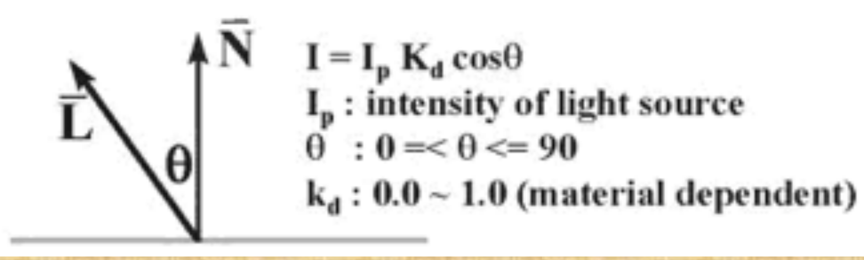
Shading

Ming Ouhyoung
歐陽明 Professor
Dept. of CSIE and GINM
NTU

Illumination model

1) **Ambient light** (漫射)
   I = Ia * ka * Obj(r, g, b)
   Ia : intensity of ambient light
   ka : 0.0 ~ 1.0, Obj(r, g, b): object color
2) **Diffuse reflection** (散射)
   I = Ip (r, g, b) * Kd * Obj(r, g, b) * COS(θ)
   Ip (r, g, b): light color

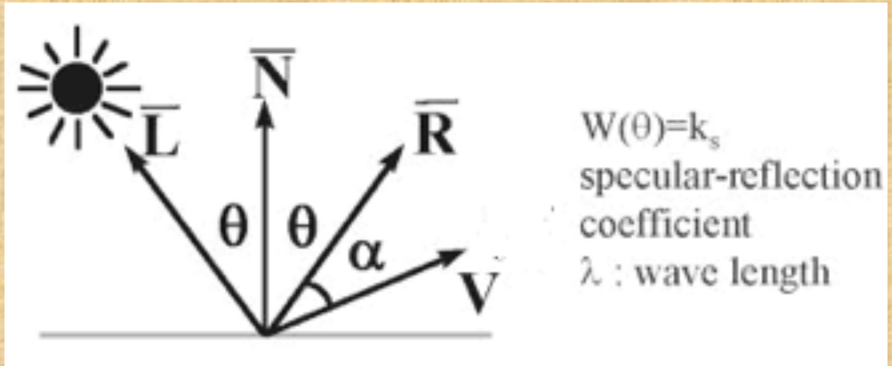


3) **Light source attenuation**

$$I = I_a k_a + fatt\, I_p K_d (\bar{N}.\bar{L}) \quad fatt = \frac{1}{d_L^{\,2}}$$

Specular reflection (似鏡面反射)

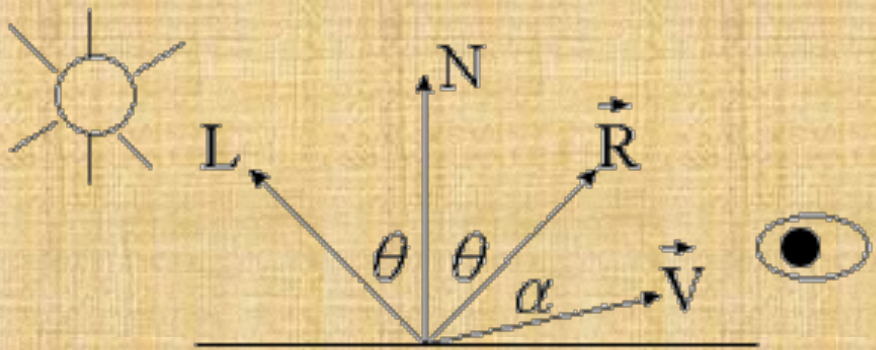
- **I = Ks * Ip(r, g, b) * COS^n(α),**
  Ks = specular-reflection coef.



- Phong illumination model
  color of object = Obj (R, G, B)
  = (Or, Og, Ob) or (light frequency)
  where 0.0 =< Od =< 1.0

Faster specular reflection calculation: Halfway vector approximation

- halfway vector



$$\vec{H} = \frac{\vec{L}+\vec{V}}{|\vec{L}+\vec{V}|}$$

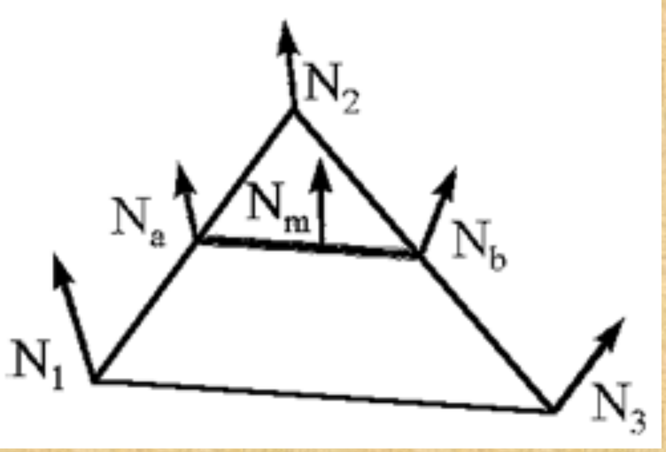
Polygon shading : linear interpolation

a. flat shading : constant surface shading.
b. Gouraud shading: color interpolation shading.
c. Phong shading: vertex normal interpolation shading

Phong Shading

- Use a big triangle, light shot in the center, as an example!
- The function is really an approximation to Gaussian distribution

macroscopic



- The distribution of microfacets is Gaussian. [Torrance, 1967] (Beckmann distribution func.)
- Given normal direction N_a and N_b, N_m = ?
  - interpolation in world or screen coordinate?
  - in practice

Bi-linear interpolation

. Linear interpolation:

A (x1, y1, z1) with color (r1, g1, b1); B (x2, y2, z2) with color (r2, g2, b2)

What is the color of point C (x3, y3, z3) located on the line AB.

C = color of A + t * (color of B- color of A), where t is | (C-A) |/ |(B-A)|

Similarly, we can process Bi-linear interpolation

Gouraud Shading with Bilinear Interpolation

- Gouraud shading (smooth shading): color interpolation, for example,

  Triangle with three vertices (x1, y1), (x2, y2),

  (x3, y3), each with red components R1, R2, R3

  color is represented as (Red, Green , Blue)

  Assuming a plane (in 3D) with vertices (x1, y1, R1), (X2, Y2, R2), and (X3, y3, R3)

Gouraud shading

- Vector equation of the plane is

  $(x,y) = s\ (x2-x1,\ y2 - y1) + t(x3-x1,\ y3-y1) + (x1,\ y1)$

  solved for (s, t) , then

  $s = A1x + B1y + C1, \quad t = A2x + B2y + C2$

  So, given point (x,y) in this plane, what is its color?

Answer: color of $(x,y) = R1 + s\ (R2-R1) + t\ (R3-R1)$, or

color $= Ax + By + C$, where

$A = A1\ (R2-R1) + A2\ (R3-R1)$

$B = B1(R2-R1) + B2\ (R3-R1)$

$C = C1(R2-R1) + C2(R3-R1) + R1$

How is the color calculated?

Since, $(x,y) = s\,(x2-x1,\ y2 - y1) + t(x3-x1,\ y3-y1) + (x1,\ y1)$

Therefore

, $(x,y,\ R) = s\,(x2-x1,\ y2 - y1,\ R2-R1) + t(x3-x1,\ y3-y1,\ R3-R1) + (x1,\ y1,\ R1)$

or, color $R = s\,(R2-R1) + t\,(R3-R1) + R1$

Complexity of visibility test

TEST FOR I N P U T

Width: W
Height: H
Triangle Area: A
Number of Triangle: N

Complexity:

One time lighting: 6 multiplication, 2 addition, table look up (Cosine alpha): min.

Flat shading:
N* one time lighting

Gouraud Shading:
N*(3*one time lighting + bi-linear interp.*A)

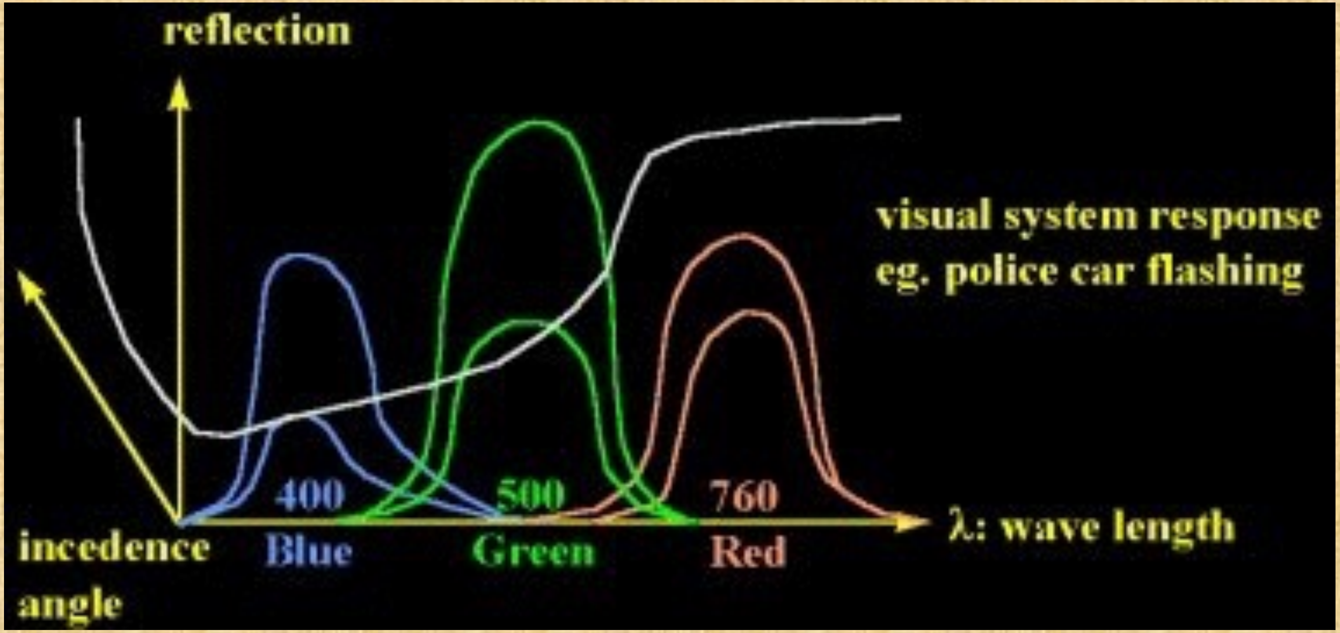
Phong Shading:
1
(bi-linear interp. + one time lighting)*N*A
A >> 3 ingeneral

Phong: under Ivan Sutherland

- **Bùi Tường Phong** (Vietnamese: *Bùi Tường Phong*, December 14, 1942–1975) was a Vietnamese-born computer graphics researcher and pioneer.
- He came to the University of Utah College of Engineering in September 1971 as a research assistant in Computer Science and he received his Ph.D. from the University of Utah in 1973.
- Phong knew that he was terminally ill with leukemia while he was a student. In 1975, after his tenure at the University of Utah, Phong joined Stanford as a professor. He died not long after finishing his dissertation

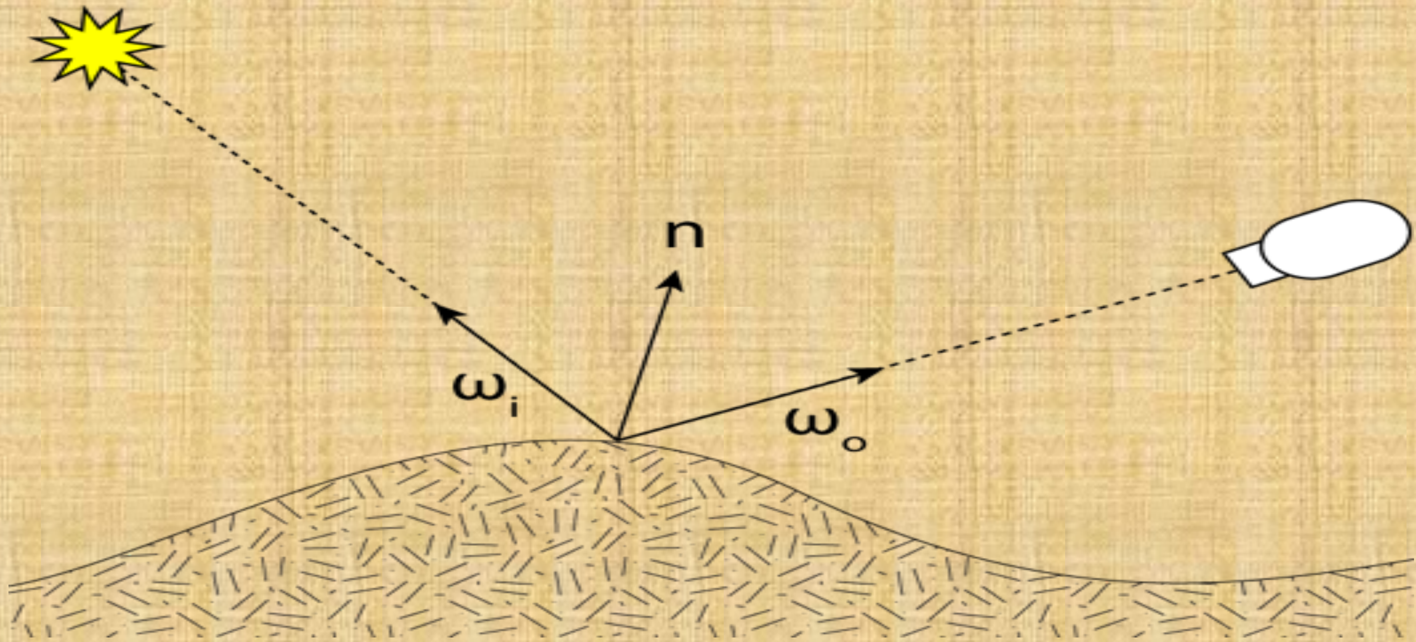
What is the color of copper?

- Reflection of copper
  - drastic change as a function of incidence angle [Cook, 82"]

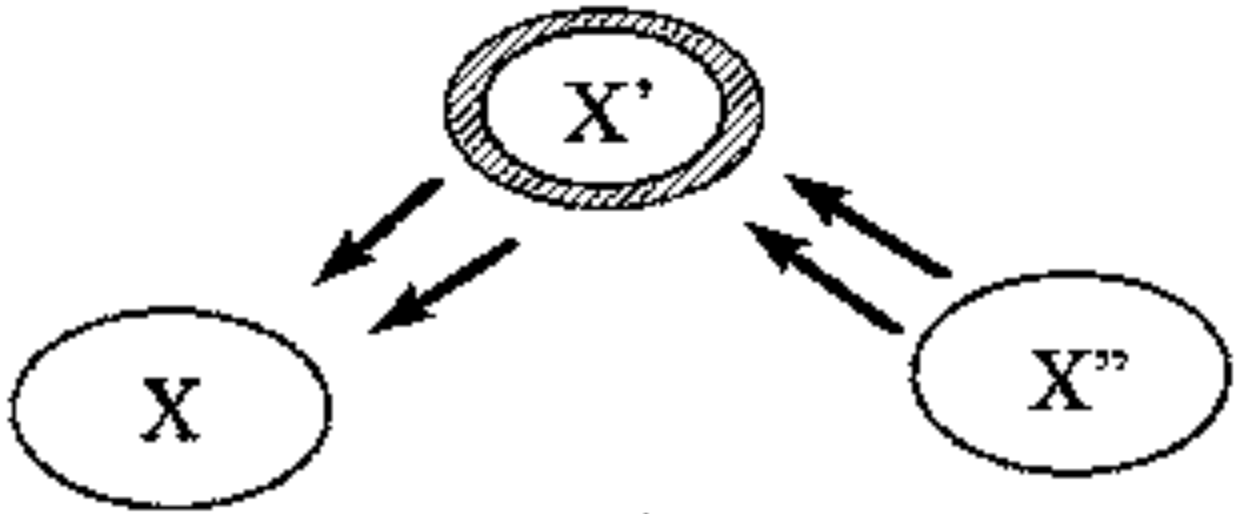


New method: BRDF:Bi-directional Reflectance Density Function

- Use a camera to get the reflection of materials from many angles
- Light is also from many angles



The Rendering Equation: Jim Kajiya



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

BRDF

- BRDF: a four-dimensional function that defines how light is reflected at an opaque surface.
- The BRDF was first defined by Edward Nicodemus around 1965[1]. The modern definition is:

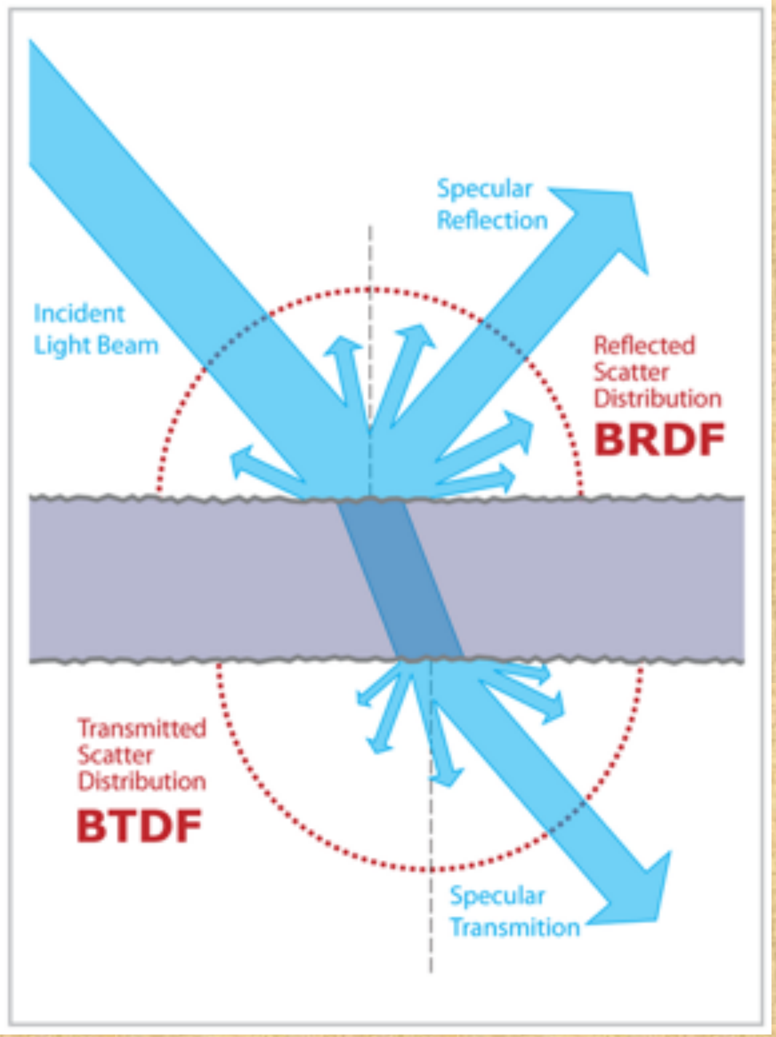
$F(\omega_{i,}\ \omega_o) = dL(\omega_o)/dE(\omega_i) = dL(\omega_o)/L(\omega_i)\cos\theta_i\, d\omega_i$

  - where L is the radiance, E is the irradiance, and θ_i is the angle made between ω_i and the surface normal, n.

BSSRDF

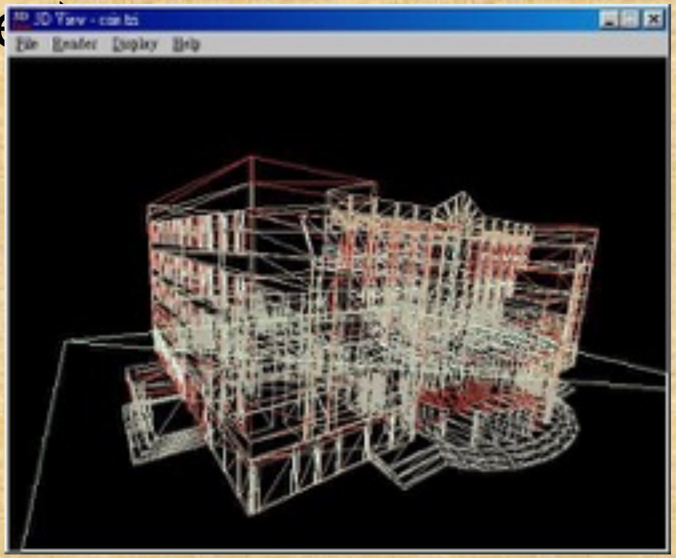
- The **Bidirectional Surface Scattering Reflectance Distribution Function** (BSSRDF), is a further generalized 8-dimensional function $S(X_i,\omega_i,X_o,\omega_o)$, in which light entering the surface may scatter internally and exit at another location. X describes a 2D location over an object's surface.
- non-local scattering effects like shadowing, masking, interreflections or subsurface scattering.

