

A Lecture for Game Mathematics, Real-time Computer Graphics & Shader Programming

Game Graphics

王銓彭

墨匠科技 BlackSmith CEO

cwang001@mac.com

cwang001@blacksmis.com

Contents

- 遊戲圖學的簡介
- 三角函數 (Trigonometric Functions) , 向量 (Vectors) , 矩陣 (Matrix)
- 仿射轉換 (Affine Transformations)
- 碰撞偵測 (Collision Detection – Intersections)
- 基本幾何的表示法 (Basic Geometry Representation)
- 攝影機系統 (Camera Viewing & Projection)
- 隱藏面消除法 (Hidden-surface Removal – Z-buffer)
- 基本的材質與打光 (Materials & Lighting Using Phong Reflection)
- 貼圖技術 (Texture Mapping)
- 即時3D渲染流程 (Real-time 3D Rendering Pipeline)

Contents

- 法向量貼圖 (Normal Maps)
- 多次渲染的概念 (Multi-pass Rendering)
- 影子的做法 (Shadow Maps)
- 高動態對比影像 (High Dynamic Range Imaging, HDRi)
- 反射與折射 (Reflection and Refraction)
- 非幾何打光法 (Real-time Image-based Lighting)
- 環境光遮蔽 (Ambient Occlusion, AO)
- 次表面散射 (Sub-surface Scattering – Skin Rendering)
- 非擬真渲染 (Non-photorealistic Rendering)

Introduction

Computer Graphics 的起源：1960-1970

- Wireframe graphics
- 1963 Dr. Ivan Sutherland 的博士論文
 - “Project Sketchpad”
 - Display processors
 - 現代顯示卡的濫觴
 - 3D clipping algorithm

伊凡·愛德華·蘇澤蘭（英語：Ivan Edward Sutherland，1938年5月16日—），生於美國內布拉斯加州黑斯廷斯，計算機科學家，被認為是「計算機圖形學之父」^[1]。因發明Sketchpad，拓展了計算機圖形學的領域，為1988年圖靈獎得主。

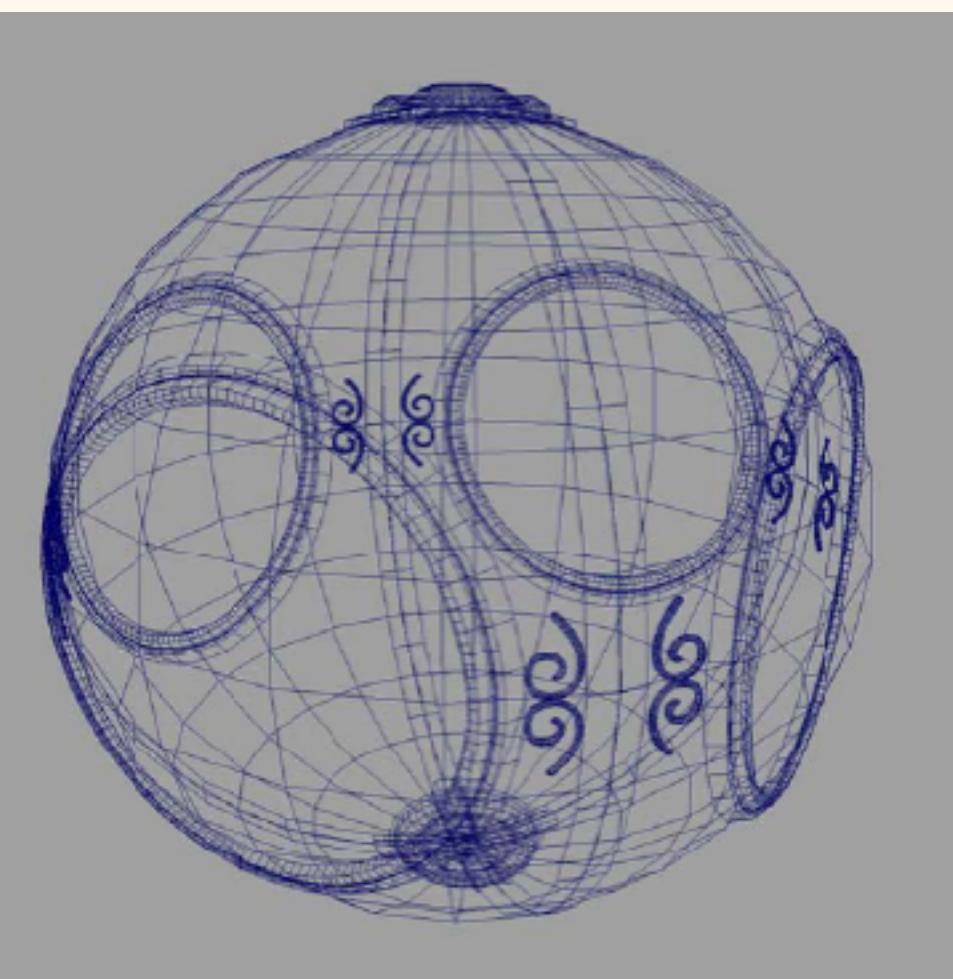
生平 [編輯]

在卡內基技術學院（今卡內基梅隆大學）取得學士學位，於加州理工學院取得碩士。1963年，在麻省理工學院取得計算機工程博士學位。他在博士論文中提出的Sketchpad程式，被認為是計算機圖形學的一大突破。

1962年，他著手進行自己的博士論文，在麻省理工學院開始創作Sketchpad程式。克勞德·夏農同意擔任他的指導教授，馬文·閔斯基等人也參與指導。

參考資料 [編輯]

1. ^ Ivan E. Sutherland Display Windowing by Clipping Patent No. 3,639,736. NIH. [13 February 2016]. (原始內容存檔於19 February 2016).
「Sutherland is widely regarded as the "father of computer graphics."」



伊凡·愛德華·蘇澤蘭
Ivan Edward Sutherland

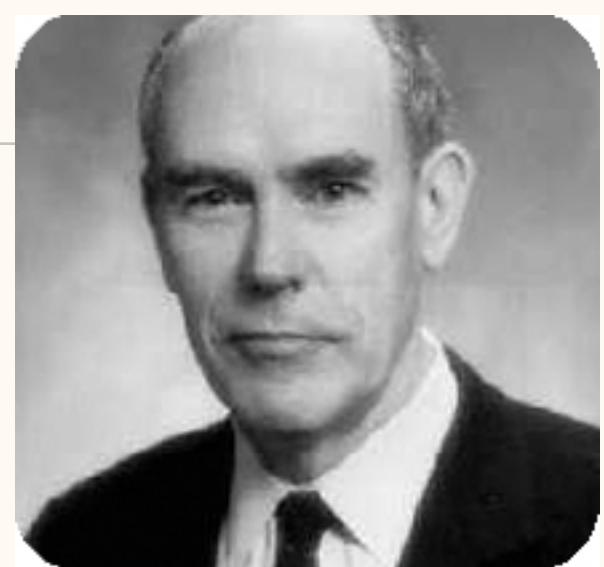


出生 1938年5月16日 (83歲)
美國，內布拉斯加州，黑斯廷斯
母校 麻省理工學院 (Ph.D., 1963)
加州理工學院 (M.S., 1960)
卡內基美隆大學 (B.S., 1959)
知名於 Sketchpad，被認為是電腦圖學的開創者
科恩－蘇澤蘭算法
獎項 圖靈獎 (1988)
電腦先鋒獎 (1985)
IEEE約翰·馮·諾伊曼獎章 (1998)
電腦協會會士，
美國國家工程院 院士，
美國國家科學院 院士，
京都獎
計算機歷史博物館 院士 (2005)

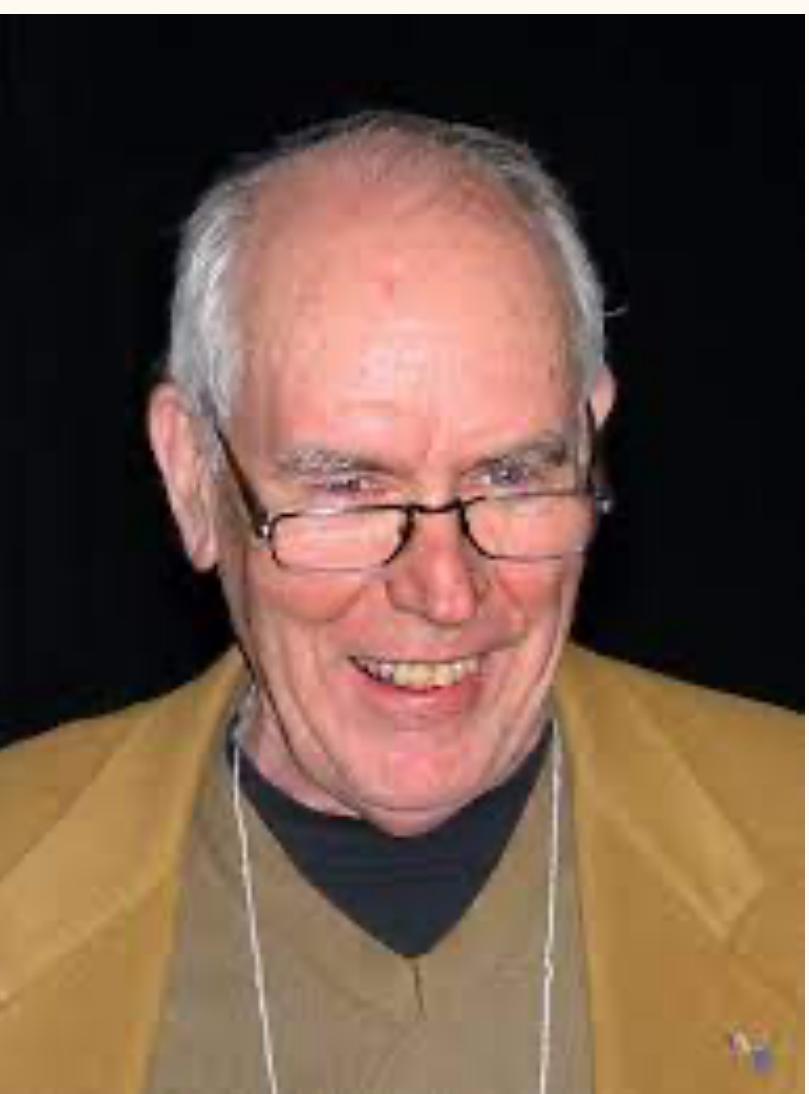
科學生涯

研究領域 計算機科學
網路
電腦圖學
機構 哈佛大學
猶他大學
Evans and Sutherland
加利福尼亞理工學院
卡內基梅隆大學
昇陽電腦
波特蘭州立大學
國防高等研究計劃署 (1964 – 1966)
論文 Sketchpad, a Man–Machine Graphical Communication System (1963)

Project Sketchpad

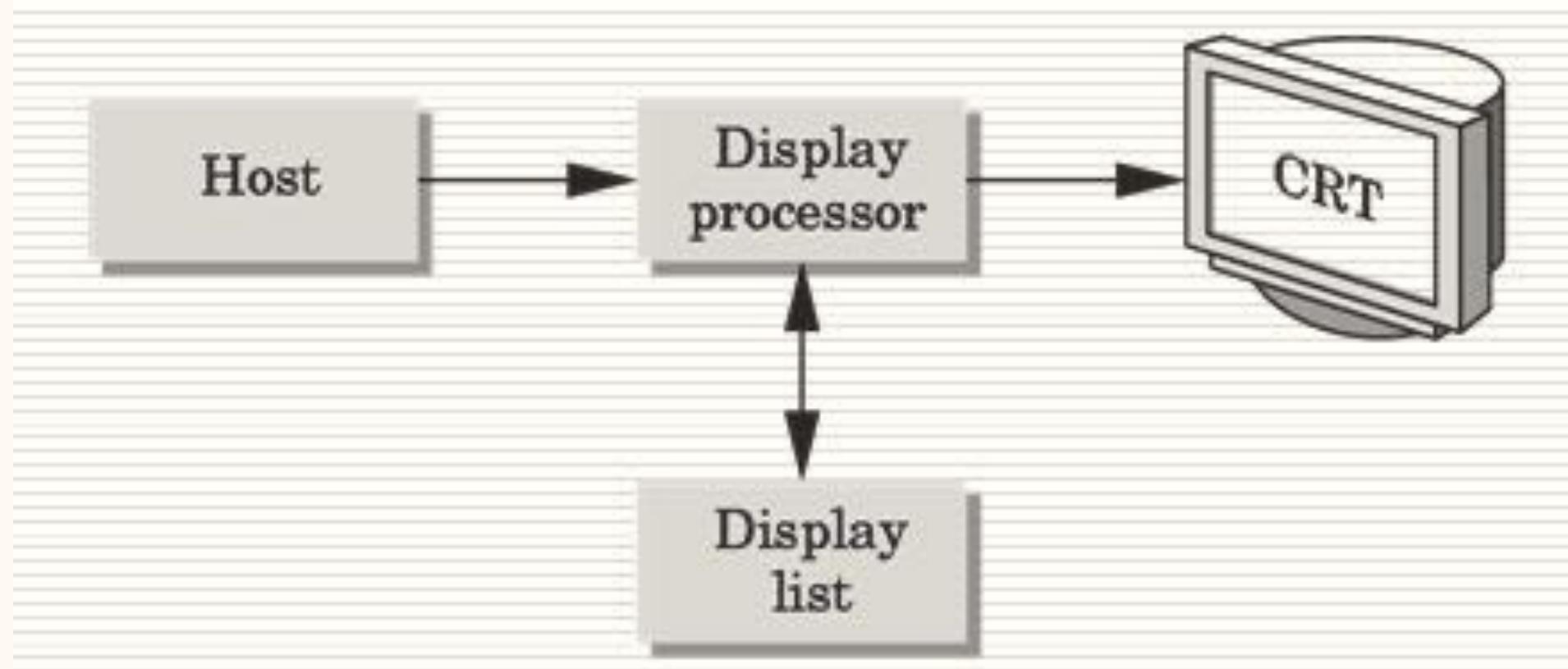


- Project Sketchpad
 - Dr. Ivan Sutherland 在麻省理工的博士論文
 - 被尊稱為電腦圖學之父“Father of Computer Graphics”
 - 最早研究VR/AR的先驅
 - 人機互動的開始
 - 光筆(Light pen)
- 渲染迴圈
- 現代渲染演算法的起源
 - 3D clipping是其中最知名之一



Project Sketchpad

- Display processor
 - 現代渲染硬體的雛型
 - Display Processor Unit (DPU)



- 圖形資料儲存在“Display list”中
- 電腦主機編譯 display list 的內容，將其送至 DPU 渲染
- 繪圖程式引擎的起源

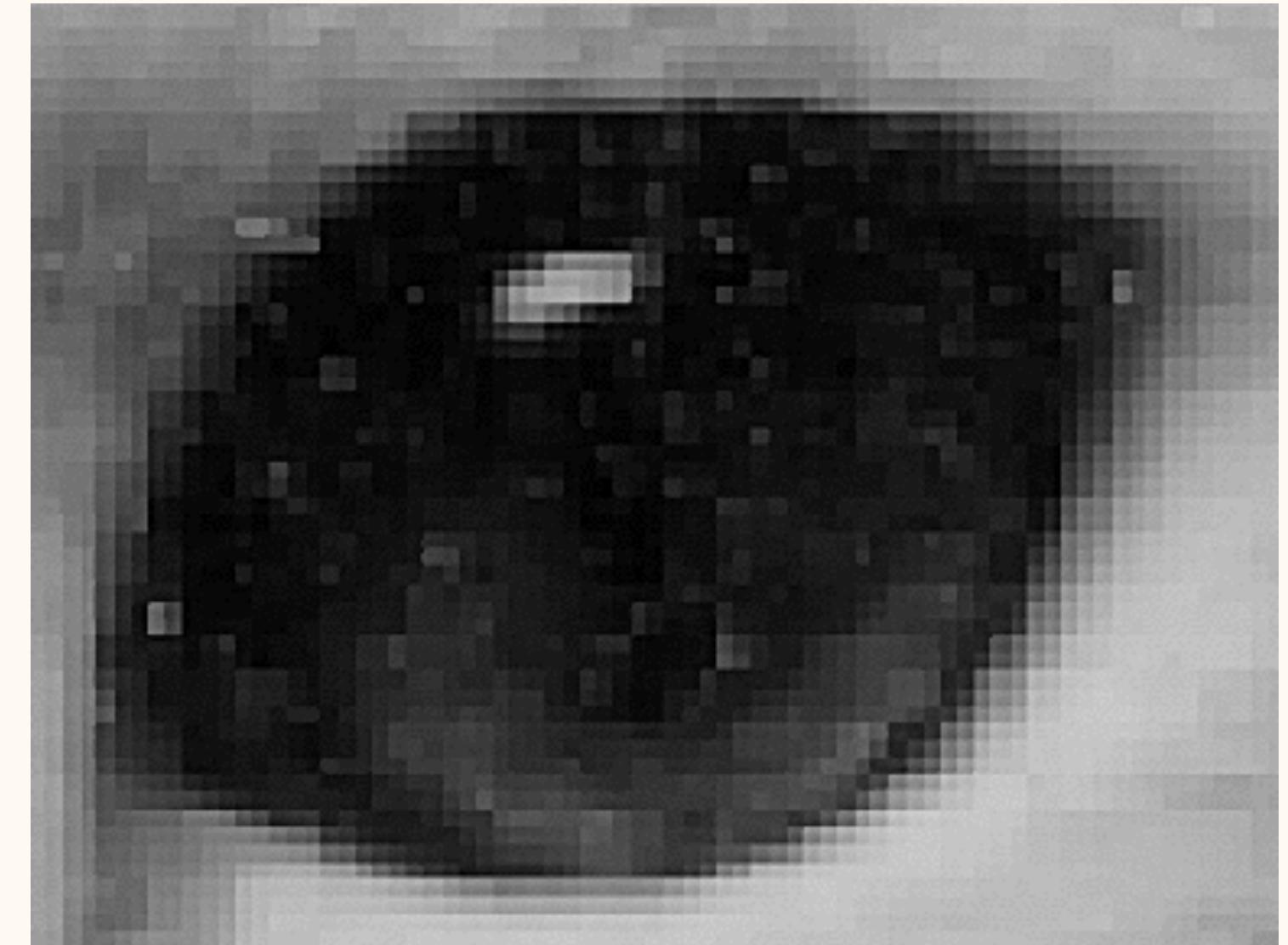
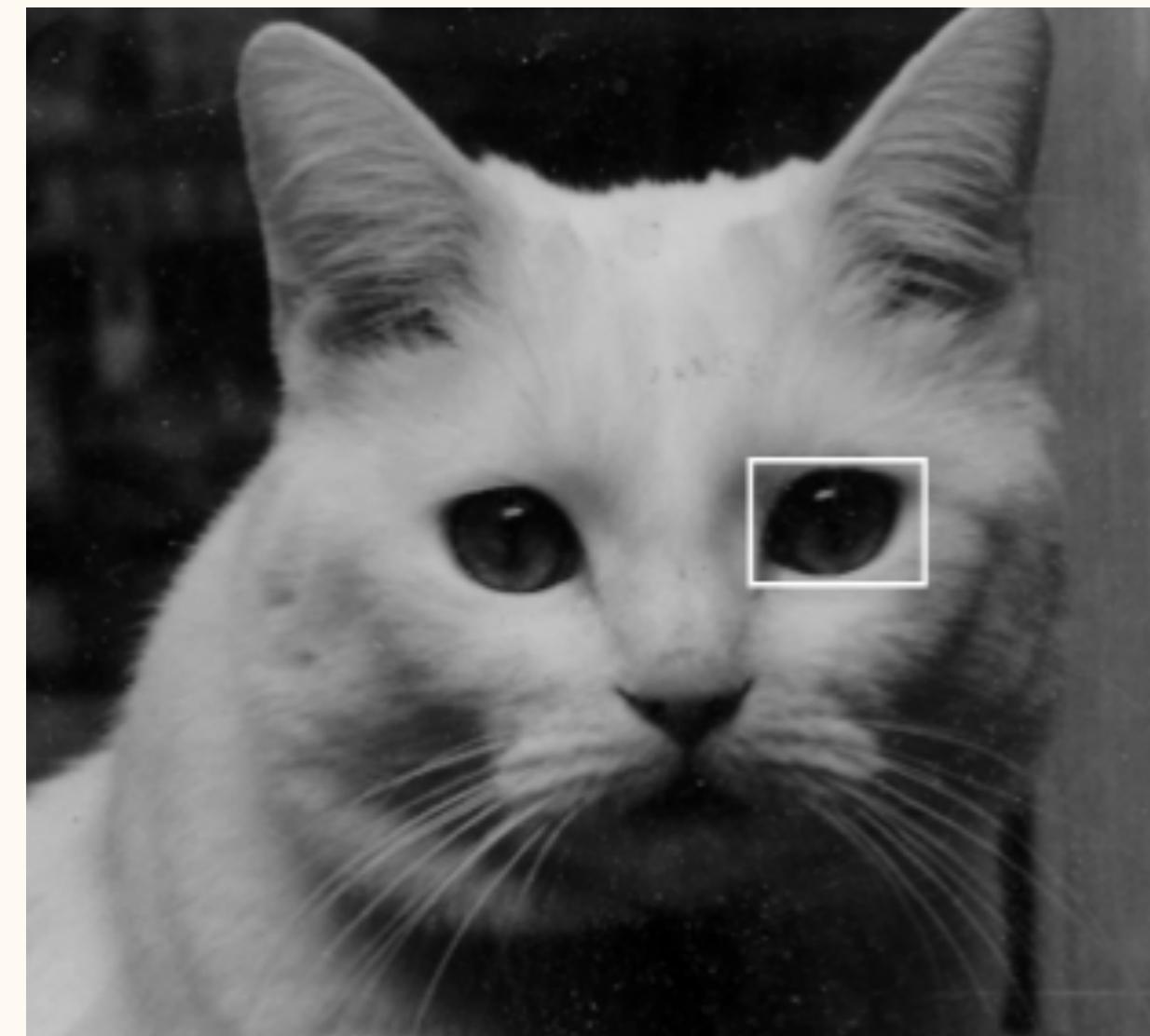
Rendering loop

```
while (true) {
    Read input from light pen
    Process 3D data
    Rendering
    Sleep a while
}
```

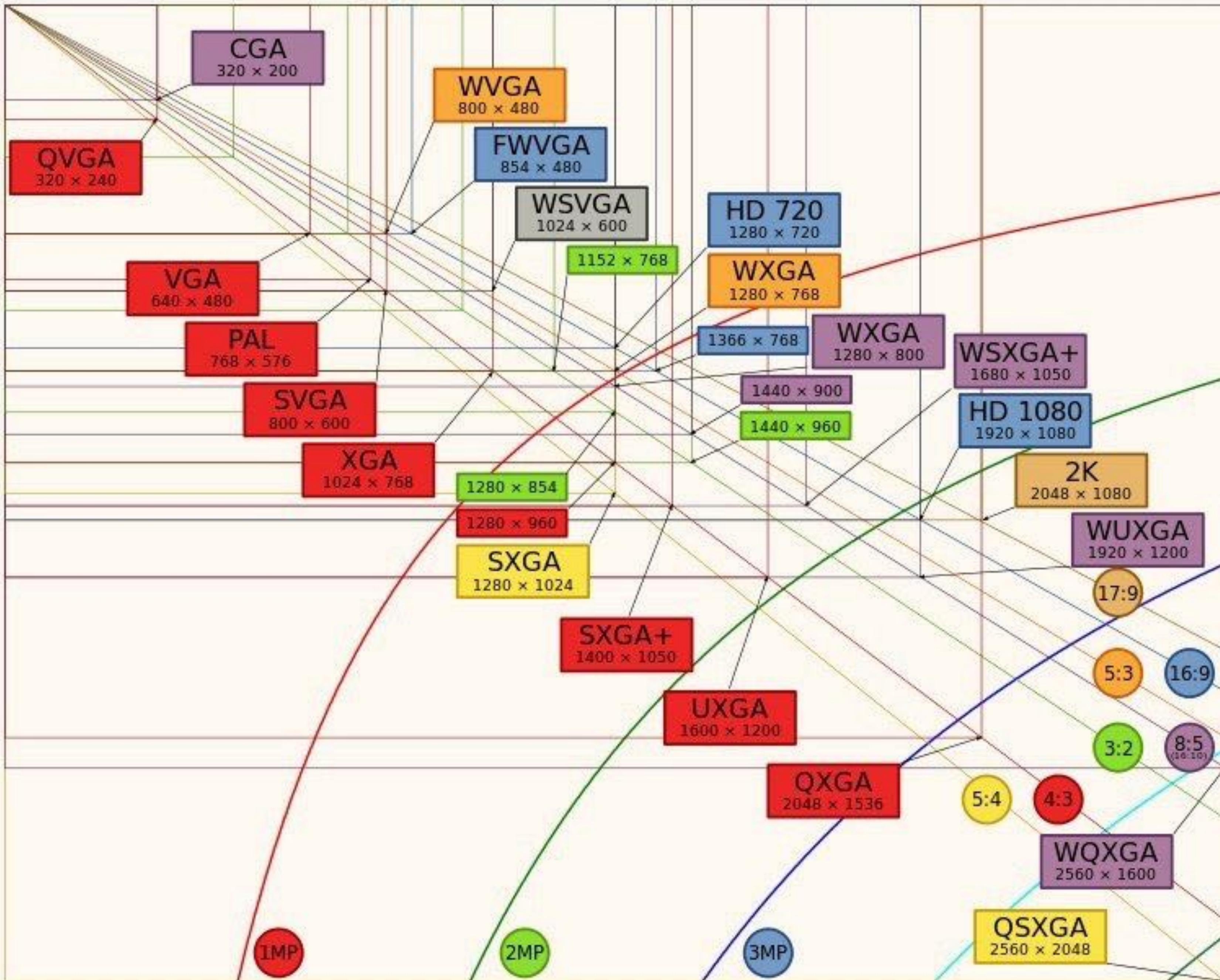
1970-1980 進入著色式的顯示方式

- 著色式的圖形 (Raster Graphics)
 - 影像以陣列方式儲存在顯示記憶體 (Video Memory) 中
 - Frame buffer
 - 像素 (Pixel)
 - Picture element 縮寫

```
00011010011...
01101100010...
11011110110...
...
```

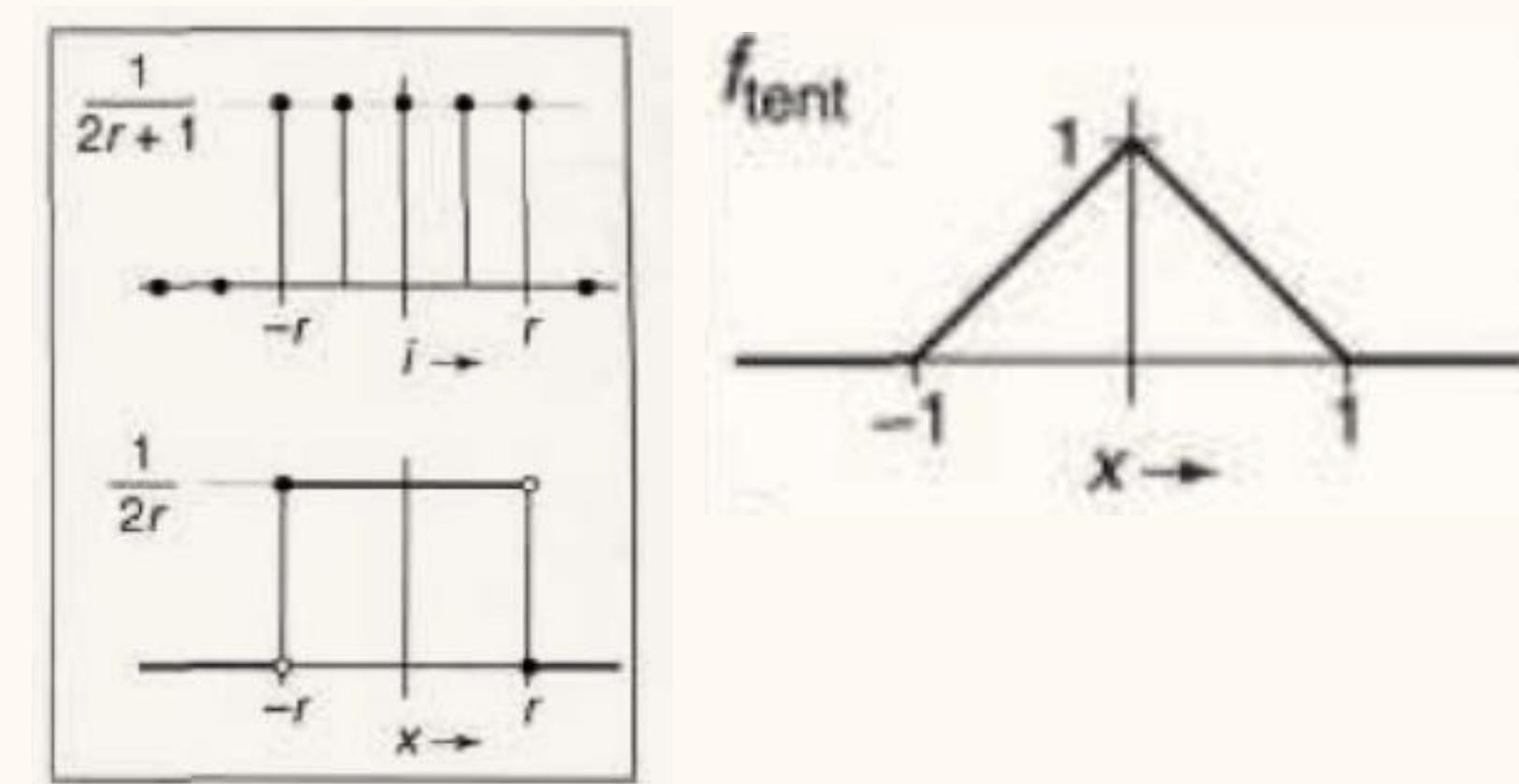


解析度



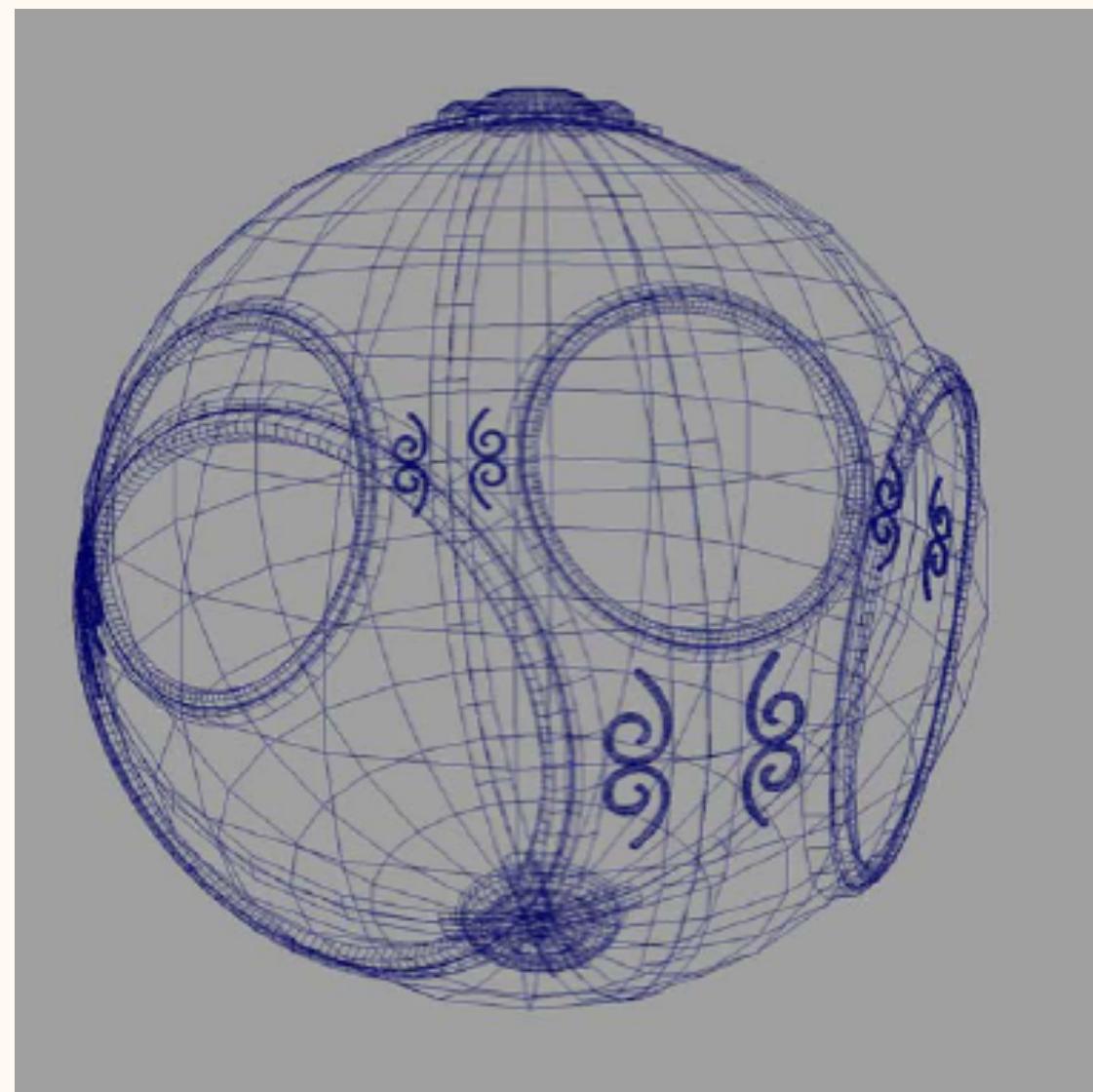
Aliasing & Anti-aliasing

- 著色式顯示的圖像主要在視覺上的特徵：
 - 鋸齒現象 (Aliasing problem)
 - Use a filter function to average color of the regions of drawing
 - 反鋸齒消除 (Antialiasing)
 - “box filter”, the “tent filter”, “Gaussian filter”, ...

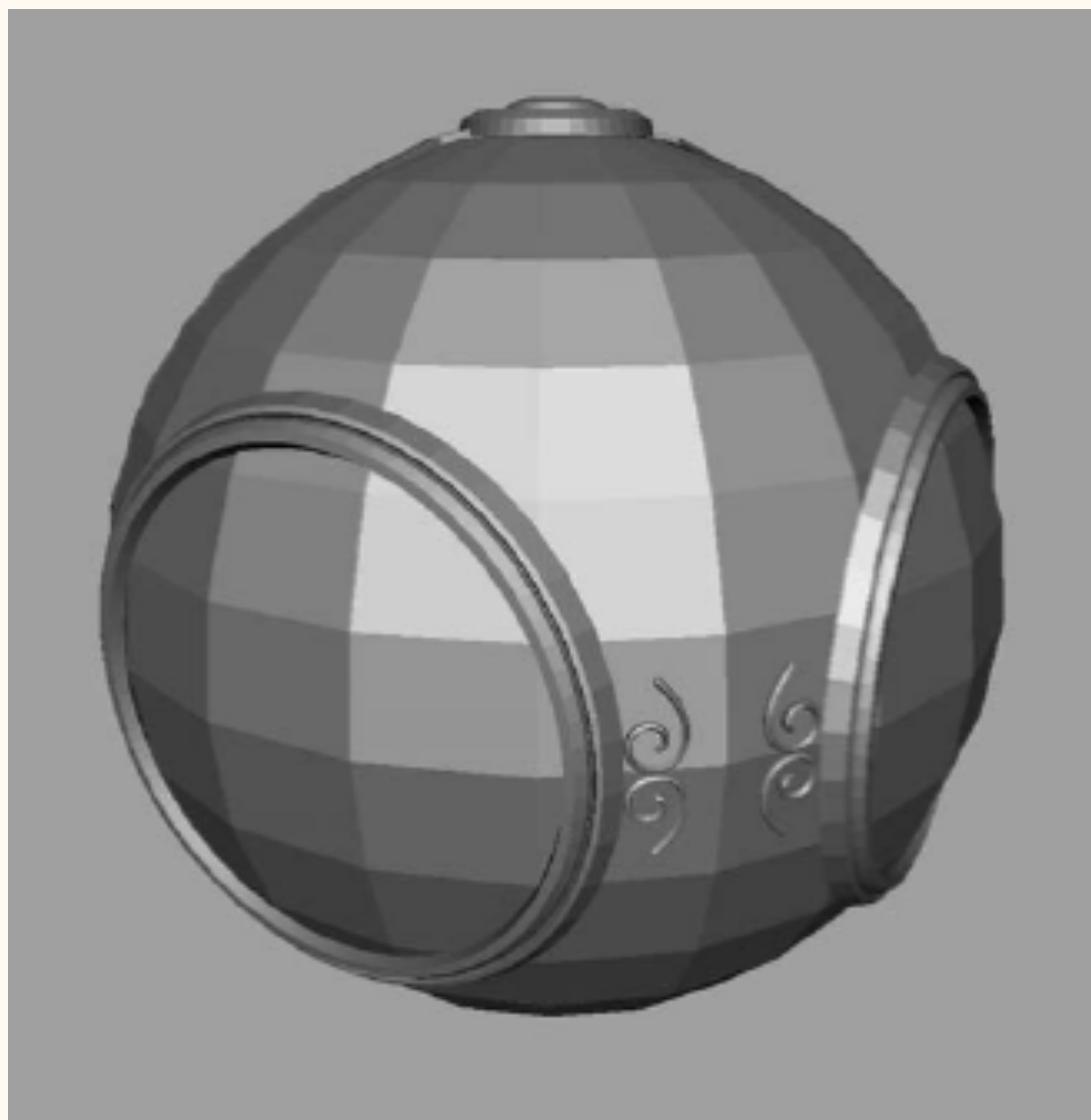


1970-1980 進入著色式的顯示方式

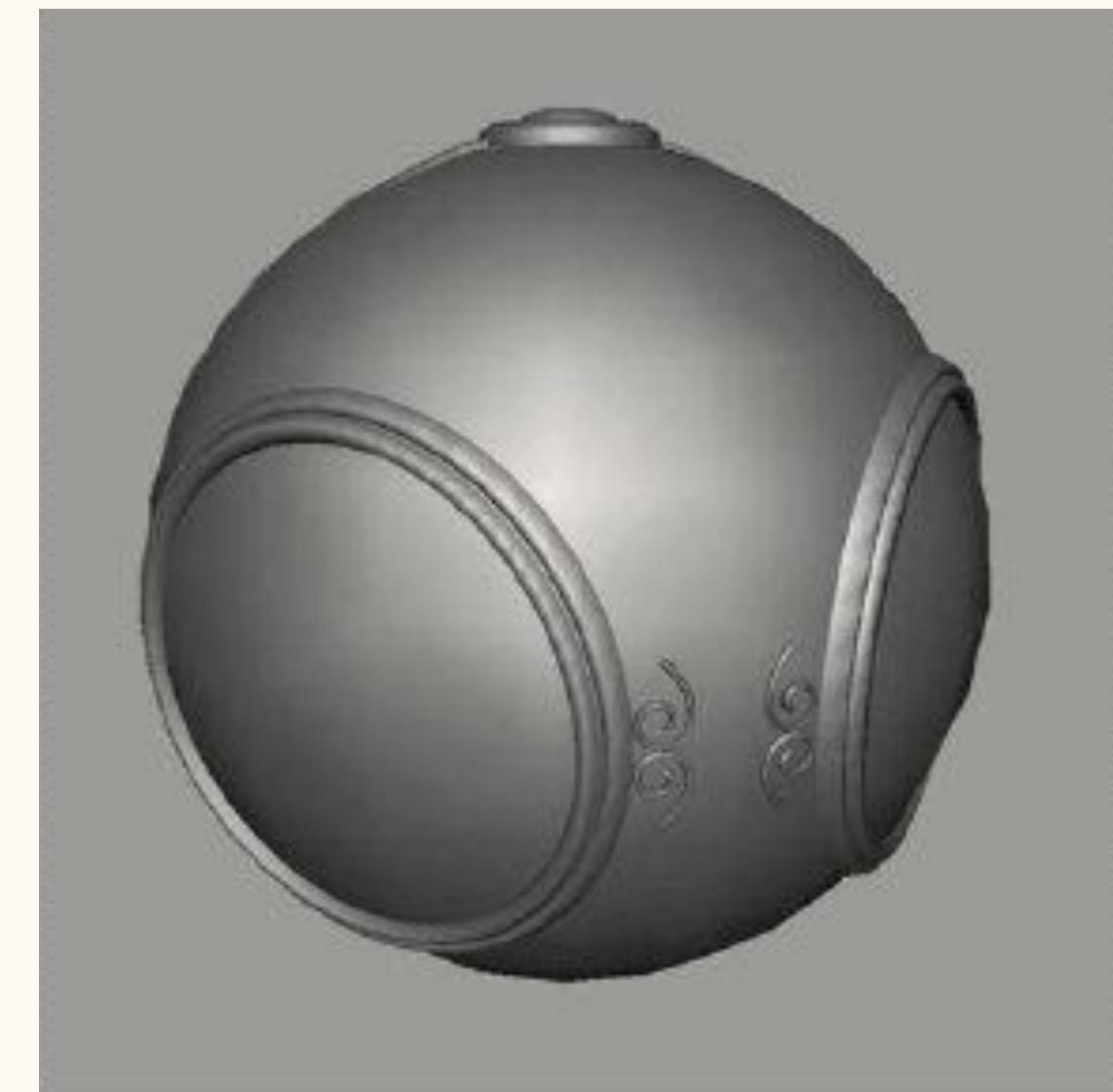
- 開始從畫線框(Wireframe)時代進入打光填色的時代
 - Shading & Lighting



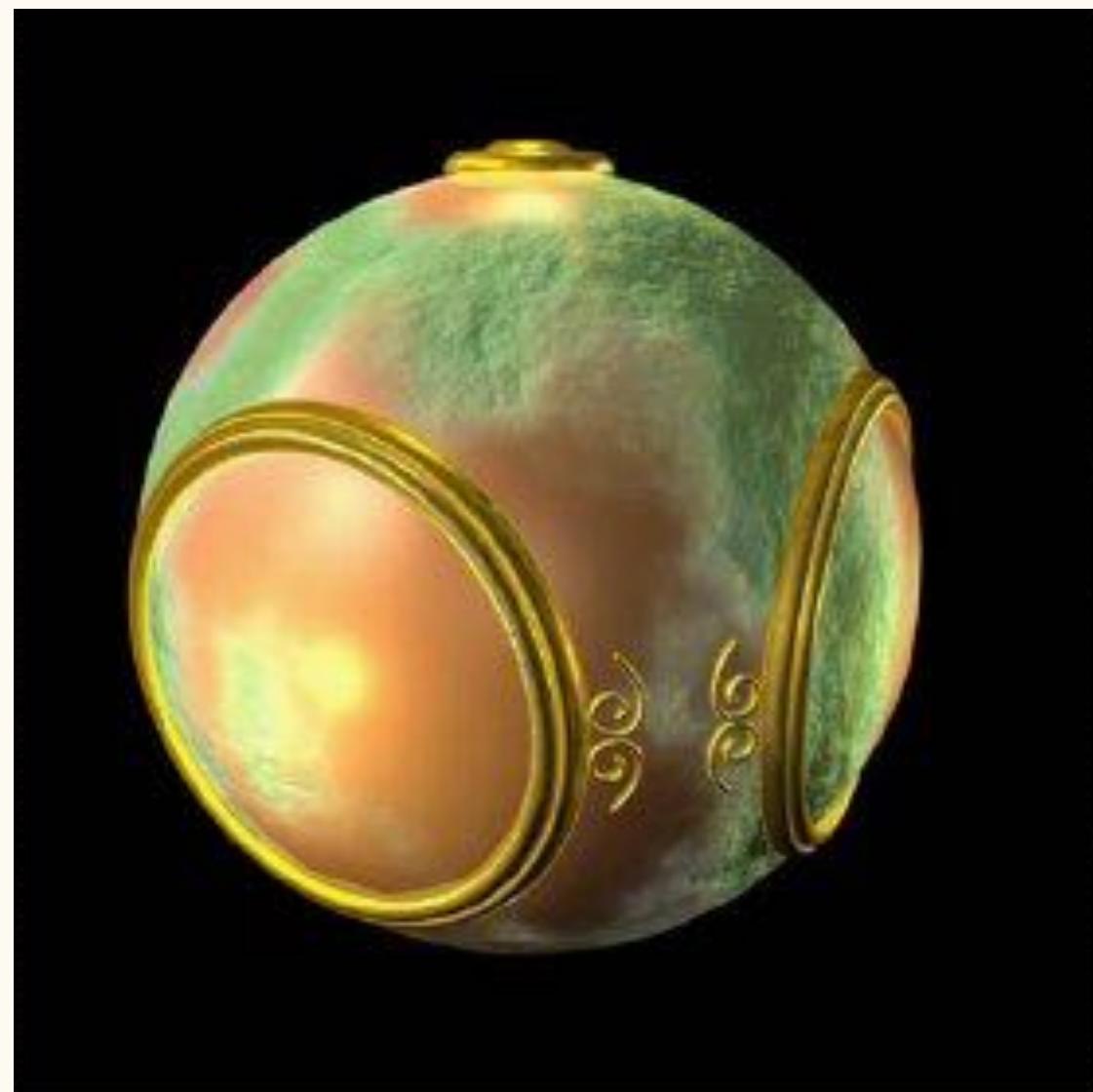
Wireframe Graphics



Flat Shading



Smooth Shading
(Gouraud, 1971)



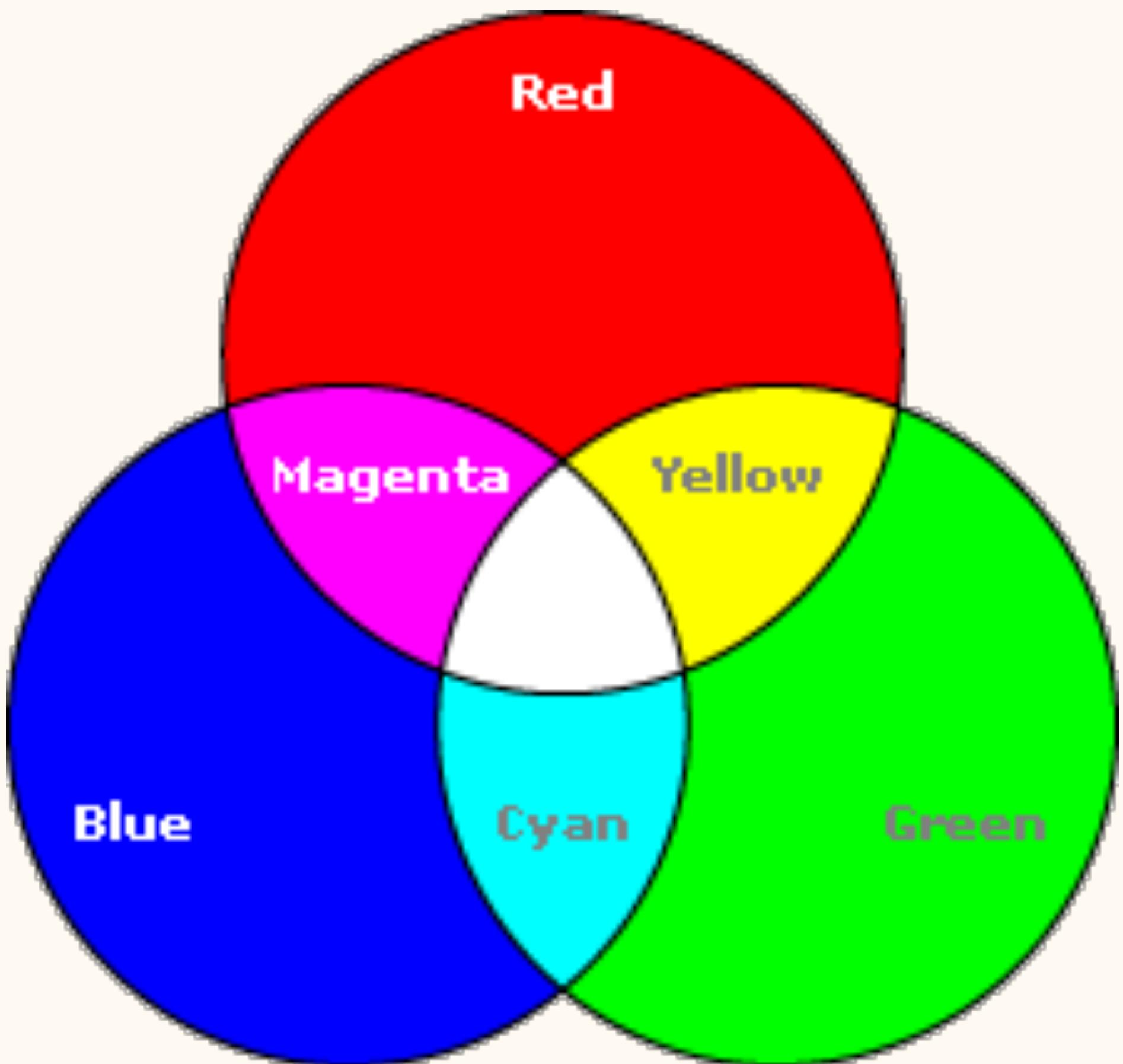
Textures
(Ed Catmull, 1974)

Color Model (色彩模型)

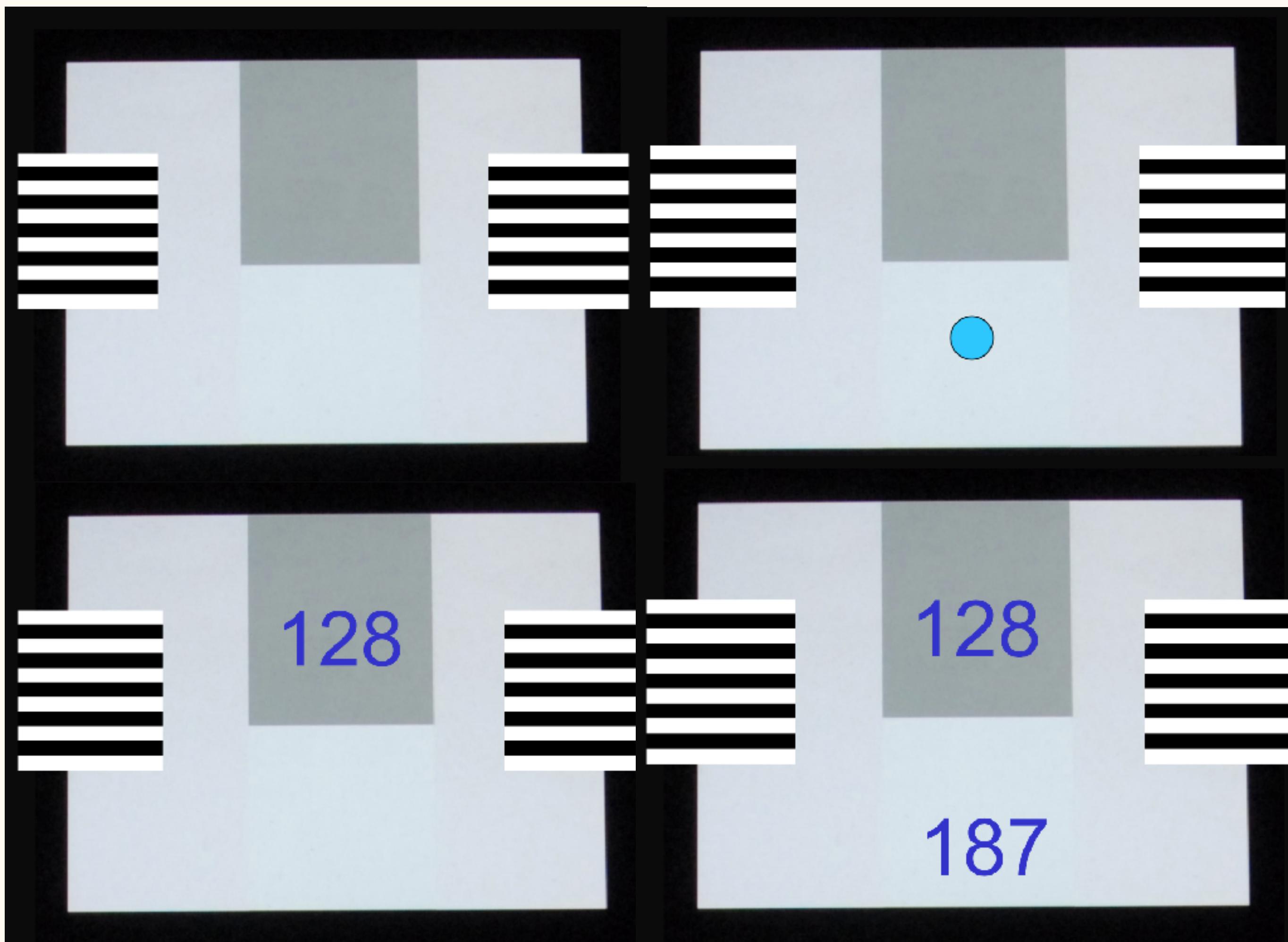
- 色彩模型 (Color Model)，也稱為色彩空間 (Color Space)
 - RGB color model
 - HSV color model
 - CMYK color model
 - CIE Lab color model
- RGB color for display
 - 24 bit color : (R8, G8, B8)
 - 256*256*256 colors
 - 全彩模式 (True color)
- 3D圖學通常是使用 (0.0 ~ 1.0)來表示顏色數值

RGB Color Model

- 光的三原色(r, g, b)
- 顏色範圍 0.0 – 1.0 for each component
 - Black = (0, 0, 0)
 - White = (1, 1, 1)
 - 相對亮度的概念
- Alpha channel
 - Opacity
- Color vector : (r, g, b, a)



灰階測試



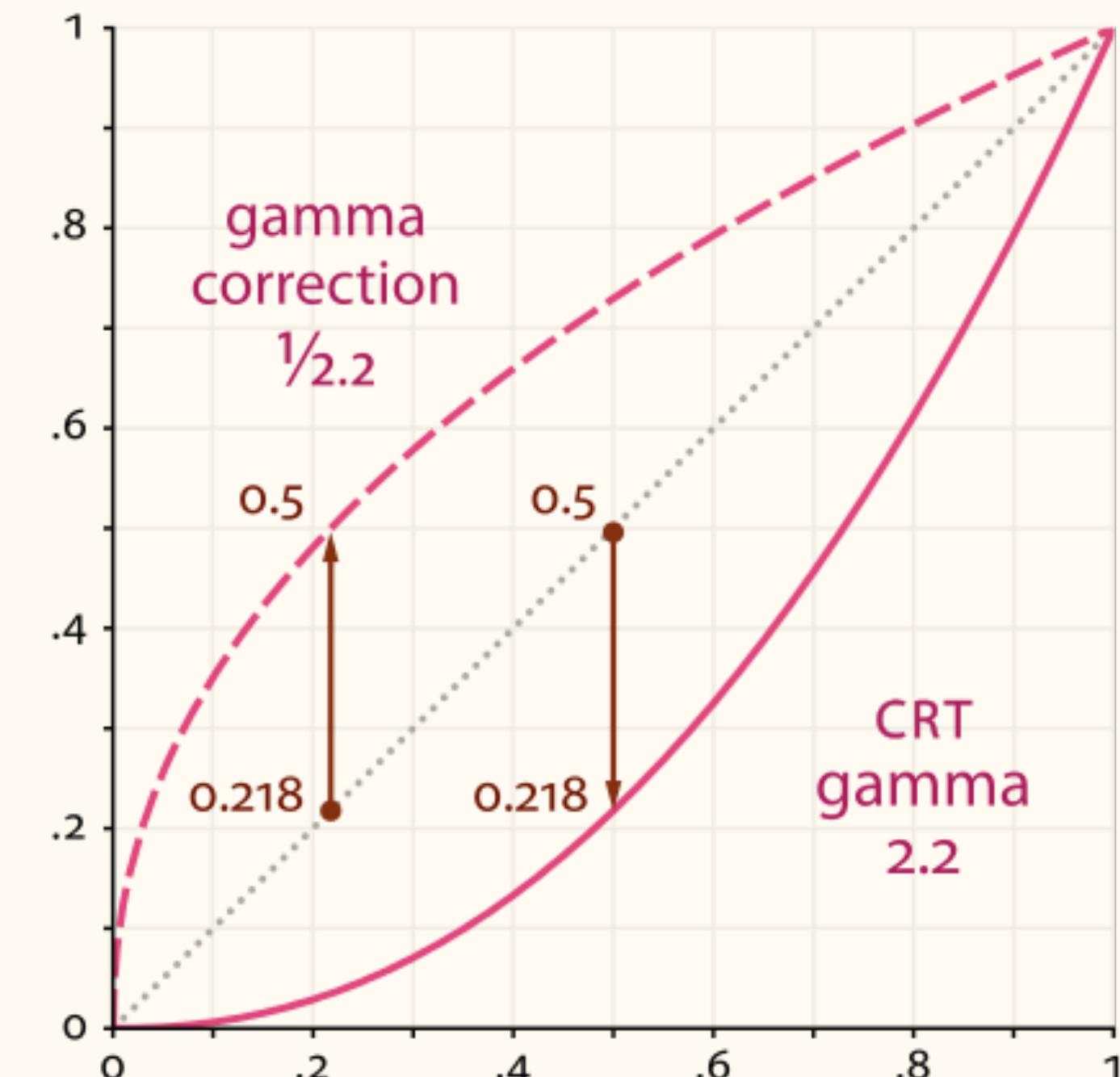
Gamma 校正，Gamma Correction

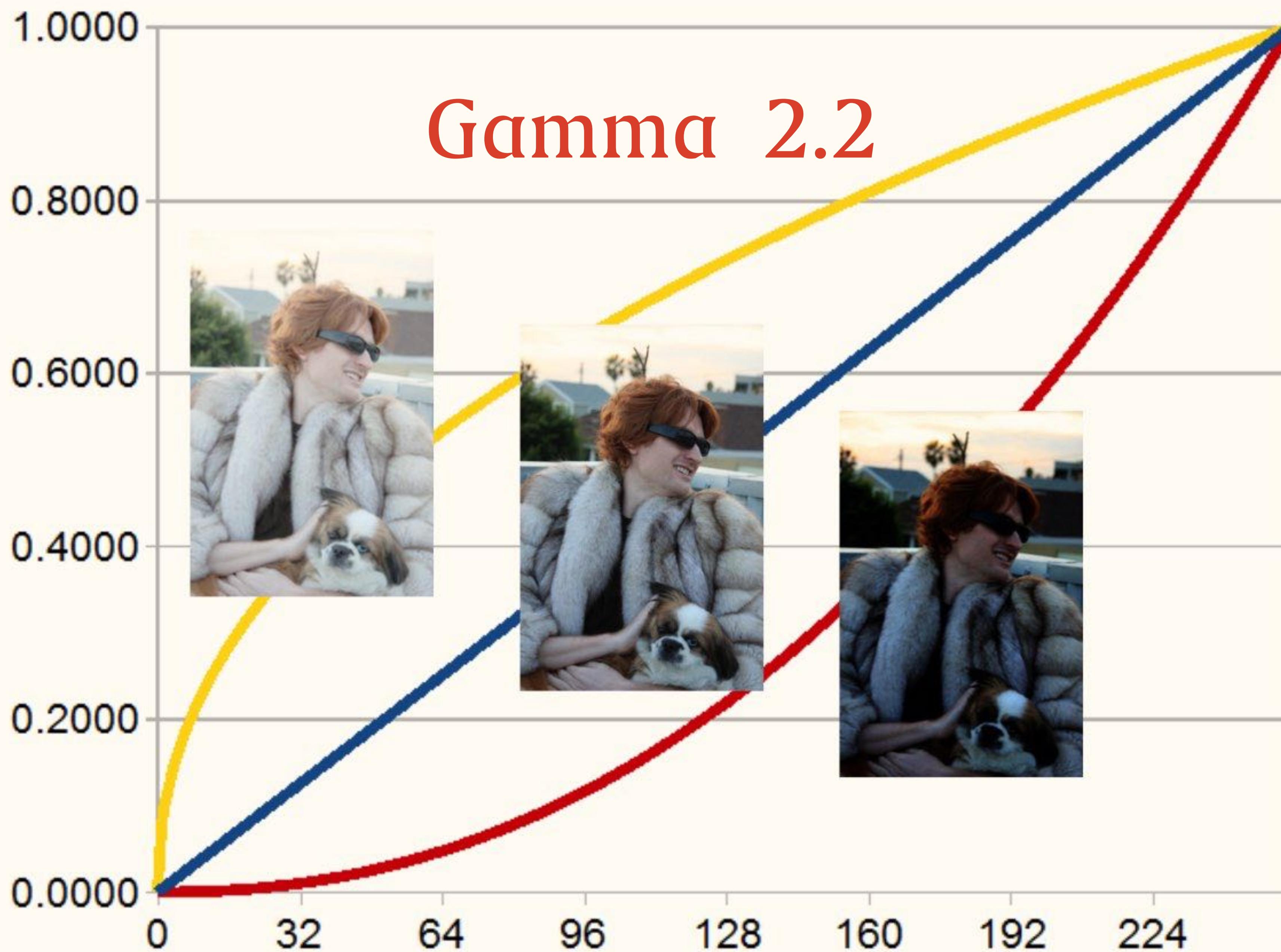
- 又稱：Gamma nonlinearity, gamma encoding
- 一般的照明的環境下，人類的視覺系統對暗色調的層次的敏感度比對亮色調要敏感且能分辨細節
- 其變化可以以指數函數來模擬

$$V_{out} = A * V_{in}^\gamma$$

通常 $A = 1$

$$\gamma = 2.2$$





Realtime 3D Computer Graphics

- 3D game graphics (3D 遊戲圖學)
 - 3D computer graphics for games
- 特點：
 - In real-time performance (即時/實時)
 - 每秒至少30幀畫面更新
 - 30 frame-per-second (fps) or up
 - GPU
 - Triangles mostly (通常只渲染三角面)
 - Multiple textures (多層貼圖架構)
 - Running shaders (靠 Shader 完成工作)
 - Video memory (圖形資料存放在視訊記憶體)

2D ?

- 現代的智能手機、平板、和電腦都是 3D Graphics 系統
- 2D = 3D in orthogonal projection view
 - All popular graphics APIs are 3D.
 - Direct3D
 - MS Windows
 - OpenGL
 - MS Windows / MacOS / Linux
 - OpenGL ES
 - Android / iOS / Web (WebGL)
 - Metal
 - iOS / MacOS

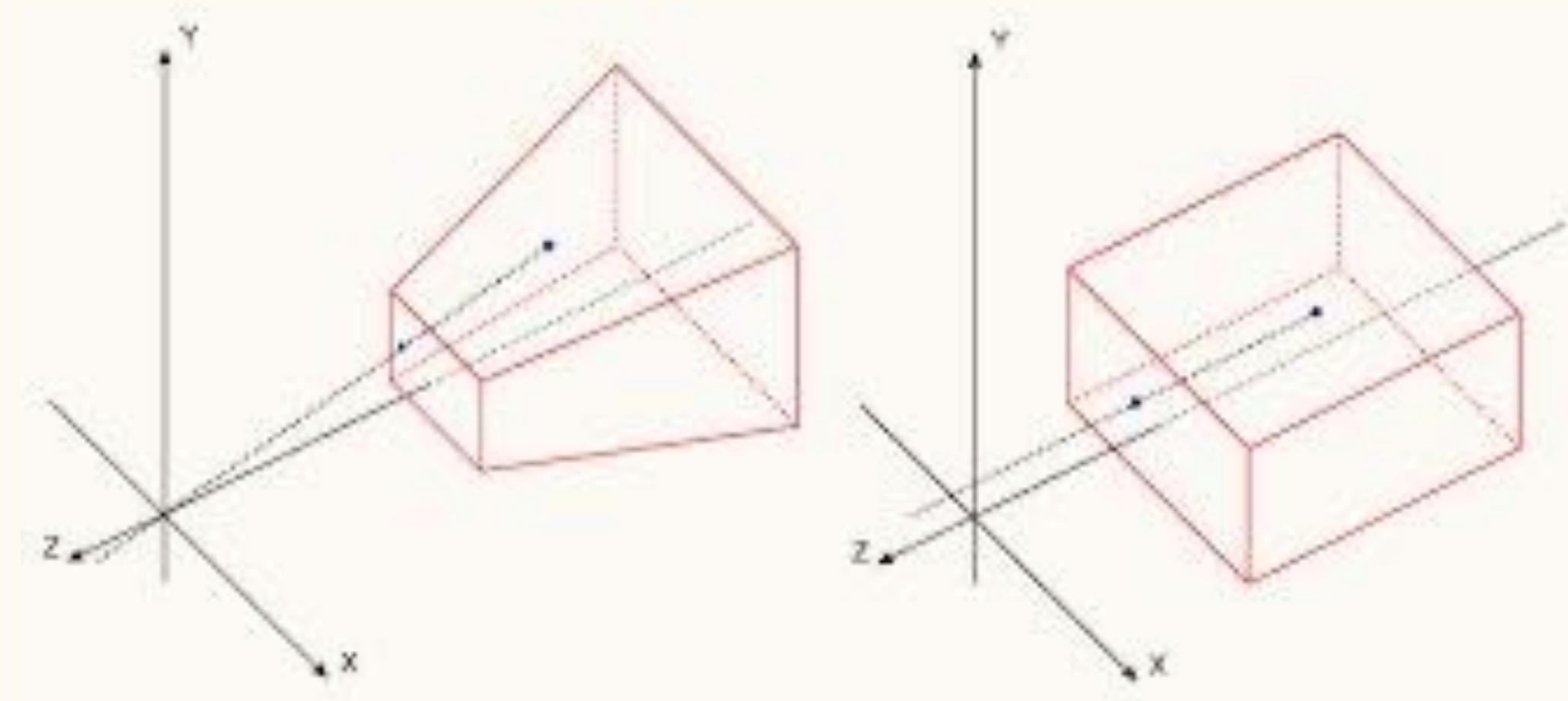
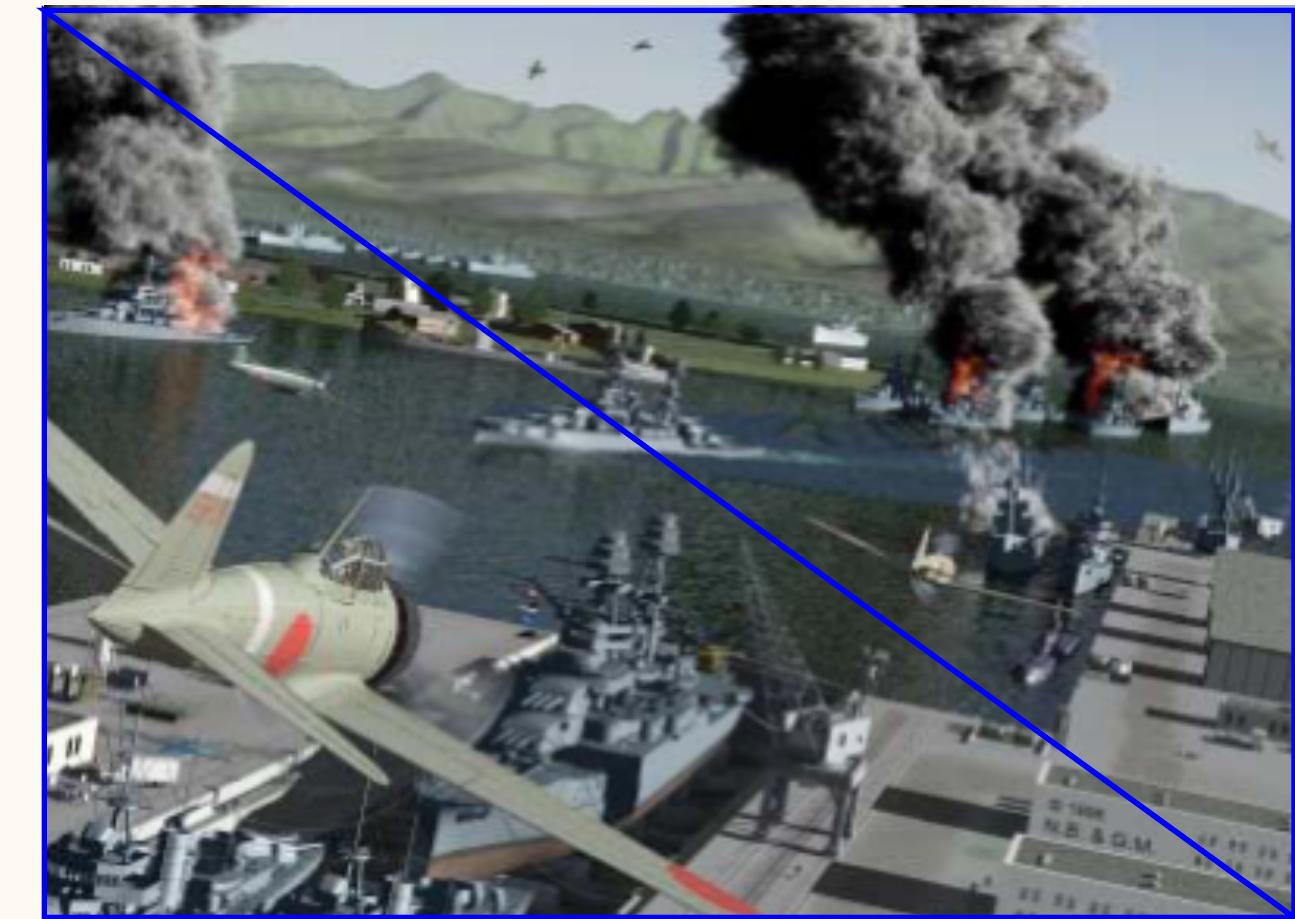


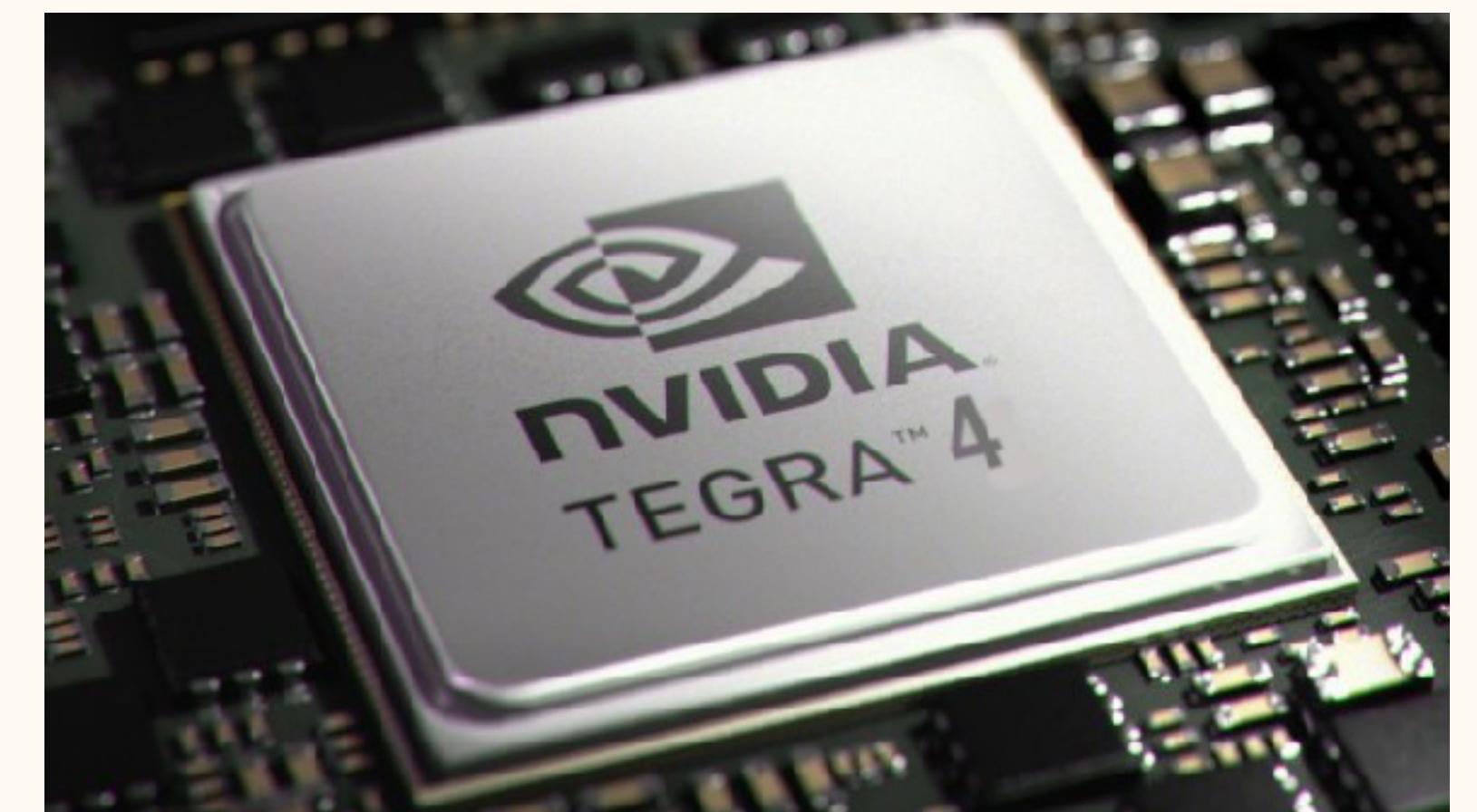
Image ? Texture ?

- 2D = Texture on Two Triangles
- A Texture = An Image + Addressing + Filtering
 - Texture surface = raw image data
 - Texture (sampling) = **fragment**
 - Filtering
 - **Sample** texels for screen pixels
 - “Texel” is the pixel in a texture
 - Addressing
 - The way to map texture on geometry



GPU

- The Graphics Hardware Revolution (繪圖硬體革命)
 - GPU-based Graphics Hardware (多核心圖形運算硬體)
 - Multi-core (1000 Cores +)
 - Designed for Floating-point Computation (浮點運算)
 - Vector Acceleration (向量加速運算)
 - Vertex/Pixel Processing in Parallel (平行運算)
 - Thin-weight Multi-threading (1024 threads +)
 - Super Computing Power (超級電腦運算能力)
 - Much, Much, and Much Faster Than CPU

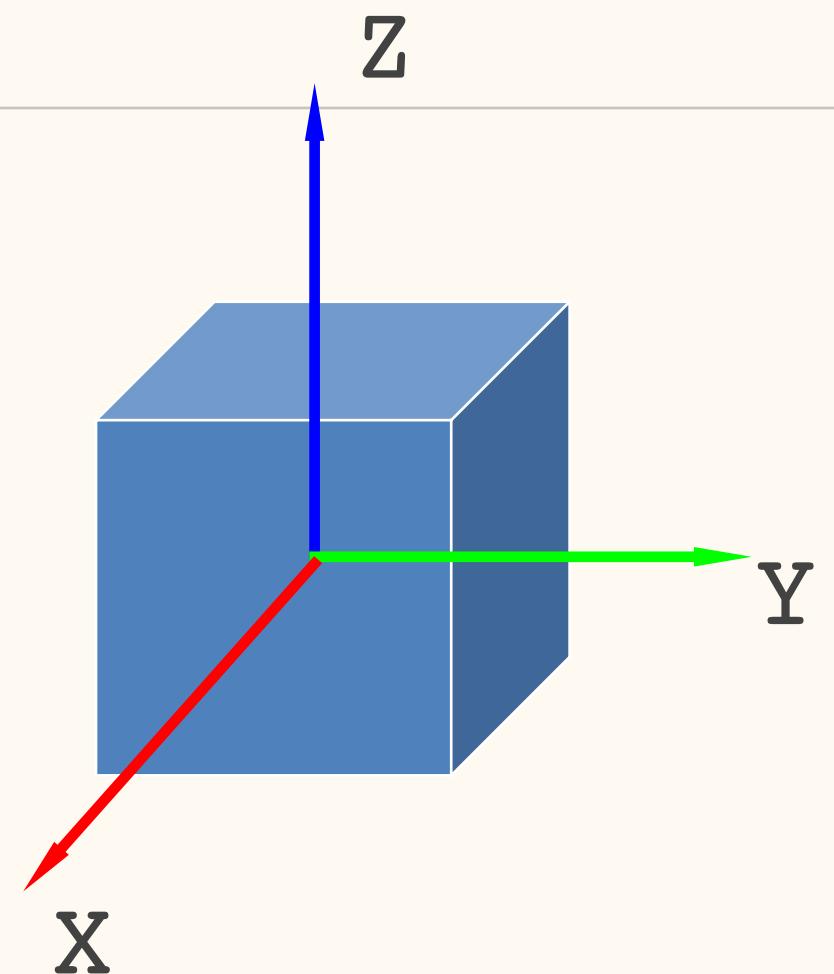


NVIDIA GeForce GTX 280

- NVIDIA-speak:
 - 240 stream processors
 - “SIMT execution” (automatic HW-managed sharing of instruction stream)
- Generic speak:
 - 30 processing cores
 - 8 SIMD functional units per core
 - 1 mul-add (2 flops) + 1 mul per functional units (3 flops/clock)
 - Best case: 240 mul-adds + 240 muls per clock
 - 1.3 GHz clock
 - $30 * 8 * (2 + 1) * 1.3 = 933 \text{ GFLOPS}$
- Mapping data-parallelism to chip:
 - Instruction stream shared across 32 fragments (16 for vertices)
 - 8 fragments run on 8 SIMD functional units in one clock
 - Instruction repeated for 4 clocks (2 clocks for vertices)

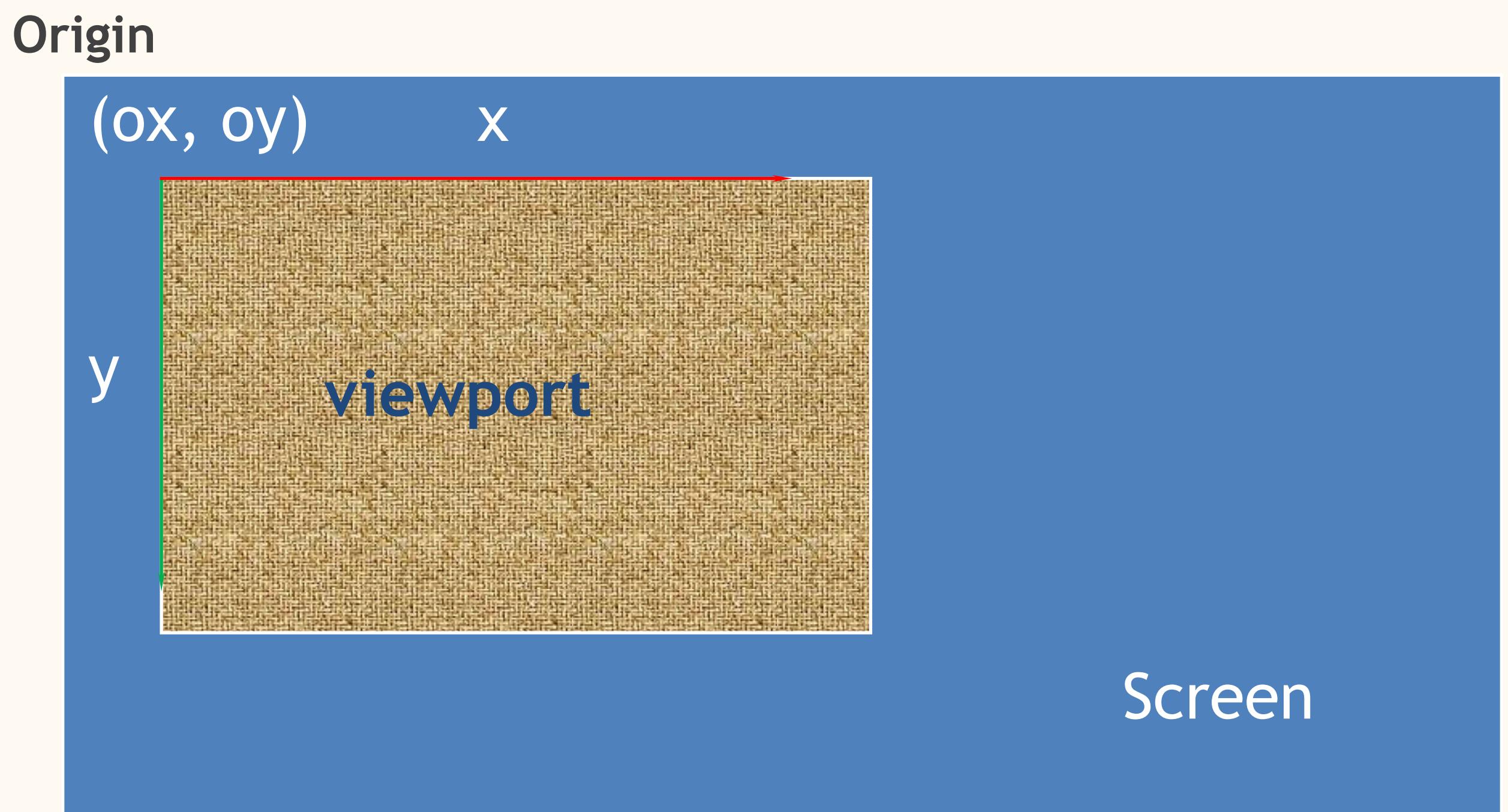
3D座標系統

- 直角坐標系
 - 笛卡兒座標系統(Cartesian coordinate system)
 - 通常使用右手定則，但非絕對
 - Unity 使用左手系統
 - Z-up or Y-up
 - 以往學校或3D軟體習慣以Z軸為朝上的座標軸，近來流行使用Y軸
 - Unity 使用 Y-up.
 - 3ds Max uses Z-up.



2D座標系統

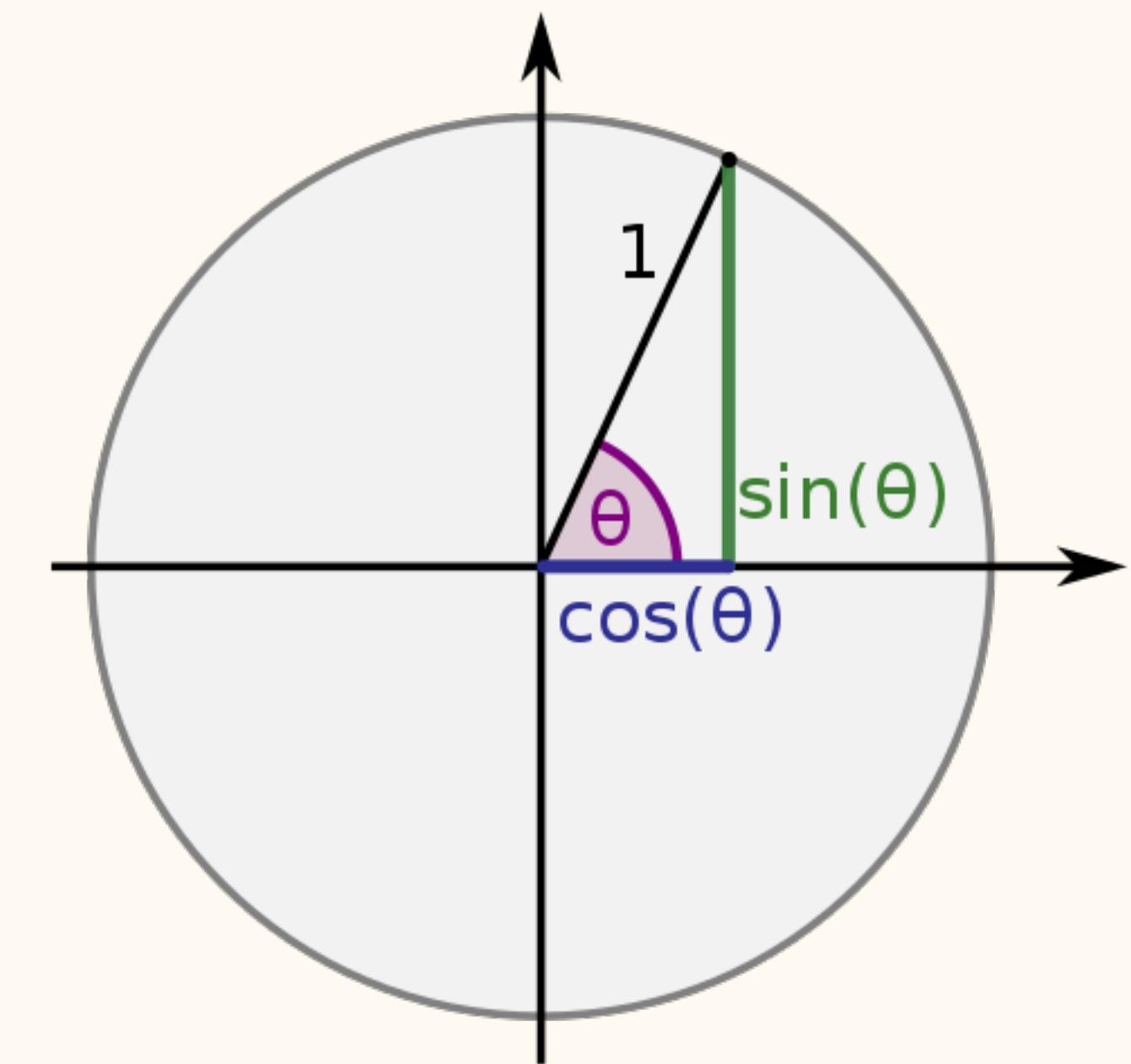
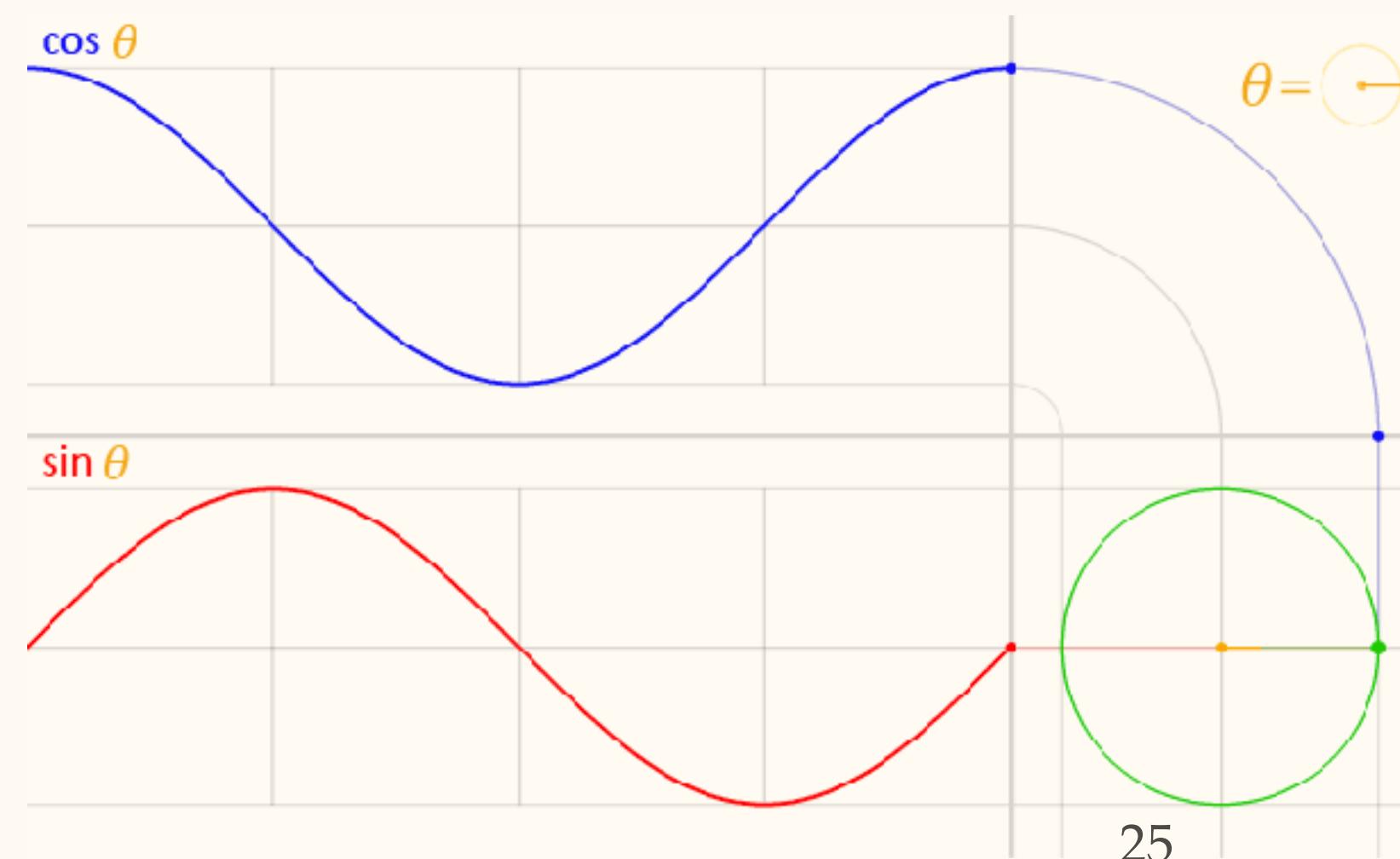
- 2D座標系統，我們習慣使用左手系統
 - Windows Desktop (Screen)
 - Viewport for rendering



Trigonometry, Vectors, and Matrices

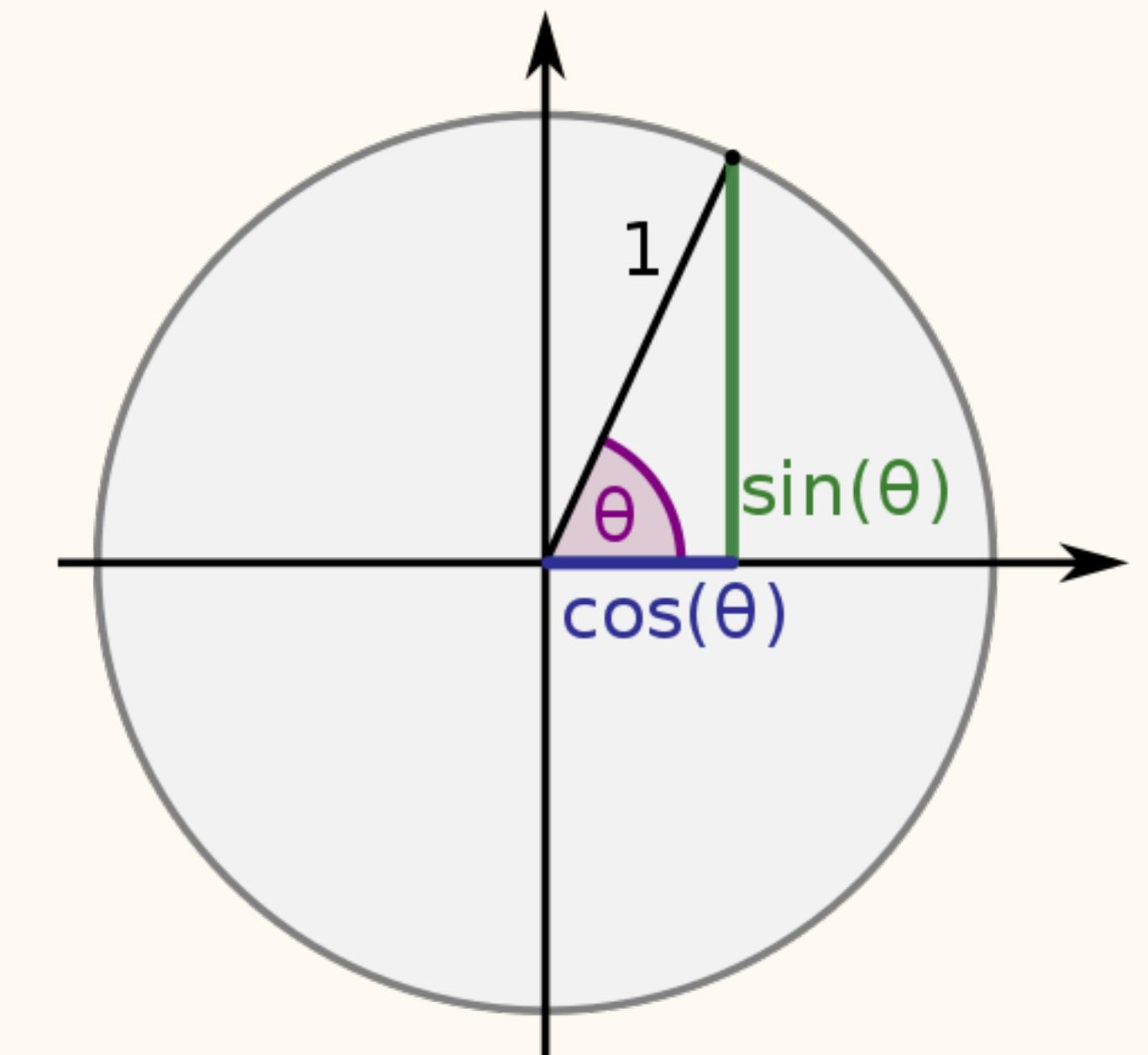
Trigonometric Functions (三角函數)

- 定義
 - 單位圓 (a circle with radius 1 unit)
 - 圓上的一點和原點連線與 X 軸的夾角, θ
 - 此點的 y-座標 = $\sin(\theta)$
 - 此點的 x-座標 = $\cos(\theta)$



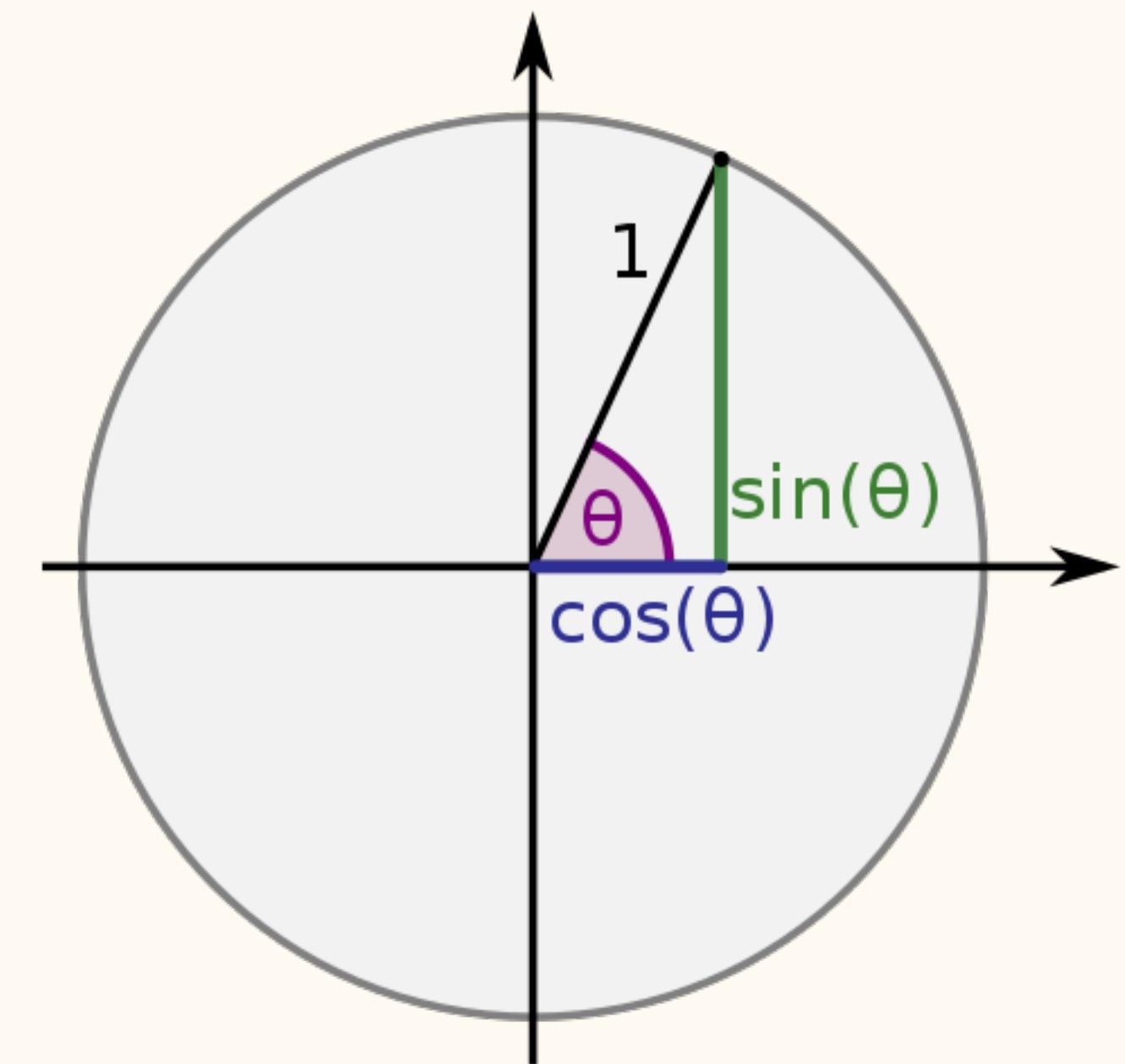
Trigonometric Functions (三角函數)

- 六個三角函數
 - Sine function, $\sin\theta$
 - Cosine function, $\cos\theta$
 - Tangent function, $\tan\theta = \sin\theta/\cos\theta$
 - Slope of the triangle
 - Cotangent function, $\cot\theta = 1/\tan\theta$
 - Secant function, $\sec\theta = 1/\cos\theta$
 - Cosecant function, $\csc\theta = 1/\sin\theta$
- 主要的函數還是 : $\sin\theta$
 - $\cos\theta = \sin(\theta + \pi/2)$



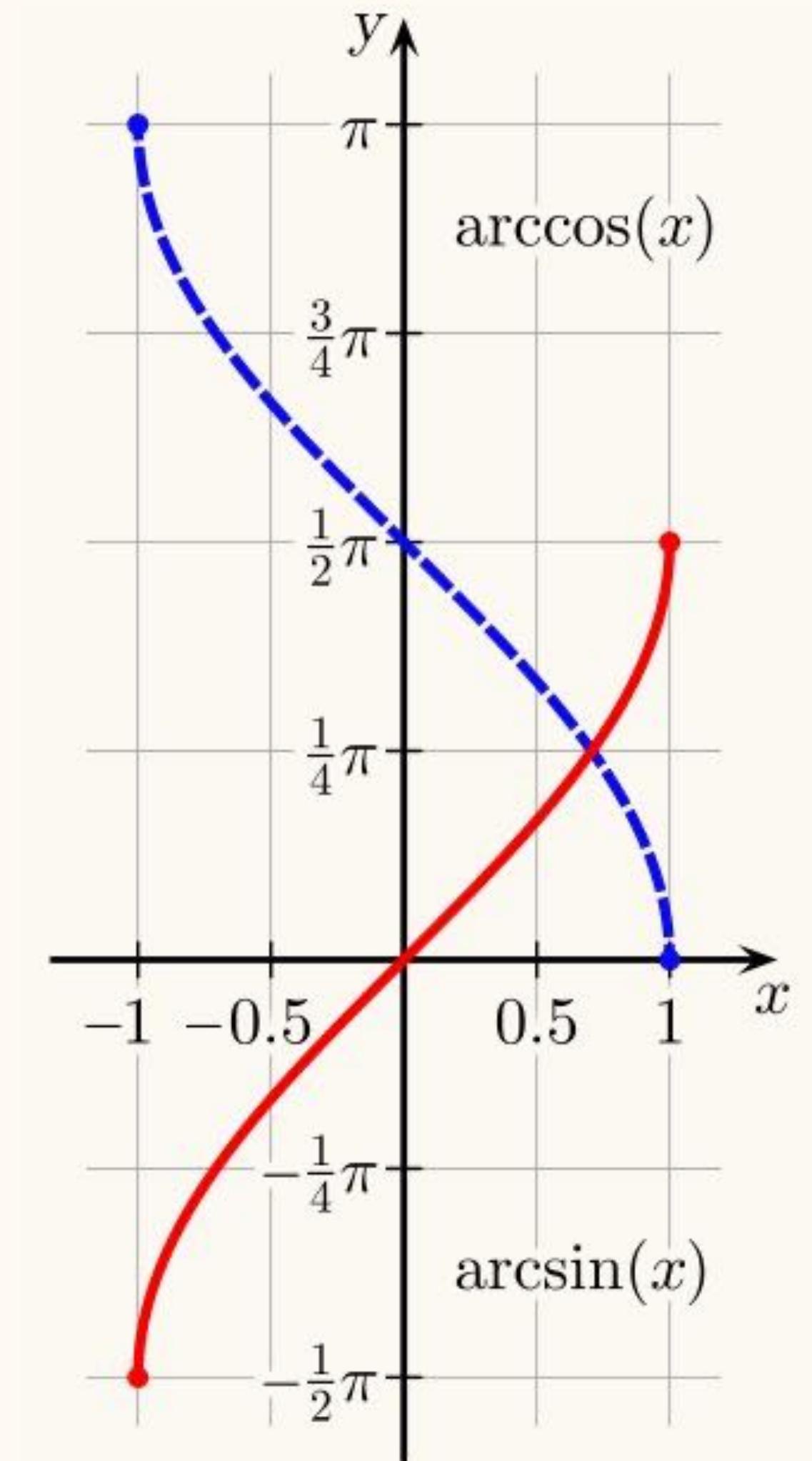
Trigonometric Functions (三角函數)

- 常用的三角函數公式：
 - $\sin^2\theta + \cos^2\theta = 1$
 - $\sin 2\theta = 2\sin\theta\cos\theta$
 - $\cos 2\theta = \cos^2\theta - \sin^2\theta$
 - $\tan 2\theta = 2\tan\theta / (1 - \tan^2\theta)$



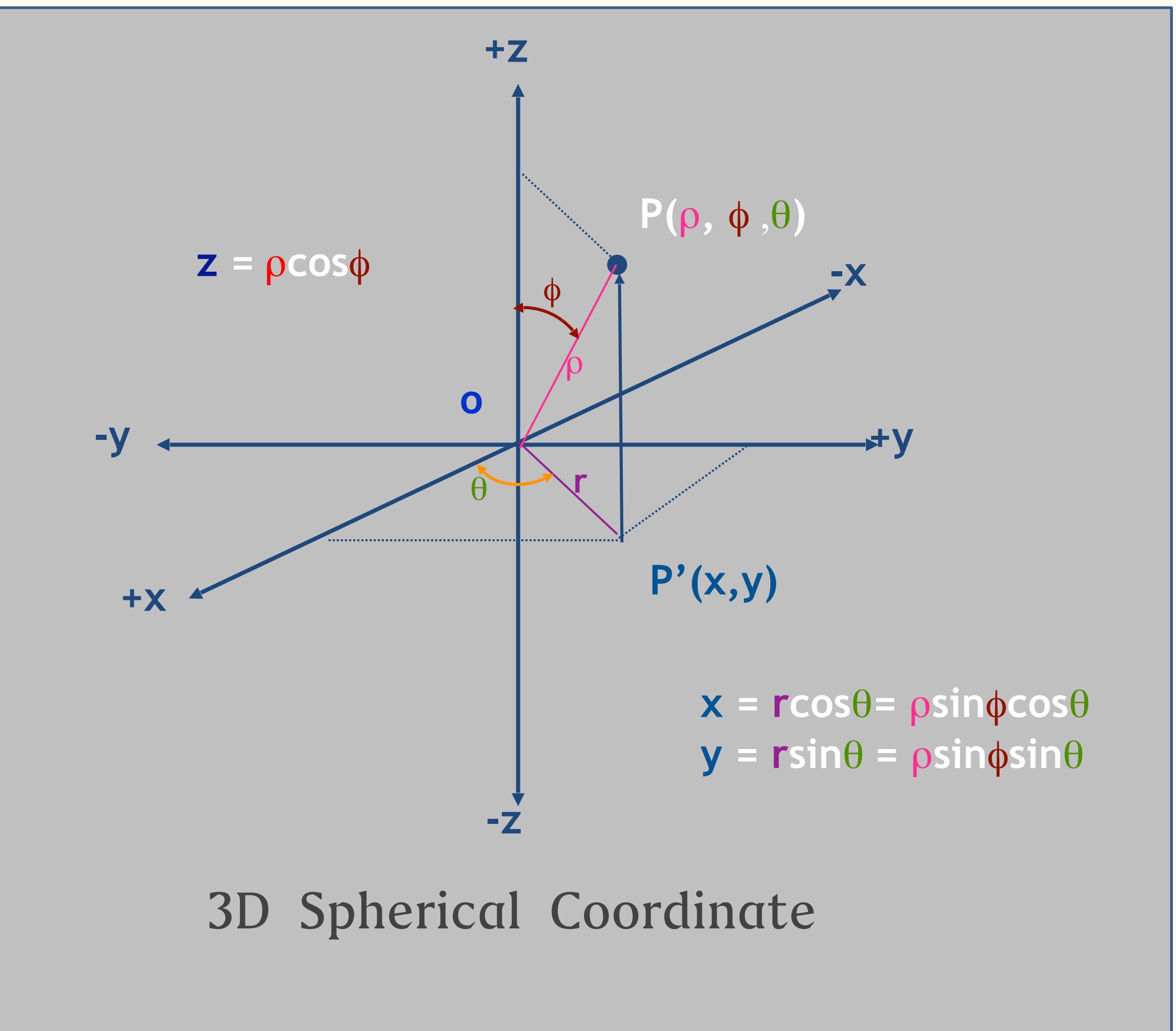
Inverse Trigonometry (反三角函數)

- $\sin^{-1}(\theta)$
 - arcsine function
- $\cos^{-1}(\theta)$
- $\tan^{-1}(\theta)$
- $\cot^{-1}(\theta)$
- $\sec^{-1}(\theta)$
- $\csc^{-1}(\theta)$
- 小心使用反三角函數：
 - Numerical error in floating-point computing



Spherical Coordinate System

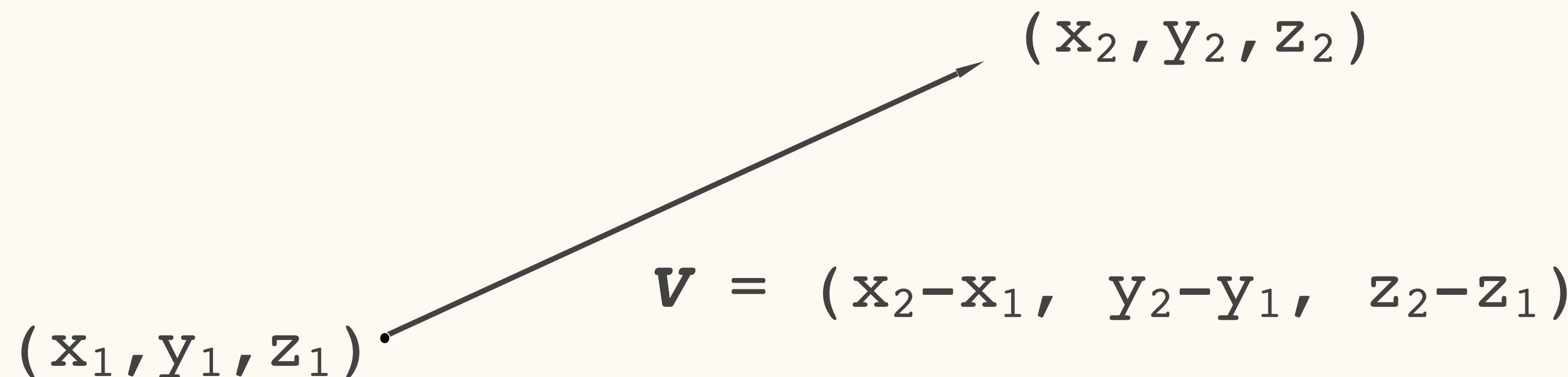
- 球座標系
 - $P(\rho, \phi, \theta)$
 - ρ : the distance to origin
 - ϕ : the angle from Z-axis
 - θ : the angle from X-axis to the projection of the P on XY-plane



3D Spherical Coordinate

向量 Vectors

- A vector 具備 magnitude (大小) and direction (方向).
- A 3D vector :
 - $V = (v_1, v_2, v_3)$
- A ray (射線) 具備 position(位置), magnitude (大小) and direction (方向).



向量 Vectors

- 向量長度

$$\|V\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

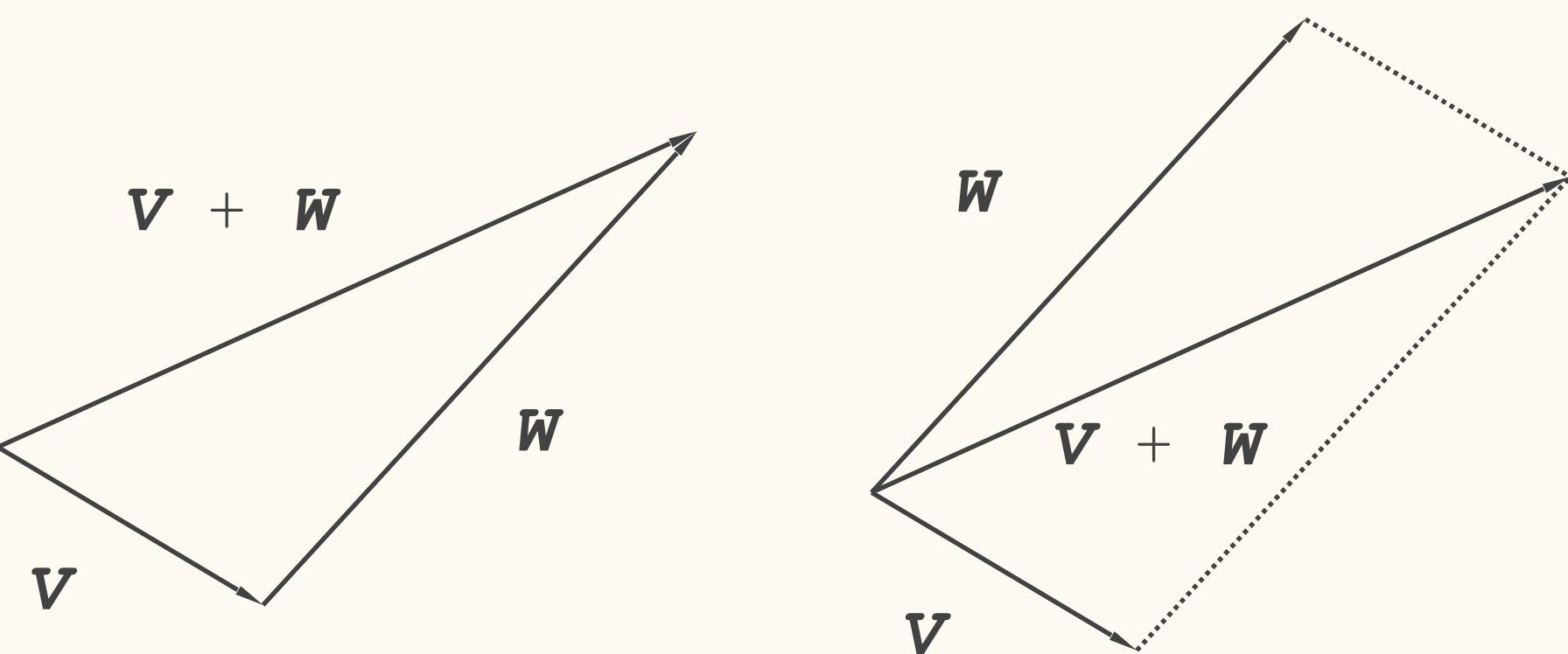
- Unit vector
 - 單位向量
 - Length = 1.0

$$U = \frac{V}{\|V\|}$$

向量 Vectors

- 向量加法

$$\begin{aligned}X &= \mathbf{v} + \mathbf{w} \\&= (x_1, y_1, z_1) \\&= (v_1 + w_1, v_2 + w_2, v_3 + w_3)\end{aligned}$$



向量 Vectors

- 向量外積 (Cross product)

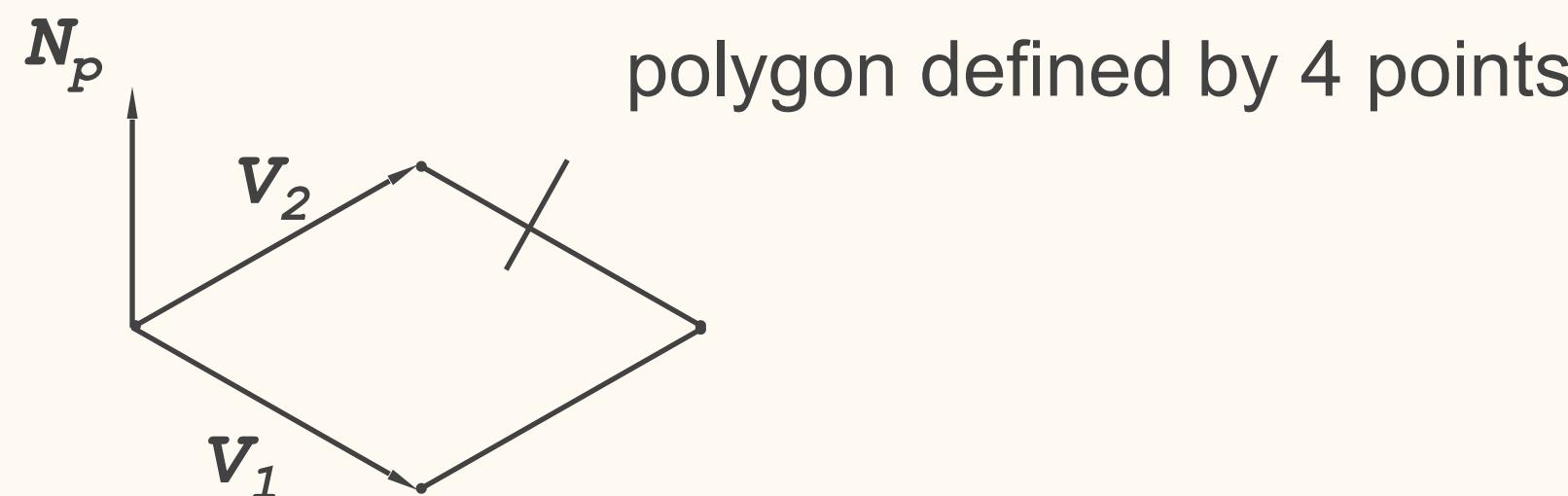
$$\begin{aligned} \mathbf{X} &= \mathbf{V} \times \mathbf{W} \\ &= (\mathbf{v}_2\mathbf{w}_3 - \mathbf{v}_3\mathbf{w}_2) \mathbf{i} + (\mathbf{v}_3\mathbf{w}_1 - \mathbf{v}_1\mathbf{w}_3) \mathbf{j} + (\mathbf{v}_1\mathbf{w}_2 - \mathbf{v}_2\mathbf{w}_1) \mathbf{k} \end{aligned}$$

where \mathbf{i} , \mathbf{j} and \mathbf{k} are standard unit vectors :

$$\mathbf{i} = (1, 0, 0), \quad \mathbf{j} = (0, 1, 0), \quad \mathbf{k} = (0, 0, 1)$$

- 主要應用：三點決定一個平面，平面法向量可以由向量外積求得

$$\mathbf{N}_P = \mathbf{v}_1 \times \mathbf{v}_2$$



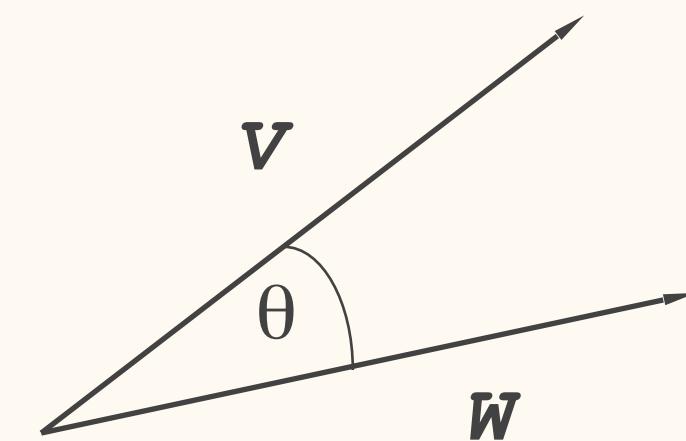
向量 Vectors

- 向量內積 (Dot product)

$$\begin{aligned}x &= \mathbf{v} \cdot \mathbf{w} \\&= v_1 w_1 + v_2 w_2 + v_3 w_3\end{aligned}$$

- 主要應用：求投影長

$$\cos \theta = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|}$$



矩陣 Matrix

- 定義

- A is a “ $m \times n$ ” matrix :

$$A = (a_{ij}) = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$

- 轉置矩陣 (Transpose)

$$C = A^T = (a_{ji})$$

- 矩陣加法運算

$$C = A + B \quad c_{ij} = a_{ij} + b_{ij}$$

矩陣 Matrix

- 純量與矩陣相乘

$$C = \alpha A$$

$$(c_{ij}) = \alpha(a_{ij})$$

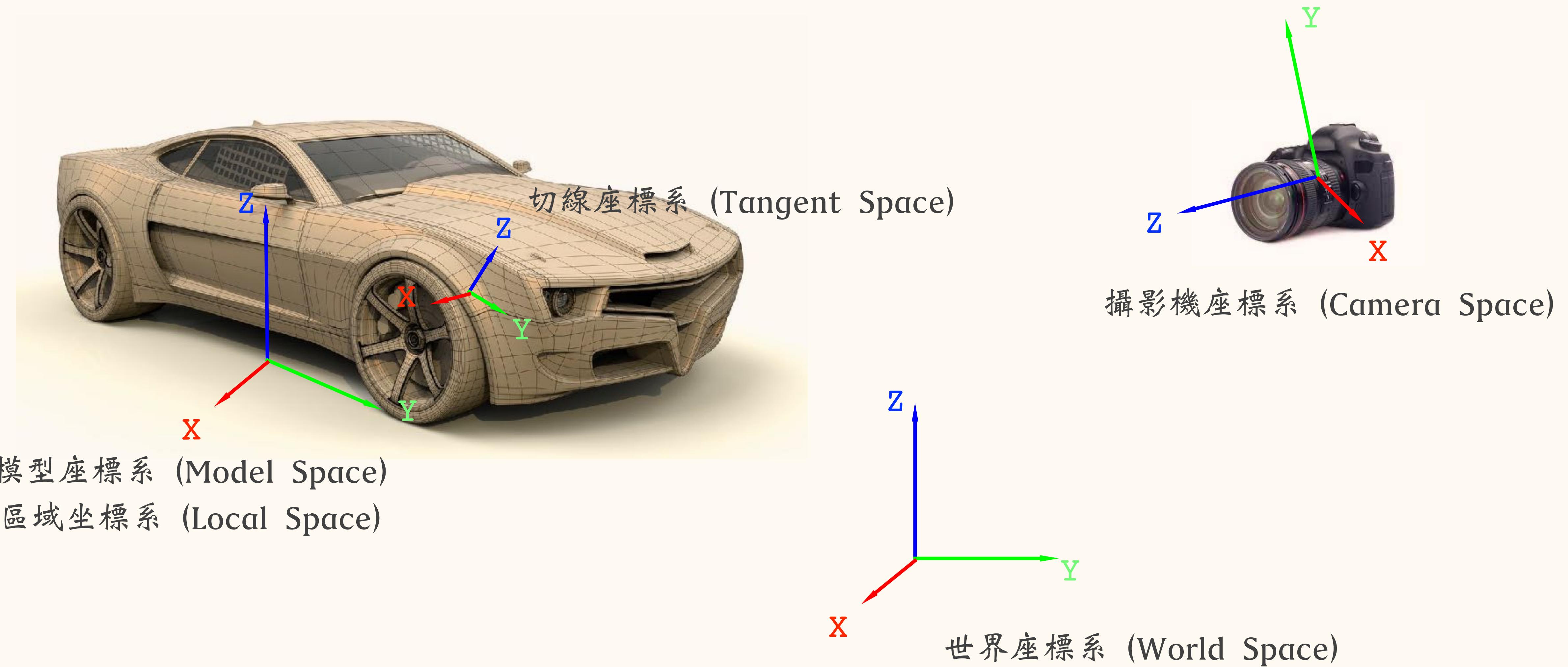
- 矩陣乘法

$$C = AB$$

$$(c_{ij}) = \sum_{k=1}^r (a_{ik})(b_{kj})$$

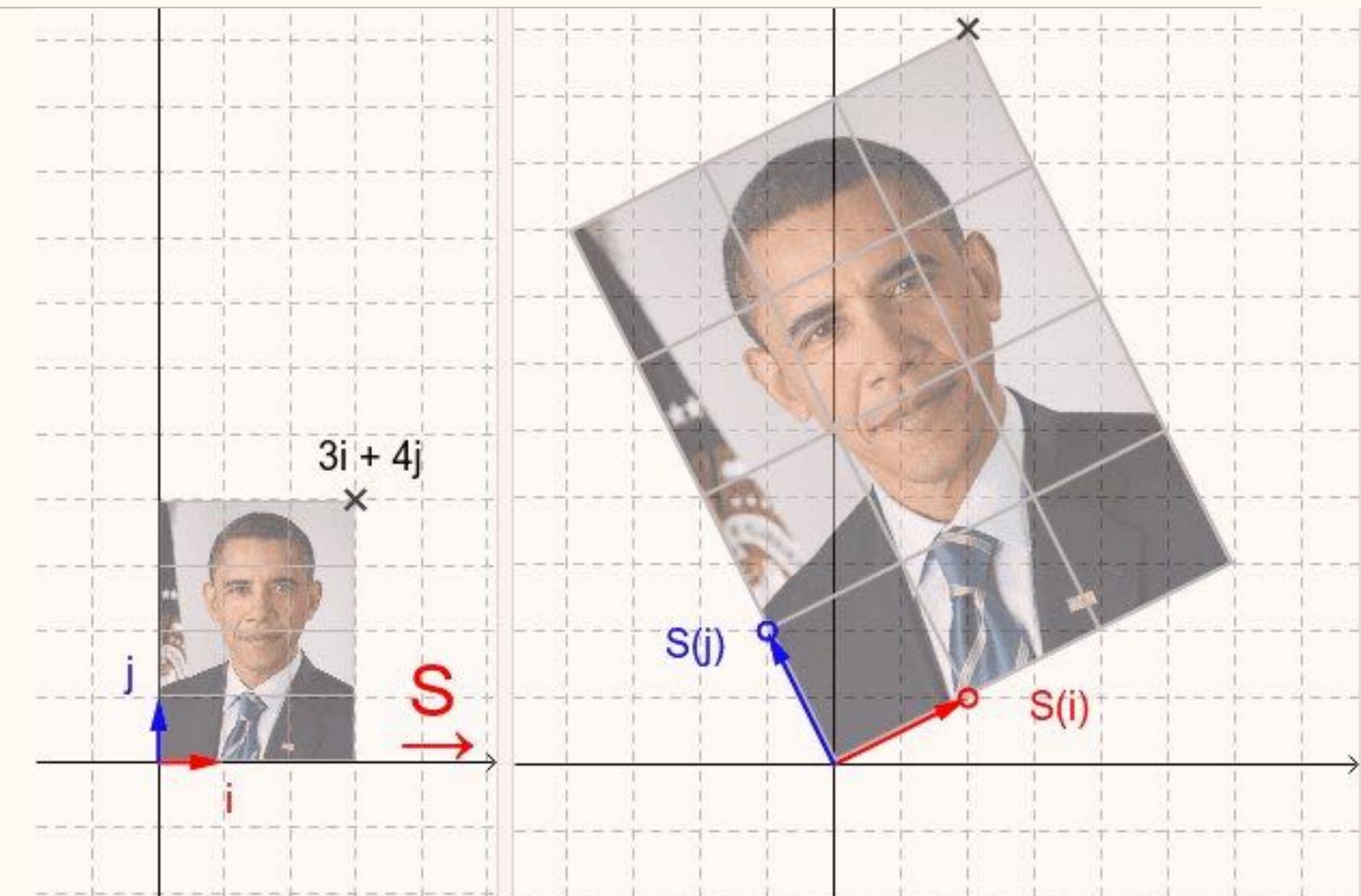
Affine Transformations

3D 座標系統



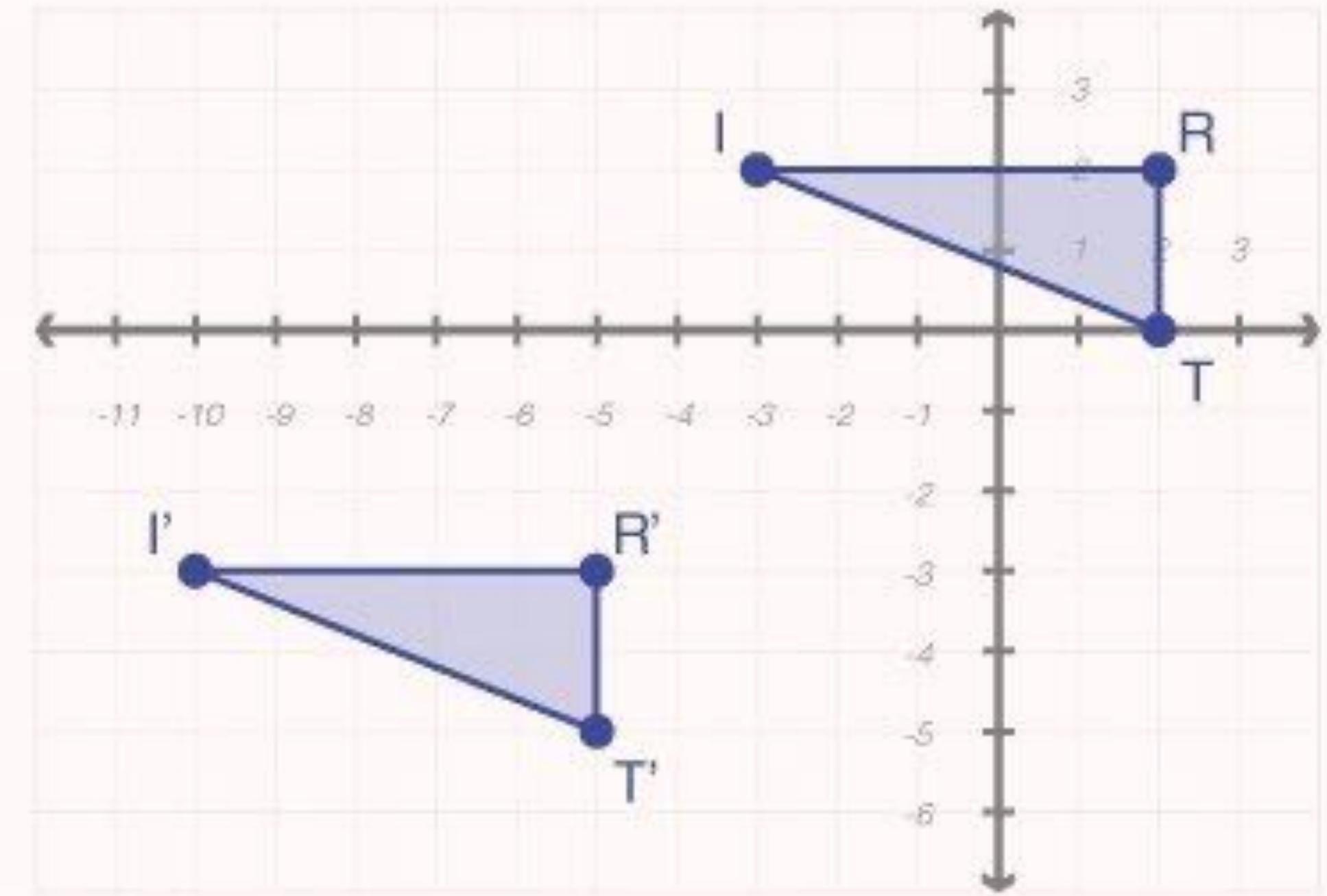
線性轉換 Linear Transformations

- 包含：
 - Scale, rotation, shear, and mirror
- 性質：
 - Origin maps to origin 原點映射至原點
 - Lines map to lines 直線映射至直線
 - Parallel lines remain parallel 平行線保持平行
 - Ratios are preserved 比例不變



仿射轉換 Affine Transformations

- Affine transformations = Linear transformation + Translation (平移)
 - Linear transformations, and translation



Transformations in Matrix Form

- 一個3D的點可以用矩陣來表示： $V^T = [x \ y \ z]$
 - Column matrix
 - 因此，平移、旋轉、縮放可以運用矩陣來表示

$$V' = V + D$$

D 是平移向量

$$V' = SV$$

S 是縮放矩陣

$$V' = RV$$

R 是旋轉矩陣

平移 Translation

- Translation transformation :

$$x' = x + T_x$$

$$y' = y + T_y$$

$$z' = z + T_z$$

$$V' = V + D$$

D is the translation vector : (T_x, T_y, T_z)

旋轉 Rotations

- 3D 應用常用的線性轉換的工具
 - 常見的數學工具：
 - 尤拉角 (Euler angles)：
 - 由三個分別對 x / y / z 軸的旋轉所組成的： $(\theta_x, \theta_y, \theta_z)$
 - 對任意軸旋轉 (Axis, Angle)： (x, y, z, θ)
 - 四元數 Quaternion： (w, x, y, z)
 - 在此我們僅以對 z 軸旋轉來討論旋轉與矩陣的關係
 - Rotation with Z-axis : $V' = R_z V$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

縮放 Scaling

- 縮放轉換 Scaling Transformation :

$$x' = xS_x$$

$$y' = xS_y$$

$$z' = xS_z$$

$$V' = SV$$

$$S = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$$

連續的 Transformations

- 矩陣相乘沒有交換律

$$A \times B \neq B \times A$$

- 矩陣相乘有連結律

$$\begin{array}{ccc} V_1 = M_1 V & \equiv & M_3 = M_2 M_1 \\ V_2 = M_2 V_1 & \equiv & V_2 = M_3 V \end{array}$$

連續的 Transformations

- 如果有一個模型有10萬個點，連續在空間中運動：

$$V' = \underline{M_n M_{n-1} M_{n-2} \dots M_1 V}$$

$$V = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \\ y_1 & y_2 & y_3 & \dots & y_n \\ z_1 & z_2 & z_3 & \dots & z_n \end{bmatrix}$$

哪一部分先相乘呢？

齊次座標系 Homogeneous Coordinate System

- 為了能讓平移轉換可以利用到矩陣相乘的優點，我們導入 homogeneous coordinate system

$$V^T = [x \ y \ z \ w] \quad w = 1$$

$$V' = TV = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

齊次座標系 Homogeneous Coordinate System

- Scaling can be represented as :

$$V' = SV \quad S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

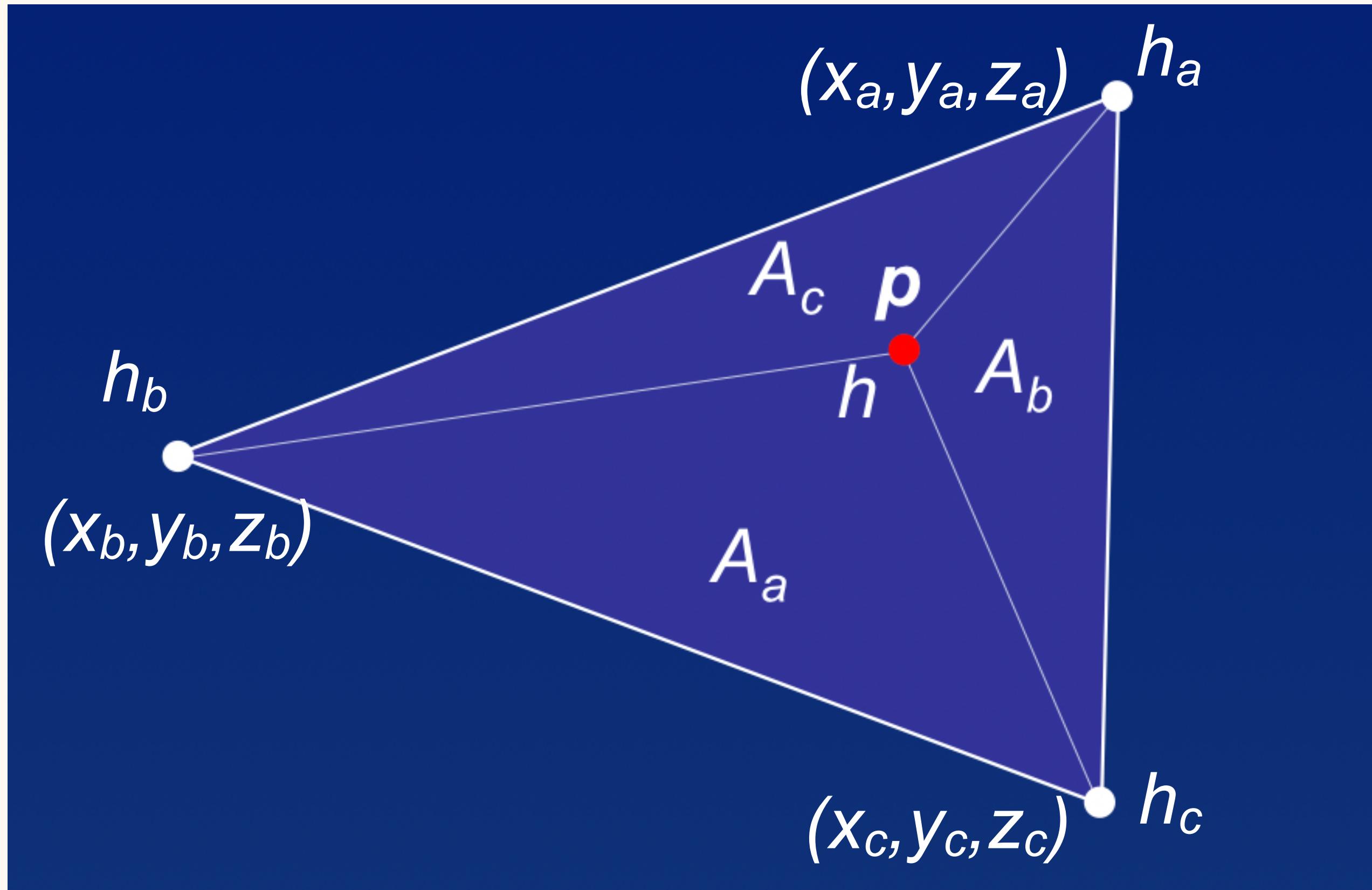
齊次座標系 Homogeneous Coordinate System

- Rotation to z-axis can be represented as :

$$V' = R_z V \quad R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Intersections

重心座標系 Barycentric Coordinate System



$$h = h_a \frac{A_a}{A} + h_b \frac{A_b}{A} + h_c \frac{A_c}{A}$$

其中 $A = A_a + A_b + A_c$

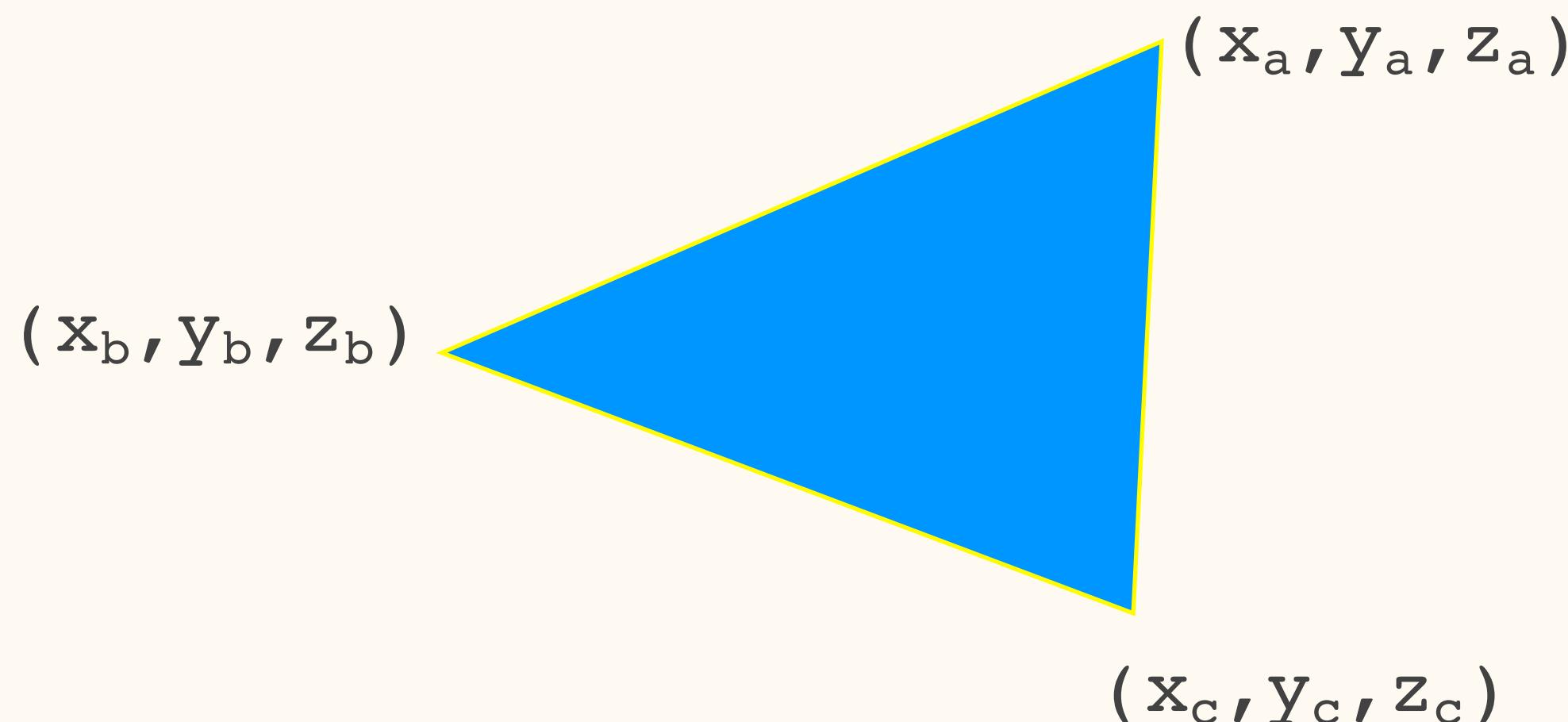
如果 $A_a < 0 \parallel A_b < 0 \parallel A_c < 0$

p 在三角形的外側

三角形面積 - 2D Solution

- 如果只考慮 2D 面積：

$$A = \frac{1}{2} \begin{vmatrix} x_a & y_a \\ x_b & y_b \\ x_c & y_c \\ x_a & y_a \end{vmatrix}$$

