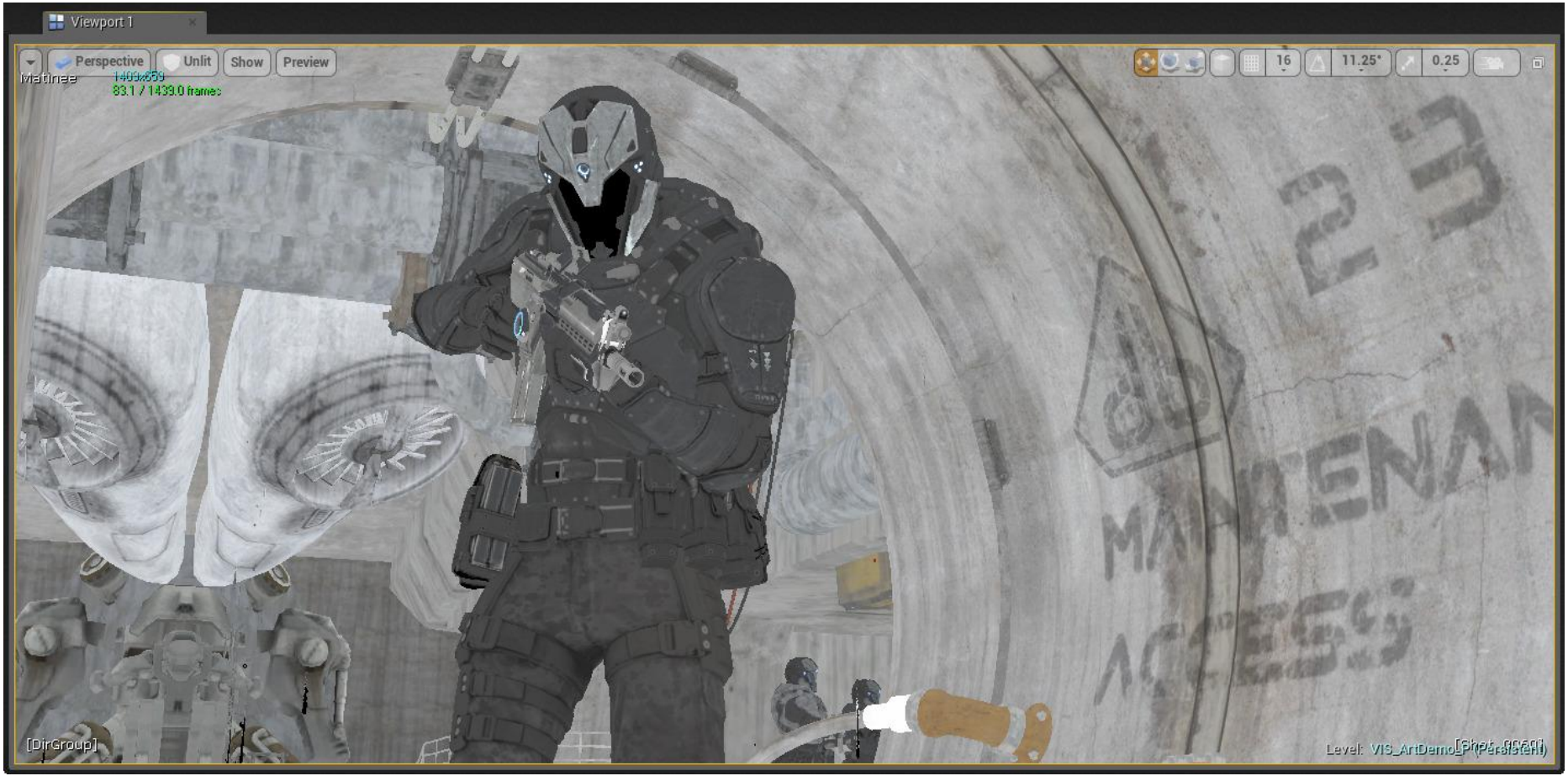
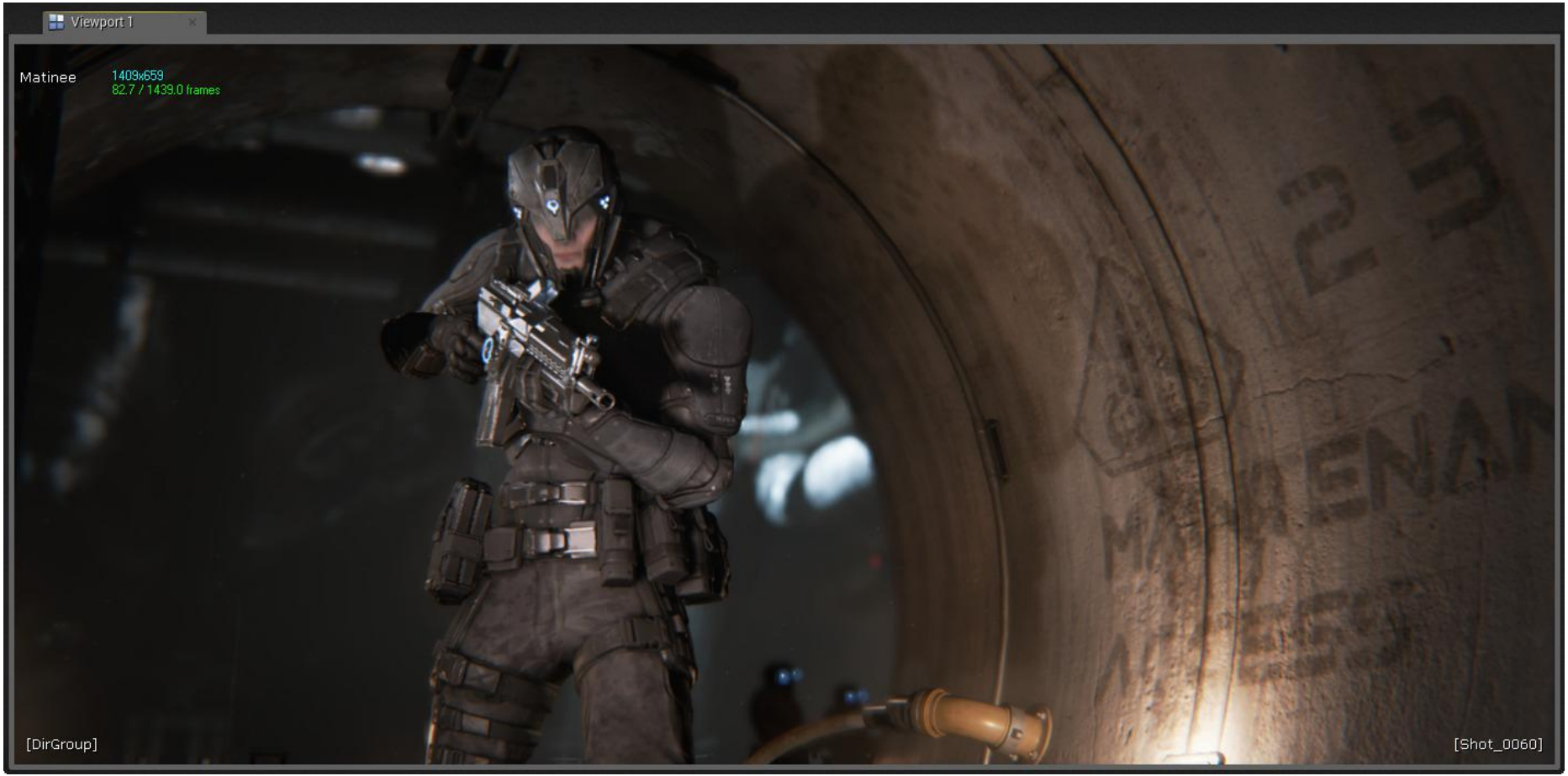
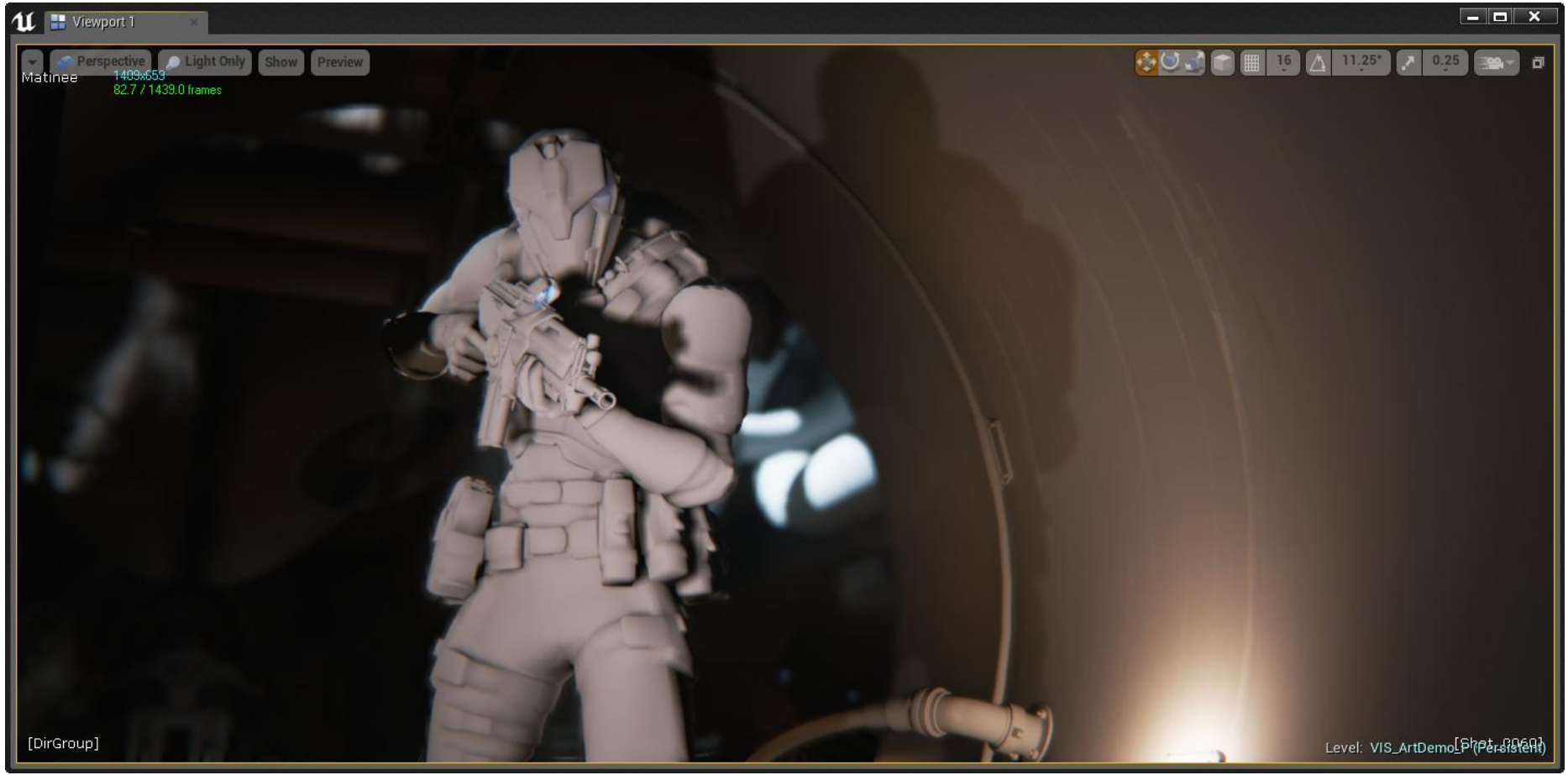
Computer Graphics labs

Lab 6 - Light

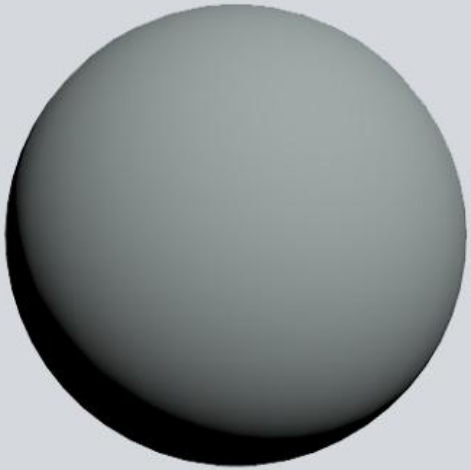

Viewport 1
Perspective
Unlit
Show
Preview
Matinee
1405x659
83.1 / 1439.0 frames
16
11.25°
0.25
[DirGroup]
Level: VIS_ArtDemo_P (Persistent)



What is the difference?

