Topics

1. Introduction: What is Computer Graphics?
2. Raster Images (image input/output devices and representation)
3. Scan conversion (pixels, lines, triangles)
4. Ray Casting (camera, visibility, normals, lighting, Phong illumination)
5. Ray Tracing (shadows, supersampling, global illumination)
6. Spatial Data Structures (AABB trees, OBB, bounding spheres, octree)
7. Meshes (connectivity, smooth interpolation, uv-textures, subdivision, Laplacian smoothing)
8. 2D/3D Transformations (Translate, Rotate, Scale, Affine, Homography, Homogeneous coordinates)
9. Viewing and Projection (matrix composition, perspective, Z-buffer)
10. Shader Pipeline (Graphics Processing Unit)
11. Animation (kinematics, keyframing, Catmull-Romm interpolation, physical simulation)
12. 3D curves and objects (Hermite, Bezier, cubic curves, curve continuity, extrusion/revolve surfaces)
13. Advanced topics overview

Topic 4.

Ray Casting

*Adapted from slides by Steve Marschner

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

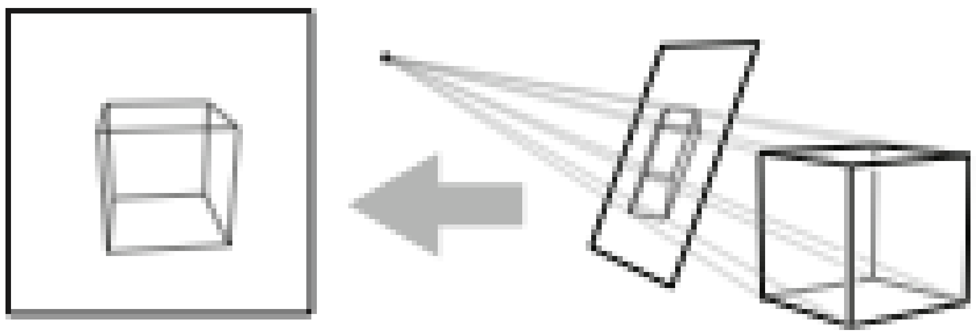
```
for each pixel in the image {
    for each object in the scene {
        if (object affects pixel) {
            do something
        }
    }
}
```

