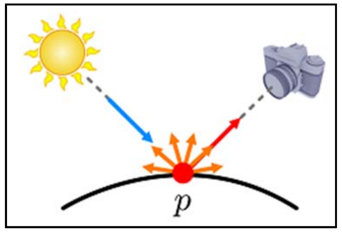
Chapter IX
Lighting

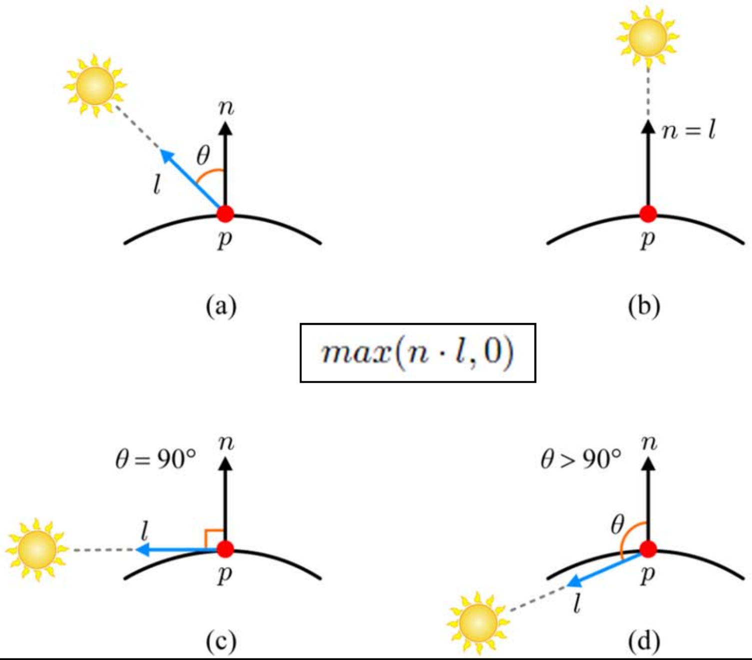
Phong Lighting Model

- Lighting refers to the techniques handling the interaction between light sources and objects.
- The most popular lighting method is based on the *Phong model*. It is widely adopted in commercial games and lays foundations of various advanced lighting techniques.
- The Phong model is composed of four terms:
  - diffuse
  - specular
  - ambient
  - emissive

Phong Lighting Model - Diffuse Term

- There can be various light types such as point, spot, and directional light sources.
- For now let’s take the simplest, the directional light source, where the *light vector* (l) is constant for a scene.
- The diffuse term is based on Lambert's law. Reflections from ideally diffuse surfaces (Lambertian surfaces) are scattered with equal intensity in all directions.
- So, the amount of perceived reflection is independent of the view direction, and is just proportional to the amount of incoming light.

