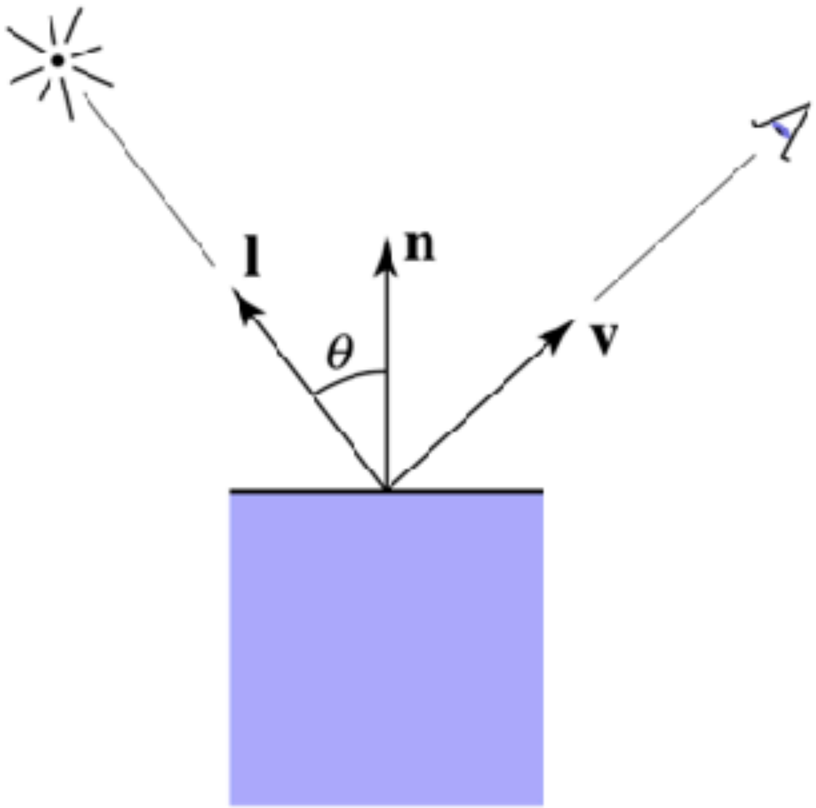
- **More useful to think in terms of reflectance**
  - reflectance is the fraction reflected (between 0 and 1)

$$L_d = \frac{R_d}{\pi} E$$

  - will have to explain the factor of pi later

Lambertian shading

- **Shading independent of view direction**



$$L_d = \underbrace{\frac{R}{\pi}}_{\text{diffuse coefficient}} \overbrace{\frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2}}^{\text{irradiance from source}} I$$

diffuse reflectance

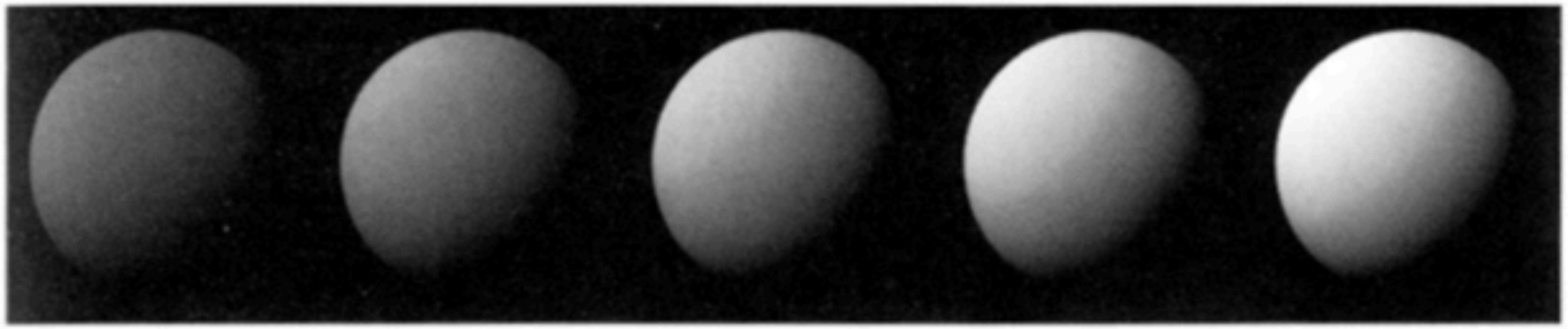
diffusely reflected radiance

distance to source

intensity of source

Lambertian shading

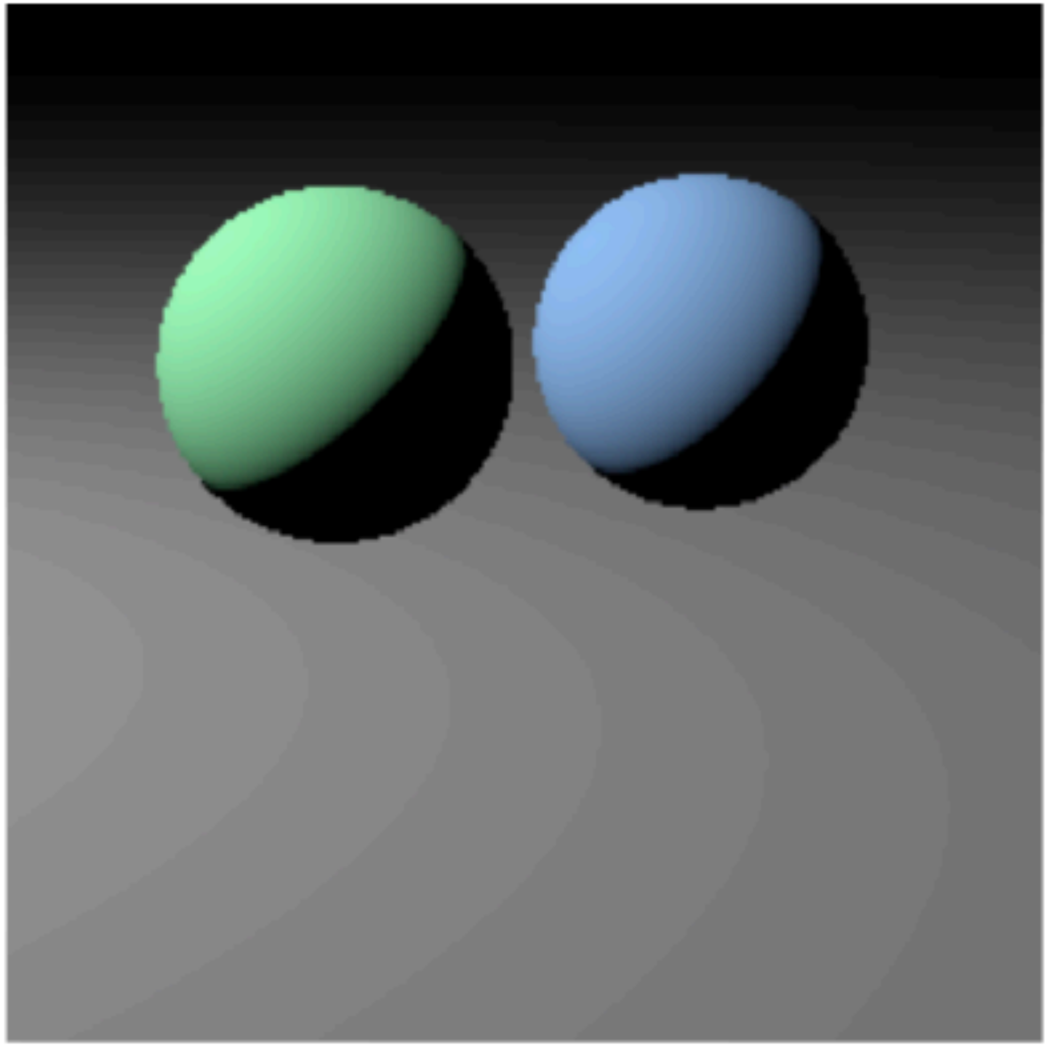
- **Produces matte appearance**



$k_d \longrightarrow$

Image so far – diffuse shading

```
Scene.trace(Ray ray, tMin, tMax) {
    surface, t = hit(ray, tMin, tMax);
    if surface is not null {
        point = ray.evaluate(t);
        normal = surface.getNormal(point);
        return surface.shade(ray, point,
            normal, light);
    }
    else return backgroundColor;
}

...

Surface.shade(ray, point, normal, light) {
    v = −normalize(ray.direction);
    l = normalize(light.pos − point);
    // compute shading
}
```



Shadows

- **Surface is only illuminated if nothing blocks the light**
  - i.e. if the surface can "see" the light
- **With ray tracing it's easy to check**
  - just intersect a ray with the scene!

