HOMEWORK 1 – INTERACTIVE GRAPHICS

Denise Landini – 1938388

NOTE: THE PROJECT WORKS ONLY IN GOOGLE CHROOME.

1. Replace the cube with a more complex and irregular geometry of 20 to 30 (maximum) vertices. Each vertex should have associated a normal (3 or 4 coordinates) and a texture coordinate (2 coordinates). Explain in the document how you chose the normal and texture coordinates.

   The **object** that I choose is a sort of closed jar. The jar is composed by twenty- vertices expressed in four homogenous coordinates (x, y, z, 1).
   In particular, the faces are composed by using both triangle and quad functions.

   Each **vertex** has a normal that is computed as the cross product of two vectors. To do this I subtract two pairs of vertices. I choose this type of computation because it is the classical way to compute the normal for a polygon.
   However, I use the right-hand rule to compose the faces of the jar to have the desired effect.

   The **texture** coordinates are used to identify points in the image to be mapped and each vector is composed by 2 coordinates as required. In particular, the vectors go from (0,0) to (1,1) and I choose to divide it into 2 parts:
   - The vectors from (0, 0) to (0.5, 0.5) are used to assign a texture for the jar;
   - the vectors from (0.6, 0.6) to (1, 1) are used to assign a texture for the cylinder. I choose to do this to assign a different texture for the jar and cylinder.

2. Compute the barycenter of your geometry and include the rotation of the object around the barycenter and along all three axes. Control with buttons/menus the axis and rotation, the direction, and the start/stop.

   To compute the **barycenter** of my geometry, I choose to apply the average of the positions of each vertex of my object, that are identifies with (x, y, z) coordinates. Because my object has a uniform density distribution.

3. Add the viewer position (your choice), a perspective projection (your choice of parameters) and compute the ModelView and Projection matrices in the Javascript application. The viewer position and viewing volume should be controllable with buttons, sliders, or menus. Please choose the initial parameters so that the object is clearly visible, and the object is completely contained in the viewing volume. By changing the parameters, you should be able to obtain situations where the object is partly or completely outside of the view volume.

   The table illustrate the intervals that I choose for the parameters and also the current value.

	Min	Max	Current value
Parameters used for viewer position:			
Radius: represents the distance from the origin.	1.0	4.0	4.0
Theta: used to change the camera view.	-180	180	10
Phi: used to change the camera view.	-180	180	-145
Parameters used for perspective projection:			
Fovy: how wide the eyes open along y.	10	50	45.0
Aspect	0.5	2.0	1.0
Near: the near plane distance from camera for the jar.	0.1	2.5	0.3
Far: the far plane distance from camera for the jar.	1.0	5.0	10.0

The advantage of using the two parameters theta and phi in that interval is to have a global view of the object (360 degree). The radius and all the other parameters are chosen to have the object clearly visible as required and the intervals are chosen to obtain a lor of situations in which the jar is partially or completely outside of the view volume.

The modelViewMatrix is computed by the *lookAt* function in which I passed the parameters **eye**, **at** and **up** that are 3-dimensional vectors. This function allows to position the camera.
In particular, I set:

```
at = vec3(0.0, 0.5, 0.0)
up = vec3(0.0, 1.0, 0.0)
```

The projectionMatrix is computed with the *perspective* function in which I passed the parameters **fovy**, **aspect**, **near**, **far** with the value explained in the table before.

For all the values of viewer position and perspective projection I implemented, in the html file, the sliders to change them.

4. **Add a cylindrical neon light, model it with 3 light sources inside of it and emissive properties of the cylinder. The cylinder is approximated by triangles. Assign to each light source all the necessary parameters (your choice). The neon light should also be inside the viewing volume with the initial parameters. Add a button that turns the light on and off.**

   I decide to approximate the cylinder using an octagon. In details, my cylinder is composed by eighteen vertices and thirty-two faces and, to compose it, I decide to use only the triangle function. To give the emissive property to the cylinder I created an **emissionsArray** and for each vertex I add to this array a value for the emission.
   The vertex of the cylinder has emission given by vector `vec3(3.0, 3.0, 3.0)`.
   The vertex of the jar has emission given by the vector `vec3(1.0, 1.0, 1.0)`.

   For each of the three light sources I set an RGBA for **lightAmbient**, **lightDiffuse**, **lightSpecular** and I have a **lightPosition** that characterize the position of the light sources.
   I decide to put in those vectors the same parameters for all the three lights in the cylinder to obtain an homogenous effect.
   In particular I choose:

```
lightAmbient = vec4(0.2, 0.0, 0.2, 1.0)
lightDiffuse = vec4(0.9, 0.9, 0.9, 1.0)
lightSpecular = vec4(0.5, 0.5, 0.5, 1.0)
```

   In the html, I add the button that allow me to turn on and turn off the lights. Obviously, when the three light are on or off at the same time.

5. **Assign to the object a material with the relevant properties (your choice).**

   The properties of the material affect the colors of the object besides of the color of the lights. All the three light sources influence the material.
   In order to assign a material to the object I used for variables with the following parameters:

```
materialAmbient = vec4(0.8, 0.3, 0.4, 1.0)
materialDiffuse = vec4(0.2, 0.2, 0.2, 1.0)
materialSpecular = vec4(0.2, 0.2, 0.2, 1.0)
materialShininess = 1.0
```

   I choose this setting also to have the desired color and the desired material of my jar because the color of my object depends on the lights but also on the material that I choose.

6. Implement both per-vertex and per-fragment shading models. Use a button to switch between them.

Figure 1: Per vertex

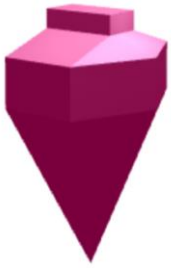

Figure 2: Per fragment

Per vertex and per fragment are used to compute color and lighting for polygons.
In per vertex, I compute the color for each vertex. The computation is done in vertex shader, so for every vertex we process it, and I also compute the color for the vertex.
In per fragment, we compute the color for each fragment and the computation is done in fragment shader.
So, the difference is not the equation that is the same, but the difference is where I use it.

As we can see in the two figures, with per fragment we have the upper surface that is lighter than which with per vertex.

In the html file I implemented a switch that allow me to change from per vertex to per fragment.

7. Create a procedural normal map that gives the appearance of a very rough surface. Attach the bump texture to the geometry you defined in point 1. Add a button that activates/deactivates the texture.

To create the rough surface, I created a texture by using random numbers between 0 and 1. I choose this method because it is very simple, and the result is good.
To use in a correctly way the bump map, I use also the tangent.
Furthermore, I add the bump texture only in the per fragment and not in the per vertex.

In the html file I implemented a switch that allow me to enable the texture only when I set per fragment.