WebGL Report

Requirement 1

To show anything on the screen we must have a structure for the shape (its geometry) and a material to be applied on the geometry: a mesh mixes a geometry and a material into a visible shape. A geometry is comprised of vertices (the points) and the faces (triangles made using three vertices). A simple cube has 8 vertices and 6 square faces: hence 12 triangle faces. We are asked to draw a cube at the origin with corner points being (-1, -1, -1) and (1, 1, 1).

```
// Make a new geometry
geometry = new THREE.Geometry();
// Adding the corner vertices
geometry.vertices.push(
   new THREE.Vector3(-1, -1,  1),
   . . .
   new THREE.Vector3( 1,  1, -1),
);
geometry.faces.push(
   new THREE.Face3(0, 3, 2),
   . . .
   new THREE.Face3(4, 5, 1),
);
const material = new
THREE.MeshBasicMaterial
({color: 0x0120FF});
mesh = new
 THREE.Mesh(geometry, material);
scene.add(mesh);
```

It must be noted that for this example the vertices and faces were manually inputted since the shape is not complex. The manual defining of the shape is inefficient and encourages errors but provides a greater level of control than using the THREE.BoxBufferGeometry function.

Requirement 2

Drawing 3 lines to label the x, y and z axis. Change the argument in the AxesHelper to change the length of the axis lines. Alternatively, 3 lines that pass through the corresponding axis could be drawn. The lines can be drawn by defining points that are on strictly only on the corresponding axis to form a line geometry which can be used to define a THREE.Line.

```
var axisHelper = new
THREE.AxisHelper(5);
scene.add(axisHelper);
```

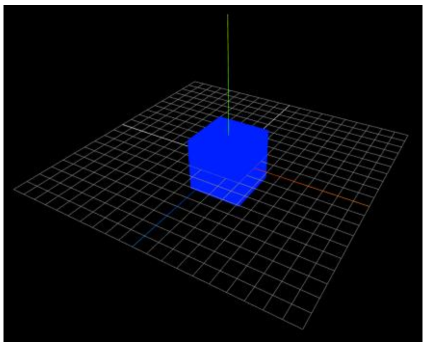


Figure 1 Requirement 1 and 2: Loading a cube and axes

Requirement 3

To rotate in a certain axis, we need to change the angle that the object/mesh is rotated in the respective axis. You can change the value that the angle changes by, this increases the speed of the rotation as angle increases quicker.

```
// x axis
mesh.rotation.x += 0.01;
// y axis
mesh.rotation.y += 0.01;
// z axis
mesh.rotation.z += 0.01;
```

Requirement 4

Naturally, the cube has only 8 vertices so applying a points material to the geometry applies that material to the vertices. This leads to the rendering of the vertices. We can change the size of the points by changing the size value – for greater or lesser exposure (refer to Figure 5).

```
//vertices
const material = new
THREE.PointsMaterial
({color: 0x0000ff, size: 0.1});
mesh = new THREE.Points
(geometry, material);
```

By selecting the option to expose the wireframe to true, the material exposes/renders only the edges instead of the faces (refer to Figure 6).

```
// edges/wireframe
const material = new
THREE.MeshBasicMaterial
({color: 0x0120FF, wireframe:
true});
```

Faces are rendered when we apply the material from requirement 1 (refer to Figure 1).

```
// Faces
const material = new
THREE.MeshBasicMaterial
({color: 0x0120FF});
```

Requirement 5

To move the camera, we need to translate it in the direction of a specific axis. By changing the parameter that we want to translate the camera by, we change how quickly move in the desired direction.

```
// move up
camera.translateY(0.05);
// move down
camera.translateY(-0.05);
// move right
camera.translateX(0.05);
// move left
camera.translateX(-0.05);
// move forward
camera.translateZ(-0.05);
// move backward
camera.translateZ(0.05);
```

Requirement 6

Like Orbit controls I wanted to use the mouse as the controls. The way the controls work is that the left click button of the mouse has to pressed down and dragging the mouse causes the camera to orbit. First, we need to tell three js to listen to mouse actions.

```
domObject.addEventListener("moused
own", mouseDownHandler);
domObject.addEventListener("mousem
ove", mouseMoveHandler);
domObject.addEventListener("mouseu
p", mouseUpHandler);
```

