Homework 1 - Interactive graphics

The provided codebase for this homework consisted of a cube with each face with its own color; the camera and the object had a fixed position.
On top of that codebase, a wide range of features had to be implemented which will be detailed below.

Handling viewer position

The first feature requested by the specification was a way to customize the viewer position.
In order to implement this feature, I declared three variables: *theta*, *phi*, and *radium*.
Those three variables were used to compute the variable "*eye*" which represents the camera position.

```
eye = vec3(radius*Math.sin(phi), radius*Math.sin(theta), radius*Math.cos(phi));
```

This variable eye will be later used to compute an updated model-view matrix using the *lookAt()* function which transforms vertices from world space to camera space.

```
modelViewMatrix = lookAt(eye, at , up);
```

Instead of using the *lookAt* function we could do manually a series of normalizations and multiplication (replicating how *lookAt* works) but this path is more complex and, considering that we are not interested in changing how *lookAt* works, it also doesn't provide any advantage.

The *modelViewMatrix* plays an important role because it combines the viewing matrix and modeling matrix into one matrix and when used to compute *gl_Position* in the vertex shader it will allow us to influence what is shown on the screen.

```
gl_Position = modelViewMatrix*vPosition;
```

The specification required that the user could control the viewer position dynamically from the HTML page, to do that a set of buttons were implemented.
Those buttons have attached a listener which changes at runtime the values of phi and theta in order to influence the eye variables in the render loop.

Scaling and translating the model

After handling the viewer position, there was the necessity of scaling and translating the model along the *X*, *Y* and *Z* axis.
In order to do that a translation and a scaling matrix were introduced in the render loop.

```
    // Translation matrix
    var translationMatrix = [
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        fTranslateX, fTranslateY, fTranslateZ, 1.0
    ];
    // Scaling matrix
    var scalingMatrix = [
        fScaler, 0.0, 0.0, 0.0,
        0.0, fScaler, 0.0, 0.0,
        0.0, 0.0, fScaler, 0.0,
        0.0, 0.0, 0.0, 1.0
    ];
```

Those matrices are computed at runtime in the render loop and presents some parameters that may change depending on the user's interaction on the sliders of the HTML page.
Those matrices will be multiplied with each other when computing the *gl_Position* in the vertex shader.
So at this point *gl_Position* is computed as

```
gl_Position = modelViewMatrix * translationMatrix * scalingMatrix * vPosition;
```

Computing an orthogonal or a perspective projection

The provided specification also required to computed (and render) an orthogonal projection of the model while at the same time letting the user control the view volume.
To let the user controls the viewing volume a set of variables which influences the resulting projection were declared, the values of these variables can be dynamically changed using the slider and buttons in the HTML page.
WebGL provides an easy way to compute the projection matrix through its APIs, the method offered to produce an orthogonal projection is *ortho()*

```
projectionMatrix = ortho( left, right, bottom, top, near, far );
```

The variables passed as arguments influence the projection and its viewing volume, in particular, *left* and *right* represents the coordinates of the vertical clipping planes, *bottom* and *top* represents the horizontal clipping planes while *near* and *far* control the distances to the nearer and farther depth clipping planes.

The values of those variables can be changed using the sliders in the HTML page by the user, obviously changing the resulting projection.

Note that when ration *far/near* increases the projection loses precision, so it is highly recommended to

not use a deep and if needed to scale the scene.
The resulting projection matrix plays a role, as the other matrices presented above, in computing the gl_Position.

```
gl_Position = projectionMatrix * modelViewMatrix * translationMatrix * scalingMatrix * vPosition;
```

If we are interested in doing a perspective projection the procedure is very similar to what was already shown, the main difference is the API used to compute the projection matrix

```
projectionMatrix=perspective( fovy, aspect, near,far );
```

Using the perspective method we don't have to set the coordinates of the horizontal or vertical clipping planes but we have to provide the *field of view angle*, the *aspect ratio* that determines the field of view in the x direction and *near/far* which holds the same meaning of before.

