Visual computation report

Introduction

This report describes my thought process to meet the requirements of the THREE.js coursework. Namely the task itself, the steps taken to complete the task and validation that the task was met. The report then closes with limitations and future aims.

Task 1: Draw a cube

I first consulted the three.js create a scene tab to familiarise myself with Three.js concepts. By the end of the tutorial, I was confident with understanding that a three.js mesh is comprised of a material and a geometry. In order for it to be visible, it has to be added to the scene by the *scene.add()* function.

I first started to code the cube by writing code in the *init()* function. Within that function, I first created a geometry for the cube using THREE.BoxGeometry; a class for initialising a rectangular cuboid by specifying width, height and depth (three.js documentation). I then created a material using Mesh.BasicMaterial. To finish the cube creation, all that was required was to combine the geometry and the material together with THREE.Mesh. I then added the mesh to the scene with the *scene.add()* function. No repositioning was required because, lecture 11 of visual computing highlighted that all objects are rendered at (0,0,0) of the world coordinates.

Further enhancements involved changing MeshBasicMaterial to MeshPhongMaterial and also adding texture to the cube. According to the three.js documentation, MeshPhongMaterial has the advantage of computing light per pixel. This results in a better appearance for the mesh. To add the texture, I loaded it to the cube by THREE.TextureLoader. I then applied the loaded texture to the material of the cube by using the map attribute of the material.

```
const texture=new THREE.TextureLoader().load
('textures/crate.png');
var geometry = new THREE.BoxGeometry( 2, 2, 2);
var material = new THREE.MeshPhongMaterial
({map:texture});
mesh = new THREE.Mesh(geometry,material);
scene.add(mesh);
```

Figure 1: Code snippet for the creation of the cube.

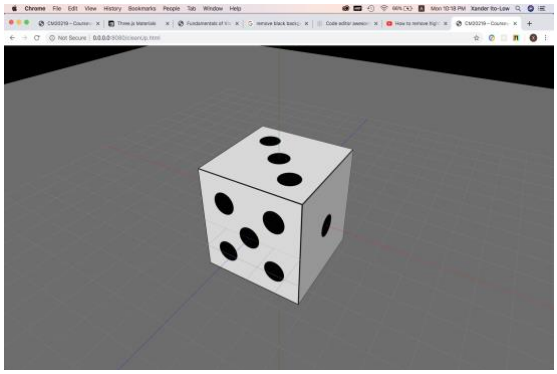

Figure 2: Visible cube with length 2 and axes

Task 2: Draw a coordinates system

Initially, my solution consisted of the creation of an Axis helper. It was capable of creating the lines required; which were orthogonal to one another. The axis helper in three.js is capable of creating the object with red representing the x-axis, green representing the y-axis and blue representing the z-axis (three.js documentation). The initial axes can be seen in figure 5.

When it came to the demo however, I realised that the task was not fully met as I was required to represent the axes for the entire grid. Axes helper didn't offer functionality to solve the task. Therefore, I was required to create the lines for the x, y and z-axes respectively.

Creation of the lines involved creating a geometry from scratch and specifying lineBasicMaterial for the line. The creation of the x-axis involved creating an empty geometry, then pushing the vertices of the edges of the grid; (-5,0,0) and (5,0,0). The final step was applying the material with the required colour attribute. The same process was applied to the y and the z-axes, but with different vertices. Figure 2 highlights this result.

```
var materiallineX= new THREE.LineBasicMaterial
({color: 0xff0000});
var linex = new THREE.Geometry();
linex.vertices.push(new THREE.Vector3(-5,0,0));
linex.vertices.push(new THREE.Vector3(5,0,0));
var line1 = new THREE.Line(linex,materiallineX );
scene.add(line1);
```

Figure 3: Code snippet for the creation of the line for the x-axis

Task 3: Rotate the cube about the x, y and z-axes

The three.js documentation from task 1 provided me with adequate knowledge about how to implement rotations about the x, y and z-axes through the getting started example. There is a property called rotation that represents the object's local rotation through the use of Euler angles.

From lecture 9 of visual computing, I learnt about Euler angles and how they are applied to the object's original point to form a rotated point by the following equation:

$$P' = R_z(\psi)R_y(\phi)R_x(\theta)P$$

Equation 1: Euler angle rotation in lecture slides 9 of visual computing.

