Computer Graphics

4. Load a Model, Put Light on it

Dr Joan Llobera – joanllobera@enti.cat

Spring 2019

Outline

1. Load a Model
2. Introduction to Lightning
3. Theory on Lightning
4. The Phong Lightning Model

1.1 What's in an OBJ

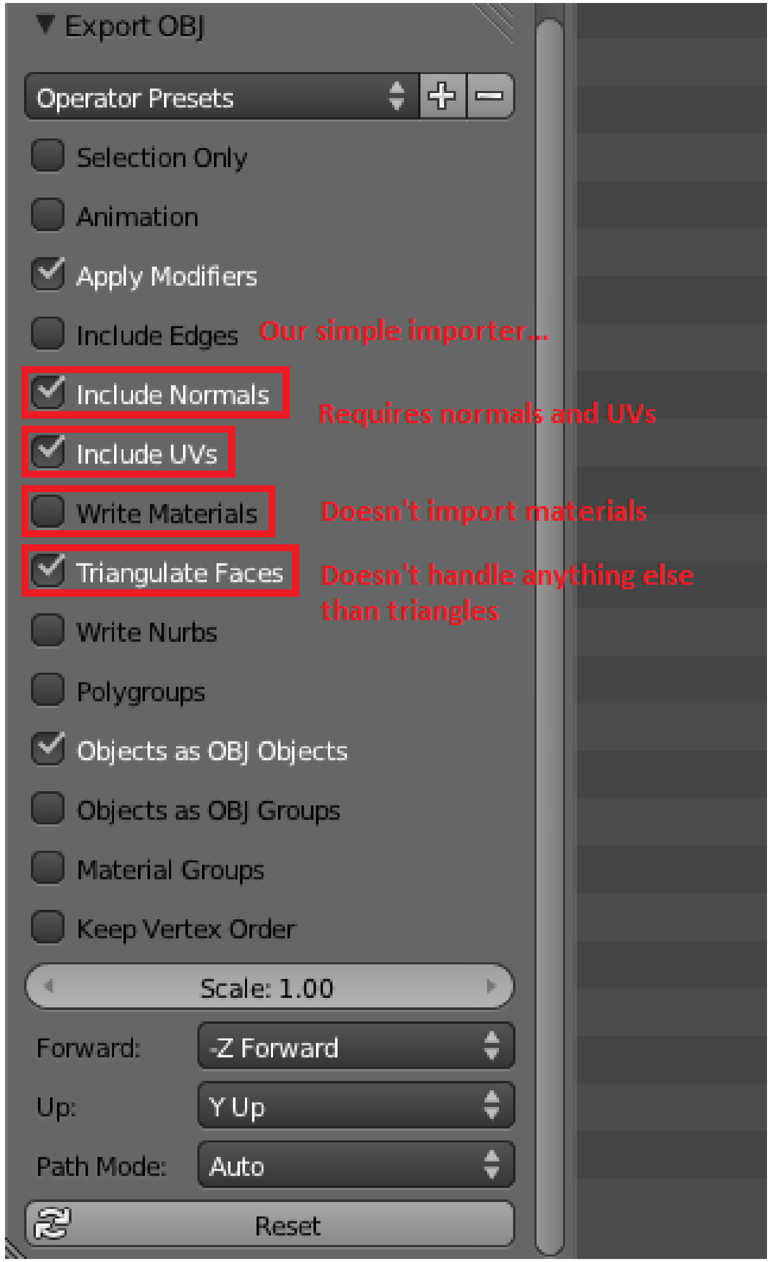
v	vertex	
Vt	texture coordinates	
Vn	vertex normal	
f	a face	(indexes start at 1, not 0)

```
# Blender3D v249 OBJ File: untitled.blend
# www.blender3d.org
mtllib cube.mtl
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vt 0.748573 0.750412
vt 0.749279 0.501284
vt 0.999110 0.501077
vt 0.999455 0.750380
vt 0.250471 0.500702
vt 0.249682 0.749677
vt 0.001085 0.750380
vt 0.001517 0.499994
vt 0.499422 0.500239
vt 0.500149 0.750166
vt 0.748355 0.998230
vt 0.500193 0.998728
vt 0.498993 0.250415
vt 0.748953 0.250920
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
vn -0.000000 -0.000000 1.000000
vn -0.000001 0.000000 1.000000
vn 1.000000 -0.000000 0.000000
vn 1.000000 0.000000 0.000001
vn 0.000000 1.000000 -0.000000
vn -0.000000 -1.000000 0.000000
usemtl Material_ray.png
s off
f 5/1/1 1/2/1 4/3/1
f 5/1/1 4/3/1 8/4/1
f 3/5/2 7/6/2 8/7/2
f 3/5/2 8/7/2 4/8/2
f 2/9/3 6/10/3 3/5/3
f 6/10/4 7/6/4 3/5/4
f 1/2/5 5/1/5 2/9/5
f 5/1/6 6/10/6 2/9/6
f 5/1/7 8/11/7 6/10/7
f 8/11/7 7/12/7 6/10/7
f 1/2/8 2/9/8 3/13/8
f 1/2/8 3/13/8 4/14/8
```

1.2 Export from Blender

We are doing a very simplified drawing, we need a simplified export

To start, just use the cube made available online

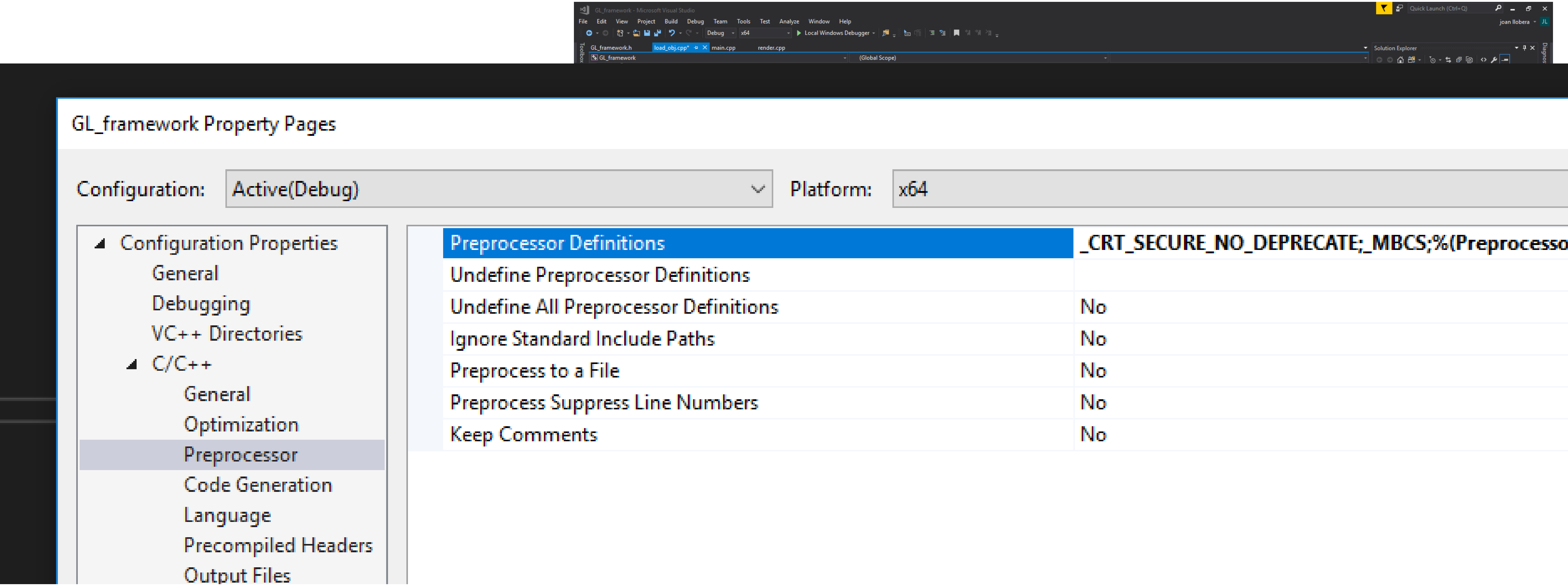


1.3 Load model in cpp

- In the basic project, create a new file called load_obj.cpp
- Include the needed headers
- We will be following http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

```
// Include standard headers
#include <stdio.h>
#include <stdlib.h>
#include <vector>

#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>
```


