- Computer graphics : The use of computers to synthesize and manipulate visual information.

  Ex. Movie, Animations, Design (Ikea), Visualization, Virtual Reality (VR glasses), Augmented Reality (pokemon), Digital Illustration (photo shop), Simulation, Graphical User Interfaces, Typography

- Fundamental Intellectual Challenges :
  - Creates and interacts with realistic virtual world
  - Requires understanding of all aspects of physical world
  - New computing methods, displays, technologies
- Technical challenges :
  - Math of perspective projections, curves, surfaces
  - Physics of lighting and shading
  - Representing / operating shapes in 3D
  - Animation / simulation

- Main Topic:
  - Rasterization 柵格化
  - Curves and Meshes 幾合的部分...
  - Ray Tracing 光線追蹤
  - Animation / Simulation 動畫 / 模擬
- Graphics' Dependencies
  - Basic methematics : Linear algebra, calculus, statistics
  - Basic physics : Optics, Mechanics
  - Misc : Signal processing, Numerical analysis
  - aesthetics

• Vectors : def

properties : 1. Usually written as $\vec{a}$ or bold **a**

2. Use start and end point $\vec{AB} = B - A$

3. Direction and length

4. No absolute starting position

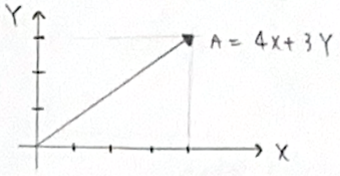
normalization : Magnitude of a vector written as $\|\vec{a}\|$

→ Unit vector : 1. A vector with magnitude of 1

2. Finding the unit vector of a vector : $\hat{a} = \frac{\vec{a}}{\|\vec{a}\|}$

3. Used to represent direction

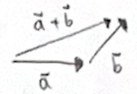
Cartesian Coordinates: X and Y can be any (usually orthogonal unit) vector



def $\vec{A} = \begin{pmatrix} x \\ y \end{pmatrix}$

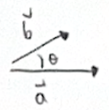
→ $\vec{A}^T = (x, y)$, $\|\vec{A}\| = \sqrt{x^2 + y^2}$

Vector addition :



Vector multiplication :

∴ Dot (scalar) product : def $\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos\theta$

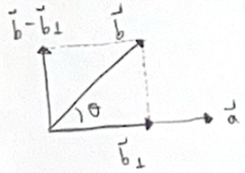


→ $\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} \overset{\text{unit vector}}{=\!=} \vec{a} \cdot \vec{b}$

↳ the angle between $\vec{a}$ & $\vec{b}$

$\vec{a} \cdot \vec{b} = \begin{pmatrix} x_a & y_a \end{pmatrix} \begin{pmatrix} x_b \\ y_b \end{pmatrix} = x_a x_b + y_a y_b$

★ Find the projection of $\vec{b}$ on $\vec{a}$:



$\vec{b}_\perp = k\vec{a}$, where $k = \|\vec{b}_\perp\| = \|\vec{b}\| \cos\theta$

properties : 1. Measure how close two directions are

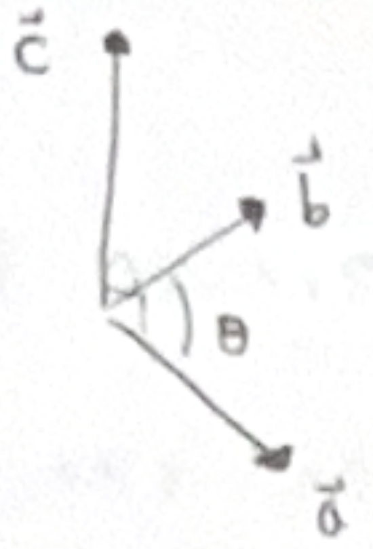
2. Decompose a vector ($\vec{b} < \vec{b}_\perp$, $\vec{b} - \vec{b}_\perp$)

3. Determine forward / backward

$\vec{a} \cdot \vec{b} > 0$ $\quad$ $\vec{a} \cdot \vec{b} < 0$

* Cross (vector) product : [def] $\vec{a} \times \vec{b} = -\vec{b} \times \vec{a} = \vec{c}$

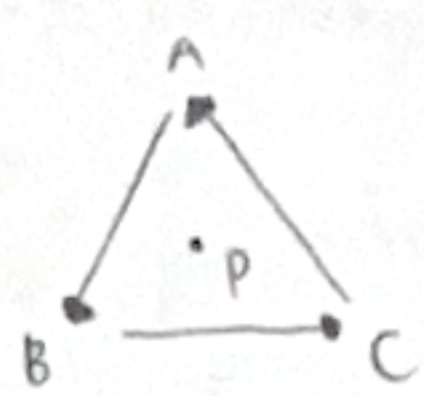
$\vec{a} \times \vec{b} = \vec{c}$

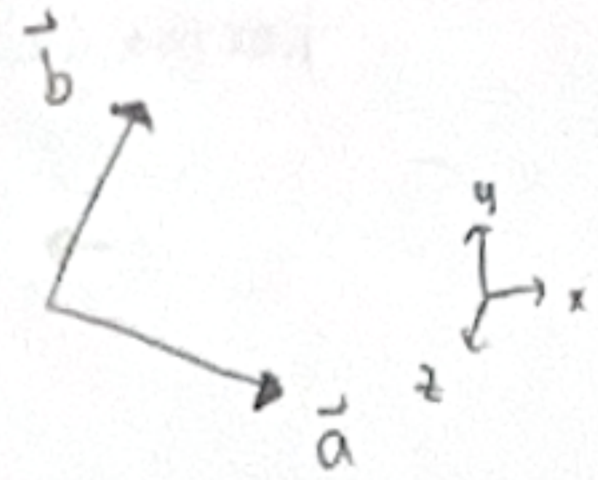


$\| \vec{a} \times \vec{b} \| = \| \vec{a} \| \| \vec{b} \| \sin\theta$

$\rightarrow \vec{a} \times \vec{b} = A^{*} b = \begin{pmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{pmatrix} \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix}$

★ Determine left / right, inside / outside :





Inside : $\overrightarrow{AB} \times \overrightarrow{AP}$, $\overrightarrow{BC} \times \overrightarrow{BP}$, $\overrightarrow{CA} \times \overrightarrow{CP}$ same sign

left : $\vec{a} \times \vec{b} \rightarrow +\hat{z}$

Outside : one has different sign

right : $\vec{a} \times \vec{b} \rightarrow -\hat{z}$

on the line : self-defined

* Orthonormal Coordinate frames :

- Any set of 3 vectors (in 3D) that
  1. $\| \vec{u} \| = \| \vec{v} \| = \| \vec{w} \| = 1$, unit vector
  2. $\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{w} = \vec{u} \cdot \vec{w} = 0$, perpendicular to each other
  3. $\vec{w} = \vec{u} \times \vec{v}$, by right-handed rule

$\rightarrow \vec{p} = (\vec{p} \cdot \vec{u})\vec{u} + (\vec{p} \cdot \vec{v})\vec{v} + (\vec{p} \cdot \vec{w})\vec{w}$

M1

- Matrix : [def] Array of numbers $m \times n$ (m → rows, n → columns)

ex. 3×2 matrix : $\begin{pmatrix} 1 & 3 \\ 5 & 2 \\ 0 & 4 \end{pmatrix}$

Matrix-Matrix Multiplication :

1. columns in A = rows in B

   ex. $(3 \times 2)(2 \times 4) = 3 \times 4$: $\begin{pmatrix} 1 & 3 \\ 5 & 2 \\ 0 & 4 \end{pmatrix} \begin{pmatrix} 3 & 6 & 9 & 4 \\ 2 & 7 & 8 & 3 \end{pmatrix} = \begin{pmatrix} 9 & 27 & 33 & 13 \\ 19 & 44 & 61 & 26 \\ 8 & 28 & 32 & 12 \end{pmatrix}$

