Computer Graphics

Ch10 Global Illumination

Instructor: Dr. MAO Aihua

ahmao@scut.edu.cn

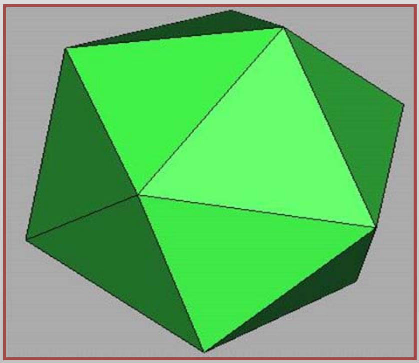
Overview

Direct (Local) Illumination

- Emission at light sources
- Scattering at surfaces

Global illumination

- Shadows
- Refractions
- Inter-object reflections

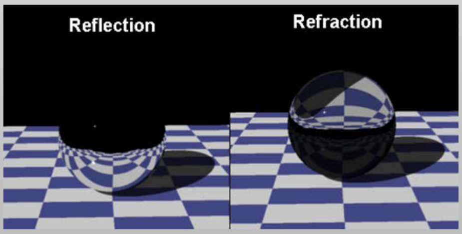


Direct Illumination

Global Illumination

Global Illumination

- The notion that a point is illuminated by more than light from local lights; it is illuminated by all the emitters and reflectors in the global scene
  - *Ray Tracing*
  - *Radiosity*



Local vs. Global Illumination

- Local illumination: Phong model (OpenGL)
  - Light to surface to viewer
  - No shadows, inter-reflections
  - Fast enough for interactive graphics
- Global illumination: Ray tracing
  - Multiple specular reflections and transmissions
  - Only one step of diffuse reflection
- Global illumination: Radiosity
  - All diffuse interreflections; shadows
  -

Image vs. Object Space

Image space: Ray tracing
– Trace backwards from viewer
– View-dependent calculation
– Result: rasterized image (pixel by pixel)

• Object space: Radiosity
– Assume only diffuse-diffuse interactions
– View-independent calculation
– Result: 3D model, color for each surface patch
– Can render with OpenGL

The ‘Rendering Equation’

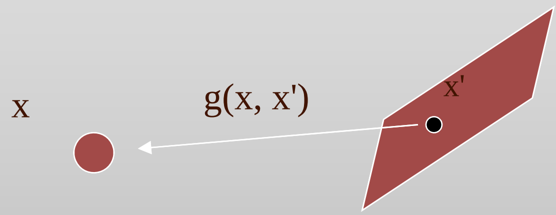
Jim Kajiya (Current head of Microsoft Research) developed this in 1986

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

I(x, x’)	= total intensity from point x’ to x
g(x, x’)	= 0 when x/x’ are occluded = $1/d^2$ otherwise (d = distance between x and x’)
ε(x, x’)	= intensity emitted by x’ to x
ρ(x, x’,x’’)	= intensity of light reflected from x’’ to x through x’
S	= all points on all surfaces

The ‘Rendering Equation’

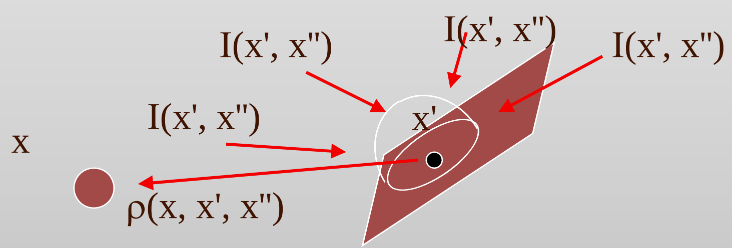
$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$



0 or $1/d^2$
depending on emissivity

The ‘Rendering Equation’

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$



intensity of light reflected from x” to
x through x’

The ‘Rendering Equation’

The light that hits x from x’ is the direct illumination from x’ and all the light reflected by x’ from all x’’

To implement:

- Must handle recursion effectively
- Must support diffuse and specular light
- Must model object shadowing

The ‘Rendering Equation’

What’s really hard about computing this?

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

The integral…

- How can one compute I(x, x') for all points x and x'?
- Approximate!!!

