Introduction to Graphics Programming and its Applications

繪圖程式設計與應用

Shading

Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS5507

Codeblock Conventions

Yellow Codeblock => Application Program (CPU)

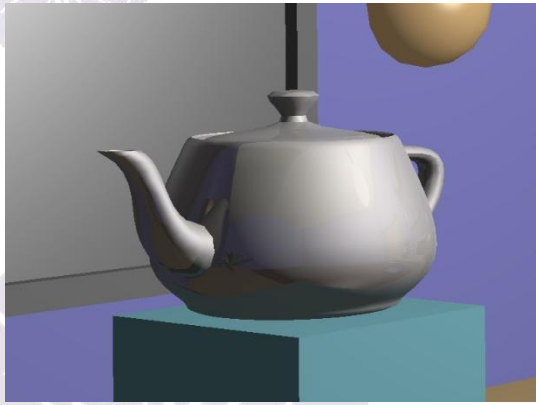
- Create OpenGL Context
- Create and Maintain OpenGL Objects
- Generate Works for the GPU to Consume

Blue Codeblock => Shader Program (GPU)

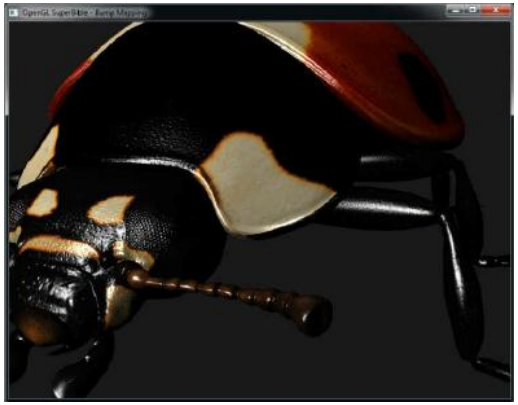
- Shader for a Certain Programmable Stage
- Process Geometry or Fragment in Parallel
- Starts with #version 410 core Declaration

What You'll Learn in This Lecture

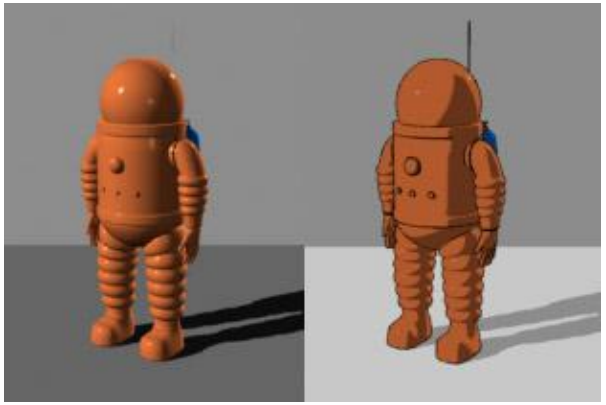
- How to lit up the pixels in your scene
- The techniques for adding extra resolution and realism to an object by various maps
- How to render a scene in cartoon-style



Shaded Scene



Normal Mapped Model

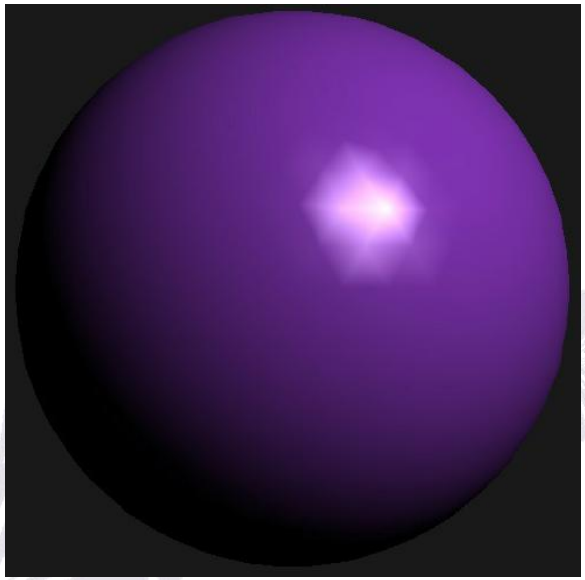


Cartoon Style Shading

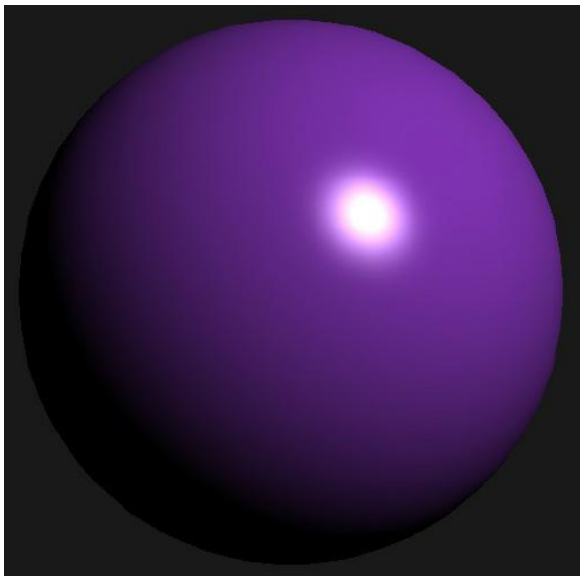
SHADING

Shading

- Shading is a ***light-material interaction*** in determining the color for each pixel of the rendered scene



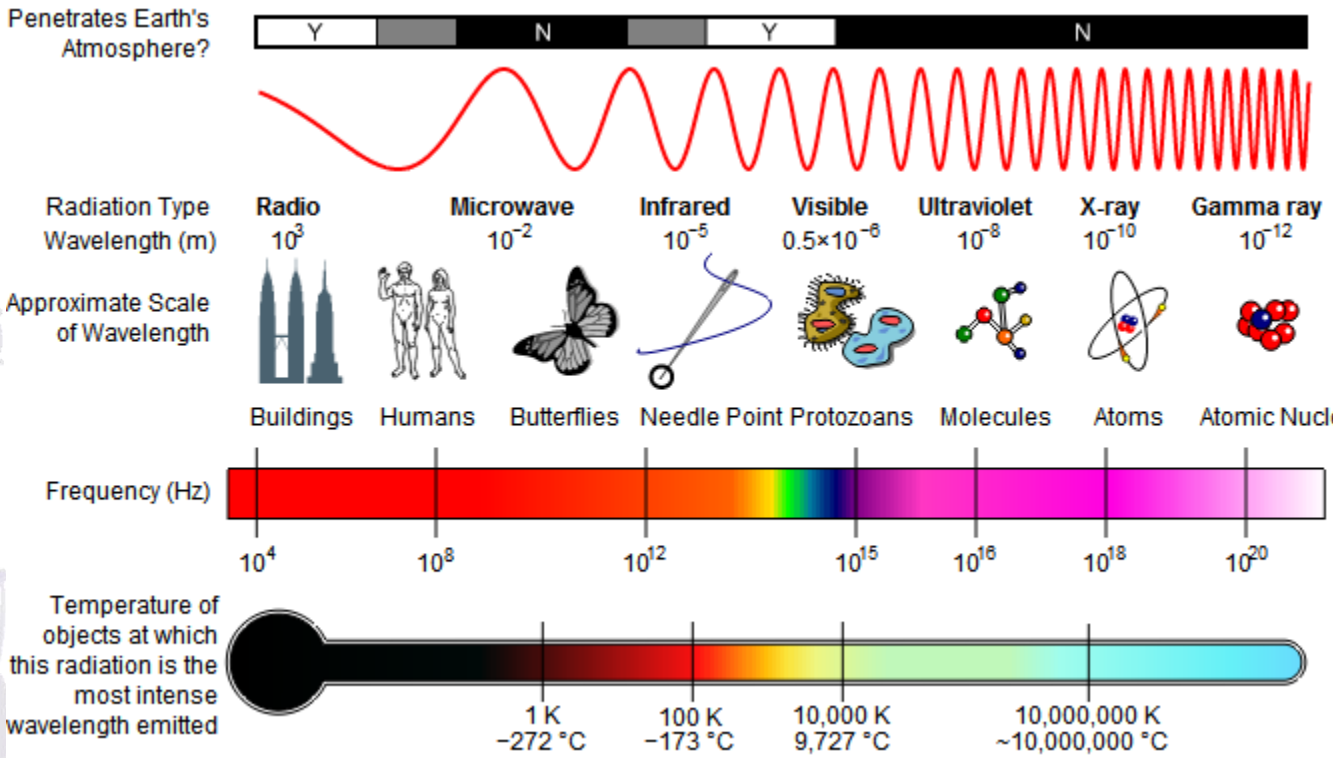
Gouraud Shading



Phong Shading

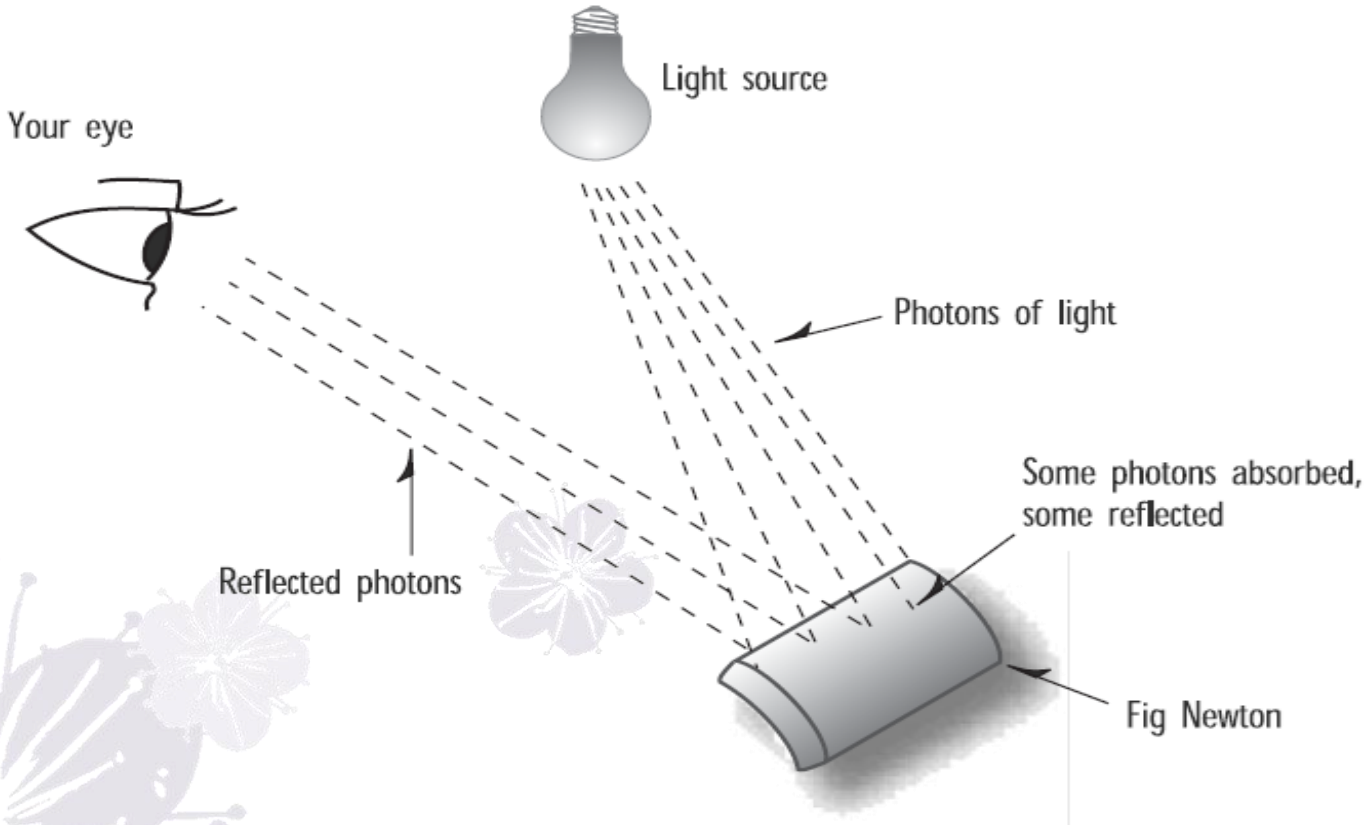
Physics: Light and Color

- Light as wave



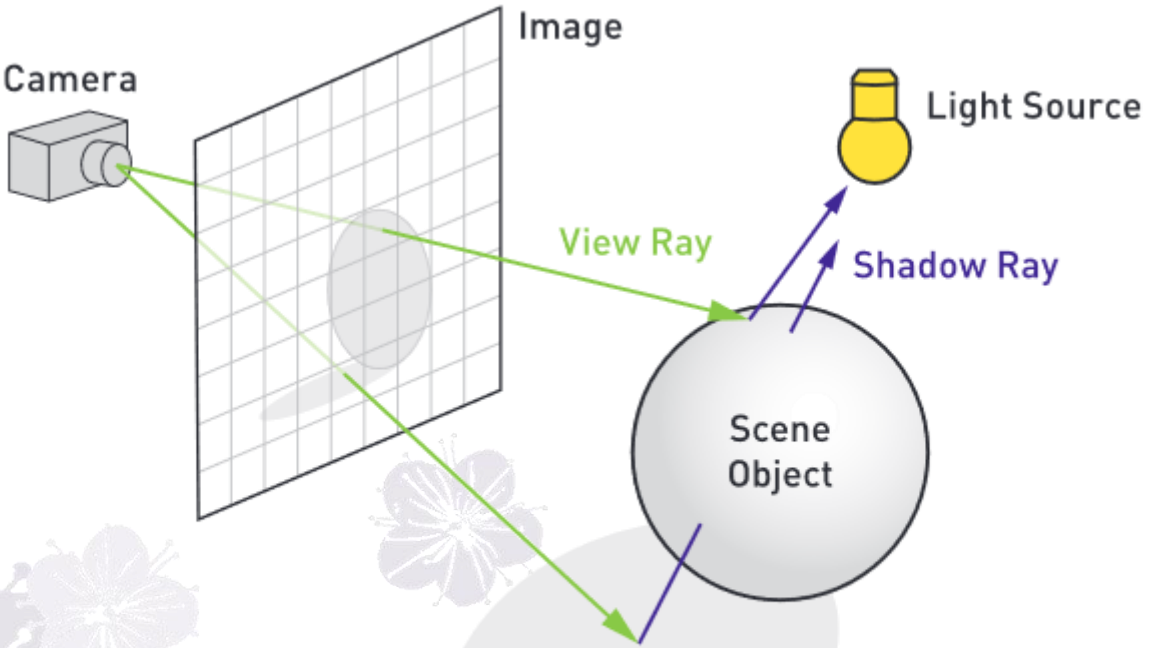
Physics: Light and Color

- Light as particles/photons



Graphics: Ray and Pixel

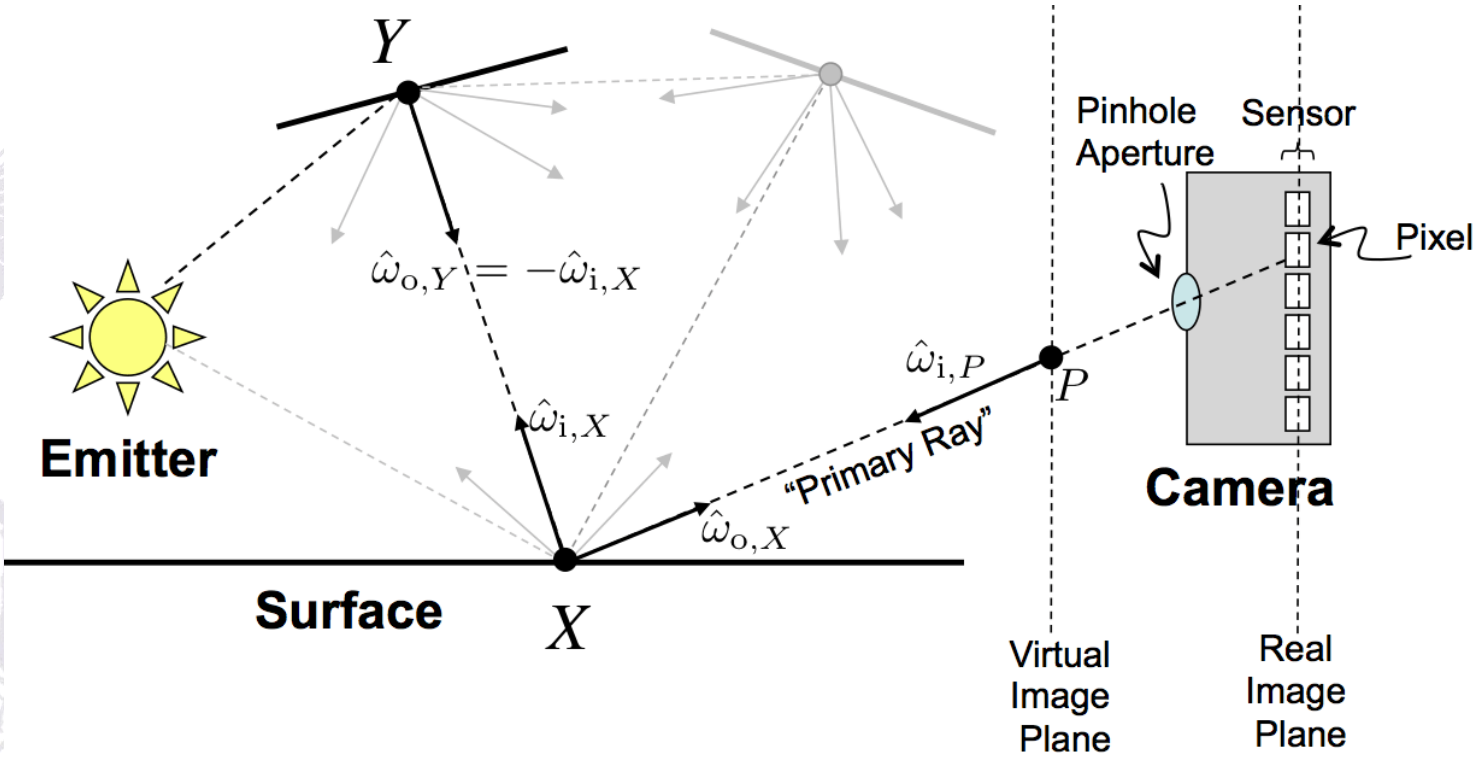
- Ray as direction



Surface Rendering Equation

Rendering a pixel I relates the outgoing light at X to the incoming light from the rest of the scene

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) + \int_{\mathbb{S}^2} L_i(X, \hat{\omega}_i) f_r(X, \hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i$$

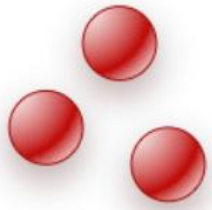


https://graphicscodex.courses.nvidia.com/app.html?page=_rn_rndEq

Radiometric Units (Physics)

The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**



Q

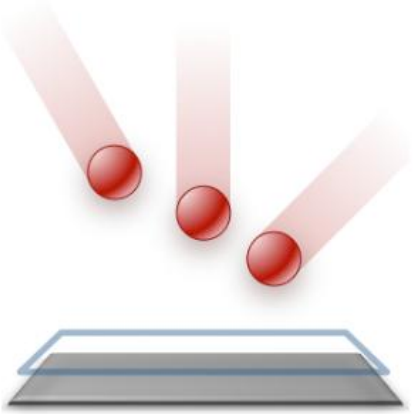
Energy [J]

Radiometric Units (Physics)

The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**

② **Power [W=J/s]**: To restrict the measurement of light to a single instant, I divided energy by time



$\Phi(t)$

Power [W]

Energy transmission rate through a **surface** from **all** directions

Energy per unit time

$$\Phi = \frac{dQ}{dt}$$

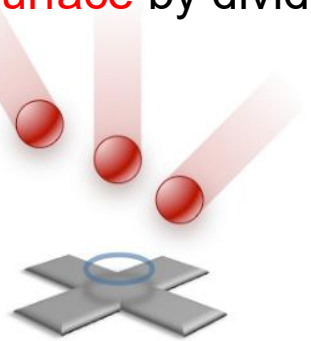
Radiometric Units (Physics)

The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**

② **Power [W=J/s]**: To restrict the measurement of light to a single instant, I divided energy by time

③ **Irradiance [W/m²]:** I then restricted the measurement to a single point on a surface by dividing by the area measure of that surface



$E(X)$

Irradiance [$\mathrm{W/m^2}$]

Incoming power at a **point** from **all** directions

Energy per unit time per unit area

$$E = \frac{d\Phi}{dA}$$

Radiometric Units (Physics)

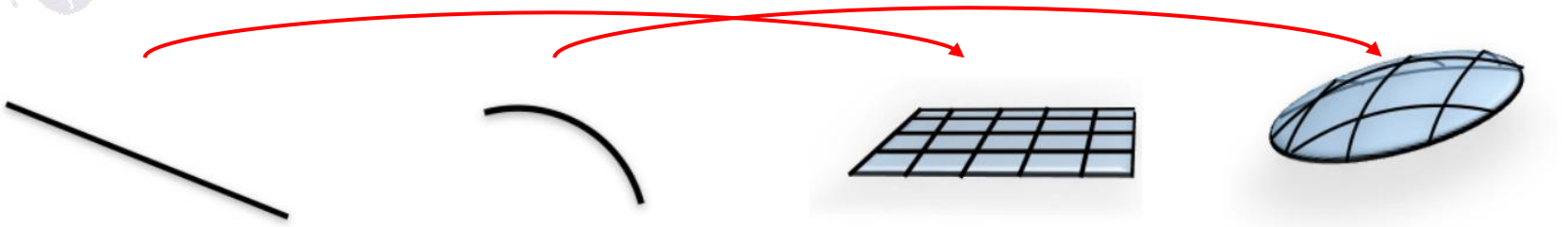
The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**

② **Power [W=J/s]**: To restrict the measurement of light to a single instant, I divided energy by time

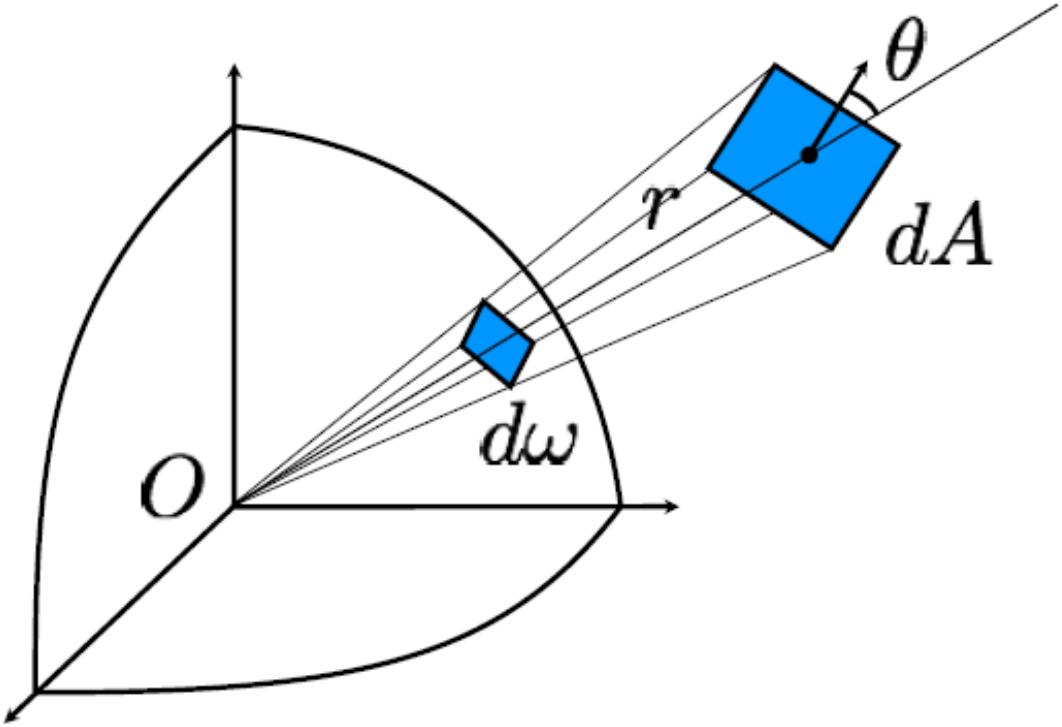
③ **Irradiance [W/m²]:** I then restricted the measurement to a single point on a surface by dividing by the area measure of that surface

④ **Solid Angle [sr=rad²]:** The area of a shape projected onto a unit sphere



d	θ	$\|\mathbb{A}\|$	$\|\Gamma\|$
Length [m]	**Angle Measure** [rad]	**Area Measure** [m^2]	**Solid Angle Measure** [sr]
Size of a curve in $\mathbb{R}^3$	Size of an arc on the unit circle	Size of a surface in $\mathbb{R}^3$	Size of a surface on the unit sphere, $\mathbb{S}^2$

Solid Angle



Full sphere has 4π steradians

Radiometric Units (Physics)

The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**

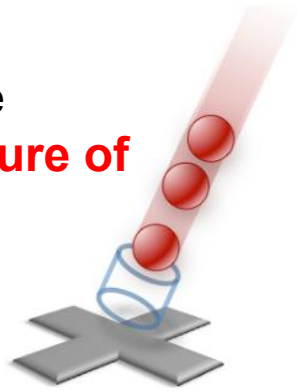
② **Power [W=J/s]**: To restrict the measurement of light to a single instant, I divided energy by time

③ **Irradiance [W/m^2]:** I then restricted the measurement to a single point on a surface by dividing by the area measure of that surface

④ **Solid Angle [sr=rad^2]:** A set of directions forms a region on the unit sphere

⑤ **Radiance [W/(m^{2}sr)]:** I divide by the measure of the solid angle subtended to restrict the measurement to a direction **(the measure of light at a point and a direction)**

$$L = \frac{dE}{d\omega} = \frac{d^2\Phi}{d\omega\, dA}$$



$L(X, \hat{\omega})$

Radiance [W/(m^{2}sr)]

Power at a **point** in **one** direction

Radiometric Units (Physics)

The light field function measures ***radiance*** $L(X, \hat{\omega}_o)$

① **Energy [J]:** Light is an electromagnetic wave. It carries energy, which is transported by discrete photons **(a ray of light is a stream of balls)**

② **Power [W=J/s]**: To restrict the measurement of light to a single instant, I divided energy by time

③ **Irradiance [W/m^2]:** I then restricted the measurement to a single point on a surface by dividing by the area measure of that surface

