# CS323 Assignment Report

Sion Griffiths - sig2

November 18, 2015

# Contents

# 1 Introduction

This report will detail the steps undertaken in constructing a visualisation of the Sun, Earth and Moon using WebGL and specifically the Three.js library. It should be noted that for this visualisation and throughout this document, the horizontal axes in the system are X and Z and the vertical is Y.

To access the visualisation use the following link - `http://users.aber.ac.uk/sig2/cs323/CS323_SolarSystem/`

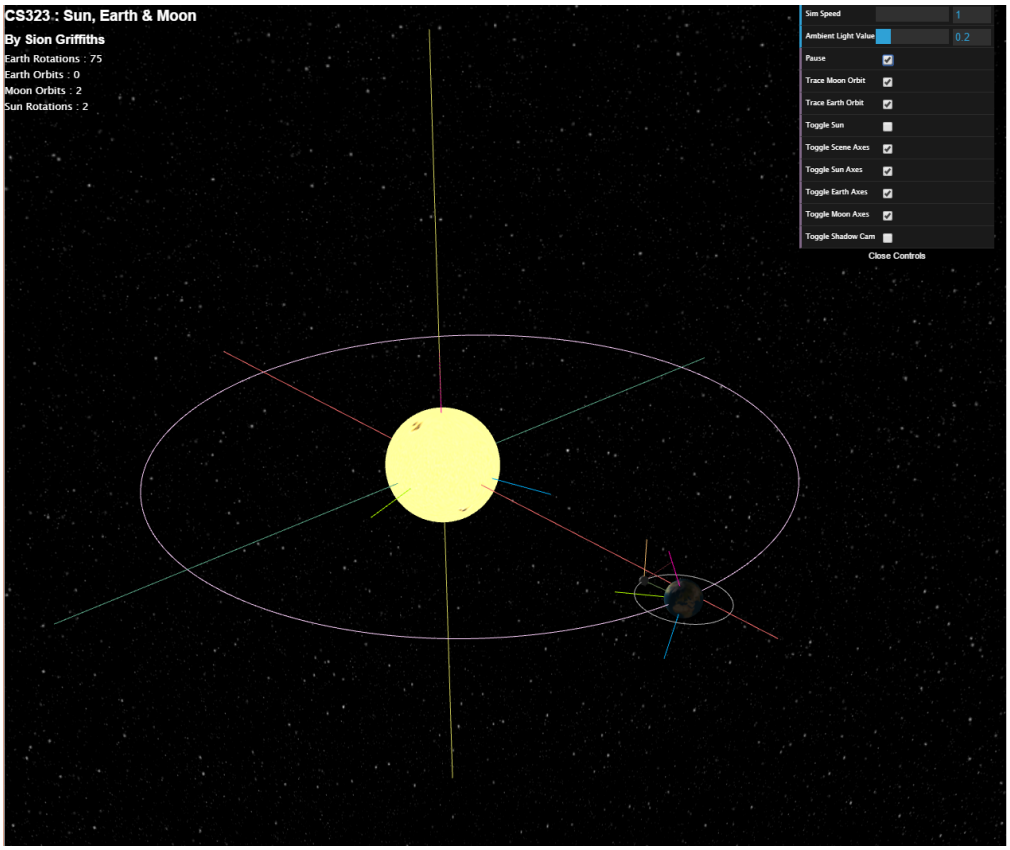


Figure 1: A screen capture showing an overview of the developed system

## 1.1 A note on image figures

Unless otherwise stated, all images included in this document should be recreatable using the GUI options shown and careful manipulation of the camera using the standard orbit controls. If no options are shown or information given in the caption, default parameters should be assumed.

# 2 Structure & Implementation

The code developed for this assignment can be split into 2 distinct packages, Model and Utils. The javascript files contained within the model package are essentially the descriptions of the entities within the system, the Earth, Sun and Moon. These files contain data regarding the state of the entities (geometry etc) along with functions which control their behaviour, for example the 'update' functions which control the changes to the entities(position etc) each iteration of the main logic loop.

The utils package contains utility code, split across various files depending on the nature of the code. The package contains files with utility code for the following: matrix functions, simple geometry functions and functions relating to the computation of orbits.

Separate from these two packages is the main.js file. This file serves as the entry point and initialisation of the system along with controlling the updating of the various parts of the system, entities, camera, GUI variables etc.

