CM20219 WebGl-report

Chibeze J. Nwangwu

I. Introduction

For CM20219 coursework2, we worked with three.js, which is a cross-browser JavaScript library used to create and display animated 3D computer graphics in a web browser using WebGL. The report will follow a simple structure: (1) Explanation of each requirement and their implementation; this section will include code snippets, mathematical back-grounds and also discussions and testing. (2) this section will focus on the creative requirement, the discussion and other feature I have implemented. To begin viewing and manipulating the scene, the following objects are required. The scene, the camera and the render-er luckily in the skeleton code all these were provided.

II. Requirements

A. *Draw a simple cube Req1*

We can split req1 into three stages of implementation. Stage1: creating the object geometry and material, for this stage I have used the BoxGeometry from the geometry class. I used the *boxGeometry(2,2,2)* constructor to create a cube of length 2, and *MeshBasicMaterial()* to create a basic material. By default, the cube is centered at the origin *(0,0,0)* as well as having a rotation vector *(0,0,0)*. Stage2: creating a triangular polygon mesh-based object using THREE.mesh(), the mesh constructor takes two parameters *Mesh(geometry, material)*. The geometry defines the shape of the mesh, and the material defines the material of the mesh. The material we chose will affect how the mesh interacts with light in the scene, for this reason in **II-D** Req. 4; I will be changing the cube material to *MeshStandardMaterial()* as I implement other light sources in my scene. Stage3: adding the cube to the scene, here we add the new mesh to the scene, where a scene is an object used to specify what and where is to be rendered by three.js.

```
// TO DO: Draw a cube (requirement 1).
var geometry = new THREE.BoxGeometry( 2, 2, 2 );
var material = new THREE.MeshBasicMaterial( {color:
    0xffff00} );
cube = new THREE.Mesh( geometry, material );
scene.add( cube );
```

To see the resultant cube see Fig.1
Discussion: for further understanding on how boxGeometry works, we can look at the boxGeometry code. BoxGeomerty actually calls BoxBufferGeometry, which creates the box using buffers (indices, vertices, normals and UVs), which are essentially arrays. The *boxBufferGeometry* class then calls *buildpanel()*, which builds each panel (face) of the box.

B. *Draw coordinate system axes Req2*

This requirement required us to build the world axes using lines, as well as colouring the axes RGB, for x, y, z respectively. Luckily the *AxesHelper* class [4] which extends *LineSegments* does this for us. The class uses *bufferGeometry()* and *lineBasicMaterial()* to create the axes. Code can be seen below.

```
// TO DO: Visualise the axes of the global
    coordinate system (requirment 2).
axisHelper = new THREE.AxesHelper( 4);
scene.add( axisHelper );
```

C. *Rotate the cube Req3*

For this requirement, we were asked to rotate the cube about each axis (X, Y, Z) with the world axes, and the camera remains fixed. The solution involves (1) handling keyboard input, (2) rotating the cube and animating the rotation and (3) resetting the rotation.
(1) Handling keyboard input is quite straight forward using *document.addEventListener(keydown, function)*. Inside the *function* I use a switch case to determine which key is pressed and subsequently what rotation takes place. The code snippet below shows one case.

```
// TO DO: add code for starting/stopping rotations
    (requirement 3).
case 88: // x key
  if (rotating && axes == 'x') {rotating = false;
      break;}
  axes = 'x';
  if (!rotating) {rotating = true, animateRotation
      ();}
  break;
```

The if statements are used to start and stop the cube's rotation at any point. Essentially, each (x, y, z) key turns on and off rotation for their specific axes.
(2) For rotating the cube, I have used Quaternions as opposed to using *.Rotation* to represent spatial rotation. Quaternions are used to represent rotations in three.js and are unaffected by gimbal-lock. In this case, Quaternions allows for rotation about a fixed X, Y, Z axes, this means that I can perform a combination of rotations, for example, I can perform a rotation about the Y-axis (without changing the direction of the z-axis), then perform a rotation about the Z-axis. With Quaternions, I can apply an axis angle rotation using a simple vector to delineating the axis and an angle to delineating the angular increment per frame. See code snippet below.

```
function animateRotation() {
  if (rotating){
    requestAnimationFrame(animateRotation);
    const rSpeed = 0.02;
    const quant = new THREE.Quaternion();
    var vector;
    if (axes == 'x' )
      vector = new THREE.Vector3(1, 0, 0);
    if (axes == 'y' )
      vector = new THREE.Vector3(0, 1, 0);
    if (axes == 'z' )
      vector = new THREE.Vector3(0, 0, 1);
    if (axes == null)
      vector = new THREE.Vector3(0, 0, 0);
    // rotates the object about the vector given
    quant.setFromAxisAngle( vector, rSpeed);
    cube.applyQuaternion(quant);
  }
  renderer.render(scene, camera);
}
```

See Fig.1 for the result. Regarding the rotation animation I have used *requestAnimationFrame(rotationAnimation)* to call the animation almost recursively. RequestAnimationFrame is derived from the web-API in the window class. I have wrapped this request within an if statement, allowing me to stop and start the rotation animation as opposed to using *cancelAnimationFrame(id)*. I am essentially using an if statement to produce the same result as cancelling the animation.