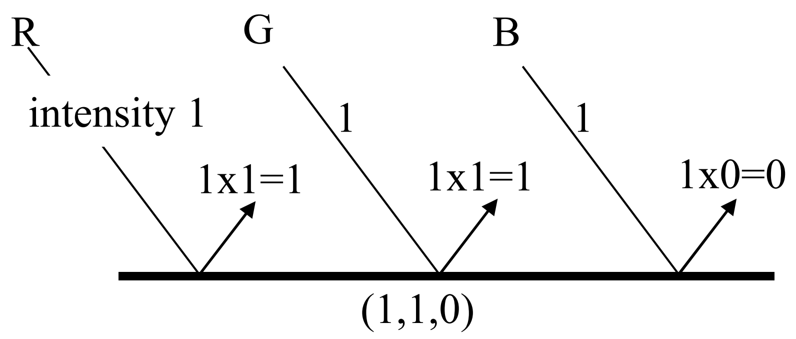


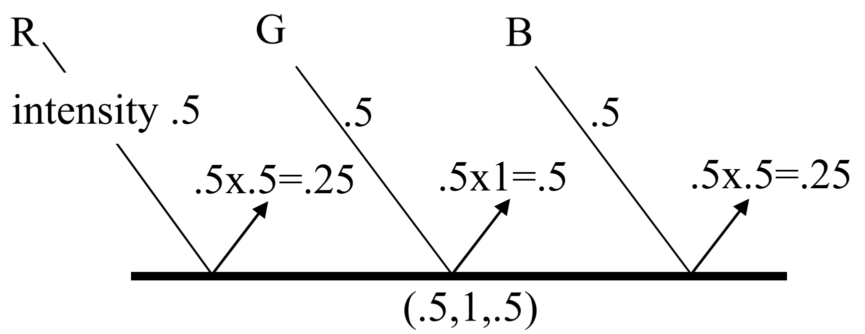


$$\boxed{max(n \cdot l, 0)}$$

Phong Lighting Model - Diffuse Term (cont'd)

- Suppose a white light (1,1,1). If an object lit by the light appears yellow, it means that the object reflects R and G and absorbs B. We can easily implement this kind of filtering through material parameter, i.e., if it is (1,1,0), then (1,1,1)⊗(1,1,0)=(1,1,0) where ⊗ is component-wise multiplication.





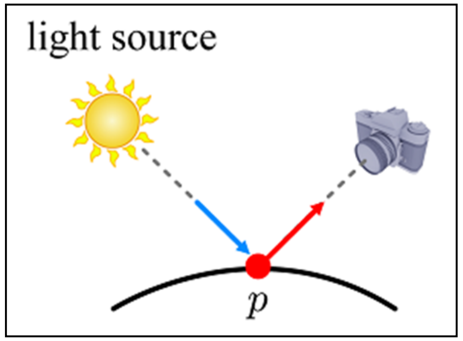
$\boxed{s_d \otimes m_d}$

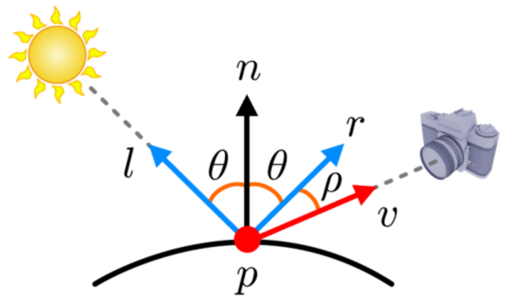
- The diffuse term: $max(n \cdot l, 0)s_d \otimes m_d$

- In general, the texture provides m_d.

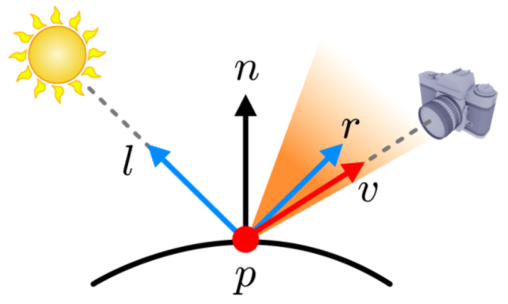
Phong Lighting Model - Specular Term

- The specular term is used to make a surface look shiny via *highlights*, and it requires *view vector* (v) and *reflection vector* (r) in addition to *light vector* (l).



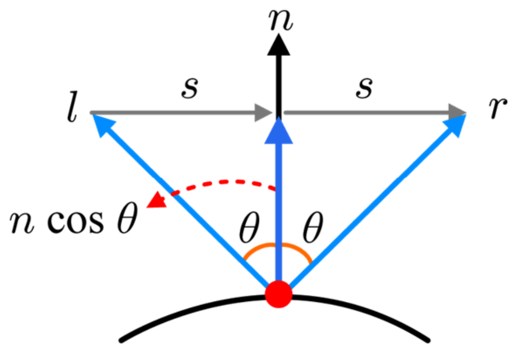


(a)



(b)

