# Photorealism

- **Ray Tracing**
- **Texture Mapping**
- **Radiosity**

# Photorealism -- Taking into Account Global Illumination

- Light can arrive at surfaces indirectly
- This light called global illumination
- To now we've approximated it with a constant, diffuse ambient term
  - This is wrong
- Need to take into account all the multiply reflected light in the scene
- Two different approaches:
  - Ray Tracing -- specularly reflected light
  - Radiosity -- diffusely reflected light

# Photorealism: Ray Tracing

- See CS-460/560 Notes at:

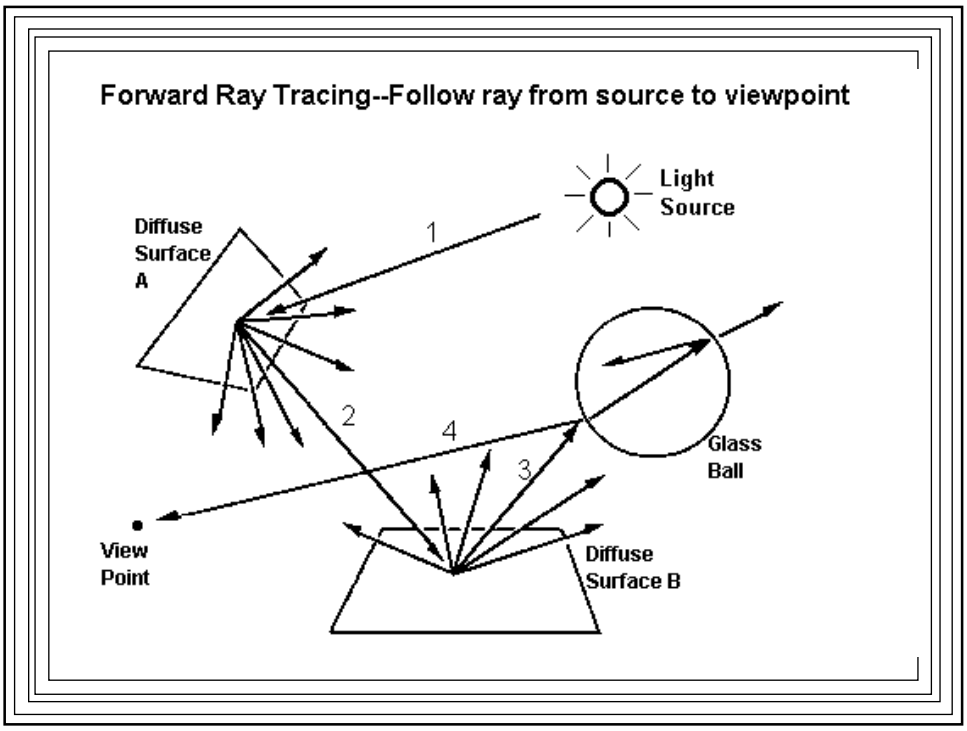
http://www.cs.binghamton.edu/~reckert/460/raytrace.htm
http://www.cs.binghamton.edu/~reckert/460/texture.htm

- Persistence of Vision Ray Tracer (free):

    http://povray.org/

# Ray Tracing

- What is seen by viewer depends on:
  - rays of light that arrive at his/her eye
- So to get “correct” results:
  - Follow all rays of light from all light sources
  - Each time one hits an object, compute the reflected color/intensity
  - Store results for those that go through projection plane pixels into observer’s eye
  - Paint each pixel in the resulting color