The approach taken was an iterative one, starting with implementing the basic requirements using the native Three.js functionality, this allowed a basic understanding of the task at hand and provided a means to further familiarise myself with the Three.js framework. Once basic functionalities were implemented, prototyping of the extended requirements could begin, again in an iterative manner, with ongoing testing throughout. Testing was done mostly via careful observation and use of debugging tools native to browsers and the WebStorm IDE provided by JetBrains.

# 3 Orbits

## 3.1 Basics

### 3.1.1 Circular orbits

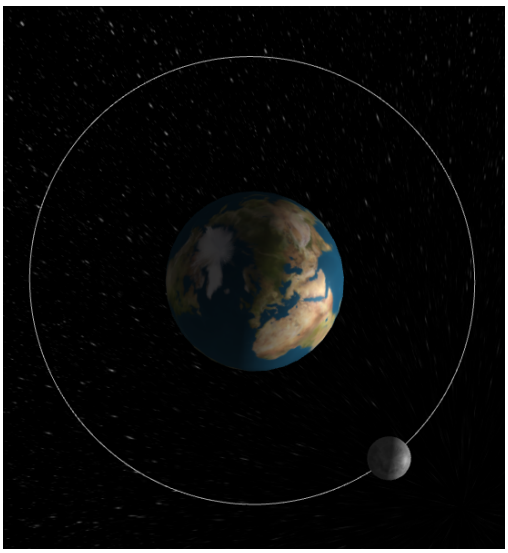

Figure 2: Screen captures displaying the circular orbit of the Moon around the Earth, a code change is required to display a circular orbit within the finished system. This can be achieved by setting the first parameter of generateElliptical() to 0 on line 88 of main.js

During the initial prototyping stage of development, the orbits of the Earth and Moon were implemented as simple circles. This was achieved using simple trigonometry to update the relative positions on the x and z axes. The function to achieve this is shown in the code snippet below,:

```
1  var earthDistanceFromSun = 24;
2  var earthRotationSpeed = 0.2;
3
4  var eathRot = function(earthMesh, sunMesh){
5
6      var x = earthDistanceFromSun * -Math.cos(earthRotationAngle * (Math.PI / 180));
7      var z = earthDistanceFromSun * -Math.sin(earthRotationAngle * (Math.PI / 180));
8      earthRotationAngle-= earthOrbitRotationSpeed;
9
10     earthMesh.position.x = sunMesh.position.x + x;
11     earthMesh.position.z = sunMesh.position.z + z;
12 }
```

Listing 1: Simple orbit calculation

As the value for `earthRotationAngle` decreases with each iteration of the function, the positions calculated will 'swing' around an origin point. The calculated positions are added onto the sun's central position and applied to the Earth's position, such that the Earth will appear to circle around the sun.

#### 3.1.2 The earth orbits the centre of the sun & The moon orbits the centre of the earth

Using the technique described above, it's clear to see that this can be applied to any mesh to ensure that it's rotation is around the centre of another. Even if the other mesh has a dynamic position (as, in the case of the Moon orbiting the Earth as the Earth is moving around the Sun.), as the positions applied are relative as opposed to being absolute. The relative positions can be added to any other position to create the desired circular orbit.

### 3.2 Extended

#### 3.2.1 Elliptical orbits & Non-uniform orbital velocities

Using the information given within the assignment brief, elliptical orbits and non-uniform orbital velocity functionality was implemented within the system. The provided equations, once translated into code, allowed an elliptical orbit to be pre-computed as a list of vectors that would then be traversed and applied as position for the centre of an orbiting mesh to provide an animated orbit.

