Simulation of the Perihelion Advance of Mercury: Virtual Reality Solar System

King's College London

6CCP3131

Author: Wong Siu Tung
Supervisor: Prof . Eugene Lim

April 24, 2023

Abstract

The main objective of this project is to create a simulation of the Solar System in the game engine Unity, with an added emphasis on the accuracy of Mercury's orbit. The main code of the project, called TBOrbitBase1 permutes through a list of GameObjects, affecting their velocity and gravity, allowing for mutual interaction. Applying the Schwarzschild metric and the geodesic equations, the effective potential for general relativity can be derived; when taking the derivative of the potential with respect to the orbiting radius, the force between planets can be derived. The advance of Mercury's perihelion can be shown by rearranging the general relativistic potential into the orbital radius as a function of the orbital angle. An additional term of $2\pi\alpha$ is found, which causes the perihelion shift. Applying this finding to TBOrbitBase1, where the gravity term now contains the orbital radius as a function of the orbital angle, the simulation would produce universal gravitation more akin to reality. Further implementation of virtual reality packages in the Unity package management, such as Open XR and XR interaction, would allow for the Oculus Quest 2 to perform a Quest Link with Unity, and XR interaction would allow for the three-dimensional movement of the player.

Contents

Chapter 1

Introduction

1.1 Purpose of the Project

In 1679, Robert Hooke wrote to Isaac Newton about his hypothesis concerning orbital motion, which is partly dependent on the inverse-square force. In 1684, Hooke and Newton had proven the inverse-square law governing planetary motion. Unfortunately, through disagreements, Newton published his book The Mathematical Principles of Natural Philosophy, which hypothesised the inverse-square law of universal gravitation without references to Robert Hooke. The original formula was (1.1), where the proportionality sign was to take the place of the gravitational constant (G), which was first experimentally measured in 1798 by Henry Cavendish [1,2].

$$Force\ of\ Gravity\ \propto \frac{mass\ of\ object\ 1 \times mass\ of\ object\ 2}{distance\ from\ centres^2} \tag{1.1}$$

Newton's law of universal gravitation continues to be used today and is an excellent approximation of the effects of gravity for most applications. However, in some instances, the limitations of Newton's law can be pretty apparent. One such scenario relates to the perihelion shift of the orbits of planets, especially Mercury [3].

The purpose of this project aims to simulate our Solar System using unity and Newtonian mechanics, with the specific goal of implementing general relativistic effects onto the Sun, treating it as a massive, non-rotating, spherically symmetric and uncharged mass and simulating how it would affect the orbits of nearby planets, with an emphasis on Mercury, along with the explanation and physical significance of equations used to describe the orbital paths of the planets and their derivations.

The main focus is on Mercury, due to its distance from the Sun; being the closes of all planets in the Solar System, it would result in the most noticeable perihelion shift; note that while other planets would also experience a perihelion shift, the correction from general relativity would be a factor of 5 times smaller

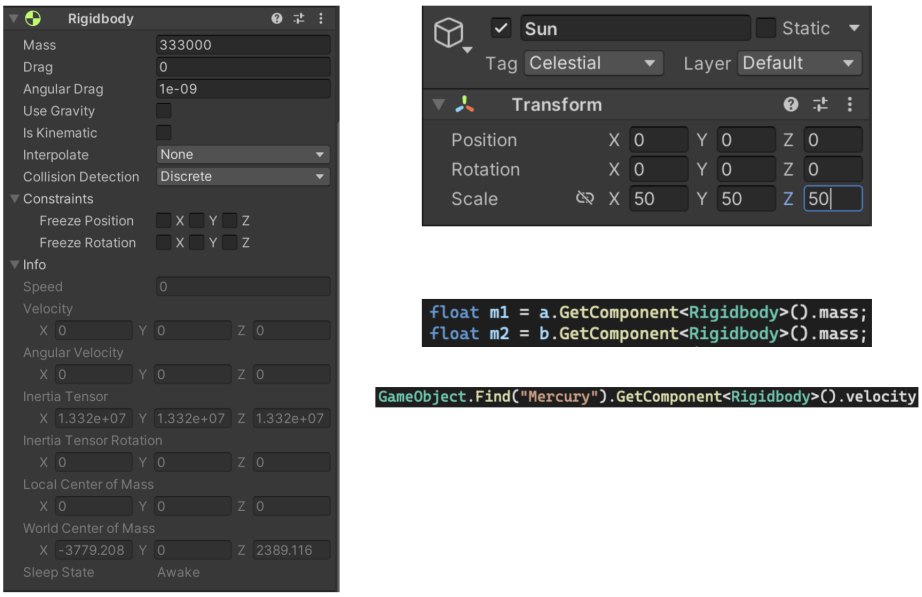

Figure 1.1: The RigidBody interface, found in Get Component for each of the GameObject, C♯ scripts can call upon the mass, for example the first two lines of code. In the Transform interface, the position can be called upon, in the form of the third line of code.

for Venus and a factor of 11 times smaller for Earth [4]. Thus even in the simulation, where the time-scale is set to 52 seconds in unity equals one year on Earth, the perihelion shift would still be imperceptible. The virtual reality aspect of this project is to provide a more engaging and enjoyable learning experience for the user.

1.2 Prerequisites in Coding

Unity[5] contains prebuilt components and game objects, which the developer can call upon. For example, the Rigidbody component would allow GameObjects in Unity to experience downward gravity (The simulation has this turned off) and collisions with incoming GameObjects. With each GameObject containing the Rigidbody component; the developer may choose the objects' mass, position, and scale (The scale only affects the visuals and collision area).

Unity engine's scripting application programming interface (API) is in C♯; in this object-oriented programming language, all perceivable values and names can be called upon in the C♯ script. (See Figure 1.2)

In addition to calling object names, another important command would be calling and appending the velocity value in Rigidbody [6], see Figure 1.1.

```
GameObject.Find(“ Name ”);

//which can be paired with…
GameObject.Find(“ Name ”).Vector3. “Something”

//Where “Something” can be replaced with “Distance(a, b)”
GameObject.Find(“ Name ”).Vector3.Distance(a, b)

//Distance(a, b) is the magnitude between position a and b.

//Where a and b can be found using the…
GameObject.Find(“ Name ”).transform

//A complete line of code, used to calculate the distance between GameObject Venus and another GameObject b would resemble…
float r = Vector3.Distance(GameObject.Find("Venus").transform.position, b.transform.position);
```

Figure 1.2: A representation of coding in C♯. Understanding how to call upon any arbitrary object, along with appending and obtaining values from that object using Vector3.

1.3 Prerequisites in Maths

Vectors can be represented as derivatives of some position; in a cartesian coordinate system with vectors.

$$\vec{R}(x,y), \vec{R}(x+h,y)$$

$$\rightarrow \frac{\partial \vec{R}(x,y)}{\partial x} = \lim_{h\to 0} \frac{\vec{R}(x+h,y) - \vec{R}(x,y)}{h}$$

$$\Rightarrow \frac{\partial \vec{R}(x,y)}{\partial x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \hat{e}_x \ , \frac{\partial \vec{R}(x,y)}{\partial y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \hat{e}_y \tag{1.2}$$

The position vector can be removed for an intrinsic geometry, and the basis vectors can be defined as differentials.