GL_framework Property Pages
Configuration:
Active(Debug)
Platform:
x64
Configuration Properties
General
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Preprocessor Definitions
_CRT_SECURE_NO_DEPRECATE;_MBCS;%(Preprocesso
Undefine Preprocessor Definitions
Undefine All Preprocessor Definitions
No
Ignore Standard Include Paths
No
Preprocess to a File
No
Preprocess Suppress Line Numbers
No
Keep Comments
No

1.3. Setup

Create load_obj.cpp

Add:

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>
```

```
bool loadOBJ(const char * path,
std::vector < glm::vec3 > & out_vertices,
std::vector < glm::vec2 > & out_uvs,
std::vector < glm::vec3 > & out_normals
){


return true;

}
```

1.3. Setup

In render.cpp:

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <glm\gtc\type_ptr.hpp>
#include <glm\gtc\matrix_transform.hpp>

extern bool loadOBJ(const char * path,
std::vector < glm::vec3 > &
out_vertices,
std::vector < glm::vec2 > & out_uvs,
std::vector < glm::vec3 > & out_normals
);
```

```
//variables to load an object:

std::vector< glm::vec3 > vertices;
std::vector< glm::vec2 > uvs;
std::vector< glm::vec3 > normals;
```

In render.cpp, within the init function

```
bool res = loadOBJ("cube.obj",
vertices, uvs, normals);
```

And put cube.obj where the project file is

The project file: GL_framework.vcxproj

The path of the project file: /code

1.4 Load a model

We follow the online tutorial, in the section “reading the file”

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

We don’t target rendering: we only want to load the model. Check with debugging that you are loading the right vertices

1.5 Load your model

Find a (simple) model that you like, import it and load it in the C++ program

2. Introduction to model lightning

Basic of basics:

Draw a uniform color

Exercise: draw a cube with a basic colour shader whose colour changes with sin(currentTime)

2. Introduction to model lightning

Basic of basics:

Draw a uniform color

Exercise: draw a cube with a basic colour shader whose colour changes with sin(currentTime)

```
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
uniform mat4 mvpMat;\n\
void main() {\n\
gl_Position = mvpMat *
vec4(in_Position, 1.0);\n\
}";
const char* cube_fragShader =
"#version 330\n\
out vec4 out_Color;\n\
uniform vec4 color;\n\
void main() {\n\
out_Color = color;\n\
}";

glUniform4f(glGetUniformLocation(cubeProgram,
"color"), 0.1f, 0.5f+0.5f*sin(currentTime),
1.f, 0.f);
```

2. Introduction to model lightning

Ambient

Exercise: draw again the cube, but this time make the ambient light change with currentTime

```
Cube::drawCube(currentTime);
```

```
const char* cube_fragShader =
"#version 330\n\
out vec4 out_Color;\n\
uniform vec4 color;\n\
uniform vec4 ambient;\n\
void main() {\n\
vec3 rgb= color.rgb * ambient.rgb;\n\
out_Color = vec4(rgb, 1.0 );\n\
}";
```

```
glUniform4f(glGetUniformLocation(cubeProgram,
"color"), .1f, 0.5f, .5f, 0.f);
glUniform4f(glGetUniformLocation(cubeProgram,
"ambient"), 0.4f + 0.2f*sin(currentTime),
0.4f+0.2f*sin(currentTime), 0.4f +
0.2f*sin(currentTime), 0.0f);
```

2. Introduction to model lightning

Ambient

Small improvement:

Consider the alpha only from the colour

To improve on this model, we need some theory of lightning.

However, first we need to make sure we understand in and out variables

```
const char* cube_fragShader =
"#version 330\n\
out vec4 out_Color;\n\
uniform vec4 color;\n\
uniform vec4 ambient;\n\
void main() {\n\
vec3 rgb= min(color.rgb
*ambient.rgb,vec3(1.0));\n\
out_Color = vec4(rgb, 1.0 );\n\
}";
```

(Input and output)

Small Exercise

1. Make the cube move horizontally

2.Make it change color in world coordinates:

-if xpos smaller than 0, red=0

-if xpos bigger than 1, red=1

-otherwise, red=xpos

3. Make the change depending on screen coordinates

(Input and output)

Small Exercise

1. Make the cube move horizontally

2.Make it change color in world coordinates:

-if xpos smaller than 0, red=0

-if xpos bigger than 1, red=1

-otherwise, red=xpos

3. Make the change depending on screen coordinates

```
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
uniform mat4 mvpMat;\n\
uniform float time;\n\
out float xcolor;\n\
void main() {\n\
vec3 temp = in_Position;\n\
temp.x = temp.x + 4*sin(time);\n\
gl_Position = mvpMat * vec4(temp, 1.0);\n\
xcolor = min(temp.x, 1.0);\n\
xcolor = max(xcolor, 0.0);\n\
}";

//xcolor = min(gl_Position.x, 1.0);\n\
//xcolor = max(gl_Position.x, 0.0);\n\

const char* cube_fragShader =
"#version 330\n\
out vec4 out_Color;\n\
in float xcolor;\n\
uniform vec4 color;\n\
uniform vec4 ambient;\n\
void main() {\n\
vec3 rgb= min(color.rgb,vec3(1.0));\n\
rgb.r = xcolor;\n\
out_Color = vec4(rgb, 1.0 );\n\
}";

glUniform4f(glGetUniformLocation(cubeProgram, "ambient"), 0.4f , 0.4f
, 0.4f , 0.0f);
glUniform1f(glGetUniformLocation(cubeProgram, "time"), currentTime);
```

(Input and output)

Small Exercise

1. Make the cube move horizontally

2.Make it change color in world coordinates:

-if xpos smaller than 0, red=0

-if xpos bigger than 1, red=1

-otherwise, red=xpos

3. Make the change depending on screen coordinates

```
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
uniform mat4 mvpMat;\n\
uniform float time;\n\
out float xcolor;\n\
void main() {\n\
vec3 temp = in_Position;\n\
temp.x = temp.x + 4*sin(time);\n\
gl_Position = mvpMat * vec4(temp, 1.0);\n\
//xcolor = min(temp.x, 1.0);\n\
xcolor = min(gl_Position.x,1.0);\n\
xcolor = max(xcolor, 0.0);\n\
}";
```

(Input and output)

Notice how certain variables (gl_Position) work the same way, but are declared by default

```
const char* cube_vertShader =
"#version 330\n\
in vec3 in_Position;\n\
uniform mat4 mvpMat;\n\
uniform float time;\n\
out float xcolor;\n\
void main() {\n\
vec3 temp = in_Position;\n\
temp.x = temp.x + 4*sin(time);\n\
gl_Position = mvpMat * vec4(temp, 1.0);\n\
xcolor = min(temp.x, 1.0);\n\
xcolor = max(xcolor.x, 0.0);\n\
}";

//xcolor = min(gl_Position.x, 1.0);\n\
//xcolor = max(xcolor.x, 0.0);\n\

const char* cube_fragShader =
"#version 330\n\
out vec4 out_Color;\n\
in float xcolor;\n\
uniform vec4 color;\n\
uniform vec4 ambient;\n\
void main() {\n\
vec3 rgb= min(color.rgb,vec3(1.0));\n\
rgb.r = xcolor;\n\
out_Color = vec4(rgb, 1.0 );\n\
}";

glUniform4f(glGetUniformLocation(cubeProgram, "ambient"), 0.4f , 0.4f
, 0.4f , 0.0f);
glUniform1f(glGetUniformLocation(cubeProgram, "time"), currentTime);
```

