

Shader Programming

Game Graphics

王銓彰

墨匠科技 BlackSmith CEO

cwang001@mac.com

kevin.cwang3@mac.com

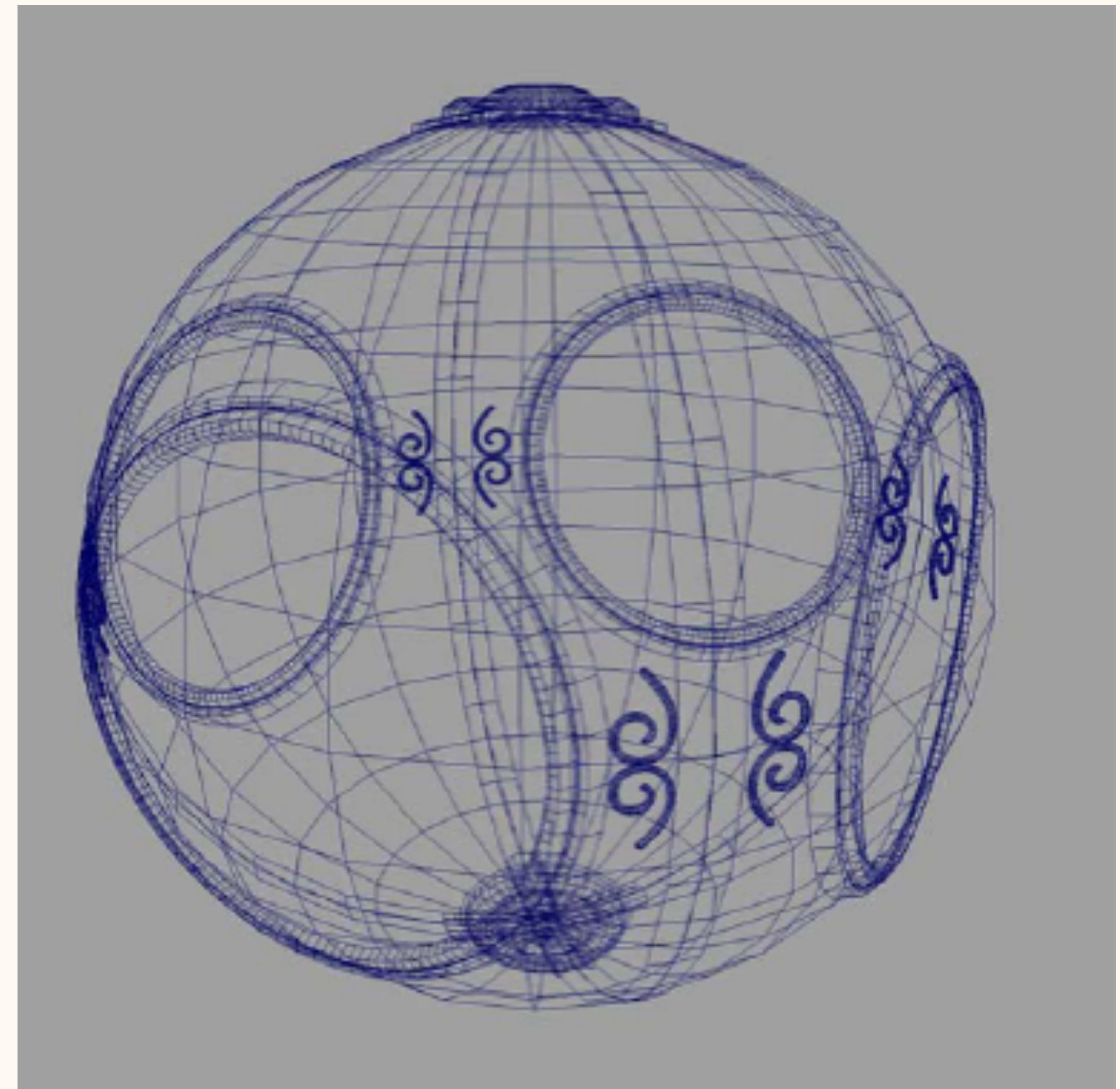
cwang001@blacksmis.com

Contents

- Introduction
- Basics in Graphics
- Real-time 3D Rendering Pipeline
- Shading Programming Basics
- Phong Shading & Phong Reflection Model
- Normal Map
- High Dynamic Range (HDR)
- Reflection and Refraction
- Real-time Image-based Lighting
- Ambient Occlusion
- Shadow Map
- Non-photorealistic Rendering
- Skin Rendering

Computer Graphics History : 1960-1970

- Wireframe graphics
- Project Sketchpad
- Display processors
- Storage tube



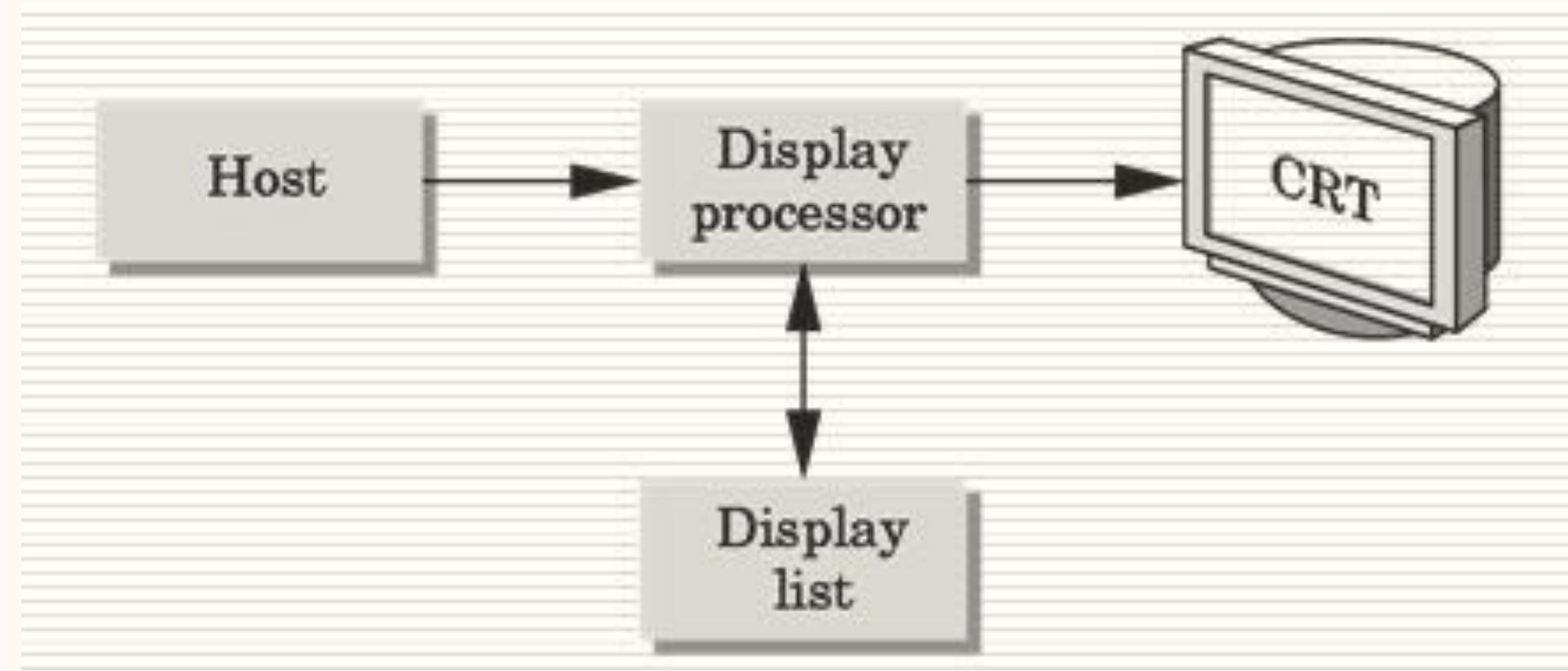
Project Sketchpad

- Project Sketchpad
 - Ivan Sutherland's PhD thesis at MIT
 - "Father of Computer Graphics"
 - Recognized the potential of man-machine interaction
- Loop
 - Display something
 - User moves light pen
 - Computer generates new display
- Sutherland also created many of the now common algorithms for computer graphics
 - Most famous one : 3D clipping



Project Sketchpad

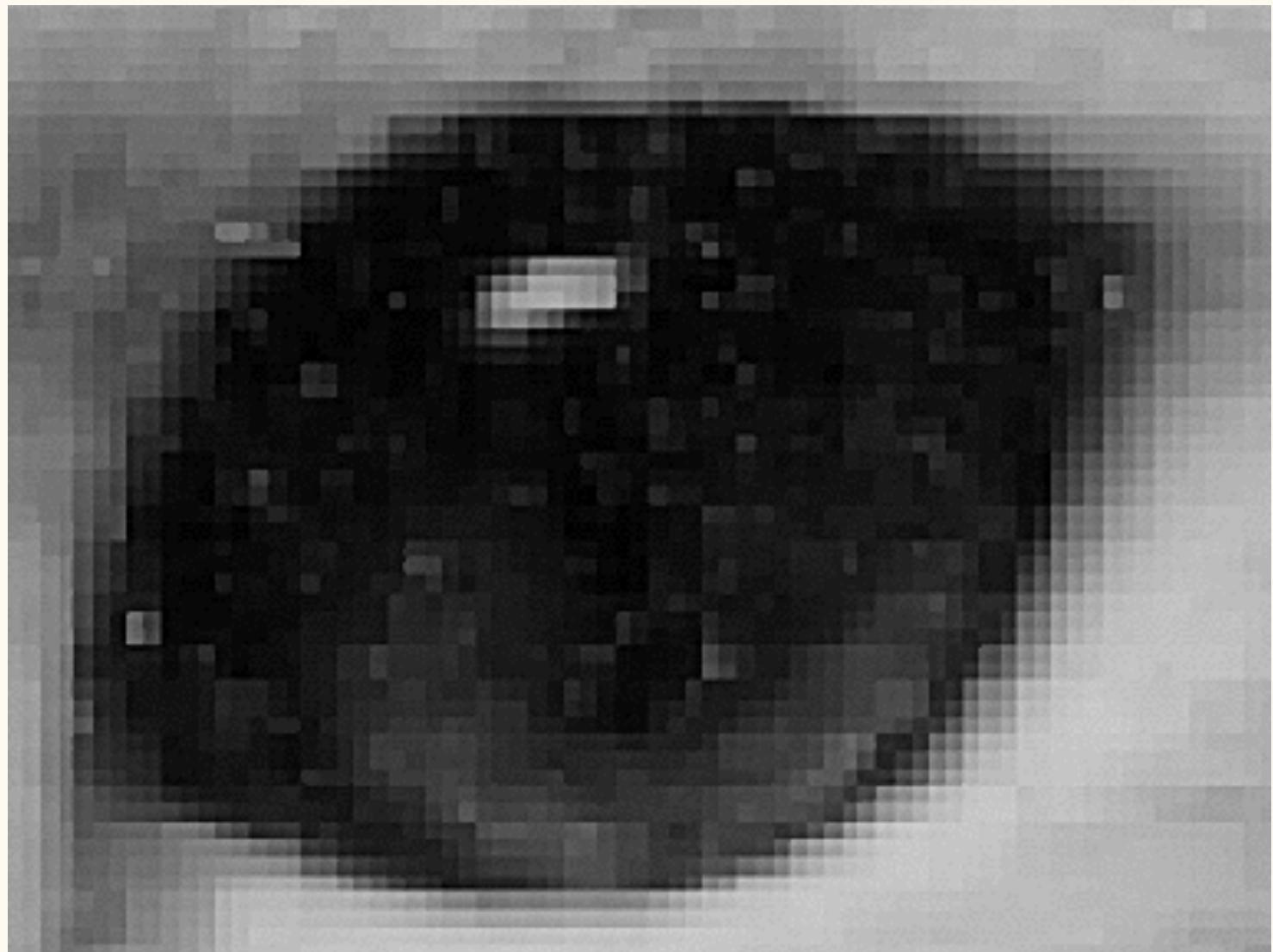
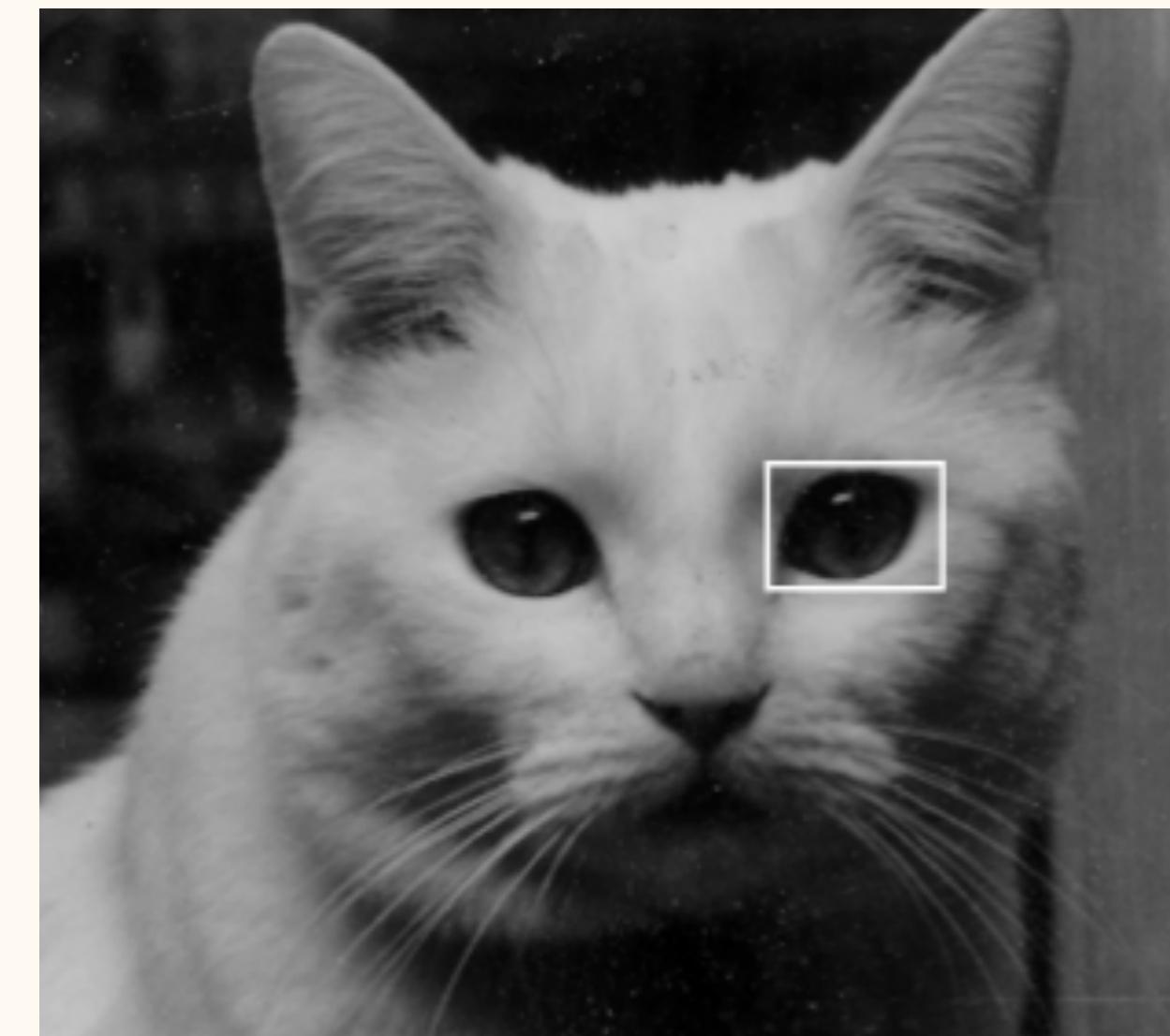
- Display processor
 - Rather than have host computer try to refresh display use a special purpose computer called a **display processor** (DPU)



- Graphics stored in display list on display processor.
- Host compiles display list and sends it to DPU.
- Open a door to use of computer graphics for **CAD** community

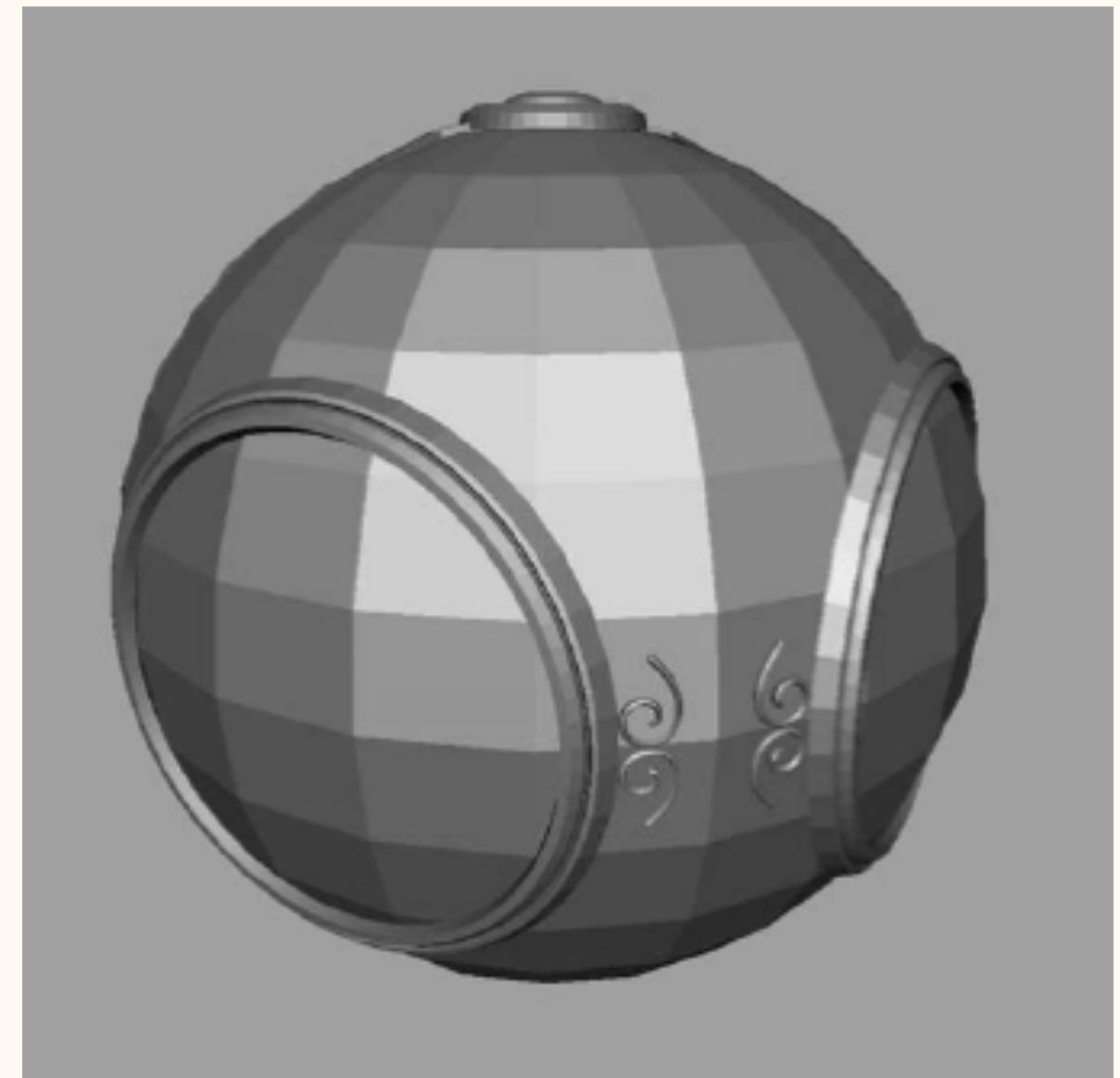
Computer Graphics History : 1970-1980

- Raster graphics
 - Image produced as an array (the raster) of picture elements (pixels) in frame buffer.
- Workstations and PCs (Apple II)
- 1974 ACM 1st SIGGRAPH Conference



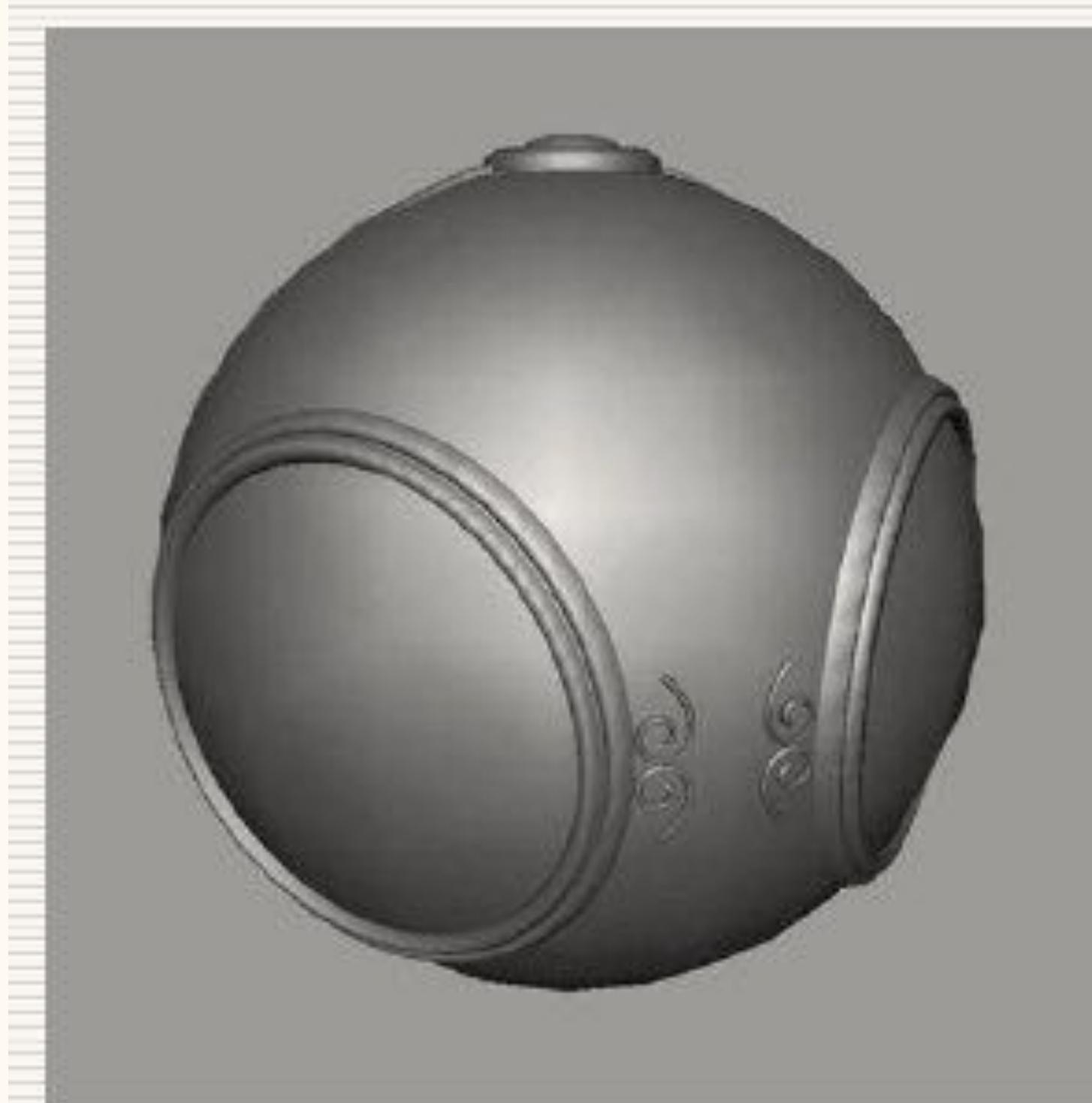
Computer Graphics History : 1970-1980

- Allows us to go from lines and wireframes to filled polygons
- Workstations
 - Although we no longer make the distinction between workstations and PCs historically they evolved from different roots.
 - Early workstations characterized by
 - Networked connection : client-server
 - High-level of interactivity
 - UNIX-based
 - Graphics-enhanced



Computer Graphics History : 1980-1990

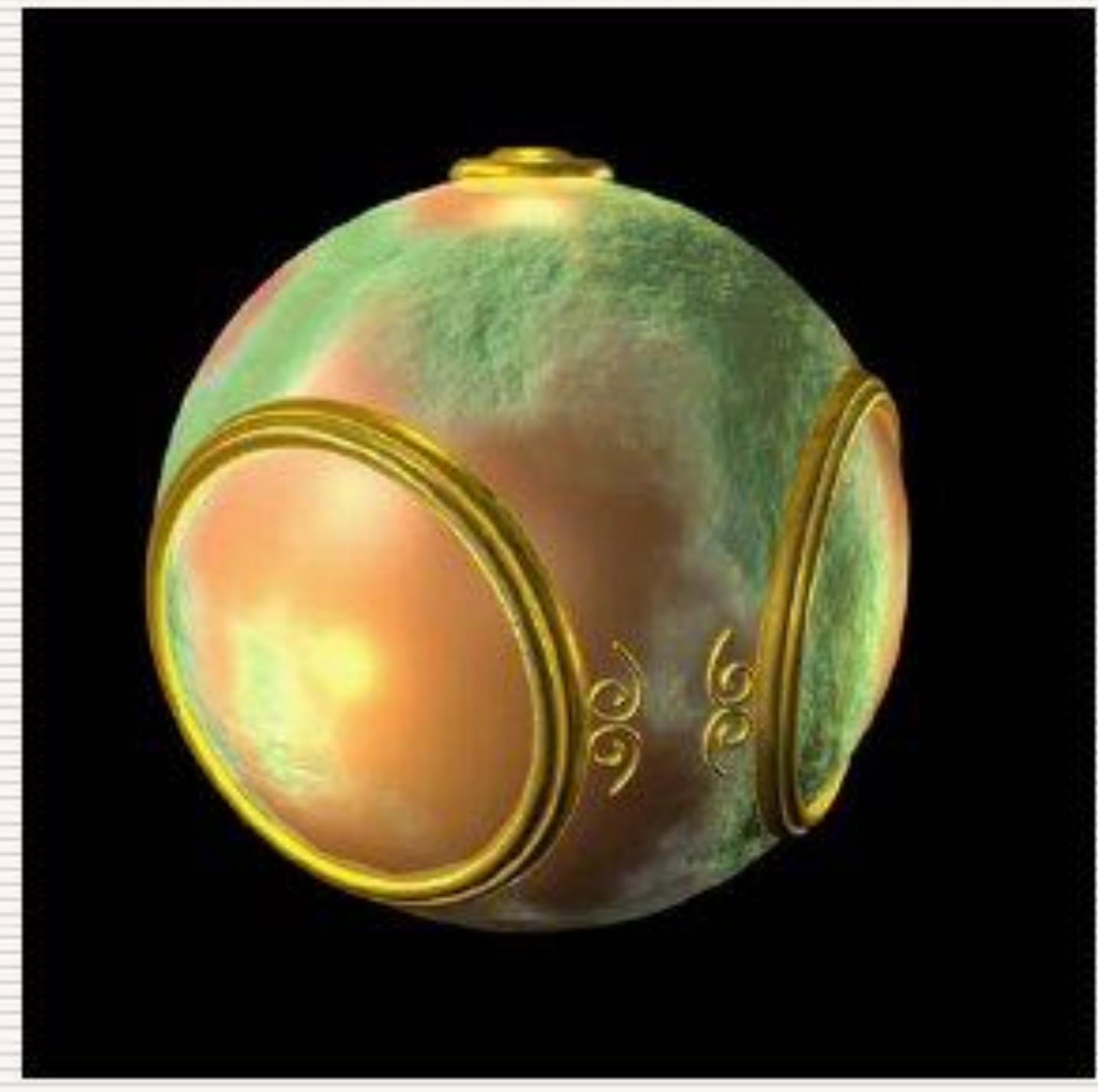
- Realism comes to computer graphics



smooth shading



environmental
mapping



bump mapping

Computer Graphics History : 1980-1990

- New research Topics in this era :
 - Human-Computer Interface (HCI)
 - Scientific Visualization
 - Virtual Reality

Computer Graphics History : 1990-2000

- OpenGL API
 - From SGI Graphics Library (Iris GL)
- Completely computer-generated feature-length movies are successful.
 - Pixar Toy Story
- Visual effects in films
 - ILM's Terminator 2
- New hardware capabilities : NVIDIA NV-1
 - Texture mapping/blending
- PC with 3D graphics card
 - 3D games
- Most fast growth decade for Computer Graphics

Computer Graphics History : 2000-2010

- Major impacts on hardware development :
 - GPU
 - Programmable rendering pipeline
 - Shader programming
 - Multi-core CPU
 - Multithreading
- Photorealism
 - In real-time : (30fps)
 - In high-definition (HD) : 720p = 1280x720
- Graphics cards (with GPU) dominate the market
 - nVidia, AMD/ATI, Intel
- Game boxes and game players determinate direction of market

Computer Graphics History : 2000-2010

- Computer graphics routine in movie industry : Maya
- Computer graphics routine in game production industry : 3dsMax

Computer Graphics History : 2010-2020

- The 4th 15-year of Computer Graphics
- Running 3D on mobile devices + Computing on cloud
 - CPU + GPU
 - 3G or higher networking (3G->4G->5G)
 - Sensors
 - Android ADK/ iOS SDK
 - OpenGL ES
- Middleware
 - Unity
 - Unreal
- XR = MR/VR/AR

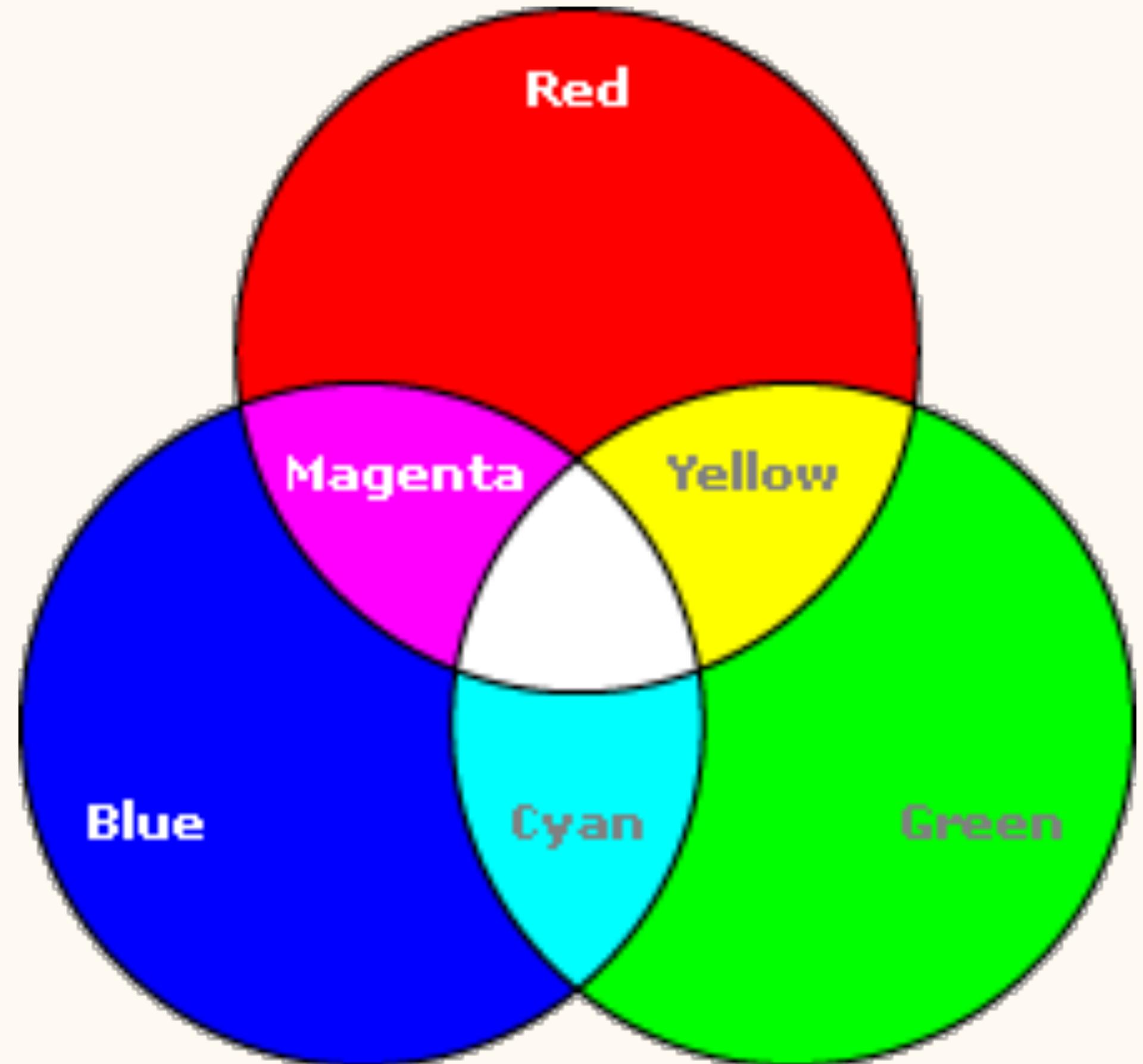
Basic

Color Model (色彩模型)

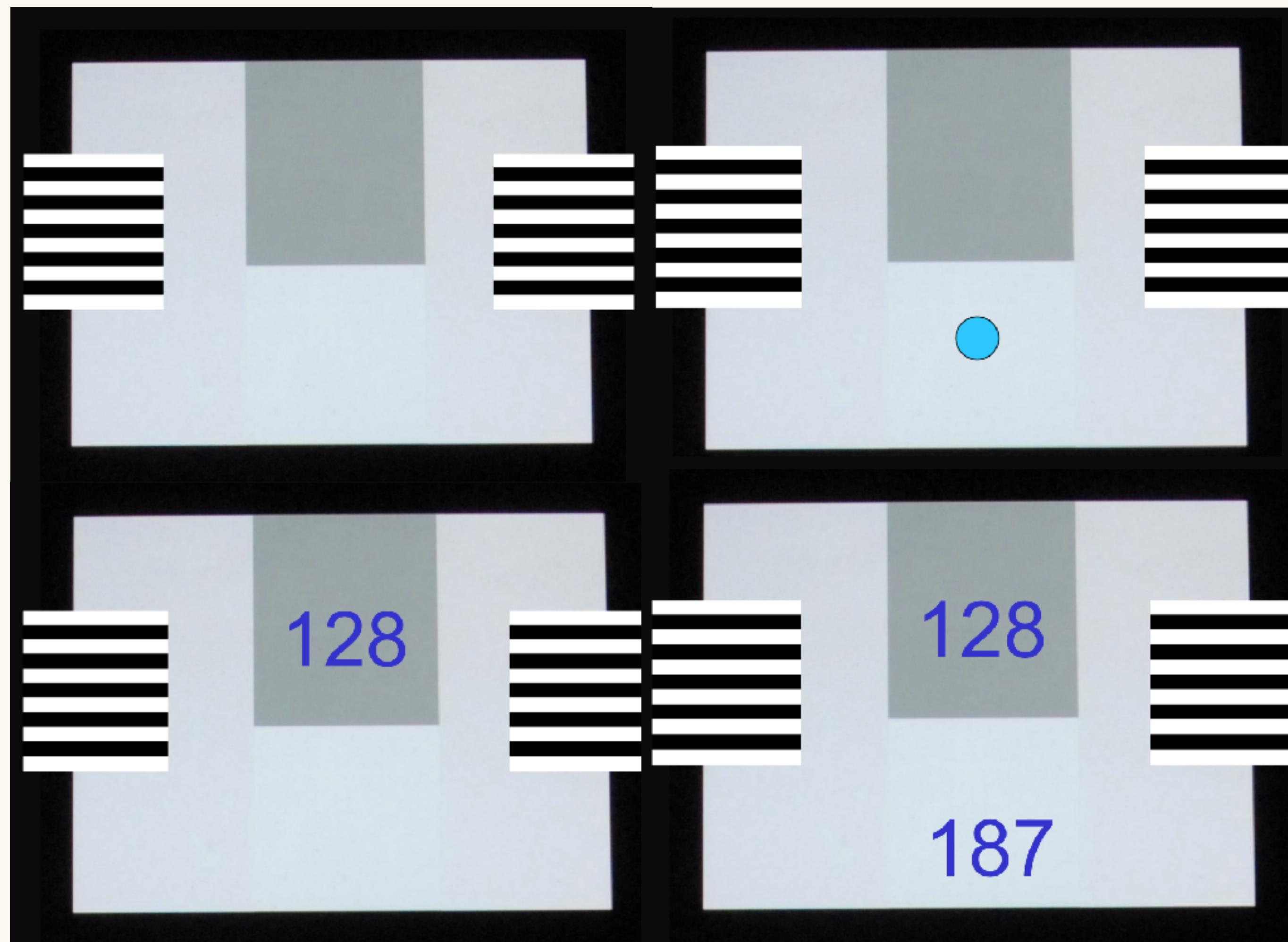
- 色彩模型 (Color Model) , 也稱為色彩空間 (Color Space)
 - RGB color model
 - HSV color model
 - CMYK color model
 - CIE Lab color model
- RGB color for display
 - 24 bit color : (R8, G8, B8)
 - 256*256*256 colors
 - True color mode
- We map the color to (0.0 ~ 1.0) in game technologies

RGB Color Model

- Most digital images are defined in red-green-blue (RGB) color model
- Range from 0.0 - 1.0 for each component
- Relative Value
 - Black = (0, 0, 0)
 - White = (1, 1, 1)
- Alpha channel
 - Opacity
- Color vector : (r, g, b, a)

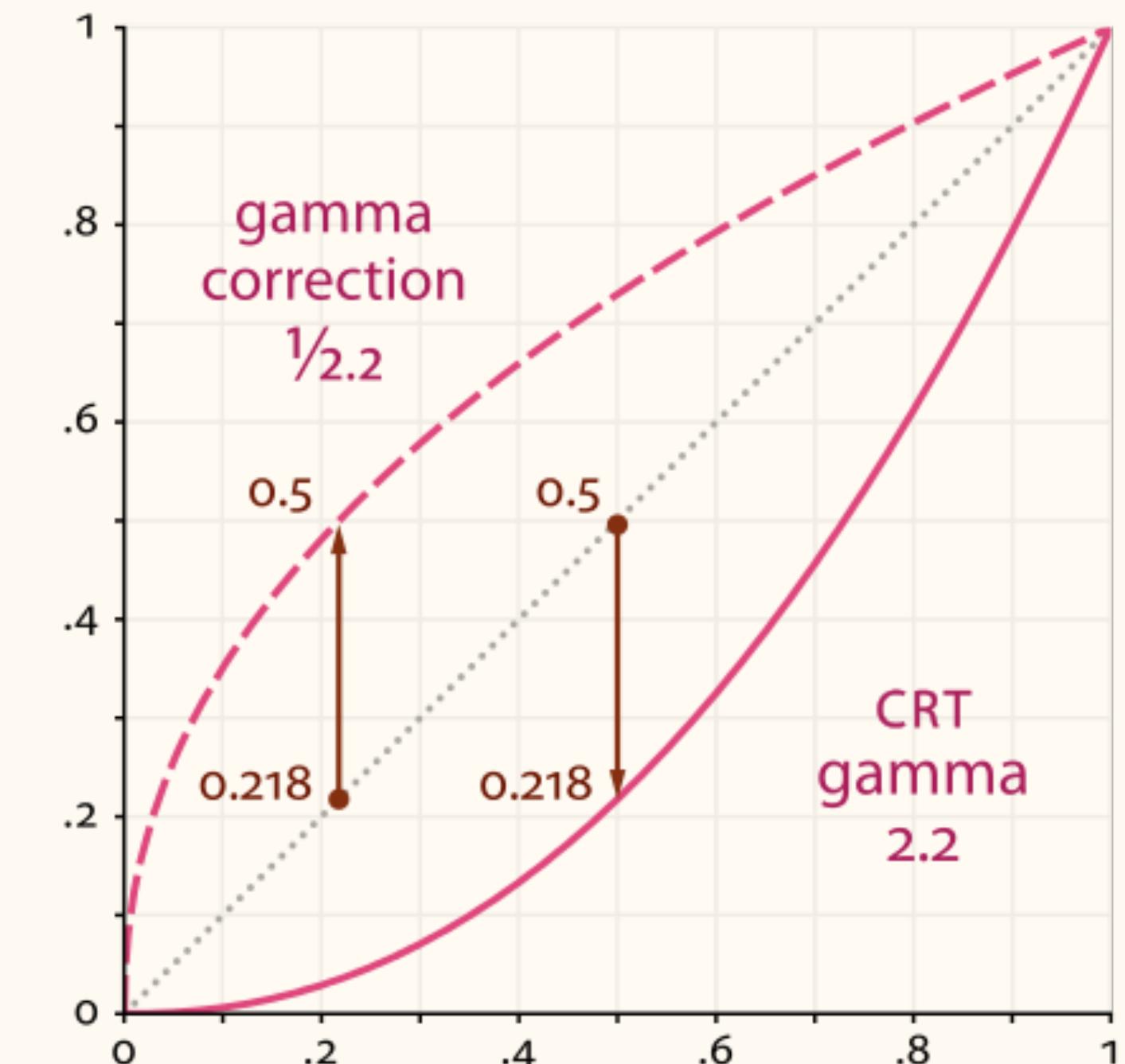


Gray Level Test

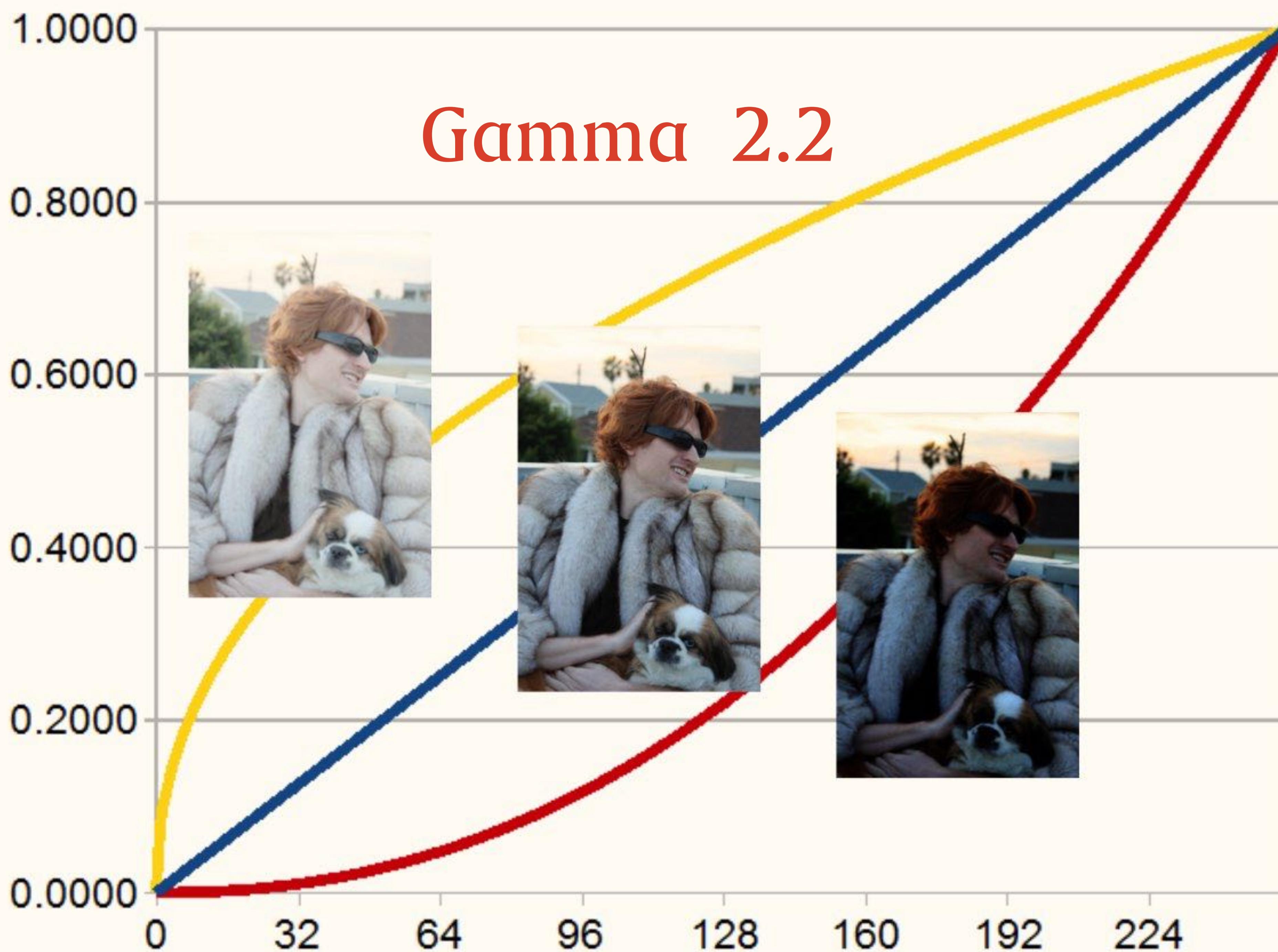


Gamma Correction

- Gamma nonlinearity, gamma encoding
- Used to optimize the usage of bits when encoding an image by taking advantages of the non-linear manner in which human perceive light and color
- Human vision, under common illumination conditions, follows an approximate gamma or power function, with greater sensitivity to relative differences between darker tones than between lighter ones
- $V_{\text{out}} = A * V_{\text{in}}^{\gamma}$
 - $A = 1$ in common
 - $\gamma = 2.2$

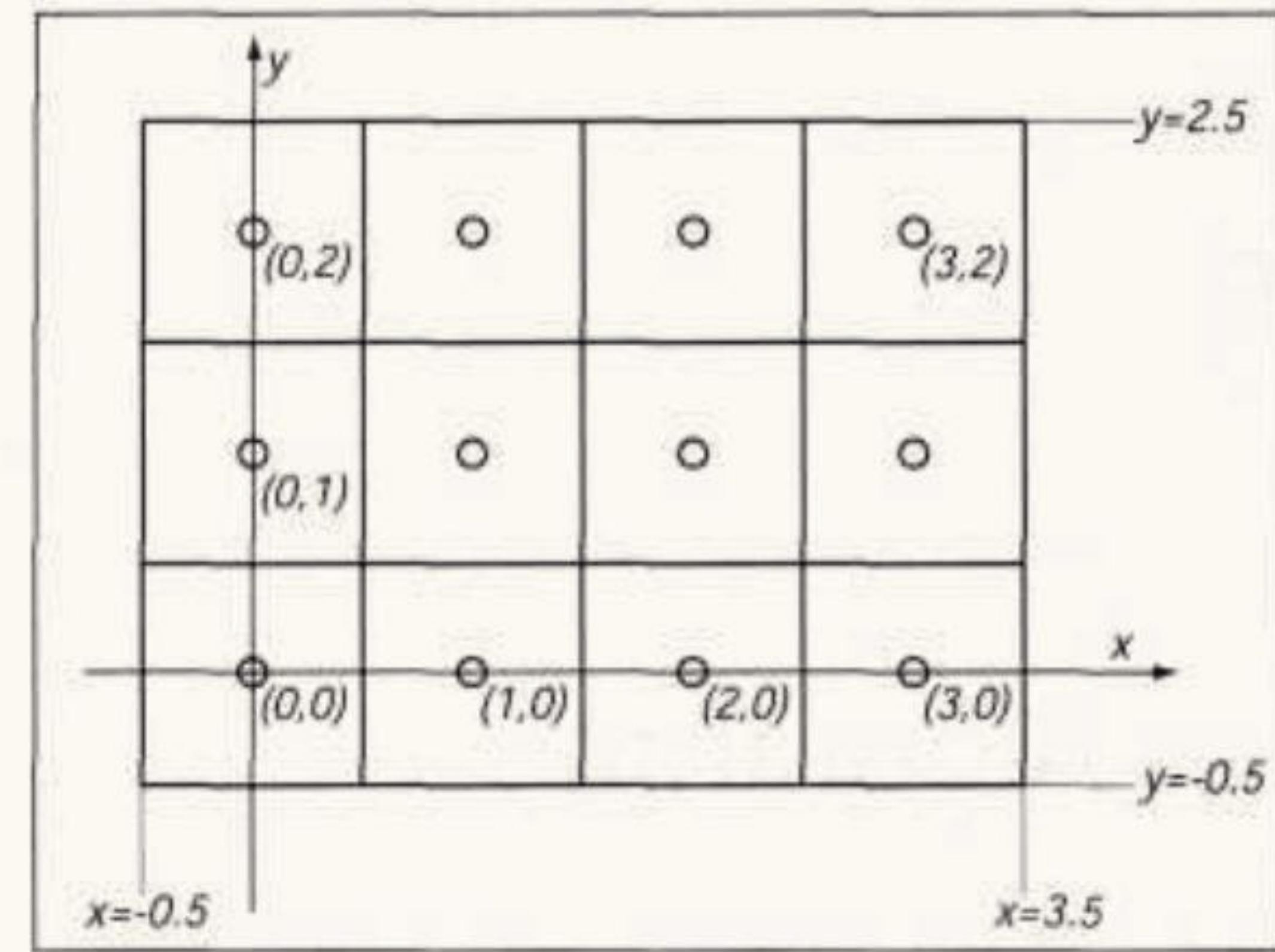


Gamma 2.2



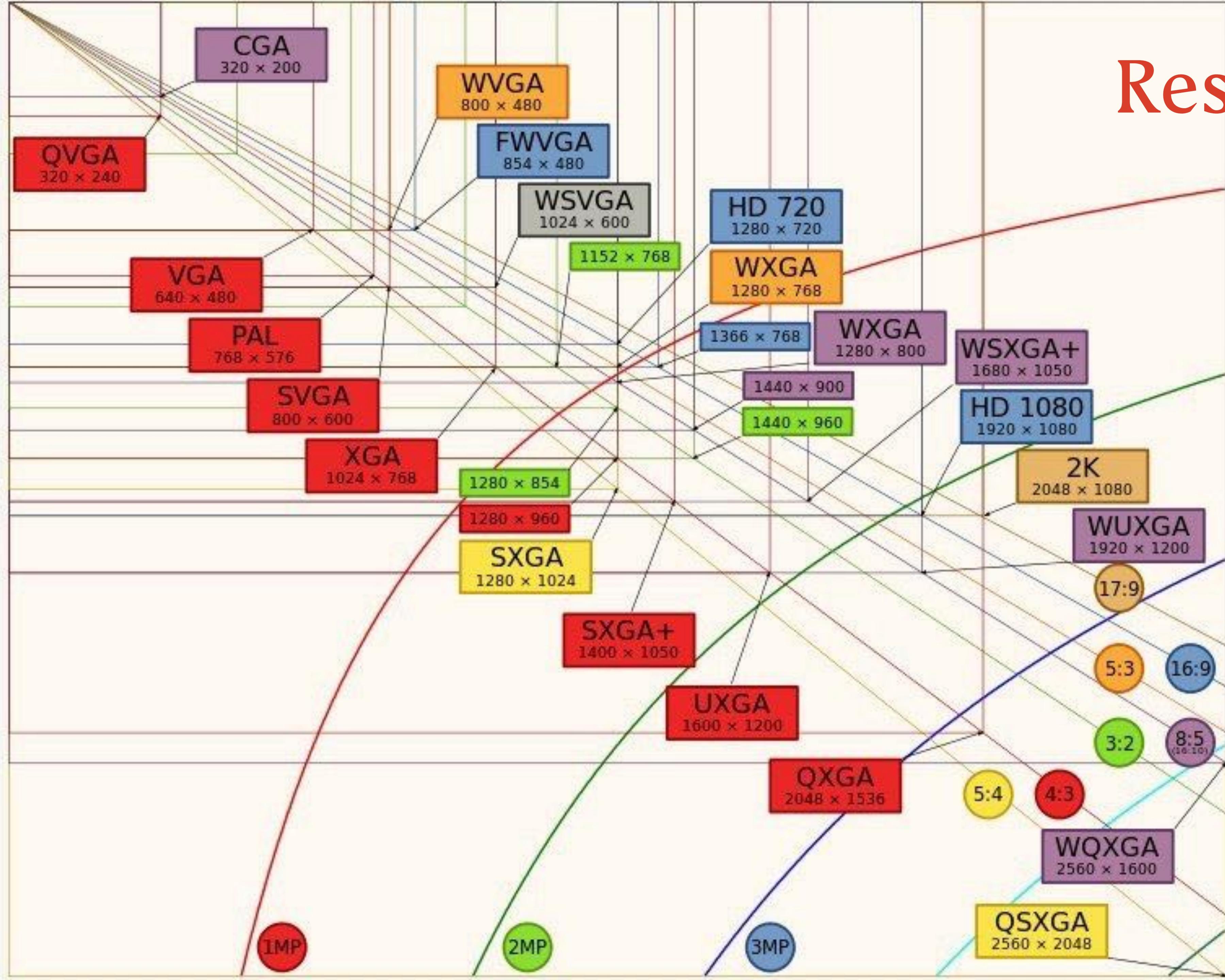
Raster Display

- Pixels
 - “Picture elements” in short
- Resolution
 - Number of pixels
- Coordinates
 - Alignment issue
 - Aligned on pixel center
 - Aligned on pixel edge
 - 2D screen coordinates
 - Left-hand
 - Right-hand



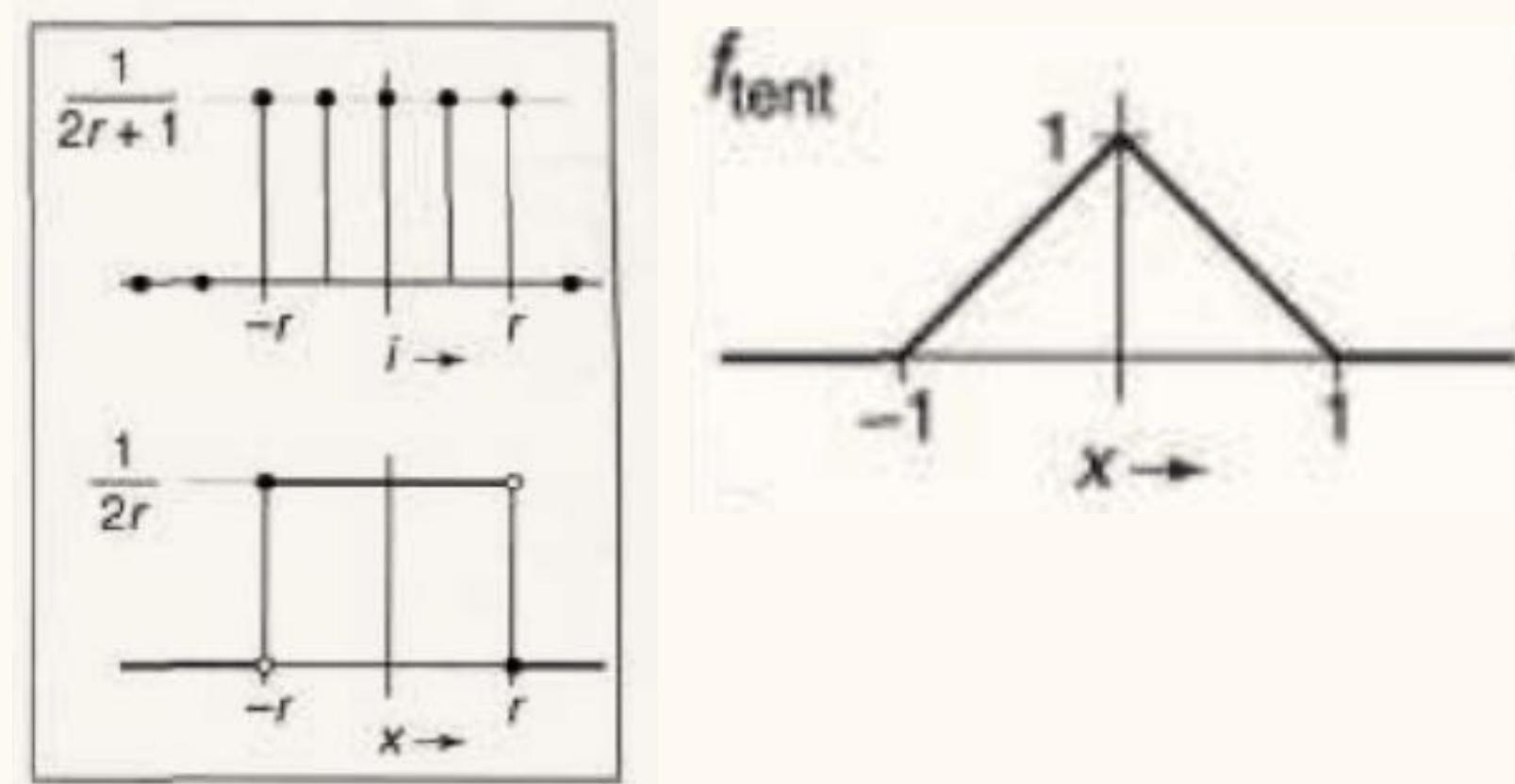
Pixel is aligned on pixel center
Right-hand

Resolution



Aliasing & Anti-aliasing

- One problem in drawing on raster display is the jaggy appearance
 - Aliasing problem
 - Use a filter function to average color of the regions of drawing
 - Antialiasing
 - For example, the “box filter”, the “tent filter”

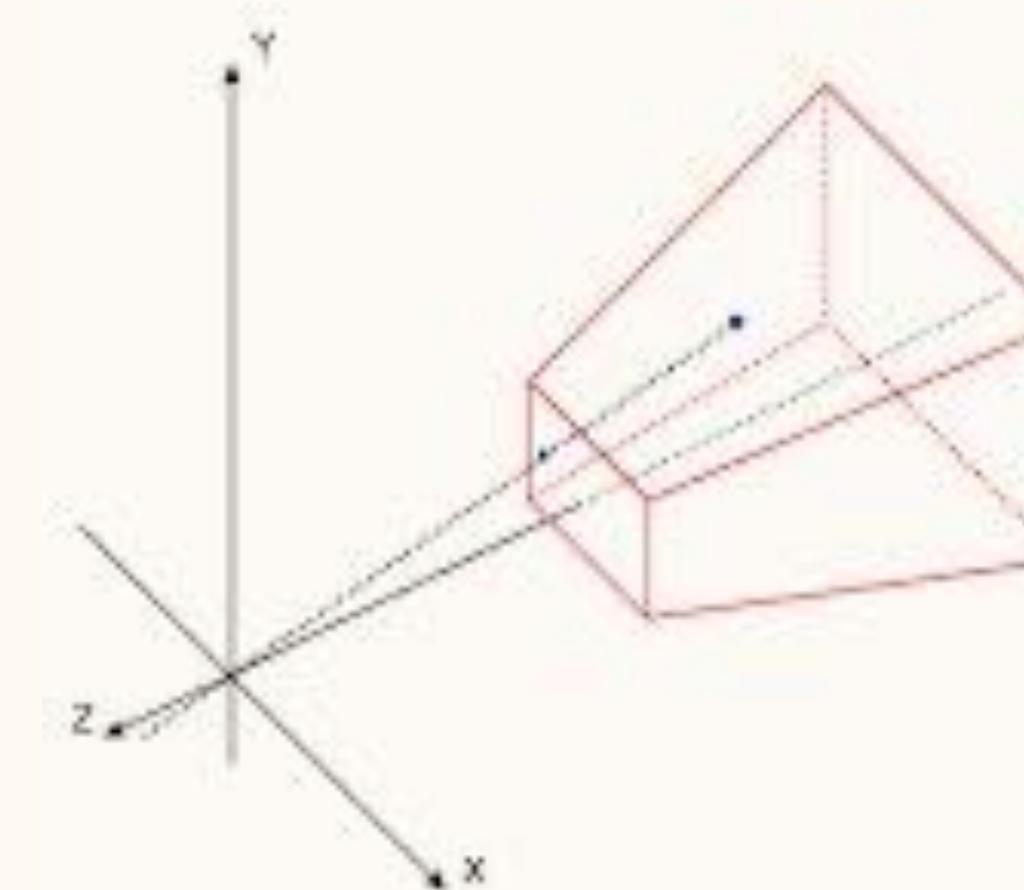


Realtime Computer Graphics

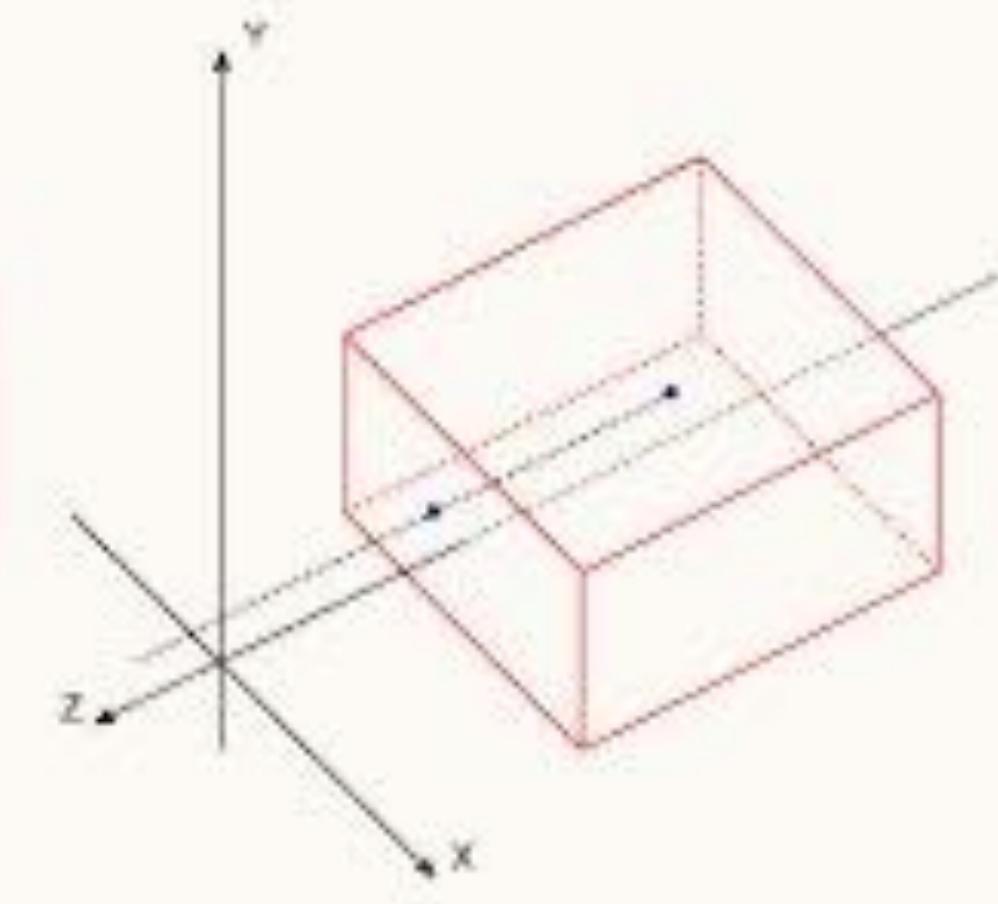
- 3D game graphics
 - 3D computer graphics for games
- Features
 - In real-time performance
 - 30 frame-per-second (fps) or up
 - GPU
 - Triangles mostly
 - Multiple textures
 - Running shaders
 - Video memory

2D ?

- No more 2D graphics directly supported by hardware now
 - DirectDraw was expired.
 - DirectX = Direct3D
- 2D = 3D in orthogonal projection view
 - All popular graphics APIs are 3D.
 - Direct3D
 - On PC / Windows Phone
 - OpenGL
 - On PC / Mac / Linux
 - OpenGL ES
 - On Android / iOS / Web (WebGL)



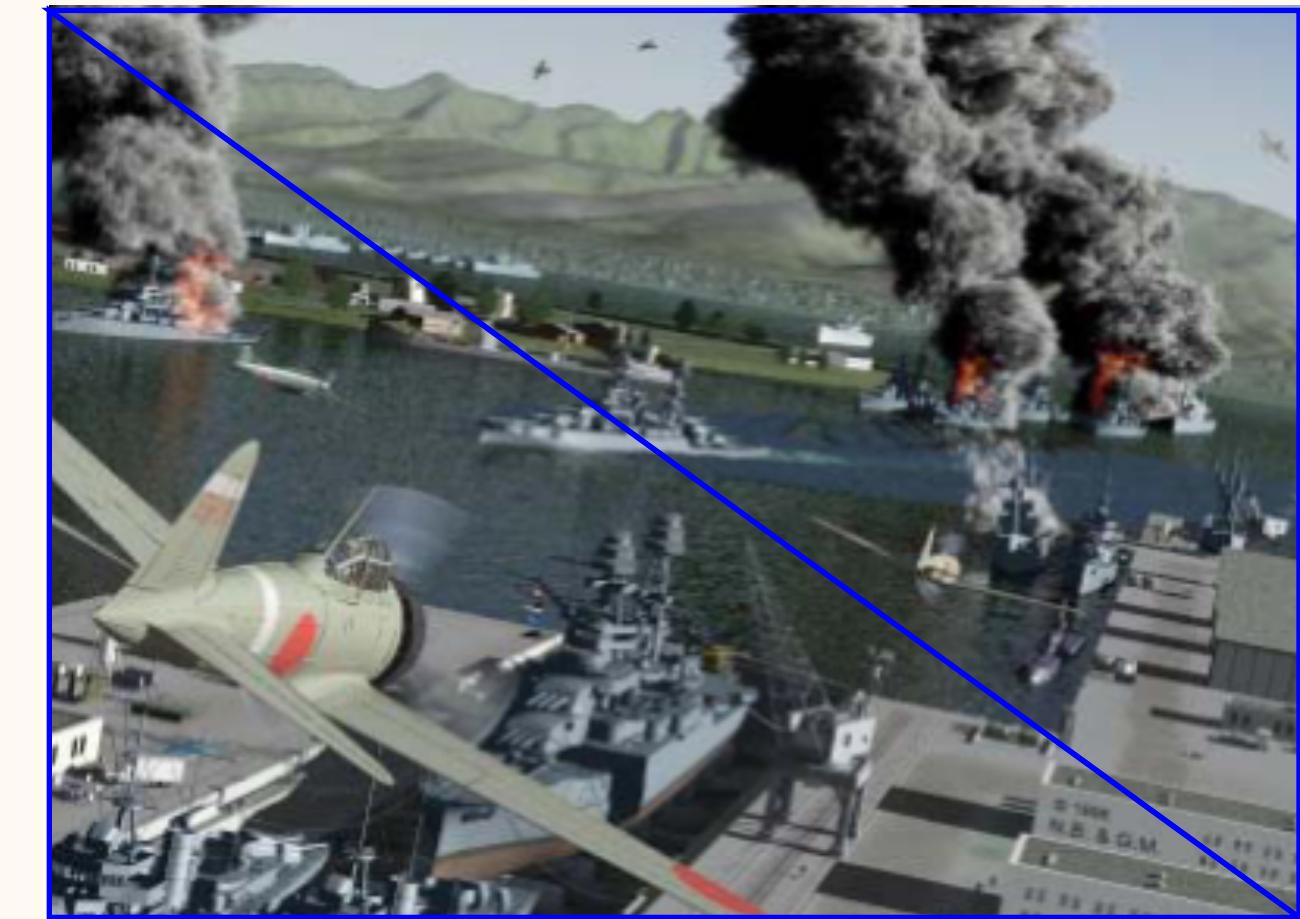
Perspective View



Orthogonal View

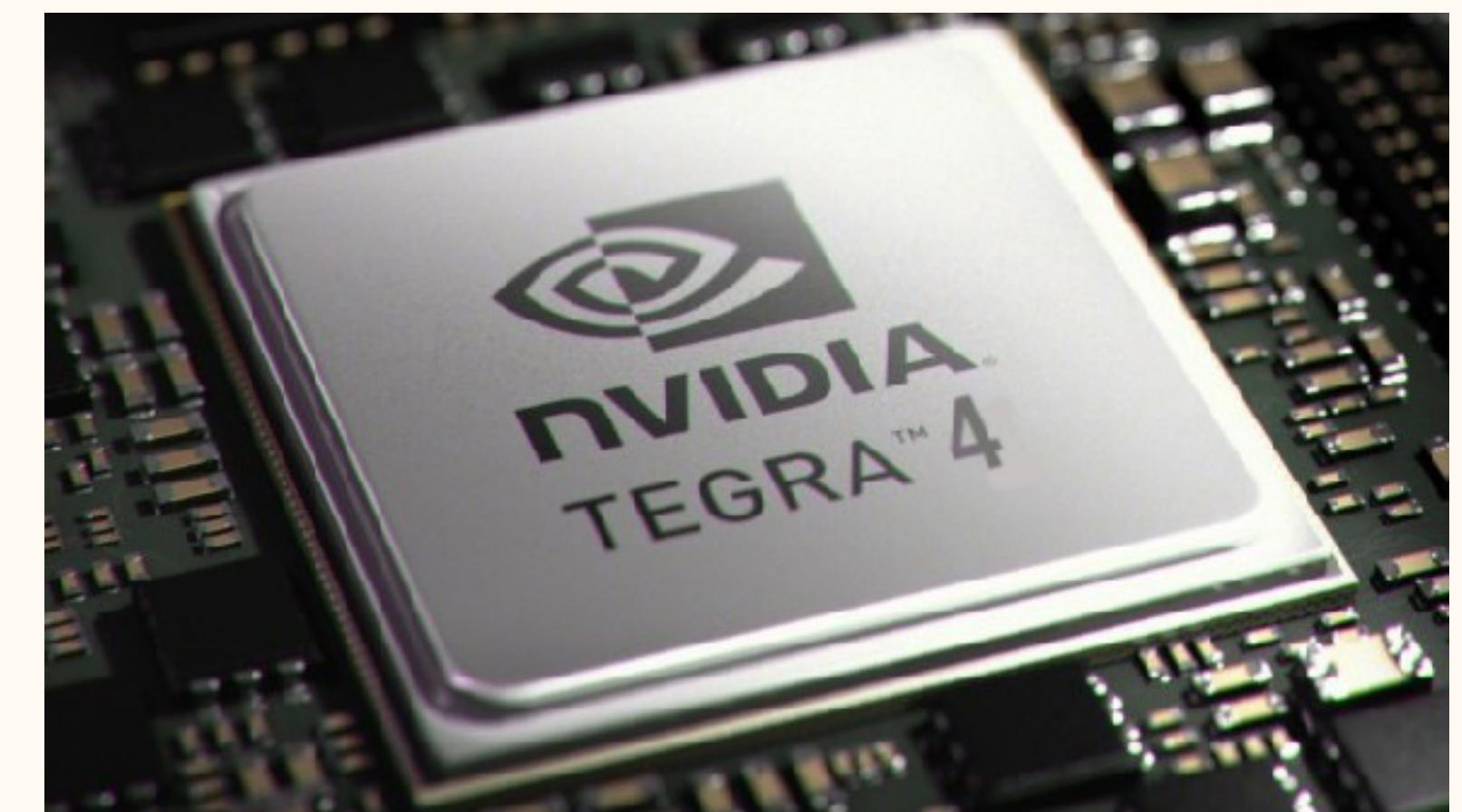
Image ? Texture ?

- 2D = Texture on Two Triangles
- A Texture = An Image + Addressing + Filtering
- No more pure image primitive at recent graphics API
 - Texture surface = raw image data
 - Texture (sampling) = fragment
 - Texture surface data (image data)
 - Filtering
 - The way to sample texels for screen pixels
 - “Texel” is the pixel in a texture
 - Addressing
 - The way to map texture on geometry



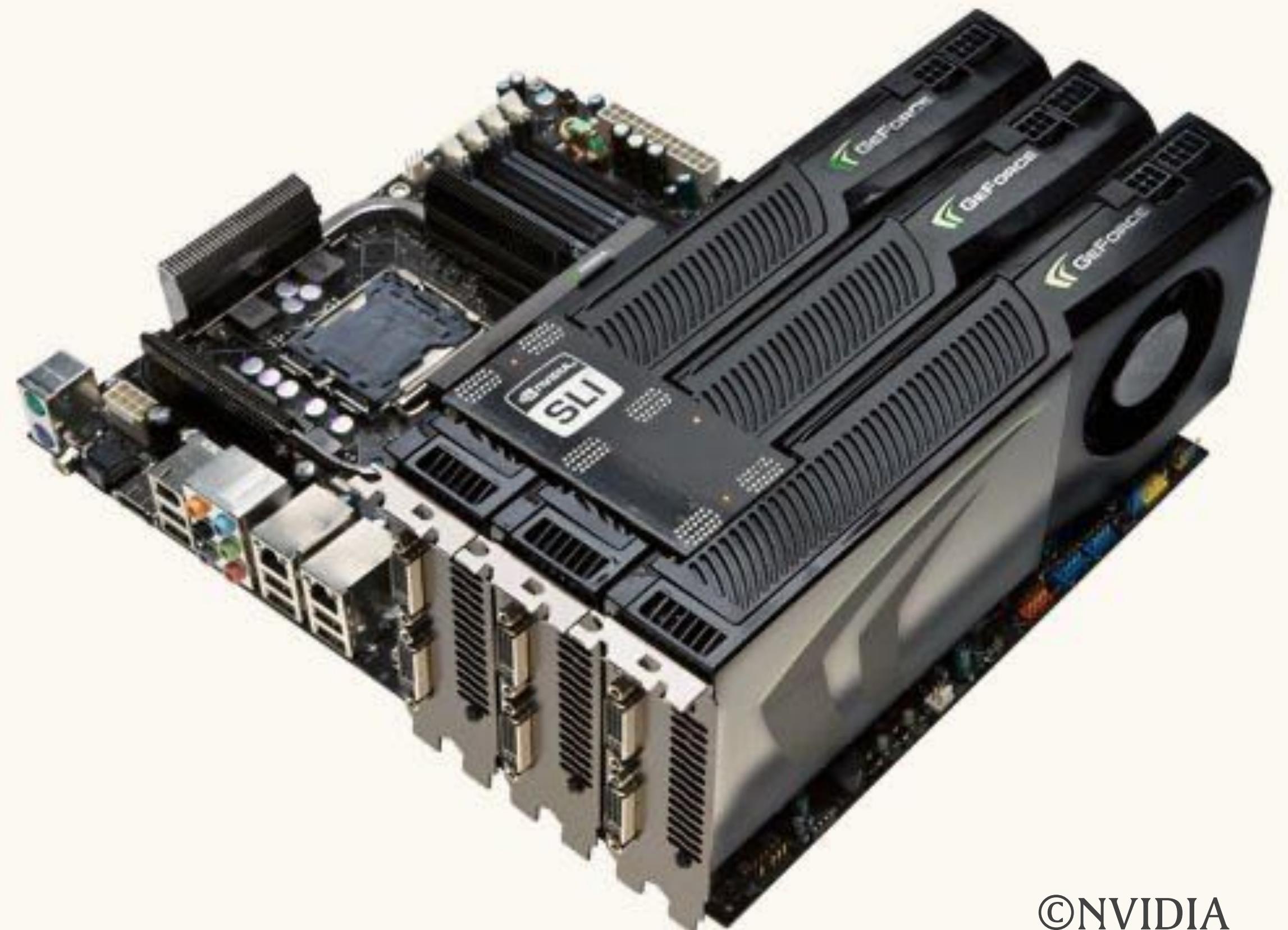
GPU

- 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



GPGPU

- General Purpose GPU Applications
 - Super-computing
 - Programming Tools :
 - nVidia CUDA
 - OpenCL
 - OpenGL Compute Shader
 - DirectX Compute Shader



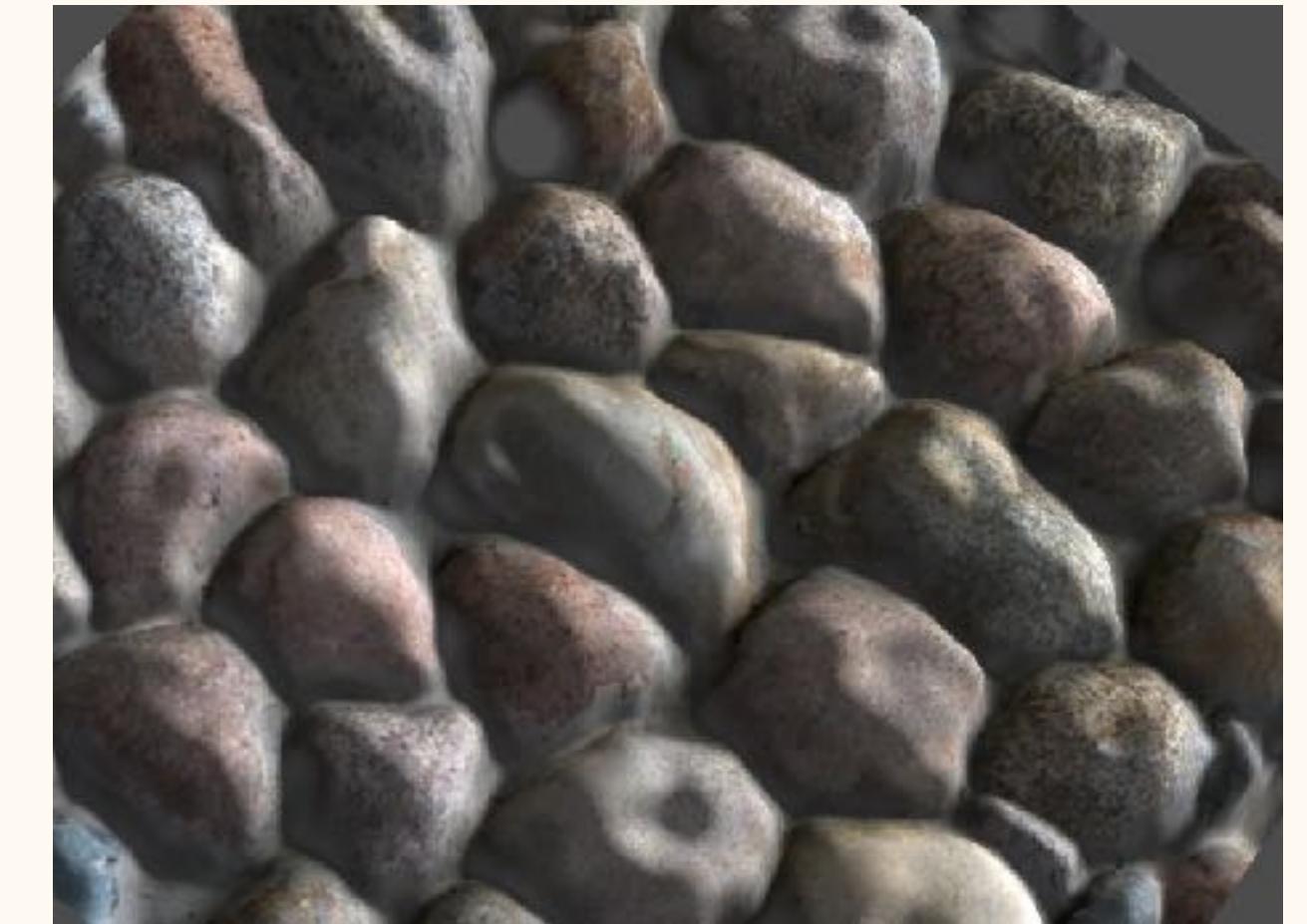
©NVIDIA

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)

Performance : An Example

- Parallax Occlusion Map Shader
 - Mac Book Pro ATI X1600 (XP, 2008) ~ 20 fps
 - Xbox 360 ~ 120 fps (not optimized)
 - 2.67 GHz QuadCore + NV 8800 GTX
 - On Vista ~ 200 fps
 - 2.67 GHz QuadCore + NV GTX280
 - On Vista ~ 300 fps
 - 2.83 GHz QuadCore + 2 x NV GTX280 SLI (not optimized)
 - On Vista ~ 400 fps
 - Mac Book Pro + AMD Radeon HD 6770M (2013)
 - On Windows 7 ~ 560 fps
 - Mac Book Pro + nVidia GeForce GT 750M (2014)
 - On Windows 8.1 ~ 900 fps



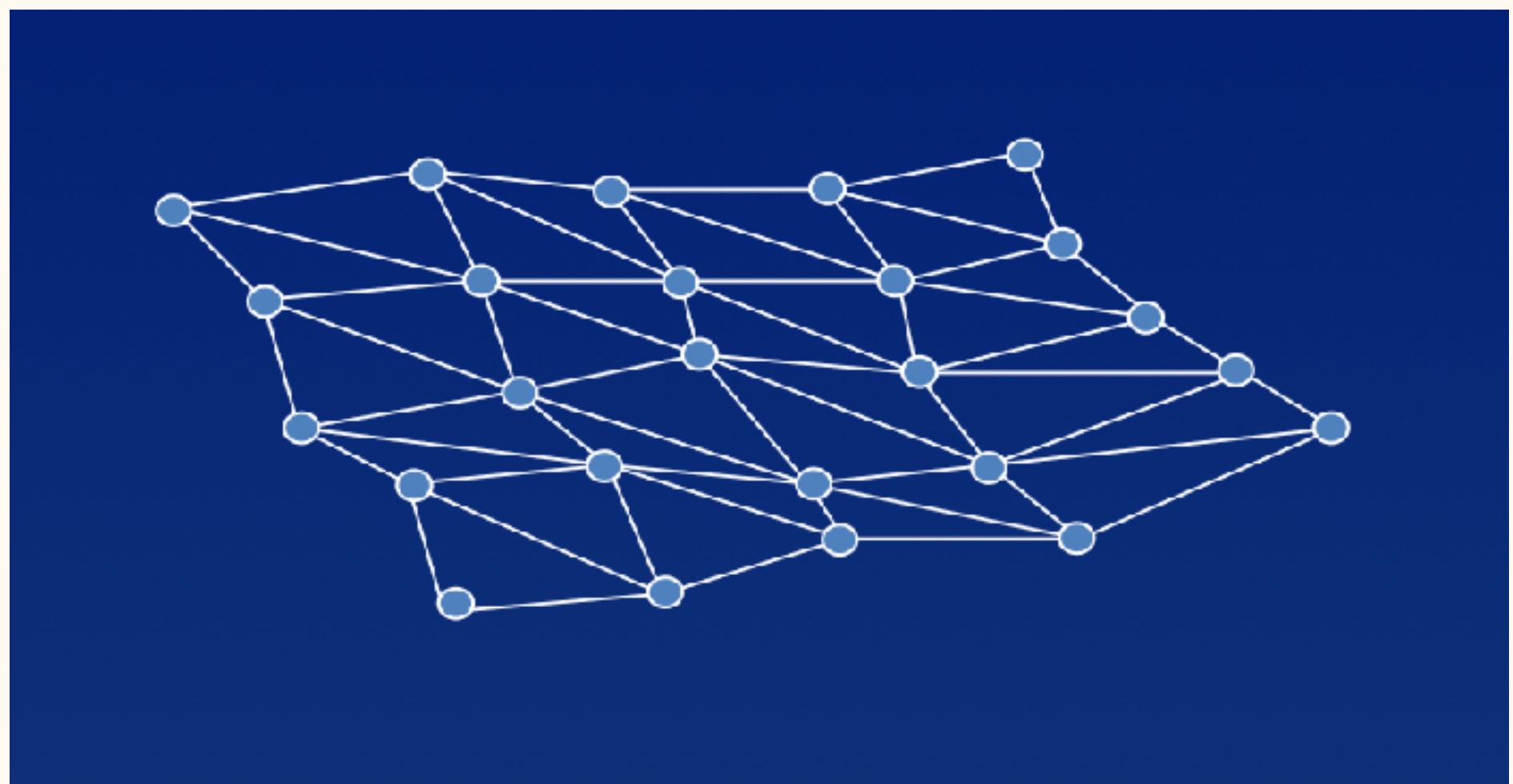
Parallax Occlusion Map

Representation of Objects

- Factors to determine the object representation :
 - Data structure
 - Cost of processing an object (from the view of 3D)
 - Final appearance of an object
 - Ease of editing the shape of the object
- Popular solutions :
 - Polygonal representation
 - Our interests
 - Surfaces
 - Constructive solid geometry (CSG)
 - Subdivision

Polygonal Representation

- Geometry (Vertices)
 - Position
 - Normal vector on vertex
 - Assign color on vertex
 - Texture coordinates
 - Tangent vector on vertex
 - Bi-normal vector
- Primitives (Topology)
 - Triangles
 - Quads
 - Polygons