We will do this first

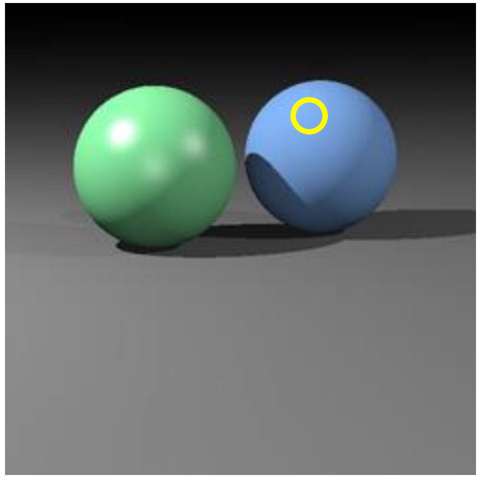
image order
or
ray tracing

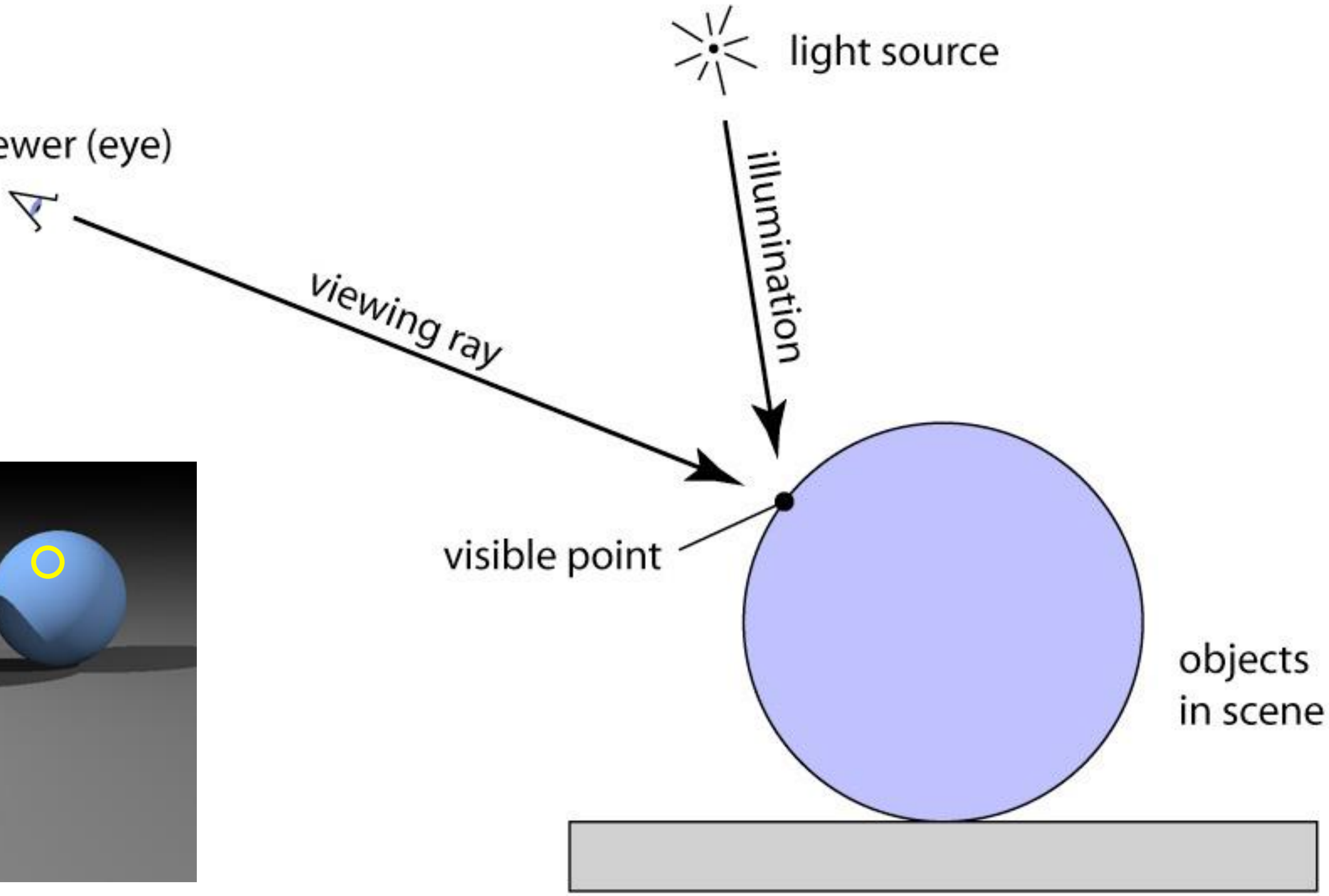
Ray tracing idea

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

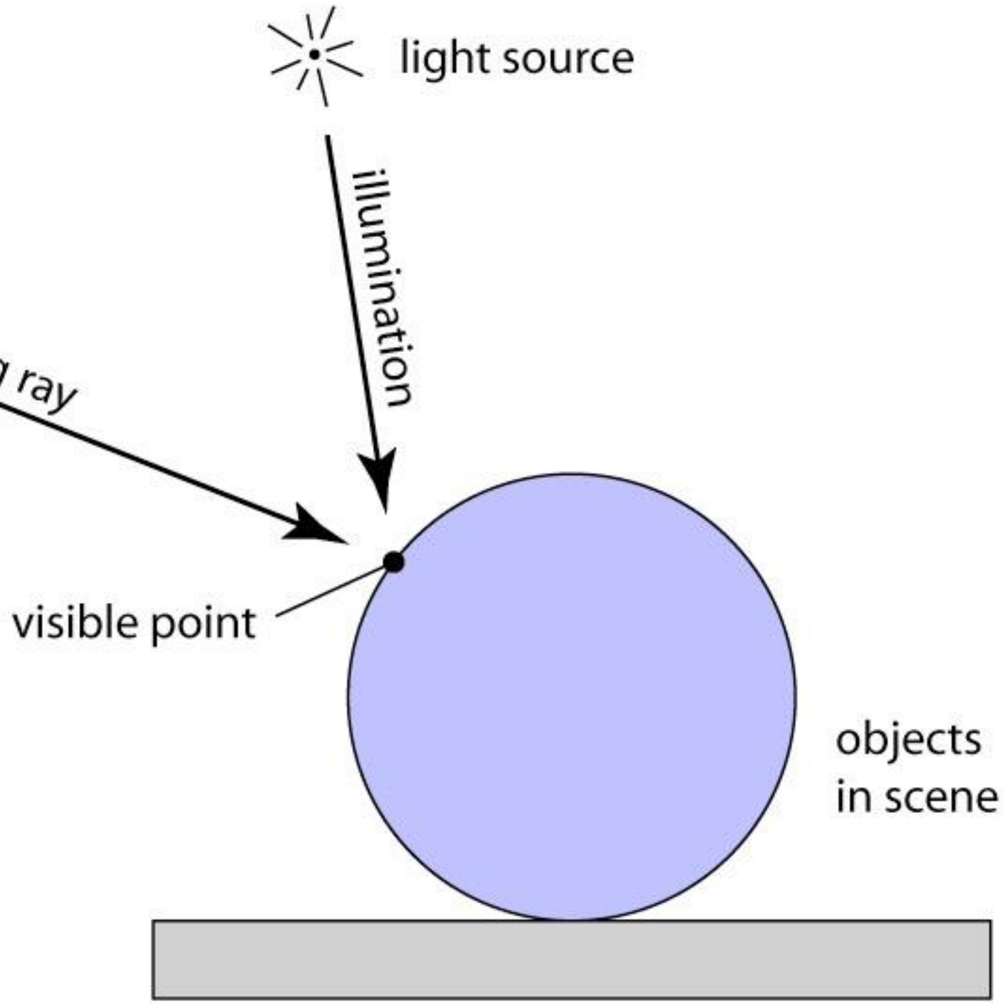


Ray tracing idea



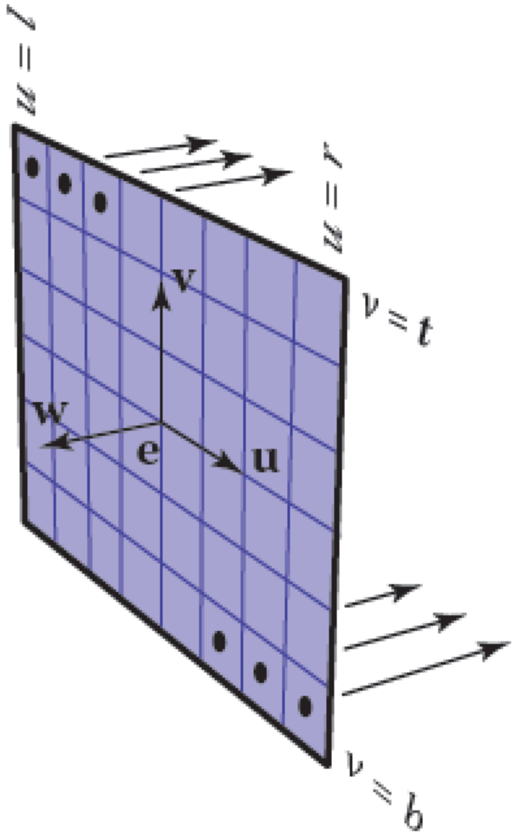


Ray tracing algorithm

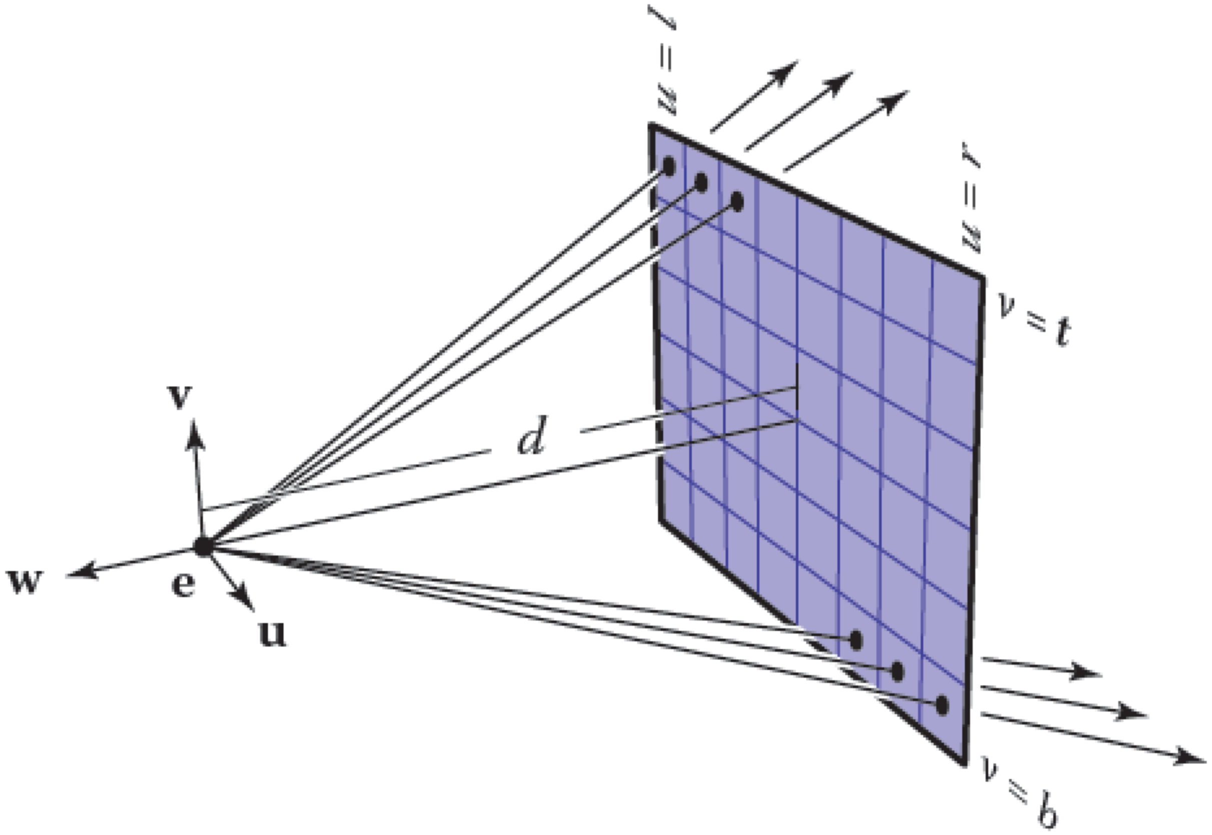


for each pixel {
compute viewing ray
intersect ray with scene
compute illumination at visible point
put result into image
}

Generating rays

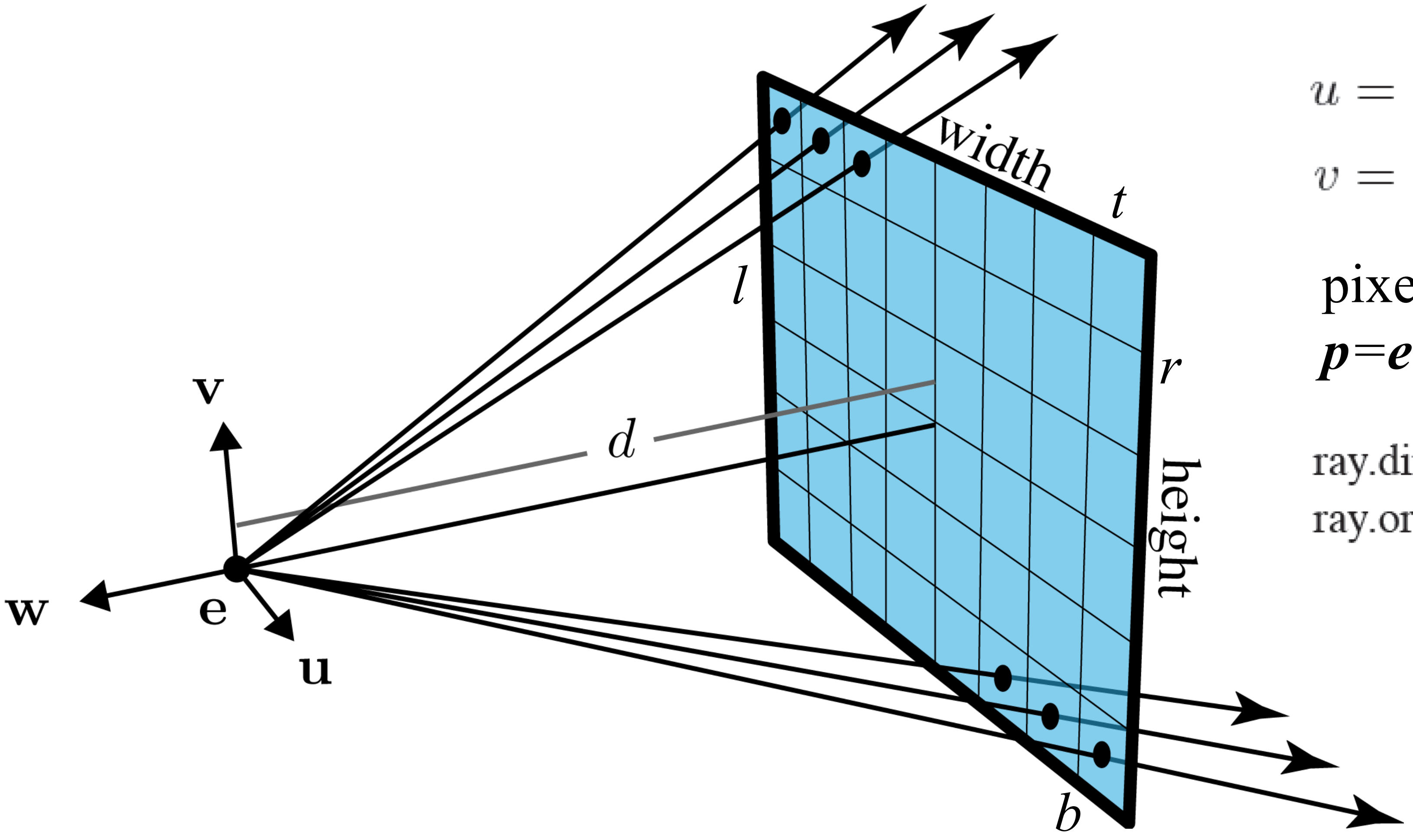


Parallel projection
same direction, different origins



Perspective projection
same origin, different directions

Perspective Camera



For $n_x * n_y$ pixel image

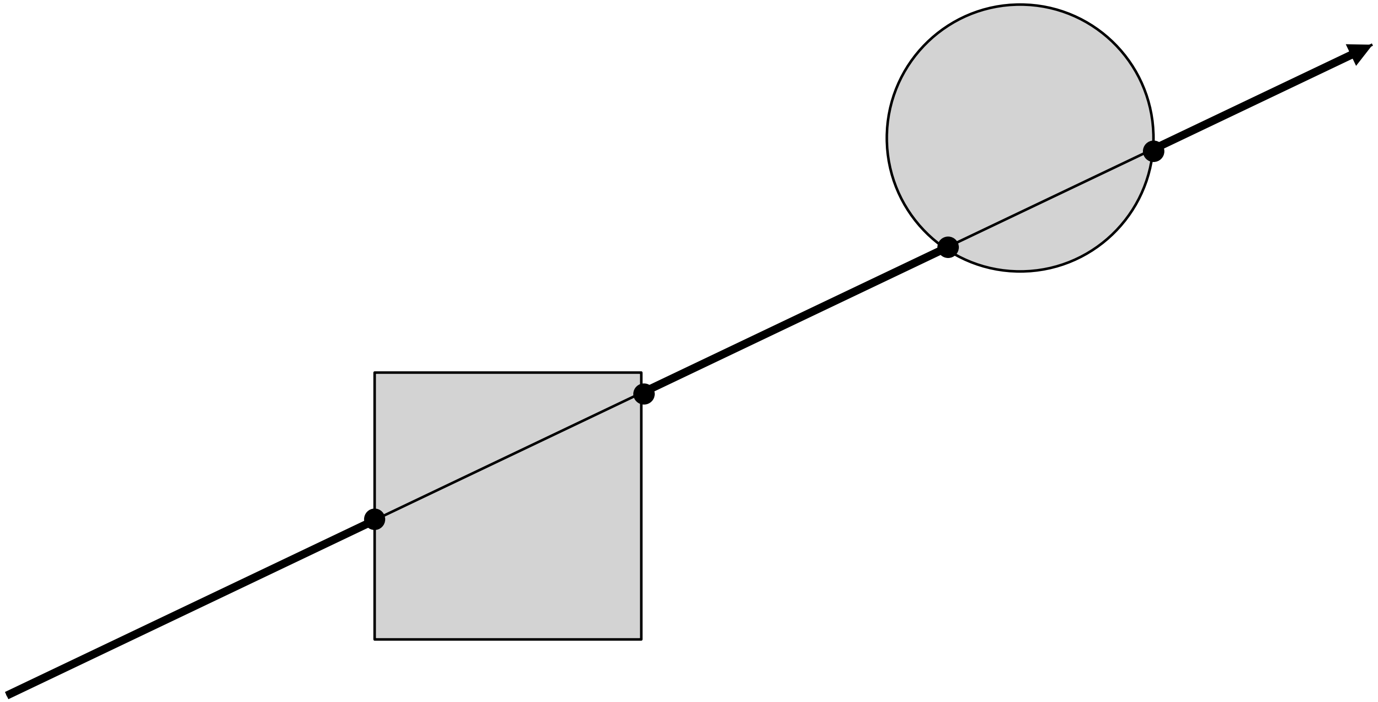
l=-width/2, r=width/2
b=-height2, t=height/2

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

pixel (i,j) is
$\boldsymbol{p} = \boldsymbol{e} - d\boldsymbol{w} + u\boldsymbol{u} + v\boldsymbol{v}$

ray.direction $\leftarrow -d\,\mathbf{w} + u\,\mathbf{u} + v\,\mathbf{v}$
ray.origin $\leftarrow \mathbf{e}$

