UNIVERSE SIMULATOR DOCUMENTATION

BY SAMUEL JOHNSTON

May 2022

Table of Contents

Note: You can click on the text above to go to that section.

Iteration 1: Stage 1 - Define & Understand

Concept Statement & Iteration 1 Goals

The concept for this project is to create a realistic 3D simulator of the solar system; in other words, a *n-body simulation* of the solar system that can be adapted to different initial conditions. This will involve implementing astrophysic equations in order to simulate realistic movement of objects (planets, moons and satellites) within the program. To assist this process, the program will be developed around two central CASE tools: Unity for program development and GitHub for project file management.

In this first iteration, the main goal will be to create the fundamentals of the simulator — that is to implement all the planetary data and astrophysic equations into the program, and get it up and running. Due to the timely nature of the task, this first iteration will not include creating a user interface. However, depending on how much time remains, it may be included.

Defining and Understanding the 'Problem'

The 'problem' this project will attempt to address is the *n-body problem* — that is the orbital mechanics of celestial objects undergoing gravitational forces. For this project, these celestial objects will be the planets and moons in our solar system (or at least parameters close to our solar system).

Extending on this, we need to consider the data of the problem, which in this case is the planetary data of our solar system. Here is the data tabulated below:

Solar System Planets	Mass (kg)	Radius (m)	Distance from Sun (m)	Orbit Eccentricity	Orbit Inclination from Ecliptic (°)	Axial Tilt to Orbit (°)	Sidereal Rotation Period (secs)
Sun	1.9890E+30	6.9640E+08	0	N/A	N/A	7.25	2164320
Mercury	3.3011E+23	2.4397E+06	5.79090E+10	0.20563	7.005	0.034	5067015
Venus	4.8675E+24	6.0518E+06	1.08209E+11	0.006772	3.39458	177.36	-20997152
Earth	5.9724E+24	6.3710E+06	1.49596E+11	0.0167086	0	23.439	86164.1
Moon	7.3477E+22	1.7374E+06	1.49980E+11	0.0549	5.145	6.687	2360591
Mars	6.4171E+23	3.3895E+06	2.27923E+11	0.0934	1.850	25.19	88643
Jupiter	1.8980E+27	6.9911E+07	7.78570E+11	0.0489	1.303	3.13	35730
Saturn	5.6830E+26	5.8232E+07	1.43353E+12	0.0565	2.485	26.73	38018
Uranus	8.6810E+25	2.5362E+07	2.87100E+12	0.04717	0.773	97.77	-62064
Neptune	1.0241E+26	2.4622E+07	4.47430E+12	0.008678	1.770	28.32	57996

Iteration 1: Stage 2 - Plan & Design

Understanding the Solution

Before we can plan a solution, we first need to understand what the solution entails. In the following 3 sections we'll take a deep dive into the maths and physics equations that are necessary to develop a solution.

---

1. Calculating Acceleration

Using Newtonian laws of gravitation and motion, it is possible to calculate the acceleration vector of an object undergoing gravitational forces. Newton's Law of Gravitation states that:

$$F = \frac{GMm}{r^2}$$

F = gravitational force (Newtons)
M = mass of other object (kg)
m = mass of this object (kg)
r = distance between the two objects (m)
G = gravitational constant (6.67×10^{-11} Nm^2kg^{-2})

As seen by the equation above, the gravitational force of two objects can be calculated. We can arrange this with Newton's Second Law of Motion (F = ma) and scalar multiply it with the object's direction vector (aka. unit vector) to calculate the acceleration of an object due to another object. We can then sum this with all other objects to calculate the acceleration vector of an object, given by the following equation:

$$\underbrace{\vec{a}_i}_{\text{Acceleration of object i}} = \underbrace{\sum_{\substack{k=1 \\ k \neq i}}^{n}}_{\substack{\text{Sum for all objects} \\ \text{except itself}}} \underbrace{\left(\frac{GM_k}{|\vec{r}_k|^3}\right)}_{\text{Magnitude}} \underbrace{\vec{r}_k}_{\text{Direction}}$$

a_i = acceleration vector of object i (ms^{-2})
n = number of objects
G = gravitational constant (6.67×10^{-11} Nm^2kg^{-2})
M_j = mass of other object j (kg)
r_j = position vector to other object j (m)

2. Implementing Motion using Acceleration

Using the acceleration in the section above, we can calculate the velocity and position of an object. However, in order to do so, we need to first select an algorithm of numerical integration. There are a number of algorithms— such as *SUVAT*, *Euler's*, *Runge-Kutta 4*, *Leapfrog*, *Verlet* — which all have their pros and cons for different scenarios.

For this project I decided to use Velocity Verlet, as it can be seen as a 'mid-point' of all the algorithms. It is a mix of both accuracy and complexity (more accurate but more complex than Euler's, less accurate but easier than Runge-Kutta 4— you can find more about them *here*).

Velocity Verlet calculates the position and velocity of an object 1 time step Δt ahead given their position, velocity and acceleration at time t. The algorithm repeats the following 3 steps for each object at each time step:

1) Calculate new position:

$$\vec{x}(t+\Delta t) = \vec{x}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)\Delta t^2$$

2) Derive new acceleration at new position:

$\vec{a}(t+\Delta t)$ can be calculated using the formula from the section above.