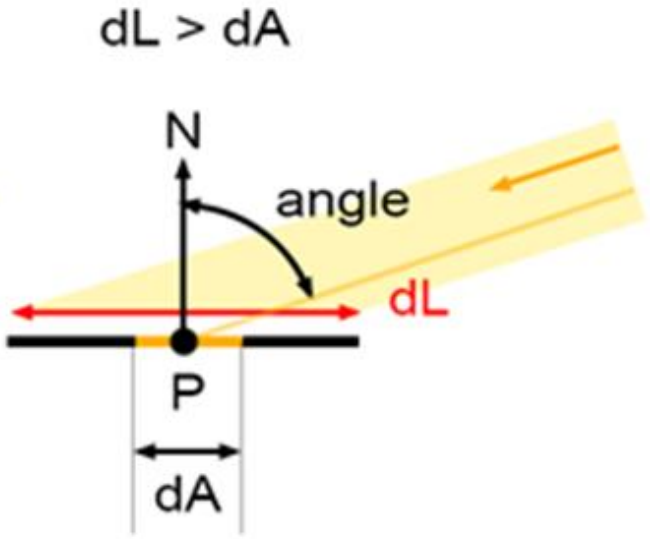
④ **Solid Angle [sr=rad^2]:** A set of directions forms a region on the unit sphere

⑤ **Radiance [W/(m^{2}sr)]:** I divide by the measure of the solid angle subtended to restrict the measurement to a direction **(the measure of light at a point (m^2) and a direction (sr))**

Lambert's Cosine Law

- Projected Area
  - The observed area of a surface decreases in proportion to the **cosine** of the **angle** between the **surface normal** and the **observation direction**.

$$L = \frac{d^2\Phi}{d\omega\, dA}\boldsymbol{cos\theta}$$



Surface Rendering Equation

Emission Recursion BSDF Cosine Law

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) + \int_{\mathbb{S}^2} L_i(X, \hat{\omega}_i) f_r(X, \hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i$$

Over a hemisphere

Symbol	*Type*	*Description*	*Ref*
$f_{X,\hat{n}}(\hat{\omega}_\mathrm{i}, \hat{\omega}_\mathrm{o})$	$\mathbb{R}\,\mathrm{sr}^{-1}$	Scattering distribution function.	[BSDF]
$L(X, \hat{\omega})$	$\mathbb{R}\,\mathrm{W/(m^2 sr)}$	Radiance through X in direction $\hat{\omega}$.	[L]
$L_\mathrm{i}(X, \hat{\omega}_\mathrm{i})$	$\mathbb{R}\,\mathrm{W/(m^2 sr)}$	Radiance transported through $X + \epsilon\hat{\omega}_\mathrm{i}$ in direction $-\hat{\omega}_\mathrm{i}$.	[Li]
$L_\mathrm{o}(X, \hat{\omega}_\mathrm{o})$	$\mathbb{R}\,\mathrm{W/(m^2 sr)}$	Radiance transported through $X + \epsilon\hat{\omega}_\mathrm{o}$ in direction $\hat{\omega}_\mathrm{o}$.	[Lo]
$\Phi(\mathbb{A}, \Gamma)$	$\mathbb{R}\ \mathrm{W}$	Power (flux) propagating through points in $\mathbb{A}$ in directions in Γ.	[power]
$S(\vec{v})$	$\mathbb{S}^n$	Normalize $\vec{v} \in \mathbb{R}^n$ by projecting it onto $\mathbb{S}^n$.	[nrmlz]
$\mathbb{S}^2$		The unit direction sphere.	[S2]
X	$\mathbb{R}^3\,\mathrm{m}$	A point in the scene at which photons may interact with matter.	[sctvar]
$\hat{n}$	$\mathbb{S}^2$	Unit normal to the surface point X.	[sctvar]
$\hat{\omega}_\mathrm{i}$	$\mathbb{S}^2$	Unit incident light direction (opposite the direction of photon propagation, pointing back at where the light came from).	[sctvar]
$\hat{\omega}_\mathrm{o}$	$\mathbb{S}^2$	Unit exiting light direction (in the direction of photon propagation, pointing forward to where the light is going).	[sctvar]
$\mathbb{A}$	$\subset \mathbb{R}^3\ \mathrm{m}$	An infinitesmal planar area neighborhood about X.	
P	$\mathbb{R}^3\,\mathrm{m}$	A point on the image plane / the origin of a primary ray.	

Shading

- Problem: How to solve the rendering equation

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) + \int_{\mathbb{S}^2} L_i(X, \hat{\omega}_i) f_r(X, \hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i$$

- Many of the integrals that arise in rendering are **difficult** or **impossible** to evaluate directly

Off-line rendering
(ray/path tracing, photon mapping)

Monte Carlo Integration

Real-time rendering
(game)

Find efficient approximated approach

Implementation

No need to be daunted by this formulation

$$L_o(X, \hat{\omega}_o) = L_e(X, \hat{\omega}_o) + \int_{\mathbb{S}^2} L_i(X, \hat{\omega}_i) f_r(X, \hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i$$

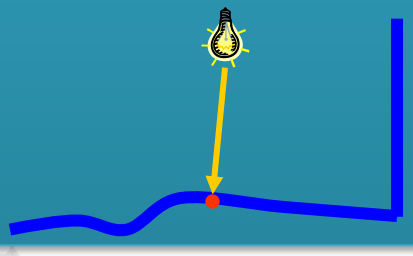
In the upcoming slides, we'll introduce an approximate solution, the Phong model, designed for real-time applications.

```
void main(void)
{
    // Normalize the incoming N, L and V vectors
    // Why we need to do normalization?
    vec3 N = normalize(fs_in.N);
    vec3 L = normalize(fs_in.L);
    vec3 V = normalize(fs_in.V);
    vec3 H = normalize(L + V);

    // Compute the diffuse and specular components for each fragment
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_albedo;
    vec3 specular = pow(max(dot(N, H), 0.0), specular_power) *
specular_albedo;

    // Write final color to the framebuffer
    color = vec4(diffuse + specular, 1.0);
}
```

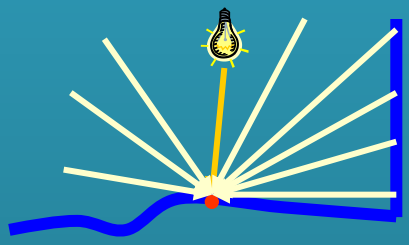
LIGHT-SURFACE INTERACTION

Global Illumination

Direct-only

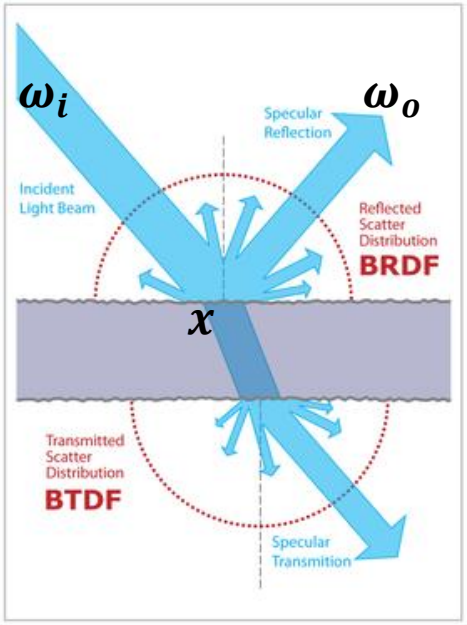


global =
direct +
indirect



Bidirectional Scattering Distribution Function

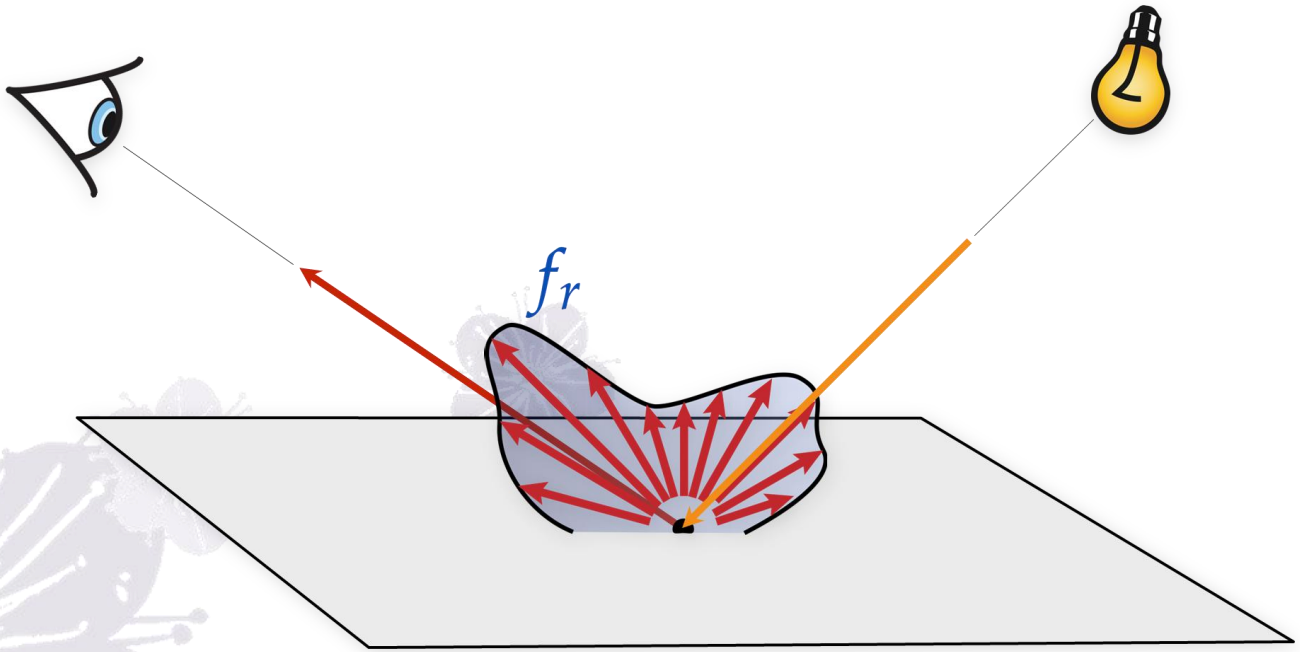
- There are two behaviors when the light is scattered at the object surface.
  - **BRDF (Reflection)** describes how the light is reflected between the incident and outgoing direction.
    - E.g., plastic, metal, …
  - **BTDF (Transmission / Refraction)**
    - E.g., glass, skin, …



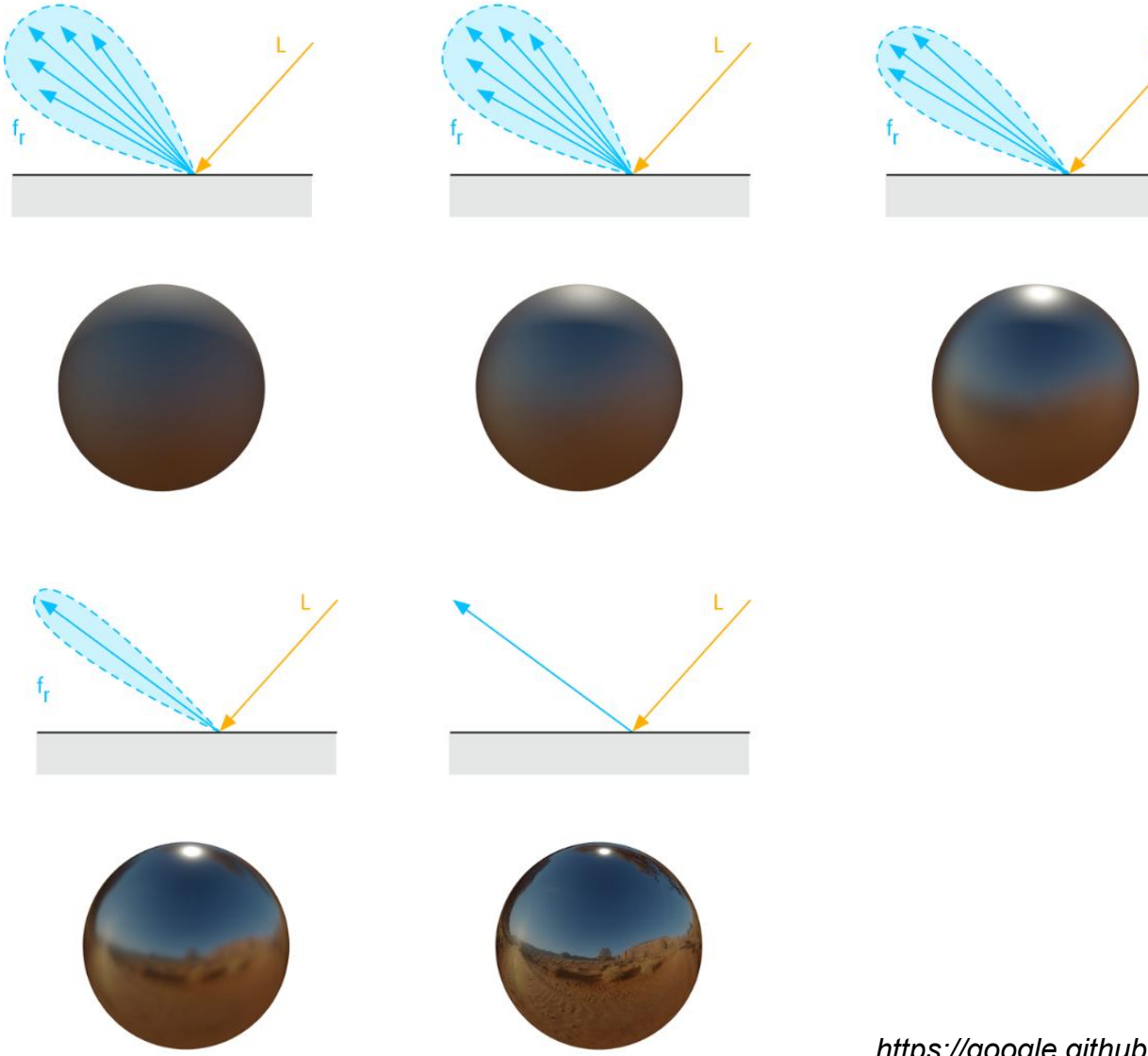
Source image: Wikipedia

Bidirectional Reflectance Distribution Function

- The BRDF represents the reflective property of object surface with certain material.

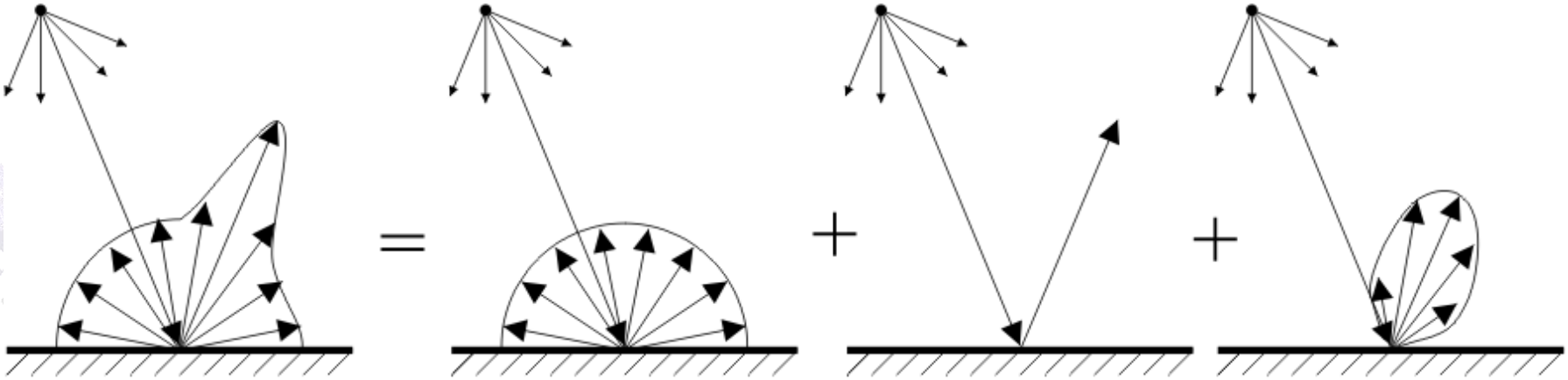


Various BRDFs



https://google.github.io/filament/Filament.html

BRDF Components



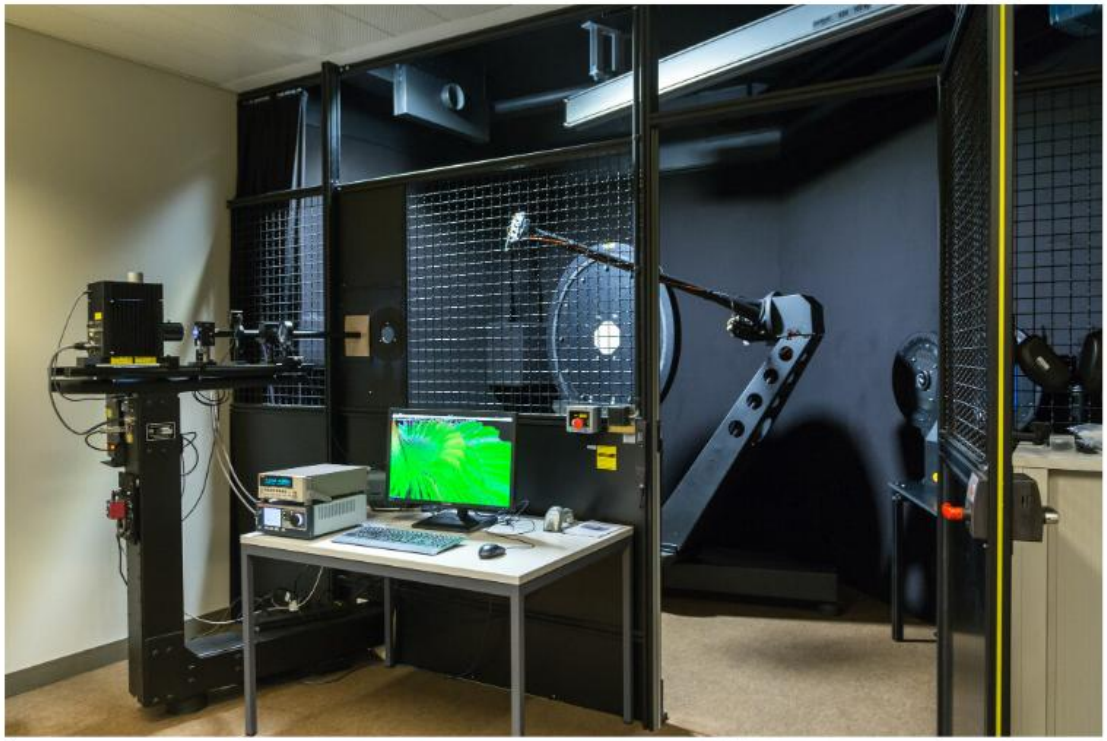
General BRDF

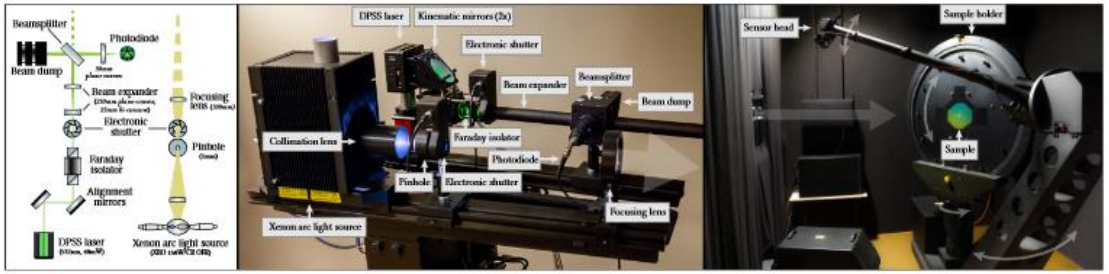
Ideal diffuse (Lambertian)

Ideal specular

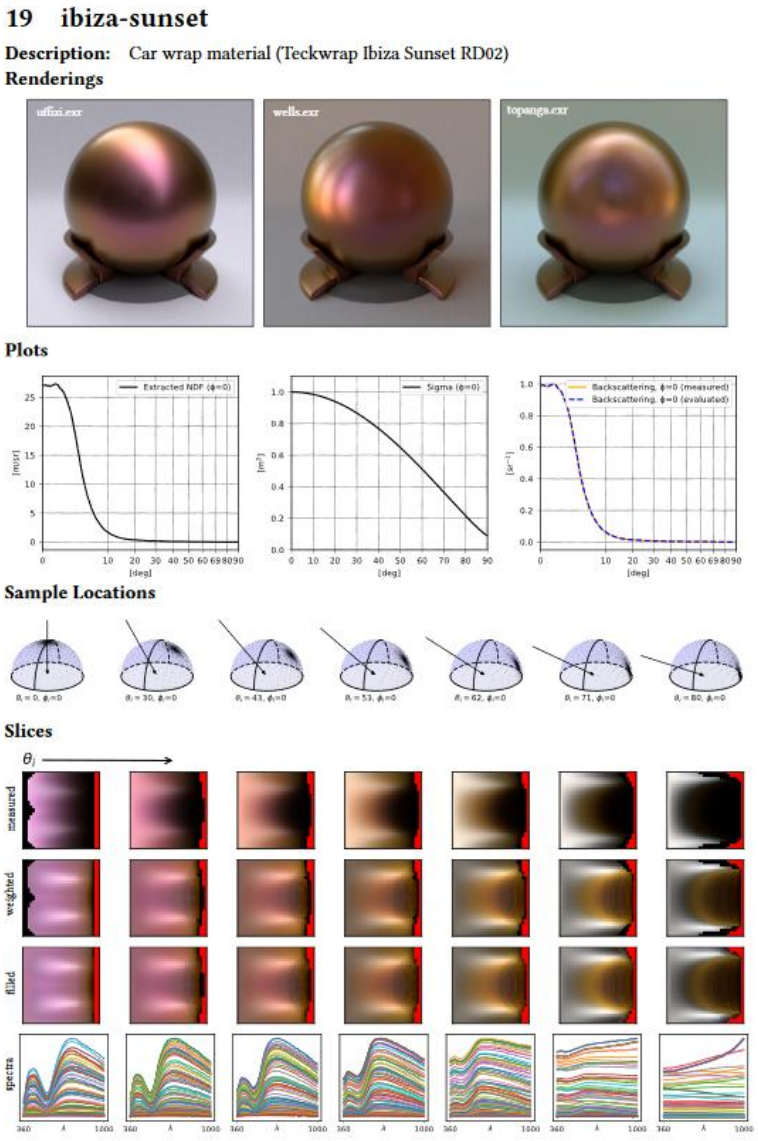
Glossy, directional diffuse

Material Acquisition





Acquisition pipeline
(see https://rgl.epfl.ch/materials)

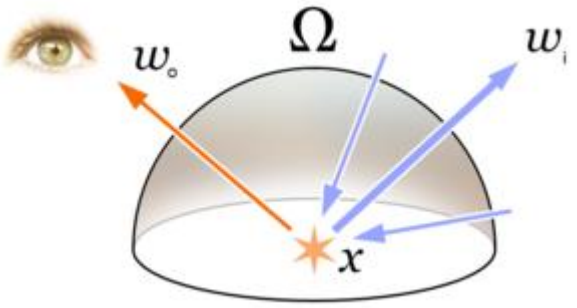


Rendering Equation

- By integrating the hemisphere above the point x, we have

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{2\pi^+} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) cos\theta_i dw_i$$

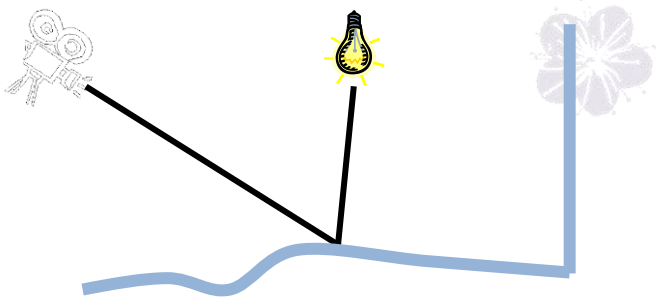
- In physical basis, the rendering equation is the law of conservation of energy
- It means that the outgoing light L_o is the sum of the emitted value L_e and all incident light L_i (recursive term).



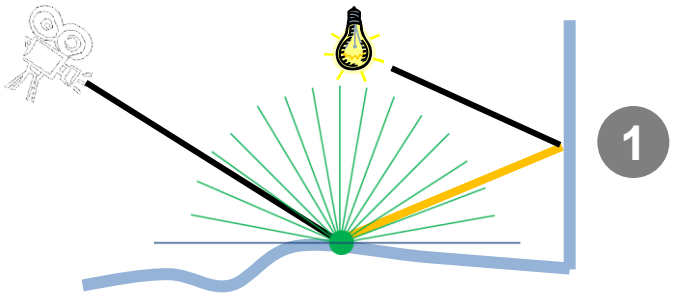
Source image: Wikipedia

Recursive Nature of Light Transport

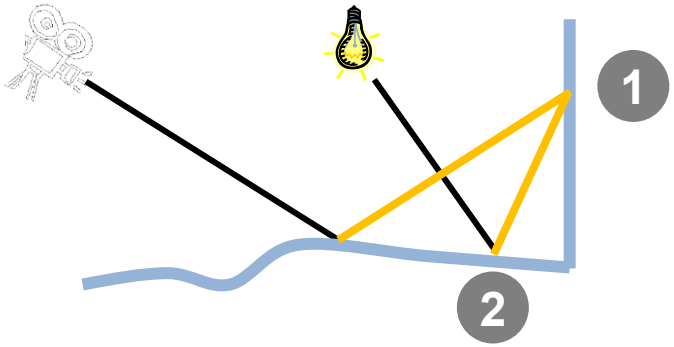
- Direct-only



- 1-bounce indirect

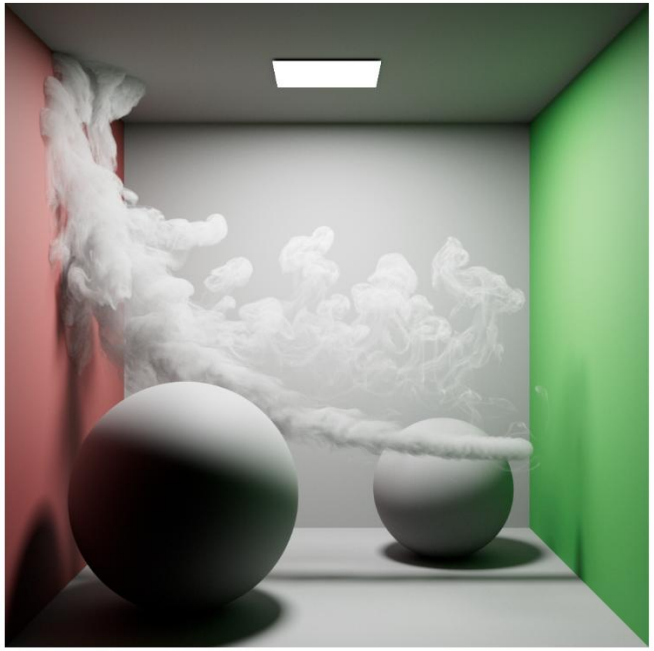


- 2-bounce indirect



Transport Equation

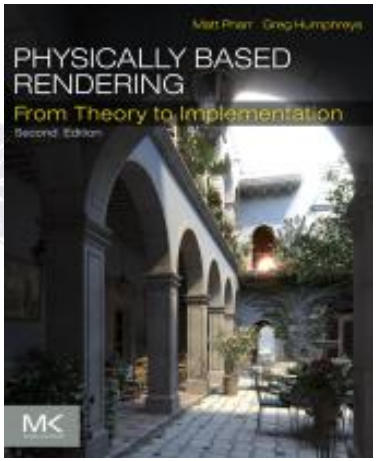
- More generally, in addition to hard surface in rendering equation, there can be participating media in volume rendering.