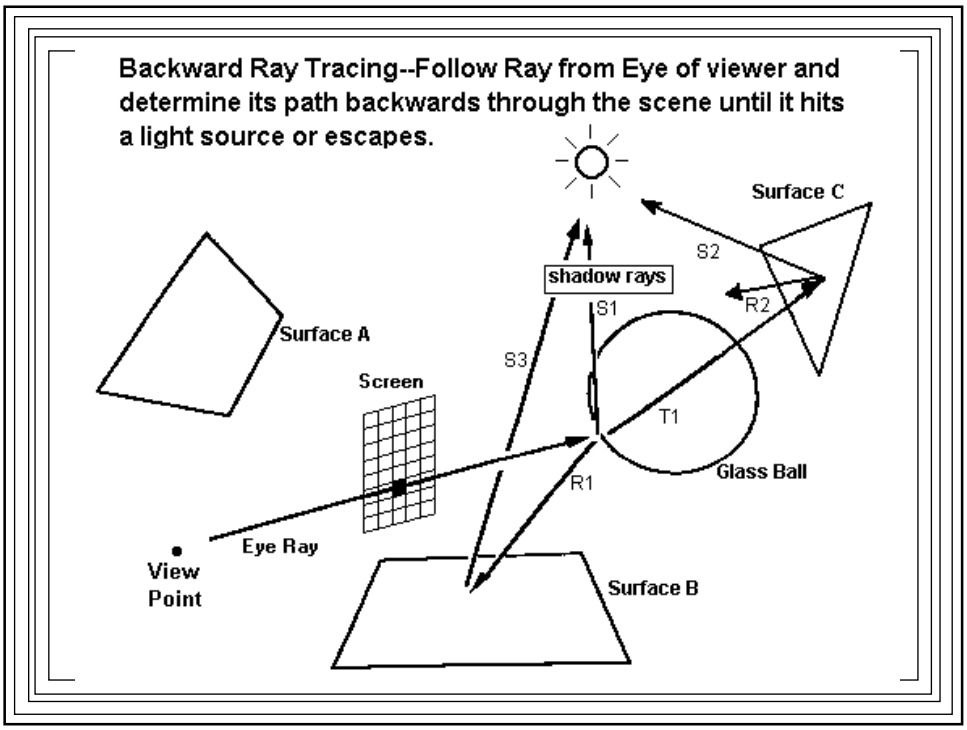



# Forward Ray Tracing

- Infinite number of rays from each source
- At each intersection with an object
  - could have an infinite number of reflected rays
- Completely intractable
- Would take geological times to compute

# Backward Ray Tracing

- Look only at rays observer sees
- Follow rays backwards from eye point through pixels on screen
  - "Eye" rays
- Check if they intersect objects
  - If so, can intersection point see a light source?
    - If so, compute intensity of reflected light
    - If not, point is in the shadow
  - If object has reflectivity/transparency
    - Follow reflected/transmission rays
    - Treat these rays as "eye" rays
    - So it's a recursive algorithm



# Recursive Ray Tracing Algorithm

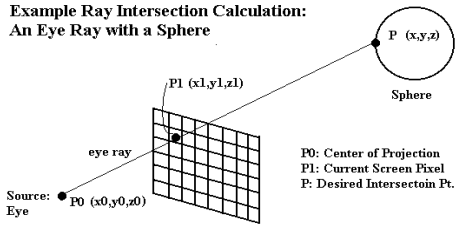
depth = 0

for each pixel (x,y)

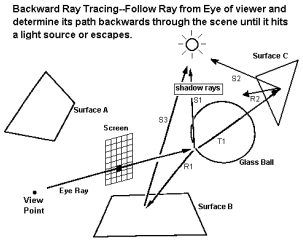
Calculate direction from eyepoint to pixel

TraceRay (eyepoint, direction, depth, *color)

FrameBuf [x,y] = color




```
TraceRay (start_pt, direction_vector, recur_depth, *color)
  if (recur_depth > MAXDEPTH)    color = Background  color
  else
    // Intersect ray with all objects
    // Find int. point that is closest to start_point
          // Hidden surface removal is built in
    if (no intersection)    color = Background_color
    else
        // Send out shadow rays toward light sources
          if (no intersection with other objects)
                local_color = contribution of illumination model at int. pt.
                    // Shadows built in
        // Calculate direction of reflection ray
          TraceRay (int_pt, refl_dir., depth+1, *refl_color)
        // Calculate direction of transmitted ray
          TraceRay (int_pt., trans_dir., depth+1, *trans_color)
        color = Combine (local_color, refl_color, trans_color)
```



