Basic shadowing technique for early animations (Toy Story, etc.) and EVERY 3D video game.

Problems with shadow maps:

1. Hard shadows (for point lights only)
2. Quality depends on shadow map resolution
3. Involves equality comparison of floating point depth values → The noise on the green image in the last page.
   means issues of scale, bias, tolerance.

hard shadow

soft shadow (not point light)

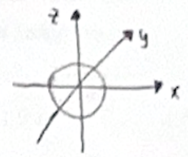
G1

- Geometry

There are many ways to represent geometry :

- Implicit : points satisfy some specified relationship

  ex. $x^2+y^2+z^2=1$ (More generally, $f(x,y,z)=x^2+y^2+z^2-1=0$)



  pros: Inside / outside test easy; good for ray-to-surface intersection

  $f(\frac{3}{4},\frac{1}{2},\frac{1}{4})=-\frac{1}{8}<0$ → inside

  cons: complex shapes is hard to represent

1. Algebraic Surface :

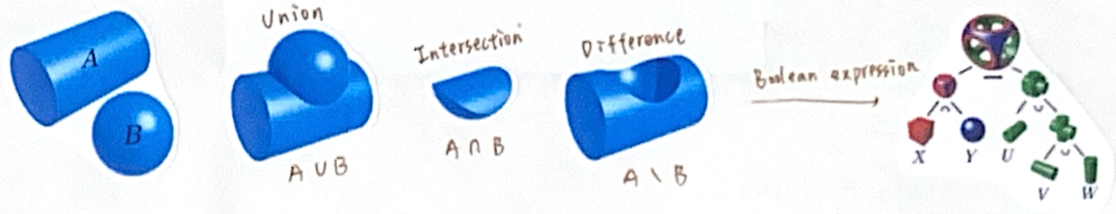
   ex. $x^2+y^2+z^2=1$ → sphere

   $(R-\sqrt{x^2+y^2})^2+z^2=r^2$ → 甜甜圈

   $(x^2+\frac{9y^2}{4}+z^2-1)^3=x^2z^3+\frac{9y^2z^3}{80}$ → 愛心

2. Constructive Solid Geometry : (CSG)

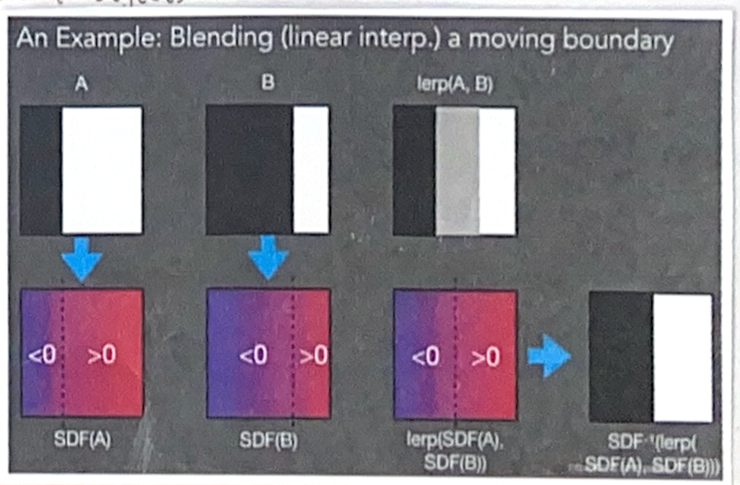
   Combine implicit geometry via Boolean operations



   $(X\cap Y)\setminus(U\cup V\cup W)$

3. Distance Functions :

   Giving minimum distance (could be signed distance) from anywhere to objects



4. Level Set Methods: (like distance function)

Store a grid values approximating function

$f(x)=0$ — surface is found when interpolated values equal zero

⇒ provide much more explicit control over shape (such as texture)

* Also use at medical data: CT, MRI...

physical simulation: find air-liquid boundary

5. Fractals:

Exhibit self-similarity, detail at all scales.

"Language" for describing natural phenomena

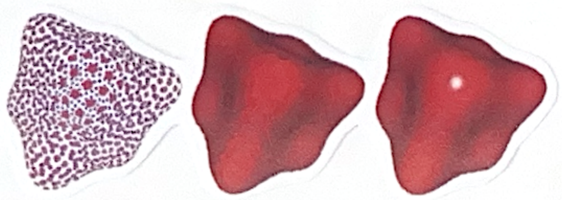
Hard to control shape!

顯示

* Explicit:

1- Point cloud:

- Easiest representation: list of points (x,y,z)
- Easily represent any kind of geometry
- Useful for LARGE datasets (>> 1 point / pixel)
- Often converted into polygon mesh
- Difficult to draw in undersampled regions



LT2 多邊形的圖形

2. Polygon mesh:

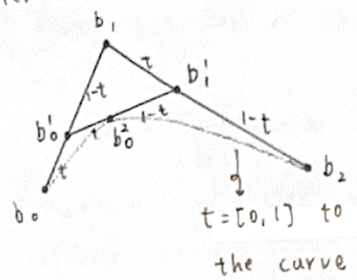
- Store vertices & polygons (often triangles or quads)
- Easier to do processing / simulation, adaptive sampling
- More complicated data structures
- Perhaps most common representation in graphics

ex. Wavefront Object File (.obj) Format

- Commonly used in Graphics research
- Just a text file that specifies vertices, normals, texture coordinates and their connectivities

3. Curves:

★ Bézier curve:



b_0, b_1, b_2

↓

$b_0'(t) = (1-t)\,b_0 + t\,b_1$

$b_1'(t) = (1-t)\,b_1 + t\,b_2$

↓

$b_0^2(t) = (1-t)\,b_0' + t\,b_1'$

⇒ $b_0^2(t) = (1-t)^2 b_0 + 2t(1-t)\,b_1 + t^2 b_2$

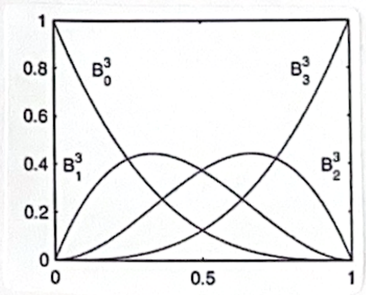
⇒ Bernstein form of a Bézier curve of order n:

$$b^n(t) = b_0^n(t) = \sum_{j=0}^{n} b_j B_j^n(t), \quad B_j^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

$b^n(t)$: Bézier curve order n (vector polynomial of degree n)

$B_j^n(t)$: Bernstein polynomial (scalar polynomial of degree n)

b_j: Bézier control point (vector in R^N)



properties of Bézier curves:

1. Interpolates endpoints: ex. for cubic Bézier: $\begin{cases} b(0) = b_0 \\ b(1) = b_3 \end{cases}$

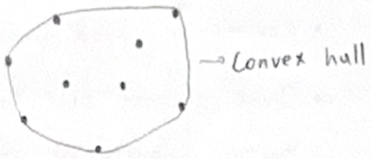
2. Tangent to end segments: ex. for cubic Bézier: $\begin{cases} b'(0) = 3(b_1 - b_0) \\ b'(1) = 3(b_3 - b_2) \end{cases}$

3. Affine transformation property:

   Transform curve by transforming control point

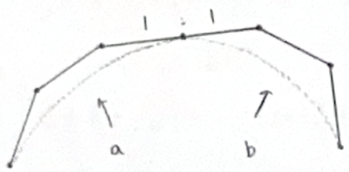
4. Convex hull property:

   Curve is within convex hull of control points



⇒ piecewise Bézier curves.

Chain many low-order Bézier curves to get high order Bézier curve.



C^0 continuity: $a_n = b_0$

C^1 continuity: $a_n = b_0 = \frac{1}{2}(a_{n-1} + b_1)$

↳ 可視為一階導數連續

⋮

* Other types of splines:

Spline: a continuous curve constructed so as to pass through a given set of a points and have a certain number of continuous derivatives.

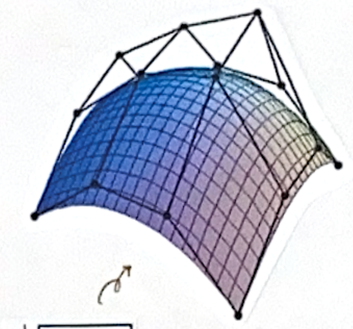
B-splines (Basis splines):
- Require more information than bézier curves.
- Satisfy all important properties that Bézier curves have.

NURBS (non-uniform rational B-splines)

4. Surface:

* Bézier surface:

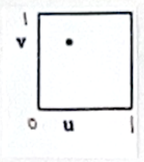
For bi-cubic Bézier surface patch,



Input: 4×4 control points

↓

Output: 2D surface parameterized by (u, v) in $[0, 1]^2$

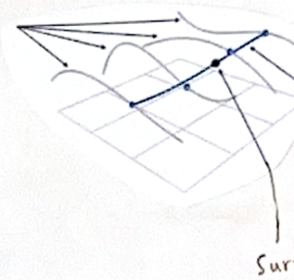


Method: Separable 1D de Casteljau Algorithm

Goal = Evaluate surface position corresponding to (u, v)

(u, v)-separable application of de Casteljau algorithm:

1. Use de Casteljau to evaluate point u on each of the 4 Bézier curves in u. This gives 4 control points for the "moving" Bézier curve.
2. Use 1D de Casteljau to evaluate point v on the "moving" curve.