2. Element (i, j) in the product is the dot product of row i from A and column j from B.

3. Non-commutative ($AB \neq BA$)

   Associative and distributive : $A(BC) = (AB)C$

   $A(B+C) = AB + AC$

4. Treat vector as a column matrix $m \times 1$: $\begin{pmatrix} x \\ y \end{pmatrix}$

Transpose of Matrix :

Switch rows and columns : $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

property : $(AB)^T = B^T A^T$

Identity Matrix and Inverses :

$I_{3\times3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

properties : 1. $AA^{-1} = A^{-1}A = I$

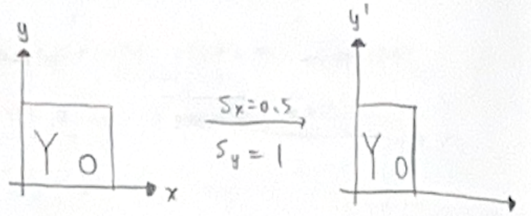
2. $(AB)^{-1} = B^{-1}A^{-1}$

- Transformation : { modeling / viewing

2D transformation :

※ Scale Matrix :

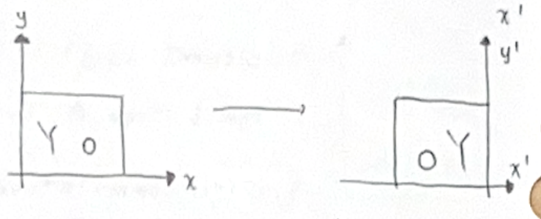
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



※ Reflection Matrix :

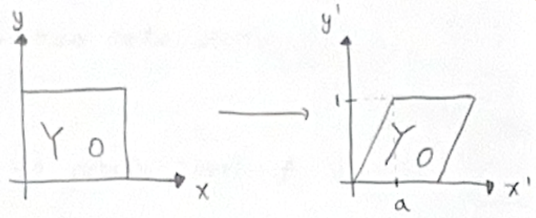
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Horizontal



※ Shear Matrix :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

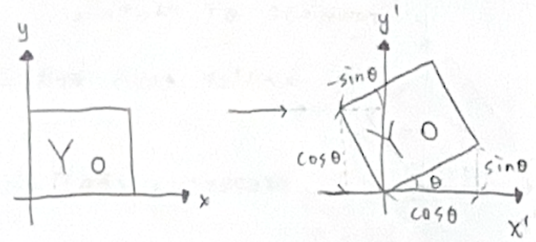


※ Rotate Matrix :

★ about origin (0,0)

★ counterclockwise

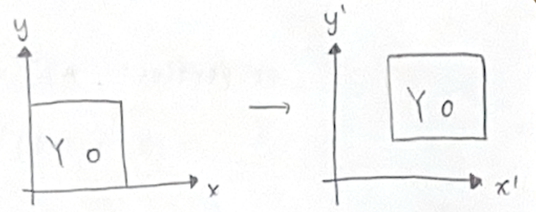
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



⇒ Linear transforms : $\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$ → $\vec{x}' = M\vec{x}$

※ Translation :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$



NOT Linear transform!

↳ Use homogenous coordinate !!

M2

Homogenous coordinates :

Add w-coordinate < 2D point = $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$, $\begin{pmatrix} x \\ y \\ w \end{pmatrix} \to \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ 1 \end{pmatrix}$, $w \neq 0$

2D vector = $\begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$

- point - point = vector
- vector + vector = vector
- point + vector = point
- point + point = point (midpoint)

※ Affine transformation :

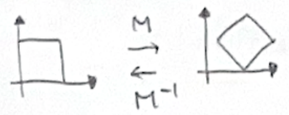
Affine map = linear map + translation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{M} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Use homogenous coordinates :

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + t_x \\ cx + dy + t_y \\ 1 \end{pmatrix}$$

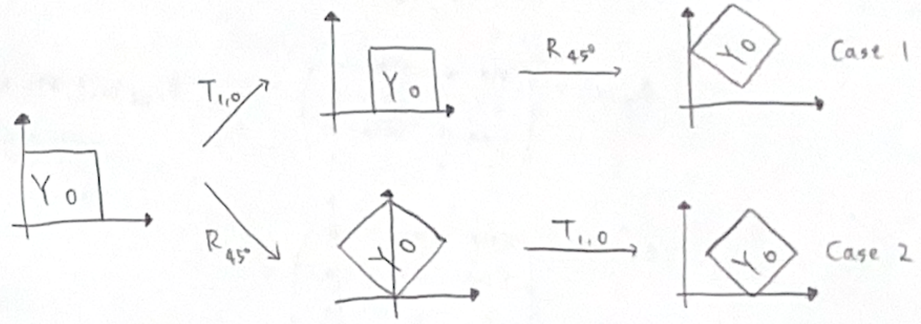
※ Inverse transform : M^{-1}



★ $R_\theta^{-1} = R_{-\theta} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = R_\theta^T$

Orthogonal matrix

※ Composing transform :



Matrices are applied right to left

Case 1 : $R_{45°} T_{1,0} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

Case 2 : $T_{1,0} R_{45°} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

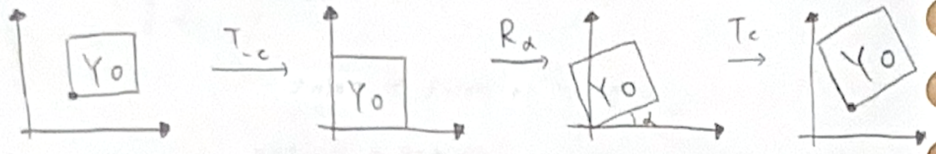
Sequence of affine transforms $A_1, A_2, A_3 \cdots$

$$A_n \cdot A_{n-1} \cdots A_2 \cdot A_1 \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = A \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Pre-multiply n matrices to obtain a single matrix representing combined transform

Decomposing complex transforms:

How to rotate around a given point c?



Matrix: $T_c R_\alpha T_{-c} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

3D transformation:

homogenous coordinates: 3D point = $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$, $\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \to \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$, $w \neq 0$

3D vector = $\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$

Affine transformations:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{linear } M} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Linear first, then translation

★ Rotation around x, y, z directions:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

⇒ $R_{xyz}(\alpha,\beta,\gamma) = R_x(\alpha) R_y(\beta) R_z(\gamma)$

Euler angles

Ex. roll, pitch, yaw

⇒ Rodrigues' rotation formula: (rotation by angle α around axis n)

$$R(n,\alpha) = \cos\alpha \cdot I + (1-\cos\alpha) nn^T + \sin\alpha \underbrace{\begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}}_{N}$$

Viewing transformation: 三維空間 → 二維圖片

How to take a photo?

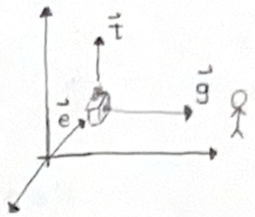
step 1: Find a good place and arrange people (model transformation)

step 2: Find a good "angle" to put the camera (view transformation)

step 3: Cheese! (projection transformation)

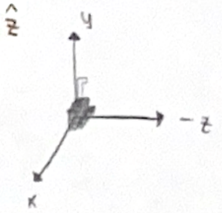
View / Camera transformation:

det camera: { position $\vec{e}$; Look at / gaze direction $\vec{g}$; Up direction $\vec{t}$



if the camera and object move together, the "photo" will be the same.