3) Calculate new velocity:

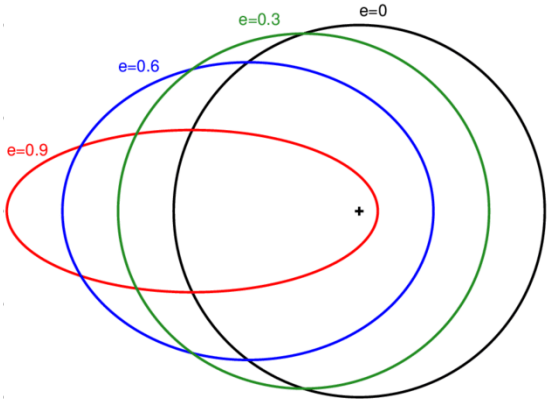
$$\vec{v}(t+\Delta t) = \vec{v}(t) + \frac{\vec{a}(t) + \vec{a}(t+\Delta t)}{2}\Delta t$$

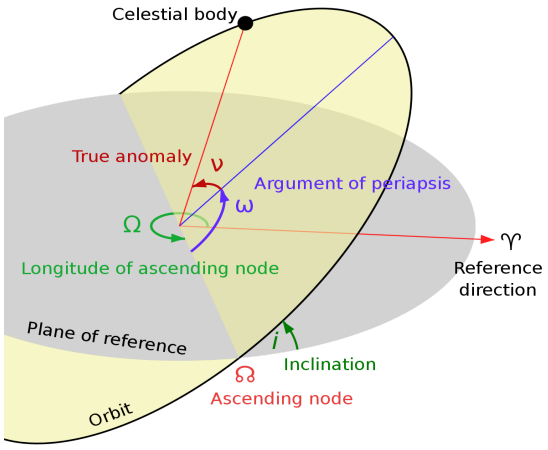
---

3. Calculating Initial Velocity for Elliptical Orbits

Before the simulation can run, it is required to calculate each planet's initial velocity. This is quite difficult and will depend on 2 main factors, their eccentricity and orbit inclination.

What is *eccentricity*? Eccentricity (or orbital eccentricity) is a numerical measure of the shape of an ellipse. In essence, it determines how much an orbit is elongated, with 0 being a perfect circle and higher values up to (but not including) 1 being more elongated. In simple terms, it determines how much the orbit is 'squashed'— as shown by the diagram to the right.





What is *orbital inclination*? As the name suggests, orbital inclination is the angle of tilt of an object's orbit. This can be quite hard to visualise, so I provided a diagram to illustrate it (ignore all the other measurements, we are only concerned with inclination — writing in dark green). Throughout this project, the plane of reference is the earth — sun orbit, also called the ecliptic.

Now onto the meatier part, calculating the initial velocity vector. This next part is quite challenging and will take quite a long time to fully explain — so I won't drive too deep into the full explanation, rather I will just provide the finalised equations for each major step.

Using the *vis-viva equation*, it is possible to determine the initial velocity of an object. The formula is as follows:

$$v_0 = \sqrt{GM\left(\frac{2}{r} - \frac{1}{a}\right)}$$

v_0 = initial velocity (ms^{-1})
G = gravitational constant ($6.67 \times 10^{-11}\ Nm^2kg^{-2}$)
M = mass of other object (kg)
r = distance between the two objects (m)
a = length of semi-major axis (m)

However, we cannot use this equation as it is now for a number of reasons. Firstly, we aren't using semi-major axis length, rather the eccentricity of the orbit. Secondly, the formula doesn't explicitly determine the direction of the velocity with orbital inclination. So, for these reasons we need to rearrange and add some components to this equation.

Firstly, we need to rearrange the equation to replace the semi-major axis length with the eccentricity of the orbit. Note, I will skip over the exact steps (which can be found *here*), but basically it results in the two following equations:

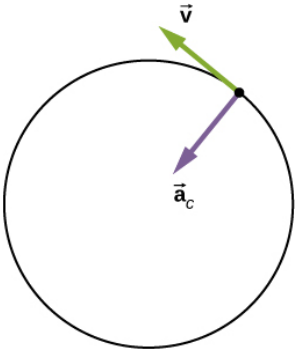
$$v_P = \sqrt{GM\left(\frac{1+e}{r_P}\right)}$$

v_P = initial velocity at perigee (ms^{-1})
G = gravitational constant ($6.67 \times 10^{-11}\ Nm^2kg^{-2}$)
M = mass of other object (kg)
e = eccentricity of orbit
r_P = distance from other object at perigee (m)

$$v_A = \sqrt{GM\left(\frac{1-e}{r_A}\right)}$$

v_A = initial velocity at apogee (ms^{-1})
G = gravitational constant ($6.67 \times 10^{-11}\ Nm^2kg^{-2}$)
M = mass of other object (kg)
e = eccentricity of orbit
r_A = distance from other object at apogee (m)

The ‘perigee’ is the point where the orbiting object is the closest to the other object, and the ‘apogee’ is the point where the orbiting object is the furthest from the other object. Initially, the distance r_P and r_A will be the same as it is the same point.



Now the next part is to determine the direction of the initial velocity. We know (from circular motion) that the velocity of an object will always be tangential to its orbital path — as shown in the diagram to the right. Therefore, the initial direction of the velocity will be perpendicular to the position vector of the two objects. However, we also need to account for the orbital inclination, which will alter the direction of the initial velocity.