Geometry processing: mesh operations:

Increase resolution (upsampling)

Decrease resolution (downsampling)

Try to preserve shape/appearance

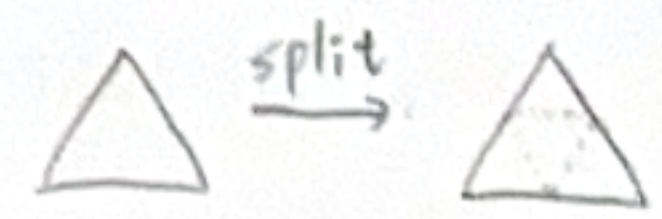
Same #triangle

Modify sample distribution to improve quality.

1. Subdivision:

* Loop subdivision: Common subdivision rule for triangle meshes
入門

Step 1. Create more triangles (vertices):



Step 2. Tune their positions:

Update the new vertex:

$\frac{3}{8}(A+B)+\frac{1}{8}(C+D)$

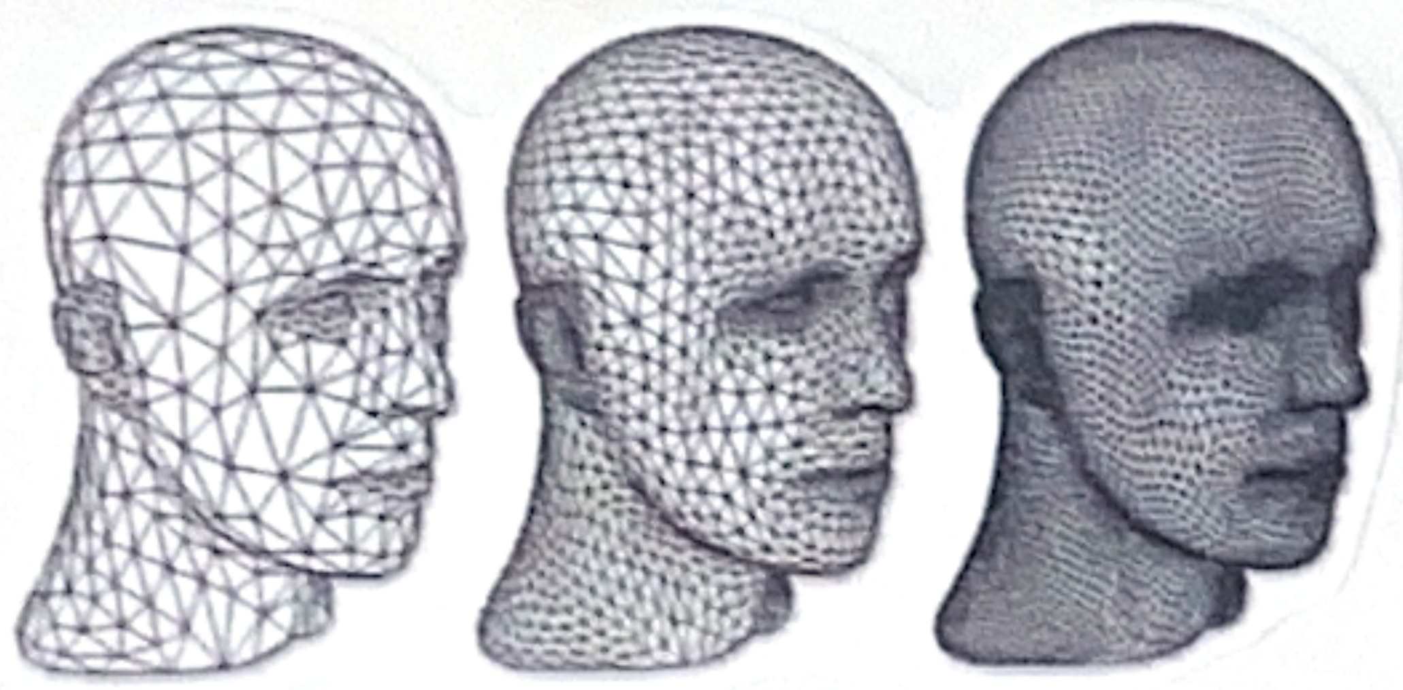
Update the old vertex:

$(1-n\cdot u)\cdot$ original_position + $u\cdot$ neighbor_position_sum

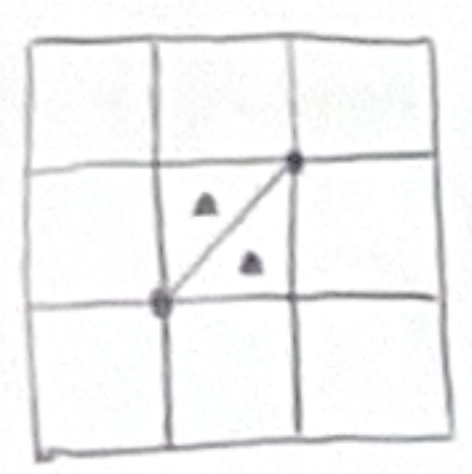
n: vertex degree (加點連接幾條邊)

$u = \frac{3}{16}$, $n=3$; $\frac{3}{8n}$, otherwise

Result:



* Catmull-Clark subdivition: for general meshes

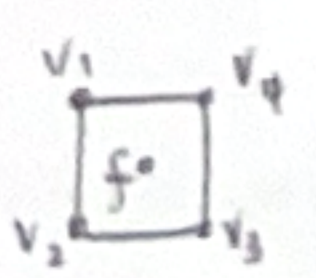


Add vertex in each face
Add midpoint on each edge
Connect all the vertices

▲ Non-square face
• Extraordinary vertex (degree ≠ 4) 奇異點

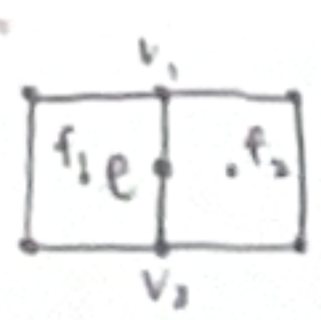
After one subdivision:
$N_{\blacktriangle} + N_{\bullet}$ extraordinary points
$n=3$ $n=5$
No more non-square face.