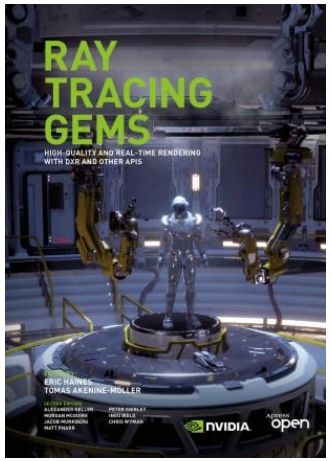
Lighting Models

- ***Simulation of light*** is one of the most important factor of photo-realistic rendering
- Physically accurate lighting models exist, but they are too computationally intensive for real-time GPU implementation
- For example, ***ray tracing algorithms*** produces excellent results, but typically takes minutes or hours to render a scene
  - It might not be true with NVidia RTX graphics card now !

A Ray Traced Scene



PBRT Textbook



Ray Tracing Gems

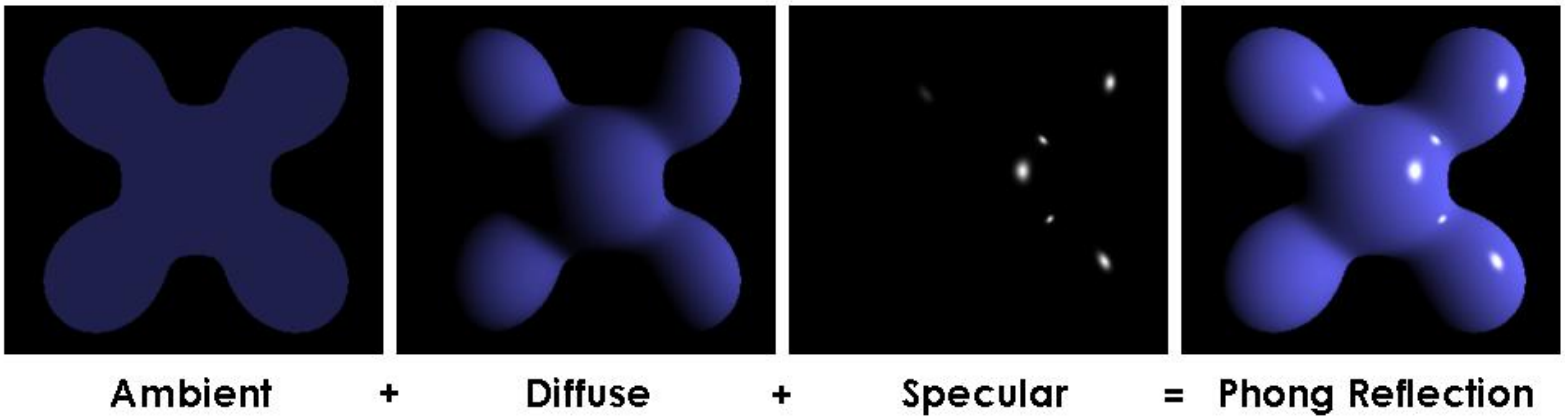
Factors that Affect Shading

- Light sources
  - Ambient light, directional light, positional light, spotlight…
- Material properties (BSDF)
  - Ambient reflection, diffuse reflection, specular reflection…
- Location of the viewer
  - Position of perceiving specular highlight
- Surface orientation
  - Surface normal, vertex normal

Phong Lighting Model

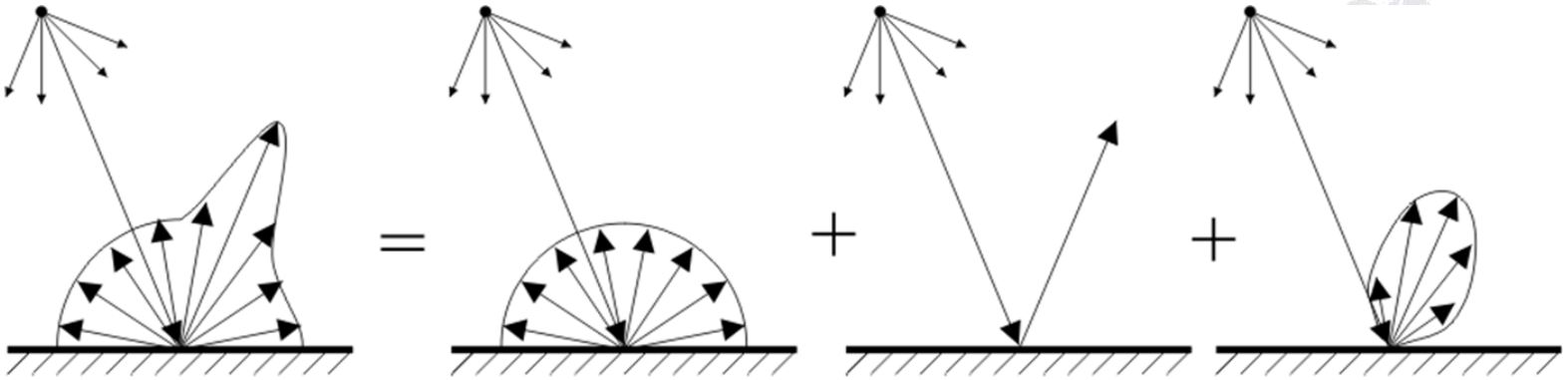
- Named after Bui, Tuong Phong (裴祥風)
- An ***empirical*** lighting model suitable for real-time GPU implementation

- $Ambient + Diffuse + Specular = Intensity$



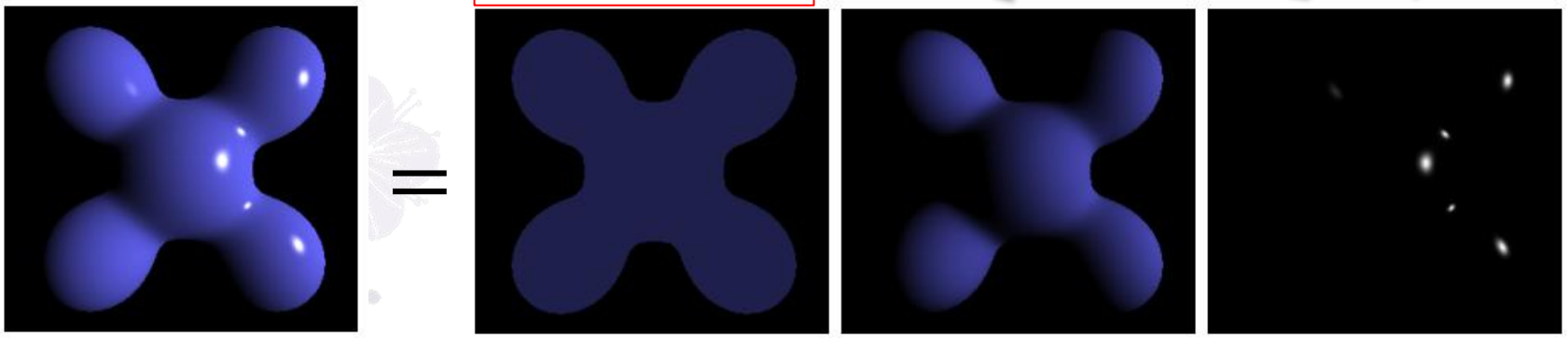
Source image: Max Planck Institute for Informatics – Computer Graphics

Phong Lighting Model



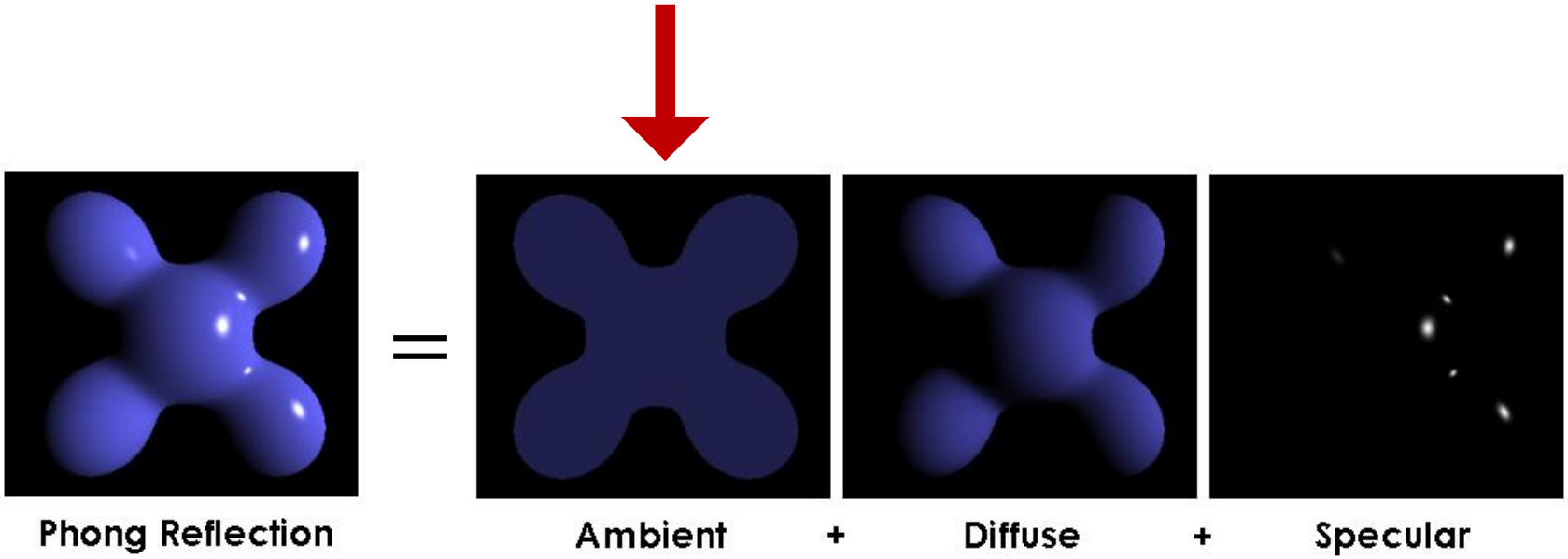
General BRDF = Ideal diffuse (Lambertian) + Ideal specular + Glossy, directional diffuse

*think ambient as indirect lighting



Phong Reflection = Ambient + Diffuse + Specular

Phong Shading Model



Ambient Light

- The mix of light rays come from various sources, and they have bounced around in the scene
- Assumed to be ***directionless*** and ***uniformly scattered*** in the scene to approximate the indirect illumination
- It's like a global "***brightening***" factor applied to the whole scene

Ambient Component

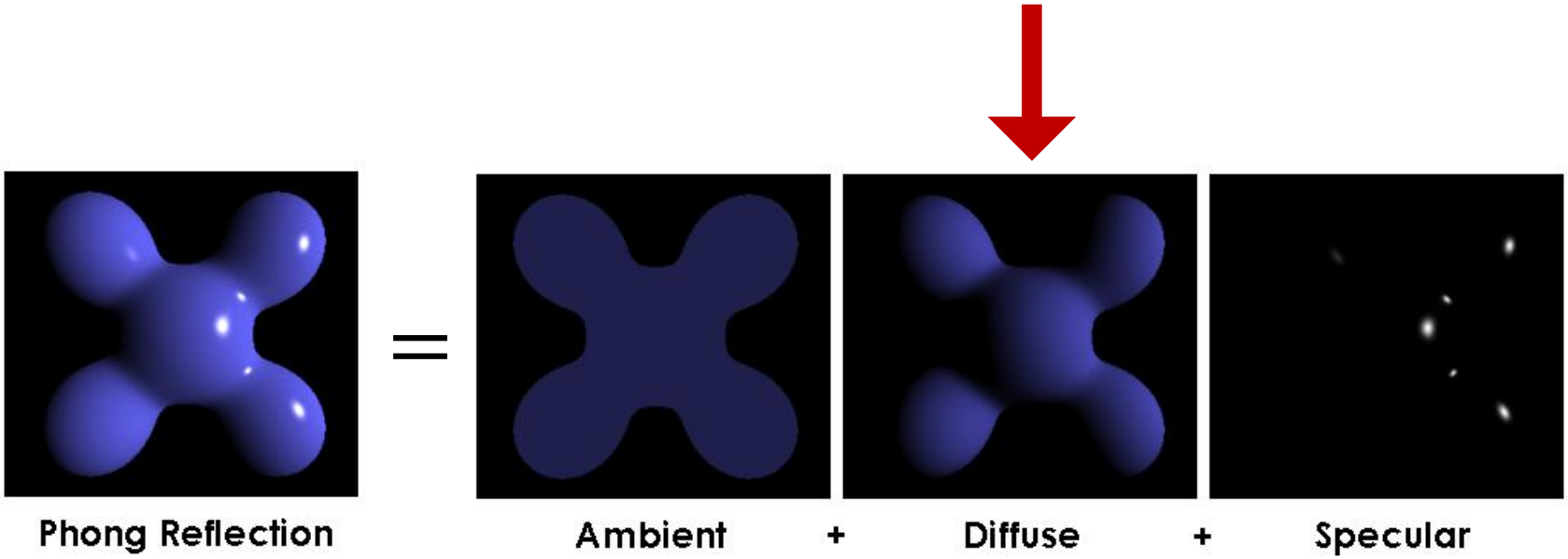
Ambient Light

- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment
- Amount and color depend on both the color of the light(s) and the material properties of the object
- The ambient term is thus defined as:

$$k_a I_a$$

  - k_a: Absorption coefficient
  - I_a: Intensity of ambient light

Phong Shading Model

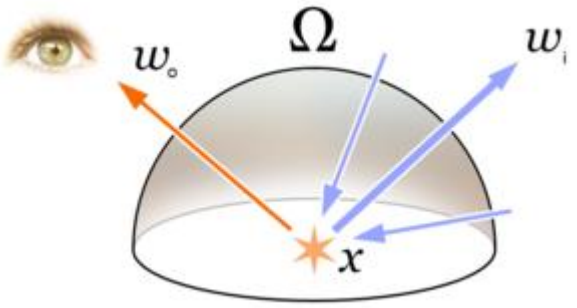


Recap: Rendering Equation

- By integrating the hemisphere above the point x, we have

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{2\pi^+} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) cos\theta_i dw_i$$

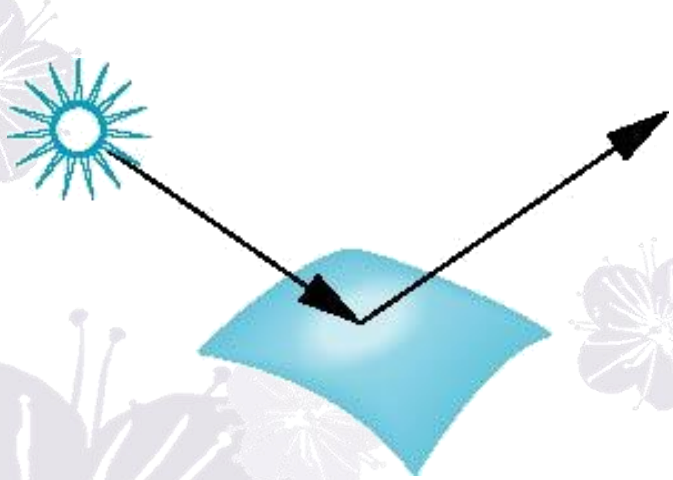
- In physical basis, the rendering equation is the law of conservation of energy
- It means that the outgoing light L_o is the sum of the emitted value L_e and all incident light L_i (recursive term).



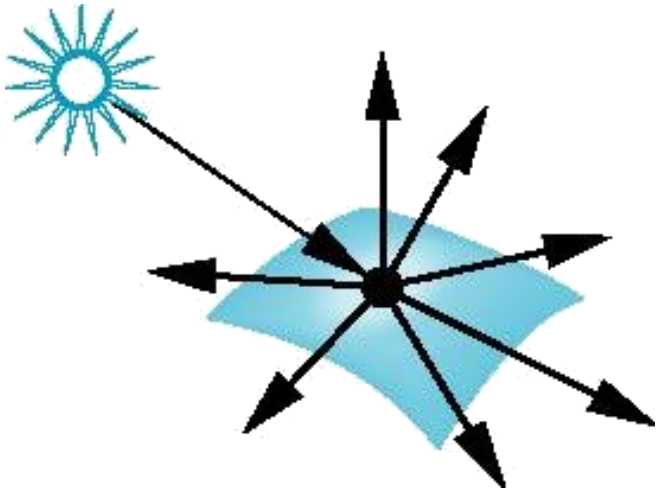
Source image: Wikipedia

Surface Types

- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflect the light
- A rough surface scatters light in all directions



smooth surface



rough surface

Diffuse Light

- The light rays that are ***diffusely reflected*** by the surface. The surface is assumed to be an ***ideal diffuse reflector (Lambertian surface)***
- Obeys ***Lambert's Cosine Law***

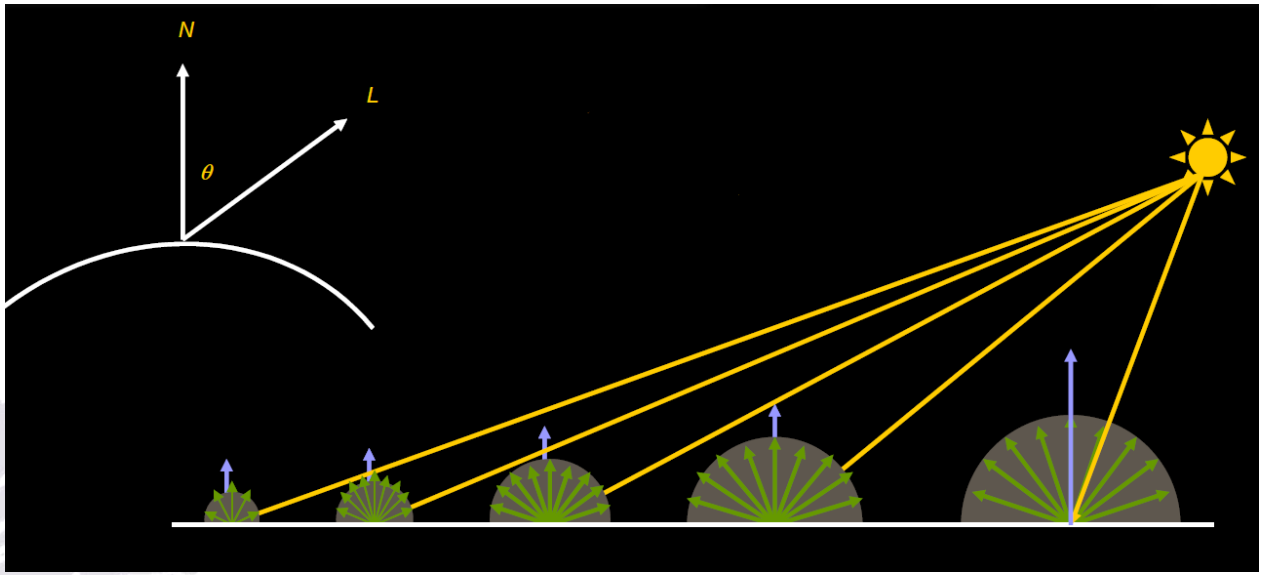
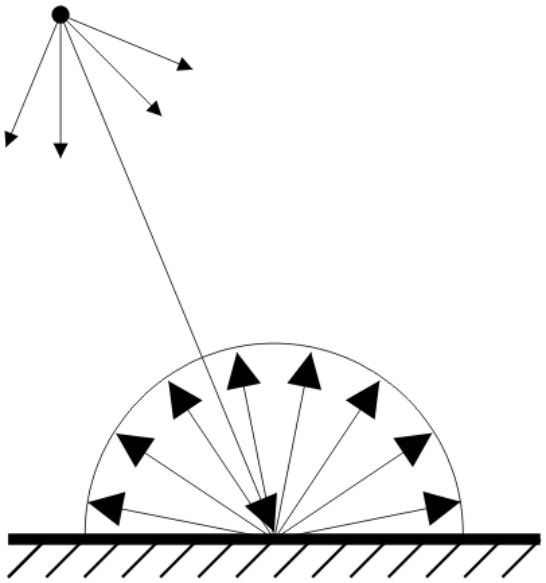


Illustration of Lambert's Cosine Law

Lambertian Surface

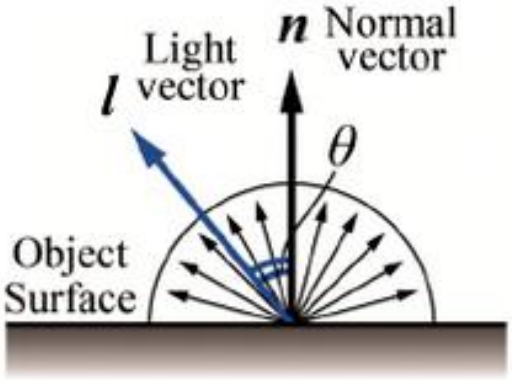
- Ideal diffuse reflector
- Light **scattered equally** in all directions
- Amount of light reflected is proportional to the vertical component of incoming light



Ideal Diffuse Reflection

- For the assumption of Lambertian surface, the reflected radiance $\boldsymbol{L_o}$ is independent of ω_o and the BRDF $\boldsymbol{f_r}$ is independent of ω_i and ω_o.