To account for the orbital inclination, we can use the *cross product* of the position vector and an upward vector rotated by the inclination angle. Using this, we can get its unit vector which will give us the direction vector of the initial velocity.

After this, we can scalar multiply the two formulas above (v_P and v_A) by the direction vector to get the initial velocity vector formula. However, if we are calculating the initial velocity of moons and satellites, then we need to sum the results of all the orbited bodies (eg. for the moon, we need to sum the initial velocity of the moon-earth and add it with the initial velocity of the earth-sun). Thus, we can write the following equations:

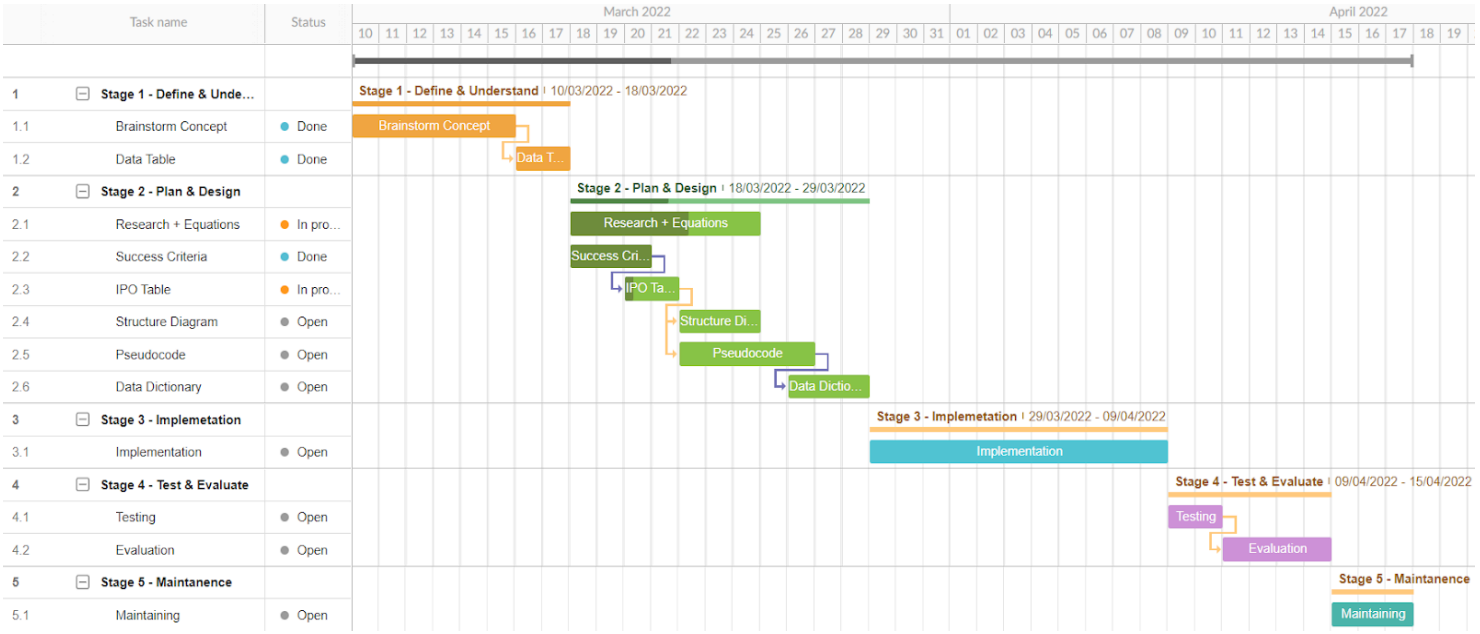
$$\underbrace{\vec{v}_A}_{\text{Initial Velocity at Apogee}} = \underbrace{\sum_{k=1}^{N}}_{\text{Sum for all orbited bodies}} \left(\underbrace{\sqrt{GM_k \left(\frac{1 - e_{k-1}}{|\vec{r}_k|} \right)}}_{\text{Magnitude}} \cdot \underbrace{\left(\frac{\vec{r}_k}{|\vec{r}_k|} \times \frac{\vec{z}_k}{|\vec{z}_k|} \right)}_{\text{Direction Vector}} \right)$$

$$\underbrace{\vec{v}_P}_{\text{Initial Velocity at Perigee}} = \underbrace{\sum_{k=1}^{N}}_{\text{Sum for all orbited bodies}} \left(\underbrace{\sqrt{GM_k \left(\frac{1 + e_{k-1}}{|\vec{r}_k|} \right)}}_{\text{Magnitude}} \cdot \underbrace{\left(\frac{\vec{r}_k}{|\vec{r}_k|} \times \frac{\vec{z}_k}{|\vec{z}_k|} \right)}_{\text{Direction Vector}} \right)$$

v_A = initial velocity vector at apogee (ms^{-1})
v_P = initial velocity vector at perigee (ms^{-1})
N = number of orbited bodies (1 for planets, 2 for moons and satellites)
G = gravitational constant (6.67×10^{-11} Nm^2kg^{-2})
M_k = mass of other object k (kg)
e_{k-1} = eccentricity of orbiting object
r_k = position vector from current object to other object k
z_k = upward vector rotated by inclination angle of object k

Gantt Chart

Here is the Gantt chart for this first iteration (made using a free trial from *GanttPRO*).