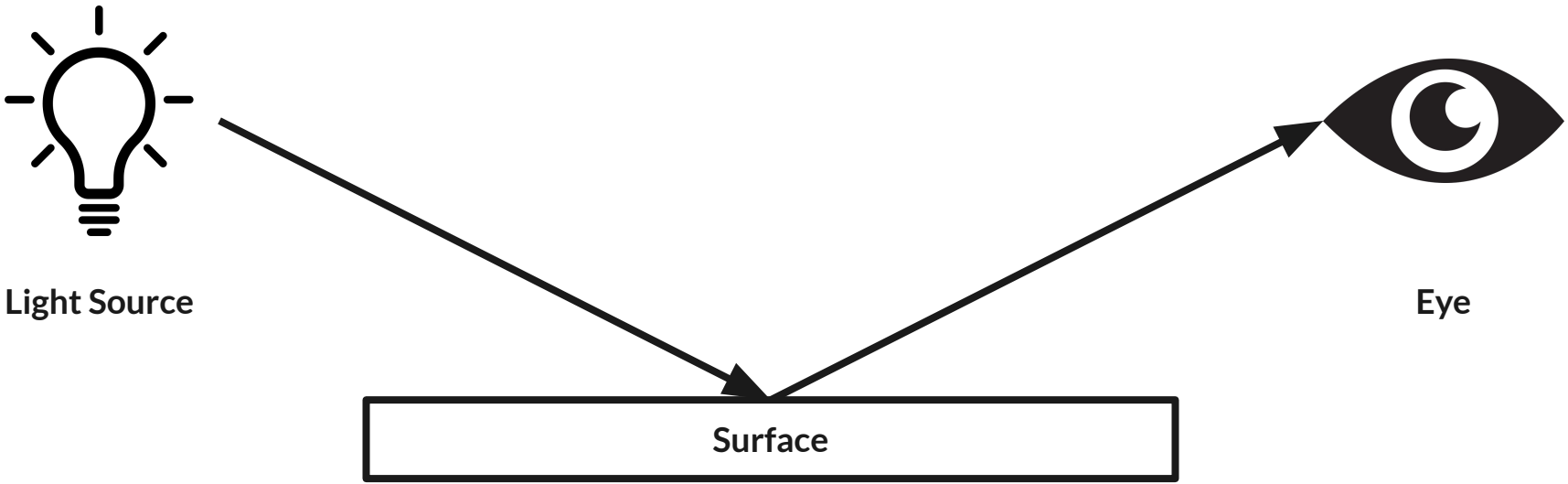


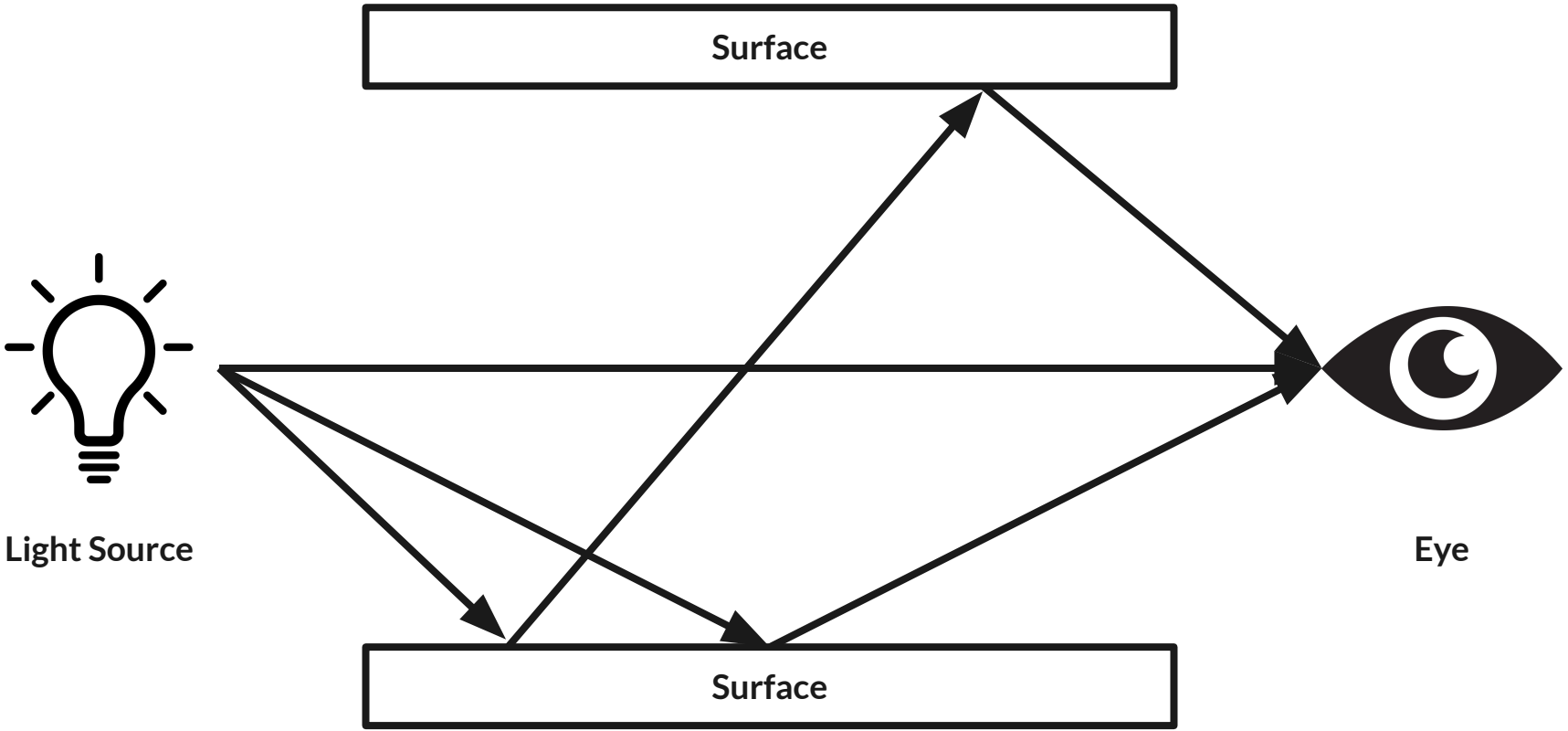
The difference is light



SHADED/LIT

FLAT SHADED/UNLIT


Light Source
Eye
Surface


Surface
Light Source
Eye
Surface

The Rendering Equation

INTRACTABLE

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\omega_i$$

All incoming directions

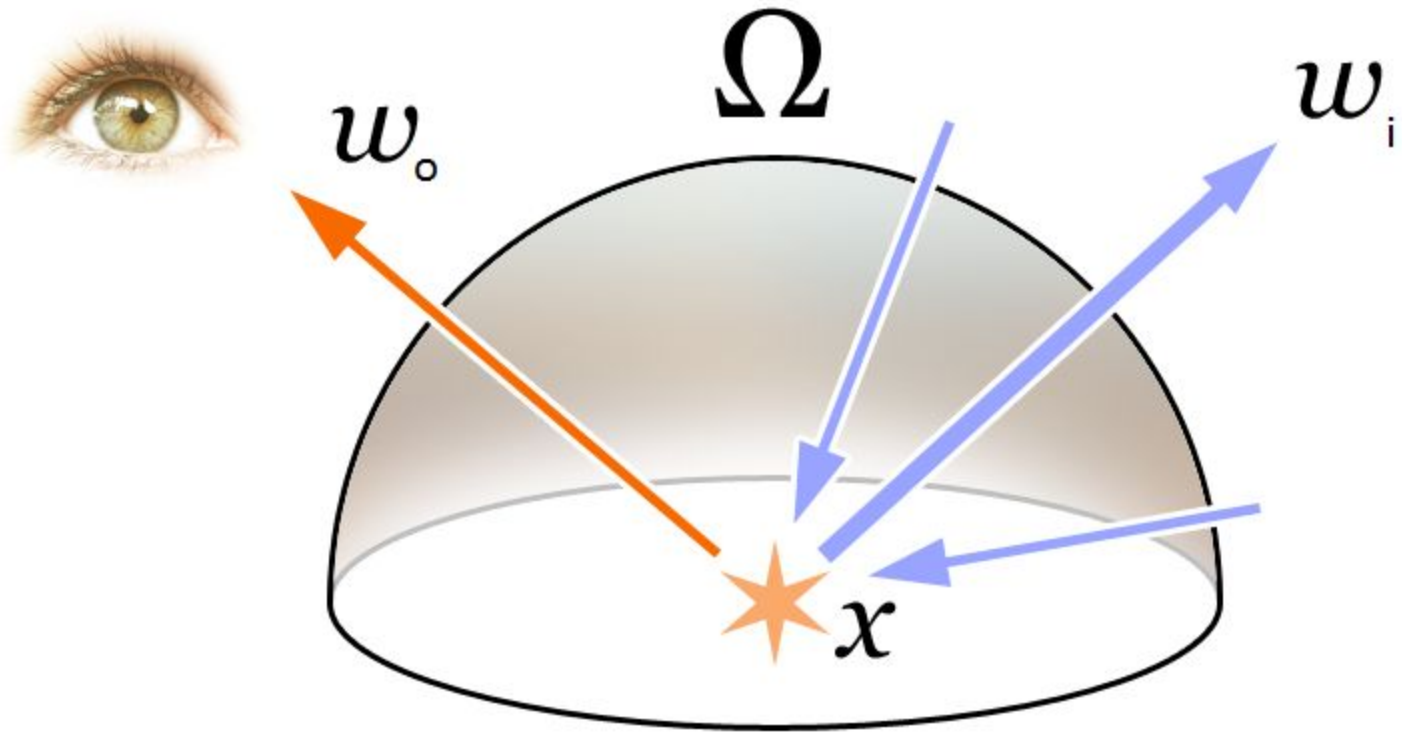
Outgoing Light

Light emitted by the object.

Bidirectional Reflectance Distribution Function (BRDF)

Incident Light

Lambert


Ω
w_o
w_i
x

Approximation 1: Monte Carlo Rendering

Instead of integrating over all incoming directions, we only take a set of random direction into consideration.

Used for path tracing & photon mapping.

High quality results but slow for real time rendering (up till now).

Used in movies and advertisements.

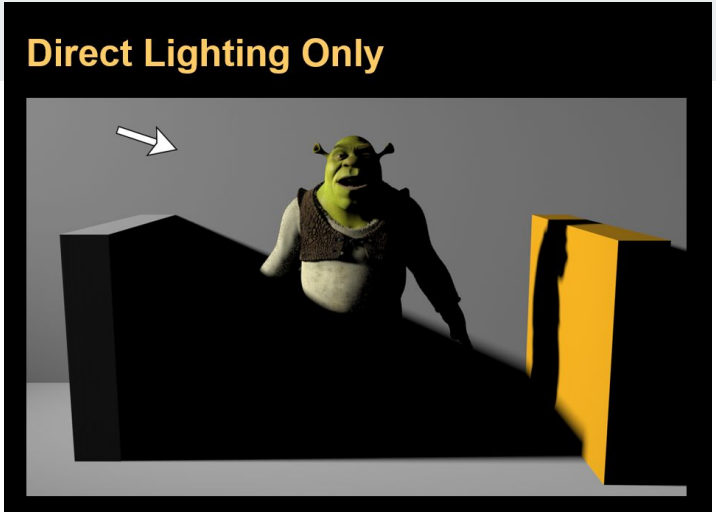
Approximation 2: Direct Light only

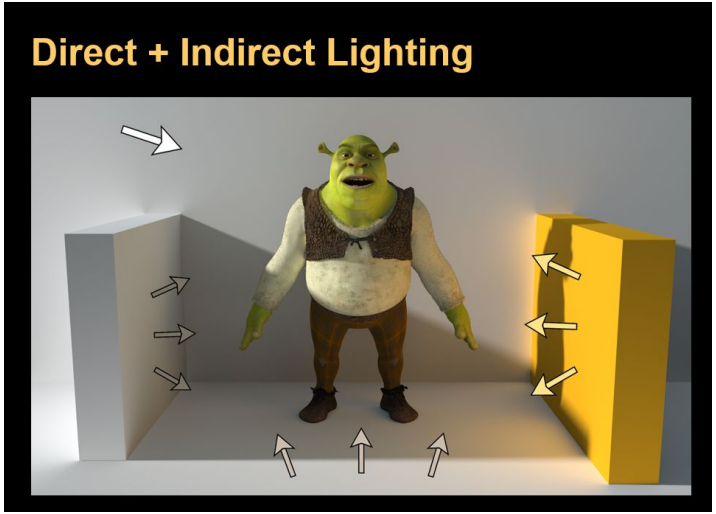
Only look in the directions of light sources.

Other directions can be approximated as a constant. (Indirect Lighting)

Used in games.

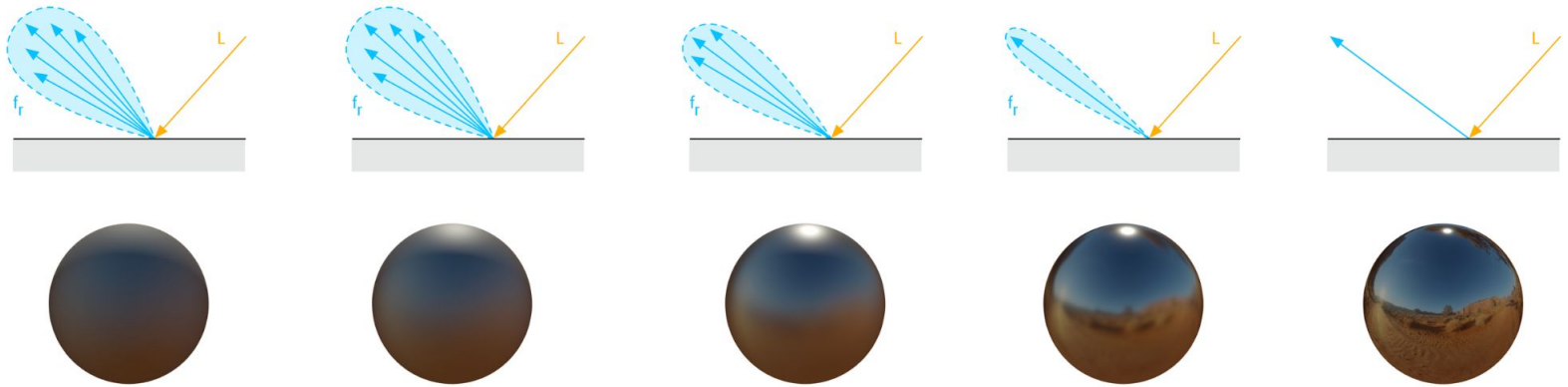
Some modern games use dynamic approximations of indirect lighting that seeks an acceptable trade off between speed and quality.

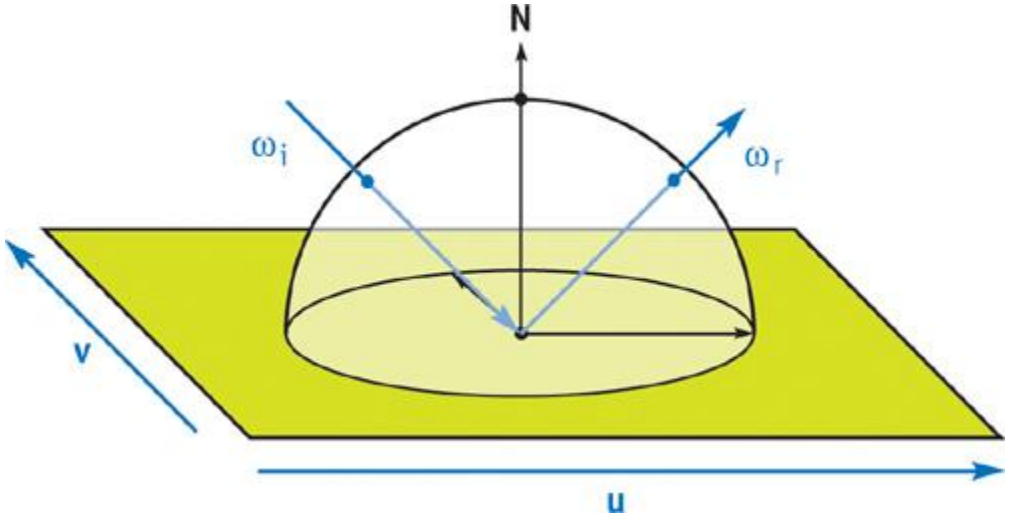


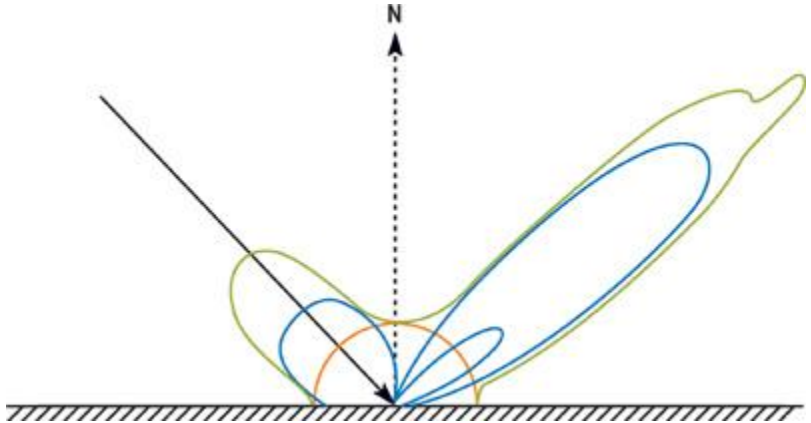


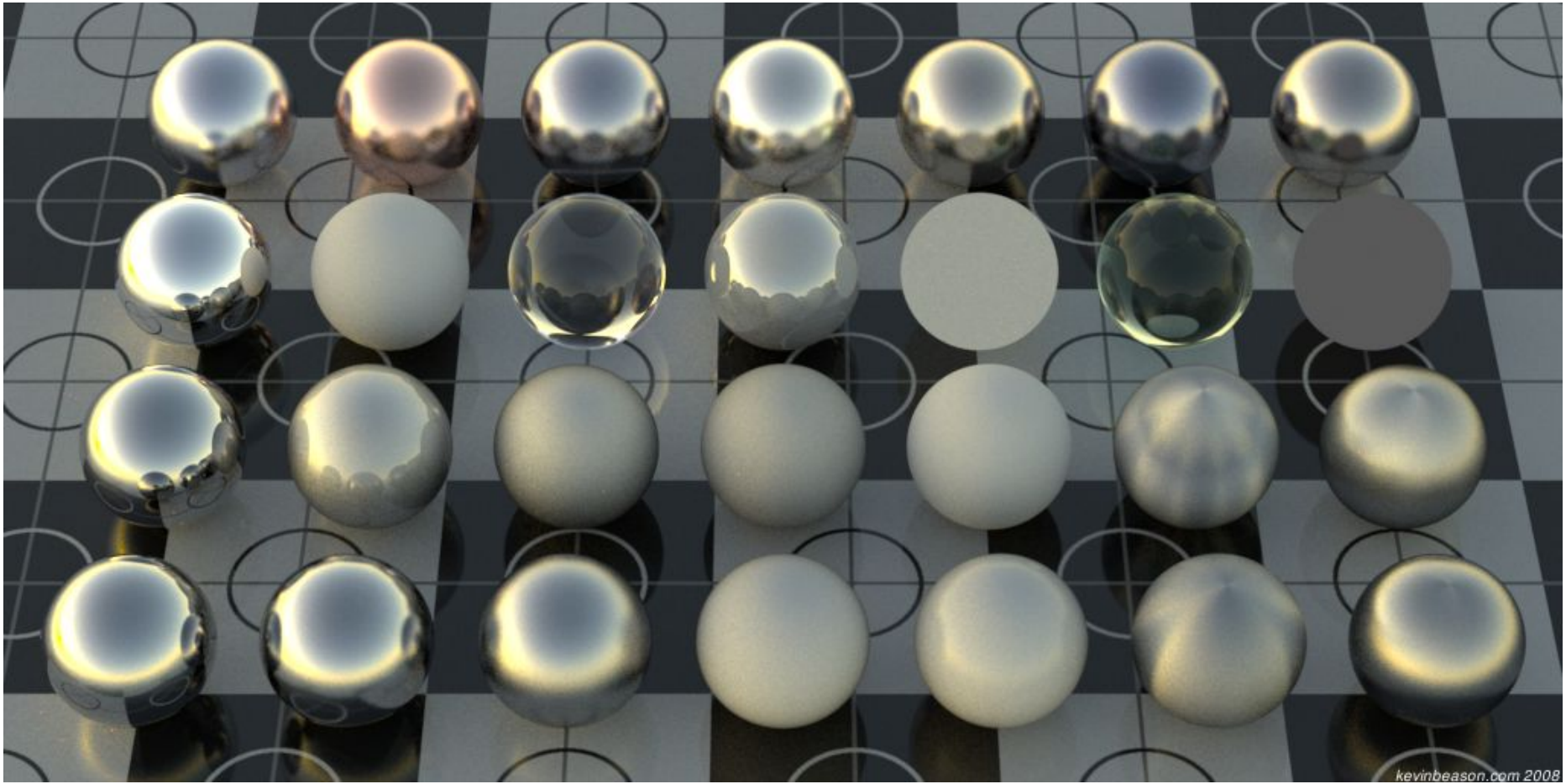
Bidirectional Reflectance Distribution Function (BRDF)

$$L_{\mathrm{o}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = L_{\mathrm{e}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$




