CS 32310: 'Virtual Sun-Earth-Moon System'

Due on Thursday, November 19, 2015

James Euesden - jee22

Contents

Introduction **3**

Simulation **3**
Parameters . . . 3
Structure . . . 4
Running the program . . . 5

Basic Features **5**
The earth orbits the centre of the Sun . . . 5
The moon orbits the centre of the Earth . . . 6
Circular orbits . . . 6
The Sun, Earth and Moon shown as texture mapped spheres . . . 7
The Sun, Earth and Moon each spinning on their own axes . . . 8
The sun shown as a self-illuminated sphere . . . 11
Earth and Moon lit by a single point light source located at the centre of the Sun 12

Additional Features **13**
User Interface . . . 13
Synchronous orbital and axial rotation of the Moon . . . 15
Lighting of the Earth and Moon in Phong shading & Non-illuminated parts of the Earth and Moon to be shown in shadow . . . 16
Constant tilt of the Earth's axis . . . 19
Constant tilt of the Moon's orbit . . . 19
Elliptical Orbits & Non-uniform orbital velocities: Kepler's 2nd Law of Planetary Motion . . . 20
Eclipse Shadows . . . 23
Other features of my choice . . . 24

Self Assessment **30**

Introduction

This assignment tasked me with using WebGL to create and display a simple animated model of the Sun-Earth-Moon system [1], requesting the basic features of display (three spheres with textures, orbits and rotations). There was also the option to implement additional functionality (lighting and shading, tilted axial and orbital rotations, etc). The model was not requested to be accurate in sizes, distances or times of rotations, although real world values were provided for reference.

Simulation

Parameters

In order to create the simple model, I used some arbitrary values that made the system look somewhat accurate without truly being accurate. This was necessary in order for a viewer to see much of anything, as real values would leave the Sun too large compared to a minuscule Earth and Moon that would also be very, very far away from the camera focus. These parameters are:

```
// Large, so we can actually see something with our model. This controls the scaling of the Earth and Moon's orbital eccentricity.
var eccentricityModifier = 50;

// Size of the Earth, scaled larger for the visualisation.
var earthSize = 10; // from -> 1 * 10;
// The distance between the Sun and Earth, scaled for the visualisation.
var earthDistanceFromSun = 156; // from -> 23500 / 150;
// The orbital period of the Earth around the Sun (365 days).
var earthOrbitalPeriod = 365;
// The 'speed' of the rotation of the Earth on its axis - The period of time (9.9 days) it takes to complete a full orbital rotation.
var earthAxisRotationSpeed= 9.972;
var earthAxialTilt = 23.44;
var earthOrbitEccentricity = 0.00167 * eccentricityModifier;

// Size of the Moon, scaled larger for the visualisation.
var moonSize = 6; // from -> 0.277 * 21;
// The distance between the Moon and Earth, scaled for the visualisation.
var moonDistanceFromEarth = 40; // from -> 60.3 / 1.5;
// The orbital period of the Moon around the Earth (365 / 12.175 days, the Moon completes a full orbital rotation in roughly 28 days).
var moonOrbitalPeriod = 365.26 / 12.175;

var moonOrbitEccentricity = 0.00549 * eccentricityModifier;
var moonAxialTilt = 1.593;
var moonOrbitalTilt = 5.145;
```

```

// Size of the Moon, scaled smaller for the visualisation.
var sunSize = 20; // from -> 109 / 7;
var sunAxisRotationSpeed = 0.25;

var starMapSize = 500;

// The 'speed' of the animation. Increasing/decreasing this scales the time of the animation through the orbital and axial rotations.
var speed = 1;
```

As you can see, the Sun is much smaller than it should be, as are the distances between the Sun, Earth and Moon. I felt these numbers gave a good representation for this model, but should not be used if the model were intended to be an accurate representation of the system.

Figure 1: The Sun, Earth, Moon system after scaling.

Structure

I have structured my JavaScript code such that there are files for each major component of the model (user interface, Sun-Earth-Moon in one file, Controller, etc) that handles the responsibility of those components, for initiation and updating on each animation loop. For the Sun, Earth and Moon, each is held in their own object, which contains data of their Mesh, a constant reference to their homogeneous coordinates and their update methods.

Most of the parameters for controlling speed, size and the tilts of the planets are contained in a single global variable file, for easy access when I was building and testing my simulation. The setup and main rendering loop are in a single Controller file, and this calls to methods in the other files in the order that things should be initialised and updated. This made it easier to ensure the correct ordering of events.