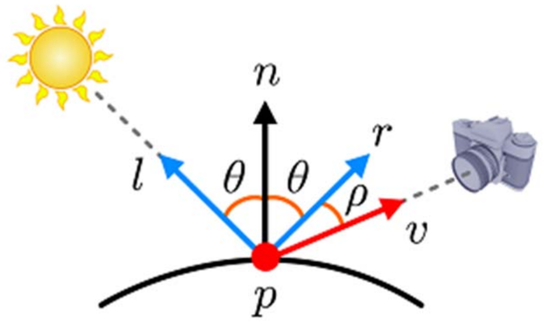
- Computing the reflection vector

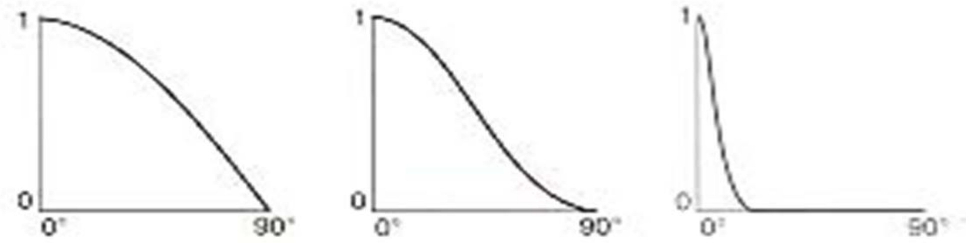


$$s = n\cos\theta - l$$
$$s = r - n\cos\theta$$
$$\begin{aligned} r &= 2n\cos\theta - l \\ &= 2n\,(n \cdot l) - l \end{aligned}$$

Phong Lighting Model - Specular Term (cont'd)

- Whereas the diffuse term is view-independent, the specular term is highly view-dependent.
  - For a perfectly shiny surface, the highlight at p is visible only when ρ equals 0.
  - For a surface that is not perfectly shiny, the maximum highlight occurs when ρ equals 0, but falls off sharply as ρ increases.
  - The rapid fall-off of highlights is often approximated by $(r \cdot v)^{sh}$, where sh denotes shininess.





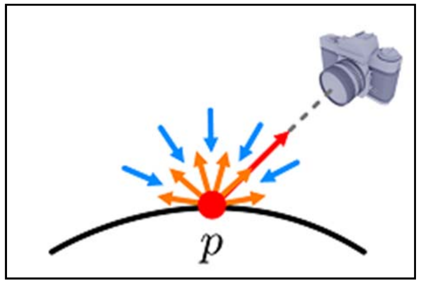
cosρ *cos*^{2}*ρ* *cos*^{64}*ρ*

$\boxed{(r \cdot v)^{sh}}$

- The specular term: $(max(r \cdot v, 0))^{sh} s_s \otimes m_s$
- Unlike m_d, m_s is usually a gray-scale value rather than an RGB color. It enables the highlight on the surface to end up being the color of the light source.

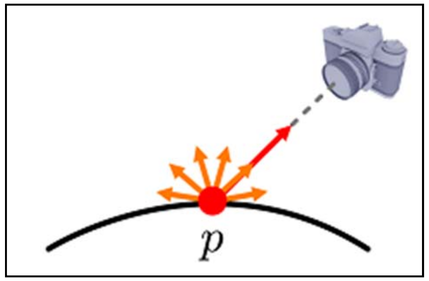
Phong Lighting Model – Ambient and Emissive Terms

- The ambient light describes the light reflected from the various objects in the scene, i.e., it accounts for *indirect lighting*.
- As the ambient light has bounced around so much in the scene, it arrives at a surface point from all directions, and reflections from the surface point are also scattered with equal intensity in all directions.



$s_a \otimes m_a$

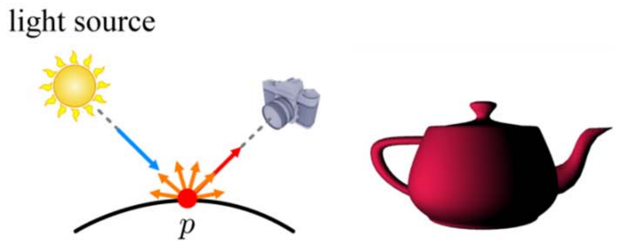
- The last term of the Phong model is the emissive term m_e that describes the amount of light emitted by a surface itself.