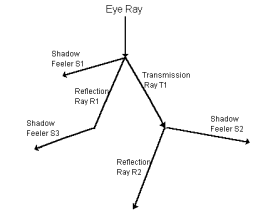
# Combining Color from Reflection Ray

- Add attenuated reflected color intensity to local color intensity:

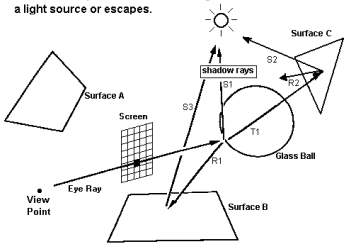
  color = local_color + k * refl_color

  - here refl_color is I(r,g,b) - color returned by reflection ray
  - local_color is I(r,g,b) - color computed by illumination model at intersection point
  - k is an attenuation factor (<1)

The Ray Tree for the above Scene




Backward Ray Tracing--Follow Ray from Eye of viewer and determine its path backwards through the scene until it hits a light source or escapes.



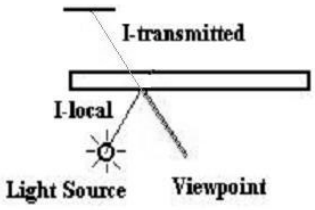

# Combining Color from Transmission Ray

- Observer sees a mixture of light reflected off surface and light transmitted through surface
- So combine colors (interpolate)

  $I(r,g,b) = k'*I_{local}(r,g,b) + (1 - k')*I_{transmitted}(r,g,b)$

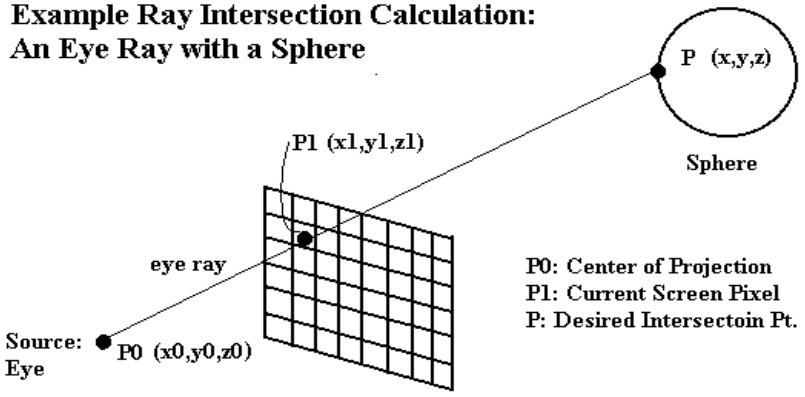
  k' is opacity factor coefficient

  k'=0 => perfectly transparent, k'=1 ==> perfectly opaque



# Ray Tracing Intersection Calculations

**Example Ray Intersection Calculation:**
**An Eye Ray with a Sphere**




# Parametric Equations for Eye Ray:

$x = x0 + (x1-x0)*t$

$x = x0 + \Delta x*t$

$y = y0 + (y1-y0)*t$

$y = y0 + \Delta y*t$

$z = z0 + (z1-z0)*t$

$z = z0 + \Delta z*t$

## Equation of a sphere of radius r, centered at (a,b,c)

$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$

Substitute ray parametric equations:

$(x0+\Delta x*t-a)^2 + (y0+\Delta y*t-b)^2 + (z0+\Delta z*t-c)^2 = r^2$

Rearrange terms:

$(\Delta x^2+\Delta y^2+\Delta z^2)t^2 + 2[\Delta x(x0-a)+\Delta y(y0-b)+\Delta z(z0-c)]t$