The Earth and Moon mesh objects have also been added into one single Object3D, earthAndMoon, and then that object has been added to the scene, instead of adding the two individually. This increases the efficiency of the rendering, as they are rendered as one item. While not necessary for such a small simulation, if the application were expanded upon in the future to include many, maybe thousands, of additional planets, this style of

structure would be much easier to maintain. It would also help group planets and their moons together, meaning they could be added and removed with their satellites easily.

Running the program

To run the simulation for demonstration, visit:

`http://users.aber.ac.uk/jee22/cs323/virtual-system/`

hosted on my university file store. For building and testing, I ran the application locally using xampp[2] . You should be able to see the application in full working order in both B59 and C56, best seen with the Firefox browser. No additional downloading is required. There is also a Credits page for any third party textures and code used in the application here: `http://users.aber.ac.uk/jee22/cs323/virtual-system/credits.html`.

Basic Features

The earth orbits the centre of the Sun

The Earth has a clear orbit around the centre of the Sun, although it is an elliptical orbit and so may look a little off centre in the screenshot. If you view the images below, you can see how the orbit is set up around the Sun. This is achieved through pre-computed points of the orbital rotation using Kepler's 2nd Law of planetary motion (more on this in Additional Features section), which generates an elliptical orbit with varying speeds depending on the point of orbit.

(a) View from above. (b) View from the side.

Figure 2: Views of the Earth's Orbit around the centre of the Sun (elliptical), shown by the orbit lines.

Originally, the Earth orbited the Sun through updating the x and z coordinates, each animation frame updating the new position of the Earth by shifting it a certain amount,

using the distance from the Sun, the speed of rotation the current x and z position and using radians to create the next step on the full anti-clockwise orbit.

The moon orbits the centre of the Earth

The Moon orbits the centre of the Earth in much the same way that the Earth orbits the Sun. The points are precomputed, yet take into account the Earth's current position too. The orbit is elliptical, and as in reality, doesn't sit squarely 'centrally circling' the Earth. More information can be found out about this later in the report.

(a) View from above. (b) View from the side.

Figure 3: Views of the Moon's Orbit around the centre of the Earth (elliptical), shown by the orbit lines.

Circular orbits

When I first built the simulation, the orbits were completely circular, but for a challenge I made them Elliptical, you can see Additional Features for more information. When they were circular though, they were done like this:

```
function updateOrbit(objectToUpdate, pivotPosition, orbitDistanceFromPivot
    , orbitAngleThisStep){
    objectToUpdate.position.x = pivotPosition.x + (orbitDistanceFromPivot
        * -Math.cos(orbitAngleThisStep));
    objectToUpdate.position.z = pivotPosition.z + (orbitDistanceFromPivot
        * -Math.sin(orbitAngleThisStep));
}
```

Where objectToUpdate was the Earth or Moon, pivotPosition is the centre of the orbit and point to orbit around, orbitDistanceFromPivot is the distance between the centre of orbit

and the object orbiting it and the orbitAngleThisStep being the angle of rotation for the next position for the object to move, in radians.

If I were to do this in matrix transformations, I would do a similar process to updating the x and z position, but rather than updating them with .x and .z =, I would turn the coordinates into a homogeneous matrix and then multiply a shift on the x matrix and z matrix transformations together, apply that to the homogeneous coordinates, then convert them back to physical coordinates, update the object's vertices with its new position and then update the face and vertex normals with the normal of the transformation matrix.

Doing this is not necessary for the orbital rotations as they are precomputed. I have, however, done this for axial rotations and this can be seen in the subsection for the Sun, Earth and Moon spinning on their own axes.

The Sun, Earth and Moon shown as texture mapped spheres

As shown below, each object has its own texture mapped to the surface. As an addition to this, the Earth and Moon have bump maps which make it look like their surfaces are not completely smooth, as in reality. The Earth additionally has a specular map, which allows the light from the Phong lighting and shading to reflect off of the reflective water of the Earth but not so much the land masses. The textures for the Earth and Moon come from planetpixel[3], while the Sun texture is credited to NASA[4]

(a) Earth texture maps

(b) Sun texture map

(c) Moon texture maps

Figure 4: The objects of the simulation, texture mapped to look like the planets they represent.

The Sun, Earth and Moon each spinning on their own axes

The Sun, Earth and Moon each spin on their own axis by a rotation on the Y axis. In my first version of the application, this was done through updating the object.rotation.y in increments, based on the objects rotation speed, every animation frame.

```
earthMesh.rotation.y += controlValues.earthAxisRotationSpeed * (Math.PI /
    180); // rotate Earth on axis
```

This ensured that each planet would rotate (anti-clockwise) based on the global variables provided (combined with the speed of animation via controlValues) and not affect the position or the axial tilt. Later, I changed the process of updating the rotation to using Matrix

transformations, and this is covered in Additional Features.

