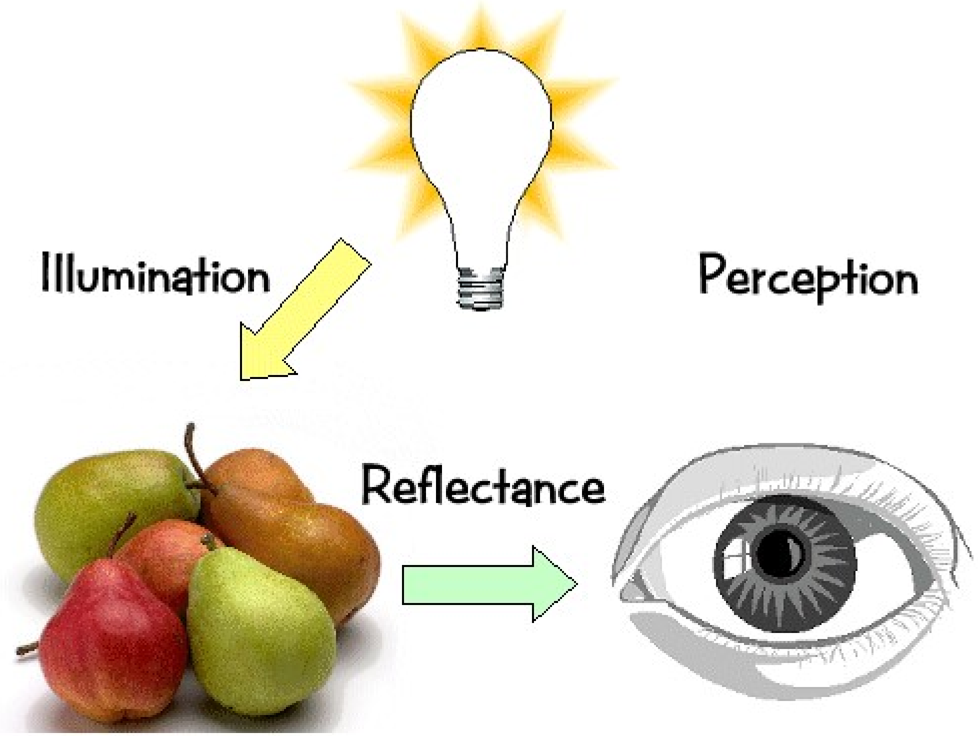
Ray Tracing

Lin Lu (吕琳)