Update face point



$f=\frac{1}{4}(v_1+v_2+v_3+v_4)$

Update edge point

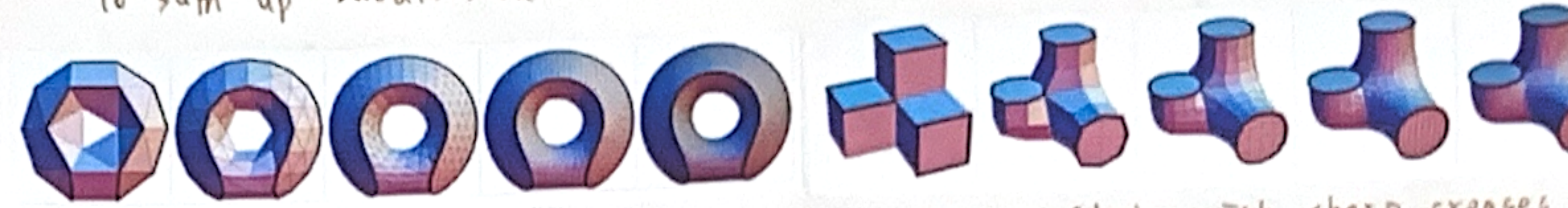


$e=\frac{1}{4}(v_1+v_2+f_1+f_2)$

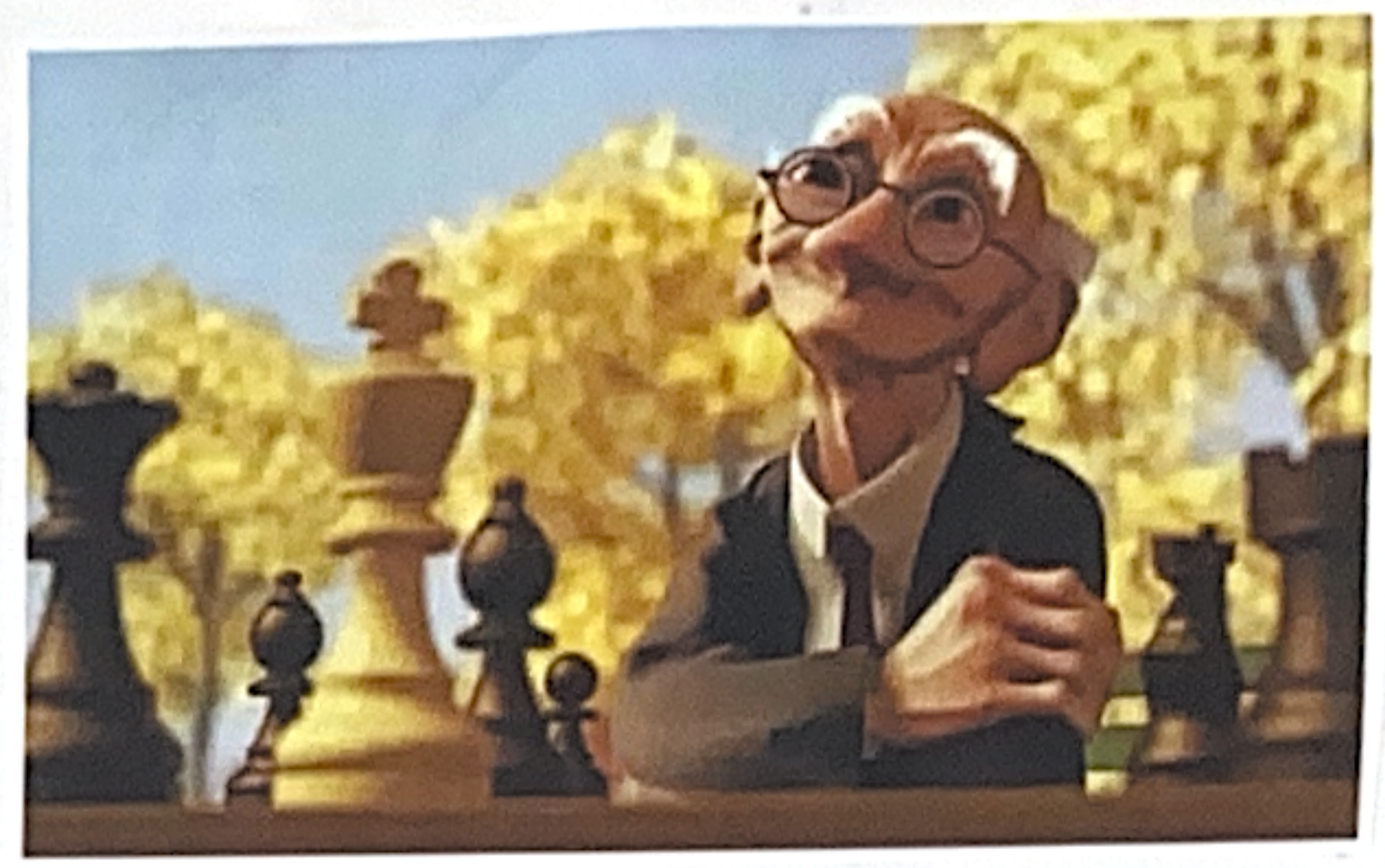
Update vertex point

$v=\frac{1}{16}(f_1+f_2+f_3+f_4+2(m_1+m_2+m_3+m_4)+4p)$ ← old vertex point

To sum up subdivision:



Loop with sharp creases   Catmull-Clark with sharp creases



Subdivision in action (Pixar's "Geri's Game")

2. Simplification:

Goal: reduce number of mesh elements while maintaining the overall shape

Use edge collapsing!!! (邊坍縮) ⟶ Method: Quadric error metrics 二次誤差度量

by average

by error quadric

The new vertex should minimize its sum of square distance to previously related triangle planes!

⇒ Iteratively collapse edges:

Step 1. which edges? Assign score with quadric error metric by approximating distance to surface as sum of distances to planes containing triangles.

Step 2. iteratively collapse edge with smallest score. (→ use stack in data structure)

Step 3: greedy algorithm... → great result!

RT1

- Ray Tracing:

The properties of rasterization:

1. It couldn't handle global effects well, such like (soft shadow) or especially when the light bounces more than once
2. Rasterization is fast, but quality is relatively low.

The properties of ray tracing:

1. ~10k CPU core hours to render one frame in production.
2. Ray tracing is accurate, but very slow.

▲ Rasterization: real-time

▲ ray tracing: offline

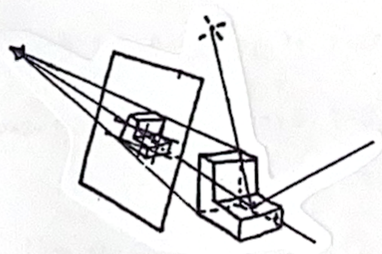
※ Basic Ray-Tracing Algorithm:

* About light rays:

1. Light travels in straight lines
2. Light rays do not "collide" with each other if they cross.
3. Light rays travel from the light sources to the eye
   → but the physics is invariant under path reversal - reciprocity.

"And if you gaze along into an abyss, the abyss also gazes into you" —— Friedrich Wilhelm Nietzsche 尼采

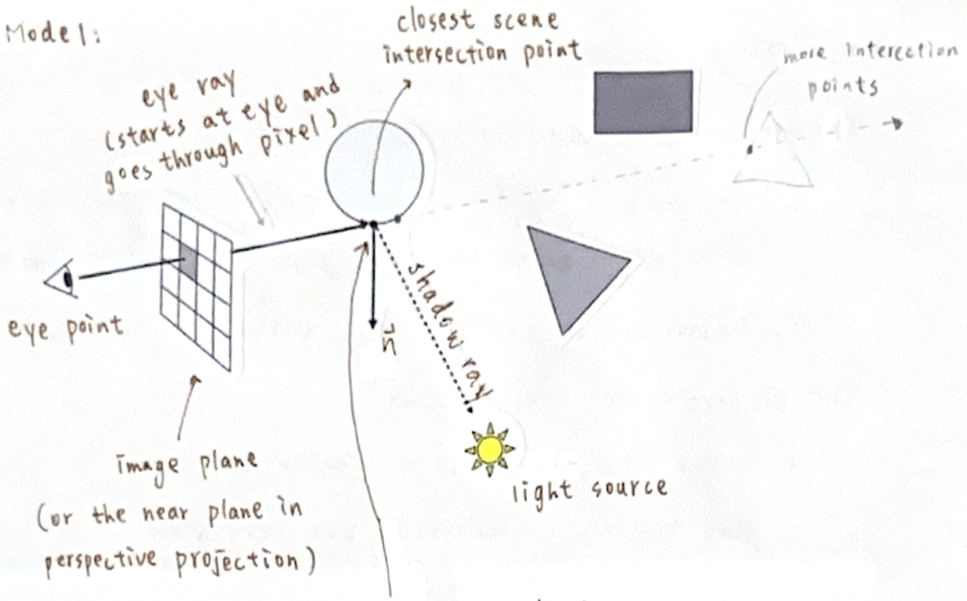
* Ray casting: (Generating eye rays)



step.1. Generate an image by casting one ray per pixel

Step 2. Check for shadows by sending a ray to the light

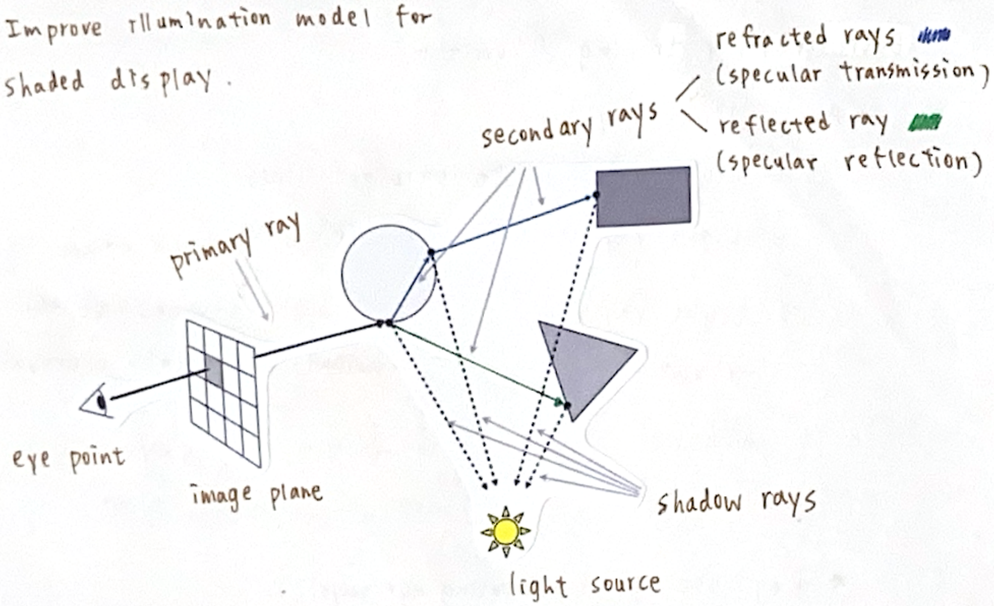
Pinhole Camera Model:



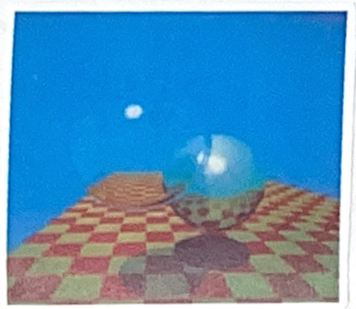
→ perform shading calculation here to compute color of pixel (e.g. Blinn Phong model)

★ Recursive (Whitted-Style) ray tracing:

⇒ Improve illumination model for shaded display.