While difficult to show in just images, you can see from figure 5 that the Earth spins on it's own axis.

(a) The Earth spinning on its own axis 1.

(b) The Earth spinning on its own axis 2.

(c) The Earth spinning on its own axis 3.

(d) The Earth spinning on its own axis 4.

Figure 5: Collections of images to present the Earth spinning on its own axis.

To see the Moon's axial rotation, we can best view it from above, such as with figure 6, showing the full axial rotation (in this case in sync with the orbital rotation around the Earth too. This is covered in more detail in Additional Features).

(a) The Moon spinning on its own axis.

(b) The Moon spinning on its own axis.

(c) The Moon spinning on its own axis.

(d) The Moon spinning on its own axis.

Figure 6: Collections of images to present the Moon spinning on its own axis.

The Sun also spins on its own axis, as seen in figure 7.

(a) The Sun spinning on its own axis 1

(b) The Sun spinning on its own axis 2

(c) The Sun spinning on its own axis 3

Figure 7: Collections of images to present the Sun spinning on its own axis.

The sun shown as a self-illuminated sphere

To present the Sun as a self-illuminated sphere, I first used an emissive map, which lights the texture applied to the Mesh, and then added some Ambient lighting, for the purpose of showing the Sun as if it had it's own light and also to help the viewer see the backs of the planets too. Additionally, I thought it looked good to have a glowing effect around the Sun. I achieved this using the technique by Lee Stemkoski[5]. I created two glow effects, one larger than the other, to give the impression of the light being more intense as the viewer looked directly at the Sun.

(a) The Sun with just an emissive map for self-illuminating

(b) The Sun with emissive map and ambient lighting.

(c) The Sun with emissive map, ambient lighting and glow effect.

Figure 8: The Sun with the three ways it is presented as self-illuminated.

Earth and Moon lit by a single point light source located at the centre of the Sun

All objects in the scene are lit slightly by a dark AmbientLight for the purpose of the animation. The light source from the Sun is a SpotLight, not a PointLight as requested by the Basic Features. The main reason for this is that Point Lights in three.js do not cast shadows (note: As of writing this, PointLights may now cast shadows, or at least a solution for this is being made. For the sake of computation costs, I have stuck with using a SpotLight).

In order to light the Earth and Moon with the Spot Light, I have the light starting

at the centre of the Sun (0,0,0) and added as part of the SunMesh for rendering efficiency, and its target set as the position of the Earth, tracking it every animation frame to give the illusion of a single light source at the centre of the Sun.

```
function buildSunLight(){
  var ambientLight = new THREE.AmbientLight(0x222222);
  scene.add(ambientLight);

  sunLight = new THREE.SpotLight(0xfff0e6, 1, 0);
  sunLight.castShadow = true;

  sunLight.target = earthMesh;
  sunLight.position.set(sunMesh.position.x, sunMesh.position.y, sunMesh.
      position.z);
  sunMesh.add(sunLight);
}
```

Additional Features

User Interface

To help the user control and view the simulation, I added a number of user interface controls. Using the sidebar interface, a user can change the animation speed, change the camera focus to look at the Sun, Earth or Moon, reset the camera, pause the simulation (pauses the updating of values while still rendering the scene) and turn on or off helpers to visually see the orbital rotations and the axes of the objects. The side bar user interface is made using dat.gui[6], which makes for quick and easy gui by adding controls and parameters for them, including values and functions.

(a) The user interface on the page.

(b) The user interface up close.

```
var params = {
    "speed": 1,
    "cameraFocus": 0,
```

```
    "resetCamera" : resetCamera,
    "pause": false,
    "orbitHelpers": true,
    "axisHelpers": true
};

function setupGUI() {
    var gui = new dat.GUI();
    gui.add( params, "speed" ).min(1).max(10).step(1).name('Change speed')
        .onChange( function( value ) {
        updateControlValues(value);
    } );

    gui.add( params, 'cameraFocus', {Sun: 0, Earth: 1, Moon: 2} ).name('
        Change camera focus').onChange( function(value) {
      if(value == 0){
        controls.target = sunMesh.position;
      } else if(value == 1){
        controls.target = earthMesh.position;
      } else if(value == 2){
        controls.target = moonMesh.position;
      }
    });

    gui.add(params, 'resetCamera').name('Reset Camera');

    gui.add(params, 'pause').name('Pause').onChange( function( value ){
        simulationPaused = value;
    });

    gui.add(params, 'orbitHelpers').onChange( function( value ){
        earthOrbitLine.visible = value;
        moonOrbitLine.visible = value;
    }).name('Orbit helpers');

    gui.add(params, 'axisHelpers').onChange( function( value ){
        earthAxisHelper.visible = value;
        moonAxisHelper.visible = value;
        sunAxisHelper.visible = value;
    }).name('Axis helpers');
}
```