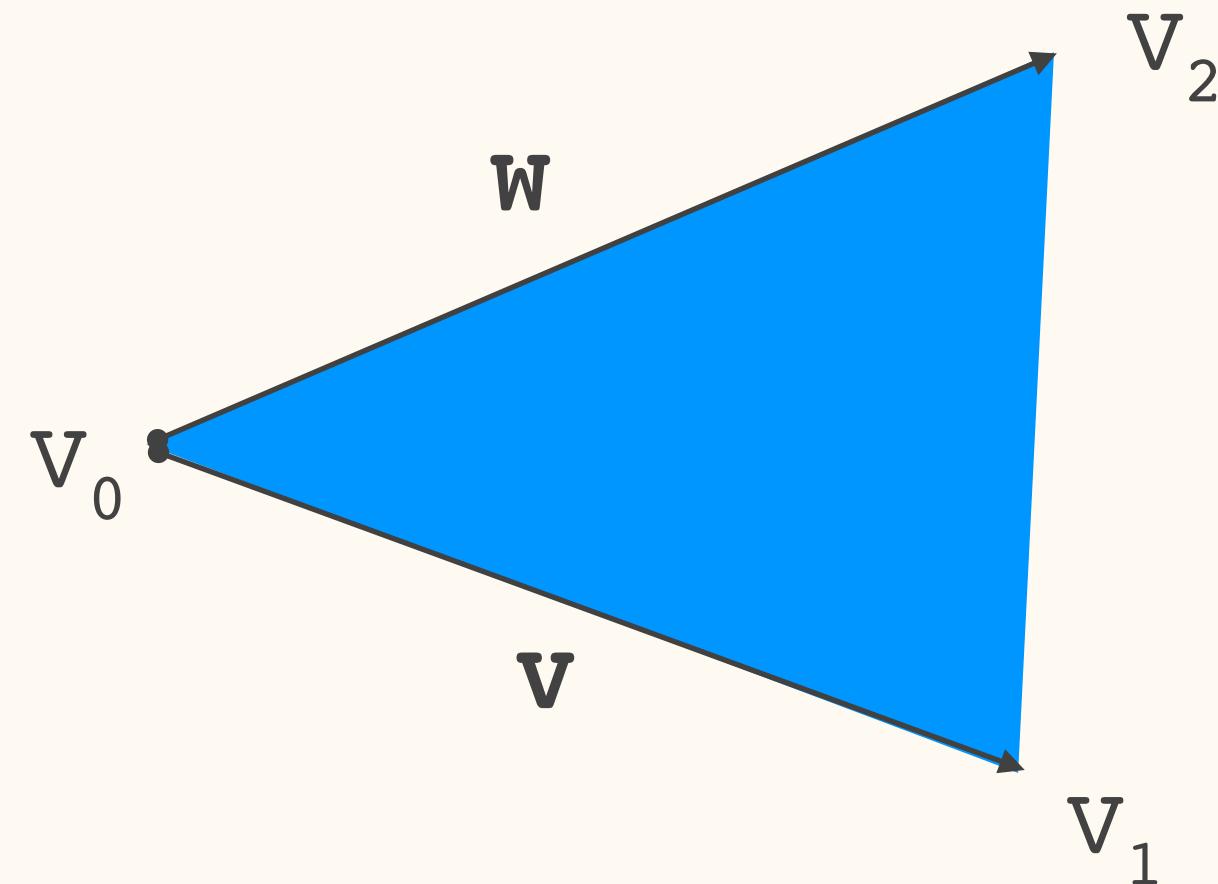


$$= \frac{1}{2} (x_a * y_b + x_b * y_c + x_c * y_a - x_b * y_a - x_c * y_b - x_a * y_c)$$

三角形面積 - 3D Solution

$$A(\Delta) = \frac{1}{2} \|v \times w\|$$

$$= \frac{1}{2} \|(V_1 - V_0) \times (V_2 - V_0)\|$$

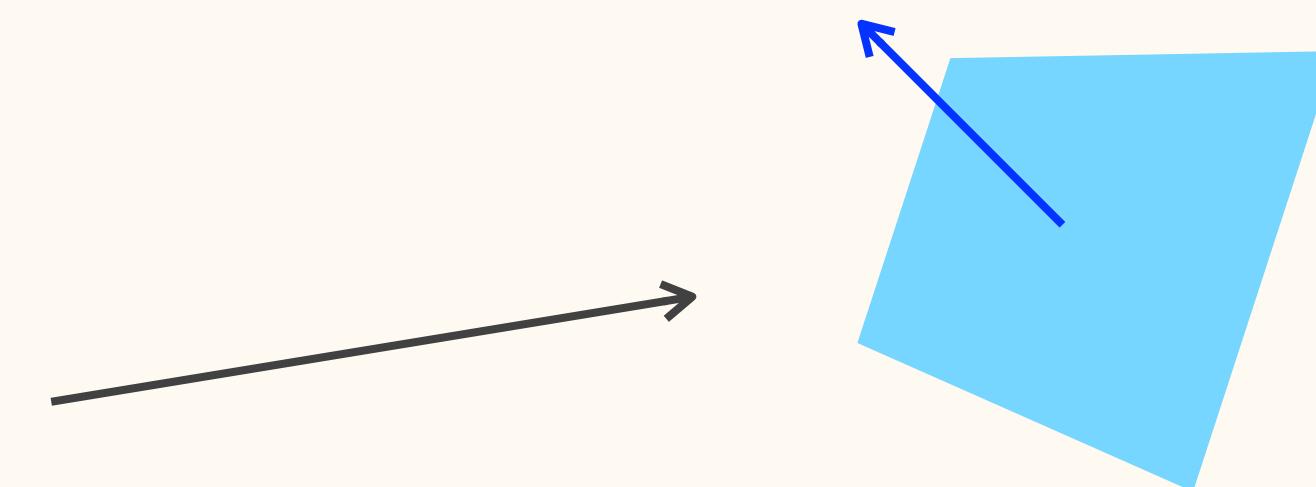


Barycentric Coordinate System 應用

- 地表追蹤 Terrain following
 - 求人物站在地面的高度
- 螢幕點擊3D物件 Hit test
- 求射線三角形與的焦點 Ray cast
- 碰撞偵測解焦點計算

物件相交 Intersection

- 許多遊戲的幾何問題如碰撞偵測等，通常最後是需要解一條直線與平面的交點
 - 模型都是三角面構成的
 - 三點可決定一個平面
 - 求三角形的法相量(Normal Vector)，可以求得平面方程式
 - 通常在遊戲中是使用參數方程式來表示一條直線
 - 通常在遊戲中是使用射線較多



平面方程式

$$ax + by + cz + d = 0$$

$$N \simeq (a, b, c)$$

$$v_1 = p_1 - p_0$$

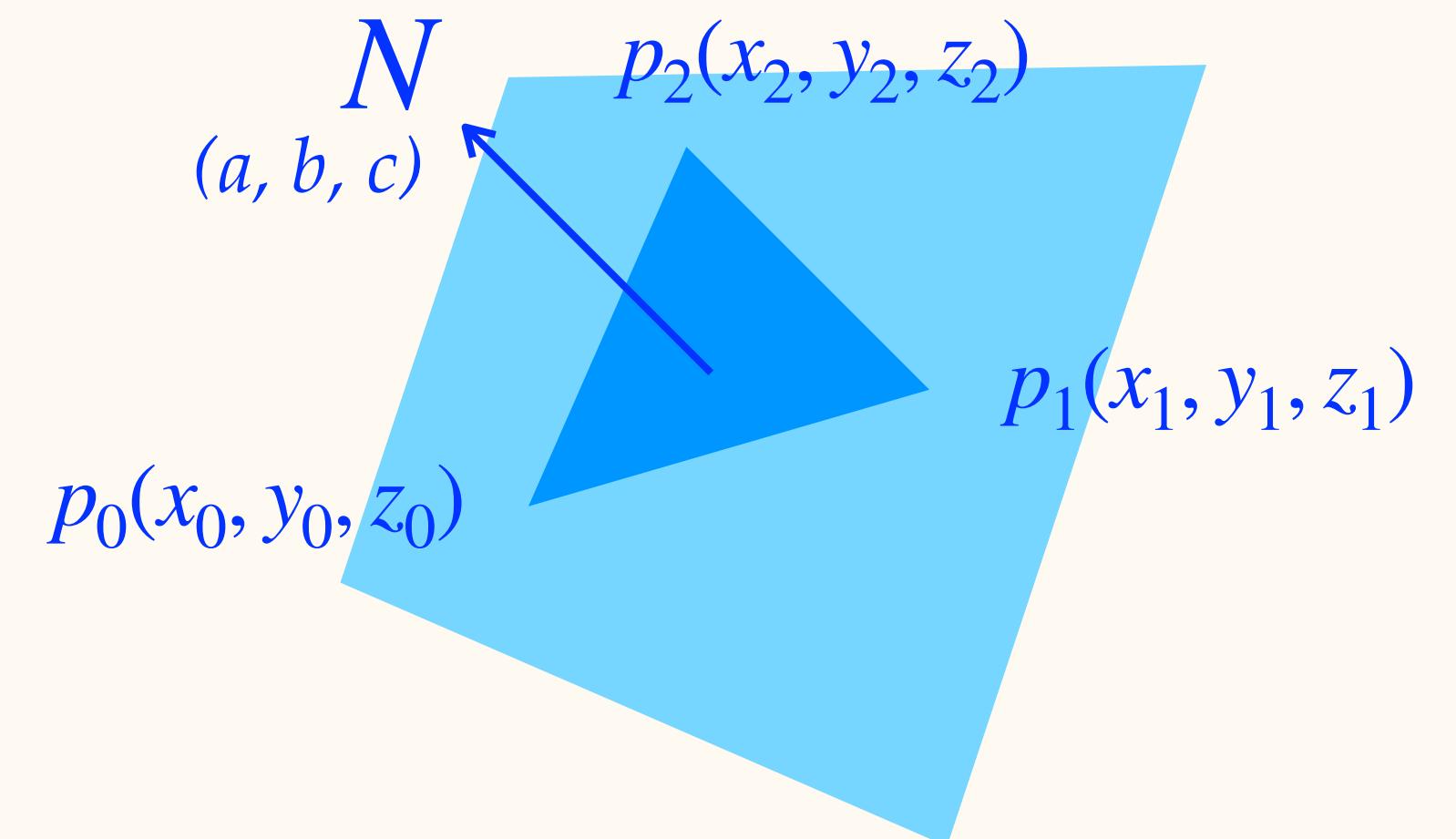
$$v_2 = p_2 - p_0$$

$$N = p_1 \times p_2 = (a, b, c)$$

$$d = -(ax + by + cz)$$

$$= -(ax_0 + by_0 + cz_0) = -N \cdot p_0$$

$$N \cdot p = N \cdot p_0$$



射線方程式

- 求一條射線與一個平面的焦點，需要射線方程式
- 射線參數方程式：(Parametric Equation)

$$x = x_0 + (x_1 - x_0)t \quad t = 0, \infty$$

$$y = y_0 + (y_1 - y_0)t$$

$$z = z_0 + (z_1 - z_0)t$$



射線方程式

- 當 $t = 0$, 交點在起點上
- 當 $t \geq 0$, 才有可能是所需要的答案
- 如果有多個交點解出, 最小的正 t 是正確的交點

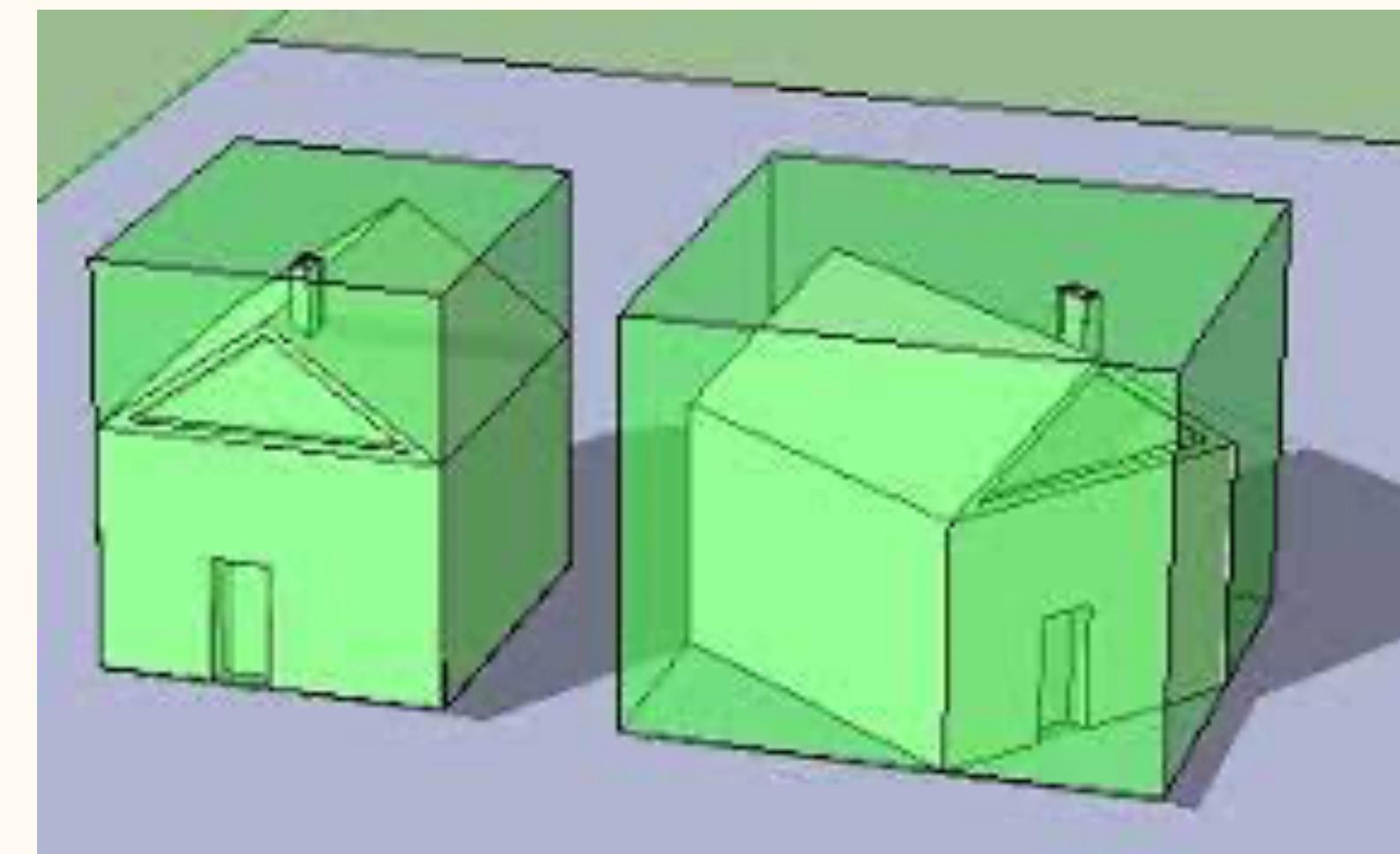
射線與平面解交點

- 將射線方程式帶入平面方程式
- 解 t ，求焦點
- 使用 Barycentric Test 確認交點是否在三角形內

碰撞偵測 Collision Detection

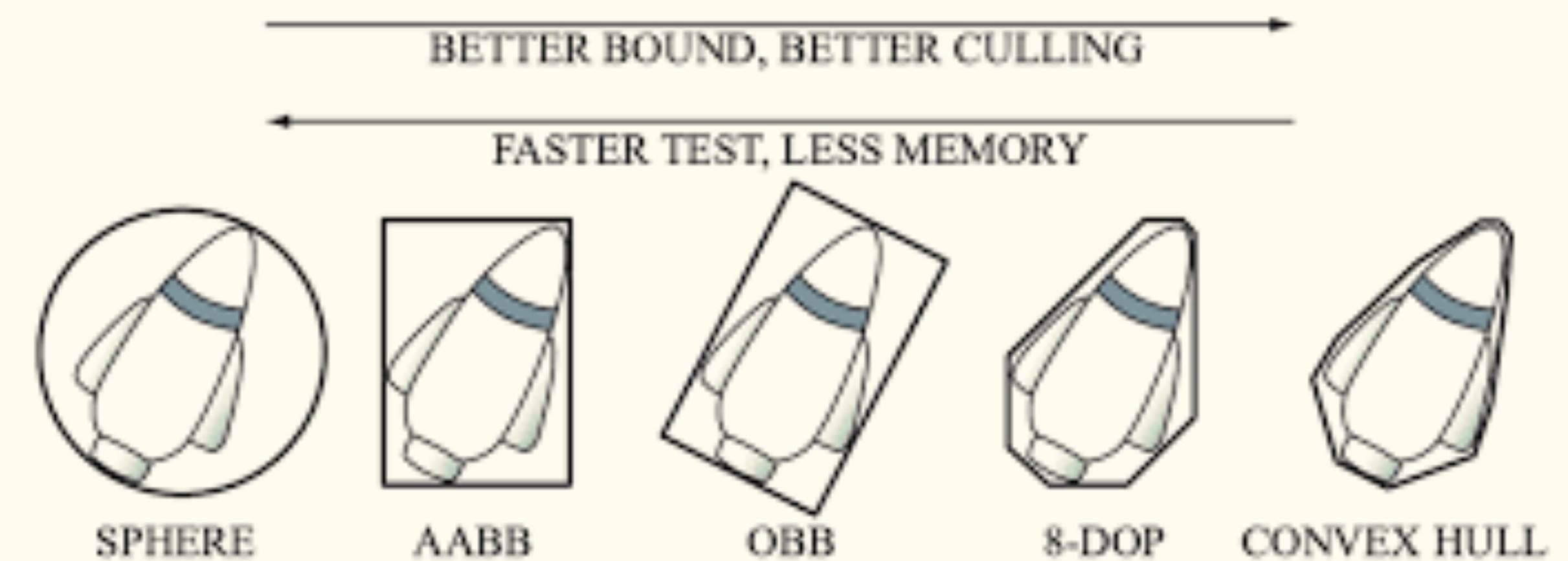
包覆體 Bounding Volume

- 一個3D物件，包覆整個模型：
 - 凸多面體
 - 簡單的數學定義或極低面數的3D物件
 - 模型的替身

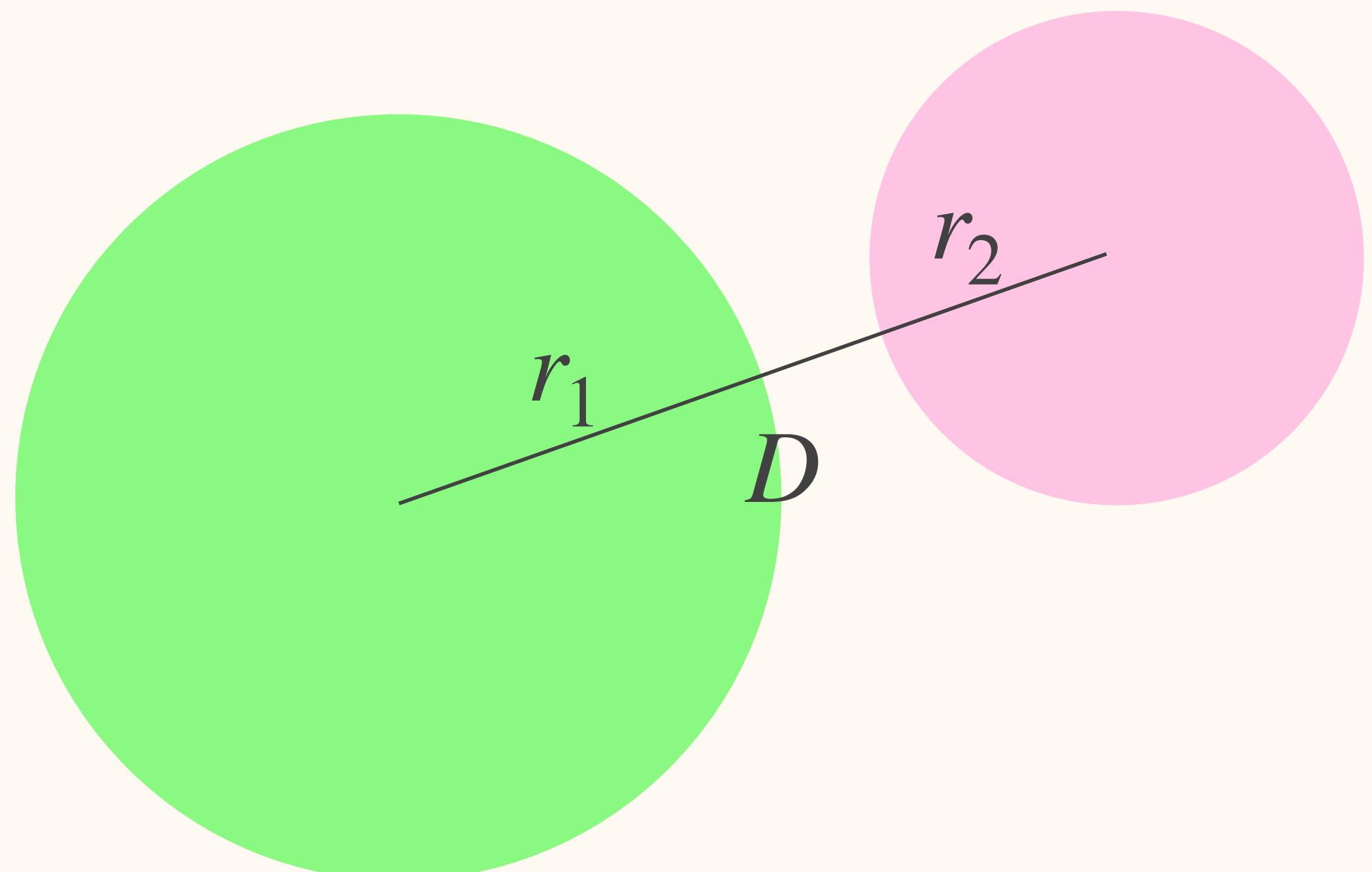


包覆體 Bounding Volume

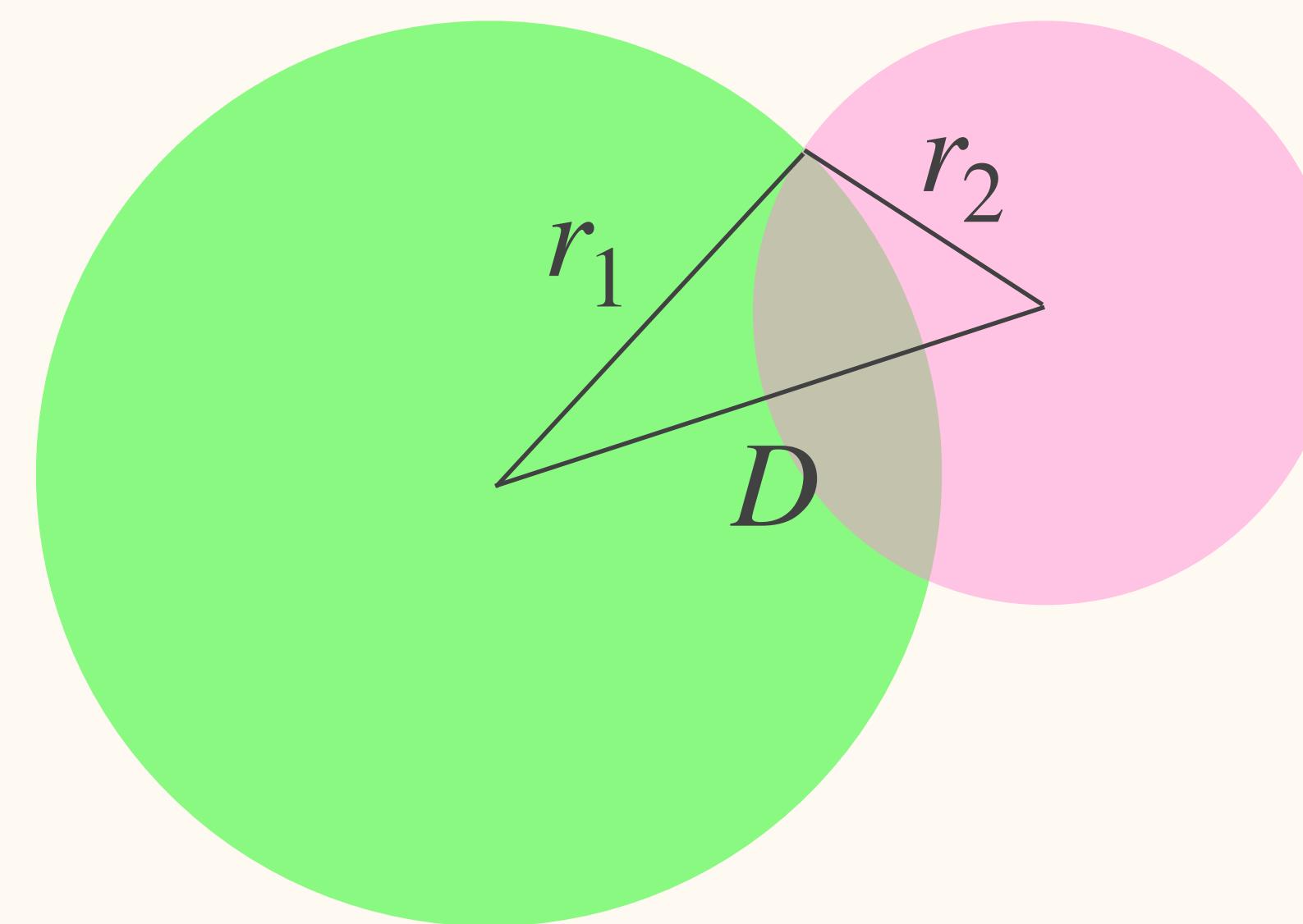
- 常見的包覆體
 - 包覆圓 (Bounding Sphere)
 - 包覆柱 (Bounding Cylinder)
 - 包覆盒 (Bounding Box)
 - k-DOP (k-discrete oriented polytope), k面的定向多面體



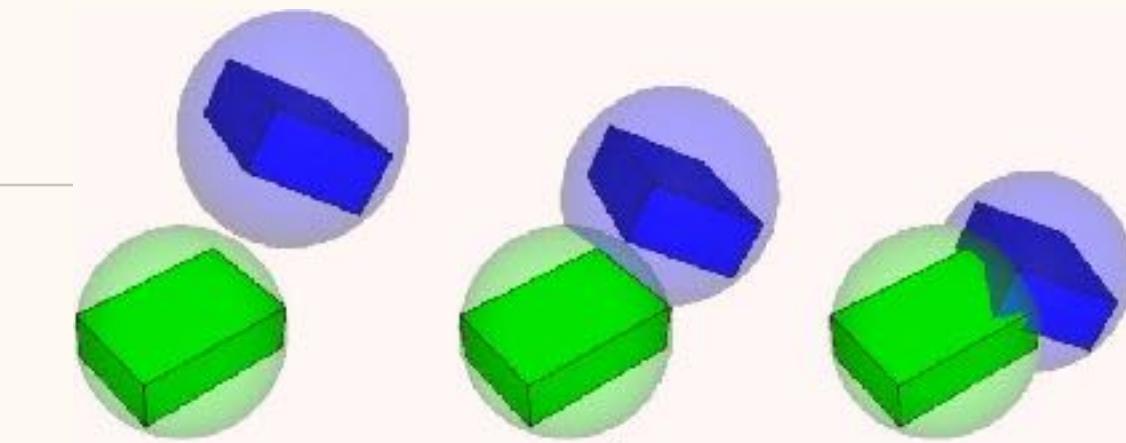
包覆圓 Bounding Sphere



$$r_1 + r_2 < D$$

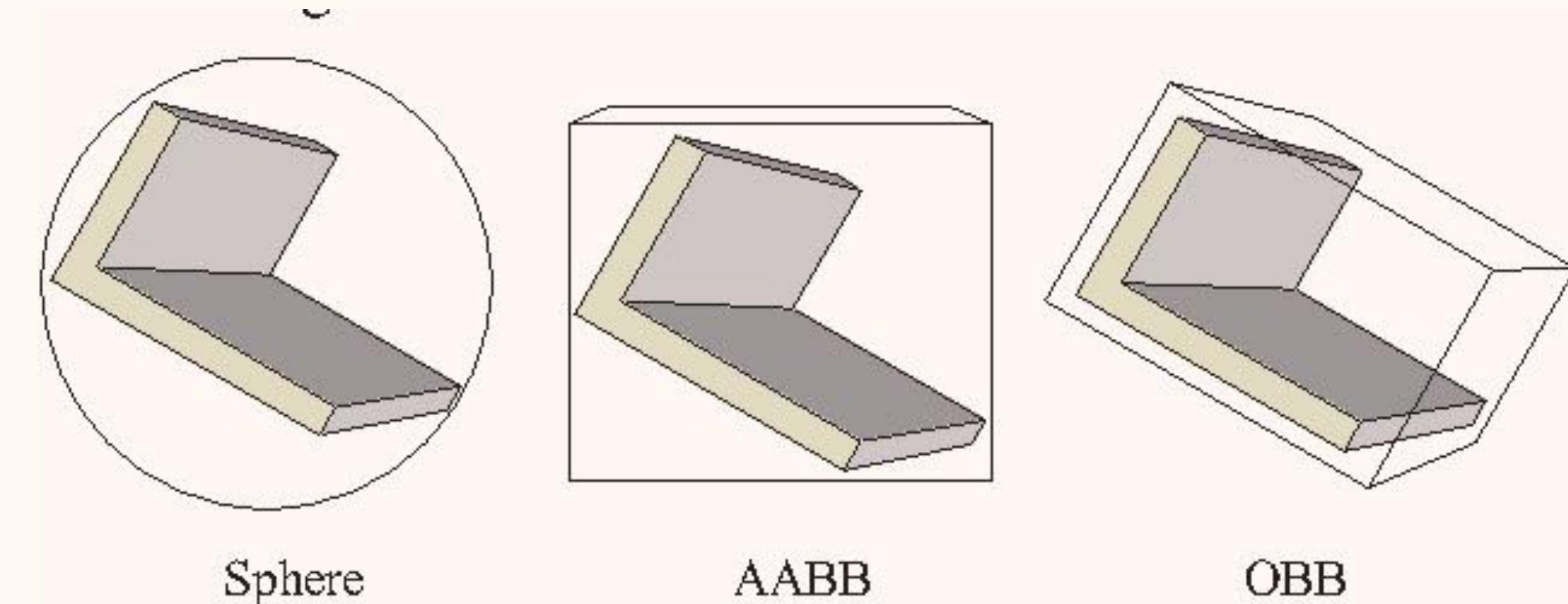


$$r_1 + r_2 > D$$



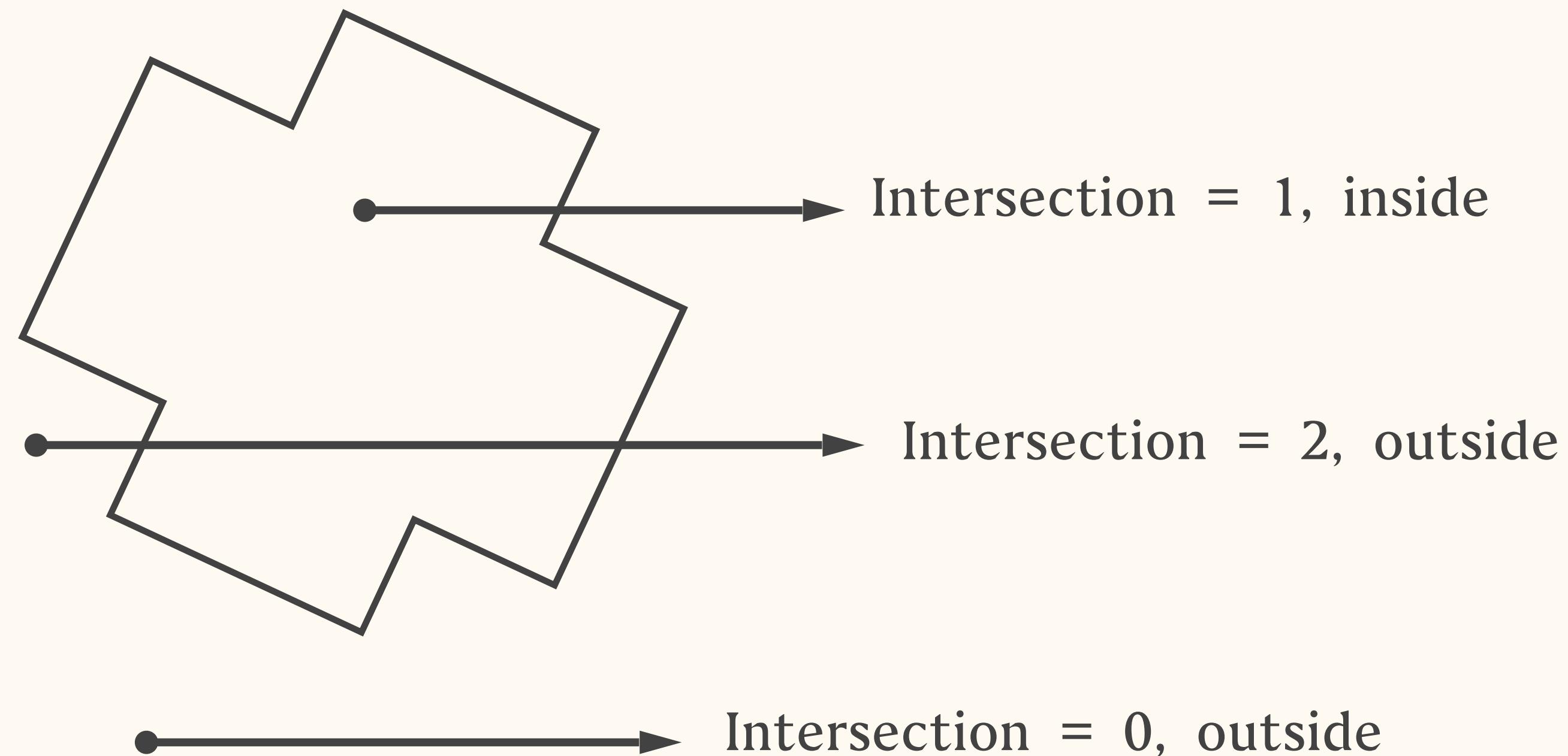
包裹盒 Bounding Box

- Axis-Aligned Bounding Box
 - AABB
- Object-oriented Bounding Box
 - OBB



容器內測試 Containment Test

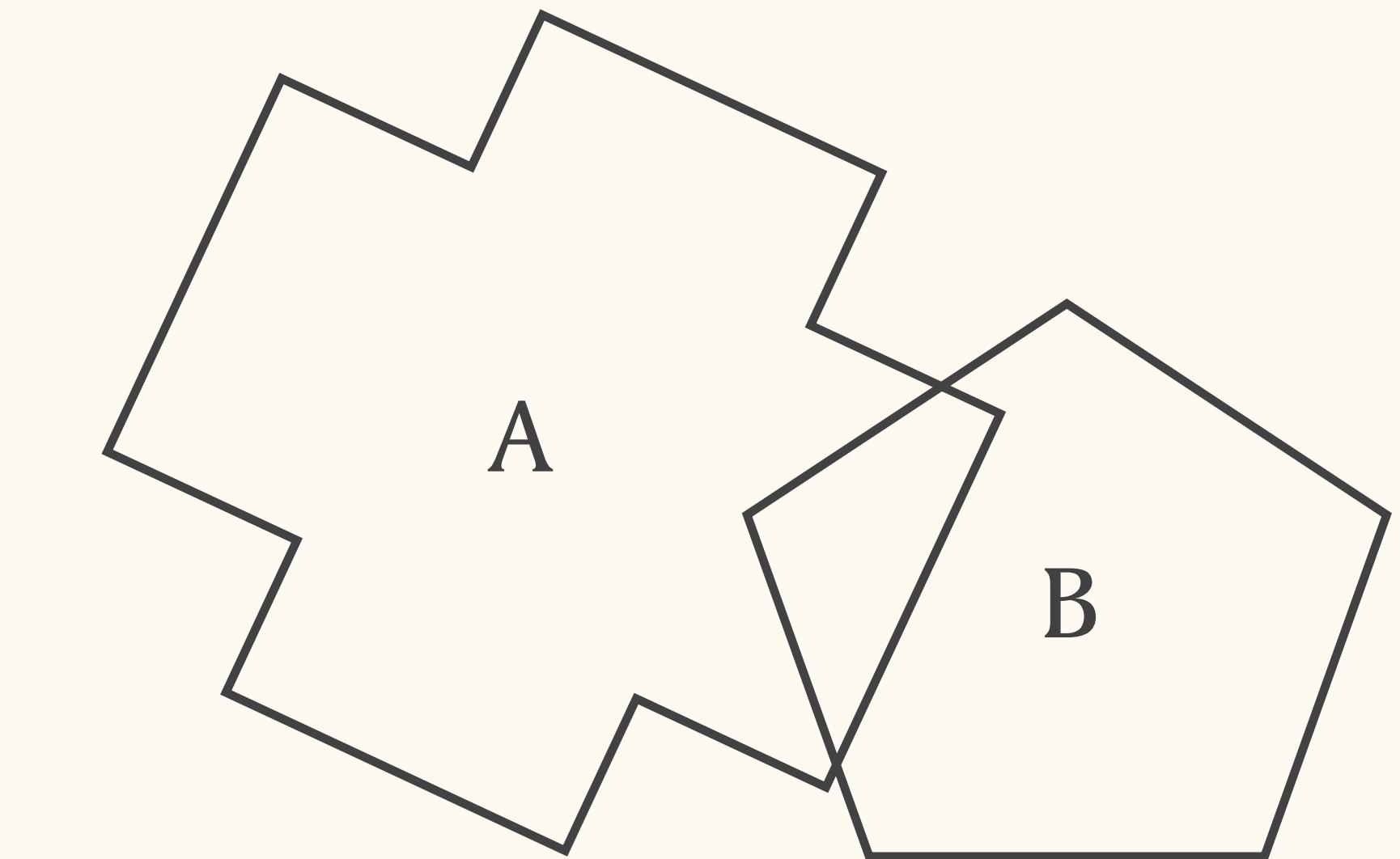
- 測試位置朝任意方向射出一條射線，求該射線與所有邊界線段/平面的交點



“如果交點數量是奇數，該點在容器內；數量是偶數，該點在容器外”

碰撞偵測使用 Containment Test

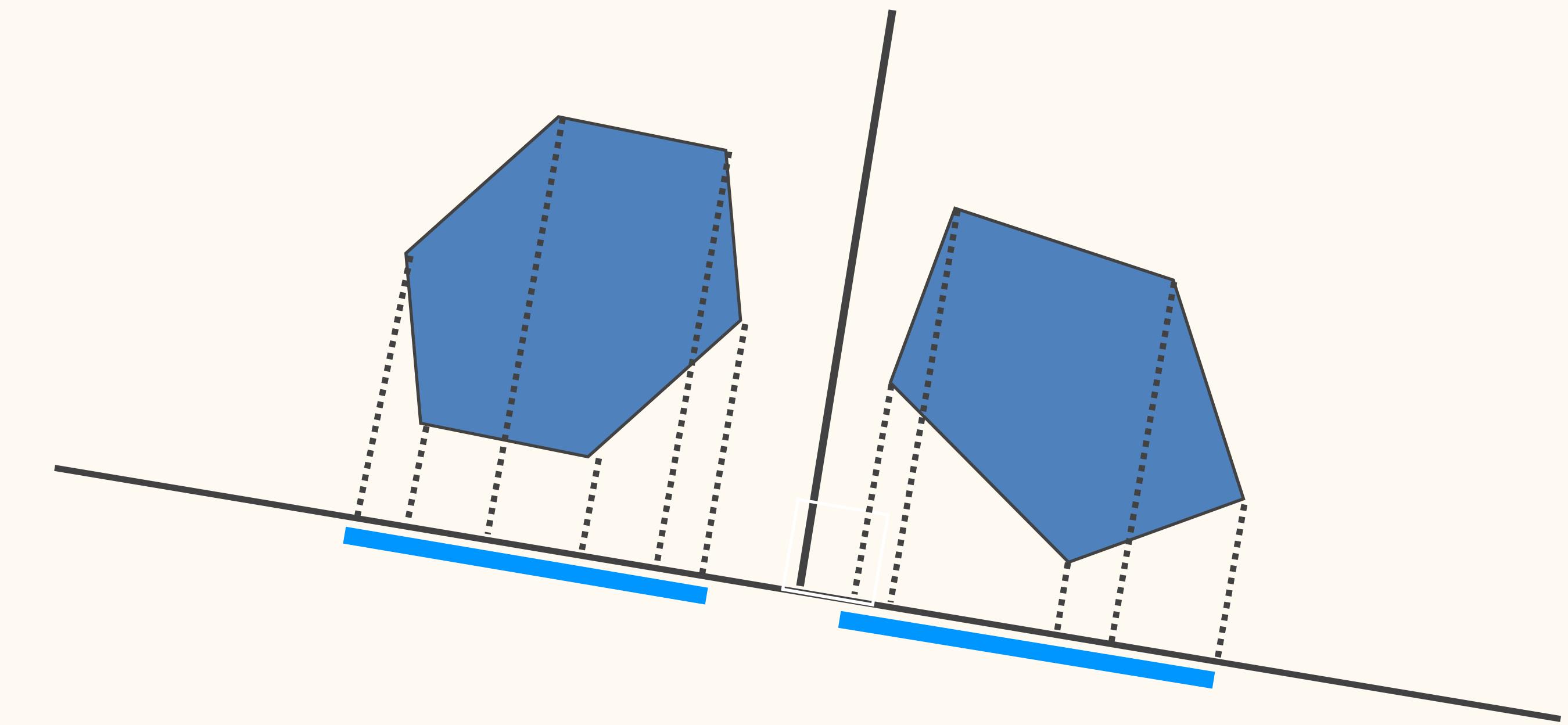
- 先由 A 的所有頂點對 B 進行 containment test
- 再由 B 的所有頂點對 A 進行 containment test
- 有任一頂點通過 containment test，兩物件碰撞成立



- Containment test 適用於任何多邊形或多面體

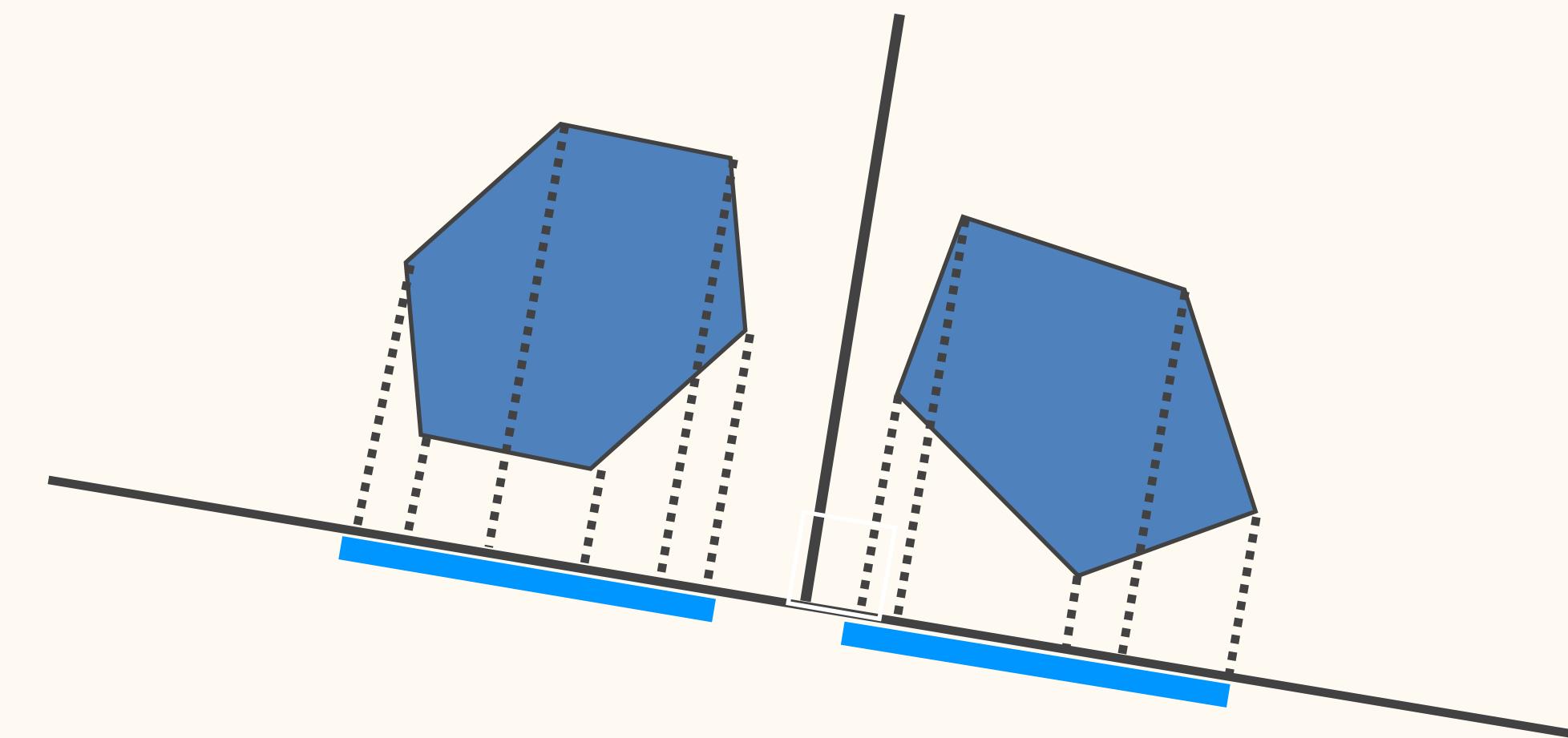
分離軸定理 Separating Axis

- 一種快速的碰撞偵測的方法，只適用於凸多邊形(2D)或凸多面體(3D)
- 找一條線(2D)或是一個平面(3D)將兩物件分離
- 關鍵：
 - 如何判斷分離？
 - 如何找到這條線或平面？



分離軸定理 Separating Axis

- 方法：
 - 找到一個垂直於測試平面的平面
 - 將所有物件的頂點投影在該垂直面上
 - 將各物件的投影範圍求出
 - 如果投影範圍沒有重疊，該測試平面為 Separating Axis



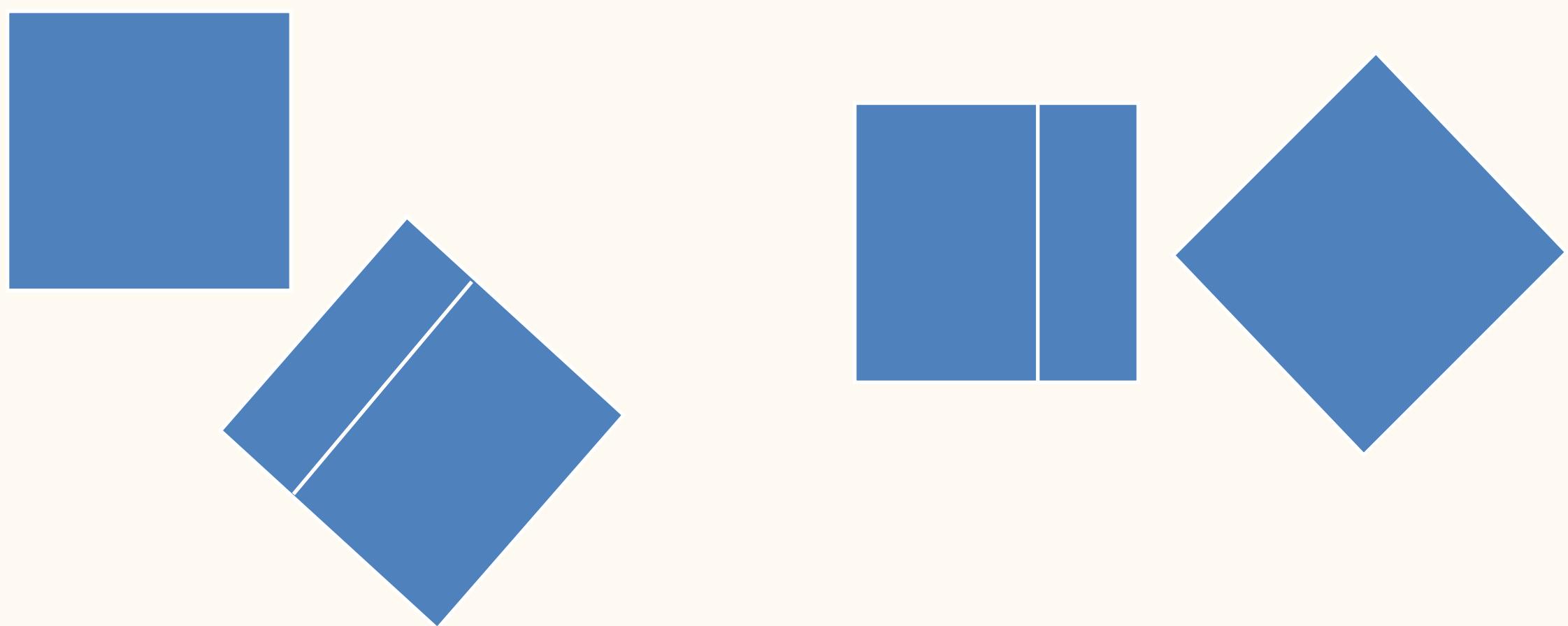
Separating Axis Algorithm

```
Bool TestIntersect(ConvexPolyhedron C0, ConvexPolyhedron C1)
{
    // test faces of C0 for separation
    for (i = 0; i < C0.GetFaceCount(); i++) {
        D = C0.GetNormal(i);
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1) return false;
    }

    // test faces of C1 for separation
    for (i = 0; i < C1.GetFaceCount(); i++) {
        D = C1.GetNormal(i);
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1) return false;
    }
}
```

Separating Axis Algorithm

```
// test cross products of pairs of edges
for (i = 0; i < C0.GetEdgeCount(); i++) {
    for (j = 0; j < C1.GetEdgeCount(); j++) {
        D = Cross(C0.GetEdge(i), C1.GetEdge(j));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1) return false;
    }
}
return true;
```



Separating Axis Algorithm

```
void ComputeInterval(ConvexPolyhedron C, Vector D,
                     double &min, double &max)
{
    min = Dot(D, C.GetVertex(0));
    max = min;
    for (i = 1; i < C.GetVertexCount(); i++) {
        value = Dot(D, C.GetVertex(i));
        if (value < min) min = value;
        else if (value > max) max = value;
    }
}
```

Geometric Representation

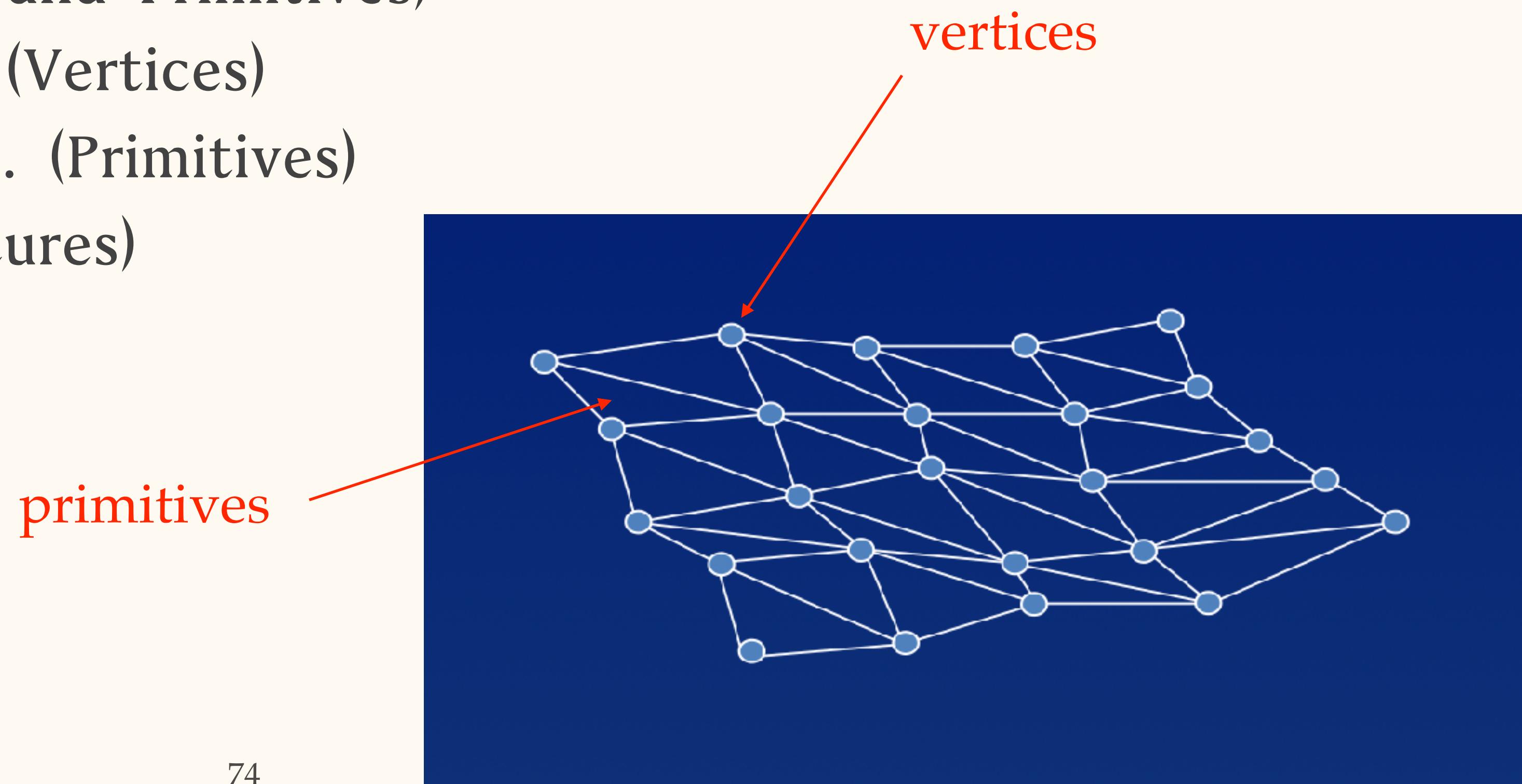
幾何的表示法

Representation of Objects

- 決定圖形物件數據表示方法的依據
 - 資料結構 Data structure
 - 處理/渲染 3D 圖形物件所需的成本
 - Cost of processing an object (from the view of 3D)
 - 3D 圖形物件的外觀 (Final appearance of an object)
 - 編輯 3D 圖形物件的容易度 (Ease of editing the shape of the object)
 - 常見的方法：
 - 多邊形 Polygonal representation
 - 遊戲主要的幾何表示法
 - 曲面 (Surfaces)
 - Constructive solid geometry (CSG)
 - Subdivision

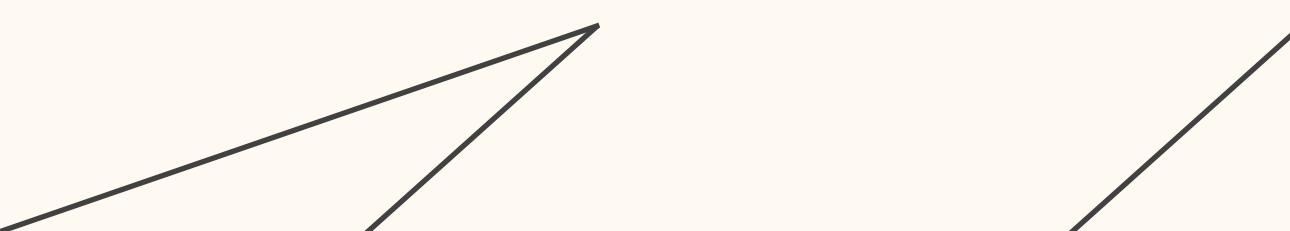
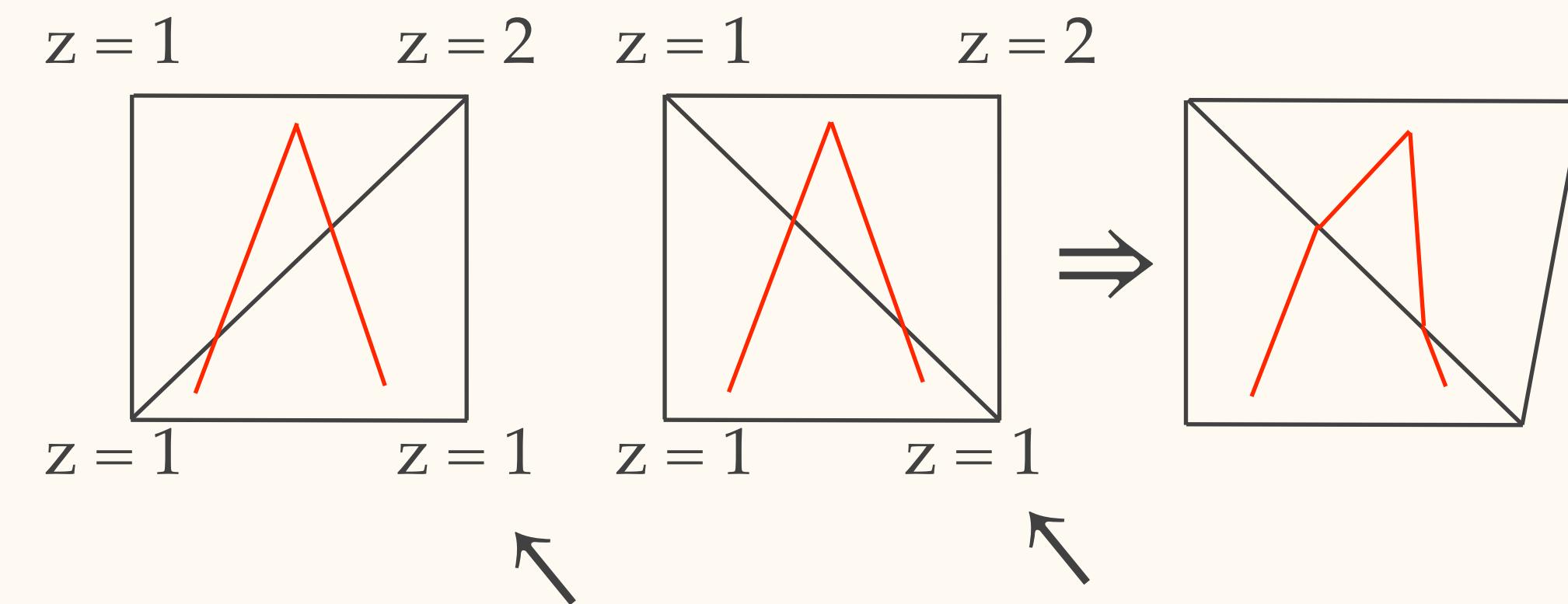
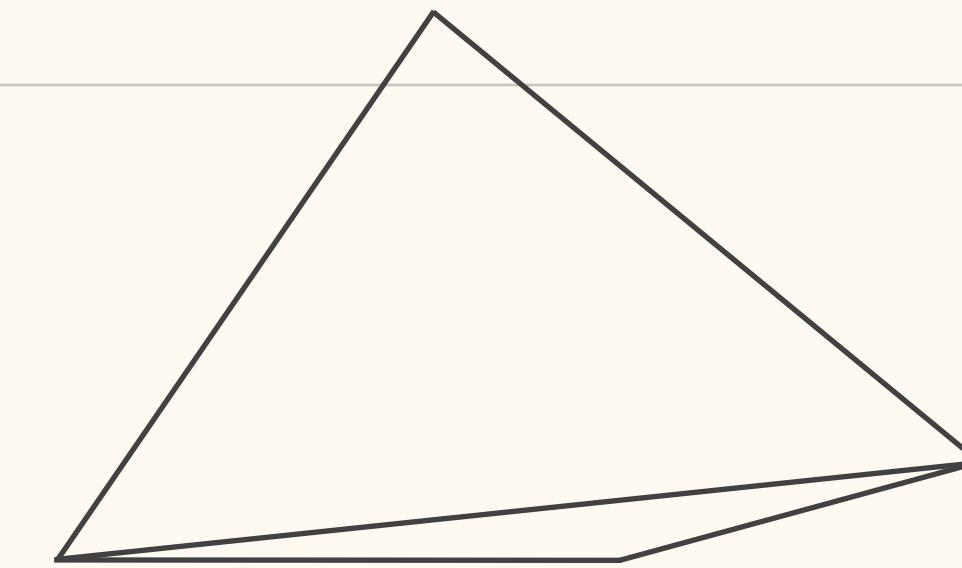
多邊形的模型

- 必須是凸多邊形 (Convex polygons)
- 遊戲渲染硬體使用三角形 (Triangles)
 - 新一代的硬體開始支援四邊形 (Quads)
- 幾何資料與幾何元件 (Geometry and Primitives)
 - 幾何資料 : (x, y, z) 頂點座標 (Vertices)
 - 幾何元件 : 三角形、四邊形、... (Primitives)
- 材質與貼圖 (Materials and Textures)
- 三角形網格 (Triangular mesh)
- 頂點次序很重要
 - 順時針
 - 決定正面或反面
 - 法向量



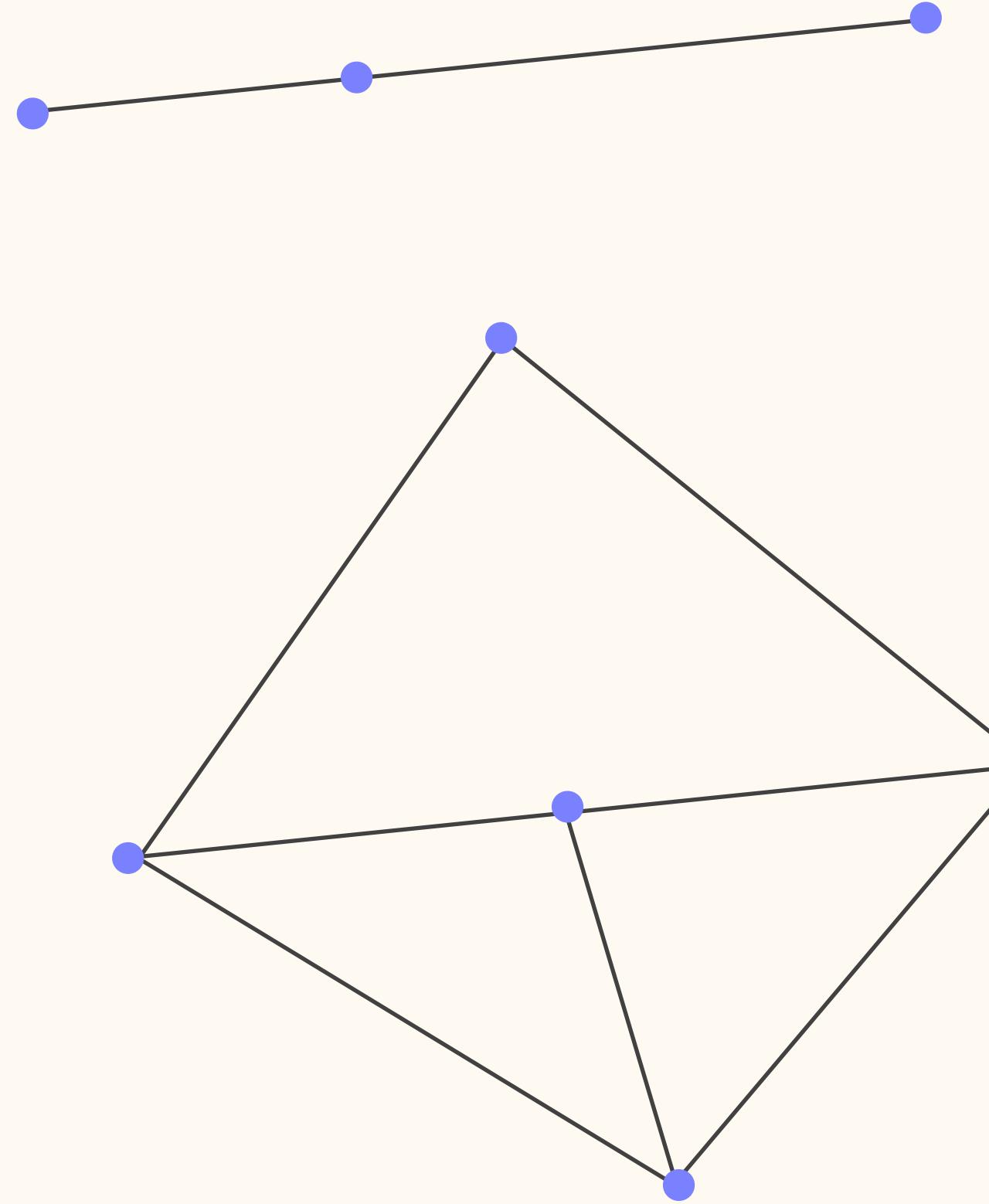
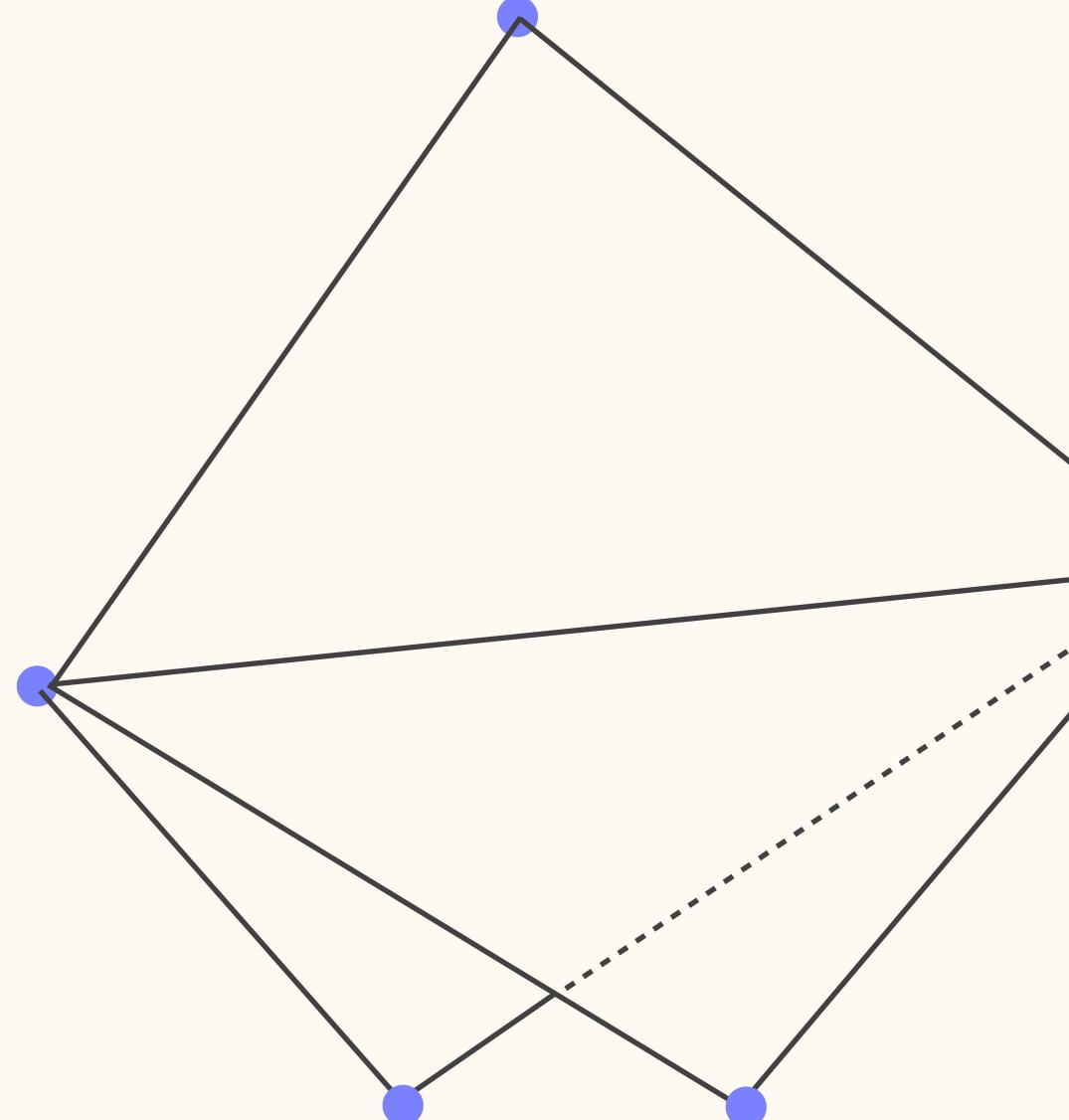
三角形 Triangles

- 遊戲渲染硬體使用三角形 (Triangles)
- 三角形的優點
 - 三點共面
 - 幾何元件單一化，沒有建模的死角
 - 資料結構單純，適合硬體加速
- 三角形的缺點
 - 三角形內差 vs 四邊形內差
 - 細長的三角形渲染效果不佳



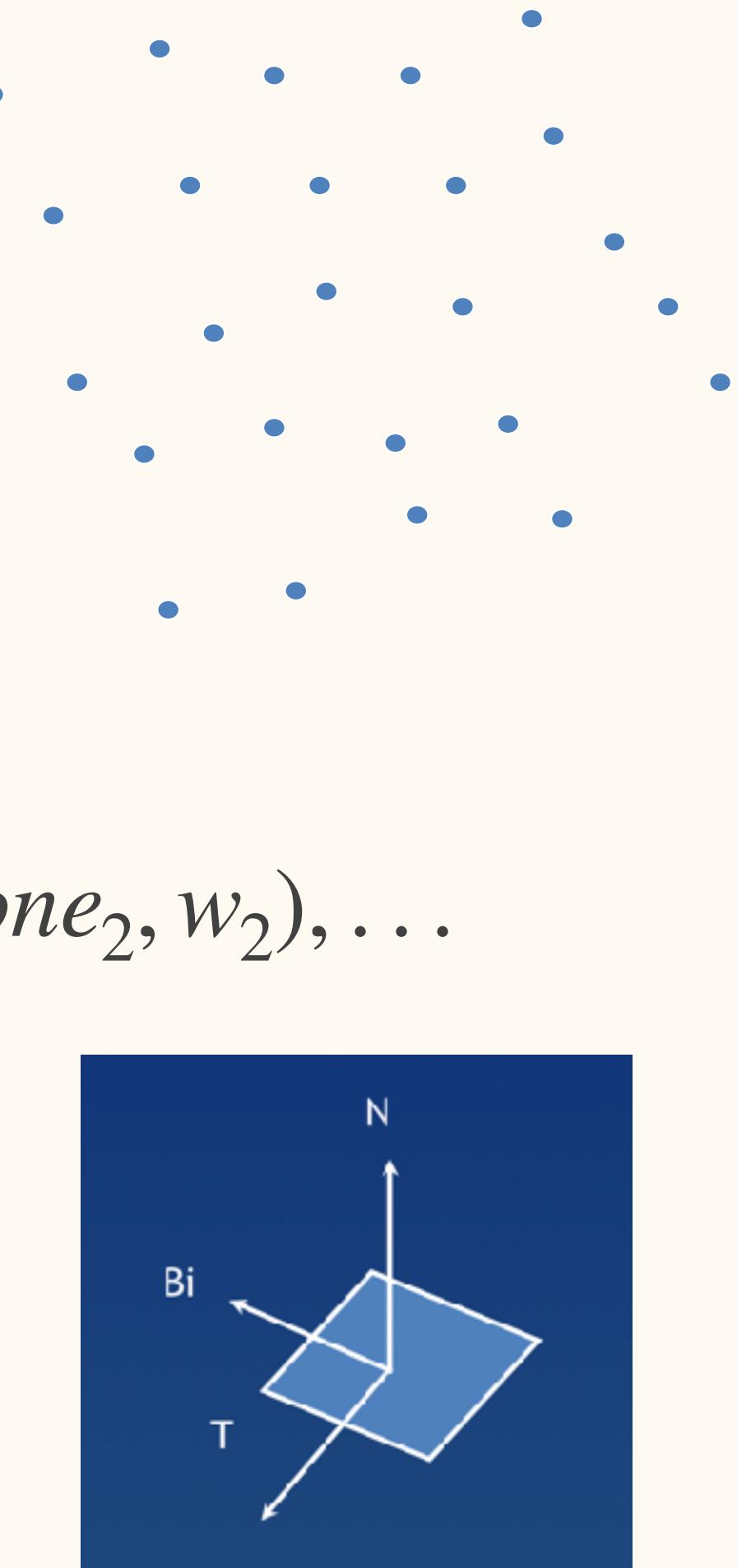
非法三角網格 Illegal Triangular Mesh

- 三點共線
 - 面積 = 0
- T-joint 問題
- 一個邊 (Edge) 被超過兩個三角形共用

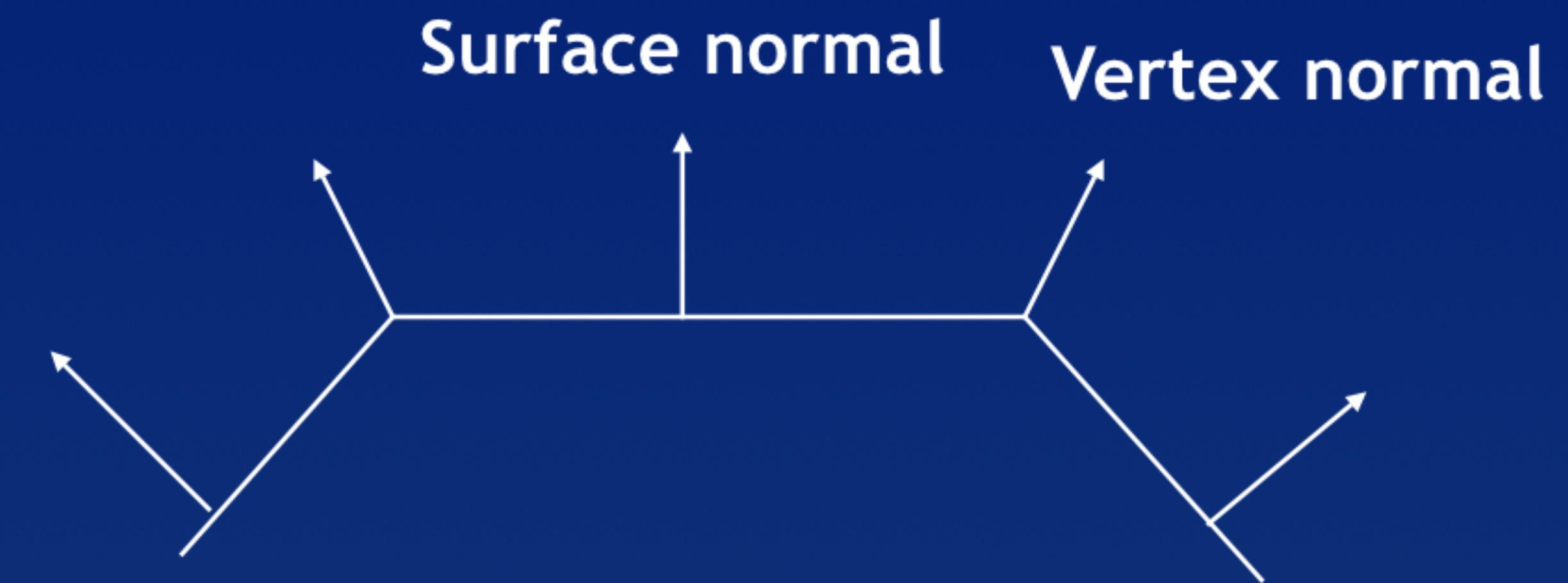
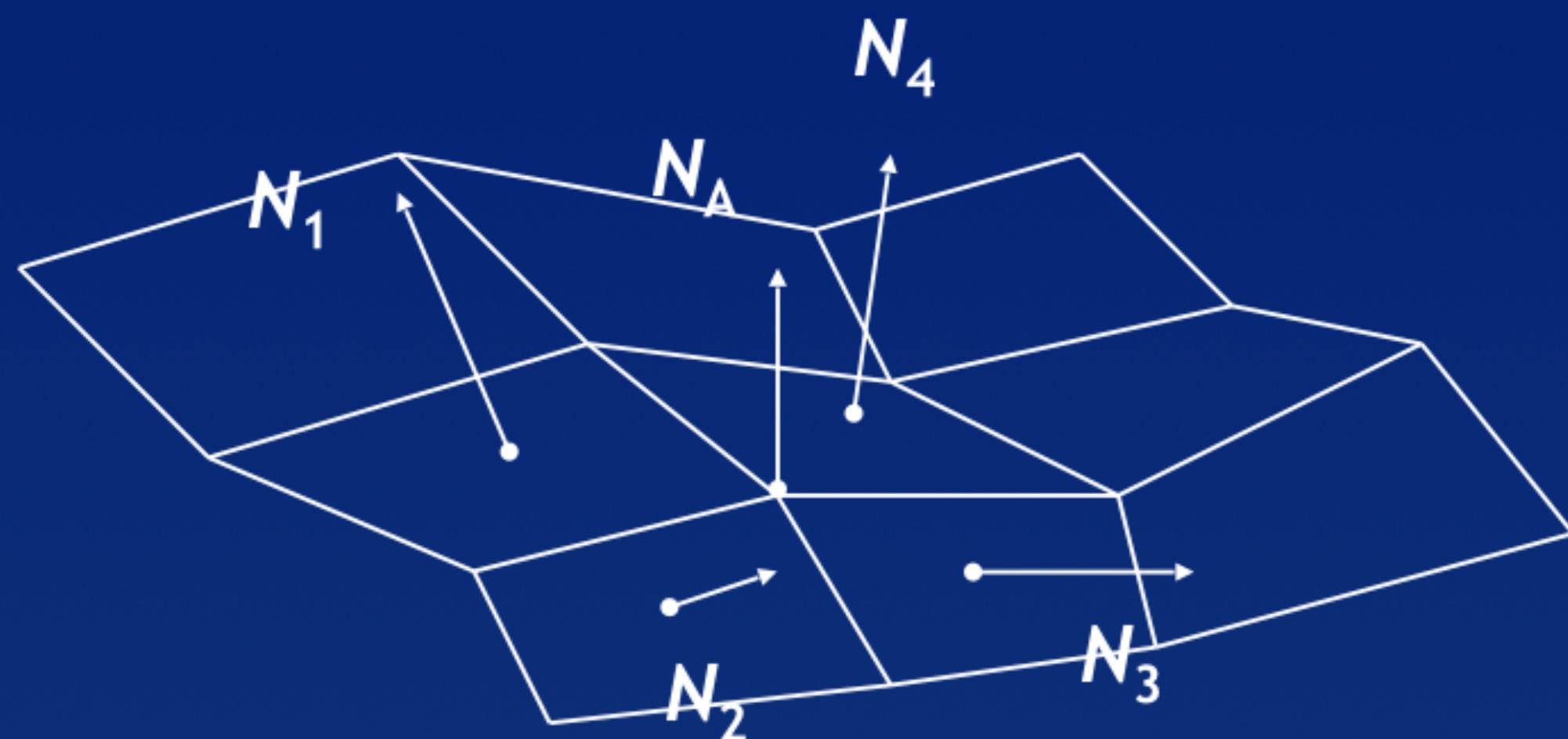


三角網絡 Triangular Mesh - Geometry

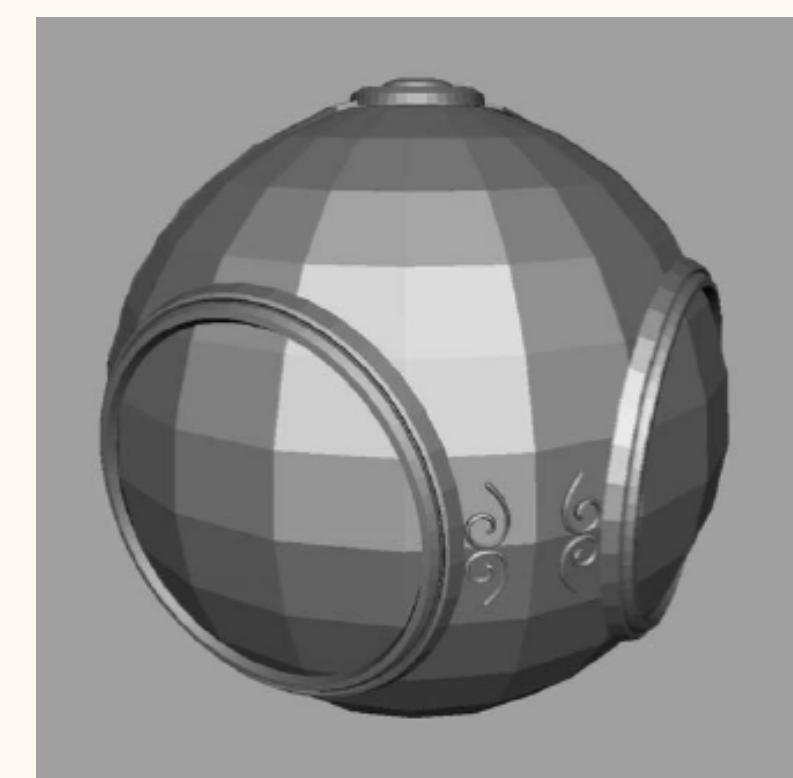
- Vertices
 - 位置 - (x, y, z)
 - 頂點法向量 (Vertex normal) - (n_x, n_y, n_z)
 - 頂點顏色 (Vertex color) - (r, g, b)
 - 貼圖座標 (Texture coordinates) - $(u_1, v_1), (u_2, v_2), \dots$
 - 切線向量 (Tangent vector) - (t_x, t_y, t_z)
 - 次法向量 (Bi-normal) - (b_x, b_y, b_z)
 - 皮膚變形參數 (Skin deformation parameters) - $(bone_1, w_1), (bone_2, w_2), \dots$
 - 其他
- Local 座標系
- Model 座標系



Vertex Normal



- $N_A = \text{Average}(N_1, N_2, N_3, N_4)$
 - $\text{Average}()$ can be weighted by
 - Internal angle scale
 - Area of polygon



幾何元件 Primitives

- 目前3D渲染硬體是以三角形為主要幾何元件
- 使用的是 indexed primitive 的作法
 - 所有的頂點資料放在一個陣列中
 - Vertex array or vertex buffer
 - 所有的三角形資料記錄在一個陣列中，記錄的是頂點的編號 (Indexing)
 - Index array or index buffer

vertex array

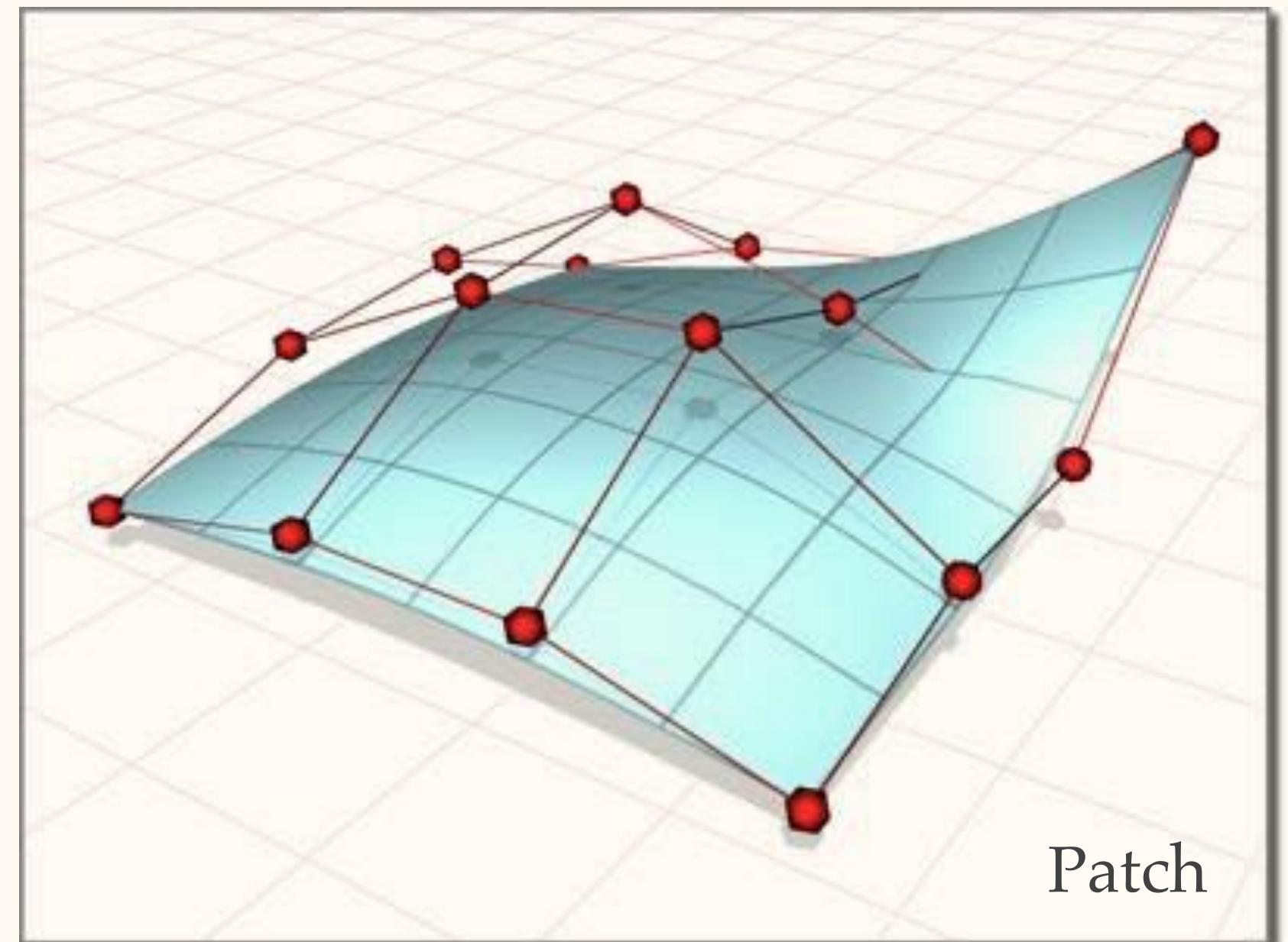
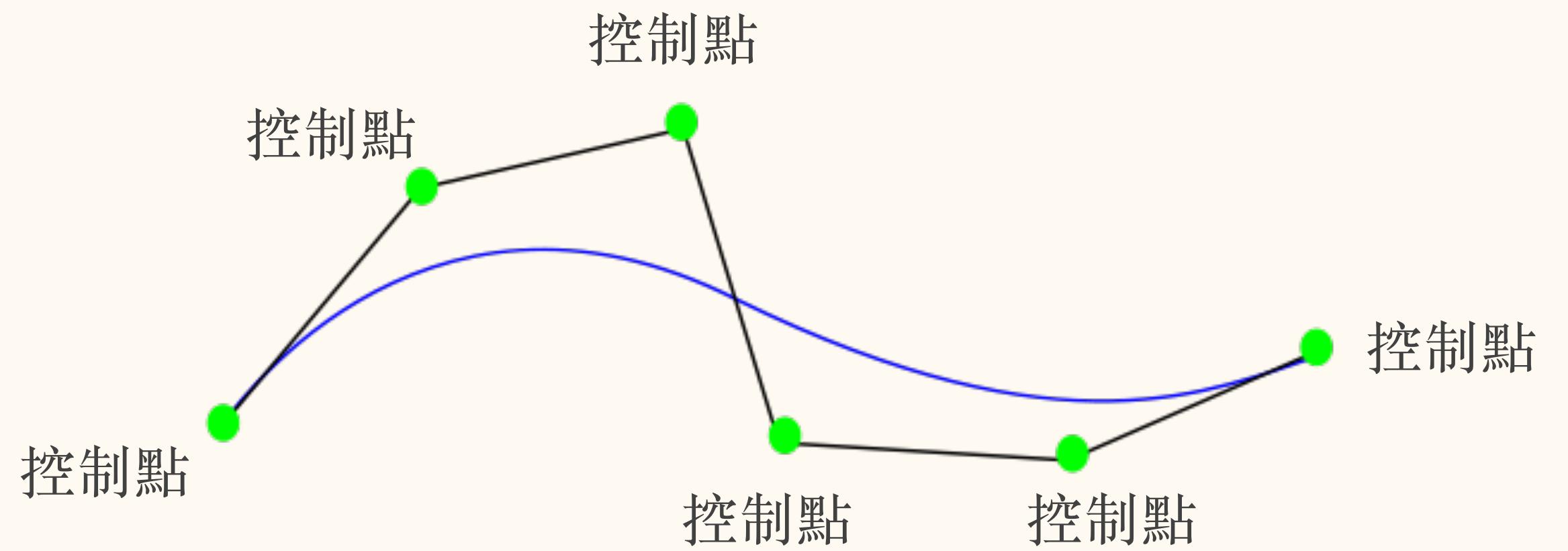
0	[$x, y, z, n_x, n_y, n_z, u, v, \dots$]
1	[$x, y, z, n_x, n_y, n_z, u, v, \dots$]
2	[$x, y, z, n_x, n_y, n_z, u, v, \dots$]
3	...

index array

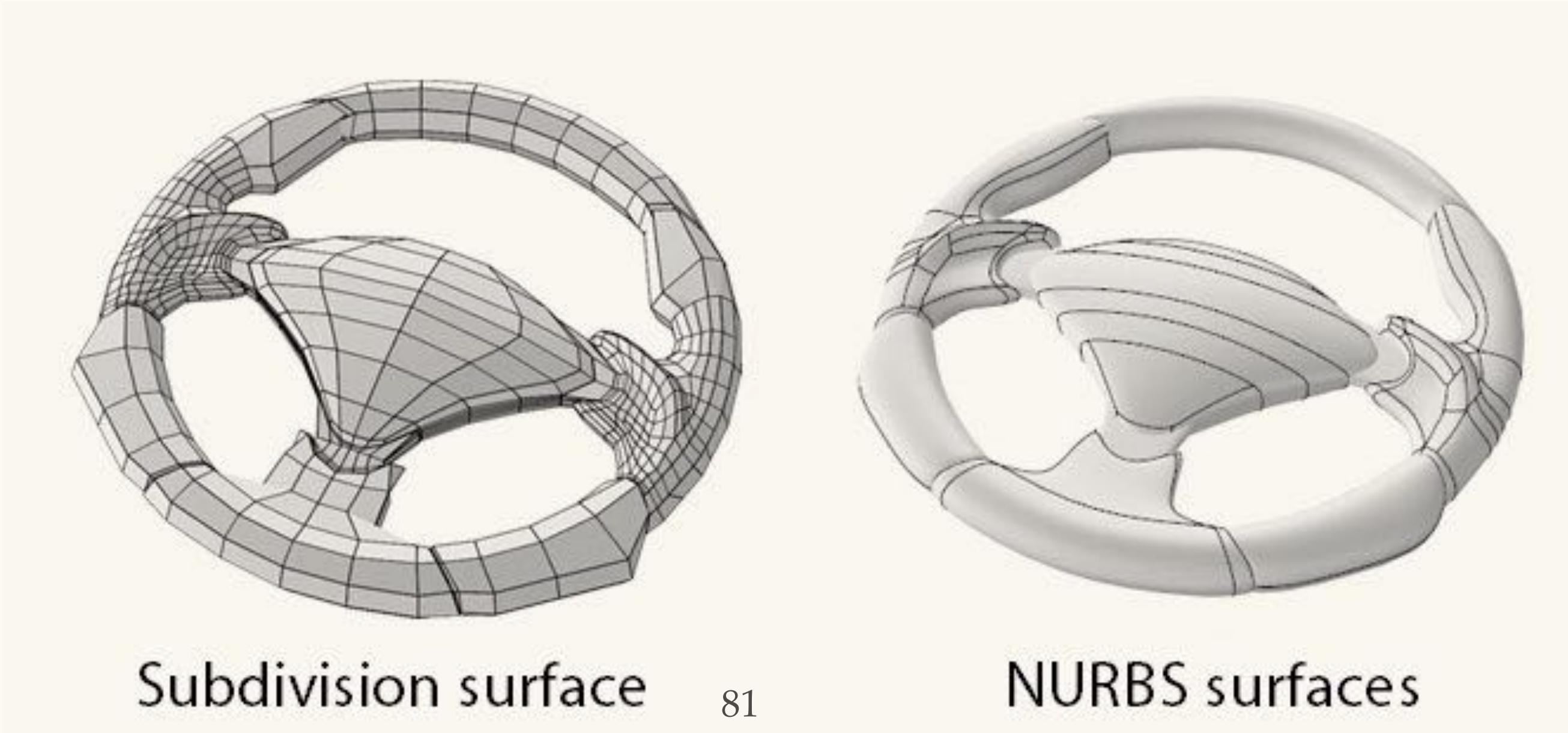
0	0,4,7
1	3,2,9
2	4,12,3
3	...

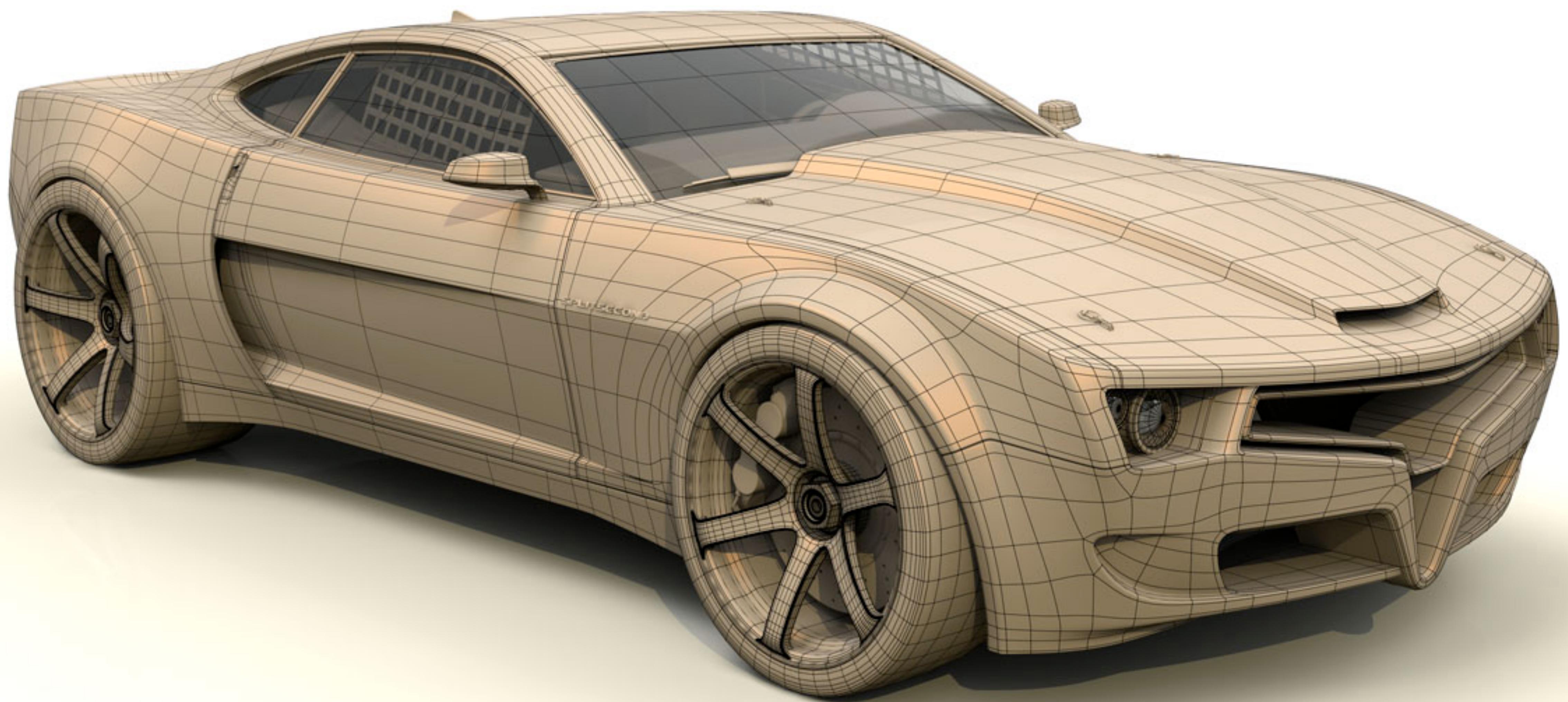
參數化曲面 Parametric Surfaces

- 例子
 - Splines
 - Bezier surfaces (貝式曲面)
- General solution :
 - Non-uniform Rational B-splines
 - NURBS



Subdivision Surfaces

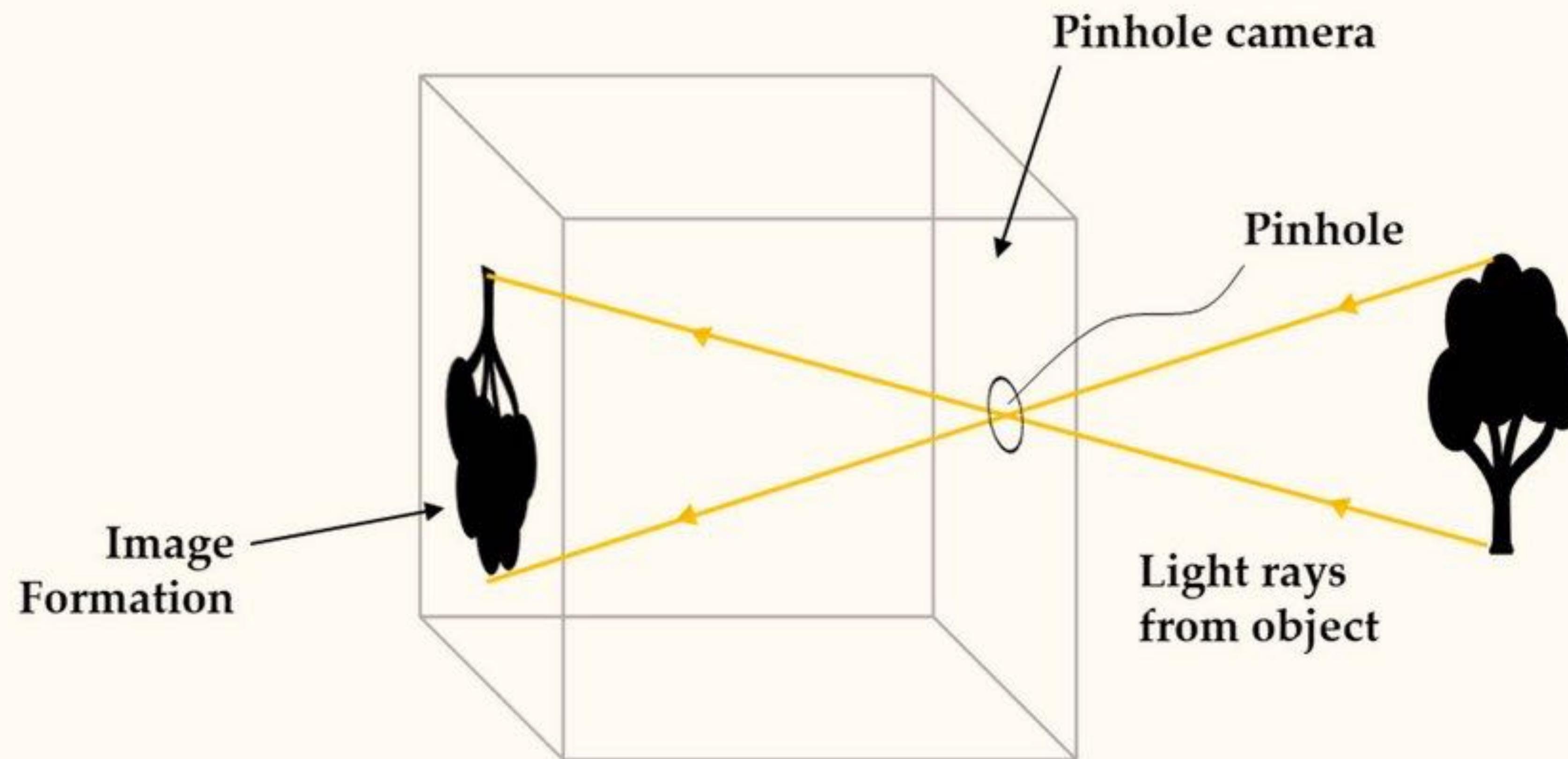




Camera Viewing & Projection

攝影機 Camera

- 電腦繪圖所使用的相機原理是基於針孔相機 (Pinhole Camera) 的設計

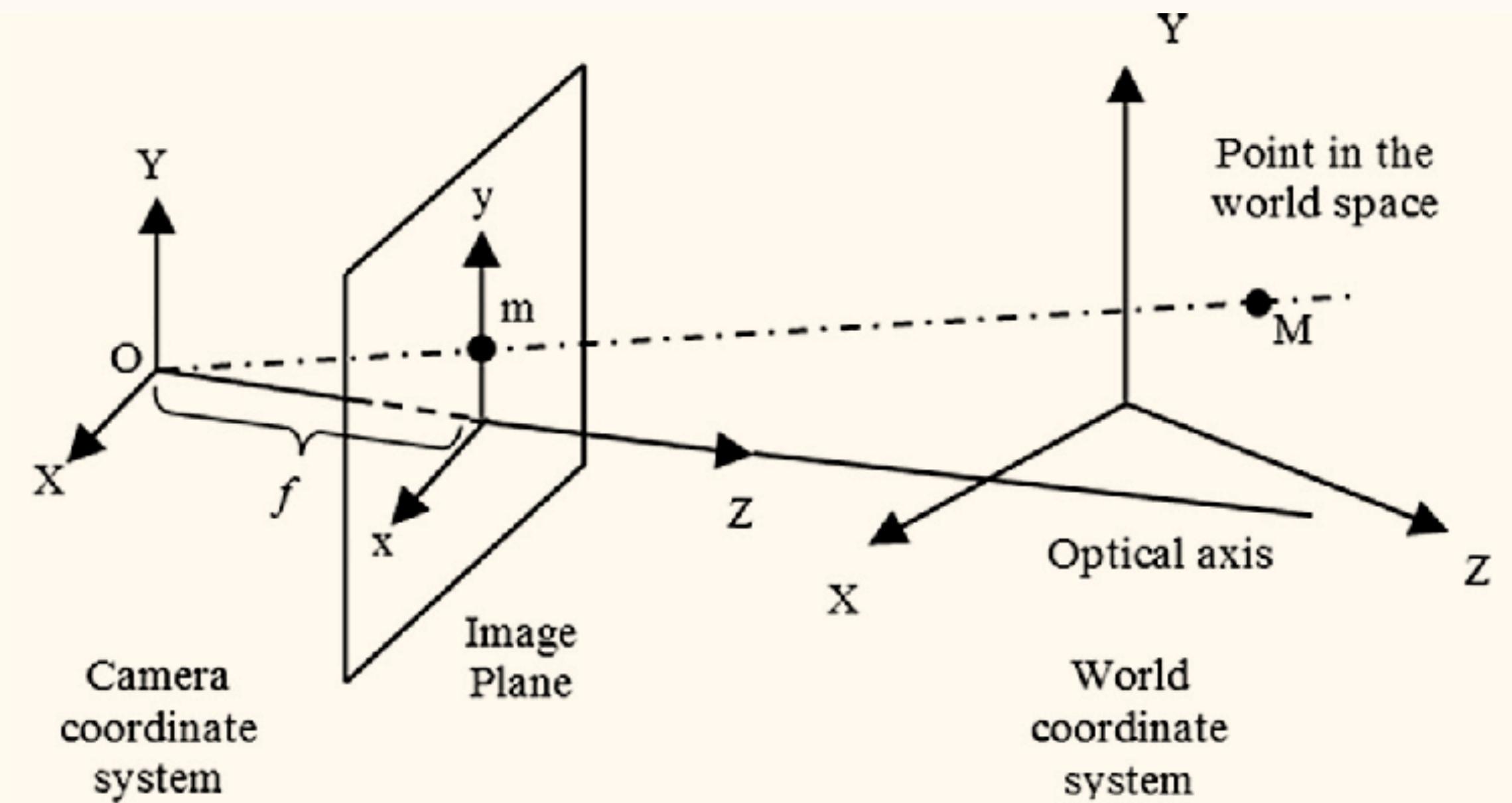


攝影機 Camera

- 相機的兩大系統：
 - Viewing 系統
 - Viewing matrix (Extrinsic matrix)
 - Projection 系統 (投影系統)
 - Projection matrix (Intrinsic matrix)

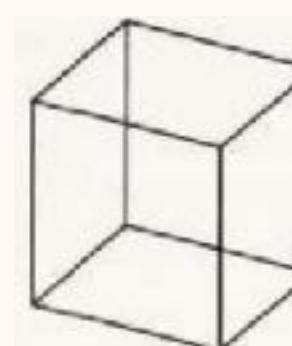
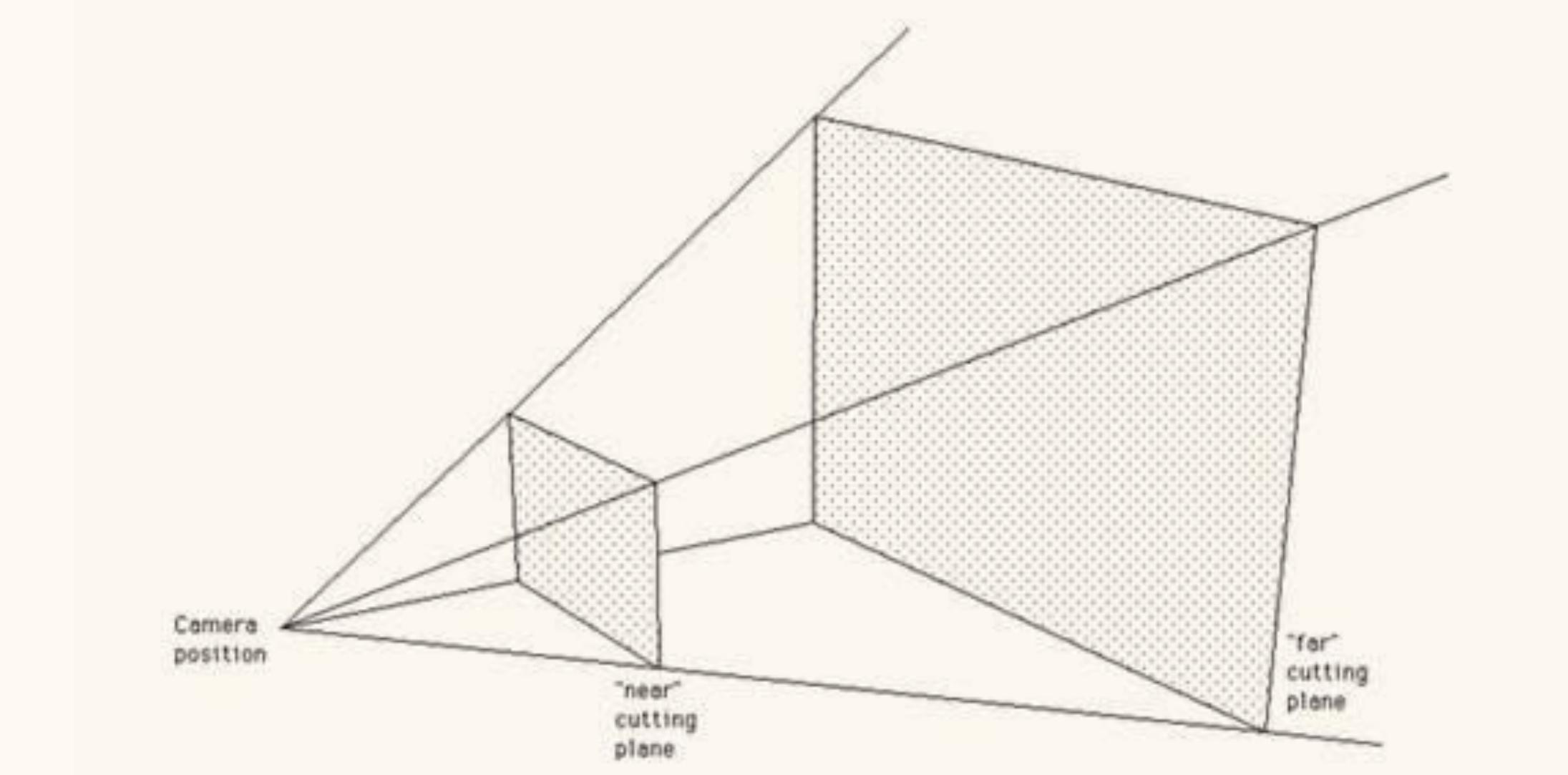
$$v_{camera} = [V]_{4 \times 4} v_{world}$$

$$v_{screen} = [P]_{4 \times 4} v_{camera}$$

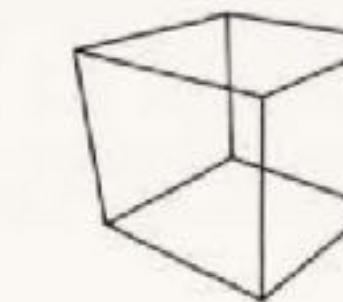


Camera Projection System

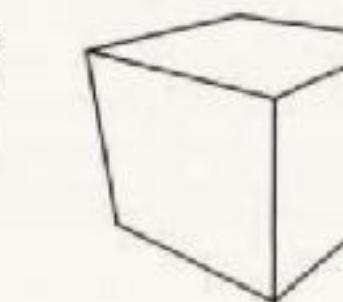
- 與相機的規格有關
 - 廣角鏡頭、望遠鏡頭、...
 - 焦長 (Focus Length)、成像大小 (Film Size)、成像比例 (Aspect Ratio)
 - 視野 (Field of View, FOV)
 - 以角度為單位
- 常用的投影方法
 - 透視投影 (Perspective Projection)
 - 透視中心位置在攝影機位置
 - 正投影 (Orthogonal Projection)
 - 透視中心位置在無窮遠



orthographic projection

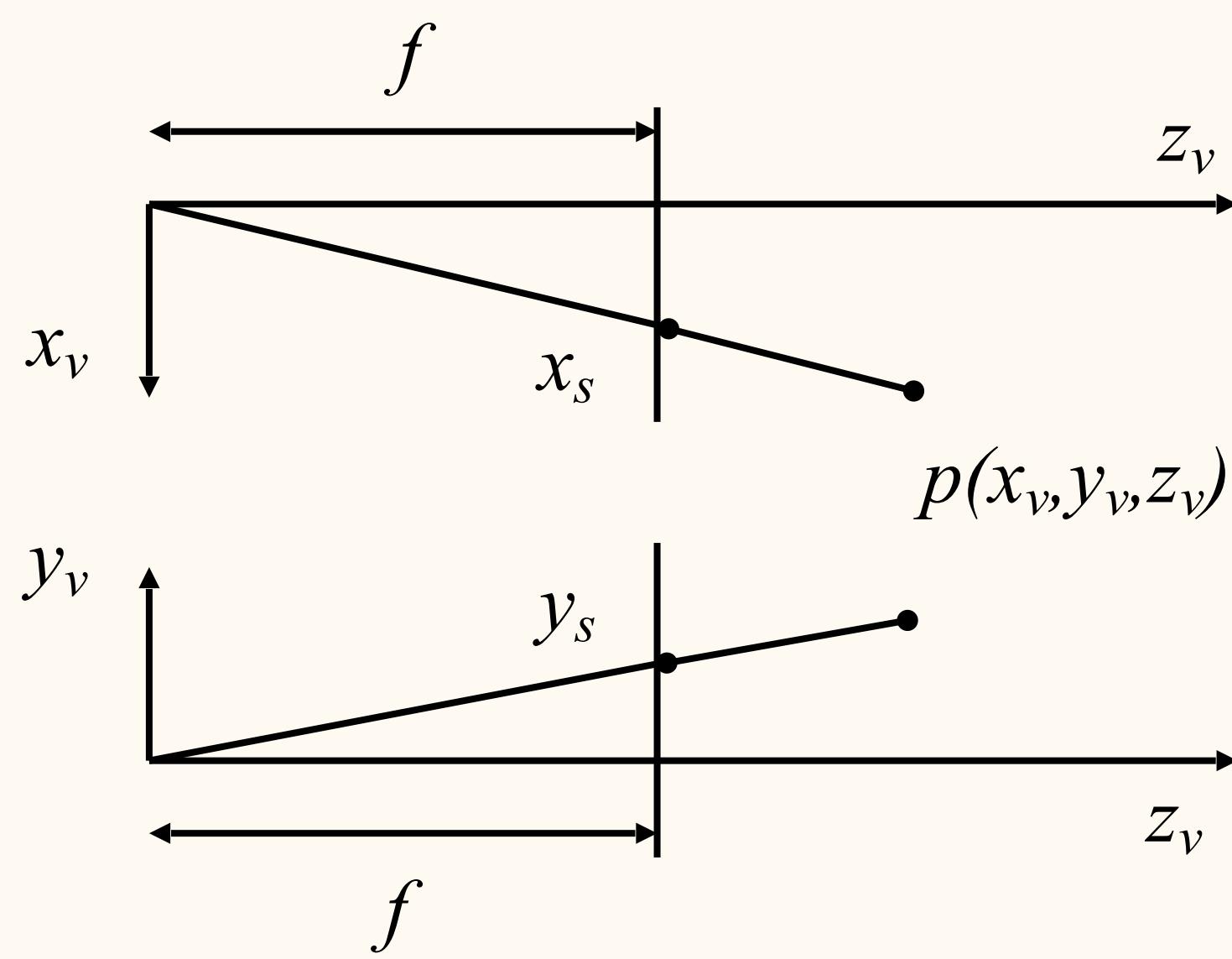
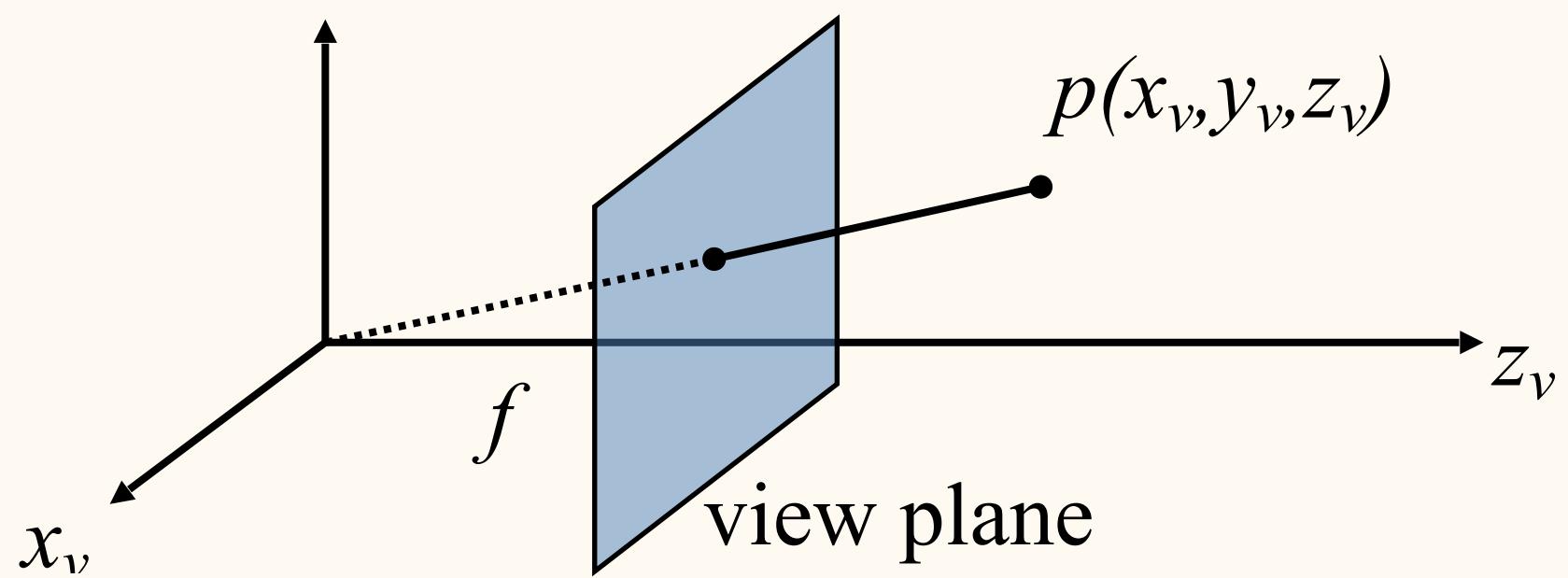


perspective projection



perspective projection + hidden line removed

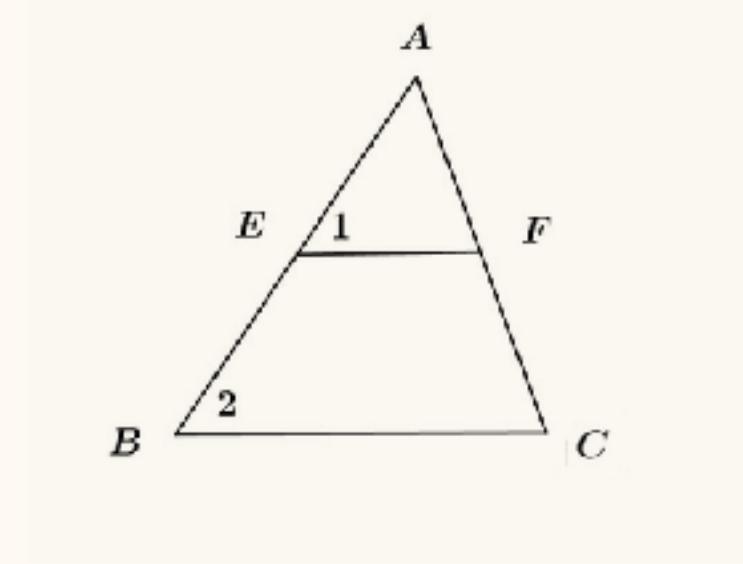
Perspective Projection



由相似三角形理論：

$$\frac{x_s}{f} = \frac{x_v}{z_v} \quad \frac{y_s}{f} = \frac{y_v}{z_v}$$

$$\Rightarrow x_s = \frac{x_v}{z_v/f} \quad y_s = \frac{y_v}{z_v/f}$$



Perspective Projection Matrix

- 為了將前一頁非線性轉換寫成矩陣相乘的表示法，將透視投影計算分成兩個步驟：
 - 第一步：使用 Homogeneous coordinate:

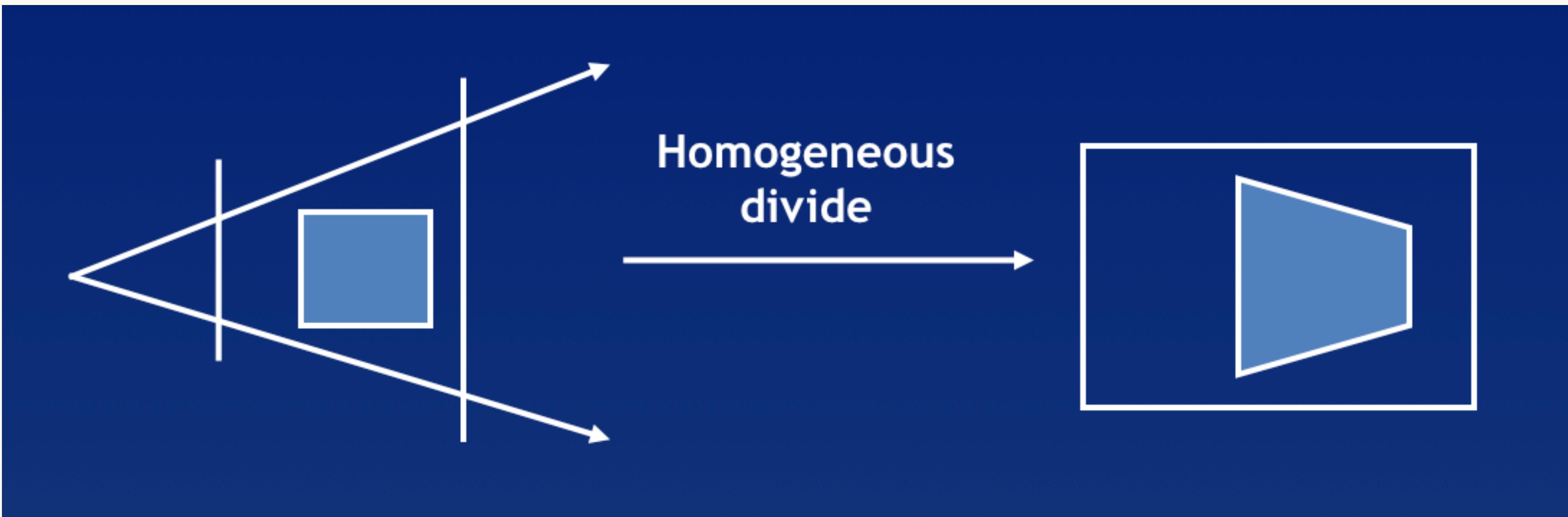
$$[X \ Y \ Z \ w]^T = [T_{persp}][x_v \ y_v \ z_v \ 1]^T \quad [T_{persp}] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix}$$

- 第二步：Homogeneous divide (Perspective divide)

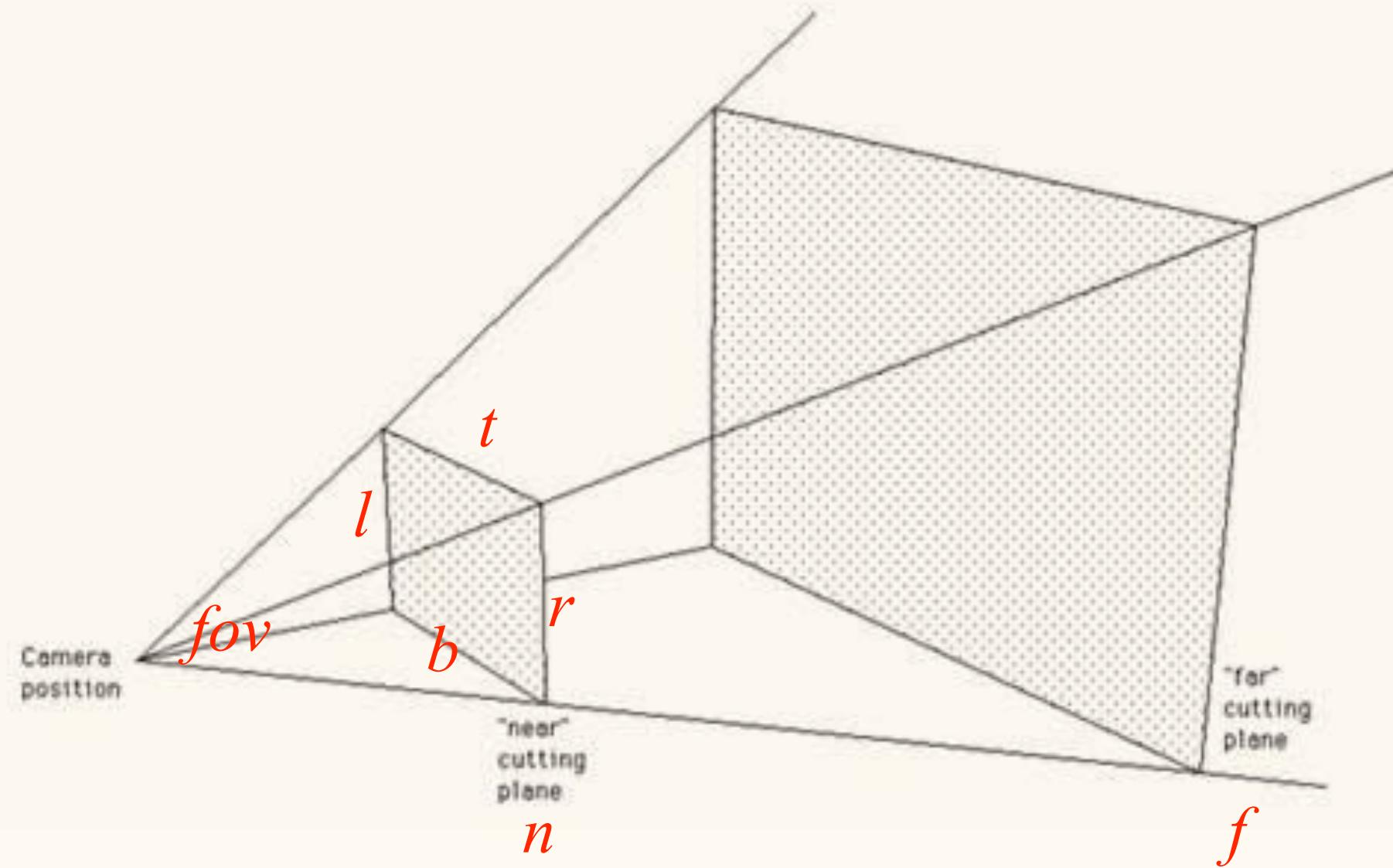
$$x_s = \frac{X}{w} \quad y_s = \frac{Y}{w} \quad z_s = \frac{Z}{w}$$

Homogeneous Divide

$$x_s = \frac{X}{w} \quad y_s = \frac{Y}{w} \quad z_s = \frac{Z}{w}$$



A General Projection Matrix



if ($l = -r$) $n = r/\tan(fov/2)$

$$[T_{persp}] = \begin{bmatrix} 2n/(r-l) & 0 & (r+l)/(r-l) & 0 \\ 0 & 2n/(t-b) & (t+b)/(t-b) & 0 \\ 0 & 0 & f/(n-f) & 2fn/(n-f) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

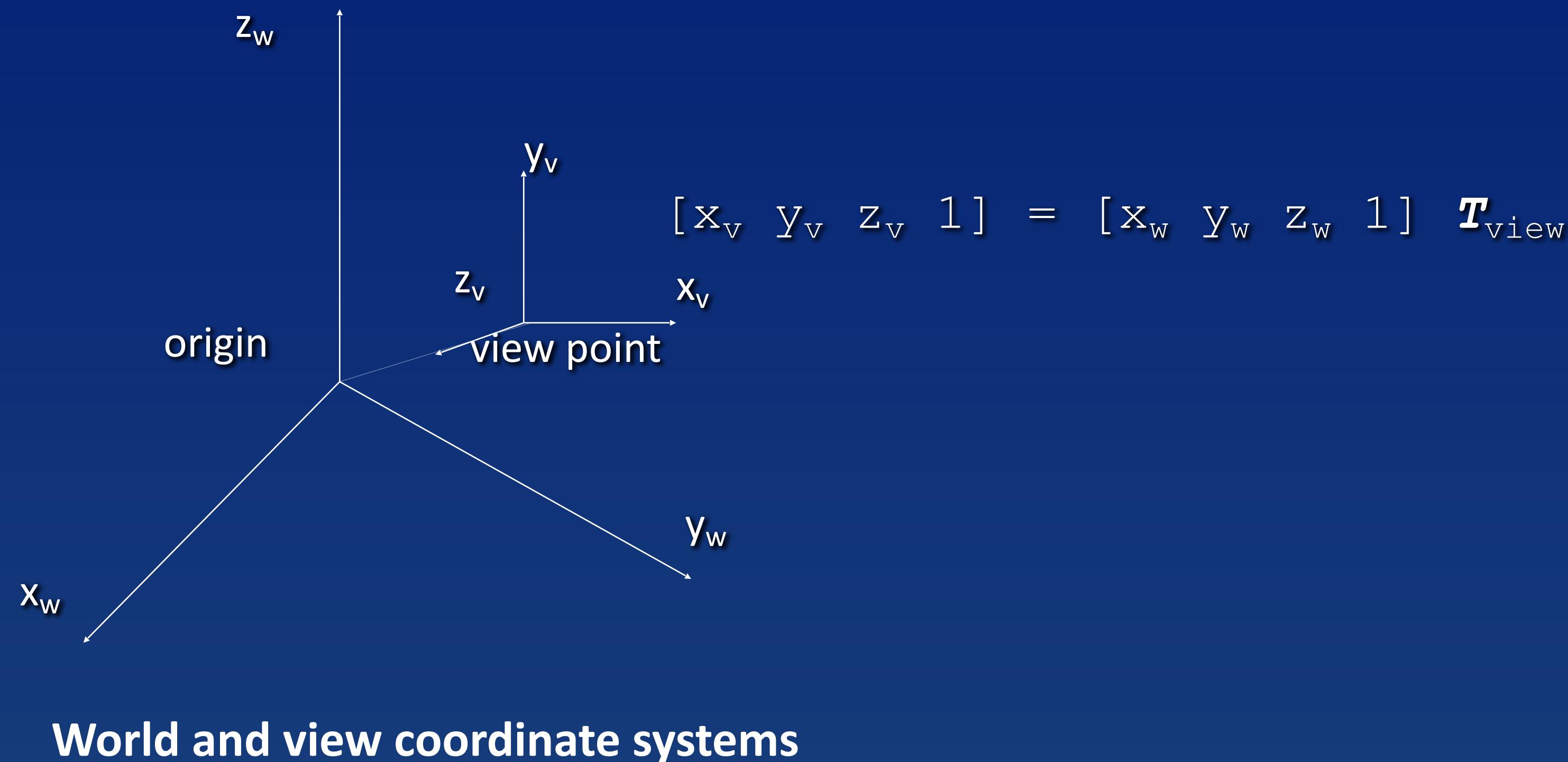
$$[T_{ortho}] = \begin{bmatrix} 2/(r-l)/s & 0 & 0 & 0 \\ 0 & 2/(t-b)/s & 0 & 0 \\ 0 & 0 & 1/(n-f) & 0 \\ (r+l)/(l-r)/s & (t+b)/(b-t)/s & n/(n-f) & 1 \end{bmatrix}$$

s : scale factor

Camera Viewing System

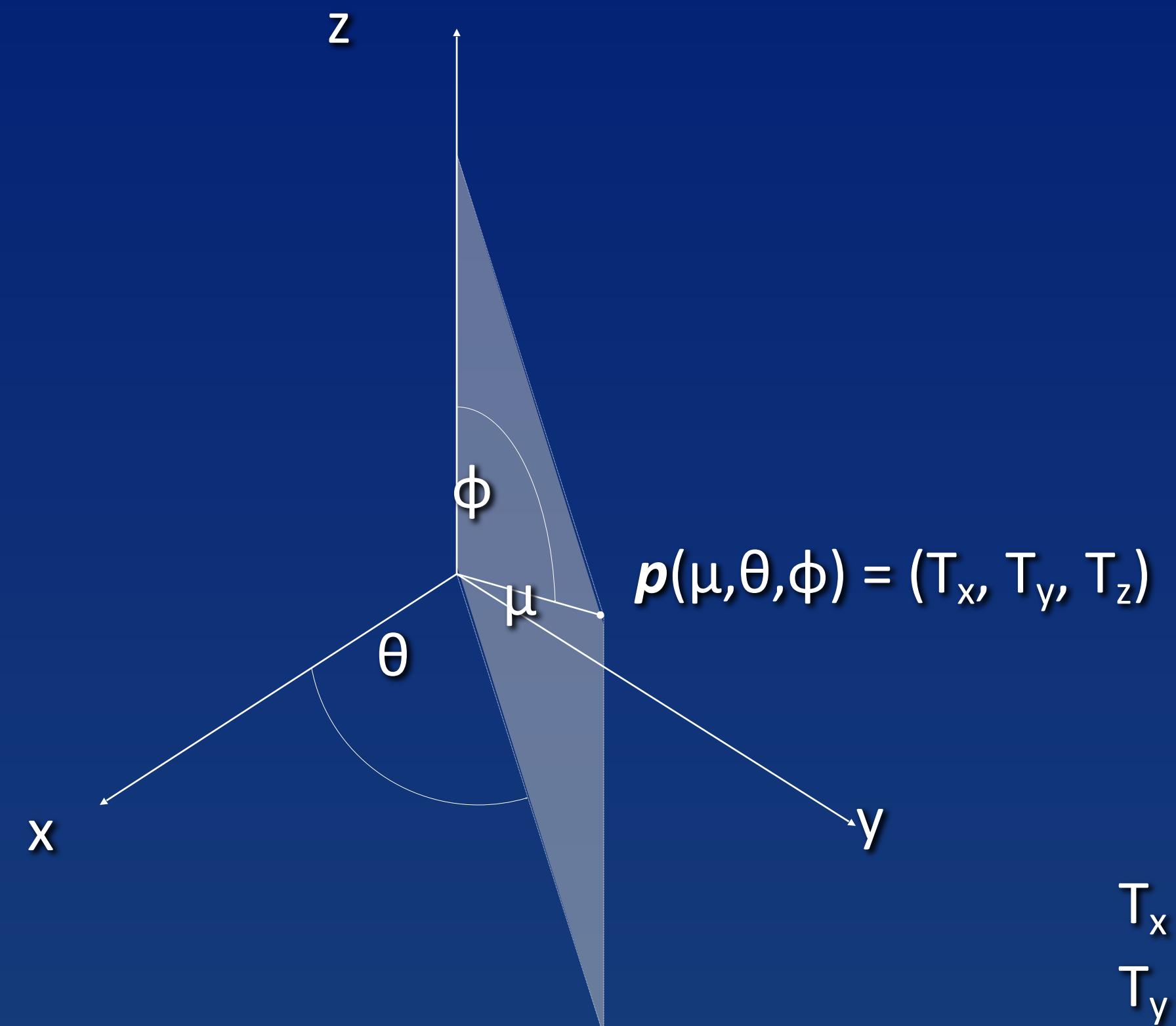
- 與相機的規格無關
- 與相機的位置與方向有關
 - 4x4 Viewing Matrix
 - 將 3D 模型從 World space 轉換到 Camera space (View Space)
 - 讓 z 軸對齊在視線軸上，可將 (x, y) 投影在視平面 (View plane) 上

A Simple Viewing Implementation Step-by-step (1)



You can parent this camera to a target object.

