Virtual Reality Solar-System Grand Tour

Third Year Project
6CCP3131

April 2022

Student:
Mohammad Shikha
Collaborators:
Jiahua Fan
Supervisor:
Professor Eugene Lim

Department of Physics
King's College London

Abstract

This paper implements Newtonian physics into the Unity engine to numerically model both the solar-system and a simplified orbit of Voyager 2, from its launch in August 1977. The benefits of Virtual Reality can be utilised to create an immersive experience of the grand-tour made by Voyager 2 during its fly-by of the outer planets. Evaluation of the accuracy of the simulation was made in comparison to the real-world data provided by the NASA Jet Propulsion Laboratory, with compromises made to the starting positions of Saturn in order to satisfy the grand-tour objective. Overall, the simulation developed was sufficient in predicting the planetary orbital motion, at a 10 year timescale, therefore a grand-tour could be achieved. Modifications to the initial conditions for the simulation were made and documented, along with other amendments to the original implementations. Overall, the simulation provides an educational package for interplanetary travel.

Contents

1 Introduction

The recent development of Virtual Reality (VR) technologies has provided Physicists the opportunity to implement interactive modelling of 3D space. As VR devices have become more widely available, applications such as science communication provide an opportunity to exhibit physics through an engaging medium.

A successful VR experience for learning can be characterised by these design frameworks as purported by (J Radianti et al)[1].

Realistic surroundings	A high quality textured environment which replicates the the targeted scene with sufficient scale.
Passive observation	Students can look around the virtual environment which also applies as they travel along a predetermined path.
Basic interaction	Students can interact with the environment in which they are presented.
Immediate feedback	Provided feedback as to whether the interactions with their environment were successful.

Table 1: VR design framework

The utilisation of VR in astrophysics has the potential for immersive and interactive experiences, providing users opportunities to fathom the scale of celestial objects and provide greater understanding on how the theory translates to the scenes implemented. Analysis of the simulation can be made through the use of ephemerides of solar-system objects, with the NASA JPL Horizons system[2] providing highly accurate and flexible data.

Therefore, implementing the orbital mechanics of the solar-system for this VR application is accessible as verification can be made. Furthermore, features of the planetary motion, such as elliptical orbits, must be implemented.

Kepler's laws in conjunction with Newtonian gravitation, are the basis to an approximate description of planetary motion[3]. Through the use of computation, simulations of the planetary motion can be made once these laws are implemented.

As a result of the understanding on the mechanics of the solar-system, the 20th century saw developments into interplanetary exploration, notably the famous Voyager missions. The alignment of the outer-planets in the solar-system lead to the potential for a grand-tour, defined as a single spacecraft visiting all the outer-planets through the use of gravity assist, as proposed by (G.A Flando)[4]. The Voyager program launched in 1977 with the second craft Voyager 2 visiting all of the outer-planets and as of 2022, is operating within interstellar space, transmitting data towards Earth[5]. The opportunity to implement the Voyager 2 flight-path in VR will provide users a first-person perspective of interplanetary travel and intuition of the data collection undertaken by Voyager 2 during its flybys with the outer-planets.

Henceforth, the Unity engine has the capability of assembling VR tools to provide experimental VR applications for free. Moreover, the engine allows for high-level programming using the $C\#$ language, allowing developers flexibility in implementing physics scripts into the application, without complex syntax and memory management. Furthermore, the detailed documentation[6] on the engine provides support for development.

Evaluation of the grand-tour simulation will be based on both the physics accuracy and VR experience based on the parameters of table (1).

2 Theory

2.1 Newton's Law of Gravitation

The fundamental equations for the simulation will use Newton's Law of Gravitation, derived by Isaac Newton. It is a physical law that states every point mass attracts every other point mass through a force acting along the distance between them[3]. Within vector form, it is mathematically defined as[7]

$$\mathbf{F}_{21} = \frac{-Gm_1m_2}{|\mathbf{r}_{21}|^2}\hat{\mathbf{r}}_{21} \tag{1}$$

- $\mathbf{F}_{21}$ is the exerted force from object 1 onto object 2, measured in Newtons (N)
- m_1 and m_2 are the masses of objects 1 and 2, measured in (kg)
- G is the gravitational constant, measured in $(\mathrm{m^3kg^{-1}s^{-2}})$
- $|\mathbf{r}_{21}|^2$ is the magnitude of the distance, between objects 1 and 2 $(\mathrm{m^2})$
- $\hat{\mathbf{r}}_{21}$ is the unit vector, forming the base of the vector space

The vector form easily conveys Newton's Third Law of motion, in which an equal and opposite force is exerted by object 2 onto object 1 in this case, $\mathbf{F}_{21} = -\mathbf{F}_{12}$.

For a simplified circular orbit, the gravitational force defined in equation (1) acts as a centripetal force, causing the tangential velocity of the secondary object to constantly change direction, thereby accelerating. In scalar form, this is provided by the equation below

$$F = \frac{mv^2}{r} \tag{2}$$

- m is the mass of the subjected object under circular motion (kg)
- v is the velocity of the object $(\mathrm{ms^{-1}})$
- r is the radius of the circular motion (m)

The value for the velocity in which circular motion occurs is defined through the substitution of equations (1) into equation (2).

$$v = \sqrt{\frac{Gm_2}{r}} \tag{3}$$

As the motion is perfectly circular, the magnitude of the velocity remains unchanged. Circular motion and Newton's Law of Gravitation provides an intuitive understanding on the orbital mechanics of the planets around the Sun. However, the circular motion approximation is inaccurate as the planets orbit elliptically at different eccentricities.

Furthermore, limitations on Newton's Law of Gravitation are highlighted when dealing with very strong gravitational fields, such as the one present between Mercury and the Sun. This was resolved following Albert Einstein's Theory of General Relativity which increases the accuracy for calculating the gravitational field overall, especially for extremely dense objects or objects at small distances between each other[3]. Despite this, Newtonian mechanics remains suitable for the aims of the simulation, as it can predict the motion of the remaining planets to a high degree of accuracy[3].

2.2 Elliptical Orbits

As stated previously, the orbits of the planets are elliptical, as described by Johannes Kepler with his three laws of planetary motion. These laws describe the motion of the planets in the presence of a universal gravitational force, in which Newton's laws of motion were later used to derive each law. Kepler's laws allow for the prediction of the position and velocity of the planets at a given time[8].

There are some assumptions underlying the planetary orbital mechanics:

- The Sun's mass is very large compared to the planets, therefore its motion is unaffected by their gravitational pull.
- The orbits of the planets act on a 2D plane as the gravitational force acts along the separation vector distance between the Sun and a given planet.
- Mechanical energy is conserved, any form of energy dissipation can be seen as negligible.

With these assumptions, Kepler's Laws can be defined.

2.2.1 Kepler's Laws

The first law states that "The orbit of every planet is an ellipse with the Sun at a focus"[8]. Therefore, understanding of the mathematics behind ellipses is required.

An ellipse is defined as a type of conic section. The general equation for any conic section is[9]

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \tag{4}$$

- Where A,B,C,D,E,F are constants

The standard form of an ellipse with a horizontal major axis is

$$\frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2} = 1 \tag{5}$$

- (h, k) is the centre of the ellipse
- a is the semi-major axis
- b is the semi-minor axis

The semi-major axis is defined as being half the length of the longest diameter of an ellipse, while the semi-minor is half the shortest diameter.

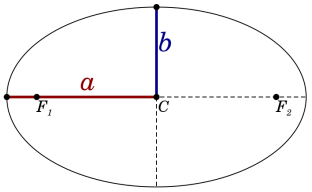

Figure 1: Ellipse labelled with the semi-major/minor axes

Calculation of the semi-major/minor axes can be completed using the coefficients of the conic equation[10] (4).

$$a = \sqrt{\frac{2(Ah^2 + Ck^2 + Bhk - 1)}{A + C + \sqrt{(A-C)^2 + B^2}}} \tag{6}$$

$$b = \sqrt{\frac{2(Ah^2 + Ck^2 + Bhk - 1)}{A + C - \sqrt{(A-C)^2 + B^2}}} \tag{7}$$

These terms can be used to describe the shape of the ellipse, specifically the eccentricity which includes the ratio between the semi-major/minor axes.[9]

$$e = \sqrt{1 - \frac{b^2}{a^2}} \tag{8}$$

The eccentricity is related to the distance between the two focii of the ellipse, with a perfect circle having the two focii placed within the same position central position, thereby $e = 0$. Moreover, the equation deduces that the eccentricity of an ellipse is always $0 < e < 1$.

Kepler's Second Law

Kepler's Second Law of planetary motion states that a line segment joining a planet and the Sun sweeps out equal areas at equal intervals of time[8]. This indicates that the orbital velocity of the planet varies as it travels across its elliptical orbit, with the planet travelling faster when at its perihelion and slowest at its aphelion. This differs from the circular motion approximation, as equation (3) indicates that the velocity magnitude remains constant.

Kepler's Third Law

The third law states that the square of a planets orbital period is proportional to the cube of the semi-major axis length[8].

$$T^2 = \frac{4\pi^2}{GM} a^3 \tag{9}$$

- T is the orbital time period (s)

- M is the combined mass m_1+m_2 of the two orbiting objects, the mass of the Sun is much greater than that of the planets, therefore the planetary mass is negligible, (kg).

