Interactive Graphics
Homework 1

Lorenzo Nicoletti - 1797464

April 27, 2021

Remark: the code has been tested only on Mozilla Firefox and on Microsoft Edge, as already written in the private comment on Google Classroom.

Exercise 1

The proposed geometry is composed by twenty-four vertices, where each of them has associated 4D position coordinates, 3D normal coordinates and 2D texture coordinates. Since all the faces of the geometry are created with quadrilaterals, the normals are directly computed as the cross product of the two segments composing the quadrilateral. Then for each face, the same normal is associated to all the four vertices forming the corresponding surface. On the other hand, the assignment of the texture coordinates is straightforward: given a surface composed by four vertices, this set of coordinates is assigned to the points according to their position in the quadrilateral, therefore, the bottom-left vertex has (0, 0) coordinates, the bottom-right (0, 1), the top-right (1, 1) and the top-left (1, 0).

Exercise 2

The barycenter is computed considering a simple approximation about the geometry: this is an almost-symmetric polygon where the trapezoid on the righter side is greater than the specular one on the lefter side, making the barycenter moving only along the x-axis. Here, I have simply computed it by averaging all the vertices coordinates, without taking into account the mass density and the volume of the figure. This is a reasonable approximation for very simple architectures like the proposed one, but it may become unfeasible for more complex objects. The rotation around the barycenter is derived as explained in the theory with $M = T(p_f) \cdot R(\theta) \cdot T(-p_f)$, where p_f is the barycenter and $T(\cdot)$ is the translation matrix. Moreover, there is a button to switch the center of the rotation from the barycenter to the origin of the object frame. Finally, the resulting ModelViewMatrix is sent to the shader as a uniform variable.

Exercise 3

The parameters related to the perspective projection are chosen in order to render strong visual distortions when the geometry is rotating (especially along the y- and z-axes). This is done by

choosing a combination of viewer position, near and far parameters that locate the camera quite close to the object. The ModelViewMatrix is computed with the $lookAt(viewerPos, at, up)$ function (that will be later multiplied with the transformation matrices), while the projectionMatrix simply with $perspective(fovy, aspect, near, far)$, instead of the pre-existing $ortho(\cdot)$. These two matrices will be sent to the shaders as uniform to compute the varying positions over time.

Exercise 4

For this exercise, I have decided to use a new program with another pair of shaders to handle the cylinder in a 'closed' way to not affect the main program that has lots of cases and variables to handle differently. The cylindrical object is created by exploiting the already implemented function to build this solid, with some practical functions to scale, translate and rotate it at will. Its vertices are concatenated to the ones of the geometry in the same array, therefore, I have taken into account this distribution in the render function with two different $drawArrays$ calls.

The cylindrical neon light is located in the lefter side of the canvas, it is parallel to the yz-plane and has its center along the x-axis. It is modeled with three light sources located on the center and near the two bases of the solid. In particular, these lights share the same lighting properties in order to render a blue neon light that is uniformly spread from the cylinder. The operation is accomplished by instantiating other six tuples of three products (ambient, diffuse and specular), three for the geometry and three for the cylinder (that is supposed to be affected by the light too), and passing to the shaders the light positions and their shininesses. Moreover, the cylinder has also an emissive parameter that is summed up in the final computation of the lighting effect. When the neon light is switched on, the lefter side will be more illuminated than the righter one that is exposed only to the environment light. Since I do not want the cylinder to rotate, I have passed to the shaders a ModelViewMatrix that is not multiplied by the rotation matrices. Moreover, the cylinder material is 'pearl', one of the visual effects close to the glass, just to give a realistic representation of the lamp. The 'a' component of the color is set to 0.5 to make it translucent.

Exercise 5

The chosen material for the geometry is gold, whose properties are reported in the table below.

Ambient	Diffuse	Specular	Shininess
(0.24725, 0.1995, 0.0745)	(0.75164, 0.600648, 0.22648)	(0.628281, 0.555802, 0.366065)	0.4

All these values are used to obtain the products with the light properties to be sent to the shaders that use them to compute the components of the final color of the vertex/fragment (according to the chosen shading technique).

Exercise 6

I have decided to integrate both Per Vertex (PV) and Per Fragment (PF) shading techniques in the same pair of shaders in order to simplify the computation and reduce the number of

needed components. PV was already implemented, while the integration of PF has required a uniform boolean flag to switch between techniques. Since there are only two shaders, a vertex and a fragment, that handle the whole shading process, my approach has caused a bit of redundancy in their input and output variables: no matter which shading I am rendering, the shaders will have all the required inputs and outputs of both PV and PF. This is the drawback in keeping the number of shaders as low as possible.

In the shader routines, I need to compute the halfway, the light and the normal vectors for each light sources (that are four, one environment light and three neon lights) to be used to compute the components of the color. This final sum will be done in the vertex shader or in the fragment shader, according to the flag that handles the PV/PF switching.

When switching between approaches, the changes are better perceived if the geometry is not rotating: as expected, we can see that the shades on the surfaces facing the light becomes darker if PV is active, while they vanishes a bit with PF.

Exercise 7

Before discussing the implementation of this exercise, I need to point out an assumption about the bump texture: when it is applied on the geometry, shading and lighting options, namely PV/PF switching and neon lights, are disabled because of the incompatibility of these different tasks that I was not able to overcome. However, I have integrated the bump handling inside the main program with a boolean flag to switch among shader routines; in particular, when the texture is applied, the vertex shader receives a 3D tangent vector in addition. This vector is created in the js file by simply taking one of the two segments composing the quadrilateral and it is associated to all its four vertices. Then, the computation of the normal, the tangent and the binormal in eye coordinates is straightforward as the 'L' vector derivation to be passed to the fragment shader, that uses it together with the diffuse product of the material to calculate the final color.

Finally, I have created a random texture to make the surface as rough as possible. The 'bump' data are used to derive the normals to create the final texture, that is handled in the render function according to the flag's value.