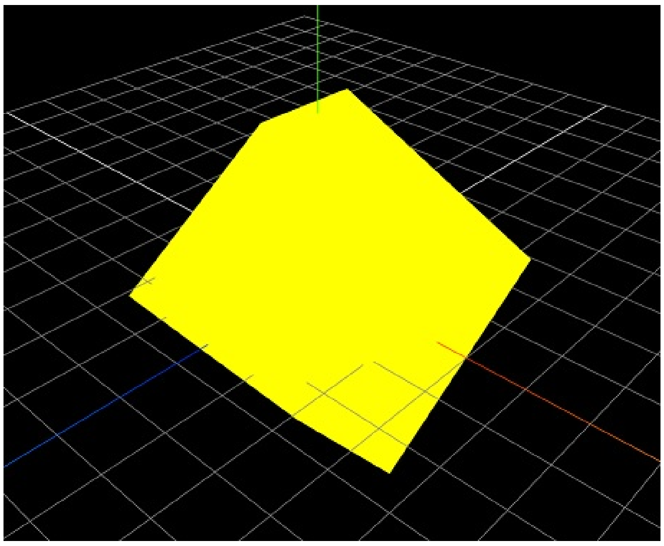


Fig. 1: Rotation of cube about axis (RGB)

(3) Resetting the rotation requires a key-down handler; in my case the event is when the R key is pressed. Pressing the R key sets the rotation of the cube to (0,0,0) and sets the axes state to null. See code snippet below.

```
case 82: // r  = reset
cube.rotation.set(0, 0, 0);
// sets axes state to null
axes = null;
break;
```

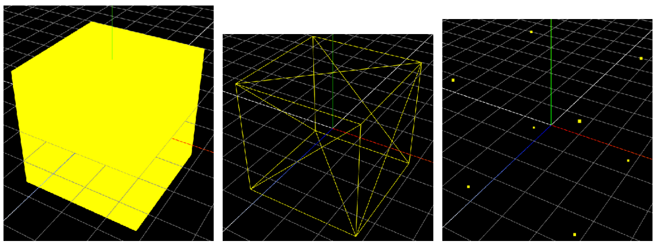
Discussion: Assuming that the users expect the axes of rotation to remain constant using *object.rotation.axes += angle* would not produce the desired results as a rotation with this method may change the direction of the axes. As discussed in the (Q&A) forum and gathering from rotation represented in the demo, we are to rotate the cube whilst maintaining fixed local axes. I have made an assumption here as the requirement were not exact as to what axes are to remain fixed (the world or the local).

D. *Different render modes Req4*

For this requirement, we were asked to implement different render modes (vertices, edge and face modes) for the cube. For this requirement, I created three different materials; (1) a face material, (2) edge material and (3) a vertices material. For the (1) face mode I decided to use *MeshStandardMaterial()*, which is a physically-based material, i.e. it allows for Physically based rendering (PBR). PBR has become quite popular in the past years, so I wanted to use something that is currently viable in the community. Regarding the lighting, I have used directional lighting for the face rendering mode. (2) Edge mode was done using the basic material and required setting the material's wireframe mode to true; see code snippet below. I used basic material as I did not want the wireframe to be affected by light, and lastly (3) vertices mode; this mode involved *PointsMaterial()* from the points class in three.js, the points are rendered using gl.POINTS [9], in my implementation points material, took parameters like the size and the color; see code snippet below. I want to render the points and not the faces of the cube, so I am using *THREE.Points(geometry, material)* rather than *THREE.Mesh(geometry, material)* to create the object. And as usual, the default position of an object upon creation is the origin *(0,0,0)* in Cartesian coordinates.

```
// TO DO: mode rendering (requirement 4).
var geometry = new THREE.BoxGeometry( 2, 2, 2 );
var material = new THREE.MeshStandardMaterial( {
    color: 0xffff00} );
cube = new THREE.Mesh( geometry, material );
scene.add( cube );
// draw the edges
var eMaterial = new THREE.MeshBasicMaterial( {
  color: 0xffff00, wireframe: true } );
wireframe = new THREE.Mesh( geometry, eMaterial);
// draw the vertices
var vMaterial = new THREE.PointsMaterial( {
  size: 0.1, color: 0xffff00});
points = new THREE.Points( geometry, vMaterial );
```

Below Fig.2 shows the three different rendering modes.



(a) Face mode (b) Edge mode (c) points mode.