1.3.1 Newtonian Gravity:

The total interaction energy between the Sun and the planets in Newtonian gravity can be derived by first introducing the distance between two objects $\vec{R}(r,\phi)$ and an initial velocity $d\vec{R}/dt$. The unit vectors in polar coordinates are represented as differentials of $\vec{R}$ with respect to r and ϕ.

$$\rightarrow \frac{\partial \vec{R}}{\partial r} \equiv \vec{e_r} \ , \frac{\partial \vec{R}}{\partial \phi} \equiv \vec{e_\phi}$$

Then, applying the multivariable chain rule, the velocity is rewritten in terms

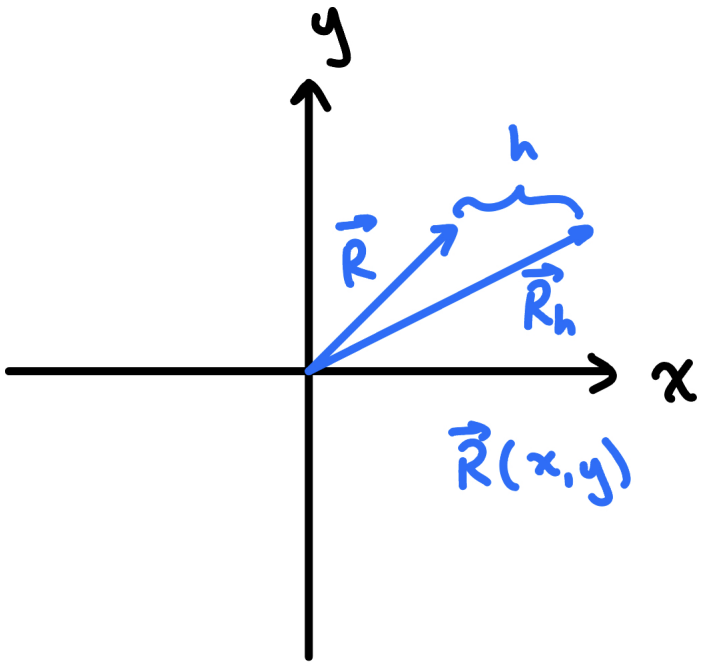

Figure 1.3: Cartesian coordinate system with unit vectors $\hat{e_x}$ and $\hat{e_y}$

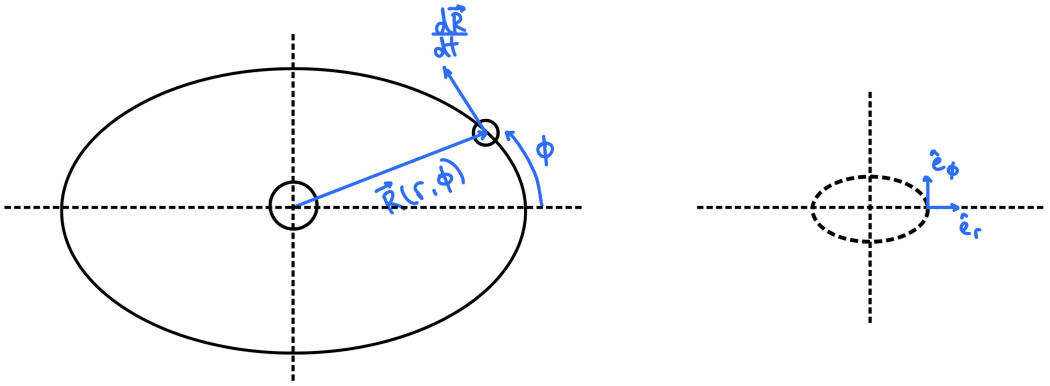

Figure 1.4: An elliptical orbit of a planet around the Sun. With basis vectors in terms of r and ϕ

of the unit vectors.

$$\frac{d\vec{R}(r,\phi)}{dt} = \dot{\vec{r}}\vec{e}_r + \dot{\phi}\vec{e}_\phi$$

$$\Rightarrow v^2 = \left\| \frac{d\vec{R}}{dt} \right\|^2 = (\dot{\vec{r}}\vec{e}_r + \dot{\phi}\vec{e}_\phi)\cdot(\dot{\vec{r}}\vec{e}_r + \dot{\phi}\vec{e}_\phi) = \dot{r}^2 + \dot{\phi}^2 r^2$$

$\dot{\phi}$ can now be represented in terms of known quantities, such as momentum, mass and radius.

$$\vec{L} = \vec{R}\times\vec{p} = \vec{R}\times m\frac{d\vec{R}}{dt}$$

$$\Rightarrow \frac{\vec{L}}{mr^2} = \dot{\phi} \tag{1.3}$$

The total energy can now be written as a function of r, which returns the Newtonian potential.

$$E = \frac{1}{2}mv^2 + mV(r)$$

$$\rightarrow E = \frac{1}{2}m(\dot{\vec{r}} + r^2\dot{\phi}^2) + mV(r);\ V(r) = -\frac{GM}{r} \tag{1.4}$$

$$\rightarrow E = \left[\frac{1}{2}m\left(\frac{dr}{dt}\right)^2 + \frac{L^2}{2mr^2}\right]_{kinetic} - \left[\frac{GMm}{r}\right]_{potential} \tag{1.5}$$

$$\Rightarrow E = \left[\frac{1}{2}m\left(\frac{dr}{dt}\right)^2\right]_{Effective\ K.E.} - \left[\frac{GMm}{r} - \frac{L^2}{2mr^2}\right]_{Effective;P.E.} \tag{1.6}$$

The Newtonian orbits can now be physically shown in terms of potential curves, where each respective curve has its angular momentum. For simplicity, let $G = M = m = 1$; and plot a series of curves. (shown in Figure 1.5)

Taking $L = 0.5$, the total energy is constant, and the kinetic and potential terms vary as the planet orbit. The planet would orbit either elliptically or approximately circularly for each specific total energy, as shown in Figure 1.6.

The foundational Newtonian circular orbits can be derived from section 1.3. The velocity of a circular orbit can be calculated from Figure 1.6; taking $\dot{r} = 0$, and differentiating the total energy with respect to r, thus, returning the force and rearranging for the orbital velocity.

$$E = \frac{L^2}{2mr^2} - \frac{GMm}{r}$$

$$F = -\frac{dE}{dr} = -\frac{d}{dr}\left(\frac{L^2}{2mr^2} - \frac{GMm}{r}\right)$$

$$\rightarrow F \equiv \frac{mv^2}{r} = \frac{GMm}{r^2} \Rightarrow v = \sqrt{\frac{GM}{r}}$$

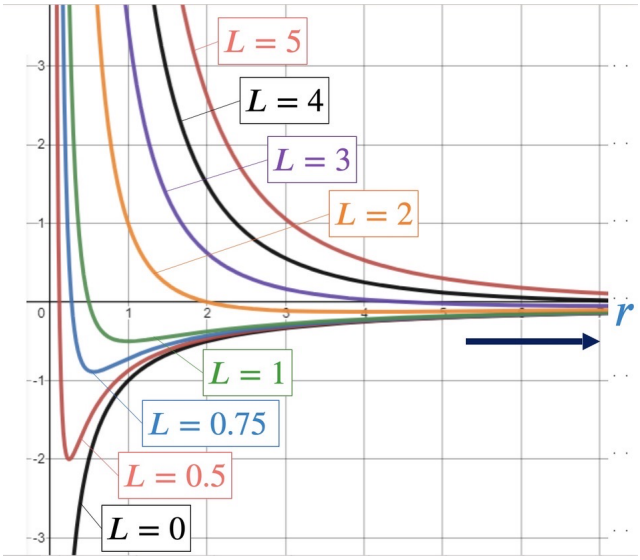

Figure 1.5: A plot of $V(r)$ with different angular momentum and setting the rest of the values as one

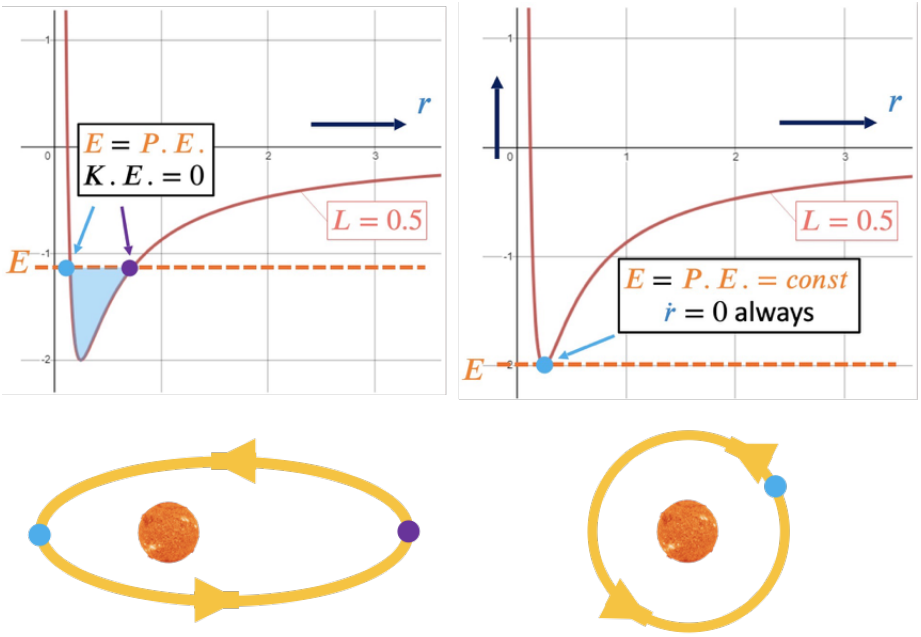

Figure 1.6: For $L = 0.5$, a total energy of approximately -1 would have the planets orbiting elliptically due the the potential curve, while a total energy of -2 would result in a relatively circular orbit.