Approximating the Rendering Equation

Don't integrate over all points, just a subset

- Ray Tracing
  - *Considered a Monte Carlo approximation*

2 Monte Carlo == Random "sampling" of real answer

by Gilles Tran

Approximating the Rendering Equation

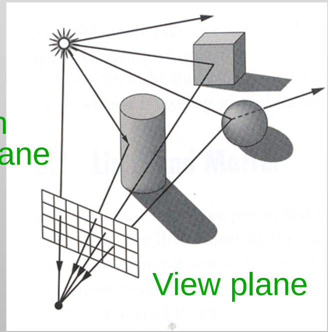
Group "all points" into sets and consider all sets

- Radiosity
  - *Considered a finite-element approximation*

Ray Casting

A simple form of Ray Tracing

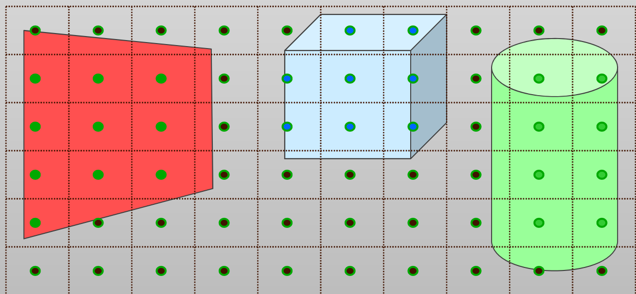
Simplest method is ray casting



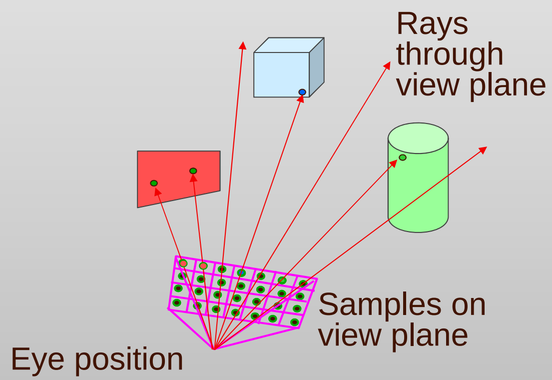
Ray Casting

To create 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



Ray Casting

Simple implementation:

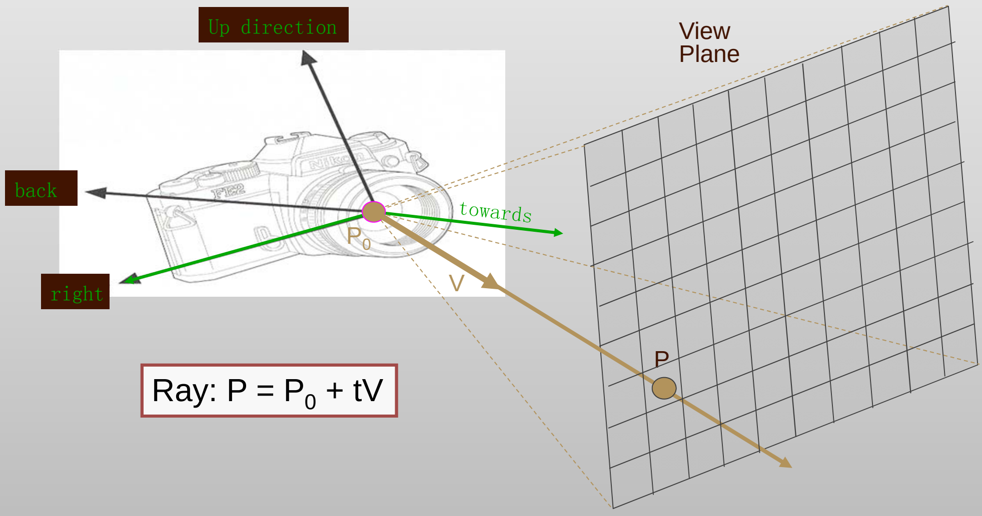
```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

Ray Casting

Simple implementation:

```
Image RayCast(Camera camera, Scene scene, int width, int height)
{
    Image image = new Image(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            Ray ray = ConstructRayThroughPixel(camera, i, j);
            Intersection hit = FindIntersection(ray, scene);
            image[i][j] = GetColor(hit);
        }
    }
    return image;
}
```

Constructing Ray Through a Pixel



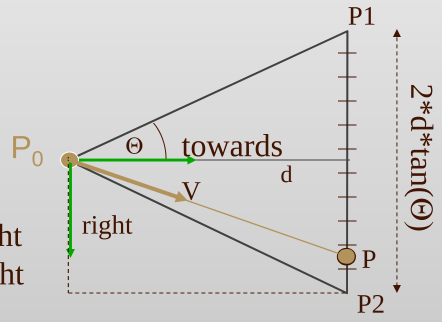
Ray: $P = P_0 + tV$

Constructing Ray Through a Pixel

2D Example

Θ = frustum half-angle
d = distance to view plane
$right = towards \times up$

$P1 = P_0 + d*towards - d*\tan(\Theta)*right$
$P2 = P_0 + d*towards + d*\tan(\Theta)*right$



Ray: $P = P_0 + tV$

$P = P1 + (i+ 0.5)/width * (P2 - P1)$
$= P1 + (i+ 0.5) /width * 2*d*\tan(\Theta)*right$
$V = (P - P_0) / ||P - P_0||$

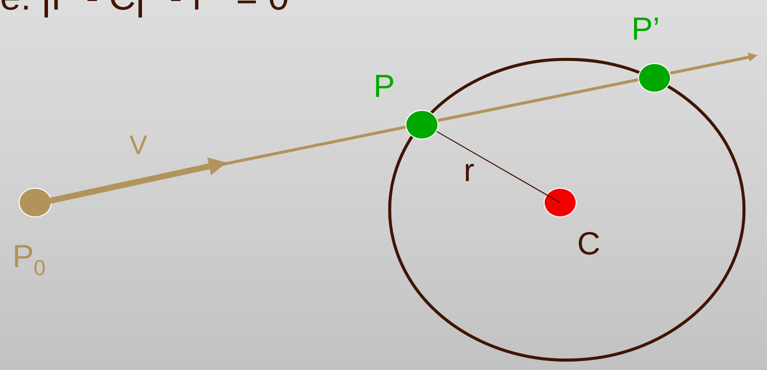
Ray-Scene Intersection

Intersections with geometric primitives

- Sphere
- Triangle
- Groups of primitives (scene)

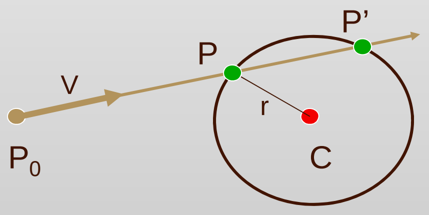
Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - C|^2 - r^2 = 0$



Ray-Sphere Intersection

Ray: $P = P_0 + tV$
Sphere: $|P - C|^2 - r^2 = 0$



Substituting for P, we get:

$$|P_0 + tV - C|^2 - r^2 = 0$$

Solve quadratic equation:

$$at^2 + bt + c = 0$$

where:

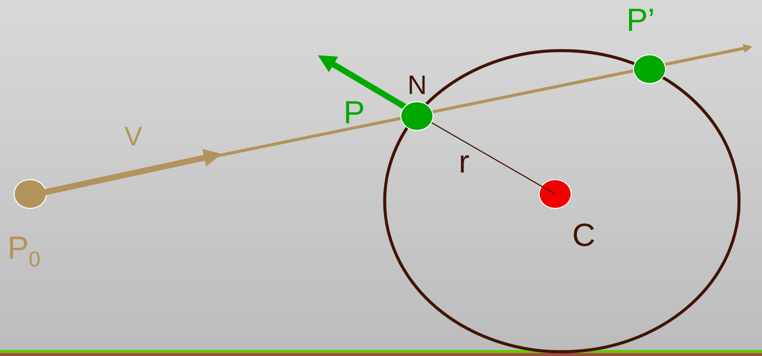
$$a = |V|^2 = 1$$
$$b = 2\,V \bullet (P_0 - C)$$
$$c = |P_0 - C|^2 - r^2$$

$\boxed{\text{If ray direction is normalized!}}$

Ray-Sphere Intersection