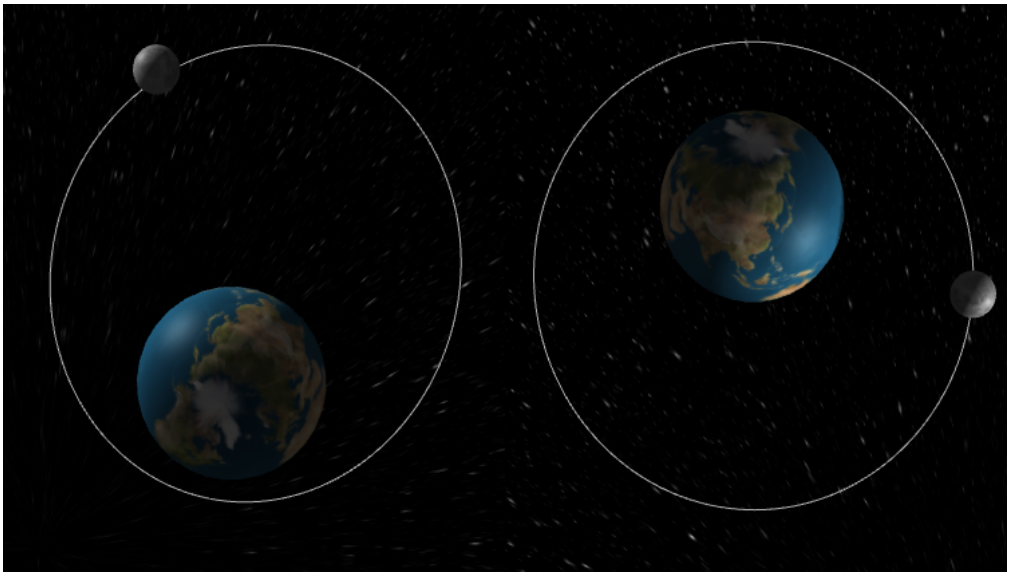

Figure 3: Screen captures displaying the orbit of the Moon calculated with different eccentricity values. 0.5 on the left, 0.3 (system default) on the right. In order to change this value the code needs to be changed in main.js, line 88. The first parameter for the function generateElliptical()

The distance between the list of vectors generated for the orbit is not constant, the effect of this is a non-uniform orbital velocity as the points are traversed in the animation, giving an approximation of planetary motion.

The code for generating the elliptical is shown below:

```
1 this.generateElliptical = function(eccentricity, periodSlices, semiMajorAxis, tiltValue){
2
3     var orbitVerts = [];
4     var theta = 0;
5     var r = 0;
6
7     while(theta <= 2*Math.PI) {
8         theta += computeTheta(eccentricity,theta,periodSlices);
9         r =  computeR(eccentricity,theta,semiMajorAxis);
```

```
10
11          //negate the result for X to achieve anti clockwise orbital rotations
12          var x = -polarXtoCart(r,theta);
13          var z = polarZtoCart(r,theta);
14          orbitVerts.push(vec3(x,0,z));
15      }
16      return orbitVerts;
17  };
18
19  //((2/N)( 1+ e cos ? )^2)/(1-e^2)^(3/2)
20  var computeTheta = function(e,theta,periodSlices){
21      return ((2/periodSlices)* Math.pow((1+ (e*Math.cos(theta))),2))  / Math.pow( (1-Math.pow
            (e,2)) ,(3/2));
22  };
23
24  //r = a(1-e^2)/(1+ e cos ? )
25  var computeR = function(e, theta, a){
26      return (a*(1-Math.pow(e,2)))/(1+e*Math.cos(theta));
27  };
```

Listing 2: Above code can be found in OrbitUtils.js and is adapted from the formulas given in the assignment brief.

Similar to the circular orbit described in an earlier section, the coordinate points are relative, they are generated with the scene origin (position 0,0,0) as the centre of their orbit. In the case of the Earth, since the centre of the Sun is at the origin point (0,0,0) no further action is required beyond specifying the orbit distance as the `semiMajorAxis` parameter in the above `generateElliptical` function. However, for the Moon to orbit the centre of the Earth we must apply these relative orbit positions to the current position of the Earth. The code to do so is detailed below:

```
1  this.getMesh().position.x = earth.getX() + this.orbitPoints[count].x;
2  this.getMesh().position.y = earth.getY() + this.orbitPoints[count].y;
3  this.getMesh().position.z = earth.getZ() + this.orbitPoints[count].z;
```

Listing 3: Code taken from Moon.js showing the moon mesh position being set as that of the Earth plus the relative axis positions from the orbital point list