N
ωi
ωr
v
u


N



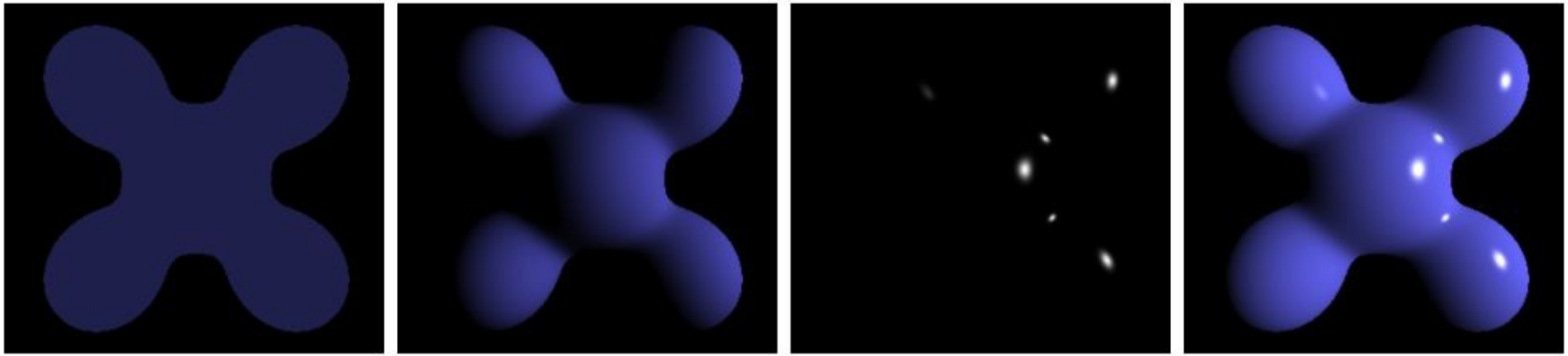
Same lighting conditions but different BRDFs

Phong Lighting Model

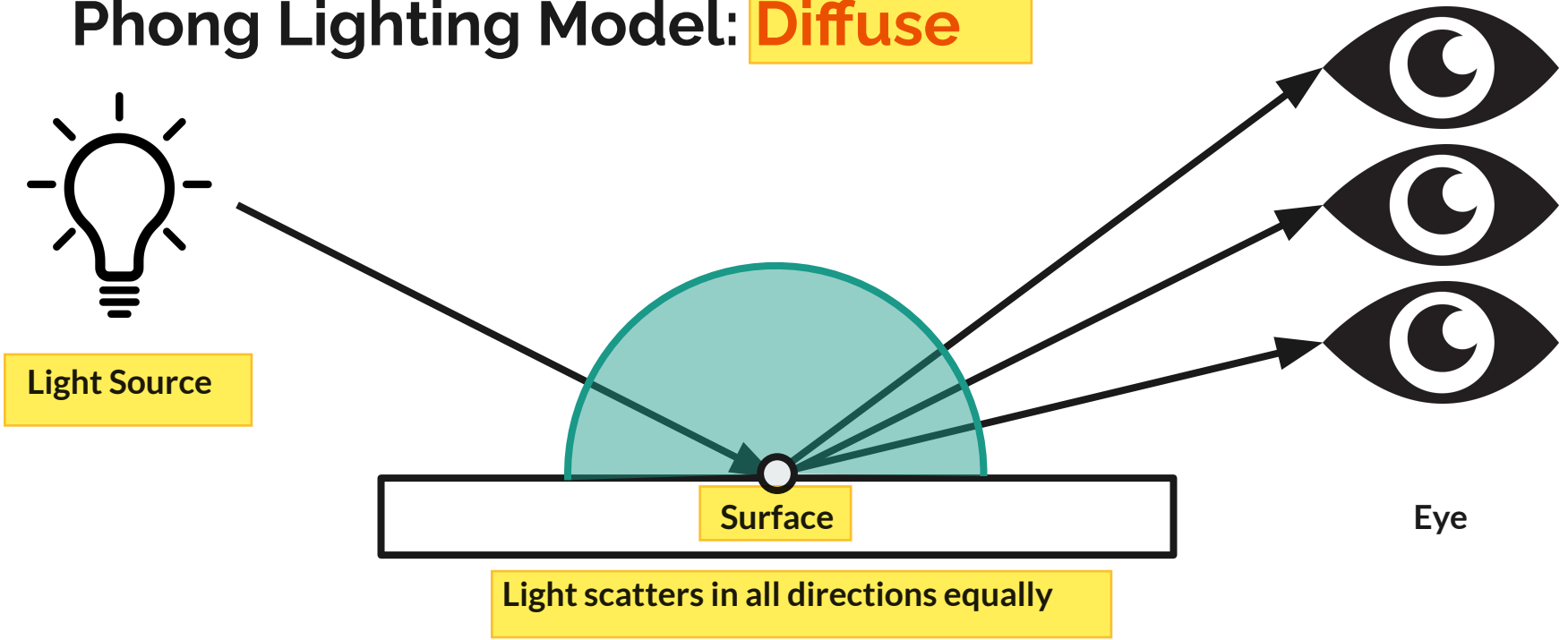
Developed by Bui Tuong Phong in 1975.

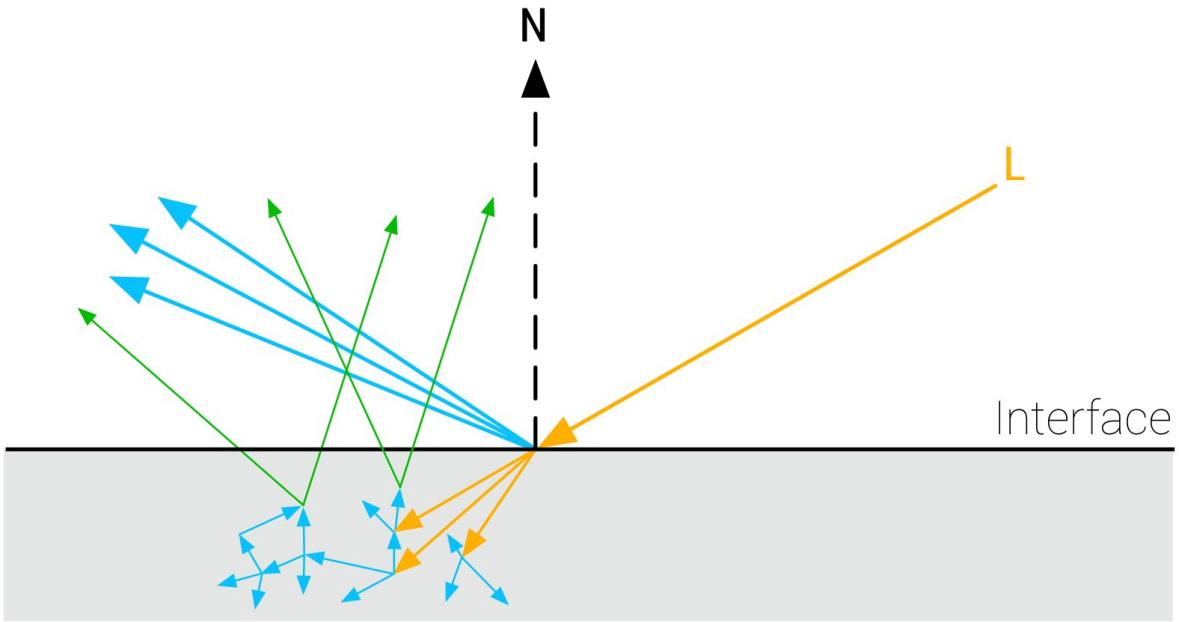
Computationally cheap and is still used in some games nowadays.

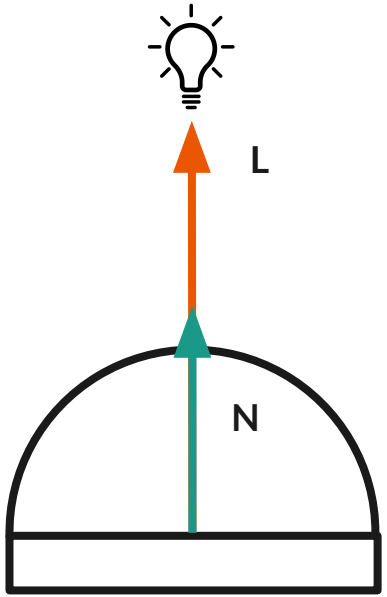
Does not adhere to the energy conservation principle.