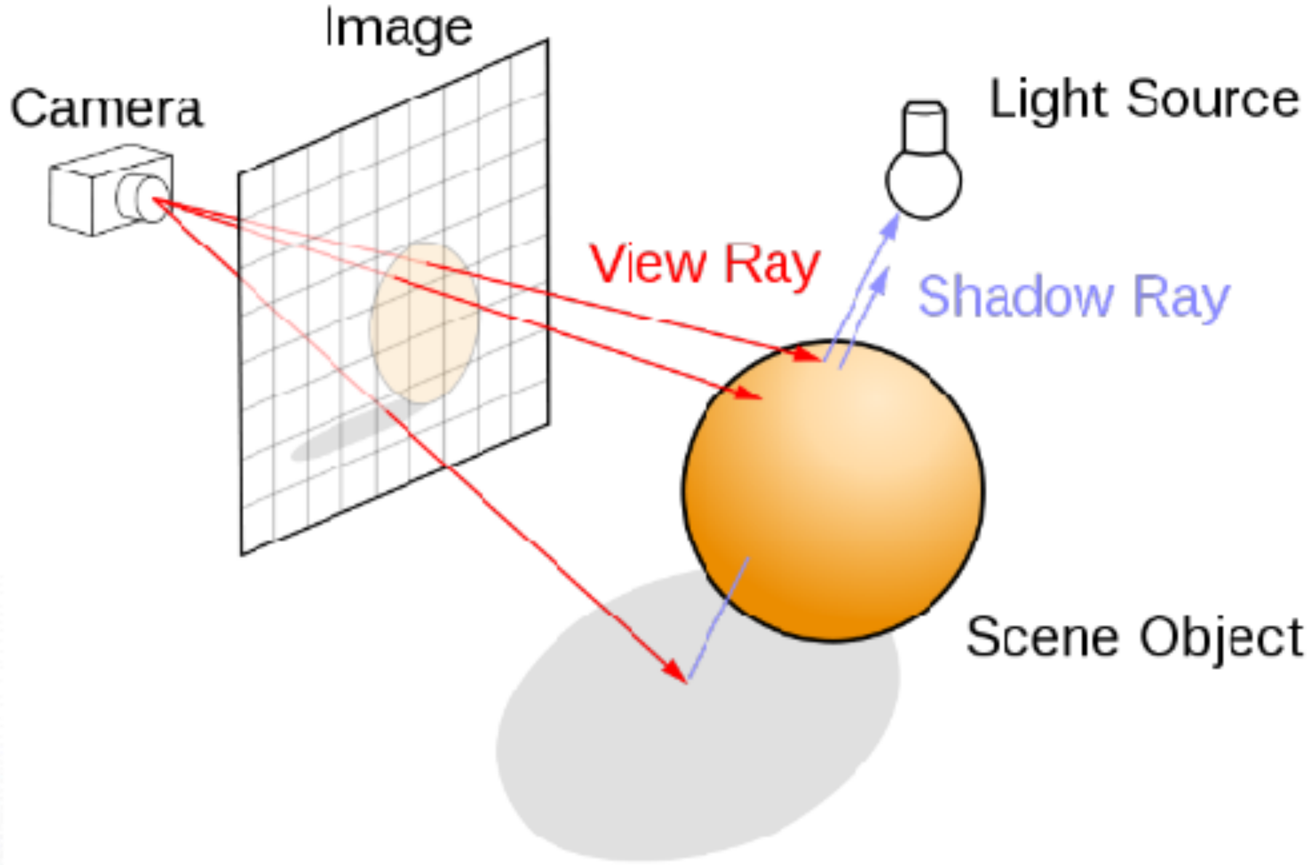
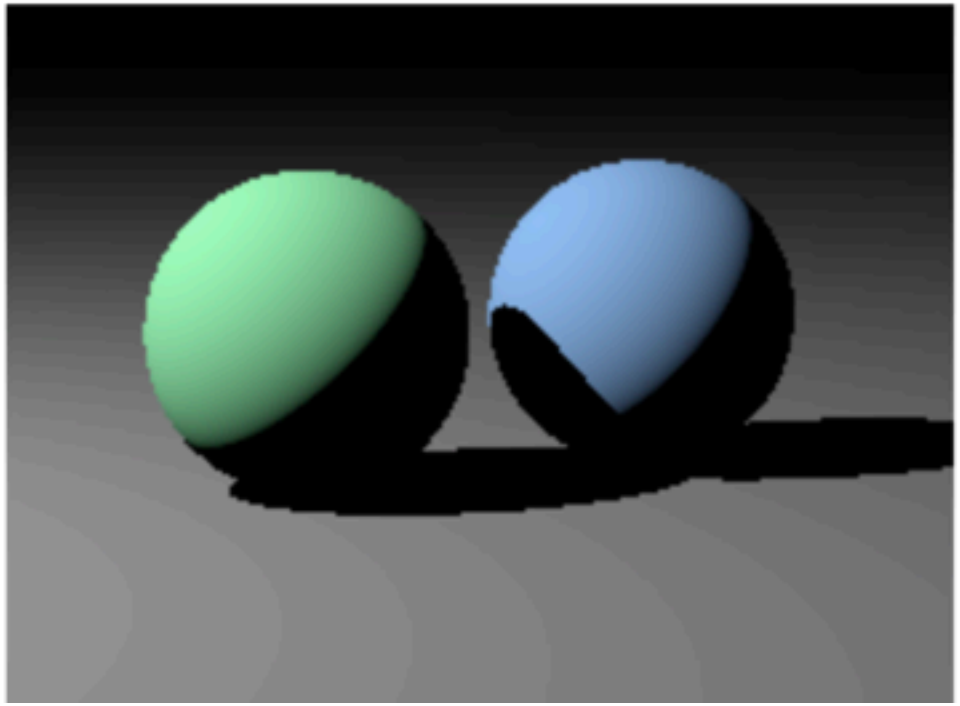


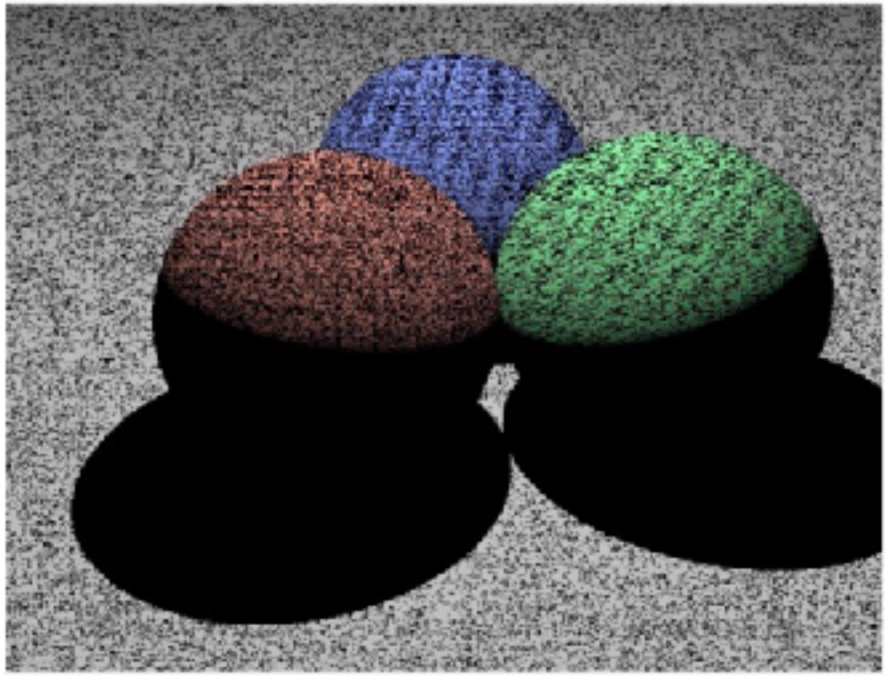
Image so far

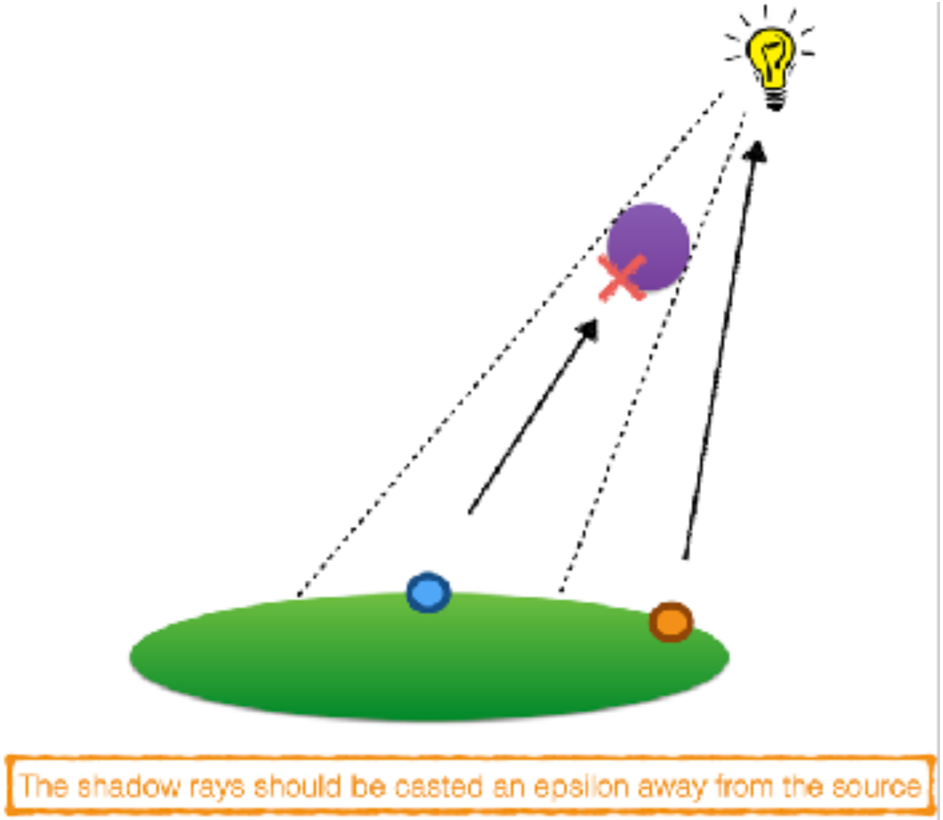
```
Surface.shade(ray, point, normal, light) {
    shadRay = (point, light.pos − point);
    if (shadRay not blocked) {
        v = −normalize(ray.direction);
        l = normalize(light.pos − point);
        // compute shading
    }
    return black;
}
```



Shadow rounding errors

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

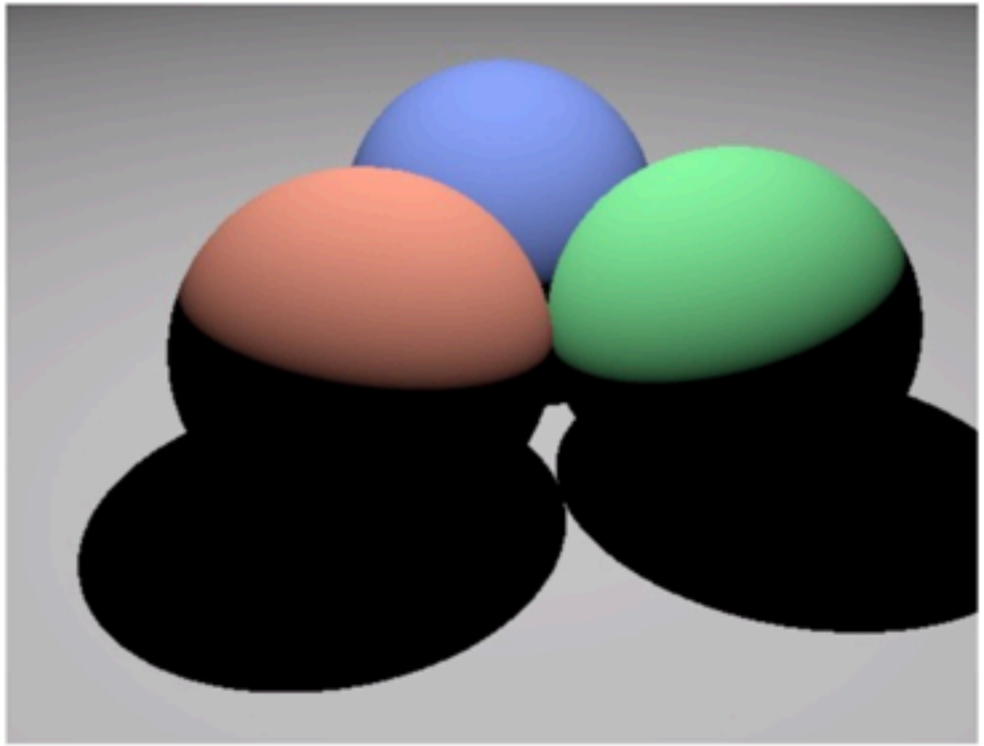




- What’s going one?
  - Hint: at what t does the shadow ray intersect the surface your are shading?