The user can also navigate around the simulation using their mouse or the arrow keys using OrbitControls[7].

Synchronous orbital and axial rotation of the Moon

By using the same speed for the Moon's orbital and axial rotation in the simulation, the Moon can be set so that the same face is always shown to the Earth. When I was updating the Moon's orbit by computing the next step every animation frame, I would use the same speed value for both the rotation update and orbit update. However, since I change to precomputing the elliptical orbit, the speeds no longer had a way to match up.

One option I could take is to continue computing the axial rotation every animation step, but only as often as I update the orbital rotation step, to ensure they stayed in sync. However, doing this, plus with the Moon's rotation being titled, requires a fair amount of computation to try and get the synchronisation correct. In order to save time on coding and avoid a 'wobbly' Moon, I used the three.js function

```
moonMesh.lookAt(earthMesh.position);
```

every animation frame. This keeps the synchronous axial and orbital rotation around the Earth.

(a) Synchronous axial and orbital rotation of the Moon 1.

(b) Synchronous axial and orbital rotation of the Moon 2.

(c) Synchronous axial and orbital rotation of the Moon 3.

Figure 10: You can see the blue axial line, always pointing out from the same face, is always pointing at the Earth.

Lighting of the Earth and Moon in Phong shading & Non-illuminated parts of the Earth and Moon to be shown in shadow

The Materials of the Earth and Moon are made from PhongMaterial, and so Phong lighting and shading is applied through this. Through using Phong shading I was able to better represent how the system looks in reality, with the parts of the Earth and Moon not in the light shown in shadow, and those in the light being brighter.

(a) Light from the SpotLight on the PhongMaterial.

(b) Shadows from the absence of light from SpotLight on the PhongMaterial.

Figure 11: The effect of light on the PhongMaterial causing Phong shading.

In particular, where I have used a specular map for the Earth's ocean, you can see the shiny light reflection. During my testing, I changed the value of the shininess of the Earth to see how it would look, which can be seen in figure 12.

(a) How the specular light is set on my simulation.

(b) With no shininess.

(c) High shininess.

(d) Maximum shininess.

(e) Showing the landmasses do no reflect the light as much.

Figure 12: Testing what happens when the shininess is turned up on the Phong material with a specular map.

As an effect of using a SpotLight and Phong Material, the sides facing away from the Sun are non-illuminated, and so shown in shadow, no matter where the objects are in relation to the Sun (and so the light), the opposite side is in shadow. There is still Ambient Light to display some detail of the shadow sections, but otherwise they are shown to be dark. This happens automatically when using any light that has a direction (Spot, Directional, Point, but not Ambient) and Phong (or Lambert) material.

(a) The side of the Earth facing the Sun is bright, while the side not illuminated is in shadow.

(b) The side of the Moon facing the Sun is bright, while the side not illuminated is in shadow.

Figure 13: The non-illuminated sides of the Earth and Moon are shown in shadow.

Constant tilt of the Earth's axis

With the aid of Axis Helper, we can see that the Earth's axis is constantly tilted by 23.44. No matter where it is on it's orbital rotation, the Earth is always tilted in the same direction, simulating the reality of the earth being 'flung' around the Sun by the orbit, not turned with it.

Figure 14: The Earth is tilted the same direction and angle at each point during its orbit, shown by the green line of the Axis Helper.