To know how much to orbit the camera, we need to find the distance the mouse is being dragged. We do this by defining the mouse location when mouse is pressed down and orbiting each time the mouse moves to a new location.

```
function mouseDownHandler(e) {
    previousLocationX = e.clientX;
    previousLocationY = e.clientY;
    isMouseDown = true;
}

function mouseMoveHandler(e) {
    if (!isMouseDown)
        return;

    // Change in X and Y
    let deltaX = e.clientX -
previousLocationX;
    let deltaY = e.clientY -
previousLocationY;

    if (orbit!=null)
        orbit(deltaX, deltaY);

    previousLocationX = e.clientX;
    previousLocationY = e.clientY;
}

function mouseUpHandler(e) {
    isMouseDown = false;
}
```

To achieve an orbital rotation, it is best to first convert the camera coordinates to spherical coordinates (Wikipedia, Wikipedia). To find the radius from the look at point (a variable that can be changed), we consider the radius calculation relative to the look at point. After testing the orbit using Wikipedia's approach, I realised the x and z felt inverted. I researched a solution for this behaviour and found out from a Stack Overflow user that three js expresses the x, y and z axis differently to how they are expressed in mathematics (Brakebein, 16). I, therefore, made the appropriate changes.

```
var radius, phi, theta;
var pos = camera.position;
pos.sub(lookAtPoint);
// conversion to spherical coordin
radius = pos.length();
phi = Math.atan(pos.x / pos.z);
theta = Math.acos(pos.y / radius);
```

After the conversion, changing phi and theta while keeping the radius the same would emulate an orbit. I used the Gamedev forum to find out which angle to change according to which change in mouse position axis (Shervanator, 2012). I.e., change in mouse position along the x axis should change phi, and change in mouse position along the y axis should change theta.

```
phi -= deltaX;
theta -= deltaY;
```

After the phi and theta changes, convert the spherical coordinates to the cartesian ones and apply them to the camera position. To implement the changeable look at point, the camera is told to look at the variable look at point vector.

```
if(pos.z < 0)
{
   // Convert the spherical to
cartesian coordinates
   camera.position.x = -radius *
Math.sin(theta) * Math.sin(phi);
   camera.position.y = radius *
Math.cos(theta);
   camera.position.z = -radius *
Math.sin(theta) * Math.cos(phi);
}
else
{
  //do the same as in if statement
(just change sign of x and z)
}

camera.position.add(lookAtPoint);
camera.lookAt(lookAtPoint);
```

Requirement 7

To apply a texture, we must first load it using a texture loader, then map the texture to a material which is then applied on the cube geometry to make a mesh. The mesh function can take materials in a variety of ways, namely a single material or an array of materials. A single material applies the same texture to all the faces whereas an array applies the material to each face individually in an orderly manner. I.e., first index corresponds to the front right face, second index corresponds to the back left face etc.

```
const boxGeometry = new
THREE.BoxBufferGeometry(2, 2, 2);
const loader = new
THREE.TextureLoader();
```

To load a texture we replace the '//url' parameter in the code below with a website URL or a local directory, that corresponds to a picture. To avoid skew, use a picture with a square resolution where the width and height are equal (e.g. 256x256, 220x220 and 400x400).

```
const material = new
THREE.MeshBasicMaterial({map:
loader.load(//url)});

const materials = [
   new
THREE.MeshBasicMaterial({map:
loader.load(//url)}), //front
right
. . .
 THREE.MeshBasicMaterial({map:
loader.load(//url)}), // back
right
];
mesh = new THREE.Mesh(boxGeometry,
materials);
```

To apply a single texture to all sides, change the materials parameter in the mesh to a single MeshBasicMaterial with a texture mapped.

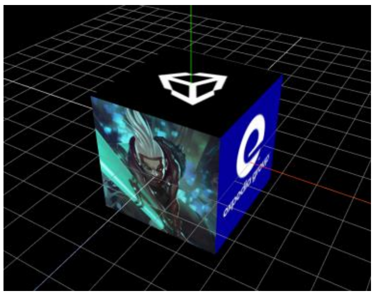


Figure 2: Front sides of cube with textures applied

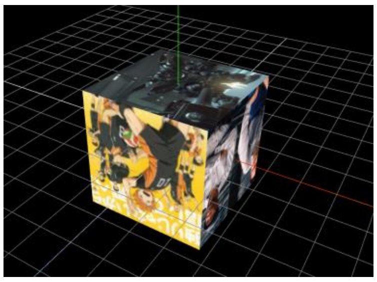


Figure 3: Back sides of the cube with textures applied

Requirement 8

To load an .obj file like the Stanford bunny, we need to use a library called OBJLoader. OBJLoader reads the file and turns it into an

