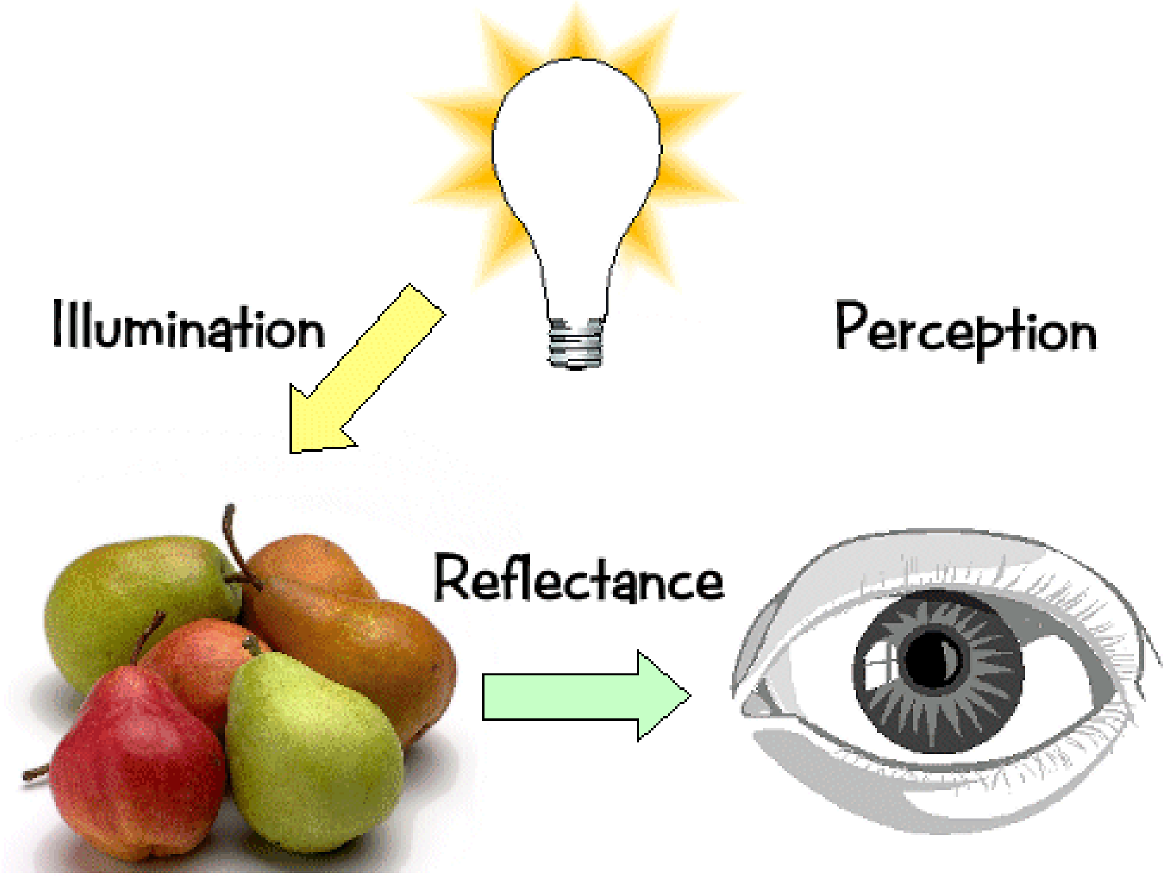


Illumination

Perception

Reflectance

# Ray Casting

- A very flexible visibility algorithm

```
loop y
  loop x
      shoot ray from eye point through pixel (x,y) into scene
      intersect with all surfaces, find first one the ray hits
      shade that surface point to compute pixel (x,y)'s color
```

# A Simple Ray Caster Program

```
Raycast()                   // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)                  // fire a ray, return RGB radiance
                                        // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface normal, material
       properties, etc.)

Shade(point, ray)           // return radiance of light leaving
                                        // point in opposite of ray direction
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

# Ray Casting

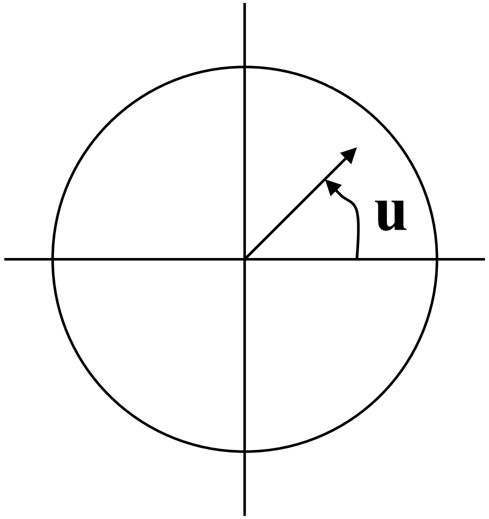
- This can be easily generalized to give recursive *ray tracing*, that will be discussed later
- calc_intersection (ray, surface) is the most important operation
  - compute not only coordinates, but also geometric or appearance attributes at the intersection point

# Ray-Surface Intersections

- How to represent a ray?
  - –A ray is p+*t*d:  p is ray origin, d the direction
  - –*t*=0 at origin of ray, *t*>0 in positive direction of ray
  - –typically assume ||d||=1
  - –p and d are typically computed in world space

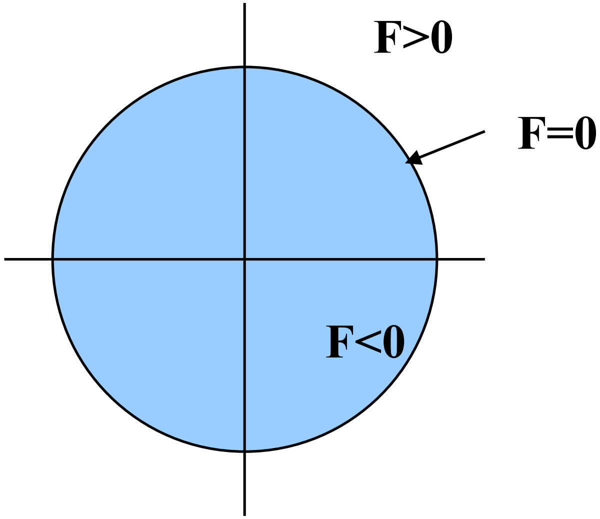
# Ray-Surface Intersections

- Surfaces can be represented by:
  - Implicit functions: $f(\mathrm{x}) = 0$
  - Parametric functions: $\mathrm{x} = g(u,v)$



**Parametric**

$$x(u) = r \cos(u)$$
$$y(u) = r \sin(u)$$



**Implicit**

$$F(x,y) = x^2 + y^2 - r^2$$

# Ray-Surface Intersections

- Compute Intersections:
  - Substitute ray equation for x
  - Find roots
  - Implicit: $f(\mathrm{p} + t\mathrm{d}) = 0$
    - » one equation in one unknown – univariate root finding
  - Parametric: $\mathrm{p} + t\mathrm{d} - g(u,v) = 0$
    - » three equations in three unknowns $(t,u,v)$ – multivariate root finding
  - For univariate polynomials, use closed form solution otherwise use numerical root finder

# The Devil’s in the Details

- General case:  non-linear root finding problem
- Ray casting is simplified using object-oriented techniques
  - Implement one intersection method for each type of surface primitive
  - Each surface handles its own intersection
- Some surfaces yield closed form solutions
  - quadrics:  spheres, cylinders, cones, elipsoids, etc…)
  - Polygons
  - tori, superquadrics, low-order spline surface patches

# Ray-Sphere Intersection

- Ray-sphere intersection is an easy case
- A sphere’s implicit function is: $x^2+y^2+z^2-r^2=0$ if sphere at origin
- The ray equation is:
  $x = p_x+td_x$
  $y = p_y+td_y$
  $z = p_z+td_z$
- Substitution gives: $(p_x+td_x)^2 + (p_y+td_y)^2 + (p_z+td_z)^2 - r^2 = 0$
- A quadratic equation in $t$.