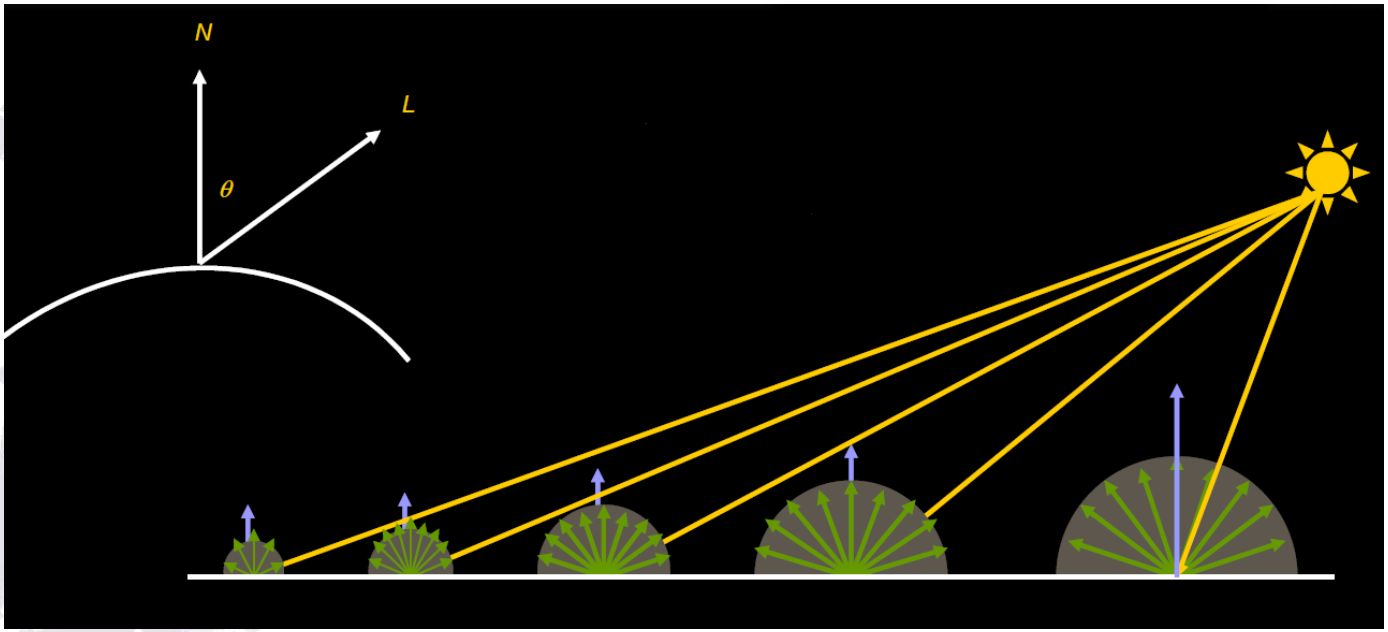
$$f_r(X, \hat{\omega}_i, \hat{\omega}_o) = k_d \text{ (diffuse coefficient)}$$



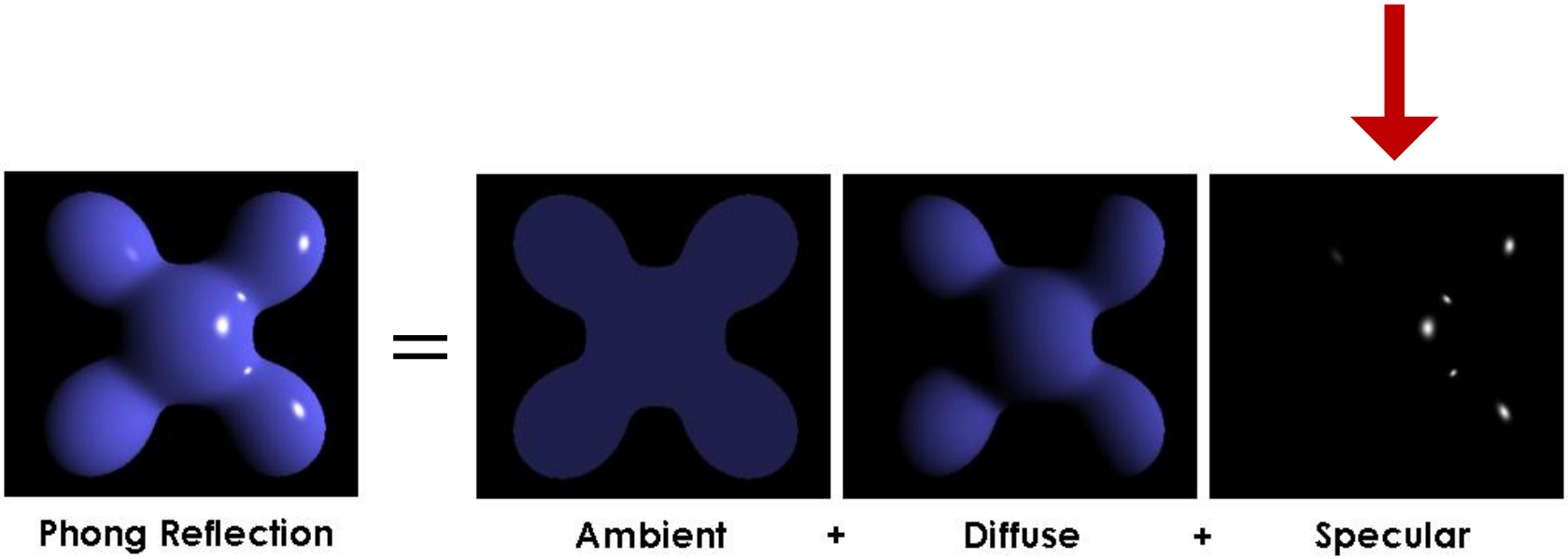
Source image: Shading of a Computer-generated Hologram by Zone Plate Modulation

Phong Model: Diffuse Reflection

- Reflected light $\propto \cos(\theta) = k_d I_d(\vec{L} \cdot \vec{N})$
  - k_d is a RGB coefficient that shows how much of each color component is reflected

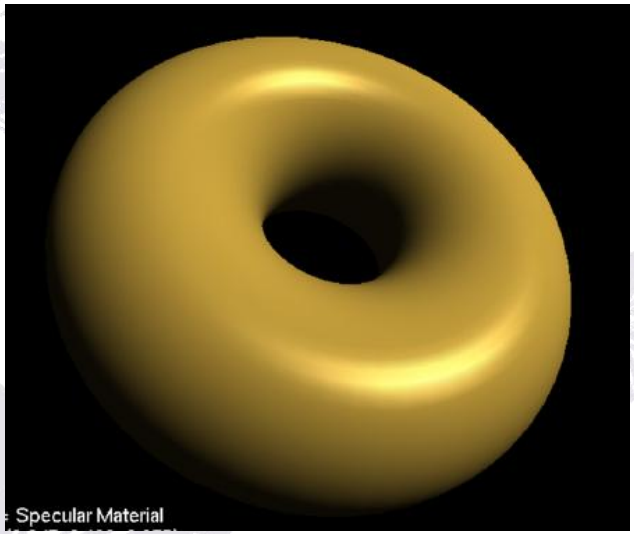


Phong Shading Model



Specular Highlight

- Specular highlight is the bright spot of light on object being illuminated
- ***Mirror-like reflection*** on the surface



Specular Highlight on Torus

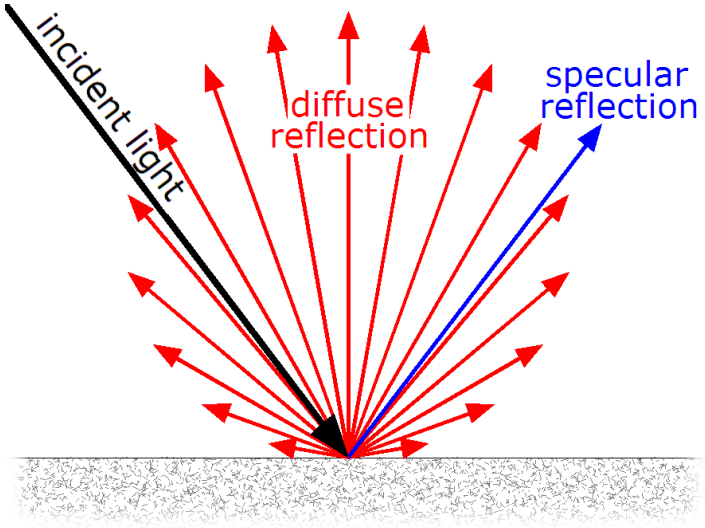
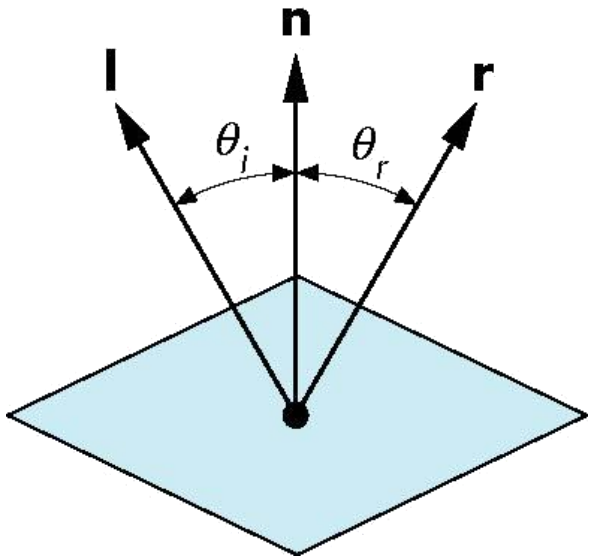


Illustration of Diffuse & Specular Light

Ideal Reflector

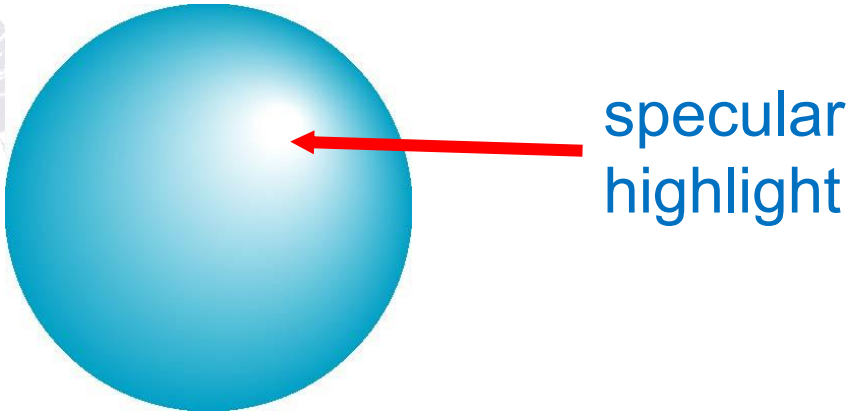
- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

$$\vec{R} = 2\left(\vec{L} \cdot \vec{N}\right)\vec{N} - \vec{L}$$



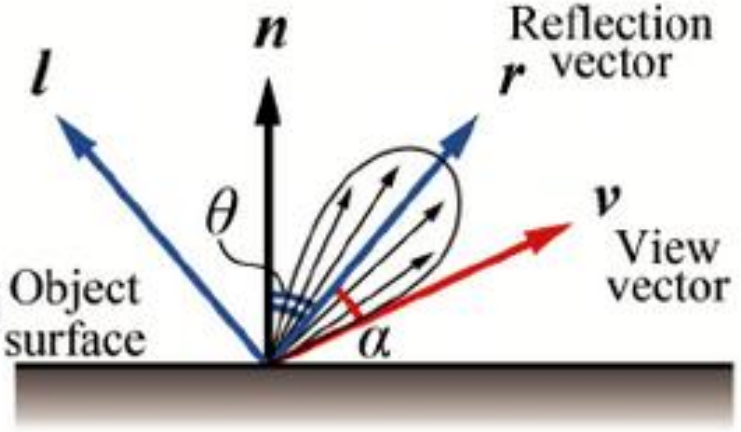
Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection



Ideal Specular Reflection

- Since the glossy specular reflection is concentrated around $\boldsymbol{r}$, the reflected radiance decreases as the angle $\boldsymbol{\alpha}$ between $\boldsymbol{r}$ and $\boldsymbol{\omega_o} = \boldsymbol{v}$ increases.



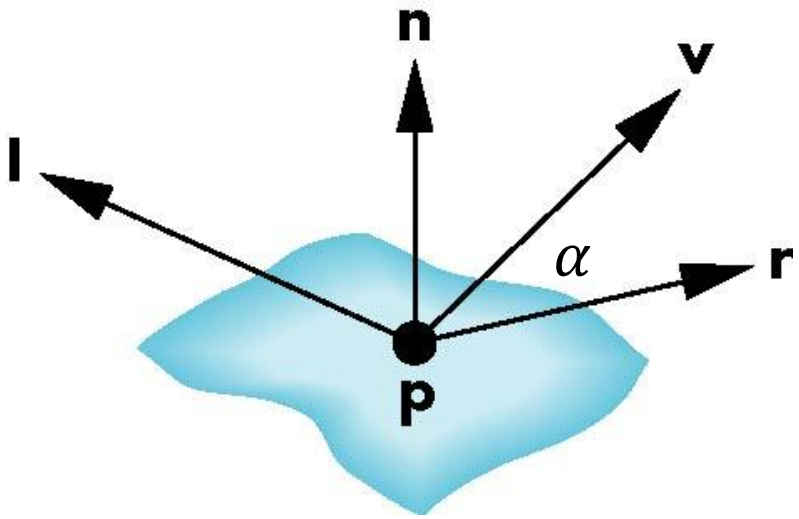
Source image: Shading of a Computer-generated Hologram by Zone Plate Modulation

Phong Model: Specular Reflections

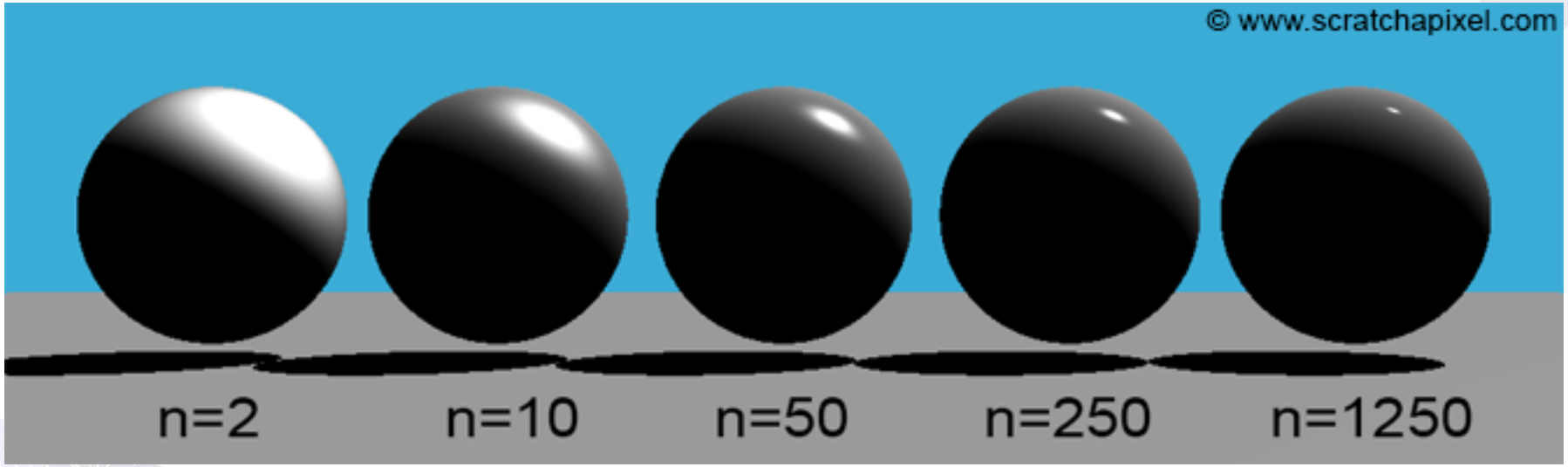
- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

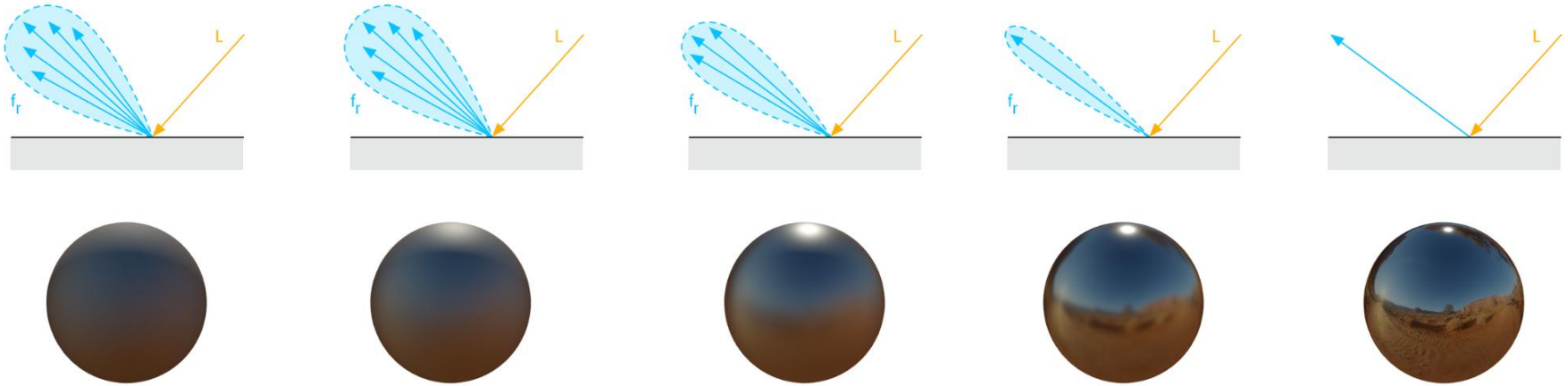
$$I_s \propto k_s I cos^n(\alpha)$$

- I_s: reflected intensity
- k_s: absorption coefficient
- I: incoming intensity
- n : shininess coefficient



Shininess



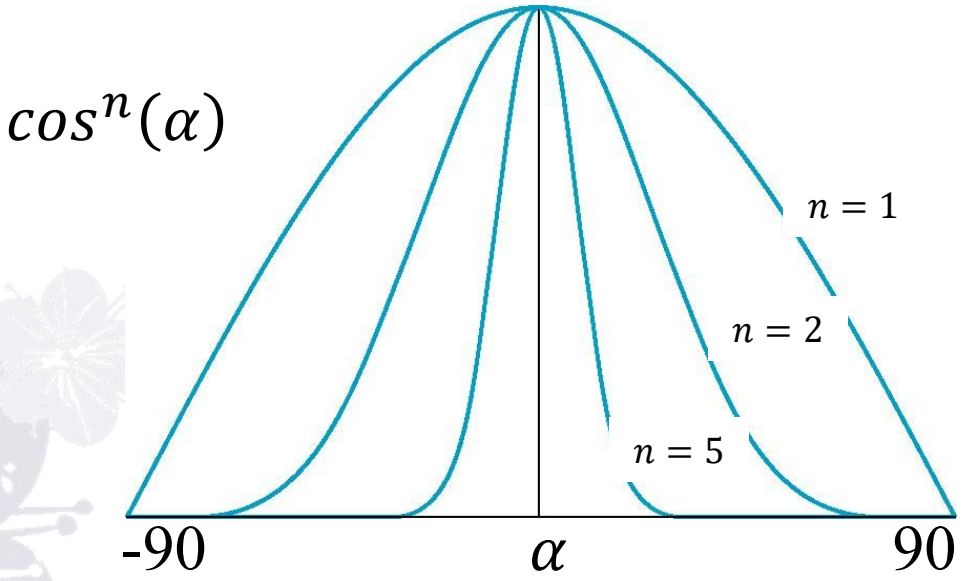


https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF

https://google.github.io/filament/Filament.html

Shininess Coefficient

- Values of n between 100 and 200 correspond to metals
- Values between 5 and 10 make surface look like plastic



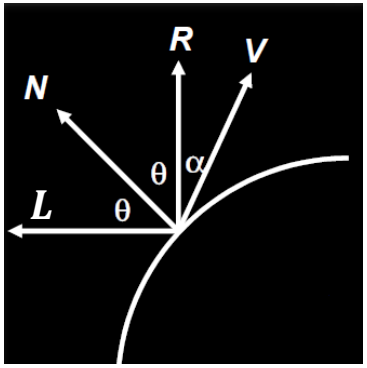
Phong Lighting Model

- Let M be the material of an object, such that:
- k_a: Ambient coefficient of M
- k_d: Diffuse coefficient of M
- k_s: Specular coefficient of M
- n: Shininess coefficient of M

- Let I be the light source intensity, such that:
- I_a: Intensity of ambient light
- I_d: Intensity of diffuse light
- I_s: Intensity of specular light
- Note: I_a, I_d, I_s may be the same depend on the lighting context

Phong Lighting Model

- Let $\vec{V}, \vec{N}, \vec{L}, \vec{R}$ be directional vectors ***in world or eye space***, such that:
- $\vec{V}$: **Normalized** viewpoint direction vector
- $\vec{N}$: **Normalized** surface normal
- $\vec{L}$: **Normalized** light vector
- $\vec{R}$: **Normalized** reflection vector



- The final illumination intensity of a point I_p is:
- $I_p = k_a I_a + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{R} \cdot \vec{V}\right)^n$

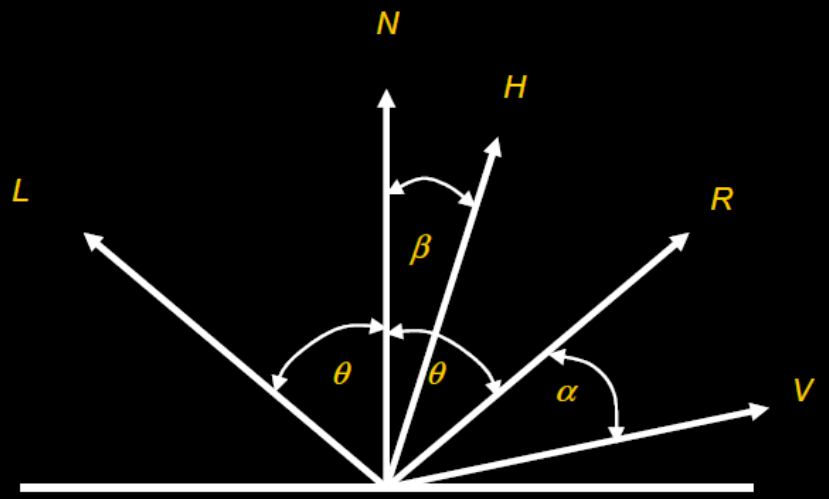
Blinn-Phong Lighting

- Reflection vector $\vec{R} = 2\left(\vec{L} \cdot \vec{N}\right)\vec{N} - \vec{L}$ can be **approximated** by halfway vector $\vec{H}$, which is computationally simpler

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

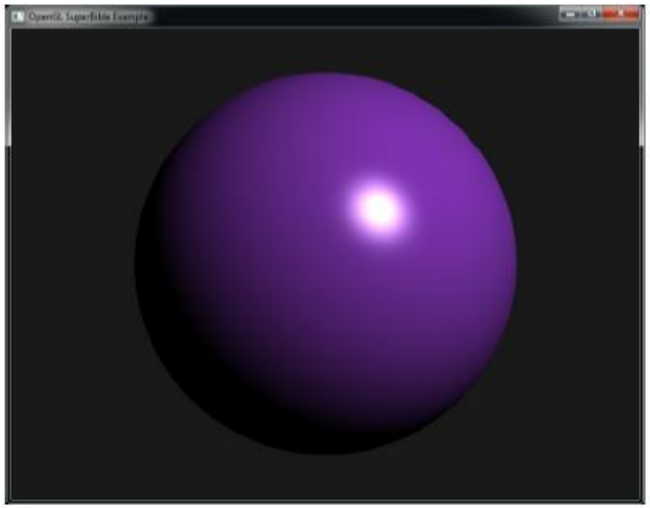
$$\theta + \beta = \theta - \beta + \alpha$$

$$\beta = \frac{1}{2}\alpha$$

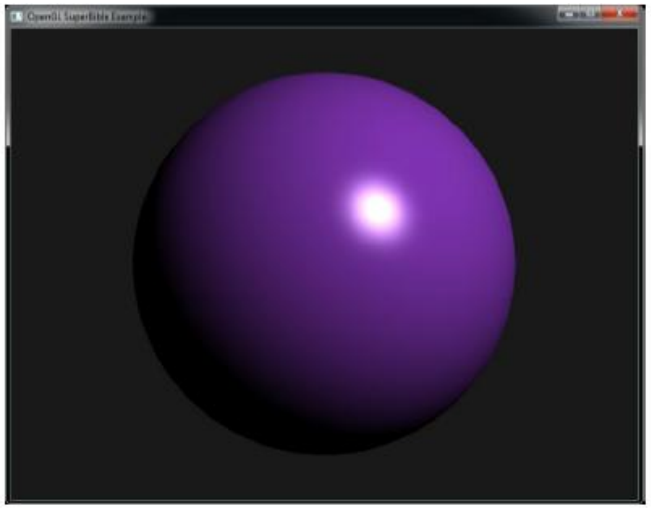


Blinn-Phong Lighting

- Replace $\vec{R} \cdot \vec{V}$ with $\vec{H} \cdot \vec{N}$
- n should also be modified because $\beta = \frac{\alpha}{2}$
- $n' > n$, for example, $n = 128$ and $n' = 200$
- $I_p = k_a I_a + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{H} \cdot \vec{N}\right)^{n'}$

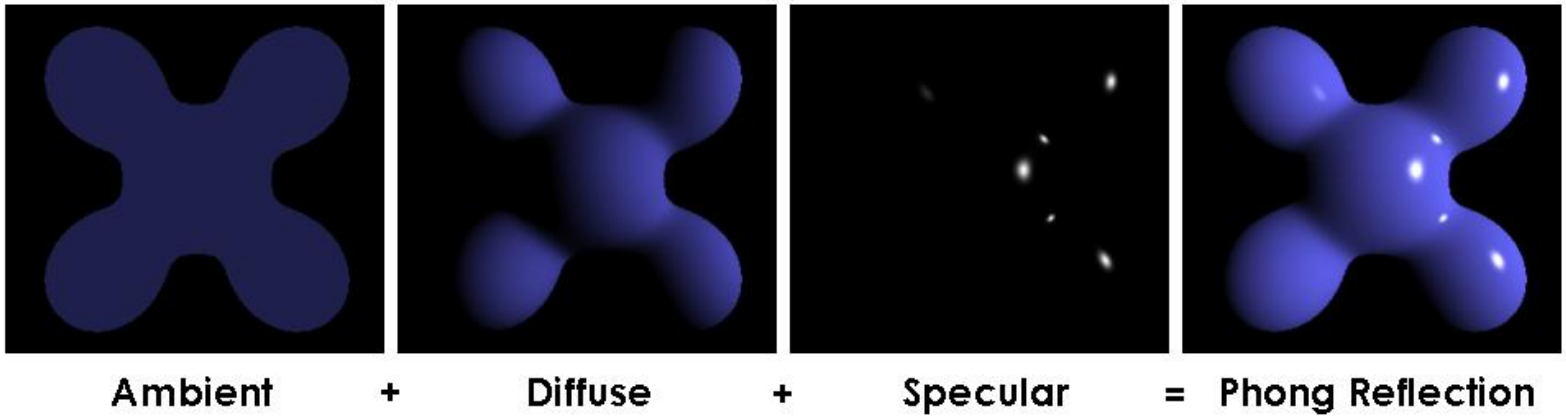


Phong Lighting



Blinn-Phong Lighting

Is the Phong model physically correct?