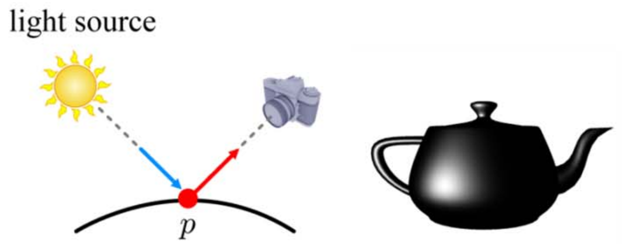
Phong Lighting Model

- The Phong model sums the four terms!!

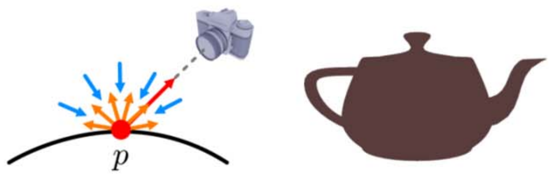
$$\boxed{max(n \cdot l, 0) s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$



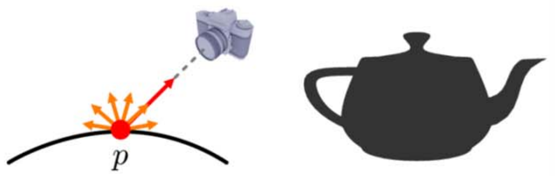
(a) diffuse



(b) specular



(c) ambient



(d) emissive

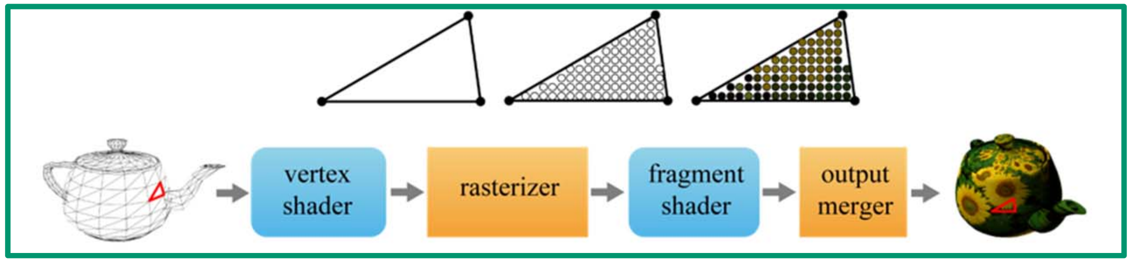
(e) sum

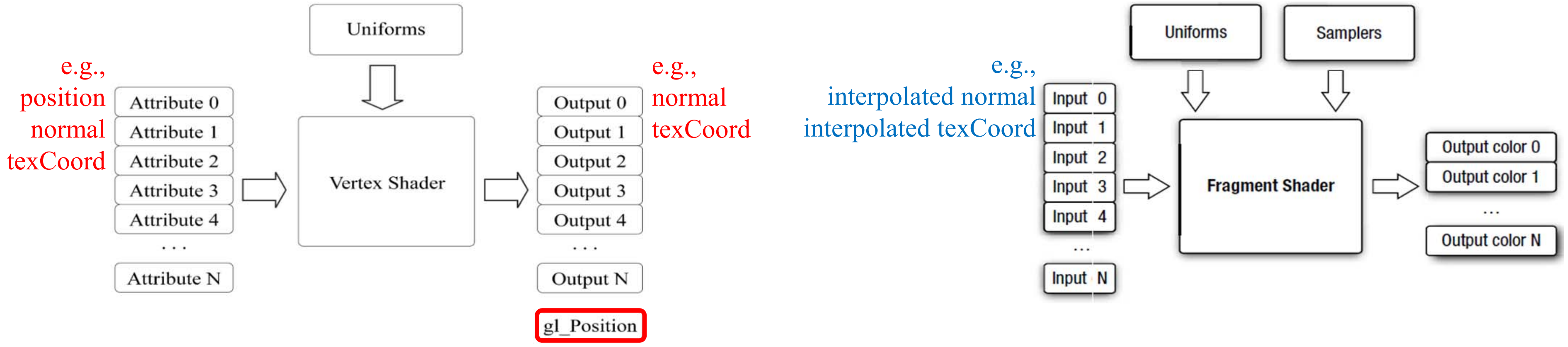
Vertex Shader (revisited)