Ambient
+
Diffuse
+
Specular
=
Phong Reflection

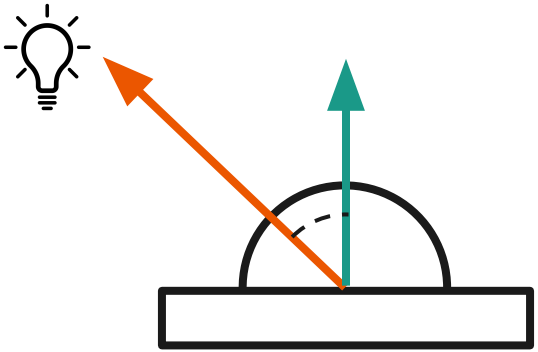
Phong Lighting Model: Diffuse



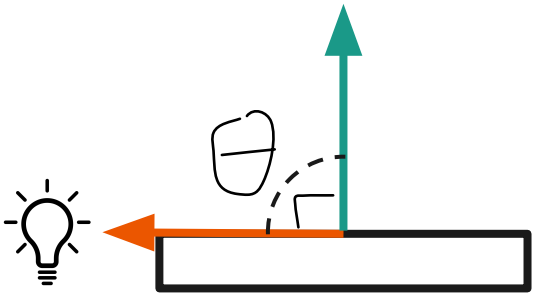

N
L
Interface



θ=0°



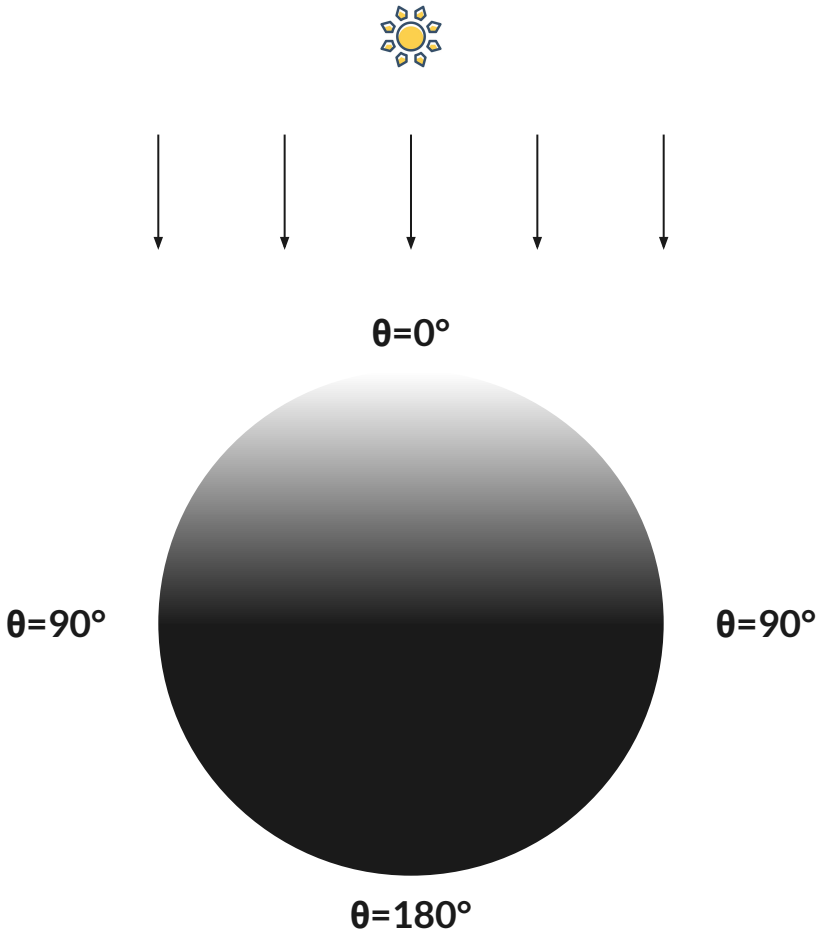
0°<θ<90°



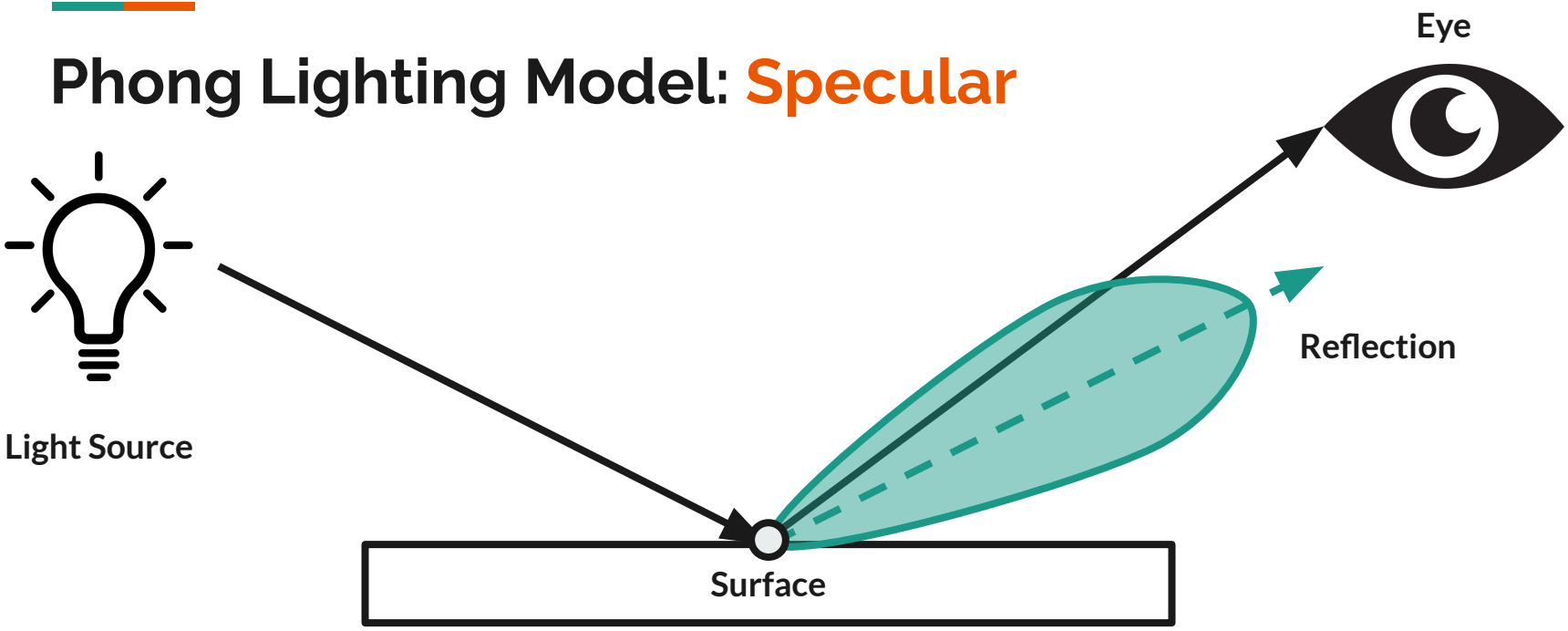
θ=90°

$$\text{Diffuse} = \max(0, \cos\theta) = \max(0, L \cdot N)$$

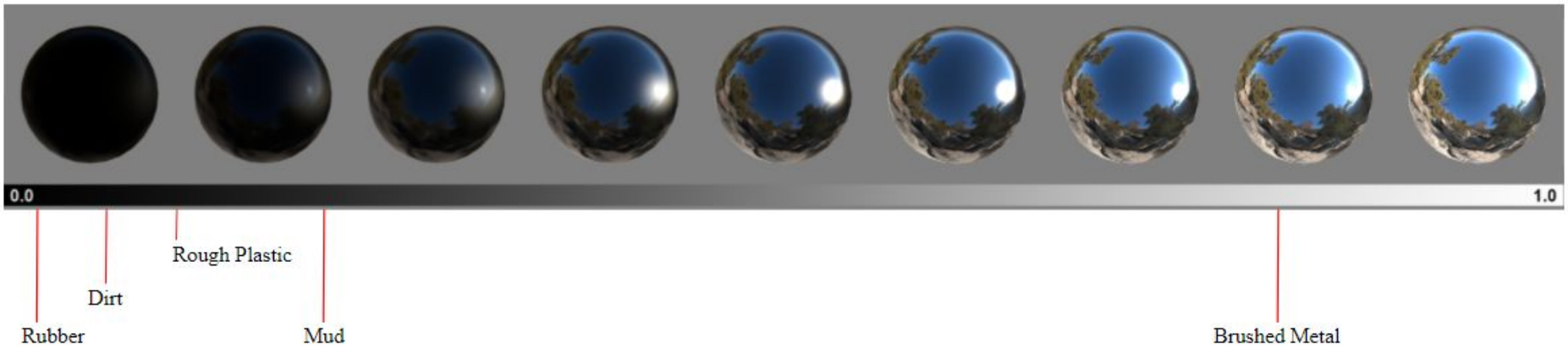
Lambert's cosine law

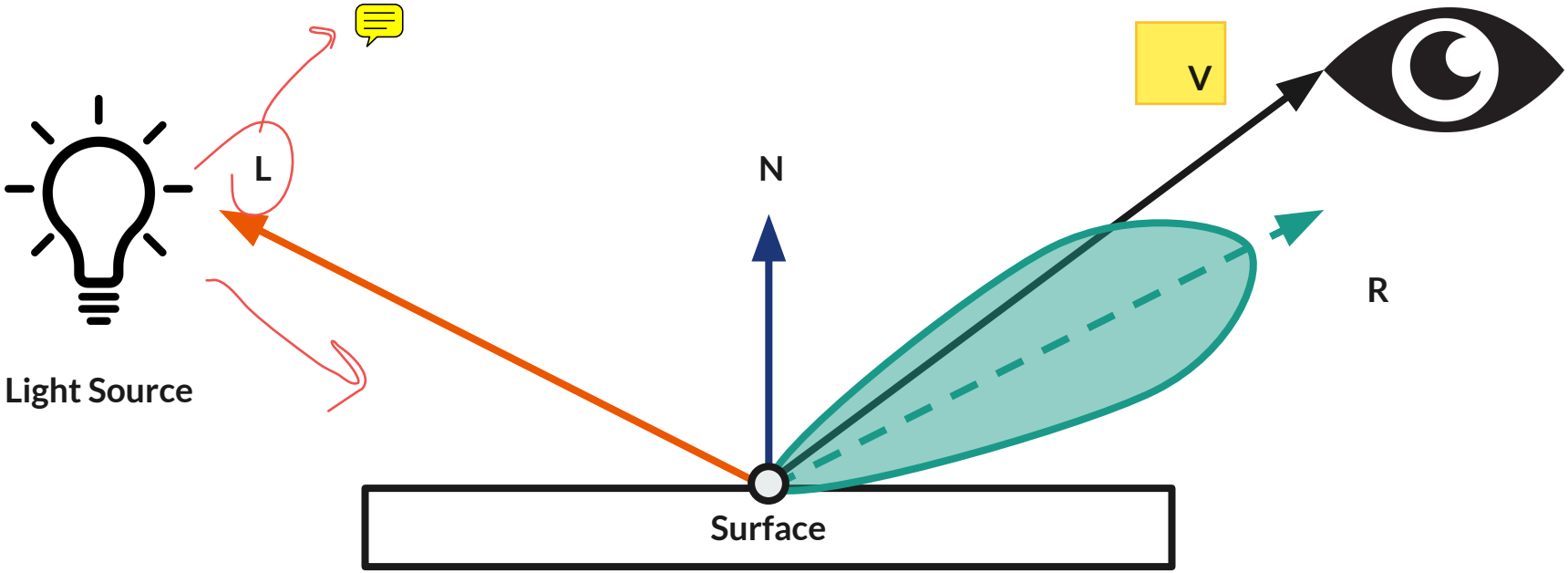

θ=0°
θ=90°
θ=90°
θ=180°

Phong Lighting Model: Specular



Light reflects in “almost” a single direction


0.0
1.0
Rough Plastic
Dirt
Rubber
Mud
Brushed Metal

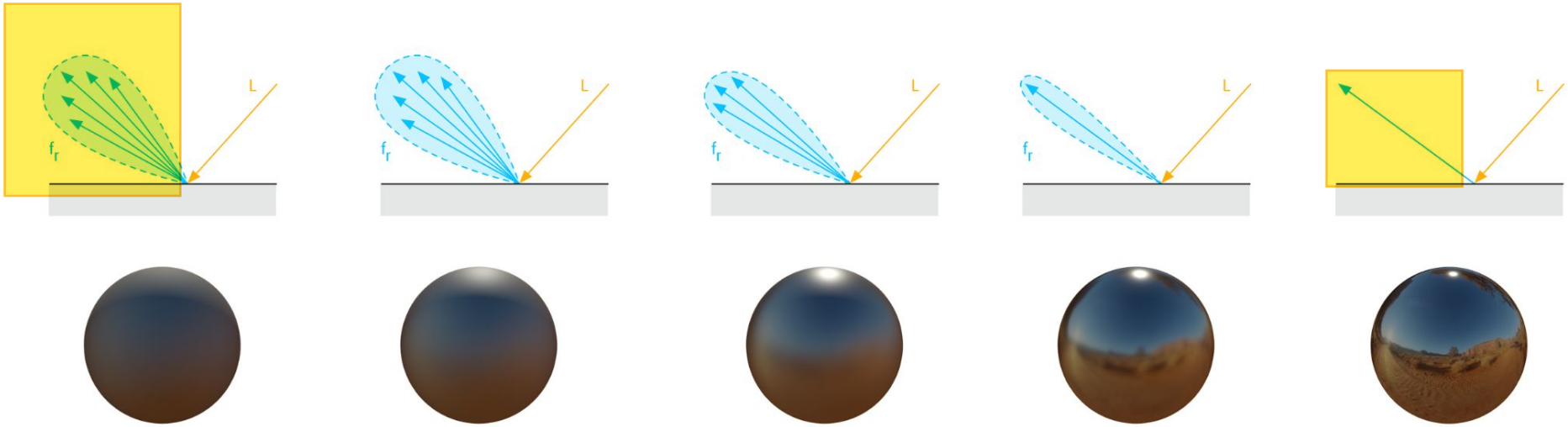


$$R = 2(N \cdot L)N - L$$

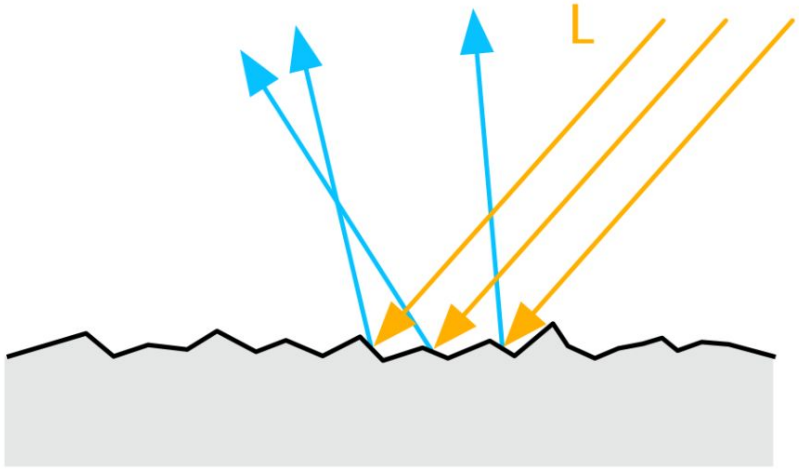
$$\text{Specular} = \max(0, V \cdot R)^{\alpha}$$

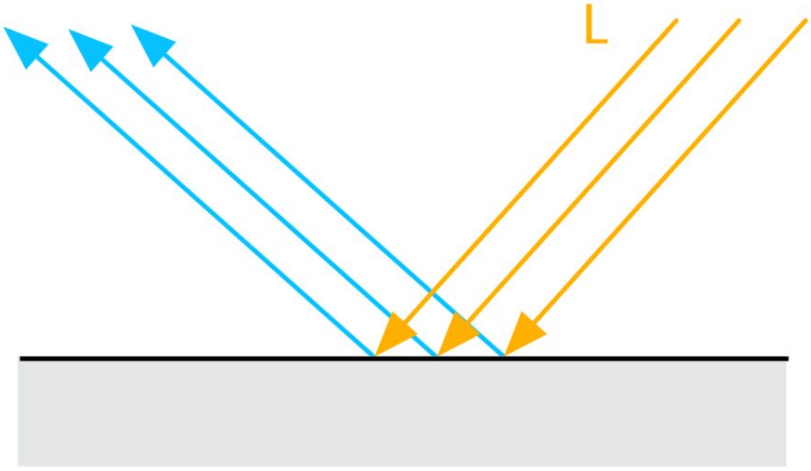
α is the shininess (specular power)