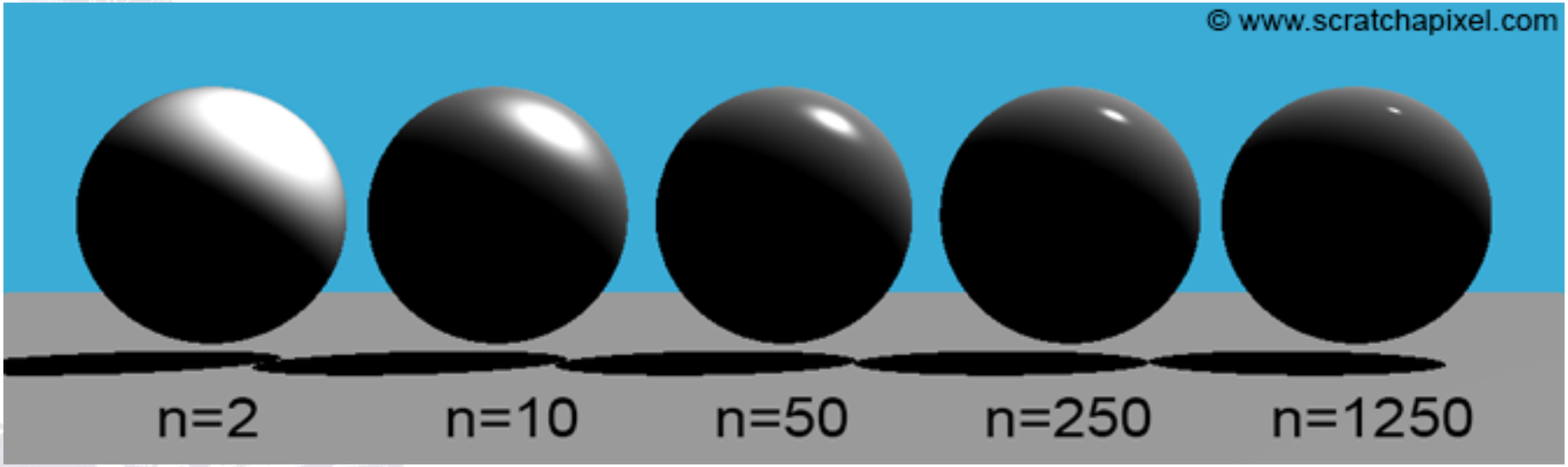


$$k_a I_a + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{R} \cdot \vec{V}\right)^n = I_p$$

NO, not energy conserving

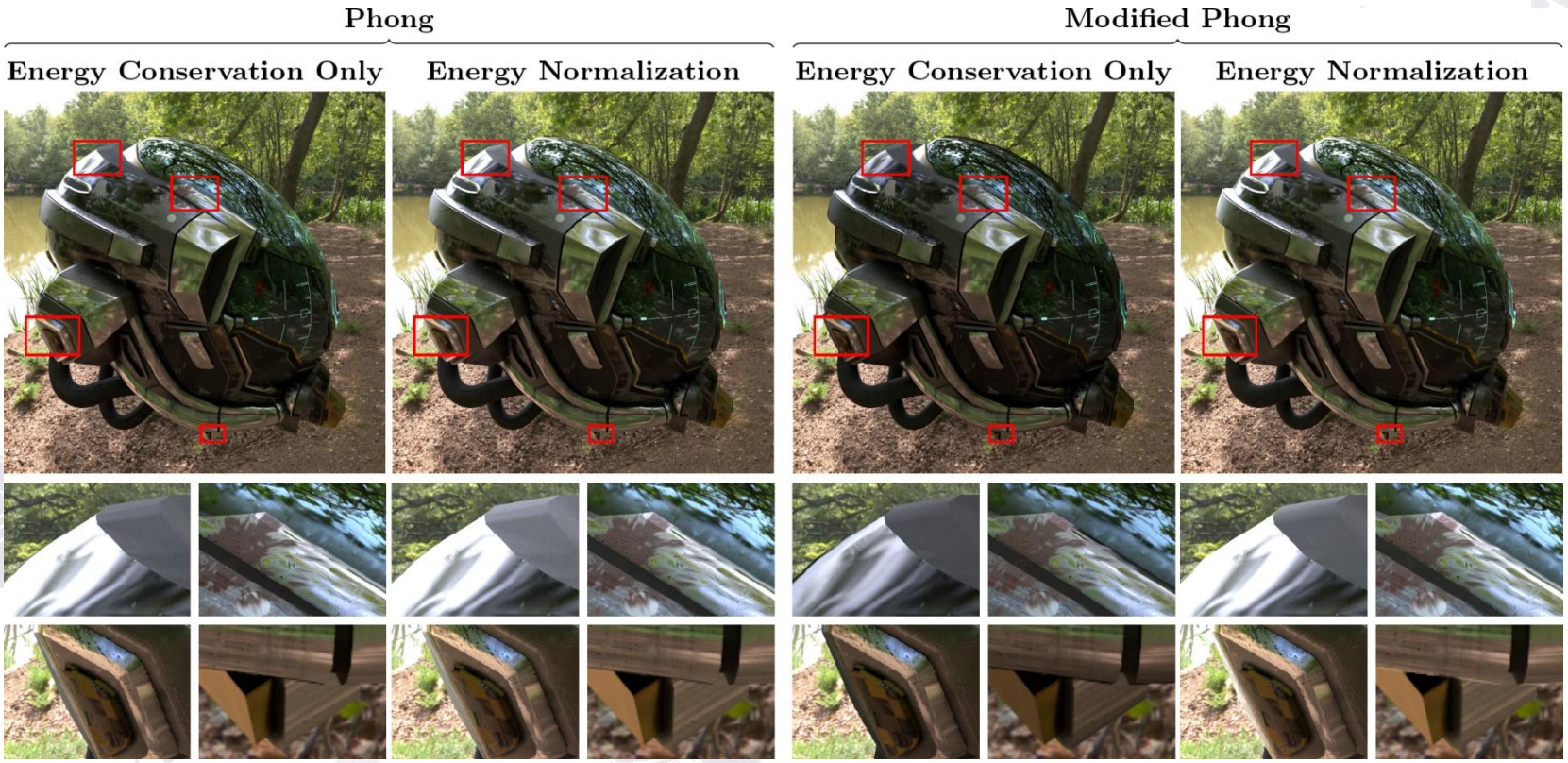
GGX PBR roughness (follow energy conservation principle)

Phong lighting model shininess (not energy conservation)



https://www.rorydriscoll.com/2009/01/25/energy-conservation-in-games/

Further Reading



Abstract: Energy normalization (that is, never lose nor gain energy) makes the Phong BRDFs more physically plausible and therefore both more practically and theoretically useful.

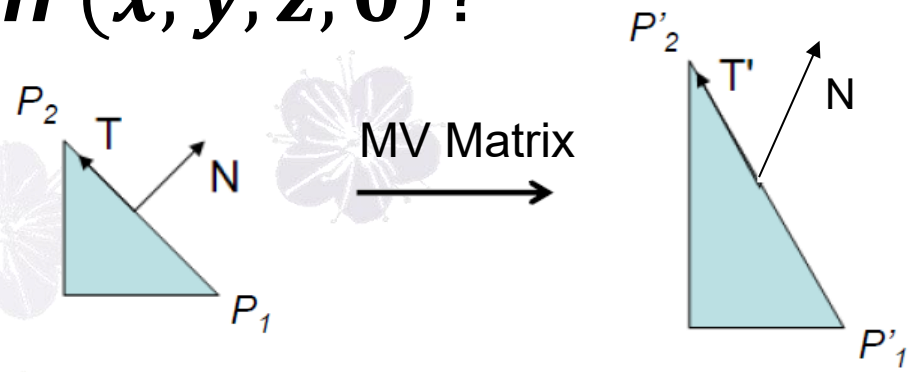
https://graphics.geometrian.com/research/normalize_phong.html

Further Reading - Physically Based Rendering

- How to render the scene photo-realistic → mimicking real-life lighting behavior

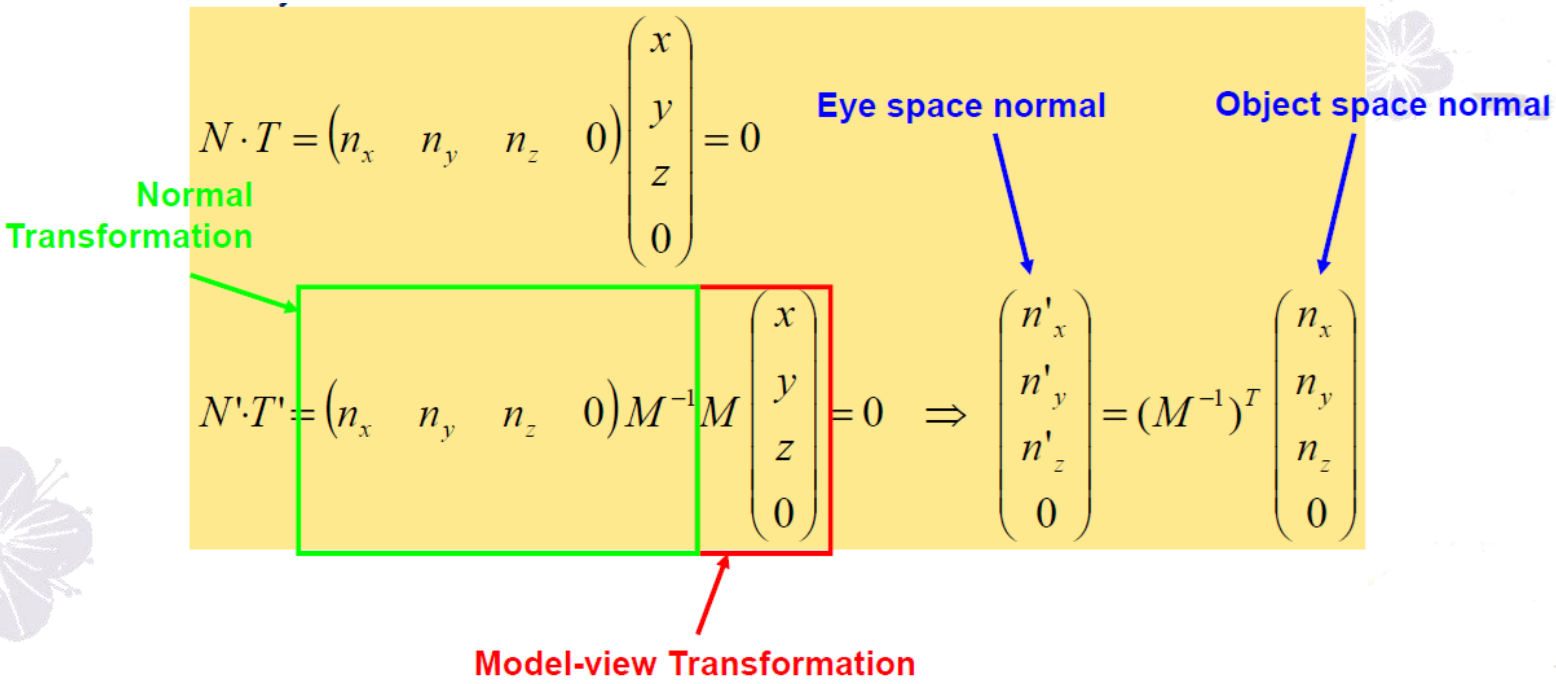
Normal Transformation

- We know that a ***point*** $(x, y, z, 1)$ can be transformed into world space, eye space and clip space by applying model, view and projection matrices
- Can we do the same transformation to a ***direction*** $(x, y, z, 0)$?



Normal Direction Is Incorrect After Transformation

Normal Transformation



- $N' = M' * N$
- $M' = (M^{-1})^T = transpose\big(inverse(M)\big)$
- Transpose inverse is not cheap, don't do this in fragment shader

Ref: https://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/

Recap: Matrix Inverse

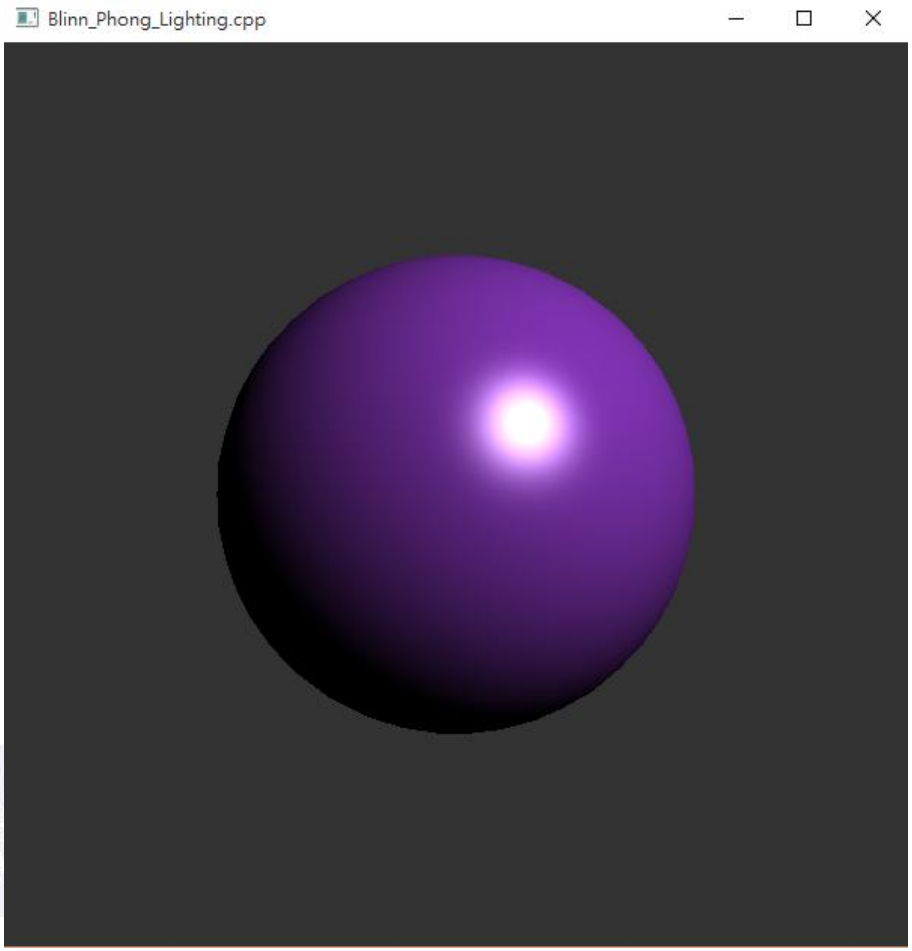
- Although the inverse of matrices can be computed using general methods:
  - Gaussian elimination, or LU decomposition
- We can exploit simple geometric observations:
  - Translation: $T^{-1}(T_x, T_y, T_z) = T(-T_x, -T_y, -T_z)$
  - **Rotation:** $\boldsymbol{R^{-1}(R, \theta) = R(R, -\theta) = R^T(R, \theta)}$
    - **Holds for any rotation matrix**
    - **Note that $\cos(\theta) = -\cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$**
  - Scaling: $S^{-1}(S_x, S_y, S_z) = S\left(\frac{1}{S_x}, \frac{1}{S_y}, \frac{1}{S_z}\right)$

Eye Space vs World Space

- **OpenGL** traditionally use ***eye space*** for lighting, which can be beneficial because all normal vectors will face the camera
- **CryEngine** and **Unreal Engine** perform lighting calculation in ***world space***, which benefits post-processing effects
- Which is better?
- It depends on your rendering pipeline; It's OK as long as you do them…

in the same space!!!

Blinn-Phong Shading



Code: Render Function

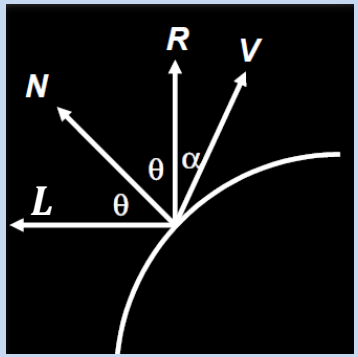
```
GLuint blinn_program;
GLuint vao;
int index_count;
mat4 proj_matrix;
GLuint uniforms_buffer;

struct uniforms_block
{
    mat4 mv_matrix;
    mat4 view_matrix;
    mat4 proj_matrix;
};

struct
{
    GLint diffuse_albedo;
    GLint specular_albedo;
    GLint specular_power;
} uniform;
```

Code: Vertex Shader

```
#version 410 core
// Per-vertex inputs
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
// Matrices we'll need
layout (std140) uniform constants
{
    mat4 mv_matrix;
    mat4 view_matrix;
    mat4 proj_matrix;
};
// Inputs from vertex shader
out VS_OUT
{
    vec3 N;
    vec3 L;
    vec3 V;
} vs_out;
// Position of light
uniform vec3 light_pos = vec3(100.0, 100.0, 100.0);
```

Code: Vertex Shader (Cont'd)

```
void main(void)
{
    // Calculate view-space coordinate
    vec4 P = mv_matrix * vec4(position, 1.0);

    // Calculate normal in view-space
    // Using only the rotational components of mv_matrix
    vs_out.N = mat3(mv_matrix) * normal;

    // Calculate light vector
    vs_out.L = light_pos - P.xyz;

    // Calculate view vector
    vs_out.V = -P.xyz;

    // Calculate the clip-space position of each vertex
    gl_Position = proj_matrix * P;
}
```



Code: Fragment Shader

```
#version 410 core

// Output
layout (location = 0) out vec4 color;

// Input from vertex shader
in VS_OUT
{
    vec3 N;
    vec3 L;
    vec3 V;
} fs_in;

// Material properties
uniform vec3 diffuse_albedo = vec3(0.5, 0.2, 0.7);
uniform vec3 specular_albedo = vec3(0.7);
uniform float shininess_power = 200.0;
```

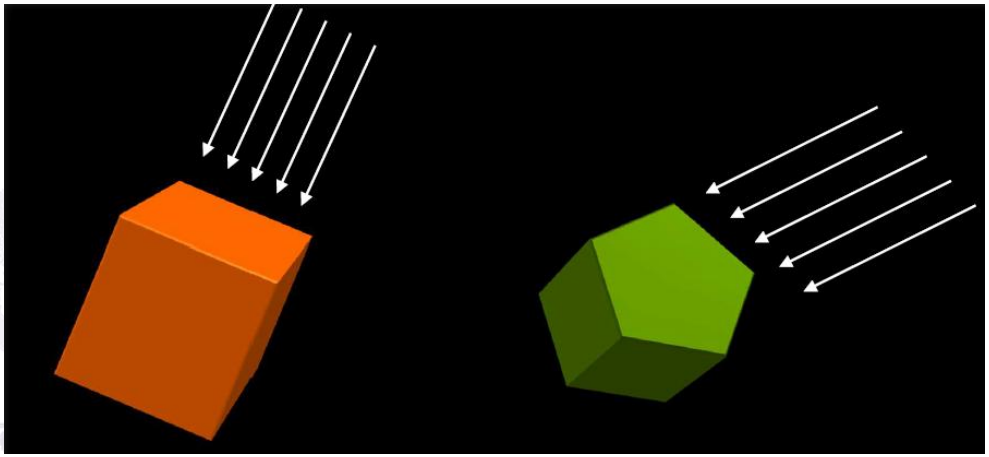
Code: Fragment Shader (Cont'd)

```
void main(void)
{
    // Normalize the incoming N, L and V vectors
    // Why we need to do normalization?
    vec3 N = normalize(fs_in.N);
    vec3 L = normalize(fs_in.L);
    vec3 V = normalize(fs_in.V);
    vec3 H = normalize(L + V);

    // Compute the diffuse and specular components for each fragment
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_albedo;
    vec3 specular = pow(max(dot(N, H), 0.0), shininess_power) *
specular_albedo;

    // Write final color to the framebuffer
    color = vec4(diffuse + specular, 1.0);
}
```

$$I_p = \cancel{k_a I_a} + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{H} \cdot \vec{N}\right)^{n'}$$

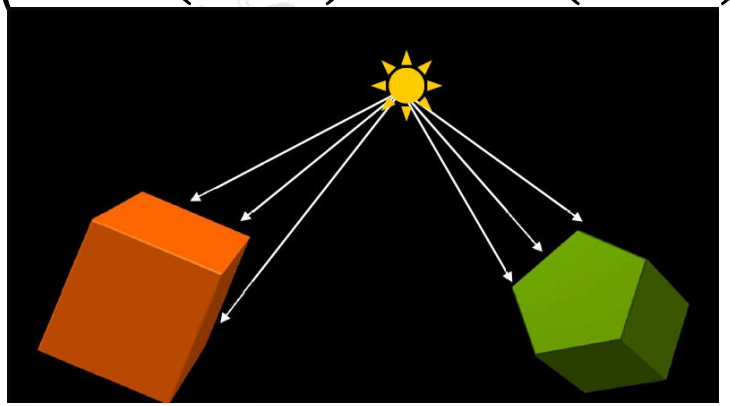
Directional Light

- Light source located at infinite distance, such as sun
- $\vec{L}$ is a fixed vector
- $I_p = k_a I_a + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{H} \cdot \vec{V}\right)^{n'}$



Point Light

- Light source located at a specific position, such as bulb or fire
- ***Attenuation factor*** f_a should be calculated and applied
- $f_a = \min\left(\frac{1}{c_1 + c_2|\vec{L}| + c_3|\vec{L}|^2}, 1\right)$, $\vec{L}$ is un-normalized light vector
- $I_p = k_a I_a + f_a\left(k_d I_d\left(\vec{L}\cdot\vec{N}\right) + k_s I_s\left(\vec{H}\cdot\vec{V}\right)^{n'}\right)$

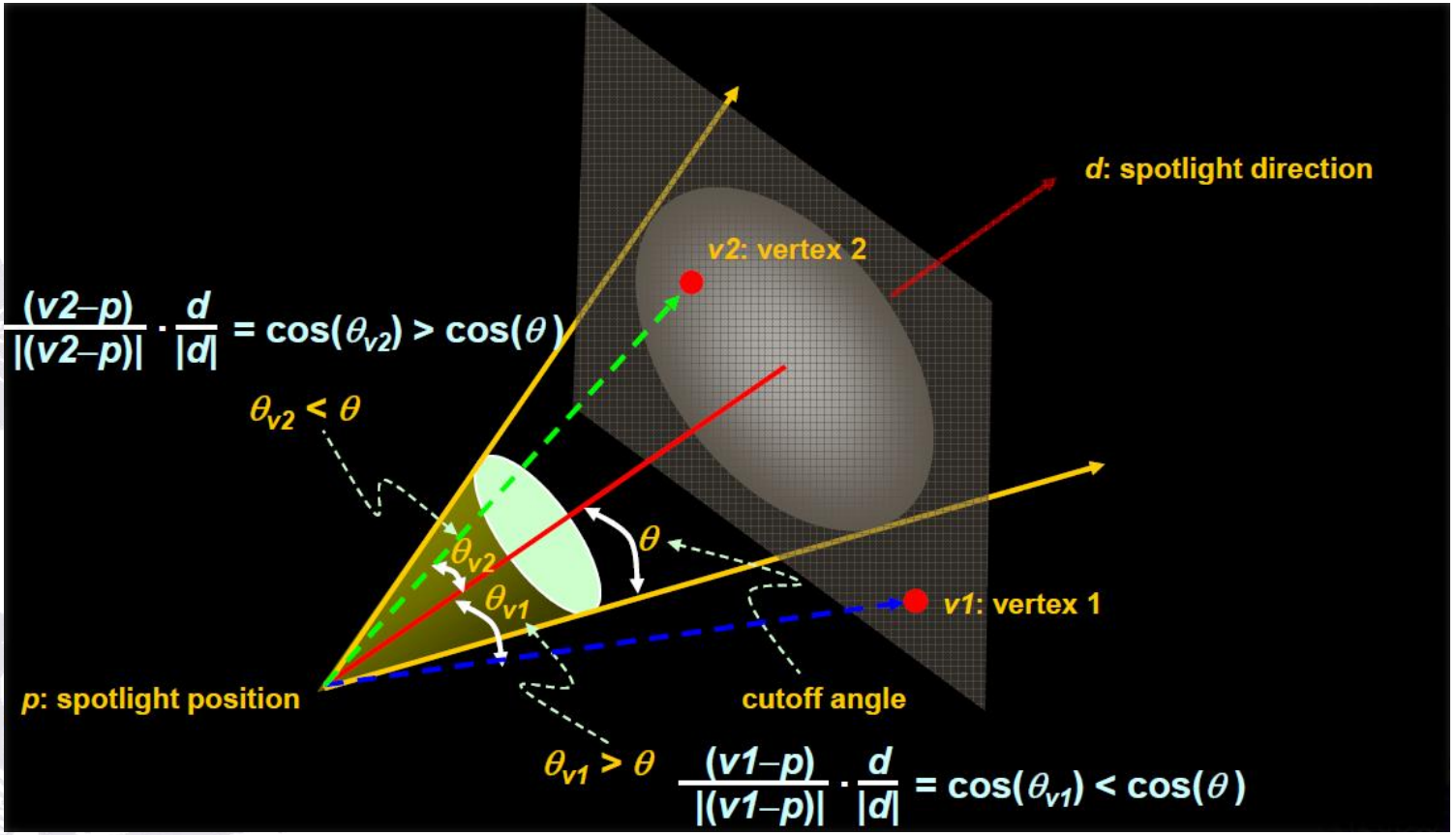


Distance Attenuation

- The light strength from a point source that reaches a surface is inversely proportional to the square of the distance between them
- The constant and linear terms soften the effect of the point source

Spotlight

- Special case of point light

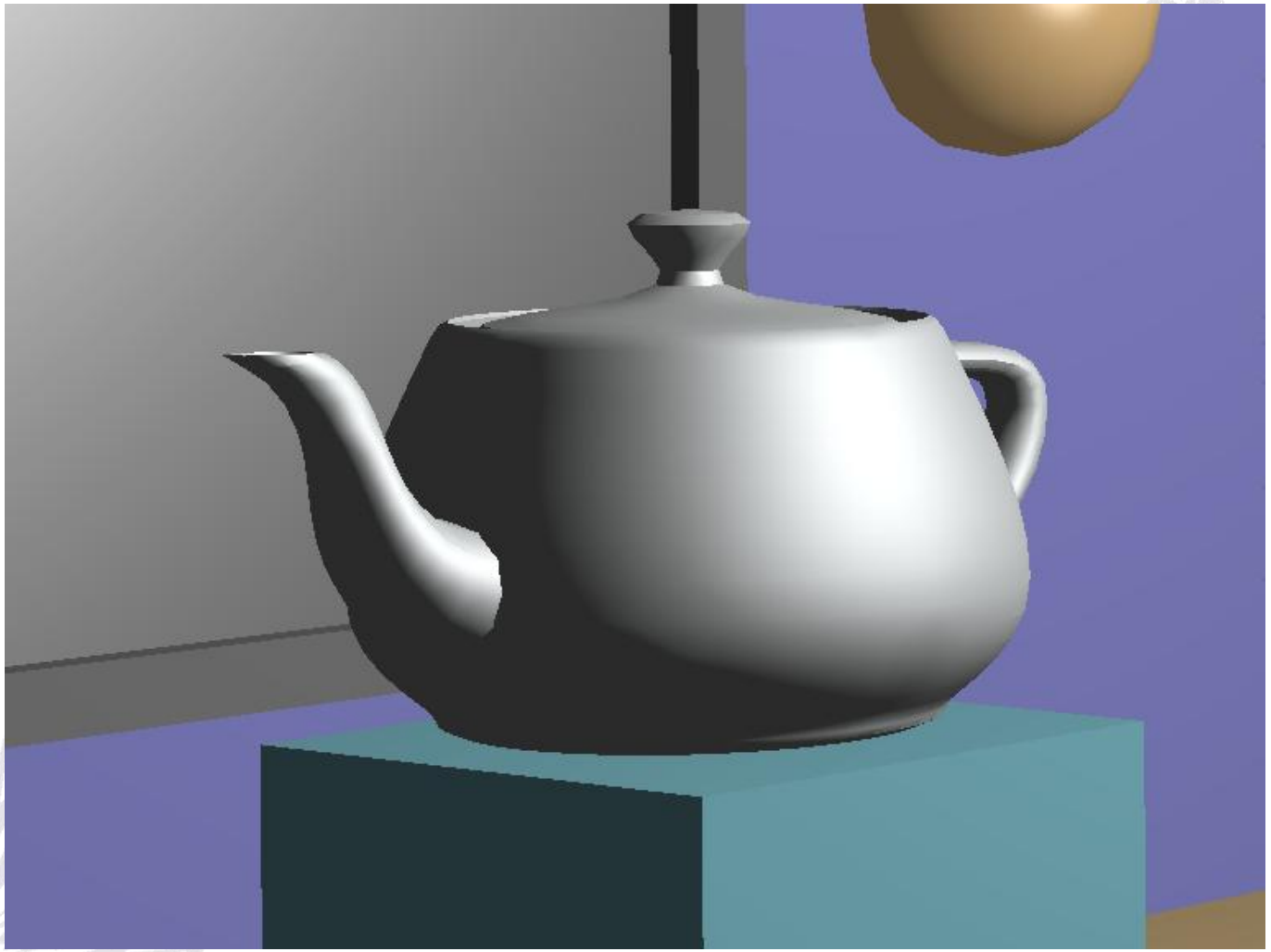


Spotlight

- Spotlight Effect
  - 1, if the light source is not a spotlight
  - 0, if the light source is a spotlight but the vertex lies outside the cone of illumination produced by the spotlight
  - $\left(\max\left(\vec{V}\cdot\vec{D},0\right)\right)^{spot\_exp}$, otherwise.
    - $\vec{V}$ is the unit vector from the spotlight to the vertex
    - $\vec{D}$ is the spotlight direction

