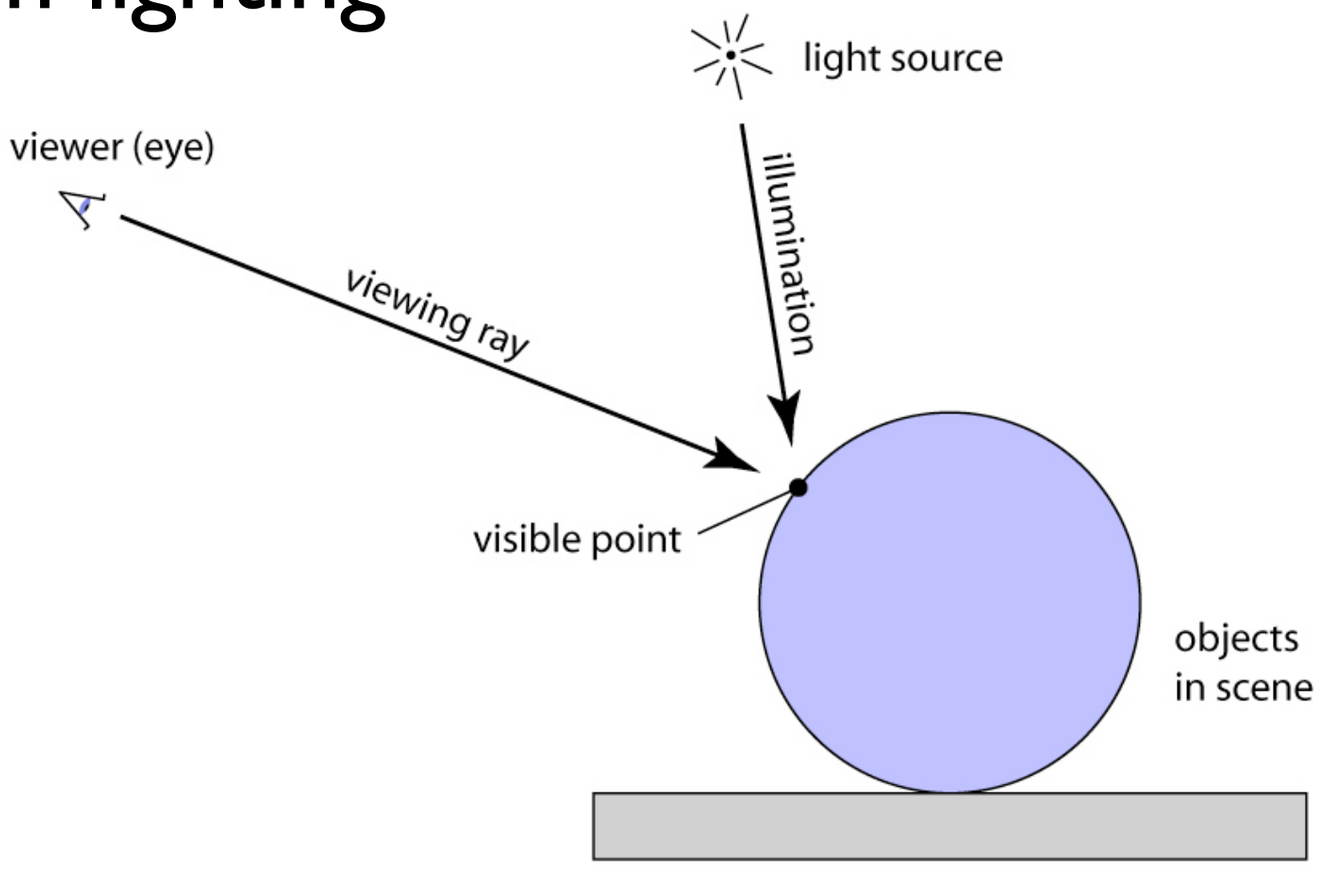
Ray Tracing

Introduction to Computer Graphics
CSE 533/333

Ray Tracing

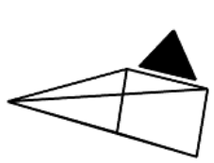
- Rendering 3D objects in a scene to produce a 2D image showing objects as viewed from a viewpoint with given lighting



Source: Steve Marschner, 2008

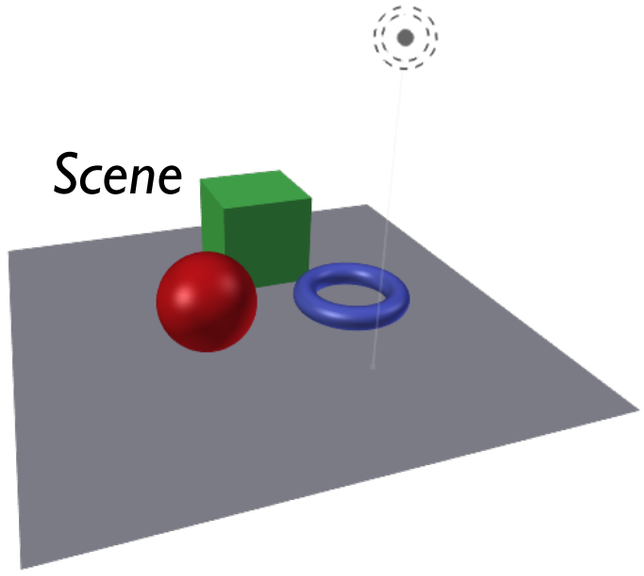
Ray Tracing

Camera

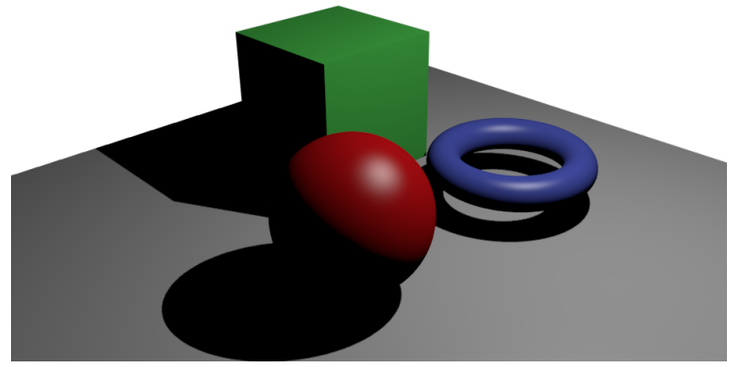


Point light source

Scene



3D scene description



Ray-traced image

(As viewed from the camera)

Ray Tracing

- Rendering is a process that takes as input a set of objects and produces an array of pixels
- Rendering can be organised in two ways:
  - *Object-order* rendering
  - *Image order* rendering
- Ray tracing is an image-order algorithm

Basic Algorithm

- A basic ray tracer has three parts:

1. **Ray generation**: computes origin and direction of each pixel's viewing ray based on camera geometry
2. **Ray intersection**: finds the closest object intersecting the viewing ray
3. **Shading**: computes the pixel colour based on the results of ray-object intersection

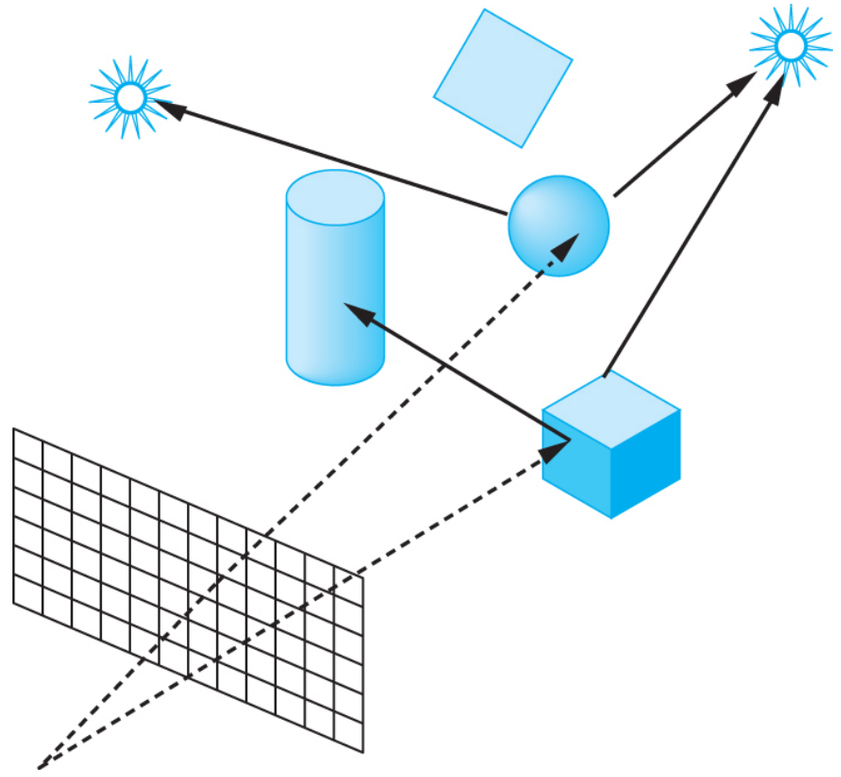
Basic Algorithm

foreach *pixel* **do**

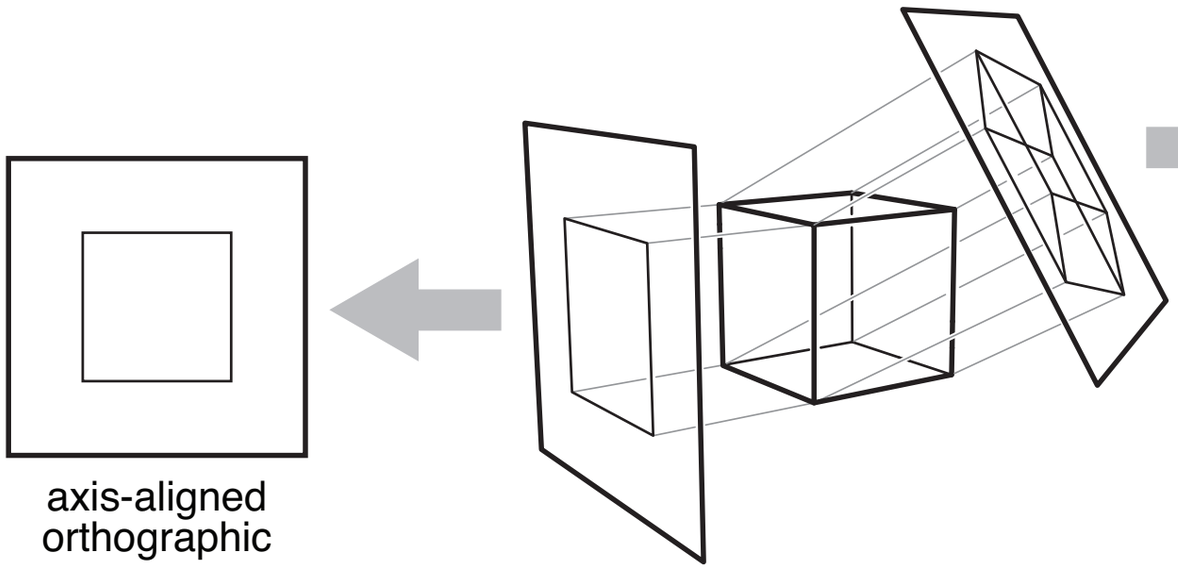
compute viewing ray

find first object hit by ray and its surface normal *n*

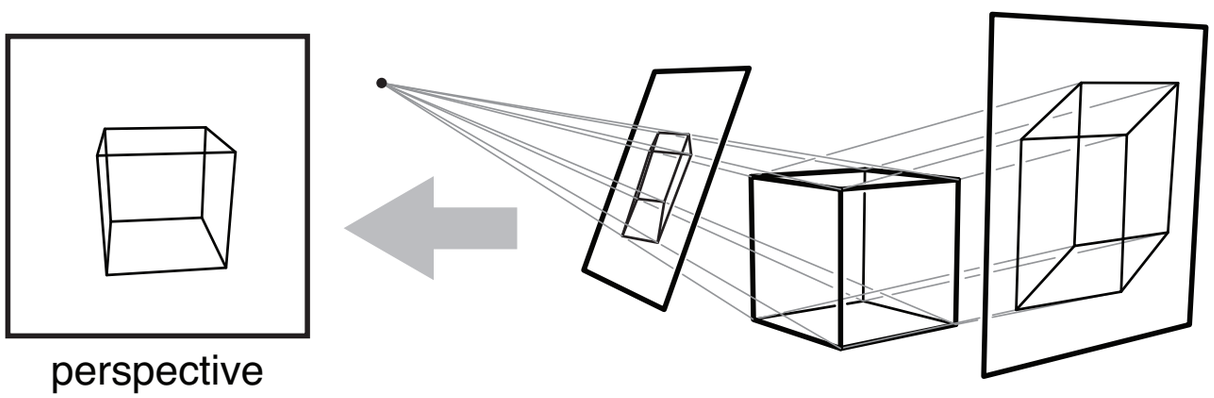
shade the pixel using hit-point, light, and *n*



Perspective



- Projection lines parallel
- Orthogonal to image plane
- Object size preserved



- Projection lines pass through viewpoint
- Object size not preserved

Computing Viewing Rays

Computing Viewing Rays

- Basic tools for ray generation are the *viewpoint* and the *image plane*
- Ray representation in parametric form:

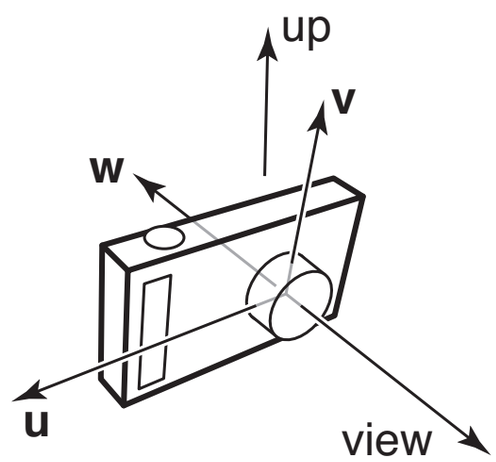
$$p(t) = \mathbf{e} + t(s - \mathbf{e}),$$

where e is eye, and s is a point on the image plane

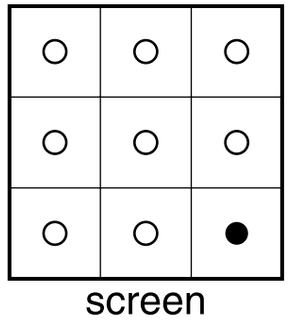
- Point e is the ray's *origin*, and
(*s* - e) is the ray's *direction*

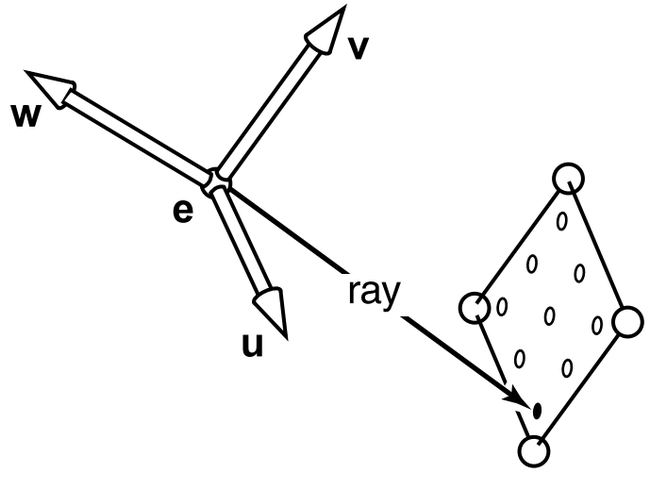
Computing Viewing Rays

- Ray-generation takes place from the camera frame
  - position e (eye point)
  - ***u, v,*** and **w** basis vectors



- Most commonly, we choose
  - e as viewpoint
  - **-w** as view direction
  - An up-vector

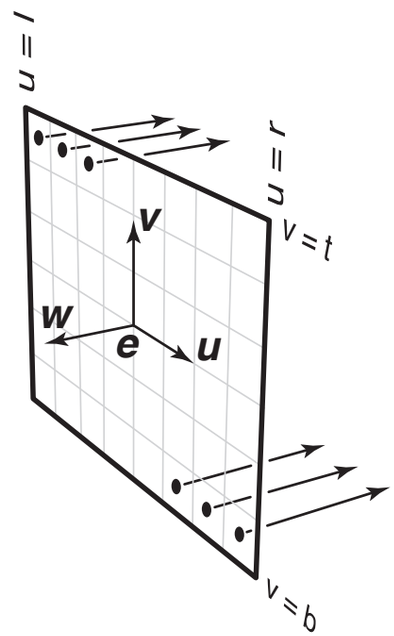




Orthographic Views

Computing Viewing Rays

- All rays will have the direction **-w**
- A viewpoint is not required but viewing rays can start from the plane containing camera
  - so that we know when an object is behind the screen



Orthographic Views

Computing Viewing Rays

- Viewing rays start on the plane defined by point e and vectors **u** and **v**

- Image dimensions
  - *l* and *r* are positions of the left and right edges of the image (measured from e along **u**), $l < 0 < r$
  - *b* and *t* are positions of the bottom and top edges of the image (measured from e along **v**), $b < 0 < t$

Orthographic Views

Computing Viewing Rays

- Pixel spacing for (n_x x n_y) image
  - Horizontal: $(r - l)/n_x$
  - Vertical: $(t - b)/n_y$
- Pixel position (i, j) in the raster image:

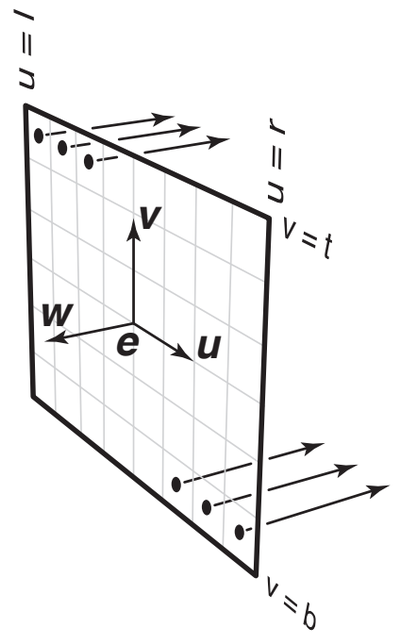
$$u = l + (r - l)(i + 0.5)/n_x,$$
$$v = b + (t - b)(j + 0.5)/n_y,$$

where (u, v) are the coordinates of pixel's position measured w. r. t. origin e and basis {**u**, **v**}

Orthographic Views

Computing Viewing Rays

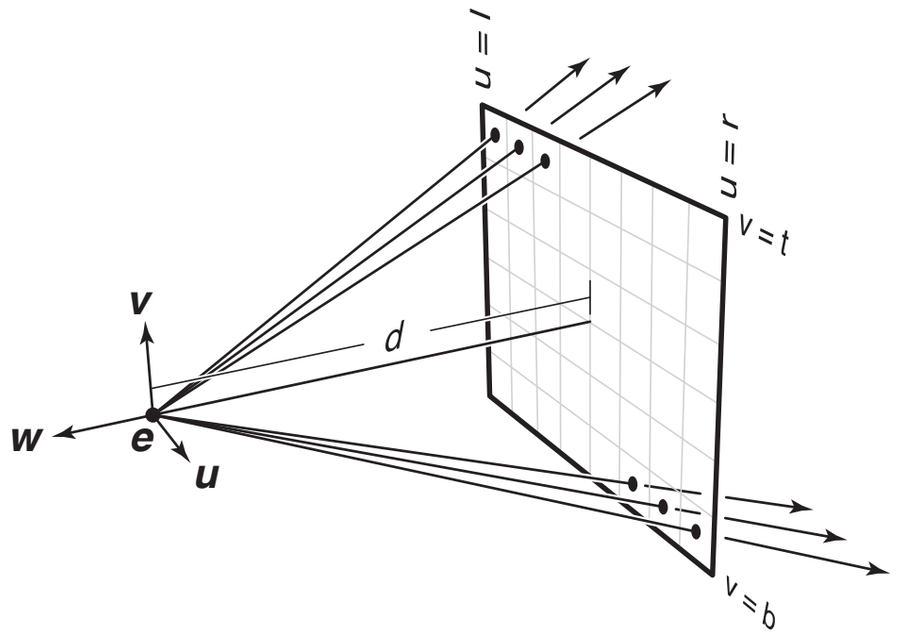
- Therefore, ray parameters are:
  - Origin: $e + u\,\mathbf{u} + v\,\mathbf{v}$
  - Direction: $-\mathbf{w}$



Perspective Views

Computing Viewing Rays

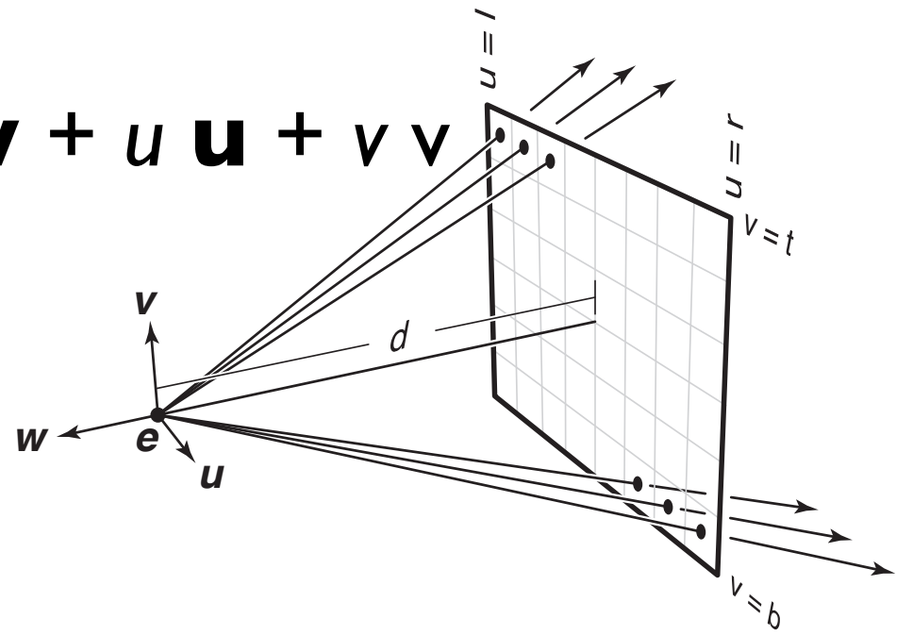
- All rays will have different directions for different pixels
- All rays will have the same origin, which is the viewpoint



Perspective Views

Computing Viewing Rays

- Image plane is positioned at a distance *d* from the camera origin e
- Ray parameters are:
  - Origin: e
  - Direction: $-d\,\mathbf{w} + u\,\mathbf{u} + v\,\mathbf{v}$



Ray-Object Intersection

Ray-Object Intersection

- Given a generated ray $\mathbf{p}(t) = \mathbf{e} + t\mathbf{d}$, find intersection (first hit) with objects in the scene such that $t > 0$

- Analytical objects:
  - Sphere
  - Triangle

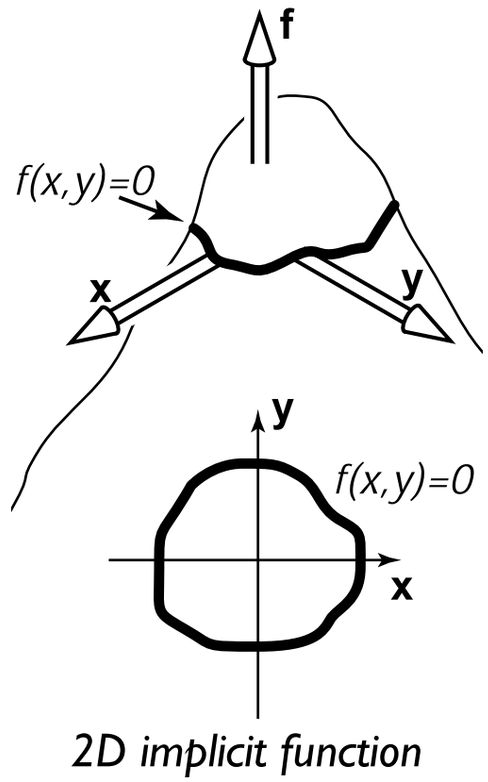
Ray-Sphere Intersection

- Given a ray:

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

and an implicit surface:

$$f(\mathbf{p}) = 0$$



2D implicit function

Seek values of t that solve the equation:

$$f(\mathbf{p}(t)) = 0 \text{ or } f(\mathbf{e} + t\mathbf{d}) = 0$$

Ray-Sphere Intersection

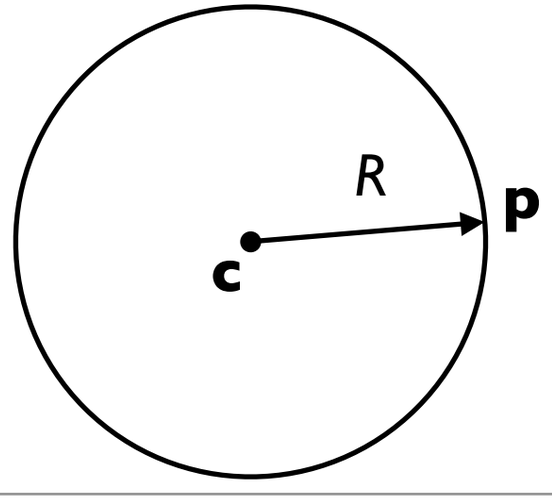
- A sphere with centre $\mathbf{c} = (x_c, y_c, z_c)$ and radius R:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 - R^2 = 0$$

or

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0$$

for any point $\mathbf{p}$ on the surface of the sphere



Ray-Sphere Intersection

- The ray $\mathbf{p}(t)$ will intersect the sphere if:

$$(\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) - R^2 = 0$$

- Rearranging, we get:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c})t + (\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2 = 0$$

- This gives the solution:

$$t = \frac{-\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) \pm \sqrt{(\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}))^2 - (\mathbf{d} \cdot \mathbf{d})((\mathbf{e} - \mathbf{c}) \cdot (\mathbf{e} - \mathbf{c}) - R^2)}}{(\mathbf{d} \cdot \mathbf{d})}$$

Ray-Sphere Intersection

- The normal vector at **p** on the implicit surface $f(\mathbf{p})$ is given by:

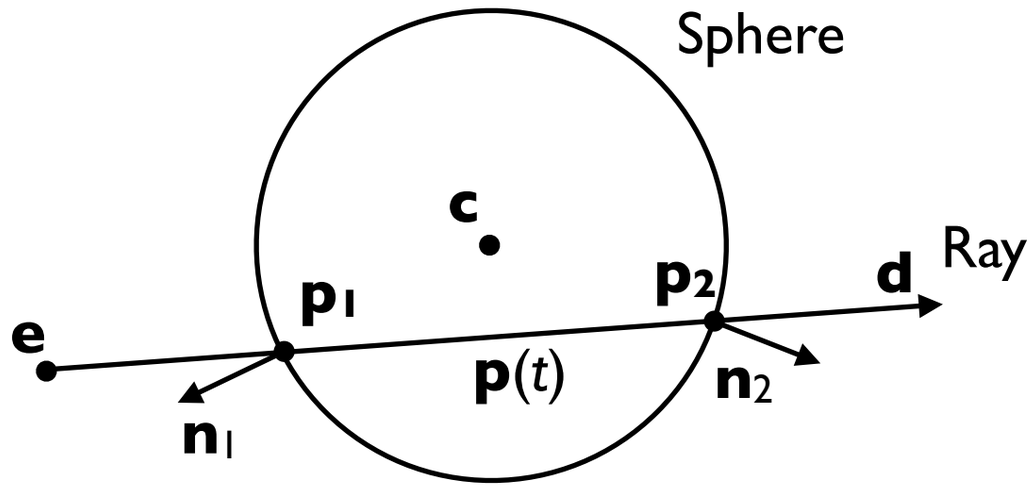
$$\mathbf{n} = \nabla f(\mathbf{p}) = \left(\frac{\partial f(\mathbf{p}}{\partial x}, \frac{\partial f(\mathbf{p}}{\partial y}, \frac{\partial f(\mathbf{p}}{\partial z} \right)$$

- For a sphere, the normal vector can be computed as:

$$\mathbf{n} = 2(\mathbf{p} - \mathbf{c})$$

and

$$\hat{\mathbf{n}} = \frac{(\mathbf{p} - \mathbf{c})}{R}$$

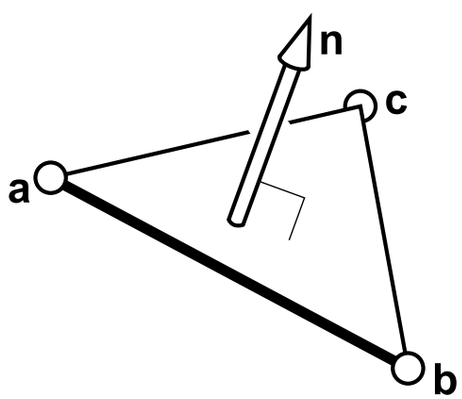


Ray-Triangle Intersection

- Intersect a ray with a parametric surface

$$\left.\begin{aligned} x_e + t x_d &= f(u,v) \\ y_e + t y_d &= g(u,v) \\ z_e + t z_d &= h(u,v) \end{aligned}\right\} = \text{ or, } \mathbf{e} + t\mathbf{d} = \mathbf{f}(u,v)$$

- Three equations and three unknowns (t, u, and v)
- For a parametric plane, the parametric surface can be represented in terms of any three points in the plane



Ray-Triangle Intersection

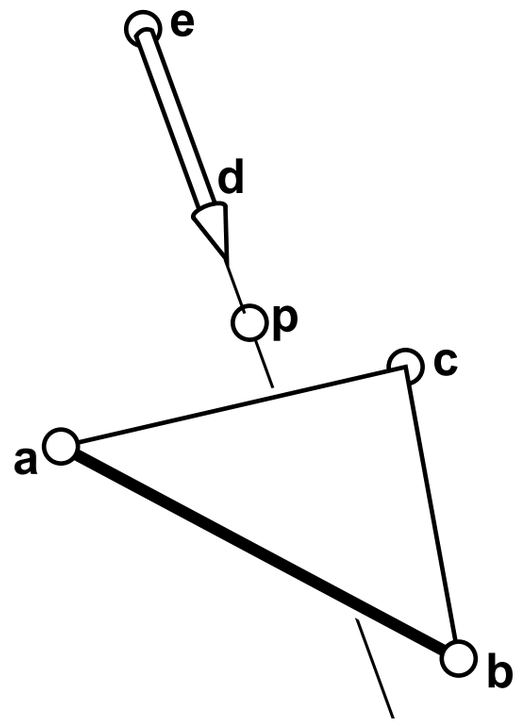
- Utilize barycentric coordinates for ray-triangle test
- For a triangle with vertices a, b, and c intersection will occur when

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

- Intersection is inside the triangle iff

$$\beta > 0, \gamma > 0, \text{ and } \beta + \gamma < 1$$

Otherwise the ray has hit the plane outside the triangle



Ray-Triangle Intersection

- To solve for $t, \beta,$ and γ, we can write

$$x_e + tx_d = x_a + \beta(x_b - x_a) + \gamma(x_c - x_a),$$
$$y_e + ty_d = y_a + \beta(y_b - y_a) + \gamma(y_c - y_a),$$
$$z_e + tz_d = z_a + \beta(z_b - z_a) + \gamma(z_c - z_a).$$

- This can be written in matrix form as:

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

Ray-Triangle Intersection

- Use *Cramer's rule* to solve 3x3 linear system

$$\beta = \frac{\begin{vmatrix} x_a - x_e & x_a - x_c & x_d \\ y_a - y_e & y_a - y_c & y_d \\ z_a - z_e & z_a - z_c & z_d \end{vmatrix}}{|\mathbf{A}|}$$

$$\gamma = \frac{\begin{vmatrix} x_a - x_b & x_a - x_e & x_d \\ y_a - y_b & y_a - y_e & y_d \\ z_a - z_b & z_a - z_e & z_d \end{vmatrix}}{|\mathbf{A}|}$$

$$t = \frac{\begin{vmatrix} x_a - x_b & x_a - x_c & x_a - x_e \\ y_a - y_b & y_a - y_c & y_a - y_e \\ z_a - z_b & z_a - z_c & z_a - z_e \end{vmatrix}}{|\mathbf{A}|}$$

$$, \mathbf{A} = \begin{bmatrix} x_a - x_b & x_a - x_c & x_d \\ y_a - y_b & y_a - y_c & y_d \\ z_a - z_b & z_a - z_c & z_d \end{bmatrix}$$

Triangle normal?

Ray-Polygon Intersection

- Given m vertices $\mathbf{p}_1$ through $\mathbf{p}_m$ and surface normal $\mathbf{n}$
- Compute intersection point between ray $\mathbf{e} + t\mathbf{d}$ and plane containing polygon $(\mathbf{p} - \mathbf{p}_1) \cdot \mathbf{n} = 0$

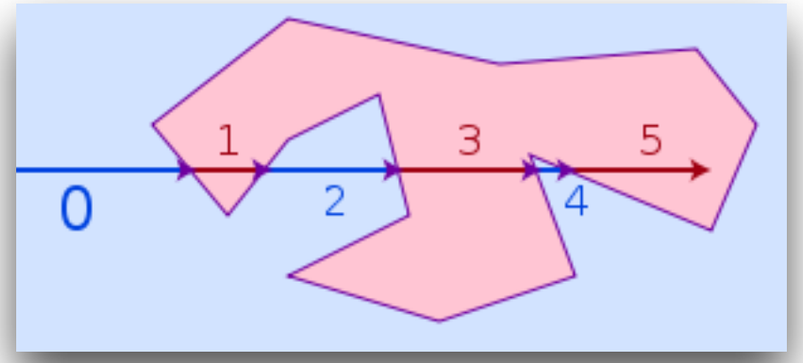
$$t = \frac{(\mathbf{p}_1 - \mathbf{e}) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

- If $\mathbf{p}$ (computed from t) is inside the polygon, then the ray hits it.

Point-in-Polygon Test

Ray-Polygon Intersection

- To test whether a point p is inside a polygon, use the Crossing number algorithm
- send a 2D ray out from p and count the number of intersections between the ray and the polygon boundary
- If intersection count is odd, point is inside the polygon



Source: Wikipedia, Point in polygon

Point-in-Polygon Test

Ray-Polygon Intersection

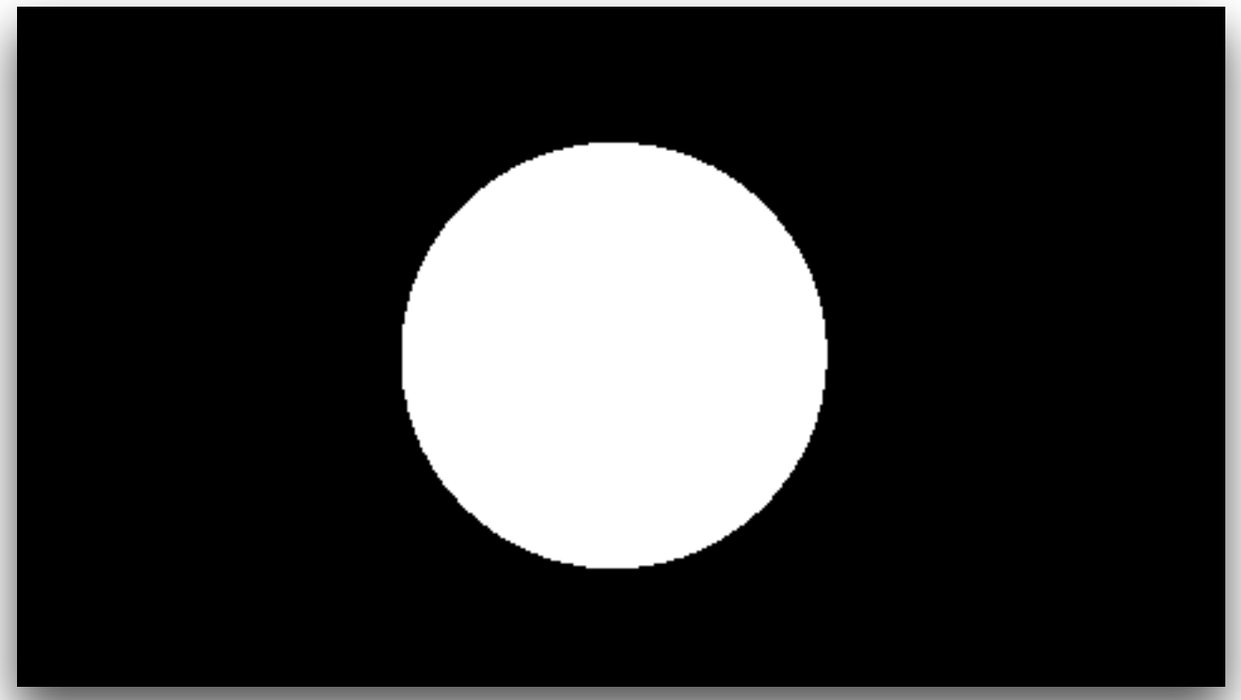
- Easily done by projecting polygon onto one of the coordinate planes: *xy*, *yz* or *zx*
- Choose a plane based on minimum dot product with the normal:

$$\arg\min_{k \in \{x,y,z\}} \hat{\mathbf{n}} \cdot \pi_k$$

- Use a test ray parallel to one of the coordinate axes

Ray Tracer

```
Surface s = new Sphere(centre, radius)
for y=0 to n_y do
    for x=0 to n_x do
        ray = camera.getRay(x, y)
        [hitSurface, t] = s.intersect(ray, 0, +∞)
        if hitSurface ≠ null then
            image.set(x, y, white)
```



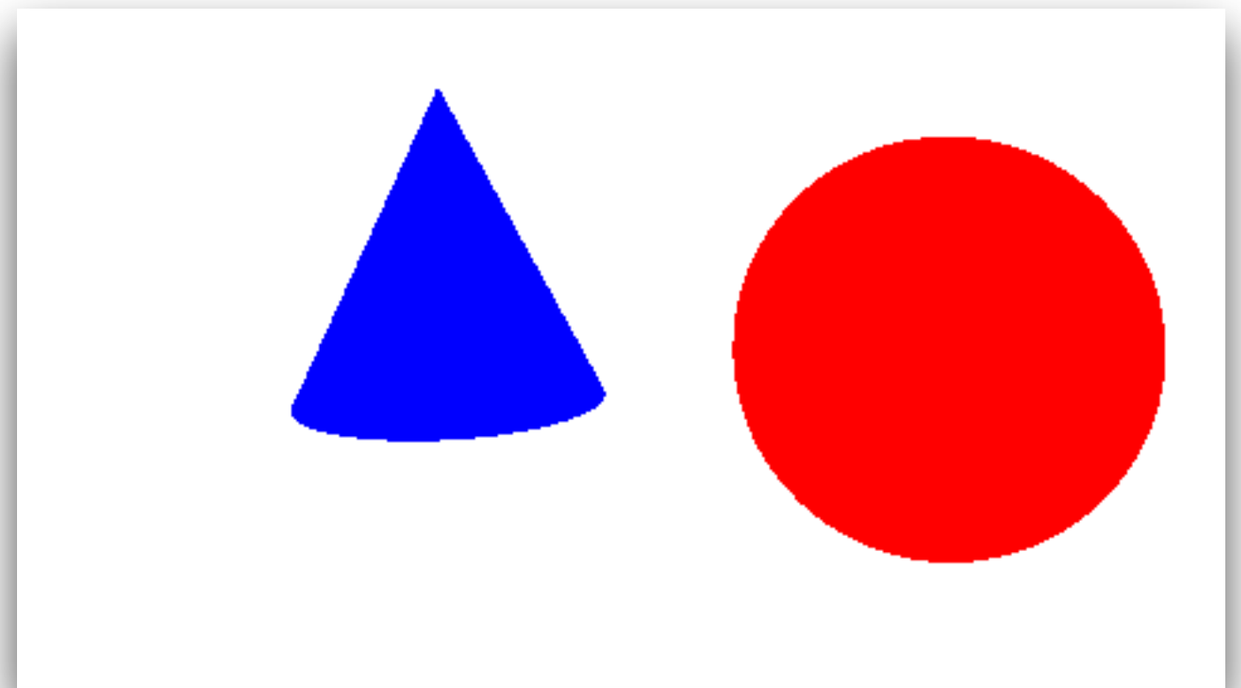
Intersecting Group of Objects

- A scene typically contains many objects
- We are interested only in the closest object intersection to the camera along the ray

```
Procedure Group.intersect(ray, t_min, t_max)
    t_best = +∞
    firstSurface = null
    for each surface in surfaceList do
        [hitSurface, t] = surface.intersect(ray, t_min, t_best)
        if hitSurface ≠ null and t < t_best then
            t_best = t
            firstSurface = hitSurface
    return [hitSurface, t_best]
```