1.3.2 Schwarzschild Metric and Geodesics:

The Schwarzschild metric is the metric for a massive spherically symmetric, non-rotating and uncharged mass. The Schwarzschild metric can represent the curvature of space-time around the Sun, where r_s is the Schwarzschild mass. For a more detailed explanation and derivation of the Schwarzschild metric, please refer to [7].

$$g_{\mu\nu} = \begin{bmatrix} \left(1 - \frac{r_s}{r}\right) & 0 & 0 & 0 \\ 0 & -\left(\frac{r}{r-r_s}\right) & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2 sin^2\theta \end{bmatrix}, r_s = \frac{2GM}{c^2}$$

The geodesic in curved space-time is the straightest possible path with zero tangential acceleration when travelling along it at a constant speed. It can be defined as parallel transport along the direction of travel. For a more in-depth understanding of geodesics, refer to [8].

$$\nabla_{\vec{V}} \vec{V} = 0$$

$$\rightarrow \nabla_{\vec{V}} \vec{V} = \left[\frac{d^2 x^\mu}{d\lambda^2} + \Gamma^\mu_{\mu\sigma} \frac{dx^\nu}{d\lambda} \frac{dx^\sigma}{d\lambda} \right] \hat{e}_\mu$$

$$\Rightarrow \frac{d^2 x^\mu}{d\lambda^2} + \Gamma^\mu_{\mu\sigma} \frac{dx^\nu}{d\lambda} \frac{dx^\sigma}{d\lambda} = 0 \tag{1.7}$$

$$\left\| \frac{d}{d\lambda} \right\|^2 = \frac{dx^\mu}{d\lambda} \frac{dx^\nu}{d\lambda} g_{\mu\nu} > 0 \tag{1.8}$$

The Schwarzschild metric and the geodesic equation of a time-like particle can be used to derive the general relativity representation of the total potential of a planet next to a massive body.

Chapter 2

Unity setup, Visual Studio and Virtual Reality

2.1 Unity Interface and Setup

The unity editor is split up into several sections; the "Hierarchy" shows all "GameObject" in the "scene", the "Project" contains all the required scripts and downloaded materials, the "Inspector" shows the specific settings of the selected "GameObject", along with added components such as "Rigidbody". Any GameObject can have child objects, where their position (0,0,0) would be the parent. XR Origin contains all downloaded VR interaction packages, such as the orientation of the Player Camera, the VR controller's 3D movement and interaction with GameObjects.

Dragging and dropping elements from one interface to another is usually possible in Unity. For example, the textures of the planets can be dragged from the Project to the Scene directly; if they are compatible, Unity will implement it onto the GameObejct accordingly. A more detailed description of the interface is presented in [9].

Figure 2.2 shows that the simulated Solar System is populated with twelve GameObjects; eleven are 3D spheres, while one is a plane. The first sphere is the Solar System, acting as a script holder; the following nine spheres are the Sun and eight planets. Saturn's ring is made by placing a plane, as a child object, under Saturn, with its position frozen. And the Sphere Volume represents the light cast by the Sun, using a bloom effect. In the project tab, there is a downloaded package from the Unity asset store called Real Stars Skybox, which would be used for the background, and the textures for the GameObjects are obtained from "(Source)".

The implementation of the background can be easily accomplished by dragging and dropping the downloaded asset onto the scene. However, this would cause Problem (1), where the original textures of the planets' would turn purple, indicating an error. (Shown in Figure 2.3)

Figure 2.1: The interface of the editor, with the GameObject Sun highlighted. The Project shows textures used for the planets, and the inspector shows the position, scale and any external external components added to the Sun.

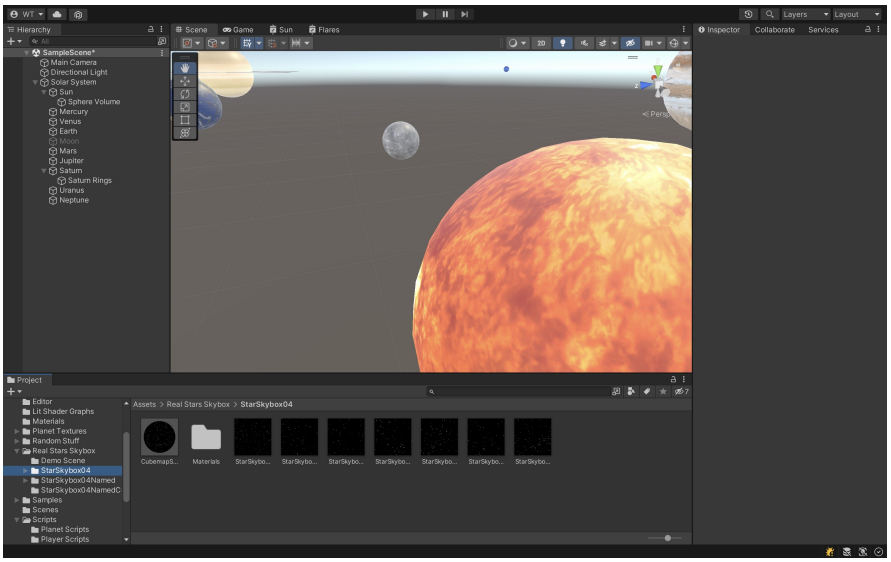

Figure 2.2: The interface, showing the downloaded asset package of the background, along with the star and planets. Currently the background is the default unity background.

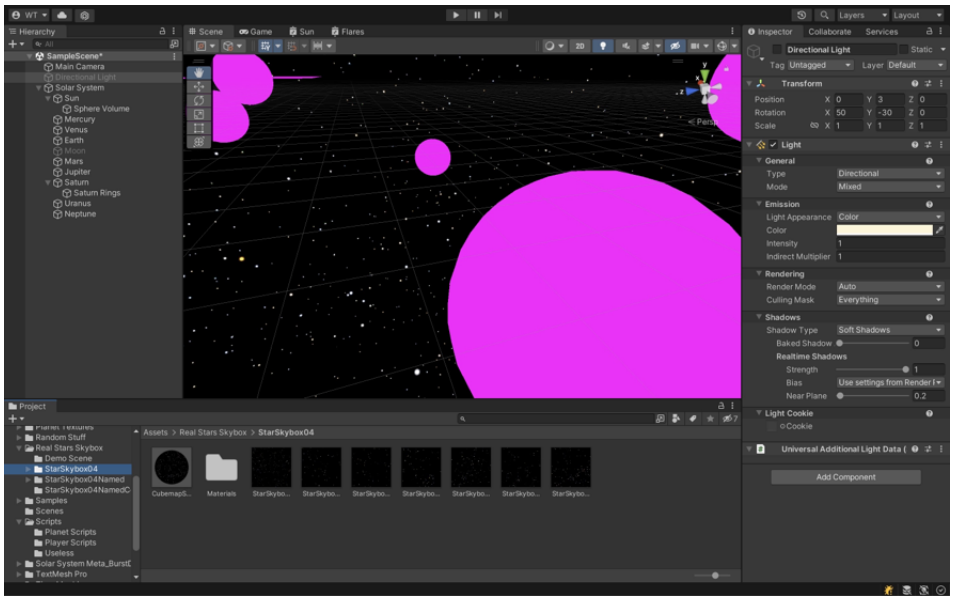

Figure 2.3: By implementing the background, the initial textures were undisplayed. This is likely due to both the downloaded background and textures of the GameObjects were incompatible.