→ always transform the camera to { the origin (0,0,0) ; look at $-\hat{z}$; up at $\hat{y}$ } (Also the object) — Mview



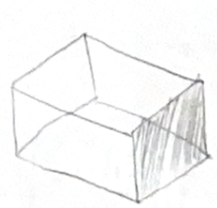
$M_{view} = R_{view} T_{view} = (R_{view}^{-1})^T T_{view}$ (先算平移再算旋轉)

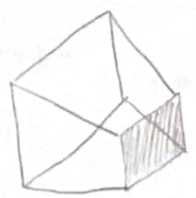
$$= \begin{pmatrix} x_{g\times t} & x_t & x_{-g} & 0 \\ y_{g\times t} & y_t & y_{-g} & 0 \\ z_{g\times t} & z_t & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} x_{g\times t} & y_{g\times t} & z_{g\times t} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

projection transformation:

Orthographic projection v.s. perspective projection





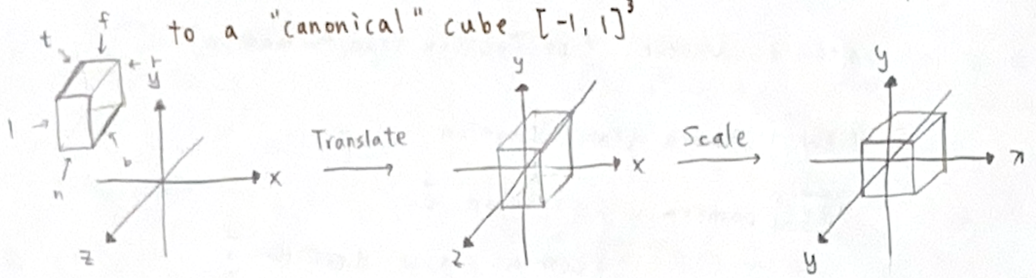
Orthographic projection:

A simple way to understand : drop z coordinate

Translate and scale the resulting rectangle to $[-1,1]^2$

→ In general, we map a cuboid $[\overset{\text{left right}}{l,r}] \times [\overset{\text{bottom top}}{b,t}] \times [\overset{\text{far near}}{f,n}]$ $f<n$

to a "canonical" cube $[-1,1]^3$



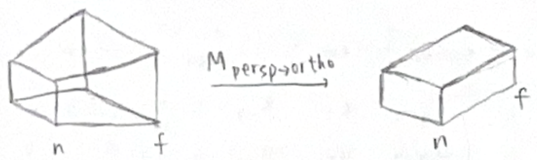
$$M_{ortho} = \text{scale} \cdot \text{translate} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

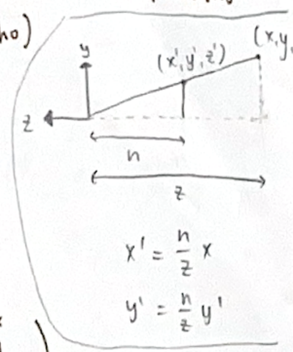
Perspective projection:

Step 1: "squish" the frustum into a cuboid ($M_{persp \to ortho}$)

Step 2: Do orthographic projection (M_{ortho})

→





In homogenous coordinate: $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \xrightarrow{M} \begin{pmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ \text{unknown} \\ 1 \end{pmatrix} == \begin{pmatrix} nx \\ ny \\ \text{unknown} \\ z \end{pmatrix}$

$$M_{persp \to ortho} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

By boundary condition: (1) any point on n will not change: $\begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} \xrightarrow{M} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} nx \\ ny \\ n^2 \\ n \end{pmatrix}$

(2) any point's z on f will not change: $\begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} \xrightarrow{M} \begin{pmatrix} 0 \\ 0 \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f^2 \\ f \end{pmatrix}$

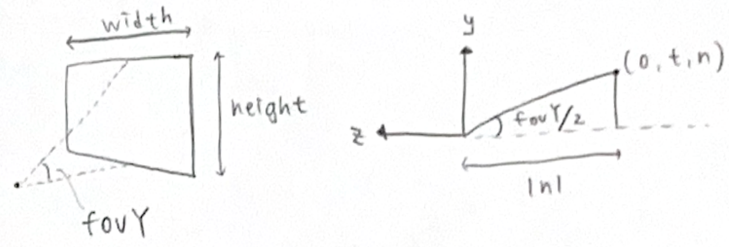
M operate on B.C. $\Rightarrow \begin{cases} An+B = n^2 \\ Af+B = f^2 \end{cases} \Rightarrow \begin{cases} A = n+f \\ B = -nf \end{cases}$

$$M_{persp} = M_{ortho} M_{persp \to ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

MVP

★ Choose the near plane: (assume symmetry $\begin{cases} l = -r \\ b = -t \end{cases}$

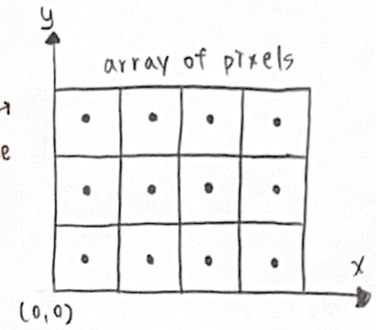
1. field of view: $\tan \frac{fovY}{2} = \frac{t}{|n|}$
2. aspect ratio: $aspect = \frac{r}{t}$



After MVP (model, view, projection transformations), SCREEN!!

Screen:
1. an array of pixels → { picture element; a little square with uniform number color is a mixture of red, green, blue
2. size of array: resolution
3. a typical type of raster display

rasterize: drawing onto the screen



- Pixels' indices are in the form of (x,y), where both x and y are integer.
- Pixels' indices are from $(0,0)$ to $(width-1, height-1)$
- Pixel (x,y) is centered at $(x+0.5, y+0.5)$
- The screen covers range $(0,0)$ to $(width, height)$

→ irrelevent z, and transform in xy plane:

$[-1,1]^2$ to $[0, width] \times [0, height]$

$$M_{viewpoint} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

R1

- Rasterization: Drawing to Raster Display

Triangles: fundamental shape primitives

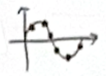
Good properties!!: 1. most basic polygon (break up other polygons)

2. guaranteed to be planar

3. well-defined interior

4. well-defined method for interpolating values at vertices over triangle (barycentric interpolation 重心插值)

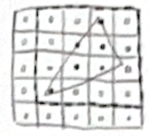
※ Sampling: discretize a function

Ex.  sampling a sine wave.

⇒ 2D Sampling:

Define binay function:

$$inside(t,x,y)=\begin{cases}1, & \text{point } (x,y) \text{ in triangle } t\\ 0, & \text{otherwise}\end{cases}$$

bounding box → faster in some case

Coding: (in C++)

```
for (int x=0; x < xmax; ++x)
  for (int y=0; y < ymax; ++y)
    image[x,y] = inside(tri, x+0.5, y+0.5);
```

use three cross products

 → Jaggies!! → Alisasing

Ex. (1) Jaggies: sampling in space

(2) Moire: undersampling images

(3) Wagon wheel effect: sampling in time

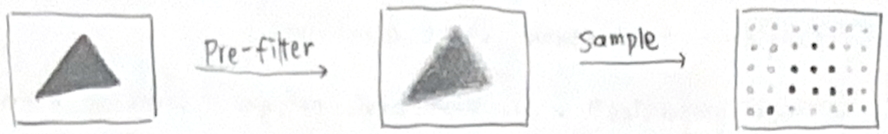
⋮

★ Why alisasing? Signals are changing to fast (high frequency) but sampled too slowly.

Supplement: Bayer filter

NO AFTER!!

Antialiasing Idea: Blurring (Pre-filtering) Before Sampling



Note antialiased edges in rasterized triangle where pixel values take intermediate values.

