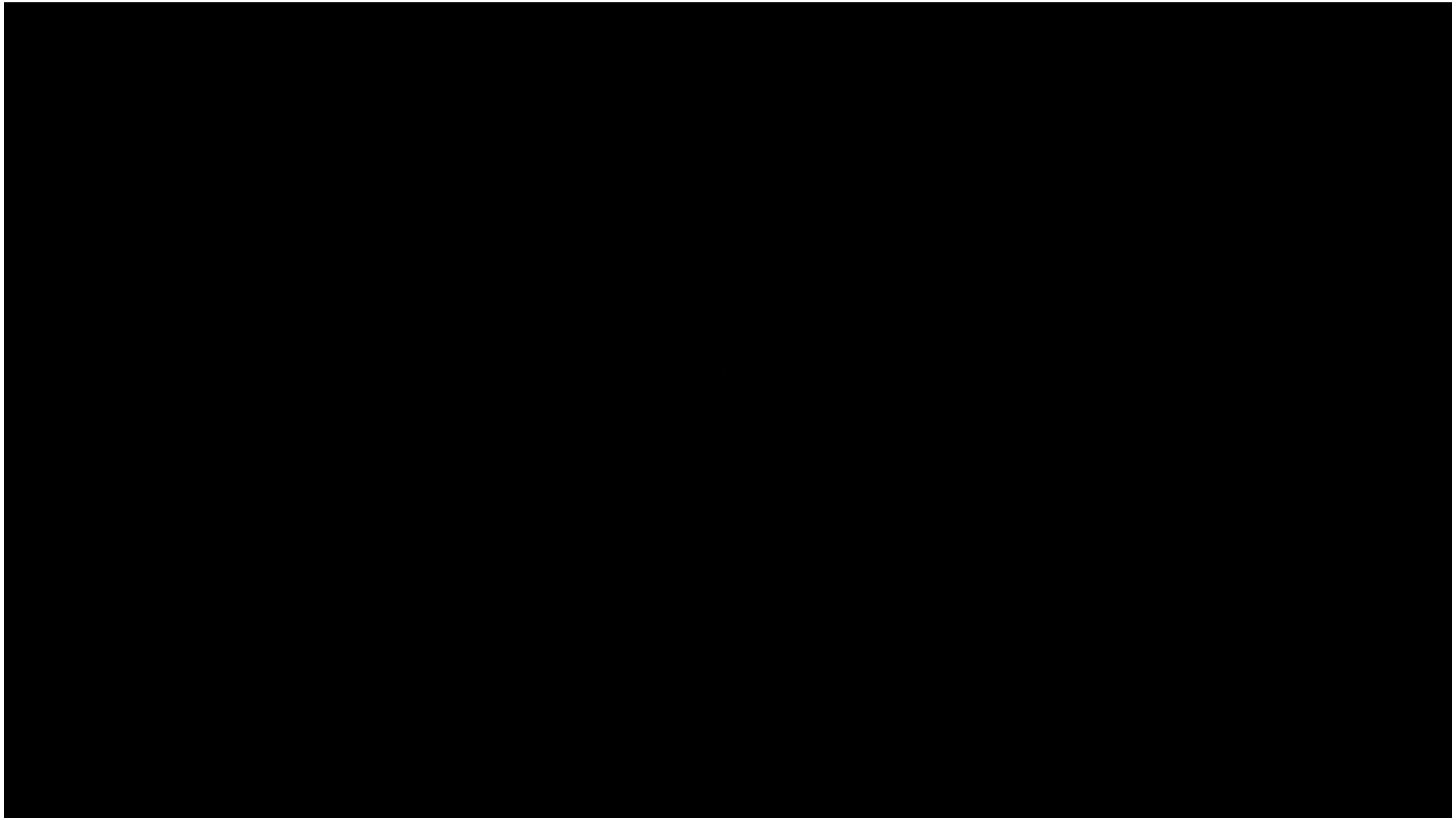
Lighting and Shading

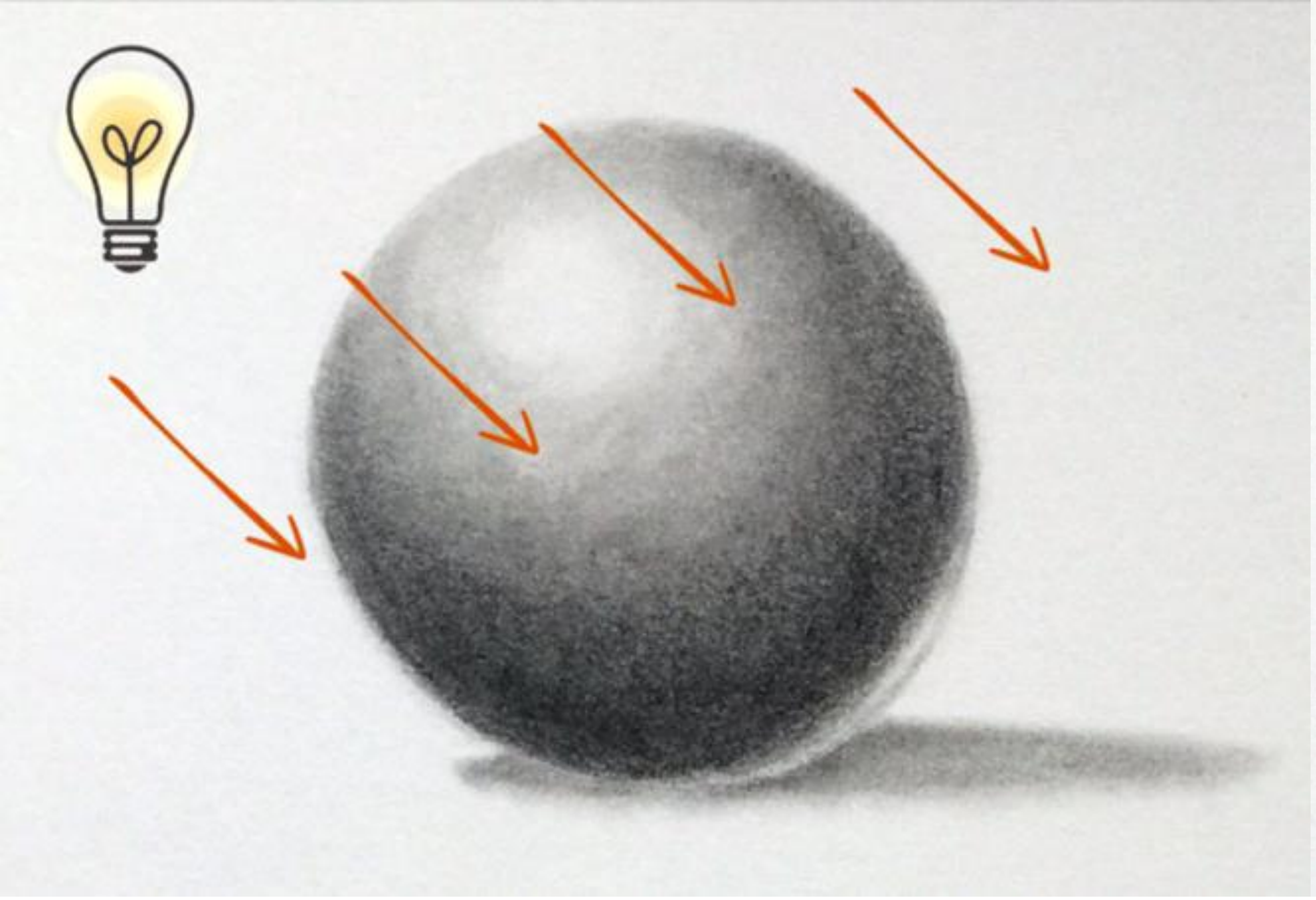
Bo Zhu

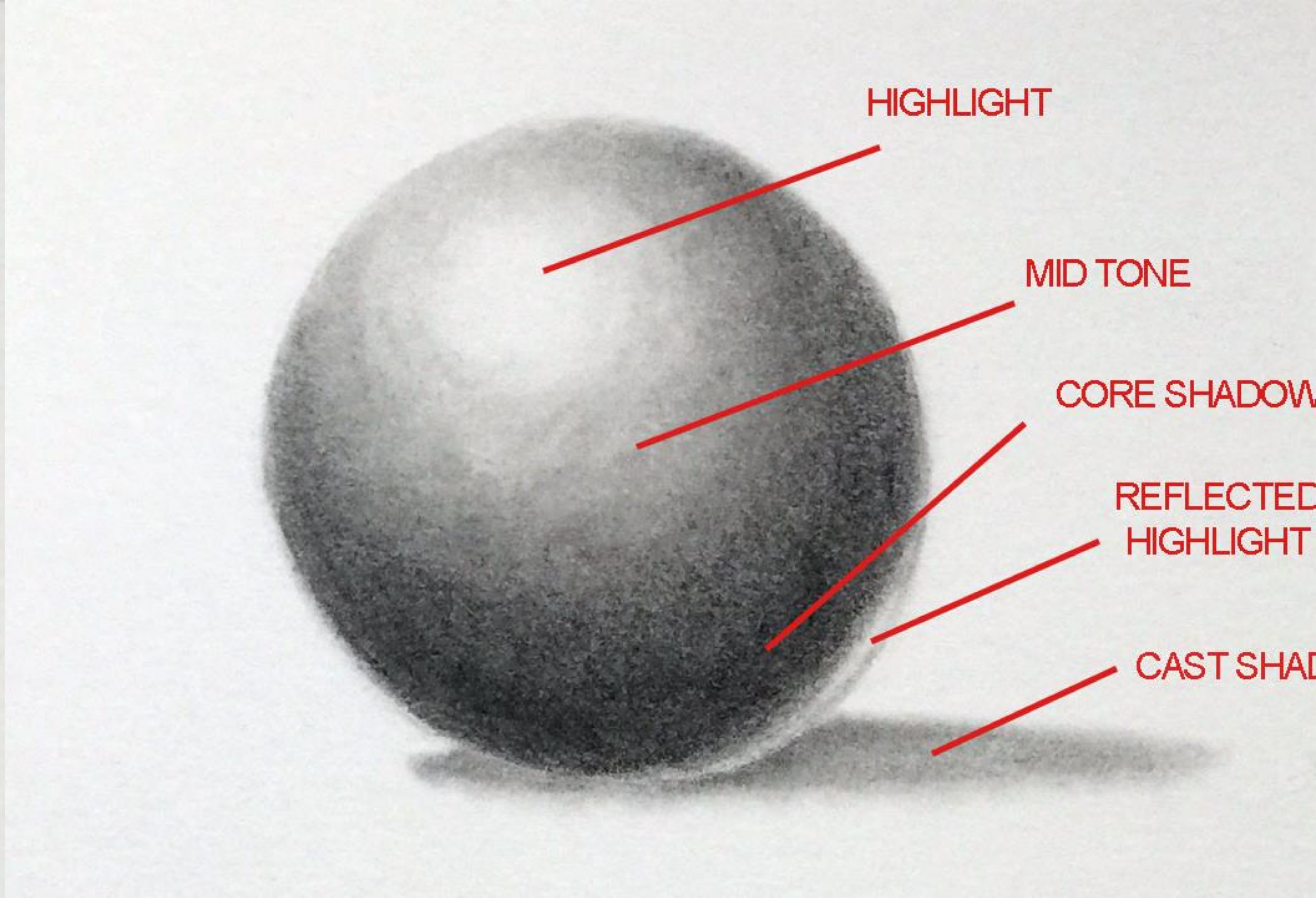
School of Interactive Computing

Georgia Institute of Technology


NVIDIA Marbles at Night: https://www.youtube.com/watch?v=NgcYLIvlp_k

Motivational Video:
Art Lesson for Kids: Shading a Sphere



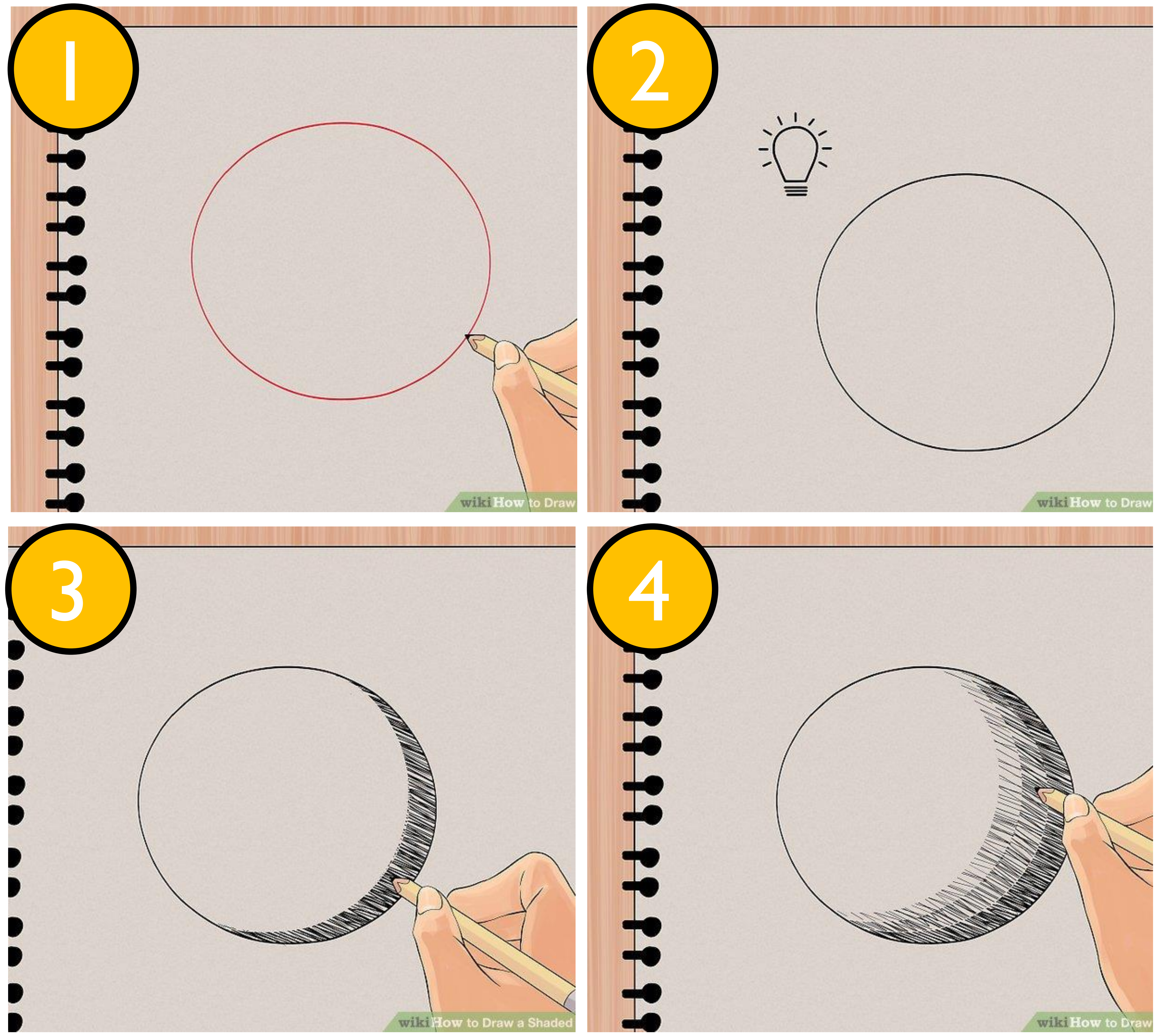


The Illusion of Light on a Sphere

- Light is how we see and shading informs us of the light within a scene.
- Light values are called tints and dark values are called shades.
- Contrast deals with difference.

https://thevirtualinstructor.com/shading-techniques-basics.html

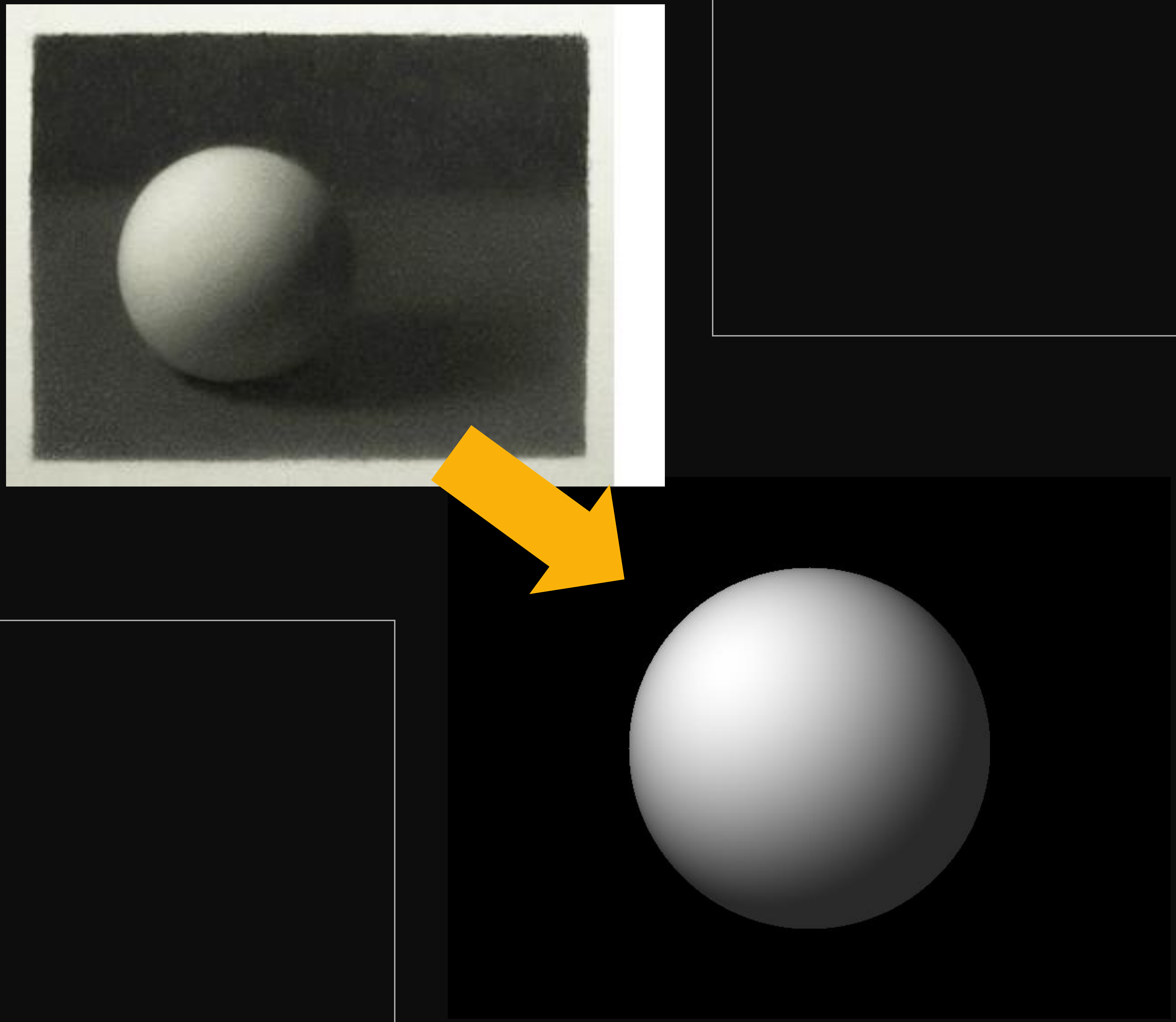
Steps of Shading a Sphere

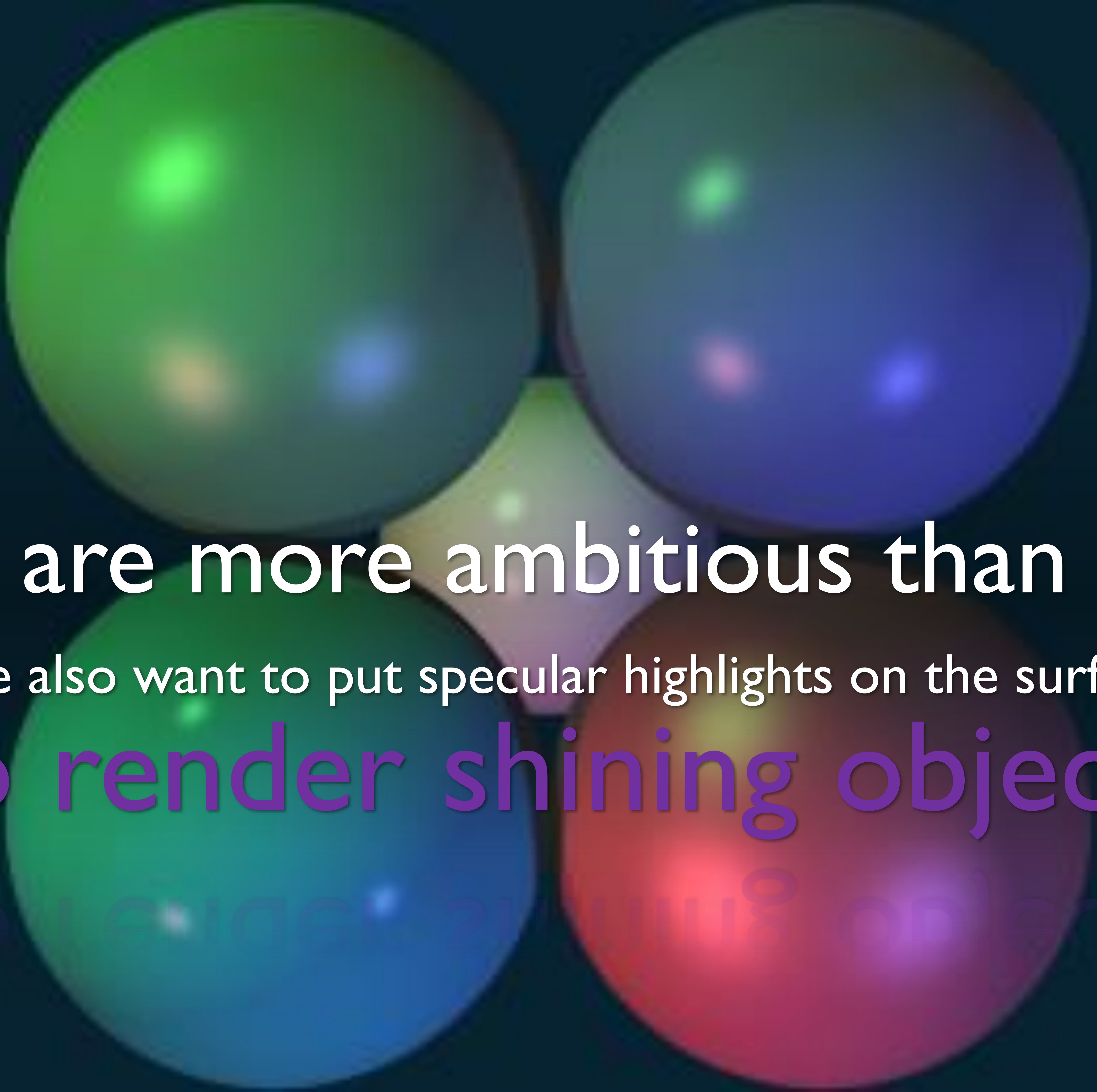


- Draw a **hollow circle** with a light line.
- Choose a **light source** for this sphere.
- Begin **shading** by starting opposite the light source and move inwards.
- As you move towards the light source, **fill the circle** with less and less pressure, heavier on the dark area, lighter on the light area.

How do we let computers to shade a sphere?

Shading a surface with math and GLSL



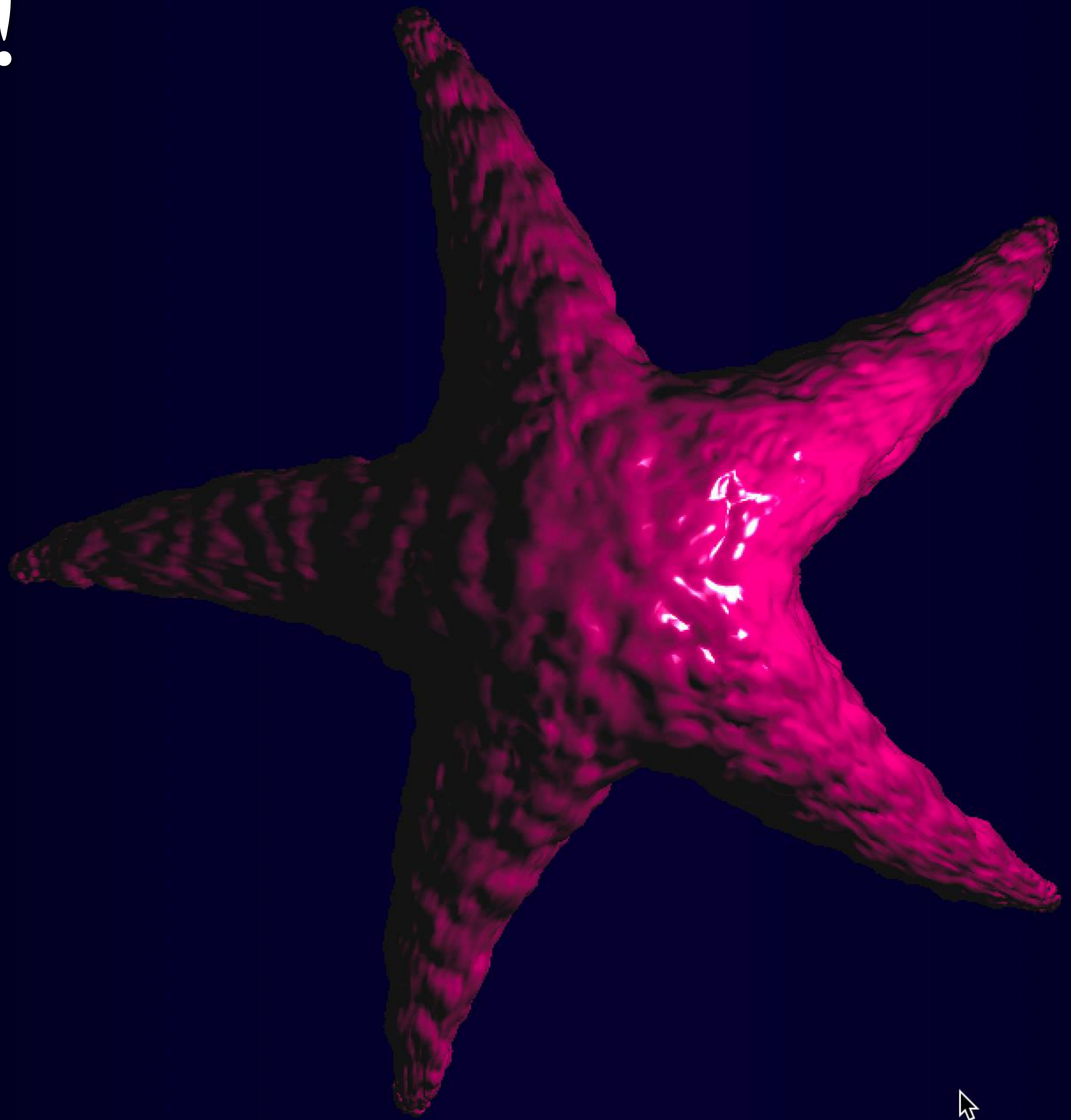


We are more ambitious than that

We also want to put specular highlights on the surface

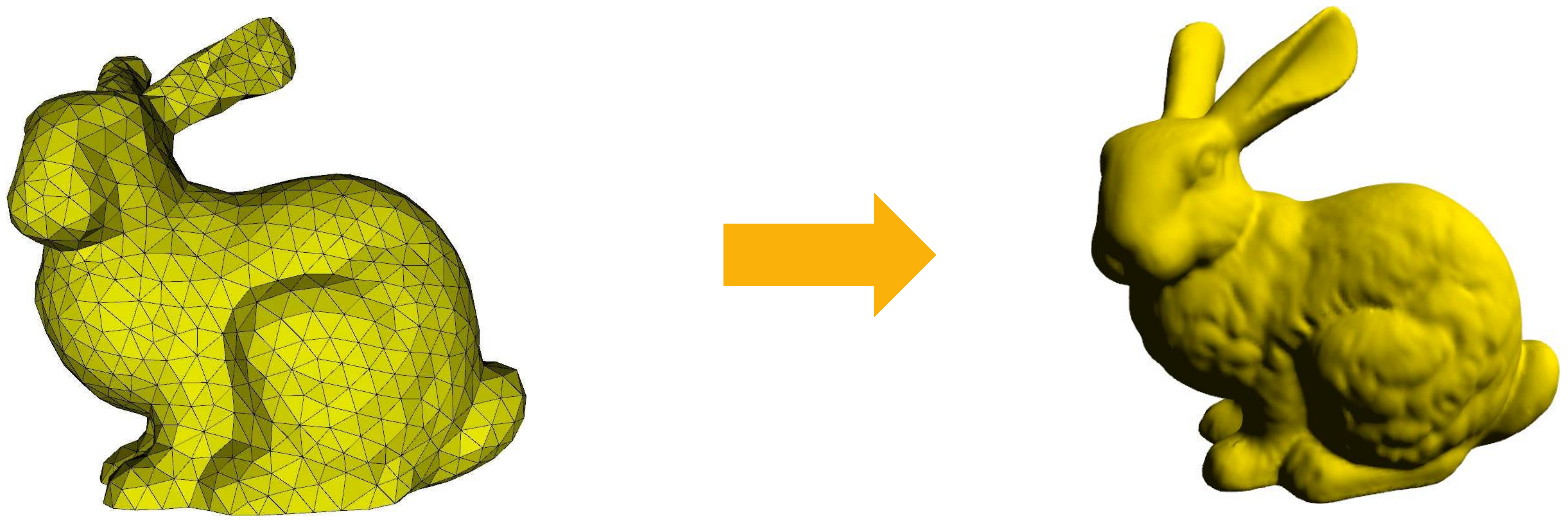
to render shining objects

like this!



Video Credit: Logan Chang, Dartmouth 24’

Today’s Study Goal: Shading a Bunny with GLSL!



