Lighting

and shading

Type of lights

in OpenGL

Main type of lights

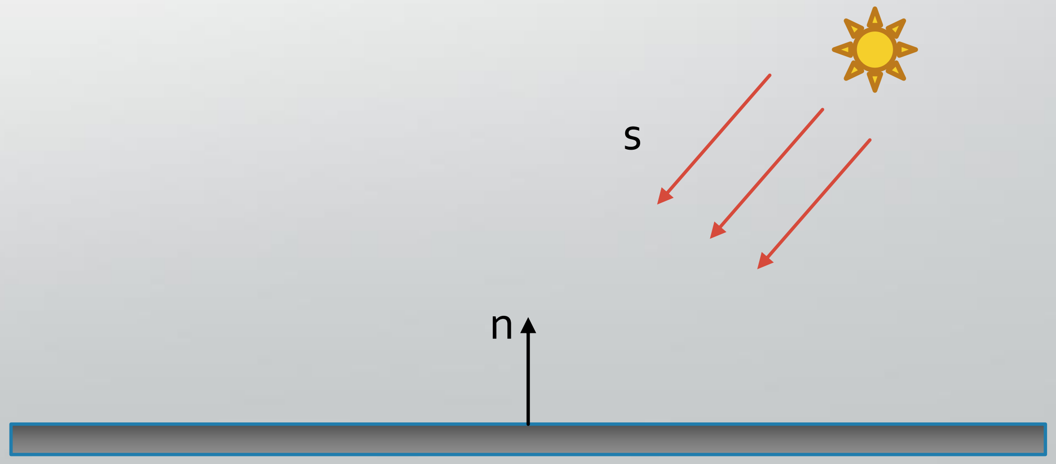
- **Directional** – the light behaves similar to the sunlight, position of the light is irrelevant only direction of the light matters
- **Point** – the light behaves like a light bulb. Has a limited range and position of the light is crucial in calculating the light direction
- **Spotlight** – the light behaves like a spotlight…duh. The position of the light is important but also the orientation of the spotlight.

Directional

light

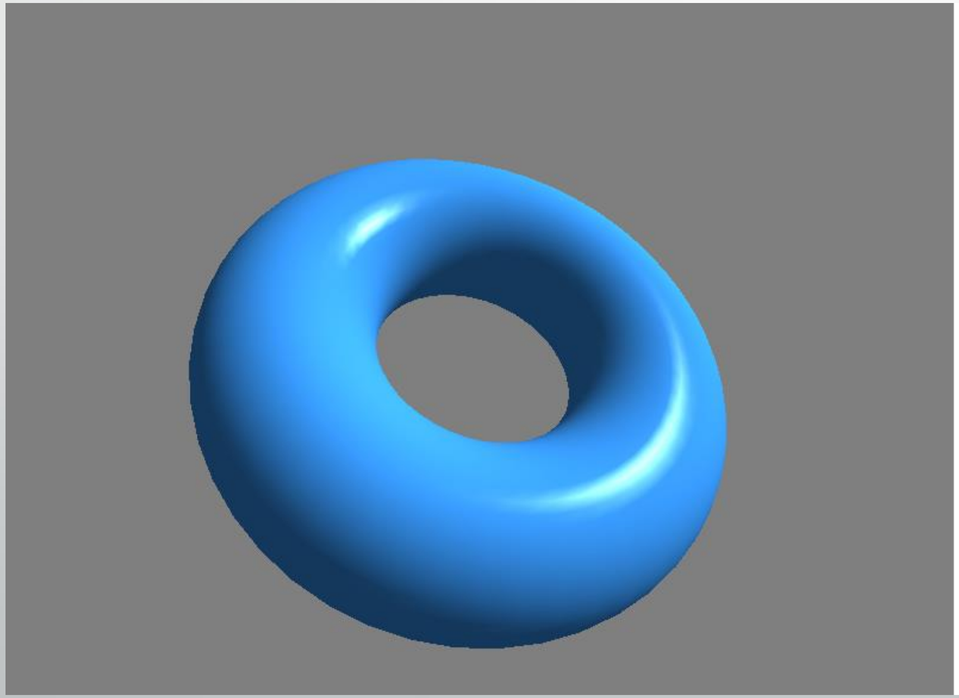
Directional light

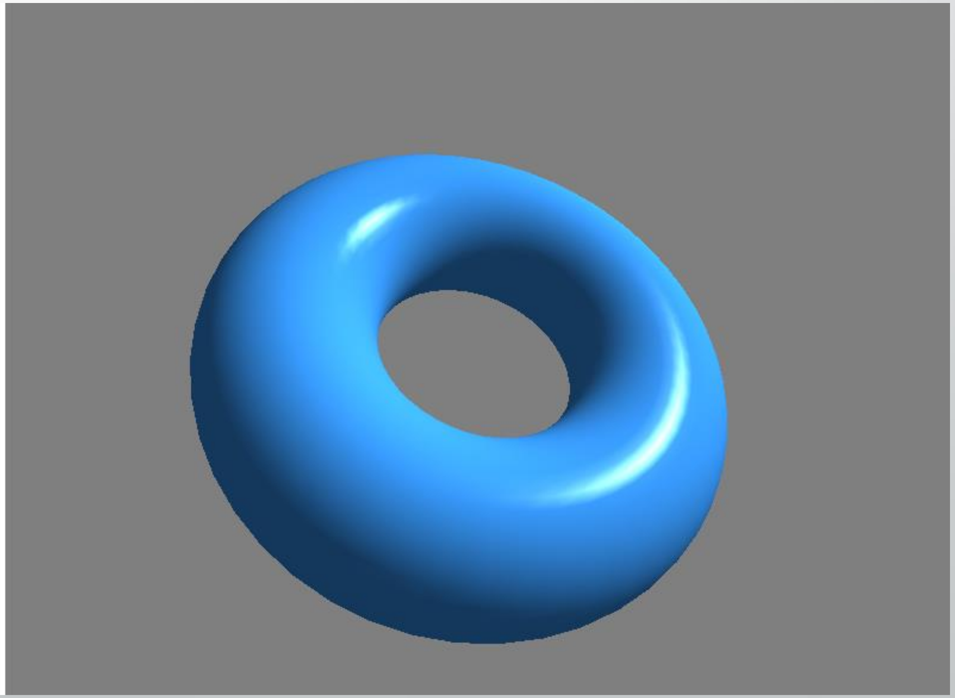
- The light is "extremely far away", that means the light rays arriving from the light source are parallel
- Light position is irrelevant only the direction of the light matters
- There's no fall off, the intensity of the light stays constant
- There is a visual difference between a point light and a directional light