Shadow rounding errors

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



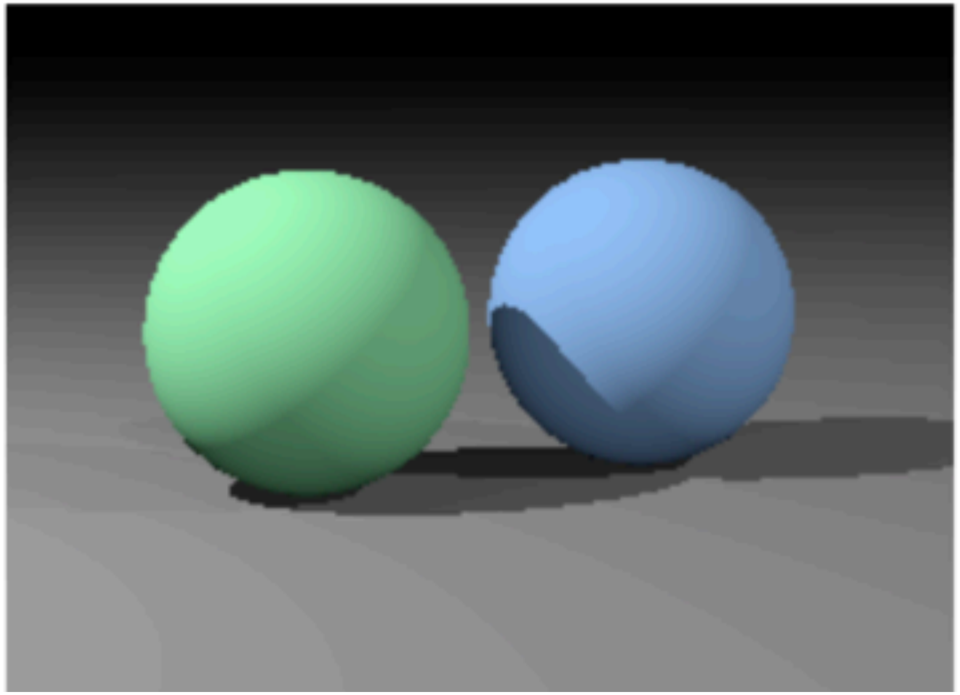
- **Do this by moving the start point, or by limiting the *t* range**

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: add a constant "ambient" color to the shading…

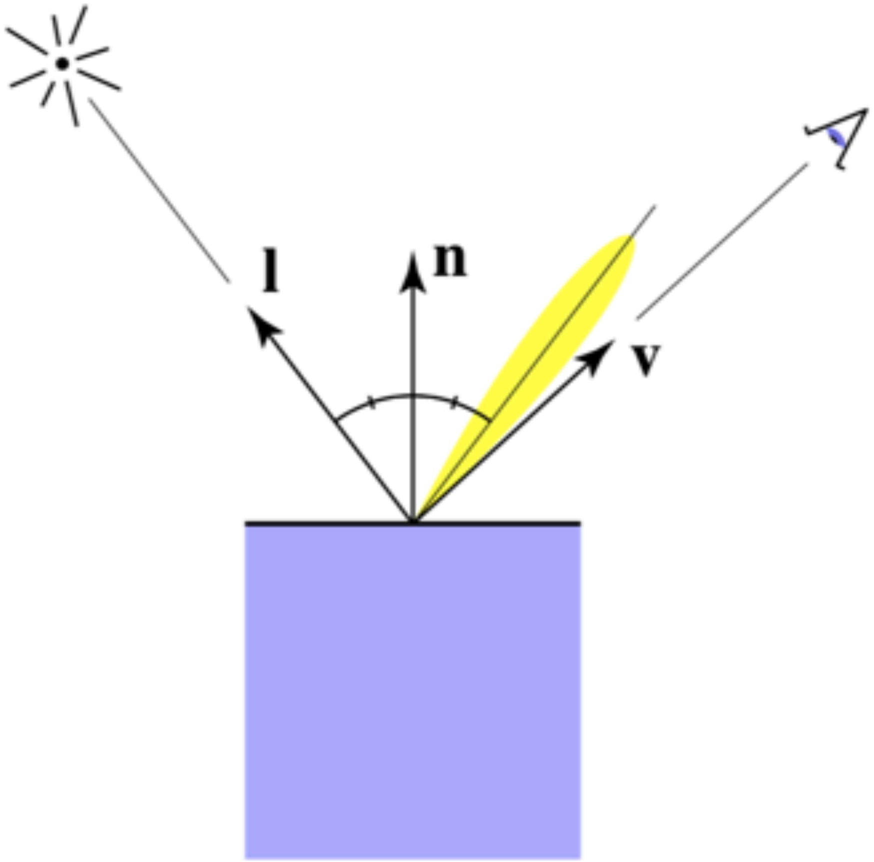
Image so far

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



Specular reflection

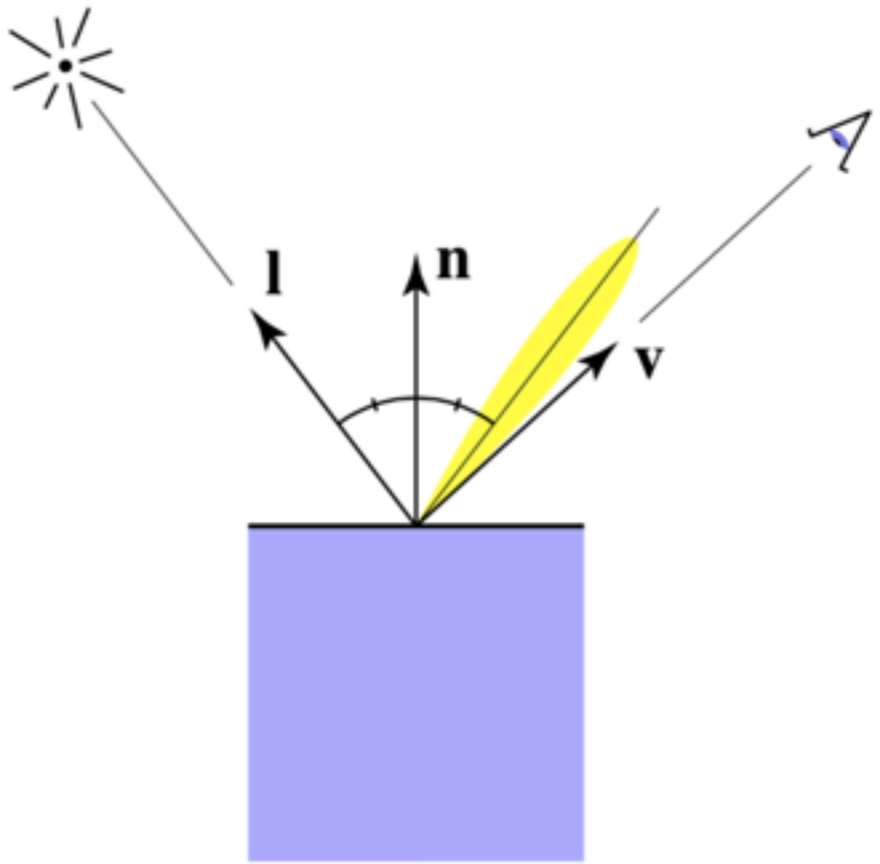
- **Intensity depends on view direction**
  - bright near mirror configuration



Caution: in notes and assignment, $\mathbf{v}$ is called ω_r and $\mathbf{l}$ is called ω_i. No meaningful difference, just notational.

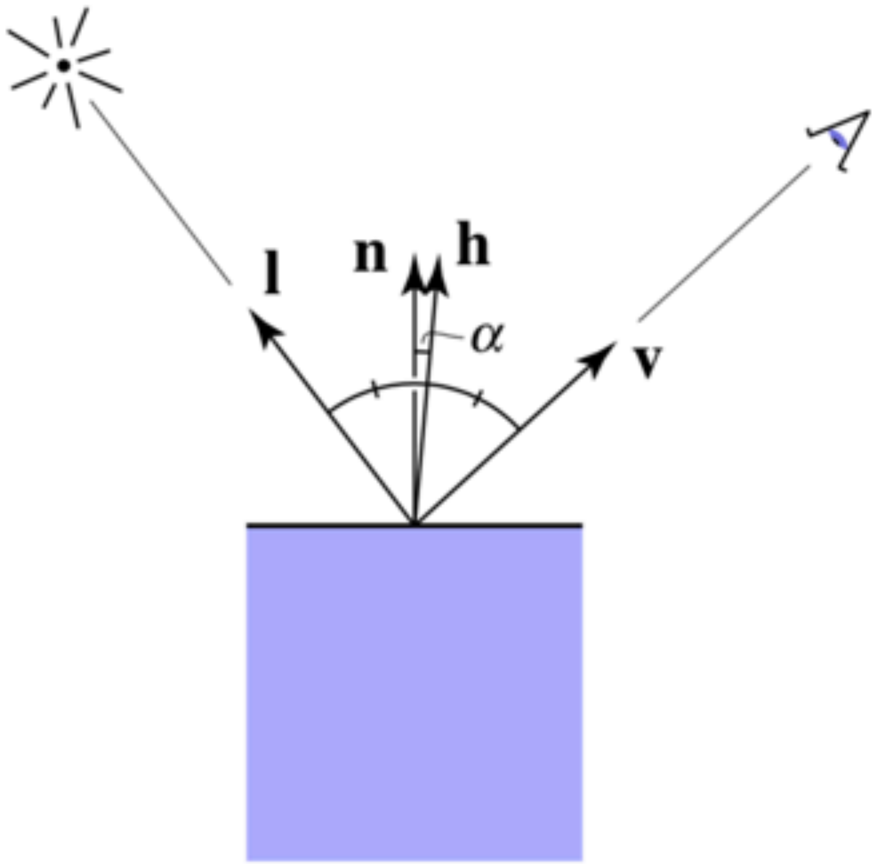
Specular shading (Blinn-Phong)

- **Intensity depends on view direction**
  - bright near mirror configuration



Specular shading (Blinn-Phong)

- **Close to mirror ⇔ half vector near normal**
  - Measure "near" by dot product of unit vectors



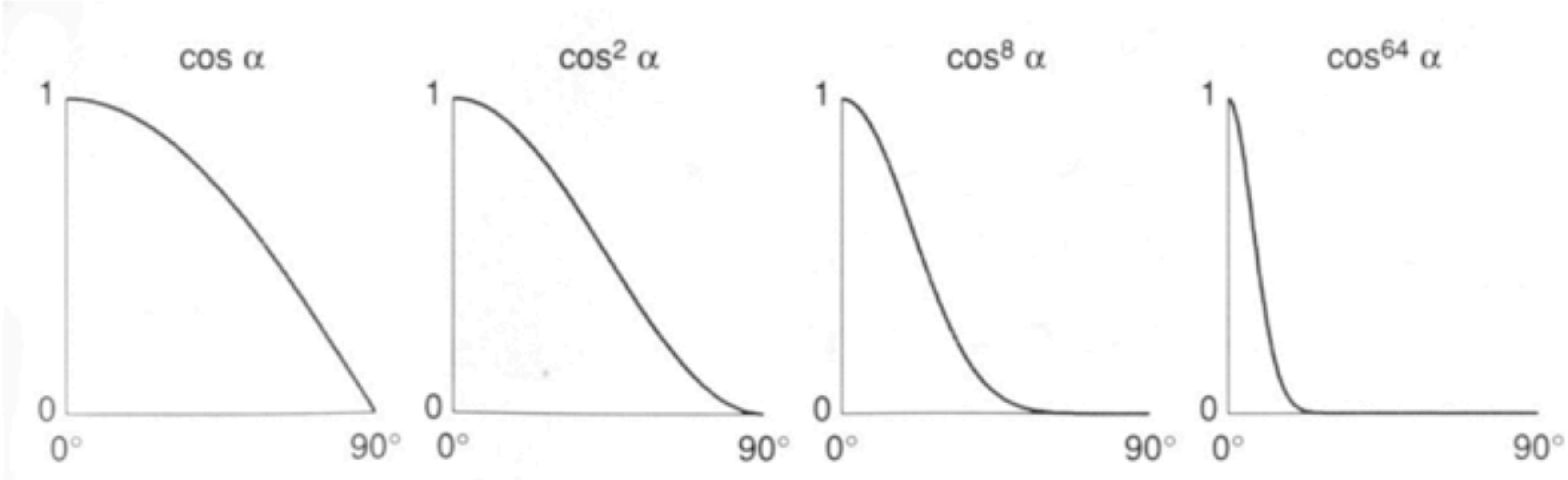
$$\mathbf{h} = \mathrm{bisector}(\mathbf{v}, \mathbf{l})$$
$$= \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

let's work with the expression:

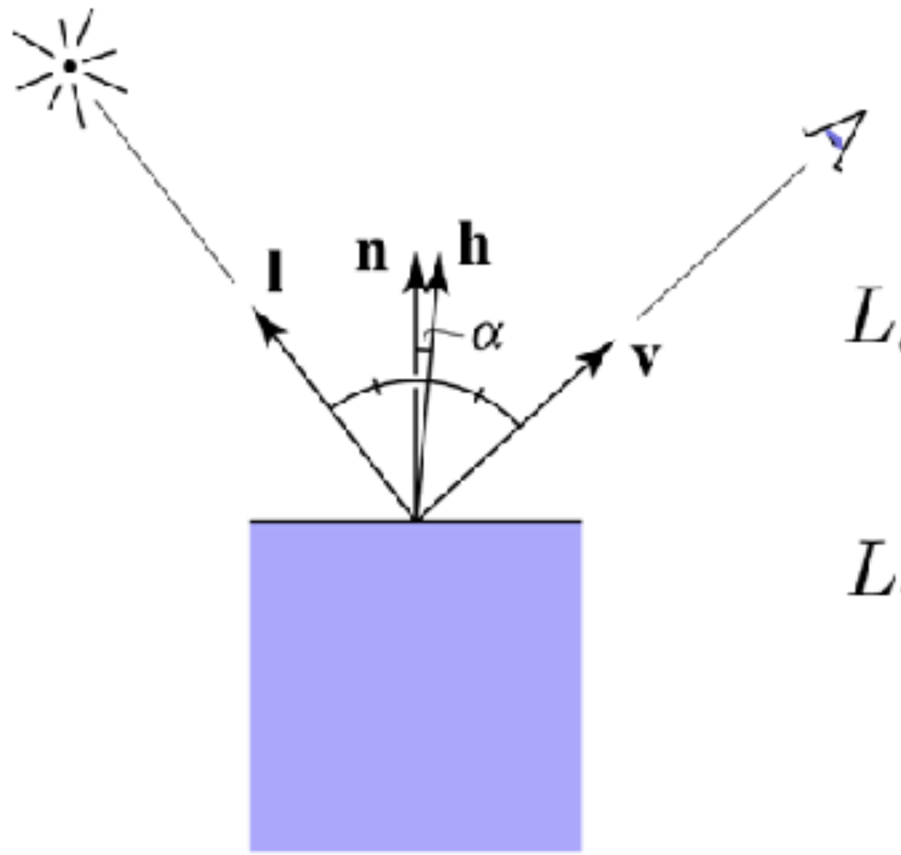
$$(\cos \alpha)^p$$
$$= (\mathbf{n} \cdot \mathbf{h})^p$$

Phong model—plots

- **Increasing *p* narrows the peak**
  - corresponds to increasing "shininess"



Specular shading (Blinn-Phong)



note: this model is officially called "modified Blinn-Phong."

$$L_d = \frac{R}{\pi} \frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2} I$$

$$L_r = \left(\frac{R}{\pi} + k_s (\mathbf{n} \cdot \mathbf{h})^p \right) \frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2} I$$

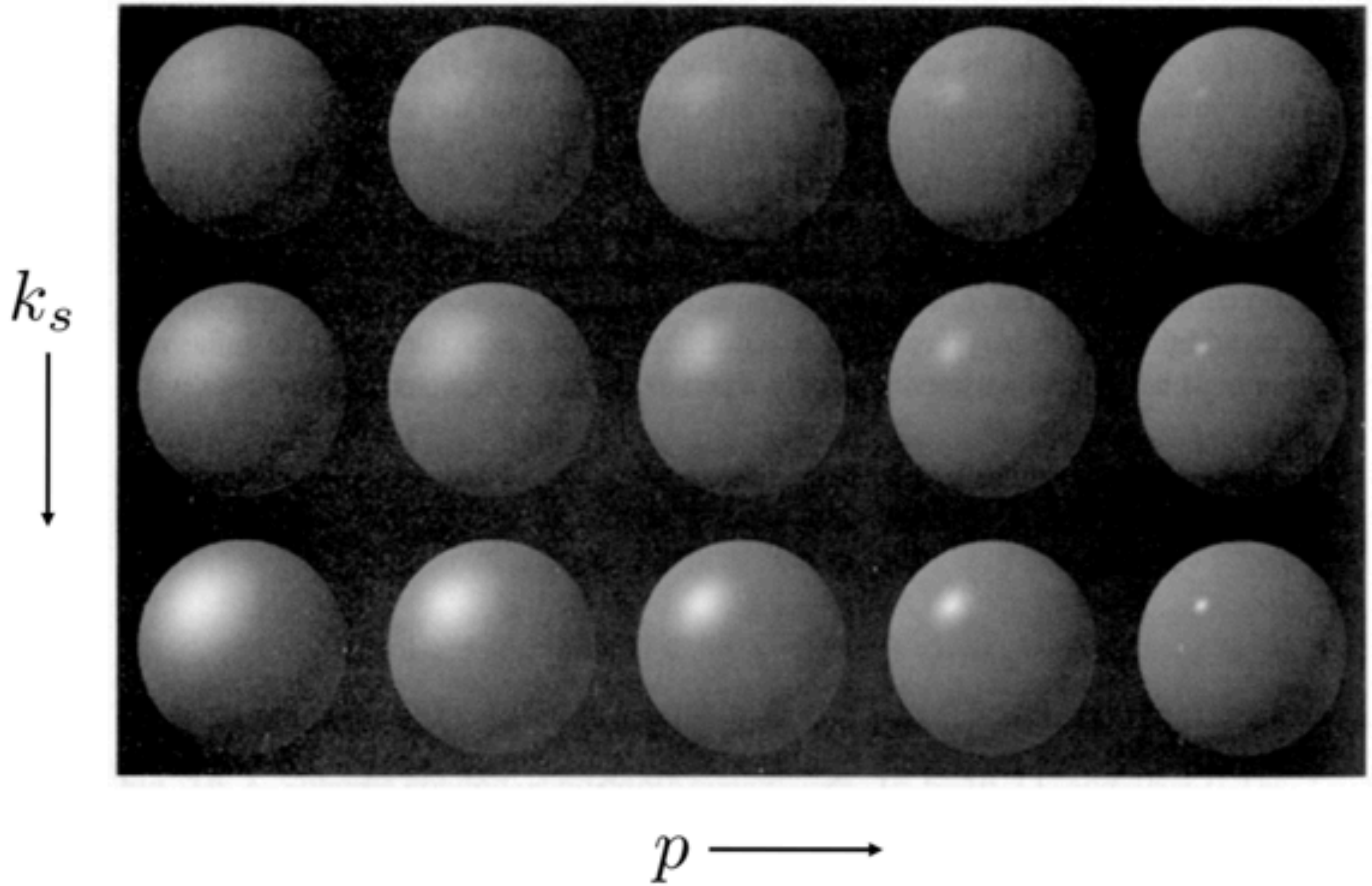
diffuse coefficient

specular coefficient

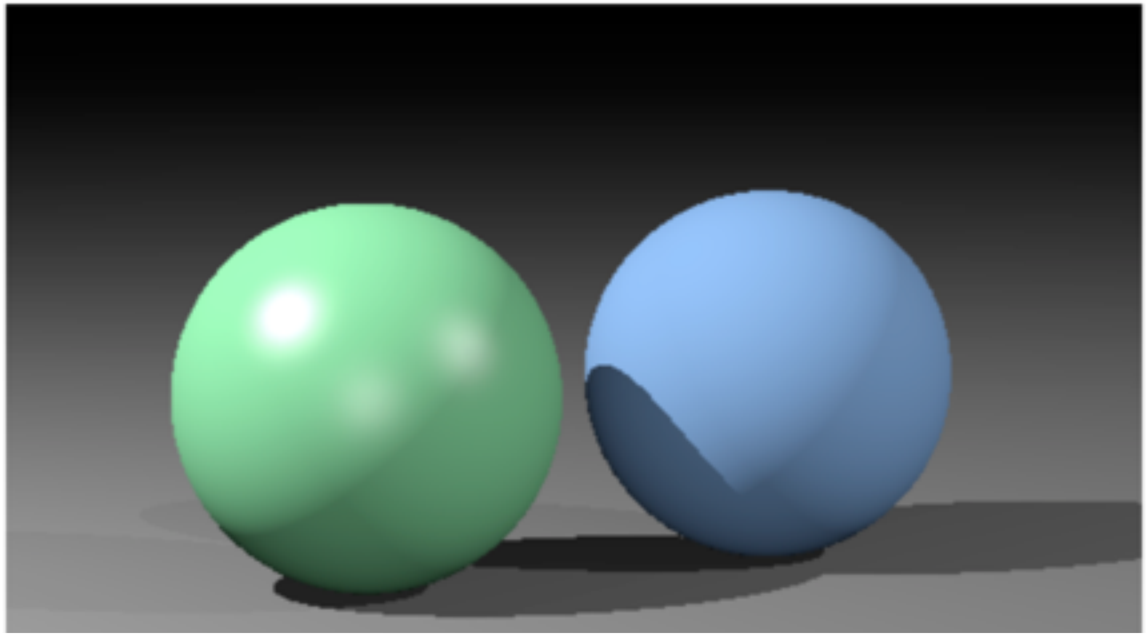
specular term

Specular shading

- **Blinn-Phong**

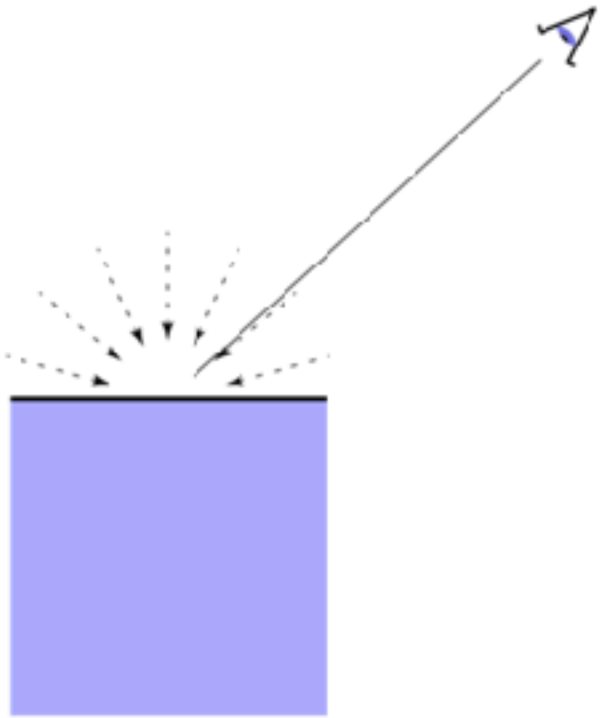


Diffuse + Phong shading



Ambient shading

- **Shading that does not depend on anything**
  - add constant color to account for disregarded illumination and fill in black shadows