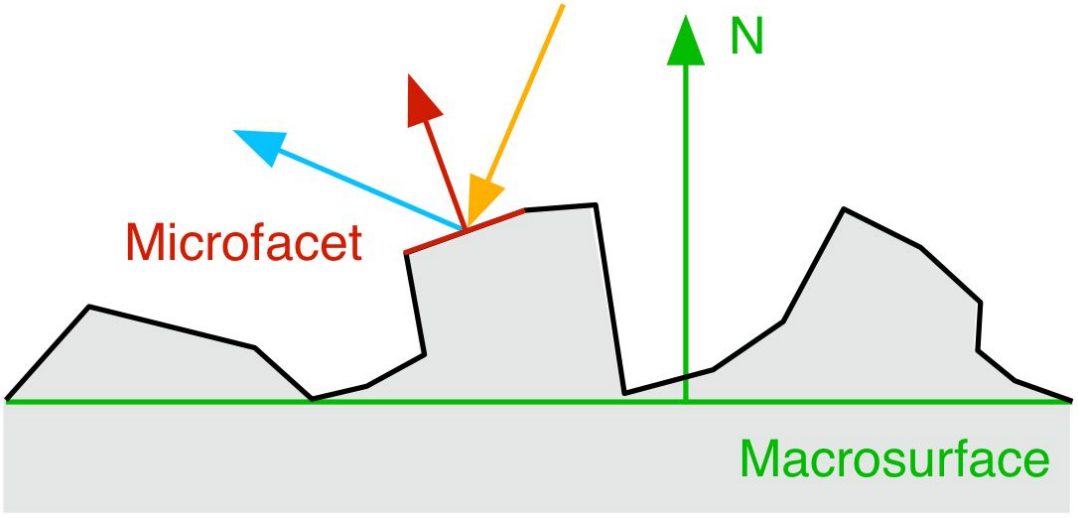
α increases



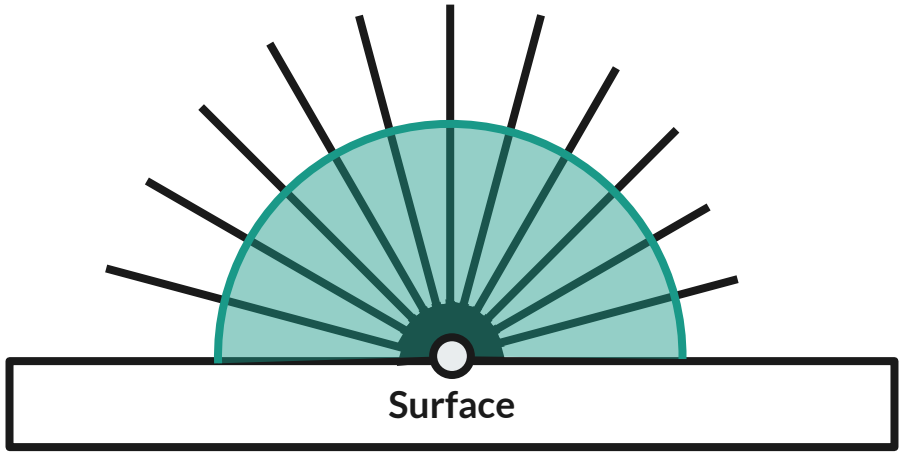
$0 < \alpha < \infty$


L

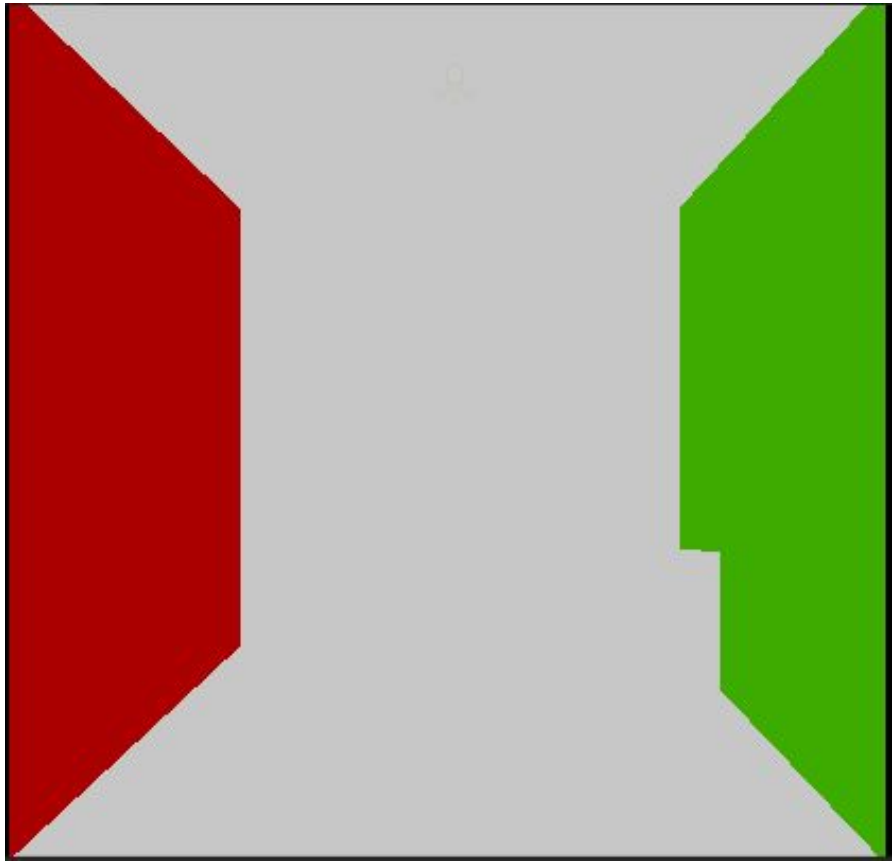

L

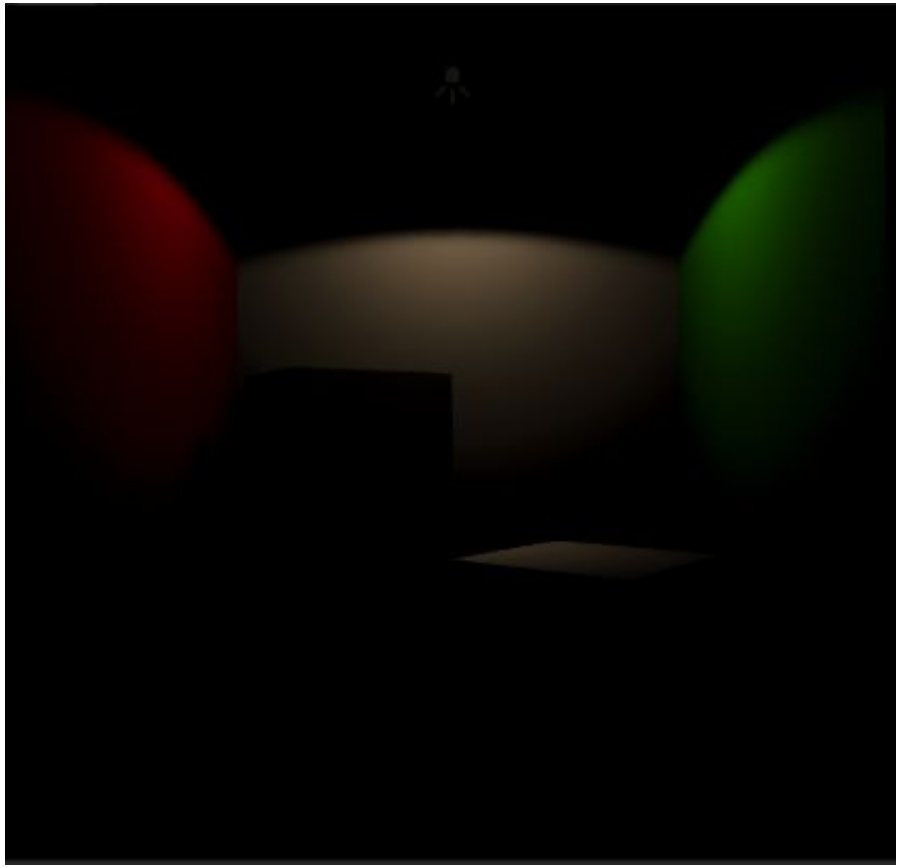

N
Microfacet
Macrosurface

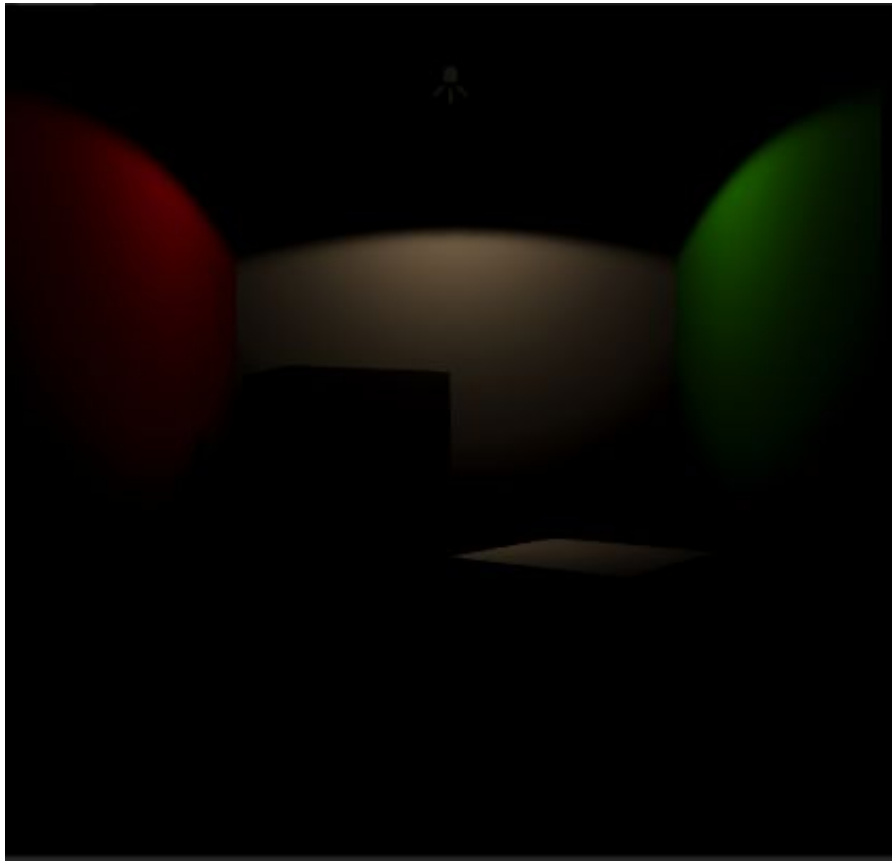
Phong Lighting Model: Ambient

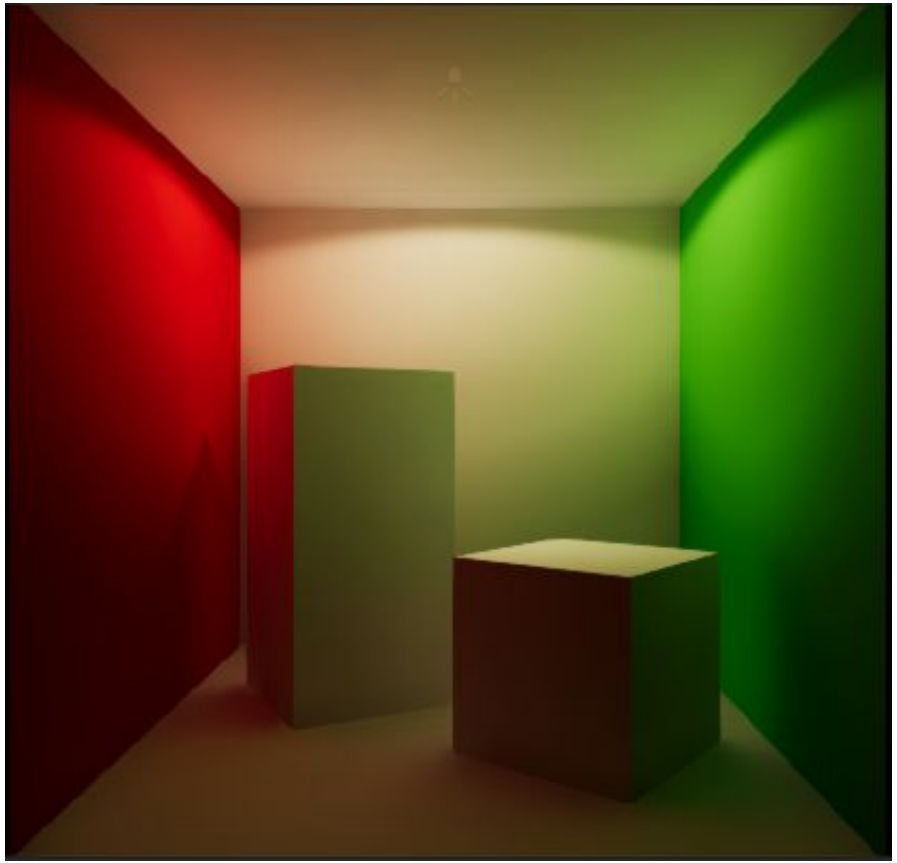


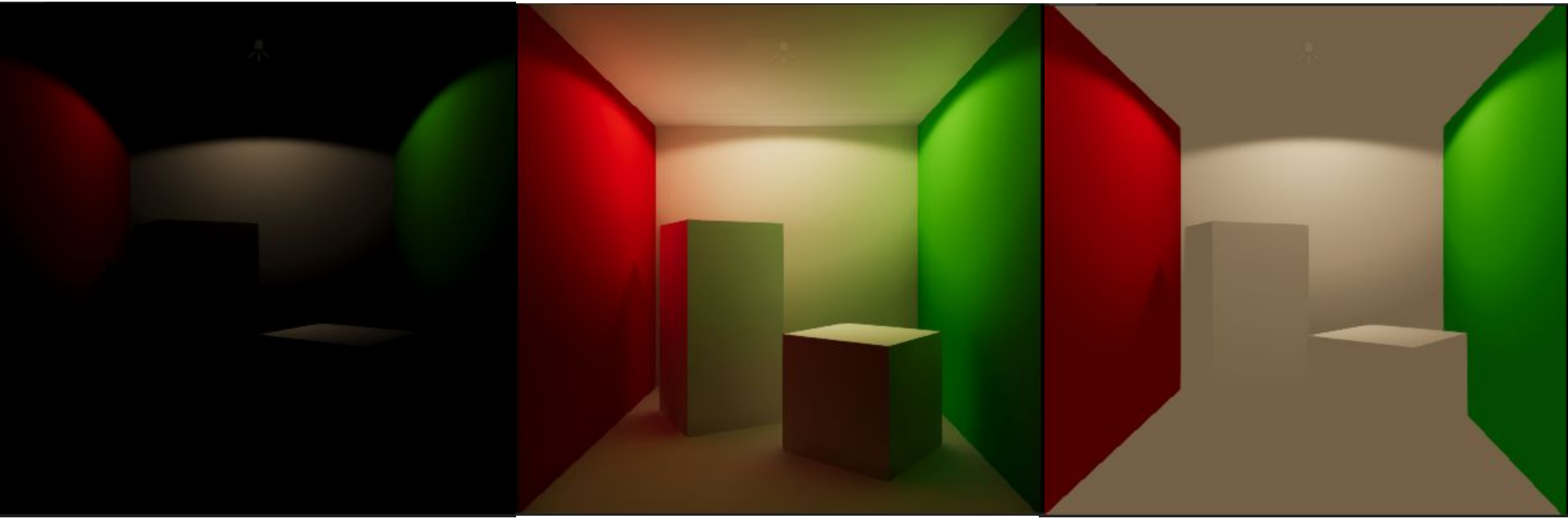
Light comes from all directions and scatters in all directions equally











Ambient = 1

$$\begin{aligned} \text{Color} = {} & M_{ambient}\, I_{ambient} + \\ & M_{diffuse}\, I_{diffuse}\, \max(0, L \cdot N) + \\ & M_{specular}\, I_{specular}\, \max(0, V \cdot R)^{\alpha} \end{aligned}$$

Where M and α are material properties
and I are light properties.

We have 3 colors for the material ambient , diffuse, specular

Diffuse

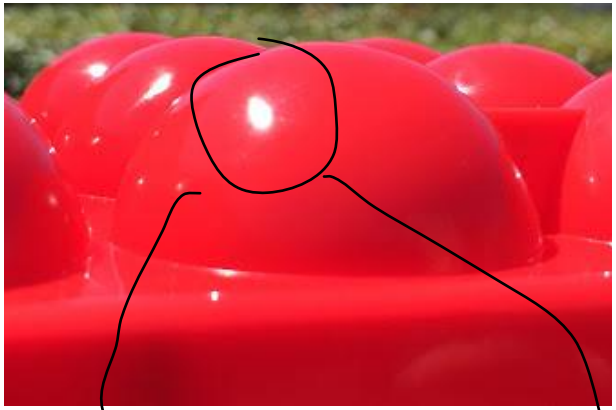
Specular

Shininess Low

Diffuse

Specular

Shininess High

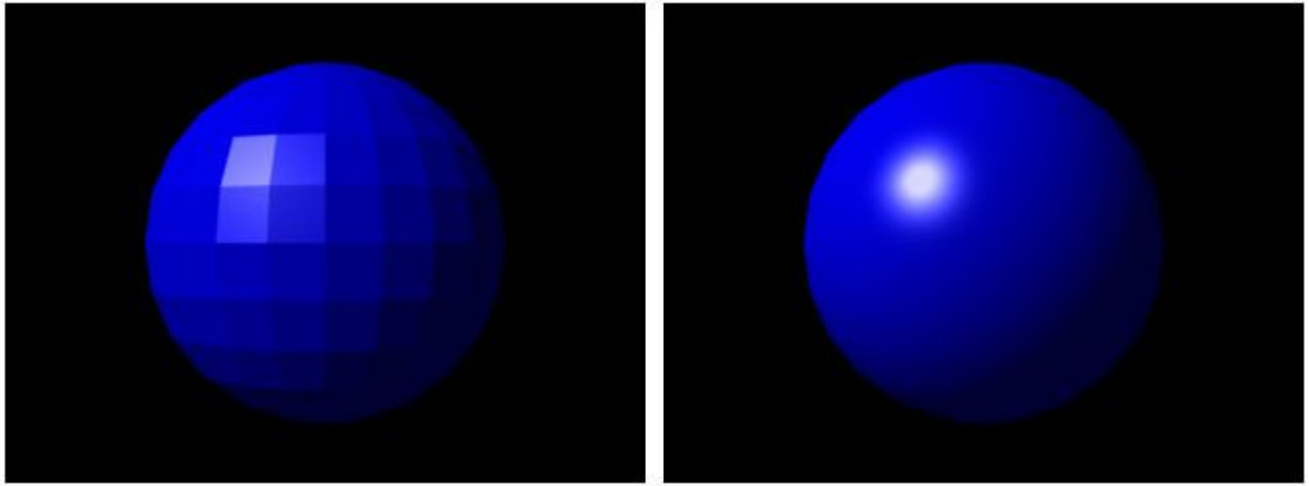


Diffuse

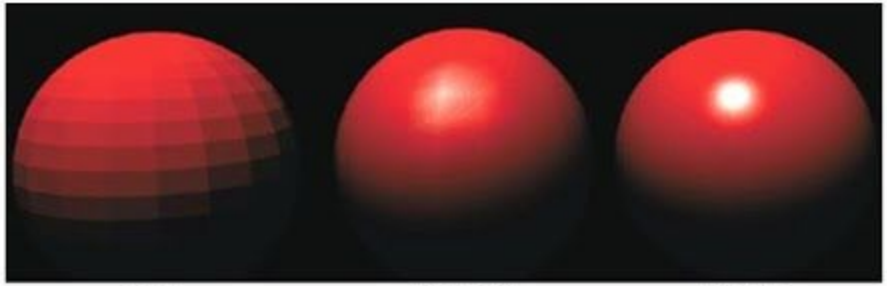
Specular

Shininess High

Flat shading vs phong shading

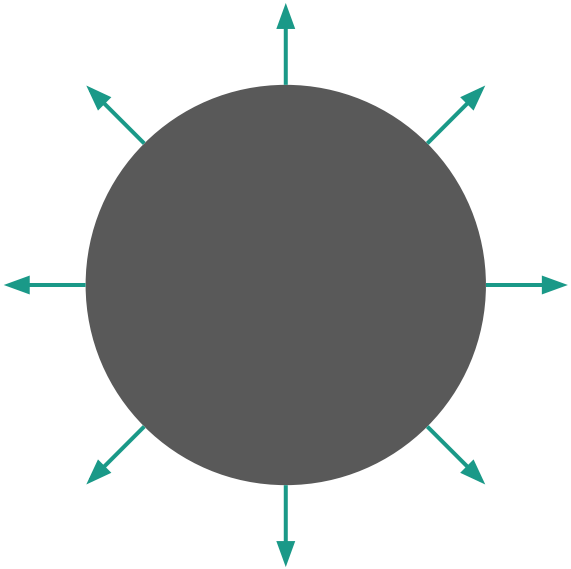


FLAT SHADING, GOURAUD SHADING & PHONG SHADING



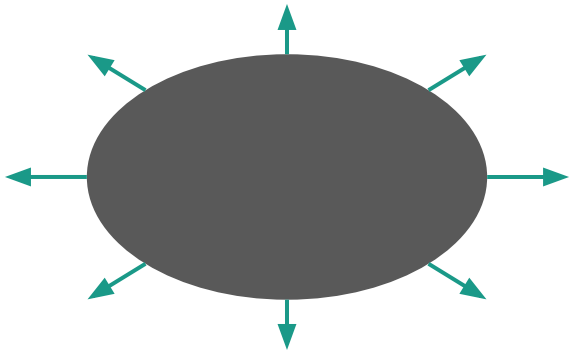
```
layout(location = 0) in vec3 position;
layout(location = 1) in vec4 color;
layout(location = 2) in vec2 tex_coord;
layout(location = 3) in vec3 normal;

uniform mat4 object_to_world;
uniform mat4 object_to_world_inv_transpose;
uniform mat4 view_projection;
uniform vec3 camera_position;

out Varyings {
   vec4 color;
   vec2 tex_coord;
   vec3 world;
   vec3 view;
   vec3 normal;
} vsout;

void main() {
   vsout.world = (object_to_world * vec4(position, 1.0f)).xyz;
   vsout.view = camera_position - vsout.world;
   vsout.normal = normalize((object_to_world_inv_transpose * vec4(normal, 0.0f)).xyz);
   gl_Position = view_projection * vec4(vsout.world, 1.0);
   vsout.color = color;
   vsout.tex_coord = tex_coord;
}
```

This problem is only relevant when using non-uniform scaling



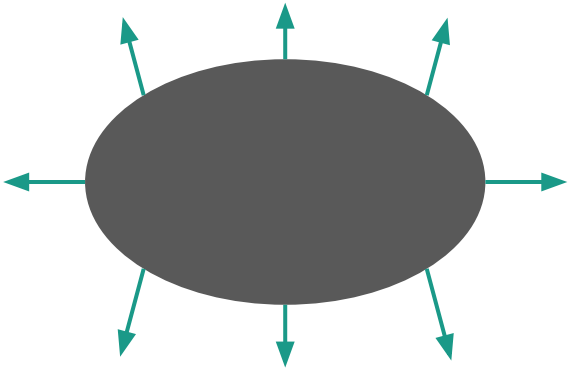
N_{local}

WRONG



$N_{world} = M * N_{local}$

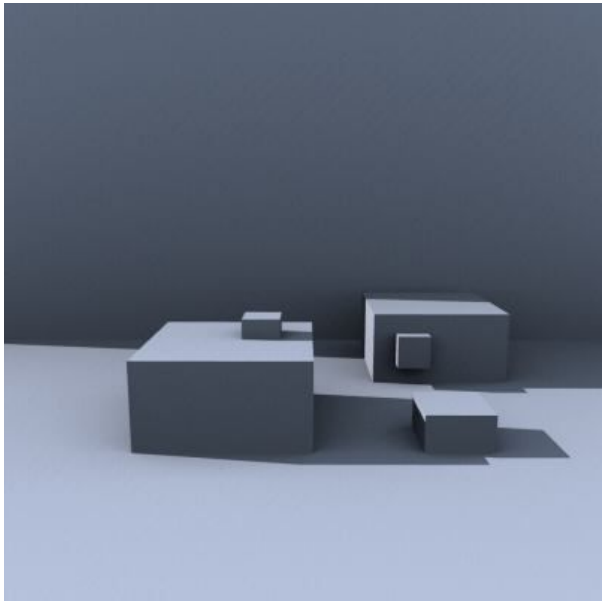
RIGHT



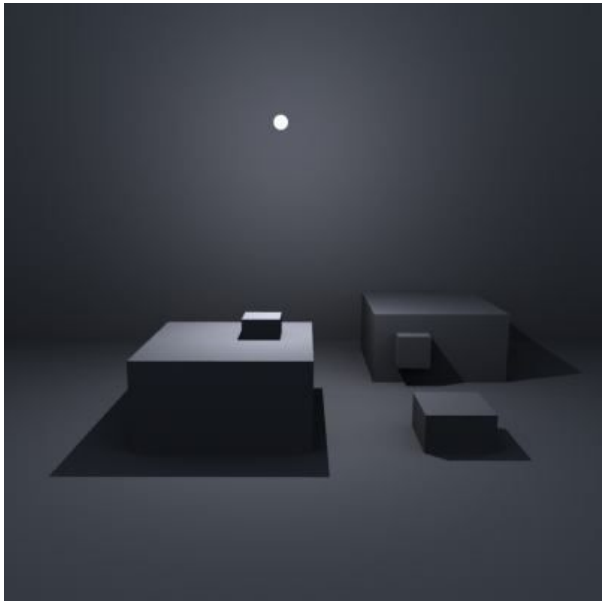
$N_{world} = (M^{-1})^{T} * N_{local}$

```
in Varyings {
    vec4 color;
    vec2 tex_coord;
    vec3 world;
    vec3 view;
    vec3 normal;
} fsin;

struct DirectionalLight {
    vec3 diffuse;
    vec3 specular;
    vec3 ambient;
    vec3 direction;
};

struct Material {
    vec3 diffuse;
    vec3 specular;
    vec3 ambient;
    float shininess;
};

uniform Material material;
uniform DirectionalLight light;

out vec4 frag_color;
```