- Solve the standard way: $A = d_x^2+d_y^2+d_z^2 = 1$ (unit vector)
  $B = 2(p_xd_x+p_yd_y+p_zd_z)$
  $C = p_x^2+p_y^2+p_z^2 - r^2$

$$At^2+Bt+C=0$$

- Quadratic formula has two roots: $t=(-B\pm sqrt(B^2-4C))/2$
  - which correspond to the two intersection points
  - negative discriminant means ray misses sphere

# Ray-Polygon Intersection

- Assuming we have a planar polygon
  - first, find intersection point of ray with plane
  - then check if that point is inside the polygon

- Latter step is a point-in-polygon test in 3-D:
  - inputs: a point x in 3-D and the vertices of a polygon in 3D
  - output: INSIDE or OUTSIDE
  - problem can be reduced to point-in-polygon test in 2-D

- Point-in-polygon test in 2-D:
  - easiest for triangles
  - easy for convex n-gons
  - harder for concave polygons
  - most common approach: subdivide all polygons into triangles
  - for optimization tips, see article by Haines in the book *Graphics Gems IV*

# Ray-Plane Intersection

- Ray: $x=p+td$
  - where p is ray origin, d is ray direction. we'll assume $||d||=1$ (this simplifies the algebra later)
  - $x=(x,y,z)$ is point on ray if $t>0$
- Plane: $(x-q)\bullet n=0$
  - where q is reference point on plane, n is plane normal. (some might assume $||n||=1$; we won't)
  - x is point on plane
  - if what you're given is vertices of a polygon
    - » compute n with cross product of two (non-parallel) edges
    - » use one of the vertices for q
  - rewrite plane equation as $x\bullet n+D=0$
    - » equivalent to the familiar formula $Ax+By+Cz+D=0$, where $(A,B,C)=n$, $D=-q\bullet n$
    - » fewer values to store

# Ray-Plane Intersection

- Steps:
  - substitute ray formula into plane eqn, yielding 1 equation in 1 unknown ($t$).
  - solution: $t = -(p \bullet n + D)/(d \bullet n)$
    - » note: if $d \bullet n = 0$ then ray and plane are parallel - REJECT
    - » note: if $t < 0$ then intersection with plane is behind ray origin - REJECT
  - compute $t$, plug it into ray equation to compute point x on plane

# Projecting A Polygon from 3-D to 2-D

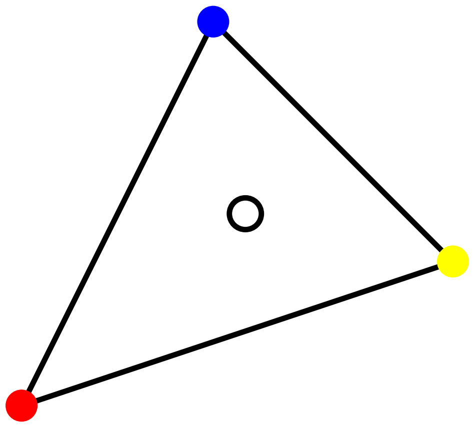
- Point-in-polygon testing is simpler and faster if we do it in 2-D
  - The simplest projections to compute are to the *xy*, *yz*, or *zx* planes
  - If the polygon has plane equation $Ax+By+Cz+D=0$, then