* Fourier transform:

$(e^{ix} = \cos x + i \sin x)$

spatial domain $f(x)$

$F(w) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i w x} dx$

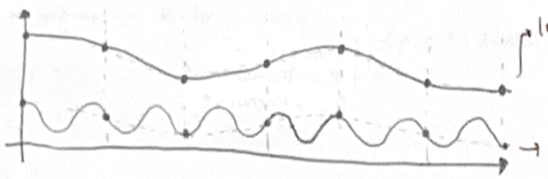
Fourier transform →

← Inverse transform

$f(x) = \int_{-\infty}^{\infty} F(w) e^{2\pi i w x} dw$

frequency domain $F(w)$

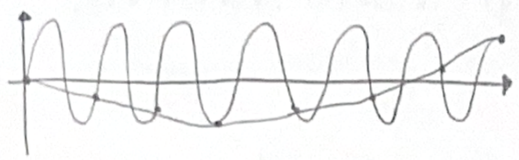
- high frequencies need faster sampling



low-frequency signal: sampled adequately for reasonable reconstruction

→ high-frequency signal: It is insufficiently sampled. Reconstruct incorrectly appears to be from a low frequency signal.

- Undersampling creates frequency aliases



→ High-frequency signal is insufficiently sampled: samples erroneously appear to be from a low-frequency signal

→ Two frequencies that are indistinguishable at a given sampling rate are called "aliases".

R2

* Filtering : Getting rid of certain frequency contents

"

Averaging

"

* Convolution :

Ex. Signal | 1 | 3 | 5 | 3 | 7 |

Filter | $\frac{1}{4}$ | $\frac{1}{2}$ | $\frac{1}{4}$ |

$\frac{1}{4} \times 3 + \frac{1}{2} \times 5 + \frac{1}{4} \times 3 = 4$

| $\frac{1}{4}$ | $\frac{1}{2}$ | $\frac{1}{4}$ |

$\frac{1}{4} \times 5 + \frac{1}{2} \times 3 + \frac{1}{4} \times 7 = \frac{9}{2}$

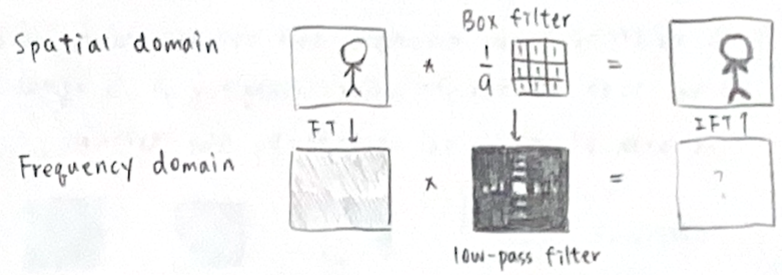
Result | | 3 | 4 | $\frac{9}{2}$ | |

→ Convolution Theorem:

Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice versa. Therefore,

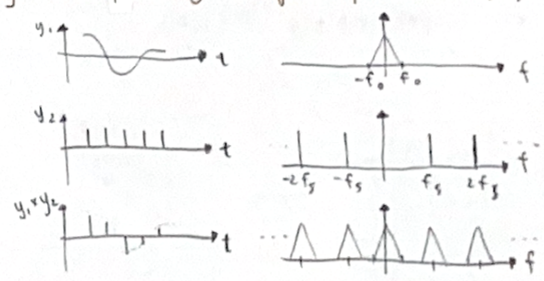
Option 1: Filter by convolution in the spatial domain.

Option 2: Transform to frequency domain (FT), multiply by Fourier transform of convolution kernel, and the transforms back to spatial domain (IFT).



!! More bigger the box filter is, more lower the frequency be filter.

* Sampling = Repeating frequency contents.



⇒ Aliasing = Mixed frequency contents

Dense 密集 sampling:

Sparse 稀疏 sampling:

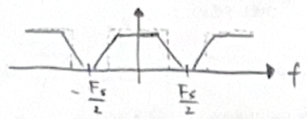
aliasing

How to reduce aliasing error?

1. Increase sampling rate
   - Essentially increasing the distance between replicas in the Fourier domain.
   - Higher resolution displays, sensors, framebuffers...
   - But: costly & may need very high resolution

2. Antialiasing
   - Making Fourier contents "narrower" before repeating
     → filter out high frequencies before sampling

Filter



Then sparse sampling

★ Antialising by averaging values in pixel area

1. convolve f(x,y) by a 1-pixel box-blur
2. sample at every pixel's center

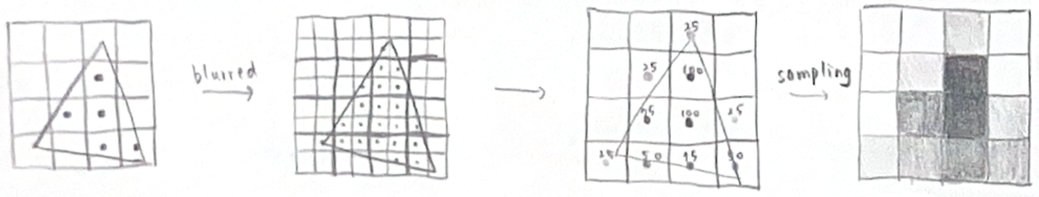
→ In rasterizing one triangle, the average value inside a pixel area of f(x,y) = inside(triangle, x, y) is equal to the area of the pixel covered by the triangle.

Original

Filter

★ Antialising by Supersampling (MSAA)

multisampling antialiasing



Notice!! This way doesn't increase the sampling rate and resolution.

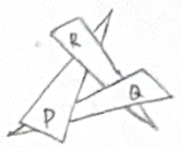
R3

How about visibility / occlusion (遮擋)?

Painter's algorithm: paint from back to front, overwrite in the framebuffer

→ requires sorting in depth with time-complexity O(n log n) for n triangles. Sort 每個三角形和所有的其他的距離

!! Can have unresolvable depth order:



★ Solve: Z-buffer:

for each pixel
- frame buffer: Store color values
- z buffer: store depth (depth buffer)

→ when rendering

z always positive
- small z → closer
- large z → further

|||| means update

Coding: (pseudo code)

```
Initialize depth buffer to ∞
During rasterization:
    for (each triangle T)
        for (each sample (x,y,z) in T)
            if (z < zbuffer[x,y])          // closest sample so far
                framebuffer[x,y] = rgb;    // update color
                zbuffer[x,y] = z;          // update depth
            else
                ⋮
```

Time complexity: O(n) for n triangles → because of no sorting → 假設各三角形面積差不多

Same result with different orders of drawing triangles.

when using MSAA, the depth is comparing between each sampling points, not pixel.

Transparent objects need special treatment !!

- Shading

Now we can do

Expected (no shadow)

def

In dictionary: The darkening or coloring of an illustration or diagram with parallel lines or a block of color.

In this course: The process of applying a material to an object.

* A simple Shading model (Blinn-Phong Reflectance Model)

- Specular highlights 鏡面高光
- Diffuse reflection 漫反射
- Ambient lighting 環境光

Shading at local:

Compute light reflected toward camera at a specific shading point.

Input: unit vector
(1) $\vec{n}$: surface normal
(2) $\vec{v}$: viewer direction
(3) $\vec{l}$: light direction (for each or many lights)
(4) surface parameters (color, shininess, ...)