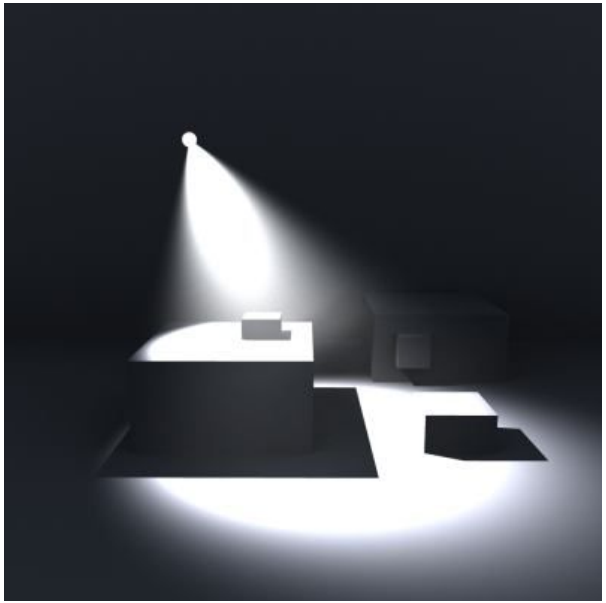
```
void main() {
   vec3 normal = normalize(fsin.normal);
   vec3 view = normalize(fsin.view);
   vec3 reflected = reflect(light.direction, normal);

   float lambert = max(0.0f, dot(normal, -light.direction));
   float phong = pow(max(0.0f, dot(view, reflected)), material.shininess);

   vec3 diffuse = material.diffuse * light.diffuse * lambert;
   vec3 specular = material.specular * light.specular * phong;
   vec3 ambient = material.ambient * light.ambient;

   frag_color = fsin.color * vec4(diffuse + specular + ambient, 1.0f);
}
```



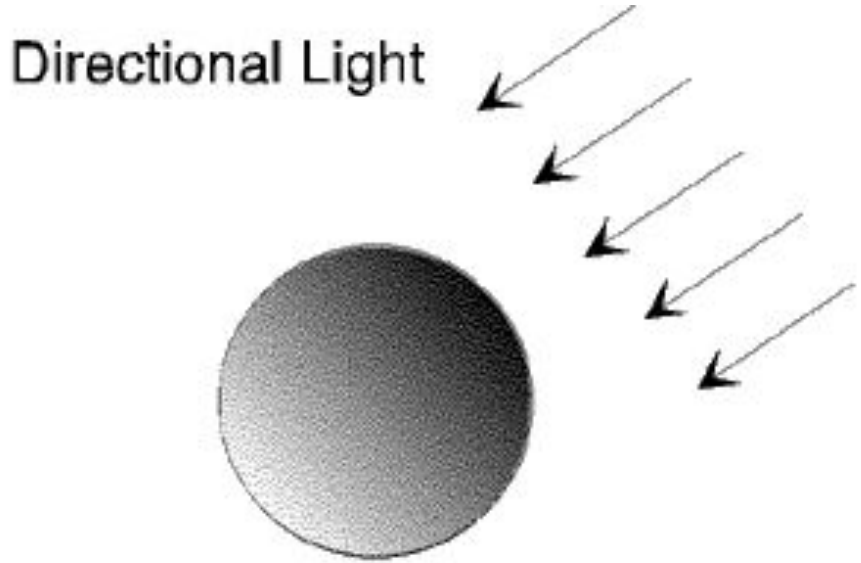
Directional Light

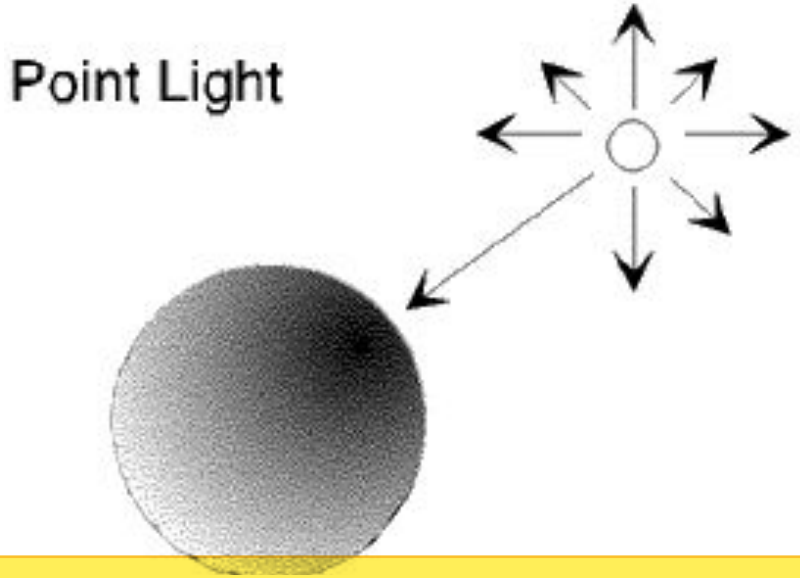


Point Light



Spot Light

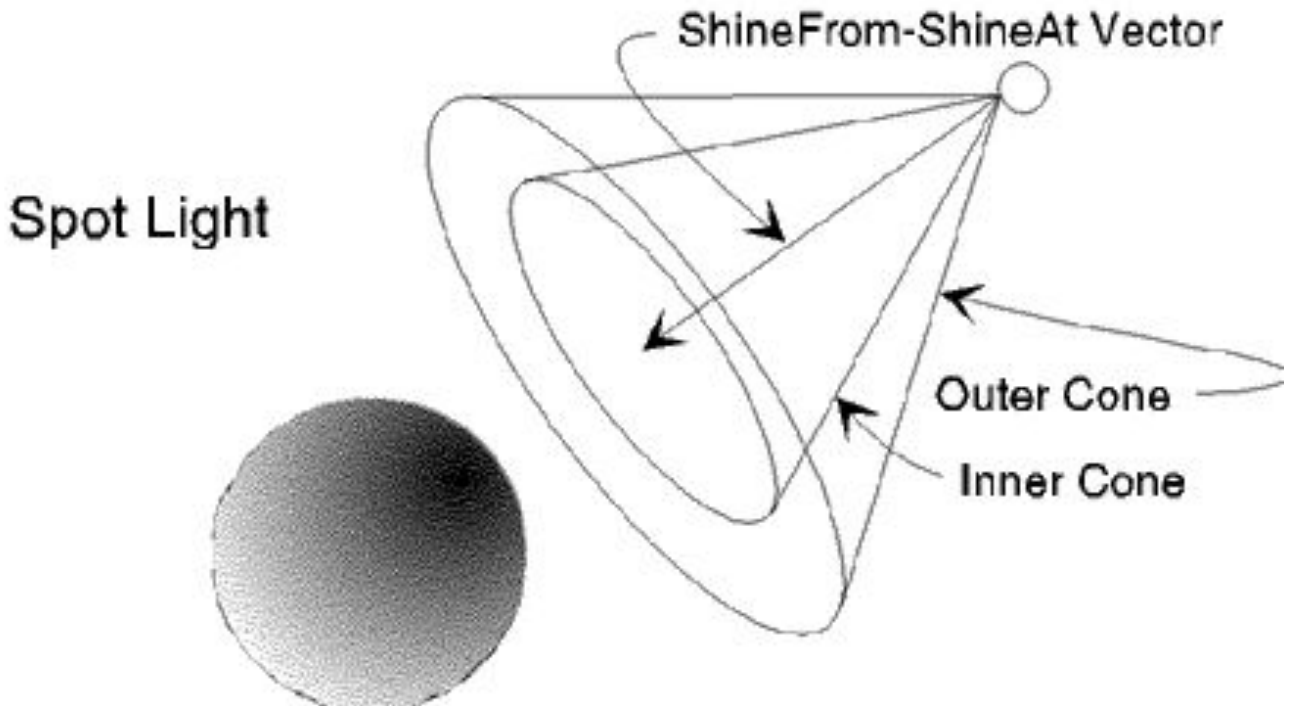

Directional Light



$$\text{int}\,ensity = \frac{1}{c + ld + qd^2}$$

```
in Varyings {
    vec4 color;
    vec2 tex_coord;
    vec3 world;
    vec3 view;
    vec3 normal;
} fsin;

struct PointLight {
    vec3 diffuse;
    vec3 specular;
    vec3 ambient;
    vec3 position;
    float attenuation_constant;
    float attenuation_linear;
    float attenuation_quadratic;
};

struct Material {
    vec3 diffuse;
    vec3 specular;
    vec3 ambient;
    float shininess;
};

uniform Material material;
uniform PointLight light;
```

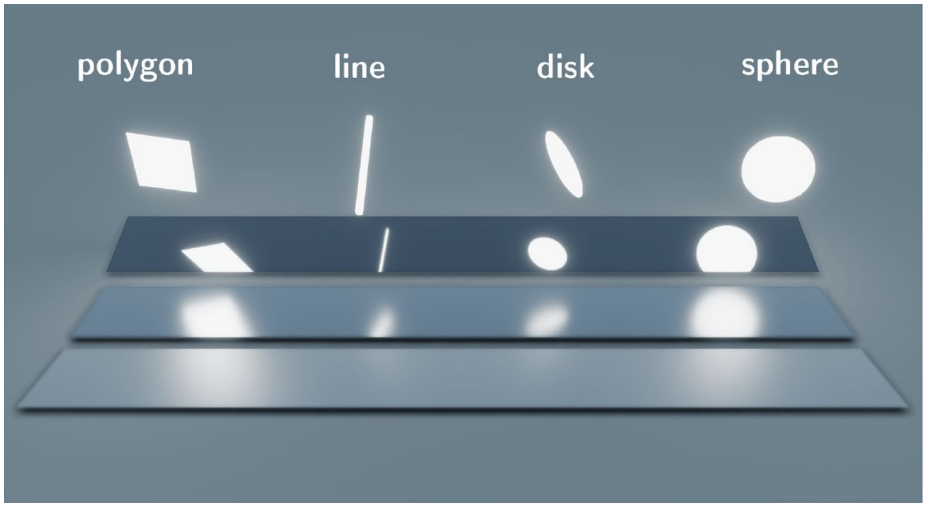
```
out vec4 frag_color;

void main() {
   vec3 normal = normalize(fsin.normal);
   vec3 view = normalize(fsin.view);

   vec3 light_direction = fsin.world - light.position;
   float distance = length(light_direction);
   light_direction /= distance;

   float attenuation = 1.0f / (light.attenuation_constant +
                        light.attenuation_linear * distance +
                        light.attenuation_quadratic * distance * distance);

   vec3 reflected = reflect(light_direction, normal);

   float lambert = max(0.0f, dot(normal, -light_direction));
   float phong = pow(max(0.0f, dot(view, reflected)), material.shininess);

   vec3 diffuse = material.diffuse * light.diffuse * lambert;
   vec3 specular = material.specular * light.specular * phong;
   vec3 ambient = material.ambient * light.ambient;

   frag_color = fsin.color * vec4((diffuse + specular) * attenuation + ambient, 1.0f);
}
```


ShineFrom-ShineAt Vector
Spot Light
Outer Cone
Inner Cone

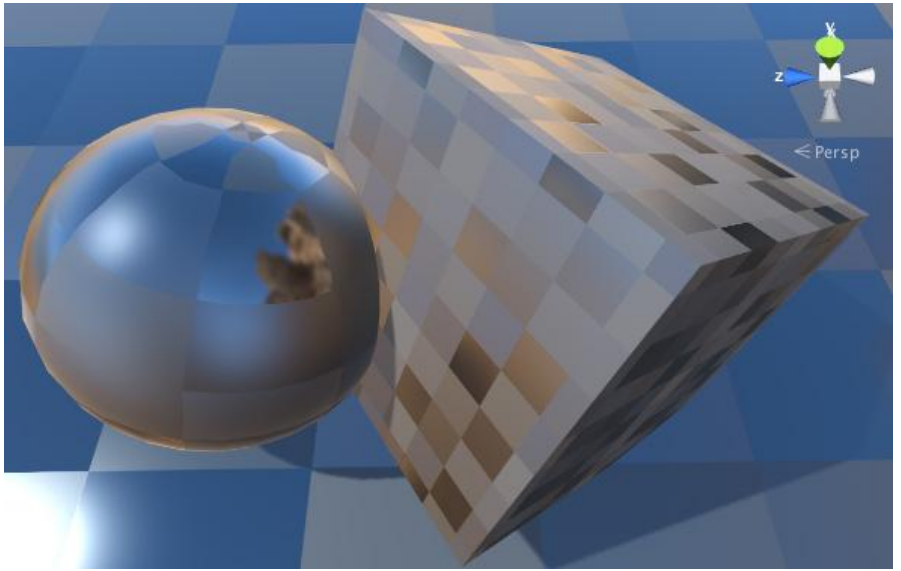
```
in Varyings {
   vec4 color;
   vec2 tex_coord;
   vec3 world;
   vec3 view;
   vec3 normal;
} fsin;

struct SpotLight {
   vec3 diffuse;
   vec3 specular;
   vec3 ambient;
   vec3 position, direction;
   float attenuation_constant;
   float attenuation_linear;
   float attenuation_quadratic;
   float inner_angle, outer_angle;
};

struct Material {
   vec3 diffuse;
   vec3 specular;
   vec3 ambient;
   float shininess;
};

uniform Material material;
uniform SpotLight light;
```

```
out vec4 frag_color;
void main() {
   vec3 normal = normalize(fsin.normal);
   vec3 view = normalize(fsin.view);

   vec3 light_direction = fsin.world - light.position;
   float distance = length(light_direction);
   light_direction /= distance;

   float attenuation = 1.0f / (light.attenuation_constant +
                        light.attenuation_linear * distance +
                        light.attenuation_quadratic * distance * distance);

   float angle = acos(dot(light.direction, light_direction));
   attenuation *= smoothstep(light.outer_angle, light.inner_angle, angle);

   vec3 reflected = reflect(light_direction, normal);

   float lambert = max(0.0f, dot(normal, -light_direction));
   float phong = pow(max(0.0f, dot(view, reflected)), material.shininess);

   vec3 diffuse = material.diffuse * light.diffuse * lambert;
   vec3 specular = material.specular * light.specular * phong;
   vec3 ambient = material.ambient * light.ambient;

   frag_color = fsin.color * vec4((diffuse + specular) * attenuation + ambient, 1.0f);
}
```

Some Other Light Types



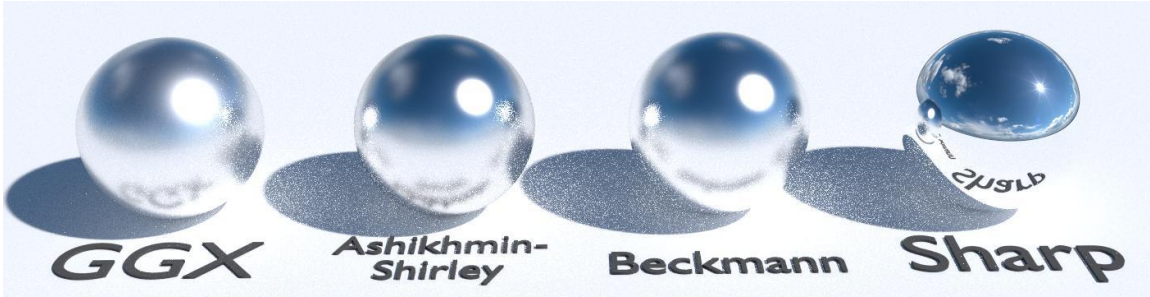
Area, Line and Sphere Lights

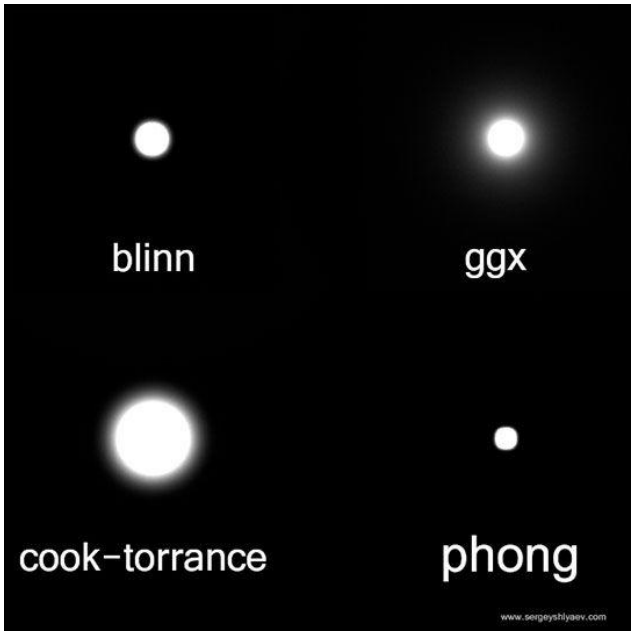


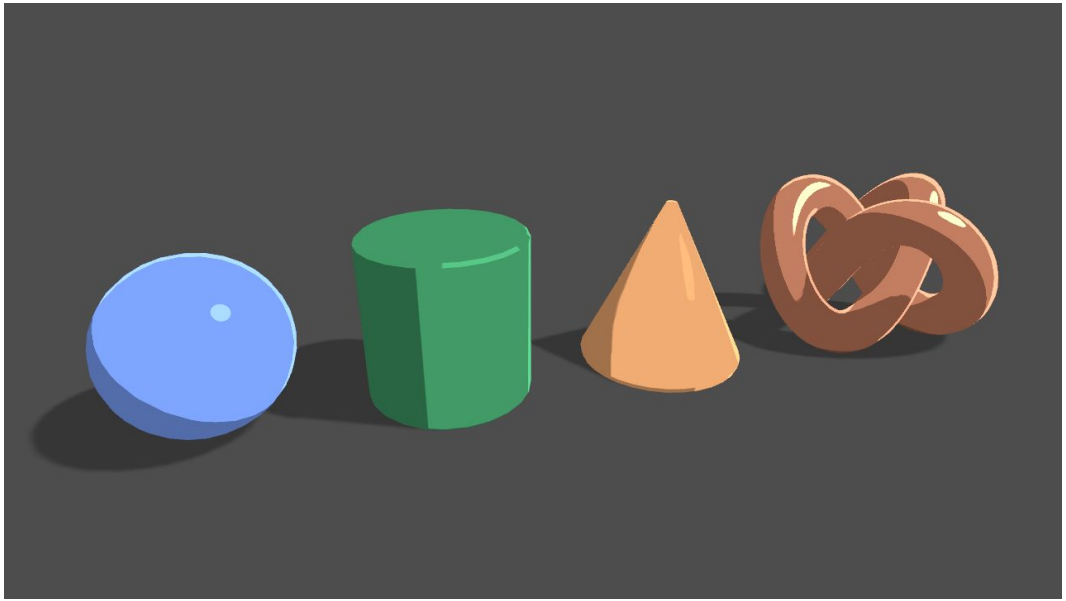
Reflection Probes (Image Based Lighting)

not importnat

Other BRDFs







How to render scenes with multiple lights!!!