Study Plan

- Vertex Normal
- Barycentric Interpolation
- Lights
- Shading Model
    - Ambient
    - Diffuse
    - Specular
    - Put Things Together: Phong Shading
- Shader Implementation

Vertex Normal

Quick Recap: Cross Product

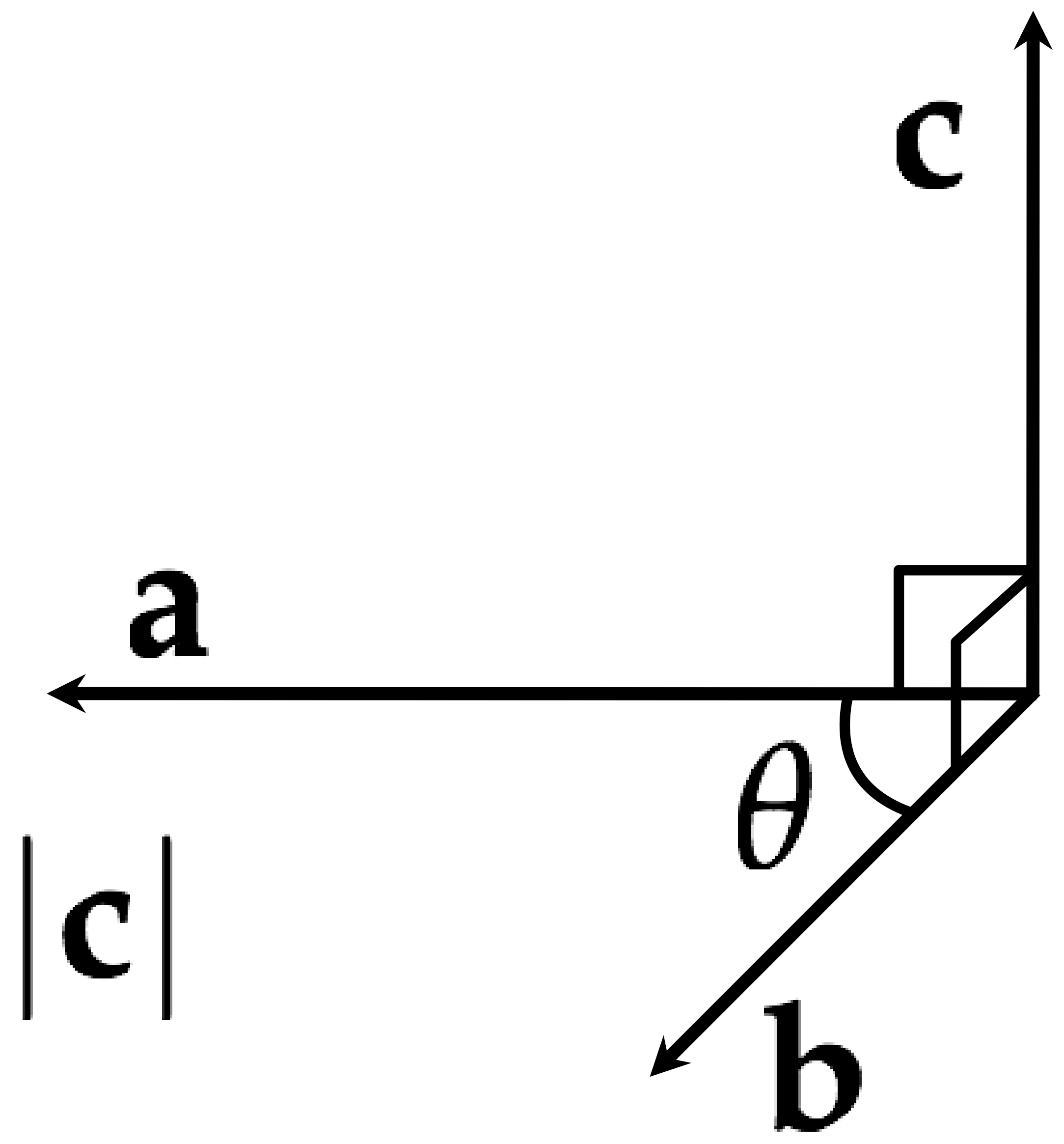
- Geometric definition:
- $\boldsymbol{a} \times \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| sin\theta$
- right-hand rule to define direction

- Coordinate formula

$$\mathbf{a} \times \mathbf{b} := \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$

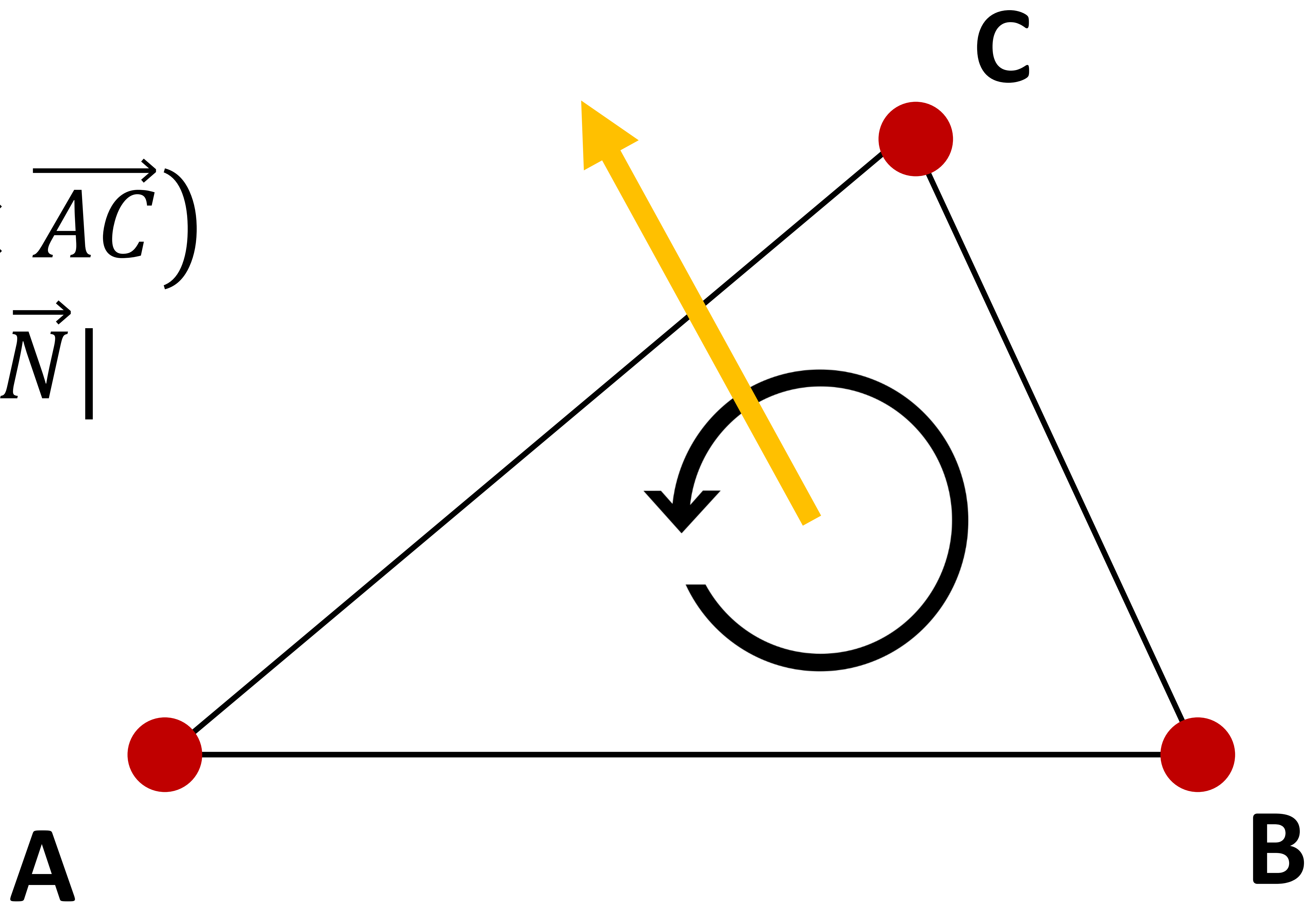
related to determinant:

$$\begin{vmatrix} \mathbf{x} & \mathbf{y} & \mathbf{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$



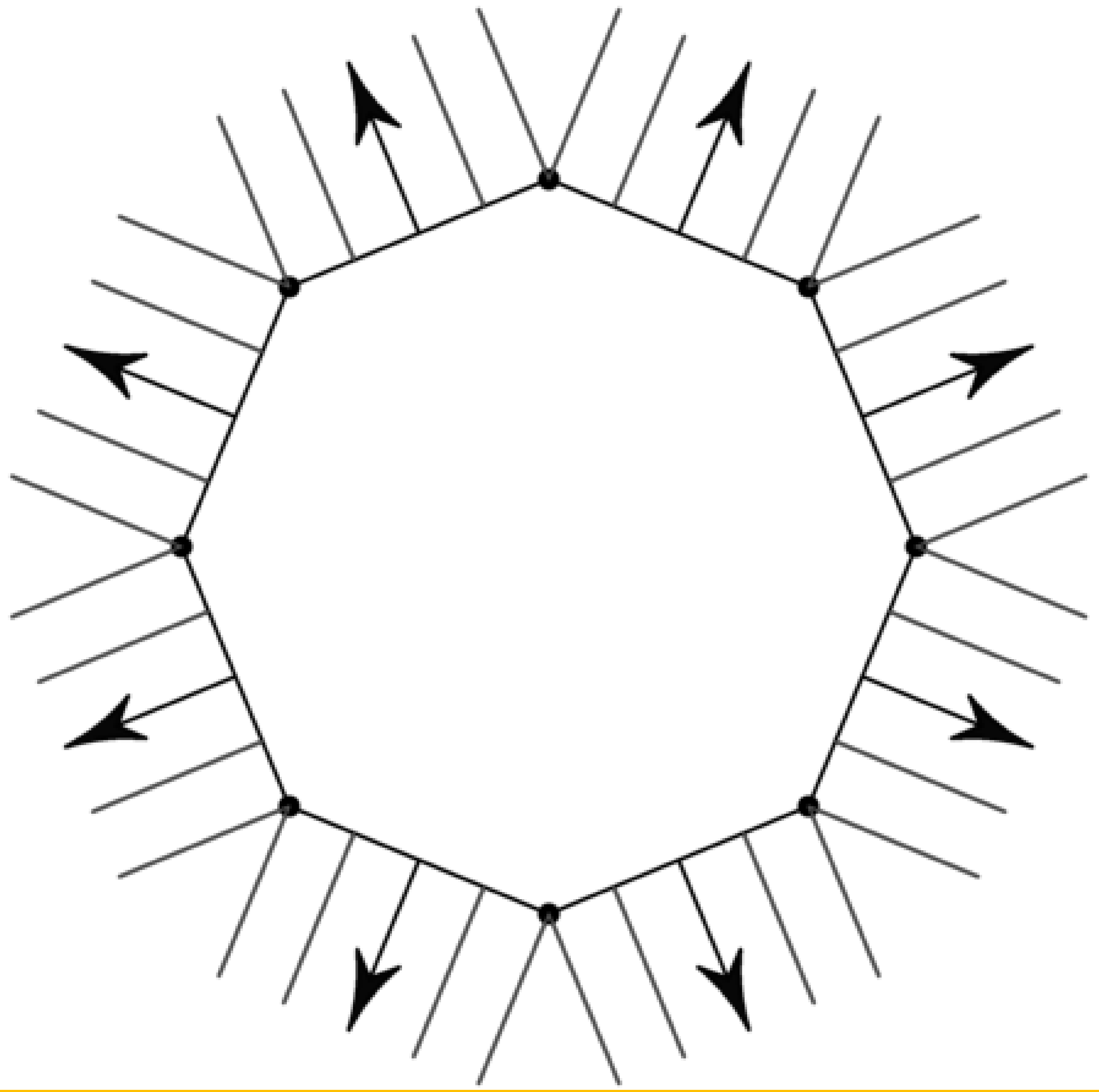
Triangle Normal

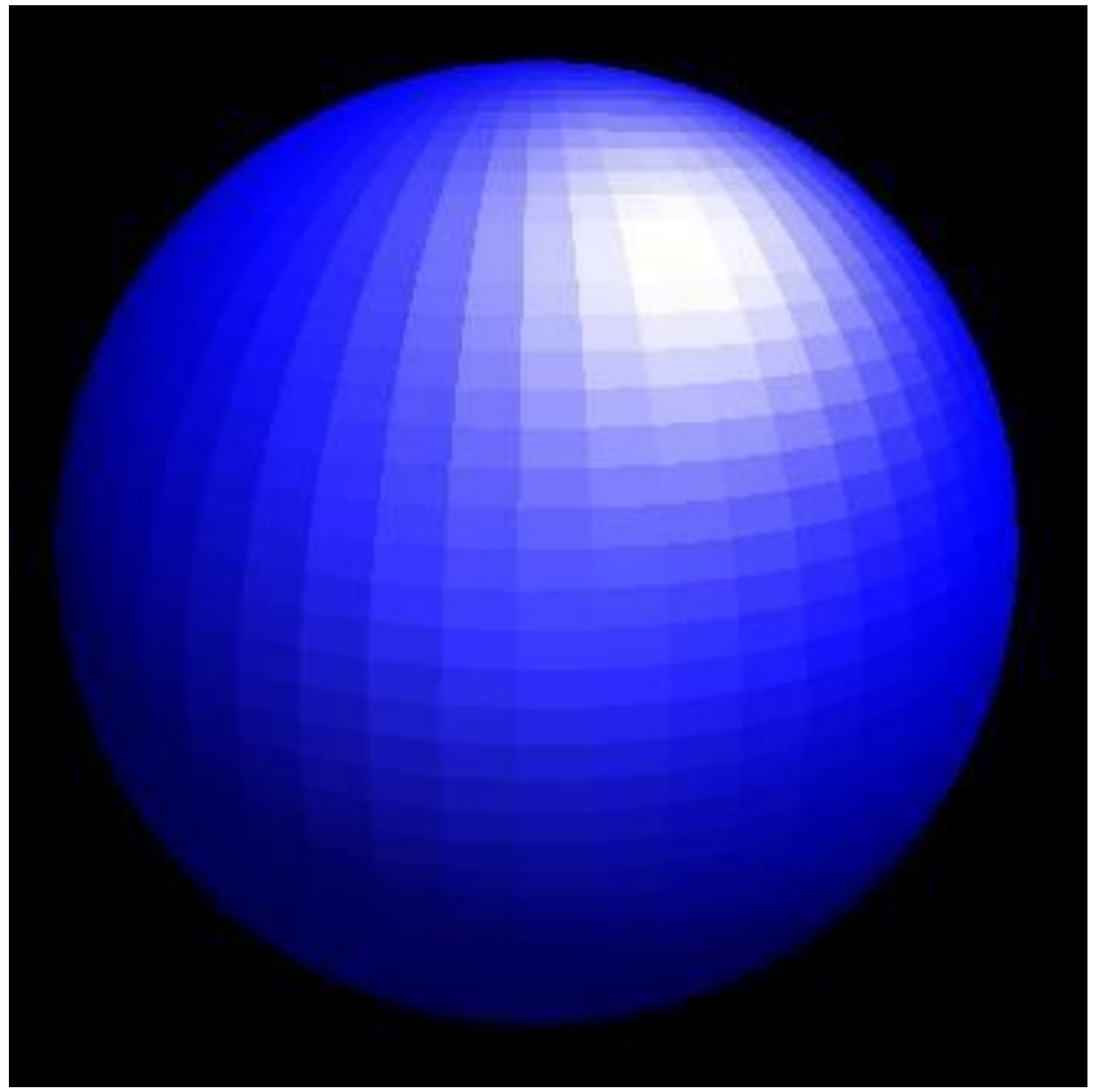
$$\vec{N} = (\overrightarrow{AB} \times \overrightarrow{AC})$$
$$\vec{N} = \vec{N}/|\vec{N}|$$



We can define a normal vector for each triangle

- Face normals: same normal for all points in face
  - geometrically correct, but faceted look

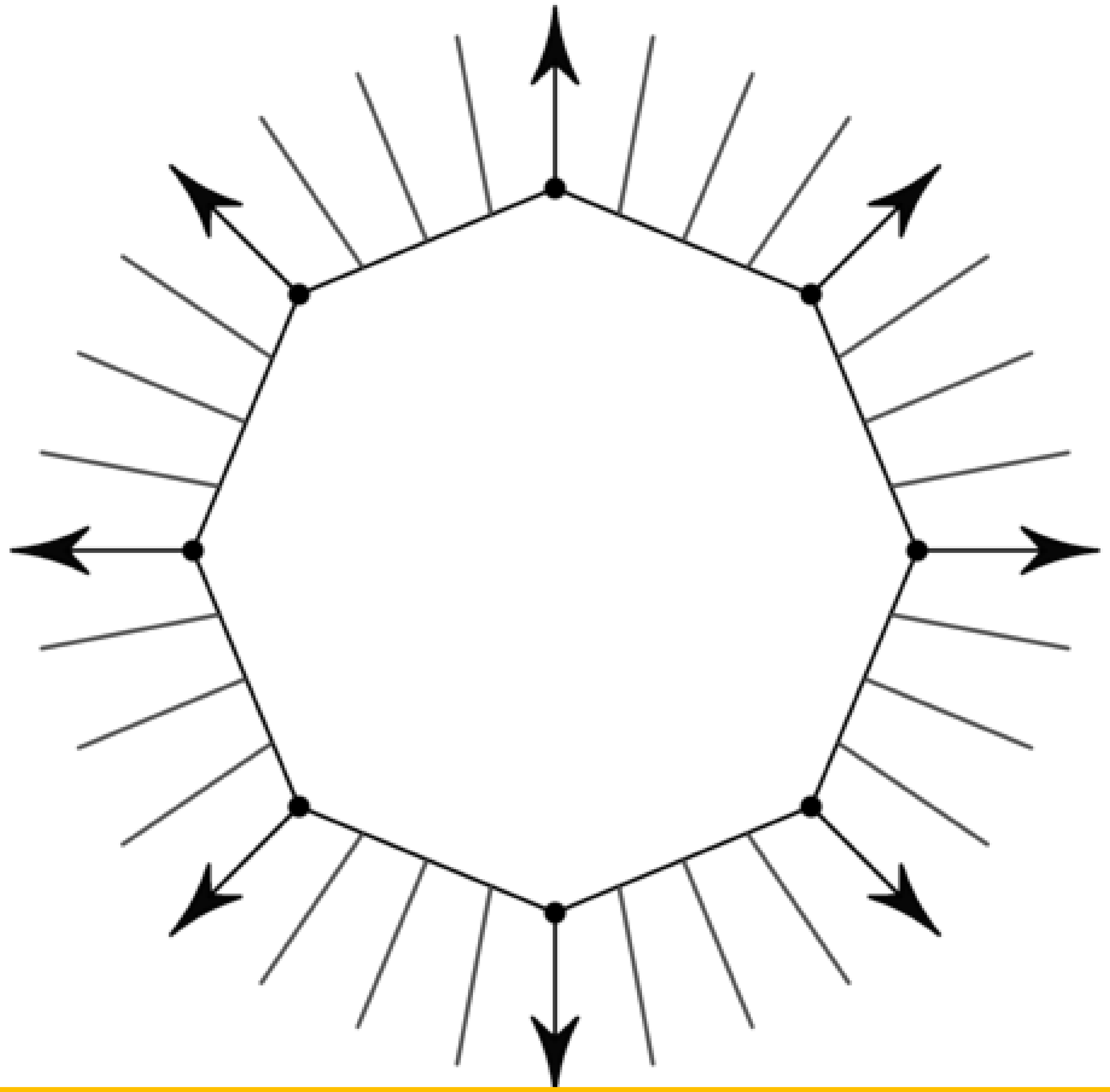


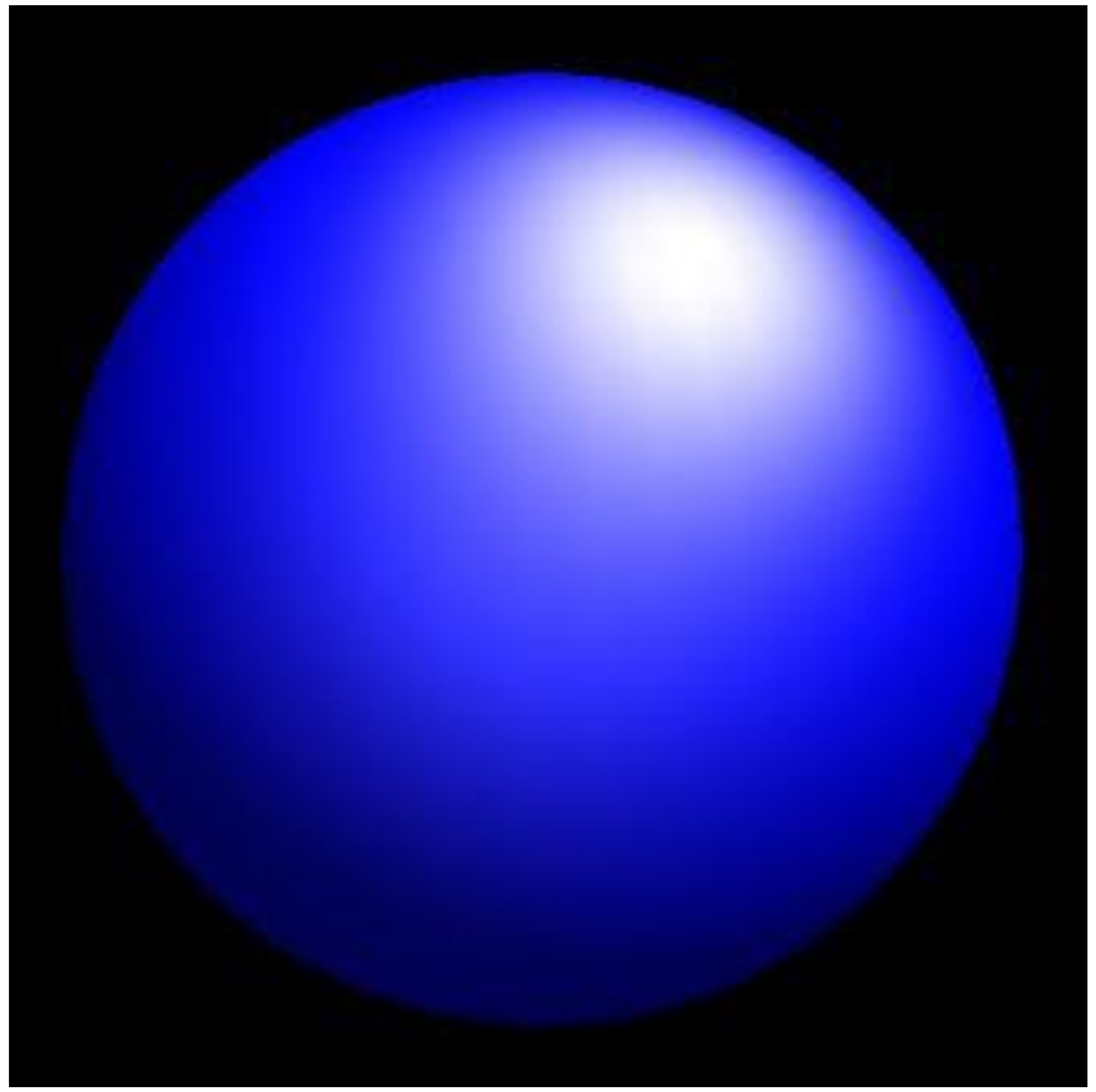


Not great for rendering a smooth surface!

Or, we can calculate a normal vector for each vertex

- Vertex normal: store normal at vertices, interpolate in face
  - geometrically "inconsistent", but smooth look





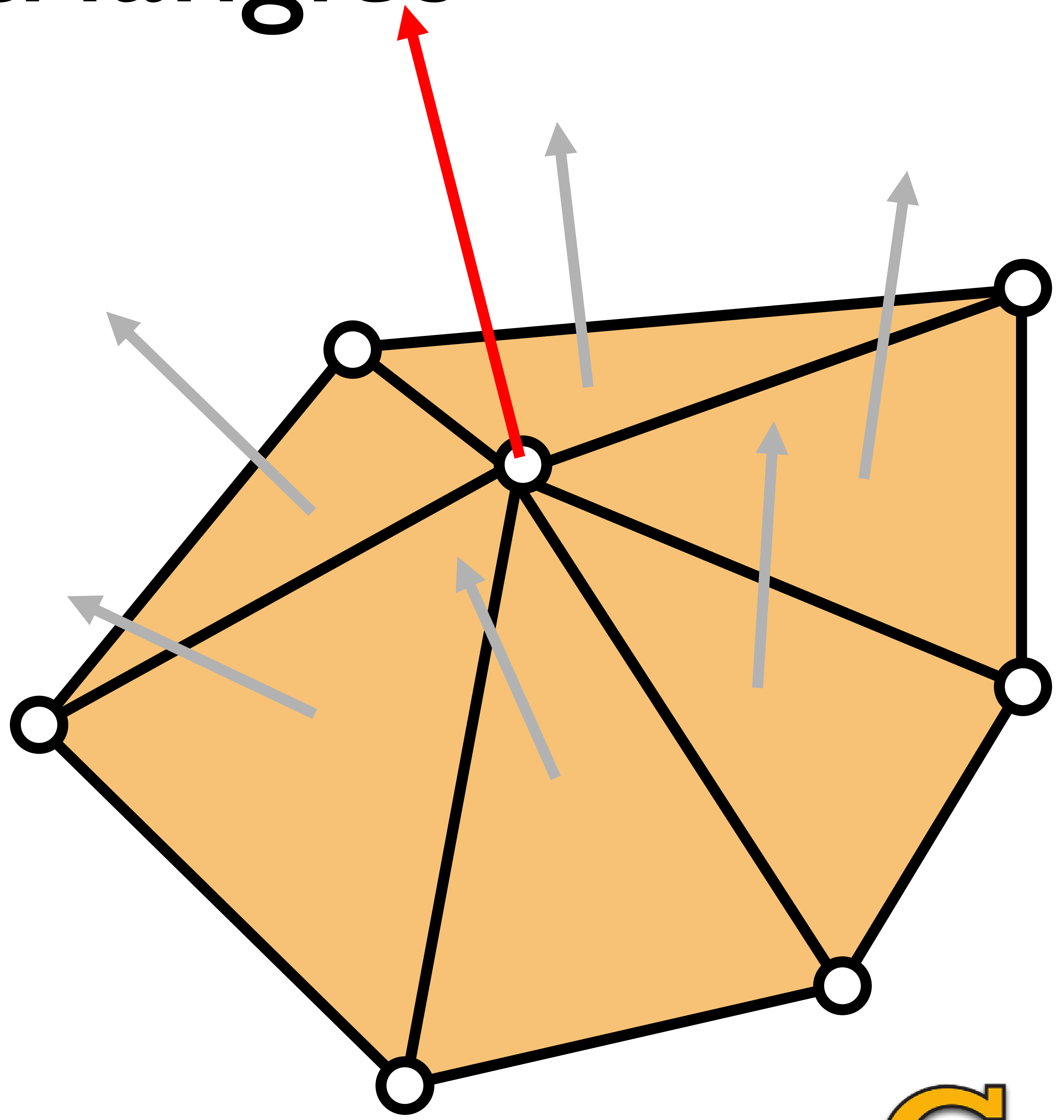
Then how to calculate a vertex normal?

Vertex Normal

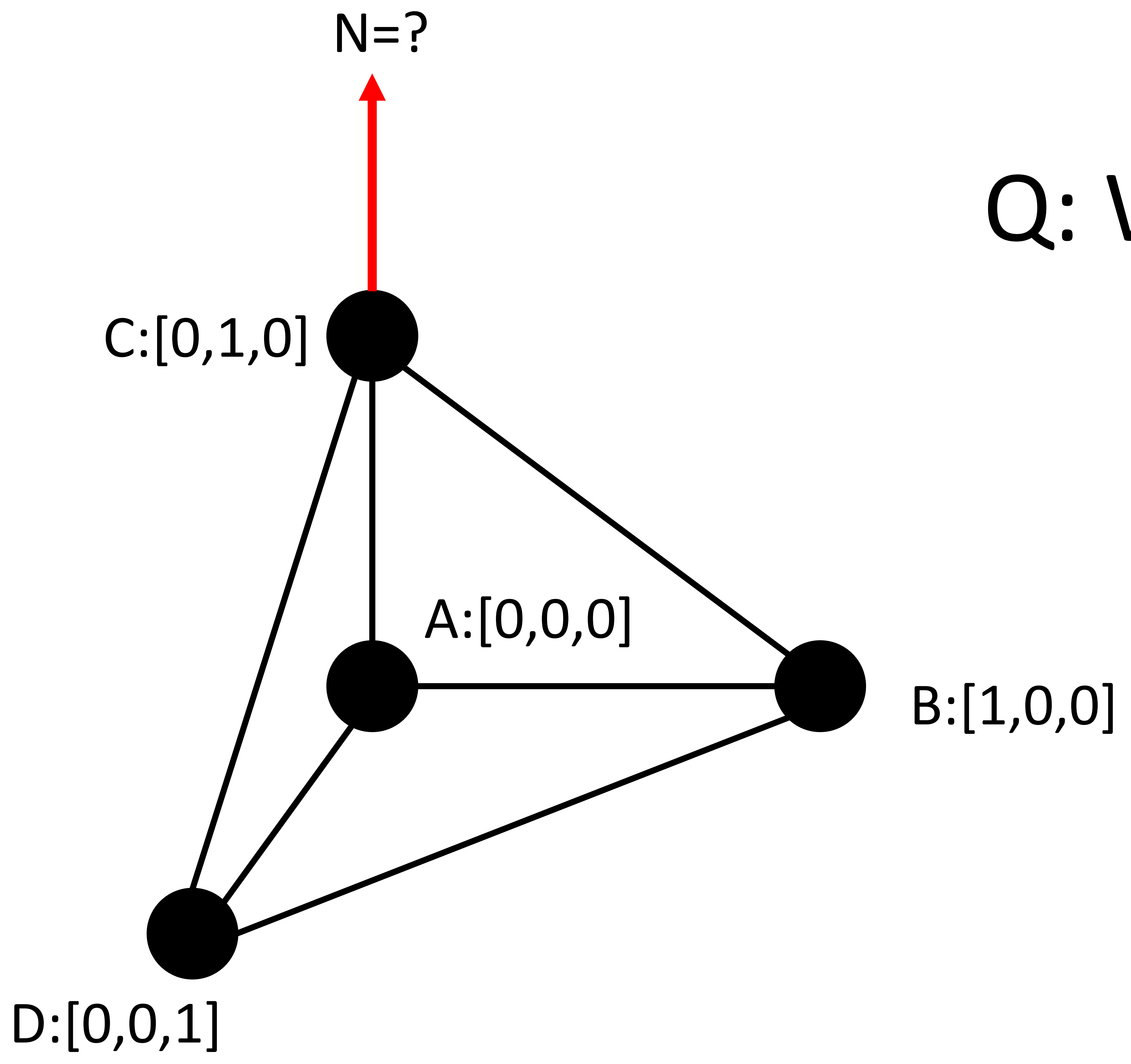
The normal vector of a vertex is the average of the normal vectors of its incident triangles

Implementation Idea: Sum all normal vectors from incident triangles, and then normalize

$$\vec{N}_v = \sum_i \vec{N}_i / |\sum_i \vec{N}_i|$$



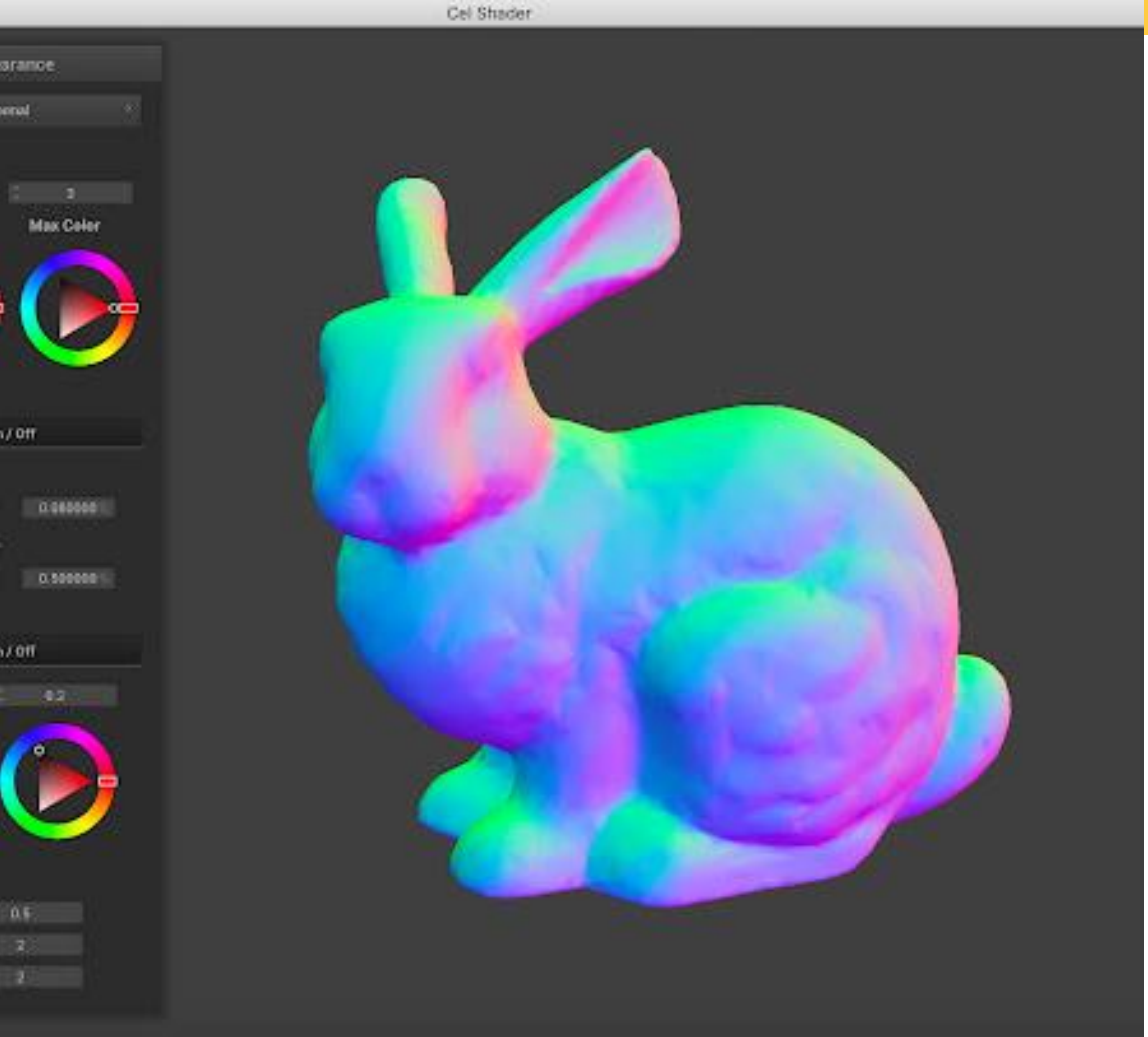
Practice: Calculate Vertex Normal



Q: What is the normal of vertex C?