(Lambertian)

* Diffuse reflection:

$$L_d = k_d \cdot \frac{I}{r^2} \cdot \max(0, \vec{n} \cdot \vec{l})$$

- L_d: diffusely reflected light
- $\frac{I}{r^2}$: energy received by the shading point
  - depend on θ between $\vec{n}$ and $\vec{l}$
  - if $\vec{n} \cdot \vec{l} < 0$, let $\vec{n} \cdot \vec{l} = 0$. (unphysical)
- $\max(0, \vec{n} \cdot \vec{l})$: energy arrived at the shading point (in 3D)
- k_d: diffuse coefficient (color) 顏色代表吸收多少量
  - $k_d = 1$ → 完全不吸收

NOTICE!! Diffuse reflection is independent of view direction $\vec{v}$

Specular term:

Shading point

$\vec{R}$ 和 $\vec{v}$ 越近 $\Leftrightarrow$ $\vec{n}$ 和 $\vec{h}$ 越近 (更好計算)

Shading point

$\vec{h}$ = 半角向量

$$\vec{h} = \text{bisector}(\vec{v}, \vec{l}) = \frac{\vec{v} + \vec{l}}{\|\vec{v} + \vec{l}\|}$$

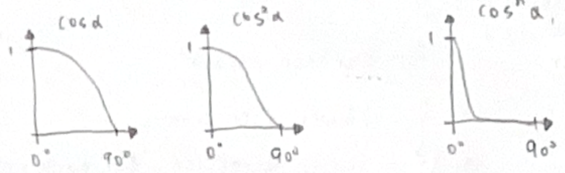
L_s = specular reflected light

$$L_s = k_s \cdot \frac{I}{r^2} \cdot \max(0, \cos\alpha)^p$$

$$= k_s \cdot \frac{I}{r^2} \cdot \max(0, \vec{n} \cdot \vec{h})^p$$

- k_s: specular coefficient
- $\frac{I}{r^2}$: energy arrived at the shading point (in 3P)

Increasing p narrows the reflection lobe



Ambient term: $L_a = k_a I_a$

Shading point

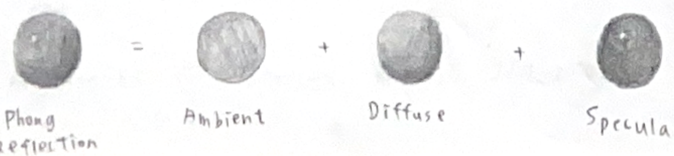
- k_a: ambient coefficient
- I_a: reflected ambient light

1. Add constant color to account for disregarded illumination and fill in black shadows
2. This is approximate (fake).

⇒ Blinn-Phong Reflection Model: $L = L_a + L_d + L_s$

$$= k_a I_a + k_d \frac{I}{r^2} \max(0, \vec{n} \cdot \vec{l}) + k_s \frac{I}{r^2} \max(0, \vec{n} \cdot \vec{h})^p$$

(p: 100↑ 越小)



Blinn-Phong Reflection | Ambient | Diffuse | Specular

* Shading Frequency

1. Flat shading:

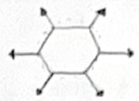
- Triangle face is flat — one normal vector
- Not good for smooth surfaces.

2. Gouraud shading:

- Interpolate colors from vertices across triangle

☆ Each vertex has a normal vector

How? Defining pre-vertex normal vector !!



Best! To get vertex normals from the underlying geometry. ex. consider a sphere

Otherwise, have to infer vertex normals from triangle faces. Simple scheme: average surrounding face normals: $\vec{N}_v = \frac{\sum_i \vec{N}_i}{\|\sum_i \vec{N}_i\|}$

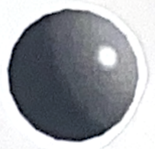
3. Phong shading:

(後面重心座標會解釋)

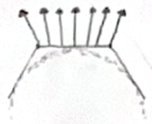
- Interpolate normal vectors across each triangle

☆ Compute full shading model at each pixel

- NOT the Blinn-Phong reflectance model



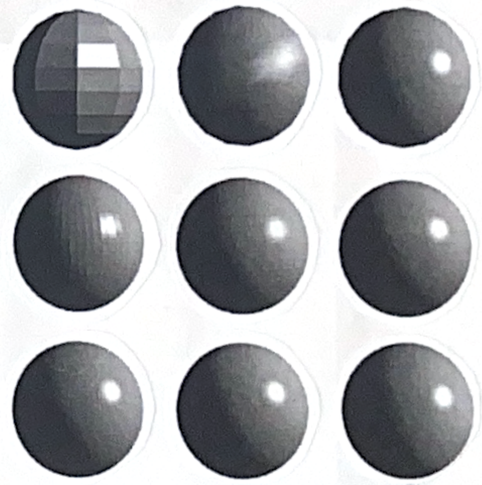
How?: Defining pre-pixel normal vectors



Barycentric interpolation of vertex normals.

should normalize

⇒ To sum up:

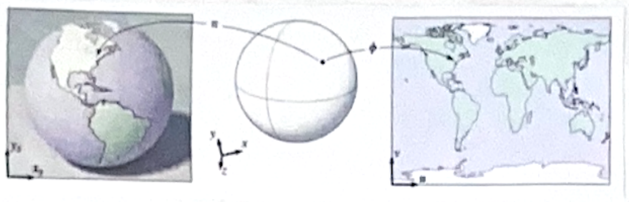


★ Texture mapping:

Different color at different place: k_d is different due to different color.

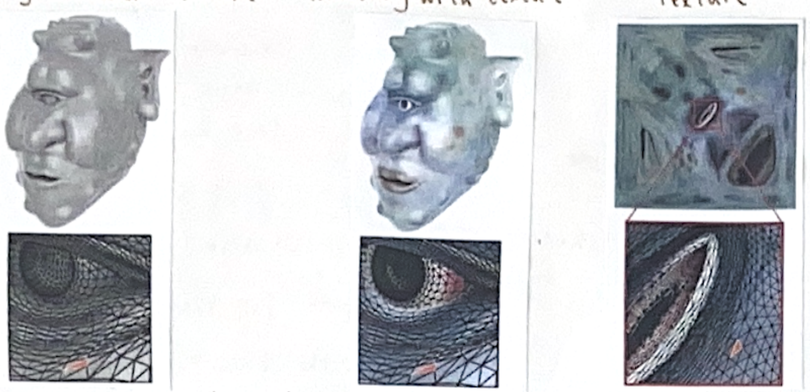
Surfaces are 2D: Surface lives in 3D world space, but every 3D surface point also has a place where it goes in the 2D image.

⇒ texture.



Texture applied to surface

Rendering without texture | Rendering with texture | Texture

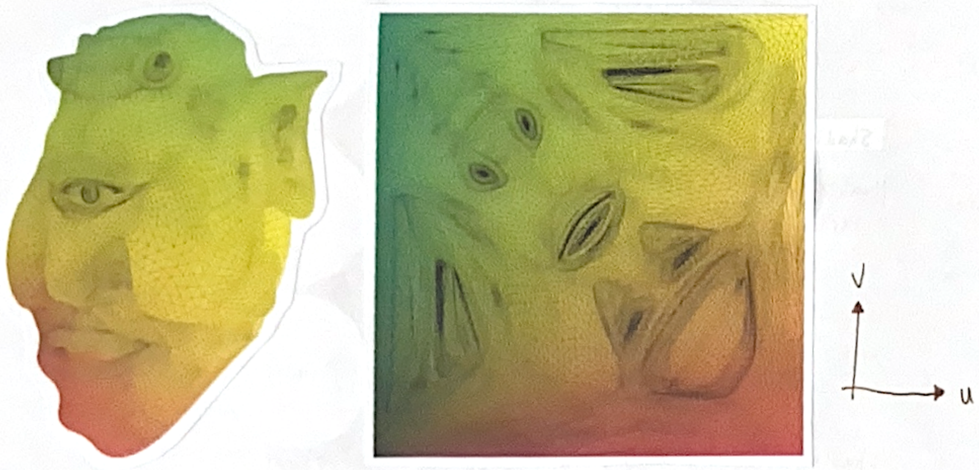


Zoom

Each triangle "copies" a piece of the texture image to the surface

Visualization of texture coordinates: each triangle vertex is assigned a texture coordinate (u, v)

Texture space



!! Textures can be used multiple times!! (地板, 木紋, 磁磚, 只要設計的好, 可以銜接)

tiled (鋪texture)

Interpolation Across Triangles

Why do we want to interpolate?

1. Specify values at vertices
2. Obtain smoothly varying values across triangles

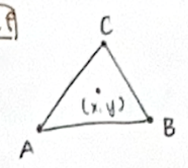
What do we want to interpolate?

Texture coordinates, colors, normal vectors, ...

How do we interpolate?

★ Barycentric coordinates: (α, β, γ)

def

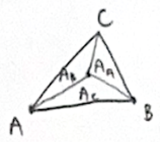


$(x, y) = \alpha A + \beta B + \gamma C$ and $\alpha + \beta + \gamma = 1$

Inside the triangle if all three coordinates are non-negative.