Success Criteria

For this first iteration to be considered successful, the project should meet the following criteria:

1. All planets should be included in the program.
2. All planets should be able to orbit around the sun.
3. The moon should also be able to orbit around the earth.
4. All planets should be correctly rotated by their axial tilt angle.
5. All planets should rotate correctly according to their respective rotation period.
6. The orbital paths of each planet should be visible and correctly rotated by their respective inclination angle.

IPO Table

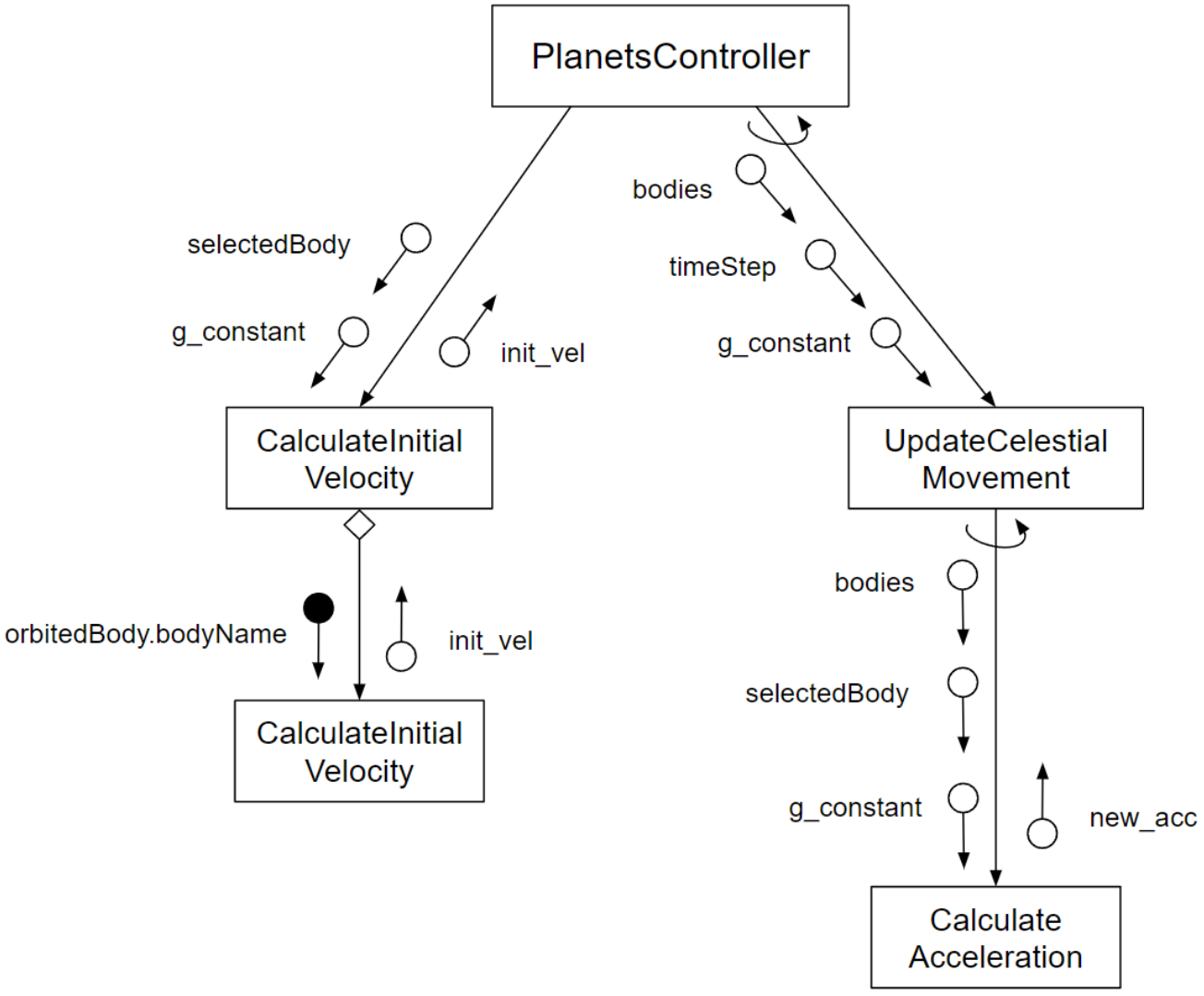
The table below shows a very simple high-level overview of the program's inputs, processes and outputs. There will be only very minimal input from the user, as most of the inputs are from the data itself (see table in stage 1).

Input	Process	Output
Selected Planet Gravitational Constant Orbited Object Inclination Angle Eccentricity	Calculate Initial Velocity of Selected Planet	Selected Planet's Initial Velocity
Planets Selected Planet Gravitational Constant	Update All Planet Movement Data	Display Planet Movement

---

Structure Diagram

The following structure diagram shows the main components of the planetsContoller Script — a script intended to control all the planets within the simulator.



Pseudocode

The following pseudocode will address the 'PlanetsController' Script and the algorithms that encompass it: calculating acceleration, updating each body's movement and calculating initial velocity. I've also included comments in red to outline what's happening at each part.

Note from Stage 3: I had to rewrite the 'UpdateCelestialMovement' algorithm as my initial pseudocode was actually wrong — so I decided to come back here and add the corrected version. More about this in the journal.