Need normal vector at intersection
for lighting calculations

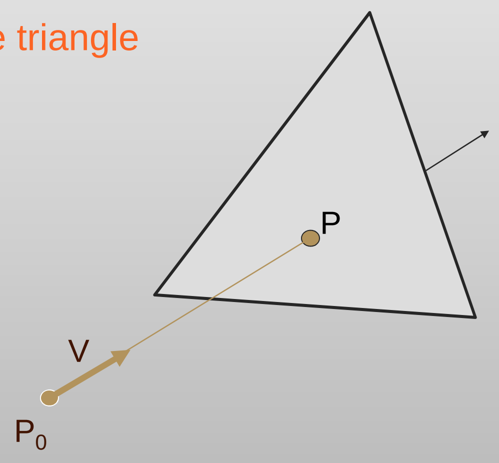
$$N = (P - C) / ||P - C||$$



Ray-Triangle Intersection

First, intersect ray with plane

Then, check if point is inside triangle

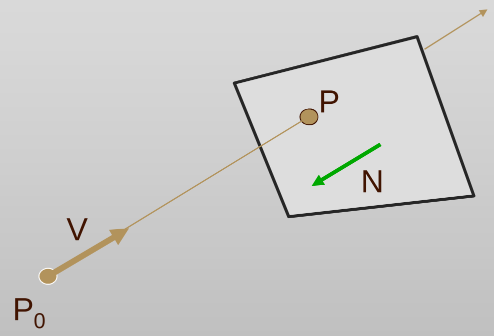


Ray-Plane Intersection

Ray: $P = P_0 + tV$
Plane: $P \bullet N + d = 0$

Substituting for P, we get:
$(P_0 + tV) \bullet N + d = 0$

Solution:
$t = -(P_0 \bullet N + d) / (V \bullet N)$

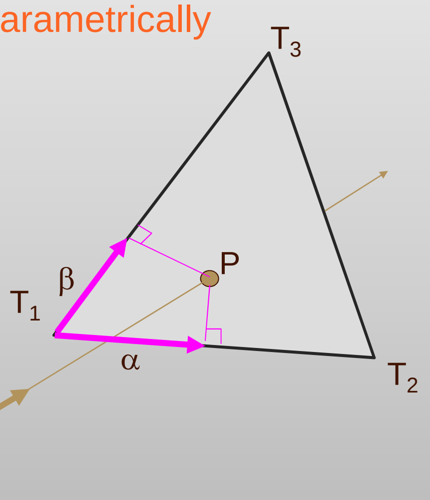


Ray-Triangle Intersectio

Check if point is inside triangle parametrically

Compute α, β:
$P = \alpha\,(T_2 - T_1) + \beta\,(T_3 - T_1)$

Check if point inside triangle.
$0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$
$\alpha + \beta \leq 1$



Other Ray-Primitive Intersections

Cone, cylinder, ellipsoid:

- Similar to sphere

Box

- Intersect 3 front-facing planes, return closest

Convex polygon

- Same as triangle (check point-in-polygon algebraically)

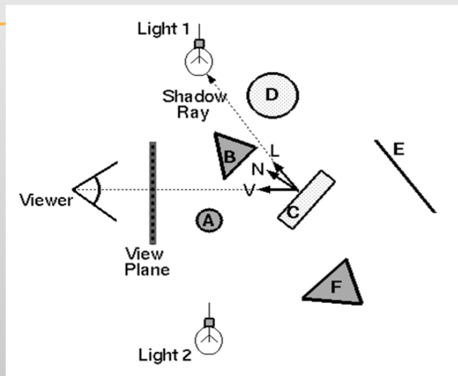
Concave polygon

- Same plane intersection
- More complex point-in-polygon test

Ray Casting – direct illumination

Trace primary rays from camera

- Direct illumination from unblocked lights only
- $S_i = 1$



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

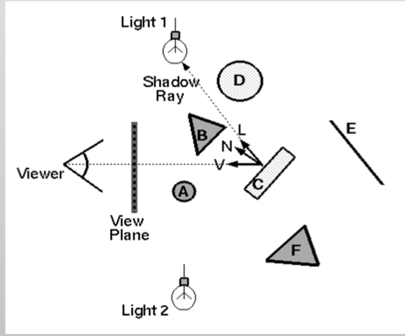
$$\downarrow$$

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L$$

Shadows

Shadow term tells if light sources are blocked

- Cast ray towards each light source L_i
