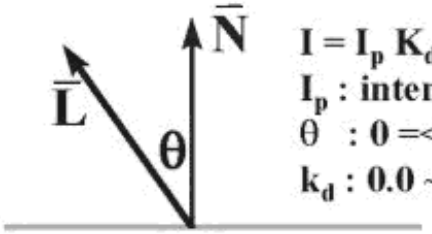
Shading

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

Illumination model

1) **Ambient light** (漫射)(or 環境反射)
   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)(or 漫反射)
   I = Ip (r, g, b) * Kd * Obj(r, g, b) * COS(**θ**)
   Ip (r, g, b): light color



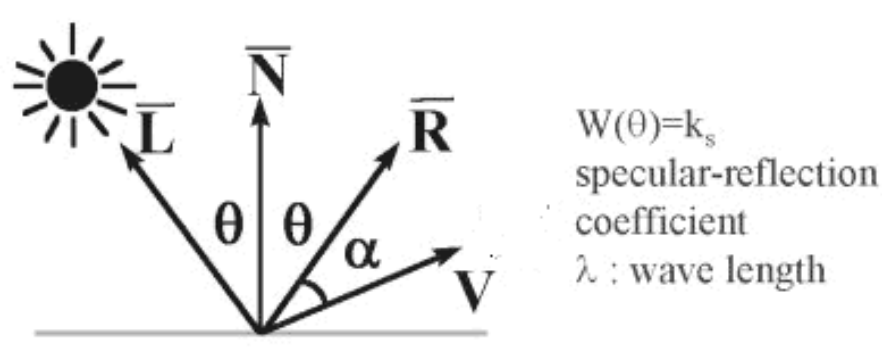
$I = I_p K_d \cos\theta$
I_p : intensity of light source
θ : $0 =< \theta <= 90$
k_d : 0.0 ~ 1.0 (material dependent)

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)

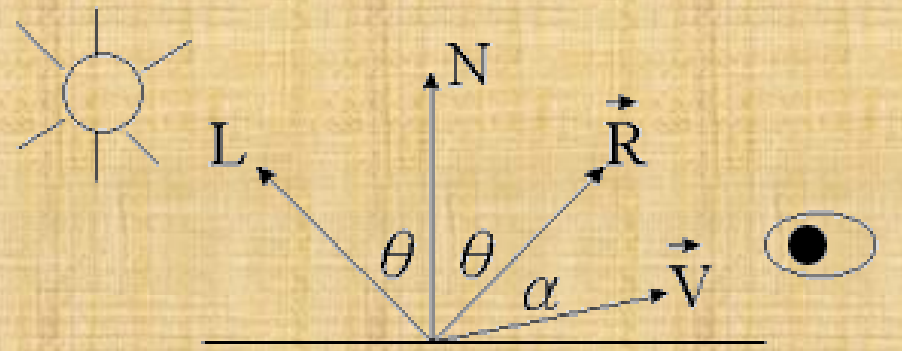
- **I = Ks * Ip(r, g, b) * $COS^n(\alpha)$,**
  Ks = specular-reflection coef.



- 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



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

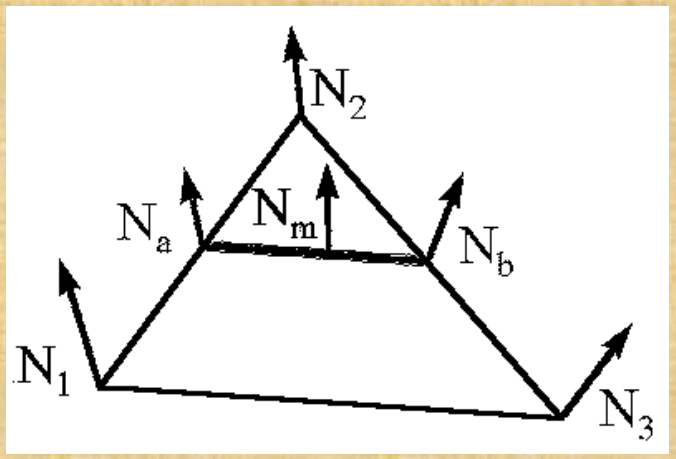
Polygon shading : linear interpolation

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)

Henri Gouraud (at 2018 Siggraph)

English pronunciation: /ger'raud/

French pronunciation: /'gu:hu:/

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

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

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 shading

TEST
FOR I
N P
U T

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

Complexity:

Flat shading:

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

N* one time lighting

Gouraud Shading:

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

Shong Shading:

(bi-linear interp. + one time lighting)*N*A

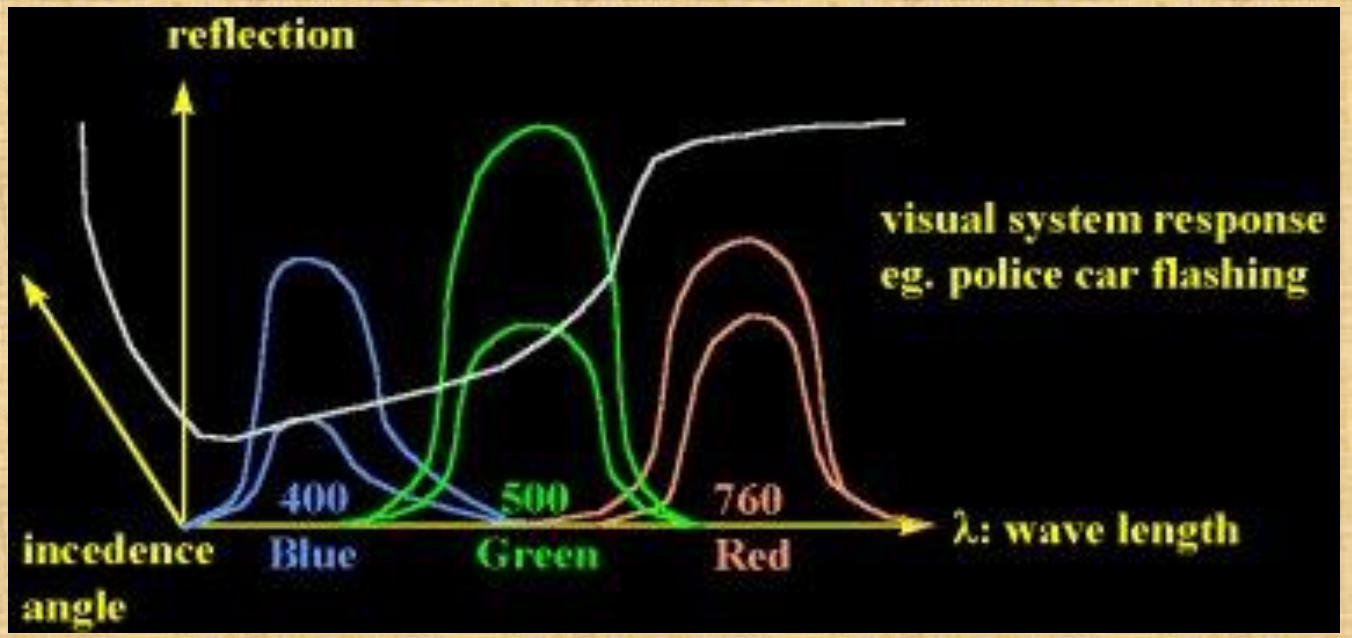
A >> 3 ingeneral

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

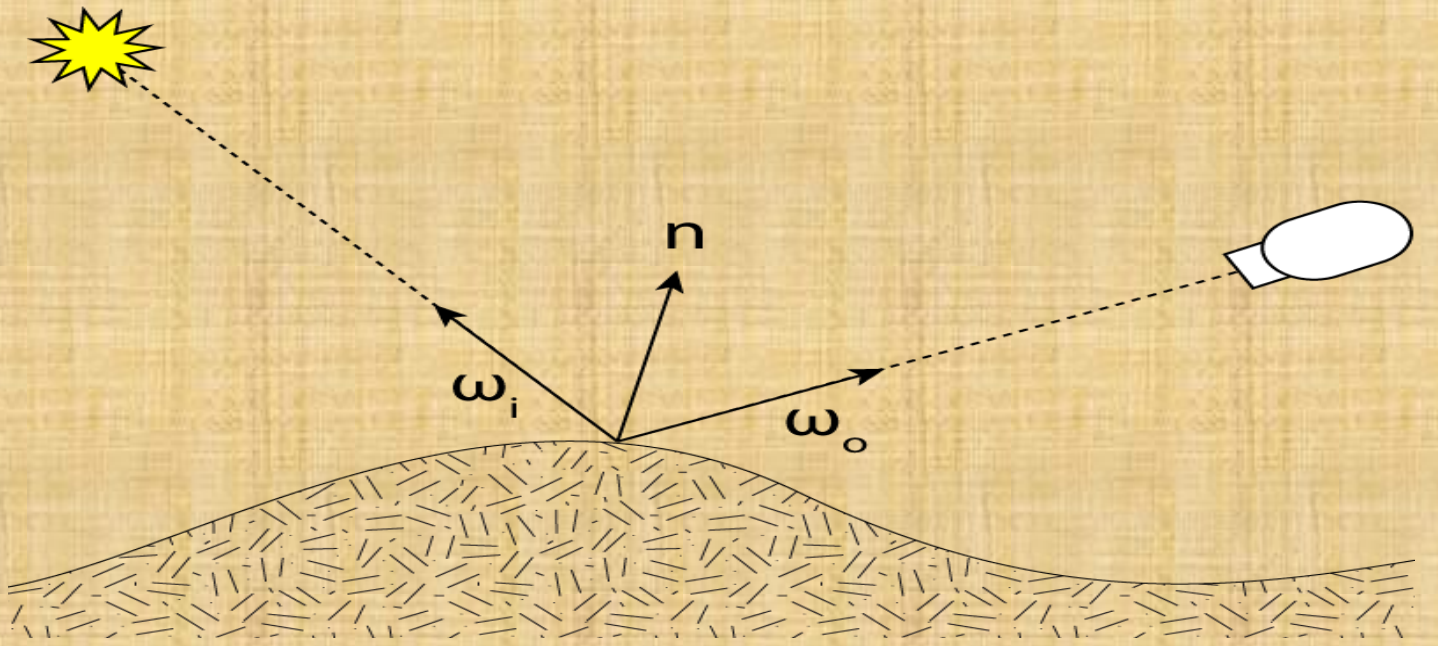
What is the color of copper?

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

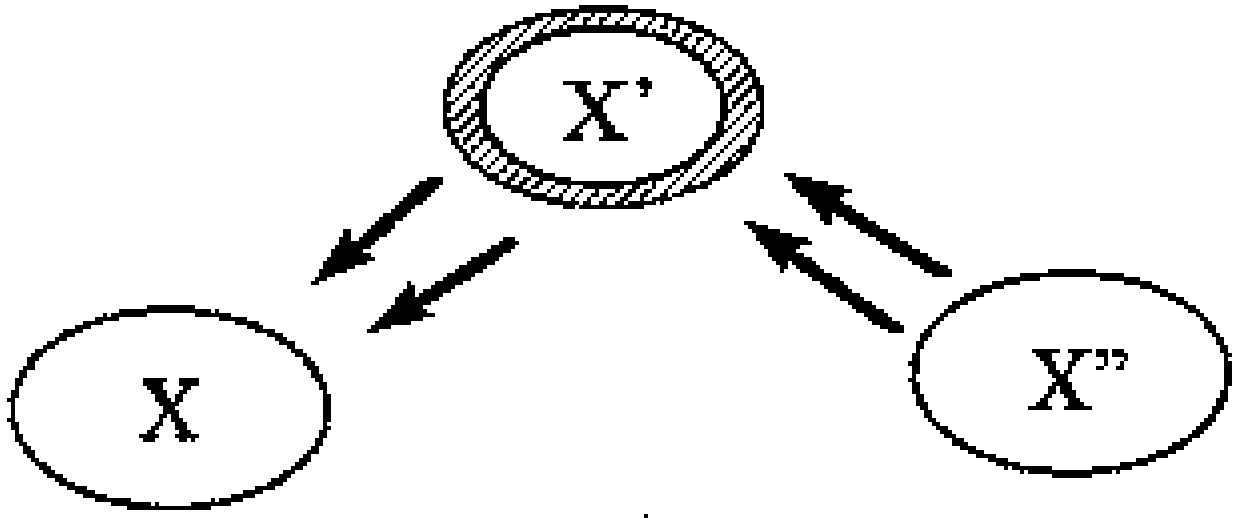


New method: BRDF:Bi-directional Reflectance Density Function

- 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:

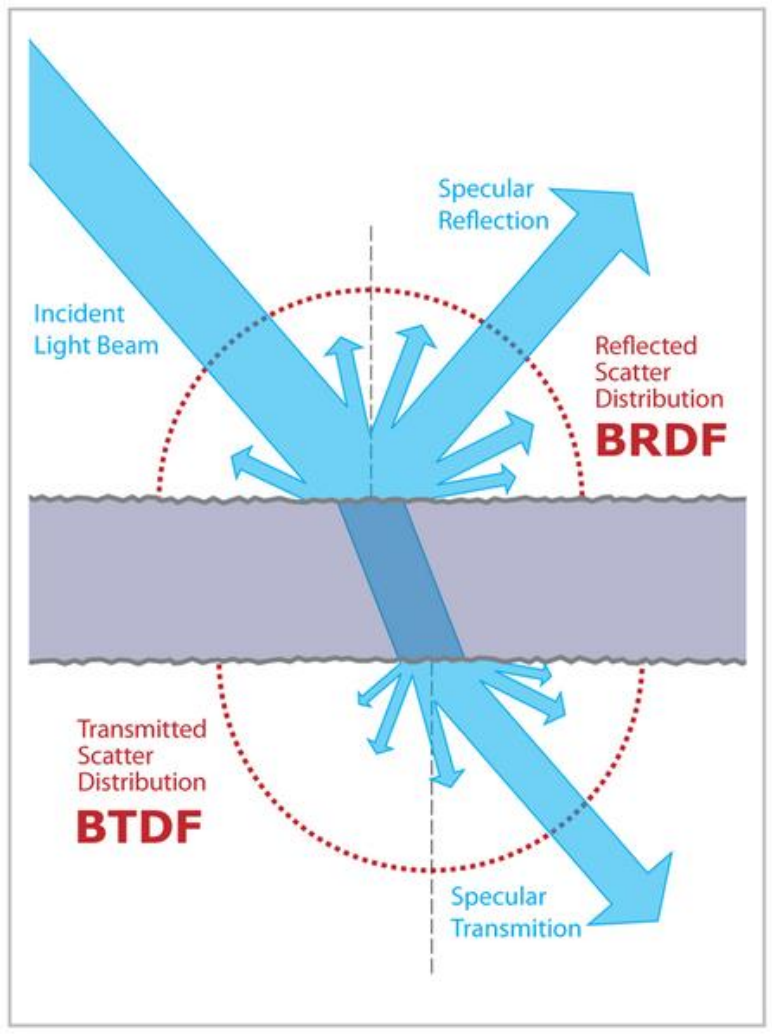
$$F(\omega_i, \omega_o) = dL(\omega_o)/dE(\omega_i) = dL(\omega_o)/L(\omega_i)\cos\theta_i \, d\omega_i$$

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

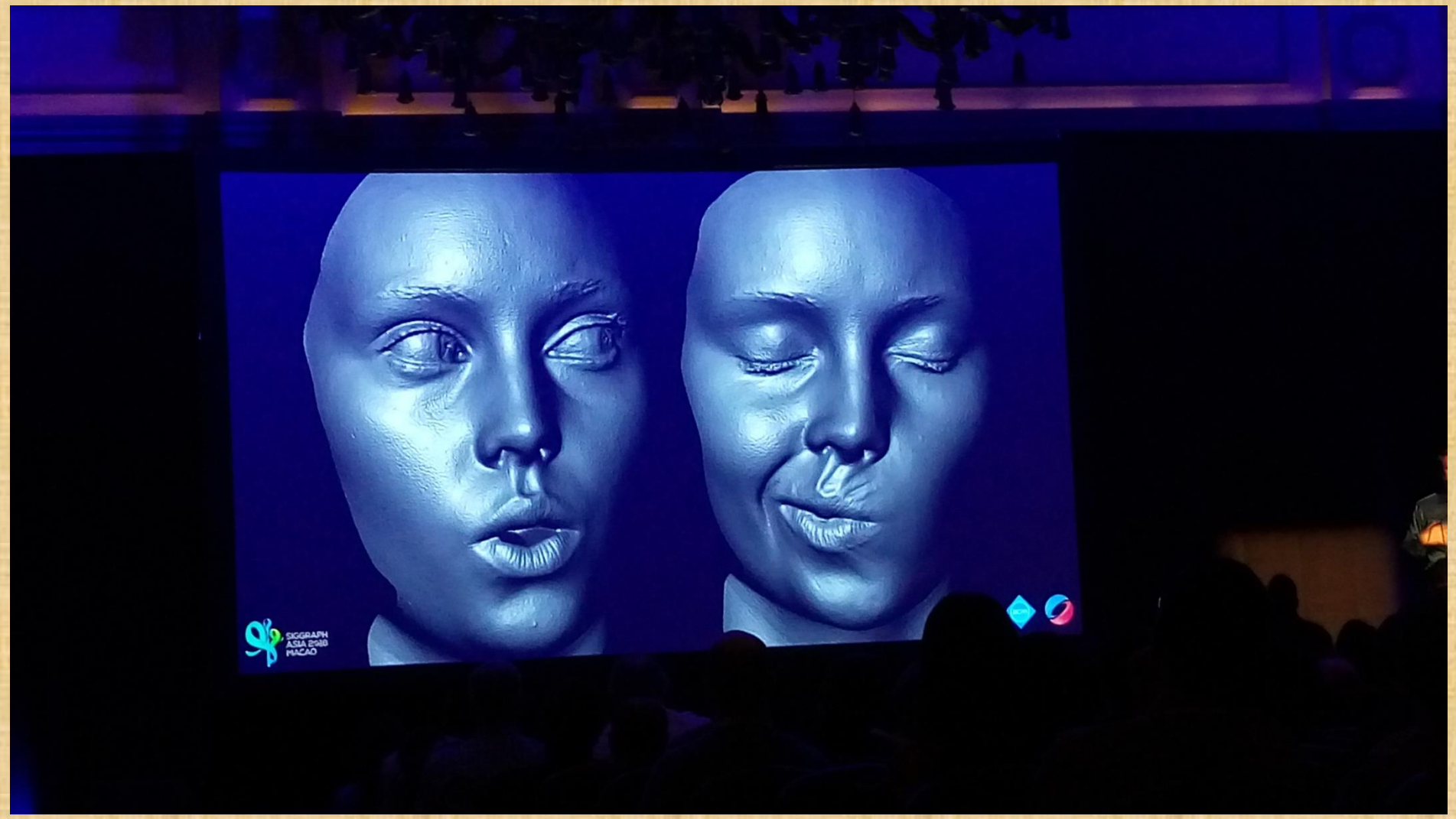
BSSRDF

- 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)**



3D face scan



Statute of Obama, US president

Homework #1

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

  Triangle fr, fg, fb, br, bg, bb

  x y z nx ny nz

  x1, y1, z1, …. ,

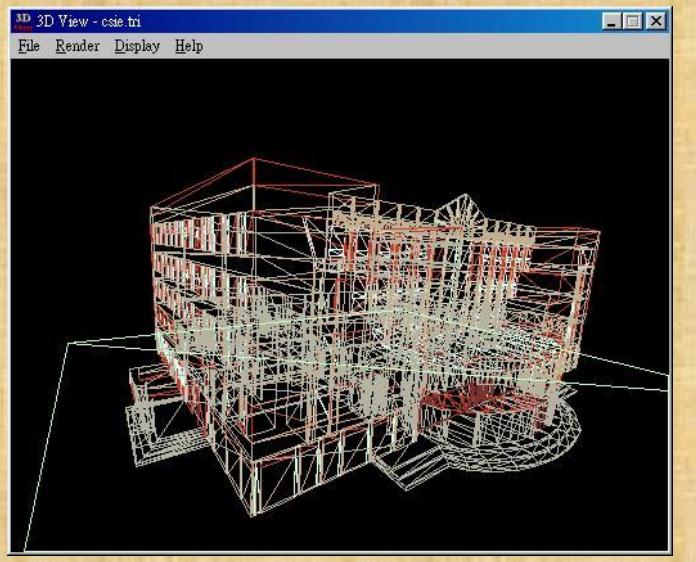
  ,X2, y2, z2 …….,

  /* 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: to be announced**
- Output: shaded objects with at least two lights
- 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 WebGL library calls

Polygon file format used

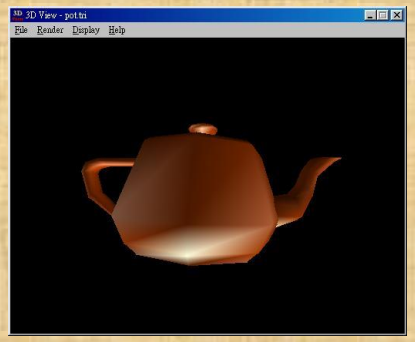
- e.q.
  Triangle fr fg fb br bg bb
  x1 y1 z1 nx1 ny1 nz1
  x2 y2 z2 nx2 ny2 nz2
  x3 y3 z3 nx3 ny3 nz3
  Triangle
  - Where

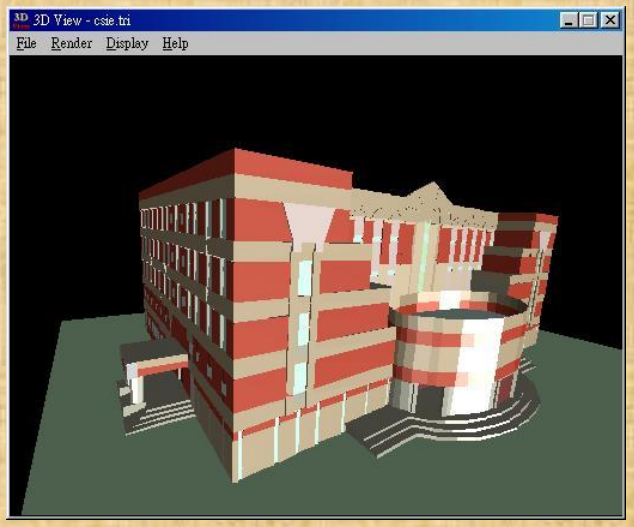
    fr, fg, fb are foreground colors (Red Green Blue)
    nx, ny, nz are vertex normal

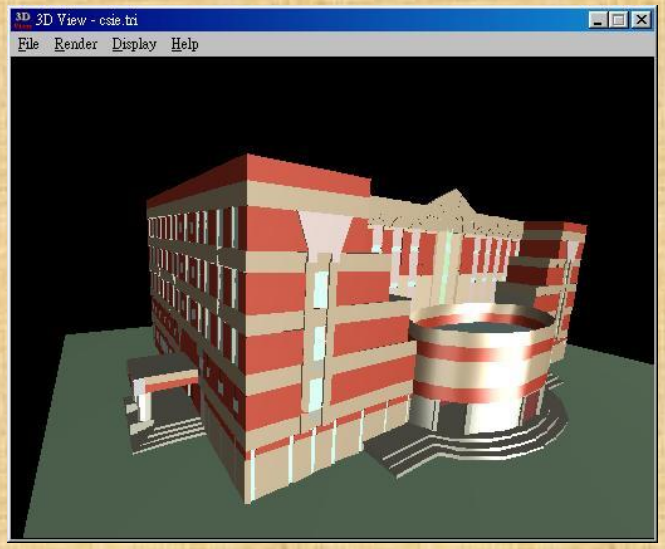
Other formats (more efficient)

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

HW#1: expected results







Visible-Surface Determination

- **The painter's algorithm**
- **The Z-buffer algorithm** (chap 5.8, textbook)
  - 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·

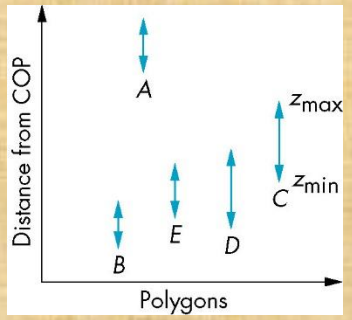
  ---v.s.---painter's algorithm

  Binary Space PartitionTrees(BSP tree) (Chap 9.10, p505, textbook)

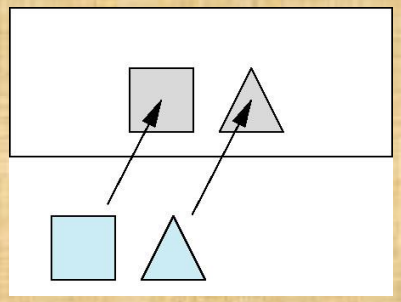
Determine the depth order!
1. Mountain, 2. bridge, 3. people 4. Hat

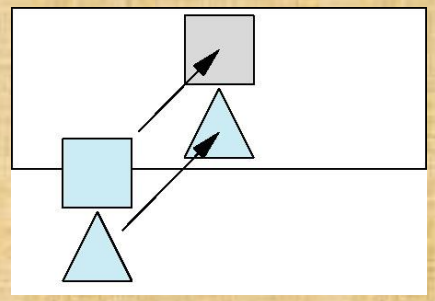
Depth Sort: Easy Cases

- A lies behind all other polygons
  - Can render

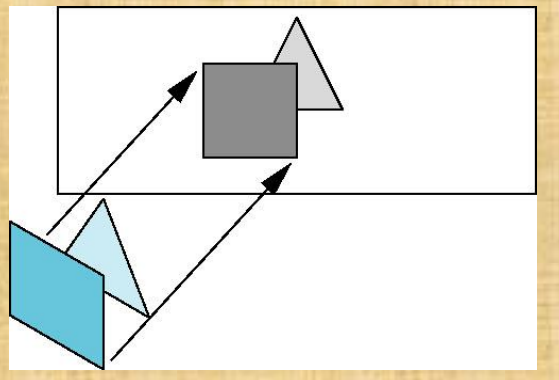


- Polygons overlap in z but not in either x or y
  - Can render independently

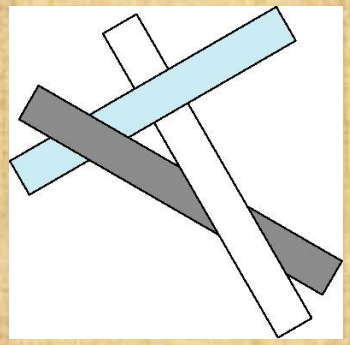




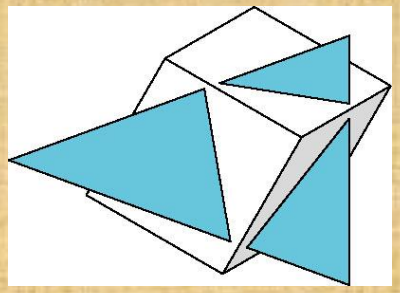
Hard Cases



Overlap in all directions but can one is fully on one side of the other

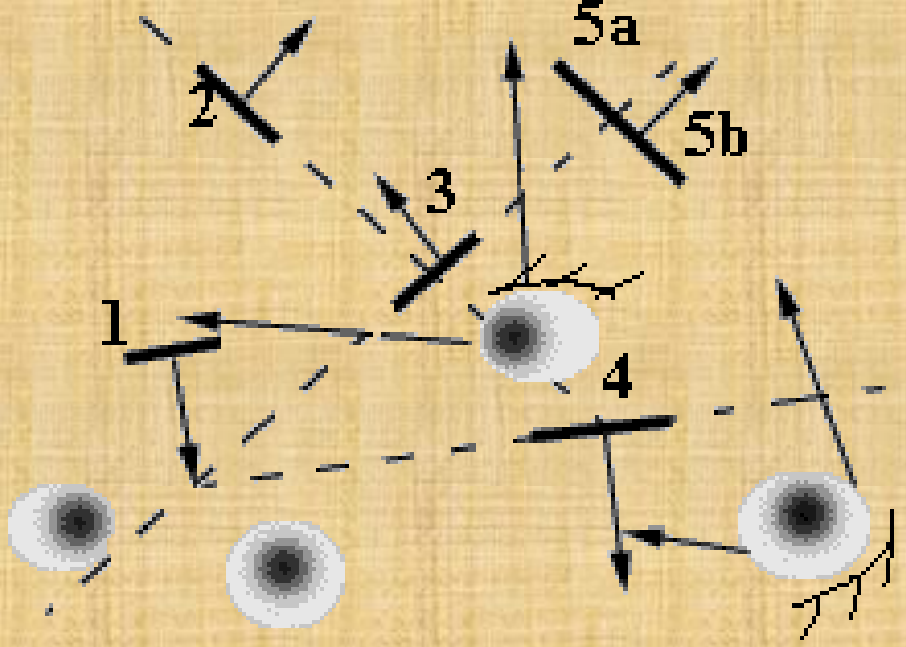


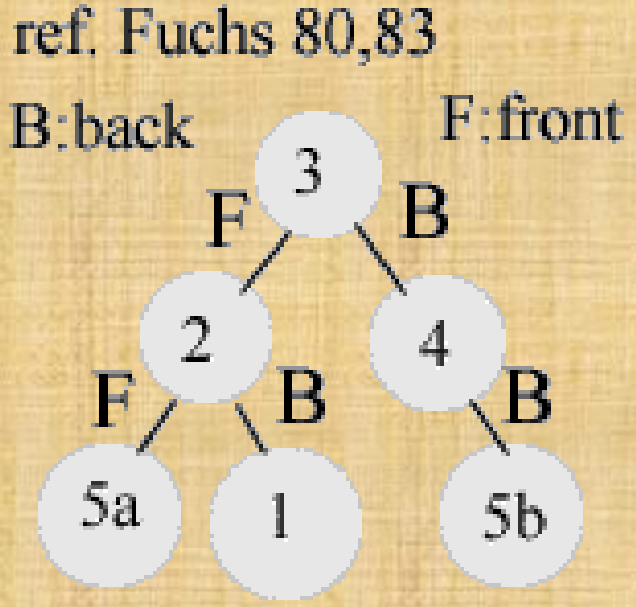
cyclic overlap



penetration

BSP: Binary Space PartitionTrees(BSP tree)





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

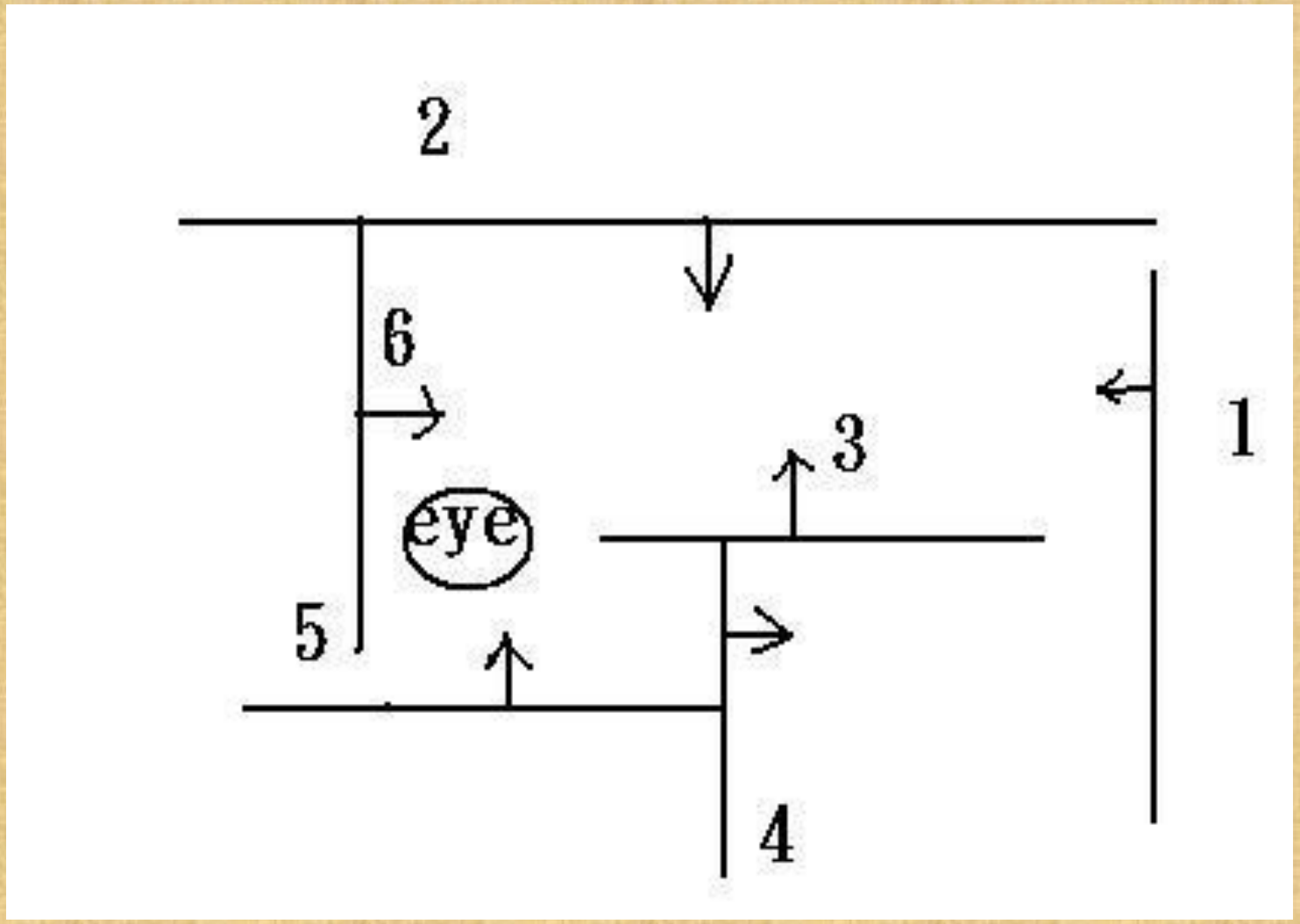
if Viewer is in front of root, then

- 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 shading

TEST
FOR I
N P
U T

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

Complexity:

Flat shading:

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

N* one time lighting

Gouraud Shading:

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

Shong Shading:

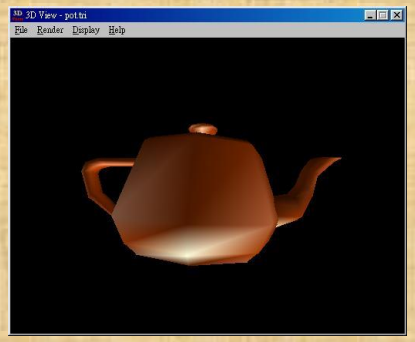
(bi-linear interp. + one time lighting)*N*A

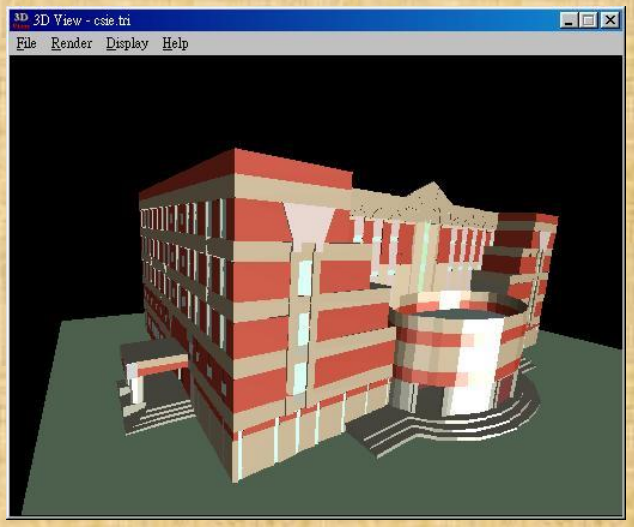
A >> 3 ingeneral

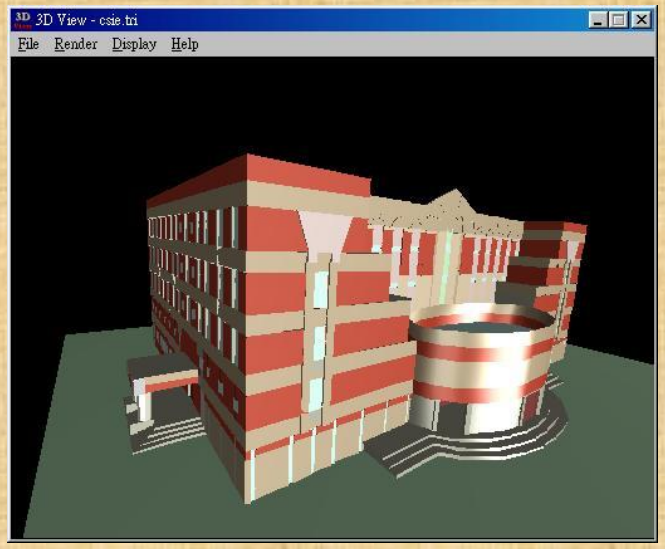
Homework #2 Unity 3D games

- Modify an exisiting game
- Change the way a canon ball is fired (weapon system), tank motion control etc.