Directional light

Directional light | vs | Point light





Directional light

- If we use a directional light, we can avoid calculating the light direction in the shader (we can send the direction vector directly through a uniform)
- At the moment we calculate light direction based on the position of the light and the vertex position.

s = lightPosition – vertex position; //direction of the light

Directional light

- lightPosition is a vec4 = (x, y, z, w)
- For light direction calculation on a point light we use only **x**, **y** and **z**
- We can use **w** to indicate if the light is directional or not

```
if(lightPosition.w == 0)
        s = lightPosition.xyz
else
        s = lightPosition – vertex position;
```

Not ideal for parallel processing/calculations on the GPU.

Directional light

- Write separate shaders, one for directional light and one for point light. That guarantees one calculation can be performed in parallel on the GPU.

- Don't use an **if statement** but still use w value. Works like before but we perform one calculation and it can take advantage of the parallel processing power of the GPU.

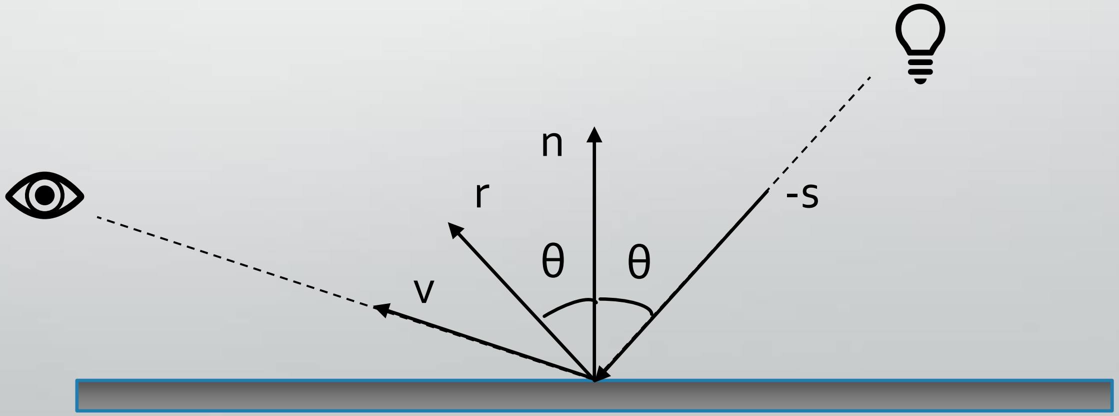
s = lightPosition – (vertex position · lightPosition.w);