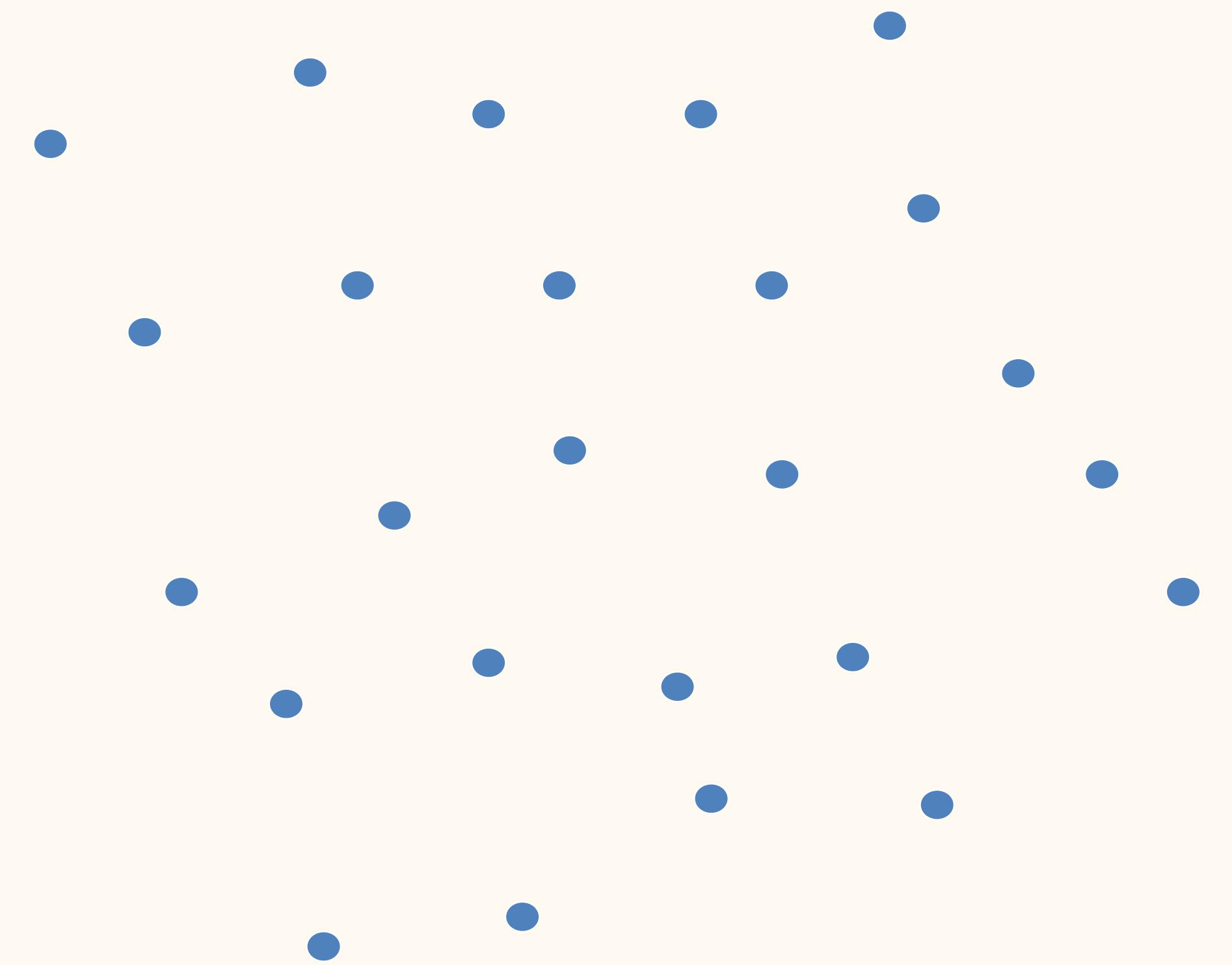
Polygonal Representation

- Surface properties
 - Materials
 - Textures
 - Shaders

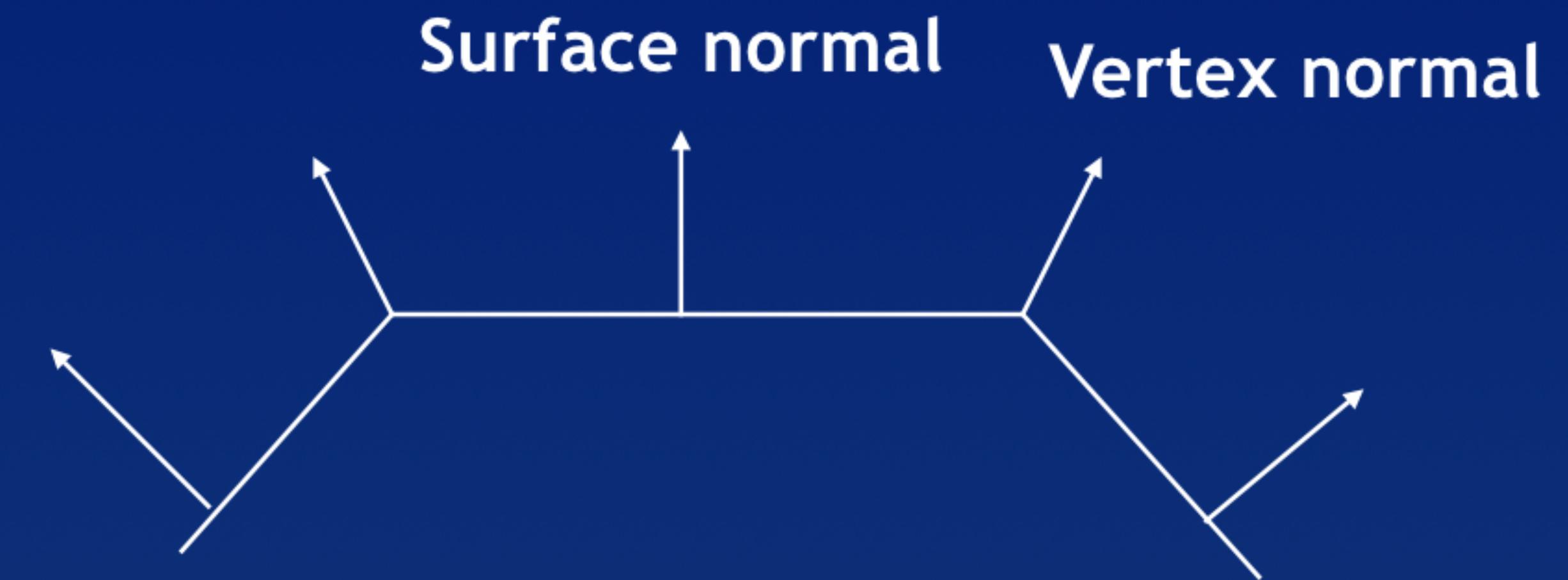
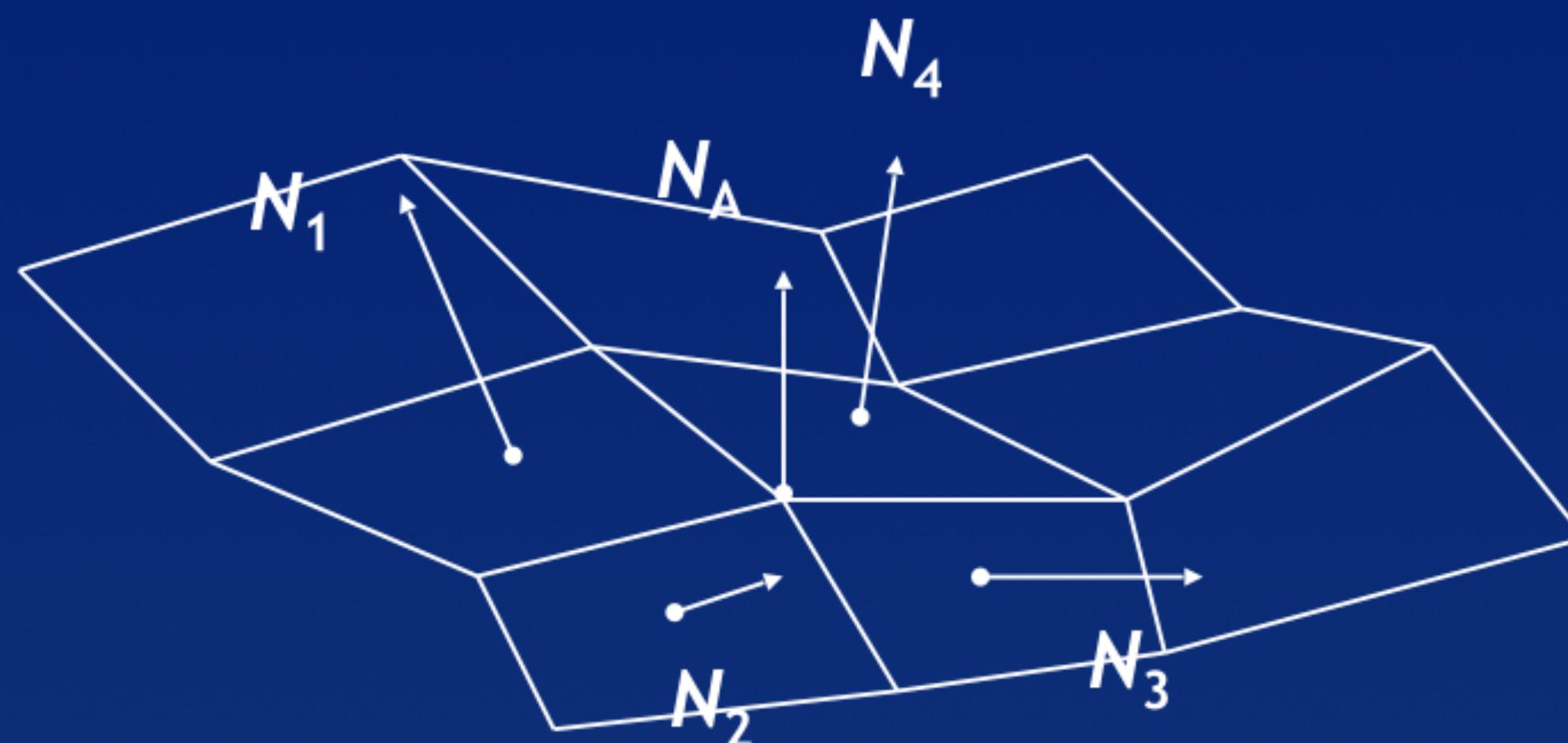


Vertices

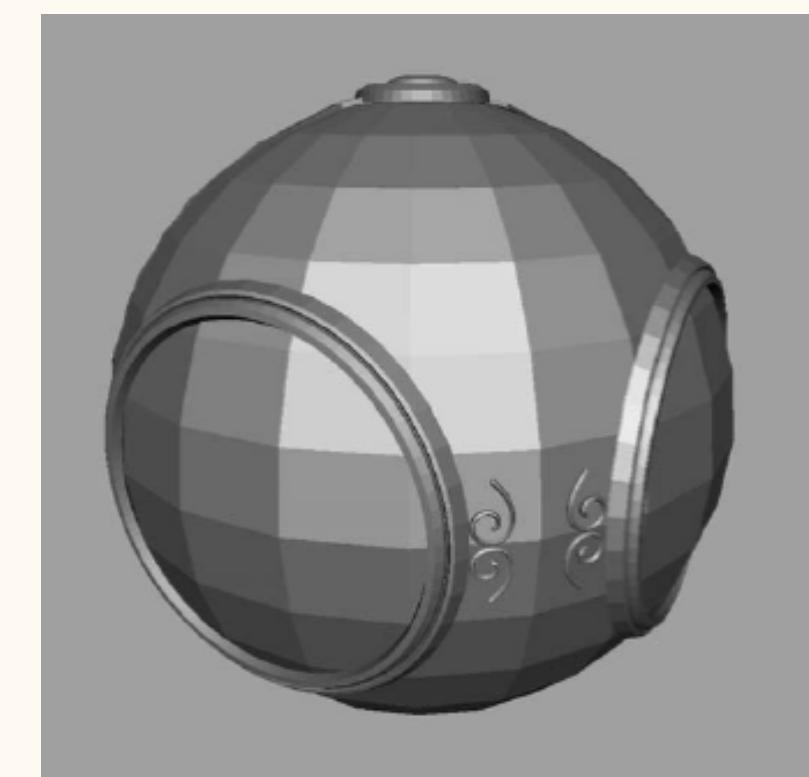
- Geometric data of a model
- In model space
 - Local space
- Vertex Position
 - (x, y, z, w)
 - $w = 1$
 - Homogeneous coordinate
- Vertex normal (N)
 - (n_x, n_y, n_z)
- Texture coordinates
 - $(u_1, v_1), (u_2, v_2), \dots$



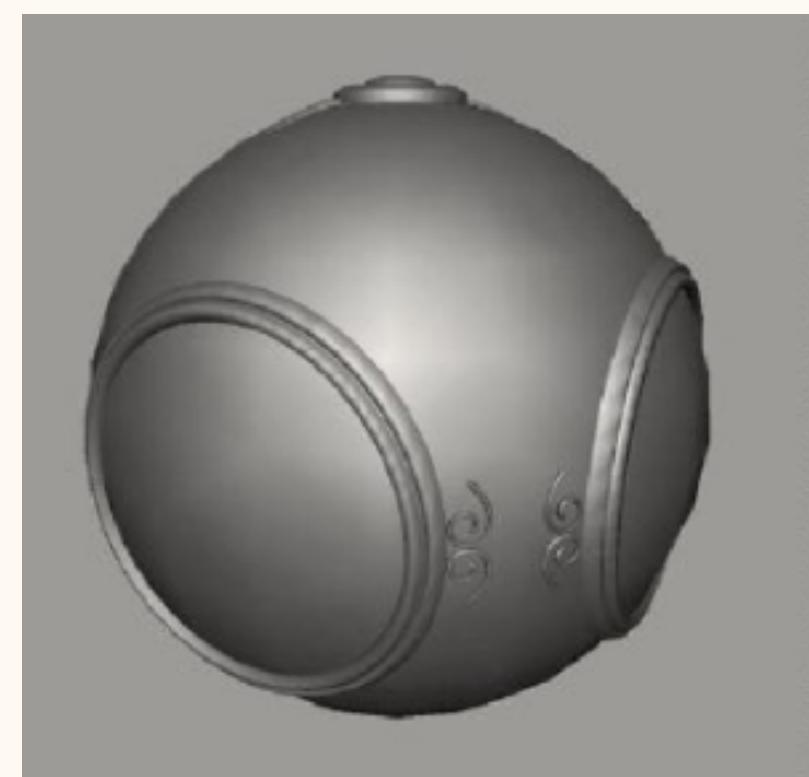
Vertex Normal



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



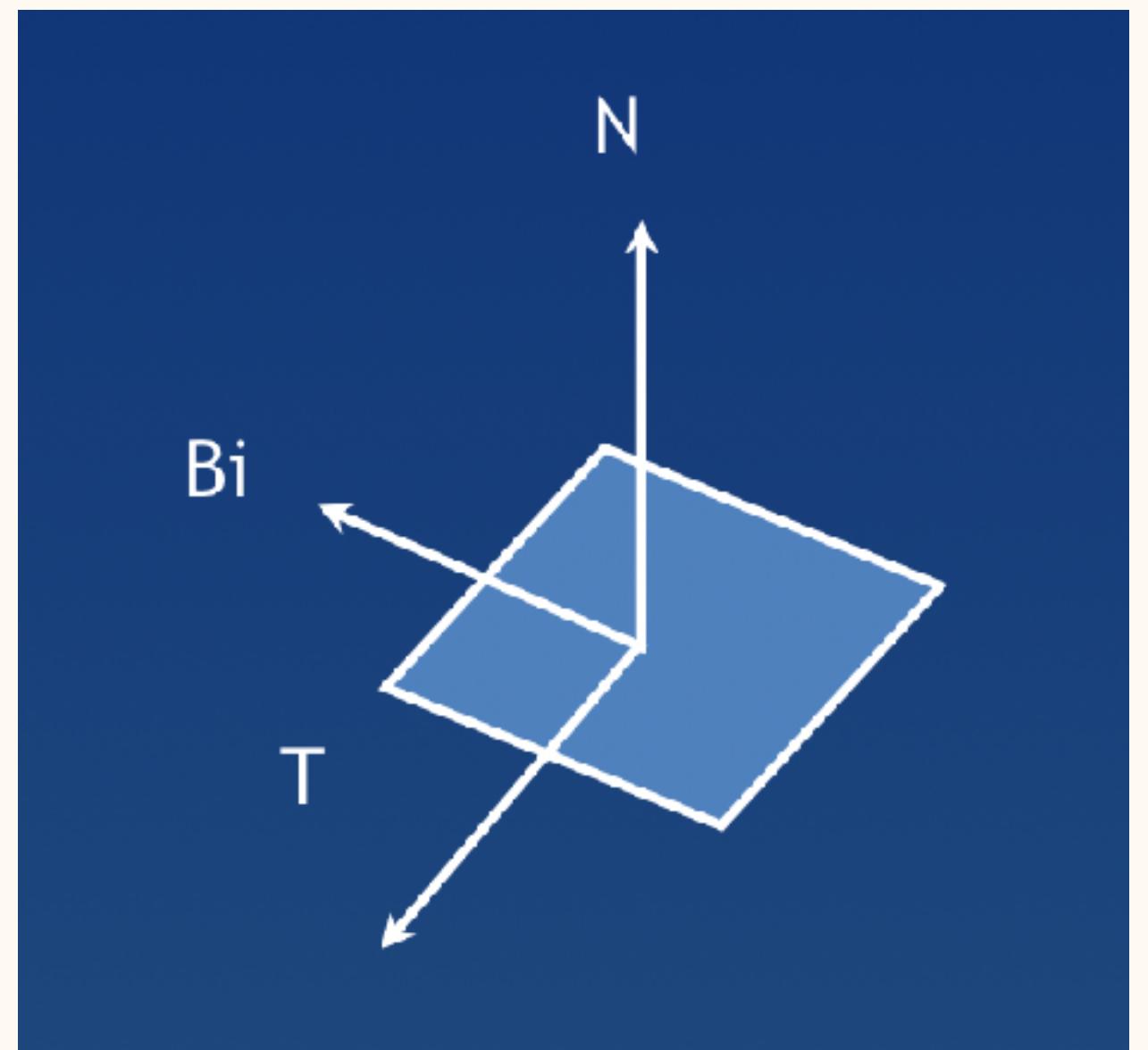
Use Surface Normal



Use Vertex Normal

Vertices

- Optional data on vertices
 - Tangent (T) & Bi-normal (Bi)
 - $Bi = TxN$
 - Skin weights
 - (bone ID 0, weight 0, bone ID 1, weight 1, ...)

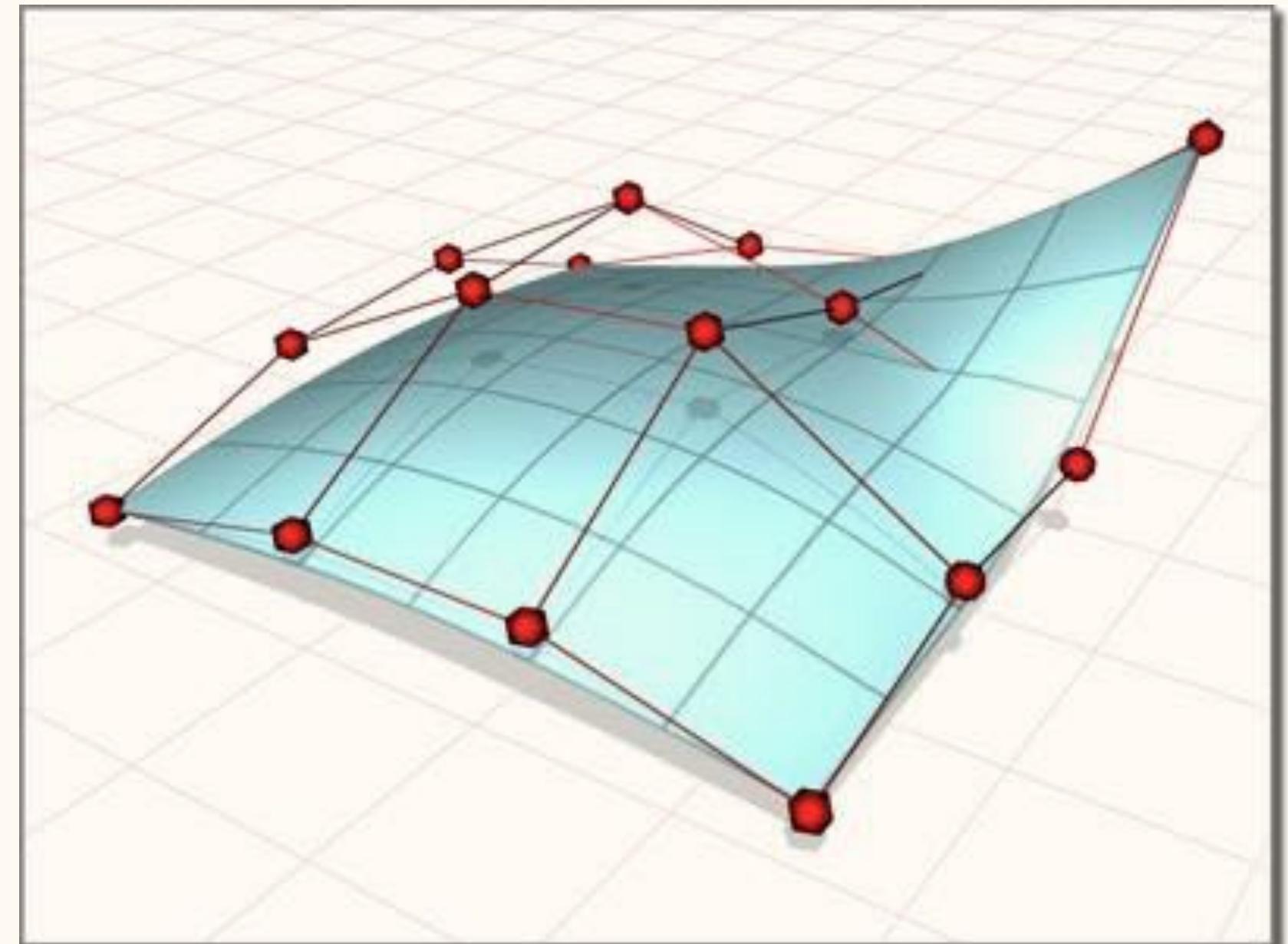
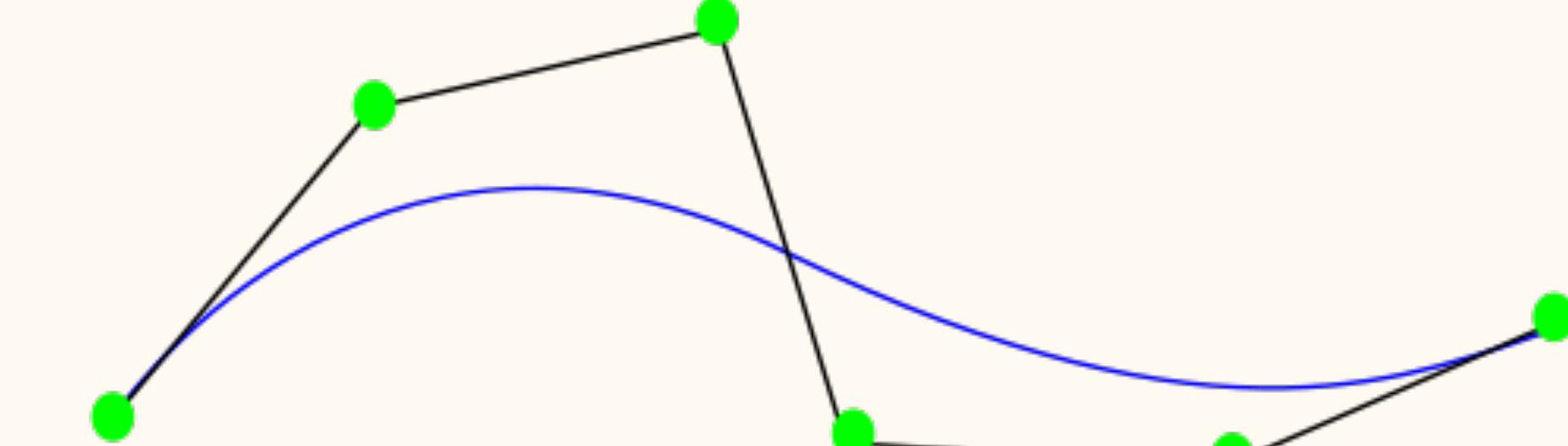


Primitives

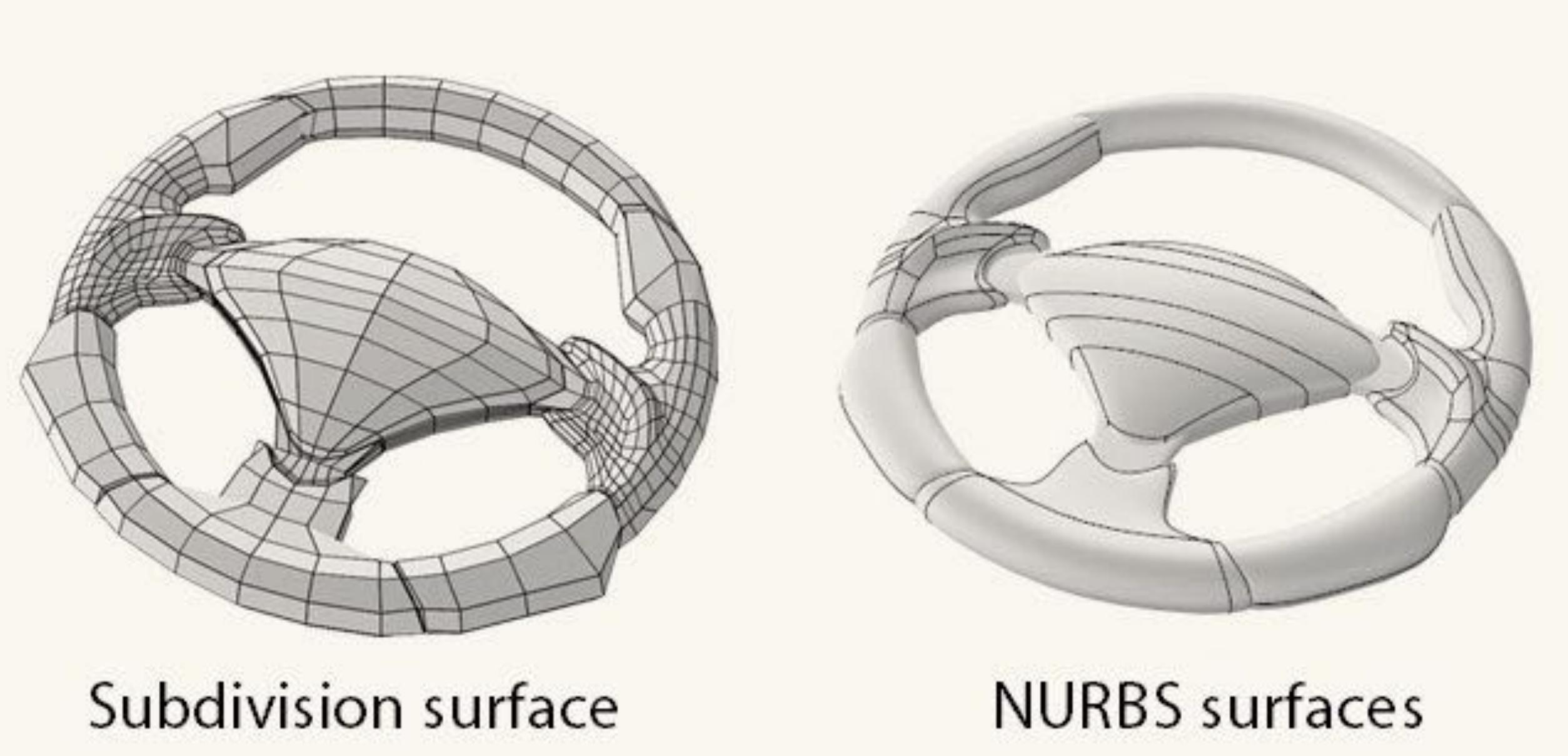
- Indexed triangles
 - Recent real-time 3D rendering hardwares use triangles
 - We always use indexed triangles
 - All vertices in an array
 - Vertex array or vertex buffer
 - Primitives use vertex index to reference the vertex used
 - Index array or index buffer
- High-order primitives
 - Not often in games but in computer animation
 - Parametric surfaces
 - Subdivision surfaces

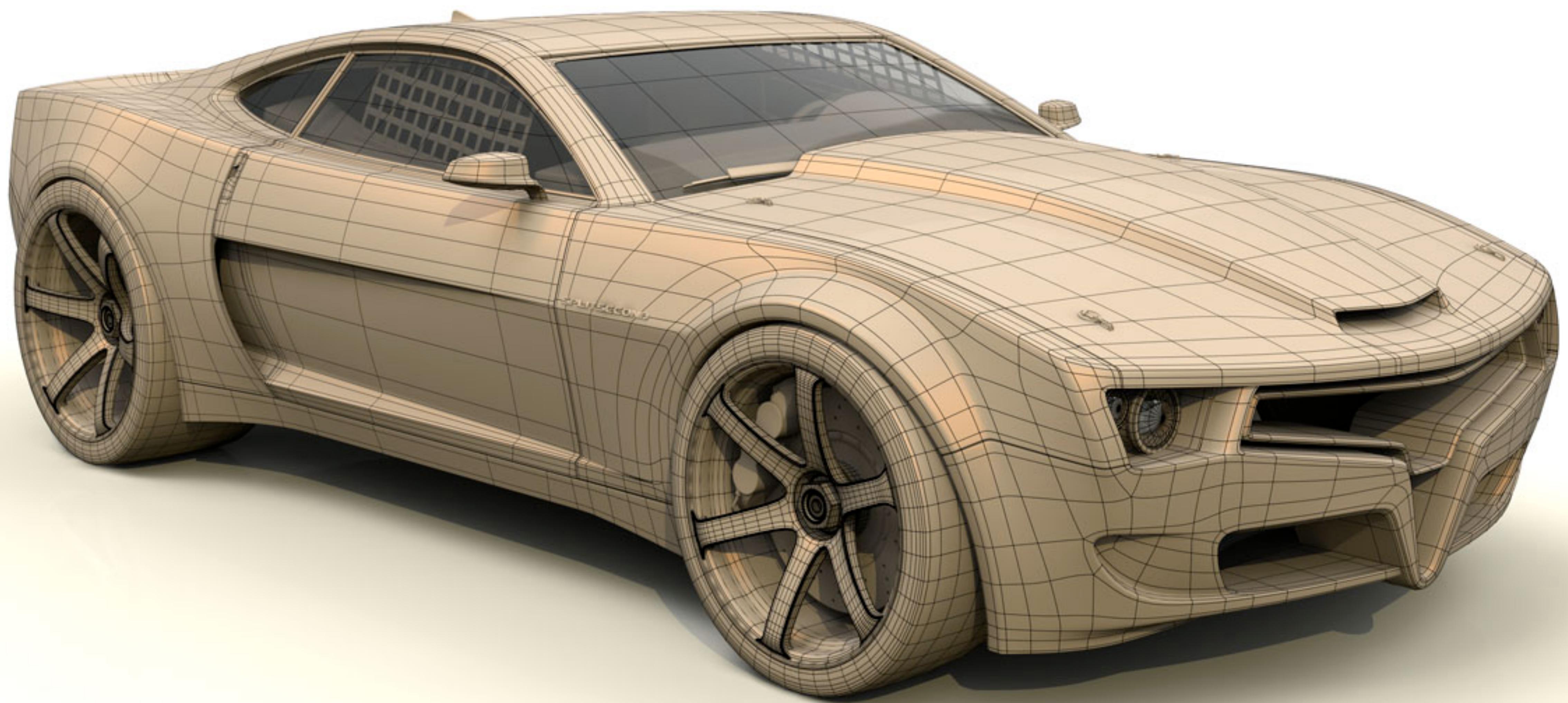
Parametric Surfaces

- You might know some of them :
 - Splines
 - Bezier surfaces
- General solution :
 - Non-uniform Rational B-splines
 - NURBS



Subdivision Surfaces

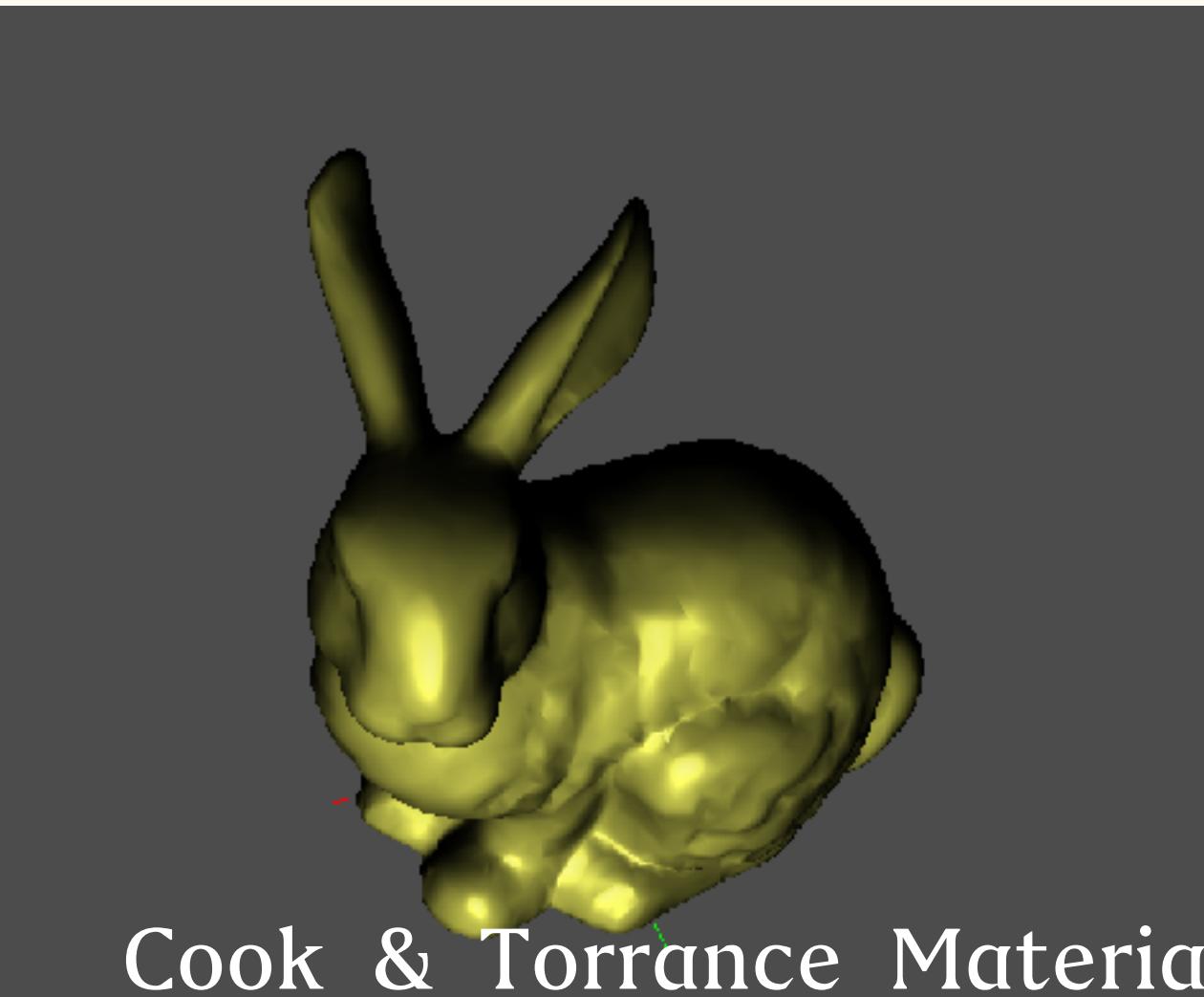




Introduction to Shader

- History of Shaders
 - Real-time Shaders
 - SGI PixelFlow
 - Stanford University
 - Stanford Real-Time Programmable Shading Project
 - <http://graphics.stanford.edu/projects/shading/>
 - Sh
 - <http://libsh.org/>
 - Open source project from University of Waterloo
 - DirectX Shader Models
 - DirectX High Level Shader Language (HLSL)
 - NVIDIA Cg
 - OpenGL Shading Language, GLSL

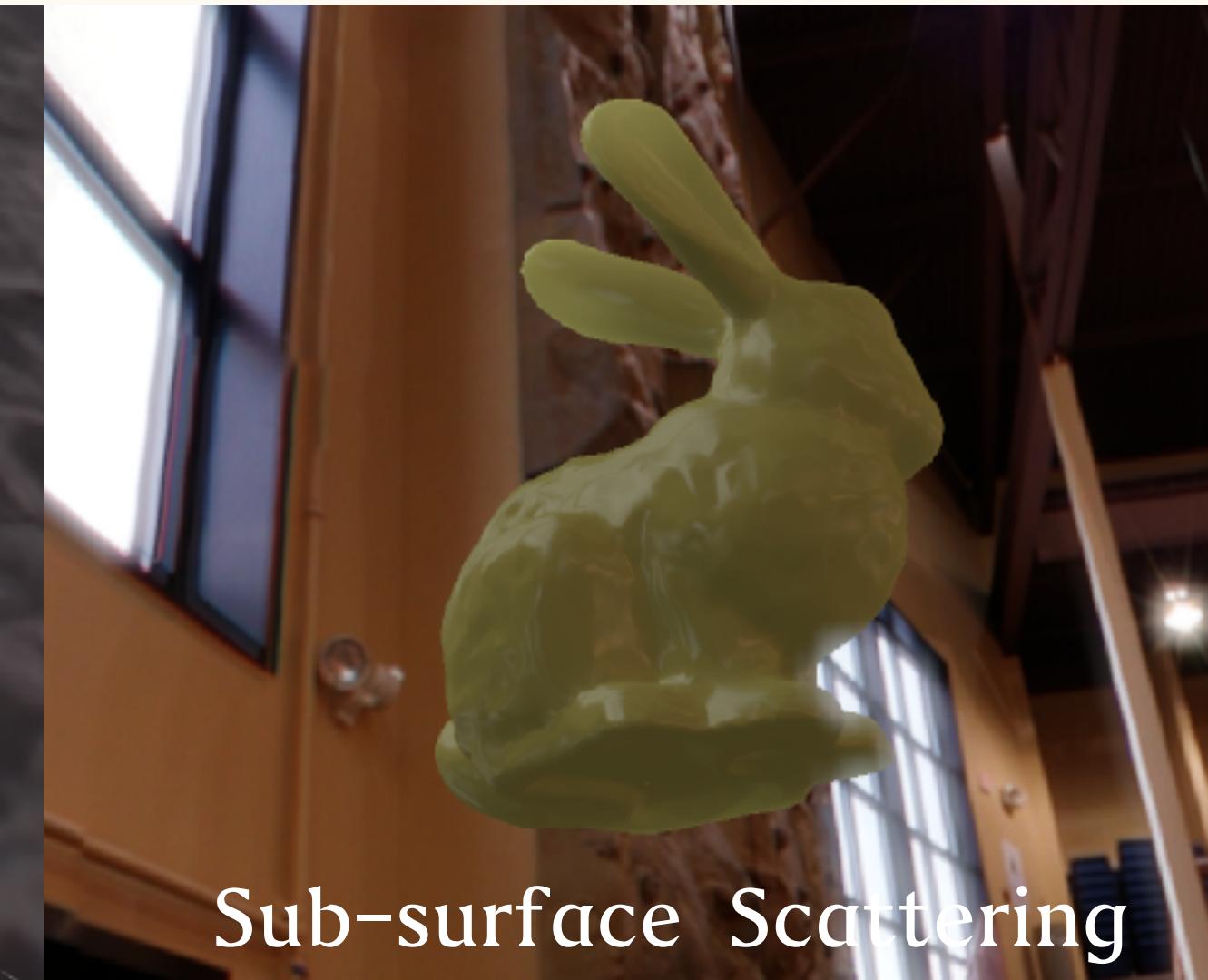
High-quality Real-time Rendering Effects



Cook & Torrance Material



Reflection



Sub-surface Scattering



Parallax Occlusion Map



High Dynamic Range

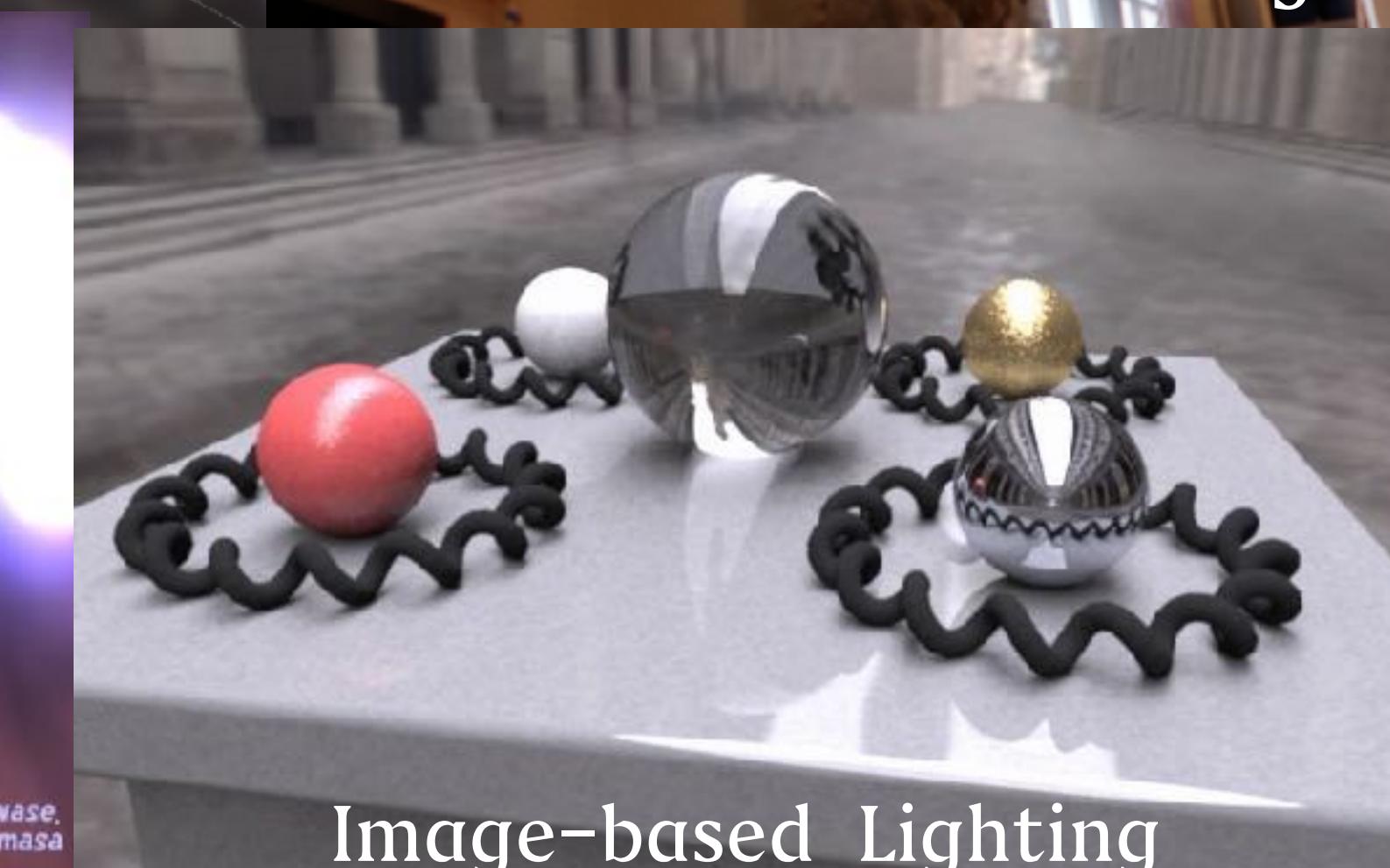


Image-based Lighting

Shader Resources

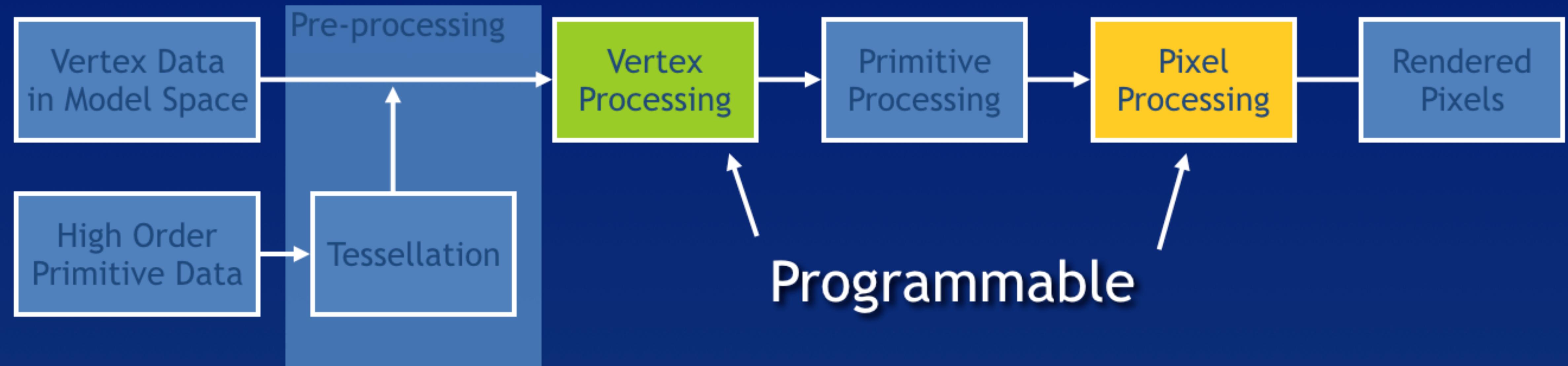
- Shader tools
 - NVIDIA FX Composer / ATI Perfstudio 2
 - MS PIX
 - HDRShop
- Resources
 - “Programming Vertex and Pixel Shaders”, Charles River Media, Wolfgang Engel
 - “ShaderX” Series
 - “GPU Gems” Series
 - “Advanced Game Development with Programmable Graphics Hardware”, A K Peters, Alan Watt, Fabio Pollicarpo
 - “Advanced Lighting and Materials with Shaders”, WORDWARE, Kelly Dempski, Emmanuel Viale

Shader Resources

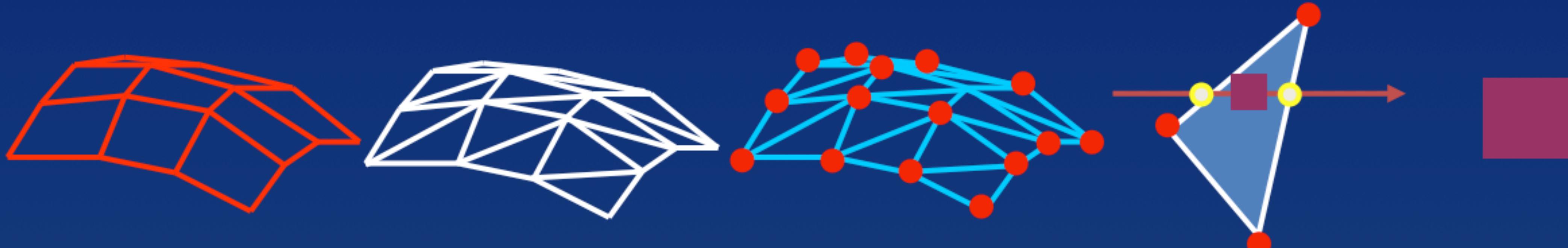
- “High Dynamic Range Imaging – Acquisition, Display and Image-based Lighting”, Morgan Kaufmann, Erik Reinhard, Greg Ward, Sumanta Pattanaik, Paul Debevec
- “Practical Rendering & Computation with Direct3D 11”, CRC Press, Jason Zink, Matt Pettineo, and Jack Hoxley
- GPGPU web site
 - <http://www.gpgpu.org/>
- NVIDIA & ATI web sites for developers
 - <http://ati.amd.com/developer/>
 - <http://developer.nvidia.com/page/home.html>
- Game Developer Conference courses
- SIGGRAPH / SIGGRAPH Asia Conferences

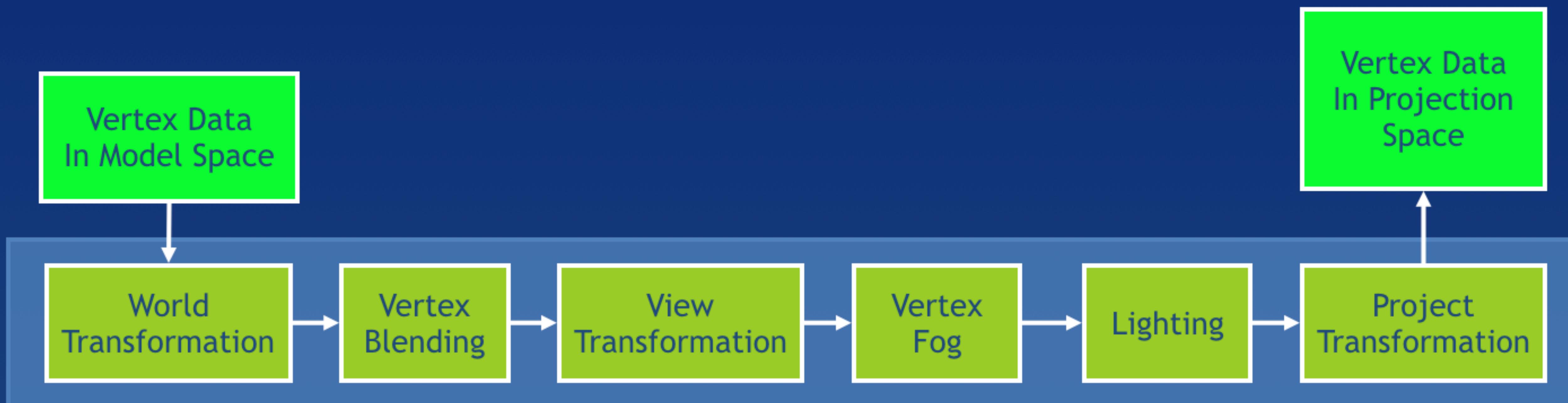
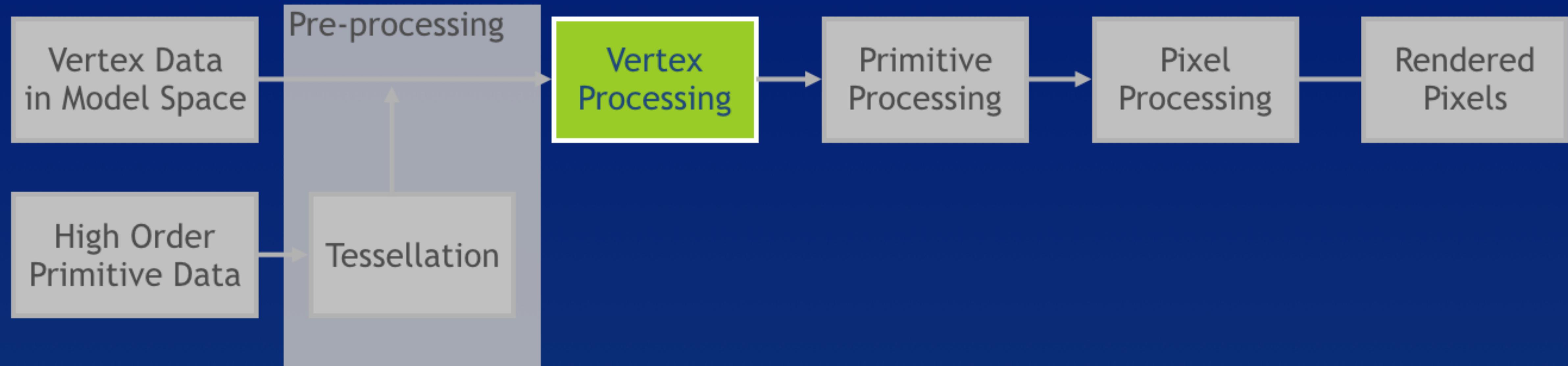


Realtime 3D Rendering Pipeline



Programmable





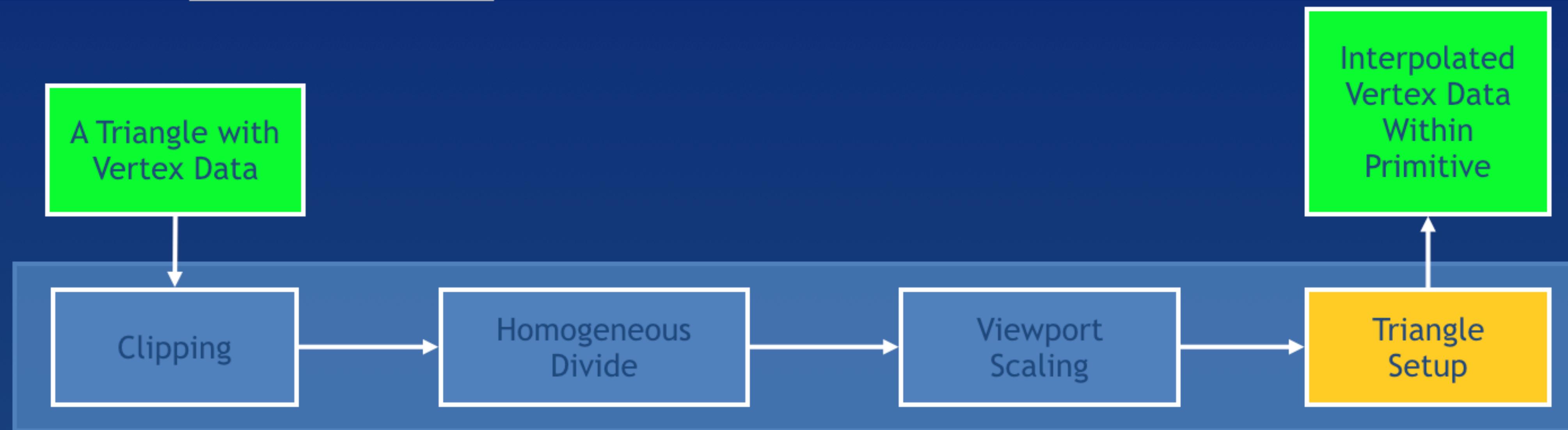
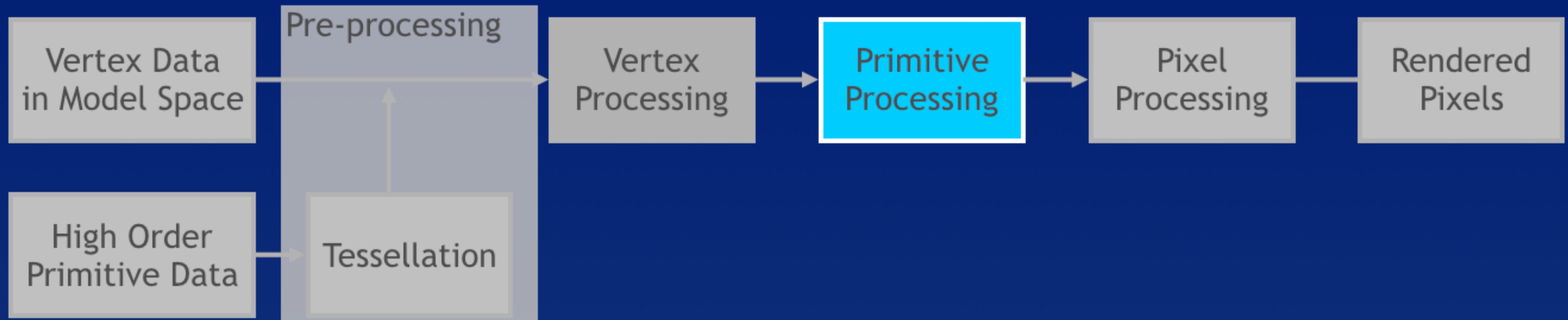
Vertex Processing

- Three 4x4 transformation matrices in this processing
 - World matrix
 - From local space to world space
 - View matrix
 - From world space to view space
 - Projection matrix
 - Orthogonal or perspective projection
- From local space to projection space :

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

Vertex Processing

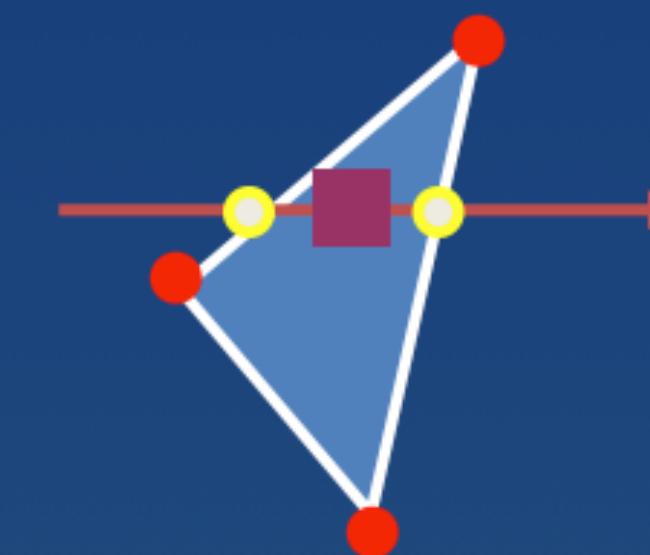
- Skin deformation is always calculated in world space
 - Vertex blending
- After viewing, vertex fog is added as necessary (Optional)
- Lighting is calculated on vertices (Optional)
 - Same as in fixed function 3D rendering pipeline
 - Suggested by hardware companies but we do it in Pixel Shader
- After the vertex processing :
 - The vertices will be in projection space with :
 - Vertex position : (x, y, z, w)
 - $w > 1.0$ (input = 1.0)
- Vertex processing is programmable.
 - Vertex shader



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

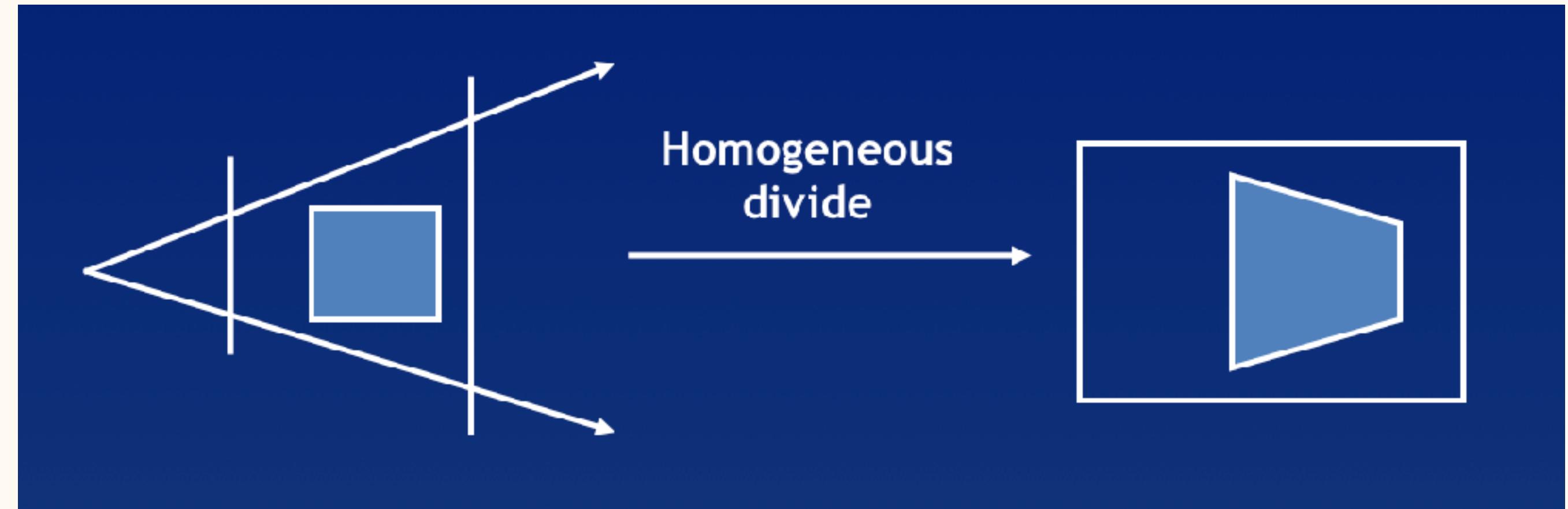
$$\begin{aligned} -1 < x/w < 1 \\ -1 < y/w < 1 \\ 0 < z/w < 1 \end{aligned}$$

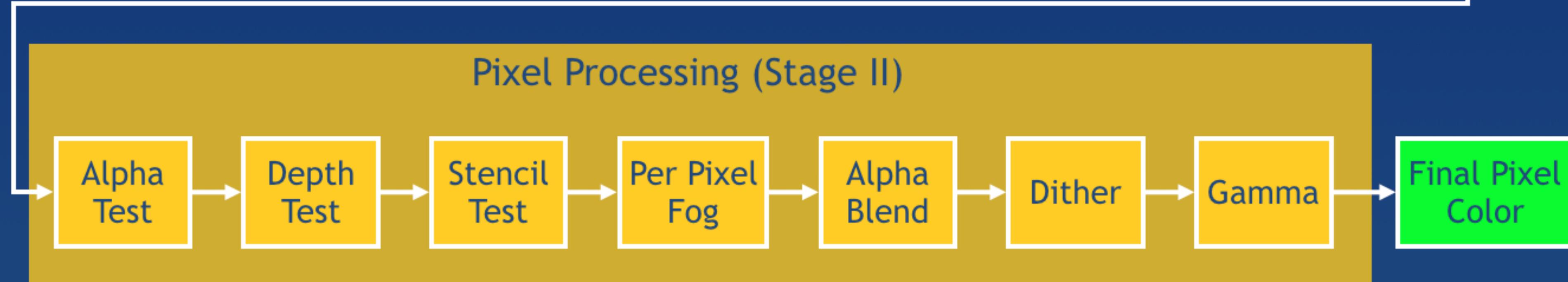
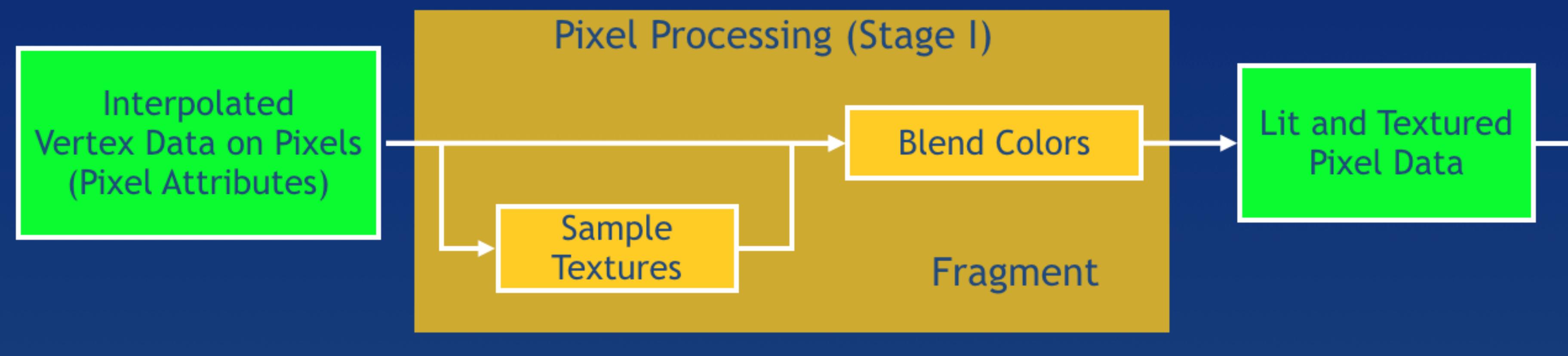
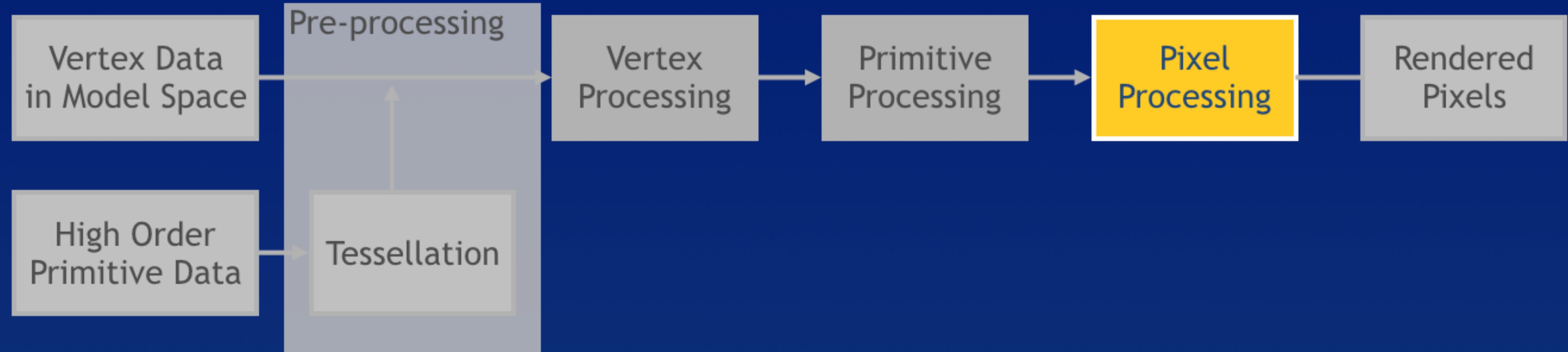
$$\begin{aligned} o_x \leq x_s < o_x + w \\ o_y \leq y_s < o_y + h \\ 0 < z_s < 1 \end{aligned}$$



Primitive Processing

- After homogeneous divide,
 - $-1 < x < 1$
 - $-1 < y < 1$
 - $0 < z < 1$
- Then mapping to a viewport
 - Scale (x, y) to viewport
 - Map z value to z buffer
- Triangle setup
 - Setup the triangle edge
 - Bi-linear interpolation using vertex data
- Converting per-vertex attributes to per-pixel attributes
- Non-programmable





Pixel Processing

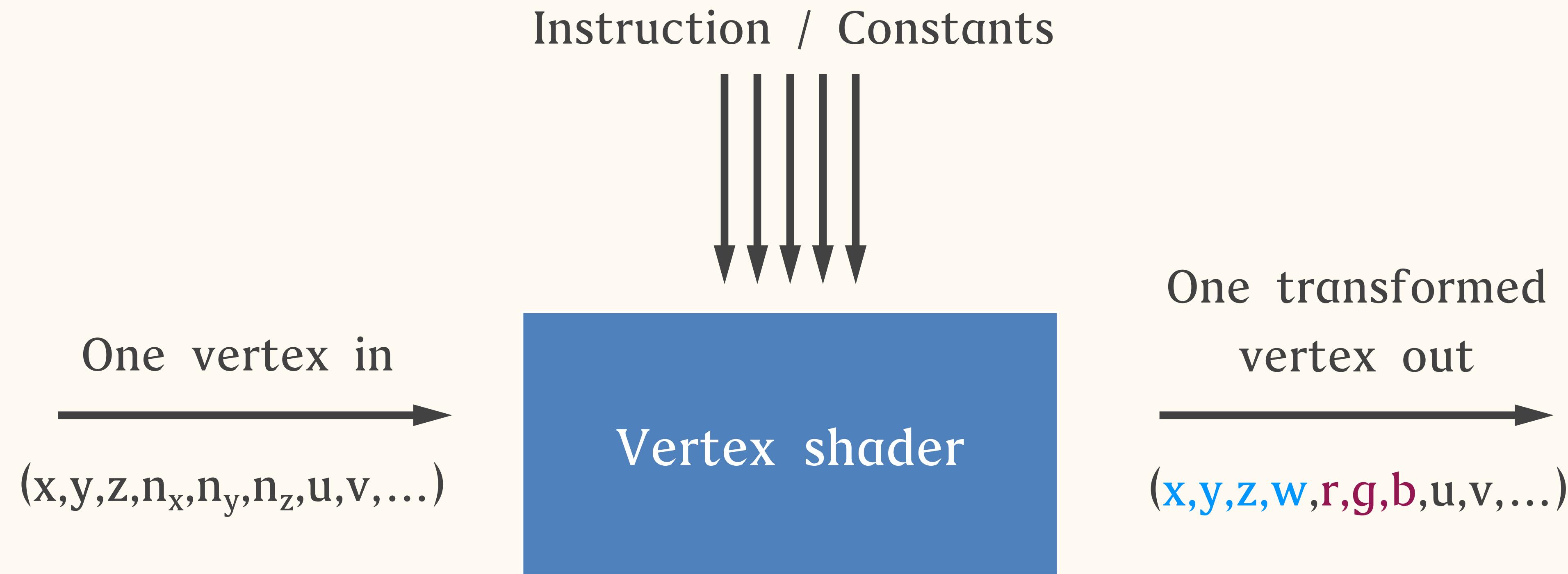
- Pixel processing has the most computing cost in the rendering pipeline.
- Two stages :
 - Stage I : fragment processing
 - Programmable
 - Stage II :
 - Non-programmable
- In stage I :
 - GPU samples multiple texture data according to the interpolated texture coordinates.
 - Then blend per-pixel attributes :
 - Diffuse and specular colors
 - Multiple texture samples

Pixel Processing

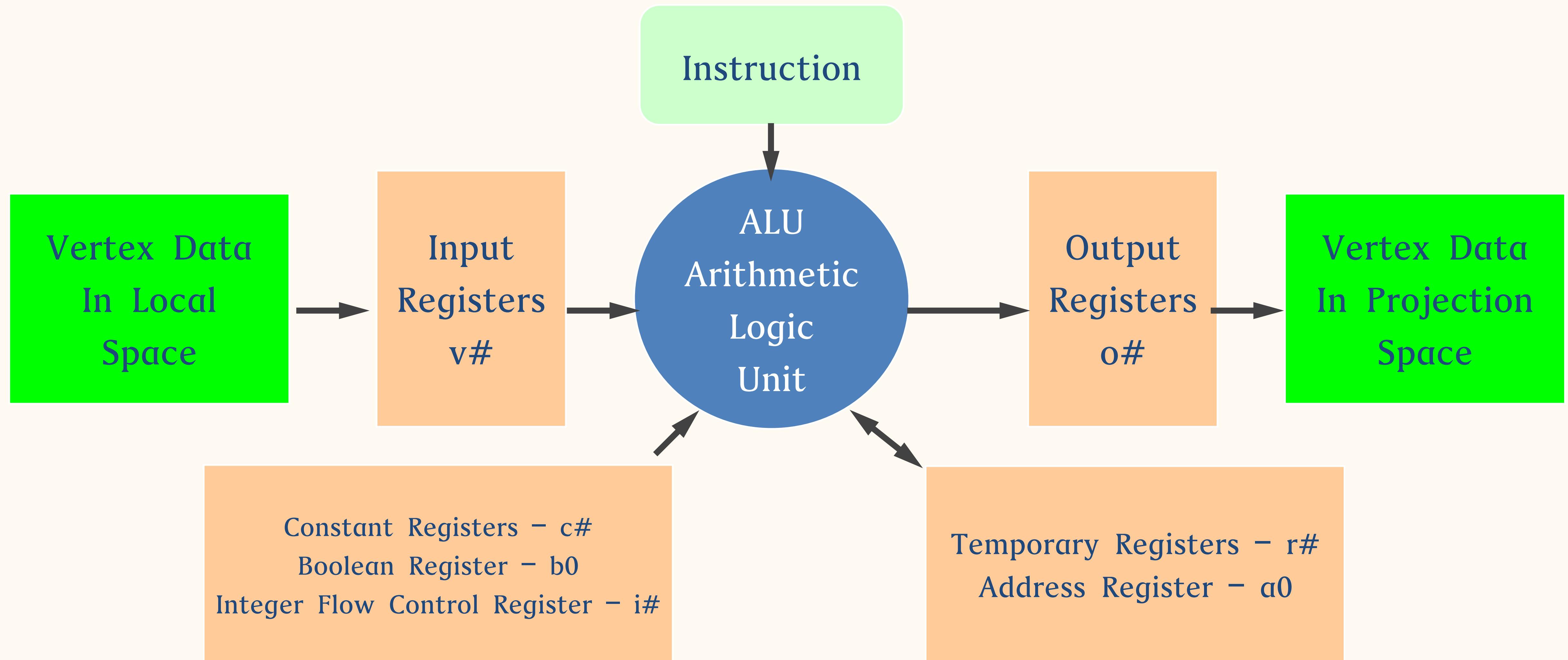
- The lit/textured pixel data in (r, g, b, a) is generated after the fragment processing
- In stage II :
 - Several tests are performed if the pixel affects the final pixel color
 - Alpha test
 - Depth test
 - Z buffer test
 - Stencil test
 - In the alpha blend
 - Apply pixel alpha to create a semi-transparent blend between a source pixel and frame buffer pixel
- Only the pixel processing stage I is programmable.

Shader Programming

Vertex Shader – Introduction



Vertex Shader Virtual Machine Block Diagram



Vertex Shader Versions

- `vs_1_1`
 - DirectX 8.x
 - Max 128 instruction slots
 - No flow control
- `vs_2_0`
 - DirectX 9.0
 - Max 256 instruction slots
 - Add a few arithmetic & macro instructions
 - Static flow control
 - `if-else-endif`, `call`, `loop`, `Repeat (rep)`
- `vs_2_x`
 - DirectX 9.0c

Vertex Shader Versions

- Max 256 instruction slots
- Dynamic flow-control
 - `break_comp`, `break_pred`, `if_comp`, `if_pred`
- Device dependent capability query
 - i.e., `VS20Caps.NumTemps`
- `vs_3_0`
 - DirectX 9.0c
 - Max 512 instruction slots
 - Output register set simplified to a single type of register
 - `O#`
 - Declarations & semantics

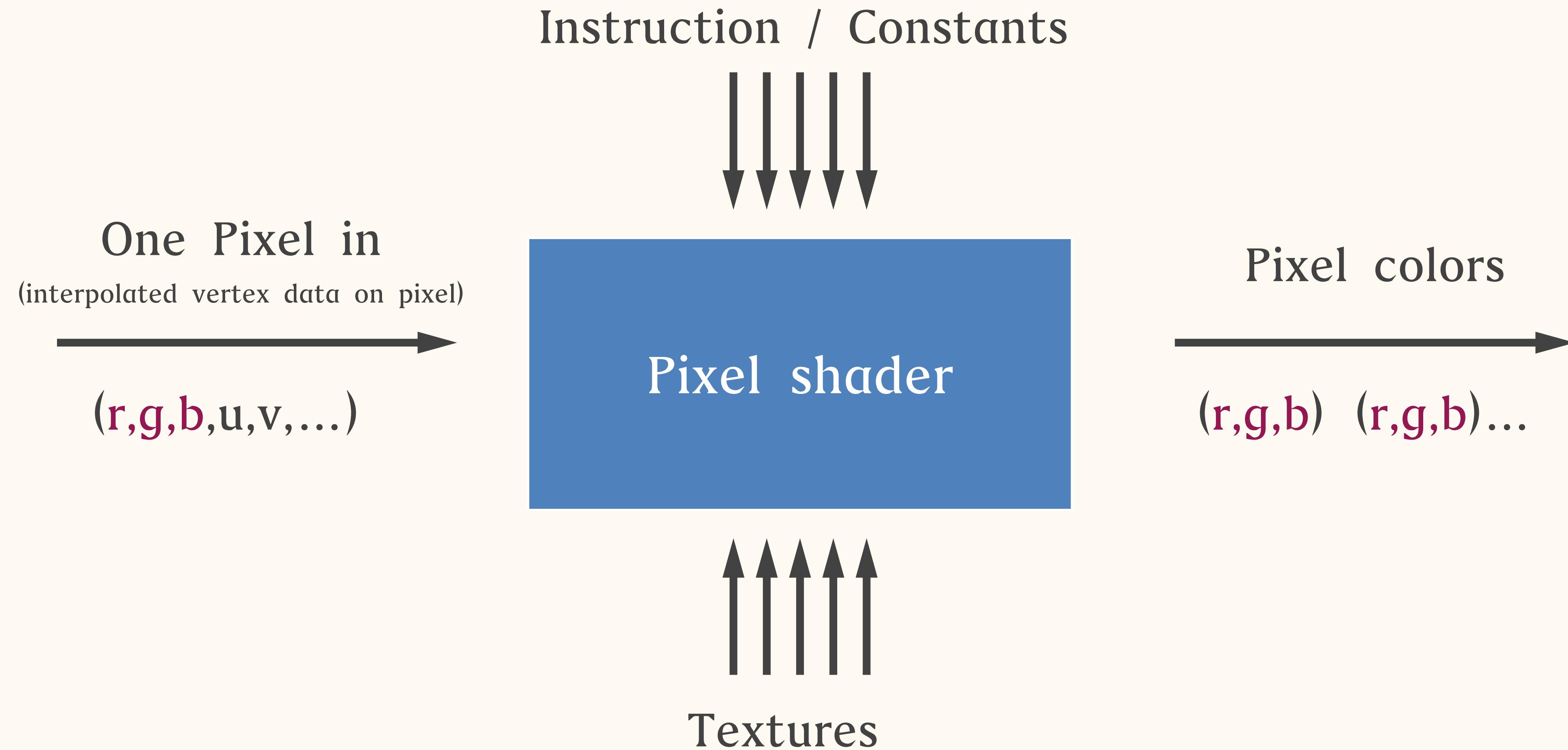
Vertex Shader Versions

- More easier to connect to pixel shader 3.0
- Texture sampling in vertex shader
- Controlling vertex streaming frequency
 - Hardware instancing
- vs_4_0
 - DirectX 10.0
- vs_5_0
 - DirectX 11.0
- vs_5_1
 - DirectX 12.0

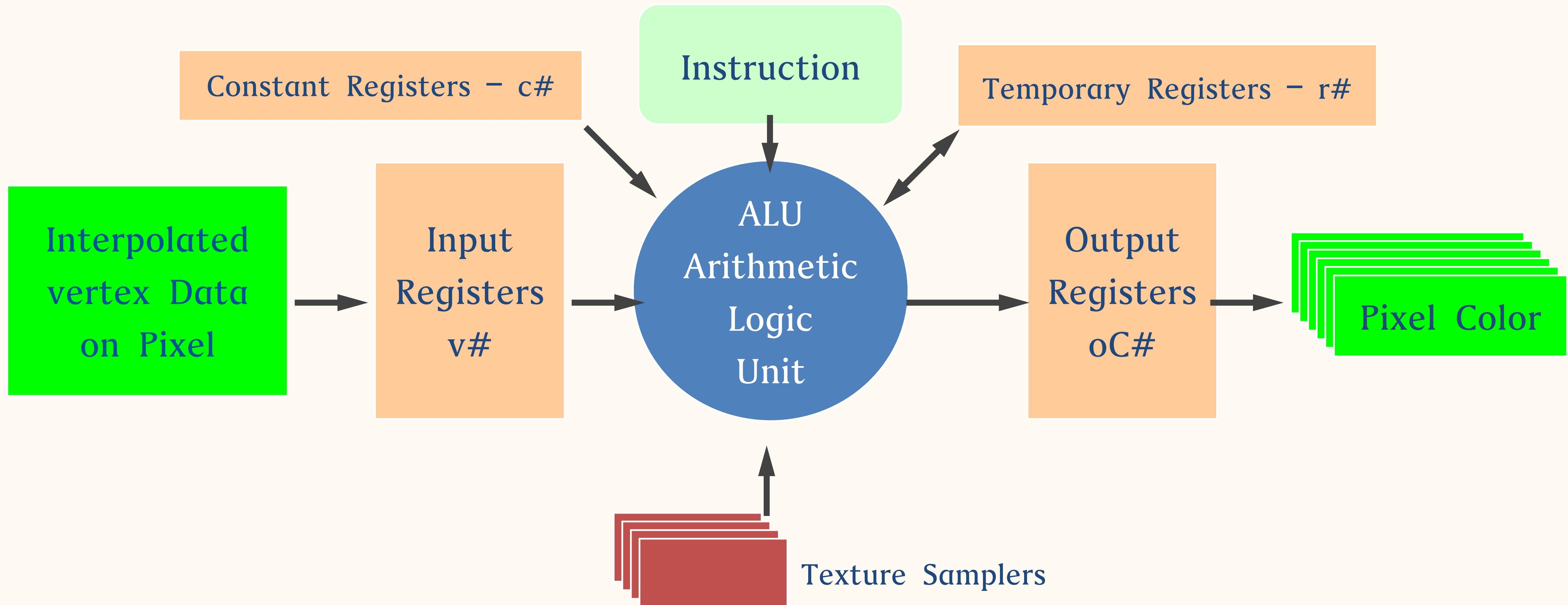
Vertex Shader Versions

- `vs_6_0`
 - Shader Model 6.0
 - Multi-thread management for GPU
 - New Wave-level Operations

Pixel Shader – Introduction



Pixel Shader Virtual Machine Block Diagram



Pixel Shader Versions

- ps_1_0
 - ps_1_1 : Max 8 instruction slots
 - ps_1_2 : Max 12 instruction slots
 - ps_1_4 : Max 14 instruction slots
- ps_2_0
 - Max 32 for texture and 64 for other instruction
- ps_2_x
 - 512 (96 minimum guaranteed)
- ps_3_0
 - 32768 (512 minimum guaranteed)
- ps_4_0 ~ ps_6_0

Shading Language

- Shader Model
 - Now is Shader Model 6.0
 - Assembly
- High-level Shading Language
 - 1st high-level shading language : nVidia Cg
 - Microsoft solution
 - High-Level Shading Language (HLSL)
 - OpenGL & OpenGL ES solution : GLSL
 - C-like but enhanced for vector operation and graphics
 - Learn one language, then you know all ...

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

- Vector data type examples :

```
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

- Swizzling refers to the ability to copy any source register component to any temporary register component.