SSBRDF, **BSDF (Bidirectional scattering distribution function)**



Homework #1

- Input : a file of polygons (triangle
- test image : a teapot, a tube
- input format :



 Triangle fr, fg, fb, br, bg, bb
 x y z nx ny nz
  x1, y1, z1, ….   ,
 ,X2, y2, z2 …….,
 /* where (fr, fg, fb) contains front face colors,
   (br, bg, bb) are background colors
   (x,y,z): 3D vertex position
   (nx,ny,nz) : vertex normal

Hw#1 requirements

- **Deadline Oct. 24**
- Output: lines with colors
- Rotation, Scaling, Translation, Shear
- Clipping (front and back, left and right, top and bottom)
- Camera: two different views
  - Object view and camera view
- C, C++, Java, etc.
  - Limited open-GL library calls

Polygon file format used

- e.q.
  Triangle fr fg fb br bg bb
  x1 y1 z1 nx1 ny1 nz1
  x2 y2 z2 nx2 ny2 nz2
  x3 y3 z3 nx3 ny3 nz3
  Triangle
  - Where

    fr, fg, fb are foreground colors (Red Green Blue)
    nx, ny, nz are vertex normal

Other formats (more efficient)

- Vertices
  1, (x, y, z)
  2, (x1, y1, z1)
  3, …
  23, …
  890, ….
  1010
- Triangle 1010, 23, 890
- Triangle 1, 2, 800

Visible-Surface Determination

- **The painter's algorithm**
- **The Z-buffer algorithm**
  - The point nearest to the eye is visible,.....
  - Very easy both for software and hardware.
  - Hardware Implementation: Parallel ---> fast display
- **Scan-line algorithms**
  - One scan line at a time
- **Area-subdivision algorithm**
  - Divide and conquer strategy
- **Visible-surface ray tracing**

List-priority algorithms

- Depth-sort algorithm

  sort by Z coord.(distance to the eye),

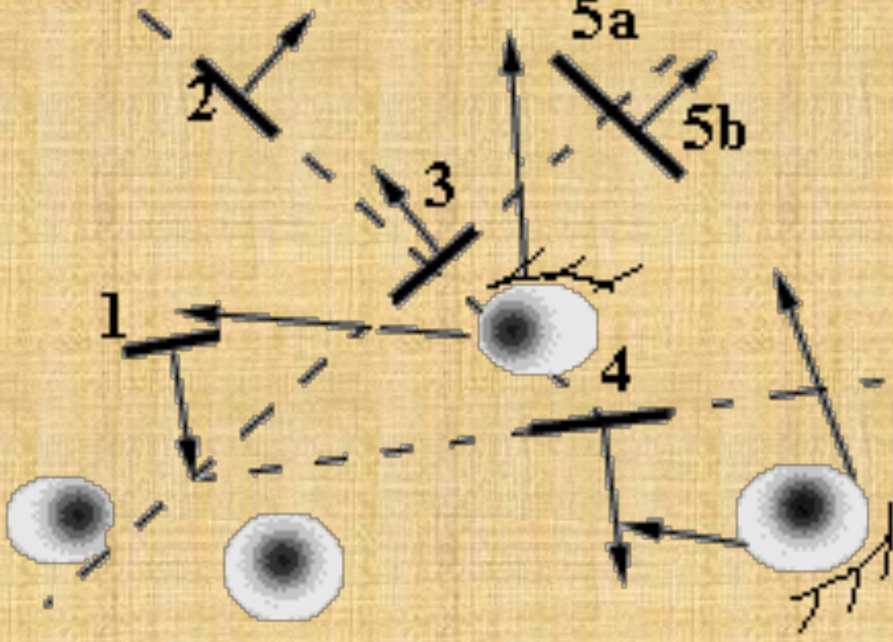
  resolve conflicts(splitting polygons), scan convert·

  ---v.s.---painter's algorithm Binary Space Partition

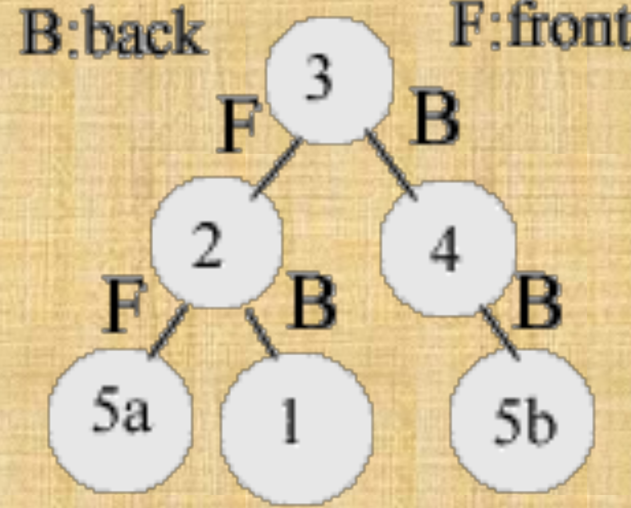
- Trees(BSP tree)

Determine the depth order!
1. Mountain, 2. bridge, 3. people 4. Hat

BSP



ref. Fuchs 80,83



The Display Order of Binary Space Partition Trees(BSP tree)

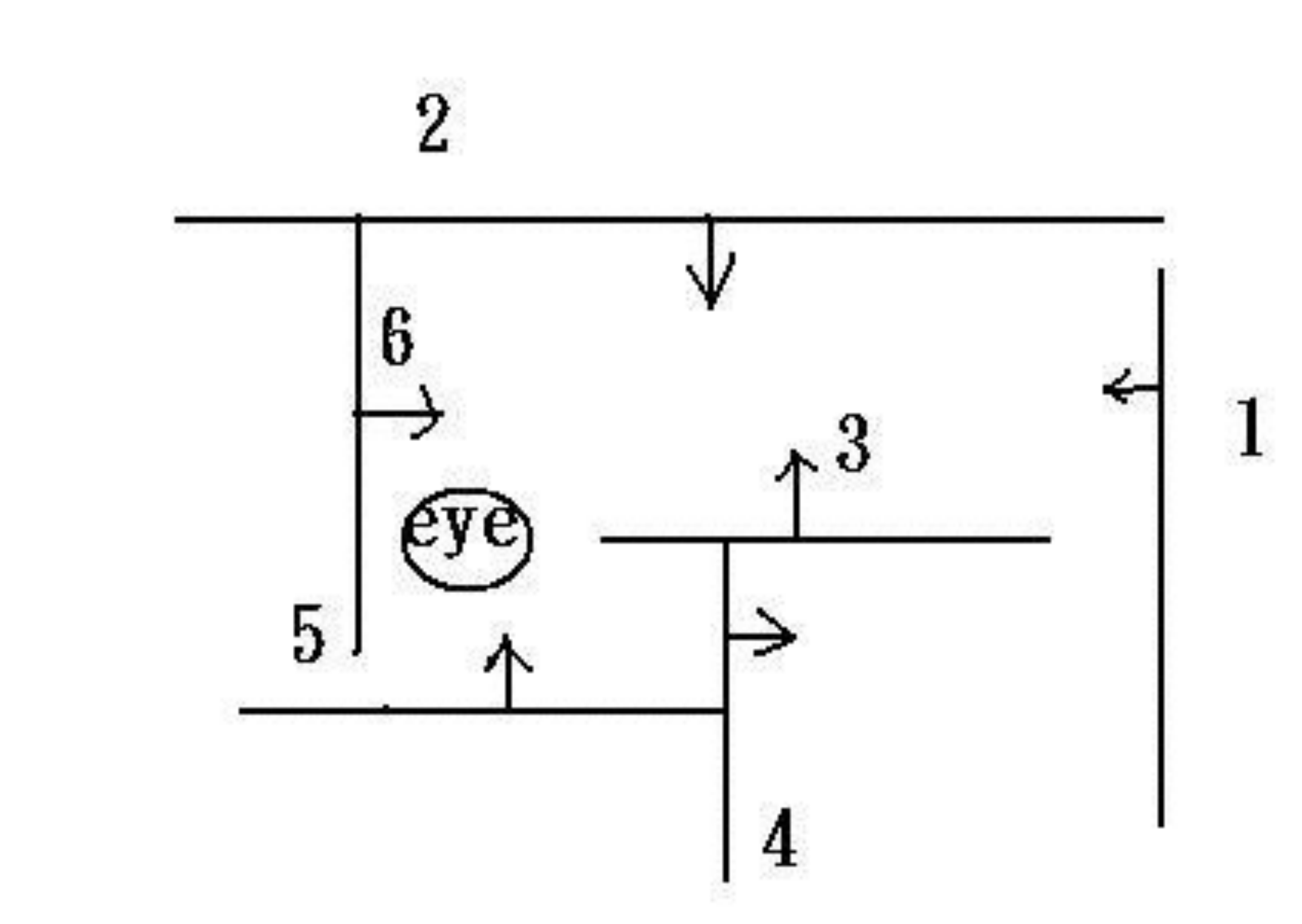
if Viewer is in front of root, then

- Begin   {display back child, root, and front child}
-     BSP_displayTree(tree->backchild)
-     displayPolygon(Tree->root)
-     BSP_displayTree(tree->frontchild)
- end

else

- Begin
-     BSP_displayTree(tree->frontchild)
-     displayPolygon(Tree->root)
-     BSP_displayTree(tree->backchild)
- end

Test: please give the BSP binary tree, and display order of this diagram. (Choose smaller number as the new root)



Visibility determination(2): Z-buffer algorithm

Initialize a Z-buffer to infinity (depth_very_far)

```
{
   Get a Triangle, calculate one point's  depth from
   three vertices by linear interpolation

   If the one point's  depth   depth_P(x,y)  is smaller
   than Z-Buffer(x,y)
         Z-Buffer (x,y) = depth_P(x,y),
         Color_at(x,y) = Color_of_P(x,y)
   else
         DO_NOTHING
 }
```

Complexity of visibility test

```
TEST      Width:  W
FOR I     Height:  H
N P       Triangle Area:  A
U T       Number of Triangle:  N
```

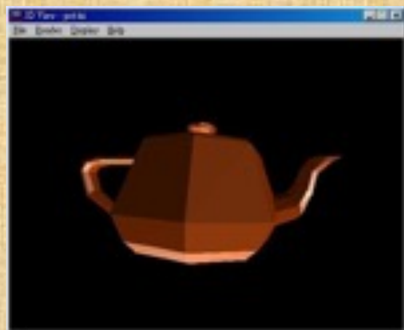
```
Complexity:        One time lighting:  6 multiplication
                                       2 addition, table
Flat shading:                           look up (Cosine
                   N* one time lighting   alpha): min.
```

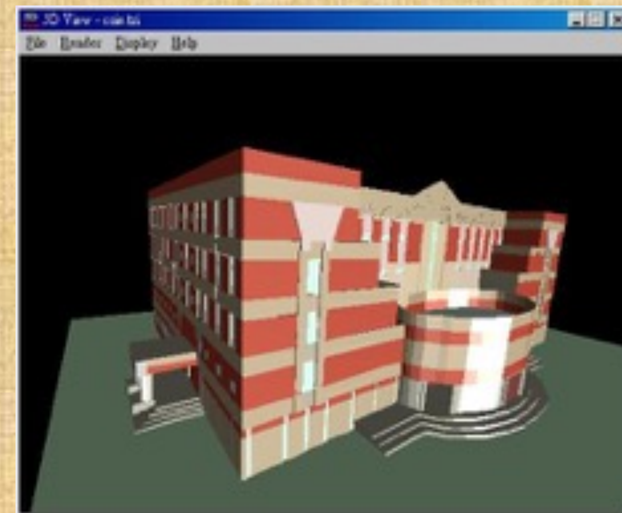
```
Gourand Shading:
      N*(3*one time lighting + bi-linear interp.*A)

Shong Shading:
      1
(bi-linear interp. + one time lighting )*N*A
 A >> 3 ingeneral
```

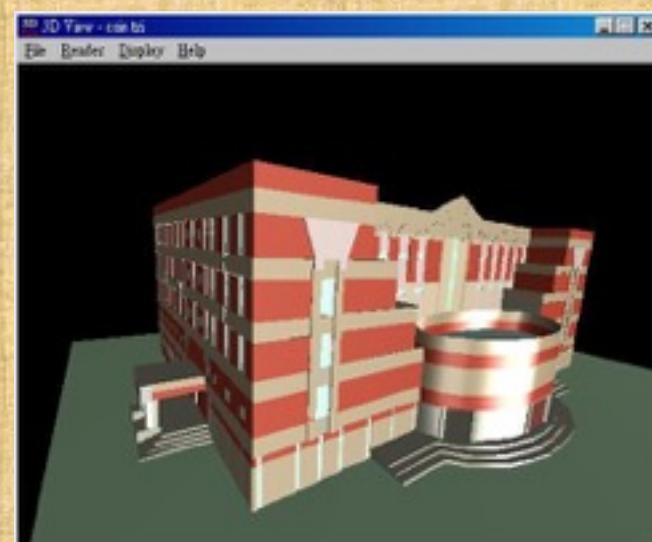
Homework #2 Shading

- Scan convert the teapot which consists of triangles
  - Using Z-buffer algorithm for visible-surface determination
  - Flat, Gouraud shading and Phong shading, three light sources
  - Multiple lights, multiple 3D models (sphere, teapot, CSIE building etc)

HW#2: expected results



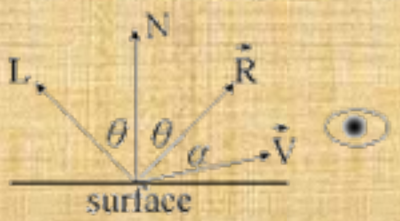




HW#2: formula

- $$I = 0.2\ Ia + 0.6 Ia * Ip(N * L) + 0.2 Ip \cos^{10} \alpha$$

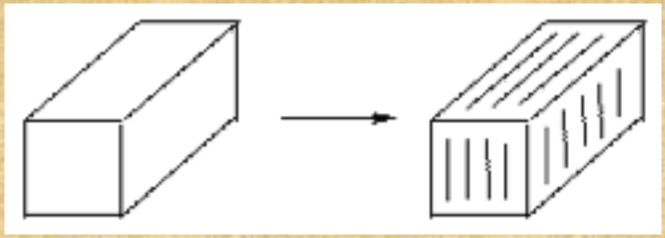
  Where $Ia = \text{object color}$, $Ip = \left(\frac{R}{MAX}, \frac{G}{MAX}, \frac{B}{MAX} \right)$

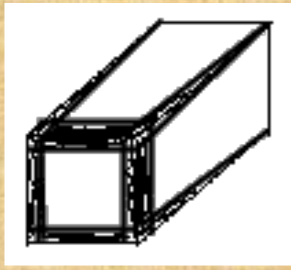
  – Ia is object color
  – Ip is the color of light, and can have multiple lights
- Note
  – color overflow problem (integer color up to 255)
  – MAX = max(R, G, B) = 255 etc.
- Output format
  – RGBx RGBx.... 256*256 pixels
  – better results: 32<=R,G,B<=230, each 1 byte binary data

Visible line determination

1. Assume that visible surface determination can be done fast (by hardware Z-buffer or software BSP tree)
   - This method is used in most high performance systems now!

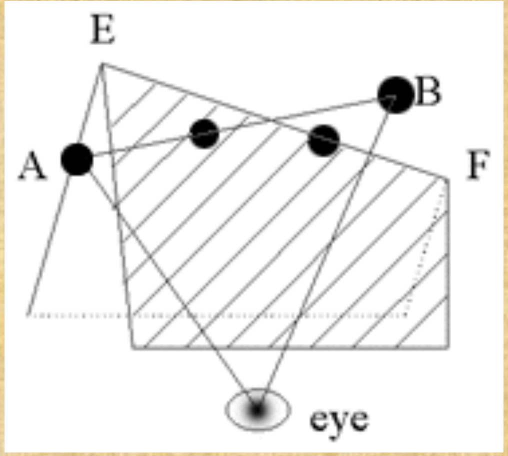


2. Depth cueing is more effective in showing 3D (in vector graphics machine, e.g. PS300). see sec. 14.3.4
   - depth cueing: intensity interpolation

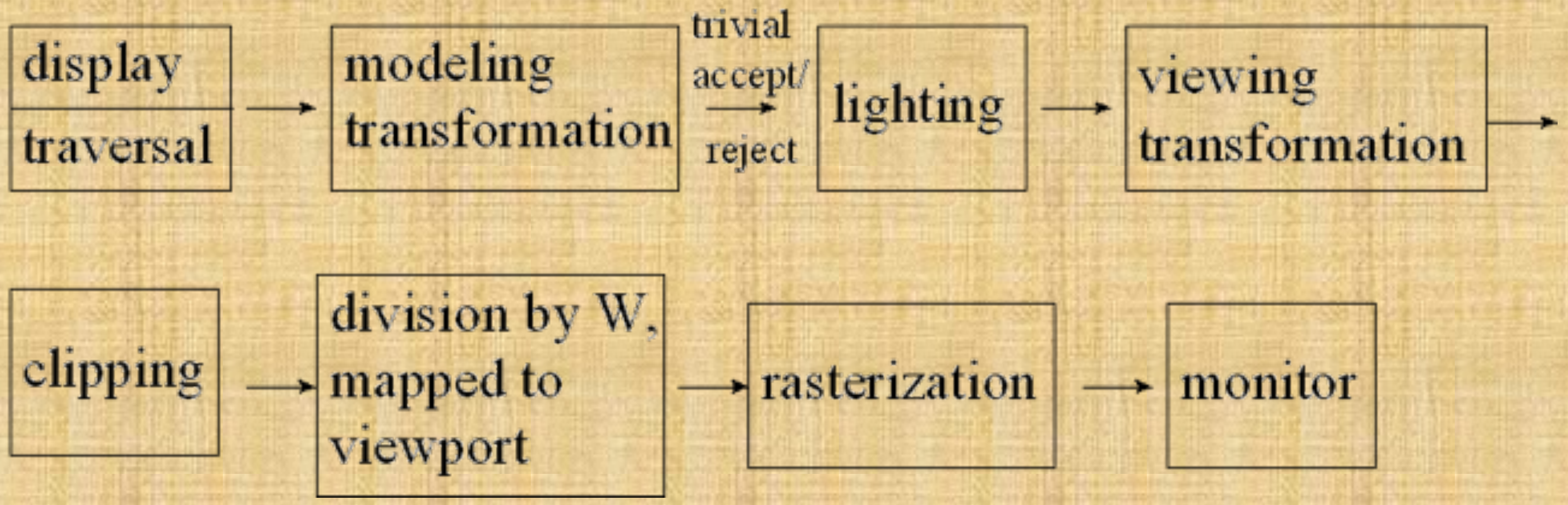


Visible - line determination: Appel's algorithm

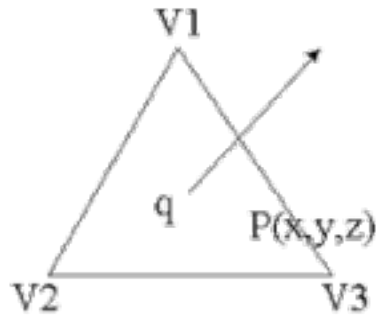
- quantitative invisibility of a point=0 --> visible
- quantitative invisibility changes when it passes "contour line".
- contour line:(define)
- vertex traversing
  - EF: contour line
  - AB: whether this line segment is partially visible?



Standard Graphics Pipeline



How to transform a plane? a surface normal?



$$P = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad N = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix}$$

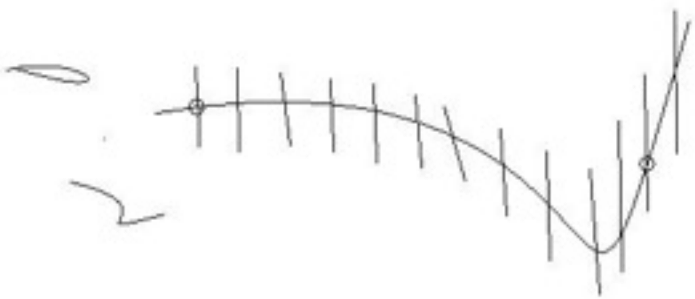
plane equation $Ax + By + Cz + D = 0$
$N^T*P=0$ since p is transformed by M,
How should we transform N?

find Q, such that $(Q*N)^T*M*P=0$ $(N')^T.(P')=0$
i.e. $N^T*Q^T*M*P=0$, i.e. $Q^T*M=I$
$\therefore Q^T=M^{-1}$ $Q=(M^{-1})^T$
similar, the surface normal is transformed by Q, not M!!

Aliasing, anti-aliasing

Aliasing effects

Sampling theory: two
times sampling
frequency



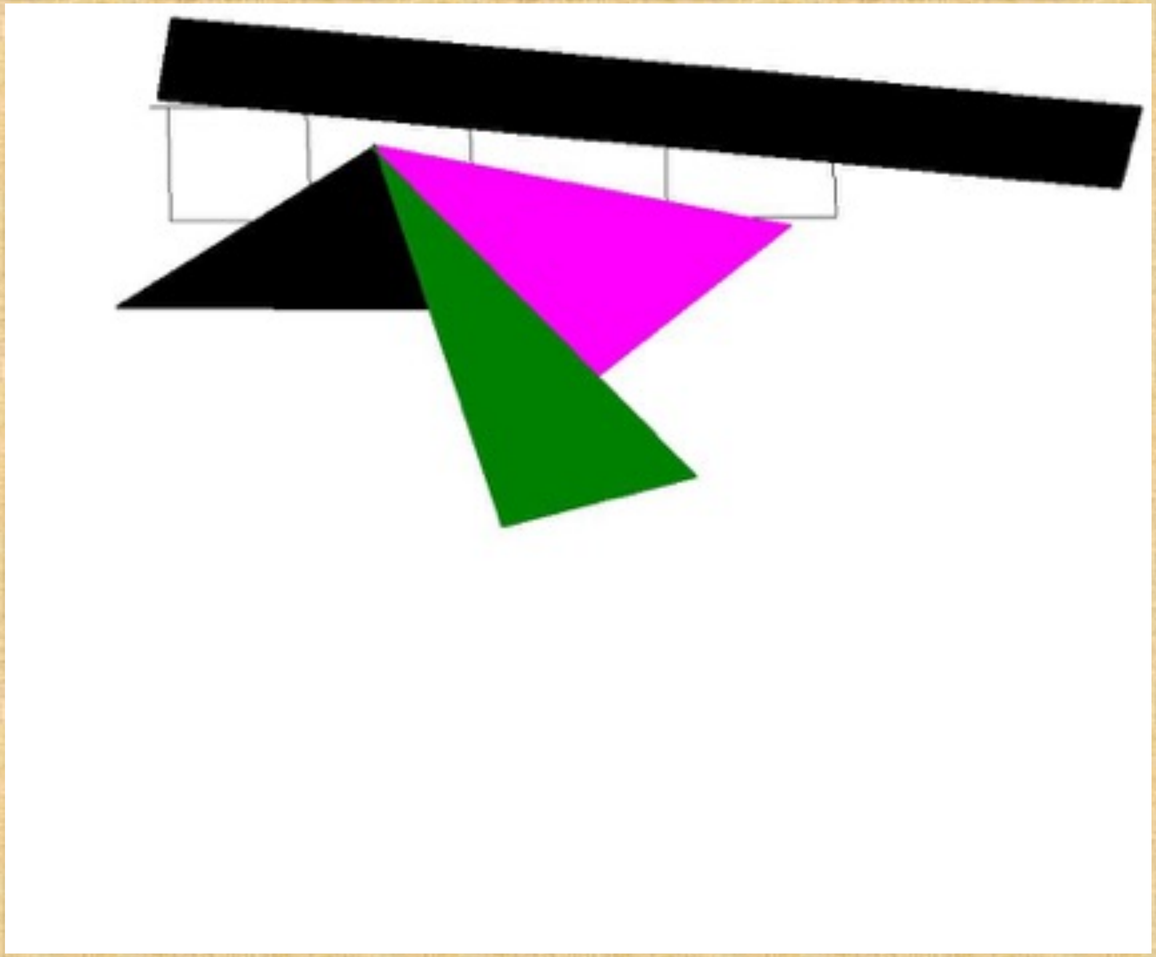
super-sampling: use 5x5
matrix, or 3x3 matrix

1, 3, 1   1,2,4,2,1

1	1	1
1	3	1
1	1	1

pixel
area weighted

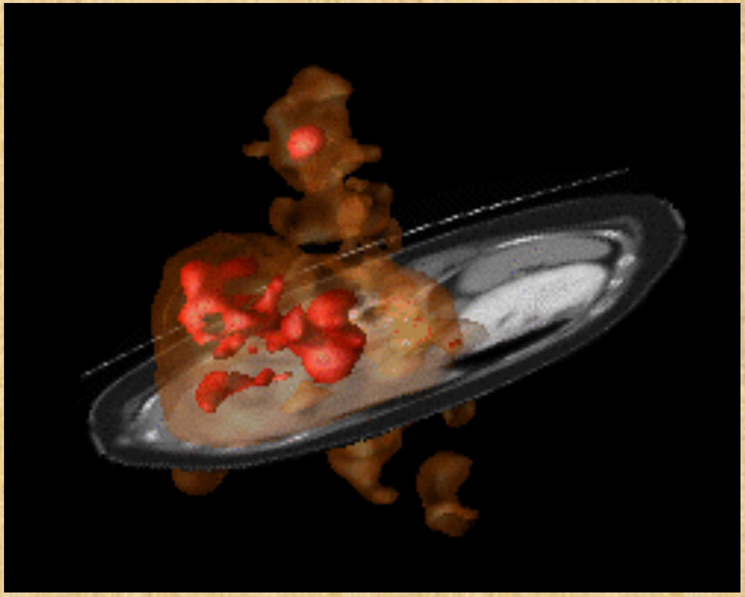
Anti-aliasing results: sharp lines and triangles



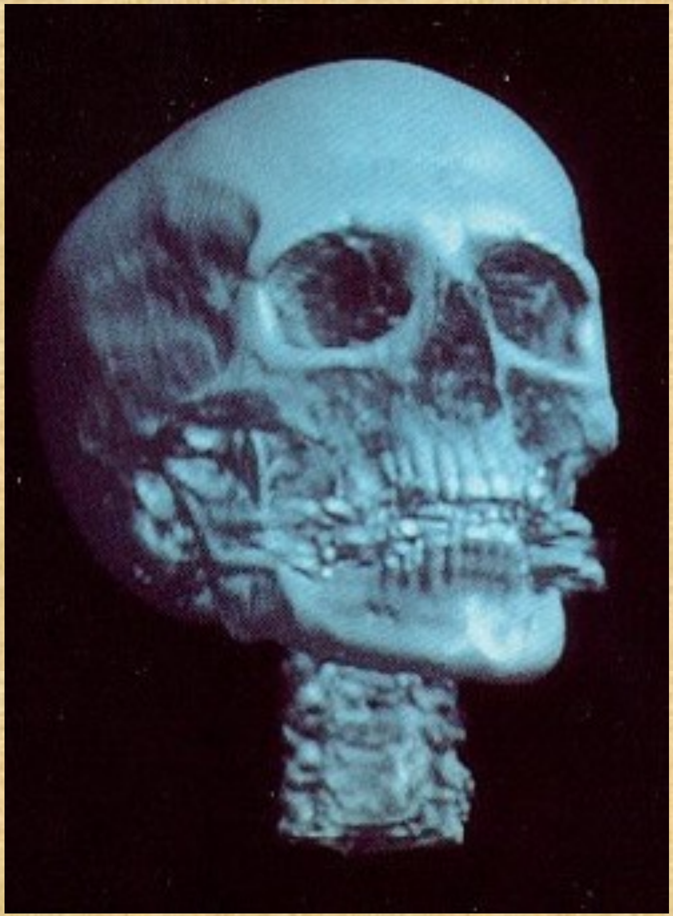
What is Volume Rendering?