Ray intersection

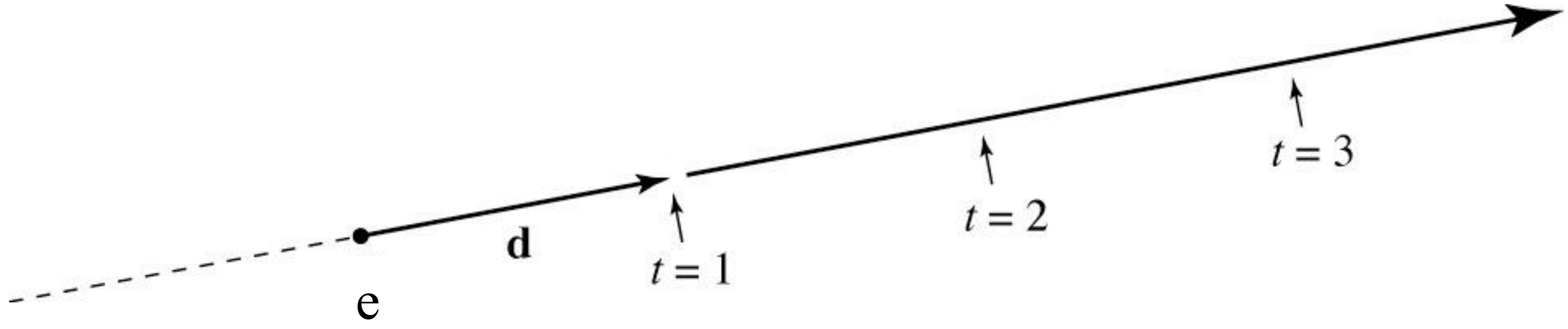


Ray: a half line

Standard representation: point p and direction d

$$\boldsymbol{p}(t)=\boldsymbol{e}+t\boldsymbol{d}$$

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



Ray-sphere intersection: algebraic

- Condition 1: point is on ray

$$\boldsymbol{p}(t)=\boldsymbol{e}+t\boldsymbol{d}$$

- 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:

$$(\boldsymbol{e}+t\boldsymbol{d}).\ (\boldsymbol{e}+t\boldsymbol{d})\ -1 = 0$$

$$t^2\boldsymbol{d.d} + \ t*2\boldsymbol{e.d} + \boldsymbol{e.e}-1 = 0$$

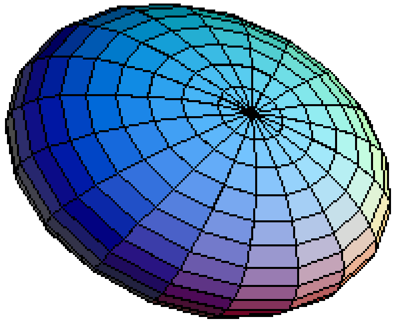
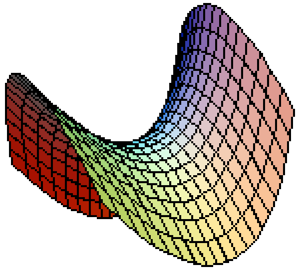
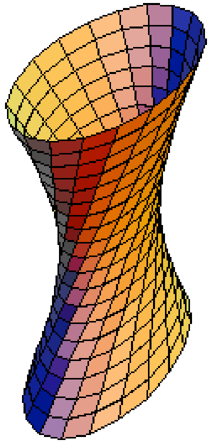
  - this is a quadratic equation in *t*

Ray-sphere intersection: algebraic

- Solution for *t* by quadratic formula:

$$t = \frac{-\boldsymbol{e}.\boldsymbol{d} \pm \sqrt{(\boldsymbol{e}.\boldsymbol{d})^2 - (\boldsymbol{d}.\boldsymbol{d})(\boldsymbol{e}.\boldsymbol{e} - 1)}}{(\boldsymbol{d}.\boldsymbol{d})}$$

Computing Ray-Quadric Intersections

Implicit equation for quadrics is

$\boldsymbol{p^T Q p = 0}$ where $\boldsymbol{Q}$ is a 4x4 matrix of coefficients.

**why 4x4 for a 3D point?*

Substituting the ray equation $\boldsymbol{e}+\boldsymbol{d}t$ for $\boldsymbol{p}$ gives us a quadratic equation in t, whose roots are the intersection points.

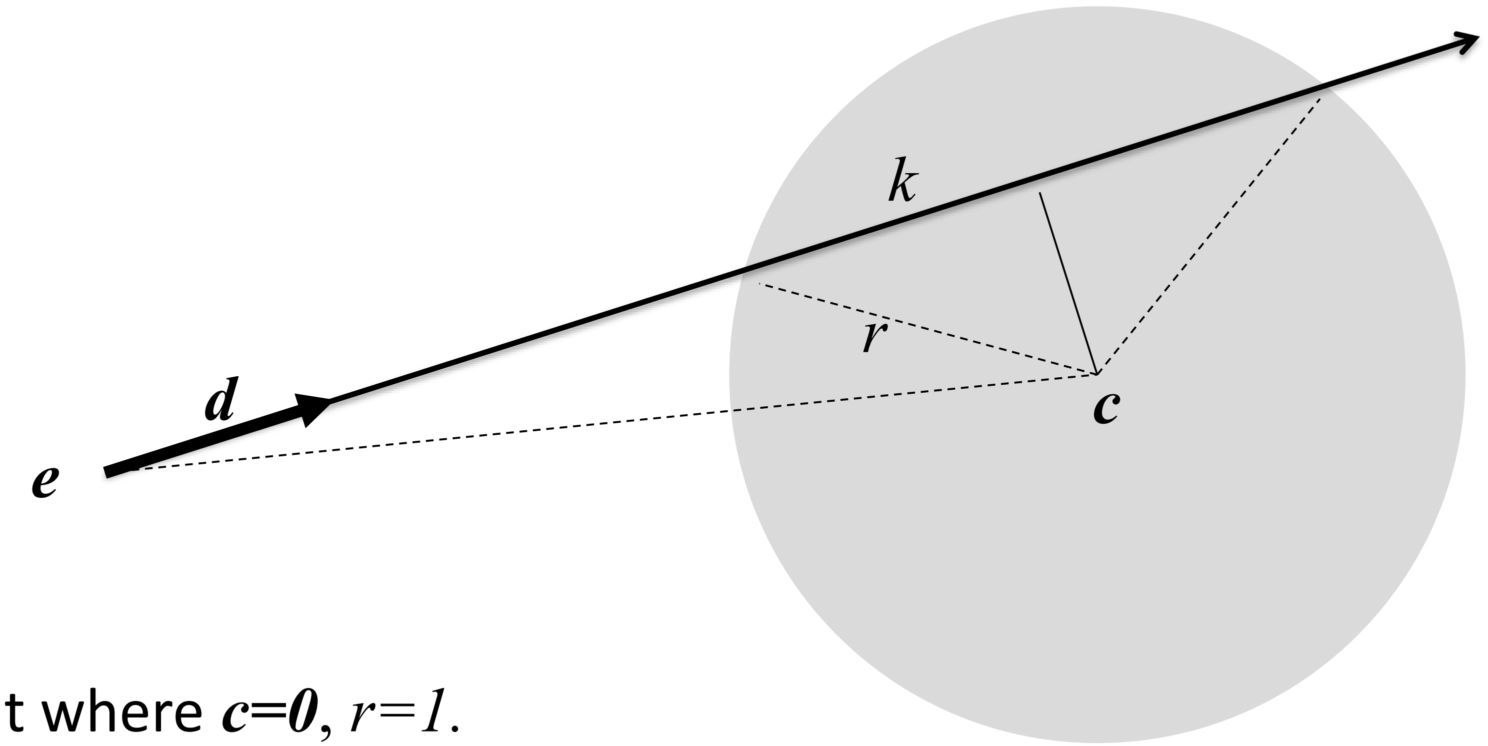
Ray-sphere intersection: geometric

$(\boldsymbol{c}-\boldsymbol{e})^2 - ((\boldsymbol{c}-\boldsymbol{e}).\boldsymbol{d})^2 = r^2 - k^2$

Solve for k, if it exists.

Intersection points:

$\boldsymbol{e}+\boldsymbol{d}\,((\boldsymbol{c}-\boldsymbol{e}).\boldsymbol{d} \;+/-\; k)$



Compare to algebraic result where $\boldsymbol{c}=\boldsymbol{0}$, $r=1$.

Ray-triangle intersection

- Condition 1: point is on ray

$$\boldsymbol{p}(t)=\boldsymbol{e}+t\boldsymbol{d}$$

- 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

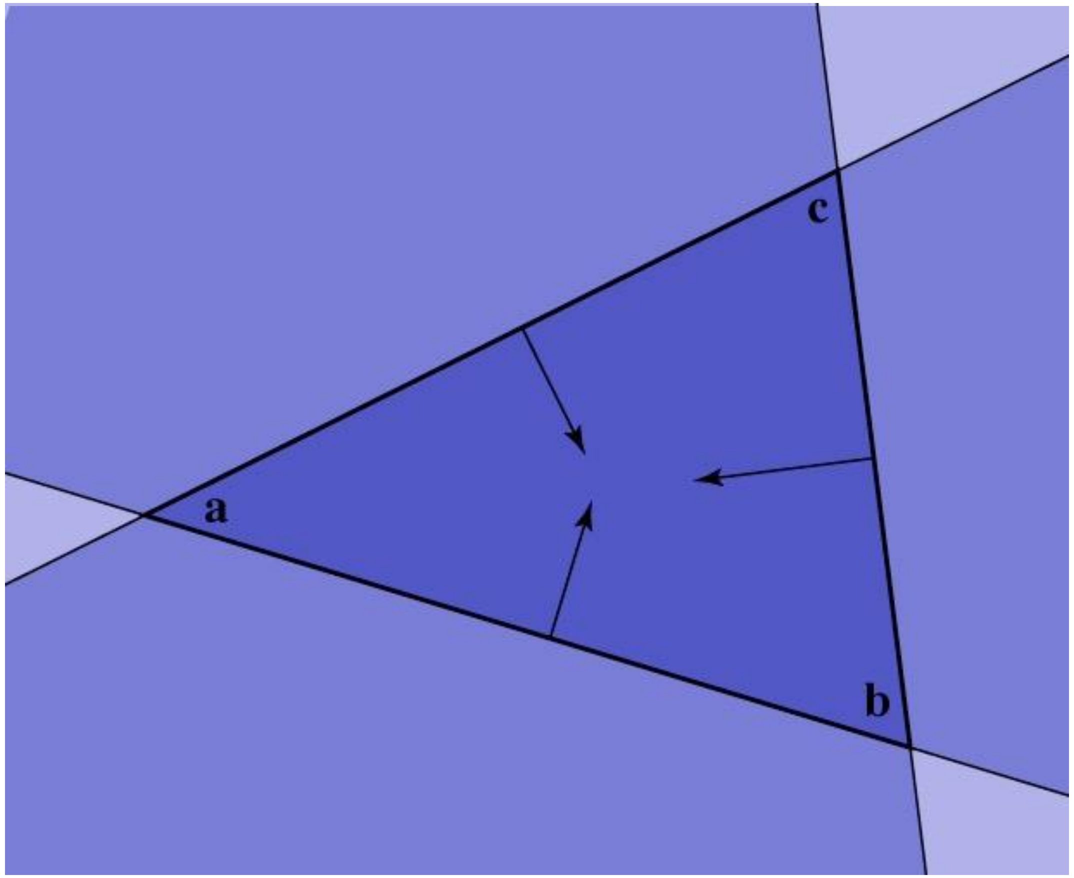
- First solve 1&2 (ray–plane intersection)
  - substitute and solve for *t*:

$$(\ \boldsymbol{e}+t\boldsymbol{d}-\boldsymbol{a}\ )\ .\ \boldsymbol{n}=0$$

$$t=(\ \boldsymbol{a}-\boldsymbol{e}\ )\ .\ \boldsymbol{n}\ /\ (\ \boldsymbol{d}.\boldsymbol{n}\ )$$