```
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)
```

HLSL – Constructor

- 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!
```

A Unity 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
}
```

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
 - Most used in games
 - 3D texture sampling
 - Volume texture
 - 2D textures in layer
 - Cubemap texture sampling
 - 6 textures attached on 6-internal-side of a cube surrounding the objects
- In this course, we introduce 2D & 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

- 2D texture sampling function
 - DirectX 9 syntax

```
float4 color = tex2D(s, uv); // s is a texture sampler  
                           // uv is a 2D texture coordinate
```

```
float4 color = tex2Dproj(s, v); //s is a texture sampler  
                           // v is a 4D vector, (x, y, z, w)  
                           // Return value is a color vector in (r, g, b, a)
```

HLSL - 2D Texture Intrinsics

- DirectX 11 syntax

```
Texture2D albedoMap;  
SamplerState albedoMapSampler;  
  
...  
  
float4 color = albedoMap.Sample(albedoMapSampler, uv);
```

HLSL - Cubemap Texture Intrinsics

- Cubemap texture sampling
 - DirectX 9 syntax

```
float4 color = texCUBE(s, v3D); // s is a texture sampler  
                                // v3D is a 3D vector.  
                                //Return value is a color vector in (r, g, b, a)
```

HLSL - Cubemap Texture Intrinsics

- DirectX 11 syntax

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

Lighting - Phong Reflection Model

Material – BRDF (Bidirectional Reflectance Distribution Function)

Generally we define a BRDF as

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

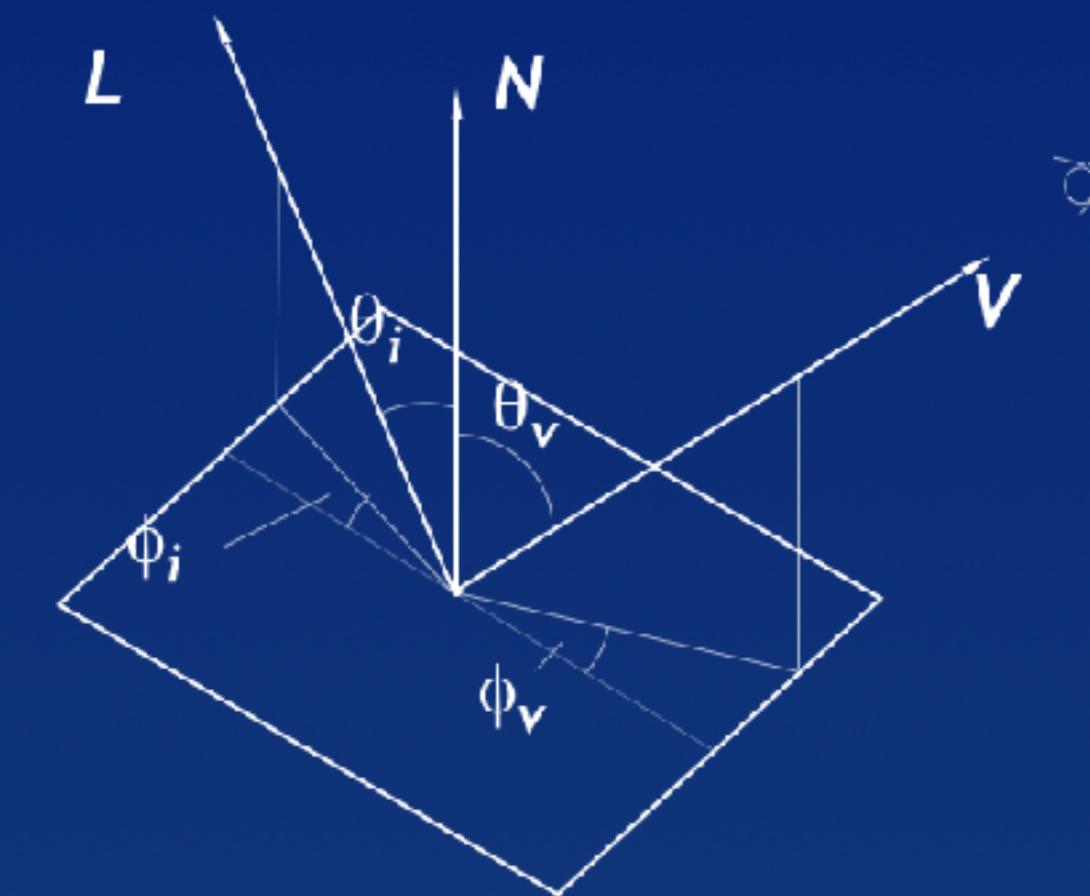
that relates incoming light in the direction (ϕ_i, θ_i) to outgoing light in the direction (ϕ_v, θ_v) . Theoretically, the BRDF is the ratio of outgoing intensity to incoming energy :

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

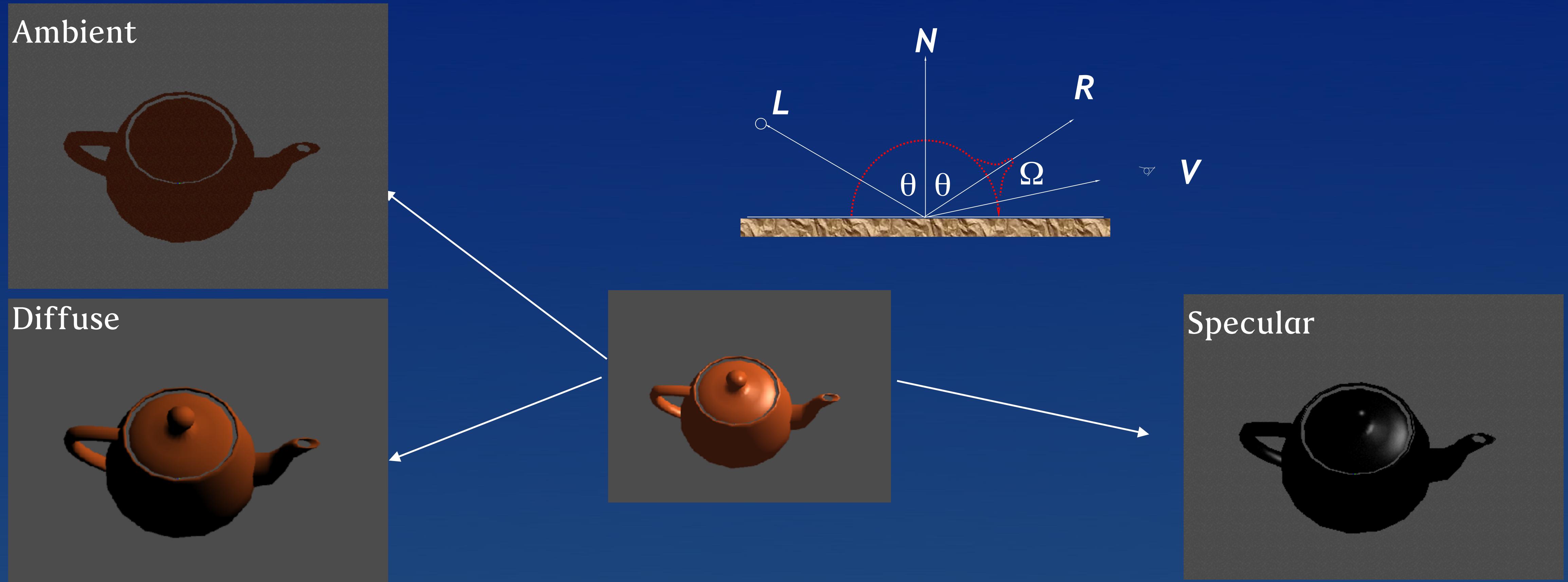
where the relationship between the incoming energy and incoming intensity is given by

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

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.



Phong Reflection Model



- 1975 By Mr. Bui-Tuong Phong

Phong Reflection Model – Ambient

- Ambient
 - Caused by in-directed lighting from environment
 - Hard to calculate
 - Solution :
 - Path Tracing
 - A frame costs many hours/days to render
- Phong suggested it can be a constant as approximation

$$\text{ambient} = I_a * K_a$$

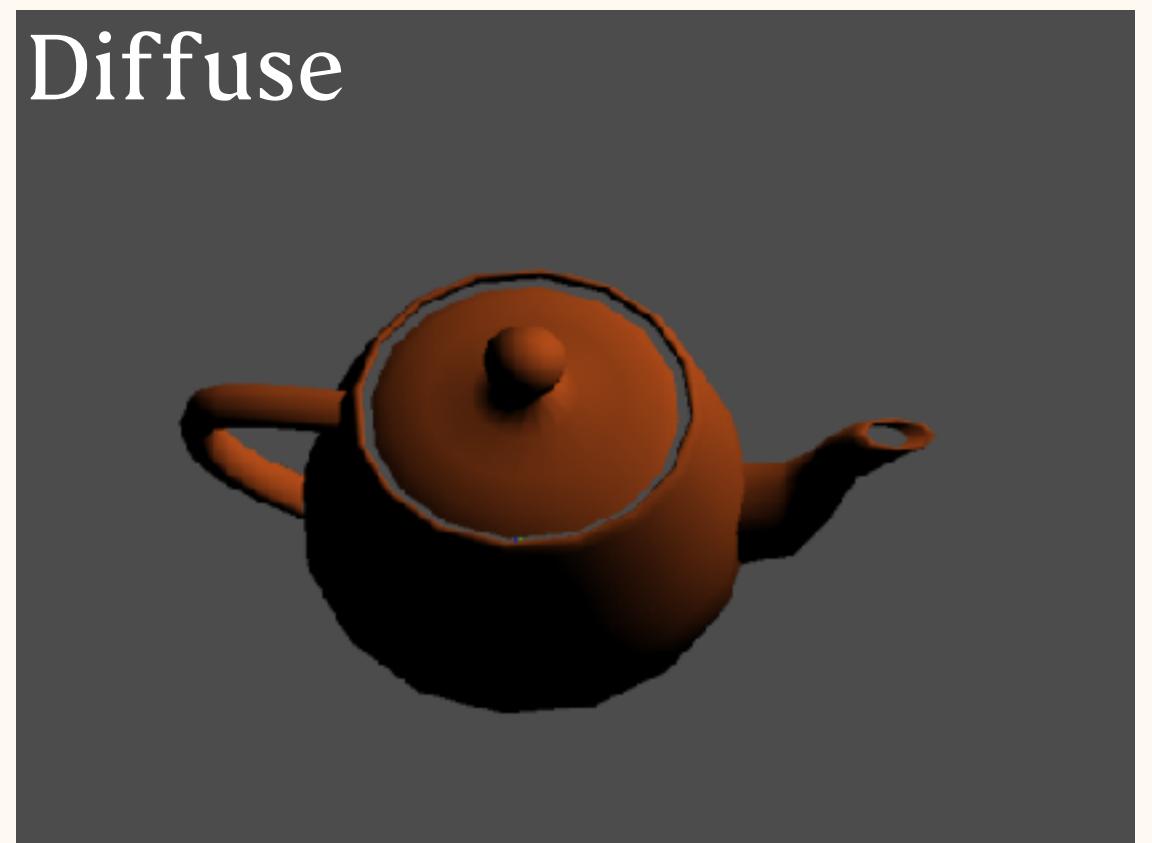
I_a = Environment Lighting Intensity

K_a = Ambient component of material



Phong Reflection Model – Diffuse

- Diffuse
 - “Ideal Diffuse”
 - This is the object’s base color.
 - The intensity is related to how much light reaching the effective area of the object surface.

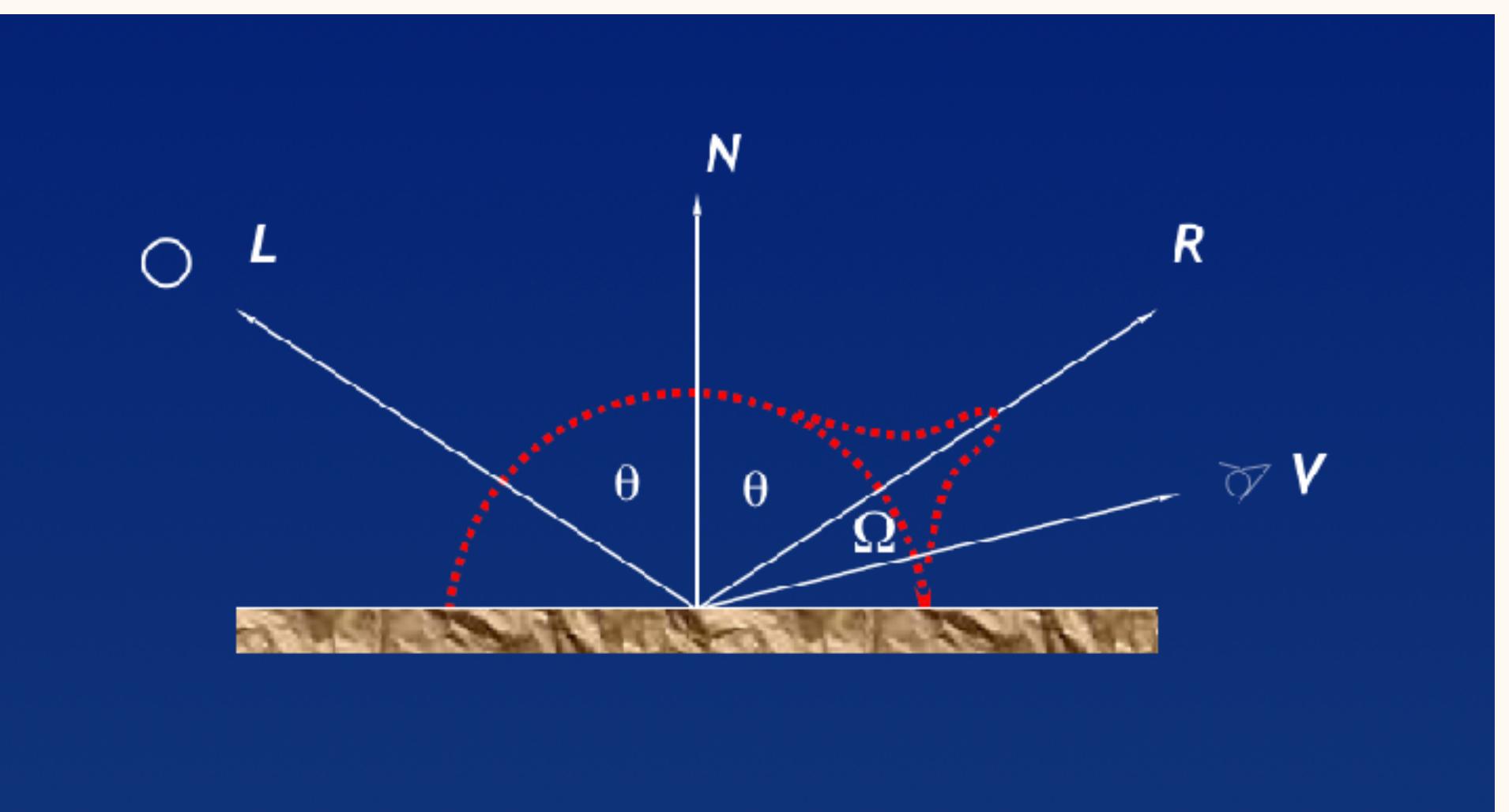


$$\text{diffuse} = K_d * I_i * \cos\theta = K_d * I_i * (N \cdot L)$$

I_i = Intensity of light source

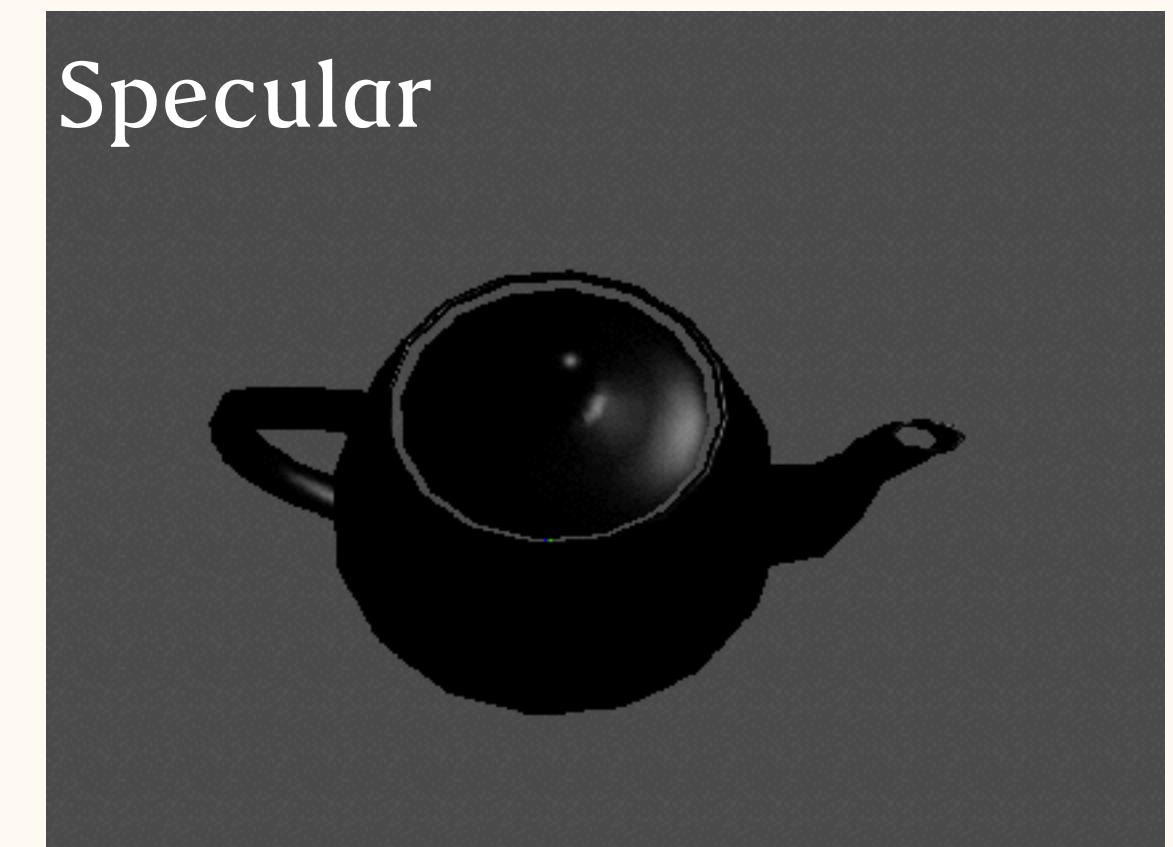
K_d = diffuse component of material

θ = angle between L & N



Phong Reflection Model – Specular

- Specular
 - Reflection image of the light source on object surface
 - Perfect specular = mirror reflection
 - Roughness of object surface = “Shininess”

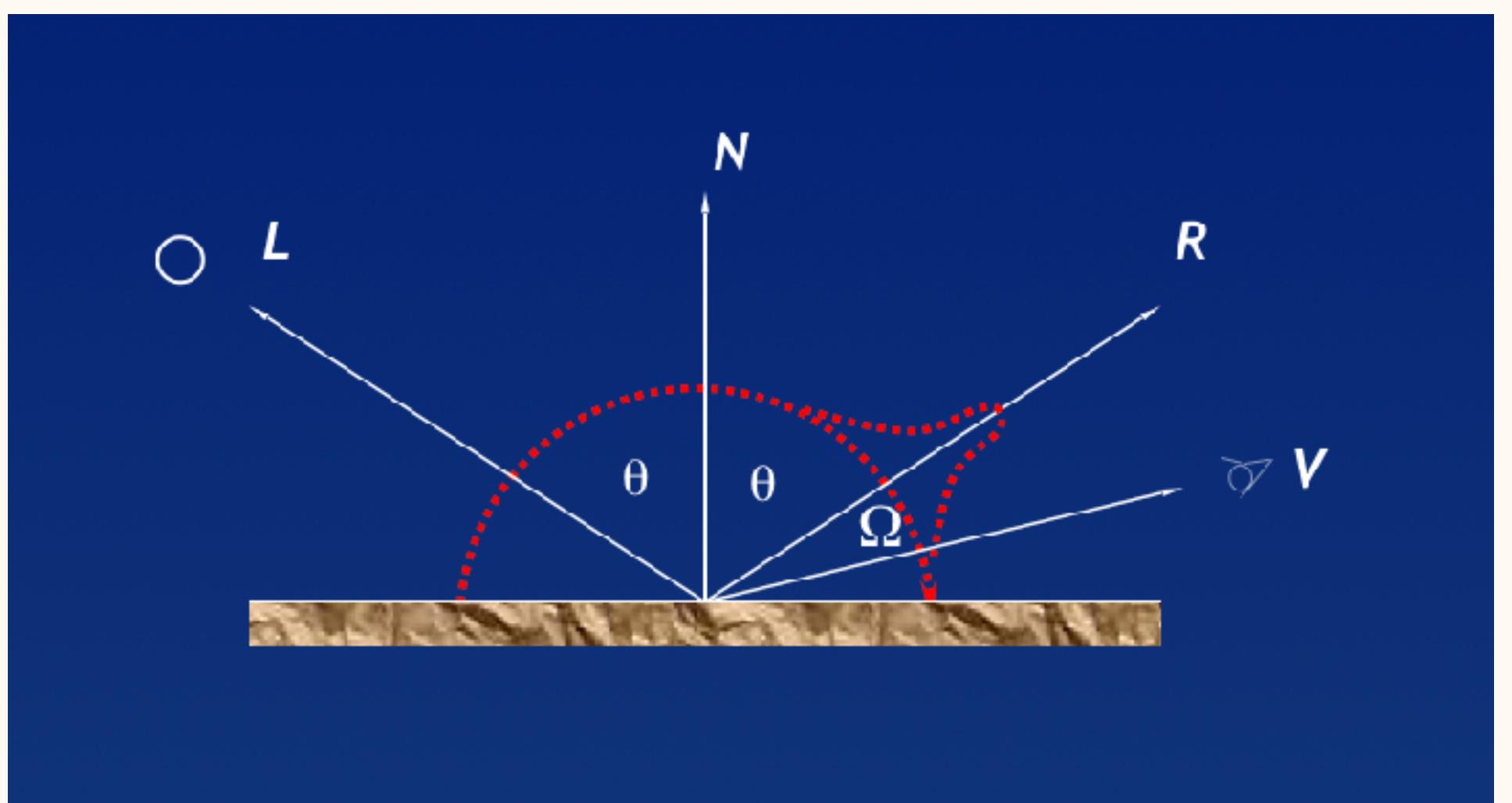


$$\text{specular} = K_S * I_i * \cos^n \Omega$$
$$= K_S * I_i * (R \cdot V)^n$$

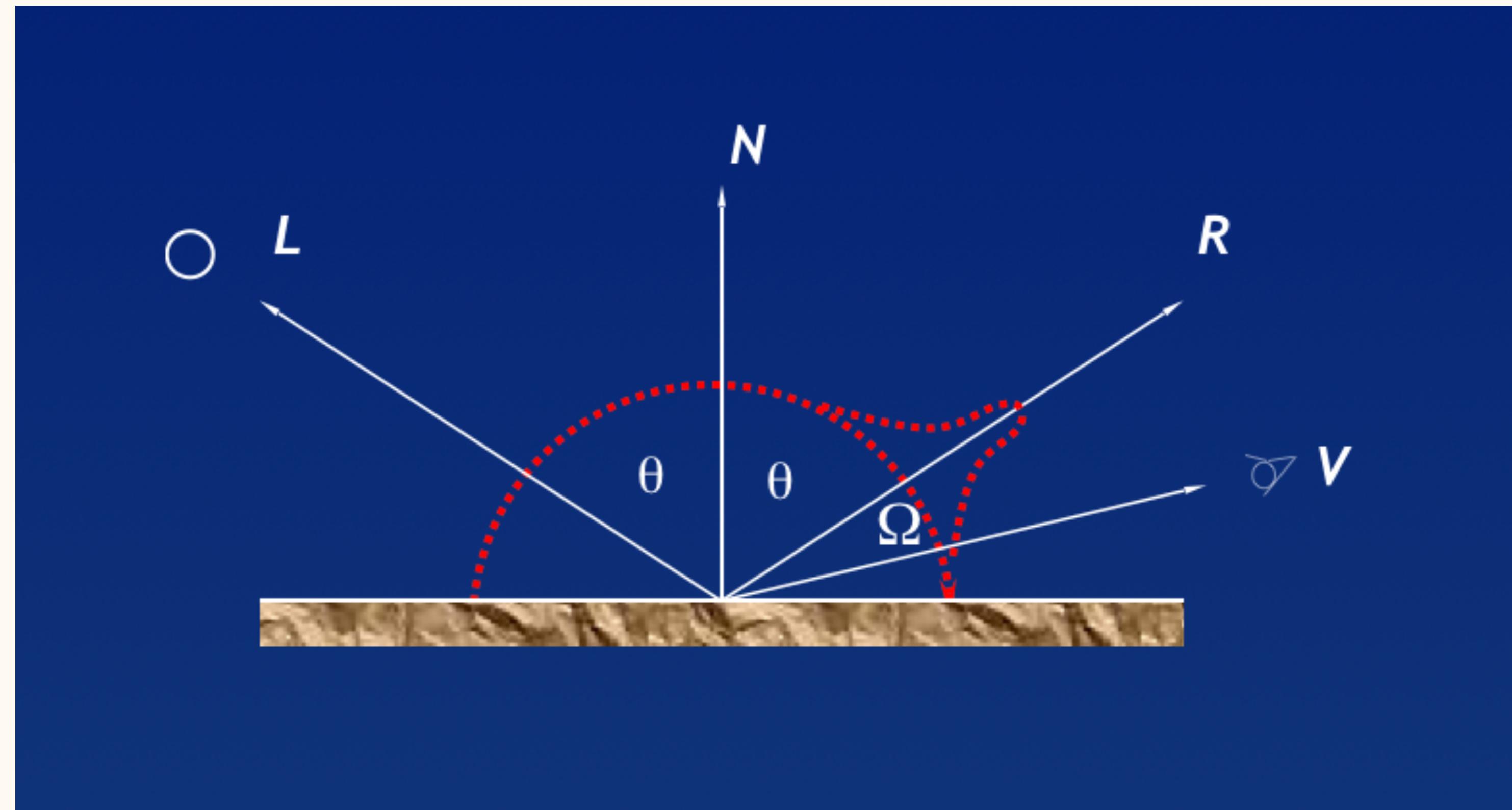
I_i = Intensity of light source

K_S = specular component of material

Ω = angle between R & V

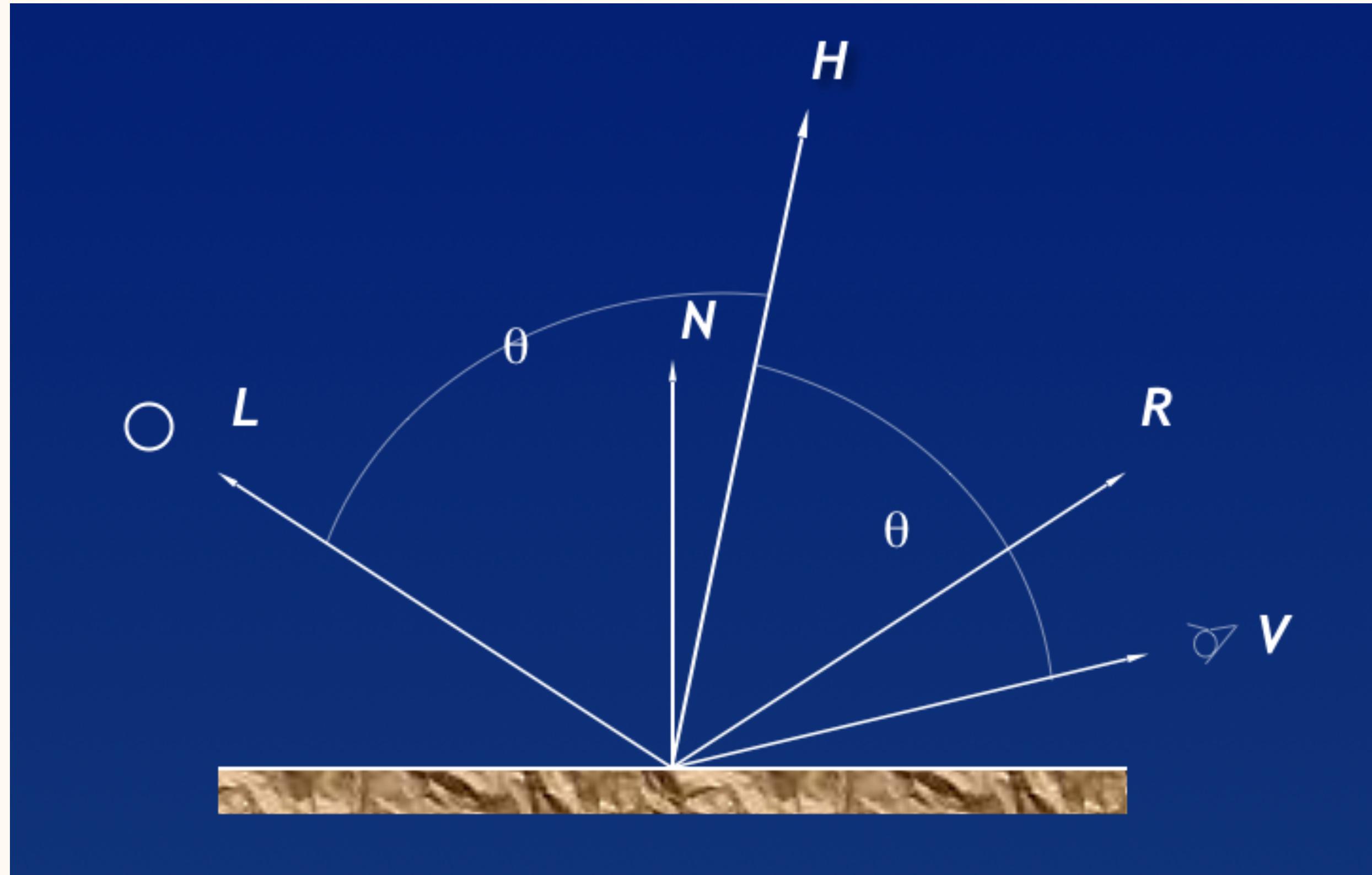


Phong Reflection Model



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

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.

Phong Reflection Model

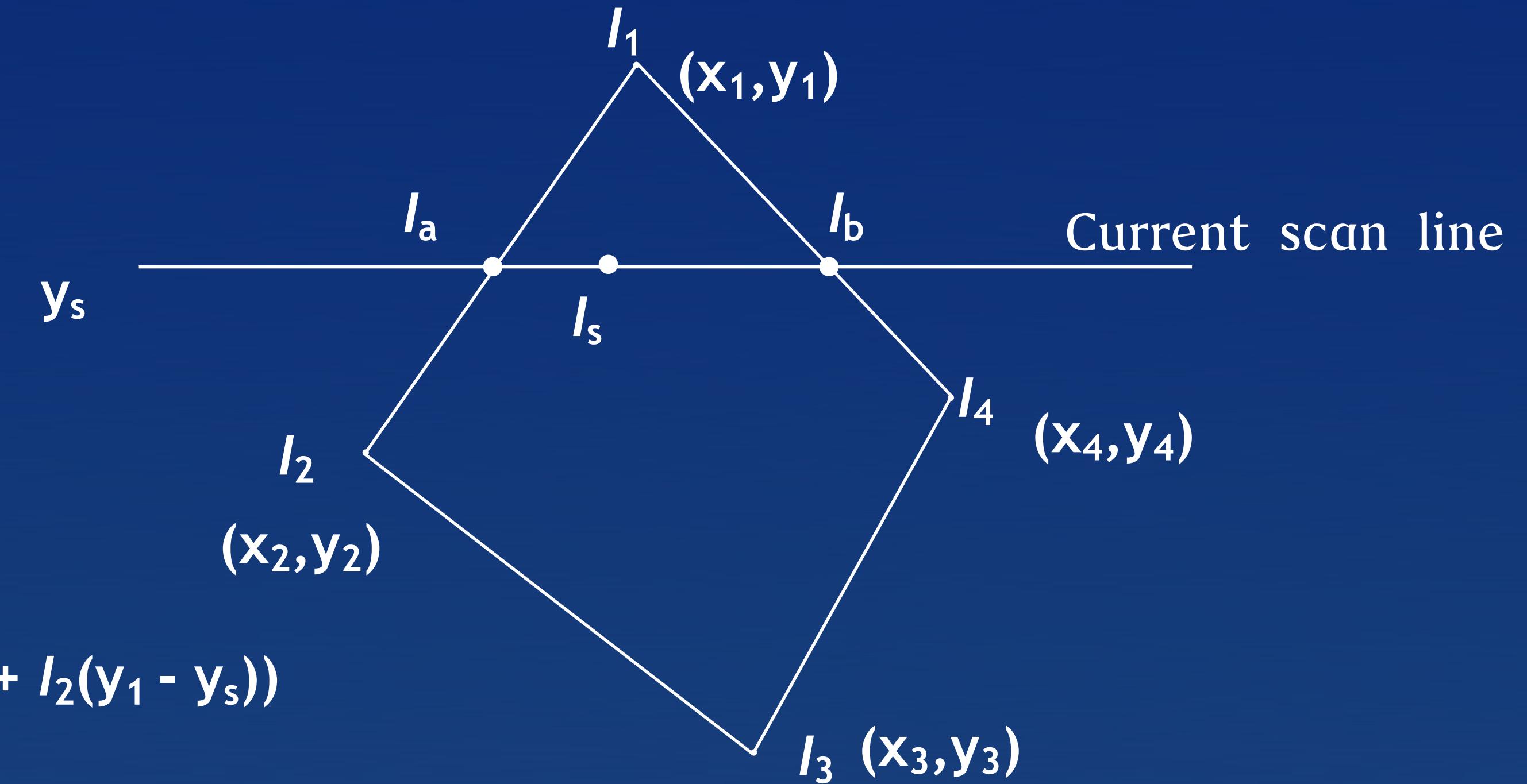
- To complete Phong Reflection lighting we need :
 - Surface material
 - Ambient (K_a)
 - Diffuse (K_d)
 - Specular (K_s) & Shininess (n)
 - Normal vectors on surface (N)
 - Light source
 - Intensity (I)
 - Light source position & direction (L)
 - Ambient lighting intensity (I_a)
 - Viewer (camera)
 - Position & orientation (v)

Shading Techniques

- Real-time shading is still based on scan conversion to render triangles
 - Interpolation for interior points is in scaling order
 - Bi-linear interpolation scheme
 - Two common solutions:
 - Gouraud shading (Gouraud 1971)
 - Phong shading (Phong 1975a)

Gouraud Shading

- Gouraud, 1971



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

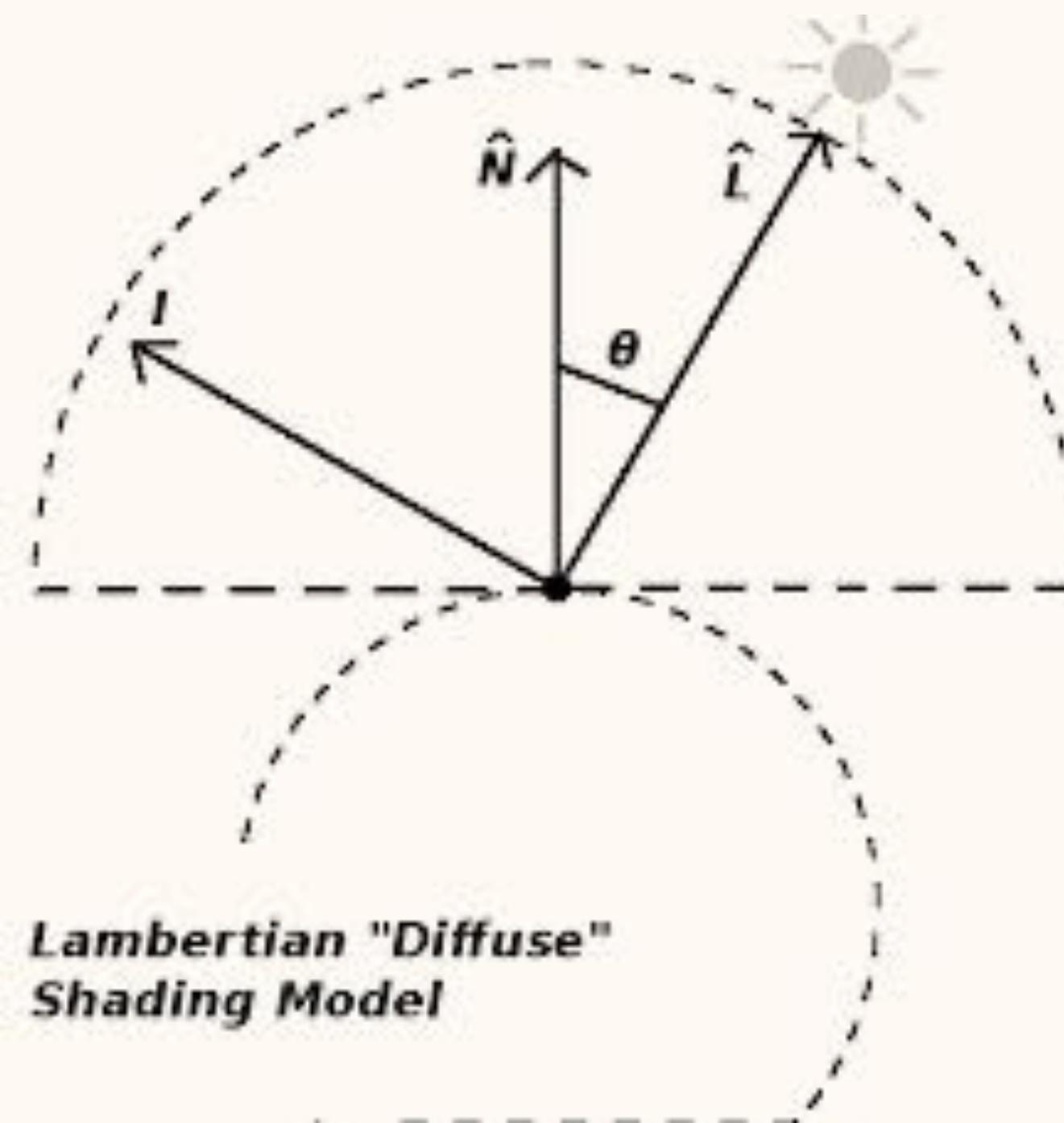
$$I_b = \frac{1}{y_1 - y_4} (I_1(y_s - y_4) + I_4(y_1 - y_s))$$

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

Do lighting in vertex shader
Per-vertex Lighting

1st Shader : Gouraud Shader

- Gouraud Shading
- Vertex Lighting
 - Lighting in vertex shader
- “Lambertian” Shading Model
 - Diffuse only



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
    float3 normW = normalize(mul((float3x3) mWorld, in1.inNorm));

    // prepare the lighting direction 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)
    out1.color = mainLightColor*diffuseInty*dif + 0.5*mainLightColor*amb;

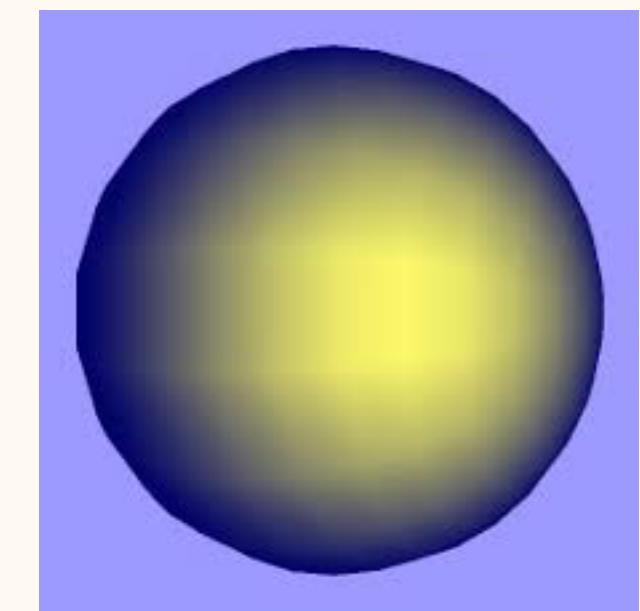
    return out1;
}
```

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

- The first scheme that overcame the disadvantages of constant shading of polygons uses bilinear intensity interpolation.
- Simple
- Economic
- Suffering from “Mach Banding”
- Restricted to the diffuse
 - Error rendering for specular component
- Fixed function rendering pipeline using Gouraud shading + specular
 - To prevent error rendering for low-resolution models, turn off the specular rendering state.



Normal Vector Transformation

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

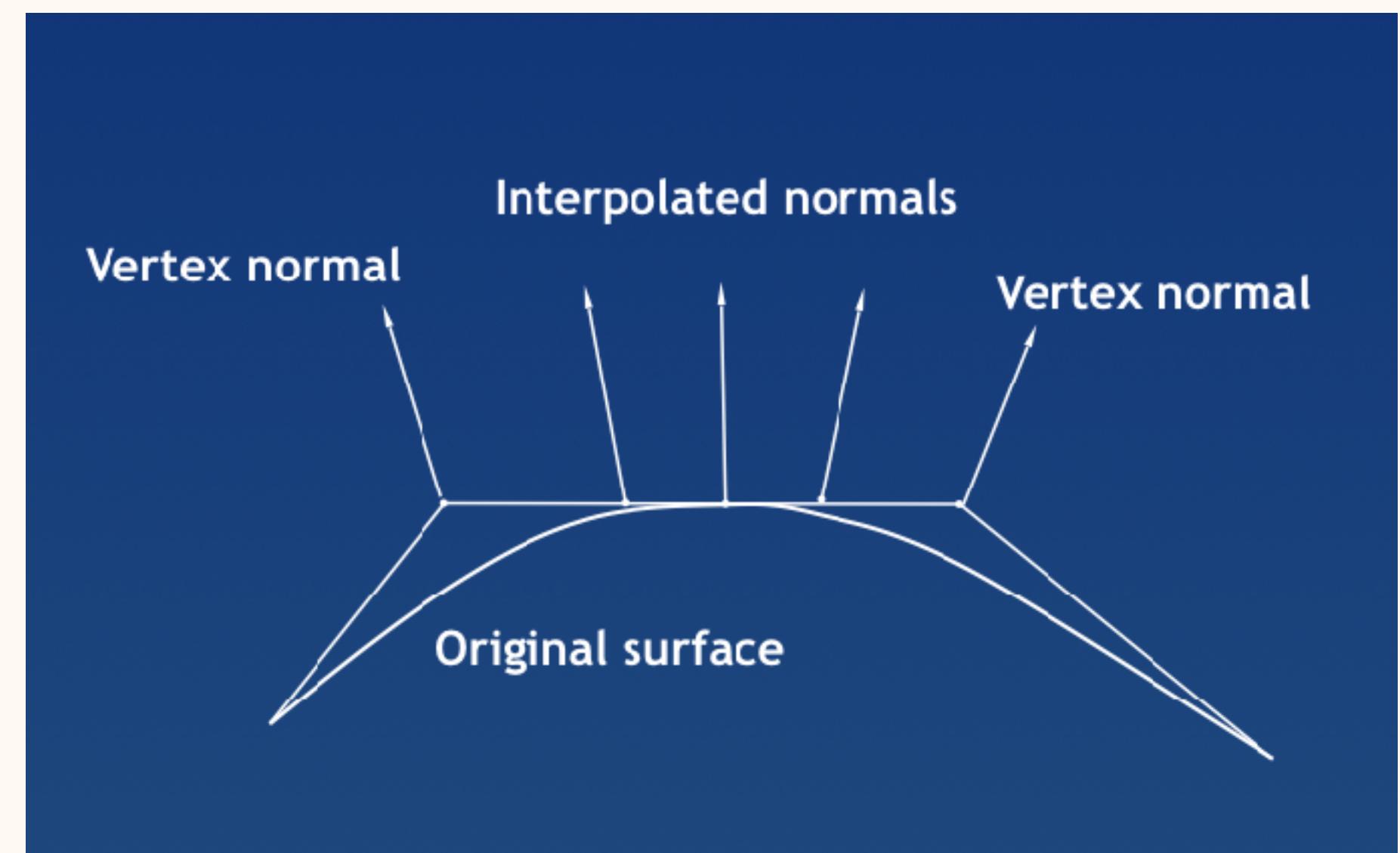
$$N = (M^{-1})^T$$

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

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

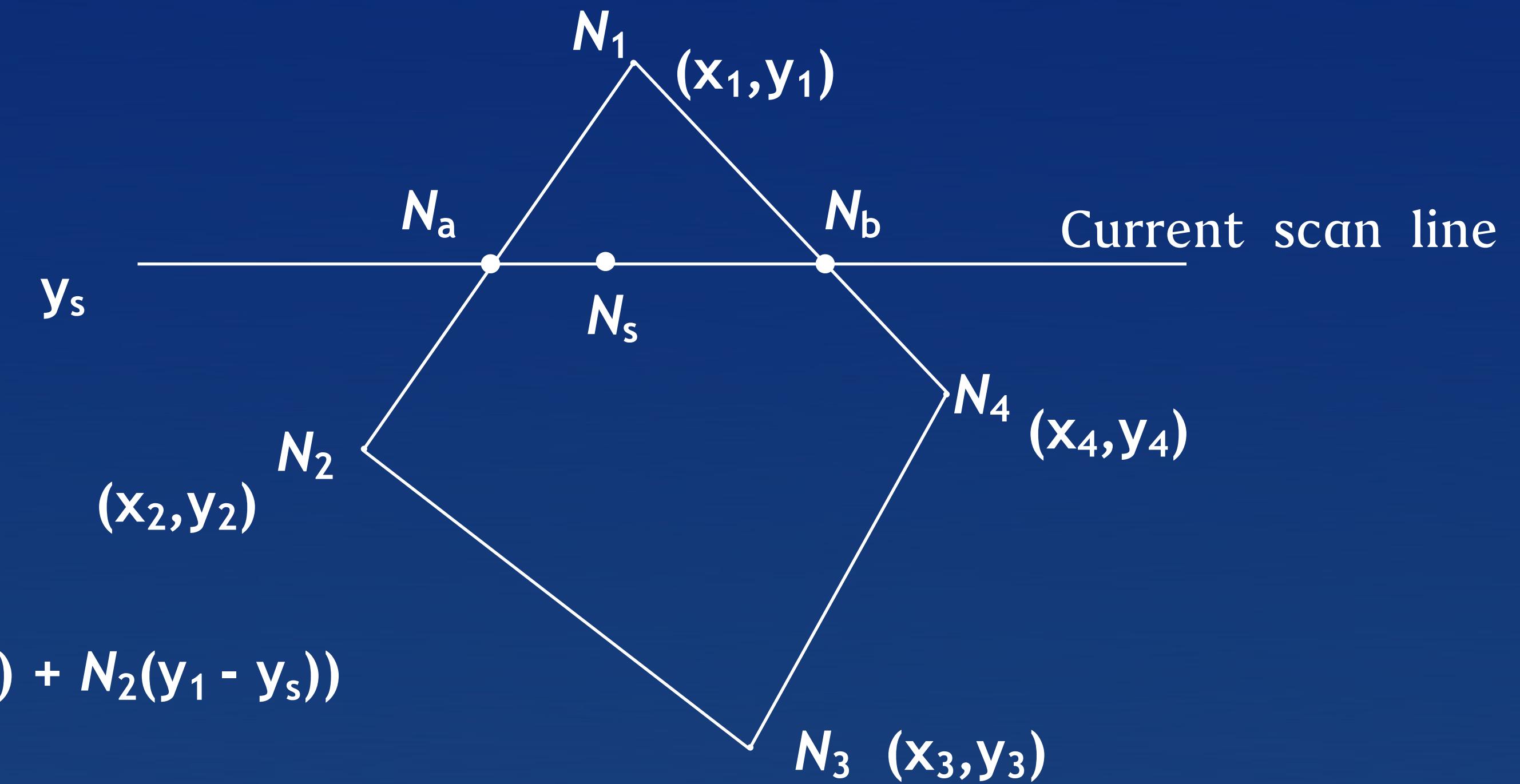
Phong Shading

- A method due to Bui-Tuong Phong (1975a) overcomes some of the disadvantages of Gouraud shading, and specular reflection can be successfully incorporated in the scheme.
- Features :
 - The attributes interpolated are the vertex normals, rather than vertex intensities.
 - A separate intensity is evaluated for each pixel from the interpolated normal.



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
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;

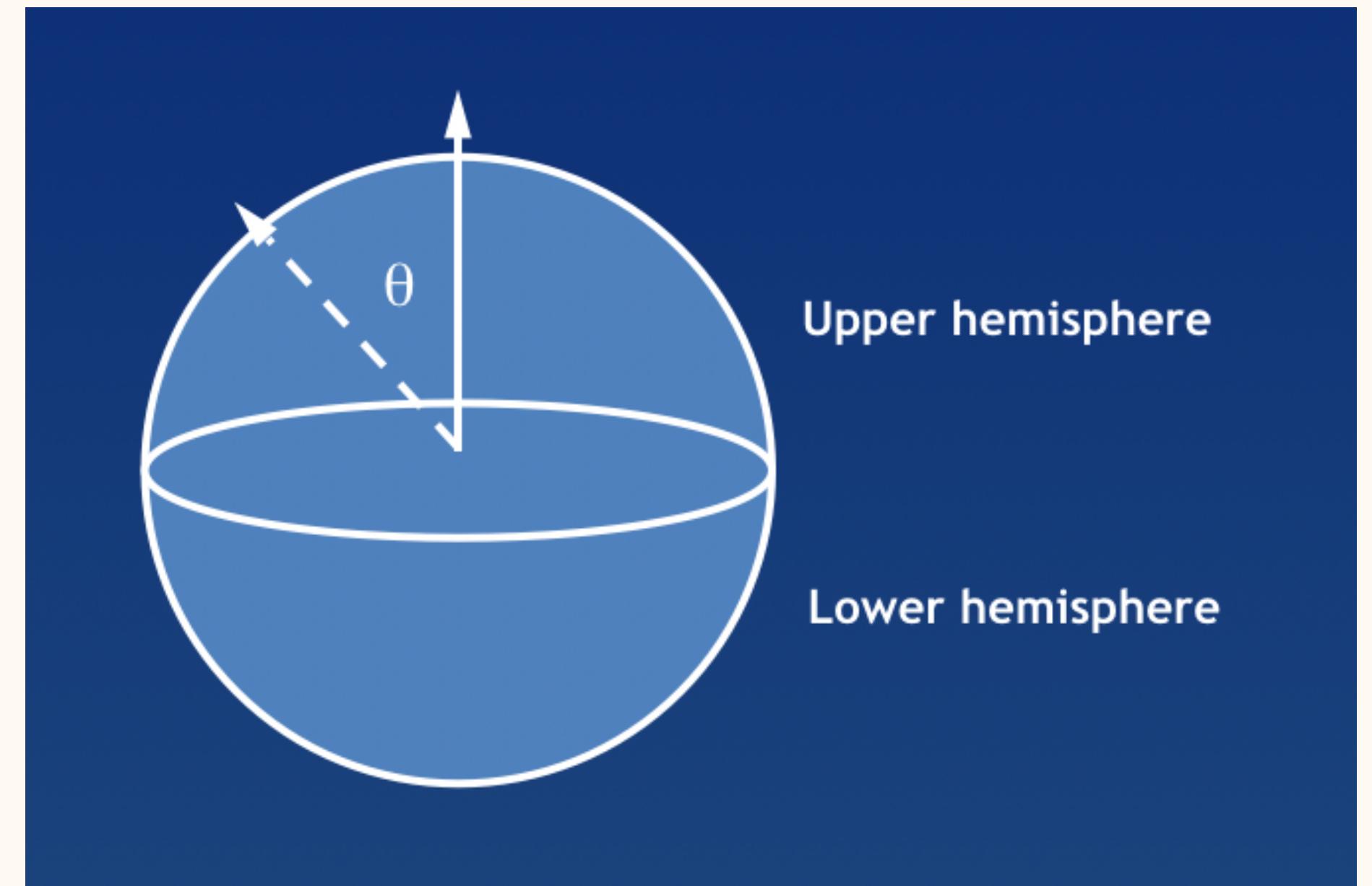
    return rgba;
}
```

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 updirection

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;
}
```

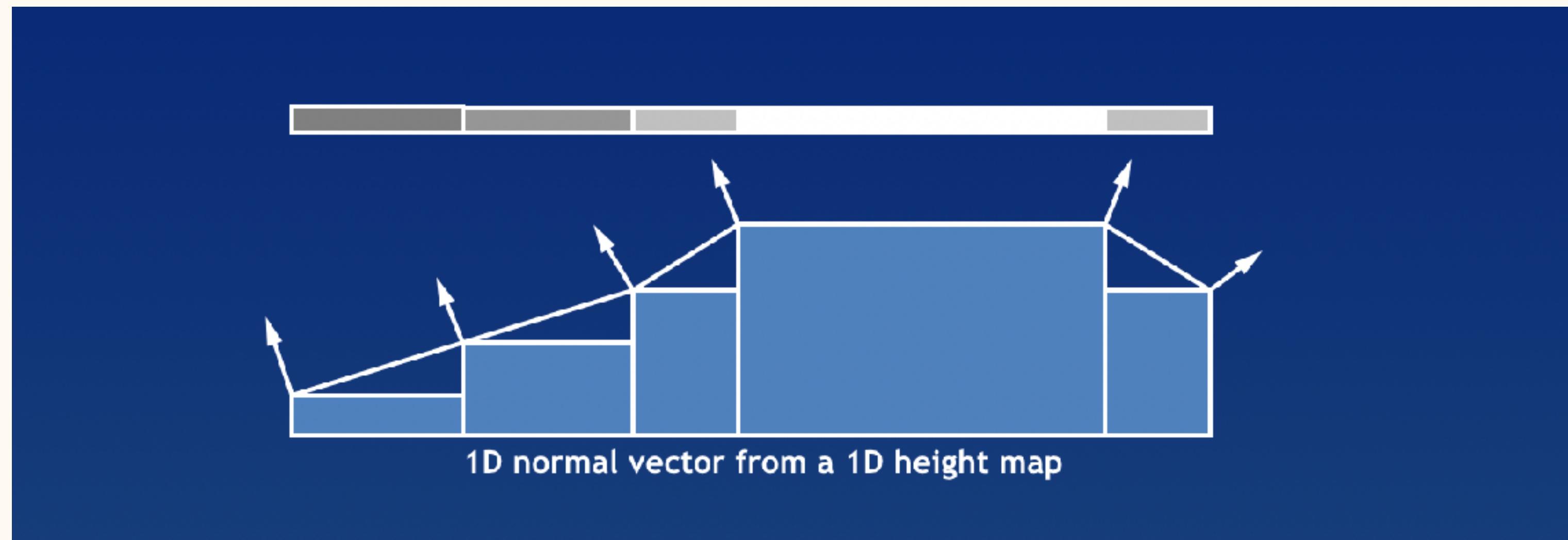
Normal Mapping

Introduction to Normal Mapping

- UseBy Jim Blinn, 1978
 - “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.
 - A fake solution
- “Bump Map”
 - Save height derivatives in textures : $(\Delta h/\Delta u, \Delta h/\Delta v)$
- “Displacement Map” or “Height Map”
 - Save relative height in textures
- Now popular in “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.



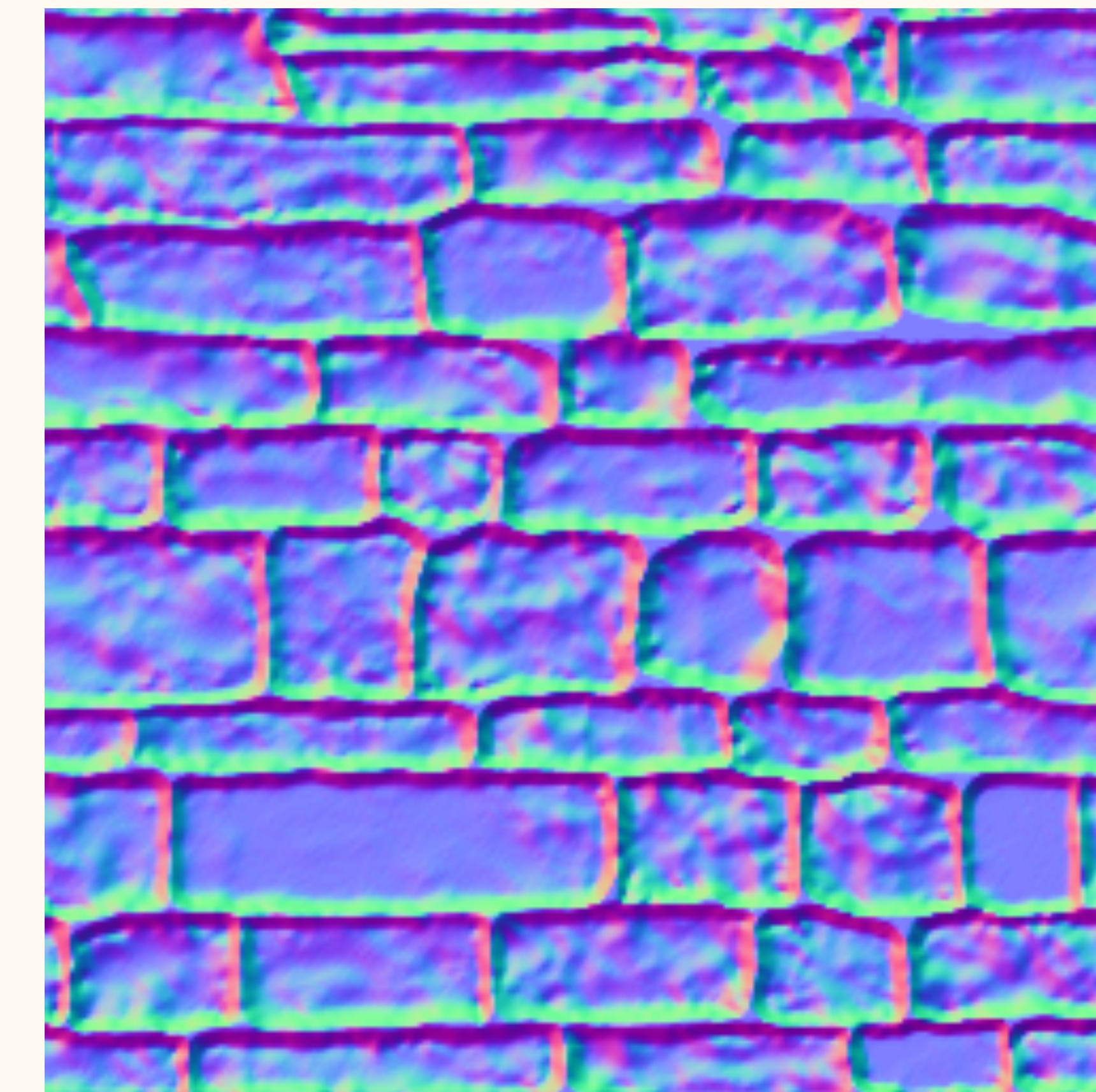
Introduction to Normal Mapping

- Two solutions
 - 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



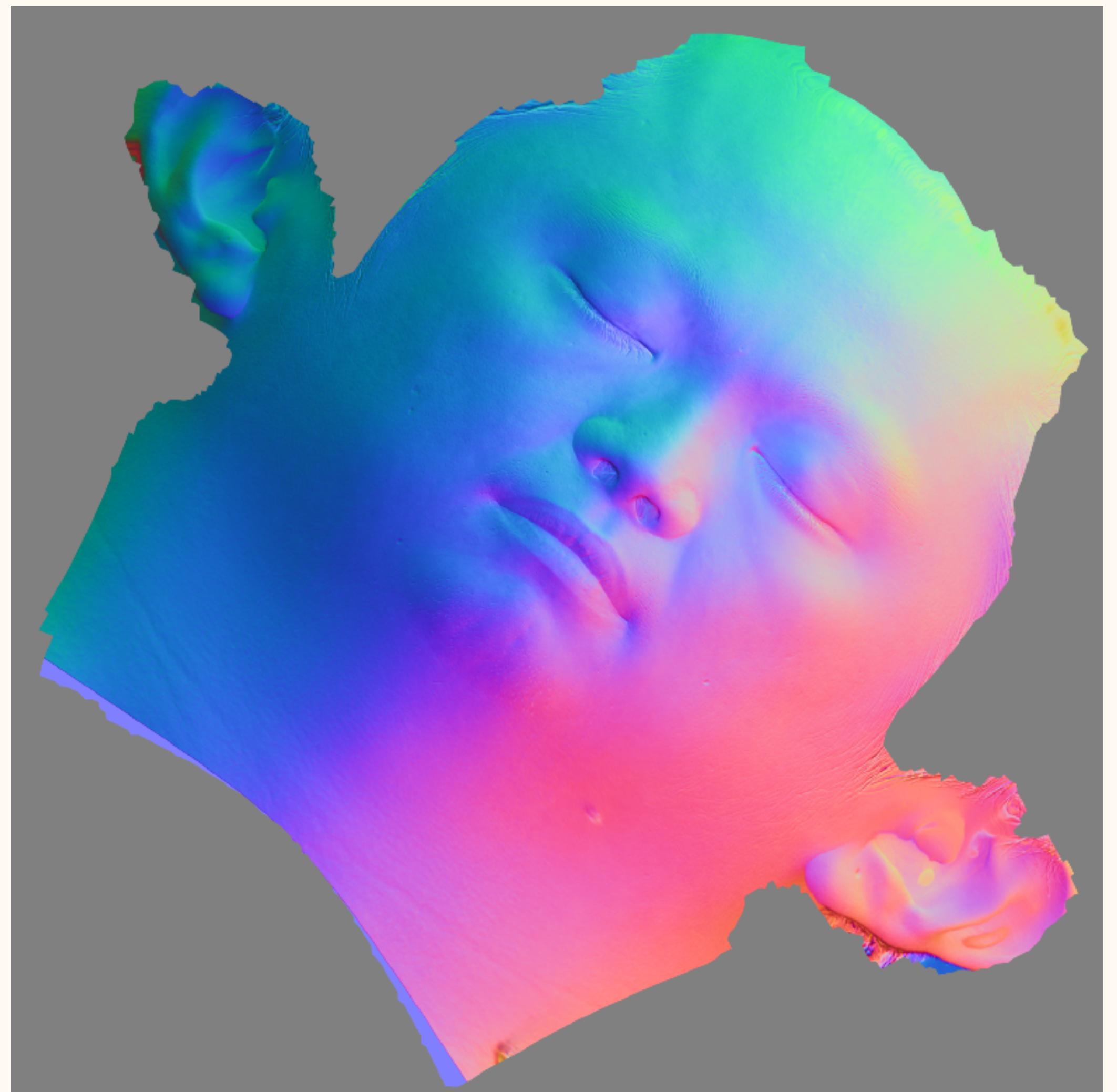
Height Map



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)
 - $\text{float3 pNorm} = 2.0 * \text{tex2D(normalMapSampler, tex0)} - 1.0;$
 - 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
 - Additional transformation in pixel shader

Tangent-space Normal Map

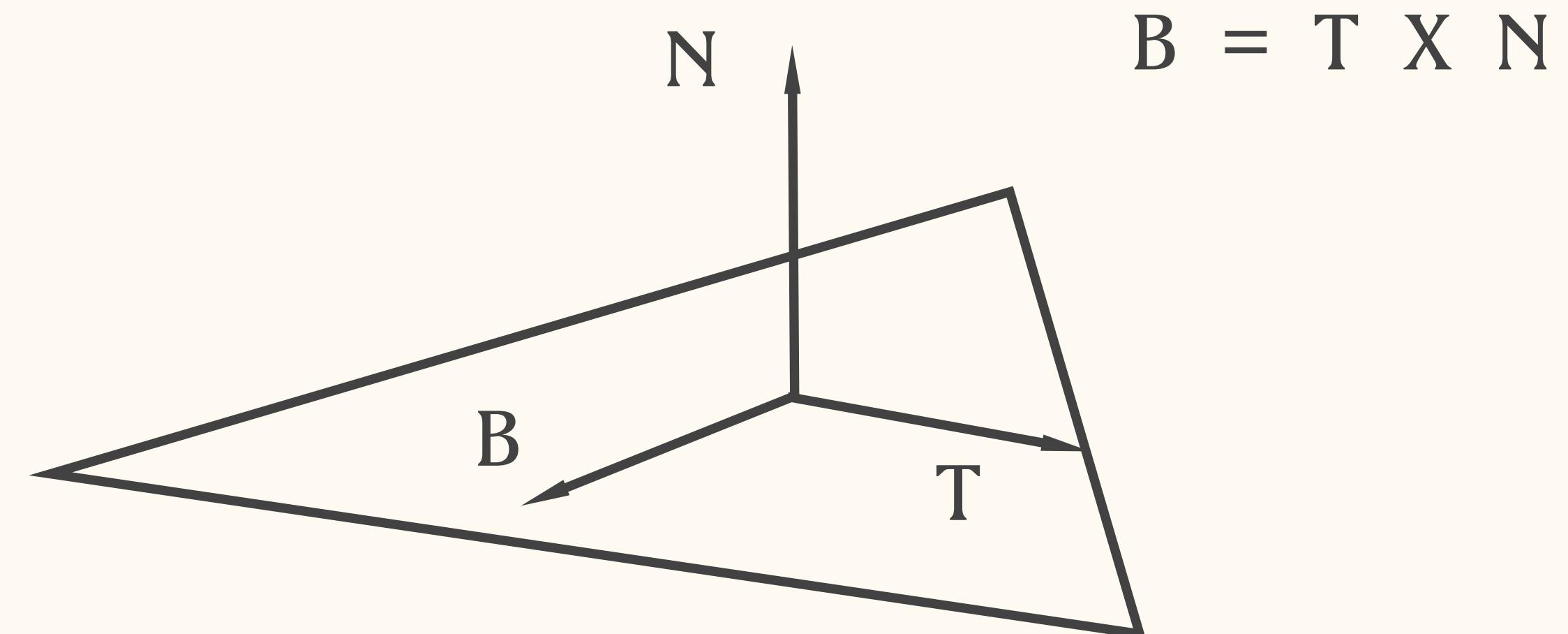
- $(r, g, b) = (N_x, N_y, N_z)$, ranging from 0.0-1.0
- Disturbance of normals in tangent space
- Need tangent and bi-normal vectors
- Geometry information hiding in tangent vector



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

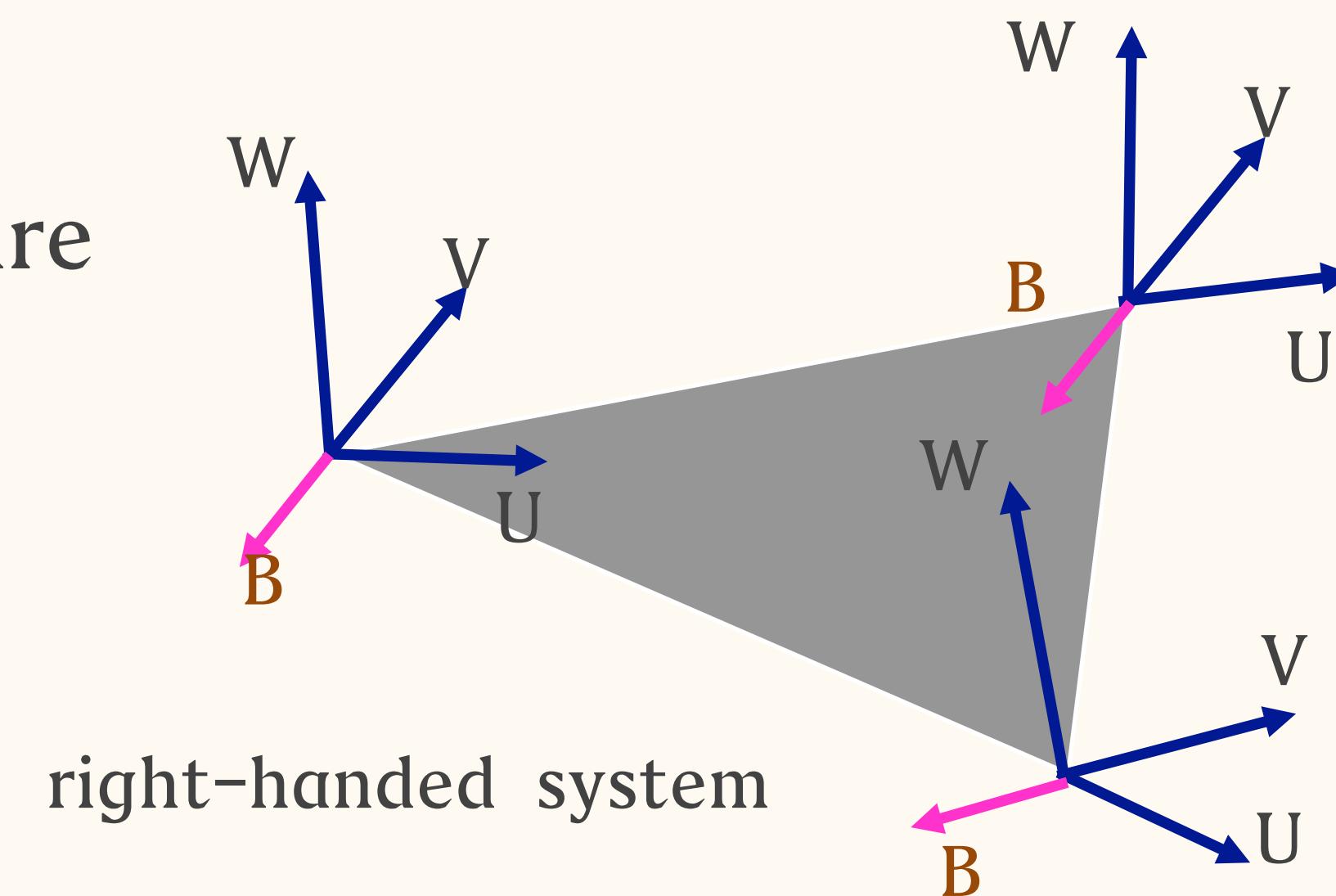


Tangent-space Normal Map

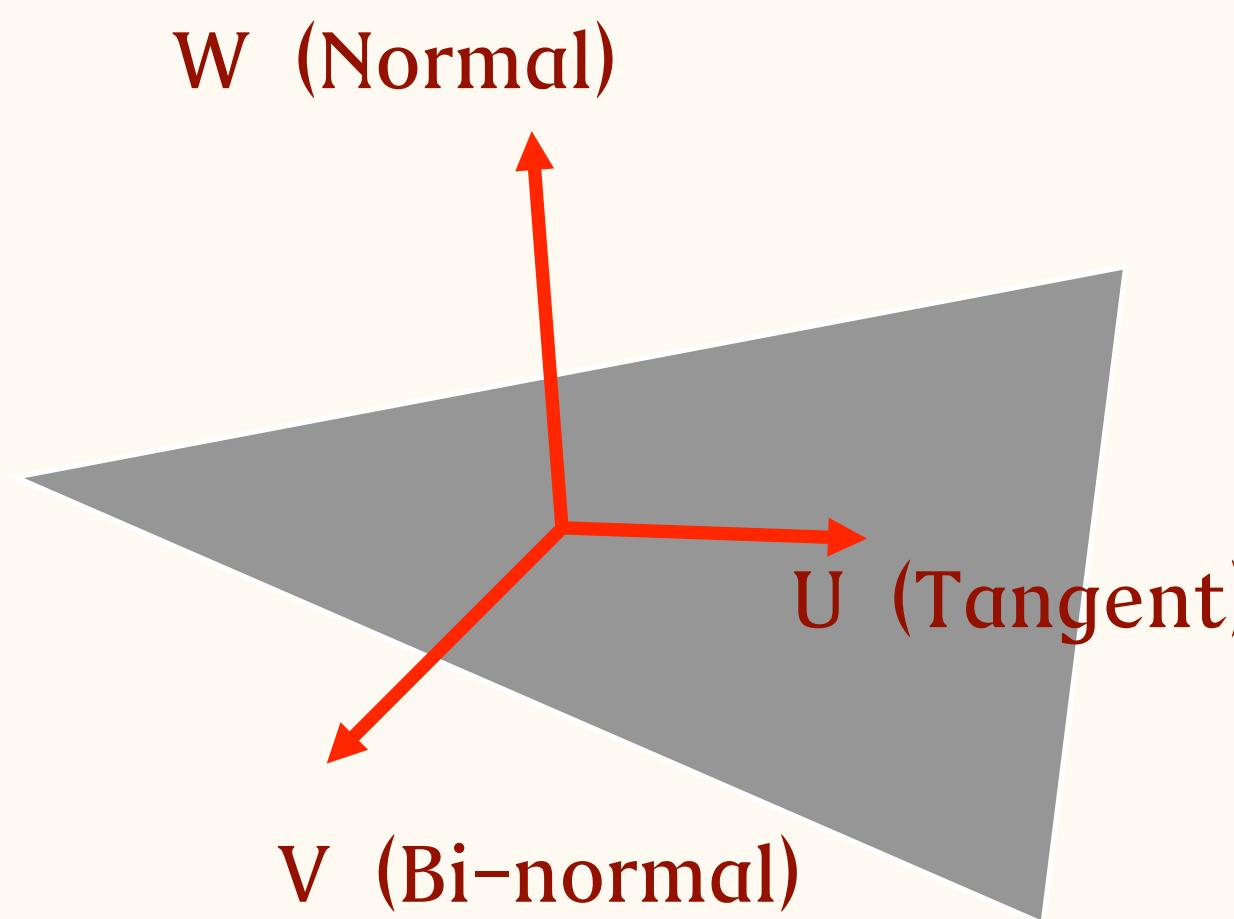
- The normal vector is in tangent space or texture space.
- Need a tangent vector to create the 3x3 matrix to convert the light vector and camera vector from world space to texture space.
- The tangent vector is calculated by the texture coordinates of the triangle vertices.

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);

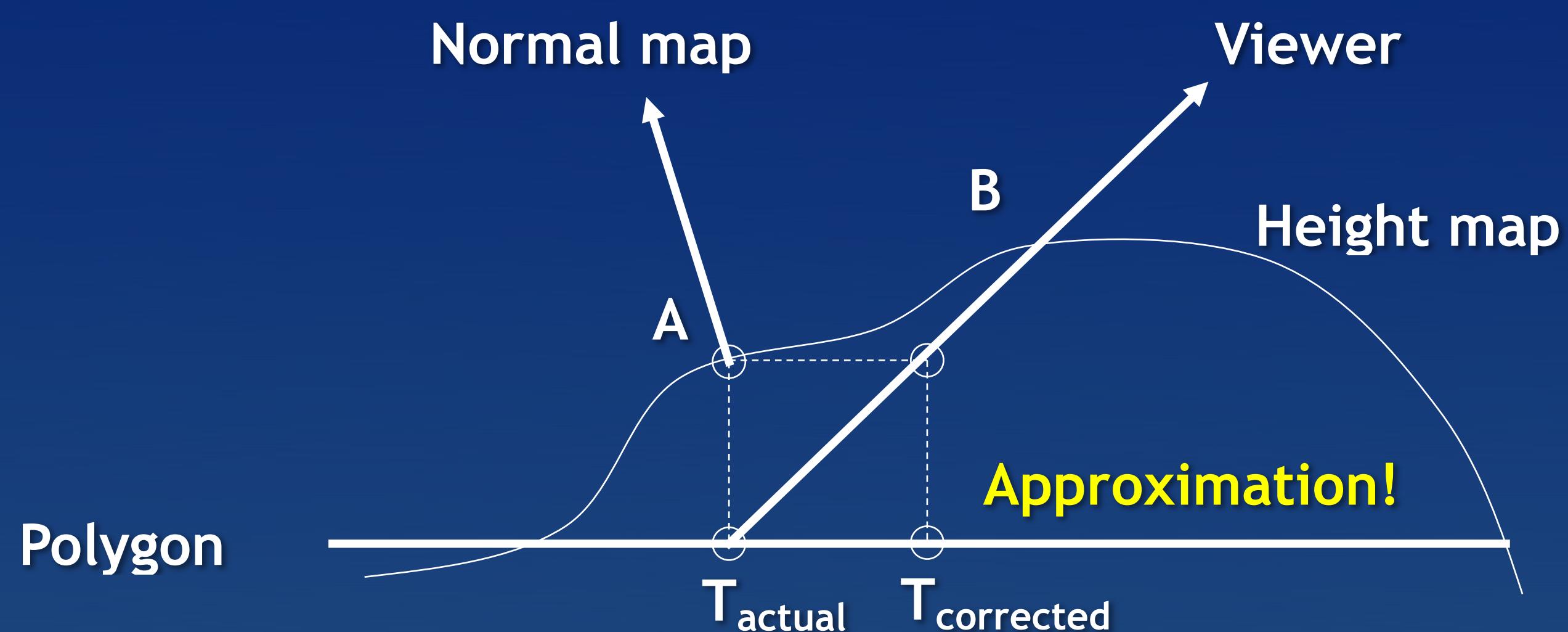
float4 imgAmb = lerp(groundLight, skyLight, dot(normDir, upDir)*0.5 + 0.5);

// 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

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

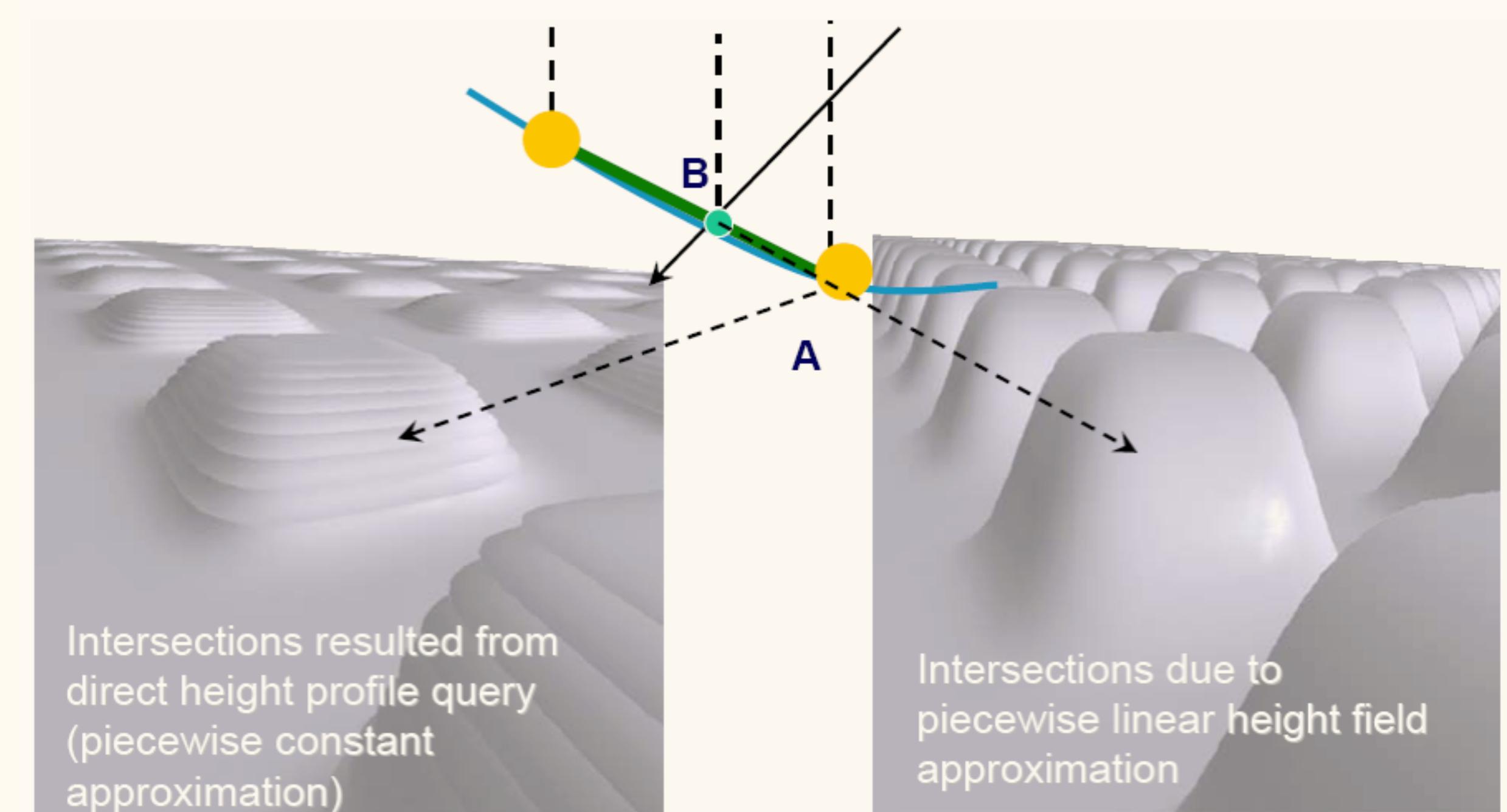
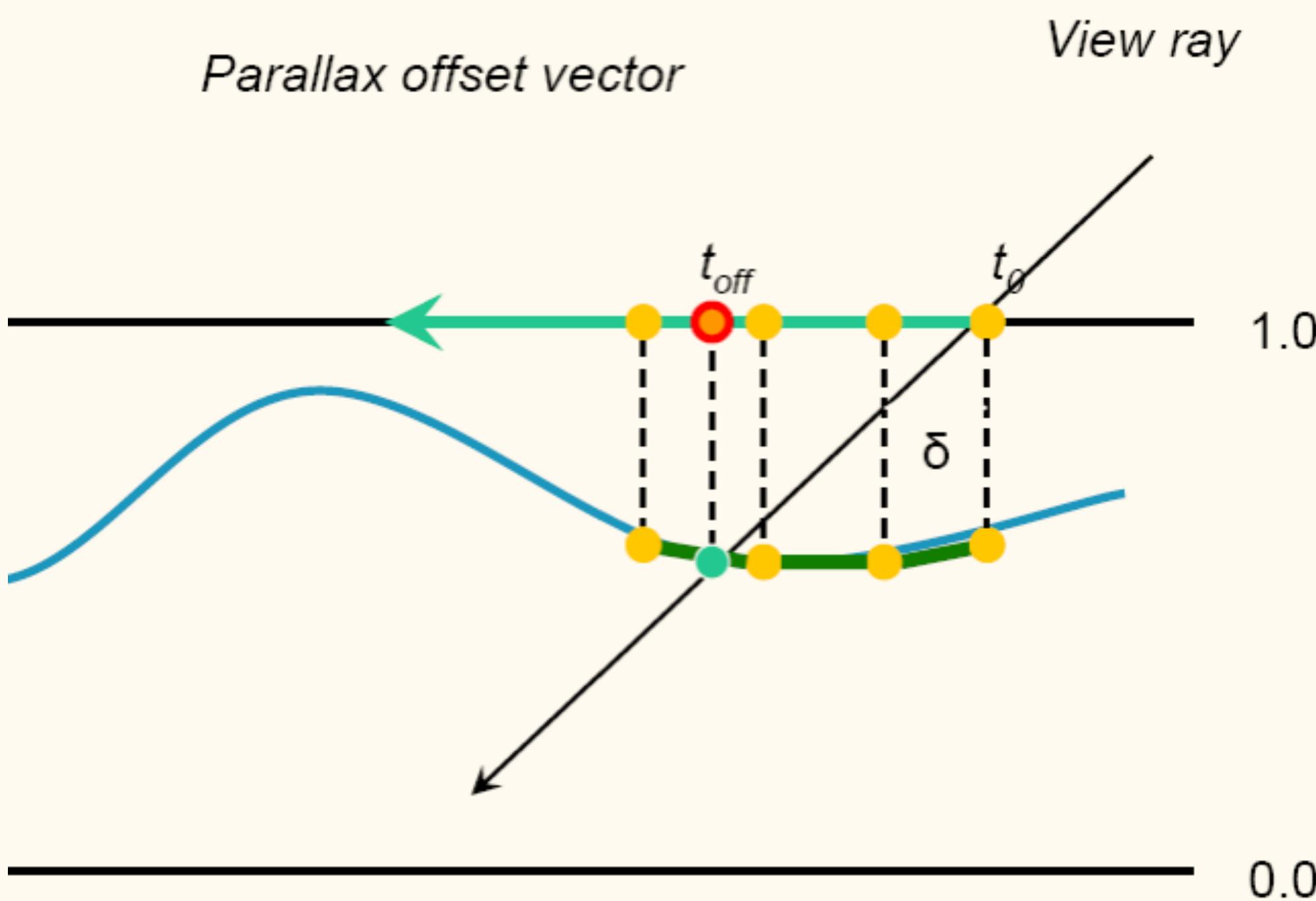
$$h_{\text{sb}} = (h_{xs}) + b$$

s = scale factor

b = bias factor

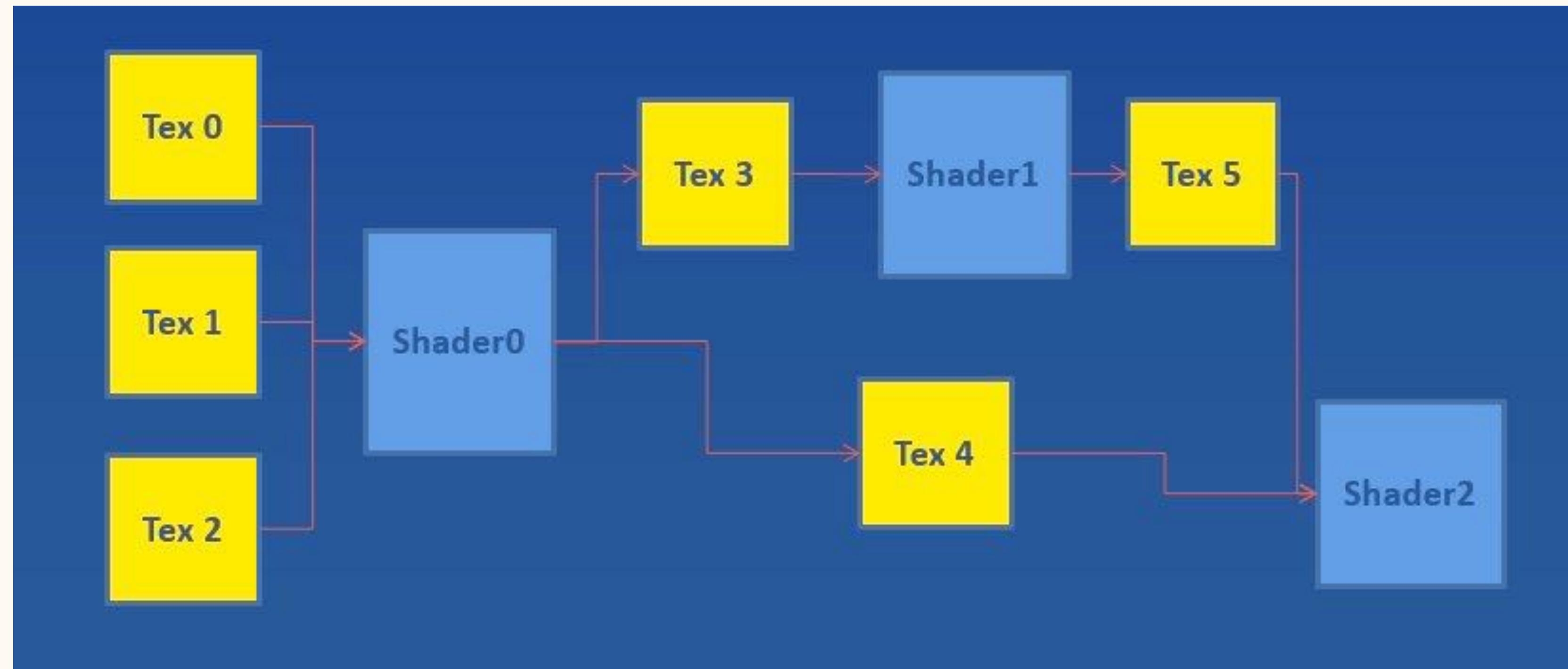
More Normal Map Solutions

- 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



Medium

Multi-pass Rendering



- Shader is a short/single-function program with highly parallelism
- Textures are saving in video memory and acting as a memory block
- We can render into textures (rendering targets)

Multi-pass Rendering

- We can divide the program into functions (shaders) and use textures as memory blocks.
- Use Draw command to trigger the shader to execute it
- Vertex shader and pixel shader are gathering data from textures
- Not broadcasting data to neighboring vertices and pixels

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

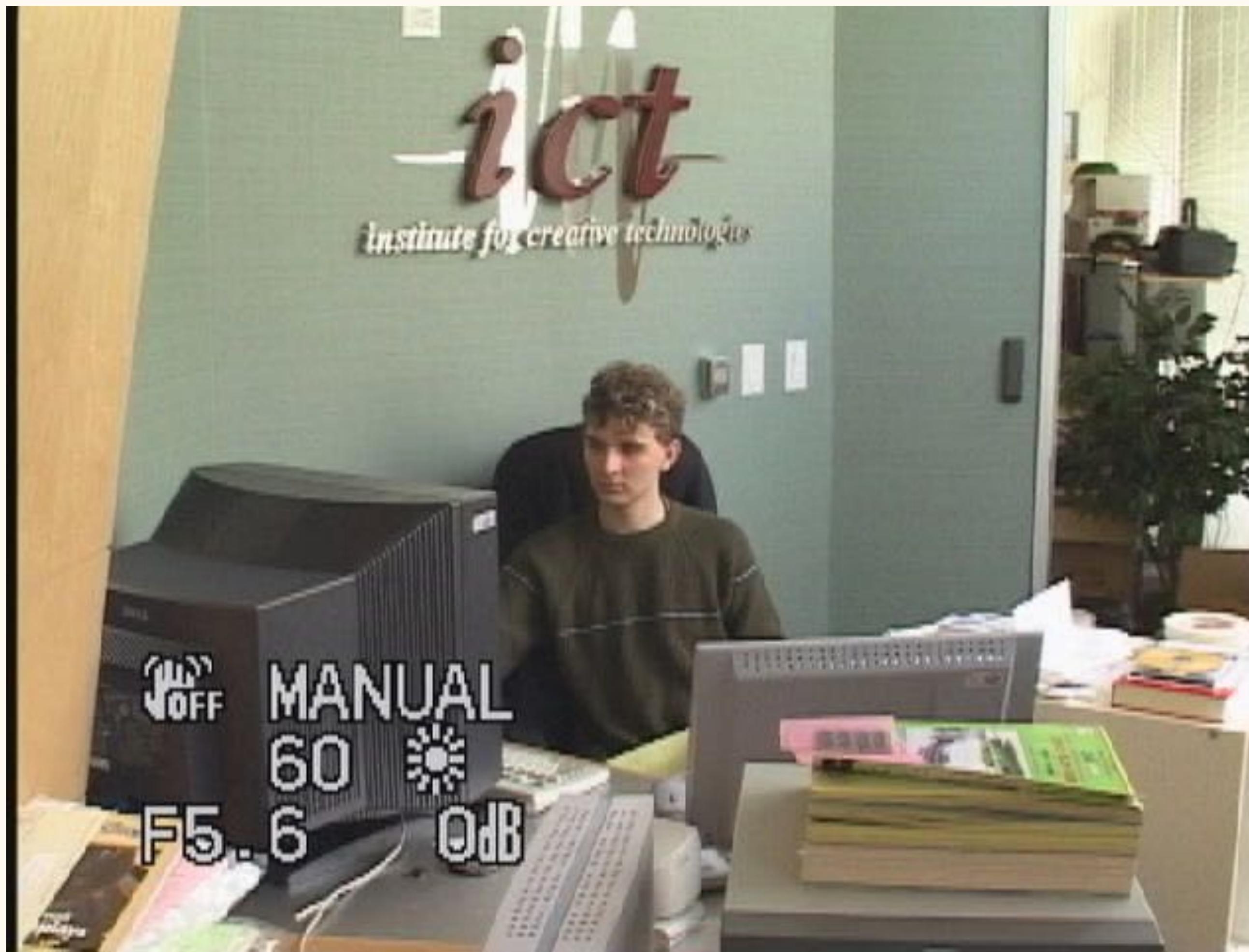
High Dynamic Range Imaging

HDR

- Dynamic range
 - The ratio of greatest value to smallest value that can be represented in color.
- Displayable image
 - Low dynamic range (LDR)
 - 2^8
 - Relative luminance (0.0 – 1.0)
- Natural phenomena
 - High dynamic range (HDR)
 - 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