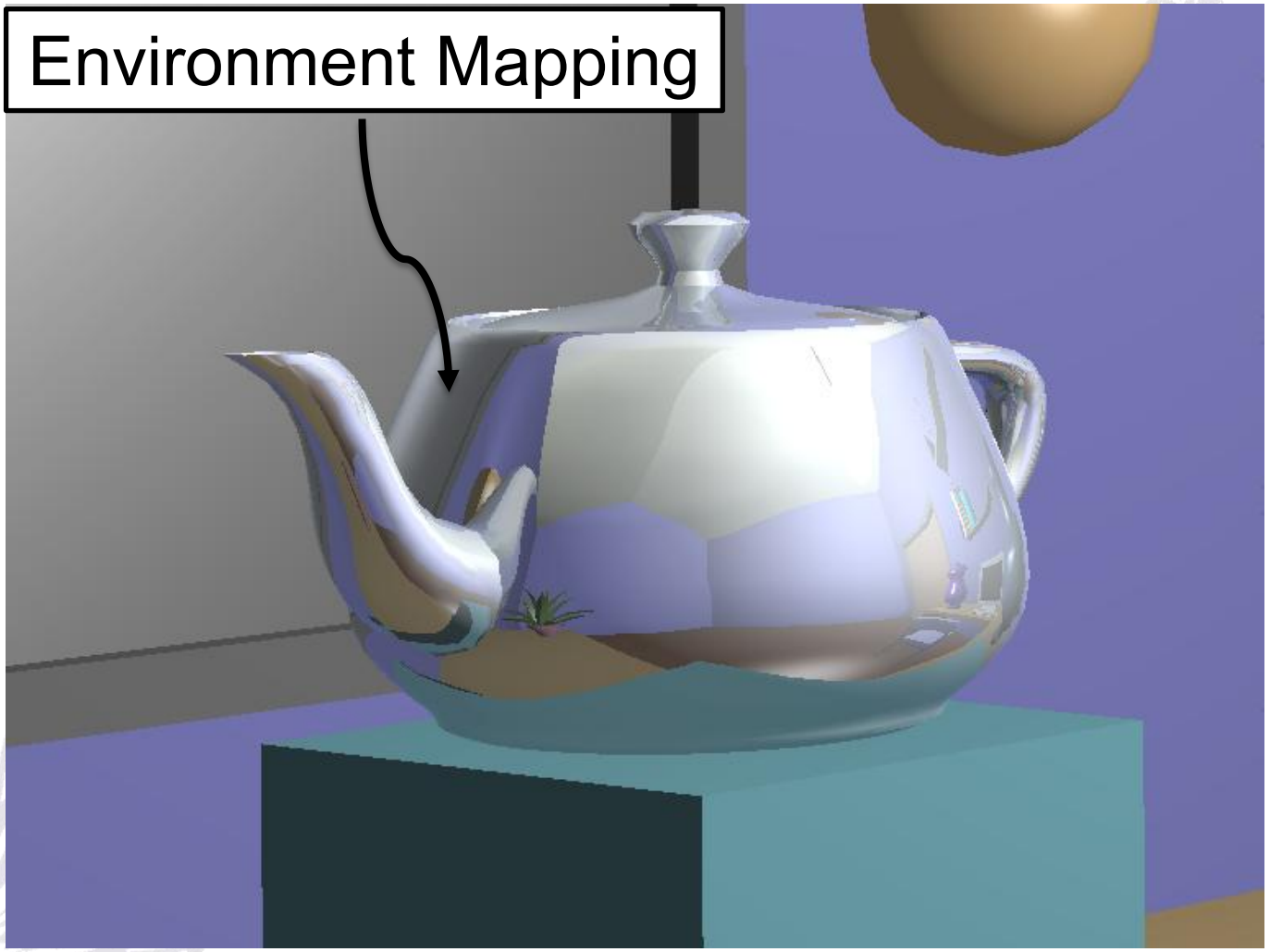
$$I_p = k_a I_a + f_{spot} * f_a\left(k_d I_d\left(\vec{L}\cdot\vec{N}\right) + k_s I_s\left(\vec{H}\cdot\vec{V}\right)^{n'}\right)$$

Aluminium

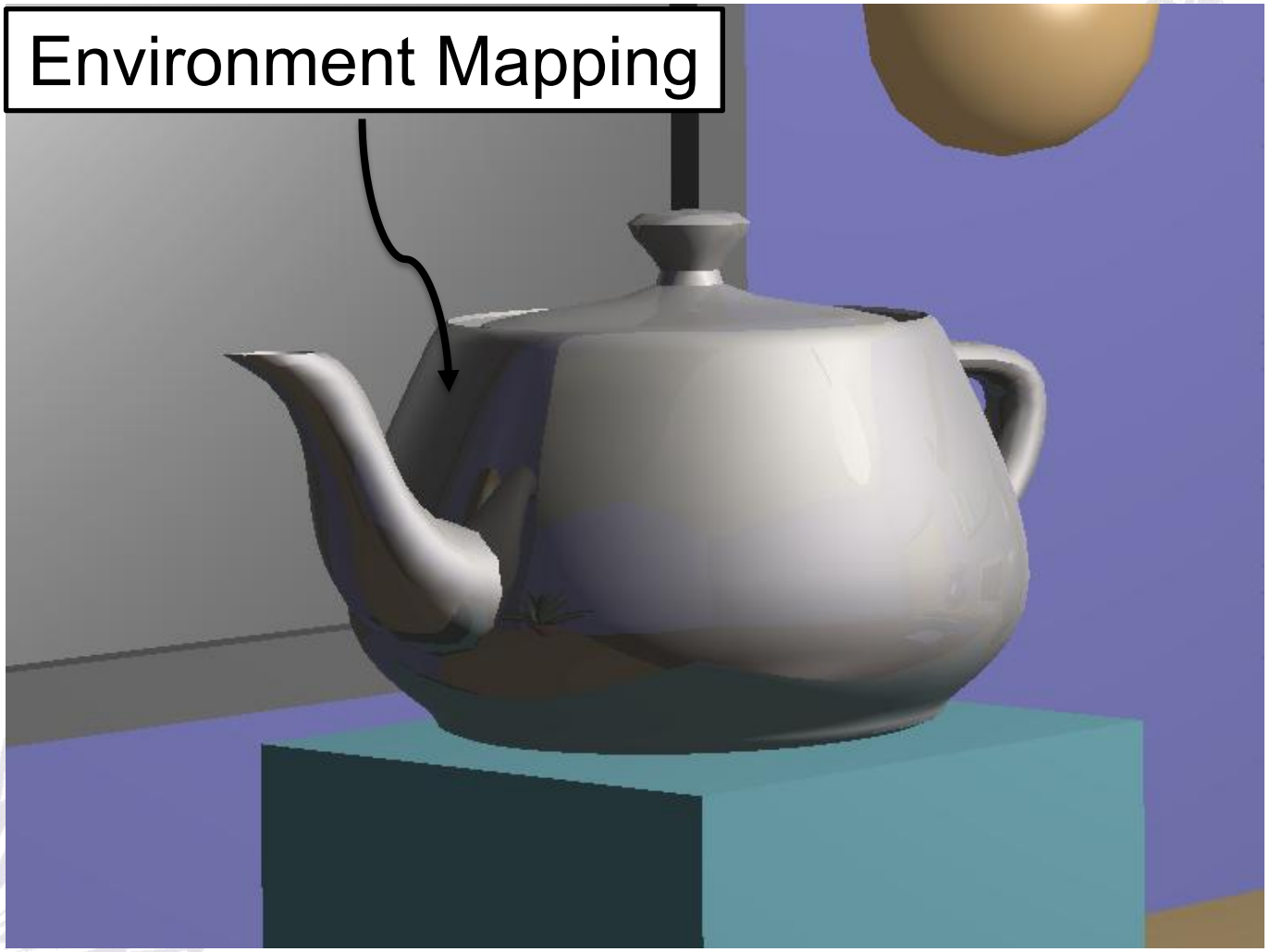


Bronze

Chrome



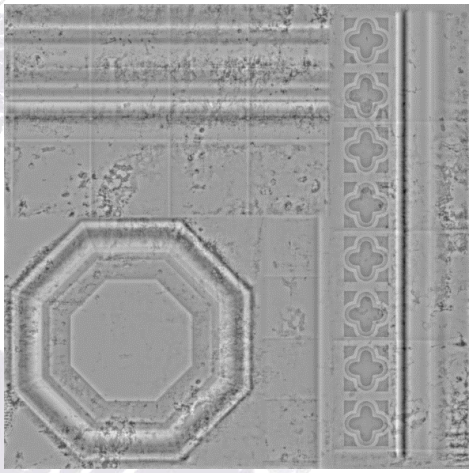
Stainless Steel



NORMAL & ENVIRONMENT MAP

Bump Mapping

- An old technique. Bump maps are grayscale 8-bit images. Values in a bump map represents up or down
- Value~255: Pull out of surface; Value~0: Pushing into surface
- Can be converted to normal maps



Bump Map from Crytek-Sponza

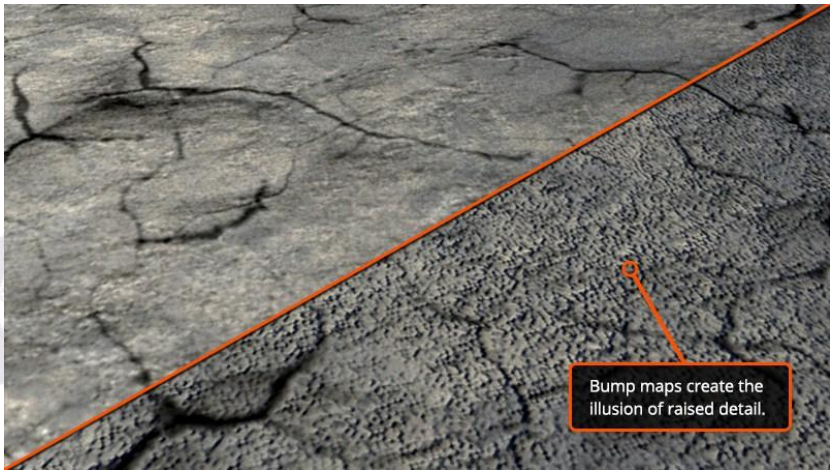


Illustration of Bump Mapping

Normal Mapping

- Use RGB value to present normal orientation
- Two types
  - ***Tangent space normal map***: good for animated object, more common and looks majorly purple/blue
  - ***Object space normal map***: good for static object, object cannot be animated, less common and looks like RGB rainbow map

A Typical Tangent Space Normal Map

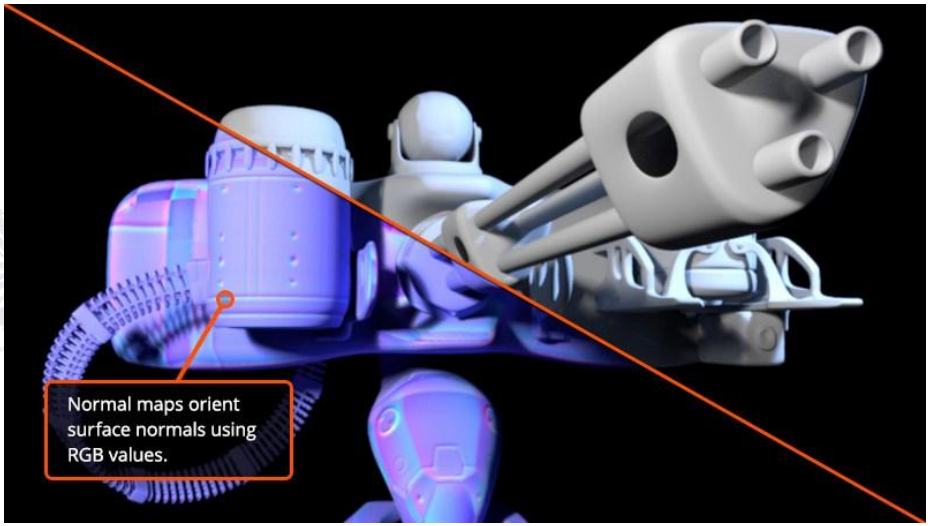


Illustration of Normal Mapping

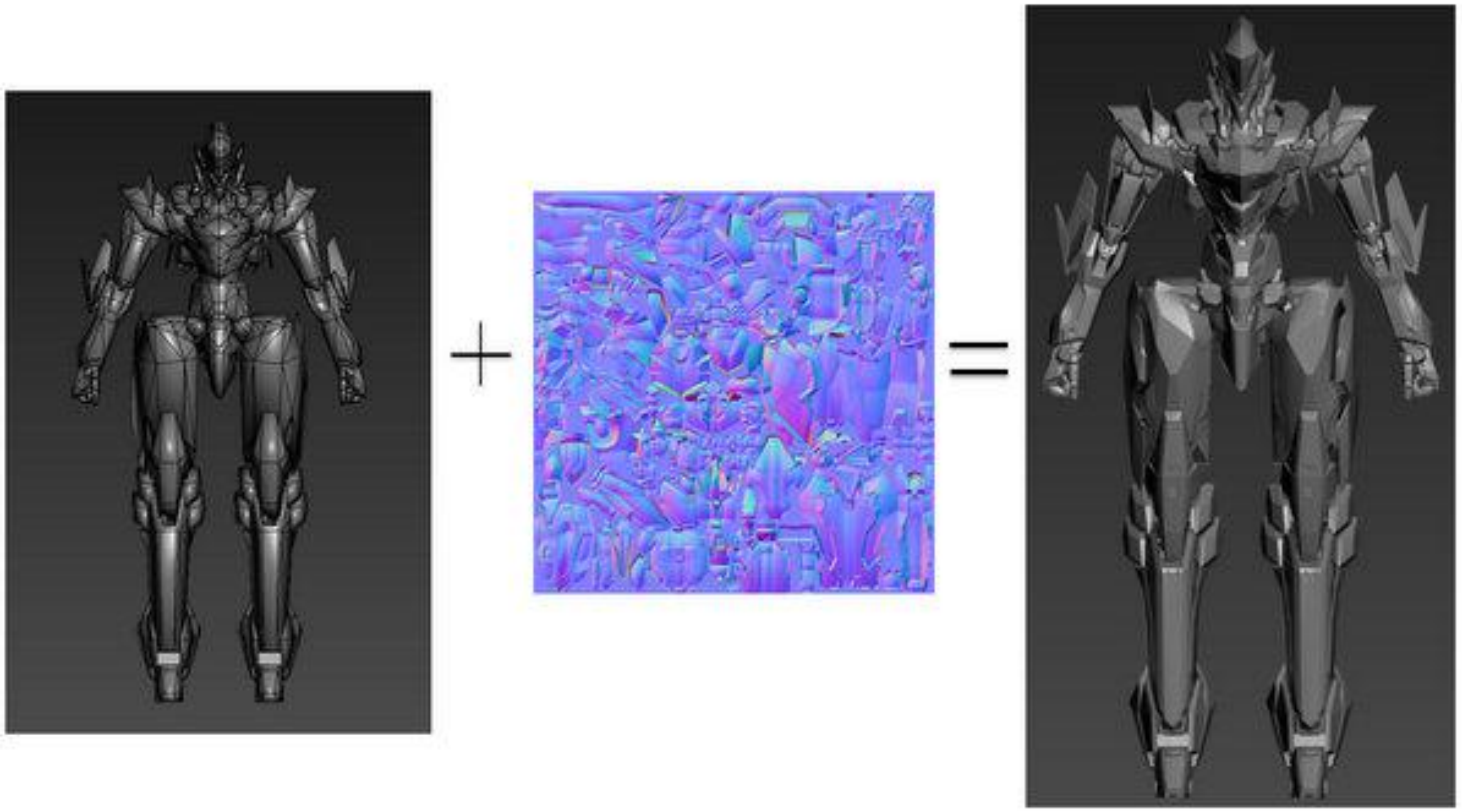
Normal Mapping

- Why are these strange textures so important?
- It's a way to ***approximate high resolution meshes with low polygon meshes*** to save GPU (i.e., vertex shader) resource
- On mobile platforms, the GPU resources are so scarce that even normal mapping can cause performance issue…
- This will eventually be ***solved by hardware one day***, but we want graphics ***as good as possible today***

Implosion, Rayark Inc.

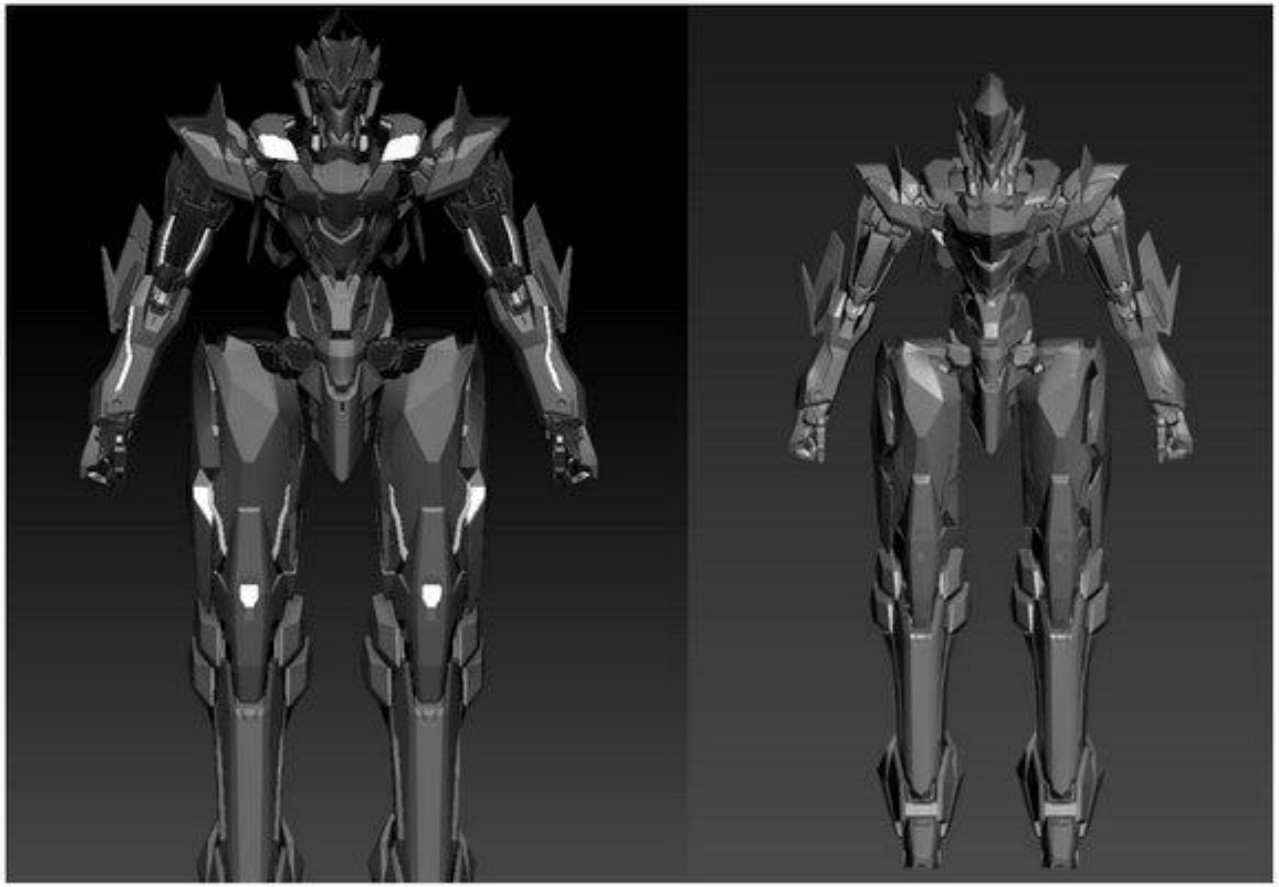
- 冠軍遊戲開發商教你打造高效能遊戲App
- http://www.ithome.com.tw/news/92171

Implosion, Rayark Inc.



Lowpoly Mesh + Normal Map = Performance + (Faked) Details

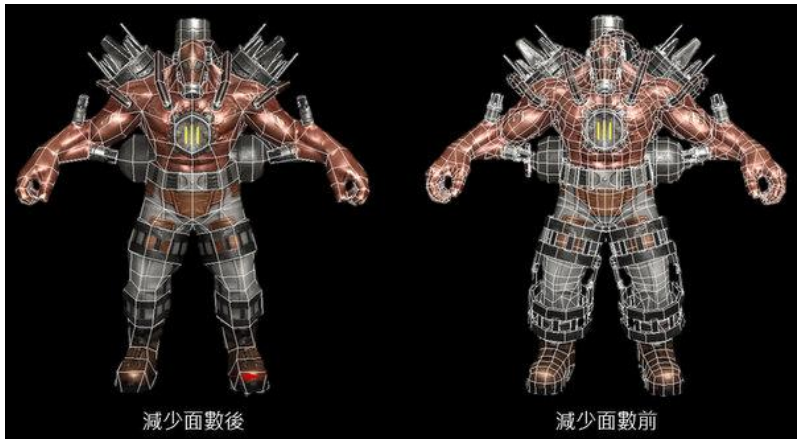
Implosion, Rayark Inc.



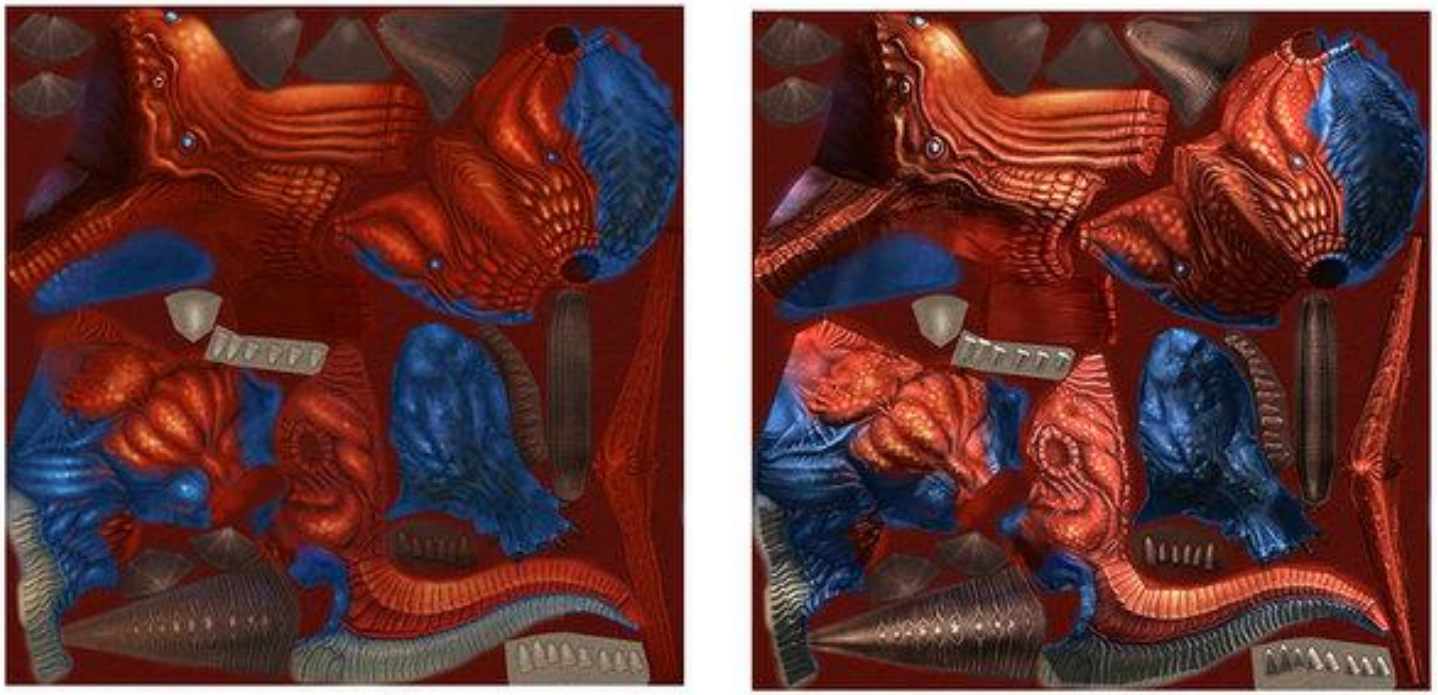
高面數模型

低面數模型 + Normal map

Implosion, Rayark Inc.



Implosion, Rayark Inc.