Fig. 2: Three different mode rendering

To alternate between render modes I have used the keys *F, E, V*, where in the event that F key is pressed, I remove the other objects (edge and vertices object) from the scene, using the *scene.remove(obj)* instruction, and add the cube to the scene using *scene.add(obj)*. Below is a code snippet:

```
case 70: // f = face
  // alert('TO DO: add code for face render mode (
      requirement 4).');
  scene.add(cube);
  scene.remove( wireframe );
  scene.remove(points);
  break;
```

Another objective 0f this requirement is that the object should maintain it's rotation about the axes whilst changing the rendering mode. for this requirement I have added to the animateRotation function discussed in **II-C** Req. 3. I have added the following line in the animateRotaion() function defined in the **II-C** Req. 3.

```
// rotates the object about the vector given
cube.applyQuaternion(quant), wireframe.
    applyQuaternion(quant), points.applyQuaternion(
    quant);
```

Discussion: There are multiple approaches for this requirement; one approach would be to use the same mesh (the cube in this case) and add all the other objects as a child to the mesh. Then changing between mode requires manipulating the material's state, for example turning on the wireframe mode using *cube.material.wireframe = true;* and then making all the faces in-visible. This approach is more elegant, than my chosen method.

E. *Translate the camera Req5*

For this requirement, we were asked to translate the camera along the x,y,z axes. I have used the camera's matrices to implement the solution. The idea was that by using the camera's matrix, I could change it's a location without changing it's orientation, in addition to changing the look-at position appropriately. Before moving on, let's introduce the concept of matrices and their relationship with 3d objects. 'A matrix is an array of numbers with a predefined number of rows and columns' [6]. Matrices are used to define the properties of a 3D object in space; matrices hold the scale, orientation, and position properties of any object in space. Now if I want to move an object in space I would need to scale the object then rotate it and then apply the translation: *transformed vector = translation*rotation*scale*originalvector*. Using this principle, I can translate the camera. Camera matrix holds the orientation, scale and position vector. Using the web console, I was able to retrieve the initial camera matrix. Camera matrix:

$$\begin{bmatrix} x-R-S & y-R-S & z-R-S & posVect \\ 0.857 & -0.291 & 0.424 & 3 \\ 0 & 0.825 & 0.566 & 4 \\ -0.514 & -0.485 & 0.707 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where x-R-S = x rotation and scale.
posVect = position vector.
Solution: if i want to translate along the x axis i have to take the first three rows from the first column in the camera's matrix; this is a direction vector containing the scale and orientation of the camera with respect to the x-axis, then perform a scalar multiplication to get my transform vector, then add this transform vector to my camera's position and look-at. see code below:

```
var panOffset = new THREE.Vector3();
case 39: //> translate left
  var v = new THREE.Vector3();
  v.setFromMatrixColumn( camera.matrix , 0 ); // get
      the x column
  v.multiplyScalar(.5);
  panOffset.add(v);
  break;
```

```
var target = new THREE.Vector3();
function updateCam() {
    var offset = new THREE.Vector3();
  var position  = camera.position;
  offset.copy(position).sub(target);
  target.add( panOffset );
  position.copy( target ).add( offset );
  panOffset.set( 0, 0, 0 );
}
```

I have, however used *camera.translateZ(1);* to move forwards and/or backwards. This is because these translations don't

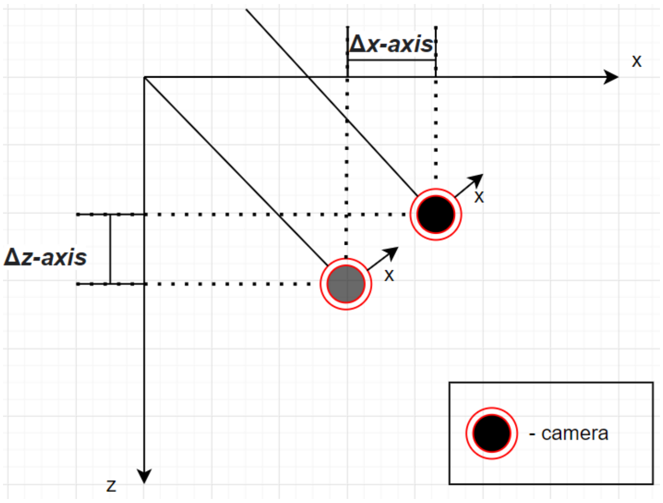


Fig. 3: camera.translate.x diagram

change the camera's look-at therefore i dint need to calculate a new look-at after every translation.

```
case 187: // + zoom in
  camera.translateZ(-1);
  break;
```

For handling input i have used the arrow keypad to pan up and right, and the plus and minus key for forwards and backwards.
Discussion: Originally, I used *camera.translate* to translate the camera with respect to the camera's local axes. I wasn't able to get it working correctly because the camera's axes may not be perpendicular to any of the world axes. Therefore translating along camera's local x-axis may offset the camera along multiple world axes of which I don't know their offset. Hence, I was not able to easily calculate the new look-at position as I don't have the transformation vector. See the diagram below for a better idea of the problem.