A solution to Problem (1) was to create a texture through Unity by using a package called Universal Pipeline Rendering (UPR); this package allows for the creation of a Shader Graph, thus by taking a sample for each of the downloaded GameObejct textures; UPR would create nine Shader Graphs, eight of them for the star and planets and one for Saturns' ring (See Figure 2.4 up); by applying these to the GameObjects, they return the original textures. (See Figure 2.4 down)

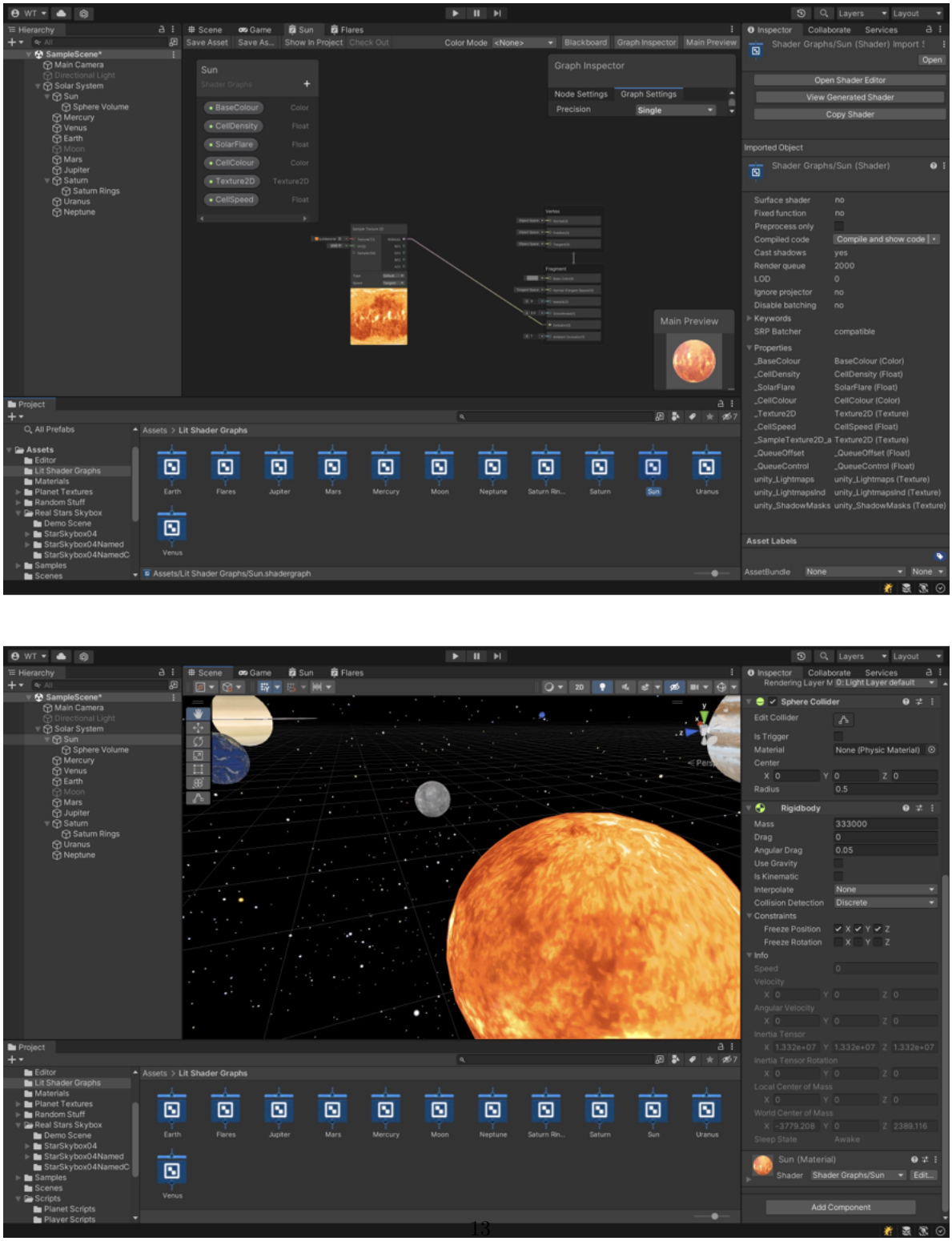

Figure 2.4: The top image shows the interface of the Shader Graph, where a sample of the Sun texture was taken, the bottom image shows the GameObjects with their respective textures along with the background.

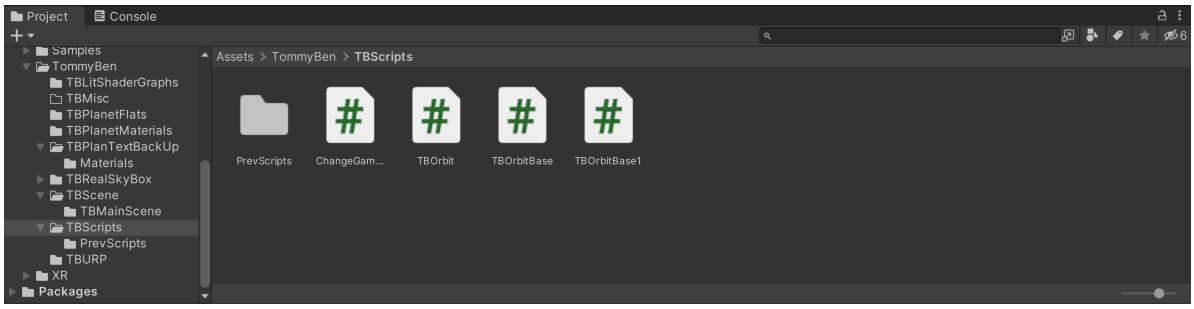

Figure 2.5: Four C♯ scripts are shown, where two of them are in used, to simulate the perihelion shift of Mercury and changing the speed of the simulation (ChangeGameTime and TBOrbitBase1). TBOrbitBase is the script containing the elliptical orbit, while TBOrbit contains the original circular orbit.

2.2 Visual Studio Coding

2.2.1 In-Game TimeScale and Units

Currently, two C♯ scripts are used in the simulation: ChangeGameTime and TBOrbitBase1. The ChangeGameTime allows the simulation to increase the speed at which the in-game time passes; this allows for the perihelion shift of Mercury to become more apparent quicker. (Observe Figure 2.6)

When choosing the units, it is essential to remember that the player must be able to see all of the GameObjects and their orbits effectively. Choosing units to scale would be more accurate if the goal was the create a realistic Solar System. Still, this simulation aims to show the perihelion shift or the orbit of Mercury; thus, the units are chosen to maximise that effect. The time selected is one unity second equals one week for Earth, the distance is one unity meter equals one billion meters, and the mass is one unity mass equals one Earth mass.

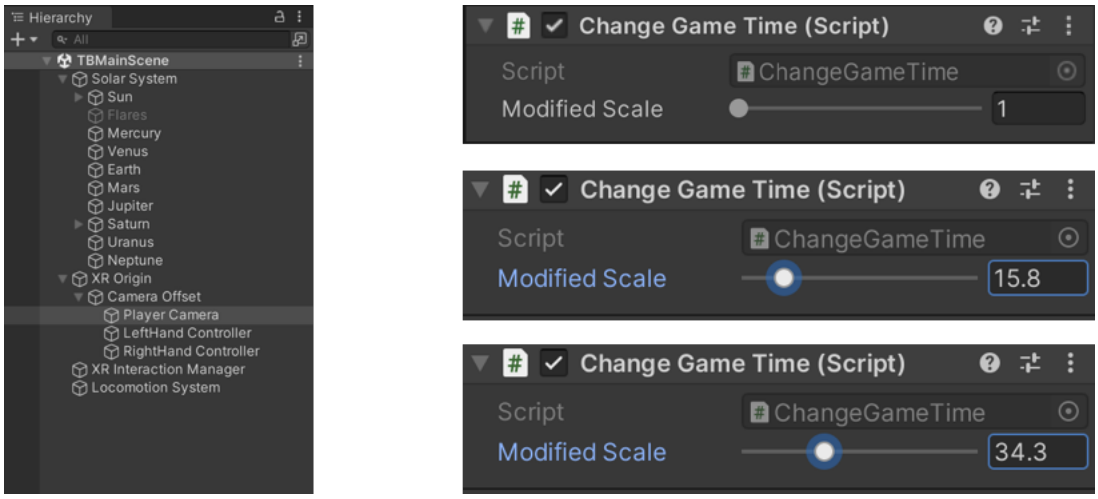

Figure 2.6: The script for ChangeGameTime is located in Player Camera, it is set to one initially, and would gradually increase to a higher scale (Depending on how powerful the computer is). The time-scale modify the in-game unity time by a factor of the number selected on the slider.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class ChangeGameTime : MonoBehaviour
6   {
7
8       [Range(1f,100)]
9
10      public float modifiedScale;
11
12      // Start is called before the first frame update
13      void Start()
14      {
15
16      }
17
18      // Update is called once per frame
19      void Update()
20      {
21
22          Time.timeScale = modifiedScale;
23
24      }
25  }
26
```

Figure 2.7: Time.timeScale = modifiedScale; would change the time-scale in Unity. Displays the codes for ChangeGameTime, where the scale ranges from 1 to 100; the time-scale is whatever the user sets it to.