2.2.2 Planetary Velocities

As a result of the Keplerian elliptical orbits, the velocity can be derived from the conservation of energy and momentum of the planetary orbits. This provides the Vis-viva equation assuming the secondary body mass is relatively small[3].

$$v = \sqrt{GM(\frac{2}{r} - \frac{1}{a})} \tag{10}$$

- v is the relative speed of the two bodies (ms^{-1})
- M is the mass of the central body (kg)
- r the distance between the centre of masses of the two bodies

As explained through Kepler's second law, the relative speed of the planet changes as it travels across its elliptical orbit, this is due to the non-constant r quantity. Calculating the initial velocity of the planets using the Vis-viva equation through the Unity engine, provides the opportunity to verify the planetary motion based on the central gravitational force.

The magnitude of the velocity is provided simply with the Vis-viva equation (10). This requires the calculation of the semi-major axis to be made for each of the planets. Henceforth, calculation of a requires the construction of conic equations (4) for each planet. A mathematical method to determine conic equations with 5 separate points along the ellipse circumference can be employed to achieve this[11]. Despite equation (4) having 6 unknown coefficients, as it is homogeneous, dividing through with a non-zero coefficient provides a new equation with the non-zero coefficient set to 1, thereby leaving 5 unknowns. Furthermore, linear algebra can now be utilised to solve for all the coefficients using the 5 known points[9].

$$\det \begin{bmatrix} x_1^2 & x_1y_1 & y_1^2 & x_1 & y_1 & 1 \\ x_2^2 & x_2y_2 & y_2^2 & x_2 & y_2 & 1 \\ x_3^2 & x_3y_3 & y_3^2 & x_3 & y_3 & 1 \\ x_4^2 & x_4y_4 & y_4^2 & x_4 & y_4 & 1 \\ x_5^2 & x_5y_5 & y_5^2 & x_5 & y_5 & 1 \end{bmatrix} = 0 \tag{11}$$

- (x_1,y_1),(x_2,y_2)... are individual points along the ellipse

These points can be retrieved from the Horizons ephemeris[2], spread across the orbital period of the planets from the launch of Voyager 2 in 1977. After finding the conic equation, calculation of the semi-major axis can be made with equation (6).

Furthermore, the unit-vector direction of the velocity must be found and applied to the calculated speed. This direction points tangentially to the ellipse, therefore the gradient of this tangent can be found with the first-differential of the conic equation and a tangential straight line equation can be made with the known starting position. This can then be normalised to a unit-vector.

$$\frac{dy}{dx} = -\frac{2A + By + D}{Bx + 2Cy + E} \tag{12}$$

Furthermore, the planets should orbit anti-clockwise when viewed from the northern pole of the Sun, this can be set based on the initial relative positions of the planets and setting the appropriate unit-vector sign to accommodate for this. After setting the initial position and velocities for the planets, Kepler's Laws should operate in conjunction with the Newtonian gravitational forces experienced by each planet.

2.3 Gravity Assist

Voyager 2 was launched on the 20th of August 1977 with the mission to visit the outer-planets, visiting Jupiter, as well as Saturn Uranus and Neptune through course correction[5]. The most economical method of undergoing inter-planetary travel is by utilising gravity assist, using the gravitational pull of planets to gain velocity relative to the Sun, which has the benefit of increasing the velocity of the spacecraft without the need for extra propellant[12].

A delicate operation is required to ensure the spacecraft falls within the gravitational field of the planet without crashing into its surface, as well as ensuring the velocity gained within the orbital field of the planet is greater than the planetary escape velocity, to prevent the spacecraft from being confined to an orbit of the planet.

In accordance to Newton's third law, any gain or loss of kinetic energy for the spacecraft is subsequently lost or gained by the planet providing the gravity-assist, ensuring conservation of energy. As the mass of the planets are much greater than the mass of the spacecraft, the energy lost by the planet is minuscule in comparison to its total energy.

Derivation of the velocity of the spacecraft and planet can be made using the elastic conservation of energy and momentum equations

$$\frac{1}{2}m_1u_1^2 + \frac{1}{2}m_2u_2^2 = \frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 \tag{13}$$

- u_1, u_2 initial velocity of the spacecraft and planet respectively, (ms^{-1})
- v_1, v_2 final velocity of the spacecraft and planet respectively, (ms^{-1})
- m_1, m_2 masses of the spacecraft and planet respectively, (kg)

$$m_1u_1 + m_2u_2 = m_1v_1 + m_2v_2 \tag{14}$$

Combining equations (13) and (14), values for the final velocity can be made with known initial velocities.

$$v_1 = \frac{m_1 - m_2}{m_1 + m_2}u_1 + \frac{2m_2}{m_1 + m_2}u_2 \tag{15}$$

$$v_2 = \frac{2m_2}{m_1 + m_2} u_1 + \frac{m_2 - m_1}{m_1 + m_2} u_2 \tag{16}$$

As the mass of the spacecraft m_1 is negligible compared to the mass of the planet m_2, the final velocities can be approximated to[13]

$$v_1 \approx -u_1 + 2u_2 \tag{17}$$

$$v_2 \approx u_2 \tag{18}$$

These formulae assume an angle of approach of $\theta = 0°$ to the planet, which causes the rocket to change direction by 180° as shown by the change in sign in equation (15). In the event of an angled approach, vector addition is required.

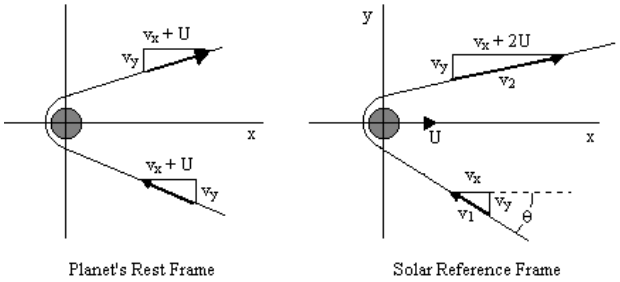

Figure 2: Reference frames with an angle of approach between the planet and spacecraft.[13]

The spacecrafts initial velocity with respect to the solar reference frame is therefore composed in the x and y-axis

$$(u_1)_x = -u_1 \cos(\theta) \tag{19}$$

$$(u_1)_y = u_1 \sin(\theta) \tag{20}$$

With the understanding that equation (15) acts along the x-axis, equation (19) is substituted into it, providing the final velocity of the spacecraft along the x-axis. The final velocity along the y-axis remains unchanged.

$$(v_1)_x = u_1 \cos(\theta) + 2u_2 \tag{21}$$

$$(v_1)_y = u_1 \sin(\theta) \tag{22}$$

Using the Pythagorean theorem, the magnitude of the final velocity of the spacecraft can be made, and is simplified as[13]

$$v_1 = (u_1 + 2u_2)\sqrt{1 - \frac{4u_2 u_1 (1 - \cos(\theta))}{(u_1 + 2u_2)^2}} \tag{23}$$

With this function being graphically modelled as such.

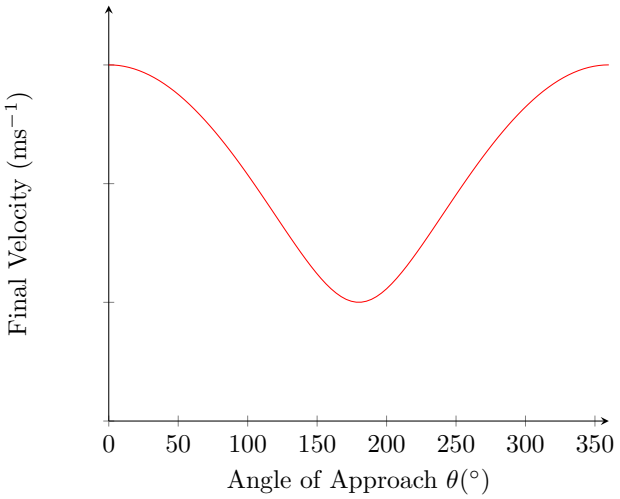

Figure 3: Final velocity against angle of approach

2.3.1 Voyager 2 Parameters

Using the Horizons ephemeris,[2] initial conditions of the spacecraft can be made with the appropriate unit conversions, these units are limited by the use of float numeric-types, as will be explained in the scaling section 2.5.

Removal of the acceleration of Voyager 2 can be achieved by setting the initial velocity 1 day after launch, thereby eliminating the complexity of rocket acceleration.

Initial	SI Unit	Unity Unit
Position	x: 1.29×10^{11}m	x: 86.6U_L
	y: -7.99×10^{11}m	y: $-53.7U_L$
Velocity	x: 1.73×10^{4}ms^{-1}	x: $1.01U_L U_T^{-1}$
	x: 3.47×10^{4}ms^{-1}	y: $2.01U_L U_T^{-1}$

Table 2: Initial starting position and velocity of Voyager 2 that should be implemented in the simulation.[2]

Physical	Unity Unit	SI Unit
Mass	$1 \times 10^{-7}U_M$	5.97×10^{17}kg
Size	x: $1 \times 10^{-7}U_L$	x: 147m
	y: $1 \times 10^{-7}U_L$	y: 147m
	z: $1 \times 10^{-7}U_L$	z: 147m

Table 3: Mass and size of the simulated spacecraft based on the smallest values of the fundamental units.