- Recall our first vertex shader presented in Chapter 6.

```
#version 300 es

uniform mat4 worldMat, viewMat, projMat;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 texCoord;

out vec3 v_normal;
out vec2 v_texCoord;

void main() {
   gl_Position = projMat * viewMat * worldMat * vec4(position, 1.0);
   v_normal = normal;
   v_texCoord = texCoord;
}
```

Vertex and Fragment Shaders



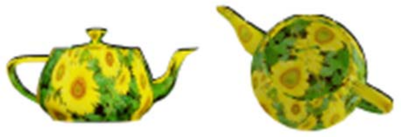


- The inputs to the fragment shader
  - Inputs: The per-vertex output variables produced by the vertex shader are interpolated to determine the per-fragment ones.
  - Uniforms to be used by the fragment shader.
  - Samplers = Textures.
- The output: one or more fragment colors that are passed to the output merger.

Fragment Shader (revisited)

- A simplest fragment shader

```
uniform sampler2D s_tex0;

in vec2 v_texCoord;
layout(location = 0) out vec4 fragColor;

void main() {
        fragColor = texture(s_tex0, v_texCoord);
}
```

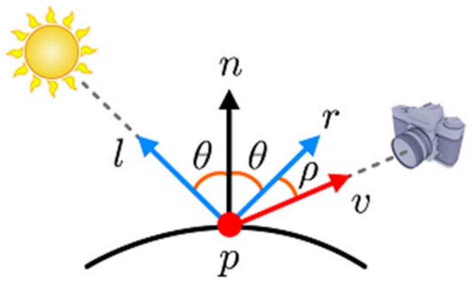


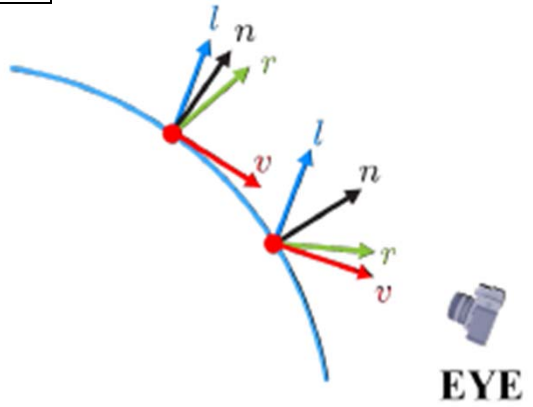
- The fragment shader declares a uniform variable, `s_tex0`, of type `sampler2D`, which represents a texture map.
- Our first vertex shader simply outputs 2D texture coordinates without modification. The per-vertex texture coordinates were interpolated so that the per-fragment texture coordinates, `v_texCoord`, are now passed to the fragment shader and used for fetching the texture, `s_tex0`.
- A built-in function `texture` accesses the texture and returns a `vec4` representing the color fetched from the texture. (The fragment shader does not accept another variable, `v_normal`, which will be used later though.)