Gimbal lock is associated with Euler coordinates. It is the problem of losing one degree of freedom when two axes align. In this task, however, it was of no concern because the task required the rotation of the cube about each axis respectively as opposed to simultaneously.

Rotations in three.js were achieved by creating keyboard events for the 'X',' Y' and 'Z' keys. These keys set the respective axis rotation speed to 0.1 and set the other axes rotation speeds to 0. An animated rotation effect was then achieved through the use of the *animation()* function. According to the three.js documentation, the *animation()* function is responsible for re-rendering the scene 60 times a second. This ensured that in the animation function I could increment the respective rotations by 0.1, resulting in rotations about the x, y and z-axes respectively.

```
//Global variables for rotation values
var xSpeed = 0,ySpeed = 0,zSpeed = 0;
…
//Change rotation speed
changeSpeed(x,y,z);
…
//In animate function
mesh.rotation.x+=xSpeed;
mesh.rotation.y+=ySpeed;
mesh.rotation.z+=zSpeed;
…
//rotation about x-axis
changeSpeed(0.01,0,0);
```

Figure 4: Code snippet for rotation about the x-axis

An enhancement made to this task was to reset the position of the cube by pressing the 'S' key. This was achieved by resetting all of the rotation values to 0 and positioning the box back at (0,0,0) by the use of the *position.set()* method.

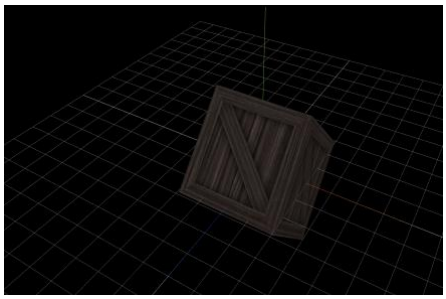

Figure 5: Rotation about the x-axis. Rotation about the y and z-axes are similar

Task 4: Different rendering modes

The first approach to this task consisted of changing the mesh explicitly into its form through the use of the global variable mesh. My function removed the mesh and added the mesh in its desired form through *scene.add().* This meant that there would be less computational power used in this solution because the renderer would only be concerned with one mesh.

To render edges, I first used THREE.EdgesGeometry to convert the original geometry into a wireframe and then created a material; using the same material from task 2. I then combined everything into a mesh. A similar approach was used with rendering vertices and faces. The material used for vertices was THREE.Points. For faces, the same procedure in task 1 was utilised except the material was green.

Looking ahead at task 9, I realised that I had to adapt the method such that it could accept an arbitrary mesh. This was achieved by capturing the mesh geometry of the original mesh through the mesh .geometry property.

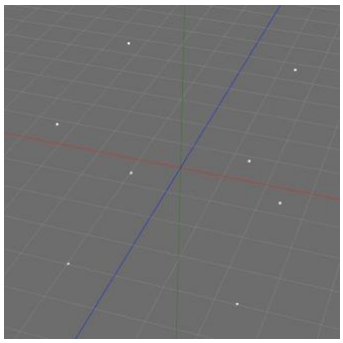

Figure 6: Vertices mode

I then passed the mesh and the variable, storing its original geometry into the *initiate()* function. This function created a new mesh based on the input received by the user. Code was further adapted to to render the different modes with rotations intact. This was achieved by getting and setting the rotation components.

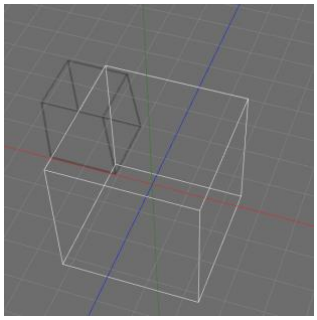

Figure 7: Edge mode

```
function initiate(event,mStore,obj){
//object will be set to the mesh mode
…
var m;
 scene.remove(obj);
 //capture rotations
 var rotationX = obj.rotation.x;
…}
var geo = mStore.clone();
//edge rendering
if(event.keyCode==69)
 var edges = new THREE.EdgesGeometry(geo);
 m = …}
…
captureRotation(rotationX,rotationY,rotationZ,m);
scene.add(m);
```

Figure 8: Code snippet for the initiate() function

Task 5: Translate the camera about its axis.

