CM20219 FUNDAMENTALS OF VISUAL COMPUTING

INTRODUCTION

The aim of this report is to analyze and describe the rendering and manipulation of a 3D model using WebGL. This would lead to a deeper understanding of the core of Visual Computing.

REQUIREMENTS

1. Draw a simple cube

Explanation: The first requirements consists in creating and displaying a cube centered at the origin of the scene (0,0,0), implying that two opposite vertices have as coordinates (1,1,1) and (-1-1,-1) respectively, with each edge parallel to one of the x-, y-, z-axes. A cube is being made by putting together the geometry (where all the vertices are created) and material (where the colour/texture is added).

Implementation: All the code is added in the `init()` function in order for the cube to be displayed withe the `scene.add()` method, which automatically places the cube at the center of the scene. For the geometry part the THREE.`BoxGeometry(1, 1, 1)` class was used (it creates a rectangular shape by receiving as inputs the width, height and depth of the shape, all of this were set to 1 to meet the requirement). When it comes to the material a sensible solution is to use `THREE.MeshPhongMaterial()` which provides a shiny surface allowing the mesh to have highlights, since it uses specular light. The given parameter is referring to colour (green in this case `0000ff`), all the values of colours are in hexadecimal values, denoted by the `0x` at the beginning of every value. The Phong shading is used to determine the shade of every pixel colour consisting in 3 components (one for each RGB colour). An ambient light was used `THREE.AmbientLight(0xffffff))` making it impossible for the highlights and shadings to be observed. There are alternatives for the Phong material such as Lambert material which uses Ground shading(using polygons) and therefore allowing for no highlights to be created. By using the `THREE.Mesh()` method, the geometry and the material are put together and created the mesh required as seen in *fig. 1.1* and *code 1.1* below. .

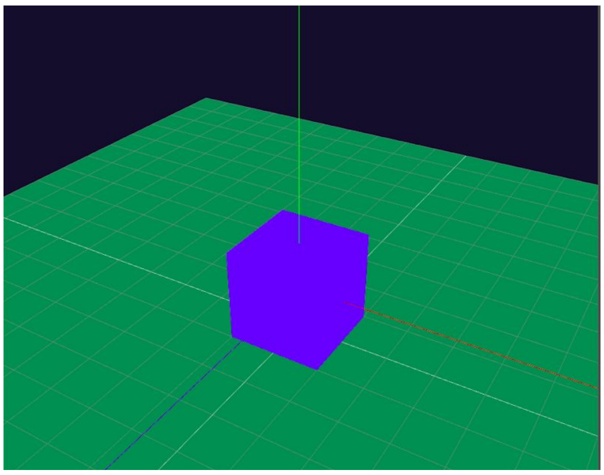


fig. 1.1 (the cube is set at the center of the world, while the floor is set to -0.5 on the y axis)

```
geometry = new THREE.BoxGeometry(1, 1, 1);
var material = new THREE.MeshPhongMaterial({ color: 0x00ff00 });
cube = new THREE.Mesh(geometry, texture);
scene.add(cube);
```

code 1.1

2. Draw coordinate system axes

Explanation: Every system has its own axes and having them visible at all times makes testing easier. This requirement is meet by drowint 3 orthogonal lines as the x, y, z axes in red, green and blue. In order to satisfy the second requirement, the geometry and the material are created firstly and then added together to create the three lines.

Implementation: All the code is added in the `init()` function in order for the lines to be displayed withe the `scene.add()` method. For the geometry the basic function `.Geometry()` seemed like the most reasonable solution, because the simplicity of the object and the alternative (BufferGeometry), would me more appropriate for complex geometry. The geometry of the line consists in 2 connected points. The length of every axe is 5. In *code. 2.1* it is showed how the axes are implemented and in *fig 1.1* they are also displayed. It goes the same for all three of them, by, changing of course, the points' x, y and z value respectively as well as the colour of material. Three.js provides the method `LineBasicMaterial()` which creates the material for a continuous line. This function receives as a parameter the colour of the line (`0x0000ff` for bue, `0xff0000` for red, `0x00ff0` for green). the following code exemplifies just for the x axis due to space reasons.

```
var material_x = new THREE.LineBasicMaterial({ color: 0xff0000 });
var geometry_x = new THREE.Geometry();
geometry_x.vertices.push(new THREE.Vector3(5, 0, 0));
geometry_x.vertices.push(new THREE.Vector3(0, 0, 0));
var line_x = new THREE.Line(geometry_x, material_x);
scene.add(line_x);
```

code. 2.1

3. Rotate the cube

Explanation: The cube has to rotate in the x, y, z directions. Since the cube is a mesh, the built in function rotation can be used. The rotation is done on the cube's coordinates by Euler Angles in radians (this value is set to 0.1). Every time the X/Y/Z key is pressed the cube rotates by 0.1 about the according axis.