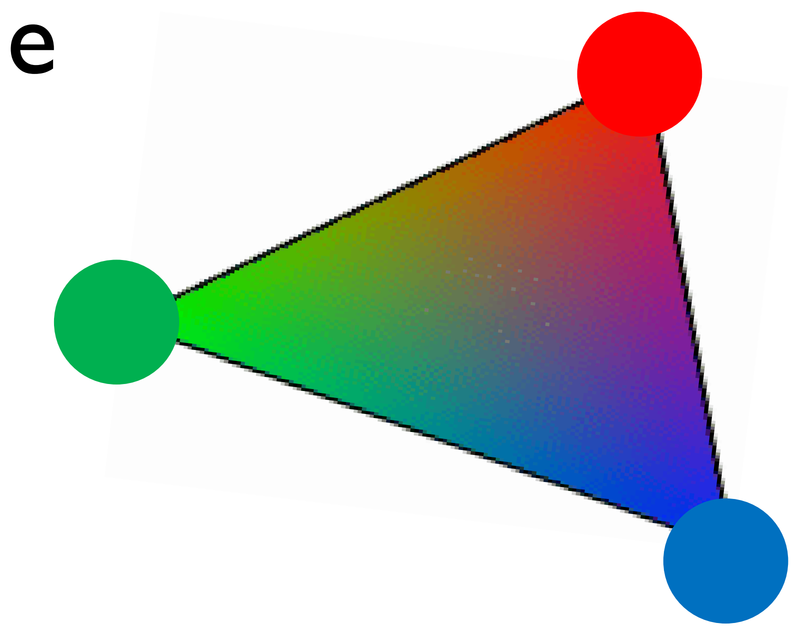
Formula of cross product:

$$\mathbf{a} \times \mathbf{b} := \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix}$$



Visualize vertex normal on a mesh?

- A nice visual trick is to draw the **xyz** of a vertex normal as **rgb** colors
- We interpolate colors within each triangle using barycentric interpolation scheme



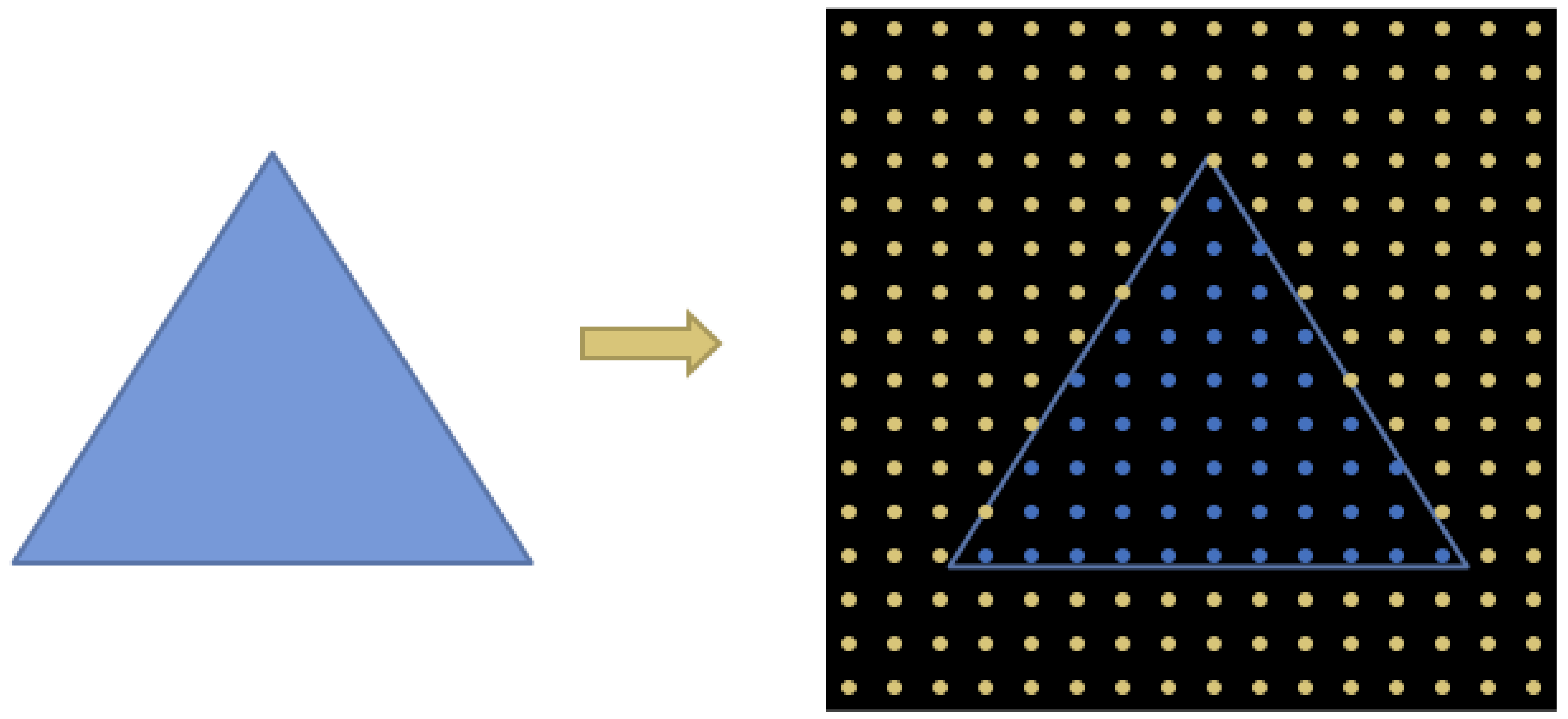
Next question: How do we interpolate with a triangle?

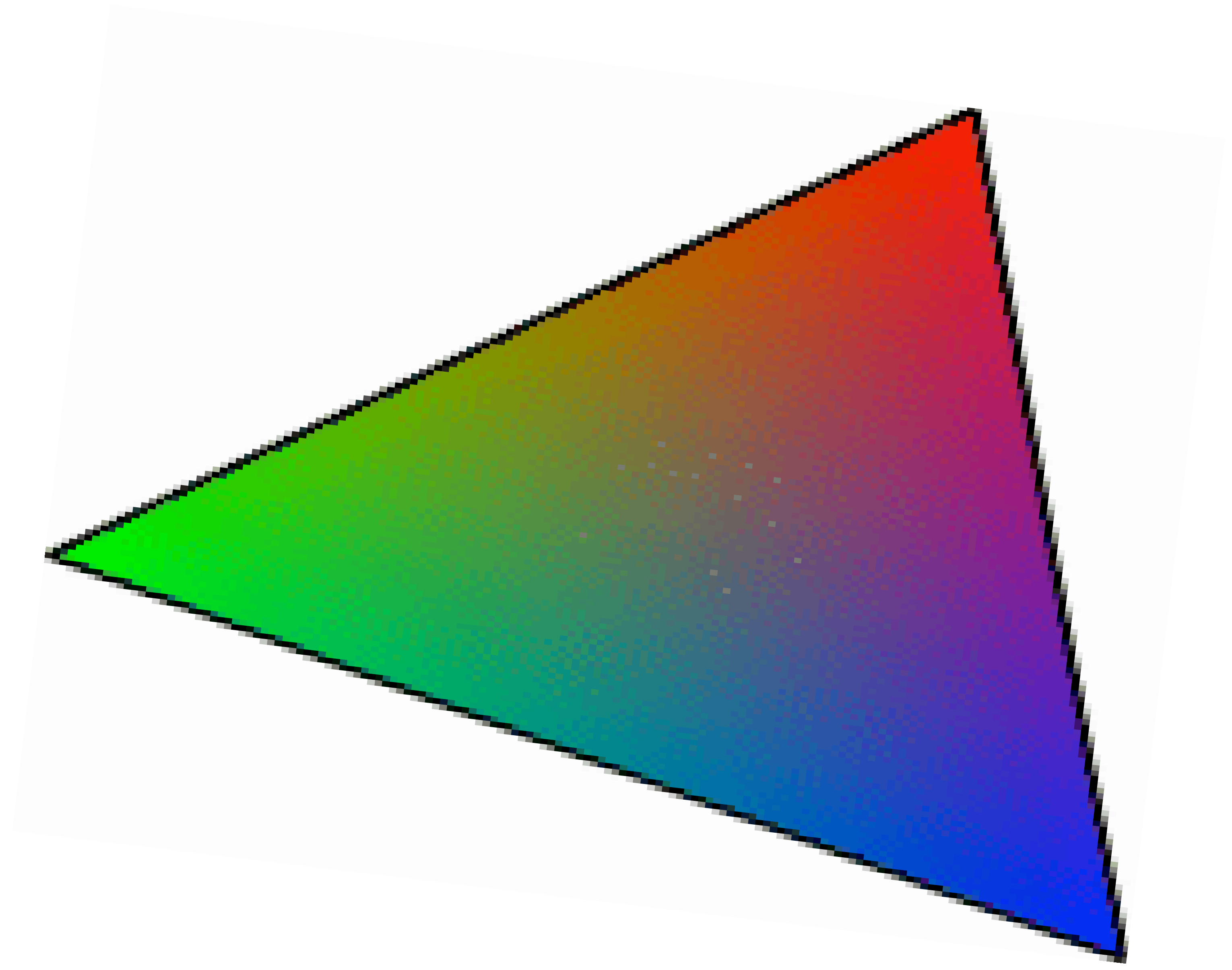
Triangle Interpolation

Barycentric Interpolation

Triangle Color

- Every pixel inside a triangle projected onto the image plane requires (R,G,B) values
- If the triangle has a uniform color, one can simply color every interior pixel accordingly
- However, if a triangle has multiple colors, we need to specify the color for every pixel
- Since the pixel locations aren't known ahead of time, we need a way of specifying color throughout the interior of the triangle

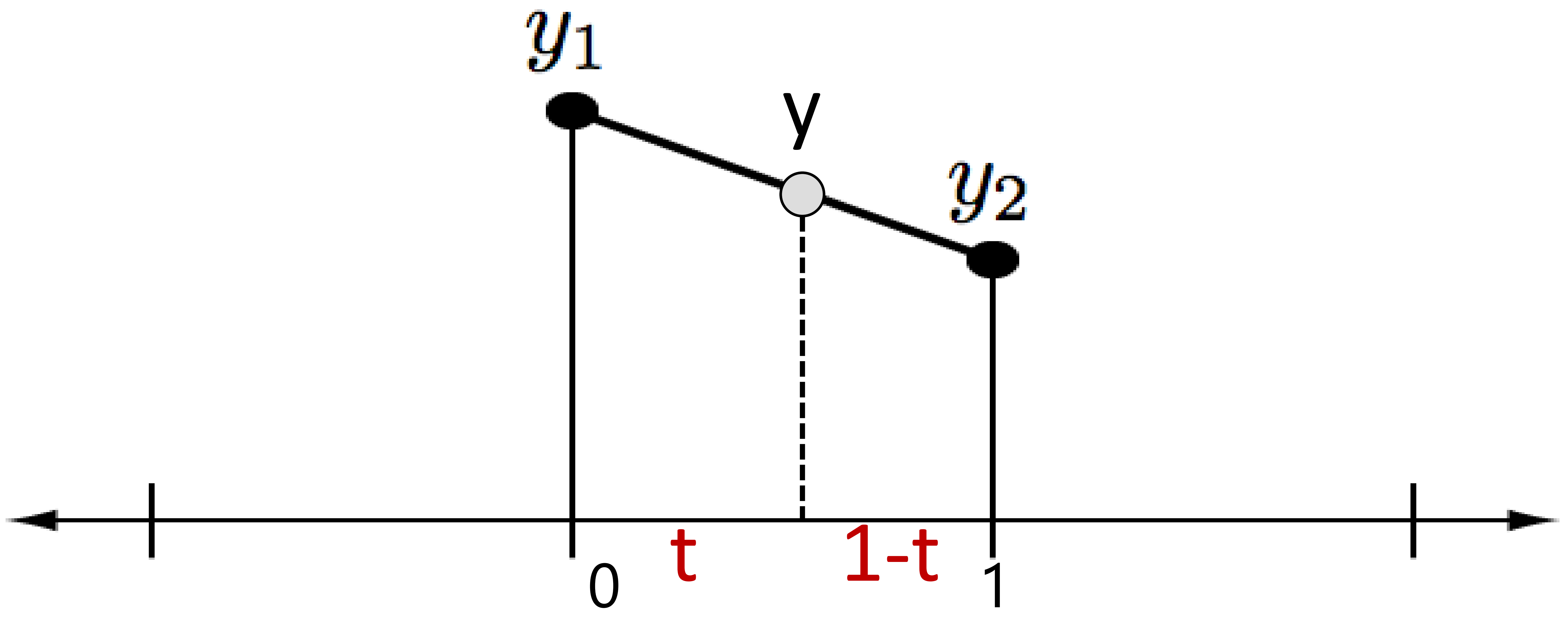




Interpolation

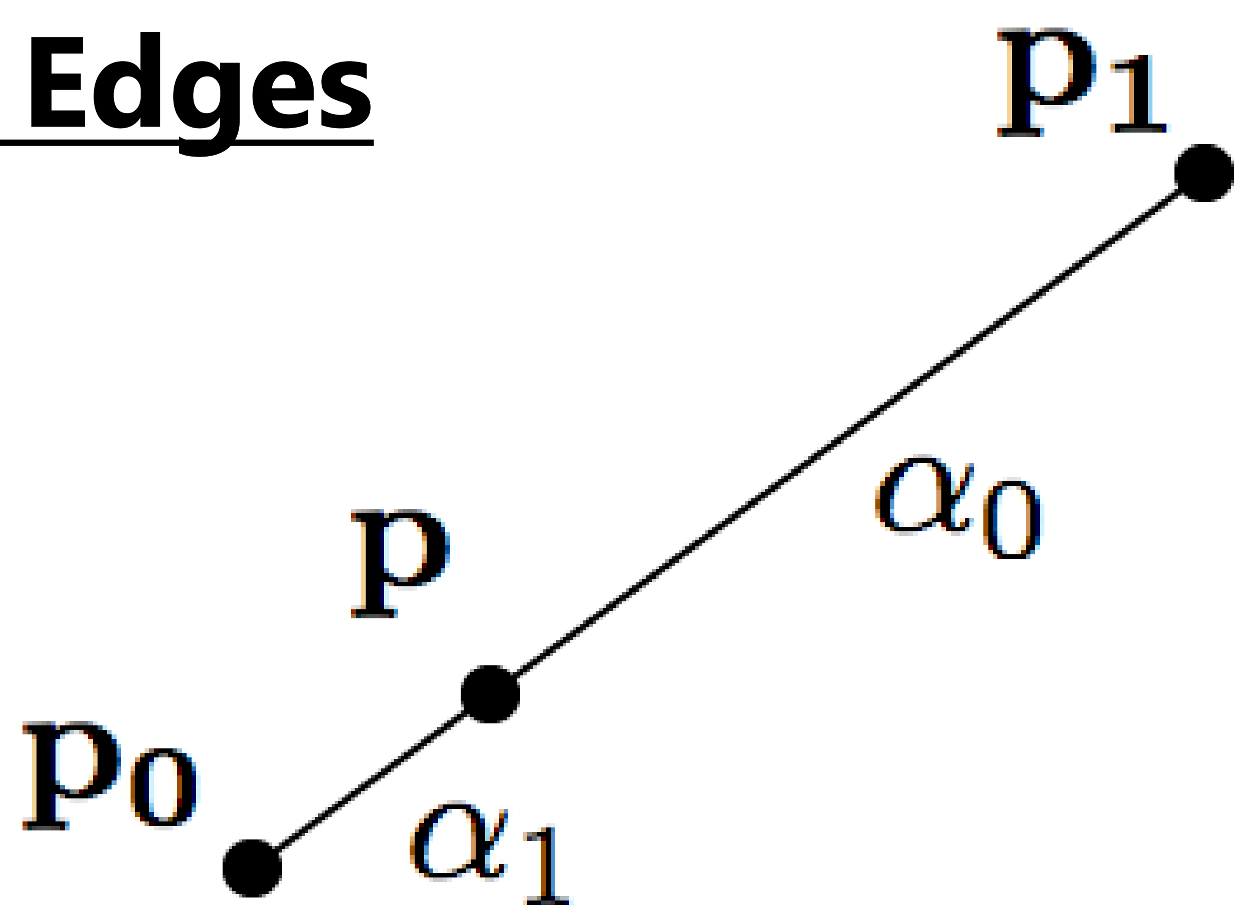
- **Convert discrete values into a continuous function by filling in all the "in between" values**

Linear Interpolation $y(t) = (1 - t)\, y_1 + t\, y_2$



Barycentric Interpolation

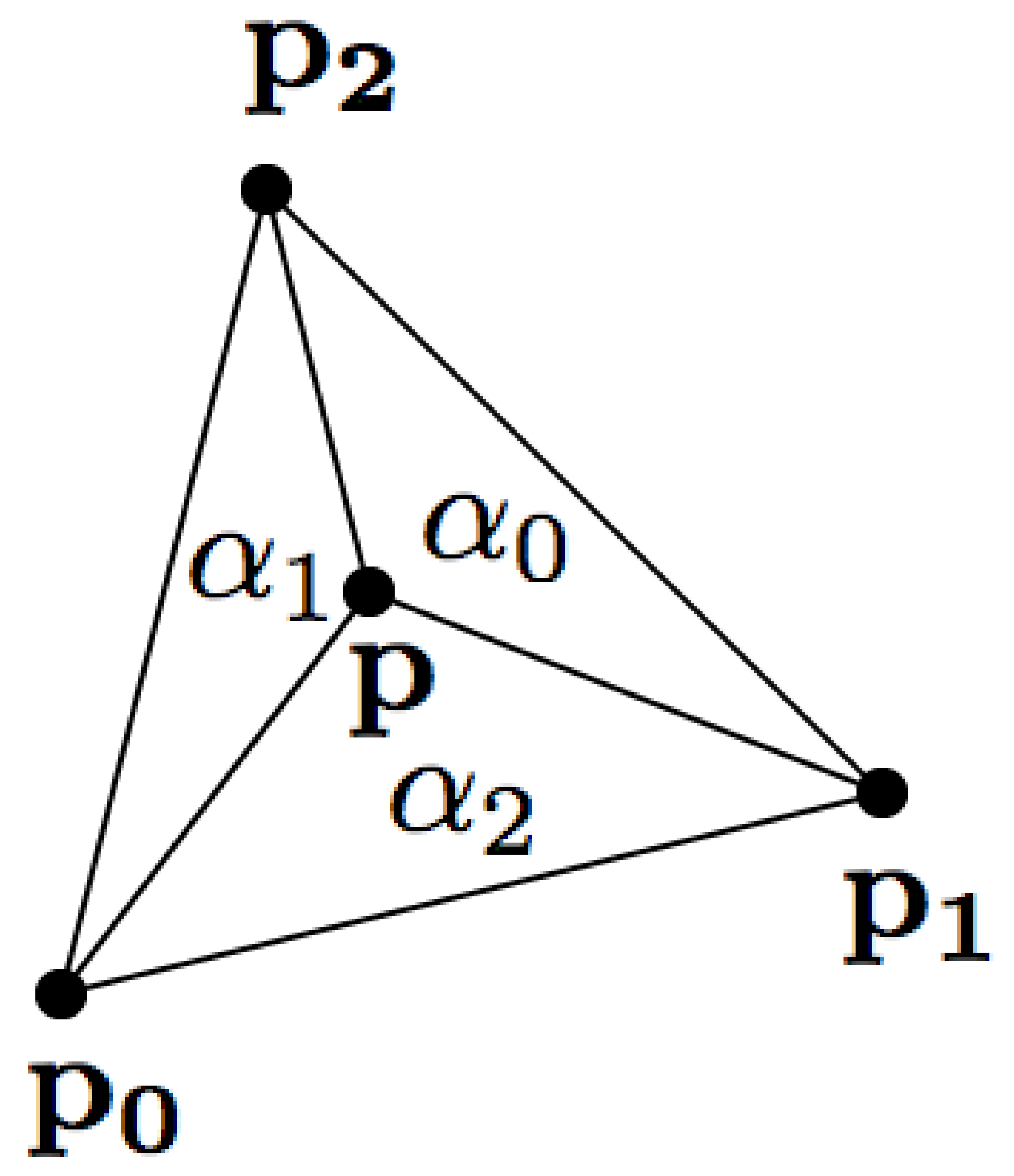
2D/3D Edges



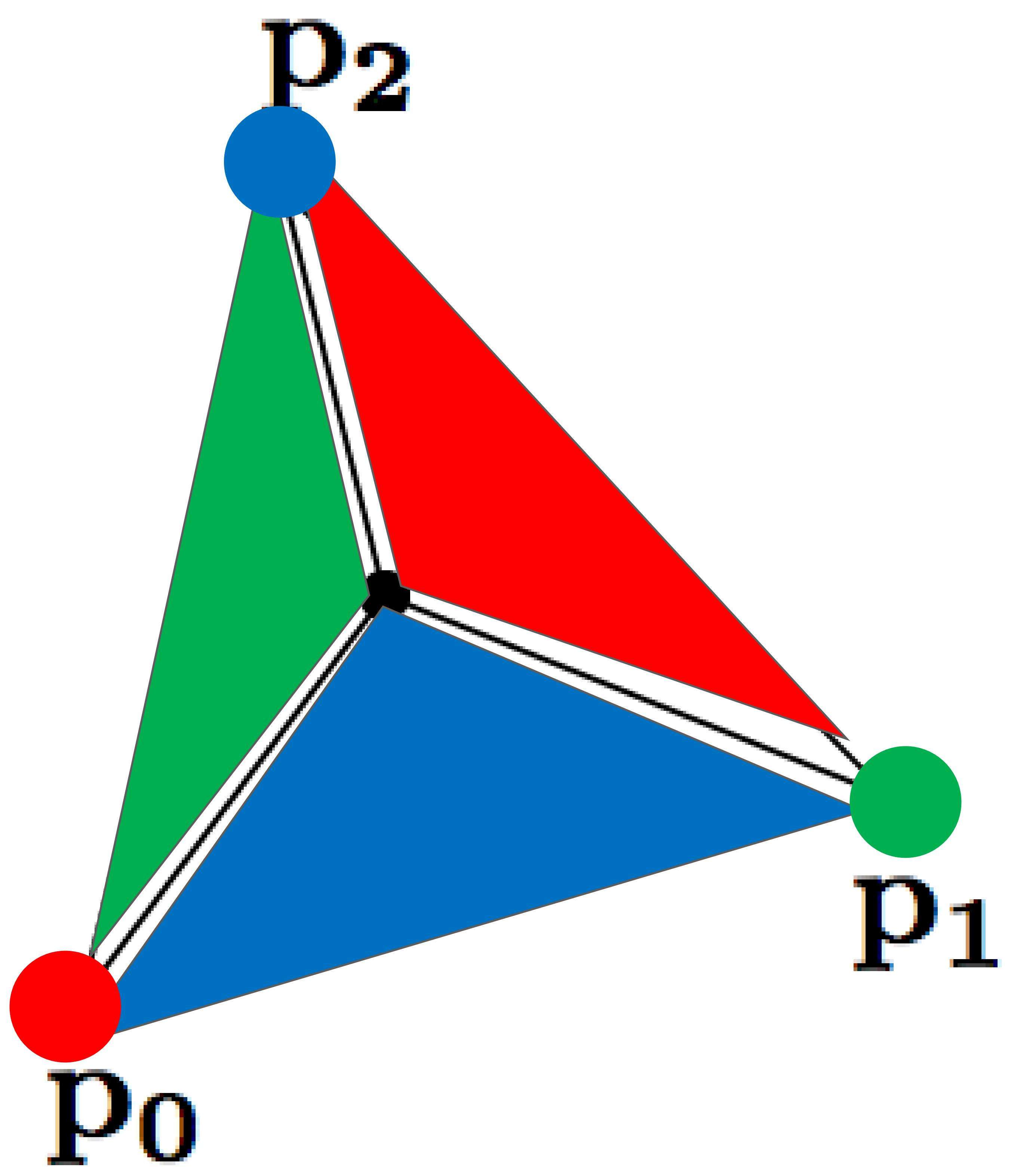
$$\mathbf{p} = \alpha_0 \mathbf{p_0} + \alpha_1 \mathbf{p_1}$$
$$\alpha_0 + \alpha_1 = 1$$

2D/3D Triangles

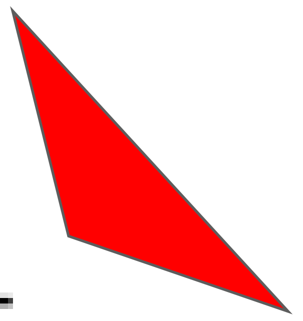
$$\mathbf{p} = \alpha_0 \, \mathbf{p_0} + \alpha_1 \, \mathbf{p_1} + \alpha_2 \, \mathbf{p_2}$$
$$\alpha_0 + \alpha_1 + \alpha_2 = 1$$



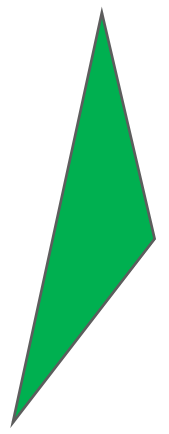
Geometric Approach to Calculate Barycentric Weights



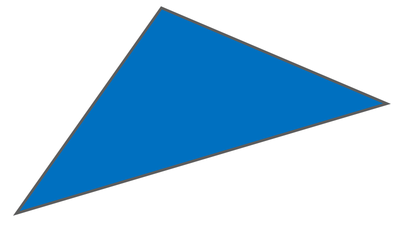
$$\alpha_0 = \frac{\text{area}(\mathbf{p}\,\mathbf{p}_1\,\mathbf{p}_2)}{\text{area}(\mathbf{p}_0\,\mathbf{p}_1\,\mathbf{p}_2)}$$



$$\alpha_1 = \frac{\text{area}(\mathbf{p}_0\,\mathbf{p}\,\mathbf{p}_2)}{\text{area}(\mathbf{p}_0\,\mathbf{p}_1\,\mathbf{p}_2)}$$



$$\alpha_2 = \frac{\text{area}(\mathbf{p}_0\,\mathbf{p}_1\,\mathbf{p})}{\text{area}(\mathbf{p}_0\,\mathbf{p}_1\,\mathbf{p}_2)}$$



$$\text{area}(A,B,C) = \frac{|AB \times AC|}{2}$$

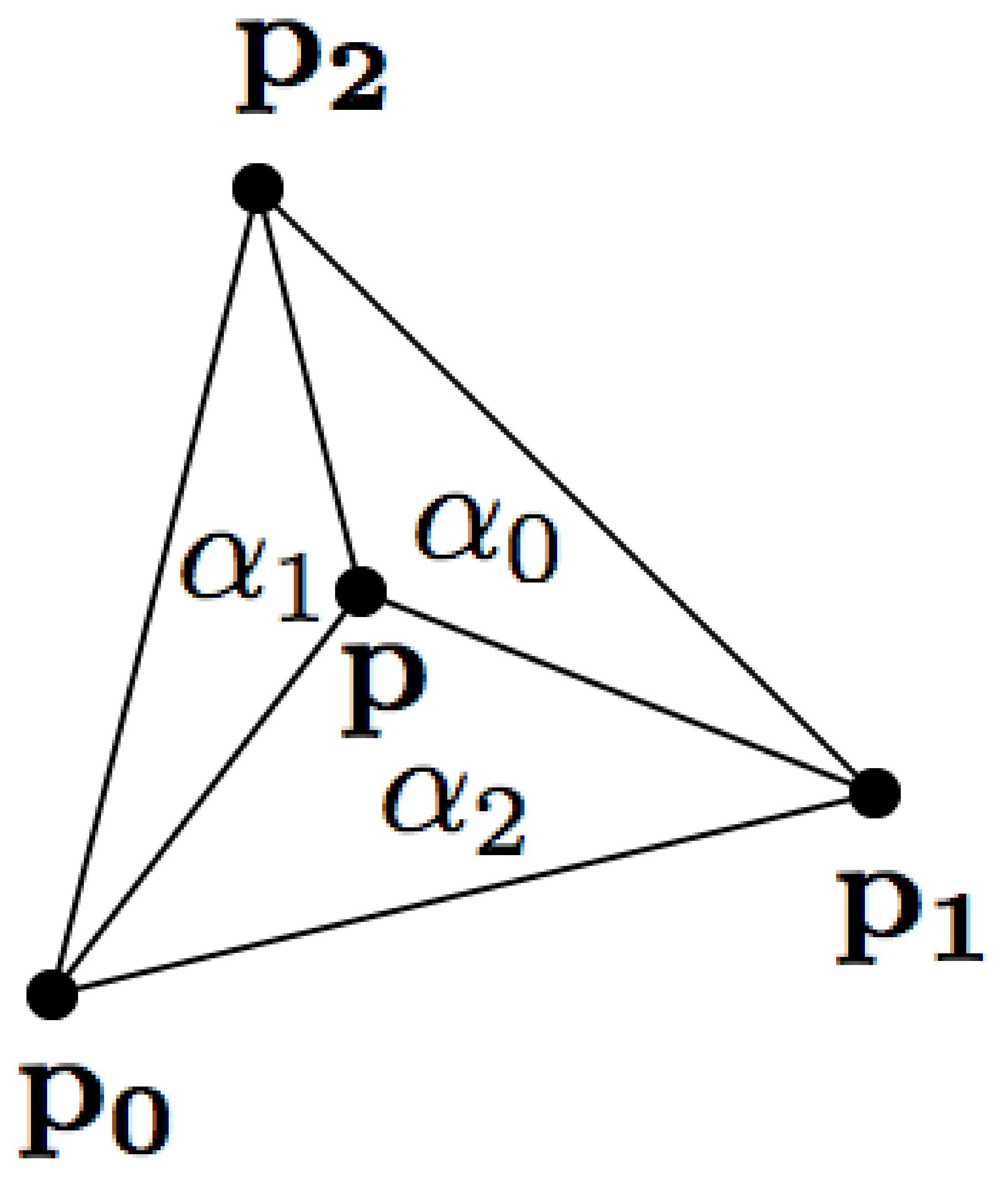
Algebraic Approach to Calculate Barycentric Weights

$$\alpha_0 \mathbf{p}_0 + \alpha_1 \mathbf{p}_1 + \alpha_2 \mathbf{p}_2 = \mathbf{p}$$

$$\alpha_0 \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \alpha_1 \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + (1 - \alpha_0 - \alpha_1) \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$$

Solve a 2x2 matrix equation:

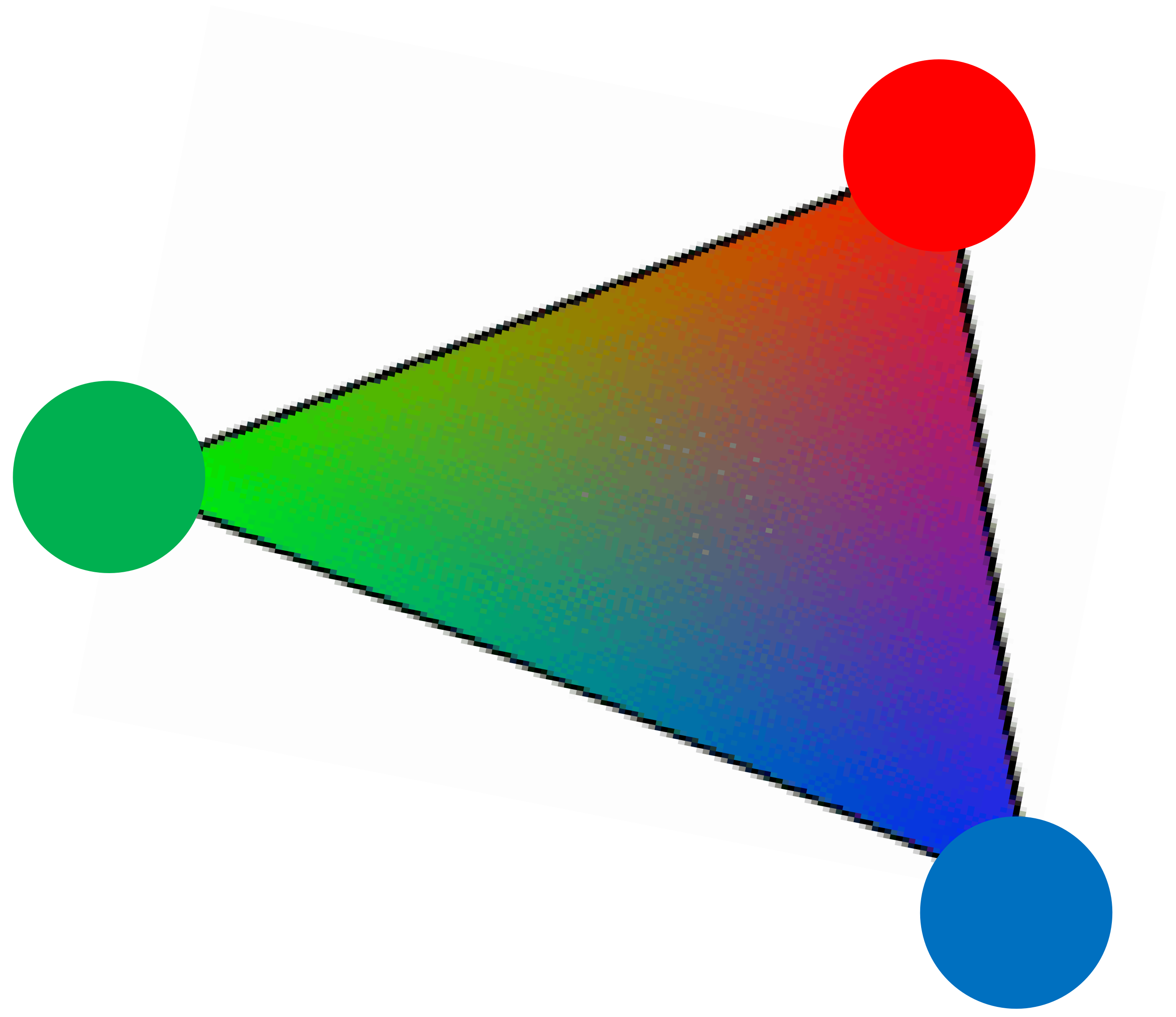
$$\begin{pmatrix} x_0 - x_2 & x_1 - x_2 \\ y_0 - y_2 & y_1 - y_2 \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$



Key Idea: If we interpolate the three vertices of a triangle using barycentric interpolation, we should be able to obtain the position of the current interpolation point.