    - » $|A|$ is proportional to projection of polygon in *yz* plane
    - » $|B|$ is proportional to projection of polygon in *zx* plane
    - » $|C|$ is proportional to projection of polygon in *xy* plane
    - » Example: the plane $z=3$ has $(A,B,C,D)=(0,0,1,-3)$, so $|C|$ is the largest and *xy* projection is best. We should do point-in-polygon testing using *x* and *y* coords.
  - In other words, project into the plane for which the perpendicular component of the normal vector n is largest

# Projecting A Polygon from 3-D to 2-D

- Optimization:
  - We should optimize the inner loop (ray-triangle intersection testing) as much as possible
  - We can determine which plane to project to, for each triangle, as a preprocess
- Point-in-polygon testing in 2-D is still an expensive operation
- Point-in-rectangle is a special case

# Interpolated Shading for Ray Casting

- **Suppose we know colors or normals at vertices**
  - **How do we compute the color/normal of a specified point inside?**



- **Color depends on distance to each vertex**
  - **How to do linear interpolation between 3 points?**
  - **Answer: *barycentric coordinates***
- **Useful for ray-triangle intersection testing too!**

# Barycentric Coordinates in 1-D

- Linear interpolation between colors $C_0$ and $C_1$ by *t*

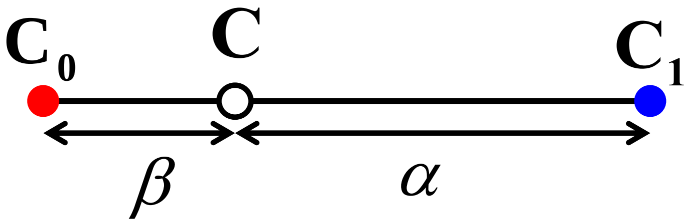
$$\mathbf{C} = (1-t)\mathbf{C}_0 + t\mathbf{C}_1$$

- We can rewrite this as

$$\mathbf{C} = \alpha\mathbf{C}_0 + \beta\mathbf{C}_1 \qquad \textit{where } \alpha + \beta = 1$$

$$\mathbf{C} \text{ is between } \mathbf{C}_0 \text{ and } \mathbf{C}_1 \Leftrightarrow \alpha, \beta \in [0,1]$$

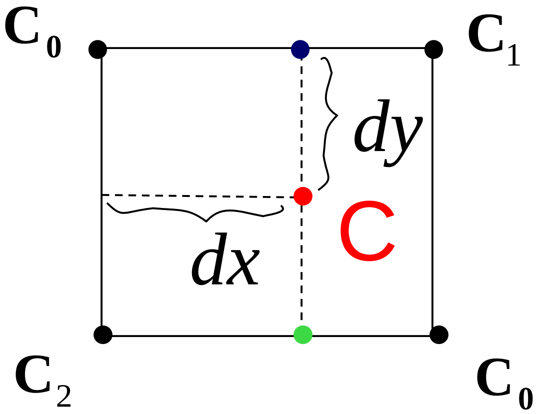
- Geometric intuition:
  - We are weighting each vertex by ratio of distances (or areas)



- $\alpha$ and $\beta$ are called ***barycentric*** coordinates

# Barycentric Coordinates in 2-D

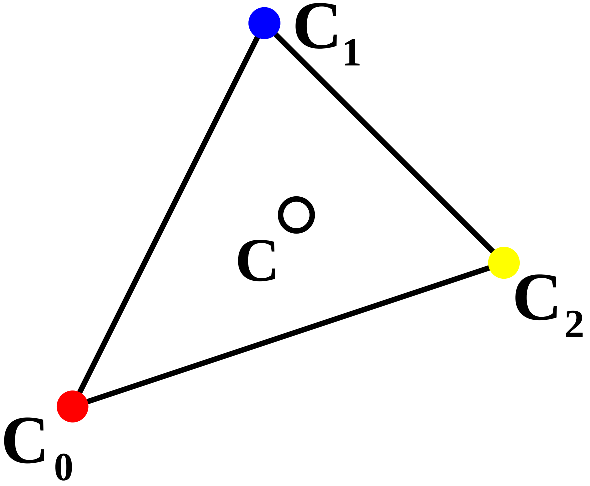
- **Bilinear interpolation: 4 points instead of 2**



$$\mathbf{C} = \underbrace{(1-dx)(1-dy)}_{\alpha}\mathbf{C}_0 + (\underbrace{dx(1-dy)}_{\beta}\mathbf{C}_1 + \underbrace{(1-dx)dy}_{\gamma}\mathbf{C}_2 + \underbrace{dxdy}_{\varphi}\mathbf{C}_3$$

# Barycentric Coordinates in 2-D

- **Now suppose we have 3 points instead of 2**



- **Define three barycentric coordinates:** $\alpha$**,** $\beta$**,** $\gamma$