Ray-triangle intersection

In plane, triangle is the intersection of 3 half spaces



Deciding about insideness

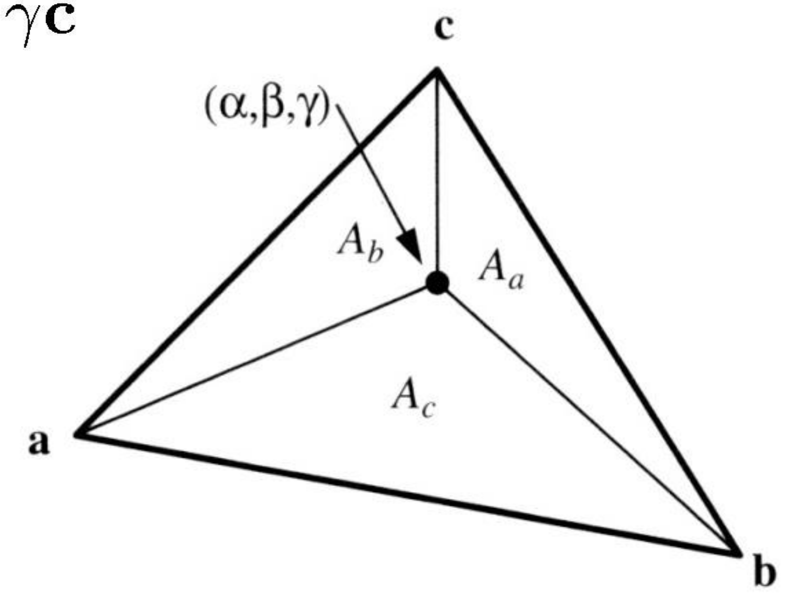
- Need to check whether hit point is inside 3 edges
  - easiest to do in 2D coordinates on the plane
- Will also need to know where we are in the triangle
  - for textures, shading, etc. … next couple of lectures
- Efficient solution: transform to coordinates aligned to the triangle

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 (area ratios):
  - Linear viewpoint (basis vectors):

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

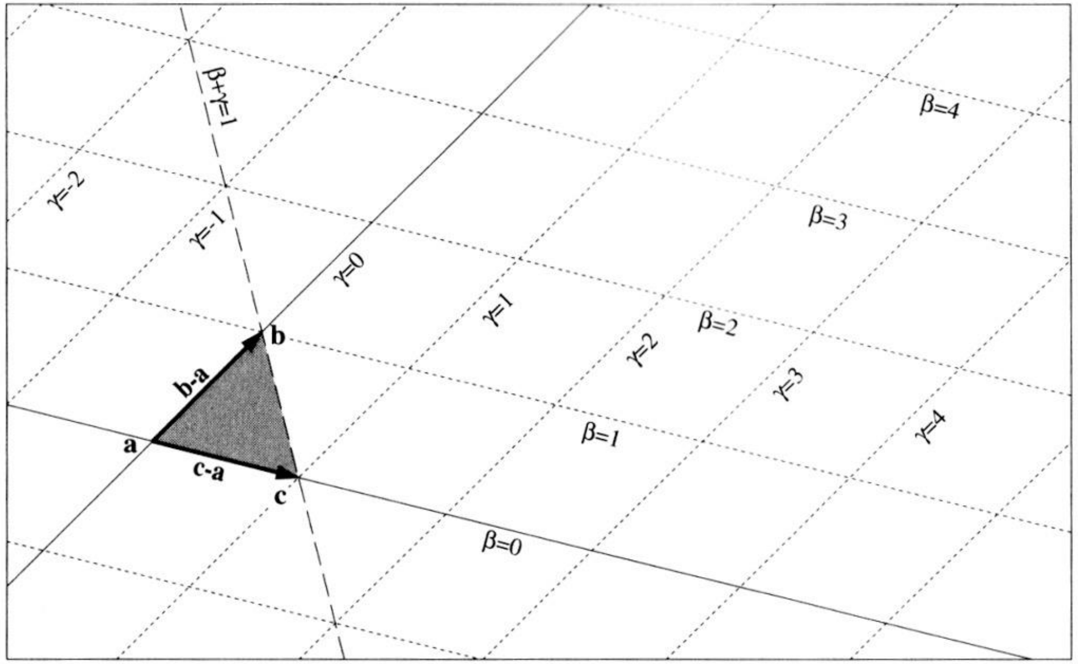
$$\mathbf{p} = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a})$$

- Triangle interior test:

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

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

$$\boldsymbol{p}(t) = \boldsymbol{e} + t\boldsymbol{d}$$

  for some number t.

- Set them equal: 3 linear equations in 3 variables

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

  solve them to get t, β, and γ all at once!

…Solve using Cramer’s rule 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 {
    …
    boolean intersect(Intersection result, Ray r);
}
```

was there an intersection?

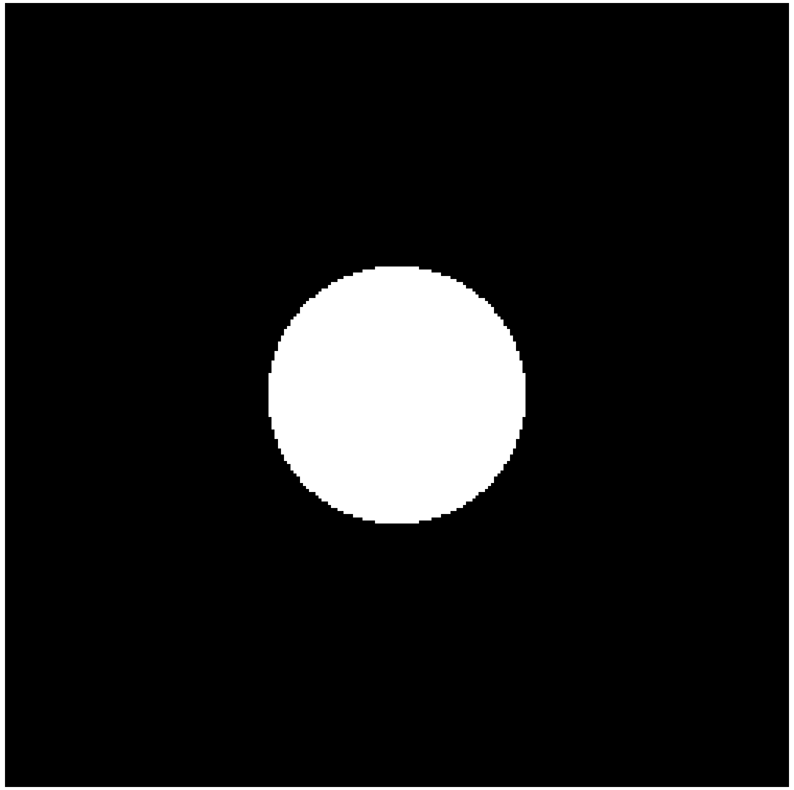
information about first intersection or list of ALL intersections

```
class Intersection {
    float t;
    Vector3 hitLocation;
    Vector3 normal;
    …
}
```

Image so far

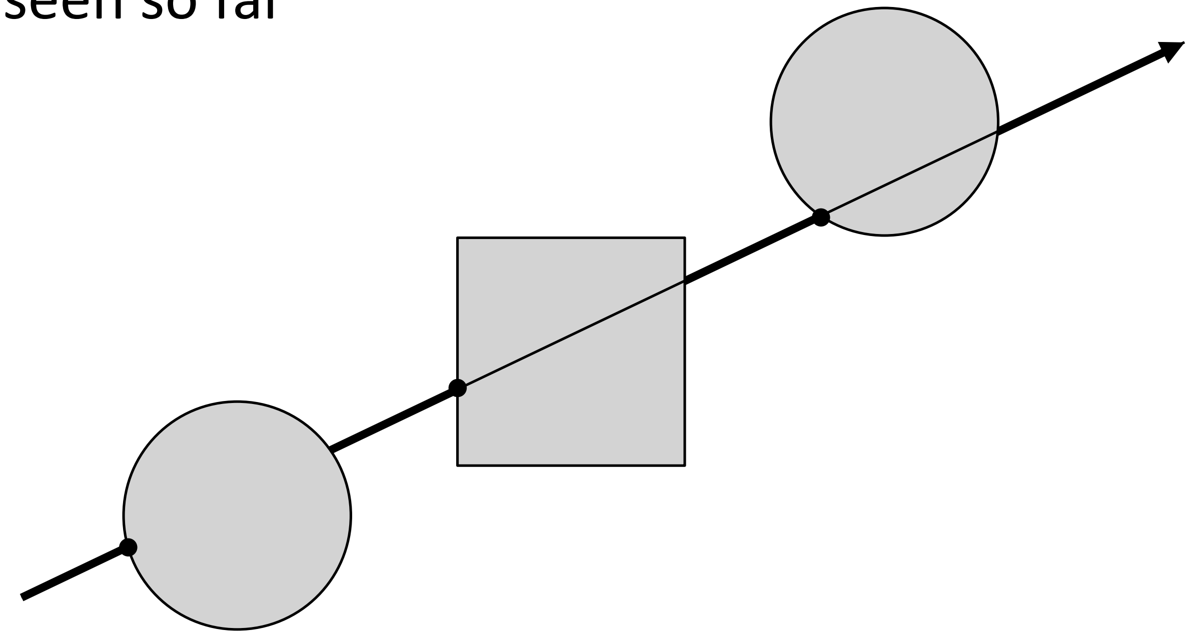
With eye ray generation and sphere intersection

```
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 = s.intersect(result,ray)
        if (hitSurface)
            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



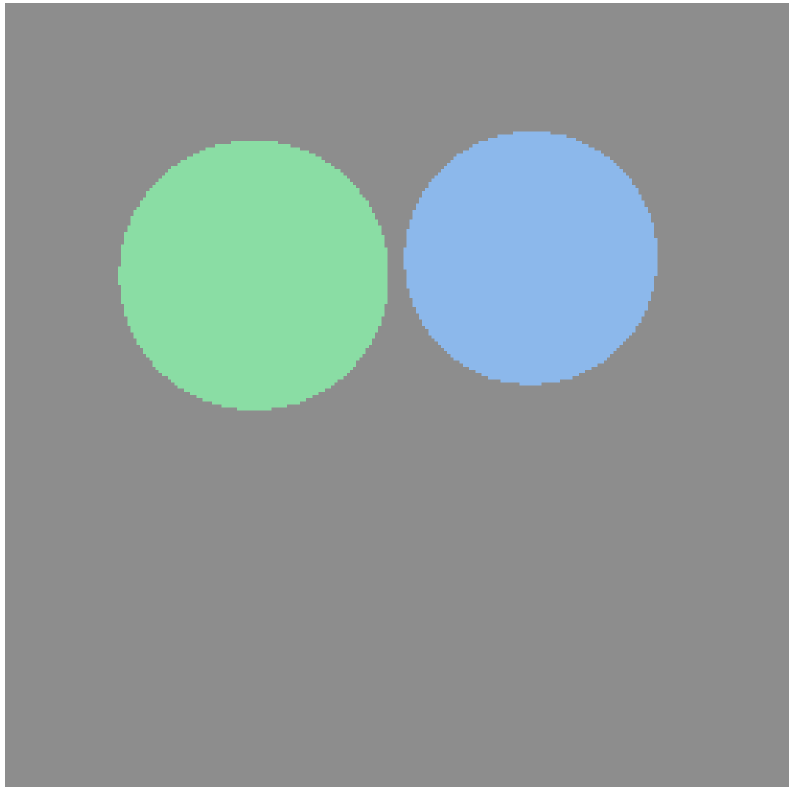
Intersection against many shapes

```
scene.intersect (ray, tMin) {
    tMin = +inf; firstSurface = null;

    for surface in scene {
        hitSurface = surface.intersect(result, ray);
        if (hitSurface && result.t<tMin {
            tMin = result.t;
            firstSurface = surface;
        }
    }
return firstSurface;
}
```

– this is linear in the number of shapes
  but there are sublinear speed-ups.

