Interactive Graphics - Homework 1

Federica Cocci - ID 1802435

I have tested my code in Chrome, Firefox and Microsoft Edge: in all of these three the code works.

Request 1

I have replaced the cube with an object of 24 vertices. There are 4 square pyramids and 2 parallelepipeds: to draw a parallelepiped, I use the already implemented function "cube" that takes as input the 4 vertices of a face representing it as 2 triangles; to draw a pyramid I have needed a function which could draw only one triangle and for this reason I have implemented the "triangle" function.
Because my object is planar, I choose to compute normal vectors with the cross product between two differences of vertices and then I normalize it getting a vector of 3 dimensions. All the vertices have been passed with the right hand rule and have the same normal because the object is basically composed by triangles.
Since I am going to do texture mapping, I need to store also texture coordinates using a 2D vector where the range for each dimension is [0,1].

Request 2

I computed the barycenter of my object computing the arithmetic mean value for x, y and z values following the geometric definition of barycenter. Then, to make the object rotating around it, I used different transformations: a translation to the centroid, possible rotations along the three axis and a translation back to the origin point.

Request 3

The viewer position expresses the eye position (so the position of the camera) and it is computed using three parameters **radius**, **theta** and **phi** which can be modified from the user through sliders that vary in range [-180, 180] for the angles **theta** and **phi** and in range [1, 4] for the **radius** (the radius is the distance from the origin).
The perspective is computed using four parameters **fovy**, **aspect**, **near** and **far**. Also these parameters can vary using sliders.
About ModelView matrix, it is computed using the implemented function "lookAt" which takes as input **eye**, **at** and **up**. **Eye** depends from the values set up with the sliders as explained before; **at** and **up** are fixed values that I have set to

```
var at = vec3(0.0, 0.0, 0.0);
var up = vec3(0.0, 1.0, 0.0);
```

Request 4

The cylinder has inside three point lights. The material which makes the cylinder has the property showed in "request 5" and in addition has the emissive property. The emissive property gives the cylinder the appearance of emitting light.

```
var neonEmission = vec4(1.0, 0.5, 0.0, 1.0);
```

Because of the values in the emission vector, the cylinder appears orange and it is important to know that the emissive parameter affects only the cylinder's colour and doesn't affect the object's appearance.
The choice I made is deleting the pre-existing light and using only the three lights of the cylinder to illuminate the object. In other words when the neon is turned off then the object and the cylindrical neon itself look black because there are no other light source.
About the three point lights, because they are composing a unique neon, I have chosen to give to all of these three the same vectors for ambient, diffuse and specular properties while, of course, they have a different positions: they are disposed vertically in little range of y-coordinate.

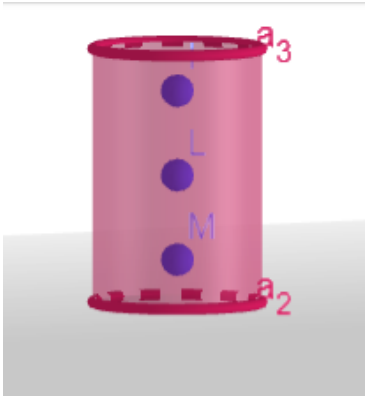

Figure 1: Cylindrical neon and its lights

After having introduced the cylinder in the scene, I decide to separate the modelViewMatrix: if this one till this request was the one calculated by the function lookAt and then multiplied to actuate different transformations (see request 2), now it is separated in **modelViewMatrixCyl** and **modelViewMatrix**. **ModelViewMatrixCyl** is the modelView computed by the lookAt function while **modelViewMatrix** is the one computed multiplying the **modeliewMatrixCyl** to apply transformations. I need to make this distinction because the cylinder doesn't have to rotate with the object when this one moves around its barycenter. In the shaders code when I calculate the position of the object I use **modelViewMatrix**, when I calculate the position of the cylinder I use **modelViewMatrixCyl**.
N.B. To draw the cylinder I have imported the code from geometry.js that is inside the code's folders of the book. In particular to put the lights inside the cylinder, I have change its values of radius, bottom center and top center.

Request 5

I have tested different materials by modifying the values of ambient, diffuse, specular reflectances and shininess. Because the color of the object depends from its properties but also

from the light hitting it, I made different test changing the light and material properties based on their interaction. My object actually appears orangish because its ambient reflectance and the ambient component of the light are set to red but the other properties have more green shadows. Summing up, I have chosen these values:

```
var materialAmbient = vec4(1.0,0.05,0.0,1.0);
var materialDiffuse = vec4(0.4,0.5,0.4,1.0);
var materialSpecular = vec4(1.0,1.0,1.0,1.0);
var materialShininess = 40.0;
```

All these values have been chosen in order to have a good react to the neon light and to the texture.

Request 6

Per-vertex shading implements the Gouraud shading method and with this shading the colour of the vertices is computed in the vertex shader so in input to the vertex shader I must have position and normal of each vertex and as output I have the colour; in the fragment shader I assign the already computed color to the fragments. The basic point of this shader is that all the computations are done in the vertex shader and then the rasterizer makes the computation of the color of the fragment by using the interpolation of the vertices colors.
Per-fragment shading implements the Phong shading method and here more information as output is sent to the fragment shader because, this time, the color is computed in the fragment shader.
To switch between per-fragment and per-vertex, I have used a flag, linked to the button, whose value is passed to the shaders where their implementations have if-else instructions. The button is referred to per-fragment shading so when its label is "switch-on per-fragment" means that in that moment per-vertex shading is activated.
Using per-vertex shading makes a big different from the point of view of GPU because in this case some computations are now computed in the vertex shader.
Thanks to made choices (about lights, materials, etc.), it is easy to notice the difference between per-vertex and per-fragment: per-fragment shading is more accurate and more realistic also because the object is more shining.

Request 7

To get a rough surface, I generate raw data (that I will use to compute the bump mapping) using integer numbers from 1 to 500 by applying the random function from Math library.
I decide to allow the application of the texture only if the user is working with per-fragment shading because bump mapping is done on a fragment-by-fragment basis: if the user is working with per-vertex shading and activates the texture then he will not see any changes to the object.
N.B. To compute bump data, bump map normals, normal texture array and to configure the texture I have used functions already implemented in bumpMap.js from the code of the book.