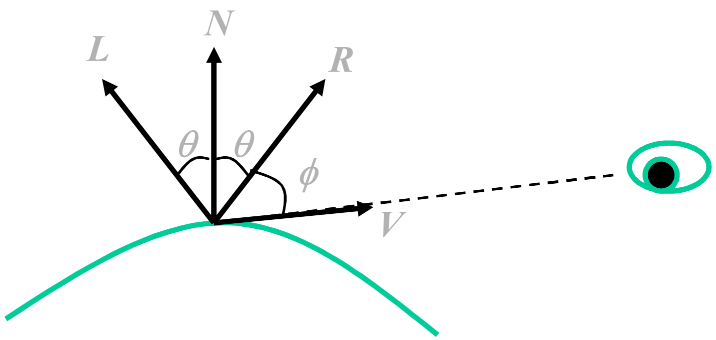
Shandong University

http://irc.cs.sdu.edu.cn/~lulin/

Recall



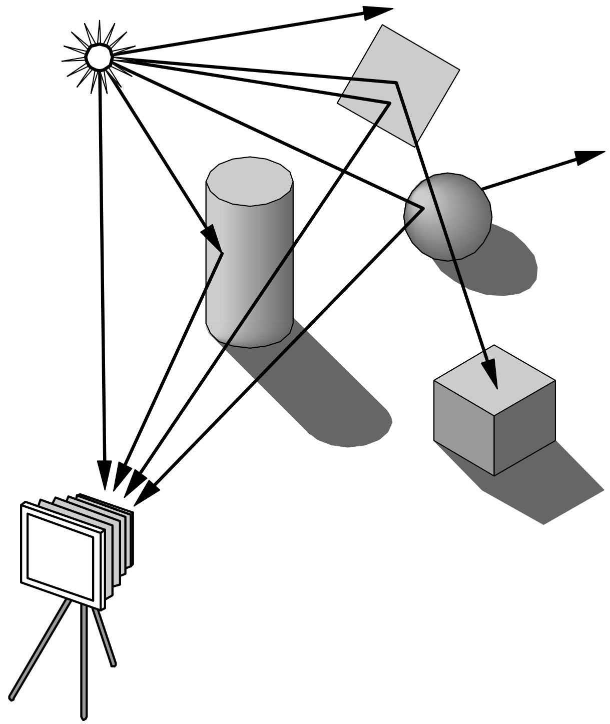
Recall



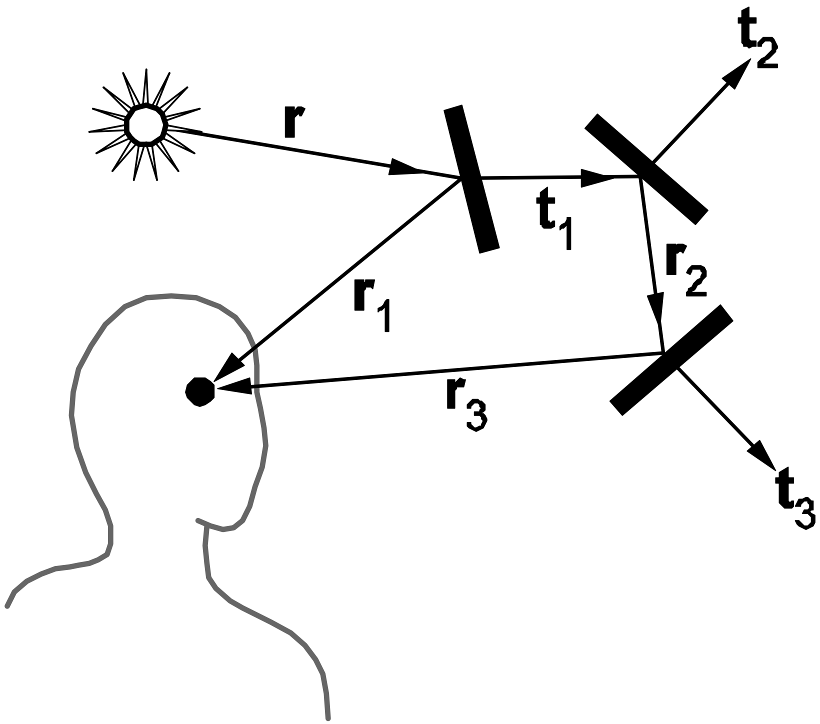
$$I = k_a I_a + f_{att} I_{light} \left[k_d \cos\theta + k_s (\cos\phi)^{n_{shiny}} \right]$$

Ray Tracing

- Follow rays of light from a point source
- Can account for reflection and transmission



Ray Trees



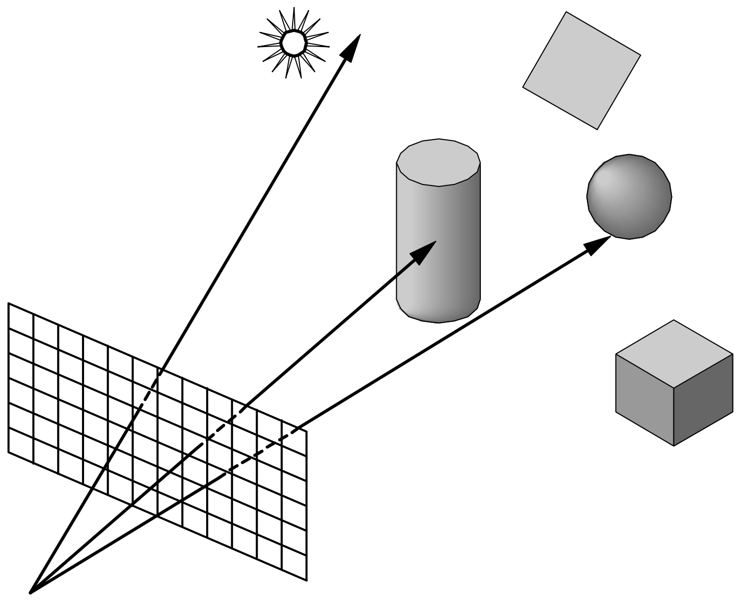
$\mathbf{t}_2$
$\mathbf{r}$
$\mathbf{t}_1$
$\mathbf{r}_2$
$\mathbf{r}_1$
$\mathbf{r}_3$
$\mathbf{t}_3$