$(\alpha, \beta, \gamma) = (1, 0, 0)$

Geometric viewpoint — proportional area



$$\alpha = \frac{A_A}{A_A + A_B + A_C}$$

$$\beta = \frac{A_B}{A_A + A_B + A_C}$$

$$\gamma = \frac{A_C}{A_A + A_B + A_C}$$

$\Rightarrow (\alpha, \beta, \gamma) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ is centroid of triangle

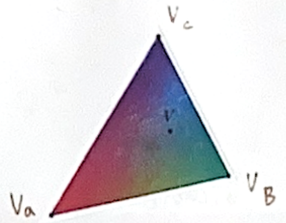
The general formula:

$$\alpha = \frac{-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)}{-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)}$$

$$\beta = \frac{-(x - x_C)(y_A - y_C) + (y - y_C)(x_A - x_C)}{-(x_B - x_C)(y_A - y_C) + (y_B - y_C)(x_A - x_C)}$$

$$\gamma = 1 - \alpha - \beta$$

Linear interpolate values at vertices:



$V = \alpha V_A + \beta V_B + \gamma V_C$

V_A, V_B, V_C can be positions, texture coordinates, color, normal, depth, material attributes ....

★★ However, barycentric coordinates are not invariant under projection!

⇒ Applying Textures :

For each rasterized screen sample (x,y), we get (u,v) which
↳ usually a pixel's center

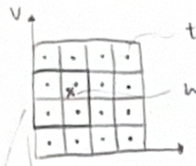
evaluate texture coordinate at (x,y).
↳ use barycentric coordinates

⇒ texcolor = texture.sample(u,v)
↓
set sample's color → the diffuse coefficient k_d in Blinn-Phong Reflectance Model.

!!! Limitation : Texture Magnification

1. What if the texture is too small? 한 픽셀 당 1개 pixel 보다 많은 texture 좌표
insufficient texture resolution

Solve: Bilinear interpolation



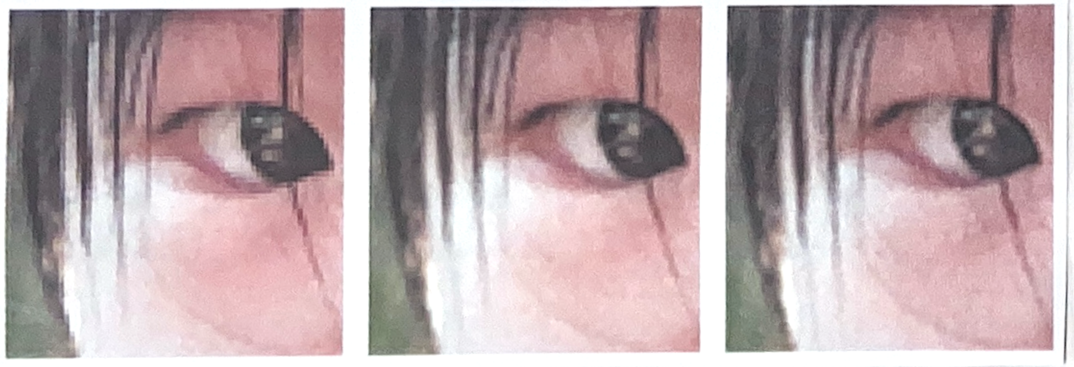
Take 4 nearest sample locations, with texture values as labeled.

Linear interpolation (1D) : $lerp(x, v_0, v_1) = v_0 + x(v_1 - v_0)$

⇒ $\begin{cases} u_0 = lerp(s, u_{00}, u_{10}) \\ u_1 = lerp(s, u_{01}, u_{11}) \end{cases}$

⇒ $f(x,y) = lerp(t, u_0, u_1)$

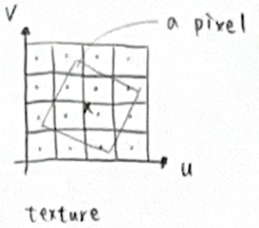
Bilinear interpolation usually gives pretty good results at reasonable costs.



Nearest　　Bilinear　　Bicubic

Sq

2. What if the texture is too large? 한 픽셀 - 1개 pixel 에 많은 texture 좌표



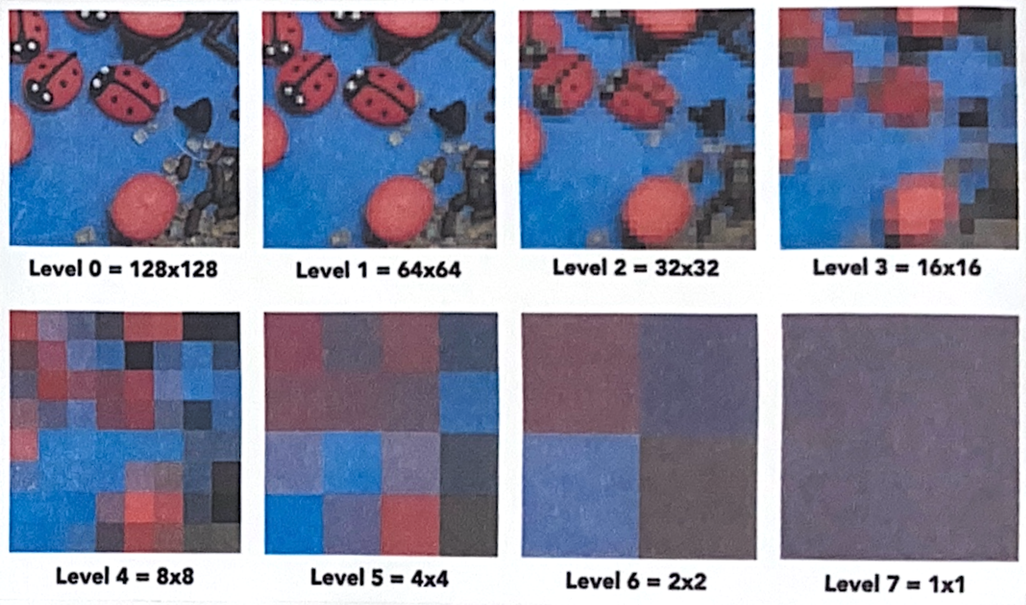
Will supersampling (pixel's) work of antialiasing?

- Yes, high quality, but costly.
- When highly minified, many texels in pixel footprint.
- Signal frequency too large in a pixel
- Need even higher sampling frequency.

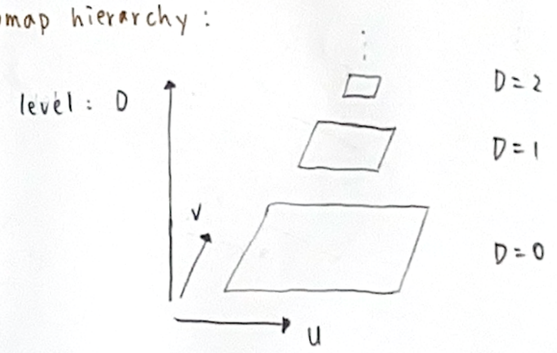
⇒ Solve : get the average value with a range!!

(From point query → average range query)

Using "Mipmap" : Allowing (fast, approximation, square) range queries
↳ means a multitude in a small space



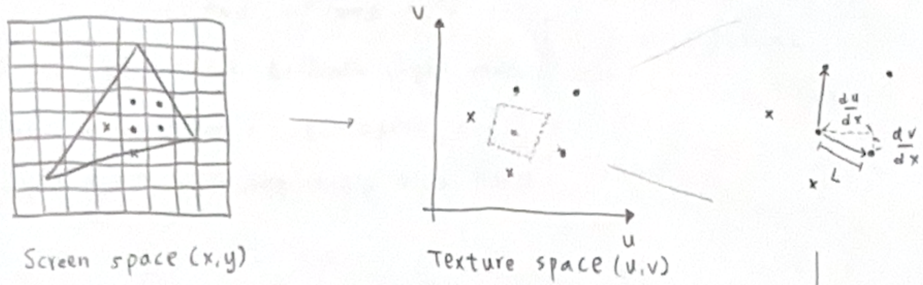
Mipmap hierarchy :



The storage overhead of mipmap is $\frac{4}{3}$ times of the original one.