F. *Orbit the camera Req6*

For this requirement, we were asked to orbit the camera about the cube with the look-at position remaining fixed. Essentially to implement something similar to three.js orbitcontrols, and of course without using orbitcintrols. I have completed this task with the help of *spherical*, using spherical itself is quite important as the arc ball orbit must not experience Gimbal-Lock. To achieve this, I have split the task into three sub-tasks; (1) handle mouse input, (2) handle rotation (up and right rotations). and lastly (3) updating the camera's position.
(1) Handling the mouse input requires three event handlers, these events are the mouse down, mouse up and mouse move events, with these three handlers I can calculate my mouse displacement (delta) and call the rotation functions. See code snippet below for a better view of the solution.

```
var down = false , var start = new THREE.Vector2();
var end = new THREE.Vector2();
function onMouseDown(event){
  down = true; start.set(event.clientX , event.
      clientY);
}
function onMouseUp(event){
  down = false;
}
function onMouseMove(event){
  if (down) {
    end.set(event.clientX , event.clientY);
    delta.subVectors(end, start);
    // rotating across whole screen goes 360
        degrees around
    rotateLeft(2 * Math.PI * delta.x /renderer.
        domElement.width );
    // rotating up and down along whole screen
        attempts to go 360, but limited to 180
    rotateUp( 2 * Math.PI * delta.y / renderer.
        domElement.height );
    start.copy( end );
    }
}
```

Explanation: with this code camera only orbits when the mouse moves whilst being down, and in my case any button on the mouse can be used to activate the orbit. In function *onMouseMove* I call *rotationUp* function and pass in the equation $2*PI*(delta/total)$ as a parameter, this equation is used to calculate the angle as a ratio of 2*PI, where the ratio depends on the distance moved whilst mouse is down. (2) For rotation orbit i have used two simple functions, rotateup and rotateleft. theses functions are used to update *spherealdelta* variable, this variable will be used to update the camera's orbit inside the *camUpdate()* function.

```
var sphericalDelta = new THREE.Spherical();
function rotateLeft( angle ) {
  sphericalDelta.theta -= angle;
}
function rotateUp( angle ) {
  sphericalDelta.phi -= angle;
}
```

(3) For updating the camera position, I call the *camUpdate* function in the animate function (loop). The update function uses spherical from the three.js lib to update the location of the camera. In the update function, I essentially copy and convert the camera's position (Cartesian coordinates) to spherical coordinates, using *spherical.setFromVector3(offset);* . Then add the spherical delta which has already been incremented in the rotation-up and rotate-left function to the spherical coordinates, where phi represents the vertical orbit and theta represents the horizontal orbit. After we have the new spherical coordinates, we now want to update the current camera's position. We do this by converting the new spherical coordinate back to Cartesian coordinates, using *offset.setFromSpherical(spherical);* and then updating the camera's position and look at point using the new coordinate Cartesian coordinates. The last step is to reset our delta and offset coordinates. These steps are simple and can be easily implemented with the help of some important functions for the three.js lib (these important functions are mentioned in the Discussion below).

```
var target = new THREE.Vector3(), spherical = new
    THREE.spherical();
function updateCam() {
  var offset = new THREE.Vector3();
  var position  = camera.position;
  offset.copy(position).sub(target);
  spherical.setFromVector3( offset );
  spherical.theta += sphericalDelta.theta;
  spherical.phi += sphericalDelta.phi;
  // restrict theta to be between desired limits
  spherical.theta = Math.max( -Infinity , Math.min(
      Infinity , spherical.theta ) );
  // restrict phi to be between desired limits
  spherical.phi = Math.max( 0, Math.min( Math.PI ,
      spherical.phi ) );
  spherical.makeSafe();
  // move target to panned location
  target.add( panOffset );
  offset.setFromSpherical( spherical );
  // update the camera position
  position.copy( target ).add( offset );
  camera.lookAt( target );
  sphericalDelta.set( 0, 0, 0 );
  panOffset.set( 0, 0, 0 );
}
```

Discussion: working with spherical coordinates in three.js is quite intuitive, at least for this task. The important functions that make this implementation an intuitive solution would be the conversion functions *setFromVector3() and setFromSpherical()* these functions are essential as they allow for seamless conversion from the Cartesian to the spherical coordinate systems. Upon finding this function, I was in the process of creating my versions, which had a lot of bugs. I have learnt that spherical coordinates provide a simple and intuitive form of orbiting about a 3d space.