A Simple Viewing Implementation Step-by-step (2)



$$p(\mu, \theta, \phi) = (T_x, T_y, T_z)$$

$$T_x = \mu \sin\phi \cos \theta$$

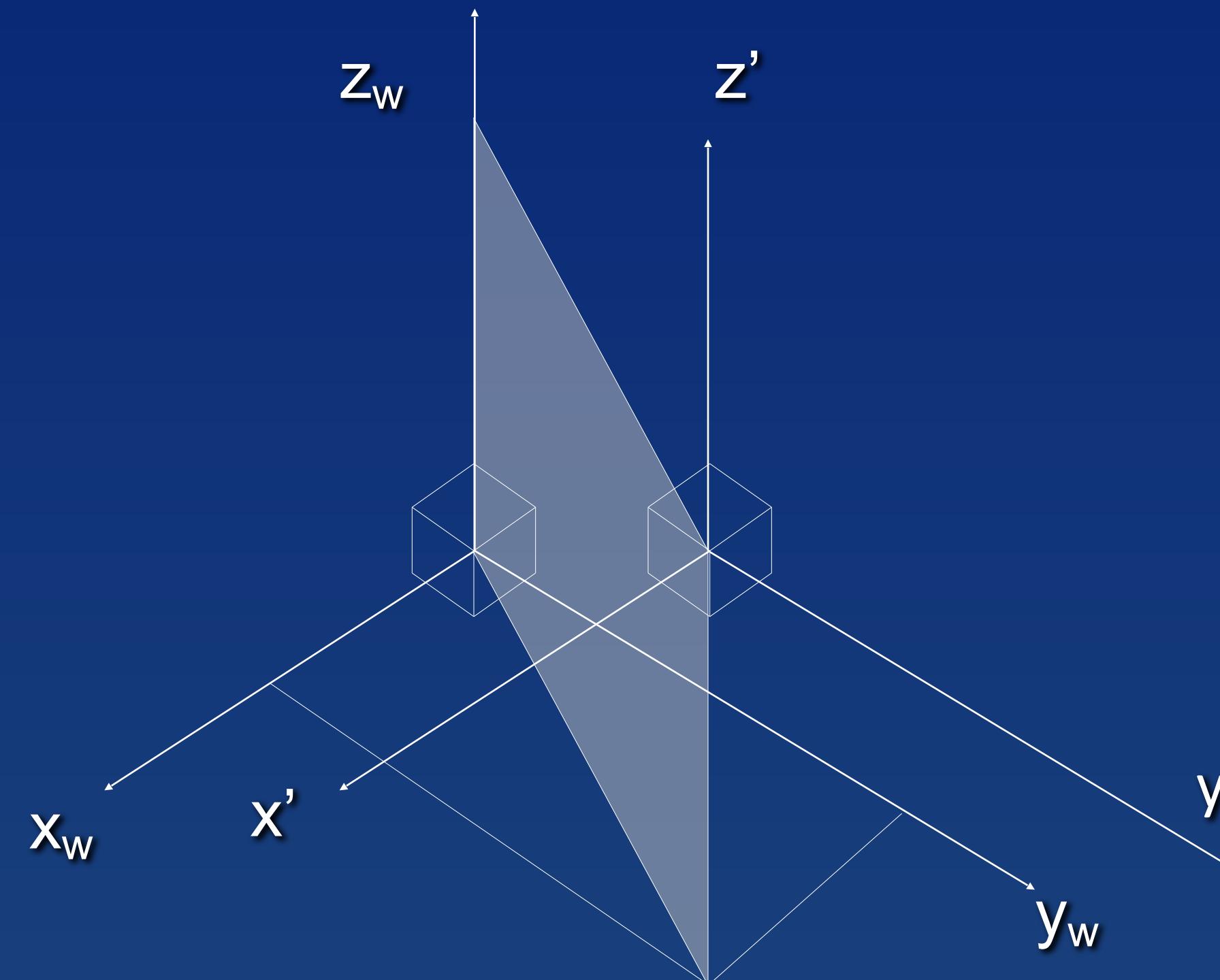
$$T_y = \mu \sin\phi \sin \theta$$

$$T_z = \mu \cos\phi$$

Spherical coordinates system

A Simple Viewing Implementation Step-by-step (3)

Step (1) Translate the origin to the view point (T_x, T_y, T_z)



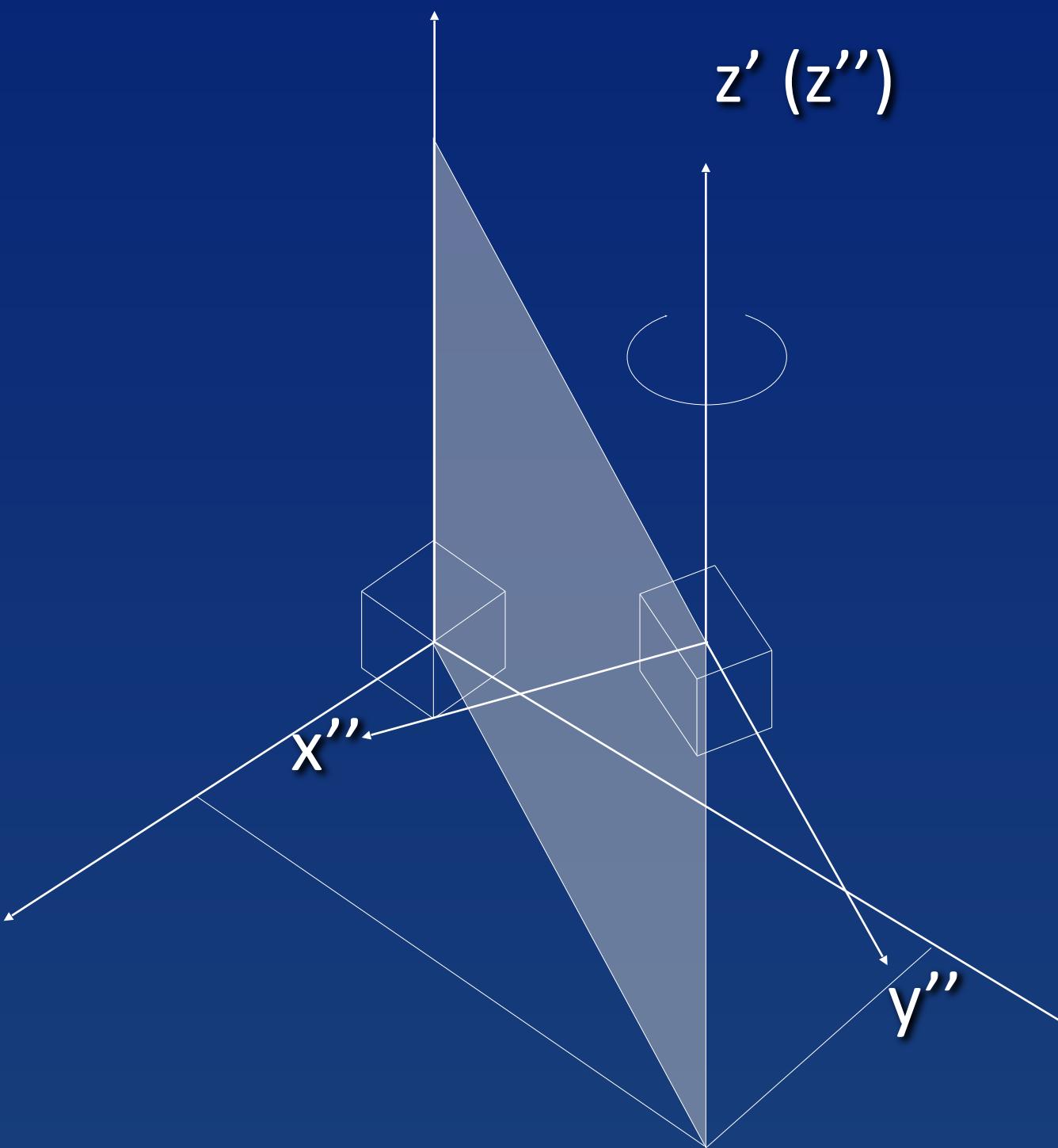
$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -T_x & -T_y & -T_z & 1 \end{bmatrix}$$

$$T_x = \mu \sin\phi \cos \theta$$

$$T_y = \mu \sin\phi \sin \theta$$

$$T_z = \mu \cos\phi$$

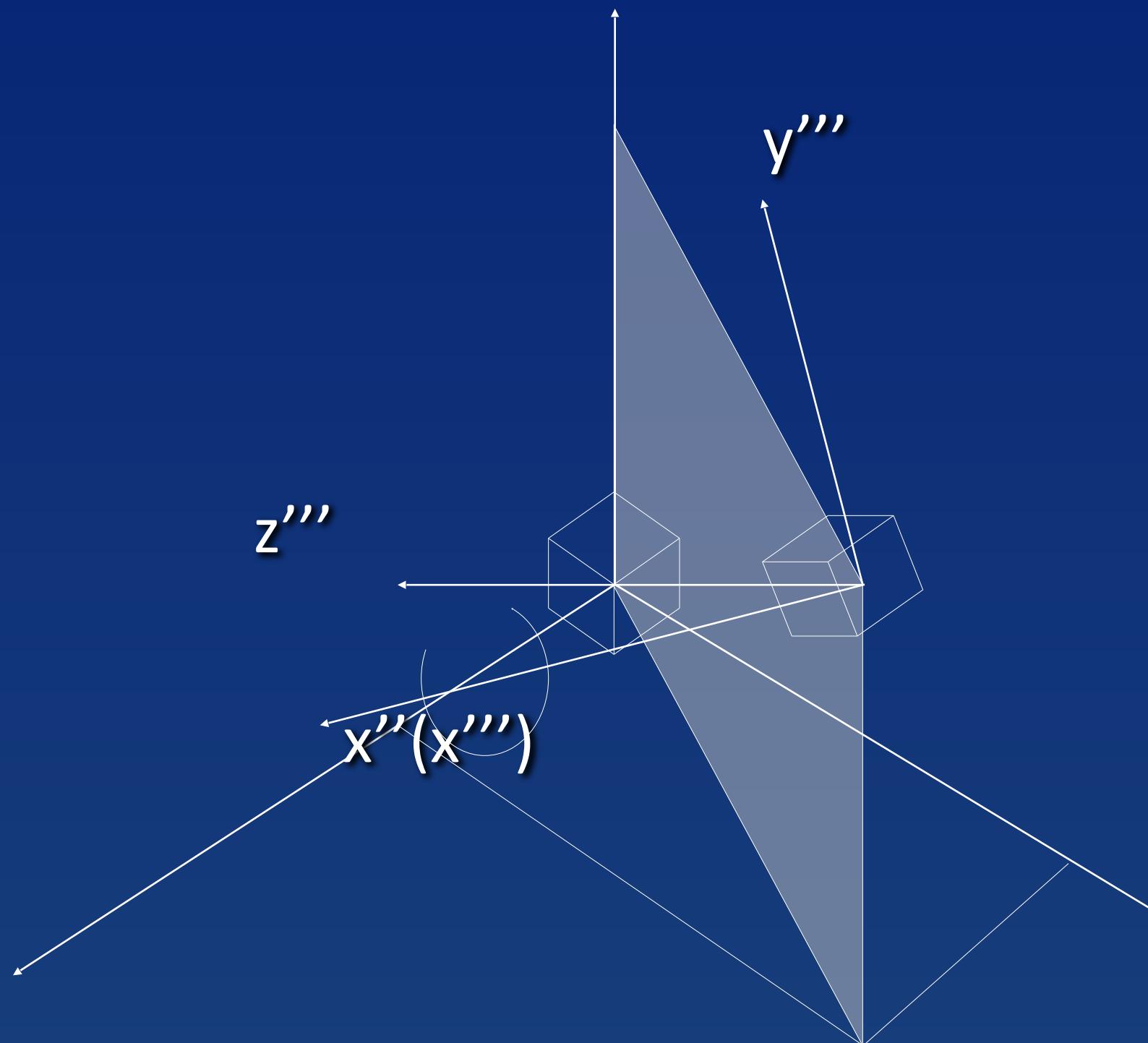
A Simple Viewing Implementation Step-by-step (4)



**Step (2) Rotate the coordinate system
through $90 - \theta$ in a clockwise
direction about the z' axis**

$$T_2 = \begin{bmatrix} \sin\theta & \cos\theta & 0 & 0 \\ -\cos\theta & \sin\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

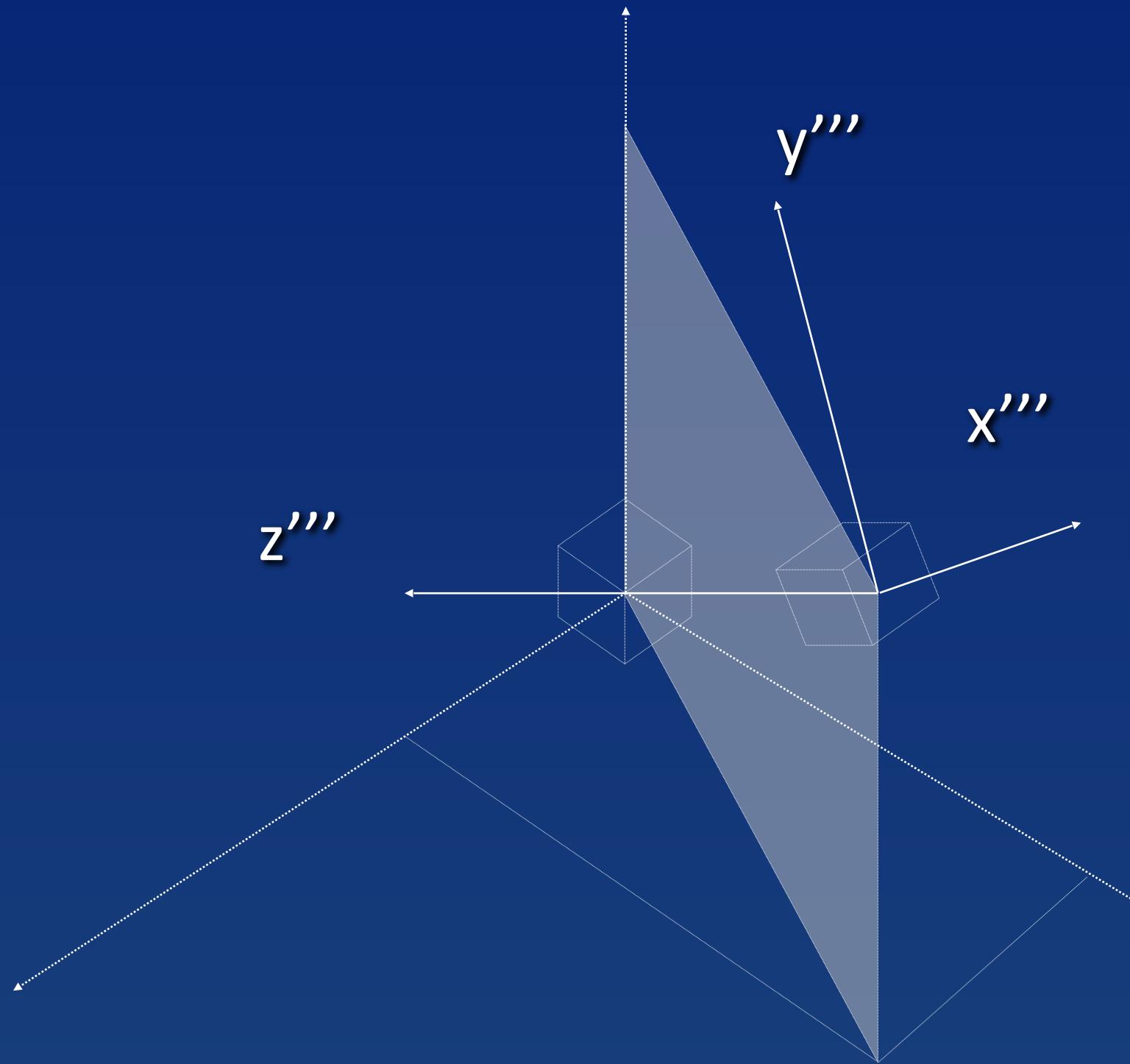
A Simple Viewing Implementation Step-by-step (5)



Step (3) Rotate the coordinate system through $180 - \phi$ in a counter-clockwise direction about the x'' axis to make z''' axis pass through the origin of the world space

$$T_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -\cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & -\cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A Simple Viewing Implementation Step-by-step (6)



Step (4) Convert to a left-handed system

$$T_4 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A Simple Viewing Implementation Step-by-step (7)

Step (5) Multiply these T s together to give the net transformation matrix required for the viewing transformation

$$T_{\text{view}} = \begin{bmatrix} -\sin\theta & -\cos\theta\cos\varphi & -\cos\theta\sin\varphi & 0 \\ \cos\theta & -\sin\theta\cos\varphi & -\sin\theta\sin\varphi & 0 \\ 0 & \sin\varphi & -\cos\varphi & 0 \\ 0 & 0 & \mu & 1 \end{bmatrix}$$

A General Camera Viewing Matrix

- 坐標系之間的轉換，可視為矩陣的反運算

$$\begin{aligned} v' &= [M_{A-B}]v \\ &\Rightarrow [M_{B-A}] = [M_{A-B}]^{-1} \\ v &= [M_{B-A}]v' \end{aligned}$$

- 如果 Transformation 是 Orthogonal, 反矩陣 = 轉置矩陣
 - Rotation transformation 是 Orthogonal

$$[M_{B-A}] = [M_{A-B}]^{-1} = [M_{A-B}]^T$$

A General Camera Viewing Matrix

$$[M] = [T][R] = \begin{bmatrix} r_0 & r_1 & r_2 & t_3 \\ r_4 & r_5 & r_6 & t_7 \\ r_8 & r_9 & r_{10} & t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_3 \\ 0 & 1 & 0 & t_7 \\ 0 & 0 & 1 & t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_0 & r_1 & r_2 & 0 \\ r_4 & r_5 & r_6 & 0 \\ r_8 & r_9 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[M]^{-1} = [R]^{-1}[T]^{-1} = R^T[T]^{-1} = \begin{bmatrix} r_0 & r_4 & r_8 & 0 \\ r_1 & r_5 & r_9 & 0 \\ r_2 & r_6 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -t_3 \\ 0 & 1 & 0 & -t_7 \\ 0 & 0 & 1 & -t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A General Camera Viewing Matrix

camera's transformation matrix

$$[M_{camera}] = [T][R] = \begin{bmatrix} r_0 & r_1 & r_2 & t_3 \\ r_4 & r_5 & r_6 & t_7 \\ r_8 & r_9 & r_{10} & t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_3 \\ 0 & 1 & 0 & t_7 \\ 0 & 0 & 1 & t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_0 & r_1 & r_2 & 0 \\ r_4 & r_5 & r_6 & 0 \\ r_8 & r_9 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

viewing matrix

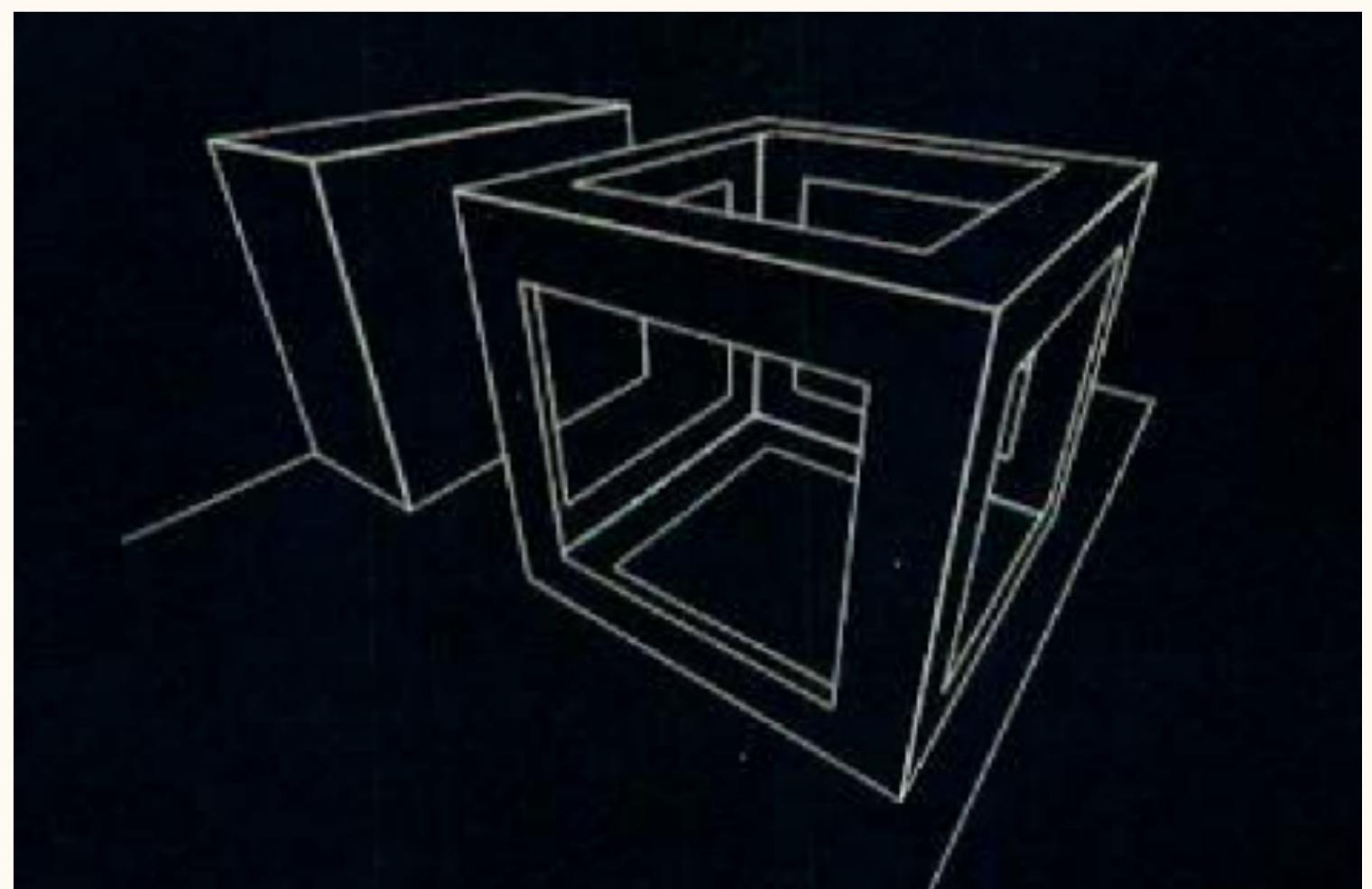
$$[V] = [M_{camera}]^{-1} = [R]^T[T]^{-1} = \begin{bmatrix} r_0 & r_4 & r_8 & 0 \\ r_1 & r_5 & r_9 & 0 \\ r_2 & r_6 & r_{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -t_3 \\ 0 & 1 & 0 & -t_7 \\ 0 & 0 & 1 & -t_{11} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hidden-surface Removal

隱藏面消除

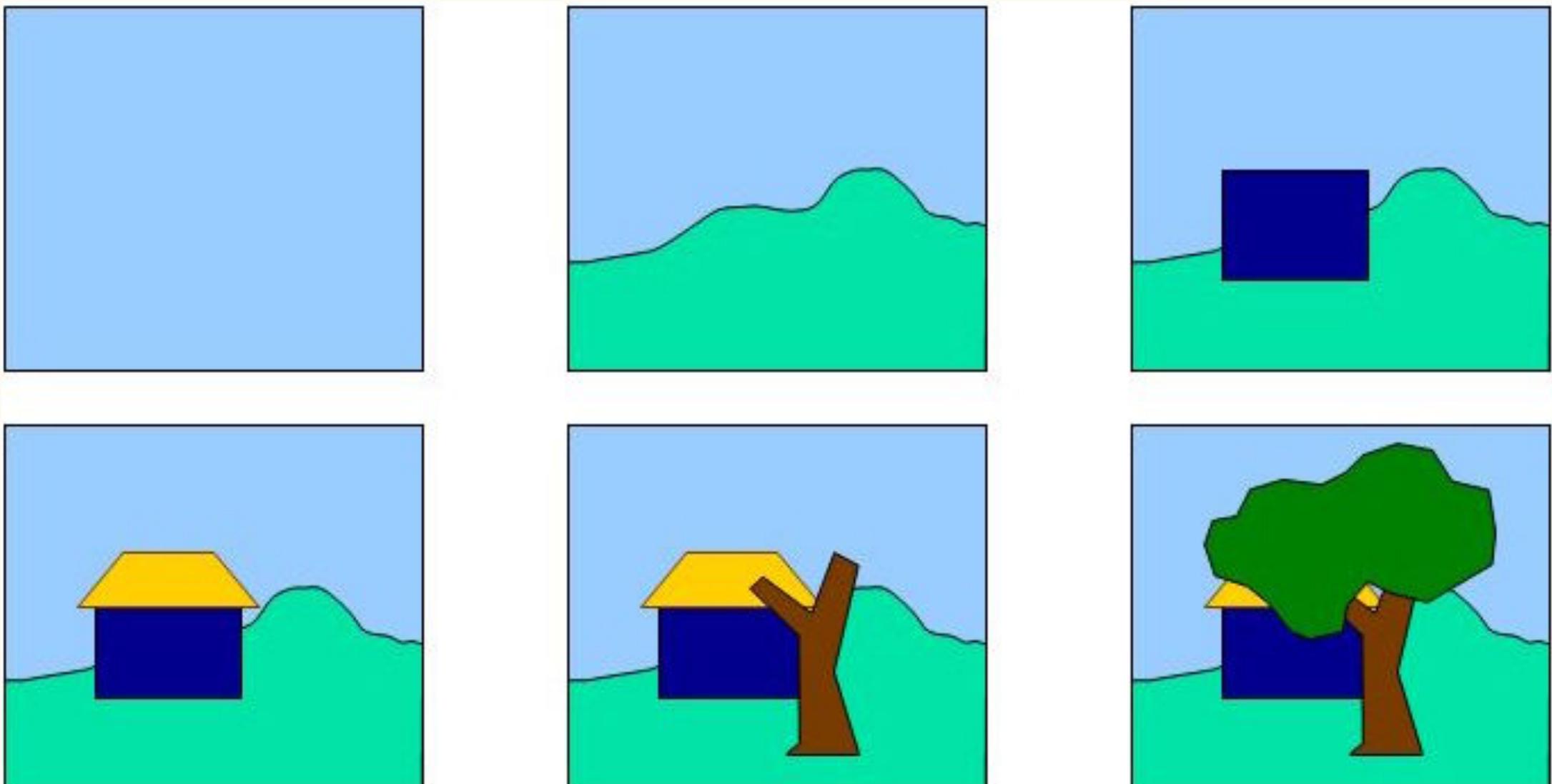
隱藏面消除演算法

- Painter's algorithm
 - “油漆匠演算法”
 - 早期3D遊戲使用的方法
 - Sony PlayStation (PS)
- Z-buffering algorithm
 - 現代硬體所使用的方法
- Binary space partitioning (BSP)
 - 僅限於靜態物件
- Ray casting

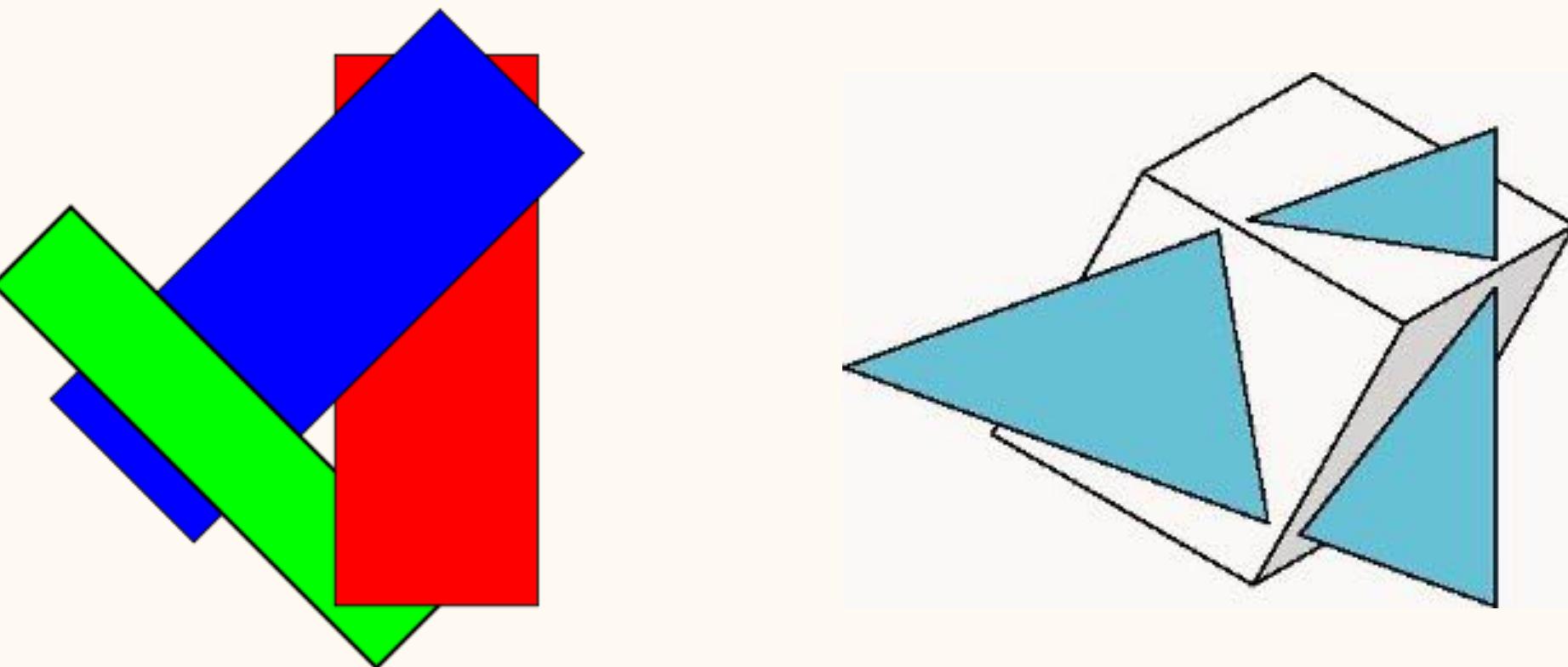


Painter's Algorithm

- 方法
 - 依與攝影機的距離排序 (Depth sorting)
 - By polygons
 - By objects
 - 再依序渲染



- 限制
 - 物件穿插
 - 物件相互重疊

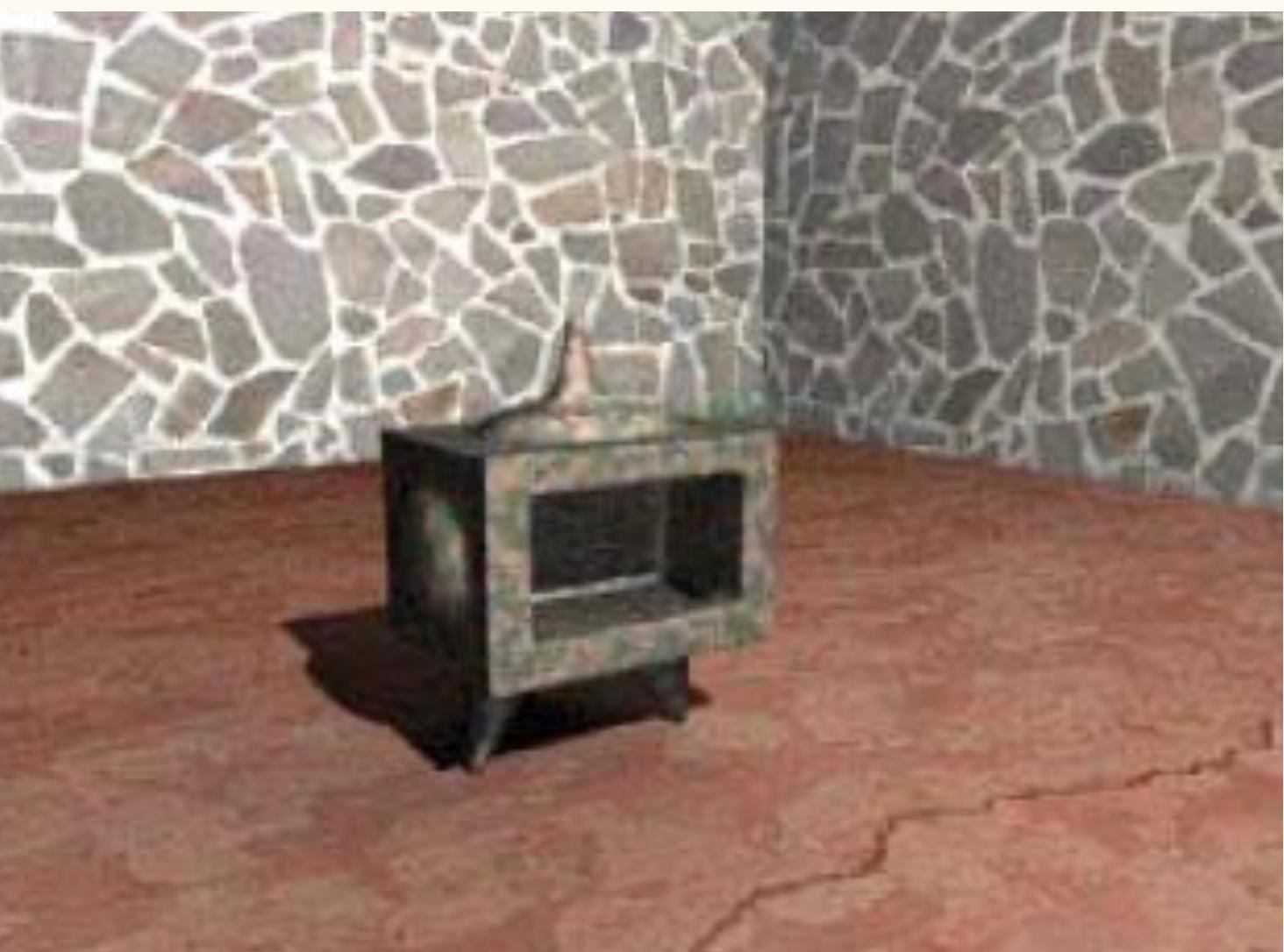


Z-buffering

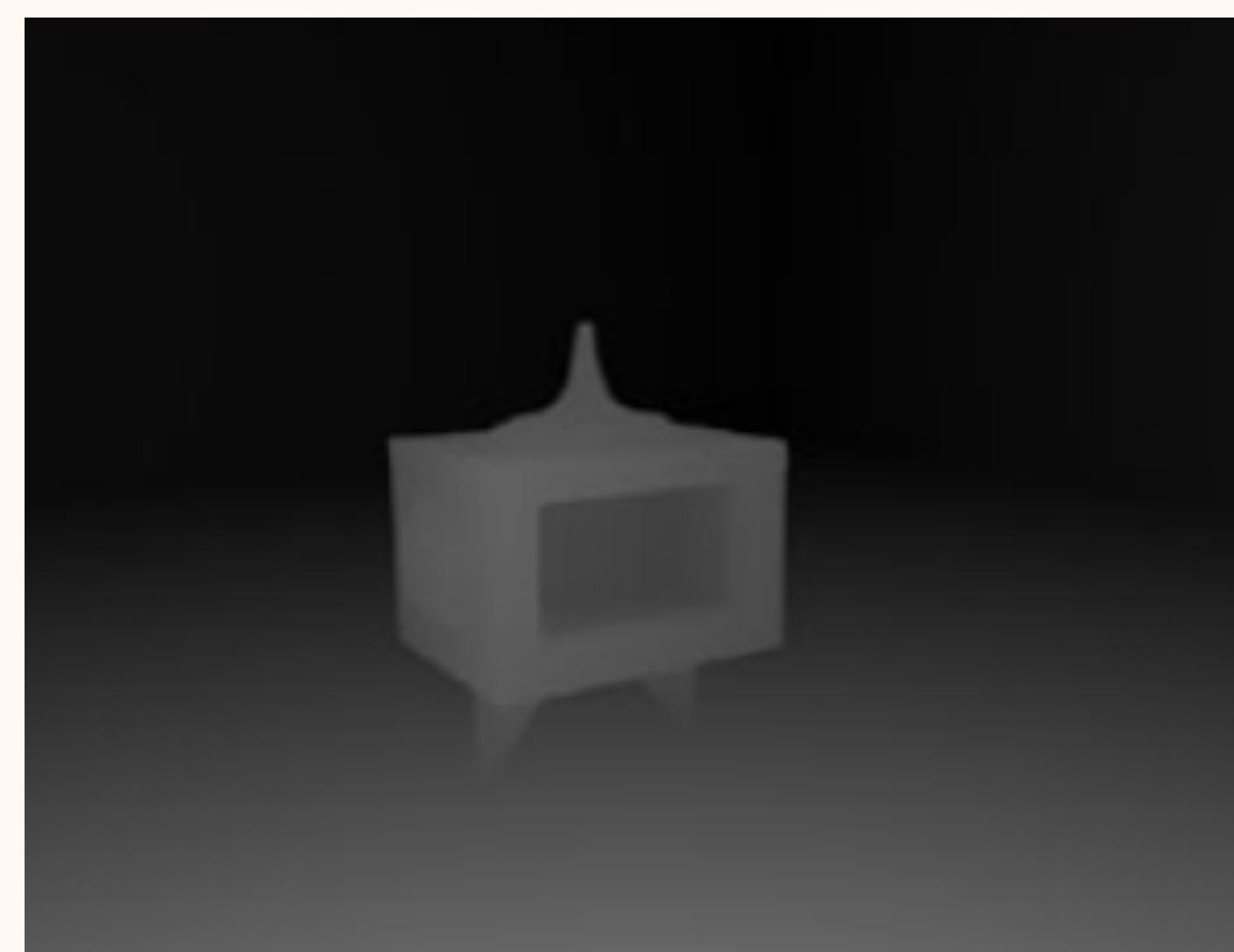
- 記錄每個像素的深度 (resolve the depth in pixel level)
 - $\text{depth} = \text{該像素目前渲染顏色者的 z 值}$ (攝影機座標系)

(r, g, b) depth

儲存在 Backbuffer
(color buffer)



儲存在 Z-buffer
(depth buffer)



Z-buffering

- Z buffer
 - 一塊與 Backbuffer 解析度相同的記憶體 (in video memory)，用來儲存 z 值
 - 當繪製一個像素時，檢查它是否比 Z buffer 中已有的更接近攝影機。如果是，繪製該像素並更新 Z buffer 中的值；否則，放棄繪製該像素

The diagram illustrates Z-buffering operations using two initial Z-buffer matrices, two addition operations (one with a scalar and one with a vector), and their resulting matrices.

Initial Z-buffer 1:

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Initial Z-buffer 2:

5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5

Addition 1: Initial Z-buffer 1 + Scalar 5 = Result 1

5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5

Addition 2: Initial Z-buffer 2 + Vector [3, 4, 3, 6, 5, 4, 3, 7, 6] = Result 2

5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5

Z-buffering Algorithm

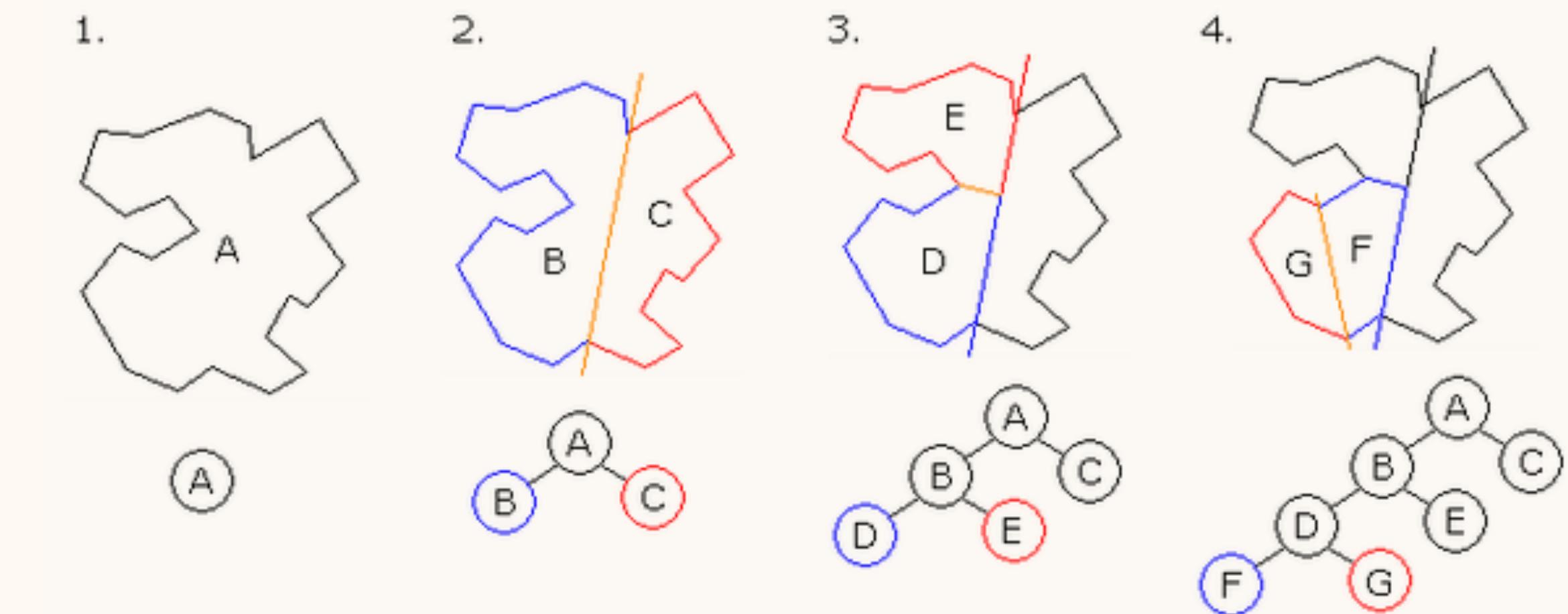
```
void zBuffer()
{
    int pz;
    for (each polygon) {
        for (each pixel in polygon's projection)  {
            pz = polygon's z-value at (x, y);
            if (pz > ReadZ(x, y)) {
                writeZ(x, y, pz);
                WritePixelColor(x, y, color);
            }
        }
    }
}
```

Z-buffering

- 優點
 - Easy
 - 對所有幾何元件都適用
 - 符合現代硬體渲染硬體架構
 - 可平行處理
 - 與 polygon 渲染次序無關
- 缺點
 - 需要配置一塊 Depth buffer = $w \times h \times 24\text{-bit}$
 - Aliasing !!!
 - Overfill !!! (重複繪製)
 - 半透明物件渲染會有狀況 (Transparency does not work well)

BSP: Binary Space Partition Tree

- 二叉空間分割樹
 - 一種空間分割的技術
 - 可運用於 rendering 上的排序
- Reference
 - Papers
 - By Fuchs, H., et al
 - “On Visible Surface Generation by a Priori Tree Structures”, (Proc. SIGGRAPH'80)
 - “Doom” by John Carmack
 - KD Tree
 - Axis-aligned BSP Tree



BSP: Binary Space Partition Tree

- Visibility algorithm using BSP (recursive version)

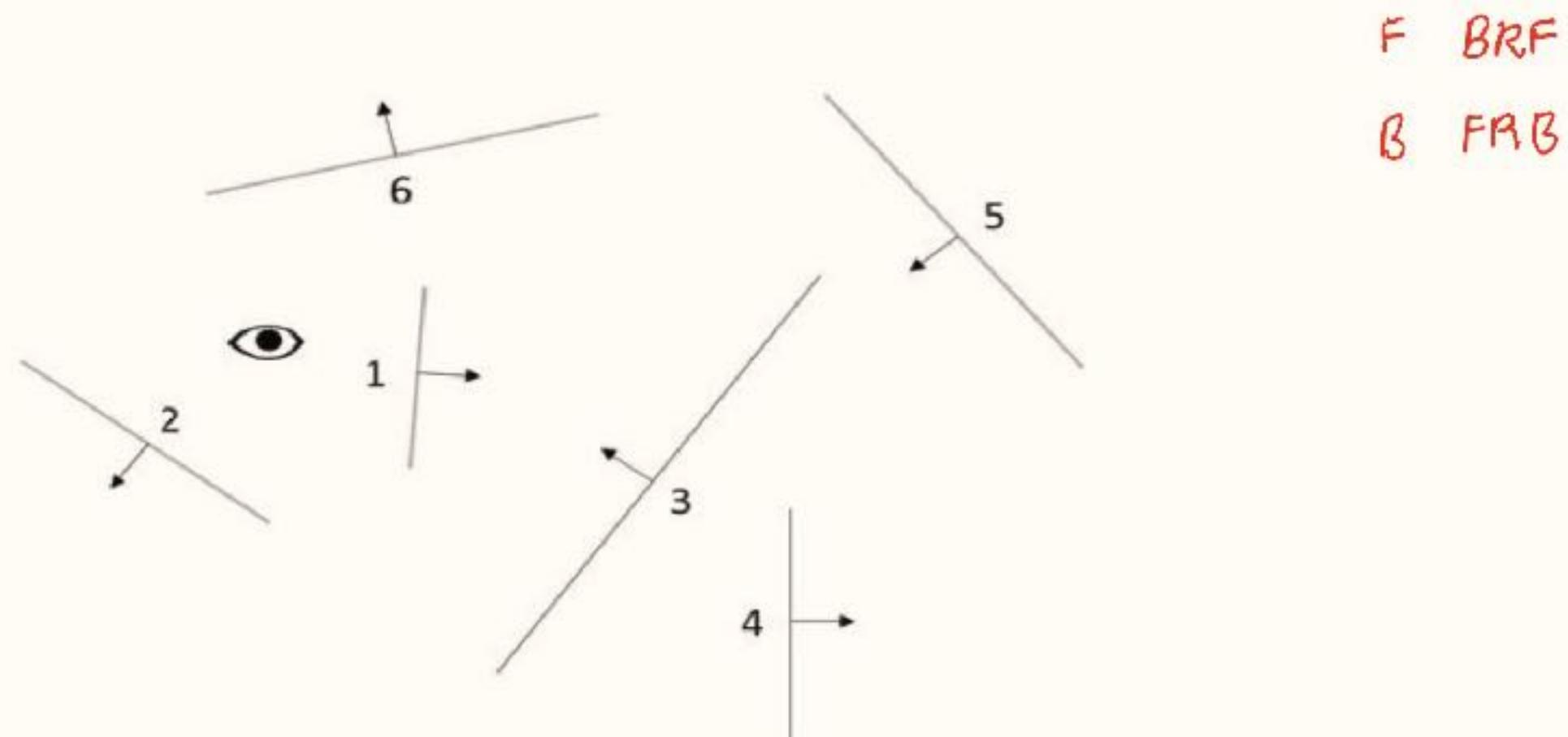
```
BSP_displayTree(tree)
{
    if (tree is NULL) return
    if viewer is in front of root, then {
        begin {display back child, root, and front child}
            BSP_displayTree(tree->backChild)
            displayPolygon(tree->root)
            BSP_displayTree(tree->frontChild)
        end
    }
    else {
        begin {display front child, root, and back child}
            BSP_displayTree(tree->frontChild)
            displayPolygon(tree->root)
            BSP_displayTree(tree->endChild)
        end
    }
}
```

BSP: Binary Space Partition Tree

- An example from 2018 NTU mid-term examination

1. BSP Tree (10%)

- Construct the Binary Space Partitioning (BSP) tree of the following figure. Please use the node “3” as the root, and choose smaller numbers as the sub-tree root node. (5%)
- From the BSP tree of previous question, derive the display sequence in terms of the given viewing position in this figure. (5%)

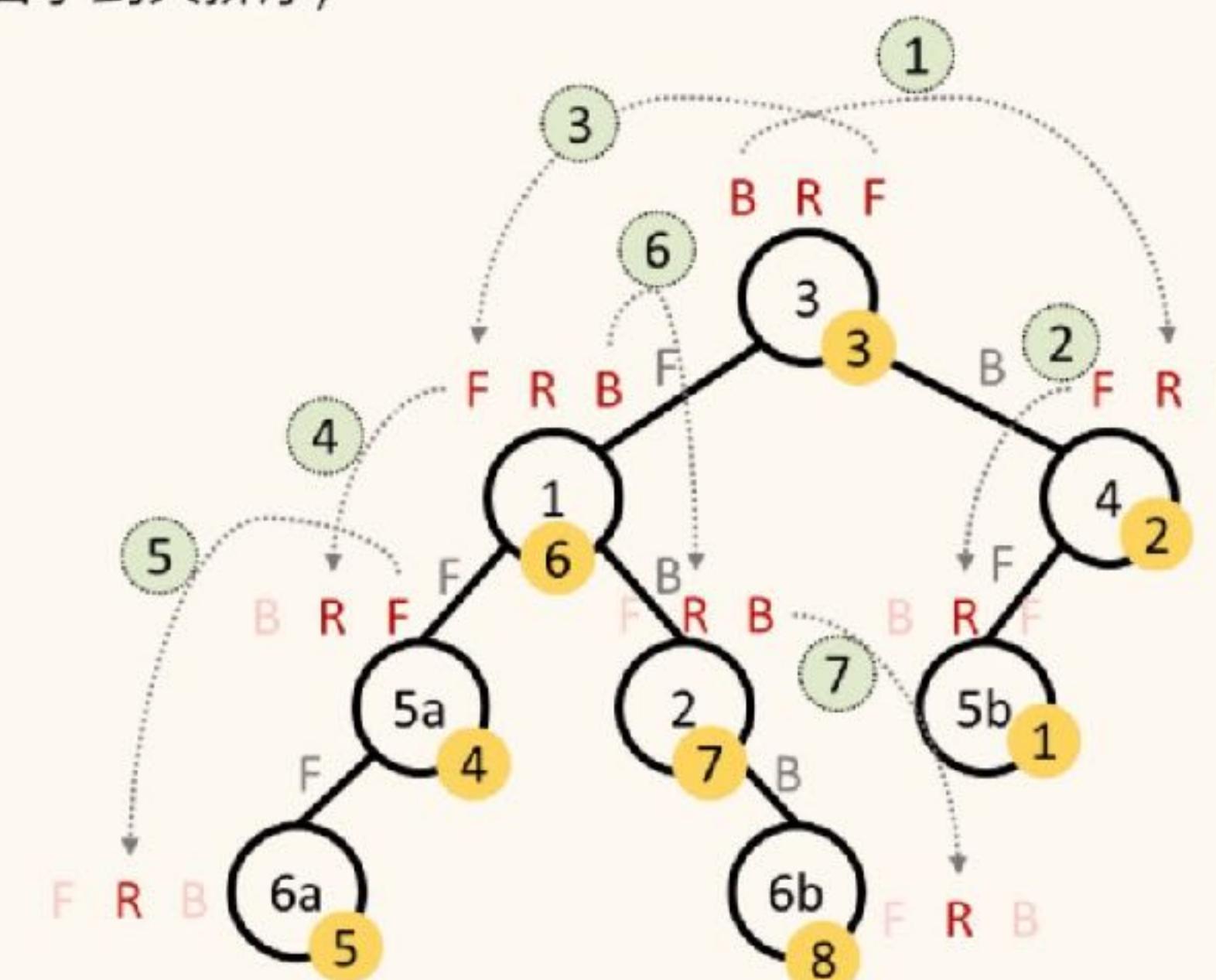
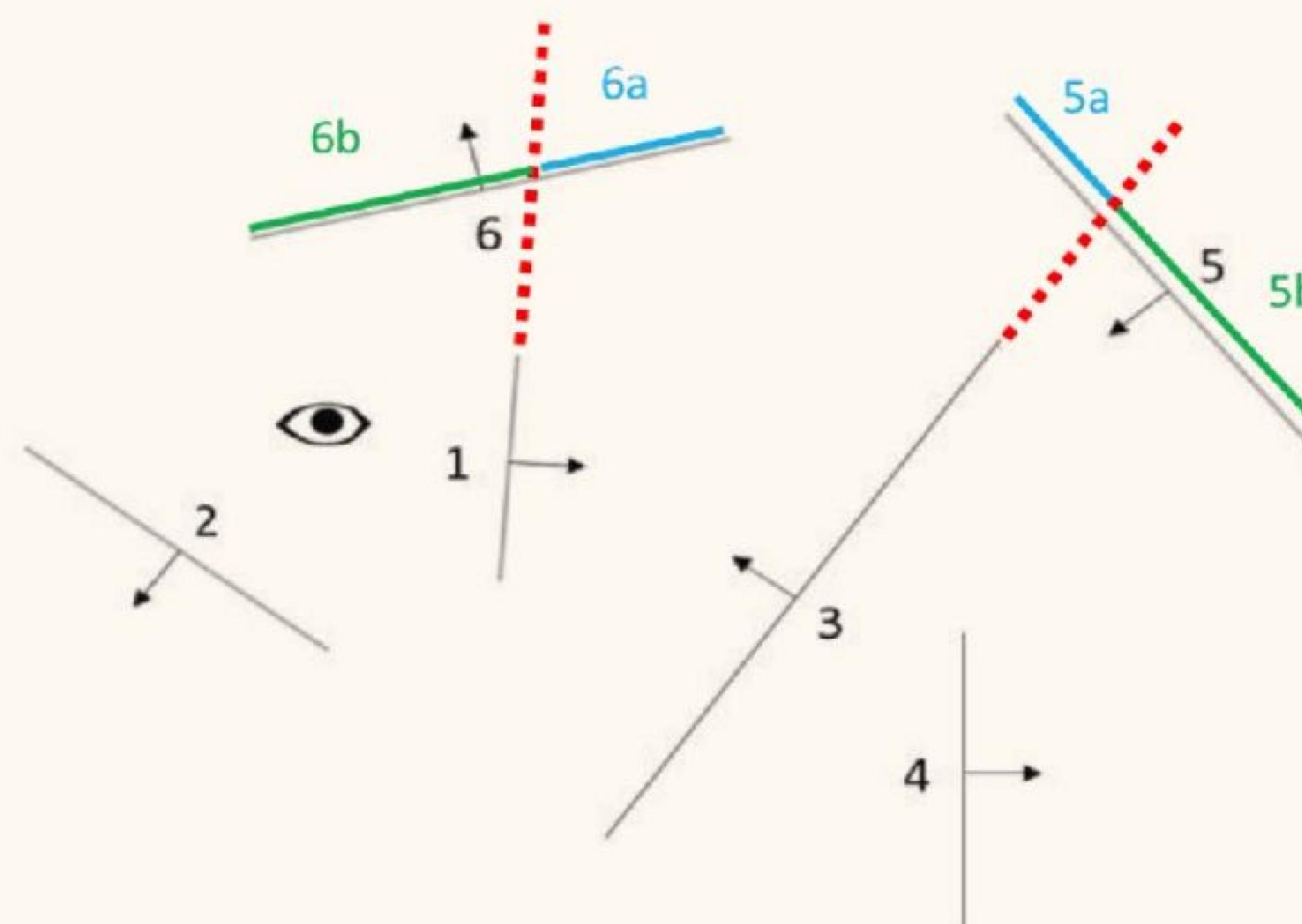


BSP: Binary Space Partition Tree

- An example from 2018 NTU mid-term examination

解答：

以上為a、下為b；若水平線才以左為a、以右為b(且數字由小到大排序)

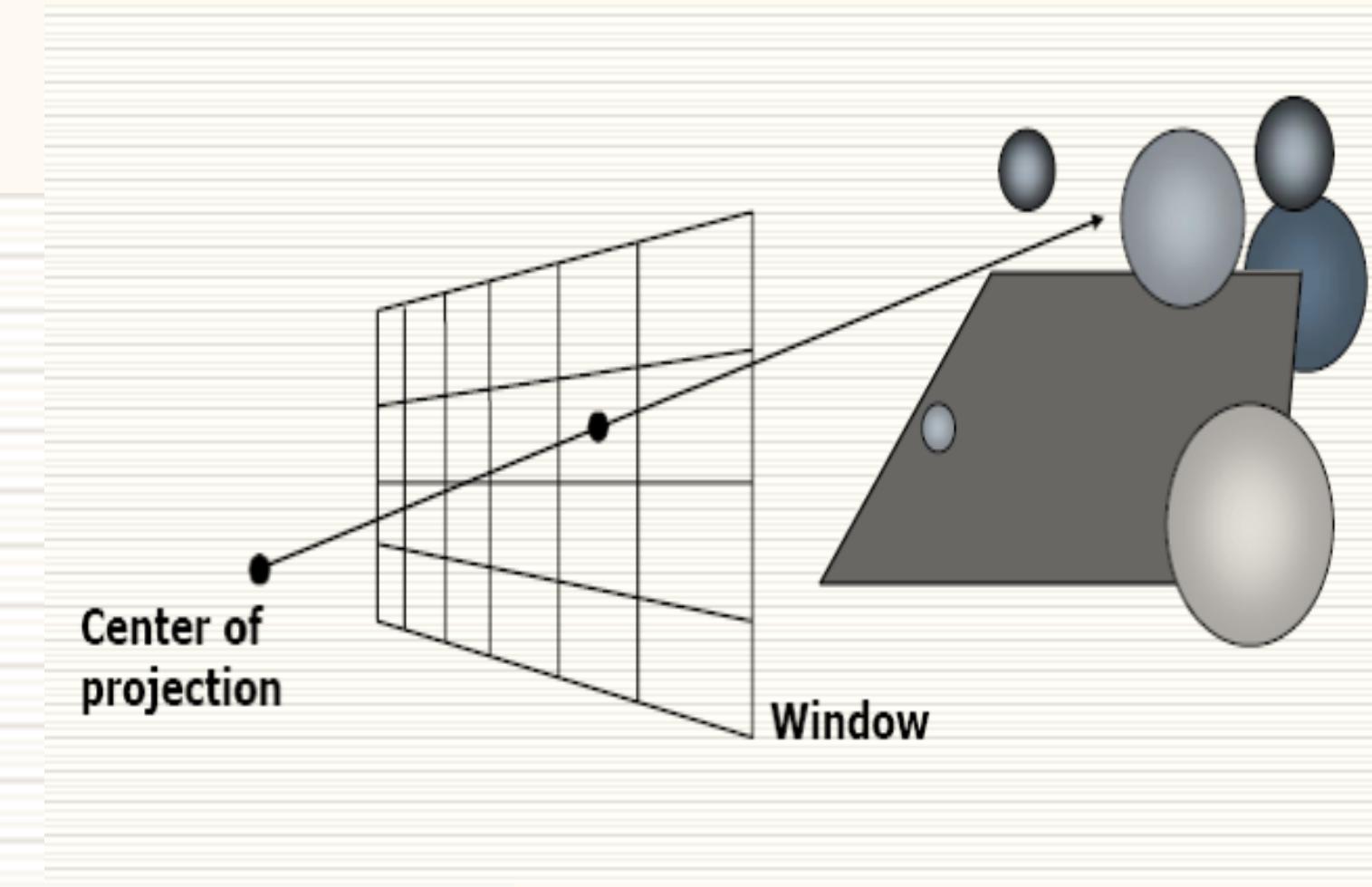


順序：5b, 4, 3, 5a, 6a, 1, 2, 6b

Ray Casting

- Rendering algorithm

```
select center of projection and window on viewplane;  
for (each scan line in image) {  
    for (each pixel in scan line) {  
        determine ray from center of projection through pixel;  
        for (each object in scene) {  
            if (object is intersected and is closest considered thus far)  
                record intersection and object name;  
        }  
        set pixel's color to that at closest object intersection;  
    }  
}
```

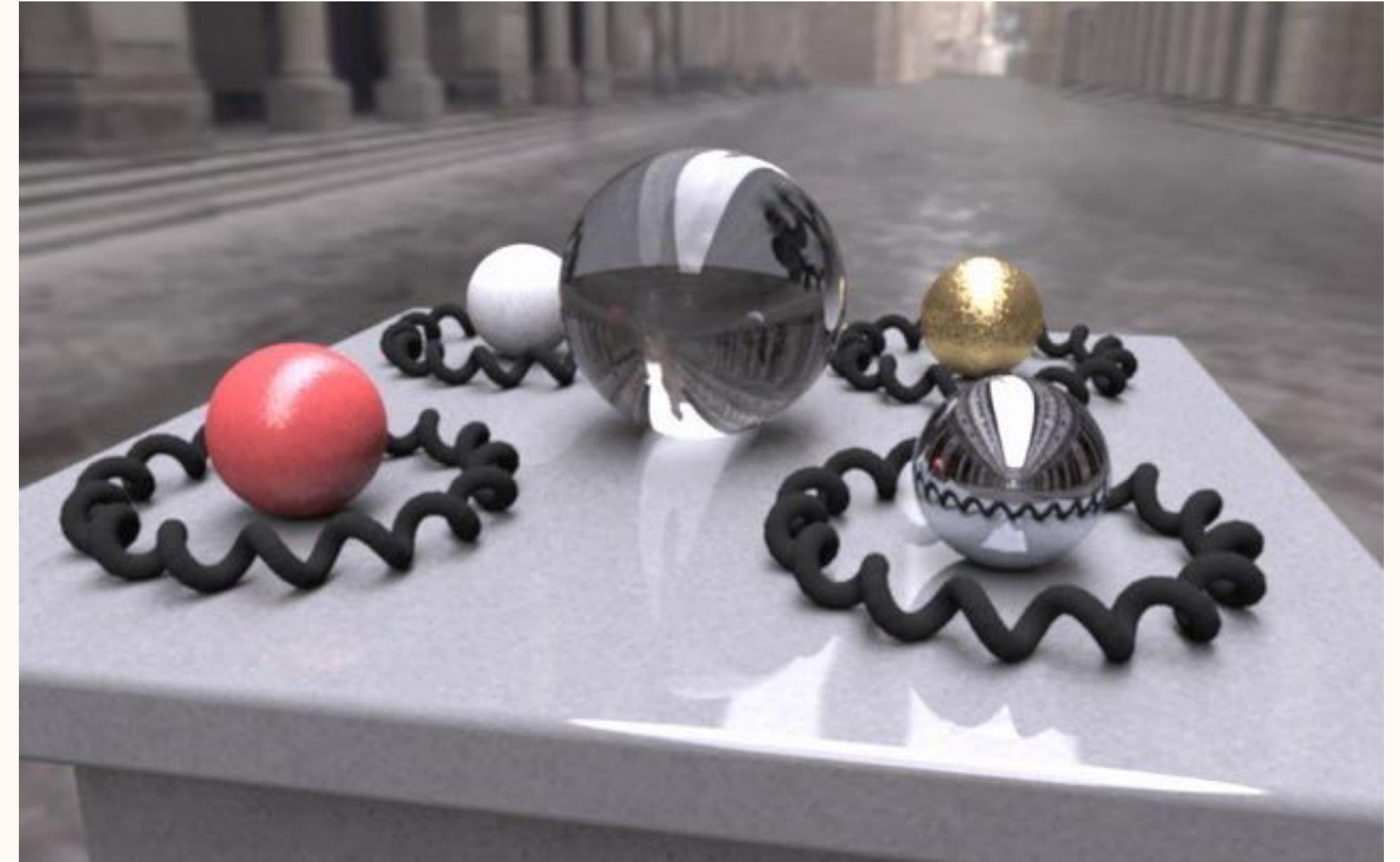


Materials & Textures

材質

物體表面材質

- Surface properties
 - Materials 材質
 - 影響物體表面被光線照射後反射的結果
 - Reflection models
 - Illumination models
 - Lighting/Shading
 - Textures 貼圖
 - 物體表面的圖形資料
 - Colors, normals, reflection parameters 等等
 - 以影像方式儲存
 - Shaders
 - 執行 Lighting 或 Shading 的程式



Material – BRDF (Bidirectional Reflectance Distribution Function)

Generally we define a BRDF as

$$R(l, \phi_i, \theta_i, \phi_v, \theta_v)$$

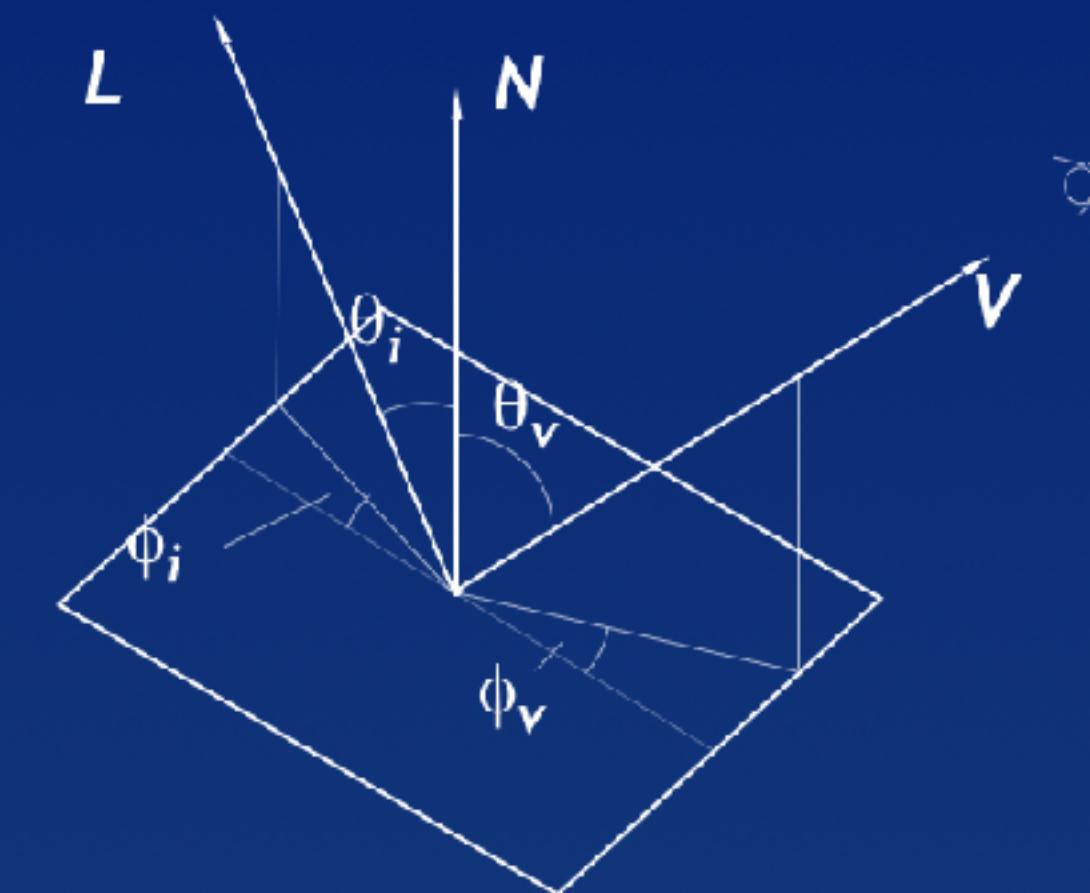
that relates incoming light in the direction (ϕ_i, θ_i) to outgoing light in the direction (ϕ_v, θ_v) . Theoretically, the BRDF is the ratio of outgoing intensity to incoming energy :

$$R(l, \phi_i, \theta_i, \phi_v, \theta_v) = \frac{I_v(\phi_i, \theta_i, \phi_v, \theta_v)}{E_i(\phi_i, \theta_i)}$$

where the relationship between the incoming energy and incoming intensity is given by

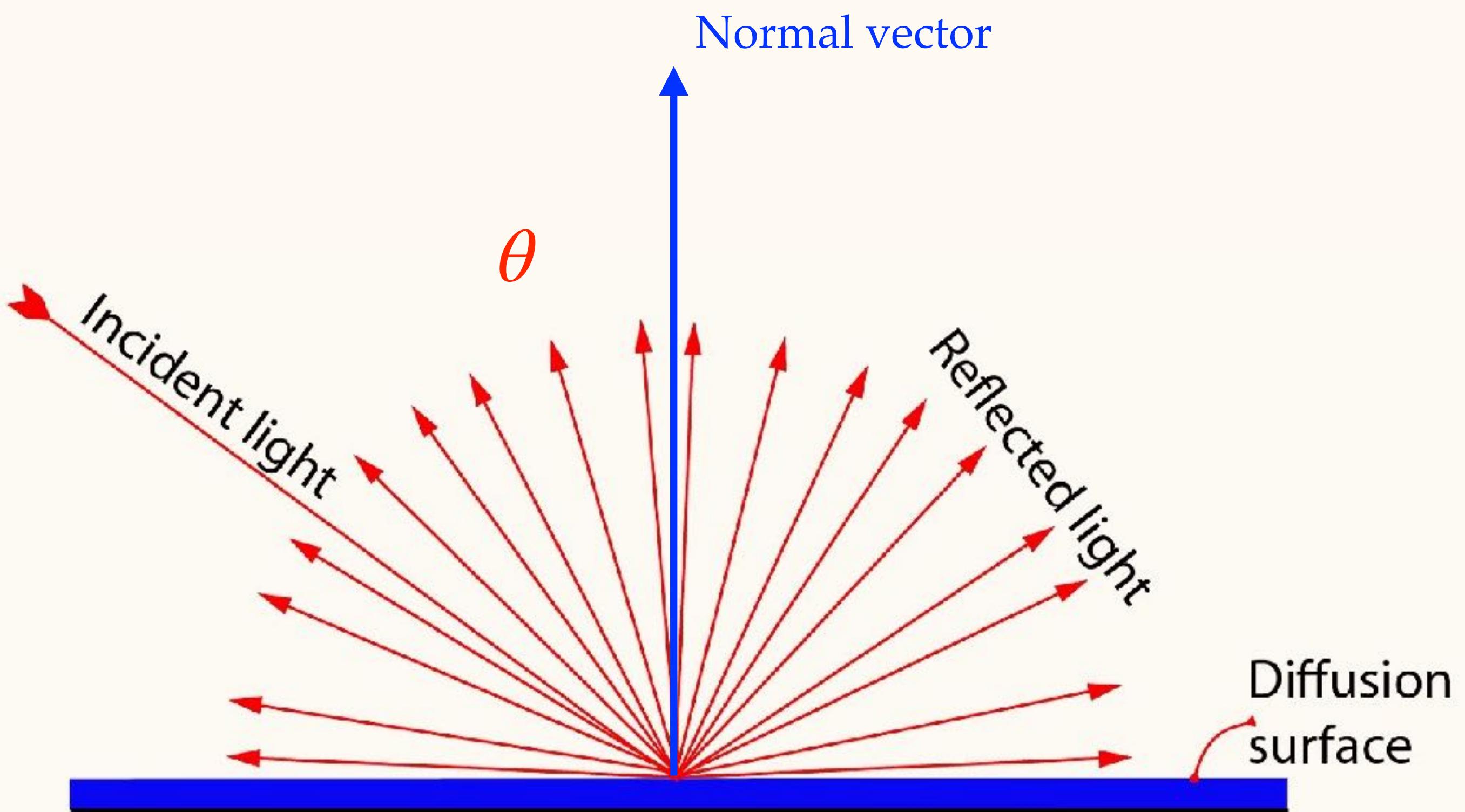
$$E_i(\phi_i, \theta_i) = I_i(\phi_i, \theta_i) \cos\theta_i d\omega_i$$

He et al.(1991) suggest that for computer graphics consideration, the BRDF should be divided into 3 components : a **specular contribution**, a **directional diffuse contribution** and an **ideal diffuse contribution**. The purpose of the division is to enable an analytical mode, based on both **physical** and **wave aspects** of optical theory, to be constructed.



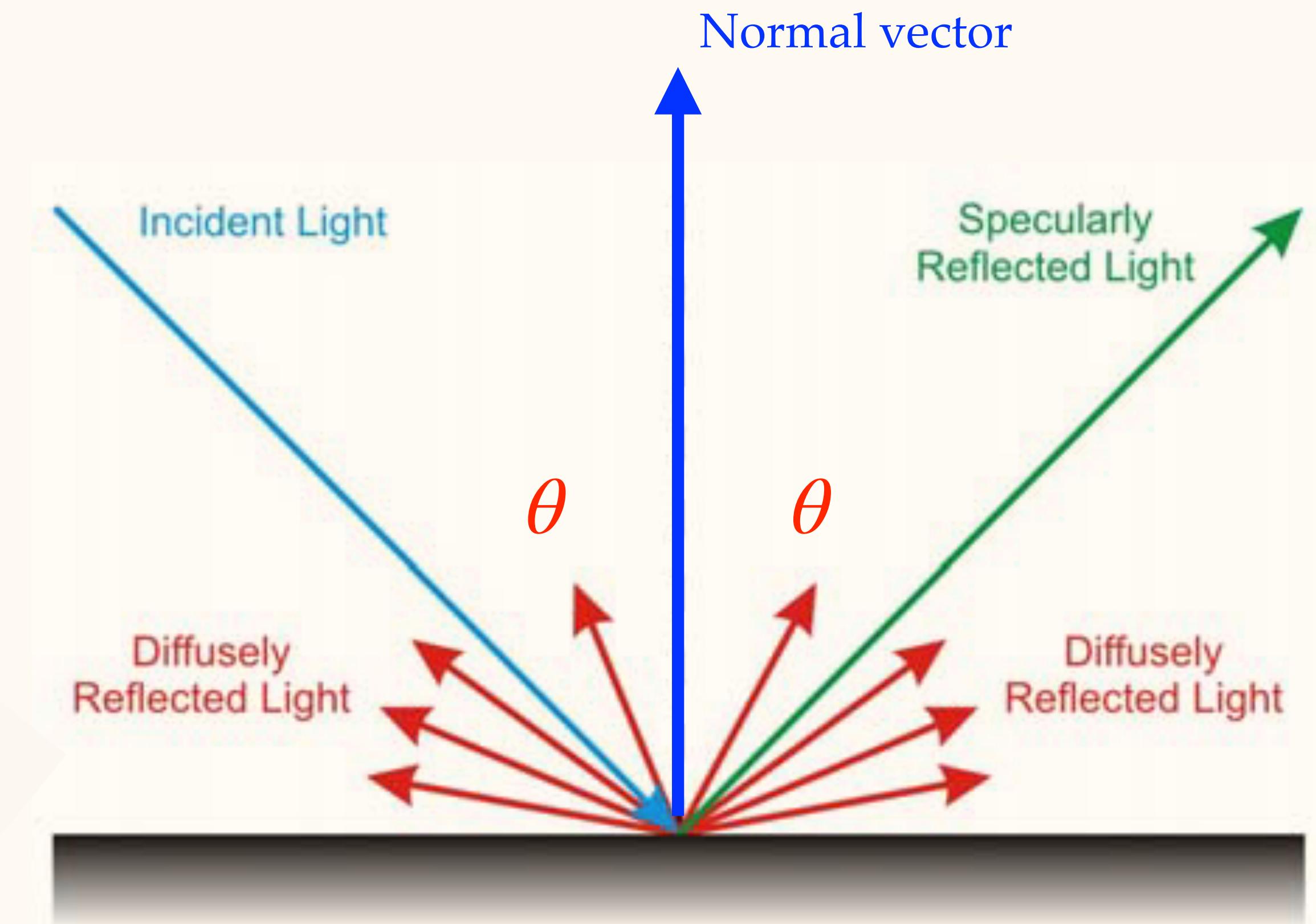
Diffuse Reflection 漫射

- 反射量與光線入射角度有關
 - 直射 vs 斜射
- 與物體表面材質特性有關
 - 光線能量被吸收後，釋放出的能量
 - 能量 = 亮度 (Intensity)
- 與觀察者的位置無關
 - View independent
- 物體的主要顏色
 - “素顏”



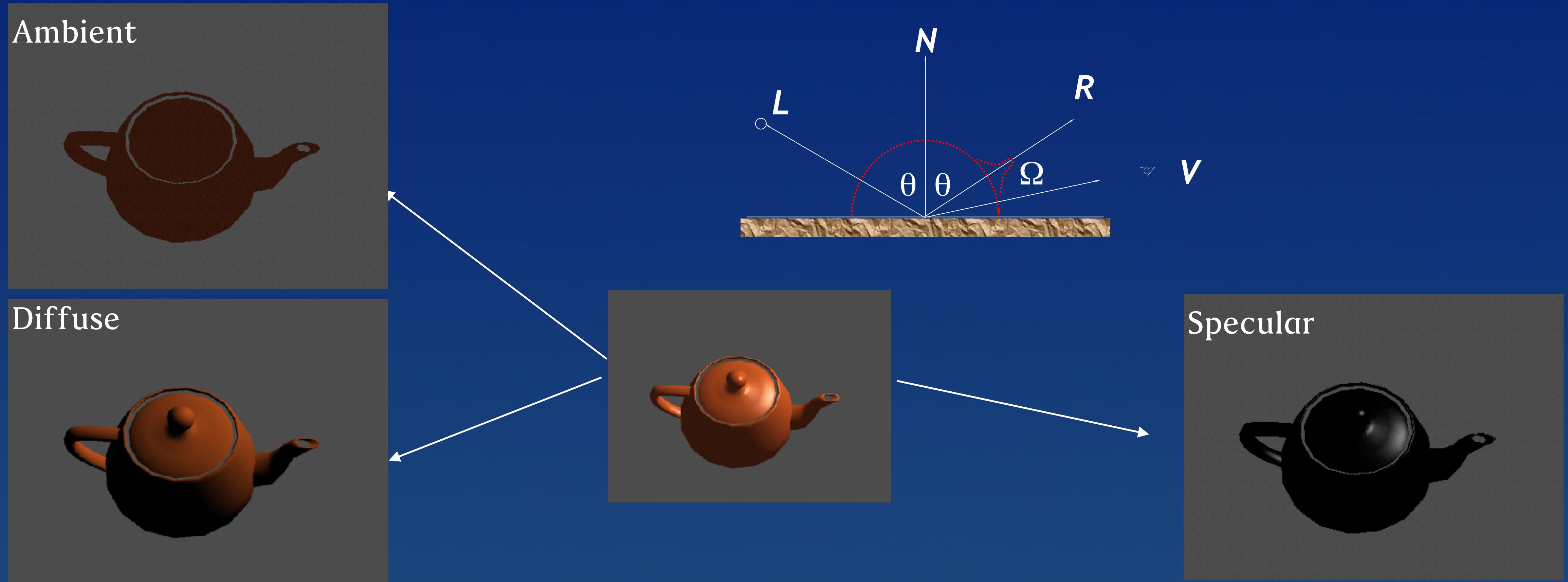
Specular Reflection 鏡面反射

- Ideal specular reflection = mirror reflection
 - 鏡面反射
 - 光源在物體表面上呈現的虛像
 - 是一種光線對物體表面的物理作用
 - 光子打在物體表面彈開
 - 表面粗糙程度會影響反射的好壞
 - 與觀察者的位置有關
 - View dependent



Phong Reflection Model

Phong Reflection Model



- 1975 By Mr. Bui-Tuong Phong

Phong Reflection Model - Diffuse

- Diffuse 漫射
 - “Ideal Diffuse”
 - 有效的直接光照

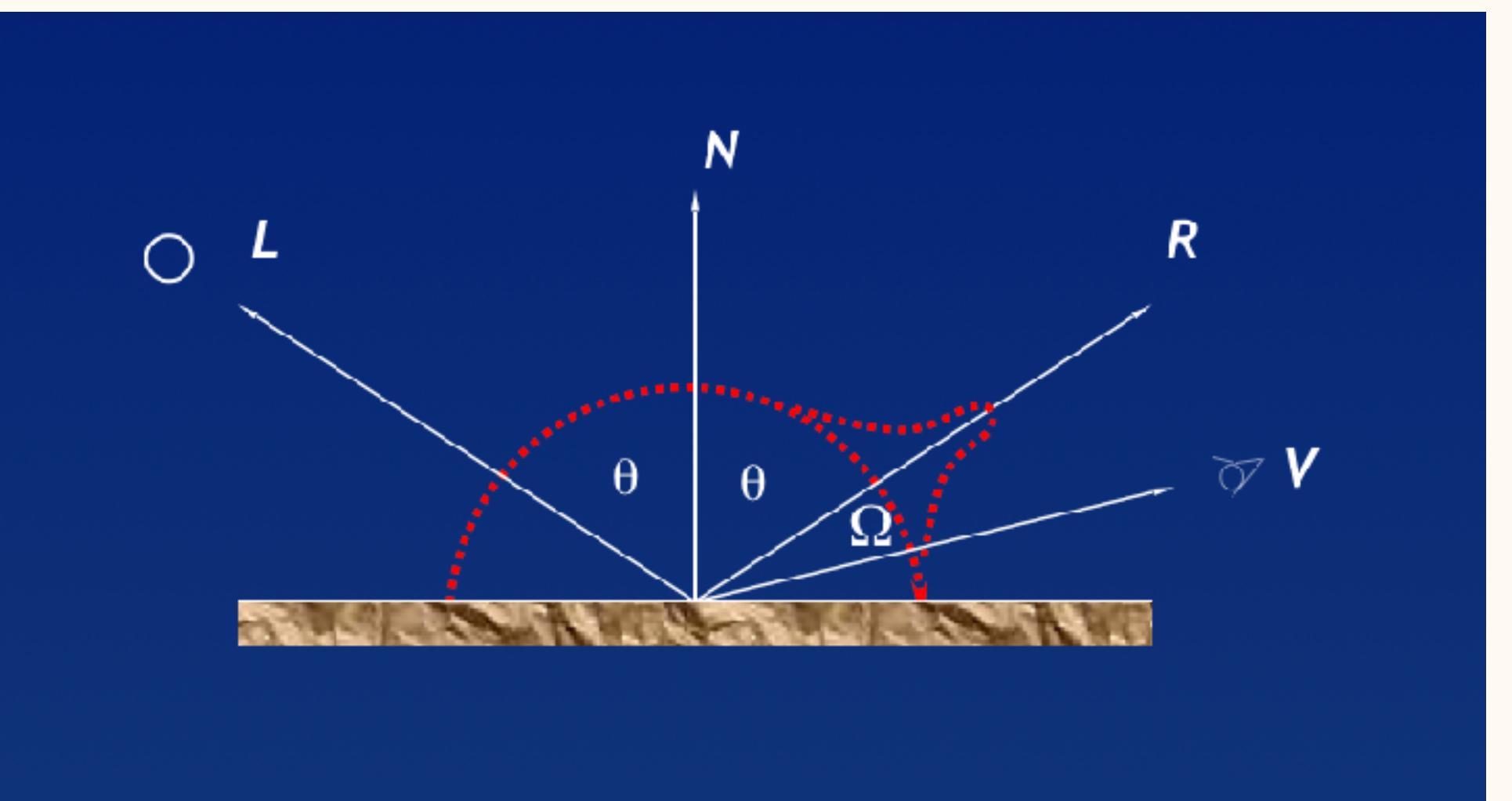
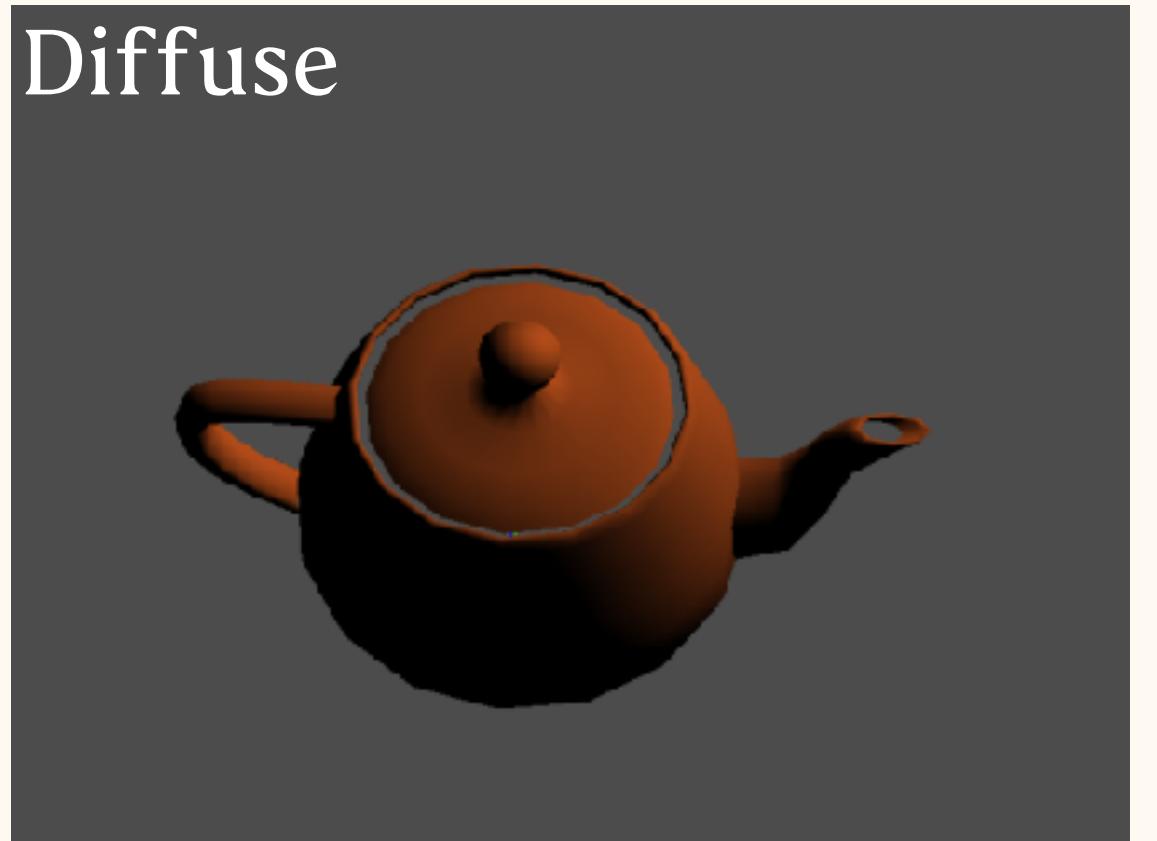
I_i 光源的顏色亮度 (r, g, b)

K_d 物件的材質 diffuse 顏色 (r, g, b)

N 渲染位置上的法向量

L 渲染位置到光源的單位向量

$$\text{diffuse} = K_d I_i \cos\theta = K_d I_i (N \cdot L)$$



Phong Reflection Model – Specular

- Specular reflection
 - Roughness of object surface = “Shininess”

I_i 光源的顏色亮度 (r, g, b)

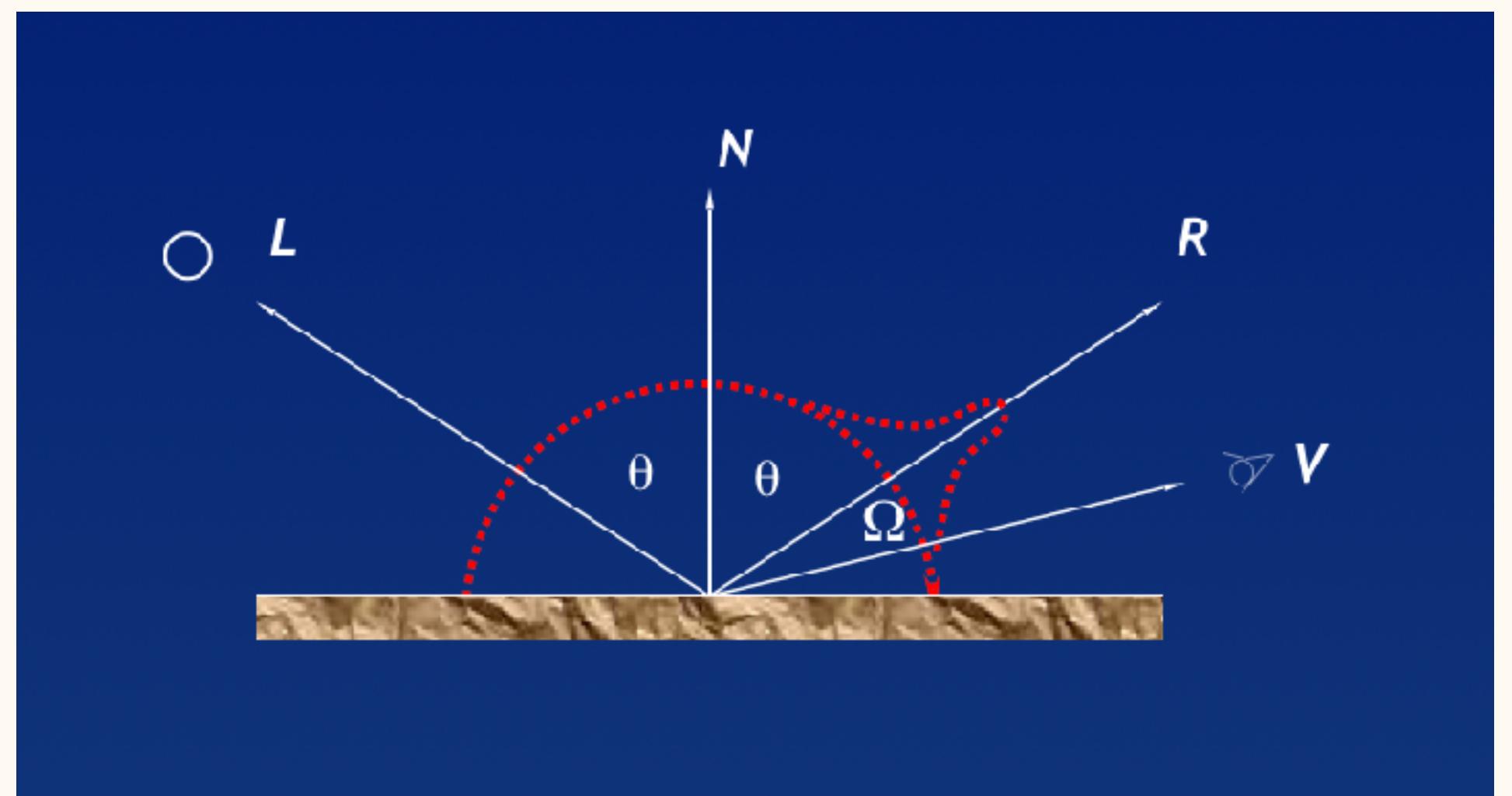
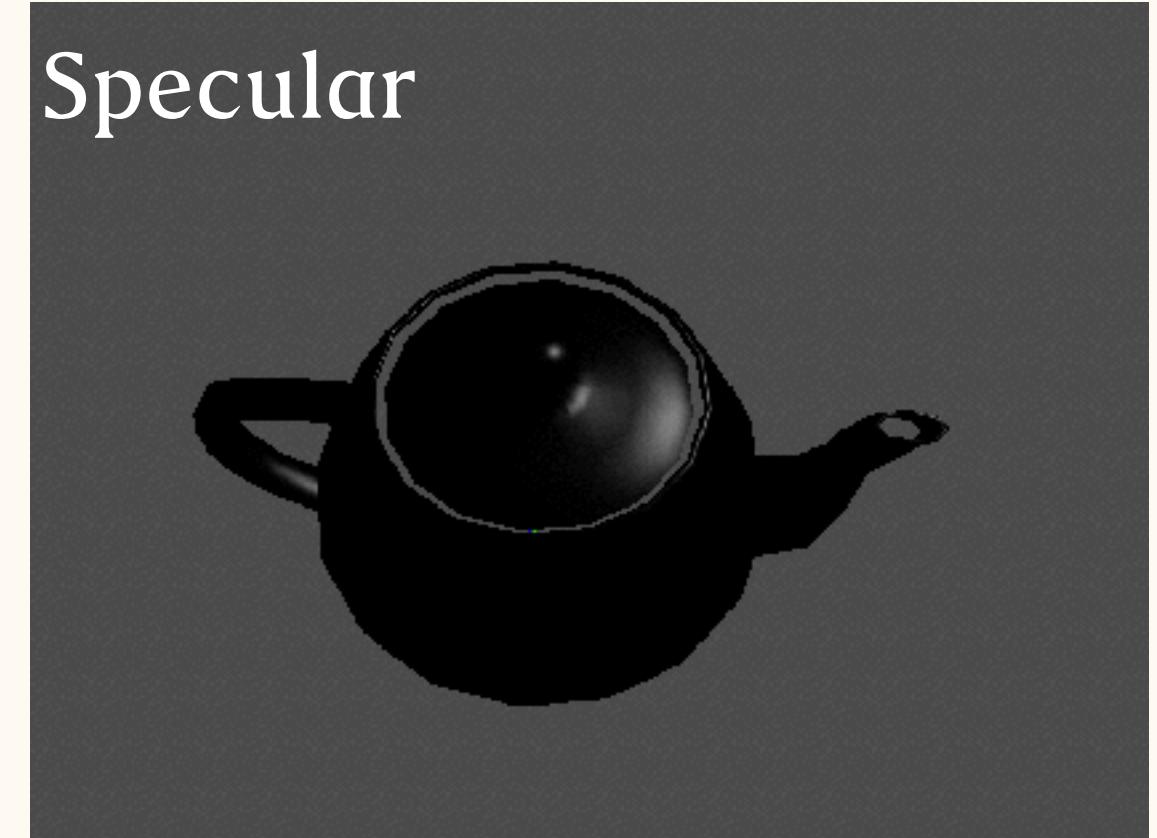
K_s 物件的材質 Specular 顏色 (r, g, b)

V 渲染位置到攝影機的單位向量

R 入射光的反射向量

n 物體表面粗糙度

$$\text{specular} = K_s I_i \cos^n \Omega = K_s I_i (R \cdot V)^n$$



Phong Reflection Model – Ambient

- Ambient
 - 由間接光照形成的反射結果
 - 無法用簡易的數學公式計算
 - Solution :
 - Path Tracing
 - A frame costs many hours/days to render
- Phong 建議設為定值

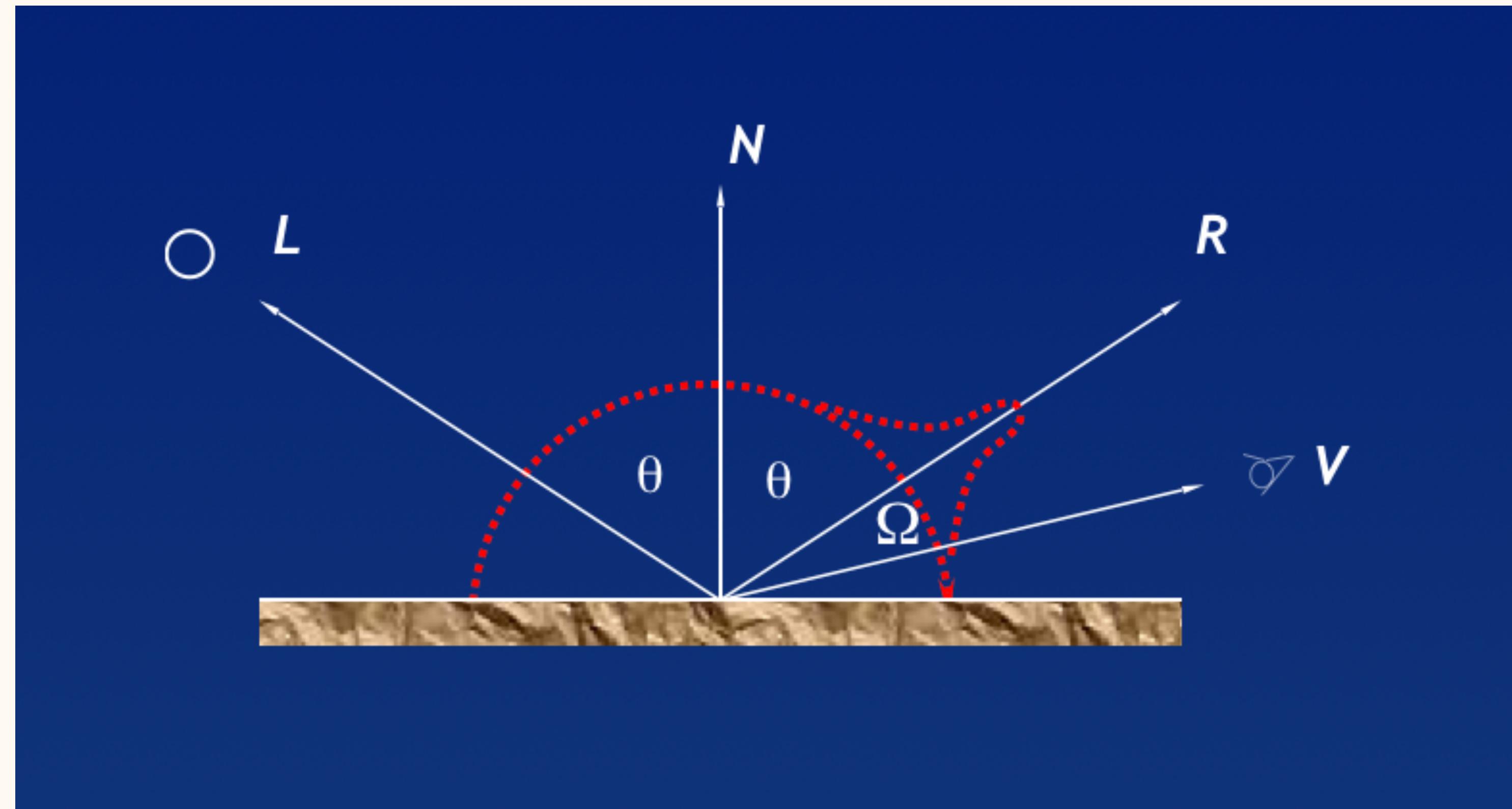
I_a 環境亮度 (r, g, b)

K_a 物件的材質 ambient 顏色 (r, g, b)

$$ambient = K_a I_a$$

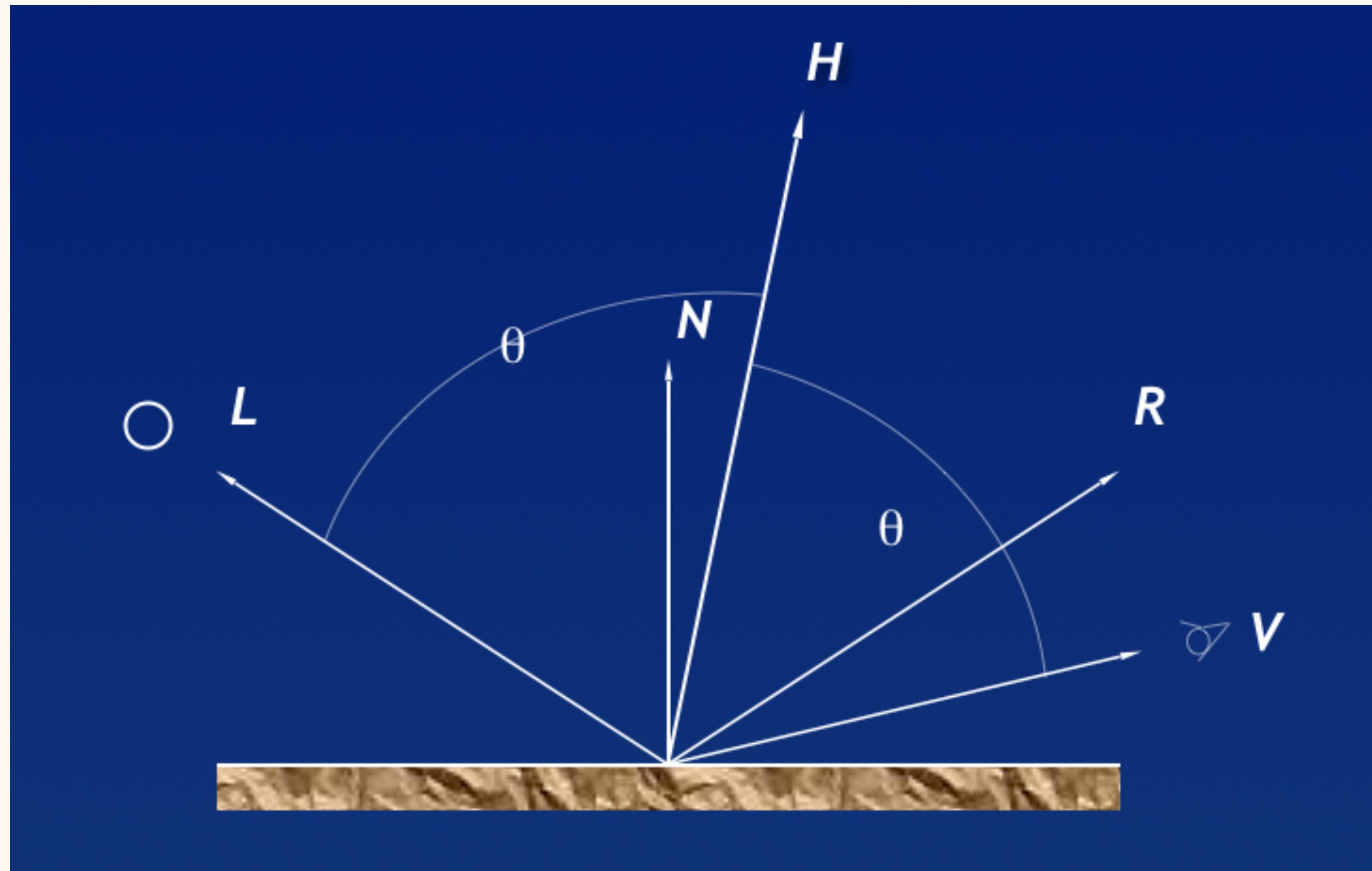


Phong Reflection Model



$$I = I_a K_a + I_i K_d (L \cdot N) + I_i K_s (R \cdot V)^n$$

Phong-Blinn Reflection Model



$$I = I_a K_a + I_i K_d (L \cdot N) + I_i K_s (N \cdot H)^n$$

$$H = (L + V)/2$$

Use this approximation to speed up the lighting calculation for Phong reflection model in real application, especially for real-time 3D rendering application.

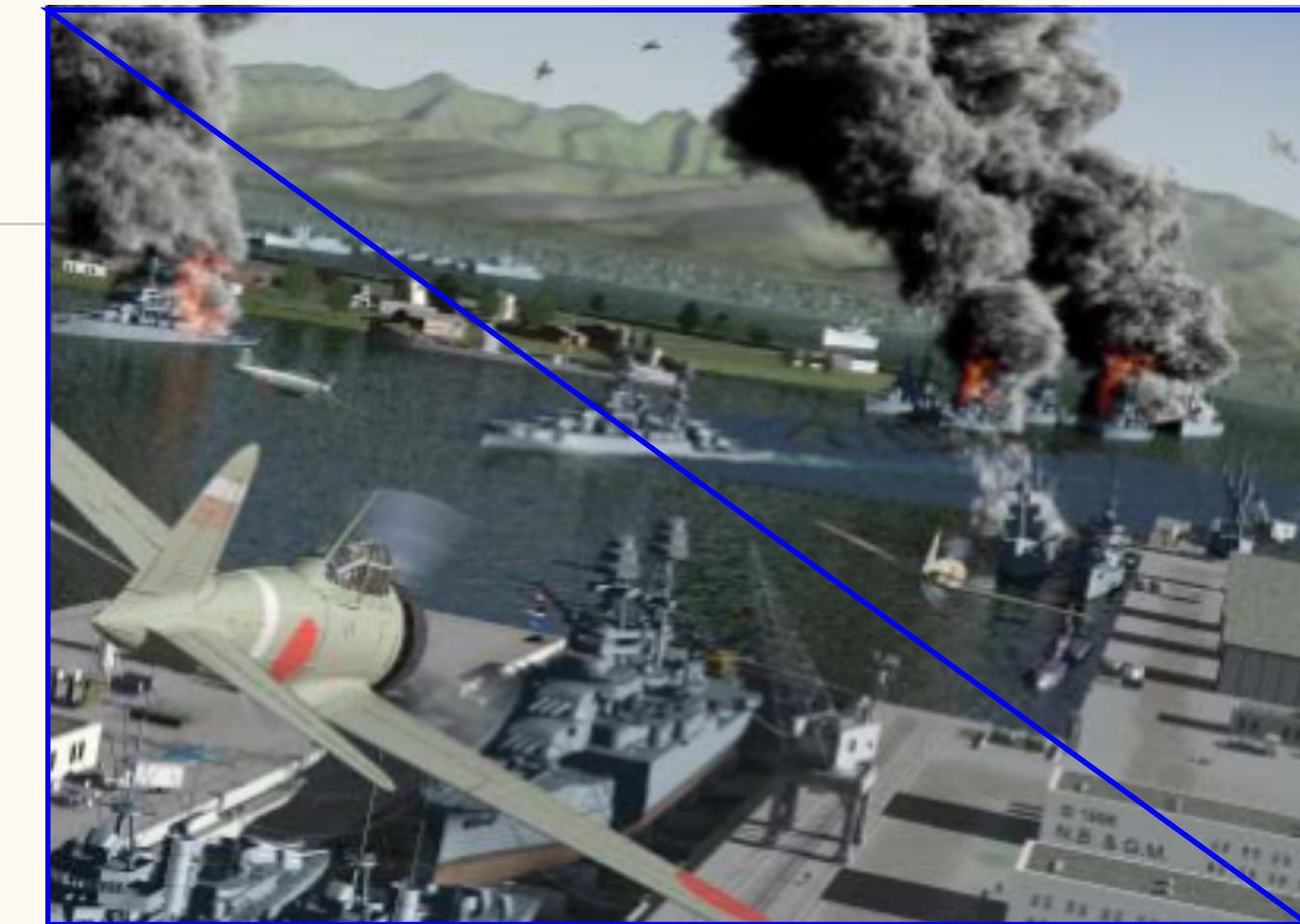
Textures

Textures

- What is a Texture?
- Texture mapping
- Texture animation
- Multi-layer textures and texture blending
- Texture formats
- Cubemap
- Mipmap

Image ? Texture ?

- 2D 遊戲的圖像 = Texture on Two Triangles
- A Texture = An Image + Filtering + Addressing
 - Texture sampling = fragment
 - Texture surface data (raw image data)
 - Filtering
 - The way to sample texture pixels (texels) for screen pixels (pixels)
 - “Texel” is the pixel in a texture
 - Addressing
 - The way to map texture on geometry



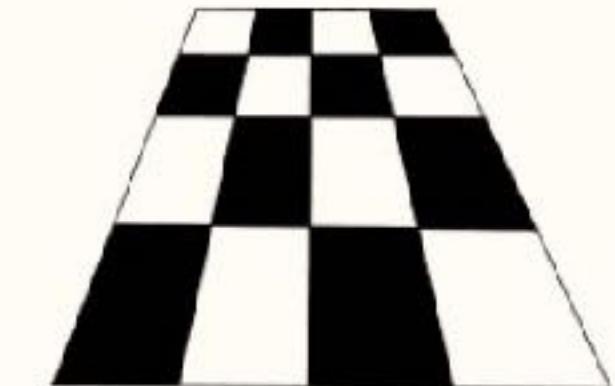
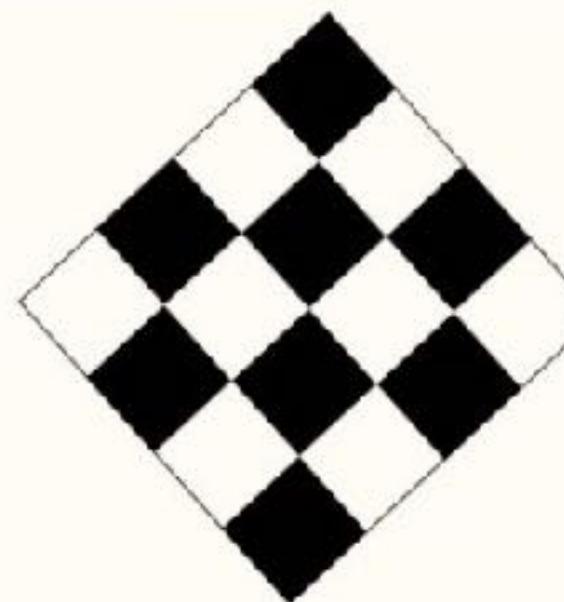
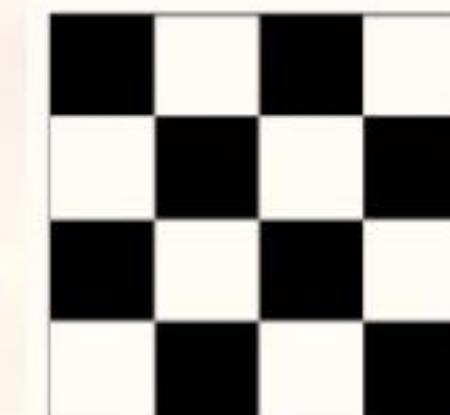
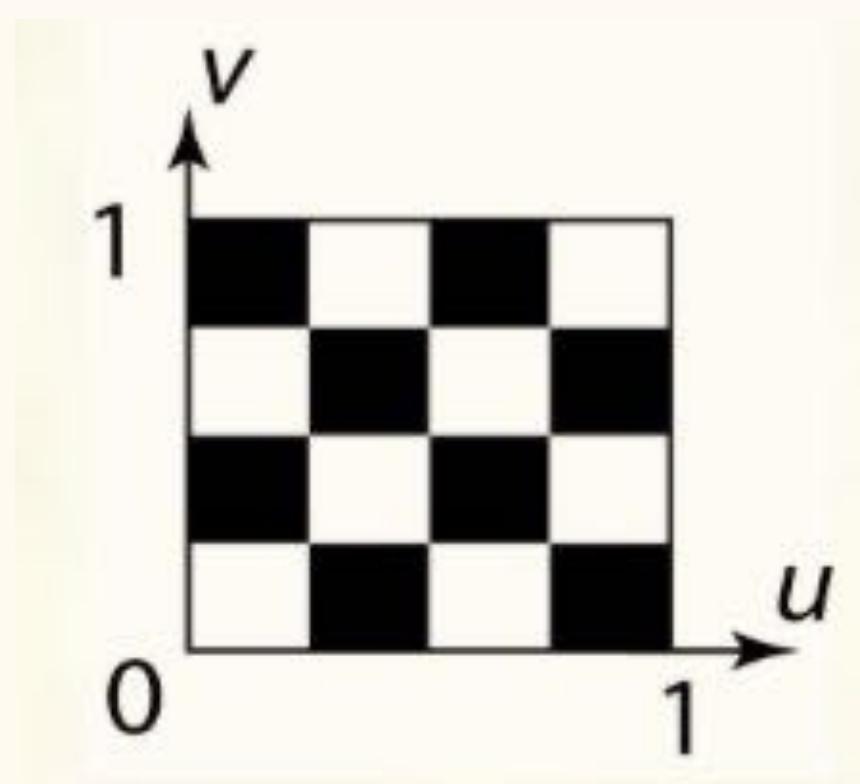
Color Textures

- 2D color image + addressing + filtering
- Colormap
- Albedo map
 - Color + ambient occlusion
- Decal
 - Color + color keying



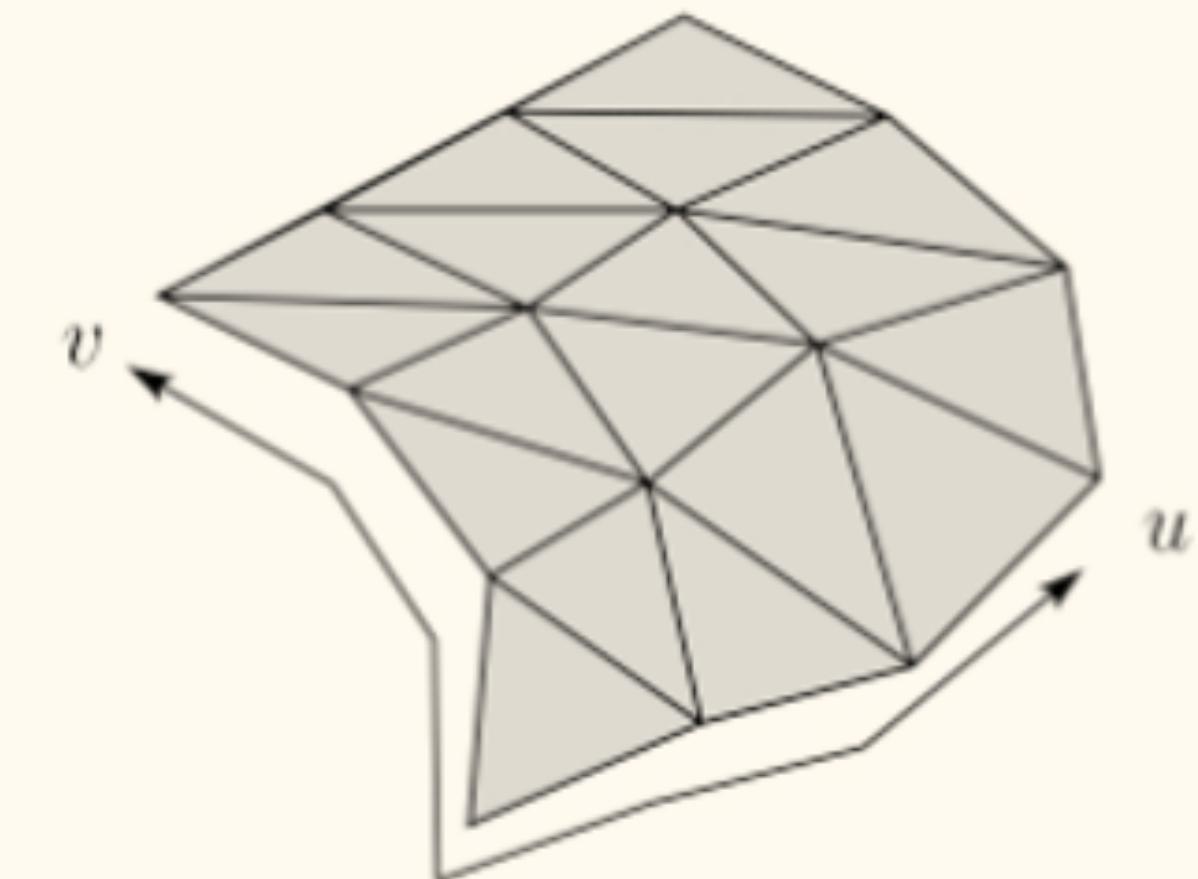
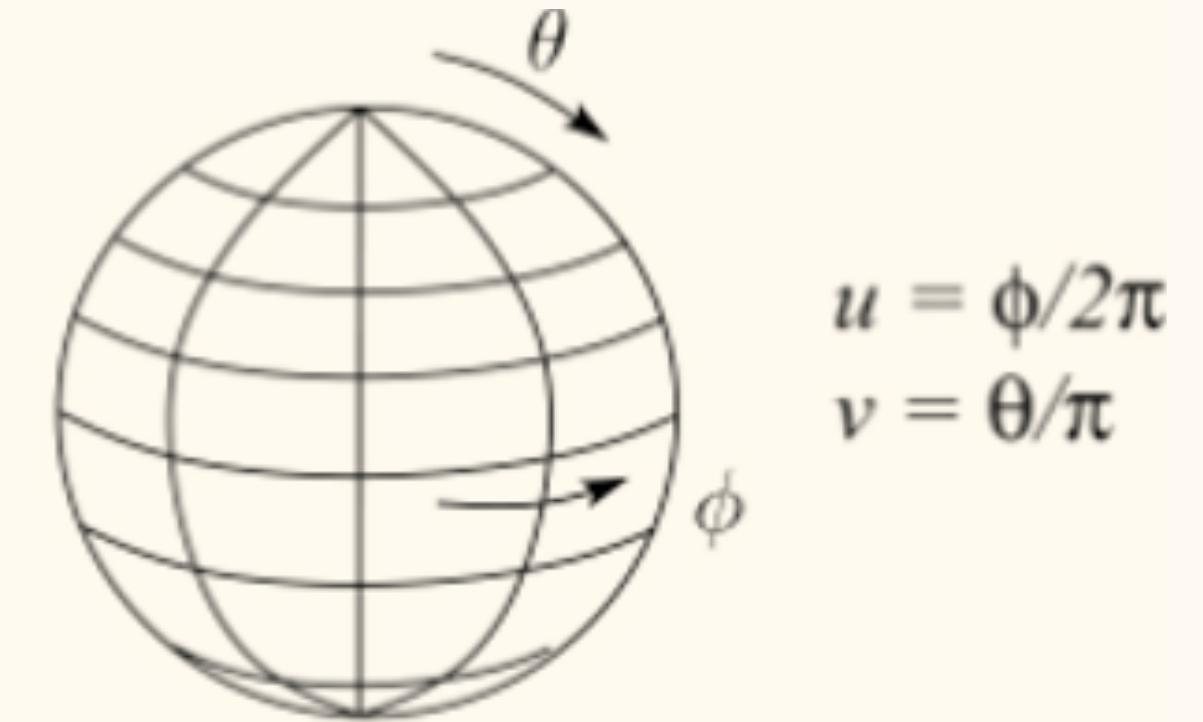
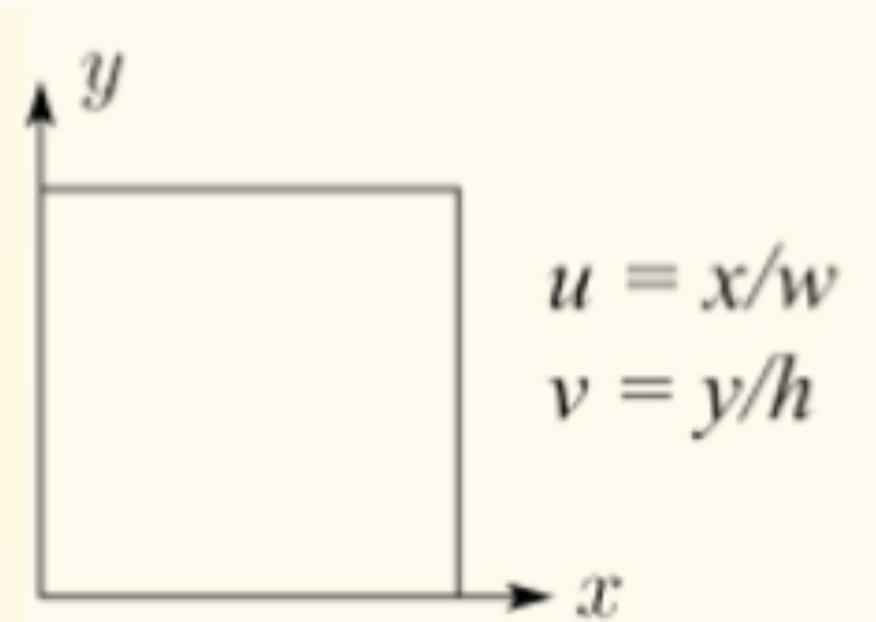
Texture Mapping

- Texture mapping 可以讓簡單的 3D 模型具有複雜且多樣的外表 (圖案)
 - 源於 Ed Catmull, PhD thesis, 1974
 - 完備於 Blinn & Newell, 1976
- 3D 模型上需要有 Texture coordinate (u, v) 貼圖座標來標定圖像
 - Texture space 貼圖座標系
 - (u, v) in the range of $([0..1], [0..1])$

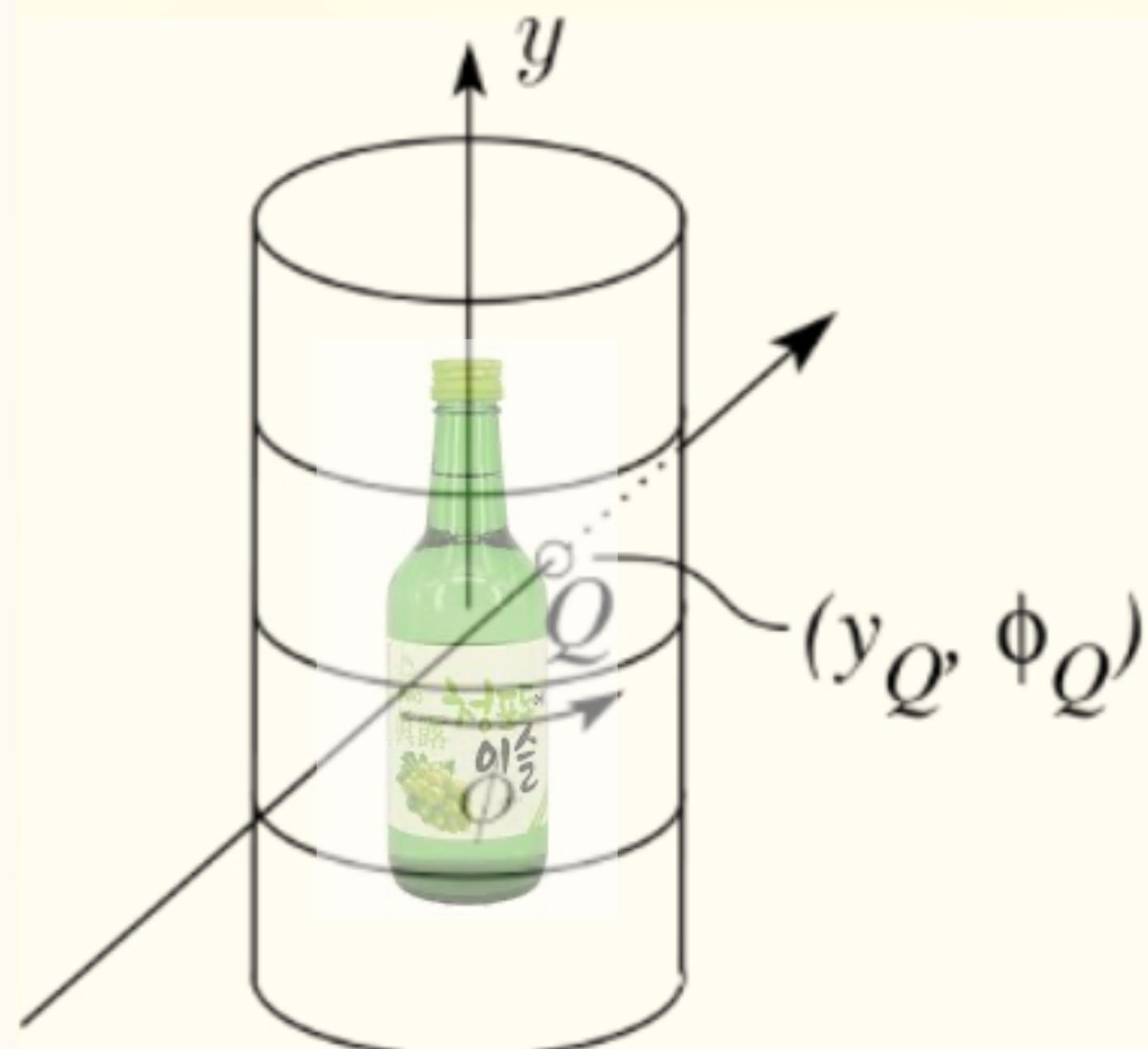


Texture Mapping

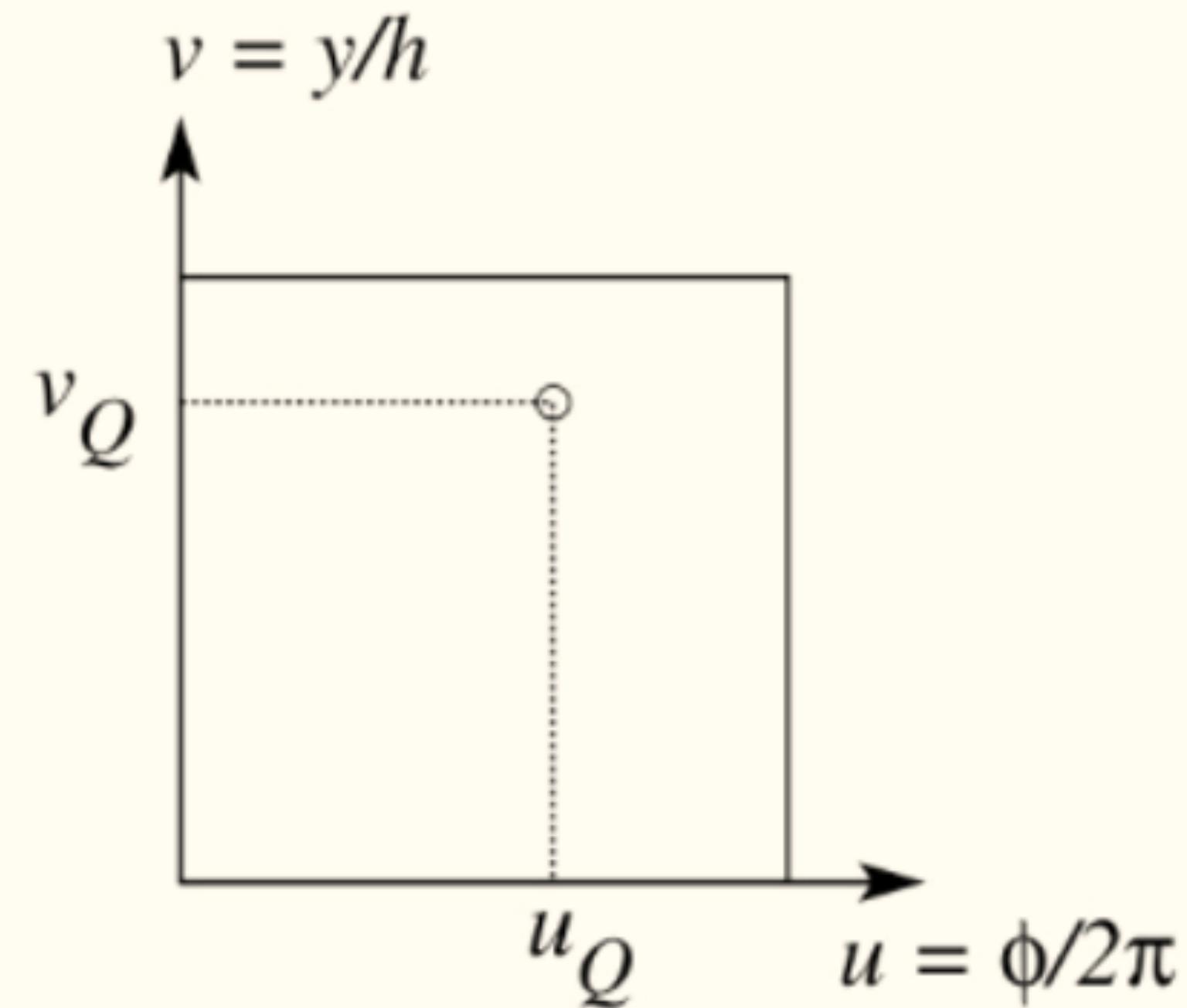
- 貼圖的動作通常需要中介的幾何模型來協助
- 常用的中介模型有：
 - 平面
 - 球
 - 圓柱體



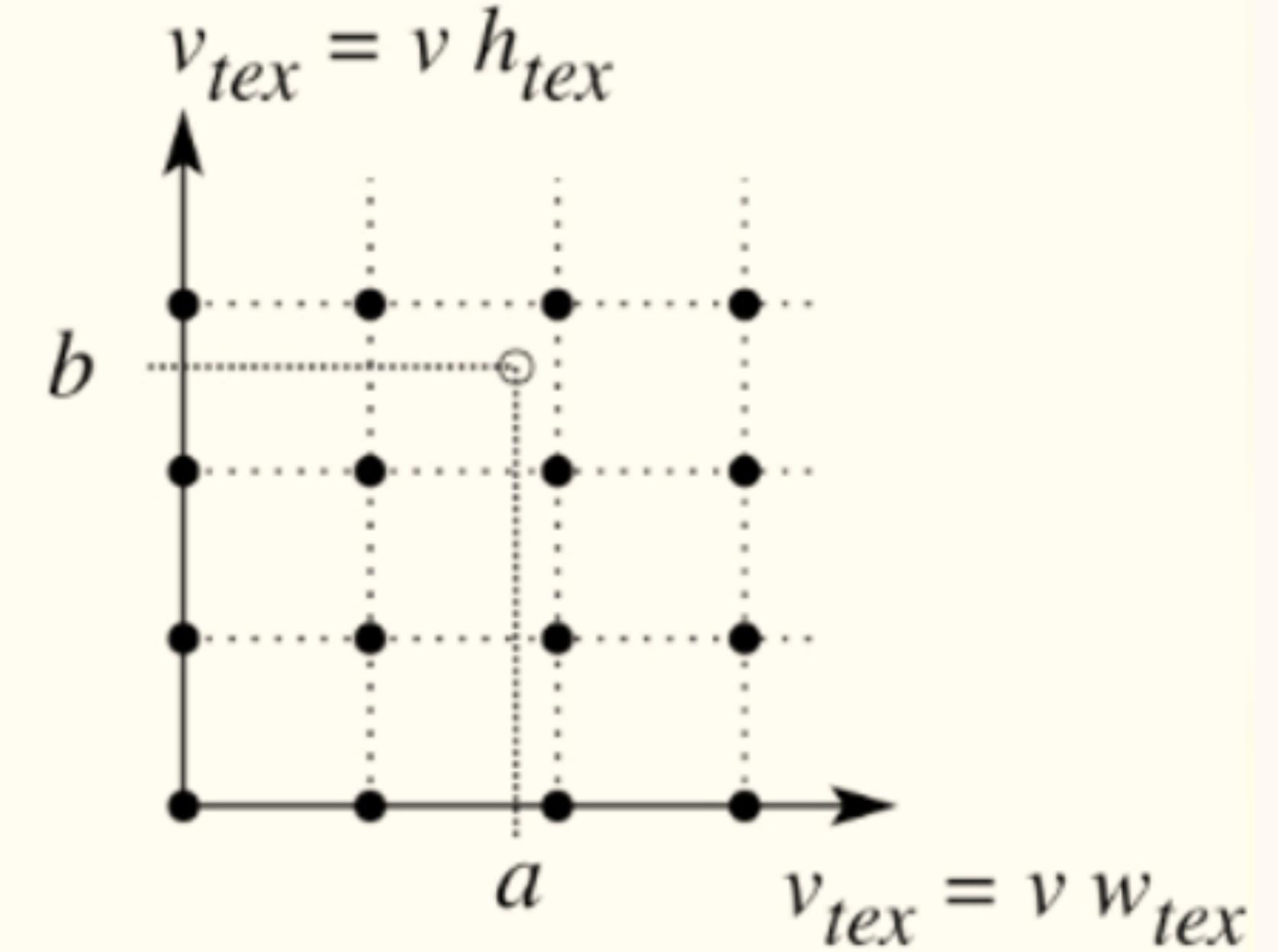
Texture Mapping



Ray intersection

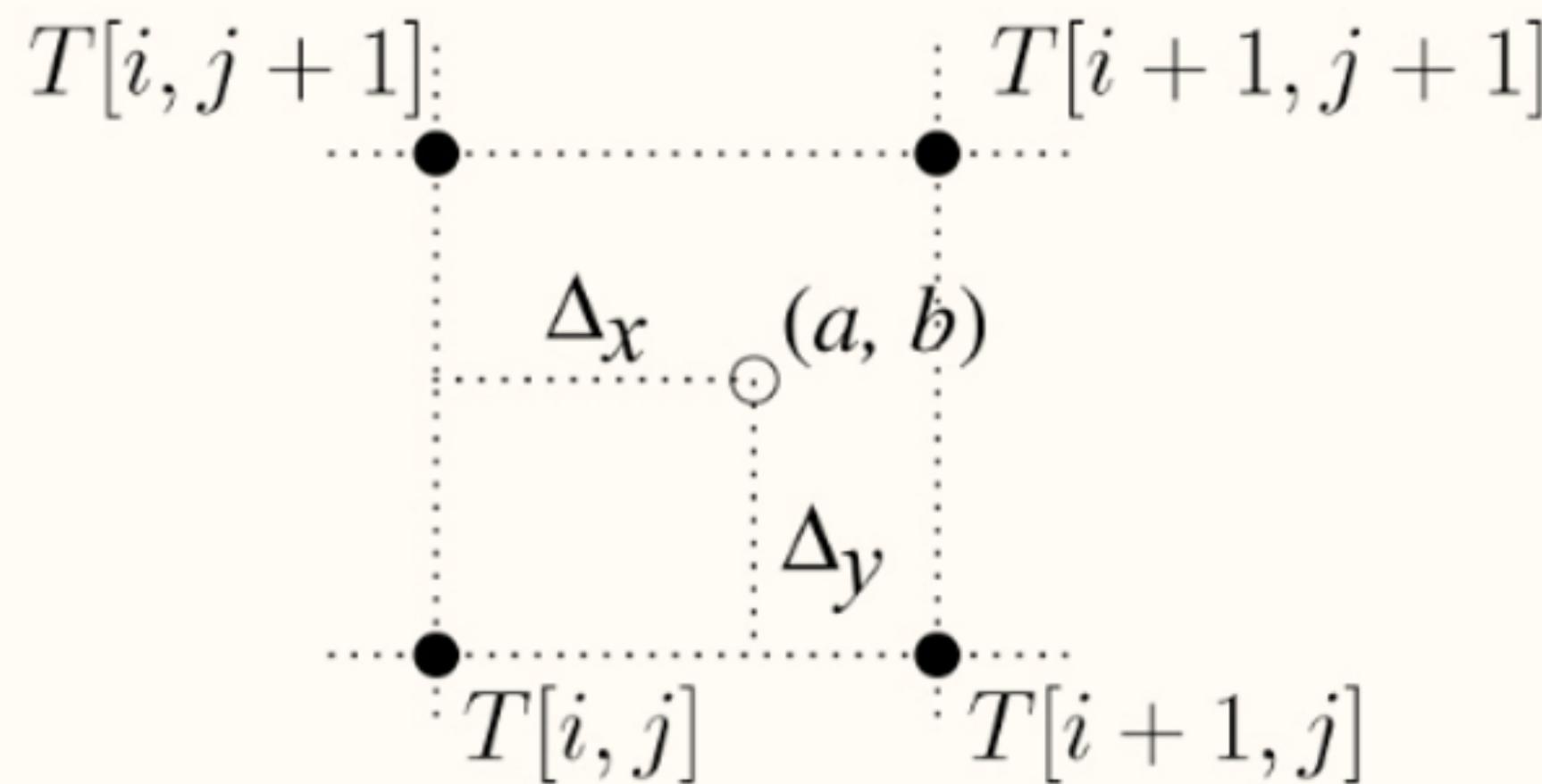


Mapping to
abstract texture coords



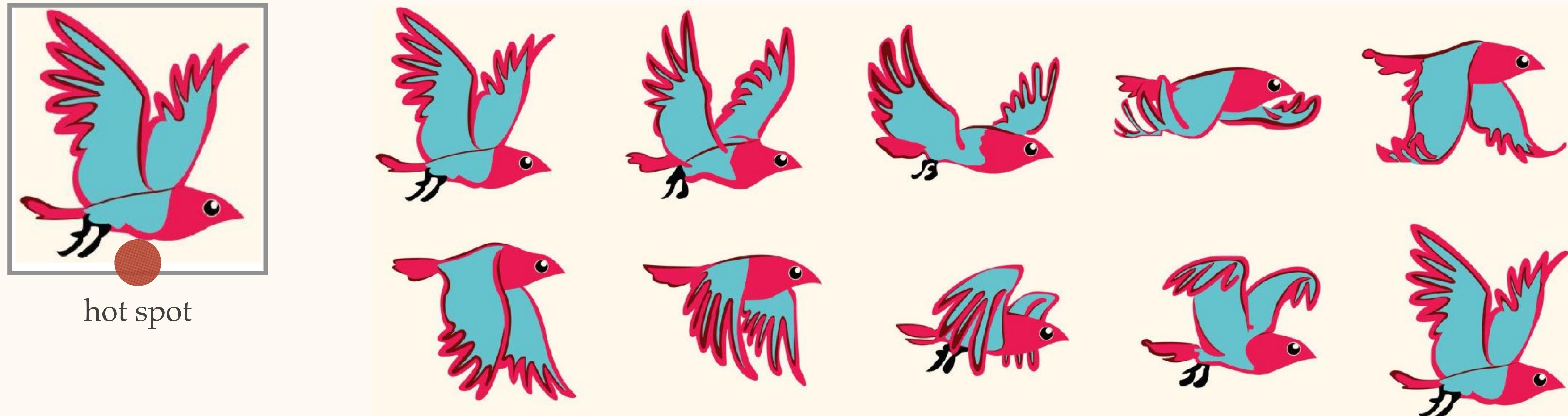
Mapping to
texture pixel coords

Texture Resampling



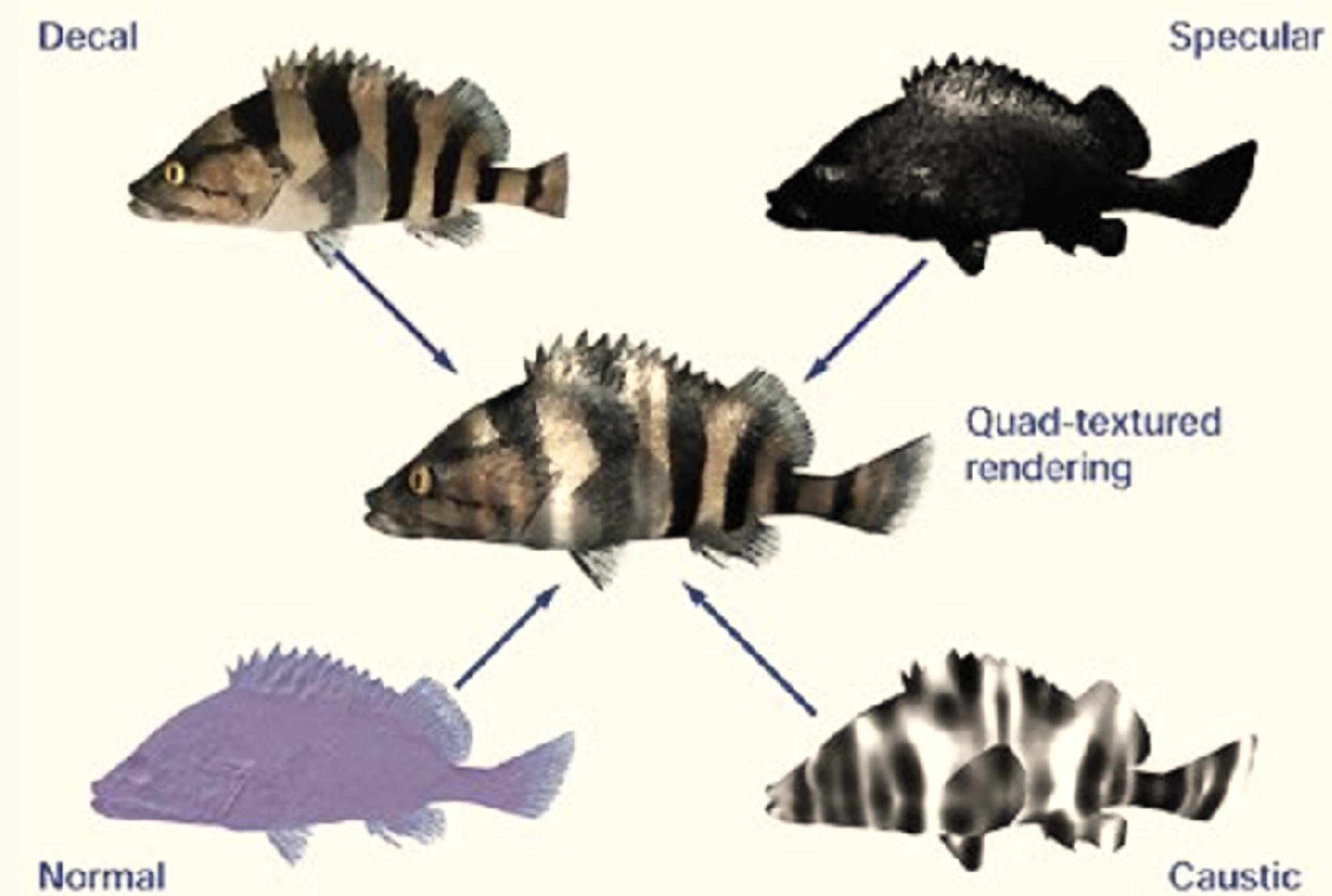
$$\begin{aligned}T(a,b) &= T[i + \Delta_x, j + \Delta_y] \\&= (1 - \Delta_x)(1 - \Delta_y)T[i, j] + \Delta_x(1 - \Delta_y)T[i + 1, j] \\&\quad + (1 - \Delta_x)\Delta_yT[i, j + 1] + \Delta_x\Delta_yT[i + 1, j + 1]\end{aligned}$$

Texture Animation on a Sprite



Multi-layer Textures

- Multi-layer textures 在遊戲應用上非常普遍
- 現代硬體支援多層貼圖架構
- Texture blending
- 在 Pixel Shader 中實作



Texture Data Formats

- Traditional texture data formats
 - 2的幂次方大小

$$2^m \times 2^n$$

- 顏色數值是整數
 - (r, g, b) 分別為 8-bit 整數 = 0~255 color value
 - Black = (0, 0, 0)
 - White = (255, 255, 255)

Texture Data Formats

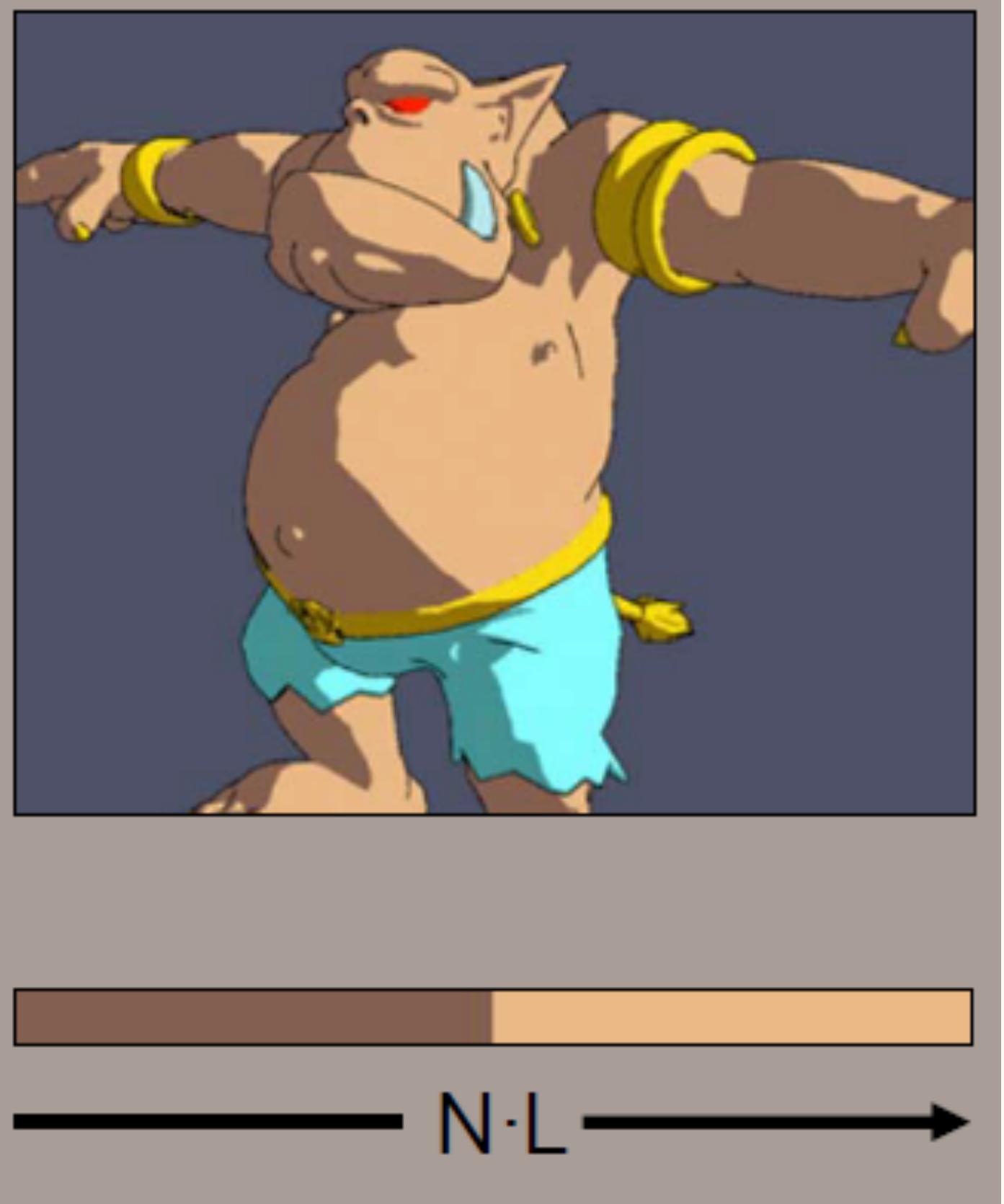
- Modern texture data formats
 - floating-point format is commonly used.
 - HDR
 - 16-bit floating-point: half
 - 32-bit floating-point: IEEE 754 standard
 - Any size is possible.
 - But 2^n -size for compressed texture and MIPMAP
 - MS DDS format (.dds)
 - DXT1, DXT3, DXT5 compression

Texture Types

- 1D textures
- 2D textures
- Volume textures (3D textures)
- Cubemap

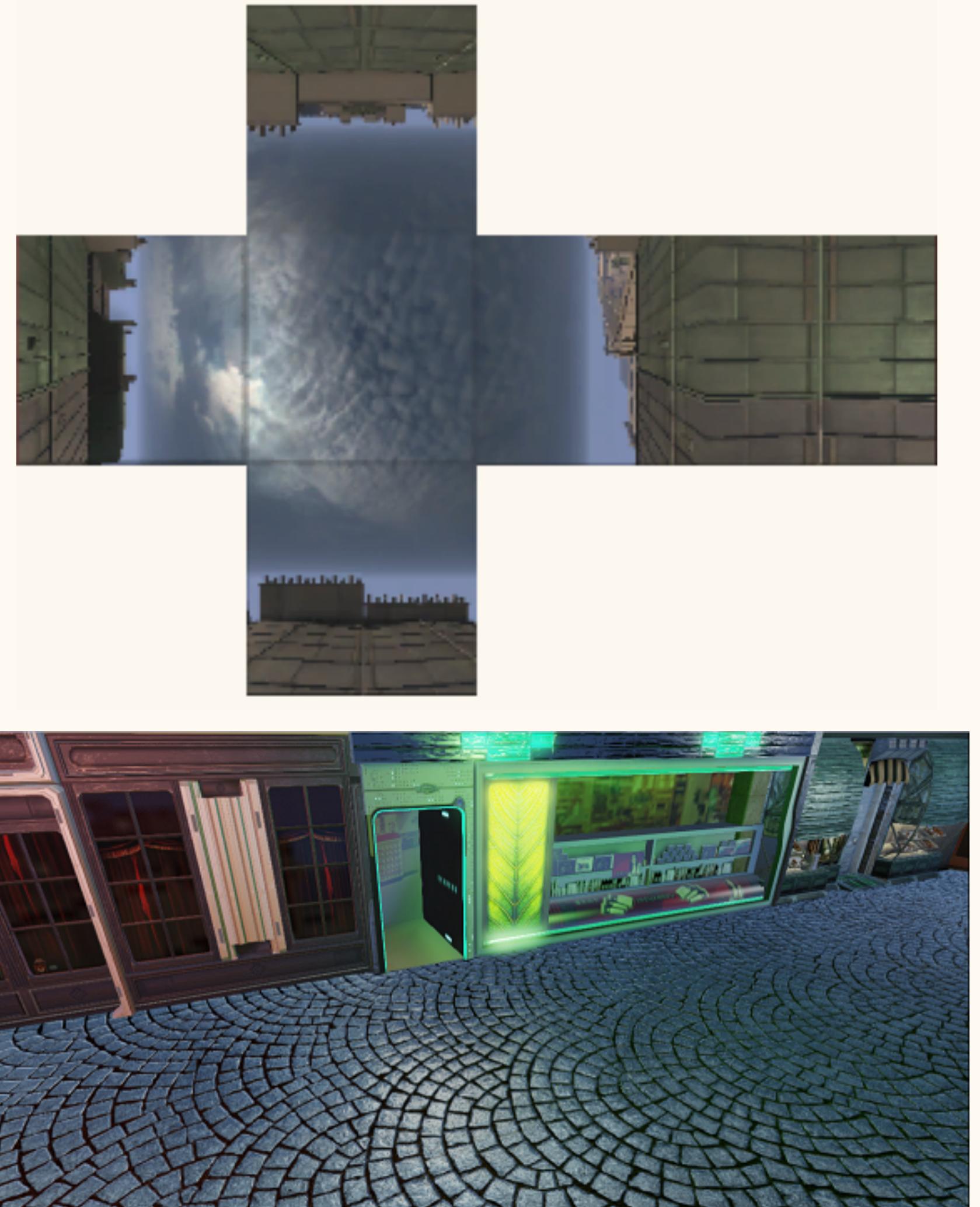
1D/2D/3D Textures

- Texture coordinates: (u, v, w)
- 我們在常用的是 2D texture
 - (u, v)
- 3D texture = volume texture
 - Textures in layer
 - 電腦斷層掃描 (CT Scan) 影像
- 1D texture 常用在 lookup table