Result:



Sphere and checkerboard, T. Whitted, 1979

The time of generating one frame:

VAX 11/780 (1979): 74 minutes

PC (2006): 6 seconds

GPU (2012): $\frac{1}{10}$ seconds

RT2

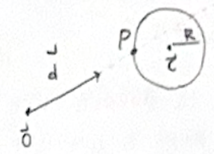
∴ Ray-Surface Intersection:

Ray equation: $\vec{r}(t) = \vec{o} + t\vec{d}$, $0 \le t < \infty$ (time)

($\vec{r}$: point along ray; $\vec{o}$: origin; $\vec{d}$: (normalized) direction)

Sphere equation: $\vec{p}: (\vec{p} - \vec{c})^2 - R^2 = 0$

→ The intersection p must satisfy both ray equation and sphere equation.



General implicit surface: $\vec{p} = f(\vec{p}) = 0$

⇒ solve $f(\vec{o} + t\vec{d}) = (\vec{o} + t\vec{d} - \vec{c})^2 - R^2 = 0$

and find t is real and posrtive.

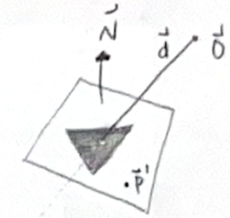
Ray intersection with triangle mesh

Why? Rendering = visibility, shadows, lighting .....

Geometry: inside/outside test

Step 1. Ray-plane intersection

Step 2. Test if hit point is inside triangle



Plane equation: $\vec{p}: (\vec{p} - \vec{p}') \cdot \vec{N} = 0$

($\vec{p}$: all point on the plane; $\vec{p}'$: one point on the plane; $\vec{N}$: normal vector)

→ Solve $f(\vec{p}) = f(\vec{o} + t\vec{d}) = (\vec{o} + t\vec{d} - \vec{p}') \cdot \vec{N} = 0$

→ $t = \dfrac{(\vec{p}' - \vec{o}) \cdot \vec{N}}{\vec{d} \cdot \vec{N}}$, and check $0 \le t < \infty$

▲ Möller Trumbore Algorithm:

A faster approach, giving barycentric coordinate directly.

$$\underbrace{\vec{o} + t\vec{D}}_{\text{ray}} = \underbrace{(1 - b_1 - b_2)\vec{P}_0 + b_1\vec{P}_1 + b_2\vec{P}_2}_{\text{重心座標}}$$

→ solve $\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{S}_1 \cdot \vec{E}_1} \begin{bmatrix} \vec{S}_2 \cdot \vec{E}_2 \\ \vec{S}_1 \cdot \vec{S} \\ \vec{S}_2 \cdot \vec{D} \end{bmatrix}$, where $\begin{cases} \vec{E}_1 = \vec{P}_1 - \vec{P}_0 \\ \vec{E}_2 = \vec{P}_2 - \vec{P}_0 \\ \vec{S} = \vec{O} - \vec{P}_0 \\ \vec{S}_1 = \vec{D} \times \vec{E}_2 \\ \vec{S}_2 = \vec{S} \times \vec{E}_1 \end{cases}$ → check $\begin{cases} 0 \le t < \infty \\ 0 \le b_1 < 1 \\ 0 \le b_2 < 1 \\ 0 \le 1 - b_1 - b_2 < 1 \end{cases}$

Accelerating Ray-Surface Intersection:

The problem of simple-scene intersection:

Exhaustively test ray-intersection with every triangle

→ Naive algorithm = # pixels × # triangles (× # bounces)

→ Very SLOW!!

⇒ For generality, we use the term objects instead of triangles.

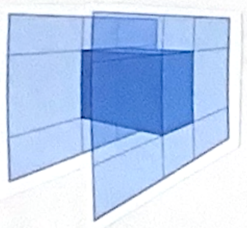
* Bounding Volumes:

Quick way to avoid intersections: bound complex object with a simple volume.

Object is fully contained in the volume

⇒ If it doesn't hit the volume, it doesn't hit the object. Therefore, test bounding volume first, then test object if it hits.

⇒ Ray-Intersection with Box



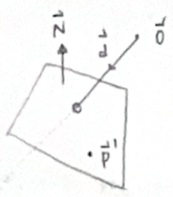
It is the intersection of 3 pairs of slabs.

Specifically: We often use an Axis-Aligned Bounding Box (AABB). (軸對齊包圍盒)

Any side of the Bounding Box is along either x, y or z axis.

- Why Axis-Aligned?

General



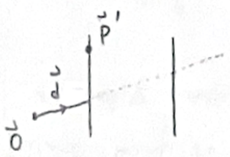
r(t)

$$t = \frac{(\vec{p}' - \vec{O}) \cdot \vec{N}}{\vec{d} \cdot \vec{N}}$$

3 subtractions, 6 multiplies, 1 division

→

Slabs perpendicular to x-axis

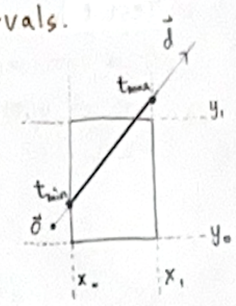


$$t = \frac{p'_x - O_x}{d_x}$$

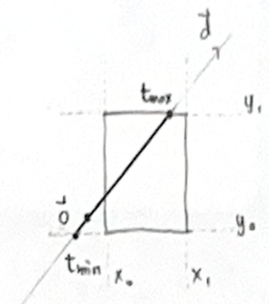
1 subtraction, 1 division

⇒ Ray Intersection with Axis-Aligned Box

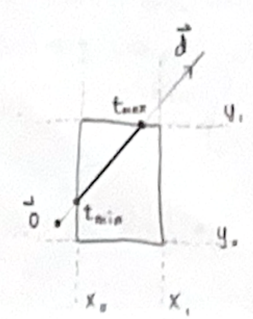
Compute intersections with slabs and take intersection of t_{min}/t_{max} intervals.



Intersection with x plane.



Intersection with y plane.



Final intersection result. (2D)

{ The ray enters the box only when it enters all pairs of slabs.
{ The ray exits the box as long as it exists any pair of slabs.

for 3D → { $t_{min,x}$, $t_{min,y}$, $t_{min,z}$ (negative is fine); $t_{max,x}$, $t_{max,y}$, $t_{max,z}$ } → { $t_{enter} = \max(t_{min})$; $t_{exit} = \min(t_{max})$ }

→ if $t_{enter} < t_{exit}$, we know the ray stays a while in the box.

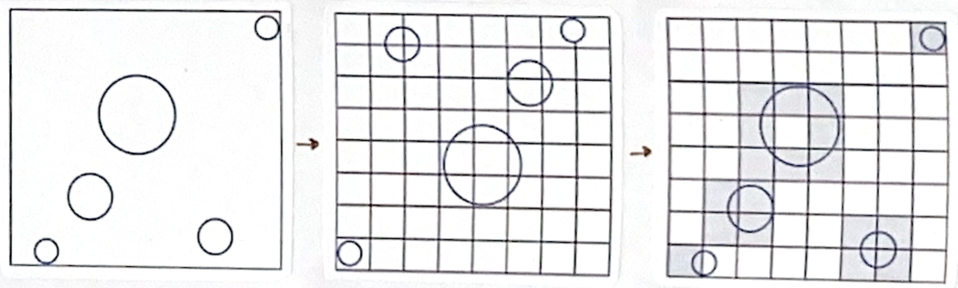
- Check whether t is negative for physical correctness!
- If $t_{exit} < 0$, the box is "behind" the ray → no intersection!
- If { $t_{exit} > 0$ & $t_{enter} < 0$, the ray's origin is inside the box → have intersection!

→ Ray and AABB intersect if $t_{enter} < t_{exit}$ & $t_{exit} \geq 0$.

* Use AABBs to accerlerate ray tracing:

1. Uniform spatial patitions grids:

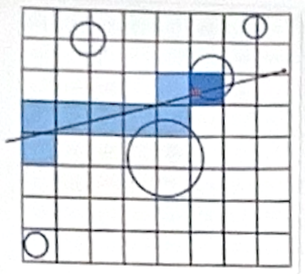
Step 1. Build acceleration grid



a. Find bounding box → b. Create grid → c. Store each object in overlapping cells

Step.2 Ray-Scene intersection

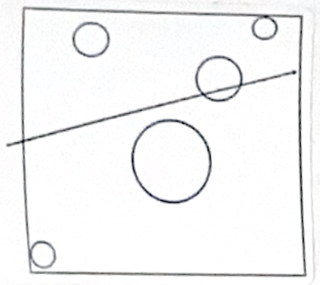
a. Step through grid in ray traversal order.



b. For each grid cell:
Test intersection with all objects stored at that cell.

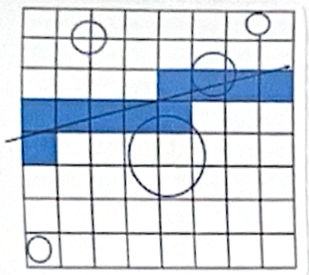
▲ Grid resolution:

One cell



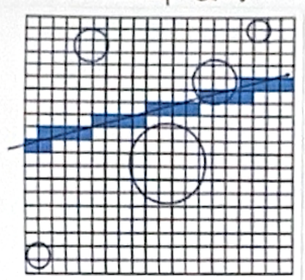
No speed up !!