The smallest value for the mass as a result of floats is quite large compared to the true mass value of Voyager 2 (826kg)[14]. Despite this, the mass is of multiple orders of magnitude smaller than the masses of the outer planets[15], meaning the planetary orbits will not be affected much. Moreover, the size of the rocket is also larger than the true size of Voyager 2, however it remains small enough to prevent the rocket from crashing into the surface of the outer planets once it reaches the closest approach.

Parameters for the closest approach from the surface of the planets and subsequent dates are also listed for Voyager 2.

Planet	DoA	Flyby(km)	Flyby(U_L)
Jupiter	9/7/1979	5.70×10^5	3.83×10^{-1}
Saturn	26/8/1981	1.01×10^5	6.78×10^{-2}
Uranus	24/1/1986	8.15×10^4	5.47×10^{-2}
Neptune	25/8/1989	4.95×10^3	3.32×10^{-3}

Table 4: Voyager 2 closest approach to each planet with the subsequent date of arrival.[14]

The kinetic energy gained as a result of the gravity-assist provided by the outer planets has been plotted alongside the escape velocity of the solar-system.

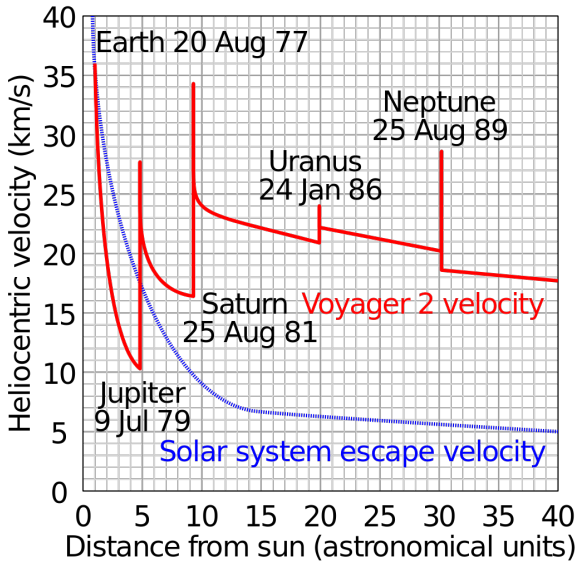

Figure 4: Voyager 2 velocity against its distance from the Sun.[16]

2.4 Unity Engine

The Unity Engine operates under a scripting lifecycle, which executes a number of event functions in a predetermined order. This includes the scene, physics engine, game logic and UI[6].

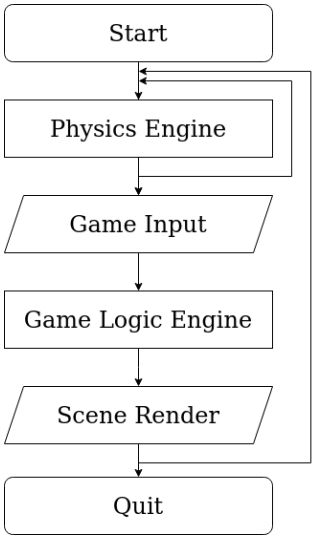

Figure 5: Unity script life-cycle flowchart.

The physics engine operates under the **FixedUpdate** event,[6] which is a frame-rate independent function that has a set frequency for computing physics calculations. Ordinarily, this frequency is set under the **Time.fixedDeltaTime** float at a default value of 0.02, which provides 50 physics calculations per second[6]. It is important for the physics engine to operate under a constant fixed change in time in order for consistent motion of objects. Decreasing the **Time.fixedDeltaTime** value provides greater accuracy in calculating the physics and movement of objects at the cost of greater computational power.

Moreover, the physics engine operates in conjunction with the **Rigidbody** component script. This component can be edited for required physics parameters and execute public methods when attached to game-objects in the simulation[6].

Scene editing, VR input events, and the HUD can all be configured in the Unity scene editor along with attached scripts to perform the required game functions.

2.5 Scaling and Units

The scaling of the simulation must first be defined by fundamental units. These 3 units include:

- The Unity Length (U_L)
- The Unity Mass (U_M)
- The Unity Time (U_T)

These fundamental units are chosen with regards to the limitations of the Unity engine and aims of the simulation. Maximising precision and performance on consumer-grade hardware is required to produce an accurate and accessible simulation, therefore a choice has to be made on the numerical variable data types used within the Unity engine. Hence, the 3 main floating-point numeric types used to represent numbers using the C# script are outlined with their attributes in the table below.

C# Type	Range(±)	Precision (digits)	Size (bytes)
float	1.5×10^{-45} to 3.4×10^{38}	6-9	4
double	5.0×10^{-324} to 1.7×10^{308}	15-17	8
decimal	1.0×10^{-28} to 7.9×10^{28}	28-29	16

Table 5: All floating-point types in C# and their properties.[17]

The float type has the least memory demands with sufficient range and precision. It is also the default numeric type for the **Vector3** data structure in Unity,[6] which contains the prerequisite vector properties required for the simulation.

This reduces the development time for constructing vector attributes using either doubles or decimals. Henceforth, floats can be used for numerical variables, so long as the fundamental units are chosen with regards to its 7-digit precision.

The final scaling for the fundamental units is provided by the table below.

Unity Unit	Real Value	SI Unit
100 U_L	1 AU	1.49×10^{11}m
1 U_M	1 M_{Earth}	5.97×10^{24}kg
1 U_T	1 Day	8.64×10^{4}s

Table 6: Fundamental simulation units and their scaling.

The rationale behind these scaled units is as follows. The Unity Length U_L was chosen to be scaled to astronomical units, defined as an approximate distance between the Sun and Earth, thereby providing sufficient scaling for other celestial objects within the solar-system, hence its use within astronomy. Moreover, defining $1U_L = 1AU$ within the scene editor of the Unity engine limits the spacing between planets, especially the inner-planets. Thus, an appropriate scale would be to define $100U_L = 1AU$ which provides sufficient inter-planetary distance without adjusting near-field camera rendering (defaulted at $0.01U_L$). Furthermore, the smallest length value of the planets would be the radius of Mercury, which converts to $1.64 \times 10^{-3}U_L$, comfortably within the 7-digit precision of floats.

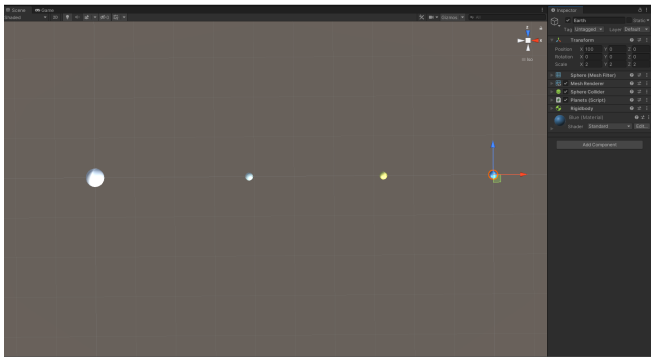

Figure 6: Placement of the inner-planets in the scene editor

Choosing a mass unit must also account for the 7-digit precision of the float type. The largest mass in the simulation is the Sun, with the smallest being the mass of Mercury.

- $M_{Sun} \approx 1.99 \times 10^{30}\text{kg} \approx 3.33 \times 10^{5}U_M$
- $M_{Merc} \approx 3.30 \times 10^{23}\text{kg} \approx 5.52 \times 10^{-2}U_M$

The range between them is 7-orders apart, therefore a mass between them should be chosen in which both masses are encompassed with sufficient precision. Moreover, the chosen mass should be towards the lower end of this range, as the spacecraft implemented in the simulation should have its mass minimised for accurate fly-bys. Using Earth masses satisfies the first criteria, it provides a large mass on the rocket, albeit small enough to not significantly affect the orbit of the outer planets.

The final defined unit is the Unity time unit. The grand-tour missions, orbital time period of the planets and fly-bys of the outer-planets can last years, months and days respectively. Setting the simulation time unit to 1 day per each second running in the simulation is appropriate, as it provides enough time for users to experience and interact with the fly-bys while the orbits of the planets vary between minutes and multiple hours. Furthermore, this unit could potentially be adjusted to speed up the simulation during inter-planetary spaceflight, and decreased back to the default U_T value during the fly-by, as determined by the user.

Below is an example in converting between units, a standard celestial velocity converted from SI, to Unity units.

$$v = 3 \times 10^{4}\text{ms}^{-1}$$

Using table (6), the substitution of the SI units can be made with the respective Unity unit scale.

$$\text{m} = \frac{100U_L}{1.49 \times 10^{11}}$$

$$\text{s} = \frac{U_T}{8.64 \times 10^{4}}$$

Substituting these relations into the velocity will provide the Unity unit equivalent value.

$$v = 3 \times 10^{4}(\frac{100U_L}{1.49 \times 10^{11}})(\frac{U_T}{8.64 \times 10^{4}})^{-1}$$

$$= 1.74U_L U_T^{-1}$$

Quantities such as the velocity of the celestial objects, remain within the 7-digit restriction along with the standard solar-properties. The unity value for the gravitational constant G from equation (1) is critical in ensuring accurate simulation of the orbital mechanics of the celestial objects, therefore this value should have sufficient precision. The value for G in Unity units is

$$G = 8.81 \times 10^{-4}U^3 U_M^{-1} U_T^{-2}$$

Which provides G 4 significant figures to be defined when using the float numeric-type, therefore the units and their scaling remain adequate and provide enough precision for the numerical quantities and variables implemented in this simulation.