- The term *volume rendering* is used to describe techniques which allow the visualization of three-dimensional data. Volume rendering is a technique for visualizing sampled functions of three spatial dimensions by computing 2-D projections of a colored semitransparent volume.

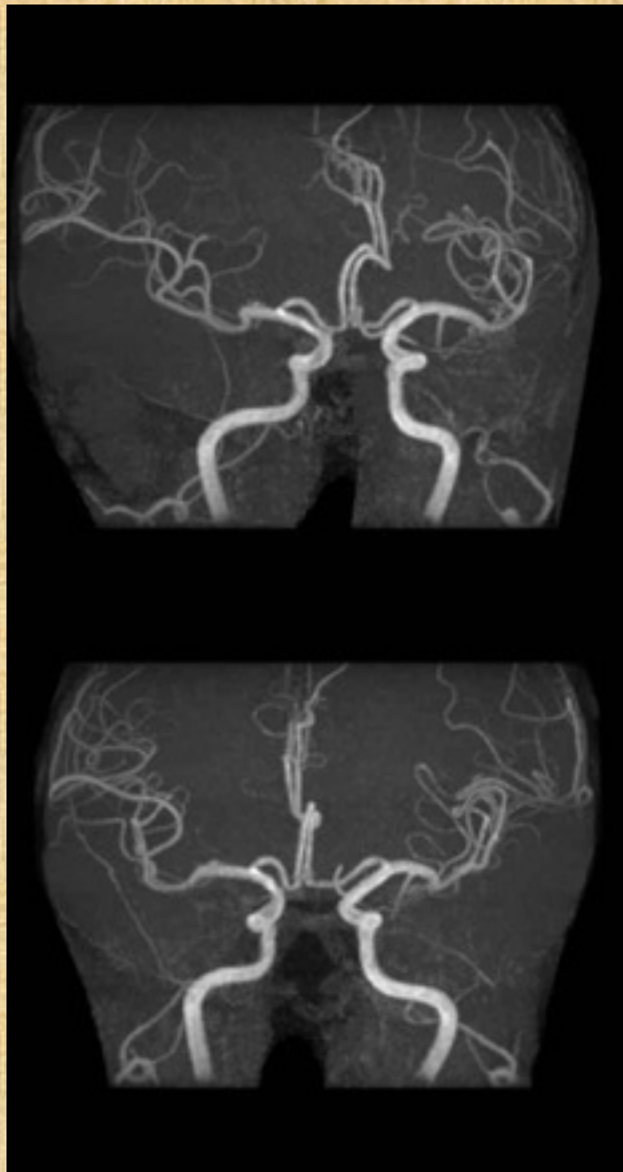
- There are many example images to be found which illustrate the capabilities of ray casting. These images were produced using IBM's Data Explorer: (left) Liver, (right) Vessels

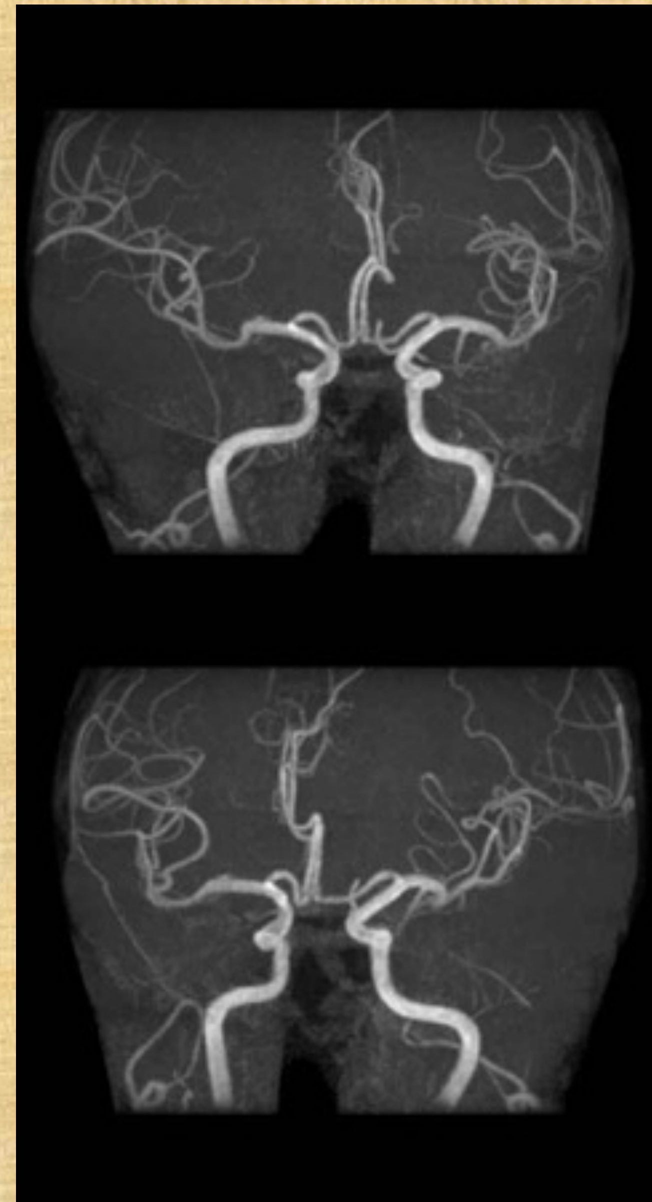


Volume Rendering: result images



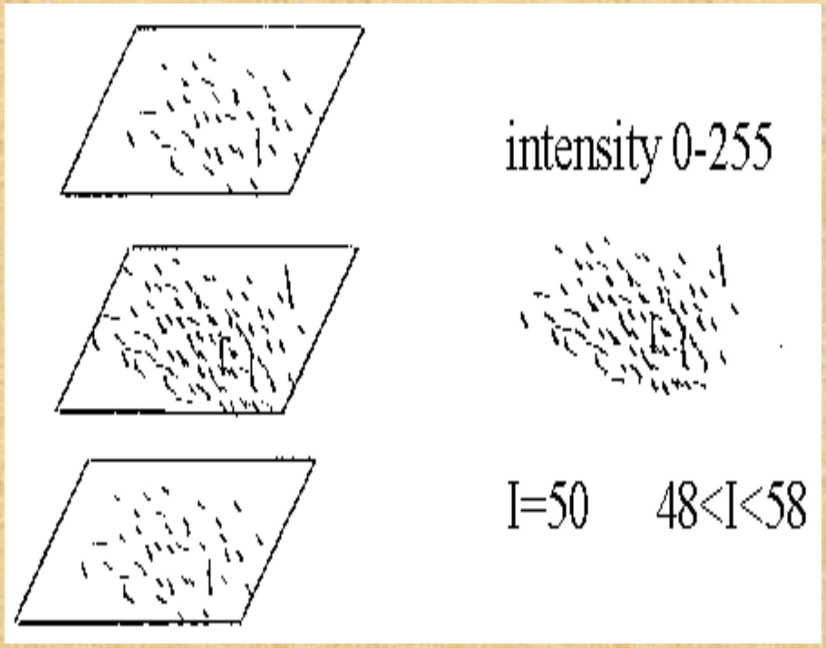
Ming’s brain vessels, MRI



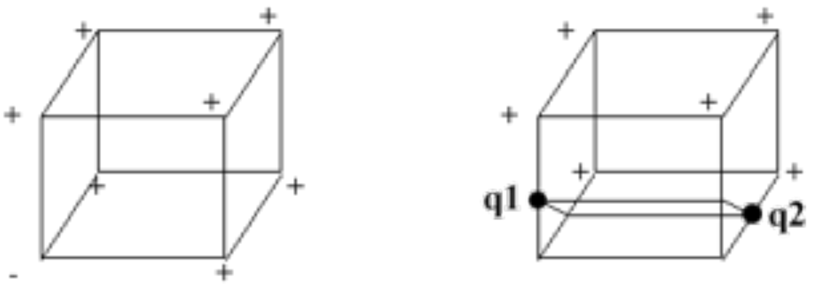


How to calculate surface normal for scalar field?

- gradient vector
- D(i, j, k) is the density at voxel(i, j, k) in slice k

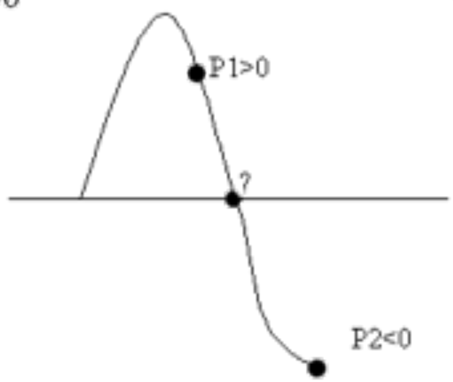


Marching cubes (squares)



+: I>50
-: I<50

2^8=256 ways reduced to 14 patterns

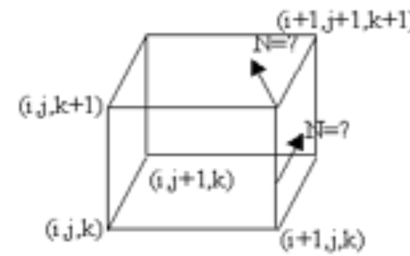


$$G(i,j,k) = \frac{D(i+1,j,,k) - D(i-1,j,k)}{\triangle x}$$

$$G(i,j,k) = \frac{D(i+1,j,,k) - D(i-1,j,k)}{\triangle y}$$

$$G(i,j,k) = \frac{D(i+1,j,,k) - D(i-1,j,k)}{\triangle z}$$

linearly intepolate

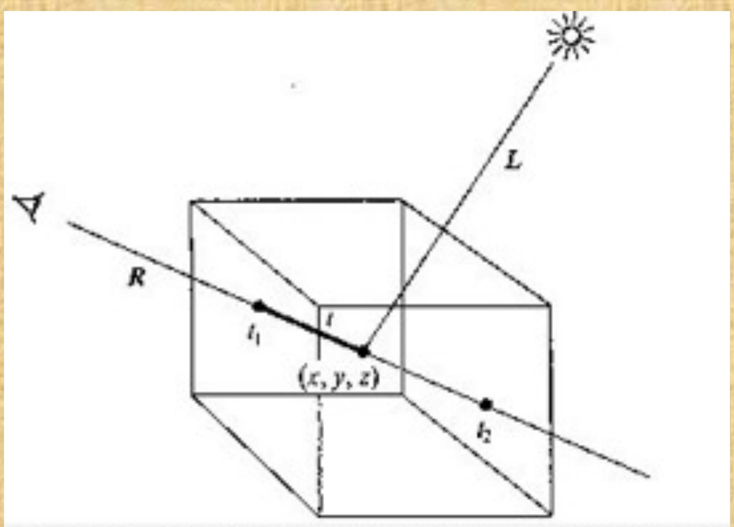


Surface normal calculation for cube corners

- Gx (i, j, k) = (D(i+1, j, k) – D(i-1, j, k))/2
- Gy (I, j, k) = (D (I, j+1, k) –D(I, j-1,k))/2
- Gz(I, j, k) = ((D(I, j, k+1) –D (I, j, k-1))/2

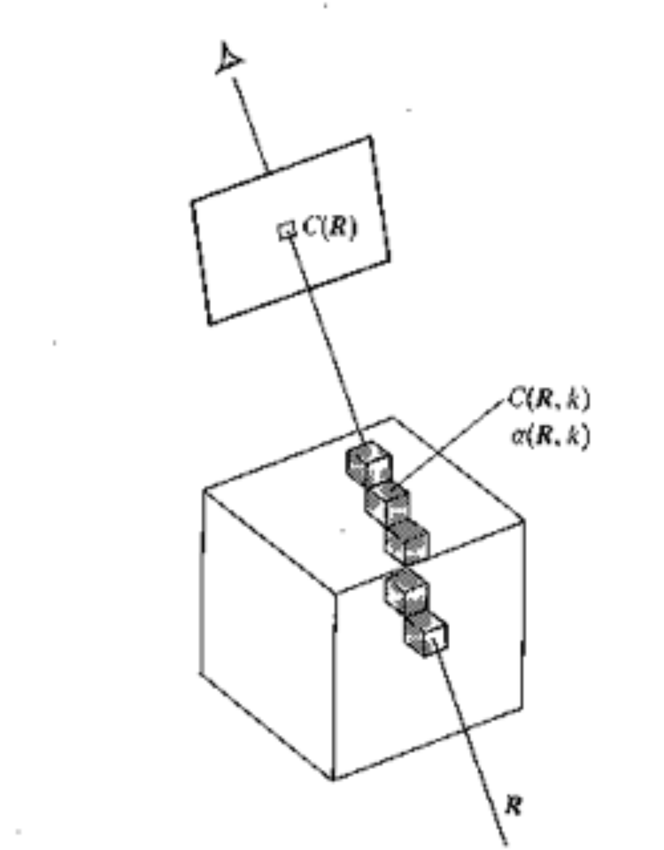
Ray casting for volume rendering

- **Theory**
  - Currently, most volume rendering that uses ray casting is based on the Blinn/Kajiya model. In this model we have a volume which has a density D(x,y,z), penetrated by a ray ***R***.



A ray R cast into a scalar function of three spatial variables

- Rays are cast from the eye to the voxel, and the values of **C(X)** and **(X)** are "combined" into single values to provide a final pixel intensity.



Transparency formula

- For a single voxel along a ray, the standard transparency formula is: $C_{out} = C_{in}(1-\alpha(x_i)) + c(x_i)\alpha(x_i)$
  where:
  – **C_{out}** is the outgoing intensity/color for voxel X along the ray
  – **C_{in}** is the incoming intensity for the voxel
- Splatting for transparent objects: back to front rendering
  – Eye ➔ Destination Voxel ➔ Source Voxel
  – $C_{d'} = (1-\alpha_s)\, C_d + \alpha_s\, C_s$

    $\alpha_{d'} = (1-\alpha_s)\, \alpha_d + \alpha_s$

    C_s : Color of source (background object color)

    $\alpha_{s:}$ Opaque index (opaque = 1.0, transparency = 0.0)

    when background $\alpha_s = 1.0$, destination $\alpha_d = 0.0$, $C_{d'} = C_s$, $\alpha_{d'} = \alpha_s$,

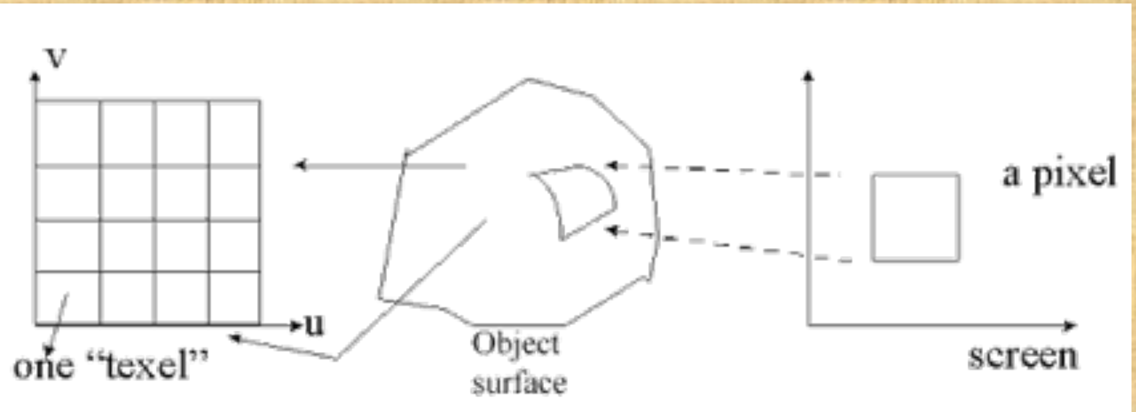
    similarly, when foreground (destination) is NOT transparent, $\alpha_d = 1.0$, $C_{d'} = C_d$ (color of itself)

3D Modeling Methods

- Creation of 3D objects
  - Revolving
  - 3D polygon
  - 3D mesh, 3D curves
  - Extrusion from 2D primitives (set elevation in Z-axis)
  - An example (new CS building construction) (step bye step demo) of AutoCAD
  - Feature that are useful
    - VPOINT, LIMITS, LINE, BREAK, Elevation, SNAP, GRID, etc
  - 3D digitizers

Texture mapping

1. What is texture?
2. How to map a texture to an object surface?



<- : direction of mapping

pixel value = sum of weighted texels within the four corners mapped from a pixel

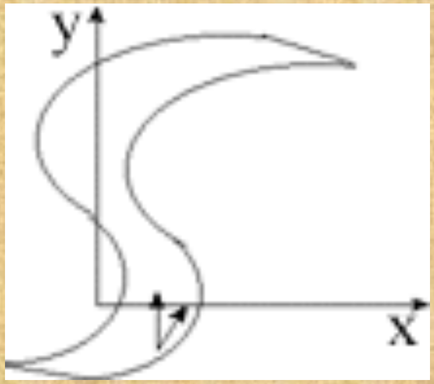
3. See pictures

Curves and surfaces

- Used in airplanes, cars, boats
- Patch (補片)

How to model a teapot?

- How to get all the triangles for a teapot?
- What kind of curved surfaces?
- How to display (scan convert) these surfaces?



- Can we show an implicit surface equation easily?

  e.g. $f(x,yx,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$
- Given (x,y), find z value
- Double roots, no real roots?

- What's the surface normal?
- Discuss ways to "define" a curved surfaces.

Curves and Surfaces

- Topics
  - Polygon meshes
  - Parametric cubic curves
  - Parametric bicubic surfaces
  - Quadric surfaces

Parametric cubic curves

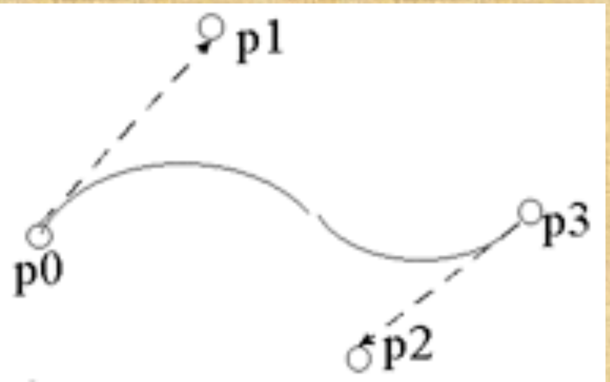
$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

- Continuity conditions
  - Geometric continuity (G_0): join together
  - Parametric continuity (C^1) (see below)
  - C^ncontinuity: d^n / $dt^n[Q(t)]$continuous

B'ezier Curve



$Q'(0) = 3\ (p1-p0)$

$Q'(1) = 3\ (p3-p2)$

Why choosing "3" ?