Interpolating Vertex Data

- Interpolate colors
  - $R = \alpha_0 R_0 + \alpha_1 R_1 + \alpha_2 R_2$
  - $G = \alpha_0 G_0 + \alpha_1 G_1 + \alpha_2 G_2$
  - $B = \alpha_0 B_0 + \alpha_1 B_1 + \alpha_2 B_2$
- Interpolate normal vectors
  - $N = \alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2$
- Interpolate texture coordinates $u = \alpha_0 u_0 + \alpha_1 u_1 + \alpha_2 u_2$
  - $v = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2$



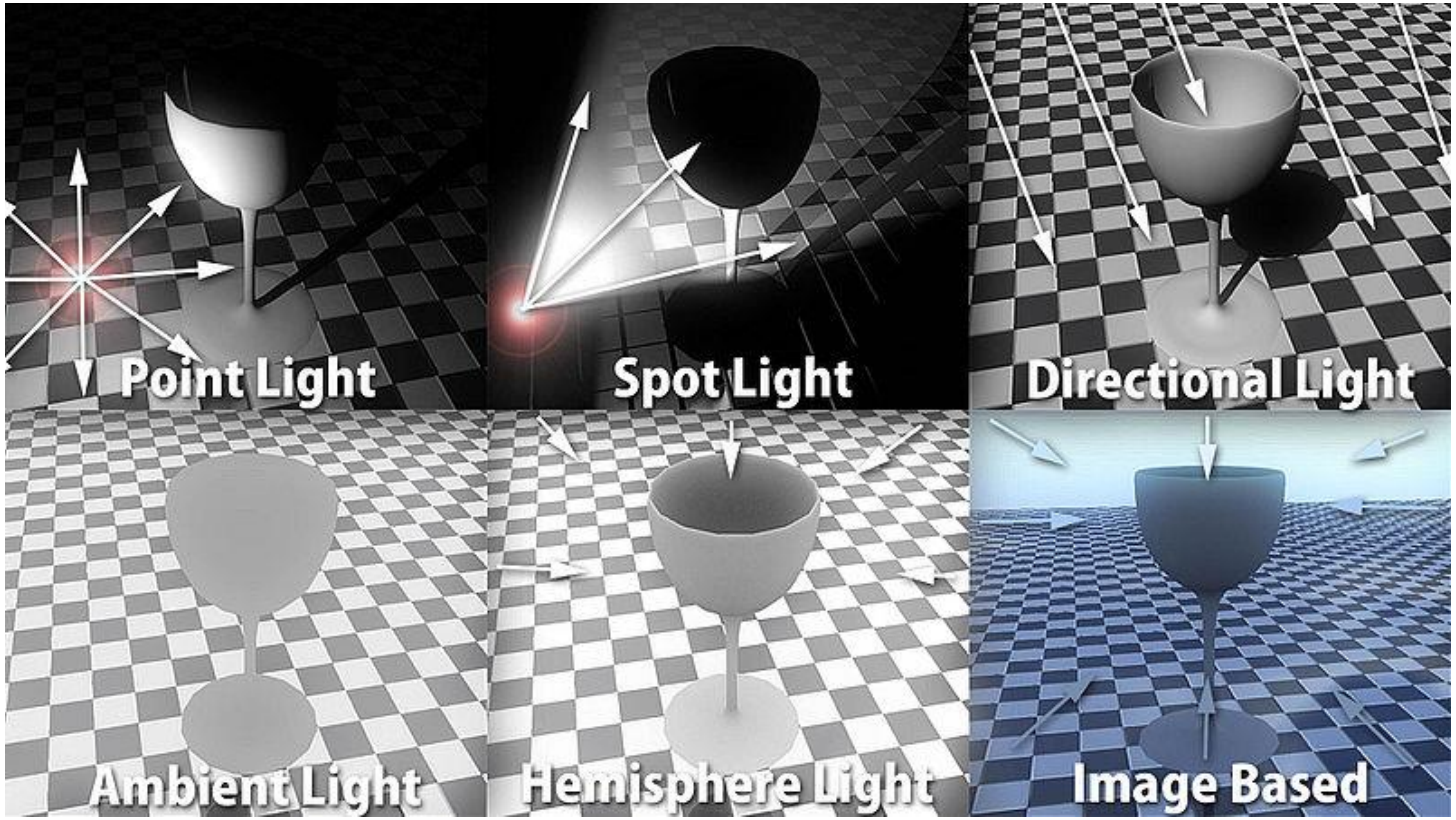
This step happens in the rendering pipeline between the vertex shader stage and the fragment shader stage --- but it's not programmable!

- We don't need to implement our barycentric interpolation, all the vertex->fragment interpolation within each triangle happen automatically.

Lights

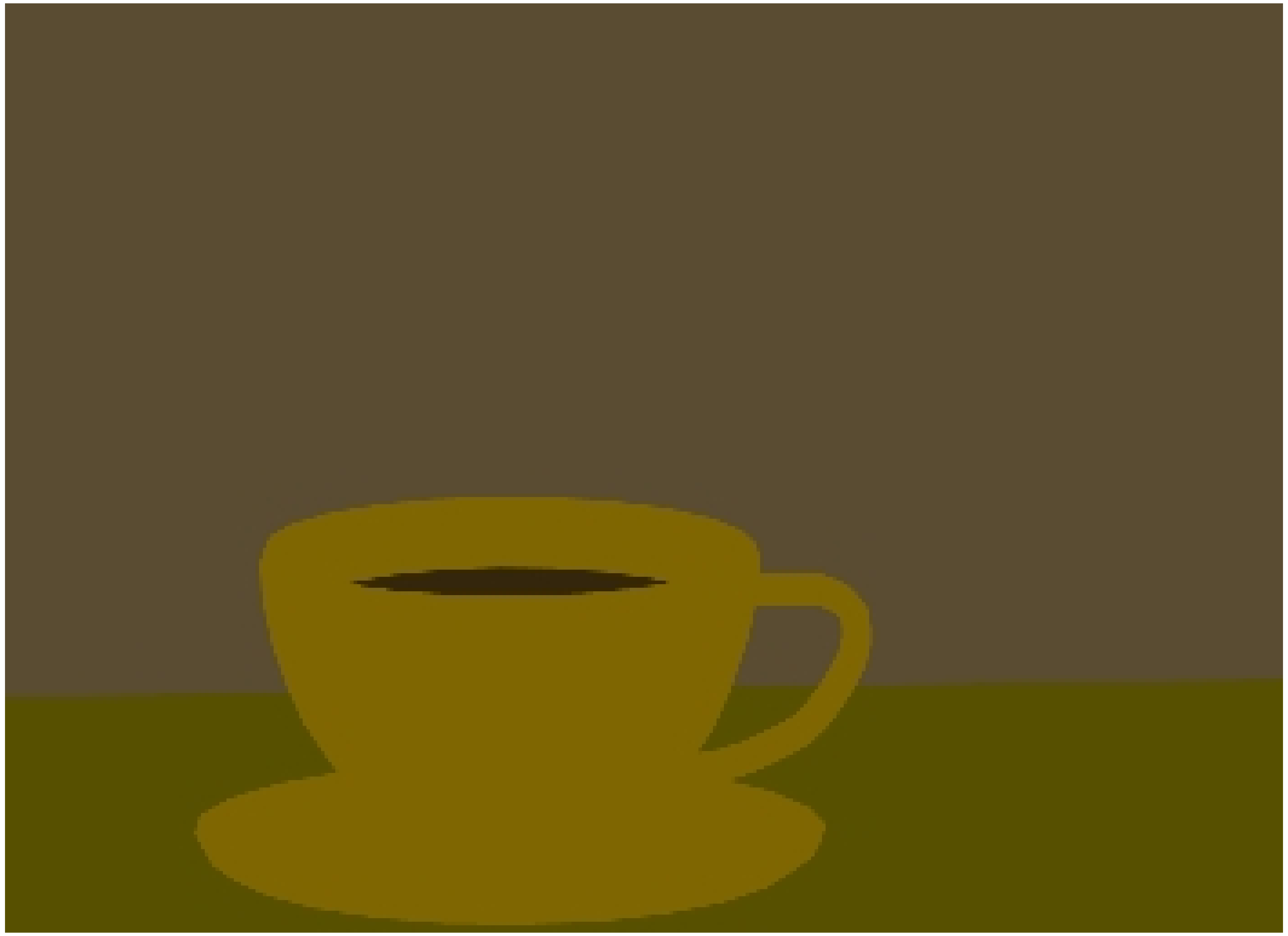
Different Types of Light

- Ambient light
- Directional light
- Point light



Ambient Lights

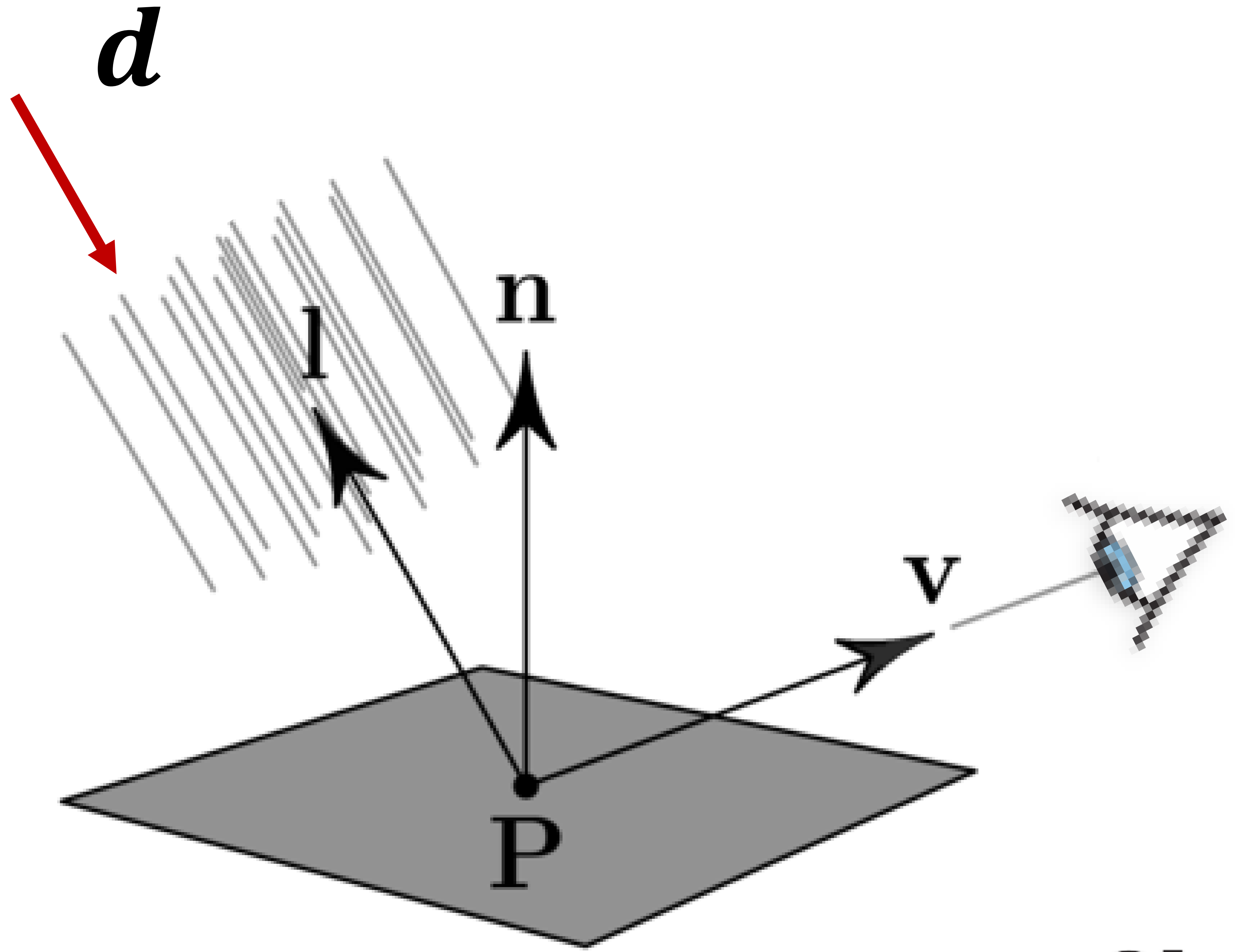
- Ambient lighting models a constant illumination independent of the incident light angle
- Useful for adding some light in the shadowed regions that would otherwise potentially be completely black



Directional lights

- Light is emitted from infinity in **one direction** $\vec{d}$
- Simulate distance lighting, e.g. sun, same at all surface points $\vec{p}$
- Light direction: $\vec{l} = -\vec{d}$

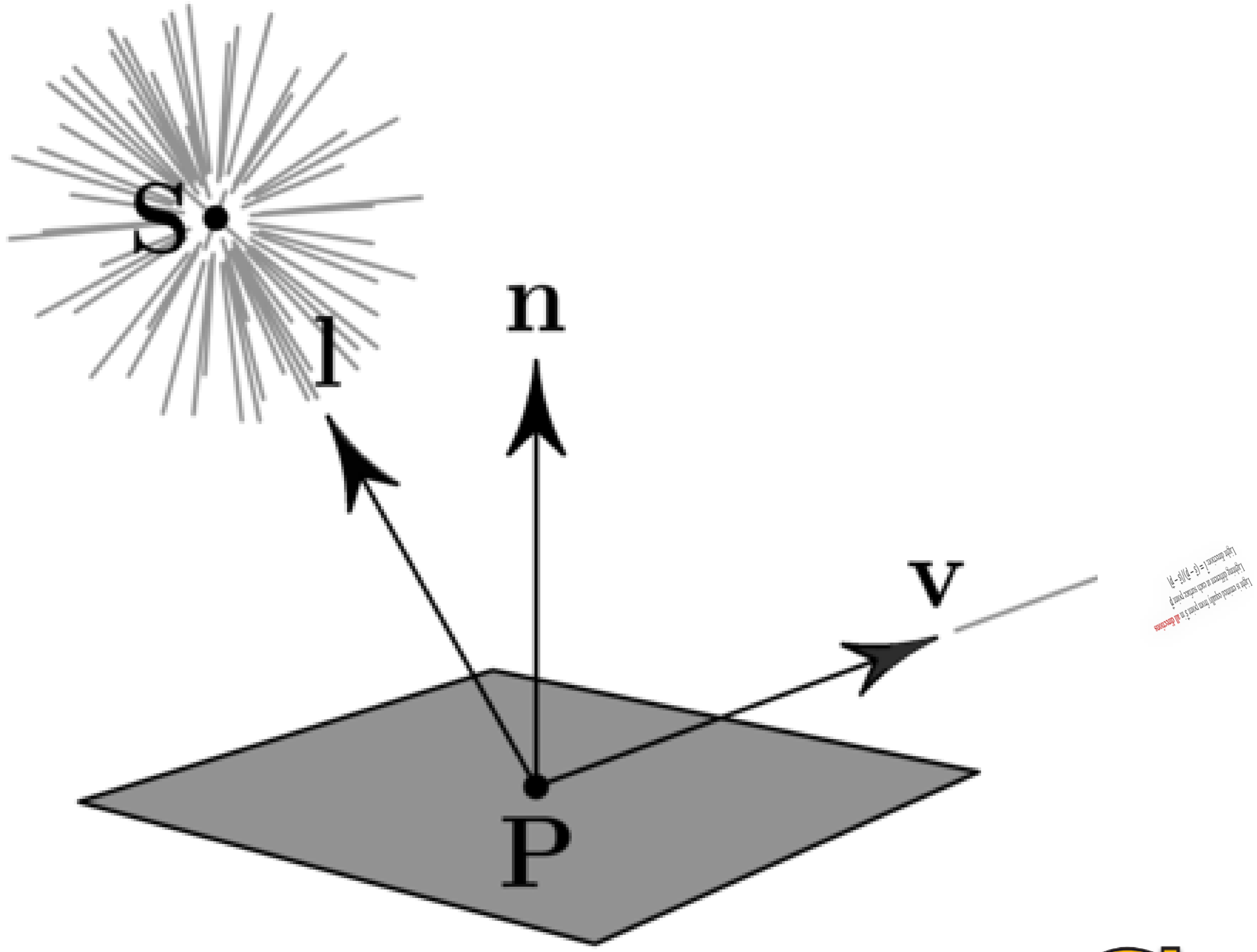
Note the negative sign here: we typically use vector $\vec{l}$ to denote the vector pointing from a surface point to the light source



Point lights

- Light is emitted equally from point $\vec{s}$ in **all directions**
- Lighting different at each surface point $\vec{p}$
- Light direction: $\vec{l} = (\vec{s} - \vec{p})/|\vec{s} - \vec{p}|$

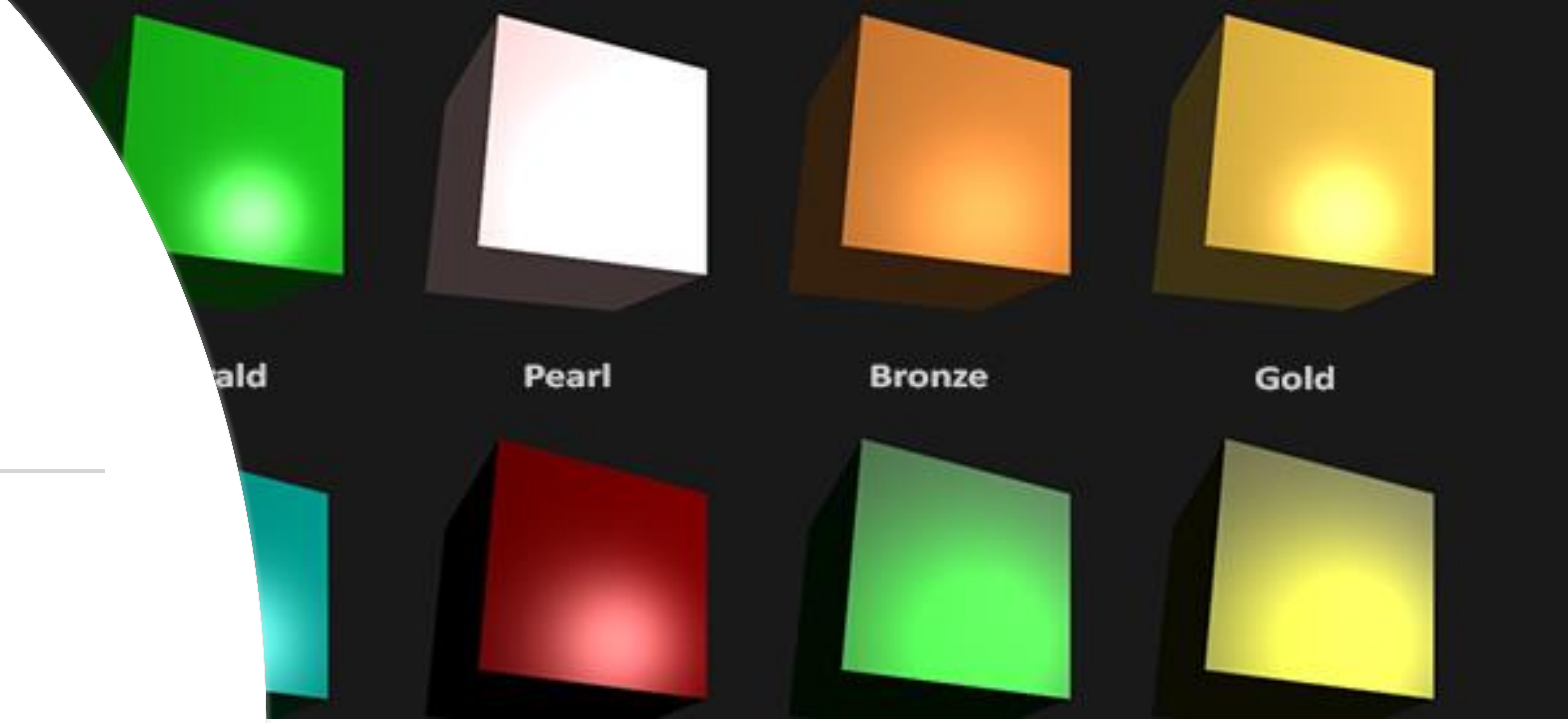
Again, here $\vec{l}$ is defined as a vector pointing from a surface point to the light source

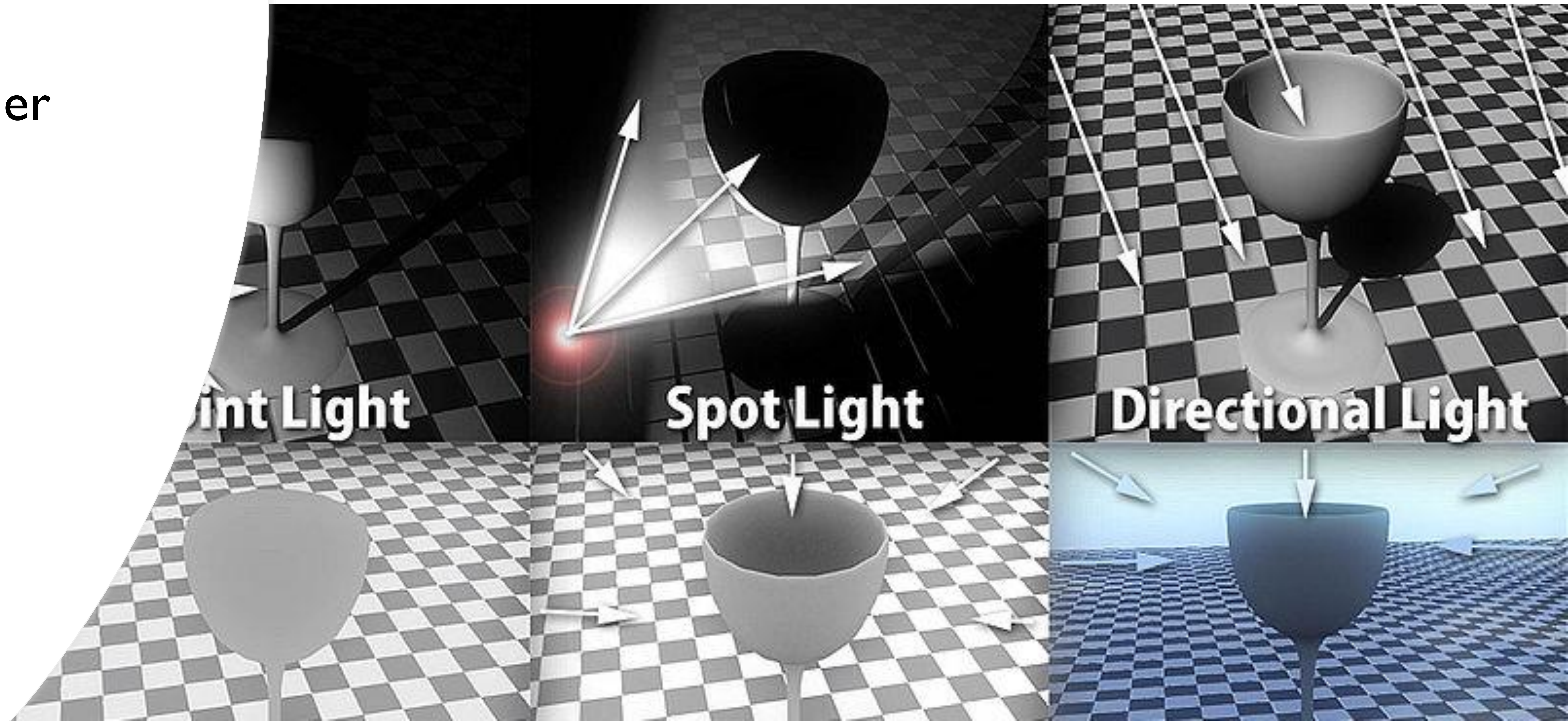


Shading Model

What is a shading model

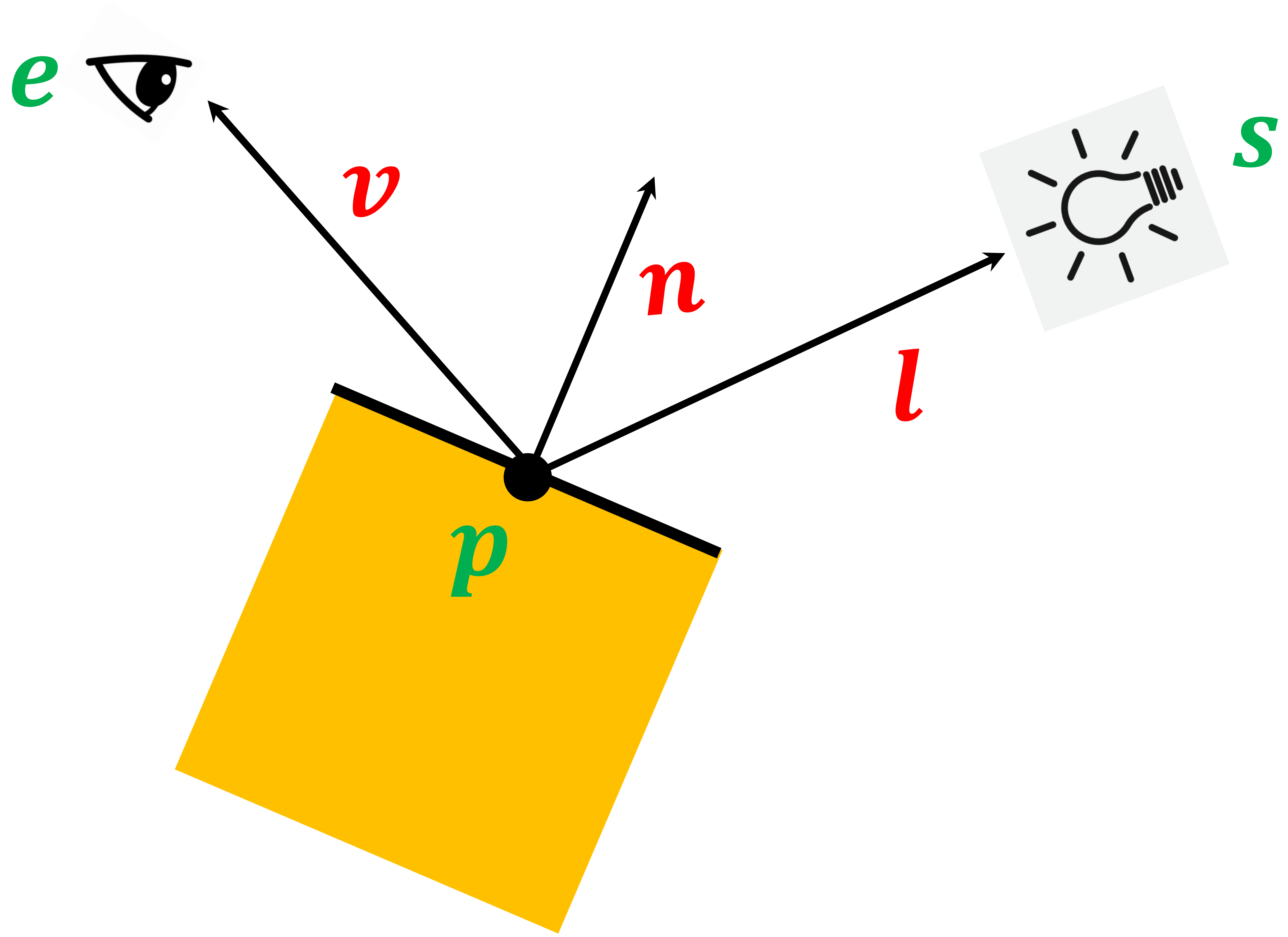
- A shading model simulate the way light interacts with object surfaces
  - They define how lights and shades appear on the surface of the object
- A few key properties we need to consider when devising a shading model:
  - Type of lights
  - Surface material properties
  - Light-material interaction
  - View dependency





Mathematically, how do we describe a shading model?