Implementation: This can be implemented by creating keyboard shortcuts(keys X, Y, Z for the respective axes) for every direction and added in the `handleKeyDown(event)` function as switch cases using ASCII values. This function is later called in `.addEventListener('keydown', handleKeyDown)` as a parameter after `init()` and `animate()`. The rotation radius is set to 0.1 and is used when rotation.x function is called on the cube. The cube's x position is rotated by 0.1 every time the key is pressed. Same goes for y and z axis. The code used for rotation is showed below(*code 3.1*) as well as some pictures to exemplify the explanation and implementation described above*(fig. 3.1, 3.2)*.

```
case 88: cube.rotation.x += 0.1; break; //[x]
case 89: cube.rotation.y += 0.1; break; //[y]
case 90: cube.rotation.z += 0.1; break; //[z]
```

code 3.1

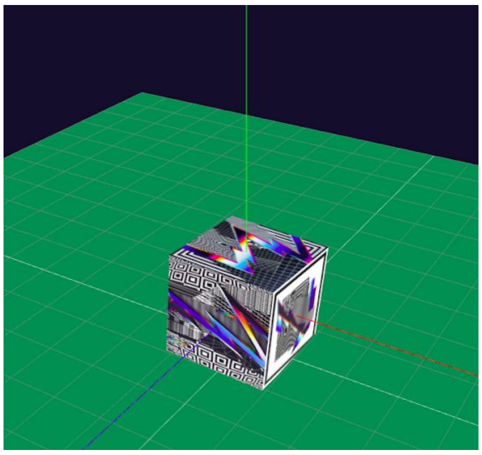


fig 3.1
(the cube is in the initial position)

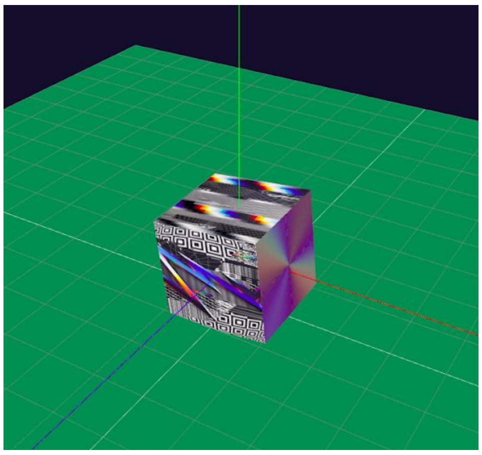


fig 3.2
(the cube has been rotated about the x, y, z axes)

Testing: In order to do the testing for this requirement I checked if all the key shortcuts work properly. And I displayed the values of the cube position on console.

4. Different render modes

Explanation: The requirement asks for the cube to be rendered in 3 modes. First one is by using only the 8 vertices of the cube, the second one renders the 12 edges of the cube by using a wireframe and the third rendering mode has to be the faces. The cube has the same geometry for all rendering modes but the material changes for every rendering mode because the cube has different properties when it comes to the material, therefore a new object has to be created for every rendering mode. It is also worth mentioning that rendering is the process of generating an image from a model.

Implementation: Since we already have the geometry of the cube in the init() function from the firsts requirement we will continue by creating the other material needed for this requirement. The material used for the vertices is `PointsMaterial({color: 0xf74c05, size: 0.1})` since we wish to create an object formed only by the 8 vertices of a cube. Additionally, we set the colour and the size of the points. This step is followed by the initializing a new object using the Point constructor that takes as parameters the geometry of the cube and the point material. (*fig 4.1*)

The material for edge rendering is created using `MeshBasicMaterial` since it has the boolean property `wireframe` that when is set to true makes the material a wireframe(renders as triangles instead of flat polygons). A new object is created afterwards using the Mesh constructor that takes as parameters the geometry of the cube and the edge material. (*fig 4.2*) The face rendering mode is the same as the one in requirement 1 but using a basic material and a texture(explanation in requirement 7) instead of just colour. (*fig 4.3*)Additional lights can be created in order for the faces to better get distinguished. Directional light can create this effect, casting light on one of the cube's faces. *Code 4.1* shows how the implementation of the above.

