Interactive Graphics 1^{st} homework

Fabrizio Casadei - 1952529

March-April 2021

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 this exercise, we have defined the vertices to construct the irregular geometry. Has been modelled a polyhedron made up of 30 vertices, making it is symmetrical with respect to all the axes. Moreover, to perturb this symmetry has been added the possibility of adding 2 more vertices (used also as a test for the second exercise), the two configurations are shown below. each vertex has been expressed as a vector $(x, y, z, 1)$, and for the 2 cases, we have reached respectively 28 and 30 faces.

For the calculus of each vertex's normal, first using the pre-implemented function has been computed the normal related to each face, which comes out from the cross-product of 2 quad's edges (subtraction of vertices). Then for having a more realistic aspect using the shaders, at each vertex has been associated an average of the face normals which involve the specific vertex. This gives a "smoothed" representation of normals, reducing the sharp edges phenomena.

Then we have defined the texture coordinates, used to map points of a given texture on the geometry. Using the parametric coordinates U and V, we have associated to each vertex of all the quad a very simple mapping that binds each texture corner to a vertex. This has been done always considering that a quad is represented by 2 triangles. This texture will be used in the last exercise, however, to test this step has been defined a chess texture and has been applied to the model.

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.

For this task, we have figured out 2 ways of doing the barycenter, one with a higher level of appoximation and other which is a really good representation of the actual barycenter.

1. Barycenter as a centroid of finite number of vertices: $b = \frac{\sum_{i=1}^{n} x_i}{n}$, with n the number of vertices.

2. Barycenter as a weighted average of centroids. The weights are the areas of each face, and the centroids represent the barycenter of each face.
   $b = \frac{\sum_{i=1}^{q} C_i A_i}{\sum_{i=1}^{q} A_i}$, with q the number of quads.

The face centroid computed in the second manner, can be seen as the barycenter calculus done in the first method, only using the vertices of a face. The area as been approximated as the sum of the 2 triangles areas that made up the quad (this is an approximation whether we have points that are not on a same plane). Obviously these 2 methods used with the first configuration of the polyhedron return the origin coordinates, given the symmetry. While if we use the second configuration, we have an alteration of the y coordinates respect to the origin frame. Using the first method the y coordinates is translated of +0.06 while using the second of +0.12.
To apply the rotation around the barycenter, the *model-view matrix* it's first multiplied by the translation to the barycenter, then we apply the rotation, and finally we multiply the matrix to make it return to the origin. The prompts for the managing of rotations around the axes has been already implemented on the initial version of the homework.

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.

For applying these effects we can divide parameters to be handled into 2 sets: the perspective and eye parameters.

Starting from the perspective ones, we have defined: **Z near**, **Z far**, **FOVy** and the **aspect**. Z near and Z far model the distance from the camera to the frontal and posterior plane in the viewing volume, the FOVy represents the field of view on the y axis which modifies the volume size, and the aspect is the ratio between width and height of the volume. Passing these parameters to the function perspective we define the *Projection matrix*.

While the view parameters concern the motion of the camera position in polar coordinates using **Radius**, **Theta** and **Phi**. Radius represents the distance from the origin, theta and phi are 2 angles used to move the camera position, to have a complete 360°view of the object. these 3 values are used to compute the camera position (through cosine and sine periodic functions used with phi and theta).
Then we define the *at* and *up* coordinates, the first represents the position that we are looking at, the second instead is the orientation of the camera. Finally, we compute the related *Model-View matrix* using the function lookAt.

All these parameters are controlled using sliders in HTML file.

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.

To obtain the neon light effect on the polyhedron, first, has been defined the cylindrical geometry using a pre-implemented library found in the course code folder, called: "geometry.js" and then using the scale, translate and rotate methods has been positioned horizontally above the main figure. Subsequently Three point light sources has been specified inside the cylinder with the following parameters:

- Ambient term: **vec4(0.2, 0.2, 0.2, 1.0)**
- Diffuse term: **vec4(0.25, 0.25, 0.25, 1.0)**
- Specular term: **vec4(0.25, 0.25, 0.25, 1.0)**
- Ambient Emissive term **vec4(0.0, 0.08, 0.0, 1.0)**
- Diffuse-Specular Emissive term **vec4(0.0, 0.8, 0.0, 1.0)**

Has been used two different emissive terms in order of having a more realistic light effects. In the shaders this effect has been managed using a flag which specify if using the pre-defined directional light or this one. The light computation for the final colour is done in the similarly manner used for the directional, but this time we consider the direction from the surface to the light, and we sum the contribution of each light in the cylinder. The neon light can be activated and disabled using a specific button in the page user interface. Follows the result obtained.

Exercise 5

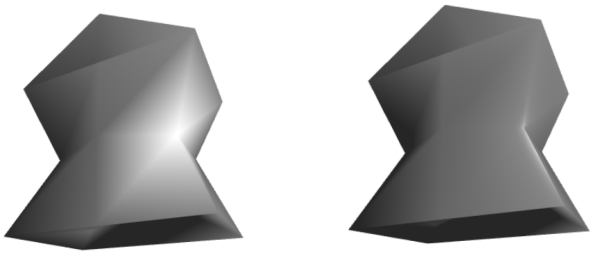
Assign to the object a material with the relevant properties (your choice).
The properties assigned for this task are the following ones:

- The colour of the polyhedron, established passing in the colour buffer, the same value for each vertex, which is a dark gray represented with this vector: **vec4(0.1,0.1,0.1,1.0)**
- The material properties of the polyhedron as a chrome object with ambient term **vec4(0.25, 0.25, 0.25, 1.0)**, diffuse term **vec4(0.4, 0.4, 0.4, 1.0)**, specular term **vec4(0.774597, 0.774597, 0.774597, 1.0)** and shininess equal to 60.

the ambient, diffuse and specular terms are multiplied by the light properties and then passed to the shader

Exercise 6

Implement both per-vertex and per-fragment shading models. Use a button to switch between them.
Everything until now has been done in per-vertex modality, so for compute this task has been defined a switcher for passing from per-vertex to per-fragment. Has been decide of not defining other shaders and use the already defined one adding the possibility of change the shading modality. Using a flag (just like in the exercise number 4), we switch between them. To implement the per-fragment shading, we just compute in the vertex shader the normal and the light direction data then in the fragment shader we compute the light calculation on a per-fragment basis, computing the contribution of the lights using interpolated values from the *rasterizer*. Making the color computation at the level of the fragment shader has led to the following results:



On the left the per-vertex shading while on the right the per-fragment shading

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.
Using the bump mapping in this exercise, we have tried to alter the normal vectors for the rendering process.
First has been defined the bump data using a the random number generator, and then based on those data has been defined the bump map normals and the consequent normal texture array. The bump mapping requires the process of each fragment independently, hence, this task must be processed at the level of the fragment shader. Therefore, we compute the normals in the fragment-shader using the bump texture and the texture coordinates assigned in the first exercise, in order of defining the diffuse term to be multiplied with the colour of each fragment.