3 Method

Access to the entire code can be found in the following GitHub repository:

https://github.com/owen97779/Unity-

The font formatting is defined here:[1]

There are 3 main scripts that are used within this simulation:

[1]
- Scripts
- Classes, objects, attributes and components
- Functions and methods
- Lists
- Variables and strings

- Main.cs
- CelestialObject.cs
- InitialPrefab.cs

The Main script executes all the game functions required in the simulation. CelestialObject contains a public class called CelestialObject which contains all the methods and attributes associated with the game-objects in the simulation. InitialPrefab confirms that all the game-objects are loaded within the game, with the attributes of CelestialObject

3.1 Implementation of Physics

To begin with, a file containing the required initial conditions of the scene, including all the celestial objects must be created. This acts a primitive prefab system to store all the components of the game-objects; a prefab asset acts a template from which new prefabs can be easily created.

```
START
ISINITIALPREFAB
type = star
mass = 333000
name = Sun
position = x=0 y=0 z=0
radius = 15
END

START
ISINITIALPREFAB
type = planet
mass = 0.0553
name = Mecury
orbits = Sun
position = x =38.7 y=0 z=0
radius = 2
END
ENDTXT
```

Listing 1: Early version of the data.txt file that contains the properties of the Sun and Mercury

The mass and starting positions have been converted to the simulation Unity units with further properties such as the type of celestial object. Each object is enclosed by the **START** and **END** tags. The current positions in Listing (1) are based on the circular orbits and positions relative to the Sun in order to test Newtonian mechanics in the Unity engine, before implementing elliptical orbits. All orbits are confined to an orbital plane on the x-z axis.

Importing the data in this prefab file into the Unity engine requires a script that can read each line, target the specific string values and convert the values into attributes for each celestial object.

The Start function is called only once during the simulation and is initiated in the first frame, before the execution of the game logic or physics engine.

```
1  void Start()
2  {
3      InitialPrefabs = FindObjectsOfType<InitialPrefab>();
4      initializePrefabs();
5      foreach(InitialPrefab ip in InitialPrefabs)
6      {
7          CelestialObjects.Add(ip);
8      }
9      string[] dataOfEachLineFromFile = System.IO.File.ReadAllLines(@"data.txt");
10     readFileData(dataOfEachLineFromFile, 0, 0);
11 }
```

Listing 2: Start function that loads the properties defined in Listing (1)

Line 3 loops through all game-objects in the scene with the InitialPrefabs script attached and places them in a InitialPrefabs list, defined as a variable outside the Start function. Line 4 executes a function initialisePrefabs that assigns all of the game-objects with the Rigidbody component and disables the default gravity property so that Newtonian Gravitation can be implemented later, it also assigns each initial prefab into the parent class CelestialObject. Lines 5-8 adds all the initial prefabs, which are now part of the CelestialObject class, into a list called CelestialObjects. Lines 9-10 parses the text of the data file with the relevant tags, converts the text into a string and assigns the relevant attributes to each CelestialObject in the list. The intricacies of the readFileData function can be found in Main.

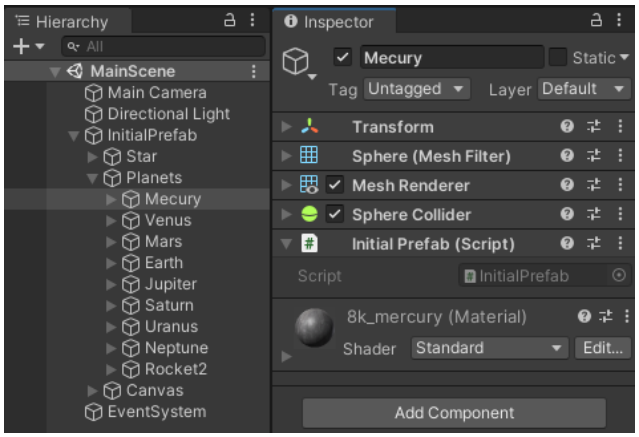

Figure 7: Game-objects with initial prefab script attached. The implementation of Newtonian Mechanics is made within the FixedUpdate function.

```
1  void calculateNetForce()
2  {
3      for (int i = 0; i < CelestialObjects.Count -1; i++)
4      {
5          for (int j = i+1; j < CelestialObjects.Count; j++)
6          {
7              Vector3 separationVector = CelestialObjects[j].getRigidbody().transform.position -
       CelestialObjects[i].getRigidbody().transform.position;
8              Vector3 Force = G*CelestialObjects[i].getRigidbody().mass*CelestialObjects[j].
       getRigidbody().mass * separationVector.normalized / (Mathf.Pow(separationVector.magnitude, 2));
9              CelestialObjects[i].setNetForce(Force);
10             CelestialObjects[j].setNetForce(Force * (-1));
11         }
12     }
13 }
```

Listing 3: Function that calculates the net force experienced for each planet

Using equation (1), the gravitational force can be calculated numerically for each **CelestialObject** in the **CelestialObjects** list. Two nested loops are used in order to remove redundant calculations of the gravitational force between each **CelestialObject**, thus saving unnecessary computation. The **CelestialObject** class has a method called **getRigidBody**, which is used to return the relevant **Rigidbody** attributes such as the **position** and **mass** of the object. Both the separation distance and gravitational force are vectors and are defined and calculated as such in lines 7-8. The computed gravitational force is then applied to both **CelestialObject** in the list with the method **setNetForce**, while the secondary **CelestialObject** is applied with a force equal in magnitude but opposite in direction, in accordance to Newton's third law.

Once the forces are calculated for each **CelestialObject**, they can be added as the force acting as part of the **Rigidbody** component.

```
1  void applyNewtonLawGravitation()
2  {
3      calculateNetForce();
4      foreach (CelestialObject co in CelestialObjects)
5      {
6          co.getRigidbody().AddForce(co.getNetForce());
7          co.clearNetForce();
8      }
9  }
```

Listing 4: Function that applies the calculated force onto the **Rigidbody** components.

Lines 4-7 loops through the entire **CelestialObjects** list and assigns the **Rigidbody** force from the one calculated and assigned to the **CelestialObject** class. This force attribute is then cleared from the class so that it can be calculated again in the next **FixedUpdate** instance.

After applying an initial velocity based on equation (3), circular orbits form.

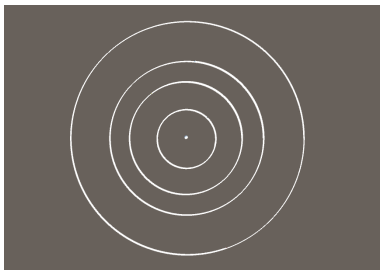

Figure 8: Circular orbit of the planets

Implementing elliptical orbits requires the solution of conic equations for the orbits of each of the planets. Using linear algebra and equation (11), this Python script can be used alongside the Horizons ephemeris[2] to provide 5 planetary positions starting from the launch of Voyager 2.

```
1  import numpy as np
2  def conic_section(p1, p2, p3, p4, p5):
3      x1, y1, x2, y2, x3, y3, x4, y4, x5, y5 =
       p1[0], p1[1], p2[0], p2[1], p3[0], p3[1],
       p4[0], p4[1], p5[0], p5[1]
4      a = np.array([[x1 * y1, y1 * y1, x1, y1,
       1],
5                    [x2 * y2, y2 * y2, x2, y2,
       1],
6                    [x3 * y3, y3 * y3, x3, y3,
       1],
7                    [x4 * y4, y4 * y4, x4, y4,
       1],
8                    [x5 * y5, y5 * y5, x5, y5,
       1]])
9      b = np.array([-x1 * x1, -x2 * x2, -x3 * x3
       , -x4 * x4, -x5 * x5])
10     x = np.linalg.solve(a, b)
11     B2, C2, D2, E2, F2 = x[0], x[1], x[2], x
       [3], x[4]
12     A2 = -(B2 * x1 * y1 + C2 * y1 * y1 + D2 *
       x1 + E2 * y1 + F2) / (x1 * x1)
13     return A2, B2, C2, D2, E2, F2
```

Listing 5: Section of main.py file that calculates the coefficients of the conic equation.

After finding the constants for the conic equations, the calculation for the semi-major and semi-minor axes can be made using equations (6) and (7). These are then verified by Kepler's third law (9) with the known orbital period of the planets. Henceforth, the unit-vector velocity is found for each planet using equation (12) and a tangential line equation is found. These two quantities are then inserted in the data file as shown in Appendix A, to be loaded by the Unity engine. The developed python script provides all of these values[18].