ESCOLA DE NOVES TECNOLOGIES INTERACTIVES

Universitat de Barcelona

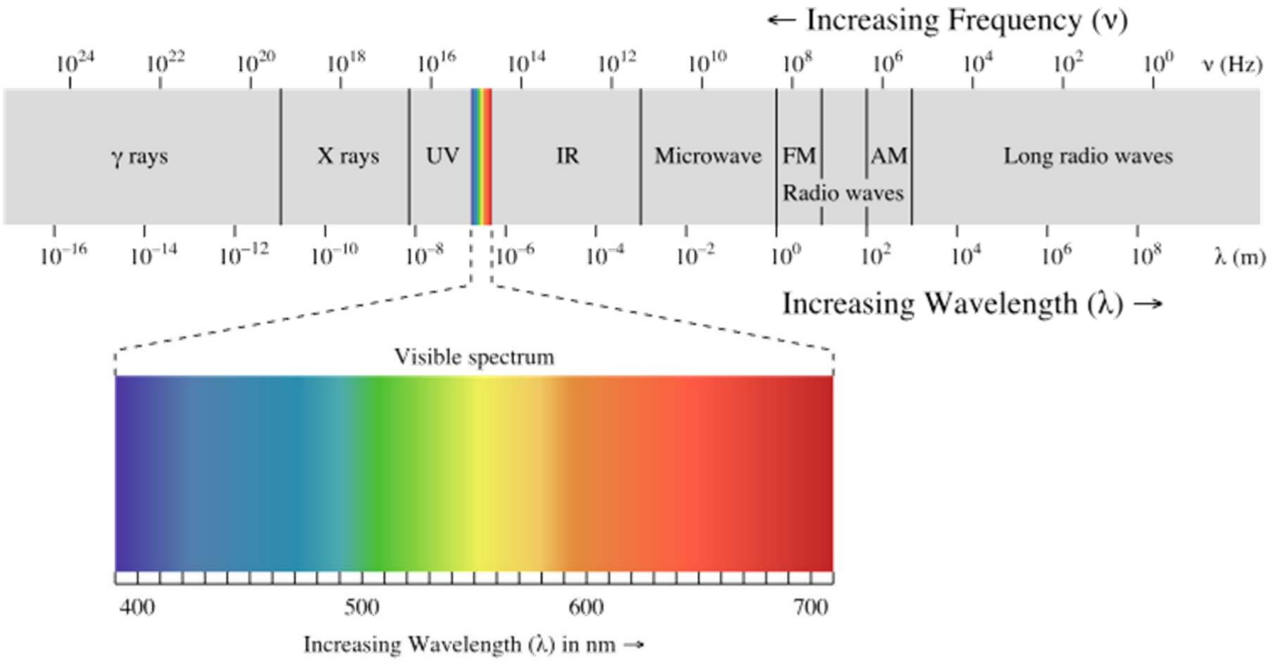
3. Theory on Lightning

3.1 Why do we need shading?

- **Objects have not a uniform color** because the light-material interactions cause each point to have a different color or shade
- The **final color** will depend on:
  - The light sources
  - The material properties
  - The location of viewer
  - The surface orientation
- Therefore, we need to **understand and model** how the **light affects the color** of the objects in order to increase the reality of our graphics
- As we can see in the below example, it is not same draw an object with an uniform color than an object in which the color depends on the **light-material interaction**

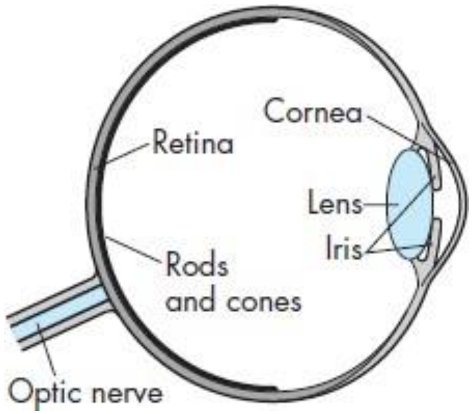
3.2 Visible Light

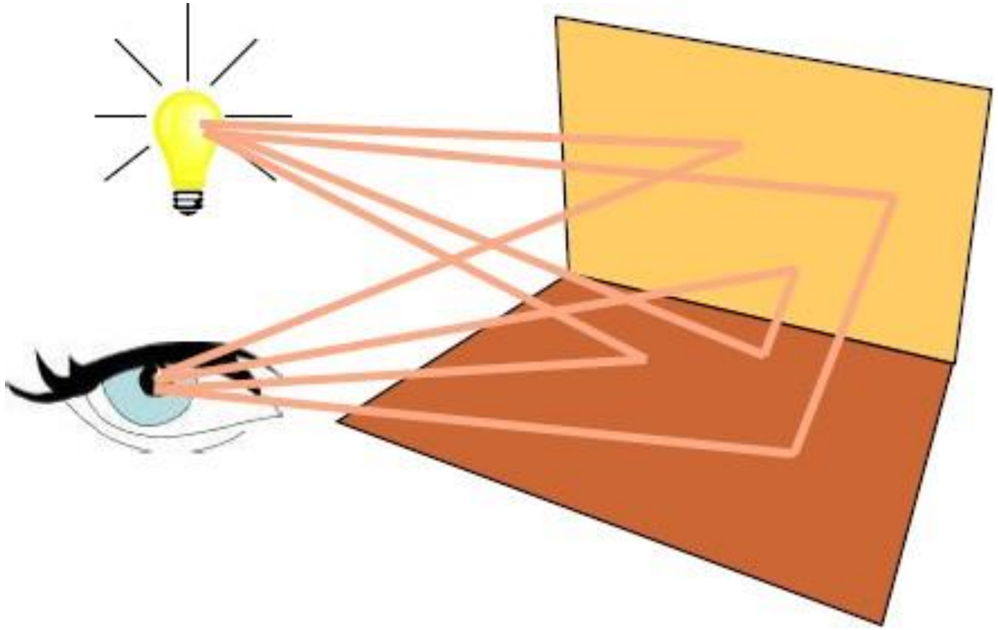
- **Light** is electromagnetic radiation within a certain portion of the electromagnetic spectrum
- The word usually refers to **visible light**, which is visible to the human eye and is responsible for the sense of sight
- Each **light source** has a characteristic spectrum



3.3 A simple model for the eye

- A **light source** emits **photons** which go **straight forward** until they strike a surface and, then, photons are **reflected**, **refracted** and/or **absorbed** by the surface based on its **material** properties
- The **human eye** is able to catch these photons, which stimulates the rods and cones
- If there were **no light sources**, the objects would be dark and there would be **nothing visible**





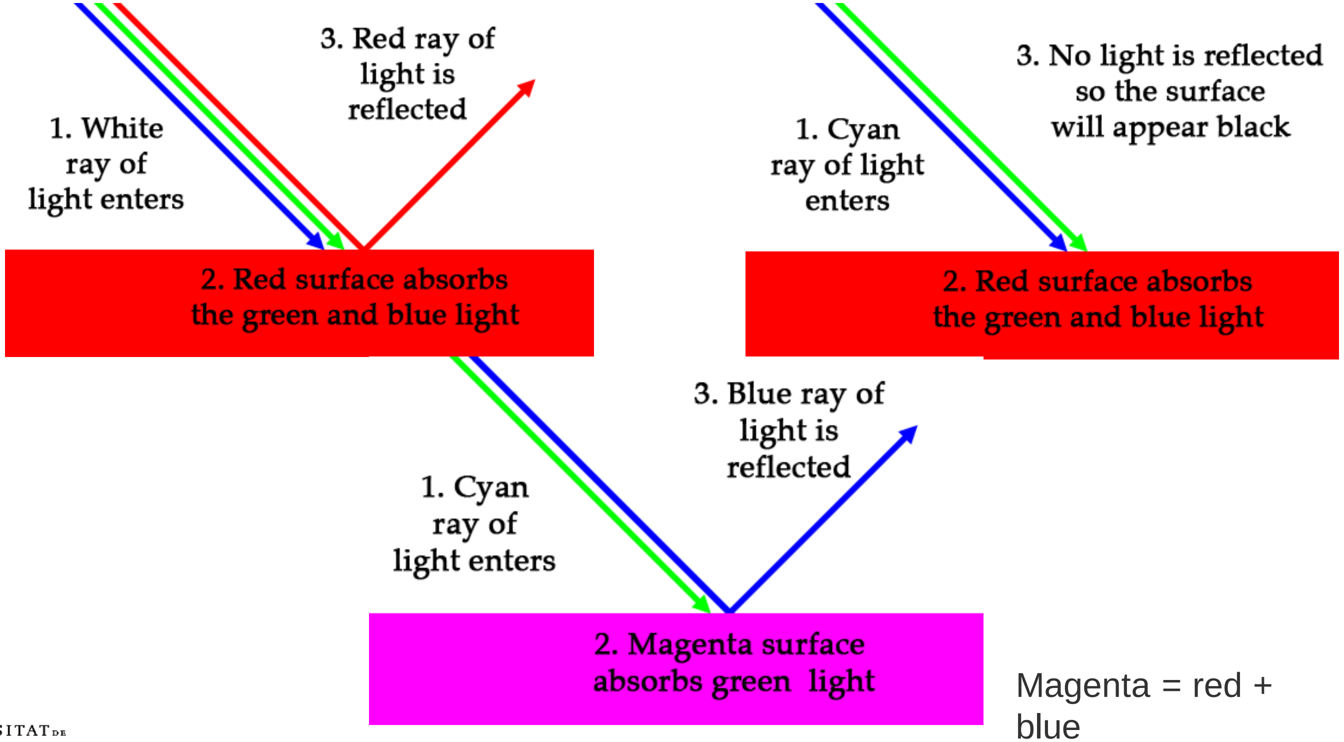
3.3 Light-material interactions

- An object surface can be covered by one or more different **materials**
- **Light-material interactions** cause each point to have a different **color** or **shade**
  - Light that strikes an object is partially **absorbed** and partially **scattered** (reflected)
  - The amount reflected determines the **color** and **brightness** of the object. Thus, a surface appears red under white light because the red component of the light is reflected and the rest is absorbed
  - The **reflected light is scattered** in a manner that depends on the smoothness and orientation of the surface