HW#1: expected results



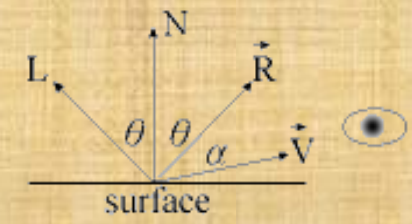




HW#1: formula

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

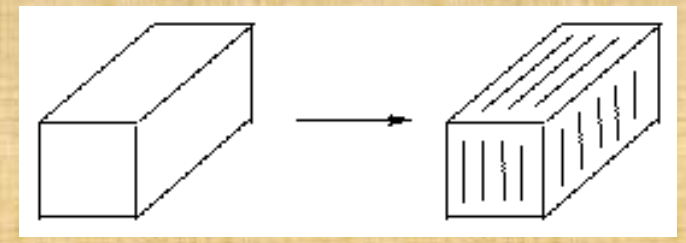
  Where $Ia = \begin{matrix}\text{object}\\ \text{color}\end{matrix}$, $Ip = \left(\frac{R}{MAX}, \frac{G}{MAX}, \frac{B}{MAX}\right)$



  - 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

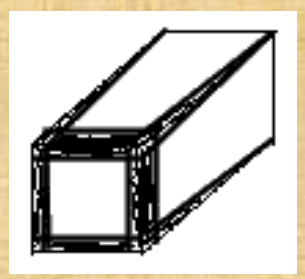
Visible line determination

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!



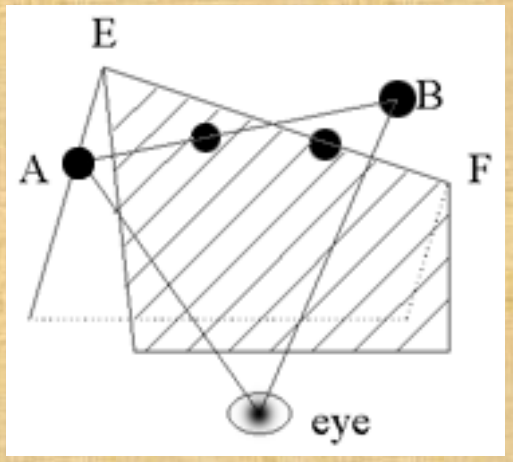
   (hollow triangle, mesh)

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

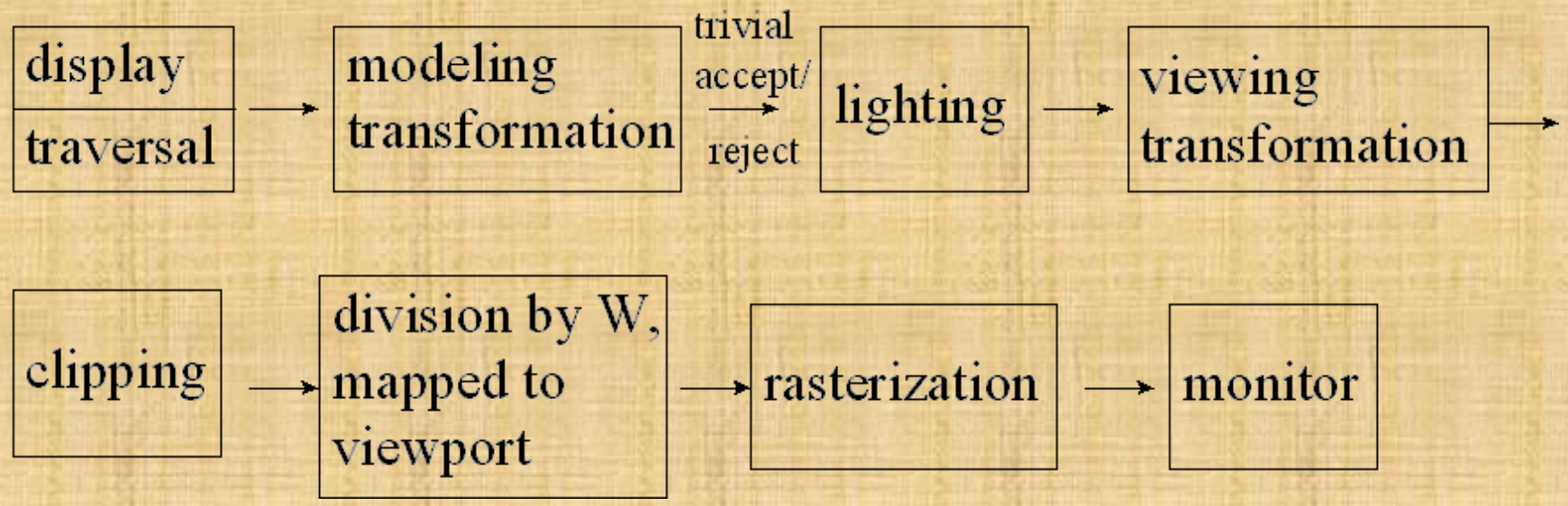


Visible - line determination: Appel's algorithm

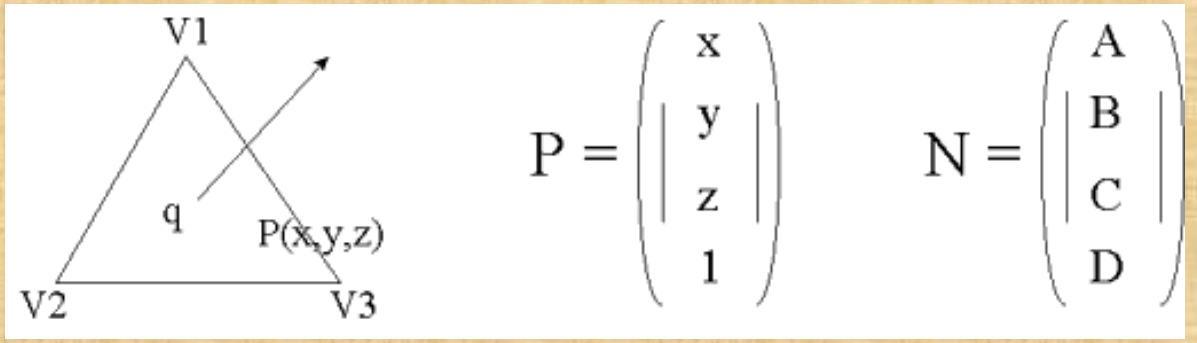
- quantitative invisibility of a point=0 --> visible
- quantitative invisibility changes when it passes "contour line".
- contour line:(define)
- vertex traversing
  - EF: contour line
  - AB: whether this line segment is partially visible?



Standard Graphics Pipeline



How to transform a plane? a surface normal?



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. $Q^T * M = I$

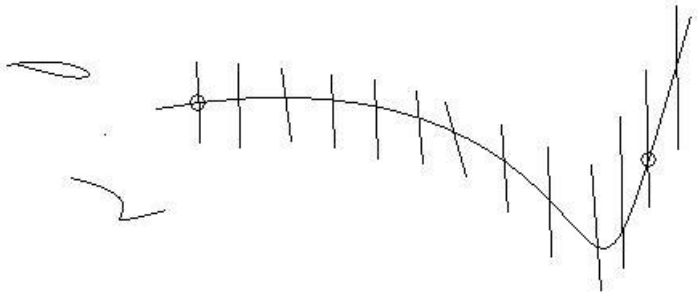
$\therefore Q^T = M^{-1}$ $Q = (M^{-1})^T$

similar, the surface normal is transformed by Q, not M!!

Aliasing, anti-aliasing

Aliasing effects

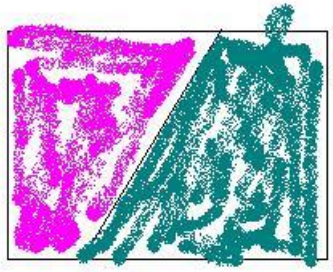
Sampling theory: two times sampling frequency



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



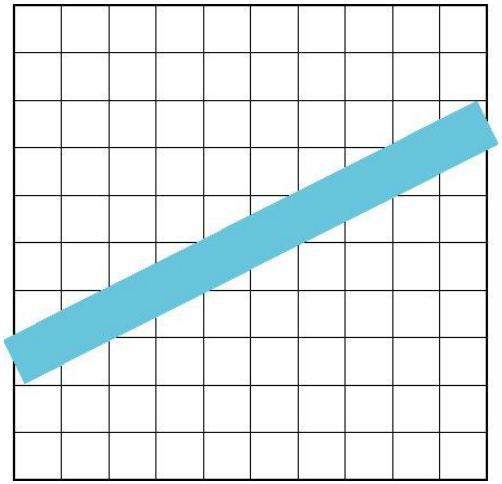
pixel

area weighted

The University of New Mexico

Aliasing

- Ideal rasterized line should be 1 pixel wide



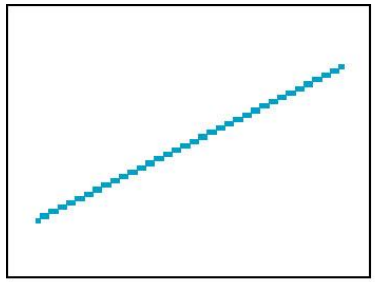
- Choosing best y for each x (or visa versa) produces aliased raster lines

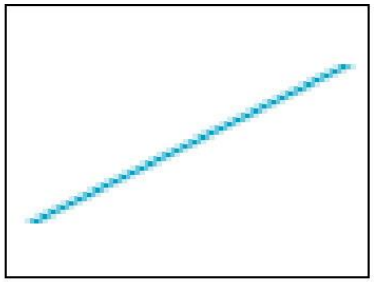
The University of New Mexico

Antialiasing by Area Averaging

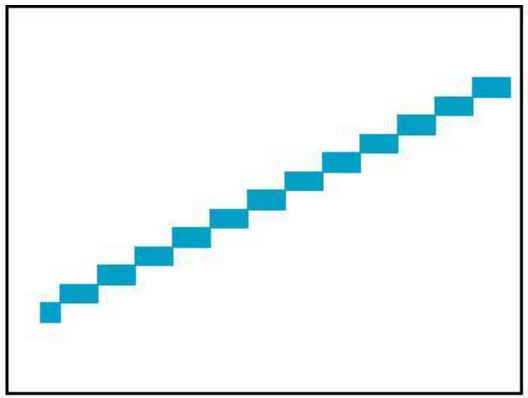
- Color multiple pixels for each x depending on coverage by ideal line

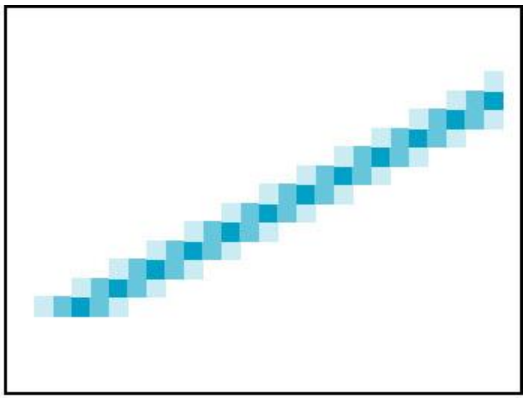
original





antialiased



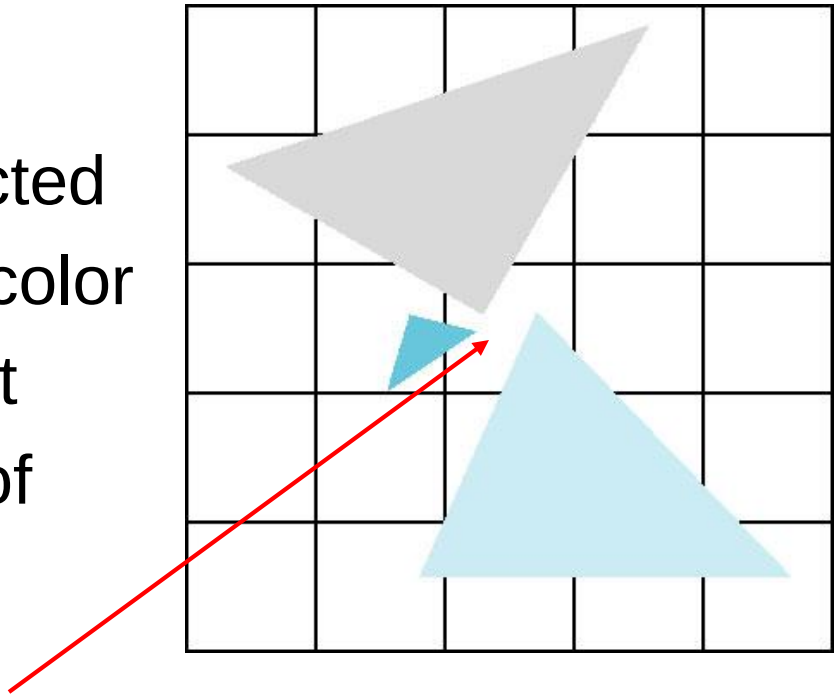


magnified

The University of New Mexico

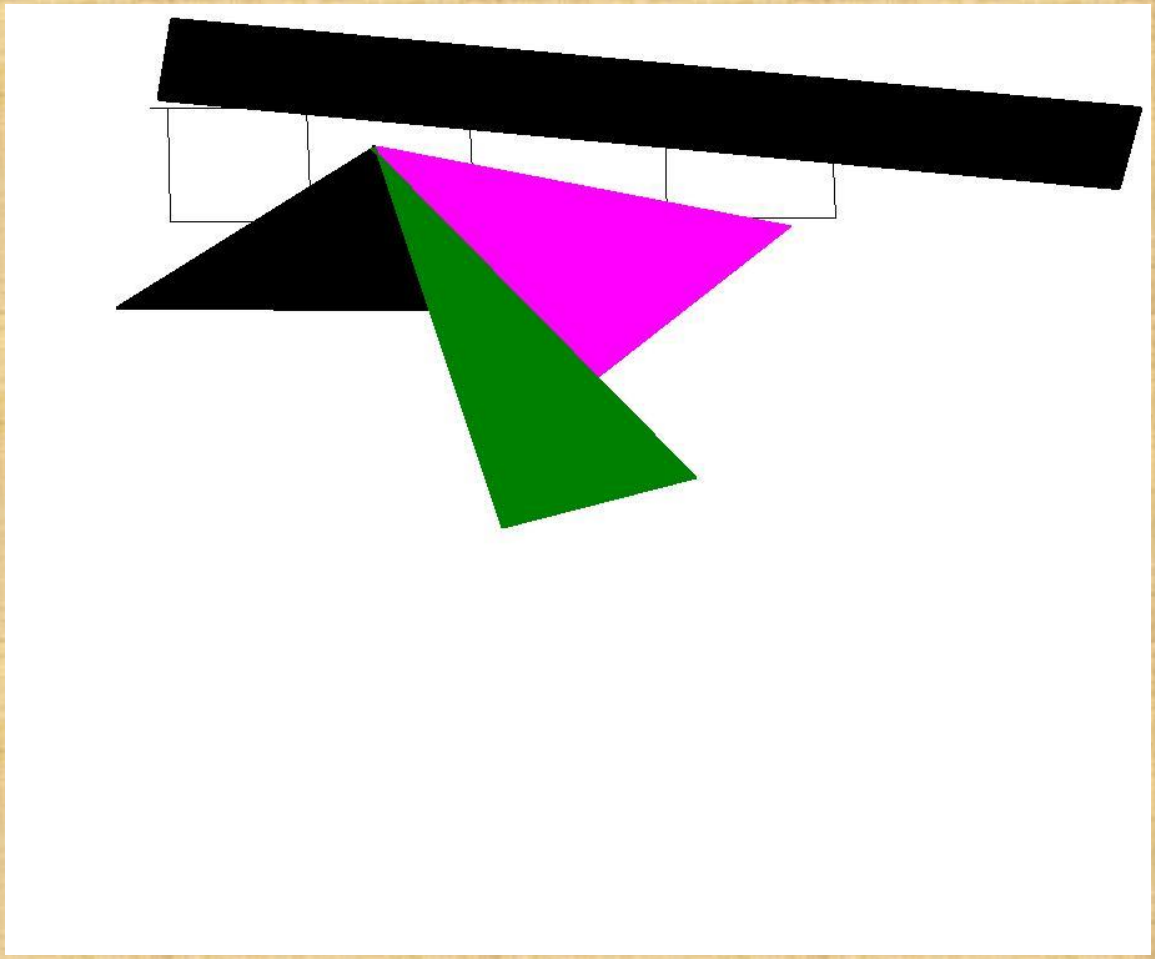
Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Jaggedness of edges
  - Small polygons neglected
  - Need compositing so color of one polygon does not totally determine color of pixel



All three polygons should contribute to color

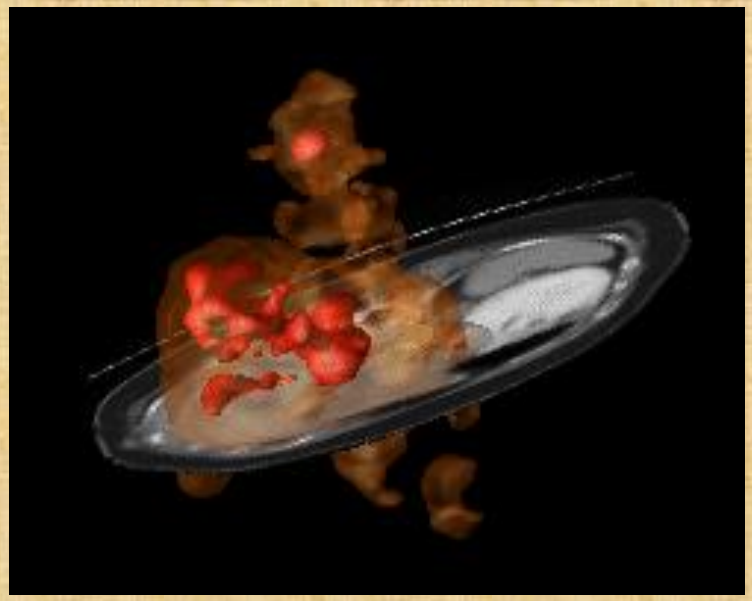
Anti-aliasing results: sharp lines and triangles



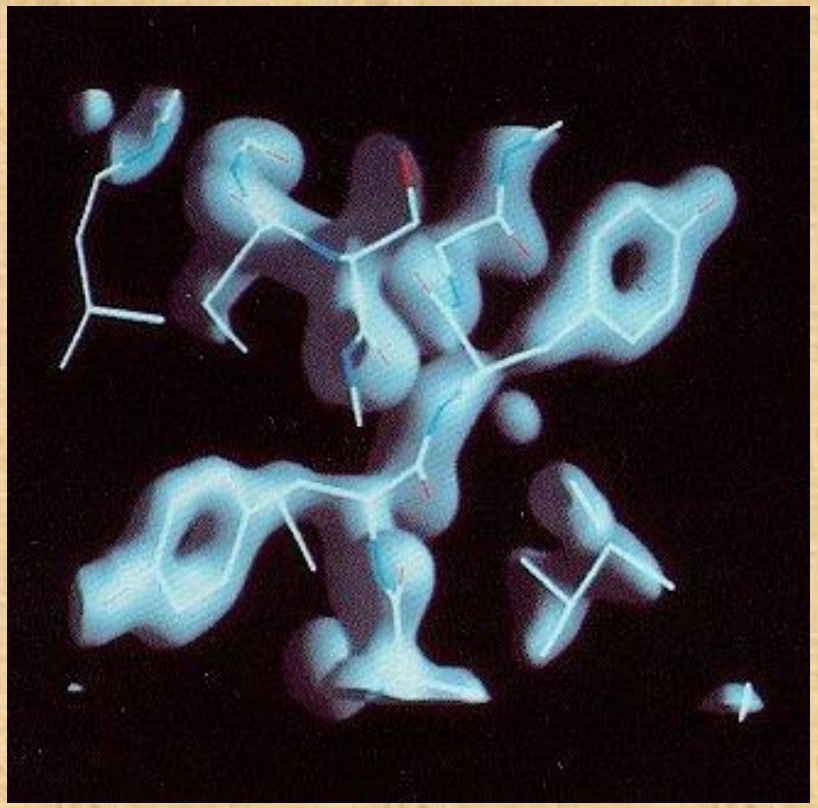
What is Volume Rendering?

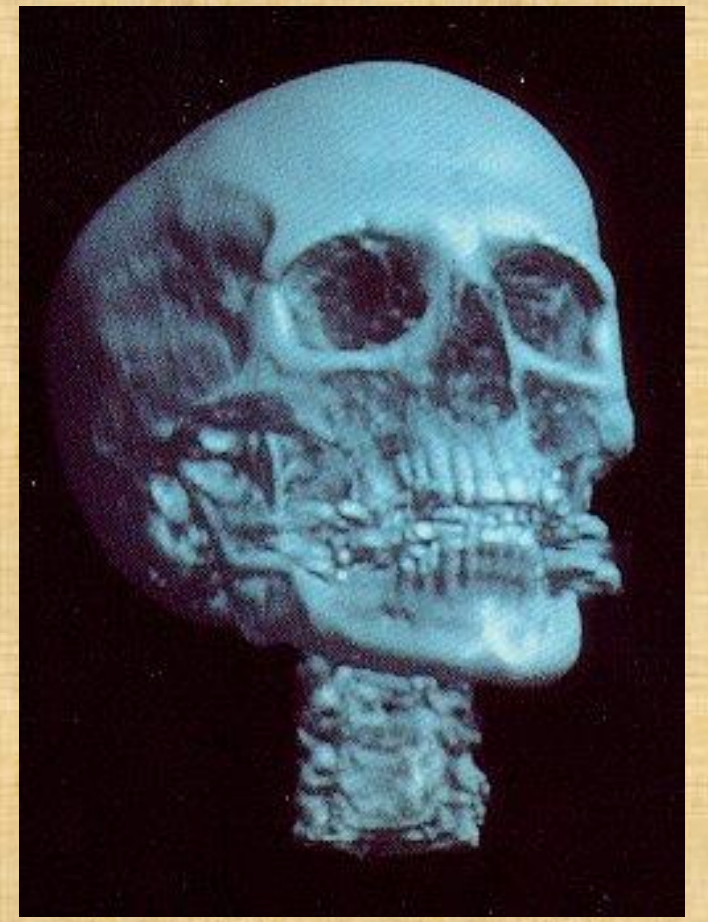
- The term *volume rendering* is used to describe techniques which allow the visualization of three-dimensional data. Volume rendering is a technique for visualizing sampled functions of three spatial dimensions by computing 2-D projections of a colored semitransparent volume.

- 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

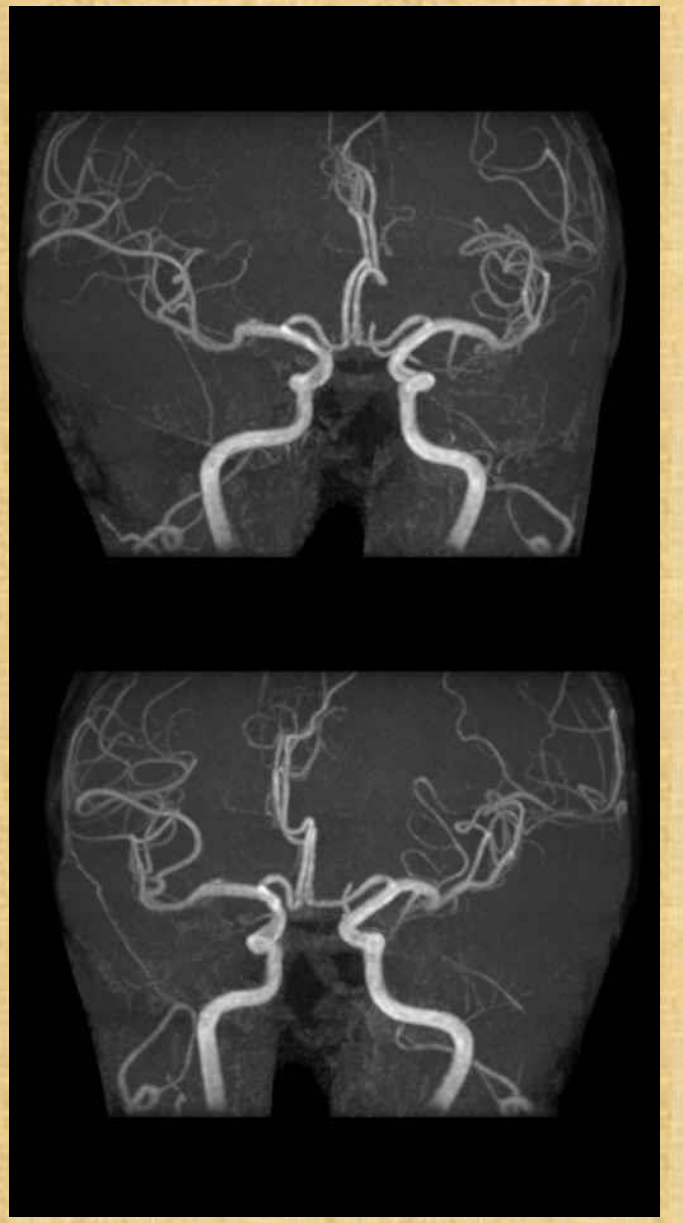


Volume Rendering: result images



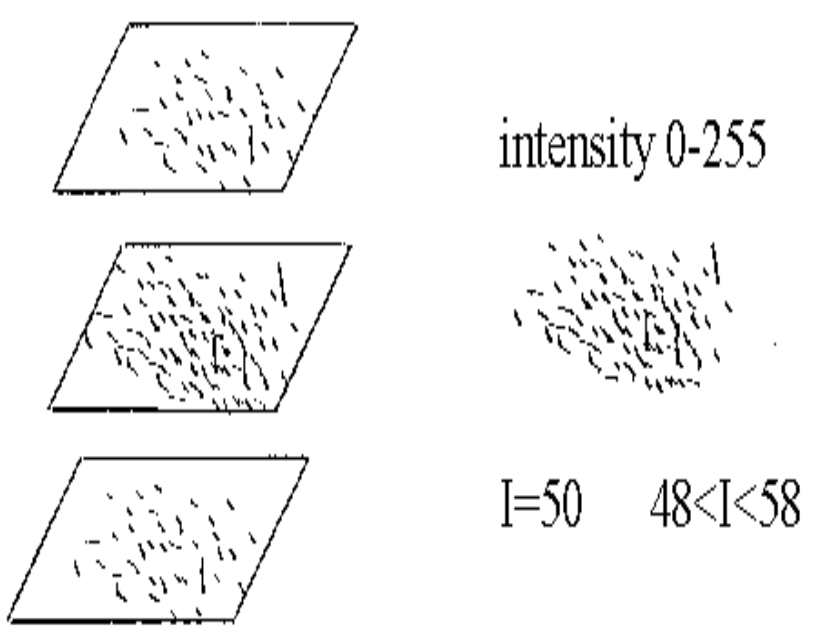


Ming’s brain vessels, MRI

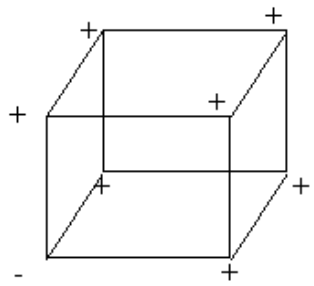


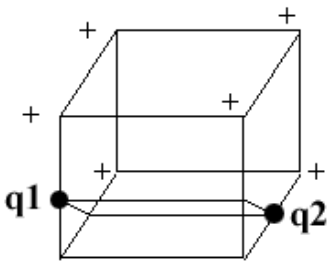
How to calculate surface normal for scalar field?