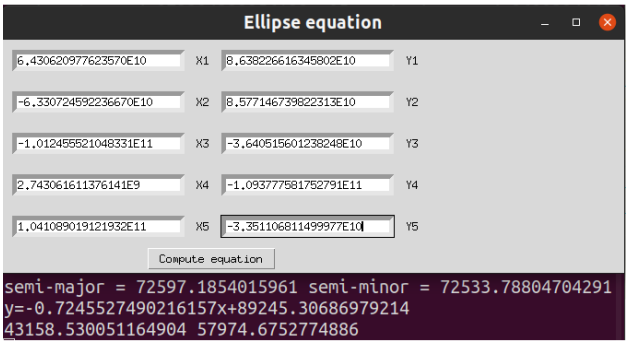

Figure 9: Front-end of main.py file that calculates a, b, tangential equation and (h,k).

The calculated values for a, b and $\hat{v}$ are implemented in the simulation as shown.

```
1  void setInitialVelocity()
2  {
3      calculateNetForce();
4      foreach (CelestialObject co in CelestialObjects)
5      {
6          if(!co.getSetVelocityStatus()  co.getOrbitTarget() != null)
7          {
8              Vector3 separationVector = co.getOrbitTarget().getRigidbody().transform.position - co.
       getRigidbody().transform.position;
9              float speed = Mathf.Sqrt(G*co.getOrbitTarget().getRigidbody().mass * (2/
       separationVector.magnitude - 1/co.getSemiMajorAxis()));
10             Vector3 velocityDirection = calculateVectorDirection(co);
11             Vector3 velocity = velocityDirection * speed;
12             co.getRigidbody().velocity = velocity;
13         }
14         co.getGameObject().SetActive(true);
15         co.getRigidbody().isKinematic = false;
16     }
17 }
```

Listing 6: Function that applies the initial Vis-viva velocity equation to all planets.

This function sets the initial velocities for each of the planets and is called at the end of the **Start** function in listing (2). To do this, the gravitational force for each planet is calculated in line 3, as shown in listing (3). Whilst looping through each planet in the **CelestialObjects** list, an **if** statement is made based on an updated parameters of the data file from Listing (1). These parameters include an initial velocity (as set to the spacecraft later on) and an orbit target attribute which can be included for other potential celestial objects such as planetary moons that also elliptically orbit planets as well as the Sun; as the planets do not have a predefined velocity from the data file (only a unit-vector), nor an orbit target other than the Sun, lines 8-15 are executed. Lines 8-9 are used to calculate the magnitude of the velocity using equation (10), while the direction is calculated using the **calculateVectorDirection** function in line 10. This function is important so that the planets all orbit anti-clockwise around the Sun (when viewed above the north pole of the Sun), setting this direction is dependant upon the starting positions of the planets, segmented by 4 quadrants on the Sun. The **Rigidbody** velocity is then assigned to each of the planets, while lines 14-15 ensure they are active in the simulation and are only affected by the physics scripts made in the simulation.

This completes the orbital mechanics of the planets in the solar-system, implementation of Voyager 2's grand-tour can now be made.

After creating a prefab for a rocket in the data file and setting the correct parameters based on table (2), the initial velocity of the rocket should be implemented within the simulation.

```
1 START
2 ISINITIALPREFAB
3 name = Rocket2
4 mass= 1e-7
5 radius = 1e-7
6 position = x =86.59662318 y=0 z=-53.75973066
7 velocity = x=1.005472645 y=0 z=2.012340433
8 END
```

Listing 7: Section of data file for simulated spacecraft.

As this initial velocity is already assigned in the data file, the function **readFileData** as called in listing (2), executes multiple functions in order to parse and apply this velocity into the **Rigidbody** component of the rocket.

The foundations for the physics have now been implemented, ideally the simulation runs with the outcomes expected from table (5.2).

3.2 Implementation of Visuals

The Unity engine provides easy implementation of visual elements to make the simulation look true to life. Textures of each of the planet are applied to the Unity scene editor, along with a skybox that models the milky-way.

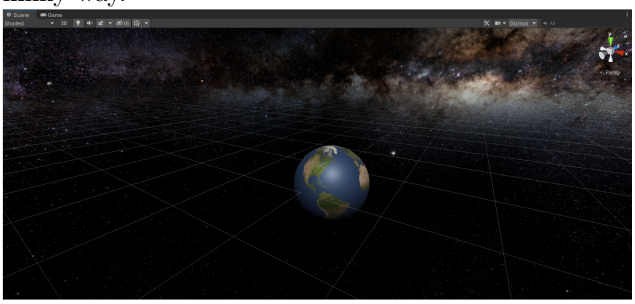

Figure 10: Textures applied to Unity scene.

The sizing of planets is a decision that requires balancing of both visual satisfaction and accuracy. Sizing all the celestial objects to scale would entail substantial empty space, with the planets being indiscernible as seen from the perspective of the spacecraft. Therefore, enlarging the inner-planets so that they are visible during the launch of Voyager 2 satisfies this aesthetic criteria, while having the outer-planets remain to scale so the fly-bys remain accurate. The final radii sizing can be found in appendix A.

Moreover, to aid in the visualisation of the movement of the planets, the Unity engine allows for trails for each game-object. The planets are provided with white trail while Voyager 2 is given a red trail.

The heads-up-display (HUD) of the simulation must provide the simulation speed that can be adjusted by the user, a date and time calendar, and a minimap of the current position of the rocket relative to other planets, so that the user can gauge the distance to the oncoming planet.

There are 2 methods of changing the simulation speed, changing the scaling of the Unity time unit U_T (with the adjustment of G and velocity of planets), or using the Unity engine to change the timescale of the simulation. Analysis of both methods will be made in the discussion 5.2, nevertheless the choice to adjust the timescale of the engine is implemented.

```
1 public void hasSpeedChanged(float multiplier)
2 {
3     Time.timeScale = multiplier;
4     simulationSpeed.text = multiplier.ToString
      () + " X" ;
5 }
```

Listing 8: Function that changes the simulation speed.

This function is assigned to a UI slider that parses through a number multiplier to adjust the simulation speed. This simulation speed is adjusted by the **Time.timeScale** float, when this value is set to 1, the Unity engine operates at its normal speed, increasing this value increases the number of executions of the Unity life-cycle as shown in figure (5). The maximum multiplier is set to (x30), the point before the frame-rate decreases below 60 frames-per-second on the development hardware. The multiplier value is then outputted to a string on the HUD in line 4.

Based on the U_T scaling and the 50 **FixedUpdate** executions per U_T, a date system can be implemented with the following code within **FixedUpdate**.

```
1  if(updateFixedUpdateCountPerSecond >=
       unitySecond * Time.timeScale)
2  {
3      day++;
4      //If the current month is Feb and a leap
       year
5      if(month == 2  leapYear == 4)
6      {
7          if(day > 29)
8          {
9              day = 1;
10             month++;
11             leapYear = 0;
12         }
13     }
14     //If the current month is Feb
15     else if(month == 2)
16     {
17         if(day > 28)
18         {
19             day = 1;
20             month++;
21         }
22     }
23     //If the current day is the last day of
       the month
24     else if(day > maxDay || (month == 4  day >
        maxDay - 1)|| (month == 6  day > maxDay -
        1)|| (month == 9  day > maxDay - 1)|| (
       month == 11  day > maxDay - 1))
25     {
26         day = 1;
27         month++;
28         //If it is the 31st of Dec
29         if(month > maxMonth)
30         {
31             month = 1;
32             year++;
33             leapYear++;
34         }
35     }
36     setDateTimeString();
37     updateFixedUpdateCountPerSecond =0;
38 }
39 updateFixedUpdateCountPerSecond += Time.
       timeScale * Time.fixedDeltaTime;
```

Listing 9: Function that changes the date dependant on simulation run-time.

The **if** statement has a condition that executes if the elapsed time is currently greater than the scaling of U_T, defined as 1, and scaled by **Time.timeScale** if the user changed the simulation speed. At the end of the script in line 39, the counter increases after each **FixedUpdate** execution. The logic between lines 2-38 simply increases the days, months and years dependant on the parameters of the Gregorian calendar, such as the months with 28, 29, 30 and 31 days while adhering to leap years as each U_T passes. Based on the default values, each second passing in real-time causes 1 day to pass in the simulation. This is outputted to **String** values on the HUD, which starts from the launch of Voyager 2 on the 20th August, 1977.

A minimap can be implemented by using a fixed camera on the rocket at a specified height in the Unity scene editor and converting the output into a texture to be displayed on the HUD.

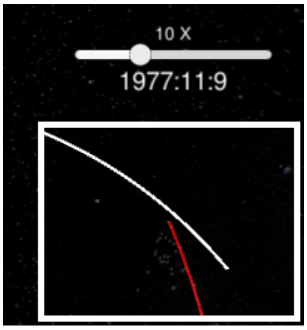

Figure 11: Simulation HUD.

3.3 Implementation of VR

The Unity engine requires installation of the XR package in order to implement VR functionality[6]. This package contains multiple scripts that offer left and right hand control, headset camera control as well as a VR HUD. The VR headset in use is the Oculus Rift.

Two cameras must be implemented in the simulation, a free-cam that provides camera movement control and total view of the solar-system, as well as a first-person camera of the Voyager 2 rocket.