$+(x0-a)^2 + (y0-b)^2 + (z0-c)^2 - r^2 = 0$

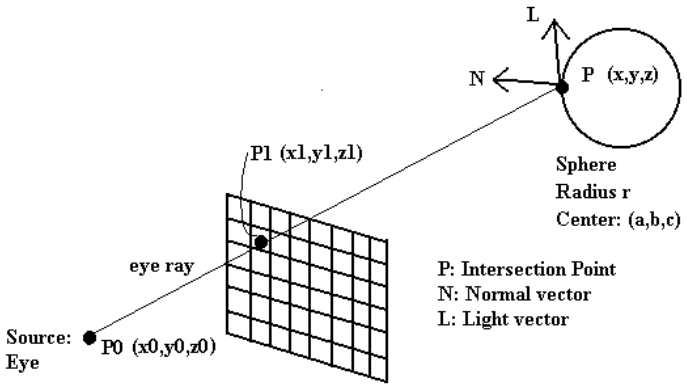
A quadratic in t, use quadratic formula to solve for t

## This is a quadratic in parameter t

- Solution(s): value(s) of t where ray intersects sphere
- Three cases
  - No real roots ==> no intersection point
  - 1 real root ==> ray grazes sphere
  - 2 real roots ==> ray passes thru sphere
    - Select smaller t (closer to source)

# Sphere Normal Computation

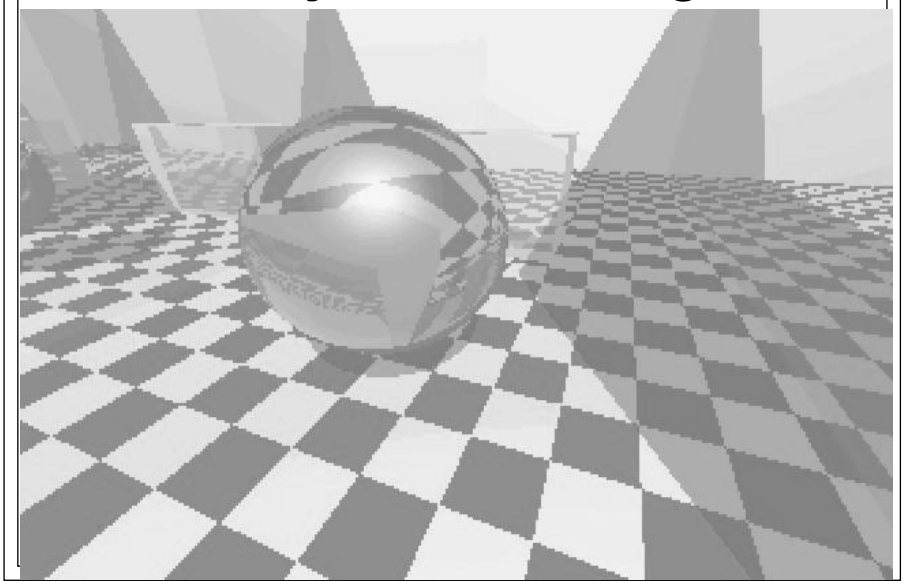
- To apply illumination model at intersection point P, need surface normal
- Can show: N = ( (x-a)/r, (y-b)/r, (z-c)/r )




# Intersections with other Objects

- Handled in similar fashion
- May be difficult to come up with equations for their surfaces
- Sometimes approximation methods must be used to compute intersections of rays with surfaces

# A Ray-Traced Image



# Disadvantages of Ray Tracing

- Computationally intensive
  - But there are several acceleration techniques
- Bad for scenes with lots of diffuse reflection
  - But can be combined with other algorithms that handle diffuse reflection well
- Prone to aliasing
  - One sample per pixel
    - can give ugly artifacts
  - But there are anti-aliasing techniques

# Persistence of Vision Ray Tracer

- POVRay free software
- Lots of capabilities
- Great for playing around with ray tracing
  - http://povray.org/