This meshes are displayed when certain keys are pressed (E for edges, V for vertices and F for faces). They are added to the scene using the `scene.add()` function after removing any object displayed before. All of these are being done in the `handleKeyDown(event)` function by means of switch cases. *Code 4.2* shows how the implementation of the above just for the face rendering, the same goes for the edge and vertex rendering.

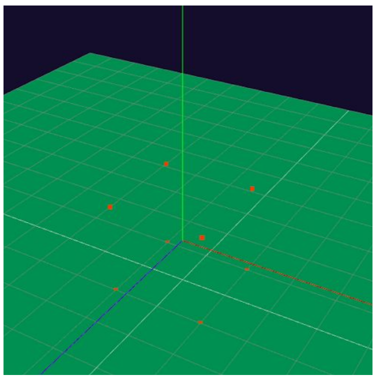


fig. 4.1
(vertex rendering)

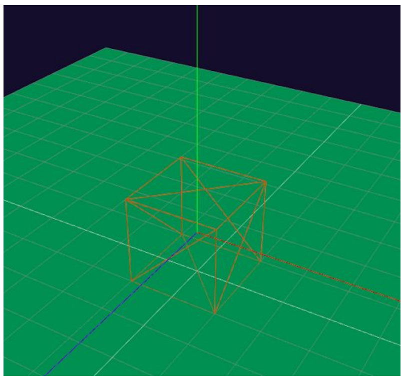


fig. 4.2
(edge rendering)

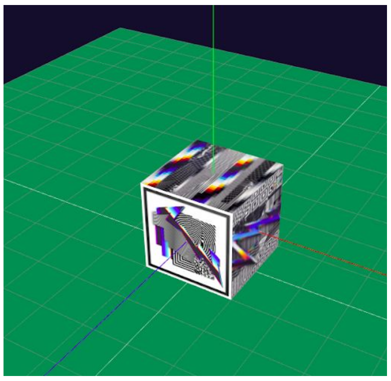


fig. 4.3
(face rendering)

```
geometry = new THREE.BoxGeometry(1, 1, 1);
cube = new THREE.Mesh(geometry, texture);//face
//edge
var  wireframe_material  = new THREE.MeshBasicMaterial({ color: 0xf74c05, wireframe: true,
transparent: true });
cube_edge = new THREE.Mesh(geometry, wireframe_material);
//vertex
var material_vertex = new THREE.PointsMaterial({color: 0xf74c05, size: 0.1});
cube_vertex = new THREE.Points( geometry, material_vertex);
```

code 4.1

```
case 70: // f = face
scene.remove(cube_edge);
scene.remove(cube_vertex);
scene.remove(bunny);
scene.remove(bunny_edge);
scene.remove(bunny_vertex);
scene.add(cube);
break;
```

code 4.2

Testing: In order to do the testing for this requirement I checked if all the key shortcuts work properly. The figures are concludent.

5. Translate the camera

Explanation: This task requires the Manipulation of the camera's location by translating it along its up/down, left/right, and forward/backward directions. The translation the camera should be done along the camera's left/right, up/down, forward/backward vectors, not the axes of the global coordinate system. For the left and right translation the camera vector position has to be changed at the x coordinate. For the up and down translation the camera vector position has to be changed at the y coordinate. For the forward and outward translation the camera vector position has to be changed at the z coordinate.

Implementation: The camera has the inbuilt function for translating it on the x, y, z axes. This property is added to the `handleKeyDown(event)` function. the values given to the function are negatives for the down, left and forward movement and positive values for the other ones. the figures shown below exemplify the camera translation (fig. 5.1, 5.2).

```
case 38: camera.translateY(0.1); break;//Camera UP [arrow up]
case 40: camera.translateY(-0.1); break; //Camera DOWN [arrow down]
case 37: camera.translateX(-0.1); break;//Camera LEFT [arrow left]
case 39: camera.translateX(0.1); break;//Camera RIGHT [arrow right]
case 80: camera.translateZ(-0.1); break;//Camera FORWARD [P]
case 76: camera.translateZ(0.1); break;//Camera BACKWARD [L]
```

