CM20219 - Fundamentals of Visual Computing

Coursework Part 2: Viewing and analysing 3D Models using WebGL

Elias Abou Farhat 199246766 10/01/2022

1 Introduction

Visual computing 3D applications have been used for many applications like website design, healthcare simulation, and other key elements present in our lives. This project consists of demonstrating of creation website 3D animations using JavaScript. We included a cube and objects in this project and applied many commands to control this 3D scene. The report will mainly summarize three requirements done with their explanation and go through the small game application created using basic mathematical concepts.

2 Libraries Used

I used WebJl and JavaScript for this project and, more precisely, an API called three.js used for graphics and 3D animations. For easy control and good user experience, I used a library dat.gui [1], a graphical user interface that provides a control panel.

3 Requirements

I created a three.js scene for starters that englobes multiple components, as shown in Figure 1.0. We can see that the scene has a camera, lights, and different rendered component. We also define the 3D into three axes x, y and z. The cube is a 2*2*2 cube centred in origin, while the object is a loaded object representing a bunny explained in one of the requirements of this report.

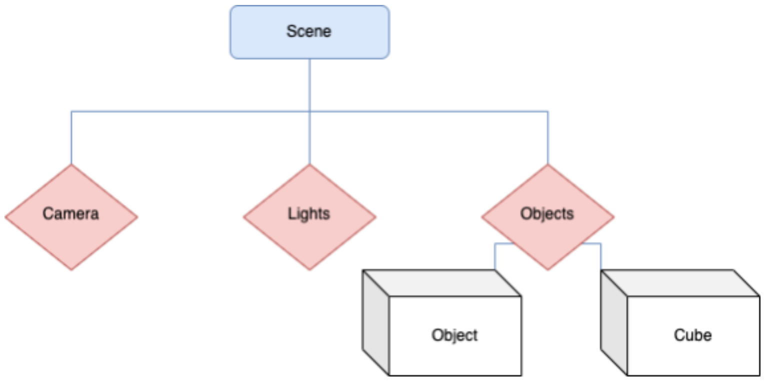


Figure 1: Diagram representing components in a ThreeJS scene

3.1 Rotation

3.1.1 Explanation

The first requirement highlighted in the report is the rotation of the cube or the object. This 3D function lets us rotate the object regarding the three axes. Assuming that the camera and axes are fixed, the rotation of an object in a 3D plane is made using Euler angles[2]. Euler angles are three angles (x,y,z) in our case, where they describe the object orientation with the assumption made above[3]. Three JS rotations angles are called intrinsic Tait-Bryan angles. The difference between Euler angles and these tait-Bryan angles is the number of axes rotated in respect with.

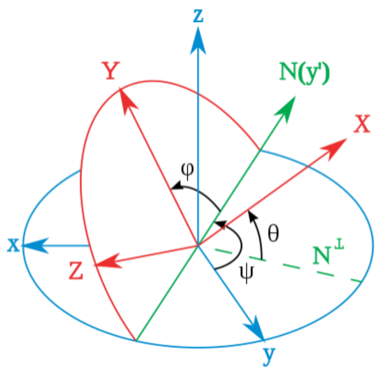


Figure 2: Figure representing the Euler Angles

As you can notice in Figure 2.0, we first have the tree fixed axes x,y,z. An order is specified, which define the order of rotation. A possible combination is " X, Y, Z" by default. Initially, X, Y and Z are in the same position as x,y,z. The capital letters axes get updated after each orientation by rotation using three Euler angles.

3.1.2 Implementation

The graphic user interface presents a tickbox for cube rotation and object rotation. When these commands are pressed, global variables representing these options will be set to 1, and if they are turned off, it will be set back to 0. The scene has a function animate that gets run with the machine's frame rate. This shows that it is faster than the human eye leading to a continuous animation. I called the rotations method inside the animate function. The code snippets are presented below:

```
if (drawCube == 1){
    // Rotate about the x axis
    if (axesRotationX == 1) {
        cube.rotation.x += 0.01
    }
}
```

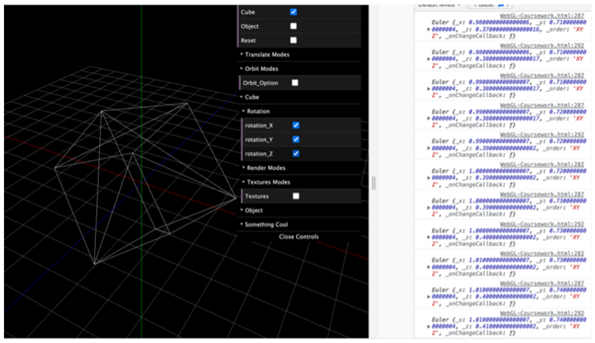
The rotation is applied if the object is loaded and rendered. Then when one of the global variables is set, we can keep adding 1 to the according to rotation vector, representing the Euler angles explained above. We can rotate on multiple axes at the same time.

