Homework 1

Solution by Pignata Giovanni 1913547

1. The Table

To model the table, I modified the cube function to draw cubes with faces of different lengths. Then I created the object with a stack of three cubes, for a total of 24 vertices and 36*3 vertex coordinates.

Since the object was formed by quadrilateral (each of them was approximated by 2 triangles), in order to compute the normal associated to each vertex, I used the following code inside the function **myCube()**:

```
        var t1 = subtract(vertices[idx[i][1]], vertices[idx[i][0]]);
        var t2 = subtract(vertices[idx[i][2]], vertices[idx[i][1]]);
        var normal = cross(t1, t2);
        var tangent = vec3(t1[0],t1[1],t1[2]);
        normal = vec3(normal[0],normal[1],normal[2]);
```

For the texture coordinates, I used the following code in order to link the extreme vertices of the **wood.jpg** texture and the vertices of the to draw:

```
    const texIdx = [0, 1, 2, 0, 2, 3];

    const texCoord = [
        vec2(0, 0),
        vec2(0, 1),
        vec2(1, 1),
        vec2(1, 0)
    ]
//for each vertex:
      texCoords.push(texCoord[texIdx[j]]);
```

2. The Animation

I set defined the origin of the rotation in **origin = vec4(10.0, 0.0, 0.0, 1.0)** and defined a **transformMatrix** to compute the rotation animation as follow:

```
/* In render()*/
//  Compute Transformation Matrix...
    theta[axis] += 2.0 * direction;
    transformMatrix = modelViewMatrix;
    transformMatrix = mult(transformMatrix,translate(origin[0],origin[1],origin[2]));
    transformMatrix = mult(transformMatrix, rotateX(theta[0]));
    transformMatrix = mult(transformMatrix, rotateY(theta[1]));
    transformMatrix = mult(transformMatrix, rotateZ(theta[2]));
transformMatrix = mult(transformMatrix,translate(-origin[0],-origin[1],-origin[2]));
```

where **direction** is the variable that control the animation [0 = motionless, 1 = counterclockwise, -1 clockwise] and theta is the angle of rotation. The transformation matrix is then used in the vertex shader in order to compute the vertex position.

3. The Perspective Camera

The following are the parameters used to set up the camera:

```
// Initial Camera Coordinates:
var xCamera = 0.0;
var yCamera = 20.0;
var zCamera = 10.0;
```

```
var eye = vec3(xCamera, yCamera, zCamera);    // Viewer Position:
const at = vec3(0.0, 0.0, 0.0);    // Set where the camera point.
const up = vec3(0.0, 1.0, 0.0);    // Set the top of the camera.
```

In order to compute the **modelViewMatrix**, I used the **lookAt()** function as follow

```
  // in render():
  modelViewMatrix = lookAt(eye, at, up);
```

For the perspective projection, the **projectionMatrix** is computed as follow:

```
// ProjectionMatrix Parameters::
var projectionMatrix;
var near = 1.0;              // Distance of the near face of the viewing Volume.
var far = 60.0;              // Distance of the far face of the viewing Volume.
var fovy = 80.0;             // Field-of-view in Y direction angle (in degrees).
var aspect;                  // Viewport aspect ratio.
// In render():
projectionMatrix = perspective(fovy, aspect, near, far);
```

I used these matrices in the vertex shader to compute the final vertex position:

```
gl_Position = uProjectionMatrix * uTransformMatrix * aPosition;
```

4. The Spotlight

These are the spotlight parameters:

```
// SPOTLIGHT Parameters:
var lightEnabled = true;     // Boolean that controls if the spotlight is turned on.
var lightPosition = vec4(0.0, 30.0, 0.0, 1.0);      // Position of the spotlight.
var lightDirection = vec4(                          // Spotlight Initial Direction.
    lightPosition[0],
    lightPosition[1],
    lightPosition[2],
    0.0);
var lightAngle = 20;                            // Cut-off Angle of the spotlight.
// Light Attenuation factors:
var lightAttenuationCostant = 1.0;
var lightAttenuationLinear = 0.01;
var lightAttenuationQuadratic = 0.0005;
```

In the shaders, in order to control if a point is inside the cone of light, I wrote the following code:

```
    vec3 position = (uTransformMatrix * aPosition).xyz;
    vec3 lightPosition = (uModelViewMatrix * uSpotLight.position).xyz;
    L = (lightPosition - position);
    D = (uModelViewMatrix * uSpotLight.direction).xyz;
    vec3 l = normalize(L);
    vec3 d = normalize(D);
    float dl = dot(l,d);
    if (dl > uSpotLight.threshold){
            // Compute the light equation.
      }
```

where **uSpotLight.threshold** is the cos(lightAngle).

For the attenuation factor, I computed it as follow:

```
      float a = uSpotLight.attenuationConstant;
      float dist2 = (dot(L,L));
      a +=  (uSpotLight.attenuationQuadratic * dist2);
      float dist = sqrt(dist2);
      a += (uSpotLight.attenuationLinear*dist);
```

5. Object Material

```
// Material parameters:
var materialAmbient = vec4(0.5, 0.5, 0.5, 1.0);
var materialDiffuse = vec4(.56, 0.32, 0.32, 1.0);
var materialSpecular = vec4(1.0, 1.0, 1.0, 1.0);
var materialShininess = 50.0;

// Light Color Parameters:
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);
var lightDiffuse = vec4(1.0, 1., 0.8, 1.);
var lightSpecular = vec4(1.0, 1.0, 0.8, 1.0);

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

The last three variables are used in the shaders in order to compute the three term of the light equation.

per-Vertex and per-Fragment shaders

In one of the two shaders, the terms of the light equation are computed in the same way:

```
vec4 color = uMaterial.emissive;

vec4 ambient = uMaterial.ambientProduct/a;

N = (uModelViewMatrix * vec4(aNormal,0)).xyz;
vec3 n = normalize(N);
float kd = max(dot(l,n), 0.0);
diffuse = (kd/a) * uMaterial.diffuseProduct;

V = normalize(-position);
vec3 v = normalize(V);
vec3 h = normalize(l+v);
float ks = pow(max(dot(n,h),0.0), uMaterial.shininess);
specular = ks/a * uMaterial.specularProduct;

color = color + ambient + diffuse + specular;
color.a = uMaterial.emissive.a;
```

L, V, N are alway computed in the vertex shader and are passed to the fragment shader. If per-fragment shading is enabled, in the fragment shader these vectors are normalized and used to compute the terms above.

² The Texture

I included the texture **wood.jpg** in the html file:

```
<img id = "texImage" src = "wood.jpg" hidden></img>
```

and used it to configure the texture in the **init()** function:

```
// Texture...
    var texBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, texBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoords), gl.STATIC_DRAW);

    var texCoordLoc = gl.getAttribLocation(program, "aTexCoord");
    gl.vertexAttribPointer(texCoordLoc, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(texCoordLoc);

    var image = document.getElementById("texImage");
    configureTexture(image);
```

the code of **configureTexture(image)** is the follow:

```
function configureTexture( image ) {
    texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB,
         gl.RGB, 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);

    gl.uniform1i(gl.getUniformLocation(program, "uTexMap"), 0);
}
```