The free-cam is implemented within the Unity scene editor with a script that controls the movement on a keyboard. It is initially placed above the north of the Sun's pole, within view of the inner planets, Jupiter and Saturn.

The Voyager 2 camera is a child object of the rocket prefab, which synchronises its position to the rocket as it moves. Implementing VR camera control requires assigning an XR Origin script that converts the Oculus headset tracking into movement of the camera gameobject.

Switching between the two cameras can be achieved using the input manager of the Unity engine. This is made within the **Update** function in the Unity engine as it is frame-rate dependant and apart of the gamelogic.

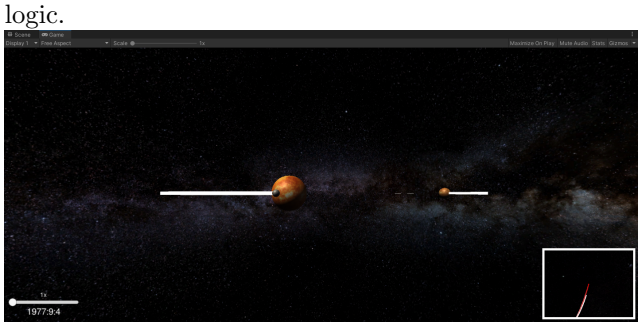

Figure 12: First-person camera of the spacecraft.

Interaction is made with the environment by allowing the Voyager 2 rocket to take images from the perspective of the camera under the control of the user. This is implemented within the **Update** function

```
if(Input.GetKeyDown(KeyCode.Space))
{
    ScreenCapture.CaptureScreenshot("
    Screenshot.png");
}
```

Listing 10: Simulation screenshot function that captures in-game photo when the space bar is pressed

4 Results

The final product and results of the simulation are provided in this section. All the data provided were exported directly.

4.1 Elliptical Orbits

The calculated eccentricities of the planets using the python script in listing (5) can be compared with the true values.

Planet	True e	Calc. e	Error (%)
Mercury	0.206	0.228	11
Venus	0.007	0.041	490
Earth	0.017	0.029	71
Mars	0.094	0.103	10
Jupiter	0.049	0.036	36
Saturn	0.052	0.060	15
Uranus	0.047	0.045	4
Neptune	0.010	0.040	300

Table 7: Comparison of the eccentricity between calculated and real values[2]

Accuracy of the orbits can be quantified and compared to the real positions, with simulations taken 1 month, 1 year and 10 years into the future from 1977.

Positions After 1 Month (19/09/1977)			
Planet	Real Pos. (U_L)	Sim. Pos. (U_L)	Error (%)
Mercury	x: 16.4 y: 26.2	x: 15.0 y: 26.8	9 2
Venus	x: -14.8 y: 70.1	x: -15.7 y: 70.0	6 0
Earth	x: 101 y: -5.90	x: 101 y: -5.65	0 4
Mars	x: 73.2 y: 131	x: 72.8 y: 131	1 0
Jupiter	x: 60.1 y: 508	x: 60.0 y: 508	0 0
Saturn	x: -715 y: 580	x: -715 y: 580	0 0
Uranus	x: -1.39×10^3 y: -1.25×10^3	x: -1.39×10^3 y: -1.25×10^3	0 0
Neptune	x: -750 y: -2.95×10^3	x: -750 y: -2.95×10^3	0 0

Table 8: Comparison of the positions of the planets after 1 month[2]

Positions After 1 Year (20/08/1978)			
Planet	Real Pos. (U_L)	Sim. Pos. (U_L)	Error (%)
Mercury	x: 35.2	x: 34.1	9
	y: −18.2	y: −20.5	13
Venus	x: 12.7	x: 10.3	8
	y: −72.4	y: −72.6	0
Earth	x: 86.2	x: 84.7	2
	y: −54.9	y: −56.7	3
Mars	x: −128	x: −128	0
	y: −94.6	y: −94.4	0
Jupiter	x: −192	x: −191	1
	y: 487	y: 488	0
Saturn	x: 836	x: 837	0
	y: 407	y: 406	0
Uranus	x: -1.30×10^3	x: -1.30×10^3	0
	y: -1.35×10^3	y: -1.35×10^3	0
Neptune	x: −648	x: −649	0
	y: -2.97×10^3	y: -2.97×10^3	0

Table 9: Comparison of the positions of the planets after 1 year[2]

Positions After 10 Years (20/08/1987)			
Planet	Real Pos.(U_L)	Sim. Pos.(U_L)	Error (%)
Mercury	x: −30.1	x: −29.5	2
	y: 18.3	y: 22.4	22
Venus	x: −59.8	x: −55.6	7
	y: 41.1	y: 50.2	22
Earth	x: 85.0	x: 75.7	11
	y: −54.6	y: −63.6	16
Mars	x: −145	x: −142	2
	y: 84.6	y: 92.7	10
Jupiter	x: 468	x: 463	1
	y: 166	y: 169	2
Saturn	x: 992	x: 994	0
	y: −171	y: −169	1
Uranus	x: −146	x: −151	3
	y: -1.92×10^3	y: -1.92×10^3	0
Neptune	x: 375	x: 372	1
	y: -3.01×10^3	y: -3.01×10^3	0

Table 10: Comparison of the positions of the planets after 10 years[2]

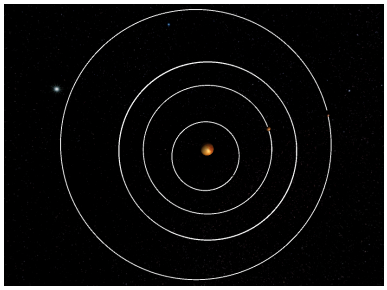

Figure 13: Footage of the elliptical orbits in the simulation: https://youtu.be/i45K7uRjq74

4.2 Grand-Tour

All the planets in the simulation are correctly initialised to the starting position of the 20th August 1977, with the exception of Saturn. In order to accommodate for the Voyager 2 course correction that occurred after Voyager 2's fly-by with Jupiter[5], the starting position of Saturn was adjusted in this simulation to the 18th April 1977, in order to align with the trajectory of the spacecraft.

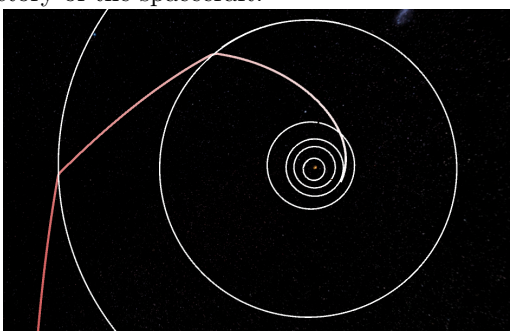

Figure 14: Footage of the grand-tour and fly-bys of Jupiter and Saturn: https://youtu.be/TwocJWuvaTg

Below is a table of the flyby properties in the simulation in comparison to the true values.

Planet	Real DoA	Sim. DoA	Real Flyby (U_L)	Sim. Flyby (U_L)
Jupiter	9/7/1979	27/4/1979	3.83×10^{-1}	3.44×10^{-1}
Saturn	26/8/1981	26/12/1980	6.78×10^{-2}	1.42×10^{-1}

Table 11: Simulated values for the closest approach to each planet and the date of arrival.

The angle of approach for both fly-bys can be calculated using equation (23).

Planet	Angle of Approach (°)
Jupiter	90.4
Saturn	73.5

Table 12: Angle of approach for each planet in the simulation

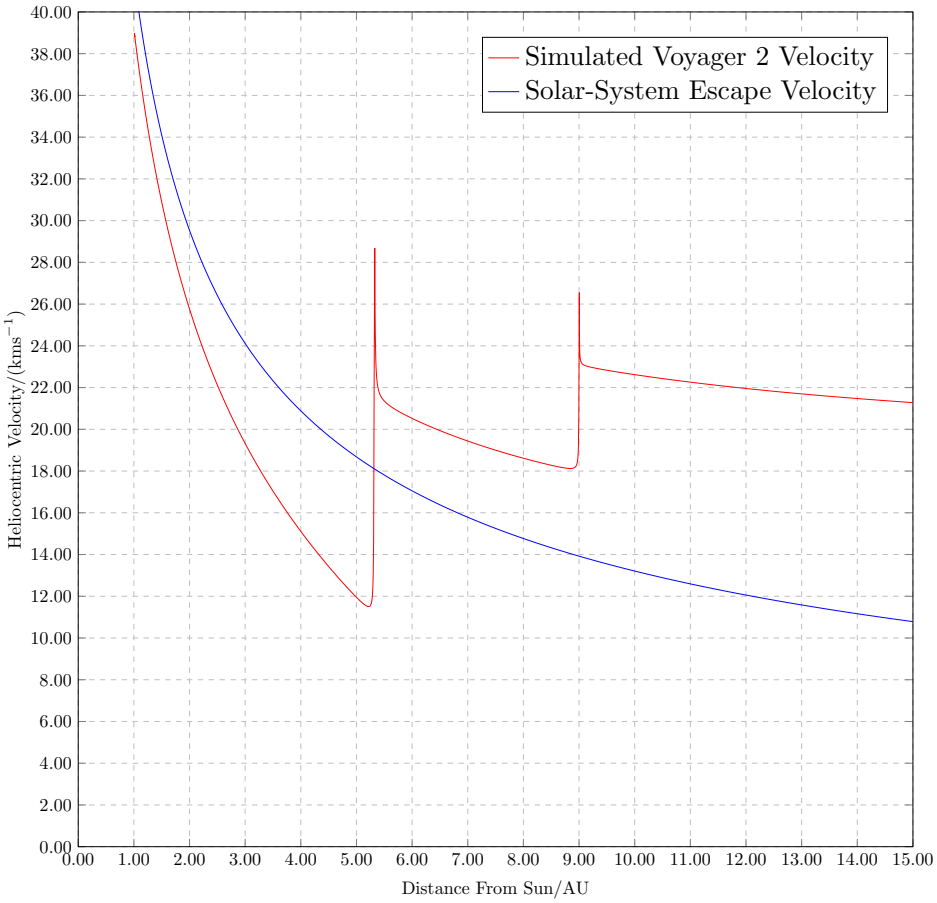

Figure 15: A graph of the simulated heliocentric velocity of the spacecraft against its distance from the Sun along with the solar-escape velocity

5 Discussion

The accuracy of the simulation can now be evaluated based on the Horizons ephemeris[2]. Other potential methods on achieving the aims of the simulation will be discussed and evaluated against the chosen approaches made.