$$L_a = k_a\, I_a$$

ambient coefficient

reflected ambient light

Mirror reflection

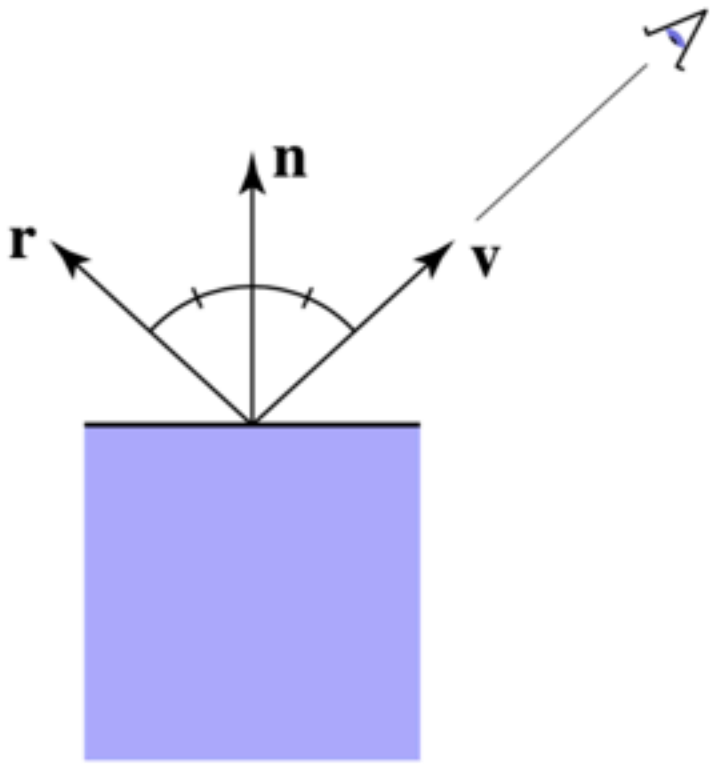
- **Consider perfectly shiny surface**
  - there isn't a highlight
  - instead there's a reflection of other objects
- **Can render this using recursive ray tracing**
  - to find out mirror reflection color, ask what color is seen from surface point in reflection direction
  - already computing reflection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

$$L = L_a + L_r + L_m$$

  - where L_m is evaluated by tracing a new ray

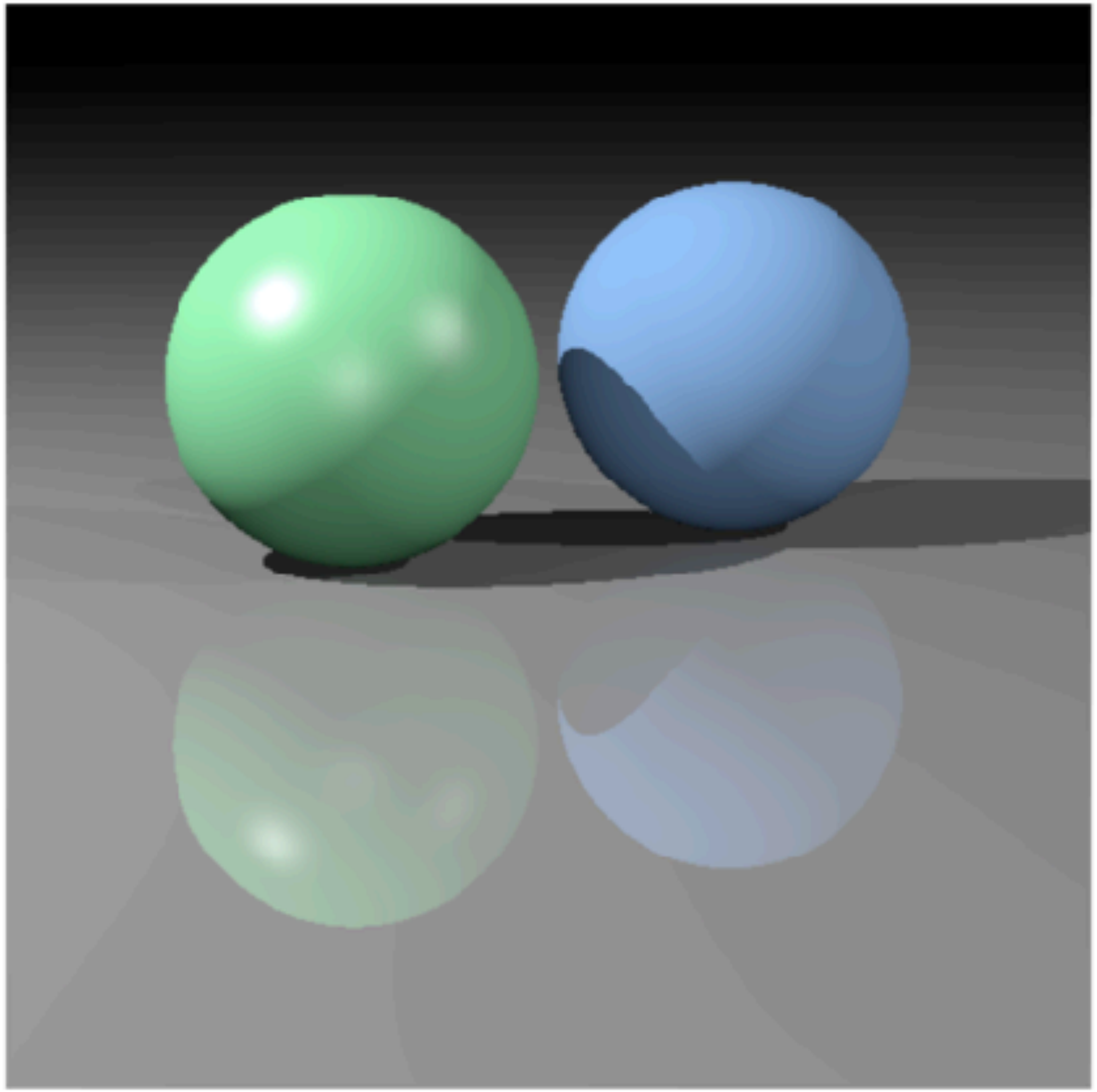
Mirror reflection

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



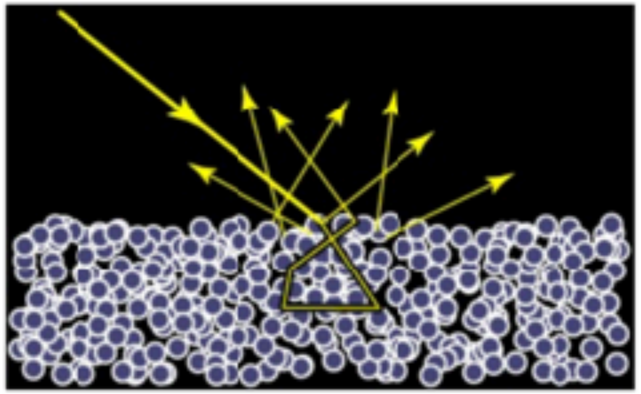
$$\mathbf{r} = \mathbf{v} + 2((\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v})$$
$$= 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}$$

Diffuse + mirror reflection (glazed)

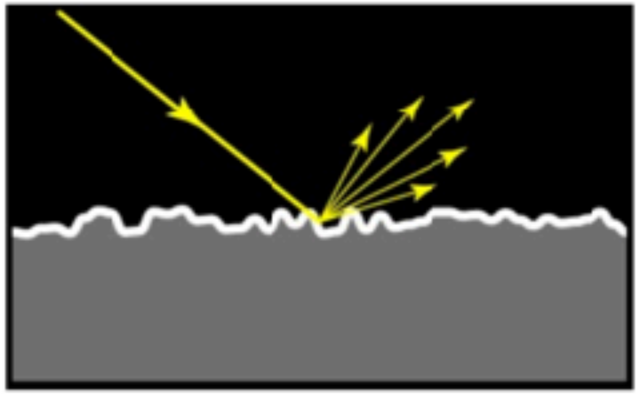


(glazed material on floor)

Specular shading



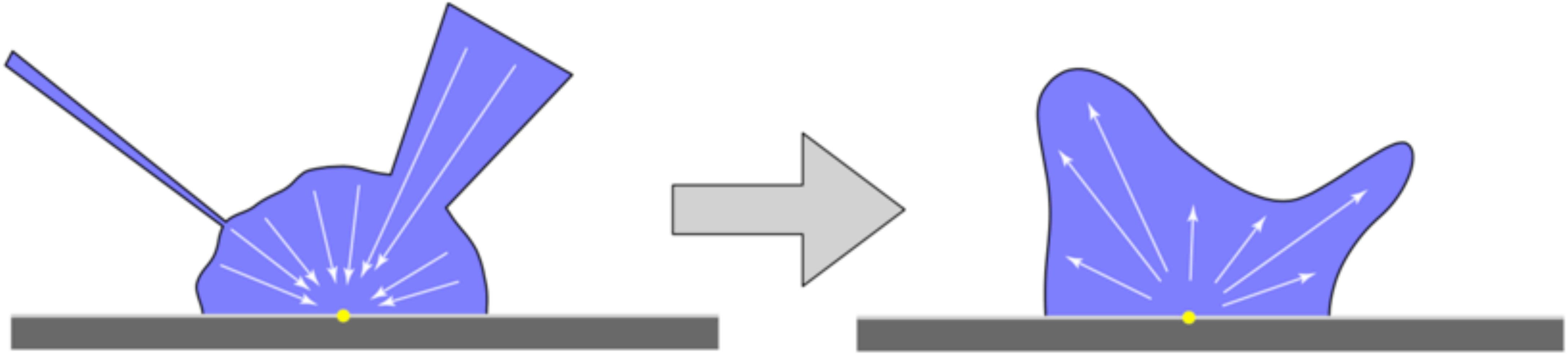
diffuse



specular

Light reflection: full picture

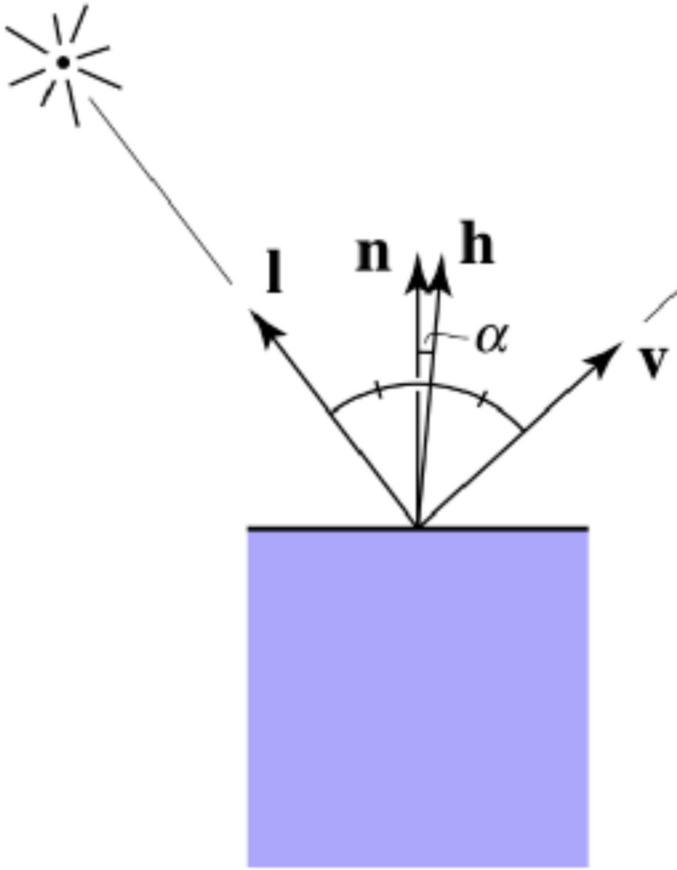
- **when writing a shader, think like a bug standing on the surface**
  - bug sees an incident distribution of light arriving at the surface
  - physics question: what is the outgoing distribution of light?



incident distribution
(function of direction)

reflected distribution
(function of direction)

General shading by bidirectional reflectance distribution function (BRDF)



$$L_s = f_r(\mathbf{n}, \mathbf{l}, \mathbf{v}) \overbrace{\frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2} I}^{\text{irradiance from source}}$$