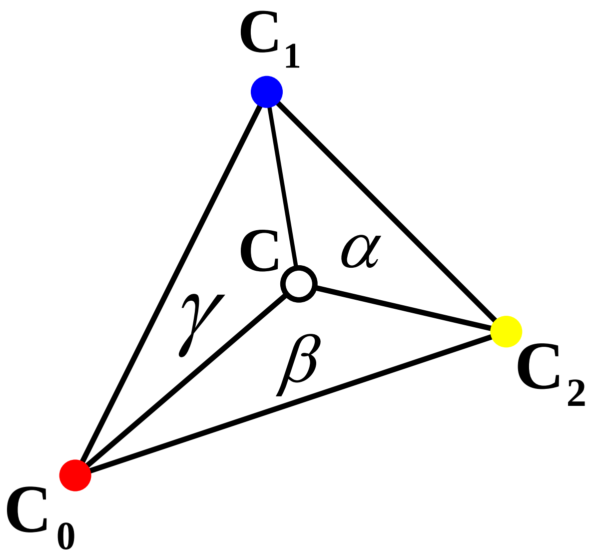
$$\mathbf{C} = \alpha \mathbf{C}_0 + \beta \mathbf{C}_1 + \gamma \mathbf{C}_2 \quad \text{where } \alpha + \beta + \gamma = 1$$

$$\mathbf{C} \text{ is inside } \mathbf{C}_0\mathbf{C}_1\mathbf{C}_2 \Leftrightarrow \alpha, \beta, \gamma \in [0,1]$$

- **How to define** $\alpha$**,** $\beta$**, and** $\gamma$**?**

# Barycentric Coordinates for a Triangle

- **Define barycentric coordinates to be ratios of triangle areas**



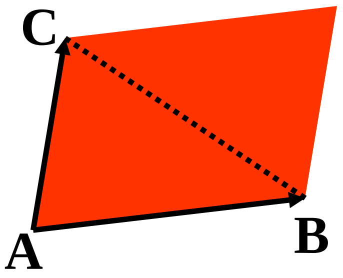
$$\alpha = \frac{Area(CC_1C_2)}{Area(C_0C_1C_2)}$$

$$\beta = \frac{Area(C_0CC_2)}{Area(C_0C_1C_2)}$$

$$\gamma = \frac{Area(C_0C_1C)}{Area(C_0C_1C_2)} = 1 - \alpha - \beta$$

# Computing Area of a Triangle

- **in 3-D**



  - *Area*(**ABC**) **= parallelogram area / 2 = ||(B-A) x (C-A)||/**2
  - **faster: project to** *xy*, *yz*, **or** *zx*, **use 2D formula**

- **in 2-D**
  - *Area*(*xy-projection*(**ABC**)) **=** $[(b_x-a_x)(c_y-a_y) - (c_x-a_x)(b_y-a_y)]/2$

    **project A,B,C to** *xy* **plane, take** *z* **component of cross product**
  - **positive if ABC is CCW (counterclockwise)**

# Computing Area of a Triangle - Algebra

**That short formula,**

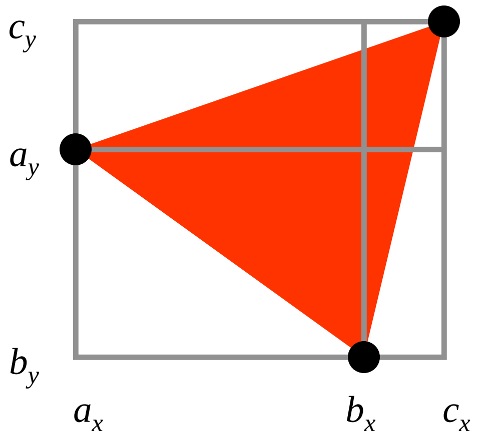
$$Area(\mathbf{ABC}) = [(b_x-a_x)(c_y-a_y) - (c_x-a_x)(b_y-a_y)]/2$$

**Where did it come from?**

$$Area(ABC) = \frac{1}{2}\begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix}$$

$$= \left( \begin{vmatrix} b_x & c_x \\ b_y & c_y \end{vmatrix} - \begin{vmatrix} a_x & c_x \\ a_y & c_y \end{vmatrix} + \begin{vmatrix} a_x & b_x \\ a_y & b_y \end{vmatrix} \right) / 2$$

$$= (b_x c_y - c_x b_y + c_x a_y - a_x c_y + c_x a_y - a_x c_y)/2$$



**The short & long formulas above agree.**

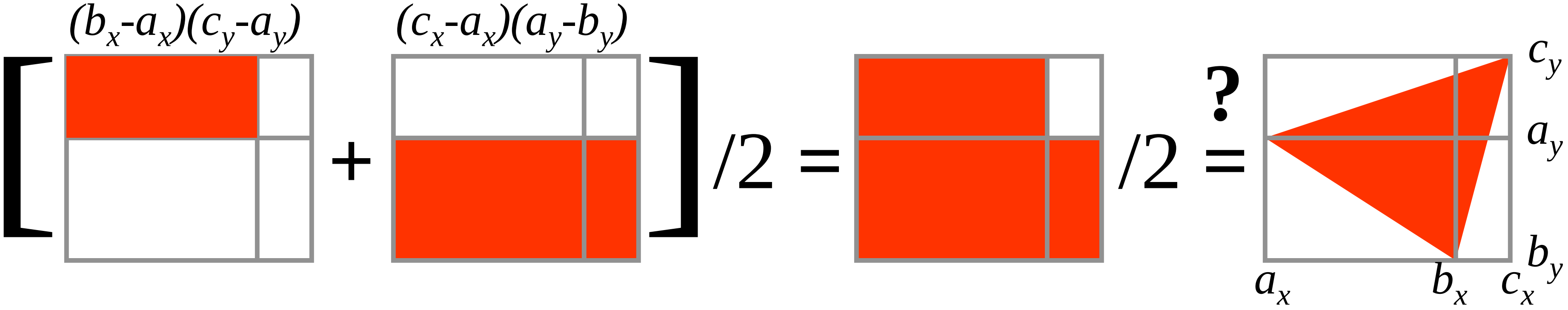
**Short formula better because fewer multiplies. Speed is important!**

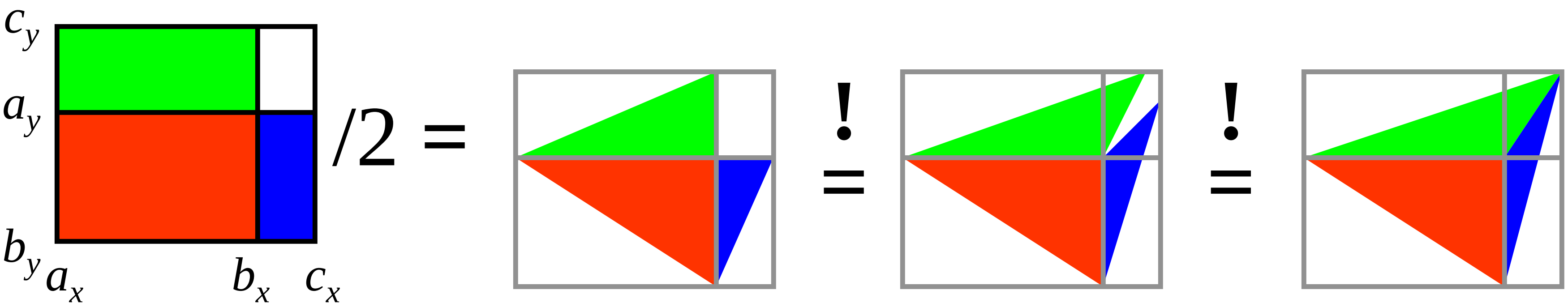
**Can we explain the formulas geometrically?**

# Another Explanation

$$Area(\mathbf{ABC}) = [(b_x-a_x)(c_y-a_y) - (c_x-a_x)(b_y-a_y)]/2$$

**is a sum of rectangle areas, divided by 2.**



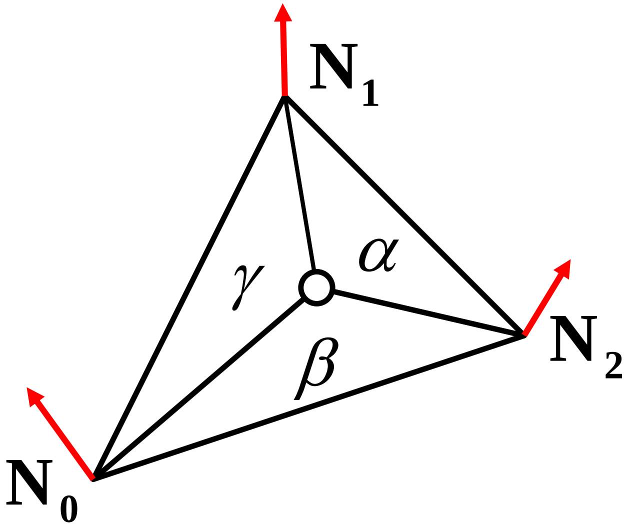


**since triangle area = base*height/2**

***it works!***

# Uses for Barycentric Coordinates

- Point-in-triangle testing!
  - point is in triangle **iff** $\alpha$, $\beta$, $\gamma$ the same sign