This task was initially completed by creating event handlers to translate the camera about its axes, using *.translateX()*, *.translateY()*, *.translateZ()* provided by three.js. According to the three.js documentation, the above functions are capable of translating an object with respect to its local axis as opposed to the world axis. I suppose the functionality was accomplished with a compound transformation matrix translating the camera's coordinates with respect to its frame of reference. Homogenous coordinates are also employed when computing the camera's new coordinates.

To improve on this implementation, I wanted to enable translations by using the mouse. From MDN web docs, I was able to discover that the mouse has a mouse.movementX property and a mouse.movementY property. These properties are responsible for getting the X-movement of the mouse and the Y-movements of the mouse relative to the mouse's last position. With this in mind, I recorded these values and then passed them into the cameras *translate X,Y,Z* functions. To get these translations working, I was required to create an event listener for mouse movements, such that the *mouse Movement()* function would be called automatically when the mouse was moved.

```
window.addEventListener('mousemove',
mouseMovement,false);
…
function mouseMovement(event){
var movement1 = event.movementX;
var movement2 = event.movementY;
camera.translateX(movement1*0.01);
camera.translateY(movement2*0.01);}
```

Figure 9: Code for x and y translations with mouse input

To translate the camera across its z-axis using the mouse, I was required to use the mouseWheelEvent offered by MDN web docs. I recorded the DeltaY; which is the wheel movement in the y-direction. I then applied this result to translate the camera in the z-direction. To make a smoother translation in the x, y and z directions, I multiplied all mouse movements by 0.01. The task was complete. Figure 14 shows this result.

```
window.addEventListener('wheel',mouseWheel,
false);
//Translation accross the Z-axis
function mouseWheel( event ) {
camera.translateZ(event.deltaY * 0.01);}
```

Figure 10: Code for z translation with mouse input

Task 6: Arc-ball mode for the camera

To complete this task, knowledge of an alternative coordinate system was required. This coordinate system is known as spherical coordinates. The reason spherical coordinates were required is because they capture circular movements in both latitudinal and longitudinal directions; making it simpler to achieve the proposed arc ball mode.

The first method of achieving arc ball mode was to implement it using the arrow keys of the keyboard. My method consisted of first finding the distance from the camera to the center point. This was achieved by utilizing equation 2.

With the radius captured, the next step was to convert the x, y and z coordinates of the camera into spherical coordinates and increment them by 0.1 radians. The following formulas were utilised in converting cartesian

coordinates into spherical coordinates. When φ and θ were found, I incremented them by 0.1 radians and then converted the result back to cartesian coordinates utilising the equations below:

$$r = \sqrt{(x_2+y_2+z_2)}$$

Equation 2: Calculate the distance from the origin

$$\varphi = \arctan\left(\frac{y}{x}\right)$$

Equation 3: Calculate phi

$\theta = \arccos\left(\frac{z}{r}\right)$, where r is from equation 3.

Equation 4: Calculate theta

$$x = r\sin\theta\cos\varphi$$

Equation 5: Calculate x

$$y = r\sin\theta\sin\varphi$$

Equation 6: Calculate y

$$z = r\cos\theta$$

Equation 7: Calculate z

The above equations are from (Wikipedia, 2019)

For my initial implementation to be complete, I had to set the camera's lookAt property to the center. This ensures that the camera is always focused on the center when it is rotated.

In theory, this should have worked. However, when clicking the keys to enable arc ball mode, unpredictable behaviour occurred. After I had completed my demo, it was found that the unpredictable behaviour was caused because the axes in figure 11 are interpreted differently. In three.js figure 11 is rotated 90 degrees anticlockwise. This results in new equations for the arc ball mode, namely changing sin θ to cos θ and arcos to arcsin (Wikipedia,2019).

My implementation in the demo required the consultation of learnopengl.com, where I used the equations provided on their page to interpret the spherical coordinate equations.