Office interior

Indirect light from window

1/60th sec shutter

f/5.6 aperture

0 ND filters

0dB gain

HDR in Real World



Outside in the shade

1/1000th sec shutter

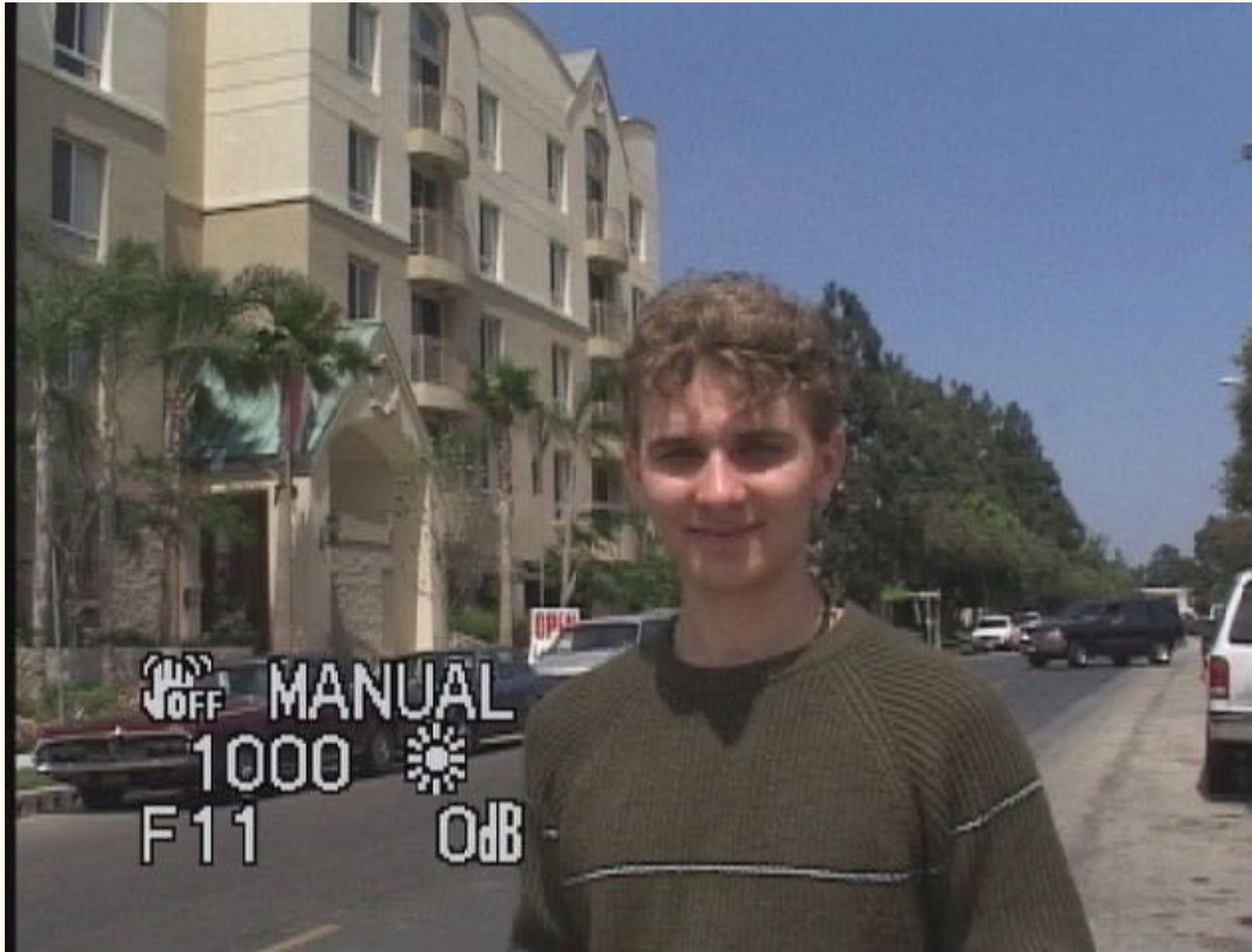
f/5.6 aperture

0 ND filters

0dB gain

16 times the light as inside

HDR in Real World



Outside in the sun

1/1000th sec shutter

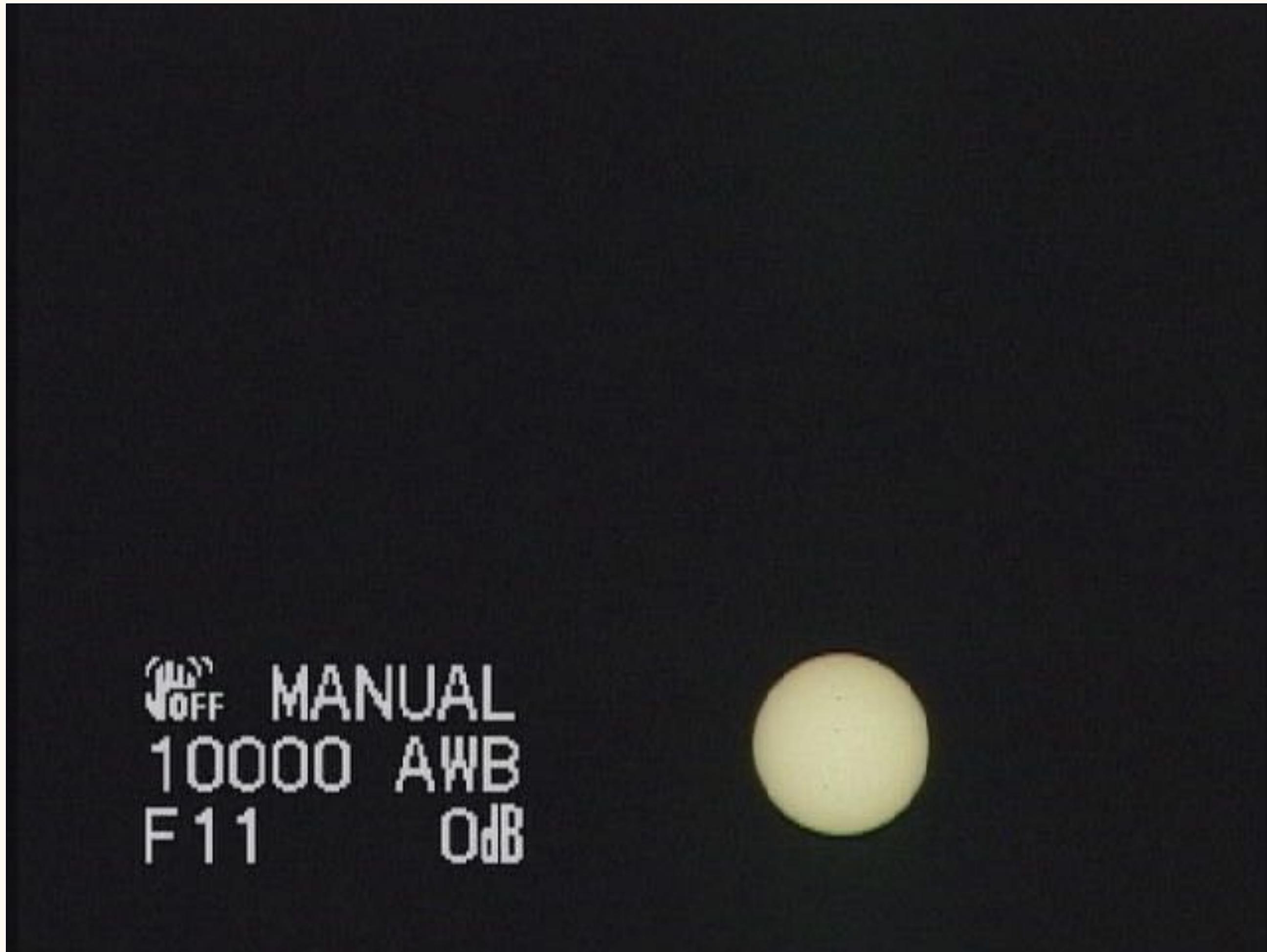
f/11 aperture

0 ND filters

0dB gain

64 times the light as inside

HDR in Real World



Straight at the sun

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



Very dim room

1/4th sec shutter

f/1.6 aperture

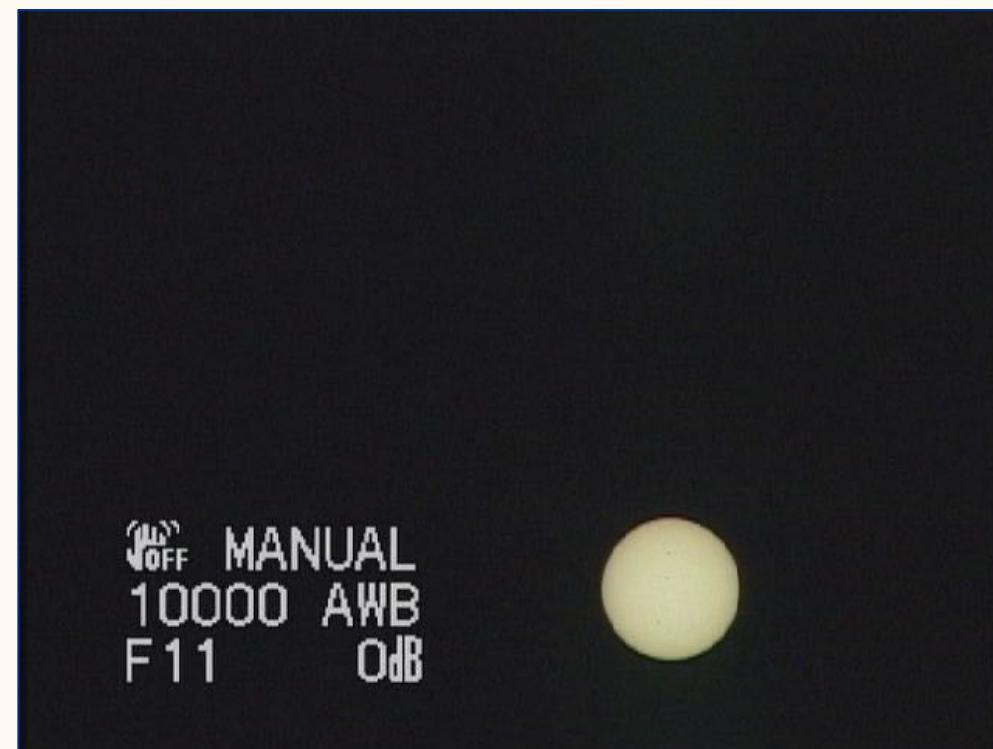
0 stops ND filters

18dB gain

OFF MANUAL
4 AWB
F1.6 18dB

1/1500th the light than inside

HDR in Real World



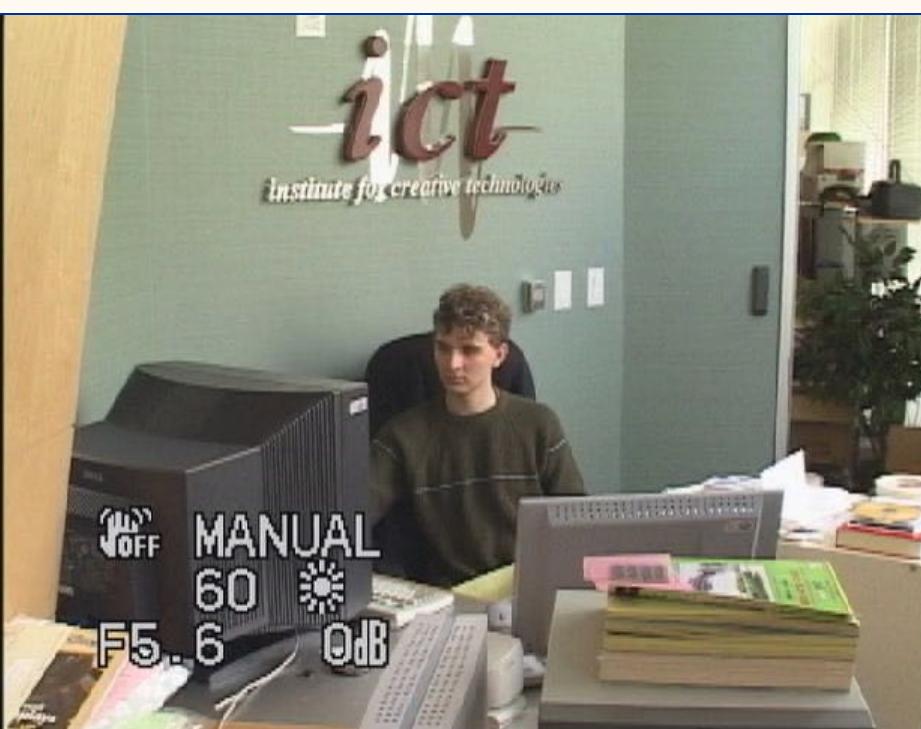
2,000,000,000



400,000



25,000

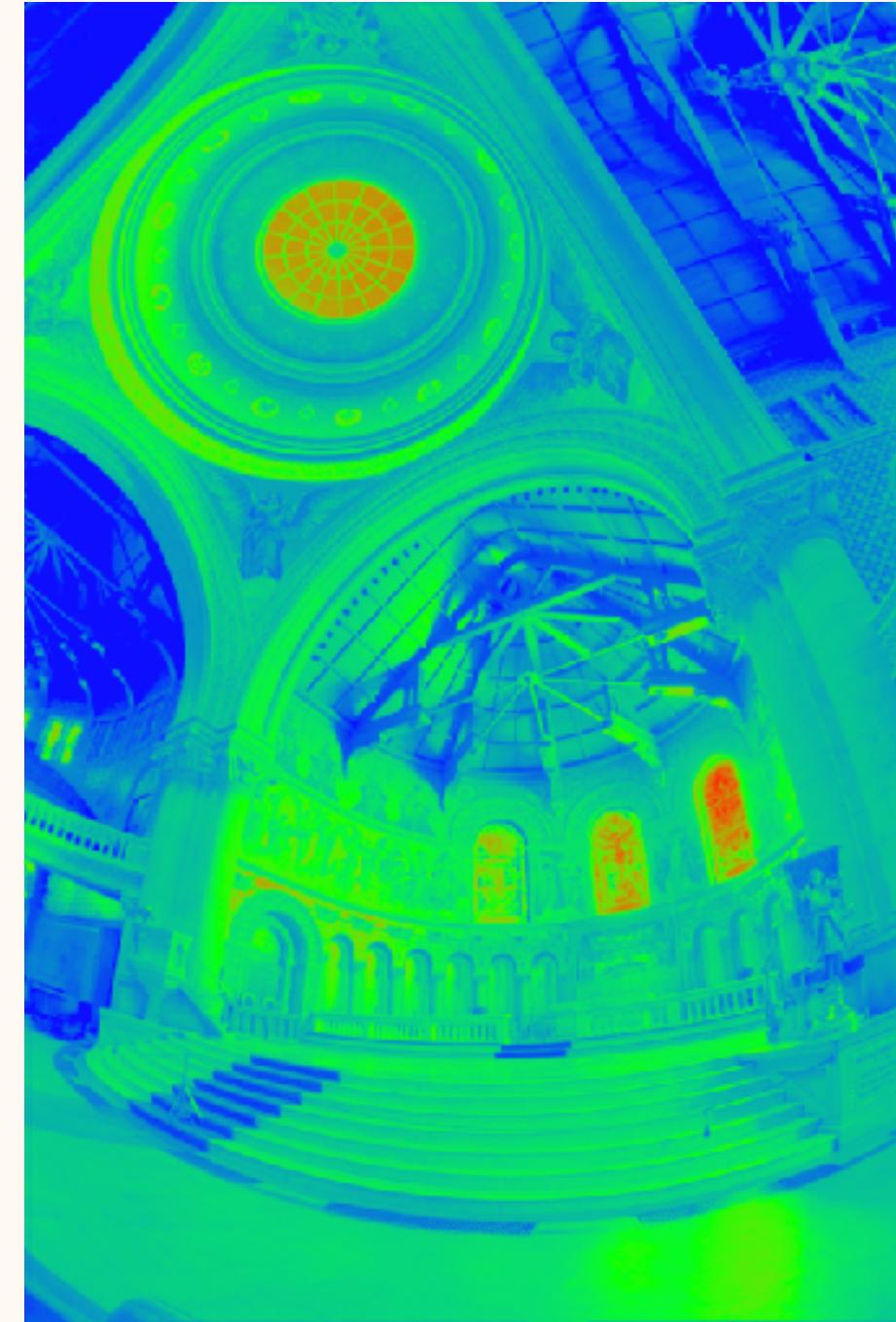


1,500



1

Generate HDR Image



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

300,000 : 1

HDR Applications in Games

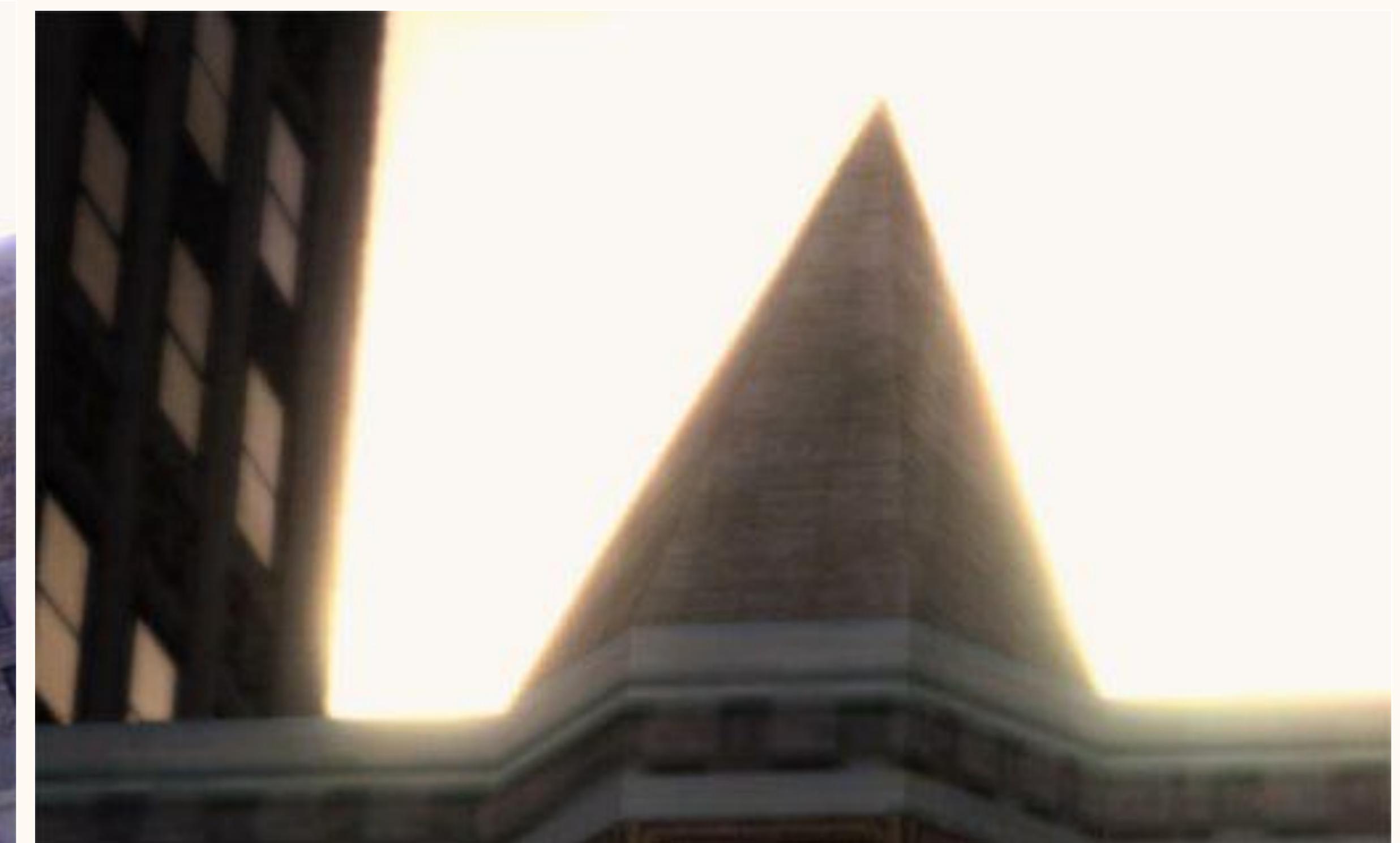
- HDR Post-processing Effect
 - Mapping rendered HDR image to LDR for display on screen
- Tone mapping
- Image-based Lighting
 - Take HDR image as the global light source
 - Simulate natural lighting

The First HDR Example

- RNL
 - "Render Natural Lighting"
 - SIGGRAPH'98 Electronic Theater
 - Rendered by RADIANCE
- AMD/ATI made it as a real-time demo for 9700 HW



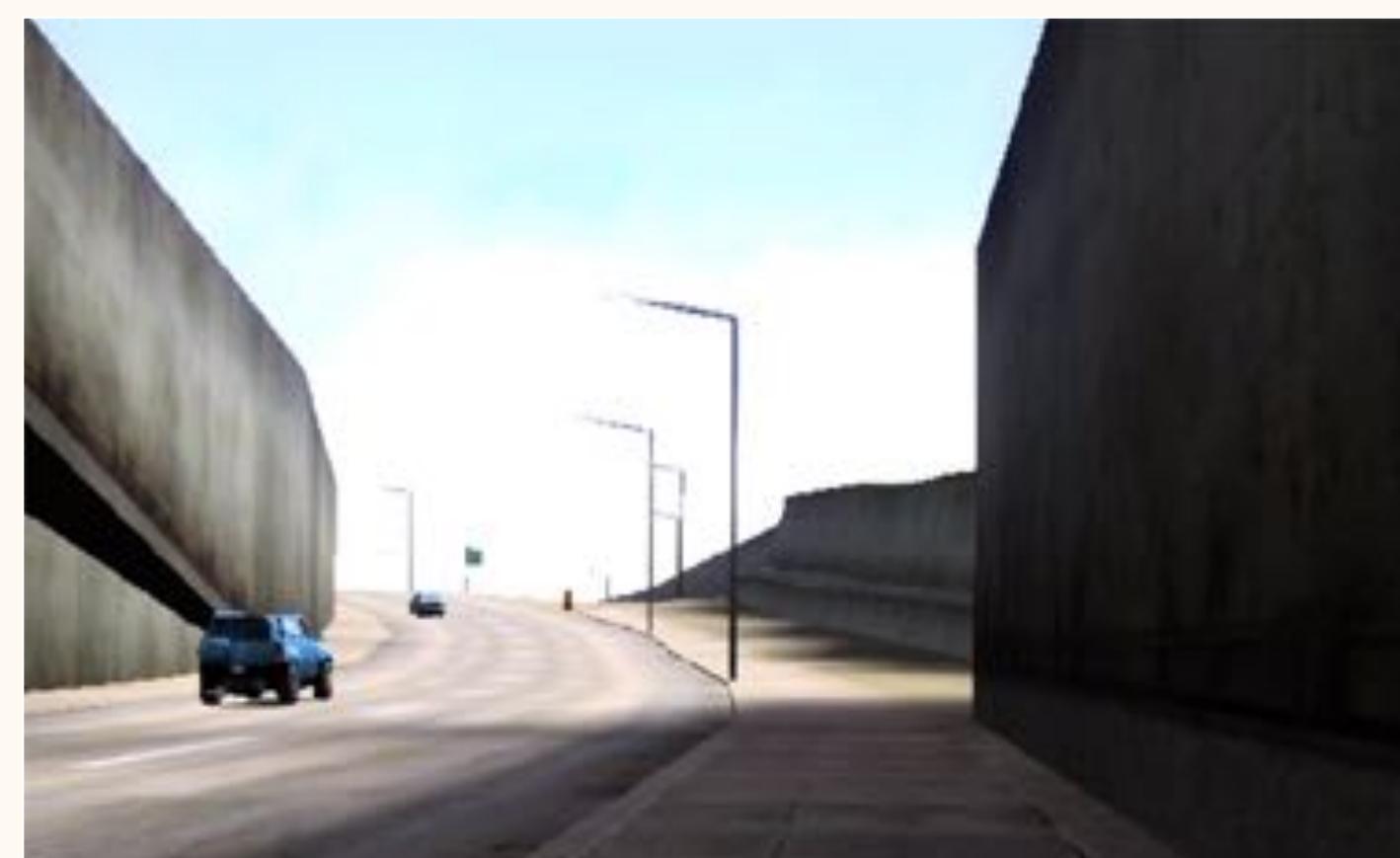
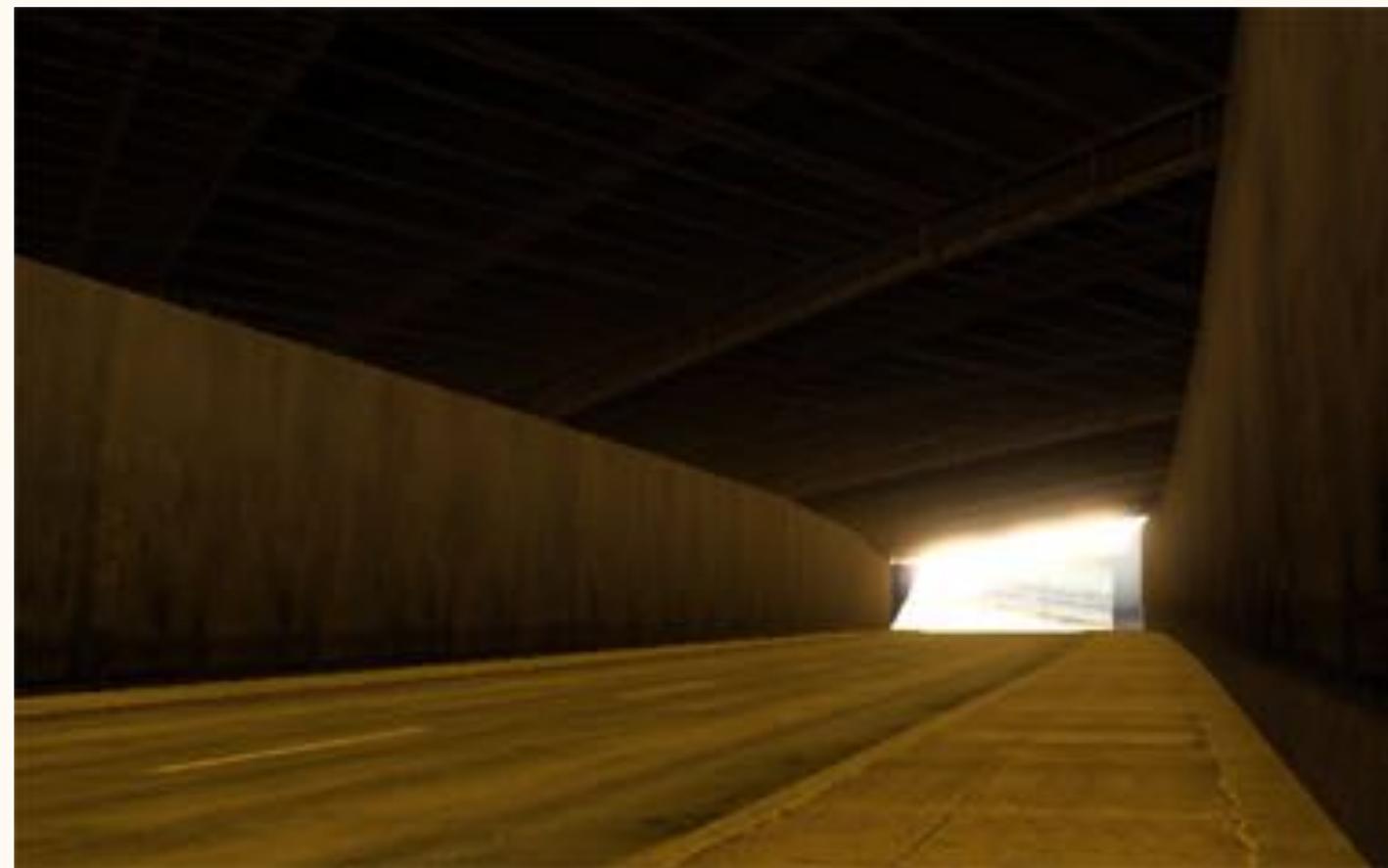
HDR Effects



Dazzling Effect

HDR Effects





Dynamic Tone Mapping

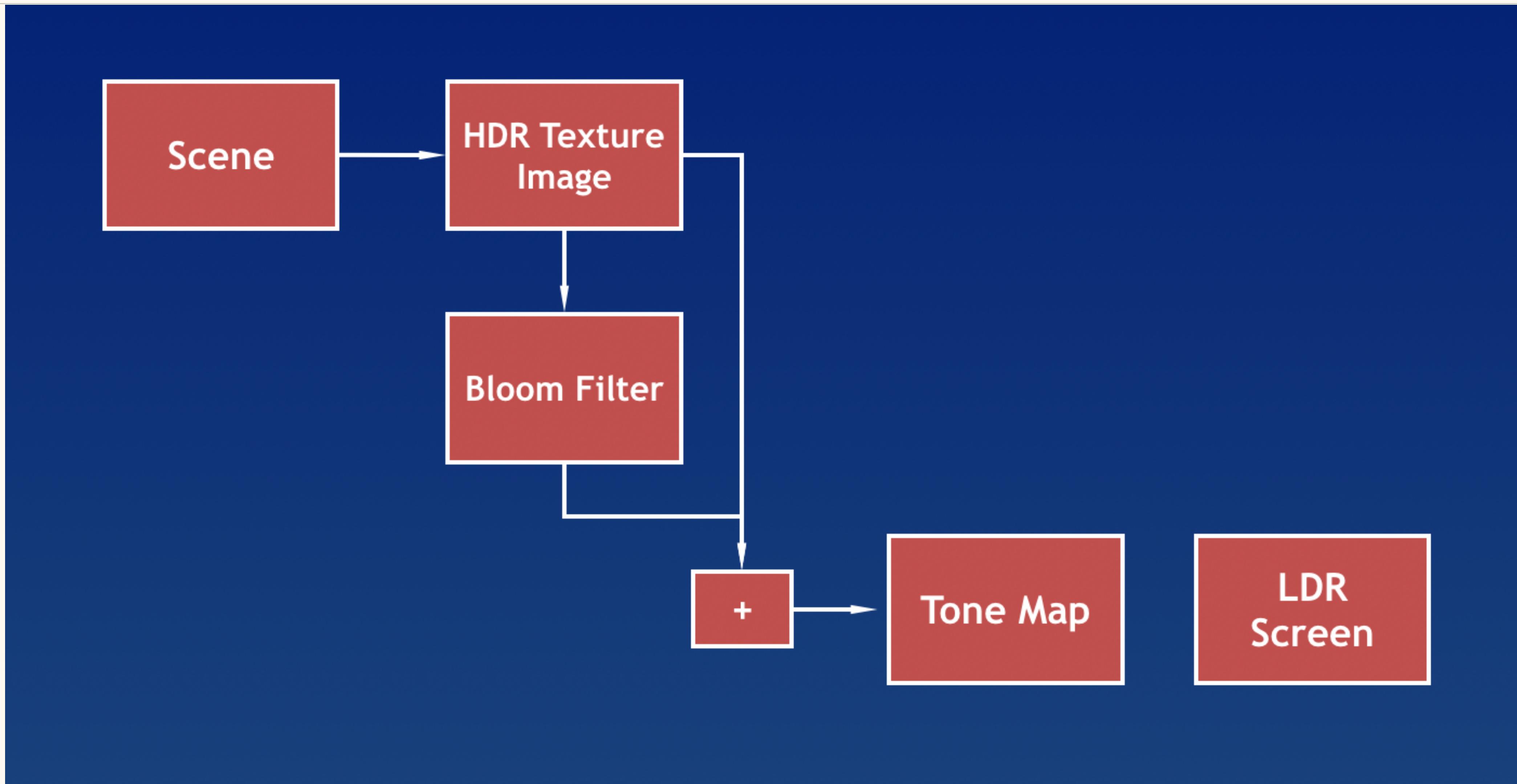
Half-Life 2 HDR Demo



Fixed Aperture

High-Dynamic Range

Real-time HDR Post-processing

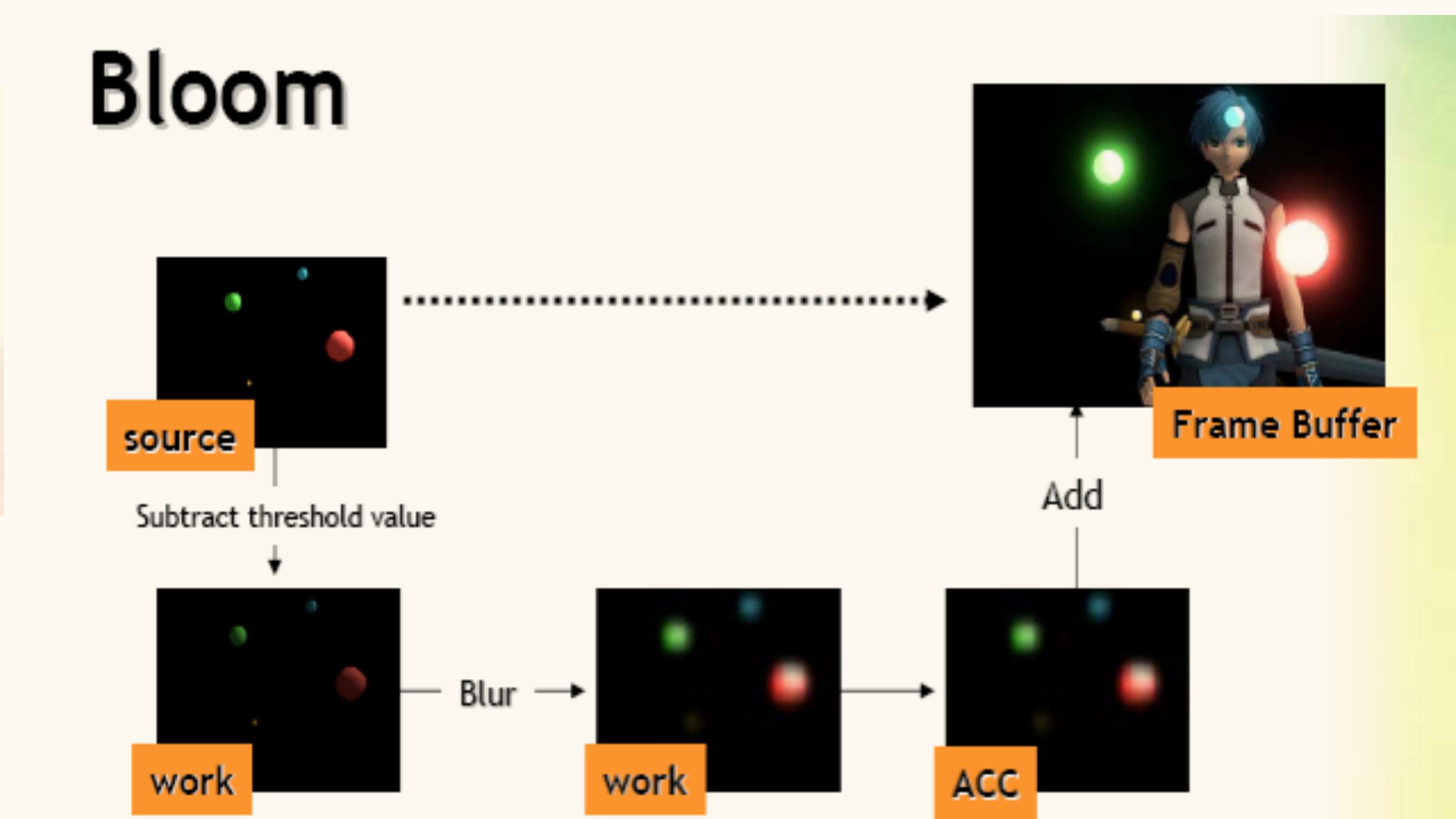
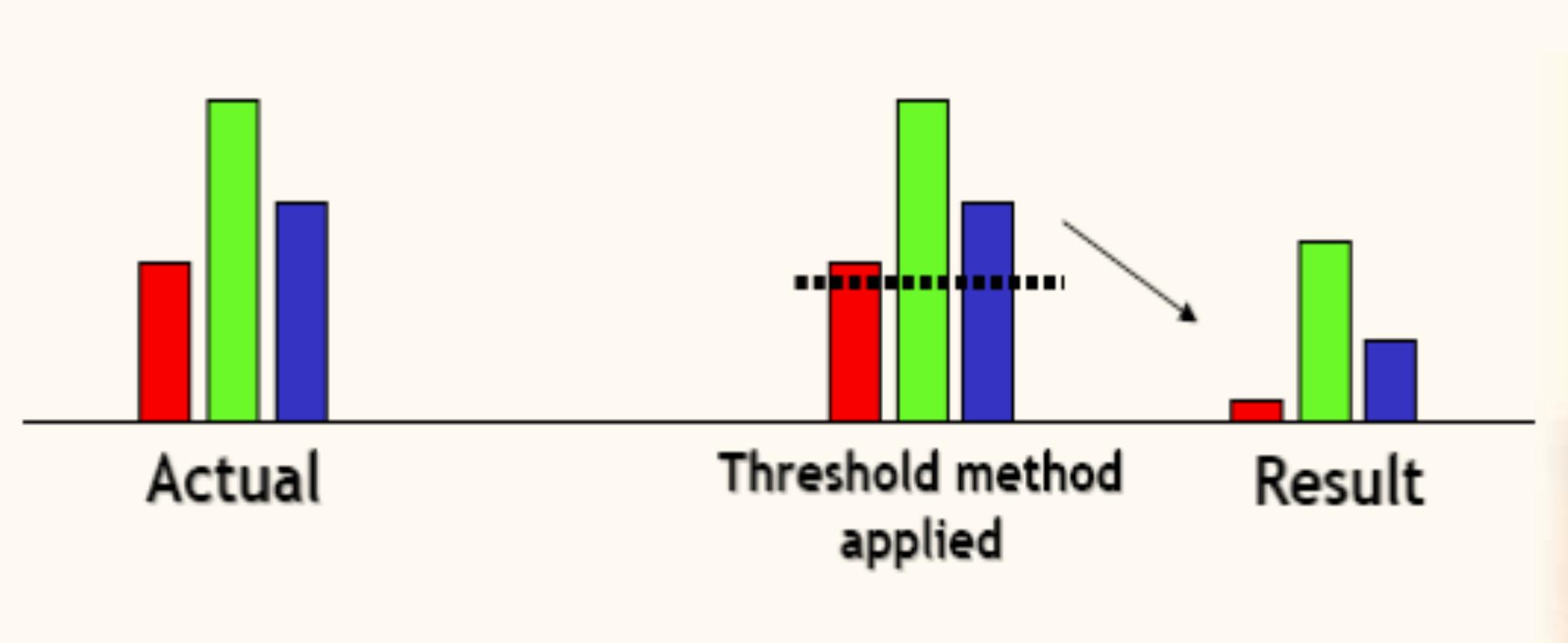


Real-time HDR – Step I

- Render the 3D scene on a HDR texture
 - RGB format on screen is LDR :
 - R8G8B8A8
 - $(0, 0, 0) \sim (255, 255, 255)$
 - HDR :
 - RGBA > 16 bits
 - floating-point real number
 - color value > 1.0

Real-time HDR - Step II

- Get the bright part & bloom it



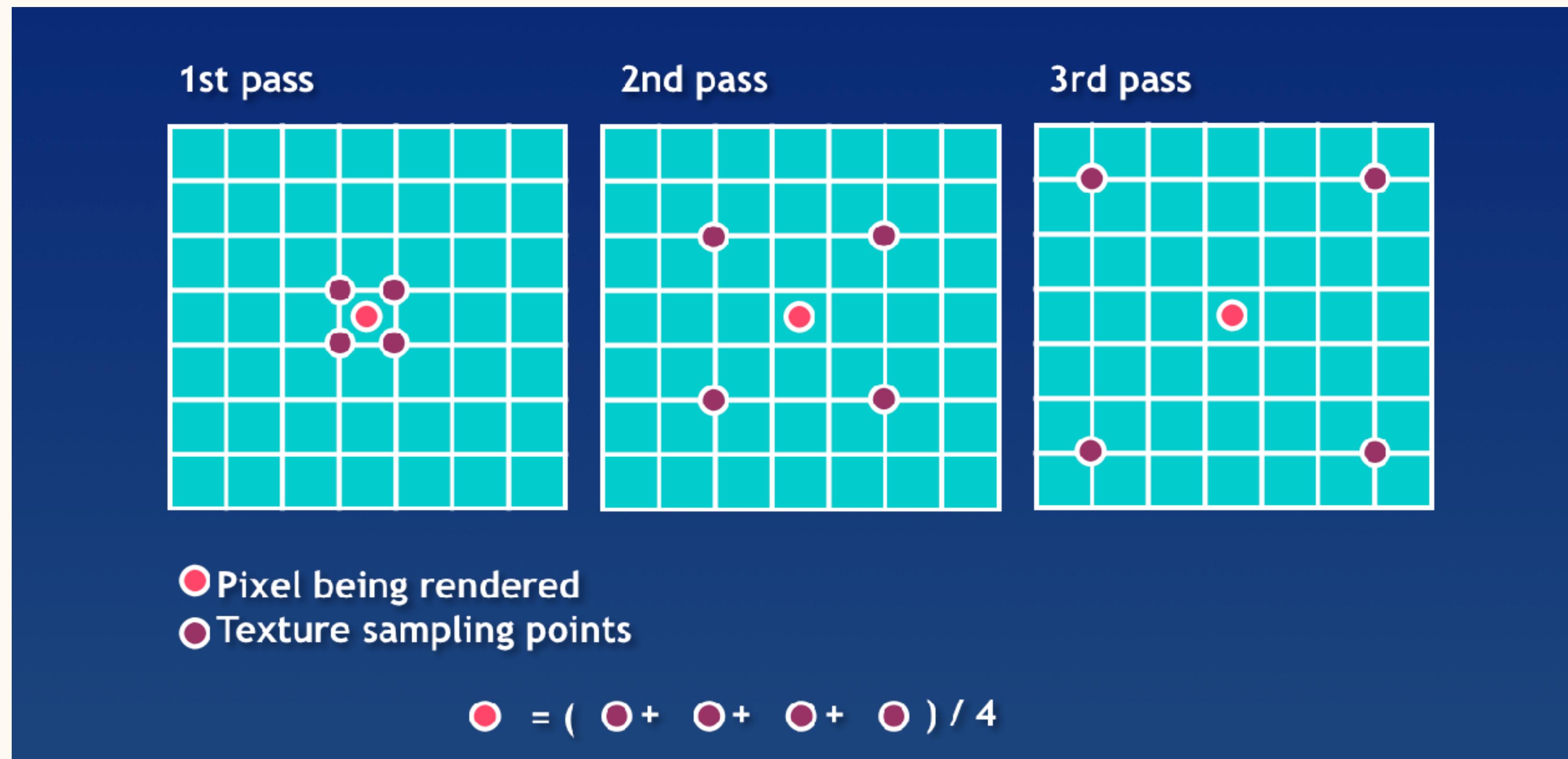
Real-time HDR - Step II

- Another way to get the bright pass :
 - 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;
```

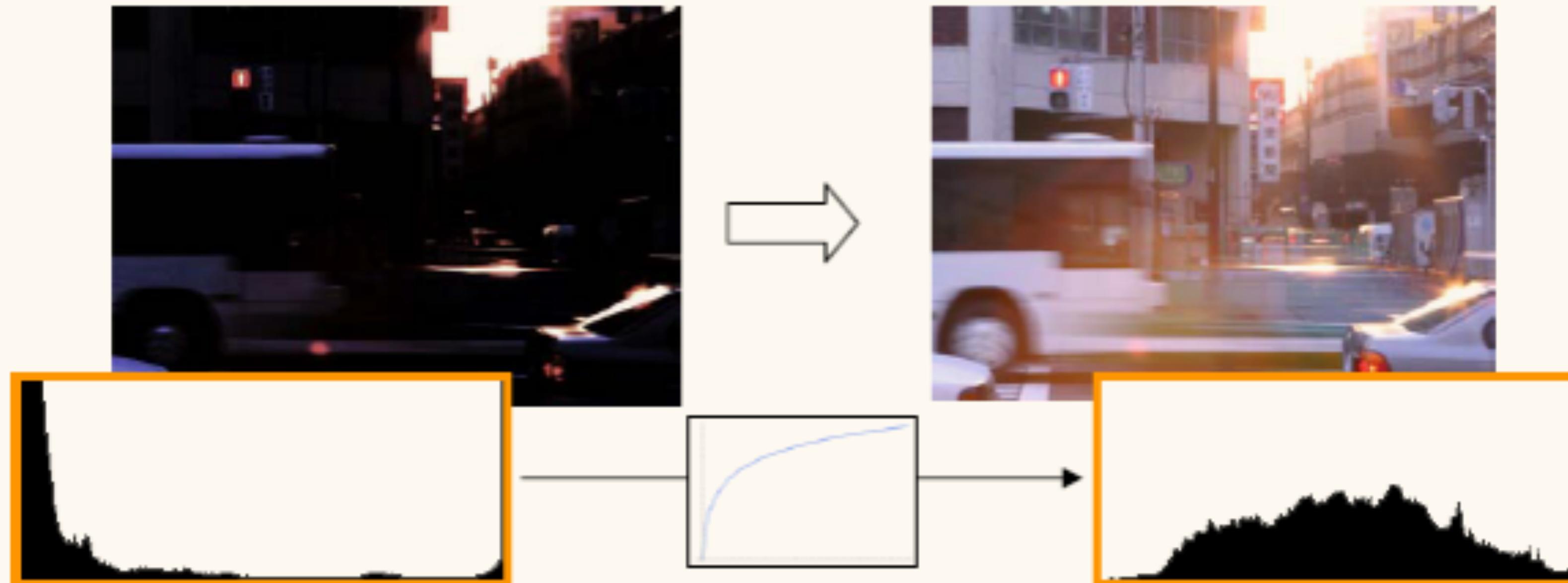
Real-time HDR - Step II

- Blooming
 - Using Kawase bloom filter in 8 passes any more passes more blur



Real-time HDR – Step III

- Tone mapping
 - Mapping HDR Image to LDR of screen

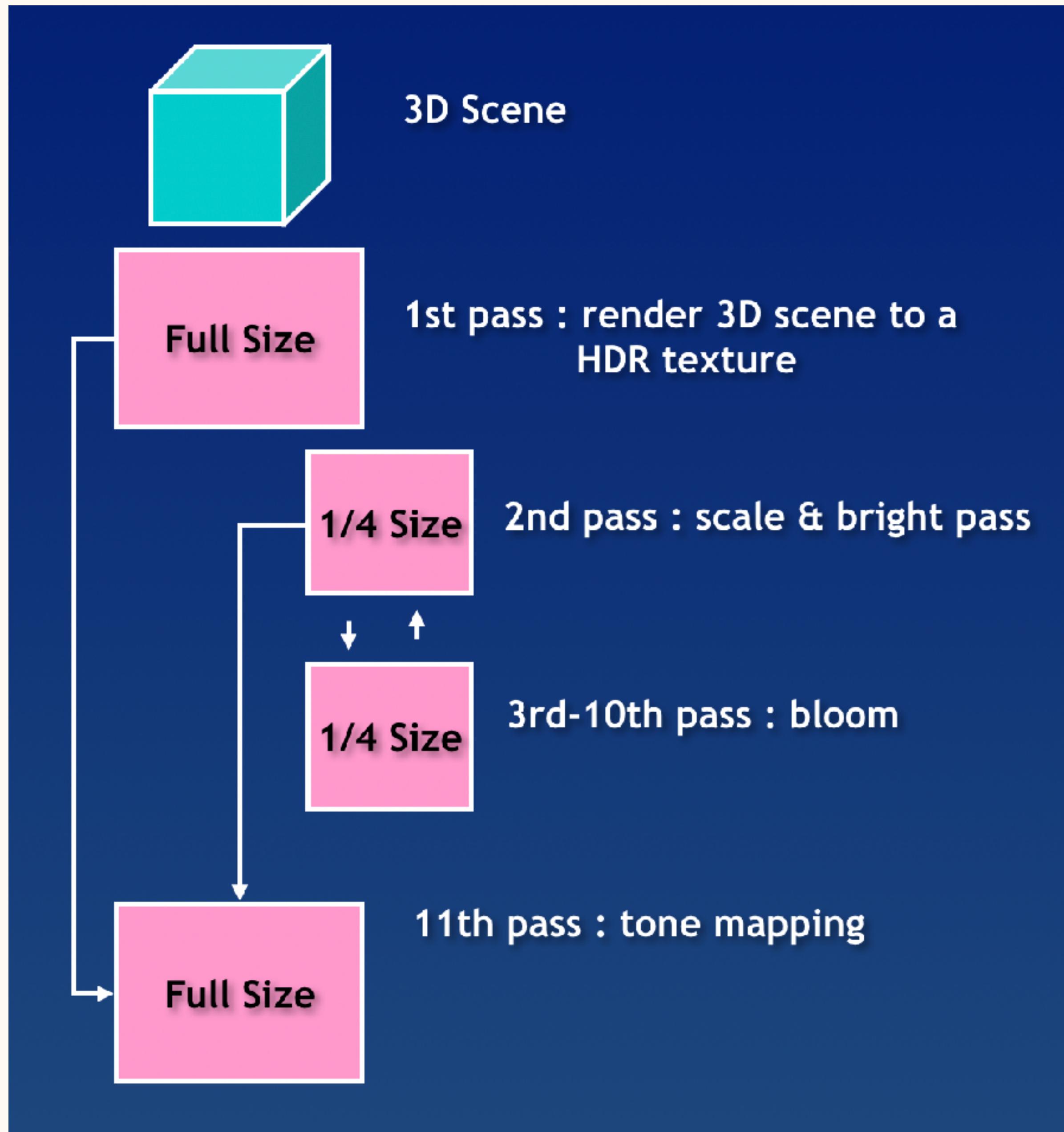


HDR image, visible image
and histogram of intensity

Real-time HDR – Step III

- Reinhard Tone Mapping
 - Basic idea :
 - $LDR = HDR / (1 + HDR)$
 - Full formula : (Lum_{White} is the white color in the game)

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

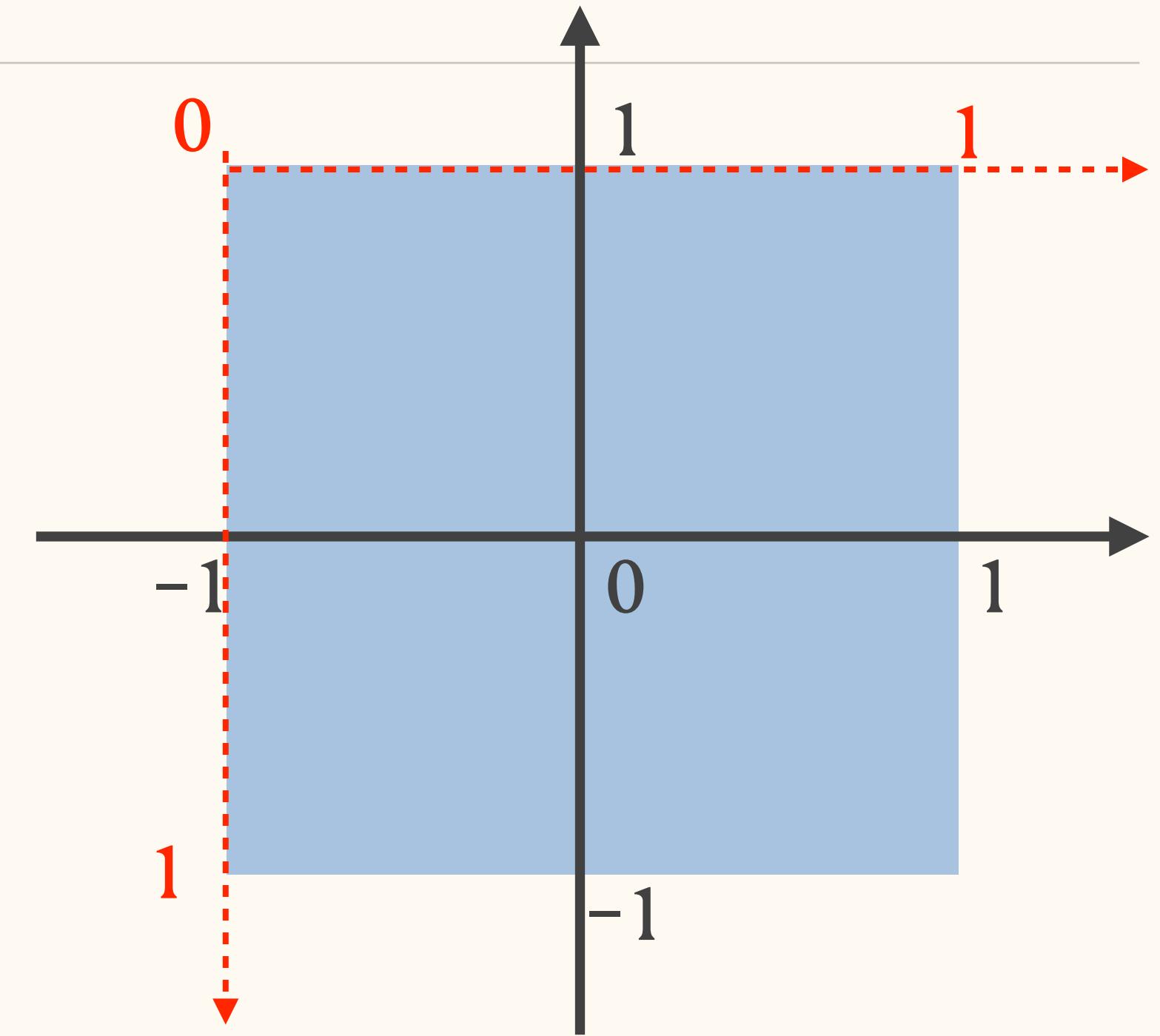


HDR Post-processing



HDR Shaders

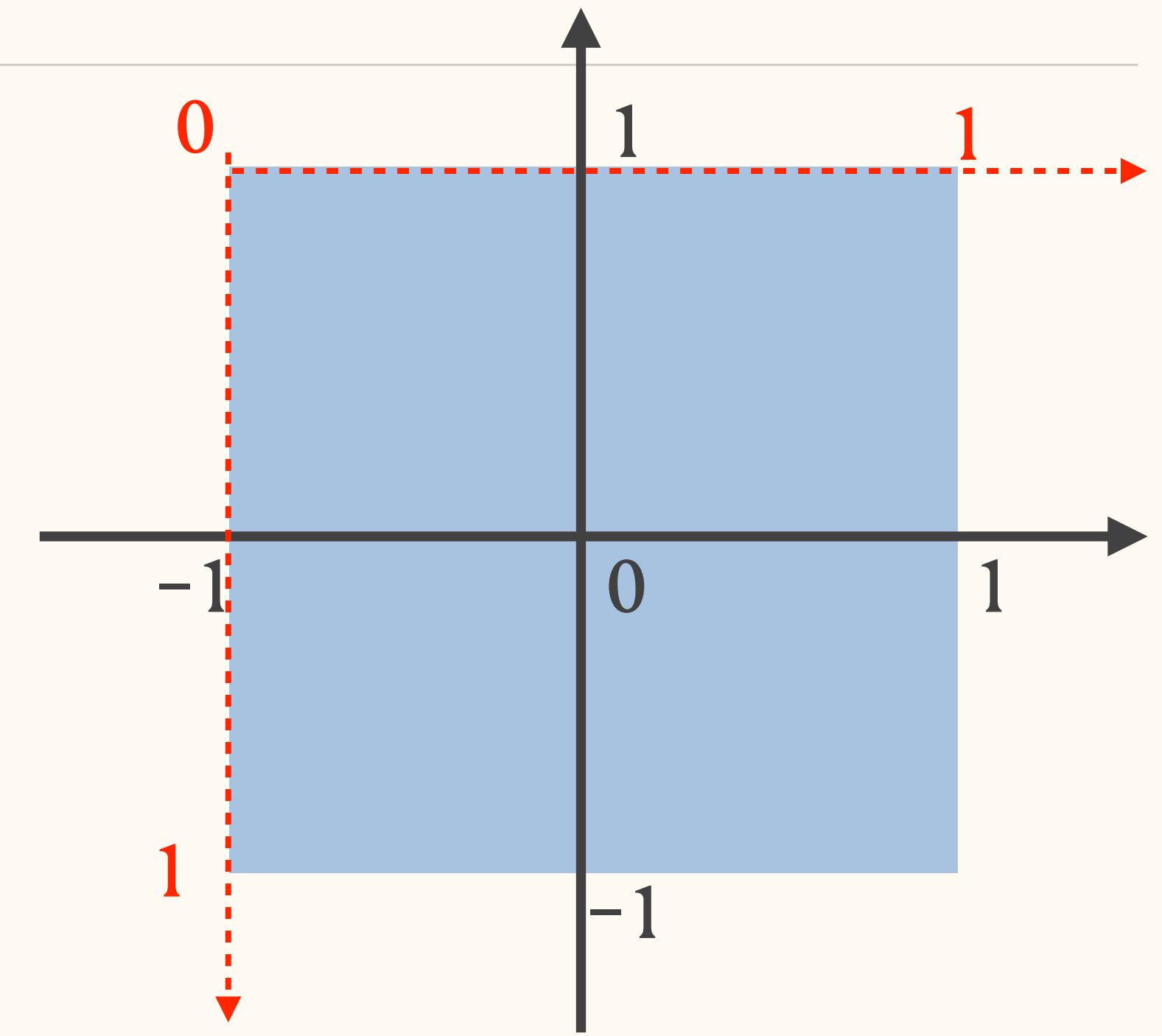
- 3 Shaders to complete the HDR post-processing
 - HDR_Bright_Pass Shader
 - HDR_Bloom_Shader
 - HDR_Tonemap_Shader
- For post-processing effect :
 - To render a full-screen quad, we need to create at least two triangles with x ranging $(-1.0, 1.0)$, y ranging $(-1.0, 1.0)$, $z = 0.5$ and $w = 1.0$. The texture uv is ranging in $(0.0, 1.0)$



Vertex Space in black color
texture Space in red color

Post-Processing Common Vertex Shader

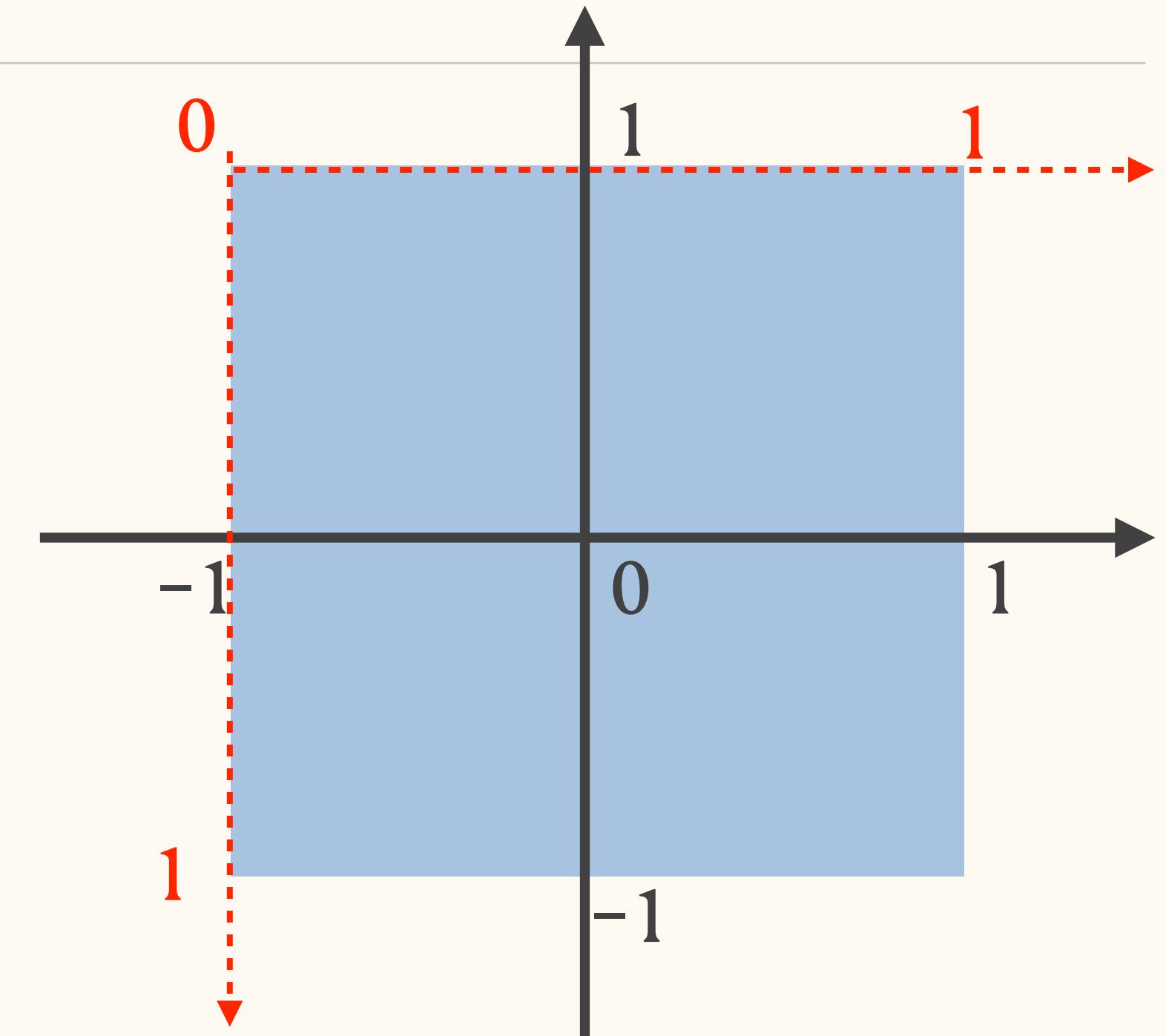
```
/*-----  
common vertex shader for post-processing effect  
-----*/  
  
// vertex shader input  
struct VS_INPUT  
{  
    float4 pos    : POSITION;  
    float2 tex0   : TEXCOORD0;  
};  
  
/*-----  
output channels  
-----*/  
struct VS_OUTPUT  
{  
    float4 pos    : SV_POSITION;  
    float2 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 VSBloom(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

```
/*-----
    pixel shader for making bloom effect with kawase blur method
-----*/  
  
// 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;

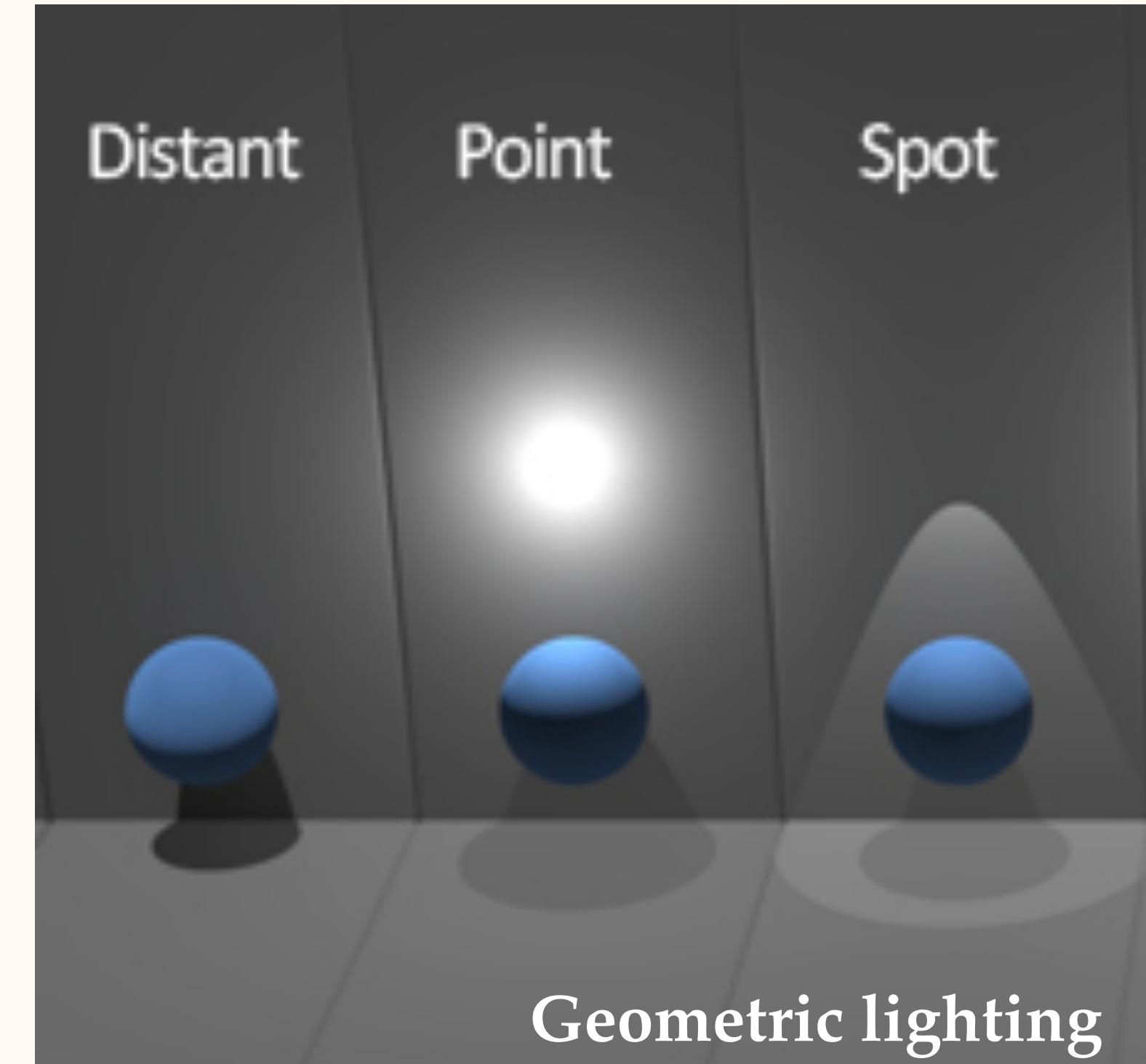
    // add the blurred one to the full sceen image
    color.rgb = fullColor.rgb + blurredImage.rgb*weight;

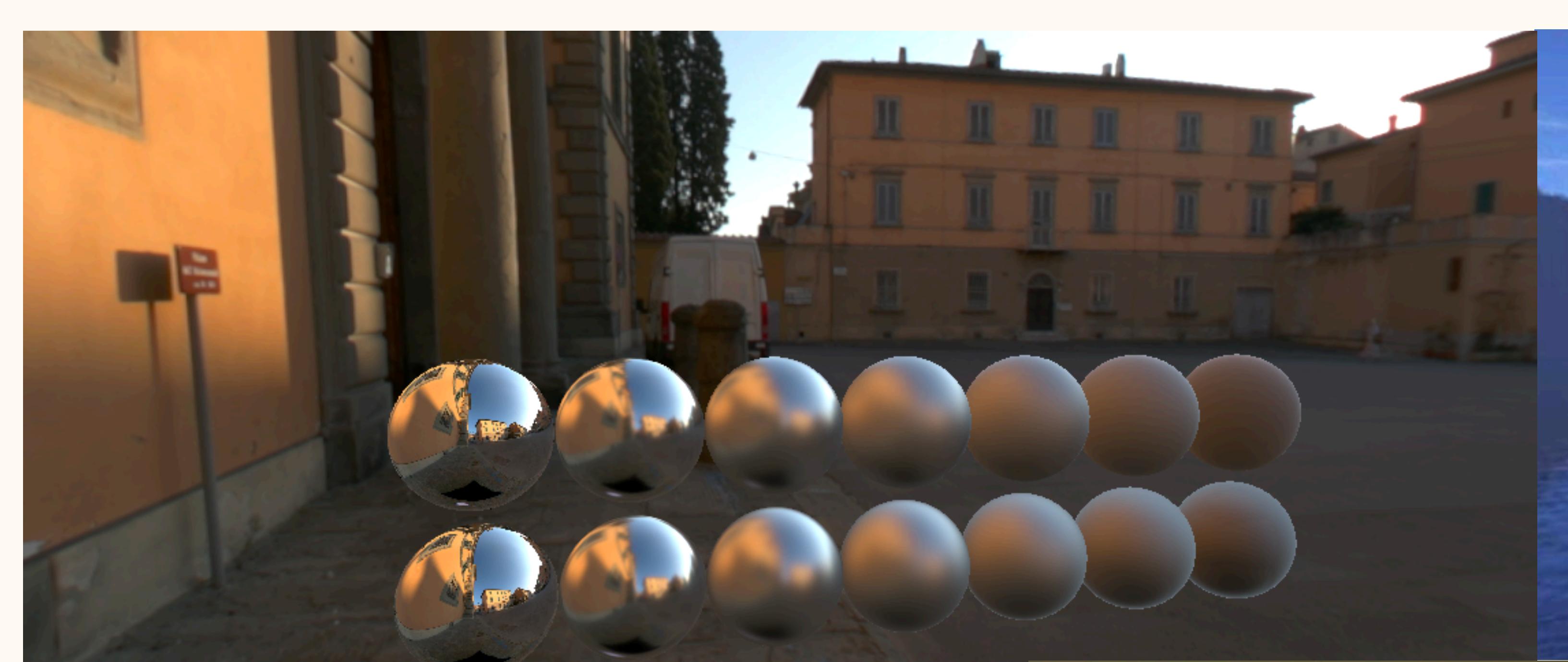
    // apply tone mapping
    color.rgb = color.rgb/(color.rgb + 1.0)*(1.0 + color.rgb/white2);
    color.a = 1.0;
    return color;
}
```

Image-based Rendering

Introduction to Image-based Lighting

- Geometric Lighting
 - Point light
 - parallel light
 - Distant light
 - Spot light
- Image-based Lighting (IBL)
 - Not geometric lighting
 - We will talk about :
 - Reflection & Refraction (Environment Mapping)
 - Diffuse/Specular Environment Mapping (Realtime IBL)
 - Ambient Occlusion

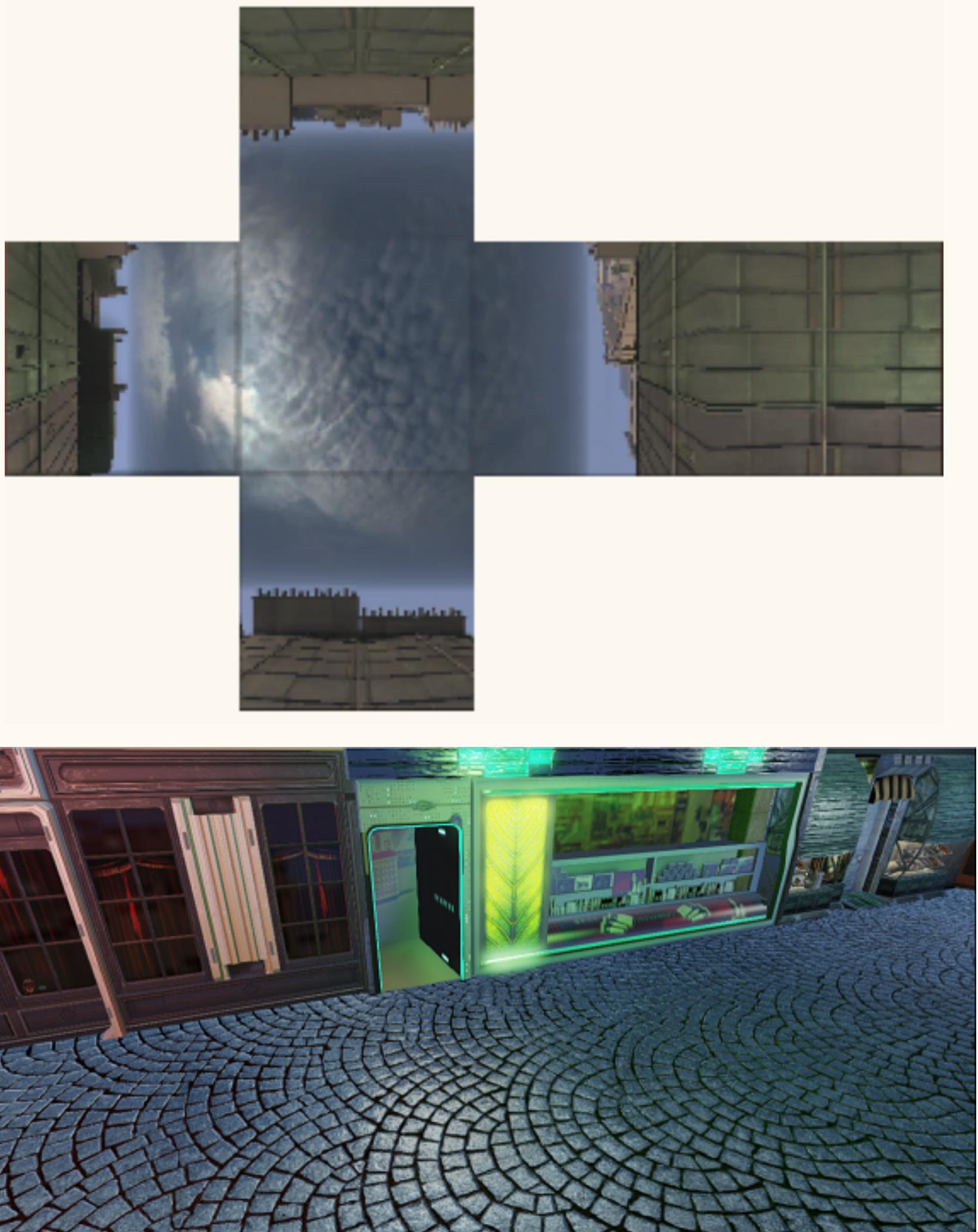




Reflection & Refraction

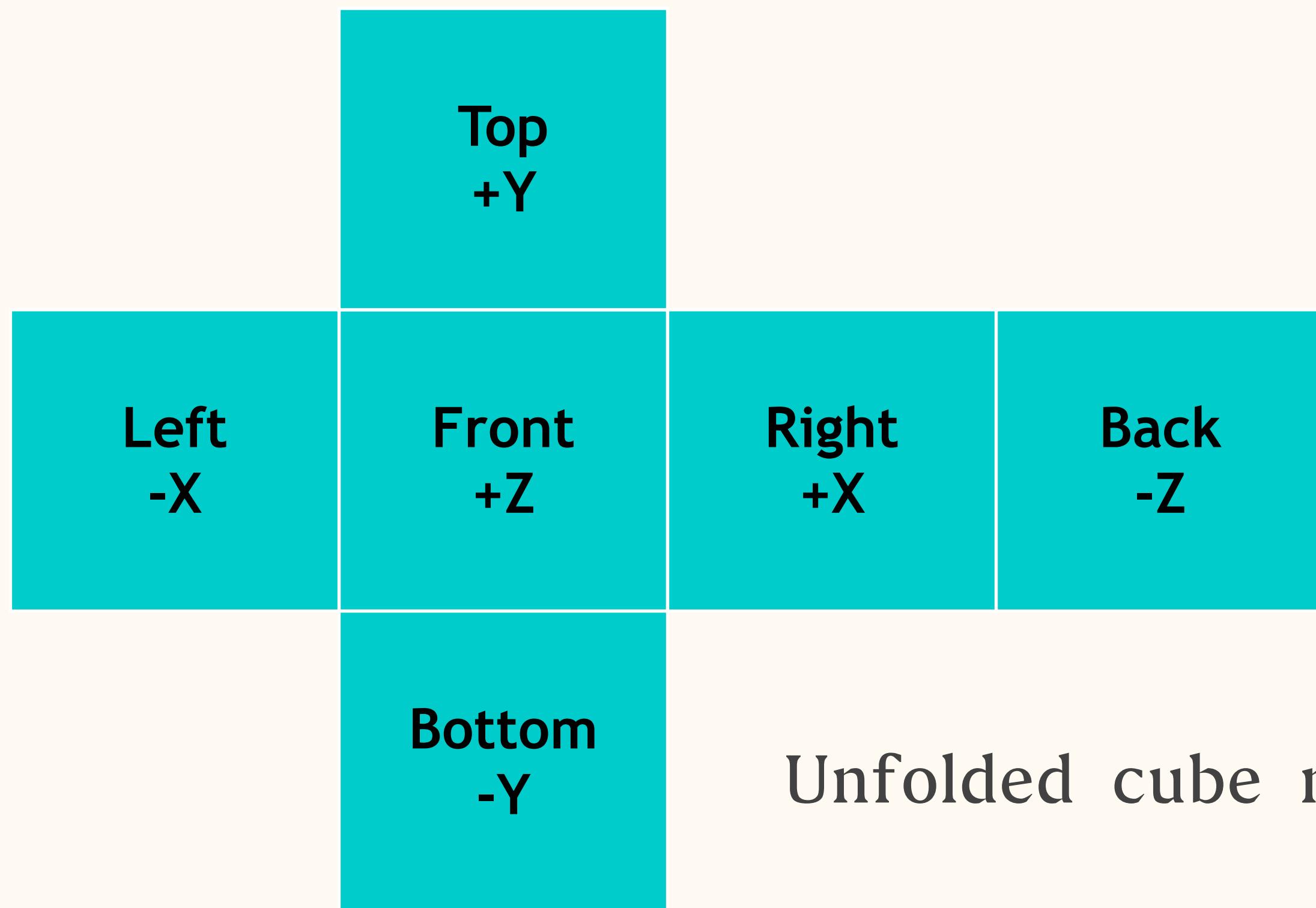
Panorama Images

- Distant 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

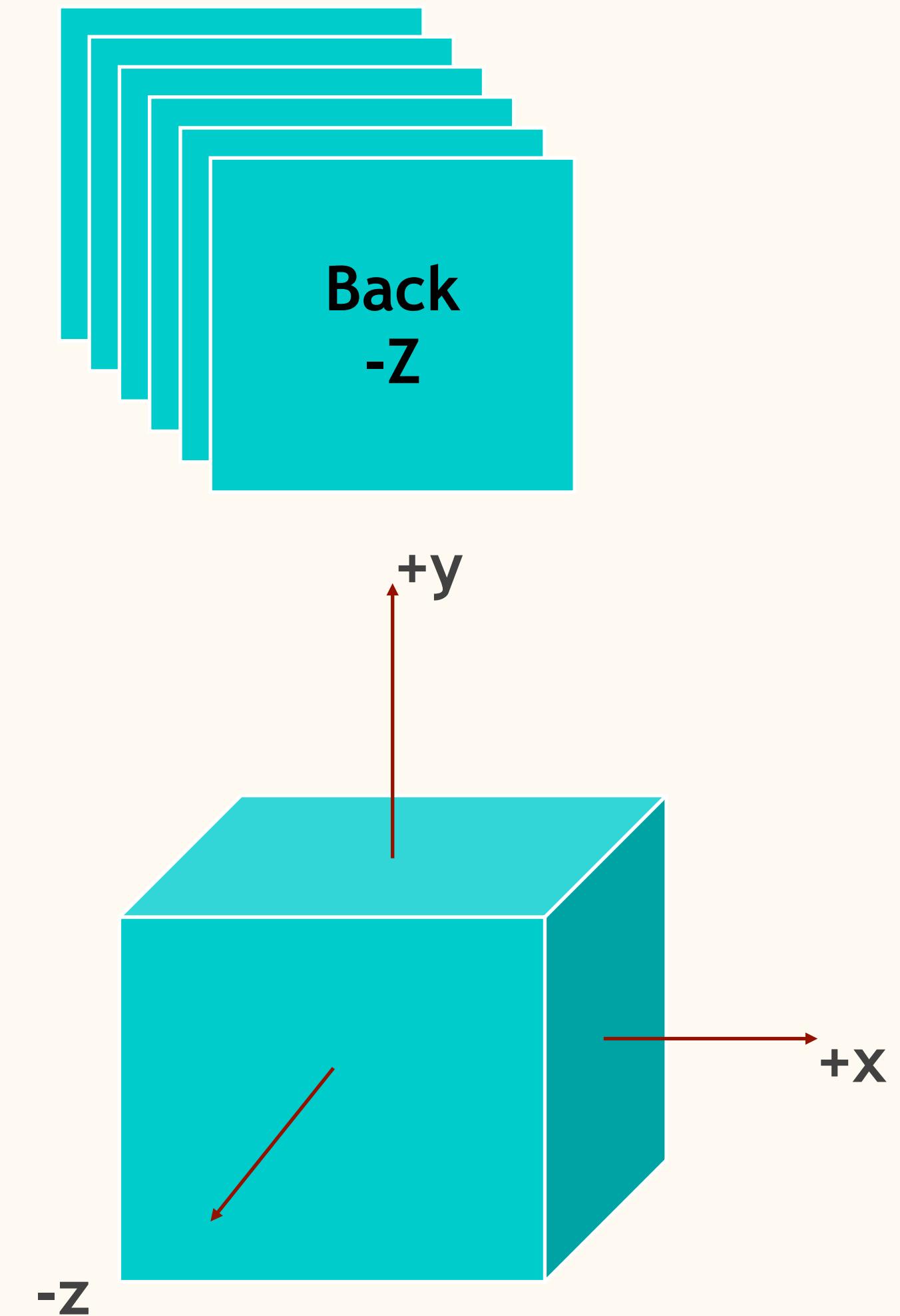




Cube Map

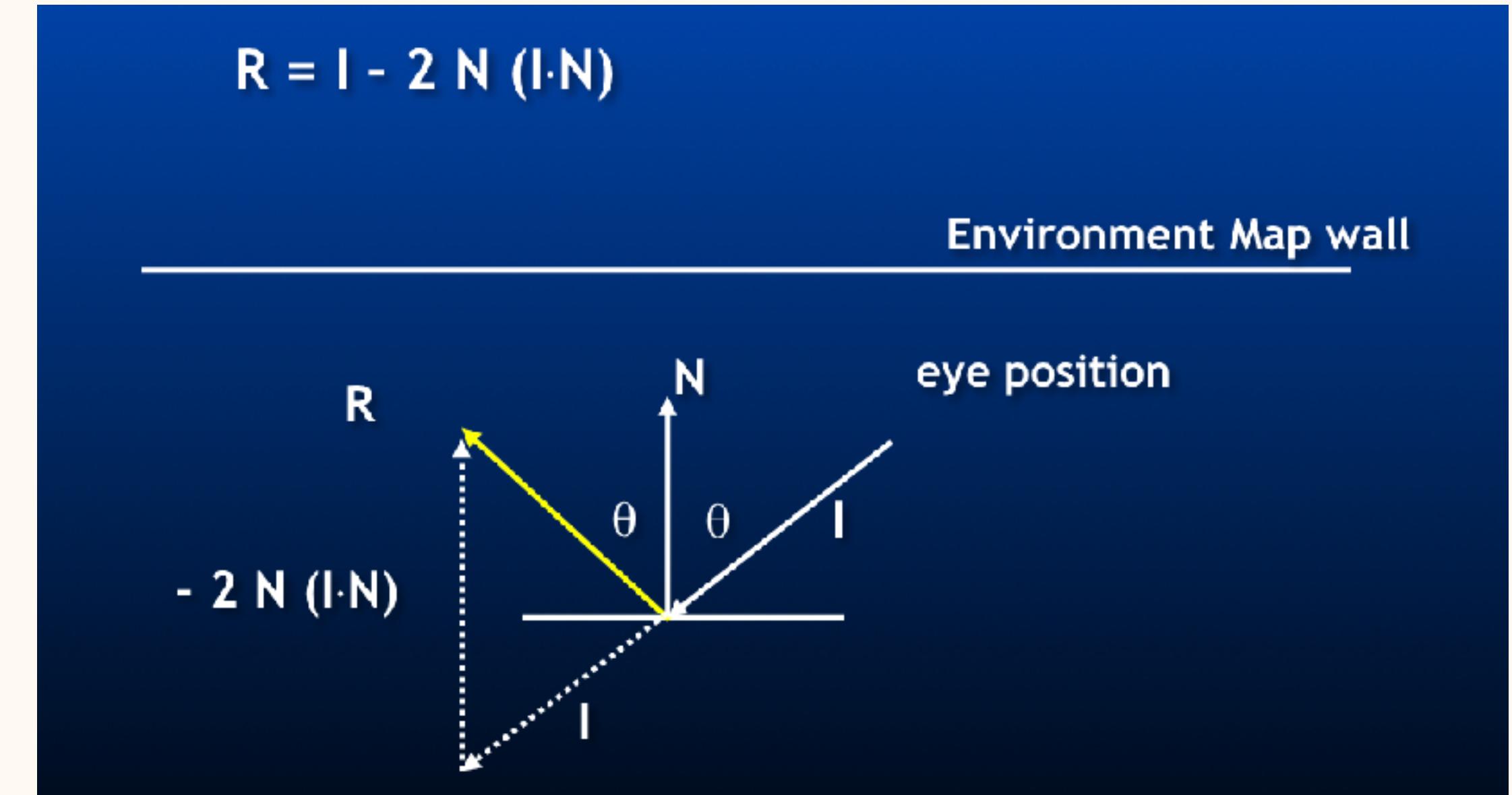


Unfolded cube map



Reflection Vector

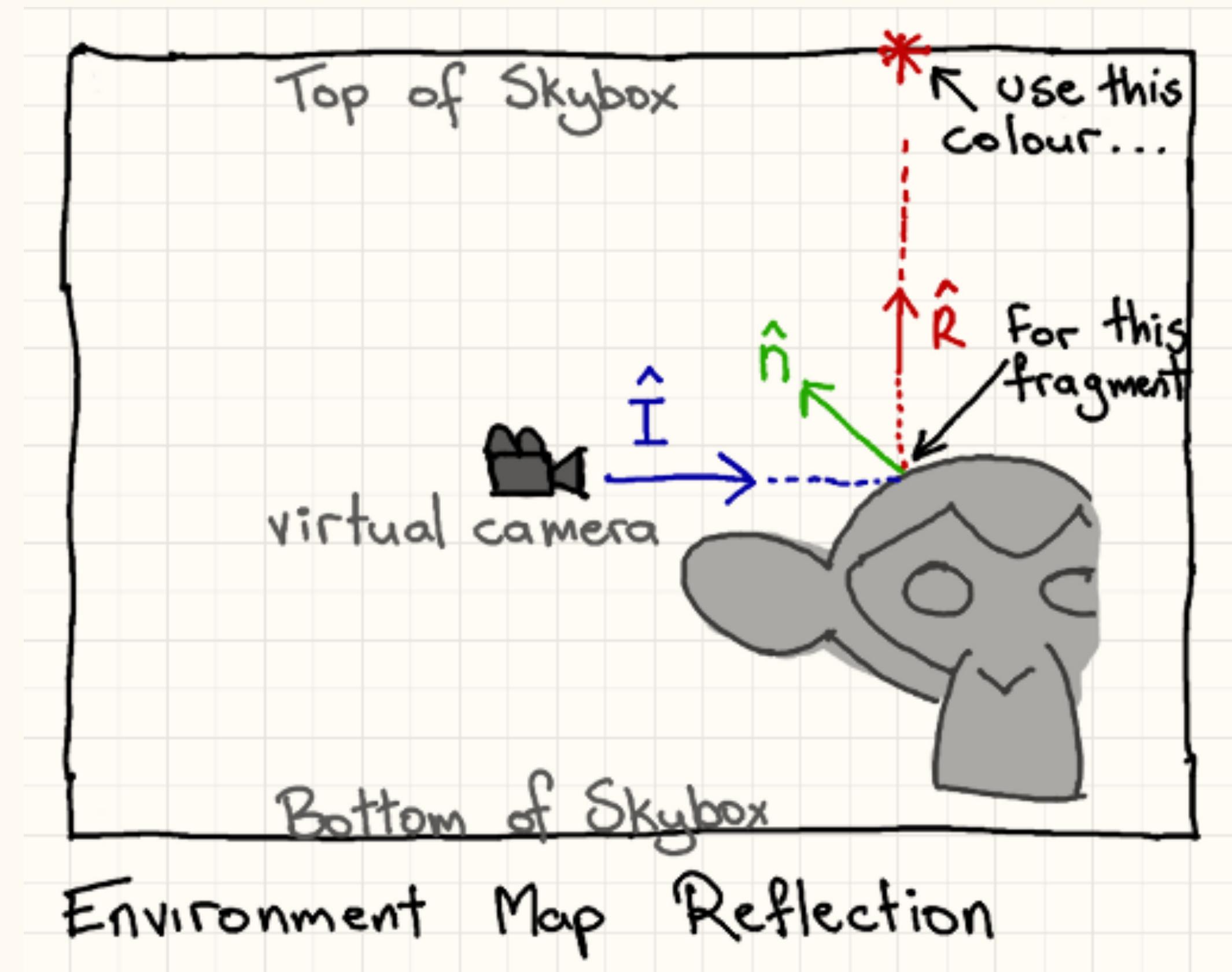
- Generate a camera's reflection vector with the surface normal to get the (r, g, b) in a cube map to simulate the reflection effect on a mirror-like object surface.
- To calculate the reflection vector :



- 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



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;
}
```

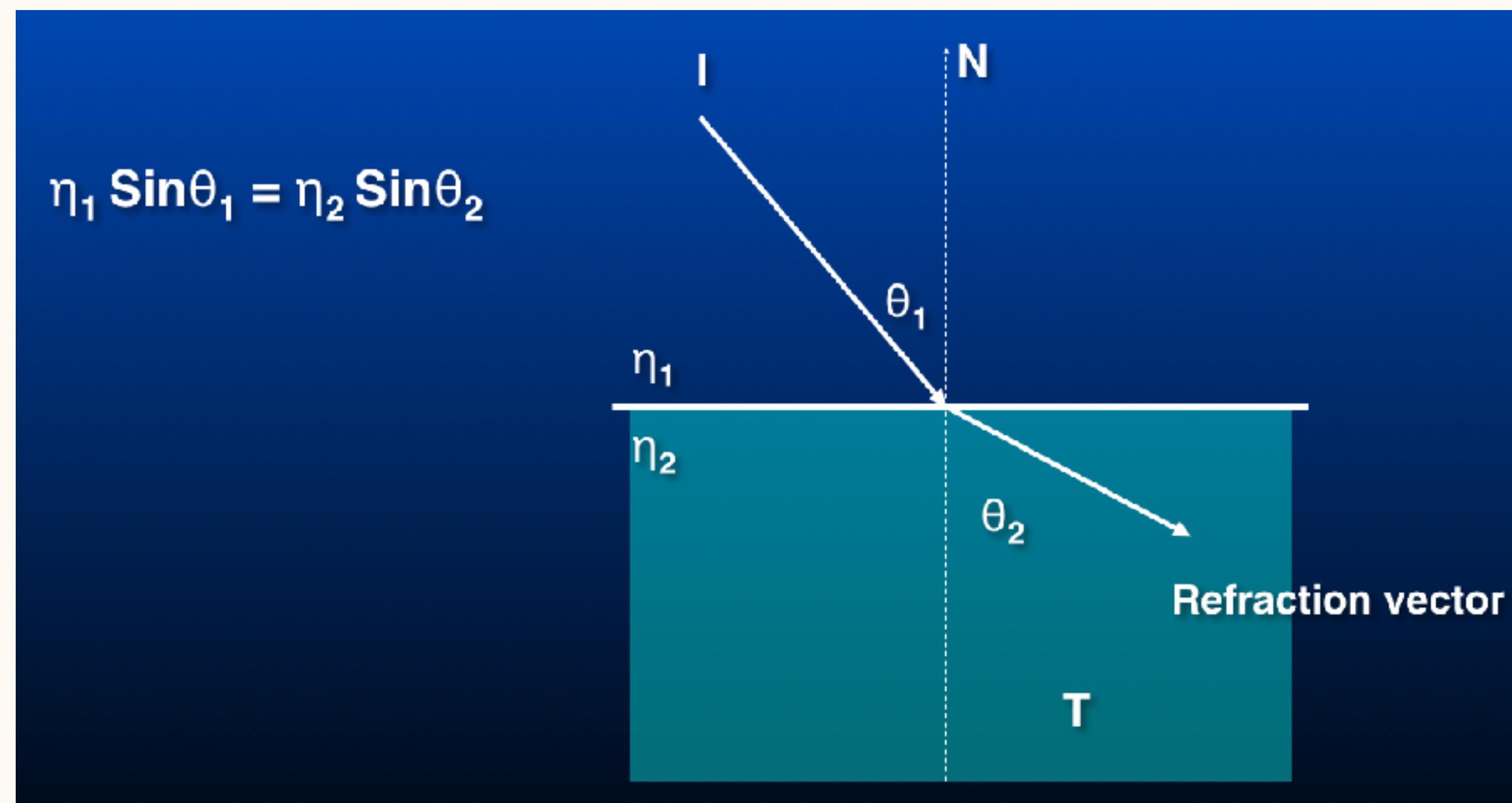
```
// textures and samplers
TextureCube cubeMap : register(t0);
SamplerState cubeMapSampler : register(s0);