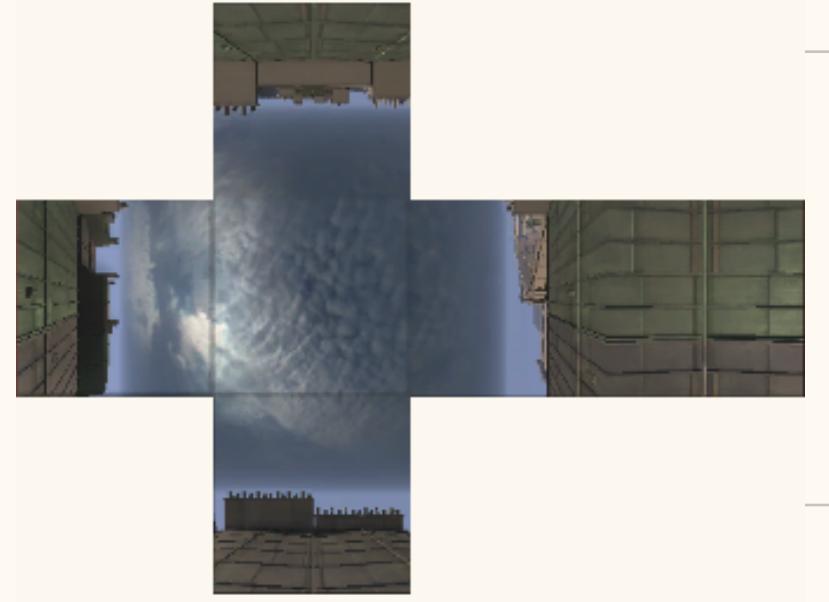


Panorama Images 全景圖像

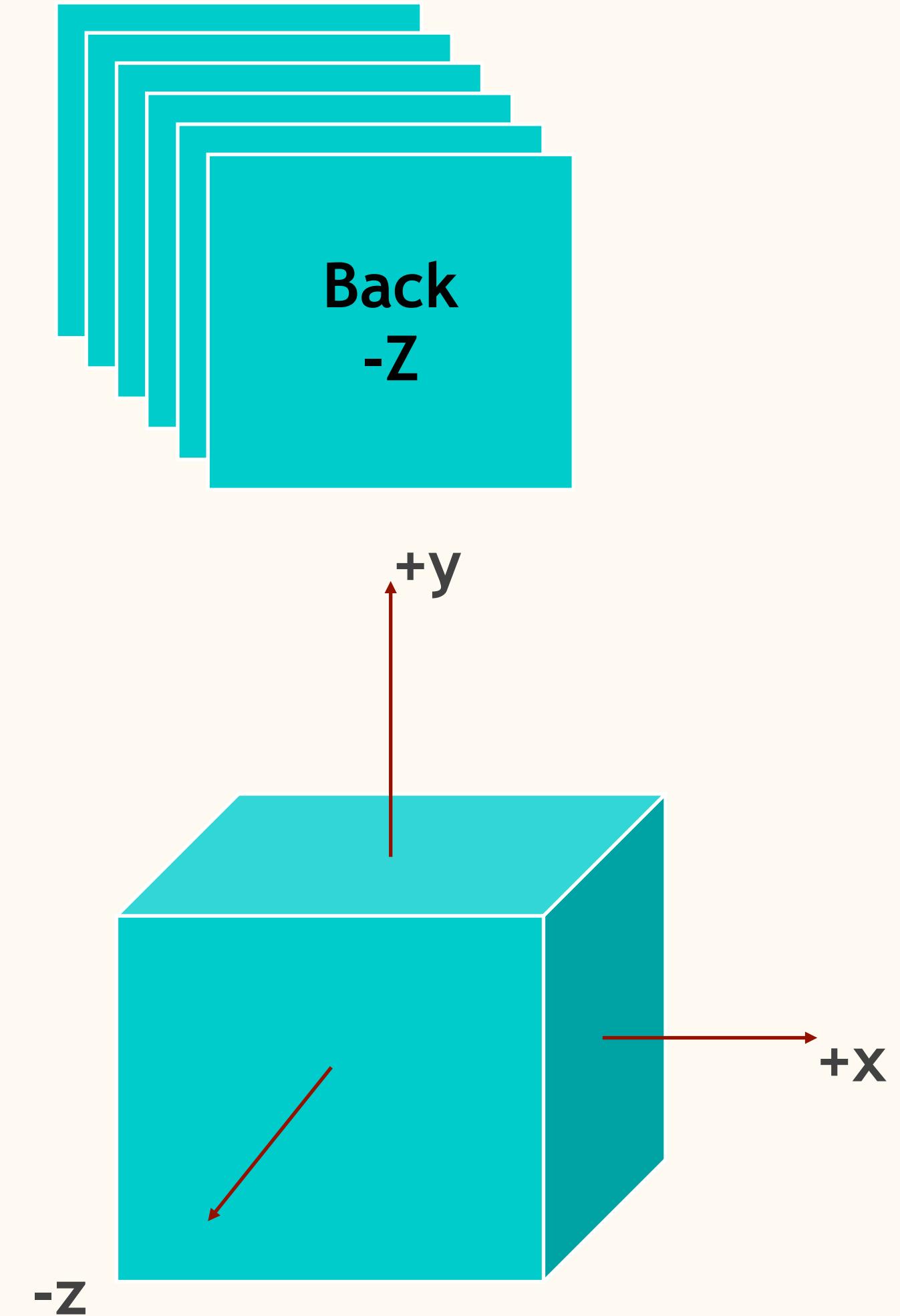
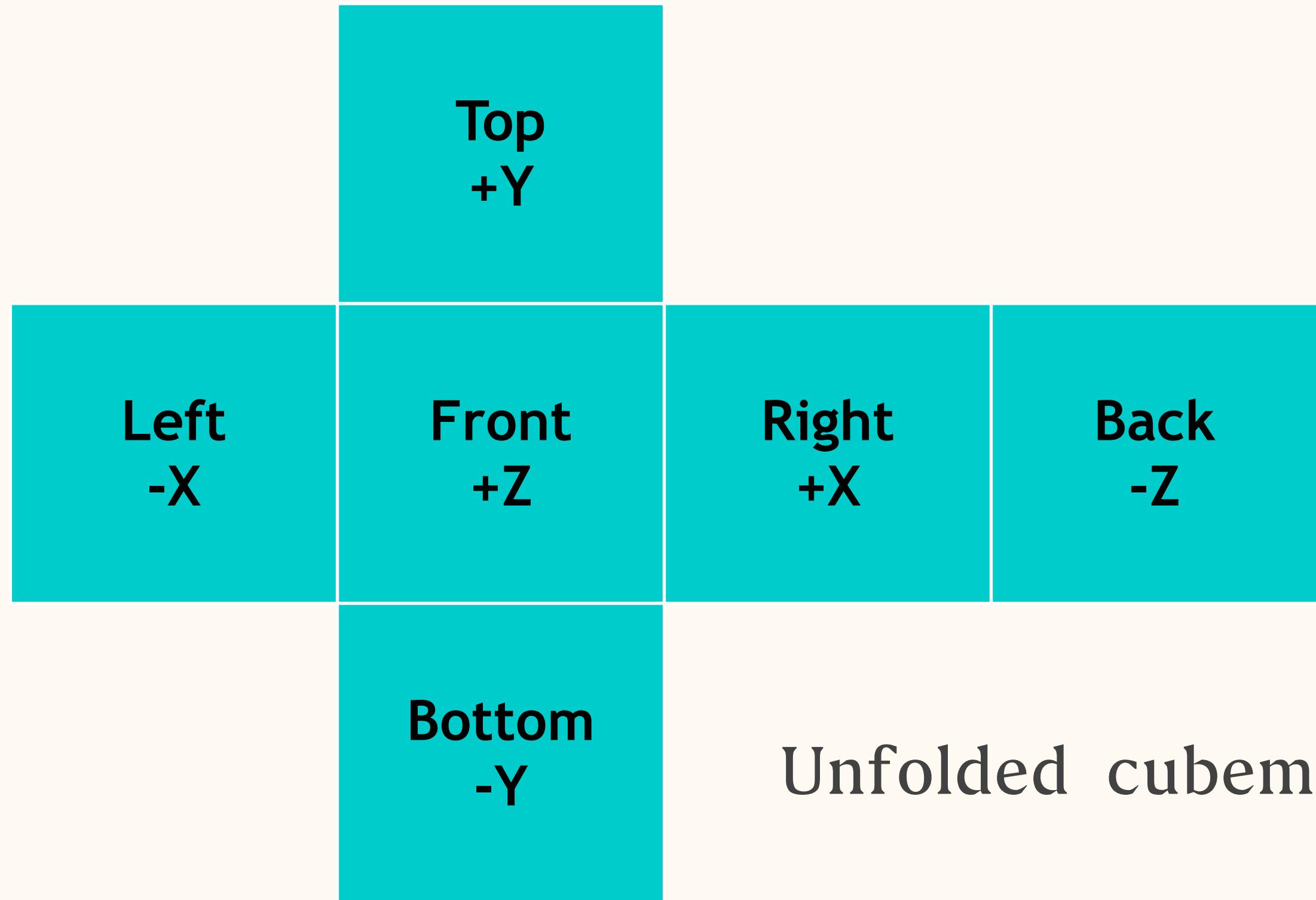
- Panorama mapping solutions :
 - Cubemap (most used in games)
 - Sky Box
 - Light probe (光探頭)
 - 經緯圖 (Latitude-longitude map, LL Map)
 - 常見於 VR360 攝影機
 - 360° panorama







Cubemap



Environment Mapping using Cubemap



環境貼圖
反射特效

Mipmap

- Mipmaps (also MIP maps) 是一組預先計算且最佳化的圖像序列，存放在一種特定的資料結構中
- 每張圖像都是由其前一張圖像依序縮圖來的
- 縮圖的大小是依照2的幕次方
- 優點：
 - 加速渲染速度
 - 減少高頻影像鋸齒現象



Physically-based Materials

Physically-based Materials

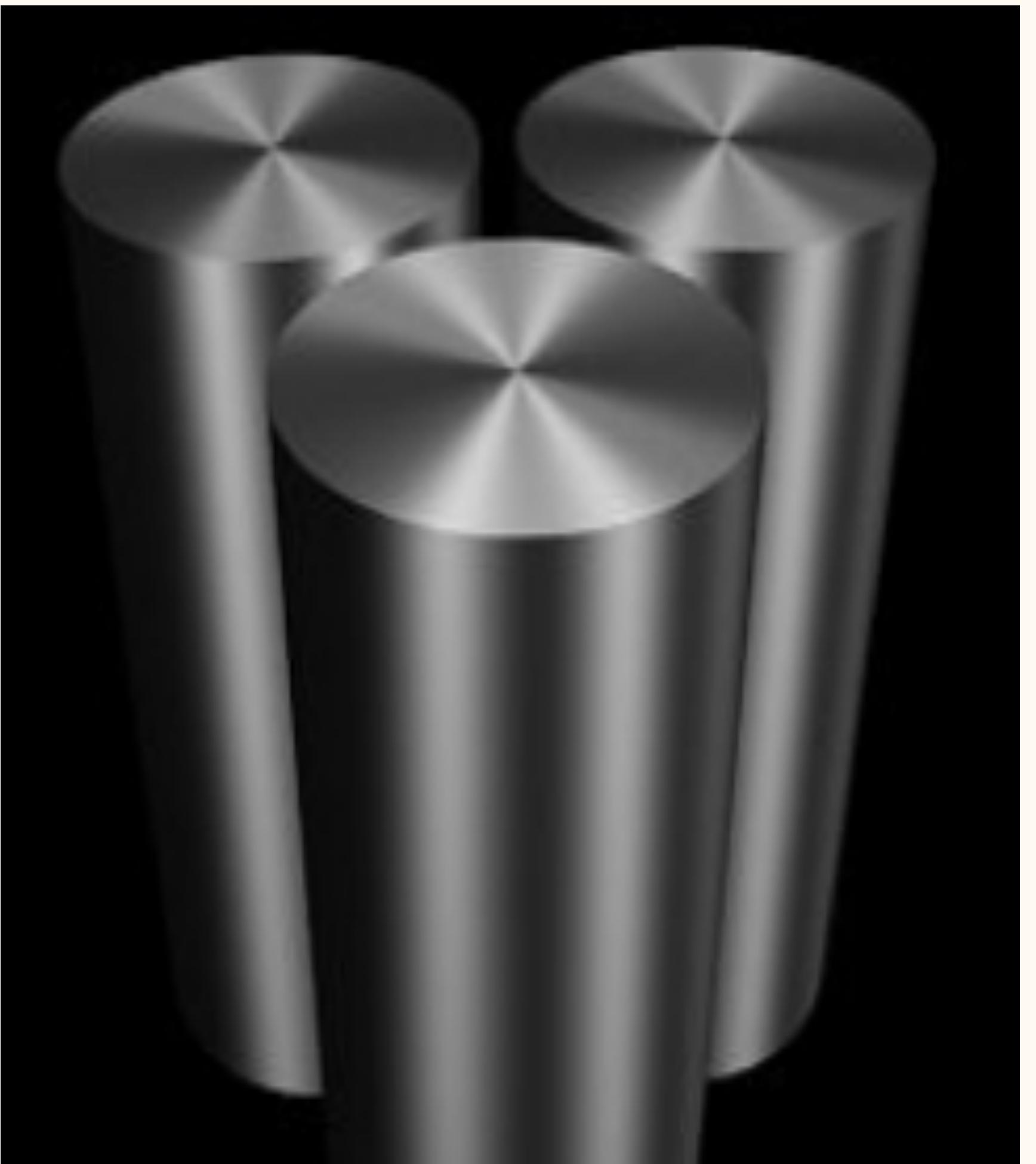
- 基於光現在真實世界行進的模型所推論的材質模型
- 三種常見的模型：
 - 兩種屬於 isotropic model (等向性材質)
 - Cook-Torrance Reflection Model (1981)
 - Oren-Nayar Reflection (1992-1994)
 - 一種屬於 anisotropic model (非等向性材質)
 - Ashikhmin-Shirley Reflection Model (2000)
- Physically-based rendering (PBR)

Isotropic Materials

- Invariant under a rotation with the normal vector
 - The angle between V & L
 - The angles between V, L & N
 - The distribution of microfacets
- 一般常見的材質均屬之

Anisotropic Materials

- Variant under a rotation with the normal vector
 - The angle between V & L
 - The angles between V, L & N
 - The distribution of microfacets
- 例如：
 - 金屬光澤反射
 - 頭髮
 - CD
 - 特殊纖維

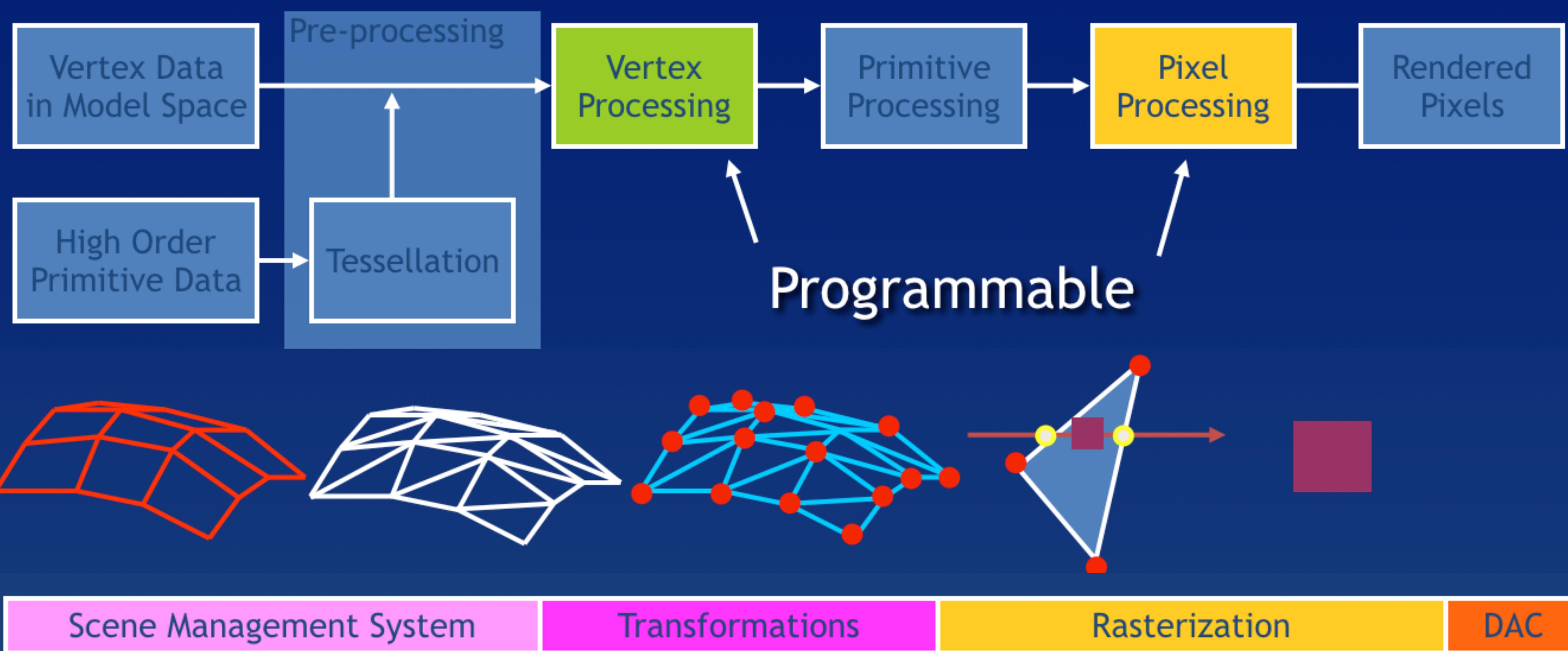


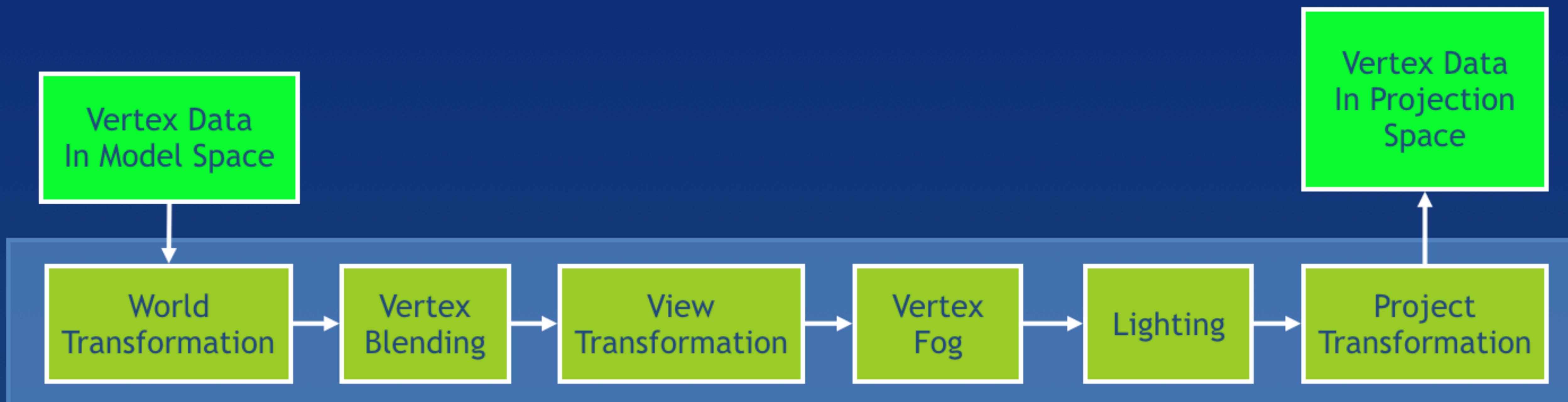
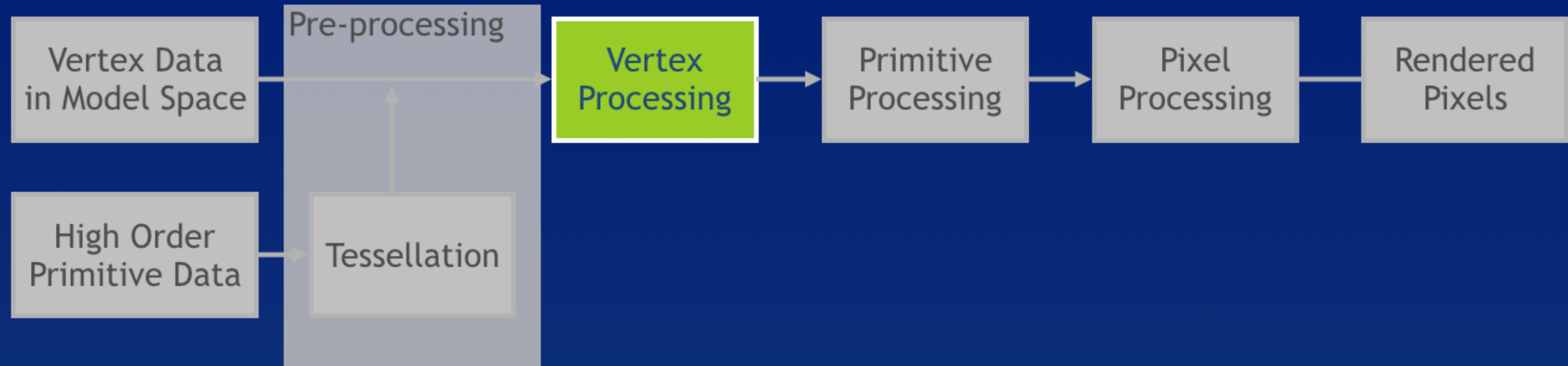
Real-time 3D Rendering Pipeline

Real-time 3D Rendering Pipeline

- 先以 Shader Model 3.0 (DirectX 9) 為基礎來介紹 GPU-based 3D 渲染流程
 - Vertex Processing -> Primitive Processing -> Pixel Processing
 - 95% 以上的 Shaders
- 再介紹以 Shader Model 5.0 (DirectX 11) 的渲染流程
 - 整合 Geometry Shader、Hull Shader、Tessellator、Domain Shader

DX9 Programmable Rendering Pipeline





Vertex Processing

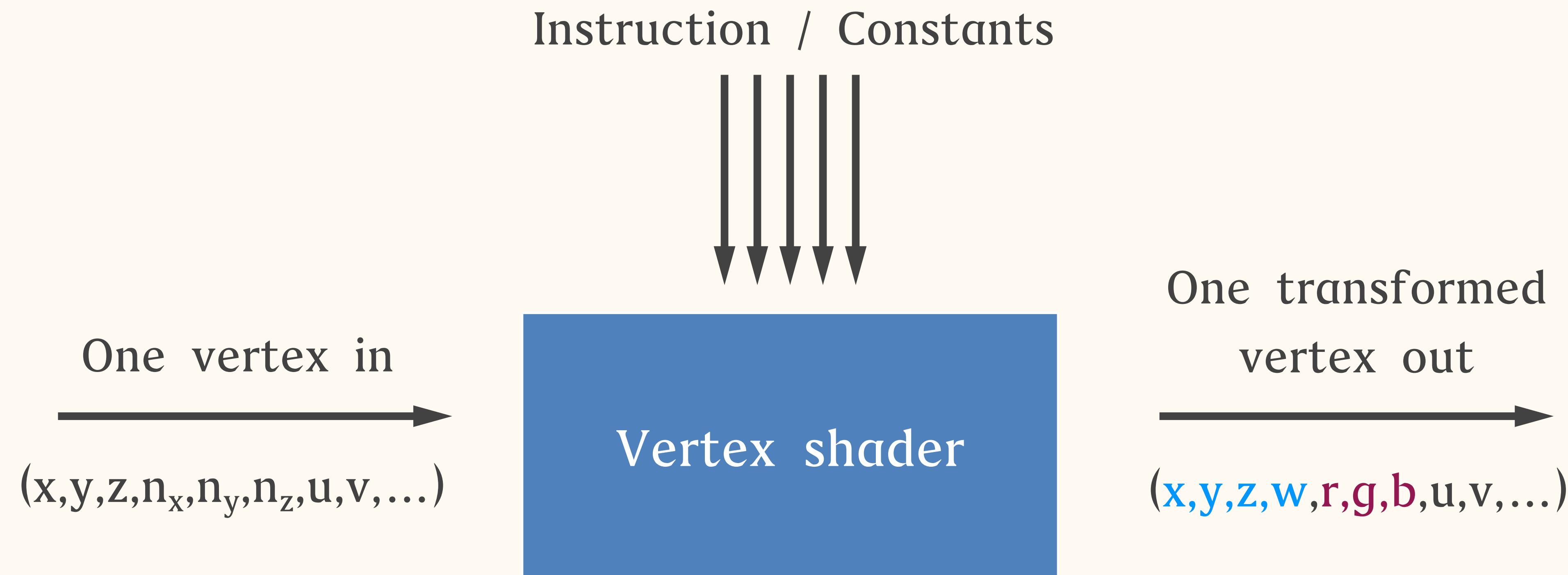
- 需要 3個 4x4 transformation 矩陣
 - World matrix
 - 將模型從模型座標系 (Model Space) 轉換到世界座標系 (World Space)
 - View matrix
 - 將模型從世界座標系 (World Space) 轉換到攝影機座標系 (View Space)
 - Projection matrix
 - 將攝影機坐標系的模型投影至螢幕上
 - Orthogonal or perspective projection (正投影或是透視投影)

$$\mathbf{v}' = \mathbf{M}_{\text{Projection}} \mathbf{M}_{\text{view}} \mathbf{M}_{\text{world}} \mathbf{v}$$

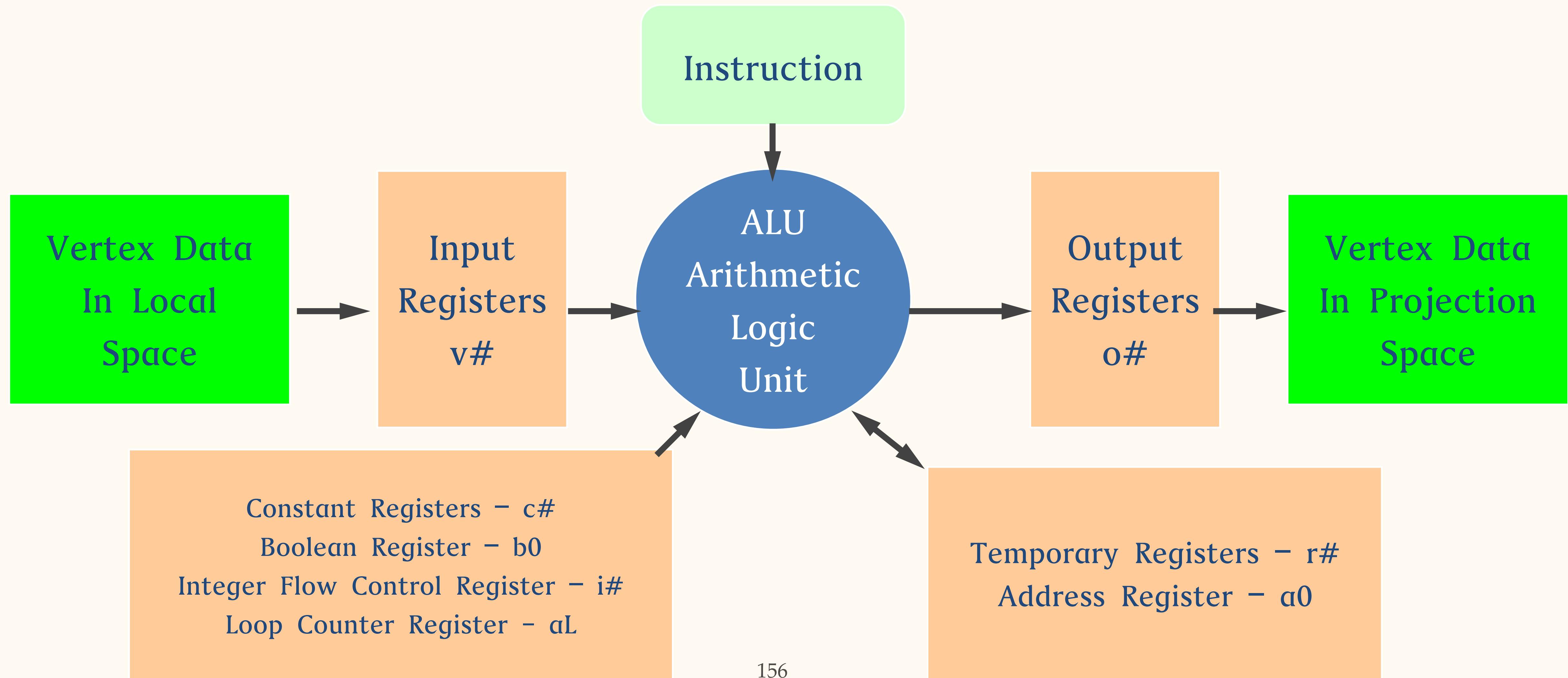
Vertex Processing

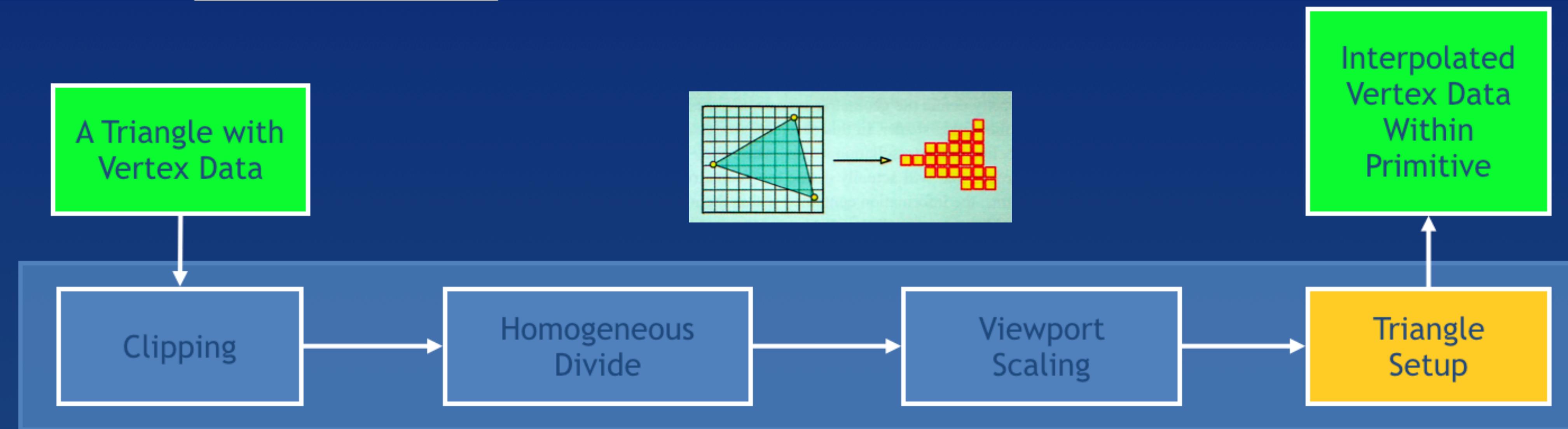
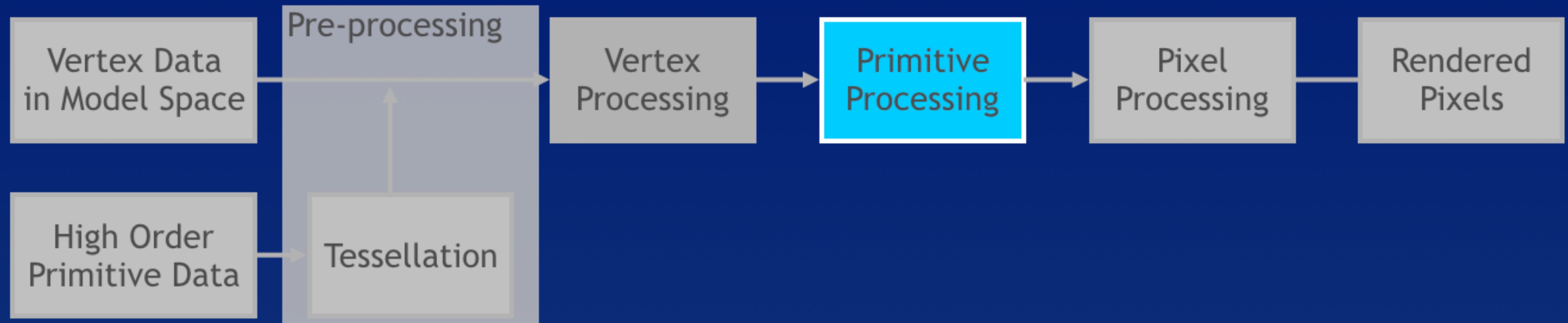
- 皮膚變形計算是需要在世界座標系中，通常在 vertex processing 階段處理
 - Vertex blending
- 最簡單的霧效果可以在 vertex processing 階段處理 (vertex fog)
 - 使用 z 值
- 可以在 vertex processing 計算光影
 - 結果與 fixed function 3D rendering pipeline 相同
 - 不過考慮渲染品質，我們會在 Pixel Shader 計算光影
- 當完成 vertex processing：
 - 頂點結果會是在螢幕座標系 (logical screen)：
 - Vertex position : (x, y, z, w)
 - w > 1.0 (input = 1.0)
- Vertex processing 是可程式化的.
 - Vertex shader

Vertex Shader



Vertex Shader Virtual Machine Block Diagram

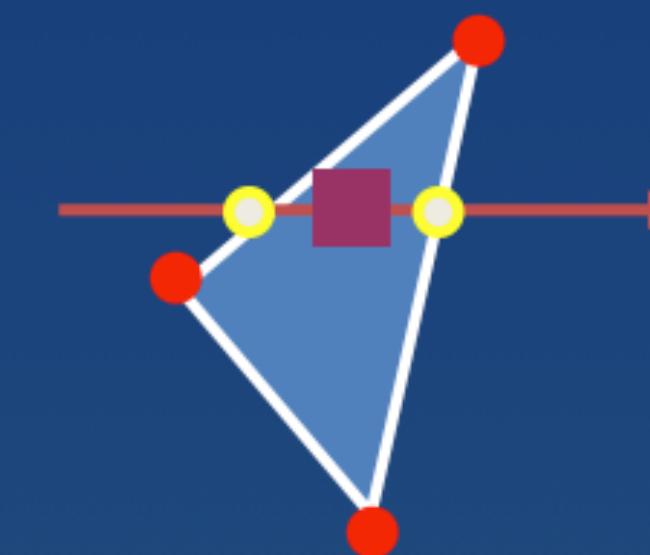
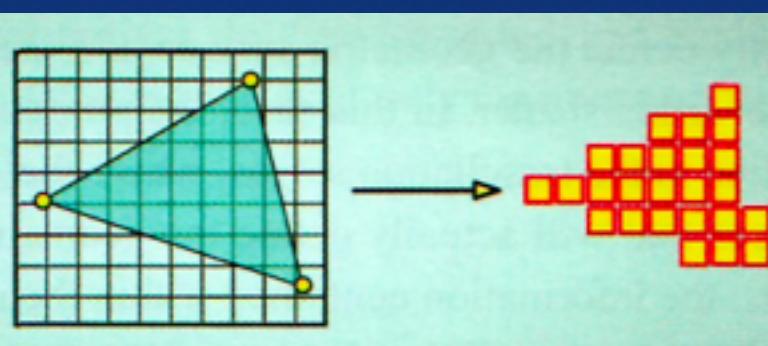




$$\begin{aligned} -w < x < w \\ -w < y < w \\ 0 < z < w \end{aligned}$$

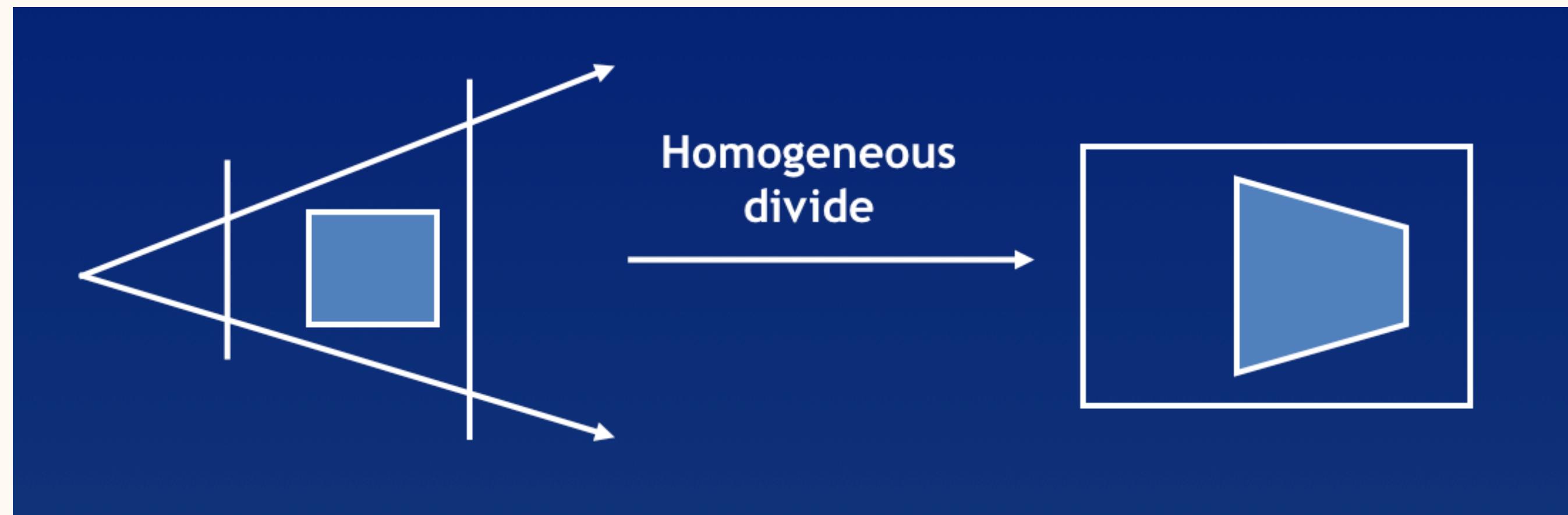
$$\begin{aligned} -1 < x/w < 1 \\ -1 < y/w < 1 \\ 0 < z/w < 1 \end{aligned}$$

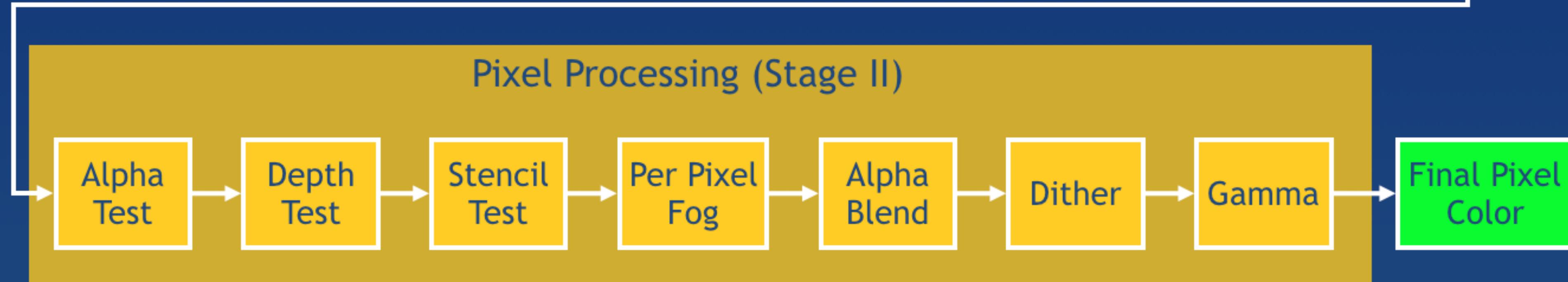
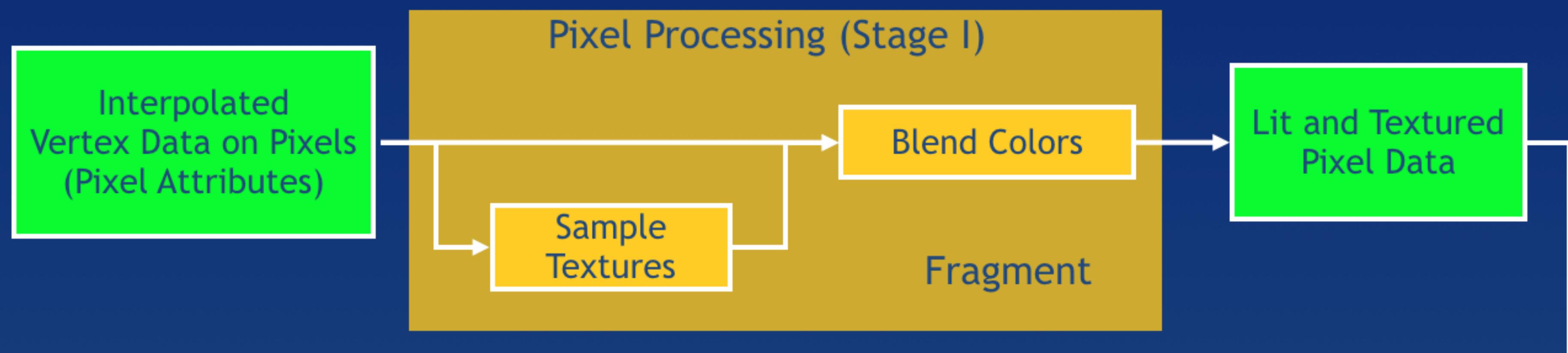
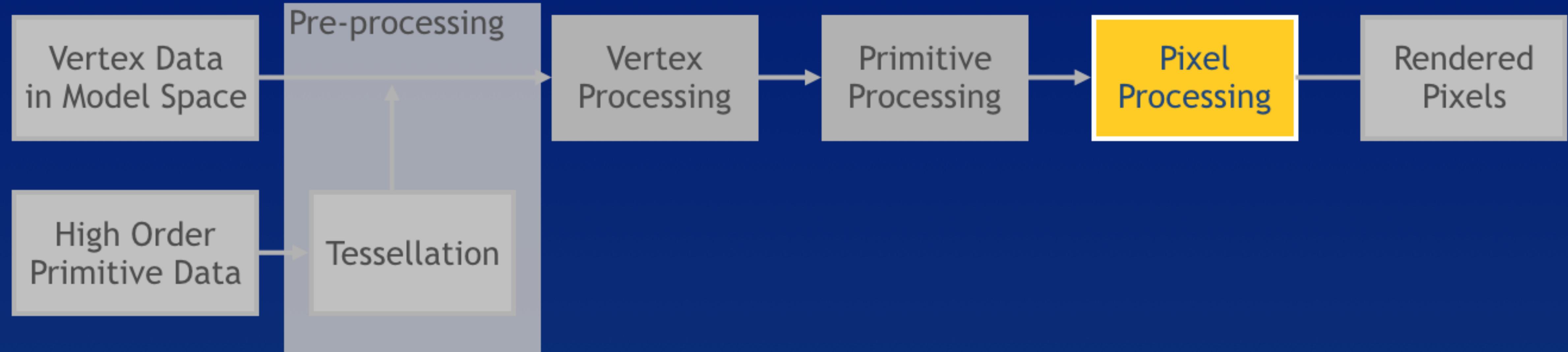
$$\begin{aligned} o_x \leq x_s < o_x + w \\ o_y \leq y_s < o_y + h \\ 0 < z_s < 1 \end{aligned}$$



Primitive Processing

- After homogeneous divide ($x/w, y/w, z/w$)
 - $-1 < x < 1$
 - $-1 < y < 1$
 - $0 < z < 1$
- 接著轉換至真實的顯示螢幕位置
 - Scale (x, y) to viewport
 - Map z value to z buffer
- 解出三角形內部的點 (Triangle setup)
 - Setup the triangle edge
 - Bi-linear interpolation using vertex data
- Primitive process 主要的功能是將 per-vertex 資料轉換出 per-pixel 資料
- Primitive process 是固定的 (Non-programmable)





Pixel Processing

- Pixel processing 是 rendering pipeline 中最關鍵的一環
- 兩階段：
 - Stage I : fragment processing
 - Programmable
 - Stage II :
 - Non-programmable
- In stage I :
 - GPU 根據像素上內差計算出的貼圖座標將貼圖內容取出
 - 進行資料融合計算 (blend per-pixel attributes) :
 - Diffuse and specular colors (input color registers)
 - Multiple texture samples

Pixel Processing

- 結果為多組完成光影與貼圖計算的 (r, g, b, a) 資料
- 這個過程也稱為 fragment processing
- In stage II :
 - 進行下列各項測試： (硬體執行)
 - Alpha test
 - Depth test (Z buffer test)
 - Stencil buffer test
 - In the alpha blend (半透明混色)
 - Apply pixel alpha to create a semi-transparent blend between a source pixel and frame buffer pixel
- Only the pixel processing stage I is programmable.

Alpha Blending

- 基本公式

$$C_f = C_s \alpha_s + C_d \alpha_d$$

- 通用公式

$$C_{final} = C_1 \alpha_1 + C_2 \alpha_2$$

$C_1, C_2 = C_s, 1 - C_s, C_d, 1 - C_d, Zero, One$

$\alpha_1, \alpha_2 = C_s, 1 - C_s, C_d, 1 - C_d, Zero, One, \alpha_s, 1 - \alpha_s, \alpha_d, 1 - \alpha_d$

$$C_f = C_{final}$$

$$C_s = C_{source}$$

$$C_d = C_{destination}$$

$$\alpha_s = \alpha_{source}$$

$$\alpha_d = \alpha_{destination}$$

Alpha Blending

- 常用的範例：
 - 加色法：遊戲特效

$$C_{final} = C_s + C_d$$

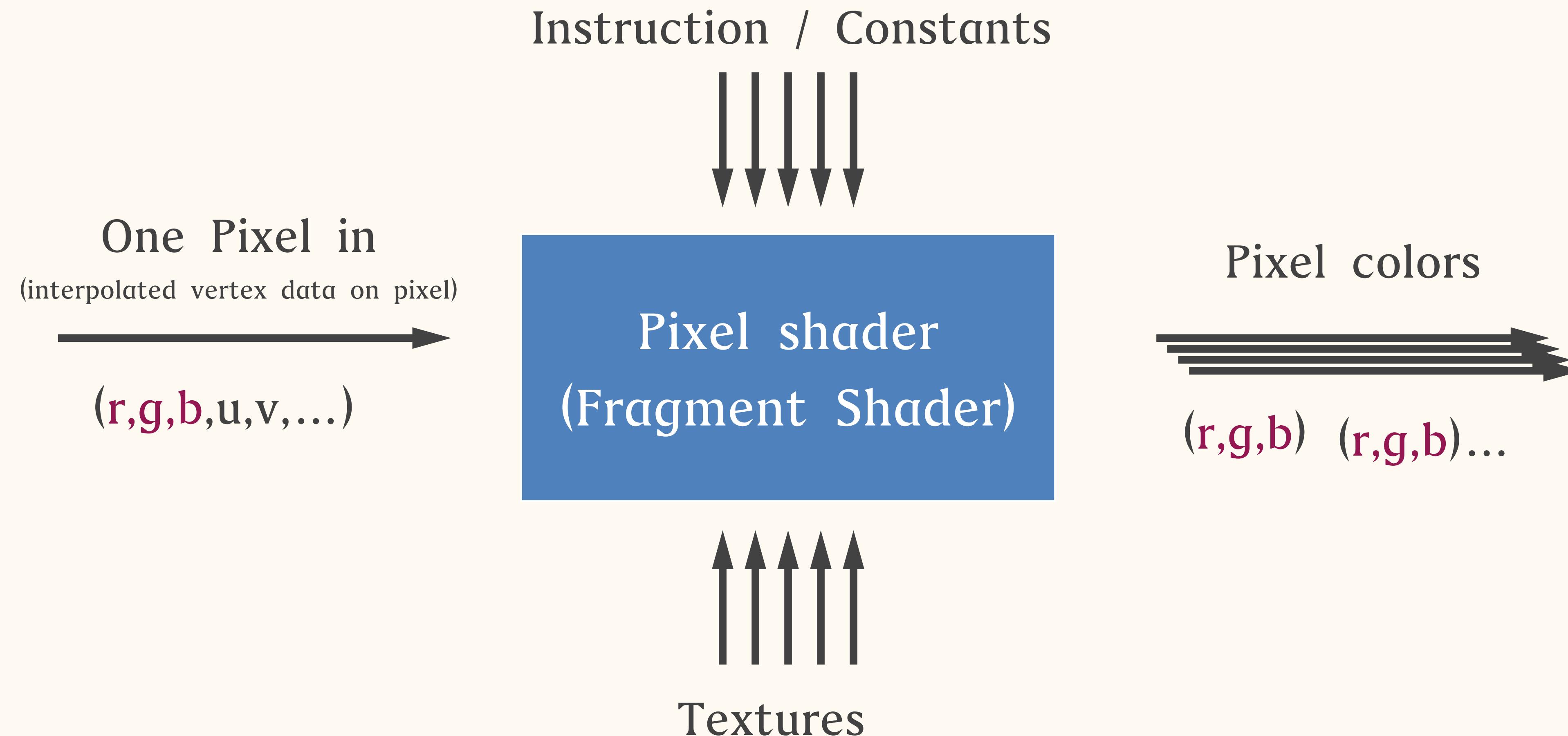


- 減色法：遊戲特效、影子、黑煙、Lightmap、...

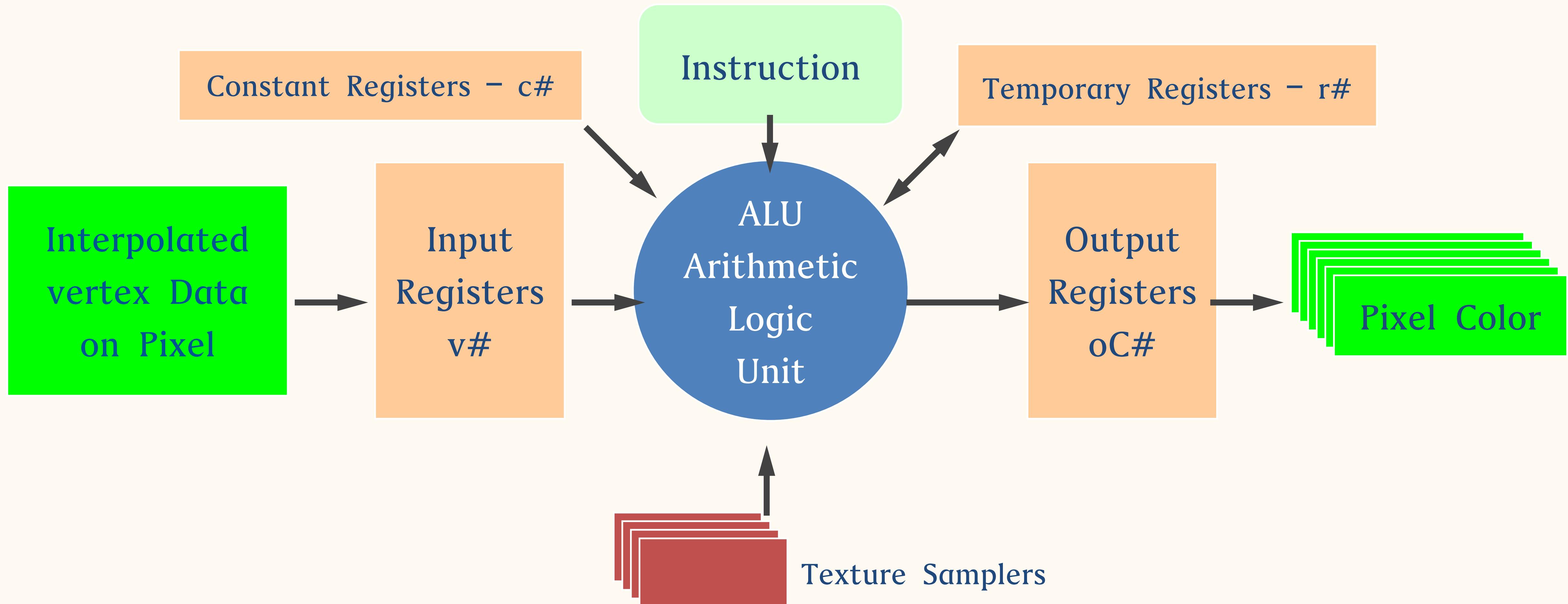
$$C_{final} = C_s * 0 + C_d(1 - C_s)$$



Pixel Shader

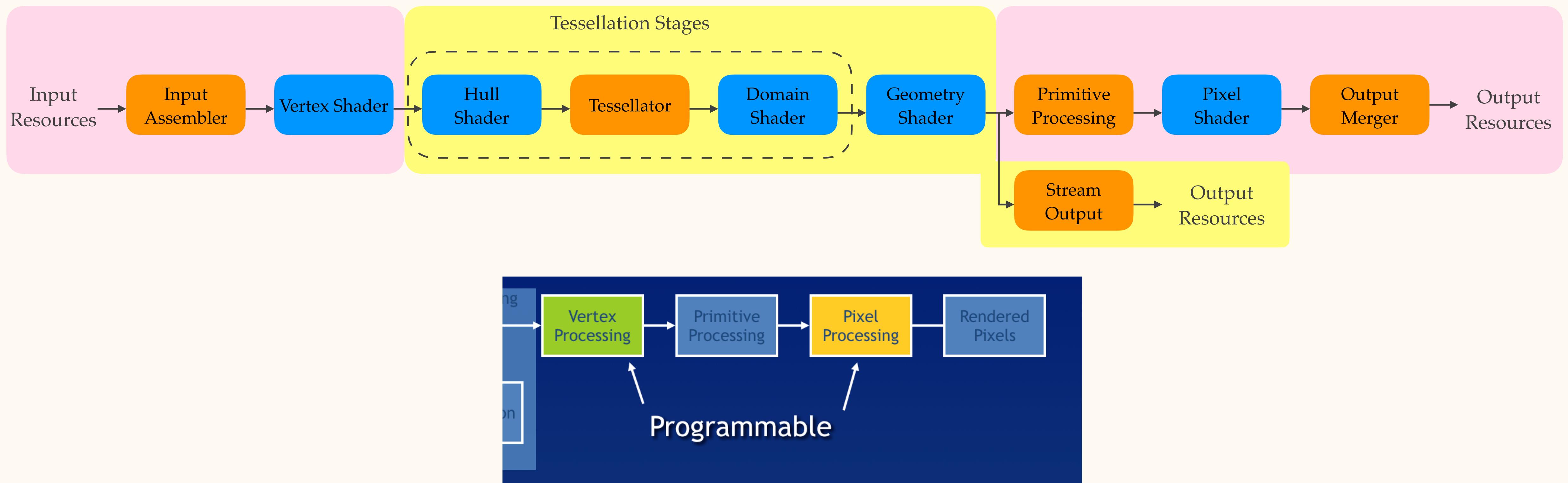


Pixel Shader Virtual Machine Block Diagram



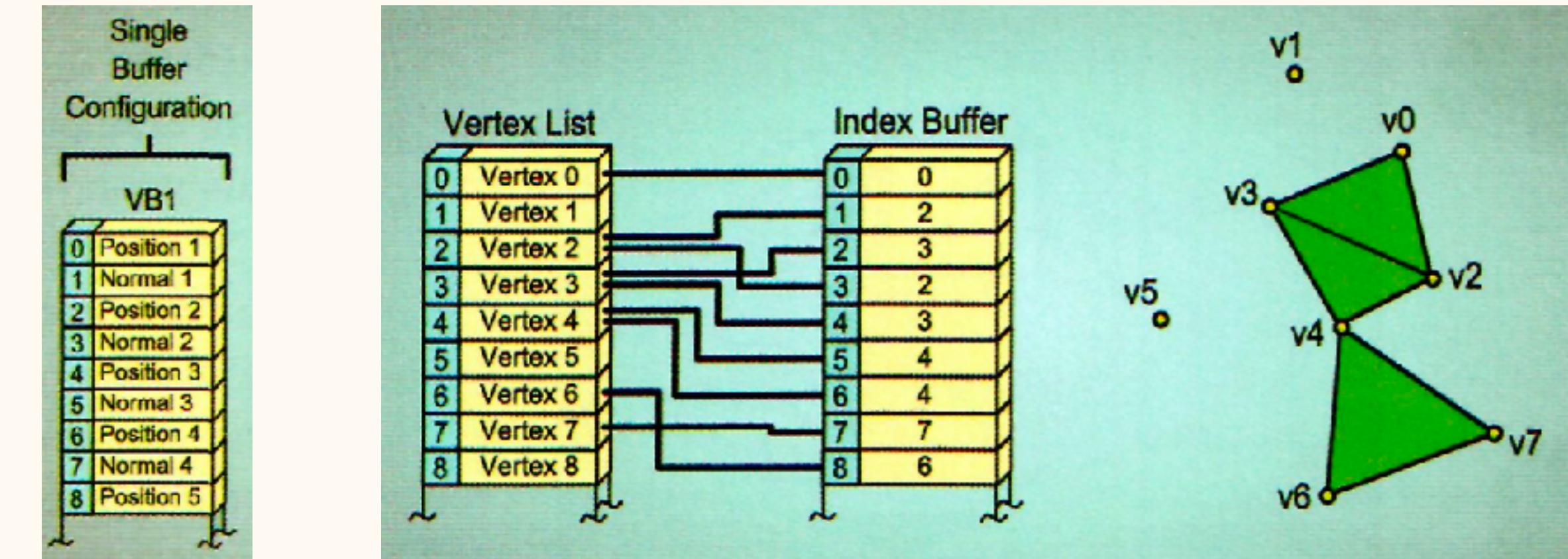
DX11 Programmable Rendering Pipeline

- Shader Model 5.0
- DirectX 11



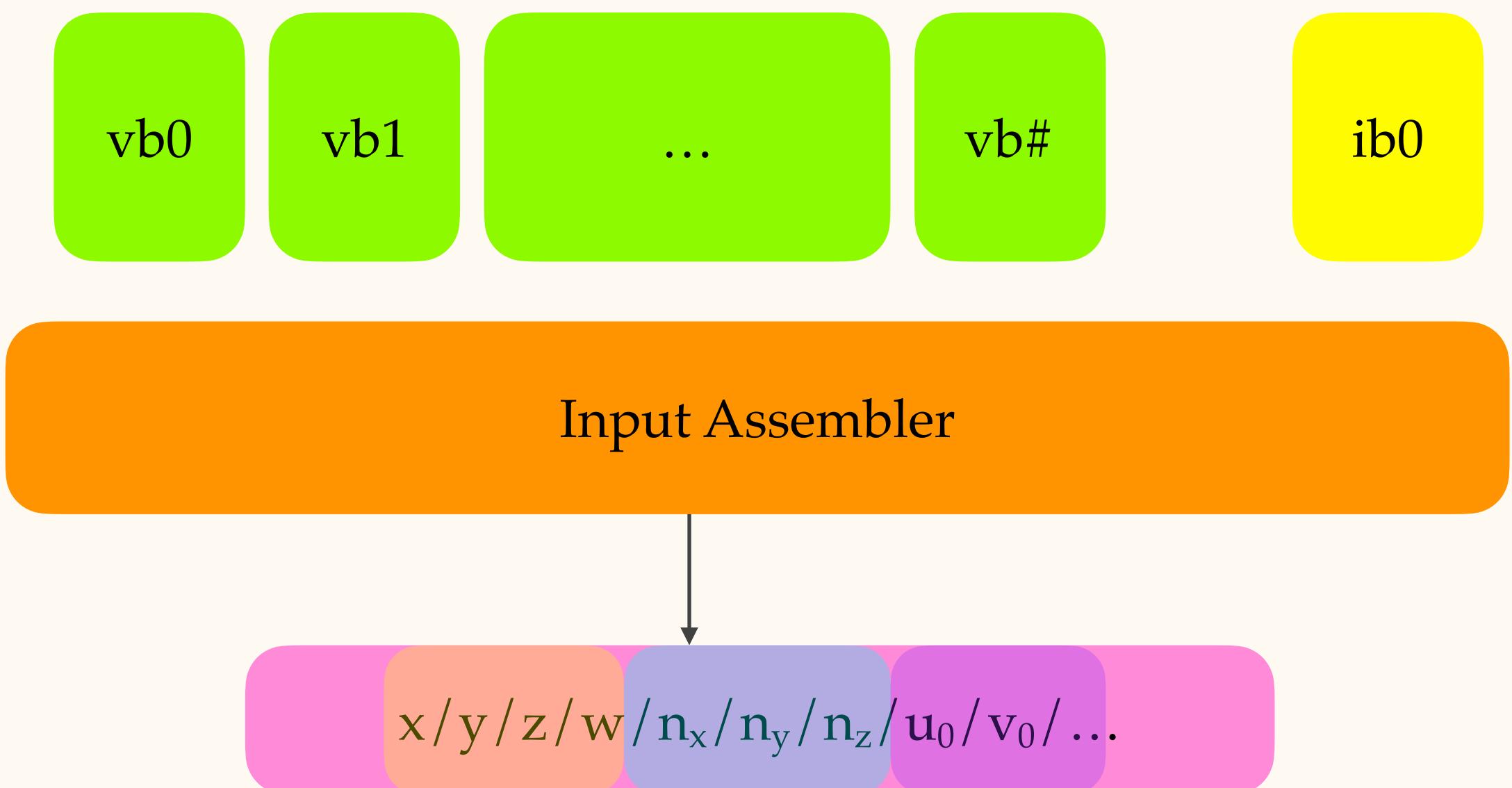
Input Assembler

- Indexed Primitives
 - Vertex buffers (vb0, vb1, ...)
 - Index buffers (ib0)
- 組裝渲染時需要的頂點資料做為 vertex shader 輸入的資料

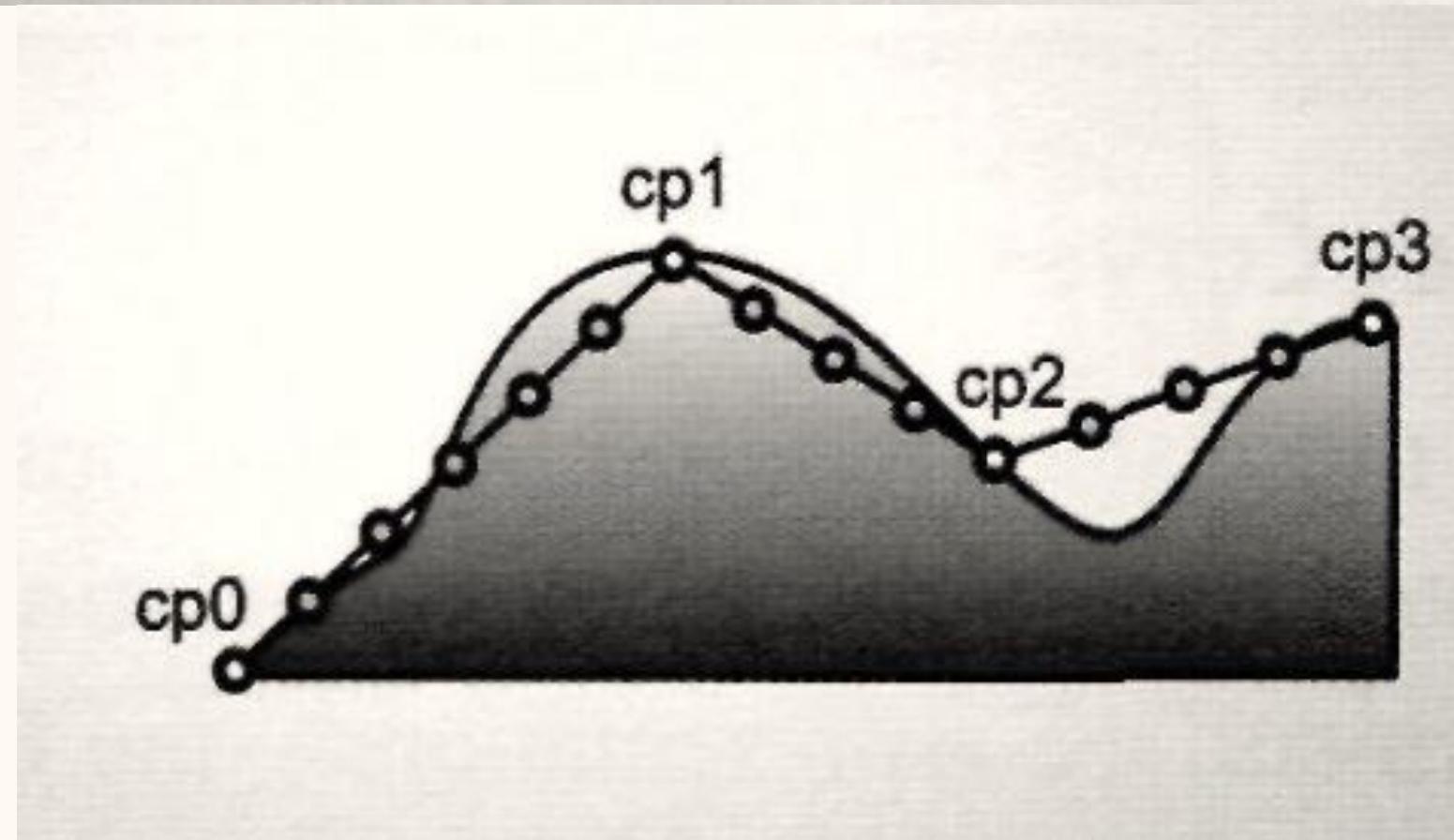
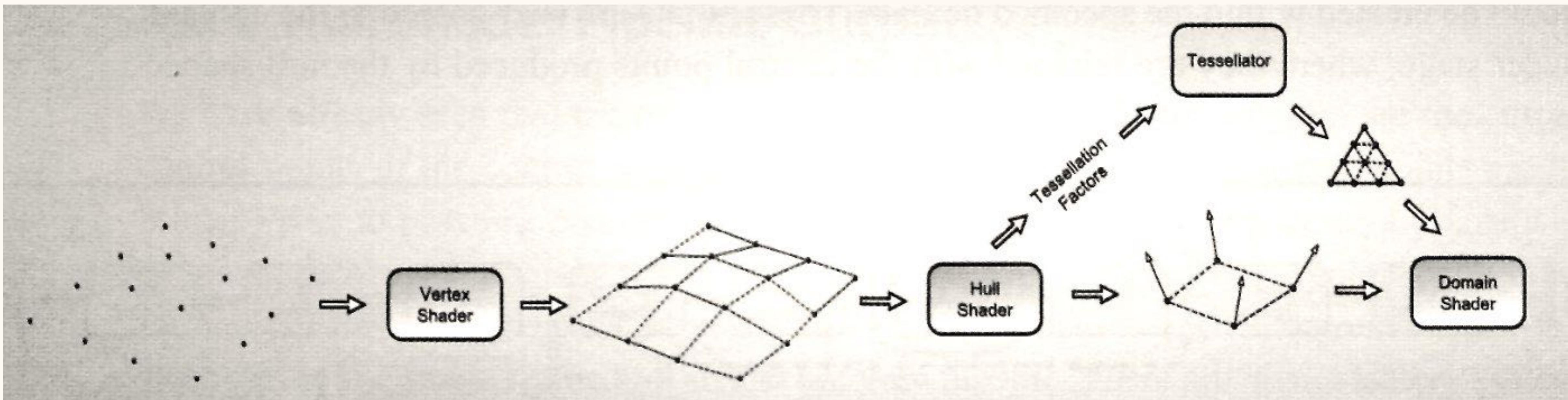


```
struct VS_INPUT
{
    float4 position : POSITION;
    float3 normal : NORMAL;
    float2 tex : TEXCOORDS;
};

VS_OUTPUT VSMAIN(in VS_INOUT v)
{
    ...
}
```

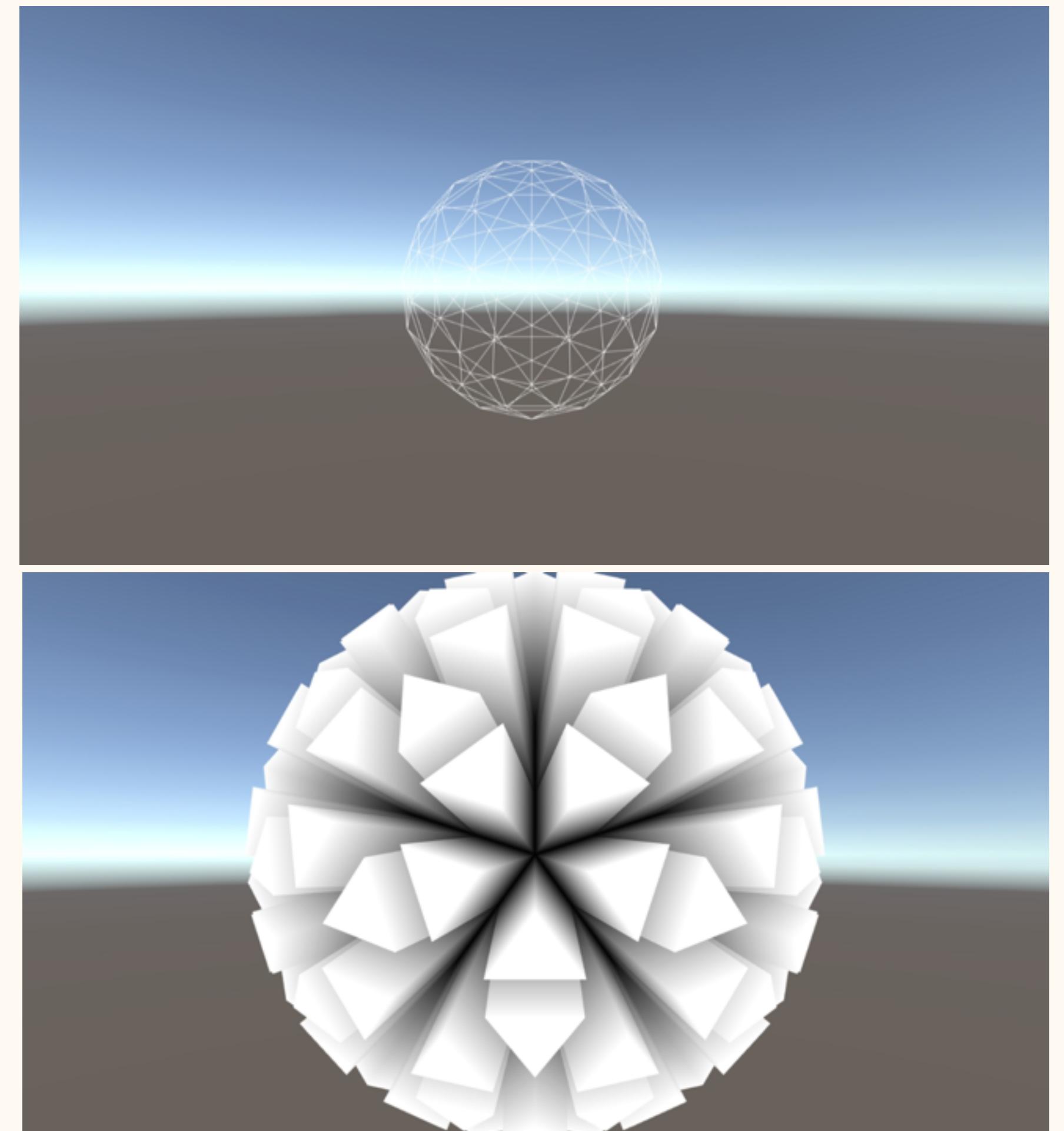
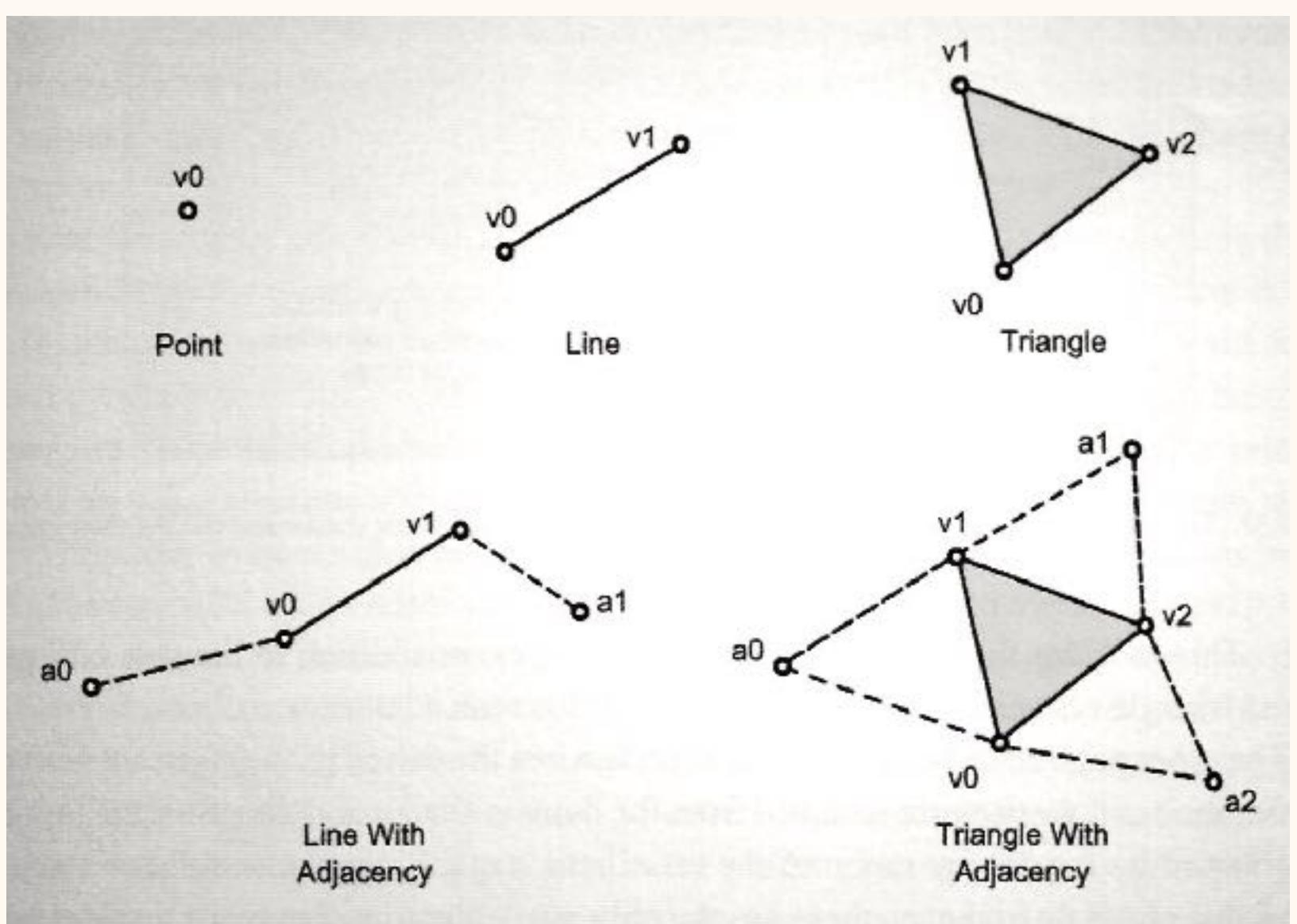
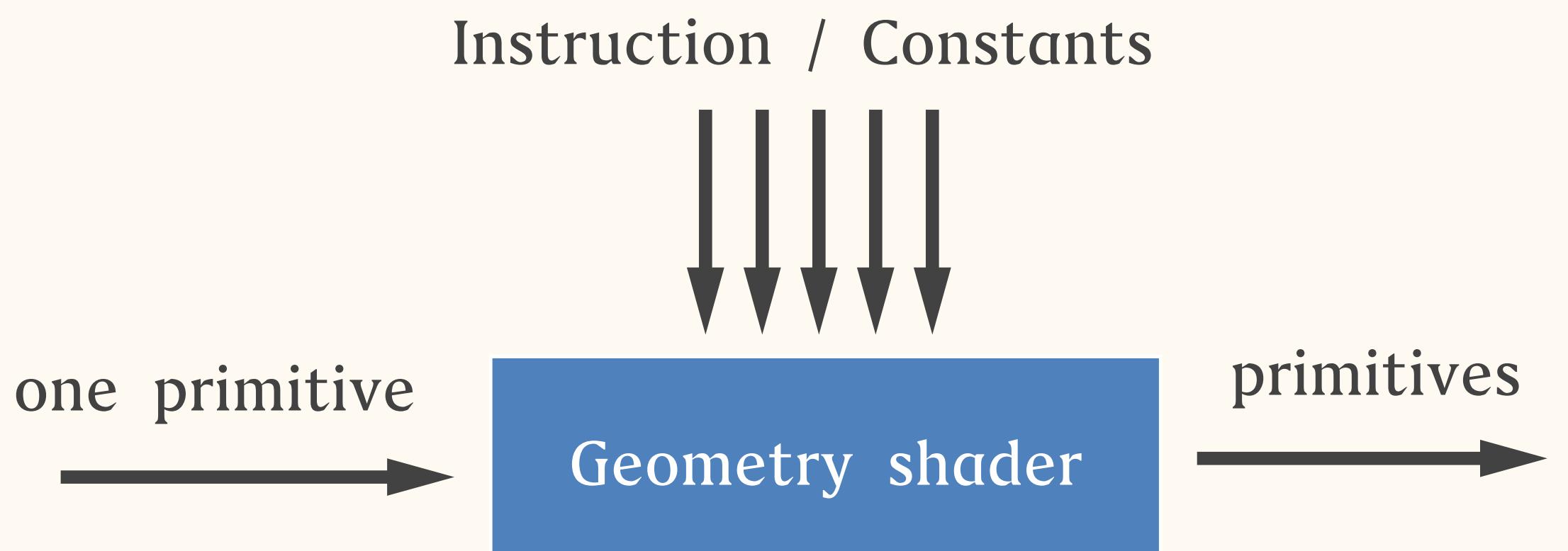


Hull Shader + Tessellator + Domain Shader



$$\mathbf{B}(t) = \sum_{j=0}^n t^j \mathbf{C}_j \quad \mathbf{C}_j = \frac{n!}{(n-j)!} \sum_{i=0}^j \frac{(-1)^{i+j} \mathbf{P}_i}{i!(j-i)!}$$

Geometry Shader

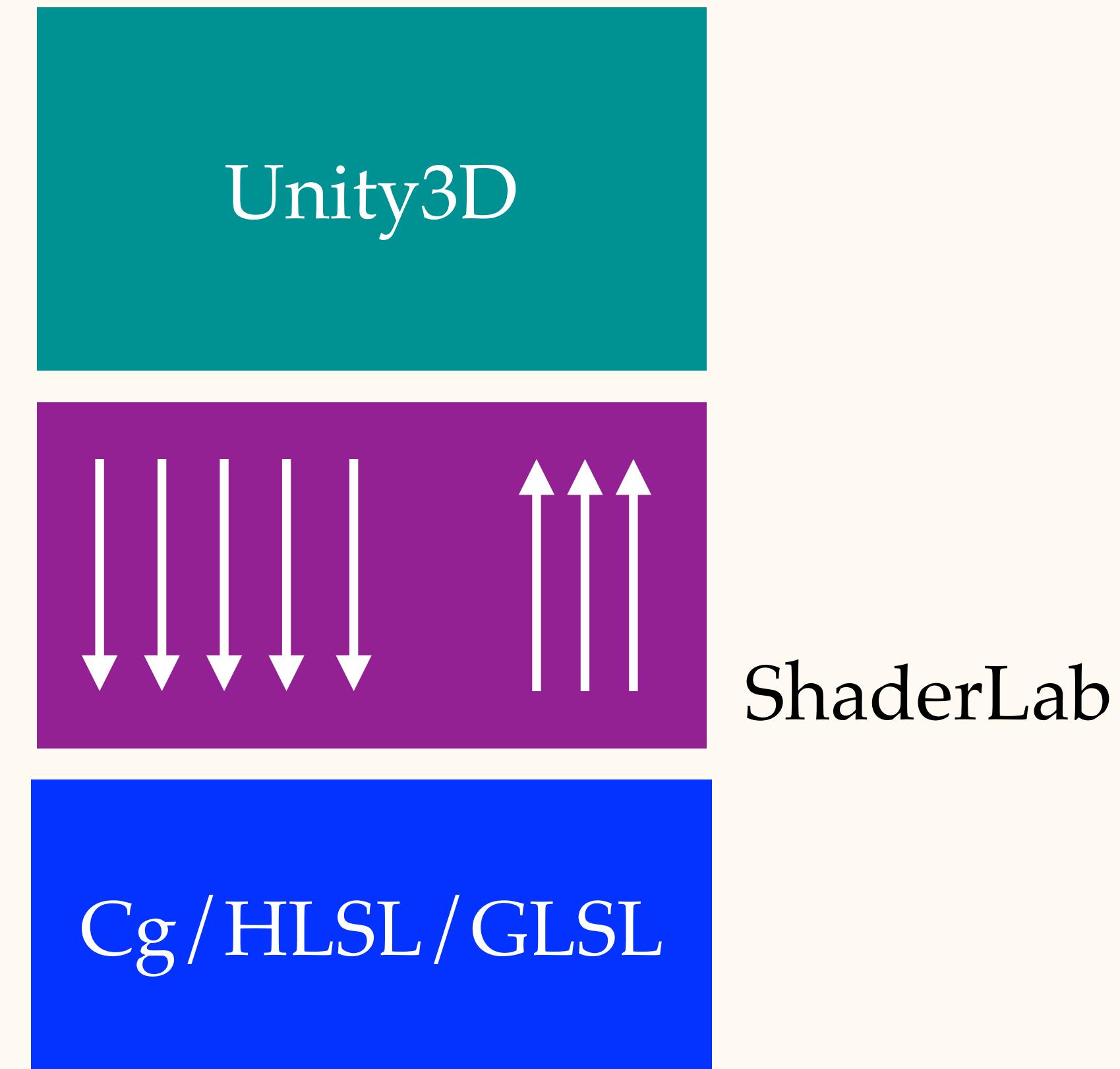
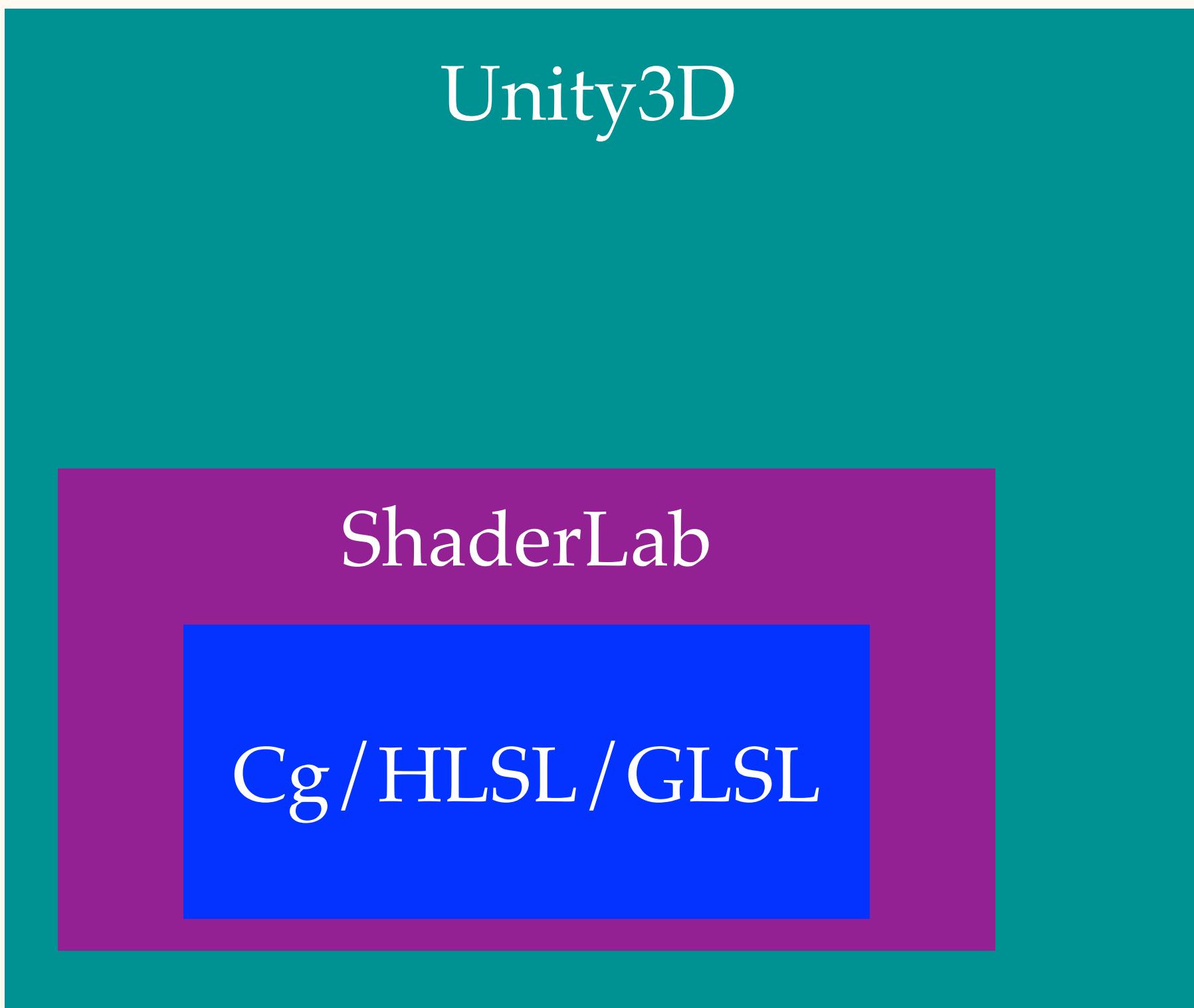


Shader Programming Basics

Shading Language

- Shader Model
 - Shader Model 1.0 ~ 6.0
 - Assembly code
- High-level Shading Language
 - 第一個 high-level shading language : nVidia Cg
 - Microsoft HLSL
 - High-Level Shading Language
 - OpenGL & OpenGL ES : GLSL
 - 採用 C 語言 語法，加上向量化與圖形等特性
 - 所有的高階 Shader 語言的書寫方式與架構，甚至於語法都類似，學習上很容易上手
- 我們上課的內容以 HLSL 為主

Shader 與遊戲引擎的關係



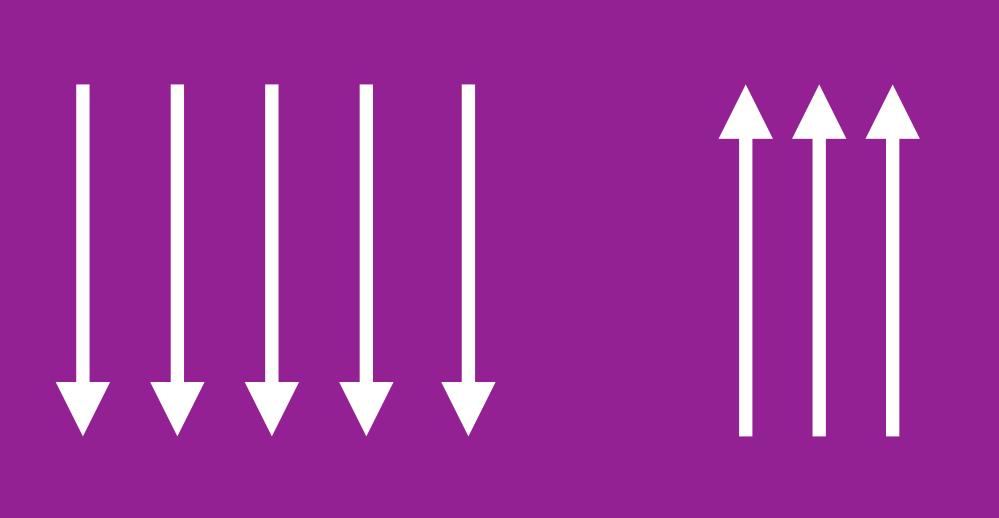
```

Pass {
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"
        struct v2f {
            float4 pos : SV_POSITION;
            fixed4 color : COLOR;
        };
        v2f vert (appdata_base v) {
            v2f o;
            o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
            o.color.rgb = v.normal * 0.5 + 0.5;
            o.color.a = 1.0;
            return o;
        }
        fixed4 frag (v2f i) : COLOR0
        {
            return i.color;
        }
    ENDCG
}

```

ShaderLab

Unity3D



Cg

HLSL - Data Types

Cg/HLSL scalar types

Data Type	Representable Values
bool	true or false
int	32-bit signed integer
half	16-bit floating-point value
float	32-bit floating-point value
double	64-bit floating-point value

HLSL – Data Types

- 向量資料宣告範例：

```
float3 vecLight;  
float vecLight[3];  
vector vecPos;  
vector <float, 4> vecPos;  
  
float4 pos = { 3.0f, 5.0f, 2.0f, 1.0f };  
float value = pos[0];  
float value = pos.x;  
float value = pos.r;  
float2 value = pos.xy;  
float3 value = pos.rgb;
```

HLSL – Swizzling & Constructor

- Swizzling 是指將任何來源向量組件資料複製到另一個向量組件資料的能力

```
float4 vect1 = { 4.0f, -2.0f, 5.0f, 3.0f };
float2 vect2 = vect1.yx;      // (-2.0f, 4.0f)
float scalar = vect1.w;       // 3.0
float3 vect3 = scalar.xxx;   // (3.0, 3.0, 3.0)
```

- Constructor examples :

```
float3 vPos = { 4.0f, -2.0f, 5.0f };
float fDiffuse = dot(vNormal, float3(1.0f, 0.0f, 0.0f));
float4 vPack = float4(vPos, fDiffuse);
```

HLSL – Write Mask

- Masking controls how many components are written.

```
float4 vect3 = { 4.0f, -2.0f, 5.0f, 3.0f };
float4 vect1 = { 1.0f, 2.0f, 3.0f, 4.0f };
vect1.xw = vect3;           // (4.0f, 2.0f, 3.0f, 3.0f)
vect1.y = vect3;           // (4.0f, -2.0f, 3.0f, 3.0f)
```

- Assignment can not be written to the same component more than once.
- Component namespaces can not be mixed.

```
f_4D.xx = pos.xy;          // error!
f_4D.xg = pos.rg;          // error!
```

HLSL – Intrinsics

- Math intrinsics
- Texture sampling intrinsics
- Wave-related intrinsics
 - Shader Model 6.0
- Quad-related intrinsics
 - Shader Model 6.0

HLSL – Math Intrinsics

- $\text{abs}(x)$
 - Absolute value(per component)
- $\text{acos}(x)$
 - Calculate the arccosine of each component of x
- $\text{all}(x)$
 - Test if all components of x are non-zero
- $\text{any}(x)$
 - Test if any component of x is non-zero
- $\text{asin}(x)$
 - Calculate the arcsine of each component of x

HLSL – Math Intrinsics

- `atan(x)`
 - Calculate the arctangent of each component of x
- `atan2(y, x)`
 - Calculate the arctangent of y/x
- `ceil(x)`
 - Return the smallest integer that is greater or equal to x
- `clamp(x, min, max)`
 - Clamp x to the range(\min, \max)
- `discard`
 - Discard current pixel

HLSL – Math Intrinsics

- $\cos(x)$
 - Return the cosine of x
- $\cosh(x)$
 - Return the hyperbolic cosine of x
- $\text{cross}(a, b)$
 - Return the cross product of two 3D vectors, a & b
- $\text{ddx}(x)$
 - Return partial derivative of x with respective to the screen x axis
- $\text{ddy}(x)$
 - Return partial derivative of x with respective to the screen y axis

HLSL – Math Intrinsics

- `degree(x)`
 - Convert x from radians to degrees
- `determinant(m)`
 - Return the determinant of a square matrix m
- `distance(a, b)`
 - Return the distance between two points, a & b
- `dot(a, b)`
 - Returns the dot product of two vectors, a & b
- `exp(x)`
 - Return base-e exponential of x

HLSL – Math Intrinsics

- `exp2(x)`
 - Return base-2 exponential of x
- `faceforward(n, i, ng)`
 - Test if a face is visible
 - Return $-n \cdot \text{sign}(\text{dot}(i, ng))$
- `isnan(x)`
 - Return true if x is NAN or QNAN
 - $0/0$ causes NAN
- `ldexp(x, exp)`
 - Return $x \cdot 2^{\text{exp}}$
- `len(v)`
 - Vector length

HLSL – Math Intrinsics

- `length(v)`
 - Return the length of the vector, v
- `lerp(a, b, s)`
 - Return $a + s(b-a)$
- `log(x)`
 - Return the base-e logarithm of x
- `log10(x)`
 - Return the base-10 logarithm of x
- `log2(x)`
 - Return the base-2 logarithm of x
- `max(a, b)`
 - Select the greater of a & b

HLSL – Math Intrinsics

- $\min(a, b)$
 - Select the lesser of a & b
- $\text{modf}(x, \text{out } ip)$
 - Split the value of x into fractional and integer part
- $\text{mul}(a, b)$
 - Perform matrix multiplication between a & b
- $\text{normalize}(v)$
 - Return the normalized vector $v/\text{length}(v)$
- $\text{pow}(x, y)$
 - Return x^y

HLSL – Math Intrinsics

- **radians(x)**
 - Convert x from degree to radians
- **reflect(i, n)**
 - Return the reflection vector v
 - i is entering ray direction
 - n is the surface normal vector

```
float3 reflect(float3 I, float3 N)
{
    // R = I - 2 * N * (I·N)
    return I - 2*N*dot(I, N)
}
```

HLSL – Math Intrinsics

- `refract(i, n, eta)`
 - Return the refraction vector v
 - i is entering ray direction
 - n is the surface normal vector
 - η is the relative refraction index
- `round(x)`
 - Return x to nearest integer which is less than x
- `rsqrt(x)`
 - Return $1/\sqrt{x}$
- `saturate(x)`
 - Clamp x to the range $[0, 1]$

HLSL – Math Intrinsics

- $\text{sign}(x)$
 - Compute the sign of x
 - Return -1 if x is negative, 0 if $x = 0$, 1 if x is positive
- $\sin(x)$
 - Return sine of x
- $\text{sincos}(x, \text{out } s, \text{out } c)$
 - Return the sine and cosine of x
- $\sinh(x)$
 - Return hyperbolic sine of x

HLSL – Texture Sampling

- 4 types of texture sampling
 - 1D texture sampling
 - 2D texture sampling
 - 3D texture sampling
 - Volume texture
 - Cubemap texture sampling

HLSL – Texture Sampling

- If the texture data format is in integer form :
 - A8R8G8B8 / A16R16G16B16 / DXT1 / DXT3 / ...
 - When read texture data, you will get the color is range of 0.0 - 1.0.
 - When you output data to an integer form rendering target texture, the color will be automatically mapping to texture data format.
- if the texture data format is in floating-point form :
 - the texture will keep the data as you have in shader!

HLSL - 2D Texture Intrinsics

- DirectX 11 syntax

```
Texture2D albedoMap;
```

```
SamplerState albedoMapSampler;
```

```
...
```

```
float4 color = albedoMap.Sample(albedoMapSampler, uv);
```

HLSL - Cubemap Texture Intrinsics

- DirectX 11 syntax

```
TextureCube cubeMap;  
SamplerState cubeMapSampler;  
  
...  
  
float4 color = cubeMap.Sample(cubeMapSampler, v3D);
```

Shader 程式的基本組成

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP : packoffset(c0); // matrix from local to screen space
};
struct VS_INPUT
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
struct v2f {
    float4 pos : SV_POSITION;
    float4 color : COLOR;
};
v2f vert (VS_INPUT v) {
    v2f o;
    o.pos = mul (mWVP, v.vertex);
    o.color.rgb = v.normal * 0.5 + 0.5;
    o.color.a = 1.0;
    return o;
}
```

HLSL vertex shader

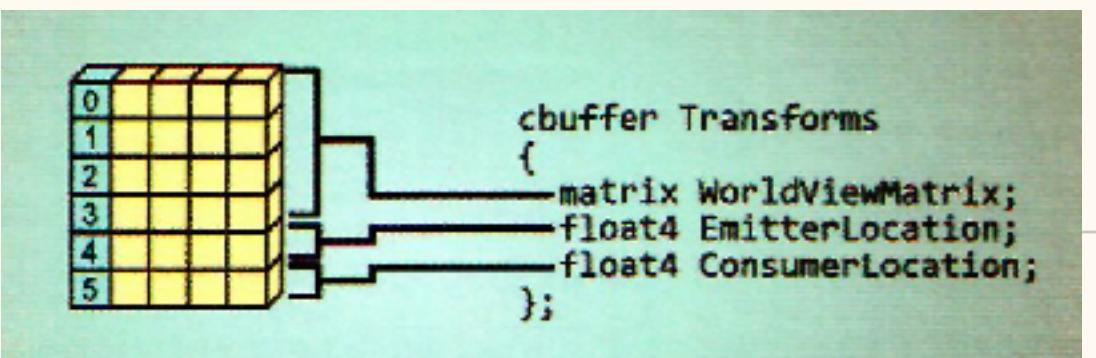
```
struct v2f {
    float4 pos : SV_POSITION;
    float4 color : COLOR;
};
float4 frag (v2f i) : SV_TARGET0
{
    return i.color;
}
```

HLSL pixel shader

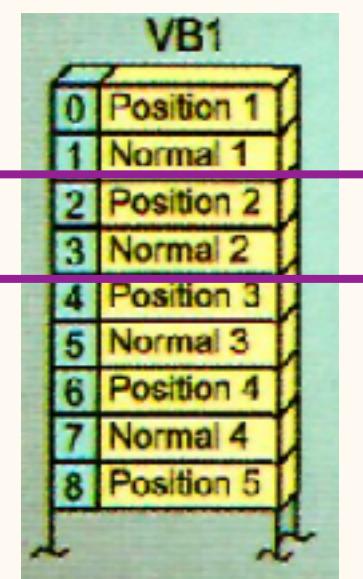
```
Pass {
    CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag
        #include "UnityCG.cginc"
        struct v2f {
            float4 pos : SV_POSITION;
            fixed4 color : COLOR;
        };
        v2f vert (appdata_base v) {
            v2f o;
            o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
            o.color.rgb = v.normal * 0.5 + 0.5;
            o.color.a = 1.0;
            return o;
        }
        fixed4 frag (v2f i) : COLOR0
        {
            return i.color;
        }
    ENDCG
}
```

Unity shader

Shader 程式的基本組成



Constants
(constant buffer)



vertex shader
Input

vertex shader
Output

vertex shader
code

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP : packoffset(c0); // matrix from local to screen space
};
```

指定所使用的暫存器編號

```
struct VS_INPUT
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
};
```

semantics

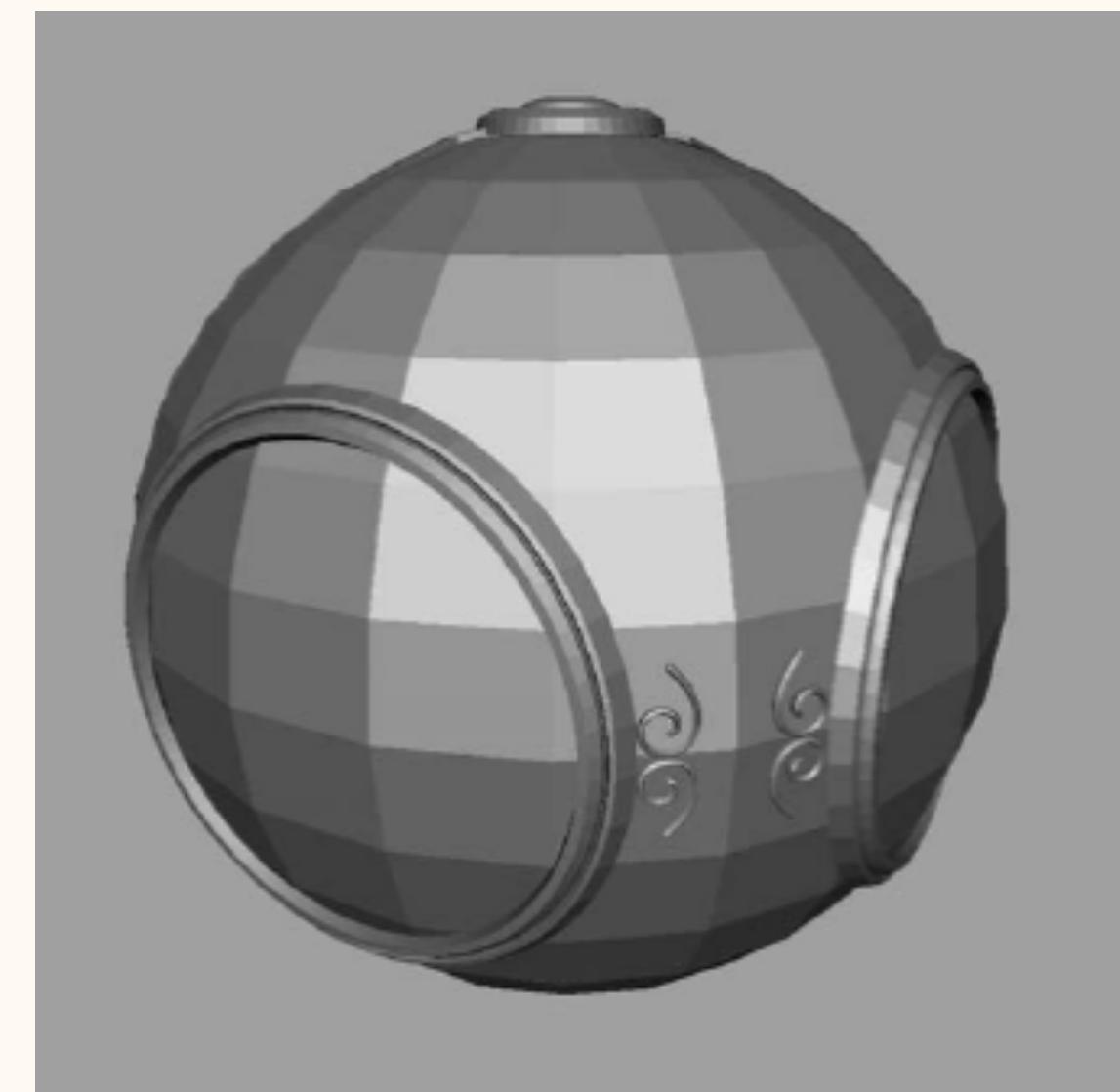
```
struct v2f {
    float4 pos : SV_POSITION;
    float4 color : COLOR;
};
```

```
v2f vert (VS_INPUT v) {
    v2f o;
    o.pos = mul (mWVP, v.vertex);
    o.color.rgb = v.normal * 0.5 + 0.5;
    o.color.a = 1.0;
    return o;
}
```

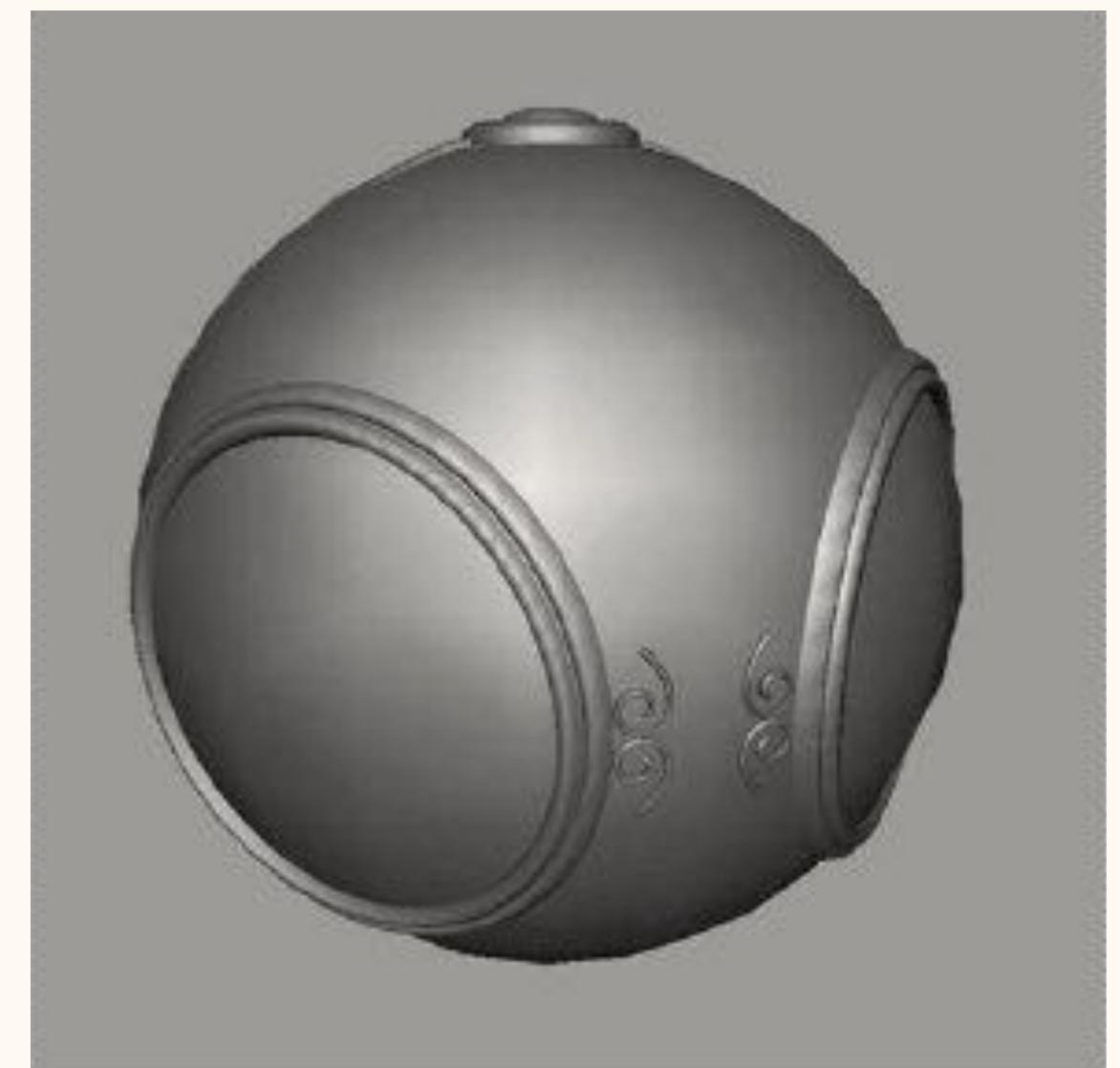
Shader - Hello World

Shading Techniques

- Real-time shading is still based on “scan conversion” to render triangles
- Flat shading
- Smooth shading
 - 利用掃描線 (Scanline) 次序 (y座標)，再沿著掃描線 (x座標) 內差三角面內的資料
 - Bi-linear interpolation scheme
 - 兩種常用的方法：
 - Gouraud shading (Gouraud 1971)
 - Phong shading (Phong 1975a)



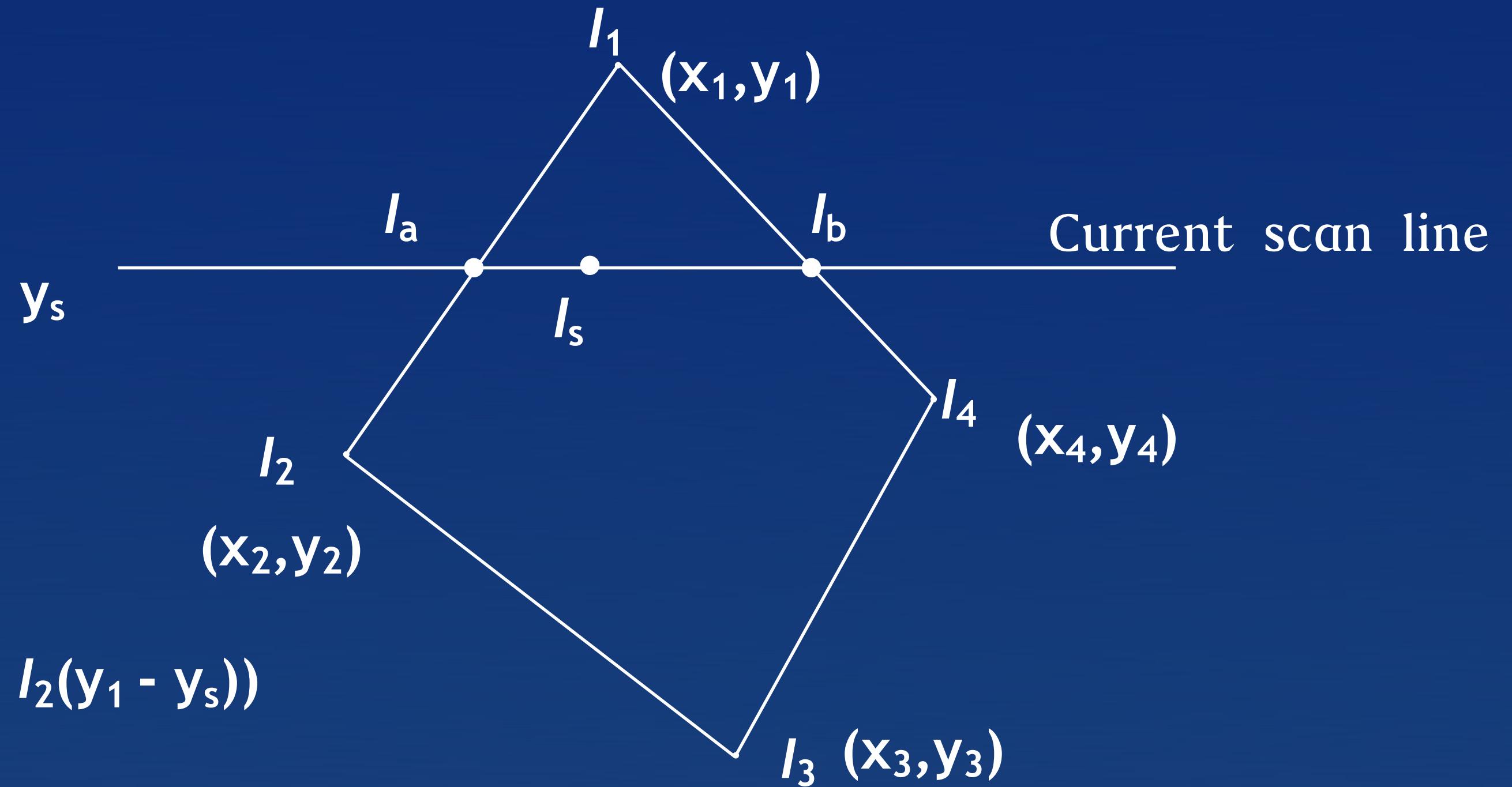
Flat Shading



Smooth Shading
(Gouraud, 1971)

Gouraud Shading

- Gouraud, 1971



$$I_a = \frac{1}{y_1 - y_2} (I_1(y_s - y_2) + I_2(y_1 - y_s))$$

$$I_b = \frac{1}{y_1 - y_4} (I_1(y_s - y_4) + I_4(y_1 - y_s))$$

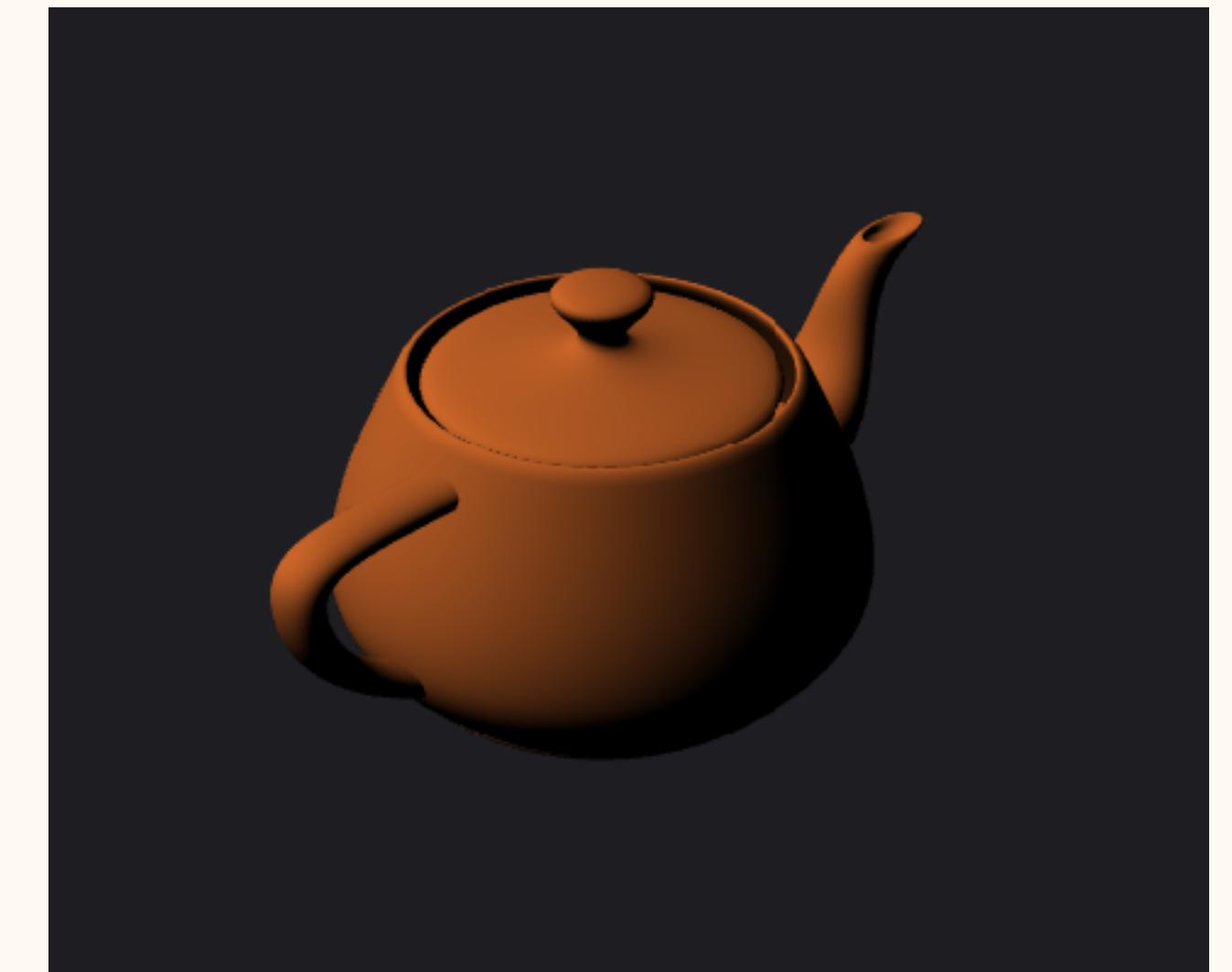
$$I_s = \frac{1}{x_b - x_a} (I_a(x_b - x_s) + I_b(x_s - x_a))$$

$$I = K_d I_l \cos \theta = K_d I_l (N \cdot L)$$

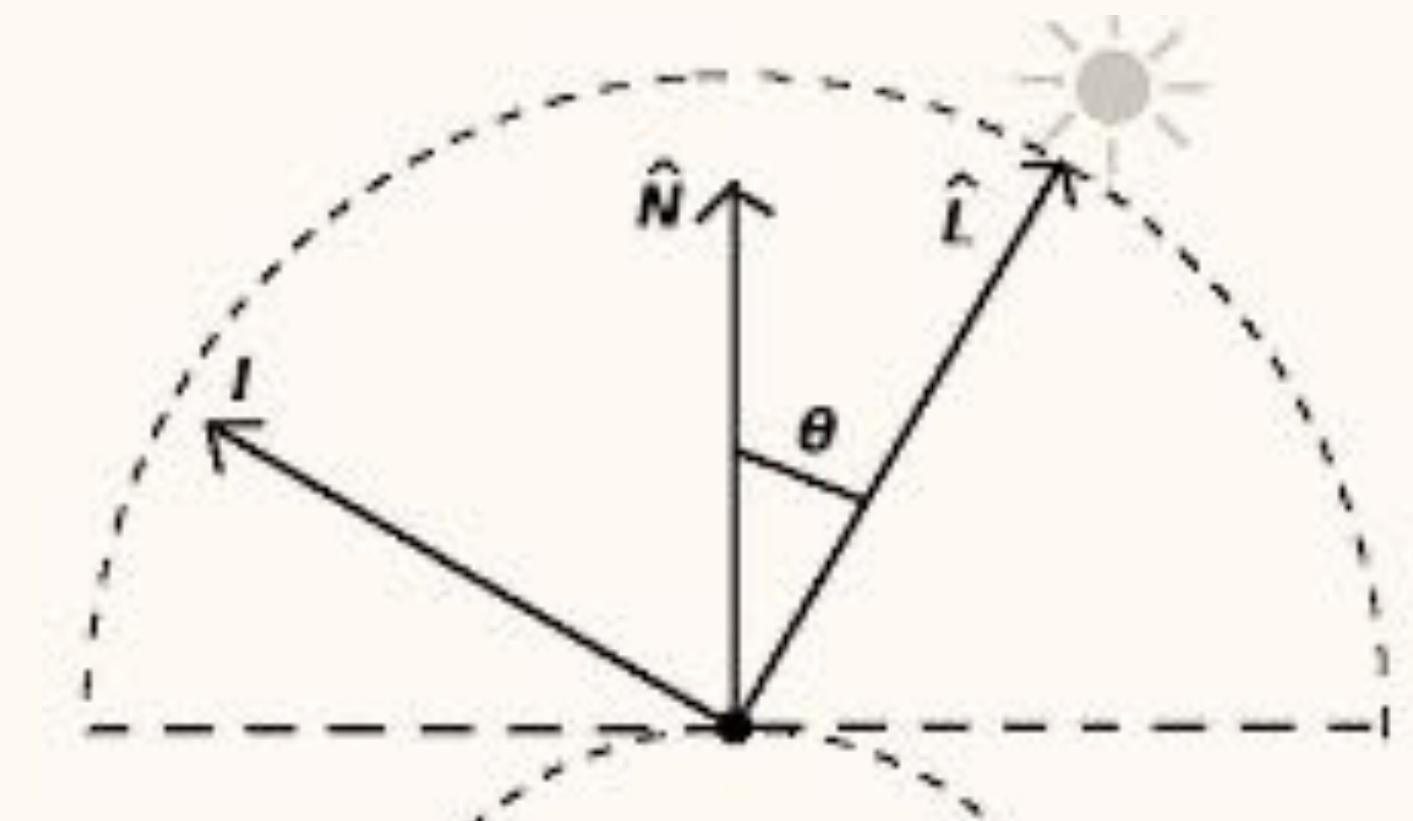
Do lighting in vertex shader
Per-vertex Lighting

Gouraud Shader : Hello World

- Gouraud Shading
 - 1971 Gouraud 的論文
 - 在頂點運算時計算光影的強弱
 - Lighting in vertex shader
 - 無法有效且正確顯示 specular，只有 diffuse 項目
 - “Lambertian” Shading Model



$$I = K_d I_l \cos\theta = K_d I_l (\hat{N} \cdot \hat{L})$$



Gouraud Shader : Constants

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP          : packoffset(c0);      // matrix from local to screen space
    matrix mWorld         : packoffset(c4);      // matrix from local to world space
    float3 mainLightPosition : packoffset(c8);    // position of the "mainLight"
    float4 mainLightColor   : packoffset(c9);    // color of the "mainLight"
    float4 dif            : packoffset(c10);     // diffuse component of the material
    float4 amb            : packoffset(c11);     // ambient component of the material
};
```

Gouraud Shader : Vertex Shader

```
// vertex shader input
struct VS_INPUT
{
    float4 inPos    : POSITION;
    float3 inNorm   : NORMAL;
};

// vertex shader output
struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float4 color    : COLOR0;
};
```

```
// the vertex shader
VS_OUTPUT MainVS(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // convert the vertex from local to world space
    float4 a = mul(mWorld, in1.inPos);

    // convert the vertex to screen space
    out1.pos = mul(mWVP, in1.inPos);

    // prepare the normal in world space for lighting (方法不對但效果一樣)
    float3 normW = normalize(mul((float3x3) mWorld, in1.inNorm));

    // prepare the lighting dirction L
    float3 L = normalize(mainLightPosition - a.xyz);

    // do Lambertian shading (N dot L)
    float diffuseInty = saturate(dot(L, normW));

    // get final result = I*Kd*(N dot L) + 一點環境光
    out1.color = mainLightColor*diffuseInty*dif + 0.5*mainLightColor*amb;

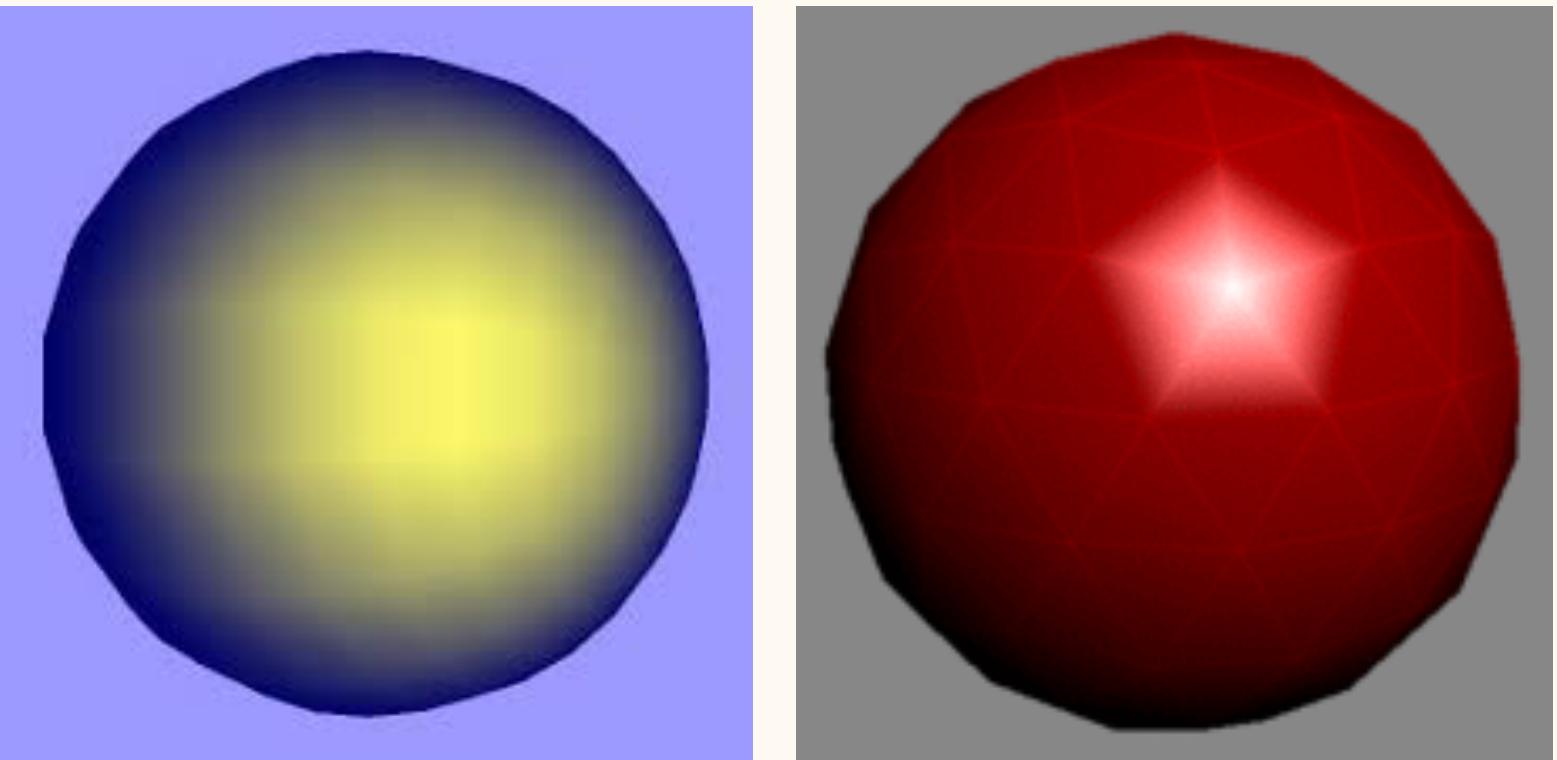
    return out1;
}
```

Gouraud Shader : Pixel Shader

```
// This shader implements Gouraud shading
//  
  
// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float4 color    : COLOR0;
};  
  
// the pixel shader
float4 MainPS(PS_INPUT in1) : SV_TARGET0
{
    return in1.color;
}
```

Gouraud Shading 的特性

- 第一個 smooth shading 的演算法
- Simple
- Economic
- 有 “Mach Banding” 效應
- 只能渲染 diffuse
 - Error rendering for specular component
- 過去 Fixed function rendering pipeline 是使用 Gouraud shading + specular 是一種錯誤的設計，現在是撰寫 Shader 來實現 Gouraud shading 要避免此錯誤



Normal Vector Transformation

- For linear transformation, the normal vector transformation is simplified as :

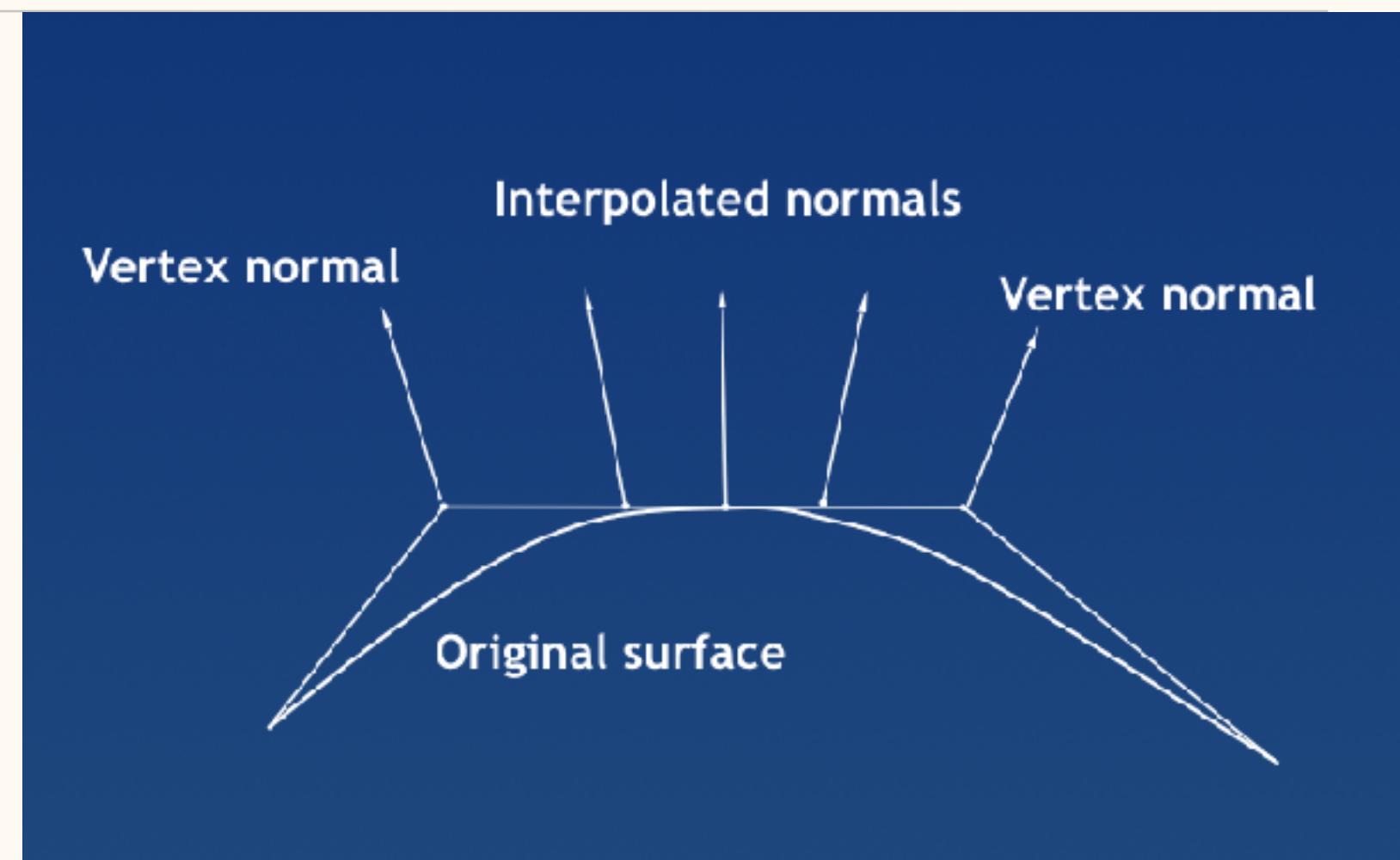
$$N = (M^{-1})^T$$

- For orthogonal transformation (rotation only), the normal vector transformation is as same as geometric transformation.

$$N = (M^T)^T = M$$

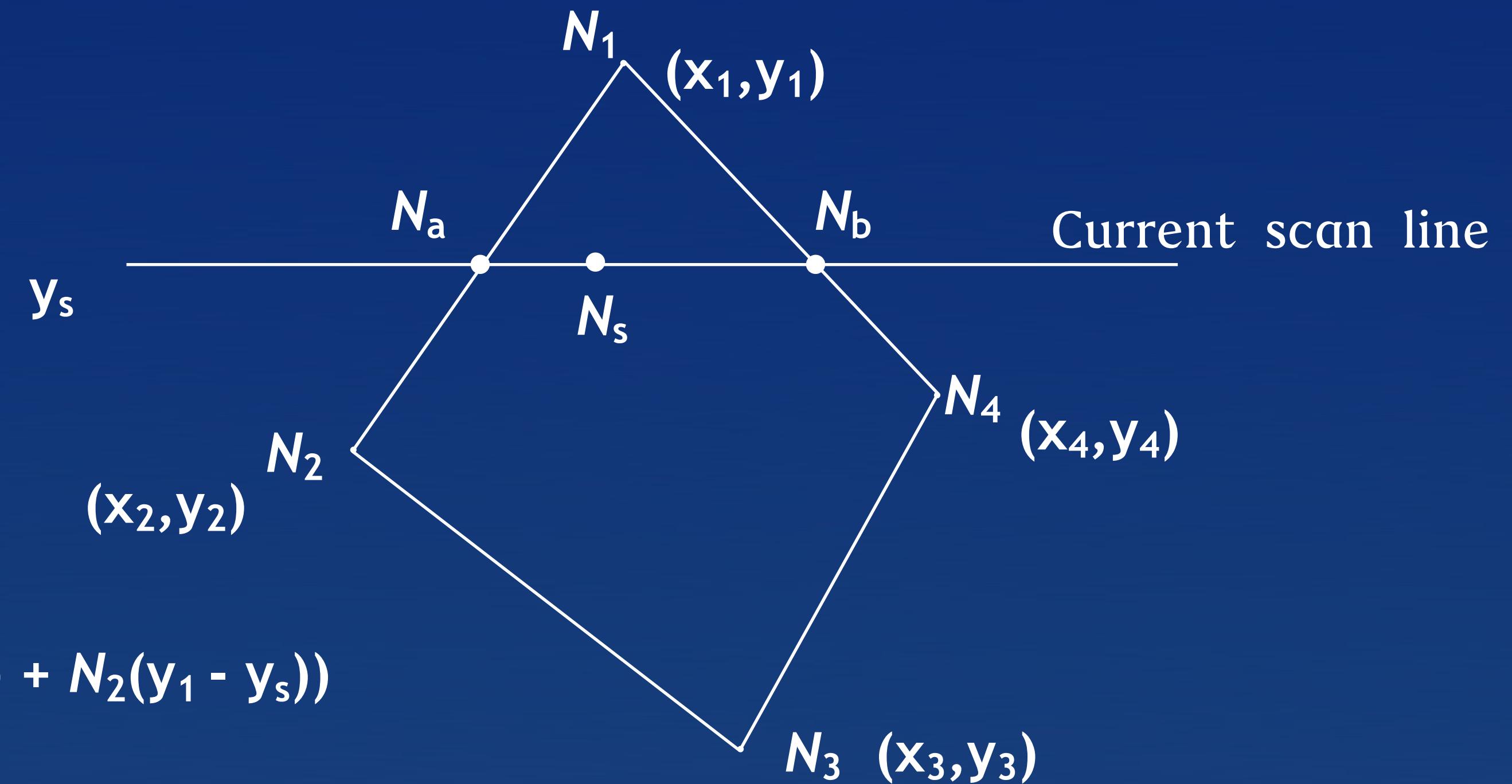
Phong Shading

- 為了解決 Gouraud shader 的缺點與提供 Phong reflection model 中 specular 的計算，Bui-Tuong Phong (1975a) 提出 Phong shading 的演算法.
- 要點：
 - 使用 vertex normal 取代顏色做為 rasterization 時內差的參數.
 - 由內差出的法向量，在每個像素渲染時帶入 Phong reflection model 公式計算光影.
- 因為原始 Shader 架構設計並沒有設計專屬法向量暫存器，因此，法向量資料的傳遞需要使用到貼圖座標暫存器



Phong Shading

- Phong, 1975a



$$N_a = \frac{1}{y_1 - y_2} (N_1(y_s - y_2) + N_2(y_1 - y_s))$$

$$N_b = \frac{1}{y_1 - y_4} (N_1(y_s - y_4) + N_4(y_1 - y_s))$$

$$N_s = \frac{1}{x_b - x_a} (N_a(x_b - x_s) + N_b(x_s - x_a))$$

Use vertex normals for rasterization
Do lighting in pixel shader
Per-pixel Lighting

Phong-Blinn - Vertex Shader Constants

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP           : packoffset(c0);      // matrix from local to screen space
    matrix mWorld          : packoffset(c4);      // matrix from local to global space
    float3 mainLightPosition : packoffset(c8);    // position of the "mainLight"
    float3 camPosition     : packoffset(c9);      // camera position
    matrix mWorldInv       : packoffset(c10);     // inverse of world matrix
};
```

Phong-Blinn - Vertex Shader

```
// vertex shader input
struct VS_INPUT
{
    float4 inPos    : POSITION;
    float3 inNorm   : NORMAL;
};

// vertex shader output
struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float3 norm     : TEXCOORD1;
    float3 camDir   : TEXCOORD2;
    float3 lgtDir   : TEXCOORD3;
};
```

Phong-Blinn - Vertex Shader

```
// the vertex shader                                N = (M-1)T
VS_OUTPUT PhongVS(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // convert the vertex from local to global
    float4 a = mul(mWorld, in1.inPos);

    // get the vertex in screen space
    out1.pos = mul(mWVP, in1.inPos);

    // prepare the normal and camera vector for pixel shader
    out1.norm = normalize(mul(in1.inNorm, (float3x3) mWorldInv));
    out1.camDir = normalize(camPosition.xyz - a.xyz);
    out1.lgtDir = normalize(mainLightPosition - a.xyz);

    return out1;
}
```

Phong-Blinn - Pixel Shader

```
// constants
cbuffer cbPerObject : register(b0)
{
    float4 mainLightColor    : packoffset(c0);    // color of the "mainLight"
    float4 amb               : packoffset(c1);    // ambient component of the material
    float4 dif               : packoffset(c2);    // diffuse component of the material
    float4 spe               : packoffset(c3);    // specular component of the material
    float   shine             : packoffset(c4);    // material shininess
};

// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float3 norm     : TEXCOORD1;
    float3 camDir   : TEXCOORD2;
    float3 lgtDir   : TEXCOORD3;
};
```

```
// the pixel shader
float4 PhongPS(PS_INPUT in1) : SV_TARGET0
{
    // be sure to normalize the vectors
    float3 camDir = normalize(in1.camDir);
    float3 normDir = normalize(in1.norm);
    float3 lgtDir = normalize(in1.lgtDir);
    float3 halfDir = normalize(lgtDir + camDir);

    // check light color
    float3 lgtC = mainLightColor.rgb;

    // (N dot L)
    float diff = saturate(dot(normDir, lgtDir));

    // (N dot H)^n
    float spec = pow(saturate(dot(normDir, halfDir)), shine);

    // Phong-Blinn reflection model
    float4 rgba;
    rgba.rgb = 0.5*lgxC*amb + lgtC*(dif.rgb*diff + spe.rgb*spec);
    rgba.a = dif.a;

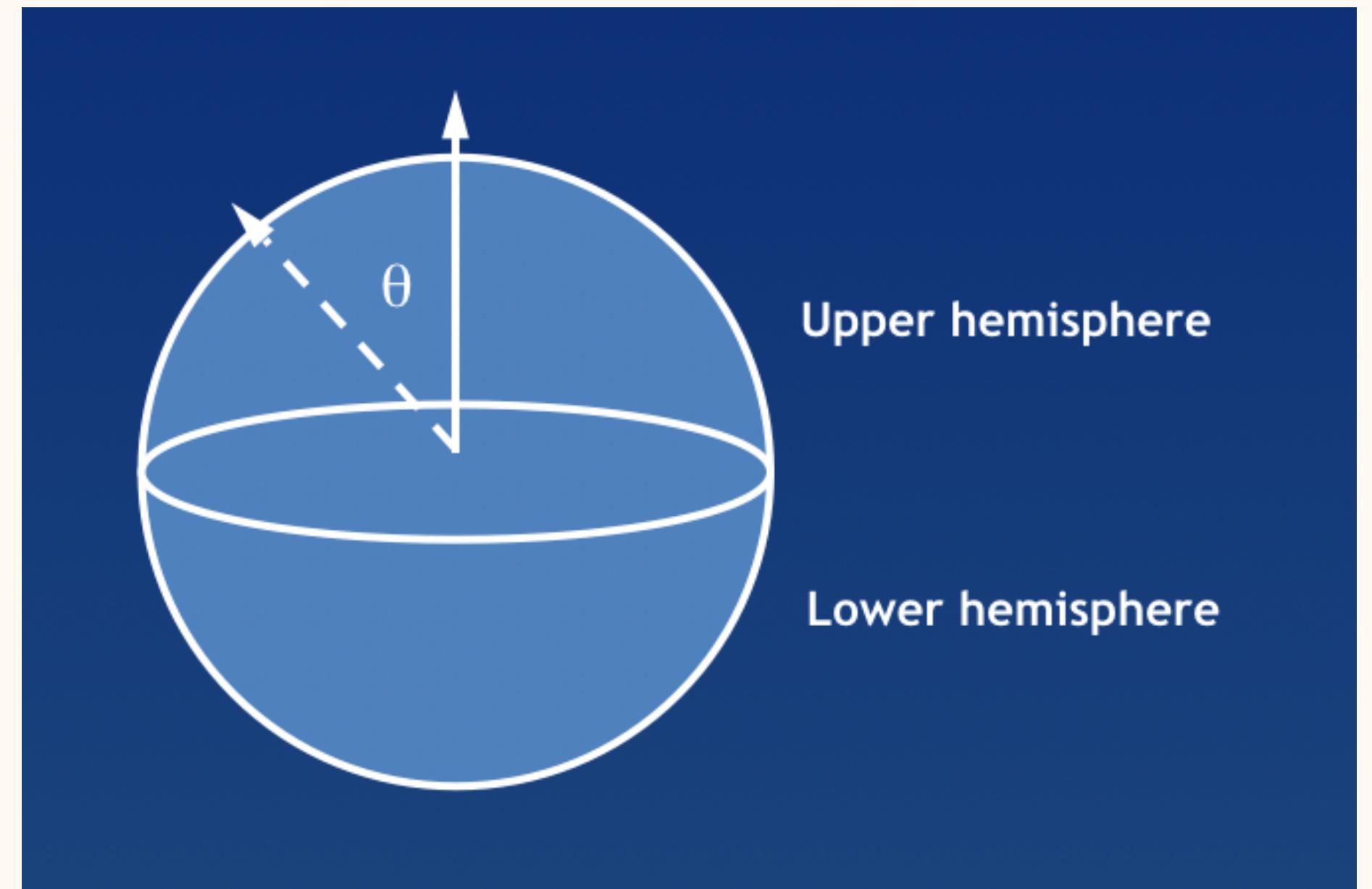
    return rgba;
}
```

Some Tips for Shader Programming

- Most of the shader programming is focusing on “pixel shader”
- Multiple texture coordinate data channels (at most 8 in DX9) to pass the data to pixel shader
 - The data will be interpolated in primitive processing
 - If it is a unit vector, normalize it before using it in pixel shader.
 - Each texture coordinate data channel is a vector in 4 components.