Approach 1: Single Pass Forward Lighting

- Send all lights to shader as an array.
- Loop over array and shaders and sum the result of all the lights.

```
in Varyings {
   vec4 color;
   vec2 tex_coord;
   vec3 world;
   vec3 view;
   vec3 normal;
} fsin;

#define TYPE_DIRECTIONAL    0
#define TYPE_POINT          1
#define TYPE_SPOT           2

struct Light {
   int type;
   vec3 diffuse;
   vec3 specular;
   vec3 ambient;
   vec3 position, direction;
   float attenuation_constant;
   float attenuation_linear;
   float attenuation_quadratic;
   float inner_angle, outer_angle;
};
```

```
struct Material {
    vec3 diffuse;
    vec3 specular;
    vec3 ambient;
    float shininess;
};

uniform Material material;

#define MAX_LIGHT_COUNT 16
uniform Light lights[MAX_LIGHT_COUNT];
uniform int light_count;

out vec4 frag_color;

void main() {
    vec3 normal = normalize(fsin.normal);
    vec3 view = normalize(fsin.view);

    int count = min(light_count, MAX_LIGHT_COUNT);
    vec3 accumulated_light = vec3(0.0);
```

```
    for(int index = 0; index < count; index++){
        Light light = lights[index];
        vec3 light_direction;
        float attenuation = 1;
        if(light.type == TYPE_DIRECTIONAL)
            light_direction = light.direction;
        else {
            light_direction = fsin.world - light.position;
            float distance = length(light_direction);
            light_direction /= distance;
            attenuation *= 1.0f / (light.attenuation_constant +
                                   light.attenuation_linear * distance +
                                   light.attenuation_quadratic * distance * distance);
            if(light.type == TYPE_SPOT){
                float angle = acos(dot(light.direction, light_direction));
                attenuation *= smoothstep(light.outer_angle, light.inner_angle, angle);
            }
        }
        vec3 reflected = reflect(light_direction, normal);
        float lambert = max(0.0f, dot(normal, -light_direction));
        float phong = pow(max(0.0f, dot(view, reflected)), material.shininess);
        vec3 diffuse = material.diffuse * light.diffuse * lambert;
        vec3 specular = material.specular * light.specular * phong;
        vec3 ambient = material.ambient * light.ambient;
        accumulated_light += (diffuse + specular) * attenuation + ambient;
    }
    frag_color = fsin.color * vec4(accumulated_light, 1.0f);
}
```

Approach 1: Single Pass Forward Lighting

- PROS:
  - Only one draw call to render an object with all the lights.
  - Fast for small number of lights.
- CONS:
  - Limited to the amount of lights defined in the shaders.
  - Slow for large numbers of lights.
  - Shader is complex since it has to handle all types of lights.

```
not important
```

Approach 2: Multi-Pass Forward Lighting

- For each object:
  - For each light:
    - If this is not the first light, enable blending and set it to be additive.
    - Draw object using the shader for the current light.

OR

- For each light:
  - If this is not the first light, enable blending and set it to be additive.
  - Draw all objects using the shader for the current light.

```
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE, GL_ONE);
```

```
if(first_light){
    glDisable(GL_BLEND);
    first_light = false;
} else {
    glEnable(GL_BLEND);
}
```

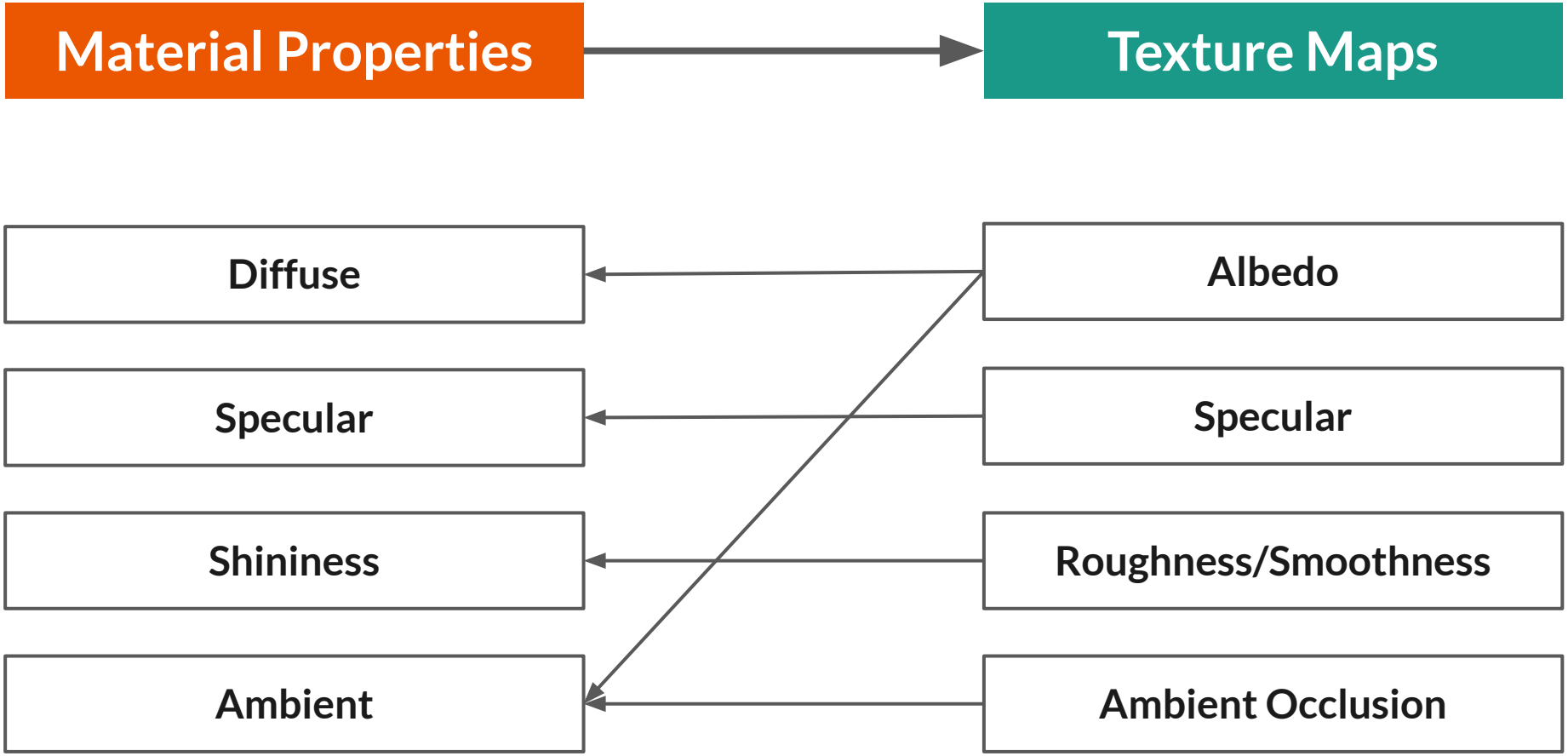
Approach 2: Multi-Pass Forward Lighting

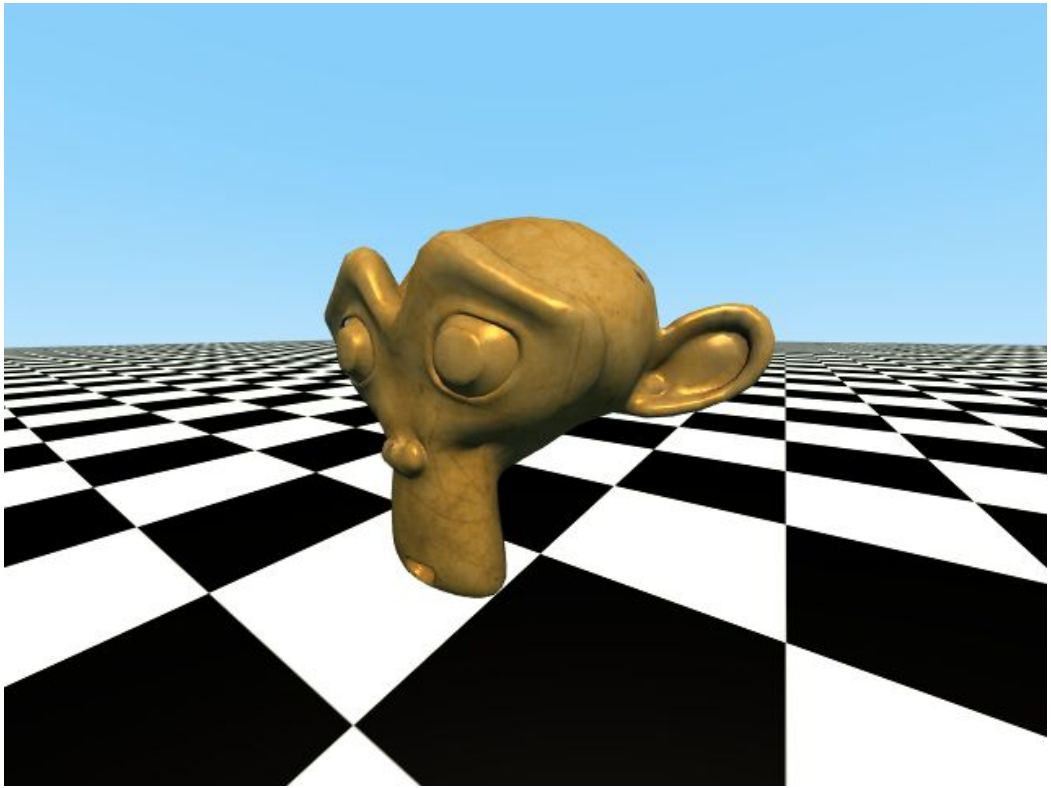
- PROS:
  - Not limited by the amount of lights defined in the shaders.
  - Fast for small number of lights.
- CONS:
  - Repeats the vertex processing. Slow for objects with lots of vertices.
  - Slow for large numbers of lights.
  - Needs lots of state changing and draw calls.

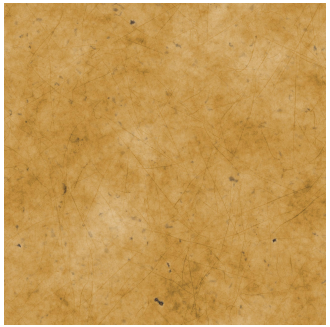
Other Approaches

- Deferred Rendering
  - Renders objects to G-Buffers to store their material properties and data needed for lighting.
  - Render each light using the G-Buffers to the accumulation buffer.
- Light Prepass
  - Render object depth and normal to a buffer.
  - Render lights to a light buffer.
  - Render objects again and use the lighting stored in the light buffer.
- Tiled/Clustered Forward
  - Render object depth and mark the tiles/clusters they intersect.
  - Store lights into a linked list for each relevant tiles/clusters that intersect with light.
  - Render objects and only use lights from their tile/cluster.

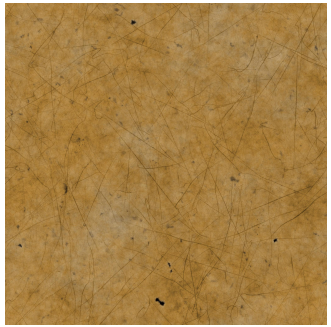
How to compute light for objects with textures.


Material Properties
Texture Maps
Diffuse
Specular
Shininess
Ambient
Albedo
Specular
Roughness/Smoothness
Ambient Occlusion