5.1 Accuracy of Elliptical Orbits

From table (7), there is a large range in the percentage error for the eccentricity between the planets in the solar-system. Both Venus and Neptune had the highest errors in proportion to their true eccentricity whereas Mars and Uranus were the most accurate. Most of the error is due to the selection of the 5 points to determine the conic equation for each planet. Rather than spacing the 5 points between the starting date of August 20th 1977 to the orbital period of the planet, points were selected at random throughout the entire available timeline of the data-set. This potentially explains the anomalous eccentricities of Venus and range of percentage error across the planets, as some of the chosen positions may have been multiple years after 1977 and potentially affected by gravitational perturbations from other celestial objects. Moreover, the omission of the z-axis positions could also provide some error, most significantly in Mercury and Venus as they have the highest orbital plane tilt in comparison to the invariant plane of the Sun, at 6.3° and 2.2° respectively[15]. Henceforth, floating-point rounding errors can also be attributed to the calculations made within the python script in listing (5), as the accuracy of the script is limited by the precision of float variables.

To improve the accuracy of the elliptical orbits in future, an average eccentricity must be taken with multiple groups of 5 points that are spaced evenly with regards to the orbital period of each planet. Fortunately, the orbits of Venus and Neptune are somewhat inconsequential on the current build of the grand-tour. This is because the other planets, particularly Jupiter and Saturn, require the most accuracy in their orbital mechanics as they contribute most to the planetary gravitational forces due to their masses and proximity to Voyager 2 during launch, as well as participating in the grand-tour fly-bys.

Nevertheless, the simulation of the orbits between a month, year and 10 years show that the percentage error on the relative positions of the planets compound over time. Moreover, a 10 year simulation reached errors greater than 10% on the inner-planets. This is due to the inaccuracies of the conic equation calculations, as well as other factors such as floating-point errors as the simulation progresses, significant with the 7 digit precision of floats in C#. Moreover, the outer-planets have the lowest errors, which could be due to the lim-

itations of Newton's Law of Gravitation for stronger gravitational fields, such as the relatively high, but expected, error on Mercury. Despite not modelling other celestial objects in the solar-system that may perturb the gravitational forces of the planets, the accuracy of the elliptical orbits are sufficient for a grand-tour of a solar-system lasting less than 10 years.

5.2 Accuracy of Voyager 2 Fly-bys

Based on the fly-by properties of the spacecraft simulated on Jupiter, the fly-by itself was quite accurate with only a 10% percentage error. Some of the error could be due to the omission of the Jovian moons that also provide gravitational forces onto the spacecraft. Moreover, the date of arrival for the closest approach of Jupiter was made 115 days earlier than anticipated, with the initial conditions from table (4). The reasoning remains unclear, however it can be resolved by manually decreasing the magnitude of the velocity to align the trajectories for an accurate arrival. This was not implemented in the final build so that the initial conditions of the spacecraft remain true to Voyager 2.

Furthermore, changing the initial position of Saturn was required in order to align it with the trajectory of the spacecraft after its fly-by with Jupiter. This is a simpler method than implementing a course correction as was done with Voyager 2[5]. Despite this, the closest approach to Saturn without collision provides a fly-by distance just slightly more than double the true value, as shown in table (4). Choosing a starting position a few hours later would decrease the fly-by distance without collision, but as the minimum resolution of the simulation is defined as $U_T = 1$ day, this would slightly increase the inaccuracy of the measured elliptical orbit as a systematic error is introduced. Decreasing U_T would increase the resolution and possible positions to choose from, thereby allowing a more accurate fly-by approach distance.

During the testing of the simulation, a method to increase the simulation speed was made by changing the U_T and variables that included it as a unit, significantly the gravitational constant G. This had the effect of changing the trajectory of the spacecraft as it entered a fly-by whilst the simulation speed was being adjusted, thereby no 2 simulations were the same. Increasing the U_T increases the gravitational field strength of the planets, in which Newtonian gravitation is less accurate. The implemented method of changing **Time.timeScale** speeds up the simulation without any change in accuracy.

Nevertheless, comparison between the velocity of the spacecraft and Voyager 2 can be made with figures (4) and (15). Figure (15) is extremely accurate to the velocity of Voyager 2, until it reaches the Saturn fly-by at around 9 AU away from the Sun. The gravity assisted-velocity gain was less than expected, due to the larger fly-by distance and large approach angle, which reduces the final velocity gain as shown in figure (3).

5.3 Potential Improvements

Potential to visit all of the outer-planets is desirable in order to have a more accurate simulation of Voyager 2's grand-tour. Therefore, study of the flight-plan must be made with the understanding and implementation of the parameters of the course corrections it undertook. This excludes the need to adjust the initial positions of the outer-planets so that they align with the trajectory of the spacecraft.

Moreover, the fundamentals of the simulation have been designed in mind with the possibility of including other celestial objects into the simulation, particularly moons, as they are heavily featured within the flight-plan of Voyager 2[14]. The use of the prefab system and data file makes this process much more efficient.

Using table (1), analysis of the VR experience can be made. Although the user can interact with the environment by taking pictures during the fly-by, more VR interaction could be implemented to increase immersion. Particularly use of the VR tracking controllers can be made, by using them for pointing to celestial objects and provide an information graphic containing all the parameters of the **Rigidbody** on the UI. This would aid in improving the knowledge of the attributes of the motion of celestial objects.

6 Conclusion

To conclude the end of this report, an evaluation of the success of the simulation is made against the aims of the project, namely to create an accurate simulation of the grand-tour of Voyager 2, along with a feature-rich VR experience so that it can be an effective learning tool for science communication.

The implementation of Newtonian gravitation and Kepler's laws remains largely accurate based on the simulation findings in tables (8), (9) and (10). It can be inferred that the Unity engine is a sufficient platform to model the planetary orbital mechanics, with the choice to use the float numeric-type appropriate due to the overall accuracy and performance on consumer-level hardware. Moreover, it also showcases the accuracy of Newtonian mechanics for objects under solar-system like gravitational fields, therefore it is an appropriate basis for the physics of a grand-tour simulation.

Despite the simplifications made to the initial conditions of the spacecraft, an accurate fly-by was achieved with Jupiter, albeit at an earlier date of arrival. The conditions for gravity-assist, such as the angle of approach and initial velocity, were true to the values of Voyager 2, as the kinetic energy gained by the spacecraft is almost identical, when compared with figures (4) and (15). As Voyager 2 had made multiple mid-course corrections to its orbit[5], implementing these corrections within the simulation would allow for an accurate fly-by and gravity-assist for Saturn, thereby allowing the potential for accurate fly-bys with Uranus and Neptune in the simulation. There is also potential to expand the types of celestial objects present in

the simulation, such as the Jovian moons in which they are visible during the fly-by of Jupiter, along with data available on the closest approaches.

Nevertheless, the simulation contains all of the design frameworks required for a successful VR experience to varying degrees. The simulation details the scene as the solar-system with life-like visuals, the user can observe the environment with multiple cameras and receive immediate feedback on the trajectory of the grand-tour. Some interaction is present in the form of photographic data collection, however more can be implemented to utilise the controllers of the Oculus Rift, such as a information graphics as the user points to a celestial object.

Overall, the simulation satisfies the aims of the project and produces a functioning VR experience that stimulates the curiosity of the capabilities of astrophysics. Along with the potential to develop it further, the simulation enhances the understanding of concepts such as gravity-assist and more mathematically abstract orbital mechanics through visual representation, and can therefore be utilised as a powerful tool for education.

Public Summary

With the advent of the 21st century, the popularity of space exploration has grown, with the drafting of missions to Mars and beyond. These interplanetary distances are incredibly large and with the limitations on fuel capacity, concepts such as gravity-assist provide enormous energy savings for the spacecraft to reach its destination. It functions by using the gravitational pull of intermediary planets along its trajectory, as a result, it is dragged slightly in the direction of the planet's motion, gaining velocity.