Regarding further enhancements, I wanted to be able to click and move the mouse to rotate it about the axis. Mr Doob's code segment helped me achieve this. In my final version of the code, when the mouse is clicked the mouse event handler is called and it records the x and y position of the mouse. When the mouse is moved, the camera coordinates are updated by Mr. Doob's function. I had to set a flag as to when the mouse was clicked because it would mean that I could encapsulate the functionality of arc ball mode and the camera. To keep the mouse movements controlled, I set the minimum value of $\varphi(phi)$ to -89 degrees and the maximum to φ degrees ensuring that the view didn't reverse (learnopengl.com)

I noticed when loading the bunny that rotations were unstable at 89 degrees, hence I set the minimum and maximum phi to 80 degrees. The task was complete.

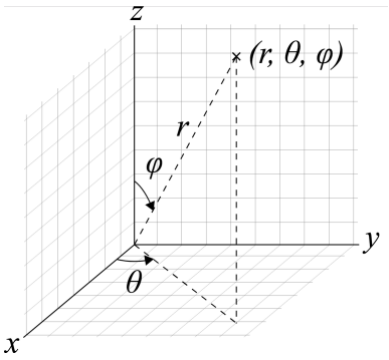

Figure 11: Spherical coordinate representation from Wikipedia

```
//cap phi so no unexpected behaviour
phi = Math.min(80, Math.max(-80, phi ));
camera.position.x = radius * Math.sin( theta
*Math.PI / 180) *Math.cos( phi * Math.PI /
180 );
camera.position.y = radius *
Math.sin( phi*Math.PI / 180 );
camera.position.z = radius * Math.cos( theta
*Math.PI / 180 ) * Math.cos( phi * Math.PI /
180 );
…
```

Figure 12: Mr. Doob's implementation with my modification to update x, y, z coordinates.

```
Phi = (Math.asin(camera.position.y/radius))
+addPhi;
if(phi>=80.0*(Math.PI/180)){
 phi = 80.0*(Math.PI/180);}
if(phi<-80.0*(Math.PI/180)){
 phi = -80.0*(Math.PI/180);}
 theta = (Math.atan(camera.position.z
 /camera.position.x))+addTheta;
 camera.position.x = radius * Math.cos(phi)*
 Math.cos(theta);
 camera.position.y = radius * Math.sin(phi);
 camera.position.z = radius * Math.cos(phi) *
Math.sin(theta);
```

Figure 13: My original implementation to update x, y, z coordinates.

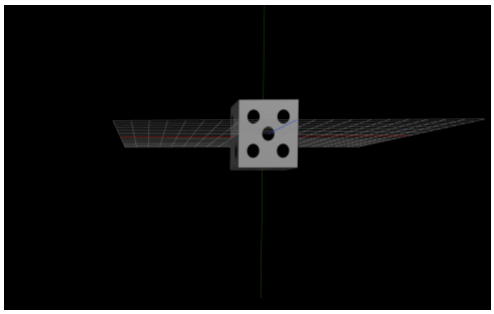

Figure 14: Arc ball mode and translation of the camera

Task 7: Texture map different faces for the cube

This task had been partially achieved from the results from task 1. To complete the task, different faces needed to be mapped to the cube.

```
const texture0 = new
THREE.TextureLoader().load('textures/face1.jpg')
;
…
var cubeMaterial = [
new THREE.MeshPhongMaterial({map:texture0}),
…]
```

Figure 15: Code for loading different faces to the cube

The Sonar system tutorial about textures and colors showed that I needed to create an array of materials with each material loading a different image corresponding to different faces of the cube. The material array was then passed to the cube mesh constructor as done in task 1. The images collected from clipart.com were in powers of 2 by default, hence there were not any issues with the map function as it is a problem with WebGL (three.js documentation). Figure 2 displays my results of mapping faces of a die to my cube. The initial texture used in the code for task 1 was replaced with this texture for the creation of the cube.

Task 8: Loading the bunny into the cube

This task required knowledge of how to access an object through the .OBJ class and how to fit it into a cube without the use of explicit numbers.

The first thing I did in this task was to create the object loader to load the .obj file provided; the three.js documentation gave me insight into how an object loader is constructed. From that information, I instantiated an object loader, loaded my object and passed a callback function to add the object to the scene.

I then transformed the bunny by an explicit amount using *scale.set() and translateX()* within the callback function. I later found out from the lab assistants that this wasn't the best way to ensure that the bunny fit exactly into the cube.