- For a surface point, we typically consider the relation among **eye**, **light source**, and **surface normal** to calculate a light model



Let's start with a simple **ambient model**

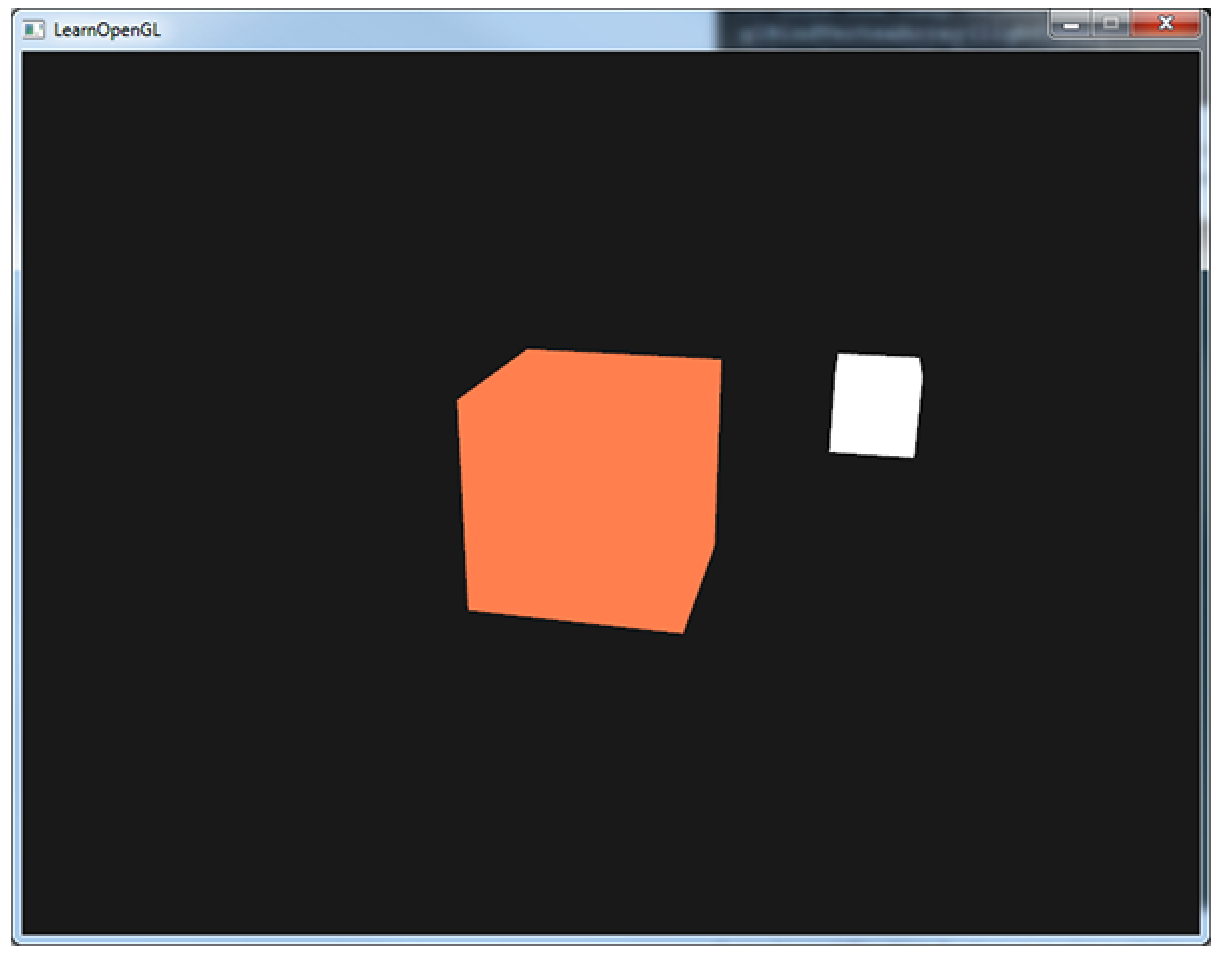
Why do we distinguish k_a and I_a?
- We use k_a for material property and I_a for light property.
- The final color is the **component-wise multiplication** of light color and material color.

light source ambient intensity

$$L_a = k_a \boldsymbol{I}_a$$

material ambient coefficient

ambient color



Each color component is treated in separate!

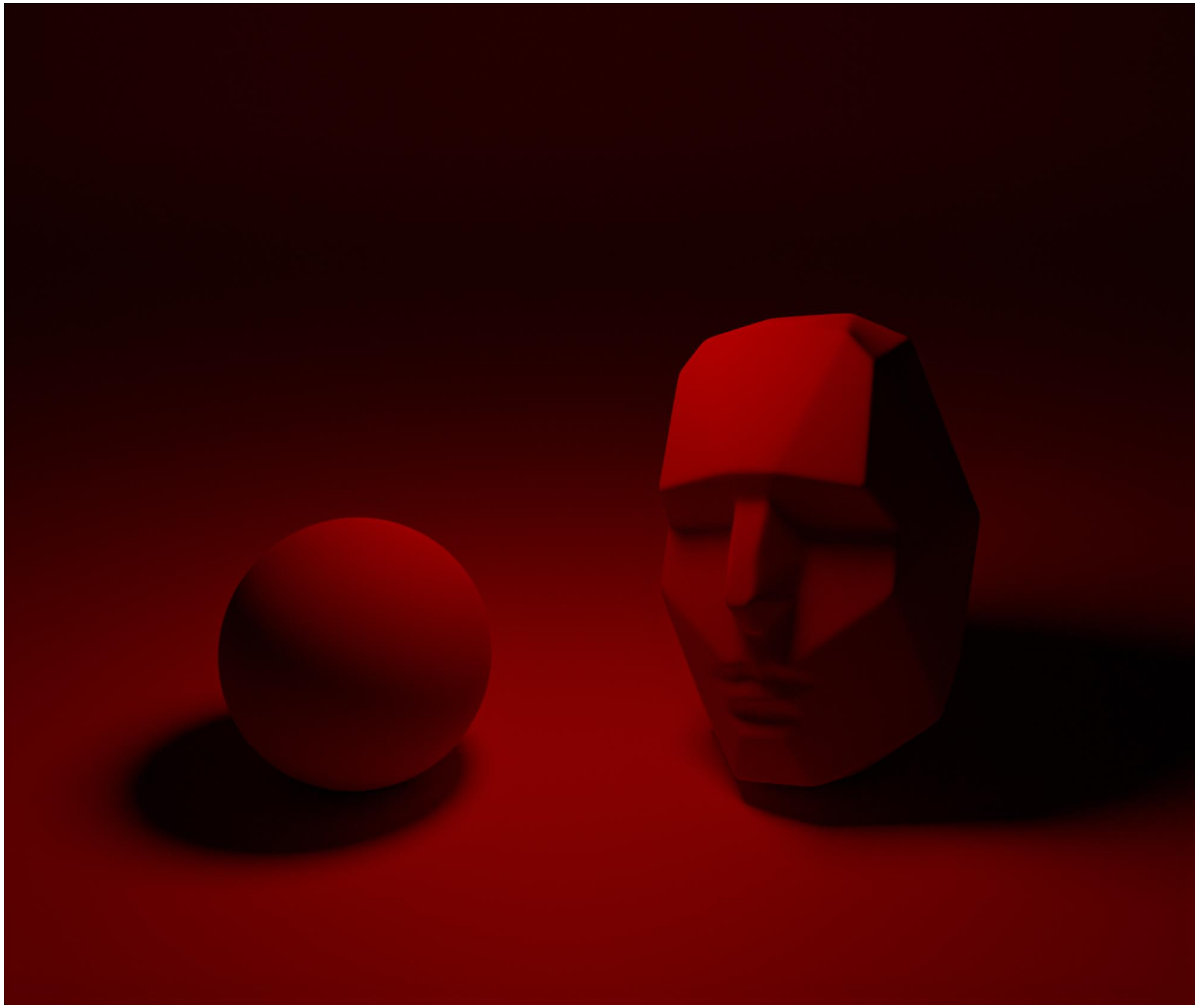
Visual Intuition

- You see some object in red, it could be because

(1) the object's surface material is red, or

(2) the object is casted by some red light

It is difficult to distinguish these two cases visually



Component-wise Multiplication

- Both k_a and $\boldsymbol{I}_a$ are vec3
- We conduct multiplication for each component, i.e.:

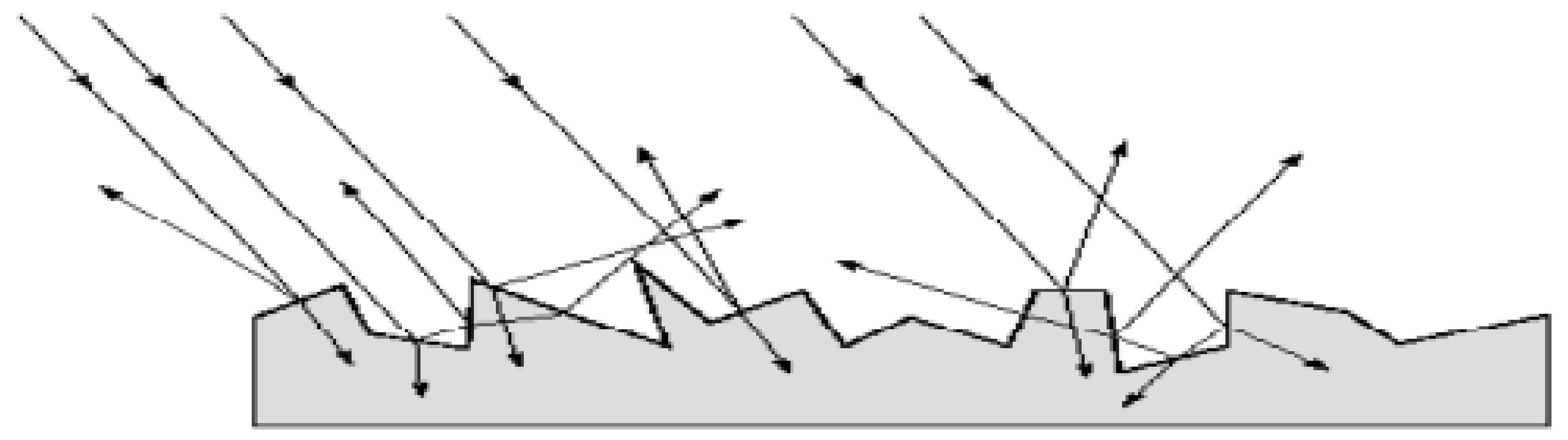
$$L_a = k_a \boldsymbol{I}_a \quad \Rightarrow \quad L_a = (k_{ar} * I_{ar}, k_{ag} * I_{ag}, k_{ab} * I_{ab})$$

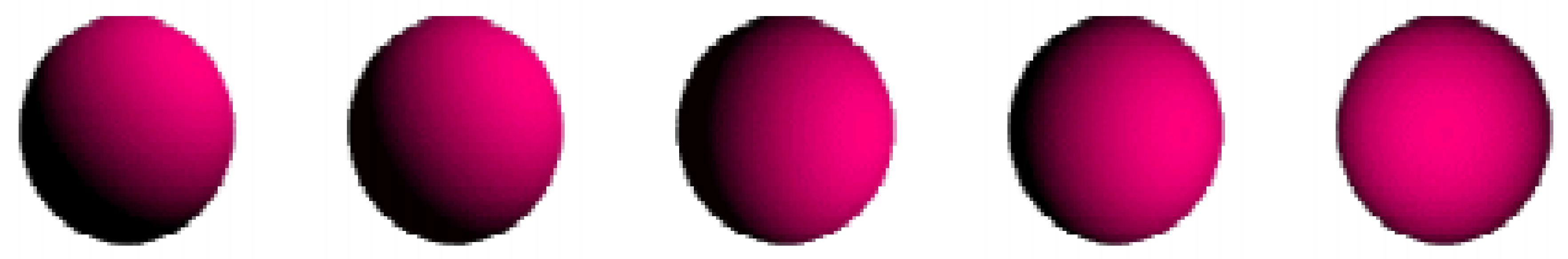
- Component-wise multiplication is neither dot nor cross product
- It can be simply implemented with * operator in GLSL, e.g.,

```
vec3 a=vec3(1., 0.5, 0.3);
vec3 b=vec3(0.2, 0.4, 0.2);
vec3 c=a*b;        ////c=vec3(0.2, 0.2, 0.06);
```

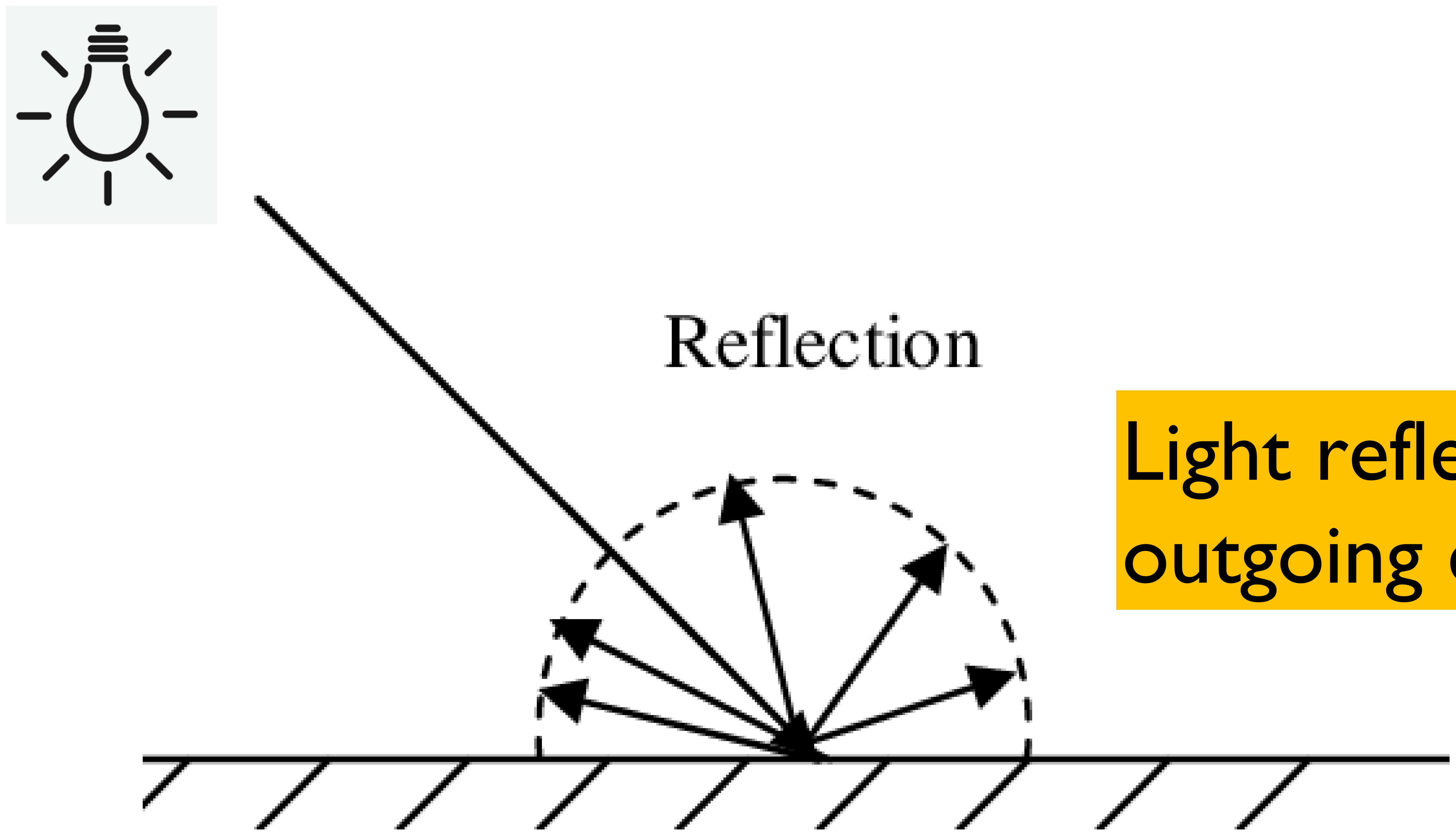
Next, let's make it a bit more complicated by adding a **Diffuse Model**

- The diffuse model models a very rough surface with lots of microfacets that randomly reflect incoming light in **every** outgoing direction
- The shading does not depend on the position of the camera/viewer
- The shading still depends on the position of the light
  - Light sources on the top or on the side with produce different colors on the object's surface
- This is a good model for ideally diffuse/dull/matte surfaces, such as chalk





Let’s illustrate a diffusive surface with simple geometry

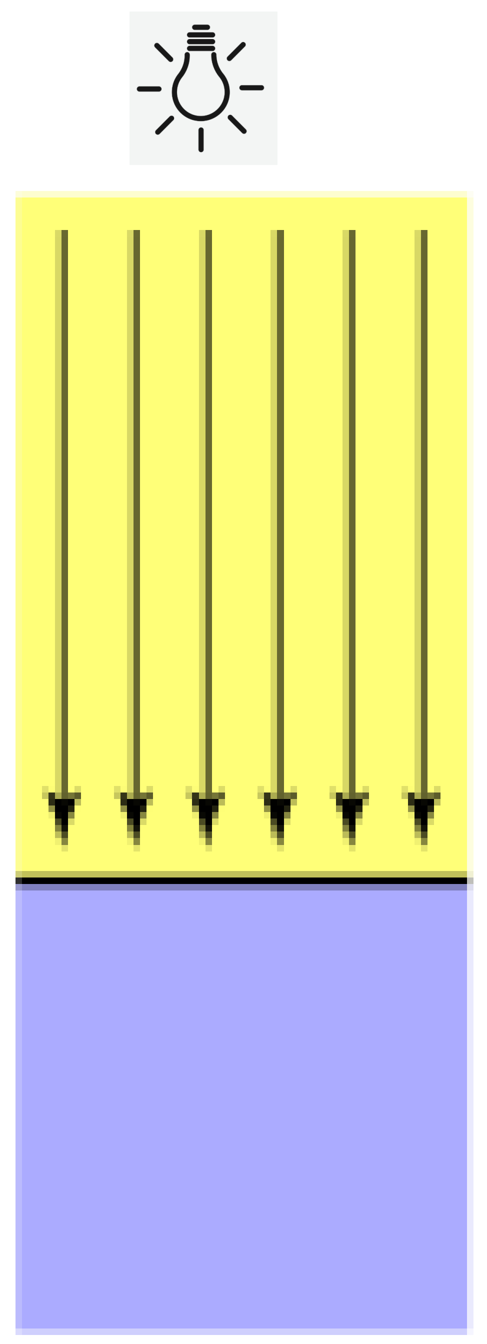


Light reflects in every outgoing direction

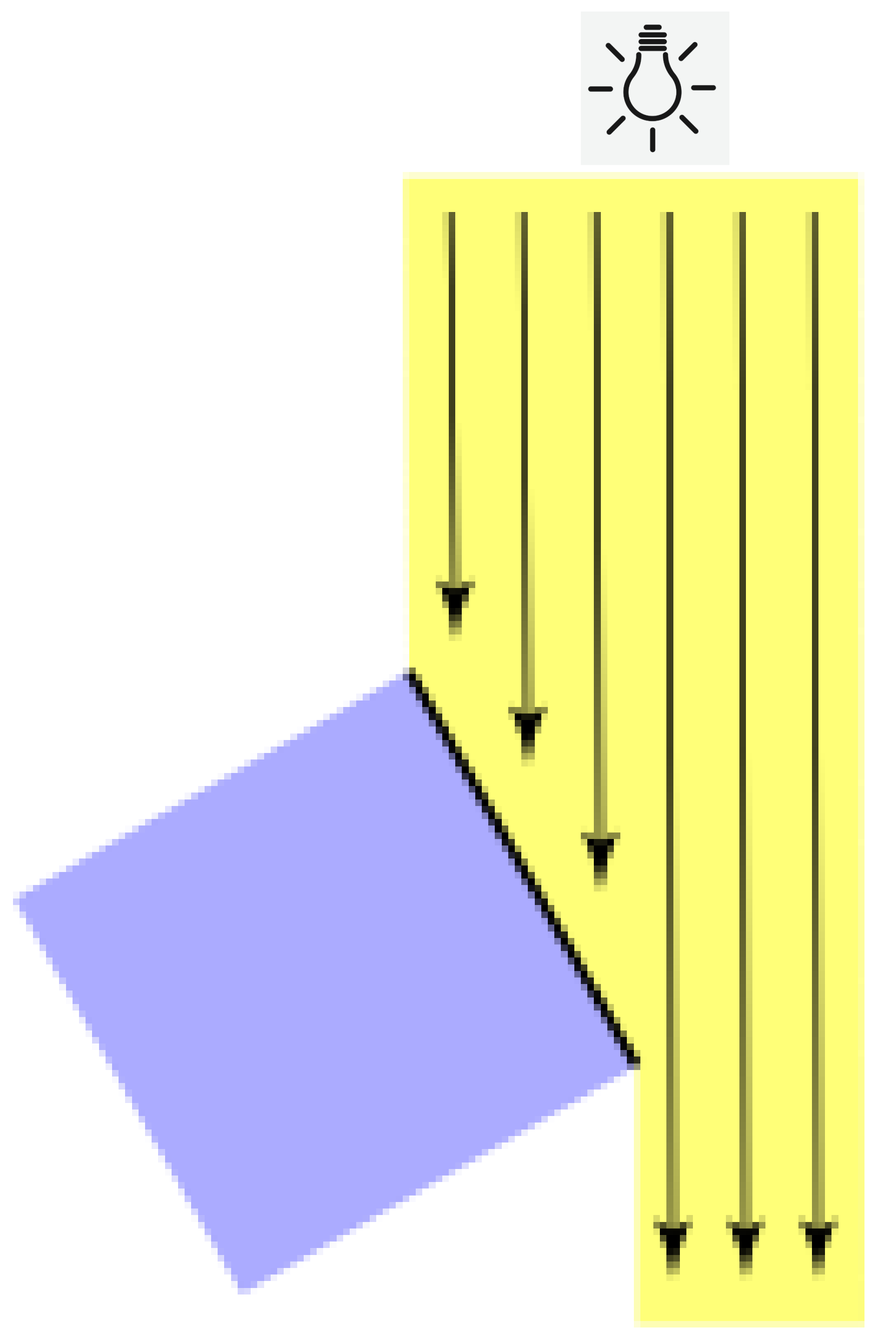
Also called ideal **Lambertian** reflection

Lambert's Cosine Law

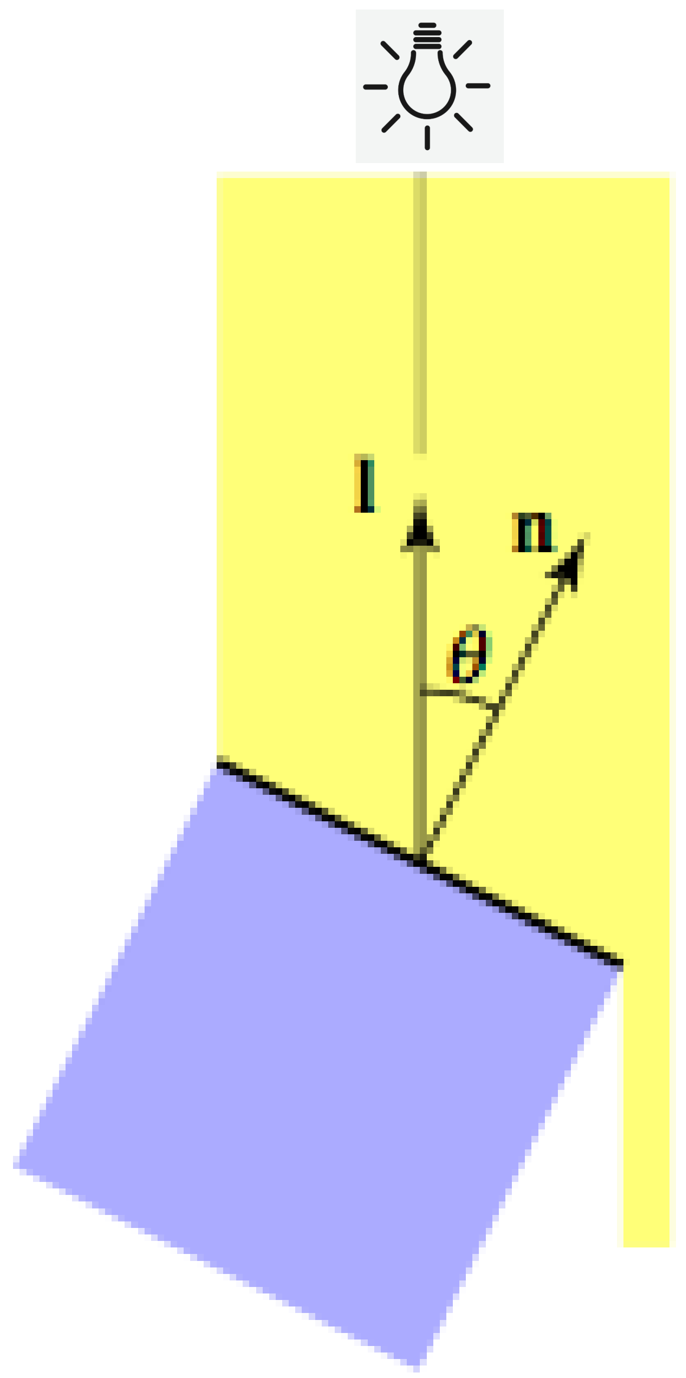
- Light source position matters!



Top face of cube receives a certain amount of light



Top face of 60° rotated cube intercepts half the light

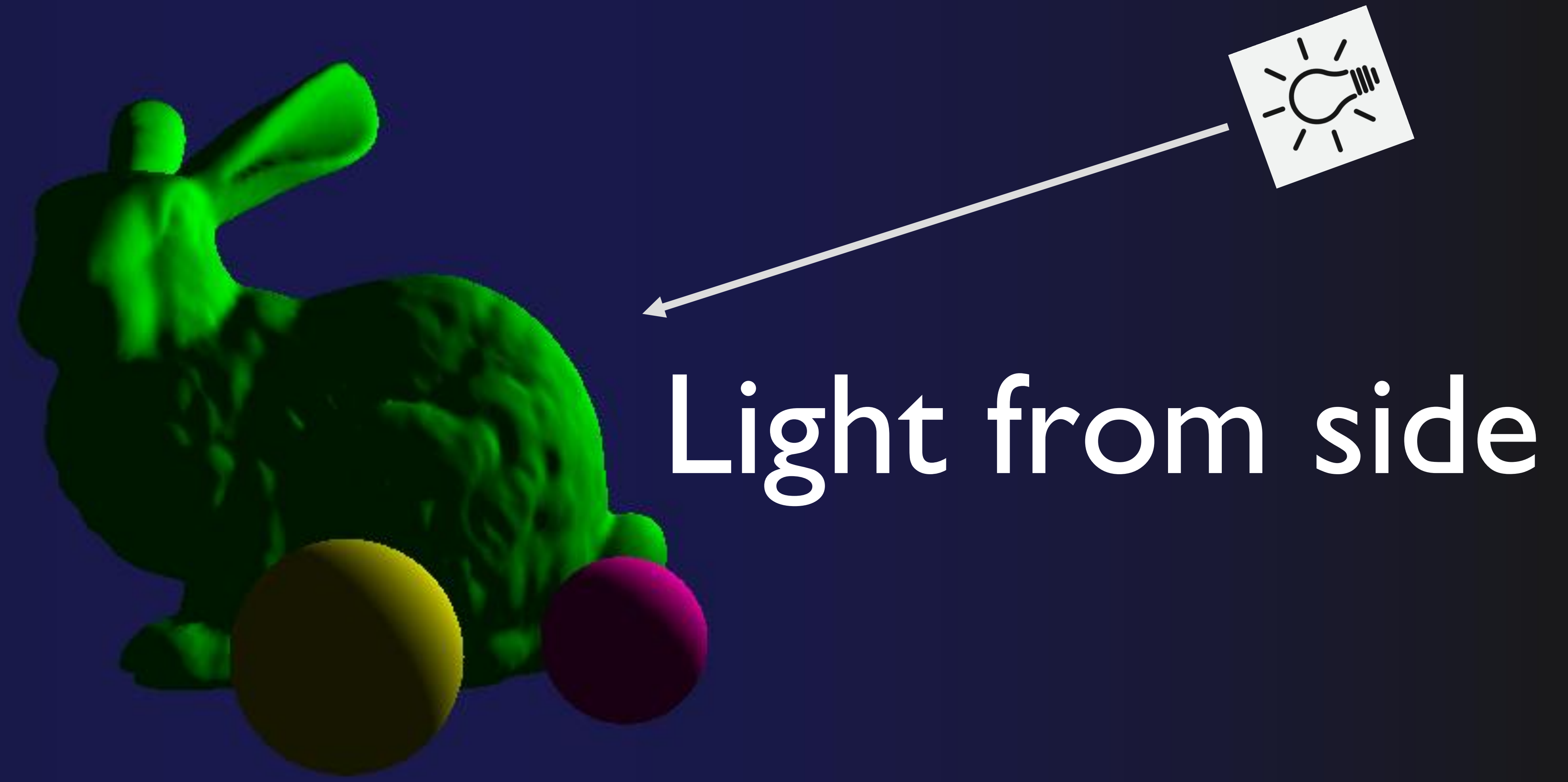


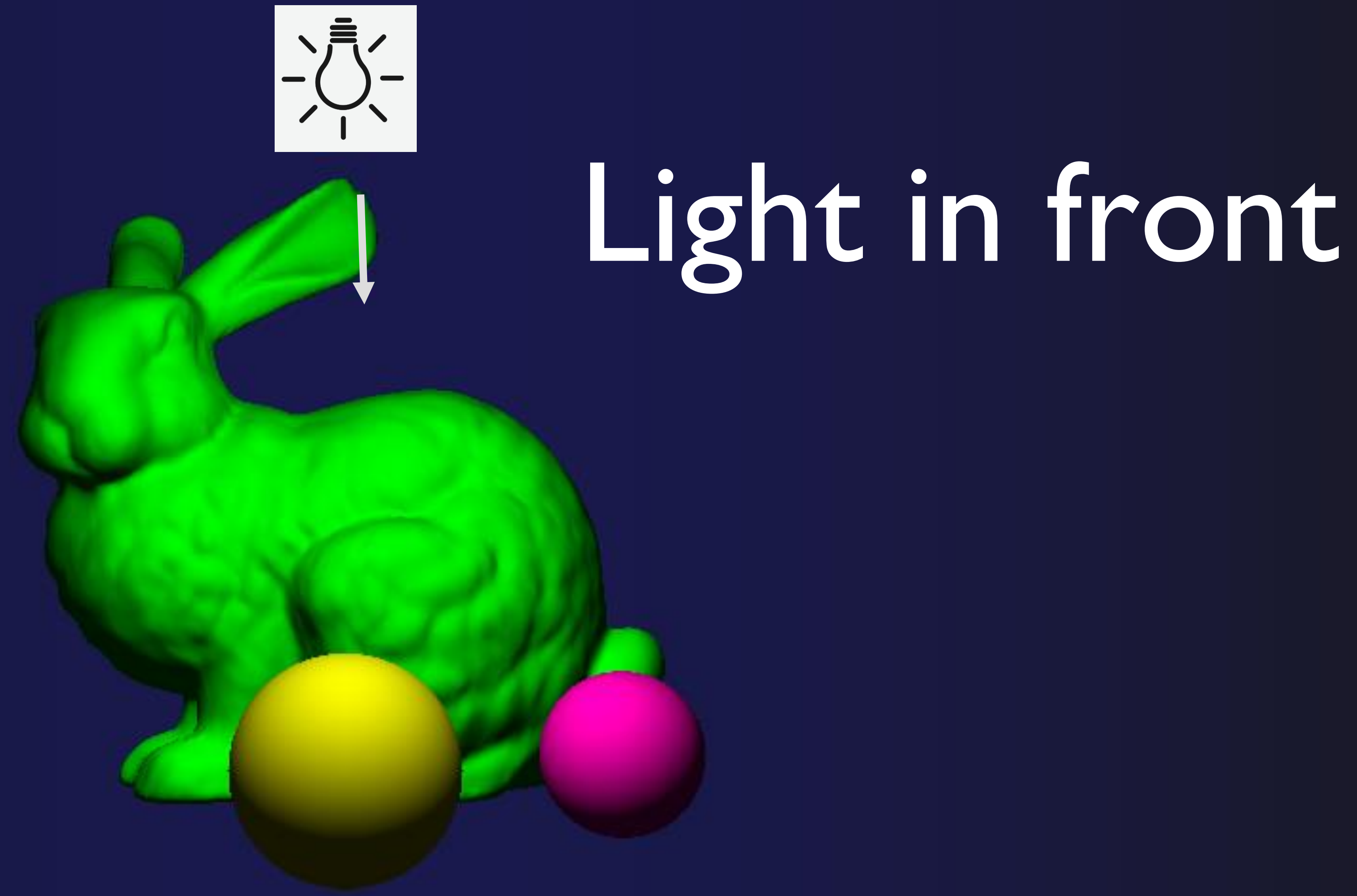
In general, light per unit area is proportional to

$$cos\theta = \vec{l} \cdot \vec{n}$$

Visual Intuition

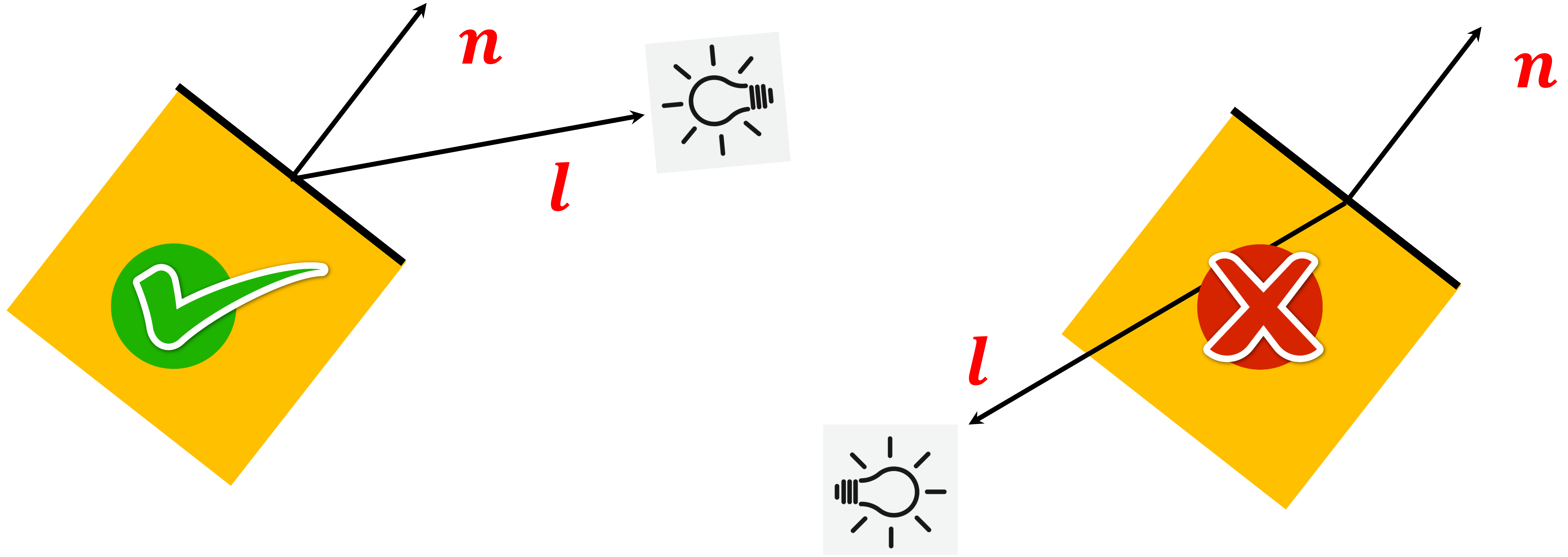
- Objects will appear brighter when illuminated by a light source directly, and darker when the light hit them at an angle.