# An Algorithm Animation of a Ray Tracer

Ray Tracing Algorithm Animator in VC++ (with David Goldman)

See:

http://www.cs.binghamton.edu/~reckert/3daape_paper.htm

Ray Tracing Algorithm Animation Java Applet and Paper (with Brian Maltzan)

See:

http://www.cs.binghamton.edu/~reckert/brian/index.html

# Pattern/Texture Mapping

- Adding details or features to surfaces
  - (variations in color or texture)

# General Texture Mapping

- Pattern Mapping Technique
  - Modulate surface color calculated by reflection model according to a pattern defined by a texture function
    - (“wrap” pattern around surface)
    - Most common: a 2-D Texture function: T(u,v)
      - Define in a 2-D texture space (u,v)
      - Could be a digitized image
      - Or a procedurally-defined pattern

# Inverse Pixel Mapping (Screen Scanning)

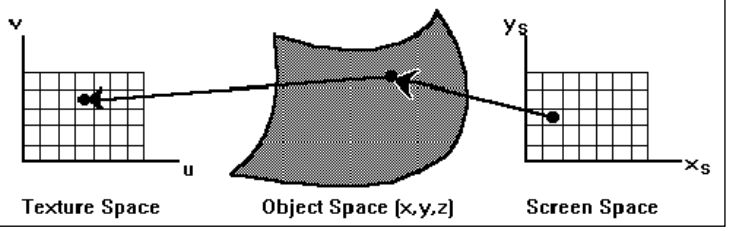
For each pixel on screen (xs, ys)

Compute pt (x,y,z) on closest surface projecting to pixel (e.g., ray tracing)

Determine color (e.g., apply illumination/reflection model)

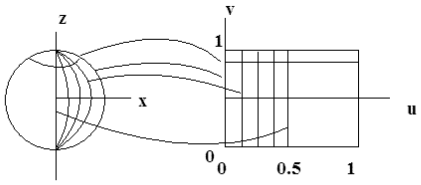
Compute (u,v) corresponding to (x,y,z) (inverse mapping)

Modulate color of ( xs,ys ) according to value of T(u,v) at (u,v)




# Inverse Mapping a Sphere

- Lines of longitude: constant u, corresponds to theta
- Lines of latitude: constant v, corresponds to phi

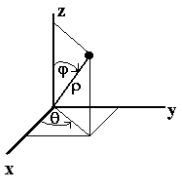



x=R sin(phi) cos(theta)
y=R sin(phi) sin(theta)
z=R cos(phi)

phi = arccos(z/R) = arccos(N.z)
theta = arccos(x/R sin(phi))
theta = (arccos(N.x/sin(phi))

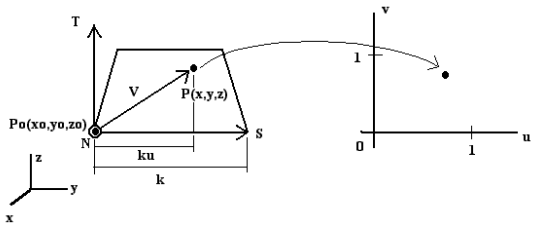
v = (pi-phi)/pi  phi=0, North Pole, v=1; phi=pi, South Pole, v=0

u = theta/2*pi if N.y is +  theta=0, u=0; theta=pi, u=0.5  (+y side)
u = 1-theta/2*pi if N.y is -  theta=pi, u =0.5; theta=0, u=1.0  (-y side)

