Interactive Graphics - Homework 1

Engineering on Computer Science

Daniele Cantisani 1707633

May 2020

1

To satisfy the first request of the homework, initially I looked for figures with 20 to 30 vertices. Some of the thirteen Archimedeans solid satisfied the request. I personally chose the truncated cube. The idea was to build the peculiar cube and then modify its vertices to make it a more irregular figure.
The truncated cube has fourteen faces (six octagonal and eight triangular): to build it I used the Cartesian coordinates

$$(\pm\varepsilon, \pm 1, \pm 1),\ with\ \varepsilon = \sqrt{2} - 1$$

having for center the point $O = (0, 0, 0)$. All the permutations of the coordinates shape the figure with 24 vertices. The **vec4(x, y, z, 1.0)** structures were used to represent them.
The figure was build using the method **gl.drawArrays(gl.TRIANGLES, 0, numvertices)**. Each face was built through triangles; for this reason it was necessary to use six additional vertices: in fact, to build the six octagons through the use of triangles it was necessary to know the central point of each octagon. For this reason, although 24 vertices appear in the truncated cube, the total of the vertices declared in the code is 30.
To create the 14 faces I instantiated two functions, called

$$octagon(a, b, c, d, e, f, g, h, i, j); \qquad triangle(a, b, c);$$

Each octagon was built thinking of an octagonal cake divided into 8 triangles of equal size. The a parameter is the center point and the triangles were constructed one at a time. First a, b, c then a, c, d then a, d, e etc. Triangle a, i, j closes the octagon and obviously $j = b$.
After completing the composition of the truncated cube I modified the values of the vertices making the figure **irregular**. After the modification, the central points of four of the octagons also become independent vertexes of the figure, having been moved in space. The total of the vertices of the figure will therefore be **28**.
Along with the vertexes of the figure, I also declared two other types of vertexes: the normal ones and the texture ones.
As for the normal vertices, they are calculated each time one of the two functions is called (*octagon* or *triangle*). For the octagon I need to calculate the normal for each triangle. To compute the normal vector, I identified two vectors of the triangle analyzed, and I ran the cross product between the two vectors. Then i push the vector in its list $normalsArray$. Like the vertices vectors, I used the vec4 structure.
As for the texture vectors I decided to construct an octagon and a triangle in a plane with two coordinates (x, y). Then I used the vec2 structure constructing an octagon with 9 vectors (one center and 8 external vertices) and a triangle with 3 vectors.
I inserted the texture vectors in their list with a precise order during the execution of the two functions that build the shape: in this way when in the requirement **6)** I will insert the image for the texture, in each face of the figure there will be an image continuity with very few interruptions.

2

To fulfil the second point it was necessary to build two matrices. The first is called ModelViewMatrix and the second is ProjectionMatrix.
Initially, I retrieve the default uniform position in intermediate variables from the vertex shader in the html file. Then I call the two functions that define the position of the figure: **lookAt()** for ModelView and **perspective()** for Projection.
I used the **lookAt(eye, at , up)** function to compose our View matrix. This function takes three parameters into input. Imagining that there is a camera that represents the viewpoint: **up** indicates how the camera is rotated. To use the classic reference system I set the vector to $(0, 1, 0)$. The parameter **at** indicates where our camera is pointed (again for simplicity I used the default vector $(0, 0, 0)$). Finally, the most important parameter is **eye** which defines where our camera is located. This value depends on predetermined values such as the two viewing angles *theta* and *phi* and the distance radius of the camera, called *radius*.
To determine the perspective, I used the **perspective(fovy, aspect, near, far)** function that has 4 parameters: The **fovy** parameter is the vertical angle of the fictitious camera (I chose 42 degrees). The **aspect** ratio parameter is the width divided by the height of the canvas window. Finally the **near** and **far** parameters are the minimum and maximum values of the clipping plane where the figure is shown.
After computing the two matrices, I update the position with the new values by passing the changes to the vertex shader. In fact in the shader I perform the most important operation to getting the perspective and the final position of the figure:

gl_Position = uProjectionMatrix*uModelViewMatrix*aPosition;

To move the figure I introduced some sliders and two buttons: the buttons reduce or increase the **near** and **far** variable modifying the clipping plane in the $projectionMatrix$. The first slider reduce or increase the radius of distance from which the camera is located (**radius**) modifying the $ViewMatrix$. Finally there are two sliders that move on the two angles *theta* and *phi*. When the render() function is called, the **eye** parameter will be modified with the new values.