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



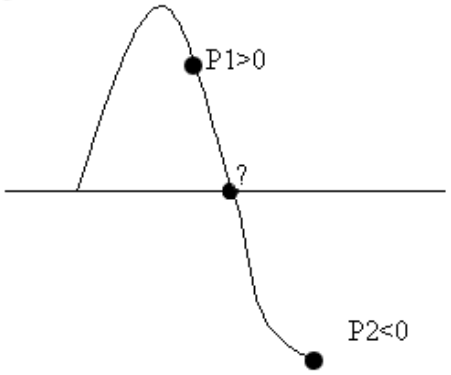
Marching cubes (squares)





+: I>50
-: I<50

2^8=256 ways reduced to 14 patterns

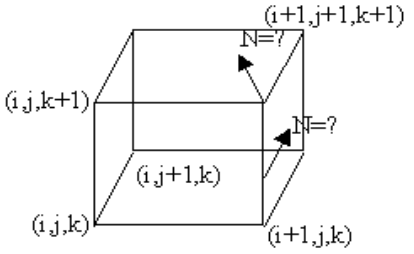


$$G(i,j,k) = \frac{D(i+1,j,,k) - D(i-1,j,k)}{\triangle x}$$

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

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

linearly intepolate



Surface normal calculation for cube corners

- $G_x(i, j, k) = (D(i+1, j, k) - D(i-1, j, k))/2$
- $G_y(I, j, k) = (D(I, j+1, k) - D(I, j-1, k))/2$
- $G_z(I, j, k) = ((D(I, j, k+1) - D(I, j, k-1))/2$

Marching Cubes Algorithm

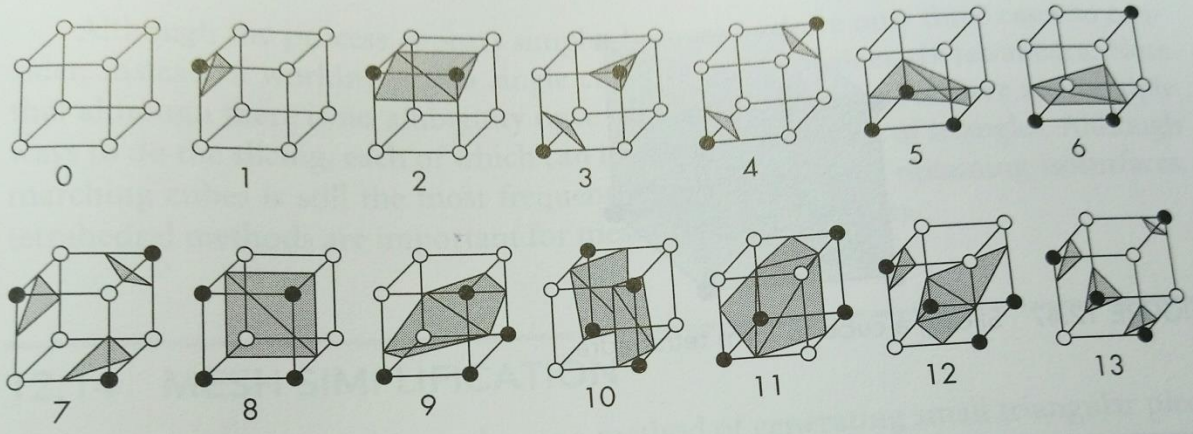


FIGURE 12.45 Tessellations for marching cubes.

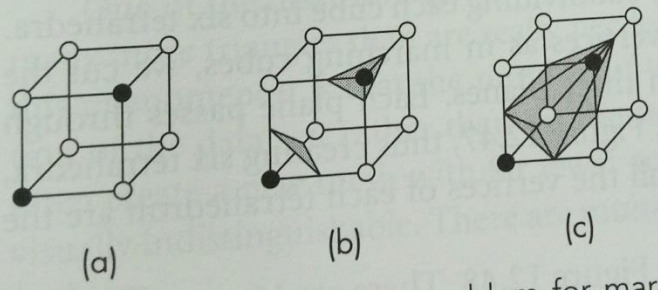
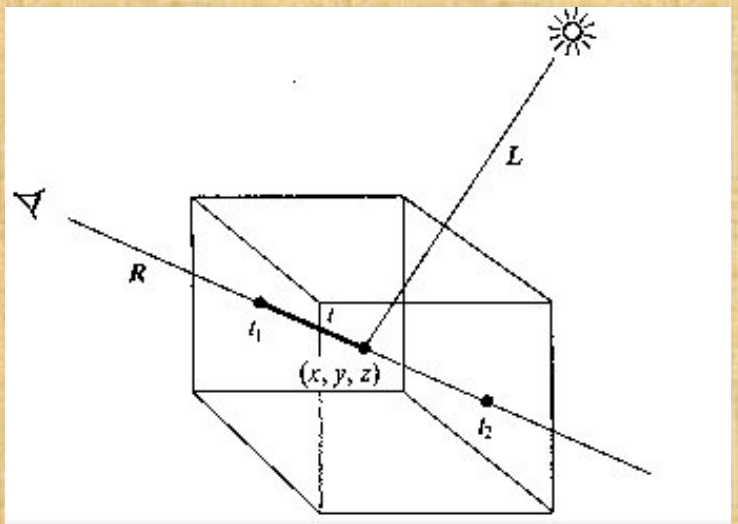


FIGURE 12.46 Ambiguity problem for marching cubes. (a) Cell. (b) One interpretation of the cell. (c) A second interpretation.

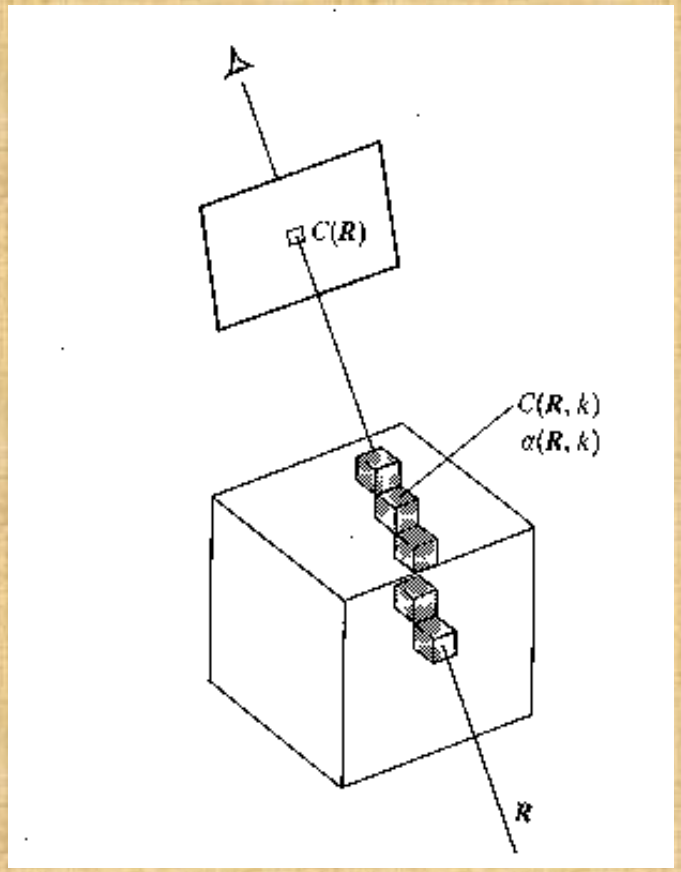
Ray casting for volume rendering

- **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)

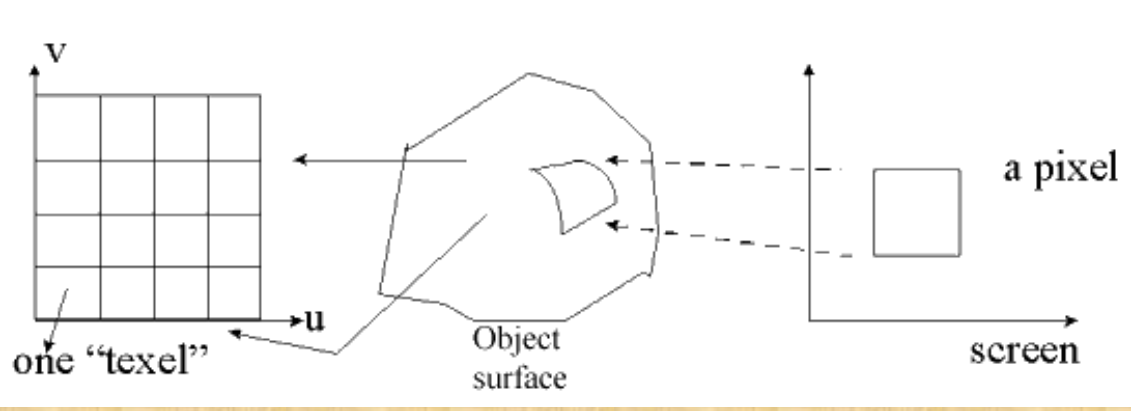
3D Modeling Methods

- 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

Driverless Car (LiDar etc.)

Texture mapping

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



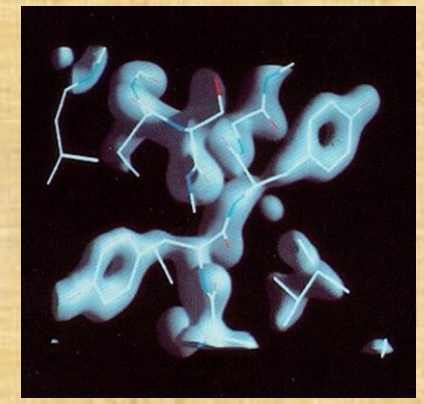
<- : direction of mapping

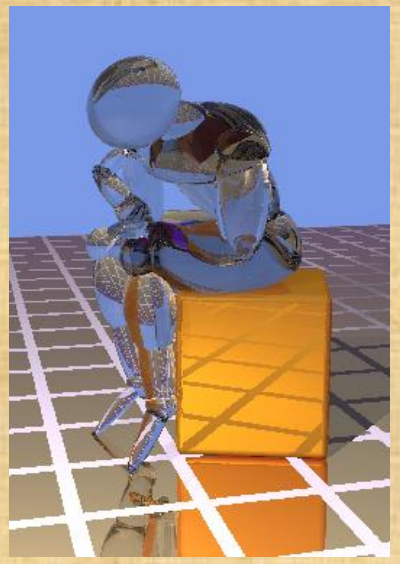
pixel value = sum of weighted texels within the four corners mapped from a pixel

3. See pictures

Curves and surfaces

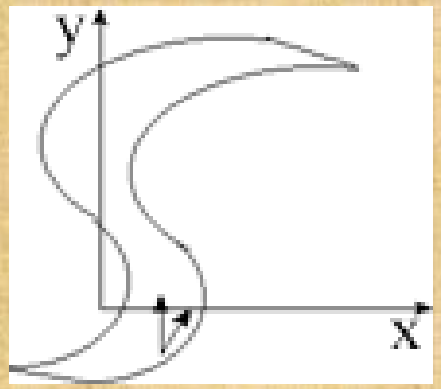
- Used in airplanes, cars, boats
- Patch (補片)





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. $f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$
- 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 (Textbook Chap 11)

- 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

(Curves) Geometry and drag equation

- In fluid dynamics, the **drag equation** is a formula used to calculate the force of drag experienced by an object due to movement through a fully enclosing fluid. The equation is:

$$F_D = \frac{1}{2}\rho u^2 C_D A$$

F_D is the drag force, which is by definition the force component in the direction of the flow velocity

ρ is the mass density of the fluid,[1]

u is the flow velocity relative to the object,

A is the reference area, and

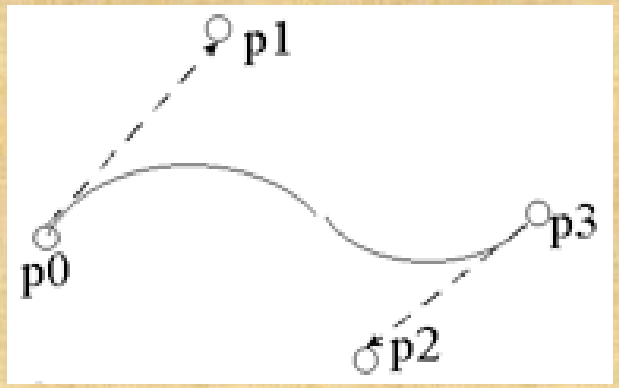
C_D is the drag coefficient – a dimensionless coefficient related to the object's geometry and taking friction and form drag, in general C_D depends on the Reynolds number.

- This equation is attributed to Lord Rayleigh, who originally used L^2 in place of A (with L being some linear dimension).[2]

Form Drag

- **Form drag** or **pressure drag** arises because of the shape of the object. The general size and shape of the body are the most important factors in form drag; bodies with a larger presented cross-section will have a higher drag than thinner bodies; sleek ("streamlined") objects have lower form drag. Form drag follows the drag equation, meaning that it increases with velocity, and thus becomes more important for high-speed aircraft.
- Form drag depends on the longitudinal section of the body. A prudent choice of body profile is essential for a low drag coefficient. Streamlines should be continuous, and separation of the boundary layer with its attendant vortices should be avoided.

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)

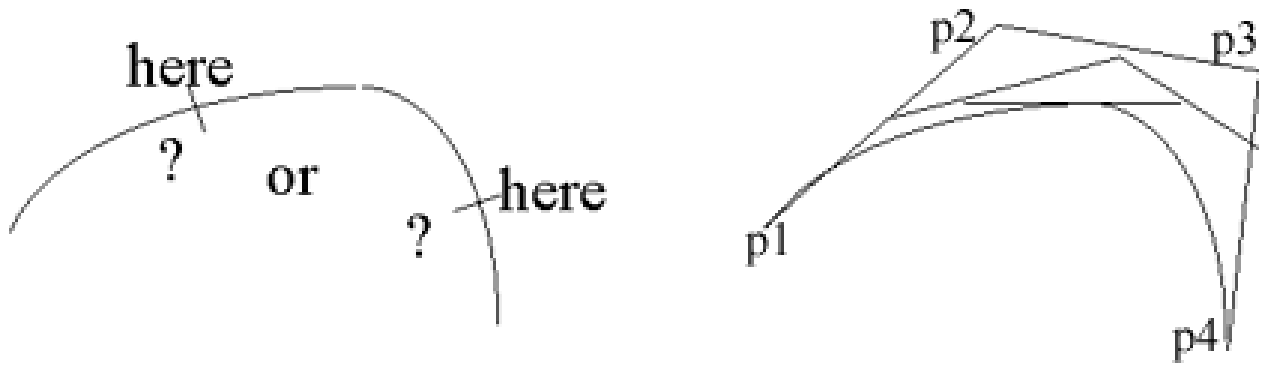
in matrix form $T*M_B*G_B$

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

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

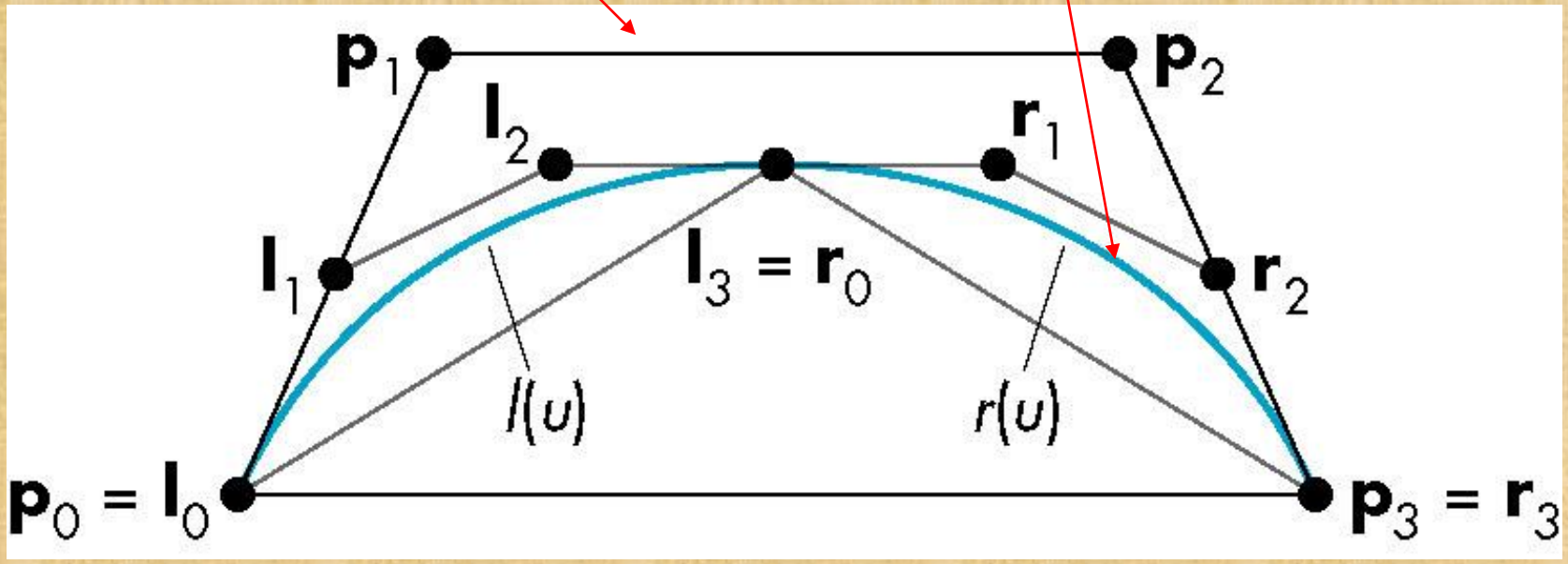
Subdividing B'ezier curves(2)

- 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

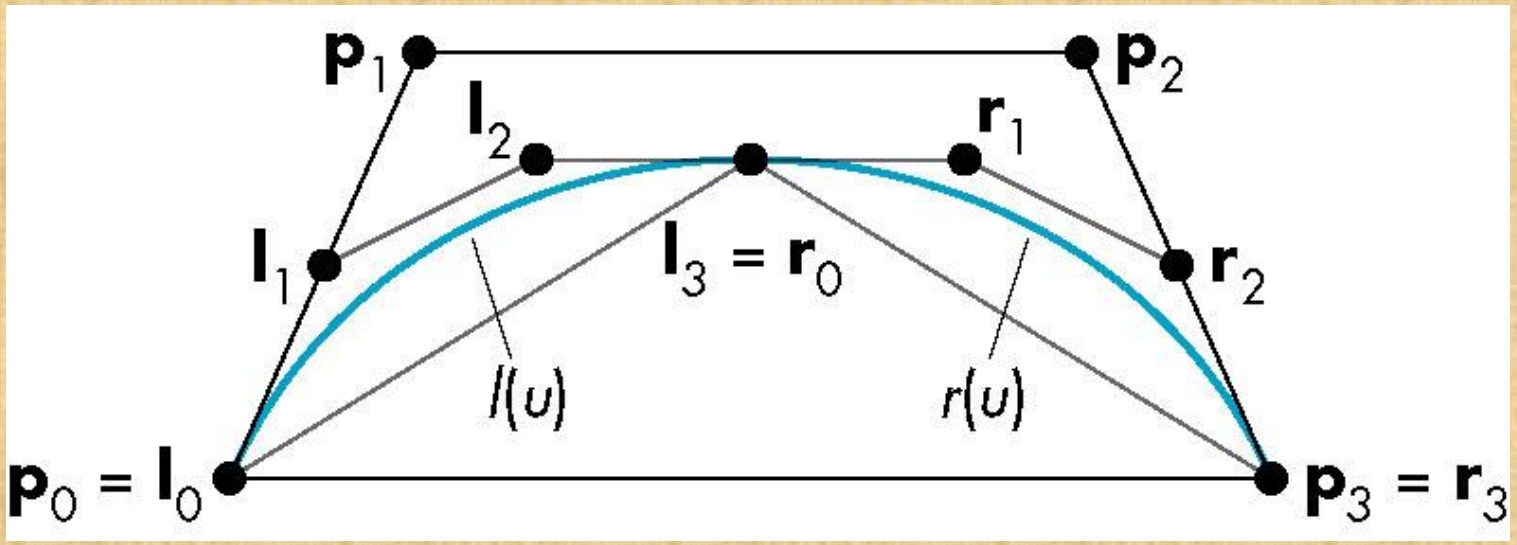
p_0, p_1, p_2, p_3 determine a cubic Bezier polynomial and its convex hull



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

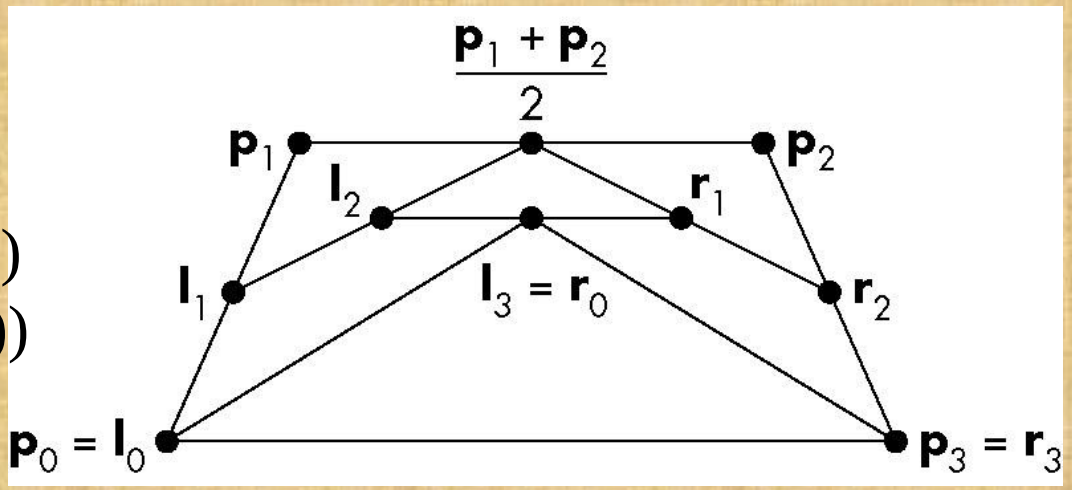
l(u) and r(u)

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

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

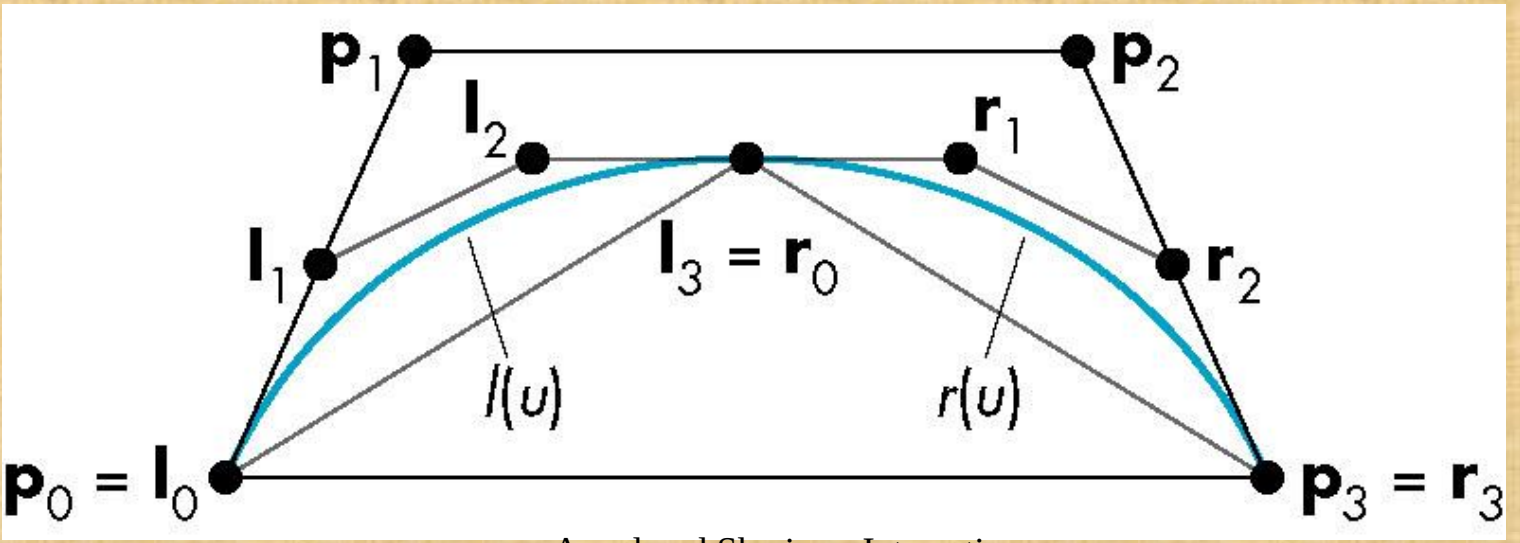


Requires only shifts and adds!

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 $\mathbf{q}$

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

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

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

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

Bezier

Dr. Pierre Bezier

Engineer, Inventor, Author, and Mathematician

Inventor of the Bezier Curves

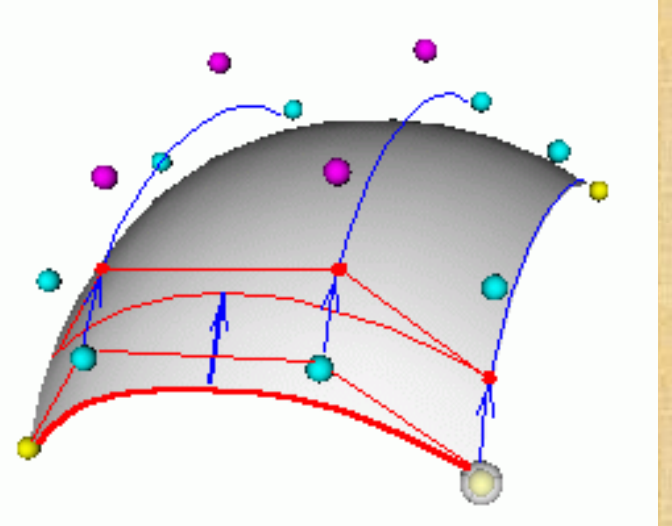
Pierre Etienne B’ezier Introduction

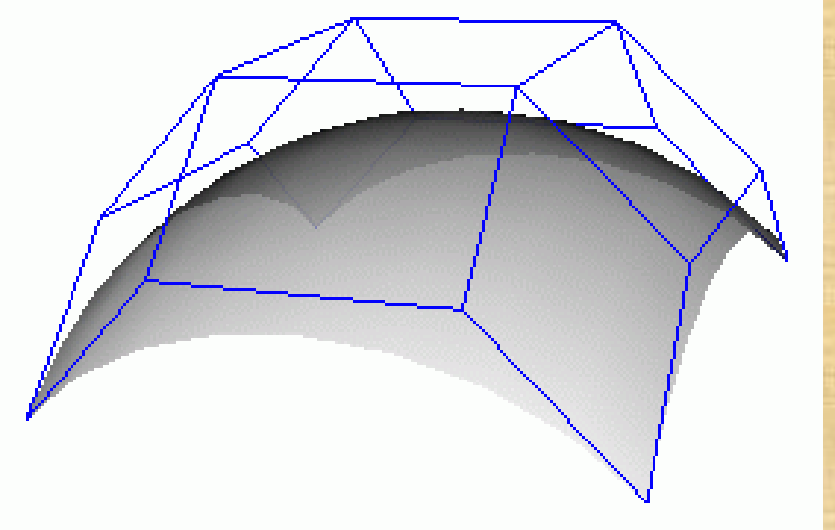
- **Pierre Etienne Bezier was born on September 1, 1910 in Paris. Son and grandson of engineers, he chose this profession too and enrolled to study mechanical engineering at the Ecole des Arts et Metiers and received his degree in 1930. In the same year he entered the Ecole Superieure d'Electricite and 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 surfaces => $S*M*G*M^T*T^T$

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

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

Parametric bicubic surfaces (cont.)

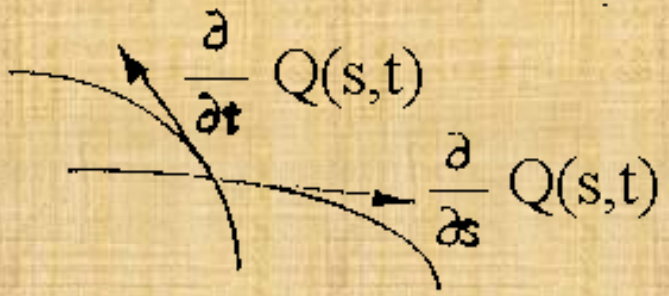
- Therefore
  - $X(s, t) = S*M*G_x*M^T*T^T$
  - $Y(s, t) = S*M*G_y*M^T*T^T$
  - $Z(s, t) = S*M*G_z*M^T*T^T$

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?

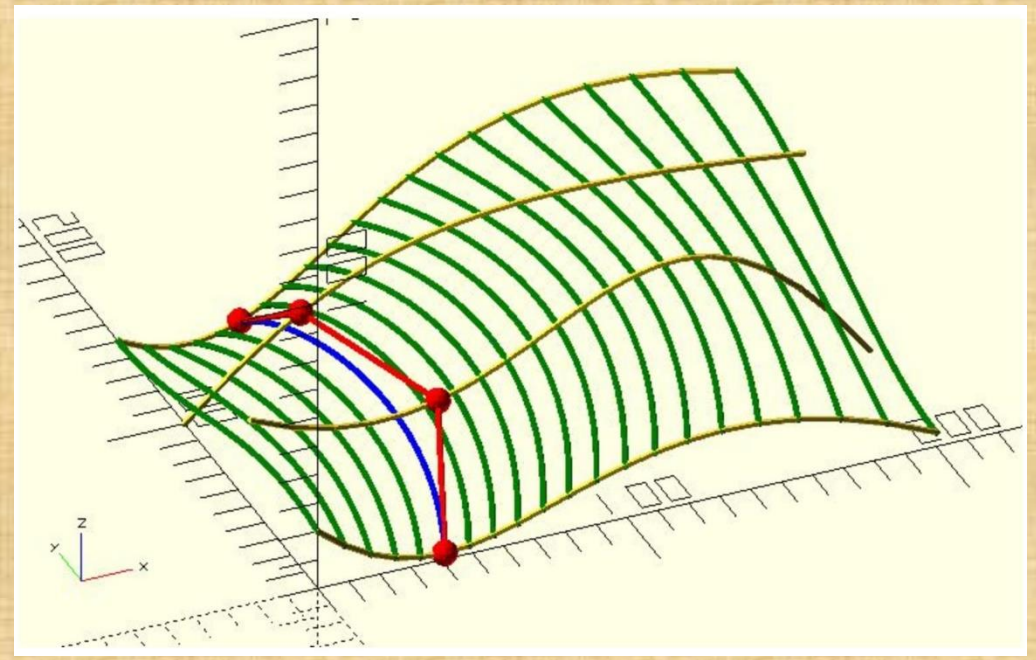


Cross-product of tangents

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

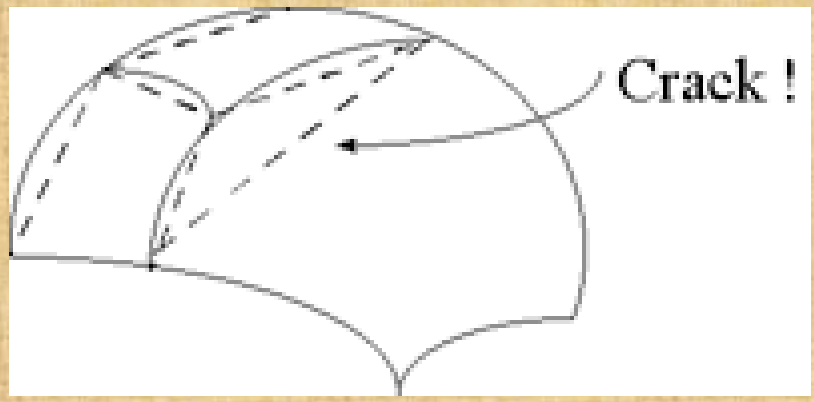
A typical Bezier Patch: with control points

```
ctrl_pts = [
[[0, 0, 20],  [60, 0, -35],        [90, 0, 60],          [200, 0, 5]],
[[0, 50, 30], [100, 60, -25],   [120, 50, 120], [200, 50, 5]],
[[0, 100, 0], [60, 120, 35],   [90, 100, 60], [200, 100, 45]],
[[0, 150, 0], [60, 150, -35], [90, 180, 60],  [200, 150, 45]] ];
```



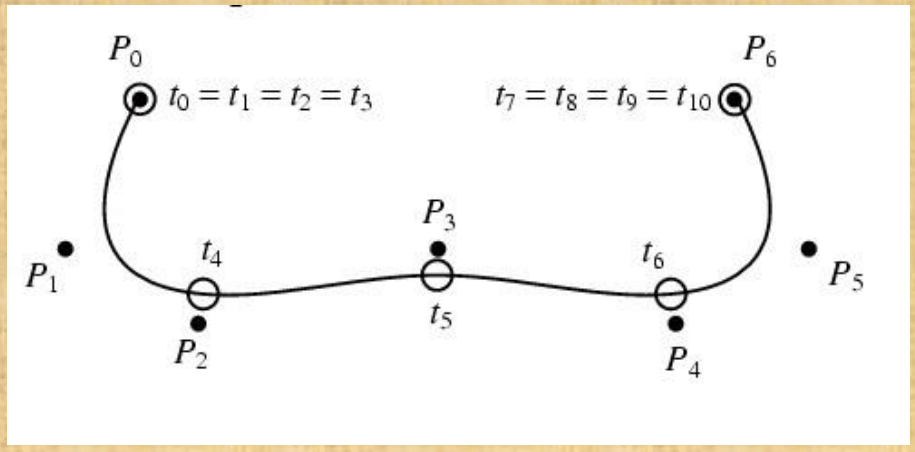
B'ezier patches display

- 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

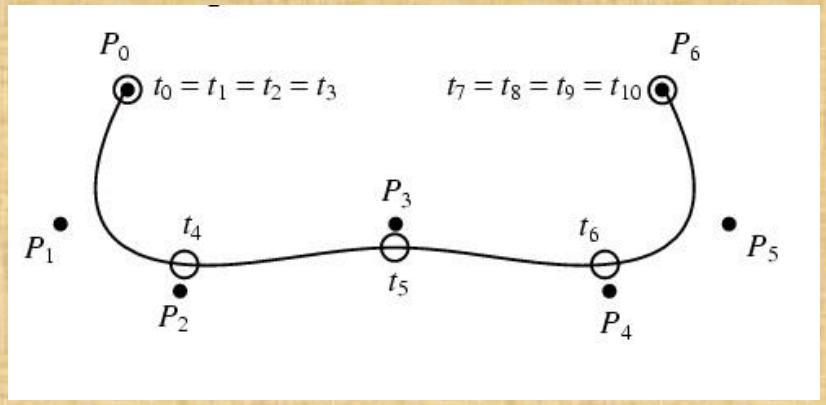
$$N_{i,0}(t) = \begin{cases} 1 \ if \ t_i \le t < t_{i+1} \ and \ t_i \le t_{i+1} \\ 0 \ otherwise \end{cases}$$

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

- Then the curve defined by

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

is a B-spline.



