Tutorial 10 Shading

Although not explicitly discussed, the shading model we have used for the previous tutorials was actually Gouraud shading, where reflection calculation is performed on the vertices of a model. In this tutorial, we use Phong shading model (in combination with Phong lighting model). The Phong shading provides more accurate lighting (see the highlight on the floor in Figure B), but it is more computationally involved than Gouraud shading because the lighting model must be evaluated on every fragment.

Figure A Gouraud Shading

Figure B Phong Shading

The lighting calculation for Gouraud shading is necessarily done in the vertex shader, because it is place where the per-vertex operations are done. For Phong shading, the lighting calculation must be done in the fragment shader (hence the name of *fragment-shading*).

In vertex shader, the vertex coordinates and normals are still pass through via the attribute variables from the WebGL buffer objects.

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
```

The `aVertexPosition` is transformed into the camera coordinate system by applying the modelview matrix, `uMVMatrix`, to get `vertexPositionEye4` (in homogeneous coordinates). It is then converted back from the homogeneous coordinate back to the usually coordinate.

```
vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
                          vertexPositionEye4.w;
```

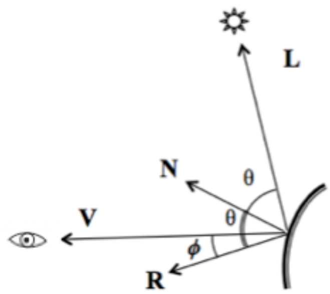
The normal, `aVertexNormal`, presented in the local coordinate system of the object, is transformed into a vector in the camera coordinate system by applying the inverse of modelview matrix, `uNMatrix`. During this transformation, the size of the vector has changed once it is expressed in terms of the camera coordinate and is no long a unit vector; therefore it has to be normalised (i.e., making it a unit vector) :

```
vNormalEye = normalize(uNMatrix * aVertexNormal);
```

In the fragment shader, the interpolated fragment normal vector, **N**, is used for calculating the unit vectors that represent the directions of the incident and reflective rays, **L** and **R**, which vary from fragment to fragment.

$$I_{total} = c_a + \sum_{i=1}^{all\_lights} \left(c_d \cos\theta + c_s (\cos\phi)^{\alpha} \right)$$

$$= c_a + \sum_{i=1}^{all\_lights} \left(c_d \mathbf{L_i} \bullet \mathbf{N} + c_s (\mathbf{V} \bullet \mathbf{R})^{\alpha} \right)$$



Calculate the vector (**L**) to the light source (How had we do this last week?):

```
vec3 vectorToLightSource = normalize(uLightPosition -
                                     vPositionEye3);
```

Calculate **N** dot **L** for diffuse lighting

```
float diffuseLightWeighting = max(dot(vNormalEye,
                                     vectorToLightSource), 0.0);
```

Calculate the reflection vector (**R**) that is needed for specular light

```
vec3 reflectionVector = normalize(reflect(
                        -vectorToLightSource, vNormalEye));
```


Calculate the view vector (**V**) in eye coordinates as (0.0, 0.0, 0.0) - vPositionEye3

```
vec3 viewVectorEye = -normalize(vPositionEye3);
float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
float specularLightWeighting = pow(rdotv, shininess);
```


The Phong reflection model is evaluated afterwards and the resulted fragment colour are used to modulate (multiply) the texel colour for the fragment:


```
vec3 lightWeighting =
          uAmbientLightColor +
          uDiffuseLightColor * diffuseLightWeighting +
          uSpecularLightColor * specularLightWeighting;

vec4 texelColor = texture2D(uSampler, vTextureCoordinates);

gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb,
                    texelColor.a);
```

Exercise: Amend the shaders in the last week's tutorial so that they implement the Phong shading model. If you have not finished the last week's exercise yet, start your work from the unfinished program.

```
 . . .

<script id="shader-vs" type="x-shader/x-vertex">
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
attribute vec2 aTextureCoordinates;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;

varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;

void main() {
  // Get vertex position in eye coordinates and send to the fragment
  // shader
  vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
  vPositionEye3 = vertexPositionEye4.xyz / vertexPositionEye4.w;

  // Transform the normal to eye coordinates and send to fragment shader
  vNormalEye = normalize(uNMatrix * aVertexNormal);

  // Transform the geometry
  gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
  vTextureCoordinates = aTextureCoordinates;
}

</script>

<script id="shader-fs" type="x-shader/x-fragment">
precision mediump float;
varying vec2 vTextureCoordinates;
varying vec3 vNormalEye;
varying vec3 vPositionEye3;

uniform vec3 uLightPosition;
uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;
uniform sampler2D uSampler;

const float shininess = 64.0;

void main() {
  // Calculate the vector (L) to the light source
  vec3 vectorToLightSource = normalize(uLightPosition - vPositionEye3);

  // Calculate N dot L for diffuse lighting
  float diffuseLightWeighting = max(dot(vNormalEye,
                               vectorToLightSource), 0.0);
```

```
  // Calculate the reflection vector (R) that is needed for specular light
  vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                               vNormalEye));

  // Calculate view vector (V) in eye coordinates as
  // (0.0, 0.0, 0.0) - vPositionEye3
  vec3 viewVectorEye = -normalize(vPositionEye3);
  float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
  float specularLightWeighting = pow(rdotv, shininess);

  // Sum up all three reflection components
  vec3 lightWeighting = uAmbientLightColor +
  uDiffuseLightColor * diffuseLightWeighting +
  uSpecularLightColor * specularLightWeighting;

  // Sample the texture
  vec4 texelColor = texture2D(uSampler, vTextureCoordinates);

  // modulate texel color with lightweigthing and write as final color
  gl_FragColor = vec4(lightWeighting.rgb * texelColor.rgb, texelColor.a);
}
</script>

. . .
```