Heuristic



#cells = C × #objs
C ≈ 27 in 3D

Too many cells



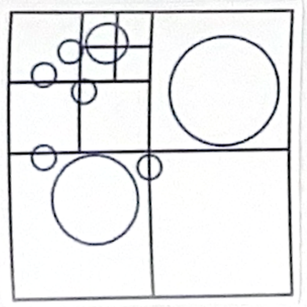
Inefficiency due to extraneous grid traversal !!

▲ Results:

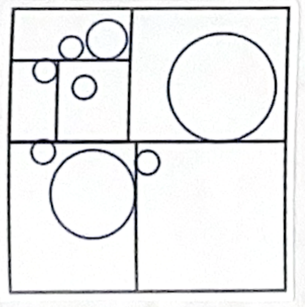
Grids work well on large collection of objects that are distributed evenly in size and space.

"Teapot in a stadium" PROBLEM!!

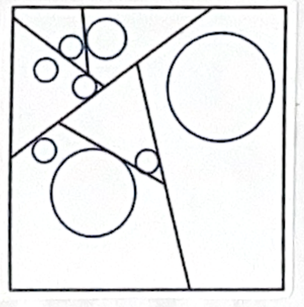
2- Spatial partitions:



Oct-Tree



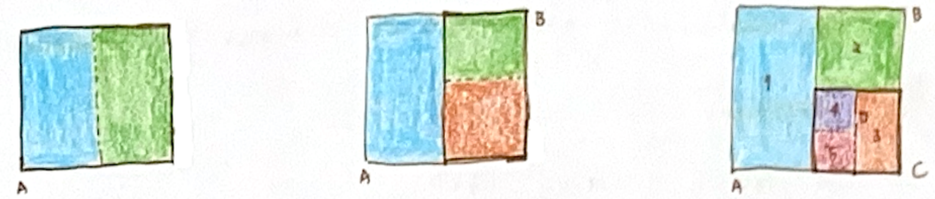
KD-Tree



BSP-Tree

★ KD-Tree

Step.1 KD-Tree Pre-processing



▲ Data structure for KD-Trees

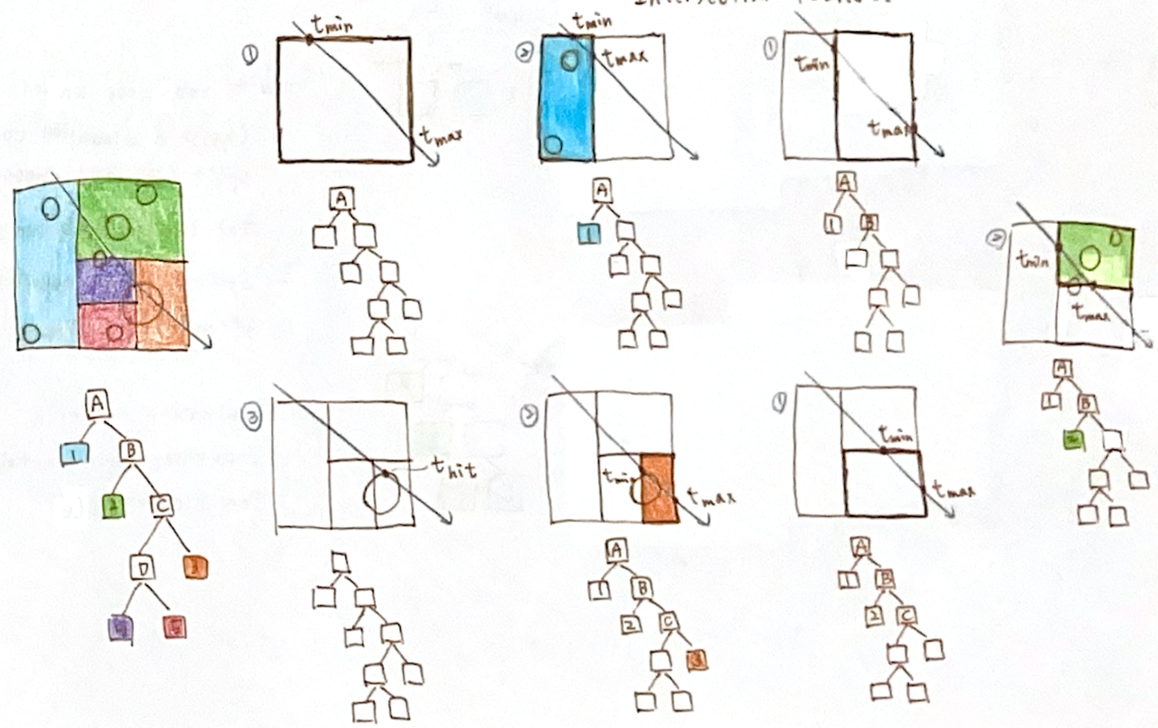
1. Internal node store
   a. split axis: x-, y-, or z-axis
   b. split position: coordinate of split plane along axis
   c. children: pointers to children nodes

   NOTICE!! NO objects are stored in internal nodes.

2. Leaf node store
   a. list of objects.

Step.2 Tranversing a KD-Tree

- Internal node: split ①
- Leaf node: intersect all objects ②
- Intersection found!! ③



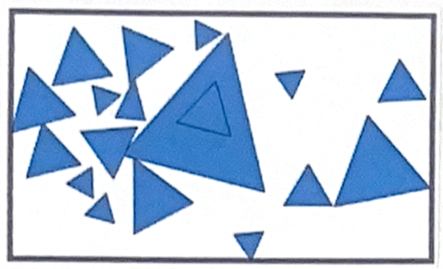
▲ KD-Trees problems:

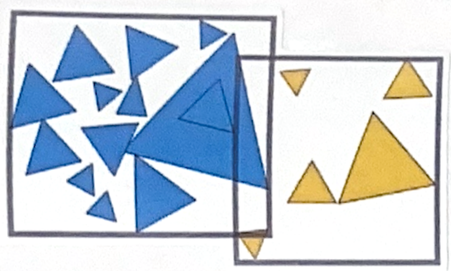
1. An object may exist in multiple bounding boxes.

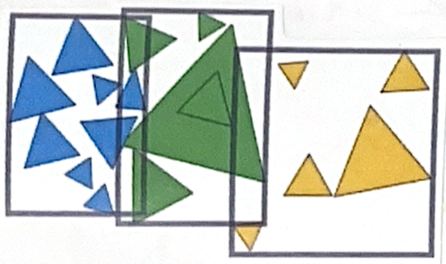
2. Must consider the intersection of triangles and bounding boxes.

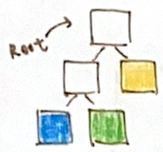
3. Object partitions:

* Bounding Volume Hierarchy (BVH)

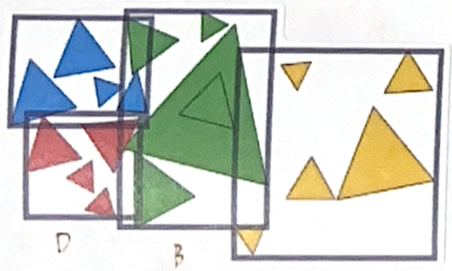


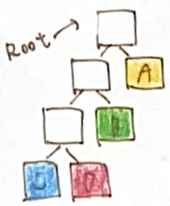






C D B A





▲ Building BVHs:

1. Finding bounding box.
2. Recursively split set of objects in two subsets.
3. Recompute the bounding box of the subsets.
4. Stop when necessary
5. Store objects in each leaf node.

How to subdivide a node?

1. Choose a dimension to split (Always choose the longest axis on node)
2. Split node at location of median object.

3. Termination criteria: stop when node contains few elements.

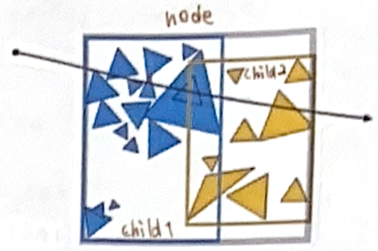
RTS

▲ Data structure for BVHs:

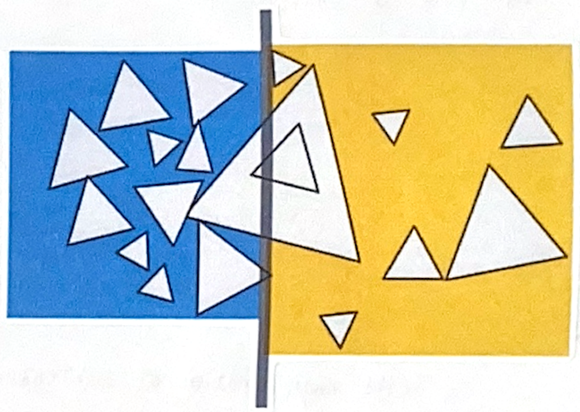
1. Internal nodes store { Bounding box; children: pointers to child nodes }
2. Leaf nodes store { Bounding box; List of objects }

3. coding (in C++):

```
Intersect (Ray ray, BVH node){
    if (ray misses node.bbox) return;
    if (node is a leaf node)
        test intersection with all objs;
        return closest intersection;
    hit1 = Intersect (ray, node.child1);
    hit2 = Intersect (ray, node.child2);
    return the closer of hit1, hit2;
}
```

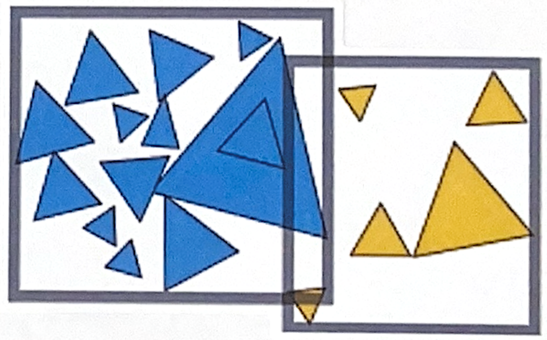


To summary:



↗ Spatial partition (e.g. KD-tree):

1. Partition space into non-overlapping regions
2. An object can be contained in multiple regions.



↗ Object partition (e.g. BVH):

1. Partition set of objects into disjoint subsets.
2. Bounding boxes for each set may overlap in space.

✱ Radiometry：(輻射度量學)

Why? In Blinn-Phong model, light intensity I is 10, for example.