A light source doesn't produce color if it is on the back

- Light source position matters!
    - Needs to check if a light source is on the backside of a face

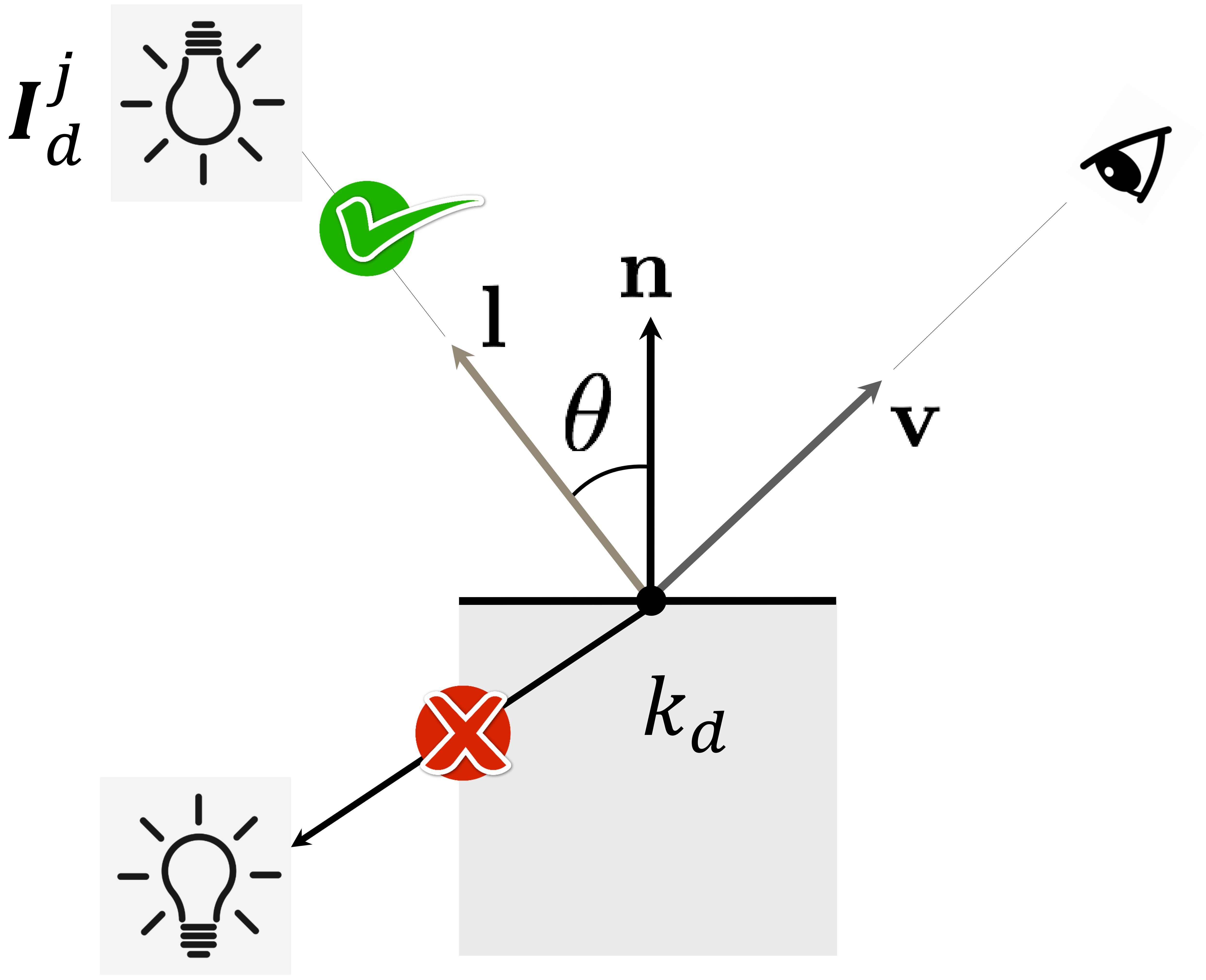


We can express this check together with Lambert's cosine law as

$$\max(0, \cos\theta) = \max(0, \vec{l} \cdot \vec{n})$$

Put everything together in one expression

- Diffusion (Lambertian) shading model:



$$L_d = k_d I_d \max\left(0, \vec{l} \cdot \vec{n}\right)$$

light source diffuse intensity

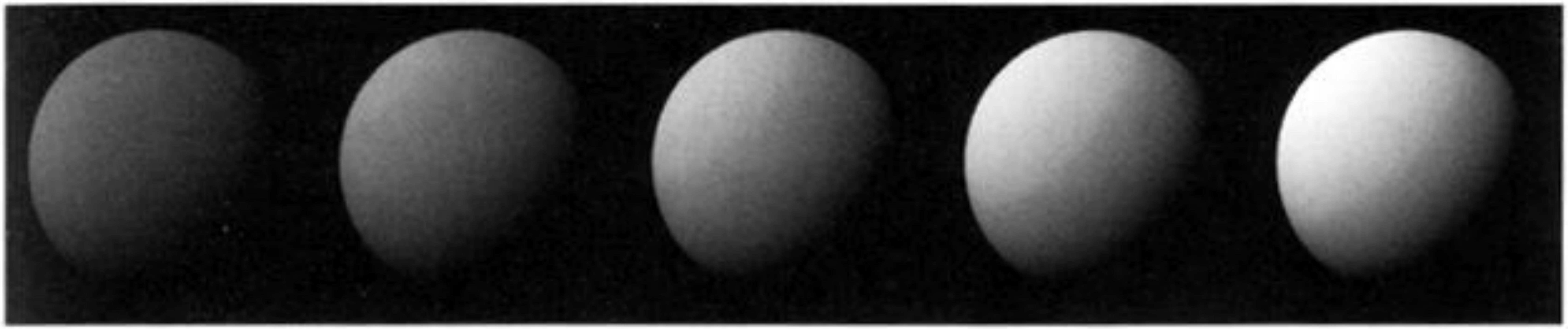
check if light is on the back and calculate its contribution

material diffuse coefficient

diffuse color

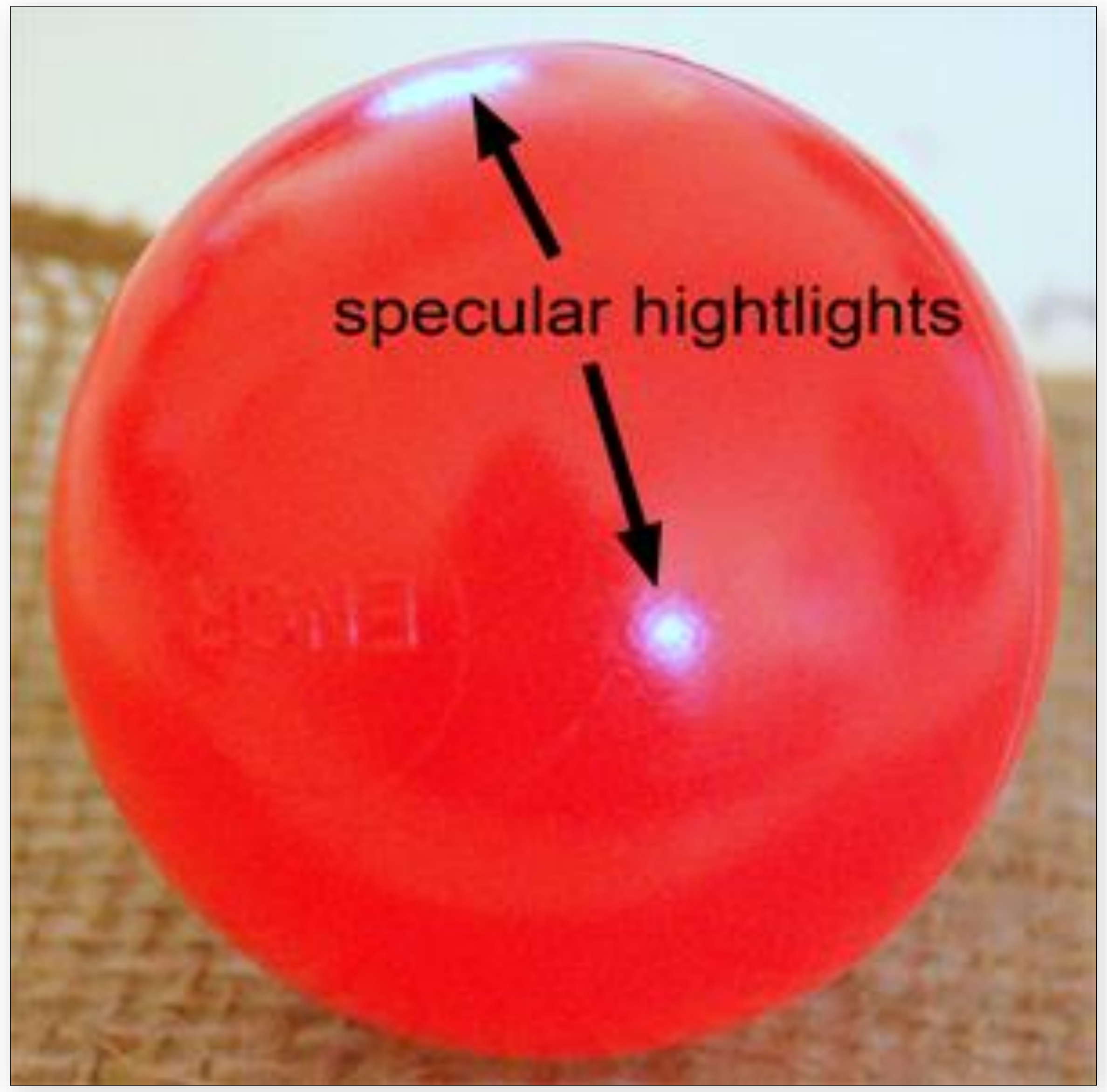
Key Takeaways for Lambertian Shading

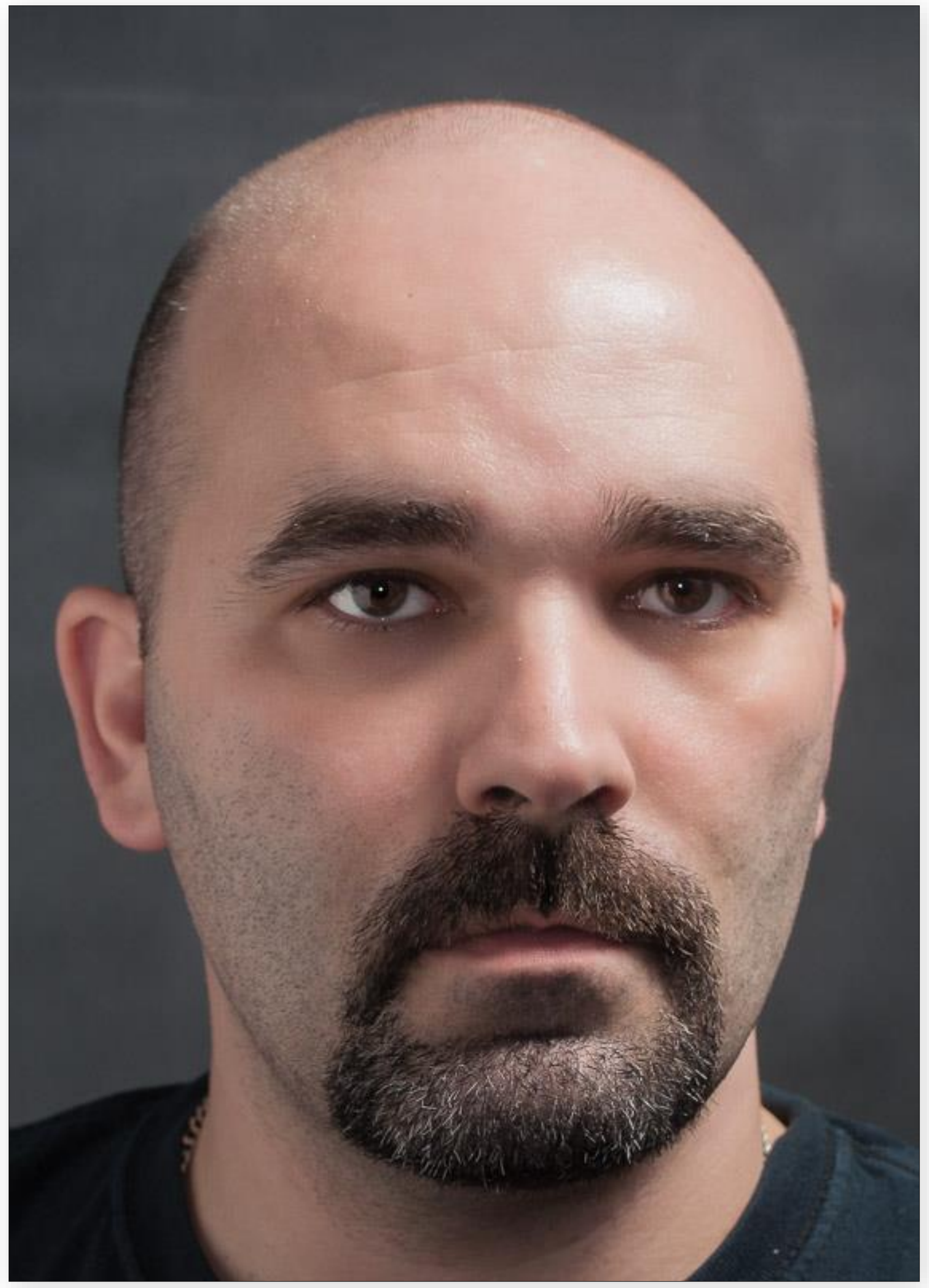
- Light source location matters!
- Observer location does not matter
- Need to check if the light source is behind the surface (by checking the dot product of surface normal and light direction)



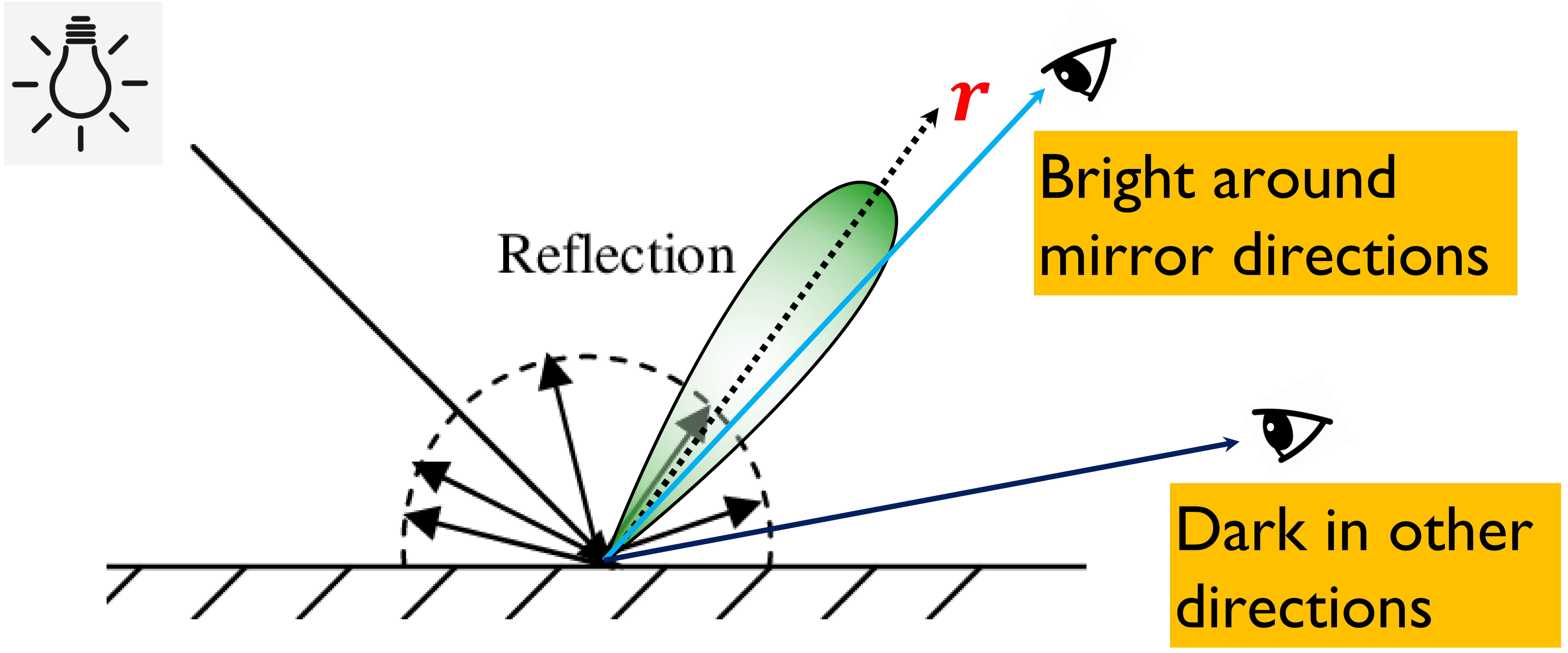
Our Next Goal: **Specular Highlights**

- These are effectively just "blurry" reflections of the light source



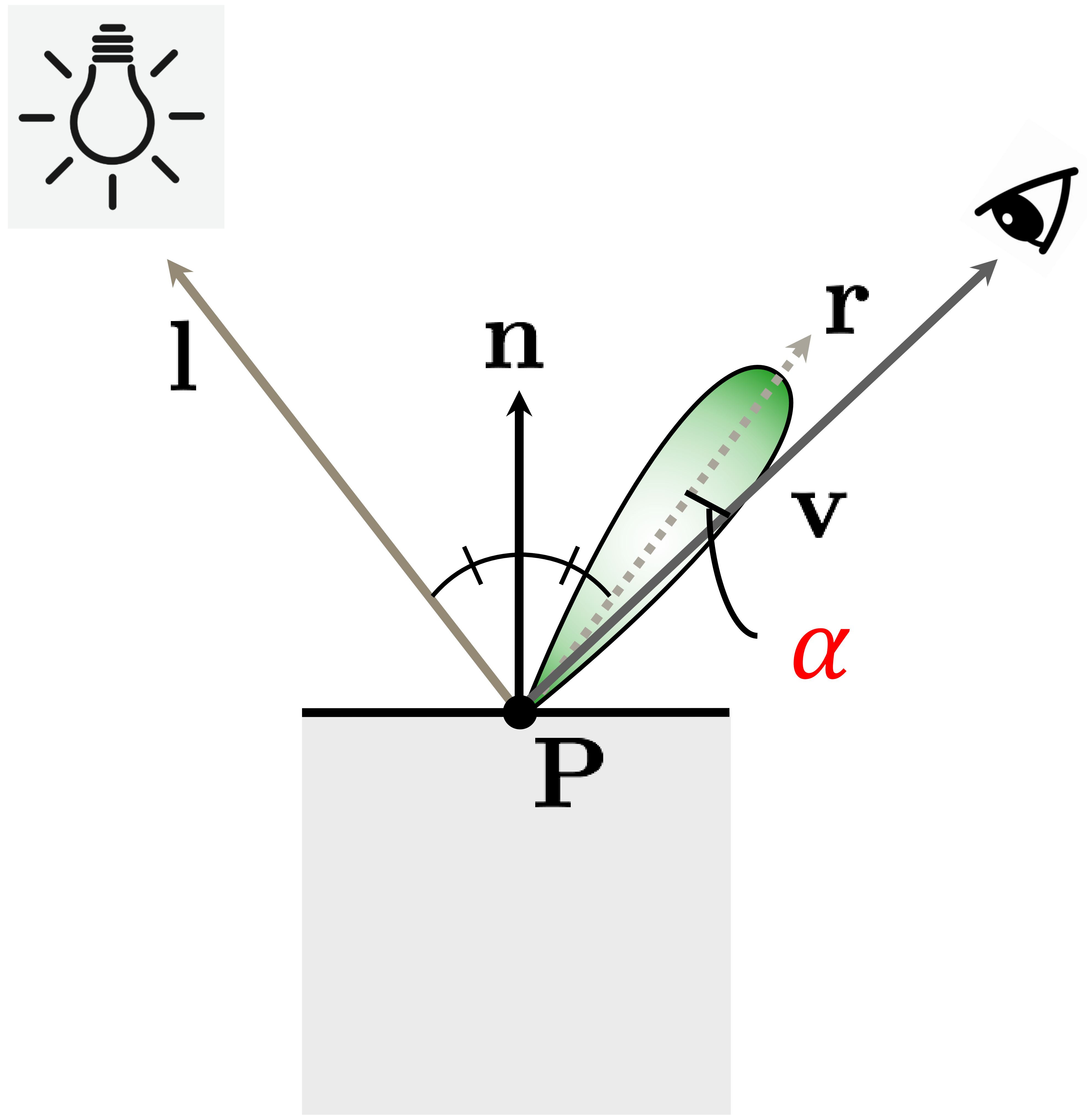


Let's illustrate a specular surface with simple geometry



Light intensity depends on view direction:
brighter around the mirror direction and darker elsewhere

Specular Highlights (Phong)



The larger the index value, the smaller the highlight region

light specular contribution according to the view

Power index controlling the size of the highlight region

light source specular intensity

shininess

$$L_s = k_s \boldsymbol{I}_s \ \max(0, \vec{v} \cdot \vec{r} \quad)^p$$

specular color

material specular coefficient

$cos\alpha$

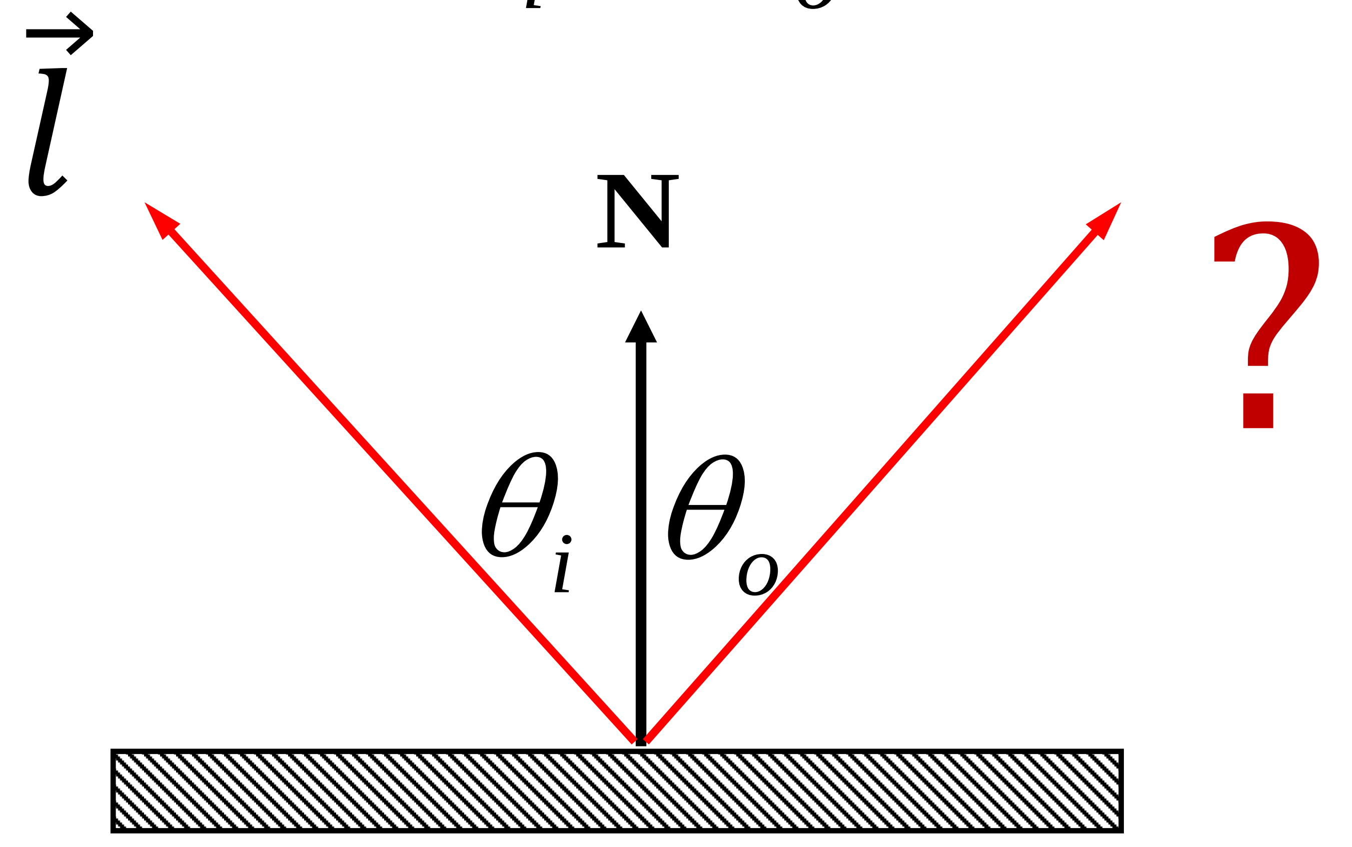
Indicates how close the mirror direction and view direction are together

How to calculate the mirror reflection vector?

Attention: The definition of I is opposite to our convention here (**you should use -I as input**)

- Mirror - ideally smooth and specular surface
- Law of reflection

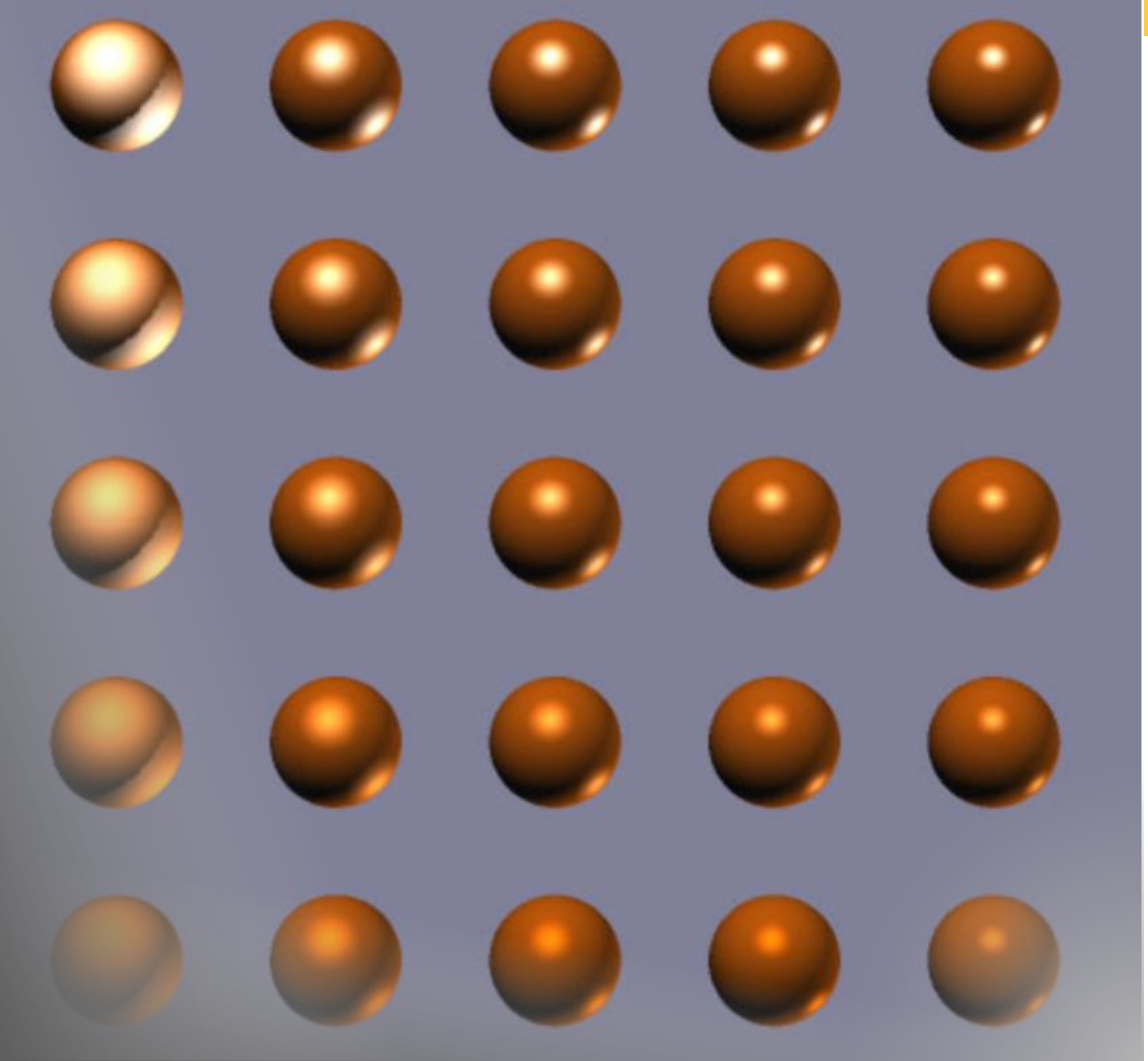
1 $\theta_i = \theta_o$



$$\vec{r} = -\vec{l} + 2\left(\vec{l} \cdot \vec{n}\right)\vec{n}$$

2 **vec3 reflect(vec3 I, vec3 N)**
For a given incident vector *I* and surface normal *N* **reflect** returns the reflection direction calculated as *I* - 2.0 * **dot**(*N*, *I*) * *N*.
N should be normalized in order to achieve the desired result.