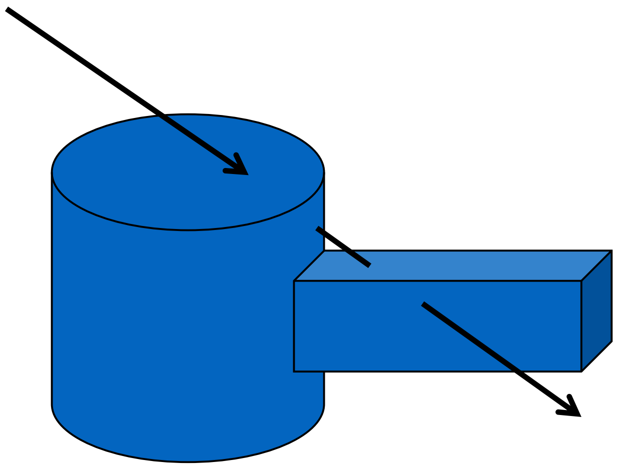
Image so far

```
for 0 <= iy < ny
   for 0 <= ix < nx
  {
      ray = camera.getRay(ix, iy);
      firstSurface = scene.intersect(result,ray);
      if (firstSurface)
         image.set(ix, iy, firstSurface.color);
      else
         image.set(ix, iy, background.color);
   }
```



Intersecting Rays & Composite Objects

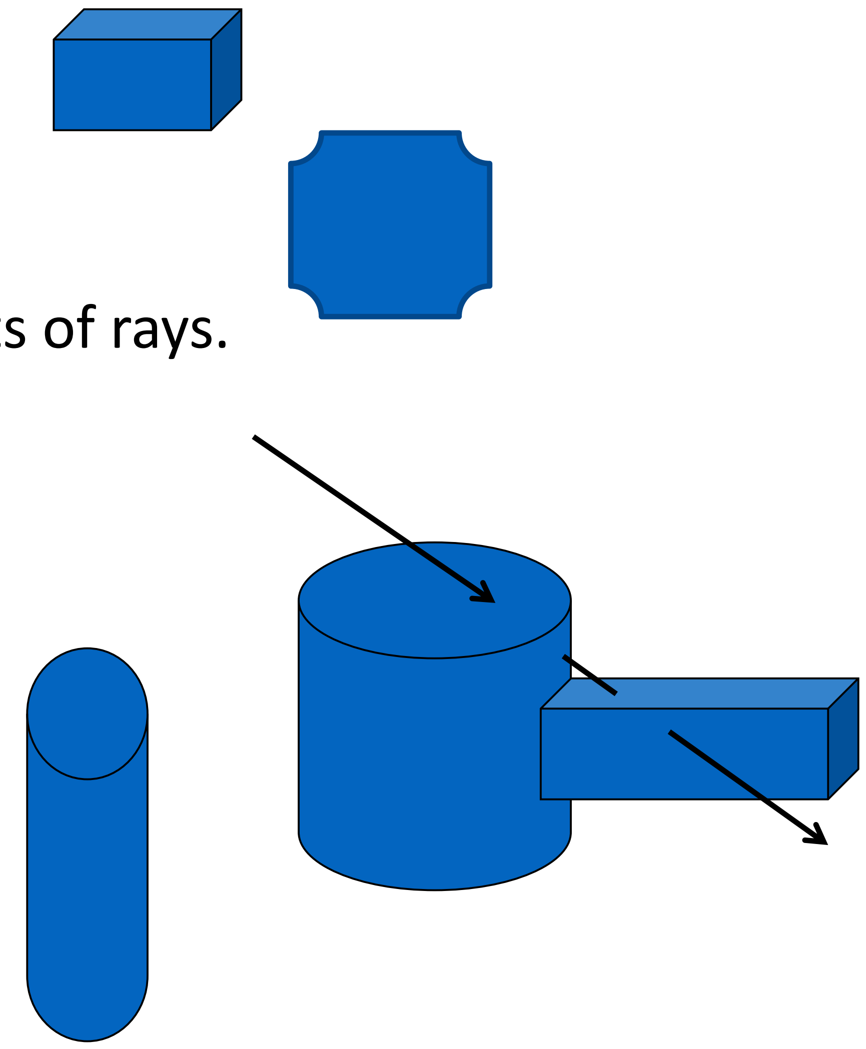
- Intersect ray with component objects
- Process the intersections ordered by depth to return intersection pairs with the object.



Ray Intersection: Efficiency Considerations

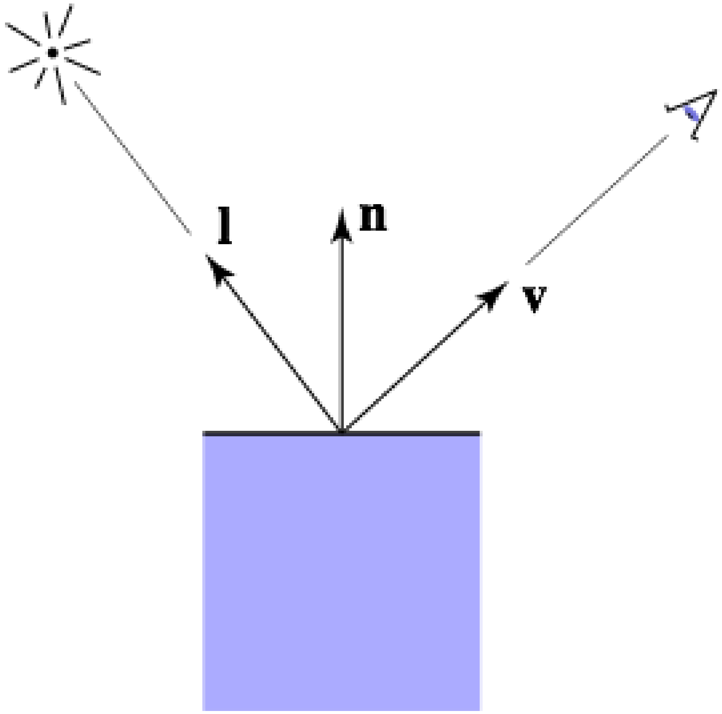
Speed-up the intersection process.

- Ignore object that clearly don't intersect.
- Use proxy geometry.
- Subdivide and structure space hierarchically.
- Project volume onto image to ignore entire sets of rays.



Shading

- Compute light reflected toward camera
- Inputs:
  - – eye direction
  - – light direction
    (for each of many lights)
  - – **surface normal**
  - – surface parameters
    (color, shininess, …)



Computing the Normal at a Hit Point

- Polygon normal: cross product of two non-collinear edges.

- Implicit surface normal $f(p)=0$:

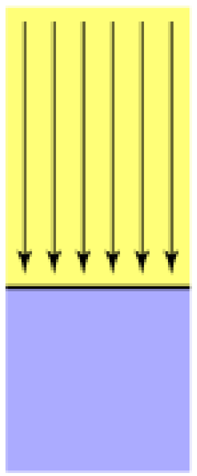
  $$gradient(f)(p).$$

- Explicit parametric surface $f(a,b)$:

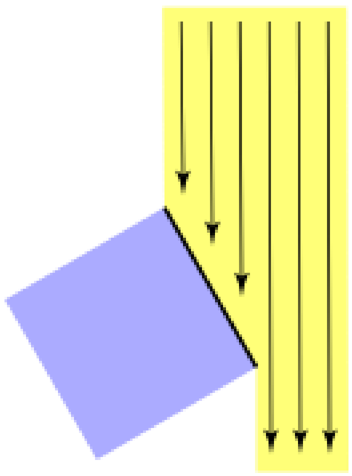
$$\delta f(s,b)/ \delta s \times \delta f(a,t)/ \delta t.$$

Diffuse reflection

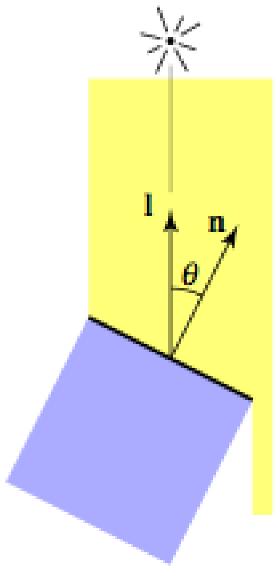
- Light is scattered uniformly in all directions
  - the surface color is the same for all viewing directions
- Lambert’s cosine law