$Q(t) = (1-t)^3 p0 + 3t(1-t)^2 p1 + 3t^2(1-\ t)p2 + t^3 p3$

.............e.q.11.29

Bezier curve(2)

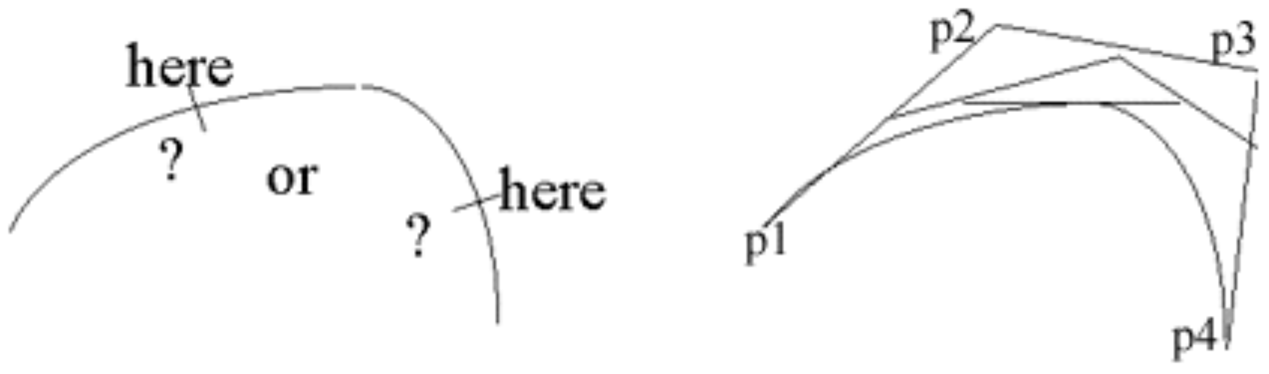
in matrix form $T*M_B*G_B$

$$[t^3, t^2, t, 1]\begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} p0 \\ p1 \\ p2 \\ p3 \end{pmatrix}$$

Note: $Q'(0) = -3(1-t)^2 p0 + 3(1-t)^2 p1 |_{t=0} = 3(p1-p0)$

if p1 - p4 is equally spaced, the curve Q(t) has constant velocity! (that's why to choose 3)

Subdividing B'ezier curves



Advantage of B'ezier curves

1. explicit control of tangent vectors
   -->interactive design
2. easy subdivision
   -->decompose into flat (line) segments

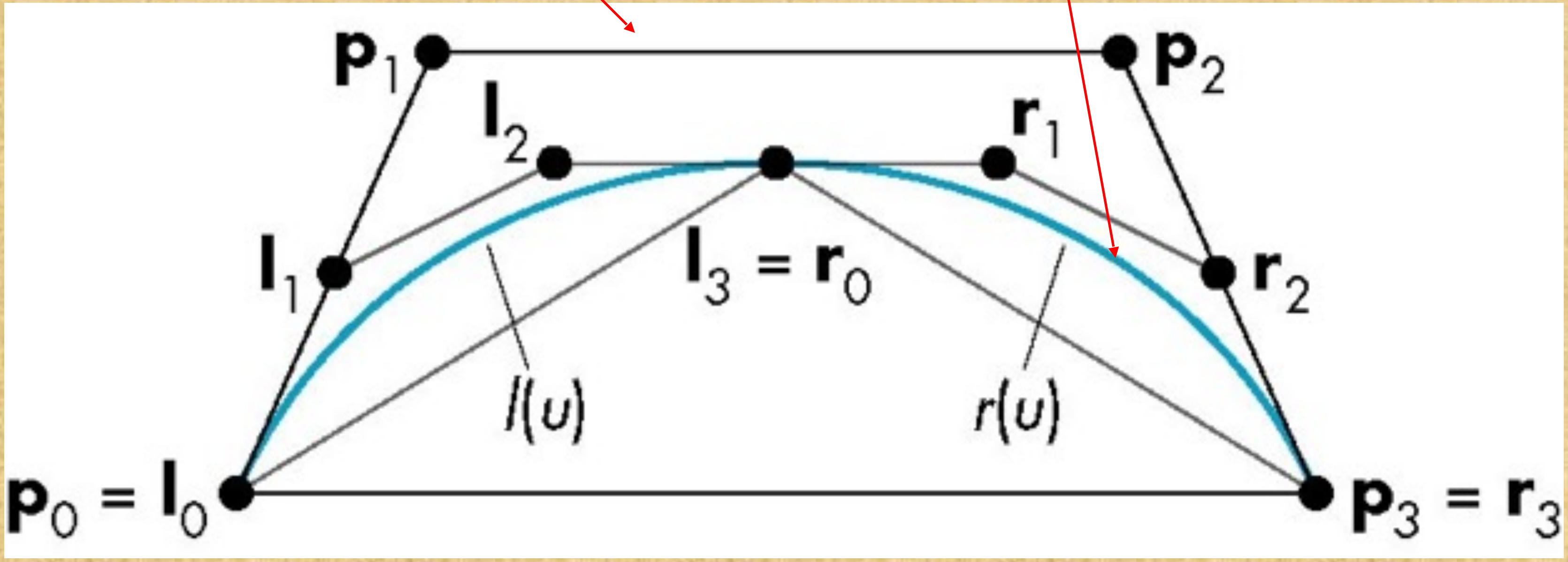
Subdividing B'ezier curves(2)

- new control points: pa, pb, pc, pe, pf,
- in addition to p1, p2, p3, p4

```
pa = 1/2(p1  + p2 )
  pb = 1/2 (p3+p4)
   pc = 1/2 (p2 + p3)
 pd = 1/2 (pa + pc)
 pe =   1/2 (pb + pc )
 pf = 1/2 (pd + pe)   new control points = p1,
pa, pd, pf .
  right half = pd, pe,  pb,  p4
```

Splitting a Cubic Bezier

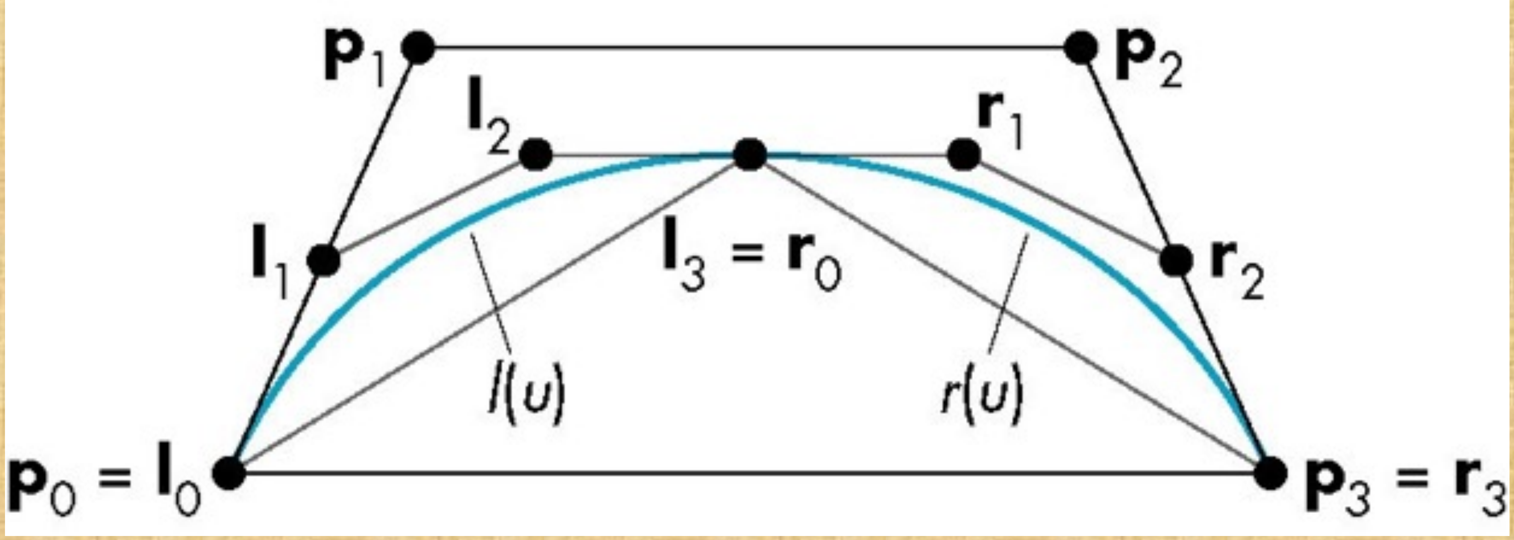
p_0, p_1 , p_2 , p_3 determine a cubic Bezier polynomial and its convex hull



Consider left half $l(u)$ and right half $r(u)$

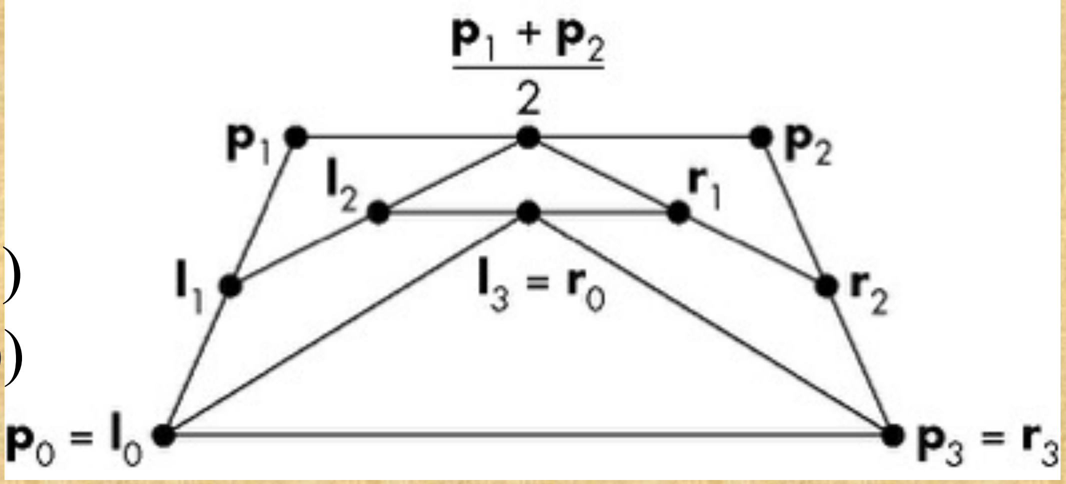
$l(u)$ and $r(u)$

Since $l(u)$ and $r(u)$ are Bezier curves, we should be able to find two sets of control points $\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ that determine them



Efficient Form

$$l_0 = p_0$$
$$r_3 = p_3$$
$$l_1 = ½(p_0 + p_1)$$
$$r_1 = ½(p_2 + p_3)$$
$$l_2 = ½(l_1 + ½(p_1 + p_2))$$
$$r_1 = ½(r_2 + ½(p_1 + p_2))$$
$$l_3 = r_0 = ½(l_2 + r_1)$$

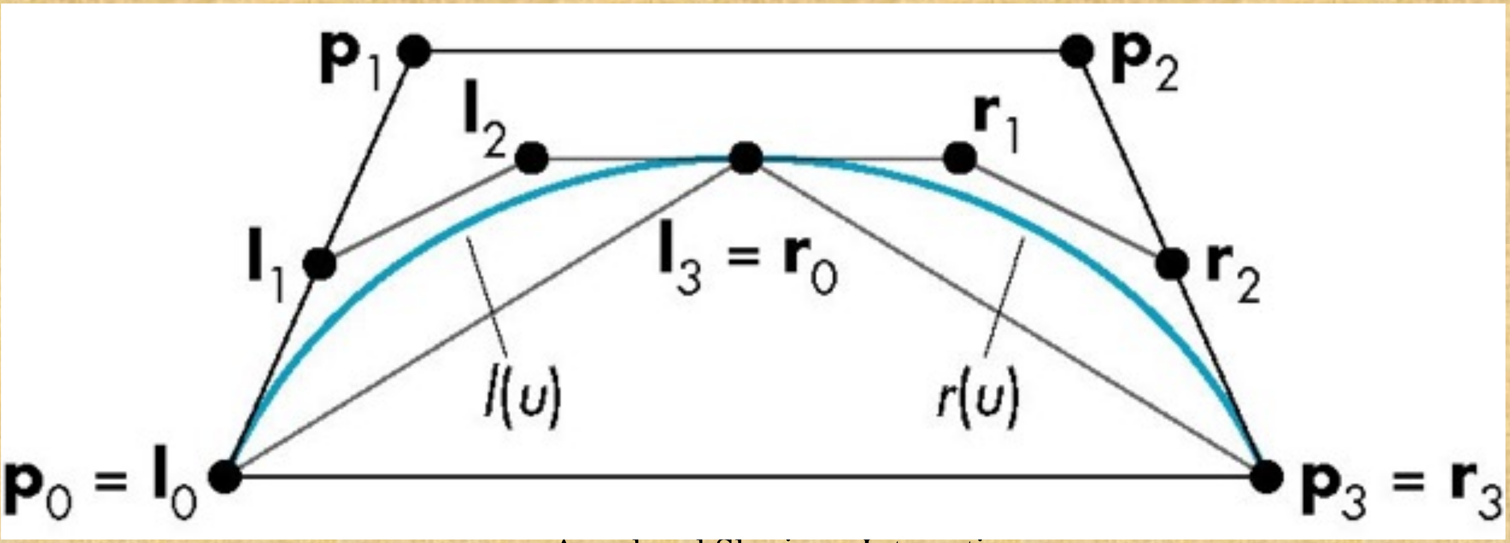


Requires only shifts and adds!

Convex Hulls

$\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ each have a convex hull that that is closer to p(u) than the convex hull of $\{p_0, p_1, p_2, p_3\}$ This is known as the *variation diminishing property*.

The polyline from l_0 to l_3 $(= r_0)$ to r_3 is an approximation to p(u). Repeating recursively we get better approximations.



Every Curve is a Bezier Curve

- We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve
- Suppose that p(u) is given as an interpolating curve with control points $\mathbf{q}$

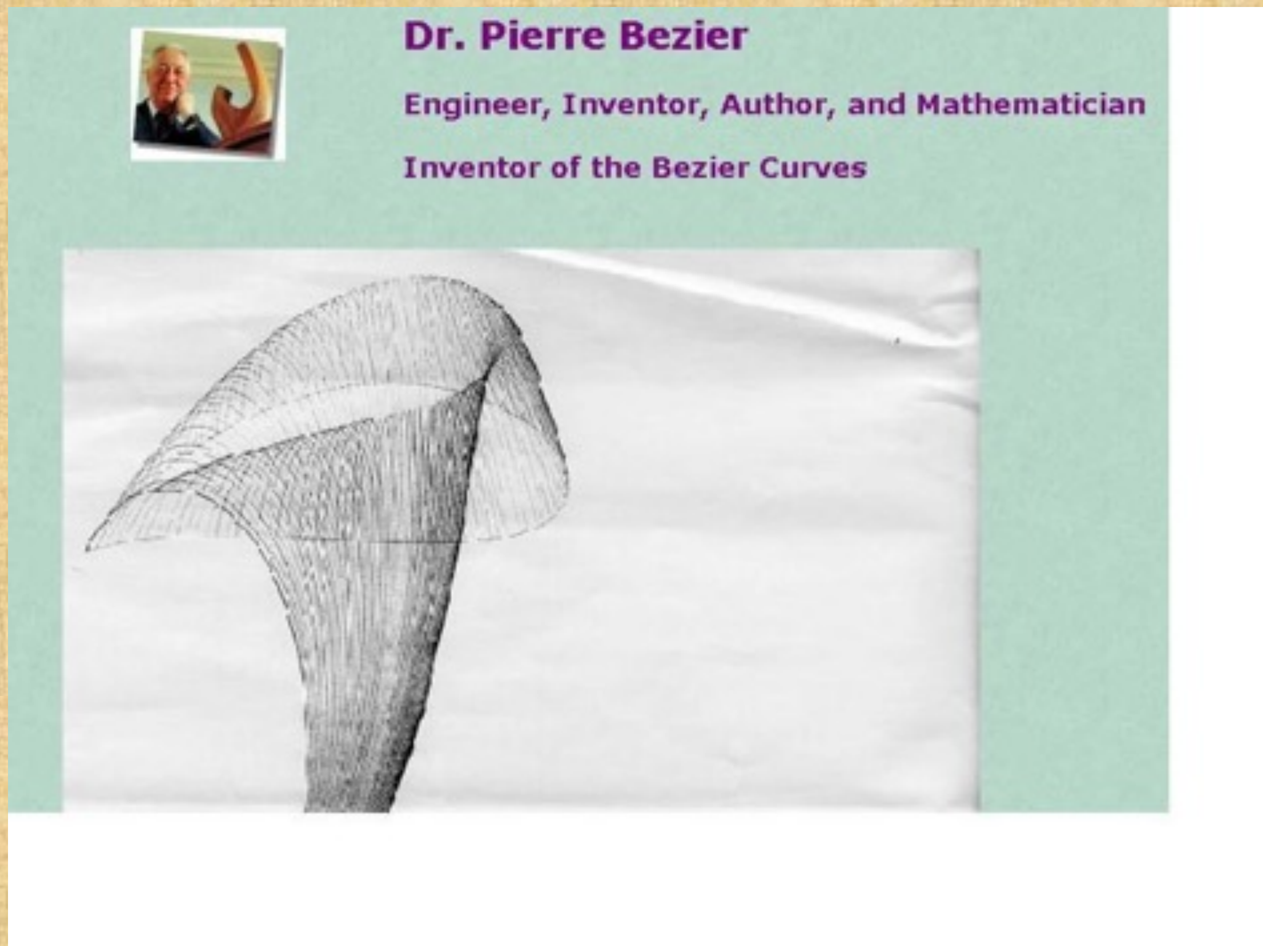
$$p(u)=\mathbf{u}^T\mathbf{M}_I\mathbf{q}$$

- There exist Bezier control points $\mathbf{p}$ such that

$$p(u)=\mathbf{u}^T\mathbf{M}_B\mathbf{p}$$

- Equating and solving, we find $\mathbf{p}=\mathbf{M}_B^{-1}\mathbf{M}_I$

Bezier


Dr. Pierre Bezier
Engineer, Inventor, Author, and Mathematician
Inventor of the Bezier Curves

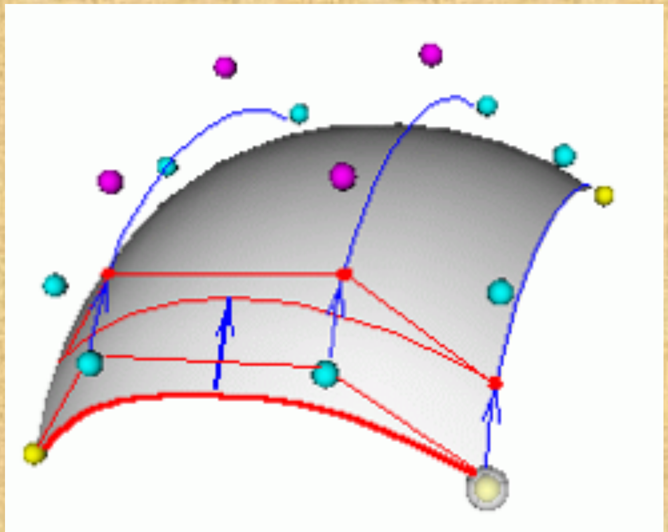
Pierre Etienne B’ezier Introduction

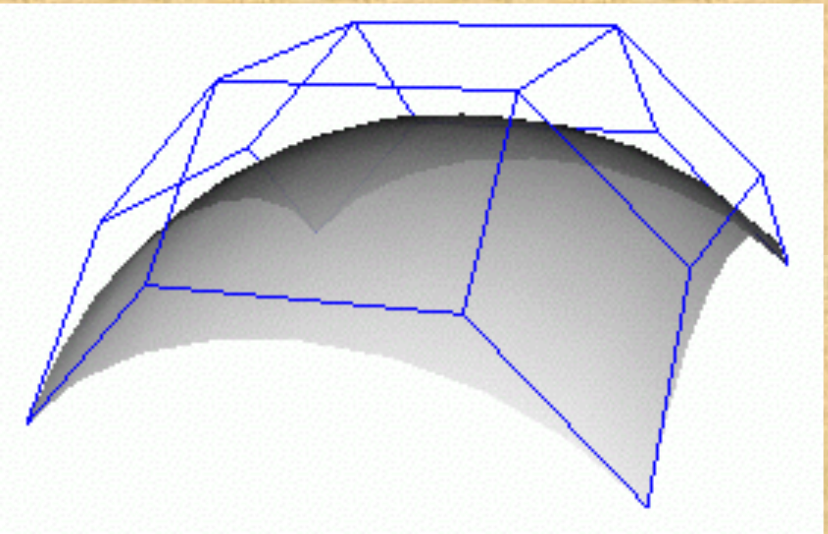
- **Pierre Etienne Bezier was born on September 1, 1910 in Paris. Son and grandson of engineers, he chose this profession too and enrolled to study mechanical engineering at the Ecole des Arts et Metiers and received his degree in 1930. In the same year he entered the Ecole Superieure d'Electricite and earnt a second degree in electrical engineering in 1931. In 1977, 46 years later, he received his DSc degree in mathematics from the University of Paris.**

In 1933, aged 23, Bezier entered Renault and worked for this company for 42 years

- **Bezier's academic career began in 1968 when he became Professor of Production Engineering at the Conservatoire National des Arts et Metiers. He held this position until 1979. He wrote four books, numerous papers and received several distinctions including the "Steven Anson Coons" of the Association for Computing Machinery and the "Doctor Honoris Causa" of the Technical University Berlin. He is an honorary member of the American Society of Mechanical Engineers and of the Societe Belge des Mecaniciens, ex-president of the Societe des Ingenieurs et Scientifiques de France, Societe des Ingenieurs Arts et Metiers, and he was one of the first Advisory Editors of "Computer-Aided Design".**

Parametric bicubic surfaces





Parametric bicubic surfaces

- First consider parametric cubic curve $Q(t) = T*M*G$
  $\therefore Q(s) = S*M*G$
- To add the second dimension, G becomes G(t)
  $G_i(t) = T*M*G_i$, where $G_i = [g_{i1}, g_{i2}, g_{i3}, g_{i4}]^T$

$$Q(s,t) = S*M*G(t) = S*M*\begin{pmatrix} G_1(t) \\ G_2(t) \\ G_3(t) \\ G_4(t) \end{pmatrix} = S*M*[G(t)]^T$$

index:(row x column)

$\therefore$Parametric bicubic suurfaces => $S*M*G*M^T*T^T$

where $S = [1, S, S^2, S^3]$

$T = [1, T, T^2, T^3]^T$

Parametric bicubic surfaces (cont.)

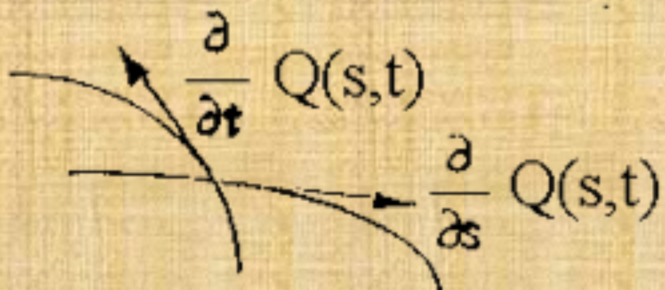
- Therefore
  - $X(s, t) = S*M*G_x*M^T*T^T$
  - $Y(s, t) = S*M*G_y*M^T*T^T$
  - $Z(s, t) = S*M*G_z*M^T*T^T$

B'ezier surfaces

- $X(s, t) = S*M_B*G_{Bx}*M_B{}^T*T^T$
- $Y(s, t) = S*M_B*G_{By}*M_B{}^T*T^T$
- $Z(s, t) = S*M_B*G_{Bz}*M_B{}^T*T^T$

- Normals to surfaces

  How to calculate?



Cross-product of tangents

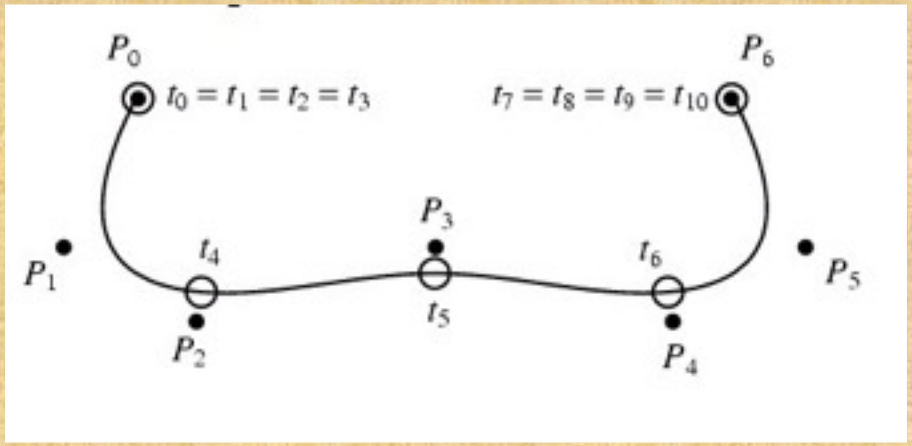
$$\frac{\partial}{\partial s} Q(s,t) \times \frac{\partial}{\partial t} Q(s,t)$$

B'ezier patches display

- How to display B'ezier patches efficiently?
  - Brute force iterative evaluation is very expensive
    Why? elaborate
  - Subdivide into smaller polygons
    need flatness test to stop subdivision
  - Adaptive subdivision is more practical
- How to avoid it?

Splines

- A B-spline is a generalization of the [Bézier curve](). Let a vector known as the [knot vector]() be defined



$$T = \{t_0, t_1, \ldots, t_m\} \qquad (1)$$

where ***T*** is a nondecreasing [sequence]() with $ti \in [0, 1]$ and define control points $\mathbf{P}_0, \ldots, \mathbf{P}_n$. Define the degree as

$$p \equiv m - n - 1. \qquad (2)$$

The "knots" $t_{p+1}, \ldots, t_{m-p-1}$ are called [internal knots]().