The **light sources**, the **material properties**, the **location of viewer** and the **surface orientation** will condition the **light-material interaction** and, therefore, the **final color**

3.3 Light-material interaction: absorption

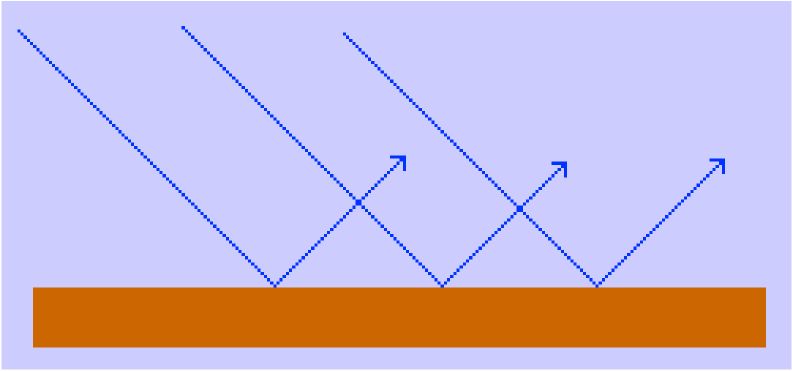
- The **color of a light** is a **combination of different intensities** of red, green and blue lights
- The **RGB color** of a surface represents how light is absorbed and reflected by that surface



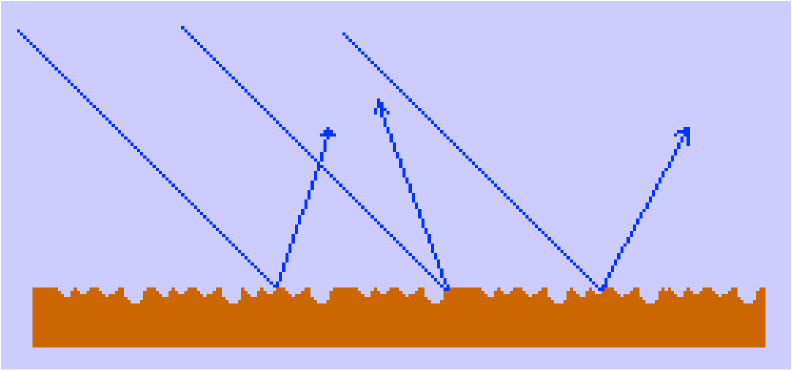
Magenta = red + blue

3.3 Light-material interaction: reflection

- **Reflection** is the change in direction of a wave front at an interface between two different media so that the wave front returns into the medium from which it originated
- Reflection of light is either **specular** (mirror-like) or **diffuse** (retaining the energy, but losing the image) depending on the nature of the interface



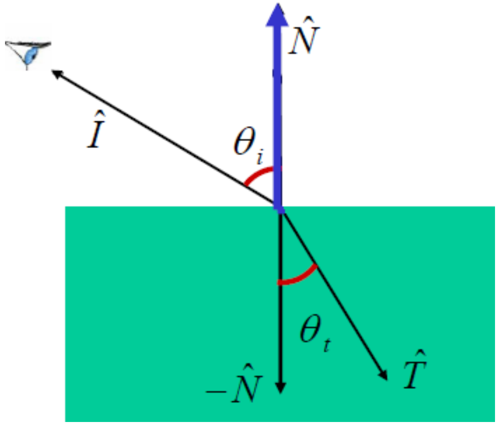
The **specular reflection** means that ray of lights are parallel bounced due to the surface is smooth



The **diffuse reflection** means that ray of lights are not parallel bounced due to the surface is rough

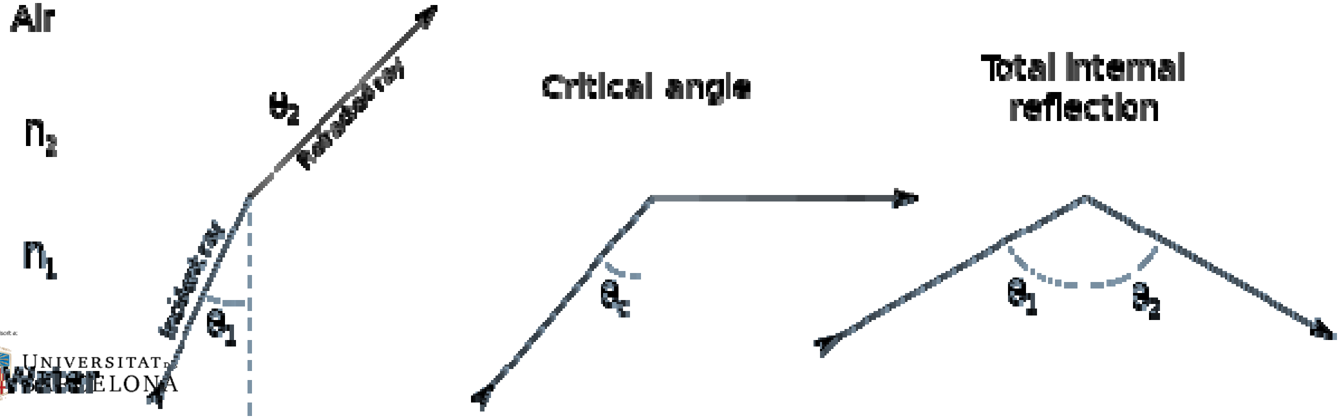
3.4 Light-material interaction: refraction

- **Refraction** happens when the direction of a ray of light is changed at an interface between two different **transparent media** with different density. Therefore, the ray of light is NOT returned to the coming media



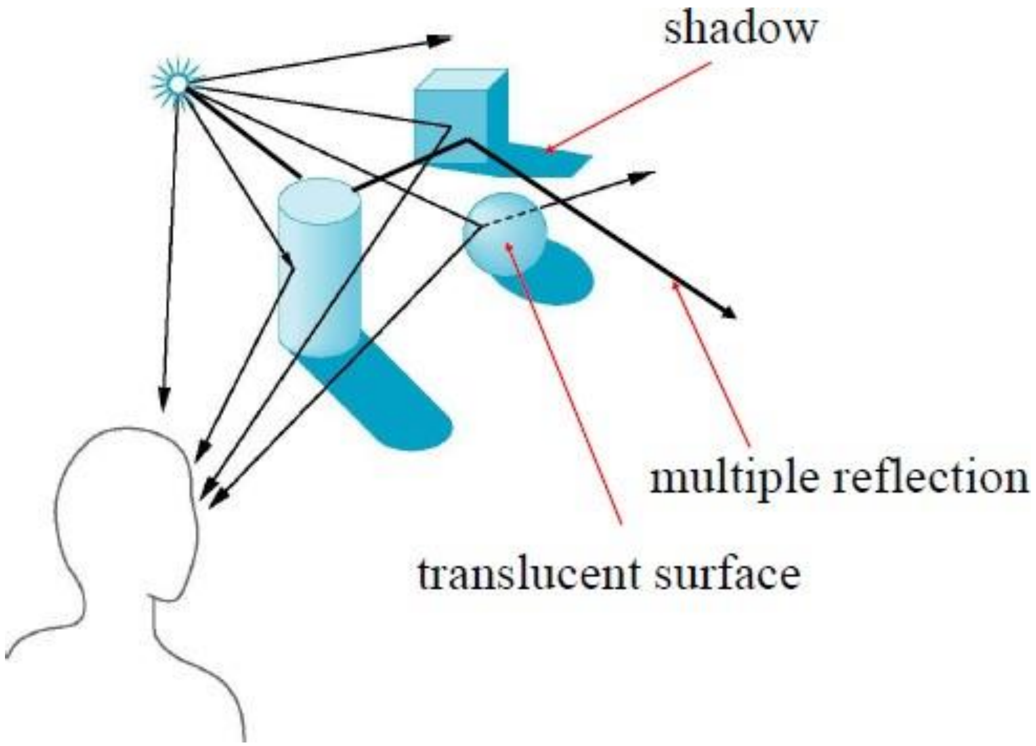
$$\frac{\sin \theta_t}{\sin \theta_i} = \frac{\eta_i}{\eta_t} = \eta_r$$