Before dealing with this, an enhancement was made in order to get the geometry from the object loaded into the .obj file. This would make it easier to implement the subsequent tasks. From the content in lecture 11 of visual computing, I concluded that the bunny object must be composed of multiple triangles. In order to access every triangle mesh's geometry, I created an empty geometry as I did in task 2. I then used the *.merge()* function in order to merge the geometries of individual triangles to construct the bunny. When reading in a triangle geometry, I had to convert this geometry because it was a BufferGeometry. I accomplished this by using the constructor from the geometry class along with the *.fromBufferGeometry()* function. (three.js documentation).

With the geometry collected, I then worked on improving the transformation. The improved method of transforming the bunny into the cube was to first utilize the three.js Box3 class to create a bounding box around the bunny object. The next step involved finding the difference between the maximum vector and the minimum vector of the bounding box in order to get the distance vector of the box.

From this result, I had to select the maximum component of that vector and scale it so that the resulting object fit in the cube's box. From observation, I realised that the scale should be set to 2 divided by the maximum vector component, as the cube's maximum component is always 2. The child's geometry was then scaled by this factor. Using this output, the original bounding box had to be re-computed in order to capture the scale of the bunny.

In order to translate the bunny into the cube, the cube's center vector had to be subtracted from the bunny's center and then the child geometry was translated by this difference. The difference of vectors was computed using the *.sub()* function and the transformation was achieved by matrix multiplication using the *.applyMatrix()* function from the three.js documentation.

Outside of the function call, I then created the mesh by passing in the geometry of the transformed bunny and a MeshPhong material with a color attribute. The mesh was added to the scene with the *scene.add()* function. The bunny was successfully loaded inside the cube when I pressed the 'L' key.

```
object.traverse(function(child){
if(child instanceof THREE.Mesh){
//So the bunny fits exactly inside the cube
 var box = new THREE.Box3().setFromObject
(object);
 var maximum = box.max;
 var minimum = box.min;
 var difference = maximum.sub(minimum);
 var maxVal = Math.max(difference.x,
 difference.y,difference.z);
 var scale = 2/maxVal;
child.scale.set(scale,scale,scale)
box.setFromObject(child);
var vector = center.sub(box.getCenter());
//apply transformations on the geometry
var a = new THREE.Geometry().
fromBufferGeometry(child.geometry);
a = a.scale(scale,scale,scale);
a = a.applyMatrix( new
THREE.Matrix4().makeTranslation(vector.x,vecto
r.y,vector.z));
meshStore2.merge(a);
}});
```

Figure 16: Code snippet to load the object and add transformed children geometry to new geometry store

Task 9: Rotations for the bunny, modes and lights

Due to the work done in task 8, namely storing a geometry for the bunny, the task was nearly met. All that was required to do was add a line to the keyboard events to set the mesh to the required mode when pressed. One modification I had to include was to check if the mesh for the bunny was null, this is because it uses the same event key as the cube mesh to change modes.

```
mesh = initiate(event,meshStore,mesh);
if(mesh2!=null){
  mesh2 = initiate(event,meshStore2,mesh2);
}…
```

Figure 17: New keyboard event format

Rotations were achieved in the same way they were achieved in task 3 with the same event keys.Bunny rotations were included in the animate function with a check if the bunny mesh is null in case it had not been loaded. Different rendering modes utilised the same functions in task 4 and worked exactly the same way.

When changing to face mode, the cube was in the way because the cube used the same event key to change its mode. In order to overcome this I created two keyboard press functions 'R' and 'A' which set the cube mesh's visible property to false and true respectively. Due to this, I had to set the visible property of the new mesh rendered based on the mesh parameter which was passed into the *initiate()* function in figure 8.

Phong lighting consists of a diffuse term, an ambient term and a specular term (Lecture 13 visual computing, 2019). To light the scene, I kept the ambient lights and added a directional light to the scene at an angle to my object. This achieved lighting such that my object was not flat. I then proceeded to add a ground plane to my scene. A ground plane, just as before, consists of a geometry and a material. The ground plane is rendered such that it does not face the ground, hence I had to rotate it by 90 degrees clockwise and position it below the grid helper.