3 Light

3.1 Directional Light

To complete the request for a directional light I declared 3 vectors:

var lightPositiondirection = vec4(1.0, 0.0, 1.0, 0.0);
var lightAmbientdirection = vec4(0.1, 0.1, 0.1, 1.0);
var lightDiffusedirection = vec4(0.7, 0.7, 0.7, 1.0);

As for the specular variable, (**var lightSpeculardirection**), it has been removed because it was not necessary to realize the shading model required later (request **5**).
Then I multiplied the last three vectors with their material counterpart (see request **4**). (ex. **mult(lightAmbientdirection, materialAmbient);**).
So I passed the results of the three products and the light position to the fragment shader. Here, following the indications of the shading, I added the vectors values to the fragment color variable $fColor$, creating a light.
To move the position of the light, sliders have been inserted that modify the three position coordinates of the directional light (x, y, z).

3.2 SpotLight

The declaration of vector variables and multiplications with the counterpart of the material is performed in the same way as explained before for the direction light. The w parameter of the lightPositionSpotlight this time will be 1.0 as the spotlight is a point

lighting and not a directional one.
In addition, this light has two more parameters:

var lightDirection = vec4(0.0,0.0,12.0,1.0);
var limit=0.9949;

The first specifies the direction of the light reflector projection while the second changes the size of the circumference of the illuminated light cone. In fact, after passing the variables to the fragment shader, a very important control takes place here. I perform the scalar product between the direction of the spotlight (**uDirection**) and the position vector of the light(**ulightPositionspot**). In this way a circumference is created: when this value is bigger than the **limit** we are inside the cone, and so we have to light the fragment; the opposite otherwise.
As for the direction light, sliders have been inserted to control the position of the light and an additional one to control the size of the circumference (the **limit** variable).

4

The material chosen was ruby. It has a very bright red colour. The defined vectors are:

var materialAmbient = vec4(0.1745, 0.01175, 0.01175, 1.0);
var materialDiffuse = vec4(0.61, 0.041, 1.0);

Here, actually, we'd be missing the **var materialSpecular = vec4(0.72, 0.62, 1.0);** and **var materialShininess = 60.0;** which, however, are not used because of the use of a simplified toon shading (request **5**). The multiplication of the light vectors with these presented, produces a ruby colour.

5

The request is to modify the shading model making it a simplified version of the Cartoon Shading Model.
The model is characterized by two variables C_i and C_s:

$$C_i = a_g \times a_m + a_l \times a_m + d_l \times d_m \qquad C_i = a_g \times a_m + a_l \times a_m$$

We already know all these variables: $a_l \times a_m$ is the product between the two ambient variables (material_Ambient $\times$ lightAmbient) while $d_l \times d_m$ is the product between the two diffuse variables (material_Diffuse $\times$ lightDiffuse). Some examples below:

var ambientProductdir = mult(lightAmbientdirection, materialAmbient);
var diffuseProductdir = mult(lightDiffusedirection, materialDiffuse);

The only variable that had not yet been declared was a_g i.e. the global_ambient_light. I decided to represent it as a simple vec3 vector that I declared in the js file and passed to the shader along with all the other variables.
The shading works like this: I make the scalar product between the light position vector and the normal fragment vector. If this product is greater than or equal to 0.5 then I use the light C_i otherwise I use C_s.
The same method explained was used for both light sources changing the light variables..

6

To accomplish the last task it was necessary make use of some code from the examples of the book *Interactive Computer Graphics 7E* by Angel and Schreiner.
The image for the texture was found in an online photo stock. After adding it in the

html file, and taking the reference in the js file, I called the function **configureTexture()** passing the image as a parameter.
Finally, I passed to the shader the texture coordinates described in request **1** and the configured image.
At the end of the fragment shader I multiplied the texture to the fragment color value coming from the previous calculations.

fColor = (uglobalambient + totalambient + totaldiffuse)*(texture(Tex0, fTexCoord));