PlanetsController

```
BEING PlanetsController ()
   Set g_constant as 6.67e-11
   Set elapsedTime as 0
   Get bodies, timeStep
   // Calculate Initial Velocity for all planets
   FOR i = 0 to bodies.Length-1 STEP 1
      IF bodies[i].bodyName != "Sun" THEN
         bodies[i].velocity = CalculateInitialVelocity(bodies[i], g_constant)
      END IF
      // Calculate Acceleration at t=0
      bodies[i].acceleration = CalculateAcceleration(bodies, bodies[i], g_constant)
      // Set axial tilt
      Rotate bodies[i] by bodies[i].axialTilt
   END FOR
   // Continue to move planets
   REPEAT
      UpdateCelestialMovement(bodies, timeStep, g_constant)
      Increment elapsedTime by how much time has passed since last iteration
      FOR i = 0 to bodies.Length-1 STEP 1
         Move bodies[i] to bodies[i].position
         // Update each planet's rotation
         Rotate bodies[i] using bodies[i].rotationPeriod and time since last iteration
      END FOR
   UNTIL never
END PlanetsController
```

UpdateCelestialMovement (INITIAL)

```
BEGIN UpdateCelestialMovement (bodies, timeStep, g_constant)
    FOR i = 0 to bodies.Length-1 STEP 1
        Set acc = bodies[i].acceleration
        Set vel = bodies[i].velocity
        Set pos = bodies[i].position
        // Step 1: Calculate x(t + dt)
        bodies[i].position = pos + (vel * timeStep) + (0.5 * acc * (timeStep^2))
        // Step 2: Derive a(t + dt)
        Set new_acc = CalculateAcceleration(bodies, bodies[i], g_constant)
        bodies[i].acceleration = new_acc
        // Step 3: Calculate v(t + dt)
        bodies[i].velocity = vel + (0.5 * (acc + new_acc) * timeStep)
    END FOR
END UpdateCelestialMovement
```

UpdateCelestialMovement (REVISED)

```
BEGIN UpdateCelestialMovement (bodies, timeStep, g_constant)
   // Step 1: Calculate x(t + dt)
   FOR i = 0 to bodies.Length-1 STEP 1
      bodies[i].position += (bodies[i].velocity * timeStep) + (0.5 * bodies[i].acceleration *
(timeStep^2))
   END FOR
   // Step 2: Derive a(t + dt)
   Set old_acc as empty array of size bodies.Length
   FOR i = 0 to bodies.Length-1 STEP 1
      old_acc[i] = bodies[i].acceleration
      bodies[i].acceleration = CalculateAcceleration(bodies, bodies[i], g_constant)
   END FOR
   // Step 3: Calculate v(t + dt)
   FOR i = 0 to bodies.Length-1 STEP 1
      bodies[i].velocity += 0.5 * (old_acc[i] + bodies[i].acceleration) * timeStep
   END FOR
END UpdateCelestialMovement
```

CalcuateAccleration

```
BEING CalculateAcceleration (bodies, selectedBody, g_constant)
   Set new_acc as 0 vector
   FOREACH otherBody in bodies
      IF otherBody != selectedBody THEN
         // Use Newton's Law of Gravitation
         positionVector = otherBody.position - selectedBody.position
         new_acc += ((G * otherBody.mass) / (positionVector.magnitude^3)) * positionVector
      END IF
   END FOREACH
   Return new_acc
END CalculateAcceleration
```

CalculateInitialVelocity

```
BEGIN CalculateInitialVelocity (selectedBody, g_constant)
   Set init_vel as 0 vector
   // Calculate initial velocity relative to orbited body
   Get orbitedBody, inclinationAngle, eccentricity
   positionVector = orbitedBody.position - selectedBody.position
   Set inclinationVector as upwards vector rotated by inclinationAngle
   Set directionVector as the cross product of positionVector and inclinationVector
   Normalise direction vector
   init_vel = Sqrt(g_contant * orbitedBody.mass * ((1f + eccentricity) /
positionVector.magnitude)) * directionVector
   // Add initial velocity of orbited body (ignore if orbited body is the sun)
   IF orbitedBody.bodyName != "Sun" THEN
      init_vel += CalculateInitialVelocity (orbitedBody, g_constant)
   END IF
   Return init_vel
END CalculateInitialVelocity
```

Data Dictionary

Variable Name	Type	Size	Description
bodies	Array of CelestialBody Classes	[variable depending on number of planets]	Array holding all the data for each planet.
bodies[i].Length	Class Property; Int	32 Bits / 4 Bytes	Returns the length of the bodies array.
bodies[i].bodyName	Class Attribute; String	[variable depending on size of string]	Name of the planet. Eg. “Earth”
bodies[i].mass	Class Attribute; Float	32 Bits / 4 Bytes	Mass of the planet in kg. Eg. 5.9724E+24
bodies[i].radius	Class Attribute; Float	32 Bits / 4 Bytes	Radius of the planet in kg. Eg. 6.3710E+06
bodies[i].position	Class Attribute; Vector3	96 Bits / 12 Bytes	Position of the planet in m. Eg. (1.49596E+11, 0, 0)
bodies[i].velocity	Class Attribute; Vector3	96 Bits / 12 Bytes	Velocity of the planet in ms^{-1}. Eg. (0, 0, 30036.12)
bodies[i].acceleration	Class Attribute; Vector3	96 Bits / 12 Bytes	Acceleration of the planet in ms^{-2}. Eg. (-0.00589, 0, 0)
bodies[i].axialTilt	Class Attribute; Float	32 Bits / 4 Bytes	Axial tilt of the planet in degrees. Eg. 23.439
bodies[i].rotationPeriod	Class Attribute; Float	32 Bits / 4 Bytes	Rotation period of the planet in secs (can be negative). Eg. 86164.1
bodies[i].inclinationAngle	Class Attribute; Float	32 Bits / 4 Bytes	Orbital inclination angle in degrees from ecliptic (can be negative). Eg. 3.39458
bodies[i].eccentricity	Class Attribute; Float	32 Bits / 4 Bytes	Orbital eccentricity of the planet (must be >0 and <1). Eg. 0.0167086
bodies[i].orbitedBody	Class Attribute; Instance of CelestialBody Class	—	Immediate orbited body of the planet. For example, the Earth would be the immediate orbited body of the moon.
g_constant	Float	32 Bits / 4 Bytes	Gravitational Constant = 6.67384E-11.
elaspedTime	Float	32 Bits / 4 Bytes	How many seconds elapsed within the simulator.
old_acc	Array; Float	Number of planets * 4 bytes	Array holding the previous time step acceleration of all planets.