Which level we use at pixel (x,y)?

Computing mipmap level D:

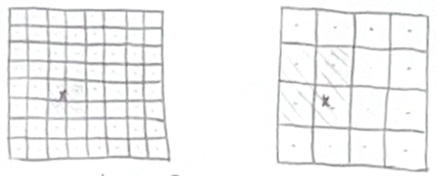


$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

$D = \log_2 L$ → 求邊長 L 在 D 等於多少的時候.

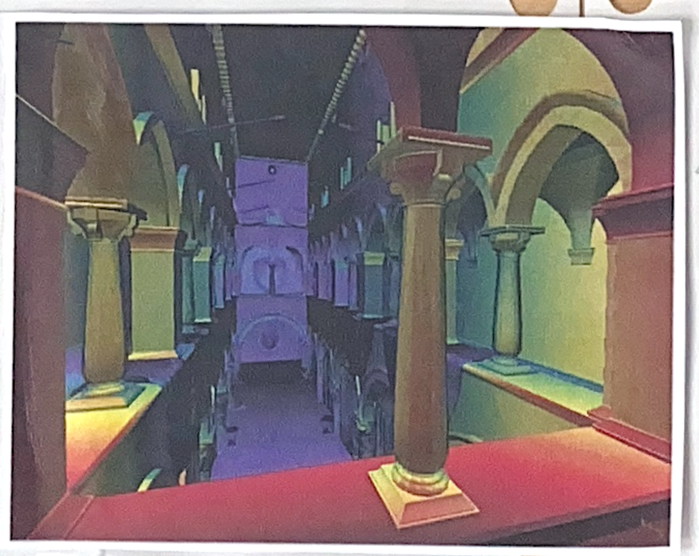
However, D need to round to nearest integer level.

Therefore, trilinear interpolation:



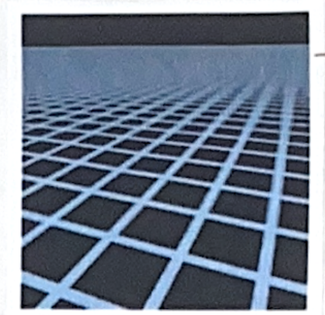
Bilinear result — Bilinear result

Linear interpolation based on continuous D value



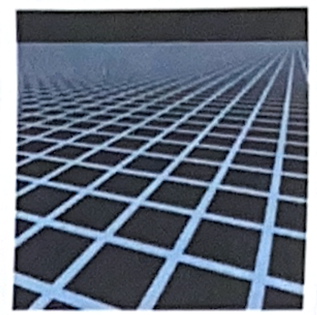
55

!!! Mipmap Limitation:



Overblur

Modify

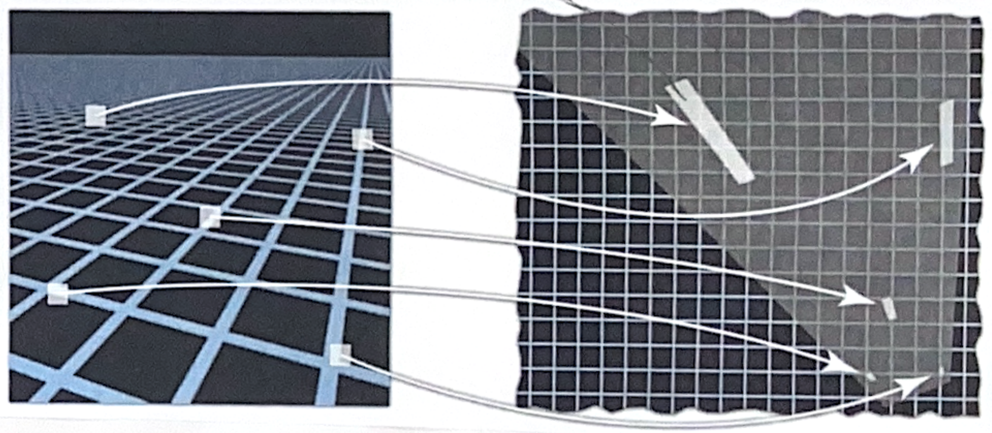


Anisotropic Filtering

Anisotropic Filterng:

Ripmaps and summed area tables:

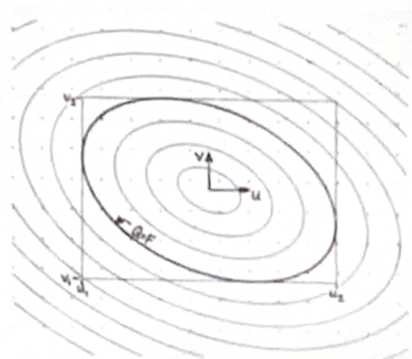
1. Can look up axis-aligned rectangular zones.
2. Diagonal footprints still a problem.



Solve

EWA filtering:

1. Use multiple lookups
2. Weighted average
3. Mipmap hierarchy still helps
4. Can handle irregular footprints



Applications of textures:

In modern GPUs, texture = memory + range query (filtering). (ex. mipmap)

⇒ means a general method to bring data to fragment calculations.
可以圖像數據化 ~~

ex. Environment lighting, Store microgeometry, Procedural textures, Solid modeling, Volume rendering...

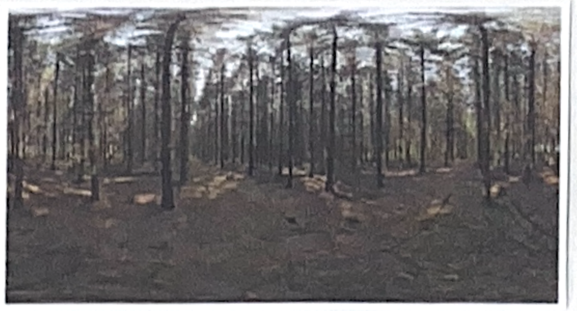
Light from the environment

Rendering with the environment

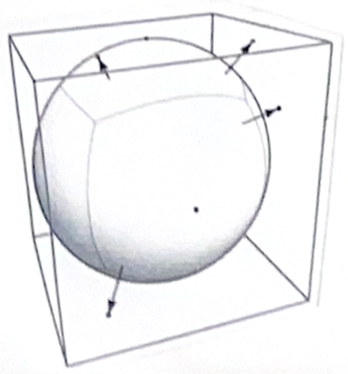
⇒ we can save the environment map on spherical. (assume the light came from infinity)

Spherical map:

Prone to distortion → (top and bottom parts)



⇒ Cube map: A vector maps to cube point along that direction. The cube is textured with 6 square texture maps.