G. *Texture mapping Req7*

For this requirement, we were asked to apply texture to the cube, each face should have a different texture, and the texture should be correctly mapped to the cube without being skewed. I have achieved this by using the Texture loader function from three.js lib. This function internally implements *loader.js* and *ImageLoader.js* from the three.js lib, using them load image files of various types. For the solution, we can build each face of the cube individually using a multi-material array then map the loaded images to each material. We then pass this array into the mesh constructor when creating the cube mesh. See the code snippet below for the implementation.

```
var geometry = new THREE.BoxGeometry( 2, 2, 2 );
// loading texture
var loader = new THREE.TextureLoader();
loader.setPath( 'texture/' );
var cubeF = [
  'f1.png', 'f2.png',
  'f4.png', 'f3.png',
  'f5.png', 'f6.jpg'
]
var fMaterial = [
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[0])}),
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[1])}),
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[2])}),
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[3])}),
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[4])}),
  new THREE.MeshStandardMaterial({  map: loader.
      load(cubeF[5])})
];
//create the triangular polygon mesh and add it the
    scene
cube = new THREE.Mesh( geometry, fMaterial );
scene.add( cube );
```

resulting cube can be seen in figure 4.

Discussion: In this requirement, the ability to pass in a material array into the Mesh constructor is vital as it makes the whole process simple and intuitive. With further research into textures, I learned that texture mapping could happen in two ways projection mapping and UV mapping. I learned that projection mapping is " widely used for things like creating shadows" [2], UV mapping is used to map images to more complex geometries, for example, mapping a face texture to a 3D-face geometry, it does this by using points by providing a connection between points on the image and points on the geometry.

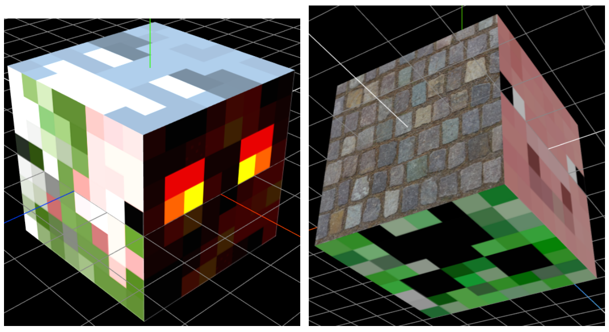

(a) view from camera position (3,4,5) (b) view from camera position (-3,-4,-5)

Fig. 4: Req. 7 texture mapping, showing 6 different faces

H. *Load a mesh model from .obj Req8*

For this requirement, we were asked to load and display the Stanford bunny object. The object should be uniformly scaled and translated to fit inside the cube. I have done this using objectloader; this is "a loader for loading a JSON resource in the JSON Object/Scene format" [7]. After creating an instance of objLoader, I called the load function and passed the path to the bunny.obj file and a callback function as parameters. This function takes the loaded object and traverses all the children of the object. After the geometry has been defined, I recenter the bunny using *_3dObj.geometry. center()* this function Centers the geometry based on the bounding box. I do this because initially, when I added the bunny to the scene, it has a predefined centre location which was not the desired center location. Scaling the object is quite simple; I have used the *object.scale.set(x, y, z)* instruction to uniformly scale the object along the x, y, and z-axis. Lastly making sure the object fits into the cube I have done this through trial and error. When I used the bounding box to scale the bunny, I would still be able to see parts of the bunny when the cube is rotating and the bunny is not rotating. This is because, at a certain angle, the bunny does not fit perfectly into the cube, and so to address this, I have picked a reasonable scale as a constant. Since the model is centered and since by default objects added to the scene are position at the origin, I need not translate the bunny to fit into the cube. Below is the code snippet of the implementation and a figure (Fig. 5) displaying the resultant bunny with the cube in edge mode.

```
// load the 3D object in to the scene
var _3dLoader = new THREE.OBJLoader();
const scale = .4;
_3dLoader.load("model/bunny-5000.obj",
// called when resource is loaded
function(object){

  object.traverse(function(child){
    if (child instanceof THREE.Mesh ){
      _3dObj = child;
    }
  } );
  // re-center the loaded object
  _3dObj.geometry.center();
  // scale to fit the cube with abituary uniform
      scale
  _3dObj.scale.set(scale, scale, scale);
  // adds the model to the scene
  scene.add( _3dObj );
});
```

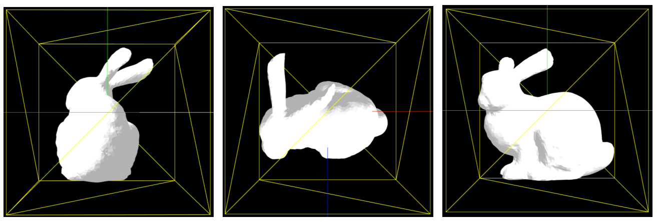

(a) view from camera position (5, 0, 0) (b) view from camera position (0, 5, 0) (c) view from camera position (0, 0, 5)

Fig. 5: Req. 8 Bunny view form x, y, and z axis, bunny inside the cube whilst cube in edge render mode

Discussion: I would like to discuss the traverse function. The traverse function is necessary for the loading of an object; this is because the traverse function visits every child in the object and virtually ensures that the object is an instance of a mesh. I enjoyed looking at this function because it uses callback functions which I had not seen before in any language. Callback functions are pretty cool; they allow you to pass a function as a parameter of another function. To me, this idea was fascinating. Callback functions are mostly used for asynchronous functions, which are functions that are running parallel to another function.