The Voyager program utilised gravity-assist in its grand-tour of the solar-system, visiting the 4 outer-planets[14]. This paper implements the Voyager 2 grand-tour into virtual-reality, comparing its accuracy to the actual orbit and analyses its VR features. The physics behind the planetary orbits was implemented using the Unity engine, a development platform that provides structured documentation[6] on applying the physics computationally, as well as detailed visuals and VR functionality.

The resulting product is a user-interactive grand-tour of Jupiter and Saturn, with textures and camera control to view an authentic simulation of the orbital mechanics of the solar-system.

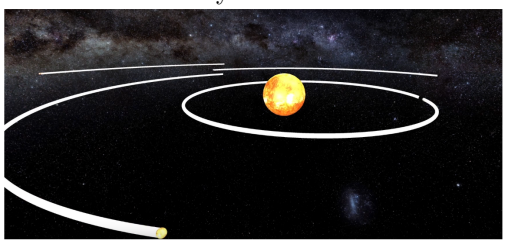

Figure 16: VR trailer for a grand-tour: https://youtu.be/F32aUpgwAY4

Analysing the data of the simulated orbits of the planets, a relatively high degree of accuracy is found in predicting the orbits within a short timescale (10 years), suitable for implementing a grand-tour.

Furthermore, the grand-tour was implemented with accurate results for the fly-by of Jupiter. Although Voyager 2 had a course correction[5] which was not implemented in the simulation and therefore accurate fly-bys of the other planets were not be achieved.

Nevertheless, the use of VR has provided an immersive medium in which people could gain greater understanding of the fundamentals of interplanetary travel.

7 Appendix A: Data.txt

```
START
ISINITIALPREFAB
type = star
mass = 333000
name = Sun
position = x=0 y=0 z=0
radius = 15
//Axial Tilt
obliquity = x=0 y=0 z=7.25
//Orbital rotation on axis
rotationVelocity = 0.2
END


START
ISINITIALPREFAB
type = planet
mass = 0.0553
name = Mecury
orbits = Sun
position = x =14.37993 y=0 z=-43.072355
radius = 2
//Unit vector direction
veldirection = x=0 y =0 z=-50.0988326232
semimajor=38.811732967455626
obliquity = x= 0 y= 0 z= 0.01
rotationVelocity = 0.0921
END

START
ISINITIALPREFAB
type = planet
mass = 0.815
name = Venus
orbits = Sun
position = x =43.1585300 y=0 z=57.97467
radius = 5
veldirection = x=0 y =0 z=89.35220774049697
semimajor=72.58260223037392
obliquity = x= 0 y= 0 z =177.3
rotationVelocity = 0.0222
END

START
ISINITIALPREFAB
type = planet
name = Earth
mass= 1
position = x =86.184195 y=0 z=-54.55507
radius = 0.0042758
orbits = Sun
veldirection = x=0 y =0 z=-194.93914455136257
semimajor=100.35996668226879
obliquity = x= 0 y=0 z=23.4392911
rotationVelocity = 5.415
END

START
ISINITIALPREFAB
type = planet
name = Mars
mass= 0.107
position = x =105.01198 y=0 z=102.11534
radius = 3
orbits = Sun
veldirection = x=0 y =0 z=228.81058791236458
semimajor=152.97591858343165
obliquity  = x=0 y=0 z=25.19
rotationVelocity = 5.262
END

START
ISINITIALPREFAB
type = planet
name = Jupiter
mass= 317.8
position = x =82.9066236372 y=0 z=504.
    080878892
radius = 0.04692014
orbits = Sun
veldirection = x=0 y =0 z=521.4006197011097
semimajor=521.6639772447088
obliquity = x= 0 y= 0 z=3.13
rotationVelocity = 13.057
END

START
ISINITIALPREFAB
type = planet
name = Saturn
mass= 95.2
//Position of Saturn moved to 18th April 1977
position = x =-666.2643840 y=0 z=632.1518855
radius = 0.039082
orbits = Sun
veldirection = x=0 y =0 z=1397.5067869005588
semimajor=956.9733015818334
obliquity = x=0 y=0 z=26.73
rotationVelocity = 12.162
END

START
ISINITIALPREFAB
type = planet
name = Uranus
mass= 14.5
position = x =-1397.860540 y=0 z=-1236.542164
radius = 9
orbits = Sun
veldirection = x=0 y =0 z=-2935.1860004491436
semimajor=1926.2576030015664
obliquity = x=0 y=0 z= 97.77
rotationVelocity = 7.517
END

START
ISINITIALPREFAB
type = planet
name = Neptune
mass= 17.1
position = x =-759.4243622 y=0 z=-2944.006695
radius = 9
orbits = Sun
veldirection = x=0 y =0 z=-3136.806328954471
semimajor=3018.076323127499
obliquity x= 0 y=0 z=28.32
rotationVelocity = 8.045
END

START
ISINITIALPREFAB
type = planet
name = Rocket2
mass= 1e-7
position = x =86.59662318 y=0 z=-53.75973066
radius = 1e-7
//Set initial velocity
velocity = x=1.00547264504 y=0 z=2.01234043386
END
ENDTXT
```

Listing 11: Final version of the data.txt file

Articles

[1] Jaziar Radianti et al. "A systematic review of immersive virtual reality applications for higher education: Design elements, lessons learned, and research agenda". In: *Computers Education* 147 (2020), p. 103778. ISSN: 0360-1315. DOI: https://doi.org/10.1016/j.compedu.2019.103778. URL: https://www.sciencedirect.com/science/article/pii/S0360131519303276.

[3] Gerhard Beutler. "Methods of Celestial Mechanics". In: *Methods of Celestial Mechanics: Volume I: Physical, Mathematical, and Numerical Principles, Astronomy and Astrophysics Library. ISBN 978-3-540-40749-2. Springer-Verlag Berlin Heidelberg, 2005* I (Jan. 2005). DOI: 10.1007/b137725.

[4] "Fast reconnaissance missions to the outer solar system utilizing energy derived from the gravitational field of Jupiter". In: 2 (1966), p. 9. URL: http://www.gravityassist.com/IAF3-2/Ref.%20203-143.pdf.

[7] George T Gillies. "The Newtonian gravitational constant: recent measurements and related studies". In: *Reports on Progress in Physics* 60.2 (Feb. 1997), pp. 151–225. DOI: 10.1088/0034-4885/60/2/001. URL: https://doi.org/10.1088/0034-4885/60/2/001.

[8] Meishu Lu et al. "Visualization of Kepler's laws of planetary motion". In: *Physics Education* 52.2 (Jan. 2017), p. 025006. DOI: 10.1088/1361-6552/aa539e. URL: https://doi.org/10.1088/1361-6552/aa539e.

[9] John H. Mathews. "The Five Point Conic Section: Exploration with Computer Software". In: *School Science and Mathematics* 95.4 (1995), pp. 206–208. DOI: https://doi.org/10.1111/j.1949-8594.1995.tb15764.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1949-8594.1995.tb15764.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1949-8594.1995.tb15764.x.

[11] Eckhard Matthias Sigurd Hitzer. "Conic sections through five points classical, projective, conformal". In: (Nov. 2003). URL: https://www.vixra.org/pdf/1306.0118v1.pdf.

[12] B Reed. "The single-mass gravitational slingshot". In: *European Journal of Physics* 35 (Apr. 2014), p. 045009. DOI: 10.1088/0143-0807/35/4/045009.

[14] R.P. Rudd, J.C. Hall, and G.L. Spradlin. "The voyager interstellar mission". In: *Acta Astronautica* 40.2 (1997). Enlarging The Scope of Space Applications, pp. 383–396. ISSN: 0094-5765. DOI: https://doi.org/10.1016/S0094-5765(97)00146-X. URL: https://www.sciencedirect.com/science/article/pii/S009457659700146X.

Web Sources

[2] NASA Jet-Propulsion Laboratory California. *NASA JPL Horizons*. URL: https://ssd.jpl.nasa.gov/horizons/. (accessed: 18.04.2022).

[5] NASA. *Voyager 2*. URL: https://solarsystem.nasa.gov/missions/voyager-2/in-depth/. (accessed: 18.04.2022).

[6] Unity Technologies. *Unity User Manual 2021.3 (LTS)*. URL: https://docs.unity3d.com/Manual/index.html. (accessed: 18.04.2022).

[10] *Ellipse General Equation*. URL: https://www.geek-share.com/detail/2607187045.html. (accessed: 18.04.2022).

[13] *Gravitational Slingshot*. URL: https://mathpages.com/home/kmath114/kmath114.htm. (accessed: 18.04.2022).

[15] David R. Williams. *Planetary Fact Sheet - Metric*. URL: https://nssdc.gsfc.nasa.gov/planetary/factsheet/. (accessed: 18.04.2022).

[16] NASA. *Basics of Space Flight*. URL: https://solarsystem.nasa.gov/basics/chapter4-1/. (accessed: 18.04.2022).

[17] Microsoft. *Floating-point numeric types (C reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types. (accessed: 18.04.2022).

[18] *Python determine a conic section through five points*. URL: https://www.bilibili.com/read/cv12840281?spm_id_from=333.999.0.0. (accessed: 18.04.2022).