Top face of cube receives a certain amount of light

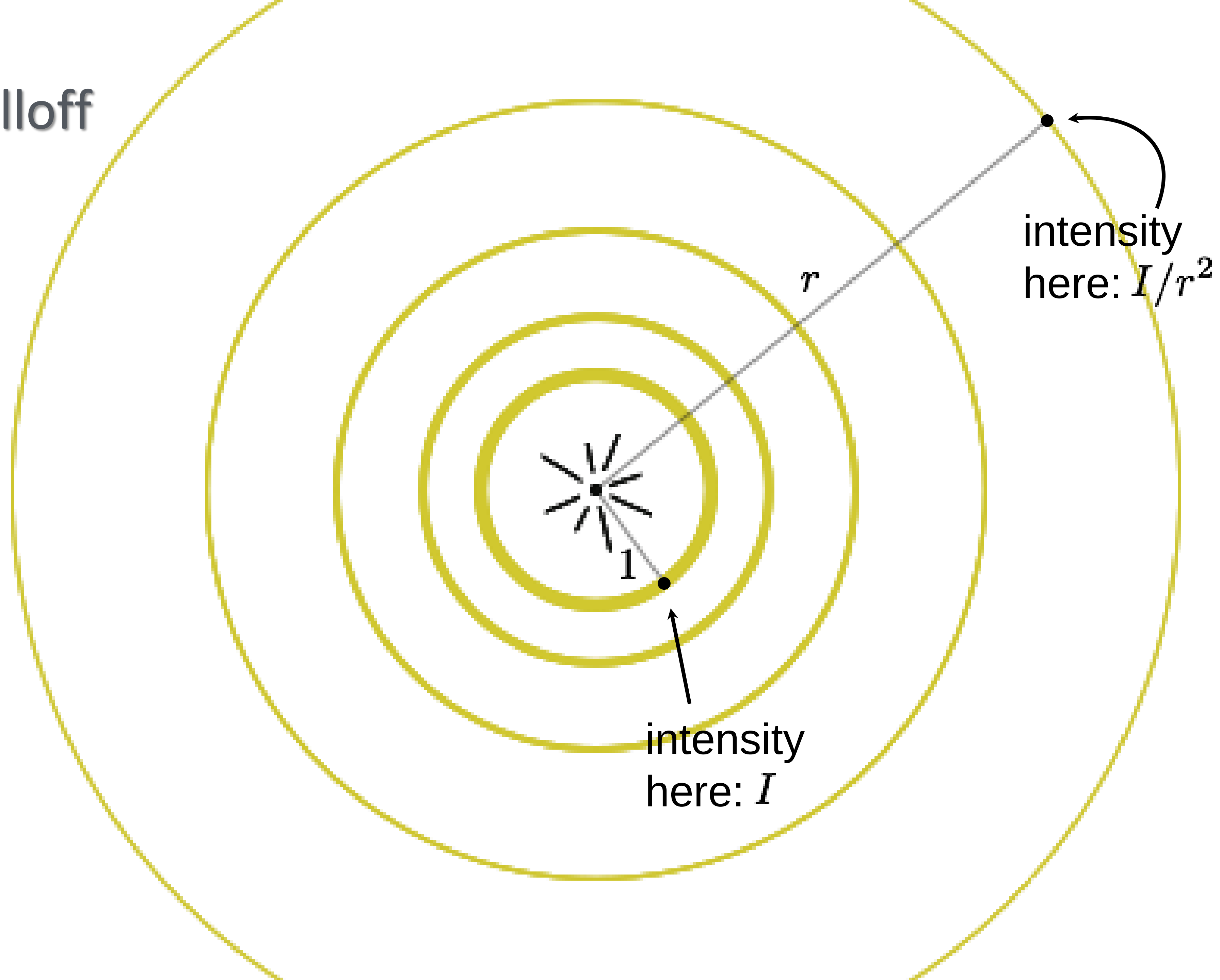


Top face of 60º rotated cube intercepts half the light



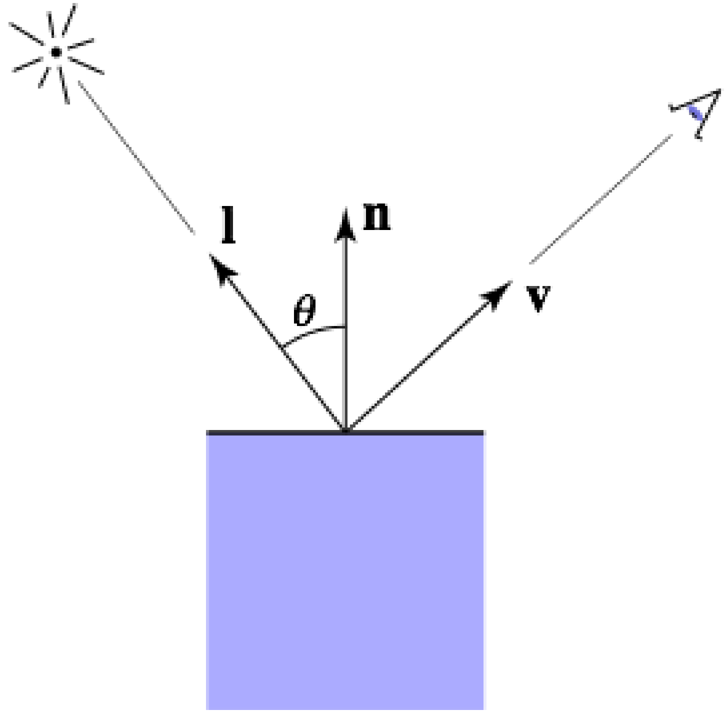
In general, light per unit area is proportional to $\cos\theta = \mathbf{l} \cdot \mathbf{n}$

Light falloff



Lambertian shading

Shading independent of view direction



illumination
from source

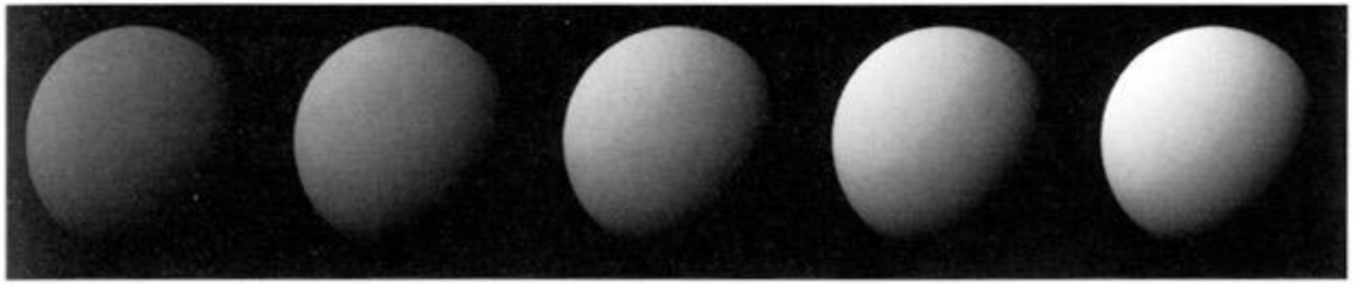
$$L_d = k_d \, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l})$$

diffuse
coefficient

diffusely
reflected
light

Lambertian shading

Produces a matte appearance



$k_d \longrightarrow$

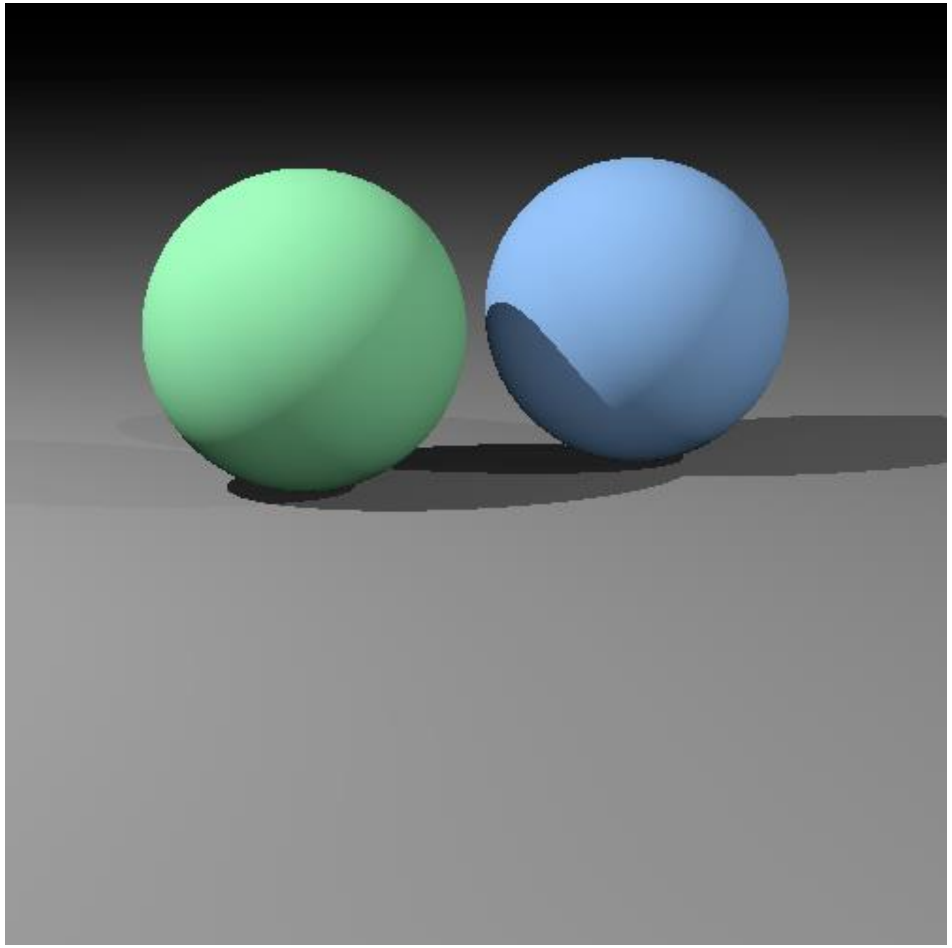
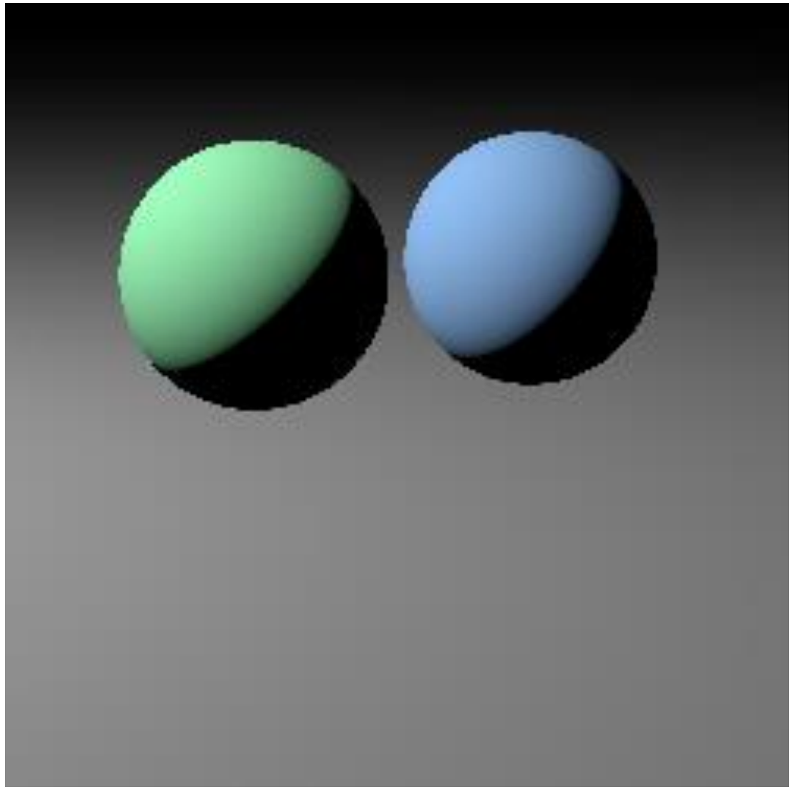


Image so far

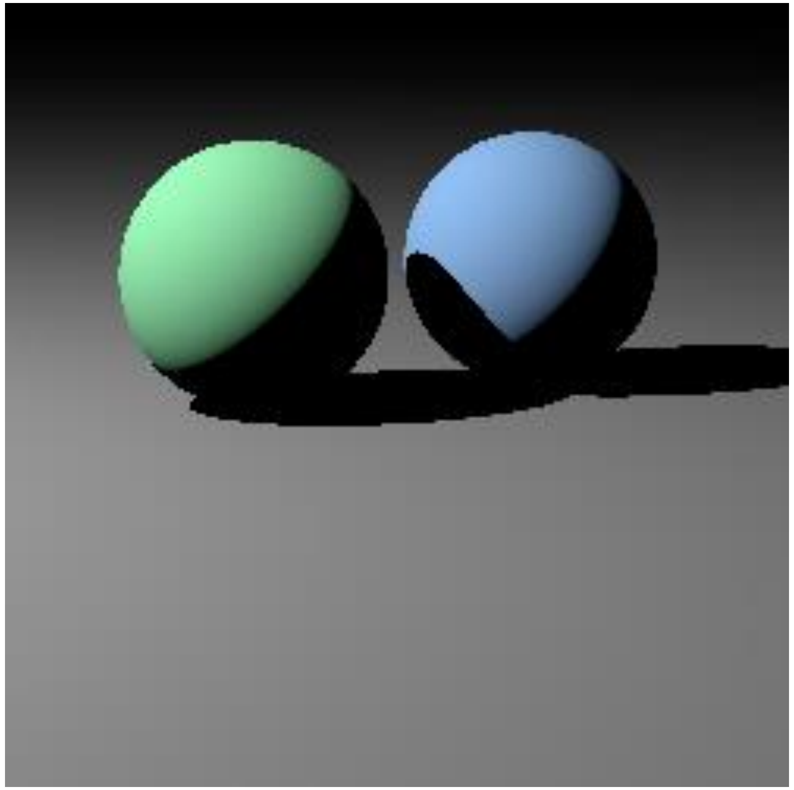
```
for 0 <= iy < ny
   for 0 <= ix < nx
  {
      ray = camera.getRay(ix, iy);
      firstSurface = scene.intersect(result,ray);
      if (firstSurface)
         image.set(ix, iy,
               firstSurface.shade(ray,light,result.point, result.normal);
      else
         image.set(ix, iy, background.color);
   }

Surface.shade(ray,light,point,normal) {
            l=light.pos-position;
            it= surface.k*light.intensity*max(0,normal.l);
            return surface.color*it;
}
```