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 $\mathbf{p}=\mathbf{M}_B^{-1}\mathbf{M}_I\,\mathbf{q}$

About the difference between Bezier and cubic B-spline curve

1. Bezier curve will pass through the two end control points, while B spline does not.

2. Each B spline curve has a small segment defined, almost between the second and the third control points, and of course, will usually NOT pass through them.

3. Because of the limitation of 1 above, in order to have the specific starting point and end points, we let the first 4 control points the same.

Similarly, the last 4 control points are the same.

About the difference between Bezier and cubic B-spline curve II

4. The Bezier curve is C^1 continuous, since the limitation of 1 above.

5. B-spline is designed to be C^2 continuous, since it does not need to pass any control points.

6. However, one small segment of B-spline can be “simulated” by a Bezier curve, given new 4 control points, as proved below.

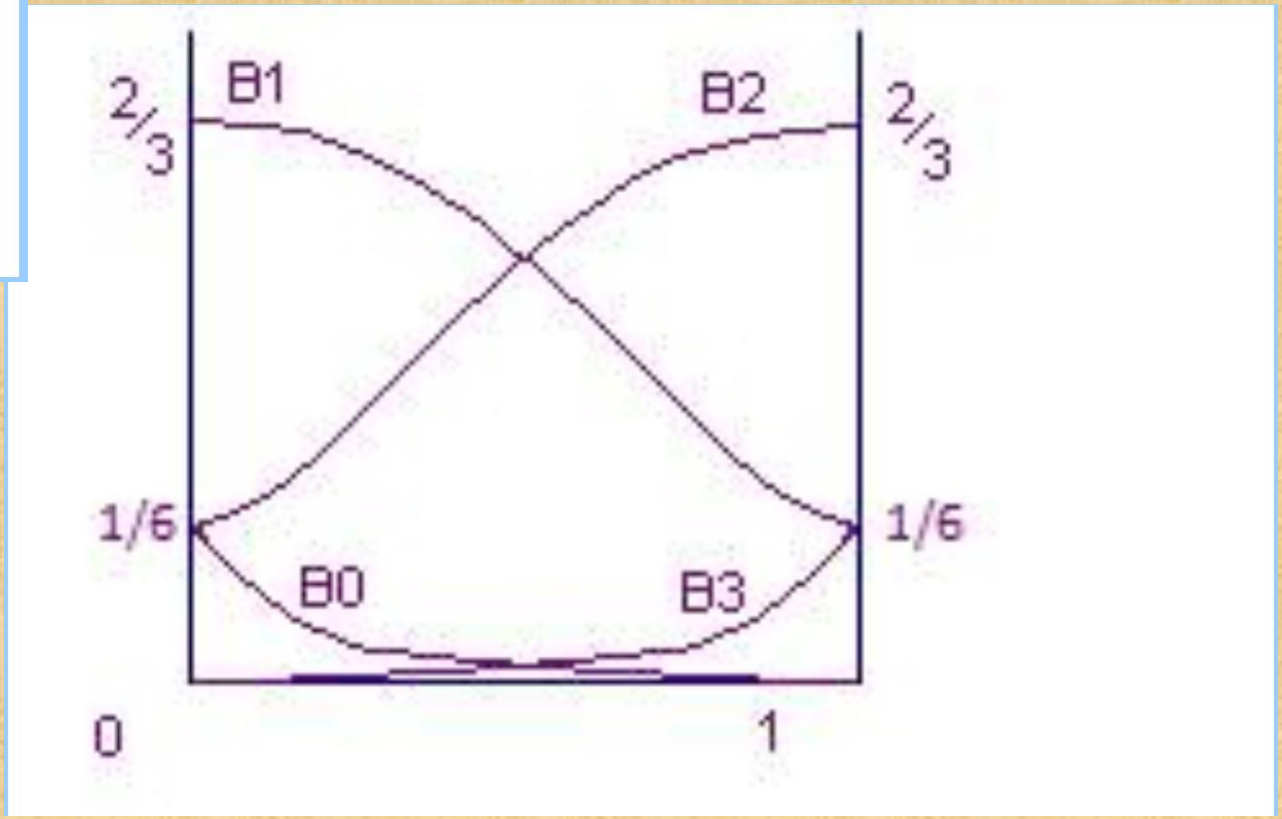
B-spline (basis spline) : for 4 control points, given B0, B1, B2, and B3 as the weighting function

$$B0(u) = \frac{(1-u)^3}{6}$$

$$B1(u) = \frac{3u^3 - 6u^2 + 4}{6}$$

$$B2(u) = \frac{-3u^3 + 3u^2 + 3u + 1}{6}$$

$$B3(u) = \frac{u^3}{6}$$



0 <= U <= 1.0

B-spline matrix: *Bs, with 4 control points Pi*

$$Q(u) = \frac{1}{6}\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix}$$

the **columns** in the matrix (multiplied by 1/6) correspond to the equations:

$B0(u) = \frac{(1-u)^3}{6} \longrightarrow \frac{(1-u)(1-u)(1-u)}{6} \longrightarrow \frac{-u^3+3u^2-3u+1}{6} \longrightarrow [-1\ 3\ -3\ 1]^T$

$B1(u) = \frac{3u^3-6u^2+4}{6} \longrightarrow [3\ -6\ 0\ 4]^T$

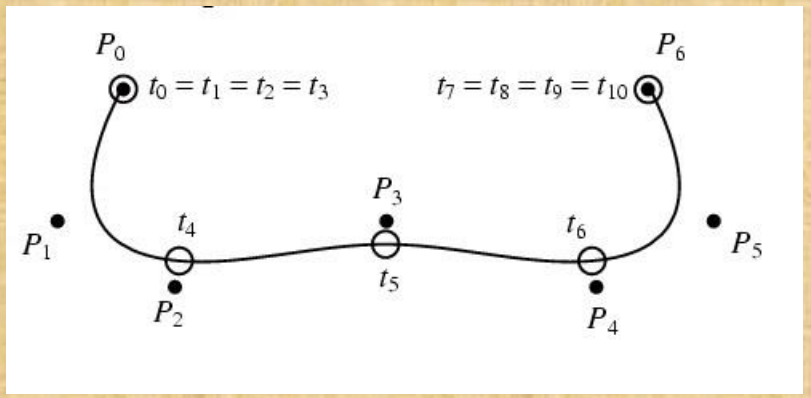
$B2(u) = \frac{-3u^3+3u^2+3u+1}{6} \longrightarrow [-3\ 3\ 3\ 1]^T$

$B3(u) = \frac{u^3}{6} \longrightarrow [1\ 0\ 0\ 0]^T$

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^2continuous, set up boundary conditions, and we can get

  if u is defined as $[1, u, u^2, u^3]$ in this order, $P(u) = u^T M_S p$



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

basis function = $Ms^T u$ = $(1/6)\ [\ (1-u)^3,\ 4-6u^2+3u^3,\ 1+3u+3u^2-3u^3,\ u^3]^T$

Every segment of a curve (B-spline, etc.) can be defined as a Bezier curve: II

$P(u) = u^T M_s p$ (p is the control point vector of B-Spline)

$P(u) = u^T M_b q$ (q is the control point vector of Bezier curve)

Therefore $q = M_b^{-1} M_s p$ (so the conversion is done)

Curve DEMO

- DEMO 1: Bezier curve

http://math.hws.edu/eck/cs424/notes2013/canvas/bezier.html (connected curve)

http://blogs.sitepointstatic.com/examples/tech/canvas-curves/bezier-curve.html (basic curve)

DEMO 2: B-spline curve demo

https://www.cs.utexas.edu/~teammco/research/bsplines/

(note: manually add more control points)

Bezier Patch demo

- DEMO 3:

  Youtube video:

https://www.youtube.com/watch?v=2tLC0oIbKS0

DEMO 4: Bezier Surfaces (patch)

http://www.ibiblio.org/e-notes/Splines/bezier3d.html

Fighter plane demo:

http://www.ibiblio.org/e-notes/webgl/deflate/yf23.html

WebGL Interactive models

- http://www.ibiblio.org/e-notes/webgl/models.htm#spline

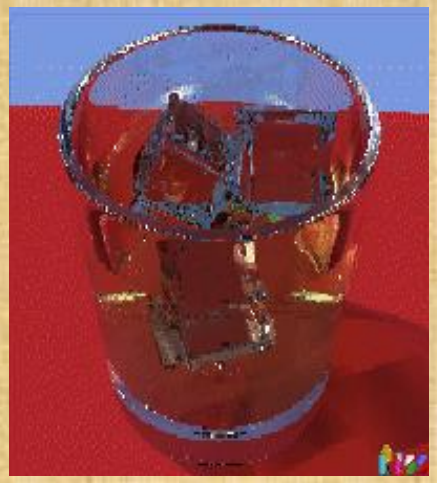
- (including animated horse, fox, eagle, shark, human elbow bones, and more static 3D models, such as bike, etc.)

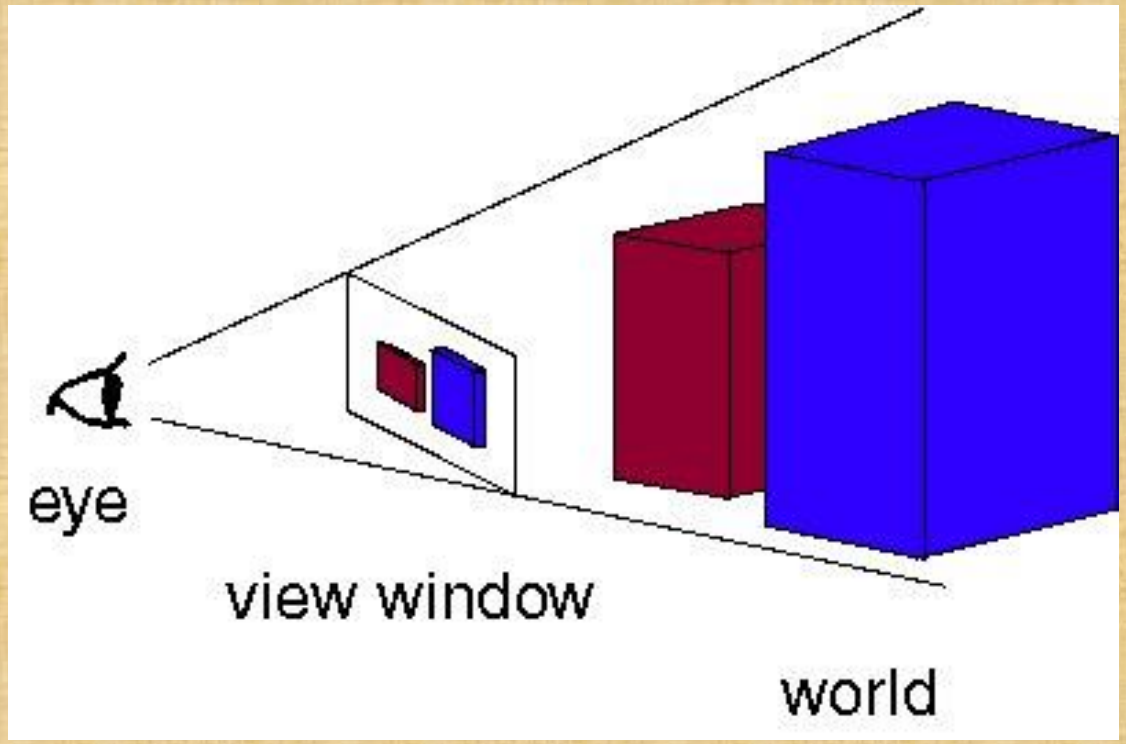
DEMO 5: more control points for B-spline curve

http://nurbscalculator.in/ (need to add more control points)(Q1: make the first three control points to be at the same position, what happens?)

Ray tracing: Turner Whitted

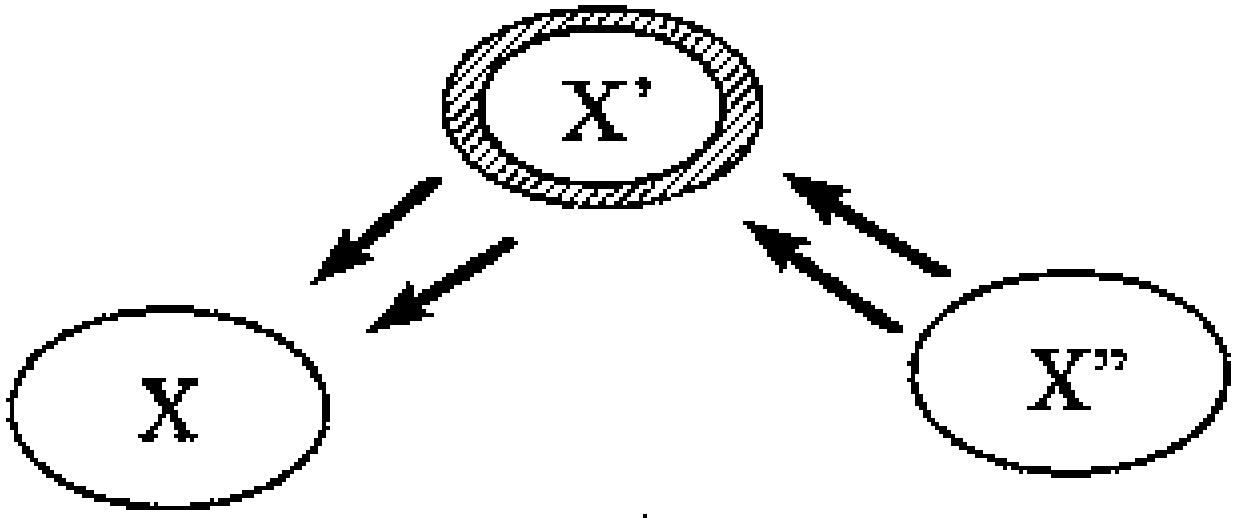
- Key to success, from light to eye or from eye to screen?





The Rendering Equation: Jim Kajiya, 1986

(渲染方程式)



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

I(x, x'): light (intensity) from patch x' to patch x

g(x,x'): visibility geometry, o: invisible, 1: visible

e(x, x'): the rate at which light is emitted from patch x' to x, when x' is an emitter.

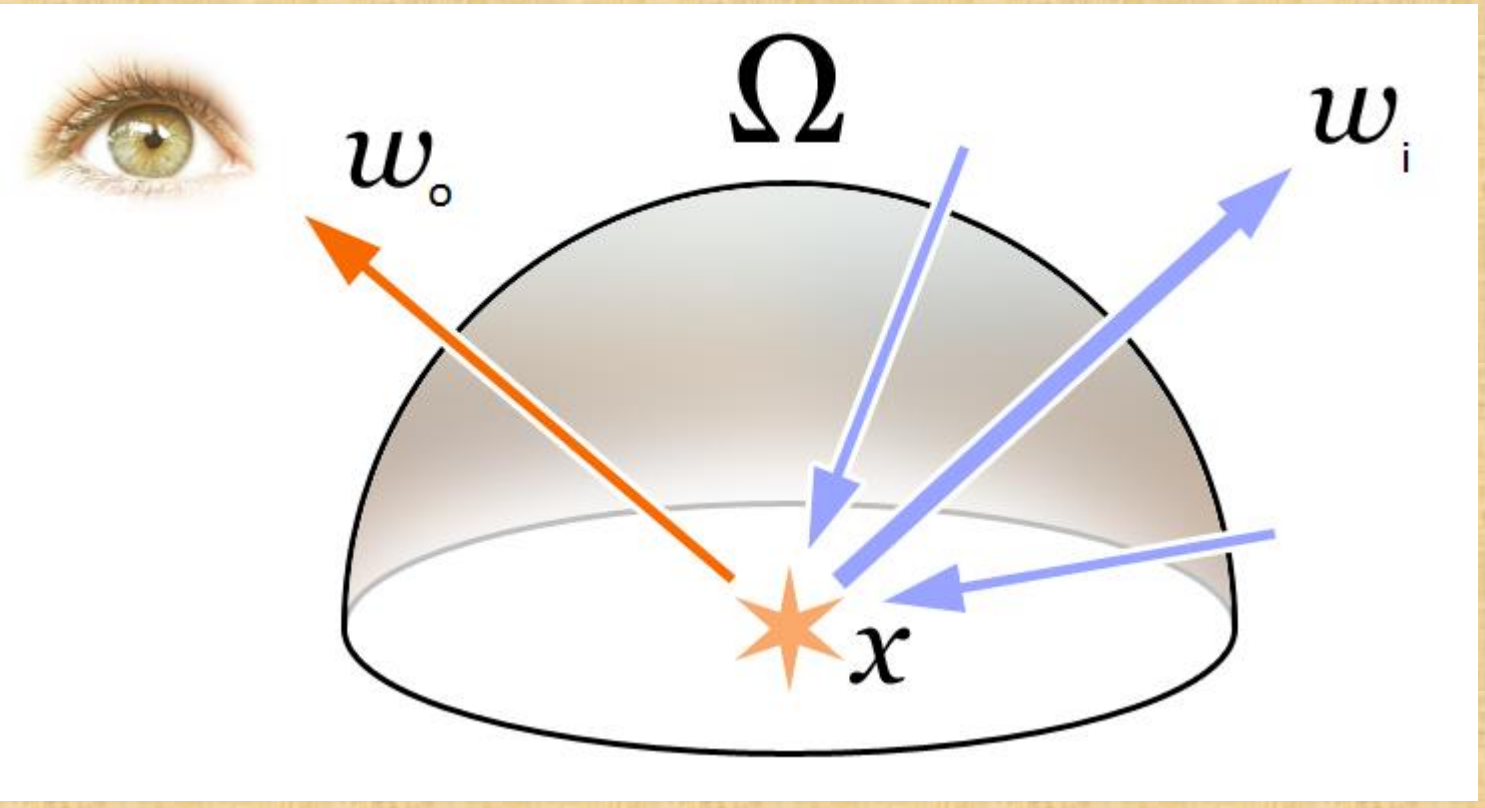
ρ (x, x', x''): patch's reflectivity,

Definitions

- where

I(x, x’): light (intensity) from patch x’ to patch x
g(x,x’): visibility geometry, o: invisible, 1: visible
e(x, x’): the rate at which light is emitted from patch x’ to x, when x’ is an emitter.
ρ (x, x’, x’’): patch’s reflectivity, light from x’’ to x’ and the ratio reflected to x

Another detailed definition of the Rendering Equation



The rendering equation describes the total amount of light emitted from a point **x** along a particular viewing direction, given a function for incoming light and a BRDF.

Explanation

1. Conservation of energy.

2. Assuming that L denotes radiance, we have that at each particular position and direction, the outgoing light (Lo) is the sum of the emitted light (Le) and the reflected light.

3. The reflected light itself is the sum from all directions of the incoming light (Li) multiplied by the surface reflection and cosine of the incident angle.

The Rendering Equation

The rendering equation may be written in the form

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_e(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)\, L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)\, (\omega_{\mathrm{i}} \cdot \mathbf{n})\, \mathrm{d}\omega_{\mathrm{i}}$$

where

- $L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$ is the total spectral radiance of wavelength λ directed outward along direction ω_{o} at time t, from a particular position $\mathbf{x}$
- $\mathbf{x}$ is the location in space
- ω_{o} is the direction of the outgoing light
- λ is a particular wavelength of light
- t is time
- $L_e(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t)$ is emitted spectral radiance
- $\int_{\Omega} \ldots \mathrm{d}\omega_{\mathrm{i}}$ is an integral over Ω
- Ω is the unit hemisphere centered around $\mathbf{n}$ containing all possible values for ω_{i}
- $f_r(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t)$ is the bidirectional reflectance distribution function, the proportion of light reflected from ω_{i} to ω_{o} at position $\mathbf{x}$, time t, and at wavelength λ
- ω_{i} is the negative direction of the incoming light
- $L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)$ is spectral radiance of wavelength λ coming inward toward $\mathbf{x}$ from direction ω_{i} at time t
- $\mathbf{n}$ is the surface normal at $\mathbf{x}$
- $\omega_{\mathrm{i}} \cdot \mathbf{n}$ is the weakening factor of outward irradiance due to incident angle, as the light flux is smeared across a surface whose area is larger than the projected area perpendicular to the ray. This is often written as $\cos\theta_i$.

Terminology

Radiance: 輻射, 光芒, light or heat that comes from something.

Illumination, 照明, light

Illuminance: 照度: 單位面積上的光通量,

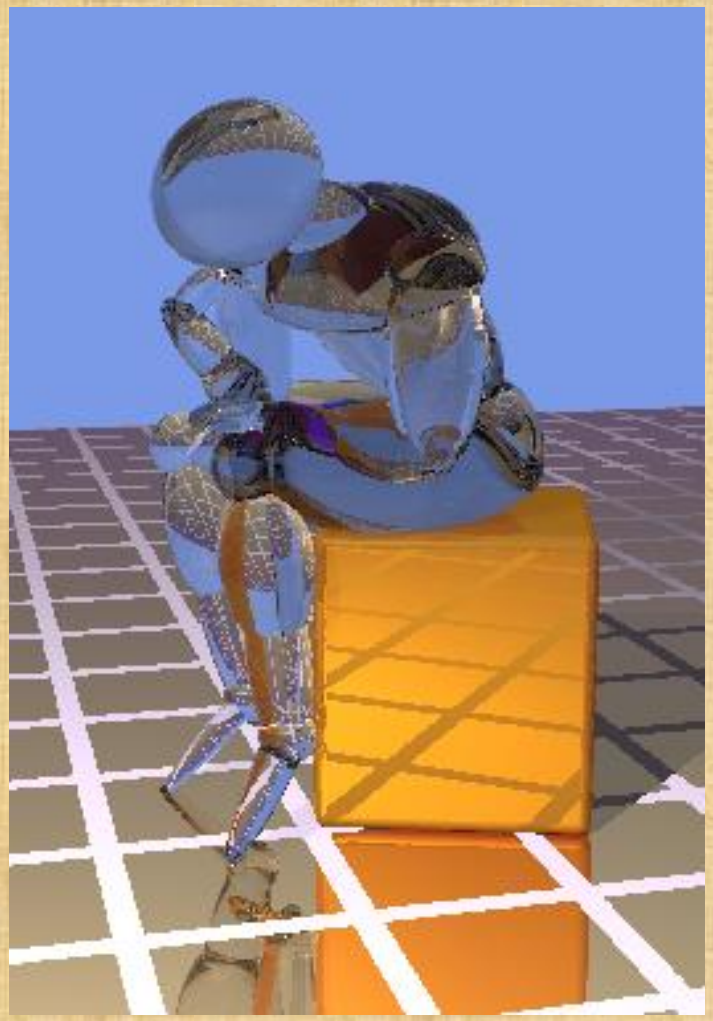
單位: LUX, 1 LUX = 1 流明/平方米,

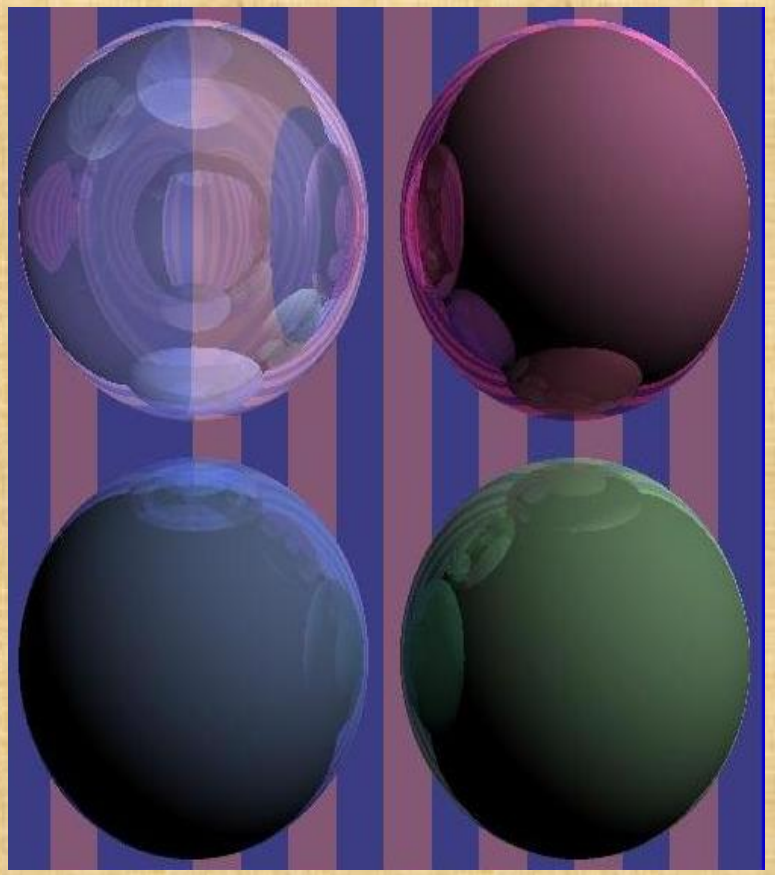
Lumen (lm) :流明, luminous flux, 1 lm = 1 cd.sr,

1 cd: Candela, 燭光 (一瓦的白燈光, 約等於一燭光),

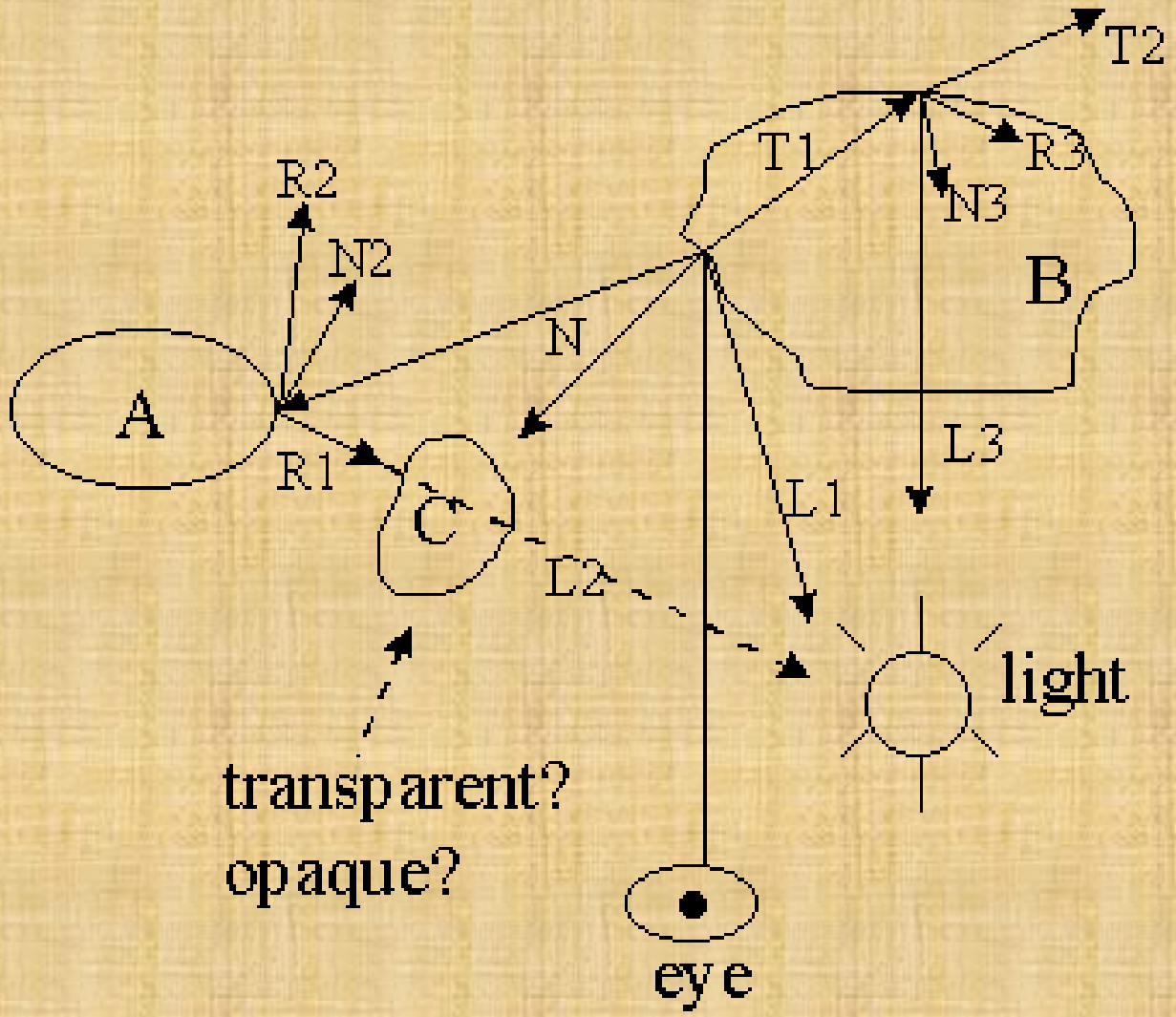
1 sr: steradian, 球面度, 立體角 (半徑 , 若張開的面積為 r^2,

則角度為 1立體角, 1 sr)





Ray tracing(1)