- $S_i = 0$ if ray is blocked, otherwise
- $0 < S_i < 1 \rightarrow$ soft shadows (hack)

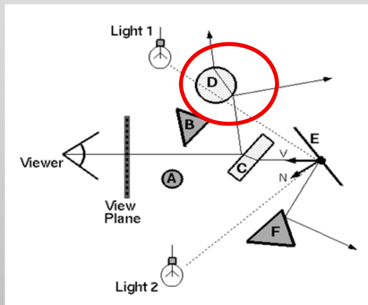


$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L$$

Recursive Ray Tracing – second-order effects

Also trace secondary rays from hit surfaces

- Global illumination from mirror reflection and transparency

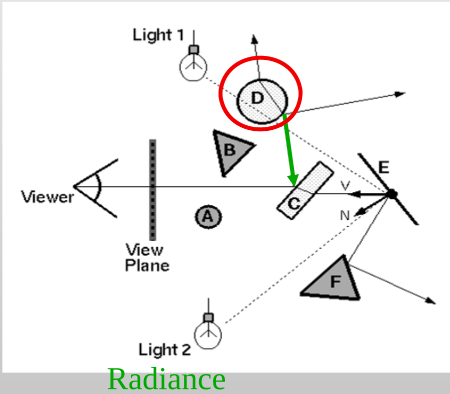


$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_R I_R + K_T I_T$$

Mirror reflections

Trace secondary ray in mirror direction

- Evaluate radiance along secondary ray and include it into illumination model



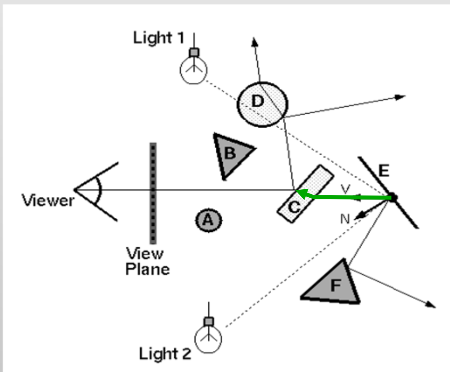
Radiance for mirror reflection ray

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_R I_R + K_T I_T$$

Transparency

Trace secondary ray in direction of refraction

- Evaluate radiance along secondary ray and include it into illumination model



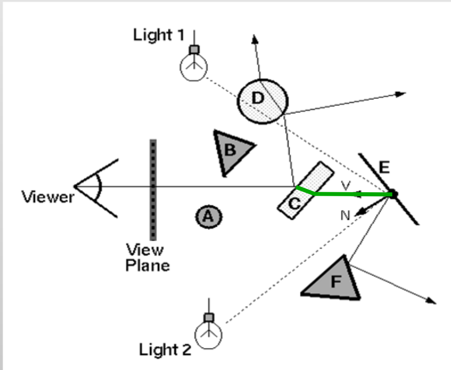
Radiance for refraction ray

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_R I_R + K_T I_T$$

Transparency

Transparency coefficient is fraction transmitted

- $K_T = 1$ for translucent object, $K_T = 0$ for opaque
- $0 < K_T < 1$ for object that is semi-translucent



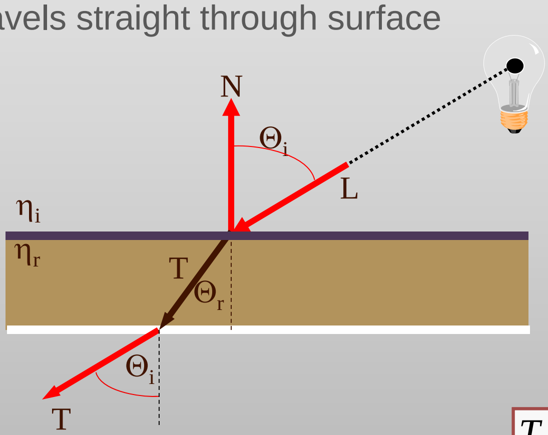
Transparency Coefficient

$$I = I_E + K_A I_A + \sum_L (K_D (N \bullet L) + K_S (V \bullet R)^n) S_L I_L + K_R I_R + K_T I_T$$

Refractive Transparency

For thin surfaces, can ignore change in direction