Bake前　　Bake後

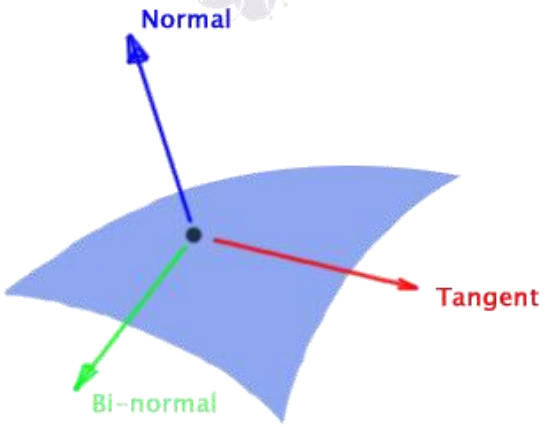
The light source is a fixed directional light in Implosion
Pre-calculate lighting and save it as diffuse map

Remark

- If a triangle produce very few (<5) pixels in a view, your model is too complicated for that view
  - Vertex shader/rasterization become the bottleneck
  - Reduce triangle count
  - Level of detail
- If some operation gives nearly the same result in every frame, pre-calculate it and save it as a map
- Try pushing expensive operations (e.g., lighting) to vertex shader, interpolation often can give nearly the same result

Tangent Space Normal Map

- Tangent space
  - A local coordinate system
  - Positive z axis is aligned with the surface normal
  - X axis: tangent
  - Y axis: bi-tangent (bi-normal)



- For best results, tangent and bi-tangent should align with the direction of the U and V in the texture coordinate
- Tangent space is not affected by movement or animation of the mesh, which is a good property

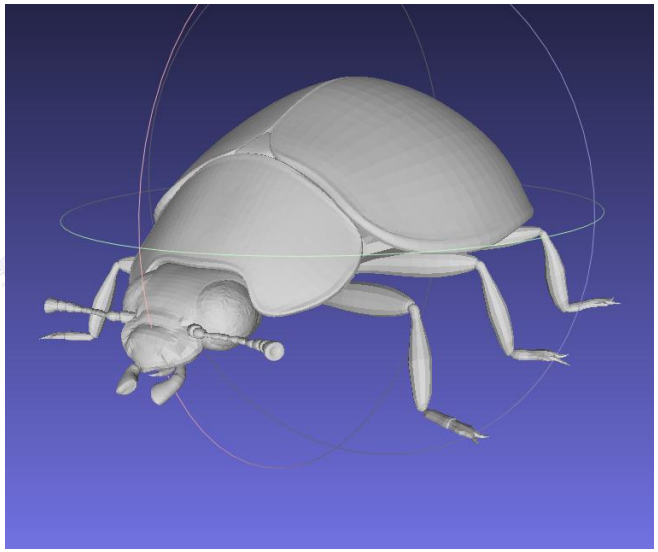
Tangent Space Transformation

- Recall that all lighting computation must be done ***in the same space***, and ***tangent space*** will do too!
- Convert light and view vector to tangent space of the vertex
- $\vec{N}$: normal, $\vec{T}$: tangent, $\vec{B}$: bi-tangent, all ***in eye space***

- $TBN = \begin{bmatrix} \vec{T}.x & \vec{T}.y & \vec{T}.z \\ \vec{B}.x & \vec{B}.y & \vec{B}.z \\ \vec{N}.x & \vec{N}.y & \vec{N}.z \end{bmatrix}$. Note that $TBN^T = TBN^{-1}$
- $\vec{V}_{tangent} = TBN * \vec{V}_{eye}$
- $\vec{L}_{tangent} = TBN * \vec{L}_{eye}$

See lecture 01: Transformation!

Normal Mapping: Inputs



Triangle mesh

Diffuse texture

Normal map

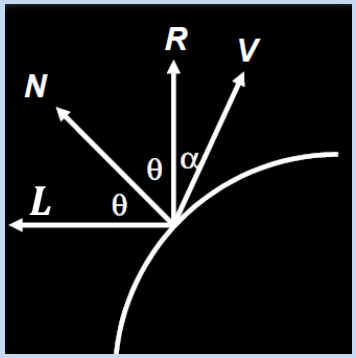
Normal Mapping: Output

Code: Vertex Shader

```
#version 410 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 tangent;
layout (location = 3) in vec2 texcoord;

out VS_OUT{
    vec2 texcoord;
    vec3 eyeDir;
    vec3 lightDir;
    vec3 normal;
} vs_out;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform vec3 light_pos = vec3(0.0, 0.0, 100.0);
```

Code: Vertex Shader (Cont'd)

```
void main(void)
{
    vec4 P = mv_matrix * vec4(position, 1.0);

    // Eye space to tangent space TBN
    vec3 T = normalize(mat3(mv_matrix) * tangent);
    vec3 N = normalize(mat3(mv_matrix) * normal);
    vec3 B = cross(N, T);

    vec3 L = light_pos - P.xyz;
    vs_out.lightDir = normalize(vec3(dot(L, T), dot(L, B), dot(L, N)));

    vec3 V = -P.xyz;
    vs_out.eyeDir = normalize(vec3(dot(V, T), dot(V, B), dot(V, N)));

    vs_out.texcoord = texcoord;

    // vs_out.normal = N; // For comparison only
    gl_Position = proj_matrix * P;
}
```



Code: Fragment Shader

```
#version 410 core

out vec4 color;
// Color and normal maps
layout (binding = 0) uniform sampler2D tex_color;
layout (binding = 1) uniform sampler2D tex_normal;

in VS_OUT
{
    vec2 texcoord;
    vec3 eyeDir;
    vec3 lightDir;
    vec3 normal;
} fs_in;
```

Code: Fragment Shader (Cont'd)

```
void main(void)
{
    vec3 V = normalize(fs_in.eyeDir);
    vec3 L = normalize(fs_in.lightDir);
    vec3 N = normalize(texture(tex_normal, fs_in.texcoord).rgb * 2.0 -
vec3(1.0));

    vec3 R = reflect(-L, N);

    vec3 diffuse_albedo = texture(tex_color, fs_in.texcoord).rgb;
    vec3 diffuse = max(dot(N, L), 0.0) * diffuse_albedo;

    vec3 specular_albedo = vec3(1.0);
    vec3 specular = max(pow(dot(R, V), 20.0), 0.0) * specular_albedo;

    color = vec4(diffuse + specular, 1.0);
}
```

$$I_p = \cancel{k_a I_a} + k_d I_d \left(\vec{L} \cdot \vec{N}\right) + k_s I_s \left(\vec{R} \cdot \vec{V}\right)^{20}$$

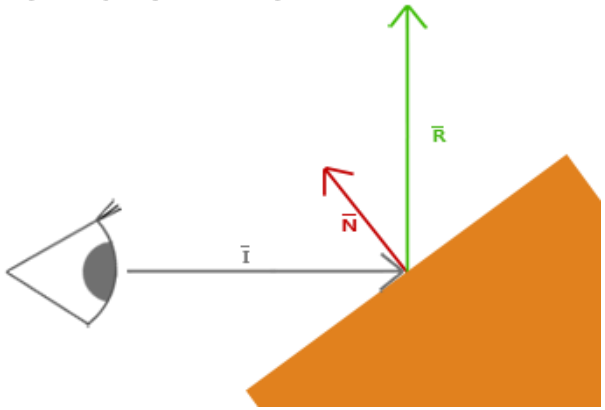
Reflect function

void reflect(genType I, genType N);

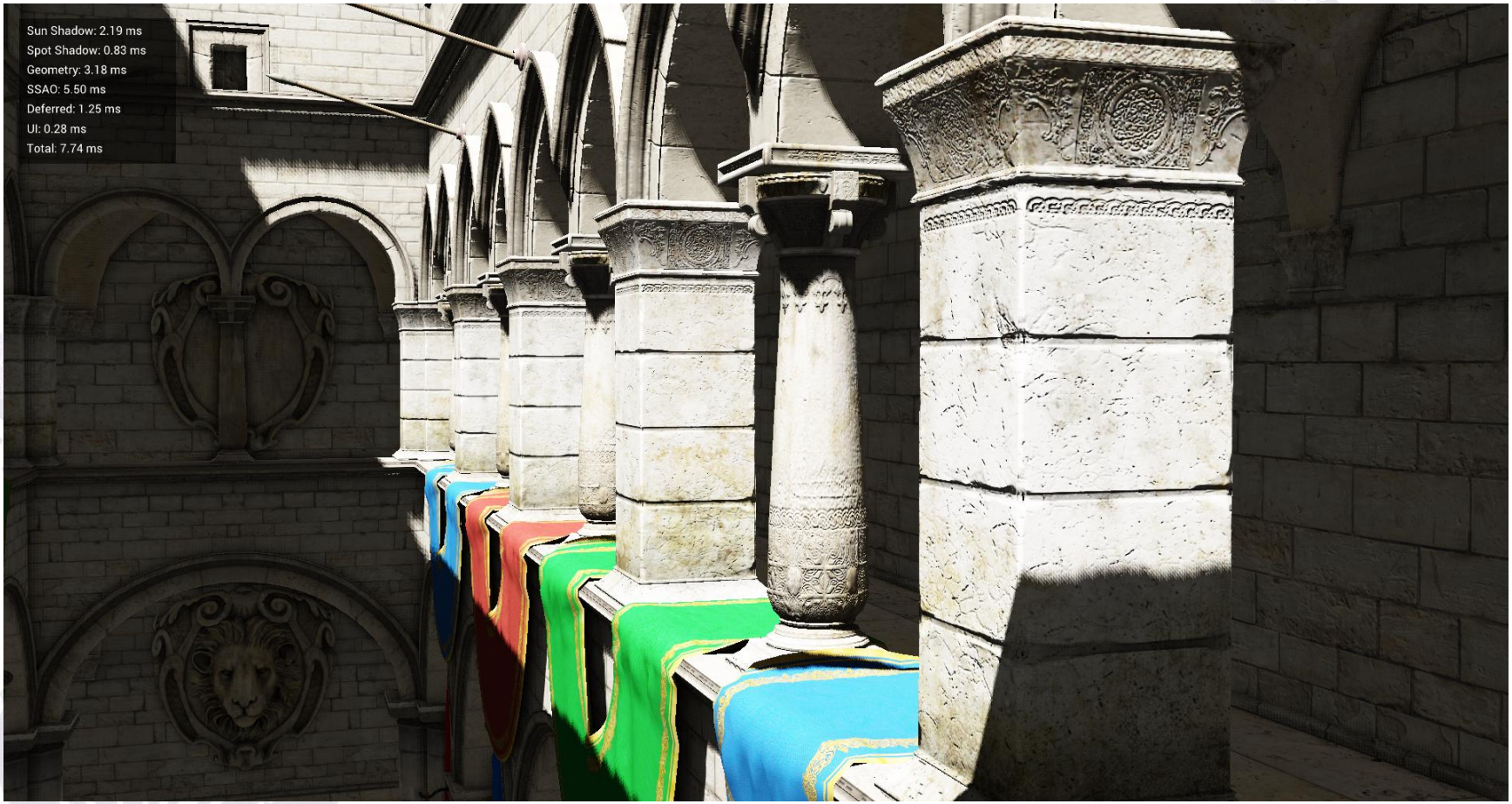
- **Description**: For a given incident vector ***I*** and surface normal ***N*** reflect returns the reflection direction calculated as

$$I - 2.0 * \text{dot}(N, I) * N.$$

- ***N*** should be normalized in order to achieve the desired result.



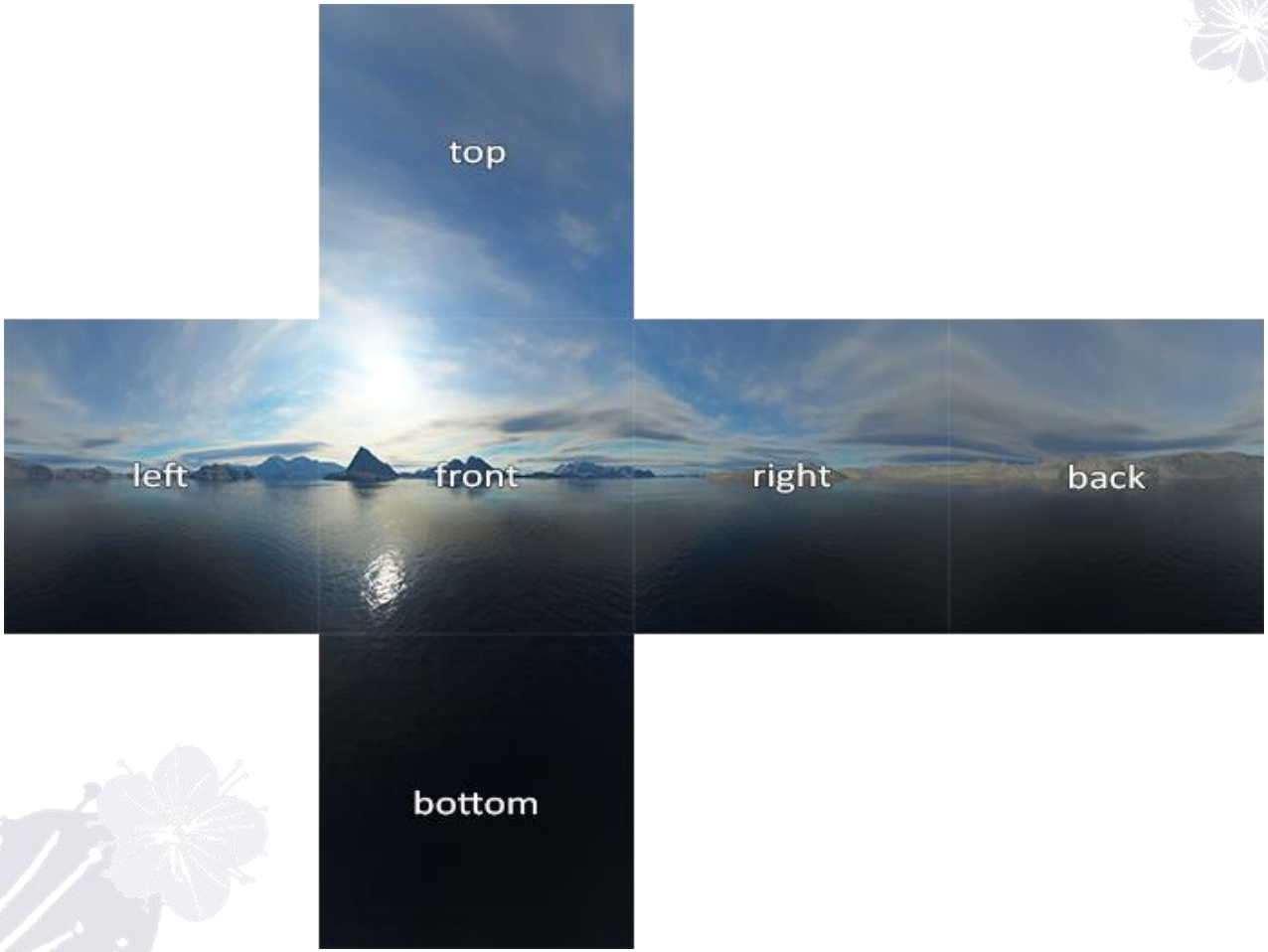
Sponza: Normal Mapped



Cubemap

- A cubemap is treated as a single texture object but it is made up of ***six square 2D images*** that make up the six sides of a cube
- Applications:
  - 3D light maps
  - Reflections
  - Highly accurate environment maps
  - Many more…

Cubemap



Cubemap

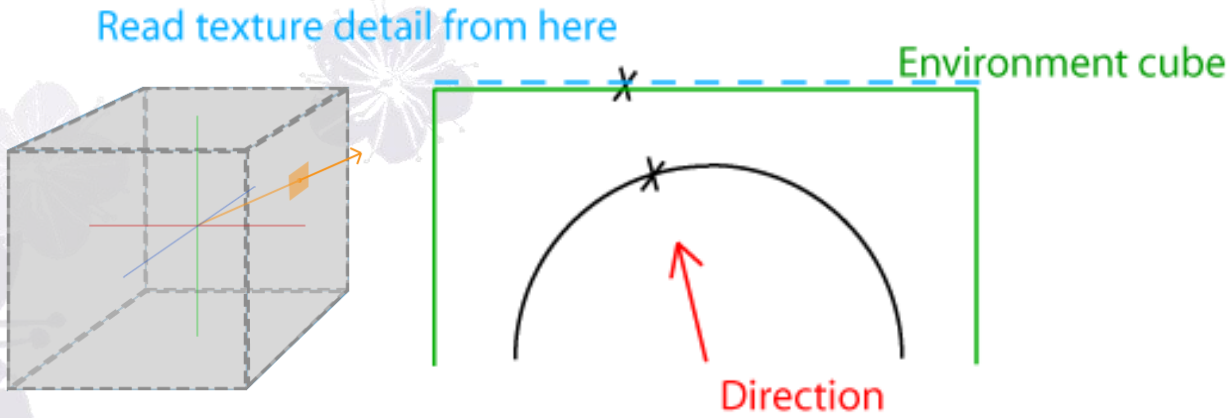
- Create a texture object by binding a new name to the **GL_TEXTURE_CUBE_MAP** target
- Image data for 6 targets must have ***same size*** and must be ***square*** (width = height)
- Call **glTexImage2D**() once for each face of the cubemap, using the same size, mipmap level, and internalformat:
  - **GL_TEXTURE_CUBE_MAP_POSITIVE_X**
  - **GL_TEXTURE_CUBE_MAP_NEGATIVE_X**
  - **GL_TEXTURE_CUBE_MAP_POSITIVE_Y**
  - **GL_TEXTURE_CUBE_MAP_NEGATIVE_Y**
  - **GL_TEXTURE_CUBE_MAP_POSITIVE_Z**
  - **GL_TEXTURE_CUBE_MAP_NEGATIVE_Z**
- These 6 targets have ***consecutive values***, so you can use **GL_TEXTURE_CUBE_MAP_POSITIVE_X** + N

Code: Load Cubemap

```
GLuint texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_CUBE_MAP, texture);
for(int face = 0; face < 6; face++)
{
    glTexImage2D(GL_TEXURE_CUBE_MAP_POSITIVE_X + face,
        0,
        0, 0,
        width, height,
        format, type,
        data + face * face_size_in_bytes
    );
}
```

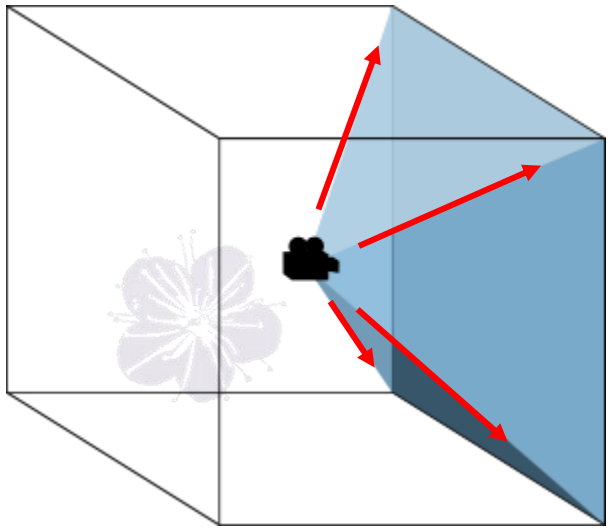
Cubemap

- Texture coordinates for cubemaps have three dimensions
- S, T, and R texture coordinates represent a signed vector from the center of the texture map pointing outward
- This vector will intersect one of the six sides of the cubemap. The texels around this intersection point are then sampled to create the filtered color value from the texture



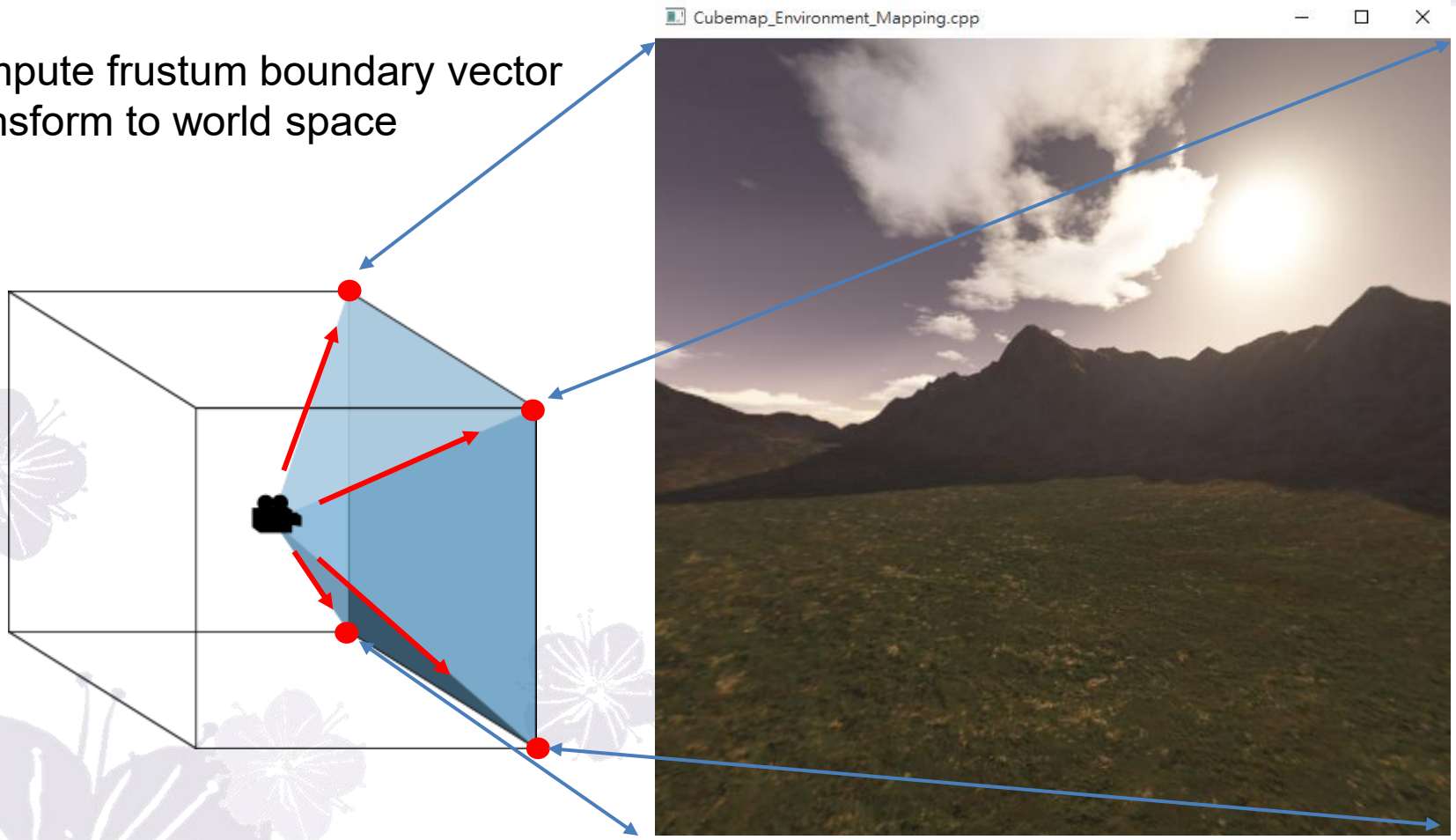
Skybox Renderer

- Render a skybox using a cubemap
- Draw a full screen quad and compute cubemap texcoord for each fragment



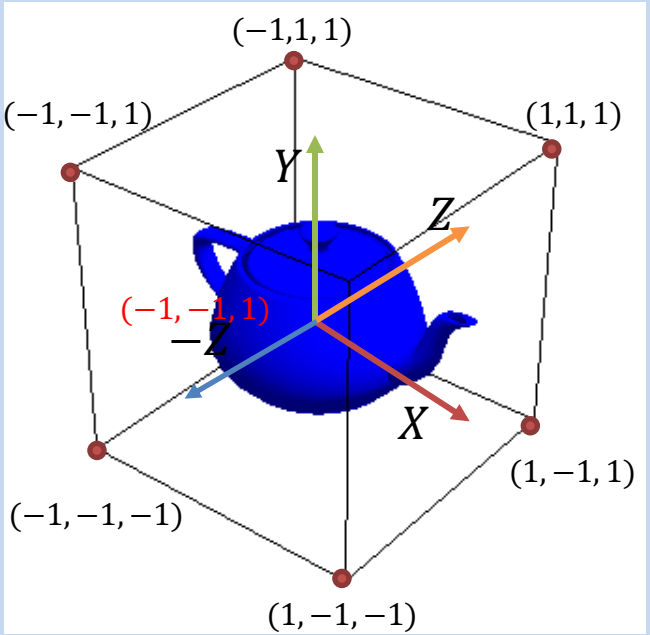
Skybox Renderer

1. Compute frustum boundary vector
2. Transform to world space



Code: Vertex Shader

```
#version 410 core

out VS_OUT
{
    vec3 tc;
} vs_out;

uniform mat4 vp_matrix;
uniform vec3 eye_position;

void main(void)
{
    // A square at far plane in clip space
    vec3[4] vertices = vec3[4](
        vec3(-1.0, -1.0, 1.0),
        vec3(1.0, -1.0, 1.0),
        vec3(-1.0, 1.0, 1.0),
        vec3(1.0, 1.0, 1.0));
    vec4 tc = inverse(vp_matrix) * vec4(vertices[gl_VertexID], 1.0);
    vs_out.tc = tc.xyz / tc.w - eye_position;
    gl_Position = vec4(vertices[gl_VertexID], 1.0);
}
```



Code: Fragment Shader

```
#version 410 core

uniform samplerCube tex_cubemap;

in VS_OUT
{
    vec3 tc;
} fs_in;

layout (location = 0) out vec4 color;

void main(void)
{
    color = texture(tex_cubemap, fs_in.tc);
}
```

Seamless Cubemap

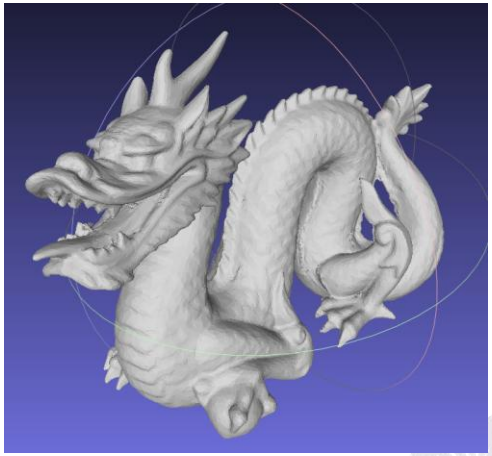
- Under the standard filtering rules for cubemaps, filtering does not work across faces of the cubemap
- This results in a ***seam*** across the faces of a cubemap
- This was a hardware limitation in the past, but modern hardware is capable of ***interpolating across a cube face boundary***
- To globally enable this, use **glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS**)