// 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

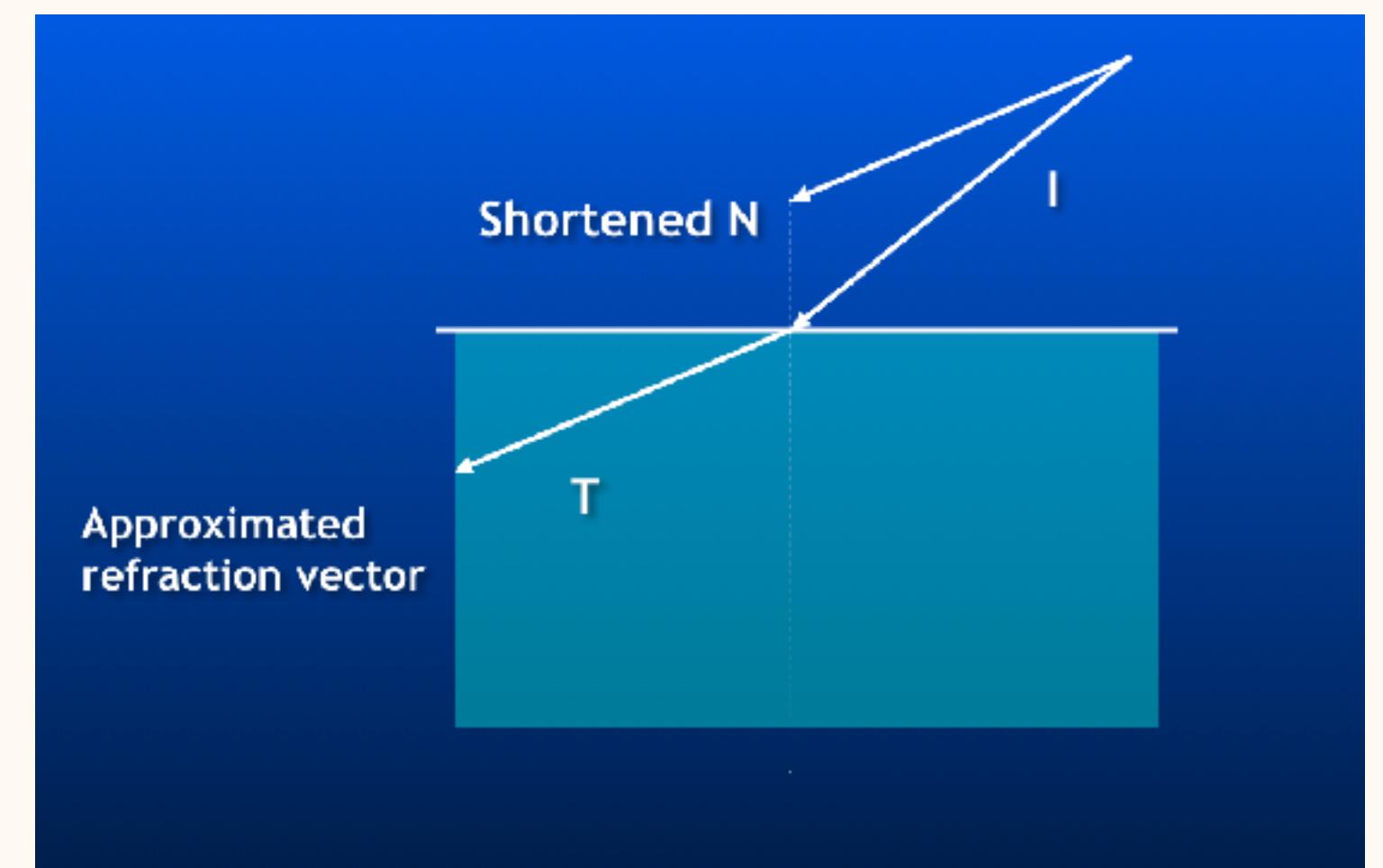
- Based on the Snell's Law
- For a closed object, any incident ray should be refracted twice.
 - We can not trace the 2nd refraction. So, the solution is not correct.
 - But the result is still pretty good to simulate the semi-transparent object



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
 - 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);

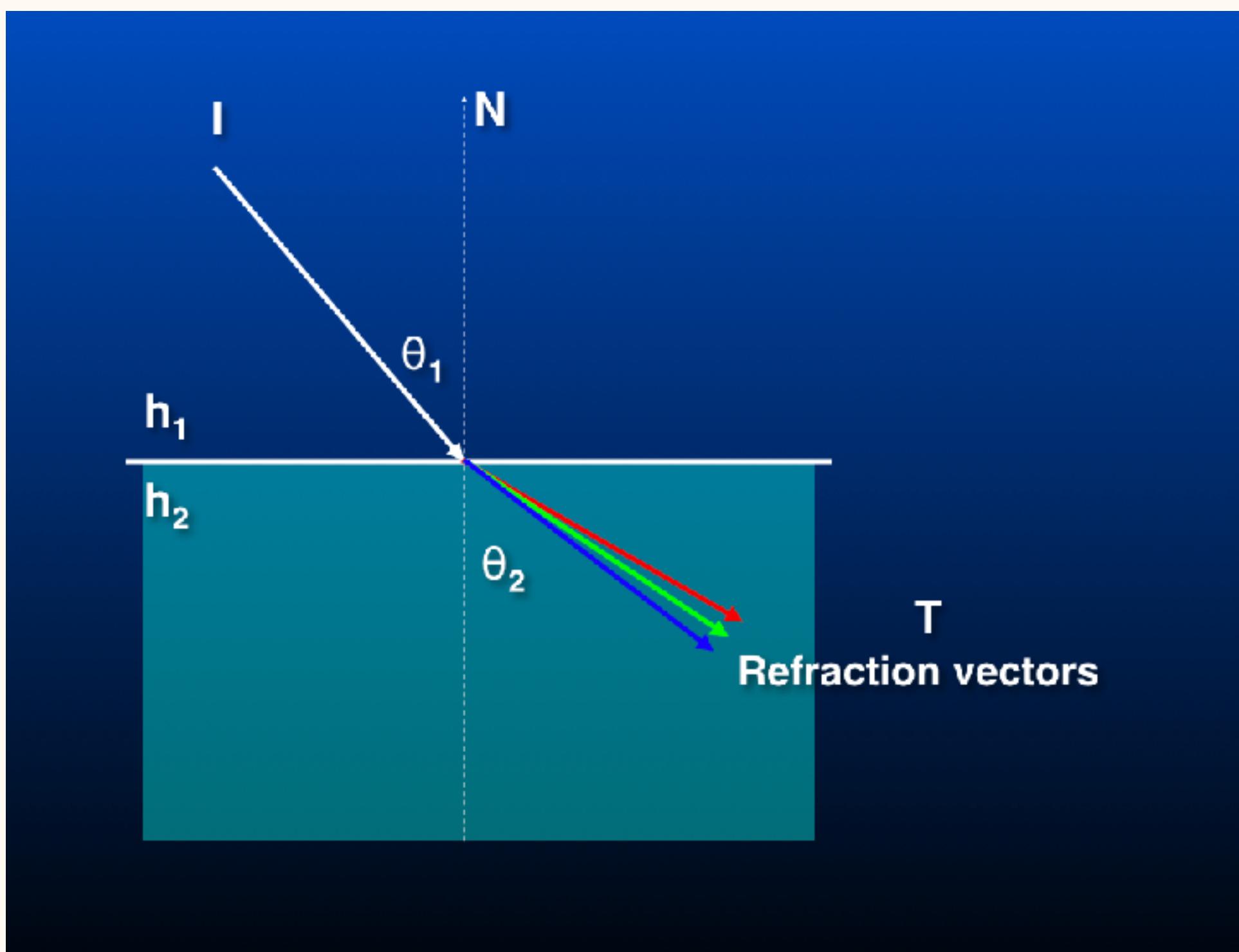
// pixel shader input
struct PS_INPUT
{
    float4 pos      : SV_POSITION;
    float3 refl     : TEXCOORD0;
    float3 refr     : TEXCOORD1;
};

// 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;
};

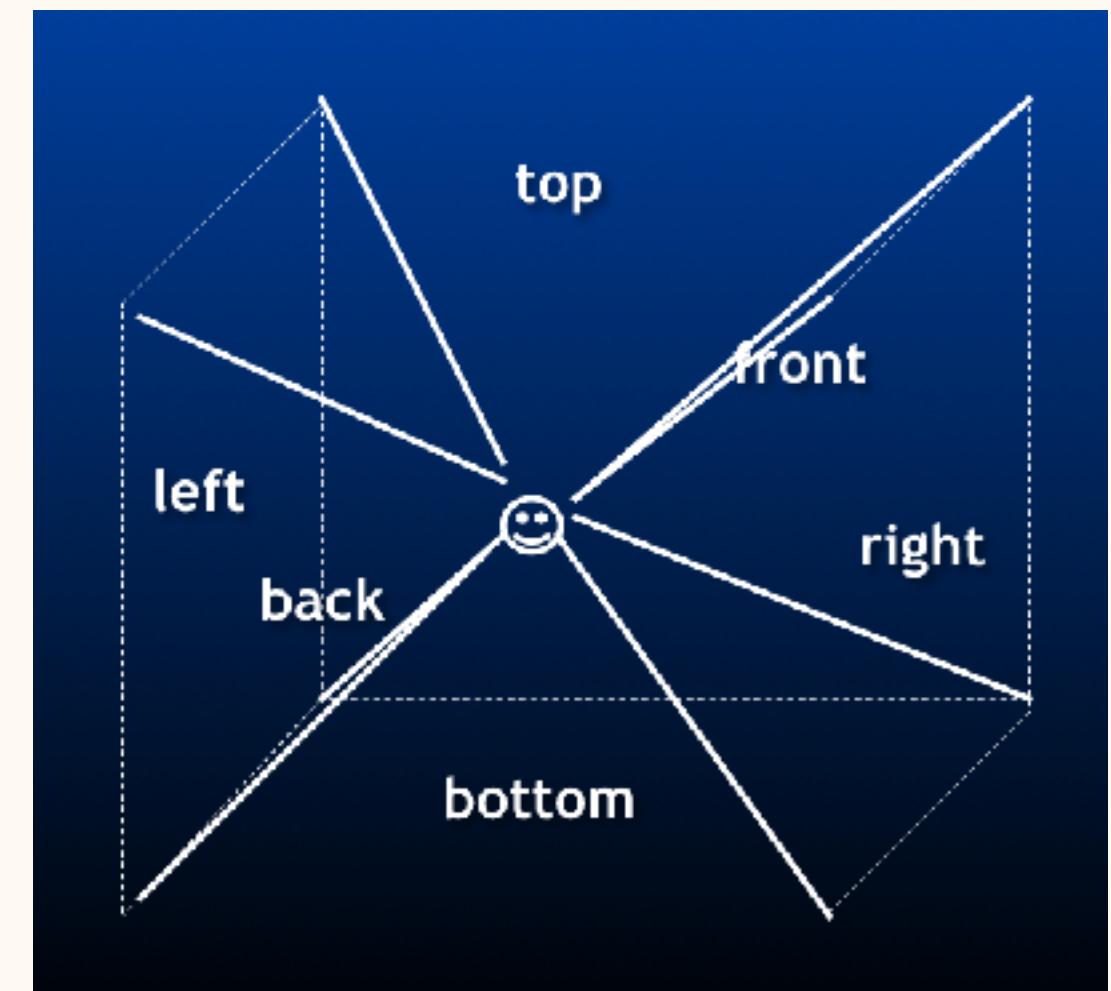
...
// calculate the refraction vector
float3 shortNormR = mul(norm, 0.485f);
float3 shortNormG = mul(norm, 0.49f);
float3 shortNormB = mul(norm, 0.495f);
float3 refrR = reflect(inc, shortNormR);
float3 refrG = reflect(inc, shortNormG);
float3 refrB = reflect(inc, shortNormB);
```

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

- Prepare the cube map dynamically
- 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 $\text{FOV} = 90^\circ$ & 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
 - The shader code is the same.



HDR Photography

HDR Image for Image-based Lighting

- Hollywood has been doing this since the 90's
- Now possible in real time applications (i.e., games)

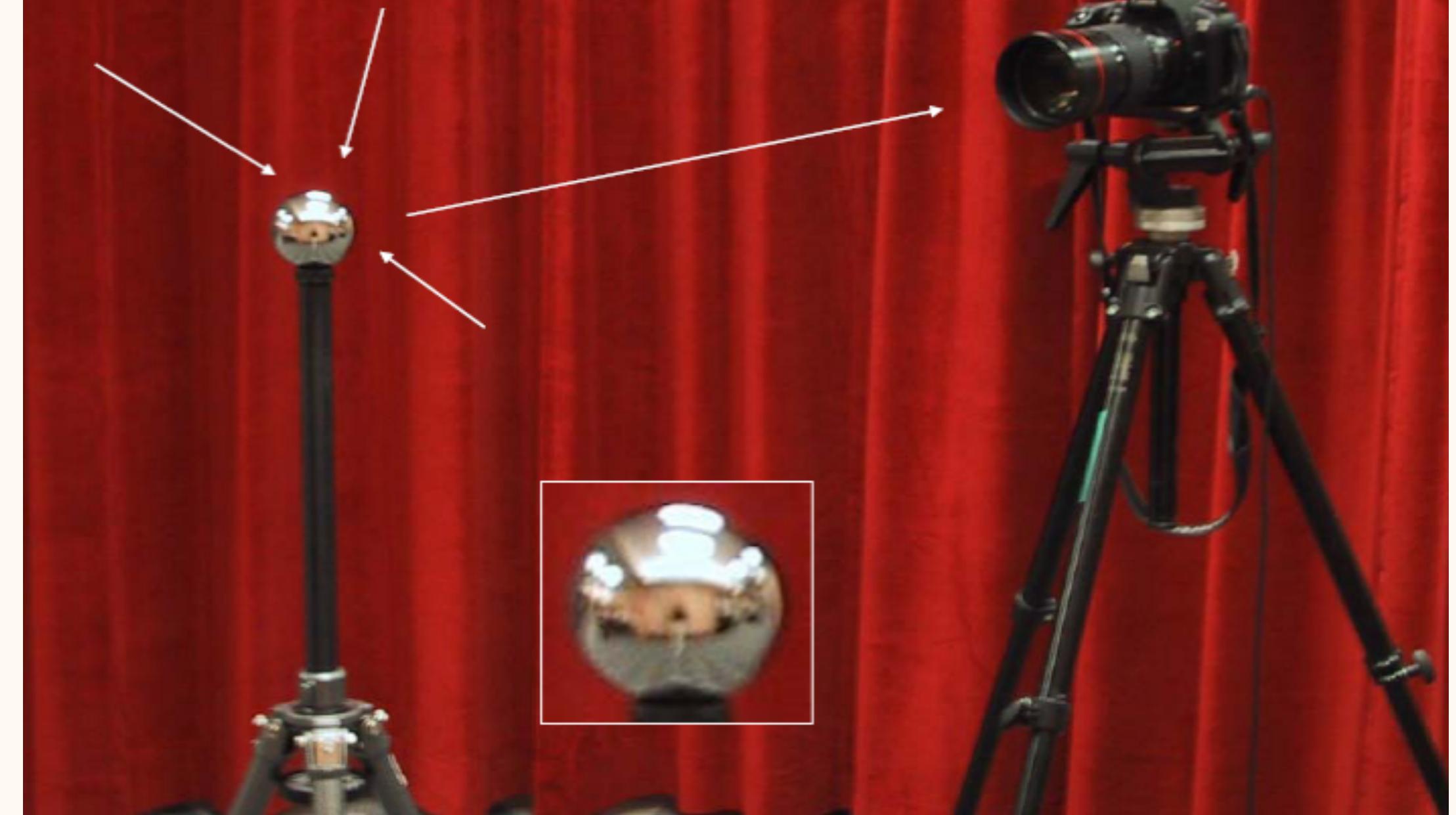


Capture HDR Light Probe Images

- Digital camera with manual controls to lighting
- Tripod & Head

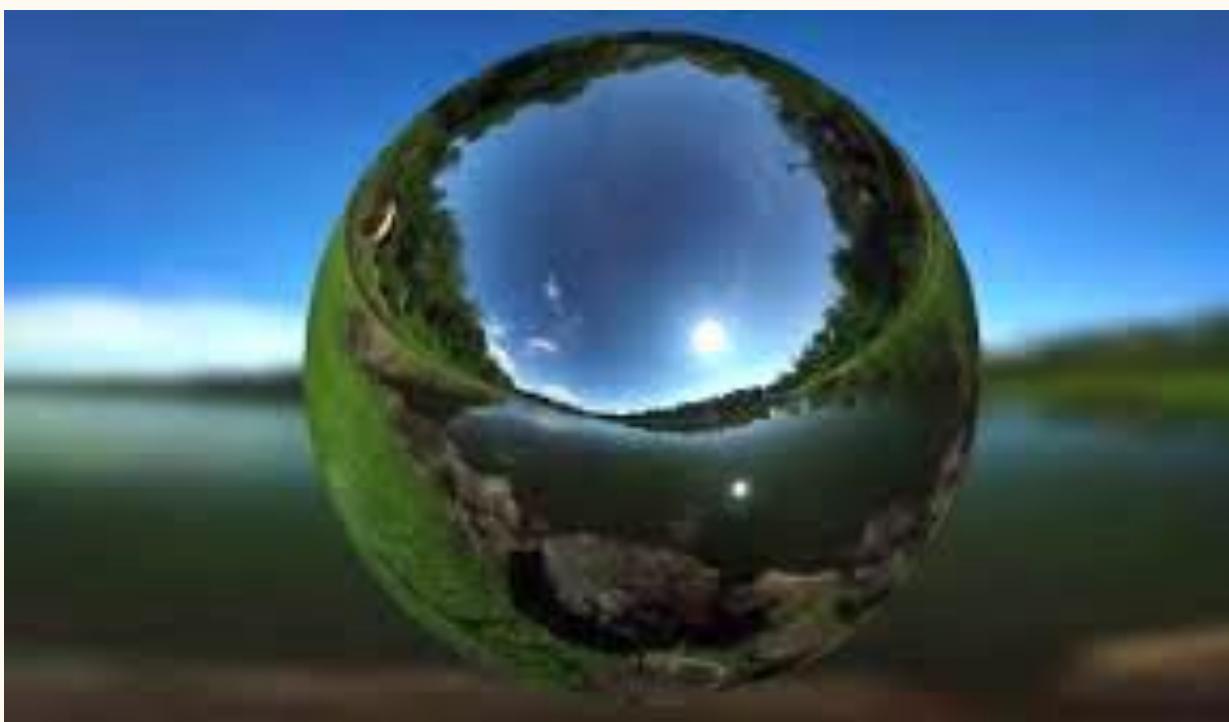


**Panoramic (Omnidirectional)
Photography**



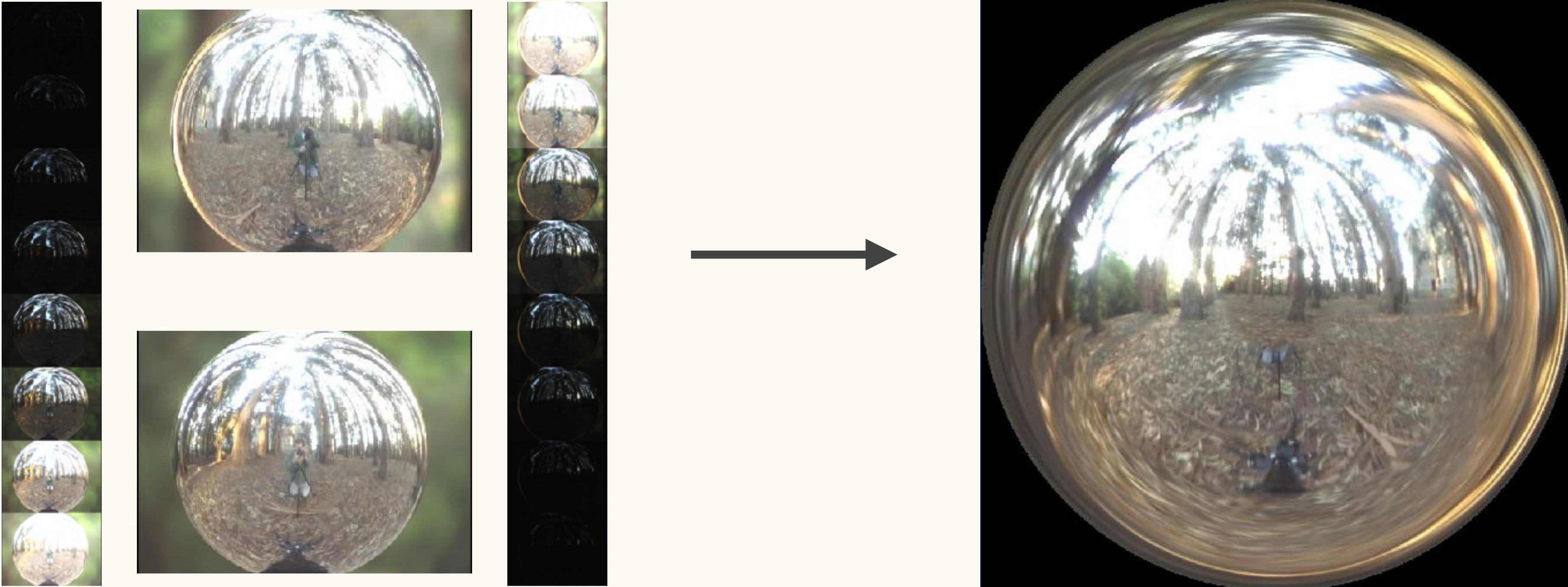
Capture HDR Light Probe Images

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



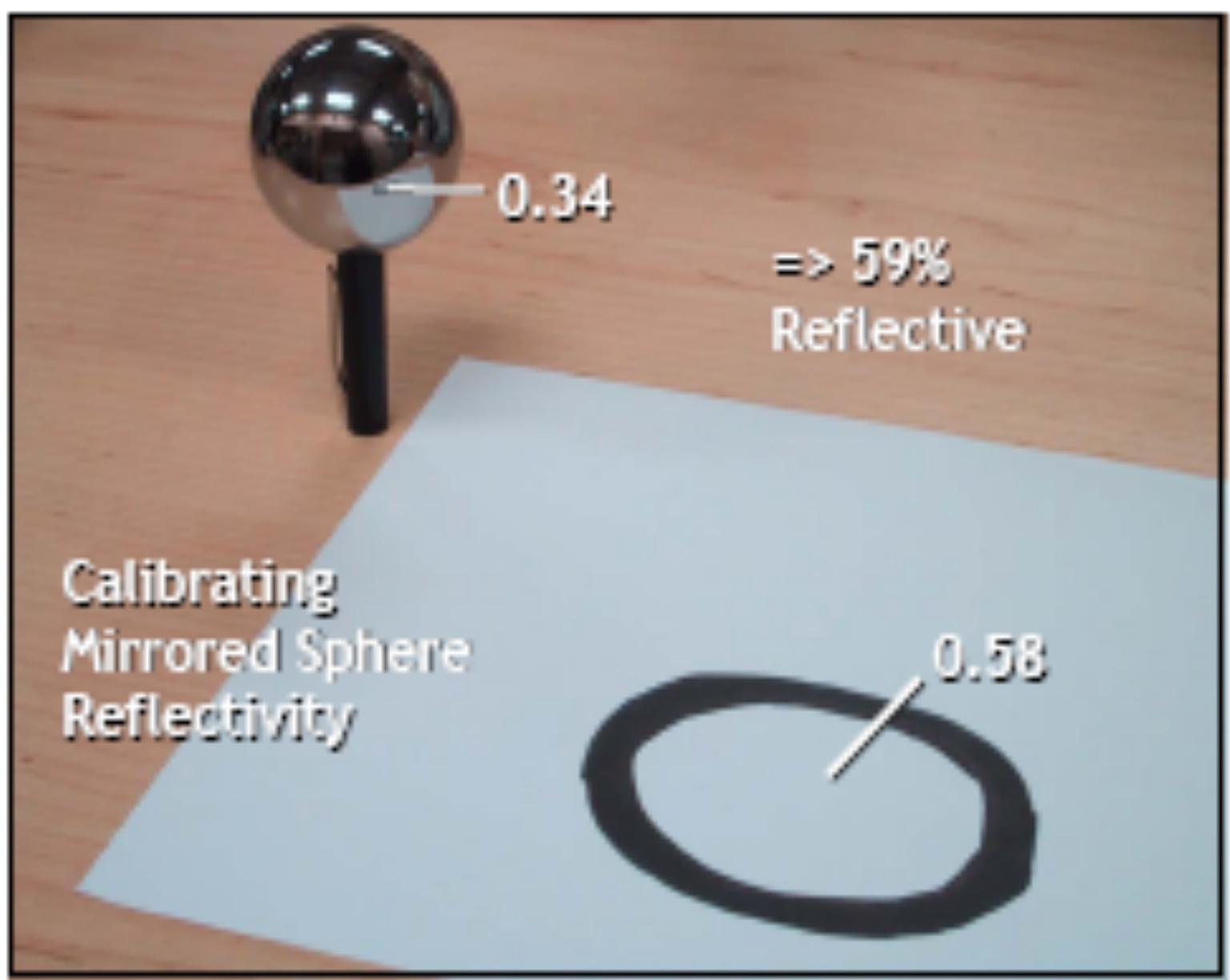
Capture HDR Light Probe Images

- Shoot a set of LDR images with (-4, -3, -2, -1, 0, 1, 2, 3) Ev
- Assembling the Light Probe from two sets of LDR images
 - In another 90° different position

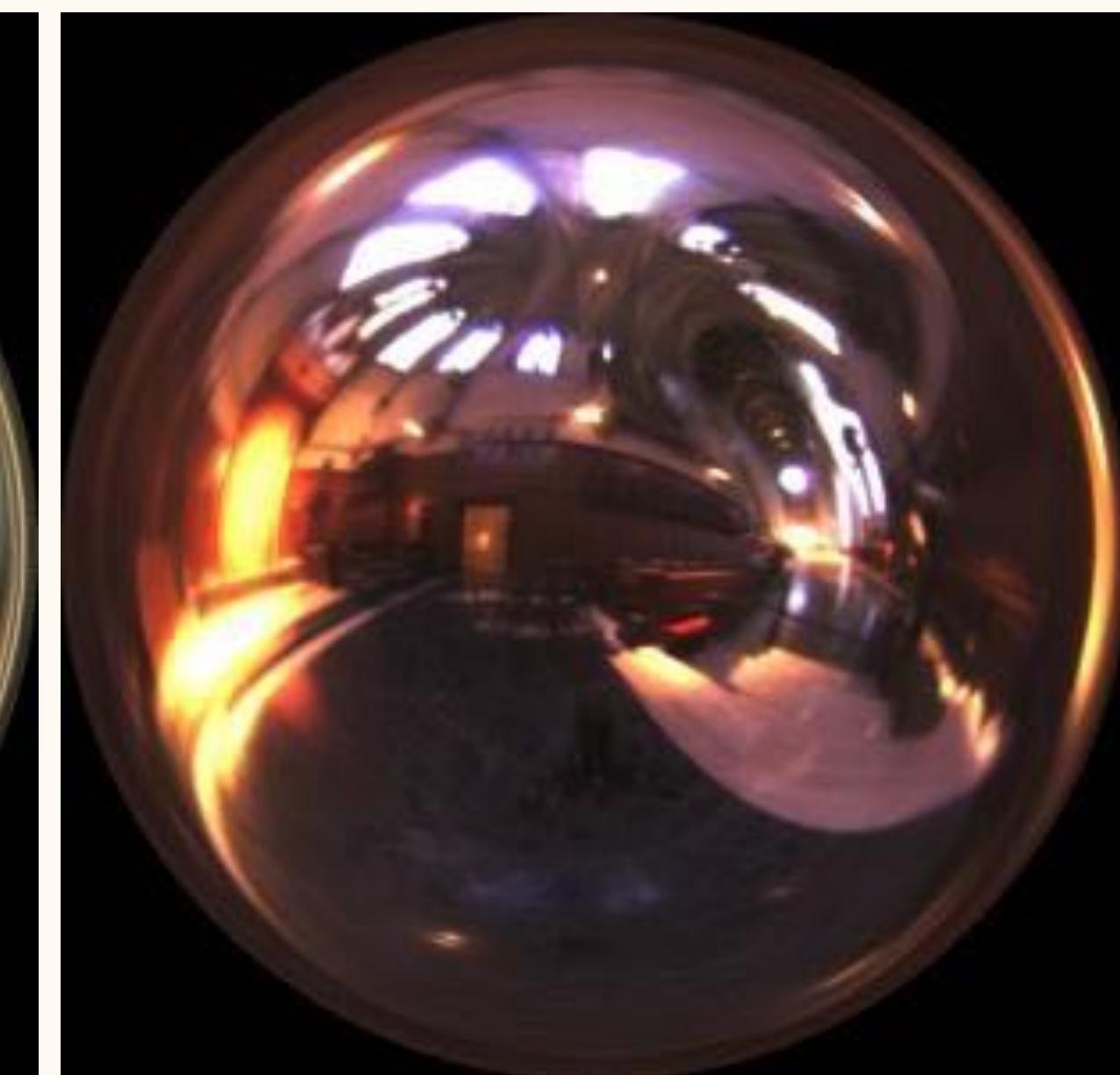
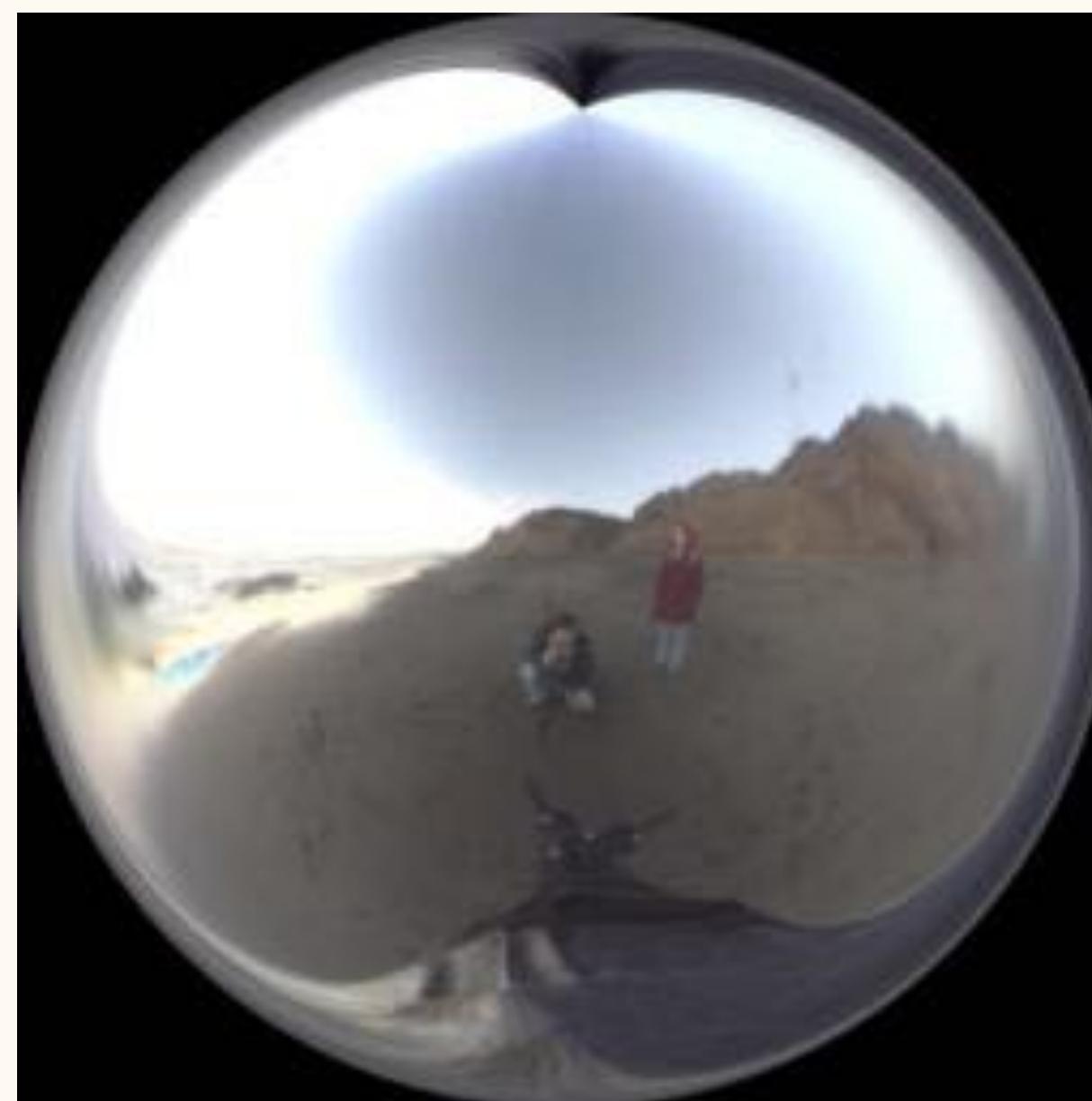


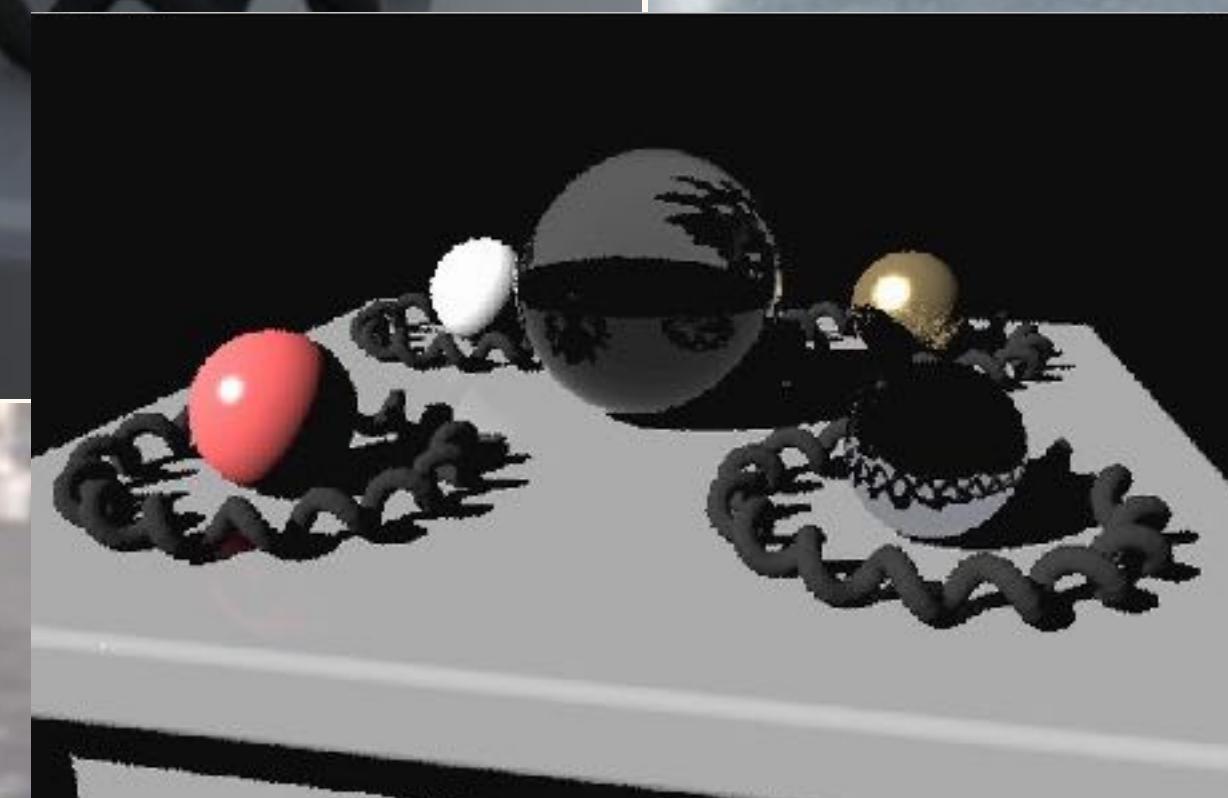
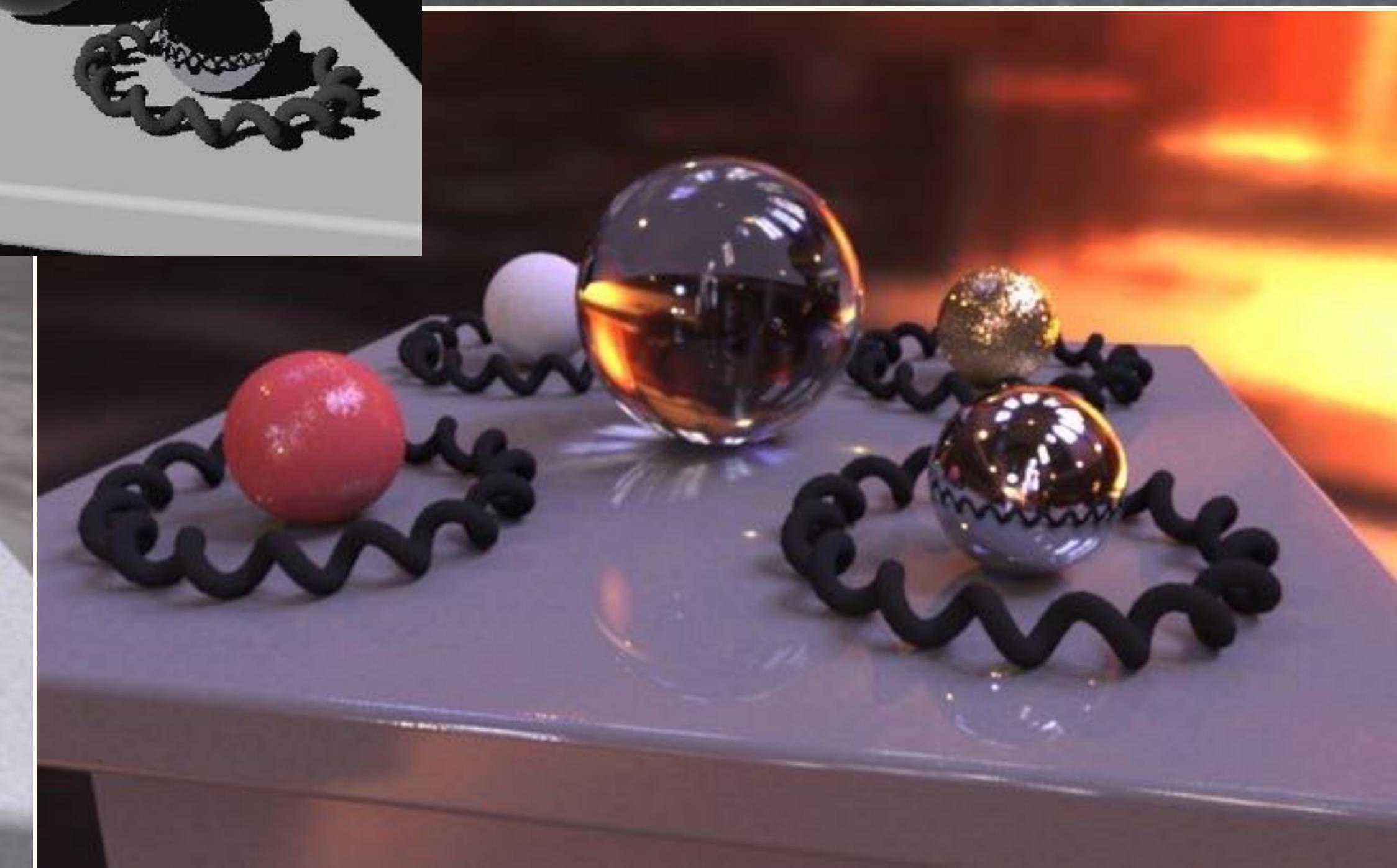
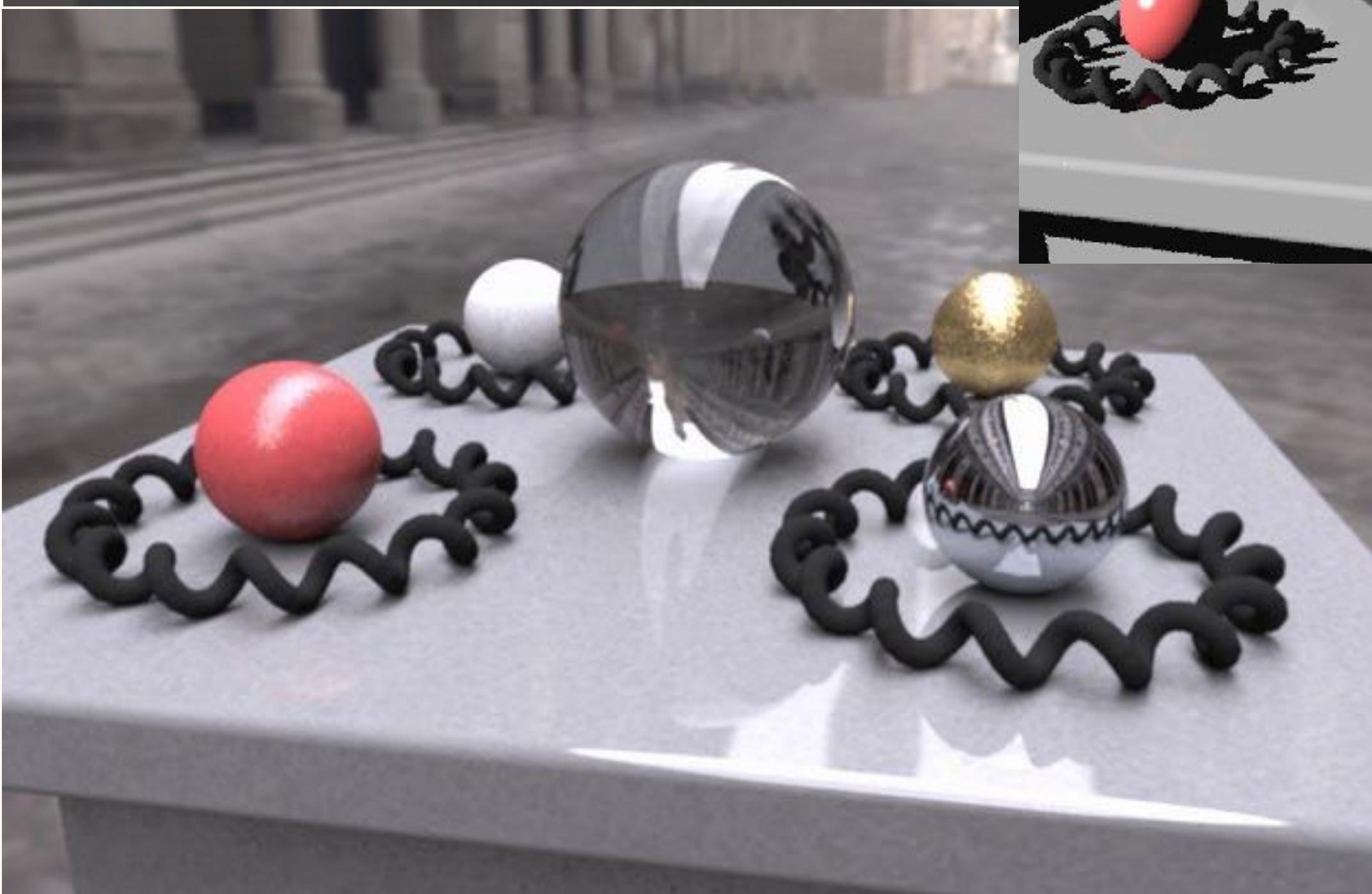
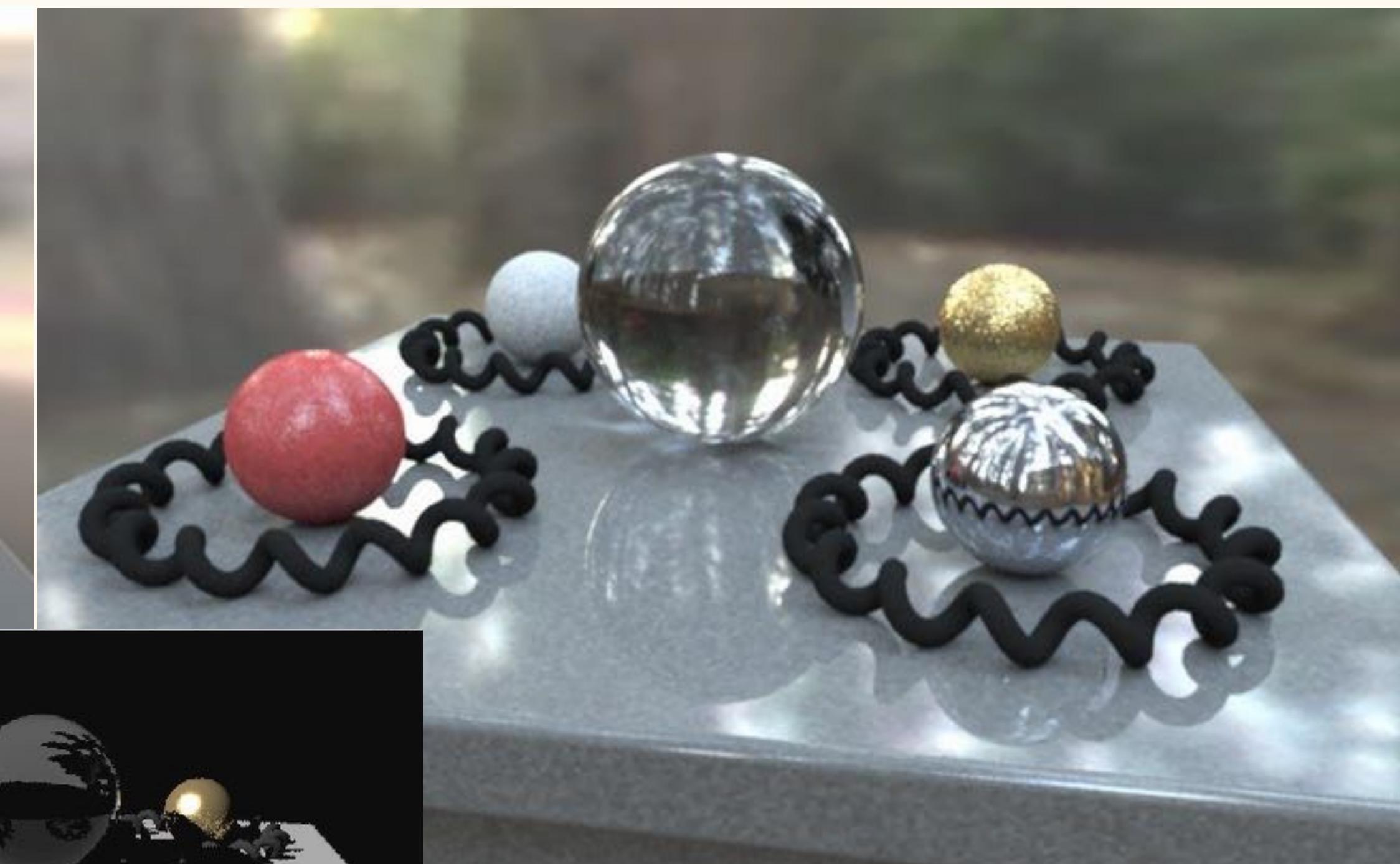
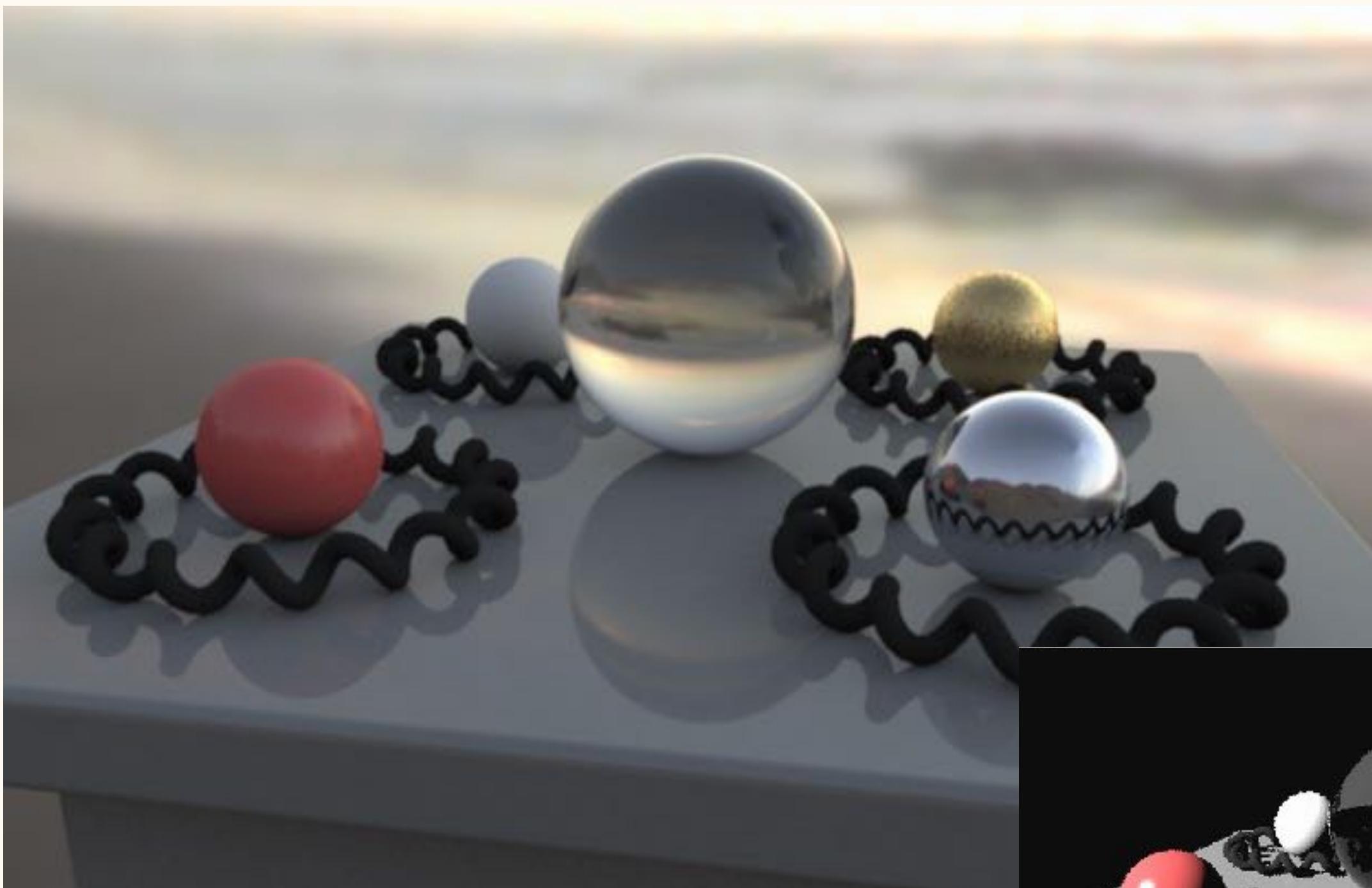
Mirrored Sphere

- Framing and focus
- Blind spots
 - Camera & photographer inside the image
 - Lack of region directly behind the sphere
- Calibrating sphere reflectivity
 - Common values (r, g, b) = (0.632, 0.647, 0.652)
- Non-specular reflection
 - Scratch
 - Oxidized
- Polarized reflection
- Image reflection



HDR Image for Image-based Lighting





Panorama Image Formats

- Convert panorama format using HDRShop software
- More panorama mapping solutions for image-based lighting :
 - Ideal Mirrored Sphere
 - Angular Map (Light probe)
 - Latitude Longitude Map
- Now we call all the panorama image for light source as the “Light Probe”

Ideal Mirrored Sphere

- Captured with 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 :

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

$$(\theta, \varphi) = (\text{atan2}(2u-1, -2v+1), 2\arcsin(r))$$

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



y-up right-hand space

Angular Map

- Converted from mirrored sphere image
- 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

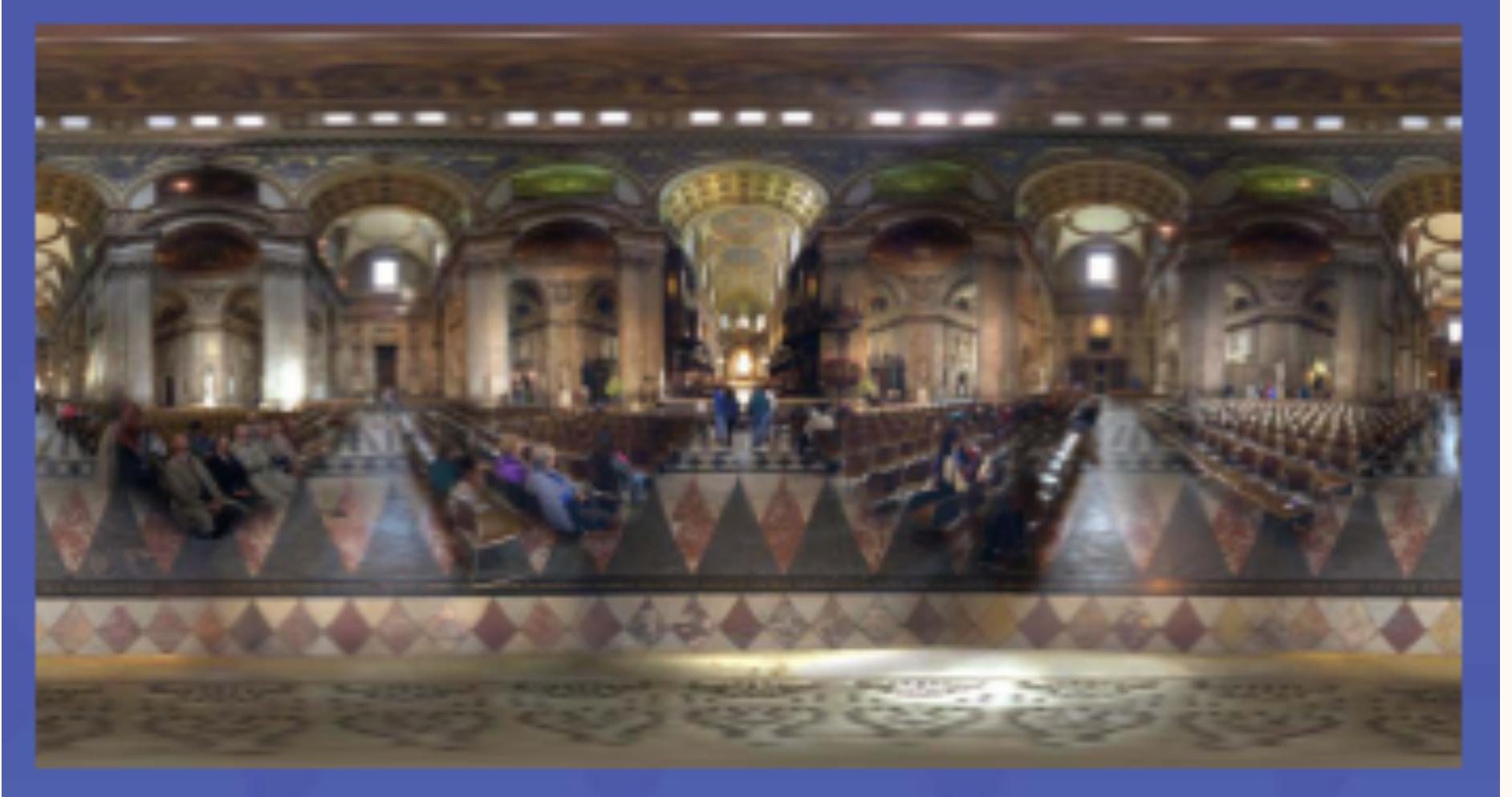
Latitude-Longitude Map

- 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);
```

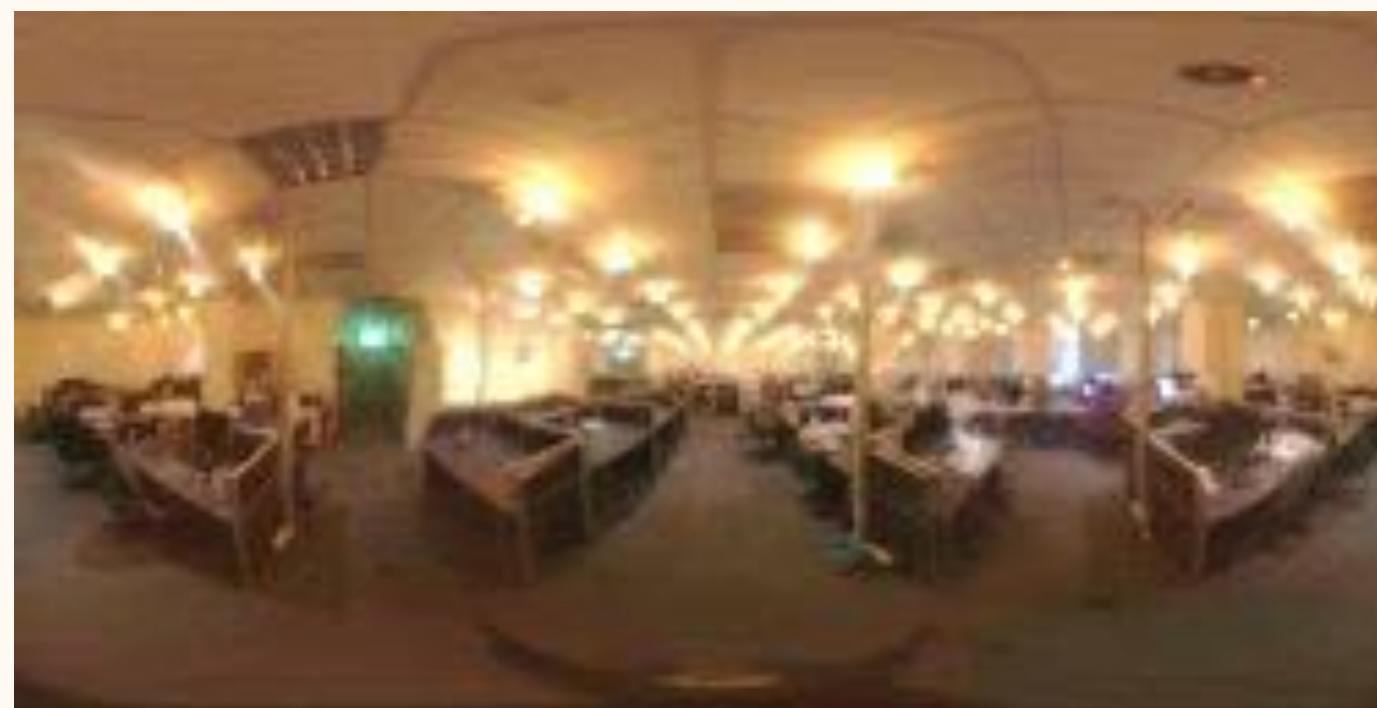
Distant RT IBL

Real-time Image-based Lighting

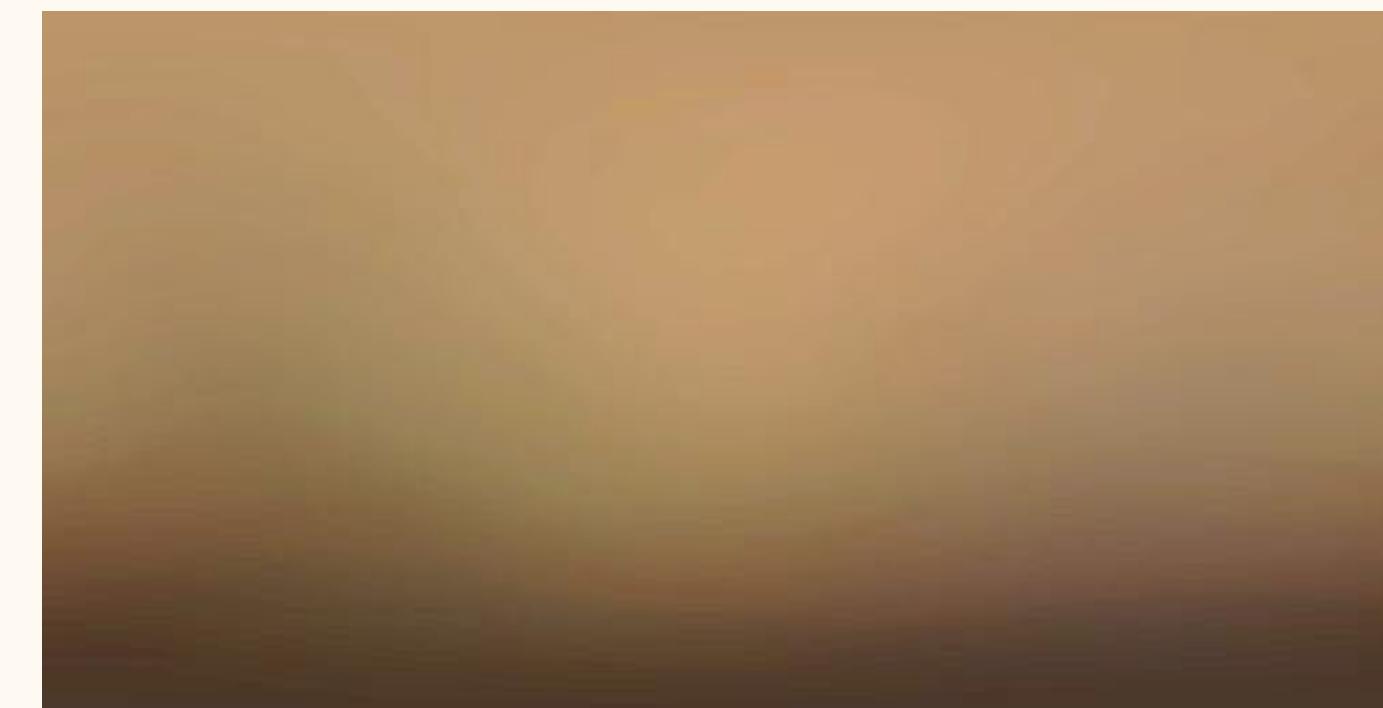
Diffuse/Specular Environment Mapping

Introduction to Distant Realtime IBL

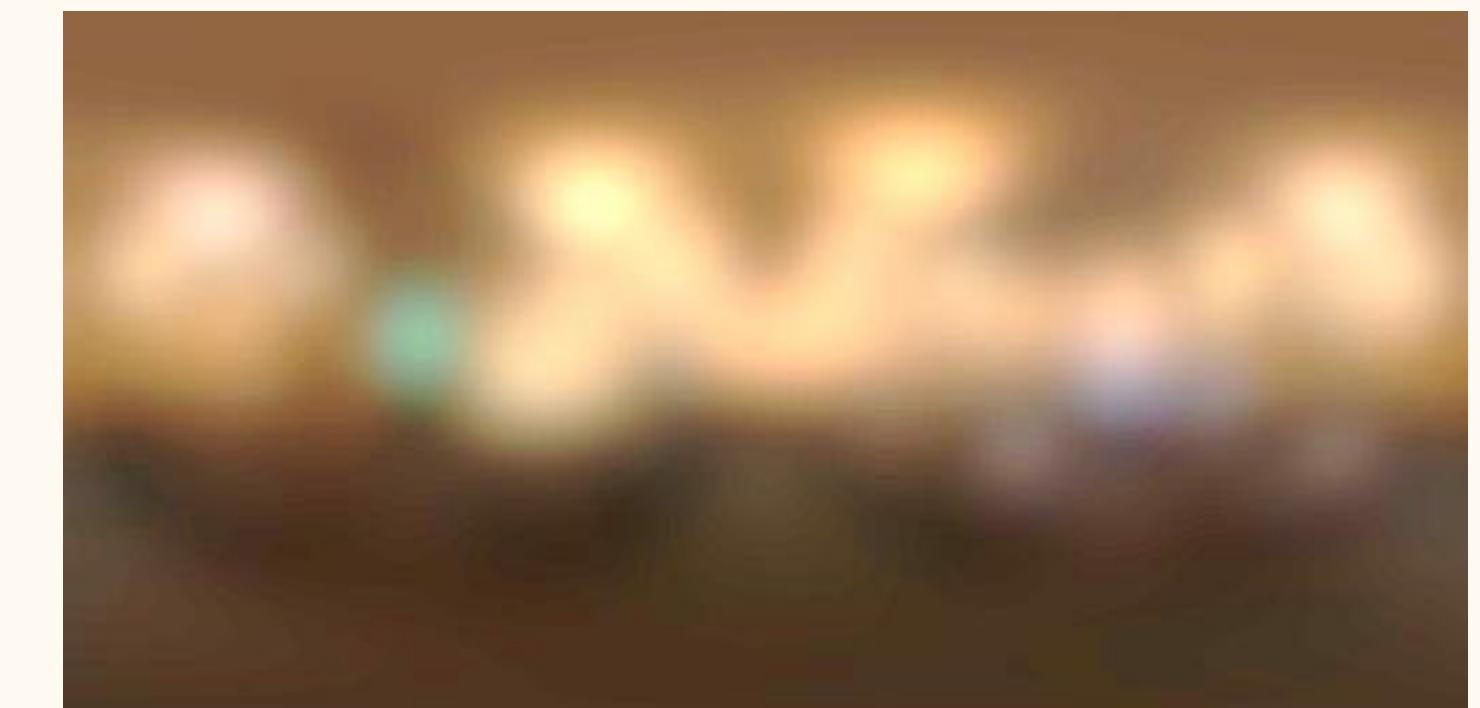
- The method is the same as reflection effect but :
 - Use a baked diffuse environment map for diffuse lighting
 - Use a baked specular environment map for specular lighting
 - What is the “baked” meaning ?
 - Prepare the data before use it !
 - It’s an offline job usually but now we can do it in real-time ...
- “Distant” means we map one panorama image to one global sphere.



HDR environment map

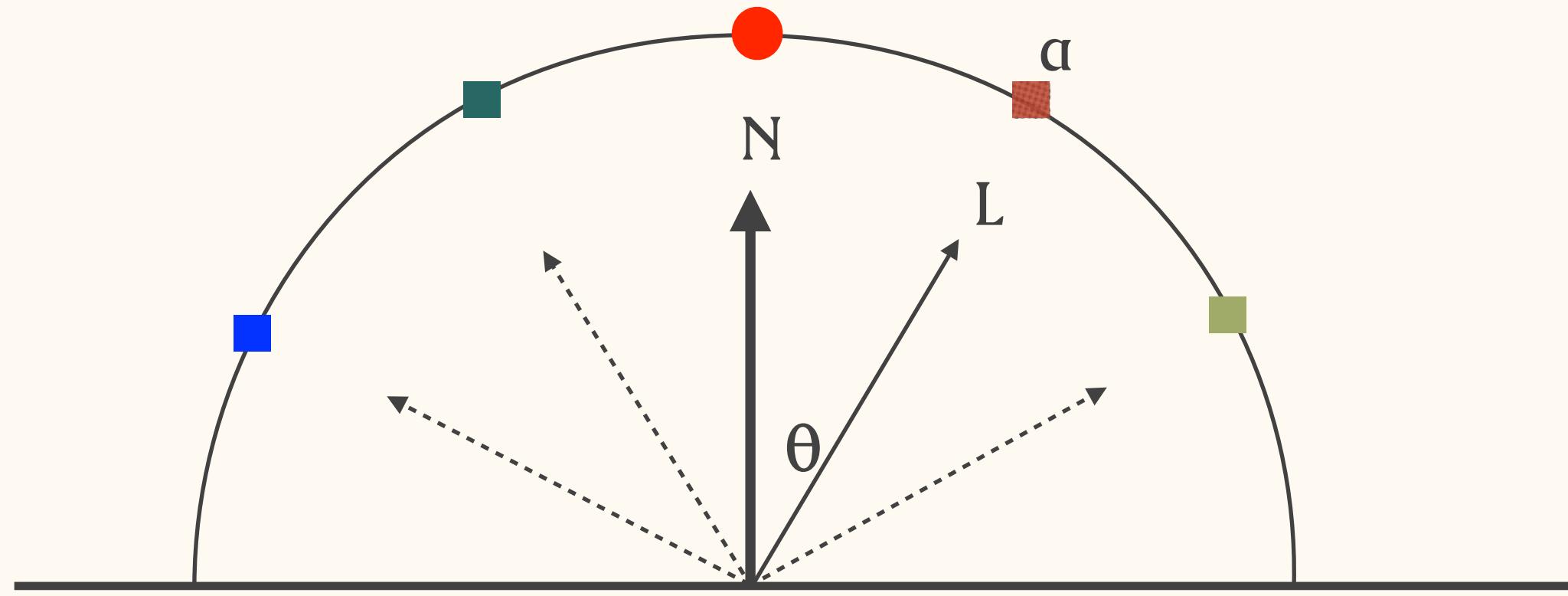


Diffuse environment map



Specular environment map

Bake Diffuse/Specular Environment Map

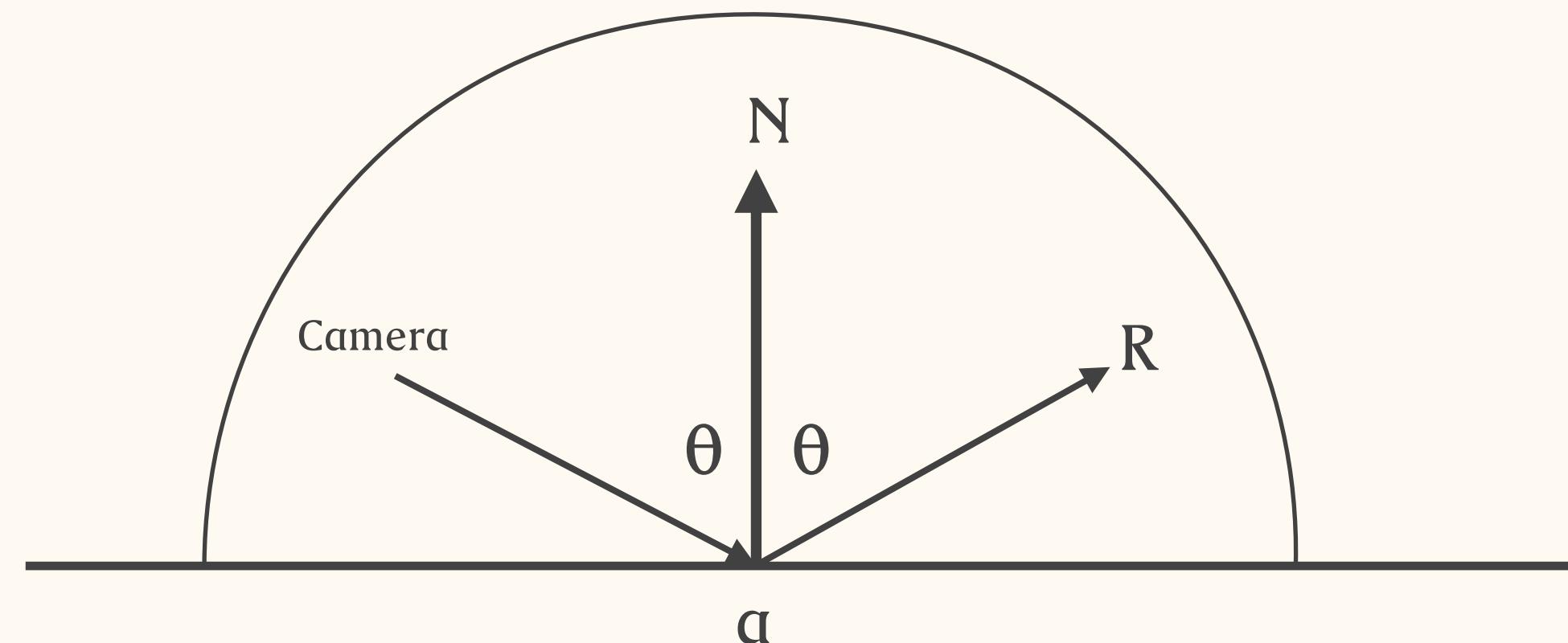


$$\text{Effective Lighting from } a = \textcolor{red}{\blacksquare} * (N \cdot L)^n$$
$$\text{Result} = \sum I_i * (N \cdot L_i)$$

- Create a new texture for the diffuse environment map
- For the UV position of the each HDR map pixel, \bullet :
 - 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

Use Diffuse/Specular Environment Map

- Use lat-long map (latitude-longitude map) as the environment map format
- For diffuse term :
 - Use the normal vector, N , to “hit” the environment map to get the average lighting for the rendering from the image as the light source.
- For specular term :
 - Use the reflection vector, R , of the camera direction to “hit” specular environment map



Distant Realtime IBL 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;

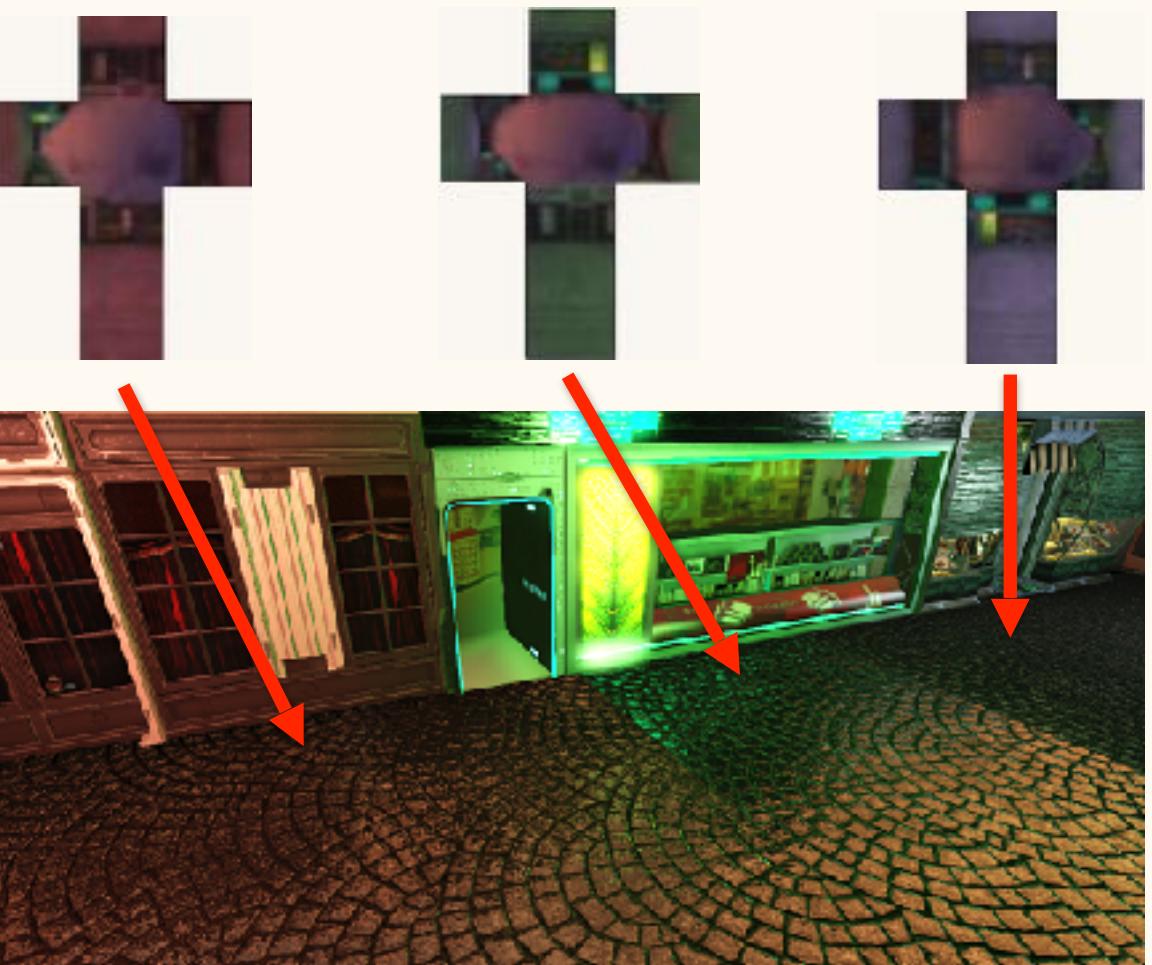
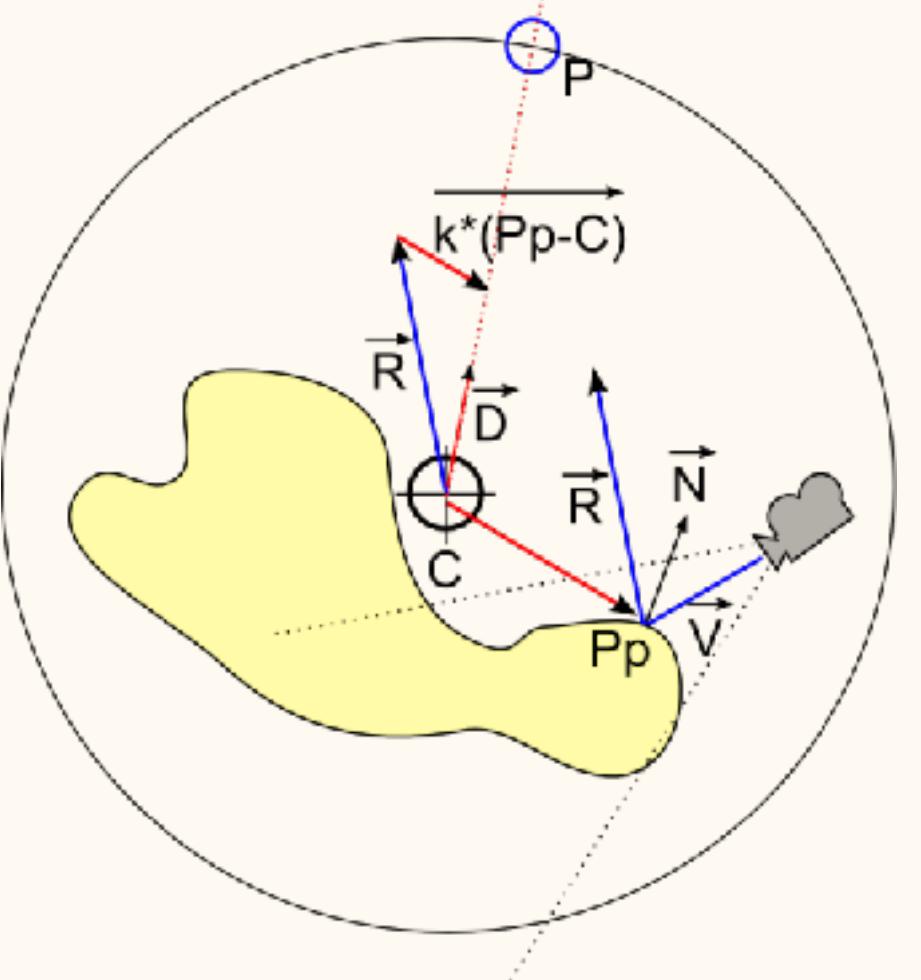
// get color texture data
float4 color = dif*txDiffuse.Sample(txDiffuseSampler, in1.tex0);

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

return rgba;
}
```

Distant RT IBL 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



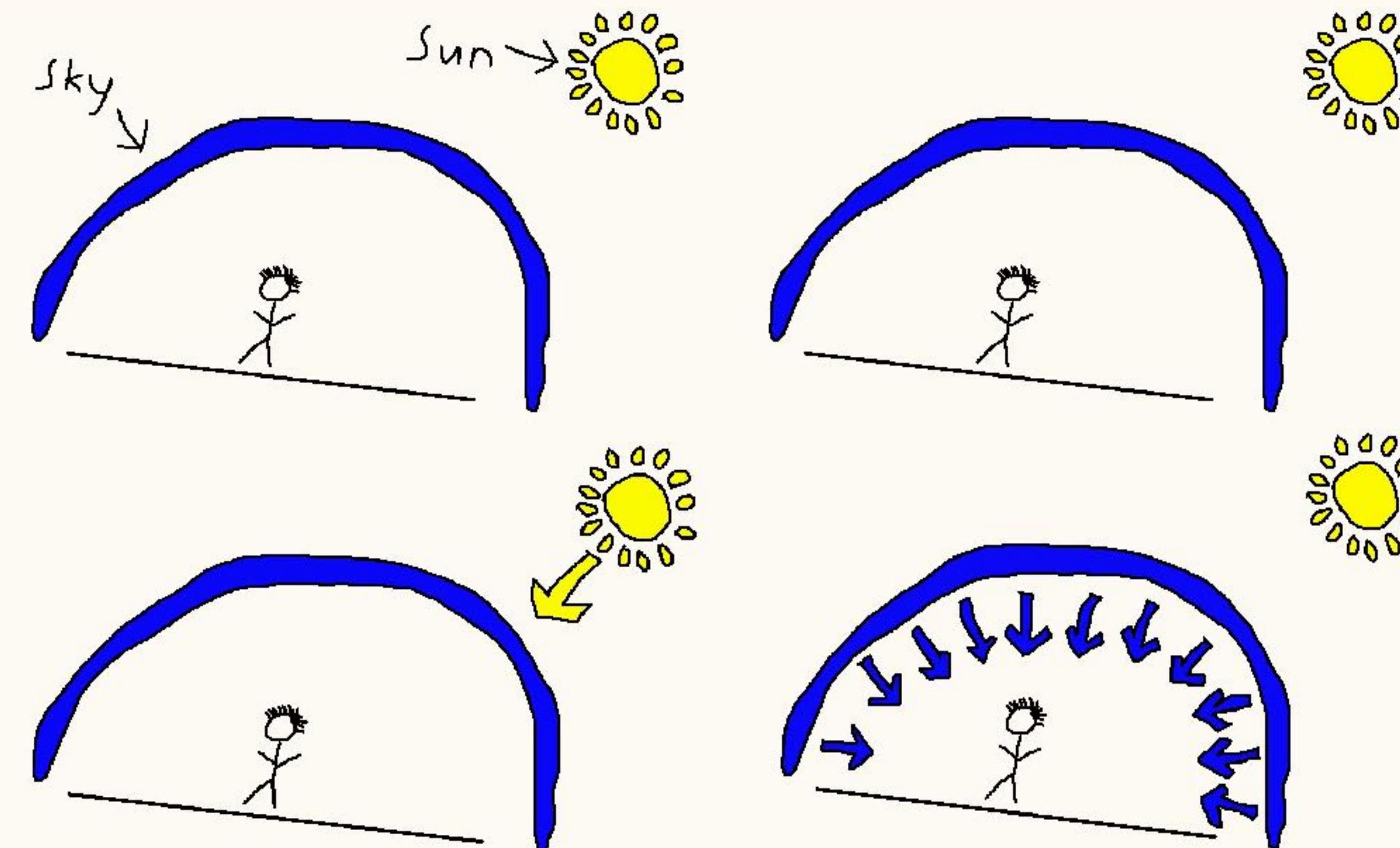
Local RT IBL References

- “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
- Bjørke, “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 ?

- Sun is Directional Light – Shadow!
- Sky is Hemisphere Light – Ambient Occlusion



What's “Ambient Occlusion“ ?

- Original method :
 - Reference : GPU Gems, Chapter 17, P. 279-292
 - The technique was originally developed by Hayden Landis (2002) and colleagues in Industrial Lights & Magic (ILM) for film production.
 - Overview
 - Pre-processing steps
 - Calculating the accessibility of light to the surface
 - Perfectly work with image-based lighting but static ...
 - AO in short
- For real-time application :
 - Screen-space ambient occlusion (SSAO)
 - SSAO != AO

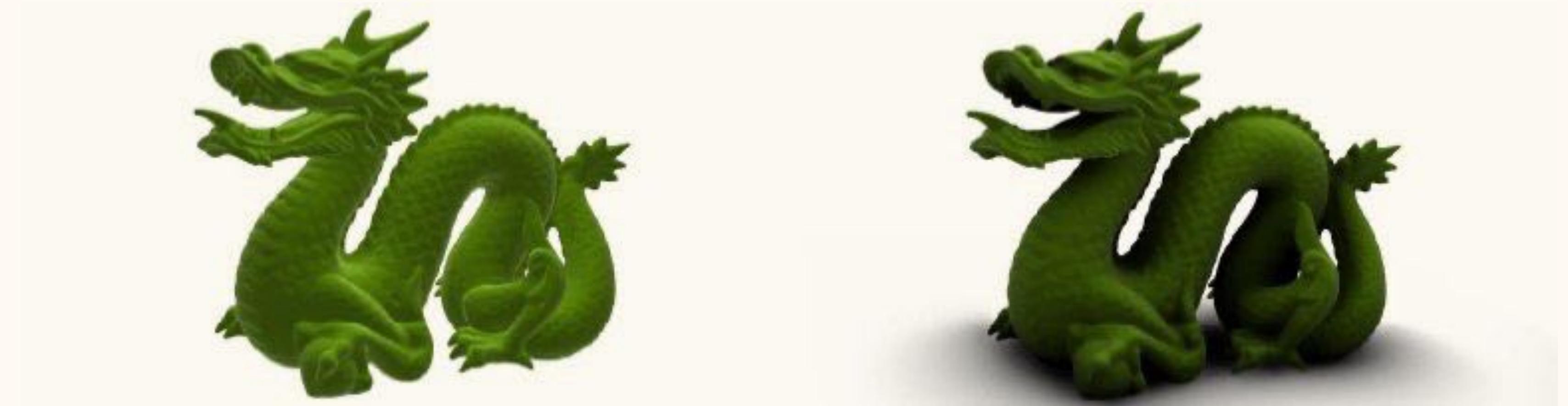
Why Ambient Occlusion ?

- Why ambient occlusion ?
 - Shadow map can not help you in this case :



Why Ambient Occlusion ?

- Given perceptual clues of depth, curvature and spatial proximity



Without AO

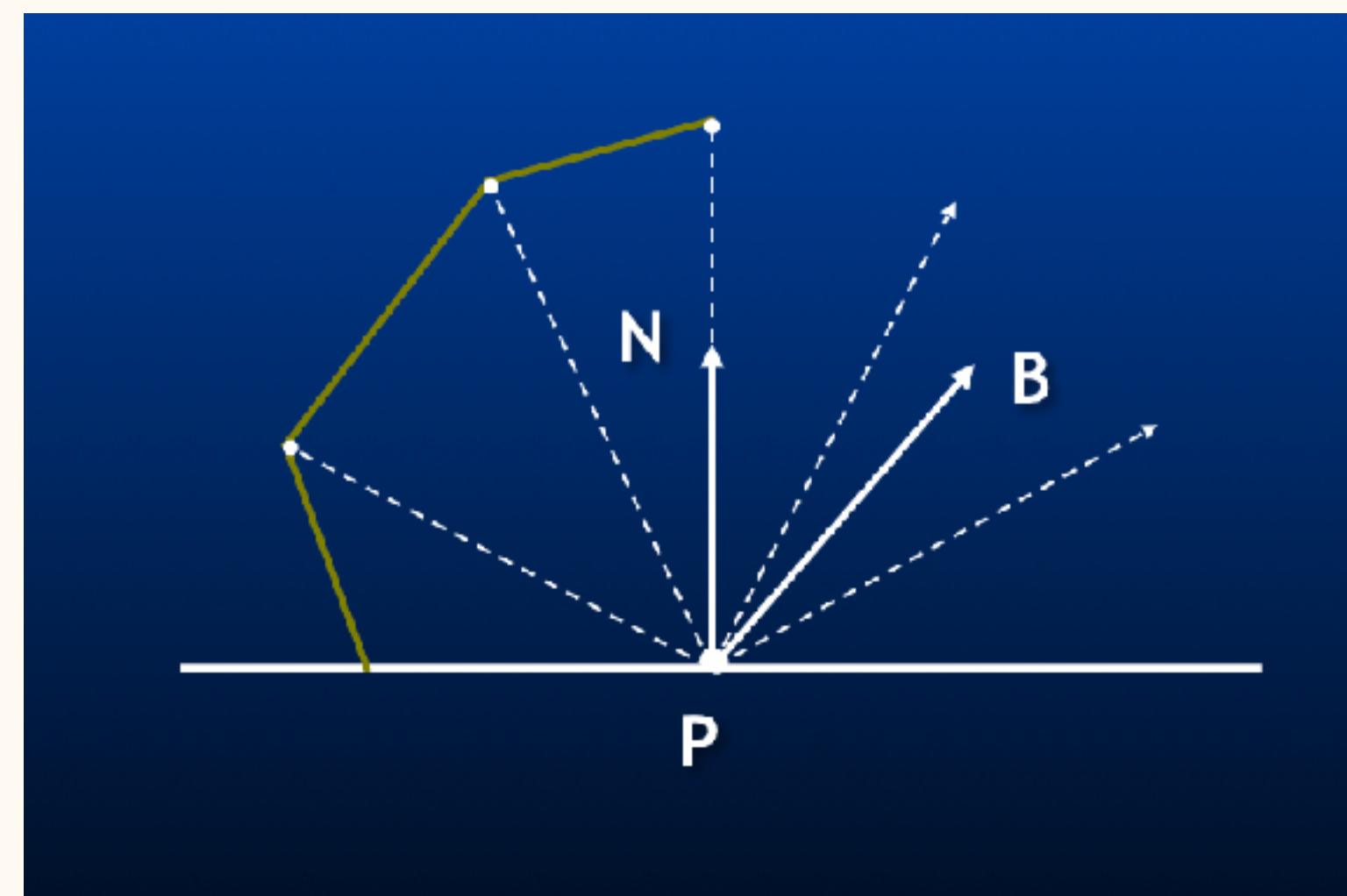
With AO

Screen Space Ambient Occlusion

- Screen-space Ambient Occlusion
 - SSAO in short
- First appearing in CryEngine 2 for game industry
 - MITTRING, M. 2007. “Finding next gen: Cry Engine 2”. In SIGGRAPH ’07: ACM SIGGRAPH 2007 courses.
- Calculate ambient occlusion in games using screen-space buffers
 - Z buffer + normals

Generate Ambient Occlusion on Vertex

- On each vertex, there are two things needed to know :
 - Light accessibility
 - What fraction of the hemisphere above that point P is un-occluded by other parts of the model
 - Average direction of un-occluded incident light, 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 Quality

- Two factors :
 - Number of ray sampled
 - More rays need more time to generate
 - A better sampling algorithm can save the cost
 - Low resolution or high resolution geometric models ?
 - Solution :
 - Ambient occlusion map
 - Can be prepared by artists

Ray Generation

- Two methods :
 - Rejection sampling
 - Sampling rate is very critical
 - Monte Carlo sampling algorithms
 - Better distribution solution

Rejection Sampling

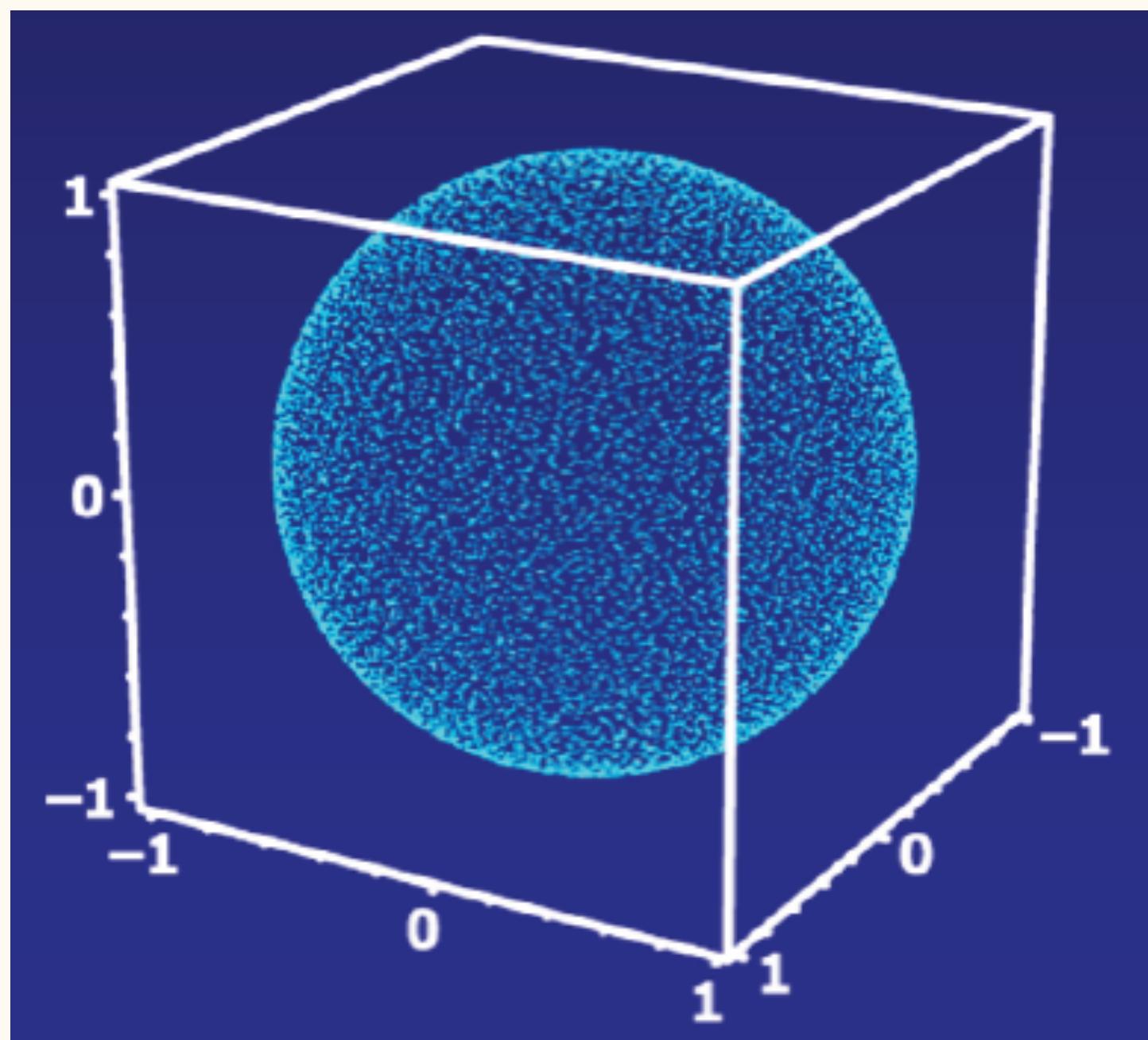
- Algorithm :