Ambient Lighting - Hemispherical Lighting

- In real world, the lighting doesn't come from a single source.
- In an outdoor setting, some of it does indeed come straight from the sun, but more comes equally from all parts of the sky, and still more is reflected back from the ground and other surrounding objects.
- Hemispheric lighting simulates the effect of the light coming from all directions :
 - Upper hemisphere
 - Lower hemisphere
- Used for ambient lighting



Implement Hemispherical Lighting

- Use lerp() to blend upper & lower hemisphere with a blend factor
- If your game is using y-direction as up-direction :

```
float4 skyLight = { 0.5, 0.5, 0.5, 1.0 };      // sky lighting
float4 groundLight = { 0.2, 0.2, 0.2, 1.0 };    // ground lighting
float3 upDir = { 0.0, 0.0, 1.0 };                // z is the up direction

float4 imgAmb = lerp(groundLight, skyLight, dot(normDir, upDir)*0.5 + 0.5);
```

Phong-Blinn Vertex Shader (with Texture)

```
// This vertex shader implements Phong shading (Phong-Blinn Model)
//

cbuffer cbPerObject : register(b0)
{
    matrix mWVP          : packoffset(c0);      // matrix from local to screen space
    matrix mWorld         : packoffset(c4);      // matrix from local to global space
    int mainLightType     : packoffset(c8);      // type of the "mainLight"
    float3 mainLightPosition : packoffset(c9);   // position of the "mainLight"
    float3 camPosition    : packoffset(c10);     // camera position
    matrix mWorldInv      : packoffset(c11);     // inverse of world matrix
};
```

```
// vertex shader input
struct VS_INPUT
{
    float4 inPos    : POSITION;
    float3 inNorm   : NORMAL;
    float2 inTex0 : TEXCOORD0;
};
```

```
// vertex shader output
struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float2 tex0    : TEXCOORD0;
    float3 norm     : TEXCOORD1;
    float3 camDir   : TEXCOORD2;
    float3 lgtDir   : TEXCOORD3;
};
```

```
// the vertex shader
VS_OUTPUT PhongVS(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // convert the vertex from local to global
    float4 a = mul(mWorld, in1.inPos);

    // get the vertex in screen space
    out1.pos = mul(mWVP, in1.inPos);

    // prepare the normal and camera vector for pixel shader
    out1.norm = normalize(mul(in1.inNorm, (float3x3) mWorldInv));
    out1.camDir = normalize(camPosition.xyz - a.xyz);
    out1.lgtDir = normalize(mainLightPosition - a.xyz);

    // bypass the texture coordinate
    out1.tex0 = in1.inTex0;

    return out1;
}
```

Phong-Blinn Pixel Shader (with Texture)

```
// This shader implements Phong shading (Phong-Blinn Model)
//

// constants
cbuffer cbPerObject : register(b0)
{
    float4 mainLightColor    : packoffset(c0);    // color of the "mainLight"
    float4 amb               : packoffset(c1);    // ambient component of the material
    float4 dif               : packoffset(c2);    // diffuse component of the material
    float4 spe               : packoffset(c3);    // specular component of the material
    float   shine             : packoffset(c4);    // material shininess
};

// textures and samplers
Texture2D colorMap          : register(t0);
SamplerState colorMapSampler : register(s0);
```

```
// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float2 tex0     : TEXCOORD0;
    float3 norm     : TEXCOORD1;
    float3 camDir   : TEXCOORD2;
    float3 lgtDir   : TEXCOORD3;
};

// the pixel shader
float4 PhongPS(PS_INPUT in1) : SV_TARGET0
{
    // be sure to normalize the vectors
    float3 camDir = normalize(in1.camDir);
    float3 normDir = normalize(in1.norm);
    float3 lgtDir = normalize(in1.lgtDir);
    float3 halfDir = normalize(lgtDir + camDir);

    // check light color
    float3 lgtC = mainLightColor.rgb;

    // (N dot L)
    float diff = saturate(dot(normDir, lgtDir));
```

```
// (N dot H)^n
float spec = pow(saturate(dot(normDir, halfDir)), shine);

float4 skyLight = { 0.5, 0.5, 0.5, 1.0 };      // sky lighting
float4 groundLight = { 0.2, 0.2, 0.2, 1.0 };    // ground lighting
float3 upDir = { 0.0, 0.0, 1.0 };                // z is the updirection

float4 imgAmb = lerp(groundLight, skyLight, dot(normDir, upDir)*0.5 + 0.5);

float4 texColor = colorMap.Sample(colorMapSampler, in1.tex0);

// Phong-Blinn reflection model
float4 rgba;
rgba.rgb = (imgAmb*amb + lgtC*(dif.rgb*diff + spe.rgb*spec))*texColor.rgb;
rgba.a = dif.a*texColor.a;

return rgba;
}
```

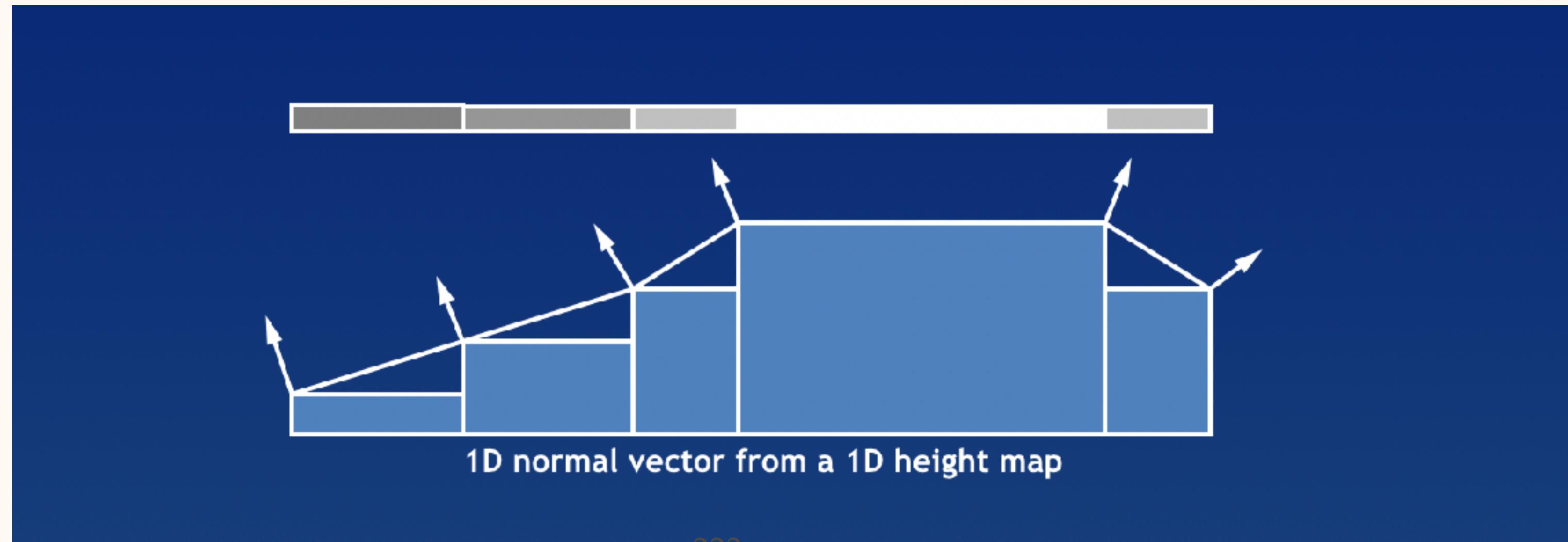
Normal Mapping

Introduction to Normal Mapping

- 1978 Jim Blinn 在 SIGGRAPH 發表的論文
 - “Simulation of Wrinkled Surfaces”, SIGGRAPH'78, pp.286-292
 - “The surface normal is angularly perturbed according to information given in a 2D normal map and this ‘tricks’ a local reflection model.”
 - 沒有改變物體外型，只影響光影效果
- “Bump Map”
 - Save height derivatives (高度變化量) in textures : ($\Delta h/\Delta u$, $\Delta h/\Delta v$)
- “Displacement Map” or “Height Map”
 - Save relative height (高解析模型與低解析模型的高度差) in textures
- 目前流行的做法：“Normal Map”
 - Save normal vector in textures

Introduction to Normal Mapping

- The RGB value of a normal map holds the normalized (x, y, z) normal vector for that particular point.
 - $(r, g, b) = (x, y, z)*0.5 + 0.5$
 - $(x, y, z) = (r, g, b)*2.0 - 1.0$
- A normal map is created by a height map (displacement map).



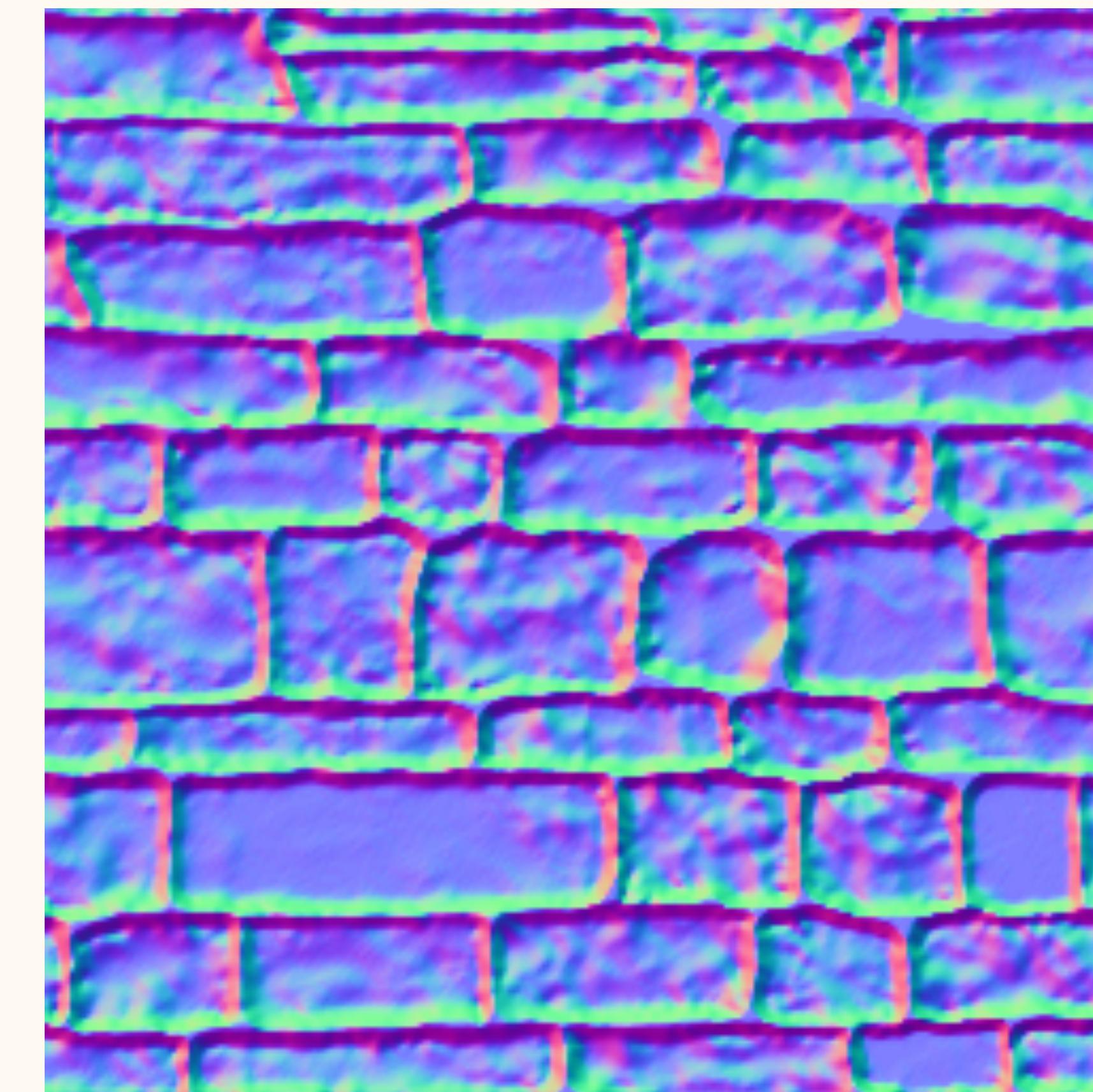
Introduction to Normal Mapping

- 兩種做法
 - Object space normal mapping
 - Normal vectors are stored in object/world space.
 - Geometry solution
 - Tangent space normal mapping
 - Normal vectors are stored in tangent/texture space.

Height Map vs Normal Map



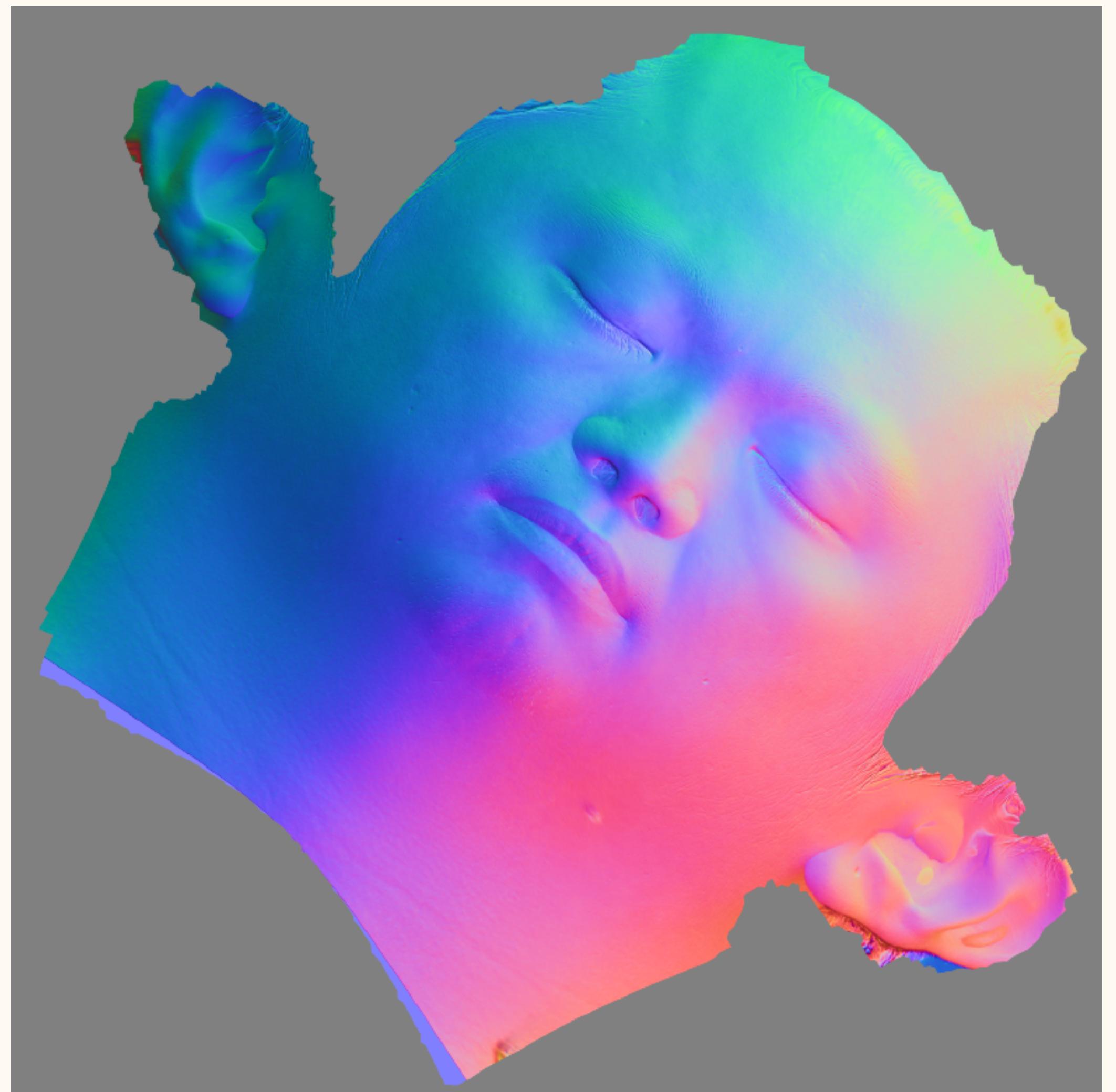
Height Map



Normal Map

Object-space Normal Map

- $(r, g, b) \Rightarrow (N_x, N_y, N_z)$
 - Ranging 0.0 – 1.0
- Color distribution = normal vectors
- Can “See” the geometry on the map



Object-space Normal Map

- In pixel shader : (DX9 HLSL example)
 - `float3 pNorm = 2.0*tex2D(normalMapSampler, tex0) - 1.0;`
 - Scale the vector from range [0,1] to range [-1,1]
 - Apply local to world transformation to the vector
 - Apply the normal data just as we did in per-pixel lighting.
- Pros
 - No need to store the tangent vector for each vertex
- Cons
 - Can not reuse the normal map as tile (無法使用磁磚式貼圖法)
 - 在 pixel shader 做 transformation 計算成本較高
 - 遊戲很少使用 object-space normal map

Tangent-space Normal Map

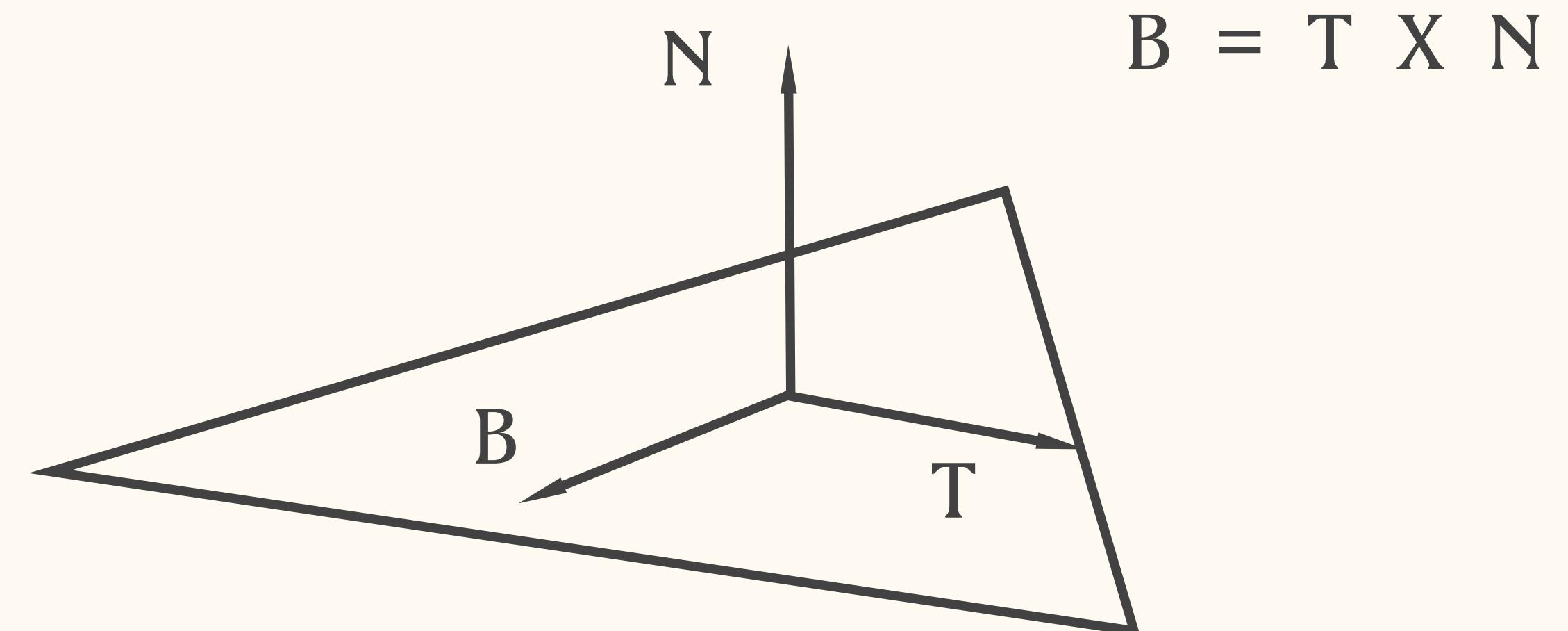
- $(r, g, b) = (N_x, N_y, N_z)$, ranging from 0.0-1.0
- Disturbance of normals in tangent space
- 在每個頂點上需要有 tangent and bi-normal vectors
- Geometry information hiding in tangent vector



Tangent, Bi-normal & Normal Vectors

- The tangent vector can be lots of choices on a plane.
- The plane normal is only one.
- Here we choose U-axis of the texture space as the tangent vector
 - -V-axis of the right-handed texture space is the bi-normal vector

instead of the tangent space.shader

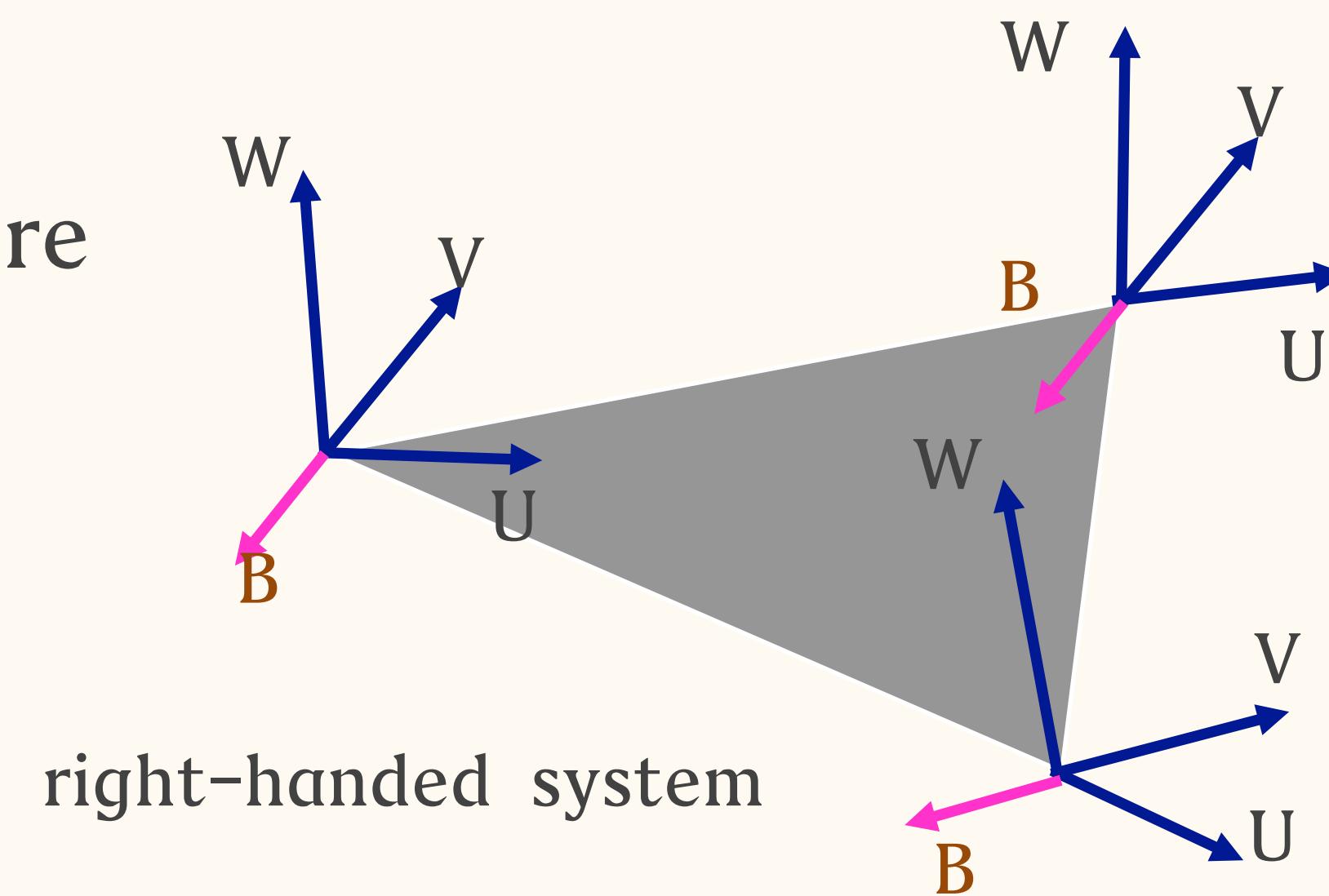


Tangent-space Normal Map

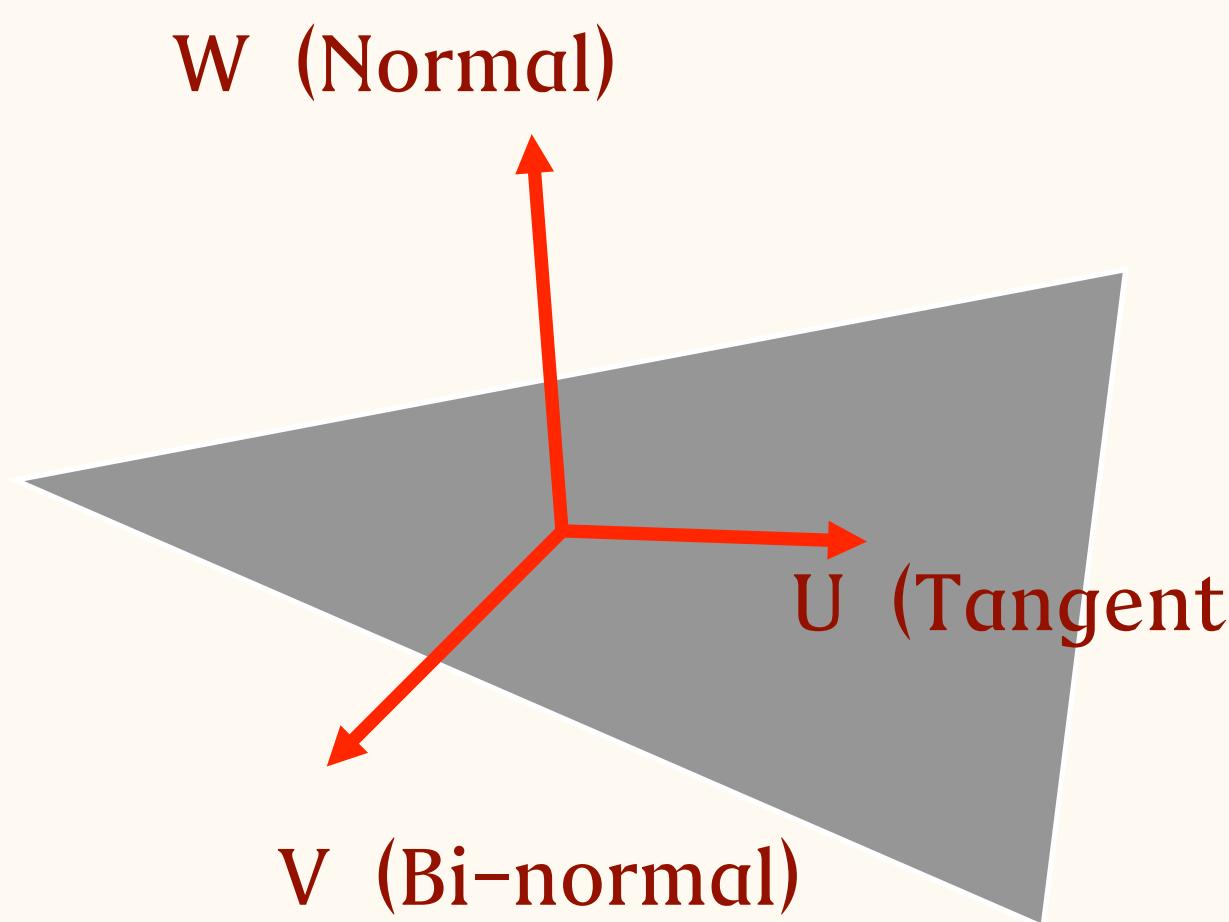
- The normal vector is in tangent space or texture space.
- Need a tangent vector to create the 3x3 matrix to convert the light vector and camera vector from world space to texture space.
- The tangent vector is calculated by the texture coordinates of the triangle vertices.

Basically, in CG the texture space is in right-handed space. But MS sets the texture space in left-handed. The conversion between right-handed to left-handed is :

$$V_{\text{left}} = 1 - V_{\text{right}}$$



Create Texture Space Basis Vectors



W is the normal vector
U is the tangent vector
V is the bi-normal vector

(x,y,z,u,v)

$$\begin{aligned}A_1x + B_1u + C_1v + D_1 &= 0 \\A_2y + B_2u + C_2v + D_2 &= 0 \\A_3z + B_3u + C_3v + D_3 &= 0\end{aligned}$$

Solve the plane equations !
To get the basis vectors for UVW space:

$$\begin{aligned}U &= [\delta x / \delta u, \delta y / \delta u, \delta z / \delta u] = [-B_1/A_1, -B_2/A_2, -B_3/A_3] \\V &= [\delta x / \delta v, \delta y / \delta v, \delta z / \delta v] = [-C_1/A_1, -C_2/A_2, -C_3/A_3] \\W &= U \times V\end{aligned}$$

Create Texture Space Basis Vectors

U.x U.y U.z

V.x V.y V.z

W.x W.y W.z is the matrix used to transform vectors from tangent space to world space .

(for orthogonal matrix $M^{-1} = M^T$)

U.x V.x W.x

U.y V.y W.y

U.z V.z W.z is the matrix used to transform vectors from world space to tangent space.

Phong-Blinn Shader (with Normal Map)

```
// vertex shader input
struct VS_INPUT
{
    float4 inPos      : POSITION;
    float3 inNorm     : NORMAL;
    float2 inTex0     : TEXCOORD0;
    float3 inTangU   : TANGENT;
};

// vertex shader output
struct VS_OUTPUT
{
    float4 pos        : SV_POSITION;
    float2 tex0       : TEXCOORD0;
    float3 camDir     : TEXCOORD1;
    float3 lgtDir     : TEXCOORD2;
    float3 upDir      : TEXCOORD3;
};
```

Phong-Blinn Shader (with Normal Map)

```
// the vertex shader
VS_OUTPUT PhongNormTangVS(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;
    float4 a = mul(mWorld, in1.inPos);
    out1.pos = mul(mWVP, in1.inPos);

    // compute the 3x3 transformation matrix for world space to the tangent space
    float3x3 wToTangent;
    wToTangent[0] = mul(mWorld, in1.inTangU);
    wToTangent[1] = mul(mWorld, cross(in1.inTangU, in1.inNorm));
    wToTangent[2] = mul(mWorld, in1.inNorm);

    // transform the rest data to tangent space
    out1.lgtDir = normalize(mul(wToTangent, mainLightPosition.xyz - a.xyz));
    out1.camDir = normalize(mul(wToTangent, camPosition.xyz - a.xyz));
    out1.upDir = normalize(mul(wToTangent, float3(0.0, 0.0, 1.0)));

...
}
```

Phong-Blinn Shader (with Normal Map)

```
// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float2 tex0     : TEXCOORD0;
    float3 camDir   : TEXCOORD1;
    float3 lgtDir   : TEXCOORD2;
    float3 upDir    : TEXCOORD3;
};

// the pixel shader
float4 PhongNormTangPS(PS_INPUT in1) : SV_TARGET0
{
    // be sure to normalize the vectors
    float3 camDir = normalize(in1.camDir);
    float3 lgtDir = normalize(in1.lgtDir);
    float3 halfDir = normalize(lgtDir + camDir);
    float3 upDir = normalize(in1.upDir);
```

Phong-Blinn Shader (with Normal Map)

```
// get normal vector
float3 normDir = txNormal.Sample(txNormalSampler, in1.tex0).xyz*2.0 - 1.0;

// (N dot L)
float diff = saturate(dot(normDir, lgtDir));

// (N dot H)^n
float spec = pow(saturate(dot(normDir, halfDir)), shine);

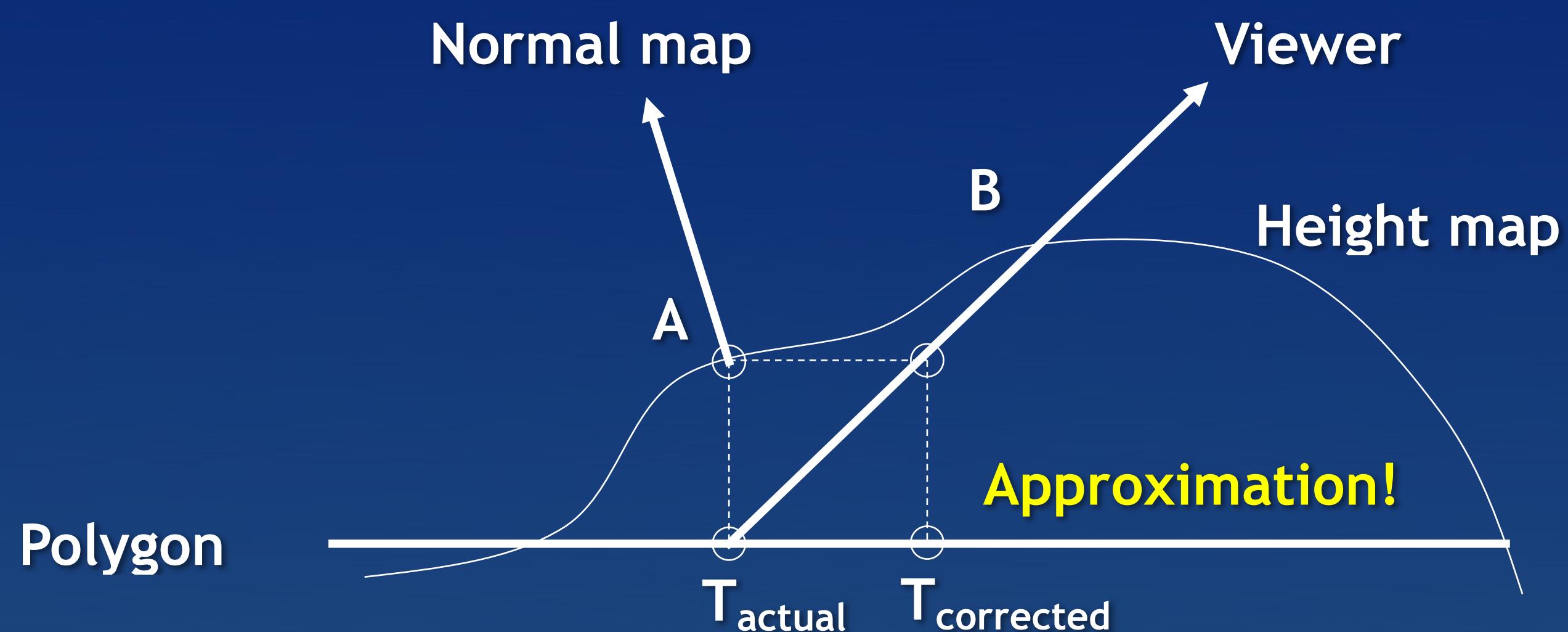
float4 imgAmb = lerp(groundLight, skyLight, dot(normDir, upDir)*0.5 + 0.5);

// Phong-Blinn reflection model
float4 rgba;
rgba = imgAmb*amb + diff*mainLightColor*dif + spec*mainLightColor*spe;
rgba.a = dif.a;

return rgba;
}
```

More Normal Map Solutions

- Parallax Map
 - 2004 by Terry Welsh, “Parallax Mapping with Offset Limiting”
 - Normal map + height map



$$T_{\text{corrected}} = T_{\text{actual}} + V_{(x,y)} \times h_{\text{sb}} / V_{(z)}$$

Or

$$T_{\text{corrected}} = T_{\text{actual}} + V_{(x,y)} \times h_{\text{sb}}$$

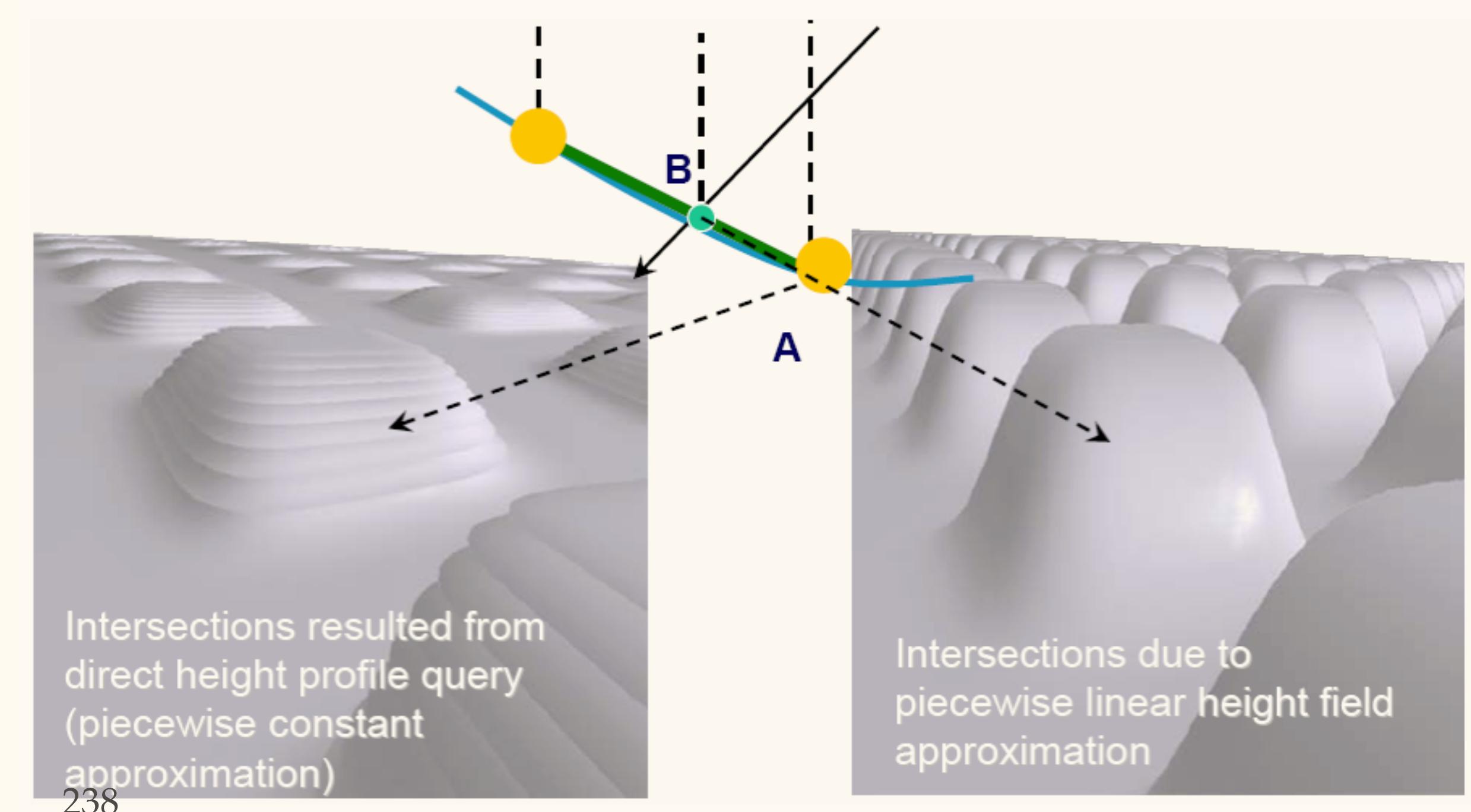
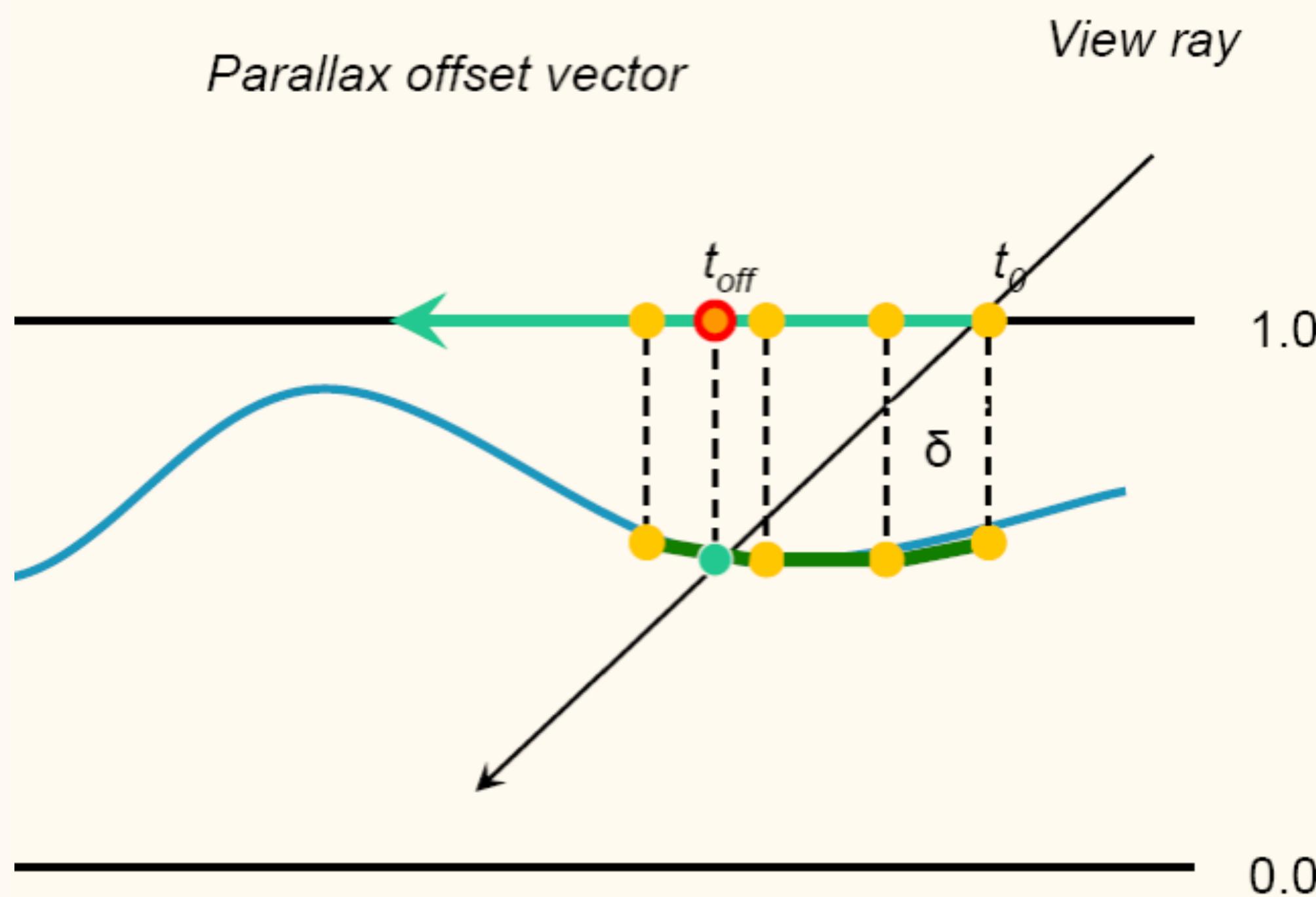
$$h_{\text{sb}} = (h_{xs}) + b$$

s = scale factor

b = bias factor

More Normal Map Solutions

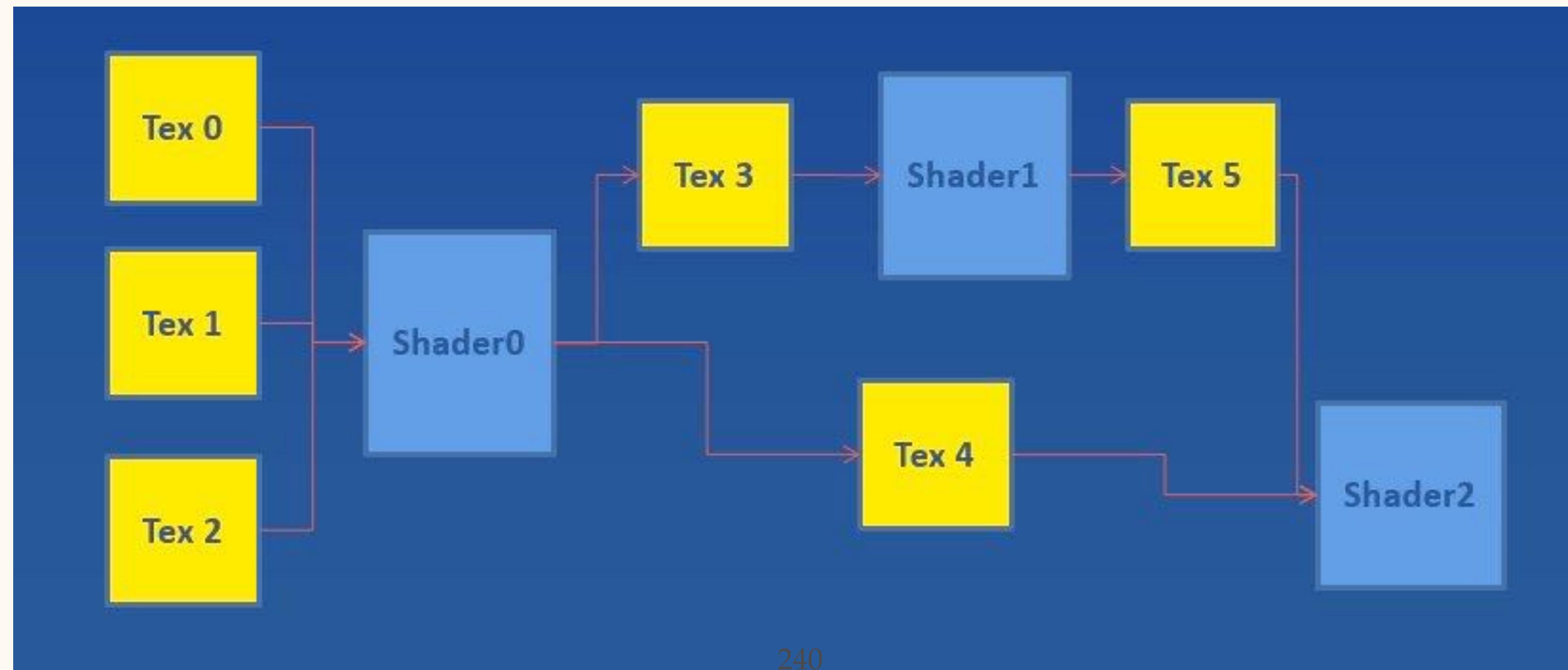
- Parallax Occlusion Map (POM)
 - 2006 by Nalayya Tartachuk at AMD “Toy Shop” Demo
 - “Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows”
 - GDC 2006 and SIGGRAPH 2006



Multi-pass Rendering

Multi-pass Rendering

- Shader 是一段高度平行計算且單一功能的程式碼，只能執行在 GPU 上
- Textures 可視為儲存在視訊記憶區的一塊記憶區 (Buffers, arrays)，只能由 GPU 調用
- Textures 是 Shader 主要輸入資料來源之一
- Shader 可將結果輸出至多組 Textures (rendering targets)



Multi-pass Rendering

- 我們可以將程式分成多個 shaders (function blocks)，使用 textures 作為 data flow 管理 (memory blocks).
- 在 CPU 上執行的主程式使用 D3D/OpenGL draw command 觸發 shader 執行
- 特性：
 - 高度平行化 (high performance)
 - 低延遲 (low latency)
 - Vertex shader and pixel shader are gathering data from textures
 - Not broadcasting data to neighboring vertices and pixels
- 可應用於非圖形應用 (GPGPU, General Purpose GPU)
 - 現在可使用 compute shader 或 CUDA 來實現

Multi-pass Rendering Examples

- HDR Rendering
- Dynamic tone mapping
- Depth of field
- Shadow maps
 - Depth map
- Cartoon shader (image solution)
- Screen Space Ambient Occlusion
- Spherical billboard
- Dynamic environment map
- Skin rendering using texture diffusion
- Deferred shading
 - G-buffers

High Dynamic Range Imaging

HDRI

HDR

- Dynamic range (動態對比)
 - 在環境或硬體上可同時出現的最高與最低亮度的比值
- Displayable image (一般數位影像顯示裝置呈現的影像)
 - Low dynamic range (LDR，低動態對比)
 - 2^8
 - Relative luminance (0.0 – 1.0) – 相對亮度
- Natural phenomena (自然光影)
 - High dynamic range (HDR, 高動態對比)
 - Physical quantity for HDR :

Sunlight vs 100-watt bulb	40,000 : 1
Sunlight vs Blue sky	250,000 : 1
100-watt bulb vs Moonlight	25 : 1

HDR in Real World



辦公室內

來自窗戶的間接光源

1/60th sec 快門 (shutter)

f/5.6 光圈 (aperture)

0 ND filters

0dB gain

HDR in Real World



戶外
影子下

1/1000th sec shutter

f/5.6 aperture

0 ND filters

0dB gain

16 times the light as inside

HDR in Real World



戶外

陽光下

1/1000th sec shutter

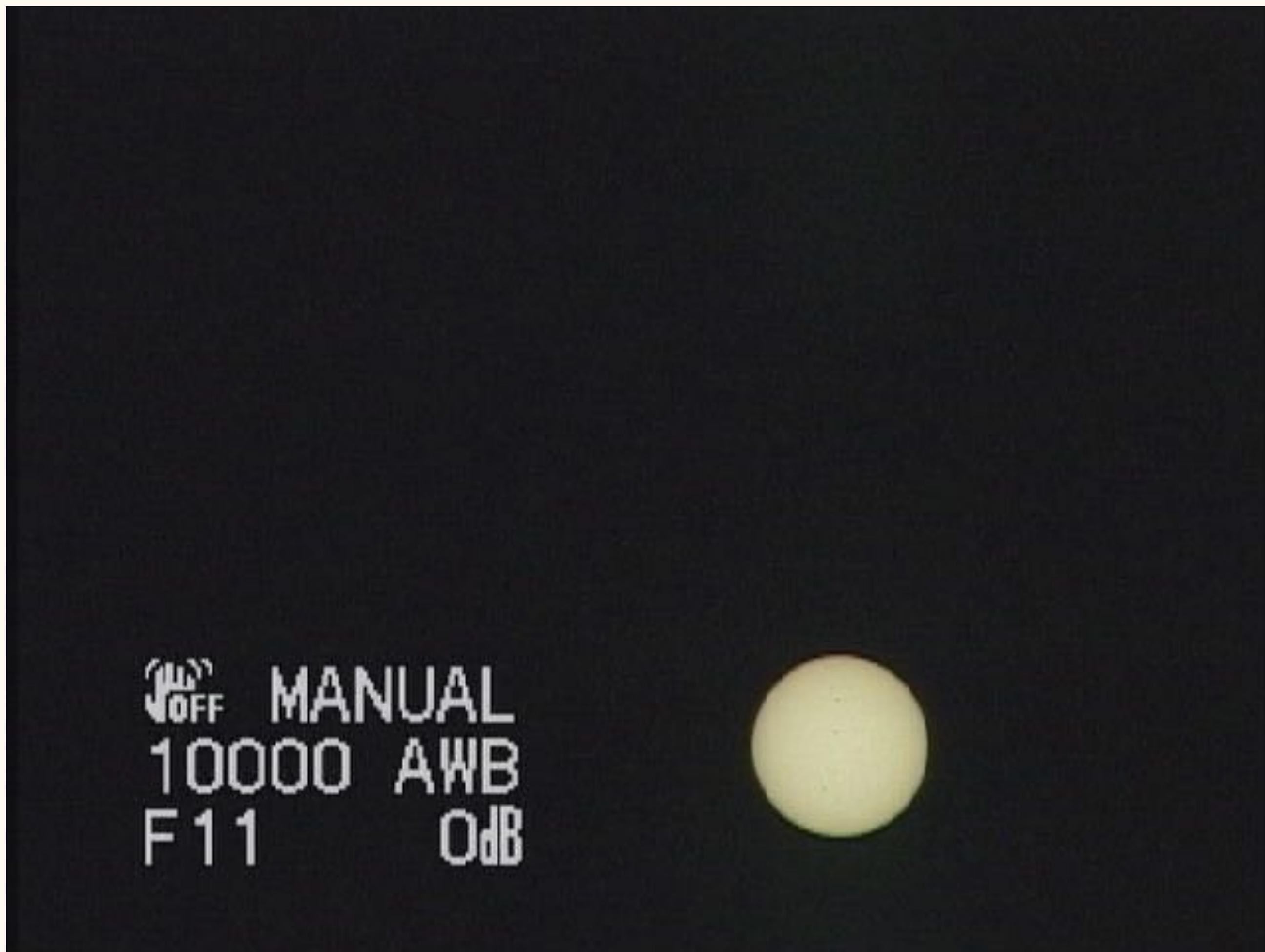
f/11 aperture

0 ND filters

0dB gain

64 times the light as inside

HDR in Real World



直接拍太陽

1/10,000th sec shutter

f/11 aperture

13 stops ND filters

0dB gain

5,000,000 times the light as inside

HDR in Real World



非常暗的房間

1/4th sec shutter

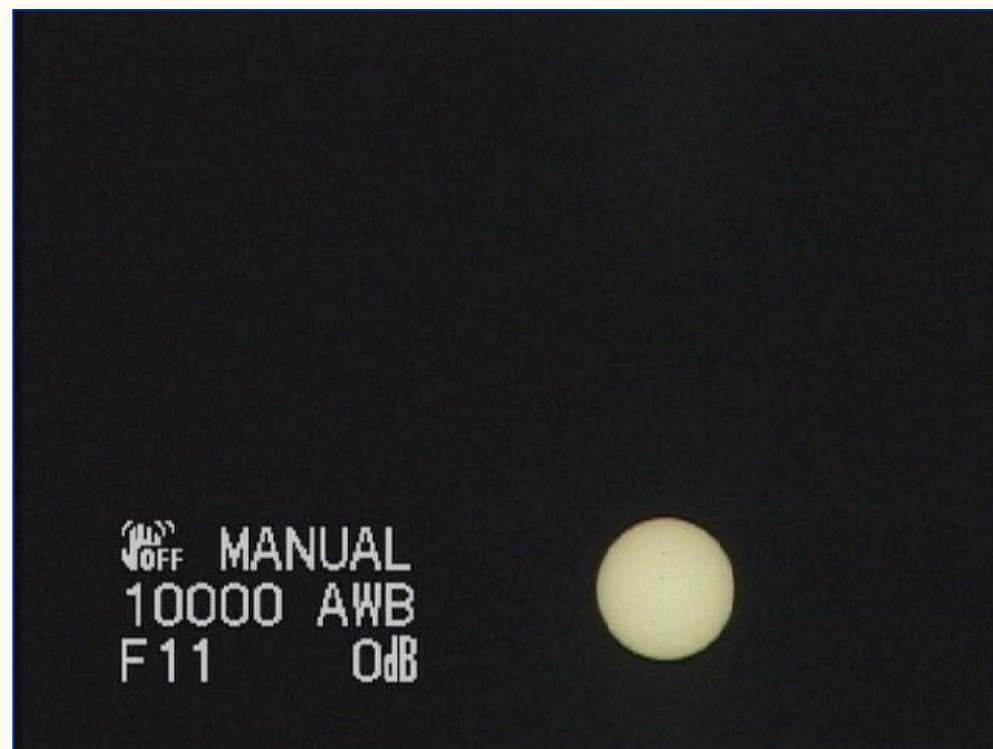
f/1.6 aperture

0 stops ND filters

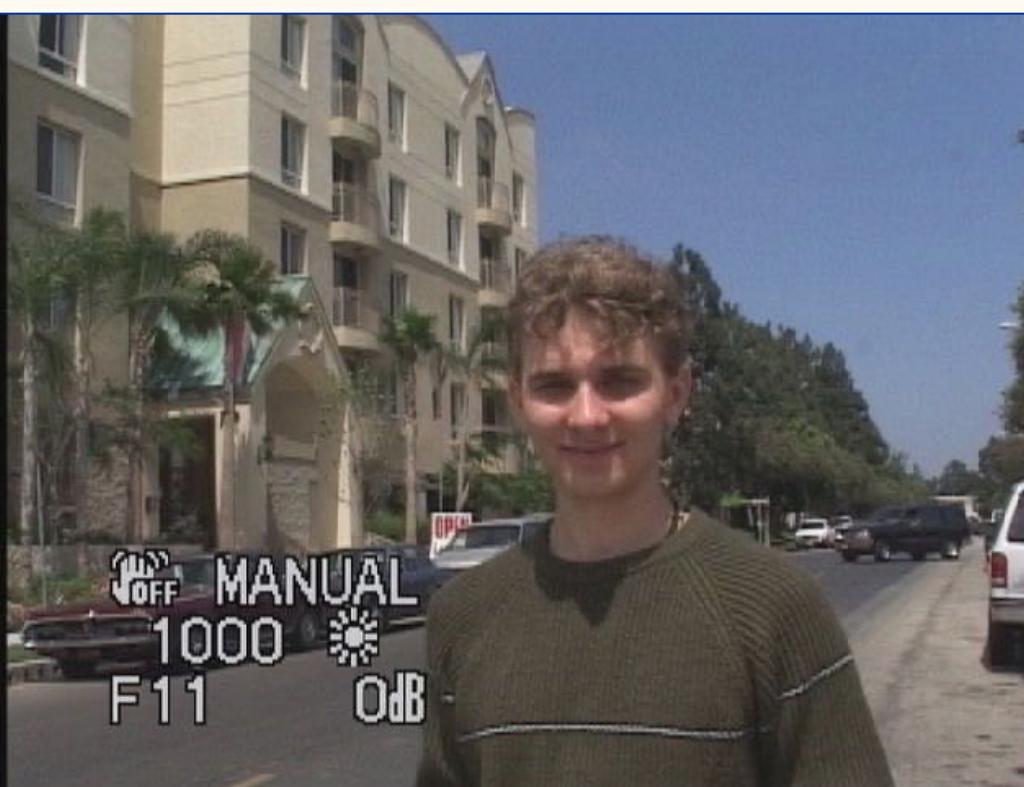
18dB gain

1/1500th the light than inside

HDR in Real World



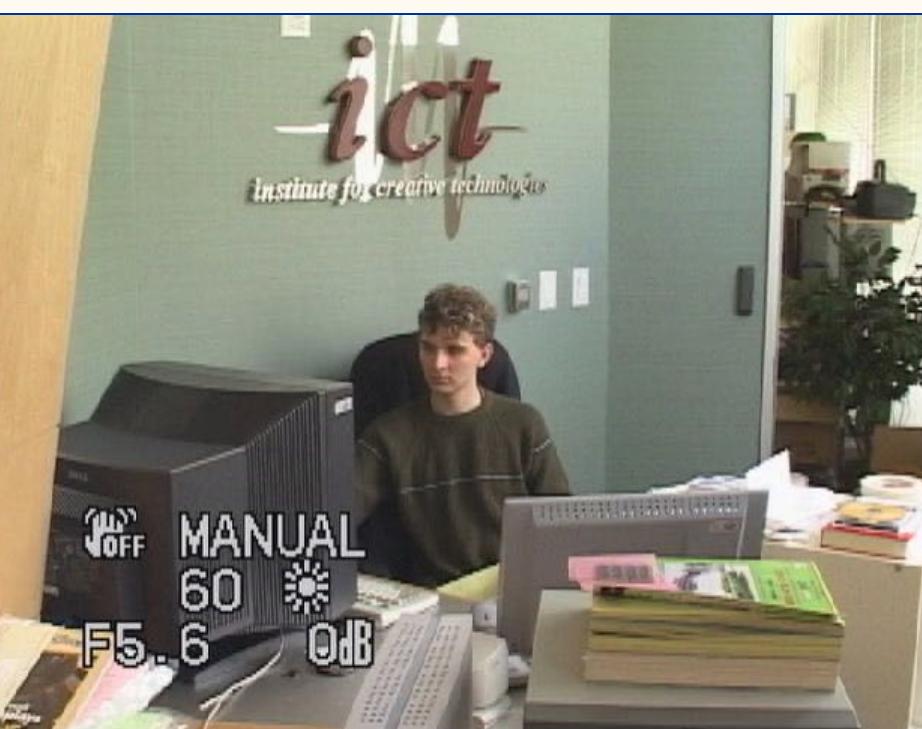
2,000,000,000



400,000



25,000

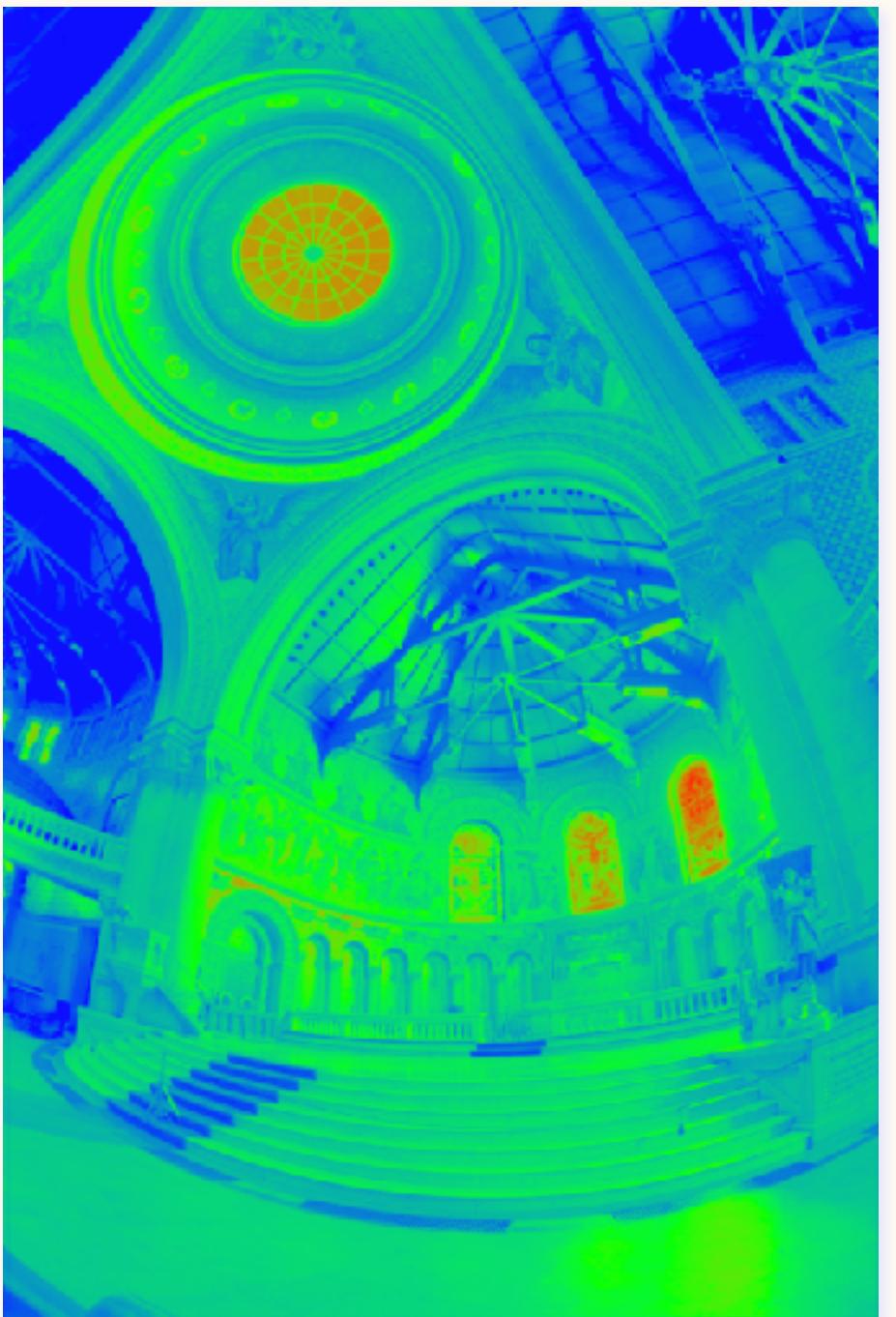
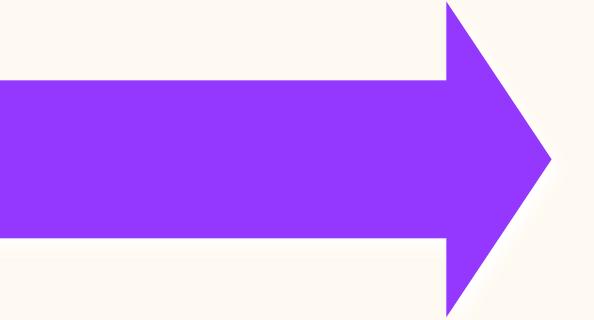


1,500



1

Generate HDR Image



Debevec and Malik, Recovering High Dynamic Range Radiance Maps from Photographs, SIGGRAPH 97

300,000 : 1

The First HDR Example



- RNL
 - "Render Natural Lighting"
 - By Dr. Paul Debevec
 - SIGGRAPH'98 Electronic Theater
 - Rendered by RADIANCE
- In 2006 AMD/ATI made it as a real-time demo for 9700 GPU



The first HDR video

HDR Applications in Games

- HDR Post-processing Effect (HDR 後製特效)
 - Mapping rendered HDR image to LDR for display on screen
 - Dynamic tone mapping
- Image-based Lighting
 - Take HDR image as the global light source
 - 模擬自然光

HDR Effects



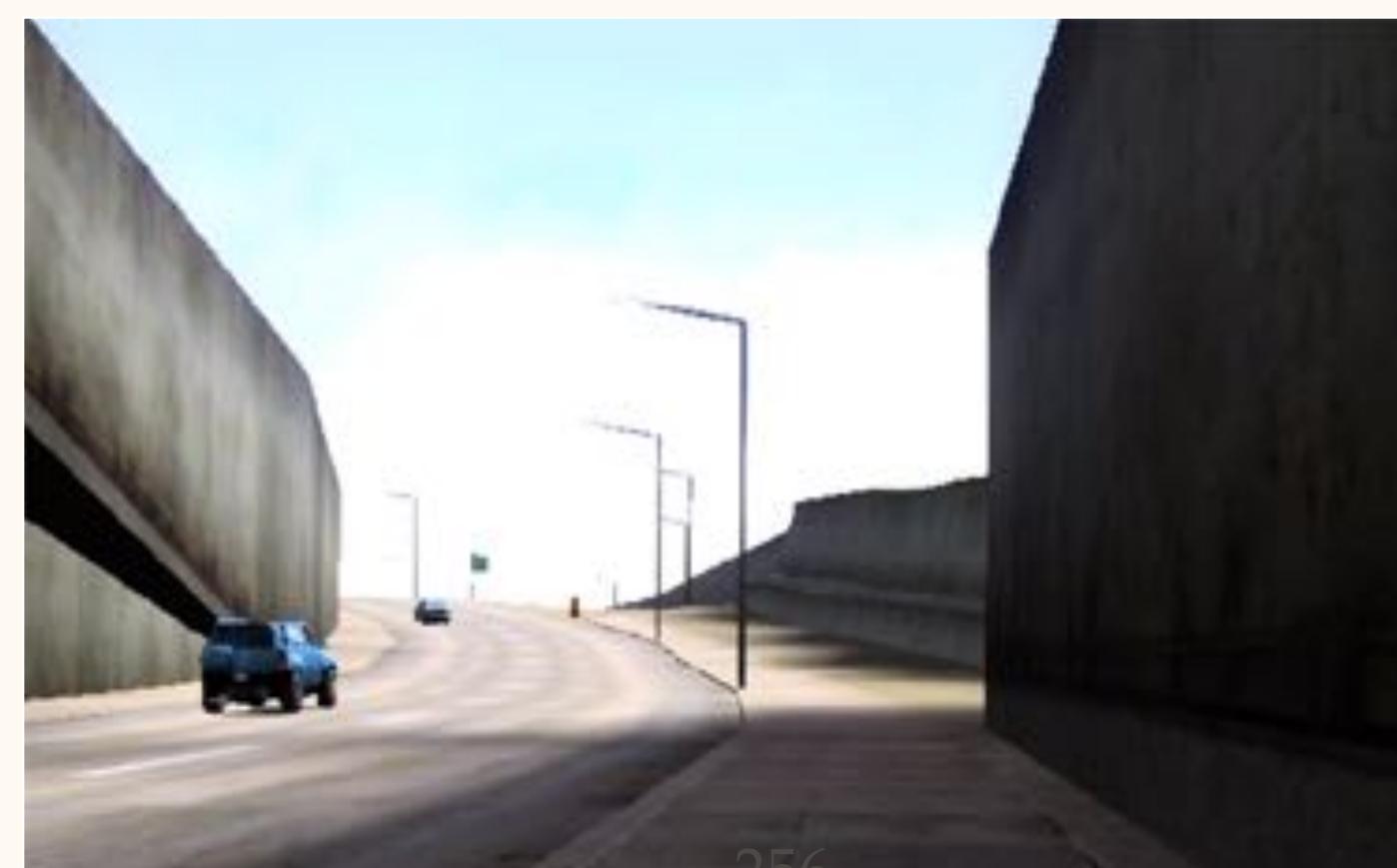
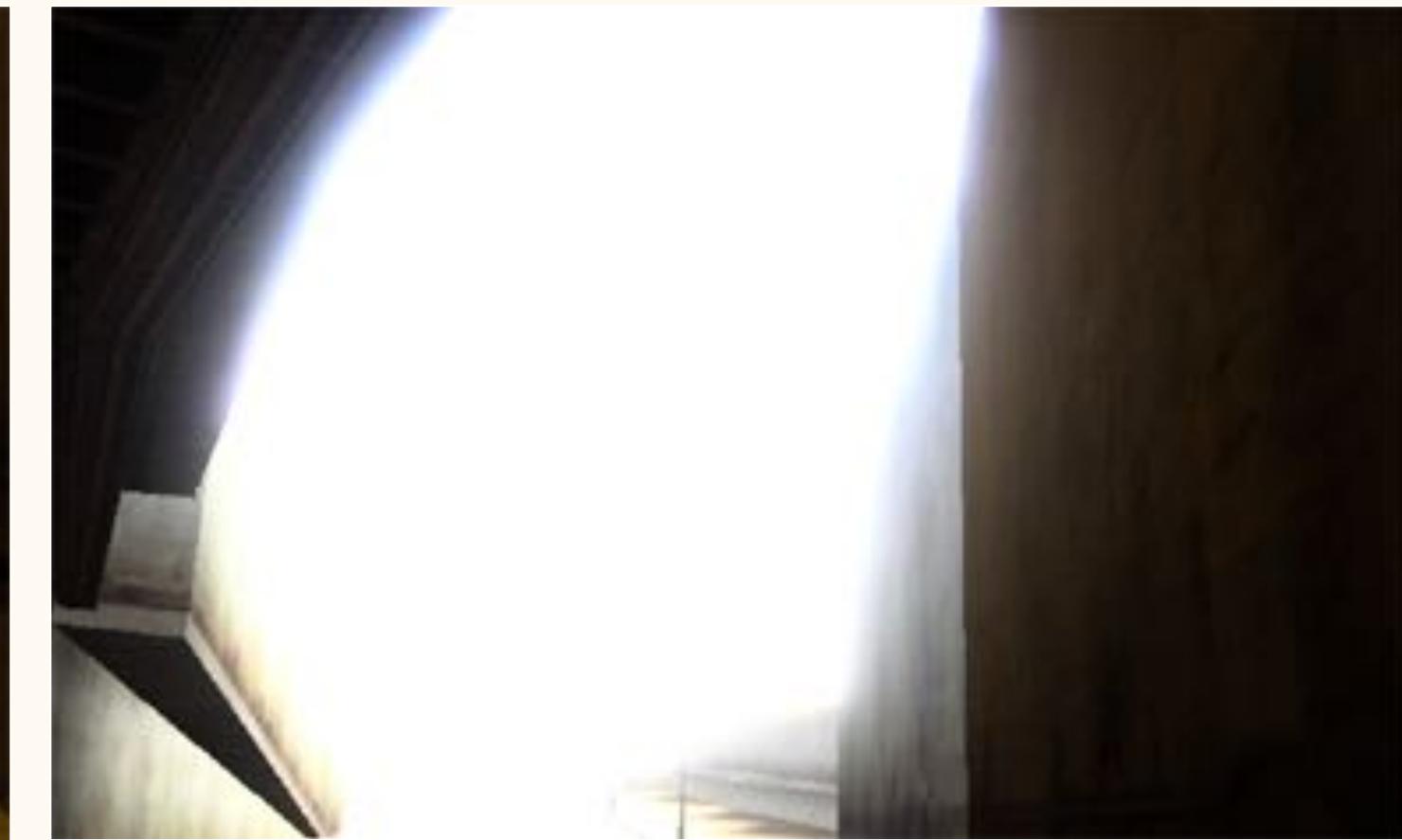
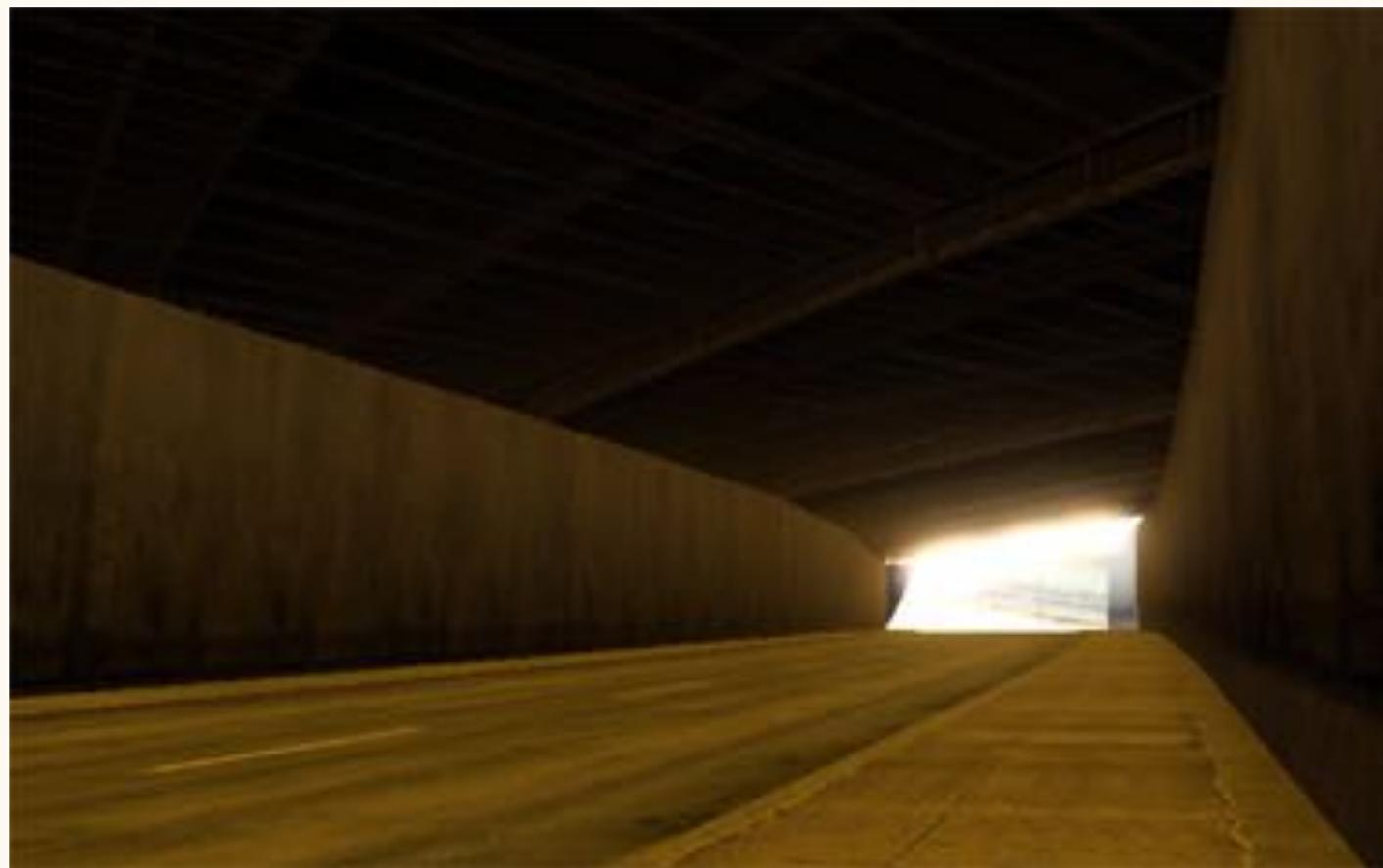
Dazzling Effect

HDR Effects

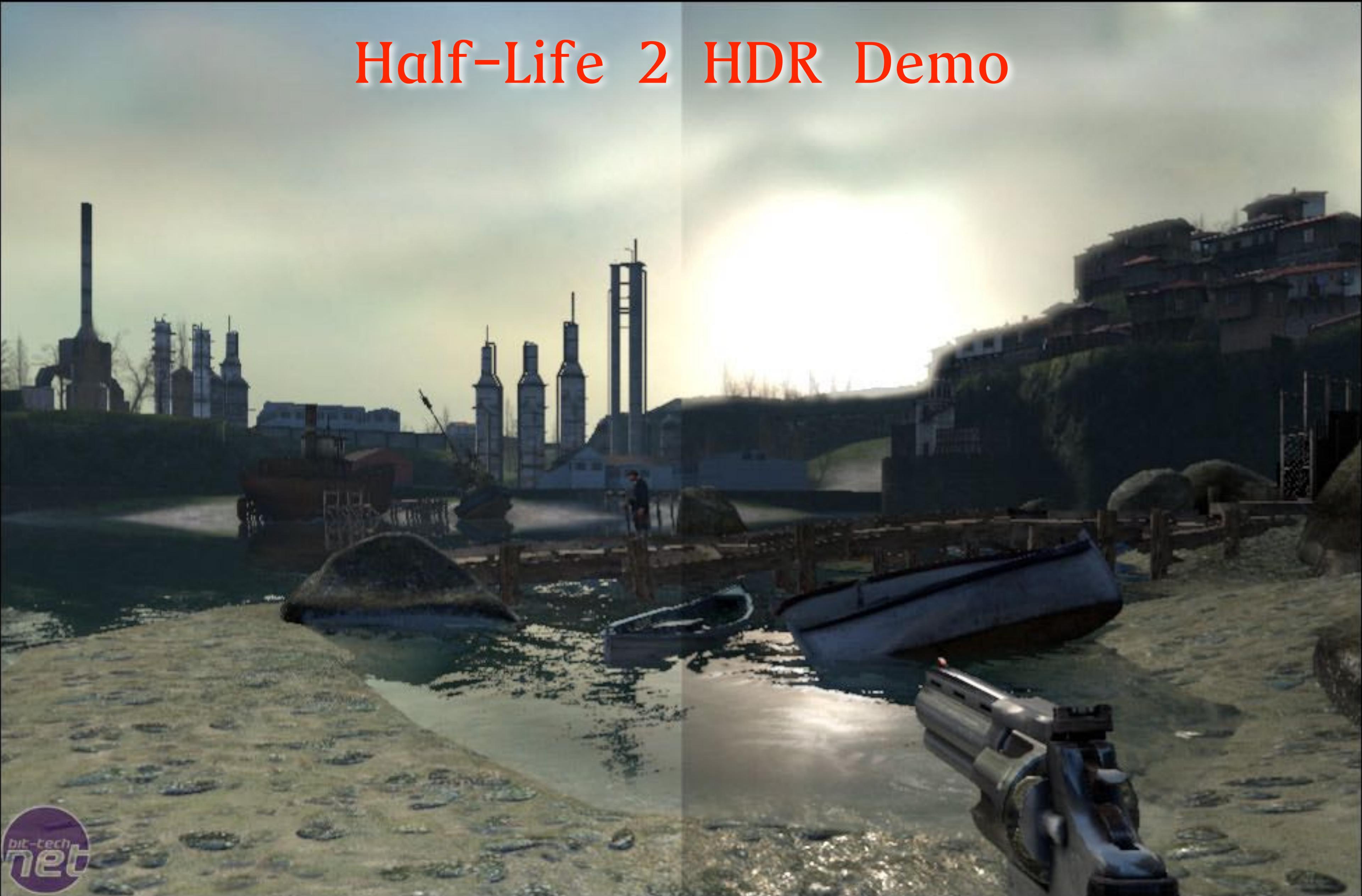




Dynamic Tone Mapping



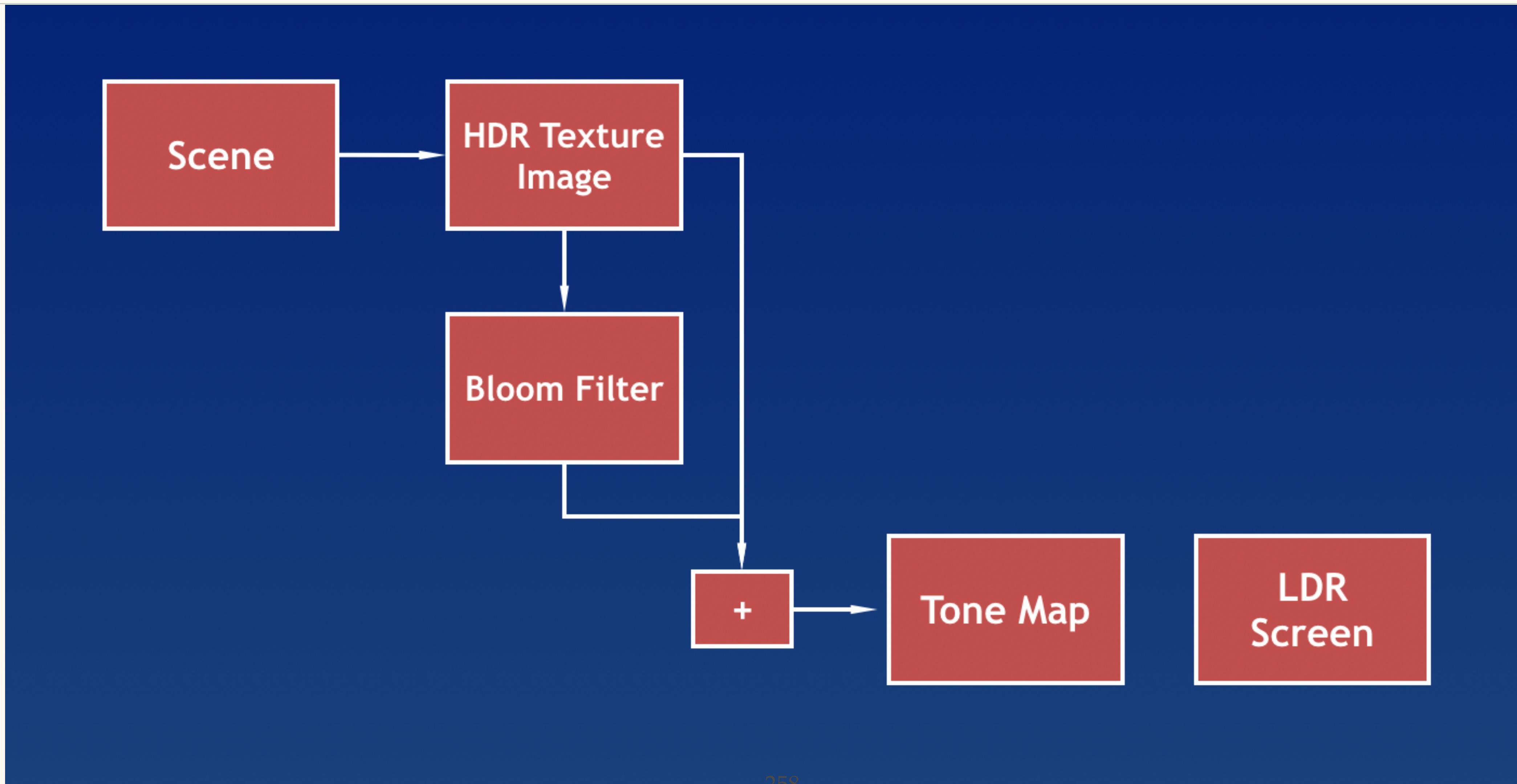
Half-Life 2 HDR Demo



Fixed Aperture

High-Dynamic Range

Real-time HDR Post-processing

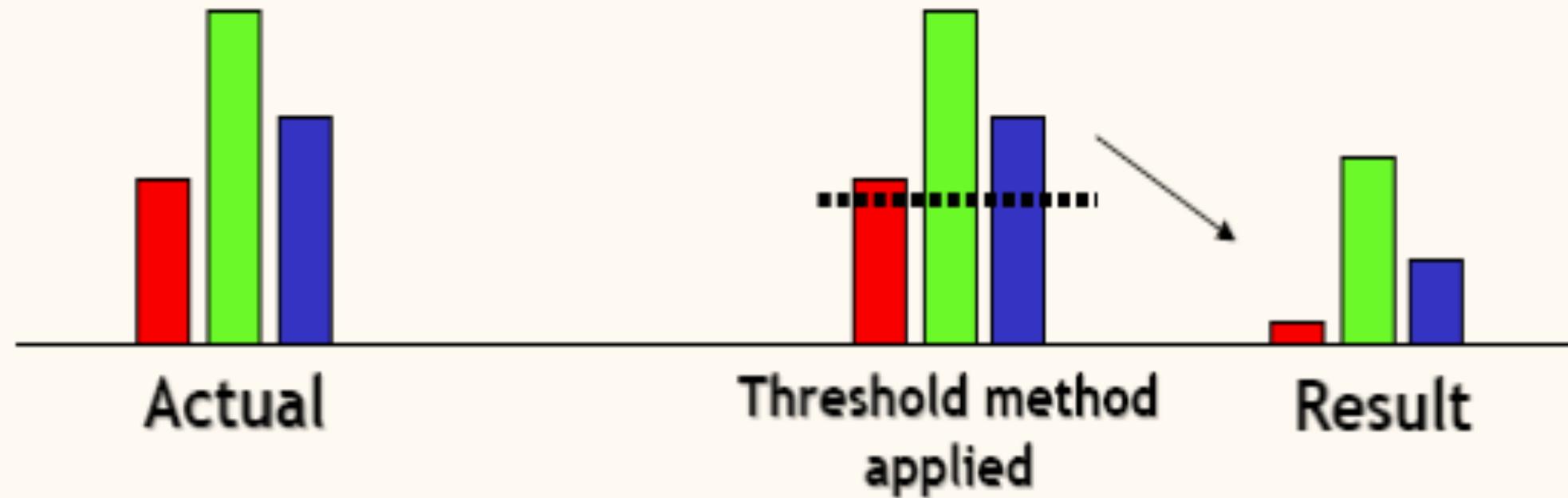


Real-time HDR FX – Step I

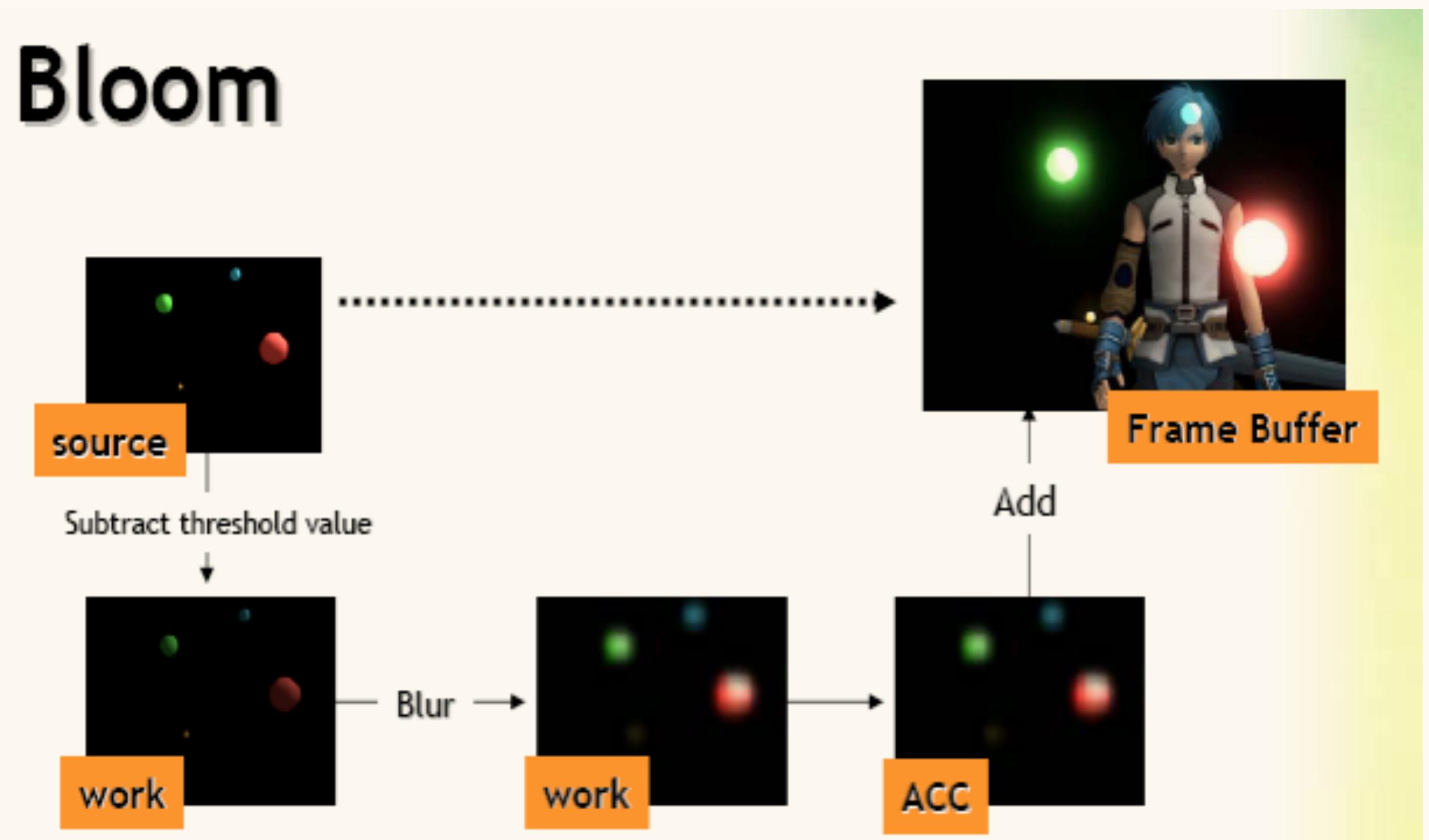
- 將 3D scene 渲染至一張 HDR texture
 - RGB format on screen is LDR：
 - R8G8B8A8
 - (0, 0, 0) ~ (255, 255, 255)
 - HDR：
 - RGB 分別 16-bit 以上
 - floating-point real number
 - 顏色值域不受上限 1.0 的限制

Real-time HDR FX – Step II

- 取出亮部的影像
- 將亮部暈開



Bloom



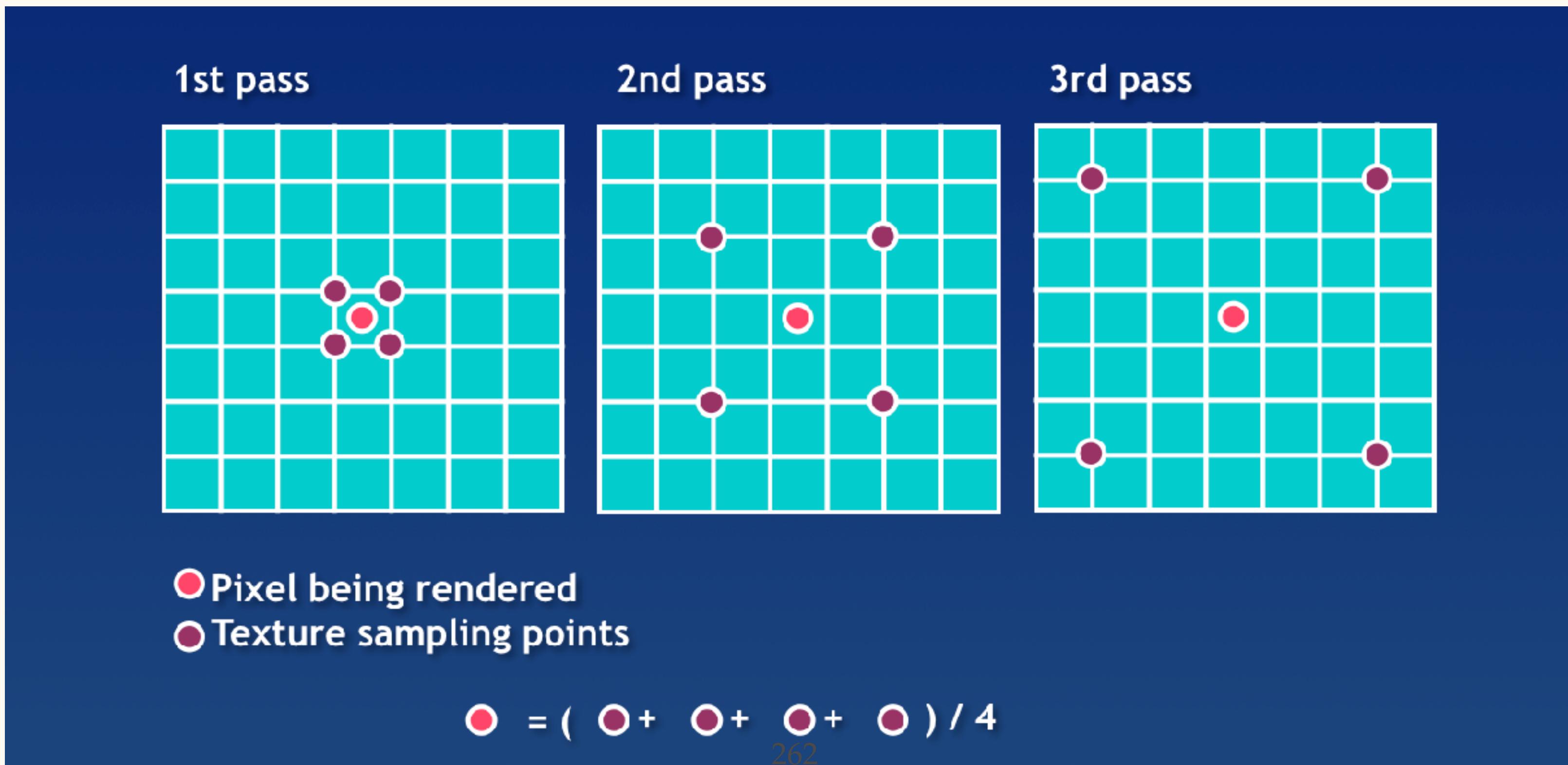
Real-time HDR FX - Step II

- 取出亮部實作：
 - Calculate the luminance of the image pixel
 - Multiple the pixel with luminance : (Shader code)

```
float luminance = dot(color, float3(0.299f, 0.587f, 0.114f));  
float4 result = color * luminance;
```

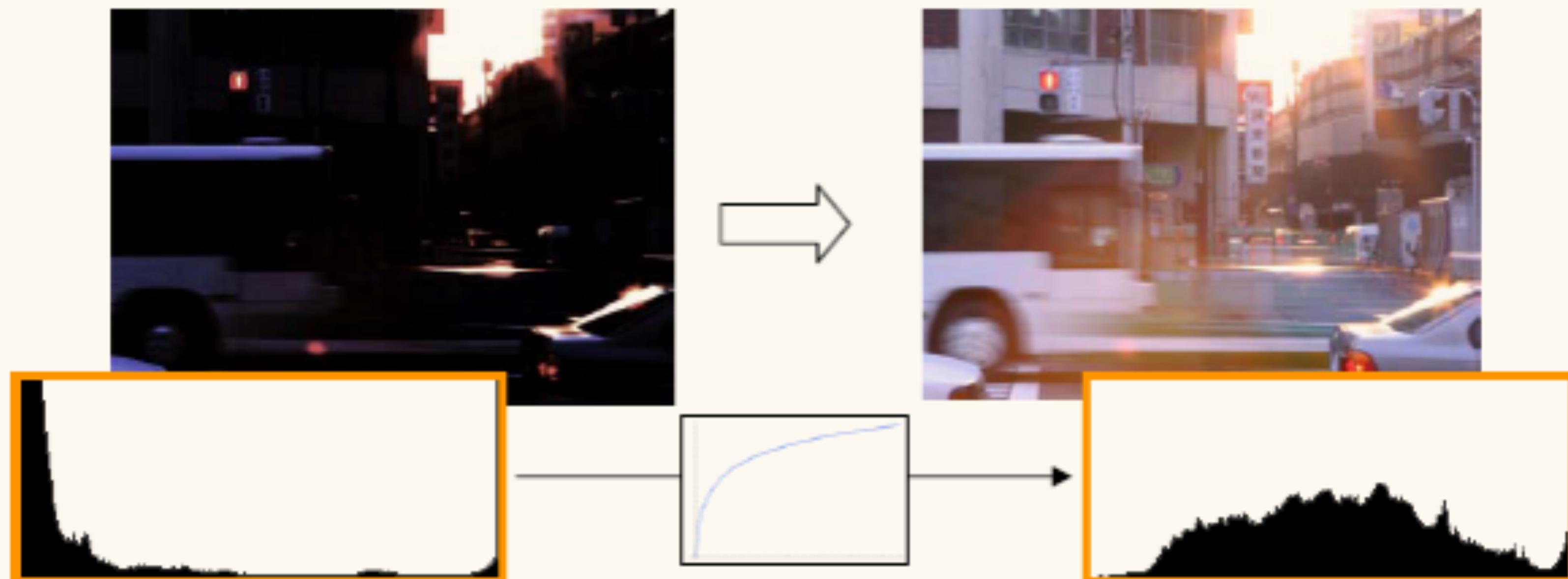
Real-time HDR FX – Step II

- 暈開亮部
 - 使用 Kawase bloom filter，速度快，效果好
 - 重複 8 次，每次調寬取樣的位置，次數越多暈光效果越好



Real-time HDR FX – Step III

- Tone mapping (色調映射)
 - 將 HDR image 轉成可顯示於螢幕的 LDR image

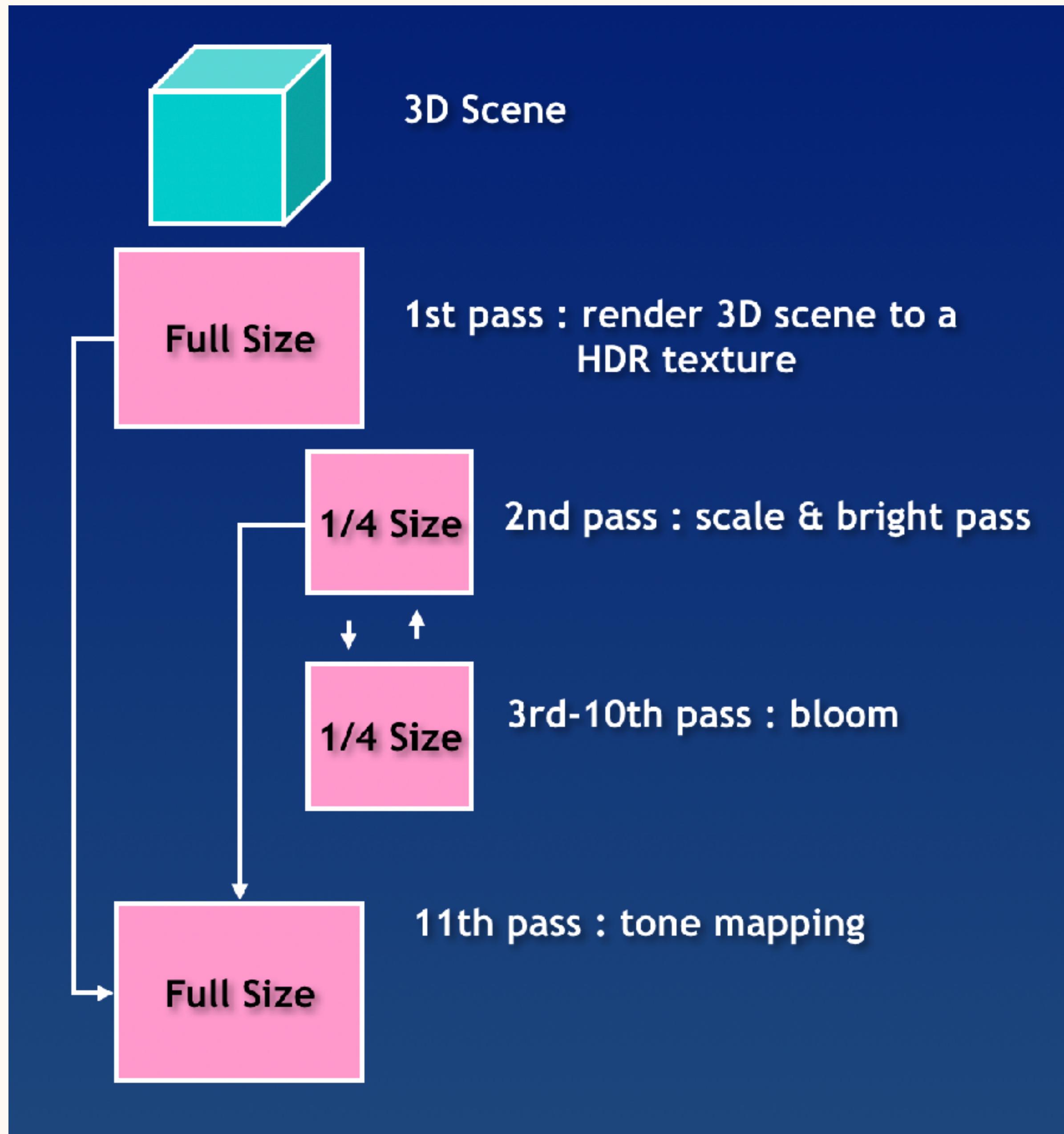


HDR image, visible image
and histogram of intensity

Real-time HDR FX – Step III

- Reinhard Tone Mapping
 - Basic idea :
 - $LDR = HDR / (1 + HDR)$
 - Useful solution : (Lum_{White} is the white color in the game)
 - 只有 global tone mapping，沒有考慮 local tone mapping (模擬沖洗相片的 zone system)

$$LDR = \frac{HDR (1 + HDR / Lum_{White}^2)}{1 + HDR}$$

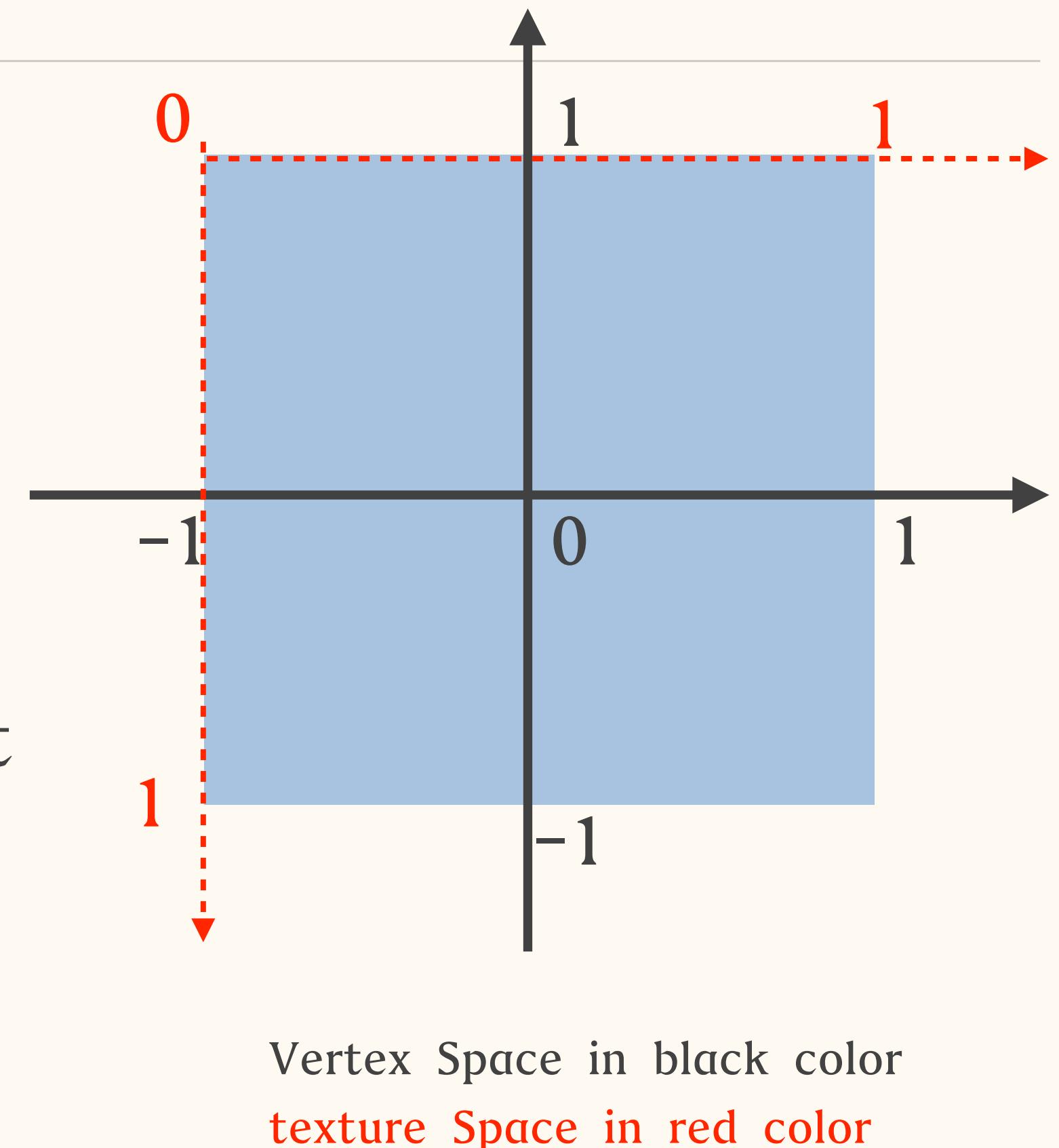


HDR Post-processing



HDR Shaders

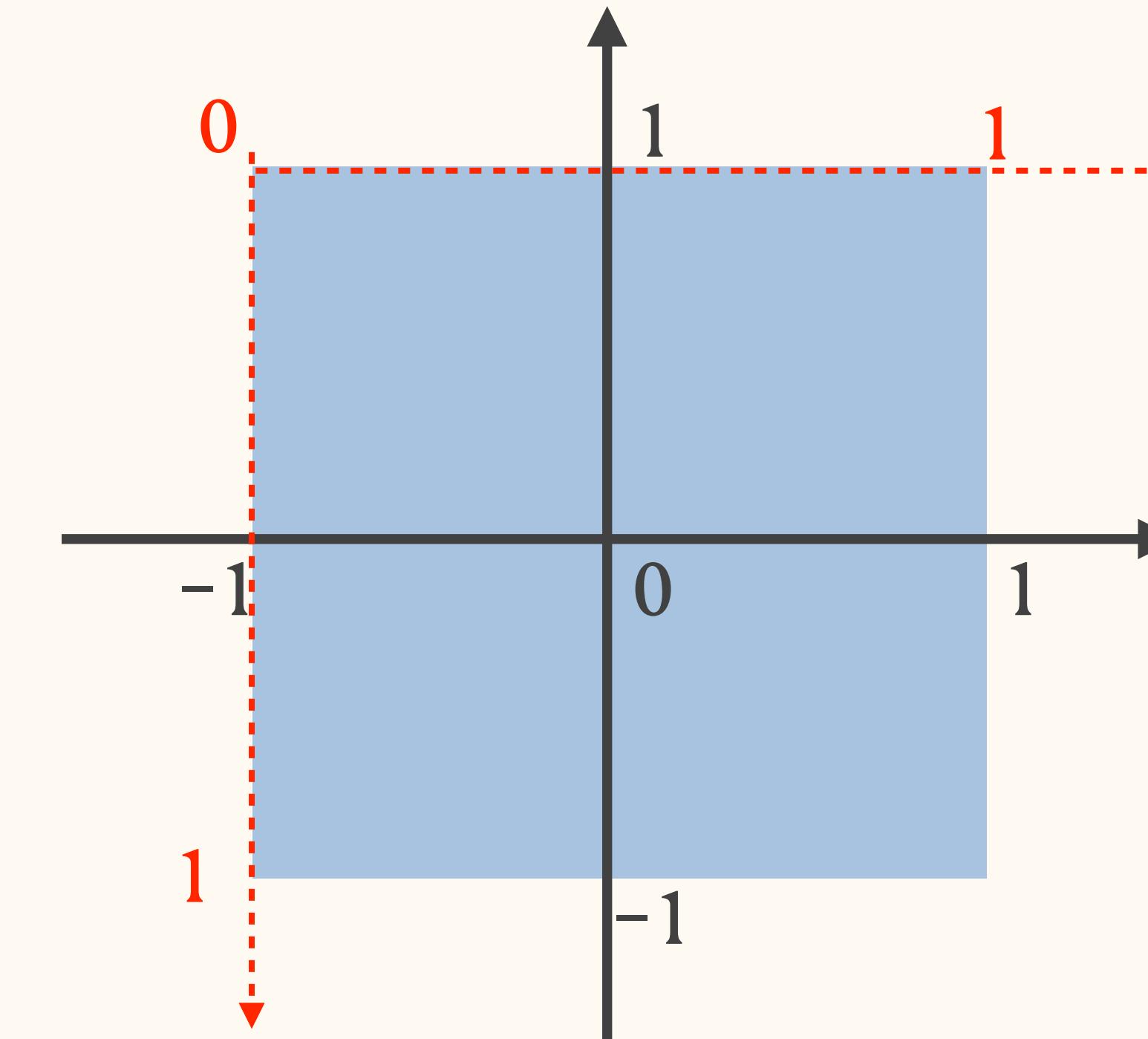
- 使用 3 支 Shader 程式組成 HDR post-processing 系統
 - HDR_Bright_Pass Shader
 - HDR_Bloom_Shader
 - HDR_Tonemap_Shader
- 注意事項：
 - To render a full-screen quad, we need to create at least two triangles with x ranging $(-1.0, 1.0)$, y ranging $(-1.0, 1.0)$, $z = 0.5$ and $w = 1.0$. The texture uv is ranging in $(0.0, 1.0)$



Post-Processing Common Vertex Shader

```
// vertex shader input
struct VS_INPUT
{
    float4 pos    : POSITION;
    float2 tex0   : TEXCOORD0;
};

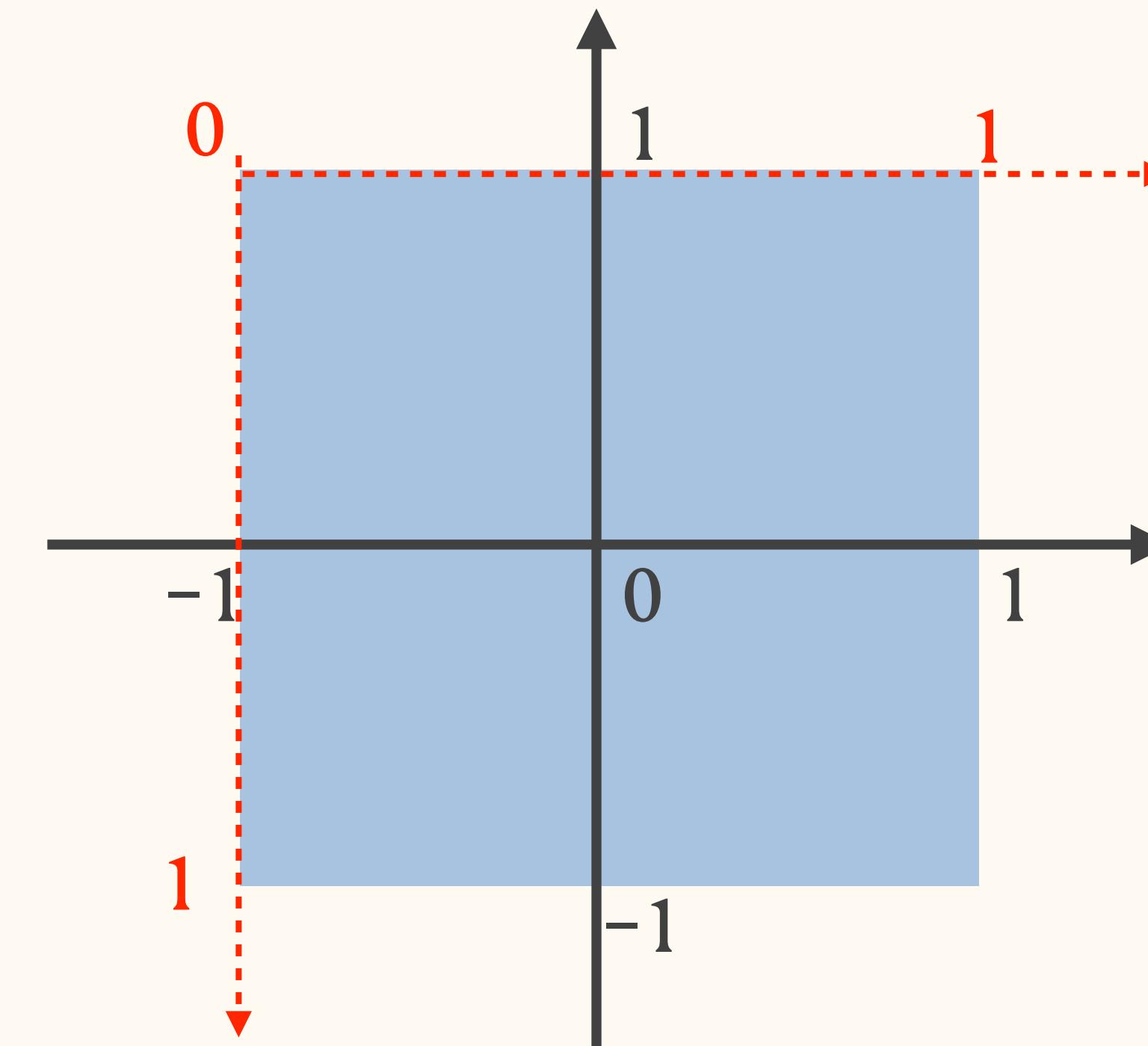
/*-----
 *----- output channels
 *-----*/
struct VS_OUTPUT
{
    float4 pos    : SV_POSITION;
    float2 tex0   : TEXCOORD0;
};
```



Vertex Space in black color
texture Space in red color

Post-Processing Common Vertex Shader

```
/*-----  
 vertex shader code  
-----*/  
VS_OUTPUT VSPostProcessComm(VS_INPUT in1)  
{  
    VS_OUTPUT out1 = (VS_OUTPUT) 0;  
  
    out1.pos.xy = in1.pos.xy;  
    out1.pos.z = 0.5;  
    out1.pos.w = 1.0;  
    out1.tex0 = in1.tex0;  
    return out1;  
}
```



Vertex Space in black color
texture Space in red color

HDR Bright-pass Pixel Shader

```
// pixel shader for extracting the bright part of the image

cbuffer cbFrame : register(b0)
{
    float w2          : packoffset(c0);      // white*white
    float blurThreshold : packoffset(c1);    // intensity threshold for blooming
};

// textures and samplers
Texture2D txColormap      : register(t0);
SamplerState tex2DSampler : register(s0);

// pixel shader input format
struct PS_INTPUT
{
    float4 pos   : SV_POSITION;
    float2 tex0 : TEXCOORD0;
};
```

HDR Bright-pass Pixel Shader

```
/*-----
    pixel shader code
-----*/
float4 PSBrightPass(PS_INTPUT in1) : SV_TARGET
{
    float4 rgba = txColormap.Sample(tex2DSampler, in1.tex0);
    float luminance = dot(rgba.rgb, float3(0.299f, 0.587f, 0.114f));
    if (luminance > blurThreshold) {
        rgba = min(w2.rrrr, rgba*luminance);
    }
    else {
        rgba = float4(0.0, 0.0, 0.0, 1.0);
    }
    return rgba;
}
```

HDR Bloom Vertex Shader

```
cbuffer cbFrame : register(b0)
{
    float2 pixelSize : packoffset(c0);      // pixel size in screen space (1/w, 1/h)
    float fIteration : packoffset(c1);      // bloom iteration number
};

struct VS_INPUT
{
    float4 pos    : POSITION;
    float2 tex0   : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 pos        : SV_POSITION;
    float2 topLeft    : TEXCOORD0;
    float2 topRight   : TEXCOORD1;
    float2 bottomRight : TEXCOORD2;
    float2 bottomLeft  : TEXCOORD3;
};

};
```

```
VS_OUTPUT VSBlom(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    out1.pos.xy = in1.pos.xy;
    out1.pos.z = 0.01;
    out1.pos.w = 1.0;

    float2 halfPixelSize = pixelSize/2.0f;
    float2 dUV = pixelSize.xy*fIteration + halfPixelSize;

    // sample top left
    out1.topLeft = float2(in1.tex0.x - dUV.x, in1.tex0.y + dUV.y);

    // sample top right
    out1.topRight = float2(in1.tex0.x + dUV.x, in1.tex0.y + dUV.y);

    // sample bottom right
    out1.bottomRight = float2(in1.tex0.x + dUV.x, in1.tex0.y - dUV.y);

    // sample bottom left
    out1.bottomLeft = float2(in1.tex0.x - dUV.x, in1.tex0.y - dUV.y);

    return out1;
}
```

HDR Bloom Pixel Shader

```
// textures and samplers
Texture2D txColormap      : register(t0);
SamplerState tex2DSampler : register(s0);

/*-----
    pixel shader input channels
-----*/
struct PS_INPUT
{
    float4 pos          : SV_POSITION;
    float2 topLeft      : TEXCOORD0;
    float2 topRight     : TEXCOORD1;
    float2 bottomRight  : TEXCOORD2;
    float2 bottomLeft   : TEXCOORD3;
};
```

```
/*-----  
    pixel shader code  
-----*/  
float4 PSBloom(PS_INPUT in1) : SV_TARGET  
{  
    float4 addedBuffer = 0.0f;  
  
    // sample top left  
    addedBuffer = txColormap.Sample(tex2DSampler, in1.topLeft);  
  
    // sample top right  
    addedBuffer += txColormap.Sample(tex2DSampler, in1.topRight);  
  
    // sample bottom right  
    addedBuffer += txColormap.Sample(tex2DSampler, in1.bottomRight);  
  
    // sample bottom left  
    addedBuffer += txColormap.Sample(tex2DSampler, in1.bottomLeft);  
  
    // average  
    return addedBuffer *= 0.25f;  
}
```

HDR Tonemap Pixel Shader

```
cbuffer cbFrame : register(b0)
{
    float white2 : packoffset(c0);    // white*white
    float weight : packoffset(c1);    // blur weight
};

// textures and samplers
Texture2D fullColormap      : register(t0);
SamplerState fullSampler     : register(s0);

Texture2D bloomMap          : register(t1);
SamplerState bloomSampler    : register(s1);

// input data format
struct PS_INTPUT
{
    float4 pos   : SV_POSITION;
    float2 tex0  : TEXCOORD0;
};
```

```
/*-----
    pixel shader code
-----*/
float4 PSTonemap(PS_INPUT in1) : SV_TARGET
{
    float4 fullColor = fullColormap.Sample(fullSampler, in1.tex0);
    float4 blurredImage = bloomMap.Sample(bloomSampler, in1.tex0);
    float4 color;

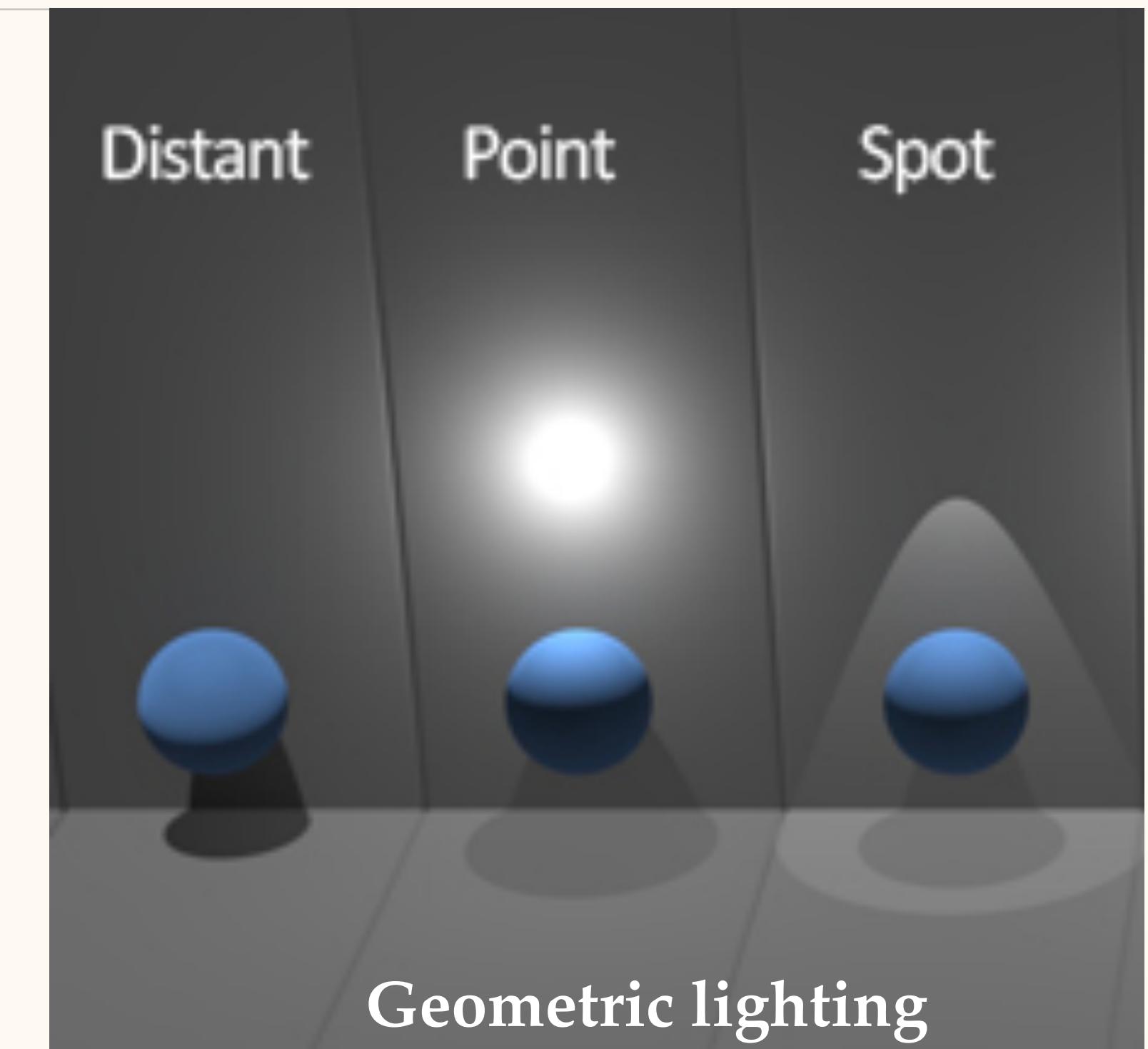
    // add the blurred one to the full sceen image
    color.rgb = fullColor.rgb + blurredImage.rgb*weight;

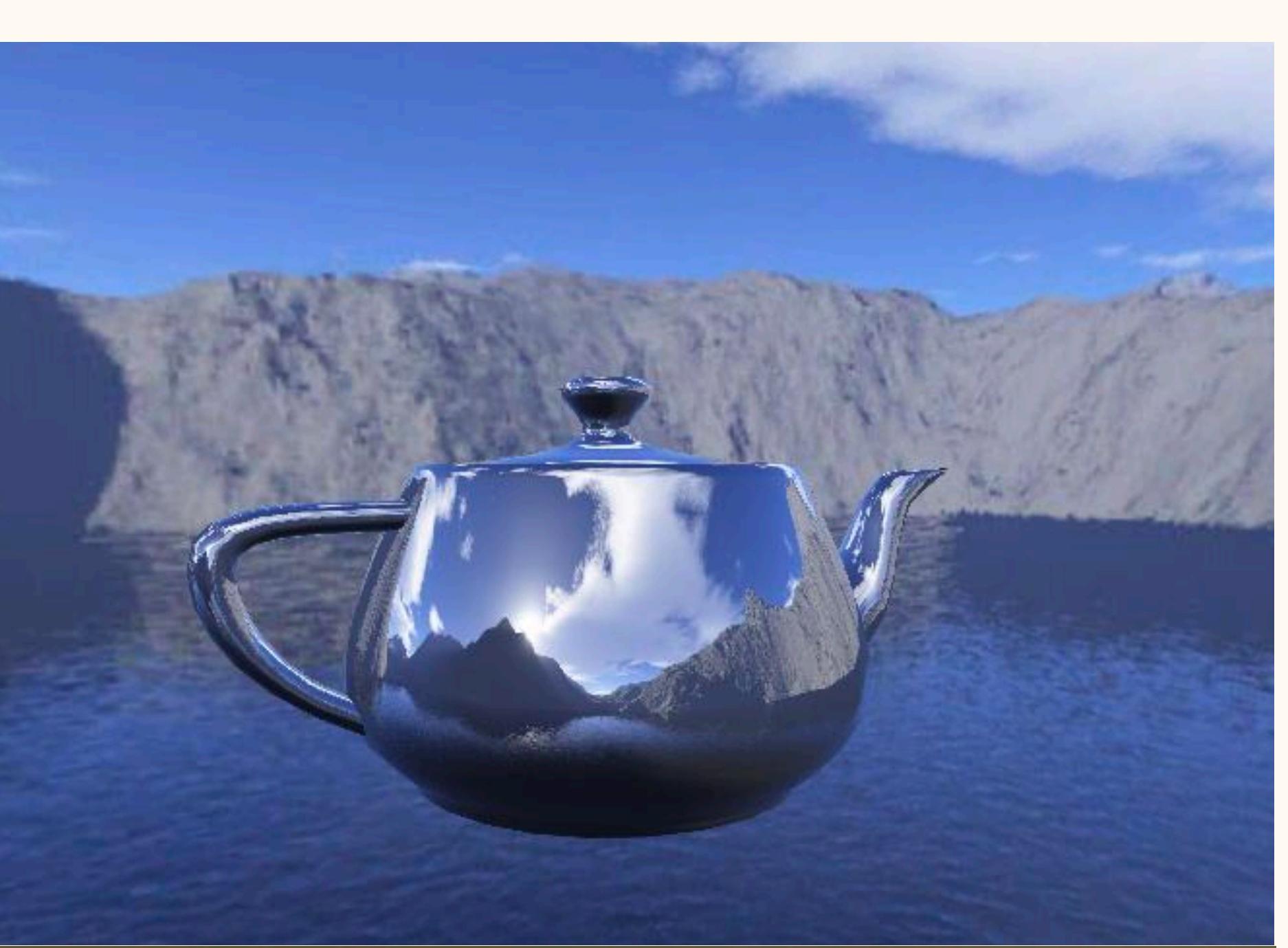
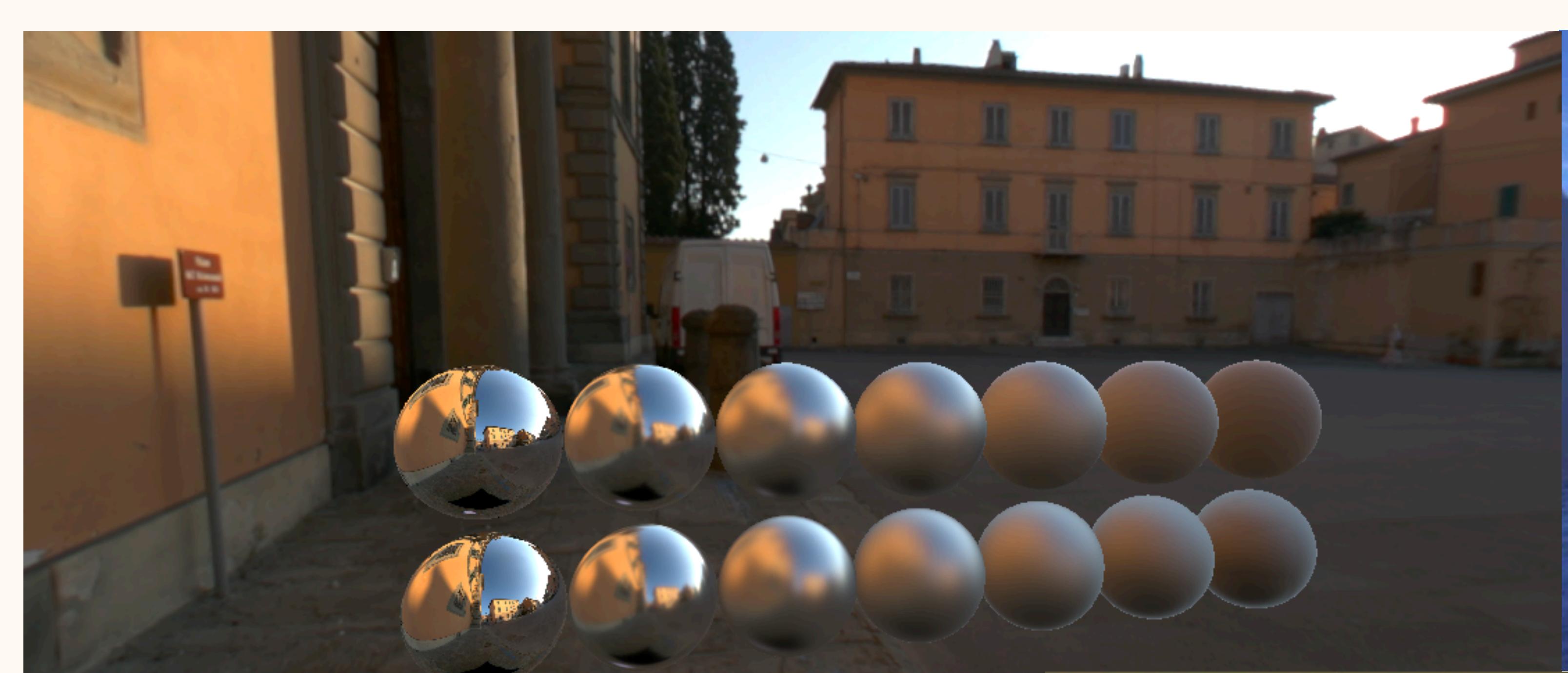
    // apply tone mapping
    color.rgb = color.rgb/(color.rgb + 1.0)*(1.0 + color.rgb/white2);
    color.a = 1.0;
    return color;
}
```

Image-based Lighting

Geometric Lighting vs Image-based Lighting

- Geometric Lighting (幾何打光)
 - Point light
 - parallel light
 - Distant light
 - Spot light
- Image-based Lighting (IBL，非幾何打光)
 - Not geometric lighting
 - 介紹常用的 3 種方法，各有其應用：
 - Reflection & Refraction (Environment Mapping)
 - Diffuse/Specular Environment Mapping (Realtime image-based lighting)
 - Ambient Occlusion

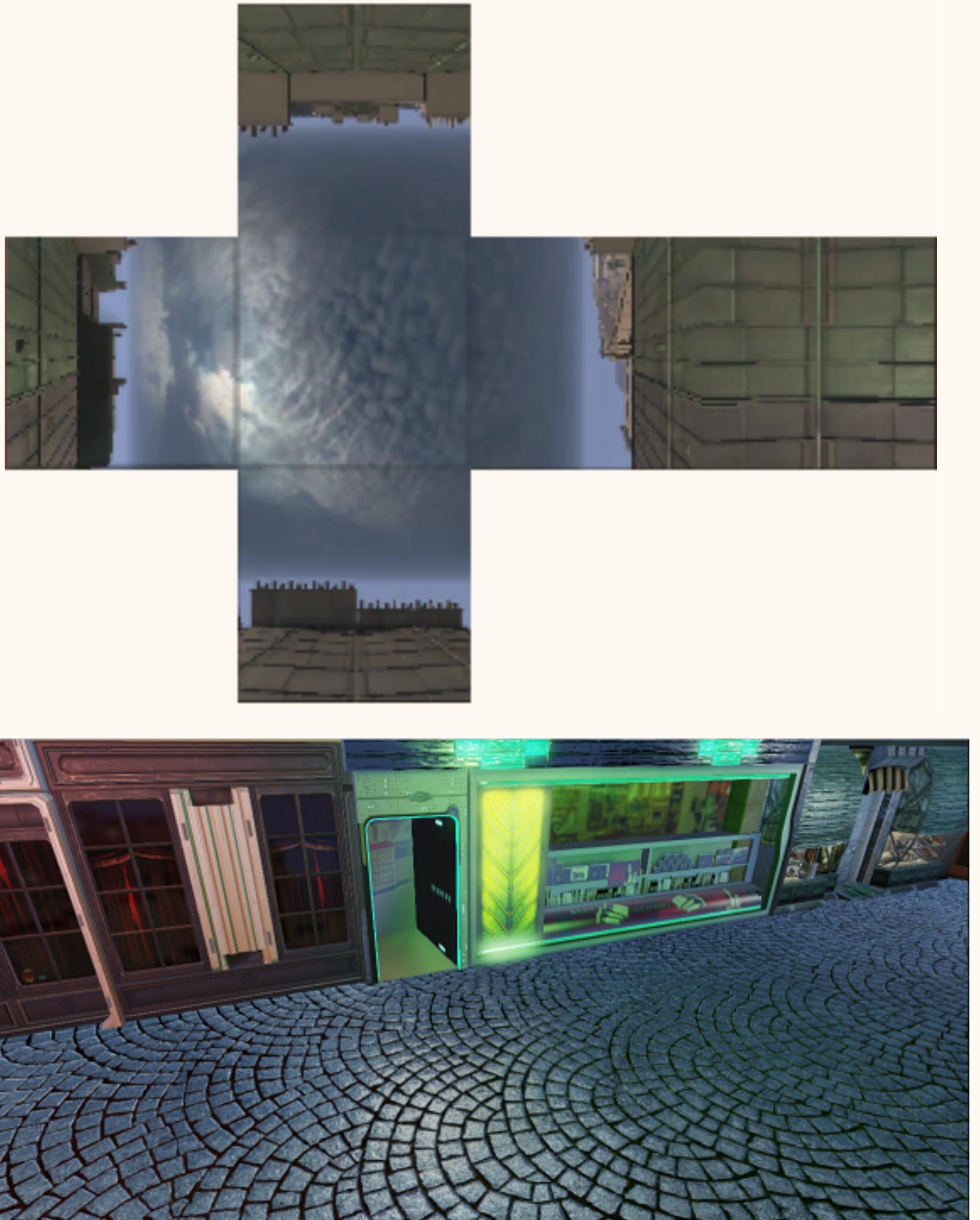




Reflection & Refraction

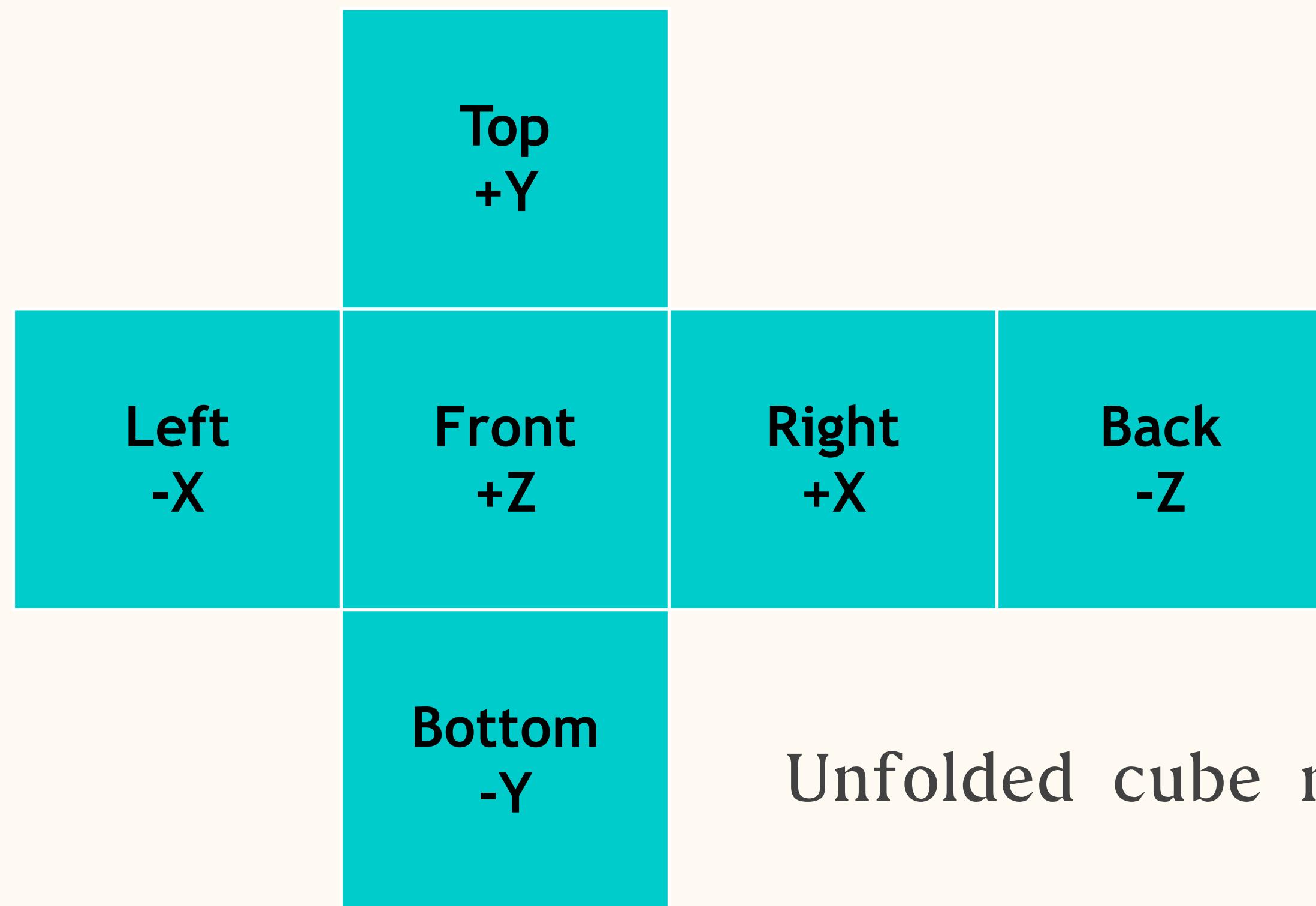
Panorama Images

- Global Image-based lighting solution
 - Reflection / Refraction
 - Diffuse/Specular Environment Map
- Panorama mapping solutions :
 - Cubemap (most used in games)
 - Sky Box
 - Mirrored sphere
 - Angular map (“Light Probe”)
 - Latitude-longitude map (LL Map)
 - Most used for VR
 - 360° panorama