### 3.2.2 Constant tilt of the Moons orbit

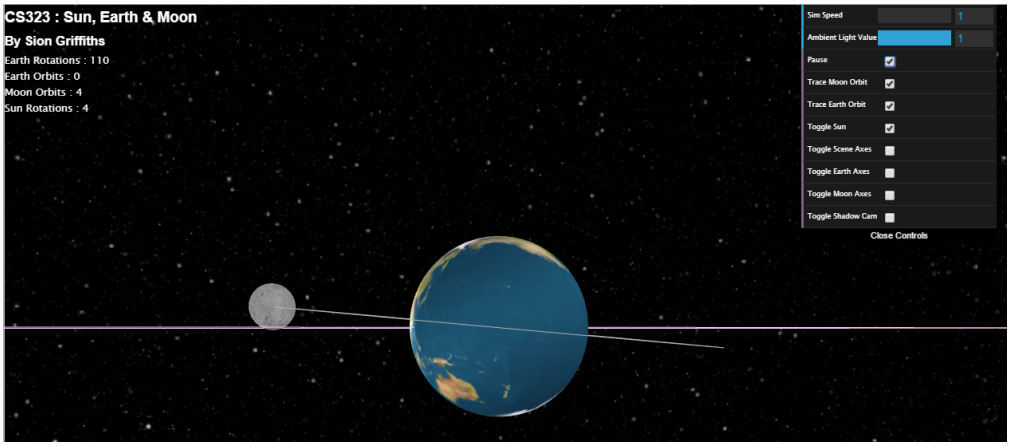


Figure 4: Screen capture showing a trace of the Moon's orbit (gray) compared to the Earth's (pink)

The process of applying a constant tilt to the Moon's orbit is fairly straight forward. Given a list of vectors defining positional data for the orbit(as generated by the elliptical orbit code in the above section for example) and the desired angle of rotation, a matrix can be applied to each vector in the list in order to apply the desired transform.

In the case of the Moon's orbit we need to apply a rotation through a horizontal axis to achieve a tilt in the vertical. In this case the Z axis. The angle of rotation is passed into the below function and converted to radians before being applied as a standard z-rotation matrix.

```
this.applyTiltToOrbit = function(angle, points){
    var tiltRotation =  getZRotationMatrixAsMat4(angle);
    for(var i = 0; i < points.length; i++){
        points[i] = applyMat4toVec3(tiltRotation,points[i]);
    }
    return points;
};
```

Listing 4: Above code can be found in OrbitUtils.js

```
function getZRotationMatrixAsMat4 (angle){

    var radians = deg2rad(angle);
    var mat4 = new THREE.Matrix4();

    mat4.set(
        Math.cos(radians), -Math.sin(radians), 0, 0,
        Math.sin(radians), Math.cos(radians), 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    );

    return mat4;
}
```

Listing 5: Definition of a rotation matrix in the Z axis. Can be found in MatrixUtils.js

# 4 Textures and lighting

## 4.1 Basics

### 4.1.1 The sun, earth and moon shown as texture mapped spheres

The spherical entities in the system (Sun, Earth and Moon) are meshes consisting of texture mapped sphere geometries. The geometries are defined as Three.js native SphereGeometry.

Texture mapping of the meshes was achieved using Three.js native loadTexture functionality, an example of the Earth mesh being defined is below:

```
this.geometry = new THREE.SphereGeometry(4, 32, 32);
this.material = new THREE.MeshPhongMaterial();
this.material.map = THREE.ImageUtils.loadTexture('assets/images/earthmap1k.jpg');
```

Listing 6: Above code can be found in sun.js

### 4.1.2 The sun shown as a self-illuminated sphere

In order for the Sun to appear as self illuminated, the material for its mesh has been defined as emissive. The sun texture image itself is used as an emission map and a colour parameter is given.

```
this.material =  new THREE.MeshPhongMaterial({
    emissive: 0xF2E9A6, //emission colour
    emissiveMap: new THREE.TextureLoader().load("assets/images/sunmap.jpg")
});
```

### 4.1.3 Earth and moon lit by a single point light source located at the centre of the Sun

This is the only requirement which could be considered unfulfilled. Indeed there is no point light in the centre of the sun. However, the Earth and Moon are lit from a light source at the centre of the Sun, in order for shadows to be implemented it was necessary to use a source other than point light since it does not support such functionality. This is a limitation imposed by Three.js for efficiency sake. Please see section 4.2.2 below for details of the shadows.