Iteration 1: Stage 3 - Implementation

Program Files

Google Drive ZIP File - Iteration 1 Project Files
https://drive.google.com/file/d/15PcaHL__sZDctfgKVEdE9A7lnO9omaN8/view?usp=sharing

GitHub ZIP File - Iteration 1 Project Files
https://github.com/SurferSamuel/2022-Major-Project-Files-Iteration_1

Please see the two links above to download the project files.

Note: Project files were created using Unity version 2021.2.17f1

Since the project doesn't currently have a way for the user to change the variables within the simulator, I decided against building the project (ie. exporting the project) — there's simply no need.

Here are the scripts I have developed (you can click each one to view it's file):

- CelestialBody.cs
- OrbitDisplay.cs
- PlanetsController.cs

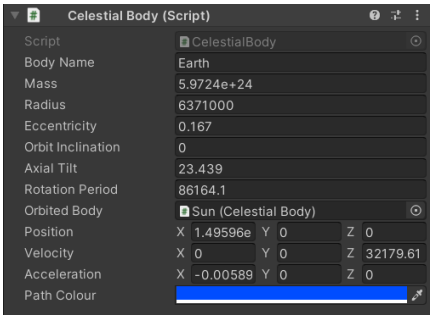
Iteration 1: Stage 4 - Test & Evaluate

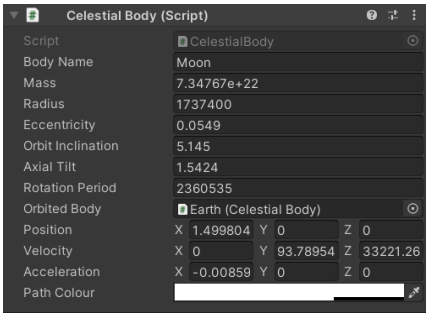
As referenced in stage 2, for the project to be deemed successful the program should meet all aspects of the success criteria. Hence (as demonstrated below), I can confidently say that this iteration has hit its targets; although the program as a whole still has quite a long way before it is finished.

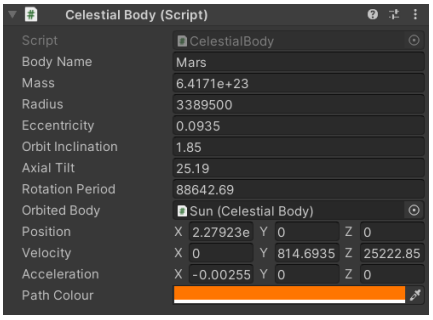
The main focus of this testing stage is to check and ensure the program meets all its requirements (as per the success criteria on stage 2). Here is the synopsis:

1. **All planets should be included in the program**

Yes, all the planets have been implemented into the program. Here are some of them:

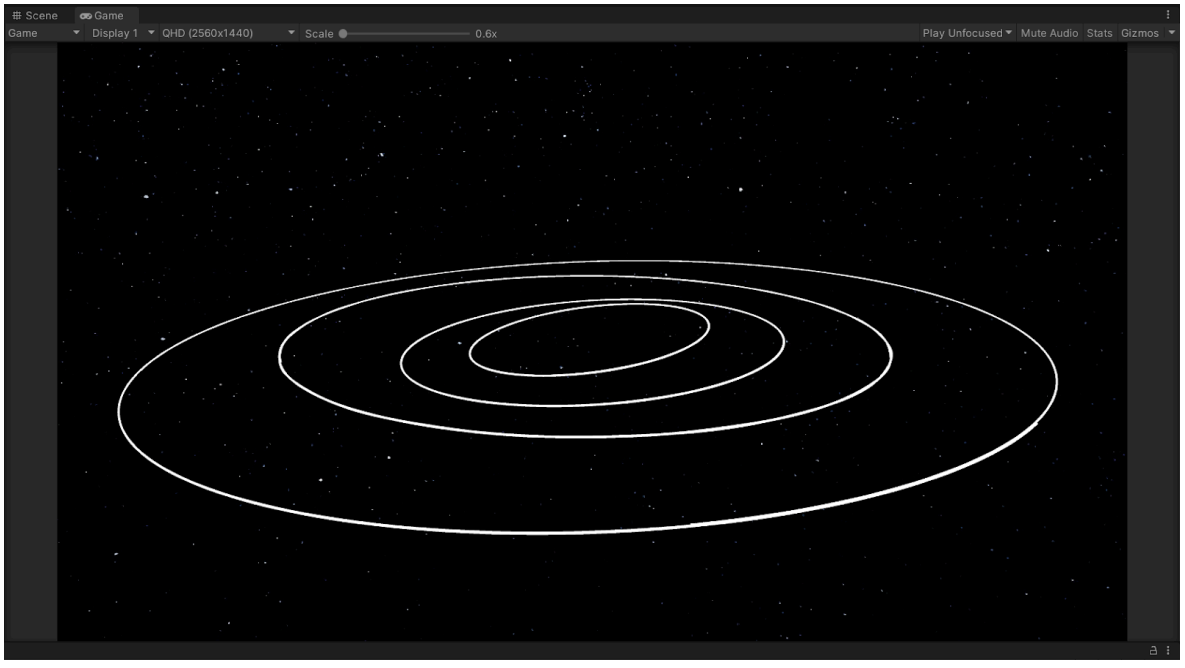






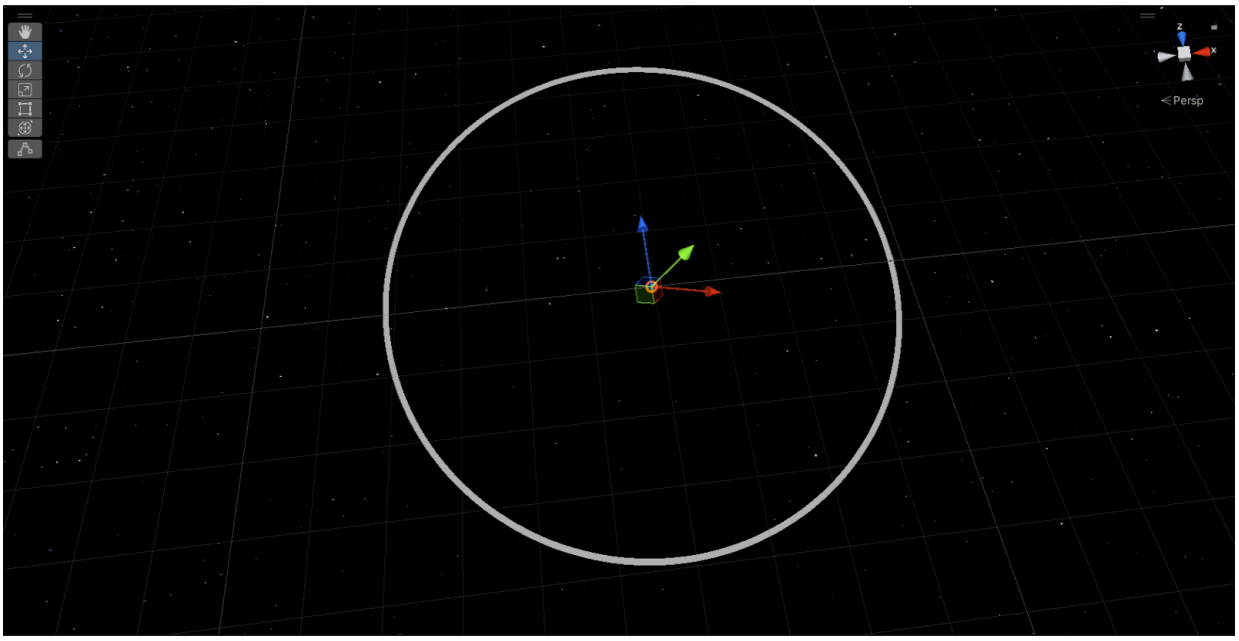
2. **All planets should be able to orbit around the sun**

Yes, all planets can orbit around the sun. Here's an image of it (although the sun is not visible in the image, it is definitely there in the middle).



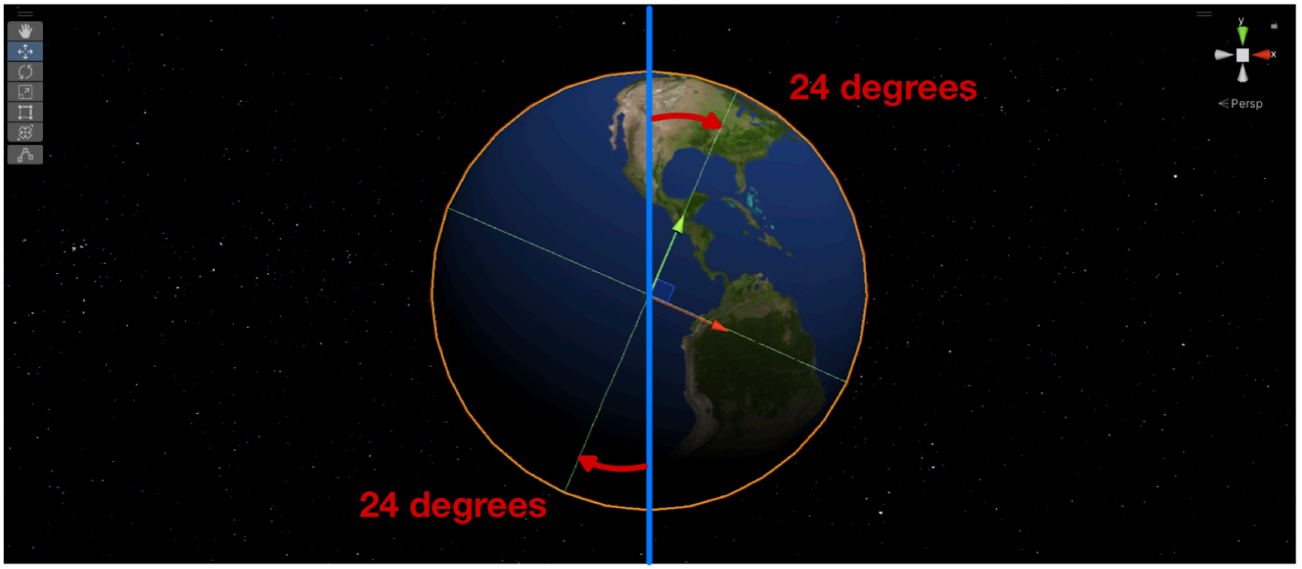
3. **The moon should also be able to orbit around the earth**

