COMP27112
Introduction to Visual Computing
Coursework Assignment 2
Sun, Planet and Moon
Tim Morris

Introduction

The aim of this exercise is to render a simple solar system with one moon orbiting one planet orbiting one sun. As with the previous exercise, you'll need a web browser to display the output and a simple text editor[1] to create the html file you'll need.

This piece of work contributes half of the coursework assessment. We suggest that you should not spend more than 15 hours in total completing Assignments 1 and 2.

Getting Started

Start off with the same code as last time for creating the basic webpage. Under "`// Your Javascript will go here.`" you'll add any global variables and calls to two functions, `init()` and `animate()`.

Creating the Scene (`init()`;)

Everything is initialised in this function.

As before, add a `scene`, `camera` and `renderer` and add the `renderer` to the `document` (don't forget to declare the `scene`, `camera` and `renderer` as global variables). You won't need a vertex shader or fragment shader this time.

Put the far plane of the camera at 10000. Locate the camera at (0, 30, 500) and add it to the `scene`.

You'll need to create nine variables for the sun's, earth's and moon's geometries, materials and meshes. These are global variables.

You can create the sun object using the following code:

[1] You should ensure that your text editor saves your html files as plain text. Some editors (e.g. TextEdit on the Mac add all kinds of stuff to what you think is a plain html file, meaning that you just see the text of your file when you open it in your browser. Don't forget to give your files the correct extension.
You could also use a suitable IDE.

```
sunGeometry = new THREE.SphereGeometry(109, 400, 200);
sunMaterial = new THREE.MeshStandardMaterial(
    {
      emissive: 0xffd700,
//      emissiveMap: texture,
      emissiveIntensity: 1,
      wireframe: false
    }
);
sunMesh = new THREE.Mesh(sunGeometry, sunMaterial);
scene.add(sunMesh);
```

This creates a spherical object (what do the arguments mean?) of a particular colour. Where is it located? The `emissiveMap` is something you may use later.

The sun must be a source of light. The `PointLight()` constructor can achieve this. Create a point light source of white light at the origin and add it to the scene. You might also add a diffuse light to give some background illumination, use `AmbientLight()`. Obviously this is physically unrealistic but it makes the objects more visible.

You can create the earth and moon using similar code with the following differences:

- The earth sphere geometry arguments are 25, 50, 50
- The moon sphere geometry arguments are 5, 40, 20
- Both the earth and moon materials are `MeshPhongMaterial`, they don't have an `emissive` argument, but do have a `color`. You can experiment with `color` values.

The `texture` argument might be used later. So the `earthMaterial` can be created using something like:

```
earthMaterial = new THREE.MeshPhongMaterial(
    {
//      map: texture,
      color: x0000ff
    }
```

The earth and moon will be grouped together into one object before being added to the `scene`. To do this we need a global variable to store the earth-moon system, we need to add the earth to it, by default it will go to the origin of this system. Then we need to add the moon to the system and set its position relative to the earth.

Three.js provides a group object for storing collections:

```
earthSystem = new THREE.Group();
```

Then the earth (and moon) can be added to this in a manner that was similar to the way we added the sun to the scene:

```
earthSystem.Add(earthMesh);
```

Don't forget to set the position of the moon within the earth-moon system, using a function something like:

```
moonMesh.position.set(orbitRadius, 0, 0);
```

A suitable value for `orbitRadius` is in the range 40 – 60.

The earth's orbit could be approximated as a circle, and the location of the earth on it could be computed as the following pseudocode:

```
earth.position.x = earthOrbitRadius * sin(2πwt);
earth.position.y = earthOrbitRadius * cos(2πwt);
```

`w` is the earth's angular velocity and `t` is some measurement of time.

It is slightly more realistic to make the orbit an ellipse. To make this more efficient we pre-compute the co-ordinates of the earth's orbit. Create a global variable with a suitable name. The points can be computed by calling the function `EllipseCurve`. This has arguments to define:

- The x co-ordinate of the centre point of the ellipse
- The y co-ordinate of the centre point of the ellipse
- The radius of the ellipse in the x direction
- The radius of the ellipse in the y direction
- The start angle for drawing the ellipse (in this case 0 radians)
- The end angle for drawing the ellipse (in this case 2π radians)
- Plus other arguments that can take default values.

You may choose to draw the orbit, in which case you will have to

- Transfer points from the orbit into a line buffer
- Create a geometry (using `BufferGeometry`) from these points
- Create a material (using `LineBasicMaterial`) and setting a suitable colour
- Create a line object using the geometry and material
- Rotate the line so it lies in the XZ plane instead of the default XY plane
- Add it to the scene

Animation (`Animate();`)

The basic animation function will contain the two lines to render the scene and request an animation frame, as in the previous exercise. If you've followed the instructions so far and now implement this, you'll see a static scene with the sun, earth and moon in fixed positions and the earth orbit (if you chose to draw it). The earth and moon should be solid coloured spheres. The next step is to add movement to the objects. The following code should be added to `Animate()` before the two lines you've just written.