Code 5

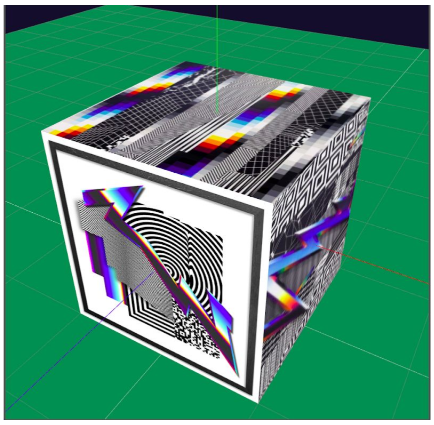


fig 5.1
(camera moving forward)

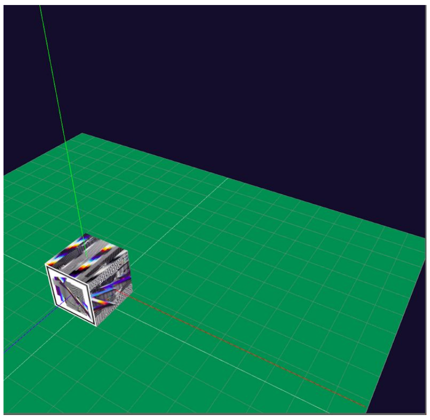


fig. 5.2
(camera moving up and to the left)

Testing: In order to do the testing for this requirement i translated the camera while checking if the axes of the world (req 2) move in the right direction.

6. Orbit the camera

Explanation: Requires to orbit the camera. The spherical coordinates are commuted using the following equations: x = Rcosθsinφ, y = Rsinθsinφ, z = Rcosφ where R = distance between origin and camera, θ = angle of x axis, φ = angle from z axis. The R (radius of the sphere) is meant to remain constant and only by changing the φ and θ the camera moves in an arc ball mode. The cartesian coordinates of the camera should be transformed to spherical coordinates.

7. Texture mapping

Explanation*:* For this task we had to map different textures on every face of the cube using a texture loader. Each face of the cube should look different and the skew is to be avoided.
Implementation*:* Firstly we are loading the images used as textures using `TextureLoader()`. The textures are pushed in the array, which is used to create the material of cube by mapping every texture. The cube mesh is then created by using the geometry from the first requirement and the texture described above. *Fig. 5.1, 3.1, 3.1* all show the different textures used on the cube (they represent artworks by artist Felipe Pantone). The following code *(code 7*) shows the implementation described, but for a single material due to space reasons.

```
var coasta = new THREE.TextureLoader().load( 'textures/felipepantone1.jpg' );
texture.push(new THREE.MeshBasicMaterial({ color: 0xffffff, map: coasta}));
…….
geometry = new THREE.BoxGeometry(1, 1, 1);
cube = new THREE.Mesh(geometry, texture);
scene.add(cube);
```

Code 7

Testing: In order to do the testing for this requirement i rotated the cube on the x, y, z axes to check if all the textures are correctly loaded and if all of them are different.

8. Load a mesh model from .obj

Explanation: This task requires to load an .obj file and translate and scale it to fit into the box. OBJLoader.js is needed for this task along with the .obj file (both of them have to be in the project file).
Implementation: This task has a similar implementation to the one used for the cube, but first, the 3D object needs to be loaded using a loader (`OBJLoader()`), I am doing this in the `init()` function. The function `.load()` called on the loader takes the 3D object file and passes it to a function (`dragonmesh(object)`)that computes the geometry and material of the 3D object. The geometry of the bunny is computed using `object.children[0].geometry.clone()`. It centers the bunny at position (0,0,0) being in the same place as the box and then calls a function that scales the bunny to fit in the box `resize()`. `resize()` computes a bounding box for the bunn and one for the cube. this is necessary in order to find the scaling ration that needs to be performed on the bunny to actually be scaled. In order to find the scaling indices, the cube size is divided by the bunny size. then we compute the minimum between the (x,y,z) values of the bunny. in this way the 3D model will not be just stretched to fit in the cube on every axes, it will just be scaled without changing the shape.
The scaling is done just once which is important for the next requirement. It is more efficient to scale just the geometry and then use it later on for the other rendering modes.
The material used is MeshStandarMaterial. Using the bunny material and geometry we compute the mesh and display it on the scene when B key is pressed. The following pictures and code show the implementation described above.