3d object that can be comprehensible by three js. The loading of an object is asynchronous meaning it is not instant, therefore different functions are used to track each part of the loading progress. The function with the xhr parameter is called while the object is being loaded. The function with the object parameter is a function that operates after the object has been loaded. Once the object has been loaded, we can assign it to an object so we can use it outside of the load function. To show the object, we can just add the object to the scene, or we can add a material to change its colour, properties, apply textures etc. We can also scale and transform the mesh to change the position and size of the object.

```
const loader = new
THREE.OBJLoader();
// load a resource
loader.load(
      // resource URL
     'http://localhost:8000/bunny-
5000.obj',
      // called when resource is
loaded
      function ( object ) {
         const material = new
THREE.MeshBasicMaterial({color:
0x00ff00});
         bunnyMesh = new
THREE.Mesh(bunnyObject.children[0]
.geometry, material);
         bunnyMesh.scale.set(0.4,
0.4, 0.4);
         bunnyMesh.translateX
(-0.4);
         scene.add(bunnyMesh);
      },
      // called when loading is in
progresses
      function ( xhr ) {
         console.log( ( xhr.loaded
/ xhr.total * 100)+ '% loaded');
      },
       // called when errors
      function ( error ) {
   console.log( 'An error
happened' );
      }
);
```

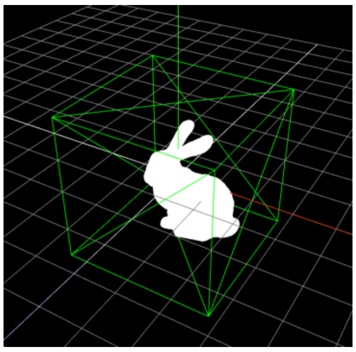


Figure 4: Rendered Bunny inside cube

Requirement 9

Explanations

Like requirement 4, after the bunny object was loaded it was stored as a global mesh variable so that it can be changed.

To rotate the mesh, I incremented the rotation angle on either the x, y, or z axis.

For the vertex rendering mode, I used a points material to render the bunny geometry's vertices.

```
// vertices
const bunnyMaterial = new
THREE.PointsMaterial({color:
0x0120FF, size: 0.01});
bunnyMesh = new
THREE.Points(bunnyObject.children[
0].geometry, bunnyMaterial);
```

For the edges rendering mode, I used a material with the wireframe value set to true to render the bunny geometry edges.

```
// edges/wireframe
const material = new
THREE.MeshBasicMaterial({color:
0x00ff00, wireframe: true});
```

For the faces I had the option to render the bunny as a material that is affected by lighting or not affected by lighting.

```
// Faces
const material = new
THREE.MeshBasicMaterial({color:
0x0120FF});
// Faces (lighting)
const material = new
THREE.MeshPhongMaterial({color:
0x0120FF});
bunnyMesh = new
THREE.Mesh(bunnyObject.children[0]
.geometry, material);
```

For each of the above renders after the mesh was generated the mesh had to be scaled and translated accordingly.

```
// Adjust scale
bunnyMesh.scale.set(0.3, 0.3,
0.3);
bunnyMesh.translateX(-0.4);
```

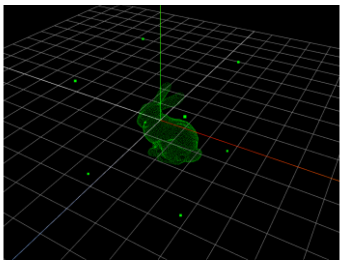


Figure 5: Vertex render for both cube and bunny

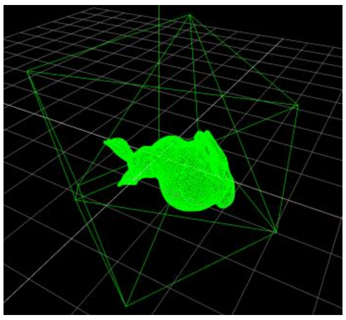


Figure 6: Edge render for both cube and bunny

Requirement 10

For this requirement I decided to create a scene similar to the scenes in the anime One Piece (Wikipedia, 2020). To achieve this, I first rendered an island to match the sea and pirate theme. I first created a plane geometry and applied a water texture to make it look like the ground is the sea.

```
// load ground
let groundPlaneGeometry = new
THREE.PlaneGeometry(300, 300 , 1,
1);
var map =
textureLoader.load('url');
let material = new
THREE.MeshStandardMaterial({map:
map});
let ground = new
THREE.Mesh(groundPlaneGeometry,
material);
```