According to three.js documentation, in order to achieve shadows, I had to enable shadowMap for the renderer. I also set the cast property to true for the lights, set the receive and cast property true for the object and finally set the receive property true for the ground plane. This task had been met as the 'F','V' and 'E' keys rendered the bunny in its respective mode. When 'X','Y' and 'Z' were pressed the bunny would rotate around its respective axis.

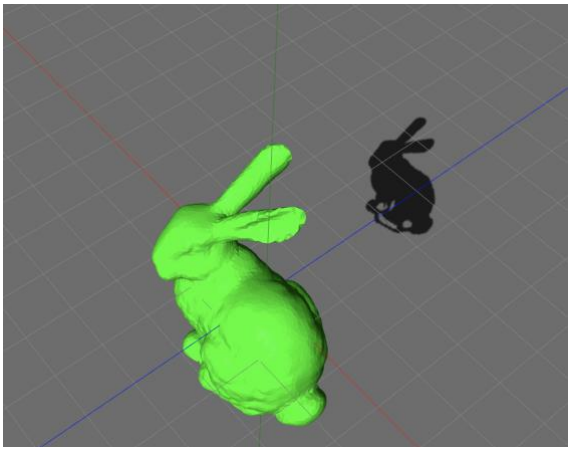

Figure 18: Bunny rendered in face mode with shadow

Task 10: Creative task

I decided to create an animation for a character in three.js by utilising blender. Unboring.net, 2016 gave me an insight that with blender, animating a character was possible as well as exporting blender to three.js.

I then searched for a character asset. The simple character I found was retrieved from free3d.com. I loaded the asset into blender and did not know how to proceed, hence I googled animating a 3d character in blender. I came across the tutorial Rig and animate ANYTHING in Blender and watched it.

From the video, I realised that in order to animate a character, rigging simplifies the task. Rigging means that the mesh contains a skeleton; this skeleton contains pivots making it easier to position the mesh and record animations. Animations in blender are achieved by taking a screenshot of the current position of a mesh, changing the position of the mesh by rotating pivots then taking another screenshot. This creates a keyframe; which is an animation clip.

I used this to create a few draft animations to familiarise myself with how to create animations. I finally settled with the idea of a mesh walking and with the help of Blender Character Animation, I achieved the walking animation for the asset.

In order to import the blender animation to three.js, I used the GLTF loader to load the .glb file of my animation to three.js. According to the three.js documentation, I found that it was possible to extract the animations using the .animations property in the function callback and the mesh itself by accessing gltf.scene.children[0].
This loader was very similar to the obj loader, except it contained different accessors.

The next step was to update the animations. The three.js documentation showed that playing an animation from a .gltf file consisted of five steps:

The first step is to access the animation from the .gltb file. The second step is to create an animation mixer for the mesh as this creates a player for the animation (three.js documentation). The next step is to then create an action for the mesh to enable the control of the animation state. In order to play the animation the *.play()* function is required. The final step consists of creating a clock and updating the animation frame in the *animate()* function by using *.update()* for the change in time passed. The walking animation I created then looped in place.

I then proceeded to add a background for the mesh using the .background property from the scene, the image was retrieved from pexels.com. It was at this point that I wanted to make my animation more realistic.
My animation was not in line with the beach image I loaded. Hence, in the GLTB callback function before the mesh is added to the scene I scaled it and translated it to ensure that my mesh was in line with my image. These transformations were achieved by the *.scale()* function and *.translate X,Y,Z* functions. I also changed my camera position and lookAt positions to ensure that the effect was achieved.

I then proceeded to translate my mesh as the animation was playing. This created the effect that the character was walking in one direction. The character would walk off screen and would continue to walk in a positive direction, hence I set an if statement rotating the character about the y axis to make the character walk in the opposite direction when the character was off of the screen. The same principle was applied when the character was walking in the opposite direction.