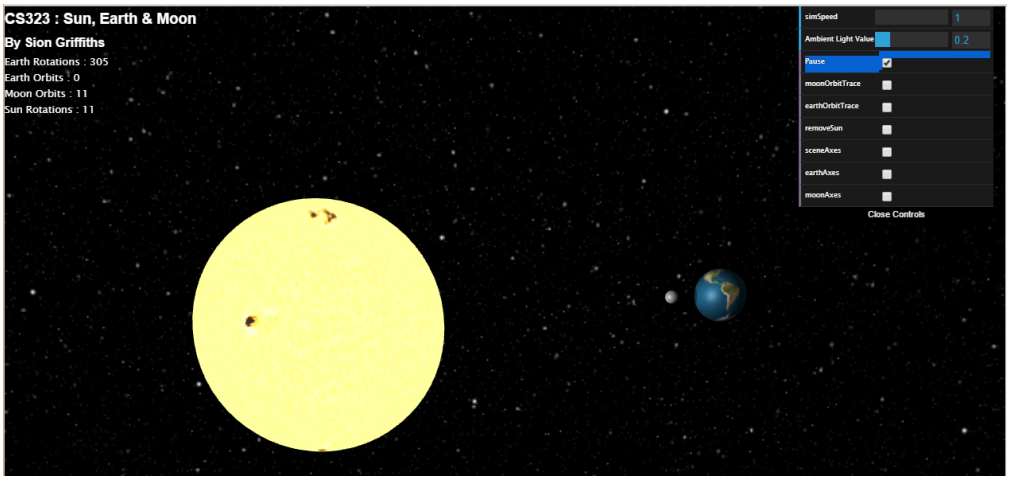


Figure 5: A screen capture showing texture detail

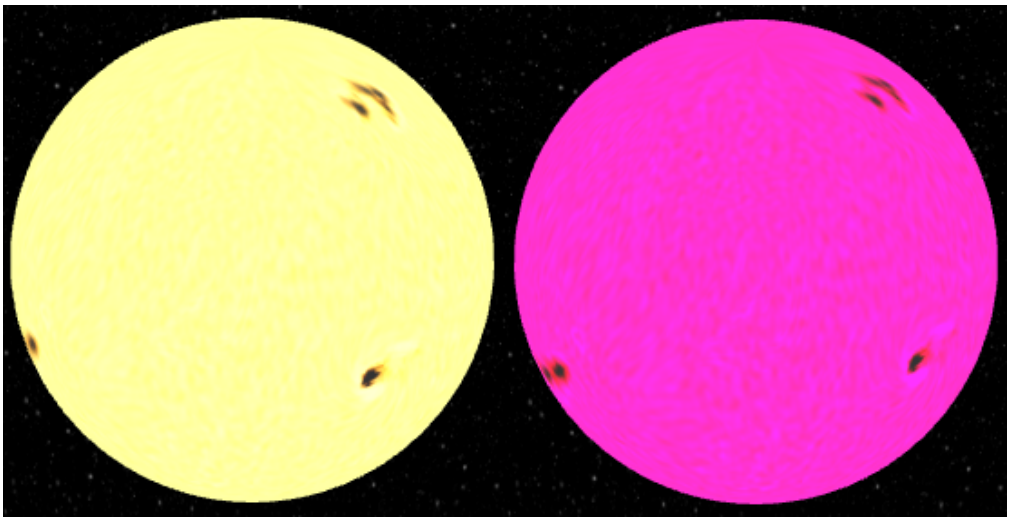

Figure 6: Emissive texture with different colours. The colour can be changed by using a different hex value on line 12 in sun.js

## 4.2 Extended

### 4.2.1 Lighting of the earth and moon in Phong shading

Phong shading is implemented in the system for the Earth and the Moon via native Three.js functionalities. The desired shading is achieved through definition of mesh material as `THREE.MeshPhongMaterial()`, inbuilt Three.js functionality then correctly configures the shaders for these meshes if a compatible light is used, for example PointLight or SpotLight.

### 4.2.2 Non-illuminated parts of the earth and moon to be shown in shadow & Eclipse shadows

The shadows in the system are achieved via use of a SpotLight, located in the Sun. The SpotLight uses native Three.js functionality to cast shadows on meshes which have been explicitly enabled for shadows via configuration of the `mesh.castShadow` and `mesh.receiveShadow` parameters. The SpotLight gives the effect of radiating light from the sun in all directions but is in fact targeting the earth explicitly with each animation update.