Then I proceeded to add the character models I could find and position them in a natural position. I made a function that was able to load a model based on the model URL. Other parameters of that function include the translate, scale and rotation vector to configure the position and look of the loaded model.

```
// a typical loader function
function loadOBJModel(url,
textureUrl, scaleVector,
translateVector, rotateVector)
{
   const loader = new
THREE.OBJLoader();
   loader.load(url,
         function ( object )
         {
               objectMesh = new
THREE.Mesh(object.children[0].geom
etry, material);

objectMesh.scale.set(scaleVector.x
, scaleVector.y, scaleVector.z);

objectMesh.translateX(translateVec
tor.x); // do the same for y and z

objectMesh.rotateX(rotateVector.x)
;     // do the same for y and z
```

After completing the other 9 requirements, I realised that the key maps for every functionality was confusing. To solve this, I implemented a GUI that would trigger functionality when the user clicks on a menu option (refer to Figure 7).

```
const gui = new GUI({width: 500
});
var function = {toggle:
function(){//code here}};
gui.add(function,
"toggle").name("//name");
```

To display the GUI, I have used an external library that abstracts a GUI being added. In the code above the function variable displays an example of how the GUI works. You create a function with a parameter. That function is assigned to the GUI menu option such that when that GUI option is selected the function is triggered. The GUI menu option has the option of adding a name- this ensures the name displayed highlights to the users what selecting the option would do. The GUI library has the option to add folders which helps divide the GUI functionality into segments.

```
const cubeFolder =
gui.addFolder("Cube");
var onlyFaces = {add:
function(){renderCube()}};
cubeFolder.add(onlyFaces,
"add").name("Only Faces");
```

Despite adding a GUI and numerous models, the scene did not feel animated or interactive enough. To solve this, I decided to implement model spawning and object movement. The model most suitable for spawning was the dolphin as it felt the most fitting. I added an option to spawn a dolphin in a random location outside of the island region.

```
function spawnRandomDolphin(){
   let randomYAngle =
Math.random() * (2 * Math.PI);

   let randomXPosition =
getRandomInteger(-140, 140);
   while(randomXPosition < 20 &&
randomXPosition > -20){
      randomXPosition =
getRandomInteger(-140, 140);
   }
   // do the above to get random Z
position value too
   loadOBJModel(...);
}
```

I did this by using the Math random function to generate random x and z coordinates as well as a random rotation angle with the y axis.

I added two ships. One which the user can move using the keyboard.

```
// controllable ship
function moveShip(x, yAngle){
   if(x!=null){
      shipMesh.translateX(x);
   }
   if(yAngle!=null){
      shipMesh.rotateY(yAngle);
   }
}
```

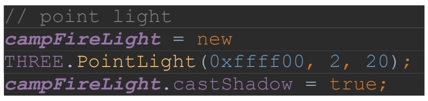
The other ship that moves in a fixed direction. To add more functionality, I decided to allow the user to change the max range that the ship can move to and the speed of the ship.

Commented [AS1]: explain

```
// automatically moving ship
function moveOtherShip(z, bound){
   // If out of bounds then rotate
   if( otherShipMesh.position.z >
bound ||
otherShipMesh.position.z < -bound)
   {
    otherShipMesh.rotateY();
    otherShipMesh.translateZ(-30);
   }
   // If still in bounds, move
   else{
    otherShipMesh.translateZ(-z);}
}
```

The above function is called in the animate function as the ship is asked to move every time the animate function loops. The variables are passed in place of the bound and z parameters, therefore when the values are changes using the GUI the function behaves accordingly. As the animate function loops different values instead of the parameters can be used at different calls of the function.

Lastly, to showcase lighting and shadows I implemented a point light. I thought the most natural way for the point light to be in the scene is for it to be the light for the 'campfire'. All models showcase lighting, but trees and the moving ship are especially interesting. The trees show how light behaves on uneven surfaces. The moving ship displays the dynamic shadows and lighting(refer to Figure 8).



```
// point light
campFireLight = new
THREE.PointLight(0xffff00, 2, 20);
campFireLight.castShadow = true;
```