This is super simple to achieve. We just set the Earth's rotation on the z axis to be 23.4 once, when we create it, and then it will always be tilted in that direction, no matter where it is or how the axial rotation on the y axis or orbital rotation and Earth position changes.

```
earthMesh.rotation.x = (earthAxialTilt/180)*Math.PI;
earthAxisHelper = new THREE.AxisHelper( earthSize * 2 );
earthAxisHelper.visible = true;
earthMesh.add( earthAxisHelper );
```

Constant tilt of the Moon's orbit

The Moon's orbit is always tilted, no matter where the Earth is, the orbit is tilted following the Earth, and doesn't wobble. To tilt the orbital rotation of the moon, when the pre-computed points of orbit are computed, they are then multiplied by a rotation on the z axis with a transformation matrix. I chose to use three.js applyMatrix4 with my own rotation matrix.

```
var rotationYMatrix4 = function (angle) {
        tempMatrix4.set(
            Math.cos(angle), 0, Math.sin(angle), 0,
```

```
            0,1,0,0,
            -Math.sin(angle), 0, Math.cos(angle), 0,
            0,0,0,1
            );
        return tempMatrix4;
}

var moonOrbitPoints = calculateOrbitalPoints(orbitSteps,
    moonOrbitEccentricity, moonDistanceFromEarth);
for(var i = 0; i < moonOrbitPoints.length; i++){
  moonOrbitPoints[i].applyMatrix4(rotationZMatrix4(-(moonOrbitalTilt * (
      Math.PI / 180))));
}
```

This constant tilt can be seen in Figure 15.

Figure 15: The constant orbital tilt of the Moon.

Elliptical Orbits & Non-uniform orbital velocities: Kepler's 2nd Law of Planetary Motion

The calculating of the elliptical orbits and using Kepler's 2nd Law of Planetary Motion tie together, in that when calculating Kepler's 2nd Law we also get the points for an elliptical orbit. I used the formula provided in the assignment brief [1] in order to achieve this, and my code can be seen here:

```
// Kepler's 2nd Law calculations (for all points along an orbital rotation
    , in n (timeStep) steps).
```

```
function calculateOrbitalPoints(timeStep, eccentricity, semiMajorAxis){
   // 2 * Pi is radians for 360 circle;
  var completeCircle = Math.PI * 2;
  var theta = 0;
  var r;
  var orbitPoints = [];
  // +1 to ensure a complete circle with no gaps.
  while(theta <= completeCircle){
    theta += calculateTheta(timeStep, eccentricity, theta);
    r = calculateR(semiMajorAxis, eccentricity, theta);
    orbitPoints.push(new THREE.Vector3(r * -Math.cos(theta), 0, r * Math.
       sin(theta)));
  }
  return orbitPoints;
}

function calculateTheta(timeStep, eccentricity, theta){
  return (2 / timeStep) * Math.pow((1 + eccentricity * Math.cos(theta)),
     2) / Math.pow((1 - Math.pow(eccentricity), 2), (3 / 2));
}

function calculateR(semiMajorAxis, eccentricity, theta){
  var r = (semiMajorAxis * (1 - Math.pow(eccentricity, 2)) / (1 +
     eccentricity * Math.cos(theta)));
  return r;
}
```

To use the function, the points are calculated by providing a timeStep (for the sake of the simulation and scale, this is set as 1000, and just means how many steps there will be for a complete orbit from starting point to end), the orbit eccentricity and the semi-major axis of the objects rotation, which is the maximum distance of the orbit of the pivot point to the object rotating around it. Once the points have been made, every time the update function is called during the animation loop, the object moves to the next point of orbit, based on the speed of animation.

```
var earthOrbitPoints = calculateOrbitalPoints(orbitSteps,
   earthOrbitEccentricity, earthDistanceFromSun);

updateEarth : function(){
    earthStep+= speed;
    if(earthStep >= earthOrbitPoints.length){
      earthStep = 0;
    }
    var currentPosition = earthOrbitPoints[Math.floor(earthStep)];
    earthMesh.position.set(currentPosition.x, currentPosition.y,
       currentPosition.z);
}
```

You can see the results of the orbital point calculations in figure 16. By changing the eccentricity, we can change the size of the ellipse, and in effect the speed of the object in its orbit as it reaches points closer to the object it is orbiting around.

(a) Earth's elliptical orbit.

(b) Moon's elliptical orbit.

(c) Moon's elliptical orbit scaled larger for testing.

Figure 16: The Earth and Moon's rotations are slightly elliptical.

During testing, I created just the orbital lines to see what the orbit path was like, and changing the number of time steps changed the speed of the orbit by the amount of steps a planet had to take to complete an orbit. Once I found a number I thought was enough timeStep points, I set it to change the speed in animation based on how many steps through the orbit are jumped each frame. We can also see that at certain points, the Earth/Moon moves faster where the points are closer and the object is closer to the central point of the orbit, simulating Kepler's 2nd Law of Planetary Motion. This is, however, difficult to show

in screenshots.

Eclipse Shadows

Through the use of a SpotLight and its shadow camera and turning the ShadowMap of the renderer on, we get the effect of the objects casting shadows upon each other and causing an Eclipse once in a while.

