Exercise 6-a: Checking normals

Duplicate *Exercise 5-b* (see Figure 1). We will prepare the normals so that the shader receives them correctly before we perform some kind of illumination in the next exercise. The solution of this exercise should look similar to *Figure 2*.

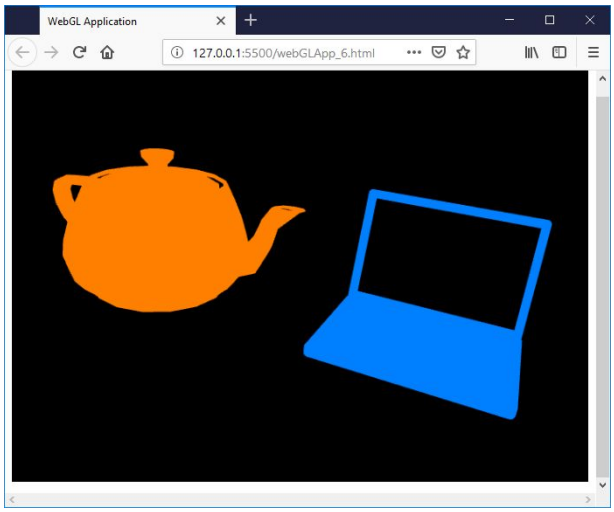

Figure 1. Solution of Exercise 5-b.

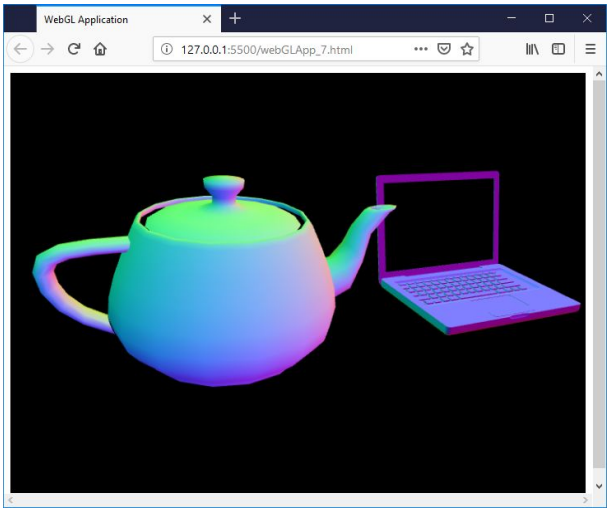

Figure 2. Surface normals shown as color.

Use the following simple vertex and fragment shaders and add the necessary modifications in the javascript code (mainly *initShaders*, *handleLoadedModel*, and *drawScene*).

Vertex shader:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

varying vec3 shadedColor;

void main(void) {
    shadedColor = aVertexNormal * 0.5 + vec3(0.5);
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

Fragment shader:

```
precision mediump float;

varying vec3 shadedColor;

void main(void) {
    gl_FragColor = vec4(shadedColor, 1.0);
}
```

Exercise 6-b: Gouraud shading

Duplicate the previous exercise to implement Gouraud shading. In Gouraud shading, we have to calculate the illumination at each vertex of the model (so, computations are done in the vertex shader) and the resulting color is passed to the fragment shader through a *varying* variable.

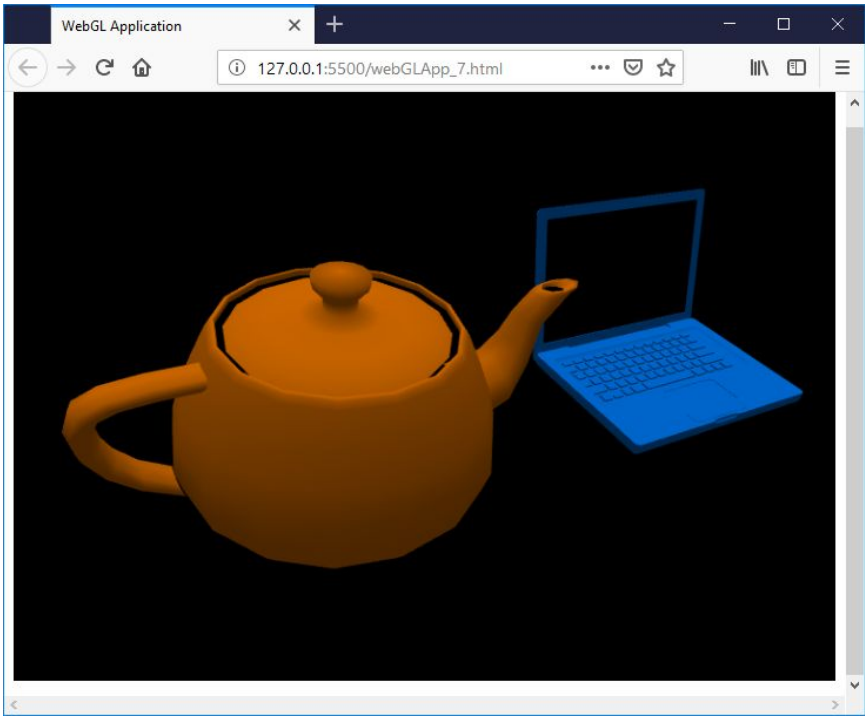

Figure 3. Gouraud shading has been applied to these models. The illumination was calculated per-vertex, so results are not so smooth as using Phong shading, but they are probably more efficient.

In the vertex shader (see below) you will notice that there are some new inputs introduced:

- **uNormalMatrix:** special matrix tailored to transform normal vectors without having them scaled improperly.
- **L, Ia, Ip:** Direction of the light and light intensities for the ambient and the diffuse term, respectively.
- **Ka, Kd:** Material absorption coefficients (material color) for the ambient and the diffuse term, respectively.