3.1.3 Testing

Testing was mainly done trying to rotate in regard to multiple axes simultaneously. Figure 3.0 shows the cube at rest after being rotated in the axes for a random amount. It also shows the vector values being printed into the console. We can see that the Euler angles are changing by 0.01 at the rate of the picture frame, which gives a smooth rotation.

3.1.4 Capabilities and limitations

This way of implementation was the simplest way to rotate an object over fixed axes in the Three Js library. The rotation is set at adding 0.01rad to the cube rotation Euler angle. I used the online converter [4] to convert Euler angles to axis angles in degree. We are adding approximately 0.579° to the respective rotation angle at the frame rate of the machine when the respective global variable is set. Increasing the constant to 0.05 will give an addition of almost 2.87° per frame. This process ensures smooth rotations. However, one limitation was the control of speed rotation. We need to manually change the constant to change the speed rotation and cannot do it while the program is running. The second limitation is that one axis rotates over only pi, whereas the other two rotate over 2π.



```
WebGL-Coursework.html:287
Euler {_x: 0.9800000000000006, _y: 0.710000000
0000004, _z: 0.3700000000000016, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:292
Euler {_x: 0.9800000000000006, _y: 0.710000000
0000004, _z: 0.3800000000000017, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:282
Euler {_x: 0.9900000000000007, _y: 0.710000000
0000004, _z: 0.3800000000000017, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:287
Euler {_x: 0.9900000000000007, _y: 0.720000000
0000004, _z: 0.3800000000000017, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:292
Euler {_x: 0.9900000000000007, _y: 0.720000000
0000004, _z: 0.3900000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:282
Euler {_x: 1.0000000000000007, _y: 0.720000000
0000004, _z: 0.3900000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:287
Euler {_x: 1.0000000000000007, _y: 0.730000000
0000004, _z: 0.3900000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:292
Euler {_x: 1.0000000000000007, _y: 0.730000000
0000004, _z: 0.4000000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:282
Euler {_x: 1.0100000000000007, _y: 0.730000000
0000004, _z: 0.4000000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:287
Euler {_x: 1.0100000000000007, _y: 0.740000000
0000004, _z: 0.4000000000000002, _order: 'XY
Z', _onChangeCallback: f}
WebGL-Coursework.html:292
Euler {_x: 1.0100000000000007, _y: 0.740000000
0000004, _z: 0.4100000000000002, _order: 'XY
Z', _onChangeCallback: f}
```

Figure 3: Screenshort representing rotation testing

3.2 Object Loading

3.2.1 Explanation

I was responsible for loading an object to the scene for this spec. The object was to be scaled and translated to always fit inside the cube. This model represented a mesh model and was a Stanford bunny. A loaded object presents children of meshes and vertices leading to geometries and materials. To load the object in different render modes, we need to traverse the children of this object, creating new material each time. To scale and translate the 3D object, we need to get the scaling and translating vector. This can be calculated by stating: $scaling = cubeSize/(objectsize * 1.5)$ The use of 1.5 is to minimize the bunny inside the cube for testing purposes and to notice the bunny's presence inside his box. The translating vector for the bunny can be determined using the cube's position. Internally, an affine 3D transformation will be performed to find the new object pixels location. The matrix uses homogeneous coordinated leading to:

$$\begin{pmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

And the new 3D model pixel position is represented by s'

$$\begin{pmatrix} s^{'x} \\ s^{'y} \\ s^{'z} \\ 1 \end{pmatrix} = M * s^{'}$$

3.2.2 Implementation

We loaded the object using an ObjectLoader() function and assigned it a MeshPhongMaterial by default for all children using the traverse function. We can change these materials or the type of loaded object to points to render different objects mode (edges, vertices, and cube). A temporary default object was loaded to facilitate the switching between different render modes. I looped through the children array of the loaded object and determined its bounding box with the computeBoudingBox() function shown below in code snippets 3.0:

```
 objectLoader.load('bunny-5000.obj', function ( object ) {
      for ( var i = 0; i < objectLoaded.children.length; i++){
           temp = new THREE.Vector3(0,0,0);
           objectLoaded.children[i].geometry.computeBoundingBox();

           temp.x = Math.abs(objectLoaded.children[i] .geometry.boundingBox.max.x)
           + Math.abs(objectLoaded.children[i].geometry.boundingBox.min.x);

           if (temp.x > sizeObject.x) {
               sizeObject.x = temp.x;
           }
           scalingObject.x = cubeSize.x / (sizeObject.x*1.5);
```

The geometry.boudingBox.max and geometry.boudingBox.min vectors give me the maximum and minimum coordinates of the bunny. The addition of the absolute values of these values gives the size of this descendant of the bunny. For all the children, we take the maximum possible size vector. The same function was used to calculate the size of the box, and the scaling vector can now be calculated. The object loaded has a function called scaling.set and position.set, which computes the affine transformation of the bunny.