Splines

- Define the basis functions as

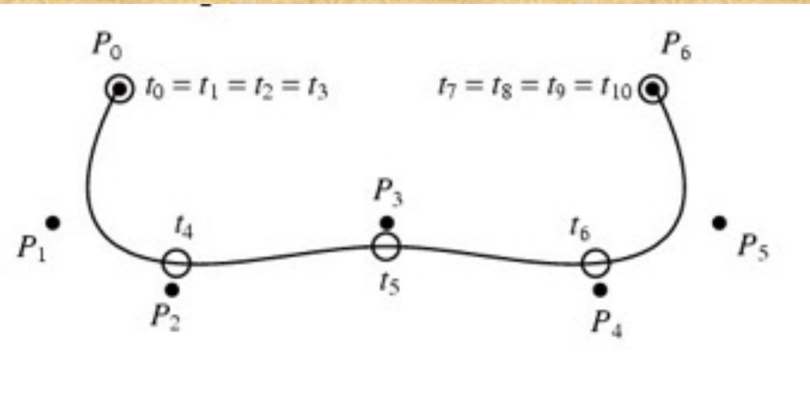
$$N_{i,0}(t) = \begin{cases} 1 & if\ t_i \le t < t_{i+1}\ and\ t_i \le t_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t)$$

- Then the curve defined by

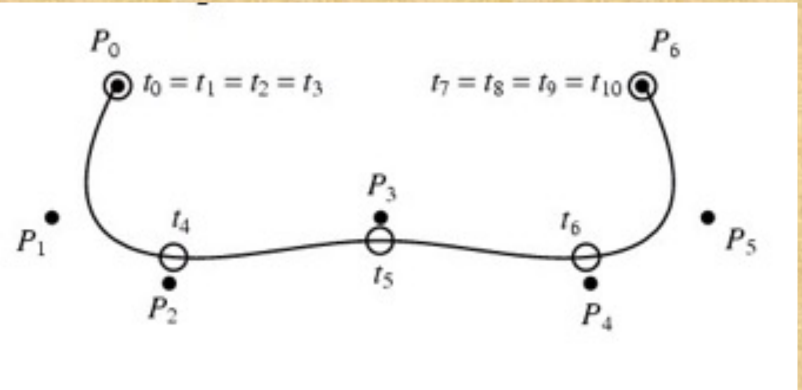
$$C(t) = \sum_{i=0}^{n} P_i N_{i,p}(t)$$

is a B-spline.



Cubic B-Spline Curve

- Cubic B-Spline Curve, C^2 continuous
- $P(u) = u^T M p$, where P is control points $[p^{i-2}, p^{i-1}, p^{i}, p^{i+1}]^T$
- At first define it to be C^1 continuous, set up boundary conditions, and we can get



$$Ms = \left(\frac{1}{6}\right)\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

$b(u) = M^T u = (1/6)\,[\,(1-u)^3,\ 4-6u^2+3u^3,\ 1+3u+3u^2-3u^3,\ u^3]^T$

$P(u) = u^T Ms\, p$ (p is the control point vector of Spline)

$P(u) = u^T Mb\, q$ (q is the control point vector of Bezier)

Therefore $q = Mb^{-1} Ms\, p$ (conversion is done)

Every Curve is a Bezier Curve

- We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve
- Suppose that $p(u)$ is given as an interpolating curve with control points $\mathbf{q}$

$$p(u)=\mathbf{u}^T\mathbf{M}_I\mathbf{q}$$

- There exist Bezier control points $\mathbf{p}$ such that

$$p(u)=\mathbf{u}^T\mathbf{M}_B\mathbf{p}$$

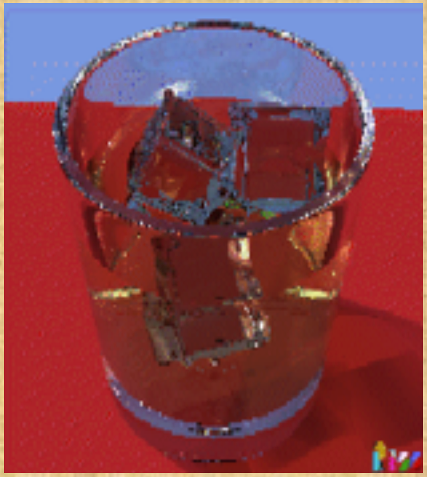
- Equating and solving, we find $\mathbf{p}=\mathbf{M}_B^{-1}\mathbf{M}_I$

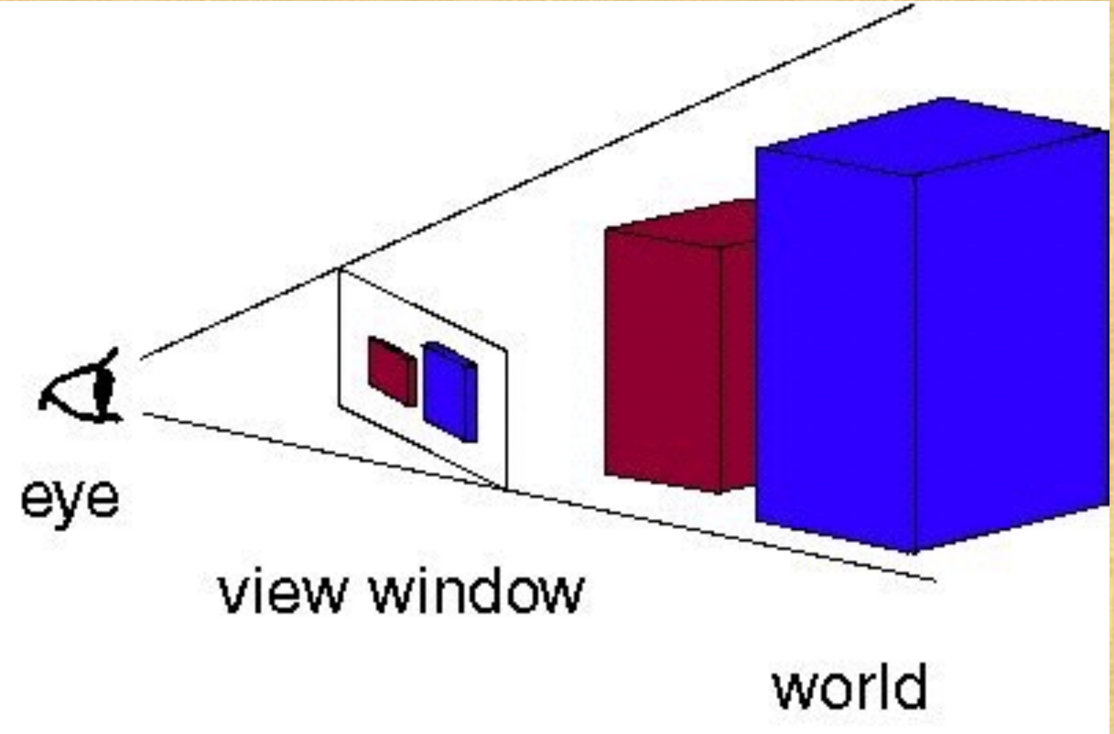
Curve DEMO

- Use web page69_1, 69_2, ….., 69_7

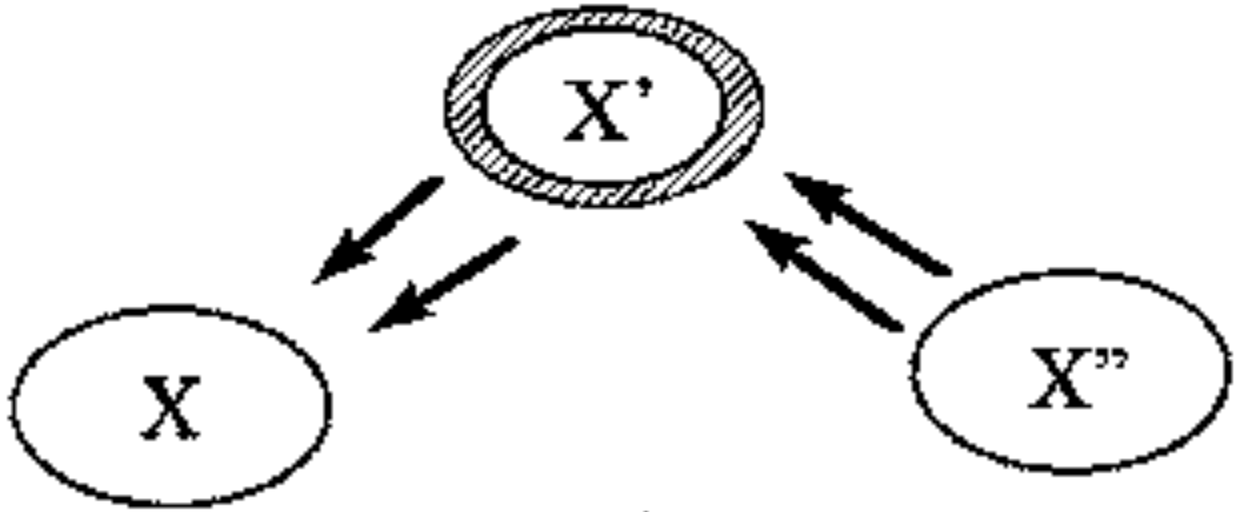
Ray tracing: Turner Whitted

- Key to success, from light to eye or from eye to screen?

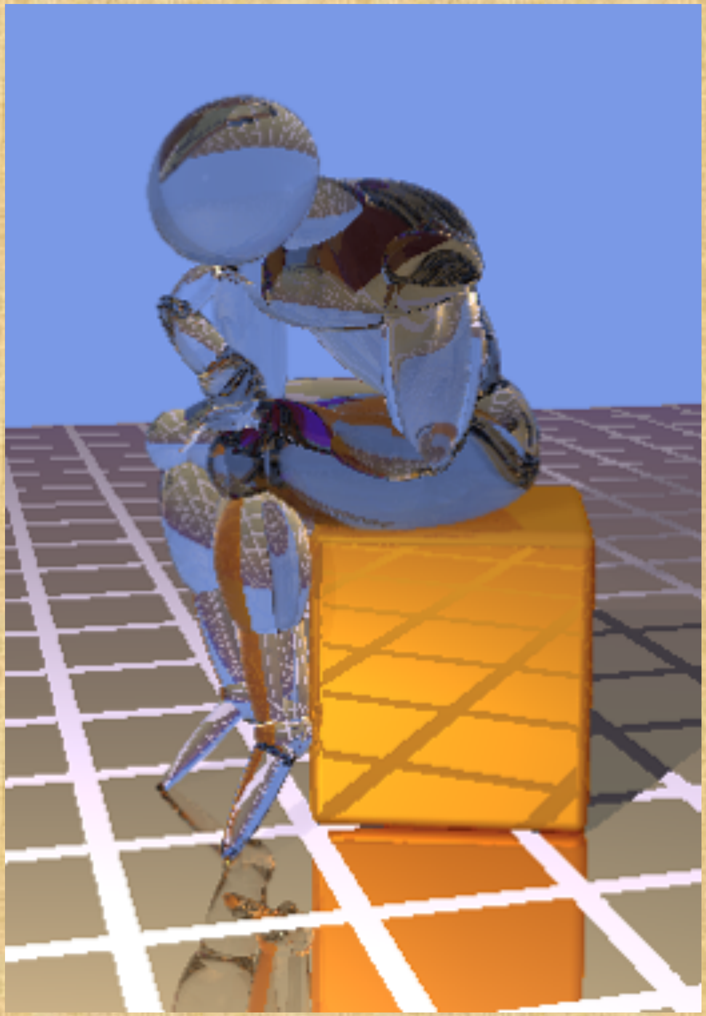


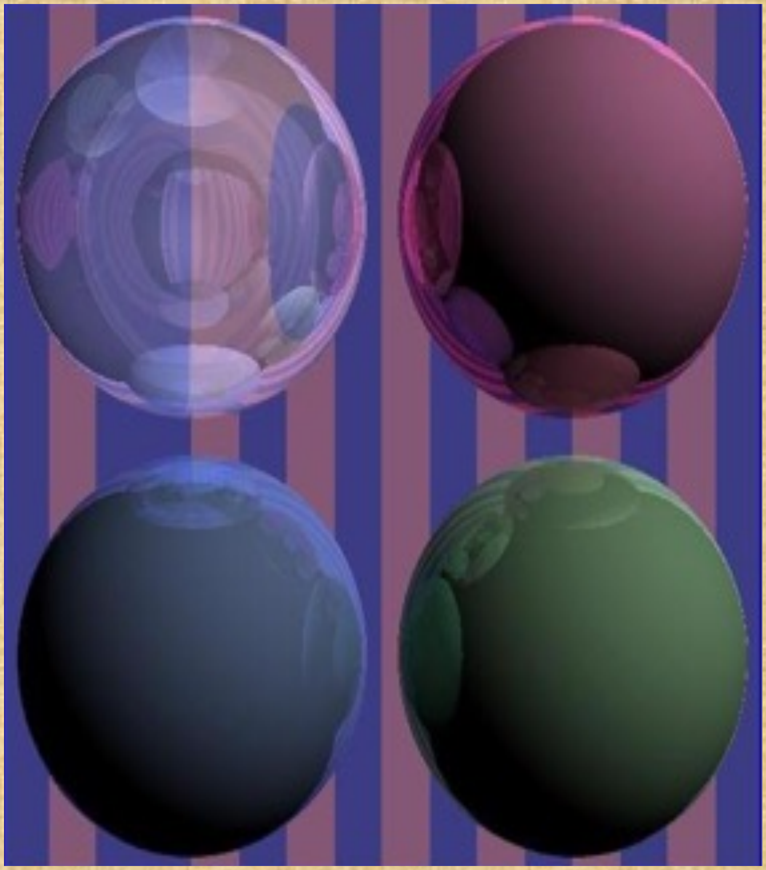


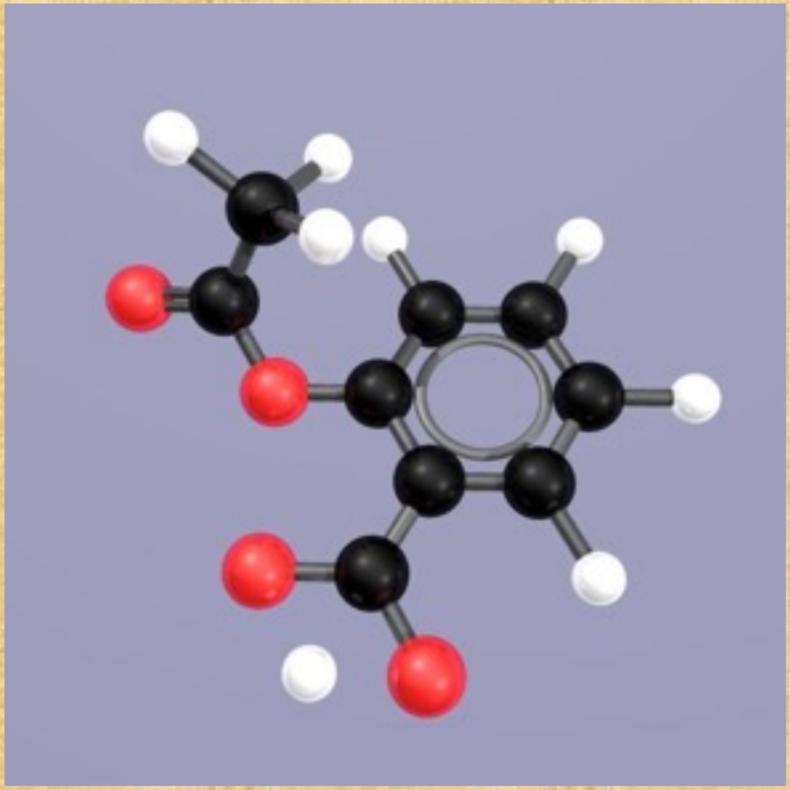
The Rendering Equation: Jim Kajiya, 1986



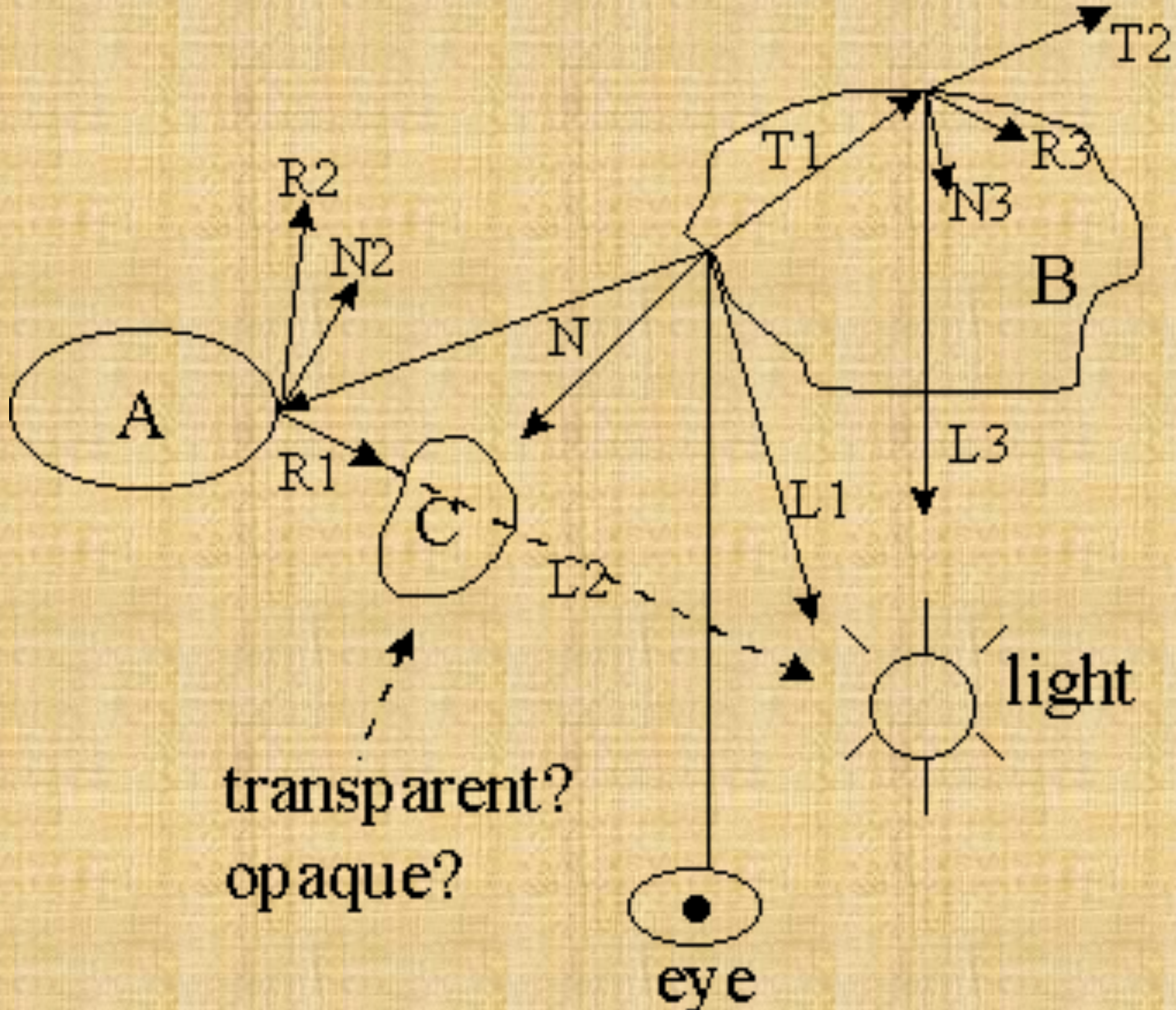
$$I(x,x')=g(x,x')\left[e(x,x')+\int_S \rho(x,x',x'')I(x',x'')dx''\right]$$







Ray tracing(1)



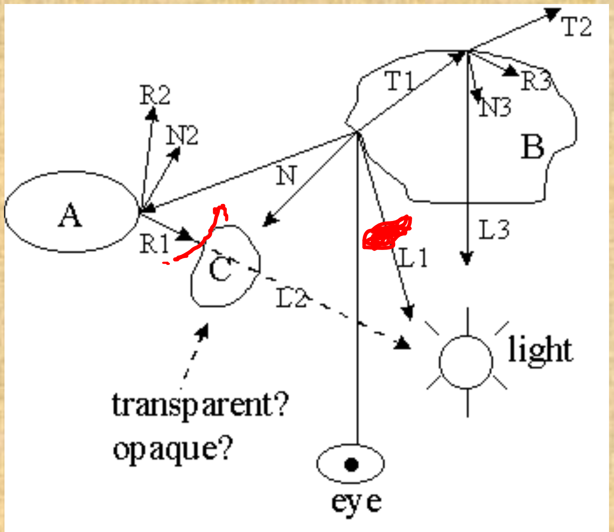
Simple recursive ray tracing

L_i: shadow ray

R_i: reflected ray

N_i: normal

T_i: transmitted ray



whether

1. $L_1=R_1+T_1$? or
2. $f^1(L_1)=f(R_1)+f(T_1)$? or
3. Color=$f(L_1,R_1,T_1$

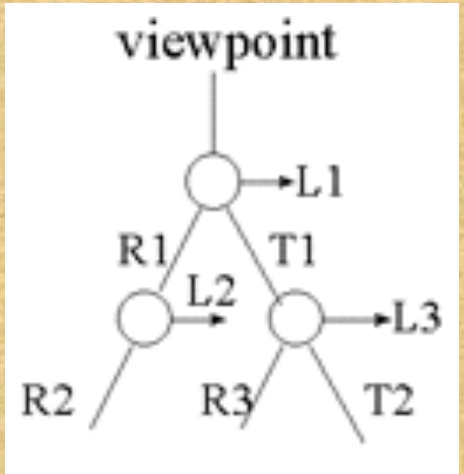
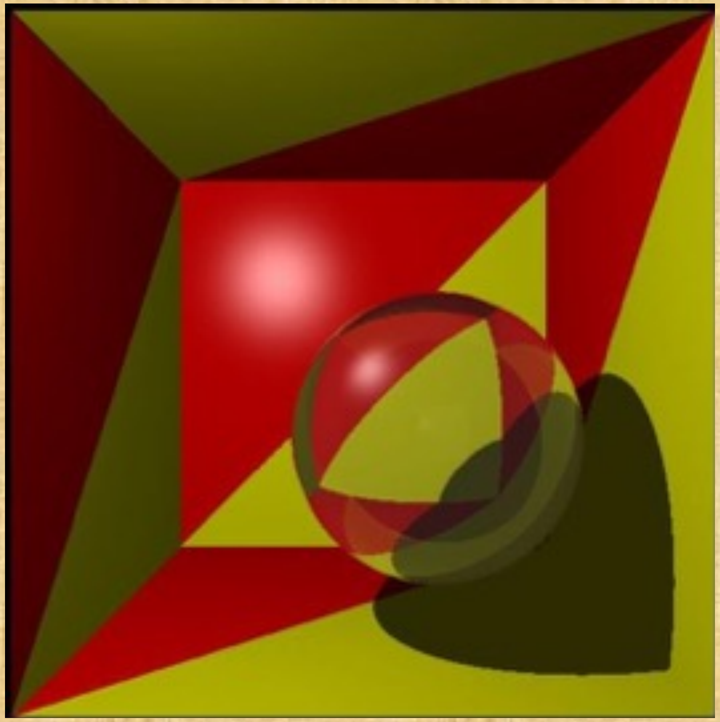
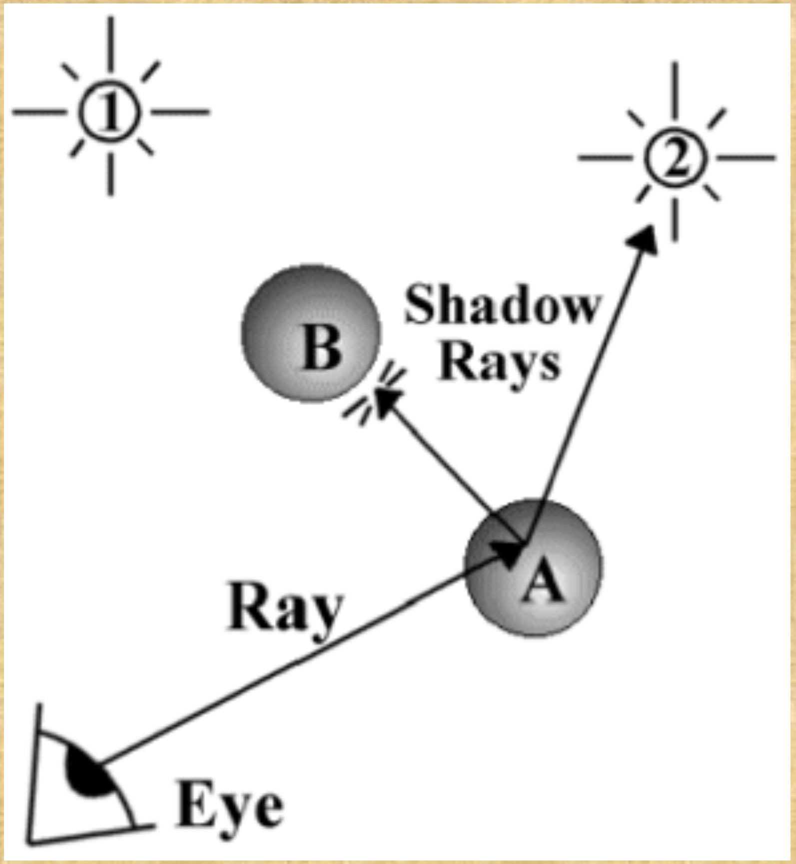


