Homework 1 - Interactive Graphics

Santilli Sofia - matricola 1813509

April 2021

1 Exercise 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.

In order to complete exercise 1, only the js file was modified. I replaced the cube with a 28 vertices-solid, almost centered in the center of the cartesian space. The position of these vertices in this coordinate system xyz can be found in the *variable vertices* defined in the js file. My solid is made up of 30 faces: 16 triangles and 14 quadrilaterals. By counting, they might seem only 28 faces, but the top and the bottom ones are the composition of 2 quadrilaterals. Each quadrilateral has been obtained by passing the four respective vertices to the *quad function* (they are the result of two triangles sharing a diagonal). Also the triangular faces have been built through the quad function: I imagined them as degenerate squares, where the third and forth vertices passed to the *quad function* are coincident. Finally, in order to obtain the *numPositions* variable, I multiplied by 4 the total number of faces.
Each vertex of the solid has associated a vector of 3 coordinates for the normal. A normal is computed for each face (through the cross product $(p_1 - p_0)x(p_2 - p_1)$, where p_i are 3 of the vertices that define a face) and then associated to every vertex belonging to that face. Vertices have to be passed to the *quad function* counterclockwise, so that the normal will be directed externally to the solid: in this way, the face will be colored properly.
Regarding texture coordinates, there are four possible texture coordinates: (0,0), (0,1), (1,1), (1,0). Therefore, each vertex of a quadrilateral has a couple (s, t) of these coordinates assigned to it. I also added the texture checkboard to the solid.

Figure 1: Texture: The first triangle shows the effect we should have when the texture coordinates are assigned in a right way, contrary to the second triangle. But in the solid created, as triangles are built as a degenerate quadrilateral, the texture coordinates are correct if they appear as in the third item of the image.

2 Exercise 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.

The solid built in point 1 is symmetric along the x and y axis. It is not symmetric along the z axis because of the two cubes that protrude from the front: they make the barycenter move forward (in the direction out of the canvas).

I computed the new barycenter of the solid by hand. I divided the main solid in basic solids (pyramids and parallelepipeds): considering one of them at the time, I multiplied its barycenter coordinates and its volume, obtaining the static moment. Summing all the static moments and dividing by the area of the total solid, I obtained the new coordinates of the barycenter: $(0.0, 0.0, -0.1167)$. It is recognizable how the barycenter shifts along the z axis, out of the canvas (the direction is negative because of the values assigned to *near* and *far* (see exercise 3)).

Once the barycenter is known, it is possible to compute the translation matrix: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.1167 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ with the barycenter vector in the forth column. Now it is enough, in the render function, to translate the main solid in the new baricenter (multiplying the translation matrix by the *modelViewMatrix*), apply the three rotations around the x, y, z axis, and bring the system back by multiplying the *modelViewMatrix* by the inverse of the translation matrix.

3 Exercise 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.

Perspective projection doesn't preserve most of the original shapes (distances and angles) of the solid, but it makes the object look more realistic. The *viewerPos* is the center of projection, in which the projectors converge: as it has been placed not so far from the origin of the coordinate system in which the solid is centered ($radius = 7$), the solid appears a bit deformed, due to this type of perspective. More we increase the radius, through the appropriate button, more the solid is further from the *viewerPos* and so the perspective projection looks similar to an orthogonal projection.
at and *up* are defined constant: *at=vec3(0.0, 0.0, 0.0)* means that the viewer is always looking in the direction of the origin; *up = vec3(0.0, 1.0, 0.0)* means that I always keep my camera perpendicular to the y axis.
near=-4.0 and *far=4.0* have been imposed in a way such that the object, along the z axis, is included between these two values.

4 Exercise 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.

In order to render two different objects, I used two different couples of shaders: *vertex-shader/fragment-shader* related to the main solid, *vertex-shader2/fragment-shader2* for the cylinder. In order to build the cylinder, I exploited the *function cylinder* (defined in the js file), that approximates a cylinder through triangles and that implements three functions (translate, scale, rotate) I used in order to establish the size and position of the cylinder in cartesian space. The cylinder is located below the solid, moved along the z axis in the direction out of the screen.
Buffers for positions (vertices), normals and texture weren't duplicated: so there is still one buffer for the positions, one for the normals, one for the texture. I pass to the *positions buffer* the concatenation of: the vector of the positions of the main solid and the vector of the positions of the cylinder. Same for the *normals buffer*. Regarding texture, I'm only interested in applying a texture to the main solid, so I don't need to concatenate or take into account the texture coordinates of the vertices of the cylinder.
Once the cylinder was built, other three new white lights were placed along the axis of the cylinder: they were built so that they have different positions in the space, but same ambient, diffuse and specular components. The original light we had until now remains and affects both the main solid and the cylinder. So, by default, only the original light affects the two objects: by clicking on the button *'switch three lights'*, the main solid is now completely illuminated (except for its top, because the neon light is placed at the bottom). Also the cylinder, constructed as an emissive material (the emissive term is added in the computation of the color of the vertex or of the fragment respectively in