(a) The Moon casting its shadow on the Earth's surface.

(b) The Earth casting its shadow on the Moon's surface.

Figure 17: The Earth and Moon casting their shadows on each other, which can result in an eclipse of the Sun.

To get this effect is very simple, as just a couple of flags need to be set for the objects that need to be affected by this. There must be the correct materials (Lambert and Phong) and lighting (Spot, Directional - Although as of writing this, I believe PointLight either works now or a solution is being worked on) in order for this effect to work.

```
renderer.shadowMap.enabled = true;
sunLight.castShadow = true;

earthMesh.castShadow = true;
earthMesh.receiveShadow = true;

moonMesh.castShadow = true;
moonMesh.receiveShadow = true;
```

In order to test that the eclipse shadows worked, I used the CameraHelper class of three.js with the shadow camera of the simulations SpotLight.

```
sunLight = new THREE.SpotLight(0xfff0e6, 1, 0);
sunLight.castShadow = true;
sunLight.shadowCameraNear = 20;
sunLight.shadowCameraFar = 190;
sunLight.shadowCameraFov = 45;
```

```
// awesome for debugging - Shows the Shadow Camera lines
//scene.add(new THREE.CameraHelper( sunLight.shadow.camera ));
sunLight.target = earthMesh;
sunLight.position.set(0, 0, 0);
sunMesh.add(sunLight);
```

Figure 18: The Sun light Shadow Camera, showing where shadow casting will be computed.

You can see the ShadowCamera of the SpotLight in figure 18, as displayed by the ShadowCameraHelper. This helper shows us what the catchment area of the cast shadows will be. Computing where shadows should fall is expensive, and so I've made the shadow camera area small, in order to reduce the cost of computation.

When I first enabled the casting and receiving of shadows I encountered issues with double shadows, where a shadow would be shown on the correct side of the object shadowed but also on the reverse side. I struggled with the problem for a long time, trying to debug where I had gone wrong within my code to cause this effect. I wasn't sure if I had slightly transparent objects, if there was an issue recalculating the face and vertex normals or if I had set up my shadow camera incorrectly. In the end, it turned out that it was something I couldn't do anything about, as it was a WebGL issue[8][9] within three.js that was only fixed a couple of weeks after the version of three.js I was using was downloaded for my application. Updating to the latest version of three.js fixed this issue!

Other features of my choice

As an addition to the previously mentioned features, I also wanted to implement something we'd discussed in lectures. The most obvious choice for this was matrix transformations. I implemented a number of different transformations for shifts and rotations, and started my project with using them to compute orbit. After I moved to precomputing the orbital

rotations for Elliptical orbits and using Kepler's 2nd Law, the matrix transformations are now just used for the axial rotations of each object and the initial setup for the tilt of the Moon's orbit.

I came across a number of problems when I first started using matrices, and while I don't have screenshots to display some of the strange graphical errors encountered, I have described the most notable below:

- The Earth gradually shrank into a cylinder - Caused by an incorrect rotation matrix that wasn't applying the 'w' coordinate properly after a shift and rotation.
- The Earth and Moon not being captured in the Sun's gravitational pull - Caused by the way my matrices were multiplied being completely incorrect, as the way that coordinates are stored in three.js are not the same as I was expecting when I first wrote my matrix multiplication method. This also caused some issues where the Earth would become a mess of pixels.
- The lighting of the Earth and Moon staying in a constant position, even as the Earth and Moon rotated on their axis and moved around on their orbital rotation, the direction of the light would stay where it was when the simulation first started. - Caused by not correctly re-calculating the face and vertex normals after a matrix transformation.
- The vertices correctly updating after a matrix transformation but no actual difference being rendered - Caused by too much computation on each render step and creating too many new Vector3 objects. This got the system stuck in updating the data of the objects, but the renderer still waiting and not updating the objects after transformation.