Computation

- Should be able to handle all physical interactions
- Ray tracing paradigm is not computational
- Most rays do not affect what we see
- Scattering produces many (infinite) additional rays
- Alternative: ray casting

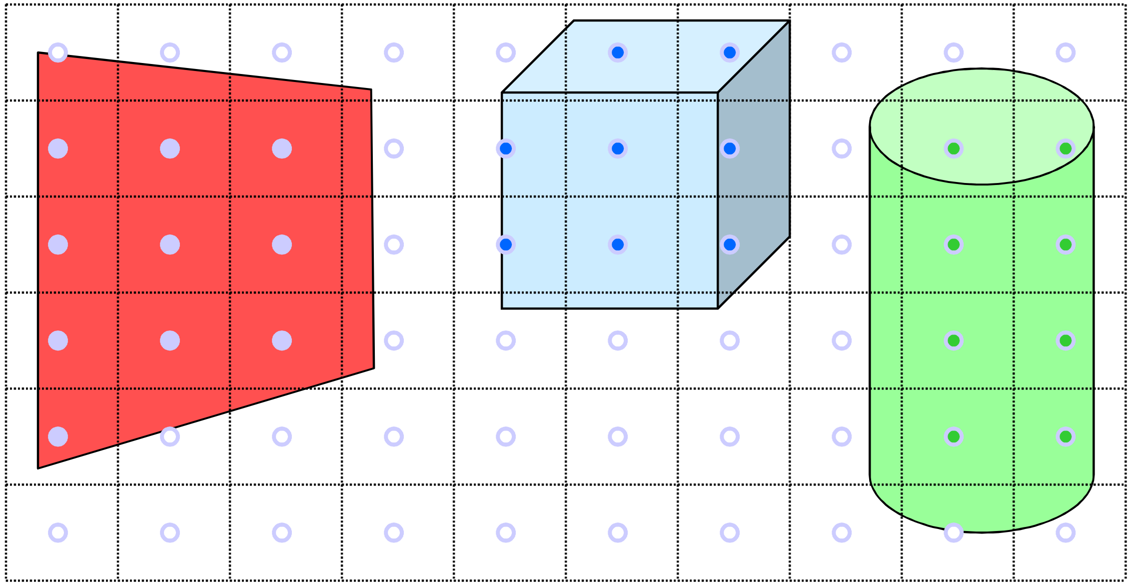
Ray Casting

- Only rays that reach the eye matter
- Reverse direction and cast rays
- Need at least one ray per pixel