I. *Rotate the mesh, render it in different modes Req9*

For this requirement, we were asked to rotate and display the mesh with different rendering modes; modes include edge, face and vertices mode. These objective have already been done before for the cube mesh, so implementation this time for the bunny mesh shouldn't be much different. For rotating the bunny, I have decided to rotate the bunny using quaternion as I did for the cube in **II-C** Req.3. I have programmed the bunny to rotate on the same button press event as the cube; therefore, the bunny and the cube will rotate together. For the implementation I have added the line shown in the code snippet below into my *animateRotation()* function defined in **II-C** Req. 3.

```
// Req, 9 bunny rotate about the x, y, and z axis
_3dObj.applyQuaternion(quant), _3dPoints.
    applyQuaternion(quant), _3dWireframe.
    applyQuaternion(quant);
```

Now Regarding the rendering modes (face, edge and vertices). I have used the same edge material to render the cube and bunny's edge modes (*eMaterial* see first code snippet in **II-D** Req. 4). I have created a new point material for the bunny's vertices mode, mainly because the point material used for the cube vertices mode was too big. I needed points of a smaller size to get the desired aesthetic for the bunny's vertices mode. For creating the mesh, I pass in the bunny's geometry and edge material to the mesh constructor. I also create the point object in the same way; by passing the bunny's geometry and points material into a points constructor (see code in the code snippet below). The resultant objects can be seen in figure 6.

```
// edge mapping on the 3D object
_3dWireframe = new THREE.Mesh( _3dObj.geometry ,
    eMaterial)
// vertecies of 3D object
var pMaterial3D = new THREE.PointsMaterial( { size:
    0.01, color: 0xffff00});
_3dPoints = new THREE.Points( _3dObj.geometry ,
    pMaterial3D )
// scale to fit the cube with abituary uniform
    scale
_3dObj.scale.set(scale , scale , scale);
_3dWireframe.scale.set(scale , scale , scale);
_3dPoints.scale.set(scale , scale , scale);
_3dObj.material.flatShading = true;
scene.add( _3dObj );
```

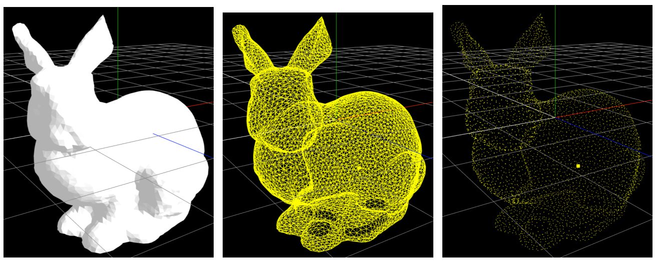

(a) face mode (b) edge mode (c) points mode

Fig. 6: Req. 9 three different rendering modes

J. *Be creative – do something cool! Req10*

For this requirement, we were asked to go beyond the previous tasks and do something cool, anything really. I have decided to make an *Art Space*, which is, in my definition, a space for displaying artwork. Displaying 3D art pieces is a novel front that has picked up some professional attention, especially in VR [10]. So for this requirement, I have played around with lights in three.js, lighting is crucial in fine art, it gives the image structure and depth [3]. But in 3D space we already have depth so now what can we use lighting for, well in this case we use lighting to create illusions and effects, be it with shadows, reflection and or a combination of both. Lighting can be instrumental in distorting perception and so can be used to create something cool. To call something cool we press the 'space-bar' this will call the art-space function. During the space-bar event the scene is cleared, by removing all it's children (using *.clear()* function in the object3D class), then the *artSpace()* function is called. In this function, new objects are added to the scene and rendered. For my something cool I have two boxes, these boxes hold my art pieces, the first box from the left holds the head art, and the second box holds a mini Minecraft terrain. For the head art, I have used four different light sources; these are the spotlight, HemisphereLight and two point lights. I have also loaded a new object with a head geometry. My composition consists of orbiting light sources and orbiting bunny and cube mesh (mesh gotten from Req. 1 and Req. 8) and a head object. To get the desired effect, I have rendered the head object in wireframe mode and have enabled shadow casting and shadow receiving on all these objects. To see inside the cube, where the art is located, I have inverted the cube surface by passing *side: THREE.BackSide* as a parameter in the Mesh standard material constructor, this has a cool effect when look into the cube from the outside. see Fig. 7 for the result.