Per-fragment Lighting

- Texturing alone is never enough to make an object look realistic. Let's light or illuminate the object! The fragment shader is in charge of lighting, and it implements the Phong model.

$$\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$

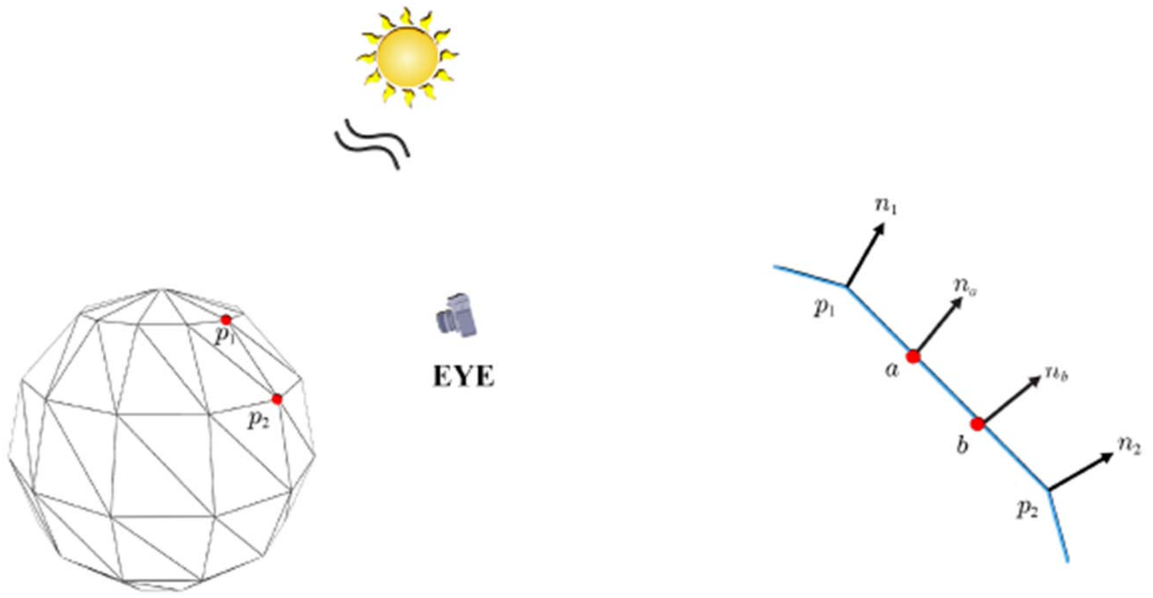




- Consider two points on the object surface, each of which corresponds to a distinct fragment. Each surface point is associated with its own *l*, *n*, *r*, and *v*.
  - As we assume the directional light, *l* is fixed and will be directly input to the fragment shader.
  - In contrast, *n*, *r*, and *v* vary across the object surface. Specifically, a unique pair of *n* and *v* should be provided for the fragment shader, which then computes *r* using *l* and *n*.

Per-fragment Lighting (cont'd)