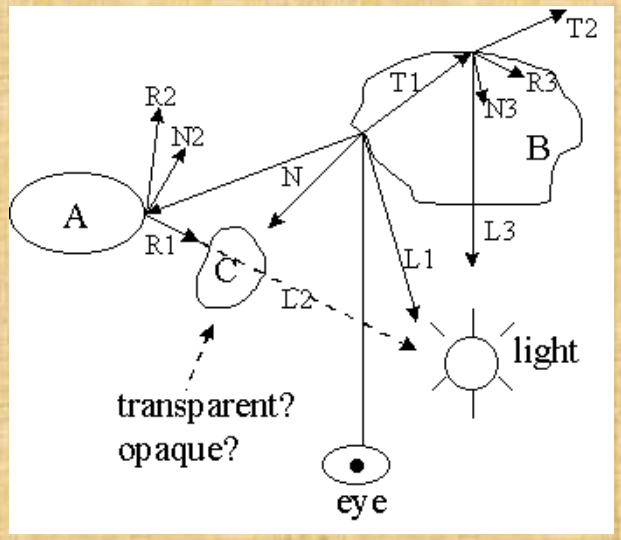
Simple recursive ray tracing

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=$f(L_1,R_1,T_1)$

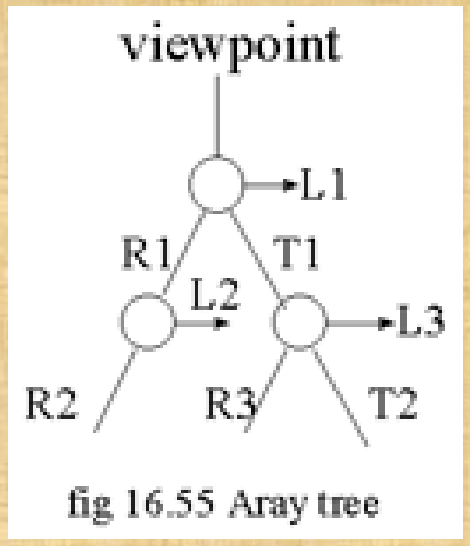
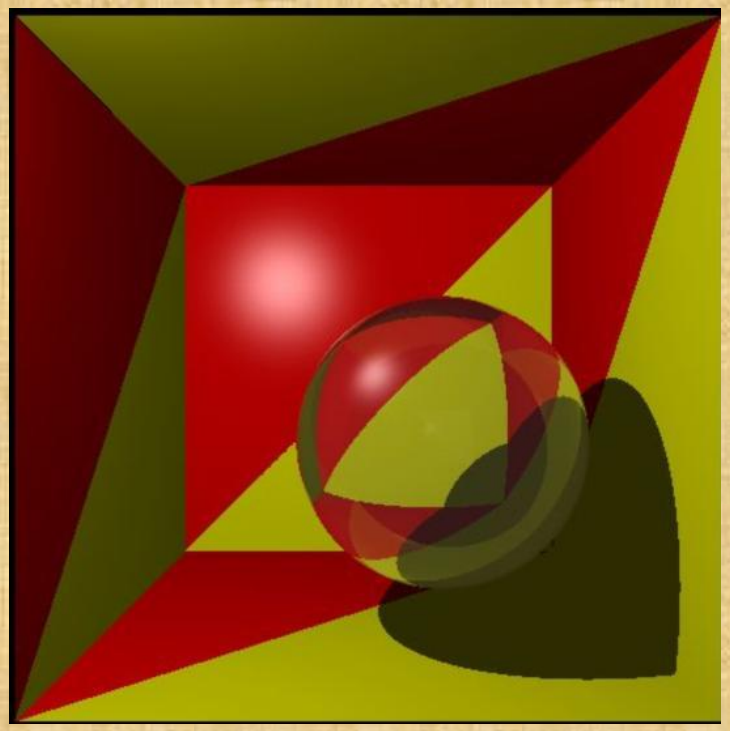
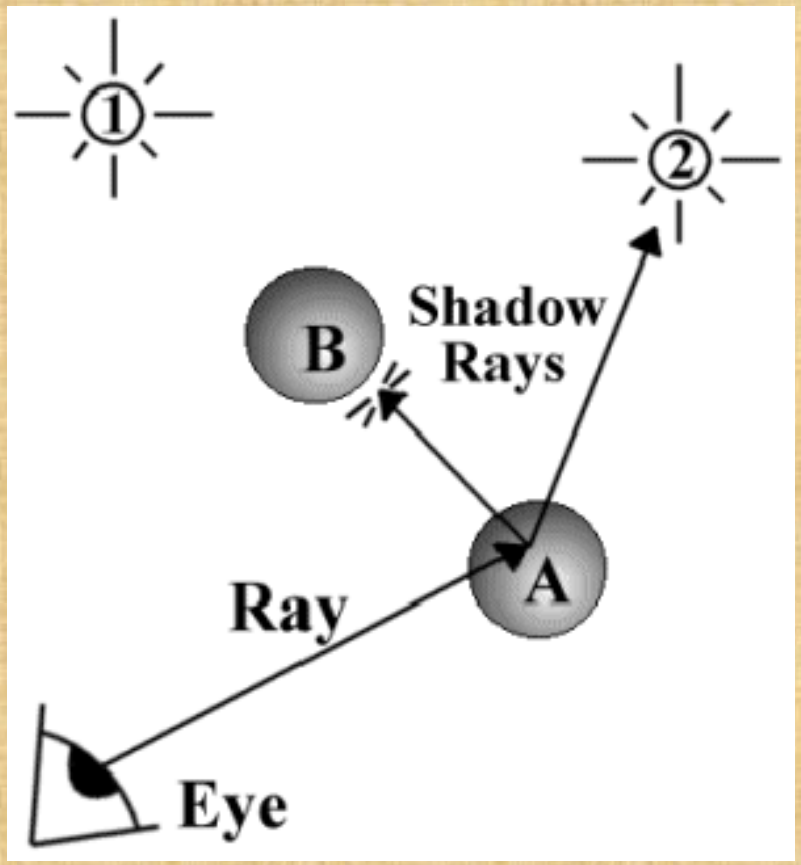