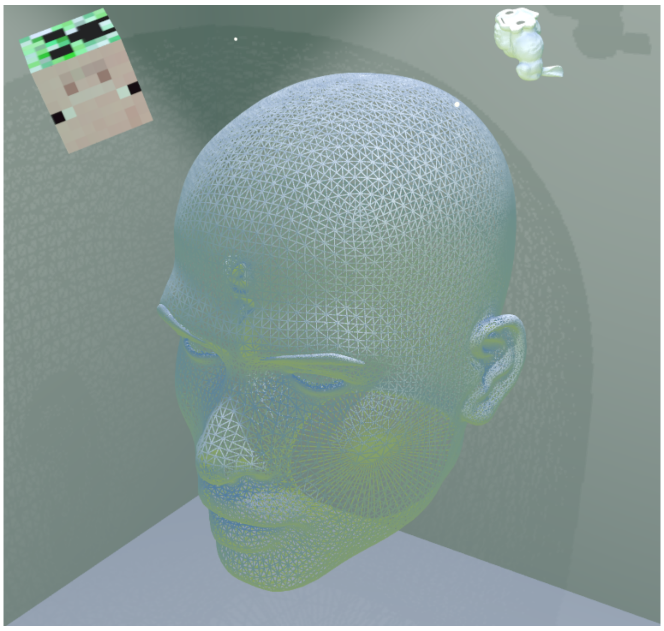


Fig. 7: Req. 10 something cool head art, with objects orbiting about the head, bunny, cube and two point lights

To orbit the objects about the head, I have used some maths. Orbiting an object about the head required me to use the idea of oscillations, in principle, I wanted the object position to change in a way that it outlines a circular path about a given point, to do this I used the sine and cos functions. I have used cos function to oscillate along the x-axis and sin function to oscillate about the z and y-axis see code snippet below for implementation.

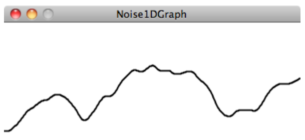
```
// maths delineating objects orbit about the head
cube.position.x = Math.cos(t/720)*2.5 ;
cube.position.z = Math.sin(t/720)*2.5;
cube.position.y = 2-Math.sin(t/900);
pointLight.position.x = Math.cos(t/720 - 1.5)*2.5;
pointLight.position.z = Math.sin(t/720 - 1.5)*2.5;
pointLight.position.y = 2-Math.sin(t/900);
...
```

I have displaced objects to different positions on the orbit. To do this I have to shift the period [5] using (*Math.cos(t/720 - 1.5)*2.5*) as shown in the code snippet above.
In the second box on the right in the something cool scene, I have implemented a mini-Minecraft terrain. I have created this using a list of thing including, Perlin noise, a random number generator and a lot of boxes (box mesh used here are dubbed blocks). Ken Perlin developed Perlin noise in the 1980's whilst he was working on the Tron movie, 'Perlin noise has a more organic appearance because it produces a naturally ordered ("smooth") sequence of pseudo-random number' [1]. To see the difference between Perlin random number generation and a standard random number generator see figure 8. Minecraft terrains are rendered smoothly and randomly; this gives the terrain an organic look as we can have mountains, valleys and flat terrain. If I am to replicate this, I will need to use perlin noise.

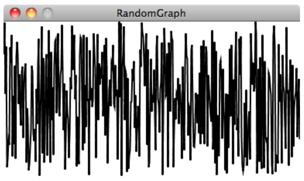
The idea behind my implementation is to use random y values for each block to create the desired terrain. To randomise the y position for each block, I have used perlin noise, as shown in the code snippet below. Firstly I seeded the perlin noise seed function with a random number; this is important as it will mean that the terrain will look different every-time the *artSpace()* function is called. The next step is to create many blocks. The syntax here is quite peculiar as I am pushing whole functions into an array. For this step, I am pushing the *block(x, y, z)* function into the blocks array.

```
// seed the noise random nunmber generator
noise.seed(Math.random());
var blocks = [];
var xoff = 0;
var zoff = 0;
var inc = .1;
var height = 5;
for (var x = size +1; x < size*2 - .5; x++){
  xoff = 0;
  for (var z = -size/2 + 1; z < size/2 - .5 ; z++){
    var v = Math.round(noise.perlin2(xoff,zoff) *
        height);
    blocks.push(new block(x, v + (size*-1)/2 +
        height, z));
    xoff += inc;
  }
  zoff = zoff + inc;
}
for(var i = 0; i < blocks.length; i++){
  blocks[i].display();
}
```

The blocks are created in the loop at the end of the code snippet above. For each block function in the blocks array we call the *display()* function located in each *block()* function. The display function builds the box geometry, loads the texture images with *tesxtureLoader*, maps each image to a face in the box geometry, constructs the mesh with the *THREE.Mesh()* constructor and then adds the block mesh to the scene. The results can be seen in figure 9.