- Assume light travels straight through surface



$$T \cong -L$$

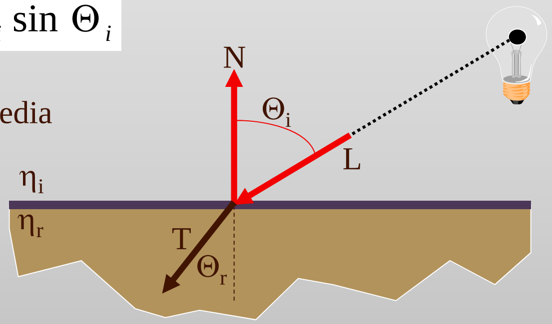
Refractive Tranparency

For solid objects, apply Snell's law:

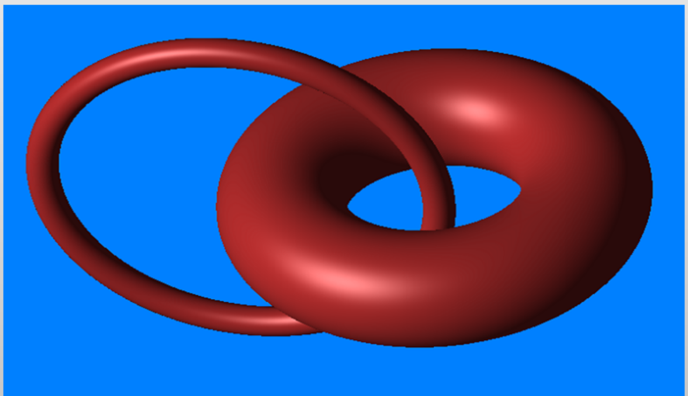
$$\eta_r \sin \Theta_r = \eta_i \sin \Theta_i$$

η_r η_i

is refractivity of the media

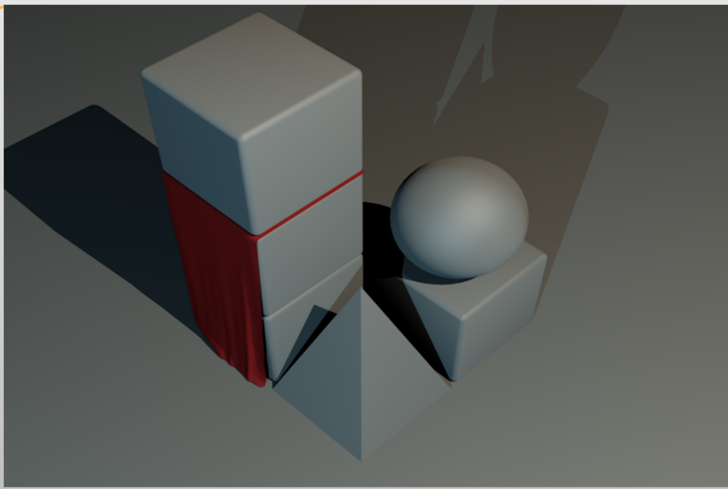


Example



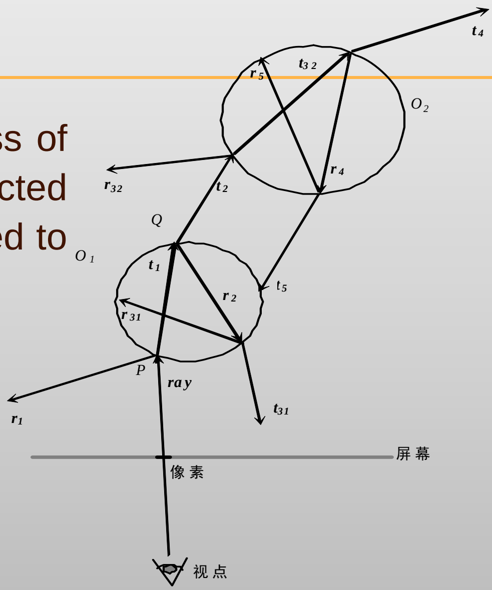
http://alice.loria.fr/publications/papers/2007/ISVC_torus//photo/torus01.png

Example

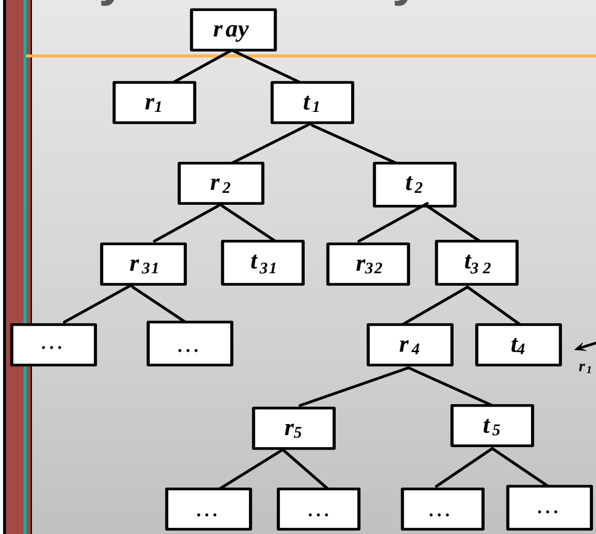


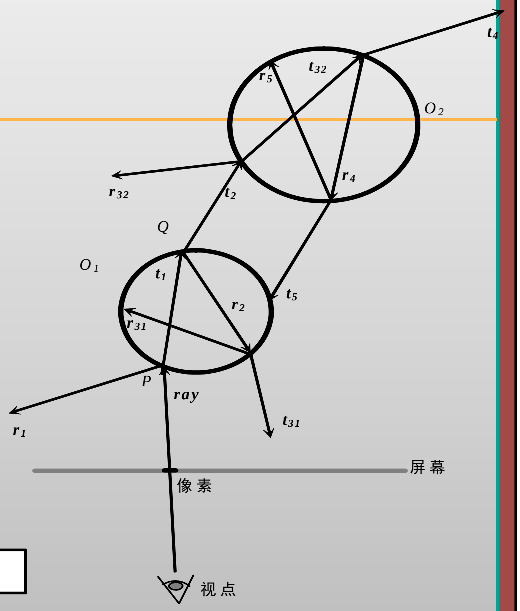
Idea of ray tracing

Inversely tracing the process of light being reflected and refracted multiple time and finally casted to the eye



Ray tree of a ray





Recursion termination

Condition 1: The ray does not intersect with any object, or intersects with pure diffusion plane

Condition 2: The contribution of the ray is small enough

Condition 3: The recursive depth reaches maximum

Algorithm of ray tracing

For each **pixel *p*** in the image

- Step 1: Shoot a **ray R** from viewpoint to pixel p
- Step 2: Compute all intersections between R and the scene, and find the visible one, **P**
- Step 3: Compute the color Ic of P using Phong model
-

Algorithm of ray tracing

For each **pixel *p*** in the image

- Step 4: Cast rays from the directions of reflection and refraction from P
  - *The surface is opaque, stop*
- Step 5：Recursive compute Ir and It(contribution from the environment)
- Step 6：p🡐Ic + Ir +It

Pseudo code (1)