fig 16.55 Aray tree

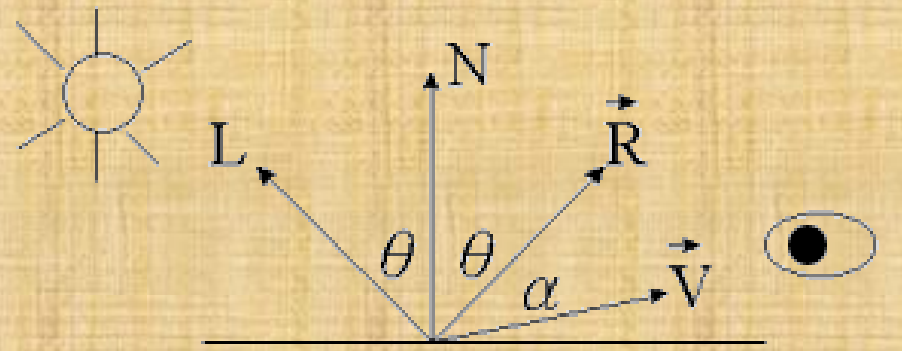
Shadow in ray tracing





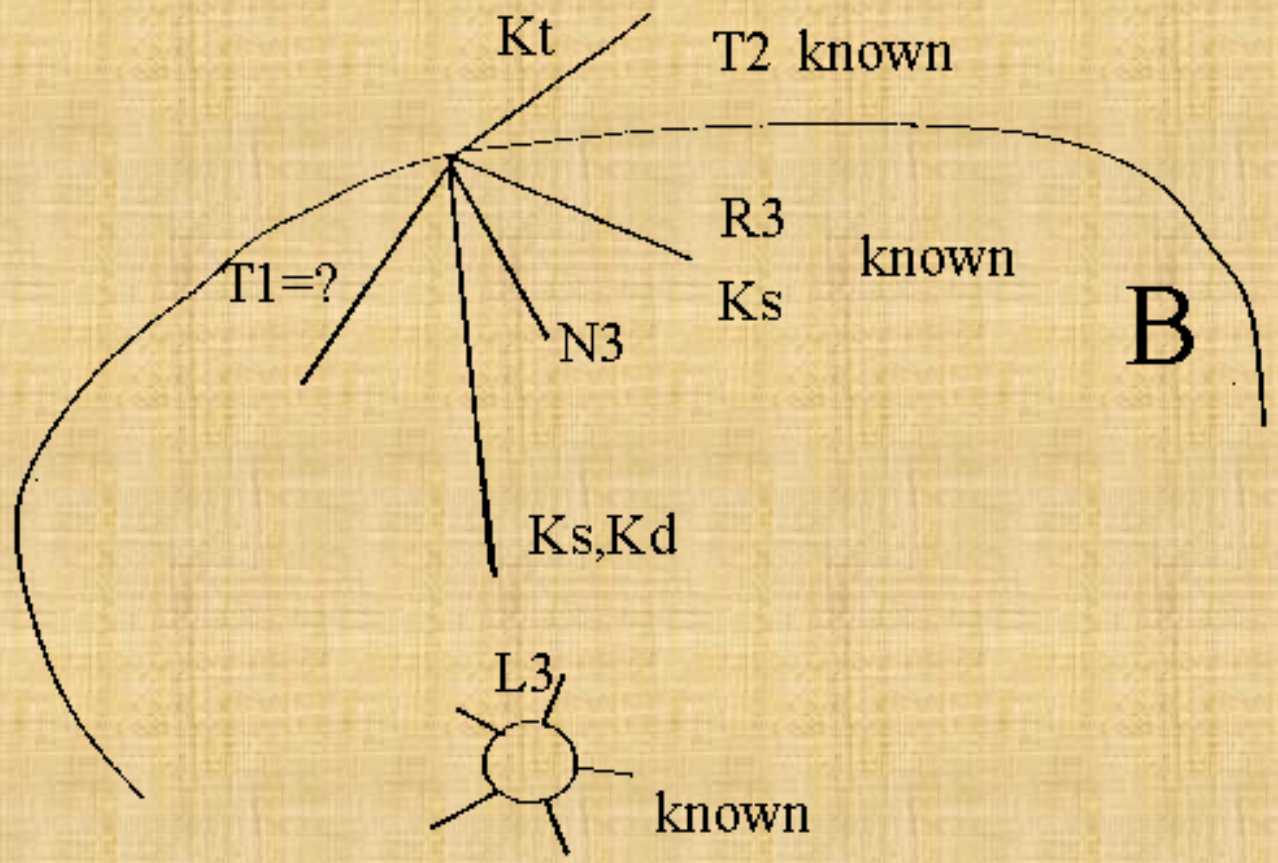
Faster: ray tracing

- halfway vector



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

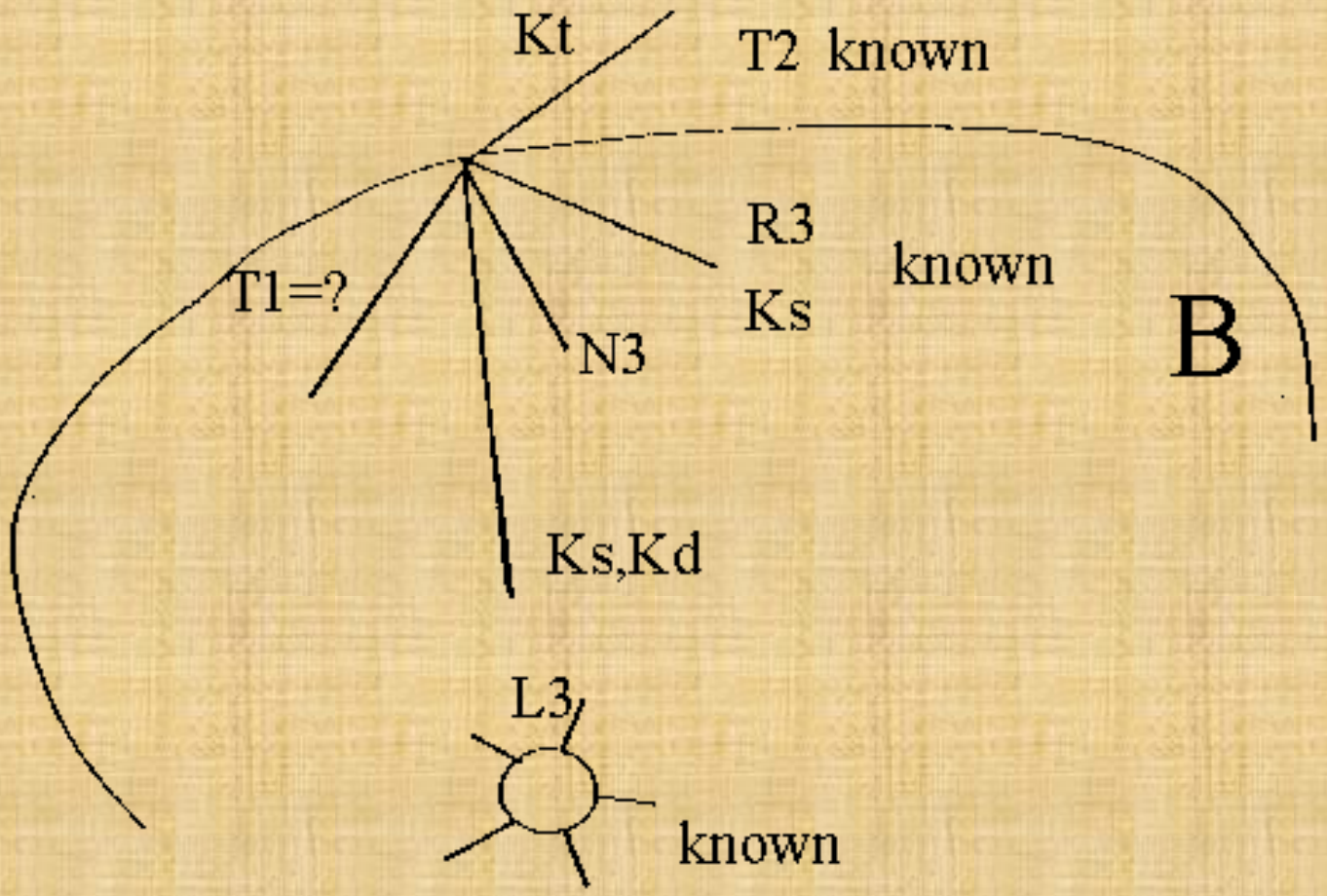
From known data to unknown



Ray Tracing Algorithm

```
Trace(ray)
    For each object in scene
        Intersect(ray, object)

    If no intersections
        return BackgroundColor

    For each light
        For the 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]

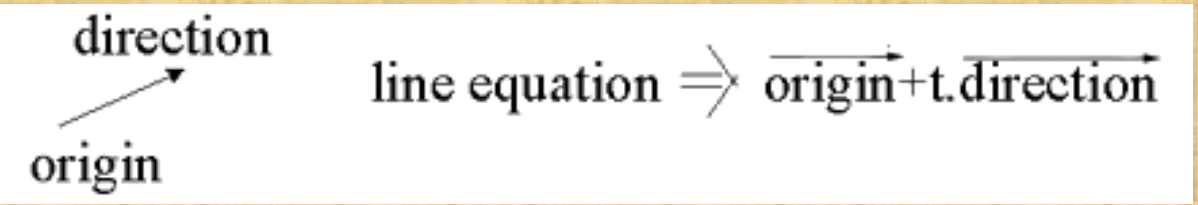
  = [2ax+2dy+2fz+2g, 2by+2dx+2ez+2h, 2cz+2ey+2fx+2j]

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

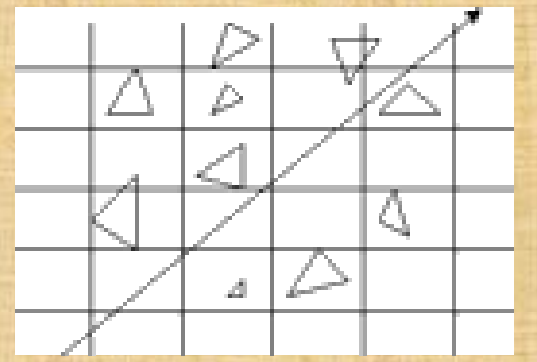
- intersection calculation:
  - let direction=(D_x, D_y, D_z), origin=(O_x, O_y, O_z)
    line ==> (x,y,z)=(O_x, O_y, O_z) + t*(D_x, D_y, D_z) **(1)**
  - quadric surface
    $f(x,y,z) = ax^2 + by^2 + cz^2 + 2dxy + 2eyz + 2fxz + 2gx + 2hy + 2jz + k = 0$ **(2)**
  - replace (x,y,z) in (2) by (1),
   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

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

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

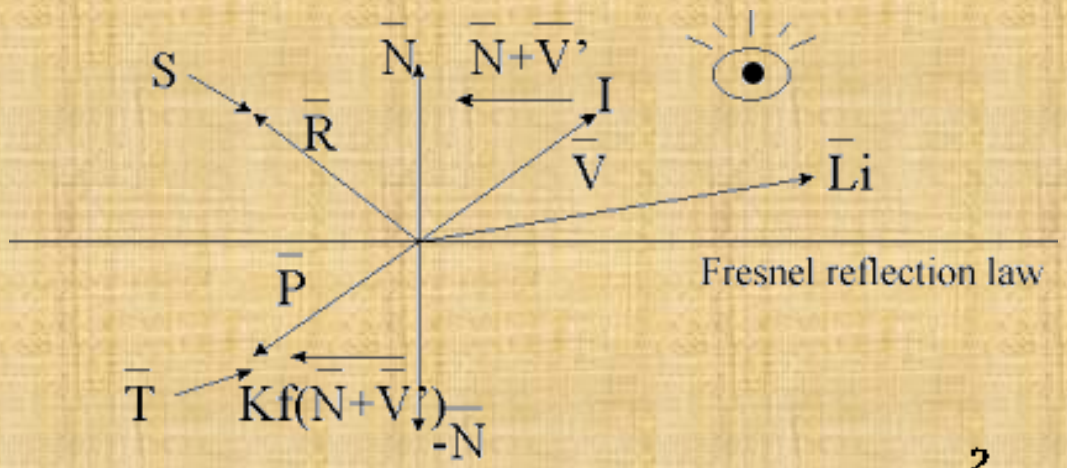
Special notice:

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

- Ultimately, this yields the following pseudo code:

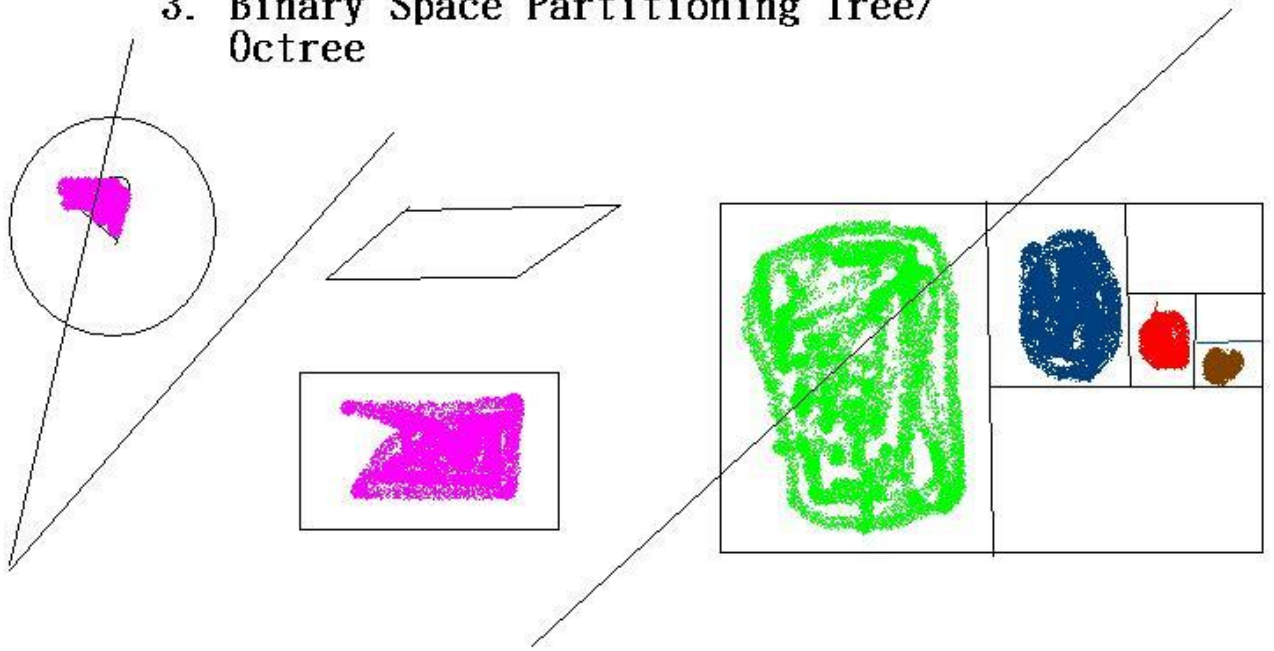
```
procedure TraceRay1(u) begin
  Ĉ(u) := 0;
  α(u) := 0;
  x1 := First(u);
  x2 := Last(u);
  U1 := ⌈Image(x1)⌉;
  U2 := ⌊Image(x2)⌋;
  {Loop through all samples falling within data}
  for U := U1 to U2 do begin
    x := Object(U);
    {If sample opacity > 0,}
    {then resample color and composite 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 TraceRay1.
```

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

Ray-object intersection acceleration

Ray Object Intersection Acceleration Methods:

1. Bounding Sphere
2. Bounding Box
3. Binary Space Partitioning Tree/ Octree



More about Ray-Tracing: Photon Mapping

- **Photon mapping** is a two-pass global illumination rendering algorithm developed by Henrik Wann Jensen between 1995 and 2001[1] that approximately solves the rendering equation for integrating light radiance at a given point in space. Rays from the light source (like photons) and rays from the camera are traced independently until some termination criterion is met, then they are connected in a second step to produce a radiance value.
- Progressive photon mapping (PPM) starts with ray tracing and then adds more and more photon mapping passes to provide a progressively more accurate render.

Photon Mapping: result

Global illumination using photon maps, EuroRendering 1996, by Henrik Wann Jensen

More about Ray-Tracing: Photon Mapping

Photon: A Modular, Research-Oriented Rendering System, Siggraph 2019 poster

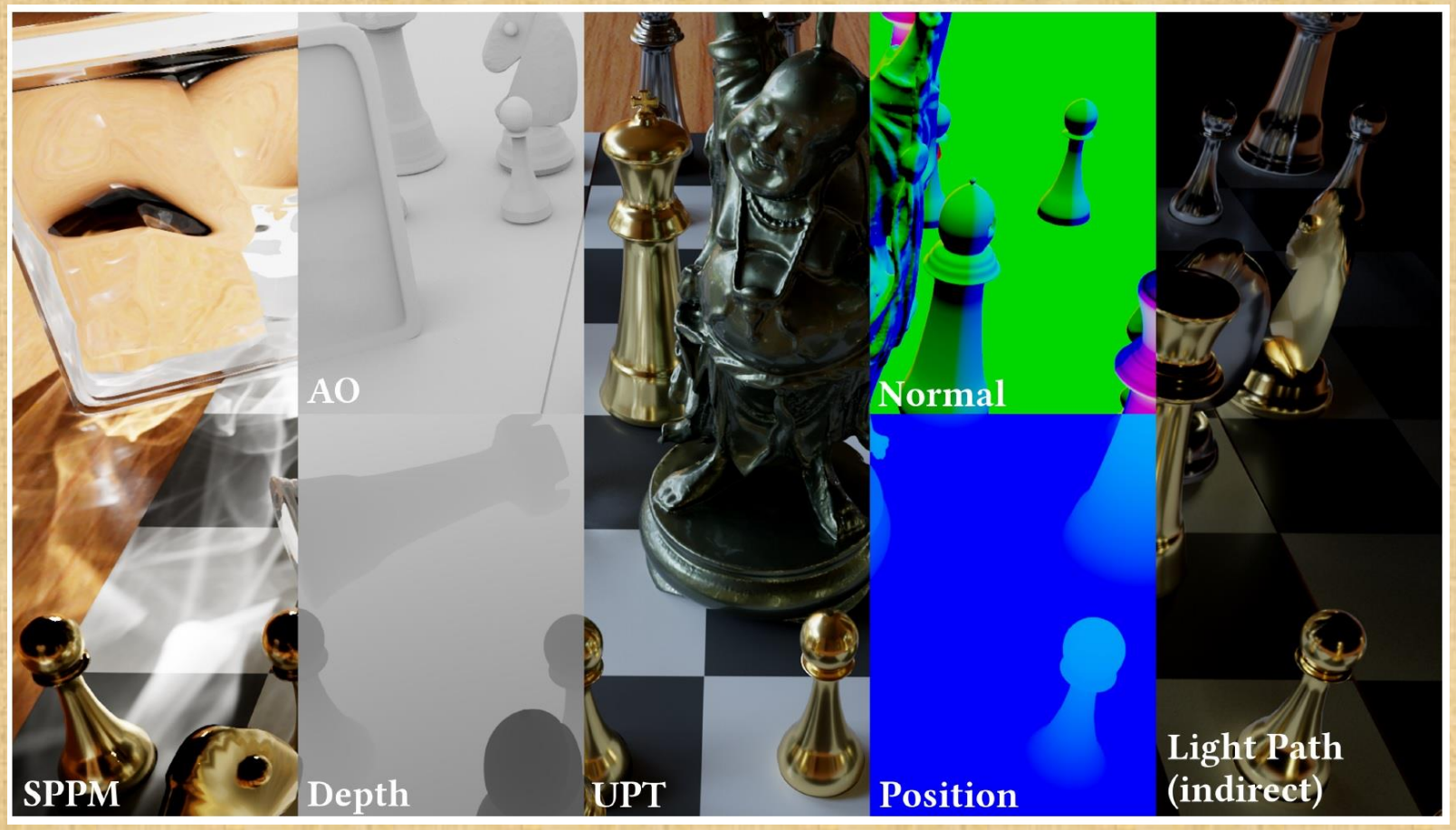
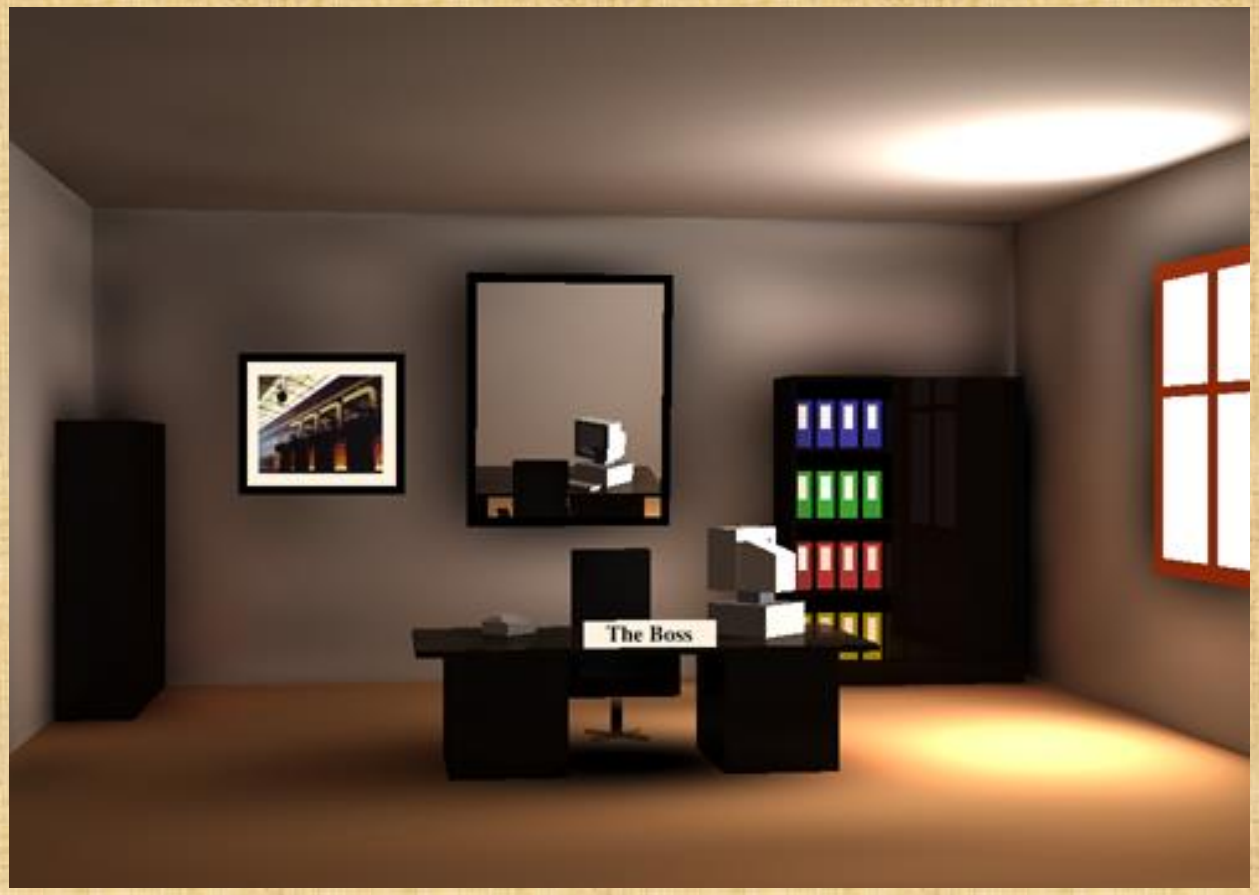


Figure 1: Several visualization techniques implemented using the building blocks provided by our system. In the center, the buddha features Belcour's BSDF model [Belcour 2018], while the left one, the glass brick induces caustics that can be rendered efficiently using particle tracing methods.

What is still missing in ray-traced images?

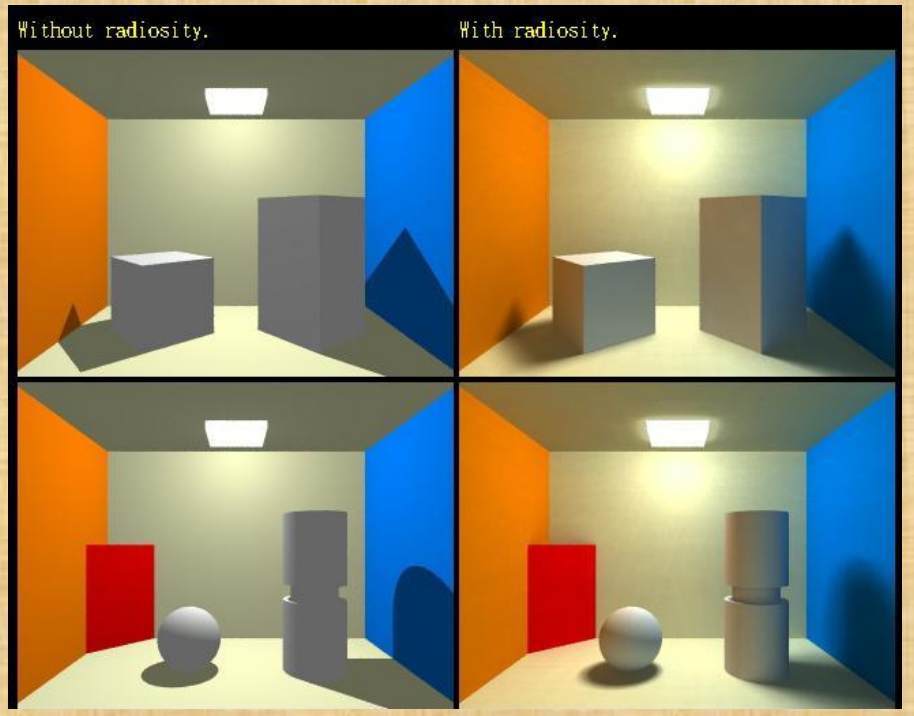
- Diffuse to diffuse reflection?

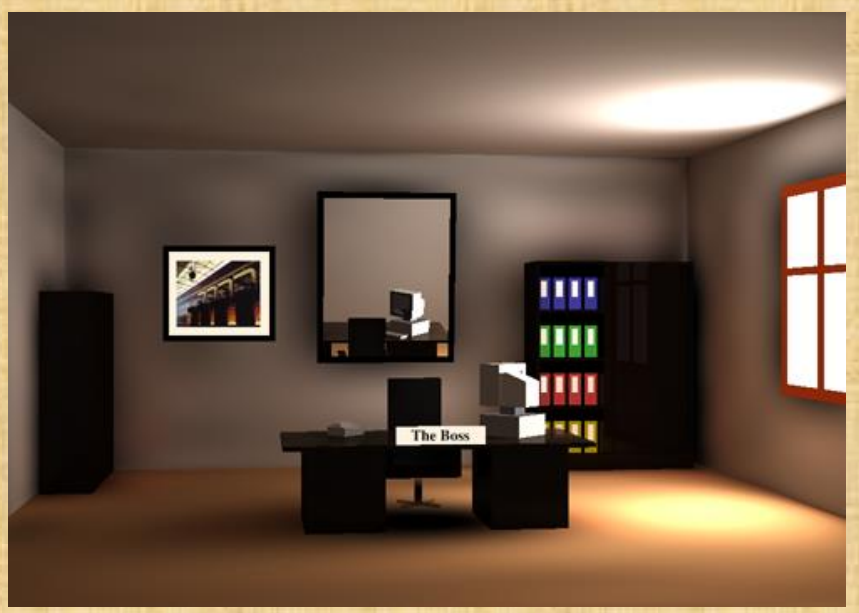


Radiosity (熱輻射法)

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'**

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

occlusion = 0 or $1/d^2$

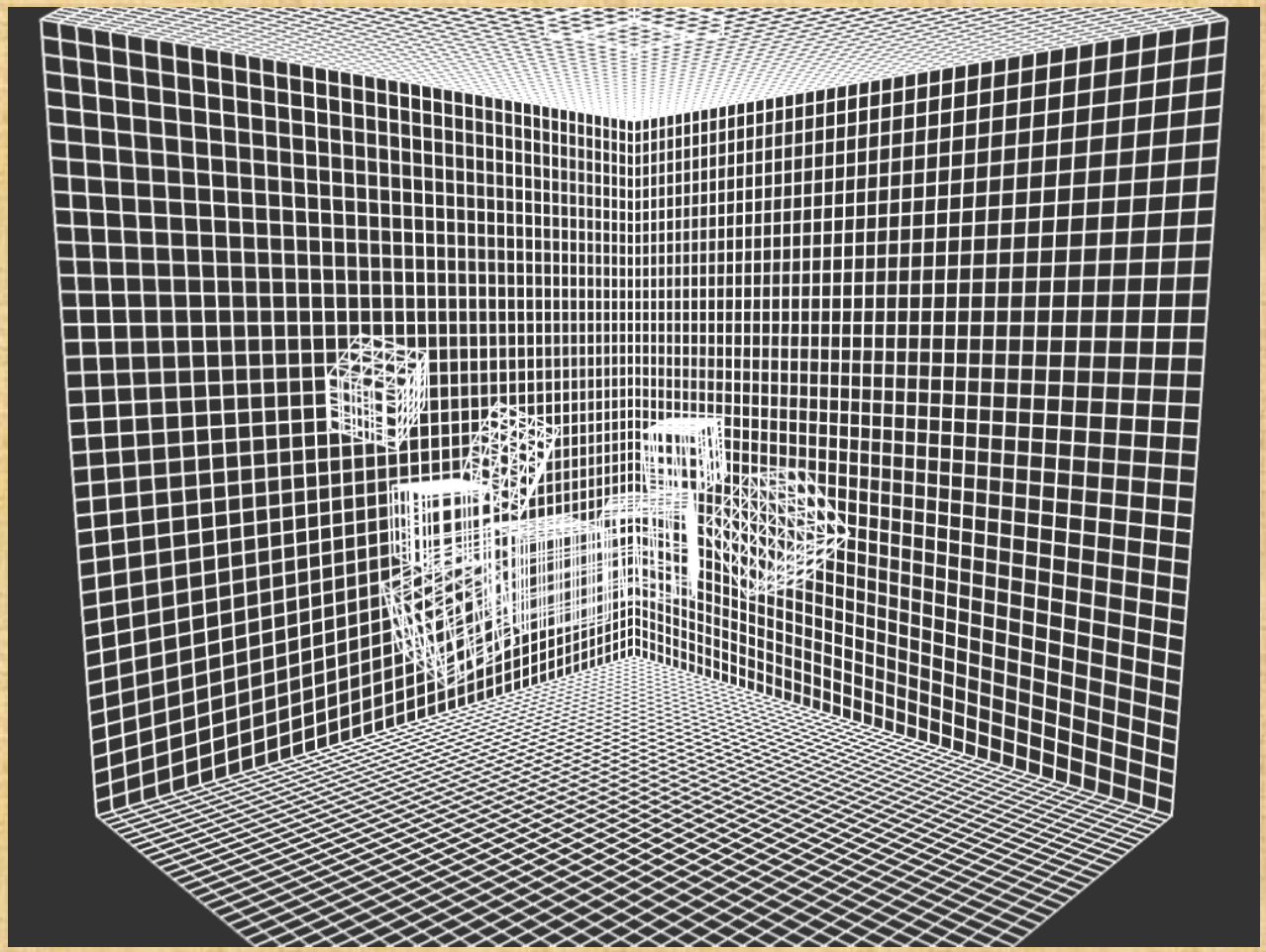
emission from **p'** to **p**

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

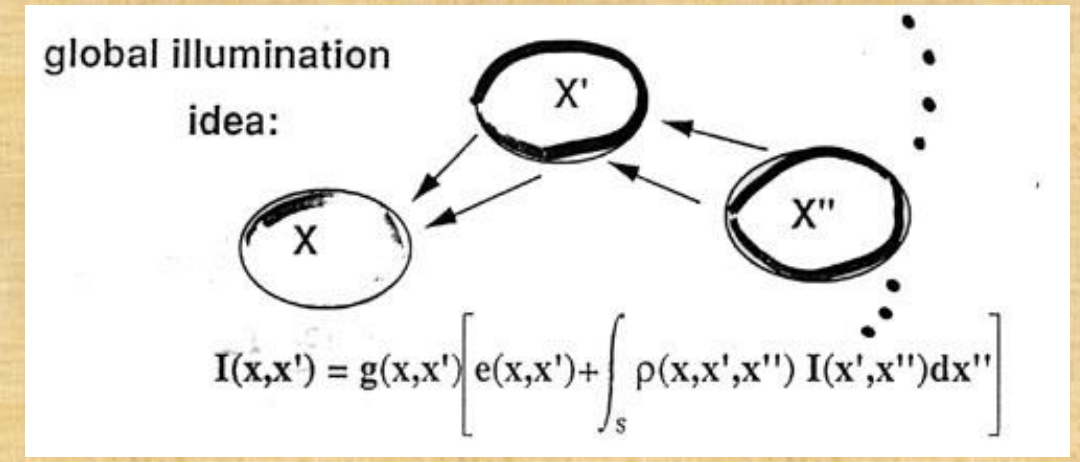
Radiosity

- 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

Patches

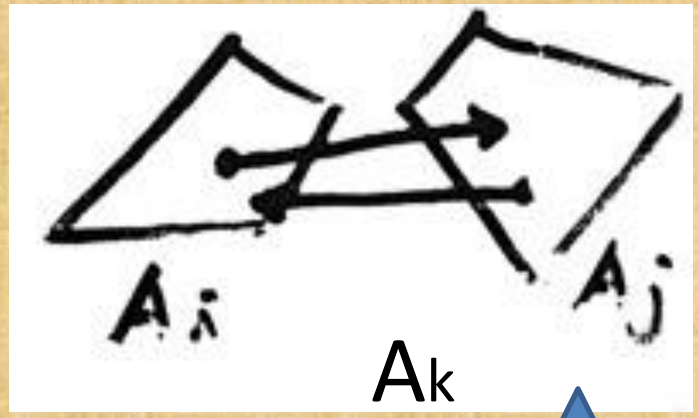


Radiosity



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

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



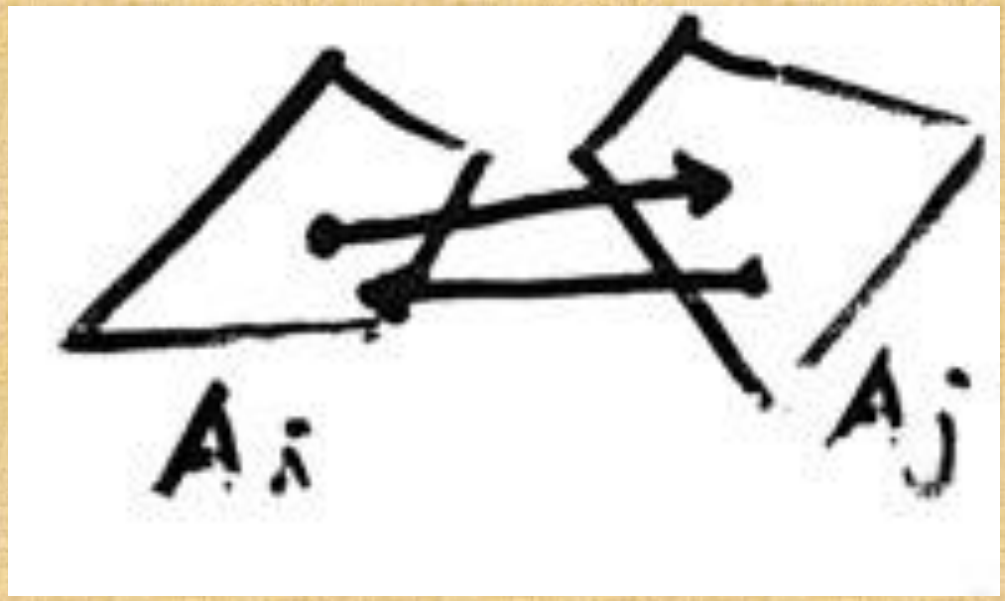
Definitions

- 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.

Reciprocity in radiosity (互惠)

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

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



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

$\mathbf{b} = [b_i]$

$\mathbf{e} = [e_i]$

$\mathbf{R} = [r_{ij}]$ $\quad r_{ij} = \rho_i$ 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 $\mathbf{F}$ 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

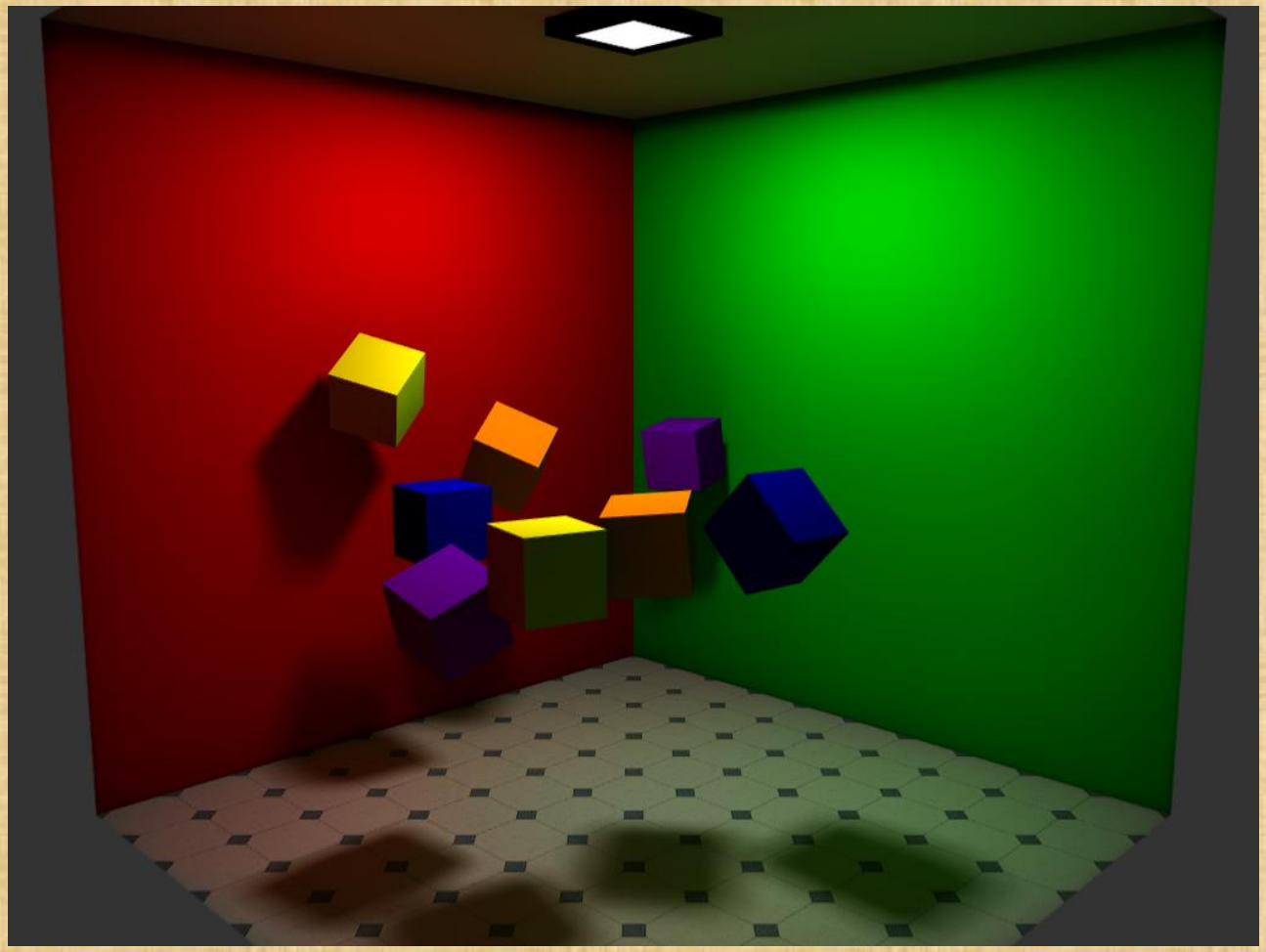
Series Approximation

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

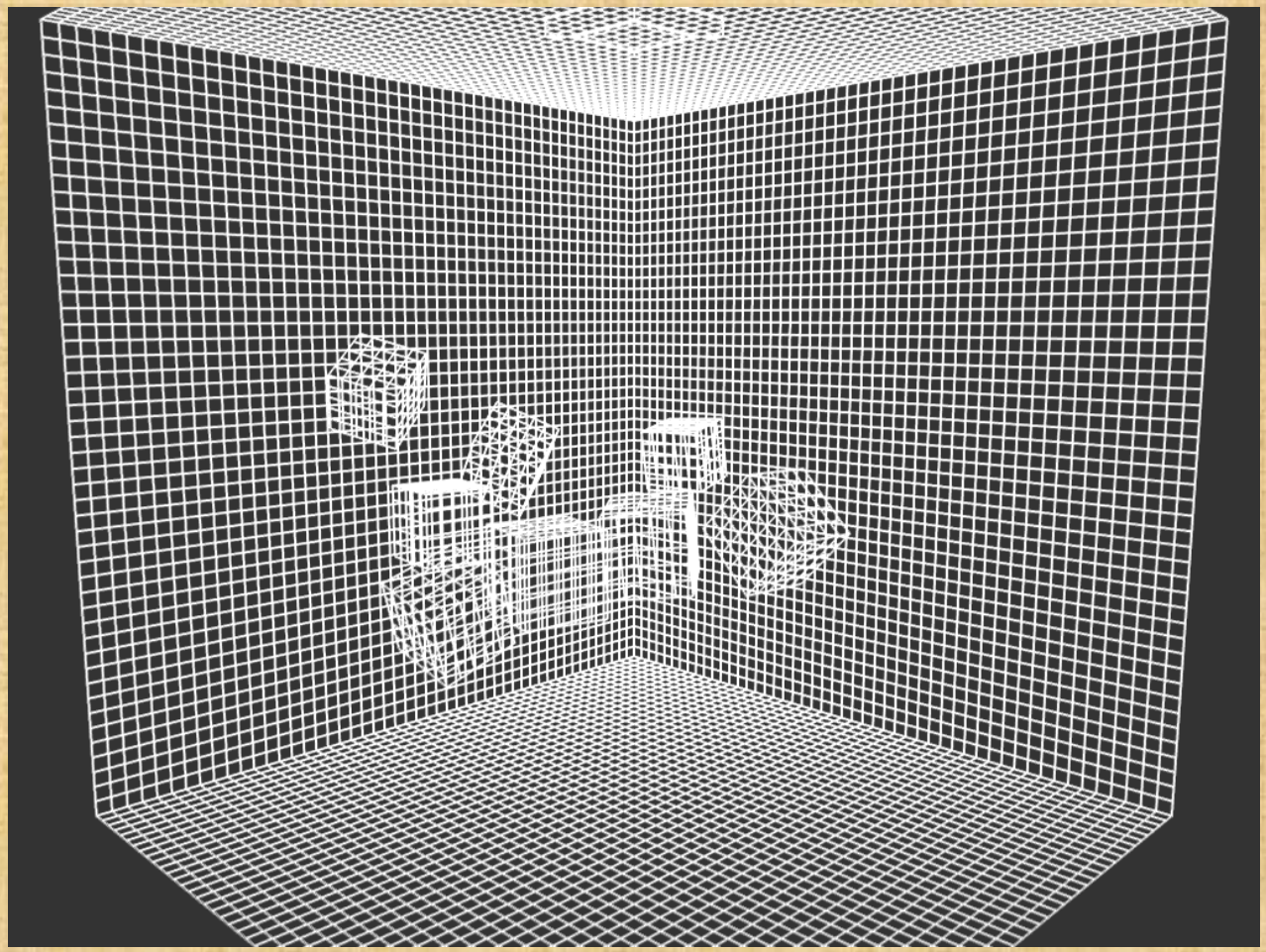
$$\mathbf{[I-RF]^{-1} = I + RF + (RF)^2 + \ldots}$$

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

Rendered Image



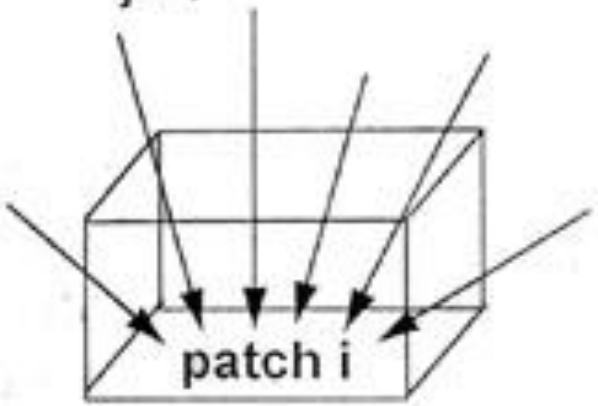
Patches



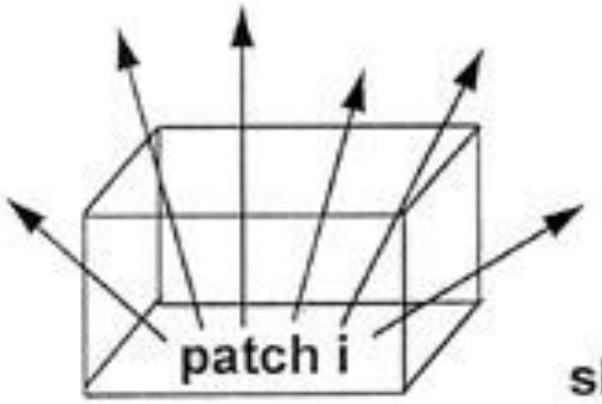