Splitting the canvas

There are multiple ways to show multiple objects at the same time in WebGL.
One way would be to use different canvas, drawing the objects in one context and then applying each object to the right canvas.
For this homework, I didn't declare multiple canvases in the HTML page but I opted to split the existing one by changing the viewport and using the scissor test which is a per-sample processing operation that creates a scissor box.
The scissor box provides a way to modify using drawing commands only the pixels that lie within the scissor box.

So at the end in the render the step that I followed to render two cubes, one with an orthogonal projection applied to it and another with a perspective one were:

- Enabled the scissor test and created a scissor box (with x,y pointing to the lower-left position) of the scissor box
- Changed the viewport accordingly
- Rendered the cube and applied an orthogonal projection
- Modify the scissor box to cover the second section of my other model
- Changed again the viewport to reflect the new scissor box
- Rendered the cube and applied a perspective projection

```
// Split canvas
gl.enable(gl.SCISSOR_TEST); //enable scissor test
var width = gl.canvas.width;
var height = gl.canvas.height;
gl.scissor(0,  height/2, width / 2, height/2); // create scissor box (x,y,width,height)
gl.viewport(0,  height/2, width / 2, height/2); // change viewport (x,y,width,height)
projectionMatrix = ortho( left, right, bottom, ytop, near, far );

gl.drawArrays( gl.TRIANGLES, 0, numVertices );

// Create perspective view of the object
var aspect = canvas.width/canvas.height;
var fovy = 95.0;
projectionMatrix=perspective(fovy,aspect,near,far);
gl.uniformMatrix4fv( projectionMatrixLoc, false, flatten(projectionMatrix) );
```

```
    gl.scissor(width / 2, height/2, width/2, height/2);
    gl.viewport(width / 2, height/2, width/2, height/2);
    gl.drawArrays( gl.TRIANGLES, 0, numVertices );
```

The reason why I opted for splitting the existing canvas was definitely less memory intensive and easier to implement than creating multiple canvases each managing its own resources.

Adding a light source

As per specification, there was the need of adding a light source to the scene.
Light in WebGL and OpenGL is described using vectors each representing a mechanism through which the light that reaches your eye.
There is the ambient light, that comes from all directions equally and is scattered in all directions equally by the polygons in your scene, the diffuse light which comes from a particular point source, the emission light emitted by the polygon itself and the specular light that takes into account the direct reflection of photons off of the surface in order to provide a shiny finish to a specific part of the object.
In the end, lighting will depend on the material and its colors, light colors, illuminance, light direction, viewer position, and normal vector.
The normal vector, in particular, is a unit vector that is perpendicular to a surface at a specific point and is often used, when dealing with lightning calculations, to compute the angle between the normal at a given surface point and the direction towards a light source or a camera.
In the homework a lightsource was described using the following vectors

```
var lightPosition = vec4(0.7, 0.5, 0.5, 0.0);
var lightAmbient = vec4(0.9, 0.9, 0.9, 1.0);
var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);
var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);
```

which are later multiplied with the materials property and passed to the shaders.

```
    var ambientProduct = mult(lightAmbient, materialAmbient);
    var diffuseProduct = mult(lightDiffuse, materialDiffuse);
    var specularProduct = mult(lightSpecular, materialSpecular);
    gl.uniform4fv(gl.getUniformLocation(program, "ambientProduct"), flatten(ambientProduct));
    gl.uniform4fv(gl.getUniformLocation(program, "diffuseProduct"), flatten(diffuseProduct));
    gl.uniform4fv(gl.getUniformLocation(program, "specularProduct"), flatten(specularProduct));
    gl.uniform4fv(gl.getUniformLocation(program, "lightPosition"), flatten(lightPosition));
```

I opted for using only one light source because it satisfied the requisites and because multiple light sources have a performance impact that cannot be ignored.

Regarding the normals, those were computed in the onLoad function using a cross-product of two edge vectors and later passed to the shader using a buffer.