```
readonly float G = 0.145802f; // readonly:NativeArray interface constrained to read-only operation,
                              // float:A keyword used to declare a variable which can store a floating point value in a certain range


readonly float c = 181314f;

GameObject[] celestials;

//public string Mercury { get; private set; }

// Start is called before the first frame update
void Start()
{

    celestials = GameObject.FindGameObjectsWithTag("Celestial");

    SetInitialVelocity();
}

// Update is called once per frame
```

Figure 2.8: This is the first section of TBOrbitBase1, which shows the values of the universal gravitation G and the speed of light c. It calls upon all the GameObjects with the tag Celestial, which encompasses the Sun and eight planets, then places them into a list called celestials (not Celestial). To place the GameObject into a list, use GameObject.Find(GameObjectName).

2.2.2 Solar System Script

The other C♯ script currently in use is TBOrbitBase1; this is the main body of our project, its purpose is to strike a balance between the simulated gravity and the applied velocity. TBOrbitBase1 accounts for the interaction between all the planets and the Sun, and its' primary purpose is to produce a relatively realistic perihelion shift of Mercury.

To call upon the position and mass of a GameObject, TBOrbitBase1 uses:

```
GameObject.Find(Mercury).transform.position;

GameObject.Find(Mercury).GetComponent<Rigidbody>().mass;
```

The radius can then be found by applying the command Vector3, which takes two positions and return its magnitude:

```
Vector3.Distance(GameObject.Find(Mercury).transform.position,

 b.transform.position)
```

The script also contains predetermined variables, such as the mass of GameObjects, universal gravitation, the speed of light and planets' angular momentum. Using these results, along with the radius and mass, TBOrbitBase1 can produce an initial velocity and gravity in the Solar System that result in a perihelion advance for Mercurys' orbit. (In section 2.3.3 there is a demonstration of the simulation)

```
void SetInitialVelocity()
{
    GameObject.Find("Mercury");
    {
        foreach (GameObject b in celestials)
        {
            if (!GameObject.Find("Mercury").Equals(b))
            {
                float m2 = b.GetComponent<Rigidbody>().mass;
                float r = Vector3.Distance(GameObject.Find("Mercury").transform.position, b.transform.position);
                GameObject.Find("Mercury").transform.LookAt(b.transform);
                //"Other" mass, m2 == M
                //Angular Momentum, L = 90.69
                float L = 90.69f;
                //Object Mass, m = 0.05527
                float m = 0.05527f;
                //Eccentricity, e = 0.2056
                float e = 0.2056f;
                GameObject.Find("Mercury").GetComponent<Rigidbody>().velocity += GameObject.Find("Mercury").transform.right
                * Mathf.Sqrt(((G * m2) / r) - ((L * L) / (2 * m * m * r * r)) + (((G * m2) / (c * c)) * ((L * L) / (m * m * r * r * r))));

            }
        }
    }
```

Figure 2.9: This is the second section of TBOrbitBase1, responsible for each planet's velocity and initial velocity. The script would locate the specified GameObject, which in this case is Mercury. Then the script would take another GameObject in the list celestials (not Mercury) and calculate their distance r. The script would transform the position of Mercury based on the other GameObjects' distance and mass; this action is performed for every element in the list of celestials. The Sun would have the largest effect, and the other planets are relatively negligible. The velocity of Mercury is also affected by its angular momentum L and its' mass, m1; the equation displayed represents the effective potential of Mercury relative to the Sun. The same code is applied for each of the planets.

```
void Gravity()
{
    foreach (GameObject a in celestials)
    {
        foreach (GameObject b in celestials)
        {
            if (!a.Equals(b))
            {
                float m1 = a.GetComponent<Rigidbody>().mass;
                float m2 = b.GetComponent<Rigidbody>().mass;
                float r = Vector3.Distance(a.transform.position, b.transform.position);