Shadows

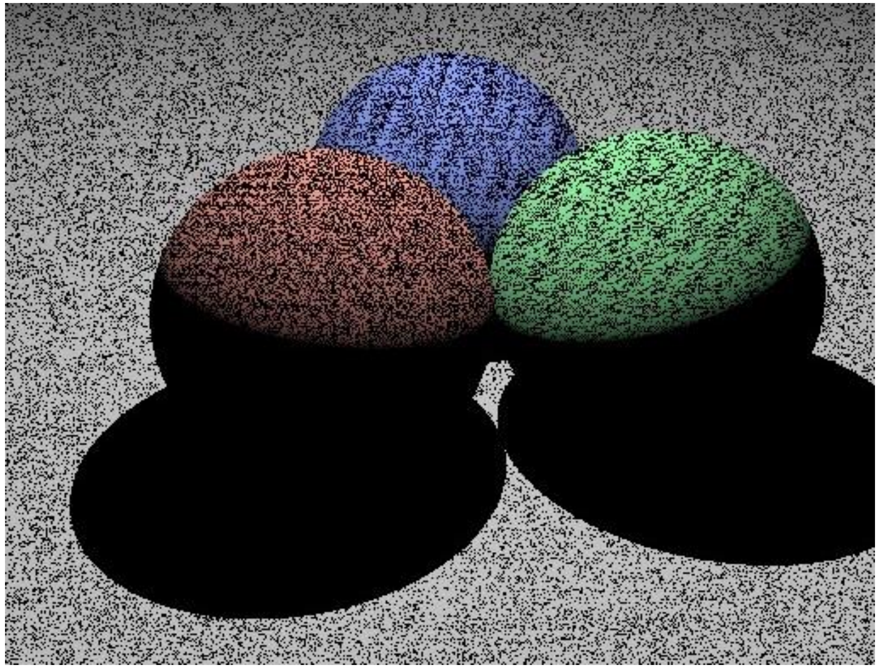
- Surface is only illuminated if nothing blocks its view of the light.
- With ray tracing it's easy to check if a point in the scene is in shadow.
  - just shoot a ray from the point to the light and intersect it with the scene!

```
Surface.shade(ray,light,point,normal) {
        l=light.pos-position;
        shadowray=(point,l);
        if  !scene.intersect(result,shadowray)
        {
            it= surface.k*light.intensity*max(0,normal.l);
            return surface.color*it;
        }
        else
            return black;
}
```



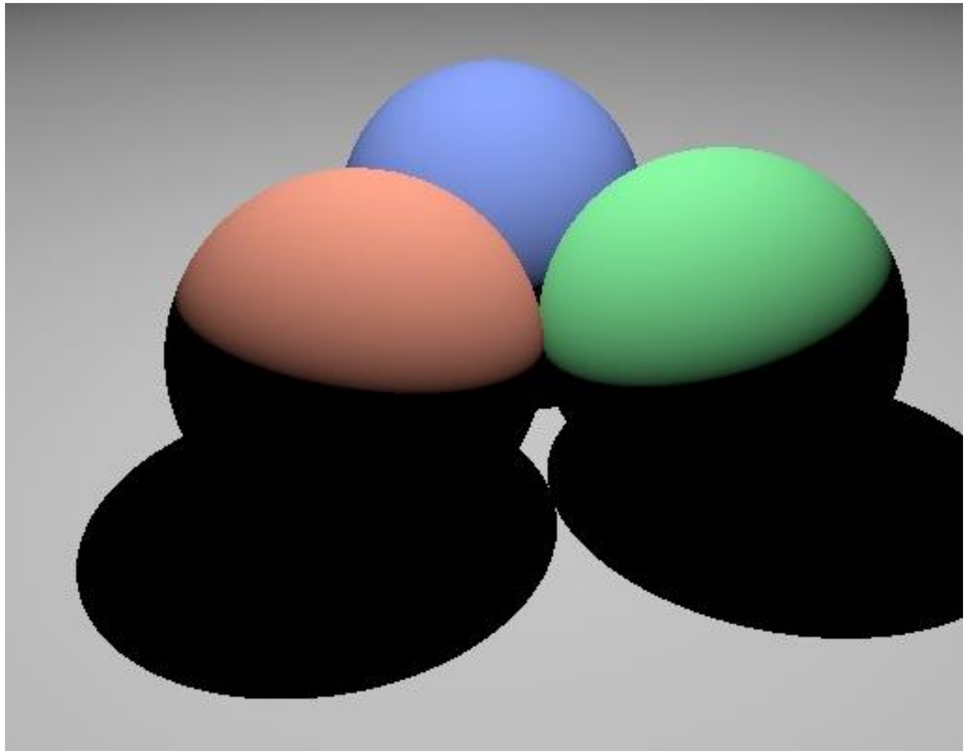
Classic shadow error

What's going on?



Classic shadow error

Start shadow rays just outside surface

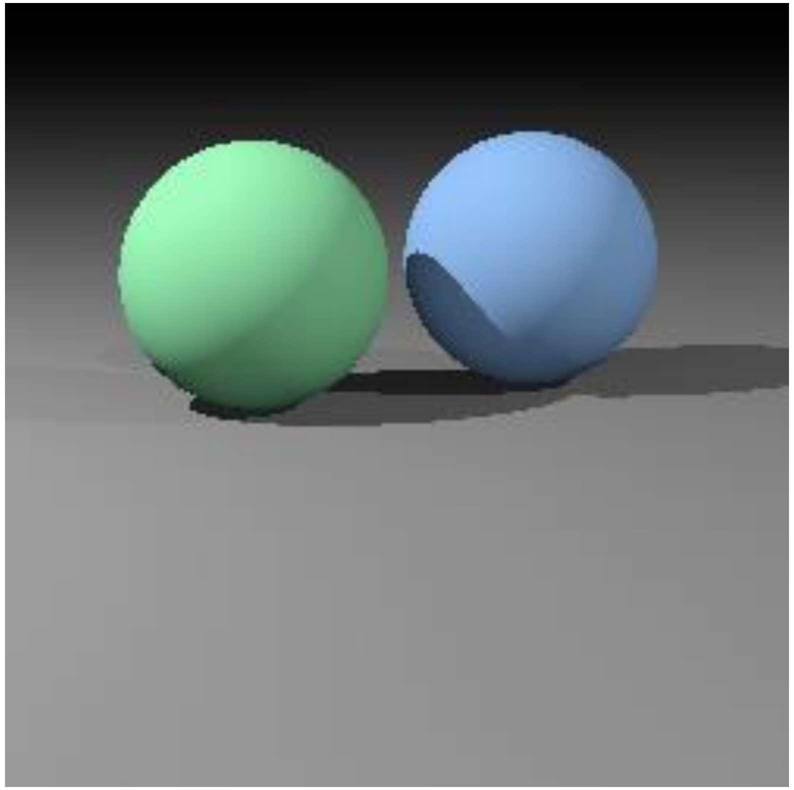


Multiple lights

- 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

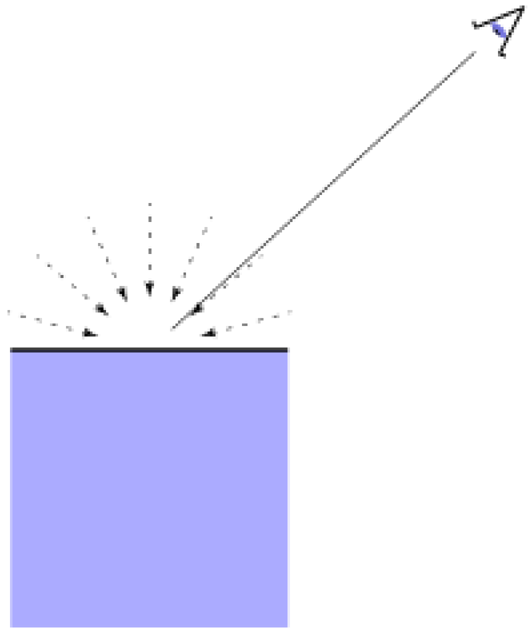
```
shade(ray, lights, point, normal) {
   result = ambient;
   for light in lights {
         l=light.pos-position;
         shadowray=(point,l);
         if  !scene.intersect(result,shadowray)
         {
               it= surface.k*light.intensity*max(0,normal.l);
               result+= surface.color*it;
         }
   }
   return result;
}
```



Ambient shading

Shading that does not depend on anything

– add constant color to account for disregarded illumination and fill in black shadows



$$L_a = k_a \, I_a$$

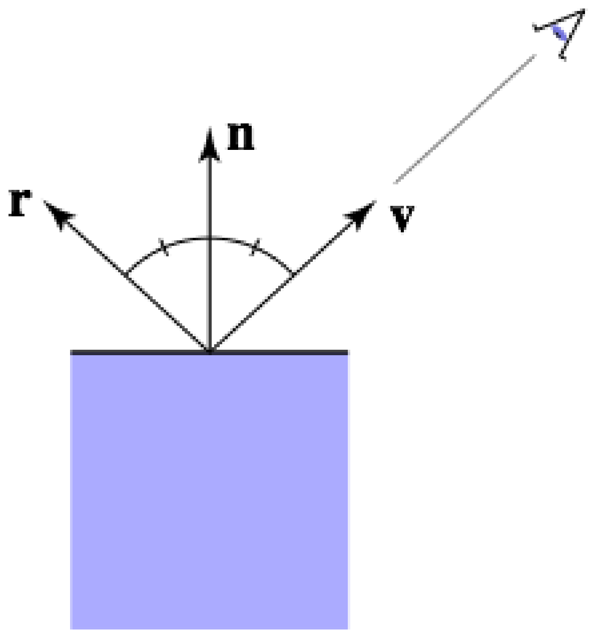
ambient coefficient

reflected ambient light

Mirror reflection

Intensity depends on view direction

– reflects incident light from mirror direction

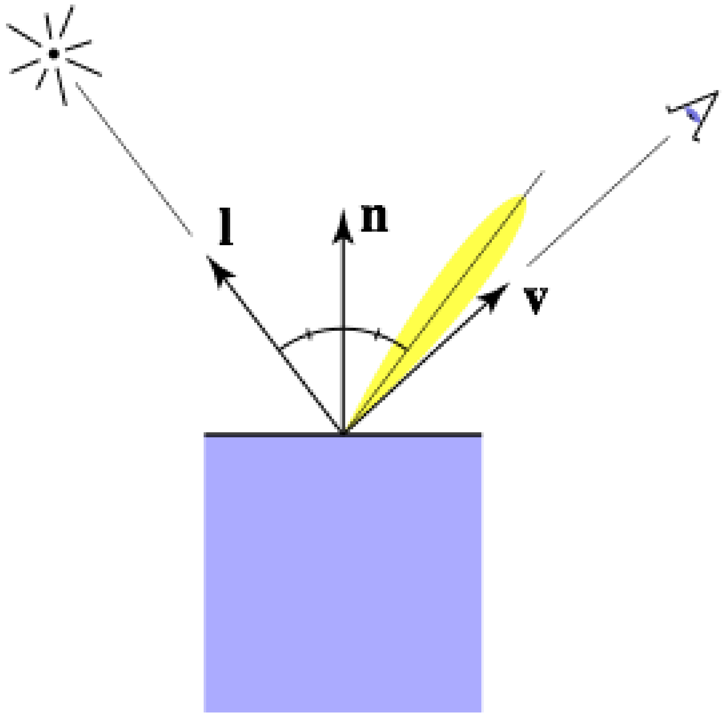


$$\mathbf{r} = \mathbf{v} + 2((\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v})$$
$$= 2(\mathbf{n}\cdot\mathbf{v})\mathbf{n} - \mathbf{v}$$