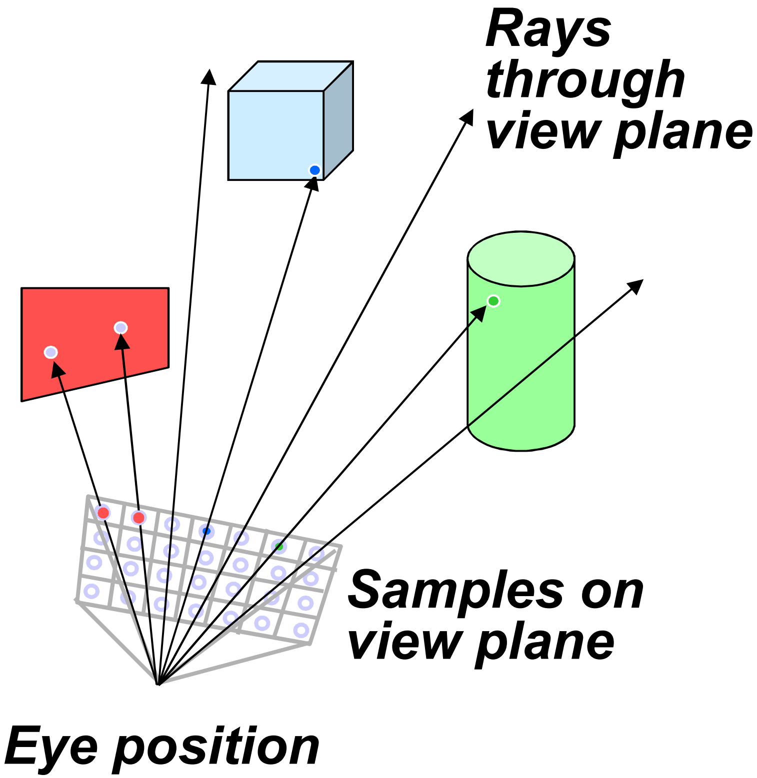
Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



Ray Casting

- For each sample …
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color sample based on surface radiance



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

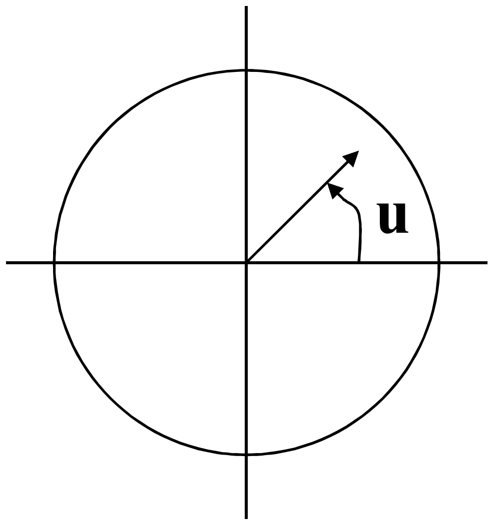
- This can be easily generalized to give recursive *ray tracing*, that will be discussed later
- calc_intersection (ray, surface) is the most important operation
  - compute not only coordinates, but also geometric or appearance attributes at the intersection point

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

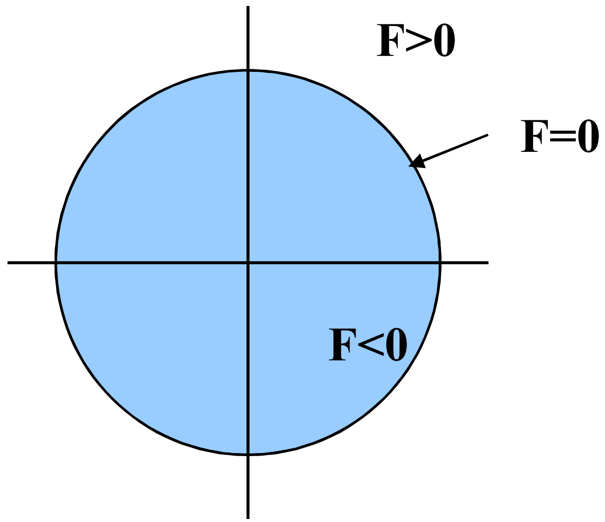
Ray-Surface Intersections

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



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, ellipsoids, 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)

  $At^2+Bt+C=0$ $B = 2(p_xd_x+p_yd_y+p_zd_z)$
  $C = p_x^2+p_y^2+p_z^2 - r^2$
- 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 3-D
  - 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+*t*d
  - 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)•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•n+*D=0*
    - equivalent to the familiar formula *Ax*+*By*+*Cz*+*D=0*, where (*A,B,C*)=n, *D*=-q•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