```
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));
}
```

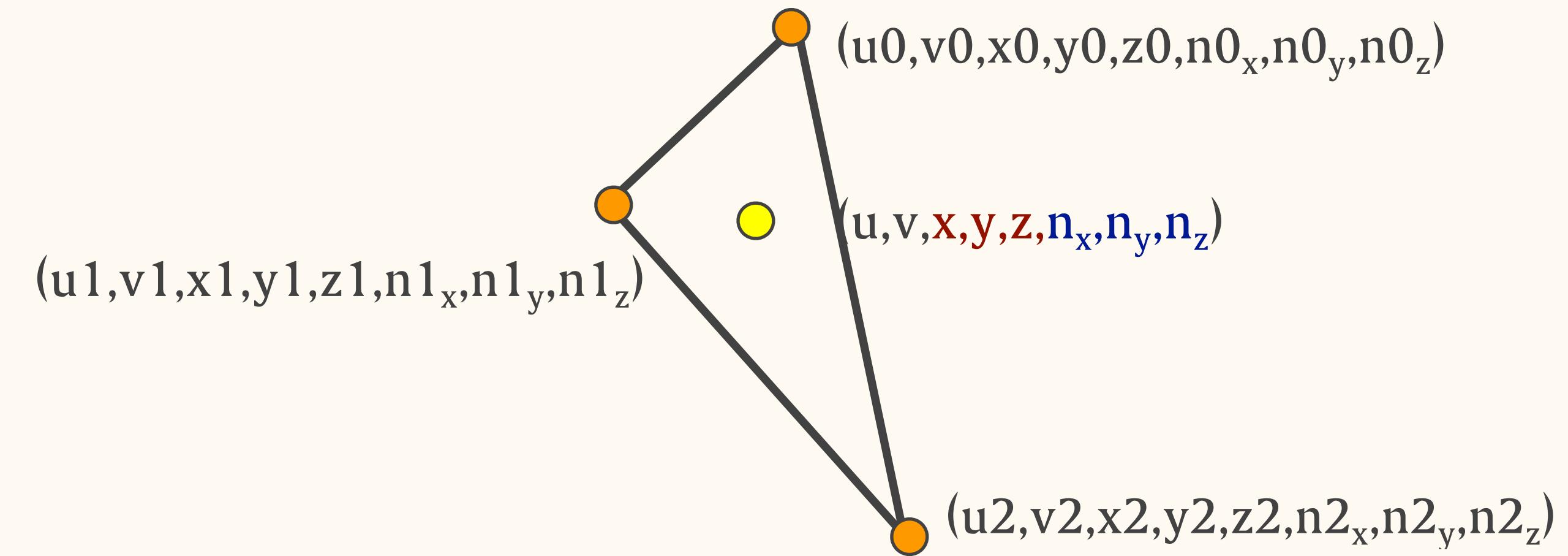
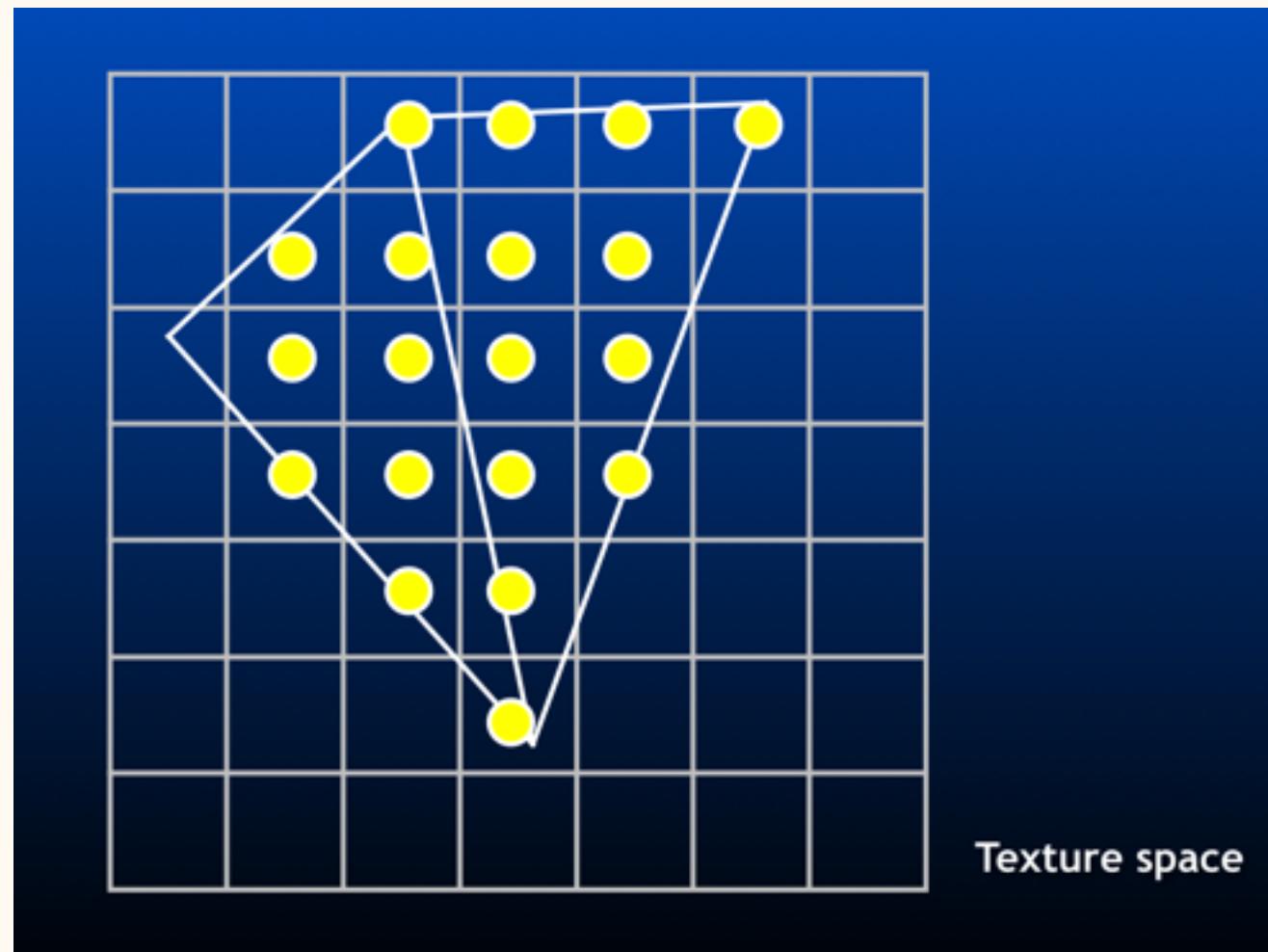
Monte Carlo Sampling

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



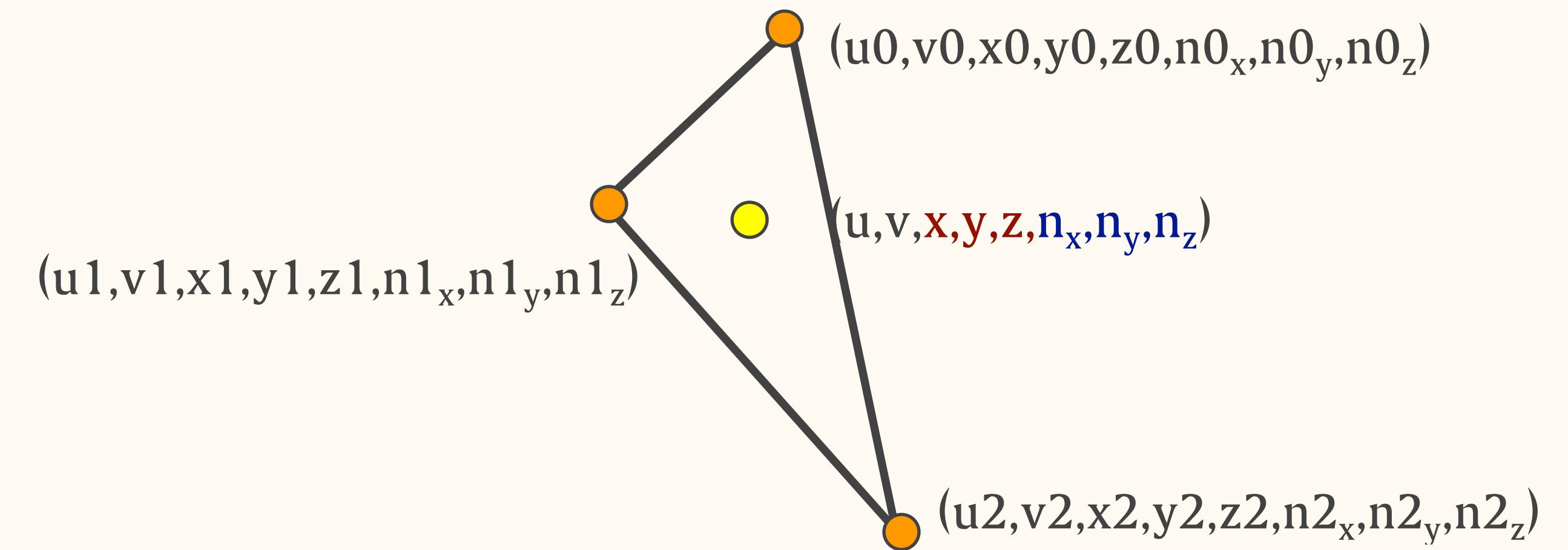
Generate Ambient Occlusion Map

- For low resolution geometry model, we save ambient occlusion data in the ambient occlusion map (texture)
- Steps
 - Unwrap the model on texture



Generate Ambient Occlusion Map

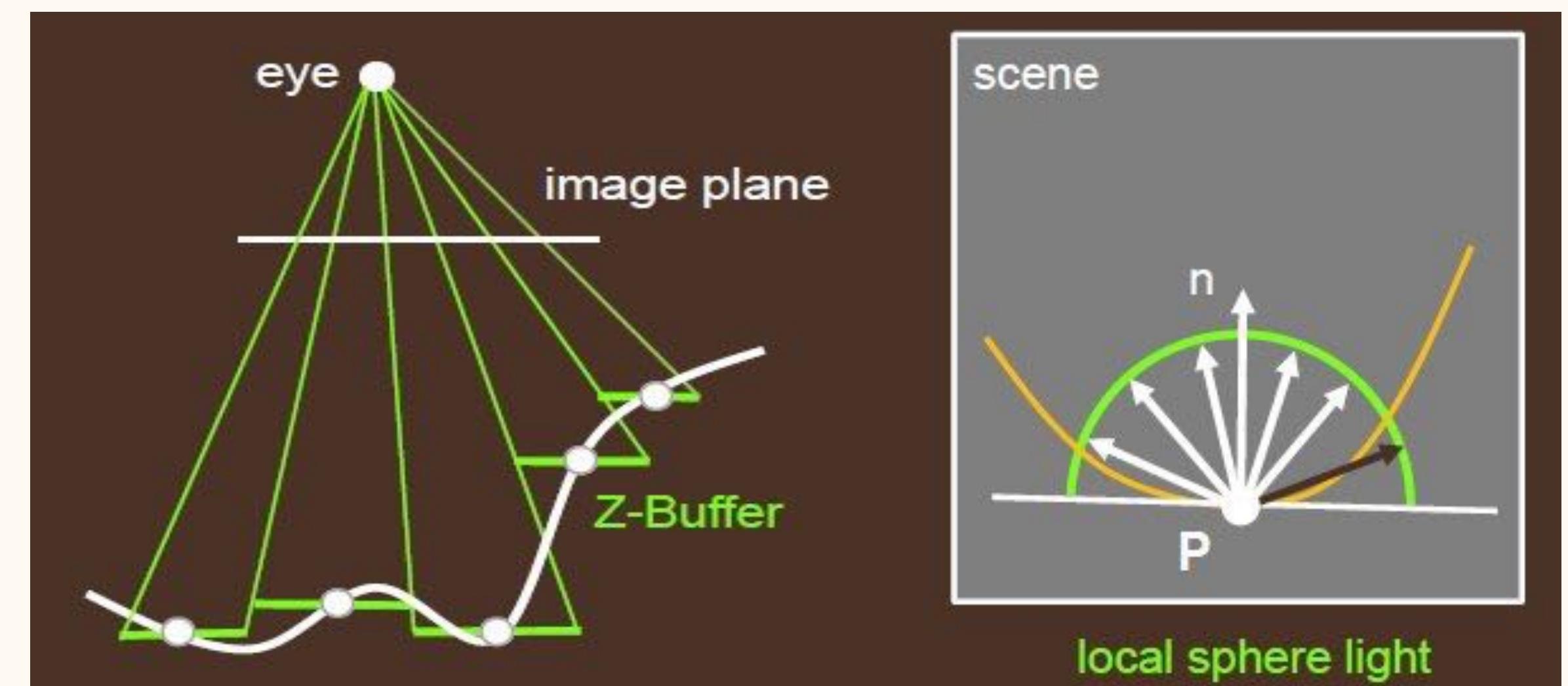
- Use texture coordinate to interpolate the texel's position and normal vector in 3D world



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

Screen Space Ambient Occlusion

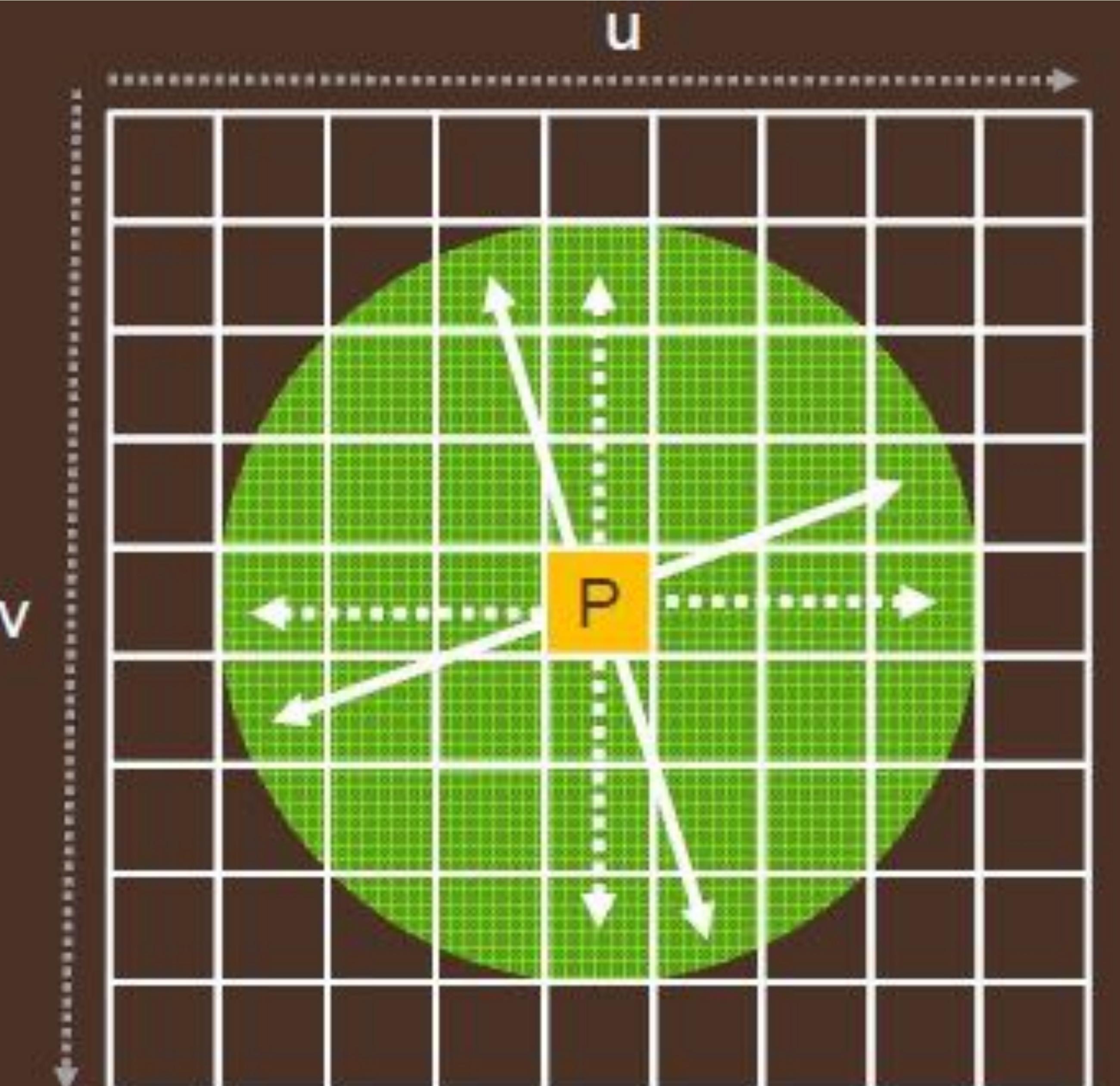
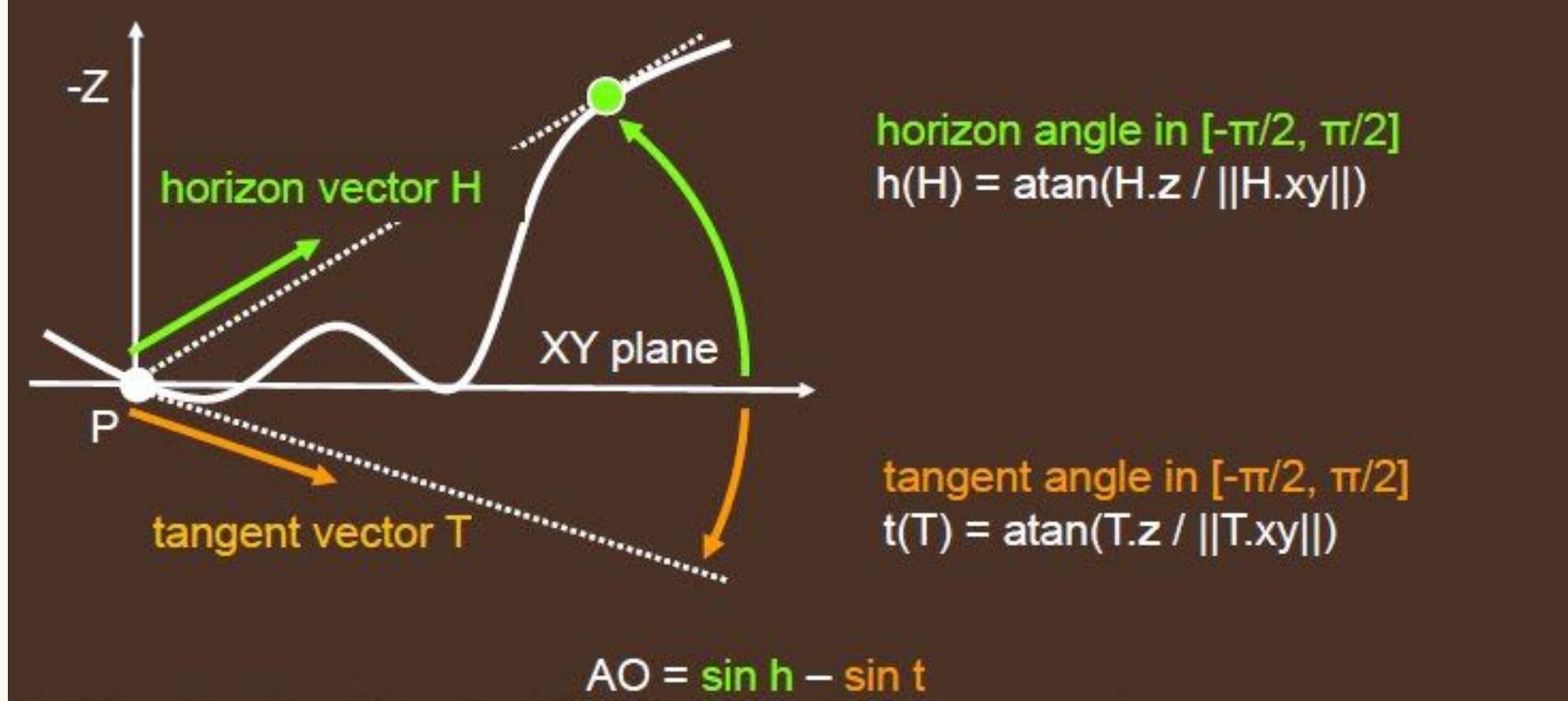
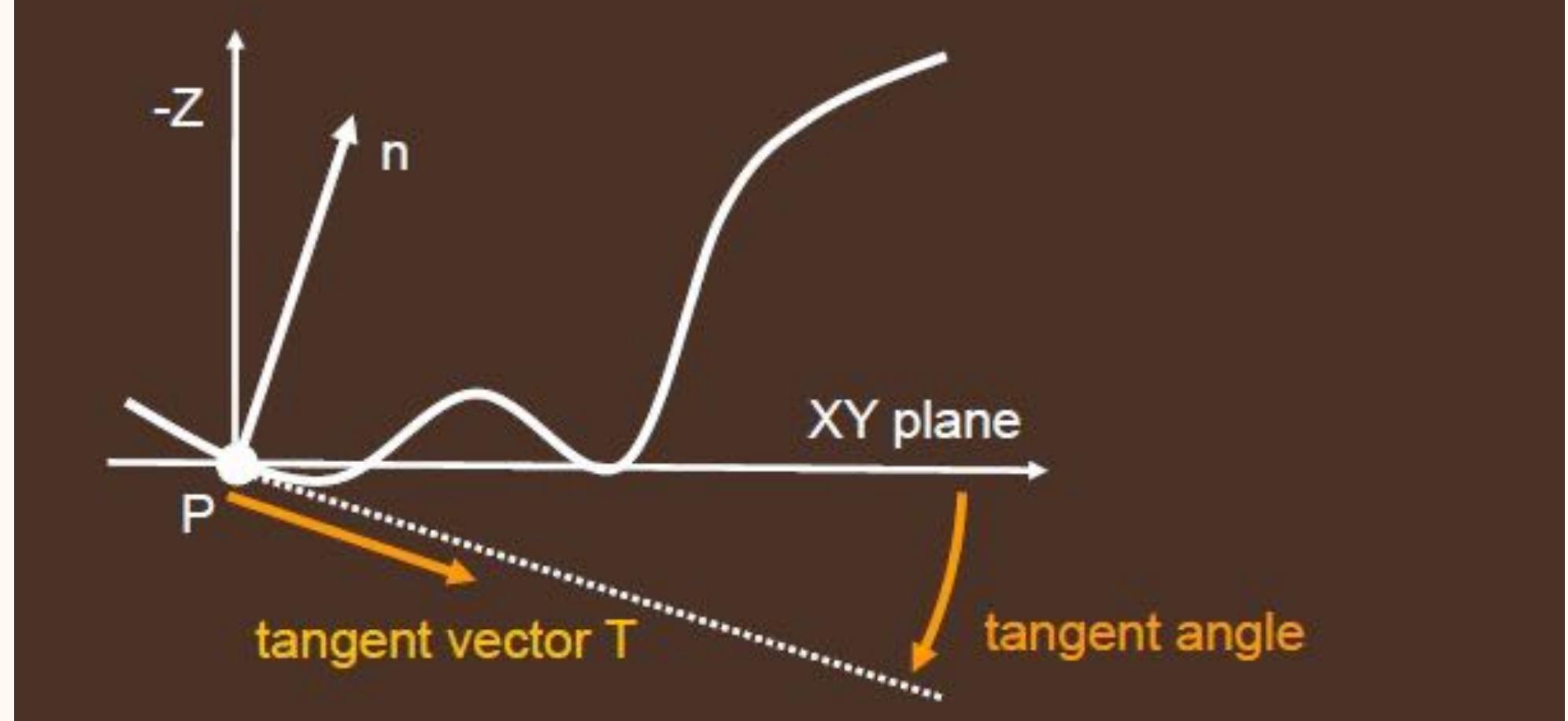
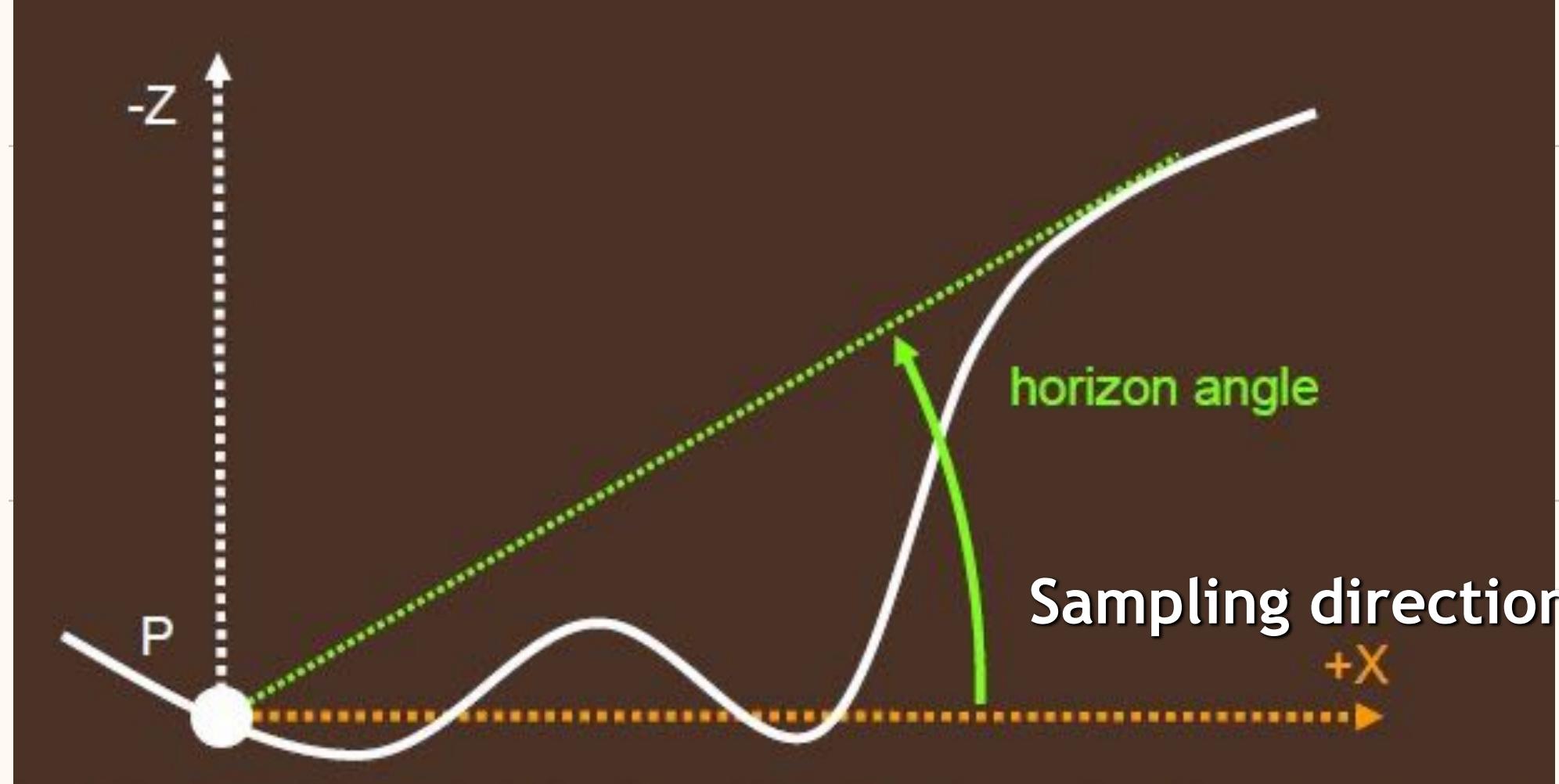
- Screen space
 - Data in screen coordinates
 - Render the 3D geometry data in buffers
- Render approximate AO for dynamic scenes with no pre-computations
- Input
 - Depth buffer (Height field), $Z = f(x, y)$
 - Normal buffer



Screen Space Ambient Occlusion

- Popular implementation is based on :
 - “Image-Space Horizon-Based Ambient Occlusion”
 - By Louis Bavoil and Miguel Sainz @ nVIDIA
 - Published in ShaderX 7
 - Presented in SIGGRAPH 2008
 - HBAO



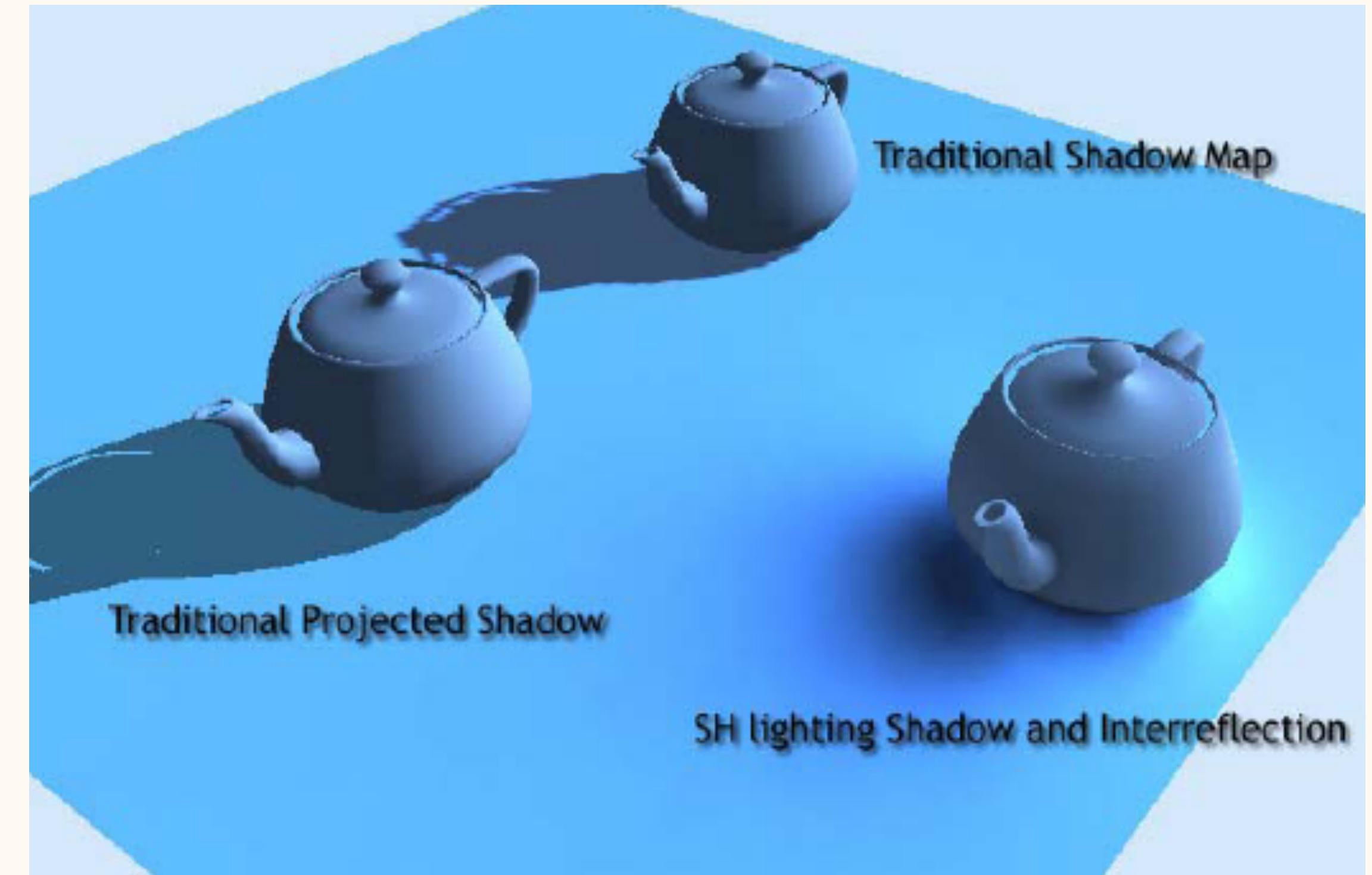
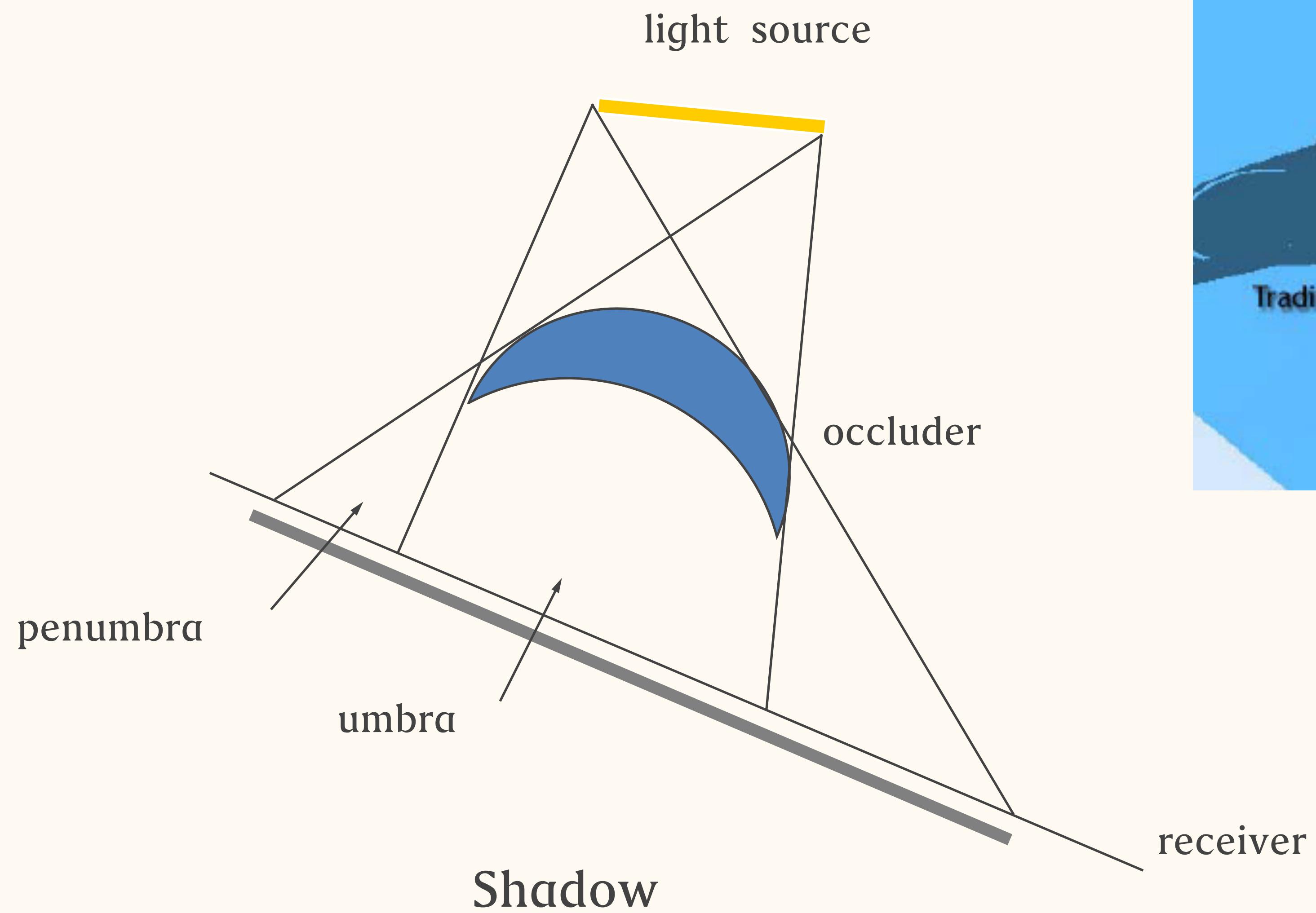


Example with 4 directions / pixel

Screen Space Ambient Occlusion



Shadow Map



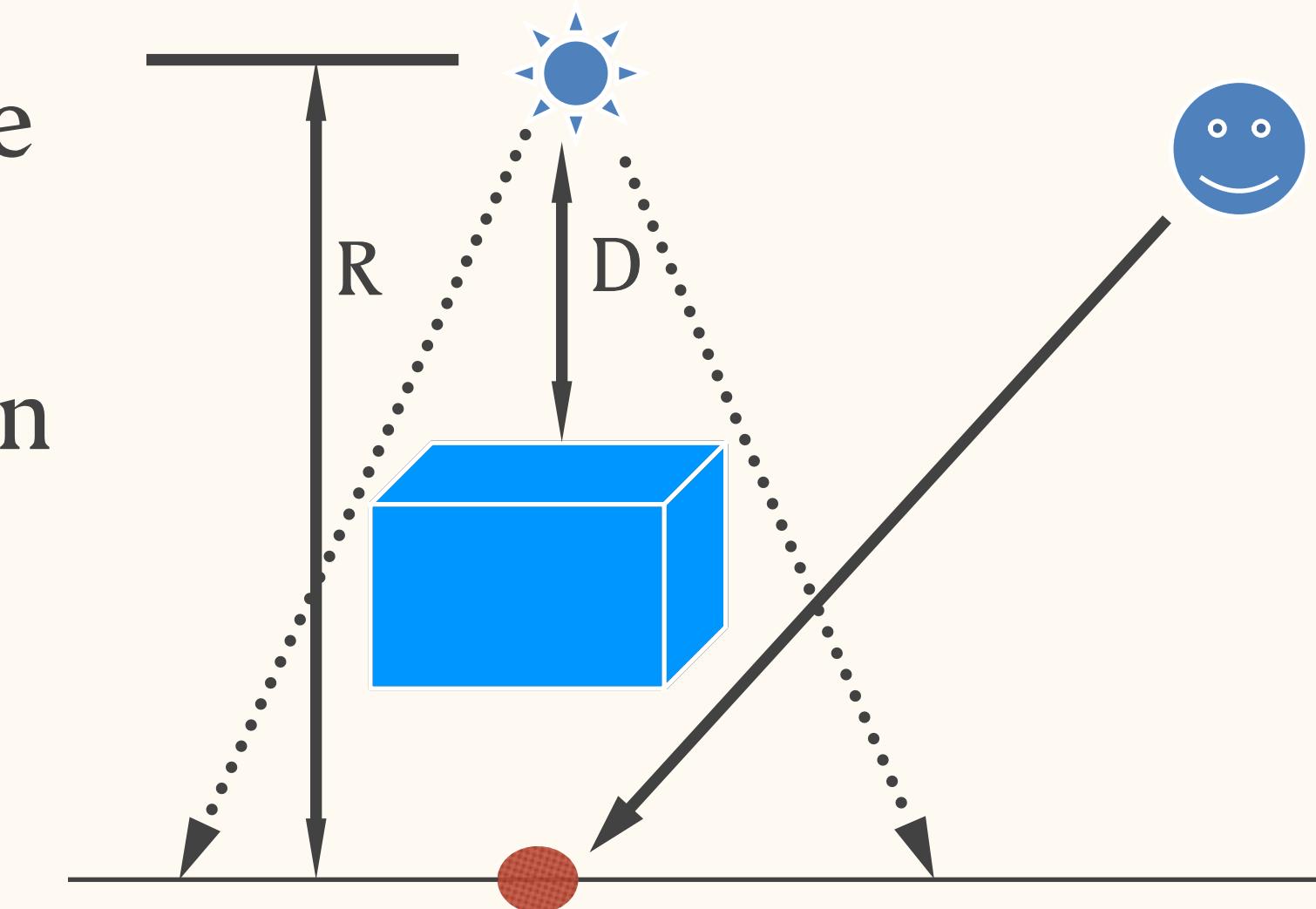
Introduction to Shadow Map

- Background
 - By Lance Williams
 - SIGGRAPH 1978 Paper
 - “Casting Curved Shadows on Curved Surfaces”
- Concept :
 - A point is shadowed if something interrupts the straight line path between the light and that point.
 - To test a point for occlusion takes two passes :
 - The scene is rendered from the light's view point, while the depth of each pixel is recorded in a depth texture, called a depth map.

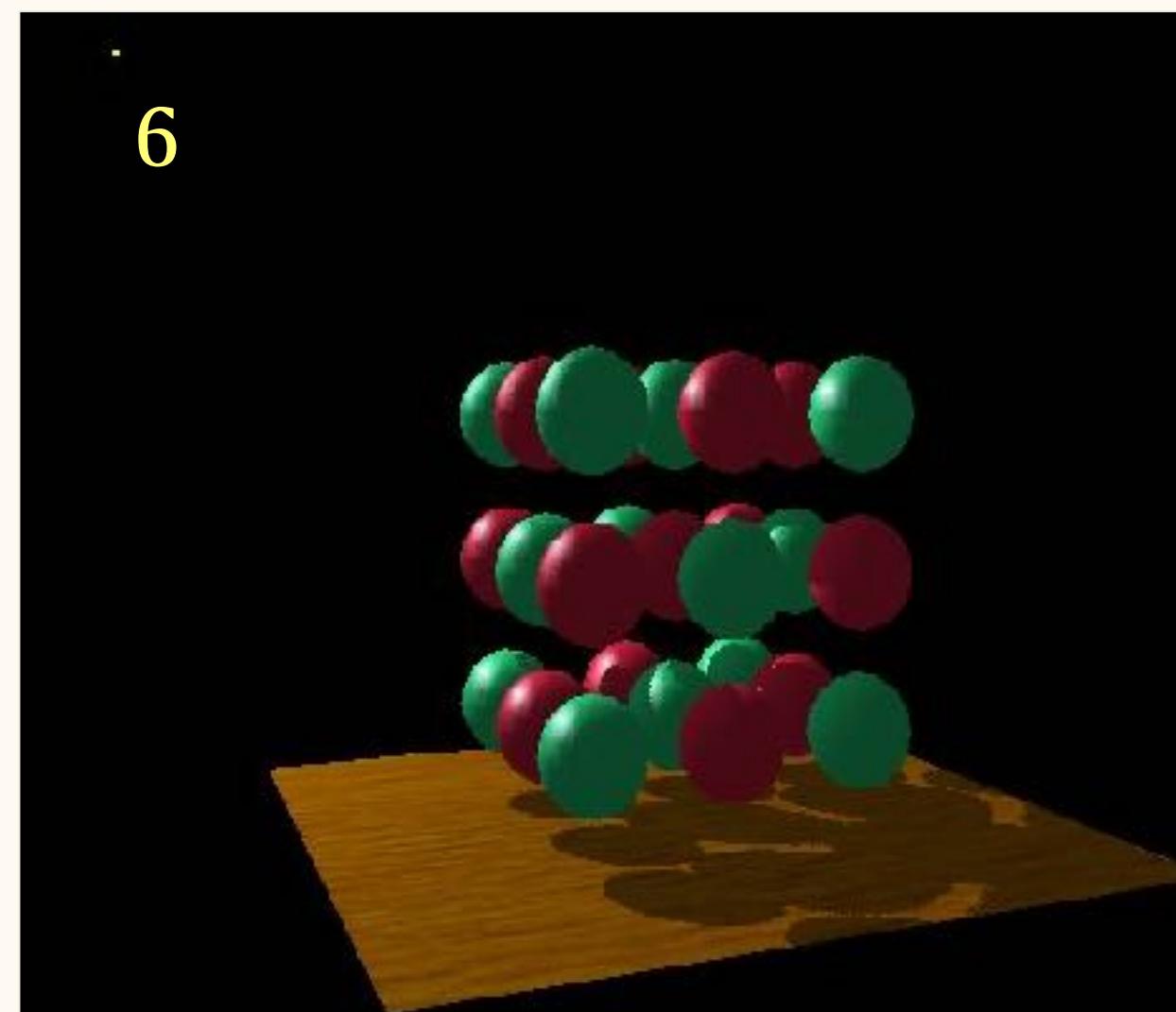
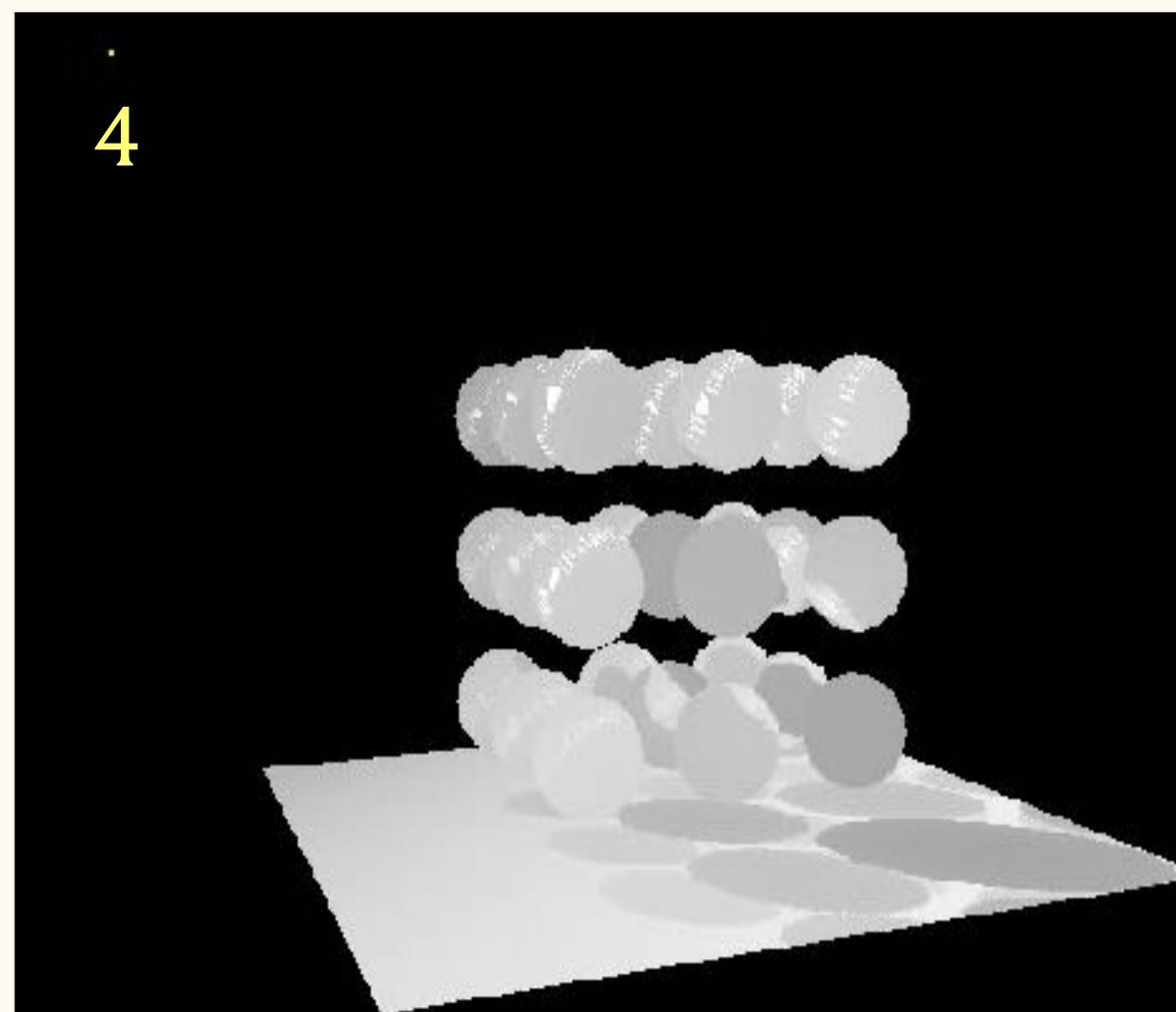
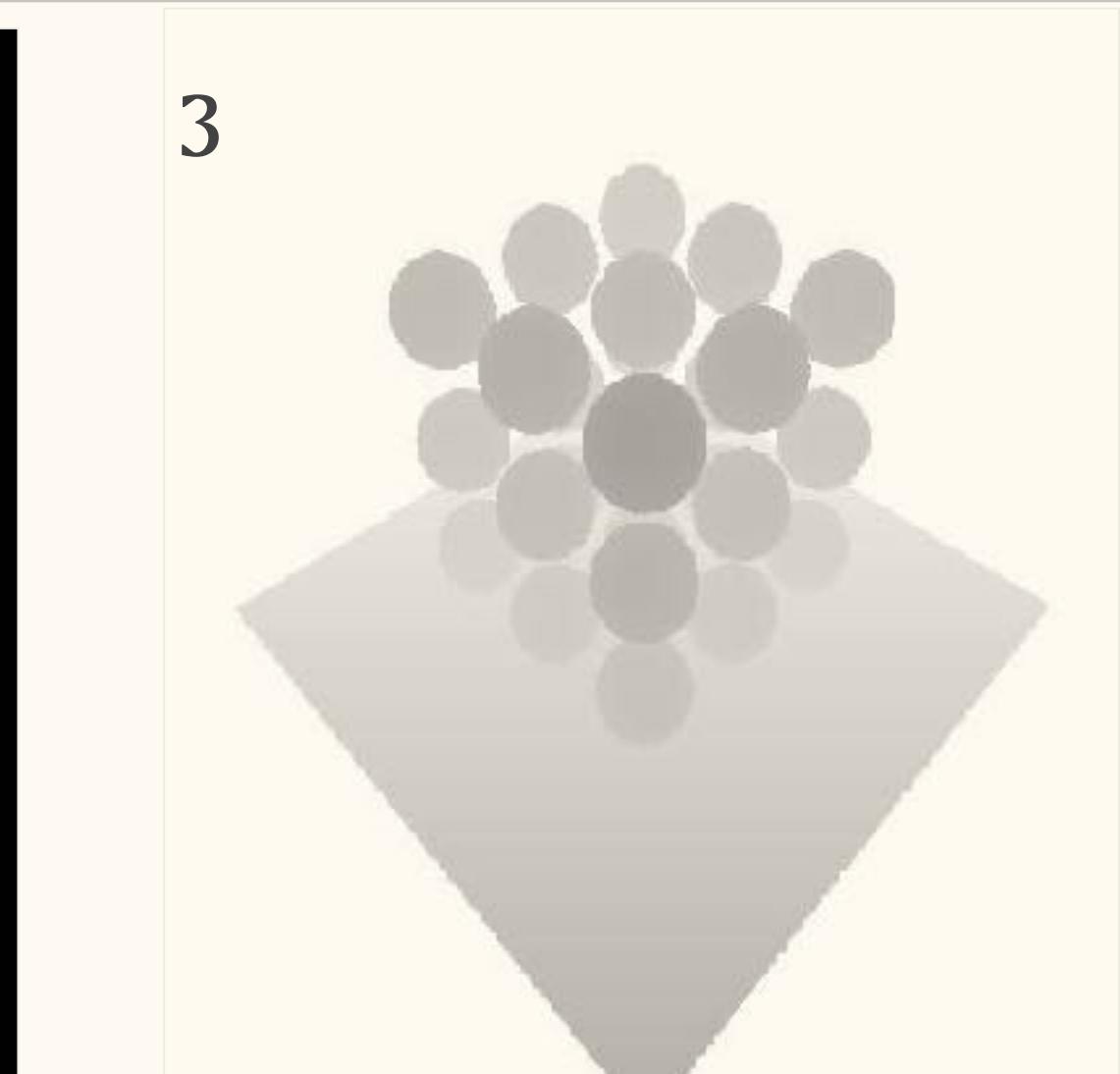
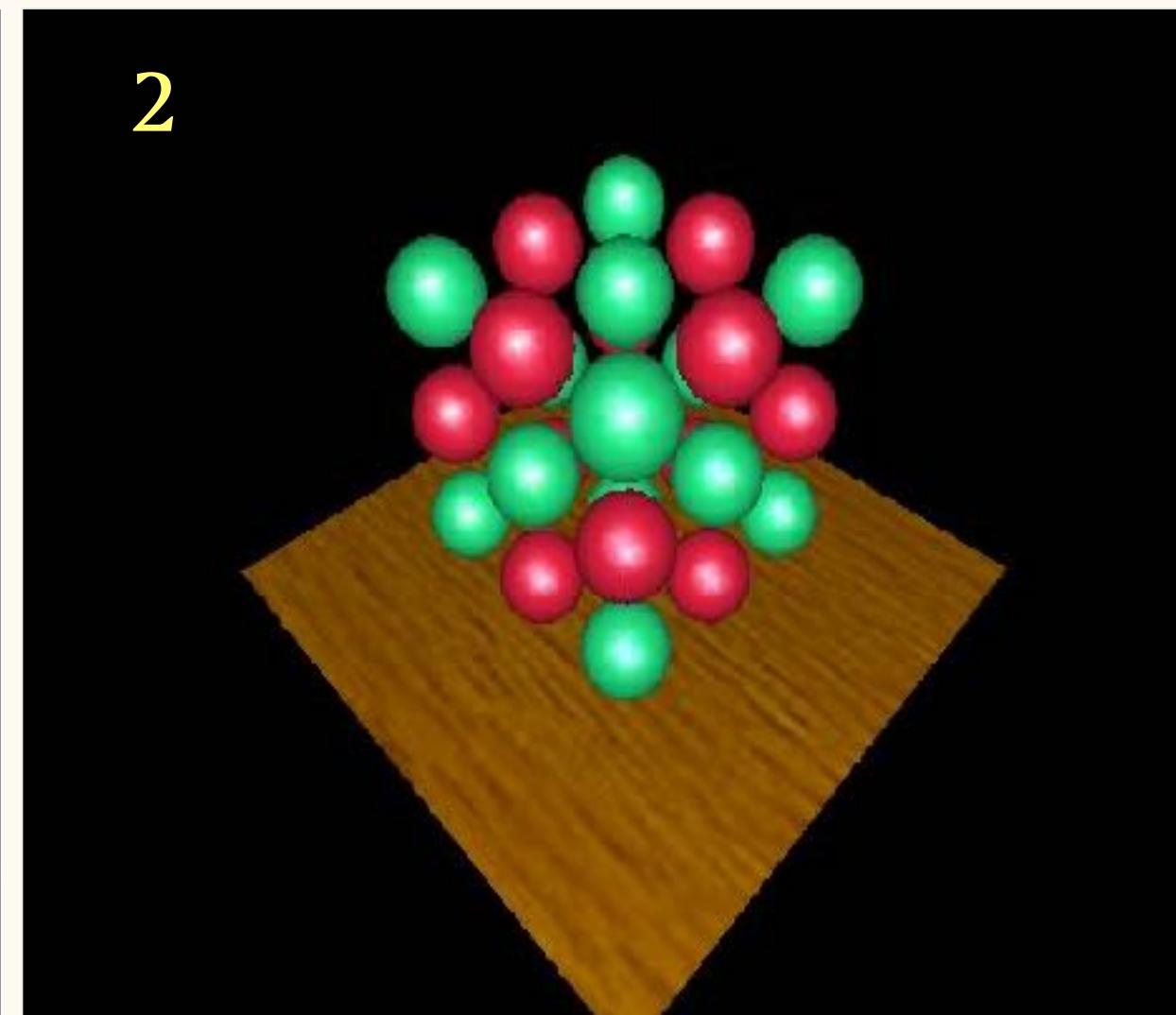
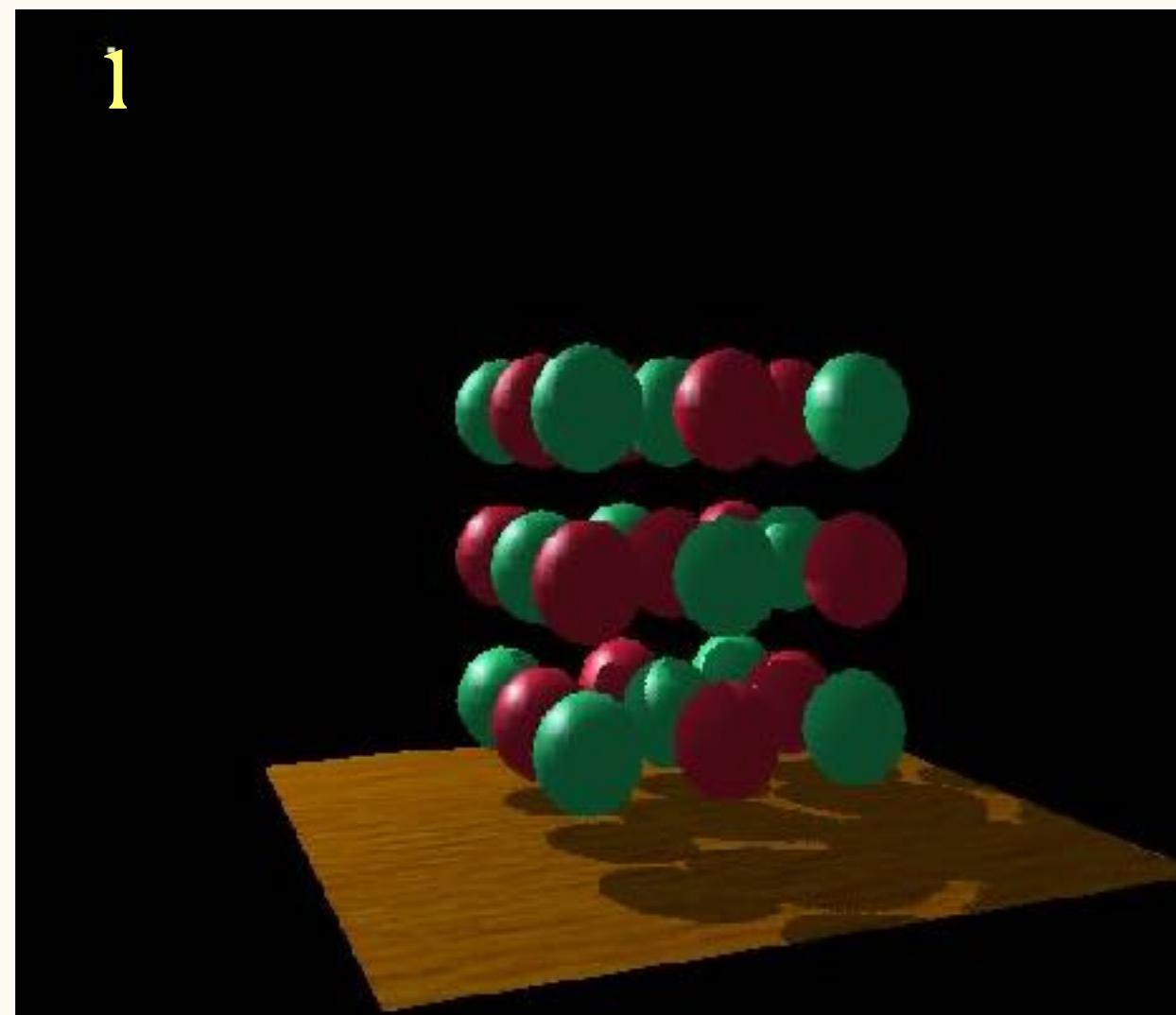


Introduction to Shadow Map

- Then the scene is rendered from eye position but with the shadow map projected down from the light onto the scene using standard projective texturing.
- At each pixel, the depth value from shadow map is compared with the fragment's distance from the light.
 - If D is the depth, R is the distance between light & point.
 - $R = D$: 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
 - Take the light source as the camera.
 - Spot light as a perspective camera
 - Parallel light as an orthogonal camera
 - Point light is very similar to a cubemap.
 - Render the objects' depth to a floating-point texture buffer.
- 2nd pass
 - Render the whole scene.
 - Calculate the distance of the rendered position to the light
 - Use the distance and the associated depth value saving in depth map to find if shadowed or not.

Render a Shadow Map

- Store the distance to camera in a texture
 - For spot lights and point lights
 - The value is normalized by lighting range

```
float dist2 = dot(in1.wPos, in1.wPos);  
float dist = sqrt(dist2)/lightingRange;
```



- Store the Z value to camera in a texture
 - For parallel lights
 - “depth map”

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

Projective Texture Mapping

- Arbitrary projection of a 2D images onto 3D models
 - Batman sign, spinning fan on ceiling, Shadow mapping, Spotlight effects
- Methods :
 - Project the 3D position to projection space of the light source
 - Do the homogeneous divide
 - Convert the result to texture space (be sure in left-handed on DirectX)
 - Use the result to lookup the texture

```
// convert location to screen space (lighting 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)

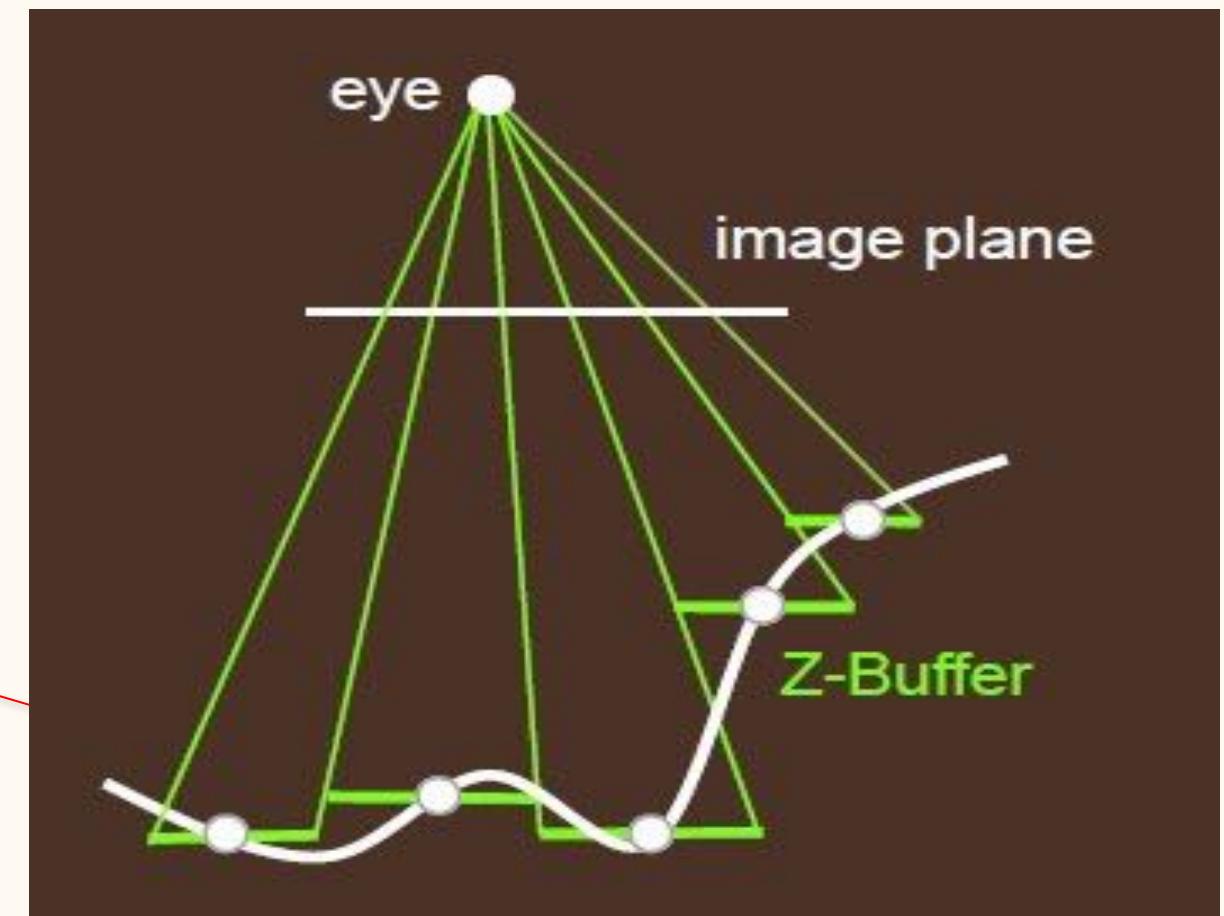
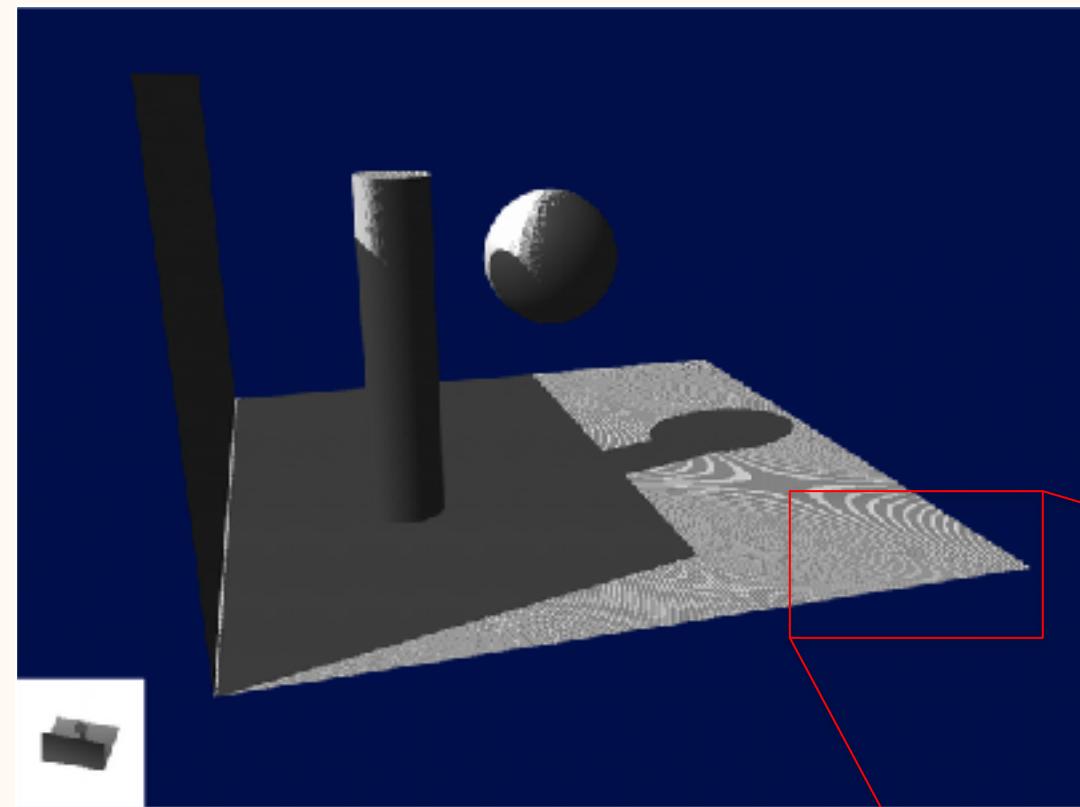
// homogeneous divide and convert to left-handed texture space (0 ~ 1)
float2 lgtTexCoord = (posLgt.xy/posLgt.w)*float2(0.5f, -0.5f) + 0.5f;
float4 smRGB = shadowMap.Sample(shadowMapSampler, lgtTexCoord);

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

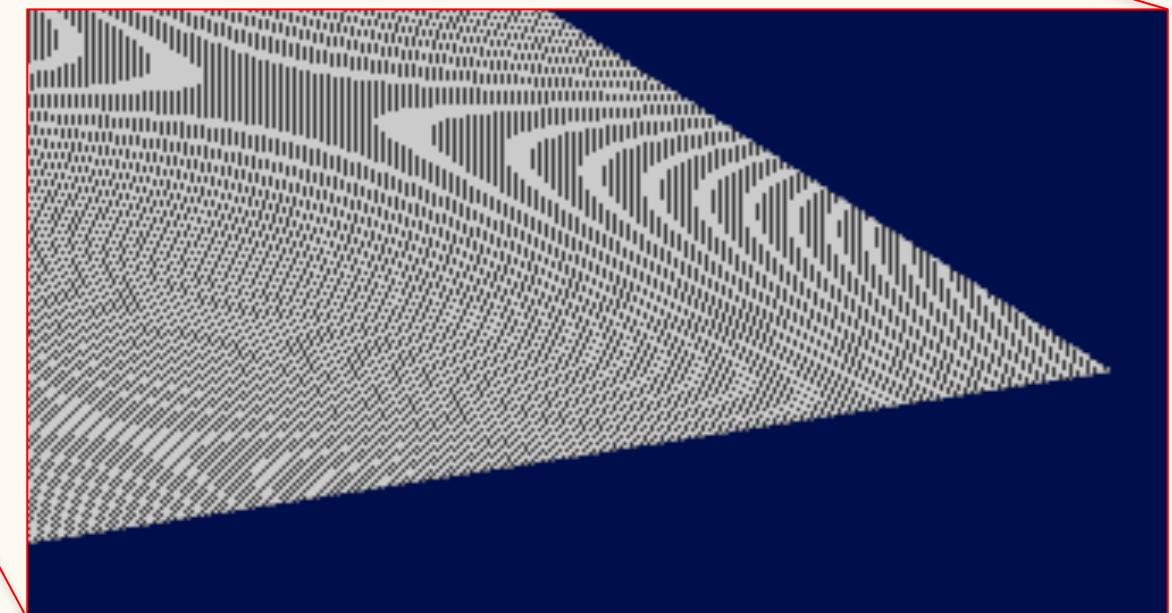
// calculate lighting weight according to the shadow map
float weight = (aToL - smRGB.x < bias) ? 1.0: 0.0;
lightIntensity *= weight;
```

Shadow Map Problems – Moire Effect

- Moire effects
 - Caused by numerical precision error
 - Self-shadowing
 - Add a tiny bias to solve it



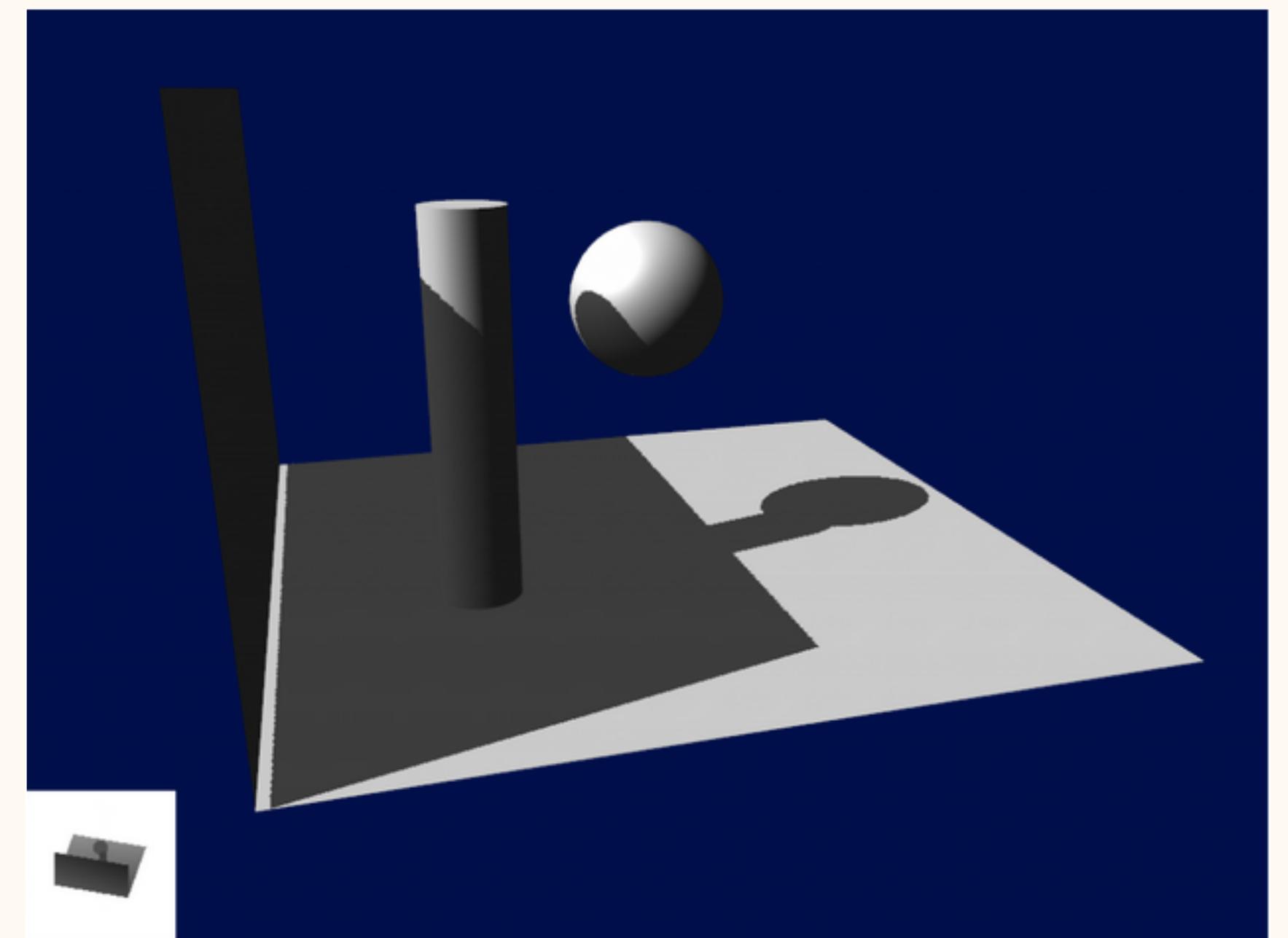
```
float bias = 0.005;  
float visibility = 1.0;  
If (sm.Sample(smSampler, uv).z < depth - bias)  
    visibility = 0.5;
```



Shadow Map Problems – Moire Effect

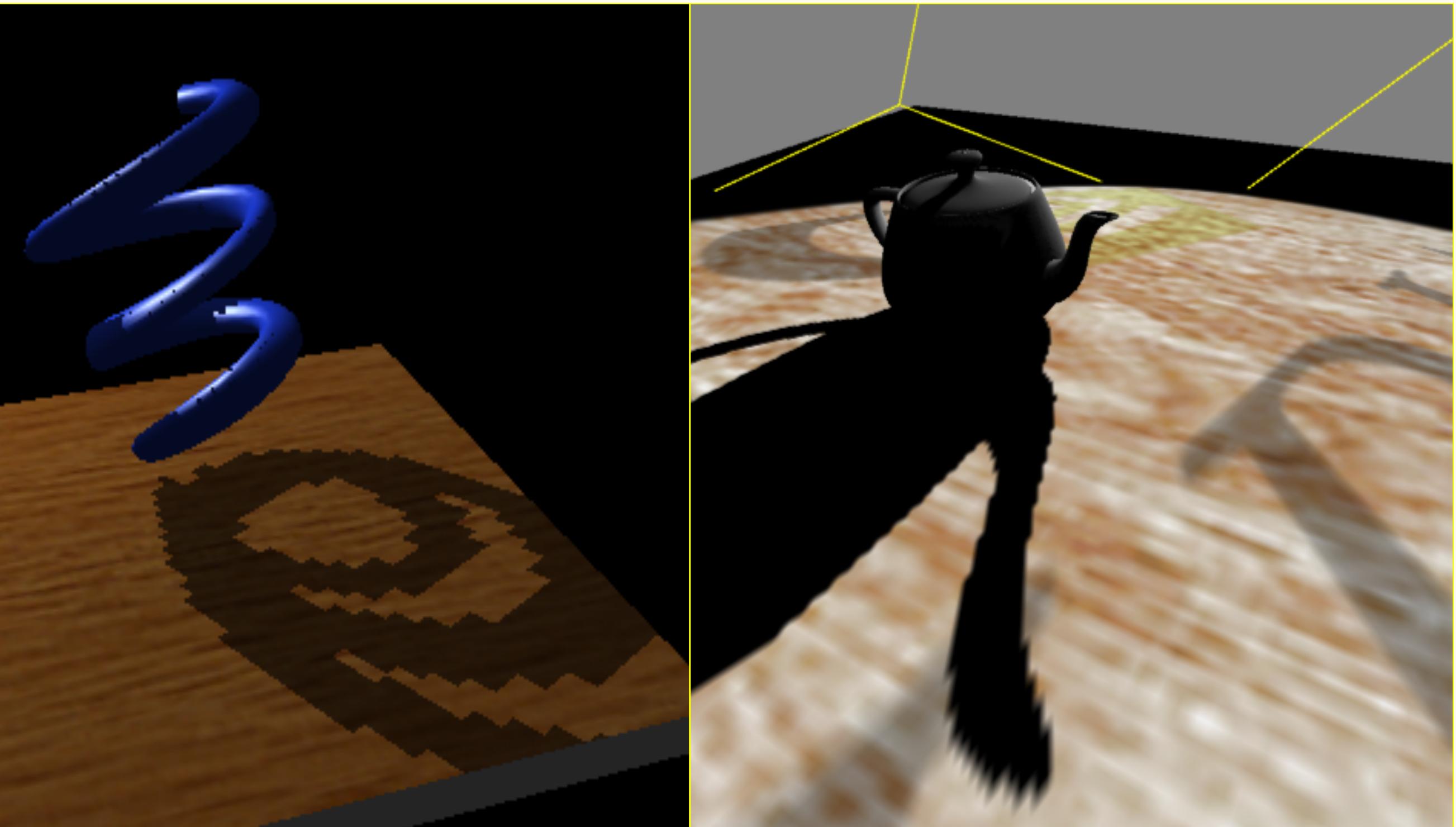
- The bias value can be adjusted according to surface curvature

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



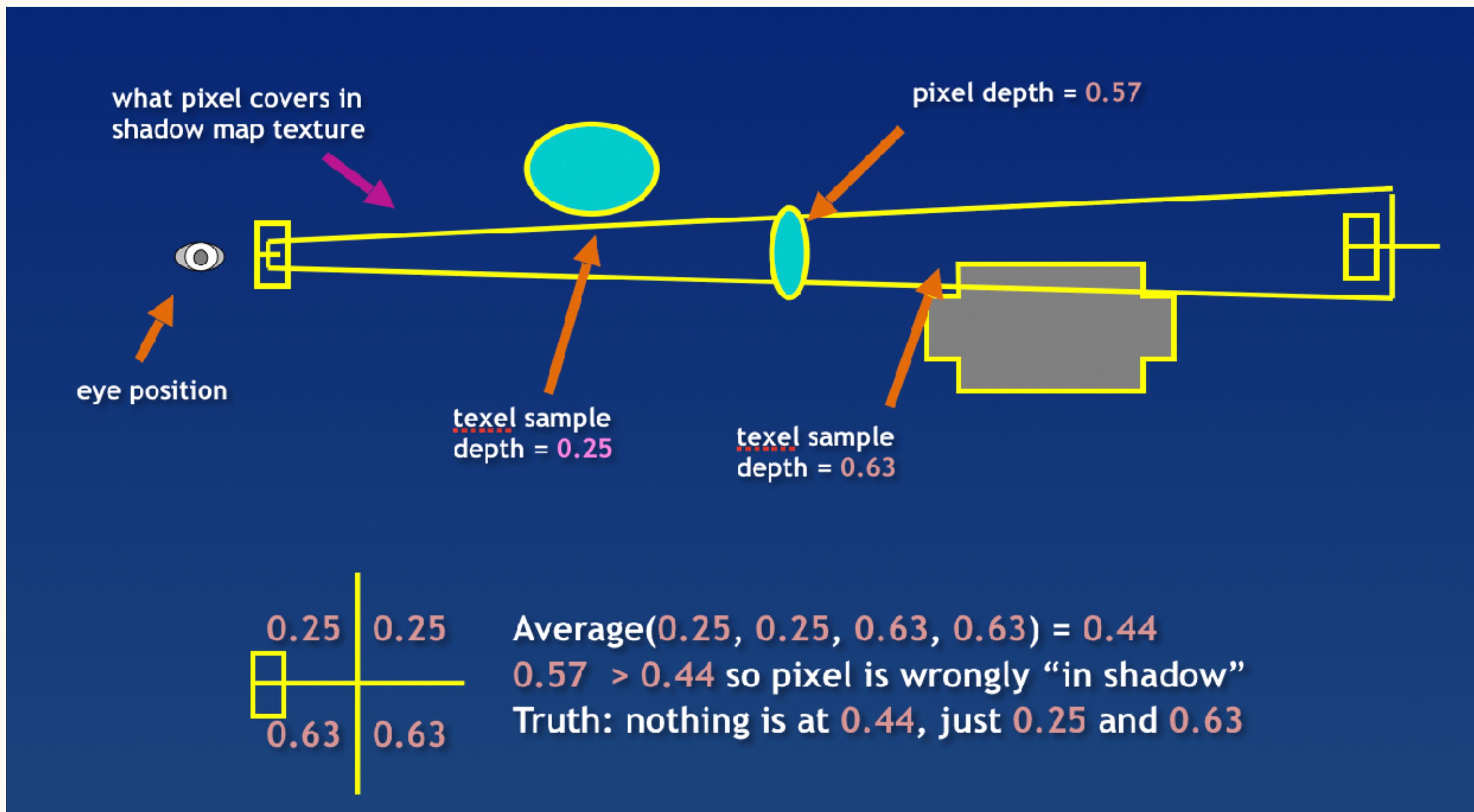
Shadow Map Problems – Aliasing Effect

- Aliasing problem
- Solutions :
 - Using the floating-point texture
 - Increasing Texture resolution
 - PCF
 - Percentage Closer Filter



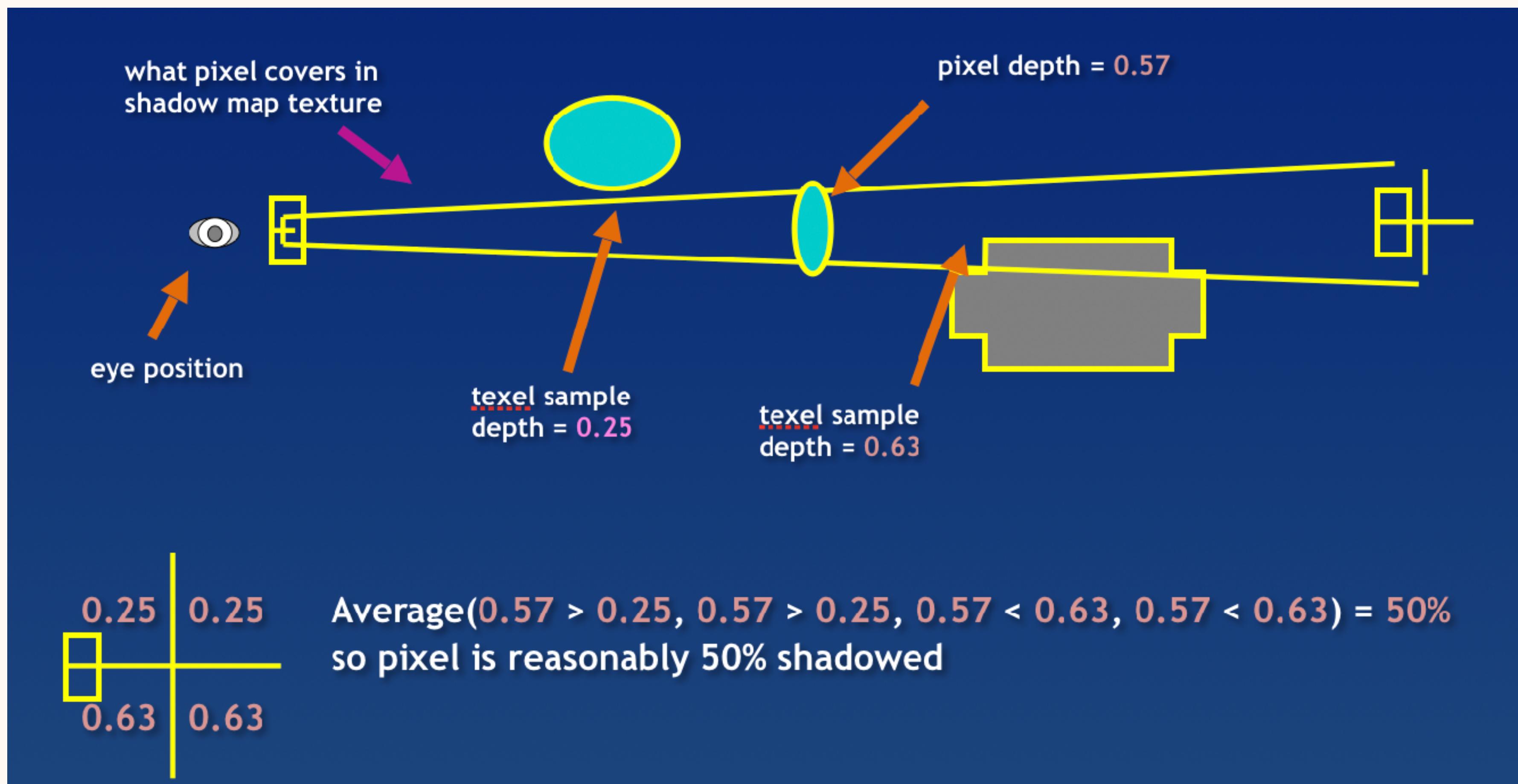
Shadow Map Problems – Aliasing Effect

- Traditional average function not working



Percentage Closer Filtering

- Solution : Average comparison result : Percentage Closer Filter



Percentage Closer Filtering

- Increasing the number of PCF taps will increase the shadow softness



1x1



9x9



17x17

- Using irregular sampling to improve quality but more randomization causes noises



Sampler Comparison State

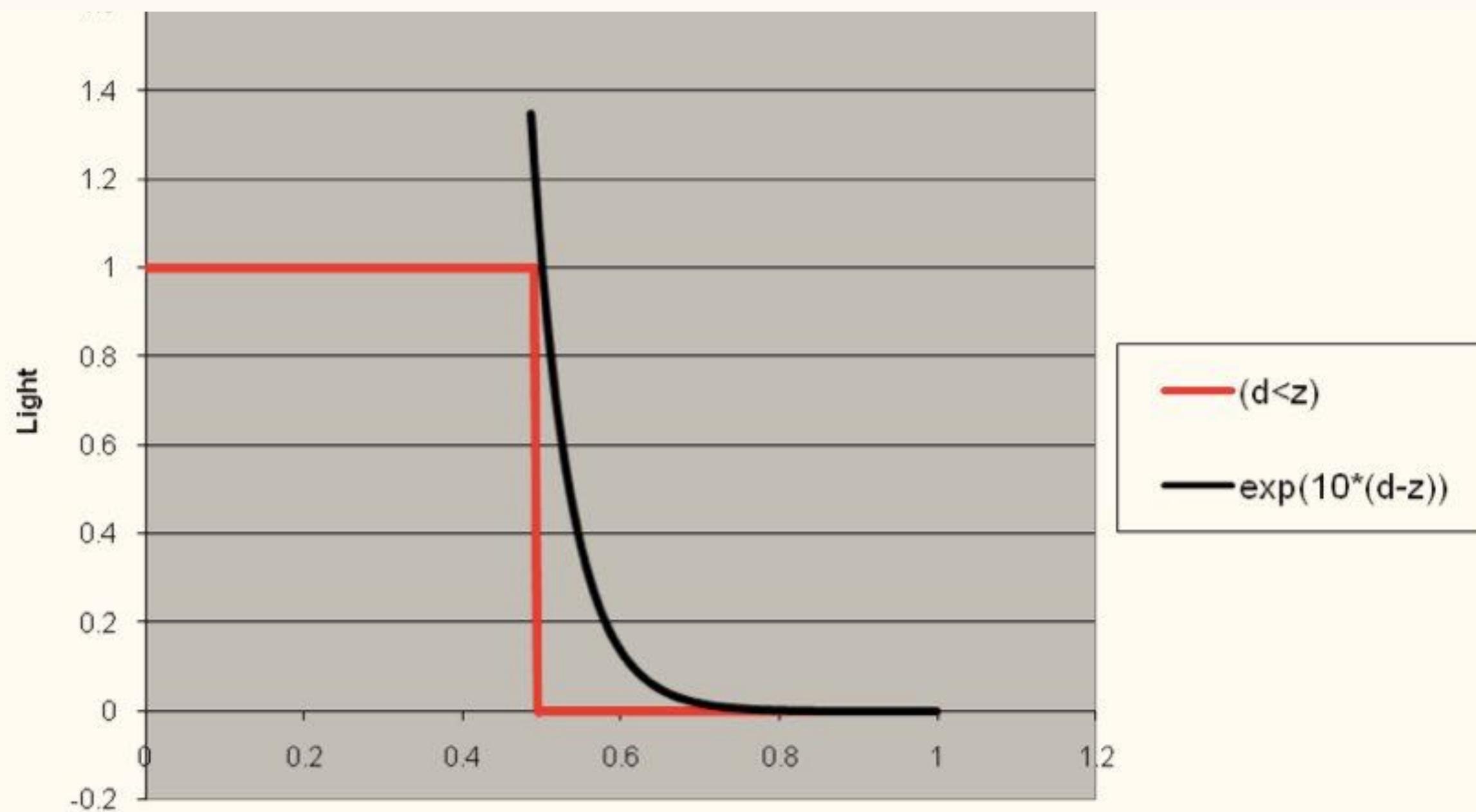
```
SamplerComparisonState ShadowSampler
{
    // sampler state
    Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
    AddressU = MIRROR;
    AddressV = MIRROR;