  - note similarity to standard point-in-polygon methods that use tests of form $a_i x+b_i y+c_i<0$ for each edge i



- Can use barycentric coordinates to interpolate any quantity
  - color interpolation – Gouraud shading
  - normal interpolation – realizing Phong Shading
  - (s,t) texture coordinate interpolation – texture mapping

# Ray Tracing

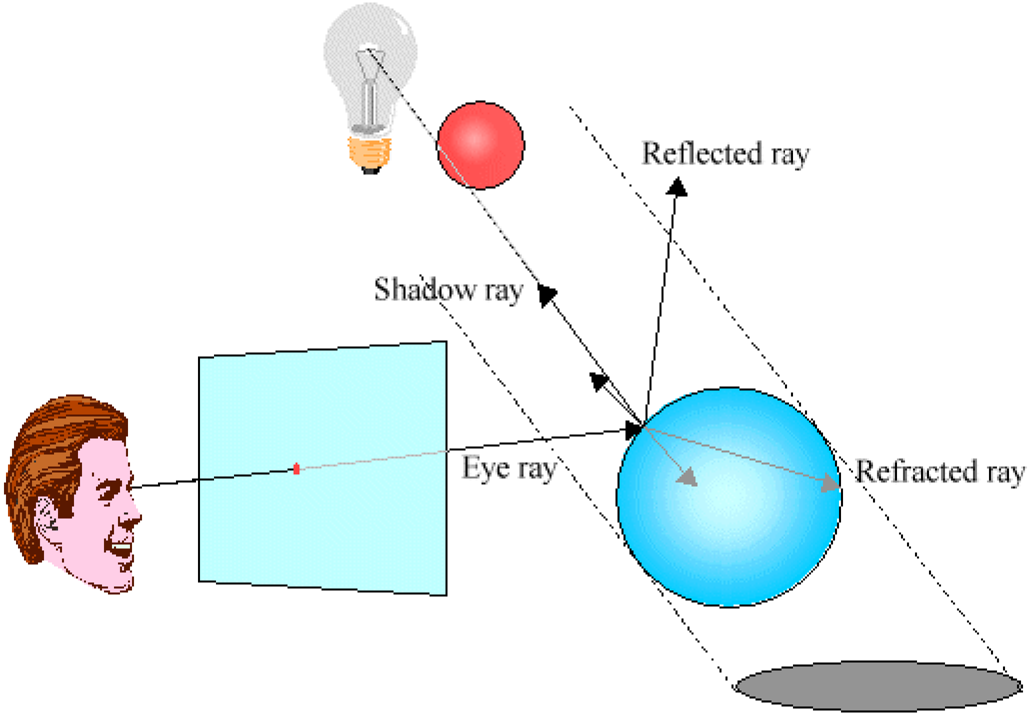
1. **(Recursive) Ray Tracing**
2. **Antialiasing**
3. **Motion Blur**
4. **Distribution Ray Tracing**
5. **other fancy stuff**

# Assumptions

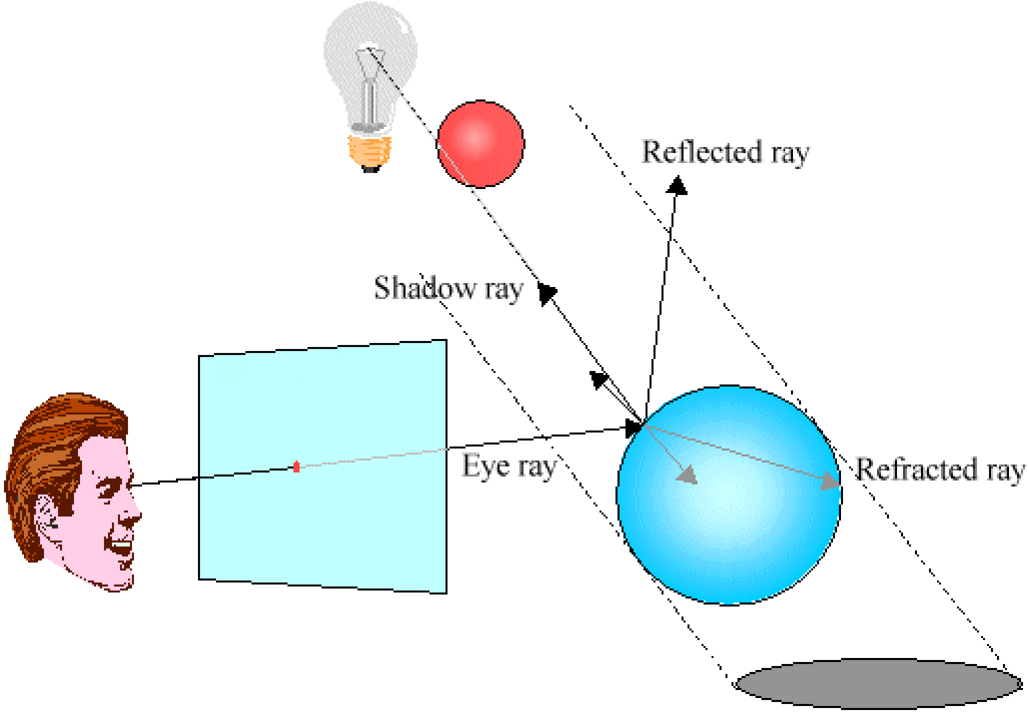
- Simple shading (typified by OpenGL, z-buffering, and Phong illumination model) assumes:
  - direct illumination (light leaves source, bounces at most once, enters eye)
  - no shadows
  - opaque surfaces
  - point light sources
  - sometimes fog
- (Recursive) ray tracing relaxes that, simulating:
  - specular reflection
  - shadows
  - transparent surfaces (transmission with refraction)
  - sometimes indirect illumination (a.k.a. global illumination)
  - sometimes area light sources
  - sometimes fog

# Ray Types for Ray Tracing

- We'll distinguish four ray types:
  - Eye rays: originating at the eye
  - Shadow rays: from surface point toward light source
  - Reflection rays: from surface point in mirror direction
  - Transmission rays: from surface point in refracted direction

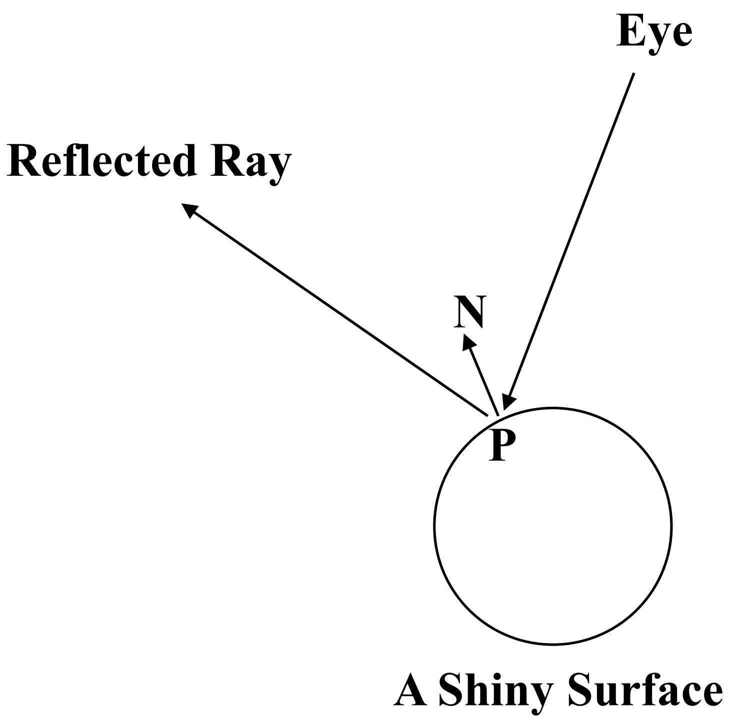


# Ray Tracing Algorithm



- – send ray from eye through each pixel
- – compute point of closest intersection with a scene surface
- – shade that point by computing shadow rays
- – spawn reflected and refracted rays, repeat

# Specular Reflection Rays

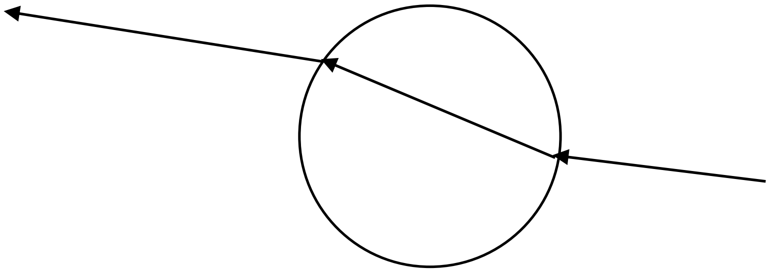


**Note: arrowheads show the direction in which we're *tracing the rays,* not the direction the light travels.**

- An eye ray hits a shiny surface
  - We know the direction from which a specular reflection would come, based on the surface normal
  - Fire a ray in this reflected direction
  - The reflected ray is treated just like an eye ray: it hits surfaces and spawns new rays
  - Light flows in the direction opposite to the rays (towards the eye), is used to calculate shading
  - It's easy to calculate the reflected ray direction

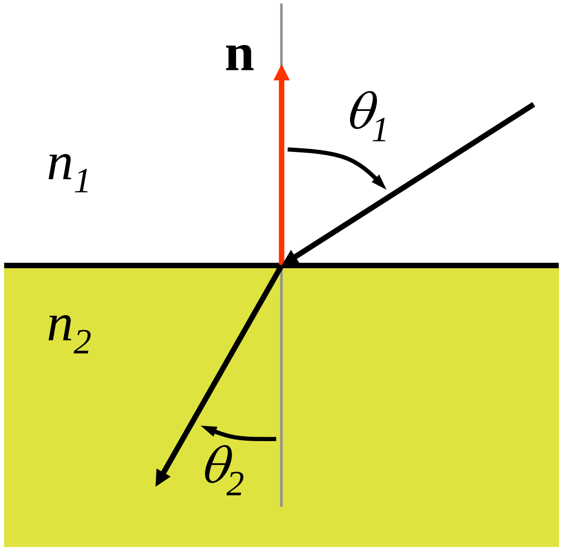