                a.GetComponent<Rigidbody>().AddForce((b.transform.position - a.transform.position).normalized * ((G * m2 * r_e * r_e) - (L * L * r_e) + (3 * G * m2 * L)));
            }
        }
    }
}
```

Figure 2.10: This is the last section of TBOrbitBase1, responsible for gravity simulations. The code takes a GameObject from the list celestials. It takes another GameObject in celestials (not the first GameObject) and transforms the first GameObject based on the mass and position of the second GameObject. The equation for the gravitational force contains r_e, which is the planet's orbit radius as a function of orbital angle (this is derived in section 3.3), along with the r cube term, resulting in a simulation of the perihelion shift of Mercury's orbit. This equation can be obtained through the derivative of the effective potential (section 3.1) with respect to the orbital radius.

2.3 Implementation of Virtual Reality

2.3.1 Perquisites to Virtual Reality and Issues:

There are several initial requirements for Unity to connect with a VR headset and interact with any third-party developer application. The required setting and packages are shown in Figure 2.11, and in addition to those requirements, the user must also register with Oculus to enable Developer Mode on the headset.

The virtual reality headset that would be used to run this simulation is the Oculus Quest 2. The Quest 2 can only store one account at a time, and this account is paired to an Oculus app; if two groups were to share one Quest 2, then the person registered with the Quest 2 would need their respective Quest app, else a factory reset is required to use it wirelessly.

(Problem (2) One of the major problems encountered in this project is due to the incompatibility between Mac OS, the Oculus app/Quest 2 and Unity; it is highly recommended that a Windows computer is used, as the Quest 2 can easily link to it through Quest Link)

(Problem (3) another important issue encountered is the virtual reality implementation; an error exists when the simulation is played off Quest 2 by selecting External Application in the app store. This caused the planets in the simulation to disappear if they had any movement associated with them (if their position were frozen through Rigidbody, they would appear in the Headset); this problem was avoided by using the Quest Link.)

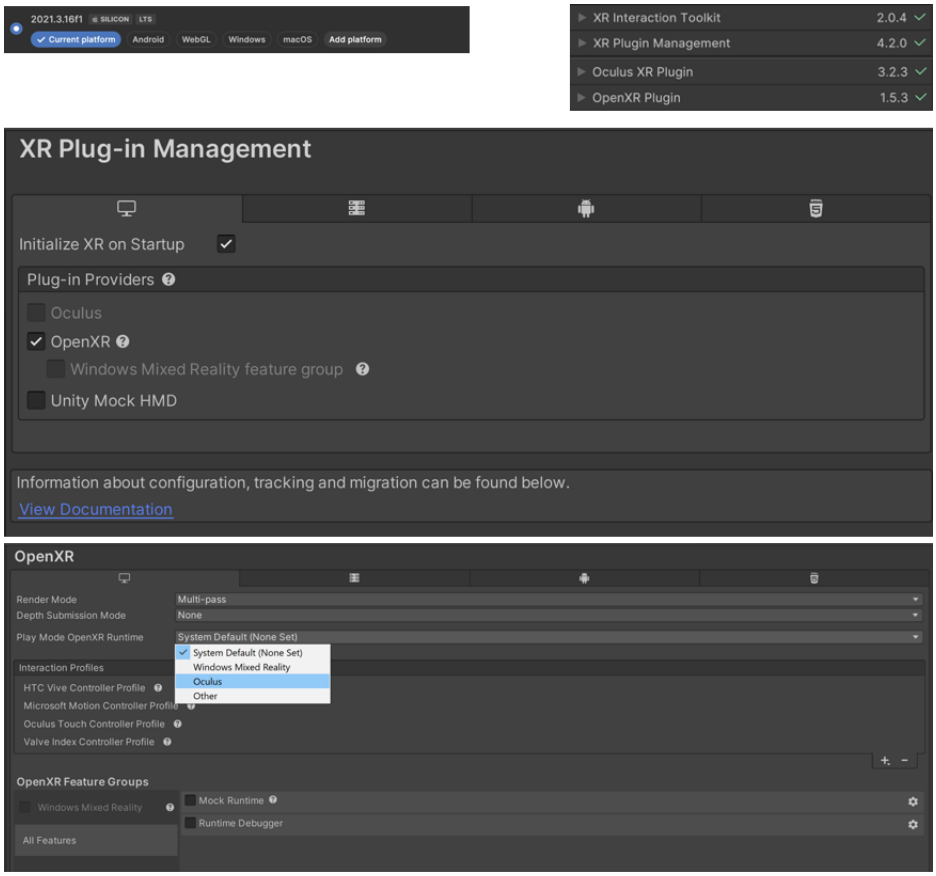

Figure 2.11: Top left of the figure, for Mac OS users, it is highly recommended to download a windows compatibility software from the Unity Hub; without this software any projects shared to a windows laptop would contain errors with scripts/packages. The top right of the figure shows the required packages to enable the implementation of VR compatibility with Unity. In the middle and bottom of the figure shows the setting enabled for VR headsets to interact with the simulation.

Figure 2.12: The figure shows what the player would obverse wearing the headset in the simulation.

2.3.2 Virtual Reality Controls and Movement:

The interface, when viewed from the headset, is shown in Figure 2.12. The player will have two light sources to interact with the Solar System. The field of view would move based on the head movement of the player.

Forward and backward movements in this space can be done using the left-handed controller while the right-handed controller rotates the camera 360 degrees. The player has complete freedom of movement in a 3D space, as the direction of motion is based on the direction the player is facing.

The player cannot interact with any of the GameObjects, in a way that disturbs their orbits, and the game object would appear hollow when the player moves into them.

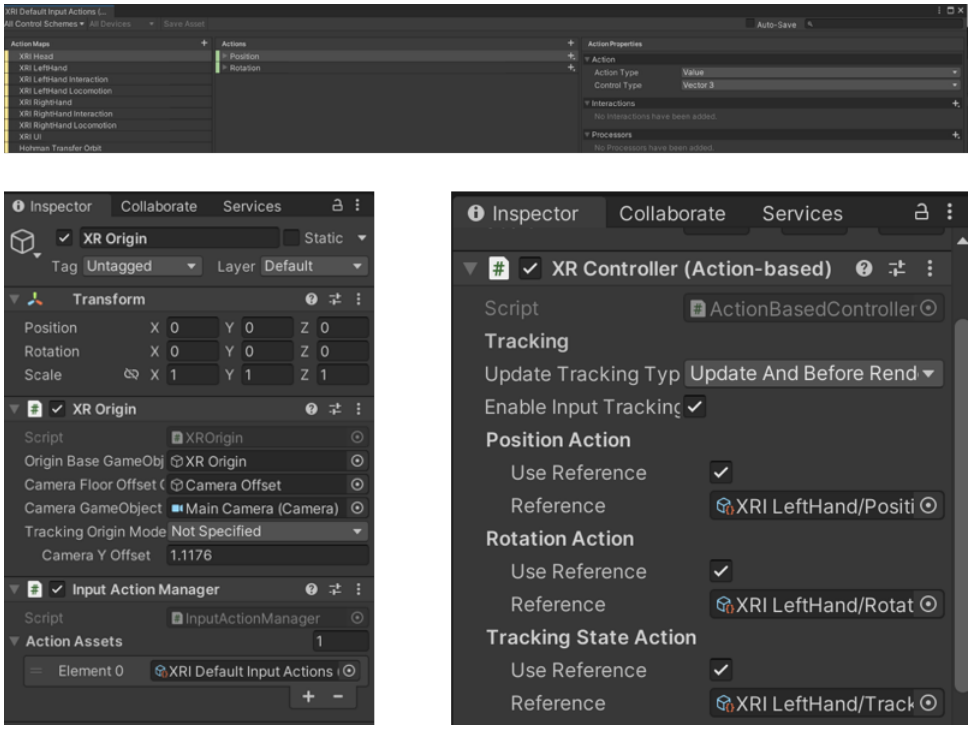

Figure 2.13: Settings for the VR controls, the top image shows the location to enable Vector 3, allowing the player to move in three dimensions freely. The bottom left image is the camera origin; this dictates where the player would start in the simulation. And the bottom right image is the script from the package XR Intractable, allowing the player to use the controllers for movement/view and GameObject interaction.

2.3.3 Virtual Reality Solar System

The link to a YouTube video of the simulation:
https://www.youtube.com/watch?v=nVYZeMKyNGg

Chapter 3

General Relativity Orbits and Perihelion Shift of Mercury

3.1 The derivation of the general relativistic potential

The equation for general relativistic orbits can be derived by combining the Schwarzschild Metric and the geodesic equations. The Christoffel symbols can be obtained from the Schwarzschild Metric:

$$\Gamma^t_{tr} = \Gamma^t_{rt} = \frac{r_s}{r(r-r_s)}, \Gamma^r_{tt} = \frac{r_s(r-r_s)}{r^3}\Gamma^r_{rr} = \frac{(r_s)}{r(r_s-r)}, \Gamma^r_{\theta\theta} = r_s - r,$$