3.2.3 Testing

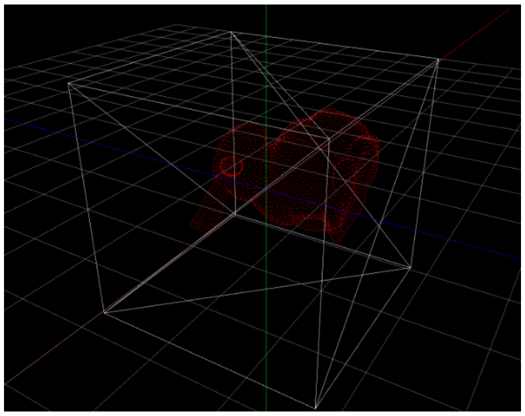


Figure 4: Screeshot representing object loading testing

Looking at Figure 4.0, we can see that the bunny is loaded with the cube in different render modes. I applied rotation for the loaded object to ensure its position inside the cube at any Euler angle combinations.

To be more precise, I translated the bunny to be placed at the origin, like the cube in the middle of the cube.

3.2.4 Capabilities and limitations

The bunny was correctly loaded and fit inside the cube for all possible rotations. The code also works for all objects regardless of the number of descendants, as we are always taking the maximum size of the descendants. However, if we did not multiply by 1.5 the size of the cube, we could have seen an error in a rotation where the object touches a side of the cube. A possible implementation is to take the smallest size of the cube as the cube size vector and then divide by this to get the maximum possible scaling vector causing the object to always be inside the cube and no need of multiplying by 1.5.

3.3 Translating

3.3.1 Explanation

Camera control had to be implemented without using any native library. These controls translate the camera: left and right, up and down, and forward and backwards. These represent the horizontal, vertical and z axes, respectively. Translating the camera is simple and requires using the affine transformation mentioned before. However, the approach to do a smooth translation will be explained below. To control this camera, I used key events in JavaScript: Pressing down and releasing the keyboard key.

3.3.2 Implementation

I also implemented a way to ensure a smooth translation. I created a vector called cameraSpeed, which stores the translation amount for each axis. We take the time elapsed from login at the beginning and end of the function and subtract them from each other. I update the camera speed vector by subtracting it from itself but multiplying by a constant 12.0 and the resulting time. When one of the keys is pressed, the global variables are updated depending on where we are moving. I am subtracting the time multiplied by 200 to the respective axis. This way, I can translate multiple axes simultaneously with a smooth movement. This way, when it is at rest, I am only translating to the "current new location". The code snippet is represented below.

```
var time2 = performance.now();
var time = ((time2 - time1) / 1000);
cameraSpeed.x -= cameraSpeed.x * 12.0 * time
   ;

if ( camLeft ) cameraSpeed.x -= 200.0 * time
   ;
if ( camRight ) cameraSpeed.x += 200.0 *
   time;

camera.translateX( cameraSpeed.x * time );
time1 = time2;
```

3.3.3 Testing

Combinations	Outcome
Forward and Backwards Testing	True
Left and Right Testing	True
Up and down Testing	True
Forward – Left – Up	True
Backward – Right – Down	True

Table 1: Table Representing testing strategy of camera translation

To test the movement, I printed the value of the vectorSpeed in the console to ensure a non-moving camera at rest, and I tested the key pressed simultaneously shown in table 1.0 above. The figure below shows the translation output after moving the camera in the possible direction:

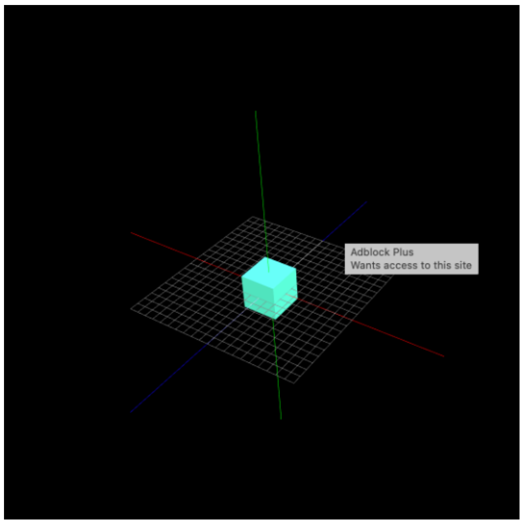


Figure 5

3.3.4 Capabilities and limitations

This smooth way of translating the camera solved my orbit requirement where I am updating the lookAt vector to the position of the object loaded or cube. This works, but a limitation at 180degrees across the z-axis, I need to flip the camera to ensure a 360° rotation in latitude. Mouse control could be implemented to ensure a quick response update and better user experience, allowing you to control speed movement. However, the trade-off of facilitating the orbit function explains my choice to handle the translation with key and not the mouse wheel event.