The final touches were to add a GUI and a beachball. The beachball was a sphere mesh with a loaded texture from robinwood.com. Using the .map function. The beachball was then animated by translating it up and down by a small amount. The GUI consisted of a button responsible for playing and pausing the animation and a button that was capable of playing music. The GUI was an imported library from three.js. The toggle switches in the GUI were achieved by setting parameters in the *.animate()* function. Music was achieved by an AudioLoader and the sound clip was downloaded from Soundbible.com. Animation in figure 20 was achieved.

```
mesh = gltf.scene.children[0];
 mesh.material = new THREE.MeshLambert
 Material();//translate and scale mesh
 mesh.scale.set(4,4,4);
 mesh.translateY(-10);
 //access my specific animation
 const animation = gltf.animations[2];
 mixer = new THREE.AnimationMixer(mesh);
 const action = mixer.clipAction(animation);
 //play my animation
 action.play();…);
```

Figure 19: Code within the GLTF loader excluding scene.add()

Although there was not a lot of maths involved because of the abstraction provided by blender, all of the animations consist of rotations around reference points; namely pivot points. All of the transformations are represented by compound matrix transformations in a 3D coordinate space. This completed my project.

Figure 20: Result of task 10

Limitations and discussion

A limitation with my implementation is that the bunny does not utilise Phong shading because I did not compute vertex normals, hence its texture does not look very smooth. It also does not utilize specular highlights. In my final task, a limitation was that I did not perform the skeletal animations in three.js mainly because I did not know that the skeleton could be accessed in three.js. Other limitations include: the background does not move with the character and I did not add texture to the asset.

If I had more time, I would aim to develop a further understanding of how to compute vertex normals for the bunny and make it look smoother. Furthermore, I would texture the asset and have the camera follow the asset. The final aim would be to perform skeletal animations in three.js.

This project has highlighted the importance of being resourceful when confronted with a problem. It has also shown that a knowledge of matrix transformations is required, especially within the field of animating. This project has interested me in the animation aspect of visual computing.

References:

Anon, *Camera* [Online]. Available at: https://learnopengl.com/Getting-started/Camera [Accessed December 8, 2019].

Anon. *Free 3D characters Models* [Online]. Available at: https://free3d.com/3d-models/characters [Accessed December 8, 2019].

Anon. *Free stock photo of background, beach, beautiful.* Available at: https://www.pexels.com/photo/background-beach-beautiful-beauty-459556/ [Accessed December 8, 2019].

Anon. *Free Dice Faces, Download Free Clip Art, Free Clip Art on Clipart Library* [Online]. Available at: http://clipart-library.com/dice-faces.html [Accessed December 8, 2019].

Anon. *MouseEvent* [Online]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent [Accessed December 8, 2019].

Anon, 2019. *Spherical coordinate system* [Online]. Available at: https://en.wikipedia.org/wiki/Spherical_coordinate_system [Accessed December 8, 2019].

Anon. *three.js docs* [Online]. Available at: https://threejs.org/docs/ [Accessed December 8, 2019].

Anon. *Texture Maps for Balls! Free Textures for Basketball, Beach Ball, Pool Balls, Softball, and Tennis Balls* [Online]. Available at: https://www.robinwood.com/Catalog/FreeStuff/Textures/TexturePages/BallMaps.html [Accessed December 8, 2019].

Anon, 2017. *Three.js Tutorial 5 – Textures and Colours* [Online]. Available at: https://www.youtube.com/watch?v=l77yAZ0E950 [Accessed December 8, 2019].

Creative R.,2016. *Rig and Animate ANYTHING in Blender* [Online]. Available at: https://www.youtube.com/watch?v=mhQY2_gVoVg. [Accessed December 8, 2019].

Doob. *Voxels* [Online]. Available at: https://mrdoob.com/projects/voxels/ [Accessed December 8, 2019].

Gammon CJ, 2016. *Three.js part 6. Lights and camera* [Online]. Available at https://www.youtube.com/watch?v=vB5lSSJRJR0 [Accessed December 8, 2019].

Koenig, M,2009. *Sea Waves Sounds: Effects: Sound Bites: Sound Clips from SoundBible.com* [Online]. Available at: http://soundbible.com/885-Sea-Waves.html [Accessed December 8, 2019].

Lague S.,2015. *Blender Character Animation: Walk Cycle* [Online]. Available at: https://www.youtube.com/watch?v=DuUWxUitJos [Accessed December 8, 2019].

SLU, U. *Workflow: Animation from Blender to three.js. unboring.net* [Online]. Available at: https://unboring.net/workflows/animation.html [Accessed December 8, 2019].