$$\Gamma^r_{\phi\phi} = (r_s - r)(\sin\theta)^2, \Gamma^\theta_{r\theta} = \Gamma^\theta_{\theta r} = \Gamma^\phi_{r\phi} = \Gamma^\phi_{\phi r} = \frac{1}{r},$$
$$\Gamma^\theta_{\phi\phi} = -\sin\theta\cos\theta, \Gamma^\phi_{\theta\phi} = \Gamma^\phi_{\phi\theta} = \cot\theta$$

Inserting these Christoffel symbols into the four geodesic equations while assuming spherical symmetry and identifying which of these equations leads to a constant of motion, the complete derivation can be found in [7].

$$\frac{d^2(ct)}{d\lambda^2} + \Gamma^t_{\mu\sigma}\frac{dx^\nu}{d\lambda}\frac{dx^\sigma}{d\lambda} = 0, \frac{d^2r}{d\lambda^2} + \Gamma^r_{\mu\sigma}\frac{dx^\nu}{d\lambda}\frac{dx^\sigma}{d\lambda} = 0$$

$$\frac{d^2\theta}{d\lambda^2} + \Gamma^\theta_{\mu\sigma}\frac{dx^\nu}{d\lambda}\frac{dx^\sigma}{d\lambda} = 0, \frac{d^2x^\phi}{d\lambda^2} + \Gamma^\phi_{\mu\sigma}\frac{dx^\nu}{d\lambda}\frac{dx^\sigma}{d\lambda} = 0$$

The equations that lead to a constant of motion are (3.1) and (3.2):

$$\frac{d^2(ct)}{d\lambda^2} + \frac{r_s}{r(r-r_s)}\frac{d(ct)}{d\lambda}\frac{dr}{d\lambda} = 0$$

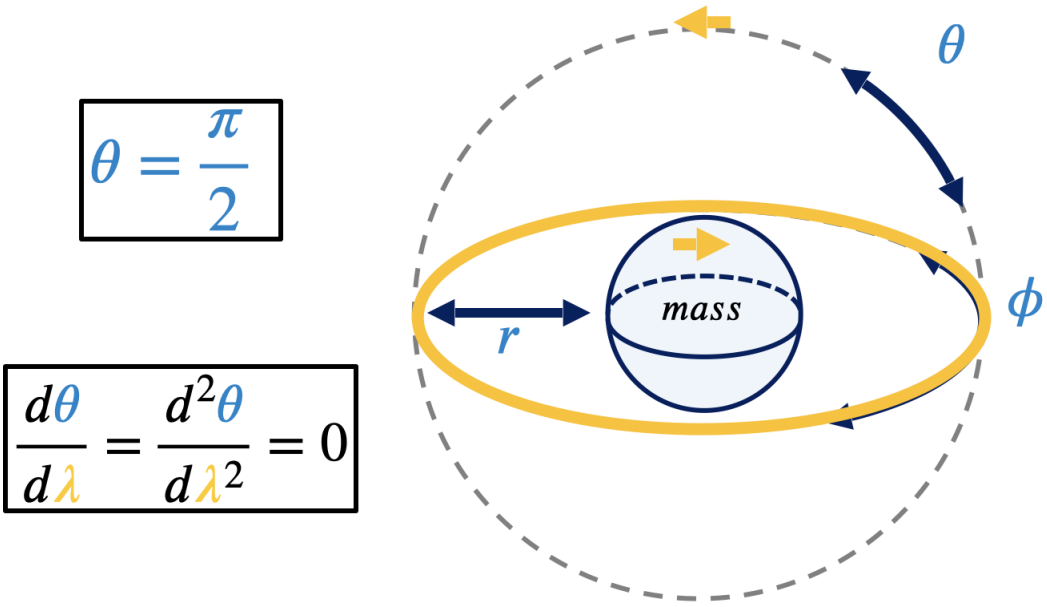

Figure 3.1: Due to a spherically symmetric central mass, an assumption for the geodesics having a constant $\theta = \pi/2$ orbit, in the plane of the equator is valid.

$$\Rightarrow \frac{d(ct)}{d\lambda}\left(1 - \frac{r_s}{r}\right) \equiv \frac{E}{m} \equiv \mathcal{E} \tag{3.1}$$

$$\frac{d^2\phi}{d\lambda^2} + \frac{2}{r}\frac{dr}{d\lambda}\frac{dr}{d\phi} = 0$$

$$\Rightarrow r^2\frac{d\phi}{d\lambda} \equiv \frac{L}{m} \equiv l \tag{3.2}$$

Equation (1.6) can be expanded with the Schwarzschild metric ; substituting in (3.1) and (3.2), let $\|d/d\lambda\|^2 = c^2$ rearranging and obtain equation (3.3). Substituting in l and ϵ to obtain equation (3.4)

$$\frac{1}{2}\mathcal{E}^2 = \frac{1}{2}\left(\frac{dr}{d\tau}\right)^2 + \frac{1}{2}c^2 - \frac{GM}{r} + \frac{L^2}{2r^2} - \frac{GM}{c^2}\frac{l^2}{r^3} \tag{3.3}$$

3.2 Potential Graphs and Orbits:

Both equations between Newtonian orbits and general relativity orbits have an energy term, a derivative respective to time which acts as the kinetic energy and the effective potential, which is a function of .

$$\left[\frac{1}{2}\left(\frac{E^2}{m}-mc^2\right)\right]_{\epsilon}=\frac{1}{2}m\left(\frac{dr}{d\tau}\right)^2+m\left[\left(-\frac{GM}{r}+\frac{L^2}{2m^2r^2}-\frac{GM}{c^2}\frac{L^2}{m^2r^3}\right)\right]_{V_{eff}(r)} \tag{3.4}$$

$$\rightarrow \epsilon=\frac{1}{2}m\left(\frac{dr}{d\tau}\right)^2+mV_{eff}(r)$$

By treating $G=M=m=1$, plots of the effective potential are shown in Figure 3.2; Figure 3.3 shows an extended potential graph; further along r is a shallow well where circular and elliptical orbits can exist, just like Newtonian gravity.

Unlike Newtonian gravity, general relativity predicts two possible circular orbits for a given angular momentum.

For an unstable circular orbit, a slight change in the energy would stop the body from orbiting; the second circular orbit farther away is a stable circular orbit, and a slight shift in the body's energy would cause the particle to enter an elliptical orbit. (See Figure 3.3)

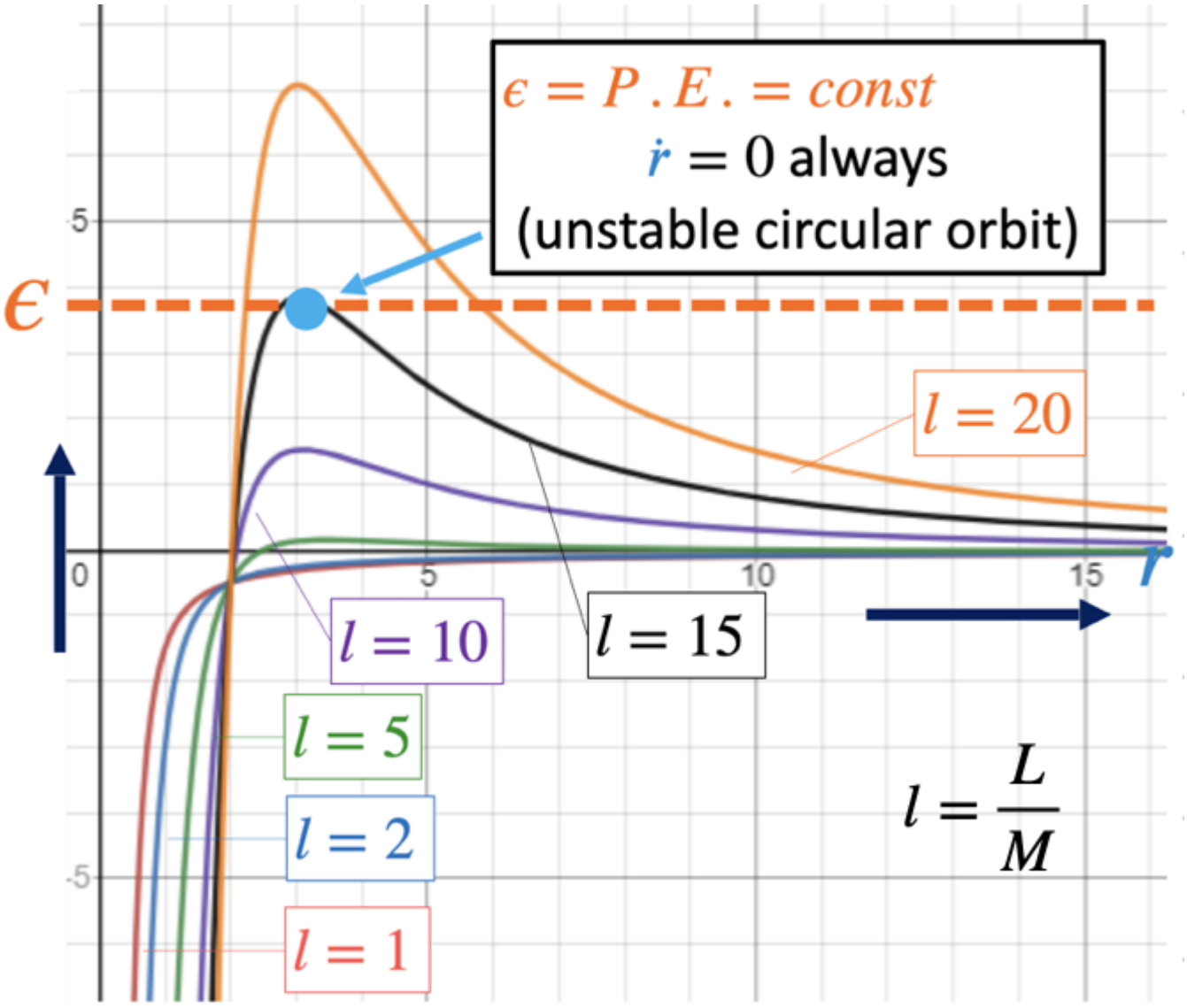

Figure 3.2: A plot of the effective potential as a function of and is representing the angular momentum divided by the mass. This plot only shows one stationary point, where the effective kinetic energy is zero.

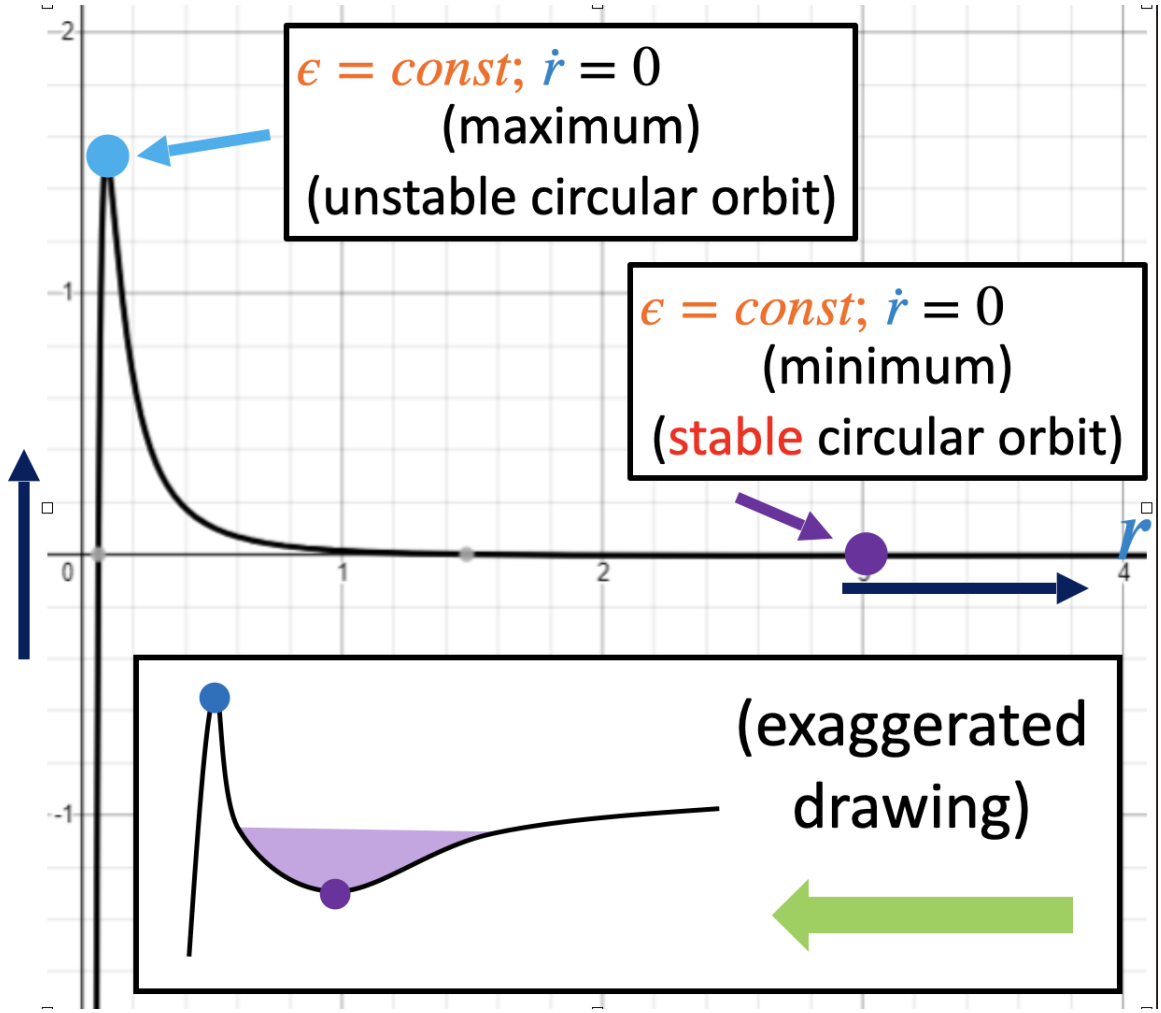

Figure 3.3: A plot of the effective potential; is extended much farther such that the second well, the stable orbit, is incorporated. The stationary point of the stable orbit is when the effective kinetic is zero.

3.3 The perihelion shift of Mercury:

Newtonian gravity predicts elliptical orbits of the planets around the sun, where the perihelion is the smallest distance between the planets and the Sun. In general relativity, the additional r^{-3} potential term causes the perihelion to shift in the direction of orbit, resulting in a flower-like orbit shape. However, this effect is minimal and is only noticeable for small values of r over long periods.

To derive perihelion shift, the goal is to get a formula for the planet's orbit radius r as a function of the orbital angle ϕ.

Equation (3.3) can be rearranged into equation (3.5); the left-hand side is in the form of a simple harmonic oscillator with a sinusoidal solution and the addition of a constant; the value e is the eccentricity. The last term on the right-hand side of equation (3.5) is the correction from general relativity and corresponds to the term in the potential. By expanding using a Taylor series and letting , the first two terms are the Newtonian solutions (w_0) with a small perturbation (w_1).

$$\alpha = \frac{3}{4}\frac{c^2 r_s^2}{l^2}, w = \frac{1}{r}\frac{2l^2}{c^2 r_s}$$

