Name: Cecilia Aponte Matricola: 1822225
Interactive Graphics Homework 1 – Report

This first homework was given to make modifications to a cube by making it rotate, translate, scale; as well as see the object in different perspectives and shadings. This report shows the results of my implementation to solve the questions requested.

The first task was to change the direction of the current rotation. I started this thinking that the task was to change the axis at which the cube was rotating. So, I wrote the following code:
<document.getElementById("ButtonC").onclick = function(){axis = ((axis+=1)%3);}; //add 1 to axis and when reach 3, reset to 0>
It wasn't until I realized later during the homework that it was asking to change the direction from clockwise to counterclockwise. After I understood this I implemented a variable that each time it would be clicked and the cube would be moving, to rotate in positive direction by adding 2 to the angle of rotation in the current axis of rotation. If it would be clicked after and the cube is moving, it would rotate in a negative direction by subtracting 2 to the angle of rotation in the current axis of rotation. If the cube is not moving after clicking the Toggle Button (flag variable), then nothing will happen as the cube is not rotating. The advantage of this solution is that it is simple to run and works every time. However, a disadvantage is that if the Direction of Rotation is clicked when the cube is not rotating, the cube rotation will still be reversed as soon as the cube starts rotating again. It is advantageous if you want that to happen but is an issue if you did not.

The next task was to move the transformation matrices from the shader to the JS file. I had a lot of issues with this question, taking me almost a whole week to solve. The first problem was that I was not sure if I was supposed to use rx, ry, and rz or if I should change it to ModelView and Projection. So, I tried moving rx, ry, and rz to the JS file and after a lot of debugging it worked and I was ready to move to the next question. However, the words ModelView and Projection kept coming back to mind and feeling like I should do something and use those I decided to start again. I started by defining the world plane and dividing it into model and view, then I started creating the rotations. I soon realized that if I kept going it would take me too long. After doing some research, I discovered the MV.js file. This file made calling the functions much simpler! Therefore, I started defining the ModelView by calling the LookAt function that uses the eye, at, and up (each as a vector of 3 components x, y, and z) as their values. The ModelView is composed of two different types of viewing ways: Model and View. The Model matrix is the position in the scene, while the View matrix is the position in front of the camera. After being transformed by the Model matrix, you get the position and orientation of an object which is relative to the other objects in the scene. After being transformed by the View Matrix, you also get the position and orientation of an object but this time it is you retain to information of the relative location and orientation of the objects in the scene. The combination of this two matrices becomes ModelView and uses the LookAt to show where in the world space the we will be looking at. To decided where we must look at we have to pick where the camera will be located (eye), in what angle we will be looking at (up), and where on that space we are going to look at (at). The next matrix is the Projection matrix, which shows the position inside the clipping volume. The volume is defined by six locations for ortho: bottom, top, left, right, near and far; and four variables for perspective: fovy, aspect, near, and far which will all be described later. If these values are not picked right, no image or a partial image of the cube will appear. All these matrices are transferred to the vertex shader (by getting the location and of the flattened matrices and uniform-ing it for global use) to be multiplied to the position of the vertices and final position of the scene with the object that you will see in the screen. The advantage of defining the matrices like this is that it enables quicker access to change and correct what you view based on what you are looking for. Another advantage is that the rotation, translation and scaling can be applied directly to the ModelView in a few lines; this reduces the number of matrices that would have to be calculated later in the vertex shader. The disadvantage is that because ModelView are linked together, we must later get back the position and orientation of the camera space (View) in the vertex shader for later use of lighting.