fig 16.55 Aray tree

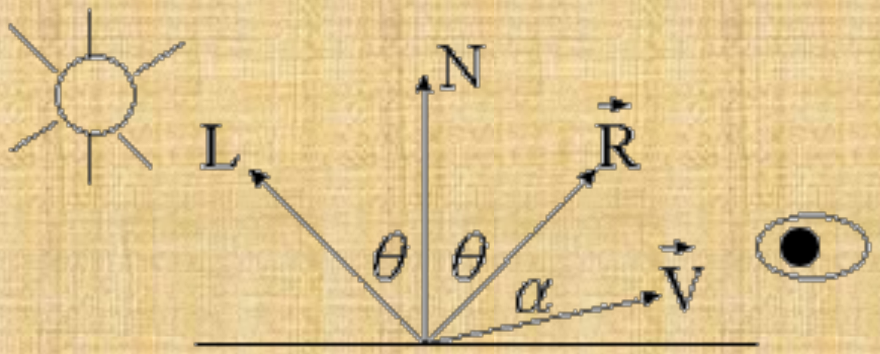
Shadow in ray tracing





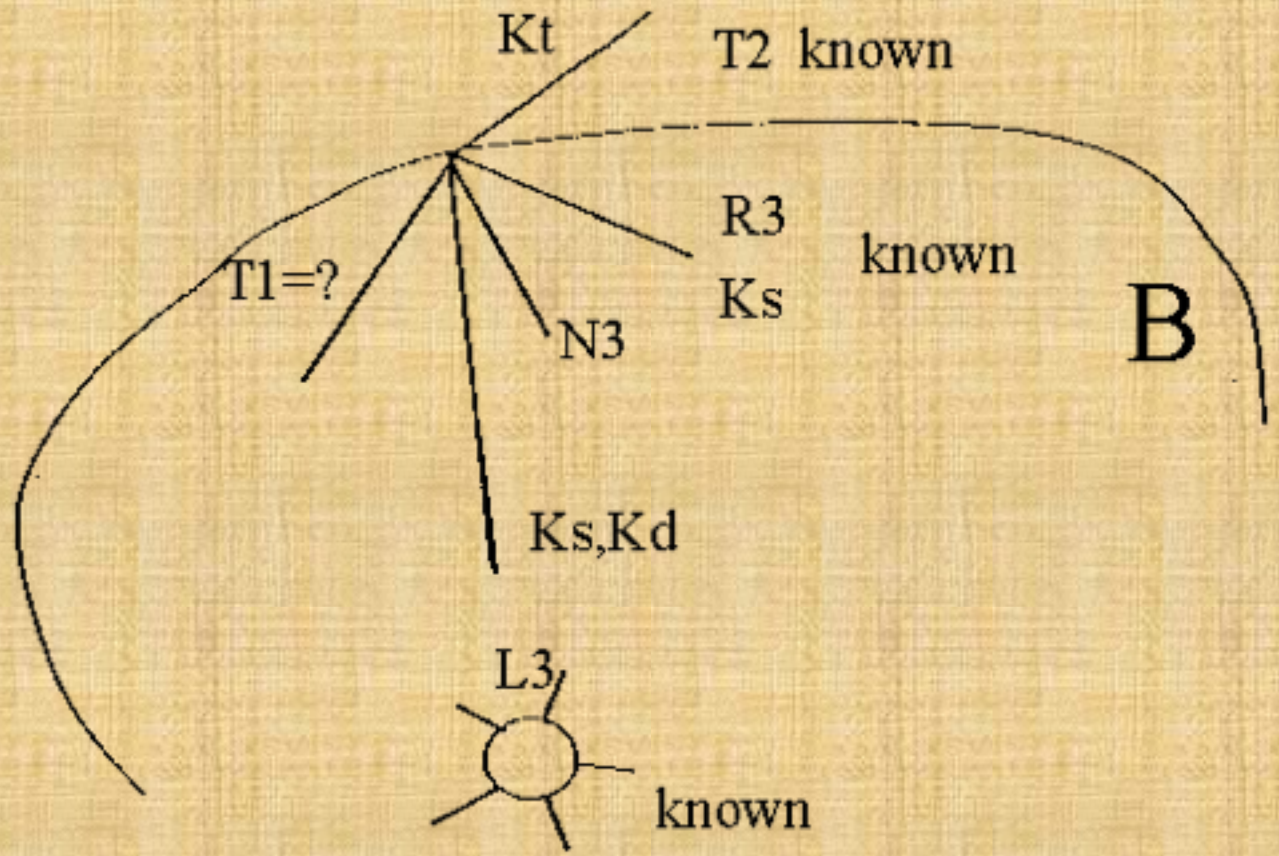
Faster: ray tracing

- halfway vector



$$\vec{H} = \frac{\vec{L}+\vec{V}}{|\vec{L}+\vec{V}|}$$

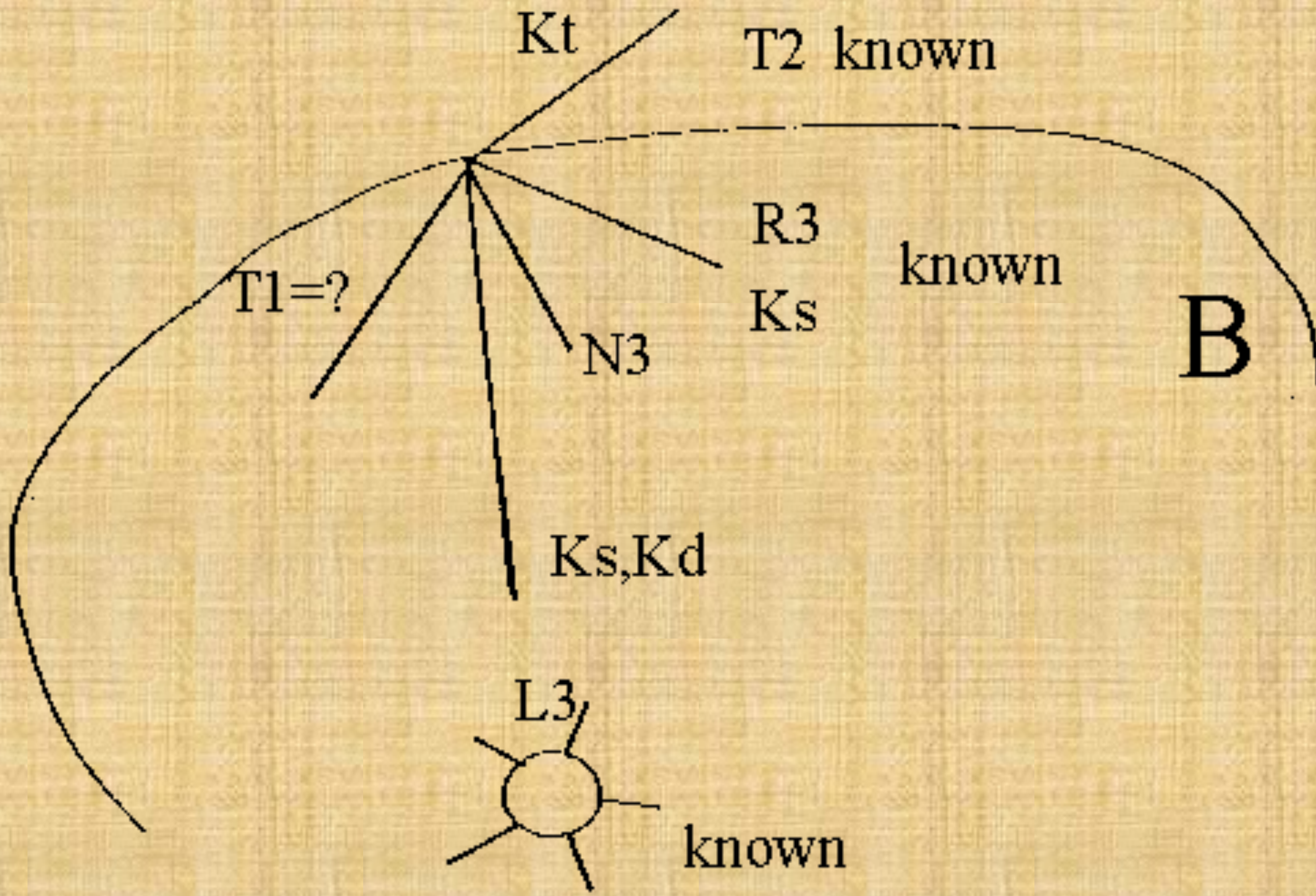
From known data to unknown



Ray Tracing Algorithm

```
Trace(ray)
    For each object in scene
        Intersect(ray, object)
    If no intersections
        return BackgroundColor
    For each light
        For each object in scene
                Intersect(ShadowRay, object)
                Accumulate local illumination
                Trace(ReflectionRay)
                Trace(TransmissionRay)
                Accumulate global illumination
```

Ray Tracing Algorithm



Code example: A simple ray tracer

- Author: Turner Whitted
  - famous for his implementation of recursive ray tracer.
- Simplified version:
  - input: quadric surfaces only
    i.e. $f(x,y,z)=ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$
  - Shading calculation: as simple as possible
- Surface normal

  $[df/dx, df/dy, df/dz]$

  $= [\ 2ax+2dy+2fz+2g,\ 2by+2dx+2ez+2h,\ 2cz+2ey+2fx+2j\ ]$

Sample program

Color trace_ray(**Ray** original_ray)
{
    Color point_color, reflect_color, refract_color
    Object obj

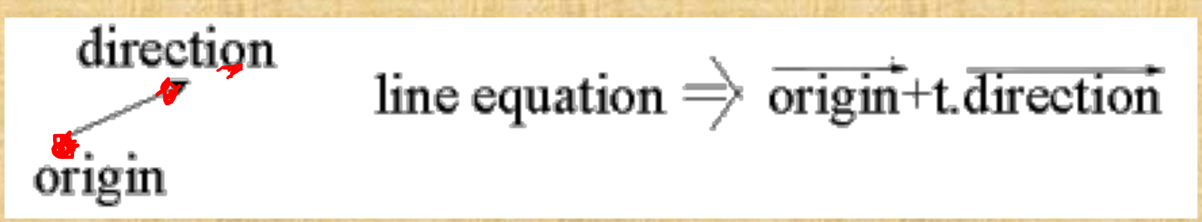
    obj = **get_first_intersection**(original_ray)
    point_color = **get_point_color**(obj)

    if (object is reflective)
        reflect_color = **trace_ray**(**get_reflected_ray**(original_ray, obj))
    if (object is refractive)
        refract_color = **trace_ray**(**get_refracted_ray**(original_ray, obj))

    return (**combine_colors**(point_color, reflect_color, refract_color))
}

Code example: A simple ray tracer

- The simple ray tracer is complete and free to copy
  [need modification to be term project]
- Input surface properties
  - r, g, b, relative_index_of_refraction, reflection_coef, transmission_coef, object_type
  - number_of_objects, number_of_surfaces, number_of_properties
- How to calculate the intersection of a ray and a quadric surface?



Ray to quadric surface intersection

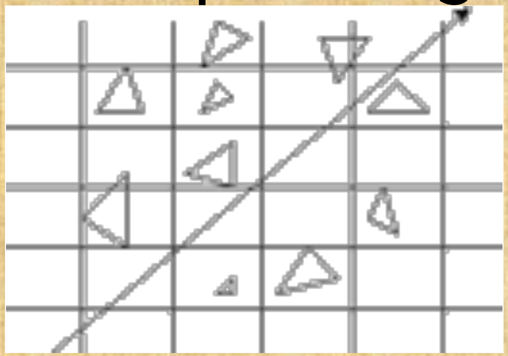
- intersection calculation:
  - let direction=(D_x, D_y, D_z), origin=(O_x, O_y, O_z)
    line ==> $(x,y,z)=(O_x, O_y, O_z) + t*(D_x, D_y, D_z)$ (1)
  - quadric surface
    $f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$ (2)
  - replace (x,y,z) in (2) by (1),
    $acoef*t^2 + bcoef*t + ccoef = 0$, solve for t

$$t=(-bcoef \pm \sqrt{bcoef^2-4acoef*ccoef})/(2*acoef)$$

    - for example:
      $acoef = a*D_x^2 + b*D_x*D_y + c*D_x*D_z + e*D_y^2 + f*D_y*D_z + h*D_z^2$

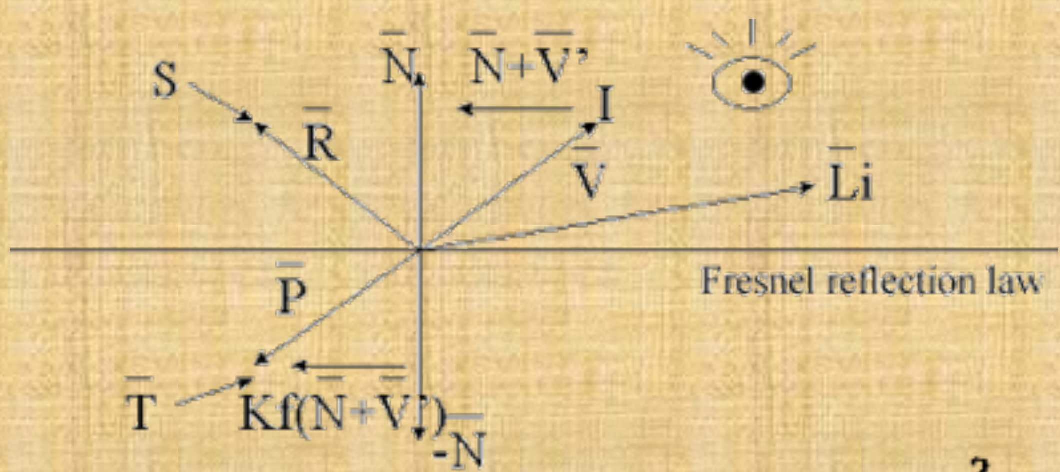
Special notice:

1. Avoid to intersect a surface twice within a tiny triange
   - e.g. t1=100, t2=100.001
   - This may happen because of numeric percision
2. If a ray doesn't hit anything, give it a non-offensive background color, (20,92,192).
   - This is the sky color(assume it is day time, of course).
   - Otherwise, choose twilight or dark sky color.
3. How to modify this program to accept triangles? Grid methods?
   - Each grid center contains a pointer to the list of triangles which are(partly) contained in the grid.



Special notice:

4. Shading model



$$\bar{V}' = \frac{\bar{V}}{|\bar{V} \cdot \bar{N}|}$$

$$\bar{R} = \bar{V}' + 2\bar{N}$$

$$\bar{P} = k_f(\bar{N} + \bar{V}') - \bar{N}$$

where $K_f = (K_n^2 |\bar{V}'|^2 - |\bar{V}' + \bar{N}|^2)^{-1/2}$

and K_n = the index of refraction

5. $$I = I_a + \sum_{j=1}^{n} \left(\bar{N} * \bar{L}_j\right) + K_S * S + K_T * T$$

- Ultimately, this yields the following pseudocode:

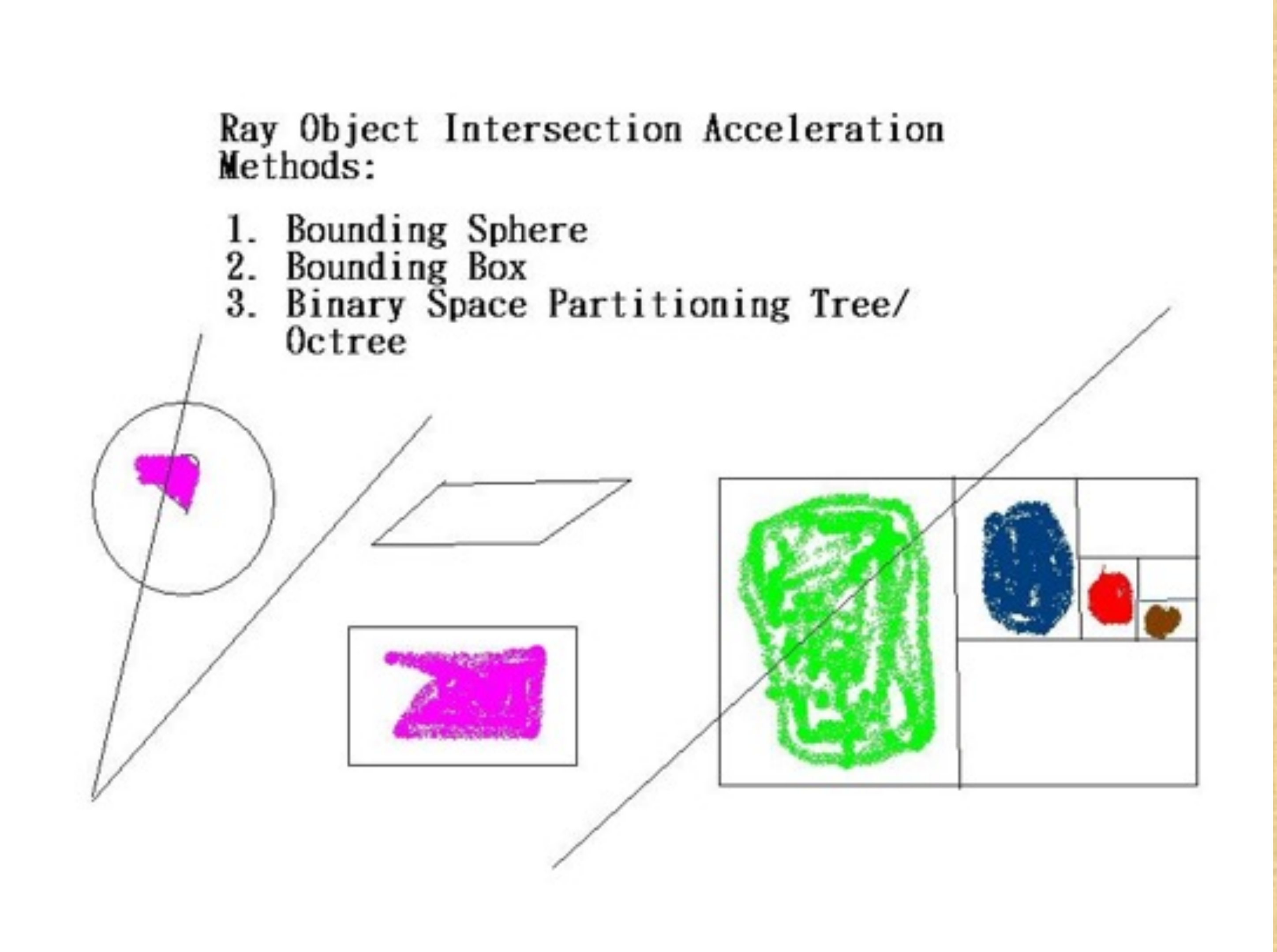
```
procedure TraceRay_1(u) begin
  Ĉ(u) := 0;
  α(u) := 0;
  x_1 := First(u);
  x_2 := Last(u);
  U_1 := ⌈Image(x_1)⌉;
  U_2 := ⌊Image(x_2)⌋;
  {Loop through all samples falling within data}
  for U := U_1 to U_2 do begin
    x := Object(U);
    {If sample opacity > 0,}
    {then resample color and composite into ray}
    α(U) := Sample(α, x);
    if α(U) > 0 then begin
      Ĉ(U) := Sample(Ĉ, x);
      Ĉ(u) := Ĉ(u) + Ĉ(U)(1 − α(u));
      α(u) := α(u) + α(U)(1 − α(u));
    end
  end
end TraceRay_1.
```

- For more info, please see my document Ray_Tracing.bw

What is still missing in ray-traced images?

- Diffuse to diffuse reflection?