But what is 10?

⇒
- Measurement system and units for illumination.
- Accurately measure the spatial properties of light.
  ex. Radiant flux, intensity, irradiance, radiance.
- Perform lighting calculation in physically correct manner.

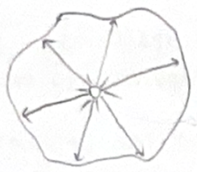
1. Radient energy and flux (Power):

Definition:

- Radient energy = Q (J = Joule) — the energy of electromagnetic radiation. (CG 不常用)
- Radient flux (power): the energy emitted, reflected, transmitted or recieved, per unit time.

"$\Phi \equiv \frac{dQ}{dt}$ (lm = lumen, W = Watt)"

2. Radient intensity: Light emitted from a source.



Definition: the power per unit solid angle emitted by a point light source.

"$I(w) \equiv \frac{d\Phi}{dw}$ ($\frac{W}{sr}$ or $\frac{lm}{sr}$ = cd = candela)" — one of SI base unit 基本單位

Angle: ratio of subtended arc length on circle to radius. (circle has 2π radians)

$\theta = \frac{l}{r}$

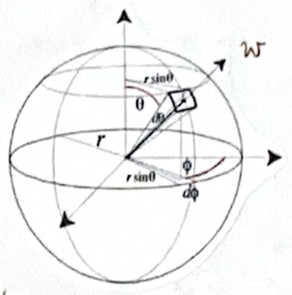
Solid angle: ratio of subtended area on sphere to radius square. (sphere has 4π steradians)

$\Omega = \frac{A}{r^2}$

▲ Differential of solid angle:

$dA = (r\,d\theta)(r\sin\theta\,d\phi) = r^2\sin\theta\,d\theta\,d\phi$

$dw = \frac{dA}{r^2} = \sin\theta\,d\theta\,d\phi$ → 對θ來說不均勻



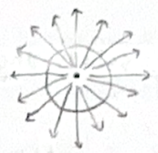
Sphere: $S^2 \rightarrow \Omega = \int_{S^2} dw$

$= \int_0^{2\pi}\int_0^{\pi} \sin\theta\,d\theta\,d\phi$

$= 4\pi$

↳ use w to denote a direction vector (unit length)

▲ Radient intensity example: Isotropic point source



Radient flux on the whole sphere:

$\Phi = \int_{S^2} I\,dw^2 = 4\pi I$

⇒ $I(w) = \frac{\Phi}{4\pi}$

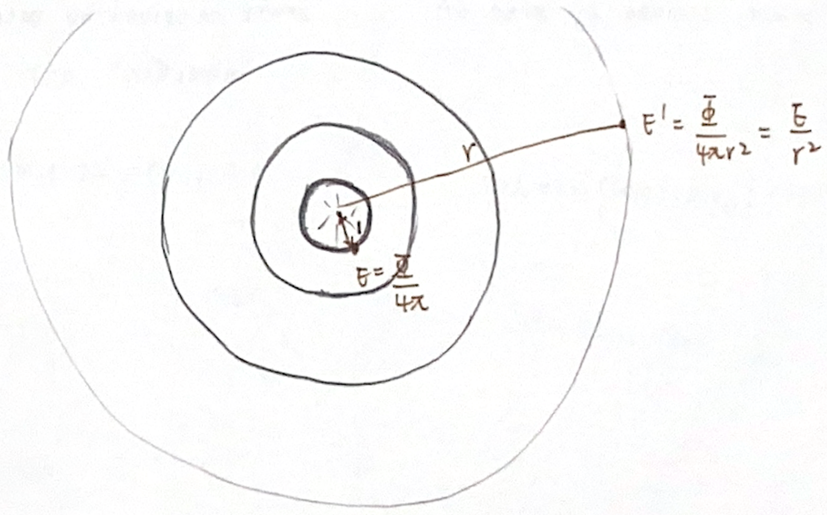
3. Irradiance: Light falling on a surface

Definition: the power per unit area incident on a surface point.

"$E(x) \equiv \frac{d\Phi(x)}{dA}$ ($\frac{W}{m^2}$ or $\frac{lm}{m^2}$ = lux)"

▲ Irradiance example:

Assume light is emitting power Φ in a uniform angular distribution, compare irradiance at surface of two spheres:



4. Radiance : light traveling along a ray

※ Rendering is all about computing radiance

Definition: the power emitted, reflected, transmitted or received by a surface, per unit solid angle, per projected unit area.

$$L(p,w) \equiv \frac{d^2\Phi(p,w)}{dw\, dA \cos\theta} \quad \left(\frac{W}{sr\,m^2} \text{ or } \frac{cd}{m^2} = \frac{lm}{sr\,m^2} = nit\right)$$

accounts for projected surface area

a. Incident Radiance: the irradiance per unit solid angle arriving at the surface.

$$L(p,w) = \frac{dE(p)}{dw\cos\theta}$$

↳ the light arriving at the surface along a given ray (point on surface and incident direction)

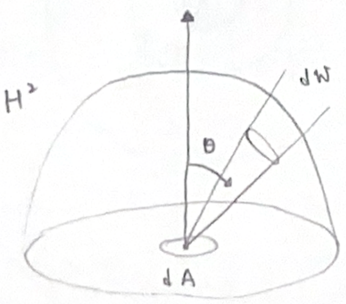
b. Exiting Radiance: the intensity per unit projected area leaving the surface.

$$L(p,w) = \frac{dI(p,w)}{dA\cos\theta}$$

↓ for an area light it is the light emitted along a given ray (point on surface and exit direction)

Irradiance	Radiance
total power received by area dA	power received by area dA from "direction" dw
$E(p) = \int_{H^2} L_i(p,w)\cos\theta\, dw$	$dE(p,w) = L_i(p,w)\cos\theta\, dw$

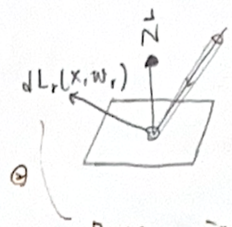
Unit hemisphere: H^2



RTT

★ Bidirection reflectance distribution function: (BRDF)
雙向反射分佈函數

Reflection at a point:



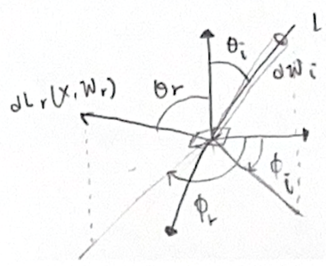
Differential irradiance incoming: Radiance from direction w_i turns into the power E that dA recieves.

$$dE(w_i) = L(w_i)\cos\theta_i\, dw_i$$

Differential radiance exiting (due to $dE(w_i)$): the power E will become the radiance to any other direction w_o.

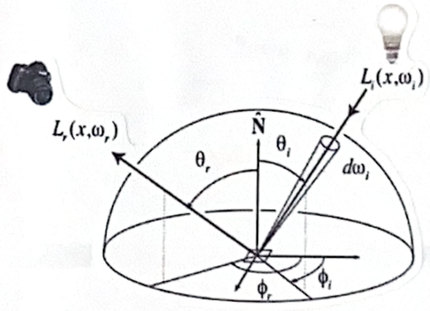
$$dL_r(w_r)$$

⇒ BRDF represents how much light is reflected into each outgoing direction w_r from each incoming direction.



$$f_r(w_i \to w_r) = \frac{dL_r(w_r)}{dE_i(w_i)} = \frac{dL_r(w_r)}{L_i(w_i)\cos\theta_i\, dw_i} \quad \left(\frac{1}{sr}\right)$$

⇒ Therefore, the reflection equation is



$$L_r(p,w_r) = \int_{H^2} f_r(p,w_i \to w_r)\, L_i(p,w_i)\cos\theta_i\, dw_i$$

1. Reflected radiance depends on incoming radiance
2. Incoming radiance depends on reflected radiance (at another point in the scene)

⇒ Last, adding emission term, it become rendering equation

$$L_o(p,w_o) = L_e(p,w_o) + \int_{\Omega^+} L_i(p,w_i)\, f_r(p,w_i,w_o)\,(n\cdot w_i)\, dw_i$$

★ NOTE!! From now, we assume that all directions are pointing outwards.