Point

light

Phong reflection

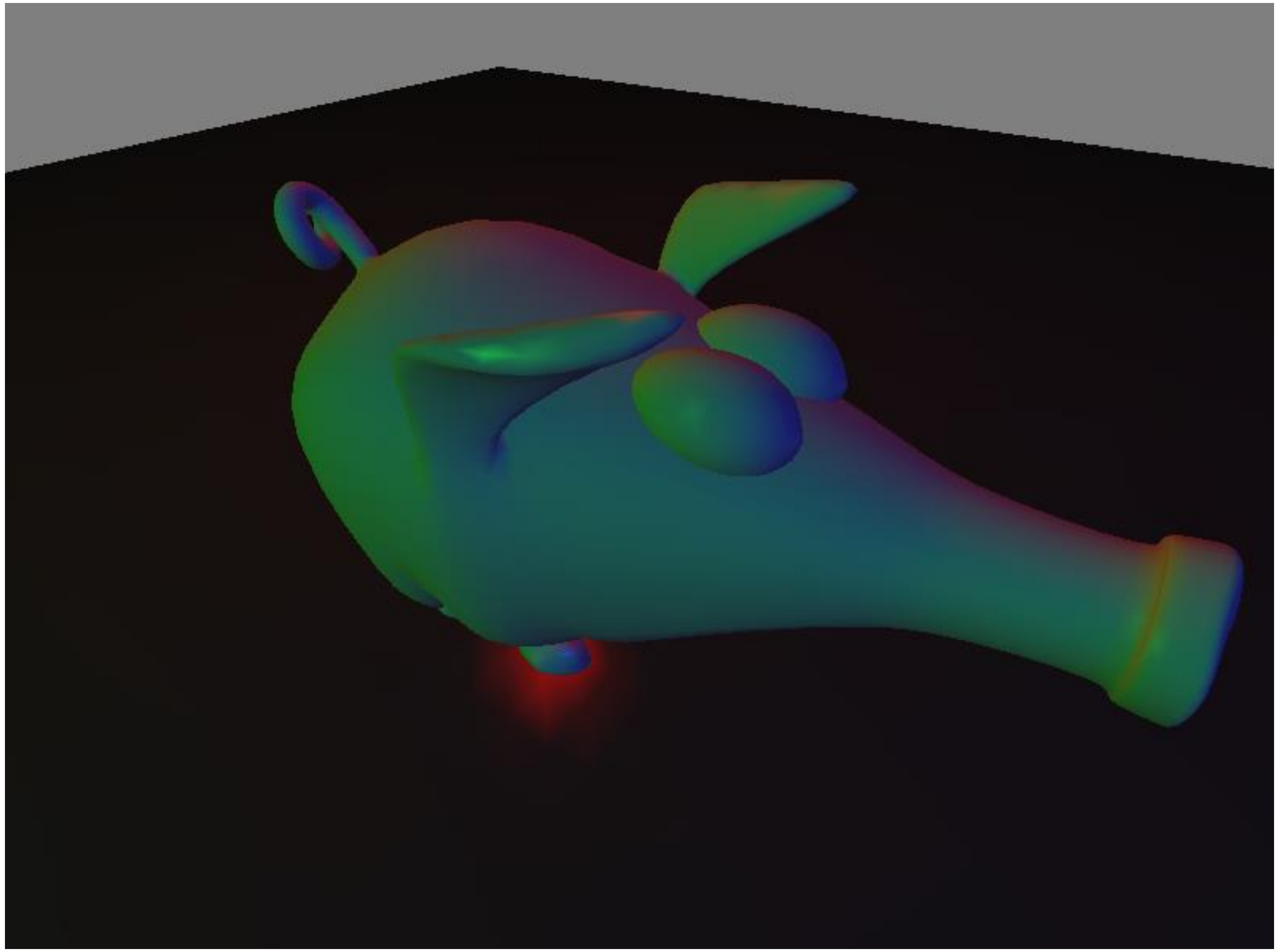
	Ambient		Diffuse		Specular
$I =$	I_a	+	I_d	+	I_s
$I =$	$K_a \cdot L_a$	+	$K_d \cdot L_d \cdot (s.n)$	+	$K_s \cdot L_s \cdot (r.v)^f$



Multiple point lights

- we need to evaluate the reflection model for each light, and sum the results to determine the total light intensity reflected by a surface
- For 3 lights:

$$P_F = P_{L1} + P_{L2} + P_{L3}$$



Multiple point lights

- We can use **uniform arrays** to store the position and intensity of each light
- Single uniform variable stores the values for multiple lights. See example bellow:

```
uniform struct LightInfo
{
    vec4 Position;      // Light position
    vec3 La;            // Ambient light intensity
    vec3 Ld;            // Diffuse light intensity
    vec3 Ls;            // Specular light intensity
} lights[3];
```

Multiple point lights

- We can optimise the code by using one light intensity for diffuse and specular

- When we set the uniforms in the code, we use the index value to access individual lights

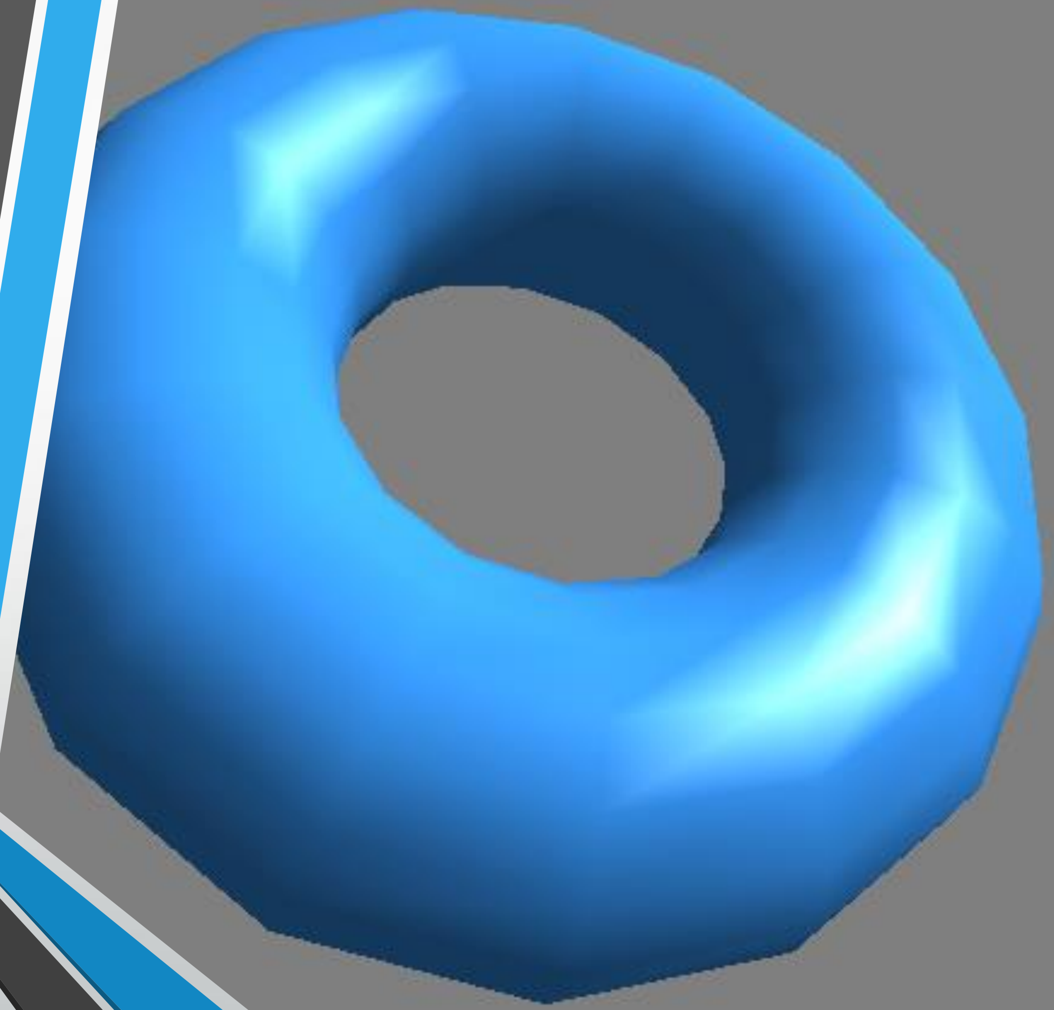
```
prog.setUniform("lights[0].Ld", glm::vec3(0.0f,0.8f,0.8f) );
prog.setUniform("lights[0].La", glm::vec3(0.0f,0.2f,0.2f) );
prog.setUniform("lights[0].Position", position);
```