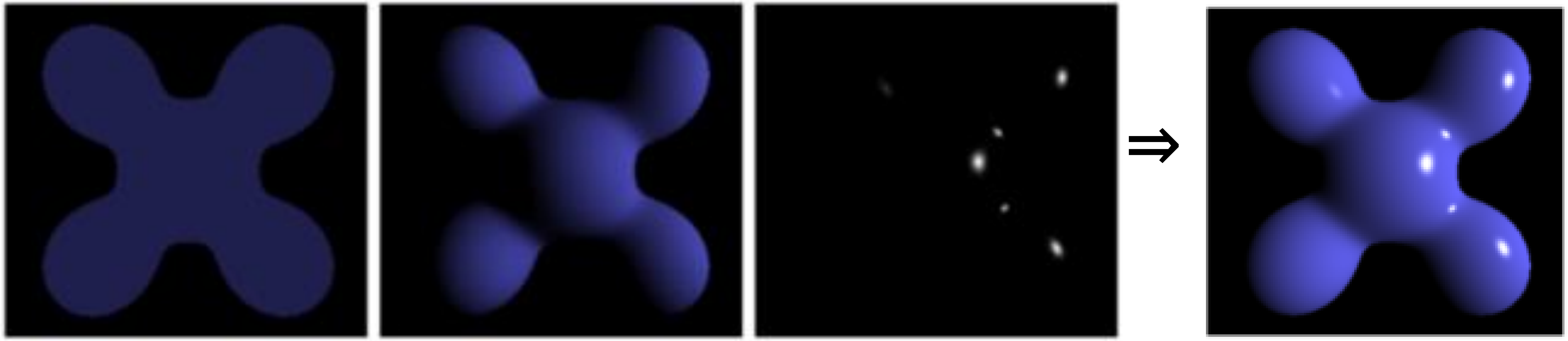
- Incident light direction $\vec{l}$, surface normal vector $\vec{n}$, and reflected light direction $\vec{r}$ are all coplanar
- In GLSL we implement reflection vector with built-in **reflect()**



Key Takeaways for Specular Shading

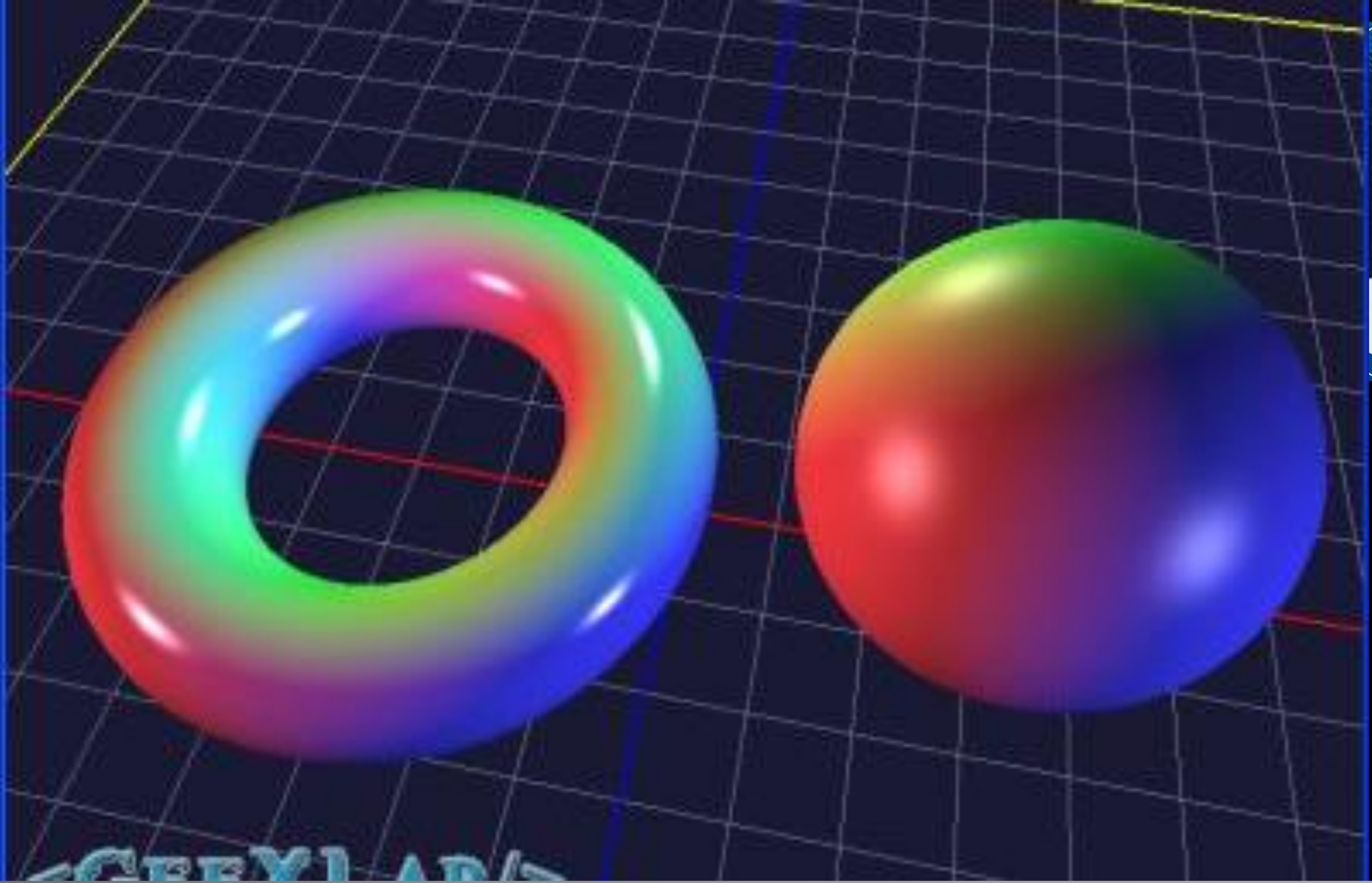
- Both light source location and viewer location matter!
- Observe brighter color when closer to the mirror direction
- The shininess power controls the size of the highlight region

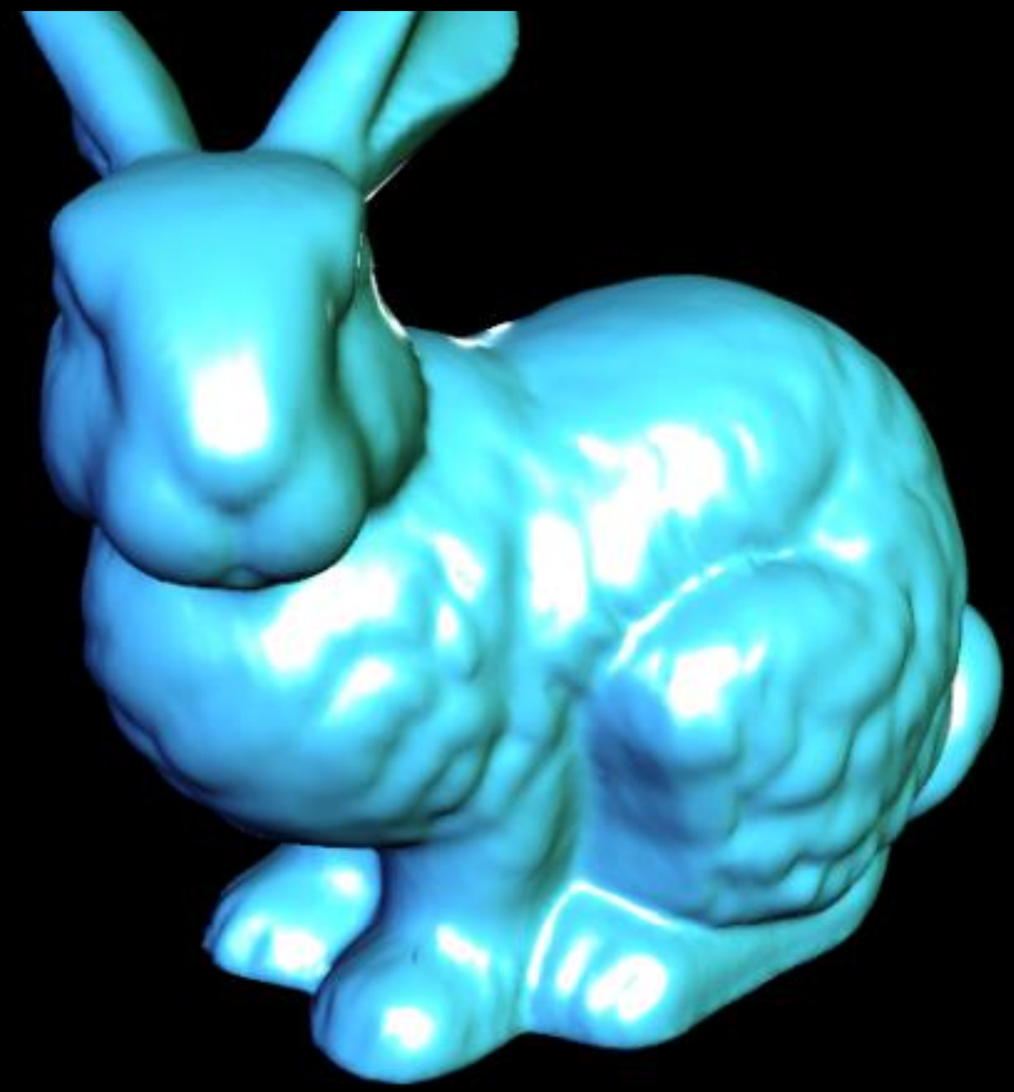
Putting Everything Together: Phong Shading!



Ambient **Diffuse** **Specular** **Final Image**

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} (\underbrace{k_a \boldsymbol{I}_a^j}_{\text{Ambient}} + \underbrace{k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right)}_{\text{Diffuse}} + \underbrace{k_s \boldsymbol{I}_s^j \max(0, \vec{v} \cdot \vec{r}^j)^p}_{\text{Specular}})$$

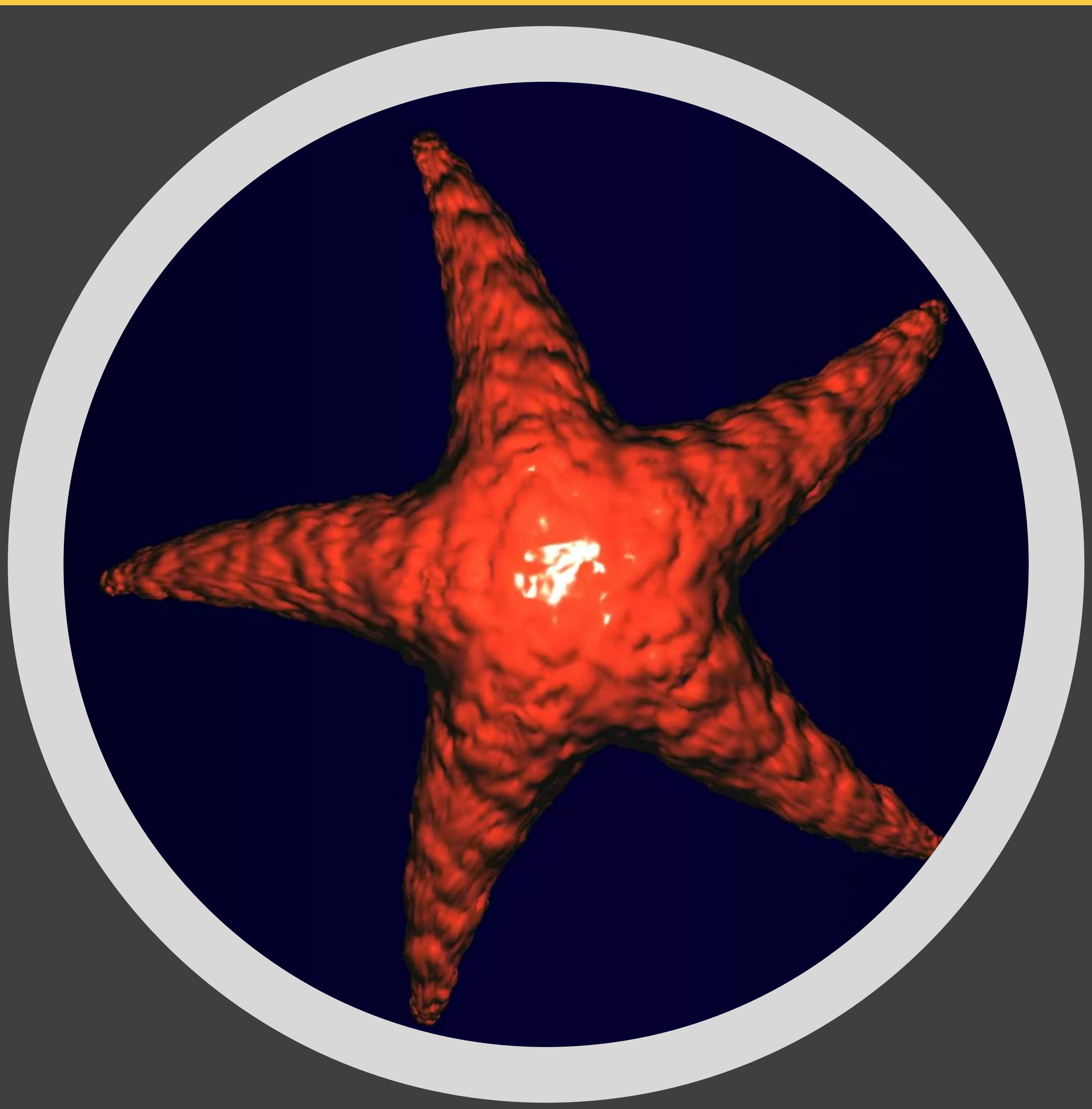


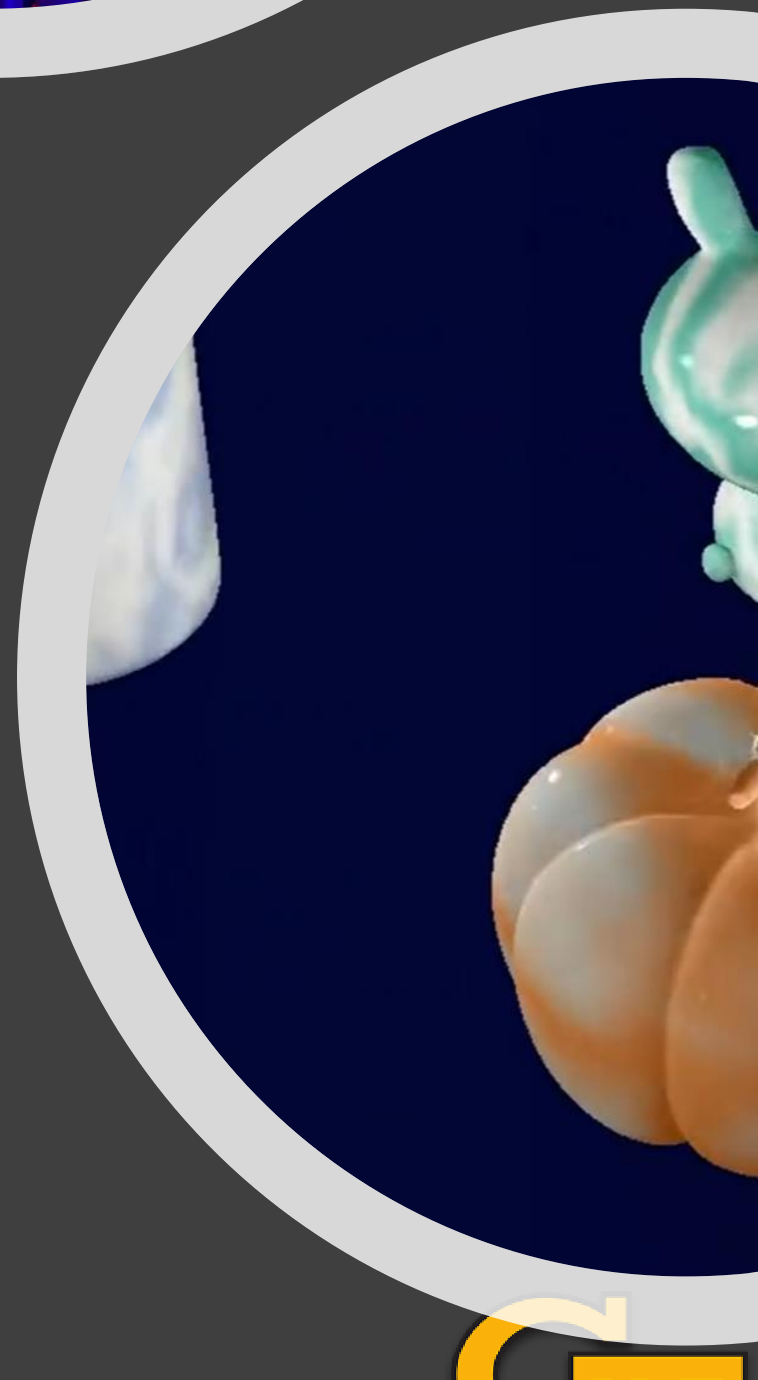


Phong Shading

invented by Dr. Bui Tuong Phong in 1973

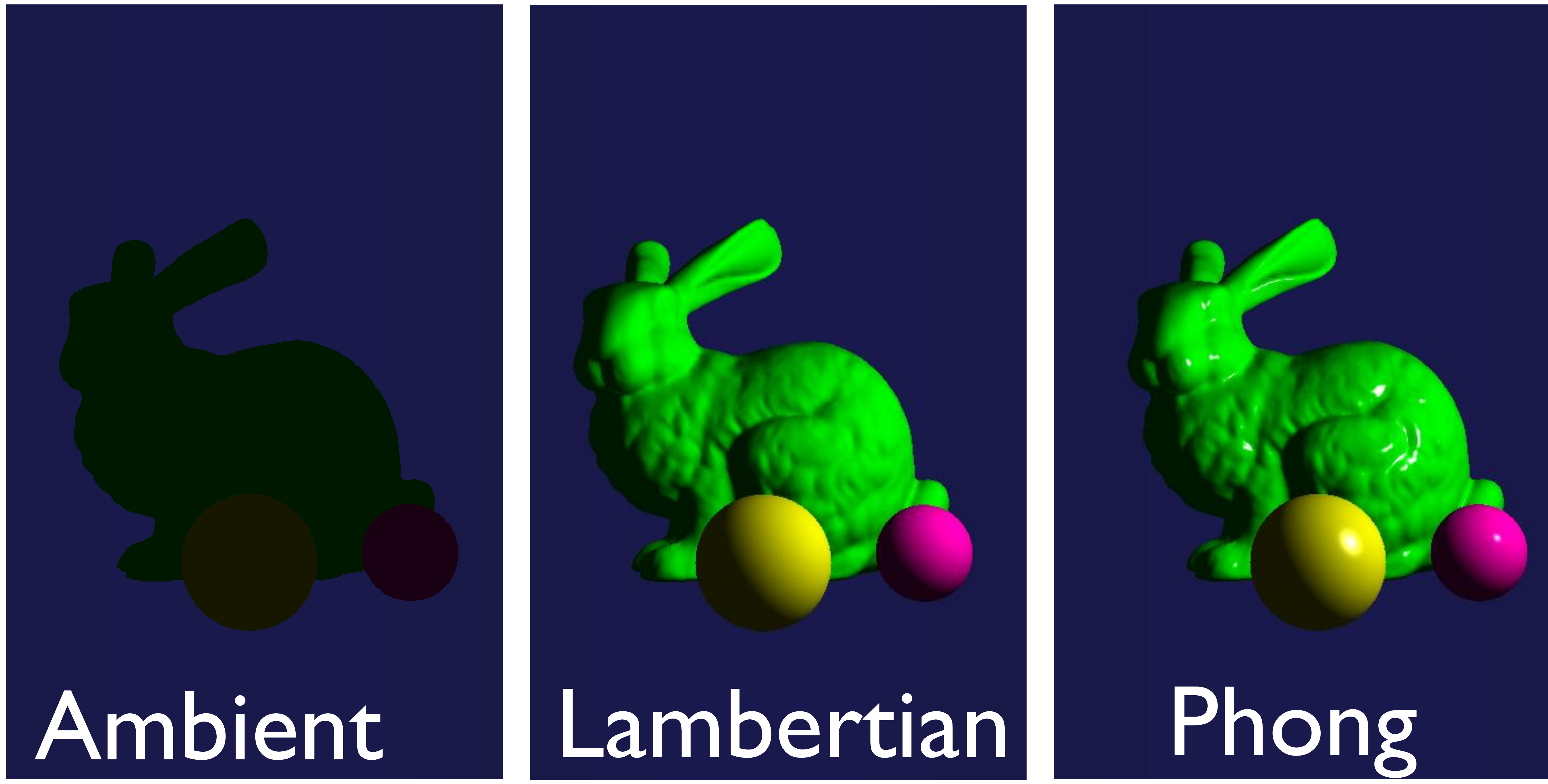
One of the greatest shading models in the history of computer graphics!





Phong Shading has enabled us to render shining objects easily with several lines of code!

Summary: Equations



- Ambient shading:

$$\boldsymbol{L}_{Ambient} = \sum_{j \in lights} (k_a \boldsymbol{I}_a^j)$$

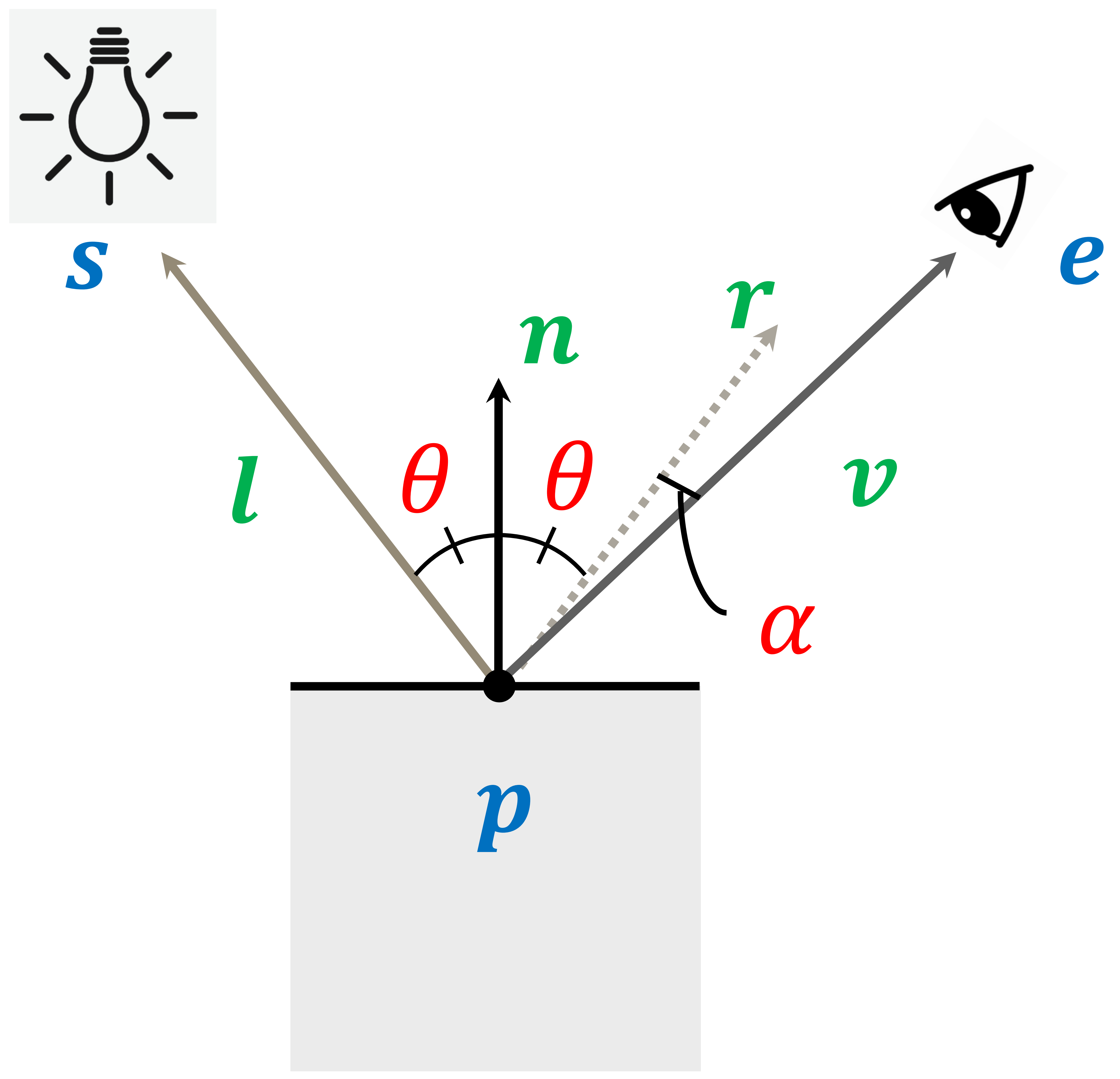
- Lambertian shading:

$$\boldsymbol{L}_{Lambertian} = \sum_{j \in lights} (k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right))$$

- Phong shading:

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} (k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right) + k_s \boldsymbol{I}_s^j \max(0, \vec{v} \cdot \vec{r}^j)^p)$$

Summary (Geometric Picture)



3 points

4 vectors

2 angles

Summary (Data Types)

- On the material side

```
vec3 ka;
vec3 kd;
vec3 ks;
float shininess;
```

- On the light side

```
vec3 la;
vec3 ld;
vec3 ls;
```

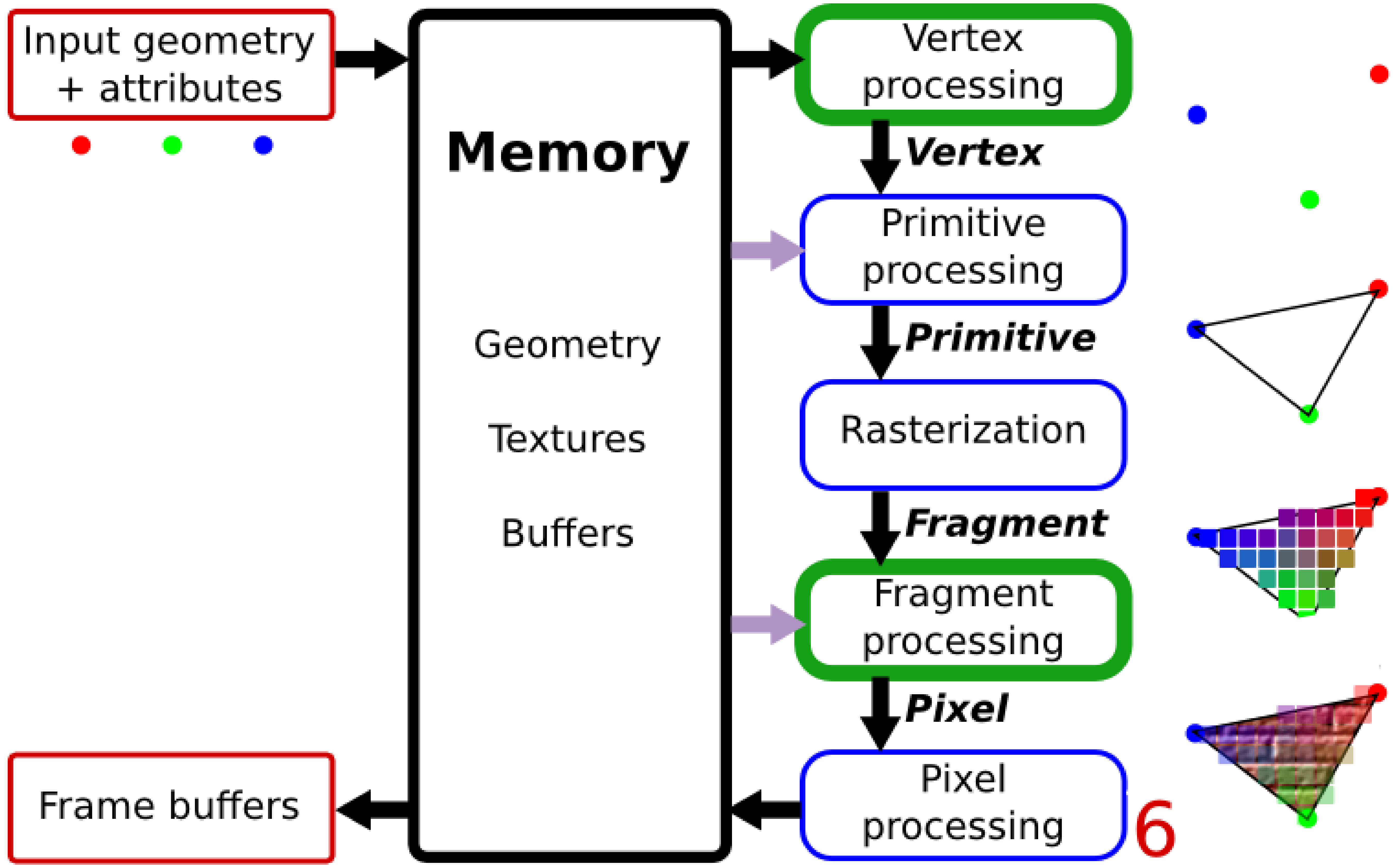
- On the geometry side

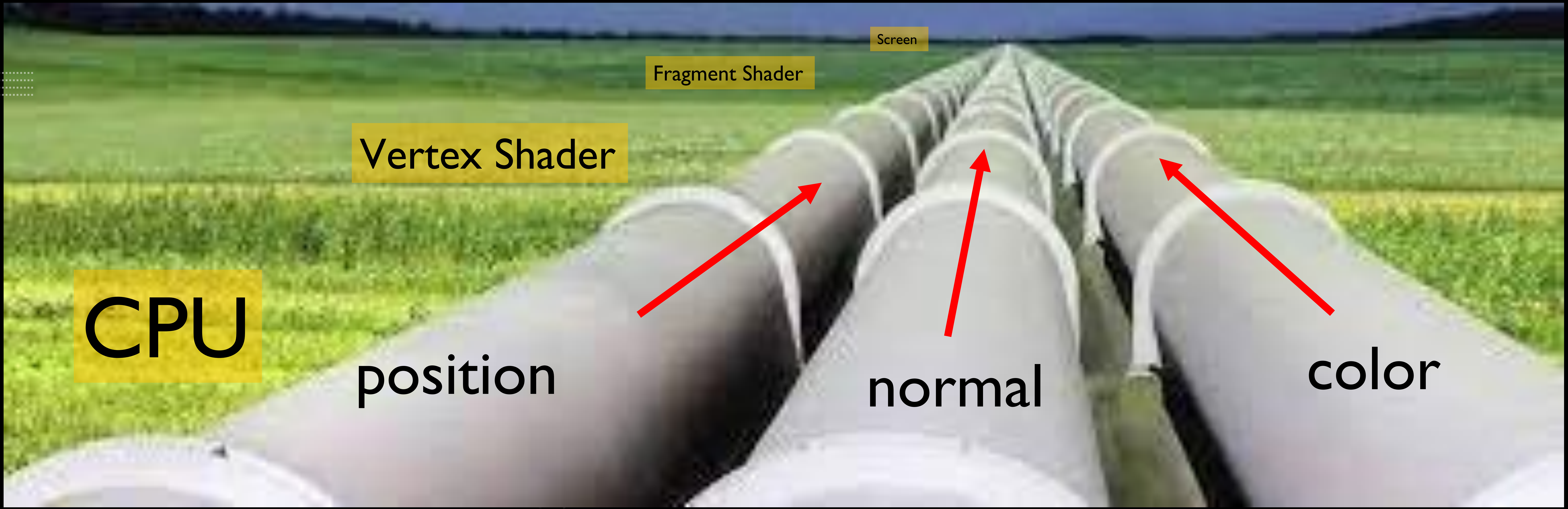
```
vec3 s;
vec3 p;
vec3 e;
vec3 n;
```

GLSL Implementation

Quick Recap:

Modern Graphics Pipeline



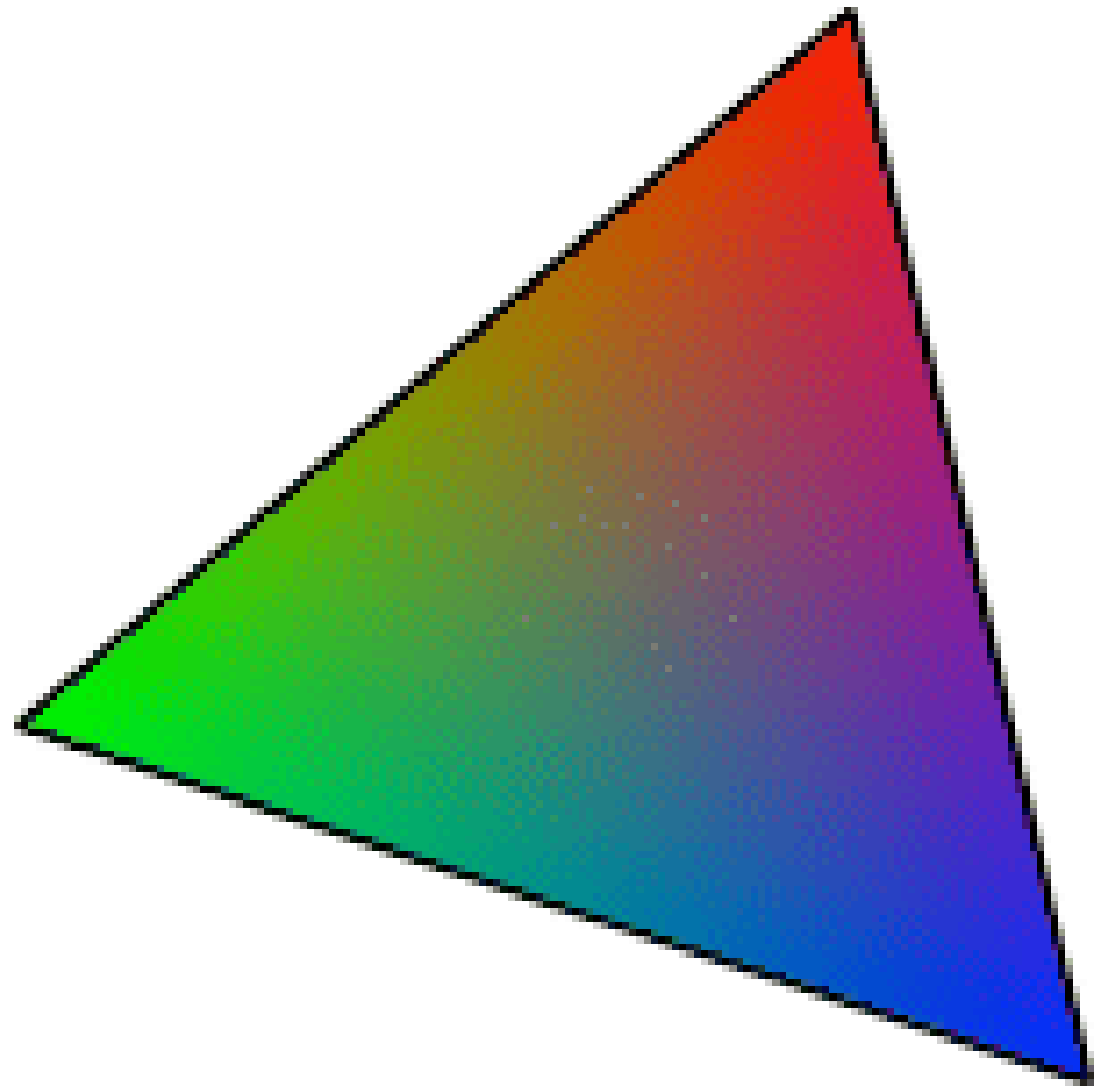


Data Flows in Pipelines

- Consider each variable as a component of a pipeline that channels data from the CPU through the Vertex Shader and Fragment Shader, and ultimately to the Screen.
- The transfer of data within each stage is facilitated by the use of 'in' and 'out' qualifiers to denote incoming and outgoing data, respectively.
- The continuity of data between stages is maintained by using the same variable name, thus allowing for a seamless data flow from one stage to the next within the pipeline.