Gathering vs. shooting

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



shooting

$$\begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix} = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix} + \begin{bmatrix} x & x & x & x & x & x & x & x \end{bmatrix} \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix}$$

$$\begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix} = \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix} + \begin{bmatrix} x \\ x \\ x \\ x \\ x \\ x \\ x \\ x \end{bmatrix} \begin{bmatrix} x \end{bmatrix}$$

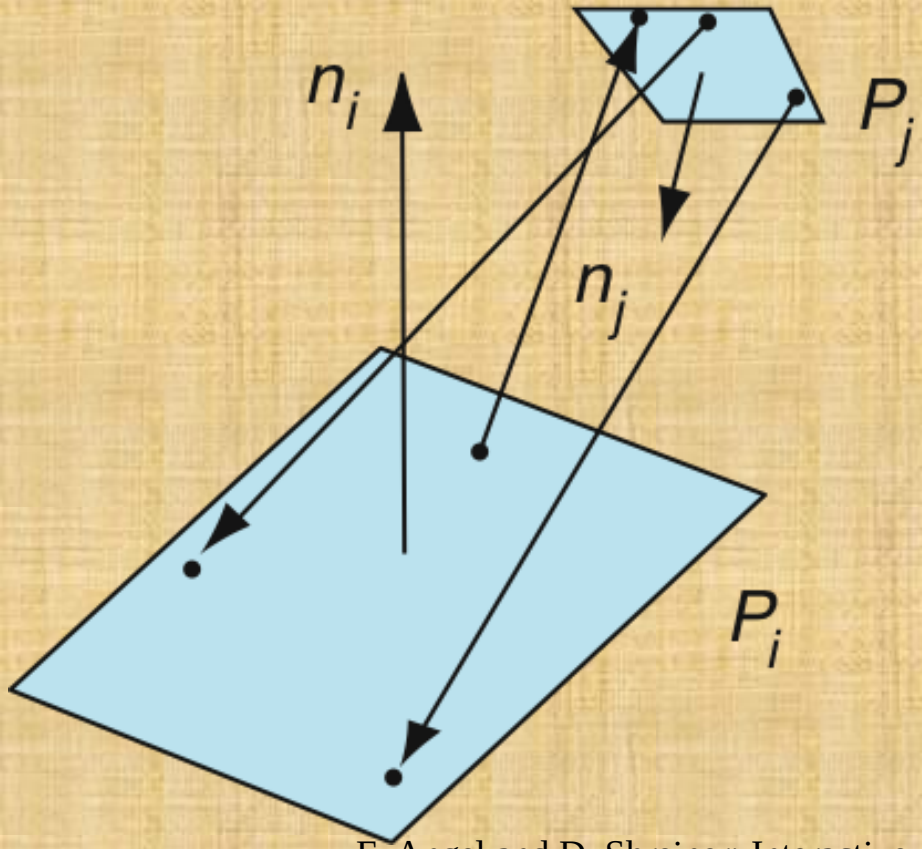
$$B_i = E_i + \sum_{j=1}^{n} (\rho_i F_{ij}) B_j$$

for all j :

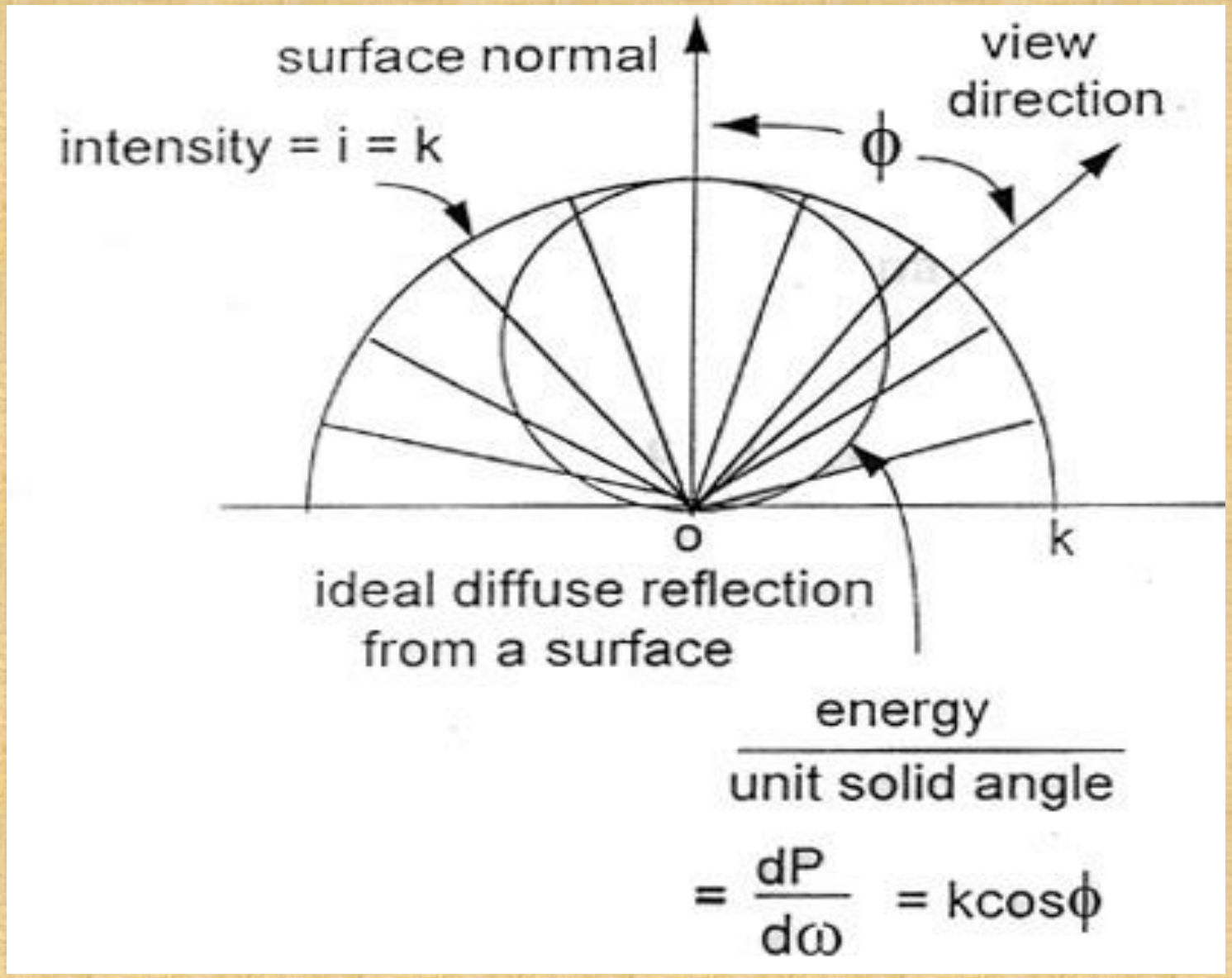
$$B_j = B_j + B_i (\rho_j F_{ji})$$

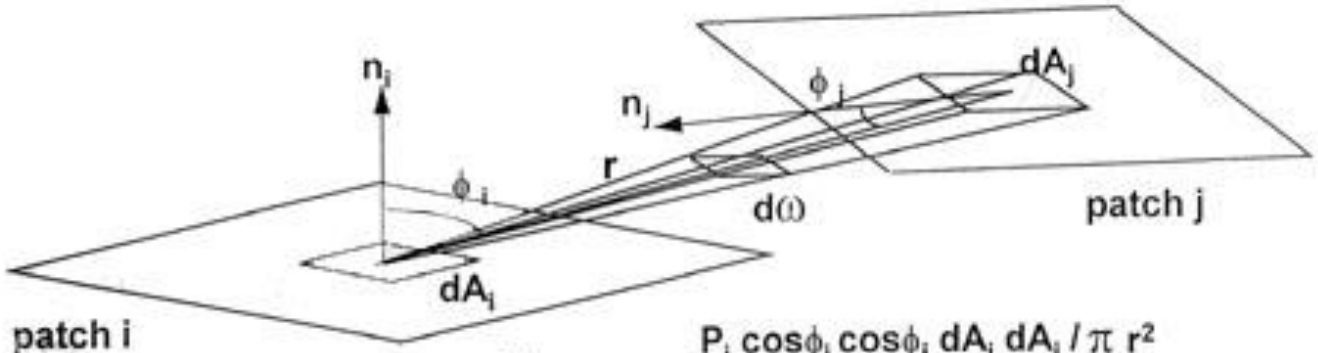
Computing Form Factors

- 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
= dP/dω = kcosϕ



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

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

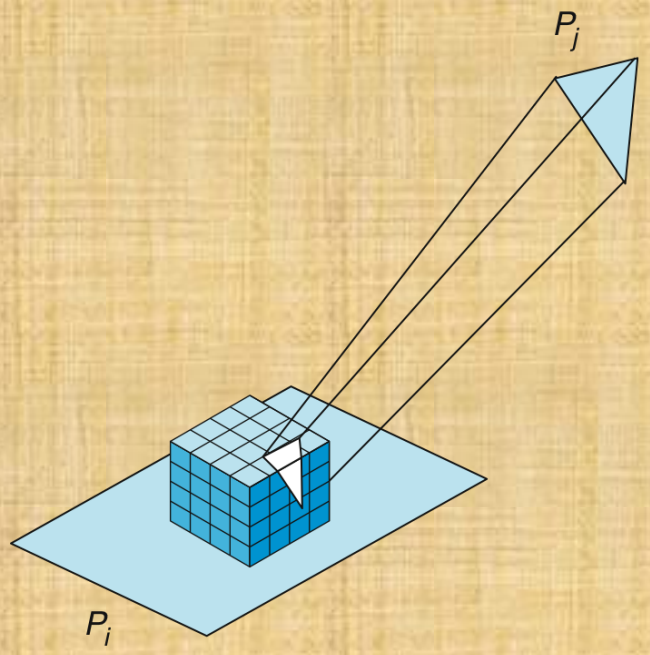
then , by (1), (2)

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

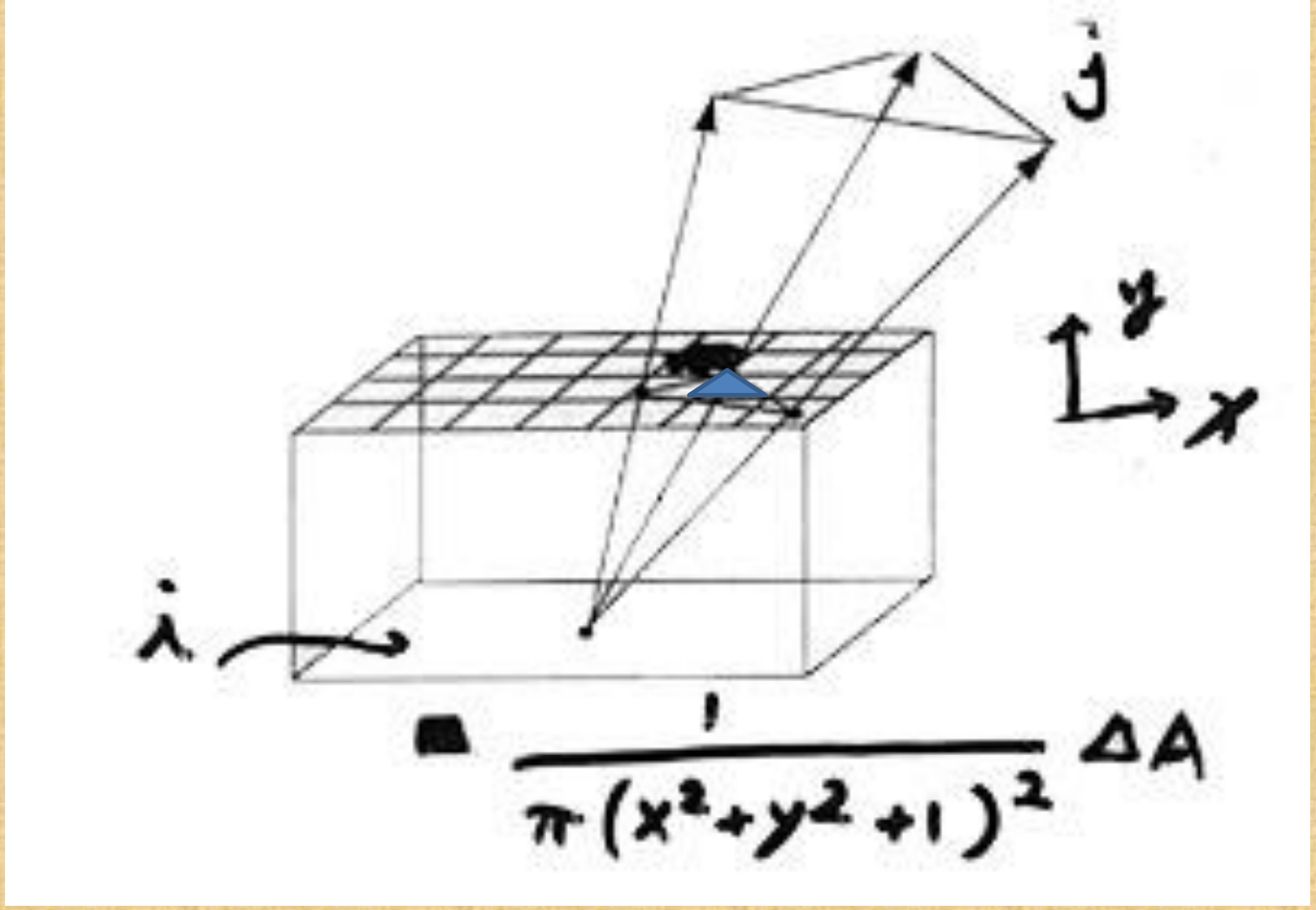
$$dP_i dA_i = i_i \cos\phi_i \, d\omega \, dA_i$$

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

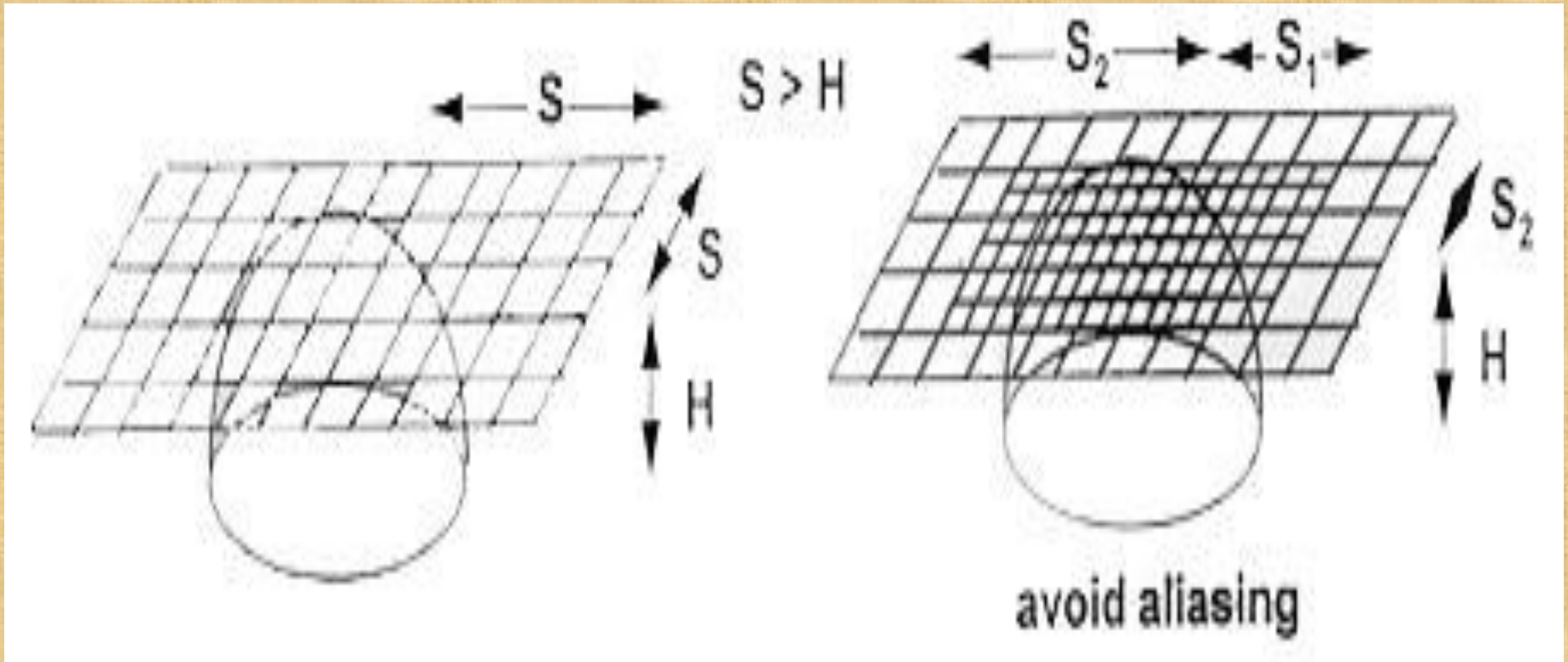
Hemicube



Hemi-Cube method, the Form-factor



Single plane algorithm

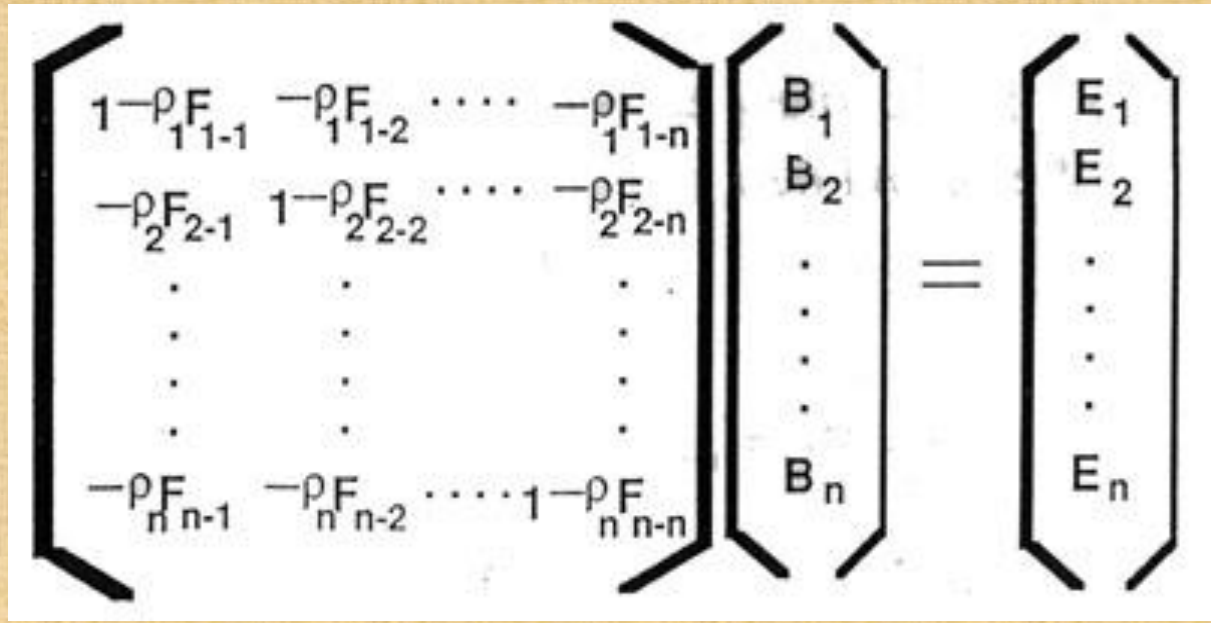


Progressive refinement of radiosity [M. Cohen, D. Greenberg]

Rearranging terms:

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

A set of simultaneous equations



for each iteration, for each patch i

for each patch j:

calculate the formfactors Fij using hemi-cube at patch i

$$\Delta Rad = \rho_j \Delta B_i F_{ij} A_i / A_j$$

$$\Delta B_j = \Delta B_j + \Delta Rad;$$

$$B_j = B_j + \Delta Rad;$$

/*update change since last time patch j shot light */

/* update total radiosity of patch j */

/* reset unshot radiosity for patch i to zero */

$$\Delta B_i = 0;$$

end,

end

initialization:

for all patch i:

if patch i is a light source,

then

$$B_i = \Delta B_i = E_i$$

else

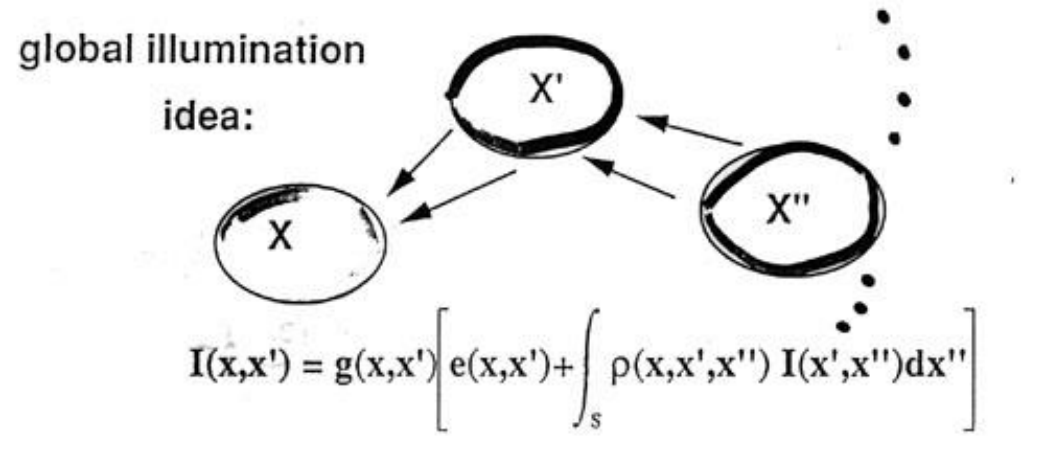
$$Bi = \Delta B_i = 0.$$

The algorithm of progressive refinement of radiosity

Evolution of CG hardware

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

Radiosity



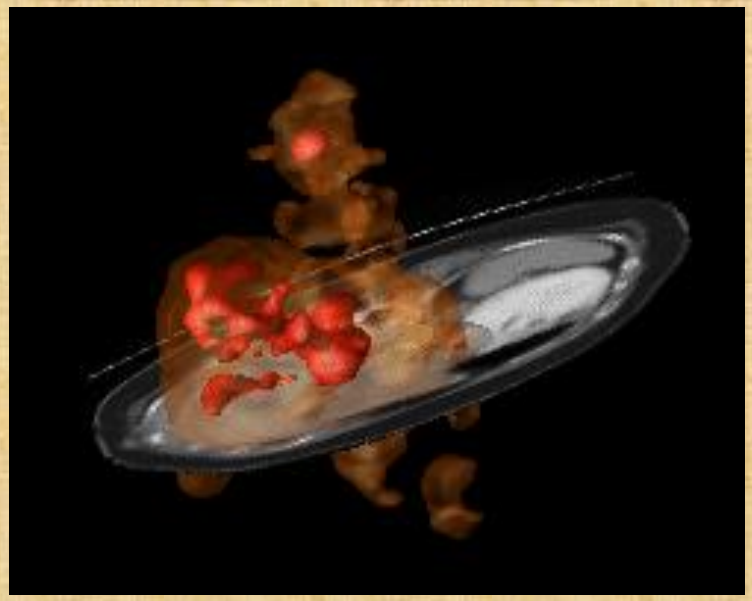
$$B_i A_i = E_i A_i + \rho_i \sum_j B_j F_{ji} A_j$$

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

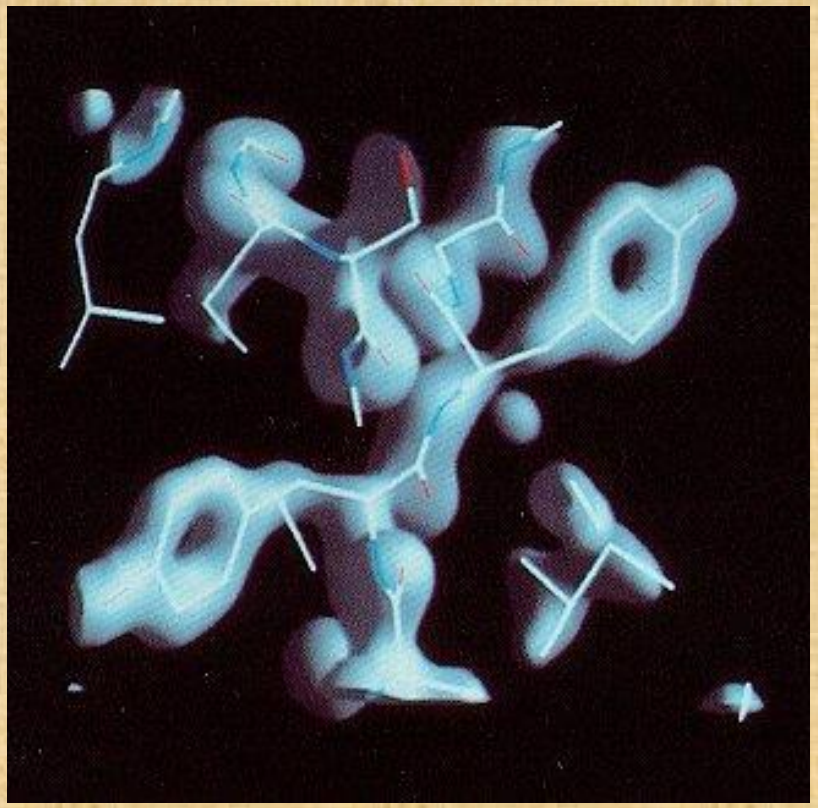
What is Volume Rendering?

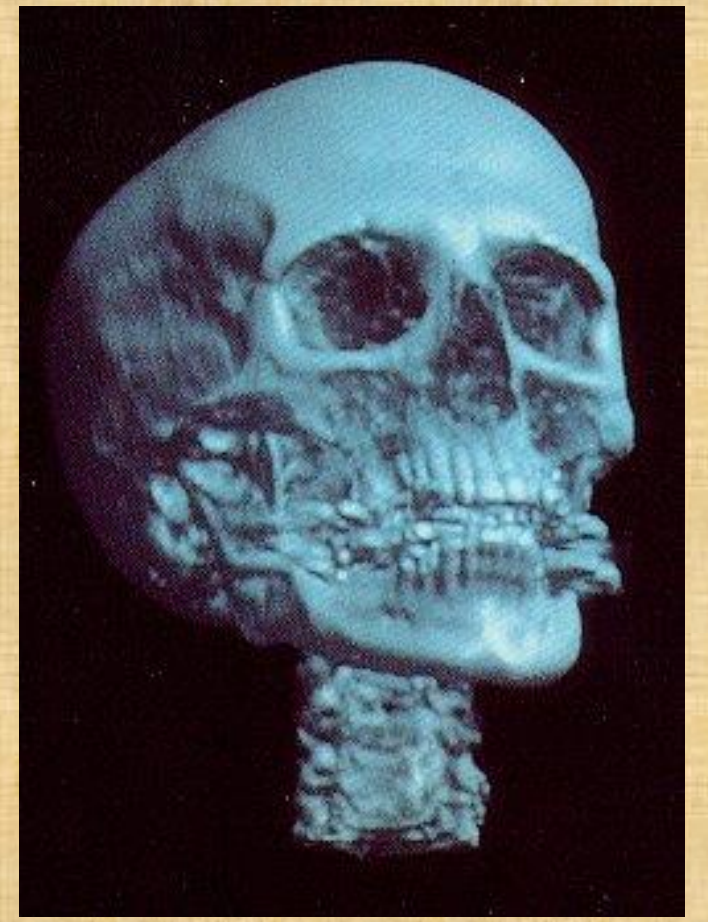
- The term *volume rendering* is used to describe techniques which allow the visualization of three-dimensional data. Volume rendering is a technique for visualizing sampled functions of three spatial dimensions by computing 2-D projections of a colored semitransparent volume.

- 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



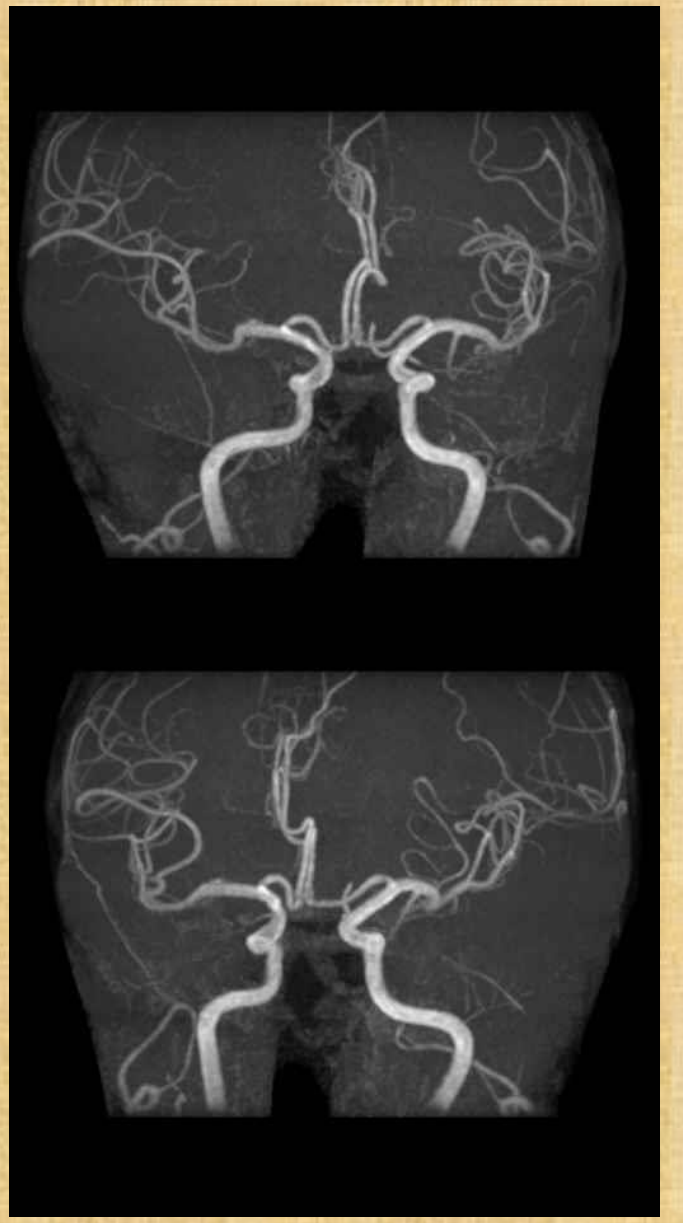
Volume Rendering: result images





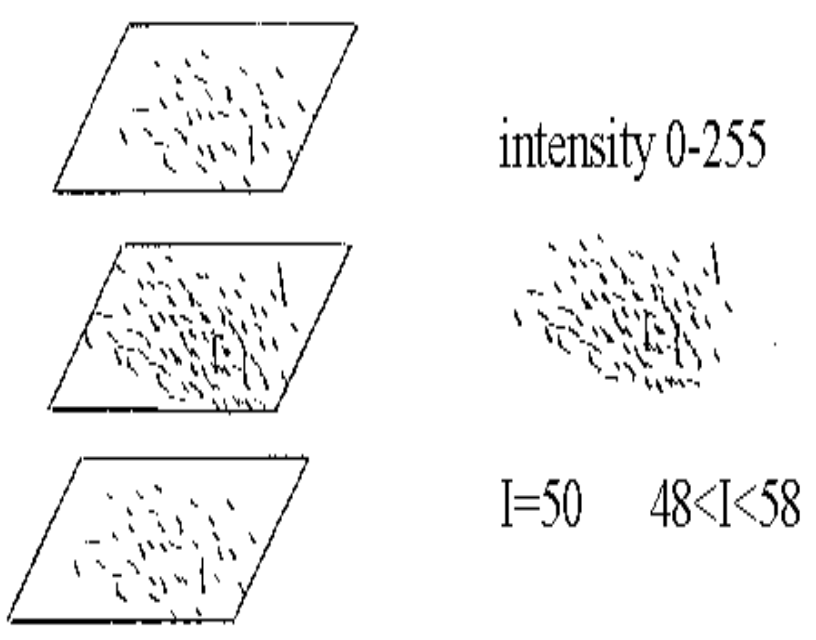
Ming's brain vessels, (MRI, 加顯影劑, angiogram)

MRI DEMO: https://youtu.be/-FJomtBoUQg



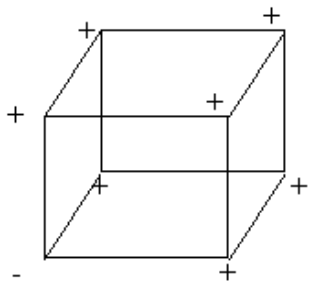
How to calculate surface normal for scalar field?

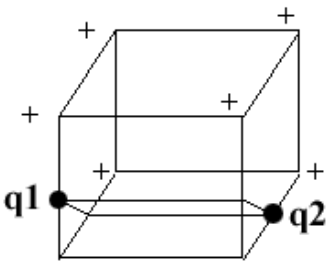
- gradient vector
- D(i, j, k) is the density at voxel(i, j, k) in slice k



Marching cubes (squares)

Paper: Siggraph 1987

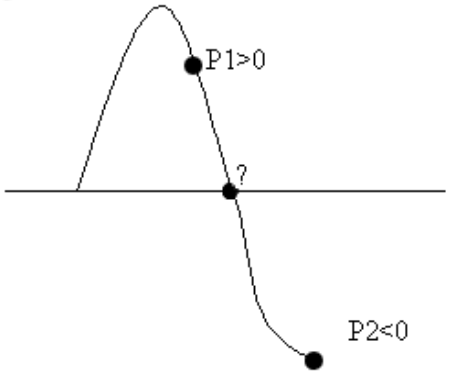


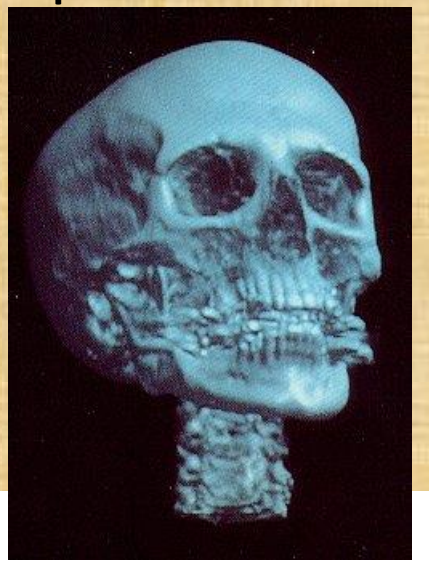


+: I>50

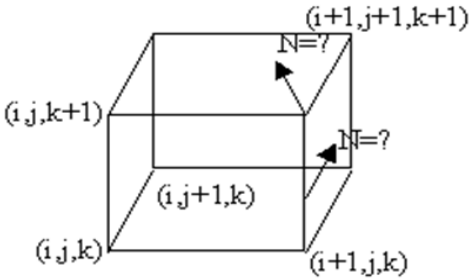
-: I<50

2^8=256 ways reduced to 14 patterns

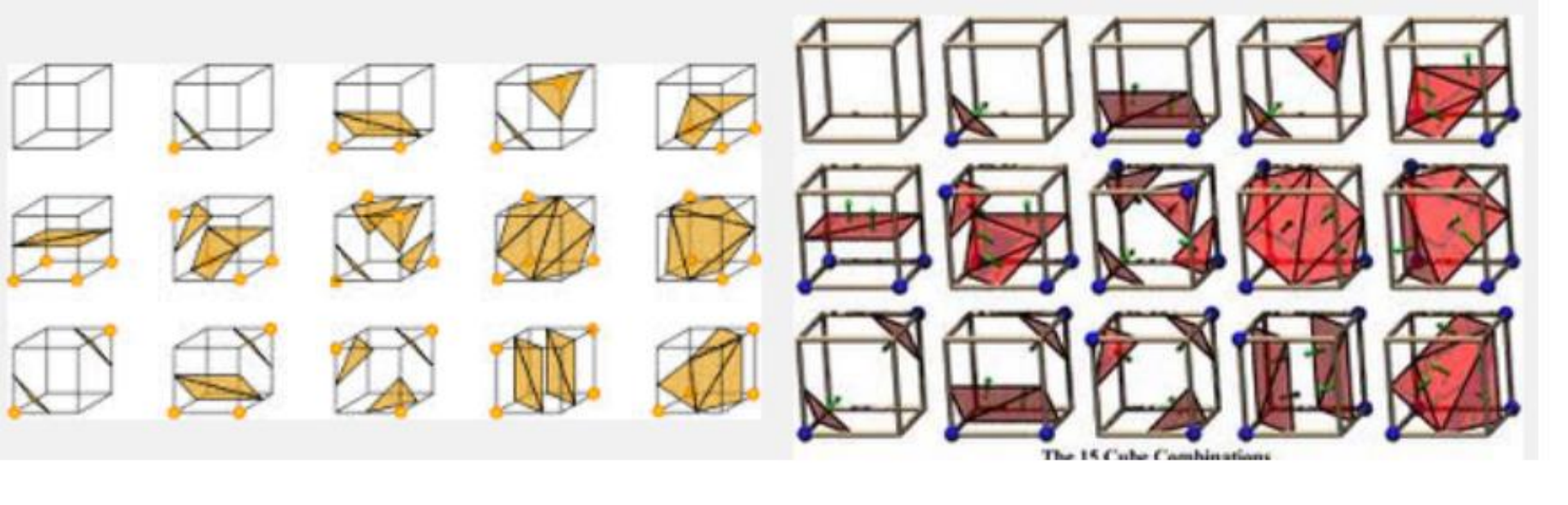




linearly intepolate



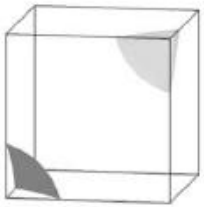
Marching Cubes: by WE Lorensen, HE Cline, as one of the best cited papers in computer graphics (15,849 times, up to 2020)

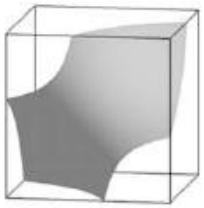


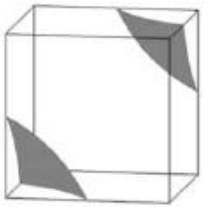
Surface normal calculation for cube corners

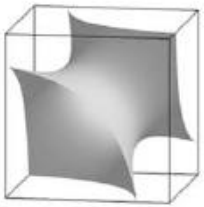
- $G_x(i, j, k) = (D(i+1, j, k) - D(i-1, j, k))/2$
- $G_y(i, j, k) = (D(I, j+1, k) - D(I, j-1, k))/2$
- $G_z(i, j, k) = ((D(I, j, k+1) - D(I, j, k-1))/2$

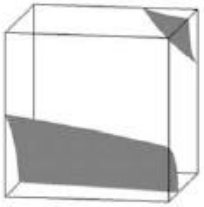
What can be improved to Marching Cubes? Inside the cube


C3.0 (C19.1)


C3.1 (C19.0)

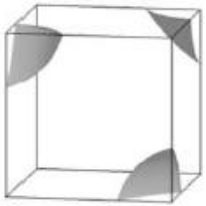

C4.0.0 (C18.0.0)


C4.0.1 (C18.0.1)

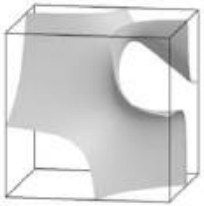

C6.0.0 (C16.1.0)


6.0.1 (C16.1.1)

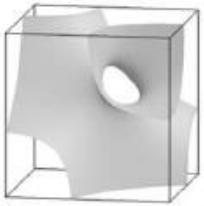

C6.1 (C16.0)

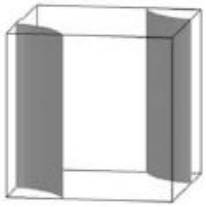

C7.0(C15.3)

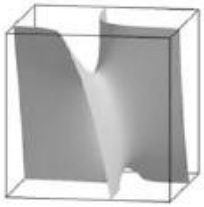

C7.1 (C15.2)

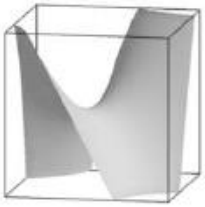

C7.2 (C15.1)

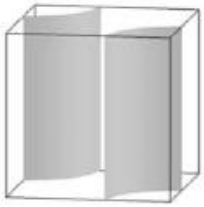

C7.3.0 (C15.0.0)


C7.3.1 (C15.0.1)


C10.0.0


C10.0.1

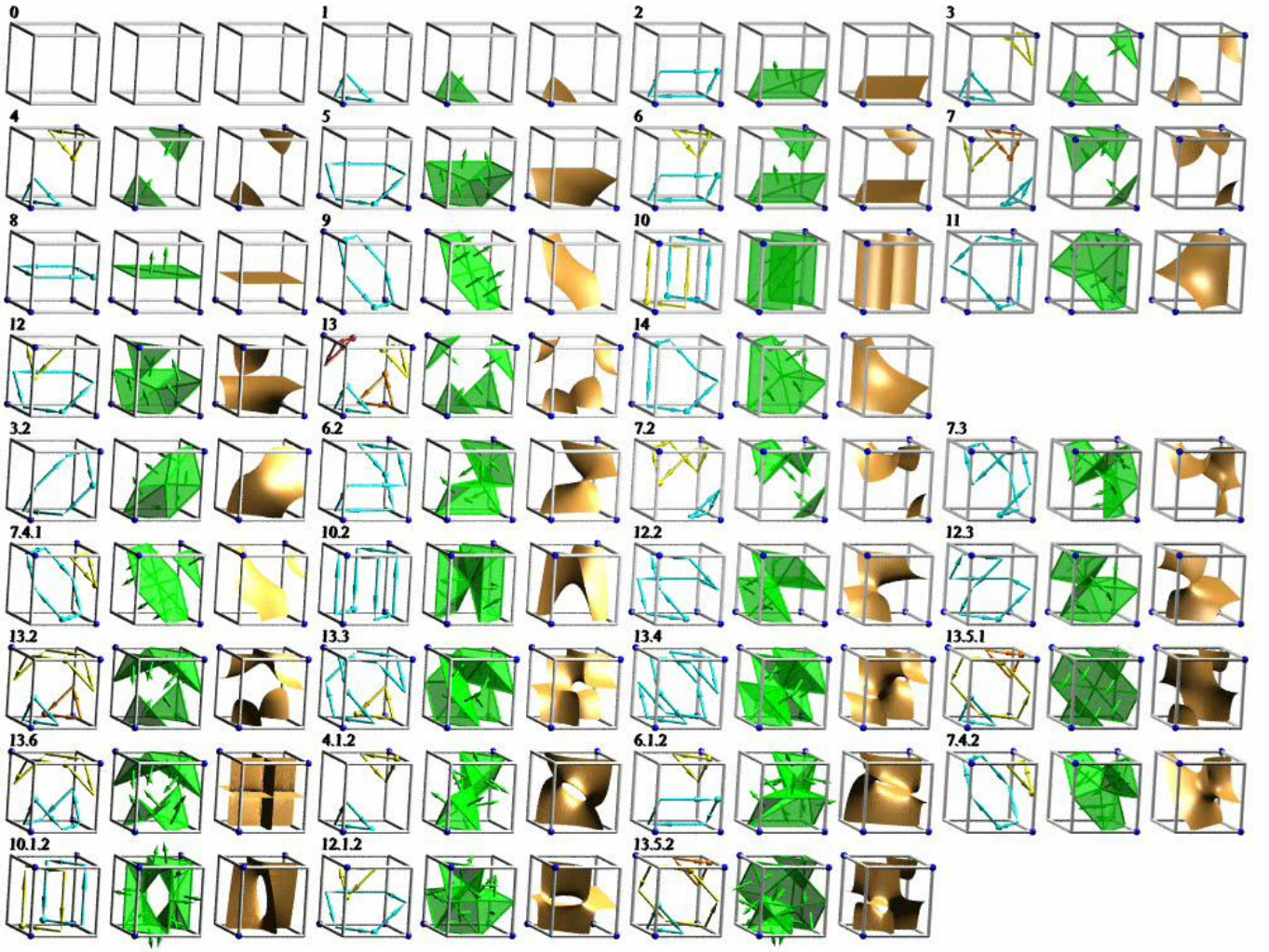

C10.1


C10.2.0


C10.2.1

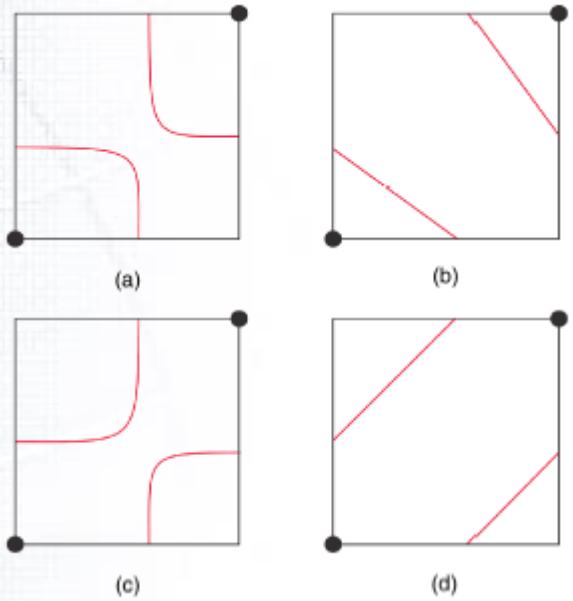
All configurations:

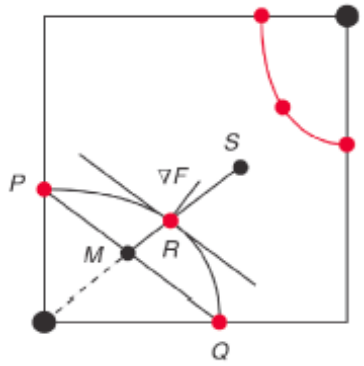
Chien-Chang Ho, Pei-Lun Lee, Yung-Yu Chuang, Bing-Yu Chen, Ming Ouhyoung, "Cubical Marching Squares: Adaptive Surface Extraction from Volume Data with Sharp Features and Better Topology", Computer Graphics Forum, (also in EuroGraphics2005), Vol. 24, No. 4, pp. 537-545, 2005

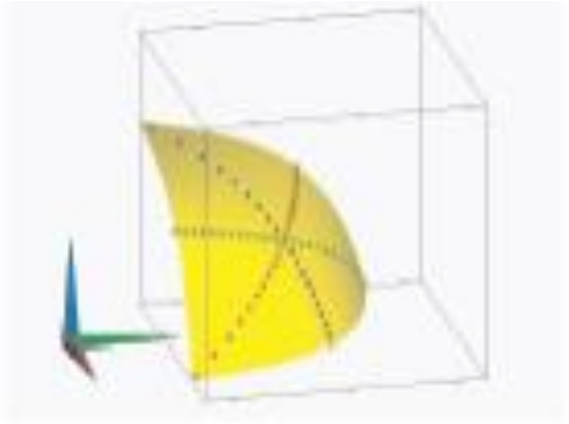


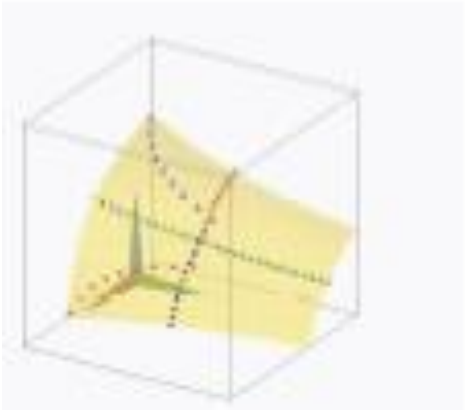
DEMO: https://youtu.be/8sf9soKi7j4

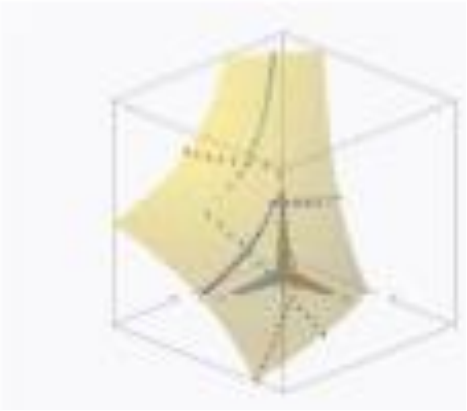
Inside the cube



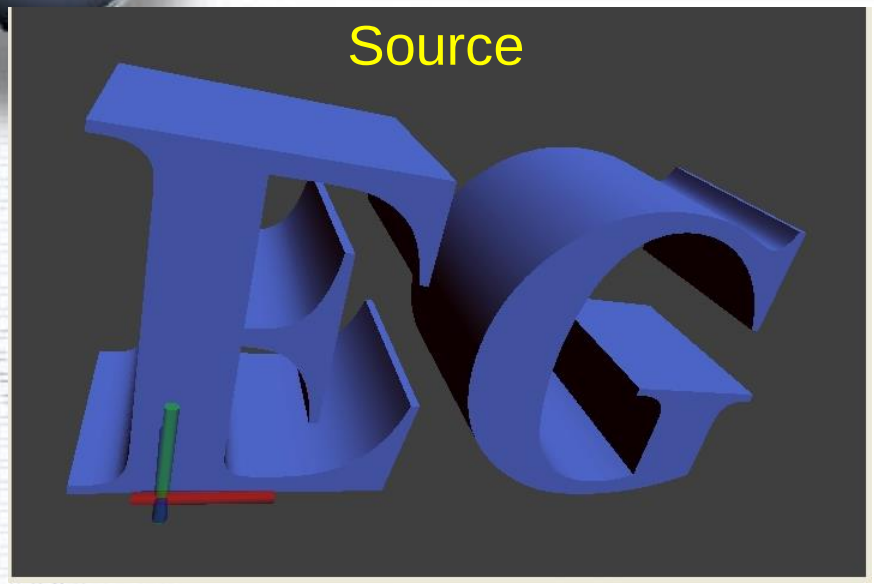


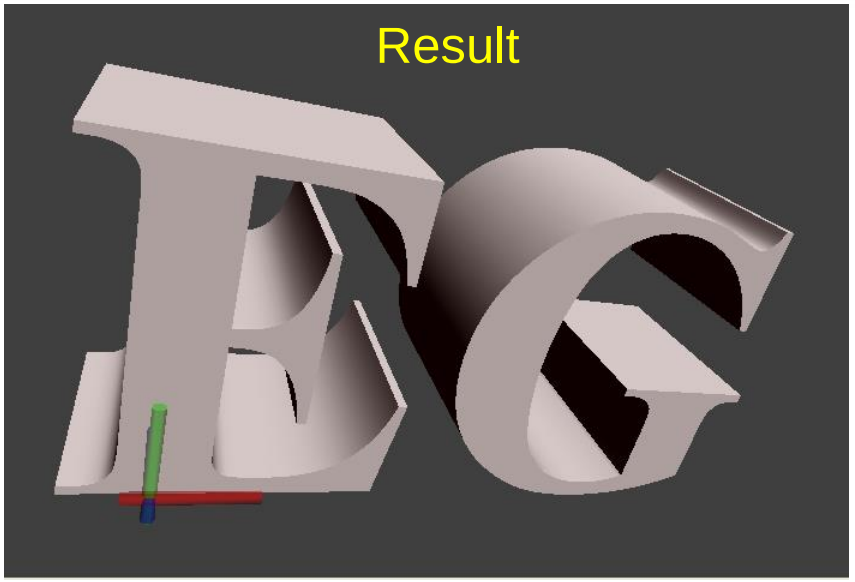


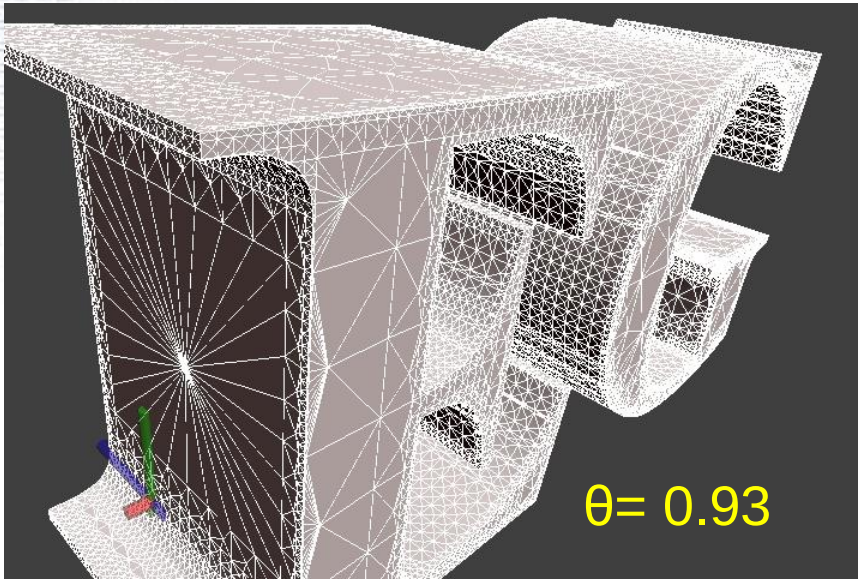


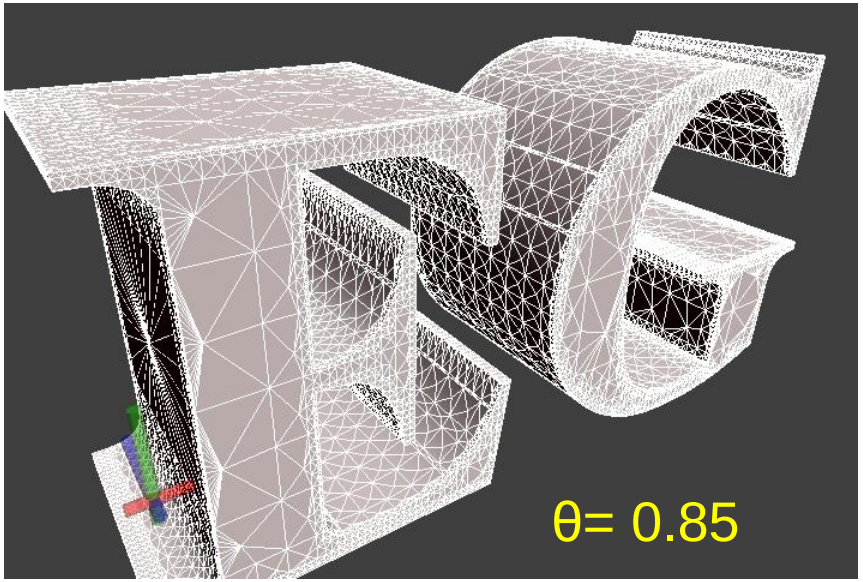


Results









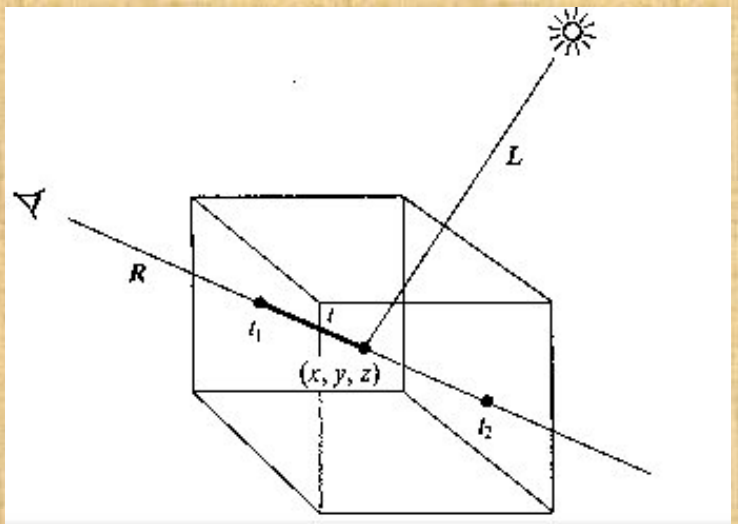
Why “Cubical Marching Squares”?

- Topology is correctly preserved by examining the sharp features of different components

- DEMO: https://youtu.be/8sf9soKi7j4
- Best Student Paper Award (EuroGraphics 2005)

Ray casting for volume rendering

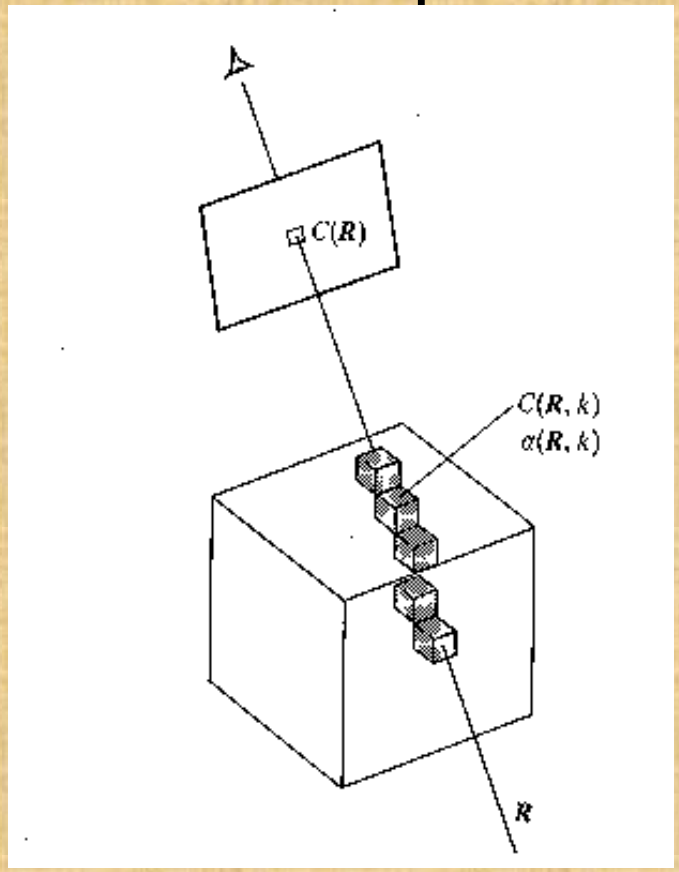
- **Theory**
  - Currently, most volume rendering that uses ray casting is based on the Blinn/Kajiya model. In this model we have a volume which has a density D(x,y,z), penetrated by a ray ***R***.



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 **α (X)** are "combined" into single values to provide a final pixel intensity.
- α : Opaque Index

  (不透明度)
- C: Color/Intensity



Splatting (潑濺, 潑墨)

This is a technique which trades quality for speed. Here, every volume element is splatted, as Lee Westover said, like a snow ball, on to the viewing surface in back to front order. These splats are rendered as disks whose properties (color and transparency) vary diametrically in normal (Gaussian) manner.

- DEMO https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/VolRenderShearWarp.gif/474px-VolRenderShearWarp.gif

Splatting: (back to front tracing)

Westover, Lee Alan (July 1991). "SPLATTING: A Parallel, Feed-Forward Volume Rendering Algorithm".

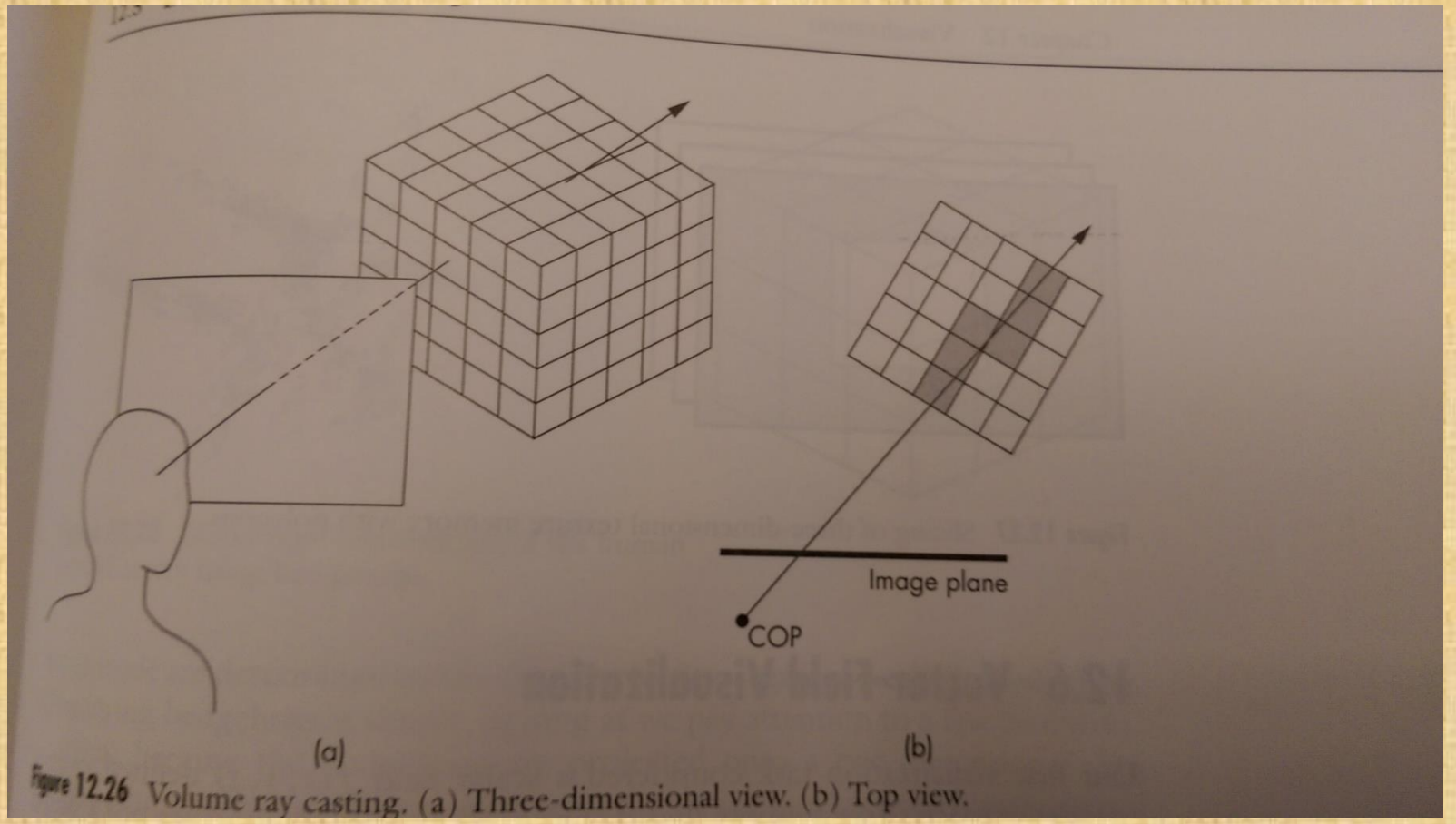


Figure 12.26 Volume ray casting. (a) Three-dimensional view. (b) Top view.

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 : from 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)

Transparency formula: II

$C_{d'} = (1-\alpha_s)\, C_d + \alpha_s\, C_s$

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

For example, 背景: 綠色 destination, 前景, 藍色 source

back to front rendering,

case 1:前景, 全透明, color $C_{d'} = C_d$, since $\alpha_s = 0$

case 2 :前景, 全不透明, color $C_{d'} = C_s$, since $\alpha_s = 1.0$

case 3 :前景, 半透明, color $C_{d'} = ½ (C_s + C_d)$, since $\alpha_s = 0.5$,

$\alpha_{d'} = 0.5\, \alpha_d + \alpha_s$

如果, 前景由很多 object (voxel) 排列而成, 由最接近背景的 Voxel 開始計算, iterative, 直到最接近眼睛的Voxel, 得出顏色 (亮度).

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

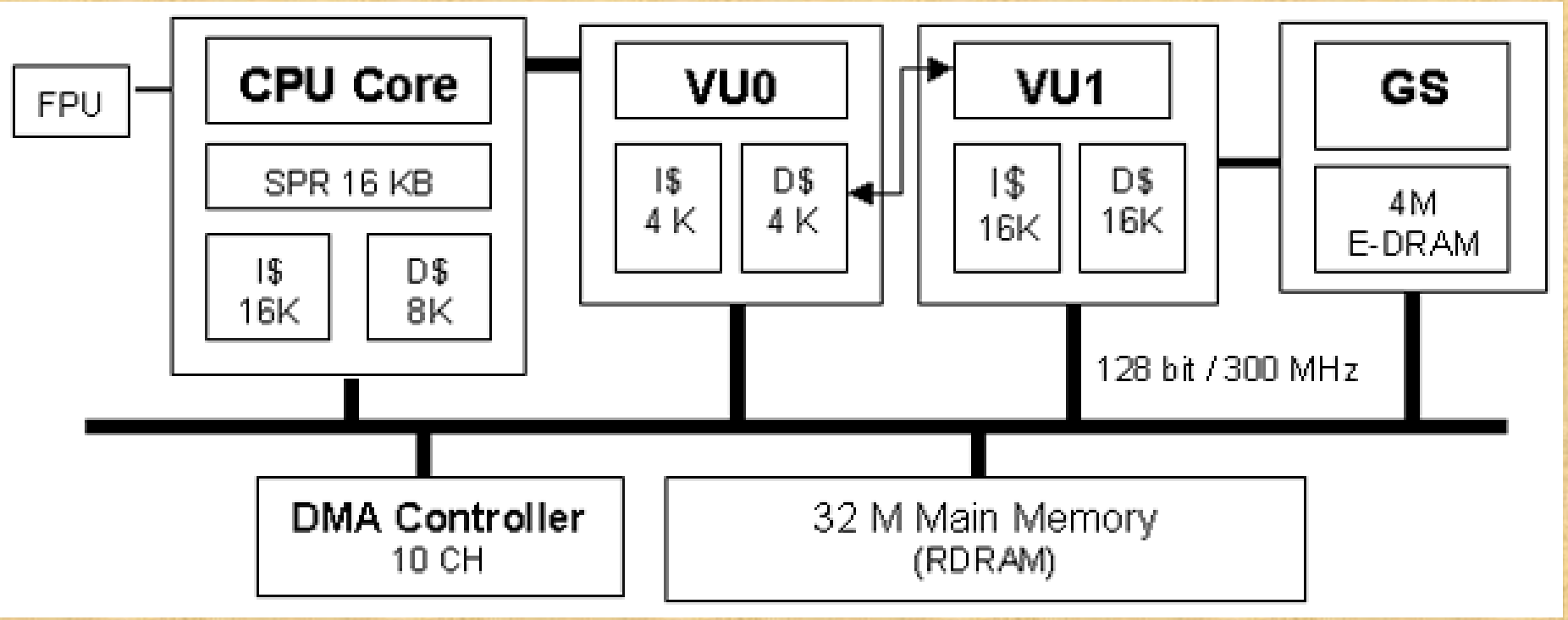
True Color display—Old Systems

- 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

ABOUT | GAMES | SPECS | GALLERY

CPU: Cell Broadband Engine™
GPU: RSX

MEMORY: 256MB XDR Main RAM 256MB
GDDR3 VRAM

HDD: 2.5" Serial ATA (60GB)

I/O: USB 2.0 x 4
Memory Stick/SD/CompactFlash Slots

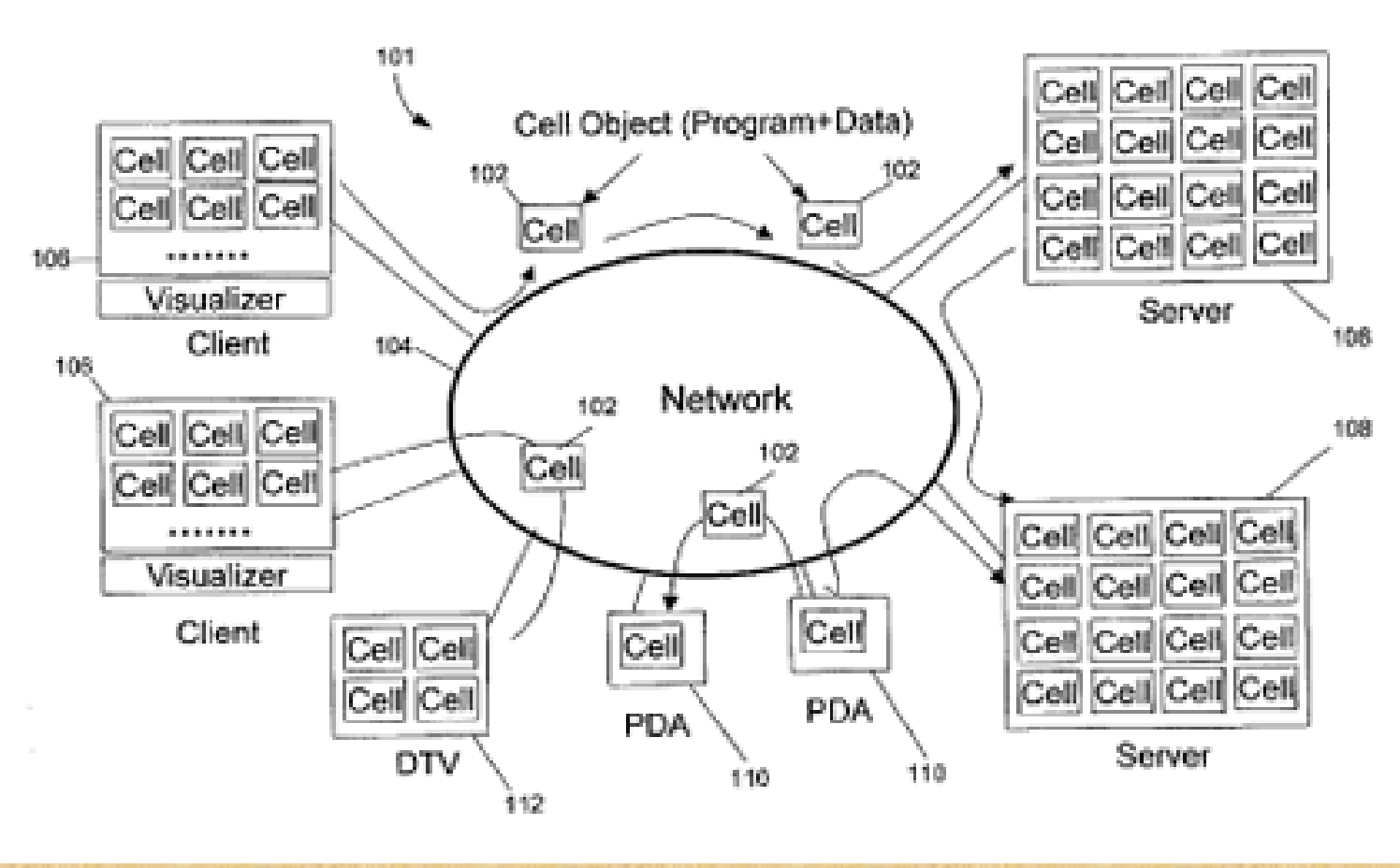
COMMUNICATION:
Ethernet (10BASE-T, 100BASE-TX, 1000BASE-T)
IEEE 802.11 b/g Wi-Fi*
Bluetooth 2.0 (EDR)
Wireless Controller Bluetooth (up to 7)

AV OUTPUT:
Screen size: 480i, 480p, 720p, 1080i, 1080p
HDMI**: HDMI out - (x1 / HDMI)
Analog: AV MULTI OUT x 1
Digital audio: DIGITAL OUT (OPTICAL) x 1
Blu-ray/DVD/CD DRIVE "read only"

DIMENSIONS: Approximately
325mm (W) x 98mm (H) x 274mm (D)

WEIGHT: Approximately 5 kg

PlayStation 3 architecture

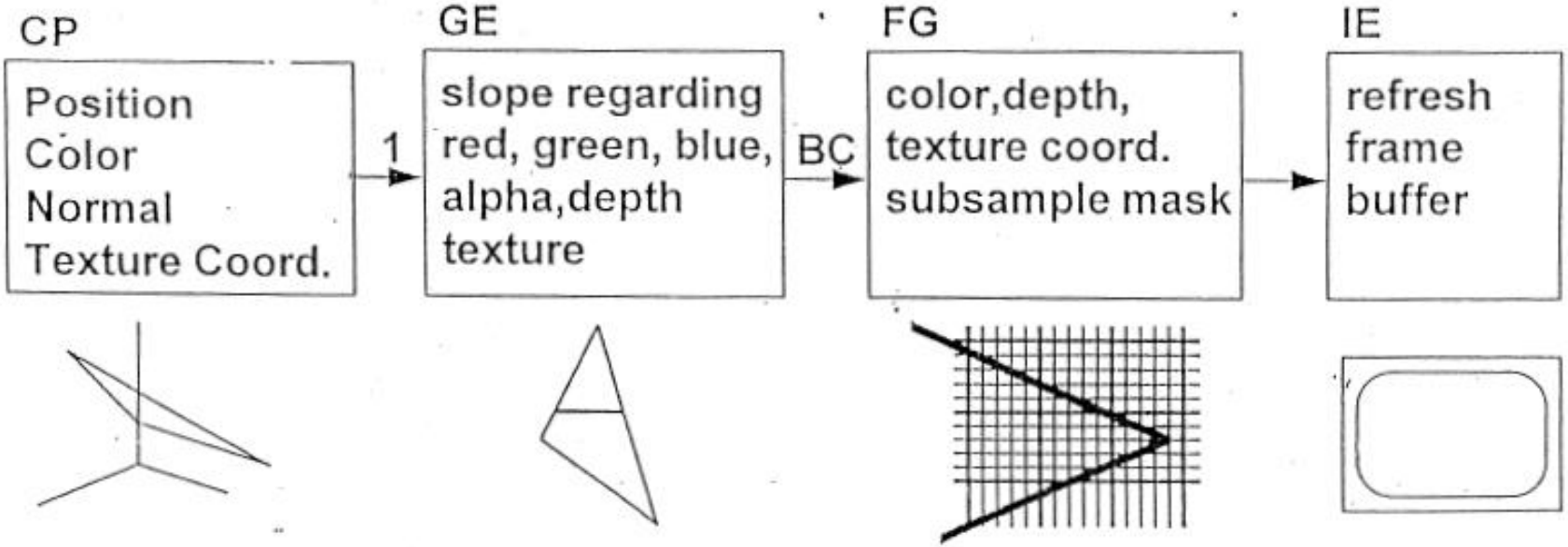


GPU/Graphics Hardware

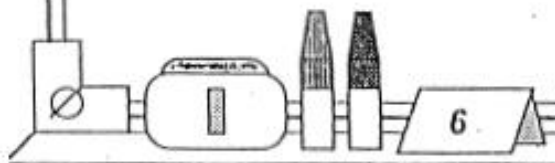
- Moor’s Law
- GPGPU
- Easy for VR development: Real-time response

Moore's law: predicts that the number of transistors per square inch has since doubled approximately every 18 months. (CPU speed, memory size)

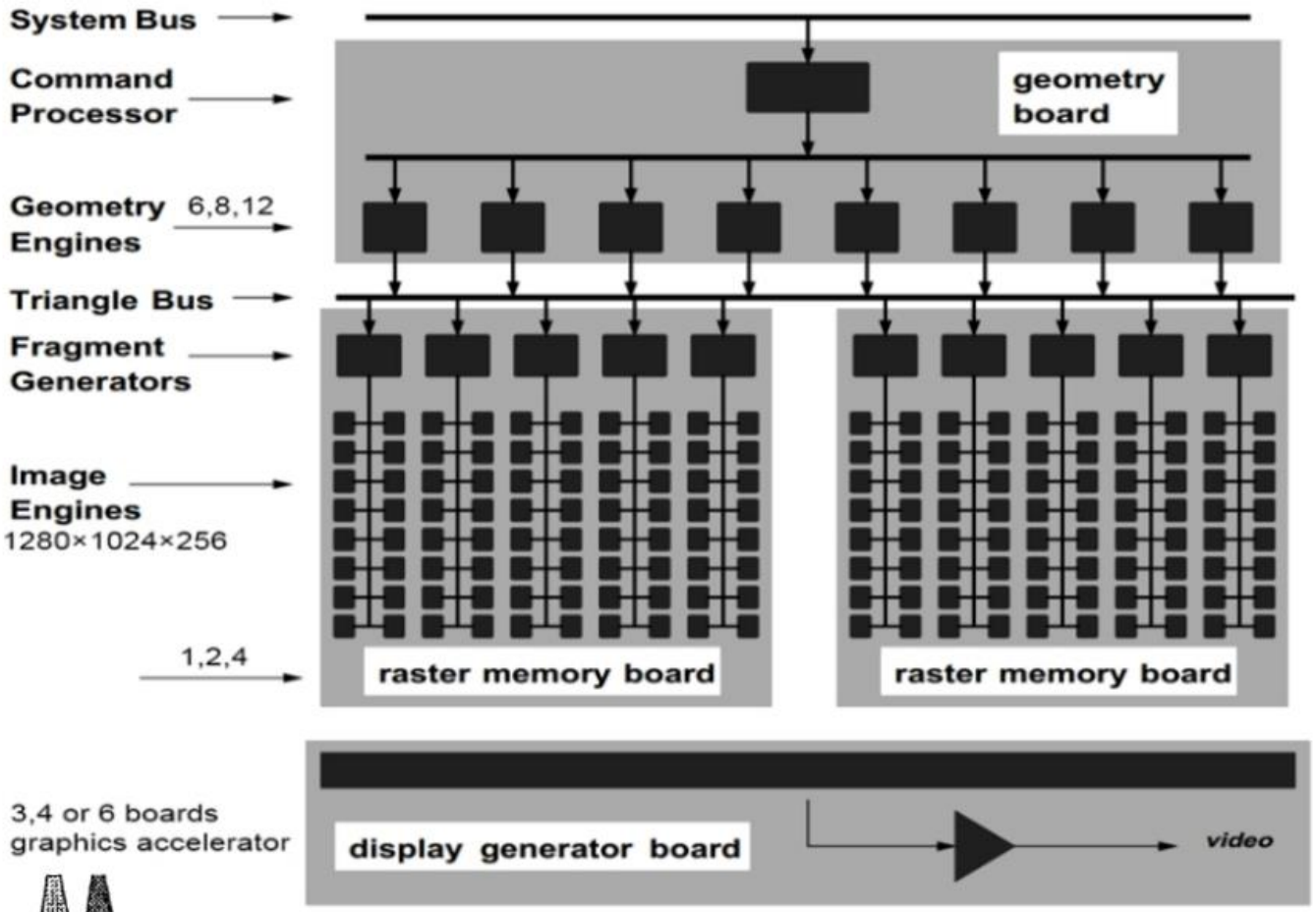
A Triangle



- The stream of rendering commands merges only at Triangle Bus
- Each triangle is processed by almost all the processors
- Moderate FIFO memories are included



RealityEngine



Illumination model

1) Ambient light (漫射)

$I = I_a \acute{} \ k_a \acute{} \ Obj(r,g,b)$

I_a : intensity of ambient light

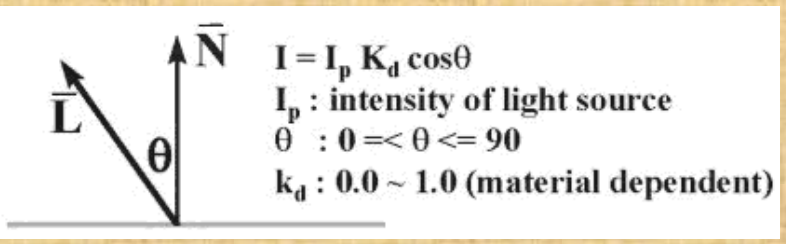
k_a : 0.0 ~ 1.0

$Obj(r,g,b)$: object color

2) Diffuse reflection (散射)

$I = I_p(r,g,b) \acute{} \ K_d \acute{} \ Obj(r,g,b) \acute{} \ \cos q$

$I_p(r,g,b)$: light color



3) Light source attenuation

$$I = I_a k_a + f_{att} I_p K_d \left(\vec{N} \cdot \vec{L}\right) \qquad f_{att} = \frac{1}{d_L^2}$$

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. Others—Human Computer Interface, Installation Arts, Water Rendering etc.
15. 3D video, stereo video, DSLR_Bokeh_blur simulation (from depth images/video), image deblur