```
main ( )
{   for( each pixel) {
        create ray R from viewpoint V to the pixel;
        depth = 0;
        ratio = 1.0;  // attenuation of the light
         RayTrace(R, ratio, depth, color);
        pixel  ← color;
    }
}
```

Pseudo code (2)

```
RayTrace(R, ratio, depth, color) //
{
    if(ratio < THRESHOLD) {
         color ← 0;  return;
    }
    if(depth > MAXDEPTH) {
         color ← 0;  return;
    }
    // to be continued
```

Pseudo code (3)

Compute all intersections between ***R*** and the scene;

find the nearest one *P*;

if(no intersection) {

    color ← *0 ;*   //set as black

    return;

}

local_color ← *local illumination*;

    // Ray casting

Pseudo code (4)

if(intersection *P* is smooth) {

    calculate reflection ***Rr***;

    RayTrace(***Rr***, *ks*ratio*, *depth*+1, *reflected_color*);

}

if(intersection *P* is transparent) {

    calculate tranparency ***Rt***;

    RayTrace(***Rt***, *kt*ratio*, *depth*+1,*transmitted_color*);

}

Pseudo code (5)

```
    combine the final color:
    color = local_color + ks*reflected_color + kt*transmitted_color;
    return;
}
```

Recursive Ray Tracing

Computing all shadow and feeler rays is slow

- Stop after fixed number of iterations
- Stop when energy contributed is below threshold

Most work is spent testing ray/plane intersections

- Use bounding boxes to reduce comparisons
- Use bounding volumes to group objects
- Parallel computation (on shared-memory machines)

Summary

Ray casting (direct Illumination)

- Usually use simple analytic approximations for light source emission and surface reflectance

Recursive ray tracing (global illumination)

- Incorporate shadows, mirror reflections, and pure refractions

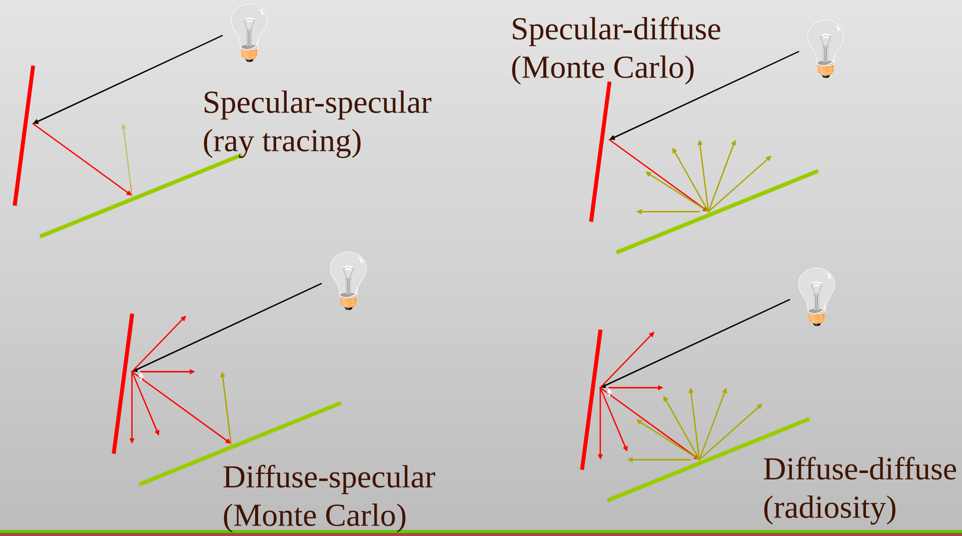
All of this is an approximation
so that it is practical to compute

Radiosity

- Ray tracing models specular reflection and refractive transparency, but still uses an ambient term to account for other lighting effects
- Radiosity is the rate at which energy is emitted or reflected by a surface
- By conserving light energy in a volume, these radiosity effects can be traced



Types of Surface Reflectance



Radiosity

All surfaces are assumed perfectly diffuse

- What does that mean about property of lighting in scene?
  - Light is reflected equally in all directions

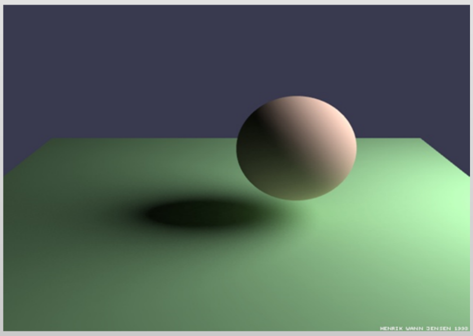
Diffuse-diffuse surface lighting effects possible

Radiosity

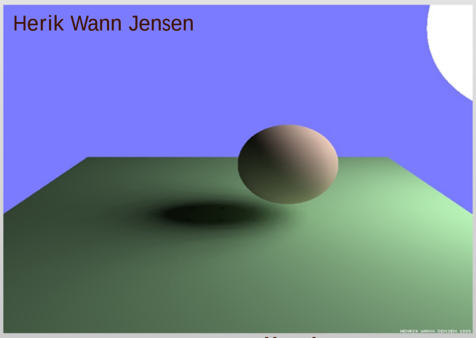