▲ Solve rendering equation

$$\underbrace{L_r(x,\omega_r)}_{\text{UNKNOWN}} = \underbrace{L_e(x,\omega_r)}_{\text{Emission}} + \int_{\Omega} \underbrace{L_r(x',-\omega_i)}_{\text{UNKNOWN}} \underbrace{f(x,\omega_i,\omega_r)}_{\text{BRDF}} \cos\theta_i \, d\omega_i$$

Reflected light (Output image)

Reflected light

$$\downarrow$$

$$l(u) = e(u) + \int l(v)\, K(u,v)\, dv$$

Kernal of equation
Light transport operator

$$\downarrow$$

$L = E + KL$ Matrix equation

L, E : matrix

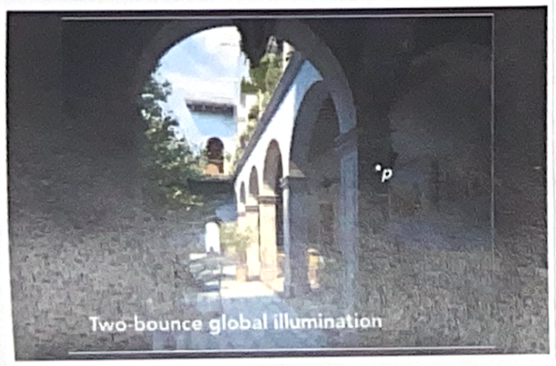
K : light transport matrix

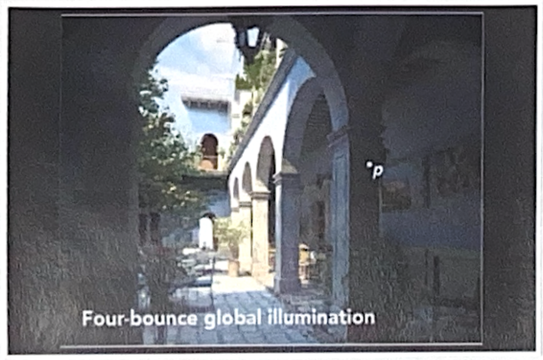
$$L = (I-K)^{-1} E$$

$$L = (I + K + K^2 + K^3 + \cdots)E = E + KE + K^2E + K^3E + \cdots$$

Shading in rasterization [

- E : Emission directly from light sources
- KE : Direct illumination on surfaces
- K^2E : Indirect illumination (One bounce indirect)
- K^3E : (Two bounce indirect illumination)



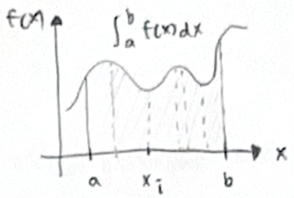


RT 8

▲ Monte Carlo Integration

Why? we want to solve an integral, but it can be too difficult to solve analytically.

How? estimate the integral of a function by averaging random samples of the function's value.



Define integral $\int_a^b f(x)\,dx$

Random variable $X_i \sim p(x)$

⇒ Monte Carlo estimator $F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$

Example : Uniform Monte Carlo estimator

Define integral $\int_a^b f(x)\,dx$

Uniform random variable $X_i \sim p(x) = C = \frac{1}{b-a}$

⇒ Basic Monte Carlo estimator $F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i)$

To sum up, $\int f(x)\,dx = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$, $X_i \sim p(x)$

① The more samples, the less variance.

② Sample on x, integrate on x.

★ Path tracing :

preview about Whitted-style ray tracing:

1. Always perform specular reflections / refractions.
2. Stop bouncing at diffuce surfaces.

problem 1.

Mirror reflection

Glossy reflection

The Utah teapot

problem 2.

Path traced: direct illumination

Path traced: global illumination

The Cornell box

▲ Solving the path tracing coding step by step : 遞歸求解問題

Because the Whitted-Style ray tracing is wrong, the rendering equation $L_o(p,w_o) = L_e(p,w_o) + \int_{\Omega^+} L_i(p,w_i)\, f_r(p,w_i,w_o)(n\cdot w_i)\, dw_i$ is correct. It involves

(1) Solving an integral over the hemisphere

(2) Recursive execution.

Suppose we want to render one pixel (point) in the following scene for direct illumination only

$L_o(p,w_o) = \int_{\Omega^+} L_i(p,w_i)\, f_r(p,w_i,w_o)(n\cdot w_i)\, dw_i$

① $f(x)$ in Monte Carlo

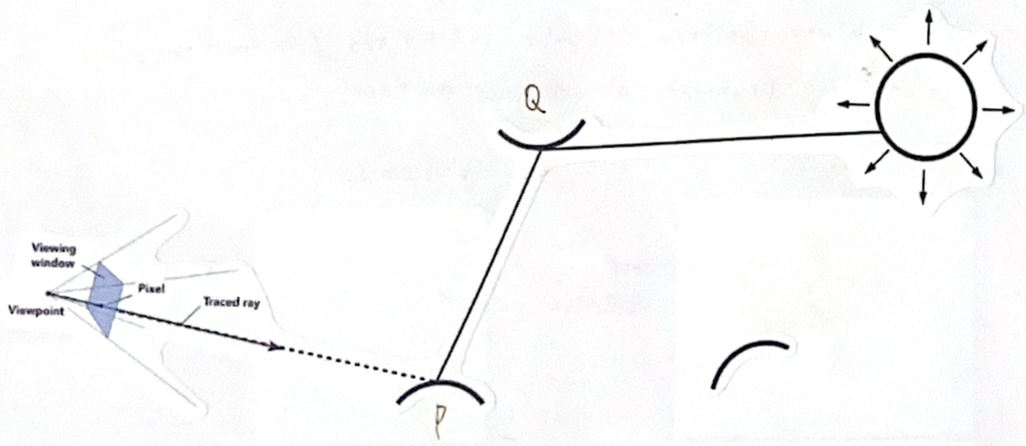
② integration over directions → Monte Carlo integration !!

③ $p(w_i) = \frac{1}{2\pi}$, because of assuming uniformly sampling the hemisphere.

$$\approx \frac{1}{N}\sum_{i=1}^{N} \frac{L_i(p,w_i)\, f_r(p,w_i,w_o)(n\cdot w_i)}{p(w_i)}$$

(code in next page without part A)

One more step forward global illumination



When the ray hit object Q directly, Q reflects light to p.

How much? The direct illumination at Q.

↓ code in the next page with part A

RTQ

Coding:

shade(p, wo)

Randomly choose N directions wi ~ pdf

Lo = 0.0

For each wi

Trace a ray r(p, wi)

If ray r hit the light

Lo += (1/N) * L_i * f_r * cosine / pdf(wi)

part A

Else If ray r hit an object at q

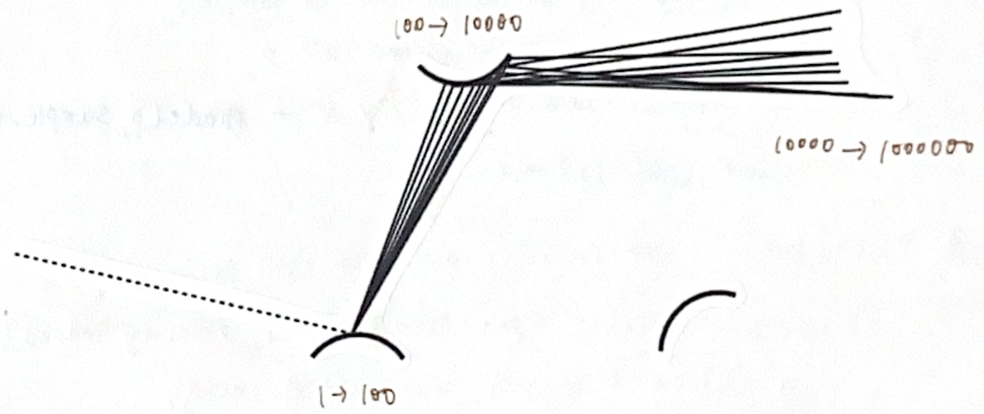
Lo += (1/N) * shade(q, -wi) * f_r * cosine / pdf(wi)

↳ NOTICE the direction!

Return Lo

PROBLEM 1 : Explosion of #rays as #bounces go up

→ #rays = $N^{\#bounces}$



SOLUTION : assume that only one ray is traced at each shading point.

Coding:

shade(p, wo)

Randomly choose ONE direction wi ~ pdf(w)

Trace a ray r(p, wi)

If ray r hit the light

Return L_i * f_r * cosine / pdf(wi)

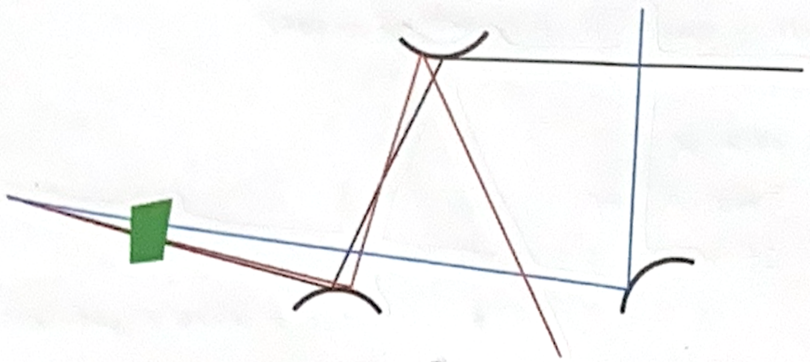
Else If ray r hit an object at q

Reture shade(q, -wi) * f_r * cosine / pdf(wi)

This is Path Tracing !!