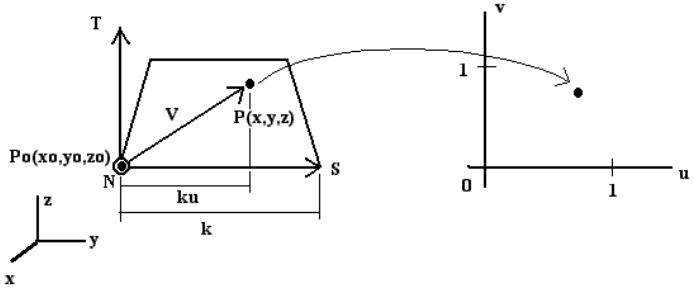


# Ex: Inverse Mapping a Polygon

- Choose axis S (unit vector) along a polygon edge
  - will align with u-axis in texture space
- Choose a polygon vertex Po(xo,yo,zo)
  - will correspond to origin in texture space
- Choose a scaling factor k
  - k = max dimension of polygon
    - 0-k in object space --> 0-1 in texture space
- Want to map point P(x,y,z) on polygon to (u,v)




- Construct vector V = P - Po
- V·S = k*u, projection of P onto S
- So u = V·S/k
- Choose orthogonal axis T in polygon plane (T=NxS)
- v=V·T/k



# There are lots of other Texture Mapping Techniques

See Section 10-17 of text book

OpenGL also has extensive support for texture mapping

See Section 10-21 of text book

# Radiosity Methods

- Alternative to ray tracing for handling global illumination
- Two kinds of illumination at each reflective surface
  - Local: from point sources
  - Global: light reflected from other surfaces in scene (multiple reflections)
  - Ray tracing handles specularly reflected global illumination
    - But not diffusely reflected global illumination

# Radiosity Approach

- Compute global illumination
- All light energy coming off a surface is the sum of:
  - Light emitted by the surface
  - Light from other surfaces reflected off the surface
  - Divide scene into patches that are perfect diffuse reflectors...
    - Reflected light is non-directional
  - and/or emitters

# Definitions

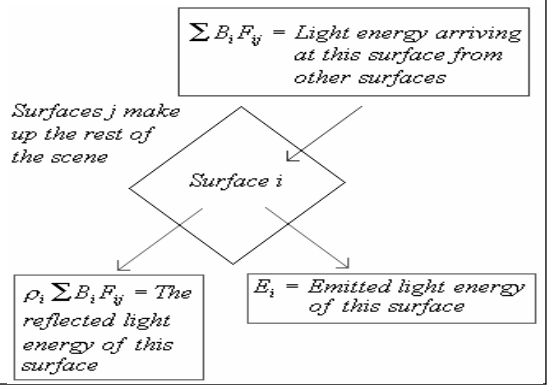
- Radiosity (B):
  - Total light energy per unit area leaving a surface patch per unit time--sum of emitted and reflected energy (Brightness)
- Emission (E):
  - Energy per unit area emitted by the surface itself per unit time (Surfaces having nonzero E are light sources)

- Reflectivity ( ρ ):
  - Fraction of light reflected from a surface (a number between 0 and 1)
- Form Factor (Fij):
  - Fraction of light energy leaving patch i which arrives at patch j
    - Function only of geometry of the environment

- Conservation of energy for patch i:
  - Total energy out = Energy emitted + Sum of energy from other patches reflected by patch i:

$$Bi*Ai = Ei*Ai + \rho_i * \Sigma_j Bj*Aj*Fji$$

$$Bi = Ei + \rho_i * \Sigma_j (Bj*Aj/Ai)*Fji$$



- Principle of Reciprocity for Form Factors
  - Reversing role of emitters and receivers:
    - Fraction of energy emitted by one and received by other would be same as fraction of energy going the other way
    - $Fij*Ai = Fji*Aj \quad ==> \quad Fji = (Ai/Aj)\ Fij$
    - So:

      $$Bi = Ei + \rho_i * \Sigma\ Bj*Fij$$

      $$Bi - \rho_i * \Sigma\ Bj*Fij = Ei$$

    - Need to solve this system of equations for the radiosities Bi

## Radiosity -- Matrix Formulation and Solution

**Assume N patches**

**$F_{ii} = 0$ – Patch i receives no energy from itself**

**Rearranging and writing out the Radiosity equation:**

$$\begin{bmatrix} 1 & -\rho_1 F_{12} & -\rho_1 F_{13} & \cdots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1 & -\rho_2 F_{23} & \cdots & -\rho_2 F_{2N} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & -\rho_N F_{N3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdots \\ \cdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \cdots \\ \cdots \\ E_N \end{bmatrix}$$