Per-fragment shading

Point light

Per-Fragment Shading

- Colour evaluation in the vertex shader (**Gouraud shading**) is an interpolation of the values between adjacent vertices
- The result is an approximation of the light's interaction with the object' surface.
- Sometimes we get some undesirable effects, especially for specular calculations



Per-Fragment Shading

- We can improve the look even with a limited number of polygons in the mesh by doing all the calculations in the fragment shader
- We interpolate the position and the normal vector and we do the colour calculation for each fragment (**Phong shading** or **Phong Interpolation**).
- It’s not perfect but you will see a clear difference, especially on the specular highlight
- Downside - it is more expensive as we evaluate colour pre-pixel rather than per-vertex

Per-Fragment Shading Implementation

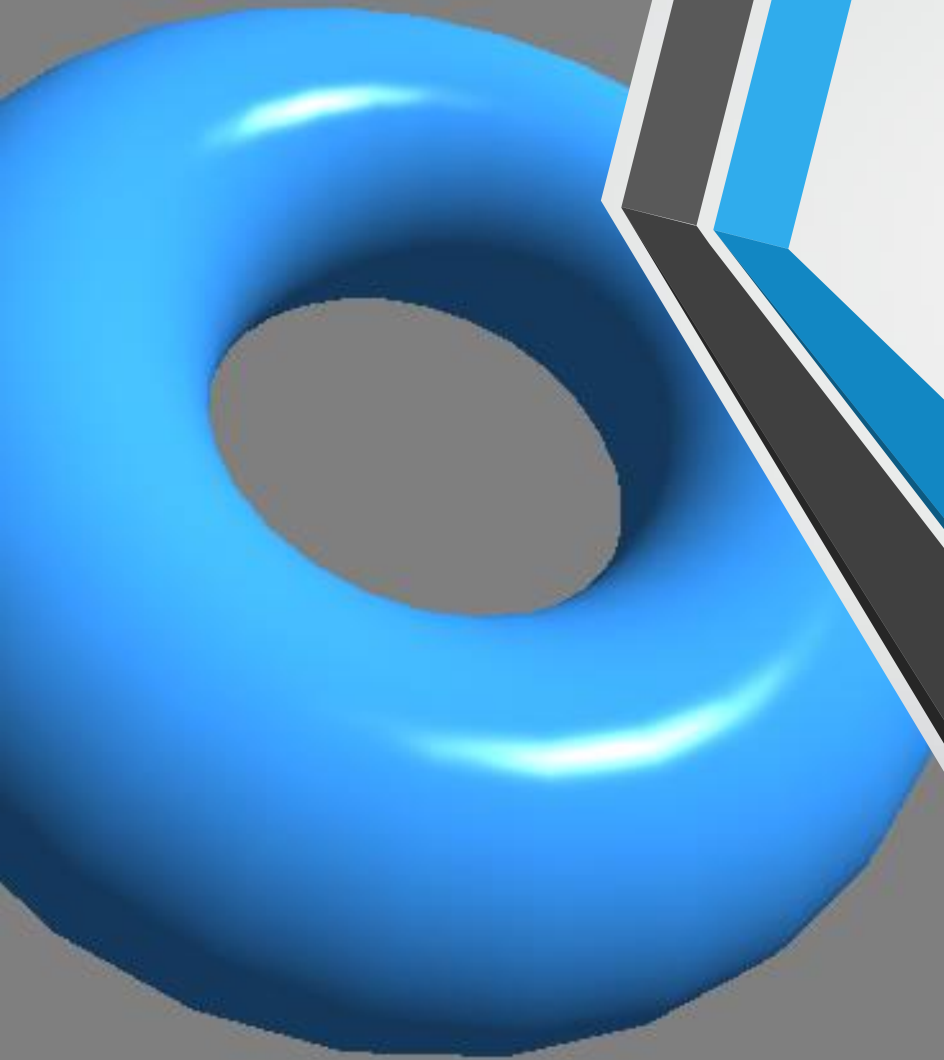
Vertex shader

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

//declare your uniform variables here

void main()
{
	Normal = normalize( NormalMatrix * VertexNormal);
	Position = ( ModelViewMatrix * vec4(VertexPosition,1.0) ).xyz;
	gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