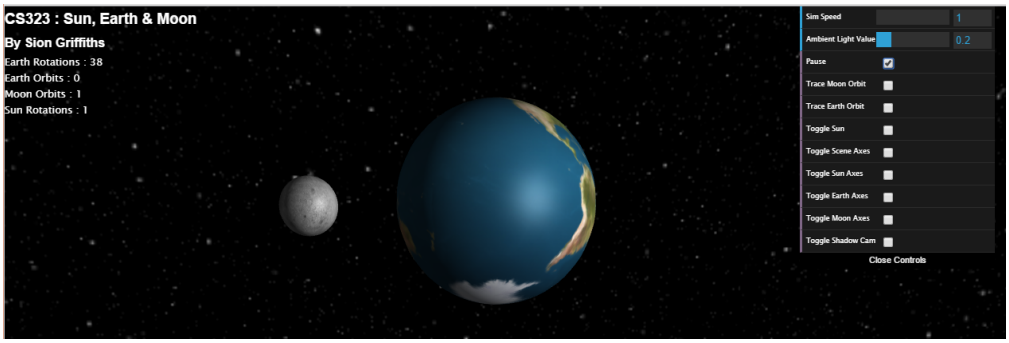


Figure 7: Screen capture showing specular reflection details

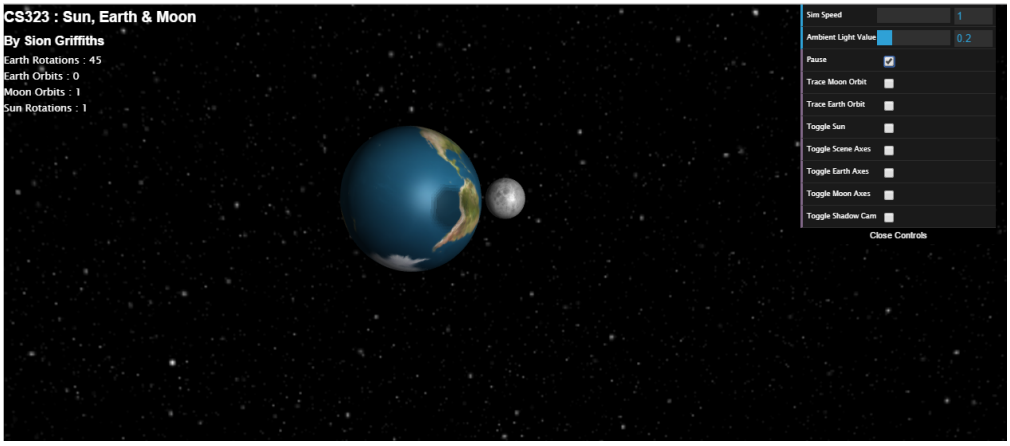


Figure 8: Screen capture detailing the moon's eclipse shadow

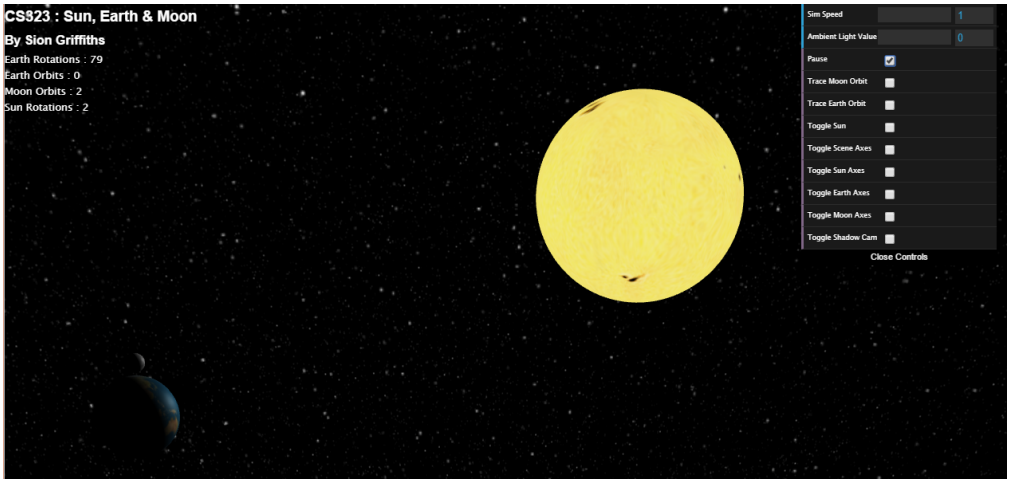


Figure 9: Screen capture showing shadow of non-illuminated surfaces

# 5 Axes and tilts

## 5.1 Basics

### 5.1.1 The sun, earth and moon each spinning on their own axes

Using the GUI options to enable axes on the sun, earth and moon, it is clear to see all meshes rotating on their own axes. This could be achieved very simply with Three.js by using the `rotation.axes` functionality

of a mesh and incrementing the value with each animation update. For the completed a different approach is taken and is detailed in a section below.