Since the matrices were defined in a very simple way in the previous part, we can rotate, translate, and rotate the cube also in a simple way. In order to do so, we have to multiply the ModelView matrix with the rotation, translations and scaling matrices that we want. One detail to note is that there must be an order of matrices to multiply. This order should be as follows, first Scaling, followed by Translation and lastly by rotation. I learned by testing that if the order is not retained, you get a scaling that is off (scales only in one direction), a translation that is in the correct place but rotates around the initial and final locations, and lastly a rotation that moves incorrectly. To do these

activities, you must create the matrices and multiply them by ModelView. You create the scale matrix by using the function scale in MV.js, however I found too many issues in both their versions of the function, so I created my own. I multiplied the scaling factor by the diagonal of the ModelView matrix. For the translation matrix, I was going to also create my own method as I had issues previously, however I found it easier and simpler using the function already defined as translate and calling the same value for all directions x, y, and z. Lastly, we must rotate which can be done with the function rotateX, rotateY, and rotateZ, since I had issues using the other function rotate, using the angle theta at each axis. These variables scale_qty, translate_x, translate_y, and translate_z are initially given a value (0.5 – since both ortho and perspective projections look as a perfect cube in this size-, 0, 0, 0 - no translation in any axis-, respectively) but are given the option to be changed using a slider in the user's page. This can be done by creating the event and retrieving its value from the slider that has been set up with a minimum, maximum and step size for its use. I have not found any disadvantages with the method used, but there are multiple advantages starting with it's simplicity of coding and directly being able to apply it to the same matrix before sending it to the vertex shader.

The next task was to define the orthographic projection and using sliders to control the near and far planes. This type of projection shows an actual rendering of a model, mainly used in engineering fields for 3D drawing in Solidworks. It maintains the length and size of the objects without adding any information about its depth. Therefore, the model will not look natural to what would be seen in the real world with farther portions appearing smaller than those that are closer. To create the matrix, I called the function ortho and defined its parameters bottom, top, left, right, near and far. These parameters must be chosen correctly to view the complete object. Bottom/top and left/right are opposite value of each other (+value / -value) since the object in ortho is in the origin. If left is greater than right, the world is mirrored about the Y axis. If bottom is greater than top, the world is mirror about the X axis. If near and far values are given incorrect values (which are not intuitive to set), then you will get portions of the object clipped out and weird shapes. I experienced my cube rotating the opposite direction. This took we a while to fix, but I finally realized that I had placed near as a negative value and so the clipping depth was backwards. The values for near and far that I picked initially were 0.1 and 3, respectively. These near and far values can be later changed using a slider that has a minimum of 0.01 for near (since if this value it too high, you will not see the image) and 1 for far; a maximum of 3 for near and 5 for far with a step size of 0.1. The advantage of using this method is that it is simple and clear how to use it and requires only one line of code. The disadvantage is that it can be troublesome to find the correct values for these, especially for near and far as these are not intuitive and if not picked correctly your entire cube can disappear and you might think there is another issue with your code.

These advantages and disadvantages for ortho are the same for the perspective projection. This projection, however, is different from ortho in that there is depth to the image. This perspective projection is more realistic as it shows us how the object would look at dependent on the parameters chosen. These parameters are fovy, aspect, near and far. Fovy is the Field of View, this is the vertical angle of the camera lens, aspect is the aspect ratio of the width by the height of the canvas, and near and far are the same as chosen for ortho. The fovy I chose was 95, as this was enough to see the entire object; if it was smaller I would see object bigger as it is zoomed in more, and vice versa if it would be bigger the object would appear smaller. I chose the value to make sure it looked like the ortho projection. The aspect I chose was one, as I didn't want any type of distortion to my cube. In order to change from ortho to perspective projection, I added a button that when clicked would change the perspective. For this I used a If-else to move from each one and since near and far are used in both they work for both projections. When looking at both projections, you can tell the difference between them and how near and far affect these; when near is increased, part of the image is clipped out and you get a weird shape but when far is increased, the cube disappears and then appears in some other value and later disappears if increased even more. Decreasing both doesn't change the image, although if far is decreased to its minimum it also disappears. While on the other hand, for ortho the cube stays normal until more or less two and later starts to get distorted as near is increased. On the other hand, if far is increased or decreased, the cube completely disappears. The same advantages and disadvantages stand for perspective projection as ortho, picking near and far can be tricky and is a disadvantage but the other values have a more direct and obvious effect. While the advantage is the ease of use and simple way to write the code in a line.

The next assignment was to introduce a light source and replace the colors by the properties of a material of my choice. I decided to use one light source that is fixed in position and orientation.