Ray Tracer

```
for y=0 to n_y do
    for x=0 to n_x do
        ray = camera.getRay(x, y)
        color = scene.trace(ray, 0, +∞)
        image.set(x, y, color)


...


Procedure Scene.trace(ray, t_min, t_max)
[surface, t ] = surfaces.intersect(ray, t_min, t_max)
if surface ≠ null then
    return surface.color()
else
    return black
```



Shading

Shading

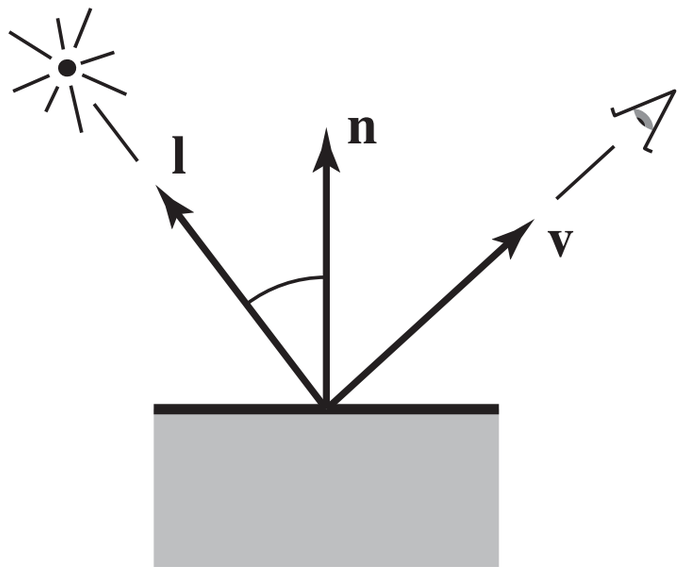
- Pixel value for a visible surface point is evaluated using a *shading model*
- Shading models are designed to capture interaction of light with matter
- Approximations to physically based lighting are used to mimic the actual behavior and speed up shading

Lambertian Shading

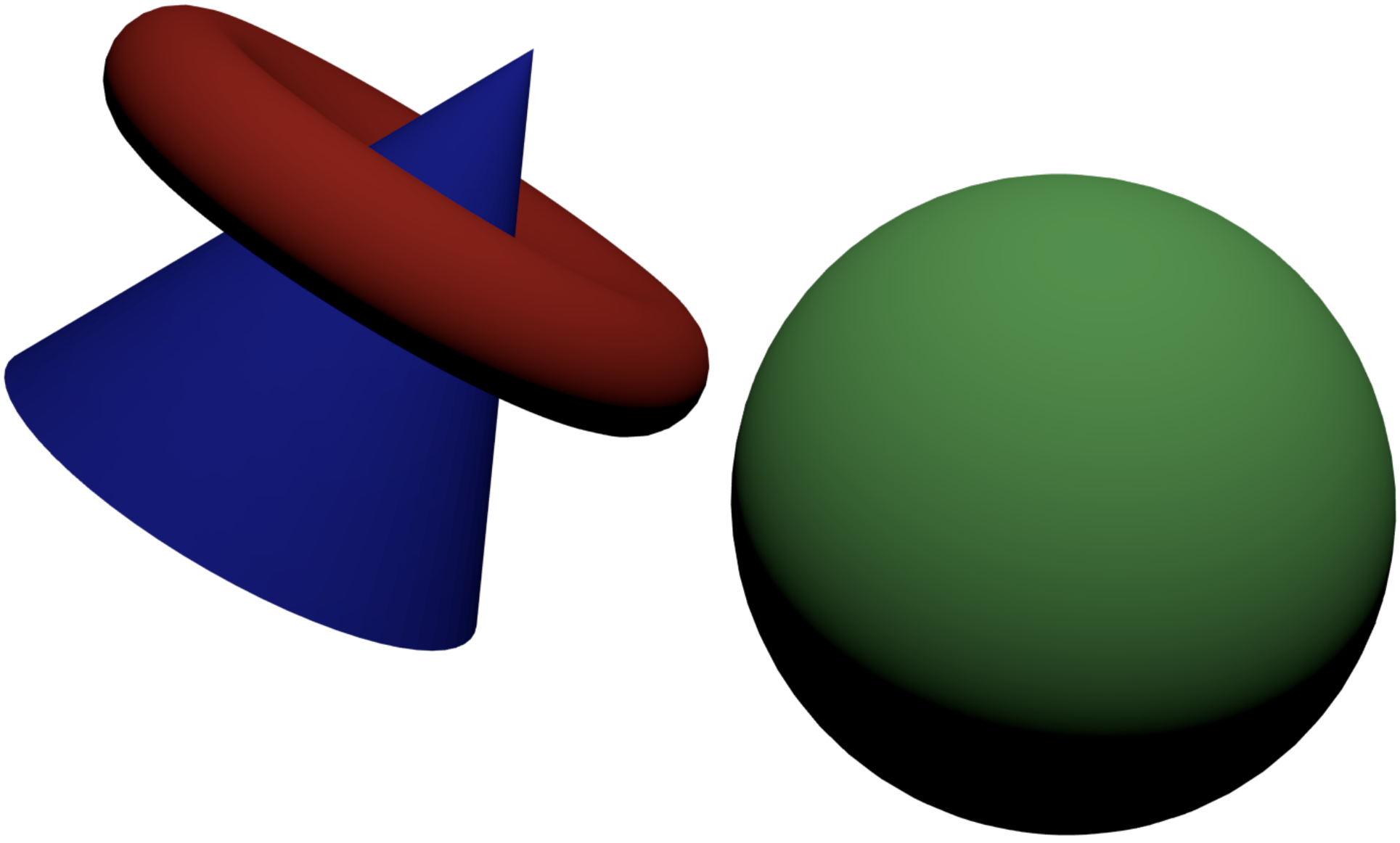
- Simplest shading model by Lambert (18th century)
- The amount of light energy from a light source that falls on an area of surface depends on the angle of the surface to the light
- *Lambertian shading model*:

$$L = k_d I \max(0, \mathbf{n} \cdot \mathbf{l})$$

L : pixel colour,
k_d: diffuse coefficient,
I : intensity of light source, and
$\mathbf{l}$: unit vector pointing toward the light



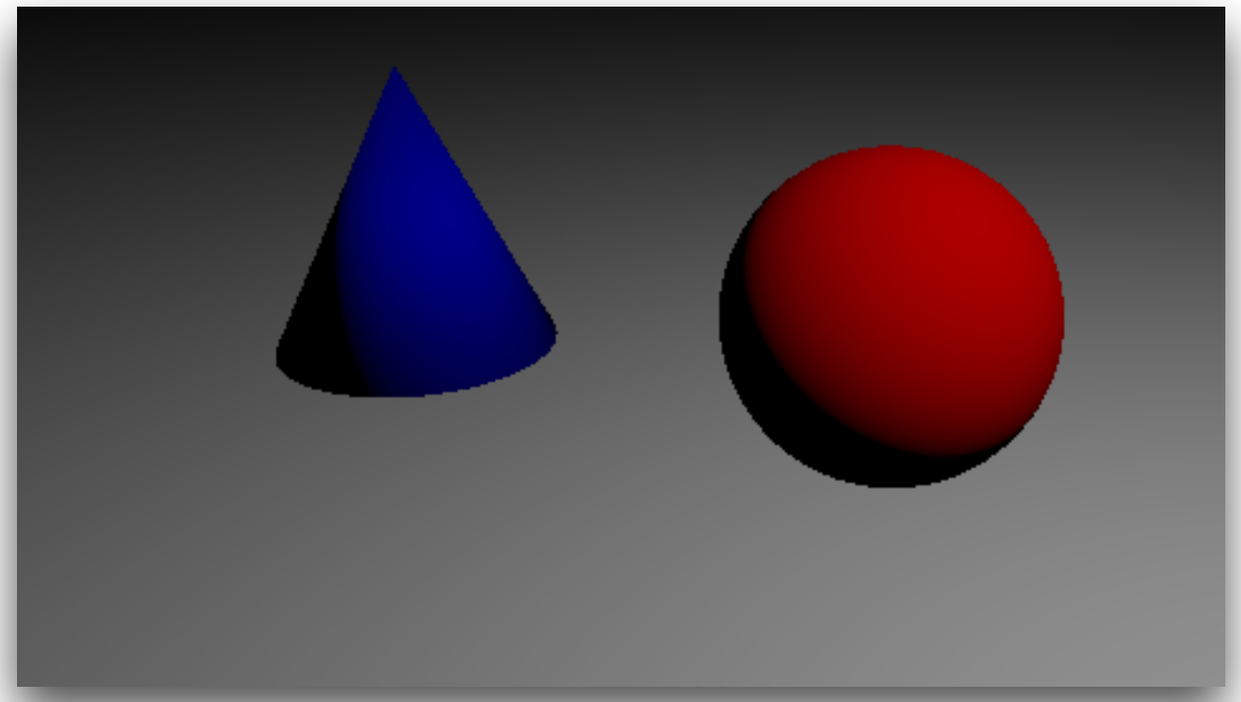
Lambertian Shading



Lambertian shading is view independent!

Ray Tracer

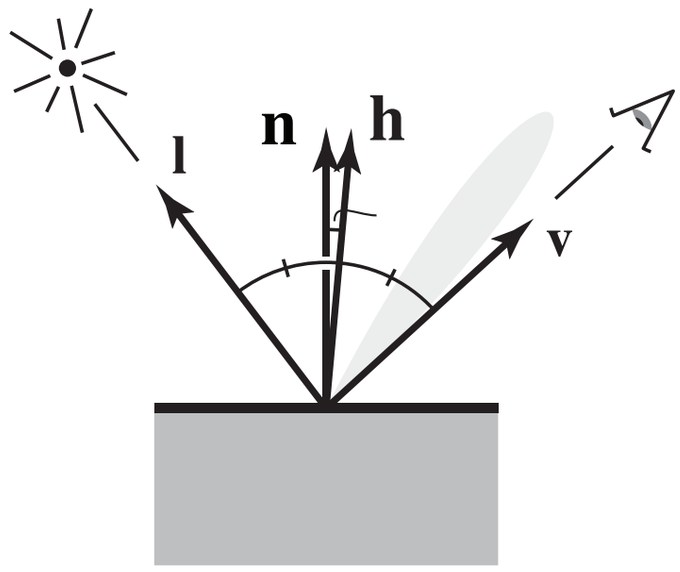
```
Procedure Scene.trace(ray, tmin,tmax)
   [surface, t ] = surfaces.intersect(ray, tmin,tmax)
   if surface ≠ null then
      point = ray.evaluate(t)
      normal = surface.getNormal(point)
      return surface.shade(ray, point, normal, light)
   else
      return backgroundColor

...

Procedure Surface.shade(ray, point,
normal, light)
   v = -normalize(ray.direction)
   l = normalize(light.position - point)
   //Compute shading
```



Blinn-Phong Shading

- Lambertian surfaces leave a matte appearance
- The Blinn-Phong model adds specular highlights that appear to move as the viewpoint changes
- Modification of the *Phong model* where the specular highlights depend on the angle between reflected light vector and the view vector:

$$\propto (\mathbf{r} \cdot \mathbf{v})^{\alpha}$$



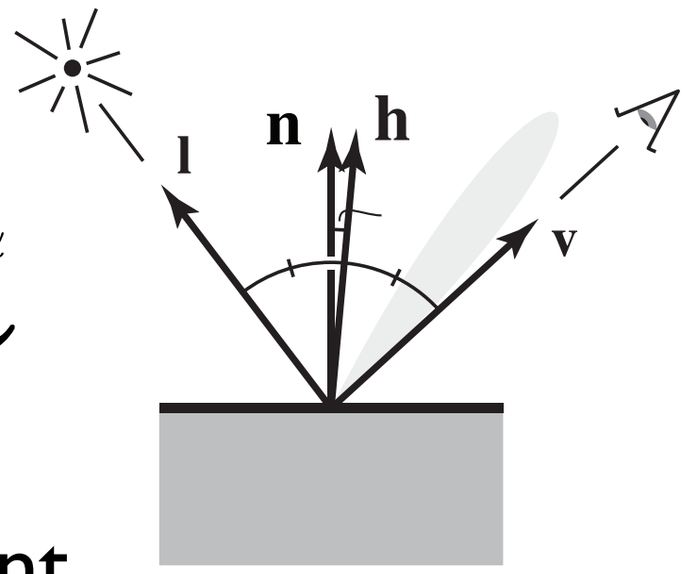
Blinn-Phong Shading

- Blinn-Phong model uses the half vector **h**

$$\mathbf{h} = \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|}$$

- The dot product $\mathbf{h} \cdot \mathbf{n}$ measures degree of specularity
- *Blinn-Phong shading model:*