There were many other smaller issues along the way, and multiple different ways of attempting to apply my own matrices to the objects, but in the end the method that works the best for me is to:

- After initialization, get the physical coordinates of the vertices of the objects and hold them in memory
- Turn the coordinates into homogeneous coordinates
- During every run step, multiple the homogeneous coordinates with the next transformation matrix for this step
- Turn the homogeneous coordinates into physical coordinates, keeping the updated homogeneous coordinates in memory
- Update the object with the new vertices
- Update the face and vertex normals with the normal of the transformation matrix

- Set flags on the object that it has been updated and needs to be rendered

In code using the Earth as an example, this looks somewhat like this (this is extracted from multiple methods in order to be demonstrated here):

```
// initiate - Get homogeneous coordinates in memory
computableEarthVertices = convertPhysicalToHomogeneous(earthMesh.geometry.
    vertices);

function convertPhysicalToHomogeneous(coordinateArray){
    var result = rearrangeVertices(coordinateArray);
    result[3] = [];
    for(var i=0; i < result[0].length; i++){
        result[3][i] = 1;
    }
    return result;
}

function rearrangeVertices(coordinateArray){
    var result = [];
    for(var x = 0; x < 3; x++){
        result[x] = [];
        for(var y = 0; y < coordinateArray.length; y++){
            result[x][y] = 0;
        }
    }
    for(var i=0; i < coordinateArray.length; i++){
        result[0][i] = coordinateArray[i].x;
    }
    for(var i=0; i < coordinateArray.length; i++){
        result[1][i] = coordinateArray[i].y;
    }
    for(var i=0; i < coordinateArray.length; i++){
        result[2][i] = coordinateArray[i].z;
    }
    return result;
}
```

```
// update the computable vertices (homogeneous) and apply it to the object
// Once per animation frame
computableEarthVertices = updateYRotation(controlValues.
    earthAxisRotationSpeed * (Math.PI / 180), earthMesh.geometry,
    computableEarthVertices);

function updateYRotation(rotation, geometry, computableVertices){
    computableVertices = multiplyMatrices(rotationYTransformation(rotation
        ), computableVertices);
    applyHomogeneousToPhysical(computableVertices, geometry.vertices);
    var normalMatrix = tempMatrix3.getNormalMatrix(rotationYMatrix4(
        rotation));
```

```
    updateFaceAndVertexNormals(normalMatrix, geometry);
    geometry.verticesNeedUpdate = true;
    geometry.normalsNeedUpdate = true;

    return computableVertices;
}

function multiplyMatrices(matrixOne, matrixTwo){
    if(matrixOne[0].length != matrixTwo.length){
        alert("MatrixOne columns do not match MatrixTwo rows. Cannot compute!");
    } else {
        var result = [];
    for(var x = 0; x < matrixOne.length; x++){
        result[x] = [];
        for(var y = 0; y < matrixTwo[0].length; y++){
            result[x][y] = 0;
        }
    }

    for(var r=0; r < matrixTwo.length; r++){
        for(var c=0; c < matrixTwo[0].length; c++){
            for(var i=0; i < matrixOne[0].length; i++){
                result[r][c] += (matrixOne[r][i] * matrixTwo[i][c]);
            }
        }
    }

    return result;
    }
}

var rotationYTransformation = function (angle) {
    return [
        [Math.cos(angle), 0, Math.sin(angle), 0],
        [0,1,0,0],
        [-Math.sin(angle), 0, Math.cos(angle), 0],
        [0,0,0,1]
    ];
}

var rotationYMatrix4 = function (angle) {
        tempMatrix4.set(
            Math.cos(angle), 0, Math.sin(angle), 0,
            0,1,0,0,
            -Math.sin(angle), 0, Math.cos(angle), 0,
            0,0,0,1
            );

```

```
        return tempMatrix4;
}

function applyHomogeneousToPhysical(homogeneous, physical){
    for(var x = 0; x < homogeneous[0].length; x++){
        physical[x].setX(homogeneous[0][x] / homogeneous[3][x]);
        physical[x].setY(homogeneous[1][x] / homogeneous[3][x]);
        physical[x].setZ(homogeneous[2][x] / homogeneous[3][x]);
    }
}

function updateFaceAndVertexNormals(normalMatrix, geometry){
    for ( var i = 0, il = geometry.faces.length; i < il; i ++ ) {
        var face = geometry.faces[ i ];
        face.normal.applyMatrix3( normalMatrix ).normalize();
        for ( var j = 0, jl = face.vertexNormals.length; j < jl; j ++ ) {
            face.vertexNormals[ j ].applyMatrix3( normalMatrix ).normalize
                ();
        }
    }

    if ( geometry.boundingBox !== null ) {
        geometry.computeBoundingBox();
    }
    if ( geometry.boundingSphere !== null ) {
        geometry.computeBoundingSphere();
    }
}
```

While this may not be the most efficient way, it was how I found I was able to extract the data from the WebGL/three.js objects, modify it and then put it back in using my own code, to show that this is something I am capable of doing and have greater control over the objects.

