INTERACTIVE GRAPHICS HW1: **GARBA MARIAM 1871871**

Exercise 1:

In order to compute the model view and projection matrices in the application, we have to position the viewer which is the camera. We need to define the camera position using a model view matrix and select the lens using a projection matrix. These matrices has been defined in the vertex shader which can be found in index.html like this:

```
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
gl_Position = modelViewMatrix * vPosition;
```

The modelViewMatrix is computed using the function lookAt(eye,at,up) in the render() function which can be found in homework1.js file. The default values of eye,at and up is as follows:

```
var eye = vec3(0.0, 0.0, 3.0);
const at = vec3(0.0, 0.0, 0.0);
const up = vec3(0.0, 1.0, 0.0);
cameraTransformationMatrix = lookAt(eye, at , up);
cameraTransformationMatrix=mult(cameraTransformationMatrix,translate(translateVector));
cameraTransformationMatrix = mult(cameraTransformationMatrix,scalem(scaleVector));
cameraTransformationMatrix = mult(cameraTransformationMatrix,rotateZ(-theta[zAxis]));
cameraTransformationMatrix = mult(cameraTransformationMatrix,rotateY(-theta[yAxis]));
cameraTransformationMatrix = mult(cameraTransformationMatrix,rotateX(-theta[xAxis]));
gl.uniformMatrix4fv( modelViewMatrixLoc, false, flatten(cameraTransformationMatrix) );
```

This lookAt() function is responsible for moving all objects in front of the camera. I have computed two projective matrix in the application, one for perspective projection and another for orthogonal projection. Each of these projections were defined in the application as:

```
projectionMatrix = perspective(fovy, aspect, near, far);
projectionMatrix = ortho(left, right, bottom, ytop, near, far);
var near = 0.3; var far = 5.0; var fovy = 45.0; var aspect = 1.0; var left = -1.0; var right = 1.0;
var ytop =1.0;var bottom = -1.0;
gl.uniformMatrix4fv( projectionMatrixLoc, false, flatten(projectionMatrix) );
```

I also made the viewing position and volume controllable in the html file using buttons and sliders.

Exercise 2:
Scaling and Translation were introduced to the projections which is controlled using sliders that is implemented in the html file.

A scale vector variable was defined in the javascript application and converted to a scaling matrix using scalem() function in MV.js file.

```
scaleVector = [1.0,1.0,1.0]
cameraTransformationMatrix = mult(cameraTransformationMatrix,scalem(scaleVector));
document.getElementById("scalingSlider").oninput = function(){
        scaleVector = [event.target.value, event.target.value, event.target.value];
}
```

A translation vector variable was defined in the javascript application and converted to a translation matrix using translate() function in MV.js file.

```
var translateVector = [0.0, 0.0, 0.0];
cameraTransformationMatrix= mult(cameraTransformationMatrix,translate(translateVector));
document.getElementById("translationXSlider").oninput = function(){
        translateVector[xAxis] = event.target.value;
}
document.getElementById("translationYSlider").oninput = function(){
        translateVector[yAxis] = event.target.value;
}
document.getElementById("translationZSlider").oninput = function(){
        translateVector[zAxis] = event.target.value;
}
```

The values of the scaleVector and translateVector can be updated with the sliders that's implemented in the html file.

Exercise 3:

The orthographic projection has been computed in the javascript application as follows:

```
projectionMatrix = ortho(left, right, bottom, ytop, near, far);
var near = 0.3; var far = 5.0; var left = -1.0; var right = 1.0; var ytop =1.0; var bottom = -1.0;
```

The projectionMatrix is sent to vertex and fragment shaders like this:

```
gl.uniformMatrix4fv( projectionMatrixLoc, false, flatten(projectionMatrix) );
```