# Specular Transmission Rays

- To add transparency:
  - Add a term for light that’s coming from within the object
  - These rays are refracted (bent) when passing through a boundary between two media with different refractive indices
  - When a ray hits a transparent surface fire a *transmission ray* into the object at the proper refracted angle
  - If the ray passes through the other side of the object then it bends again (the other way)



# Refraction

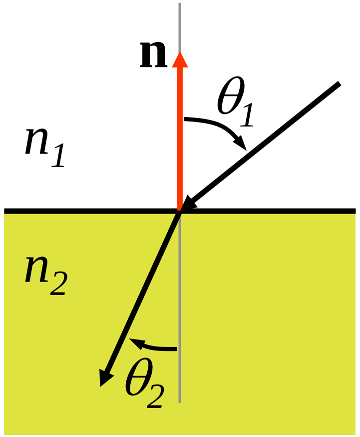
- Refraction:
  - The bending of light due to its different velocities through different materials
  - rays bend toward the normal when going from sparser to denser materials (e.g. air to water), away from normal in opposite case



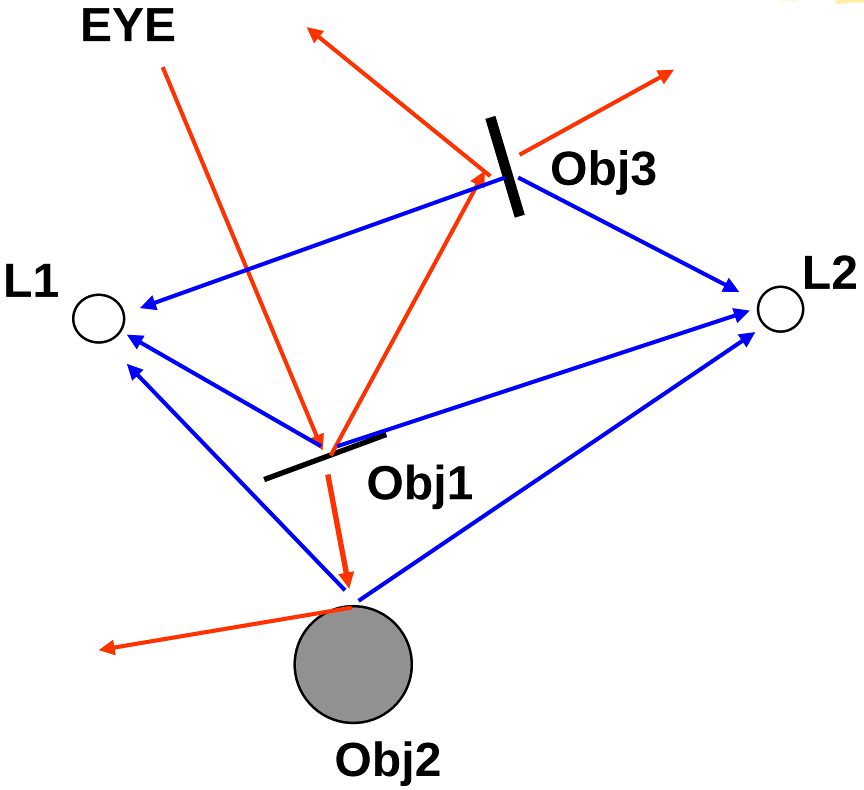
# Refraction

- Refractive index:
  - Light travels at speed *c/n* in a material of refractive index *n*
    - » *c* is the speed of light in a vacuum
    - » *c* varies with wavelength, hence rainbows and prisms
  - Use Snell's law $n_1 \sin \theta_1 = n_2 \sin \theta_2$ to derive refracted ray direction
    - » note: ray dir. can be computed without trig functions (only sqrts)

| MATERIAL | INDEX OF REFRACTION |
|---|---|
| air/vacuum | 1 |
| water | 1.33 |
| glass | about 1.5 |
| diamond 2.4 | |



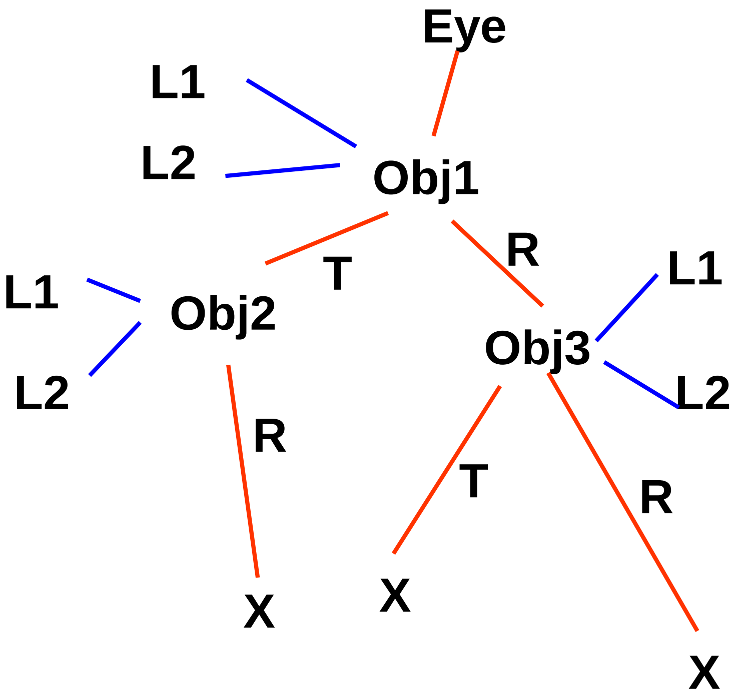
# Ray Hierarchy



RAY PATHS (BACKWARD)

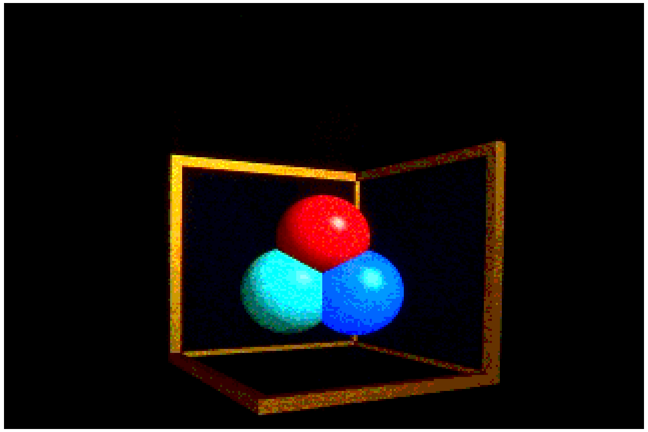
→ Shadow Ray

→ Other Ray

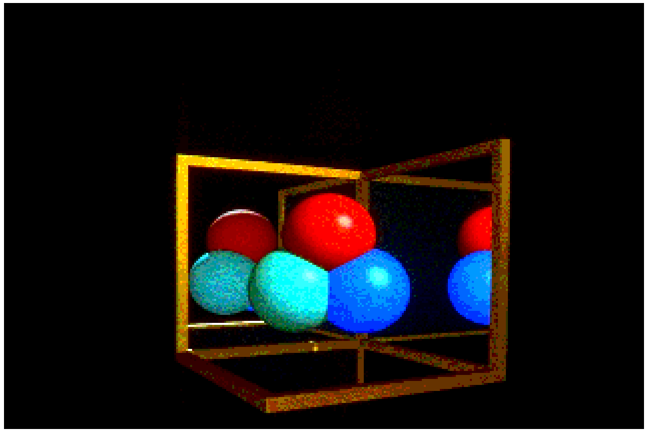


RAY TREE

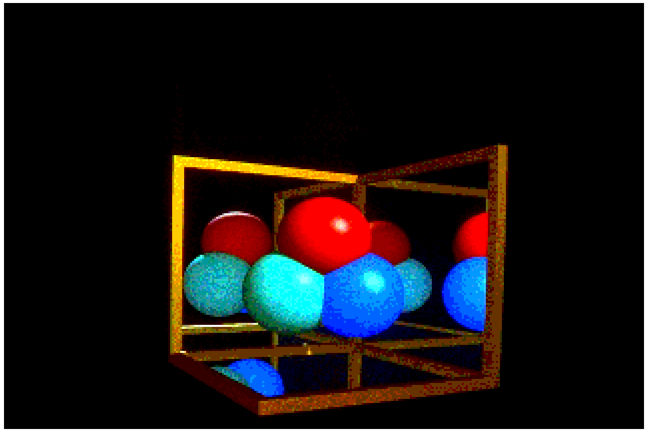
# Ray Casting vs. Ray Tracing



Ray Casting -- 1 bounce



Ray Tracing -- 2 bounce



Ray Tracing -- 3 bounce

# Review: A Simple Ray Caster Program

```
Raycast()                    // generate a picture
    for each pixel x,y
        color(pixel) = Trace(ray_through_pixel(x,y))

Trace(ray)                   // fire a ray, return RGB radiance
                                        // of light traveling backward along it
    object_point = Closest_intersection(ray)
    if object_point return Shade(object_point, ray)
    else return Background_Color

Closest_intersection(ray)
    for each surface in scene
        calc_intersection(ray, surface)
    return the closest point of intersection to viewer
    (also return other info about that point, e.g., surface normal, material
       properties, etc.)