the *vertex-shader2* and *fragment-shader2*), is affected by the presence of these three new lights, and becomes more white. Both the objects keep being affected also by the original light, that is always switched on.
In order to obtain this result, four triples of *ambientProduct, diffuseProduct* and *specularProduct* were necessary: the first describing the effect of the original light on the main solid, one for the effect of the neon on the main solid, one for the effect of the original light on the main solid and the last for the effect of the neon light on the cylinder.

5 Exercise 5

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

The material that characterizes the object is silver, obtained by imposing the following values [1]:

```
var materialAmbient = vec4(0.19225, 0.19225, 0.19225, 1.0);
var materialDiffuse = vec4(0.50754, 0.50754, 0.50754, 1.0);
var materialSpecular = vec4(0.508273, 0.508273, 0.508273, 1.0);
var materialShininess = 0.4*128;
```

The ambient and diffuse components define which color the surface reflects under lighting (the color of the object). The specular component is set to a medium-bright color, in order to set the color of the specular highlight on the surface.
The forth component of *materialAmbient, materialDiffuse, materialSpecular* adjusts the opacity of the object: it is set to 1.0 as a metal is an opaque object, that doesn't let the light go through itself, scattering much of the light that hits it.
The *Shininess coefficient* specifies the RGBA specular exponent of the material. It has been obtained by multiplying a number in the interval [0, 1] (here 0.4) by 128 (as only values in the range [0, 128] are accepted).

6 Exercise 6

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

Per-vertex and per-fragment Shading models consist in two different methods used in order to apply a lighting model to the surface. They exploit the same rendering equation, but, as recognizable from the code implemented, they do the main computations in the two different shaders.
Per-vertex (Gouraud) shading does the shading calculations in the vertex-shader: therefore, for each vertex, it is computed its color. These colors are passed to the rasterizer: in the fragment-shader, every fragment will receive its own color, by interpolation of the colors at the vertices.
Per-fragment (Phong) shading computes the color in the fragment-shader. In the case of this solid there are fewer vertices than fragments to be processed; therefore, per-vertex shading should be faster.
Although Phong requires more work, generally it is a bit more accurate and realistic. In fact, in per-vertex shading, the lighting is only evaluated at the vertices and interpolated along the primitive. So the tessellation of the surface will affect the quality of the shading: but that's not our case, for how the solid has been constructed.
Here, as the main solid material is silver (so it is a shiny object), keeping it frontal and switching (through the *'switch shading'* button) between the two shaders, the difference is easily recognizable: when activating the Phong shading, the light on the object becomes a bit more intense and occupies a larger area of the frontal face. The quality of the shading is not affected

7 Exercise 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.

By using per-fragment normals we can trick the lighting into believing a surface to be rough.
First of all, by exploiting the texture coordinates chosen in exercise 1, we assign a new texture to our solid (and we create a button *'switch rough surface'* to switch between the chessboard and the bump texture). In particular, the texture implemented in this exercise has a particular feature: it is based on an array of zeros and ones. This array

[1] http://devernay.free.fr/cours/opengl/materials.html

tells us where the displacement will take place: where zero and one data are side by side.
In fact bump mapping, in order to make a rendered surface look more realistic, simulates small displacements of the surface: but only the surface normal, and not the surface geometry, is modified as if the surface had been displaced. In order to compute these displacements, normals and tangents to the faces of the solid are used. As a tangent to a face, I decided to use a side of the face, *t1*, which was already computed in the *quad function*. So *t1* is associated to each vertex and a *tangent array* (that later we'll be passed) to the shaders is computed. Then the shaders will handle the light in order to give a bump effect to the texture on the solid.
The code in order to accomplish this task has been implemented as described above, however no bump effect resulted.