I am aware that there are better methods of applying matrices provided by three.js, and that for just rotation matrices I could also remove the step of converting to and from homogeneous coordinates. However, for the sake of the simulation, the potential for expansion in the future and my own challenge, I have kept the process this way.

As a final additional step, I added a star map to surround the simulation, to help the user navigate around the scene easier and give a tiny bit more visual interest and realism. The inspiration for this came from 'learningthreejs'[10]. The star map texture is from Paul Bourke[11].

Figure 19: A portion of the star texture map.

```
// Simple sphere that surrounds the simulation, to give the appearance
    that all takes place within the universe and the camera can move around
     it.
function buildStarMapMesh(){
  var starGeometry  = new THREE.SphereGeometry(starMapSize, 32, 32);
  var starTexture = textures.starMap;
  starTexture.wrapS = starTexture.wrapT = THREE.RepeatWrapping;
  var starMaterial = new THREE.MeshBasicMaterial( {map: starTexture } );
  starMaterial.side = THREE.BackSide;
  var starMesh = new THREE.Mesh(starGeometry, starMaterial);
  return starMesh;
}
```

Self Assessment

Task	documented	implemented		Extension	documented	implemented
1	✓	✓		a)	✓	✓
2	✓	✓		b)	✓	✓
3	✓	✓		c)	✓	✓
4	✓	✓		d)	✓	✓
5	✓	✓		e)	✓	✓
6	✓	✓		f)	✓	✓
7	✓	✓		g)	✓	✓
				h)	✓	✓
				i)	✓	✓
				j)	✓	✓
Animation can be viewed in B57, C56 without additional software						✓
All reference sources are acknowledged						✓

Your overall self-assessment out of 50: **47/50**

From implementing all of the basic features as requested and then all of the additional features, I feel my grade for those sections should be high. I have a feeling that my report may bring my mark down slightly, due to my style of writing and my tests being mainly visual rather than data tables, and that it is very long in length. Through this, I would award myself the mark shown above.

References

[1] H. Holstein and Y. Liu, "A virtual Sun-Earth-Moon system", CS32310 Advanced Computer Graphics Assignment Semester 1 2015, October 14 2015

[2] ApacheFriends.org, *'XAMPP Installers and Downloads for Apache Friends'* , [online], Available: https://www.apachefriends.org/index.html [Accessed 11/06/2015]

[3] PlanetPixelEmporium.com, *'Planet Texture Map Collection'*, [online], Available: http://planetpixelemporium.com/planets.html [Accessed 11/06/2015]

[4] NASA.gov, *'NASA'*, [online], Available:

https://www.nasa.gov/sites/default/files/images/700328main_20121014_003615_flat.jpg [Accessed 11/06/2015]

[5] Stemkoski.Blogspot.co.uk, *'Computational Contemplations: Using Shaders and Selective Glow Effects in Three.js'*, [online], Available: http://stemkoski.blogspot.co.uk/2013/03/using-shaders-and-selective-glow.html [Accessed 11/06/2015]

[6] , GitHub.com *'three.js/dat.gui.min.js at master . mrdoob/three.js'*, [online], Available: https://github.com/mrdoob/three.js/blob/master/examples/js/libs/dat.gui.min.js [Accessed 11/06/2015]

[7] GitHub.com *'three.js/OrbitControls.js at master . mrdoob/three.js'*, [online], Available: https://github.com/mrdoob/three.js/blob/master/examples/js/controls/OrbitControls.js [Accessed 11/06/2015]

[8] StackOverflow.com, *'webgl - THREE.JS Shadow on opposite side of light - Stack Overflow'*, [online], Available: http://stackoverflow.com/a/12602143 [Accessed 11/07/2015]

[9] StackOverflow.com, *'Three.js DoubleSided material doesn't cast shadow on both sides of planar parametric geometry'*, [online], Available: http://stackoverflow.com/a/20473174 [Accessed 11/07/2015]

[10] LearningThreeJS.com, *'How To Make The Earth In WebGL? - Learning Three.js'*, [online], Available: http://learningthreejs.com/blog/2013/09/16/how-to-make-the-earth-in-webgl/ [Accessed 11/07/2015]

[11] PaulBourke.net, *'Representing star fields'*, [online], Available: http://paulbourke.net/miscellaneous/starfield/ [Accessed 11/07/2015]

[12] Acknowledgement for code colouring in this report using lstlisting LaTeX:

TeX.StackExchange.com, *'javascript - language option supported in listings - TeX - LaTeX Stack Exchange'*, [online],

Available: http://tex.stackexchange.com/questions/89574/language-option-supported-in-listings [Accessed: 11/06/2015]