Shade(point, ray)            // return radiance of light leaving
                                        // point in opposite of ray direction
    calculate surface normal vector
    use Phong illumination formula (or something similar)
    to calculate contributions of each light source
```

# From a Ray Caster to a Ray Tracer

```
Shade(point, ray)                    /* return radiance along ray */
    radiance = black;                        /* initialize color vector */
    for each light source
        shadow_ray = calc_shadow_ray(point,light)
        if !in_shadow(shadow_ray,light)
            radiance += phong_illumination(point,ray,light)
    if material is specularly reflective
        radiance += spec_reflectance * Trace(reflected_ray(point,ray)))
    if material is specularly transmissive
        radiance += spec_transmittance * Trace(refracted_ray(point,ray)))
    return radiance
```

# Problem with Simple Ray Tracing: Aliasing

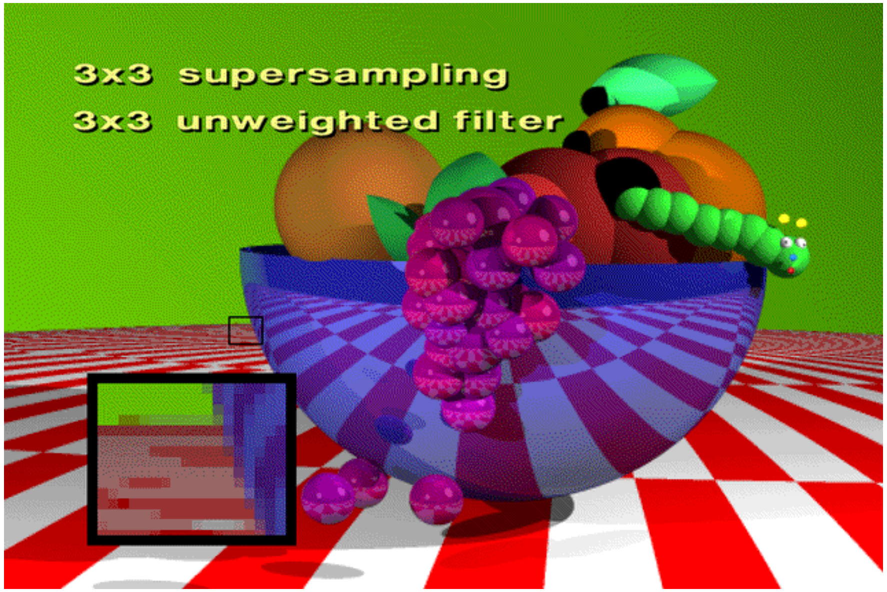
# Aliasing

- Ray tracing shoots one ray per pixel
- But a pixel represents an area; one ray samples only one point with the area; an area consists *infinite* number of points
  - These points may not all have the same color
  - This leads to *aliasing*
    - » jaggies
    - » moire patterns
- How do we fix this problem?
  - Recall antialiasing in texture mapping

# Antialiasing: Supersampling

- We talked about two antialiasing methods
  - Supersampling
  - Pre-filtering (MIP-mapping)
- Here we use supersampling
  - Fire more than one ray for each pixel (e.g., a 3x3 grid of rays)
  - Average the results using a filter (or some kind of filter)

# Supersampling



# Antialiasing: Adaptive Supersampling

- Supersampling can be done *adaptively*
  - divide pixel into 2x2 grid, trace 5 rays (4 at corners, 1 at center)
  - if the colors are similar then just use their average
  - otherwise recursively subdivide each cell of grid
  - keep going until each 2x2 grid is close to uniform or limit is reached
  - filter the result

- Behavior of adaptive supersampling
  - Areas with fairly constant appearance are sparsely sampled
  - Areas with lots of variability are heavily sampled

- Issues
  - even with massive supersampling visible aliasing is possible when the sampling grid interacts with regular structures
  - problem is, objects tend to be almost aligned with sampling grid
  - noticeable beating, moire patterns, etc… are possible

# Antialiasing: Stochastic Adaptive Supersampling

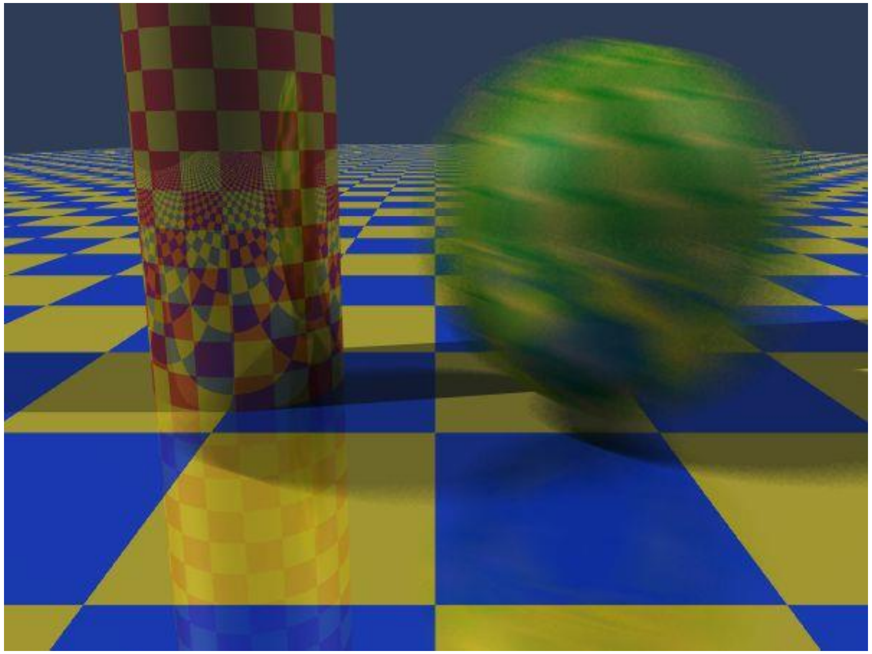
- Adaptive supersampling can be done *stochasticly*
  - instead of a regular grid, subsample randomly (or pseudo)
  - aliasing is replaced by less visually annoying noise!
  - adaptively sample *statistically*
  - keep taking samples until the color estimates converge
  - How?
    - » jittering:  perturb a regular grid
    - » Jitter pattern can be pre-generated (designed)
    - » this can be employed in OpenGL rendering as well

# Temporal Aliasing

- Aliasing happens in time as well as space
  - the sampling rate is the frame rate, 30Hz for NTSC video, 24Hz for film
  - fast moving objects move large distances between frames
  - if we point-sample time, objects have a jerky look
- To avoid temporal aliasing we need to filter in time too
  - so compute frames at 120Hz and average them together (with appropriate weights)?
  - fast-moving objects become blurred streaks
- Real media (film and video) automatically do temporal anti-aliasing
  - photographic film integrates over the exposure time
  - video cameras have persistence (memory)
  - this shows up as *motion blur* in the photographs

# Motion Blur

- Apply stochastic sampling to time as well as space
- Assign a time as well as an image position to each ray
- The result is still-frame motion blur and smooth animation
- This is an example of distribution ray tracing



# The Classic Example of Motion Blur

- From Foley et. al. Plate III.16
- Rendered using distribution ray tracing at 4096x3550 pixels, 16 samples per pixel.