$$\frac{d^2 w}{d\phi^2} + w = 1 + \alpha w^2 \tag{3.5}$$

Assuming a solution for w_1 ; find the first and second derivative, then substitute back into the series to obtain equation (3.6). Where the term, $\alpha\phi \sin\phi$ is Mercury's perihelion shift, collapse the term further to obtain equation (3.7).

$$w_1 = \left(1 + \frac{1}{2}e^2\right) + (a\phi + A)\sin\phi + (b\phi + B)\cos 2\phi$$

$$\frac{d^2 w}{d\phi^2} + w = 1 \Rightarrow w(\phi) = 1 + e\cos(\phi + \phi_0)$$

$$w = w_0 + \alpha w_1 + \alpha^2 w_2 + \ldots$$

$$\Rightarrow \frac{d^2 w_0}{d\phi^2} + w_0 = 1 \; ; \frac{d^2 w_1}{d\phi^2} + w_1 = (1 + e\cos\phi)^2$$

$$w = 1 + \alpha\left(1 + \frac{1}{2}e^2\right) + e(\cos\phi + \alpha\phi\sin\phi) - \alpha\frac{1}{6}e^2\cos 2\phi + \ldots \tag{3.6}$$

$$\cos([1-\alpha]\phi) \Rightarrow \phi = \frac{2\pi}{1-\alpha} \approx 2\pi + 2\pi\alpha \tag{3.7}$$

Finally, rearranging equation (3.8) into equation (3.9) to obtain $r(\phi)$, where the perihelion shift can be identified as $2\pi\alpha$. (Where a is the semi-major axis)

$$w(\phi) = 1 + e\cos(2\pi + 2\pi\alpha) = \frac{1}{r}\frac{2l^2}{c^2 r_s} \tag{3.8}$$

$$r(\phi) = \frac{(1-e^2)a}{1 - e\cos(2\pi + 2\pi\alpha)} \tag{3.9}$$

Chapter 4

Summary

The simulation for the perihelion shift of Mercurys' orbit in virtual reality is constructed through several stages: nine spherical GameObjects, of which eight represent the planets, and one represents the Sun, were spawned within the default Unity Scene. A background from Real Stars Sky Box was implemented, and an error regarding the texture occurred, requiring the need to import the package Universal Rendering Pipeline to use a Shader Graph that would create a compatible texture that could be re-implemented onto the GameObjects. The equation that governs the gravitational force in the Solar system can be found by differentiating equation (3.4) with respect to the orbiting radius, and the velocity of the planets can be obtained by setting the conditions presented in section 1.3.1, approximating a relatively circular orbit and obtaining the orbital velocity for general relativity. The C♯ script TBOrbitBase1 contain the derivations of orbital velocity and gravity, along with additional variables, such as the speed of light, the universal gravitation and angular momentum; the script mainly contains commands which call and retrieve variables from Vector3, Transforms and RigidBody, and the script would allow for mutual interaction between GameObjects. To enable virtual reality compatibility, download XR origins and XR interactions from the package manager in Unity; these packages allow for setting movement and camera commands on the VR controllers. XR origins allow the Oculus Quest 2 to simulate Quest Link. Through section 3.3, the perihelion advance is shown to be the component $2\pi\alpha$; this is shown in TBOrbitBase1 as r_e (representing equation 3.9), which causes the resultant perihelion advance shown in the simulation. Hopefully, this simulation could reach the hands of many interested young scientists with the aim of better engagement in education.

Bibliography

[1] : Cohen, I. Bernard; Smith, George Edwin (2002). The Cambridge Companion to Newton. Cambridge University Press. pp. 11–12, 96–97. ISBN 978-0-521-65696-2.

[2] : Cavendish, H. 'Experiments to determine the Density of the Earth', Philosophical Transactions of the Royal Society of London, (part II) 88 pp. 469–526 (21 June 1798), reprinted in Cavendish 1798

[3] : Max Born (1924), Einstein's Theory of Relativity (The 1962 Dover edition, page 348 lists a table documenting the observed and calculated values for the precession of the perihelion of Mercury, Venus, and the Earth.)

[4] : Roger A. Freedman; Todd Ruskell; Philip R. Kesten; David L. Tauck. (n.d.). College physics 3rd edition: Roger A. Freedman: Macmillan Learning. 3rd Edition — Roger A. Freedman — Macmillan Learning. Retrieved April 24, 2023, from https://store.macmillanlearning.com/us/product/College-Physics/p/1319255345

[5] : Technologies, U. (n.d.). Unity user manual 2021.3 (LTS). Unity. Retrieved April 24, 2023, from https://docs.unity3d.com/Manual/index.html

[6] : Technologies, U. (n.d.). Rigidbody component reference. Unity. Retrieved April 24, 2023, from https://docs.unity3d.com/Manual/class-Rigidbody.html

[7] : Carroll, S. (n.d.). Lecture notes on general relativity - S. Carroll. Retrieved April 21, 2023, from https://ned.ipac.caltech.edu/level5/March01/Carroll3/Carroll7.html

[8] : Lim, E. (n.d.). 6CCP6363 General Relativity. 6CPP3630 general relativity. Retrieved April 24, 2023, from https://nms.kcl.ac.uk/eugene.lim/teach/GR/GR.html

[9] : Technologies, U. (n.d.). Unity's interface. Unity. Retrieved April 24, 2023, from https://docs.unity3d.com/Manual/UsingTheEditor.html