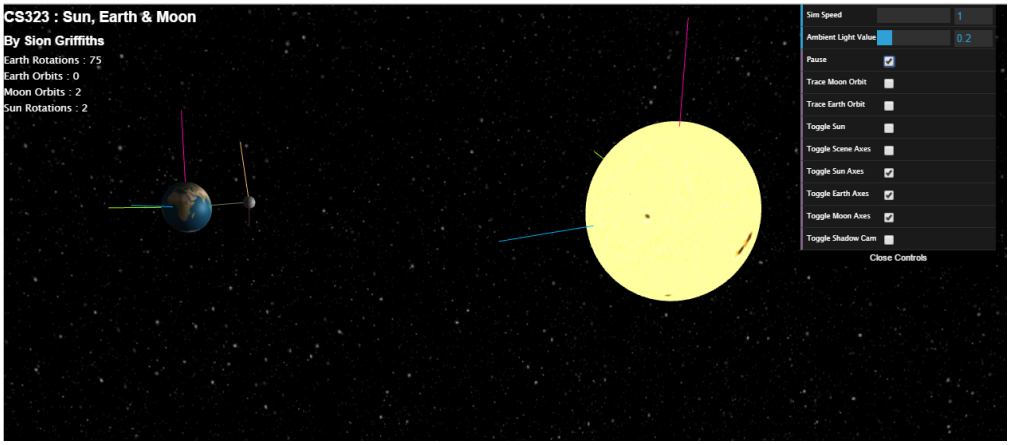


Figure 10: Screen capture showing all axes

## 5.2 Extended

### 5.2.1 Constant tilt of the Earths axis & Earth's rotation

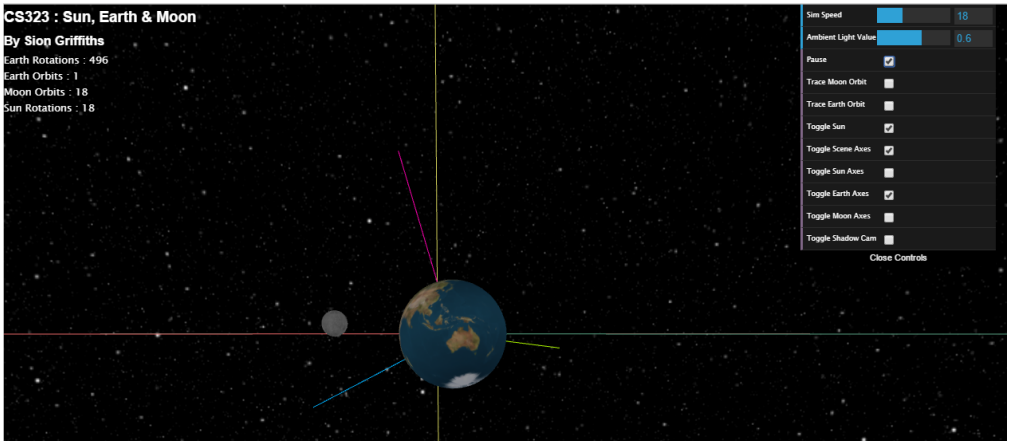


Figure 11: Screen capture showing tilt of earth axes compared to the scene axes

The tilt of the Earth's axis is essentially defined by a matrix rotation in a horizontal axis. This is applied to the Earth mesh during initialisation using the Three.js `applyMatrix` functionality. The code to define an initial axial tilt of 23.4 degrees is shown below:

```
1 this.axialTiltMatrix = getZRotationMatrixAsMat4(23.4);
2  this.mesh.applyMatrix(this.axialTiltMatrix);
```

Listing 7: Code taken from Earth.js

The situation becomes slightly more complex when we need to define further matrix transforms on the same mesh, for instance in the case of the Earth, we'd like to rotate around the tilted axis. It is also desirable to be able to control the number of rotations completed in one orbital period such that a year in days can be simulated.
In order to properly define a matrix to describe the axial rotation of the Earth we must calculate the amount of rotation per animation update. We can calculate the number of orbit points traversed in one 'day' with the following code, since the system will traverse one orbitPoint index per update :

```
this.pointsInDay = (this.orbitPoints.length/365.26);
```

Listing 8: Code taken from Earth.js

We can then use this `pointsInDay` value to calculate a rotation angle by simply dividing a whole rotation (360 degrees) by this value. We can then scale this value by the global simulation speed setting to keep everything synchronized as the user changes the speed. Once we have this value we can rotate the Earth mesh correctly per update. In order to rotate correctly and be able to preserve an axial tilt, the tilt matrix must be removed ( by applying the opposite rotation), the rotation then applied and the tilt reinstated. The code to do so is detailed below:

```
//calculate rotation per update
rotValue = 360/this.pointsInDay*Math.floor(this.globalVars.simSpeed);
this.rotationMatrix = getYRotationMatrixAsMat4(rotValue);

var transToOrigin = new THREE.Matrix4().makeTranslation( -this.getX(), -this.getY(), -this.
    getZ());
//translate mesh to origin before applying transforms
this.mesh.applyMatrix(transToOrigin);

this.mesh.applyMatrix(this.removeAxialTiltMatrix);
this.mesh.applyMatrix(this.rotationMatrix);
this.mesh.applyMatrix(this.axialTiltMatrix);

//update position in pre-calculated orbit array
this.mesh.position.z = this.orbitPoints[count].z;
this.mesh.position.x = this.orbitPoints[count].x;
```