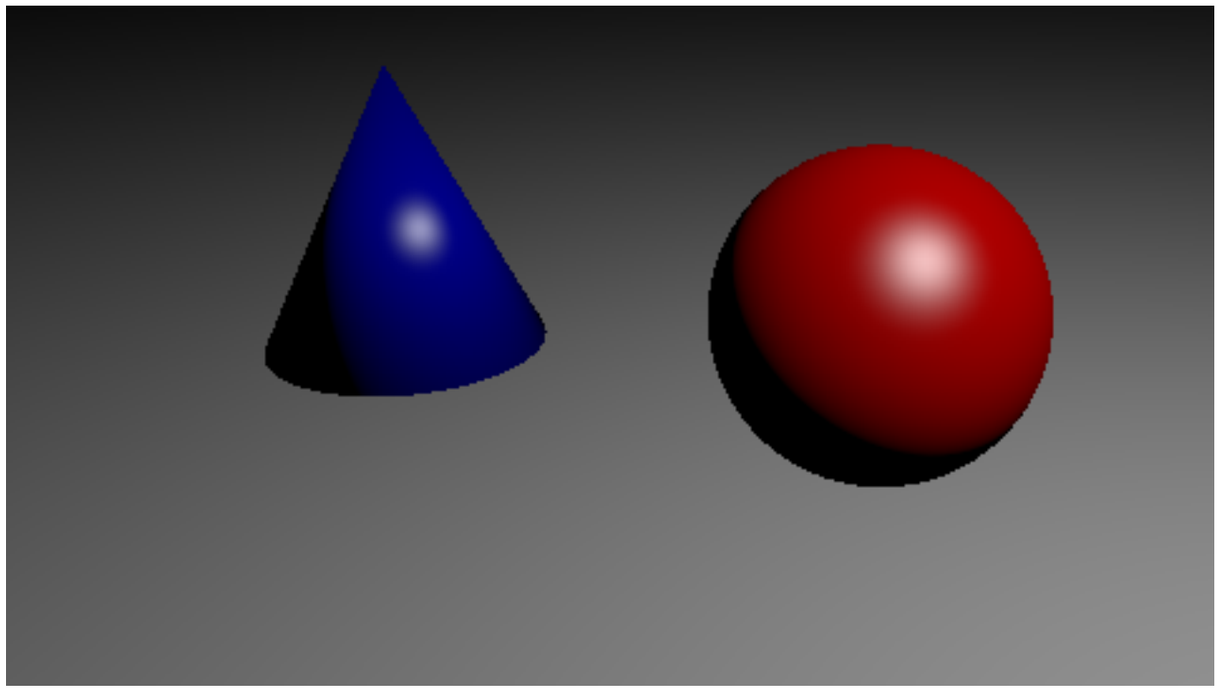
$$L = \underbrace{k_d I \max(0, \mathbf{n} \cdot \mathbf{l})}_{\text{Diffuse component}} + \underbrace{k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^{\alpha}}_{\text{Specular component}}$$



k_s: specular coefficient, α: phong coefficient

Ray Tracer

```
Procedure Scene.trace(ray, tmin, tmax)
   [surface, t ] = surfaces.intersect(ray, tmin, tmax)
   if surface ≠ null then
      point = ray.evaluate(t)
      normal = surface.getNormal(point)
      return surface.shade(ray, point, normal, light)
   else
      return backgroundColor

...

Procedure Surface.shade(ray, point,
normal, light)
   v = -normalize(ray.direction)
   l = normalize(light.position - point)
   h = computeHalfVector(v, l)
   //Compute shading:
   //diffuse and specular
```



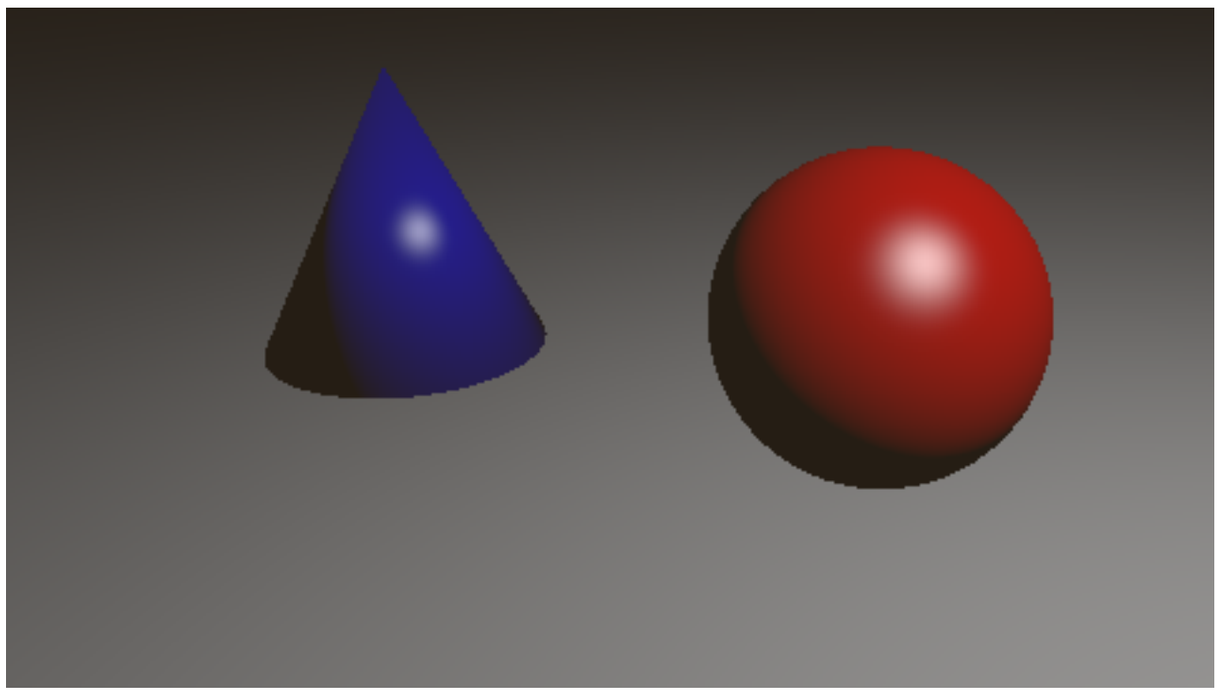
Ambient Shading

- Surfaces that receive no direct illumination are rendered completely black
- A crude heuristic to simulate the effect of indirect illumination is to add a constant *ambient* component to the shading model
- *Shading model:*

$$L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^{\alpha}$$

Ray Tracer

- Add ambient light component in shading computation



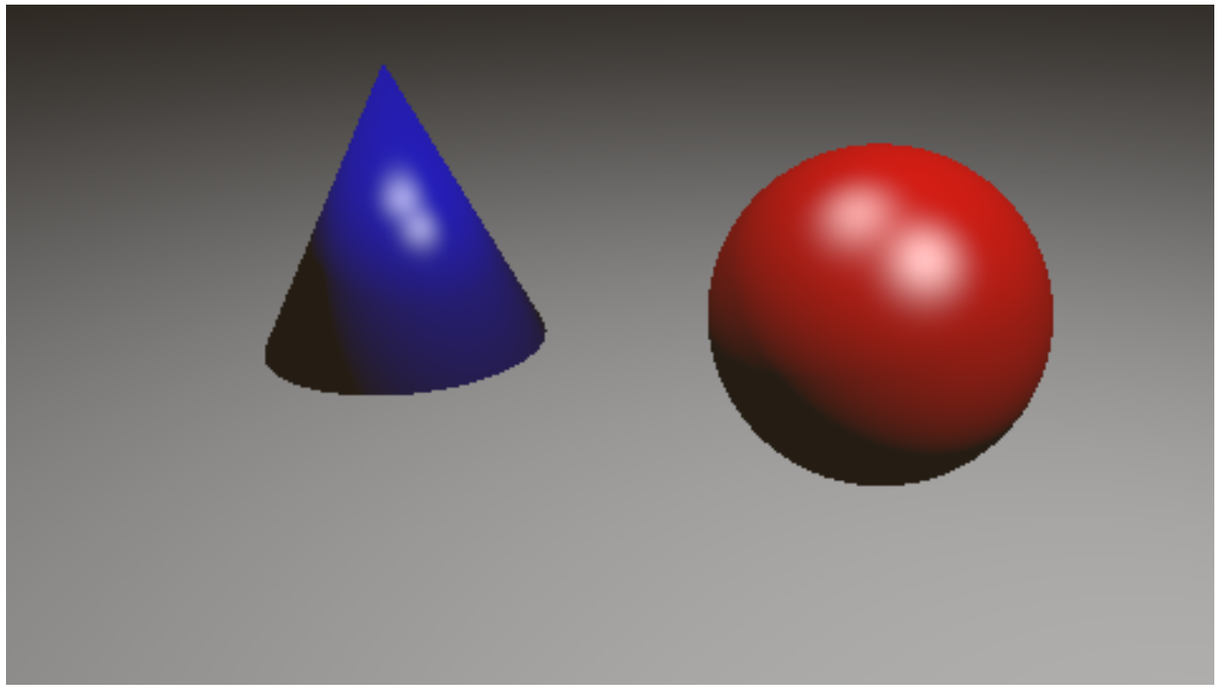
Multiple Light Sources

- Useful property of light: *superposition*
  the effect caused by more than one light source is simply the sum of effects of individual light sources

$$L = k_a I_a + \sum_{i=1}^{N} [k_d I_i \max(0, \mathbf{n} \cdot \mathbf{l_i}) + k_s I_i \max(0, \mathbf{n} \cdot \mathbf{h_i})^{\alpha}]$$

Ray Tracer

- Add effect of multiple light sources
- Limit RGB values in the range [0, 1] after accumulating



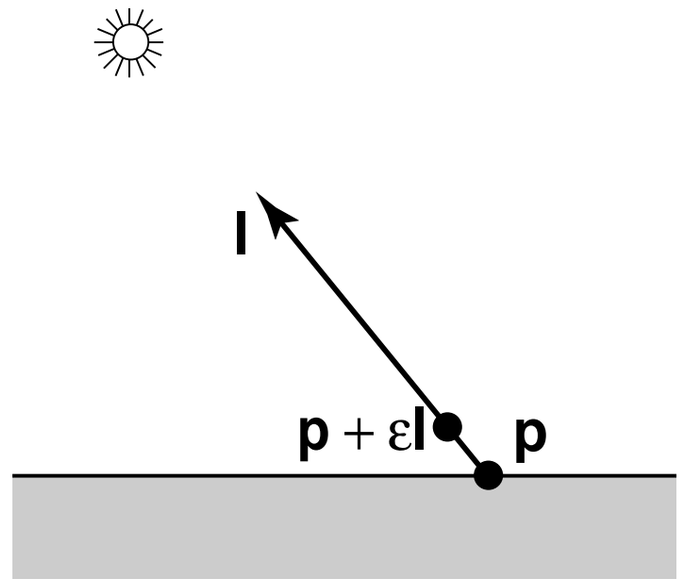
Casting Shadows

- Surface is illuminated if nothing blocks its view of light
- Shoot out a *shadow ray* from the surface hit point toward a light source
- As many shadow rays as there are lights
- Intersect the shadow ray with the scene
  - Self intersections possible for concave object
- Ideally test: $t \in [0, \infty)$

Casting Shadows

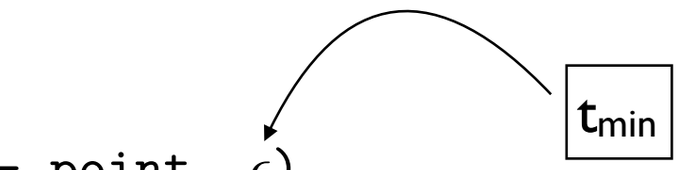
- Intersect the shadow ray with the scene
  - Self intersections possible for concave object
- To account for numerical imprecision

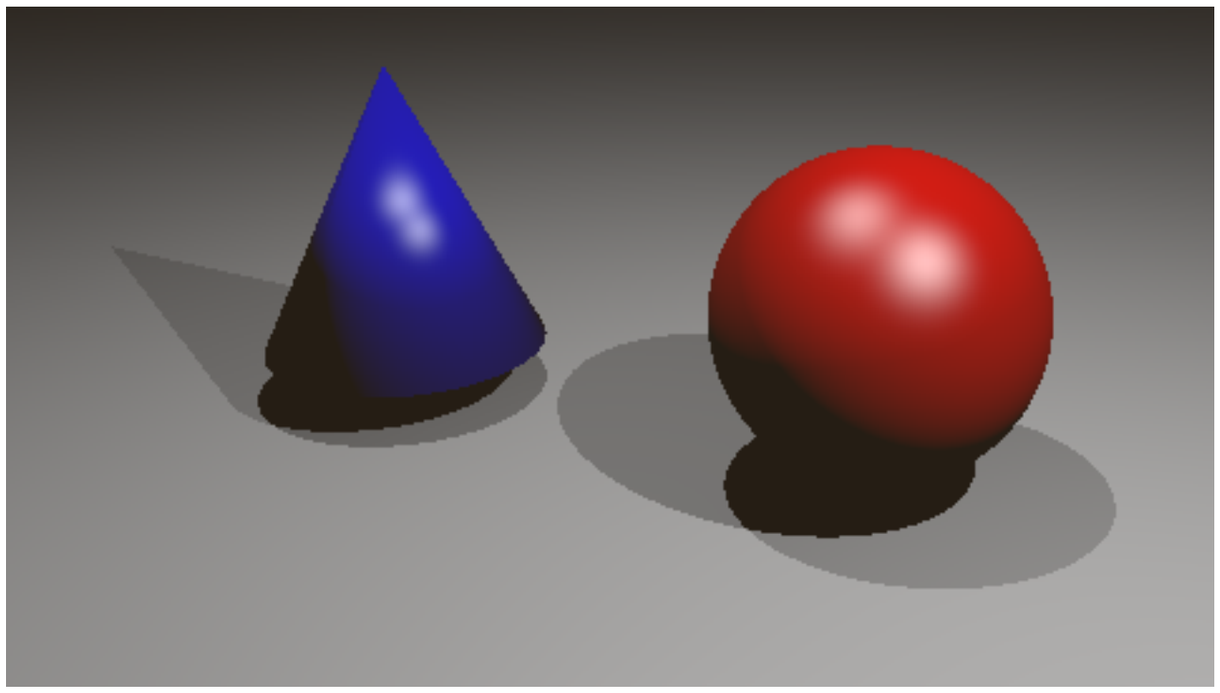
test: $t \in [\epsilon, \infty], \epsilon$ **small**