This position is 10 in the x and y coordinates which are to the right and up, respectively and -2 in the z direction which translates to away from the viewer. I later picked the properties of the light; I decided to make it a white light with perfect qualities (it would reflect everything) so ambient, diffuse and specular for the light were set to (0, 1, 1, 1, 1) which translates to white color. Next I picked my material qualities; this took me a long time to decide because I tried many combinations and I was not getting the effect that I wanted and also I was working with the light at the same time, so I was getting different colors and effects by moving just one property slightly. I also noticed that we had converted the normal values that are usually RGBa to values between 0 and 1. After having a lot of issues with this and trying to simulate a texture, I decided to do some research and found a page that showed different material property colors: http://devernay.free.fr/cours/opengl/materials.html. The material property that I chose was Jade, which is a stone that has a greenish tone to it similar to ocean water at certain depths in the shore. I chose this material because it has an amazing color that is uplifting and the Jade is a natural mineral that is used for many decorations. Using the colors referenced, I changed the values for ambient, diffuse and specular for the material. I chose a material shininess as 200, although the page suggested a lower value, because I saw many images of Jade with pretty good shininess; you could see the light reflection on the surface. To make the light sources blend with the material, both their properties need to be blended by multiplying their matrices separately. This is later sent to the vertex and fragment shaders by finding its location, flattening the matrix and uniform-ing it. The make all these properties be placed on the cube, I had to get the normal of each side of the cube. To do this, I subtracted the vertices that join at the edges and took the cross product of these. This would give me the normals for each face of the cube; normal refers to the vector perpendicular to the plane of each side of the cube as it moves. To be able to do this, a buffer is created to get the data of each as an array and send it to the vertex and fragment shaders. Finally you get to the point of creating the Phong shading in the vertex shader and passing the color to the fragment shader to place on each fragment. In the vertex shader, I created and normalized the vector of the following: the light source L as the distance between this and the object, the viewer E as the negative position of the object, and the half-way vector as the addition of these two. Later to transform vertex normal into eye coordinates, and compute terms in the illumination equation using Kd and Ks, coefficient of diffusing and specular, respectively. The final color is the combination of ambient, specular, and diffuse and is sent to the fragment shader to place on each fragment. The advantages of using the Phong method is that the shading looks smoother for curved surfaces, in our case though because we have edges this might actually be a disadvantage making the Gouraud method better (in fact it is, but will describe later). Another disadvantage of using this method is that it takes more work to get it done because it gets the vertex normal across the edges and also the edge normal across the entire cube.

The other type of shading is the Gouraud Shading which in the vertex shader finds the average of the normal at each vertex and then sends it to the fragment shader to apply the Phong model to each vertex and give the shades to the polygon in each fragment. Therefore, the difference between these models is where the Phong model is calculated (vertex shader for Phong and fragment shader for Gouraud) and what normals are used (across edges and cube for Phong and averaged for Gouraud). In order to make both shading work in the same program, a button was used to switch between these two. I had a lot of issues trying to get both to work together in the same program, as separately they worked great. I tried different ways to do this like by using if-else inside each shader and having two main functions in order to call each with their variables; I also tried to put if-else instead in the JS file when calling the program but it gave me an error that it switched programs and lost the information; I also tried to create to separate programs; also create two separate vertex and fragment shader per shading type but only the first one worked. Until I finally went back to my first option I had done and examined the error that it was giving me; it said that I was using the same variable with different values as it was replacing the old one. Therefore, I decided to create separate variables for each shading type and separate them, so they would not get confused while running. And it worked! The advantage of using this method is that it is much simpler than any other option, it's intuitive to understand and works perfectly. Another advantage is that using Gouraud shading method is better for a cube that has edged instead of surfaces. The lighting and reflections seem more realistic given the chosen parameters; it is seen as a light that shines at the cube while it rotates instead of being diffused completely in the color. It is much easier to see this effect when the scale of the cubes is increased as the light becomes more visible and less sparse with the distance to a smaller cube; this can be a disadvantage as you need to have the perfect size to see the reflection as you want. This is partly due to the fact that I chose to have a positional light source instead of a directional that just shines in all direction in an angle like the sun.