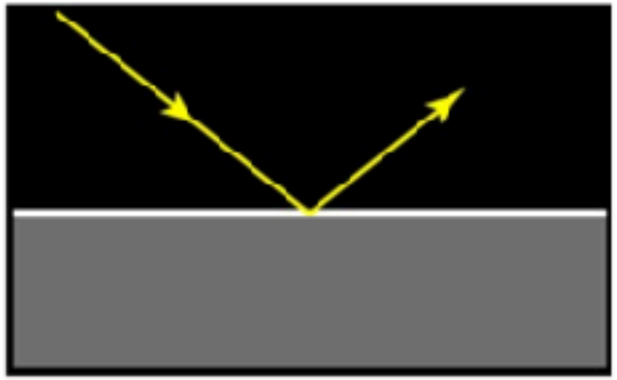
specularly reflected radiance

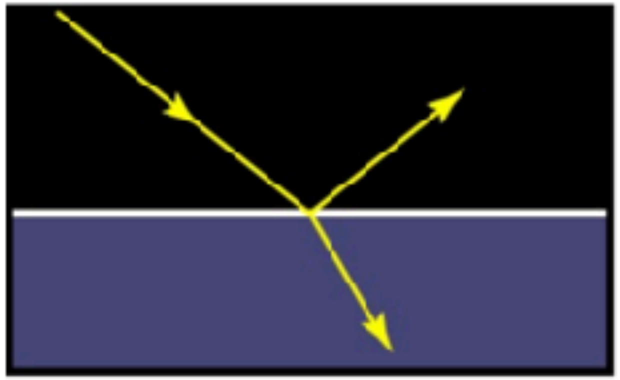
specular BRDF value

Smooth surfaces

metal

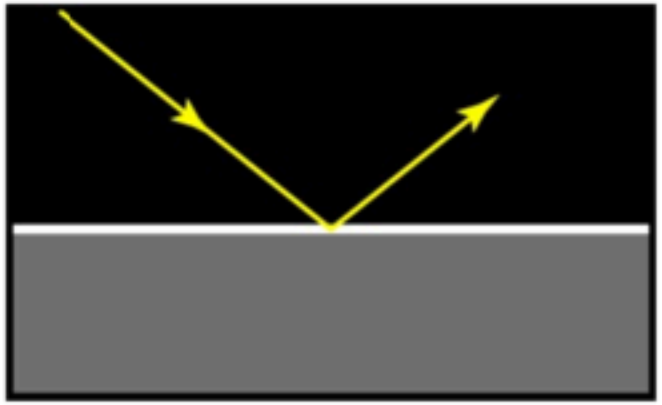
dielectric



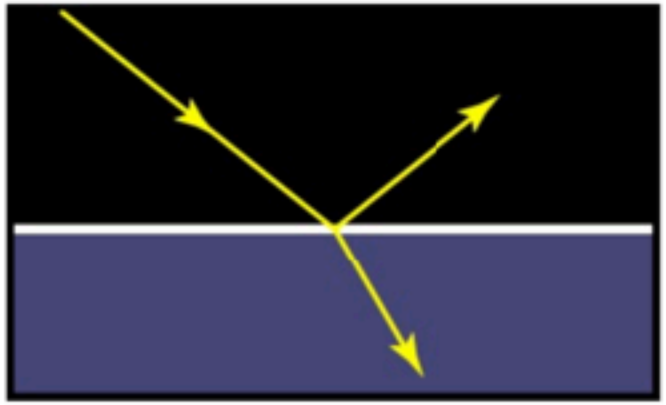


Ideal specular reflection

- **Smooth surfaces of pure materials have ideal specular reflection**
  - Metals (conductors) and dielectrics (insulators) behave differently
- **Reflectance (fraction of light reflected) depends on angle**

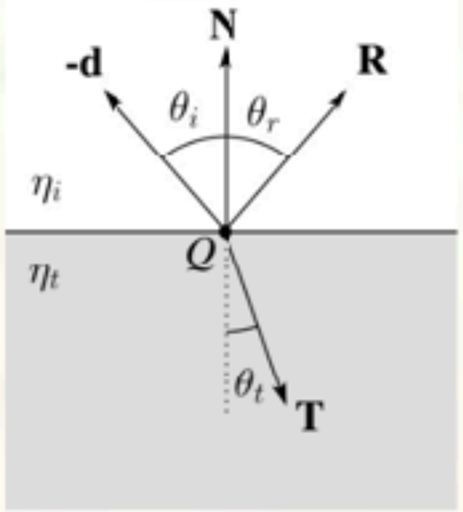


metal



dielectric

Reflection and transmission



Index of refraction is speed of light, relative to speed of light in vacuum = c/v, c is speed in vacuum

Vacuum: 1.0
Air: 1.000277
Water: 1.33
Glass: 1.49

- Law of reflection:

$$\theta_i = \theta_r$$

- Snell's law of refraction:

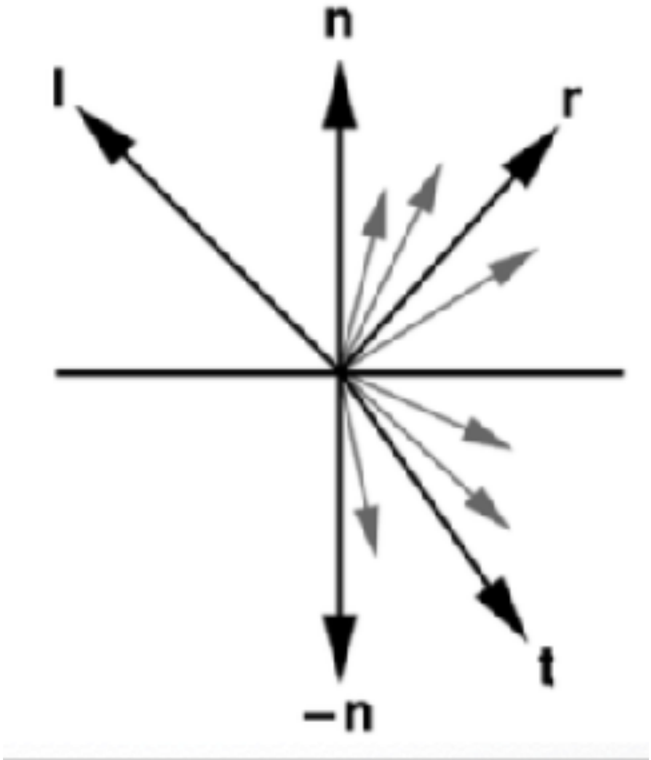
$$\eta_\mathrm{i} \sin\theta_\mathrm{I} = \eta_\mathrm{t} \sin \theta_\mathrm{t}$$

- where η_i , η_t are **indices of refraction**.

Translucency

- Most real objects are not transparent, but blur the background image

- Scatter light on other side of surface



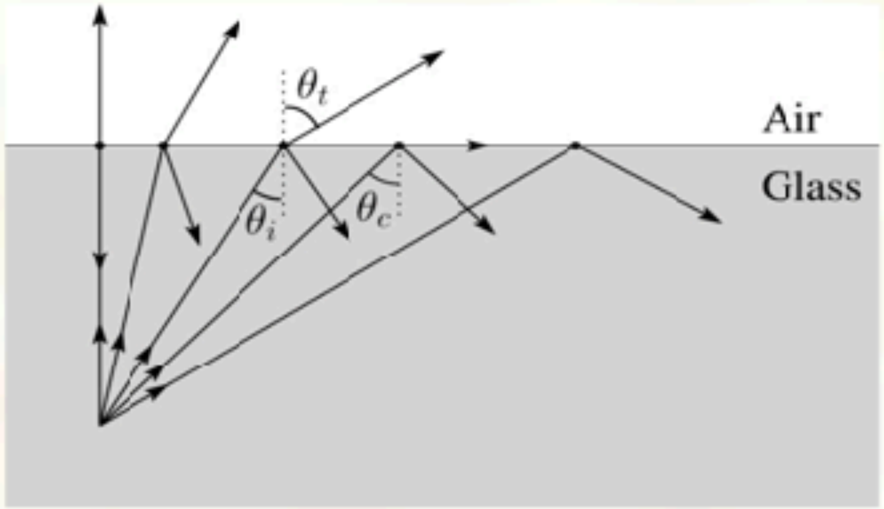
-

- Use stochastic sampling (called distributed ray tracing)

Transmission + Translucency Example

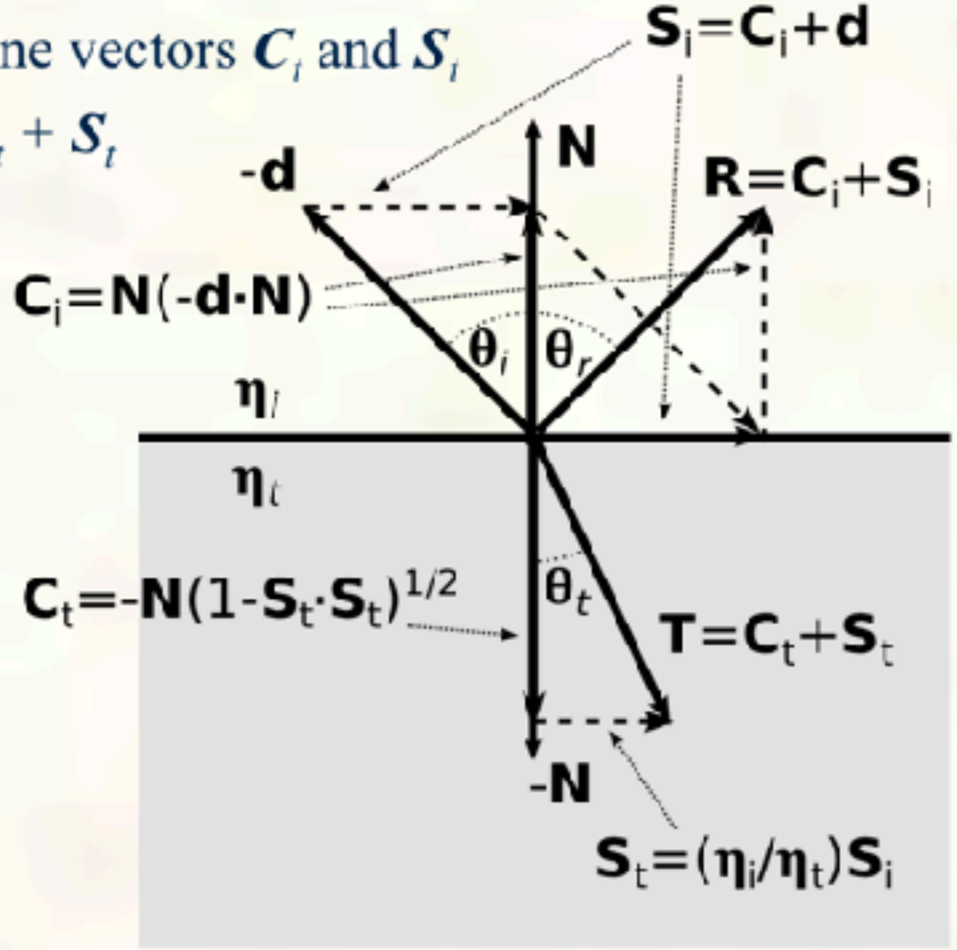
Total Internal Reflection

- The equation for the angle of refraction can be computed from Snell's law:
- What happens when $\eta_i > \eta_t$?
- When θ_t is exactly 90°, we say that θ_I has achieved the "critical angle" θ_c.
- For $\theta_I > \theta_c$, *no rays are transmitted*, and only reflection occurs, a phenomenon known as "total internal reflection" or TIR.