Listing 9: Code taken from Earth.js

After the rotation, we can explicitly set the position of the mesh from the orbital points array without having to first translate from the origin since in this case the positional values in the orbital points are relative to the origin..

#### 5.2.2 Synchronous orbital and axial rotations of the moon

Due to the nature of the Moon's motion when using an elliptical, non-constant velocity orbit it was decided to use a shortcut method to achieve the synchronisation of orbit and rotation: using the Three.js `lookAt()` function. Essentially this will set a constant face to the Earth for each animation update.

# 6 User Interface

## 6.1 Suitable user control interface for viewing, scaling, timing

A third party GUI framework, dat.GUI, was used to achieve the user controls for parameter settings along with using the OrbitControls.js bundled with Three.js to allow zooming and camera manipulations.

The implementation of the parameter controls was straight forward and well documented online. A list of variables to be controlled is defined with initial values:

```
var guiVars = {
    ambientLightIntensity : 0.2,
    moonOrbitTrace : false,
    earthOrbitTrace : false,
    removeSun : false,
    paused : false,
    sceneAxes : false,
    earthAxes : false,
    moonAxes : false,
    sunAxes : false,
    shadowCam : false,
    simSpeed : 1
```

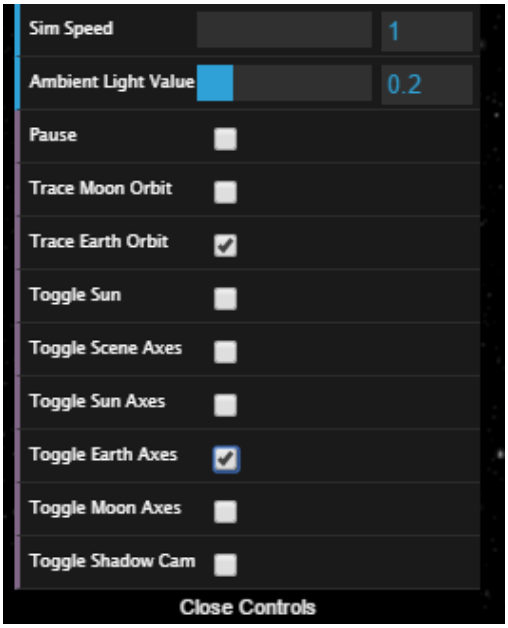


Figure 12: Screen capture showing the GUI provided through dat.GUI

```
13 };
```

Listing 10: Code taken from main.js

These are then added to a new dat.GUI object and logic based on their state can be implemented.

# 7 Running the System

As mentioned in the Introduction the visualisation can be accessed via `http://users.aber.ac.uk/sig2/cs323/CS323_SolarSystem/` Due to browsers blocking cross origin requests for assets it is not feasible to run the system correctly locally without the use of extra software which may not be available on the test machines. It is assumed then that the test machines will be equipped with a browser and able to access the university public html.

# 8 Credits

- dat.GUI for the UI
- Paul Bourke for the starmap
- planetpixelemporium for textures
- learningthreejs for various tutorials

# 9 Self Assessment

| Basic Task | Documented | Implemented |
|---|---|---|
| 1 | Yes | Yes |
| 2 | Yes | Yes |
| 3 | Yes | Yes |
| 4 | Yes | Yes |
| 5 | Yes | Yes |
| 7 | Yes | Yes |
| 7 | Yes | Yes |

| Extension | Documented | Implemented |
|---|---|---|
| a | Yes | Yes |
| b | Yes | Yes |
| c | Yes | Yes |
| d | Yes | Yes |
| e | Yes | Yes |
| f | Yes | Yes |
| g | Yes | Yes |
| h | Yes | Yes |
| i | Yes | Yes |
| j | Yes | Yes |

| | |
|---|---|
| Animation can be viewed in B57, C56 without additional software | Yes |
| All reference sources are acknowledged | Yes |
| Overall self-assessment out of 50 | 35 |