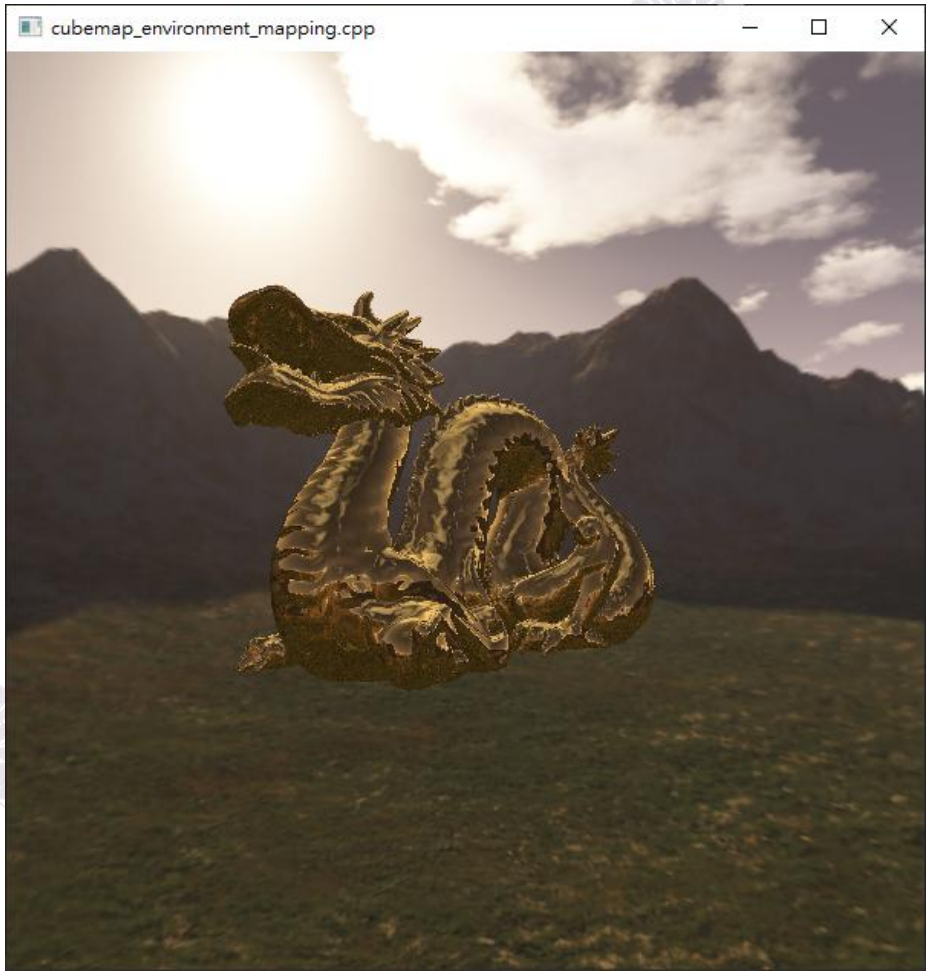
Seamless Cubemap

Skybox W/O
GL_TEXTURE_CUBE_MAP_SEAMLESS

Skybox W/
GL_TEXTURE_CUBE_MAP_SEAMLESS

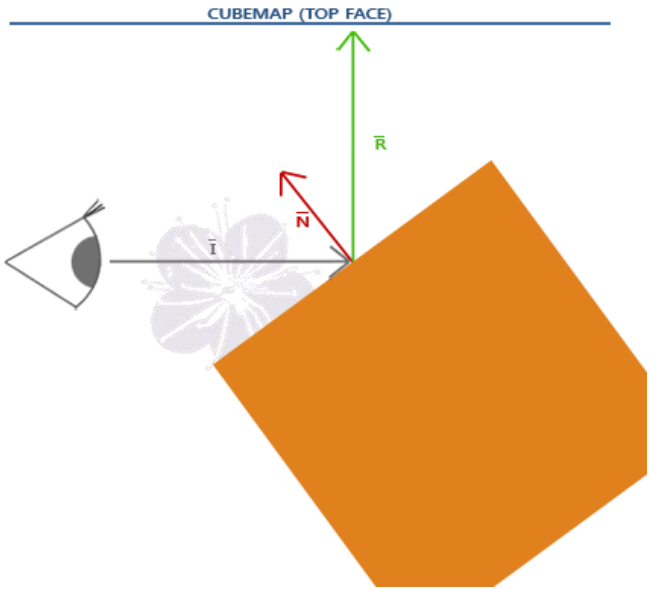
Environment Mapping





Cubemap

- Create an object that reflects its surroundings
- Calculate reflection vector $\vec{R}$ like we did in Phong lighting
- $\vec{R} = 2\left(\vec{\mathrm{V}} \cdot \vec{N}\right)\vec{N} - \vec{\mathrm{V}}$



Code: Vertex Shader

```
#version 410 core

uniform mat4 m_matrix;
uniform mat4 vp_matrix;
uniform vec3 eye_position;
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;

out VS_OUT
{
    vec3 normal;
    vec3 view;
} vs_out;

void main(void)
{
    // Compute vectors in world-space
    vec4 pos_vs = m_matrix * vec4(position, 1.0);
    vs_out.normal = mat3(transpose(inverse(m_matrix))) * normal;
    vs_out.view = pos_vs.xyz - eye_position;
    gl_Position = vp_matrix * pos_vs;
}
```

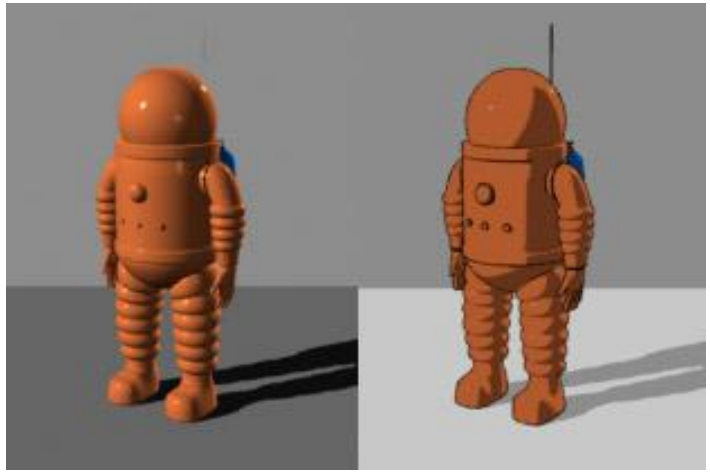
Code: Fragment Shader

```
#version 410 core

uniform samplerCube tex_cubemap;

in VS_OUT
{
    vec3 normal;
    vec3 view;
} fs_in;

out vec4 color;

void main(void)
{
    // Reflect view vector about the plane defined by the normal
    // at the fragment in world space
    vec3 r = reflect(normalize(fs_in.view), normalize(fs_in.normal));
    // Sample from reflection vector
    // Gold color = RGB(0.95, 0.80, 0.45)
    color = texture(tex_cubemap, r) * vec4(0.95, 0.80, 0.45, 1.0);
}
```

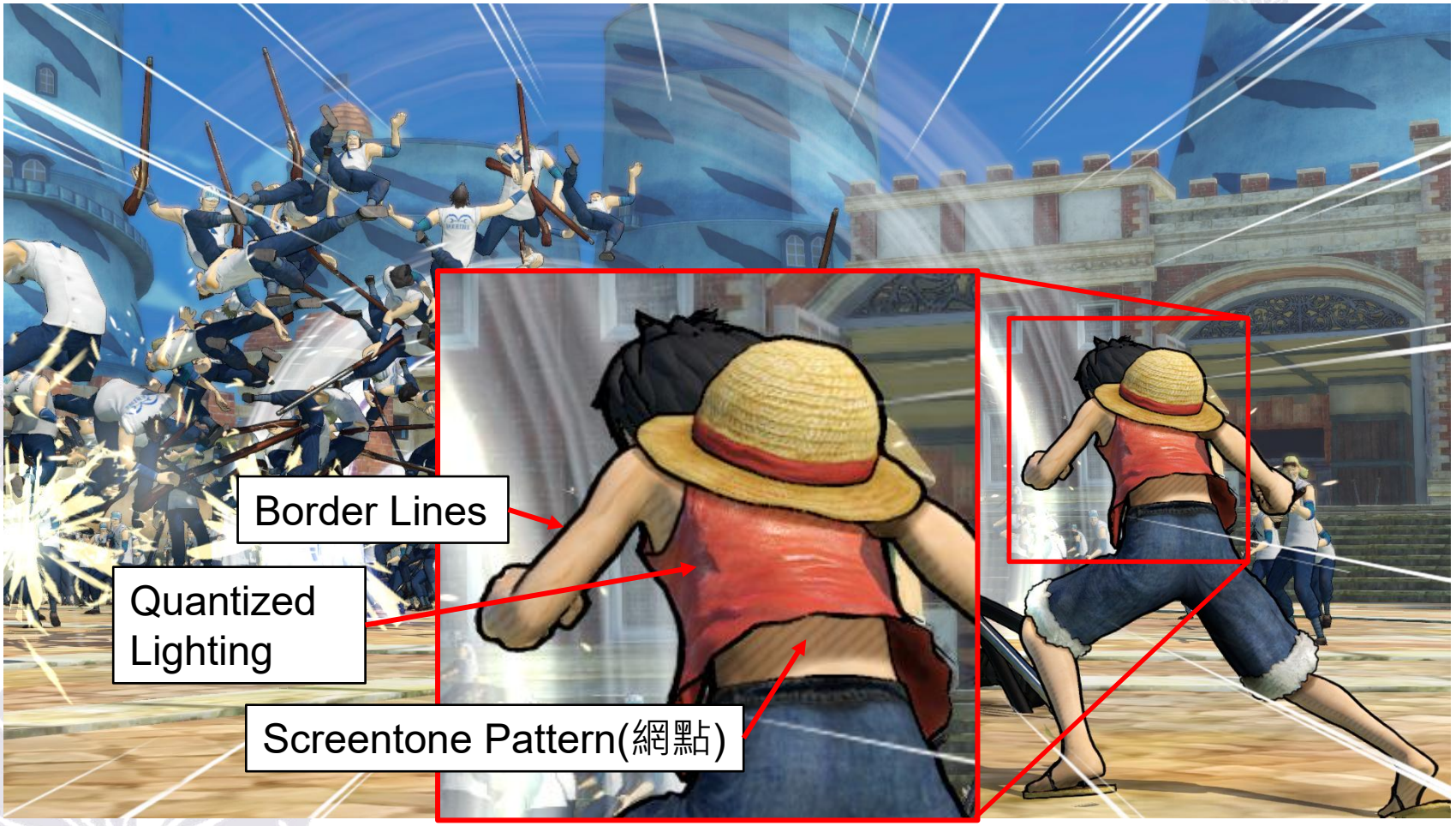
CEL SHADING

Cel Shading

- “Cel” is short for celluloid (賽璐珞)
- Also called toon shading
- Non-realistic, used to render ***3D*** scenes with ***flat***, ***comic-like*** or ***cartoon-like style***
- Consisted of a family of rendering techniques
  - Quantized diffuse shading value
  - Border detection
  - Pencil or oil painting style
  - Many more…

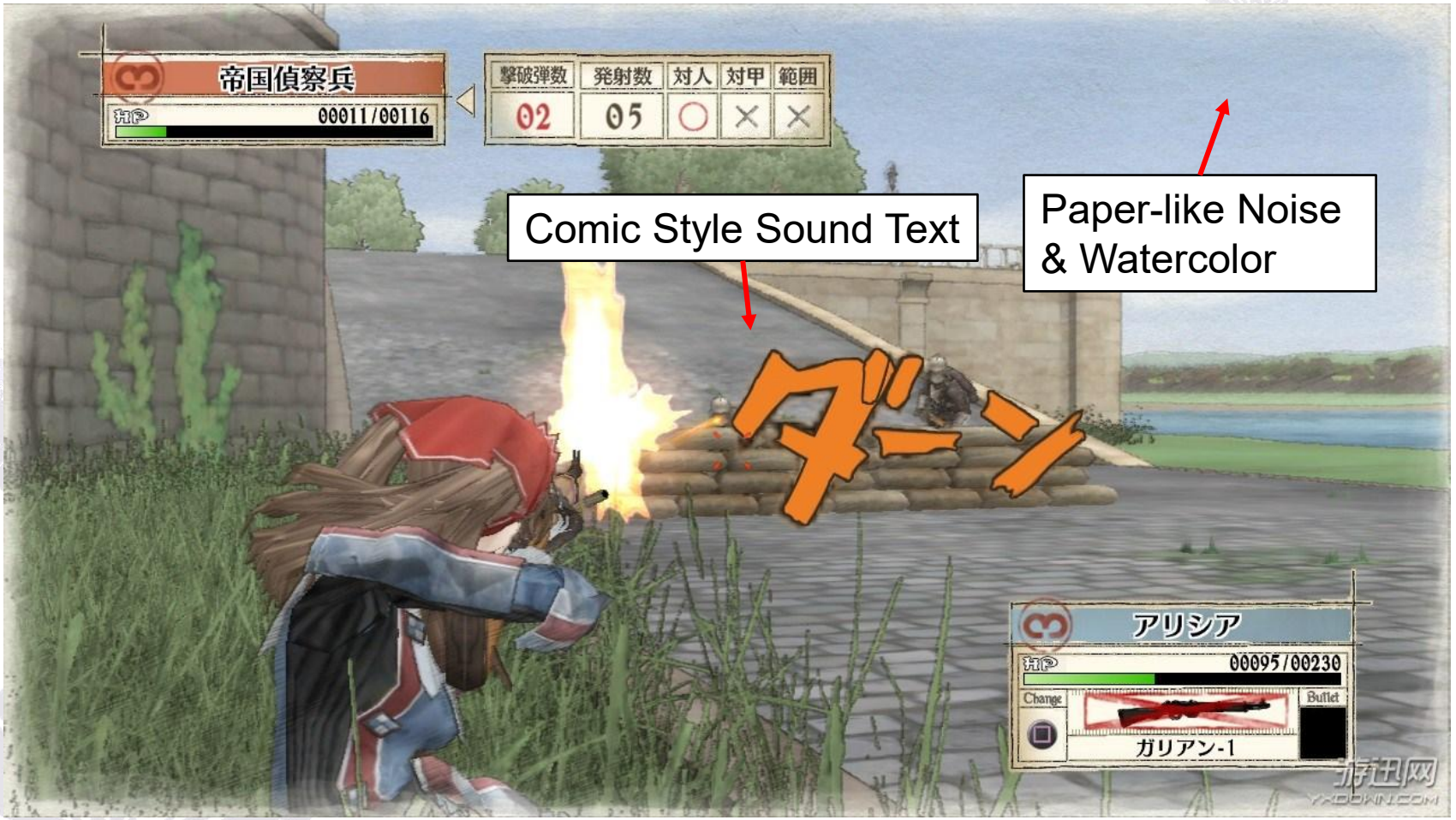


Cel Shading Examples



One Piece Pirate Warriors 3

Cel Shading Examples



Valkyria Chronicles

Cel Shading Examples

Guilty Gear Xrd

Cel Shading

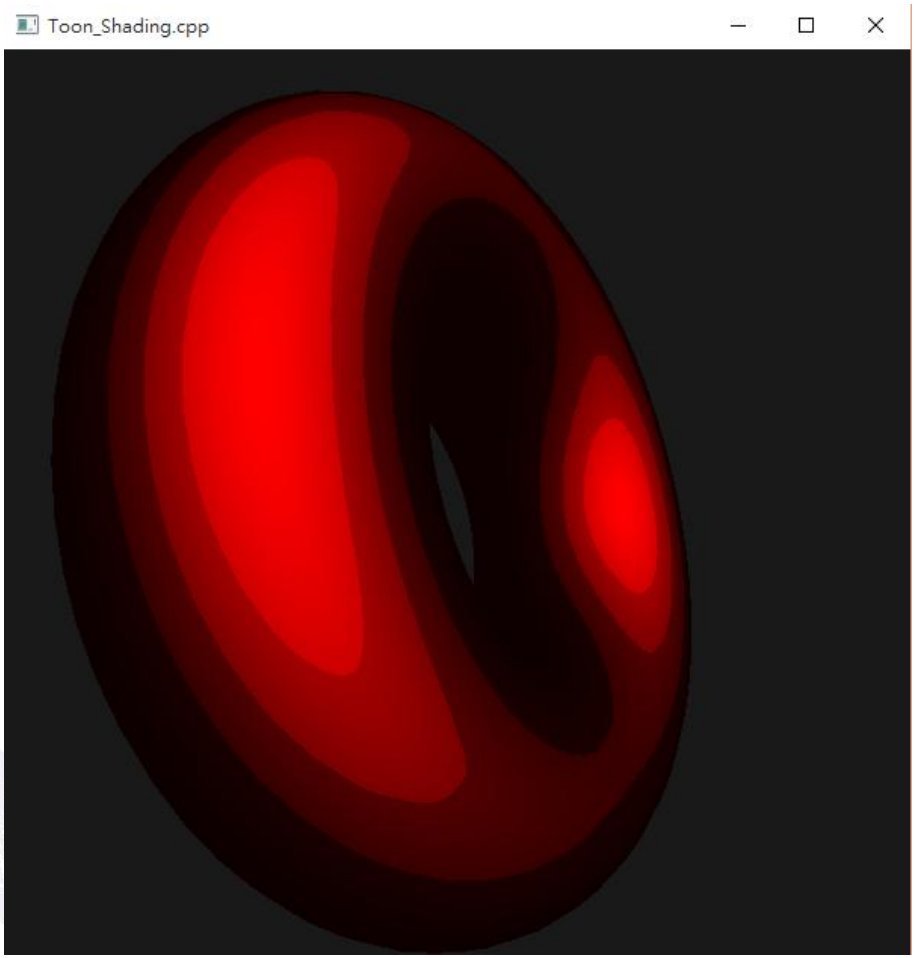
- Different from photo-realistic shading, which aims to make the shading as close to ***real world*** as possible,
- A cel shading style is often defined by ***an artistic style made by artists***
- Therefore, it is highly ***application and content aware***, a graphics programmer would work closely with artists to create a solution

Quantized Lighting

- Similar to quantization in image processing, but it is the ***lighting intensity*** that get quantized
- The basic idea is to use the diffuse lighting intensity $\vec{N} \cdot \vec{L}$ as the texture coordinate into a 1D texture that contains a gradually brightening color table

1D Color Lookup Table

A Cel-Shaded Torus



Code: Color Table 1D Texture Setup

```
static const GLubyte toon_tex_data[] =
{
    0x44, 0x00, 0x00, 0x00, // RGBA(0.25, 0.0, 0.0, 0.0)
    0x88, 0x00, 0x00, 0x00, // RGBA(0.5, 0.0, 0.0, 0.0)
    0xCC, 0x00, 0x00, 0x00, // RGBA(0.75, 0.0, 0.0, 0.0)
    0xFF, 0x00, 0x00, 0x00 // RGBA(1.0, 0.0, 0.0, 0.0)
};
glGenTextures(1, &tex_toon);
glBindTexture(GL_TEXTURE_1D, tex_toon);

glTexStorage1D(GL_TEXTURE_1D, 1, GL_RGB8, sizeof(toon_tex_data) / 4);
glTexSubImage1D(GL_TEXTURE_1D, 0,
                0, sizeof(toon_tex_data) / 4,
                GL_RGBA, GL_UNSIGNED_BYTE,
                toon_tex_data);

glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

Code: Render Function

```
static const GLfloat gray[] = { 0.1f, 0.1f, 0.1f, 1.0f };
static const GLfloat ones[] = { 1.0f };

glClearBufferfv(GL_COLOR, 0, gray);
glClearBufferfv(GL_DEPTH, 0, ones);

glUseProgram(program);

mat4 mv_matrix = translate(mat4(), vec3(0.0f, 0.0f, -3.0f));
glBindTexture(GL_TEXTURE_1D, tex_toon);

glUniformMatrix4fv(uniforms.mv_matrix, 1, GL_FALSE,
                   &mv_matrix[0][0]);
glUniformMatrix4fv(uniforms.proj_matrix, 1, GL_FALSE,
                   &proj_matrix[0][0]);

glDrawElements(GL_TRIANGLES, index_count, GL_UNSIGNED_INT, 0);

glutSwapBuffers();
```

Code: Vertex Shader

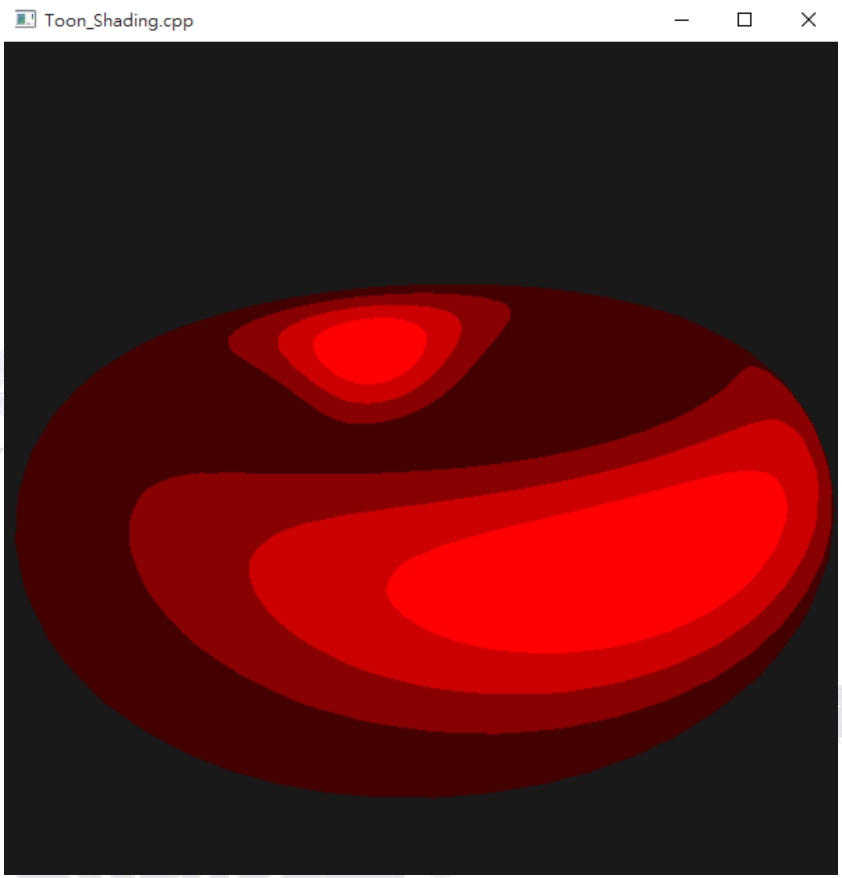
```
#version 410 core
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;
out VS_OUT
{
    vec3 normal;
    vec3 view;
} vs_out;
void main(void)
{
    vec4 pos_vs = mv_matrix * position;
    // Calculate eye-space normal and position
    vs_out.normal = mat3(mv_matrix) * normal;
    vs_out.view = pos_vs.xyz;
    // Send clip-space position to primitive assembly
    gl_Position = proj_matrix * pos_vs;
}
```

Code: Fragment Shader

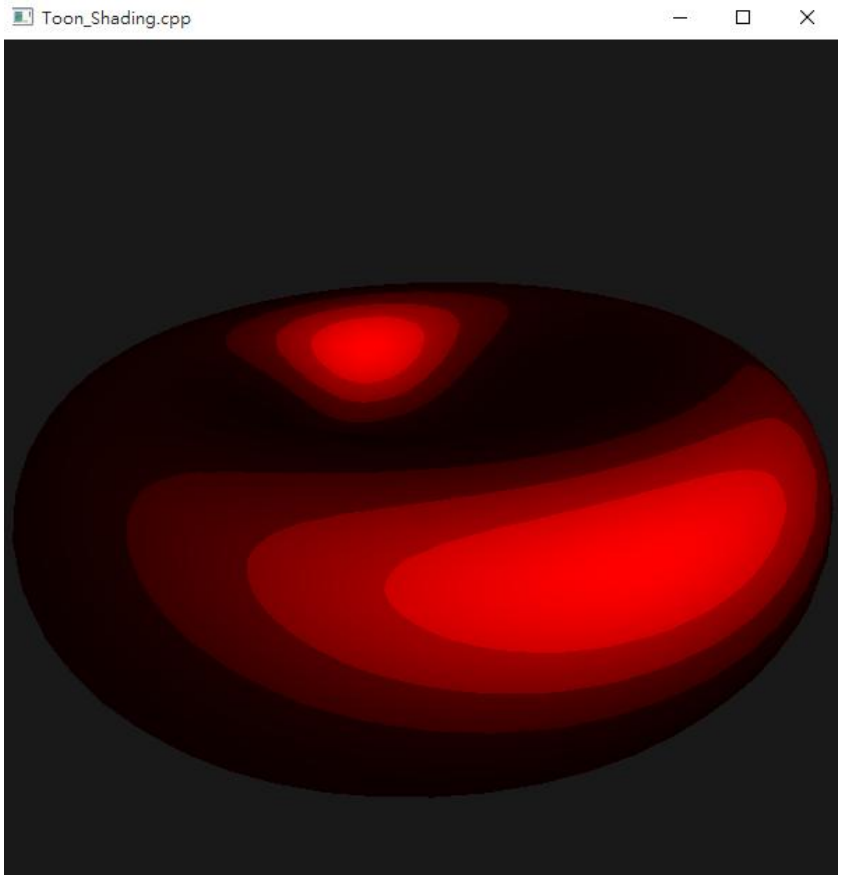
```
#version 410 core
uniform sampler1D tex_toon;
uniform vec3 light_pos = vec3(30.0, 30.0, 100.0);
in VS_OUT
{
    vec3 normal;
    vec3 view;
} fs_in;
out vec4 color;
void main(void)
{
    // Calculate per-pixel normal and light vector
    vec3 N = normalize(fs_in.normal);
    vec3 L = normalize(light_pos - fs_in.view);
    // Simple N dot L diffuse lighting
    float tc = pow(max(0.0, dot(N, L)), 5.0);
    // Sample from cell shading texture
    color = texture(tex_toon, tc) * (tc * 0.8 + 0.2);
}
```

Enhance Gradient for Better Effect

Gradient



W/O Gradient



W/ Gradient

Reference

- Global Illumination Across Industries
  - SIGGRAPH 2010 Course
- Introduction to DirectX RayTracing
  - SIGGRAPH 2018 Course
- Path Tracing in Production
  - SIGGRAPH 2019 Course
- Ray Tracing Gems
- Ray Tracing from Ground Up


thank you!
Question?