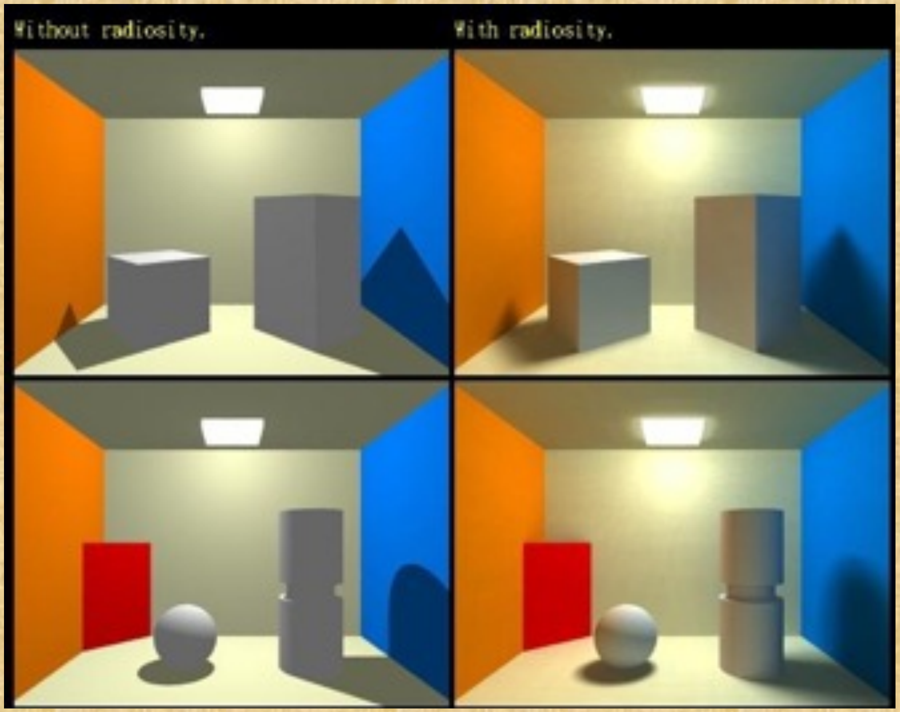
Ray-object intersection acceleration



Radiosity (熱輻射法)

Donald Greenberg and Tomoyuki Nishita

See my directory: Radiosity (page 89-96)




The Boss

Rendering Equation: Another version

Consider light at a point **p** arriving from **p**'

$$i(\mathbf{p}, \mathbf{p}') = \upsilon(\mathbf{p}, \mathbf{p}')(\varepsilon(\mathbf{p},\mathbf{p}') + \int \rho(\mathbf{p}, \mathbf{p}', \mathbf{p}'')i(\mathbf{p}', \mathbf{p}'')d\mathbf{p}''$$

emission from **p**' to **p**

occlusion = 0 or $1/d^2$

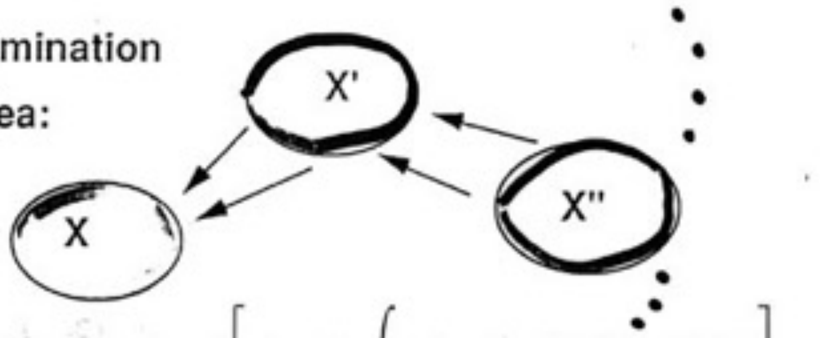
light reflected at **p**' from all points **p**'' towards **p**

Radiosity

- Consider objects to be broken up into flat patches (which may correspond to the polygons in the model)
- Assume that patches are perfectly diffuse reflectors
- Radiosity = flux = energy/unit area/ unit time leaving patch

Radiosity

global illumination idea:



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

$$B_i A_i = E_i A_i + \mathbf{P}_i \sum_j B_j F_{ji} A_j$$

$$B_i = E_i + \mathbf{P}_i \sum_{j=1}^{n} B_j F_{j,i} \frac{A_j}{A_i}$$

Definitions

- where
  Bi: B are the radiosity of patches i and j
  Ei: the rate at which light is emitted from patch i
  Pi: patch i's reflectivity
  Fj-i:formfactor (configuration factor) , which specifies the fraction of energy leaving the patch j that arrives at patch i.
  Ai, Aj areas of patch i and j.

互惠

互惠

Reciprocity in radiosity (互惠)

(2) $$A_i F_{i,j} = A_j F_{j,i}$$

(3)Simplified Eq from (1) $$B_i = E_i + \mathrm{P}_i \sum_{j=1}^{n} B_j F_{i-j}$$

Radiosity Equation

energy balance

$$b_i a_i = e_i a_i + \rho_i \sum f_{ji} b_j a_j$$

reciprocity

$$f_{ij} a_i = f_{ji} a_j$$

radiosity equation

$$b_i = e_i + \rho_i \sum f_{ij} b_j$$

Notation

n patches numbered 1 to n

b_i = radiosity of patch I

a_i = area patch I

total intensity leaving patch i = $b_i\, a_i$

$e_i\, a_i$ = emitted intensity from patch I

ρ_i = reflectivity of patch I

f_{ij} = form factor = fraction of energy leaving patch j that reaches patch i

Matrix Form

$$\mathbf{b} = [b_i]$$

$$\mathbf{e} = [e_i]$$

$$\mathbf{R} = [r_{ij}] \qquad r_{ij} = \rho_i \text{ if } i \neq j \quad r_{ii} = 0$$

$$\mathbf{F} = [f_{ij}]$$

Matrix Form

$$\mathbf{b} = \mathbf{e} + \mathbf{RFb}$$

formal solution

$$\mathbf{b} = [\mathbf{I}-\mathbf{RF}]^{-1}\mathbf{e}$$

Not useful since n is usually very large
Alternative: use observation that F is sparse

We will consider determination of form factors later

Solving the Radiosity Equation

For sparse matrices, iterative methods usually require only O(n) operations per iteration

Jacobi's method

$$\mathbf{b}^{k+1} = \mathbf{e} + \mathbf{RFb}^{k}$$

Gauss-Seidel: use immediate updates

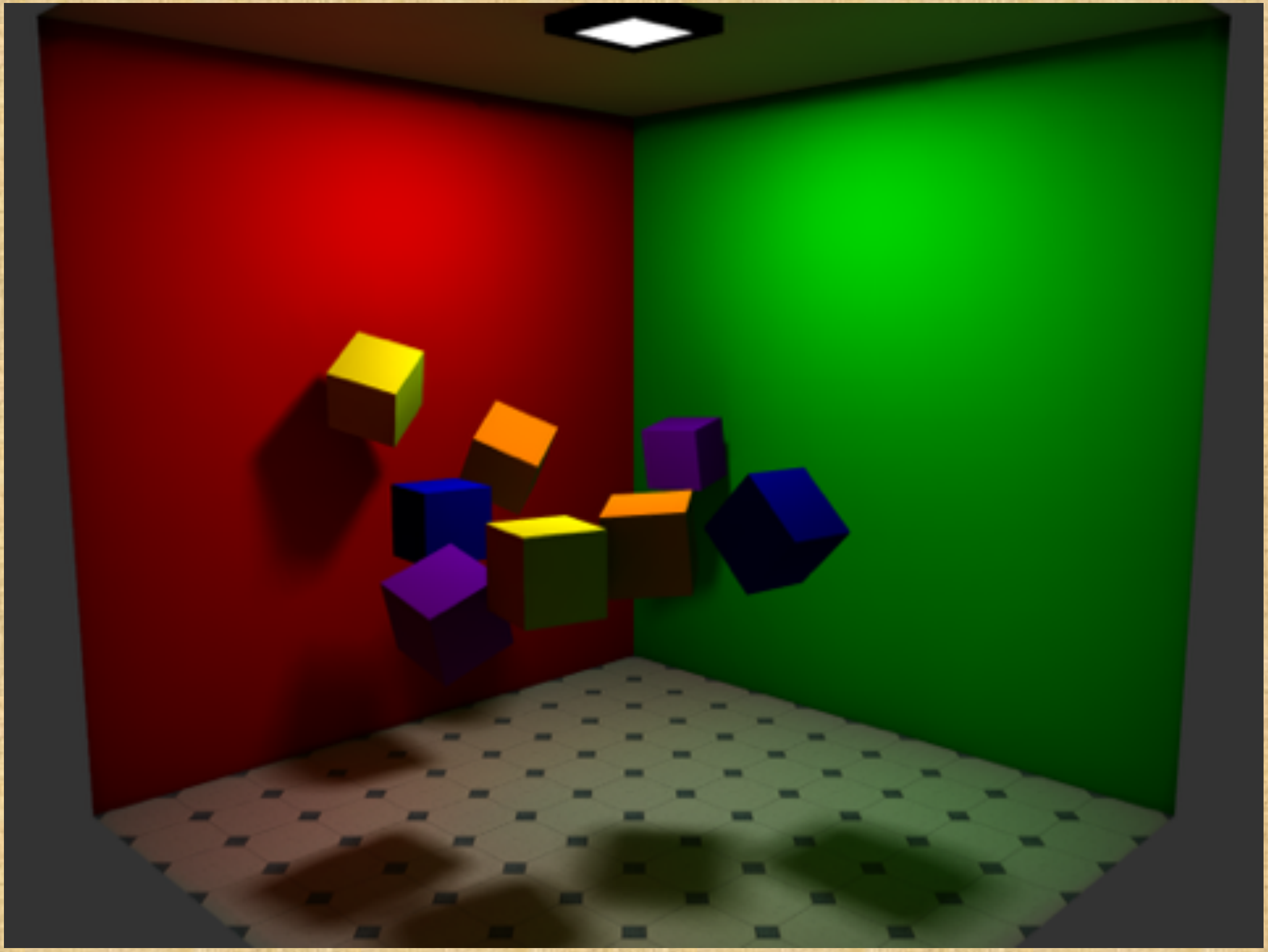
Series Approximation

$$1/(1-x) = 1 + x + x^2 + \ldots\ldots$$

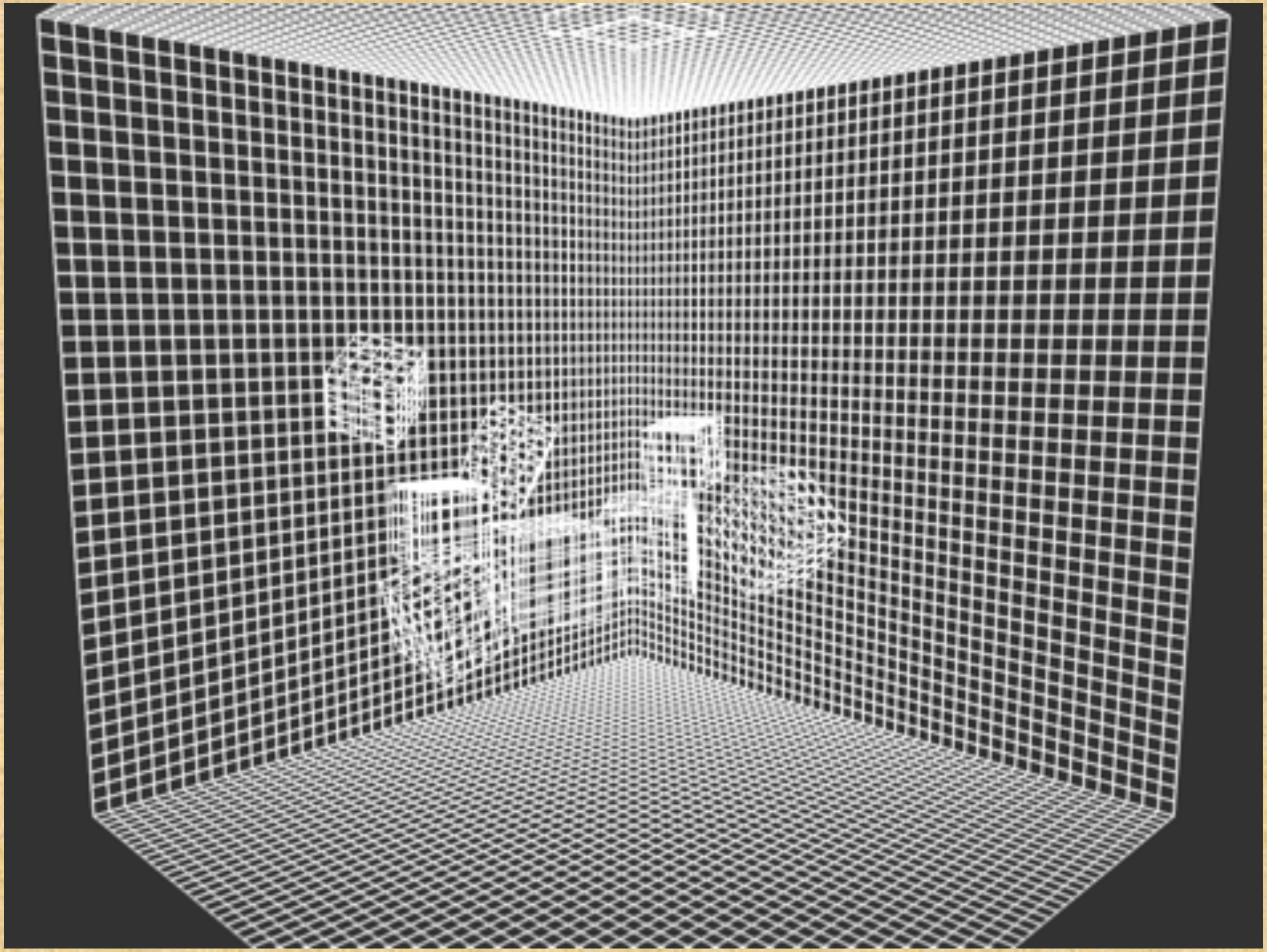
$$[\mathbf{I}-\mathbf{RF}]^{-1} = \mathbf{I} + \mathbf{RF} + (\mathbf{RF})^2 + \ldots$$

$$\mathbf{b} = [\mathbf{I}-\mathbf{RF}]^{-1}\mathbf{e} = \mathbf{e} + \mathbf{RFe} + (\mathbf{RF})^2\mathbf{e} + \ldots$$

Rendered Image



Patches

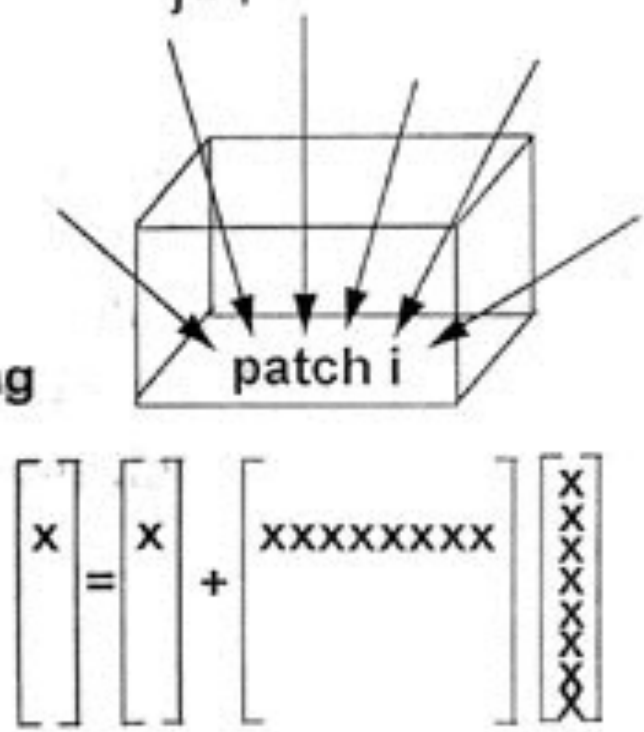


Gathering vs. shooting

Reconsider equation(5):

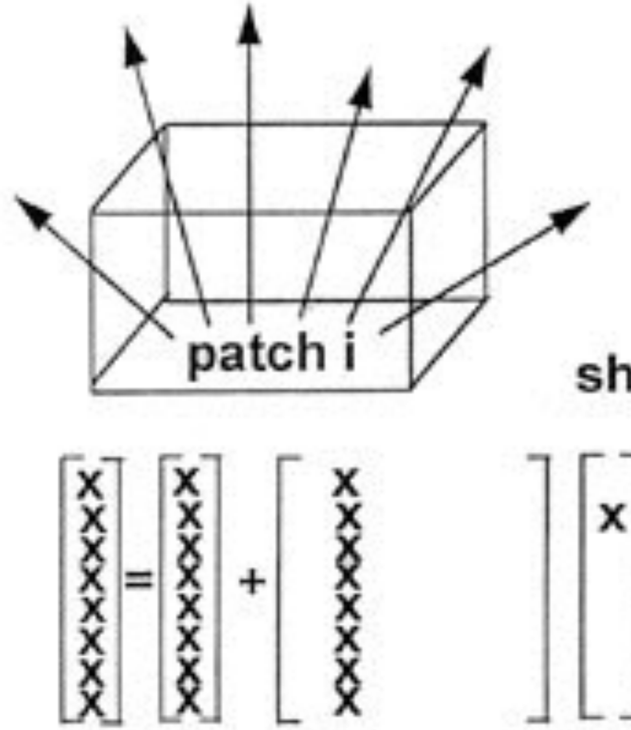
$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij}$$, we can find B_i due to $B_j = \rho_i B_j F_{ij}$

gathering



$$B_i = E_i + \sum_{j=1}^{n} (\rho_i F_{ij})\, B_j$$

shooting

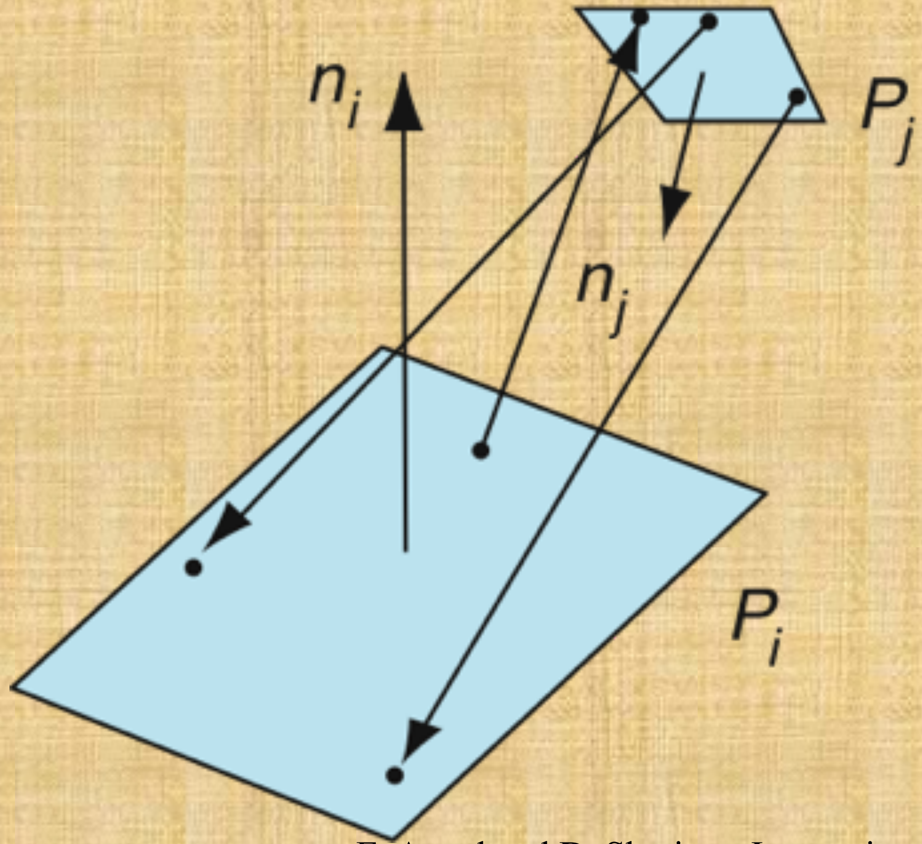


for all j :

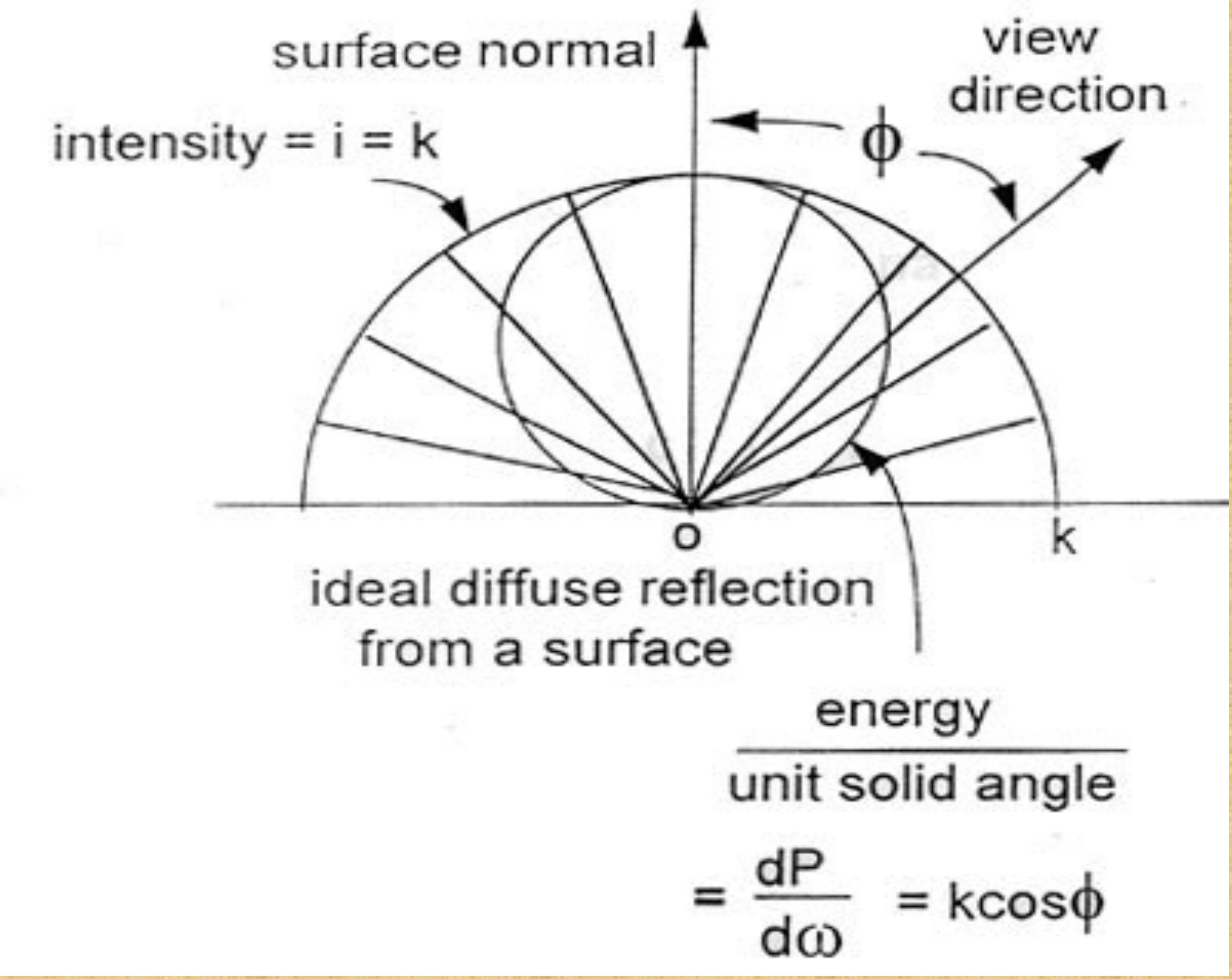
$$B_j = B_j + B_i(\rho_j F_{ji})$$

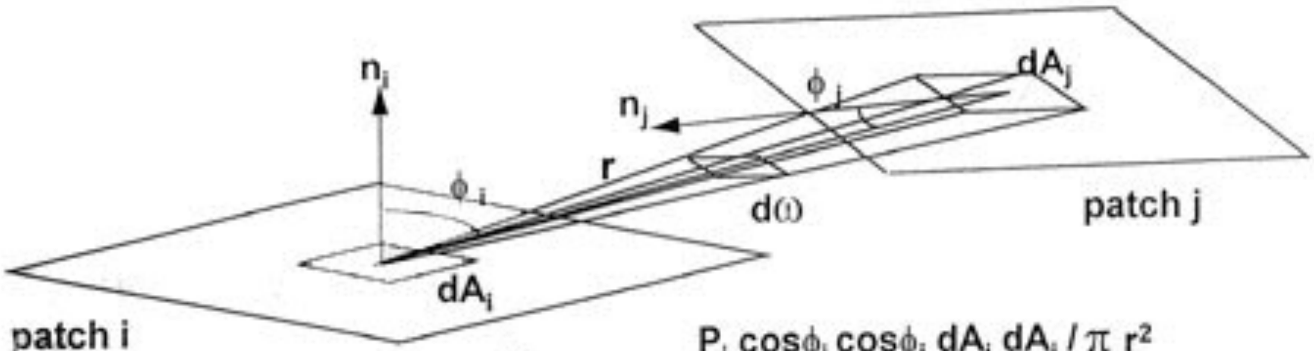
Computing Form Factors

- Consider two flat patches



Form-factor:


surface normal
view
direction
intensity = i = k
ϕ
o
k
ideal diffuse reflection
from a surface
energy
unit solid angle
$= \frac{dP}{d\omega} = k\cos\phi$



$$F_{dAi - dAj} = \frac{P_i \cos\phi_i \cos\phi_j \, dA_i \, dA_j / \pi r^2}{P_i \, dA_i}$$

$$= \frac{\cos\phi_i \cos\phi_j \, dA_j}{\pi r^2}$$

$$d\omega = \frac{\cos\phi_j \, dA_j}{r^2}$$

then , by (1), (2)