**This is the matrix equation: $M\ B = E$**

**The $E_i$ and $\rho_i$ are known, and the form factors $F_{ij}$ can be calculated so that the matrix elements $M_{ij}$ can be determined. Since the $\rho_i$ and $F_{ij}$ are all less than or equal to zero, matrix M is diagonally dominant, so the Gauss-Seidel iteration method is guaranteed to converge after a few iterations.**

# Gauss-Seidel Solution

$B_1 + M_{12}B_2 + \ldots\ldots + M_{1N}B_N = E_1$

$M_{21}B_1 + B_2 + \ldots\ldots + M_{2N}BN = E_2$

---

$M_{i1}B_1 + M_{i2}B_2 + \ldots + B_I + \ldots + M_{IN}B_N = E_I$

---

Initial guess: $B_i^{(0)} = E_i$

Next iteration $B_I^{(1)} = E_i - ( M_{i1}B_i^{(0)} + \ldots + M_{i,i-1}B_{i-1}^{(0)} + \ldots + M_{i,i+1}^{(0)} + \ldots + M_{iN}B_N^{(0)} )$

Continue iterating until all the difference between $B_i^{(k+1)}$ and $B_i^{(k)}$ is small enough for all patches

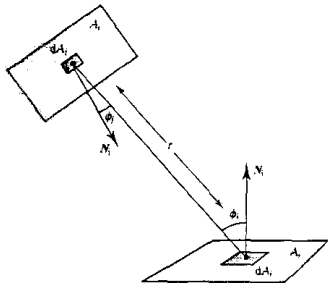
Notice that for each patch i we are "gathering" radiosity from all the other patches

Thus the scene is not finished until we have processed all patches--very time consuming.

**Problem: getting form factors**

# Computing Form Factors

Form Factor Determination



The form factor $F_{ij}$ from patch i to patch j is obtained in principle by by integrating over both patches:

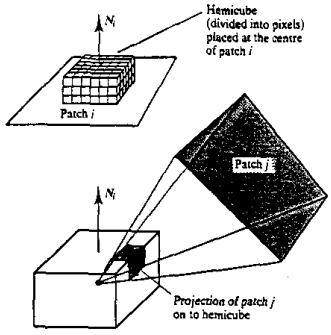
$$F_{dA_i} = F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j \, dA_i$$

But this is very difficult.

$\Delta FF = \Sigma (\cos \phi_i \cos \phi_j \Delta A_j) / (\pi r^2)$, approximately

# Hemicube Approximation

**The Hemicube Approximation**




Calculate and store the delta form factors for each element of the hemicube.

$$F_{ij} = \sum_q \Delta F_q$$

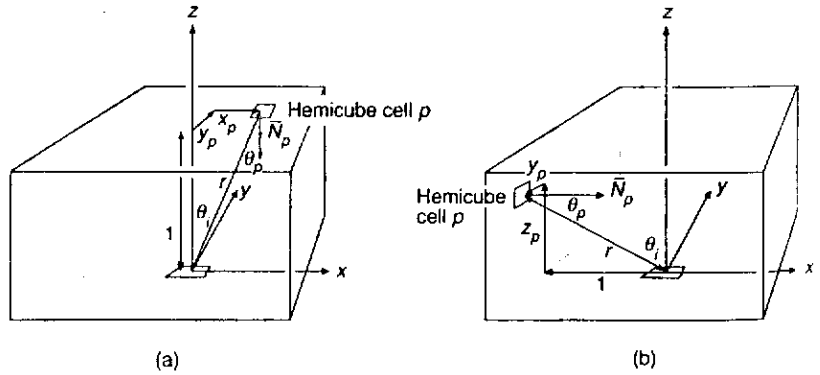
To calculate the form factor $F_{ij}$, build a hemicube centered on patch i and sum the delta form factors for those elements of the hemicube to which patch j projects.