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

```
// In onLoad function
var nBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, nBuffer );
```

```
gl.bufferData( gl.ARRAY_BUFFER, flatten(normalsArray), gl.STATIC_DRAW );
```

Adding a material to the cube

Now that we got a split canvas with different projections it's time to replace the colors of the cube with a material which describe how the object reacts when there is light on it.
A material in WebGL and OpenGL is defined by its properties, which describes: the color that is reflected under the ambient/diffuse/specular lighting and the shininess (intensity of reflection).
Those properties are represented as vectors, in particular for the homework, I decided to use the turquoise material.
Those are the vectors and the values used:

```
var materialAmbient = vec4( 0.1, 0.18725,   0.1745  );
var materialDiffuse = vec4( 0.396, 0.74151, 0.69102);
var materialSpecular = vec4( 0.297254   ,0.30829,   0.306678);
var materialShininess = 60.0;
```

As shown before those vectors are used to compute matrices that are later passed to the shader and influences the rendering of the object.

Gouraud and Phong shading

For this homework, two shading techniques were implemented: Phong and Gouraud shading.
The first one is a per-fragment color computation. In the Phong shading, the works are carried out in the fragment shader which takes the normal and position data provides the normal and position data and then interpolates these variables and computes the color.
Gouraud shading, instead, is a per-vertex color computation. This means that the vertex shader must determine a color for each vertex and pass it to the fragment shader, where it gets interpolated across the fragments thus giving the smooth shading.
Gouraud provides better performance when compared to Phong but gives a big loss of quality, especially on large primitives and strong specular highlights. shading technique because the computation is done per-fragment.
Gouraud shading is effective for shading surfaces which reflect light diffusely.
This is due to the fact that, when dealing with specular reflections, the shape of the specular highlight produced is dependent on the relative positions of the underlying polygons, Phong, instead, produces highlights which are much less dependent on the underlying polygons.

The users can change which shading technique has to be used thanks to a set of radio buttons, in the HTML page, which influences the value of a boolean variable linked to the shader that is used to decide where the color computation has to be carried out.

Implementing a procedural texture

The last feature that got implemented is the support for a procedural texture on the cube object.
A procedural texture is a texture created using an algorithm rather than directly stored data, an example of texture is 2D chessboard.
For the homework, the 2D chessboard texture is generated using the following code.

```
var image1 = new Array()
   for (var i =0; i<texSize; i++)  image1[i] = new Array();
```

```
   for (var i =0; i<texSize; i++)
       for ( var j = 0; j < texSize; j++)
          image1[i][j] = new Float32Array(4);
   for (var i =0; i<texSize; i++) for (var j=0; j<texSize; j++) {
       var c = (((i & 0x8) == 0) ^ ((j & 0x8)  == 0));
       image1[i][j] = [c, c, c, 1];
   }

var image2 = new Uint8Array(4*texSize*texSize);
   for ( var i = 0; i < texSize; i++ )
       for ( var j = 0; j < texSize; j++ )
          for(var k =0; k<4; k++)
               image2[4*texSize*i+4*j+k] = 255*image1[i][j][k];
```

This code produces a checkerboard pattern stored in an array which has its pixels expressed in the RGBA format, there are black square zones (whose dimension changes depending on the texSize value) and transparent squares in which we can see the material of which the cube is made of.
For the sake of simplicity, the texture is fixed and it doesn't change at runtime.
Later on, the array in which the pattern is stored is converted to a texture using WebGL's APIs and passed to the fragment shader.

```
var texture = gl.createTexture();
gl.activeTexture( gl.TEXTURE0 );
gl.bindTexture( gl.TEXTURE_2D, texture );
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0, gl.RGBA, gl.UNSIGNED_BYTE, image);
gl.generateMipmap( gl.TEXTURE_2D );
gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
gl.NEAREST_MIPMAP_LINEAR );
gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST );
var tBuffer = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, tBuffer);
gl.bufferData( gl.ARRAY_BUFFER, flatten(texCoordsArray), gl.STATIC_DRAW );
```

At this point, the shader will take care of computing the color of the fragment taking into account the lighting and texture, in order to look up a color from the texture the *texture2D()* method is used.

```
gl_FragColor = fColorF*texture2D( texture, fTexCoord );
```

I decided to offer to the user a button that enable or disable the texture on the cube, in order to do that I declared a flag which is checked inside the fragment shader and according to its value the *gl_FragColor* is computed considering or not the value of *texture2D()*.