The sun's movement is simple. It doesn't move. You might want to make it rotate if you add a texture to it, which will be done later.

The earth-moon system's position could be driven by using a counter that is incremented for each frame of the animation. But we'll use the time instead. A time can be obtained by calling `performance.now()`. This gives the time in milliseconds since the browser window was opened. This can be converted into a value in the range [0, 1) which will be used as an index into the values of the `EllipseCurve` you defined earlier. In pseudocode:

```
time = 0.00001 * performance.now();
t = (time mod 1)
```

We can get the earth-moon position by reading a point from the `EllipseCurve` object (assume it's called `curve`):

```
point = curve.getPoint(t)
```

Then the `earthSystem's` x and **z** positions can be set to `point.x` and `point.y` respectively. Changing the value of the multiplier (0.00001) will change the earth's orbital speed.

The moon's position is set according to

```
moon.x = orbitRadius*cos(time*speed)
moon.z = orbitRadius*sin(time*speed)
```

Time was derived above. Speed is the orbital speed of the moon – you choose a sensible value.

Optional Enhancements

Some optional enhancements follow.

Changing the viewpoint

It is possible to change the observer's viewpoint by adding the following controls.

Insert the following line after the other `import` statement.

```
import { OrbitControls } from
"https://web.cs.manchester.ac.uk/three/three.js-
master/examples/jsm/controls/OrbitControls.js";
```

Add a global variable with a suitable name, possibly `controls`.

Add the following two lines to the `init()` function:

```
controls = new OrbitControls(camera, renderer.domElement);
controls.autoRotate = true;
```

These add a controller to the scene and allow you to move the viewpoint by clicking and dragging.

Texturing

The sun, earth and moon can be textured using textures from
https://www.solarsystemscope.com/textures/download/2k_sun.jpg
https://upload.wikimedia.org/wikipedia/commons/a/ac/Earthmap1000x500.jpg
https://svs.gsfc.nasa.gov/vis/a000000/a004700/a004720/lroc_color_poles_1k.jpg

To read these you'll have to create one texture loader

```
const loader = new THREE.TextureLoader();
```

Textures can be loaded using this

```
var texture = loader.load('filename'); OR
var texture = loader.load('URL');
```

And added to the material of the object you're creating, by uncommenting the line in the example above where you created the `sun` object.

The earth and moon textures can be added similarly, except you'll add the line

```
map: texture,
```

to the material constructor. You'll also need to delete the `color` property.

The problem you may encounter when attempting to run the code is that the resource fails to load, and you have an error message such as

> Access to image at <source> from origin 'null' has been blocked by CORS policy

This is a security feature of most modern browsers. You can set up a server on your host to overcome this problem. Instructions are widely available on the web, specifically here https://threejs.org/docs/index.html#manual/en/introduction/How-to-run-things-locally
If you're using Chrome you can alternatively install an extension that allows CORS cross origin loading (https://chrome.google.com/webstore/detail/allow-cors-access-control/lhobafahddgcelffkeicbaginigeejlf?hl=en). Or in Safari you can explicitly turn off the CORS checking.

Rotations

You can make the sun, Earth and Moon rotate on their axes, much like the cube rotated in the previous exercise. Make sure you choose an appropriate length of the “day”.

Clouds

You could add clouds to the Earth. Find a cloud image (e.g. https://i.stack.imgur.com/B3c7G.jpg) and add it as a transparent texture to a sphere mesh that is slightly larger than the Earth. Make it rotate slightly slower than the Earth.

Background

You can also create a starry background for the scene. Make a very large sphere mesh – to make sure it’s big enough to contain everything. Add a texture image of the Milky Way: https://cdn.eso.org/images/screen/eso0932a.jpg

Make the sphere visible from inside as well as outside by setting the `side` member of the `material` to `THREE.DoubleSide`.

Submission

Once you have a working solution, capture a short video of your solution, no more than 15 seconds long. It must demonstrate all the properties of your solar system, and ***not so quickly that the marker can't see them clearly*** (you'll be penalised for videos that have so much zooming or camera movement that it's impossible to see the earth or moon rotating). ZIP this and your html and submit the file to the Lab 2 area in Blackboard.

Submissions that are not in the ZIP format will not be marked.

Marking Scheme

We will endeavour to mark your work in face-to-face sessions in the scheduled labs.

You will receive marks for:

- The objects being in appropriate locations and moving appropriately.
- Sensible illumination.
- Enhancements