The near and far values of the orthogonal projection can be updated using the sliders defined in the html file. It is implemented in the javascript application as follows:

```
document.getElementById("nearSlider").oninput = function(event){
        near = event.target.value;
}
document.getElementById("farSlider").oninput = function(event){
        far = event.target.value;
}
```

Exercise 4:

The canvas window was splitted vertically into two, one for orthogonal projection and the other for perspective projection. The code below was used to split the window in the javascript application:

```
{ projectionMatrix = ortho(left, right, bottom, ytop, near, far);
 renderScene(0, 0, width / 2, height,projectionMatrix);}
{ projectionMatrix = perspective(fovy, aspect, near, far);
 renderScene(width / 2, 0, width / 2, height,projectionMatrix);}
```

The function renderScene() takes in the viewport and projectionMatrix in order to render a projection on the screen.

Exercise 5:

The light source was introduced with the lightPosition variable and the colors are replaced by the properties of the material as follows:

```
var lightPosition = vec4(0.8, 0.8, 0.8, 1.0 );
var lightAmbient = vec4(1.0, 1.0, 1.0, 1.0 );
var lightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var lightSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );
var materialAmbient = vec4(0.1745, 0.01175, 0.01175, 1.0);
var materialDiffuse = vec4(0.61424,0.04136,0.04136, 1.0);
var materialSpecular = vec4(0.727811,0.626959,0.626959, 1.0);
var materialShininess = 400;
```

To get the reflected colors, we have to multiply the light properties and material properties.

```
ambientProduct = mult(lightAmbient, materialAmbient);
diffuseProduct = mult(lightDiffuse, materialDiffuse);
specularProduct = mult(lightSpecular, materialSpecular);
```

In order to compute the normals, we need to subtract two vertices of two cube faces (as two vectors) of the triangle in the quad() function and perform cross product of these two vectors.
to get the normal vector to the cube face:

```
var t1 = subtract(vertices[b], vertices[a]);
var t2 = subtract(vertices[c], vertices[b]);
var normal = cross(t1, t2);
var normal = vec3(normal);
```

Finally, I pushed the normal of each vertex to the array normalsArray.

```
normalsArray.push(normal);
```

Exercise 6:

In the html file, I added a button to change the shading method using a boolean variable phongShader initialized in JavaScript.I used two different programs (one vertex and fragment shader each) for the gouraud and phong shading. Whenever phongShader is true, the phong shading model is activated otherwise the gourand shading model is activated.

In order to implement a shading model, the following variables are initialized as uniform in the javascript application and sent to the vertex shader (for gourand shading model) and fragment shader (for phong shading model) respectively :

AmbientProduct, DiffuseProduct, SpecularProduct,
shininess,lightPosition,vNormal, vPosition

Then we convert the the light position and viewer position into the eye coordinate:

```
vec3 pos = -(modelViewMatrix * vPosition).xyz;
vec3 light = lightPosition.xyz;
```

Then we compute where the light and vertex vectors are in terms of illumination:

```
vec4 ambient = ambientProduct;
float Kd = max(dot(light_direction, N), 0.0);
vec4 diffuse = Kd * diffuseProduct;
float Ks = pow(max(dot(N, halfVector), 0.0), shininess);
vec4 specular = Ks * specularProduct;
if (dot(light_direction, N) < 0.0) {
specular = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Also for the phong model, we compute the light and vertex vectors position in fragment shader. We send same uniforms, as defined above to vertex shader. We compute normals at each vertex in object coordinates and send them to the fragment shader:

```
varying vec3 N, light_direction, E;
```

The only thing that vertex shader has to do in phong shading method, is to calculate normal, direction to the light and to the vertex vectors and send them as varying variables to fragment shader. The rest of the procedure is almost the same as calculation that we had done in vertex shader for gouraud lighting model.

Exercise 7:
The procedural texture was defined in the javascript application as:

```
var image1 = new Uint8Array(4* texSize* texSize);
        for ( var i = 0; i < texSize; i++ ) {
                for ( var j = 0; j <texSize; j++ ) {
                var patchx = Math.floor(i/(texSize/numChecks));
                if(patchx% 2) c = 255;
                else c = 0;
                image1[4* i* texSize+4* j] = c;
                image1[4* i* texSize+4* j+1] = c;
                image1[4* i* texSize+4* j+2] = c;
                image1[4* i* texSize+4* j+3] = 255;
                }
        }
```

Then I assigned texture coordinates to 4 vertices for each face of the cube:

```
var texCoord = [
vec2(0, 0),
        vec2(0, 1),
        vec2(1, 1),
        vec2(1, 0)];
```

And then push the value of the texture coordinate to texture coordinate arrays:

```
texCoordsArray.push(texCoord[0]);
```

All the textCoord can then be binded together using the configureTexture() function.
In order to apply the texture to the cube in the fragment shader, a sampler2D variable holds the result of the texture coordinate that have been computed in the application:

```
gl_FragColor = fColor* (texture2D(Tex0, fTexCoord)* texture2D(Tex1, fTexCoord))
```

Additionally, I have multiplied texture2D with fColor which is the combination of material and
light properties so that pixel color would be a combination of the color computed using the lighting model and the texture.

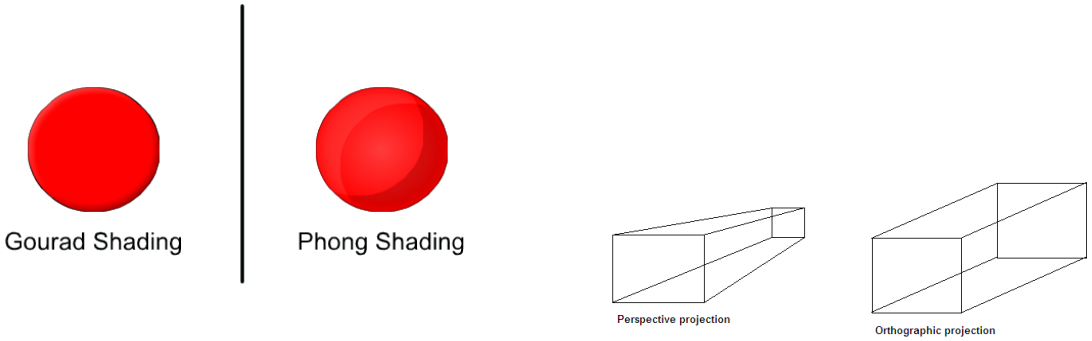
Advantages and Disadvantages of proposed solutions

I implemented two projections for viewing objects in a scene: Orthographic and Perspective projections.

Orthographic projections preserves both distances and angles. It cannot see what object really looks like because many surfaces are hidden from view. On the other hand, in Perspective projections, objects further from viewer are projected smaller than the same sized objects closer to the viewer (diminution). It looks more realistic. Equal distances along a line are not projected into equal distances (nonuniform foreshortening). Angles are preserved only in plans parallel to the projection plane. They are also more difficult to construct by hand than parallel projections (but not difficult by computer).

I also implemented two shading models: Gouraud and Phong Shading.

In Gouraud shading, intensity levels are calculated at each vertex and interpolated across the surface of the object. It removes the intensity discontinuity which exists in constant shading model. It has a problem with specular reflections. it is computationally less expensive compared to phong shading. It's disadvantage is that it can introduce anomalies known as Mach bands. On the other hand, in Phong shading, there is more accurate interpolation based approach for rendering objects. It interpolates normal vectors instead of intensity values. It greatly reduces the Mach band effect. It also works better for objects that has small specular highlights. It's disadvantage is that it requires more calculations and greatly increases the cost of shading steeply.



Features of my solution

In order to implement both Gouraud and Phong shading, I implemented one vertex and one fragment shader each for the two shading techniques. It is done in this way so as to modularise the implementation for flexibility purpose.