Basic Idea

- We can accurately model diffuse reflections from a surface by considering the radiant **energy transfers between surfaces, subject to conservation of energy laws**.
- Divide surfaces into patches (elements)
- Model light transfer between patches as system of linear equations

Which one is Better



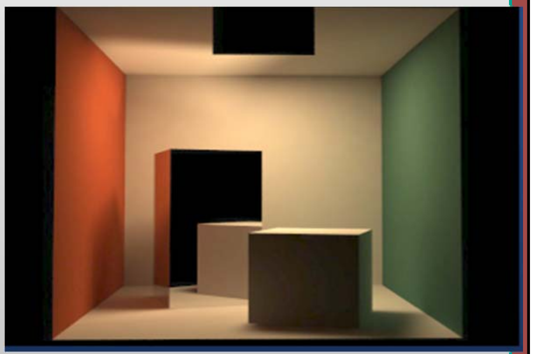
Raytraced



Radiosity

Radiosity: Cornell Experiment

Measured



Simulated

Program of Computer Graphics
Cornell University

Radiosity: Cornell Experiment

Measured

Simulated

Difference

Rendering

- Radiosity is a view-independent solution.
- Could flat shade each patch with colour depending on radiosity at the center
- Instead obtain radiosities at the vertices of the polygons

Ray Tracing vs. Radiosity

Radiosity captures the sum of light transfer well

- But it models all surfaces as diffuse reflectors
- *Can't model specular reflections or refraction*
  - *Images are viewpoint independent*

Ray tracing captures the complex behavior of light rays as they reflect and refract

- Works best with specular surfaces.
  - *Diffuse surface converts light ray into many. Ray tracing follows one ray and does not capture the full effect of the diffusion.*
  - *Must use ambient term to replace absent diffusion*

Radiosity Measure

- Radiant energy (flux) = energy flow per unit time across a surface (watts)
- Radiosity = flux per unit area (a derivative of flux with respect to area) radiated from a surface.
- These are wavelength-dependent quantities.

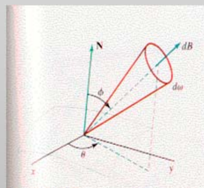


FIGURE 10-88 Visible radiant energy emitted from a surface point in direction (θ, ϕ) within solid angle $d\omega$.

Radiosity Equation

A model for the light reflections from the various surfaces is formed by setting up an "enclosure" of surfaces.

Each surface in the enclosure is either

- a reflector,
- an emitter (light source),
- or a combination reflector-emitter.

We want to calculate radiosity parameter B_i, *the total rate of energy leaving surface i per unit area.*

Radiosity Equation

B_i = total rate of radiant energy leaving surface *i* per unit area

H_i = sum of the radiant energy contributions from all surfaces in the rendered volume arriving at surface *i* per unit time per unit area

$$H_i = \sum_j B_j F_{ji}$$

F_{ji} = the form factor for surfaces *j* and *i*

= the fractional amount of radiant energy from surface *j* that reaches surface *i*.

Architectural design