Interactive Graphics homework 1

Pietro Manganelli Conforti 1754825

1. The creation of the model
2. The rotation and projection
3. Light sources and material
4. Shading model and texture

1. The creation of the model

The model I have used for this project is a representation of a plane, using 30 vertices. It's created starting from the plane() function, which use the makeVolume(a, b, c, d, e, f, g, h) one for each time a parallelepipeds has to be done, where the parametres are the respective number of the vertices in the vertices list. For practical use, the letters used to design the figure are locally defined and associated to the index of the respective vertex in the vertices list. This function in turn calls the one which create a quadrilateral six times, one for each surfaces of the volume, and passes the relative vertices using the right hand rule. In the end the quad function pushes those vertices into their buffer, which is created in the init() function as an array of float, which will be passed to the vertex shader into the variable "aPosition". This function which create quadrilateral is also used to push the normals of the surfaces: each time is called a quadrilateral is created between the vertices a,b,c,d, using two triangles. Being those two on the same plane we can consider only the first one and evaluate the difference between its sides b-a and c-a. Next, we can do the normalized cross product of them and get a vector which is orthogonal to the plane where the two triangles lie. This will be a very useful information for the calculus of the lights, as explained later. This vector is pushed in the normal list as a 3-dimentional vector in correlation to the push of the vertices used to calculate it. The last information we need to push while creating a quadrilateral are the texture coordinates, defined as a generic quadrilateral in the texture vertices list and pushed accordingly to the position of the vertex in the space which is being pushed. All those informations are stored into their lists allocated and linked to their variable into the vertex shader by the init() function, together with the properties of the buffer (all those values are float, and vertices have 4-dimention, normal 3 and texture coordinates 2).

2. The rotation and projection

After allocating all the necessary resources we can pass them into the vertex shader. To transform their values in a meaningful way also different matrices are passed, for example the four-dimensional matrices *uModelViewMatrix* , *uInstanceMatrix*, *uProjectionMatrix*. Into the vertex shader the vertices are stored into our variable *aPostion* and after the transformation are assigned to the variable *gl_position*. Those three matrices are one for the viewer position, one for the rotation of the points and one for the projection of them. The information about the camera view is stored in the view matrix, formed through the *lookAt(eye,at,up)* function from the GLU library. Next the matrix of the instances is the one used to take in consideration the rotation of vectors. A

variable previously defined theta (*[0,0,0]* in the code*)*, keeps the values of the rotating angles for each axis and is used by rotateX(theta[0]), rotateY(theta[1]) ,rotateZ(theta[2]) to get the final rotating matrix Rxyz, passed as a matrix4vf to the vertex shader, where will participate to the evaluation of the resulting position of the vertices. In the end we have the projection matrix, which is done through the *ortho(left,right,bottom,top,near,far)* function and define the view volume, where near and far are considered as the distance from the camera and top right and left bottom are the respective vertices of the parallelepiped in consideration. The product of the modelViewMatrix and the ProjectionMatrix forms the NormalMatrix, used into the light formula.

Those effect can be turned on and off using the buttons on the right, which are linked to their flag in the code. The rotation takes place on one axis at time basing on the chosen one with the buttons, which increase its relative theta of 1.0 and change the final rotation matrix Rxyz. There is also the possibility to negate that values to change the rotation verse. The second buttons section concerns the projection parameters which, as already mentioned, can be used to tune the viewing volume and are near, far, radius, theta and phi. Those last two are useful when we want to rotate the eye of the camera but for example keeping the light effect unchanged. All these values are computed in a loop through the render function and initialized to set the object clearly visible.

3. Light sources and materials

The implemented light sources are two, a directional light and a spotlight. They are initialized with a group of vectors for: the position, fixed for the directional and modifiable for the spotlight, the diffuse values, and the ambient values. Those values will be passed to the shaders and combined with the material values, chosen to improve the final shade of the object. The spotlight respect to the directional one is restrained in order to have an oriented light-cone, which direction (x, y and z coordinates) can be regulated by the buttons as well as the cutoff region, which enlarge or reduce the lightened surface. The colour of these lights can be chosen by two pickers, which on change modify the value inside the respective diffuse variable of those two light sources. Then their new value is sent to the shader. Being the returned values by these pickers in hexadecimal, it is first converted to decimal (so in 0-255 range) and next divided by 255 to have the correct RGB format to create our four-dimensional vector. Moreover, the final ambiental light resulting from those two, can be turned off with a button to see the lights and the specular highlights better. Finally, the shininess coefficient of the material is set to 150, to simulate metals reflection.

4. Shading model and texture

The cartoon shade algorithm implemented for this homework calculate the texture coordinate at each vertex of the model using Max{L · n,0} , where L · n is the dot product between the normalized vector from the vertex to the light source position and n is the normal to the surface at the vertex. In the code, this check is applied on both the L of the spot and directional lights. If the product is more than 0.5 the formula for the fragment colour takes into consideration both the ambient light times its material coefficient plus the diffuse lights values times the respective coefficient, otherwise is considered only the resulting ambient value.

For what concerns the texture images I preferred to use three different images interchangeable with a button. They represent different patterns: military, metallic and marine, applied through the texture mapping method. They are configured by a *ConfigureTexture(image)* function in the .js file, which takes as parameter the pattern and define the property and filter modes of its application. Since the patterns have not a subject in the image and being the average surface a parallelepiped, the coordinates are chosen generic. The image information is interpolated to the colour one (from light sources) in the fragment shader, where a 2D sampler return a texture colour from a texture, and is possible to turn it on and off through a button.

In addition, I have used as background a cloudy sky and the plane can be controlled by w,a,s,d letters or by the arrows, which change the rotation axis and direction.