All these new variables are passed from the javascript code so they can be configured as desired for different objects and lights. Make sure you **obtain the new uniform locations in *initShaders()*** and you **give them proper values in *drawScene()***. On the other hand, ***uNormalMatrix* will be set and send in *sendMatricesToShader()*** (take a look at the slides for more info).

Vertex shader:

The vertex shader is computing a part of the Phong reflection model for the processed vertex. Notice that specular reflection is not taken into account here, only the ambient and the diffuse terms are being calculated, so there will be no shiny reflections.

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

// Matrices
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNormalMatrix;

// Light
uniform vec3 L;
uniform vec3 Ia;
uniform vec3 Ip;

// Material
uniform vec3 Ka;
uniform vec3 Kd;

varying vec3 shadedColor;

void main(void) {
    vec3 N = normalize(uNormalMatrix * aVertexNormal);
    float NdotL = max(dot(N, L), 0.0);
    shadedColor = Ka * Ia + Kd * Ip * NdotL;

    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

Fragment shader:

As you can see, in Gouraud shading, the fragment shader simply outputs the colour calculated in the vertex shader, as the illumination computations were done in the vertex shader.

```
precision mediump float;

varying vec3 shadedColor;

void main(void) {
    gl_FragColor = vec4(shadedColor, 1.0);
}
```

Exercise 6-c: Phong shading

Duplicate the previous exercise to implement Phong shading. In Phong shading, we have to send the vertex normals from the vertex shader to the fragment shader (using a *varying* variable), and perform the illumination computations at each fragment (so, in the fragment shader). The results exhibit specular reflections as seen in *Figure 4*.

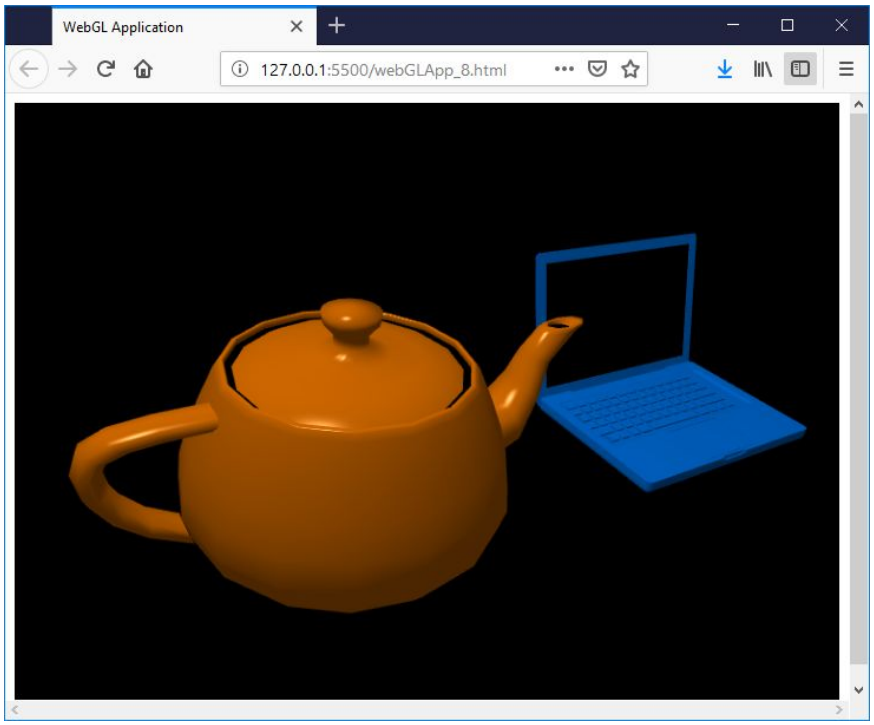

Figure 4. Phong shading has been applied to these models. The illumination was calculated per-fragment. The tree illumination terms seen at class have been applied (ambient + diffuse + specular). Look how the specular term generates sharp reflections.

You will see that the vertex shader now performs less calculations, only the transformed surface normal and position are converted to eye space (or camera) coordinates, and passed to the fragment shader.

The fragment shader, receives the light and the material parameters as uniform variables from the application, and using this information, and the surface normal and position coming from the vertex shader, performs the illumination calculations to compute the final shaded pixel color.

See below for the full vertex and fragment shaders.

Vertex shader:

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

// Matrices
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNormalMatrix;

varying vec3 normalEye;
varying vec4 positionEye;

void main(void) {
    normalEye = normalize(uNormalMatrix * aVertexNormal);
    positionEye = uMVMatrix * vec4(aVertexPosition, 1.0);

    gl_Position = uPMatrix * positionEye;
}
```

Fragment shader:

```
precision mediump float;

uniform vec3 Ia, Id, Is; // Light color
uniform vec3 Ka, Kd, Ks; // Material color
uniform float shininess; // Material shininess

varying vec3 normalEye;
varying vec4 positionEye;

const vec3 lightPositionEye = vec3(-1.0, 3.0, 0.0);

void main(void) {
    vec3 L = normalize(lightPositionEye - positionEye.xyz);
    vec3 N = normalize(normalEye);
    vec3 V = normalize(-positionEye.xyz);

    float NdotL = max(dot(N, L), 0.0);

    vec3 R = reflect(-L, N);
    float RdotV = pow(max(dot(R, V), 0.0), shininess);

    vec3 shadedColor = Ka * Ia + Kd * Id * NdotL + Ks * Is * RdotV;
    gl_FragColor = vec4(shadedColor, 1.0);
}
```