Per-Fragment Shading Implementation

Fragment shader

```
in vec3 Position;
in vec3 Normal;
//declare your uniform variables here

layout( location = 0 ) out vec4 FragColor;


vec3 phongModel( vec3 position, vec3 n ) {
  // Compute and return Phong reflection model
}


void main()
{
        FragColor = vec4(phongModel(Position, normalize(Normal)), 1);
}
```



Per-Fragment Shading

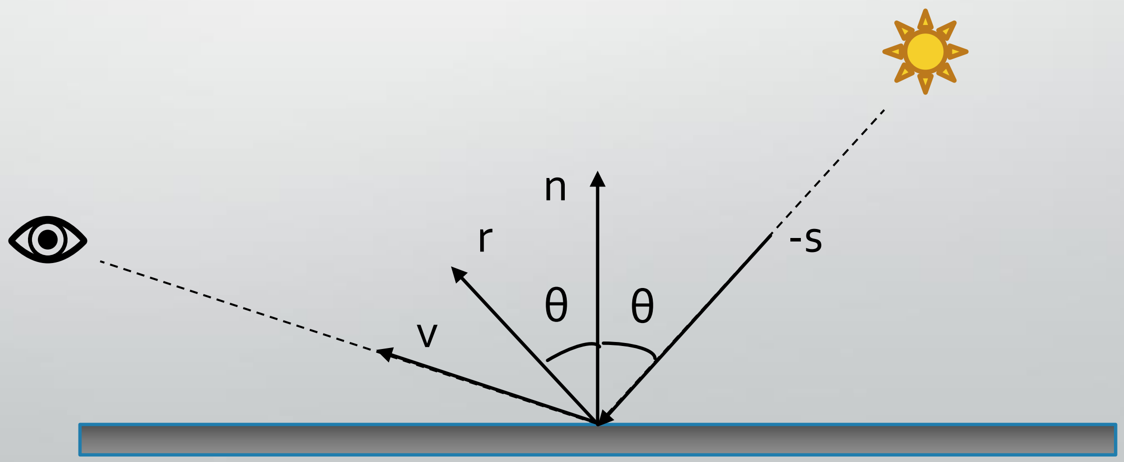
The Blinn-Phong

reflection model

Blinn-Phong

- Phong shading requires calculation of the reflection vector (r) and we used the dot product between r and direction towards viewer (v) to get the specular value

$$I_s = K_s \cdot L_s \cdot (r.v)^f$$



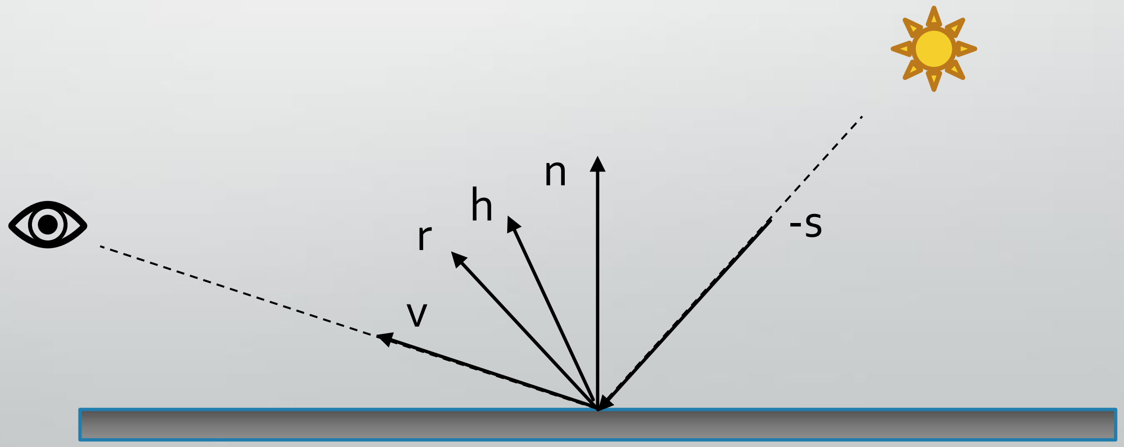
Blinn-Phong

- We can calculate a half vector (h):

  h = normalized(v + s);

- We can replace the $(r.v)$ dot product with $(h.n)$ dot product

  $I_s = K_s \cdot L_s \cdot (h.n)^f$



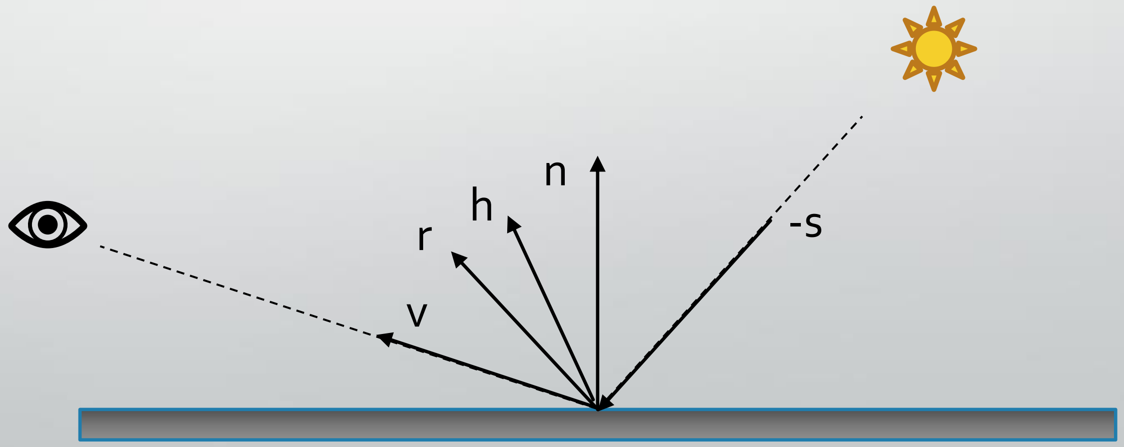
Blinn-Phong

- Computing h requires less operations, more efficient shader:

$$r = -s + 2(s \cdot n) \cdot n; \qquad h = v + s;$$

- We can replace the $(r.v)$ dot product with $(h.n)$ dot product

$$I_s = K_s \cdot L_s \cdot (h.n)^f$$



Blinn-Phong Implementation

Vertex shader

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

//declare your uniform variables here

void main()
{
	Normal = normalize( NormalMatrix * VertexNormal);
	Position = ( ModelViewMatrix * vec4(VertexPosition,1.0) ).xyz;
	gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

Blinn-Phong Implementation

Fragment shader

```
in vec3 Position;
in vec3 Normal;
//declare your uniform variables here

layout( location = 0 ) out vec4 FragColor;


vec3 blinnPhong( vec3 position, vec3 n ) {
  // Compute and return Phong reflection model
}


void main()
{
        FragColor = vec4(blinnPhong(Position, normalize(Normal)), 1);
}
```

Blinn-Phong Implementation

blinnPhong

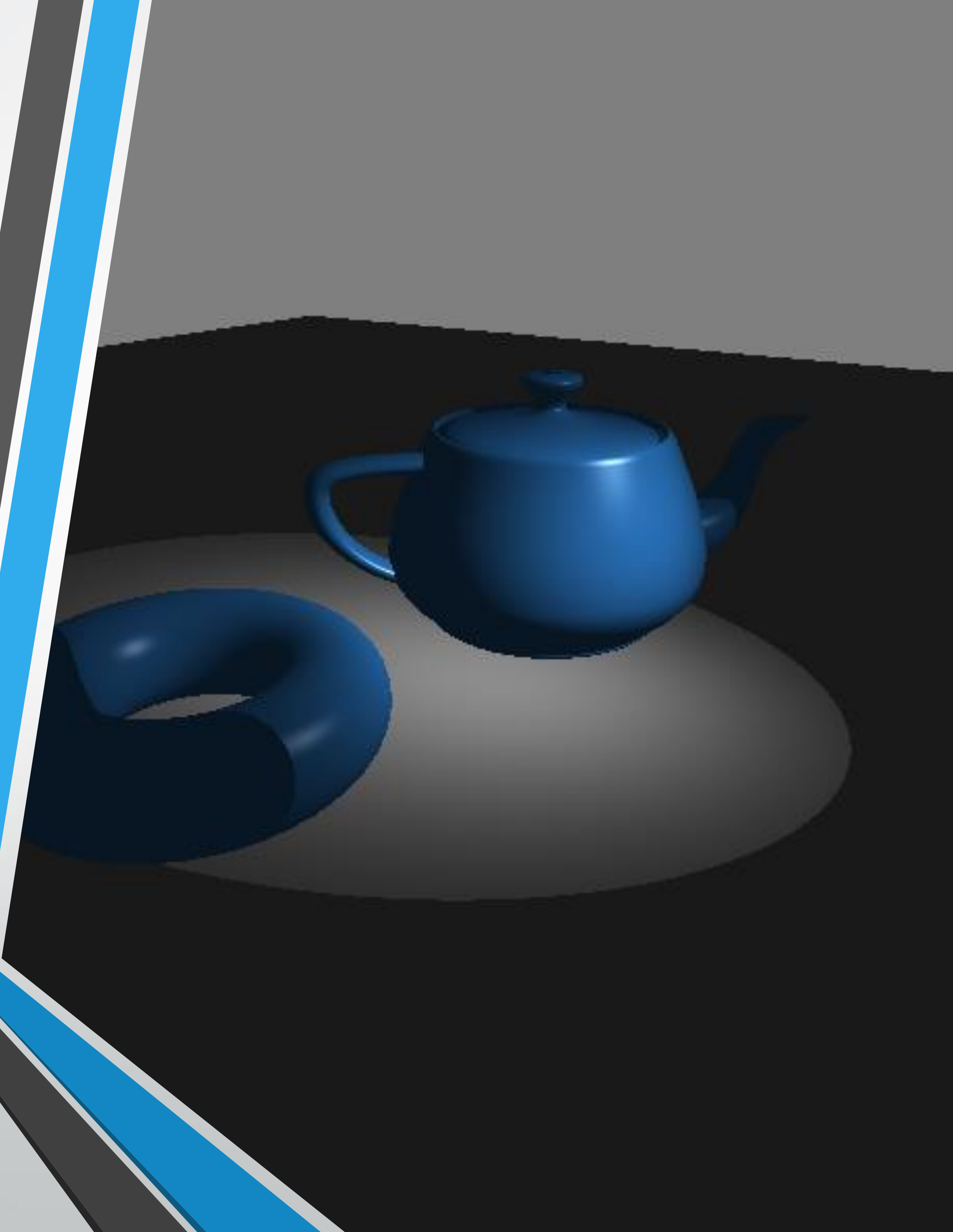
```
vec3 blinnPhong( vec3 position, vec3 n )
{
	vec3 ambient = ... ;//calculate ambient
	vec3 s = ... ;		//calculate s vector
	float sDotN = ... ;		//calculate dot product between s and n
				//(hint: use max)

	vec3 diffuse = Material.Kd * sDotN; //calculate diffuse
	vec3 spec = vec3(0.0);

	if( sDotN > 0.0 )
	{
		vec3 v = normalize(-position.xyz);
		vec3 h = normalize( v + s );
		spec = Material.Ks * pow( max( dot(h,n), 0.0 ), Material.Shininess );
	}

	return ambient + Light.L * (diffuse + spec);
}
```