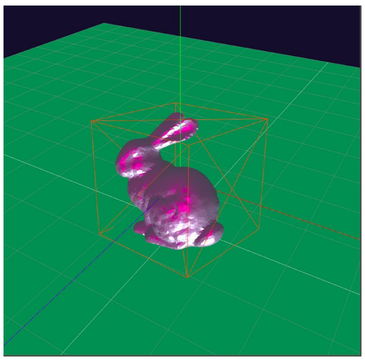


fig 8.1 (the bunny is scaled to fit in the box)

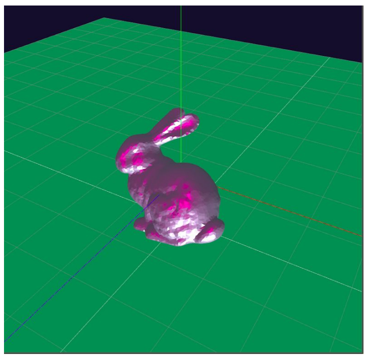


fig 8.1 (the bunny is translated to be in the same position as the box)

```
init(){
…
        var loader = new THREE.OBJLoader();
        loader.load('bunny-5000.obj',dragonmesh);
}
function resize(){
        bunny_geometry.computeBoundingBox();
        cube.geometry.computeBoundingBox();
        var bunny_size = bunny_geometry.boundingBox.getSize();
        var cube_size = cube.geometry.boundingBox.getSize();
        var bunny_scale = cube_size.divide(bunny_size);
        var bunny_scale_ratio = Math.min(bunny_scale.x, bunny_scale.y, bunny_scale.z);
        bunny_geometry.scale(bunny_scale_ratio,bunny_scale_ratio,bunny_scale_ratio);
}
function dragonmesh(object){
        bunny_geometry = object.children[0].geometry.clone();
        bunny_geometry.center();
        resize();
        //face
        var material=new THREE.MeshStandardMaterial({color: 0x7f8d9c} );
        bunny = new THREE.Mesh( bunny_geometry, material );
}
```
code 8

Testing: In order to do the testing for this requirement I allowed for the wireframe and bunny to be added to the world at the same time, in figure 8.1 the testing is successful.

9. Rotate the mesh, render it in different modes

***Explanation**:* Rotate the mesh about the x, y, z aes and render it in different modes. This process goes very similar with the one used for the cube.
***Implementation**:* I am computing the materials for the vertices and edges modes in the same dragonmesh() function using a single geometry (the one used in req 8). I am computing every mesh individually and add them using keyboard shortcuts (M for vertices and N for edges). the rotation is done just as for the cube. More lightning is added to emphasise the face rendering mode of the mesh (a point light). (see *code 9.1 due to sace reason i will not be able to put more code, but everything is similar to the cube rendering modes*) The following pictures represent the 3 render modes of the bunny.

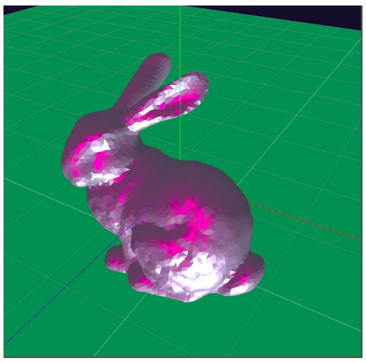


fig. 8.1(face mode)

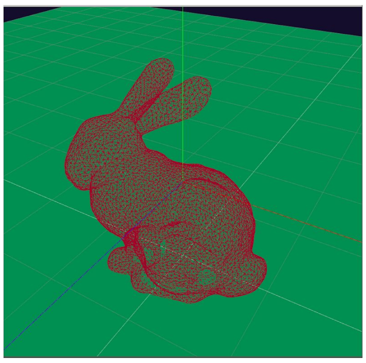


fig. 8.2(edge mode)