- Note motion-blurred reflections and shadows with penumbrae cast by extended light sources.

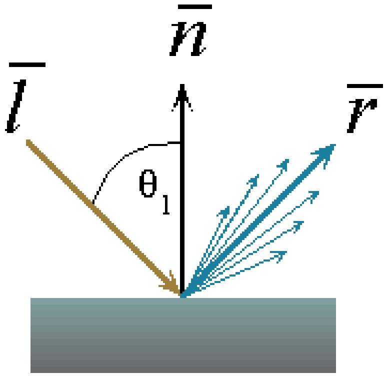
# Distribution Ray Tracing

- We've done
  - distribute rays throughout a pixel to get spatial antialiasing
  - distribute rays in time to get temporal antialiasing (motion blur)
- We can
  - distribute rays in reflected ray direction to simulate gloss
  - distribute rays across area light source to simulate penumbras (soft shadows)
  - distribute rays throughout lens area to simulate depth of field
  - distribute rays across hemisphere to simulate diffuse interreflection (radiosity)

- a.k.a. "distributed ray tracing" or stochastic ray tracing
- powerful idea! (but can get slow)

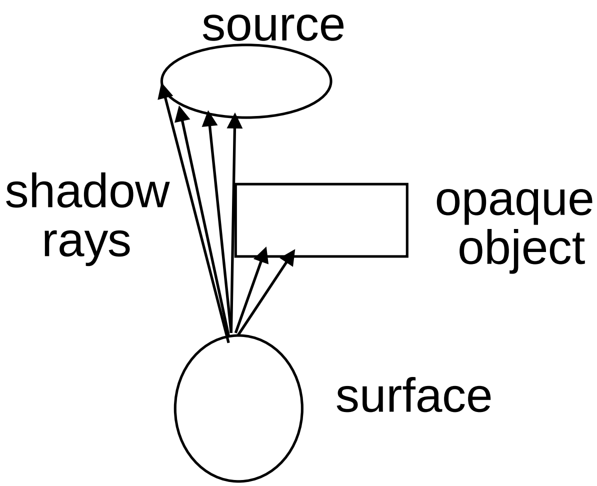
# Gloss and Highlights

- Simple ray tracing spawns only one reflected ray
- But Phong illumination models a cone of rays
  - Produces fuzzy highlights
  - Change fuzziness (cone width) by varying the shininess parameter
- The solution is to spawn a cluster of rays
- Again, *stochastic sampling* can be used
  - Stochastically sample rays within the cone
  - Sampling probability drops off sharply away from the specular angle
  - Highlights can be soft, blurred reflections of other objects

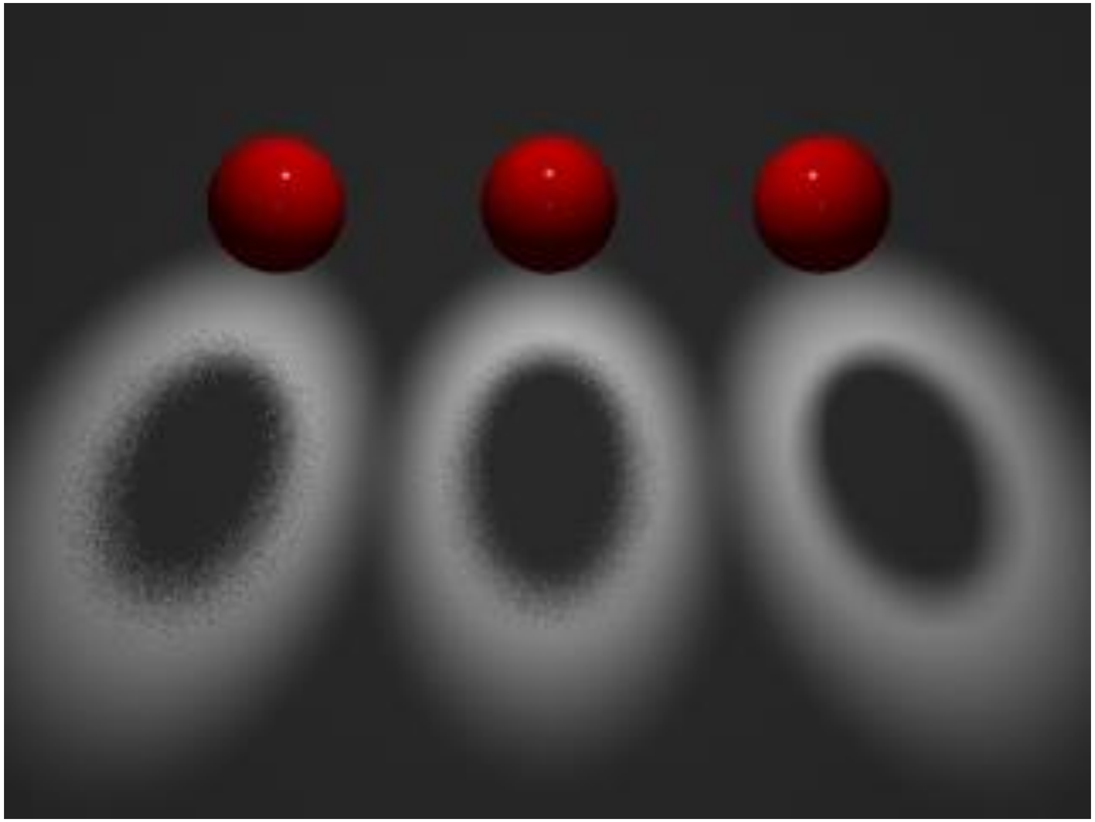


# Soft Shadows

- Point light sources produce sharp shadow edges
  - the point is either shadowed or not
  - only one ray is required
- With an extended light source the surface point may be partially visible to it (*partial eclipse*)
  - only part of the light from the sources reaches the point
  - the shadow edges are softer
  - the transition region is the *penumbra*
- Distribution ray tracing can simulate this:
  - fire shadow rays from random points on the source
  - weight them by the brightness
  - the resulting shading depends on the fraction of the obstructed shadow rays

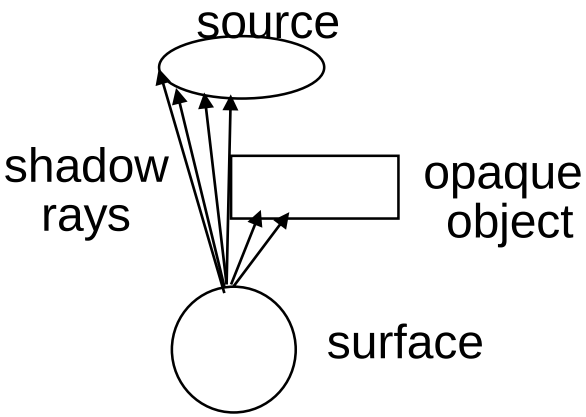


# Soft Shadows



**fewer rays, more noise**

**more rays, less noise**



# Depth of Field

- The pinhole camera model only approximates real optics
  - real cameras have lenses with focal lengths
  - only one plane is truly in focus
  - points away from the focus project as disks
  - the further away from the focus the larger the disk
- the range of distance that appear in focus is the *depth of field*
- simulate this using stochastic sampling through different parts of the lens