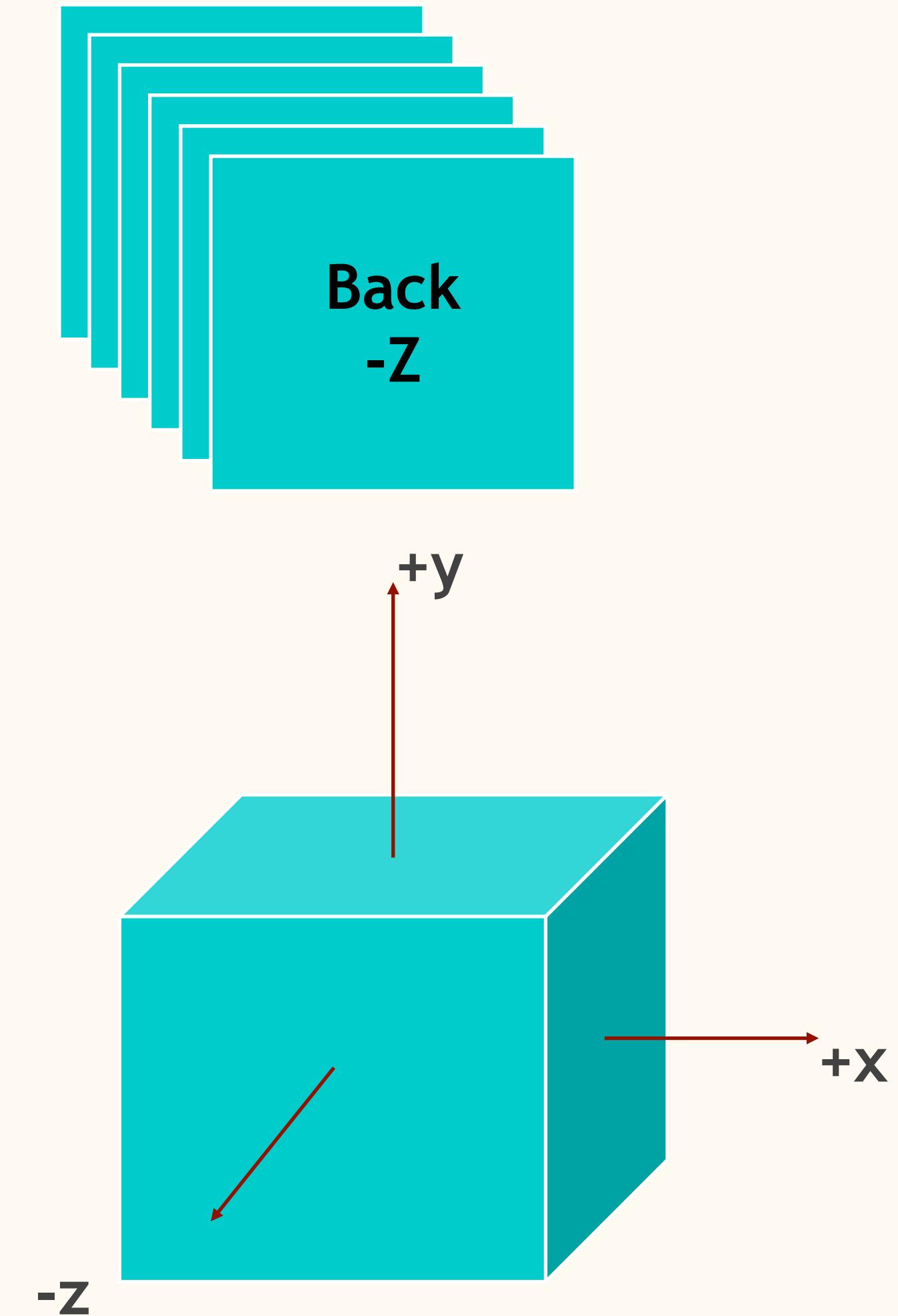




Cube Map

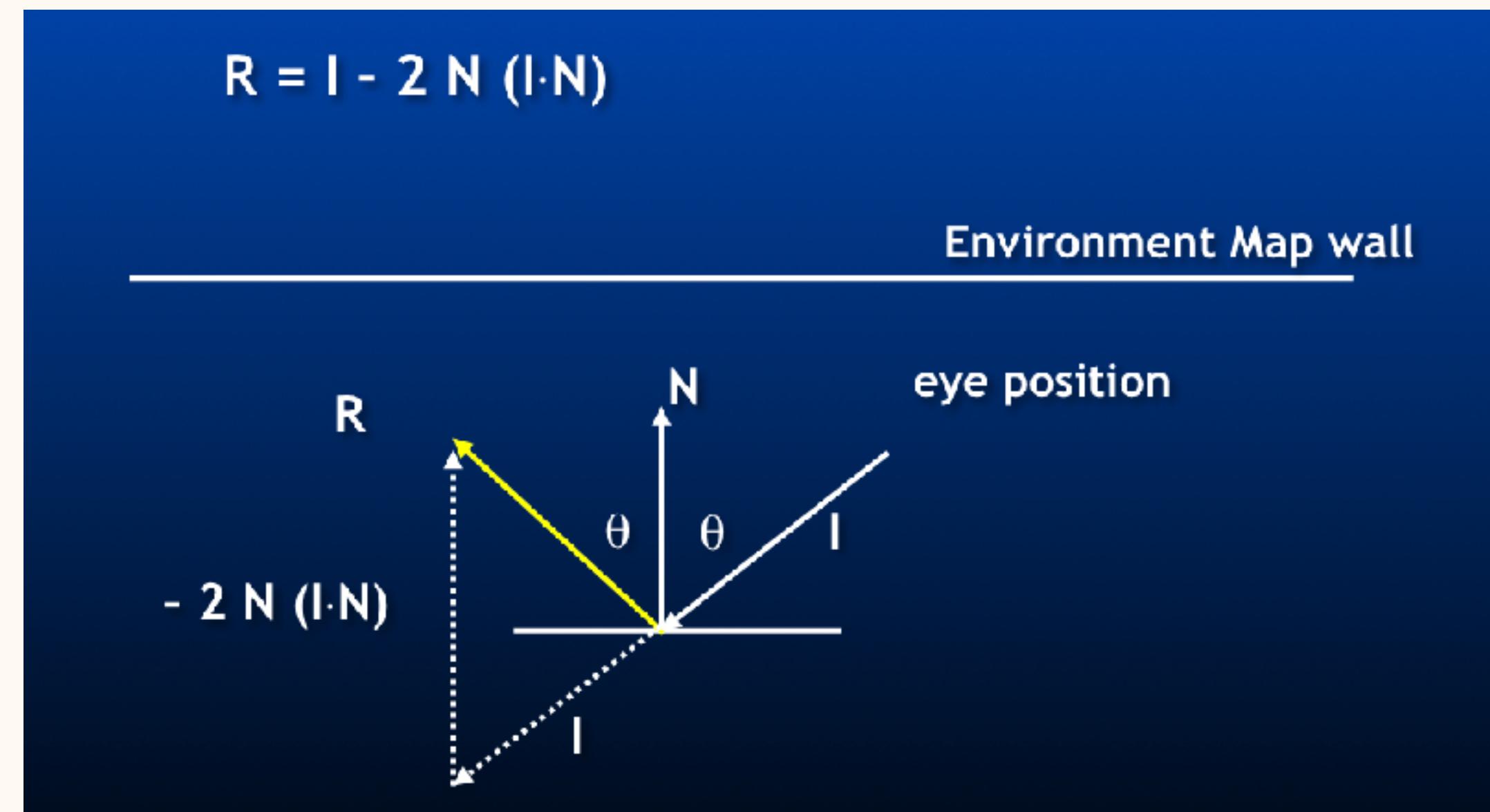


Unfolded cube map



Reflection Vector

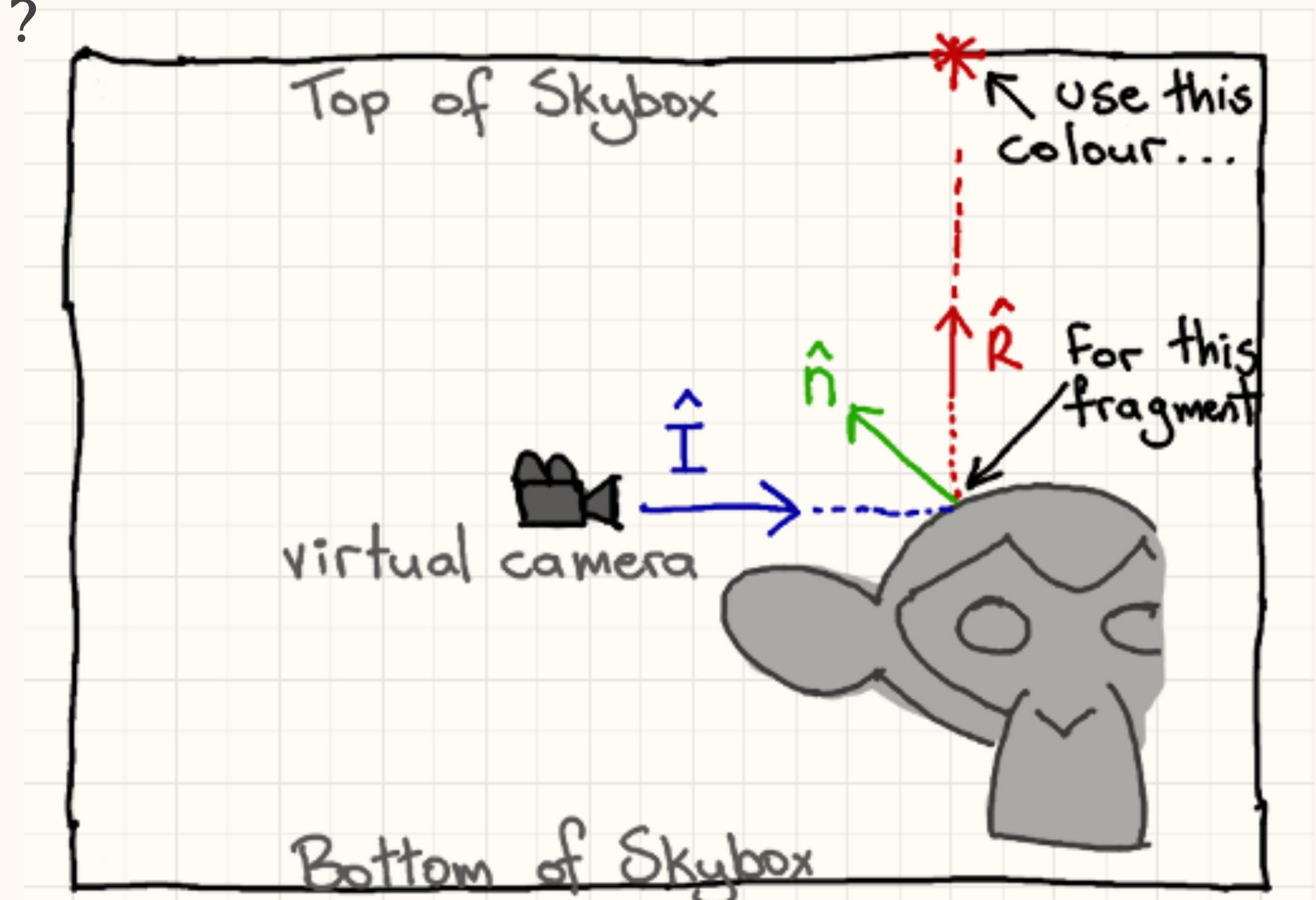
- 計算出以法向量為中心攝影機向量的反射向量 R ，利用向量 R 指向包覆於外的 cube map 求得 cube map 上的顏色 (r, g, b) ，模擬物體表面的鏡面反射的效果
- 計算反射向量 R ：



- HLSL has an intrinsic function :
 - `float3 reflect(float3 I, float3 N);`

Reflection Effect

- Use the reflection vector (\hat{R}) of the viewing to query the cube map :
 - Which color is hit by the vector \hat{R} ?
 - Add the color to object surface



Environment Map Reflection

Reflection Shader

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP          : packoffset(c0);    // matrix from local to screen space
    matrix mWorld         : packoffset(c4);    // matrix from local to global space
    float3 camPosition   : packoffset(c8);    // camera position
};

struct VS_INPUT
{
    float4 inPos  : POSITION;
    float3 inNorm : NORMAL;
    float2 inTex0 : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float3 refl    : TEXCOORD0;
};
```

```
/ the vertex shader
VS_OUTPUT CubemapVS(VS_INPUT in1)
{
    float4 a;
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // convert the vertex from local to global
    a = mul(mWorld, in1.inPos);

    // get the vertex in screen space
    out1.pos = mul(mWVP, in1.inPos);

    // calculate the vertex normal vector in world space
    float3 norm = mul((float3x3) mWorld, in1.inNorm);

    // find the incidence vector (from camera)
    float3 inc = normalize(a.xyz - camPosition.xyz);

    // calculate the reflection vector for environment cubemap
    float3 refl = reflect(inc, norm);
    out1.refl = refl;

    return out1;
}
```

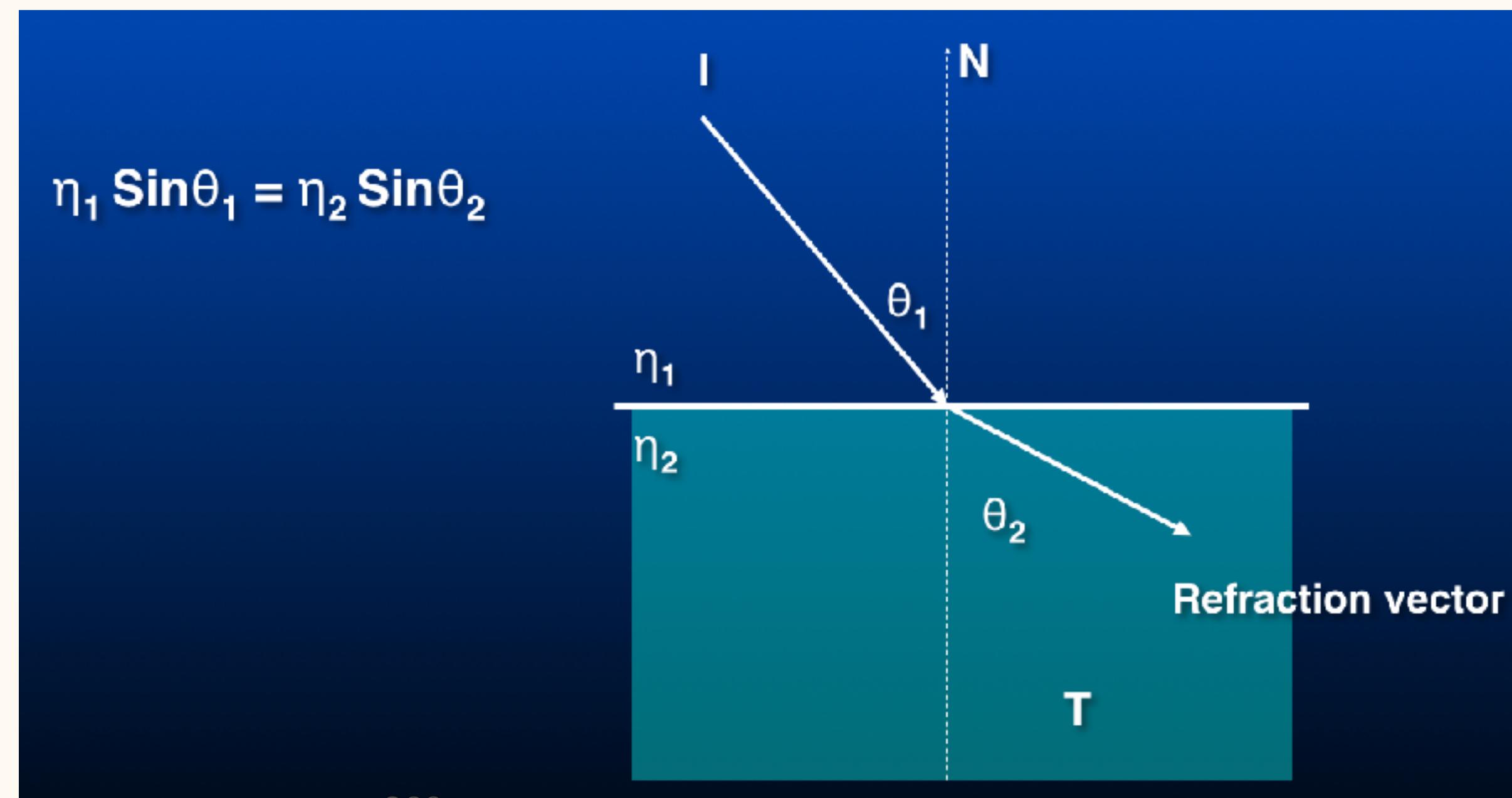
```
// textures and samplers
TextureCube cubeMap : register(t0);
SamplerState cubeMapSampler : register(s0);

// pixel shader input
struct PS_INPUT
{
    float4 pos : SV_POSITION;
    float3 refl : TEXCOORD0;
};

// the pixel shader
float4 CubemapPS(PS_INPUT in1) : SV_TARGET0
{
    // get the environment cubemap data
    return cubeMap.Sample(cubeMapSampler, normalize(in1.refl));
}
```

Refraction Effect

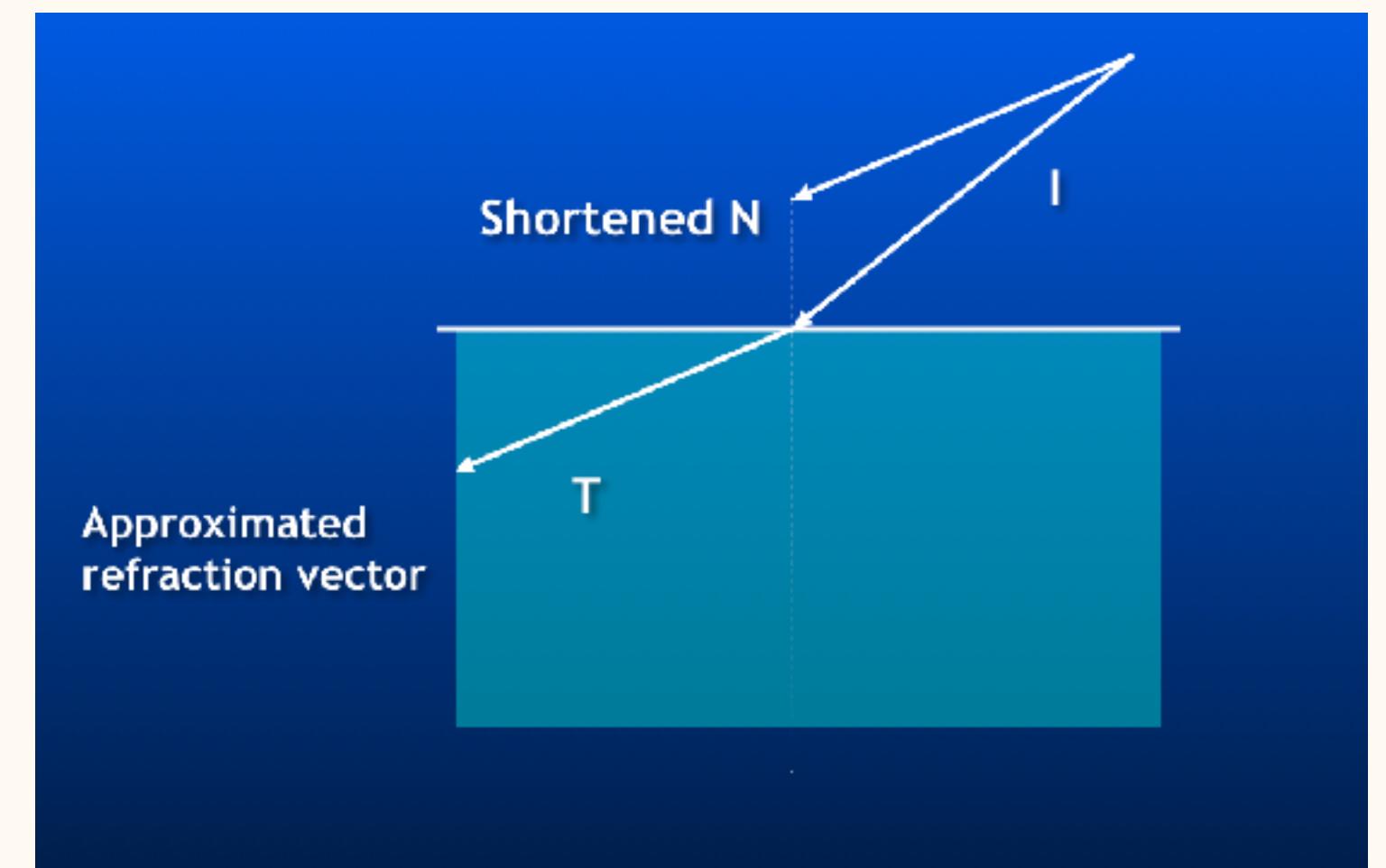
- Snell's Law (斯乃爾折射定理)
- For a closed object, any incident ray should be refracted twice.
 - We can not trace the 2nd refraction. So, the solution is not correct.
 - But the result is still pretty good to simulate the semi-transparent object



Refraction Effect

- HLSL has an intrinsic function :
 - `float3 refract(float3 I, float3 N, float ri)`
 - `ri` is the refraction index
- 2nd solution : “short normal” + reflection function (短向量法)
 - Use reflection equation with a scaled normal vector.
 - Shortening the normal leads to a bend vector that might look like a refraction effect.

```
float3 shortNorm = mul(norm, 0.4f);
float3 refr = reflect(inc, shortNorm);
```



Refraction Shader

```
cbuffer cbPerObject : register(b0)
{
    matrix mWVP          : packoffset(c0);    // matrix from local to screen space
    matrix mWorld         : packoffset(c4);    // matrix from local to global space
    float3 camPosition   : packoffset(c8);    // camera position
};

struct VS_INPUT
{
    float4 inPos    : POSITION;
    float3 inNorm   : NORMAL;
    float2 inTex0   : TEXCOORD0;
};

struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float3 refl     : TEXCOORD0;
    float3 refr     : TEXCOORD1;
};
```

```
VS_OUTPUT CubemapVS(VS_INPUT in1)           // the vertex shader
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // convert the vertex from local to world
    float4 a = mul(mWorld, in1.inPos);

    // get the vertex in screen space
    out1.pos = mul(mWVP, in1.inPos);

    // calculate the vertex normal vector in world space
    float3 norm = mul((float3x3) mWorld, in1.inNorm);

    // find the incidence vector (from camera)
    float3 inc = normalize(a.xyz - camPosition.xyz);

    // calculate the reflection vector for environment cubemap
    float3 refl = reflect(inc, norm);

    // calculate the refraction vector
    // float3 refr = refract(inc, norm, 0.99);
    float3 shortNorm = mul(norm, 0.4f);
    out1.refr = reflect(inc, shortNorm);

    return out1;
}
```

```

// textures and samplers
TextureCube cubeMap           : register(t0);
SamplerState cubeMapSampler   : register(s0);

// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float3 refl     : TEXCOORD0;
    float3 refr     : TEXCOORD1;
};

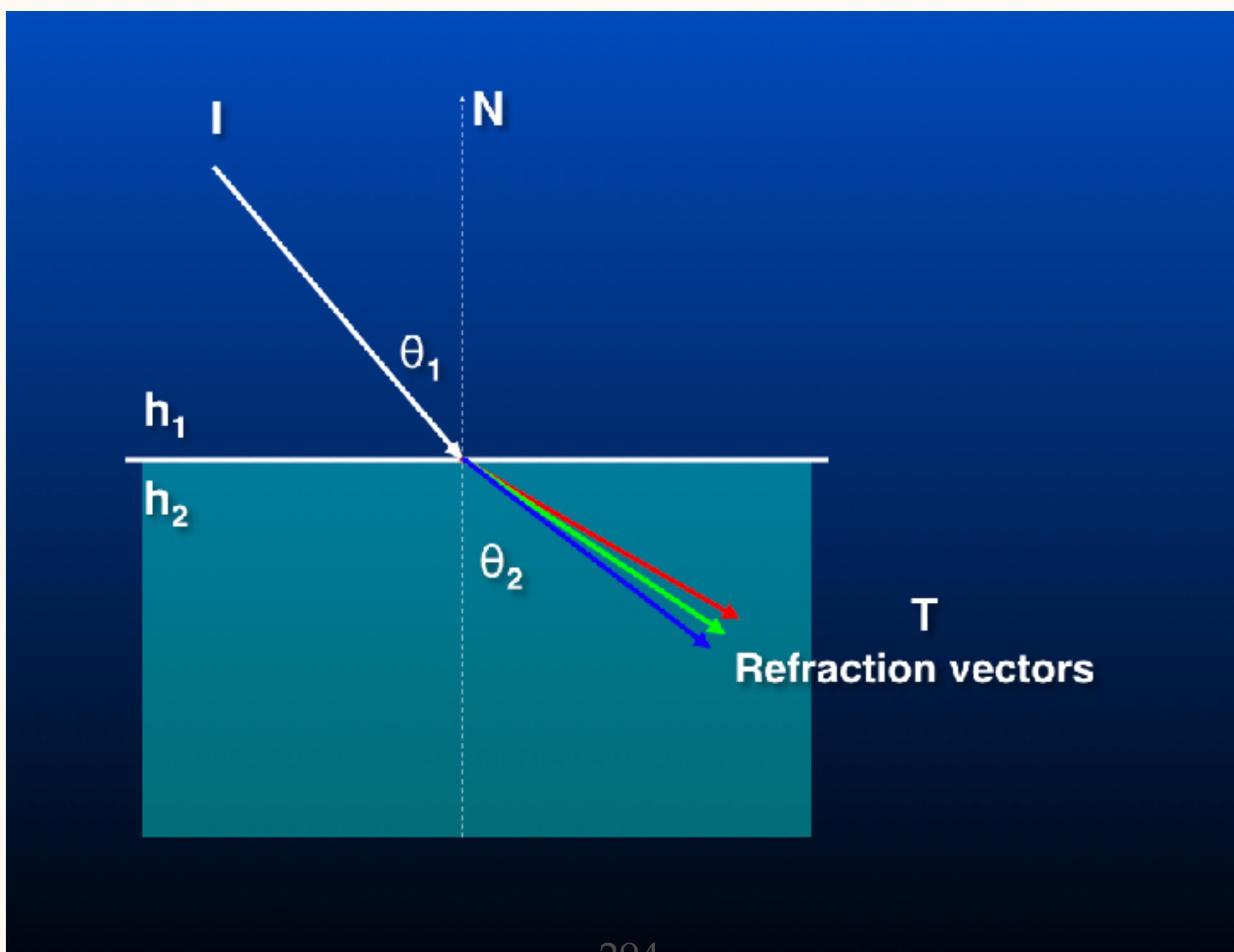
// the pixel shader
float4 CubemapPS(PS_INPUT in1) : SV_TARGET0
{
    // get the environment cubemap data
    float4 rgbaL = cubeMap.Sample(cubeMapSampler, normalize(in1.refl));
    float4 rgbaF = cubeMap.Sample(cubeMapSampler, normalize(in1.refr));

    return rgbaL*0.5 + rgbaF;
}

```

Chromatic Dispersion Effect

- Refraction not only depends on the surface normal, incident angle, and the ratio of indices, but also on incident light wavelength.
 - The red light beam is refracted more than the blue.



Chromatic Dispersion Shader

```
// vertex shader output
struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float3 refl     : TEXCOORD0;
float3 refrR   : TEXCOORD1;
float3 refrG   : TEXCOORD2;
float3 refrB   : TEXCOORD3;
};

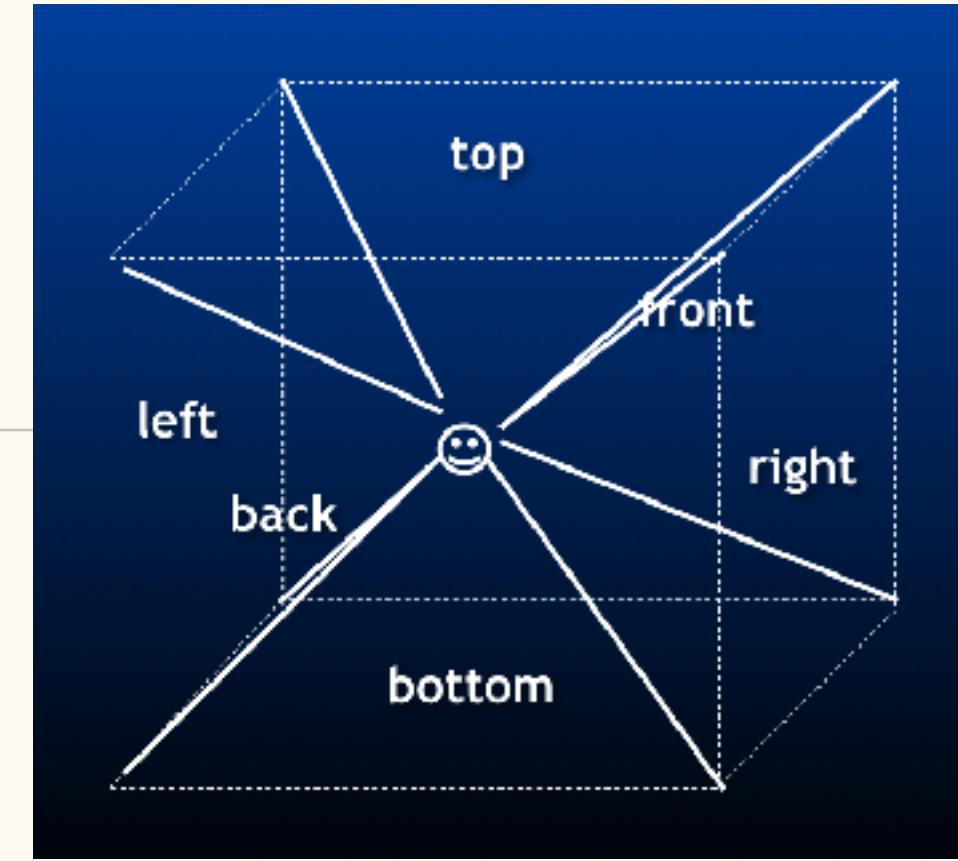
...
// calculate the refraction vector
float3 shortNormR = mul(norm, 0.485f);
float3 shortNormG = mul(norm, 0.49f);
float3 shortNormB = mul(norm, 0.495f);
float3 refrR = reflect(inc, shortNormR);
float3 refrG = reflect(inc, shortNormG);
float3 refrB = reflect(inc, shortNormB);
```

Chromatic Dispersion Shader

```
float4 rgbaF;  
  
// assembly the color of refraction  
rgbaF.r = cubeMap.Sample(cubeMapSampler, normalize(inl.refrR)).r;  
rgbaF.g = cubeMap.Sample(cubeMapSampler, normalize(inl.refrG)).g;  
rgbaF.b = cubeMap.Sample(cubeMapSampler, normalize(inl.refrB)).b;  
rgbaF.a = 1.0;
```

Dynamic Reflection Effect

- 即時計算出 cubemap
- Multi-pass rendering solution :
 - Set a **squared viewport** as a rendering area.
 - The viewport size is the **same as** the rendering target texture.
 - Set the camera's **FOV = 90°** & **aspect ratio = 1.0**
 - Use the camera to render the scene in the direction of the +x, -x, +y, -y, +z, & -z.
 - Map to the six surfaces of the cube map texture
 - 需要在渲染 reflection 效果前先利用上述設計渲染出 cubemap 上六個平面
 - Cubemap 解析度會影響渲染效能



HDR Photography

HDR Image for Image-based Lighting

- 自 90 年代 Hollwoord 就使用 HDR 影像於電影特效製作上
- 現在常見於即時渲染的應用上 (i.e., games)

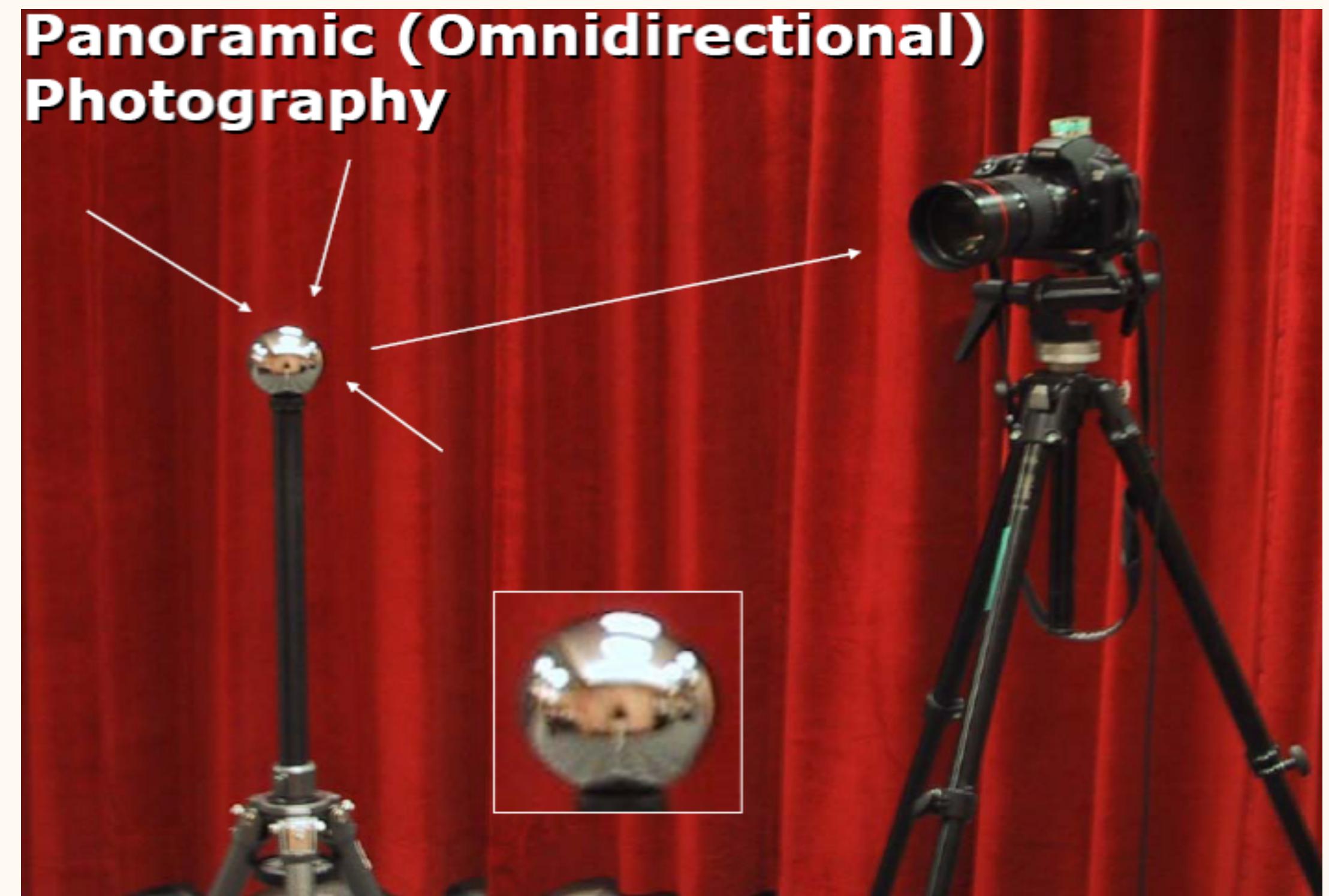


Capture HDR Light Probe Images

- 可手動曝光的數位相機
- 三角架和雲台
- 可捕捉全景圖像的設備
 - 鏡面反射的金屬反光球

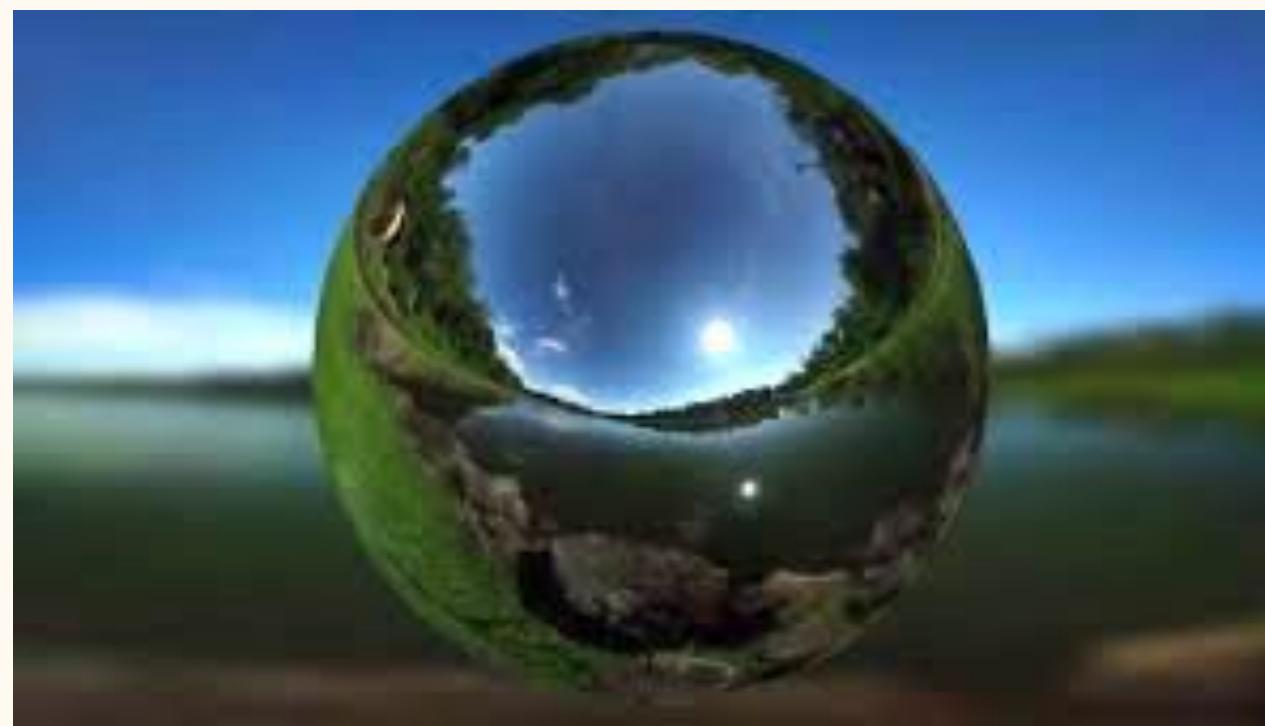


Panoramic (Omnidirectional) Photography



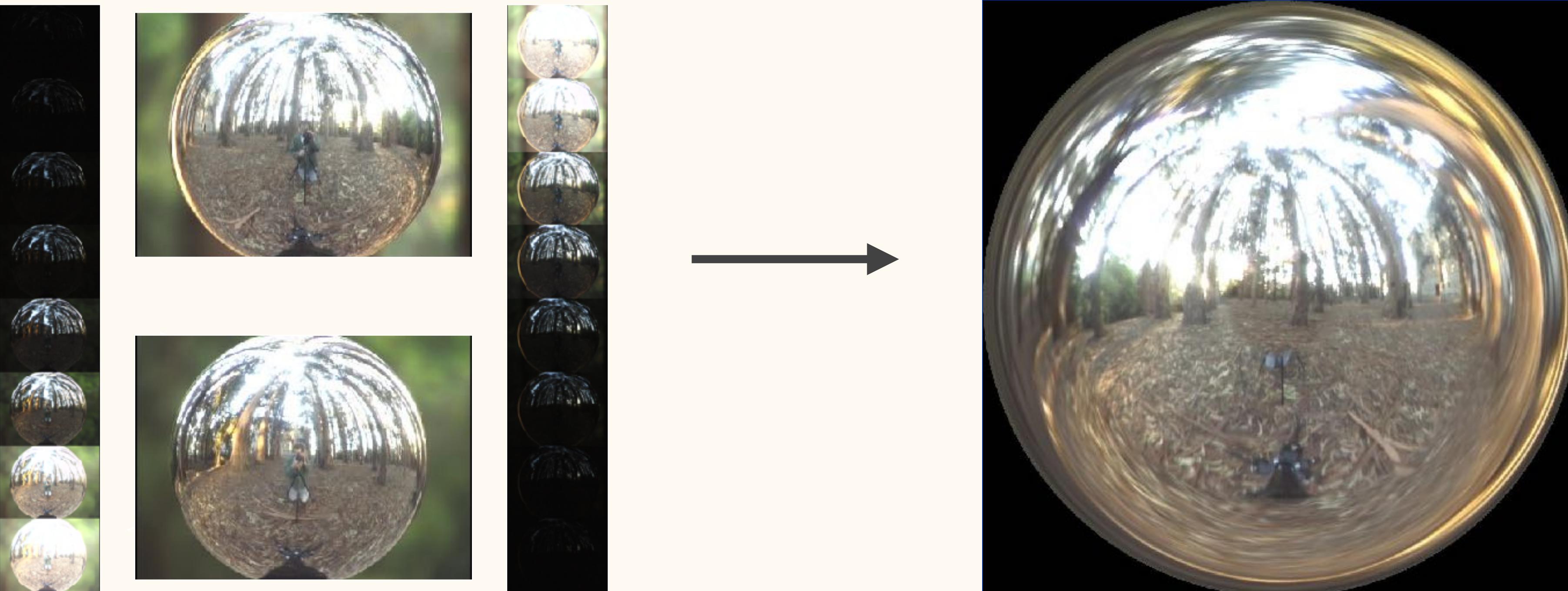
Capture HDR Light Probe Images

- A almost-100%-reflective sphere can capture more than 180° reflection
 - 8" stainless steel Gazing Sphere costs around USD20



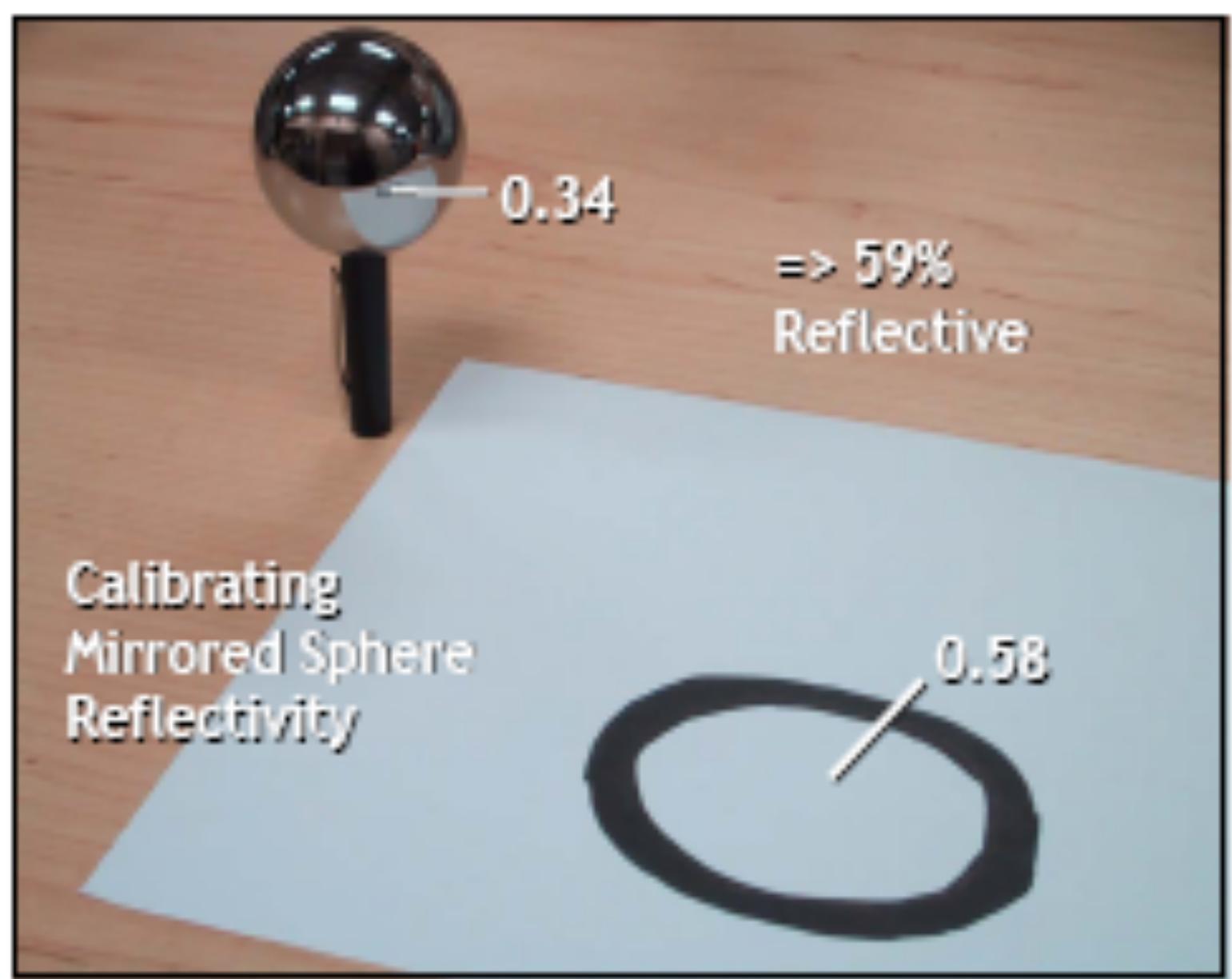
Capture HDR Light Probe Images

- 拍兩組 LDR images 曝光範圍為 (-4, -3, -2, -1, 0, 1, 2, 3) Ev
 - 兩組為同一光源位置但拍攝位置成90°
 - 使用 HDRShop 將LDR影像組合成一張HDR影像

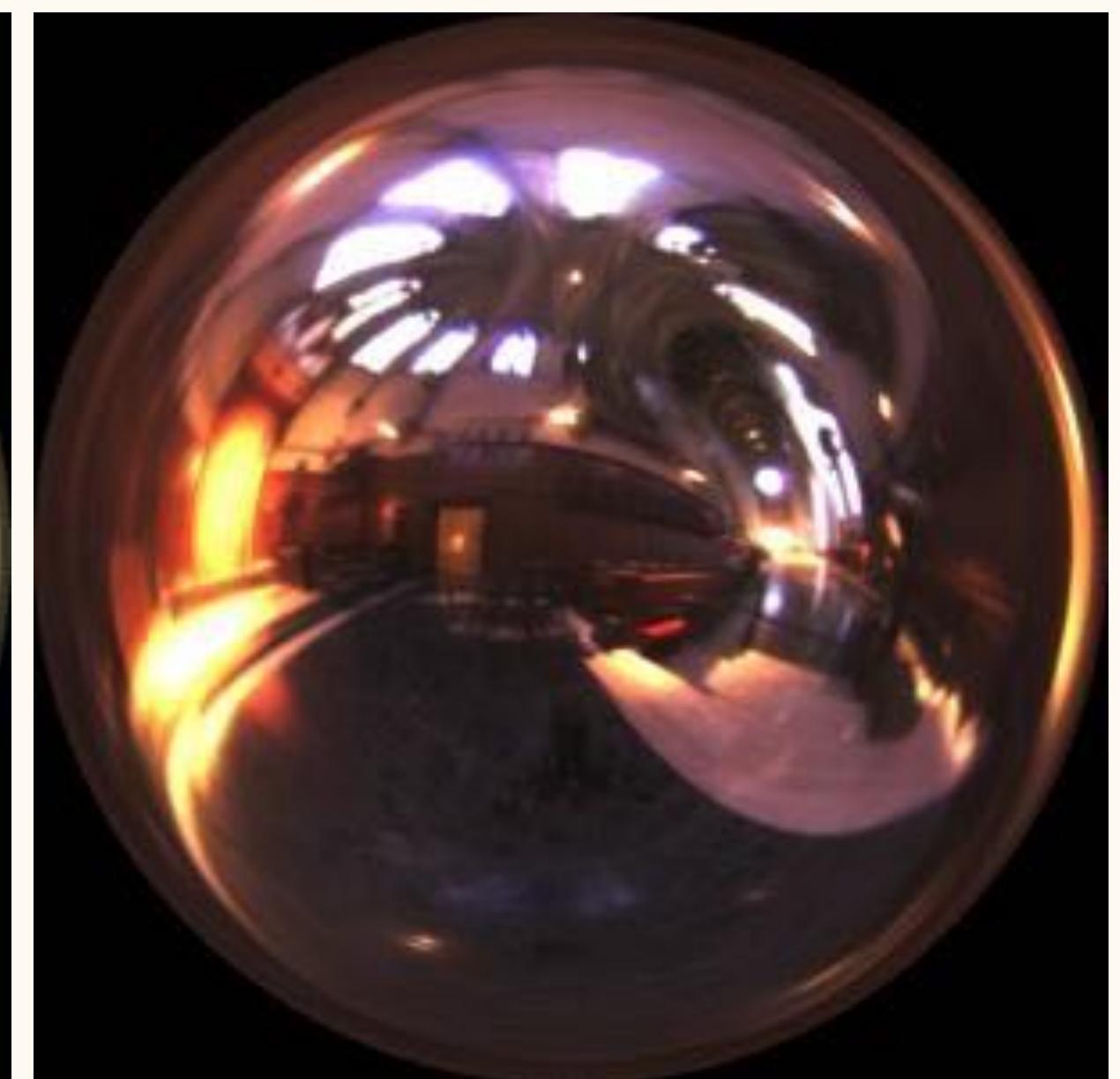
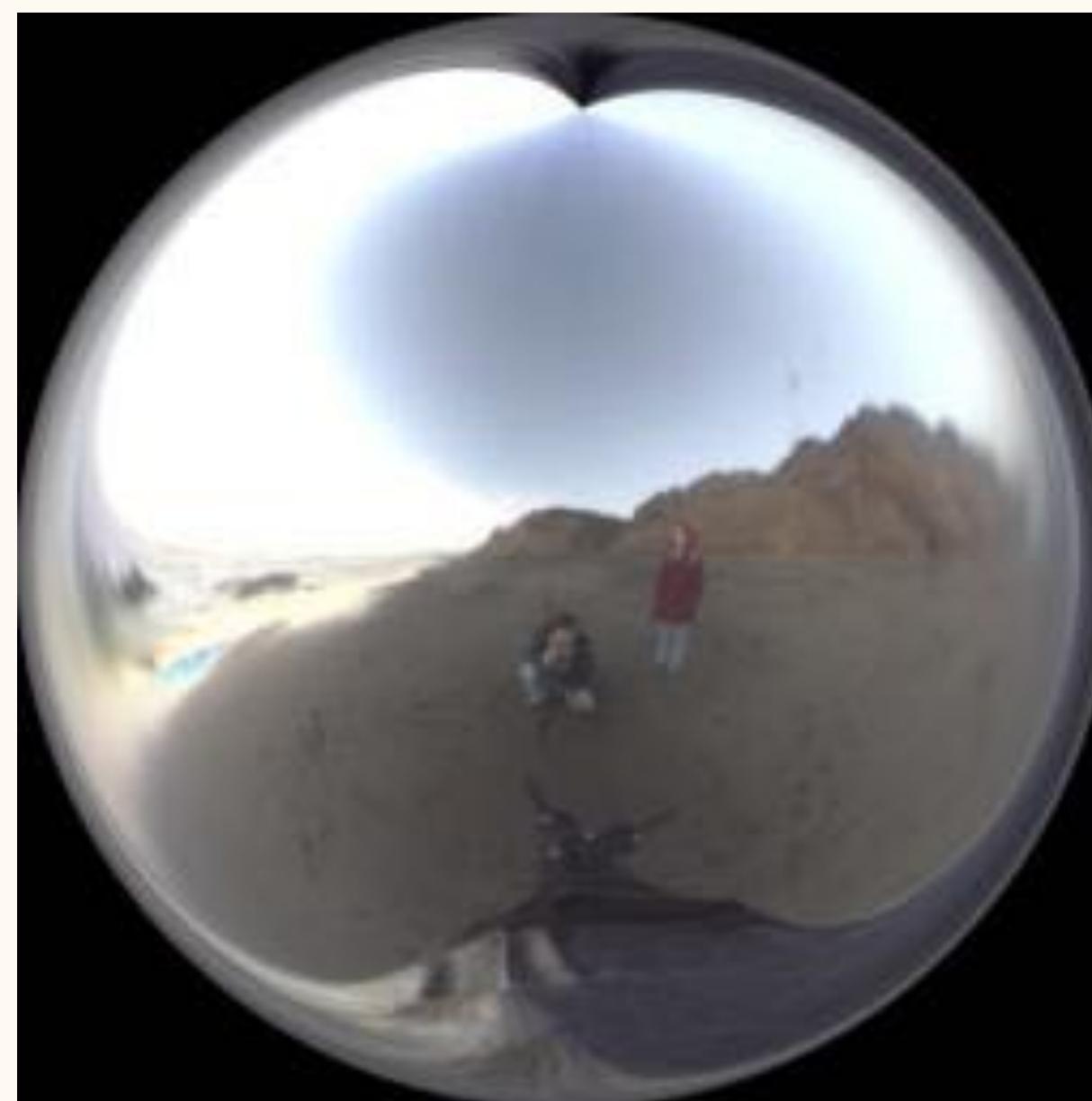


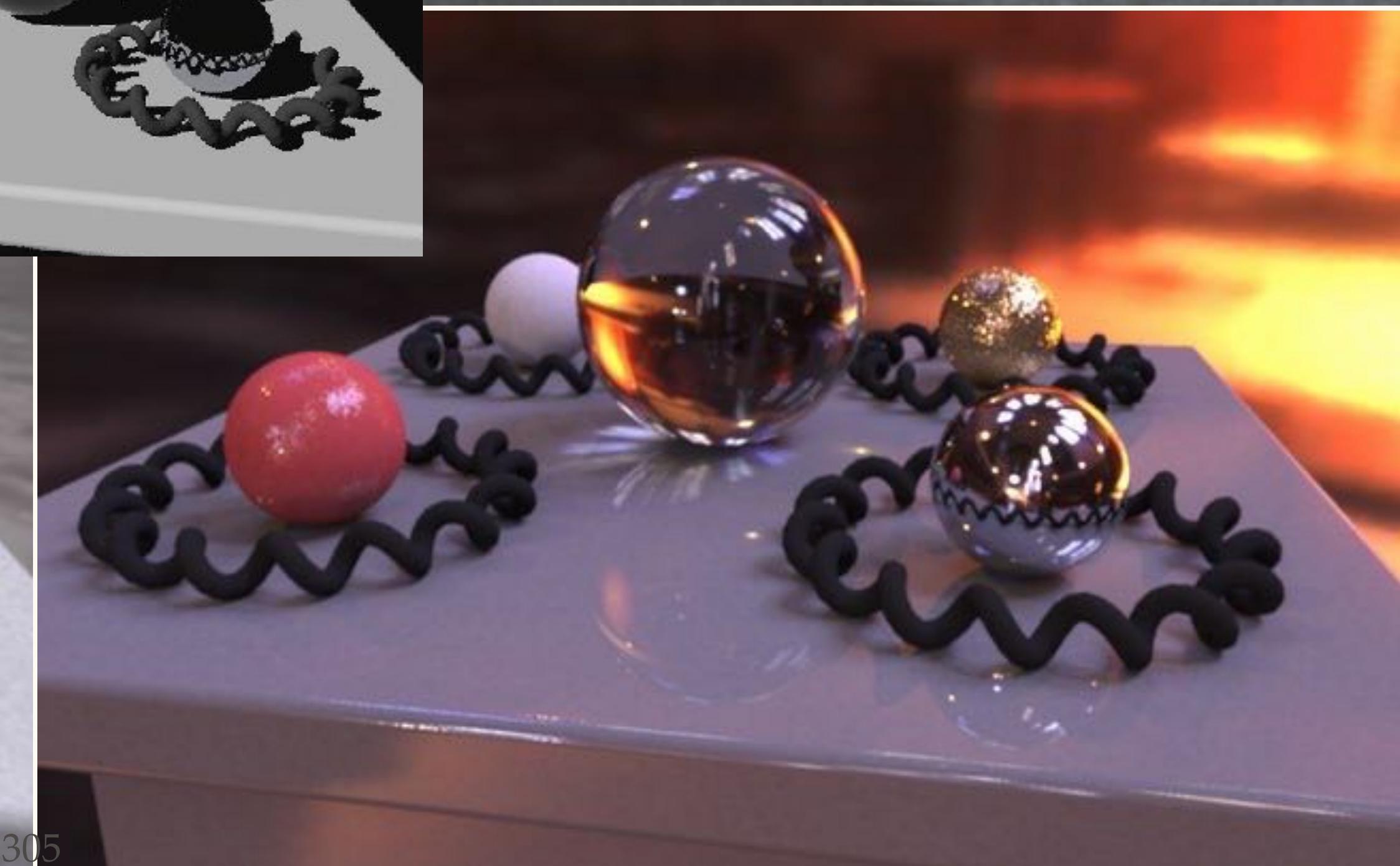
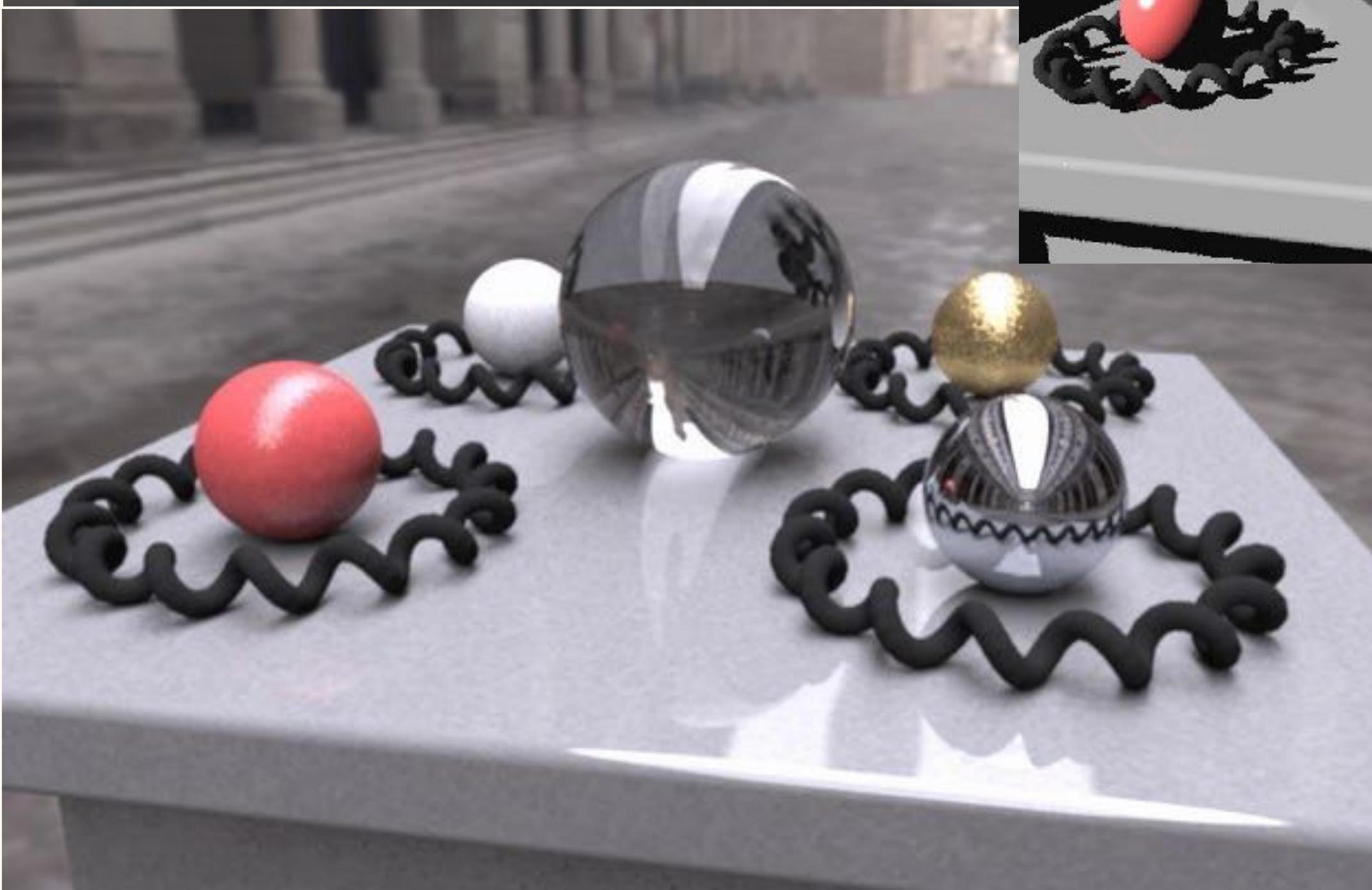
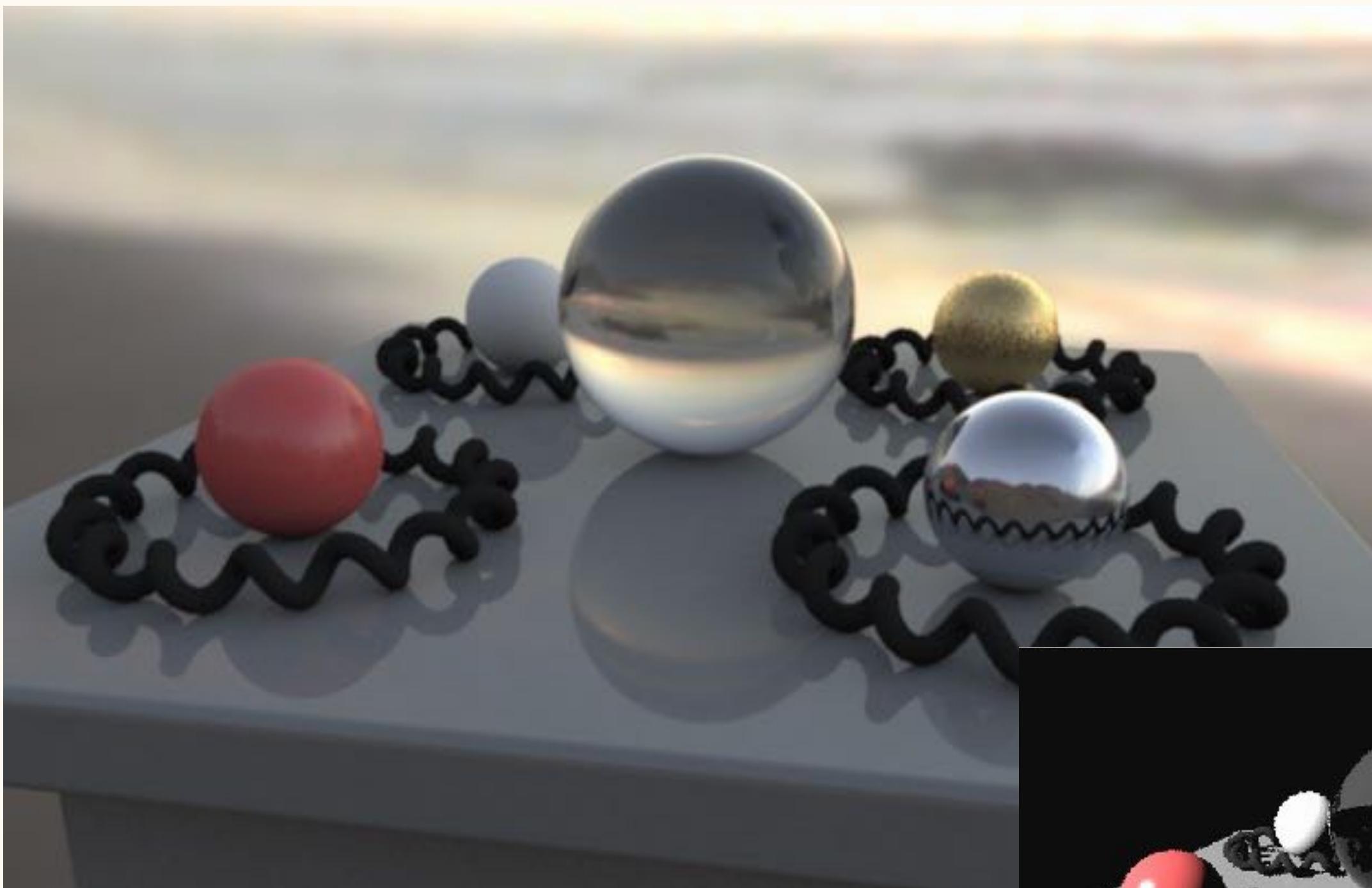
Mirrored Sphere

- 對焦與裁減
- Blind spots
 - 攝影師與腳架會入鏡
 - 金屬球的正後方拍不到
 - 來往行人與車輛
- 球反射率的校正
 - 一般反射率數值約 $(r, g, b) = (0.632, 0.647, 0.652)$
- Non-specular reflection
 - 刮傷
 - 氧化



HDR Image for Image-based Lighting





Panorama Image Formats

- 常用的幾種全景圖像格式：
 - Cubemap
 - Ideal mirrored sphere
 - Angular map (light probe)
 - Latitude longitude map
- 可使用 HDRShop software 來互轉
- Light probe 是通用的名稱

Ideal Mirrored Sphere

- 直接拍攝 mirrored sphere
- Mapping equation :
- From world to image :

$$r = \frac{\sin(\arccos(-D_z) * 0.5)}{2 * \sqrt{D_x^2 + D_y^2}}$$

$$(u, v) = (0.5 + rD_x, 0.5 - rD_y)$$

- From image to world :

$$r = \sqrt{(2u-1)^2 + (2v-1)^2}$$

$$(\theta, \varphi) = (\text{atan}2(2u-1, -2v+1), 2\arcsin(r))$$

$$(D_x, D_y, D_z) = (\sin\varphi\cos\theta, \sin\varphi\sin\theta, -\cos\varphi)$$



y-up right-hand space

Angular Map

- 由 mirrored sphere image 轉變成的
- Mapping equation :
- From world to image :

$$r = \frac{\arccos(-D_z)}{2\pi * \sqrt{D_x^2 + D_y^2}}$$

$$(u, v) = (0.5 - rD_y, 0.5 + rD_x)$$

- From image to world :

$$(\theta, \varphi) = (\text{atan}2(-2v+1, 2u-1), \pi\sqrt{(2u-1)^2 + (2v-1)^2})$$

$$(D_x, D_y, D_z) = (\sin\varphi\cos\theta, \sin\varphi\sin\theta, -\cos\varphi)$$



y-up right-hand space

Latitude-Longitude Map

- A rectangular image domain
 - 2 : 1 ratio : u in $[0, 2]$ and v in $[0, 1]$
- Mapping equation :
- From world to image :

$$(u, v) = (1 + \text{atan}2(D_x, -D_z)/\pi, \arccos(D_y)/\pi)$$



y-up right-hand space

- From image to world :

$$(\theta, \varphi) = (\pi(u-1), \pi v)$$

$$(D_x, D_y, D_z) = (\sin\varphi\cos\theta, \cos\varphi, -\sin\varphi\sin\theta)$$

Use Latitude-Longitude Map

```
#define PI 3.1415926

// use 2D texture for Latitude-Longitude map
Texture2D txLLMap : register(t0);
SamplerState txLLMapSampler : register(s0);

// calculate the v3 to uv mapping of LL map
float2 latlong(float3 v3)
{
    float theta = acos(v3.y); // +y is up
    float pi = atan2(v3.x, -v3.z) + PI;
    return float2(phi, theta)*float2(0.5/PI, 1.0/PI);
}

...
float2 uv = latlong(R); // R is the reflection vector
float4 rgba = txLLMap.Sample(txLLMapSampler, uv);
```

Global Realtime Image-based Lighting

Real-time Image-based Lighting

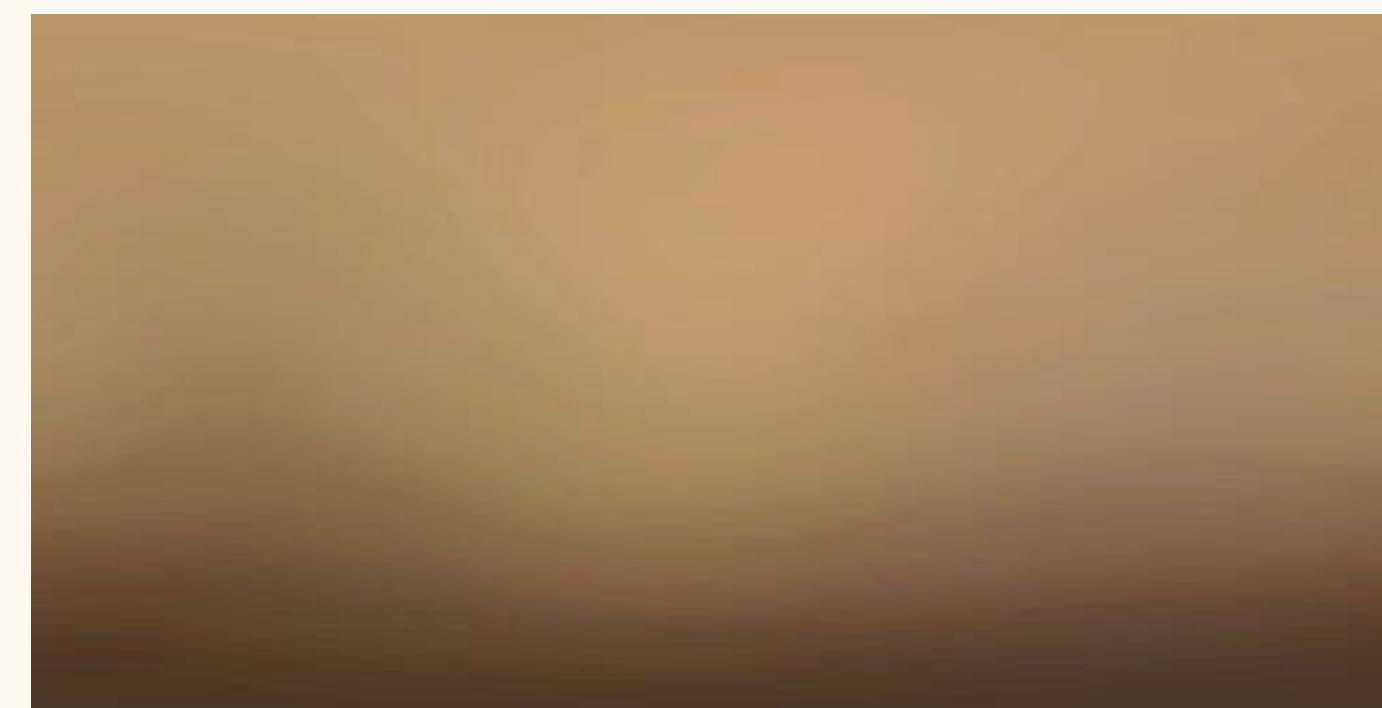
Diffuse/Specular Environment Mapping

Global Image-based Lighting

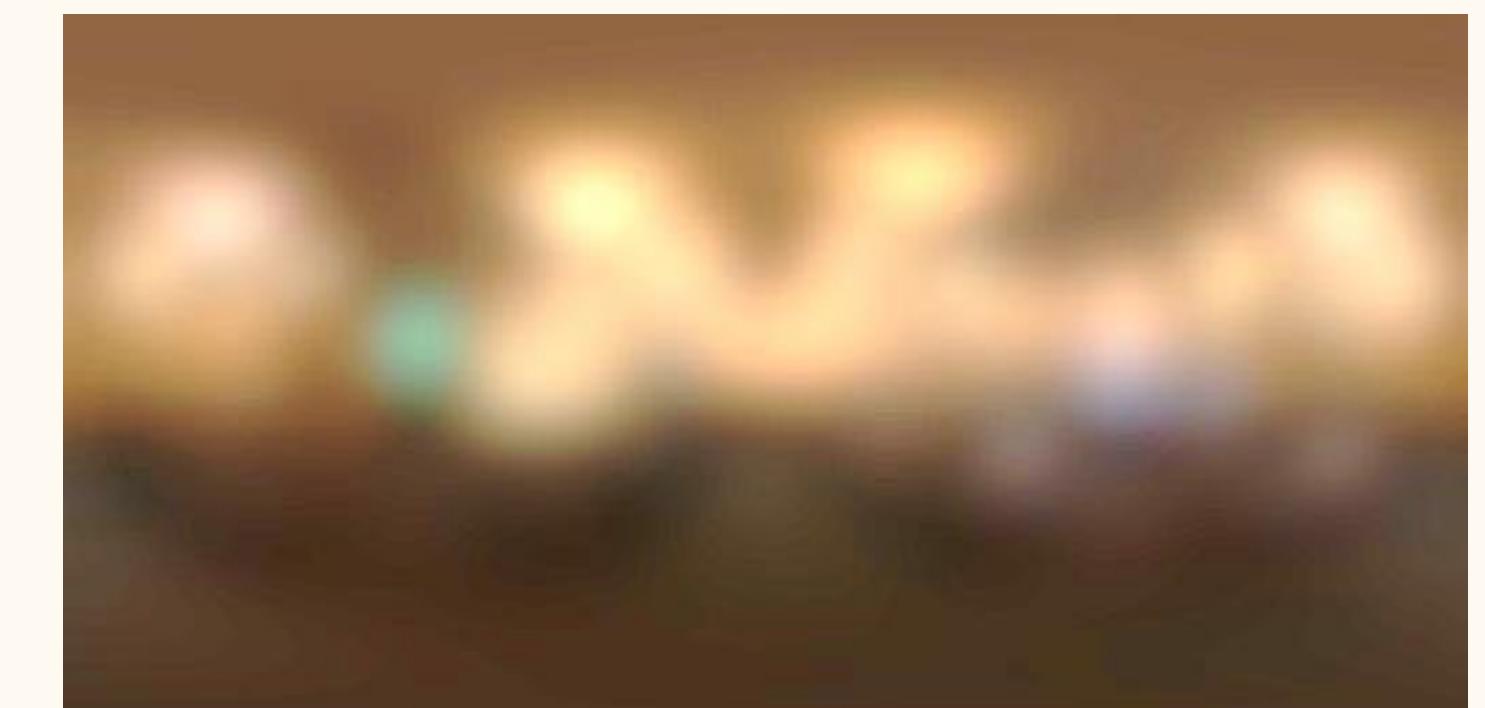
- 使用方法與 reflection 相近：
 - 預先烘焙一張 diffuse environment map 用於 diffuse lighting
 - 預先烘焙一張 specular environment map 用於 specular lighting
 - “烘焙”的意思？
 - 事先準備，一般而言需要製作時間！
 - 在適當的前提下，可以即時利用 multi-pass rendering 動態產生
- “Global”的意思是整個場景使用一張 HDR image，假設 3D 模型是位於球心位置



HDR environment map

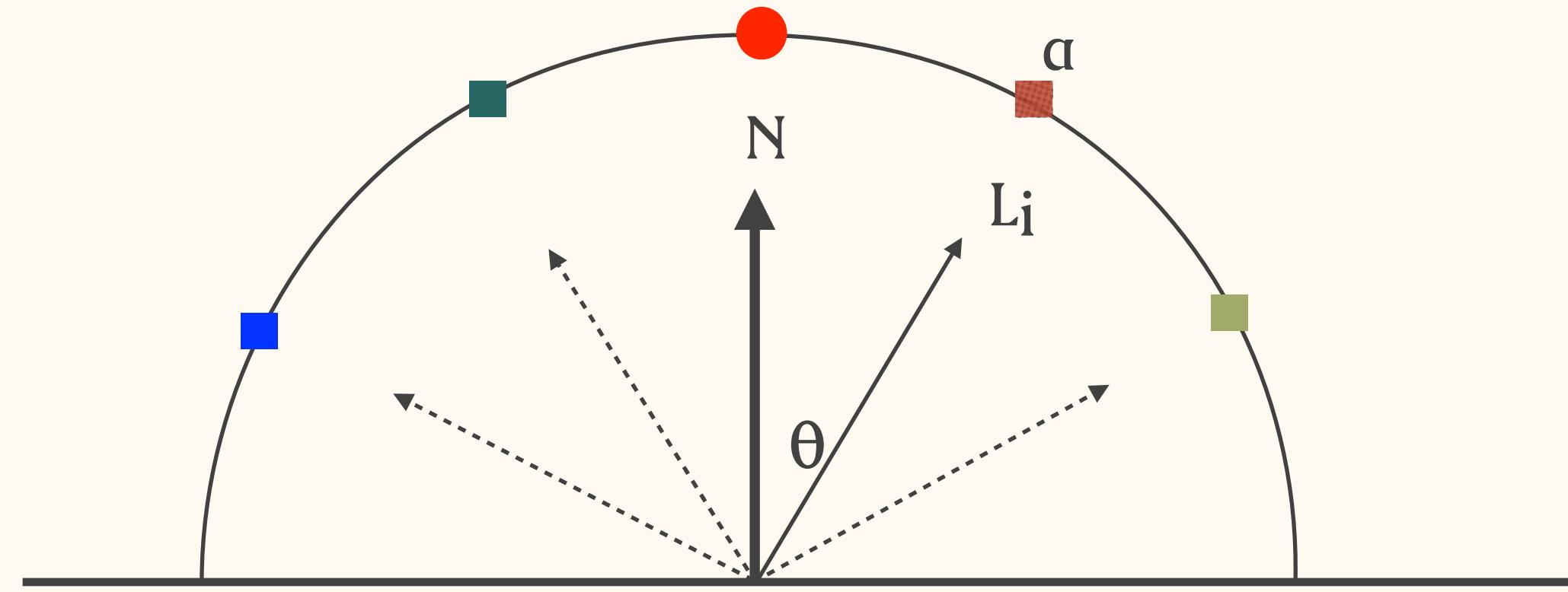


Diffuse environment map
312



Specular environment map

Baking Diffuse/Specular Environment Map



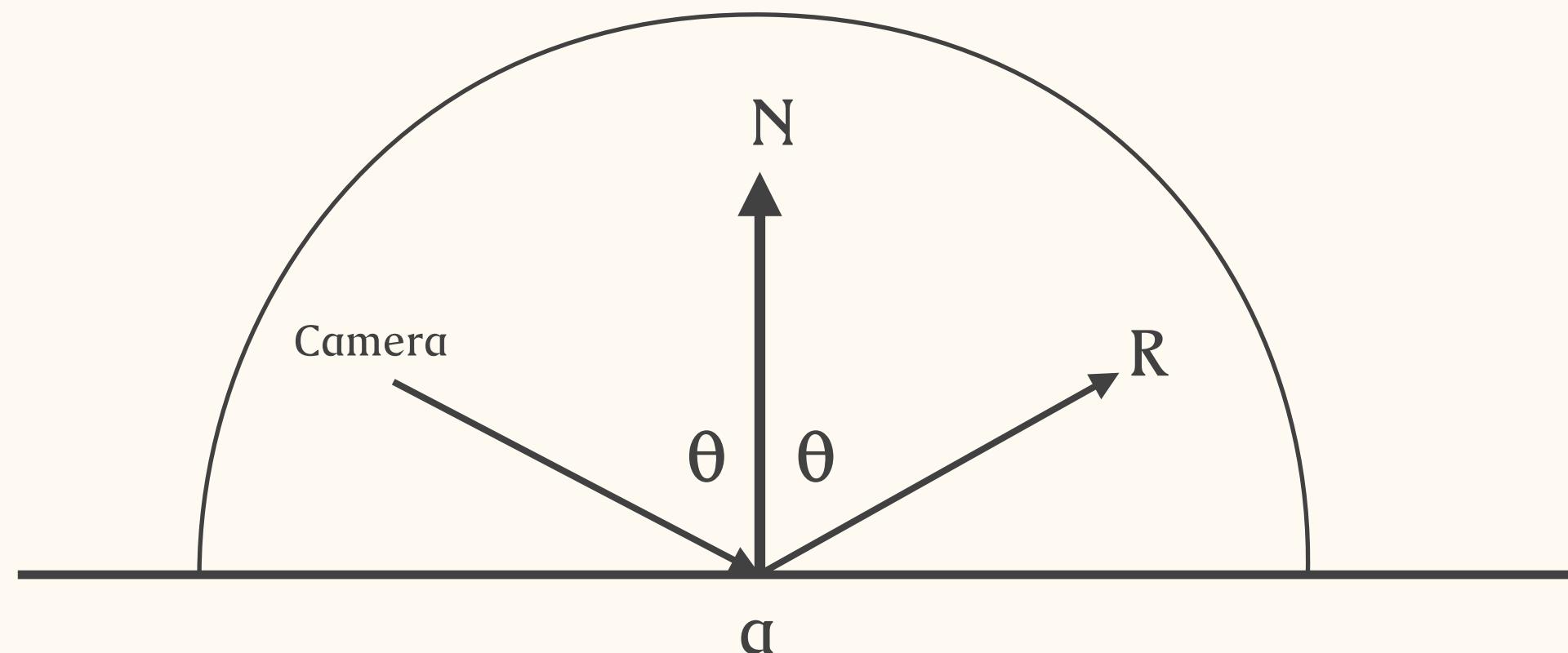
Effective Lighting from $a = \text{[color]} * (N \cdot L_i)^n$

$$\text{Result} = \sum I_i * (N \cdot L_i)$$

- Create a new texture for the diffuse environment map
- For the UV position of each HDR map pixel, ●:
 - Aligned with N , we have a hemisphere
 - Integrate all lighting from the hemisphere aligned with vector, N
 - Update it to the same UV position on the diffuse environment map
- n is the shininess, for diffuse environment map $n = 1$
- HDRShop implemented this function

Using Diffuse/Specular Environment Map

- 利用 lat-long map (latitude-longitude map) 做為 environment map format
- 計算 diffuse term：
 - 使用法向量, N , 在 diffuse 的 map 上取得以此向量為軸的所有光照總和
- 計算 specular term：
 - 使用攝影機方向的反射向量, R , 在 specular 的 map 上取得以此向量為軸的所有光照總和



Global Image-based Lighting Shader (Phong)

```
// vertex shader constants
cbuffer cbPerObject : register(b0)
{
    matrix mWVP          : packoffset(c0);    // matrix from local to screen space
    matrix mWorld         : packoffset(c4);    // matrix from local to global space
    float3 camPosition   : packoffset(c8);    // camera position
    matrix mWorldInv     : packoffset(c9);    // inverse of world matrix
};

// vertex shader input
struct VS_INPUT
{
    float4 inPos  : POSITION;
    float3 inNorm : NORMAL;
    float2 inTex0 : TEXCOORD0;
    float3 inTang : TANGENT;
};
```

```

// vertex shader output
struct VS_OUTPUT
{
    float4 pos      : SV_POSITION;
    float2 tex0     : TEXCOORD0;
    float3 norm     : TEXCOORD1;
    float3 cam      : TEXCOORD2;
    float3 tangentToWRow0 : TEXCOORD5;
    float3 tangentToWRow1 : TEXCOORD6;
};

// the vertex shader
VS_OUTPUT PhongNormTangVS(VS_INPUT in1)
{
    VS_OUTPUT out1 = (VS_OUTPUT) 0;

    // compute the 3x3 transformation matrix for tangent space to the world space
    float3x3 wToTangent, tangentToW;
    wToTangent[0] = normalize(mul(in1.inTang, (float3x3) mWorldInv));
    wToTangent[1] = normalize(mul(cross(in1.inTang, in1.inNorm), (float3x3) mWorldInv));
    wToTangent[2] = normalize(mul(in1.inNorm, (float3x3) mWorldInv));
    tangentToW = transpose(wToTangent);
    out1.tangentToWRow0 = tangentToW[0];
    out1.tangentToWRow1 = tangentToW[1];
    out1.norm = tangentToW[2];
}

```

```

// convert the vertex from local to global
float4 a = mul(mWorld, in1.inPos);

// get the vertex in screen space
out1.pos = mul(mWVP, in1.inPos);

// prepare the normal and camera vector for pixel shader
out1.cam = normalize(camPosition.xyz - a.xyz);
out1.tex0 = in1.inTex0;

return out1;
}

```

```

// pixel shader constants
cbuffer cbPerObject : register(b0)
{
    float4 amb          : packoffset(c0); // ambient component of the material
    float4 dif          : packoffset(c1); // diffuse component of the material
    float4 spe          : packoffset(c2); // specular component of the material
    float   shine        : packoffset(c3); // material shininess
};

```

```
// textures and samplers
Texture2D txDiffuse           : register(t0);
SamplerState txDiffuseSampler : register(s0);

Texture2D txNormal            : register(t1);
SamplerState txNormalSampler  : register(s1);

Texture2D probeDMap          : register(t11);
SamplerState probeDMapSampler : register(s11);

Texture2D probeSMap          : register(t12);
SamplerState probeSMapSampler : register(s12);

// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float2 tex0     : TEXCOORD0;
    float3 norm    : TEXCOORD1;
    float3 cam      : TEXCOORD2;
    float3 tangent : TEXCOORD5;
    float3 biNormal : TEXCOORD6;
};
```

```

// get the texture coordinate when using Lat-Log map for image-based lighting
float2 LatLongIM(float3 v)
{
    float3 vv = normalize(v);
    float theta = acos(vv.y);           // we use +y up
    float phi = atan2(vv.x, -vv.z) + 3.1415962;
    return float2(phi, theta)*float2(0.15916, 0.31831);
}

// the pixel shader
float4 PhongNormTangPS(PS_INPUT in1) : SV_TARGET0
{
    // be sure to normalize the vectors
    float3 camDir = normalize(in1.cam);
    float3 tangentDir = normalize(in1.tangent);
    float3 biNormDir = normalize(in1.biNormal);
    float3 normWDir = normalize(in1.norm);

    // convert tangent-space normal to world space
    float3 normDir;
    float3 nFromBump = txNormal.Sample(txNormalSampler, in1.tex0).xyz*2.0 - 1.0;
    normDir.x = dot(tangentDir, nFromBump);
    normDir.y = dot(biNormDir, nFromBump);
    normDir.z = dot(normWDir, nFromBump);
    normDir = normalize(normDir);
}

```

```
// perform the image-based lighting
// use normal vector to get diffuse lighting
float2 uv = LatLongIM(normDir);
float3 imgDiff = probeDMap.Sample(probeDMapSampler, uv)*2.0;

// use camera direction's reflection vector to get specular lighting
float3 refl = reflect(-camDir, normDir);
uv = LatLongIM(refl);
float3 imgSpec = probeSMap.Sample(probeSMapSampler, uv)*0.15;

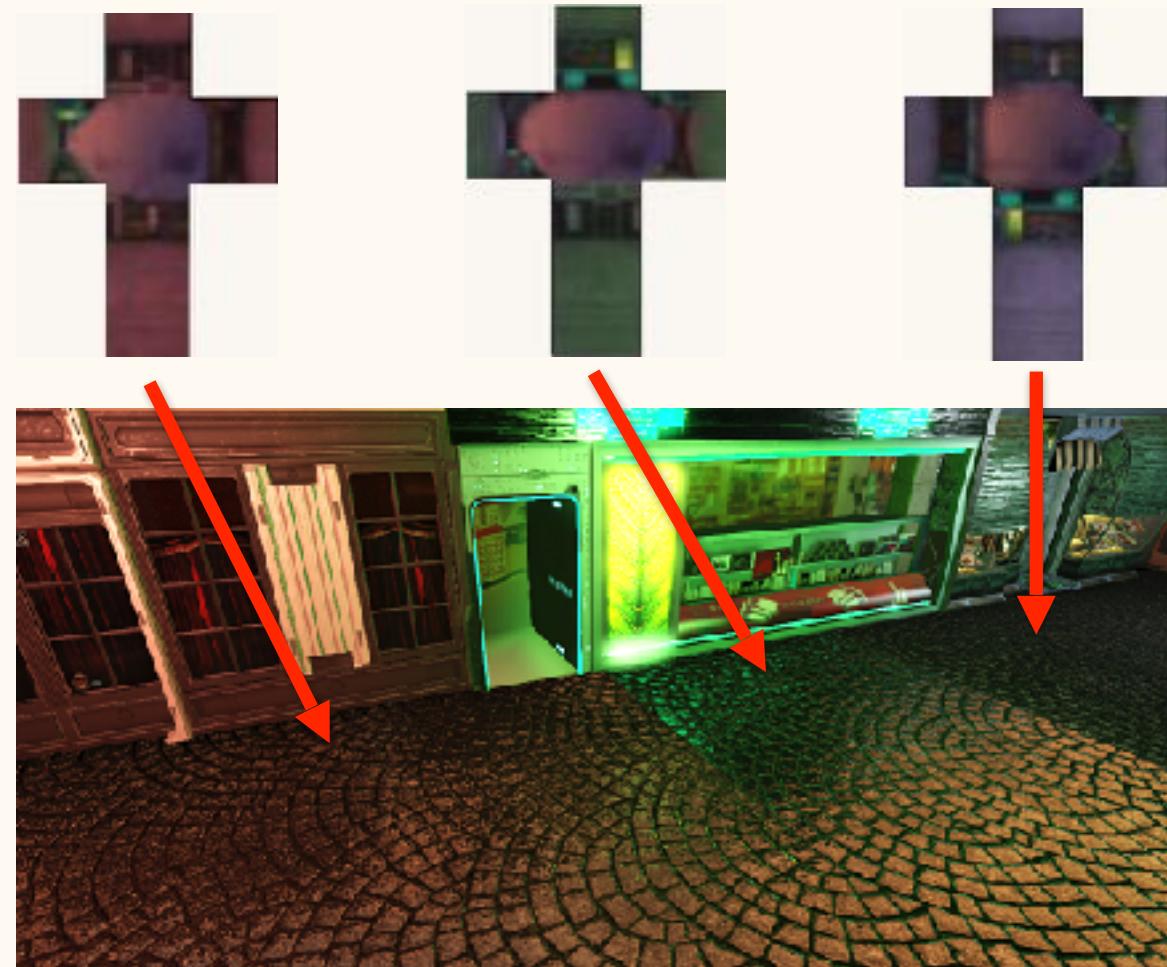
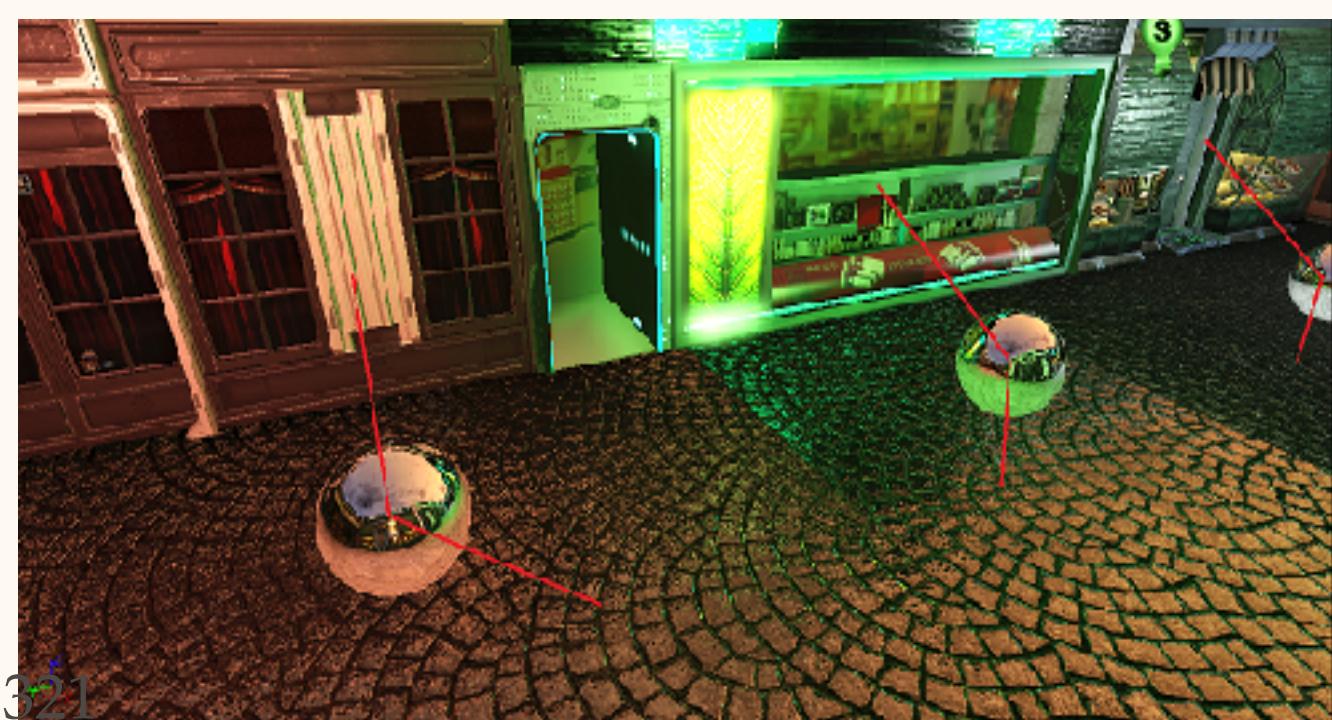
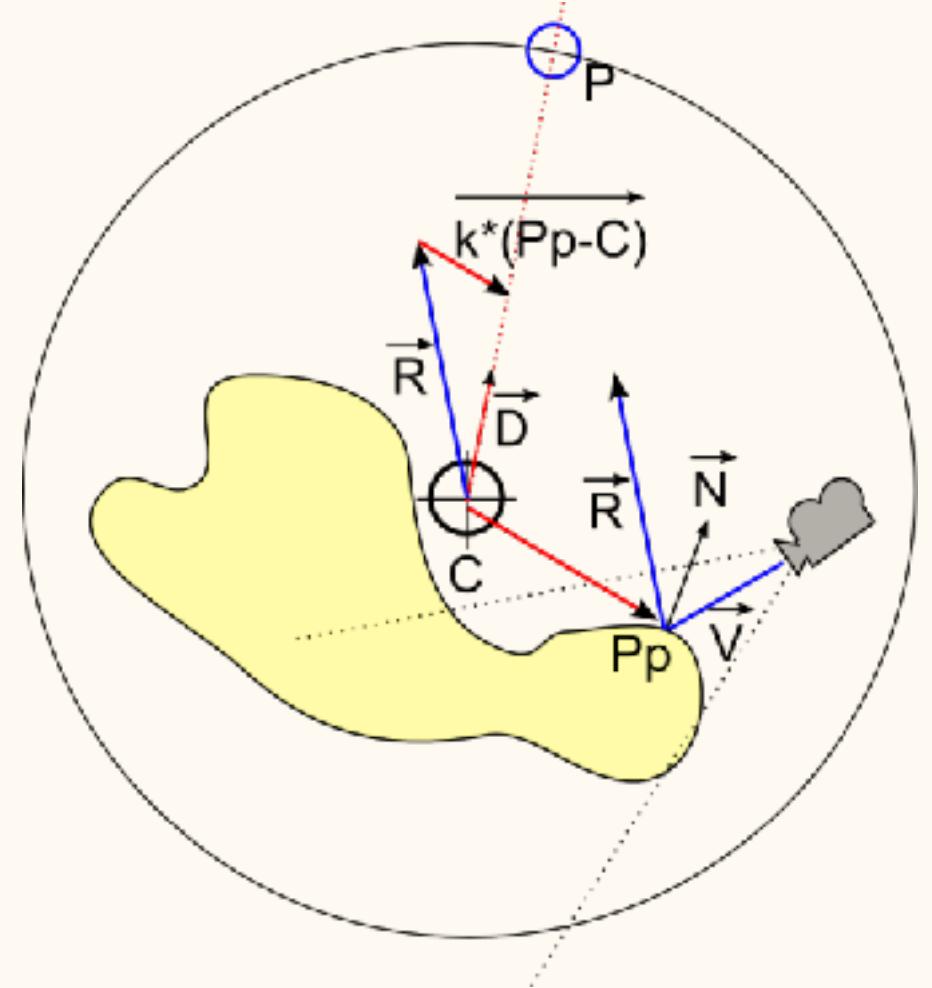
// get color texture data
float4 color = dif*txDiffuse.Sample(txDiffuseSampler, in1.tex0);

// Phong-Blinn reflection model in Image-based lighting
float4 rgba;
rgba.rgb = imgDiff*color.rgb + imgSpec*spe;
rgba.a = color.a;

return rgba;
```

Global Image-based Lighting Limitations

- One large global light probe for all :
 - Without local reflection effect.
 - Parallax error for large flat plane reflection
 - For example, the surface of a lake
- Solutions ;
 - Multiple local light probes
 - But need to solve the lighting seams between light probes
 - Dynamic baking light probes



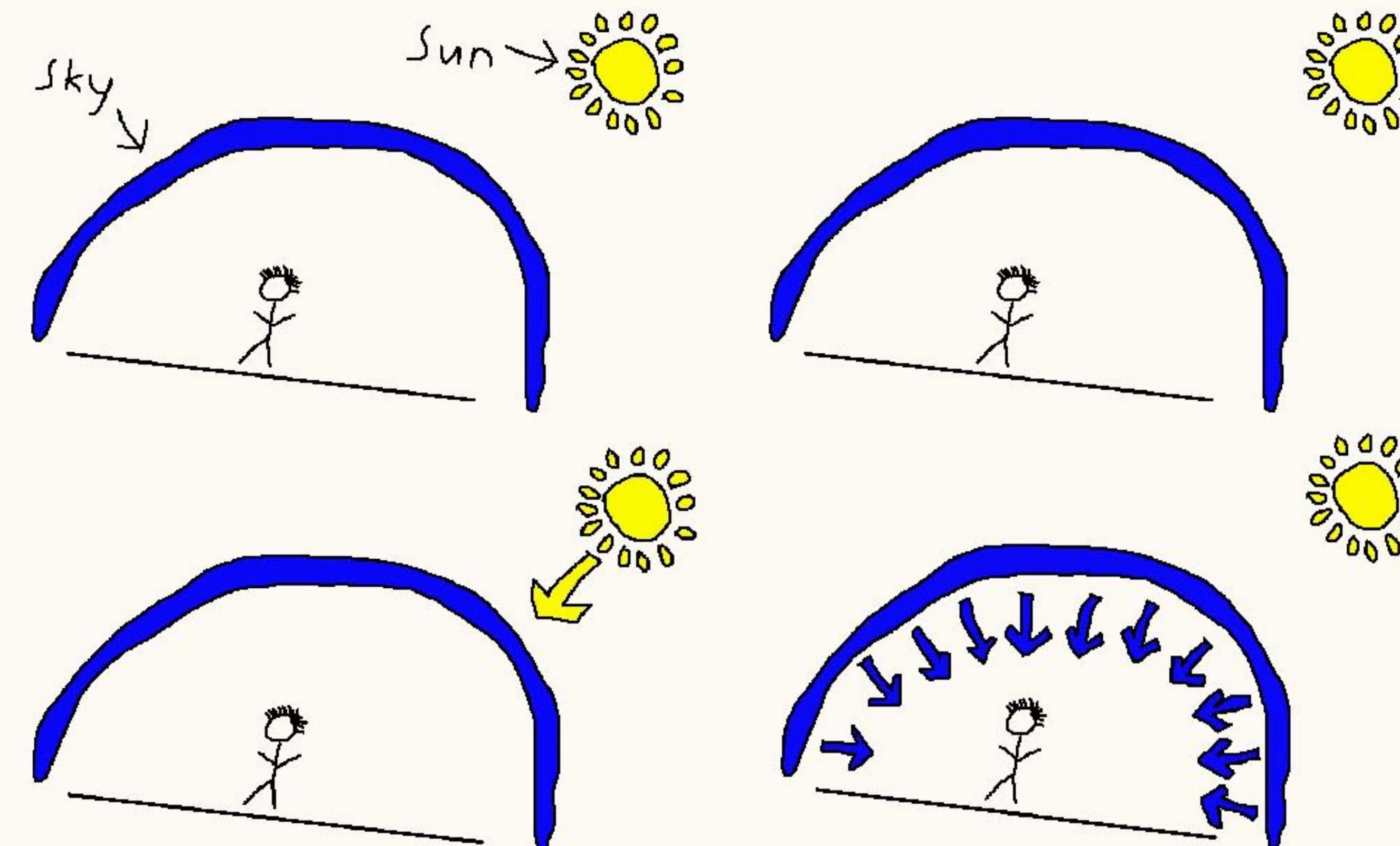
Local Realtime Image-based Lighting References

- “Local Image-based Lighting With Parallax-corrected Cubemap”
 - By Sébastien Lagarde, Antoine Zanuttini
 - SIGGRAPH 2012
 - “GPU Pro 4” book
- McTaggart, “Half-Life 2 Valve Source Shading” http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf
- Bjorke, “Image based lighting”, http://http.developer.nvidia.com/GPUGems/gpugems_ch19.html
- Behc, “Box projected cubemap environment mapping” <http://www.gamedev.net/topic/568829-box-projected-cubemap-environment-mapping/>
- Mad Mod Mike demo, “The Naked Truth Behind NVIDIA’s Demos”, ftp://ftp.up.ac.za/mirrors/www.nvidia.com/developer/presentations/2005/SIGGRAPH/Truth_About_NVIDIA_Demos.pdf
- Brennan, “Accurate Environment Mapped Reflections and Refractions by Adjusting for Object Distance”, http://developer.amd.com/media/gpu_assets/ShaderX_CubeEnvironmentMapCorrection.pdf

Ambient Occlusion

What's Ambient Occlusion ?

- 陽光是直接光照 - 產生影子 - Shadow map 方法
- 天光是環境光照 - 產生 Ambient Occlusion (環境遮蔽?)

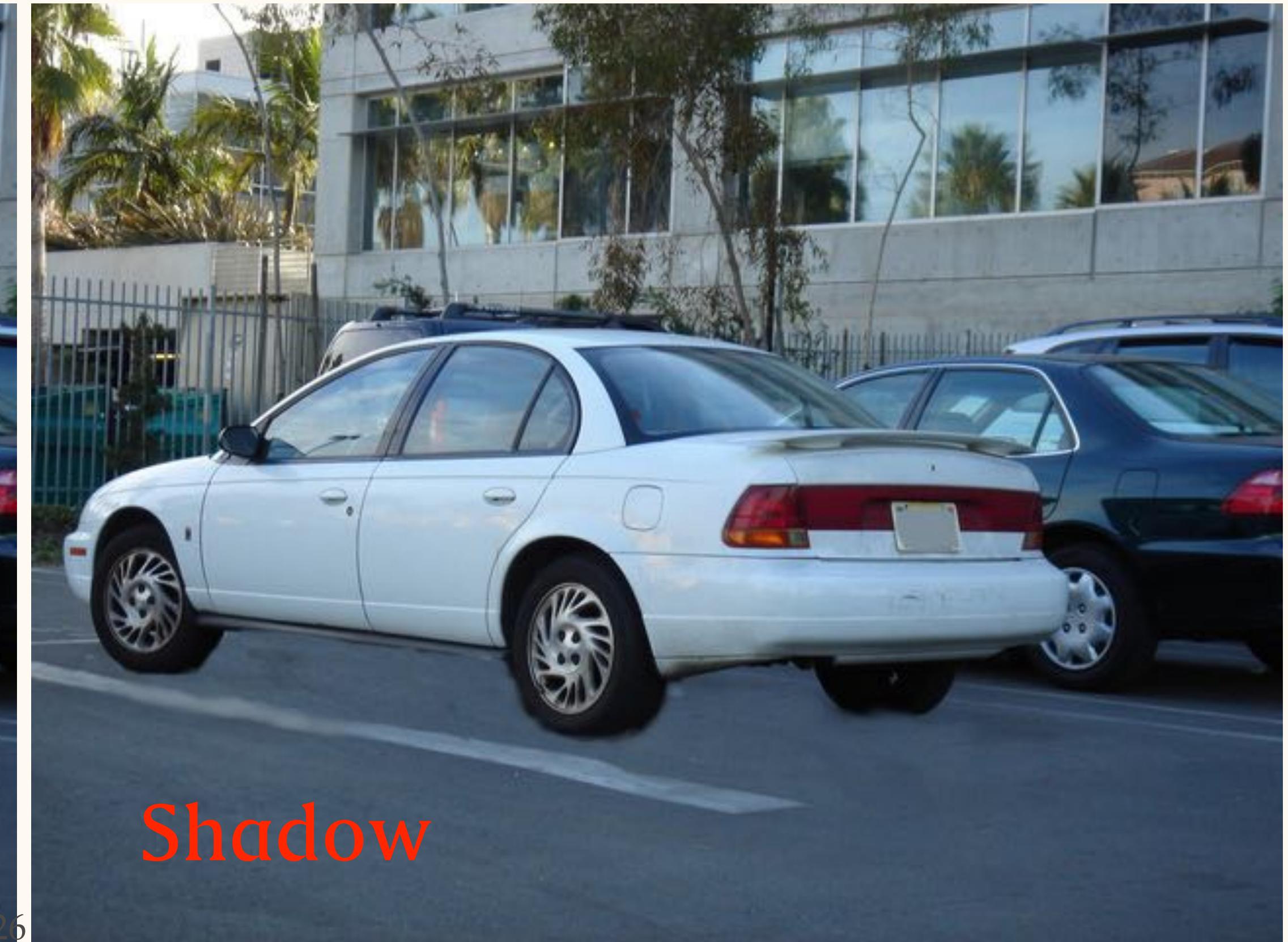


What's “Ambient Occlusion“ ?

- 緣起：
 - 請參考：GPU Gems, Chapter 17, P. 279-292
 - 最先是由 Hayden Landis (2002) 和其同事為了Industrial Lights & Magic (ILM)電影特效而開發出的做法
 - Pre-processing (事先烘焙)
 - 概念是計算環境光照到達物體表面的比例
 - 靜態光照
 - 簡稱 A.O.
 - 也被稱為 Ray traced shadow
- 現在遊戲技術普遍使用的是另一作法：
 - Screen-space ambient occlusion (SSAO)
 - SSAO != AO

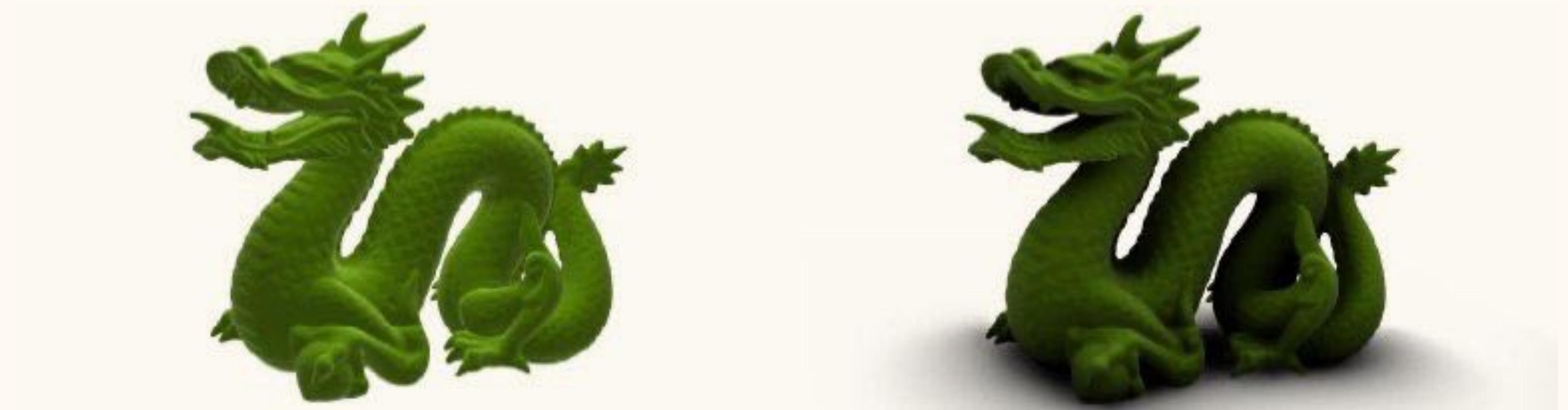
Why Ambient Occlusion ?

- 為什麼需要 ambient occlusion ?
 - 請參考下圖 :



Why Ambient Occlusion ?

- 給予 3D模型立體與空間感更具體，物件與物件的相鄰關係更清楚

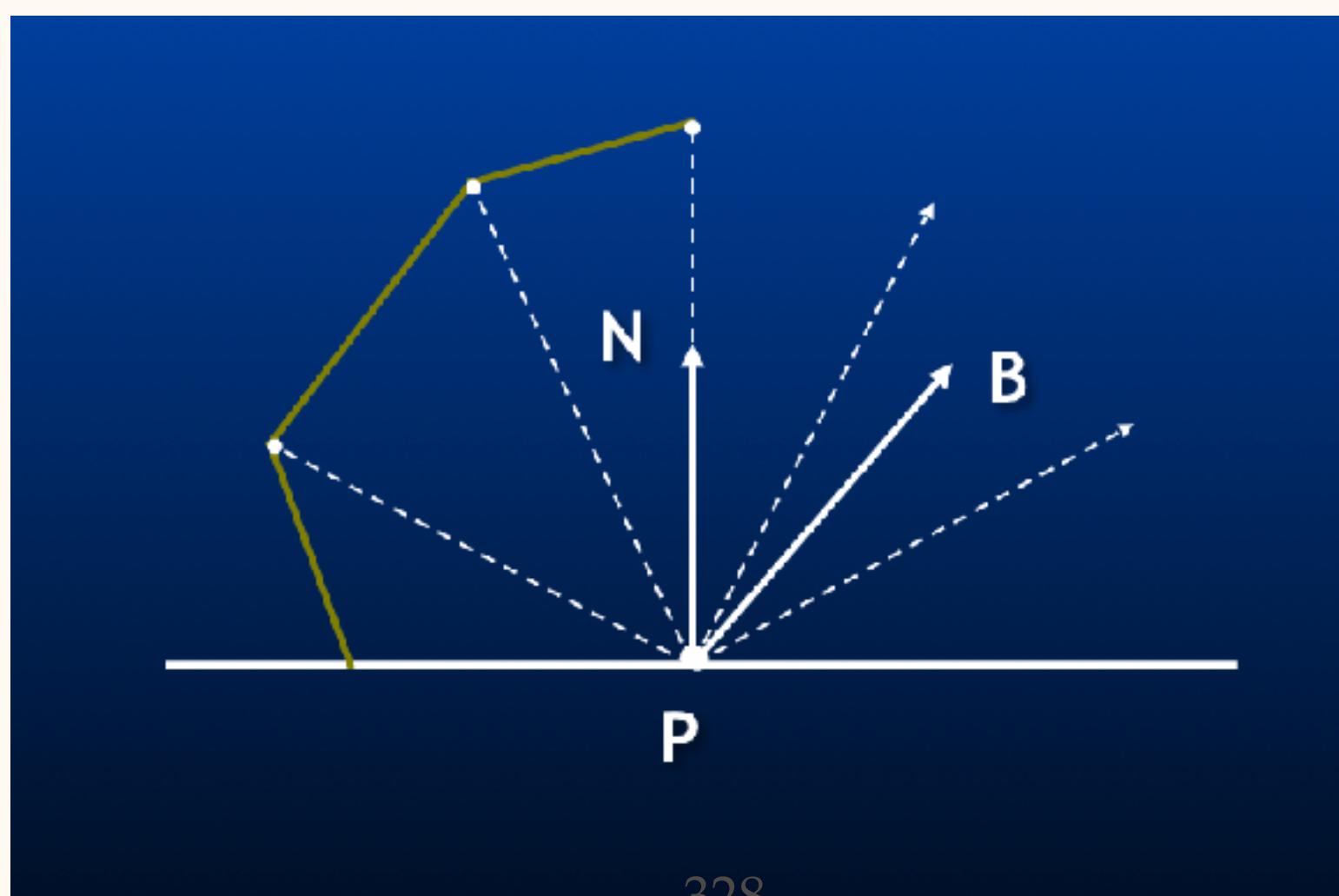


Without AO

With AO

Generate Ambient Occlusion on Vertex

- 在每個頂點上，我們事先計算出下列兩個數值：
 - 環境光照到達頂點的比率
 - 就是在頂點 P 上半球體的空間內，有多少比例的光線沒有被阻擋而能到達 P 點
 - 沒有被遮擋空間的平均入射角度， B
 - “Ray-traced Shadow”



Generate Ambient Occlusion on Vertex

- Algorithm :

```
int numRays; // number of rays
For each vertex
{
    Generate a set of rays over the hemisphere centered at the vertex normal
    Vector B = Vector(0, 0, 0);
    int numB = 0;
    For each ray
    {
        If (ray doesn't intersect anything)
        {
            B += ray.direction;
            ++numB;
        }
    }
    B = normalize(B);
    accessibility = numB / numRays;
}
```

Ambient Occlusion 品質

- 兩個因素：
 - 設限數量的多寡
 - 越多條射線形成的 A.O. 品質愈好，但所需的時間愈長
 - 一個適當的光線取樣的演算法會影響品質
 - 適當的空間分割方法，可加速解焦點的速度
 - 模型的解析度
 - 可以將 A.O. 儲存在 Texture 中
 - Ambient occlusion map
 - 大多數的動畫軟體與遊戲引擎工具可以產生 A.O. map

Ray Generation

- 介紹兩種方法：
 - Rejection sampling
 - 取樣率非常重要
 - Monte Carlo sampling algorithms
 - 同樣的取樣率時可以得到比較好的結果

Rejection Sampling

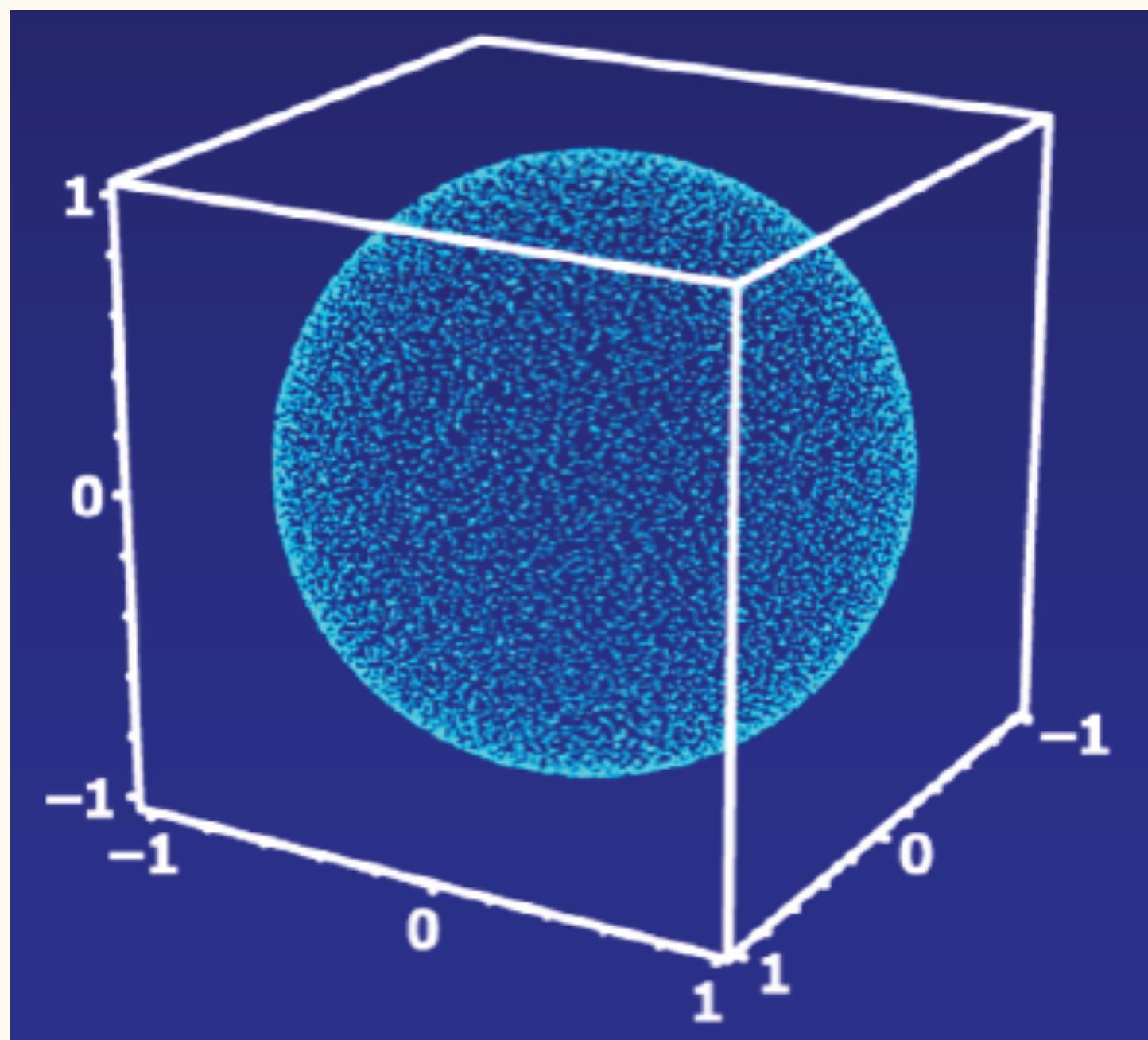
- Algorithm :

```
while (TRUE) {
    x = RandomFloat(-1, 1); // random float between -1 & 1
    y = RandomFloat(-1, 1);
    z = RandomFloat(-1, 1);
    if (x*x + y*y + z*z > 1) continue; // ignore ones outside unit sphere