Ray Tracer

```
Procedure Surface.shade(ray, point,
normal, light)
    ShadowRay = (point, light.position - point, ϵ)
    if shadowRay not blocked then
        v = -normalize(ray.direction)
        l = normalize(light.position - point)
        h = computeHalfVector(v, l)
        //Compute shading:
        //Ambient, diffuse and specular
        return shadingColor;
    else
        //Compute ambient shading
        return ambientColor;
```





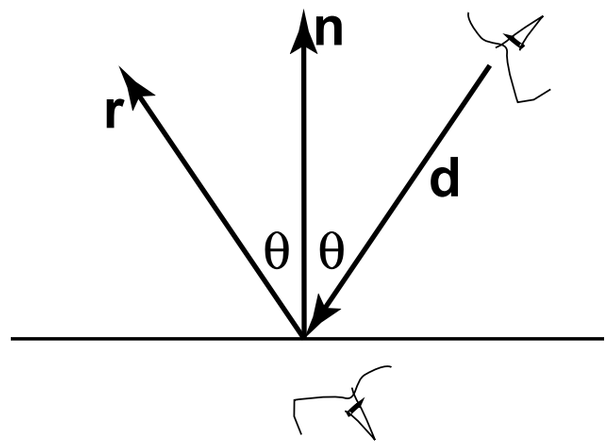
Specular Reflections

- Mirror reflections can be added by shading reflected rays

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

- Some energy is lost during reflection of light from surfaces (loss: wavelength dependent)
- Following recursion adds reflection

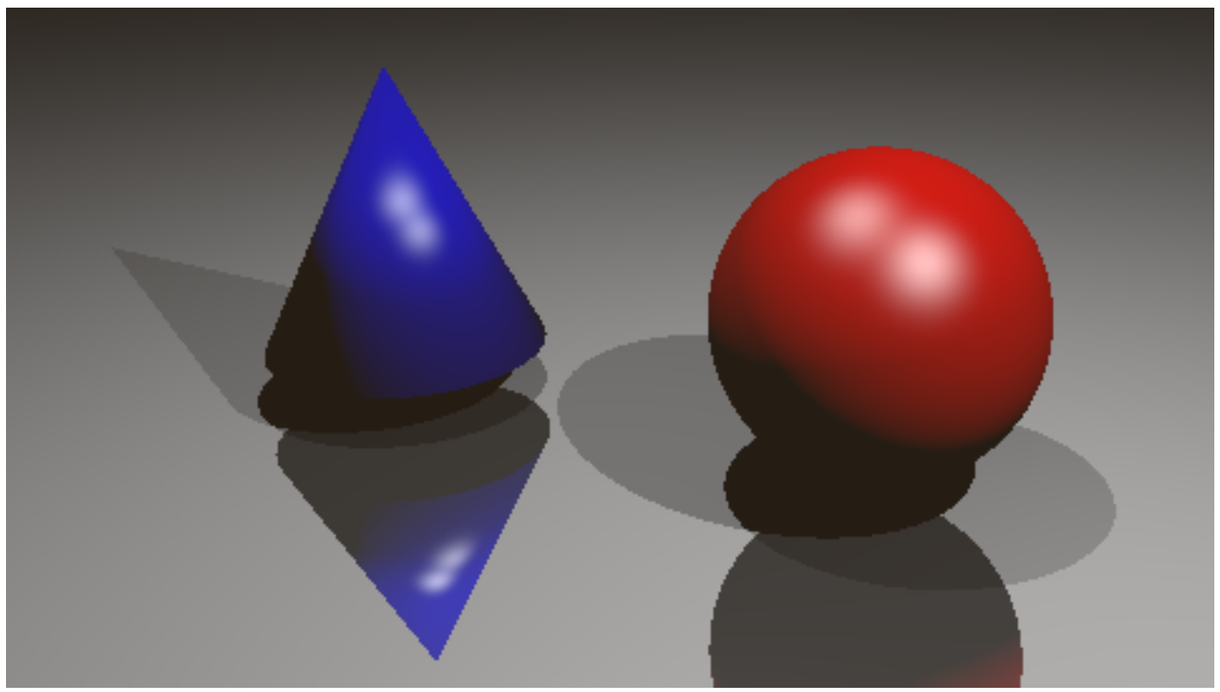
$$\text{color } c = c + k_m \text{raycolor}(\mathbf{p} + t\mathbf{r}, \epsilon, \infty)$$



k_m: specular RGB color for mirror reflection

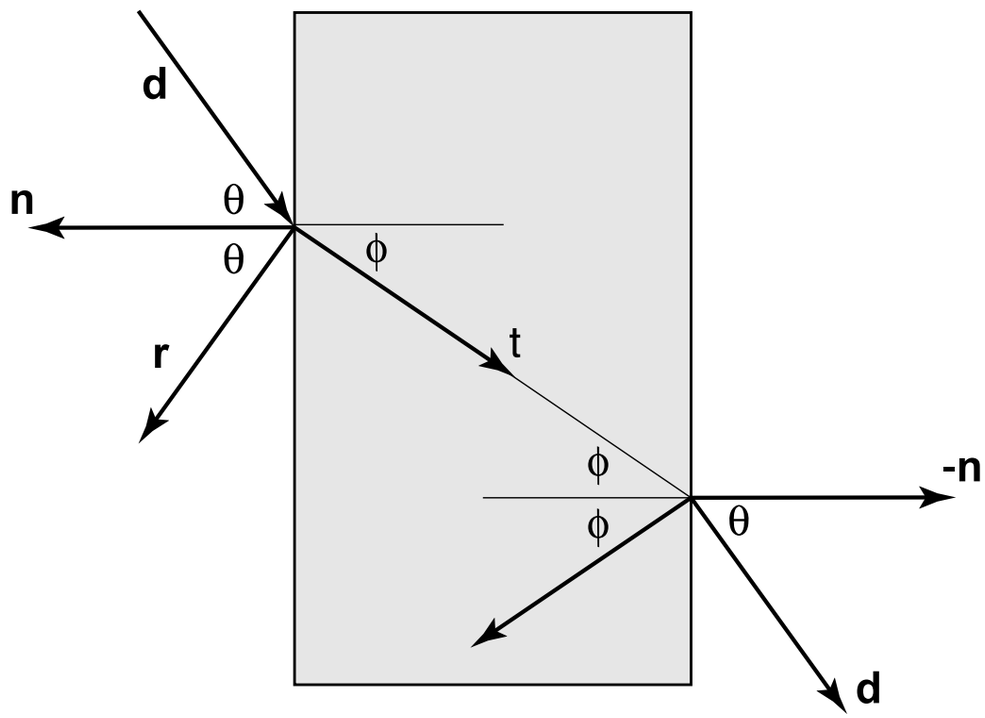
Ray Tracer

- Trace reflected rays for materials that are specular
- Limit recursive reflection for practical running time



Transparency and Refraction

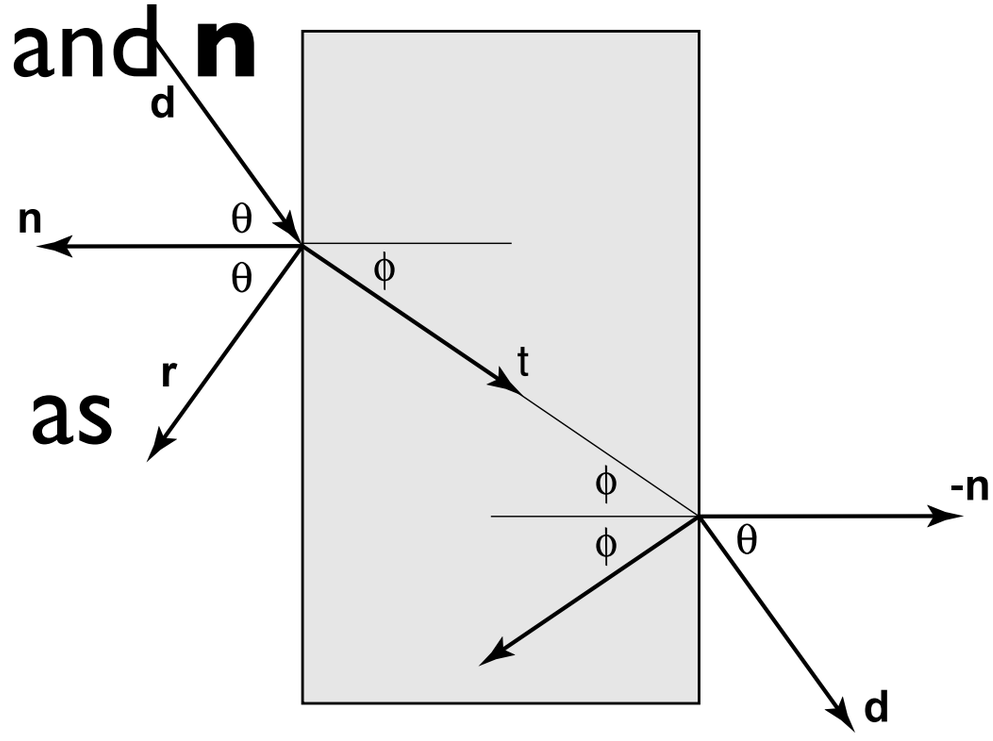
- *Dielectric*: a transparent material that refracts light
- Light bends when it travels from a medium with refractive index n to one one with n_t
- Snell's law: $n \sin\theta = n_t \sin\phi$



Transparency and Refraction

- To compute transmitted ray **t**, we define it in terms of orthonormal basis **b** and **n**

$$\mathbf{t} = \sin\phi\mathbf{b} + \cos\phi\mathbf{n}$$



- Incoming ray **d** can be written as

$$\mathbf{d} = \sin\theta\mathbf{b} - \cos\theta\mathbf{n}$$

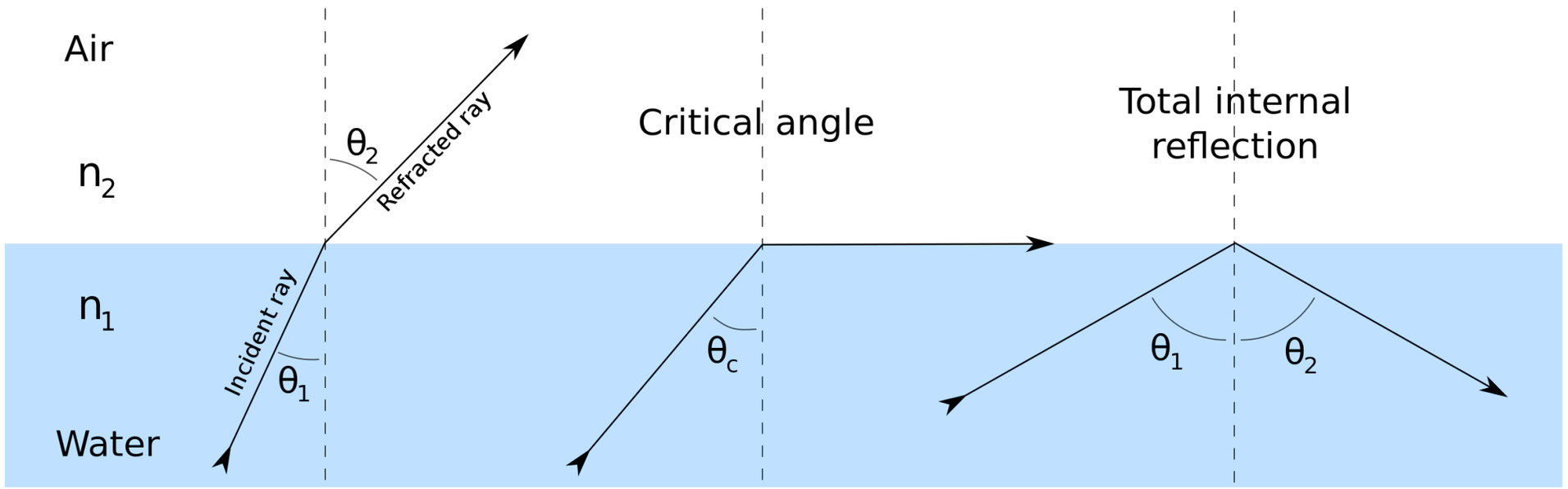
- Solving for **t**:

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

Total Internal Reflection

Refraction

- Complete reflection of light occurs when it travels from a denser to rarer medium striking the interface at an angle greater than the *critical angle*



Source: Wikipedia, Total Internal Reflection

Total Internal Reflection

Refraction

- Critical angle can be computed as:

$$1 - \frac{n^2(1 - (\mathbf{d} \cdot \mathbf{n})^2)}{n_t^2} < 0$$

$$\implies \frac{n^2}{n_t^2}(1 - (\mathbf{d} \cdot \mathbf{n})^2) > 1$$

$$\implies (\mathbf{d} \cdot \mathbf{n})^2 < \left(1 - \frac{n_t^2}{n^2}\right)$$

$$\implies \cos\theta = \sqrt{1 - \frac{n_t^2}{n^2}}$$

Note: $n > n_t$

Source: Wikipedia, Total Internal Reflection

Schlick Approximation

Refraction

- The reflectivity of a dielectric varies with the incident angle as per the *Fresnel Equations*
- Schlick approximation may be used instead

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5$$

R_0 is reflectance at normal incidence

$$R_0 = \left(\frac{n_t - 1}{n_t + 1}\right)^2$$

θ : angle in rarer medium (larger of the two angles)

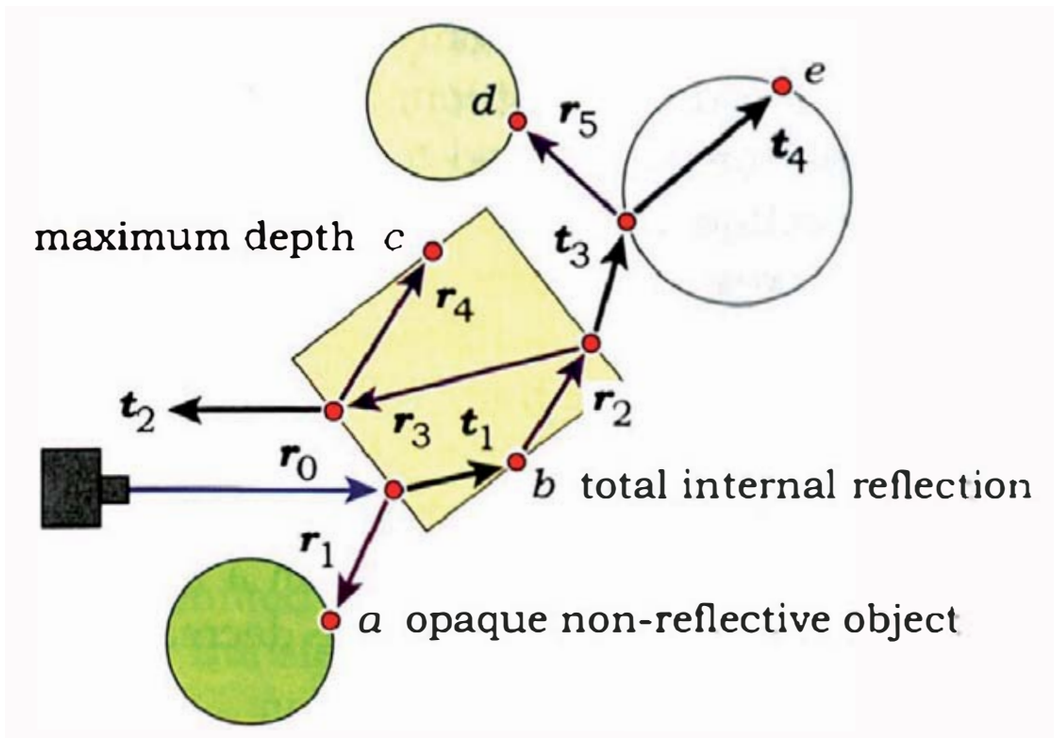
Beer's Law

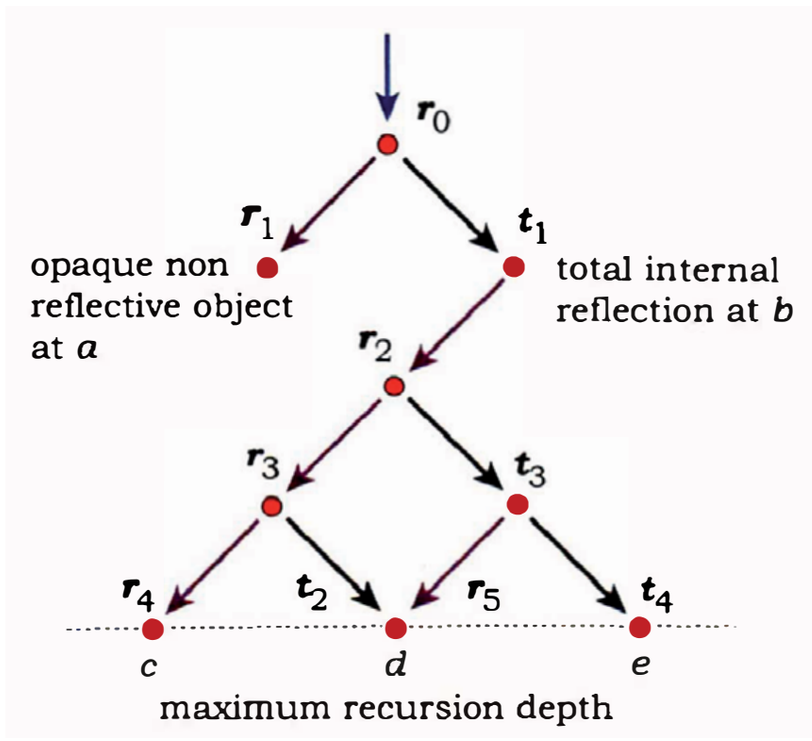
- For homogeneous impurities in a dielectric, a light carrying ray's intensity attenuates as per *Beer's law*
- Loss of intensity in the medium: $\mathrm{d}I = -CI\mathrm{d}x$
- Solution: $I = k\exp(-Cx) + k'$ with boundary conditions:

$$I(s) = I(0)e^{-\ln(a)s}$$

s: distance from the interface
a: attenuation constant
(attenuation after unit distance)

Ray Tracer

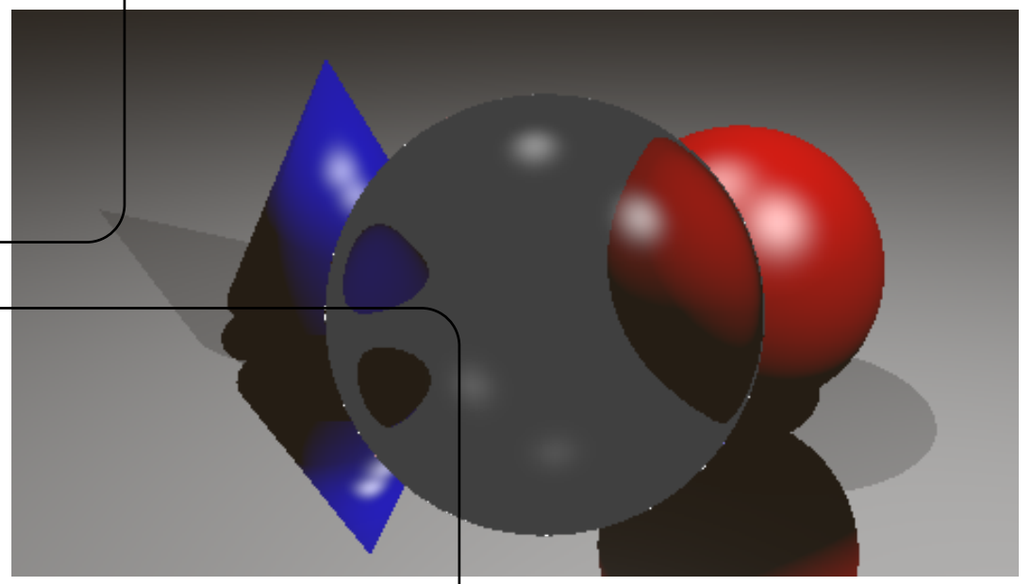




- Two rays are generated at dielectric surfaces
- No new rays generated at a (diffuse surface)
- Only one ray generated at b (total internal reflection)
- No new rays generated at c, d, and e (maximum recursion depth reached)

```
Procedure Surface.shade(ray, point, normal, light)
    if point is on dielectric then
        r = reflect(ray, normal)
        if dot(ray, normal) < 0 then
            // Are we entering a dielectric?
            refract(ray, normal, η, t)
            c = -dot(ray, normal)
            k_r = k_g = k_b = 1
        else
            // Apply Beer's law
            k_r = exp(-a_r t)
            k_g = exp(-a_g t)
            k_b = exp(-a_b t)
            if refract(ray, -normal, 1/η, t) then
                c = dot(t, normal)
            else
                return k * Scene.trace (Ray(point, r))
        R_0 = (η - 1)^2/(η + 1)^2
        R = R_0 + (1 - R_0)(1 - c)^5
        return k(R * Scene.trace (Ray(point, r)) +
            (1 - R) * Scene.trace (Ray(point, t)))
```

Entering the dielectric



Beer's law

Exiting the dielectric

Total Internal Reflection

Schlick's Approx.

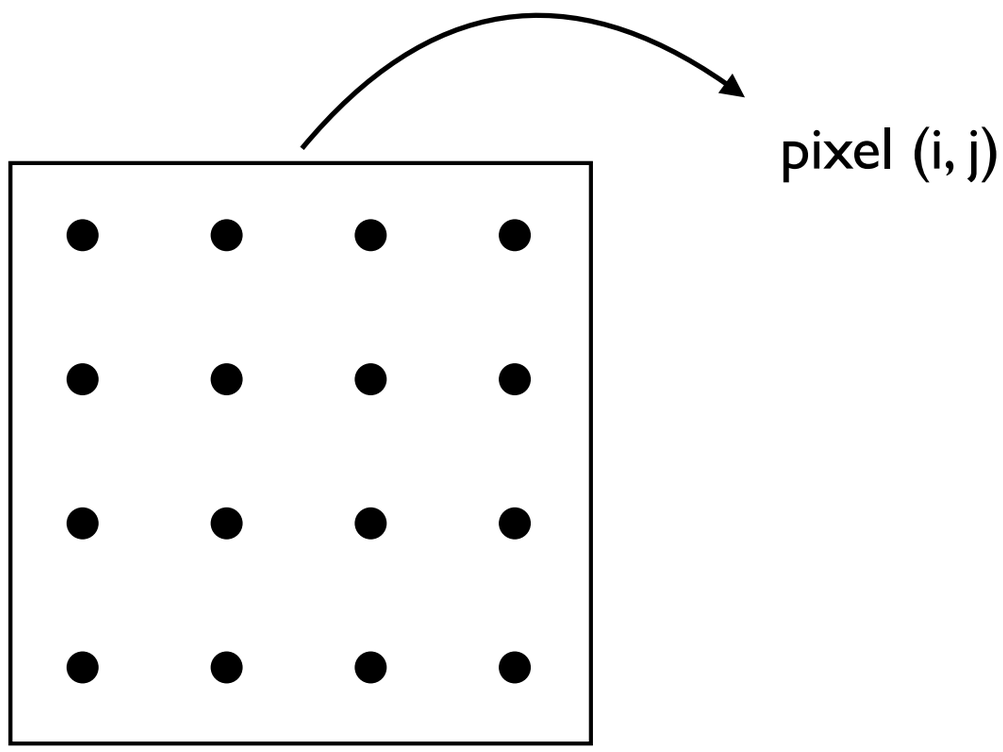
Antialiasing

- Simple way to antialias an image is to average color in a small neighbourhood of the pixel
- Two approaches to get smooth results in ray tracing:
  - regular sampling
  - random sampling
  - hybrid sampling (jittering)

Regular Sampling

Antialiasing

- Sub-sample a pixel in regular grid fashion
  - Shoot out rays passing through sub-pixels sampled regularly



Regular Sampling

Antialiasing

- Sub-sample a pixel in regular grid fashion
- Replace:

```
for each pixel (i, j) do
    c_ij = ray-color(i + 0.5, j + 0.5)
```