We need barycentric interpolation when data flows from vertex shader to fragment shader

- Interpolate colors
  - $R = \alpha_0 R_0 + \alpha_1 R_1 + \alpha_2 R_2$
  - $G = \alpha_0 G_0 + \alpha_1 G_1 + \alpha_2 G_2$
  - $B = \alpha_0 B_0 + \alpha_1 B_1 + \alpha_2 B_2$
- Interpolate normal vectors
  - $N = \alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2$
- Interpolate z-buffer depth values
  - $z = \alpha_0 z_0 + \alpha_1 z_1 + \alpha_2 z_2$
- Interpolate texture coordinates $u = \alpha_0 u_0 + \alpha_1 u_1 + \alpha_2 u_2$
  - $v = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2$



Data Flow in Shaders

- **Vertex shader:**

```
in vec4 pos;
in vec4 normal;
out vec4 vtx_pos;
out vec4 vtx_normal;
…
void main()
{…}
```

Processed and interpolated data

Fragment shader:

```
in vec4 vtx_pos;
in vec4 vtx_normal;
out vec4 frag_color;
…
void main()
{…}
```

- Variables in the same data channel must have the name;
- out in vertex shader, and in in fragment shader.

The Simplest Vertex Shader

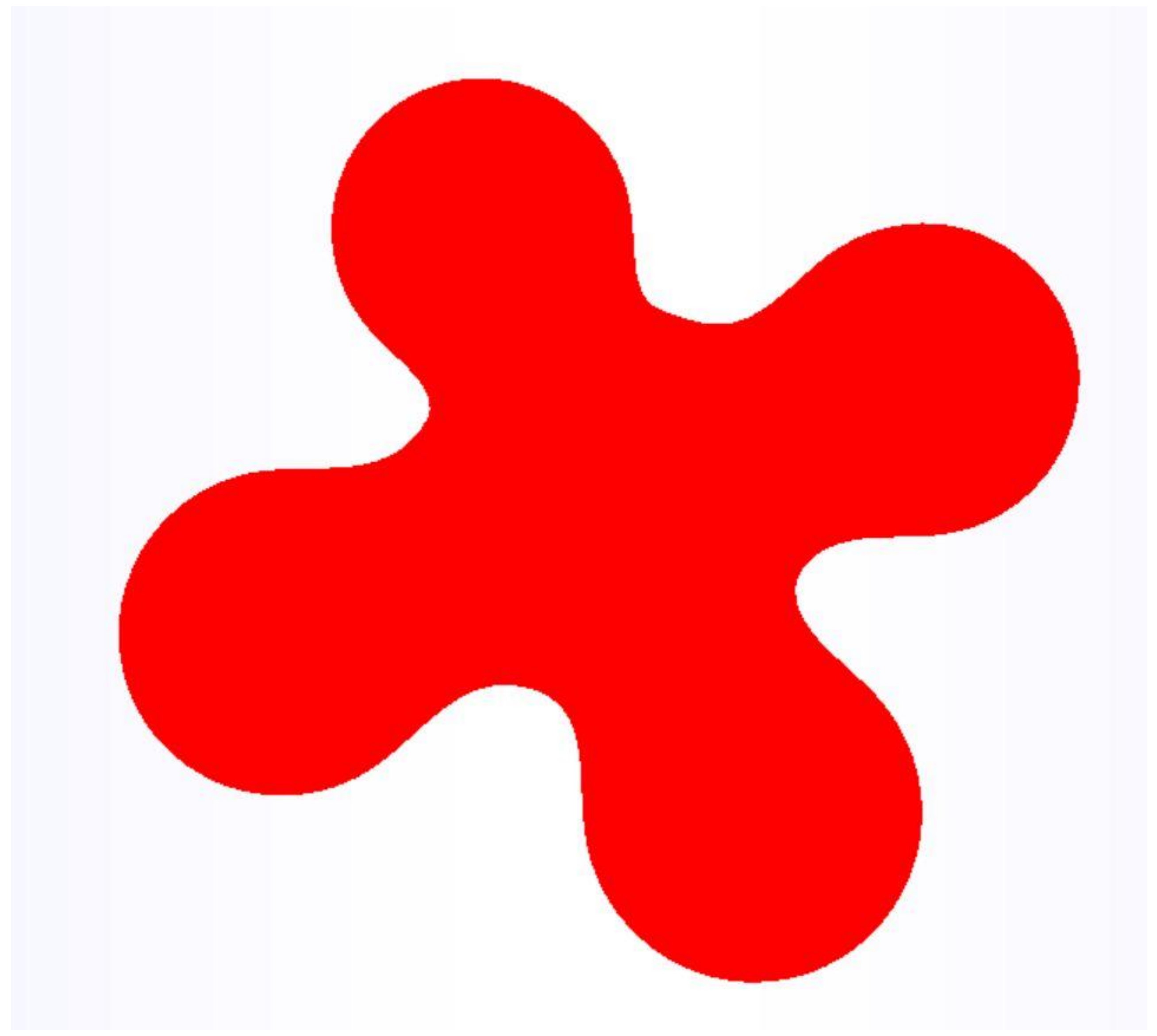
```
mat4 pvm;
in vec4 pos;
void main()

{
	gl_Position=pvm*vec4(pos.xyz,1.f);
}
```

The Simplest Fragment Shader

```
void main()

{
        frag_color=vec4(1.f,0.f,0.f,1.f);
}
```

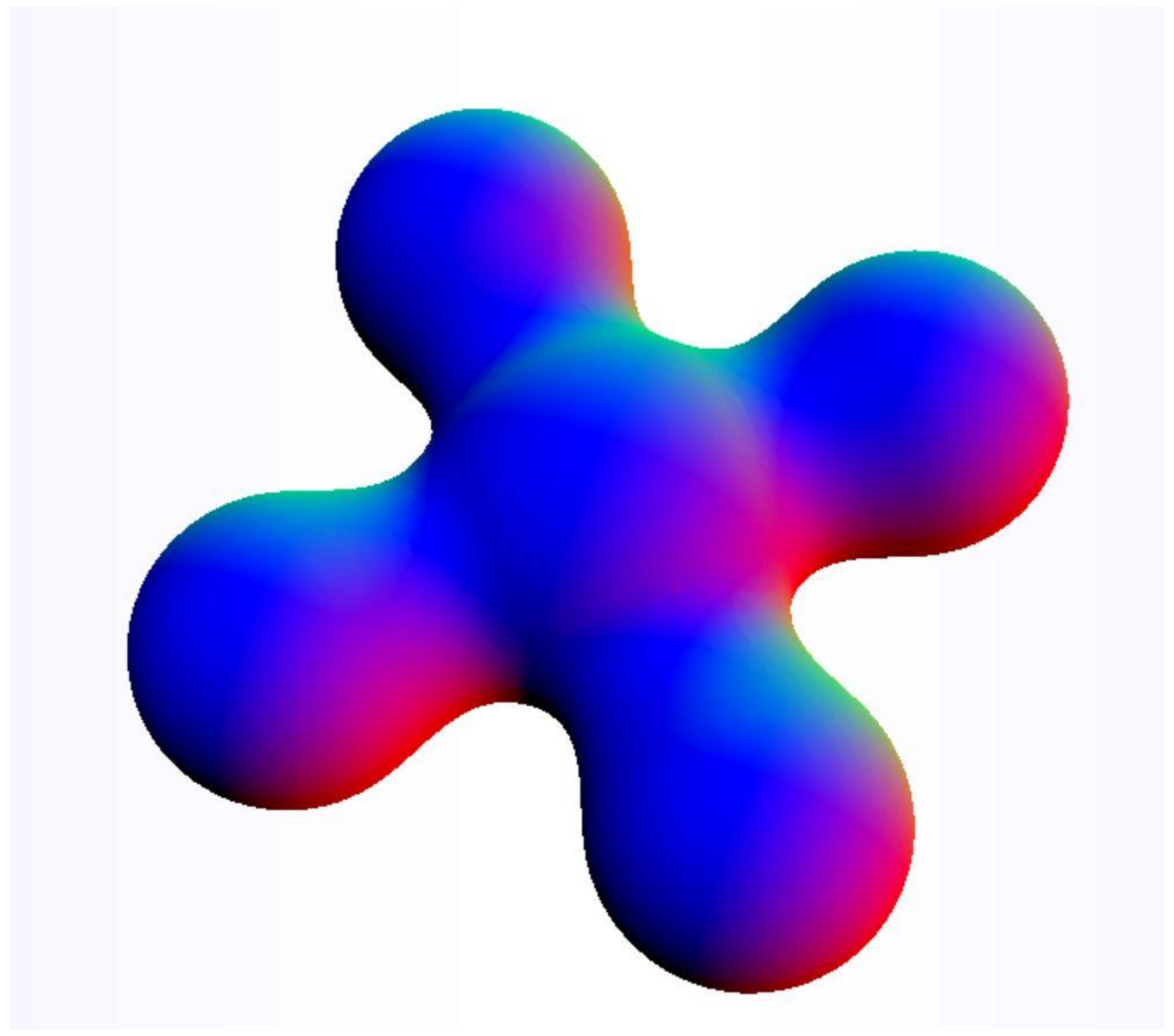


Another Example of Vertex Shader

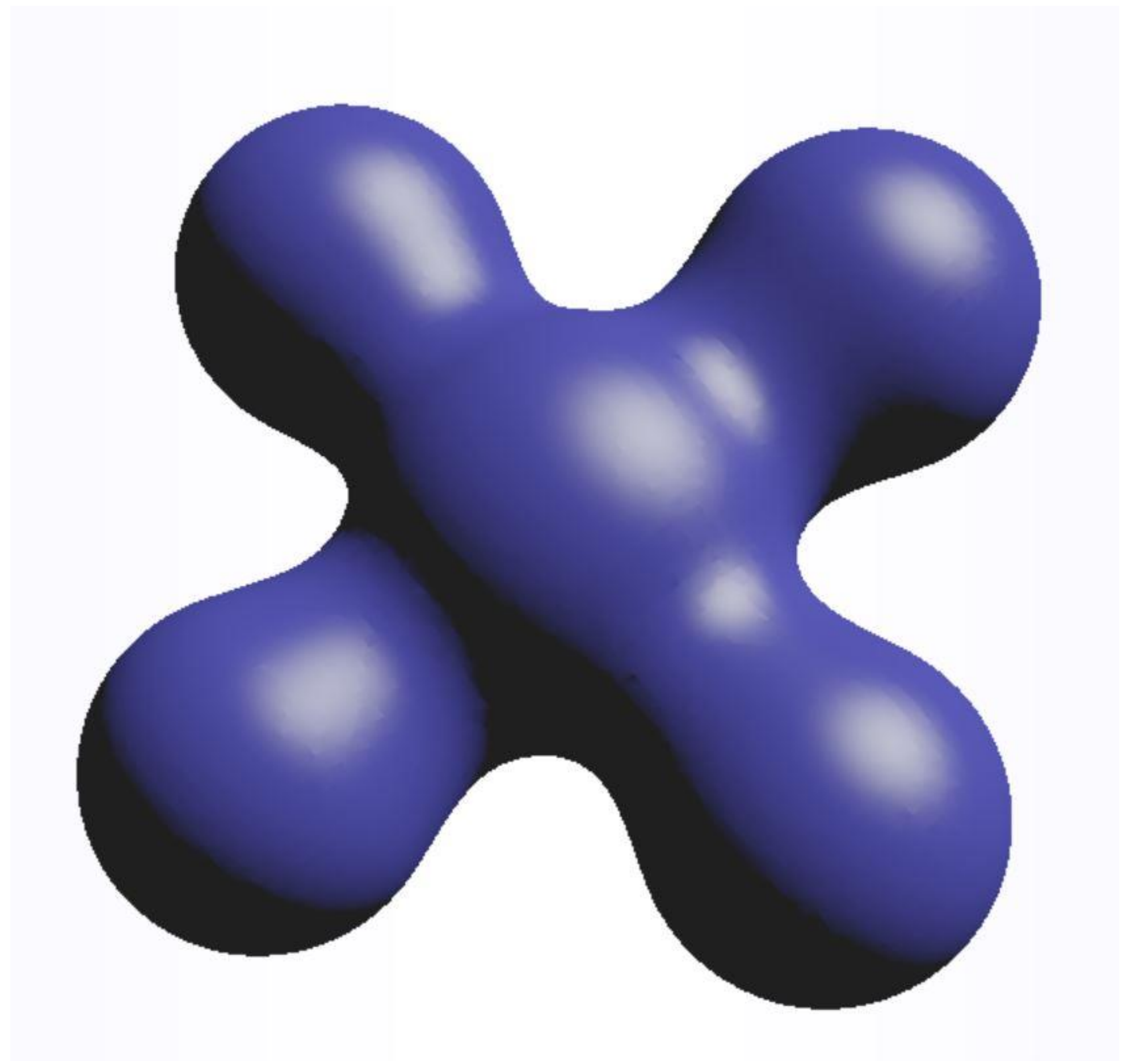
```
mat4 pvm;
in vec4 pos;
in vec4 normal;
out vec3 vtx_normal;
void main()
{
        gl_Position=pvm*vec4(pos.xyz,1.f);
        vtx_normal=normal.xyz;              ////assuming model matrix is identity
}
```

Example Fragment Shader: Use Normal as Color

```
in vec3 vtx_normal;
void main()
{
    vec3 norm=normalize(vtx_normal.xyz);
    frag_color=vec4(norm,1.f);
}
```



Example: Pseudocode for Phong Shading



```
in vec3 vtx_normal;
void main()
{
        ...
    vec3 norm=normalize(vtx_normal.xyz);
    vec3 color=phong(base_color, light_dir, norm, shininess, specular);
    frag_color=vec4(color,1.f);
}
```

Your implementation

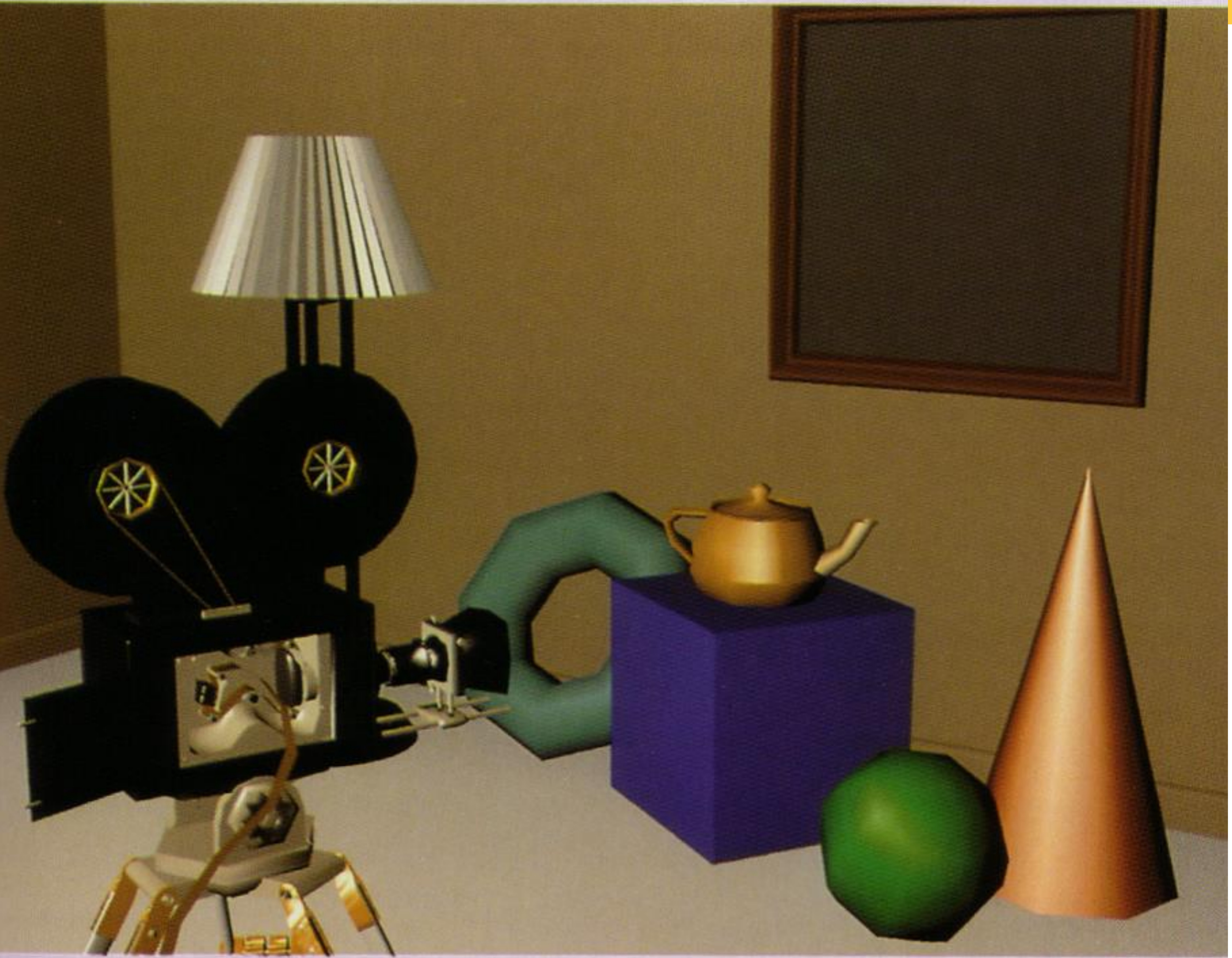
Where should we put the shading model?

1. Vertex shading:
   Compute illumination/shading **per vertex** in the **vertex stage**, and **interpolate shaded color** across triangles
2. Fragment shading:
   Compute illumination/shading **per pixel** in the **fragment stage**

Pipeline for Vertex Shading

- **Vertex stage** (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - **compute shaded color per vertex**
  - transform position (eye to screen space)
- **Rasterizer**
  - interpolated parameters: r, g, b color
- **Fragment stage** (output: color)
  - write to pixel

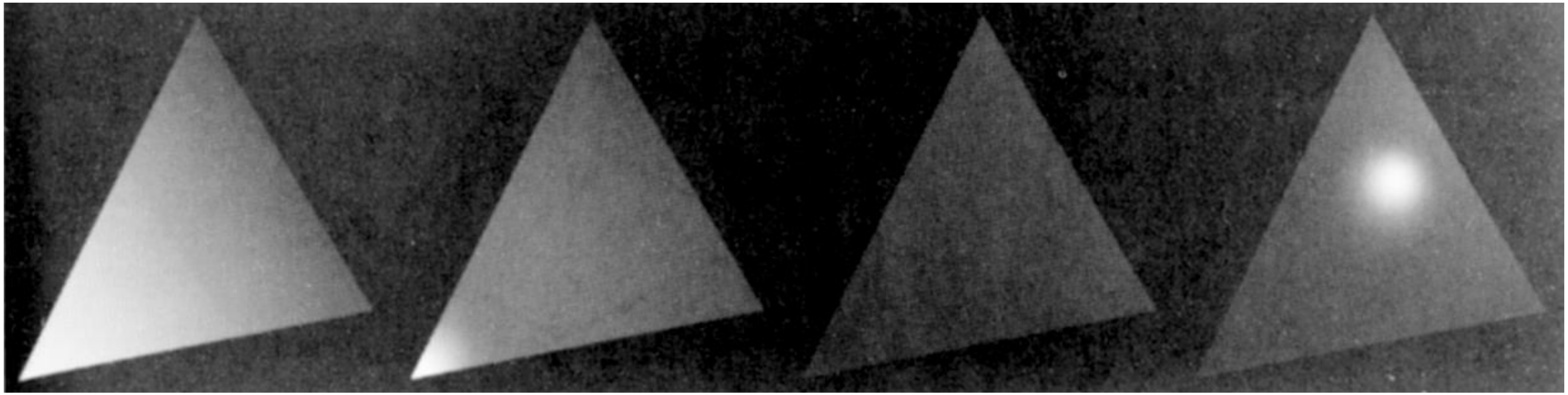
Gouraud Shading
shading and then interpolation



ɘ II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.
6.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using
'sPhotoRealistic RenderMan™ software.)

Put on Vertex Shader: Gouraud Shading

- Can apply Gouraud shading to any illumination model
  - it's just a method to interpolate per-vertex colors
- Results are not so good for fast-varying shading
  - e.g. problems with any highlights smaller than a triangle



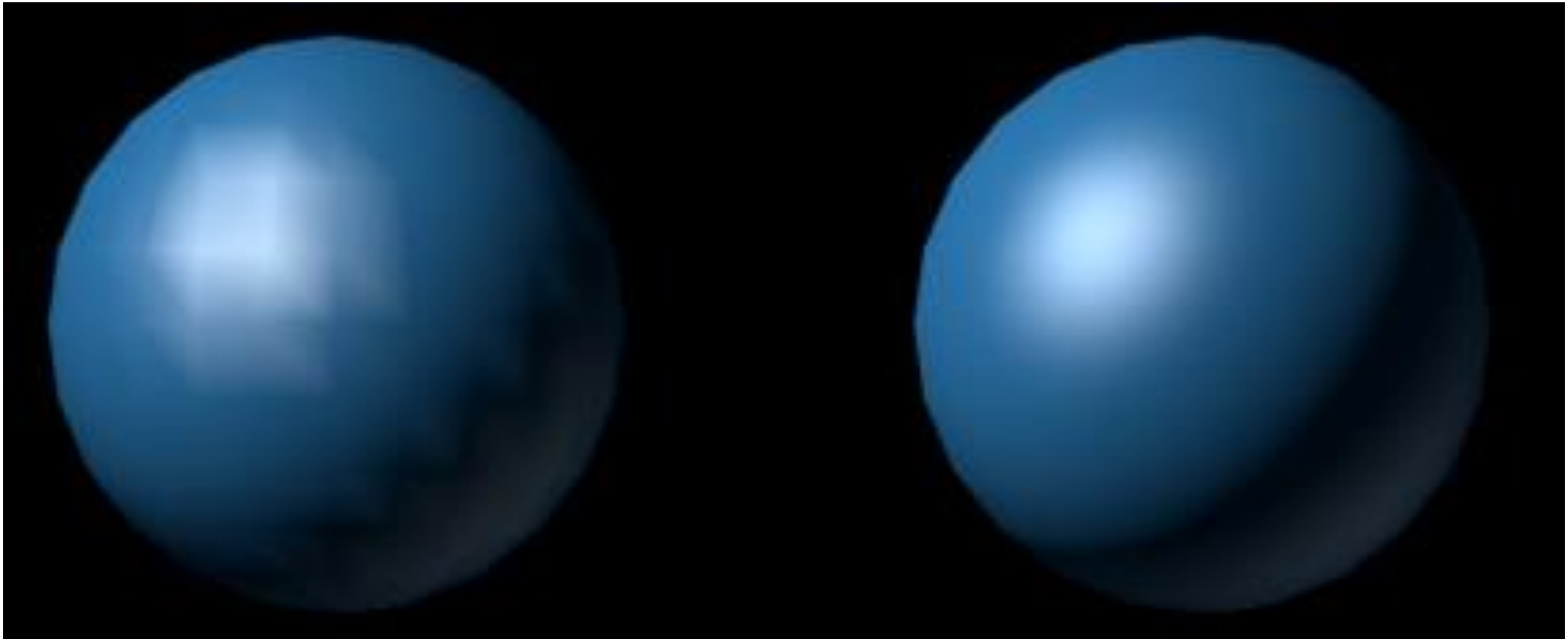
Put on Fragment Shader: Per-pixel (Phong) Shading

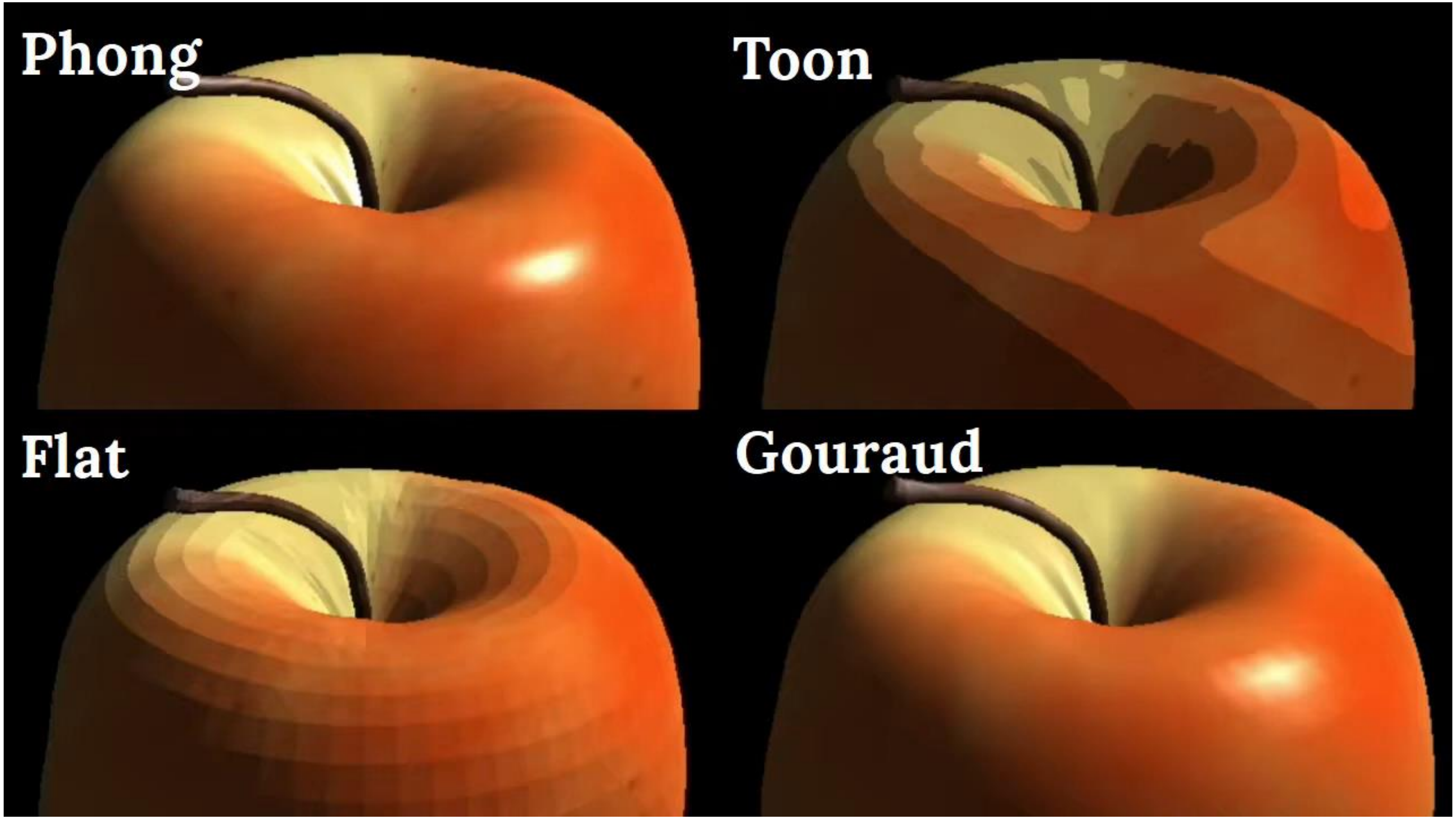
- Get higher quality by interpolating shading parameters (e.g. normal), instead of shaded color
- Evaluate the illumination model per pixel rather than per vertex (and normalizing the normal first)
    - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage

Pipeline for Fragment Shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through material color
- Rasterizer
  - interpolated parameters: r, g, b color; x, y, z normal
- Fragment stage (output: color, z')
  - **compute shading using interpolated color and normal**
  - write to pixel

Per-vertex vs Per-pixel Shading




Phong
Toon
Flat
Gouraud

Live Demo: GLSL Shading

https://gitlab.com/boolzhu/cs3451-computer-graphics-starter-code/-/tree/master/tutorials/tutorial_lighting?ref_type=heads