    if (dot(Vector(x, y, z), N) < 0) continue; // ignore "down" direction ray
    return normalize(Vector(x, y, z));
}
```

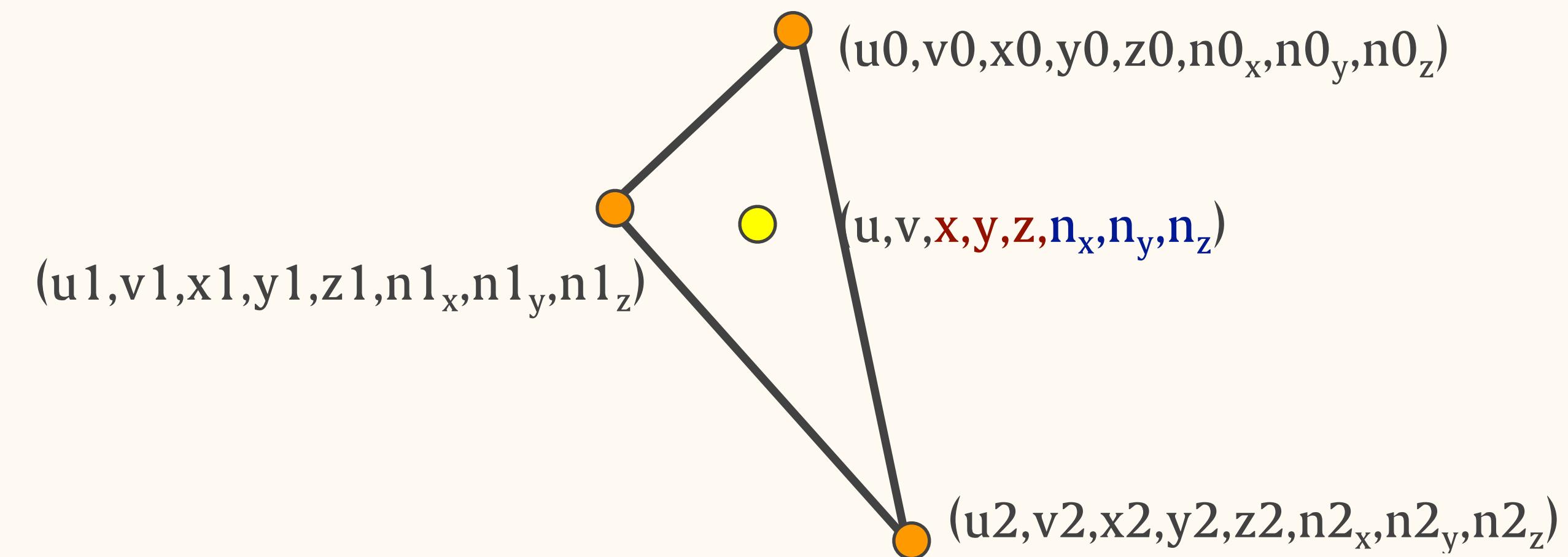
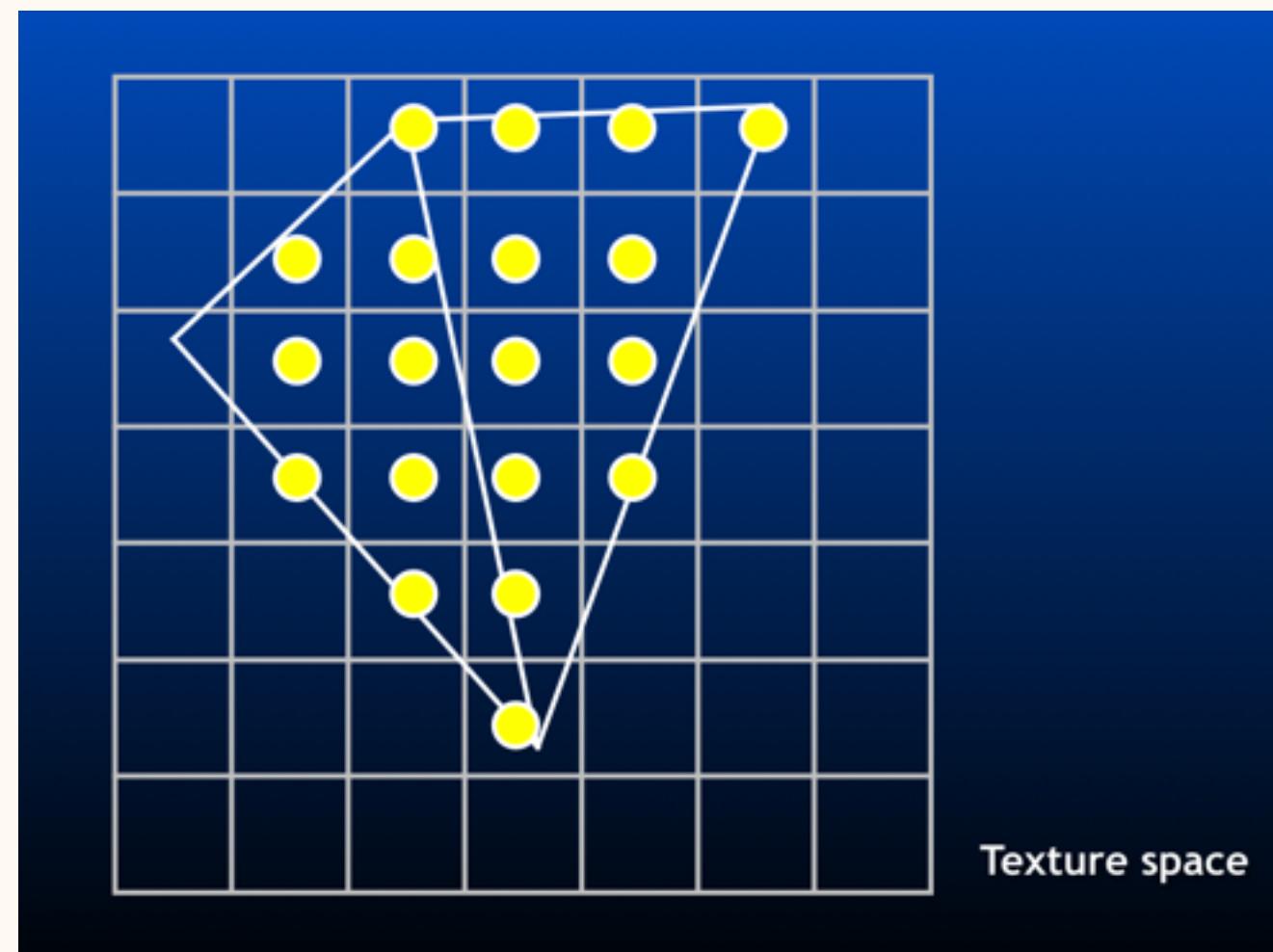
Monte Carlo Sampling

- Generate unbiased points over a sphere
 - Map points from a unit square into a spherical coordinates



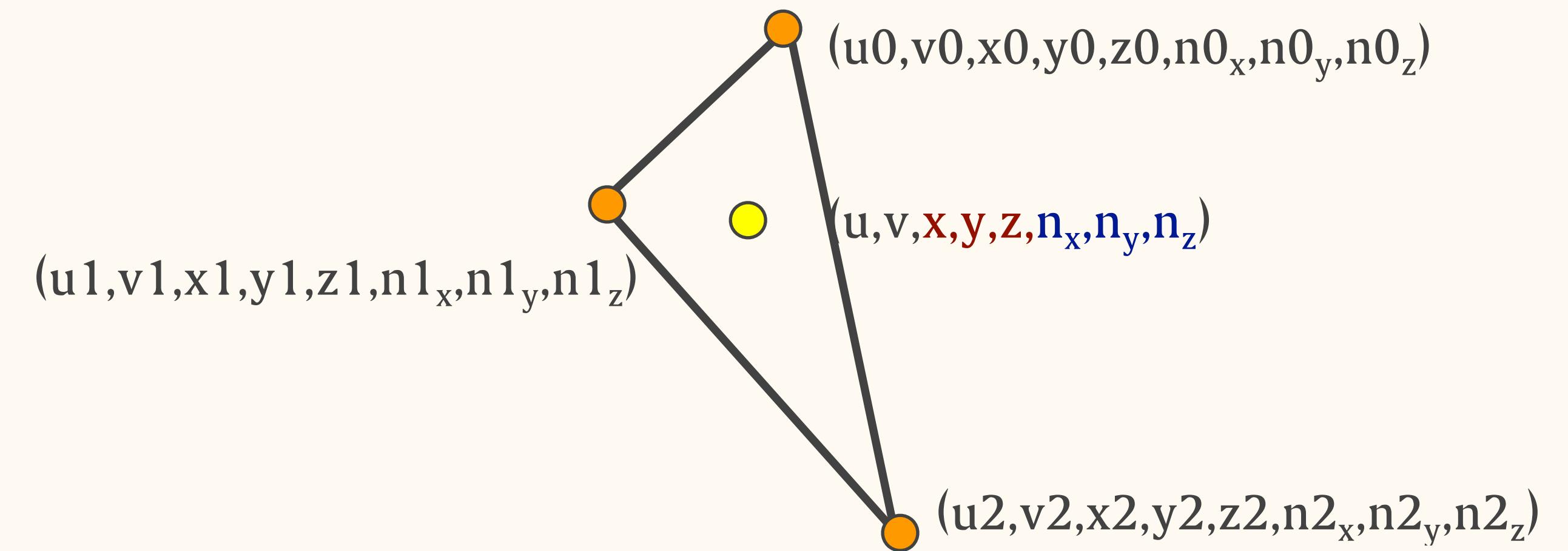
Generate Ambient Occlusion Map

- 對於低面數的模型可使用 ambient occlusion map
- 步驟：
 - Unwrap the model on texture



Generate Ambient Occlusion Map

- Use texture coordinate to interpolate the texel's position and normal vector in 3D world



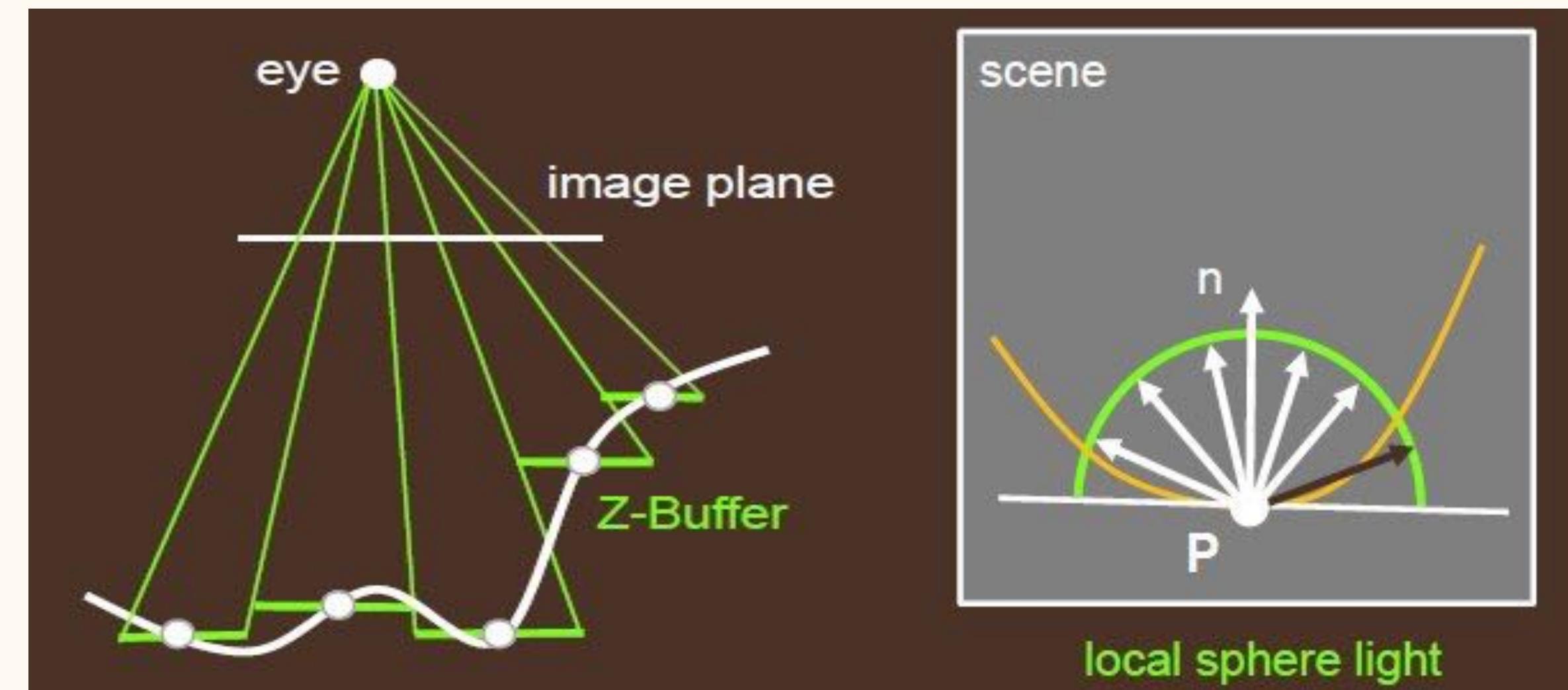
- Follow the AO algorithm to generate the AO data
- Save AO value to the pixel

Screen Space Ambient Occlusion

- Screen-space Ambient Occlusion
 - SSAO in short
- 最早出現在 CryEngine 2 遊戲引擎
 - MITTRING, M. 2007. “Finding next gen: Cry Engine 2”. In SIGGRAPH ’07: ACM SIGGRAPH 2007 courses.
- 是 deferred rendering engine 的副產品，也可以獨立使用
 - Z buffer + normal buffer

Screen Space Ambient Occlusion

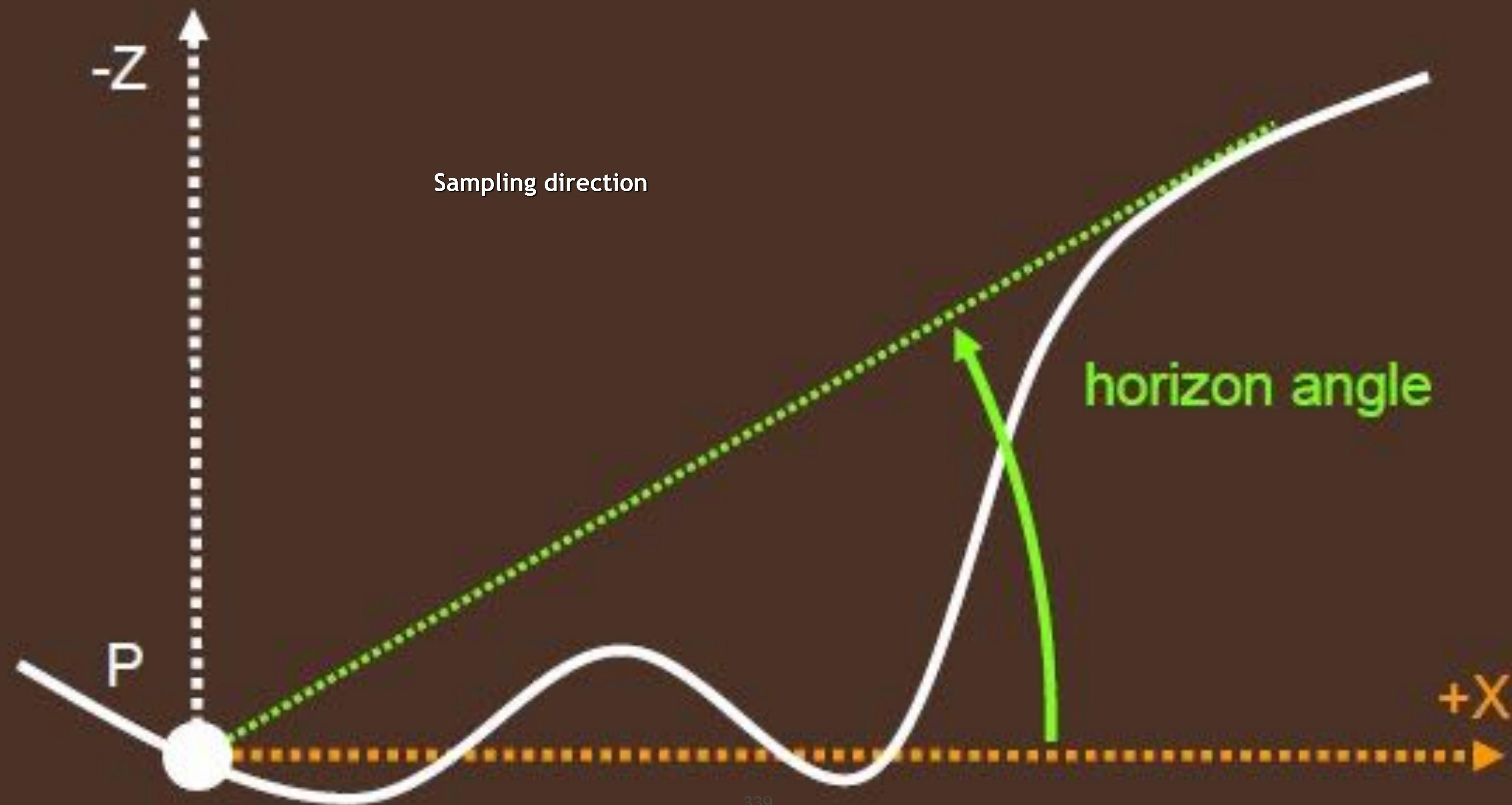
- Screen space
 - Data in screen coordinates
 - Render the 3D geometry data in buffers
- SSAO 的特點就是可以即時有效地計算每個像素上的 A.O. 值
- Input
 - Depth buffer (Height field), $Z = f(x, y)$
 - Normal buffer



Screen Space Ambient Occlusion

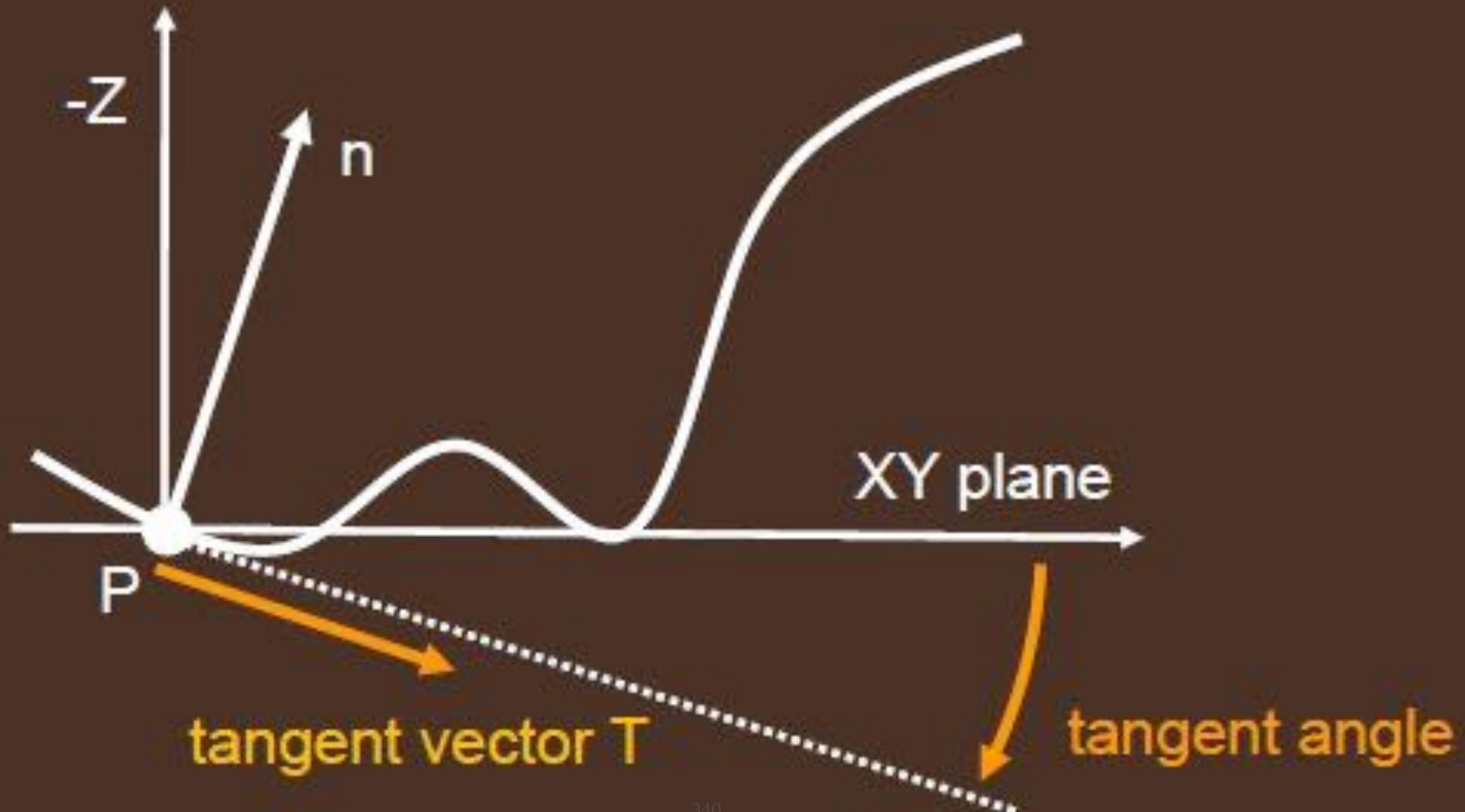
- 在此介紹 nVidia 開發的方法：
 - “Image-Space Horizon-Based Ambient Occlusion”
 - By Louis Bavoil and Miguel Sainz @ nVIDIA
 - Published in ShaderX 7
 - Presented in SIGGRAPH 2008
 - HBAO

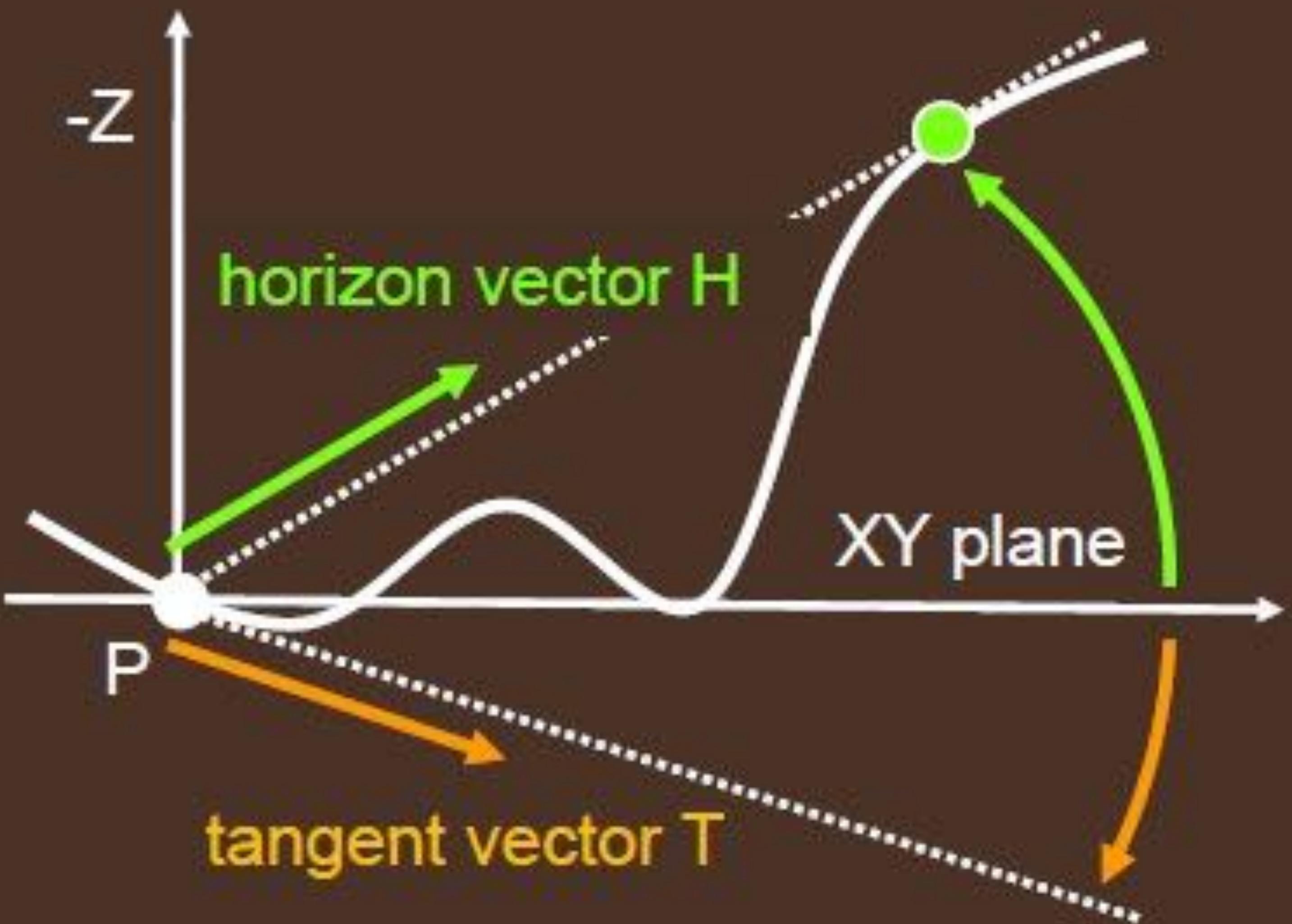




Sampling direction

horizon angle

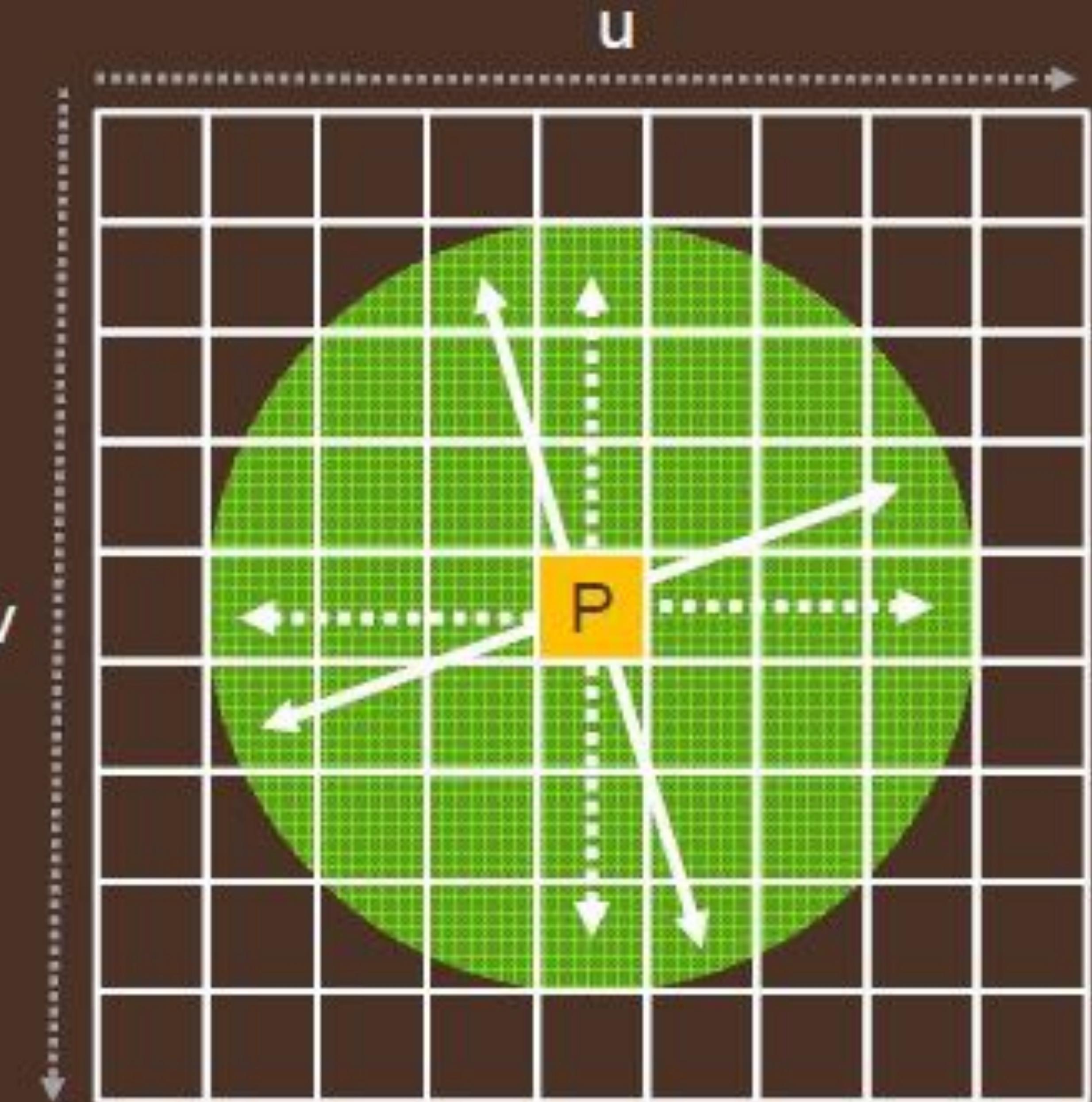
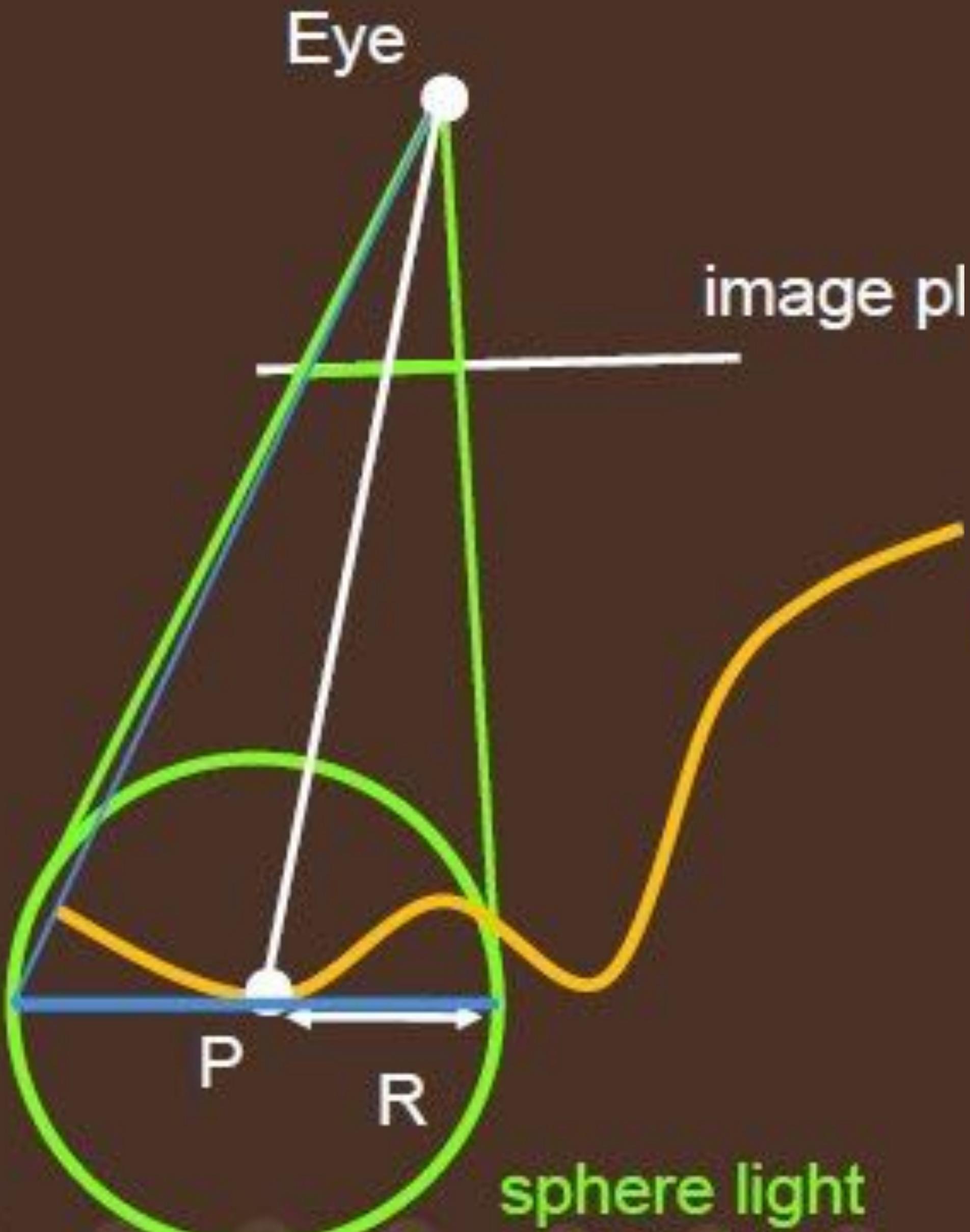




horizon angle in $[-\pi/2, \pi/2]$
 $h(H) = \text{atan}(H.z / \|H.xy\|)$

tangent angle in $[-\pi/2, \pi/2]$
 $t(T) = \text{atan}(T.z / \|T.xy\|)$

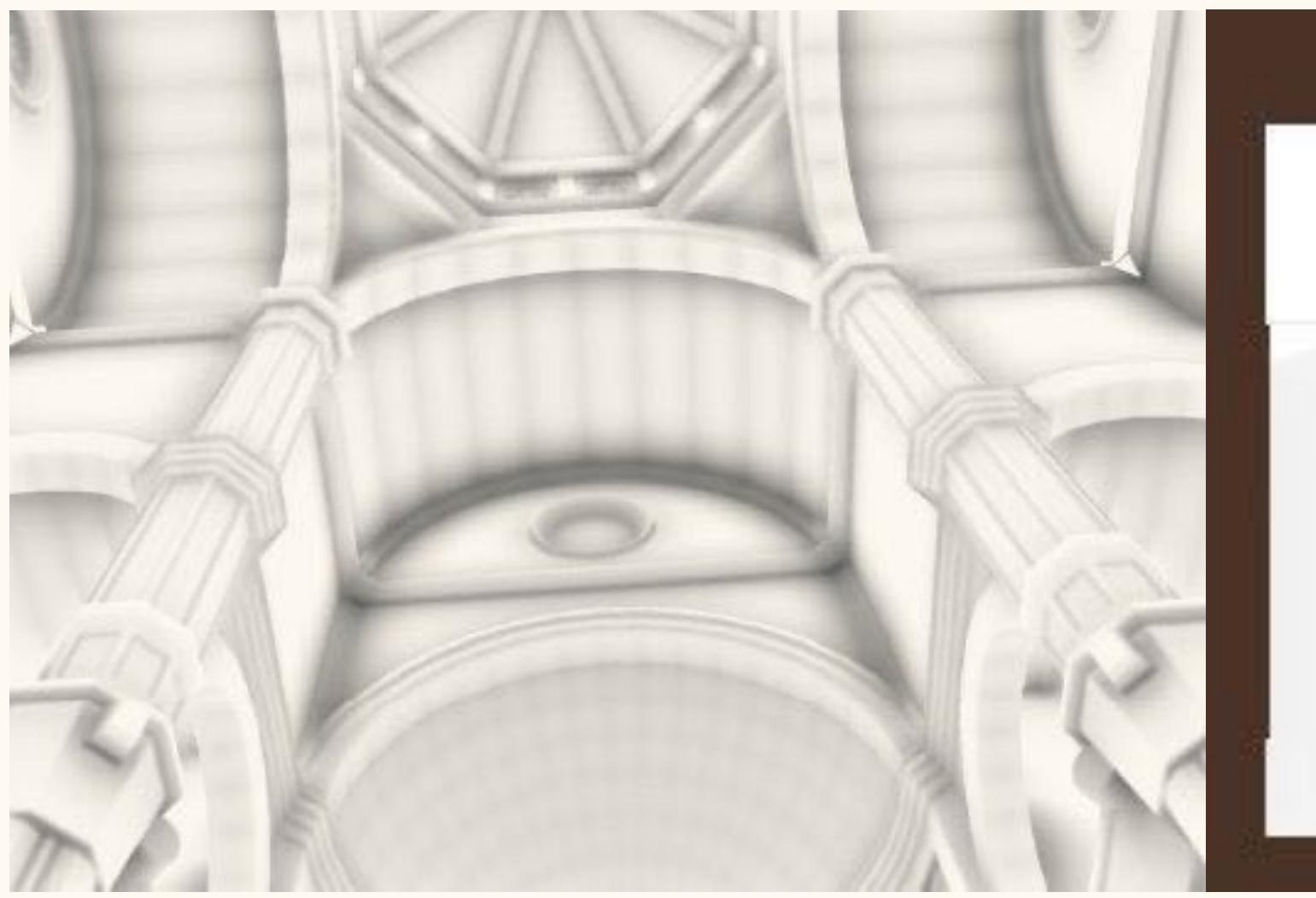
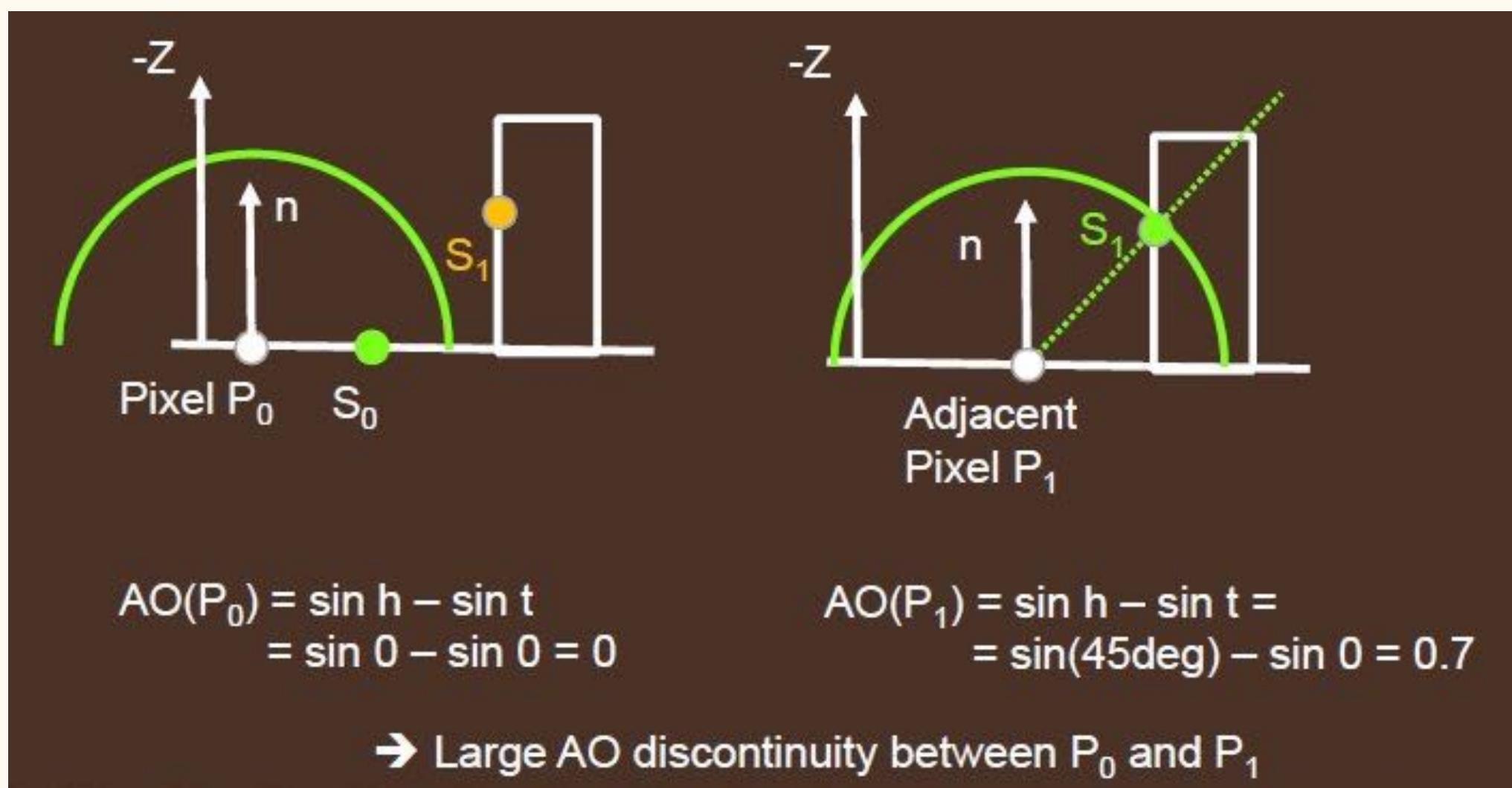
$$AO = \sin h - \sin t$$



Example with 4 directions / pixel

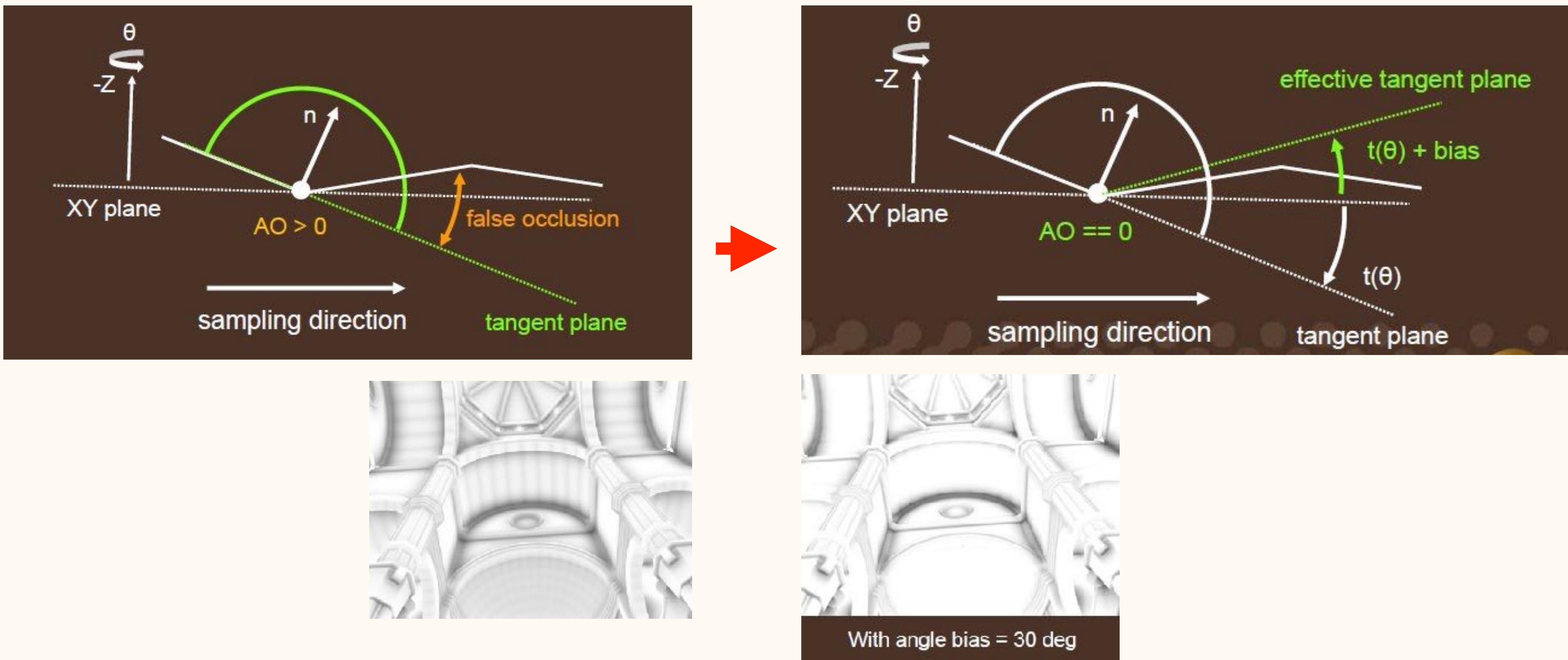
SSAO 常見的問題

- Ambient occlusion in creases
- Sampling outside the screen
- Discontinuity problems
- Noise



Ambient Occlusion in Creases

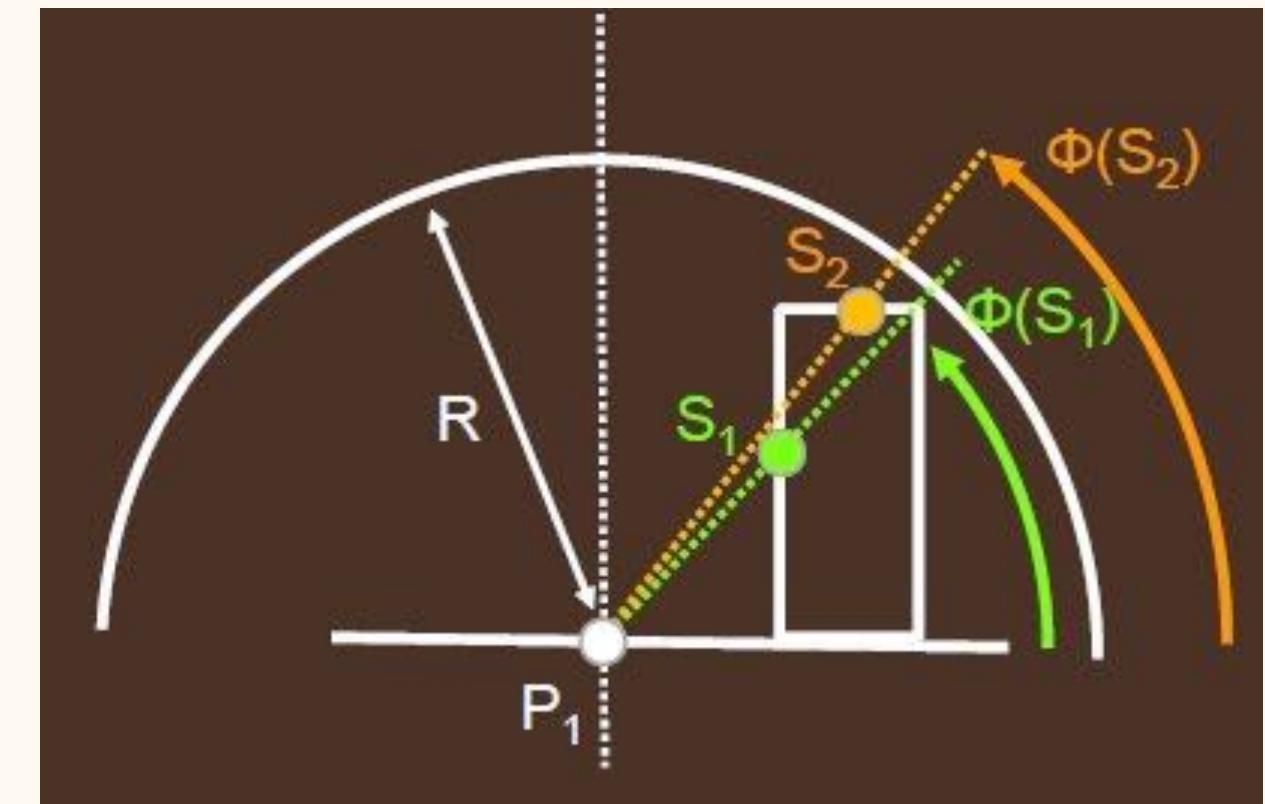
- Low-tessellation problem (低面數造成的)
- 加一個 angle bias 忽略靠近 tangent plane 的 A.O. 值來改善之



Ambient Occlusion Discontinuity Problem

- 加 attenuation function 作為權重來改善之
 - $W(r)$
 - “Falloff”
 - 考慮距離的因素

$$r = \frac{\| S - P \|}{R}$$



$$W(r) = 1 - r^2$$

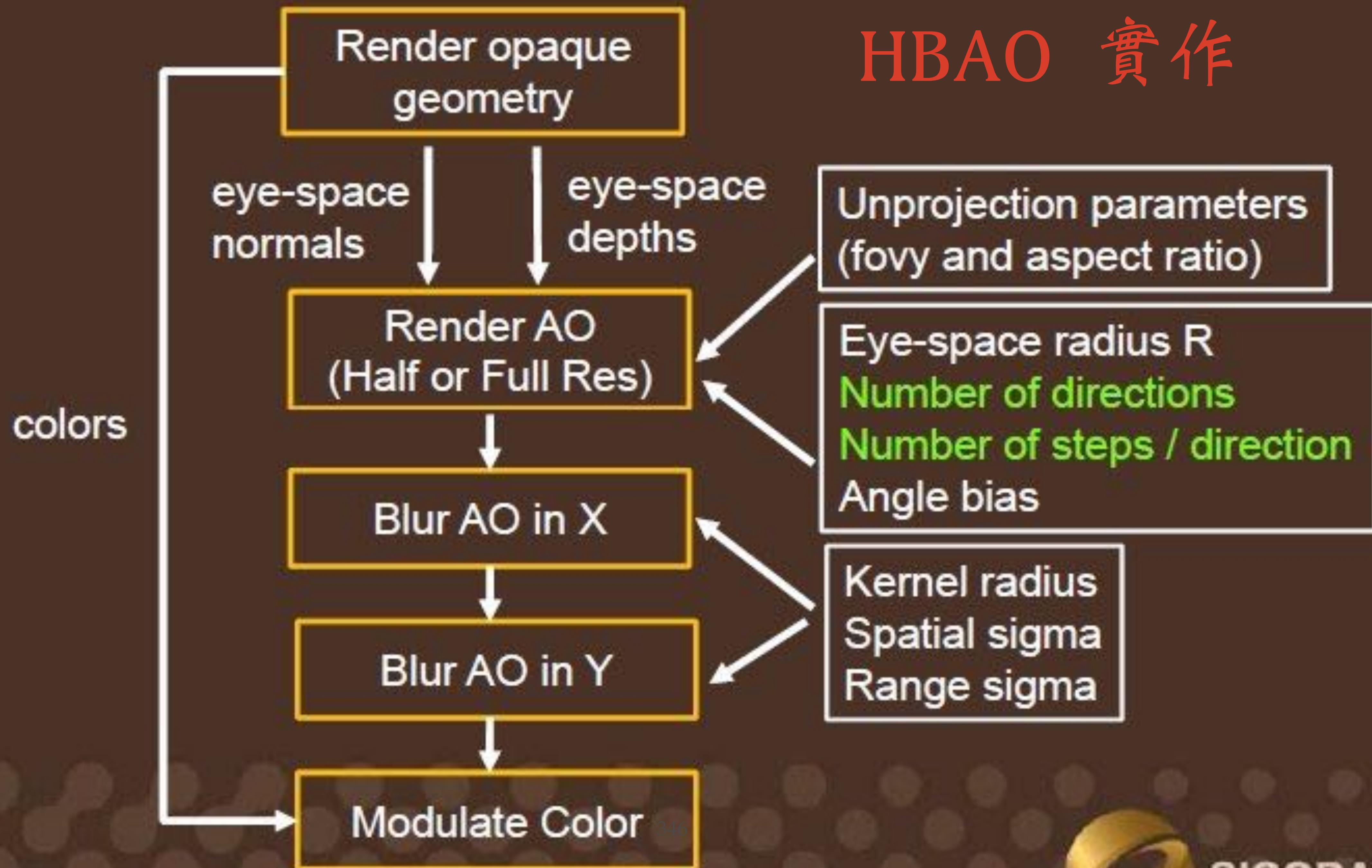


With Attenuation
 $W(r) = 1 - r^2$



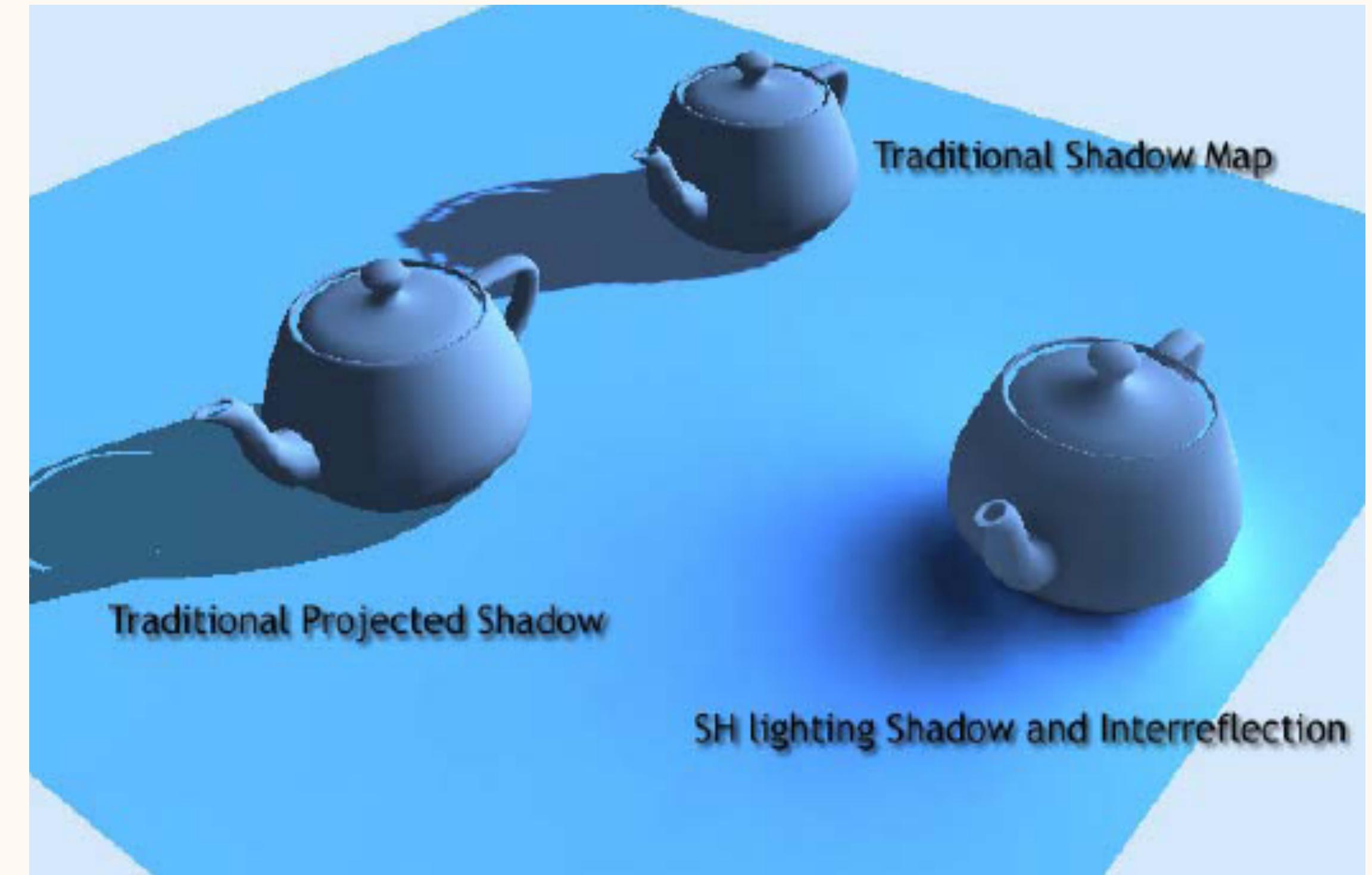
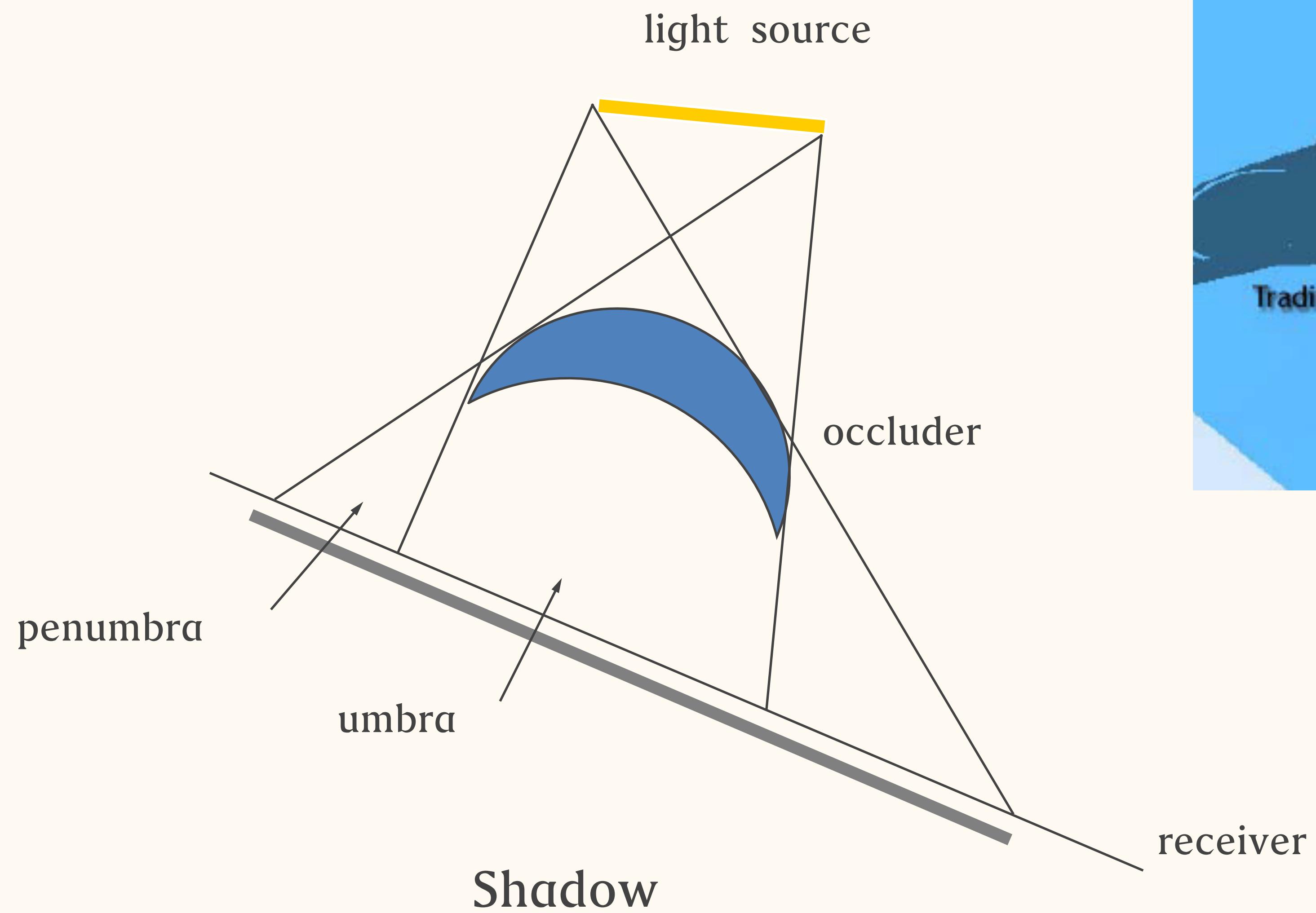
Without Attenuation
 $W(r) = 1$

HBAO 實作



SIGGRAPH

Shadow Map



Introduction to Shadow Map

- 緣起
 - By Lance Williams
 - SIGGRAPH 1978 Paper
 - “Casting Curved Shadows on Curved Surfaces”
- 概念：
 - “A point is shadowed if something interrupts the straight line path between the light and that point.”

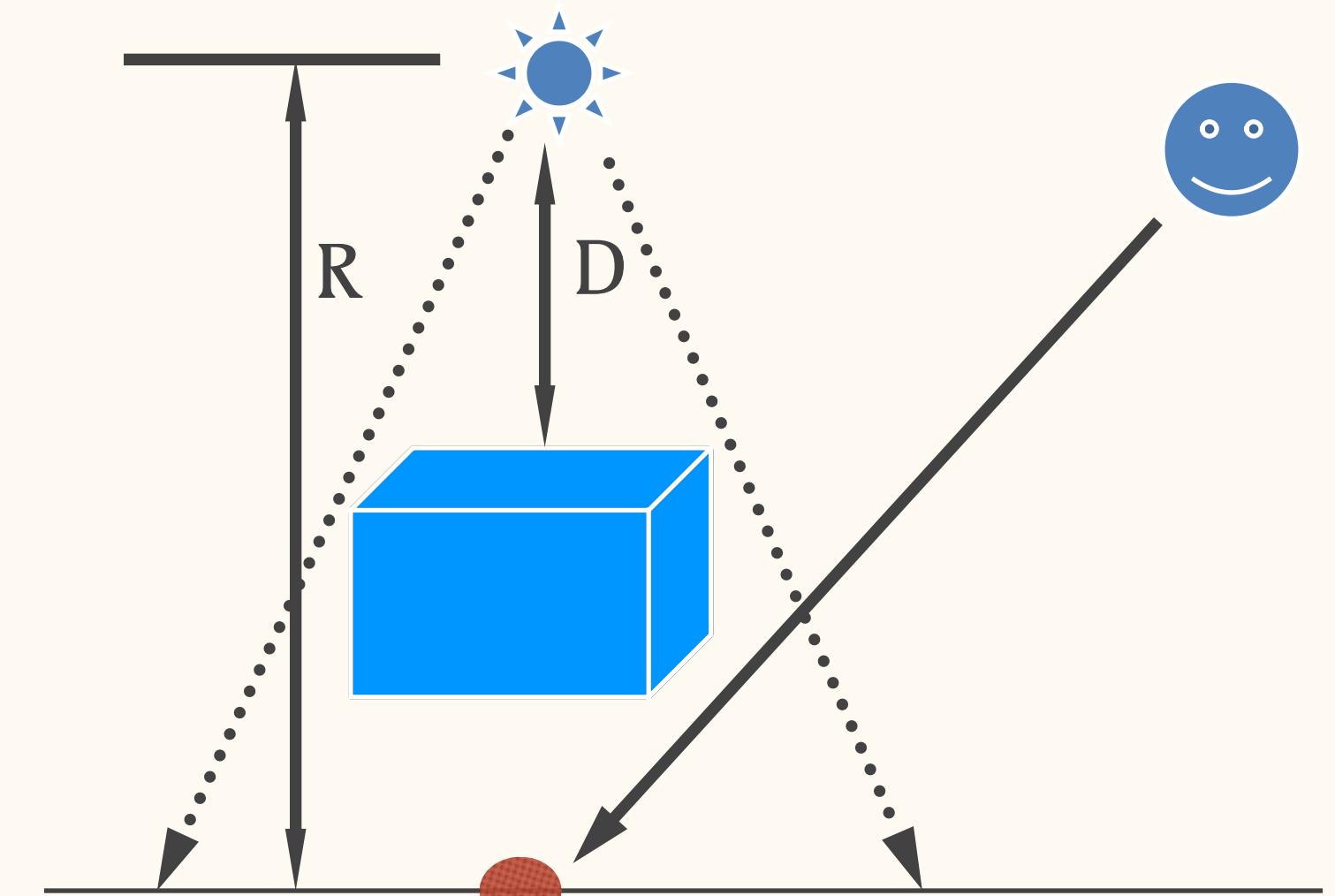
Introduction to Shadow Map

- 方法：
 - 在 3D 炫染前，先以光源為攝影機視角，將所視的物件渲染在一張貼圖中，渲染的值是所看到的物件至光源的距離或遠近的深度，此貼圖稱為 shadow map，是一種 depth map

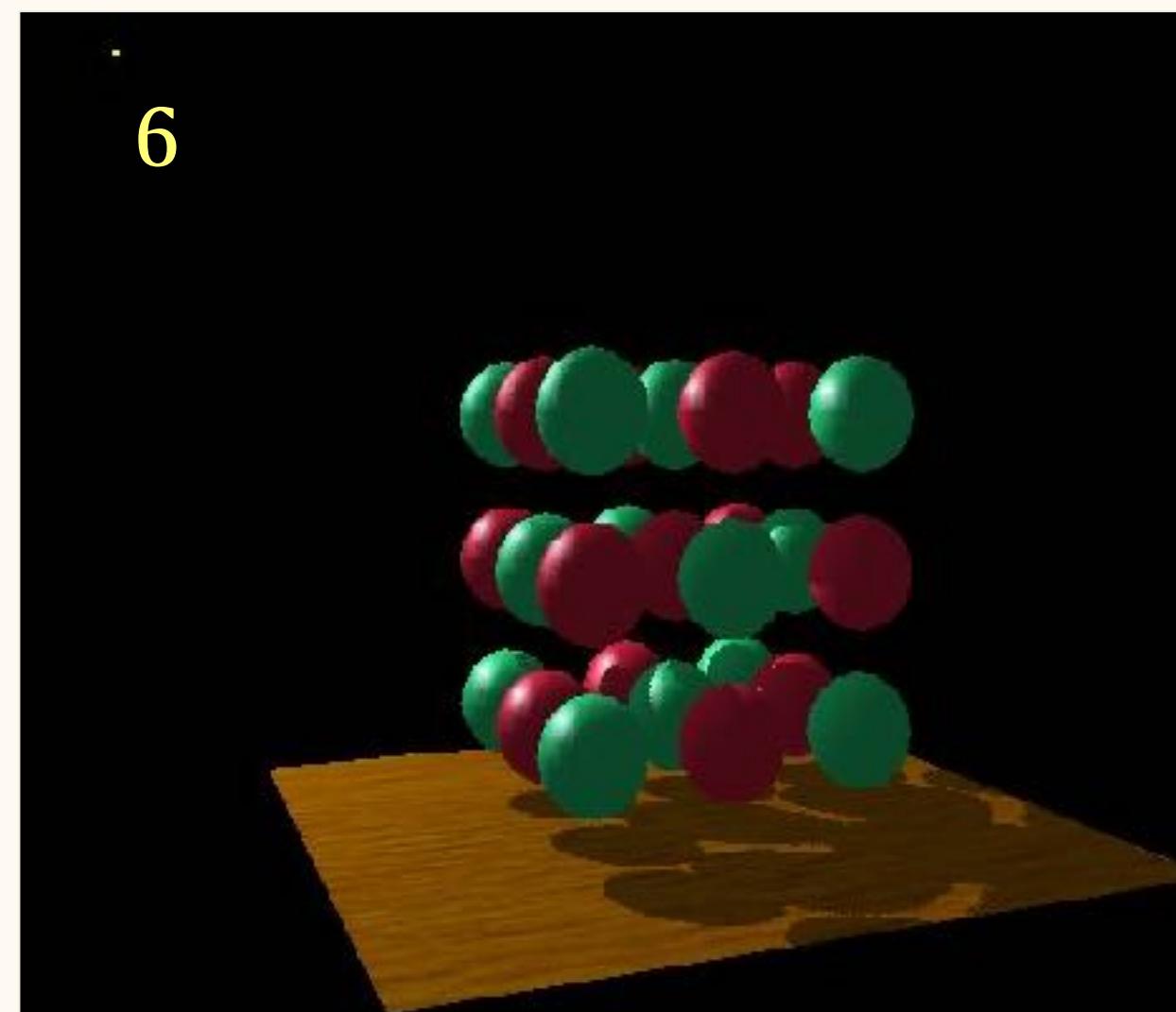
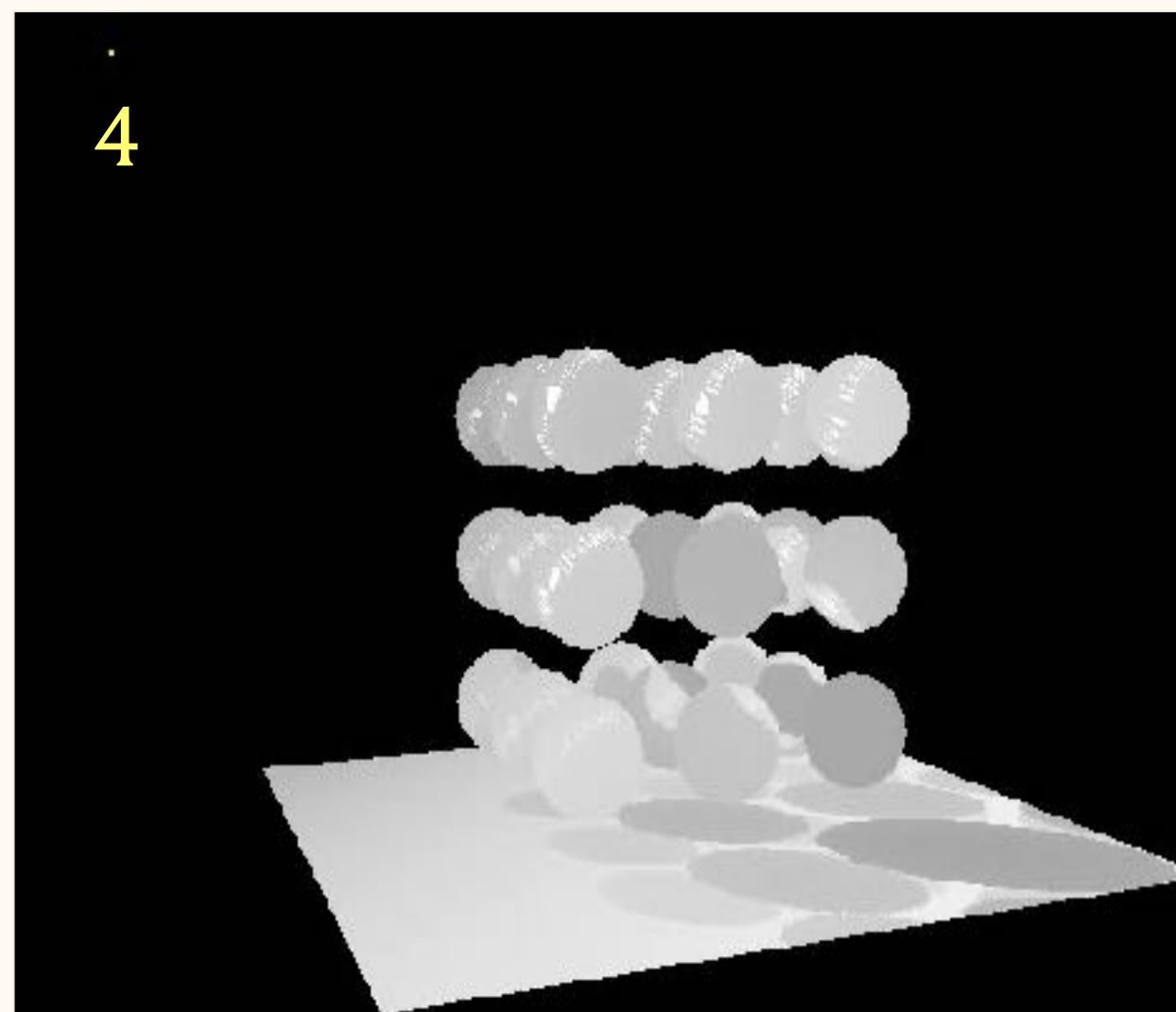
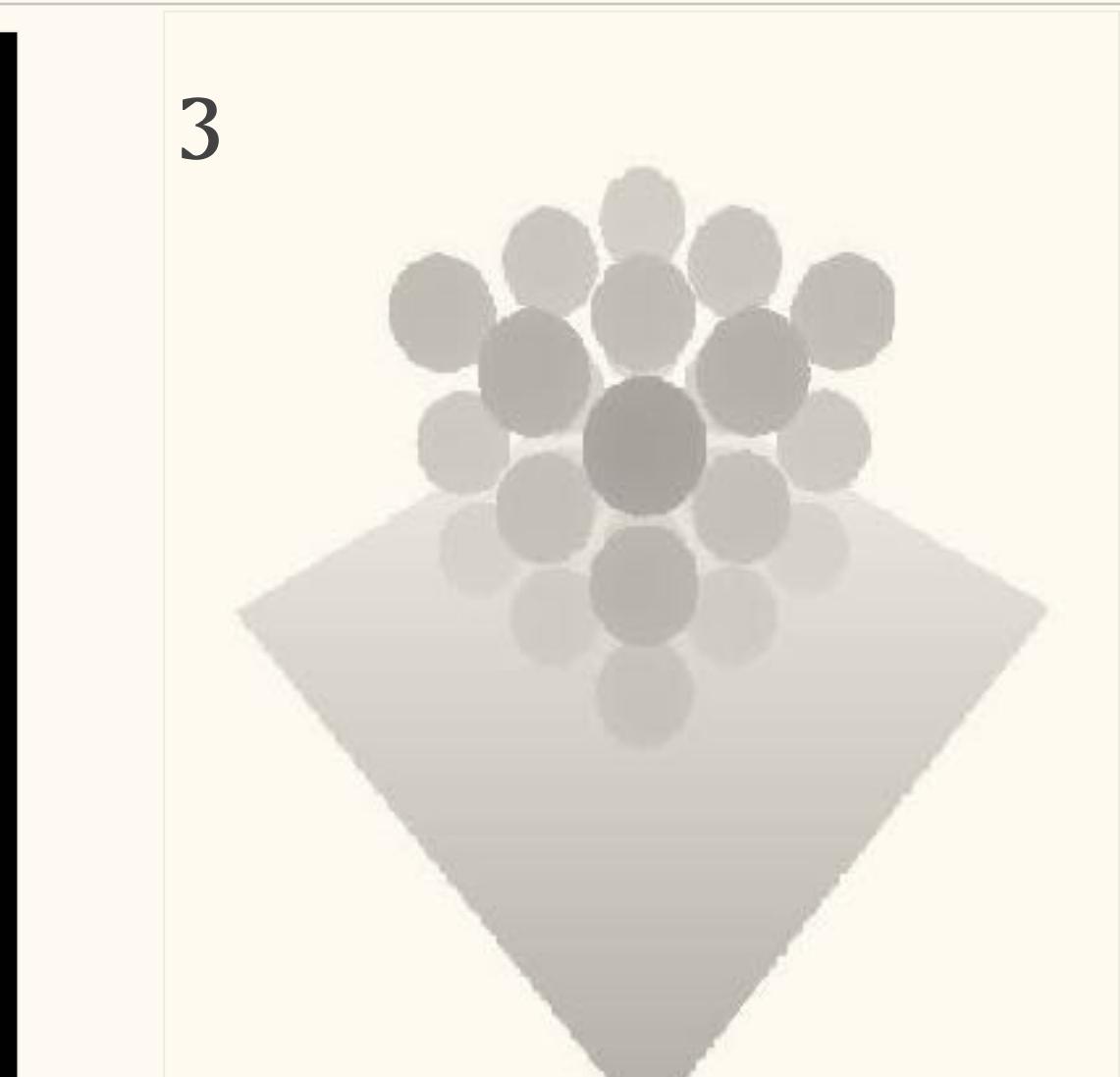
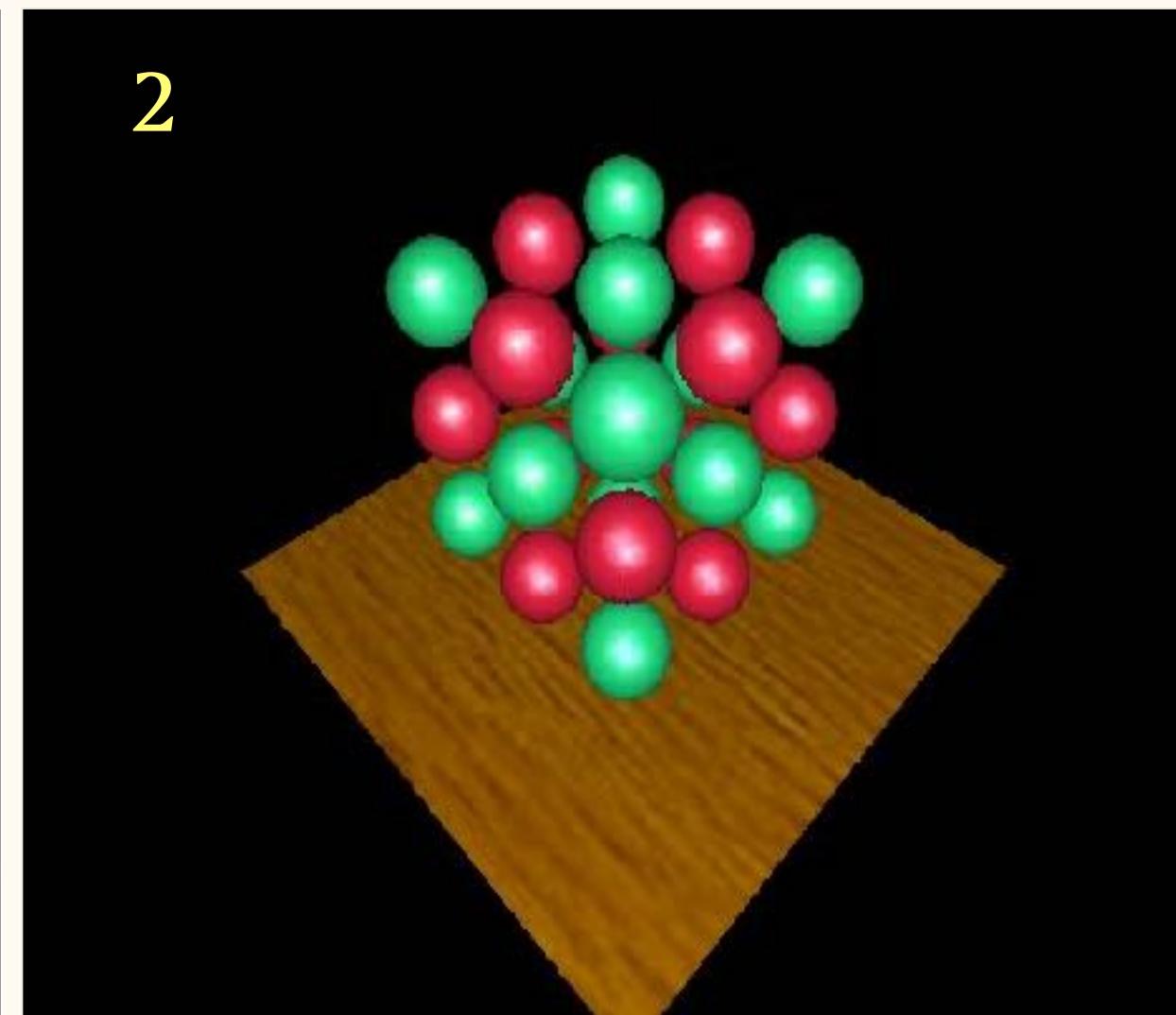
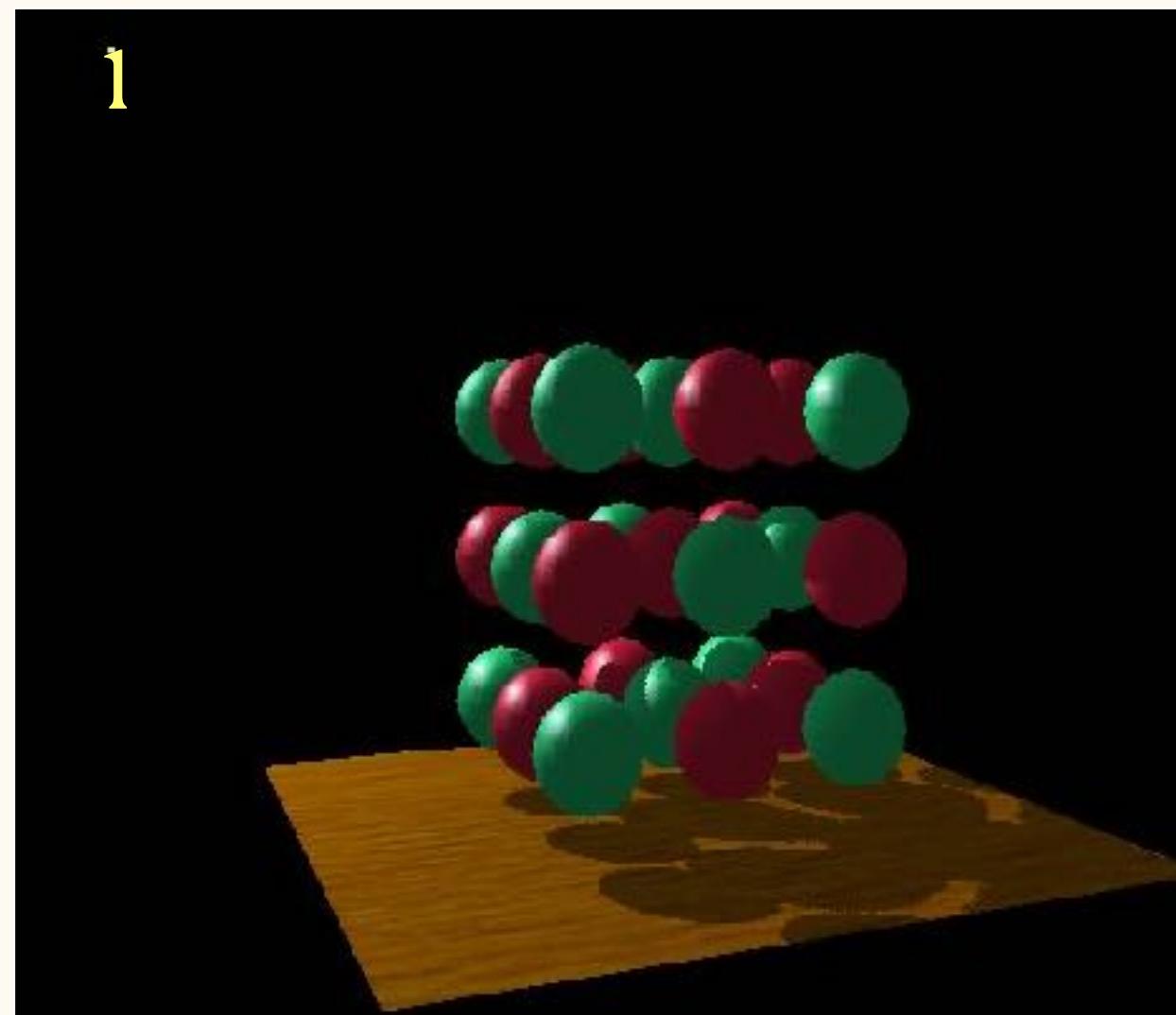


Introduction to Shadow Map

- 在正式渲染時，使用 projective texture mapping 技術將 shadow map 投射在 3D 模型上
- 因此，在每個像素渲染時，可以由 shaow map 查得到擬渲染之處最靠近光源的物件點之深度值， D
- R 是擬渲染之處到光源的直線深度資料
 - $R = D$: The point is not occluded.
 - $R > D$: The point is in the shadow.



Shadow Map Step-by-step



Shadow Map As a Two-pass Processing

- 1st pass
 - 在光源位置放置適當的攝影機
 - 如果是投射燈光，攝影機是透視攝影機 (perspective projection)
 - 如果是平行光光源，攝影機是正投影攝影機 (orthogonal projection)
 - 如果是點光源，shadow map 是 cubemap 形式，需要渲染 6個方向.
 - 建議使用 floating-point texture buffer
- 2nd pass
 - 正式渲染
 - 渲染時先計算渲染位置到光源的距離
 - 再和 shadow map 對應的深度值比對，判斷是否在影子內？
 - 如果在影子內，就跳過該光源 lighting 的計算

Render a Shadow Map

- 對於點光源與投射燈光源，深度的算法建議：
 - 計算到光源的距離：
 - The value is normalized by lighting range

```
float dist2 = dot(in1.wPos, in1.wPos);
float dist = sqrt(dist2)/lightingRange;
```
- 對於平行光源，深度的算法建議：
 - 使用到光源的 z 值 (depth value, normalized by lighting range)
 - “depth map”



```
float dist = abs(in1.wPos.z)/lightingRange;
```

Projective Texture Mapping

- 投射一張 2D 影像於 3D models
 - Batman sign, spinning fan on ceiling, Shadow mapping, Spotlight effects
- Methods :
 - 把模型 3D 位置投影到 projection space (將投影中心點視為攝影機位置)
 - 計算 homogeneous divide
 - 將結果轉換至 texture space (be sure in left-handed on DirectX)
 - 查出 color 值

```
// convert location to screen space (the projection camera)
float4 posLgt = mul(mLightVP, in1.wPos);      // posLgt is in (-w ~ w)

// homogeneous divide and convert to texture space (0 ~ 1) (left-handed)
float2 lgtTexCoord = (posLgt.xy / posLgt.w) * float2(0.5f, -0.5f) + 0.5f;
float4 data = texture.Sampler(sampler, lgtTexCoord);
```

(Spot-Light) Shadow Map Shader

```
// convert location to screen space (lighting camera)
float4 posLgt = mul(mLightVP, in1.wPos); // posLgt will be in (-w ~ w)

// homogeneous divide and convert to left-handed texture space (0 ~ 1)
float2 lgtTexCoord = (posLgt.xy/posLgt.w)*float2(0.5f, -0.5f) + 0.5f;
float4 smRGB = shadowMap.Sample(shadowMapSampler, lgtTexCoord);

// find the distance parameter to light source
float3 L = -in1.lgtVec;
float3 Ln = normalize(L);
float aToL2 = dot(L, L);
float aToL = sqrt(aToL2)/lightingRange;

// calculate lighting weight according to the shadow map
float weight = (aToL - smRGB.x < bias) ? 1.0: 0.0;
lightIntensity *= weight;
```

(Parallel Light) Shadow Map Shader

```
// convert location to screen space (lighting camera)
float4 posLgt = mul(mLightVP, in1.wPos);      // posLgt will be in (-w ~ w)

// homogeneous divide and convert to left-handed texture space (0 ~ 1)
float2 lgtTexCoord = (posLgt.xy/posLgt.w)*float2(0.5f, -0.5f) + 0.5f;
float4 smRGB = shadowMap.Sample(shadowMapSampler, lgtTexCoord);

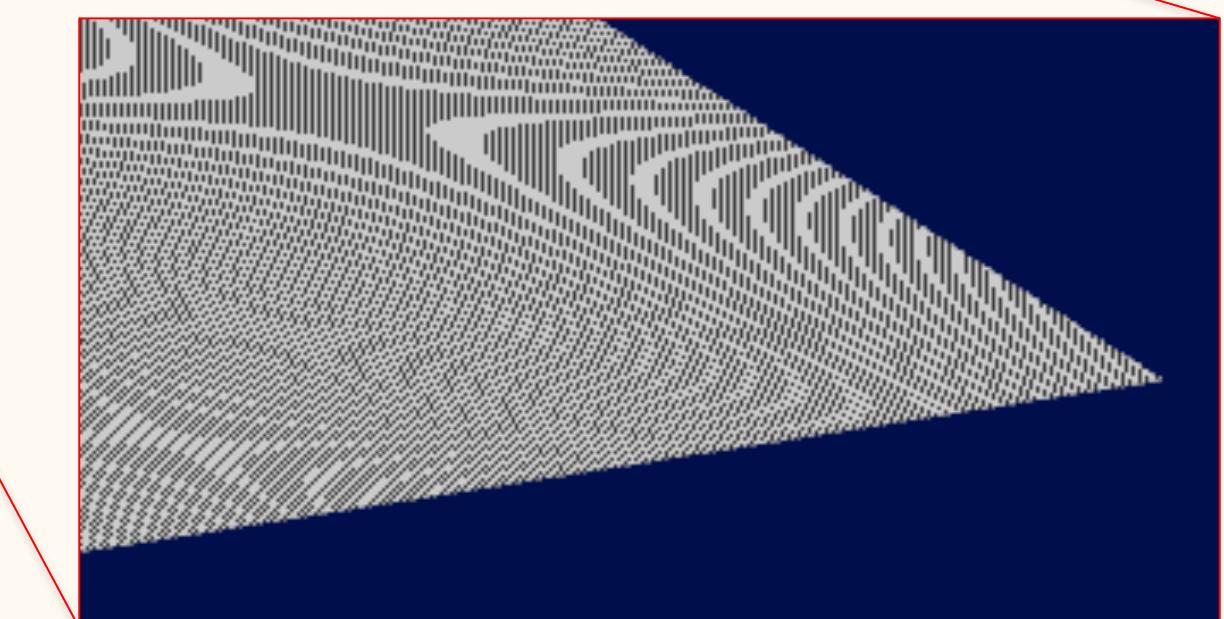
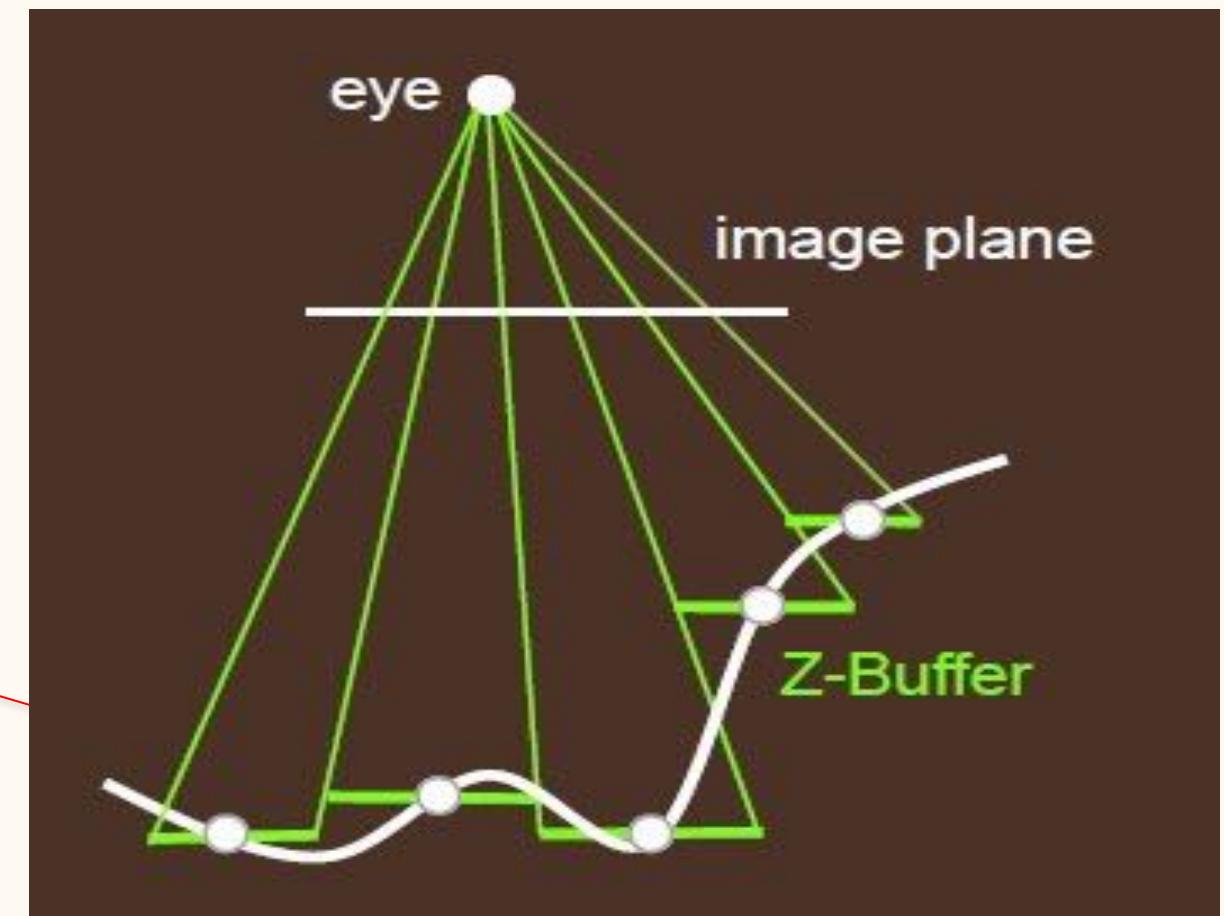
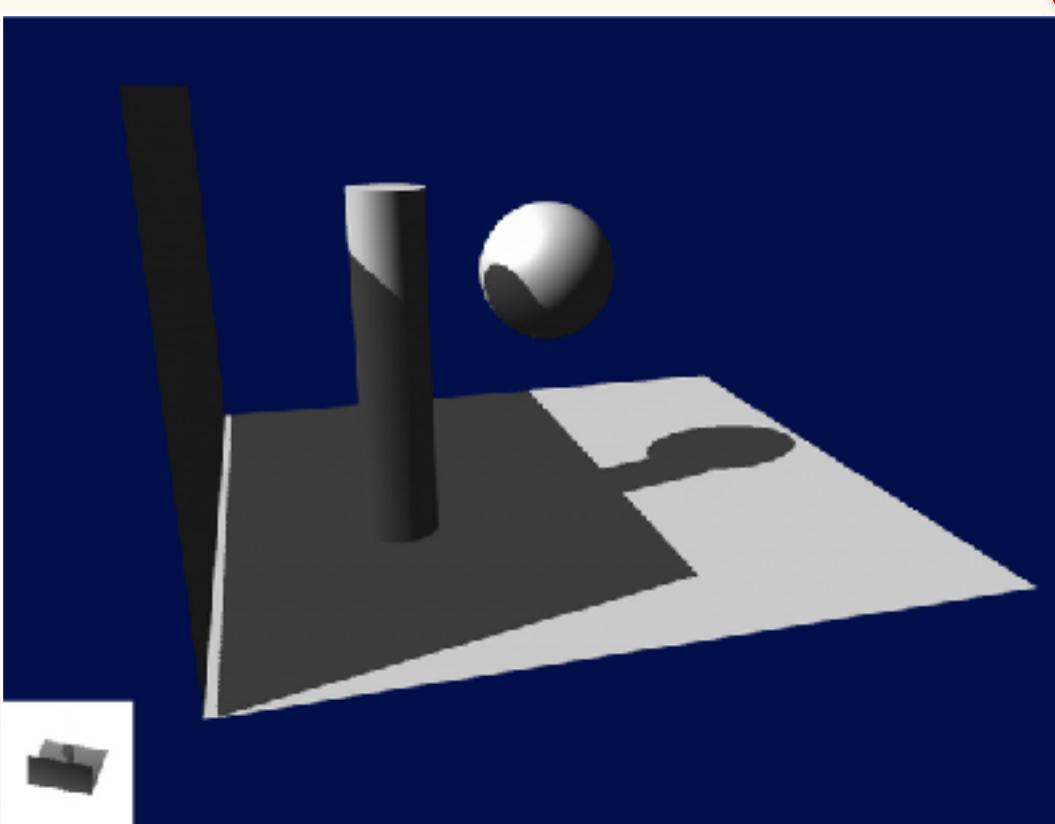
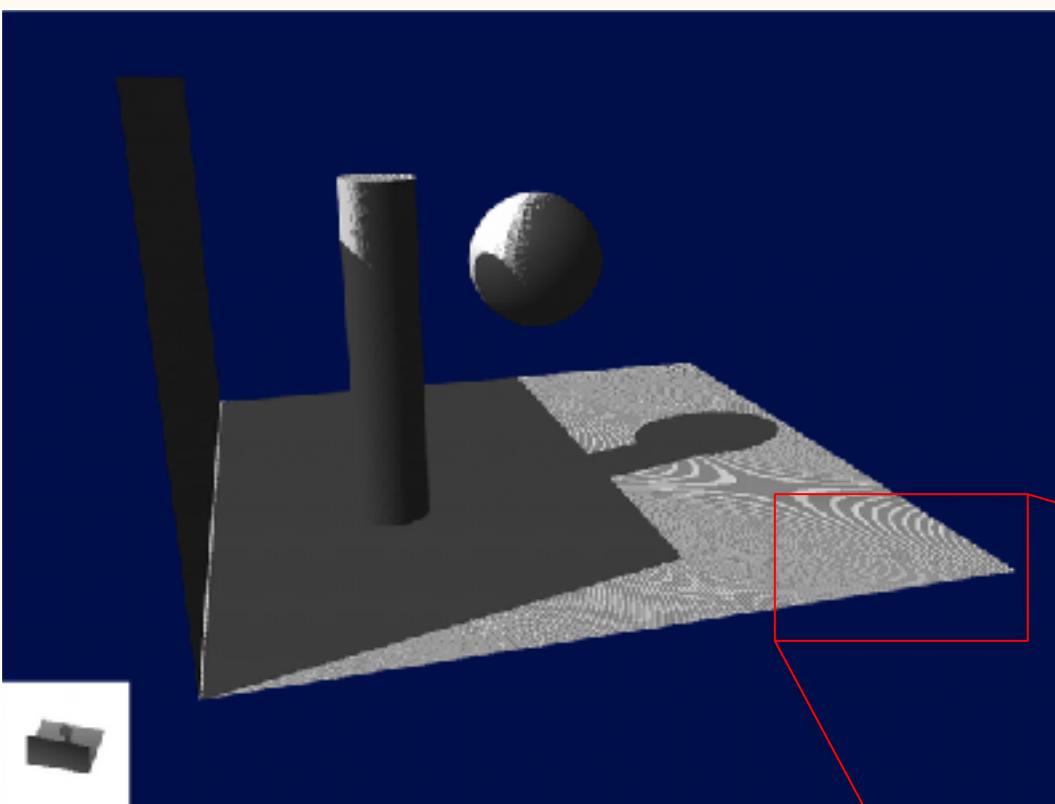
// find the distance parameter to light source
float3 L = -in1.lgtVec;
aToL = abs(dot(L, lgtDir))/lightRange;

// calculate lighting weight according to the shadow map
float weight = (aToL - smRGB.x < bias) ? 1.0: 0.0;
lightIntensity *= weight;
```

Shadow Map Problems – Moire Effect

- Moire effects (摩爾紋)
 - Numerical precision error 造成的
 - 通常發生於 Self-shadowing
 - 加一個 tiny bias 可改善

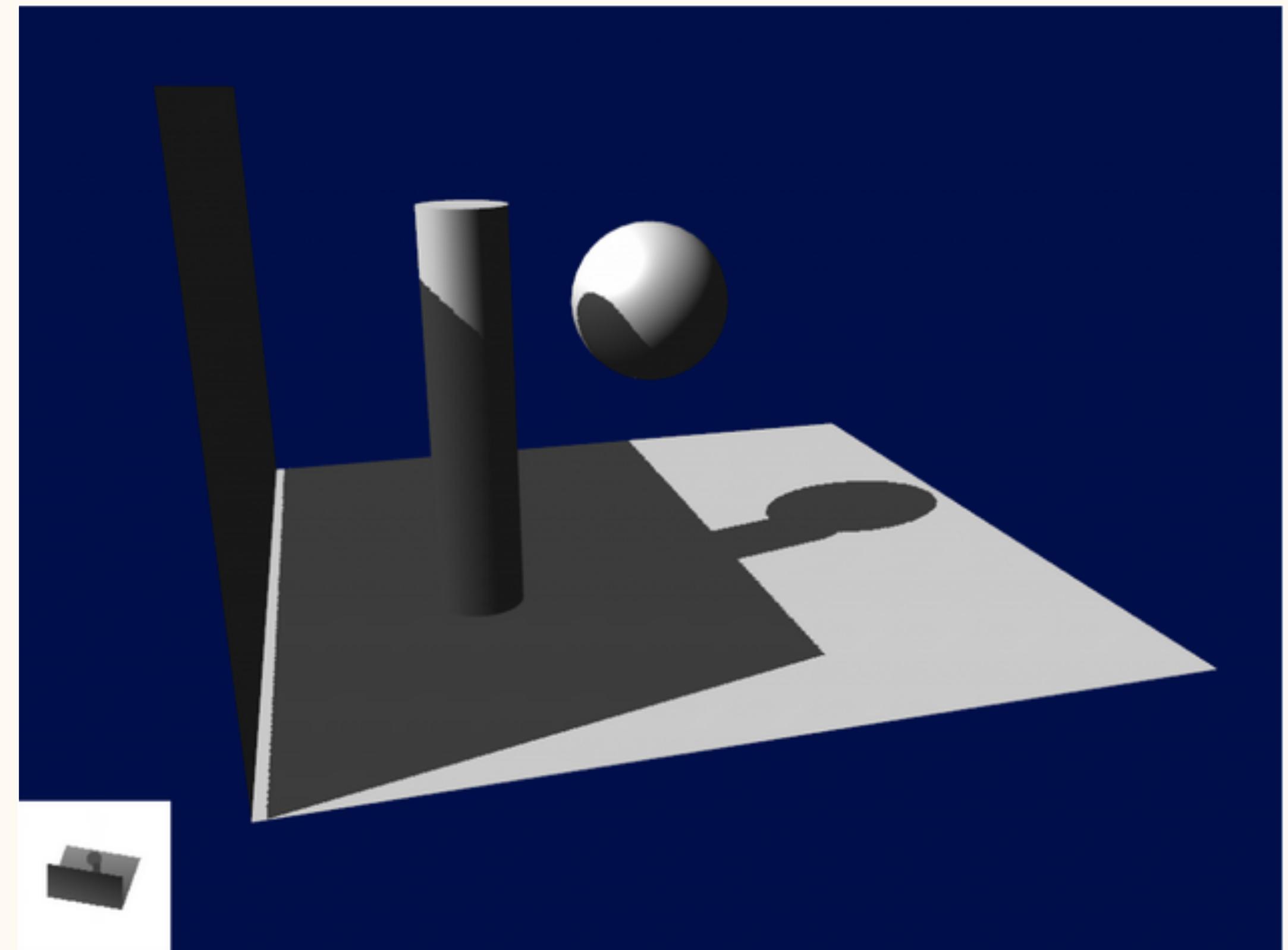
```
float bias = 0.005;  
float visibility = 1.0;  
If (sm.Sample(smSampler, uv).z < depth - bias)  
    visibility = 0.5;
```



Shadow Map Problems – Moire Effect

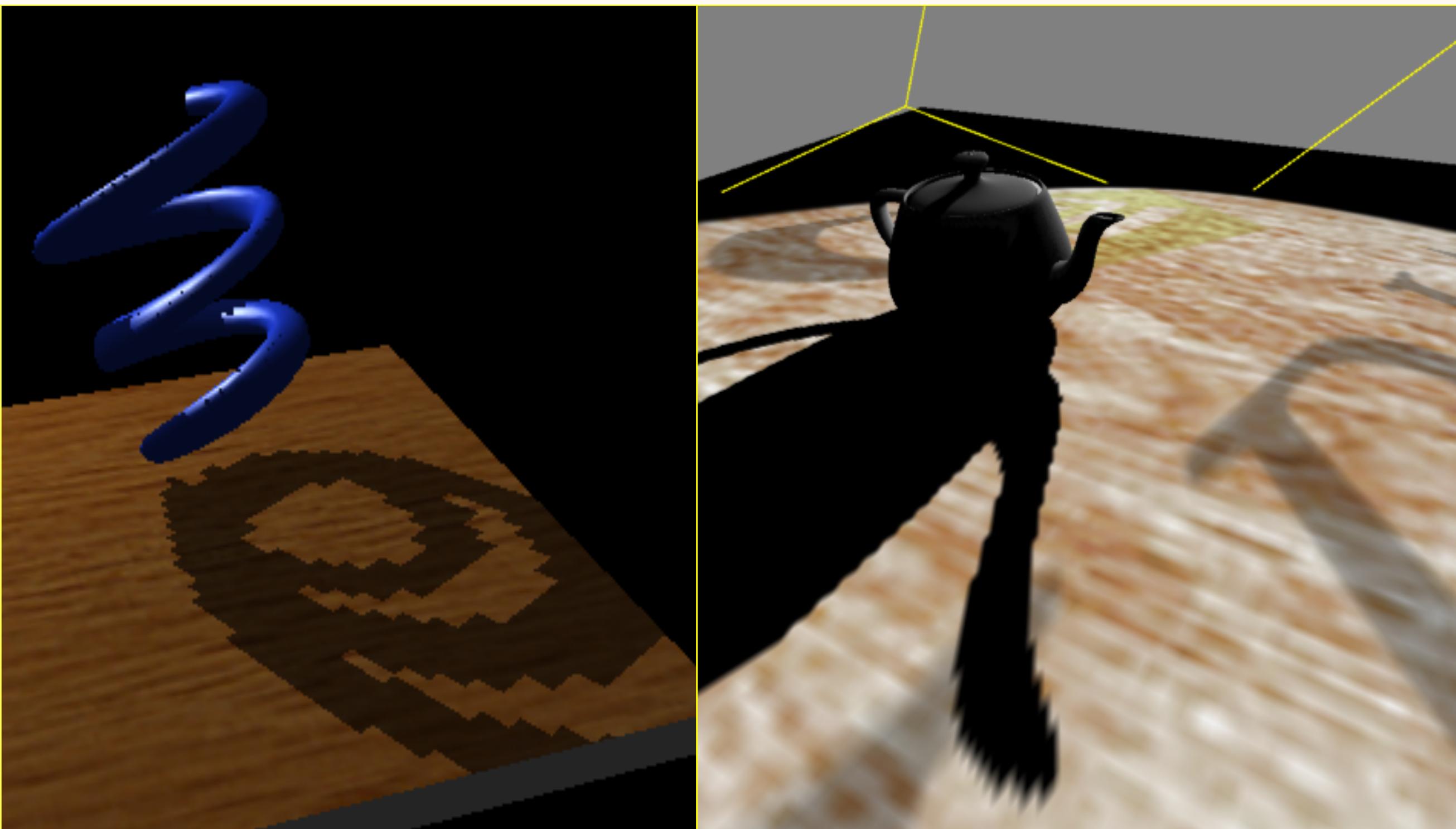
- bias 值可以隨模型曲度和斜率自動調整，通用的作法如下：

```
float cosTheta = dot(N, L);
float bias = 0.005*tan(acos(cosTheta));
bias = clamp(bias, 0.0, 0.01);
```



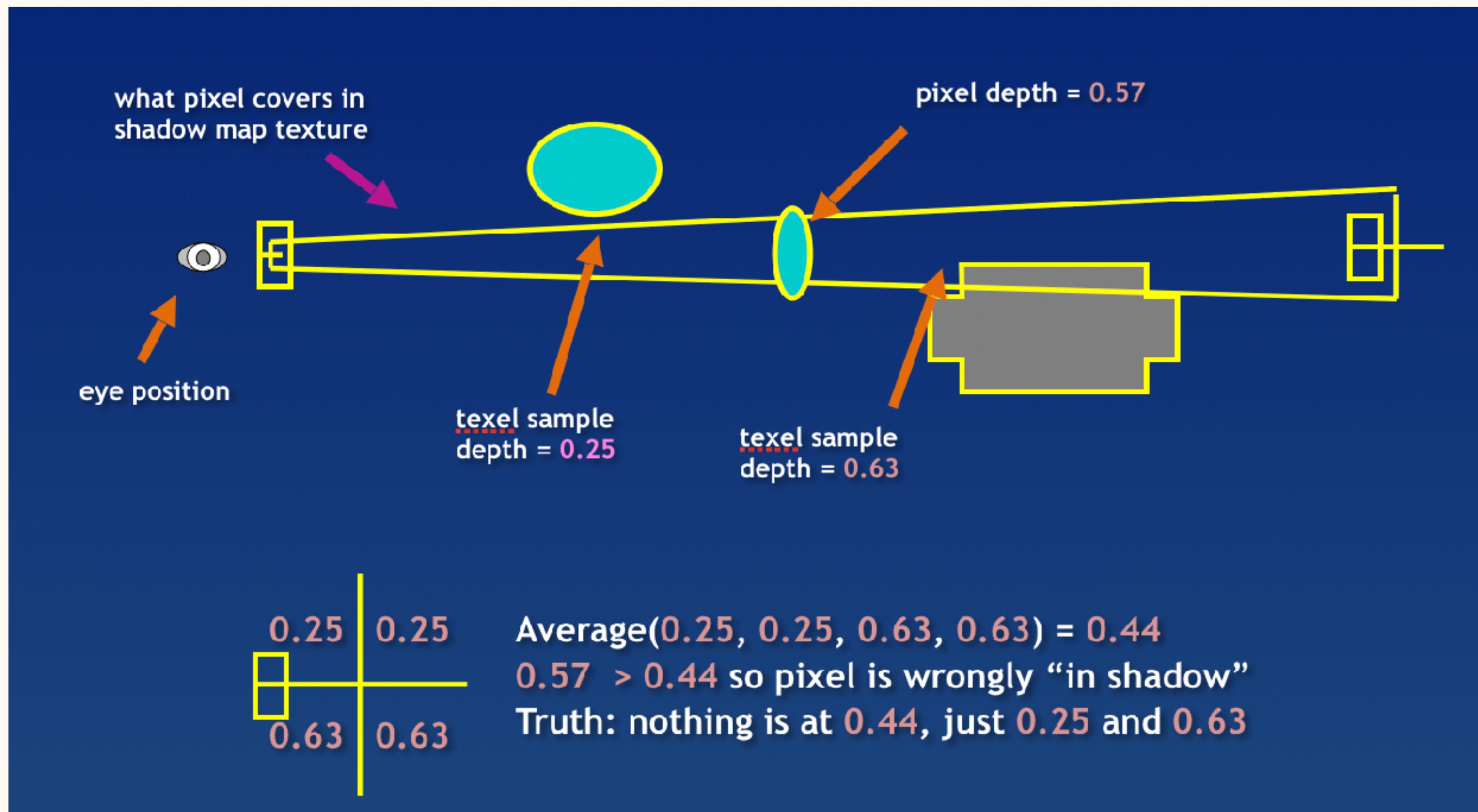
Shadow Map Problems – Aliasing Effect

- 鋸齒問題 (Aliasing)
- 方法：
 - 治標且效果有限：
 - 使用 floating-point texture
 - 增加貼圖解析度
 - 主流作法：
 - Percentage Closer Filter (PCF)



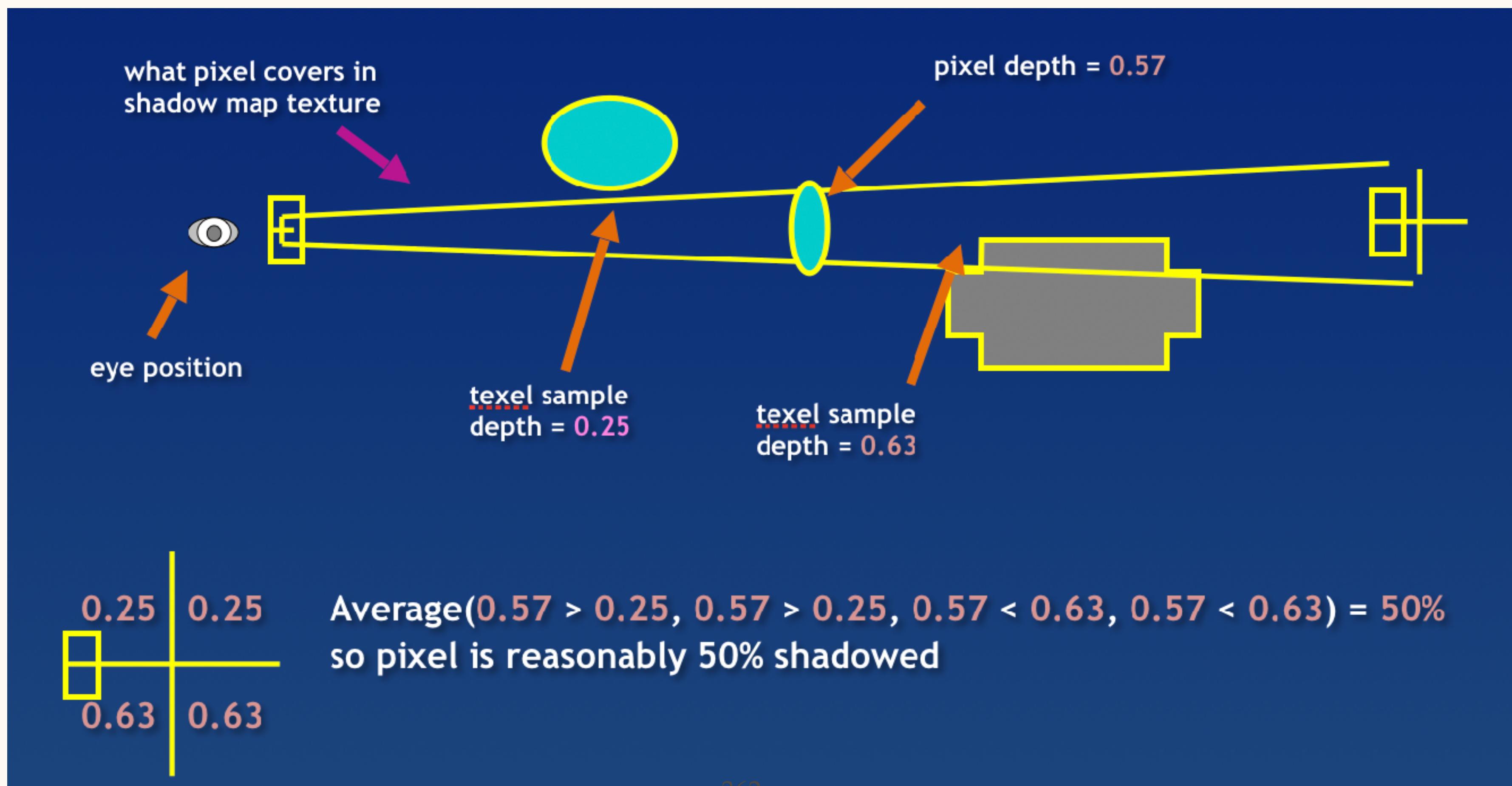
Shadow Map Problems – PCF

- 傳統 average function 無法處理 aliasing 問題



Percentage Closer Filtering

- 解決方法：將比較結果平均 (Percentage Closer Filter)



Percentage Closer Filtering

- 增加 PCF 檢查範圍可以增加影子柔邊效果



1x1



9x9



17x17

- 使用亂數取樣 (random sampling) 可以增加模糊化的效果，但容易產生 noise



Sampler Comparison State

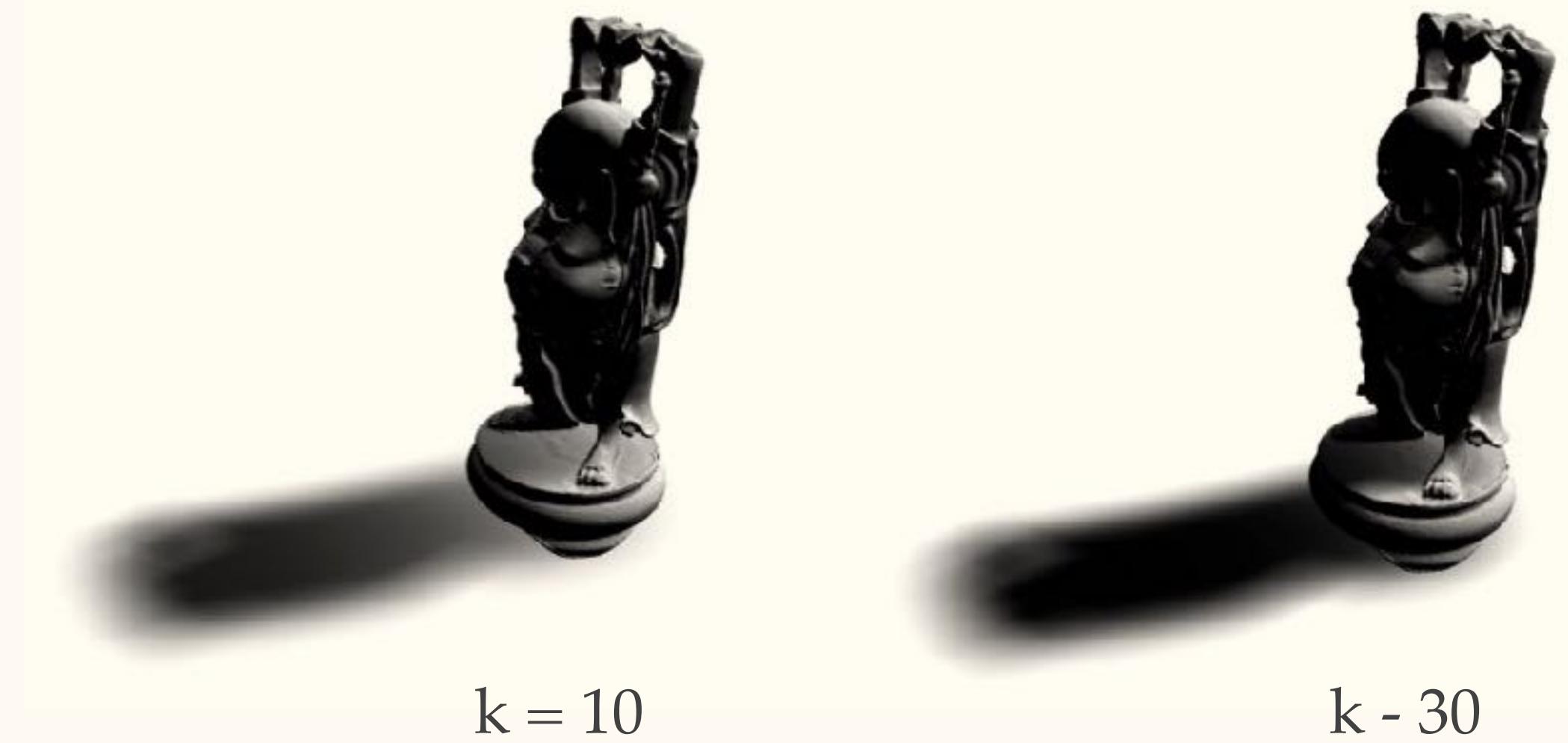
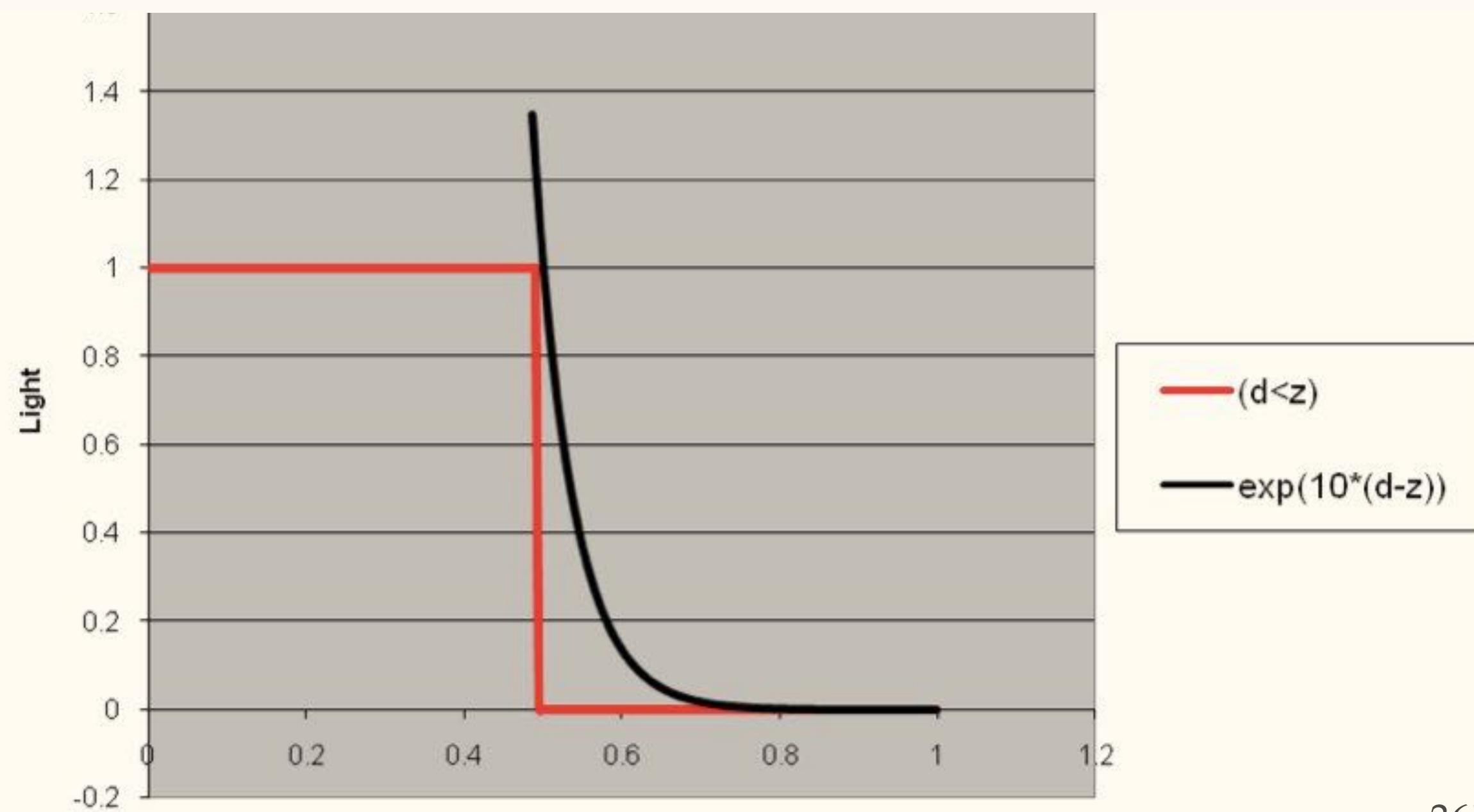
```
SamplerComparisonState ShadowSampler
{
    // sampler state
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
    AddressU = MIRROR;
    AddressV = MIRROR;

    // sampler comparison state
    ComparisonFunc = LESS;
};

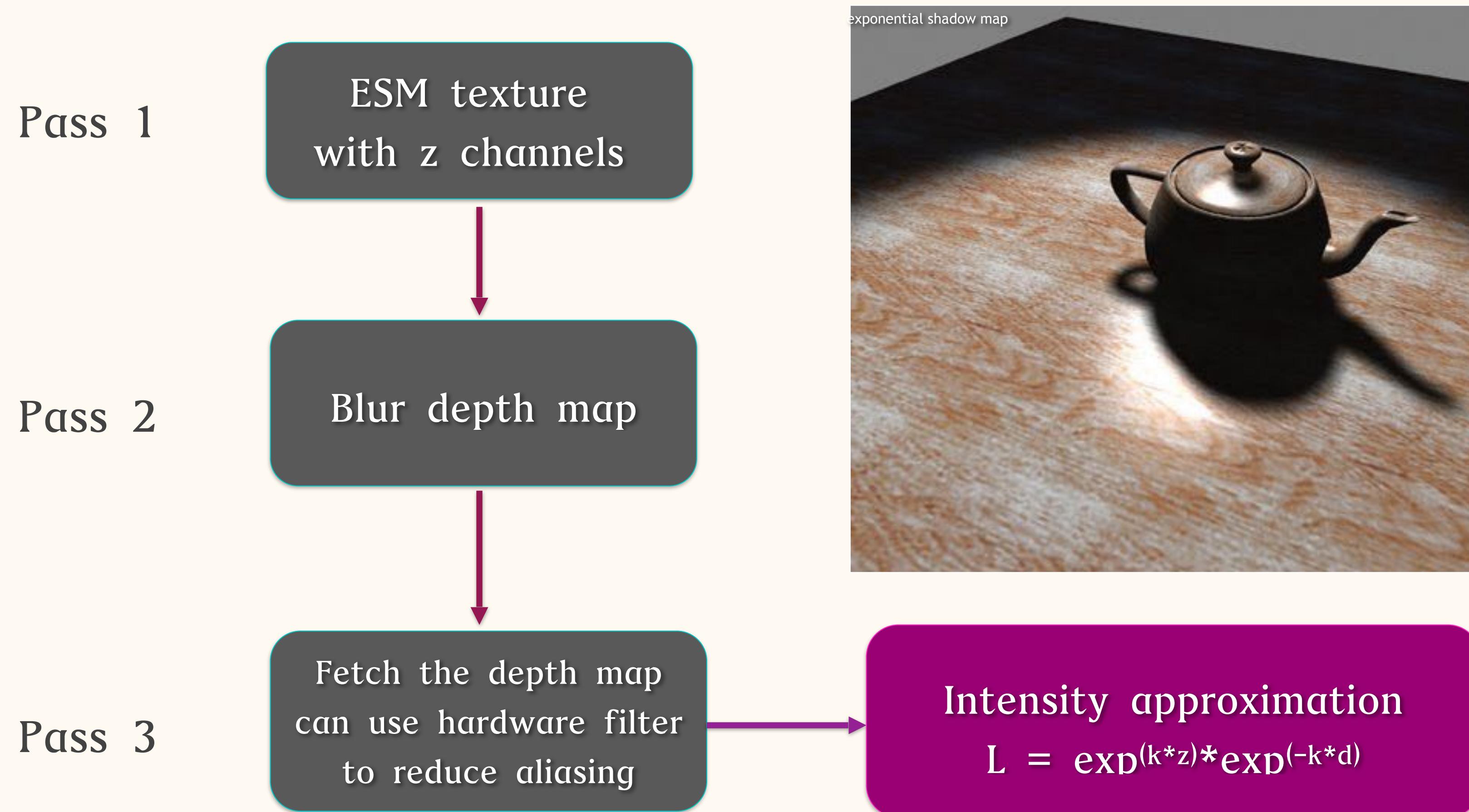
...
float3 vMProjUV = Input.ProjUV.xyz / Input.ProjUV.w;
float fShadow = shadowMap.SampleCmpLevelZero(shadowSampler, vMProjUV.xy, vMProjUV.z);
```

Exponential Shadow Map

- Shader X6 [Salvi08]
- 使用 Approximate step function ($z-d > 0$) with $\exp^{(k*(z-d))} = \exp^{(k*z)} * \exp^{(-k*d)}$
 - k 值決定近似率
 - k 越大，越近似於 step function



Exponential Shadow Map



ESM Shader (Parallel Light)

```
// convert location to screen space (lighting camera)
float4 posLgt = mul(mLightVP, in1.wPos);      // posLgt will be in (-w ~ w)

// homogeneous divide and convert to left-handed texture space (0 ~ 1)
float2 lgtTexCoord = (posLgt.xy/posLgt.w)*float2(0.5f, -0.5f) + 0.5f;
float4 smRGB = shadowMap.Sample(shadowMapSampler, lgtTexCoord);

// find the distance parameter to light source
float3 L = -in1.lgtVec;
aToL = abs(dot(L, lgtDir))/lightRange;

// calculate lighting weight according to the shadow map
float weight = (aToL - smRGB.x < bias) ? 1.0: saturate(exp(-
esmK*aToL)*exp(esmK*smRGB.x));
lightIntensity *= weight;
```

環境光照與影子

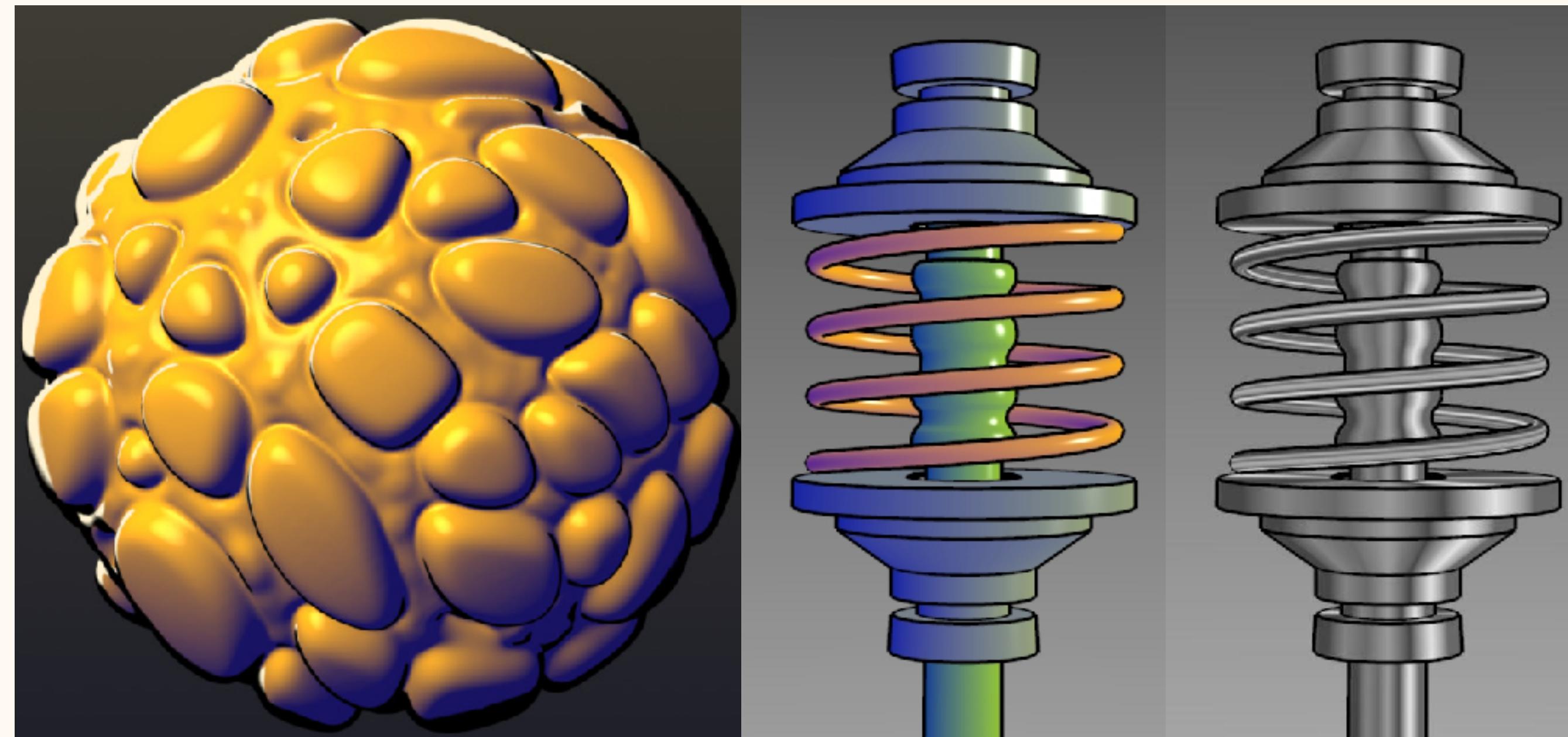
- Lighting 是加法運算 (“add”)
 - 錯誤的做法 : Lightmap / Ambient Occlusion map
- 環境光照
 - Ambient = lighting from environment
 - Ambient = no lighting area from direct light sources
 - Occluded by geometry itself
- 影子
 - Shadow = no lighting area from direct light source
 - Occluded by the other geometry
- 整合的解法 :
 - Global illumination renderers (全域光照)

Non-photorealistic Rendering

非擬真算圖

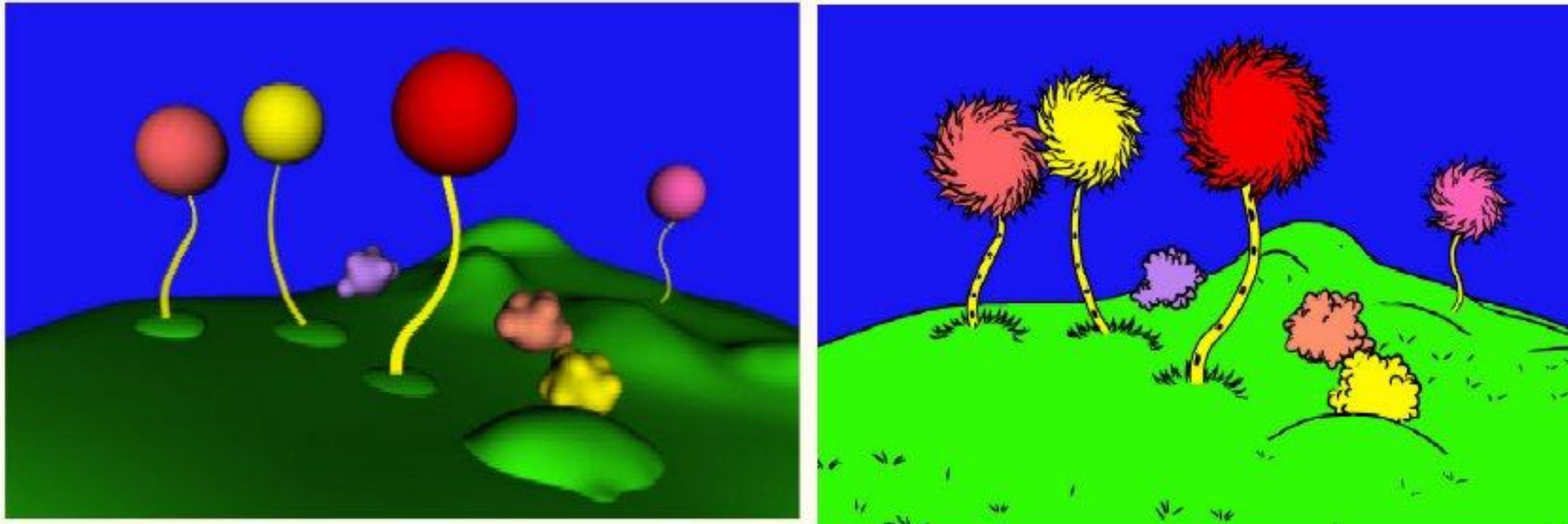
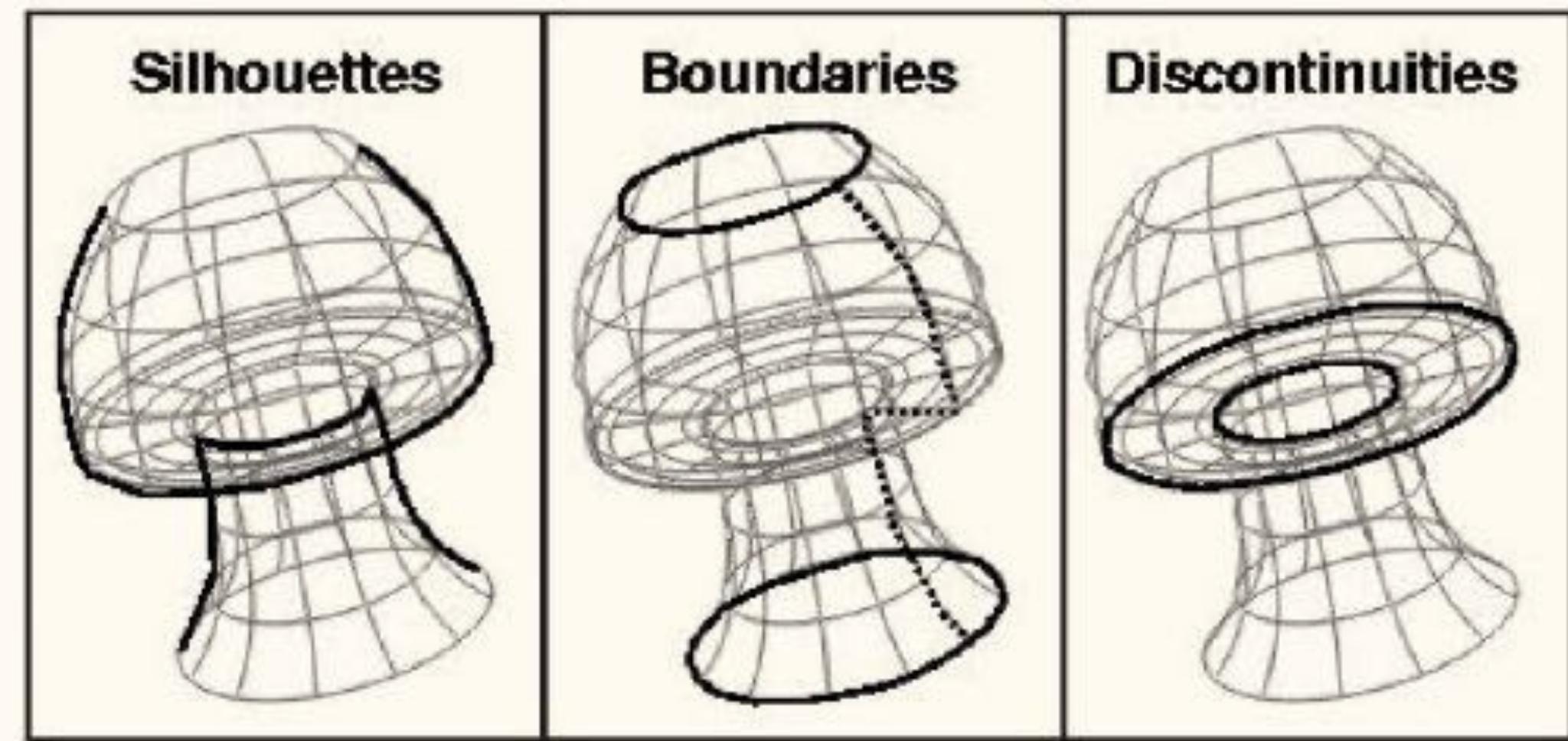
Non-photorealistic Rendering

- Techniques for rendering that don't strive for realism, but style, expressiveness, abstraction, etc
- Better terms :
 - Stylized rendering, Artistic rendering, Abstract rendering

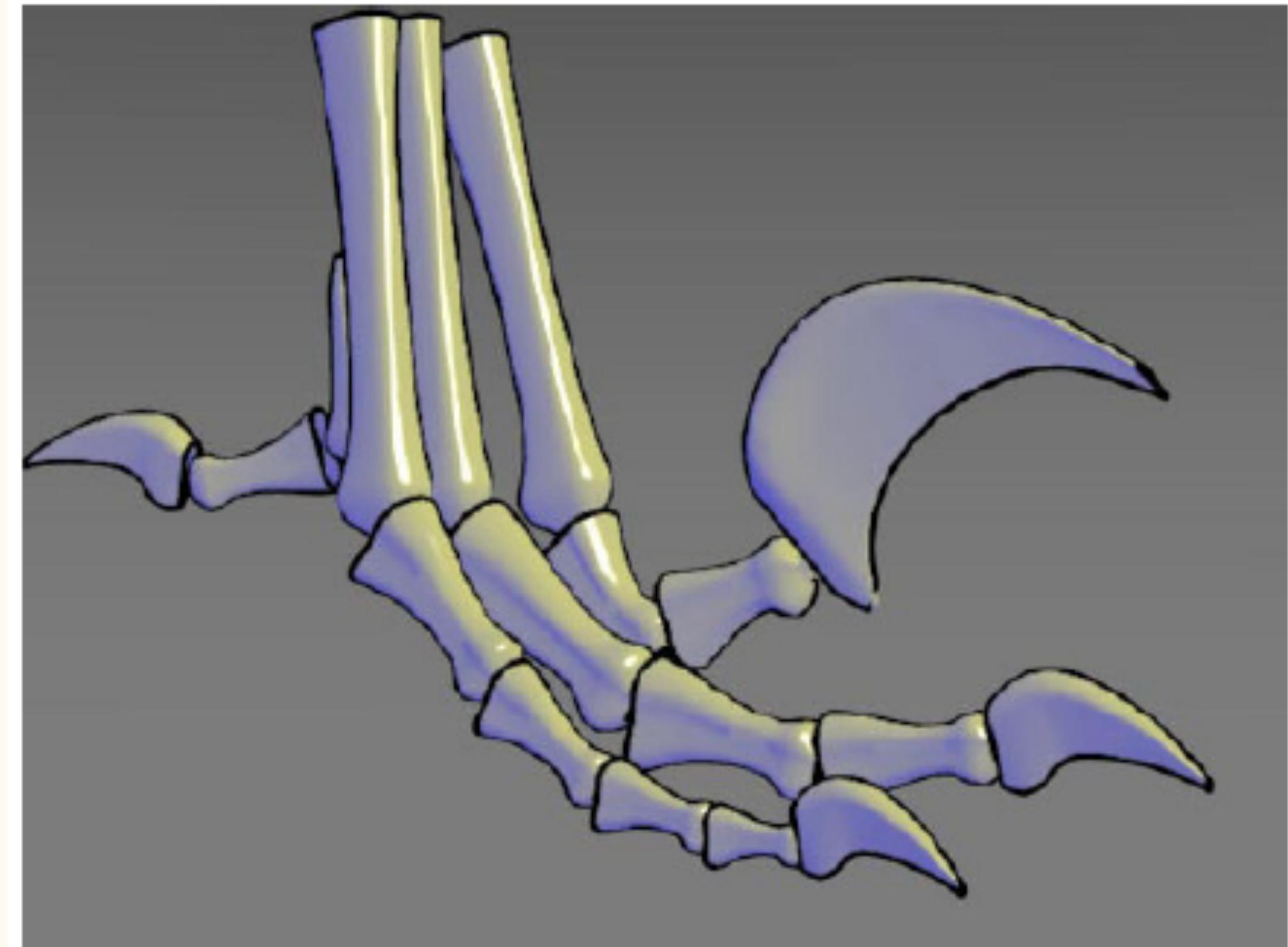
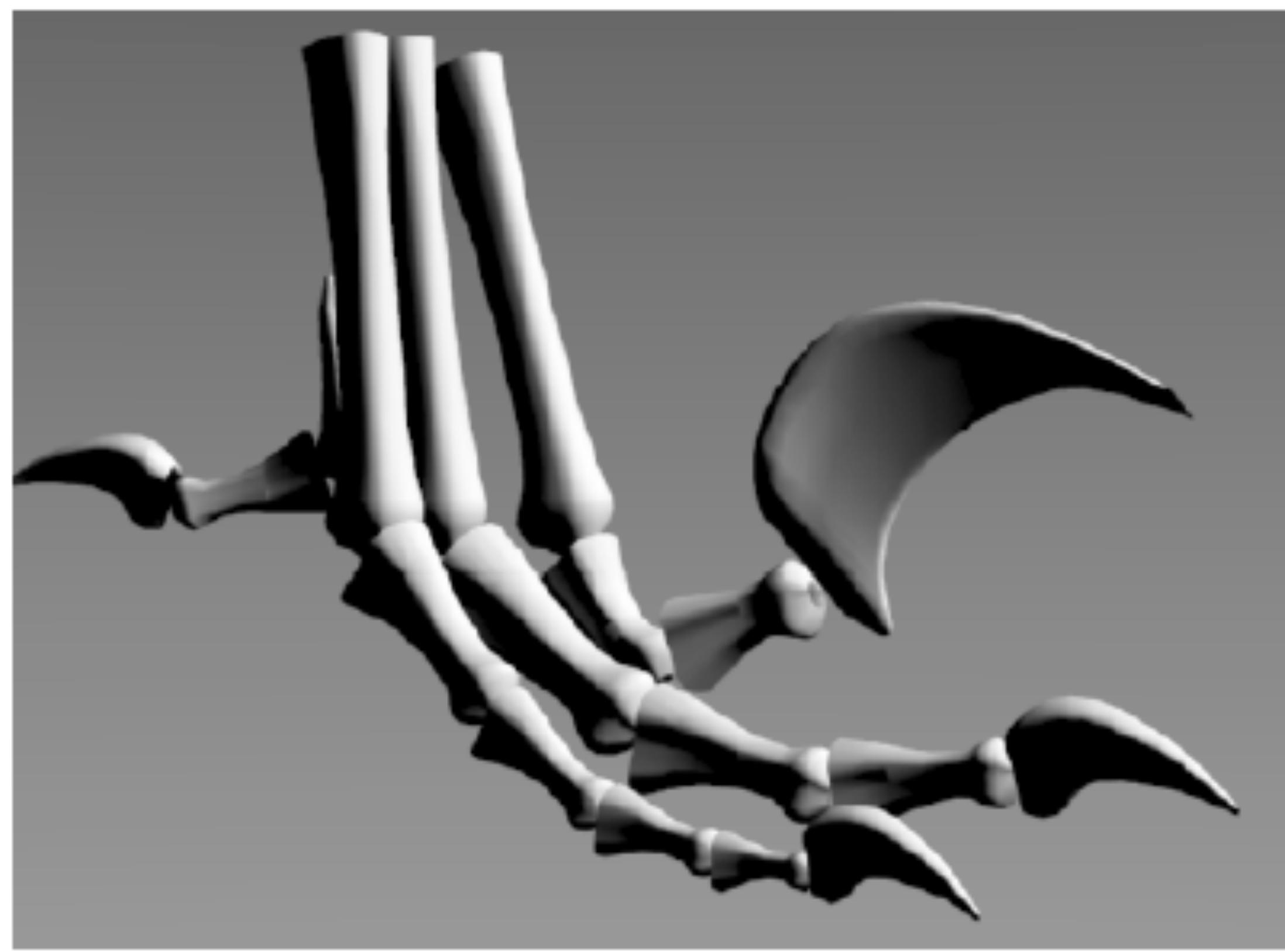


Shape Abstraction by Lines

- Boundary lines
- Silhouette lines
- Creases
- Material edges
- Various line styles can be used
- Graftals



Shape Abstraction by Shading

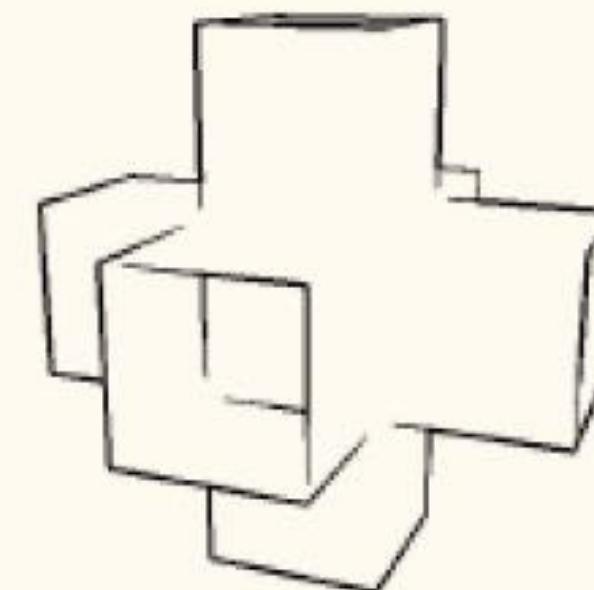
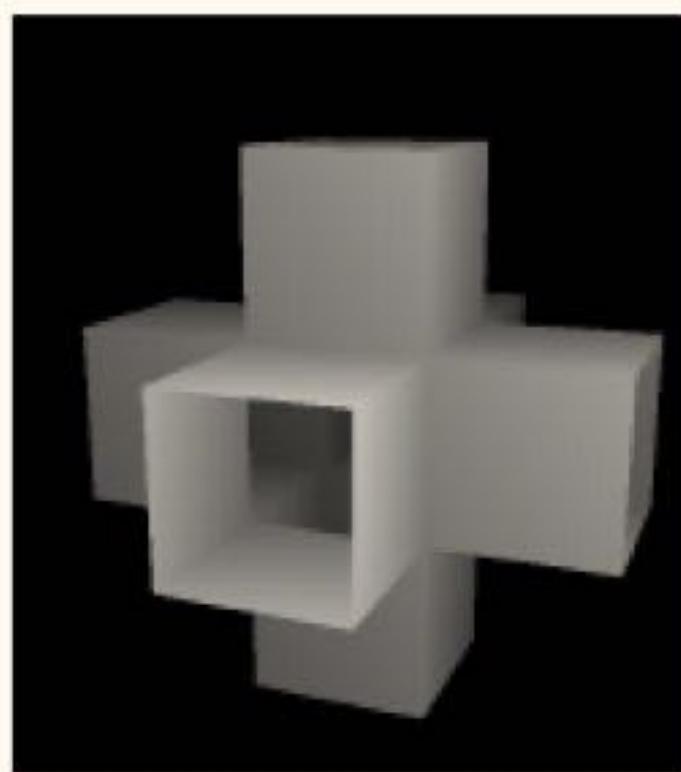


Shape Abstraction by Textures

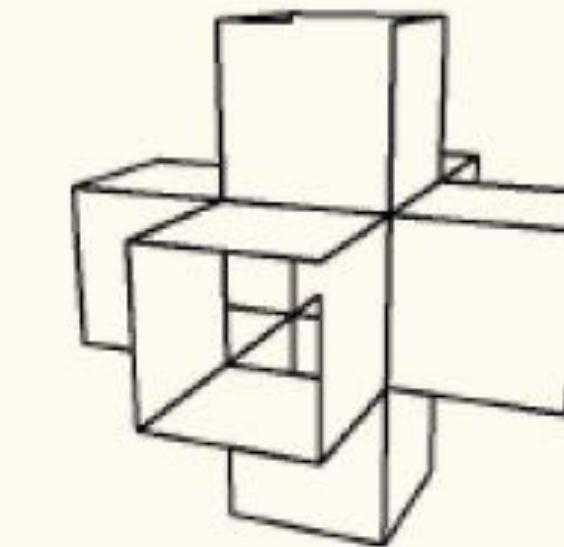
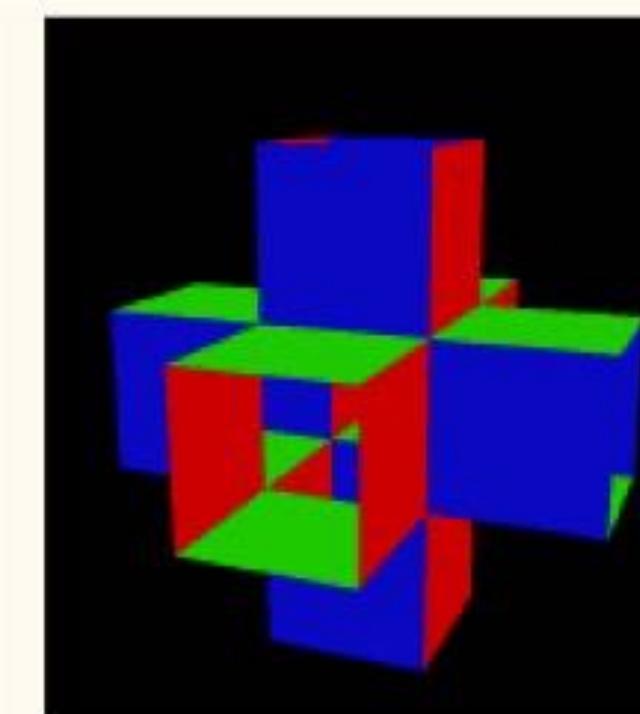
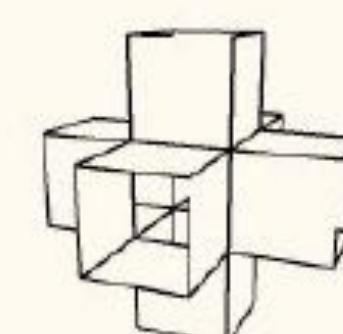
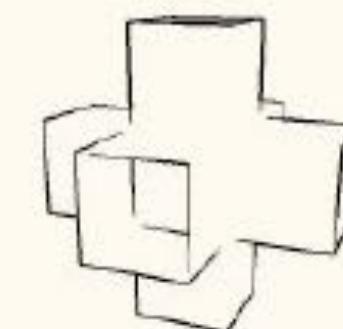
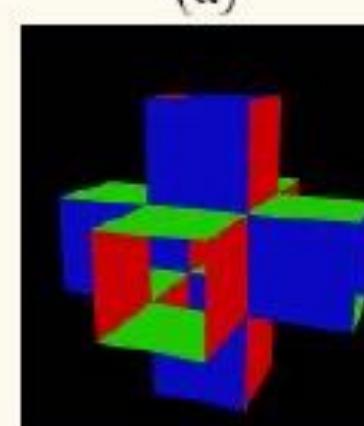
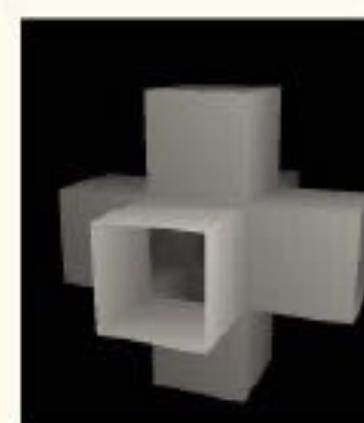


Feature Line by Edge Detection

- Analyze the depth buffer



- Combine together :



- Analyze the normal map

Edge Detector

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

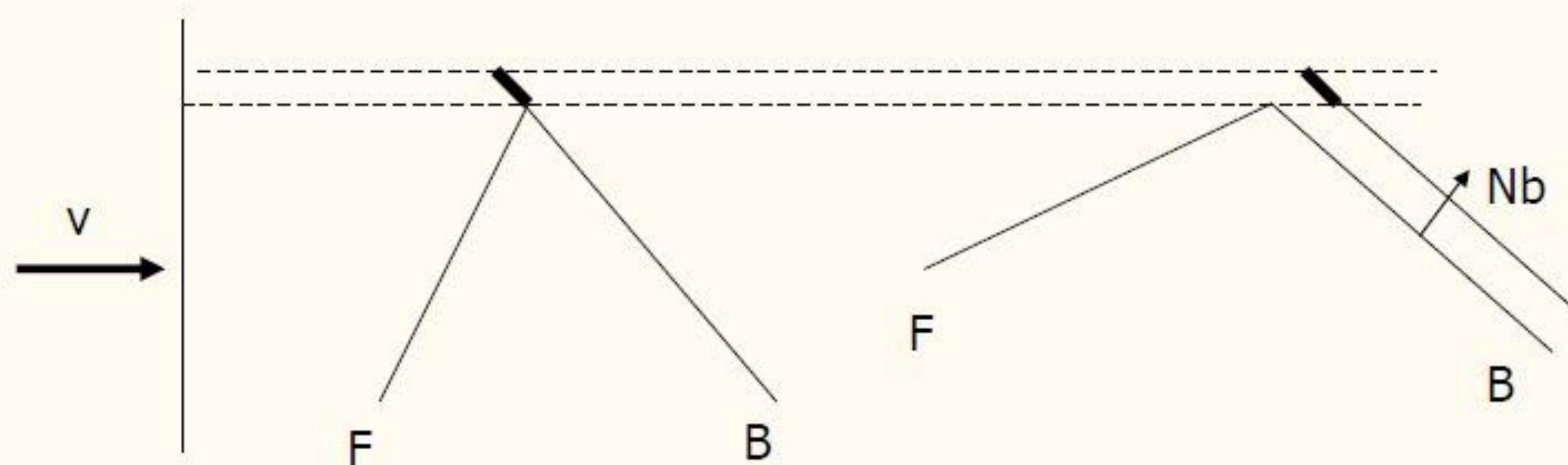
$$Ix(x,y) = I(x,y) \otimes Sx; \quad Iy(x,y) = I(x,y) \otimes Sy$$

$$IM = \sqrt{ (Ix(x,y)^2 + Iy(x,y)^2) }$$

Get edge by thresholding IM

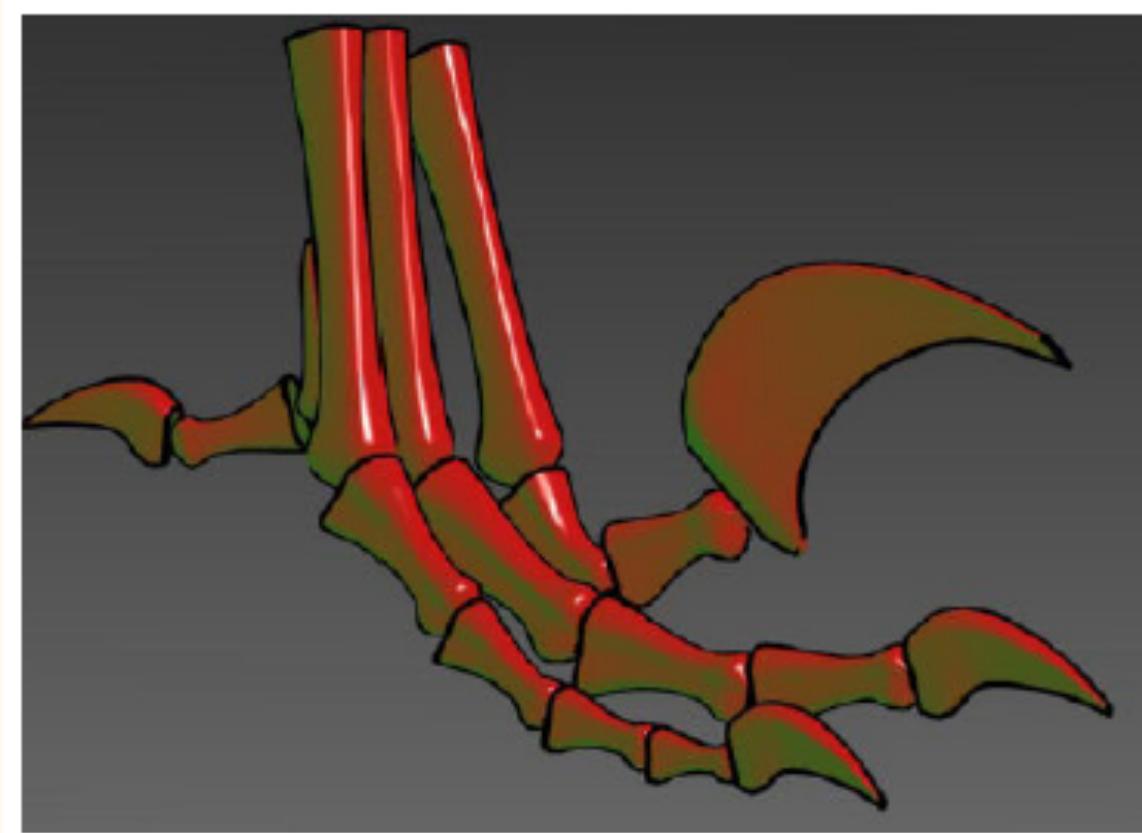
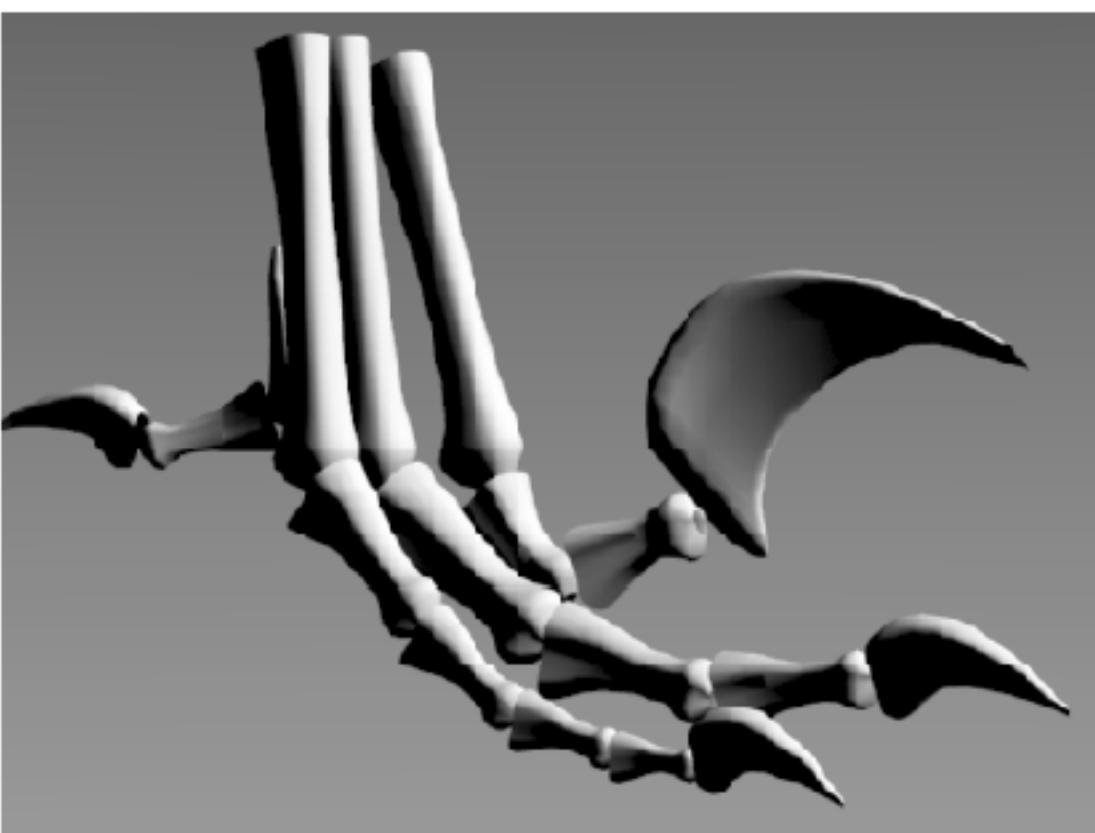
Feature Lines by Geometry

- Raskar and Cohen's solution
 - Render back faces larger in silhouette color
 - Extrude the vertex in vertex normal direction
 - Render front faces normally
- This can be achieved by two-pass rendering shader.



Tone Shading

- Phong shading model is not always satisfactory in NPR.
- Problems are in regions where $\text{dot}(N, L) < 0$
 - Only constant ambient color are seen
 - Difficult to deduce shapes
- Tone shading goals
 - Use a compressed dynamic range for shading
 - Use color visually distinct from black and white



Tone Shading – Undertone

- To further differentiate different surface orientations, we can use cool to warm color undertones
- Cool colors
 - Blue, violet, green
- Warm colors
 - Red, orange, yellow

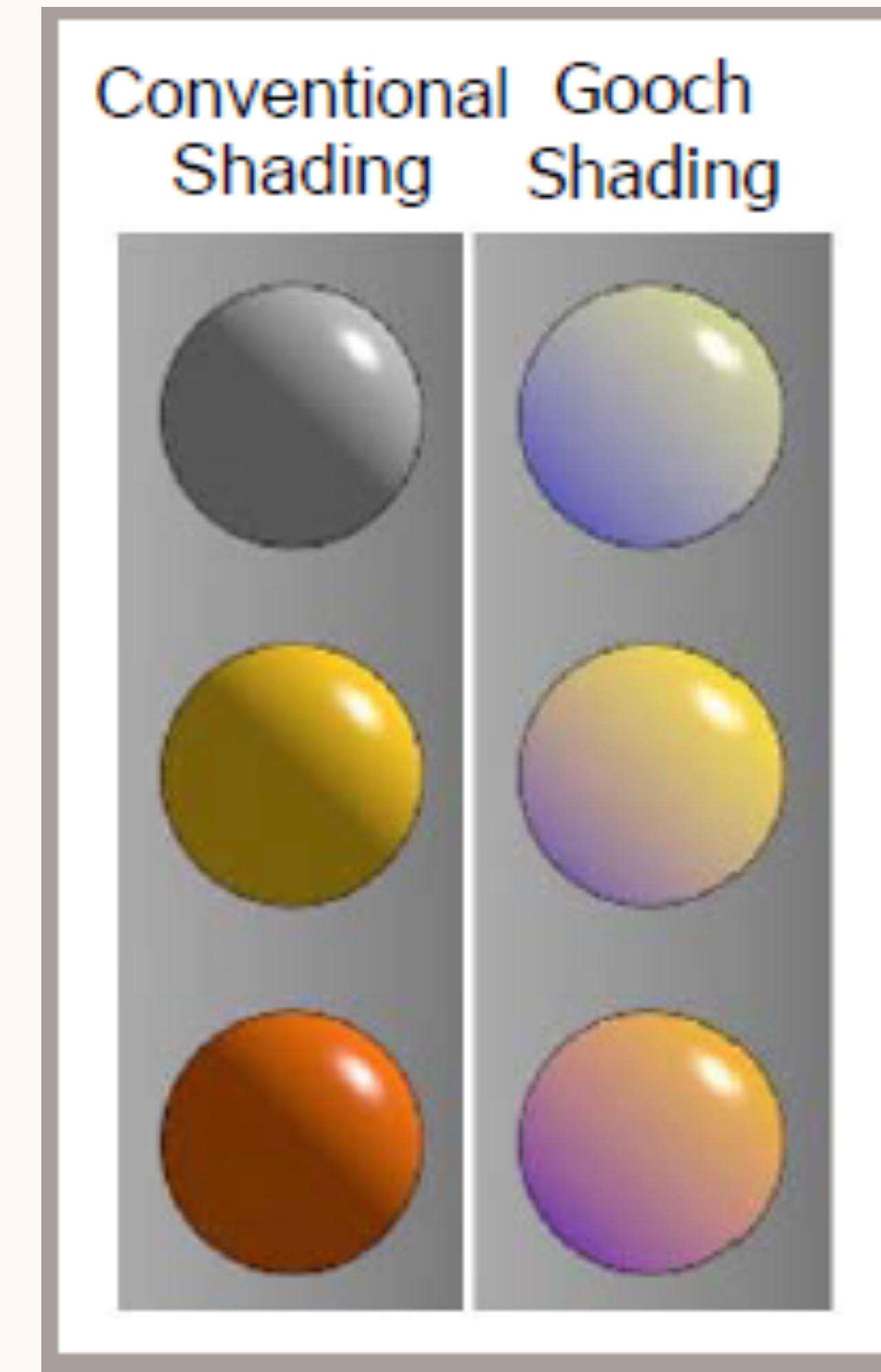
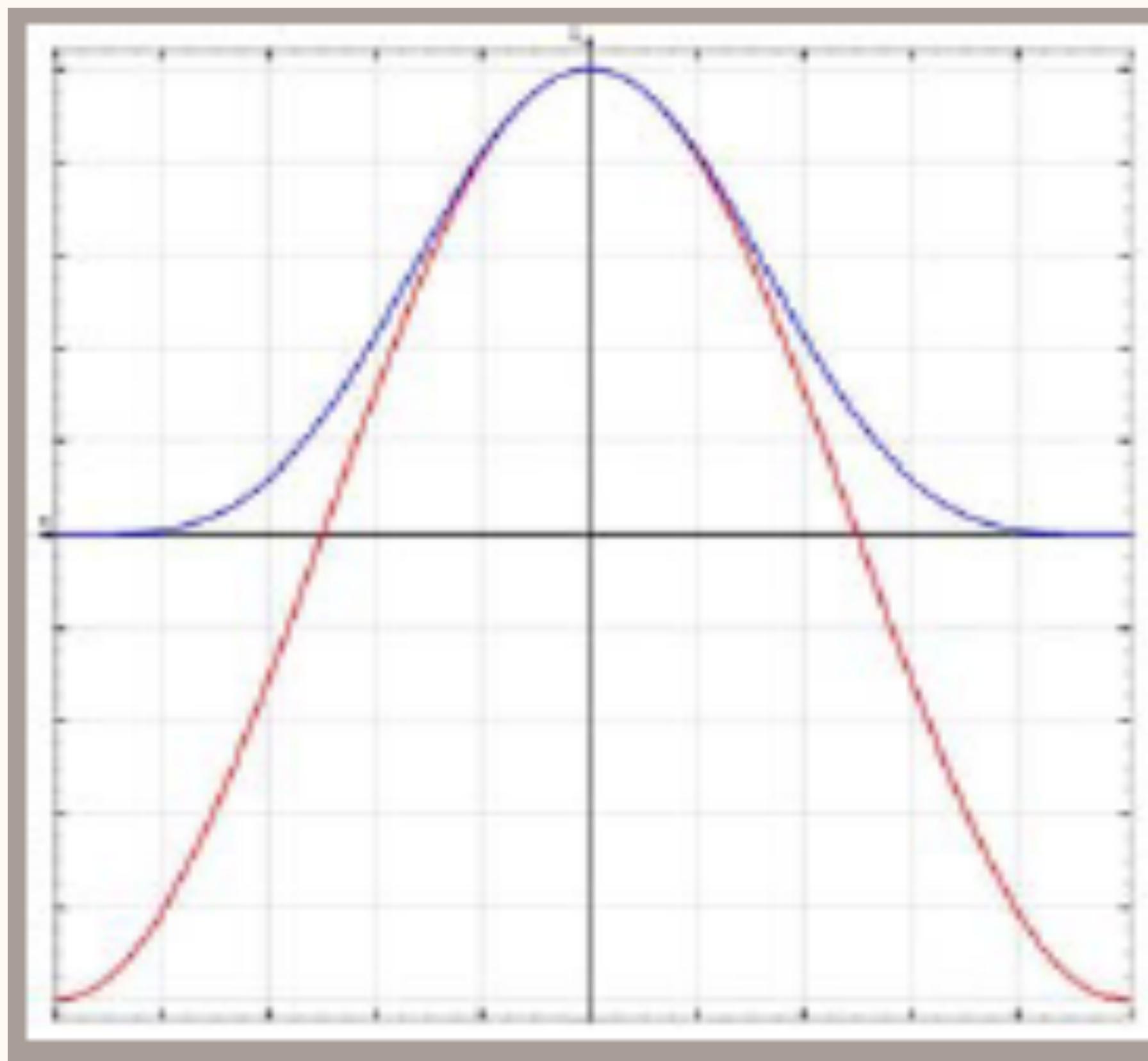


warm

cool

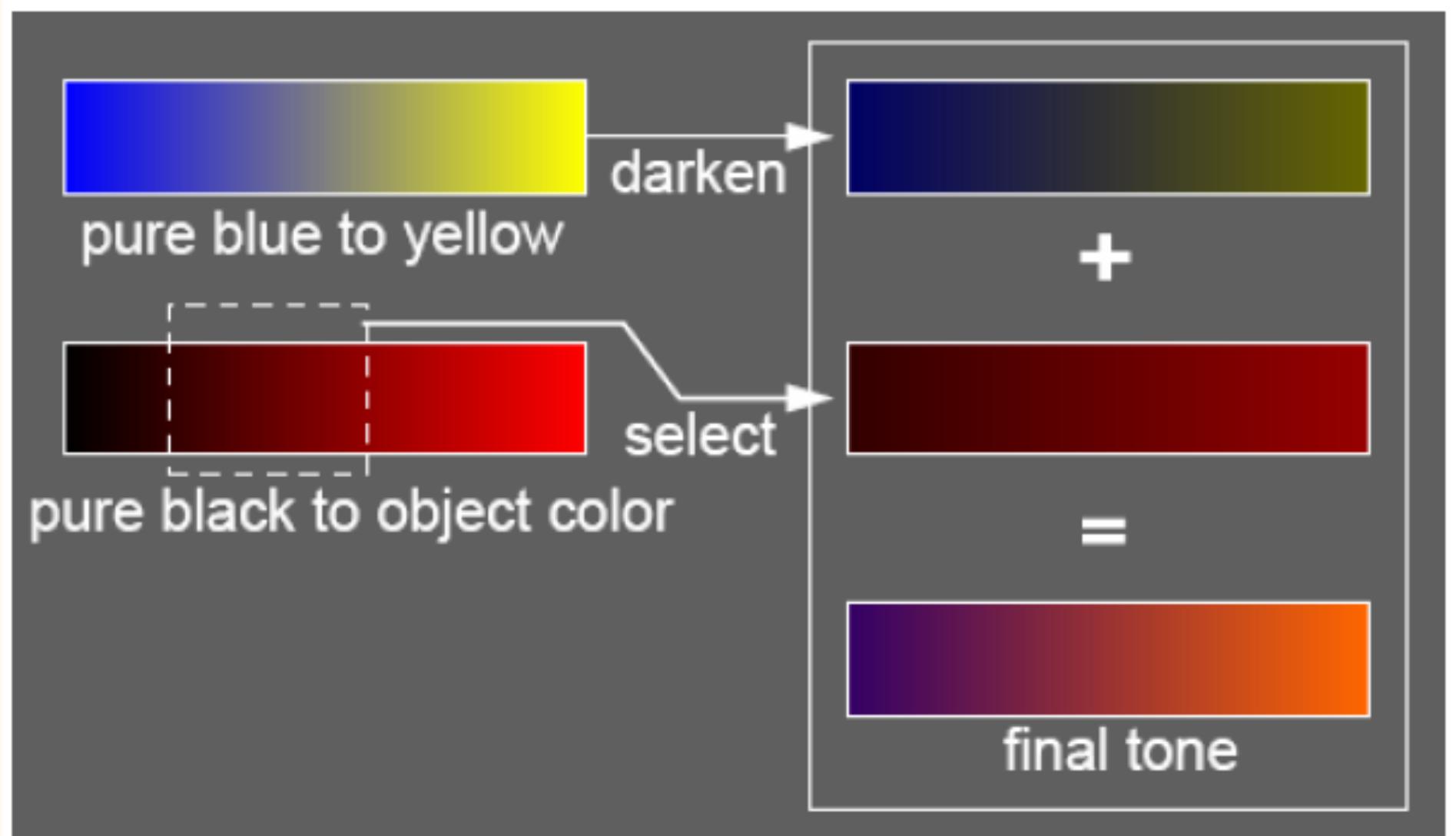
Tone Shading – Half Lambert

- Half Lambert
 - $I = \text{dot}(L, N)/2 + 1/2$
- Gooch Shading



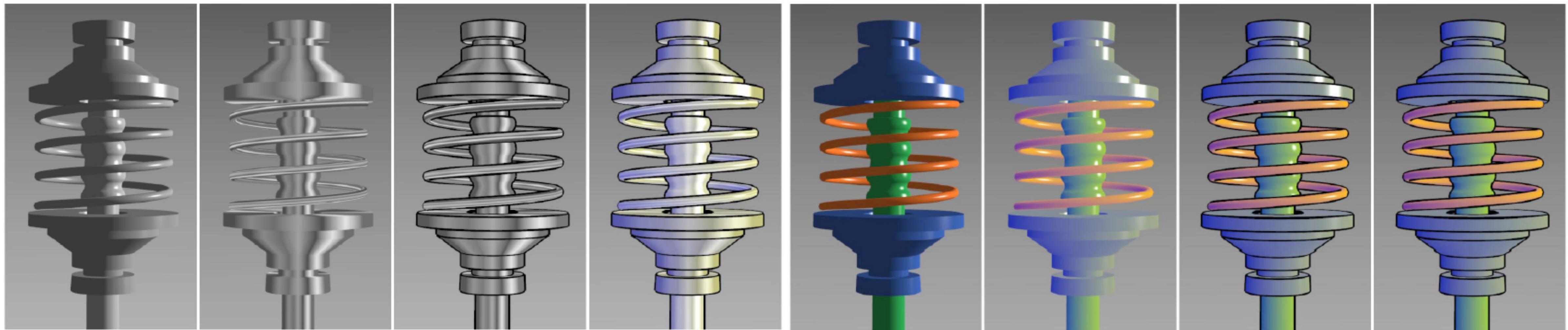
Tone Shading - Blend tone and Undertone

- Add warm-to-cool undertone to a red object



- $I = (1/2 + \text{dot}(L,N)/2)K_{\text{warm}} + (1 - (1/2 + \text{dot}(L,N)/2))K_{\text{cool}}$
 - $K_{\text{cool}} = K_{\text{blue}} + \alpha K_d$, $K_{\text{blue}} = (0, 0, b)$, b in $[0, 1]$
 - $K_{\text{warm}} = K_{\text{yellow}} + \beta K_d$, $K_{\text{yellow}} = (r, r, 0)$, r in $[0, 1]$
 - α and β are user-specified parameter

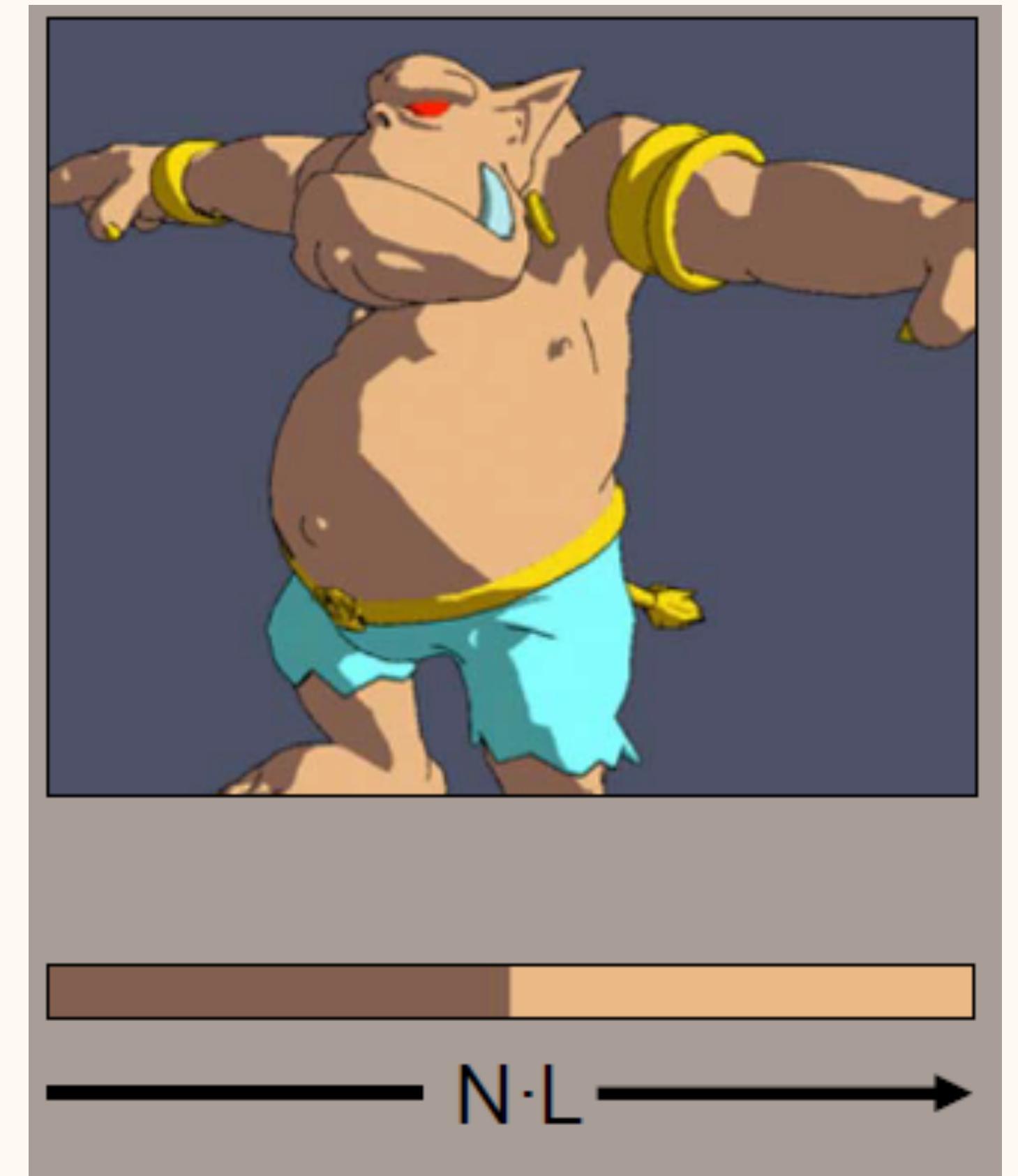
Tone Shading - Gooch Shading



Gooch 98

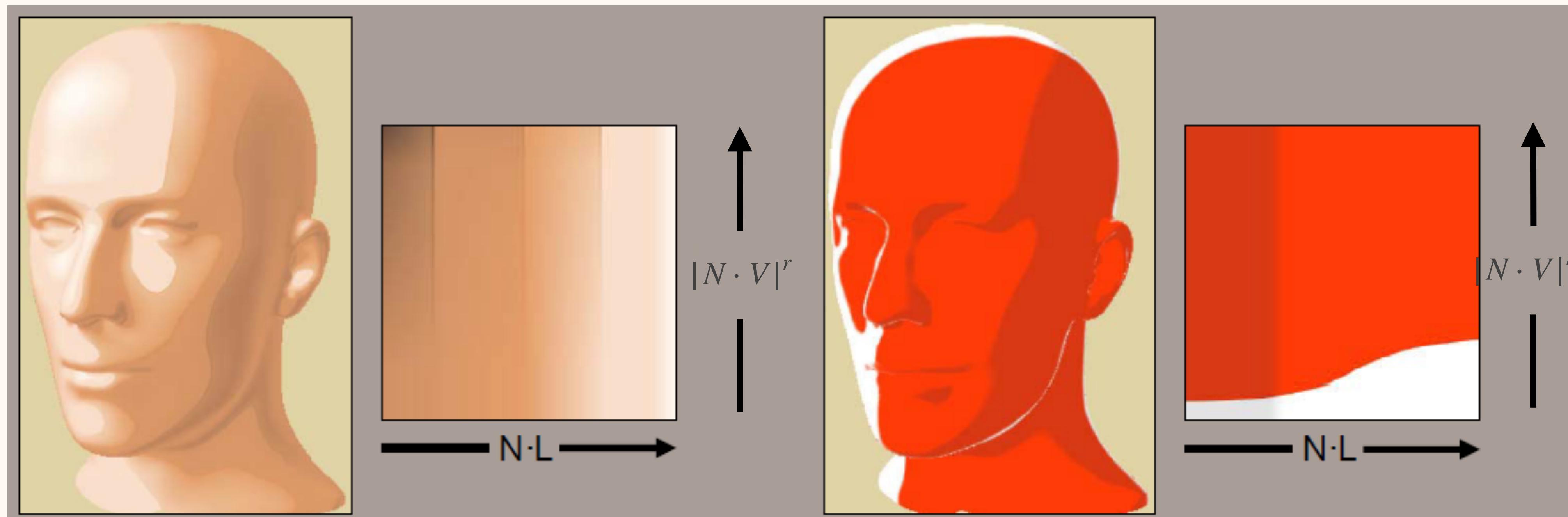
Shading As the Lookup Table

- Lake, 2000
 - Lake used a 1D texture lookup based upon the Lambertian term to simulate the limited color palette cartoonists use for painting cels
- Cartoon Shading



Shading As the Lookup Table

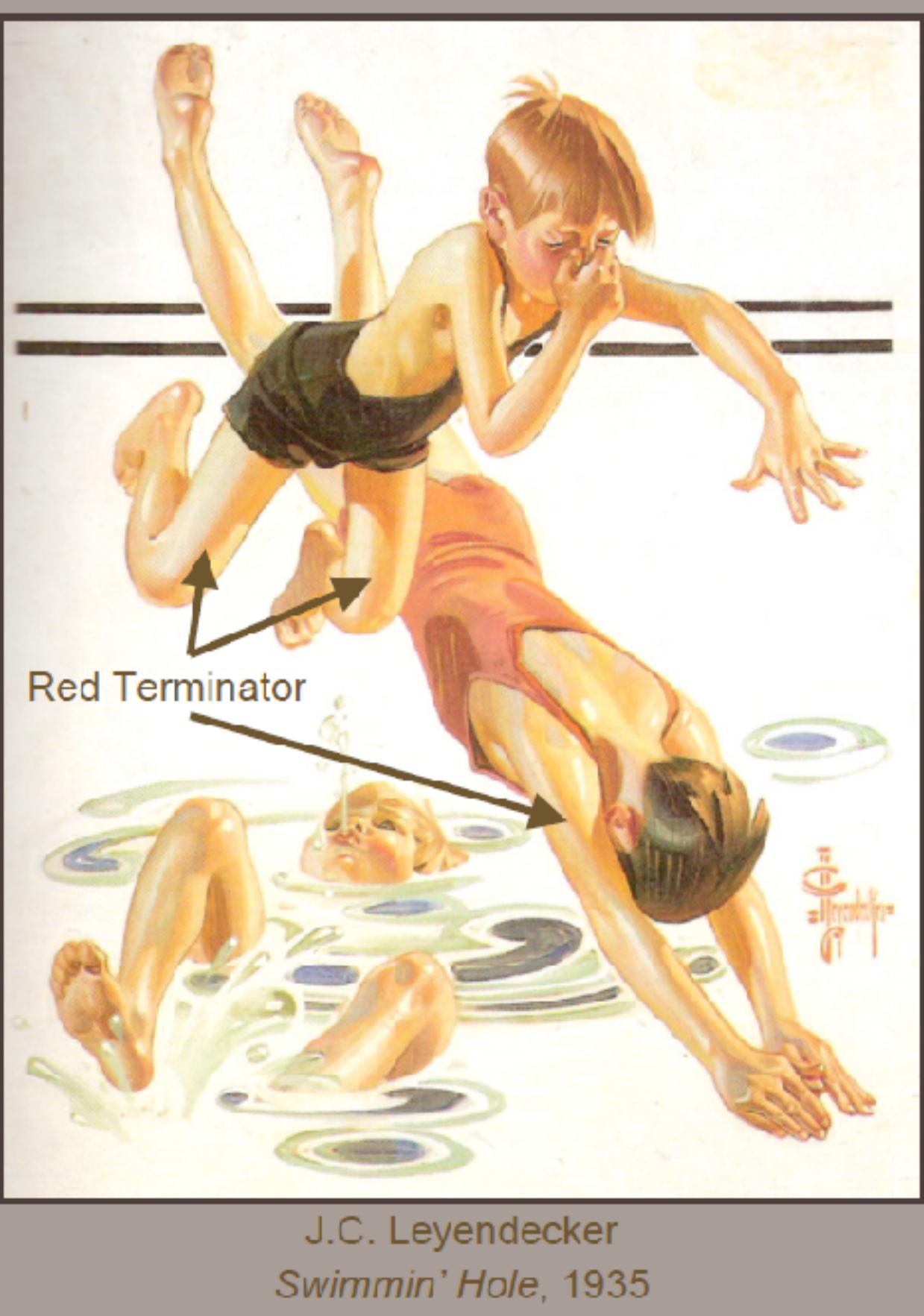
- Barla, 2006
 - Extended the same idea to 2D texture lookup
 - Fresnel-like creates a hard “virtual backlight”
 - A rim-lighting effect



Team Fortress 2 - A Case Study



- Implemented with Source Engine by Valve in 2008
- Inspired by J. C. Leyendecker
 - Rim highlights, red terminator, clothing folds, warm-to-cool hue shift



Team Fortress 2 - Character Lighting

VIEW INDEPENDENT

$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w \left(\left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right) \right] +$$

$$\sum_{i=1}^L \left[c_i k_s \max \left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

VIEW-DEPENDENT



VIEW INDEPENDENT TERMS

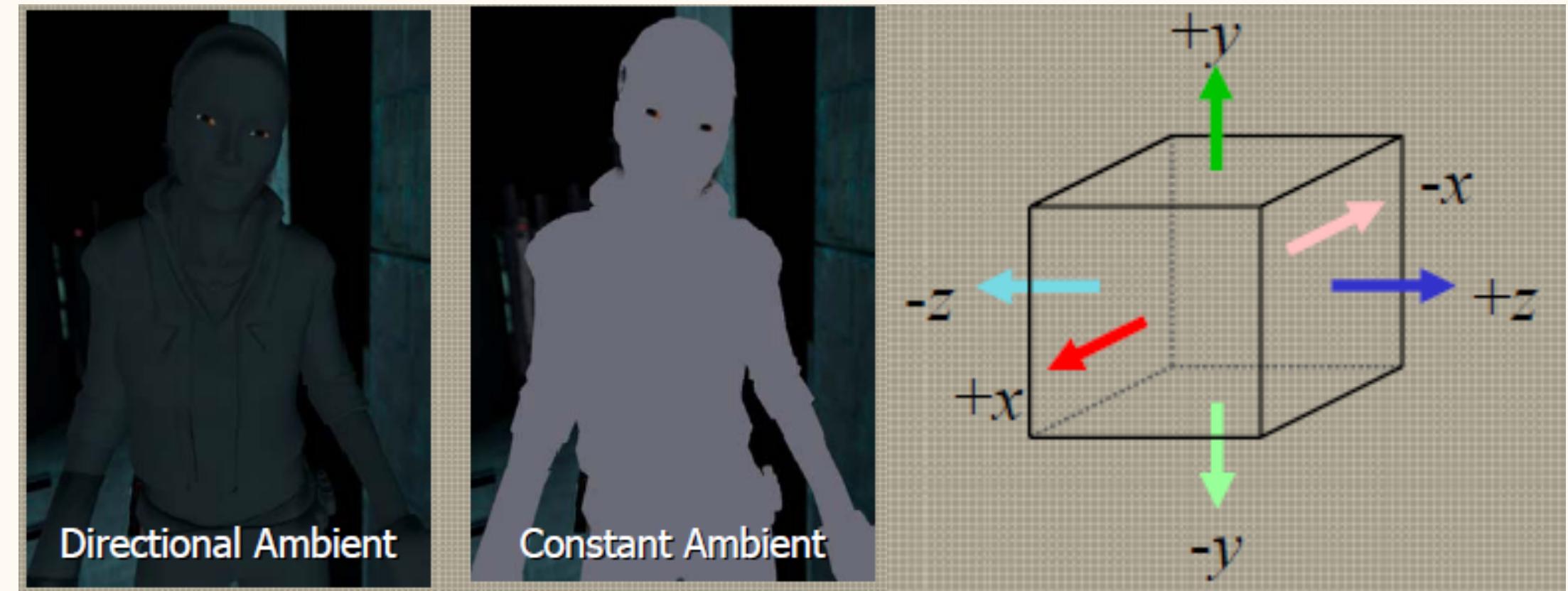
$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w \left(\left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right) \right]$$

- Spatially-varying directional ambient

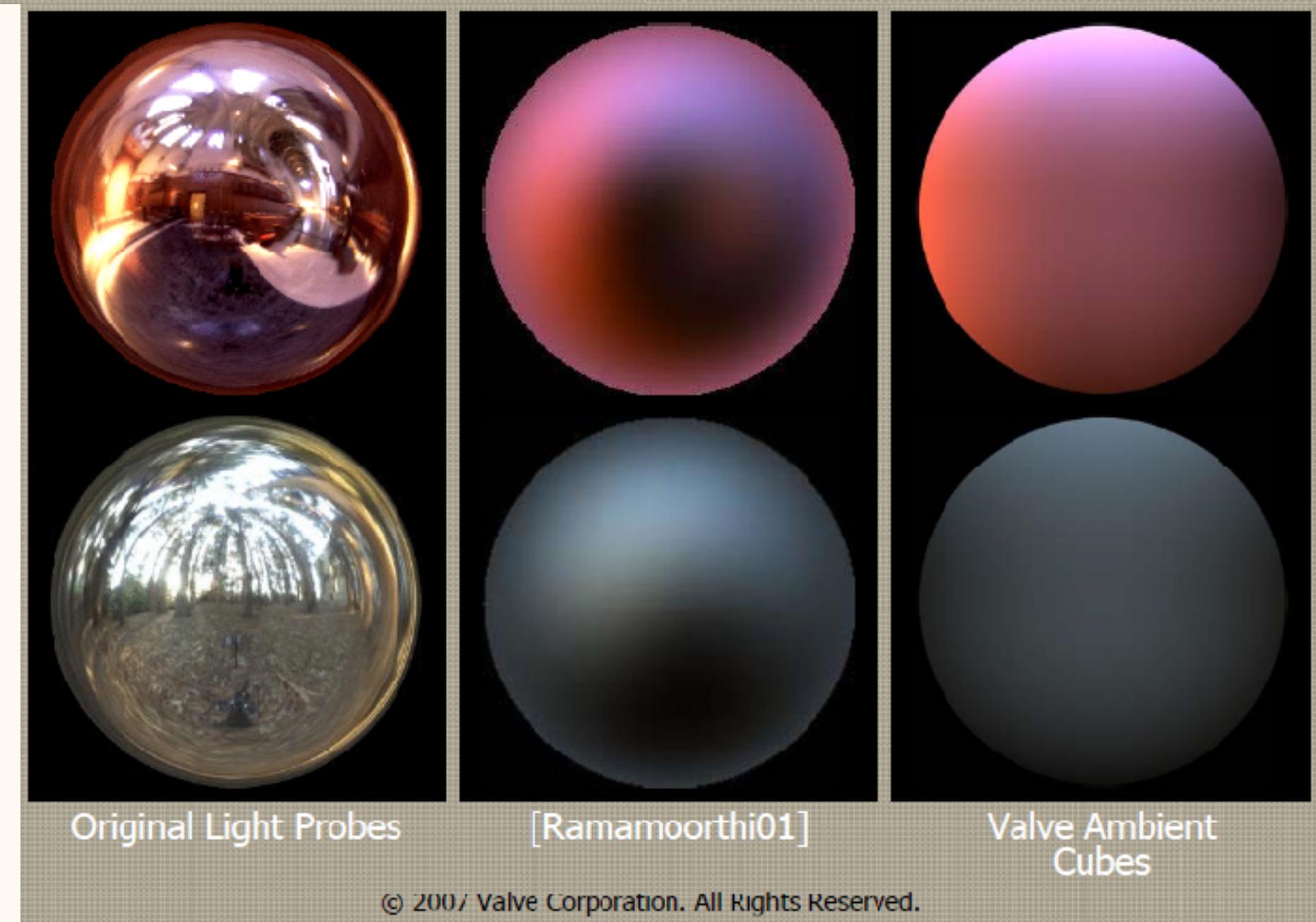


Spatially-varying Directional Ambient

- Spatially-varying directional ambient
- Ambient cube
 - Six RGB lobes stored in shader constants