    // 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]
- Approximate step function ($z-d > 0$) with $\exp^{(k*(z-d))} = \exp^{(k*z)} * \exp^{(-k*d)}$
 - k determines how good the approximation
 - The greater the value, the better the approximation
 - Large values cause precision loss but cause sharper shadow boundaries

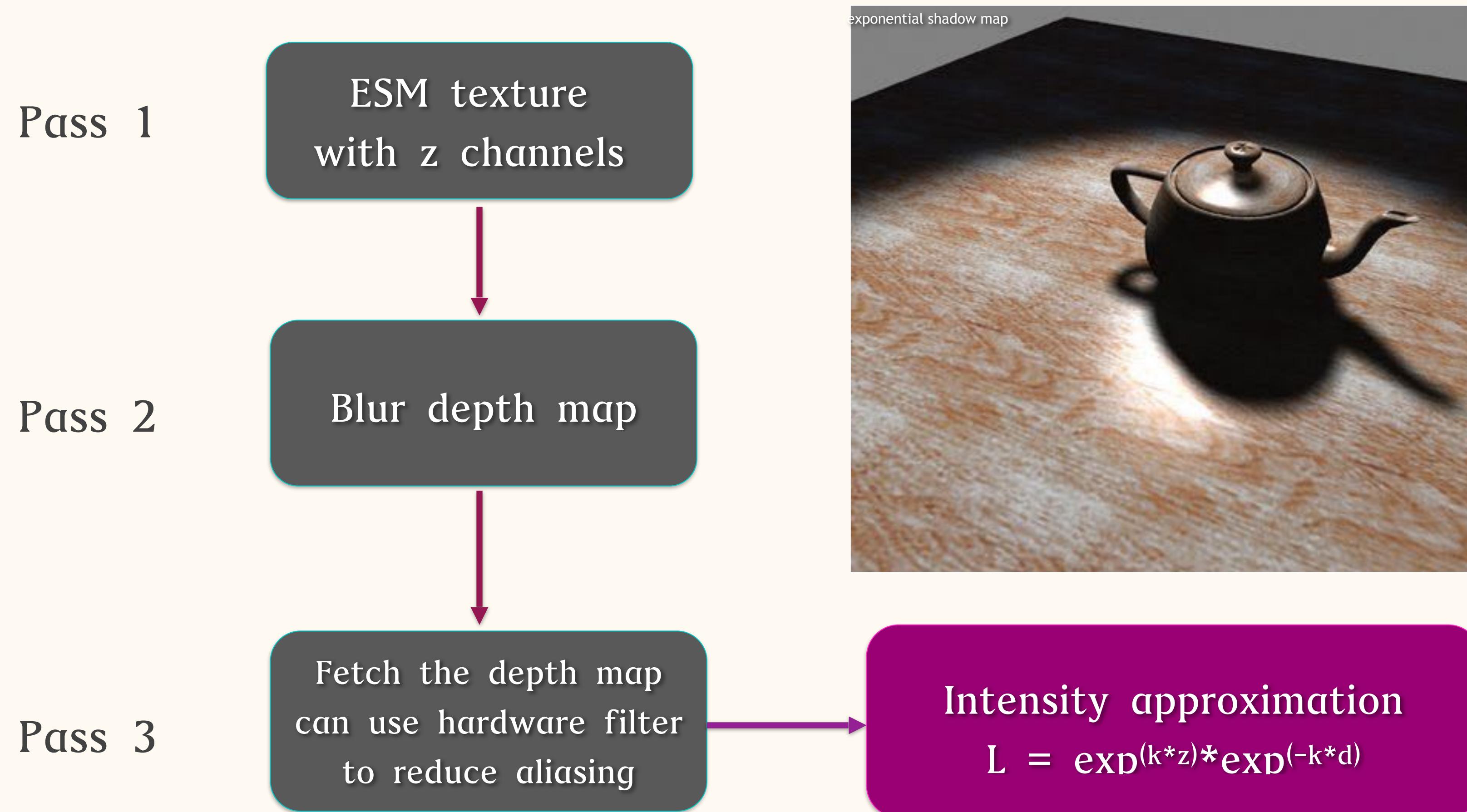


$k = 10$



$k = 30$

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;
```

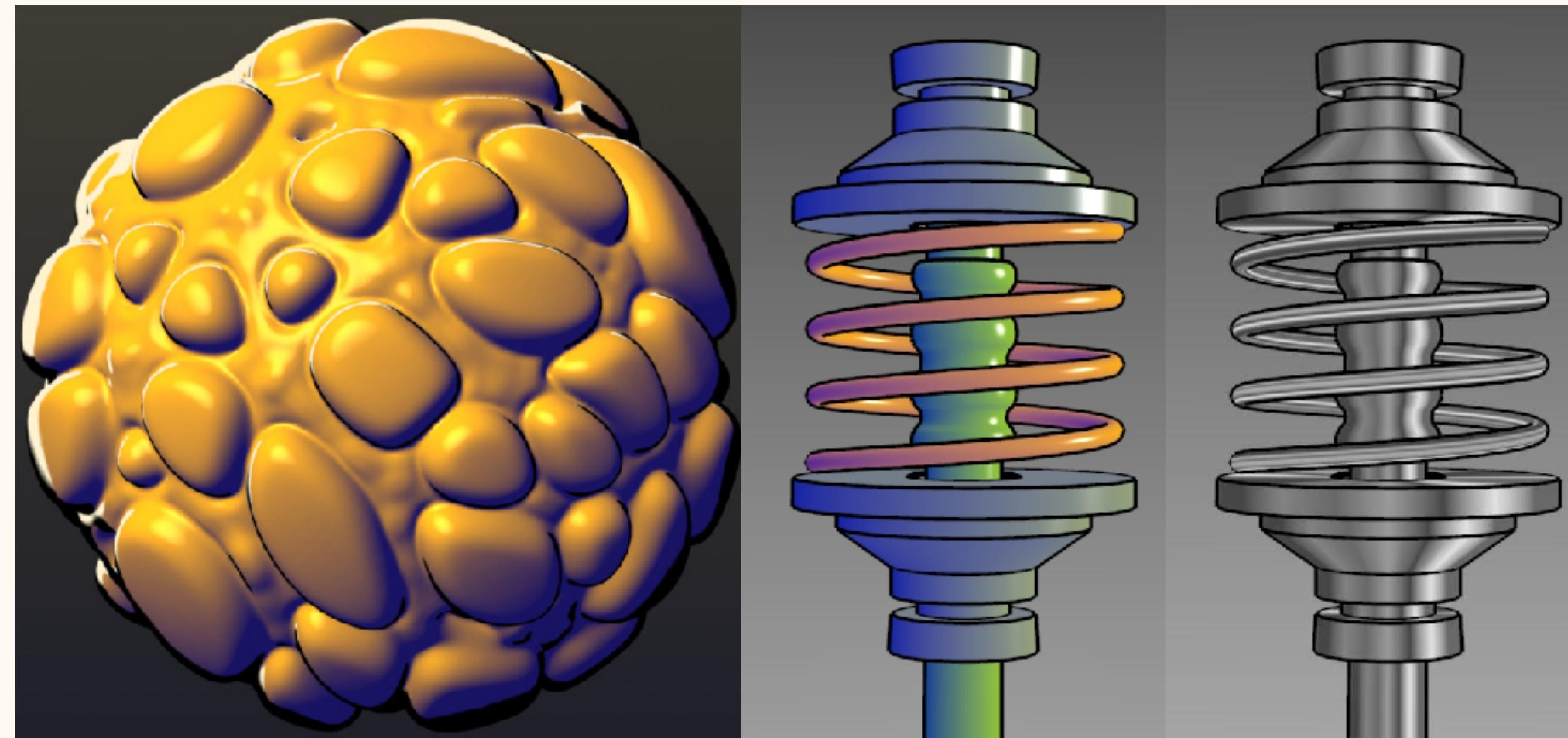
Ambient and Shadow

- Lighting is an “add” operation
 - Fake solutions : Lightmap / Ambient Occlusion map
- Ambient issue
 - Ambient = lighting from environment
 - Ambient = no lighting area from direct light sources
 - Occluded by geometry itself
- Shadow issue
 - Shadow = no lighting area from direct light source
 - Occluded by the other geometry
- Correct solution
 - Global illumination renderers

Non-photorealistic Rendering

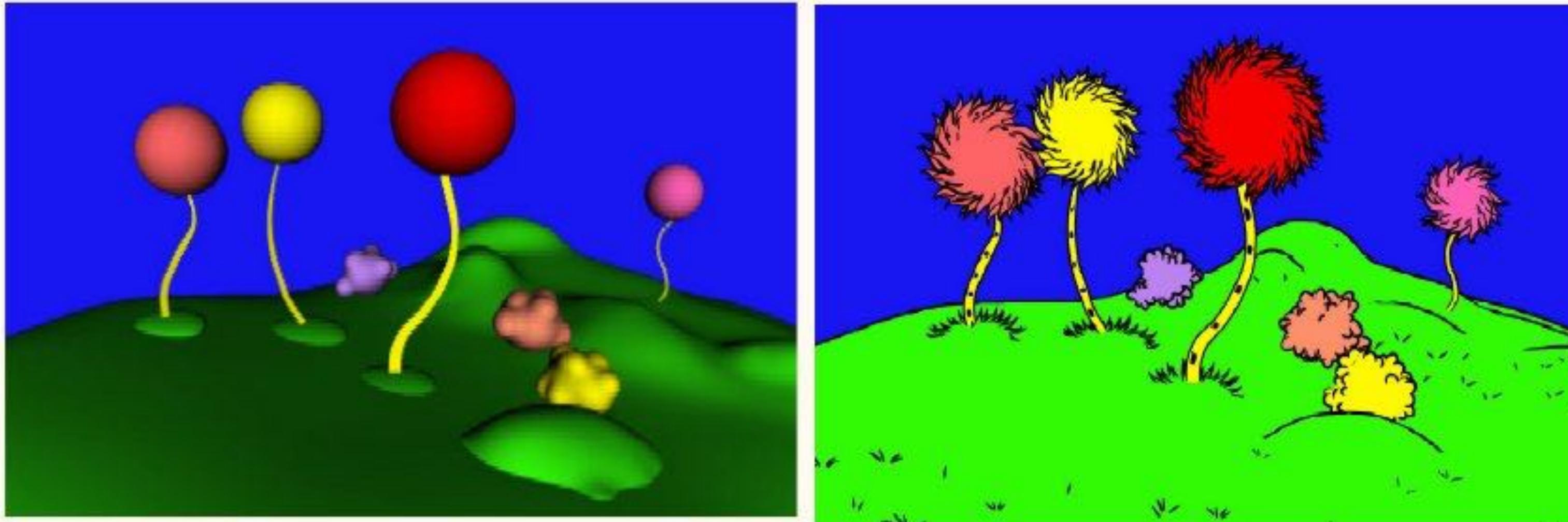
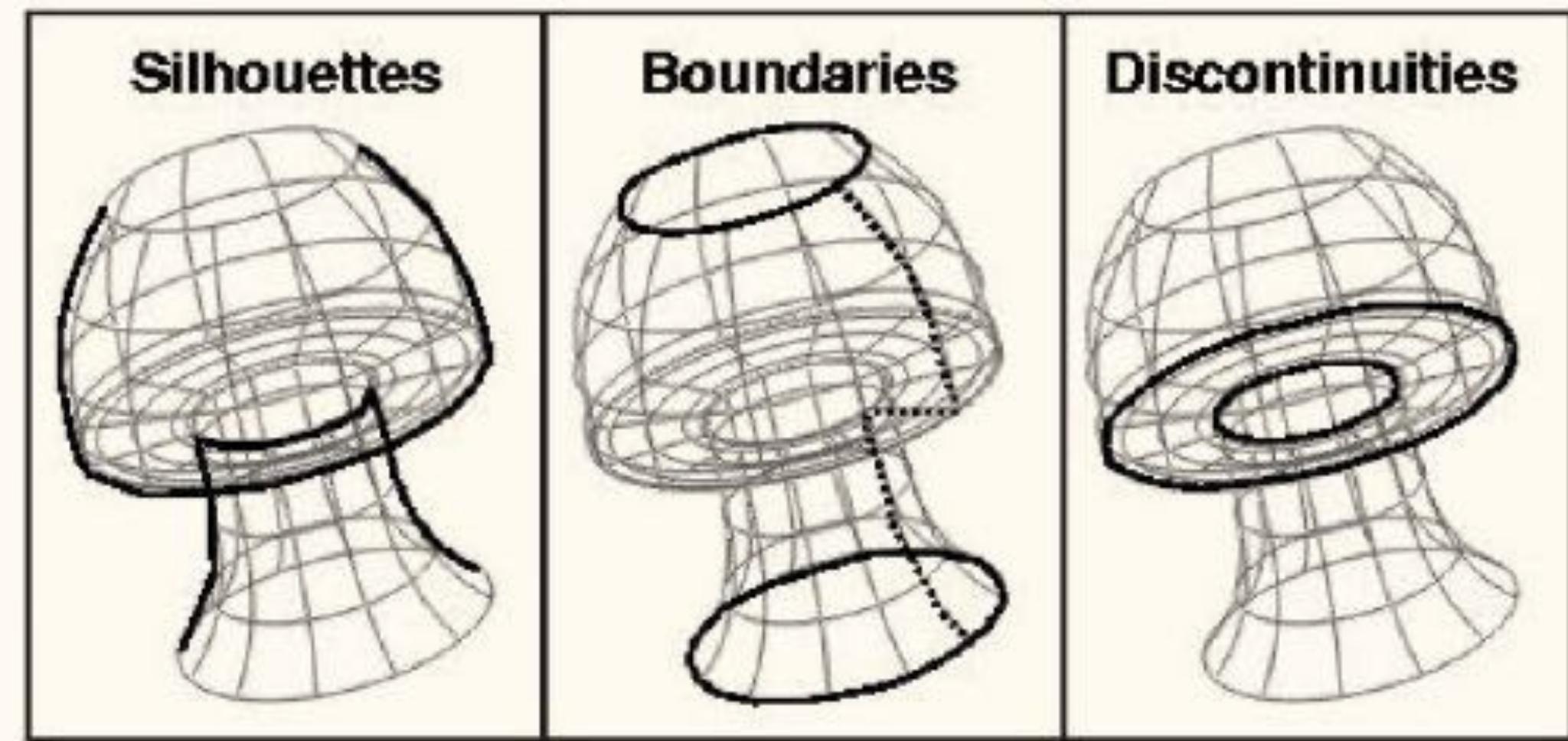
Non-photorealistic Rendering

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

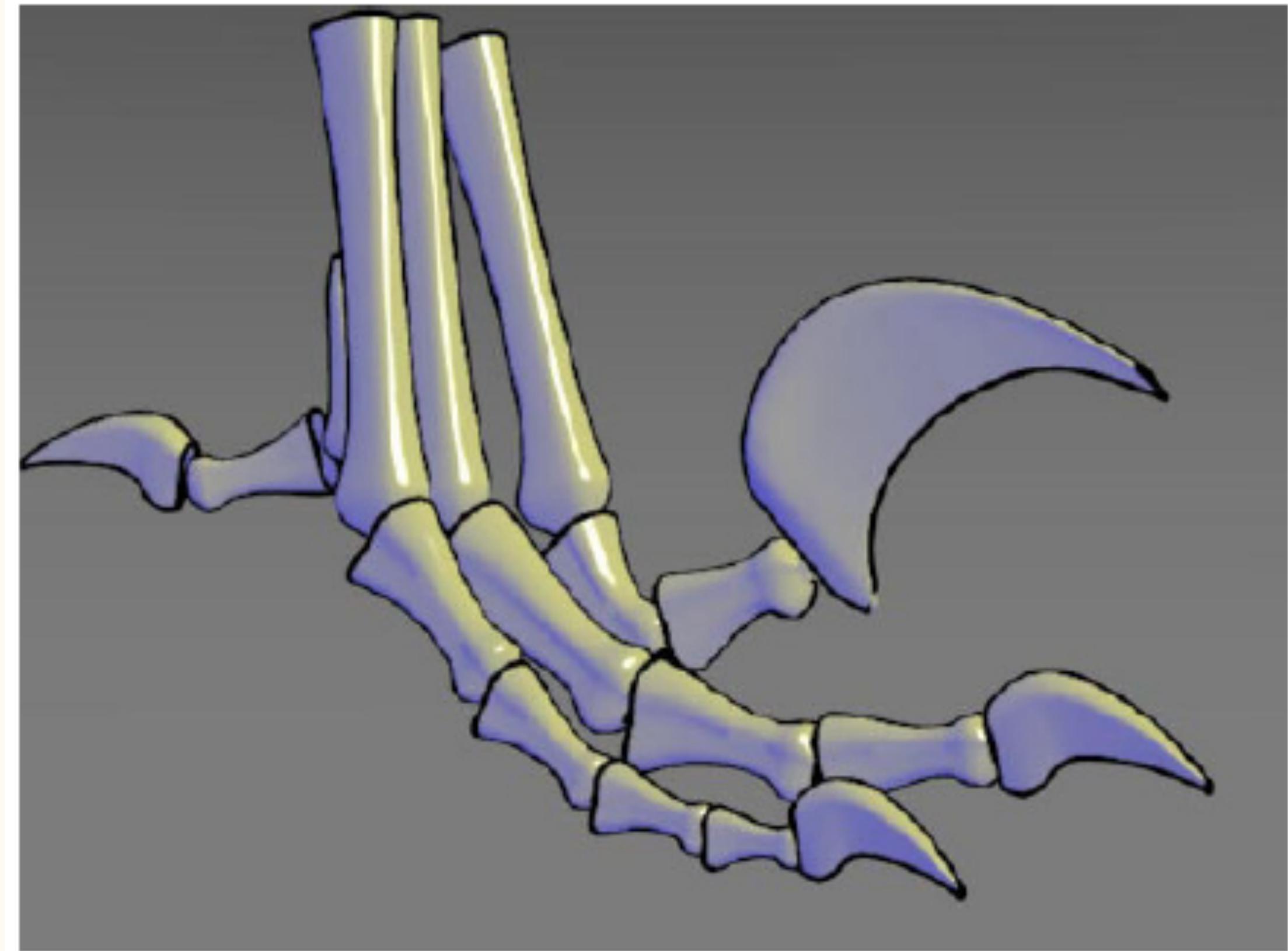
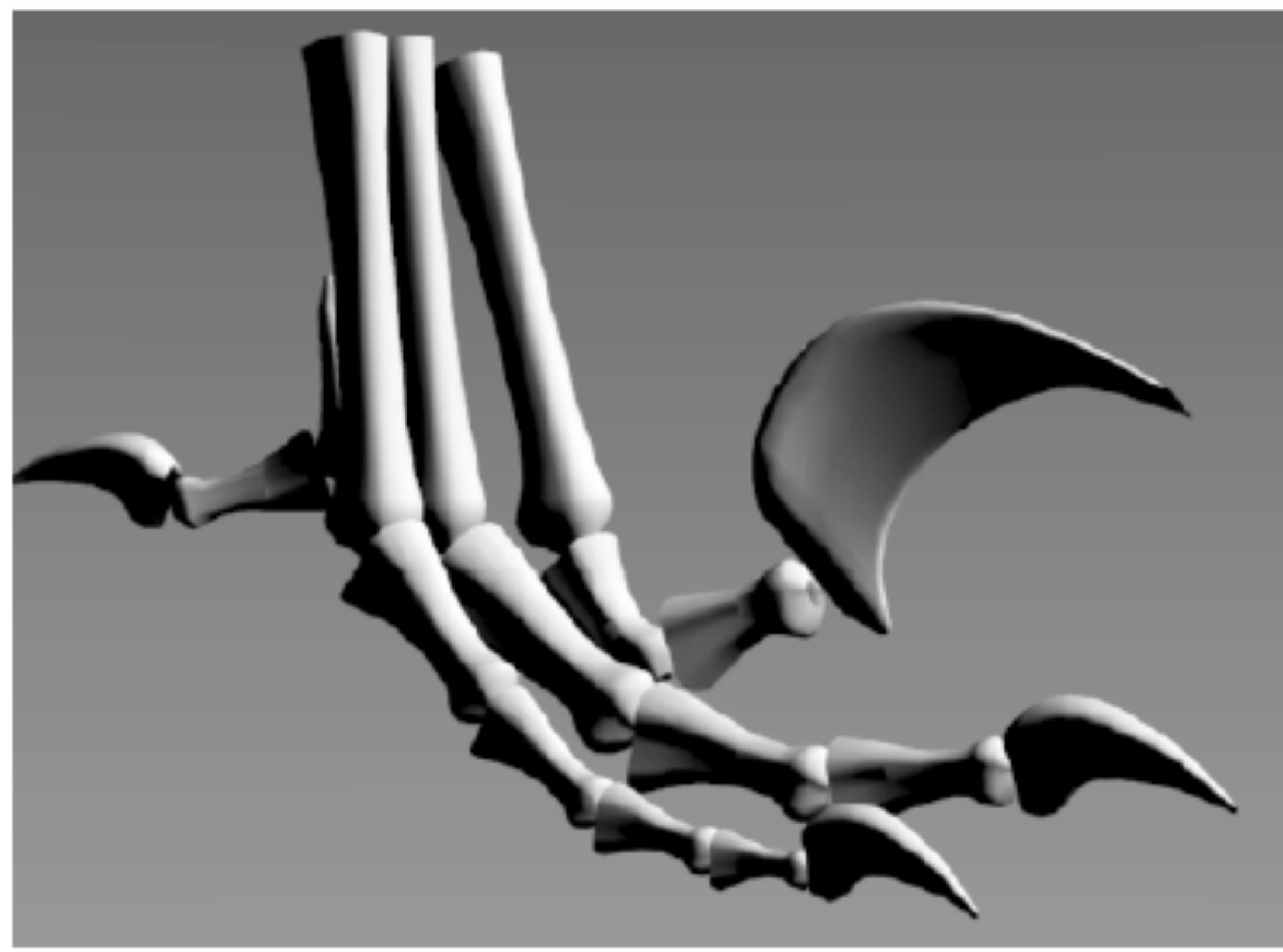


Shape Abstraction by Lines

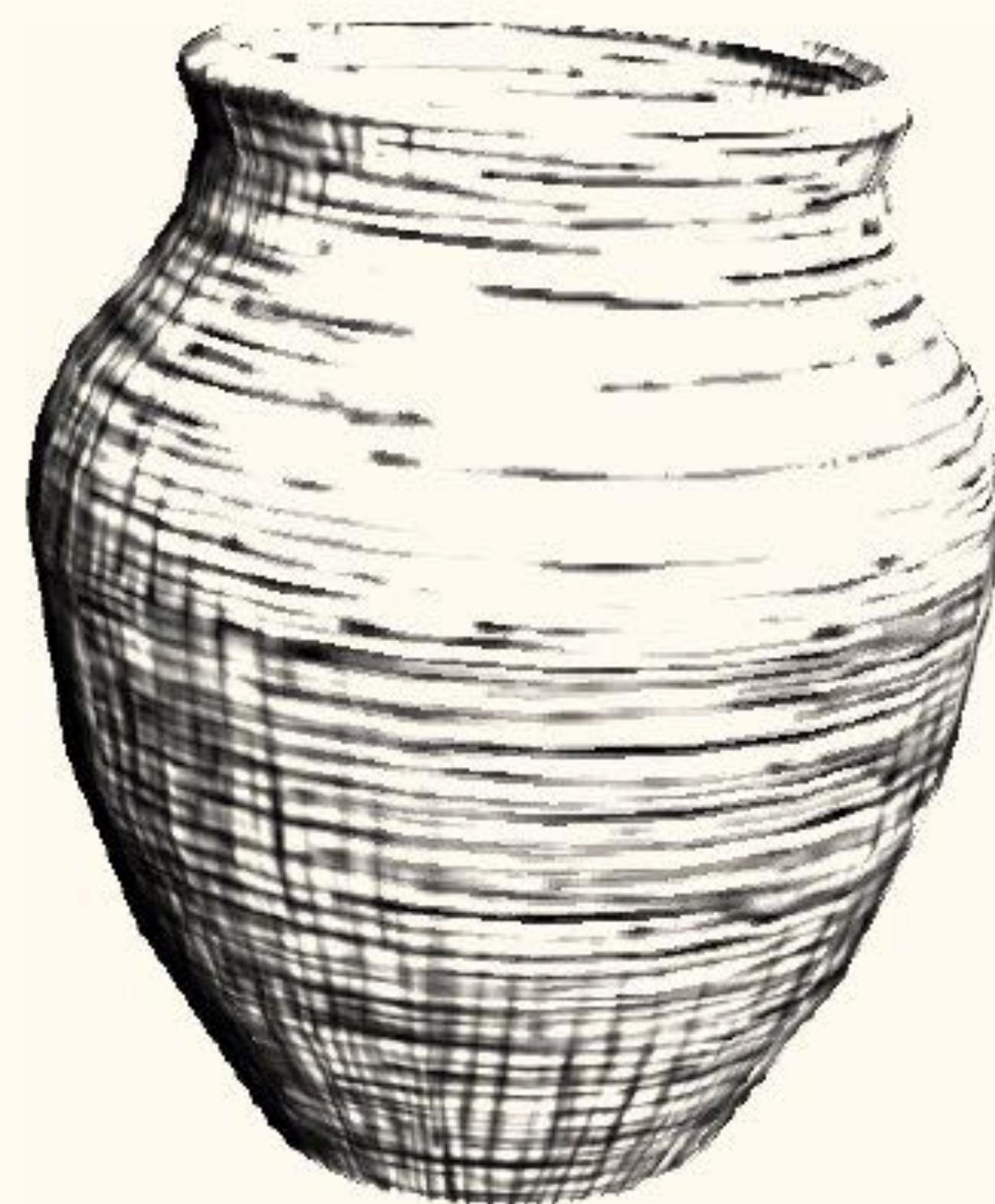
- Boundary lines
- Silhouette lines
- Creases
- Material edges
- Various line styles can be used
- Graftals



Shape Abstraction by Shading

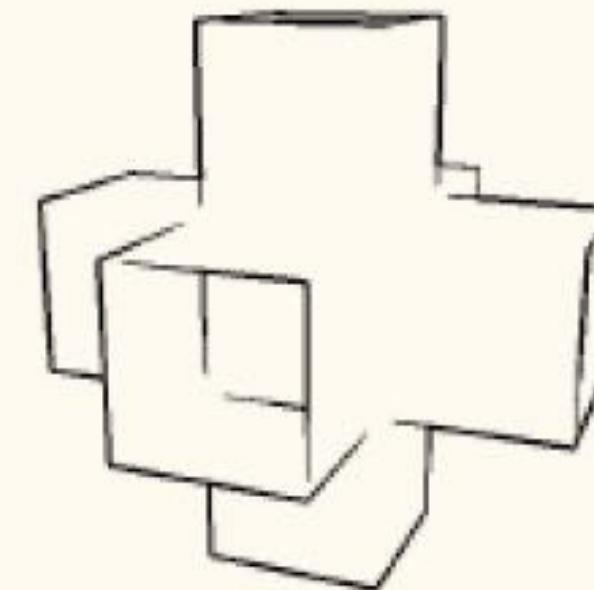
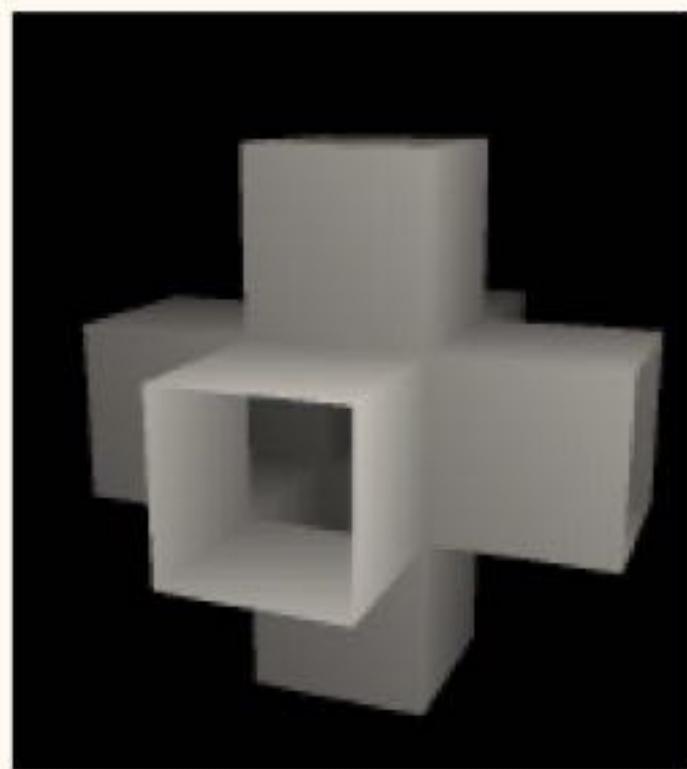


Shape Abstraction by Textures

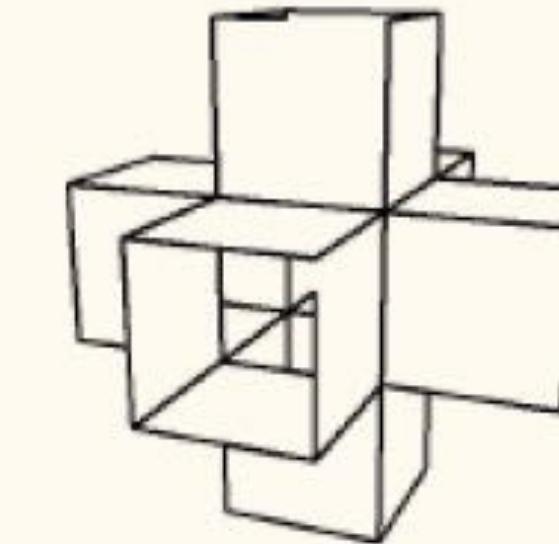
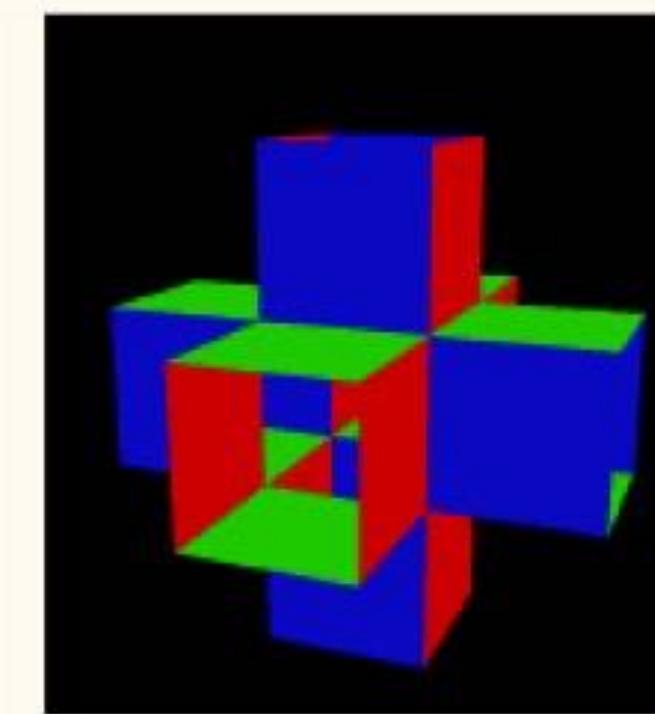
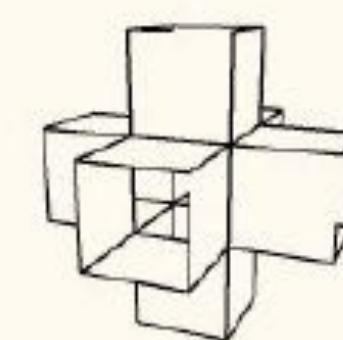
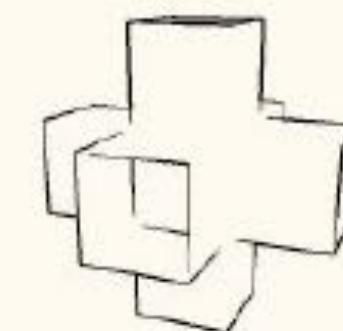
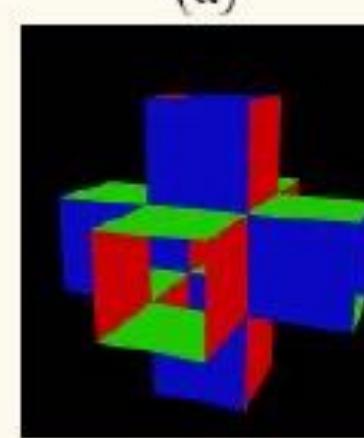
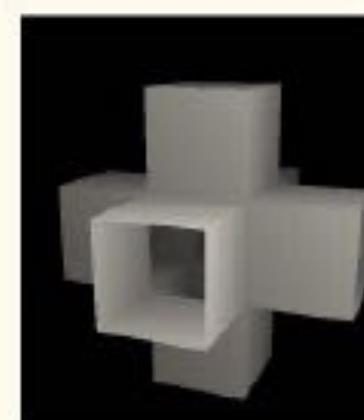


Feature Line by Edge Detection

- Analyze the depth buffer



- Combine together :



Edge Detector

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

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

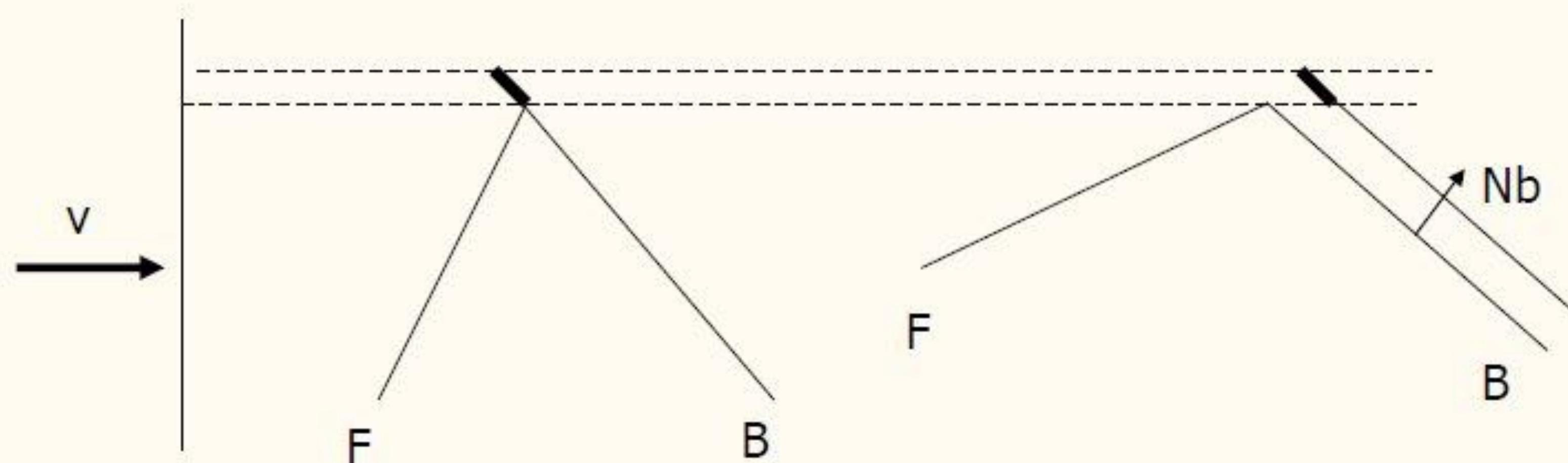
$$Ix(x,y) = I(x,y) \otimes Sx; \quad Iy(x,y) = I(x,y) \otimes Sy$$

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

Get edge by thresholding IM

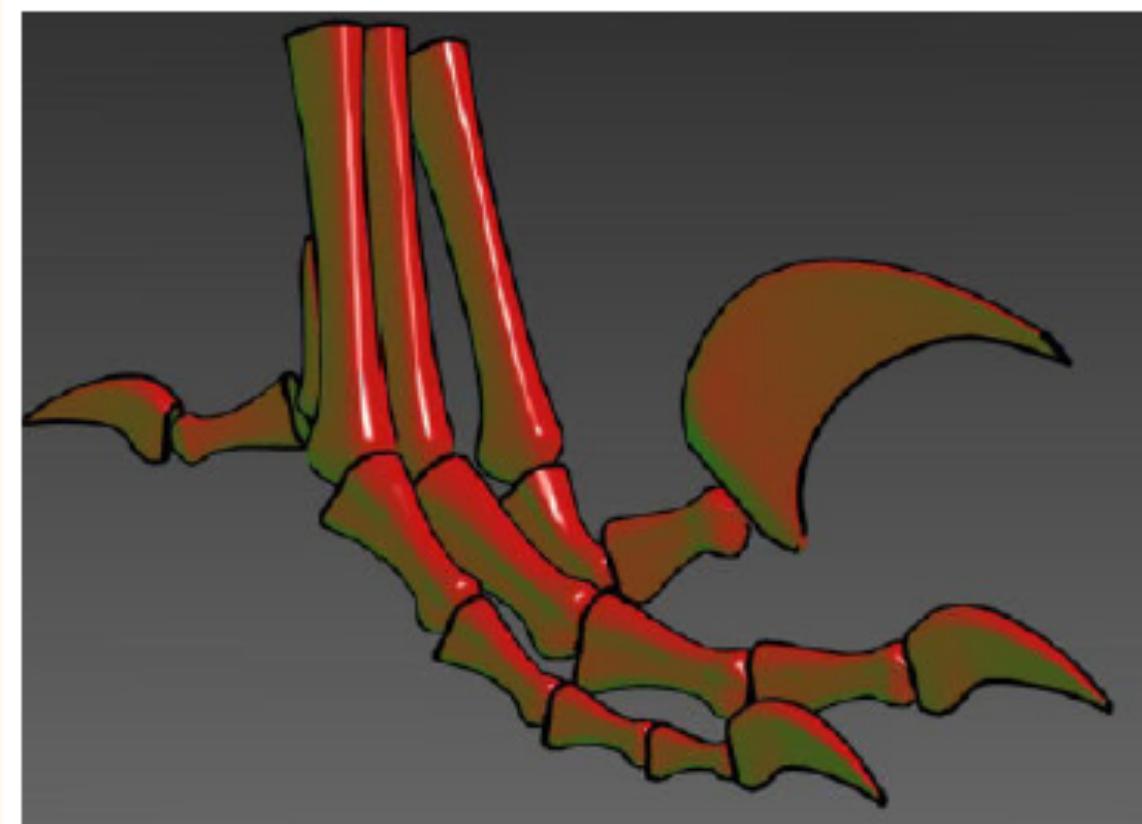
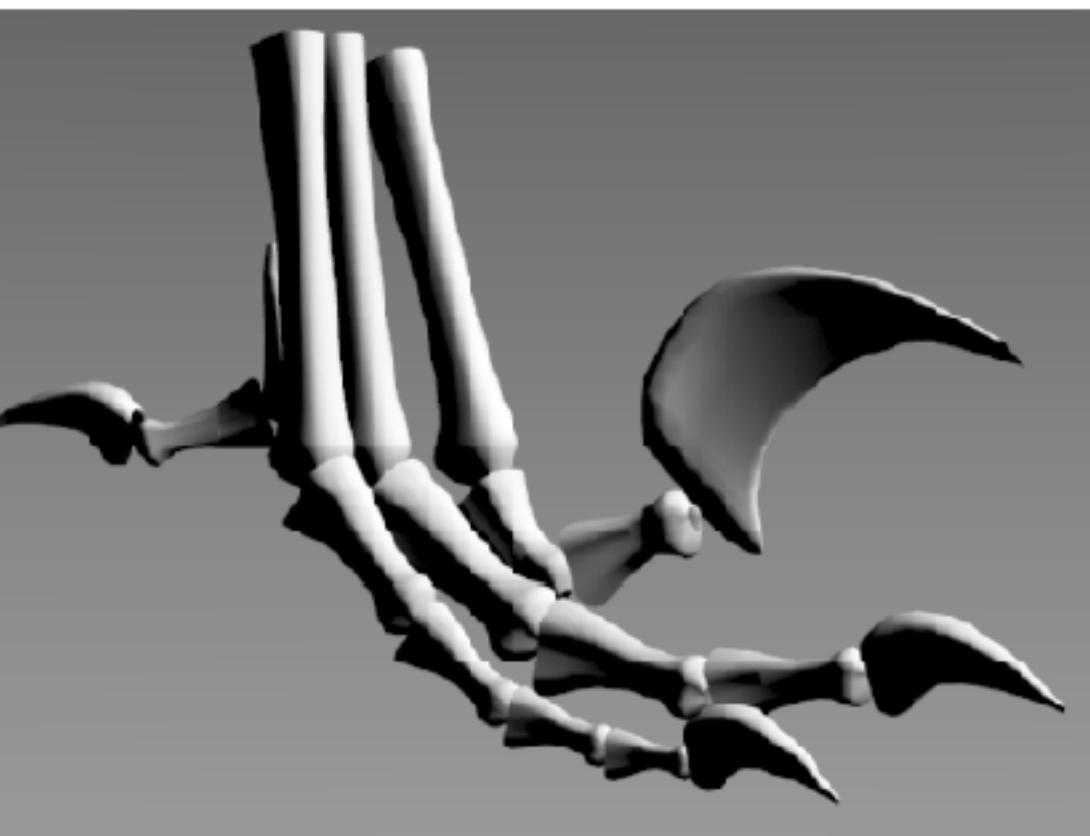
Feature Lines by Geometry

- 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

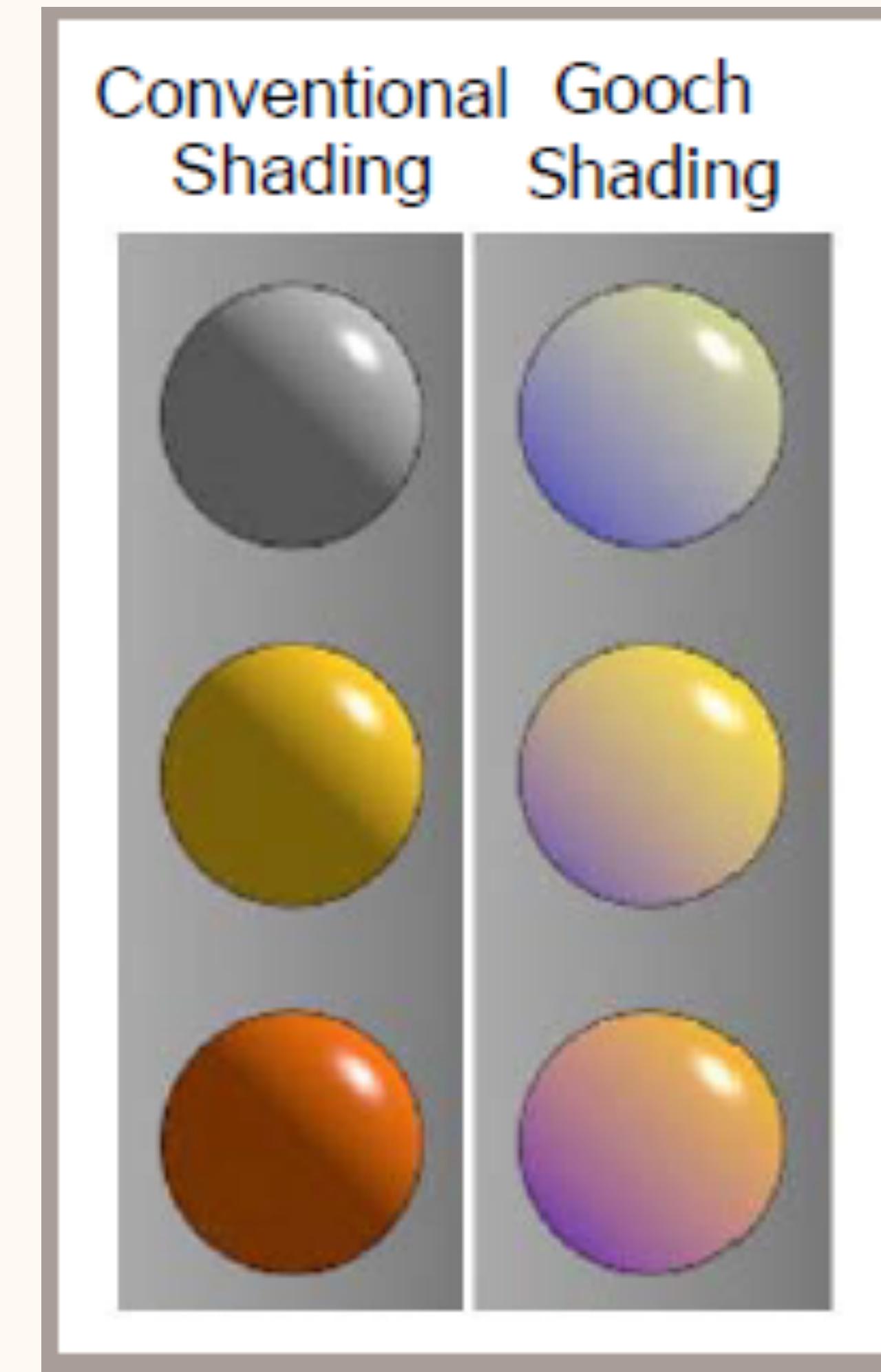
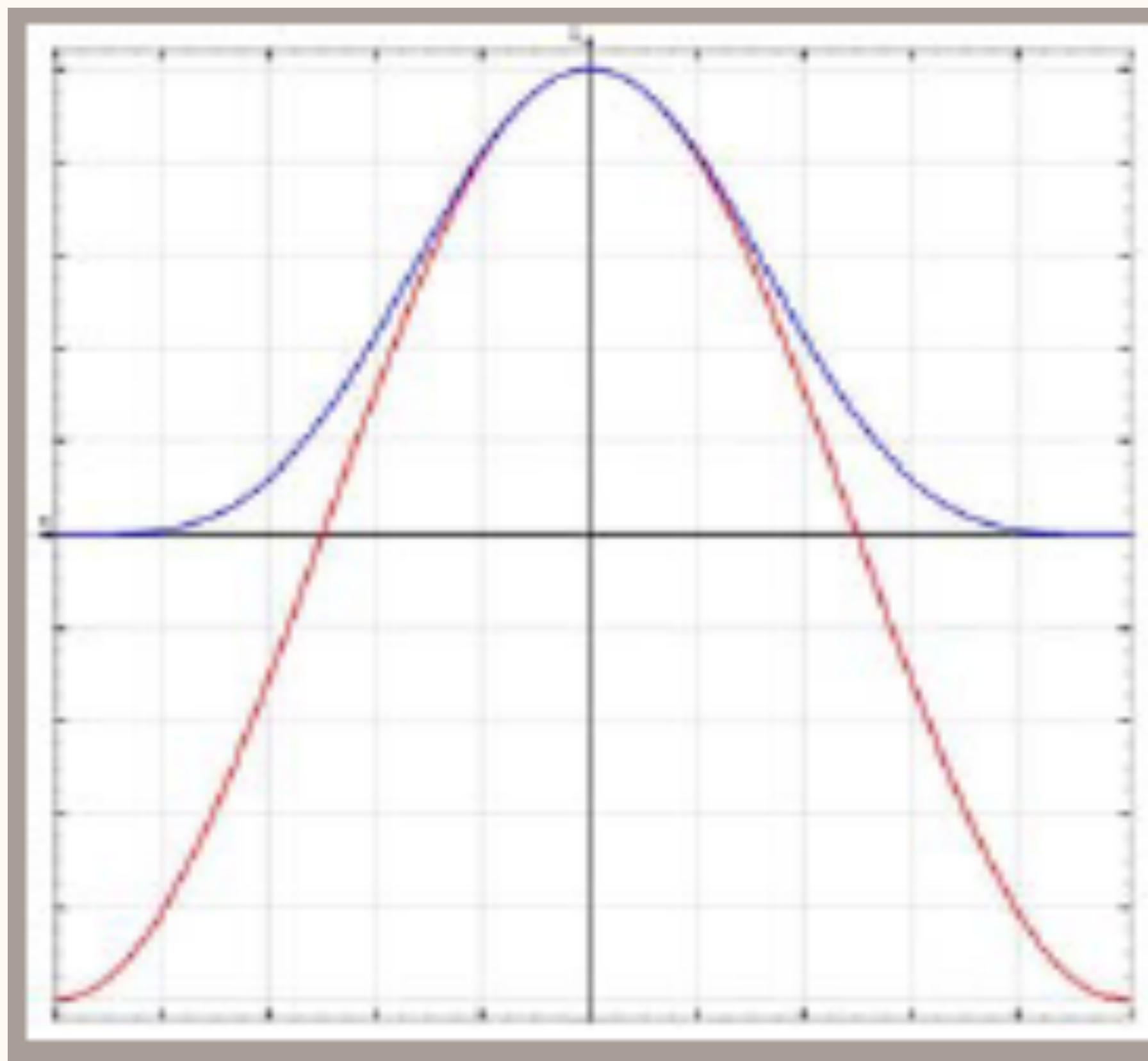


warm

cool

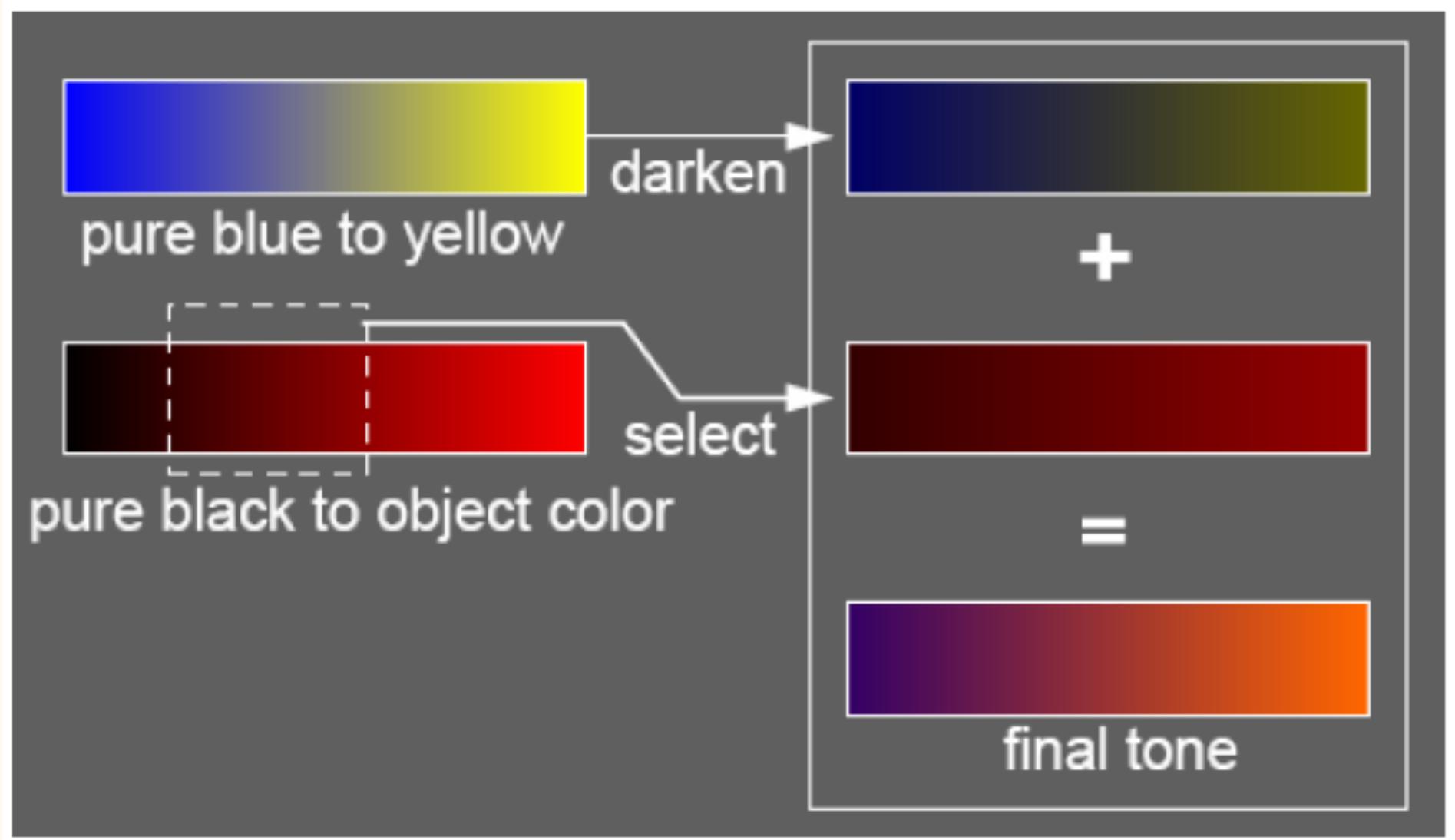
Tone Shading – Half Lambert

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



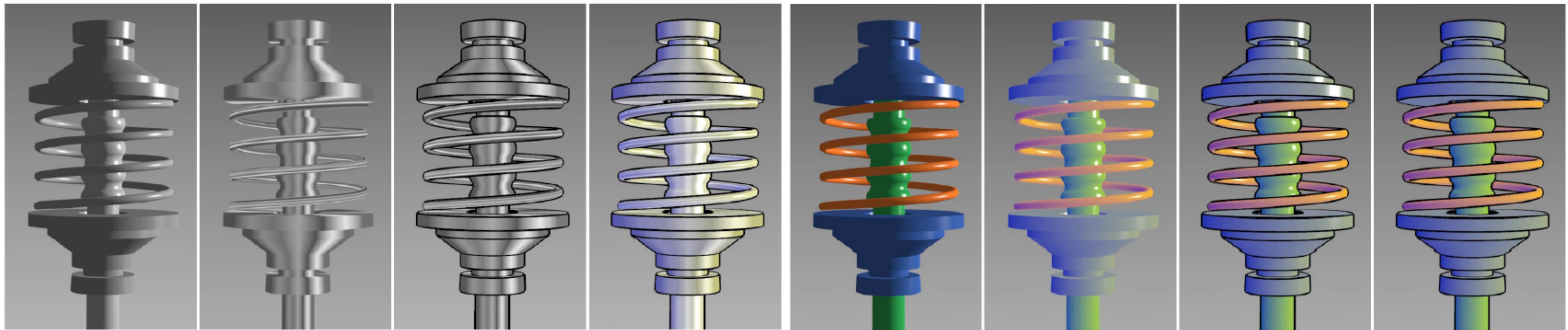
Tone Shading - Blend tone and Undertone

- 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

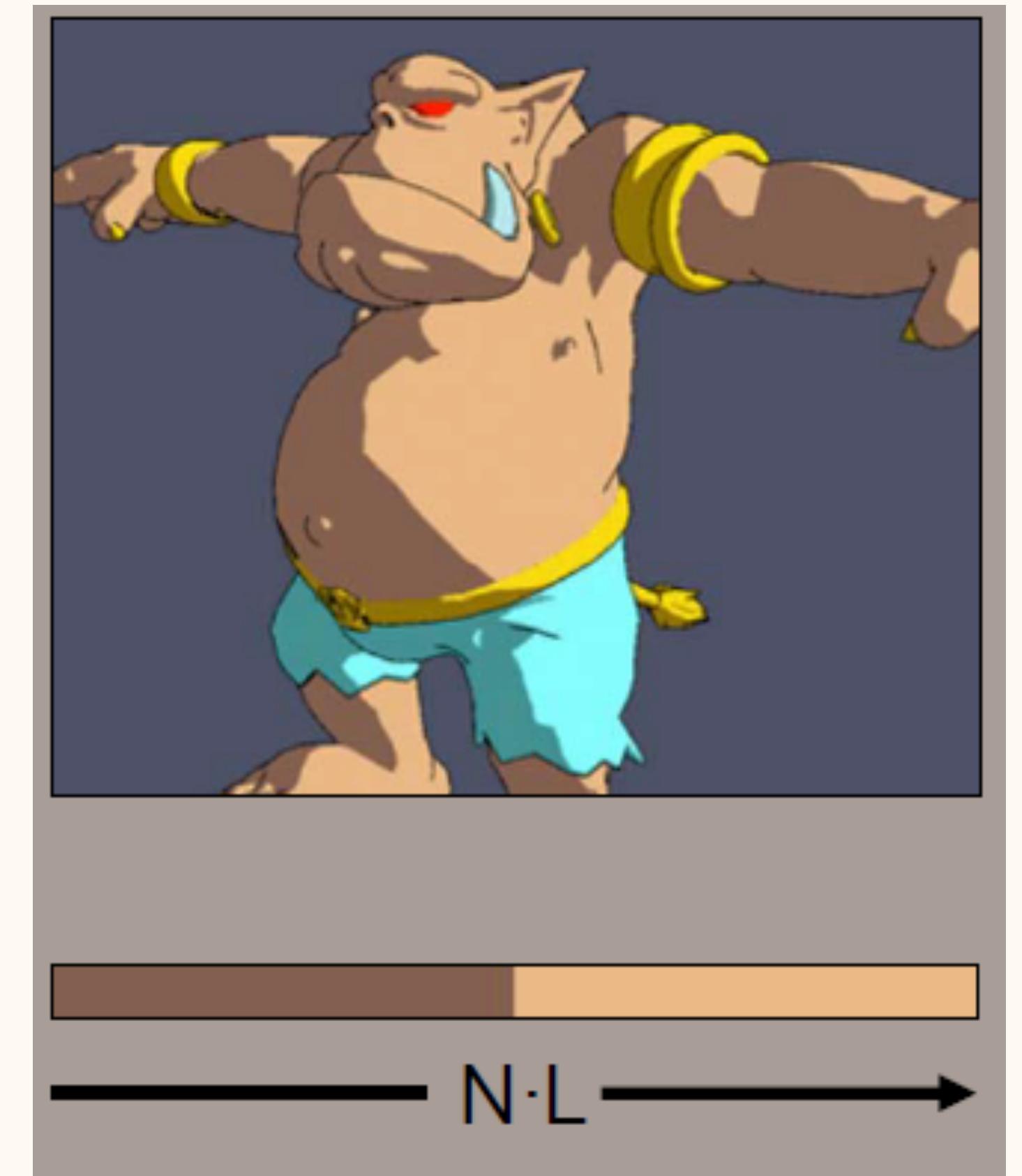
Tone Shading - Gooch Shading



Gooch 98

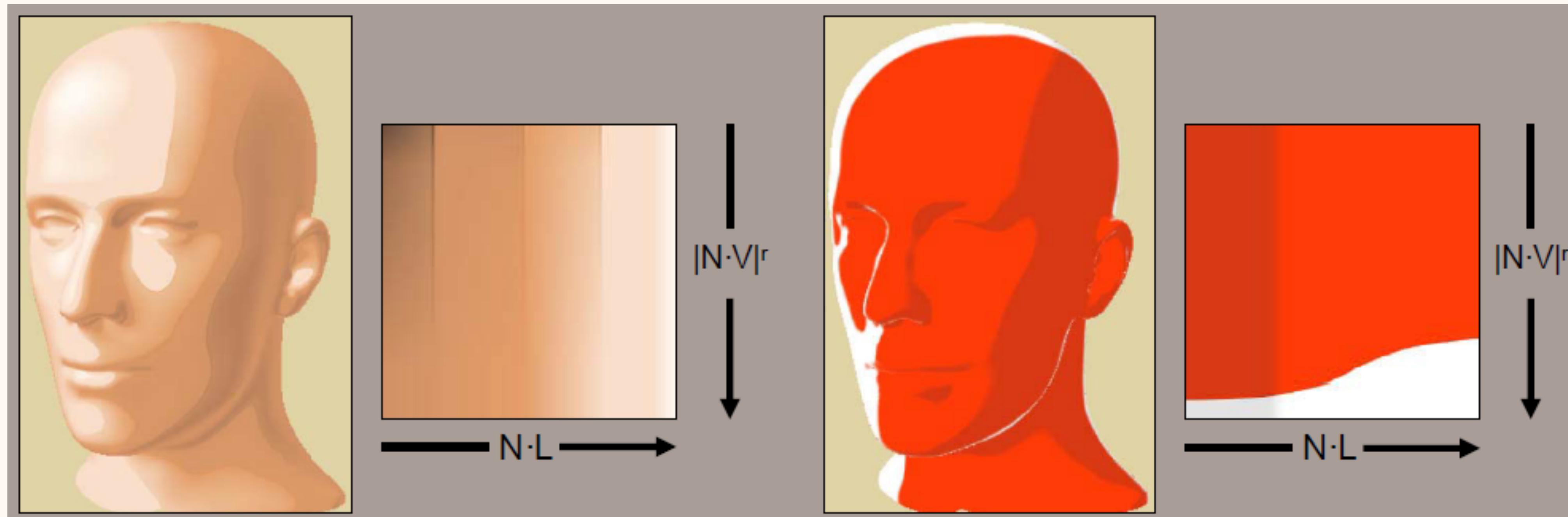
Shading As the Lookup Table

- 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

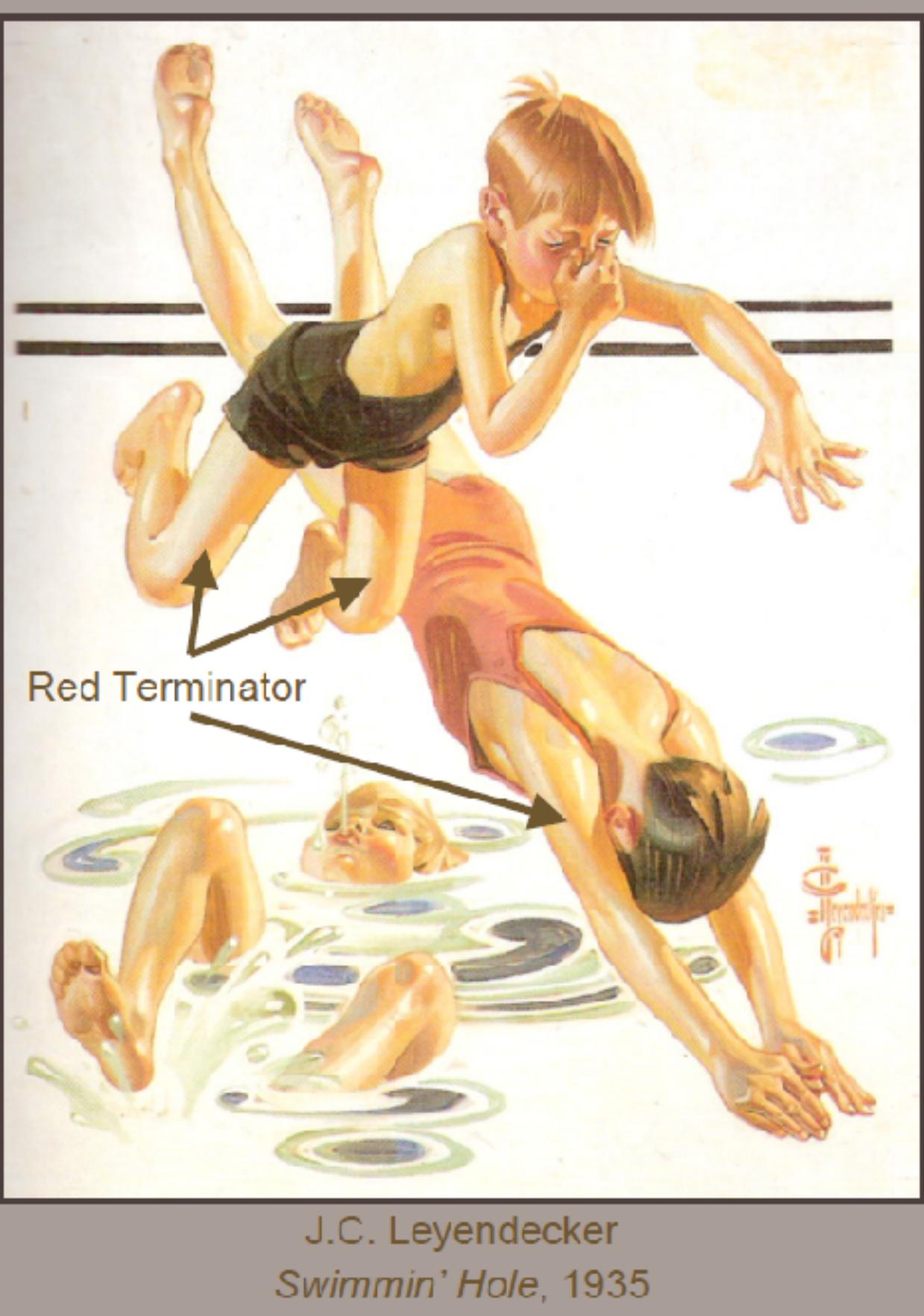
- Barla, 2006
 - Extended the same idea to 2D texture lookup
 - Fresnel-like creates a hard “virtual backlight”
 - A rim-lighting effect