Snell law



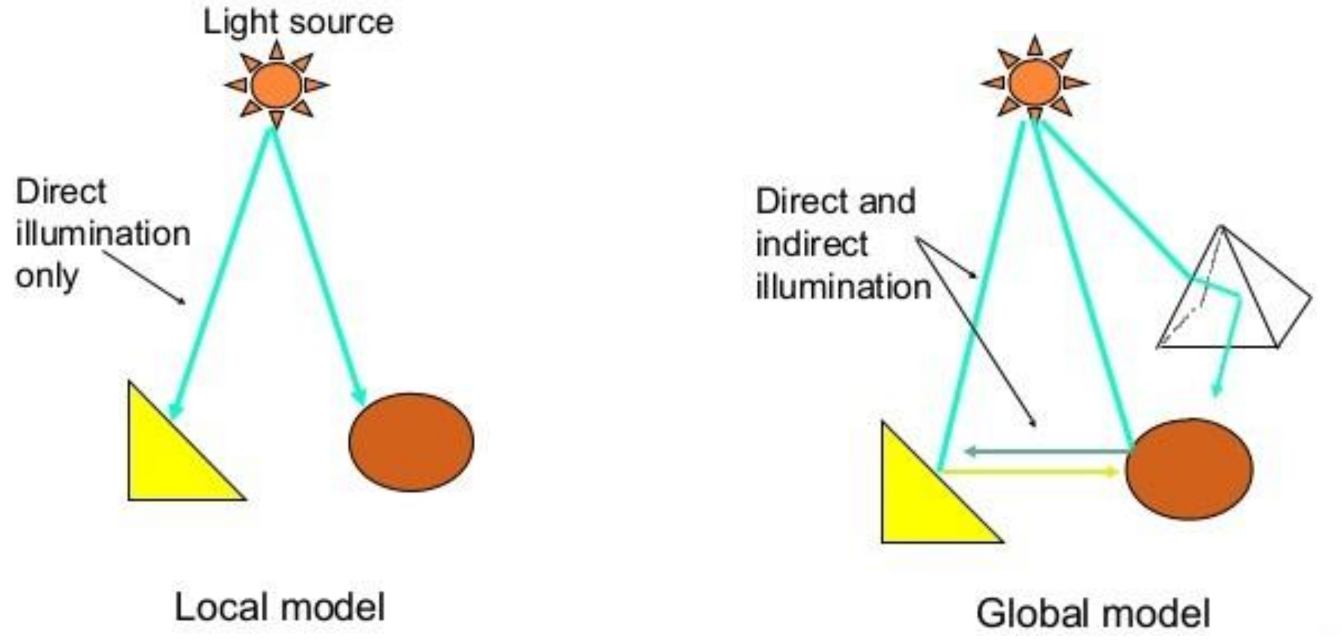
3.5 Limits of the graphics pipeline

- The infinite scattering and absorption of light will create the **global effect**, which includes **shades** and **multiple scattering** from object to object
- Exist many techniques for **approximating global effects** based on simplifying the reality



3.5 Limits of the graphics pipeline

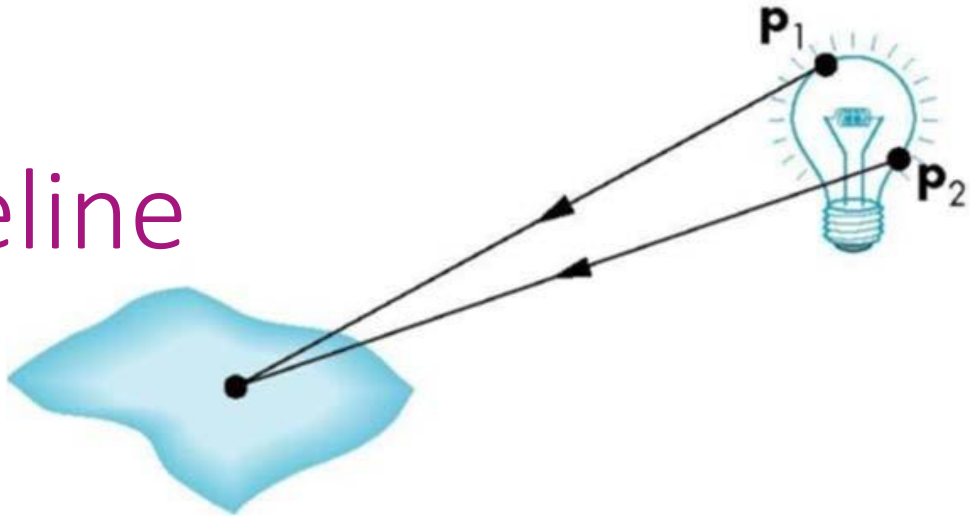
- Although correct shading requires a **global calculation** involving all objects and light sources, it is **incompatible with pipeline model** which shades each polygon independently (local rendering)



- Considers only direct illumination

- Indirect light is considered
- Light is reflected multiple times

3.5 Limits of the graphics pipeline

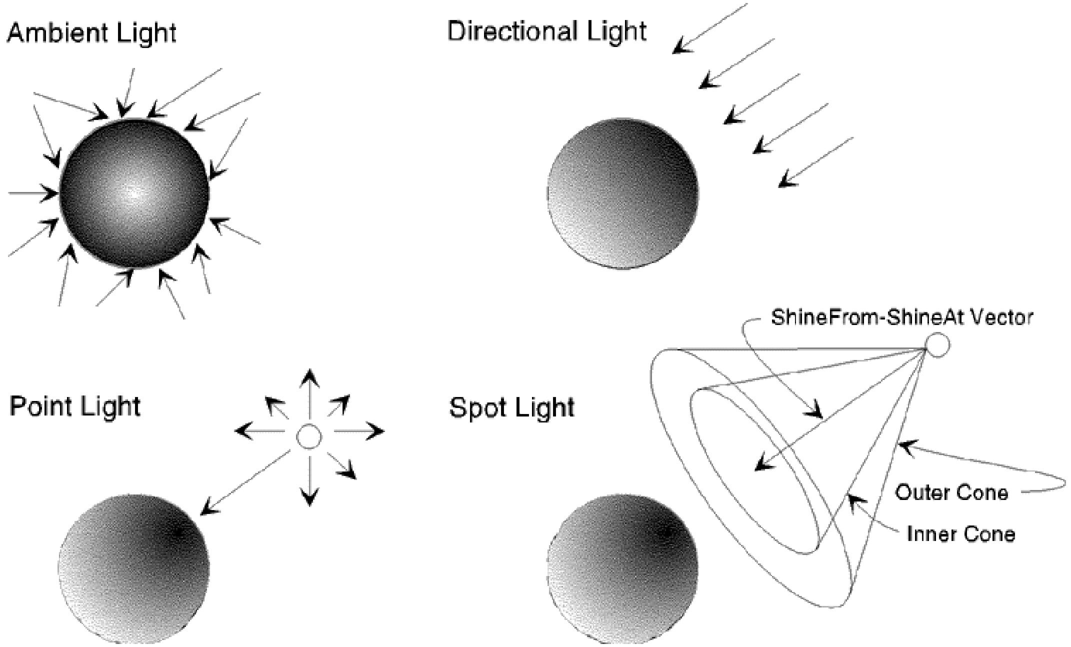


- **General light sources are difficult** to work with because we must integrate light coming from all points on the source

- For this reason, we initially consider **simple light sources**

Same amount of light everywhere in scene

A light source which is far away

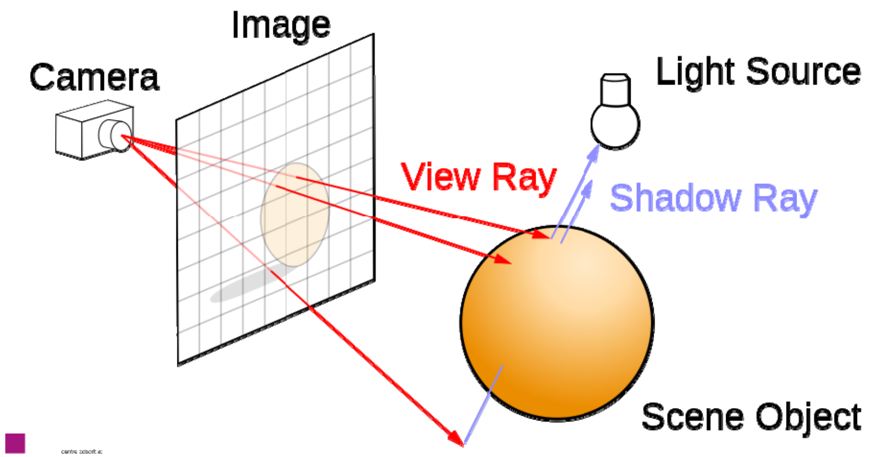


A light source modeled with a position and a color

Restrict light from ideal point

3.5 Limits of the graphics pipeline

- **Ray tracing** is an approach different from the graphic pipeline for determining the lighting and shading
- It **follow rays of light** from center of projection until they either are absorbed by objects or gooff to infinity
- The main benefits is that it can **handle global effects** such as multiple reflections and translucent objects. Therefore, it is **slow** and must have whole **data base available at all times**
- This approach is not always applicable in real-time computer graphics