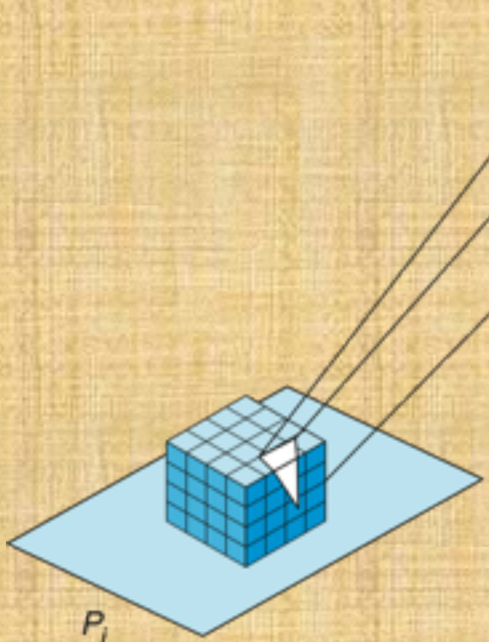
$$F_{dAi - Aj} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j \, dA_j}{\pi r^2}$$

$$dP_i dA_i = i_i \cos\phi_i \, d\omega \, dA_i$$

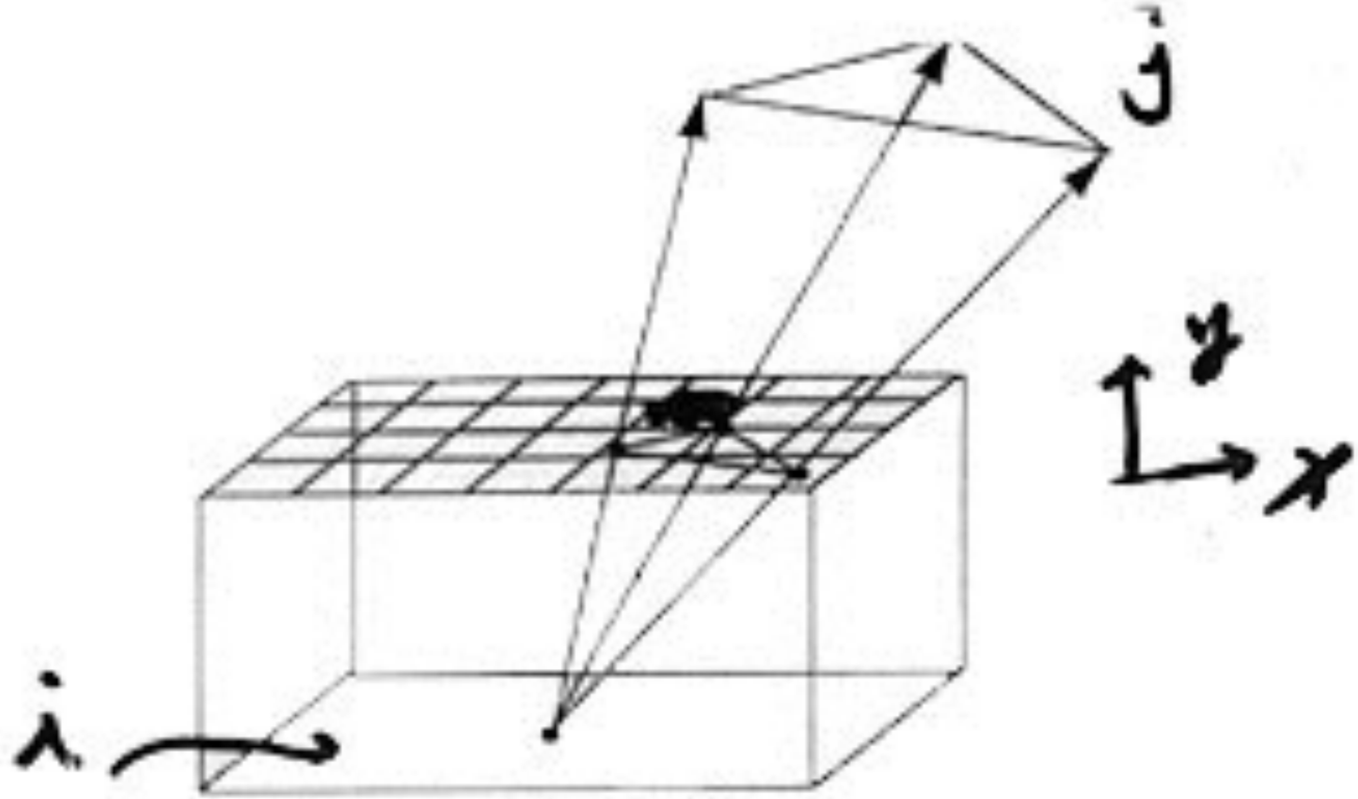
$$= \frac{P_i \cos\phi_i \cos\phi_j \, dA_i \, dA_j}{\pi r^2}$$

$$F_{Ai - Aj} = F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j dA_i \, dA_j}{\pi r^2} \quad \text{--- (4)}$$

Hemicube

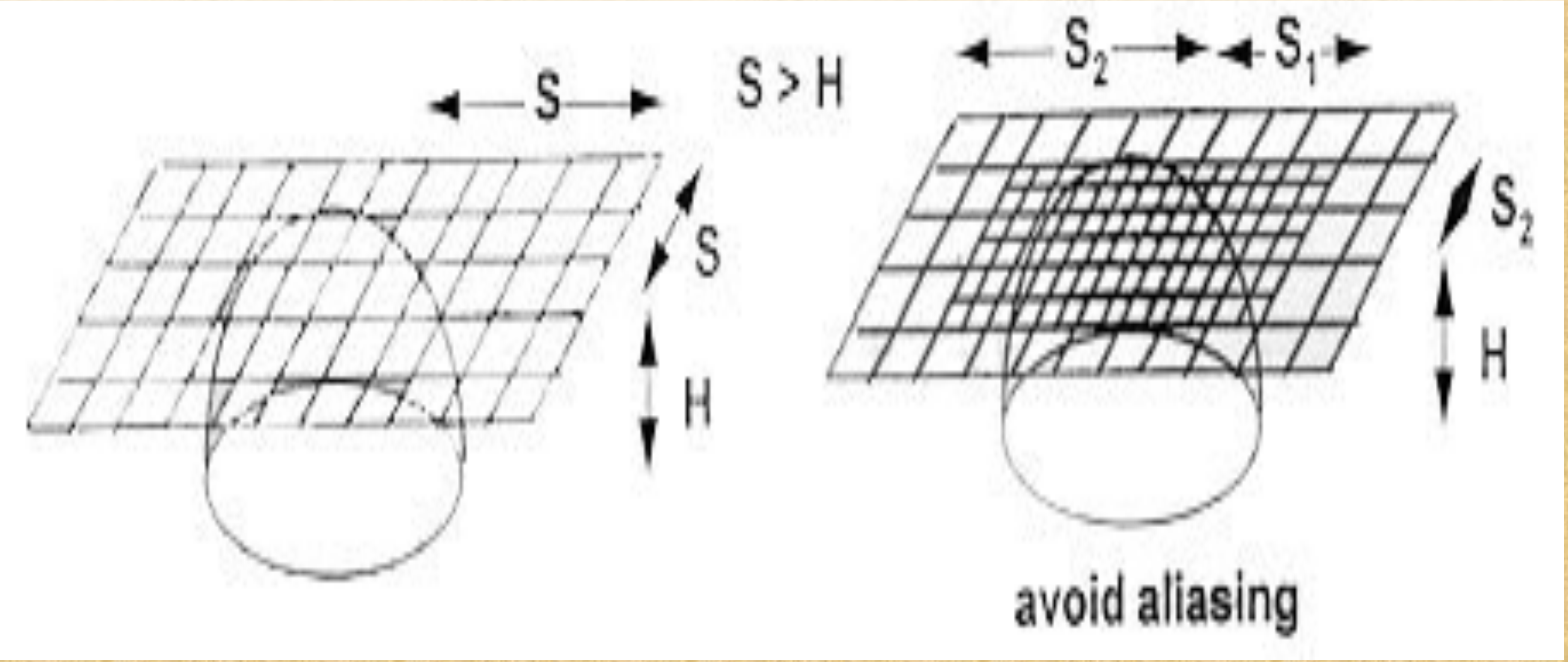


Hemi-Cube method, the Form-factor



$$= \frac{1}{\pi (x^2 + y^2 + 1)^2} \Delta A$$

Single plane algorithm



Progressive refinement of radiosity [M. Cohen, D. Greenberg]

Rearranging terms:

$$B_i - \mathrm{P}_i \sum_{j=1}^{n} B_j F_{i-j} = E_i$$

A set of simultaneous equations

$$\begin{bmatrix} 1-\rho_1 F_{1-1} & -\rho_1 F_{1-2} & \cdots & -\rho_1 F_{1-n} \\ -\rho_2 F_{2-1} & 1-\rho_2 F_{2-2} & \cdots & -\rho_2 F_{2-n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n-1} & -\rho_n F_{n-2} & \cdots & 1-\rho_n F_{n-n} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

for each iteration, for each patch i
for each patch j:
calculate the formfactors Fij using hemi-cube at patch i

$\Delta Rad = \rho_j \Delta B_i F_{ij} A_i / A_j$ **/*update change since last time patch j** shot light */

$\Delta B_j = \Delta B_j + \Delta Rad;$ /* update total radiosity of patch j */

$B_j = B_j + \Delta Rad;$

$\Delta B_i = 0;$ /* reset unshot radiosity for patch i to zero */

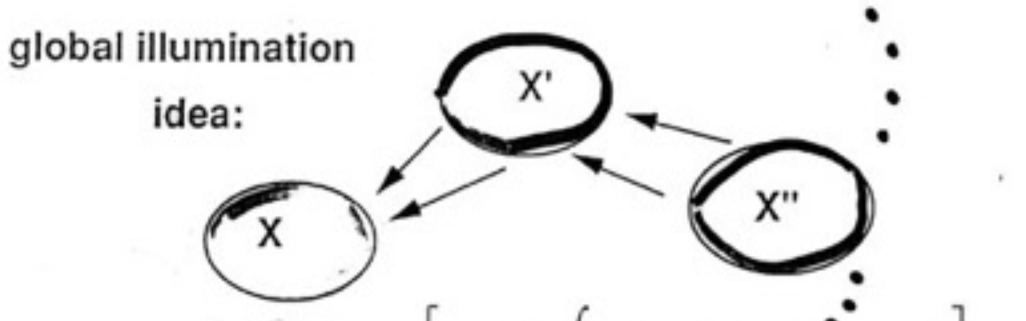
initialization:
for all patch i:
if patch i is a light source,
then $B_i = \Delta B_i = E_i$
else
$Bi = \Delta B_i = 0.$

The algorithm of progressive refinement

Evolution of CG hardware

- MMX-Intel (SIMD, Single instruction, multiple data set)
- GPU
- GPGPU

Radiosity



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

$$B_i A_i = E_i A_i + \mathbf{P}_i \sum_j B_j F_{ji} A_j$$

$$B_i = E_i + \mathbf{P}_i \sum_{j=1}^{n} B_j F_{j,i} \frac{A_j}{A_i}$$

Hardware Systems
Old Hardware Systems in 1991

- VRAM
  - consider 1280 * 1024 screen with 32 bit/pixel, refresh at 60 HZ, the memory access time=1/(1280*1024*60)=12.7 nanoseconds, ordinary DRAM is at 100 ~ 200 nanoseconds
  - parallel-in / serial-out data register as a second data port
- TMS 34020 (2D Graphics)
  - pixel-block transfer 18 million 8 bit pixels/second
  - block-write(4 memory locations/once) -> fill an area at 160 million 8 bit pixels/second

Hardware Systems –old systems(II)

- i860(3D graphics)
  - 13 MFLOPS 33 VAX MIPS, 500K vector transformation/sec
  - packed 64 bit data; for 8-bit pixels, 8 operations occur simuultaneously. 50K Gouuraud-shaded 100-pixel triangles/second
- bottlenecks
  - floating-point geometry processing
  - Integer pixel processing
  - Frame-buffer memory bandwidth

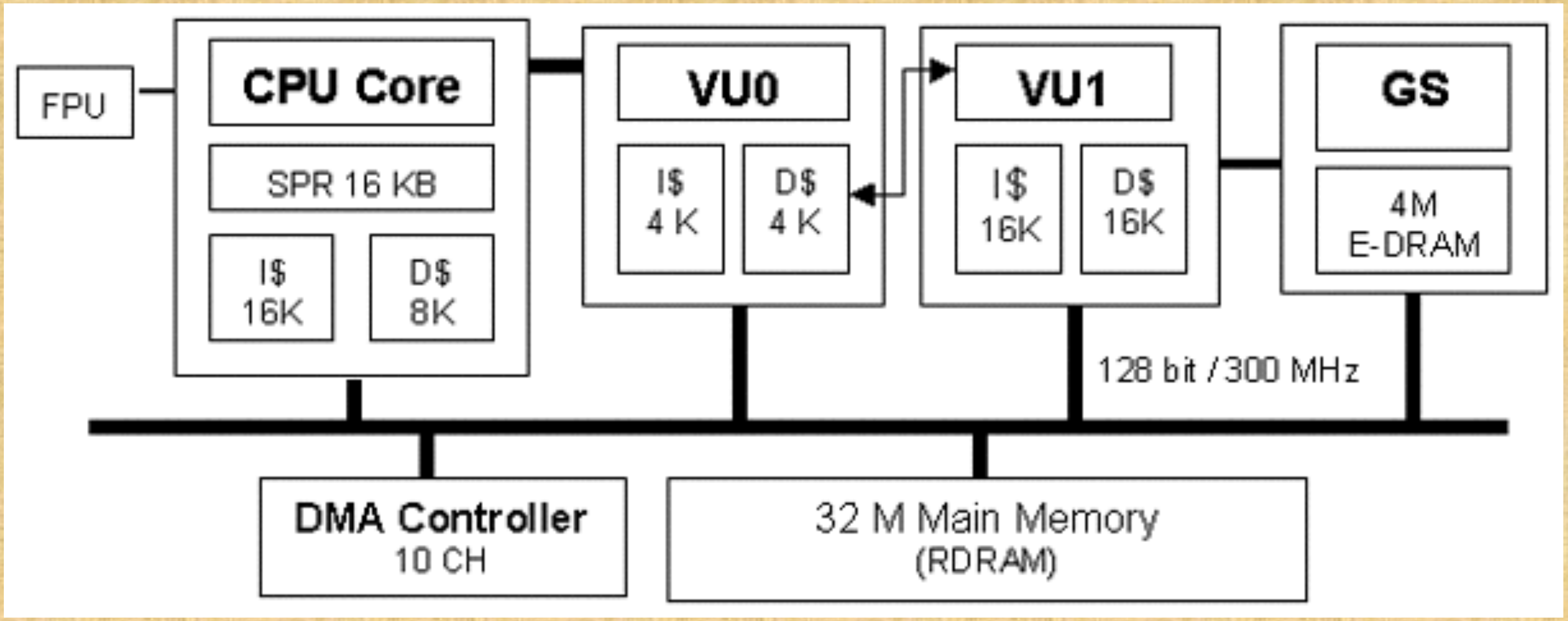
True Color display—Old Systems

- Hercules card (380 or 486 machine)
  - It contains a TMS34010 and VRAMS
  - we can program it with MicroSoft C (easy)
  - 16 bits/pixel, 5 bit red, 5 bit green, 5 bit blue,
  - 640*480*16 or double buffer 640*480*8 (for fast animation)
  - a program that can take (r, g, b, x) formats (24 bit format) and display, for example, the teapot
  - a set of demo programs, including a flight simulator

Hardware system for graphics

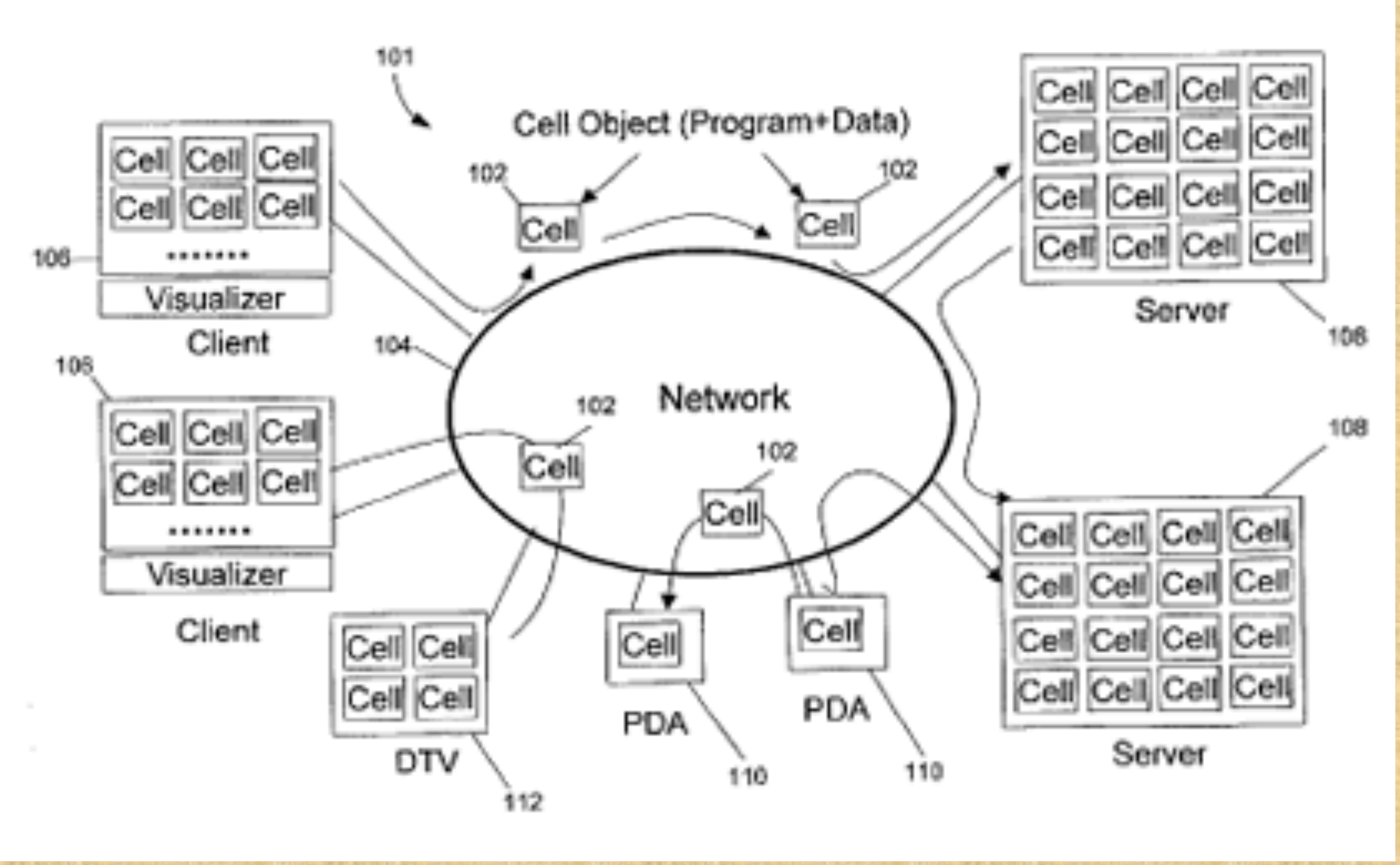
- General purpose system (MIMD: iWarp etc) H.T.Kung
- Specific system, eg: Silicon Graphics' IRIS, 4D/ 240GTX (MIMD)
  - 100,000Gouraud-shaded, Z-buffered quadrilaterals
  - CPU subsystem: 4 shaded-memory multiprocessors
  - Geometry subsystem: 5 floating-point processors, each 20 MFLOPS (Weifek 3332)
  - Scan-conversion subsystem: a long pipeline
  - Raster subsystem: 20 image engines, each for 1/20 screen, (4*5 pixel interleaved)
  - Display subsystem: fine graphics processor, each assigned 1/5 columns in the display

Graphics Game Machine Hardware PlayStation 2 architecture



PlayStation 3 spec.

PlayStation 3 architecture



NVIDIA RSX

- 550MHz Core
- 300 Million Transistors
- 136 Shader Operations per Cycle
- Independent Pixel/Vertex Shaders
- 256MB GDDR3 RAM at 22.4GB/sec
- External Link to CPU at 35GB/sec (20GB/sec write + 15GB/sec read)
- 1920x1080 Maximum Resolution

ATI Radeon X800/X850

- (540MHz / 1180MHz)
- 16 Pixel Pipelines (2 Vector + 2 Scalar + 1 Texture ALUs)
- 6 Vertex Pipelines (1 Vector + 1 Scalar ALUs)
- 92 Shader Operations per Cycle
- 256MB GDDR3 RAM at 37.76GB/sec
- External Link to CPU at 8GB/sec

GPGPU: general purpose GPU

- CUDA programming
- Course by Professor Wei-Chao Chen (陳維超)

ICG TERM PROJECT LISTING

1. Animation of articulated figures (linked)
2. Rigid body animation, domino blocks (Newton’s laws)
3. A viewing/editor system for curved surfaces with textures (curves and patches)
4. Photon Mapping, Radiosity Method
5. Recursive Ray tracing animation with software/GPU acceleration

Term project 2

6. Volume rendering for a set of tomography slides(台大醫院資料 etc.)
7. Face modeling, lip sync, face de-aging/ aging
8. Sketch system for animation (Teddy system)
9. Oil painting and water color effects for images
10. 3D morphing and animation with skeleton mapping, mesh animation

Term project 3

11. Motion retargeting (motion of cats likes that of a human)
12. Hardware Cg acceleration research and applications
13. Beautifying Images (Color harmonization, face beautification, photo beautification, photo ranking)
14. 3D video, stereo video, DSLR_Bokeh_blur simulation (from depth images/video), image deblur
15. Others—Human Computer Interface, Installation Arts, Water Rendering etc.