Reflected and transmitted rays

- For incoming ray $P(t)=P+t\boldsymbol{d}$
  - Compute input cosine and sine vectors $\boldsymbol{C}_i$ and $\boldsymbol{S}_i$
  - Reflected ray vector $\boldsymbol{R} = \boldsymbol{C}_i + \boldsymbol{S}_i$
  - Compute output cosine and sine vectors $\boldsymbol{C}_t$ and $\boldsymbol{S}_t$
  - Transmitted ray vector $\boldsymbol{T} = \boldsymbol{C}_t + \boldsymbol{S}_t$



Recursive Shading Model

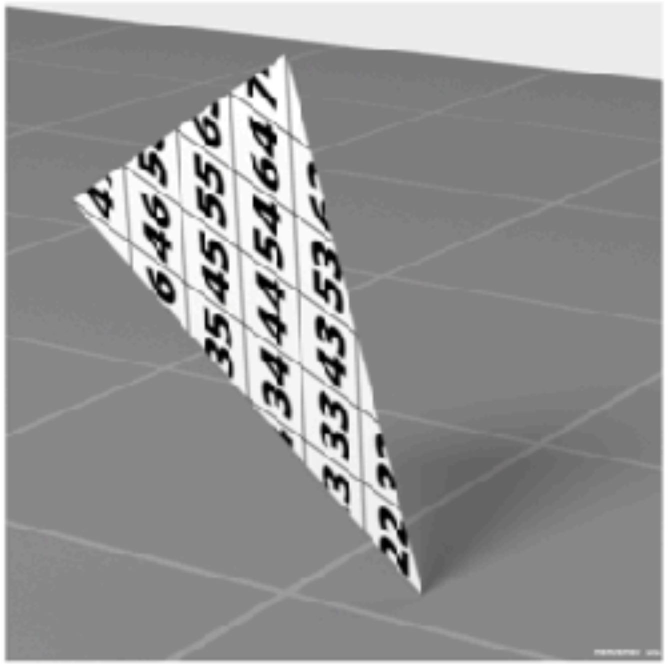
$$L_r = \left(\frac{R}{\pi} + k_s (\mathbf{n} \cdot \mathbf{h})^p \right) \frac{\max(0, \mathbf{n} \cdot \mathbf{l})}{r^2} I$$

- Global ambient term, emission from material
- For each light, diffuse specular terms
- Highlighted terms are **recursive specularities [mirror reflections] and transmission** (latter is extra)
- Trace secondary rays for mirror reflections and refractions, include contribution in lighting model

l = unit vector to light; v = l reflected about n; n = surface normal; v = vector to viewer

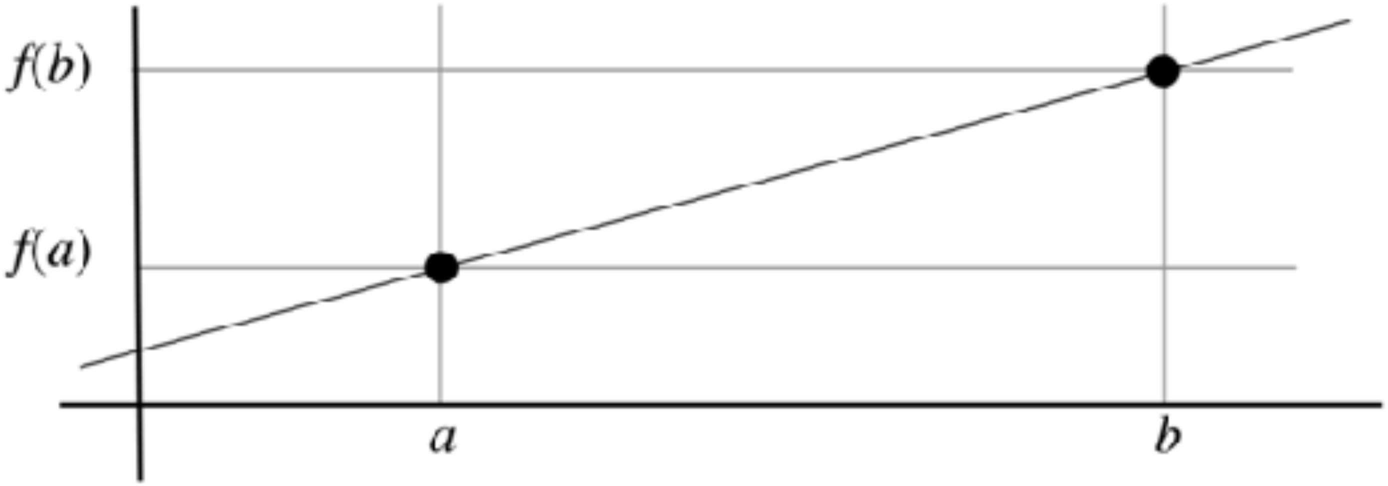
Texture coordinates on meshes

- **Texture coordinates are per-vertex data like vertex positions**
  - can think of them as a second position: each vertex has a position in 3D space and in 2D texture space
- **How to come up with (*u*,*v*)s for points inside triangles?**



Linear interpolation, 1D domain

- **Given values of a function f(x) for two values of x, you can define in-between values by drawing a line**



See textbook Sec. 2.6

  - there is a unique line through the two points
  - can write down using slopes, intercepts
  - …or as a value added to f(a)
  - …or as a convex combination of f(a) and f(b)

$$
\begin{aligned}
f(x) &= f(a) + \frac{x-a}{b-a}(f(b) - f(a)) \\
&= (1-\beta)f(a) + \beta f(b) \\
&= \alpha f(a) + \beta f(b)
\end{aligned}
$$

Linear interpolation in 1D

- **Alternate story**

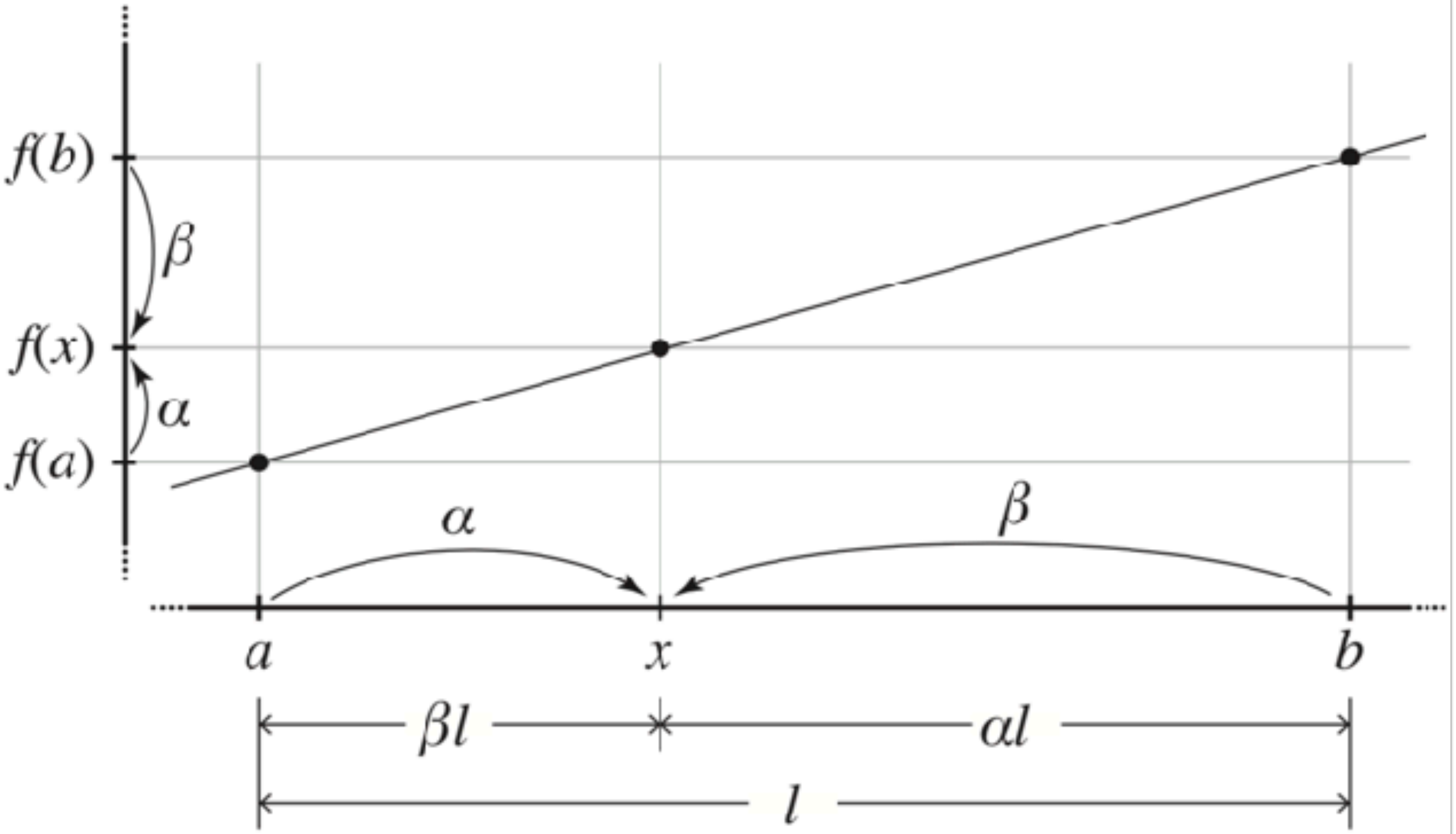
1. write *x* as convex combination of a and b

$$x = \alpha a + \beta b \quad \text{where } \alpha + \beta = 1$$

2. use the same weights to compute ***f***(***x***) as a convex combination of ***f***(***a***) and ***f***(***b***)

$$f(x) = \alpha f(a) + \beta f(b)$$

Linear interpolation in 1D



Linear interpolation in 2D

- **Use the alternate story:**

1. Write **x**, the point where you want a value, as a convex linear combination of the vertices

$$\mathbf{x} = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \quad \text{where } \alpha + \beta + \gamma = 1$$

2. Use the same weights to compute the interpolated value $f(\mathbf{x})$ from the values at the vertices, $f(\mathbf{a})$, $f(\mathbf{b})$, and $f(\mathbf{c})$

$$f(\mathbf{x}) = \alpha f(\mathbf{a}) + \beta f(\mathbf{b}) + \gamma f(\mathbf{c})$$

See textbook Sec. 2.7

Interpolation in ray tracing

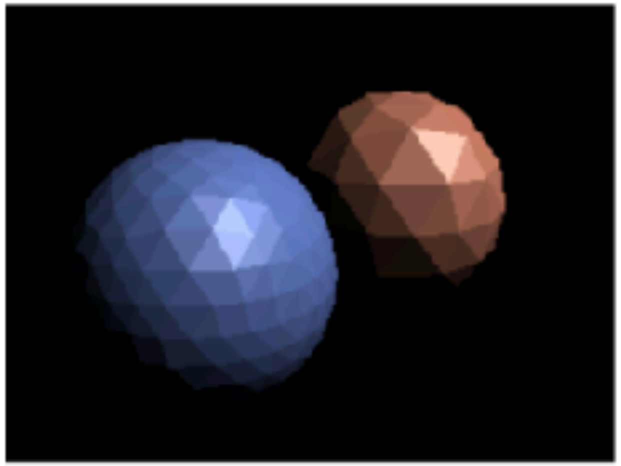
- **When values are stored at vertices, use linear (barycentric) interpolation to define values across the whole surface that:**
  1. …match the values at the vertices
  2. …are continuous across edges
  3. …are piecewise linear (linear over each triangle)

     as a function of 3D position, not screen position—more later

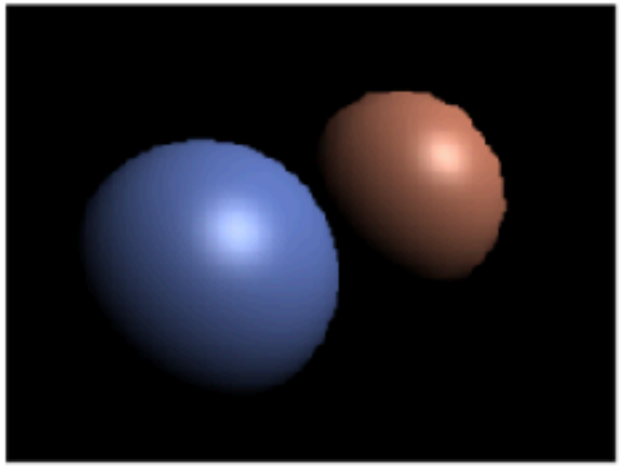
- **How to compute interpolated values**
  4. during triangle intersection compute barycentric coords
  5. use barycentric coords to average attributes given at vertices

What to interpolate?