3.4 Extension Game

3.4.1 Explanation

For the last requirement, an extension, I used all the concepts learned above to create a small game. This 3D game shows an animation of multiple objects. The game's goal is to shoot and hit targets of a moving object in a 3D plane. Your accuracy is calculated, and there is a level number with increasing difficulty. The hard part of the implementation was to create a smooth experience and an accurate game. The concepts of scaling, translation, loading objects were used in this animation.

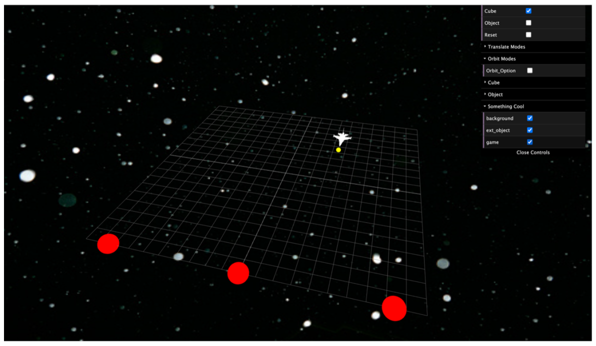


Figure 6: Figure representing the extension game setup

As you can see in Figure 6.0, I uploaded a background image representing the stars for the game. I also loaded multiple objects for the setup: A main f15 flying vessel with unlimited ammunitions represented by a small sphere—multiple targets represented by larger radius spheres. Objects sizes were calculated using the computeBoudingBox function and stored as vectors. The vessel object is translated horizontally over a range of -5 to 5 with a constant speed; however, the speed increases and impacts the object's faster movement at each level. On a pressed keyboard event (enter), the ammunition is translated over the z-axis. The x and z position of the ammunition and the target are calculated to add the radius of both the small and large spheres to ensure an accurate hit. If the target is hit, which means if the position of the small sphere plus its radius is within the position of the large sphere plus its radius, the target disappears as shown in the code snippet below:

3.4.2 Implementation

```
if( sphereAmmun.position.z <= sphereTarget_1.position.z + 0.3 && sphereAmmun.position.z >= sphereTarget_1.position.z -0.3) {
 if ( sphereAmmun.position.x <= sphereTarget_1.position.x + 0.3 && sphereAmmun.position.x >= sphereTarget_1.position.x - 0.3 ){
     console.log("Hit Sphere 1");
     sphereHit_1 = 1;
     }
 }
```

When the ammunition reaches a specific far away value over the z-axis, we can replace it at its original position, creating an illusion of new ammunition being placed.

When all the targets had been hit, I added two more targets and increased the vessel's speed, as shown in Figure 5.0. I added a counter for the ammunition shot, and the number of red balls is calculated using:

$nTargets = levels * 2 + 3$

The accuracy is then determined and printed to the console.

3.4.3 Testing

To test the program, I played many times the game with different levels and tested using different users. I had to try the accuracy of the small sphere hitting the targets and reduce timing delays effects on the game.

3.4.4 Capabilities and limitations

Many limitations are presented in the game, the most issues seen:

- Timing delays decrease the accuracy of the ammunition being translated over the z-axis.
- At some high levels, the vessel translation reaches a translation screen that causes the machine to bug, and the game starts to get unresponsive and slow.

The game could also be improved by adding moving targets orbiting the camera around the pitch, creating a new experience.

4 Conclusions

In conclusion, all the requirements were to learn about 3D animations implemented in WebJl and especially ThreeJs. As shown above, we demonstrated one of the applications of 3D animation in the world: the gaming industry. However, 3D animations have been improving and getting more accessible as an implementation. Many companies are now focused on using 3D representation in healthcare, art, and virtual reality. The world is evolving, opening new areas like meta-universe based on the 3D model.

5 References

https://github.com/dataarts/dat.gui

https://threejs.org/docs/index.html?q=Object#api/en/core/Object3D.rotation

https://en.wikipedia.org/wiki/Euler_angles

https://www.andre-gaschler.com/rotationconverter/