Specular shading (Phong)

Intensity depends on view direction

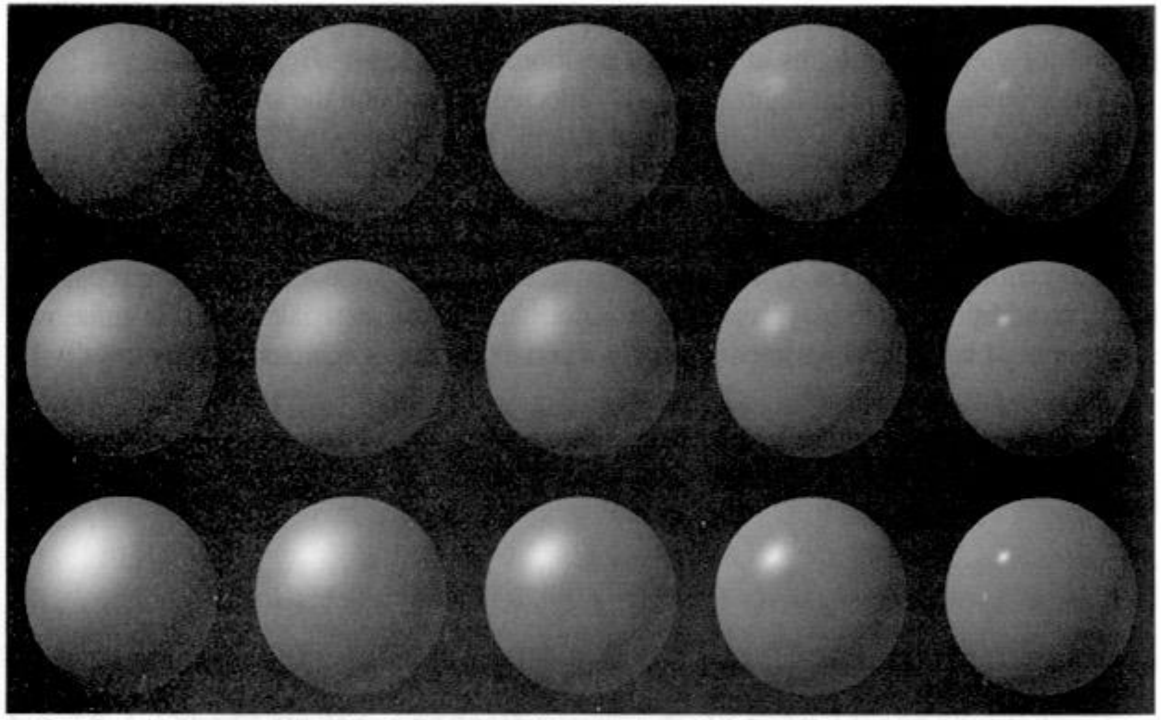
– bright near mirror configuration $k_s * I_s * (\mathbf{v} \cdot \mathbf{r})^{shiny}$



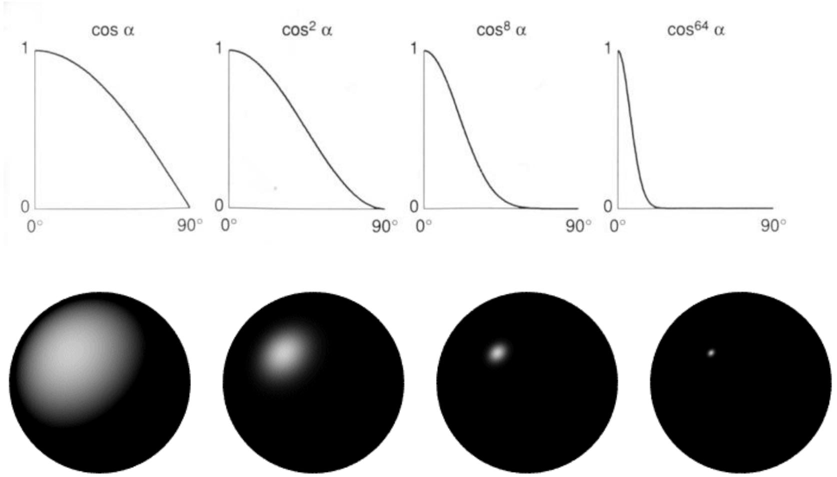
Phong model

Increasing *shiny* narrows the lobe

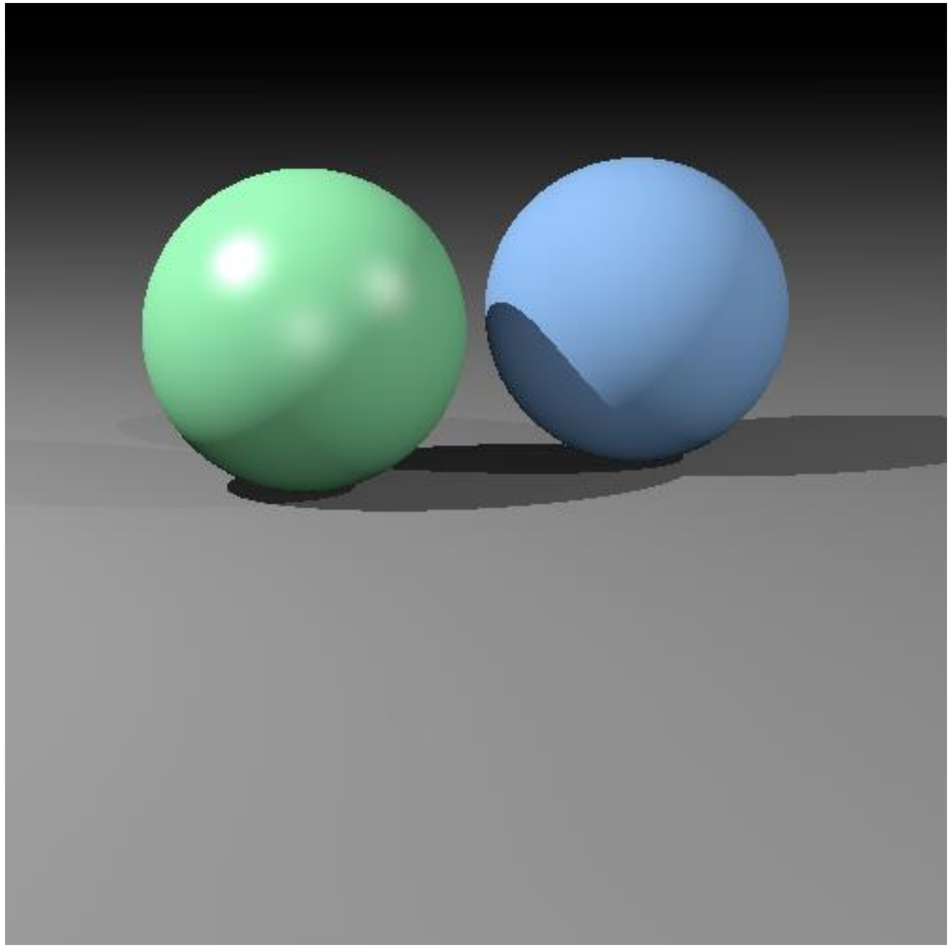
k_s



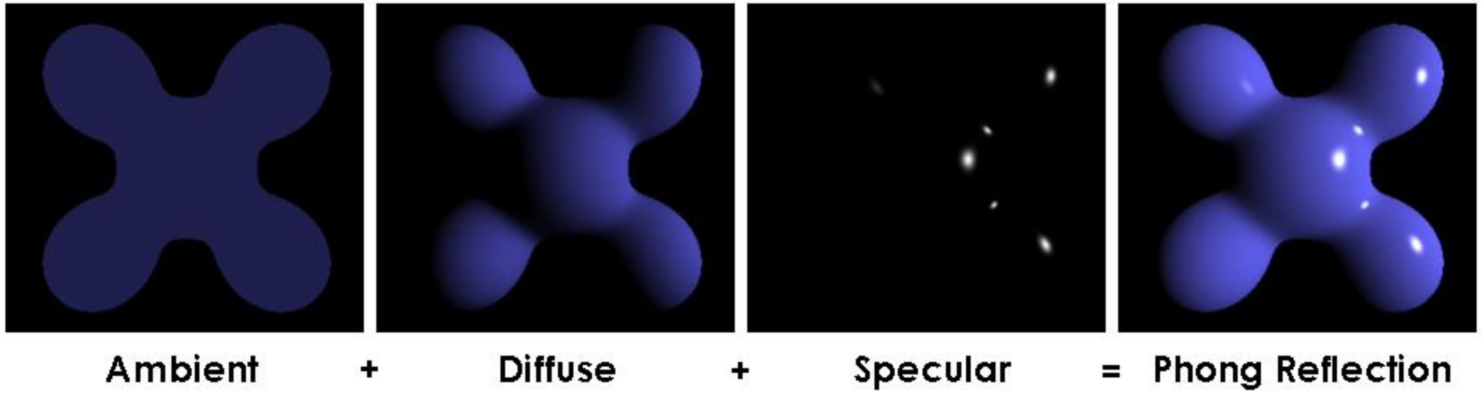
shiny



Diffuse + Phong shading



Phong Illumination



- Usually include ambient, diffuse, Phong in one model

$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, (I/r^2)\max(0, \mathbf{n}\cdot\mathbf{l}) + k_s\, (I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p
\end{aligned}
$$

- The final result is the sum over many lights

$$L = L_a + \sum_{i=1}^{N} [(L_d)_i + (L_s)_i]$$

$$L = k_a\, I_a + \sum_{i=1}^{N} \left[k_d\, (I_i/r_i^2)\max(0, \mathbf{n}\cdot\mathbf{l}_i) + k_s\, (I_i/r_i^2)\max(0, \mathbf{n}\cdot\mathbf{h}_i)^p\right]$$

Next Lecture : mirror reflections and ray tracing for global illumination