3.6 Radiometry: Radiant power of visible light

- **Radiometry** is a set of techniques for measuring electromagnetic radiation, including visible light
- The **radiation** or the **radiant power** of the visible light is measured as the **amount of energy** due to the light flow per **unit time** (watts)

Energy of one photon

$$e_\lambda = \frac{hc}{\lambda} \qquad h \approx 6.63 \cdot 10^{-34} J \cdot s \quad c \approx 3 \cdot 10^{8} m/s$$

λ: Wave length

Energy of a set of N photons

$$Q = \sum_{i=1}^{n} \frac{hc}{\lambda_i}$$

Radiant power (Watts)

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

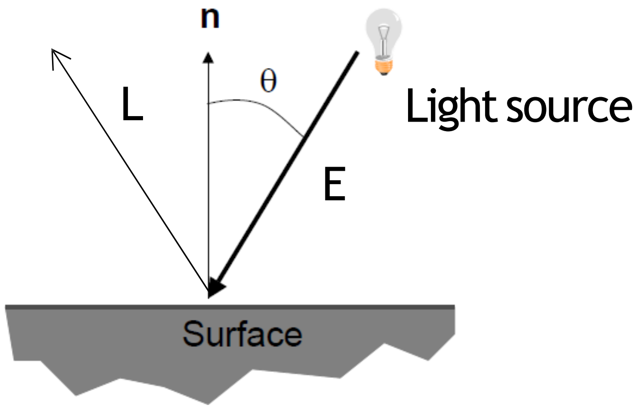
3.6 Radiometry: Radiance and Irradiance

- **Irradiance (E):** Radiant flux received by a surface per unit area
- **Radiance (L):** Radiant flux emitted, reflected, transmitted by a surface, per unit solid angle per unit projected area
- The **discrete equations** for modelling the **Irradiance** (L) and **radiance** (E) of a single point of light are

$$E = \frac{\Phi_s \cos\theta}{4\pi d^2} \quad L = \frac{\Phi_s}{4\pi d^2} \quad Units: \frac{Watt}{meter^2}$$

Φ_s – power of the light source

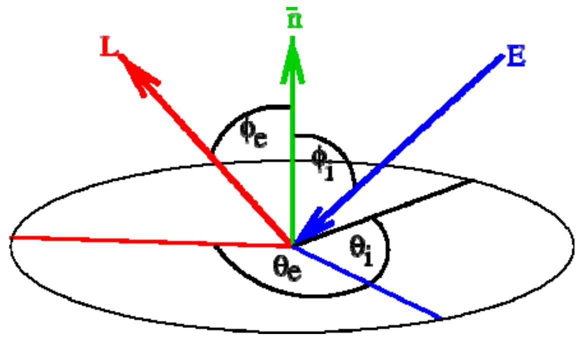
d – distance to the light source



- Let's suppose there is a **single point of light source**
  - If angle=0 then cos(0) = 1
    - The radiant power received is 100% the radiant flux
  - If angle=90 then cos(90)=0
    - The radiant power is not received

3.6 Radiometry: Reflectance

- **Reflectance** is the amount of the flow from the emitted light that is bounded
- The **total radiant power** of the reflectance is the sum of all the irradiance of all the source lights
- The **radiant power** of the reflectance depends on the **angle between the light source and the surface's normal** and the **Bidirectional Reflectance Distribution Function (BRFD)** of the surface



BRFD

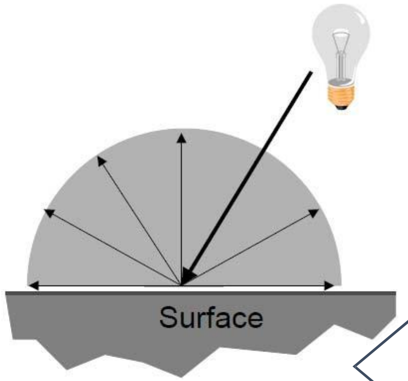
$$f_r(\theta_i,\phi_i,\theta_r,\phi_r) = \frac{dL_r(\theta_r,\phi_r)}{dE_i(\theta_i,\phi_i)}$$

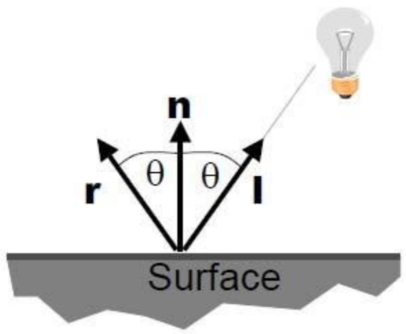
$$L_r(\omega_r) = \sum_{j=1}^{n} f_r(\omega_{ij},\omega_r)E_j = \sum_{j=1}^{n} f_r(\omega_{ij},\omega_r)\cos\theta_j \frac{\Phi_{sj}}{4\pi d_j^{\,2}}$$

4. The Phong Reflection Model

4.1 Ideal vs real reflectors

- The **ray of lights** bound and bound from one surface to other until the infinitive
- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors). This hinders the definition of the reflection model of the surface



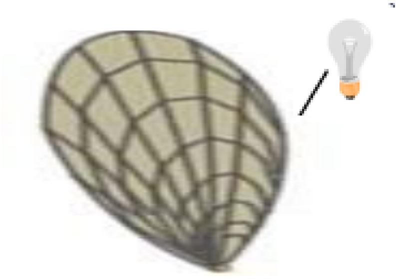


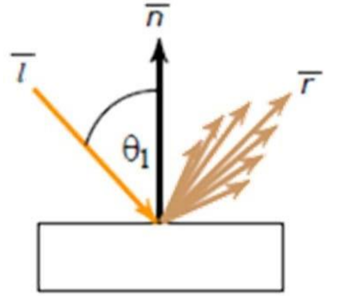
Ideal diffuse and specular reflection in 'ideal' surfaces

- The diffuse reflection reflects equally in all directions
- The specular reflection only reflects in the mirror angle