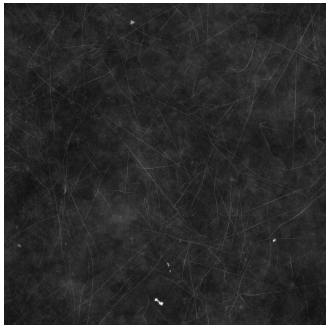




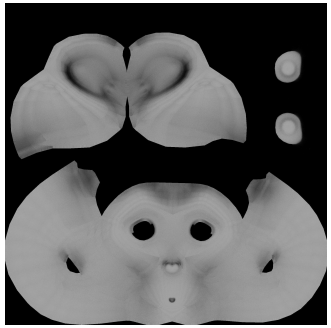
Albedo



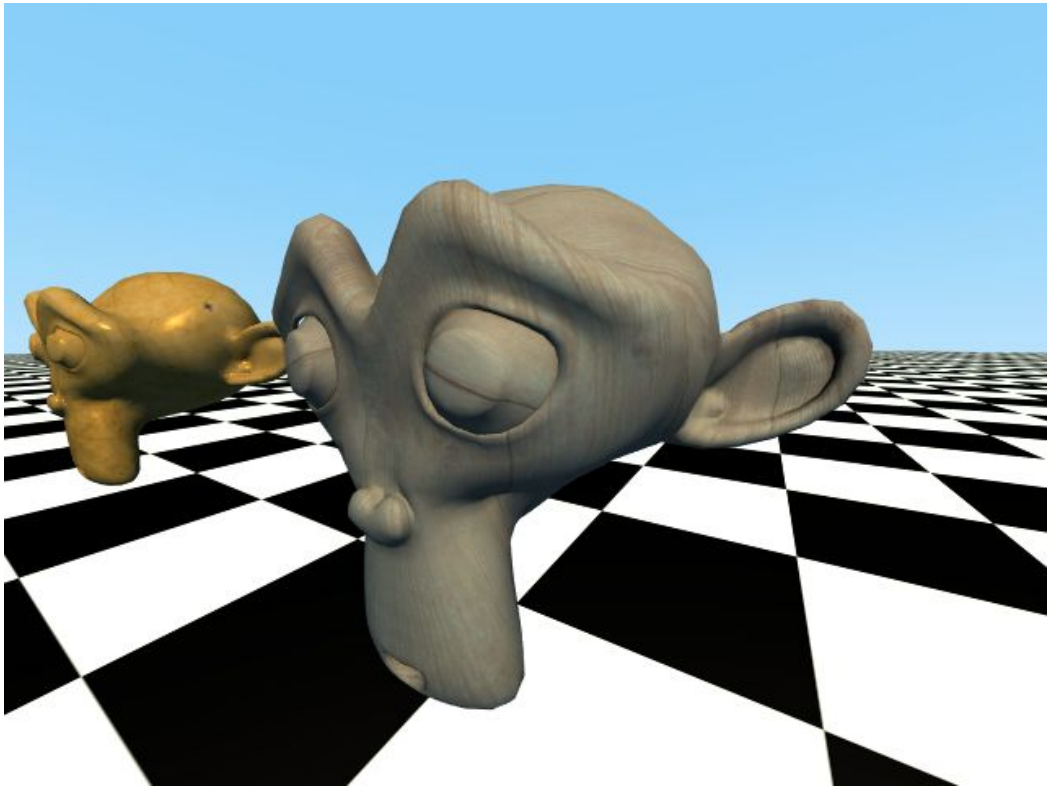
Specular



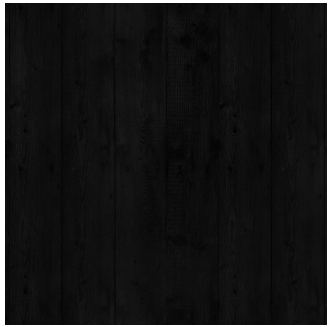
Roughness



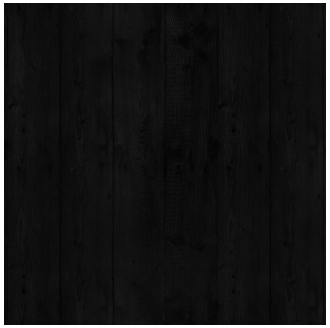
Ambient Occlusion



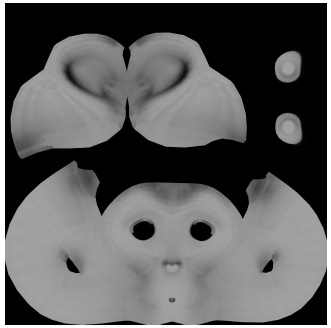
Albedo



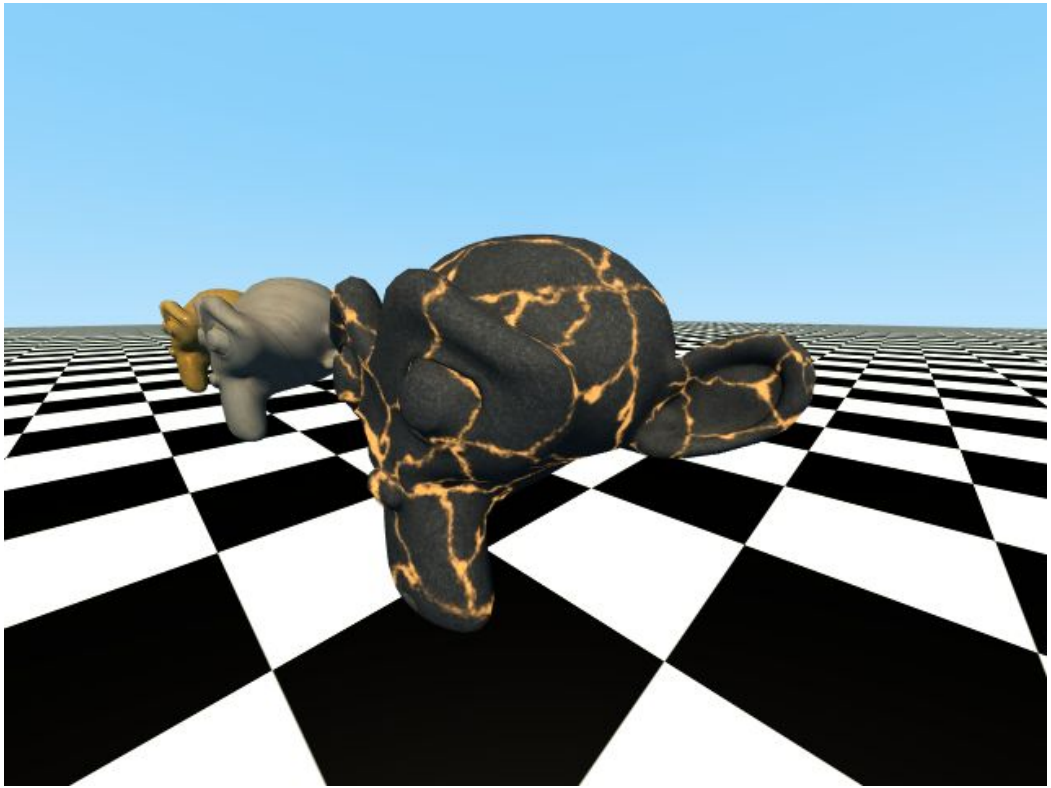
Specular

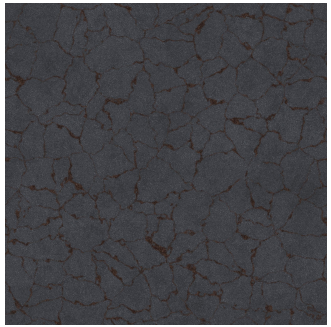


Roughness

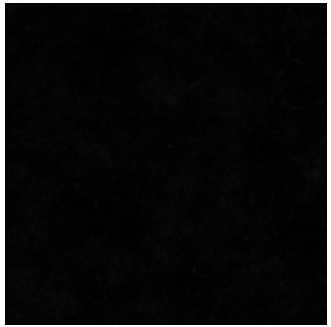


Ambient Occlusion

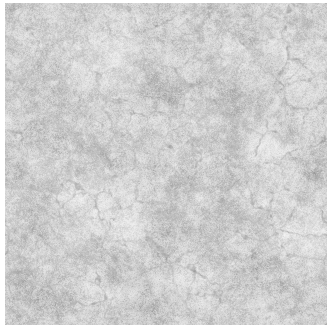




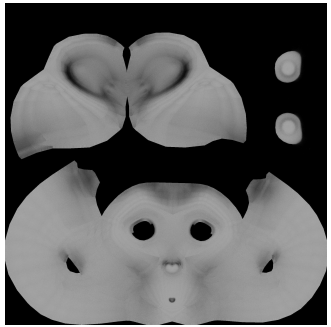
Albedo



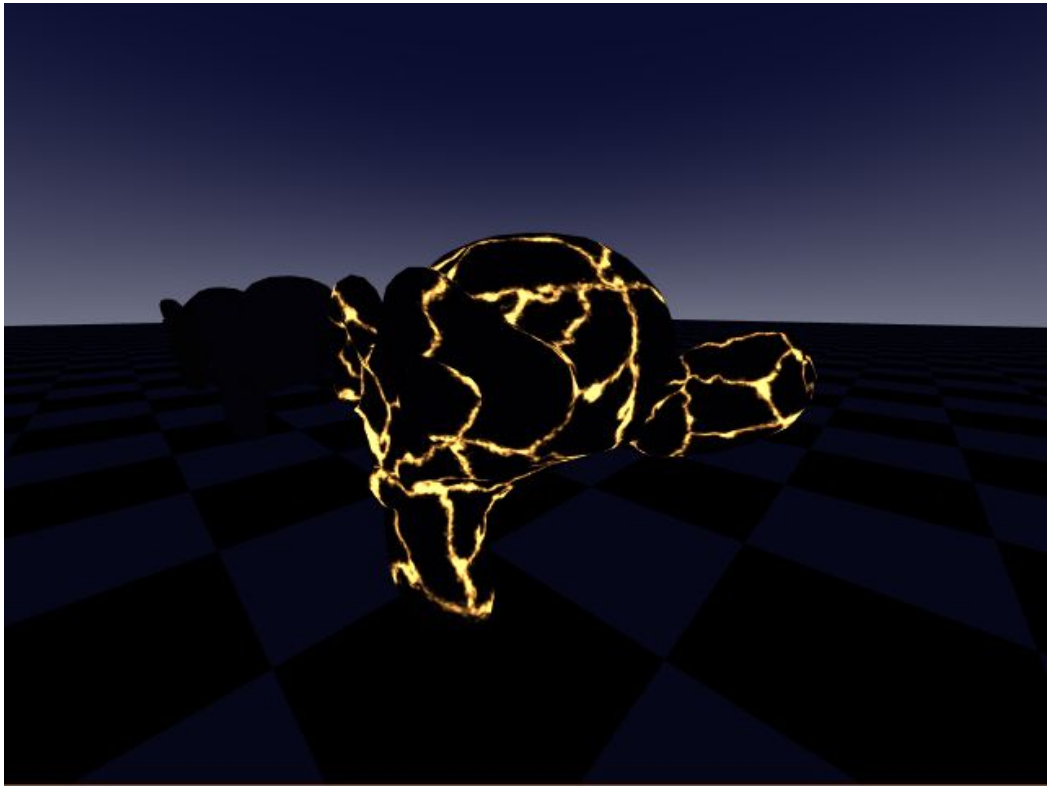
Specular

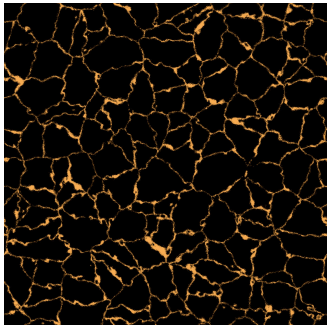


Roughness



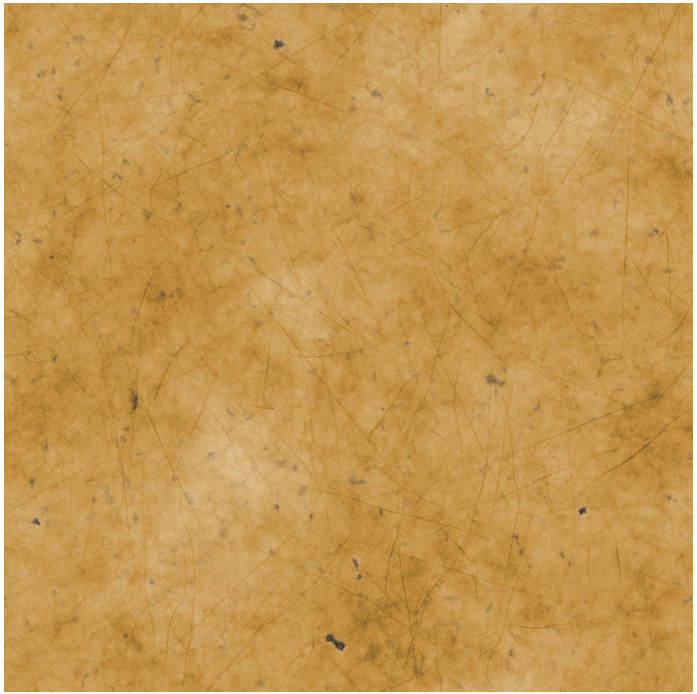
Ambient Occlusion



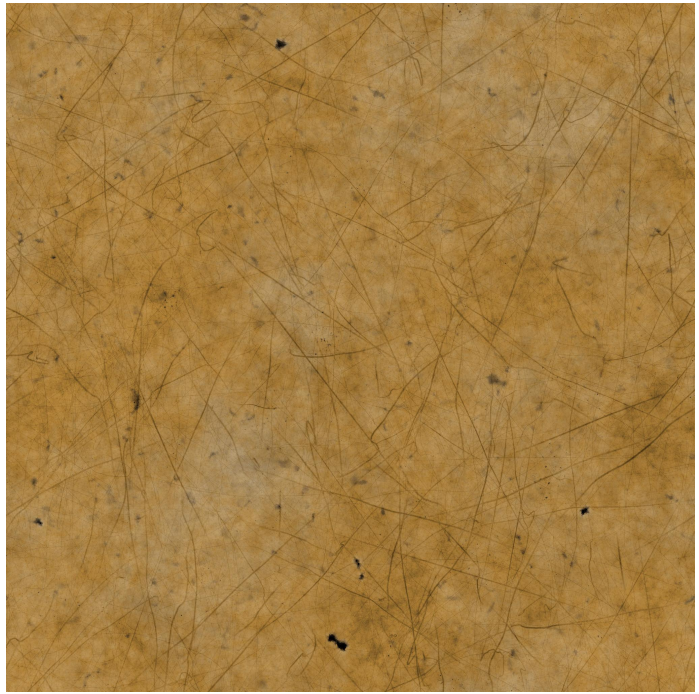


Emissive

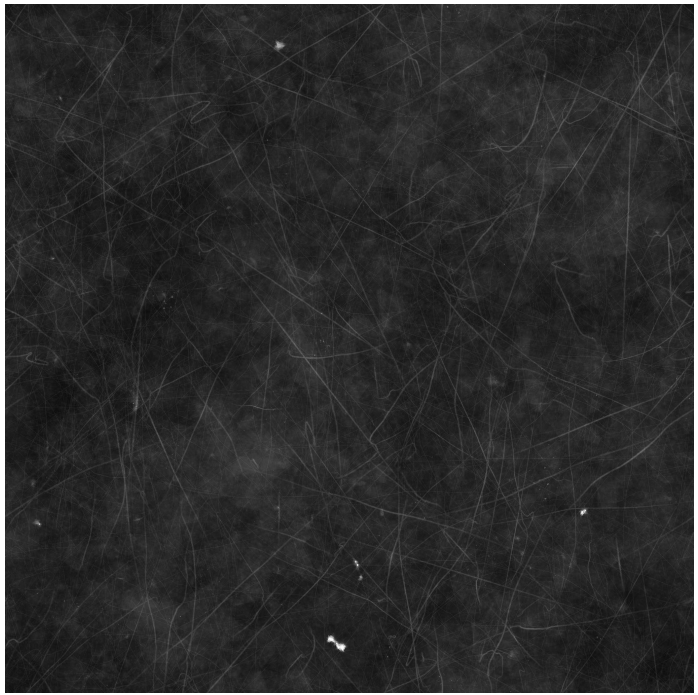
Diffuse = Albedo



Specular = Specular



Roughness maps to Shininess



The more rough the surface, the less shiny it is.

Roughness is from 0 to 1 but shininess is from 0 to ∞

To convert roughness to shininess, you can do:

$$shininess = 2/roughness^4 - 2$$

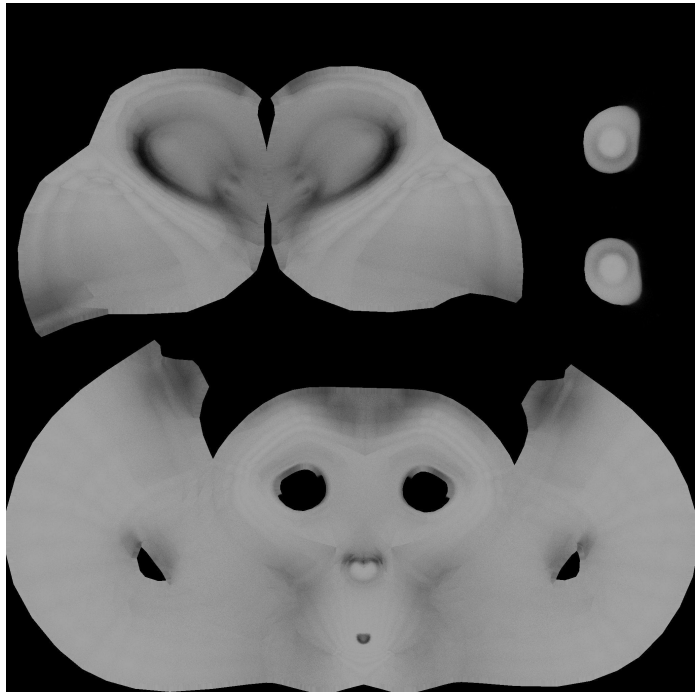
Source:
http://graphicrants.blogspot.com/2013/08/specular-brdf-reference.html

Ambient doesn’t reach all location in a similar manner.

Folded areas (e.g. Monkey Ears) get less ambient compared to its face.

We use an Ambient Occlusion (AO) map to imitate this phenomena.

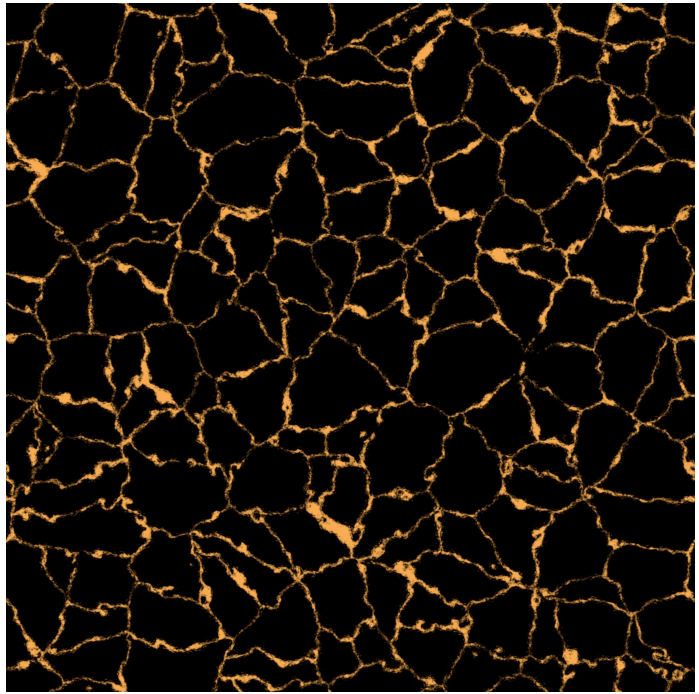
Ambient = AO * Albedo



Emissive maps is used to show that some places emit light directly to out eyes.

Emissive properties are not affected by other light sources or ambient and can just be added to the final color directly.

Color += Emissive



```
struct Material {
   vec3 diffuse, specular, ambient, emissive;
   float shininess;
};

struct TexturedMaterial {
   sampler2D albedo_map;
   vec3 albedo_tint;
   sampler2D specular_map;
   vec3 specular_tint;
   sampler2D ambient_occlusion_map;
   sampler2D roughness_map;
   vec2 roughness_range;
   sampler2D emissive_map;
   vec3 emissive_tint;
};

Material sample_material(TexturedMaterial tex_mat, vec2 tex_coord){
   Material mat;
   mat.diffuse = tex_mat.albedo_tint * texture(tex_mat.albedo_map, tex_coord).rgb;
   mat.specular = tex_mat.specular_tint * texture(tex_mat.specular_map, tex_coord).rgb;
   mat.emissive = tex_mat.emissive_tint * texture(tex_mat.emissive_map, tex_coord).rgb;
   mat.ambient = mat.diffuse * texture(tex_mat.ambient_occlusion_map, tex_coord).r;
   float roughness = mix(tex_mat.roughness_range.x, tex_mat.roughness_range.y,
       texture(tex_mat.roughness_map, tex_coord).r);
   mat.shininess = 2.0f/pow(clamp(roughness, 0.001f, 0.999f), 4.0f) - 2.0f;
   return mat;
}
```

Thank you