From the following figure, the path tracing will be noisy in one pixel.

So, just trace more paths through each pixel and average their radiance!



Very similar to ray casting and ray tracing.

Coding:

```
ray-generation (camPos, pixel)
    Uniformly choose N sample positions within the pixel
    pixel_radiance = 0.0
    For each sample in the pixel
        Shoot a ray r(camPos, cam_to_sample)
        If ray r hit the scene at p
            pixel_radiance += 1 / N * shade(p, sample_to_cam)
    Return pixel_radiance
```

* PROBLEM 1: The recursive algorithm will never stop!

Dilemma: the REAL light does not stop bouncing indeed!

→ Cutting # bounces == cutting energy

3 Bounces

17 Bounces

RTIO

SOLUTION: Russian Roulette (RR)

It is about probability { 0 < P < 1, survival; 1 − P, died

Previously, we always shoot a ray at a shading point and get the shading result L_o. Suppose we manually set a probability $P(0 < P < 1)$

→ { with probability P: shoot a ray and return the shading result divided by P: $\frac{L_o}{P}$

with probability 1−P: don't shoot a ray and you'll get 0.

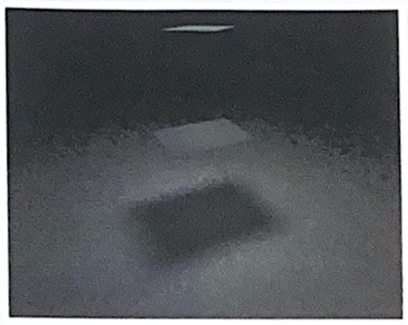
⇒ Expectation value: $E = P * \frac{L_o}{P} + (1-P) * 0 = L_o$

Coding:

```
shade (p, wo)
    Manually specify a probability P_RR
    Randomly select ksi in a uniform distance in [0, 1]
    If (ksi > P_RR) return 0.0;

    Randomly choose ONE direction wi ~ pdf
    Trace a ray r(p, wi)
    If ray r hit the light
        Return L_i * f_r * cosine / pdf(wi) / P_RR
    Else if ray r hit an object at q
        Return shade (q, -wi) * f_r * cosine / pdf(wi) / P_RR
```

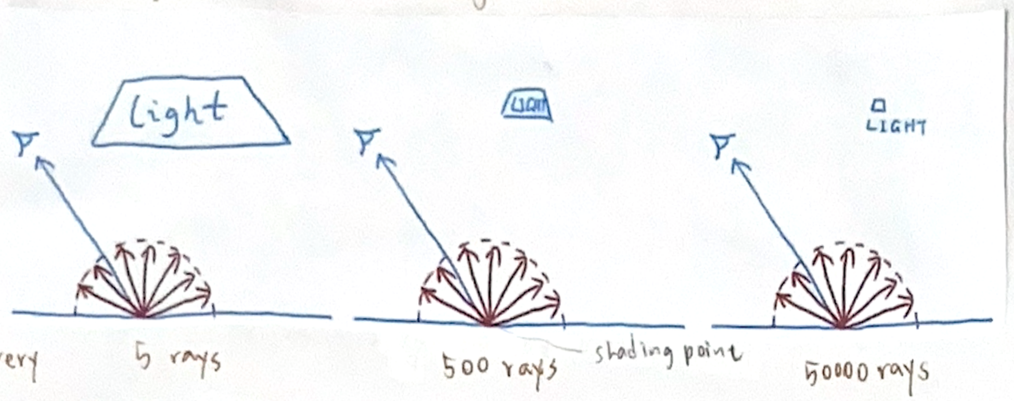
Now! we already have a correct version of path tracing, BUT not really efficient.



Low SPP

High SPP

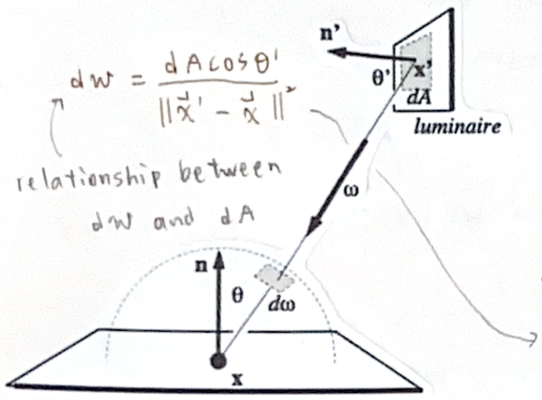
understanding the reason of being inefficient



there will be 1 ray hitting the light (according to the area of light source). So a lot of rays are "wasted" if we uniformly sample the hemisphere at the shading point.

→ SOLUTION: Sampling the light

Monte Carlo methods allow any sampling methods, so we can sample the light (therefore no rays are wasted)



$dw = \frac{dA \cos\theta'}{\|\vec{x}' - \vec{x}\|^2}$

relationship between dw and dA

Assume uniformly sampling on the light:

$pdf = \frac{1}{A}$, because of $\int pdf \, dA = 1$

However, the rendering equation integrates on the solid angle: $L_o = \int L_i f_r \cos \, dw$.

▲ Recall Monte Carlo method: sample on x & integrate on x

→ need to make the rendering equation as an integral of dA

Then, rewrite the rendering equation as

$$L_o(x, w_o) = \int_{\Omega^+} L_i(x, w_i) f_r(x, w_i, w_o) \cos\theta \, dw_i$$

$$= \int_A L_i(x, w_i) f_r(x, w_i, w_o) \frac{\cos\theta \cos\theta'}{\|\vec{x}' - \vec{x}\|^2} dA$$

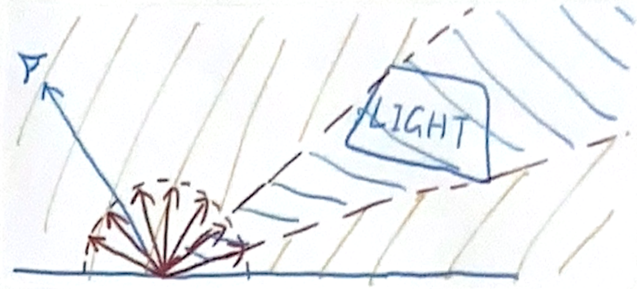
Now an integration on the light by Monte Carlo integration:

"f(x)": everything inside

pdf: $\frac{1}{A}$

Previously, we assume the light is "accidentally" shot by uniform hemisphere sampling. Now we consider the radiance coming from two parts:

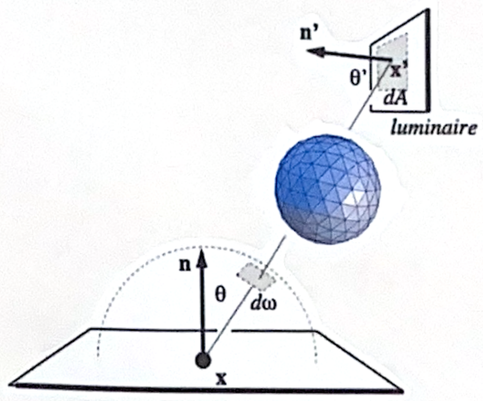
1. light source: direct, no need to have RR
2. other reflections: indirect, RR



Coding:

```
shade(p, wo)
    # contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cosθ * cosθ' / |x' - p|^2 / pdf_light

    # contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cosθ / pdf_hemi / P_RR
    Return L_dir + L_indir
```



One final thing. how do we know if the sample on the light is not blocked or not?

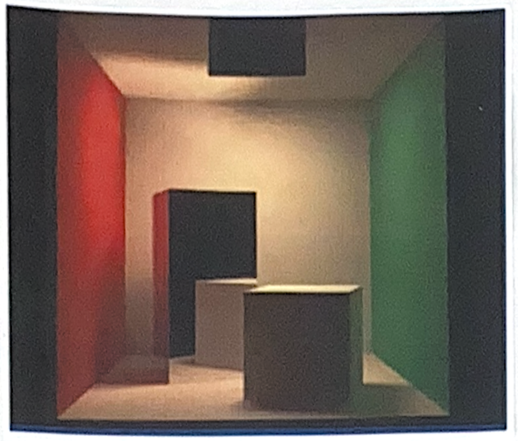
```
# contribution from the light source.
L_dir = 0.0
Uniformly sample the light at x' (pdf_light = 1/A)

Shoot a ray from p to x'
If the ray is not blocked in the middle
    L_dir = ....
```

NOW path tracing is finally done!

Path tracing is indeed difficult. It considers physics, probability, calculus, coding. However, it is almost 100% correct. It can generate the figure that is PHOTO-REALISTIC!

Photo



Path traced: global illumination