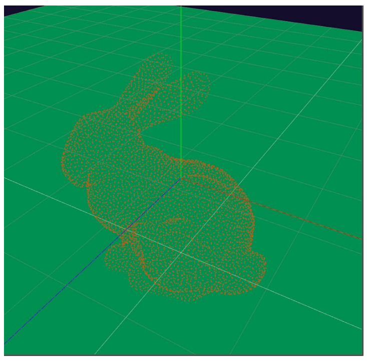
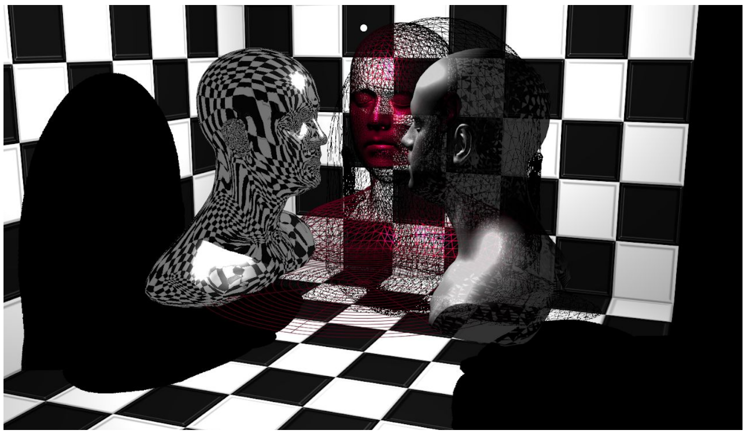


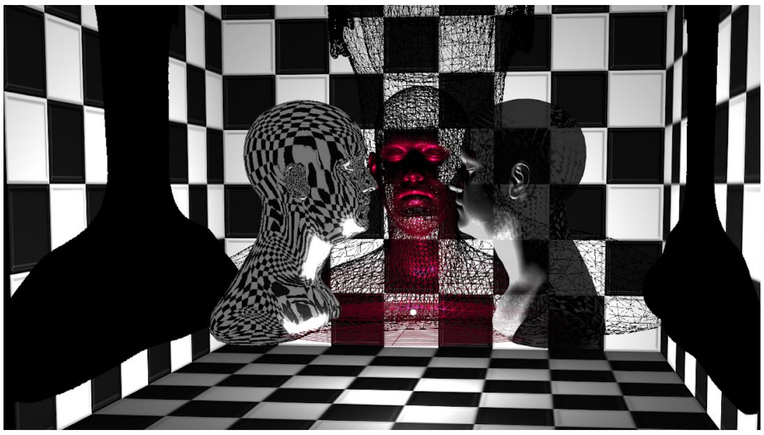
fig. 8.3 (vertex mode)

Testing: In order to do the testing for this requirement I checked if all the key shortcuts work properly. The figures are concludent.

10. Be creative – do something cool!

Explanation*:* I have created the scene from fig. 10.1, 10.2





Implementation*:* The design is formed by the tree figures rendered in different modes, a point light that moves up and down casting light and creating moving shadows on the walls, the chess room and a polar grid. I decided to make use of a few of the functionalities provided by the mesh standard material constructor. For the first Smith dead I decided to go for a phantomatic appearance. This was achieved by setting the transparency to true and roughness to 0.65 while the opacity to 0.8. The roughness allows the light to be reflected or not while the opacity makes an object transparent if set to 0 and completely opaque at 1. All of these are features of the mesh standard material. For the second Smith head I decided to make the opposite of the fist one. Nothing can get through and everything is reflected. I decided to set the metalness to 1 in order to give the illusion of complete opposite textures. in order to make the material of the second head to reflect the chess room i had to load the texture and than use envMap to map it on the 3D object. All the heads have the geometry translated and rotated to form the triangle. All the meshes sit on a polar grid and inside a room, i have used a similar technique to the one used fot texturing the cube but I added the texture inside rather than on the inside using `materialArray[i].side = THREE.BackSide`. The light is represented by a small sphere that uses sphere geometry. I have used different properties of the point light such as `emissiveIntensity or power` to make the light suitable for the chess room. After this I made the light, heads and the box to cast shadow. The walls were also receiving shadows from the heads. A very interesting effect is created if the first head is receiving shadow, but it slows the rendering. Some of the code of the light from animation frunction is shown below:

```
var time = - performance.now() * 0.0004;
               bulbLight.power = 32;
               bulbMat.emissiveIntensity = bulbLight.intensity / Math.pow( 0.02, 2.0 );
               bulbLight.position.y = Math.cos( time ) * 0.75 + 0.5;
```

DISCUSSION

The code includes a lot of repetition and it can be optimized better. the robustness is definitely missing and the report should have went into more detail regarding the mathematics behind a few processes. Further research should be put into choosing the most useful three.js functionalities.

REFERENCES

three.js/documentation
https://en.wikipedia.org/wiki/Phong_shading
https://webglfundamentals.org/webgl/lessons/webgl-2d-rotation.html#toc