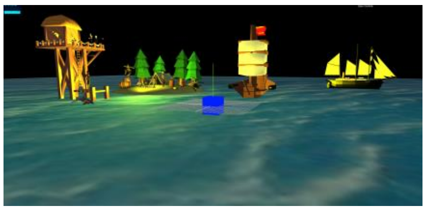


Figure 7: One Piece scene loaded with GUI closed

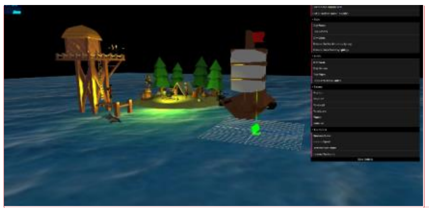


Figure 8: Scene loaded with GUI opened

Discussion

Lessons Learned

The primary website I was using was sketchfab (Sketchfab, 2020) and where different models had different output formats (obj, fbx, mb, etc). One detail I had missed is that every downloadable object has the option to export to a gltf format. That meant If I had just configured the GLTFLoader I did not need to configure other format loaders like fbx, tga and mlt, and hence spent less time. I found some models where the materials just would not load. I tried fixing them by remapping the material files, for example changing the texture directories in the mtl files. Some models worked some didn't. After searching for I learned that most good models are not free and different artists produce models to different qualities.

Another major lesson I learned was the importance of abstraction. The lack of abstraction became especially noticeable when loading numerous models using similar scripts. I spent extra time planning what aspects of the code were similar and created functions that abstract that functionality. I believe abstraction would become even more important if I decided to add numerous objects that are moving or can be controlled. The easy way to do this would be to create many variables but this would be very resource hungry and negatively impact the websites performance. Abstracting the functionality to scripts would be much more efficient.

Limitations

One limitation was the access to 3D models. Many artists that publish their models charge money for them. As I did not want to spend money I was limited to a smaller selection of models.

Another major limitation was my lack of familiarity with both JavaScript and three js. JavaScript's core functionality syntax is like familiar languages such as Java. Examples include loops and object instantiation. However, many aspects of JavaScript were unknown to me. An example is the fact that JavaScript is dynamically typed. Learning a new type of programming language took time and slowed down how quickly I could complete requirements. In addition to not knowing JavaScript, learning a library for it was difficult. Thankfully, the documentation was helpful but my lack of experience with the library meant I took a long time to achieve my goals. Additionally, the library felt less supported compared to other technologies such as the Unity game engine. This decreased the likeliness of finding a solution to a question I had.

Future Work

Three JS was quite enjoyable, and I hope to expand the knowledge I gained from this deliverable by creating more projects.

I would begin by learning the basics of JavaScript and improve the code for this deliverable to comply with best practices. To further improve the code quality, I would learn how the integration of JavaScript in

Commented [AS2]: update

Commented [AS3]: If more time:
* learn to animate and add animations
* improve code quality (not all functions in one html)
* implement collisions

Lessons learned:
- Sketchfab lol
- To abstract as much as possible (loader functions)

Limitations:
- Not wanting to pay for models
- Not knowing javascript
- Not enough help with learning three.js or javascript

HTML works to a higher degree. This is so I can abstract more code from the deliverable in separate JS files instead of all the code being in one file.

Regarding the scene I loaded, it was just basic object loading. To enhance the user experience, I would add animations. This would require me to learn how to use software such as Blender and learn animation. The reason I did not do this for this deliverable is because I felt that it would lead to less functionality but a prettier scene which is not what I wanted. To add to the scene further I would enhance the physics of the scene. I would add rigid body, gravity, and collision handling scripts. Essentially, I would aim to reproduce what would feel like a level in a video game.

References

Brakebein, S. O. u., 16. *Stack Overflow.* [Online]
Available at: https://stackoverflow.com/questions/35128148/how-to-get-camera-position-with-mouse-move-event-and-transform-into-screen-coord
[Accessed 4 December 2020].

Shervanator, G. u., 2012. *Gamedev.* [Online]
Available at: https://www.gamedev.net/forums/topic/632341-spherical-coordinates-for-orbital-motion/
[Accessed 4 December 2020].

Sketchfab, 2020. *Sketchfab.* [Online]
Available at: https://sketchfab.com/feed
[Accessed 16 December 2020].

Wikipedia, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/One_Piece_(TV_series)
[Accessed 12 December 2020].

Wikipedia, Wikipedia. *https://en.wikipedia.org/wiki/Spherical_coordinate_system#Cartesian_coordinates.* [Online]
Available at: https://en.wikipedia.org/wiki/Spherical_coordinate_system#Cartesian_coordinates
[Accessed 4 December 2020].