- **Texture coordinates**
  - without interpolating there can't really be textures
- **Surface normals**
  - for smooth surfaces approximated with meshes
  - use interpolated normal for shading in place of actual normal
  - "shading normal" vs. "geometric normal"



geometric normals



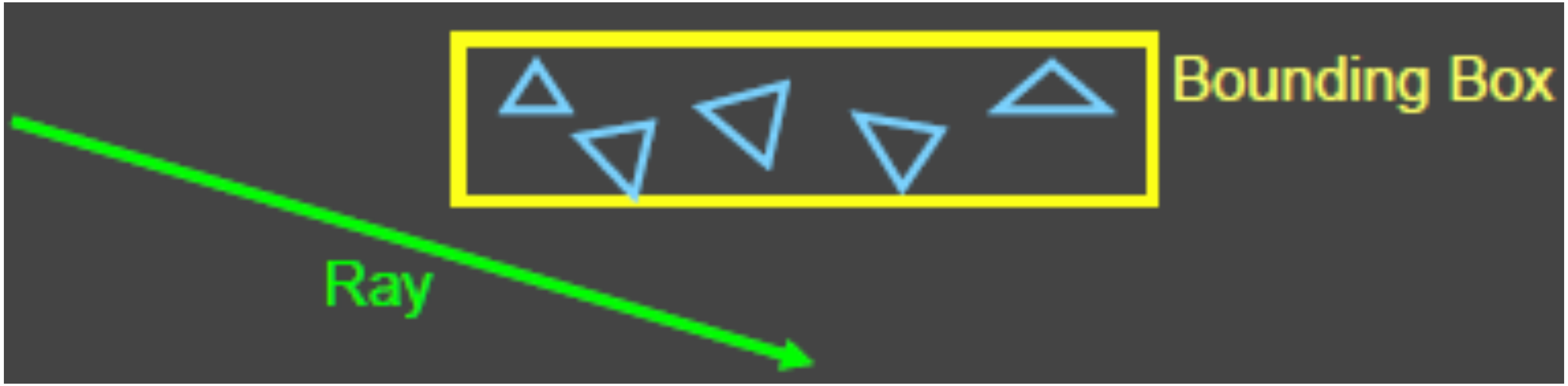
interpolated normals

Acceleration

- Testing each object for each ray is slow
  - Fewer Rays
    - Adaptive sampling, depth control
  - Generalized Rays
    - Beam tracing, cone tracing, pencil tracing etc.
  - Faster Intersections (more on this later)
    - Optimized Ray-Object Intersections
    - ***Fewer Intersections***

Acceleration Structures

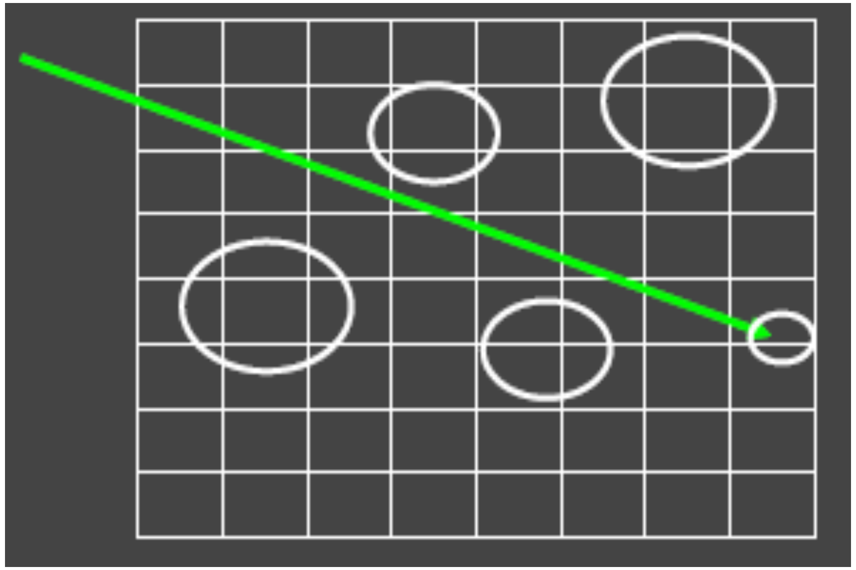
- Bounding boxes (possibly hierarchical)
  - If no intersection bounding box, needn't check objects



- Spatial Hierarchies (Oct-trees, kd trees, BSP trees)

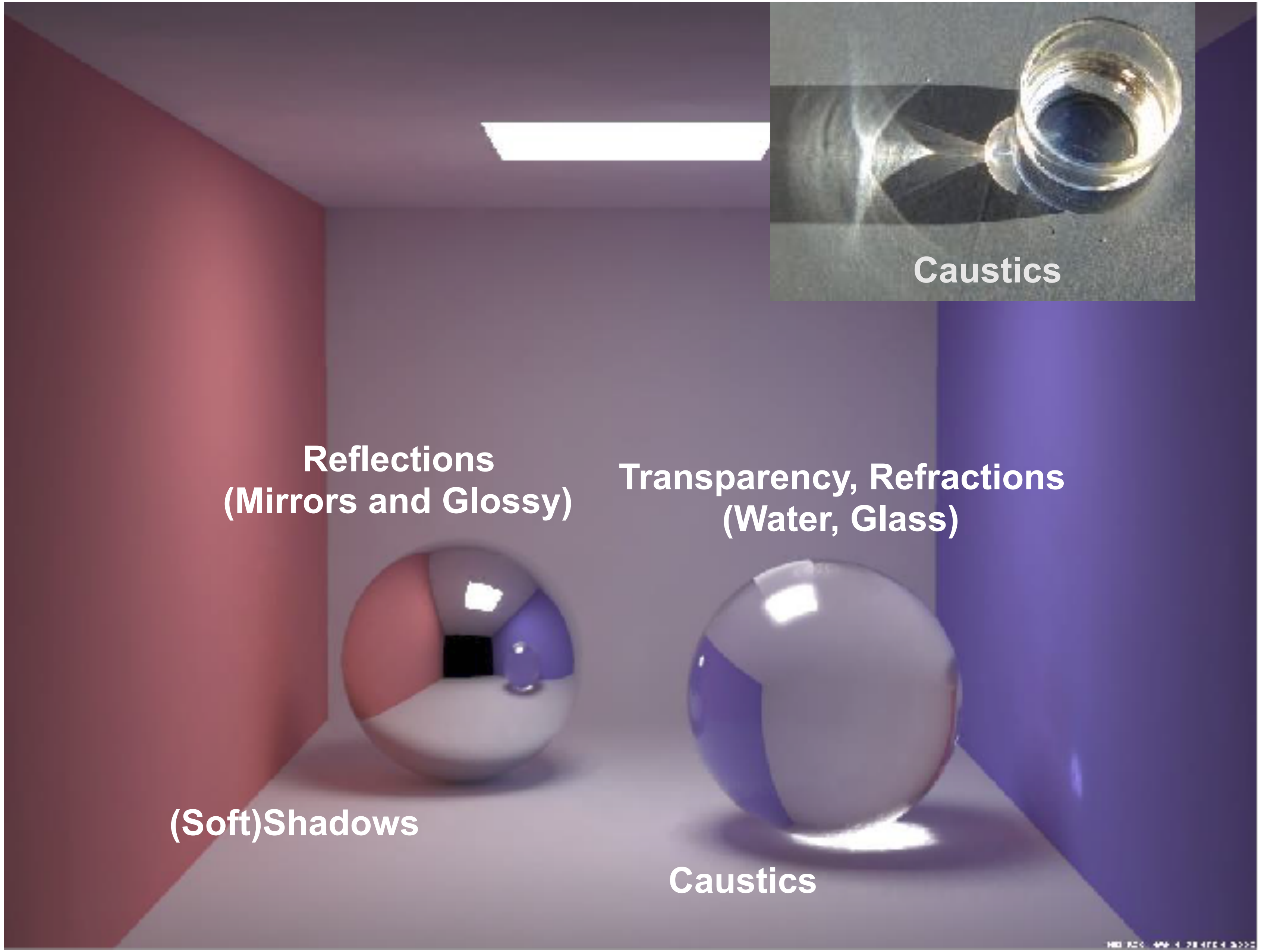
Acceleration and Regular Grids

- Simplest acceleration, for example 5x5x5 grid
- For each grid cell, store overlapping triangles
- March ray along grid (need to be careful with this), test against each triangle in grid cell

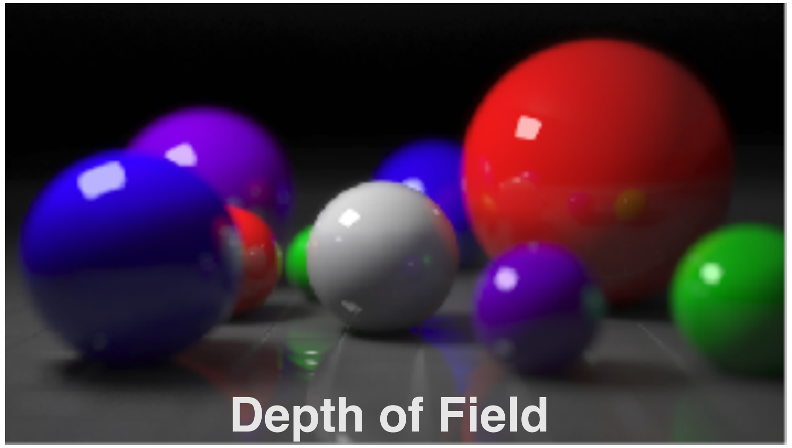


- More sophisticated: kd-tree, oct-tree bsp-tree
- Or use (hierarchical) bounding boxes

Motivation: Effects needed for Realism



Inter reflections (Color Bleeding)



Motivation: Effects needed for Realism

- (Soft) Shadows
- Reflections (Mirrors and Glossy)
- Transparency (Water, Glass)
- Inter reflections (Color Bleeding)
- Complex Illumination (Natural, Area Light)
- Realistic Materials (Velvet, Paints, Glass)
- …

References

- *Daniele Panozzo*