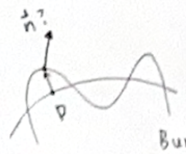
→ Much less distortion

Also, textures can affect shading!

height → normal vector → shading

FAKE the detail geometry

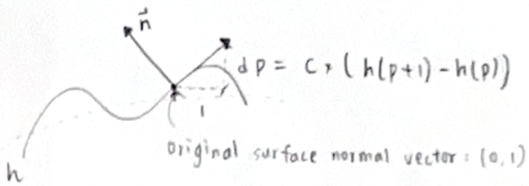
Bump mapping: adding surface detail without adding more triangles



Bump texture

1. Perturb surface normal per pixel
2. "Height shift" per texel defined by a texture
3. How to modify normal vector?

2D (flatland):



⇒ $\vec{n}(p) = (-dp, 1).\text{normalized}()$

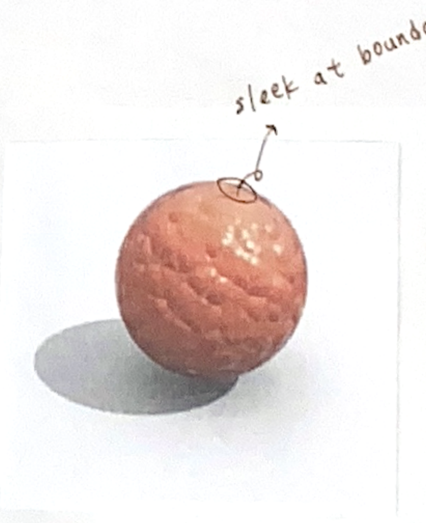
3D:

Original surface normal vector: (0,0,1)

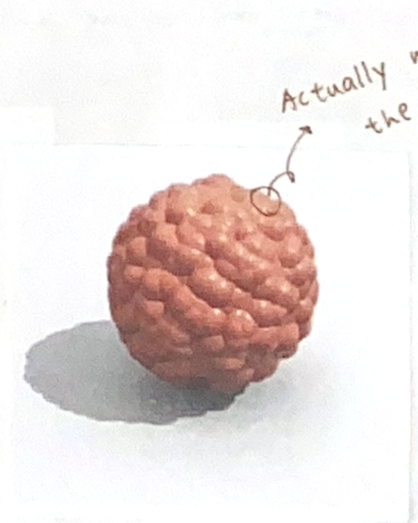
$$\begin{cases} \frac{dp}{du} = c_1 \times (h(u+1) - h(u)) \\ \frac{dp}{dv} = c_2 \times (h(v+1) - h(v)) \end{cases}$$

⇒ $\vec{n}(p) = \left(-\frac{dp}{du}, -\frac{dp}{dv}, 1\right).\text{normalized}()$

!! a more advanced approach: displacement mapping!
(Use the same texture as in bumping mapping)



Bump / Normal mapping



Displacement mapping → 比較貴:

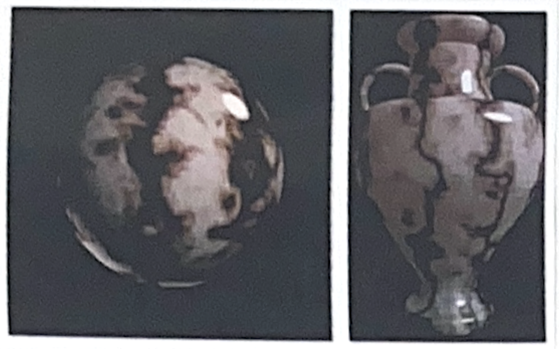
△ 的 sample freq.

V

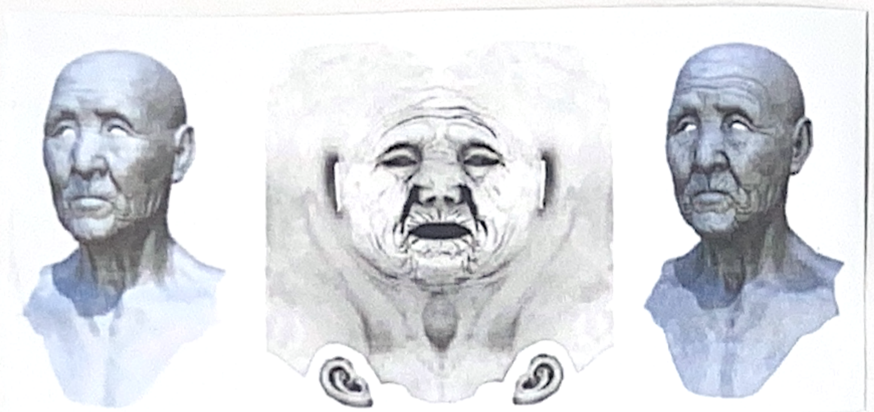
頂點 的 sample freq.

Complementary :

3D procedural noise + solid modeling:



Provide precomputed shading :

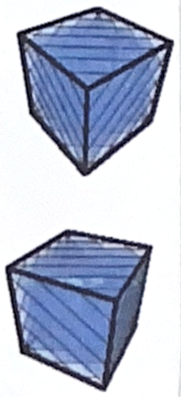


Simple shading

Ambient occlusion texture map

with ambient occlusion

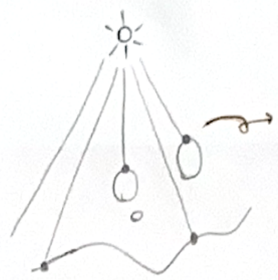
3D textures and volume rendering :



*** Shadow mapping

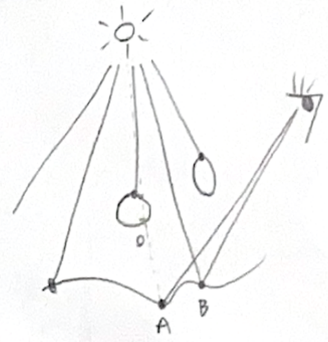
Key idea: the point NOT in shadow must be seen BOTH by the light and by the camera.

Step 1: render from light



→ Get the depth image from light source.

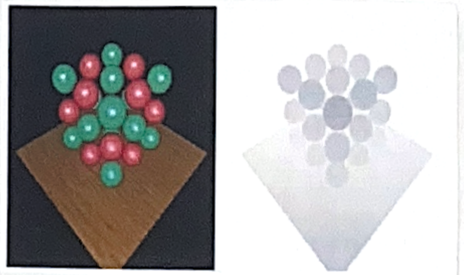
Step 2: project to light



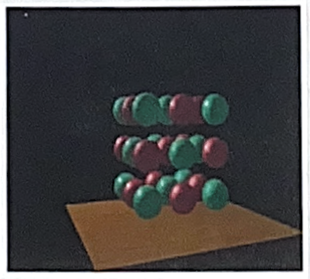
Project visible points in eye view back to light source.

- A: depths from light and eye are not the same. → BLOCKED!!
- B: depths match for light and eye. → VISIBLE !!

Visualizing shadow mapping:



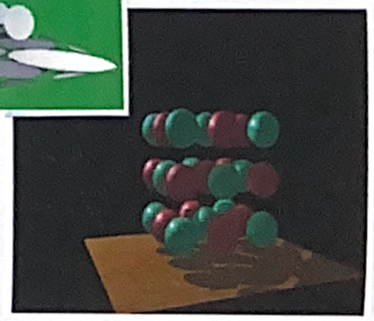
The scene AND depth buffer from the light's point-of-view.



From the eye's point-of-view. (Without shadow)

compare distance (light, shading point) with shadow map.

- Green is where the distance (light, shading point) ≈ depth on the shadow point
- Non-green is where shadows should be.



Scene with shadows!

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)

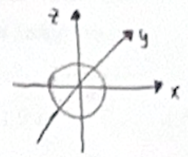
G1

- 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 :

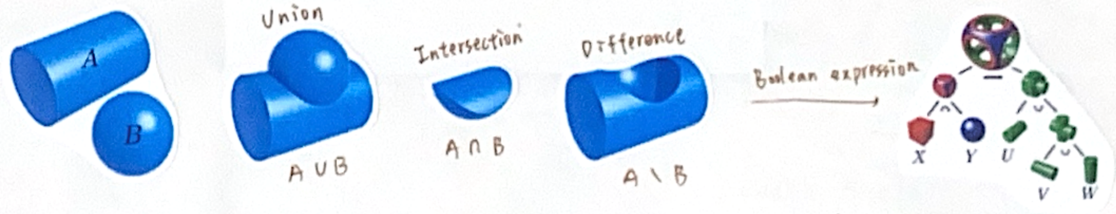
   ex. $x^2+y^2+z^2=1$ → sphere

   $(R-\sqrt{x^2+y^2})^2+z^2=r^2$ → 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