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

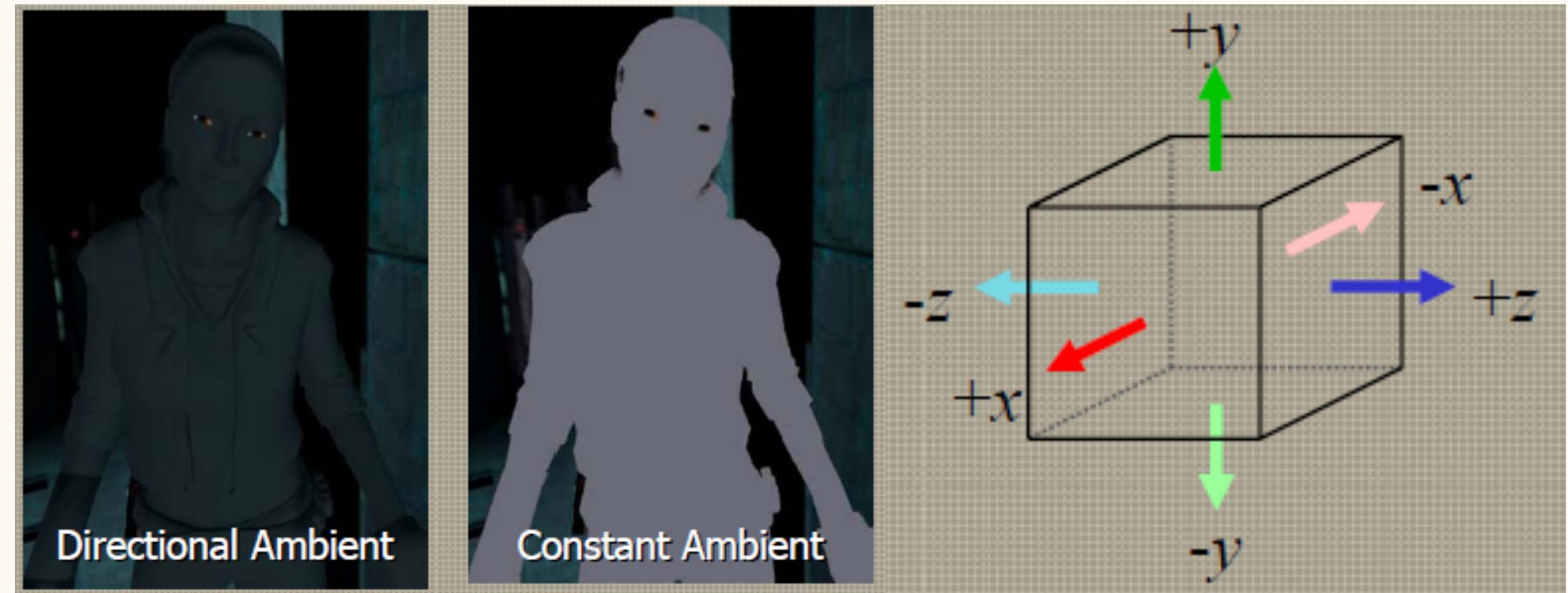
$$k_d \left[a(\hat{n}) + \sum_{i=1}^L c_i w\left(\left(\frac{1}{2} (\hat{n} \cdot \hat{l}_i) + \frac{1}{2} \right) \right) \right]$$

- Spatially-varying directional ambient

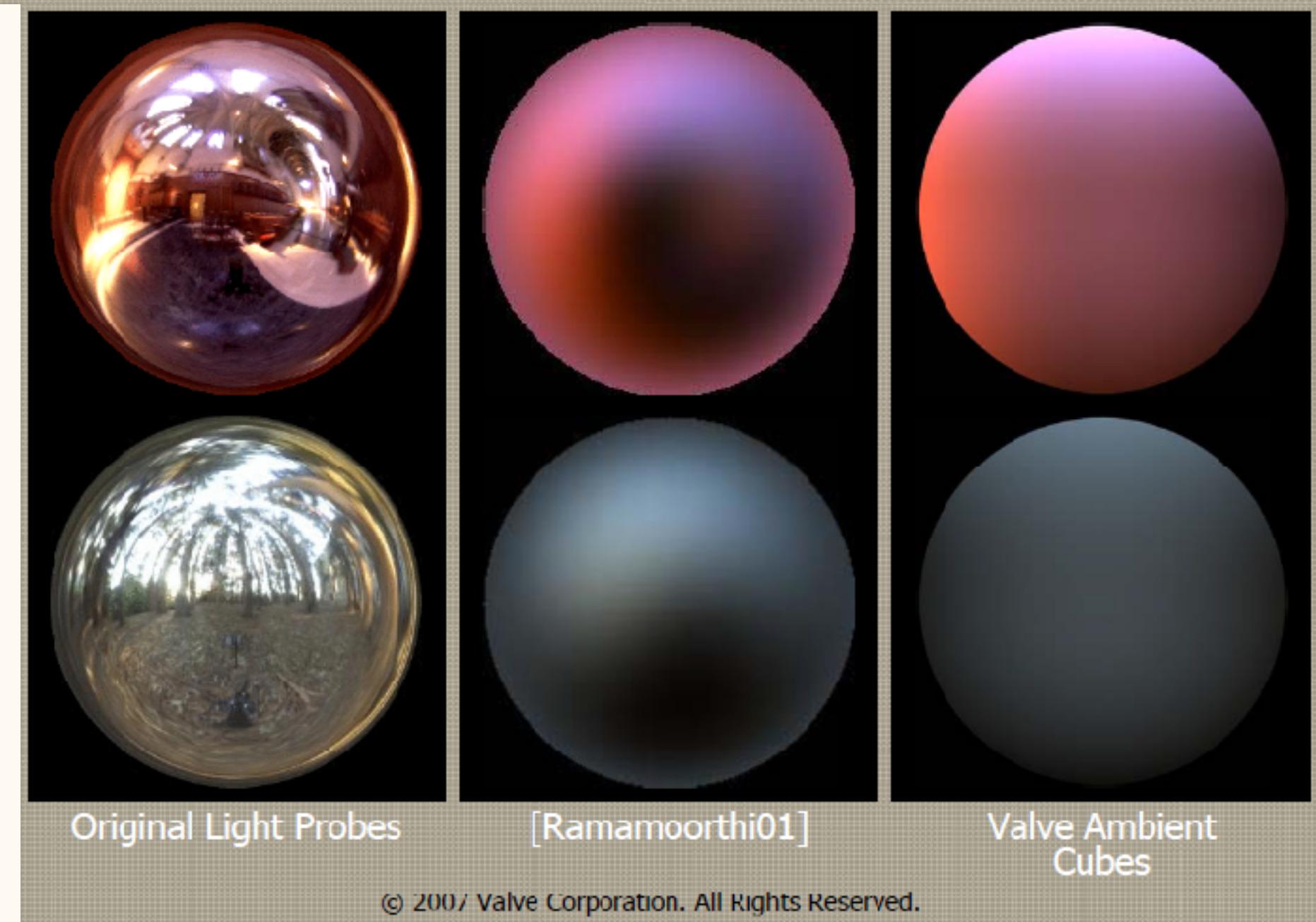


Spatially-varying Directional Ambient

- 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

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

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



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

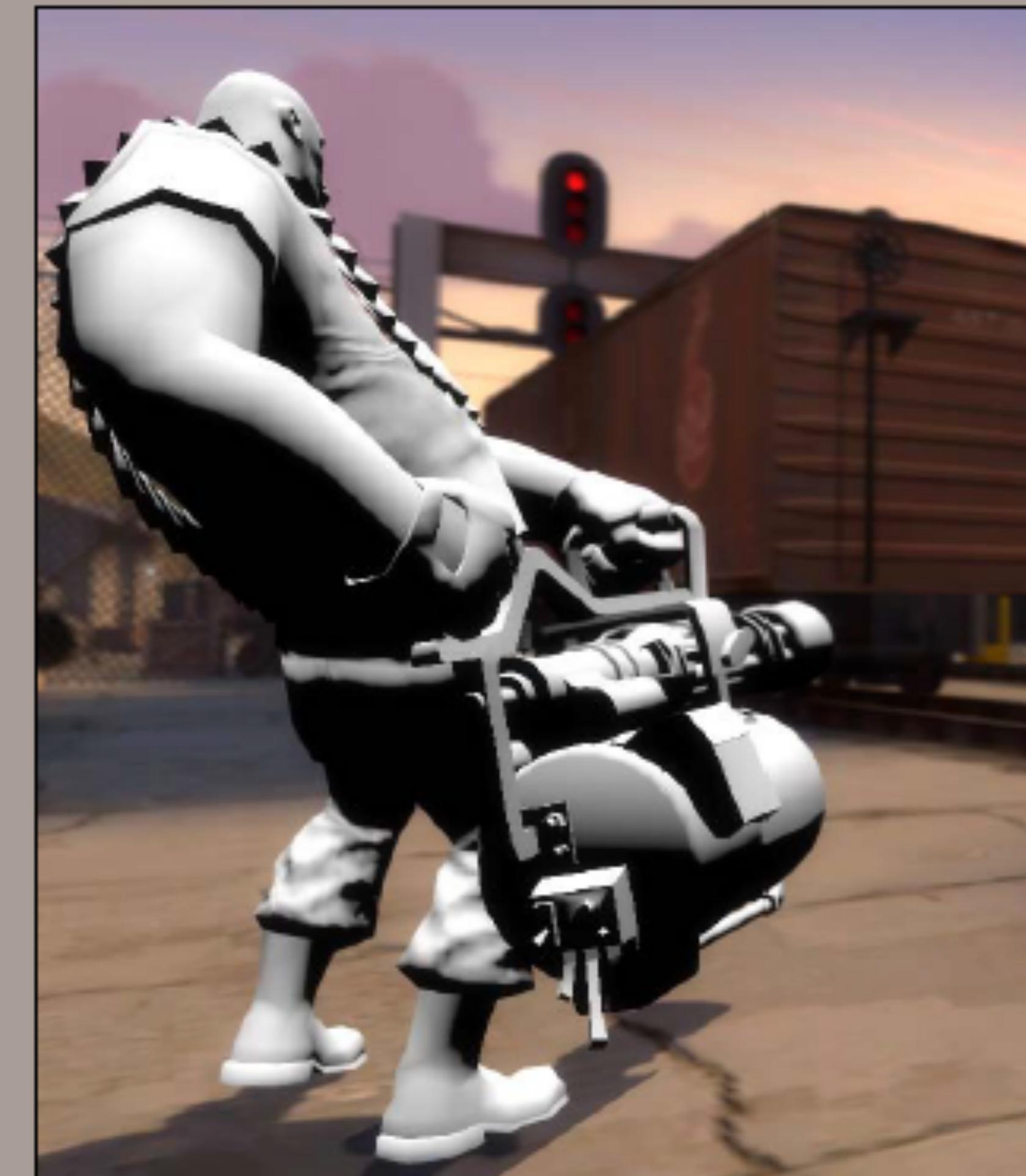
$$F = \eta + (1 - \eta) (1 - N \cdot V)^5$$



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)



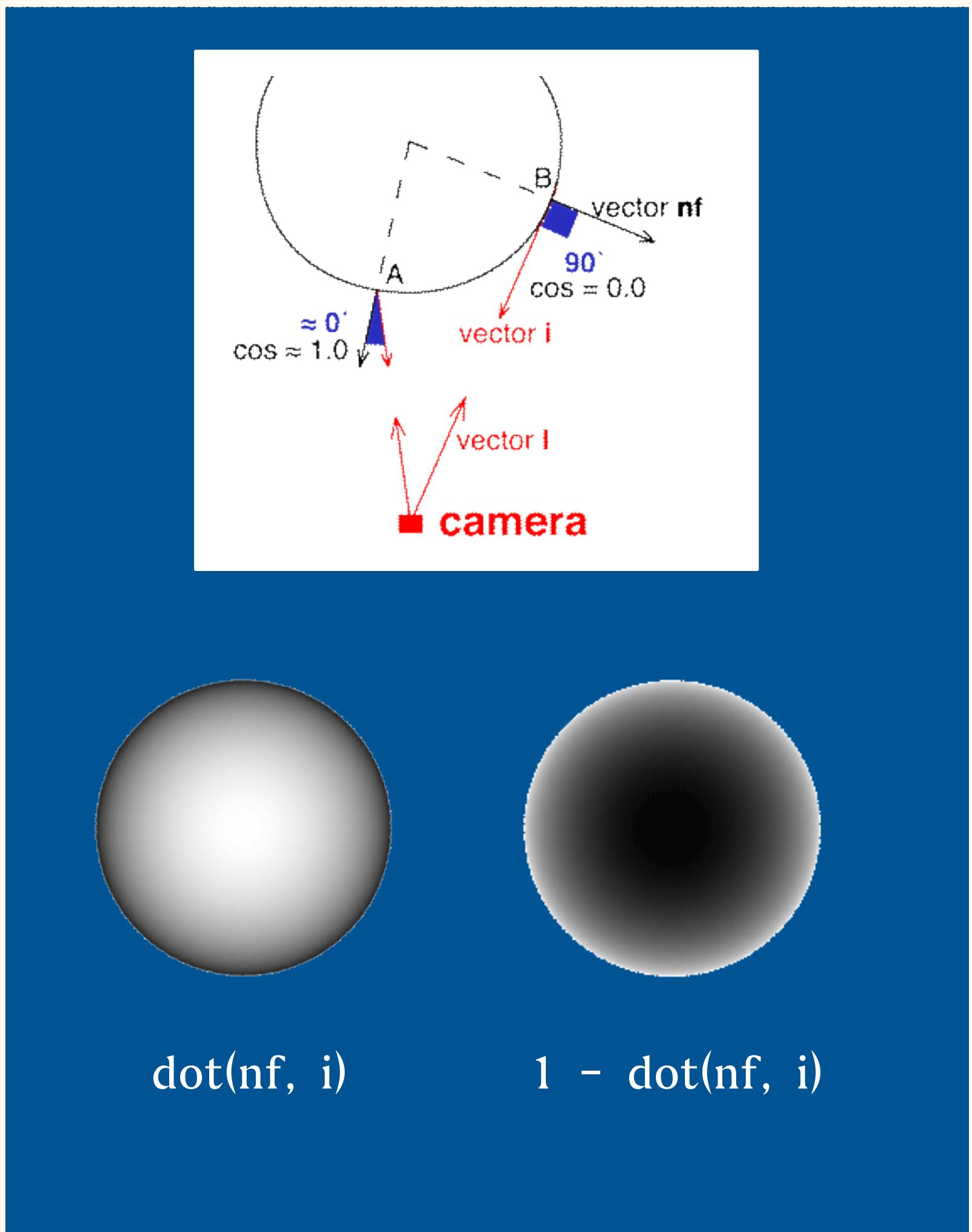
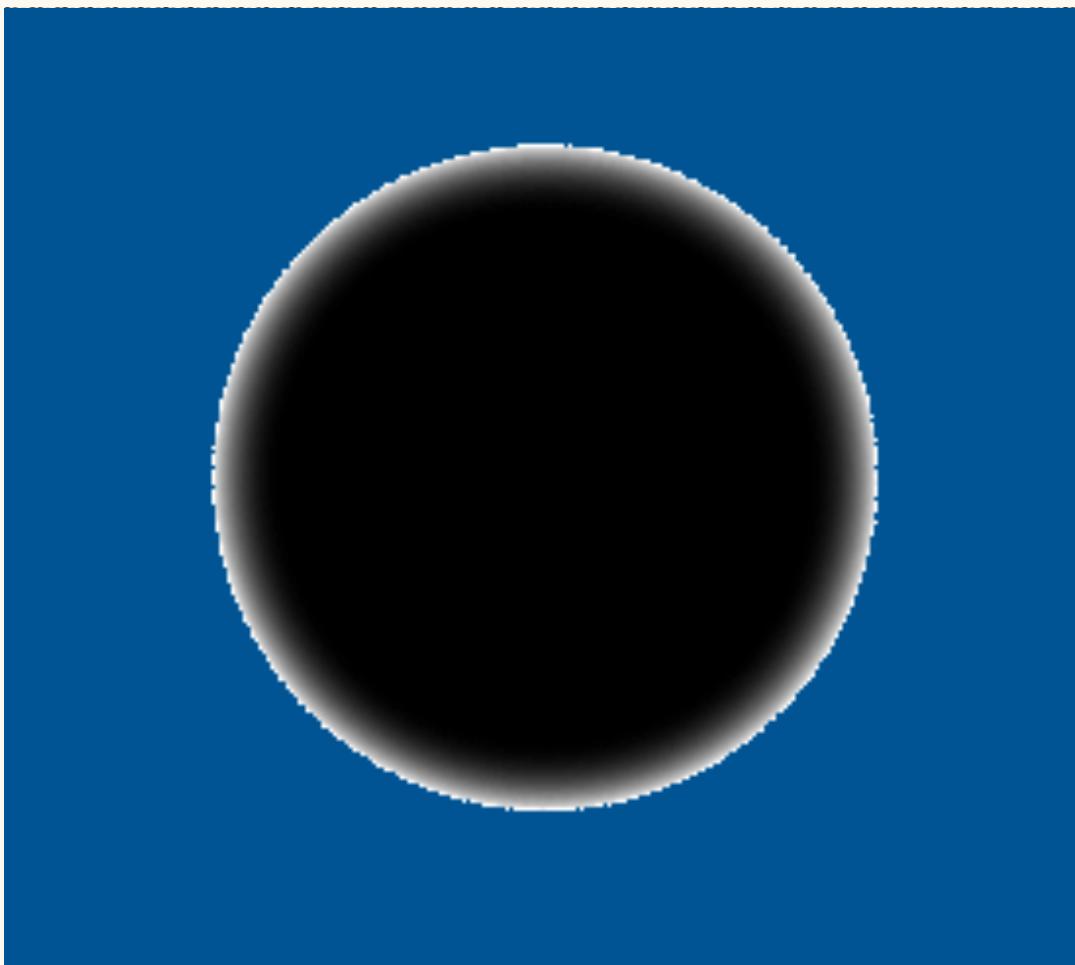


Real-time Skin Rendering

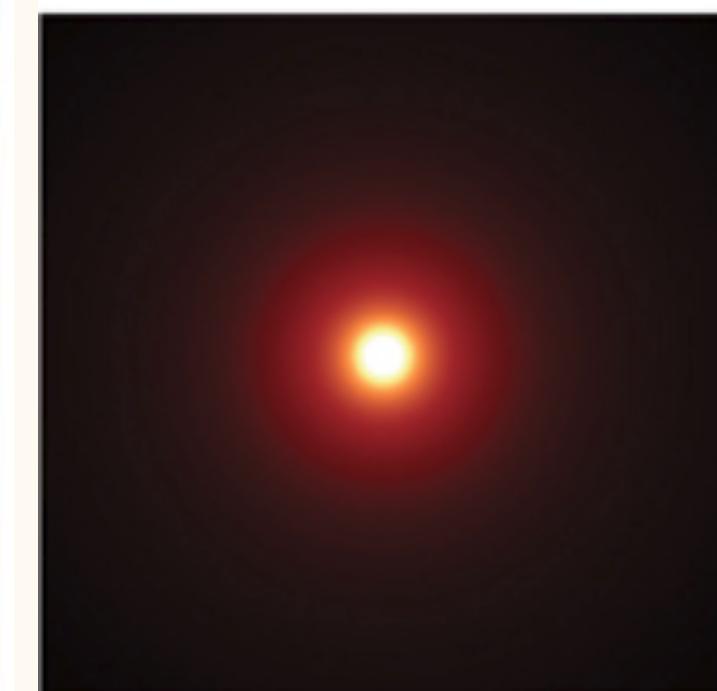
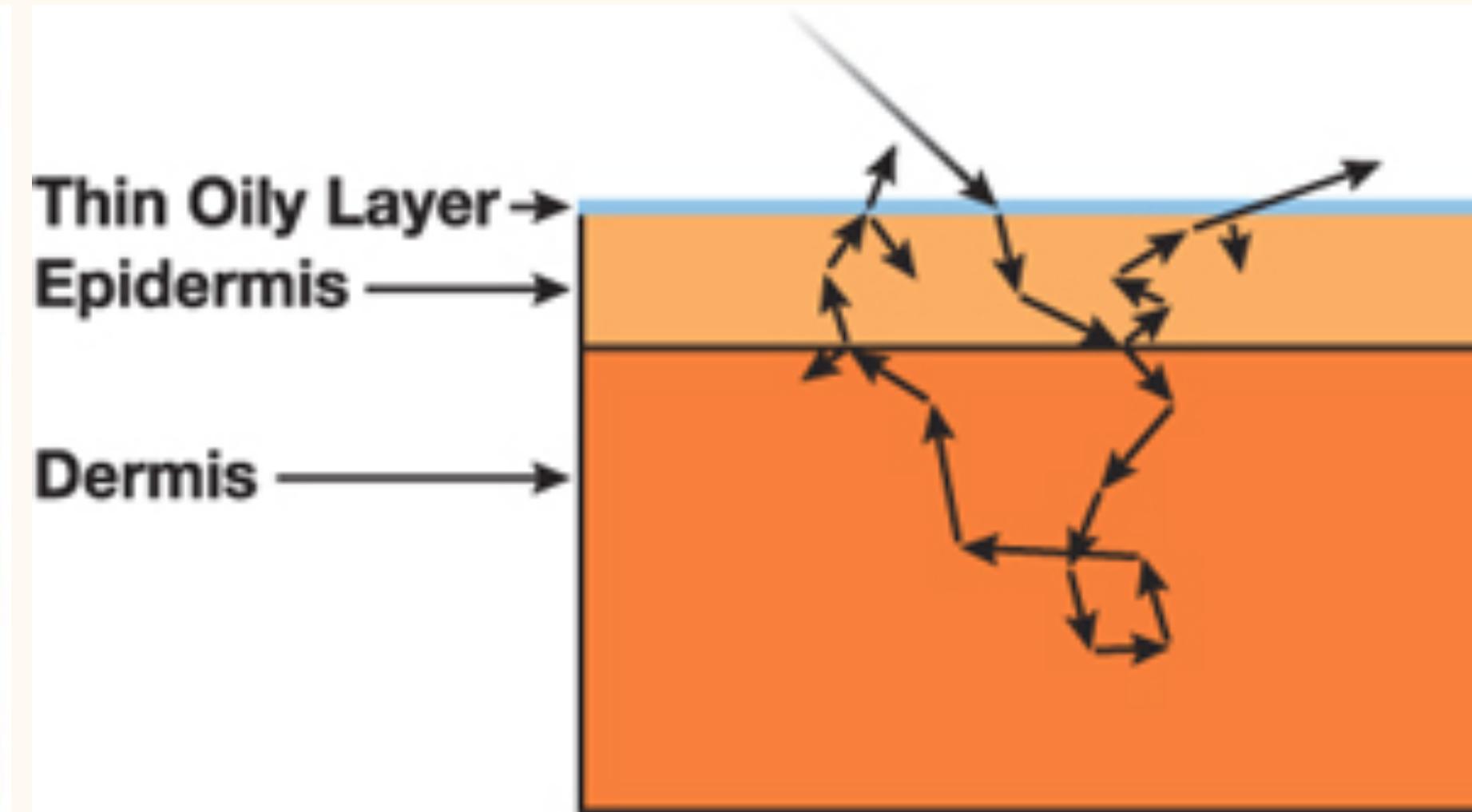
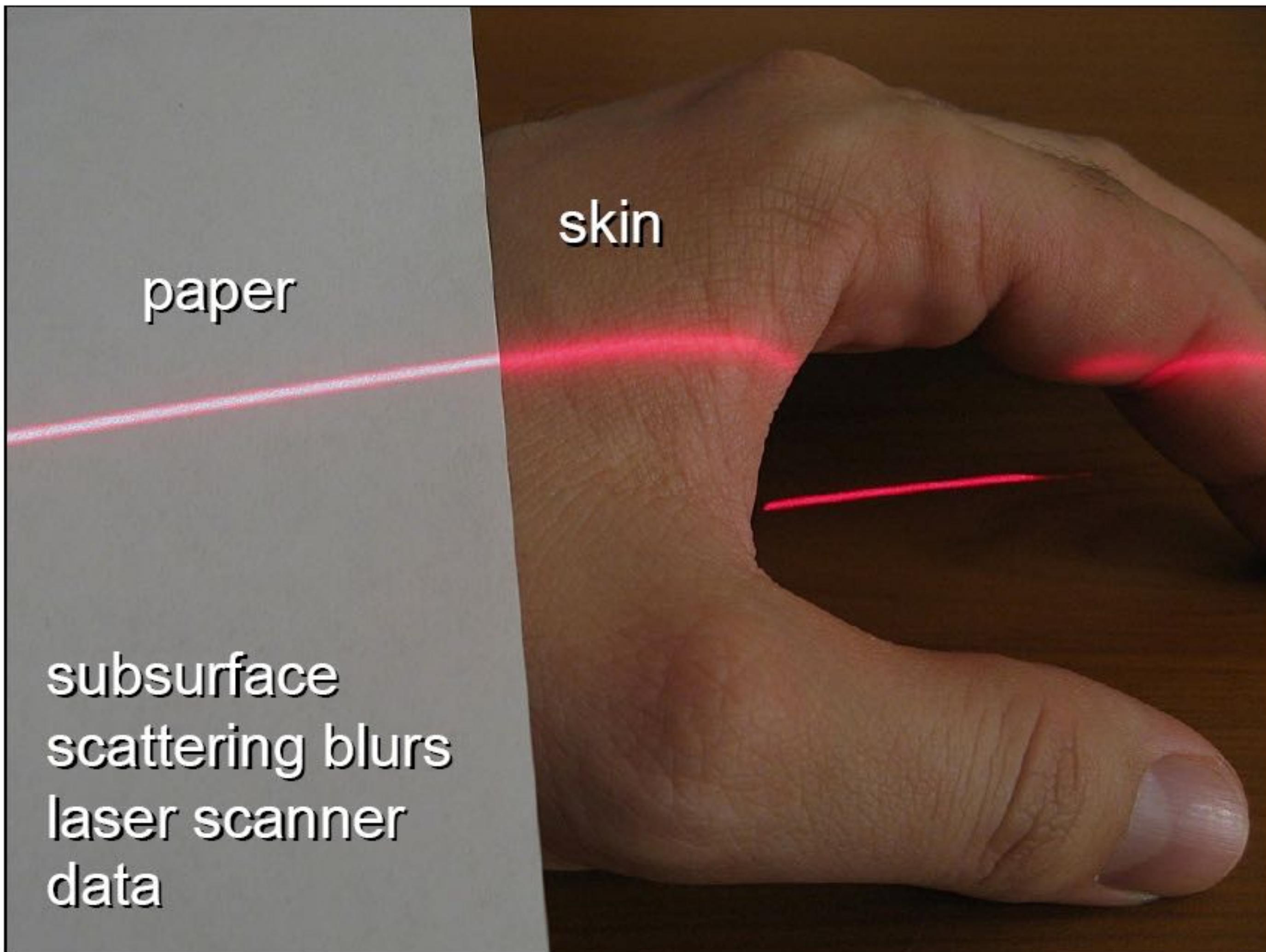
Skin Rendering – Rim Lighting

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

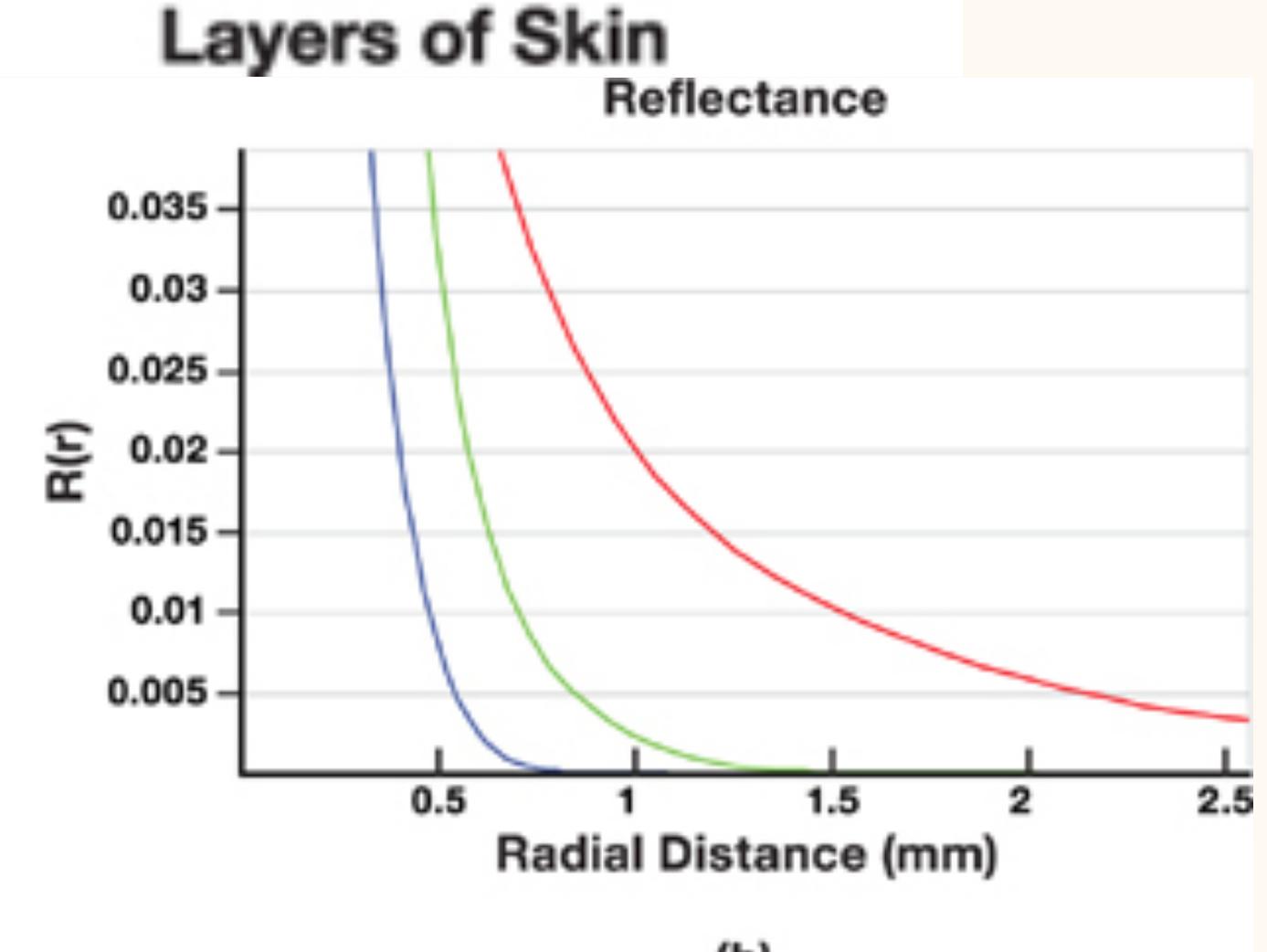
```
float dot = 1 - dot(nf, i);  
diffusecolor = smoothstep(1.0 - rim_width, 1.0, dot);
```



Skin Rendering - Sub-surface Rendering



(a)



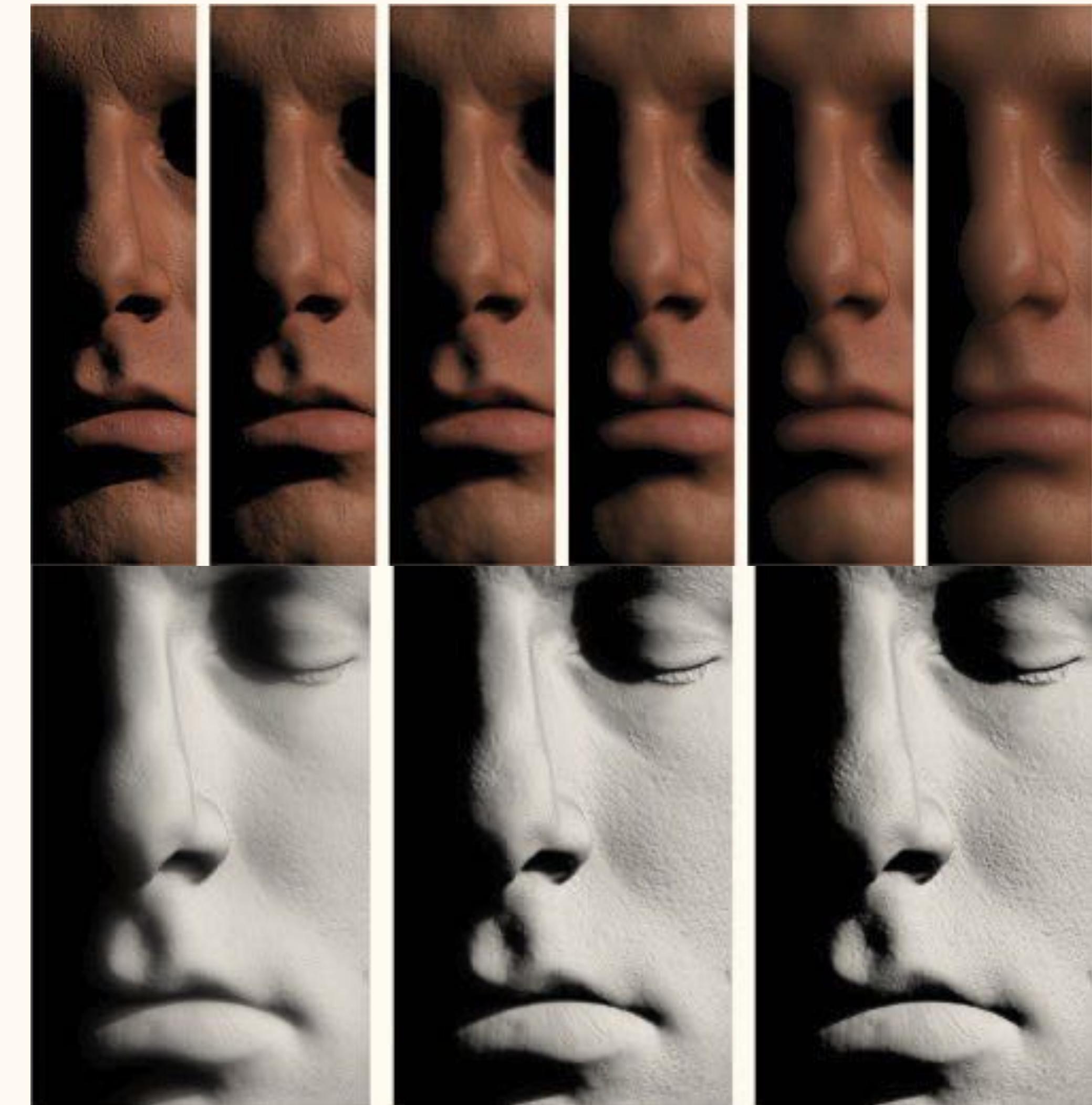
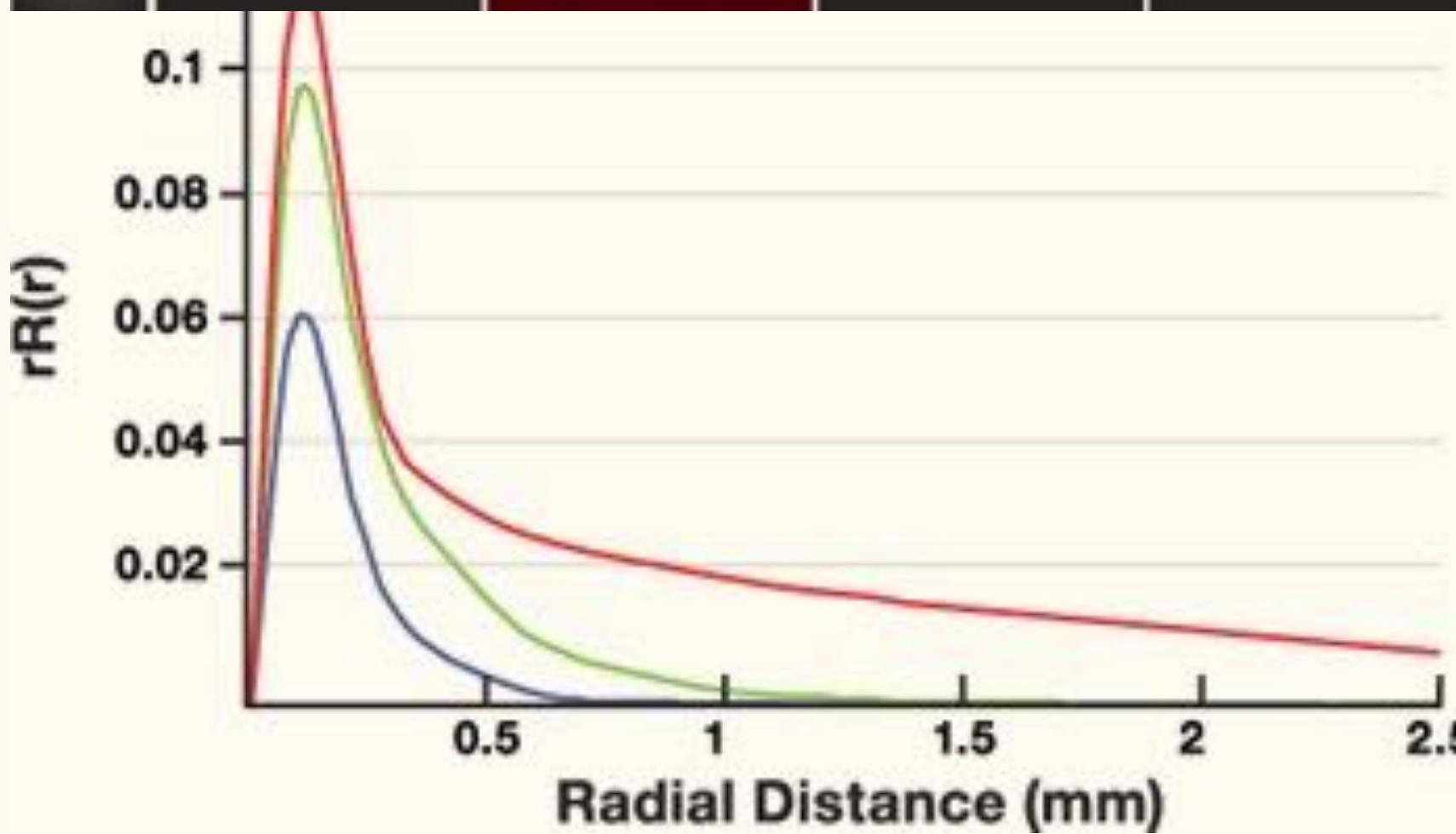
(b)

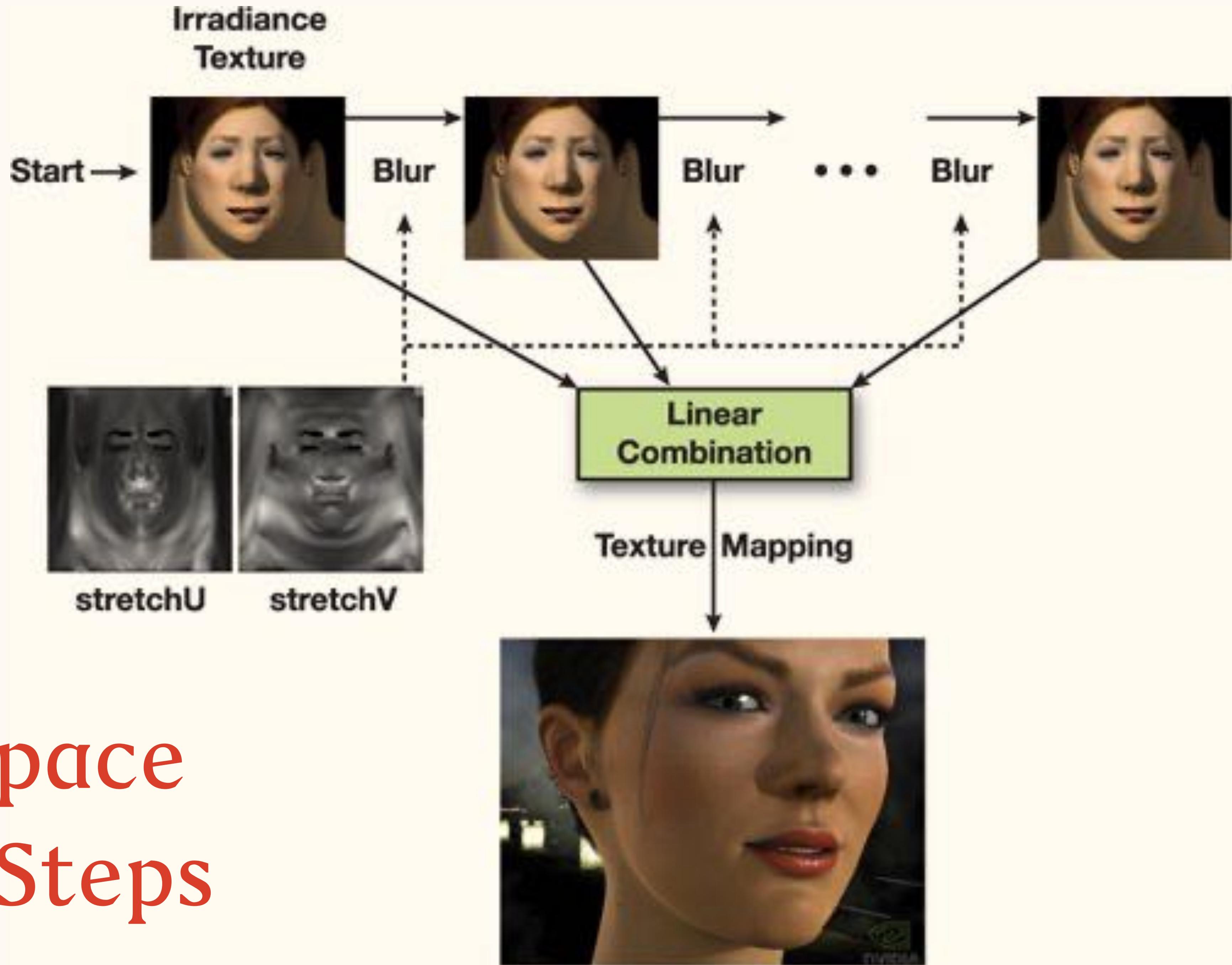
Skin Rendering – Sub-surface Rendering

- 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)	Red	Blur Weights 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

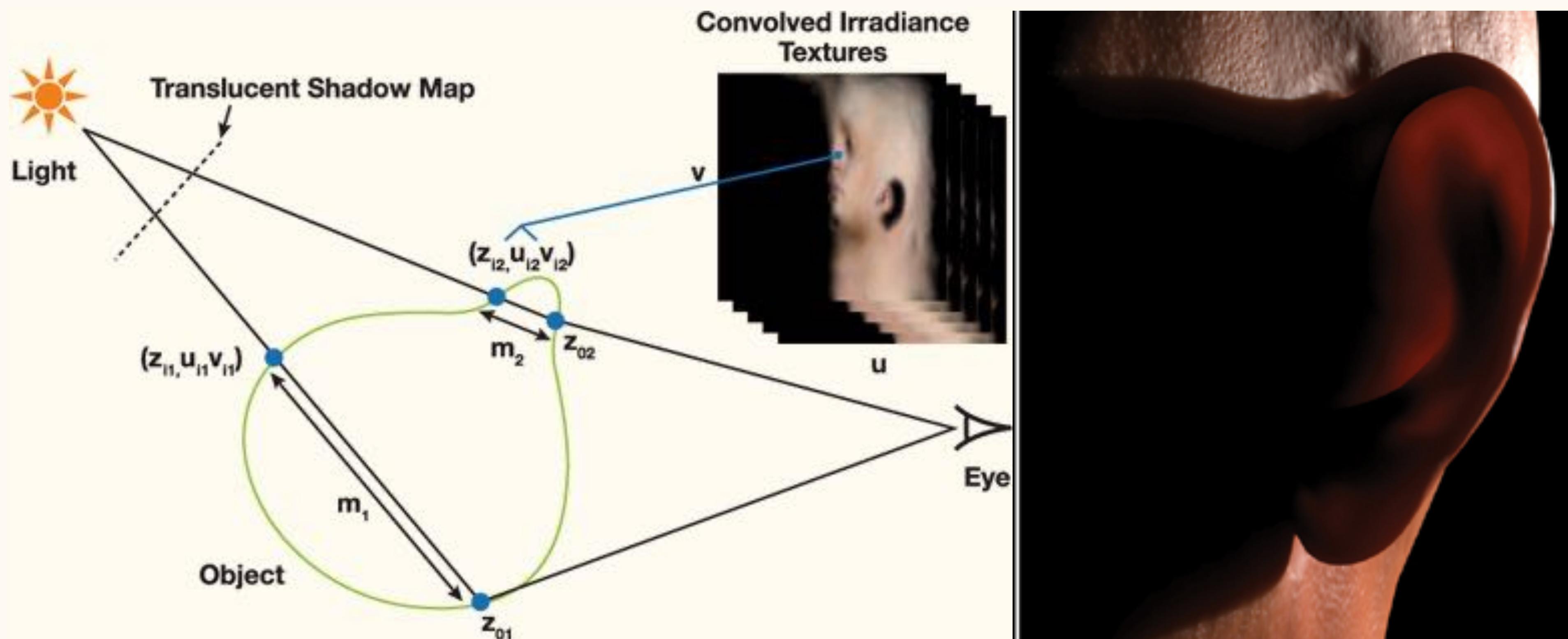


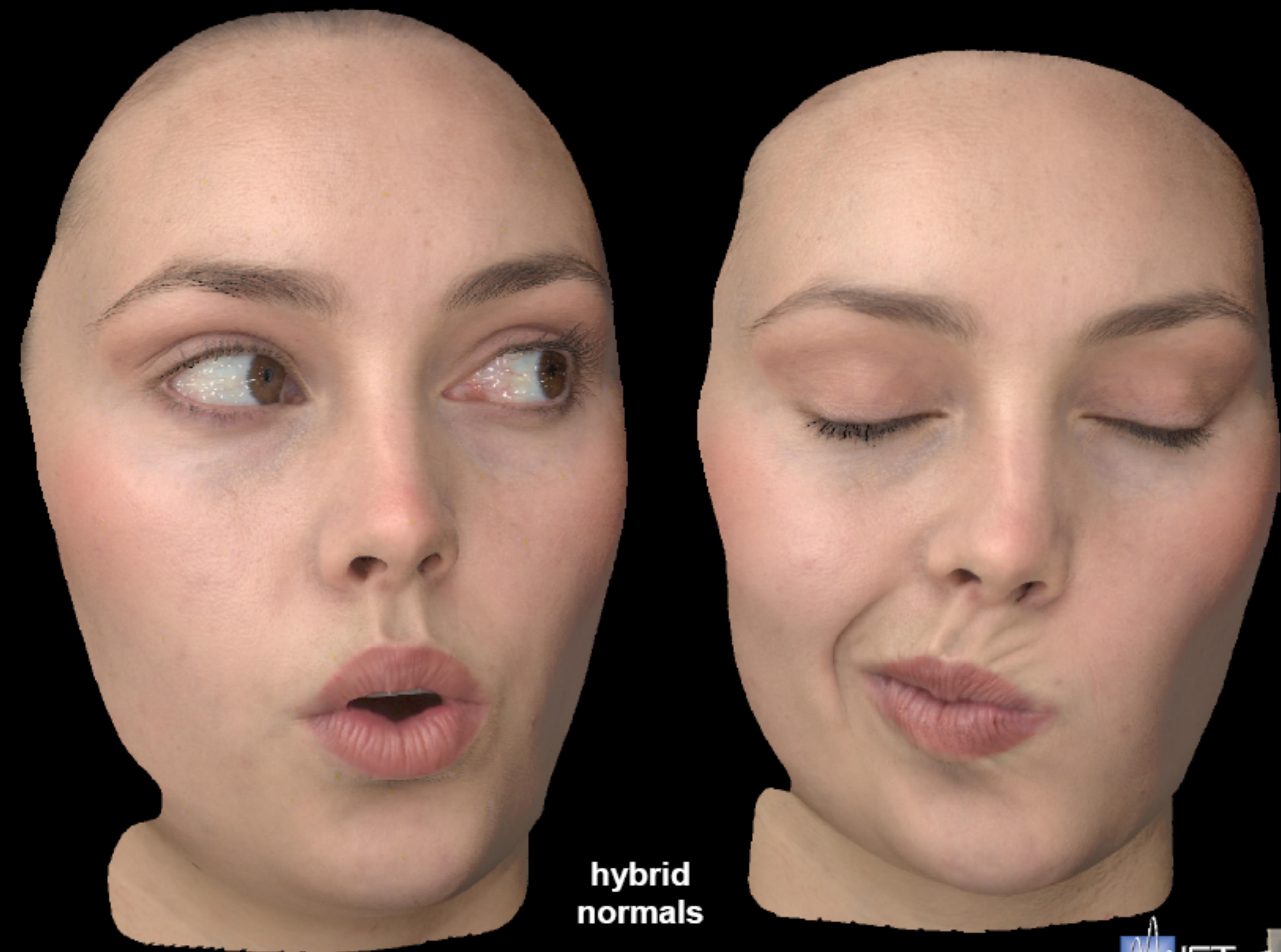


Texture-space Diffusion Steps

Real-time Subsurface Scattering Effect

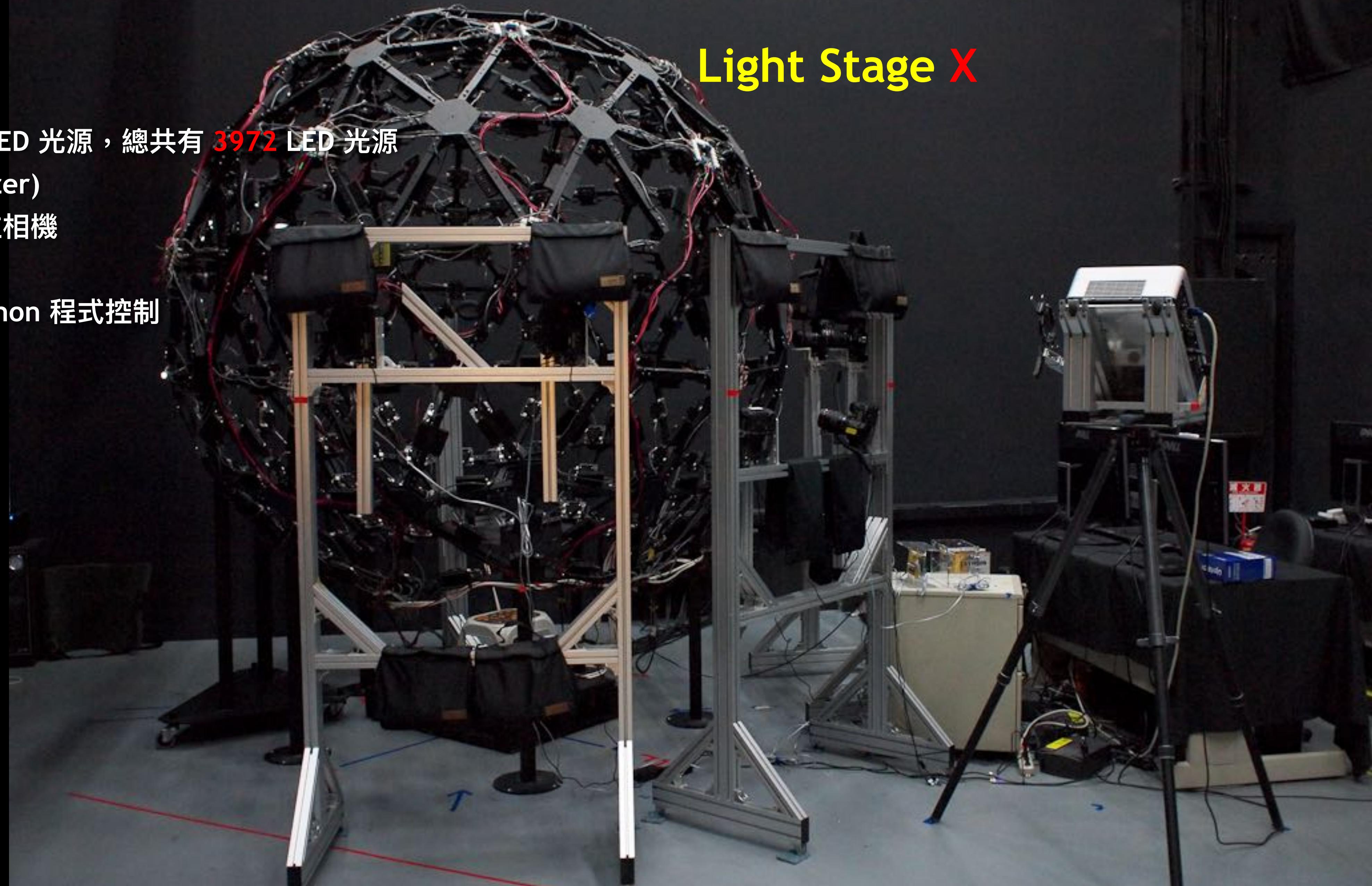
- Real-time sub-surface scattering
 - Modified translucent shadow maps



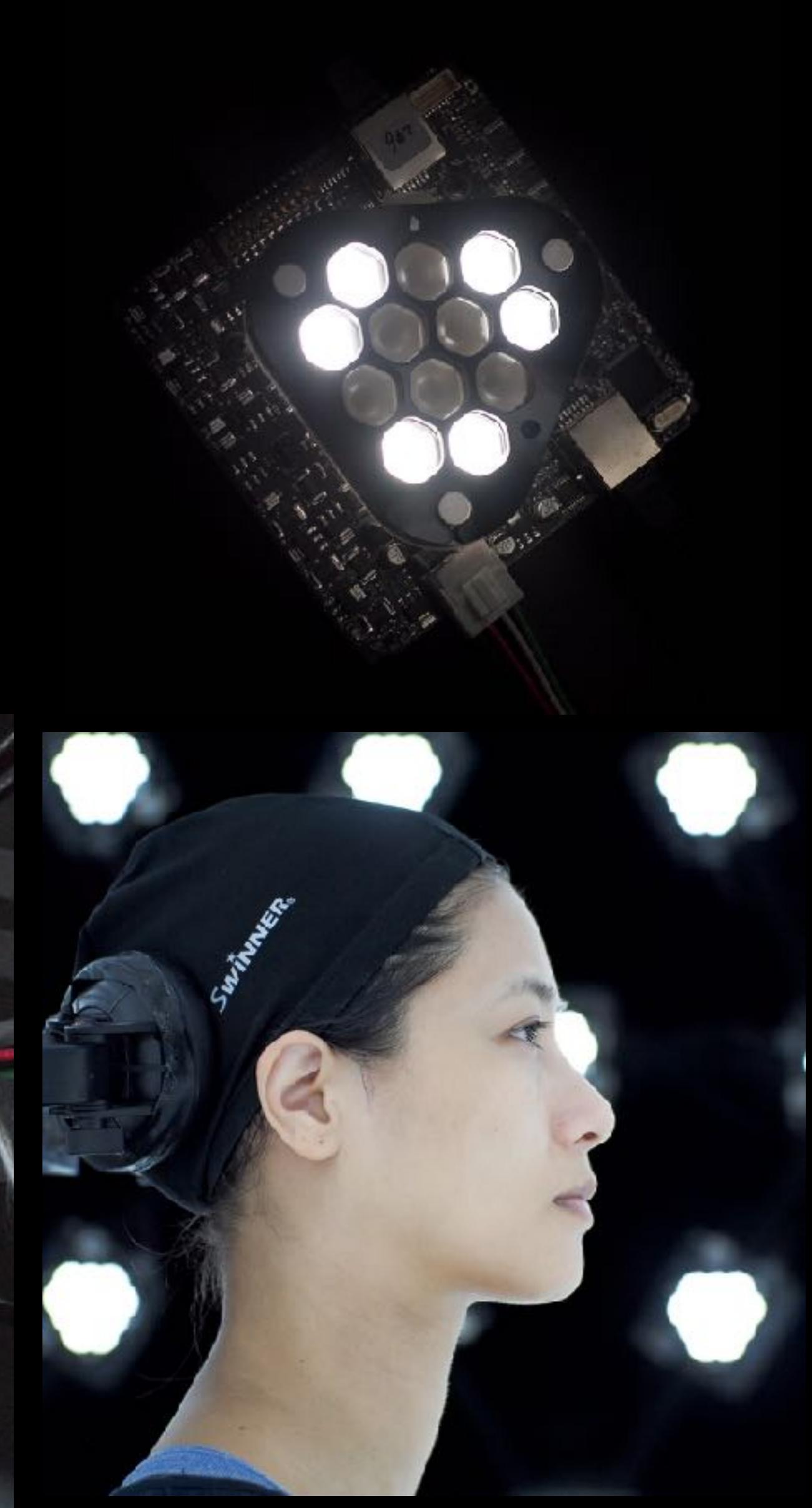
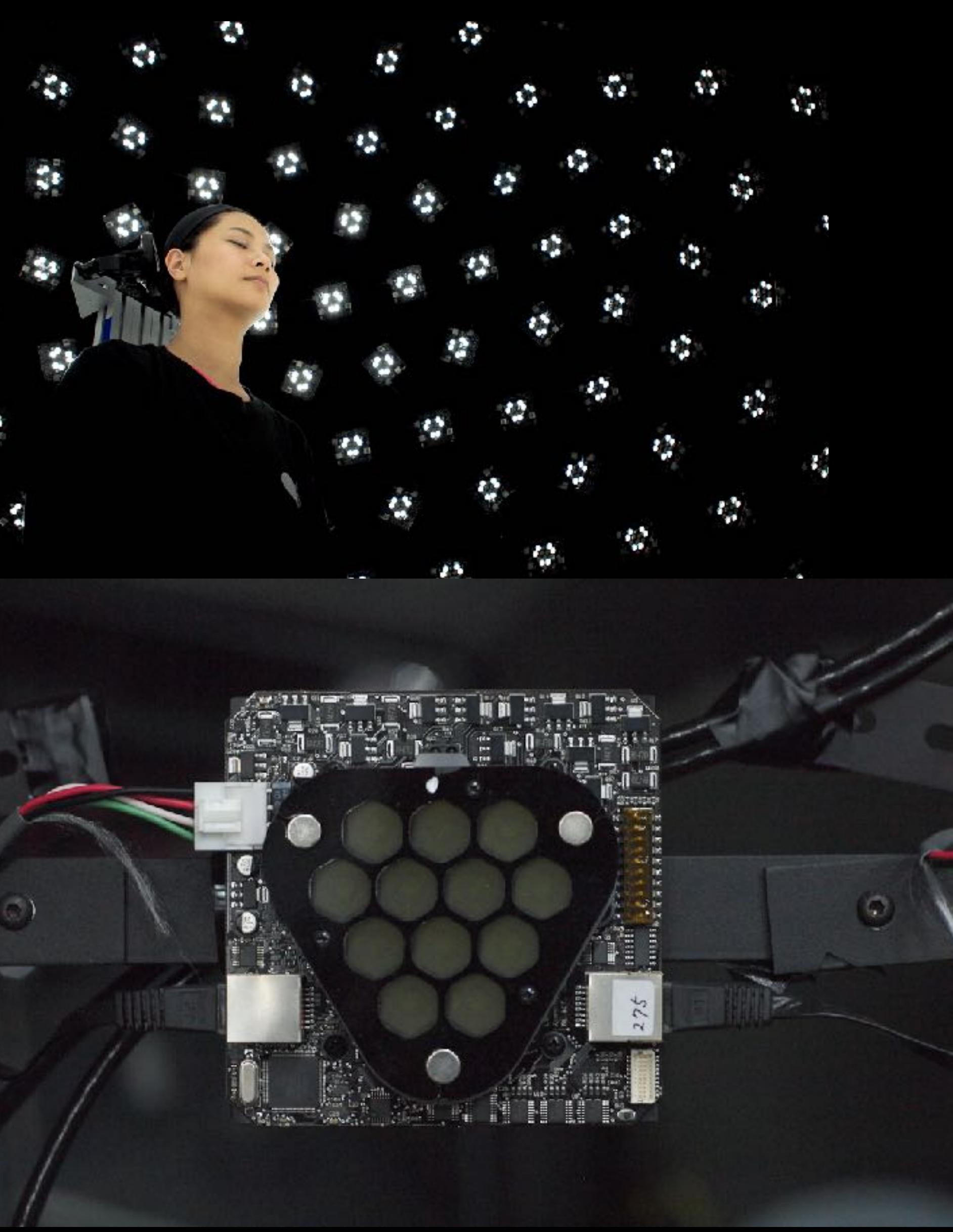
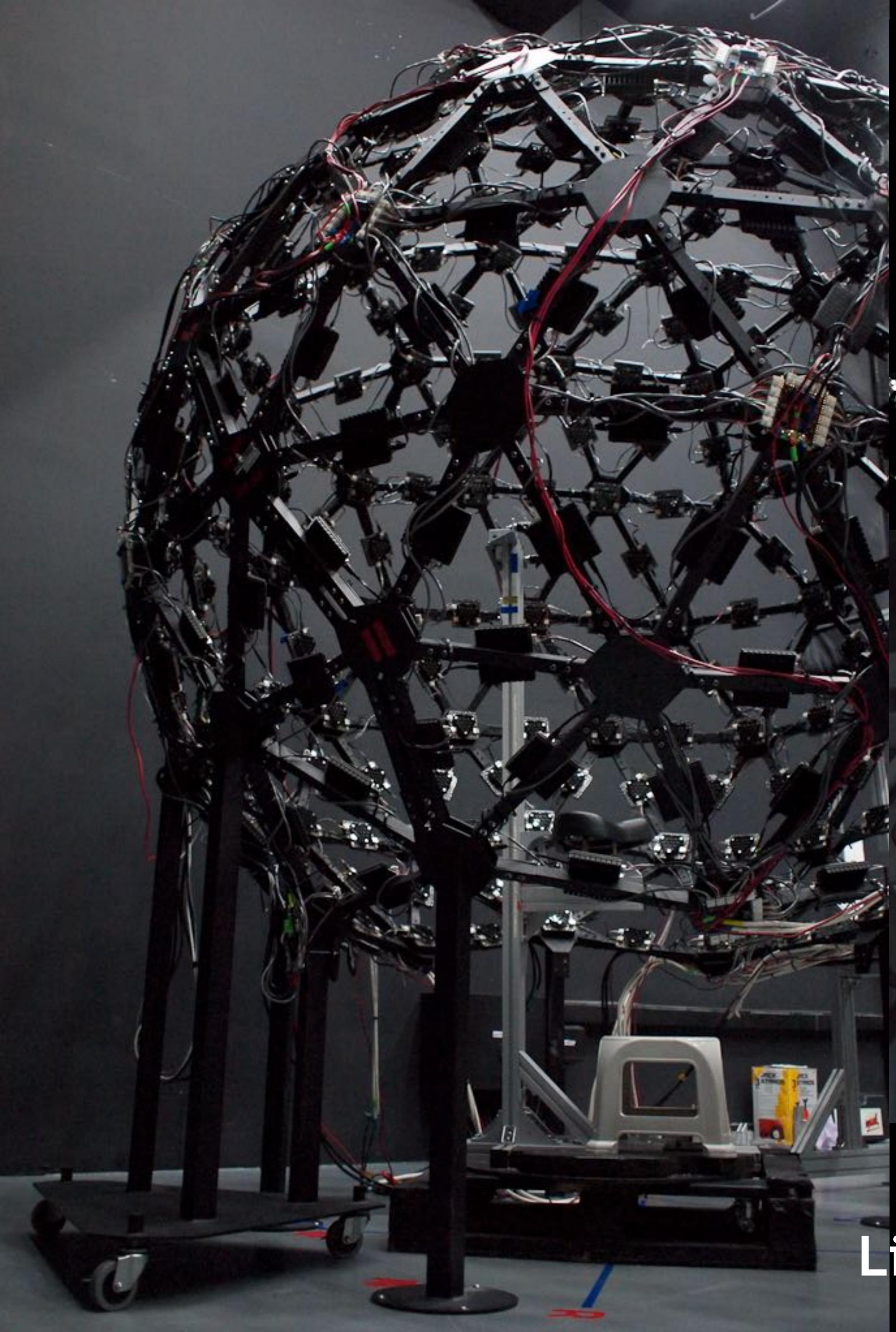


hybrid
normals

- 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



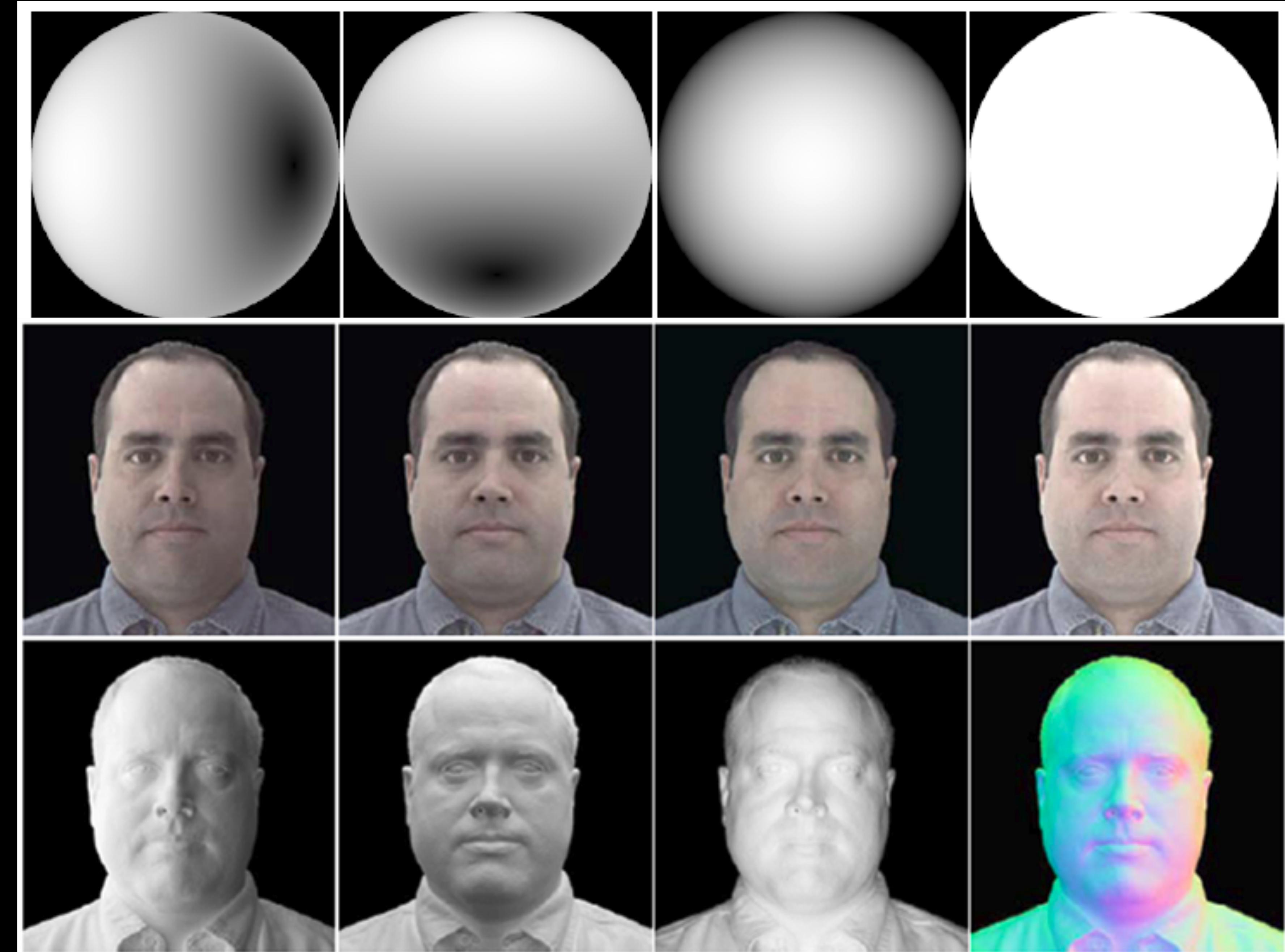
Light Stage @ Next Media Animation Media Lab, Taipei



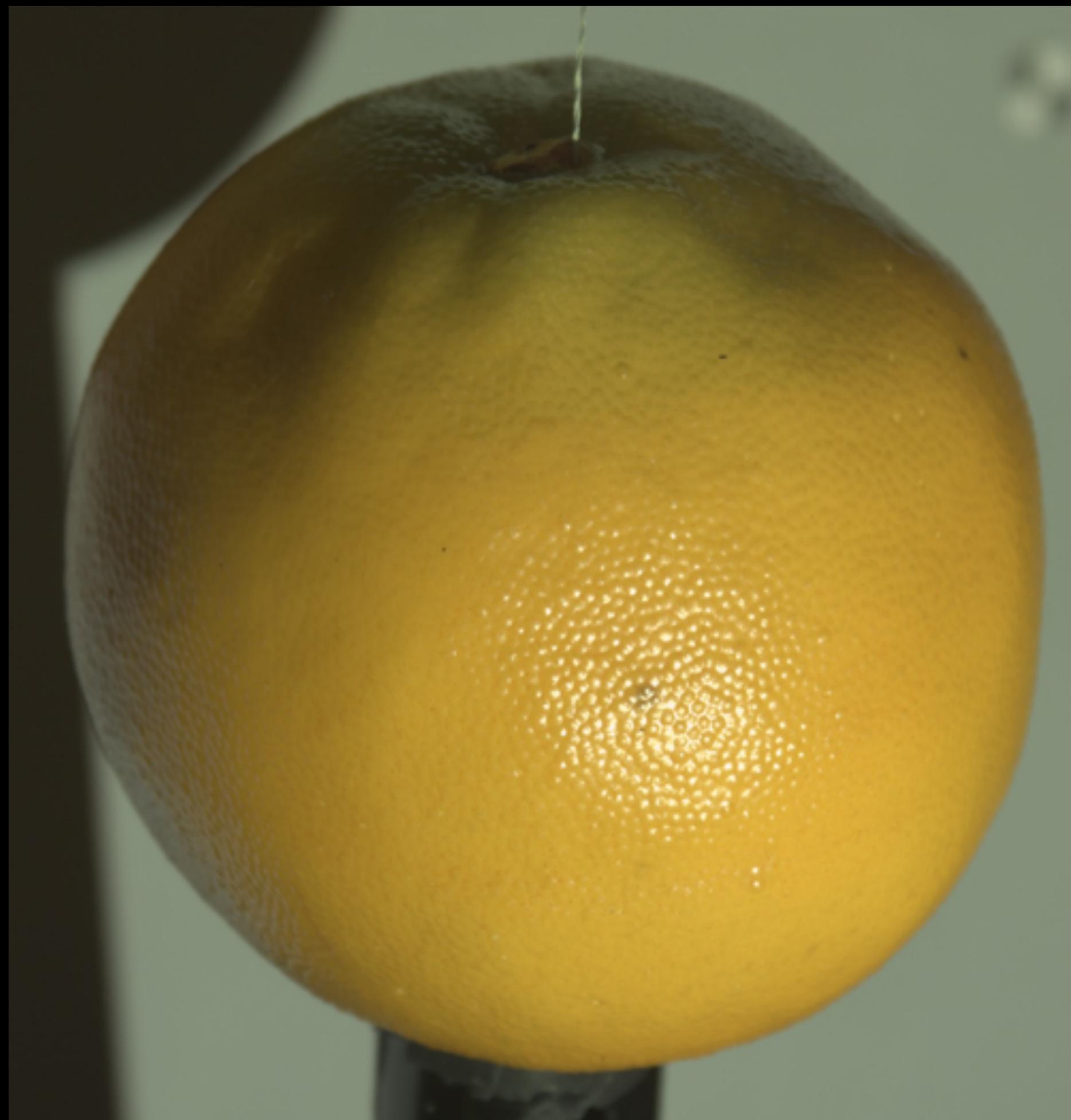
Light Stage @ Next Media Animation Media Lab, Taipei

Capture Normals

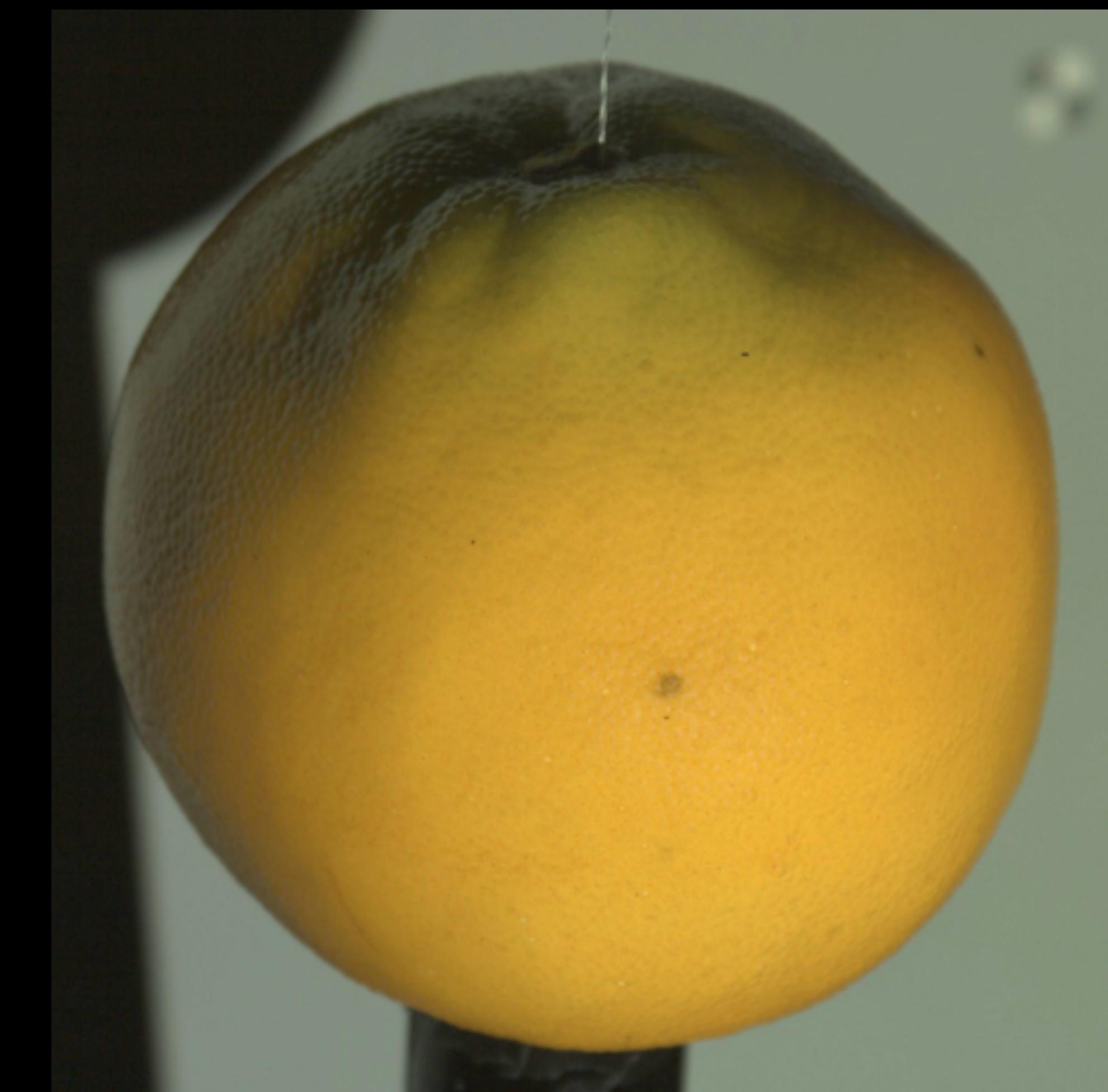
$K_d^*(N \cdot L)$



Remove Specular by Applying Linear Polarizer 線性偏光鏡



without a linear polarizer



with a linear polarizer

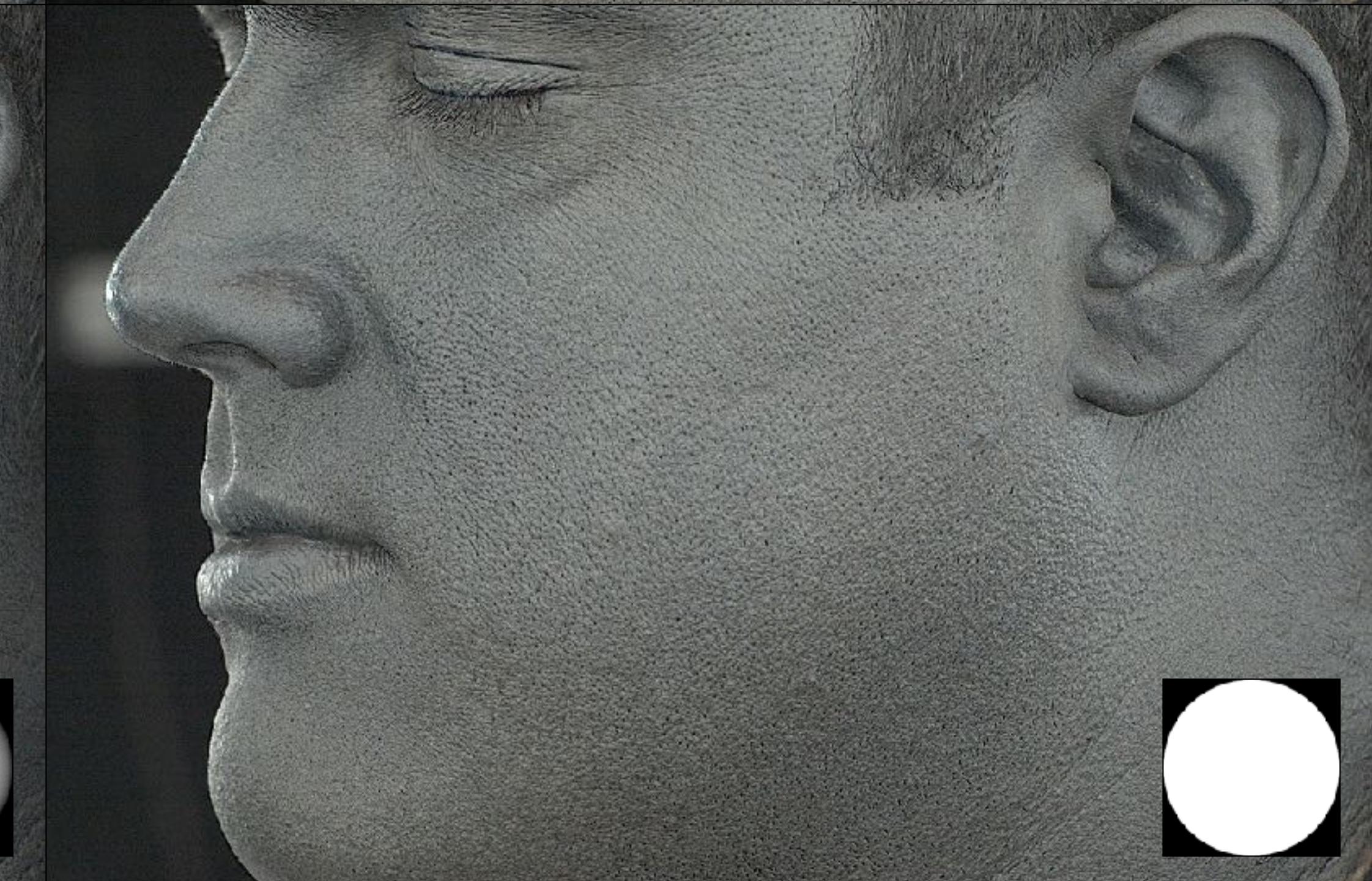
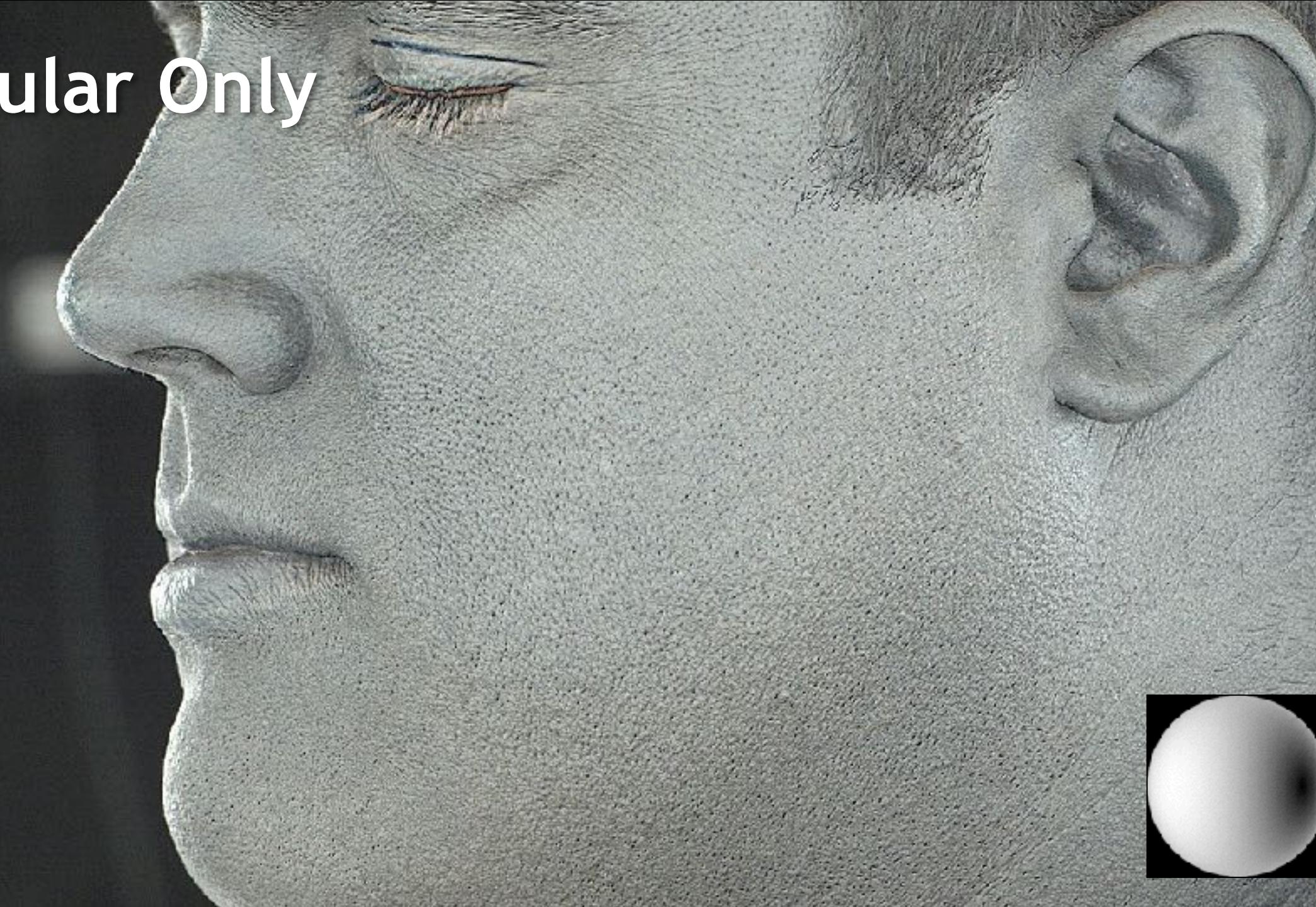
Diffuse Only



Diffuse Normal



Specular Only



Specular Normal



Normal map

Red diffuse



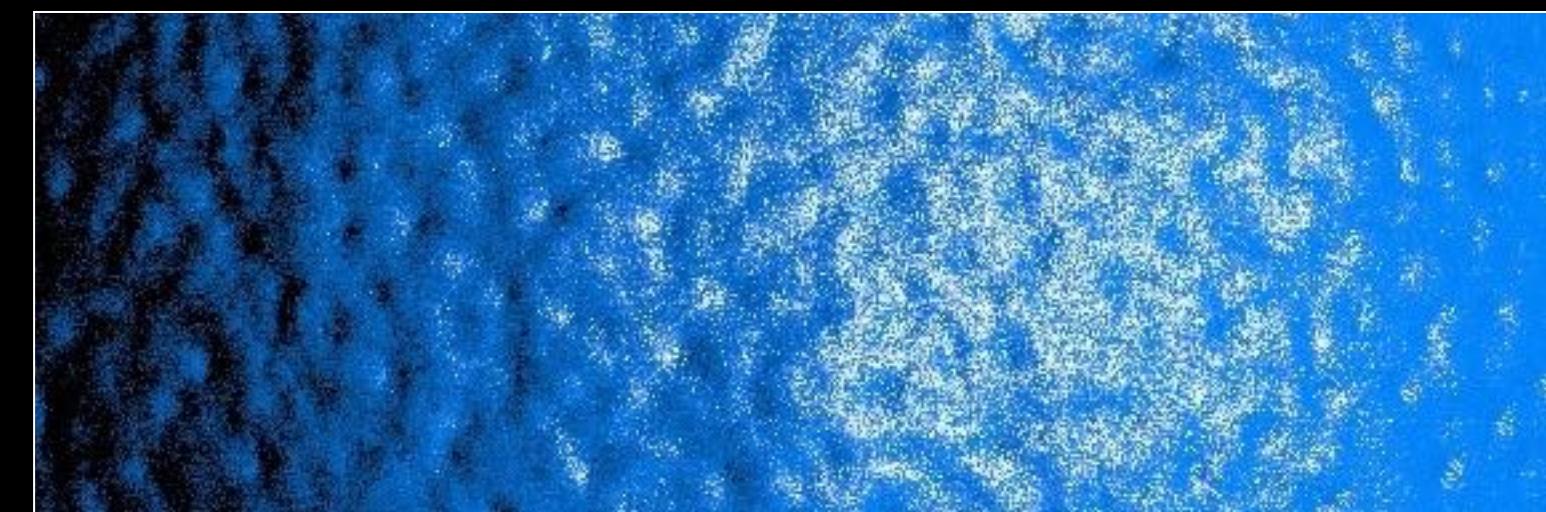
Shading



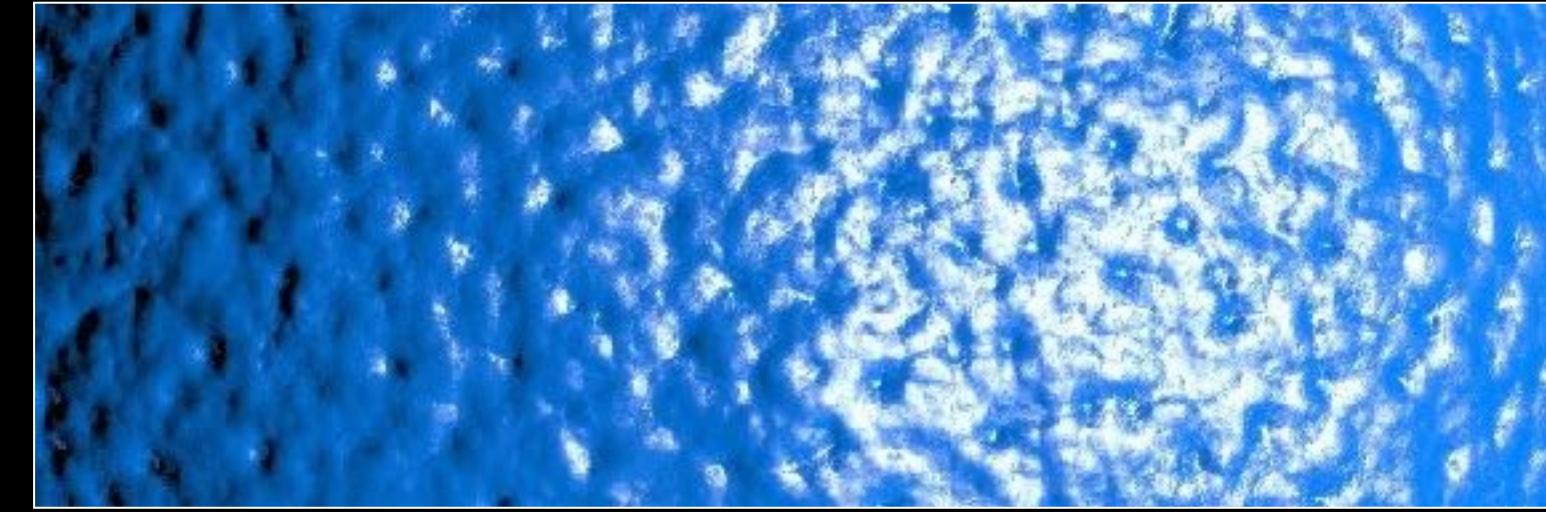
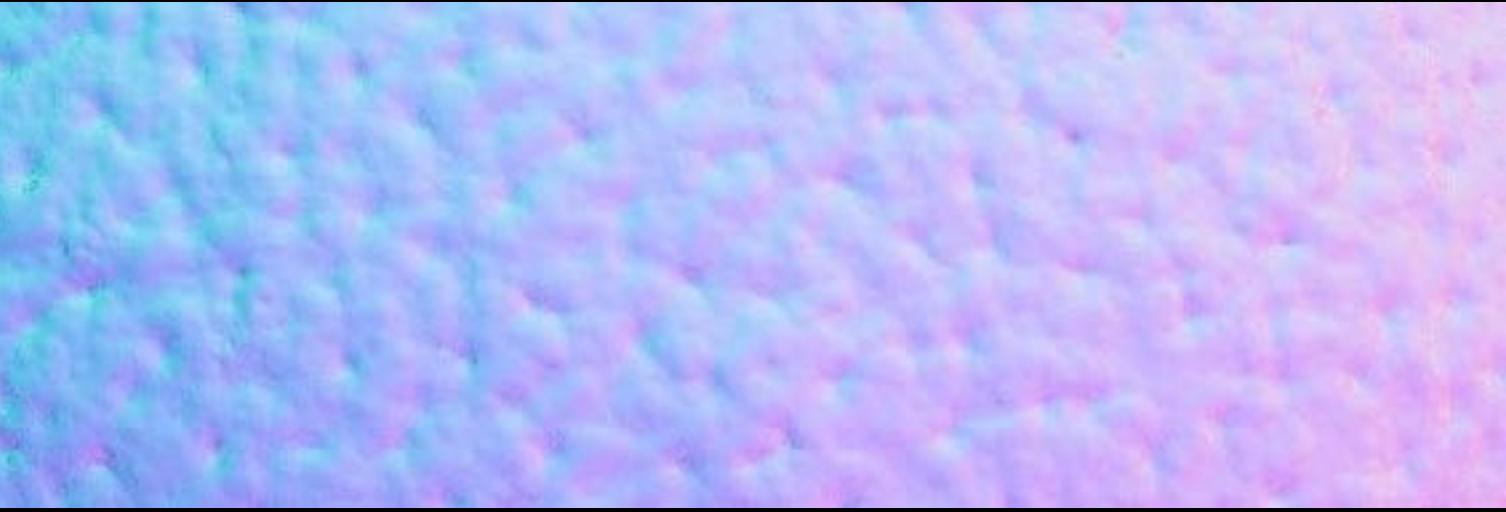
Green diffuse



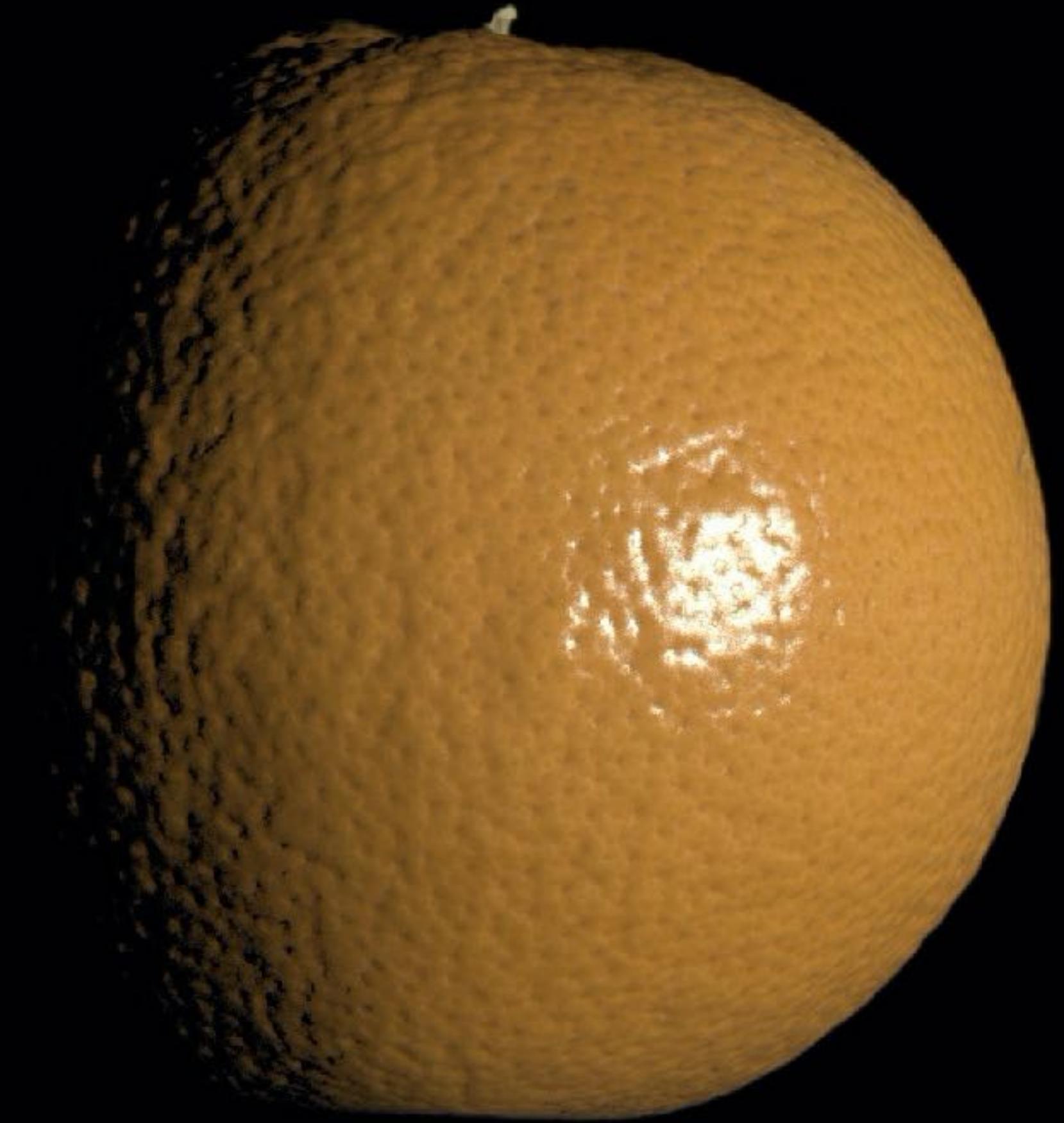
Blue diffuse



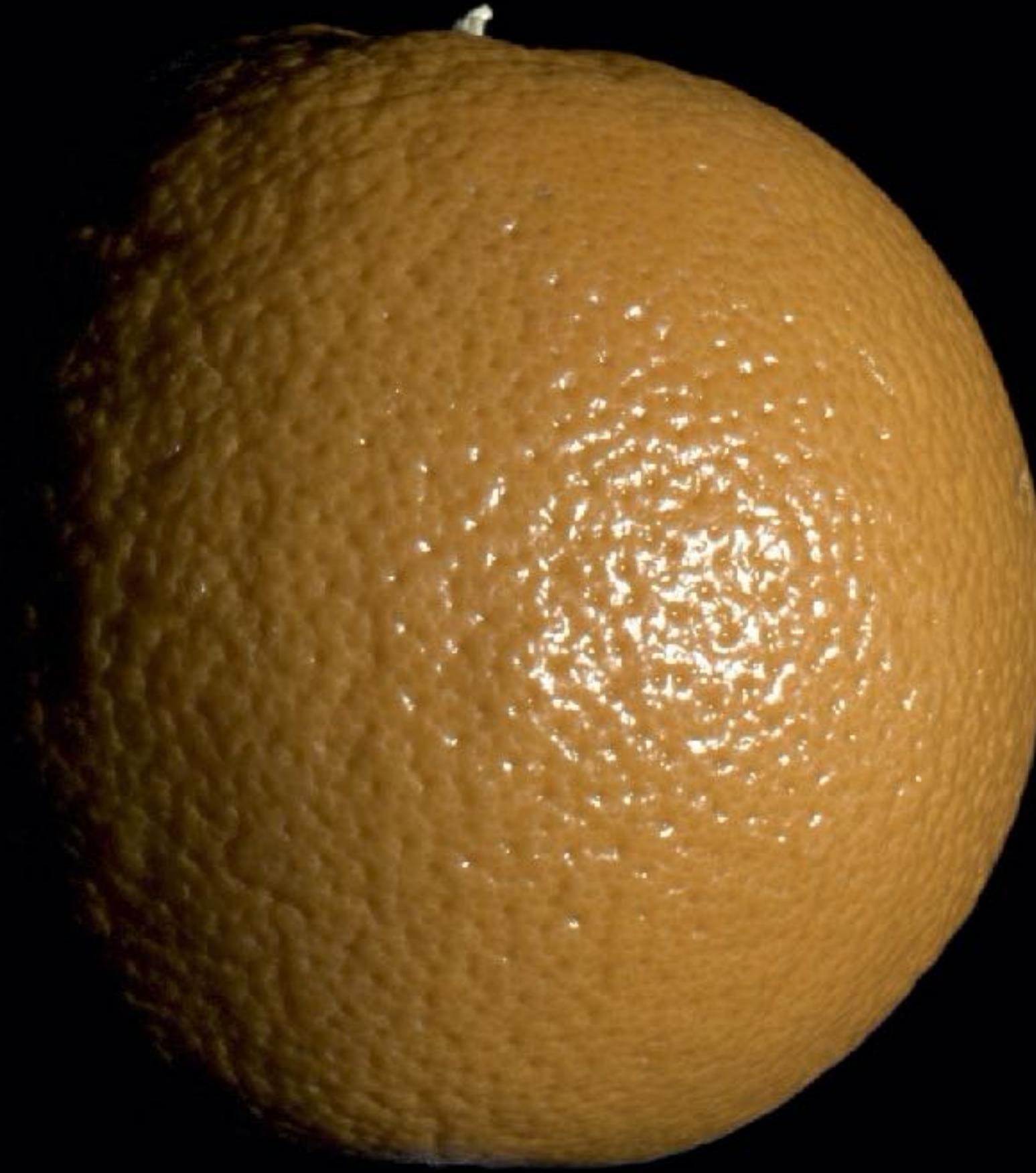
Specular



Render with Diffuse Normals



Diffuse

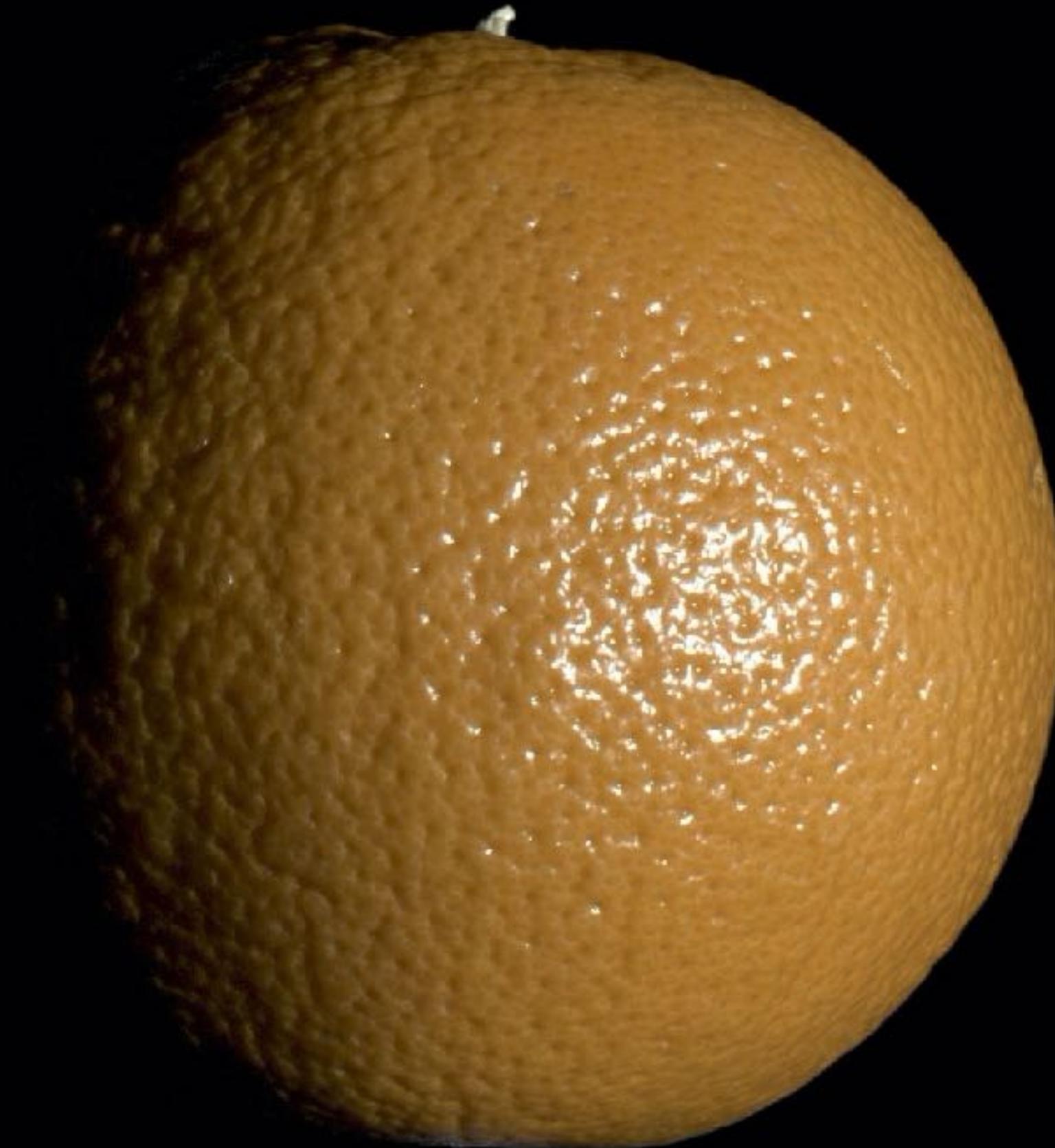


Real Photo

Render with Specular Normals

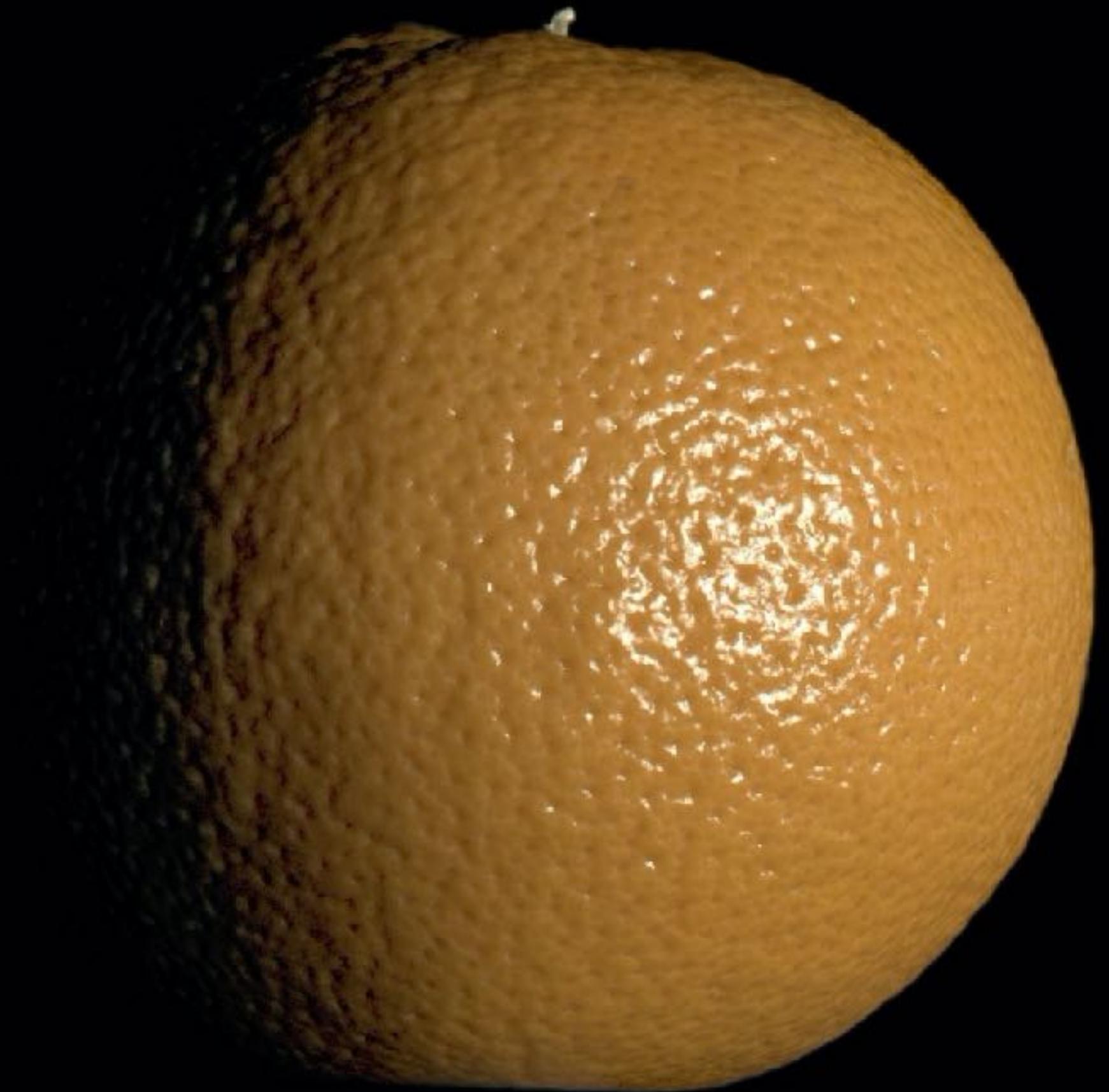


Specular



Real Photo

Render with Hybrid Normals



Hybrid



Real Photo

88.73

Render with Hybrid Normals