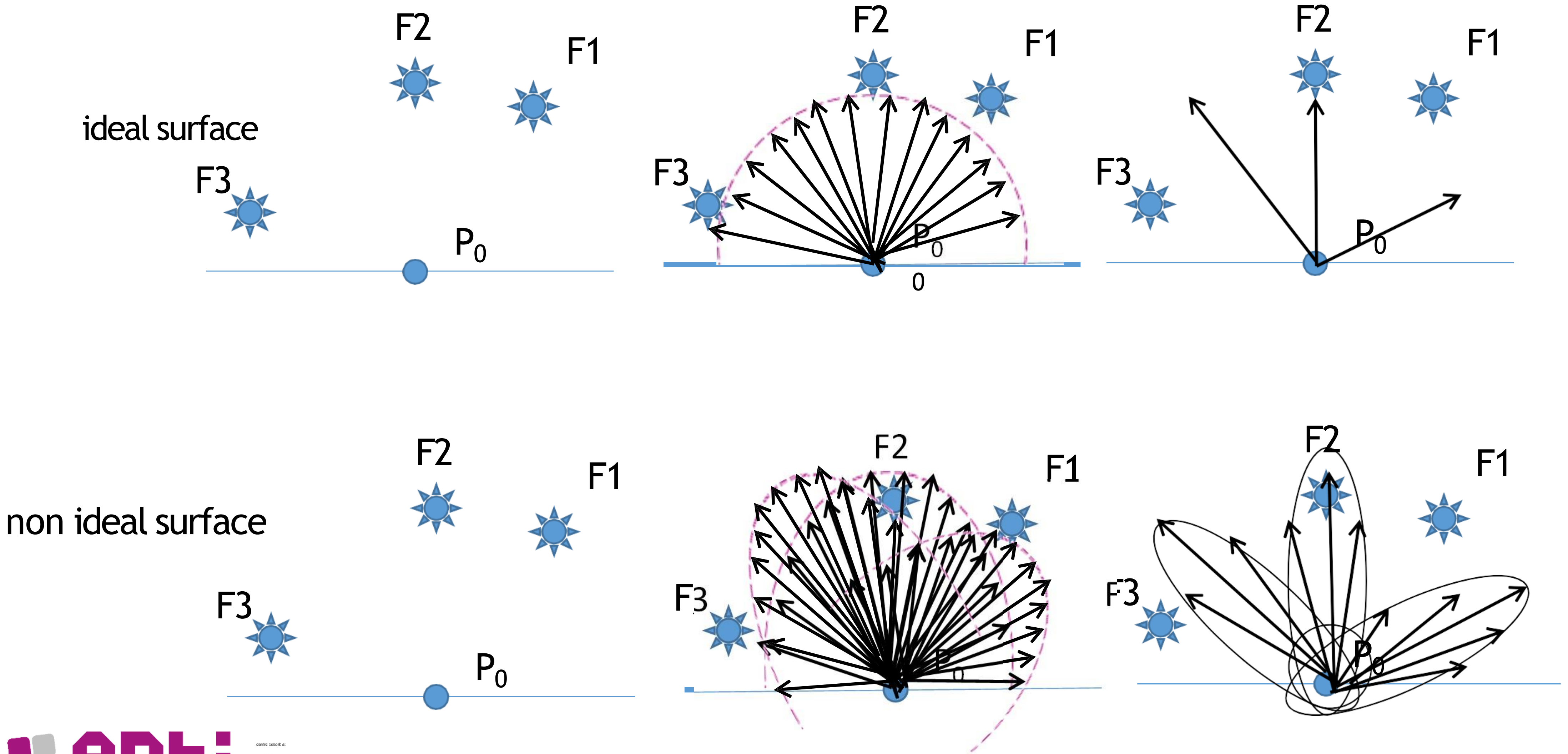
Real diffuse and specular reflection in 'real' surfaces

- The diffuse reflection doesn't reflect equally in all directions
- The specular reflection not only reflects in the mirror angle





4.1 Ideal vs real reflectors

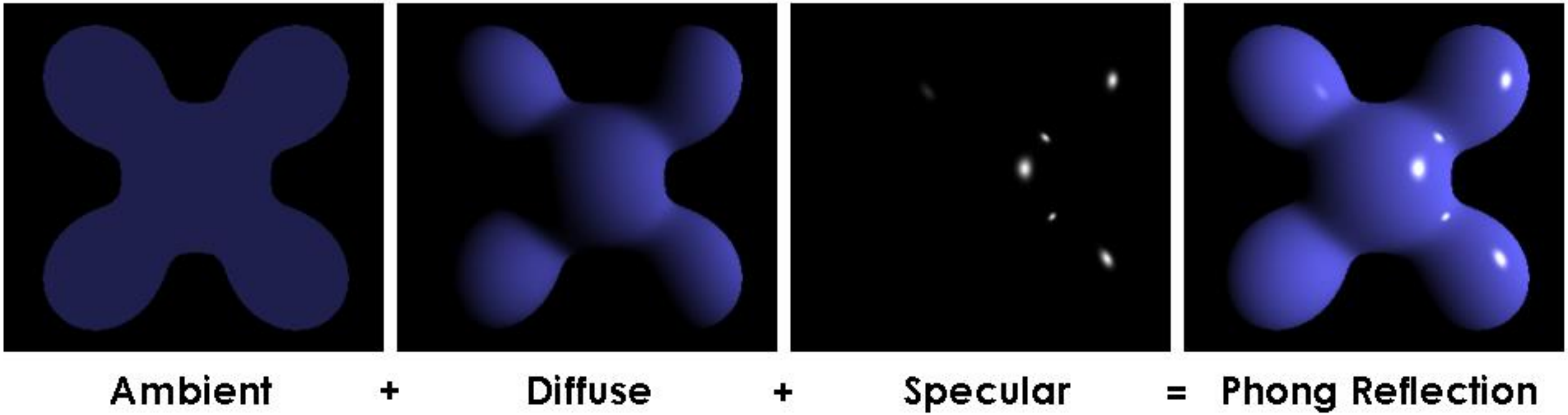


Diffuse reflection

Specular reflection

4.2 The Phong reflection model

- The **Phong reflection model** (also called Phong illumination or Phong lighting) is an empirical model of the local illumination of points on a surface
- It describes the way a **surface reflects light** as a combination of the **diffuse reflection** of **rough surfaces** with the **specular reflection of shiny surfaces**
- The model also includes an **ambient** term to account for the small amount of light that is scattered about the entire scene.



4.2 The Phong reflection model

- The **Phong reflection model** is the result of three partial radiance:
  - **The diffuse reflection**, where the reflection occurs to all directions $$L(\omega_r) = k_d(\mathbf{n}\cdot\mathbf{l})\frac{\Phi_s}{4\pi d^2}$$
  - **The specular reflection**, where the reflection only occurs in the mirror angle $$L(\omega_r) = k_s(\cos\alpha)^q\frac{\Phi_s}{4\pi d^2} = k_s(\mathbf{v}\cdot\mathbf{r})^q\frac{\Phi_s}{4\pi d^2}$$
  - **The ambient light**, which represents the indirect light as a constant $$L(\omega_r) = k_a$$

Φ_s is the radiant power of the light source

k_d, k_s, k_a are described as vectors that represents the impact in the RGB color

k_d: the diffuse reflection coefficient

k_s: the specular reflection coefficient

k_a: the ambient reflection coefficient

n: the normalized surface normal

l: the normalized light direction vector

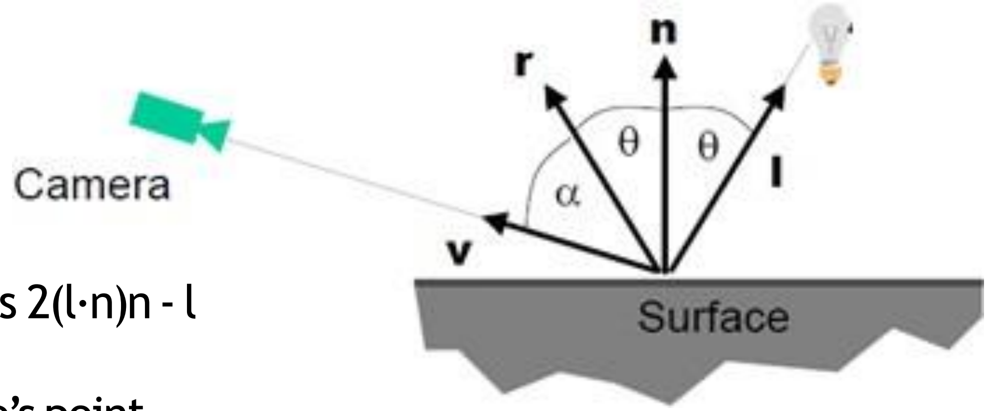
r: it is the mirror of l and it can be computed as 2(l·n)n - l

v: camera direction vector

d: distance from the light source to the surface's point

q: shininess coefficient

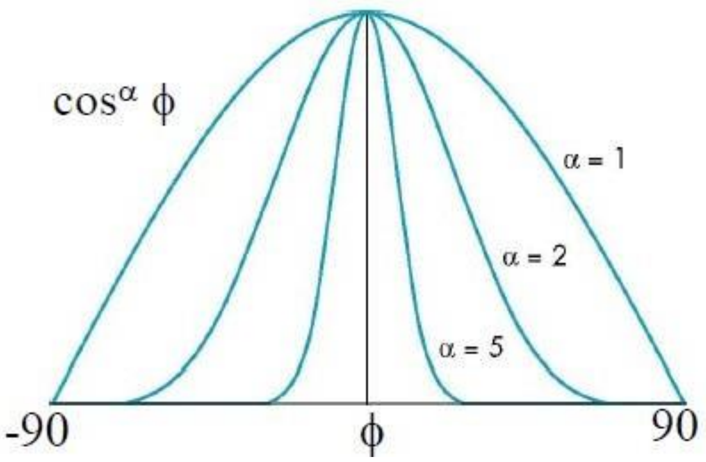
The **angle between n and l MUST be between 0° and 90°**. Otherwise, it means that the light is behind the material so it will not reflect anything.