```
float3 AmbientLight( const float3 worldNormal )
{
    float3 nSquared = worldNormal * worldNormal;
    int3 isNegative = ( worldNormal < 0.0 );
    float3 linearColor;
    linearColor = nSquared.x * cAmbientCube[isNegative.x] +
                 nSquared.y * cAmbientCube[isNegative.y+2] +
                 nSquared.z * cAmbientCube[isNegative.z+4];
    return linearColor;
}
```



VIEW INDEPENDENT TERMS

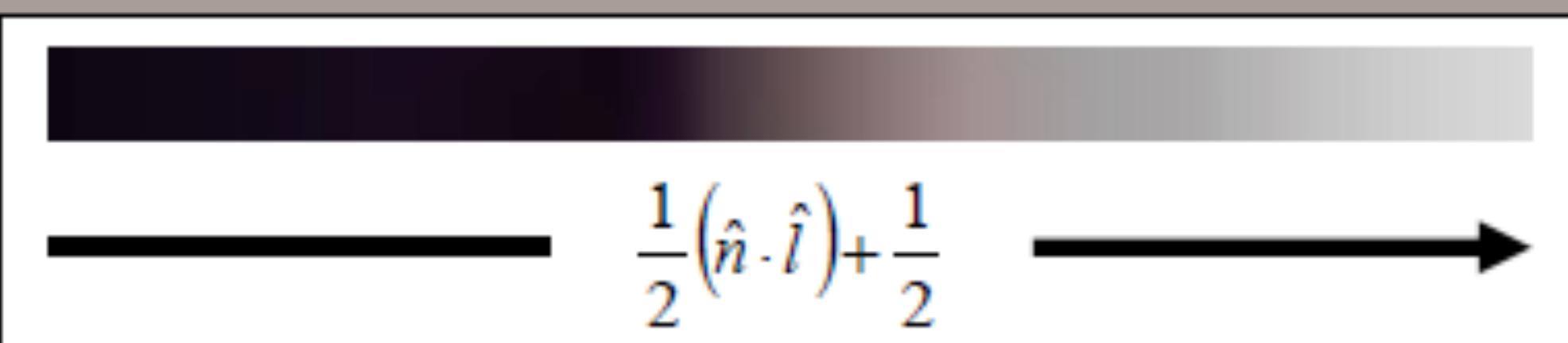
$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w \left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right]$$

- Spatially-varying directional ambient
- Modified Lambertian terms
 - Unclamped Lambertian term
 - Scale, bias and exponent



VIEW INDEPENDENT TERMS

$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w \left(\left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right) \right]$$

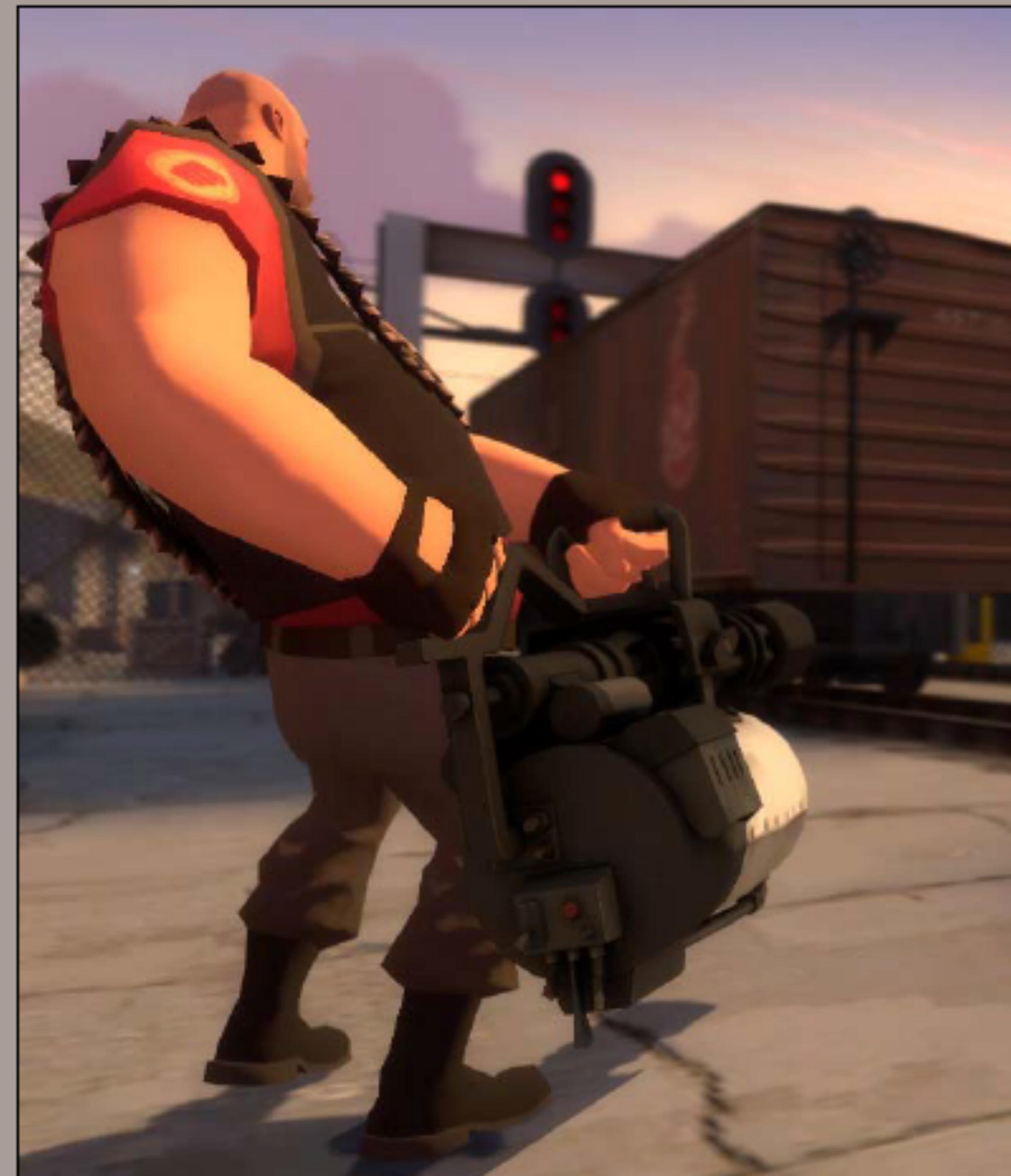
- Spatially-varying directional ambient
- Modified Lambertian terms
 - Unclamped Lambertian term
 - Scale, bias and exponent
 - Warping function
-  $\frac{1}{2} (\hat{n} \cdot \hat{l}) + \frac{1}{2}$



VIEW INDEPENDENT TERMS

$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w \left(\left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right) \right]$$

- Spatially-varying directional ambient
- Modified Lambertian terms
 - Unclamped Lambertian term
 - Scale, bias and exponent
 - Warping function
- Albedo



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max \left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term
 - f_r rim Fresnel term, $(1-(n \cdot v))^4$



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term
 - f_r rim Fresnel term, $(1-(n \cdot v))^4$
 - k_r rim mask texture
 - k_s specular mask texture



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term
 - f_r rim Fresnel term, $(1 - (n \cdot v))^4$
 - k_r rim mask texture
 - k_s specular mask texture

An approximation proposed by Schlick[1994] for Fresnel

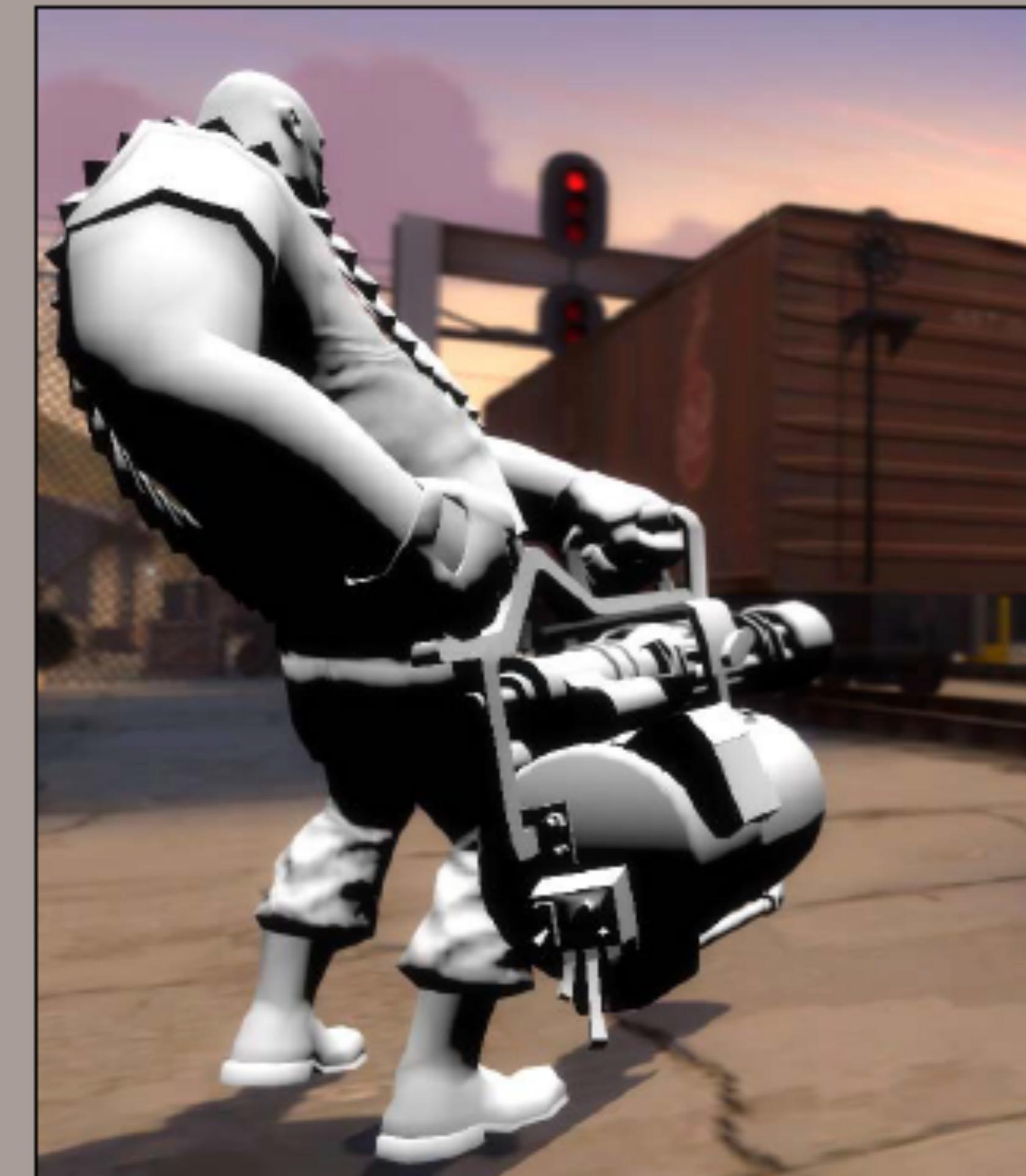
$$F = \eta + (1 - \eta) (1 - N \cdot V)^5$$



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + \boxed{(\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})}$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term
 - f_r rim Fresnel term, $(1-(n \cdot v))^4$
 - k_r rim mask texture
 - k_s specular mask texture
- Dedicated rim lighting
 - $a(v)$ Directional ambient evaluated with v
 - k_r same rim mask
 - f_r same rim Fresnel
 - $n \cdot u$ term that makes rim highlights tend to come from above (u is up vector)



VIEW-DEPENDENT TERMS

$$\sum_{i=1}^L \left[c_i k_s \max\left(f_s (\hat{v} \cdot \hat{r}_i)^{k_{spec}}, f_r k_r (\hat{v} \cdot \hat{r}_i)^{k_{rim}} \right) \right] + (\hat{n} \cdot \hat{u}) f_r k_r a(\hat{v})$$

- Multiple Phong terms per light
 - k_{rim} broad, constant exponent
 - k_{spec} exponent (constant or texture)
 - f_s artist tuned Fresnel term
 - f_r rim Fresnel term, $(1-(n \cdot v))^4$
 - k_r rim mask texture
 - k_s specular mask texture
- Dedicated rim lighting
 - $a(v)$ Directional ambient evaluated with v
 - k_r same rim mask
 - f_r same rim Fresnel
 - $n \cdot u$ term that makes rim highlights tend to come from above (u is up vector)



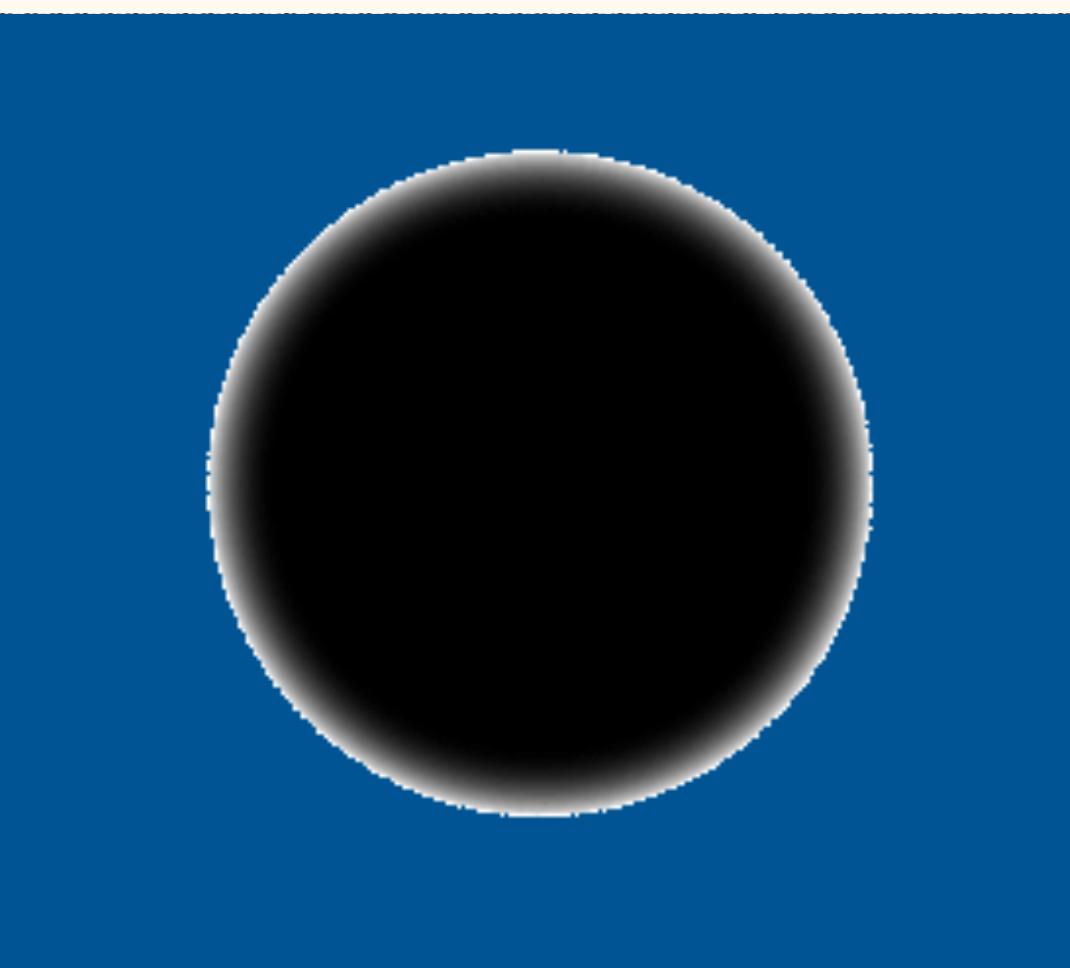
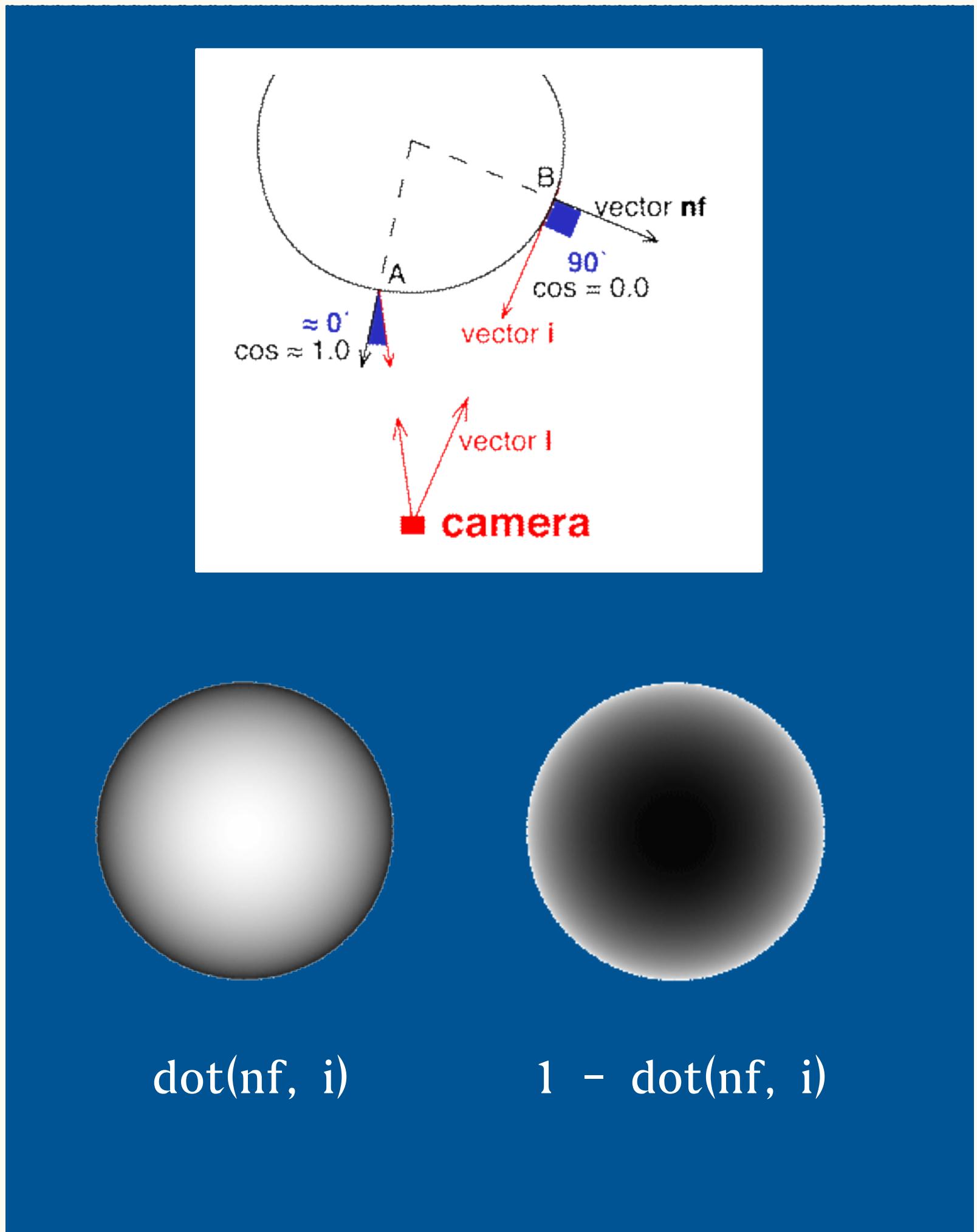


Real-time Skin Rendering

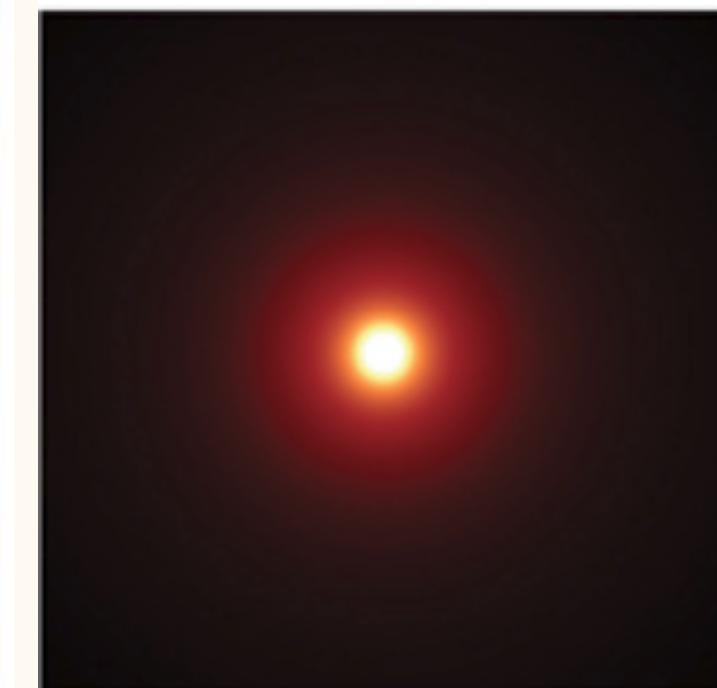
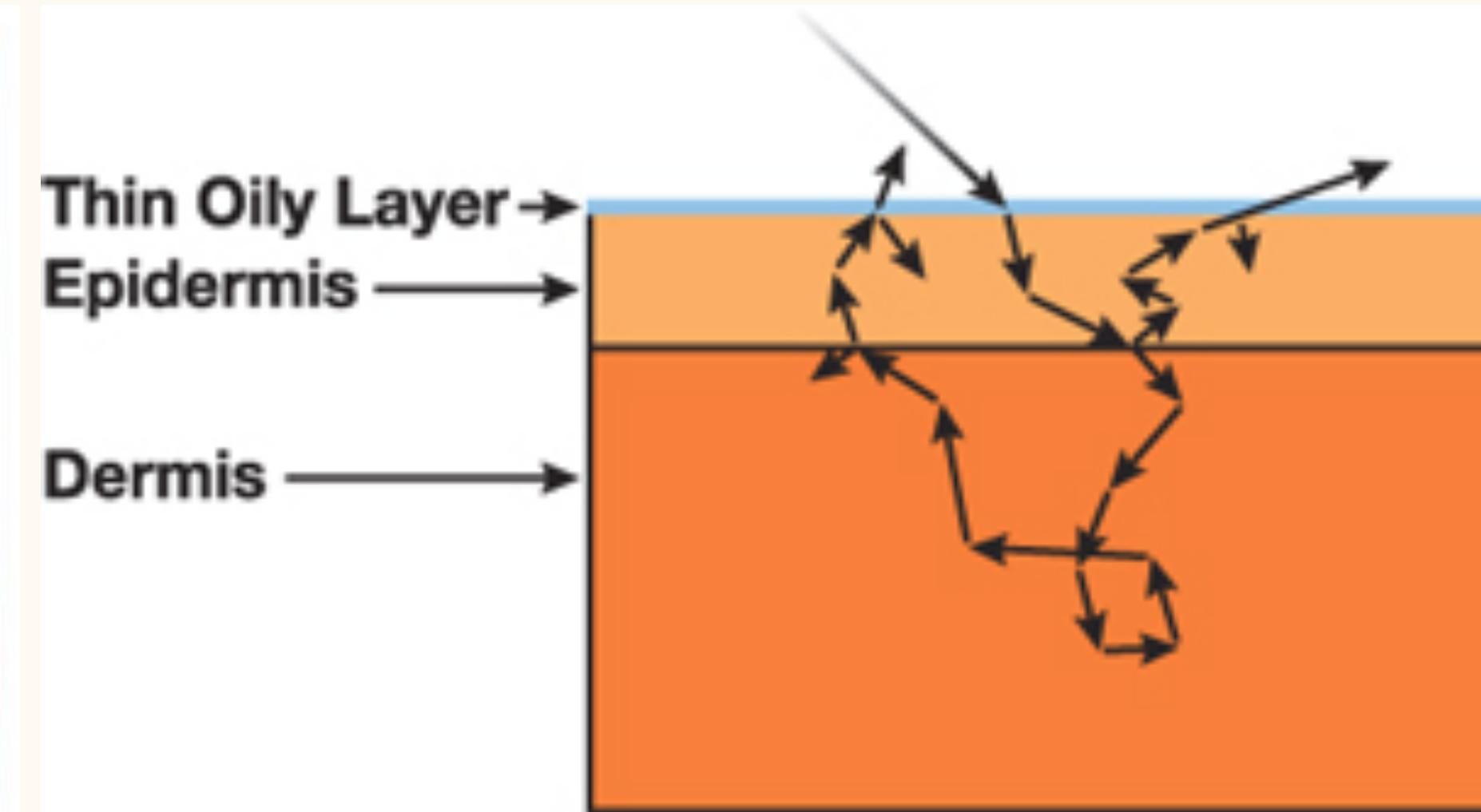
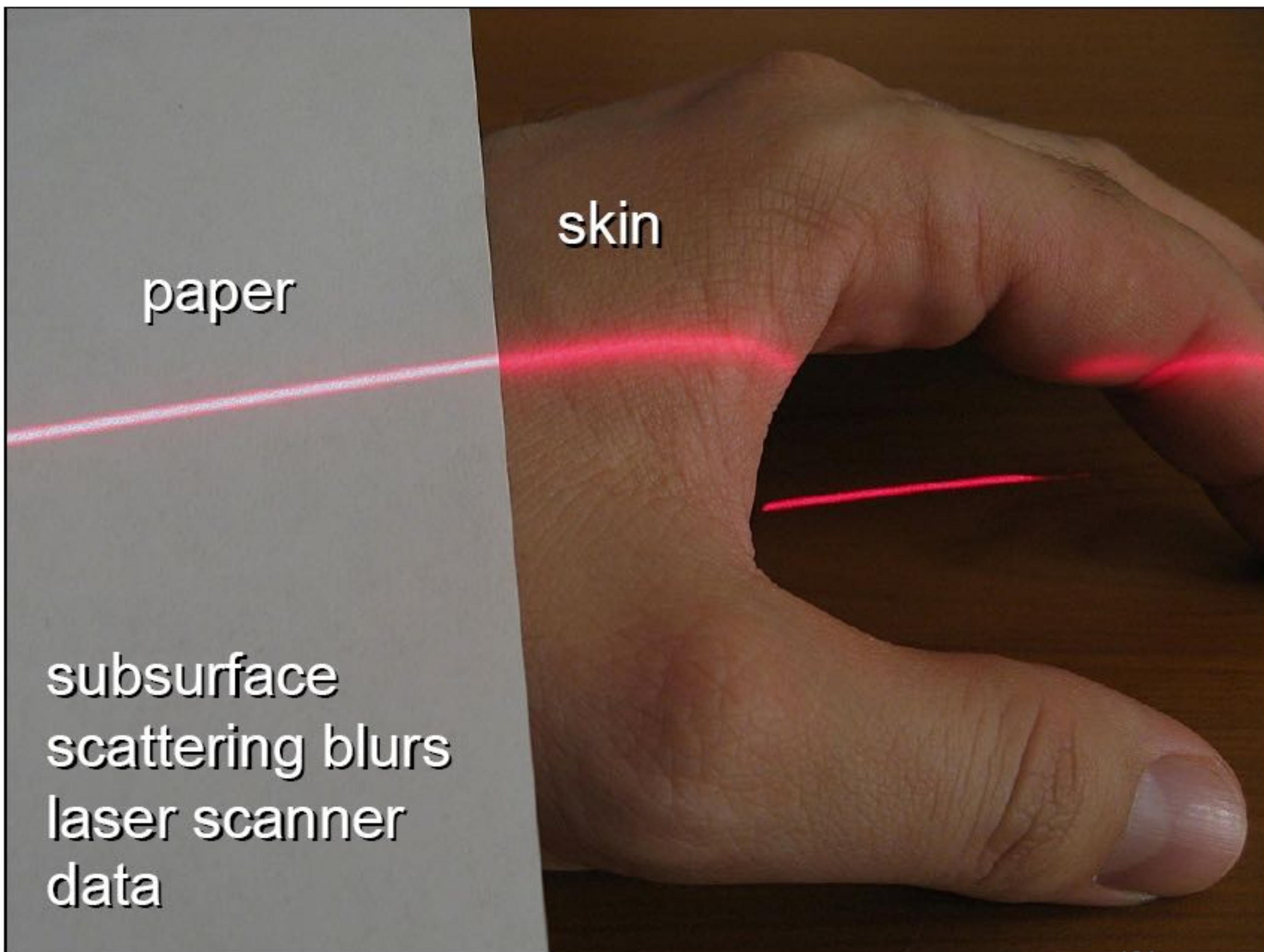
Skin Rendering – Rim Lighting

- A simple way to simulate the skin material
- Shader sample code:

```
float dot = 1 - dot(nf, i);  
diffusecolor = smoothstep(1.0 - rim_width, 1.0, dot);
```

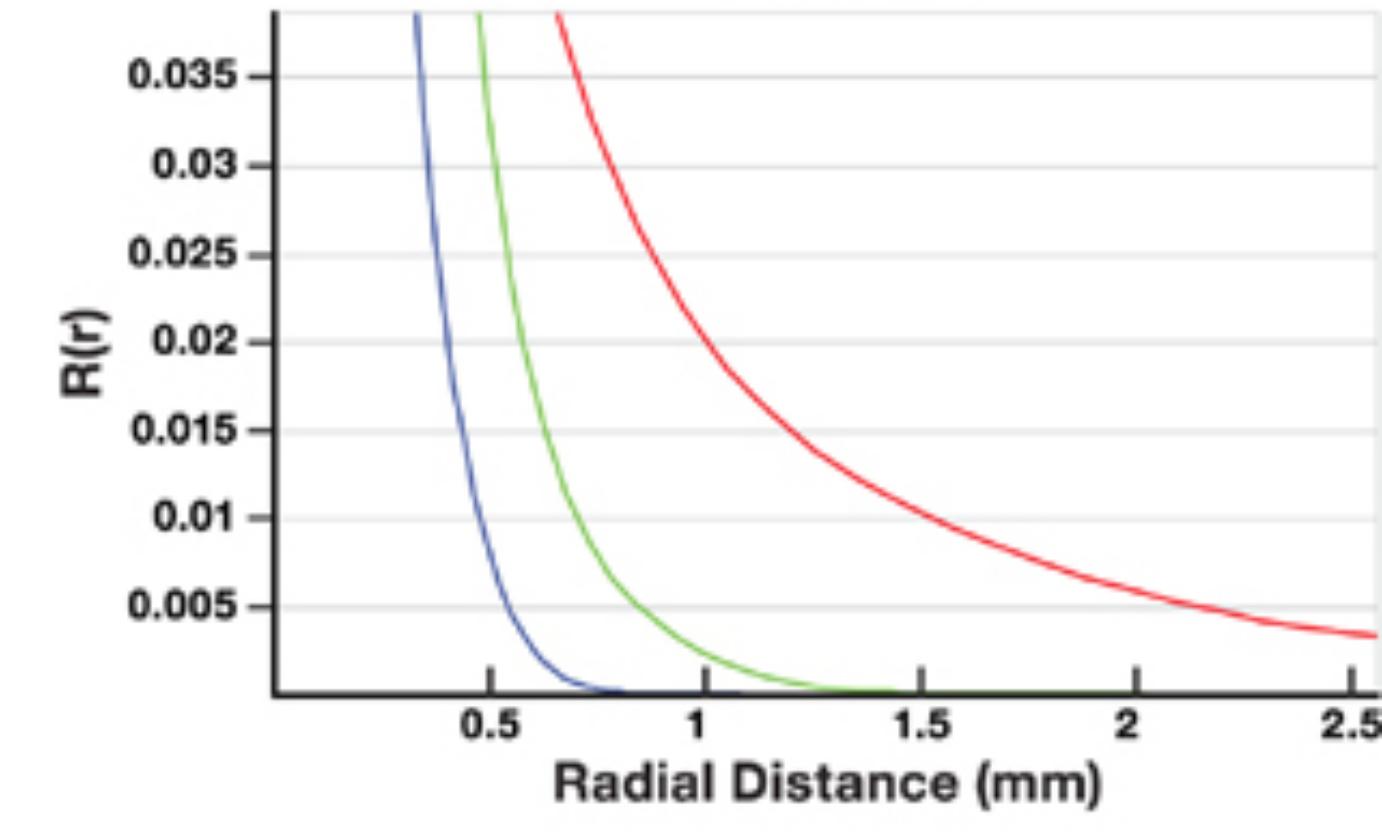


Skin Rendering - Sub-surface Rendering



401

(a)



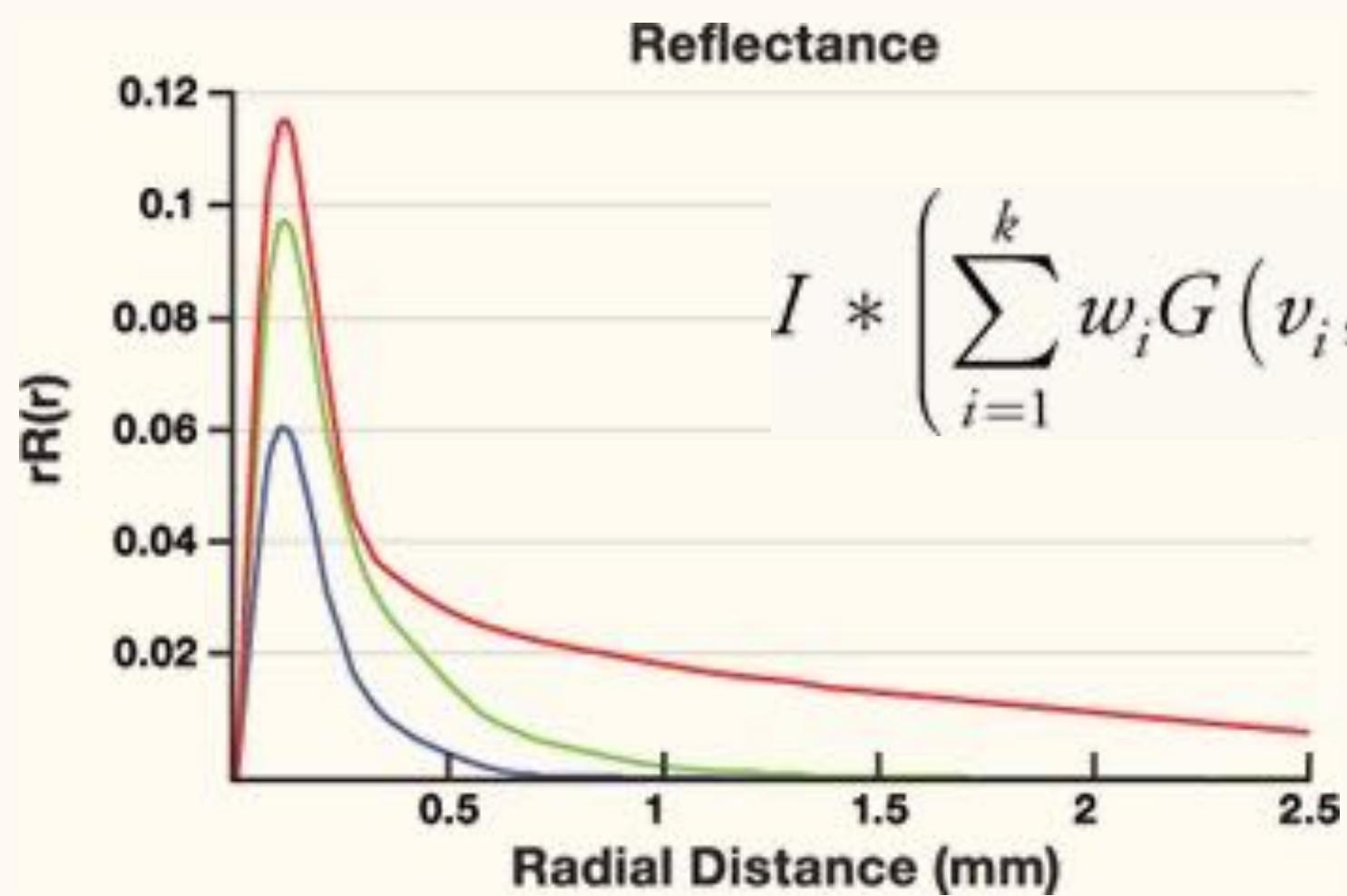
(b)

Skin Rendering – Sub-surface Rendering

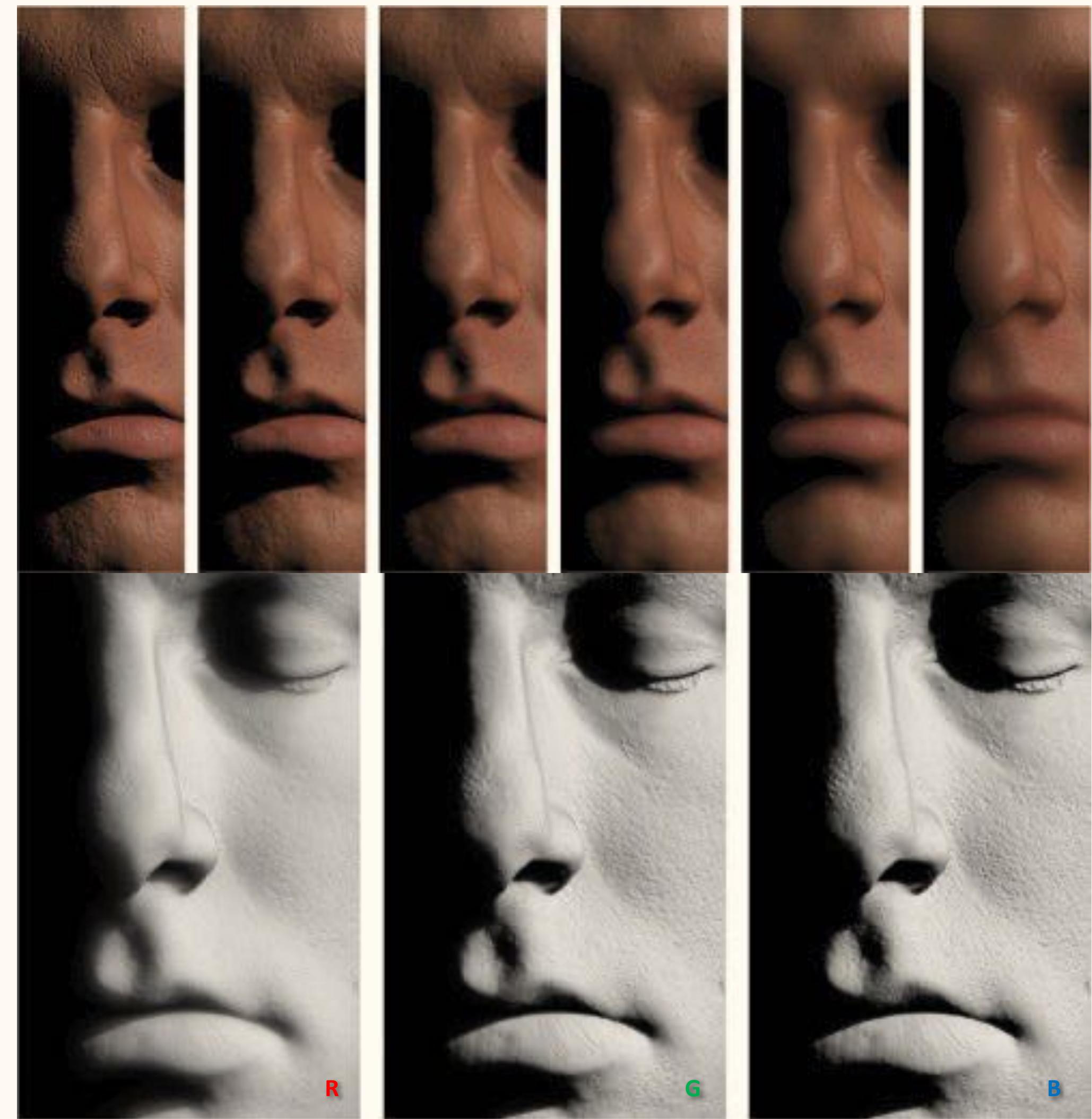
- Texture Diffusion
 - Render Radiance to Texture Space
 - Use Blurring to Simulate the Light Diffusion on Skin
 - Reference :
 - GPU Gems III, Chapter 14
 - Can be reached at nVidia developer website
 - Blend six different levels of Gaussian filters to simulate texture diffusion
- Hybrid Normal maps
 - 2007 By Alex Ma (馬萬鈞)
 - Light Stage from USC ICT
 - Now popular in AAA game tile (Activision, EA, ...)

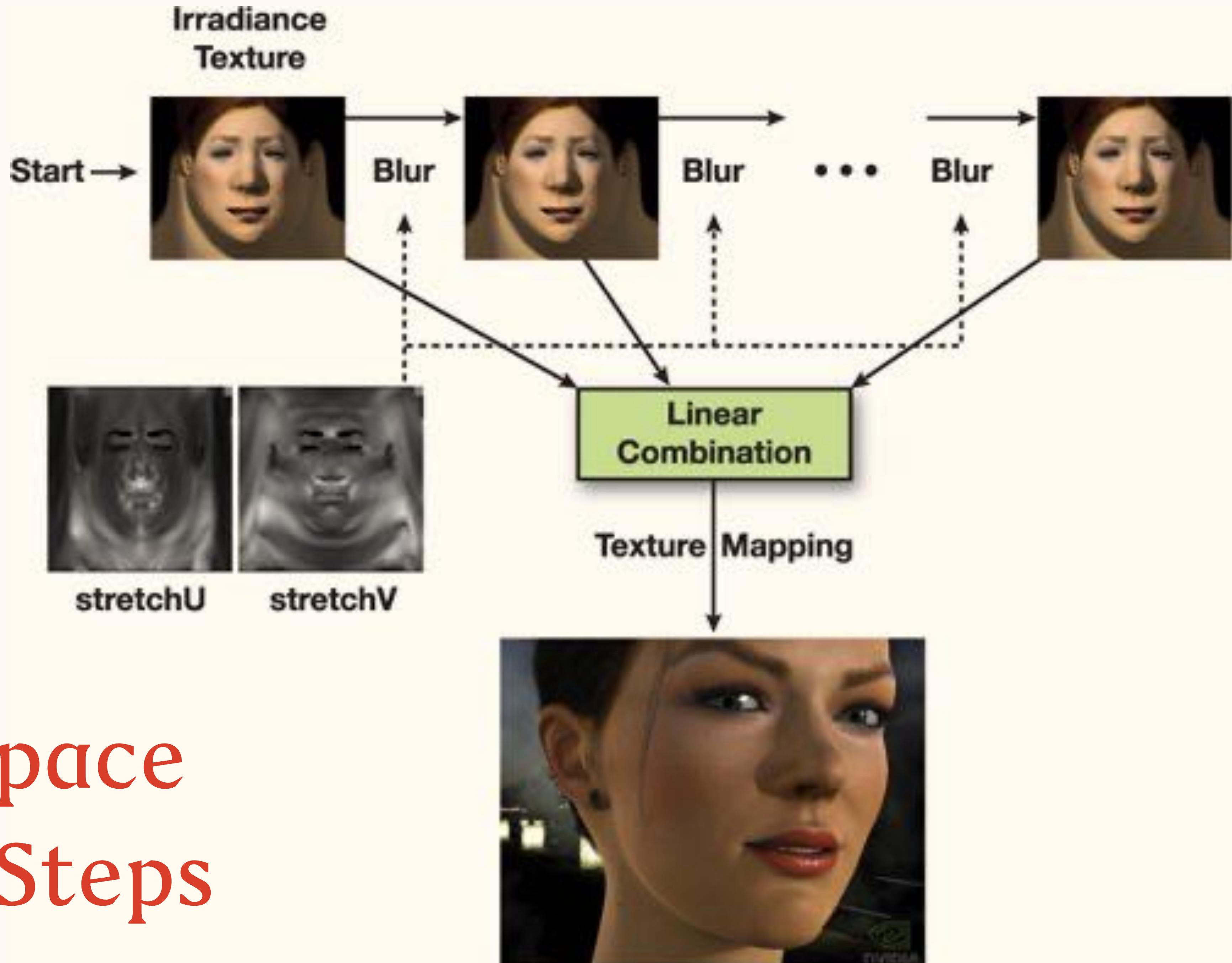
Skin Rendering - Texture Diffusion

Variance (mm ²)	Blur Weights		
	Red	Green	Blue
0.0064	0.233	0.455	0.649
0.0484	0.100	0.336	0.344
0.187	0.118	0.198	0
0.567	0.113	0.007	0.007
1.99	0.358	0.004	0
7.41	0.078	0	0



$$I * \left(\sum_{i=1}^k w_i G(v_i, r) \right) = \sum_{i=1}^k w_i I * G(v_i, r).$$

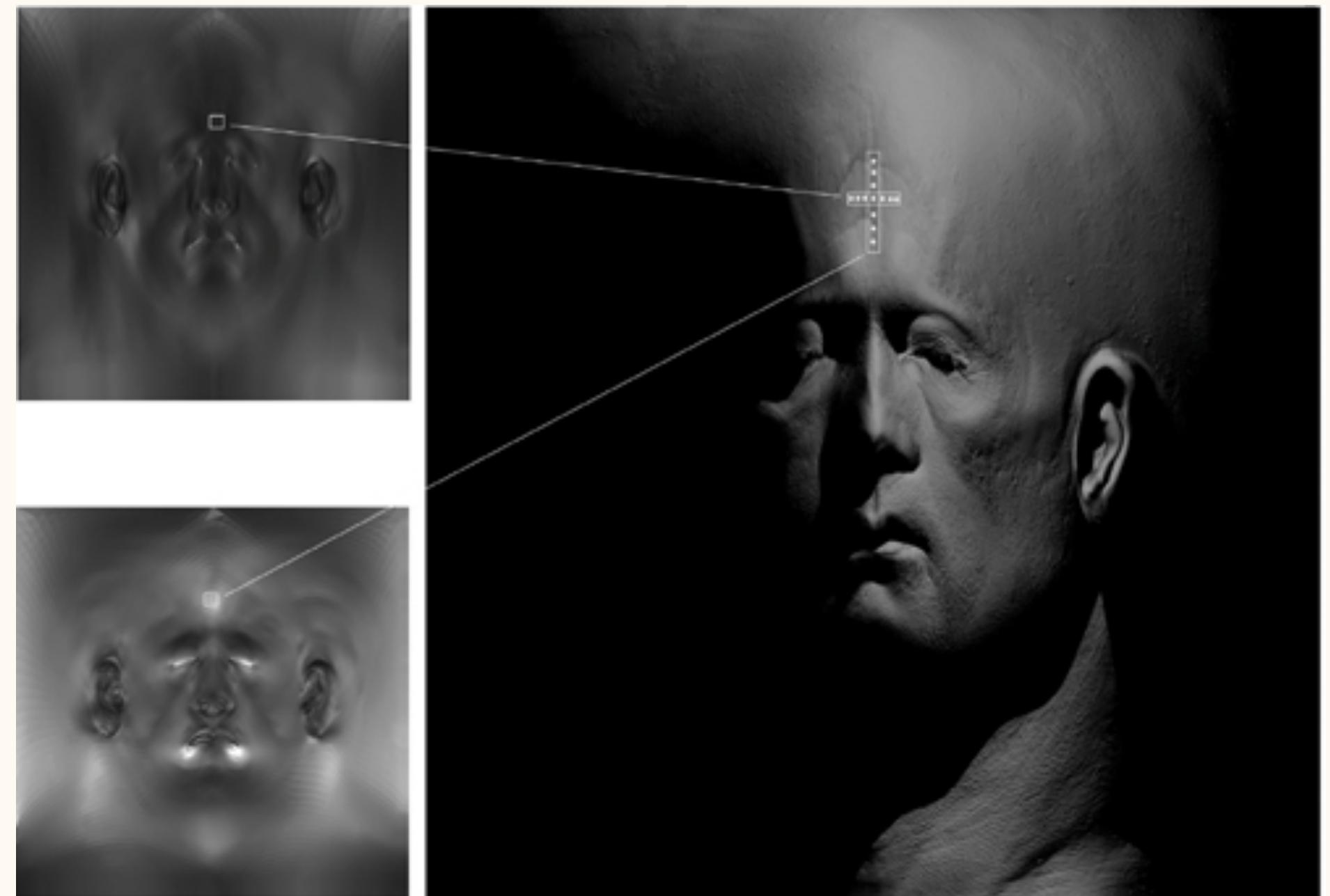




Texture-space Diffusion Steps

Correcting for UV Distribution Using a Stretch Texture

```
float2 computeStretchMap(float3 worldCoord : TEXCOORD0)
{
    float3 derivu = ddx(worldCoord);
    float3 derivv = ddy(worldCoord);
    float stretchU = 1.0/length(derivu);
    float stretchV = 1.0 / length(derivv);
    return float2(stretchU, stretchV); // a two-component texture
}
```



A Physically-based Specular Reflection for Skins

```
float KS_Skin_Specular(float3 N, // Bumped surface normal
                        float3 L, // Points to light
                        float3 V, // Points to eye
                        float m, // Roughness
                        float rho_s // Specular brightness )
{
    float result = 0.0;
    float ndotl = dot(N, L);
    if (ndotl > 0.0)
    {
        float3 h = L + V; // Unnormalized half-way vector
        float3 H = normalize(h);
        float ndoth = dot(N, H);
        float PH = pow(2.0*tex2D(beckmannTex,float2(ndoth,m)), 10.0);
        float F = fresnelReflectance(H, V, 0.028);
        float frSpec = max(PH*F/dot(h, h), 0);
        result = ndotl*rho_s*frSpec;           // BRDF * dot(N,L) * rho_s
    }
    return result;
}
```

A Physically-based Specular Reflection for Skins

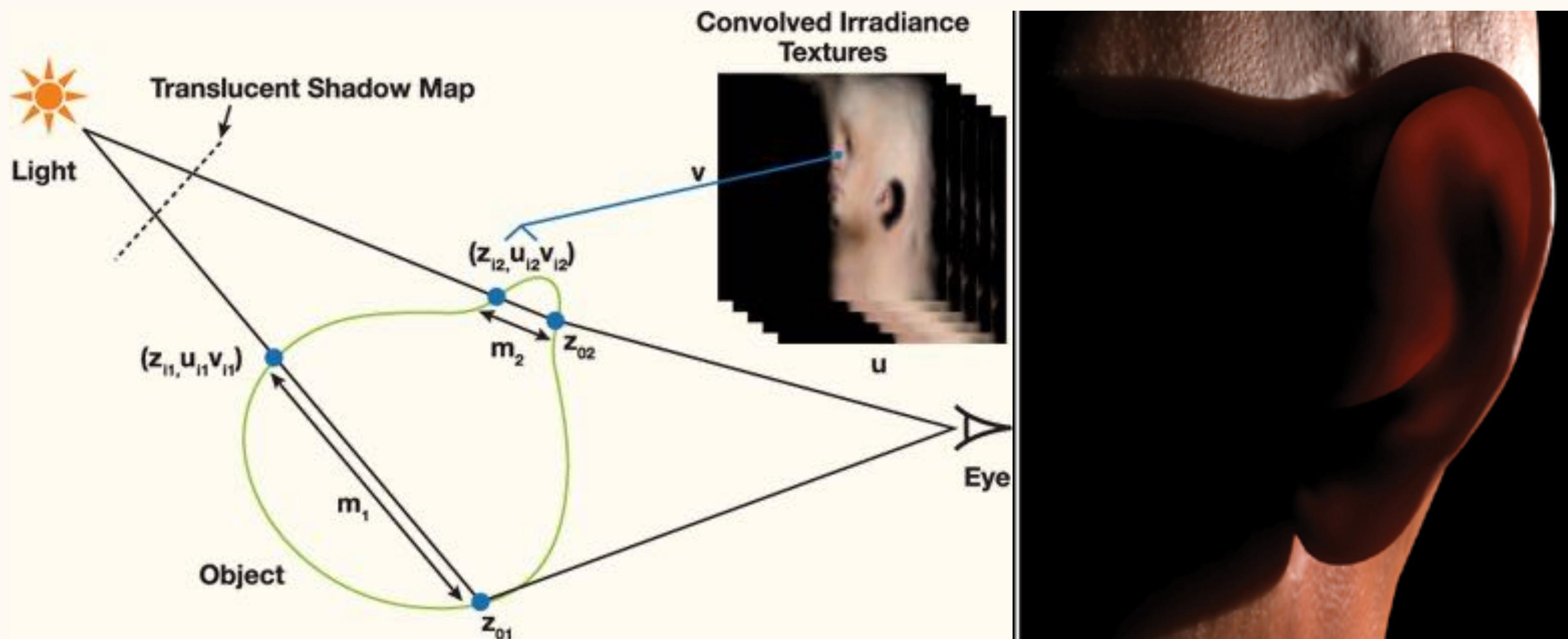
- Specular reflection from skin is white.
- Based on Kelemen/Szirmay-Kalos reflection model
 - A simplified Cook-Torrance model
 - <http://www.hungrycat.hu/microfacet.pdf>

```
specularLight += lightColor[i]*lightShadow[i]*rho_s*specBRDF(N, V, L[i], eta, m)*saturate(dot(N, L[i]));
```

```
float fresnelReflectance( float3 H, float3 V, float F0 )
{
    float base = 1.0 - dot( V, H );
    float exponential = pow( base, 5.0 );
    return exponential + F0 * ( 1.0 - exponential );
}
// note, F0 is 0.028 for skin
```

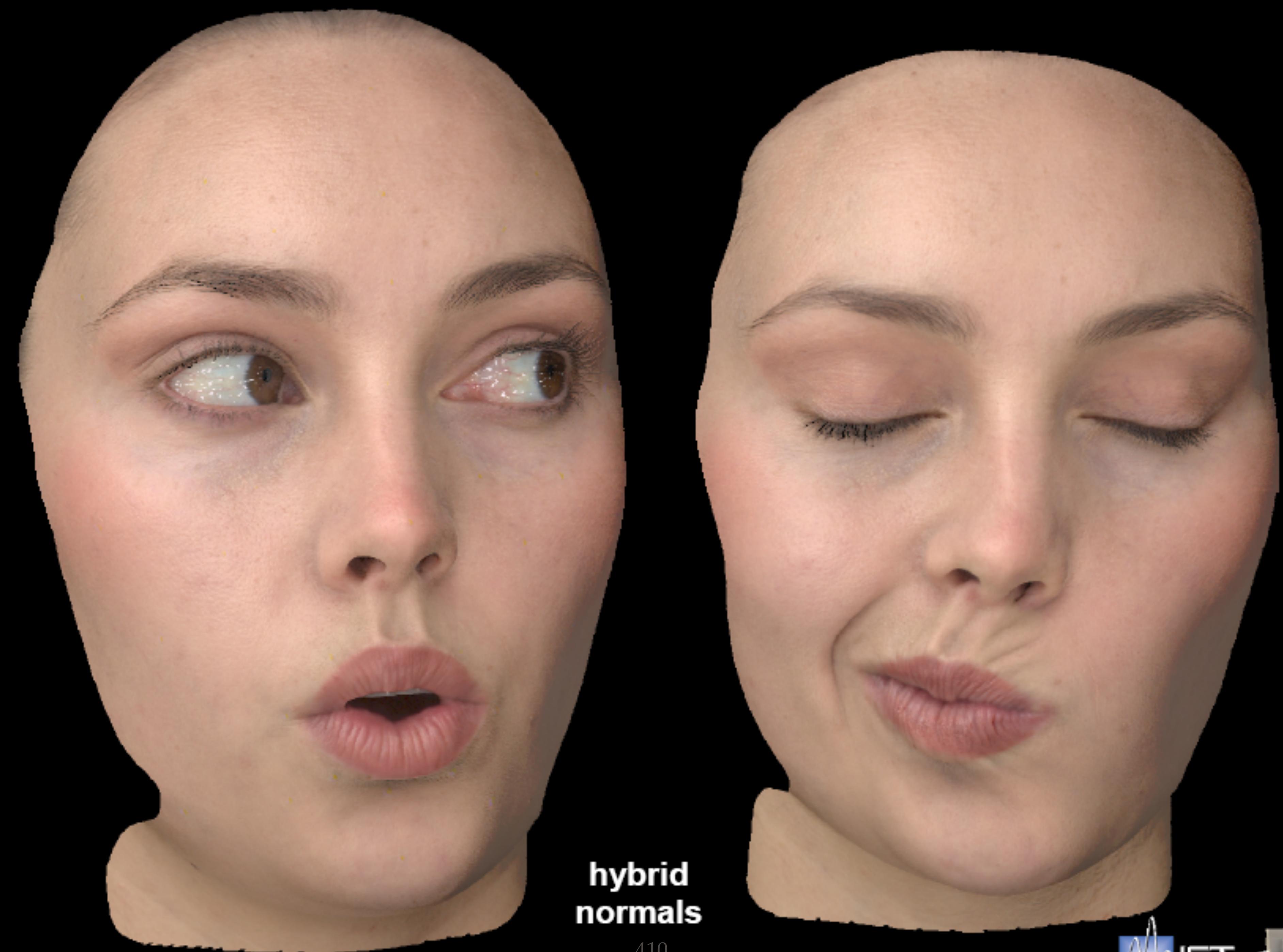
Real-time Subsurface Scattering Effect

- Real-time sub-surface scattering
 - Modified translucent shadow maps



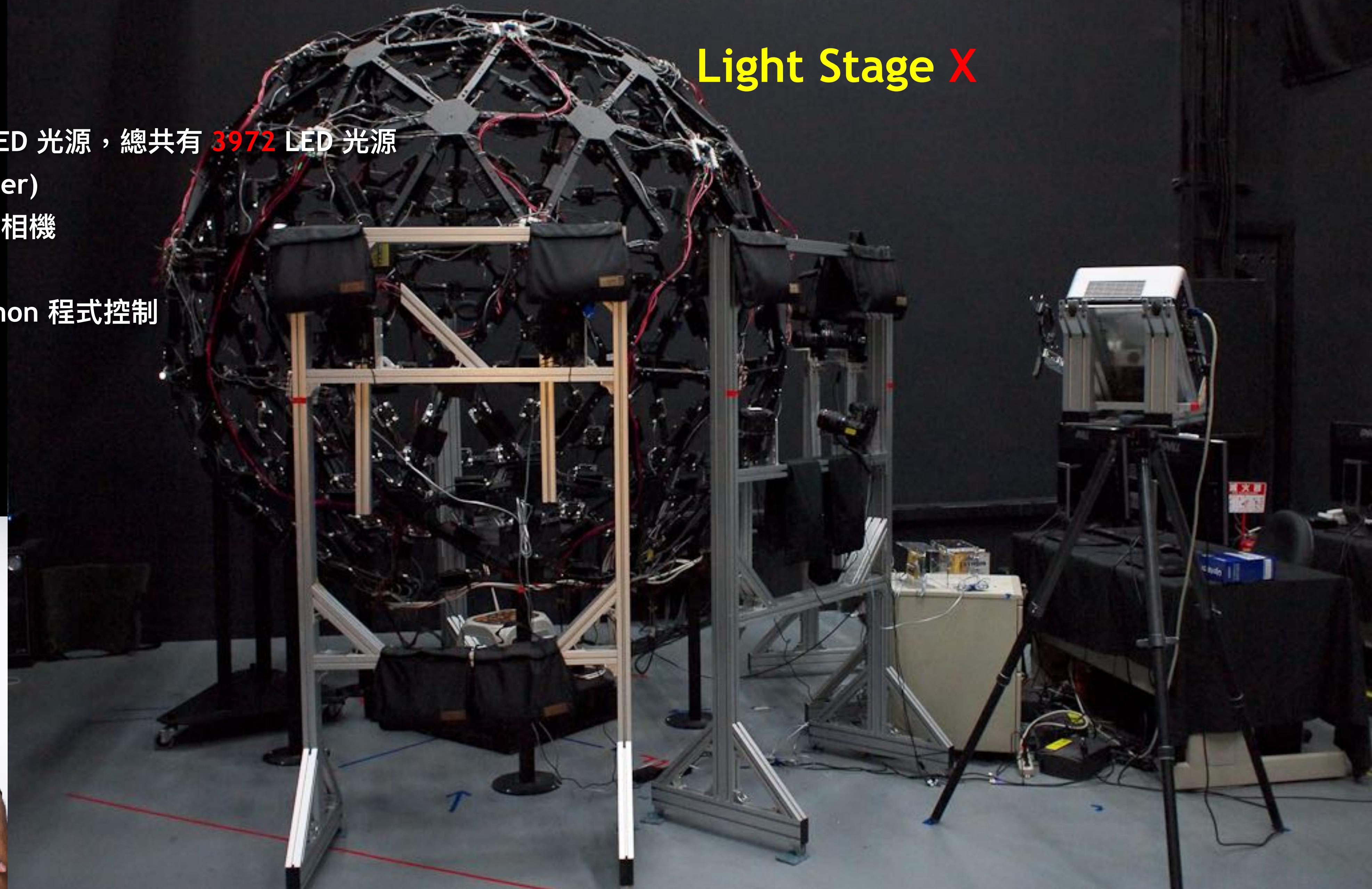
Real-time Skin Rendering

補充資料

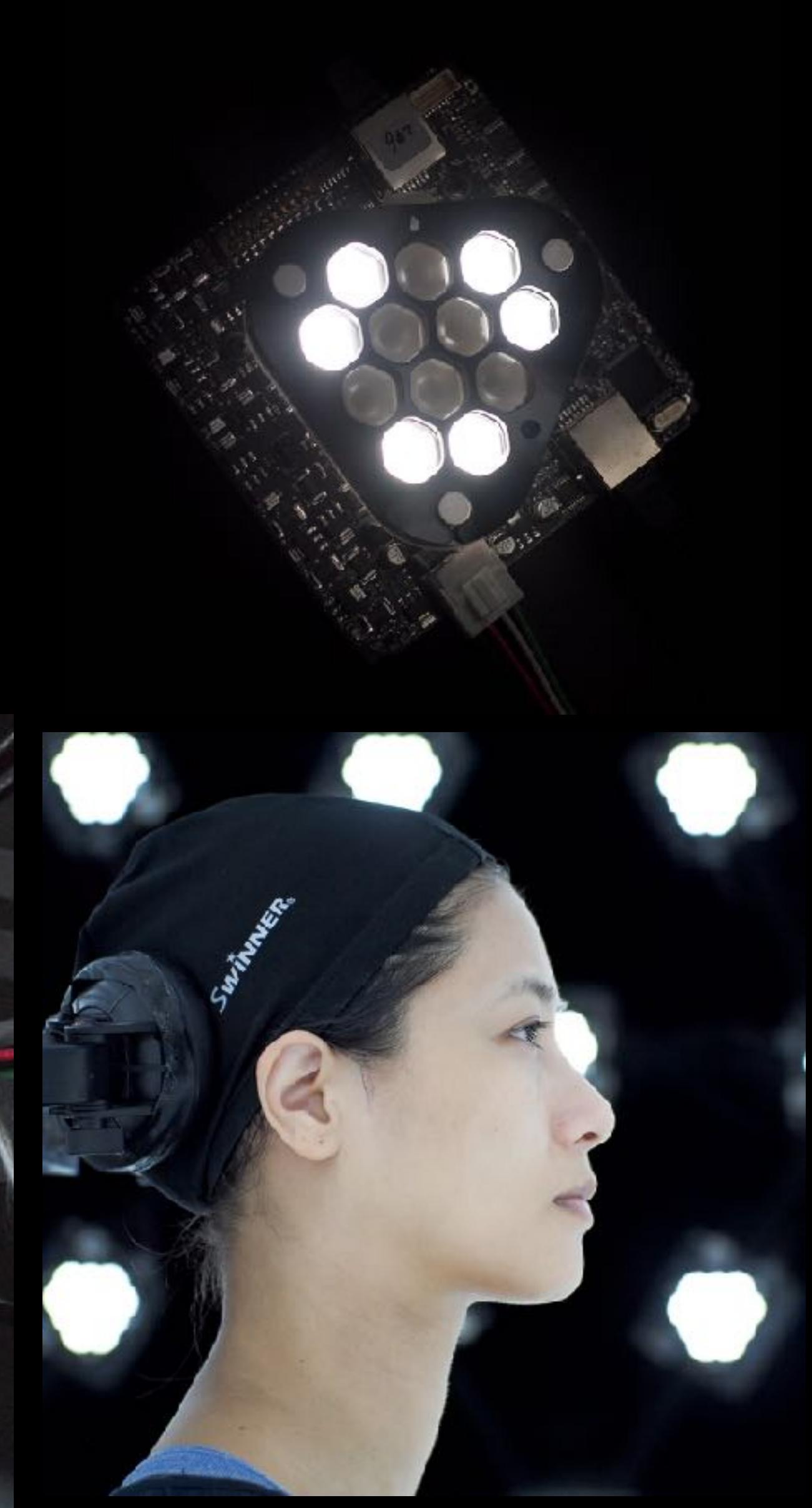
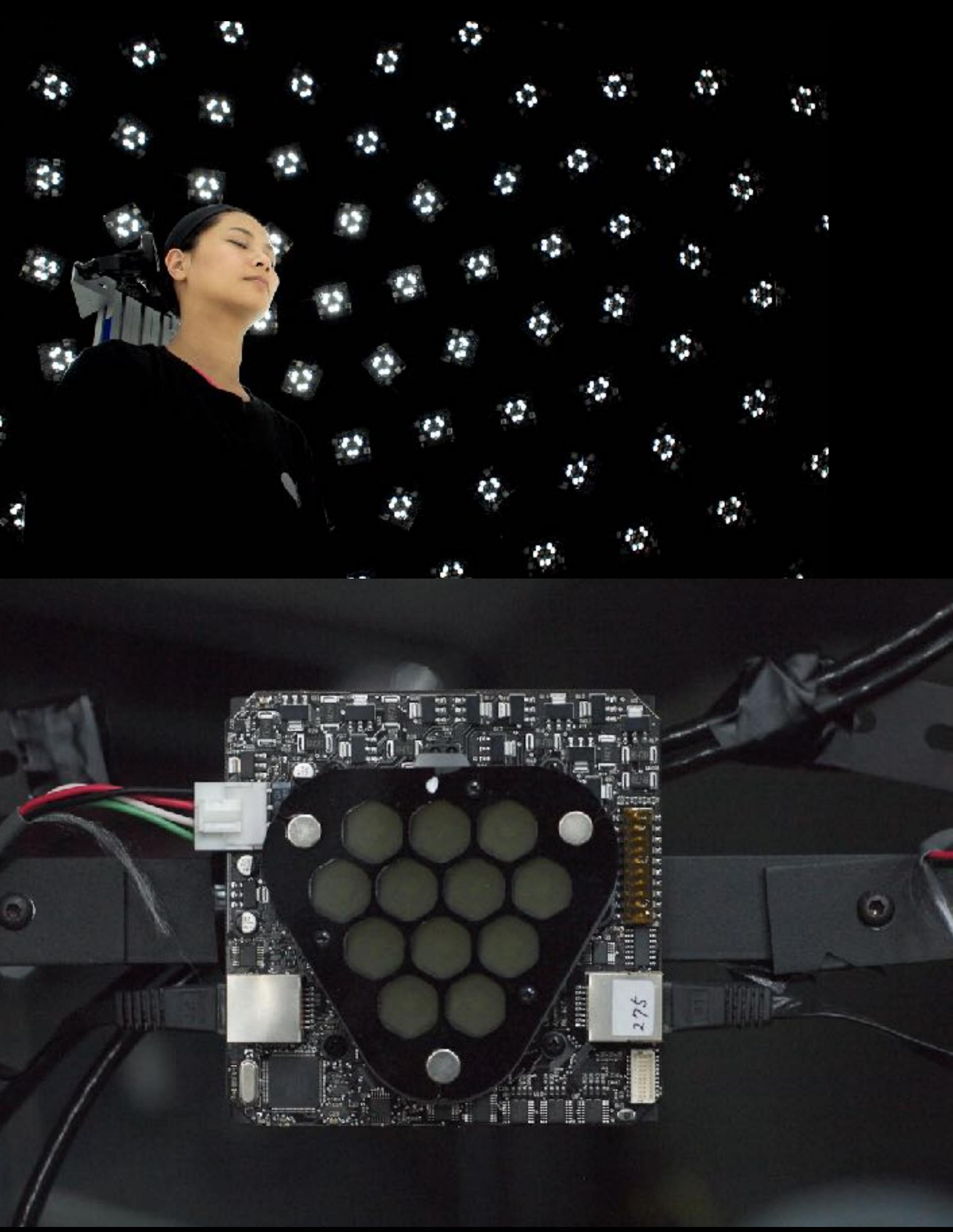
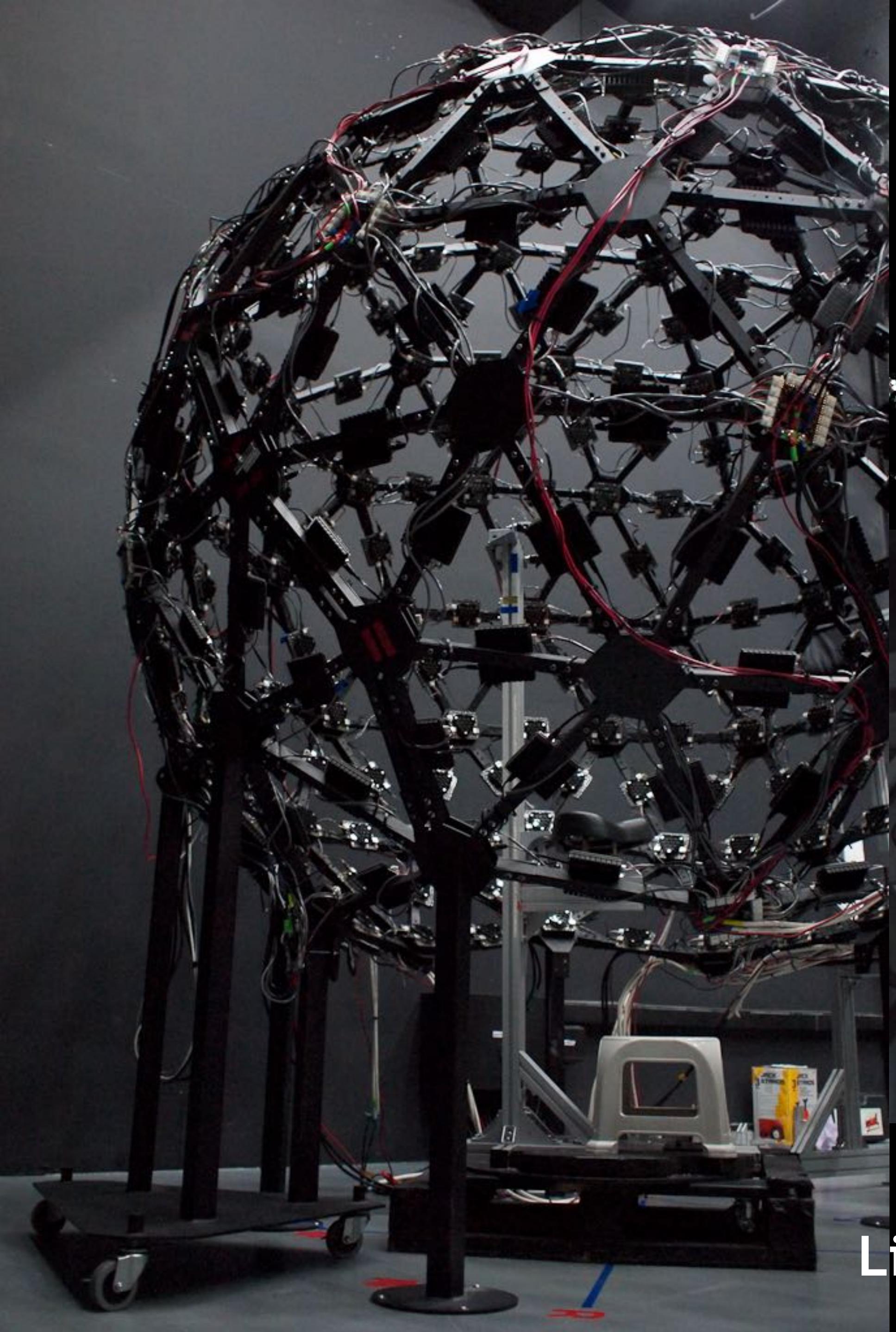


410

- 簡稱 LSX
- 直徑 2.5 m 球型結構
- 331 Lighting Units
 - 每盞 Lighting Unit 有 12 LED 光源，總共有 3972 LED 光源
 - 線性偏光鏡 (Linear Polarizer)
- 6 台 Canon 1D MK4 單眼數位相機
- 3 台投影機
- 所有設備以網路串聯，以 Python 程式控制
- Imaged based modeling
 - Human face scanning
- Designed by USC ICT

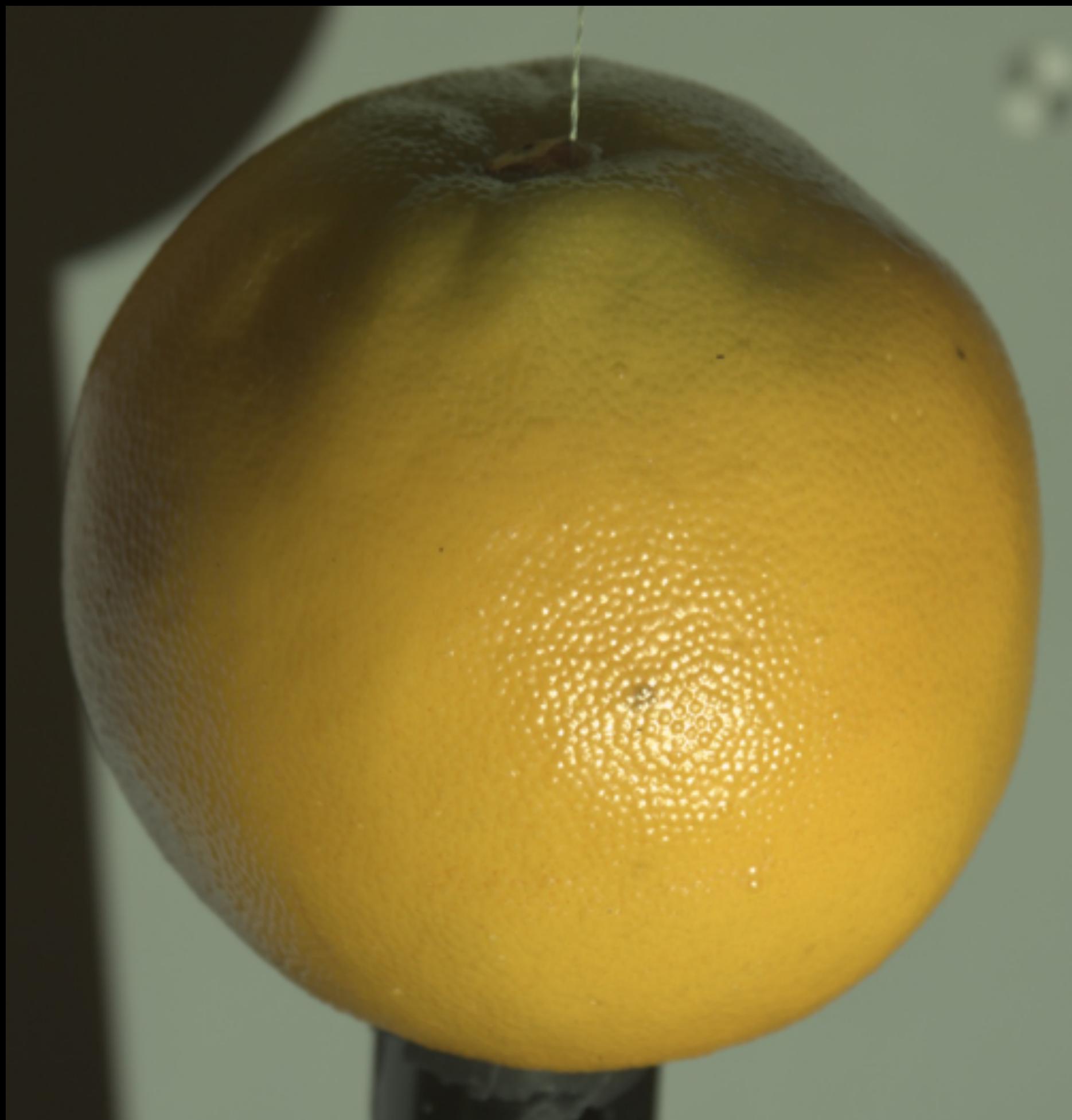


Light Stage @ Next Media Animation Media Lab, Taipei

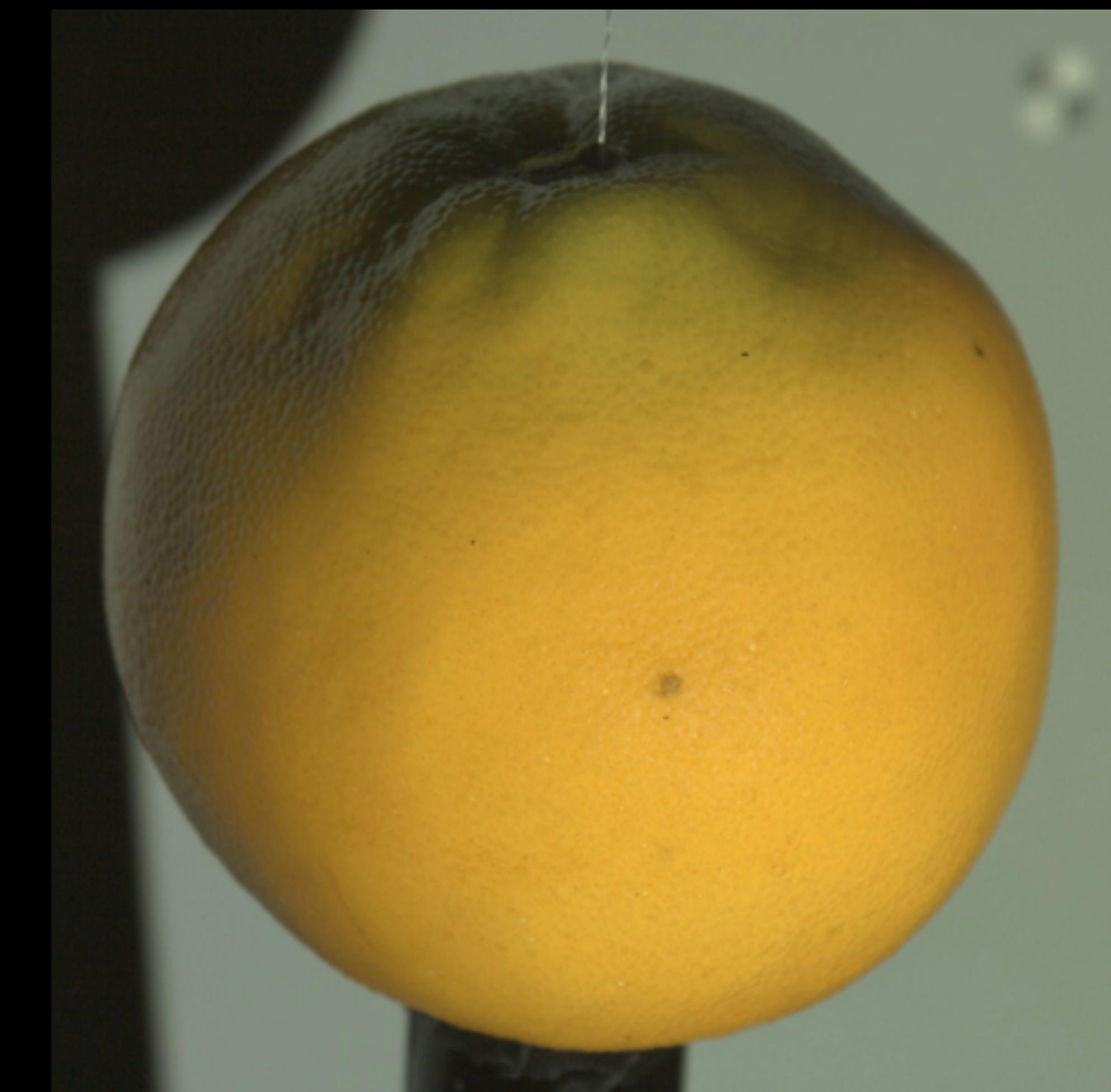


Light Stage @ Next Media Animation Media Lab, Taipei

Remove Specular by Applying Linear Polarizer 線性偏光鏡



without a linear polarizer

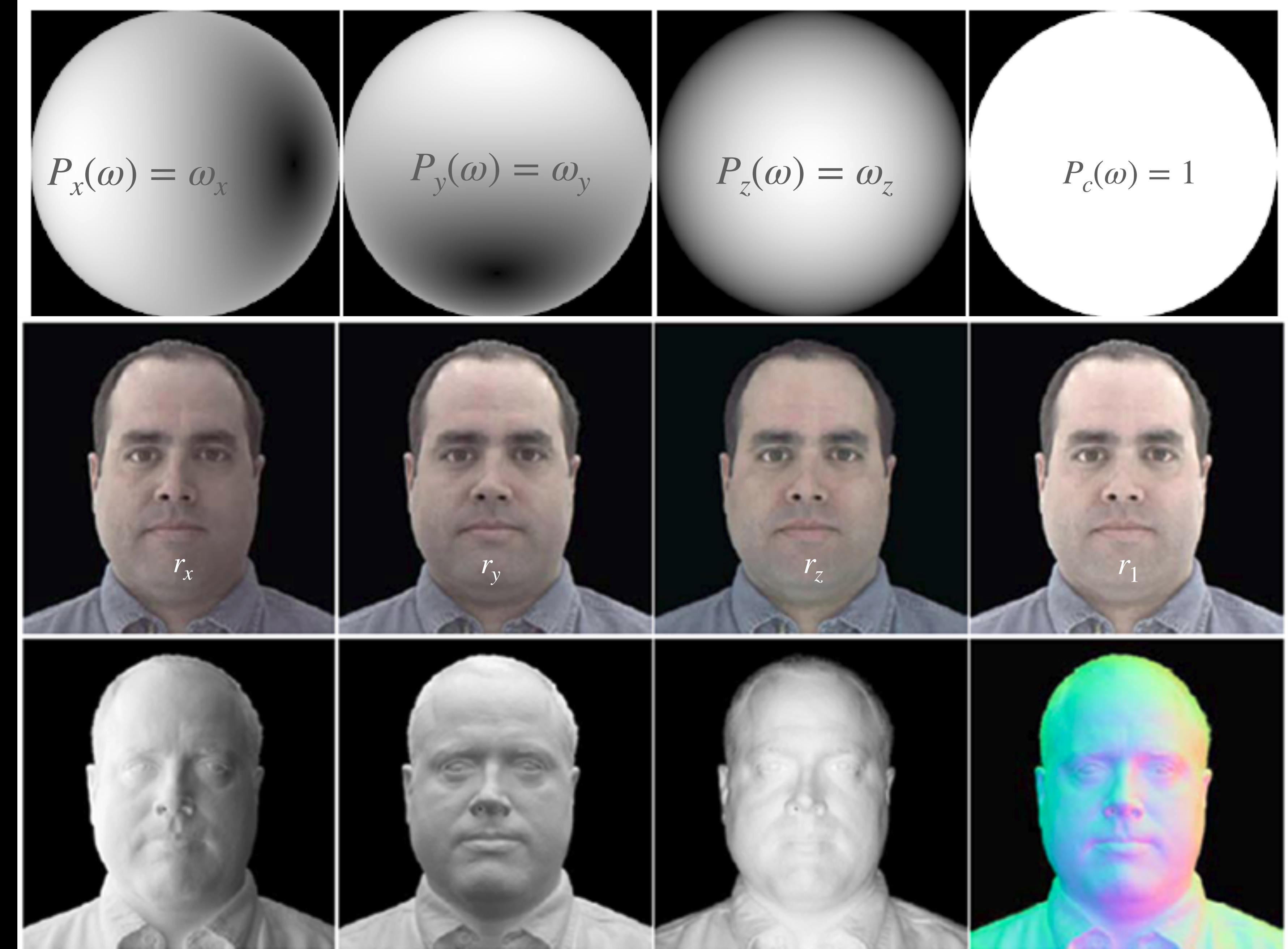


with a linear polarizer

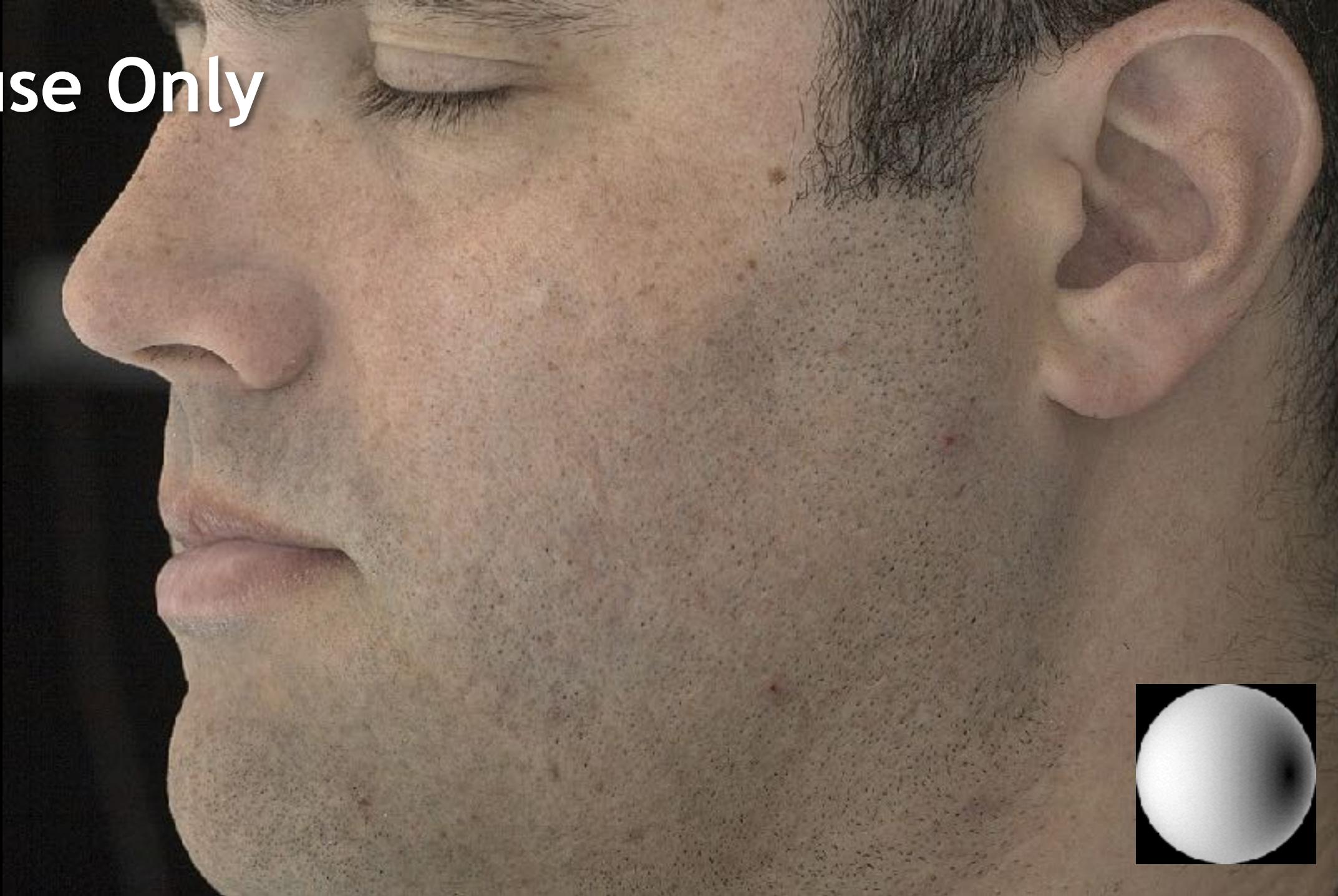
Capture Normals

$Kd^*(N \cdot L)$

$$\frac{r_x}{r_1} = \frac{2n_x + 3}{6}$$

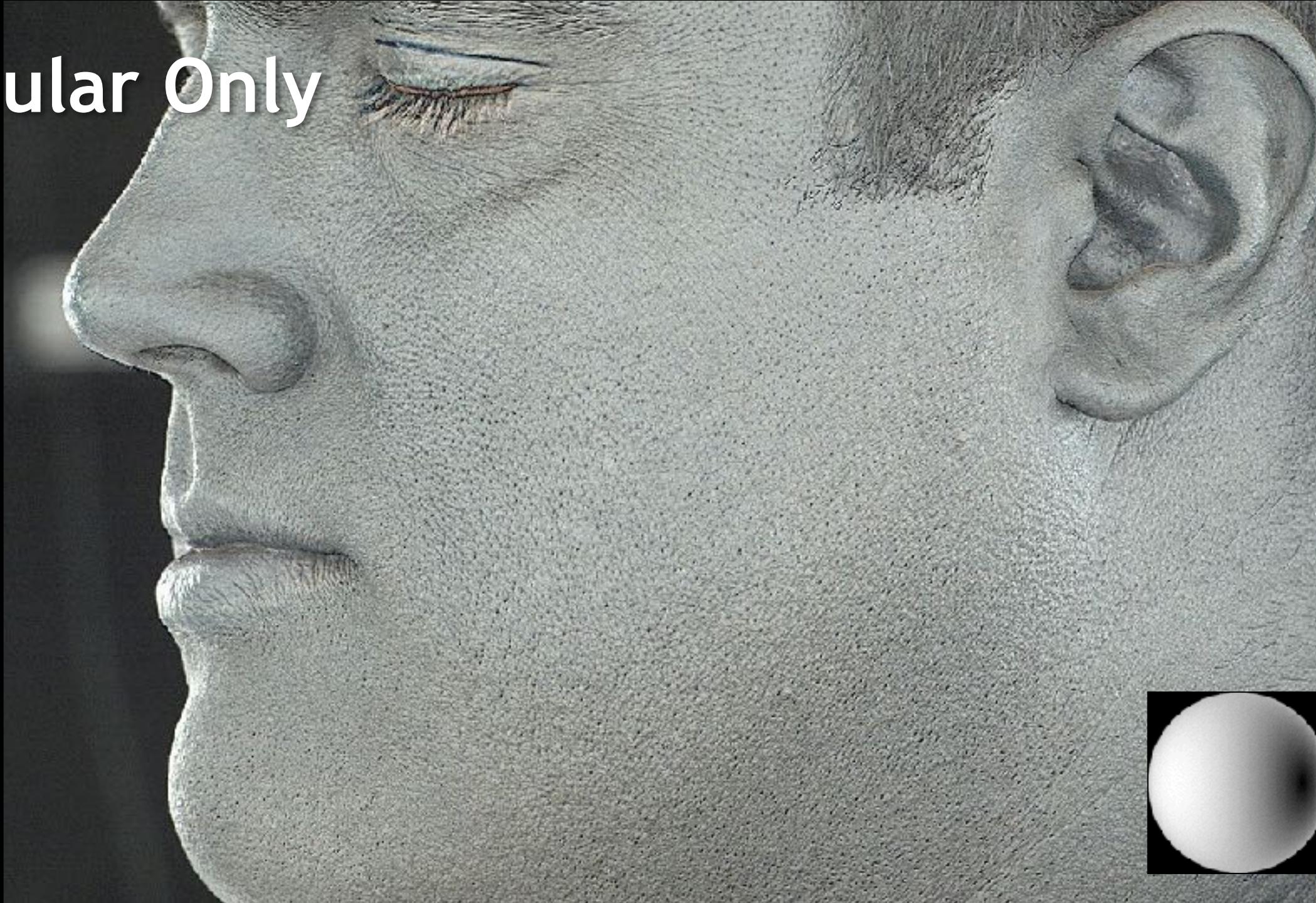


Diffuse Only



Diffuse Normal

Specular Only



Specular Normal



Normal map

Red diffuse



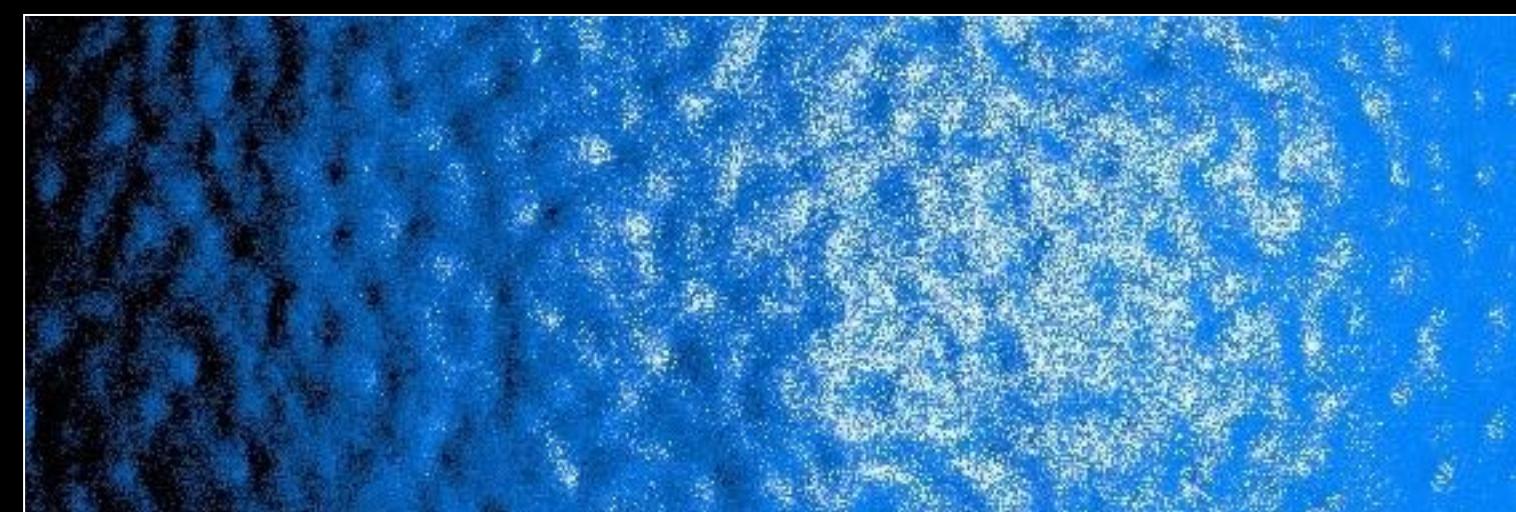
Shading



Green diffuse



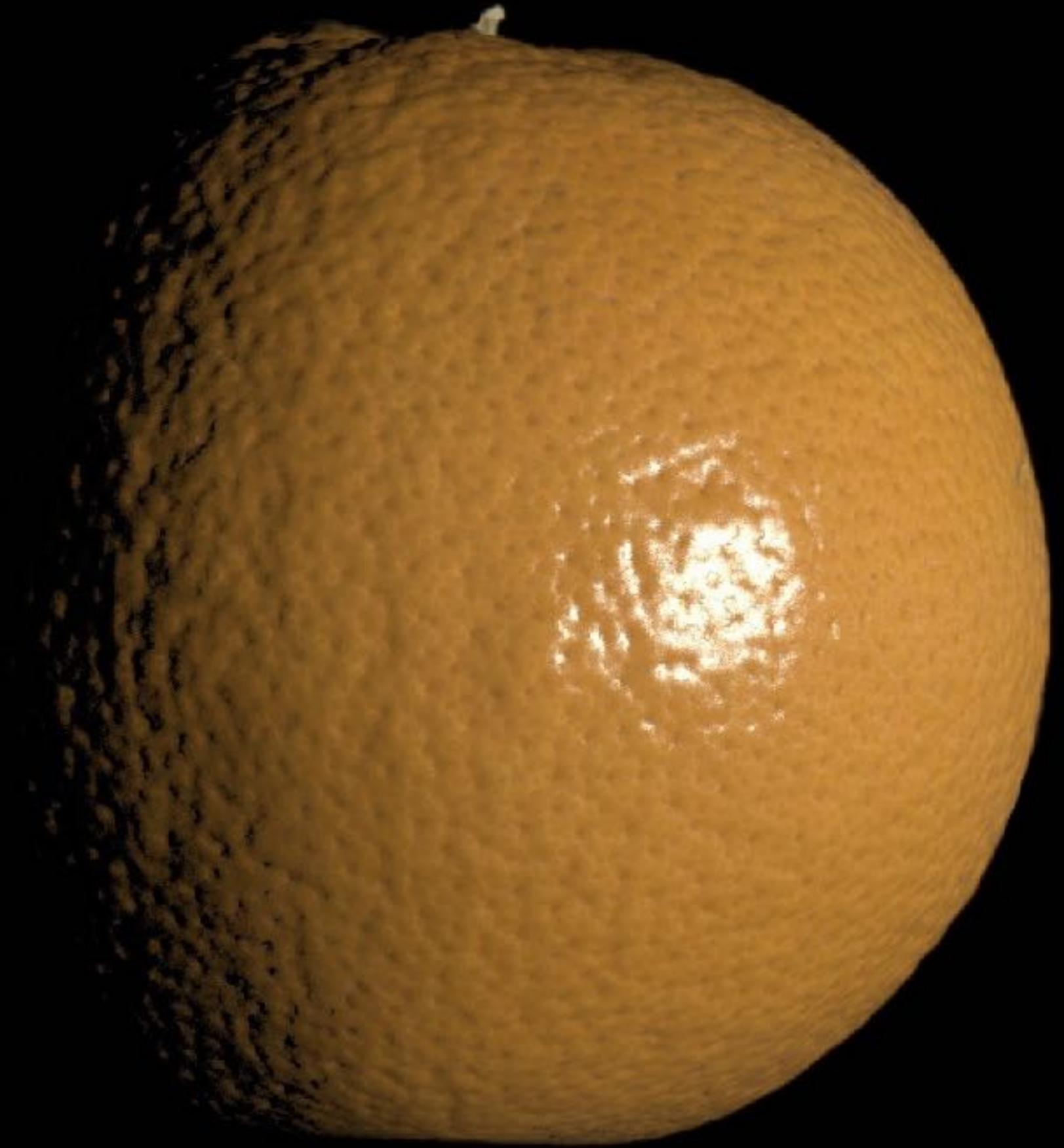
Blue diffuse



Specular



Render with Diffuse Normals

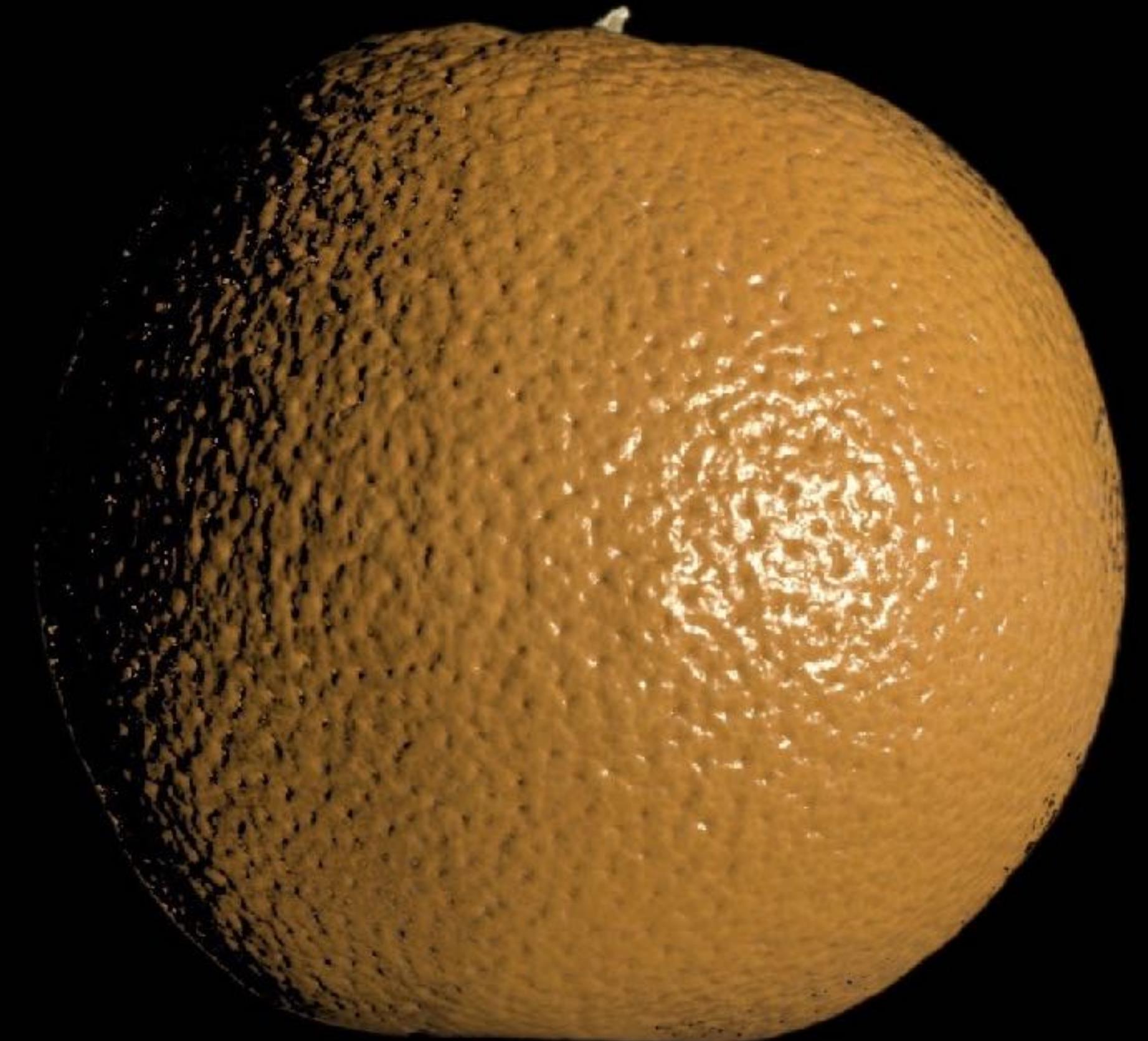


Diffuse

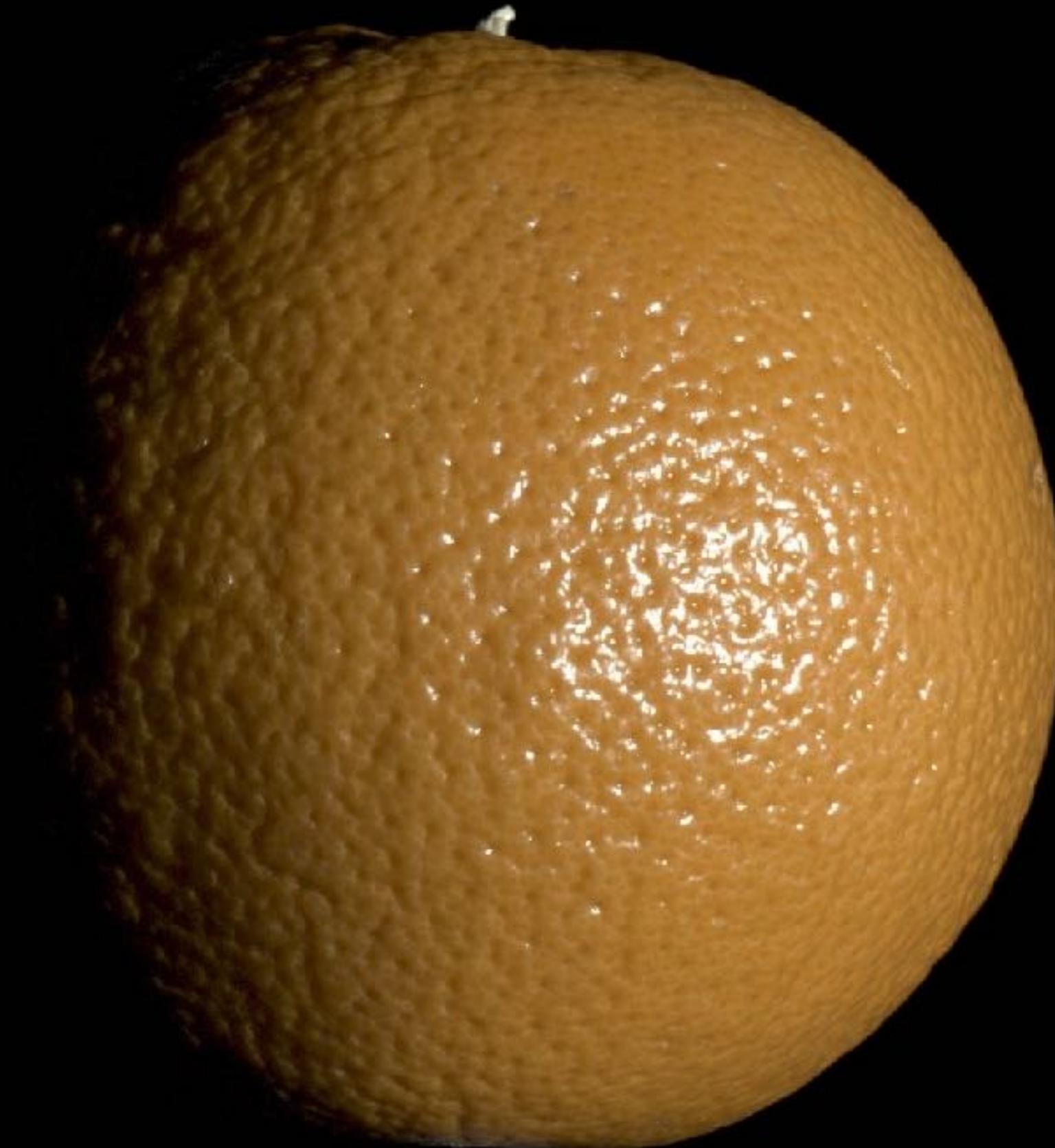


Real Photo

Render with Specular Normals

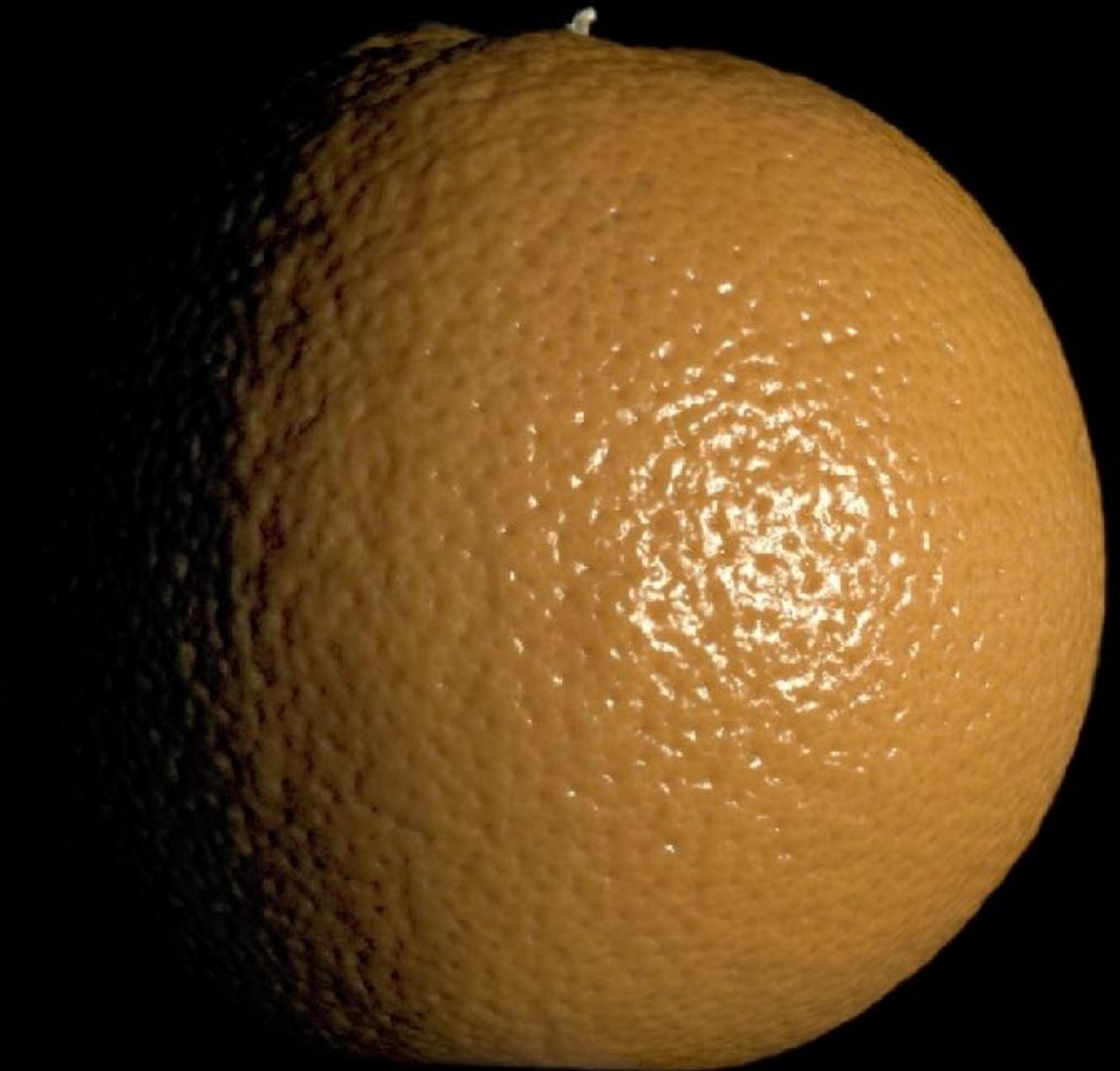


Specular



Real Photo

Render with Hybrid Normals



Hybrid



Real Photo