Spotlight

light

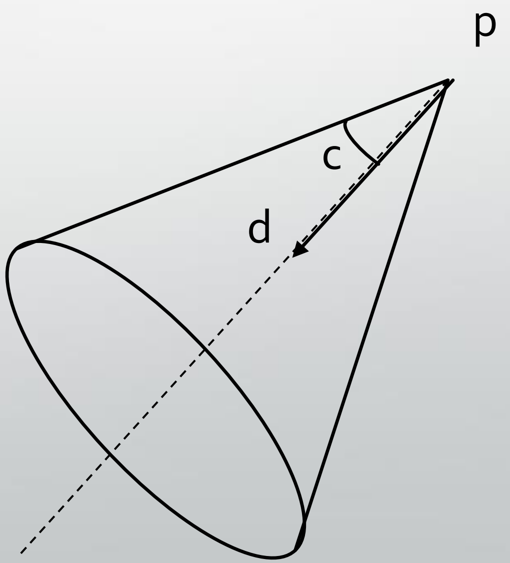
Spotlight

- Similar to a point light but the light radiates within a cone with the apex located at the light source
- Light is maximal along the axis of the cone and decreased toward the outside edges
- Creates similar visual effects to a real spotlight



Spotlight

- d: spotlight direction
- c: cut-off angle
- p: position of the light



Spotlight Implementation

Fragment shader

```
in vec3 Position;
in vec3 Normal;

uniform struct SpotLightInfo {
    vec3 Position;          // Position in cam coords
    vec3 L;                 // Diffuse/spec intensity
    vec3 La;                // Amb intensity
    vec3 Direction;          // Direction of the spotlight in cam coords.
    float Exponent;         // Angular attenuation exponent
    float Cutoff;           // Cutoff angle (between 0 and pi/2)
} Spot;

//declare your Material uniform variables here

layout( location = 0 ) out vec4 FragColor;

vec3 blinnPhongSpot( vec3 position, vec3 n ) {
   // Compute and return Phong reflection model
}

void main()
{
        FragColor = vec4(blinnPhongSpot(Position, normalize(Normal)), 1);
}
```

Spotlight Implementation

blinnPhongSpot

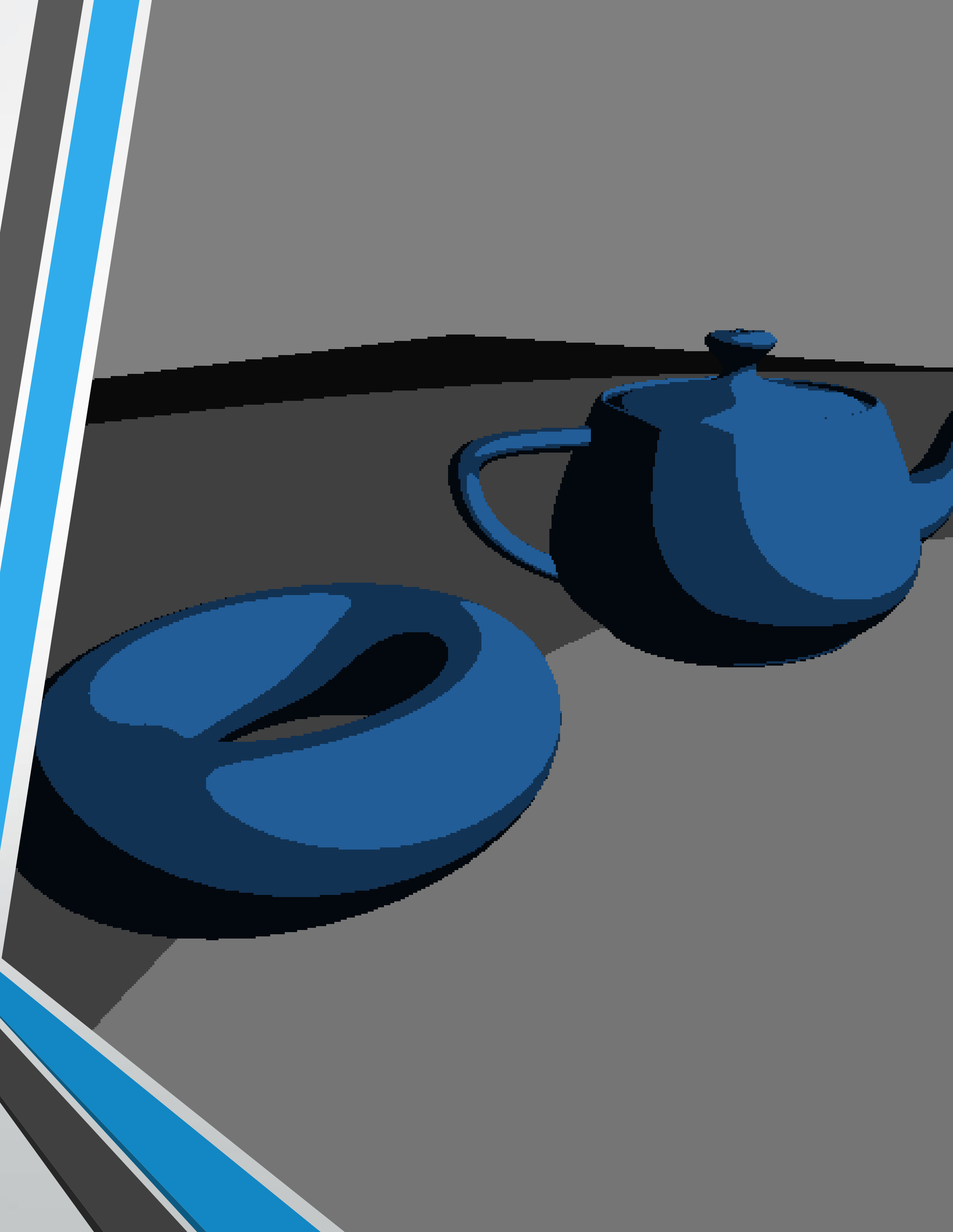
```
vec3 blinnPhongSpot( vec3 position, vec3 n )
{
        vec3 ambient = ... ;        //calculate ambient
        vec3 s = ... ;              //calculate s vector
         float cosAng = dot(-s, normalize(Spot.Direction)); //cosine of the angle
        float angle = acos( cosAng );  //gives you the actual angle
        float spotScale = 0.0;
         if(angle < Spot.Cutoff )
        {
                spotScale = pow( cosAng, Spot.Exponent );
                float sDotN = ... ; //calculate dot product between s and n
                diffuse = ... ;     //calculate the diffues
                if( sDotN > 0.0 )
                {
                        //calculate the specular
                }
        }
        return ambient + spotScale * Spot.L * (diffuse + spec);
}
```