# Hemicube Pixel Computations

$\Delta F = \cos\theta_i * \cos\theta_p * \Delta A / (\pi * r^2)$

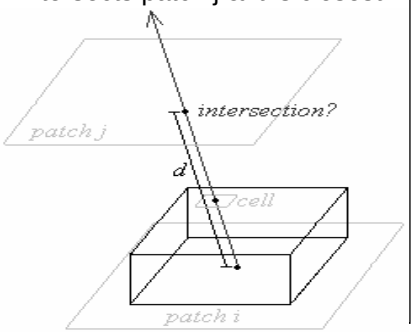
a. (top) $\Delta F = \Delta A / [\pi * (x^2 + y^2 + 1)^2]$

b. (sides) $\Delta F = z * \Delta A / [\pi * (y^2 + z^2 + 1)^2]$

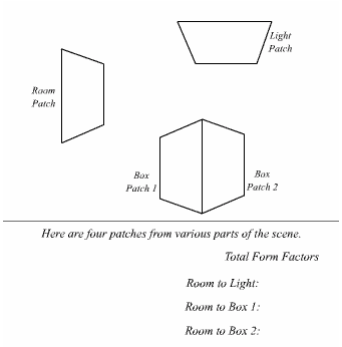


# Hemicube Form Factor Algorithm

Compute & strore all hemicube Delta-Form-Factors: ΔFF[k]
Zero all the Form Factors: Fij
For each patch i
  Place a unit hemicube at center of patch i
  For each patch j (j <> i)
    For each cell k on hemicube
      If line from origin through cell k intersects patch j & it is closest
        Fij = Fij + ΔFF[k]




# Video of Radiosity Form Factor Computation



# Steps in Applying Radiosity Method

*Summary of Steps:*

*Define a scene.*

*Divide scene into distinct patches.*

*Build a hemicube on each patch and calculate the delta form factors for cell on every hemicube.*

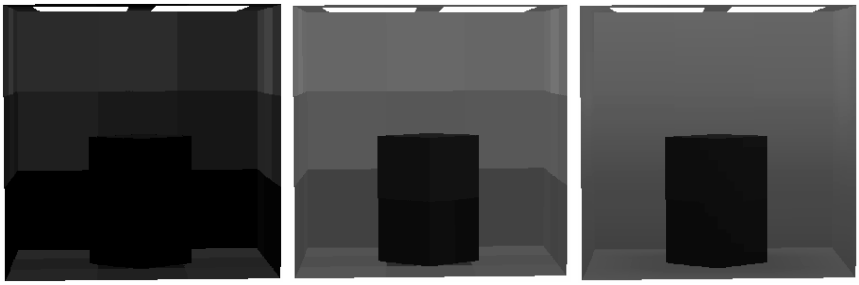
*Calculate a form factor for every pair of patches in the scene.*

*Calculate the red, green, and blue radiosities for each patch.*

*Map all radiosity values to a 0 - 255 color scale.*

*Apply Gouraud shading.*

# Three Simple Radiosity Images



After 1st Gauss-Seidel Iteration

No Gouraud Shading

Gouraud Shading

# Radiosity Summary

- Good for scenes with lots of diffuse reflection
- Not good for scenes with lots of specular reflection
  - Complementary to Ray Tracing
  - But can be combined with Ray Tracing
- Very computationally intensive
  - Can take very long times for complex scenes
    - but once patch intensities are computed, scene “walkthroughs” are fast
  - Gauss-Seidel is very memory intensive
  - There are other approaches
    - Progressive Refinement