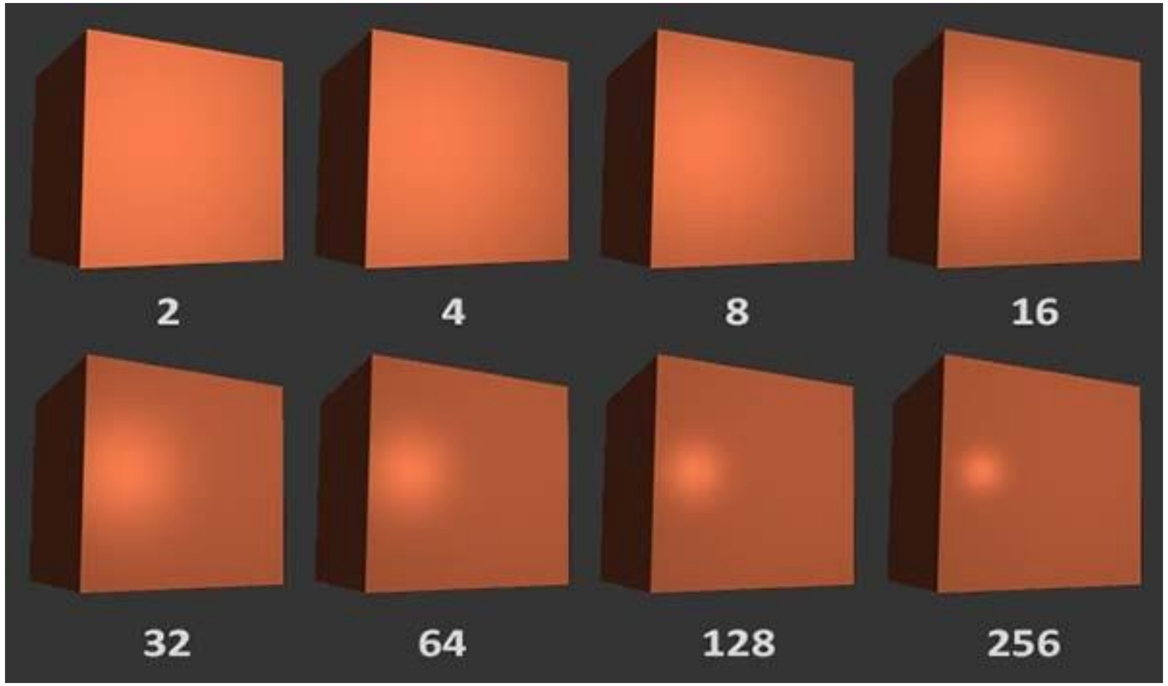
4.2 The Phong reflection model

- The **shininess coefficient** determines the “shininess” of the material and it depends on the angle between v and n
- For example:
  - q values of between 100 and 200 correspond to metals
  - q values between 5 and 10 give surface that look like plastic

Reflection



Angle between v and n



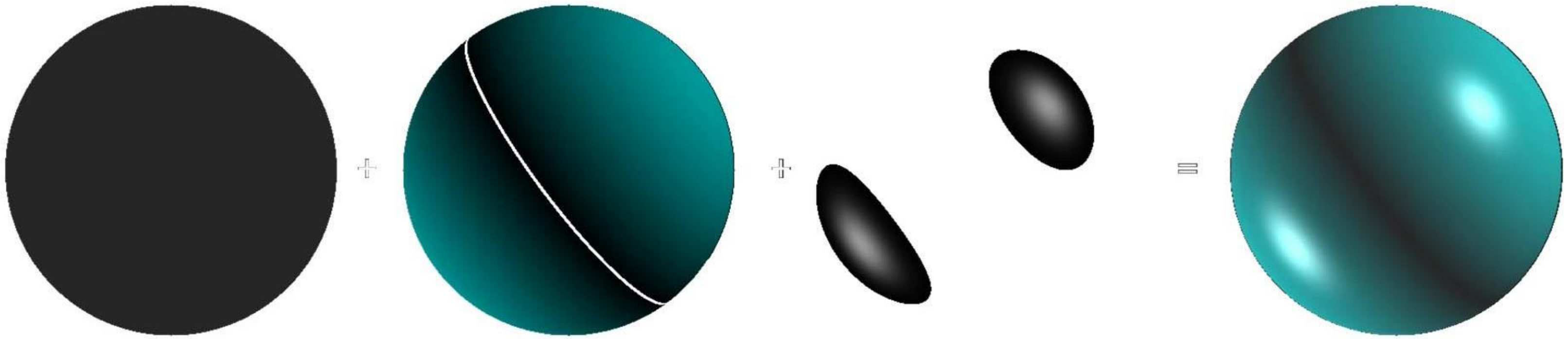
3. The Phong reflection model

- The total reflectance **radiance** is the addition of the diffuse, specular and ambient radiance

$$L(\omega_r) = k_a + \left(k_d(\mathbf{n}\cdot\mathbf{l}) + k_s(\mathbf{v}\cdot\mathbf{r})^q\right)\frac{\Phi_s}{4\pi\, d^2}$$

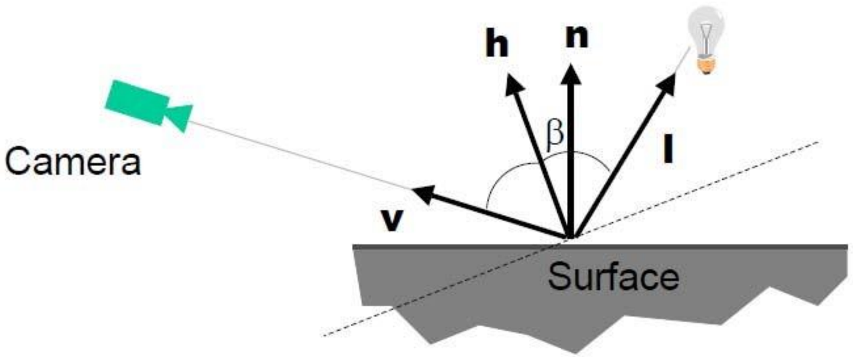
- It also can be described using a light source L_i with an indirect light L_a

$$L(\omega_r) = k_d L_a + \left(k_d(\mathbf{n}\cdot\mathbf{l}) + k_s(\mathbf{v}\cdot\mathbf{r})^q\right)L_i$$



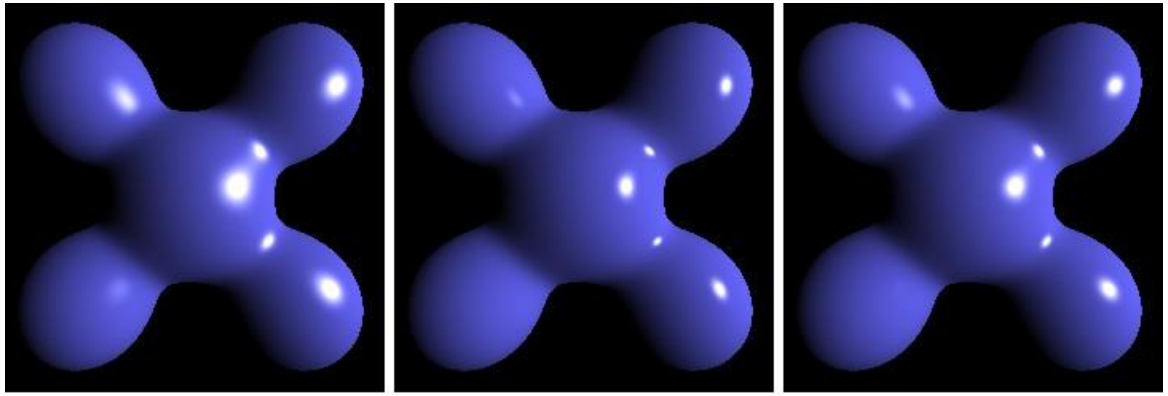
3. The Phong reflection model

- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex
- Blinn suggested **an approximation using the halfway vector** that is more efficient
- h is normalized vector halfway between l and v



$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{||\mathbf{l} + \mathbf{v}||}$$

$$L(\omega_r) = k_s (\cos\beta)^q \frac{\Phi_s}{4\pi d^2} = k_s (\mathbf{n} \cdot \mathbf{h})^q \frac{\Phi_s}{4\pi d^2}$$



Blinn-Phong | Phong | Blinn-Phong (Lower Exponent)

Resources

- [Kessenich] Kessenich et al. OpenGL Programming Guide. Chapter 7. Light and Shadow
- https://learnopengl.com/Lighting/Basic-Lighting
- http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/
- https://en.wikipedia.org/wiki/Phong_reflection_model