Toon shading

visual effects

Toon shading

- **Toon shading** (also called **cel shading**) is a non-photorealistic rendering technique that is intended to mimic the style of shading often used in hand-drawn animation
- The basic effect is to have large areas of constant colour with sharp transitions between them
- Simulates the strokes of an artist pen or brush



Toon shading theory

- We lock the value of the dot product calculated for diffuse to a fixed number of possible values

- For a shader with 4 bands of colour for $(s.n)$ values we apply these bands:
  - $(s.n)$ = between 1 and 0.75 : 0.75
  - $(s.n)$ = between 0.75 and 0.5: 0.5
  - $(s.n)$ = between 0.5 and 0.25: 0.25
  - $(s.n)$ = between 0.25 and 0.0 0

Toon shading implementation

- We'll use only the ambient and the diffuse component in a fragment shader
- We need to declare 2 constants after your uniform material declaration

```
//uniform variables declaration
const int levels = 4;
const float scaleFactor = 1.0 / levels;
```

- After we calculate sDotN, we use this line to calculate the diffuse value:

```
//sDotN calculation
vec3 diffuse = Material.Kd * floor( sDotN * levels ) * scaleFactor;
```

Fog

visual effects

Fog simulation

- Fog is typically achieved by mixing the colour of each fragment with a constant fog colour
- The amount of "fog colour" applied is determined by the distance from the camera
- Traditionally a **linear interpolation** is used or a non-linear like **exponential** interpolation

Fog simulation formula

$$f = \frac{dmax - |z|}{dmax - dmin}$$

f: fog factor.

f=0 (means 100% fog)

f = 1 (means 0% fog)

dmax: the distance where the fog colour obscures all other colours in the scene

dmin: the distance from the eye where the fog is minimal

|z|: is the distance from the eye (camera)

Fog simulation implementation

- We'll use a Blinn-Phong implementation (fragment shader)
- You'll need a uniform struct called FogInfo:

```
uniform struct FogInfo
{
    float MaxDist;     //max distance
    float MinDist;     //min distance
    vec3 Color;        //colour of the fog
} Fog;
```

Fog simulation implementation

- For main implementation:

```
void main()
{
    float dist = abs( Position.z );  //distance calculations

    //fogFactor calculation based on the formula presented earlier
    float fogFactor = (Fog.MaxDist - dist) / (Fog.MaxDist - Fog.MinDist);

    fogFactor = clamp( fogFactor, 0.0, 1.0 ); //we clamp values

    //colour we receive from blinnPhong calculation
    vec3 shadeColor = blinnPhong(Position, normalize(Normal));

    //we assign a colour based on the fogFactor using mix
    vec3 color = mix( Fog.Color, shadeColor, fogFactor );
    FragColor = vec4(color, 1.0); //final colour
}
```


1
Get to the lighthouse
37

Useful links

- To read - Chapter 6 Simulating Light (OpenGL Superbible – see link on the DLE)
- Blinn-Phong reflection model: https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model
- Jim Blinn: https://en.wikipedia.org/wiki/Jim_Blinn
- To read - Light casters: https://learnopengl.com/Lighting/Light-casters
- To read: Lighting and shading in OpenGL 4 Shading Language Cookbook
- To read – Multiple lights: https://learnopengl.com/Lighting/Multiple-lights
- acos in GLSL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/acos.xhtml
- floor in GLSL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/floor.xhtml
- mix in GLSL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mix.xhtml