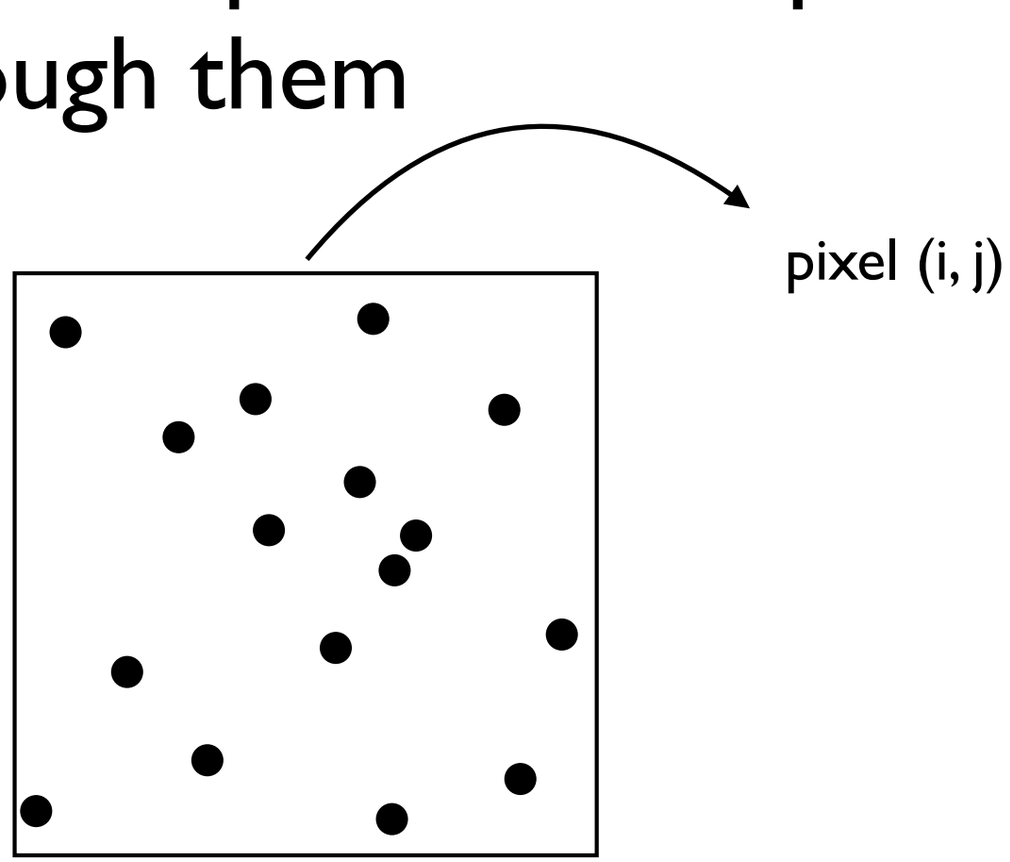
by:

```
for each pixel (i, j) do
    c = 0
    for p=0 to n-1 do
        for q=0 to n-1 do
            c = c + ray-color(i + (p + 0.5)/n,
                              j + (q + 0.5)/n)
    c_ij = c/n^2
```

Random Sampling

Antialiasing

- Moiré patterns can arise from regular sampling
- Select n^2 random samples within a pixel and trace rays passing through them



Random Sampling

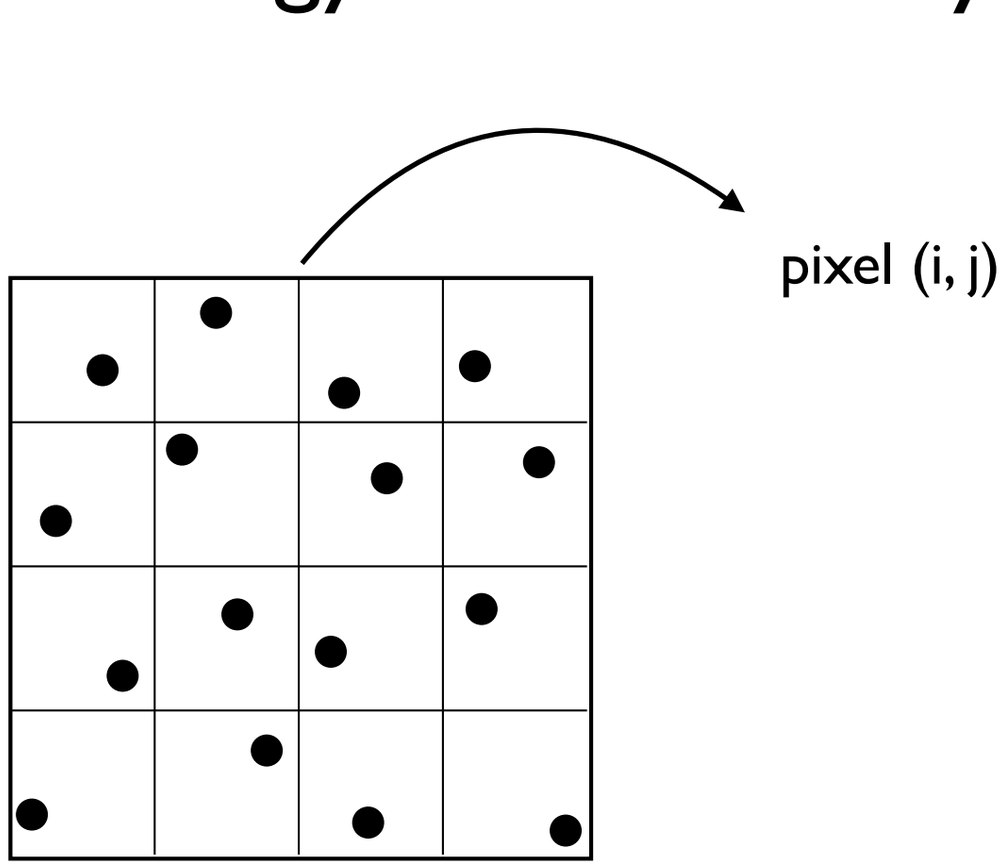
Antialiasing

- Select n^2 random samples within a pixel and trace rays passing through them

```
for each pixel (i, j) do
    c = 0
    for p=0 to n^2 do
        // Choose random number ξ ∈ [0, 1)
        c = c + ray-color(i + ξ, j + ξ)
    c_ij = c/n^2
```

Jittering

- Noise from random sampling can be noticeable
- Choose a hybrid strategy that randomly perturbs a regular grid



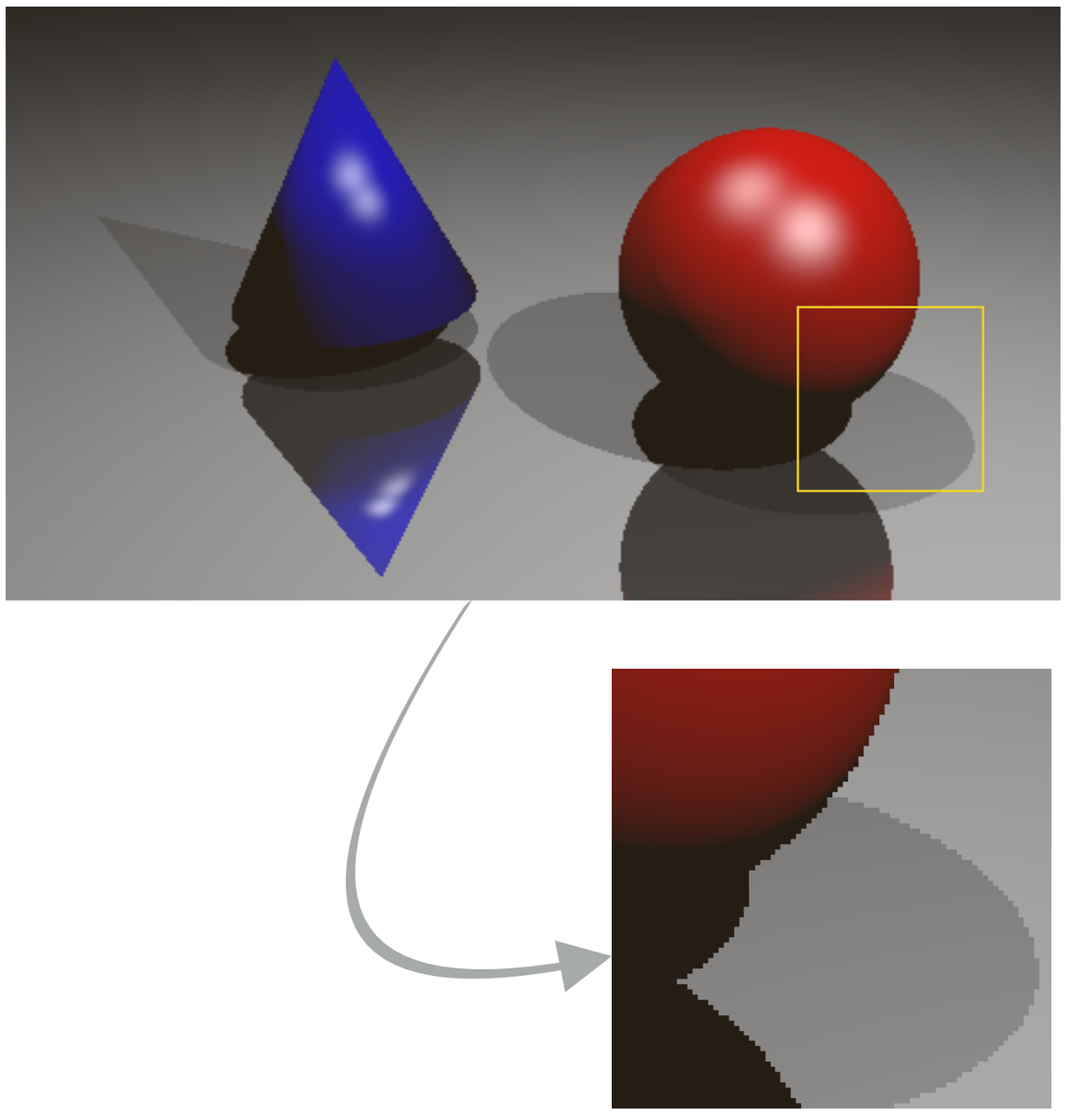
Jittering

- Choose a hybrid strategy that randomly perturbs a regular grid

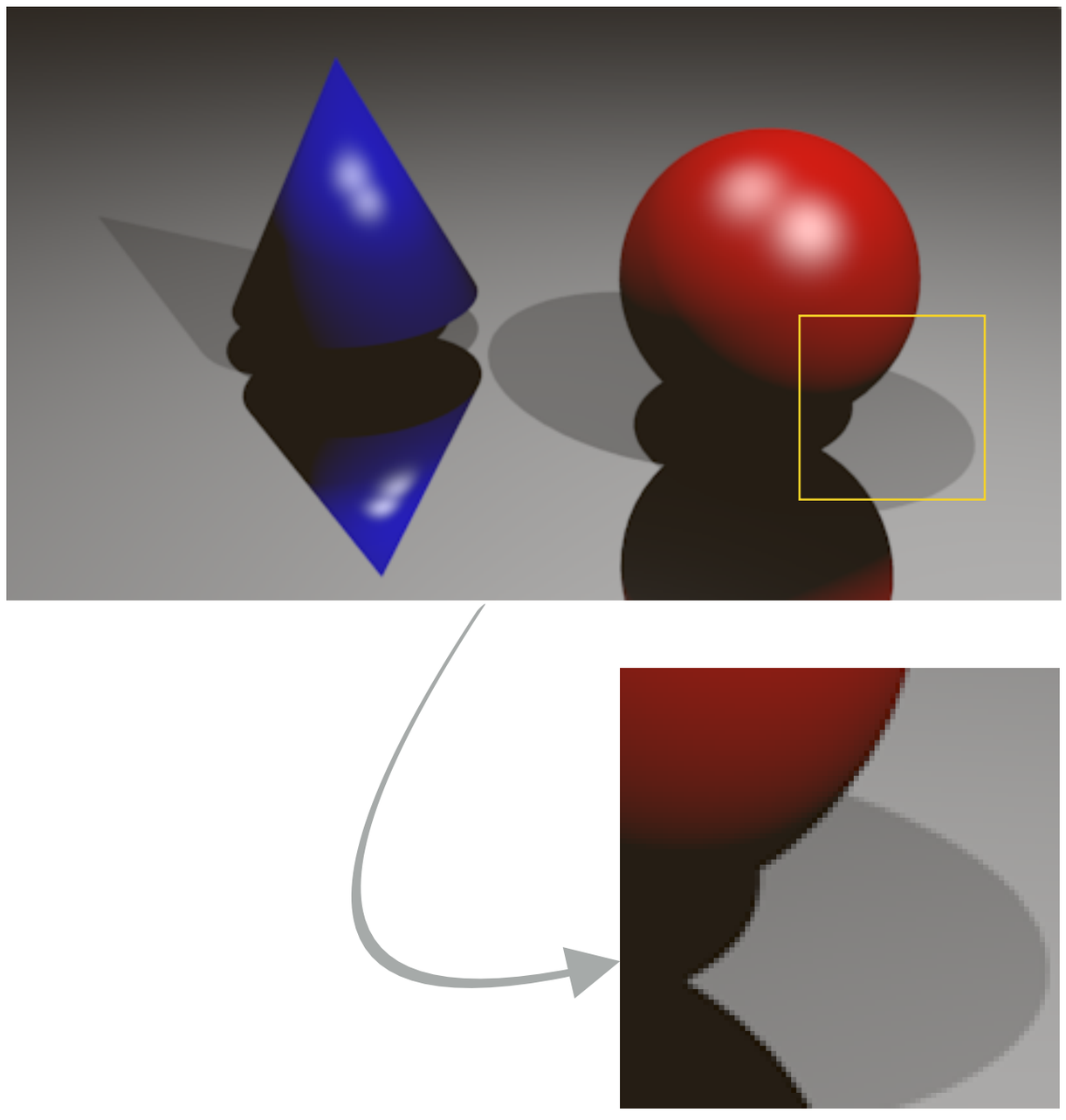
```
for each pixel (i, j) do
c = 0
for p=0 to n-1 do
    // Choose random number ξ ∈ [0, 1)
    for q=0 to n-1 do
        c = c + ray-color(i + (p + ξ)/n,
                j + (q + ξ)/n)
c_ij = c/n^2
```

Antialiasing

Before



After



Reading

- FCG: 4, 13.1, 13.4.1

ICG: Interactive Computer Graphics, E. Angel, and D. Shreiner, 6th ed.
FCG: Fundamentals of Computer Graphics, P. Shirley, M. Ashikhmin, and S. Marschner, 3rd ed.
CG: Computer Graphics, principles and Practice, J. F. Hughes, et al.