(a) perlin noise over time



(b) pure random number over time

Fig. 8: perlin noise (left) vs randomGen() (right). *images gotten from khan academy [1]*

Discussion: For this part of the coursework, I had so many ideas, but I have decided to go for the art-space because I wanted to do something that mixes elegance and complexity. My initial idea was to create three art boxes to hold three ideas. Unfortunately because of time, I was only able to implement two of boxes. Furthermore, with some regret, I also was not able to complete the Minecraft box due to a problem with text geometry and loading a font.TTF file. Otherwise, the complete Minecraft box would have included the word 'Minecraft', which would have been 3D rendered using text geometry and in the Minecraft font style. The word would be positioned to hover above the minecraft terrain, with an animation that oscillates the word along the y-axis. The third box that was not in the final program; this box would have explored light probes and environment mapping; an example of light probes can be seen in the three.js example [8].

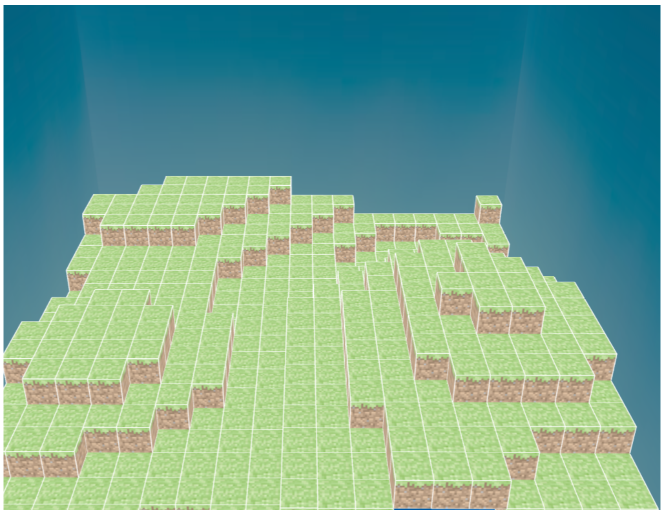


Fig. 9: Req. 10 something cool minecraft terrain in a box with sky image mapped to the inside of the box

K. Extra things Req11

In the skeleton code, the automatic resize feature seemed not to be working so I have fixed this; originally the window of interest, i.e. the scene, would only resize when the refresh button is pressed, this is not normal and certainly not desirable. The issue was that the event listener was being added to the document class rather than to the Window class. Solution: add the event handler to the window class to do this I can just call *addEventListener('resize', handelResize)*. I have added some extra features, like the ability to turn on and off the grid and some other cool camera orbiting animation. See the Readme.txt file for the key bindings

III. Final Discussion

Lesson learned: I was able to create and render cube mesh, apply texture to a mesh, perform and control animation on objects (i.e. rotation animation), orbit camera about the scene, using event handlers and loading and manipulating a 3D models in the scene. Overall, I have now learnt how to program in JavaScript and latex. The limitations are: that the mouse buttons have not been individually mapped to a specific event, dolling in and out can be made safer by setting limits (i.e. if you dolly the camera into the origin you get a glitching effect on the camera view), loading lag is a real problem when loading the art-space; so the program has a lot to improve in terms of performance, and lastly, the program is not able to go back to the cube view form the something cool. For future works I will map the dolly event to the mouse scroll button, I would also allow the users to translate the cube by dragging and drop it with the right mouse button as well as completing the Minecraft and light Probe box.

References

[1] Khan Academy. perlin noise. https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise.

[2] Lewy Blue. *A Brief Introduction to Texture Mapping*. 2020. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/.

[3] Mark Mitchell. Is lighting important in fine art?, 2016. https://www.markmitchellpaintings.com/blog/is-lighting-important-in-fine-art/#:~:text=Lighting%20gives%20a%20painting%20structure,how%20our%20eyes%20perceive%20it.

[4] Mr.doob mrdoob. Axeshelper.js, 2020. https://github.com/mrdoob/three.js/blob/master/src/helpers/AxesHelper.js.

[5] Thuy Nguyen. Exploring y = a sin(bx + c). http://jwilson.coe.uga.edu/EMAT6680Fa07/Nguyen/Assignment1-TN/Writeup1.html#:~:text=In%20the%20equation%20y%20%3D%20a,displacement%20of%20the%20sine%20curve.&text=We%20note%20that%20for%20the,now%20%5B%2D3%2C3%5D.

[6] opengl tutorial.org. Tutorial 3 : Matrices. http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/.

[7] three.js.org. Objectloader. https://threejs.org/docs/#api/en/loaders/ObjectLoader.

[8] three.js.org. webgl lightprobe. https://threejs.org/examples/?q=light.

[9] three.js.org. three.js docs, 2020. https://threejs.org/docs/.

[10] Times USA. vr is for artists, 2020. https://time.com/vr-is-for-artists/.