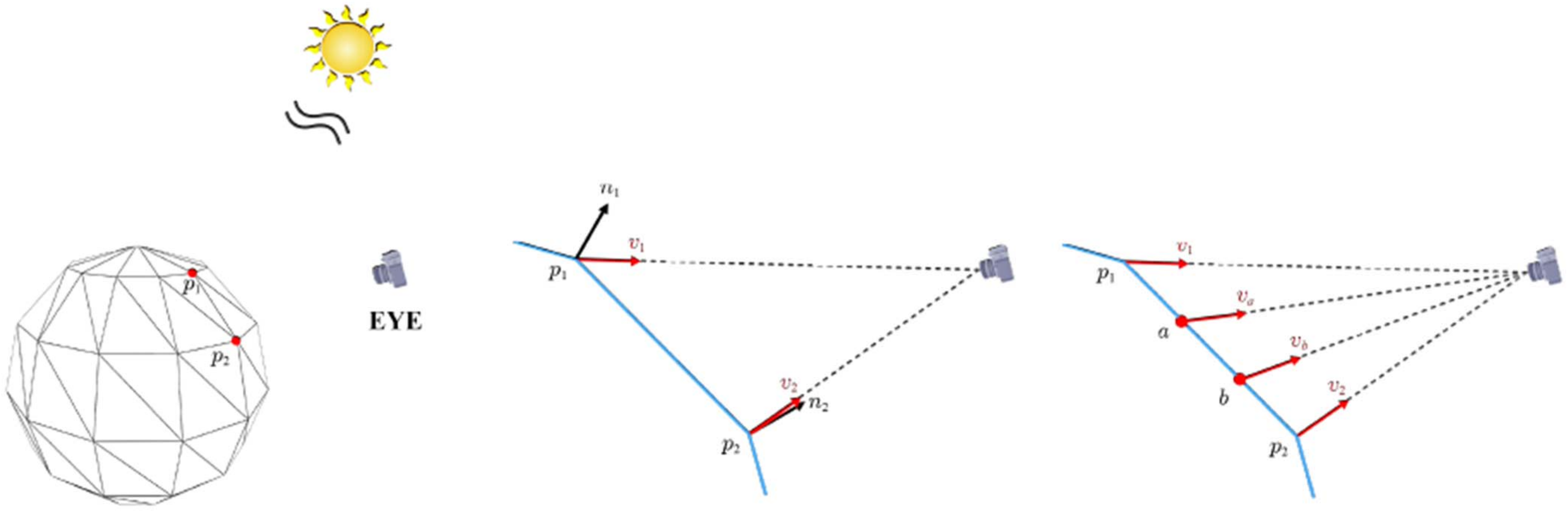
- The rasterizer interpolates the vertex normals to provide n for each fragment.
  - Note that l is defined in the world space.
  - For computing r as well as $n \cdot l$ of the diffuse term, n needs to be a world-space vector.
  - The vertex shader will world-transform the object-space normal of each vertex and then pass the world-space normal to the rasterizer.



  - In a and b, each of which corresponds to a distinct fragment, the interpolated world-space normals, n_a and n_b, are assigned, respectively.

Per-fragment Lighting (cont'd)

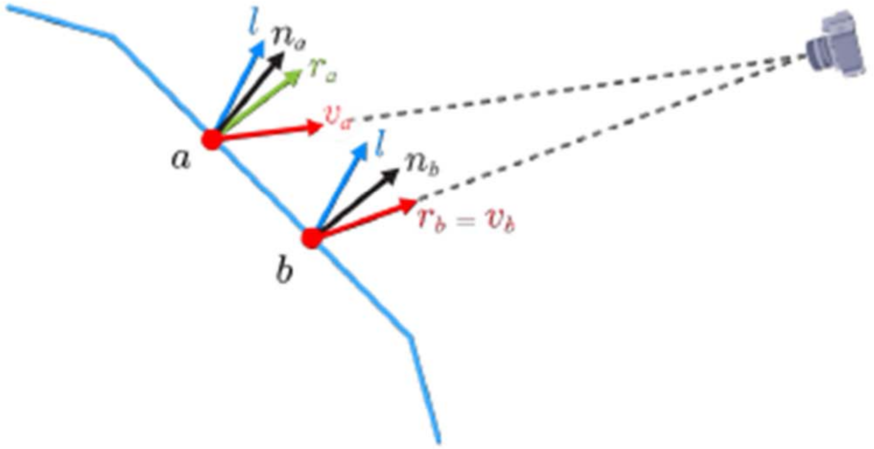
- We can also resort to the rasterizer for computing *v* for each fragment.
  - For this, the vertex shader transforms the object-space position of each vertex to the world space, connects it to the camera position, **EYE**, which was defined in the world space, and then passes the *view vector* to the rasterizer.
  - Such per-vertex view vectors are interpolated across a triangle.

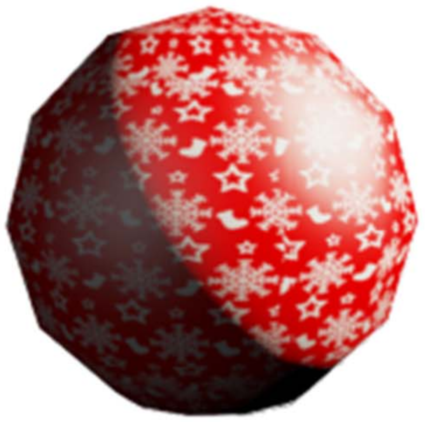


  - The fragments, *a* and *b*, are assigned the world-space view vectors, v_a and v_b, respectively.

Per-fragment Lighting (cont'd)

- The fragment shader computes r using l and n, and then implements lighting.
- Observe that l, n, r, and v are all defined in the world space. Do not get confused! The fragment shader processes such *world-space* vectors for determining the color of the *screen-space* fragment.
- Consider two fragments, a and b.
  - They have the same light vector, l, from the directional light source.
  - For a, r_a makes a considerable angle with v_a and therefore the camera perceives little specular reflection.
  - In contrast, r_b happens to be identical to v_b and therefore the camera perceives the maximum amount of specular reflection.



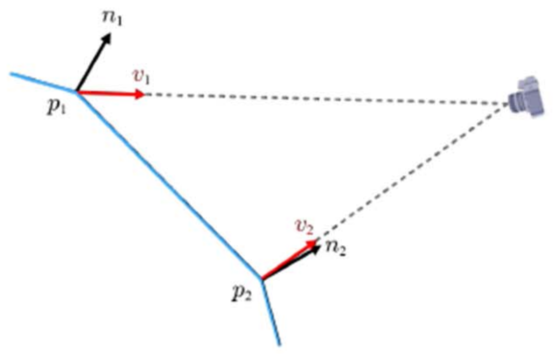


Per-fragment Lighting

- The vertex shader.

$$\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$

```
#version 300 es

uniform mat4 worldMat, viewMat, projMat;
uniform vec3 eyePos;

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;
layout(location = 2) in vec2 texCoord;

out vec3 v_normal;
out vec3 v_view;
out vec2 v_texCoord;

void main() {
    v_normal = normalize(transpose(inverse(mat3(worldMat))) * normal);
    vec3 worldPos = (worldMat * vec4(position, 1.0)).xyz;
    v_view = normalize(eyePos - worldPos);
    v_texCoord = texCoord;
    gl_Position = projMat * viewMat * worldMat * vec4(position, 1.0);
}
```



Per-fragment Lighting (cont'd)

- The fragment shader.

$$\boxed{max(n \cdot l, 0)s_d \otimes m_d + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e}$$

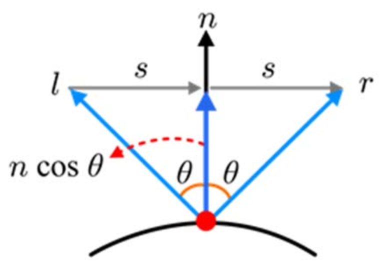
```
#version 300 es

precision mediump float;

uniform sampler2D s_tex0;
uniform vec3 matSpec, matAmbi, matEmit; // Ms, Ma, Me
uniform float matSh; // shininess
uniform vec3 srcDiff, srcSpec, srcAmbi; // Sd, Ss, Sa
uniform vec3 lightDir; // directional light

in vec3 v_normal;
in vec3 v_view;
in vec2 v_texCoord;

layout(location = 0) out vec4 fragColor;
```

Per-fragment Lighting (cont'd)

- The fragment shader.

$$max(n \cdot l, 0)s_d \otimes \boxed{m_d} + (max(\boxed{r} \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e$$

```
void main() {

    vec3 normal = normalize(v_normal);
    vec3 view = normalize(v_view);
    vec3 light = normalize(lightDir);

    // diffuse term
    vec3 matDiff = texture(s_tex0, v_texCoord).rgb;
    vec3 diff = max(dot(normal, light), 0.0) * srcDiff * matDiff;

    // specular term
    vec3 refl = 2.0 * normal * dot(normal, light) - light;
    vec3 spec = pow(max(dot(refl, view), 0.0), matSh) * srcSpec * matSpec;

    // ambient term
    vec3 ambi = srcAmbi * matAmbi;

    fragColor = vec4(diff + spec + ambi + matEmit, 1.0);
}
```



$r = 2n\,(n \cdot l) - l$