Yes, the moon is able to orbit around the earth. Here is an image of the moon's orbit path (in white) relative to earth (three arrows in the middle).



4. **All planets should be correctly rotated by their axial tilt angle.**

Yes, I tested it for all the planets using a protractor to my screen — they appear to be correct. Here is an example for one planet, earth. The blue line is at 0 degrees and the thin green lines are the rotation of the planet. The short green arrow near the middle points towards the up direction of the planet, or the north pole in this case. The value I got using a protractor (which I know may not be the most accurate way to test it) was about 24 degrees, whereas the true/real value is 23.439 degrees (as shown in the table in stage 1).

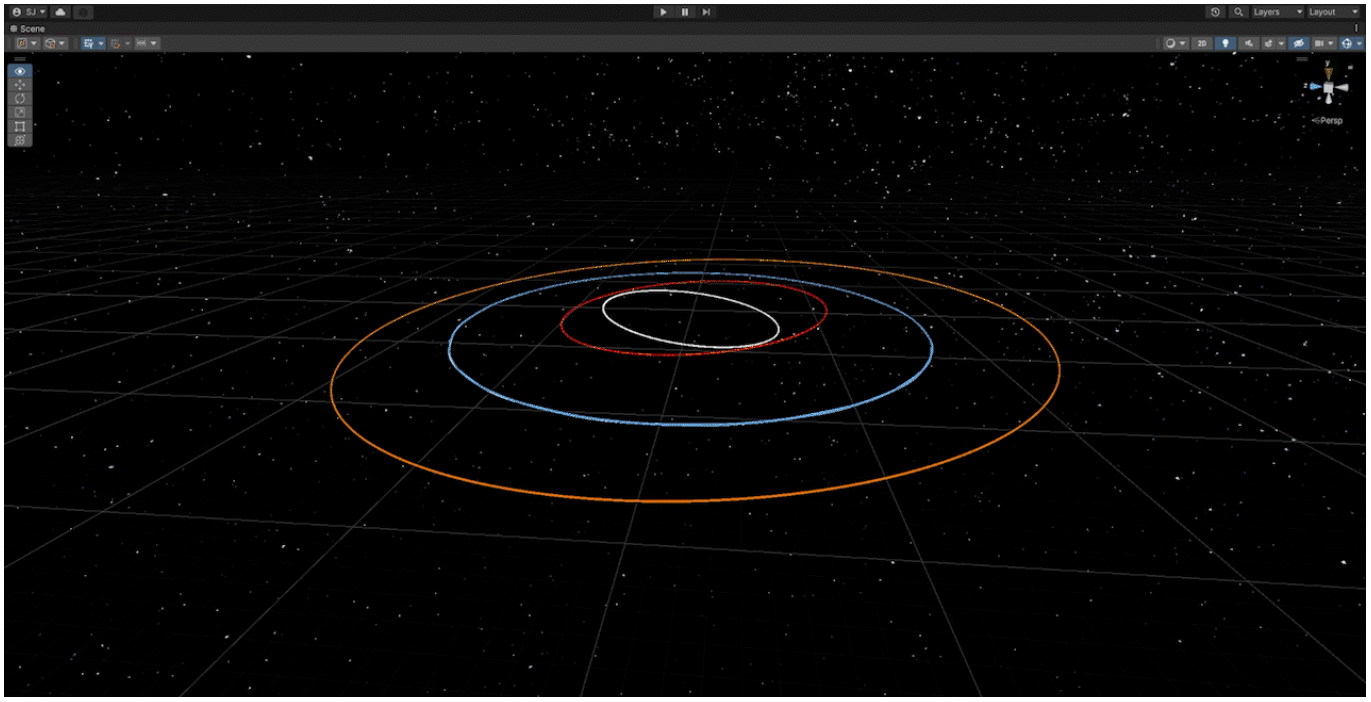


5. **All planets should rotate correctly according to their respective rotation period.**

This one I was unable to test as there is currently no way to tell the time within the simulator. So, with no time measurement I can't accurately tell if the planet is rotating at the right speed. I thought of an idea where I could count the number of times the planet rotates during one orbit (ie. around once back to its starting point), but it's very hard to tell when the planet reaches back to its starting point.

6. **The orbital paths of each planet should be visible and correctly rotated by their respective inclination angle.**

Yes, all planets had a trail renderer component attached which displays their orbital path. The orbital inclination angle for each planet can also be seen, as shown in the GIF below:



Evaluation

Overall I am quite satisfied with the progress I have made during this first iteration, but I know in the back of my head that this is really only the halfway point — there is still the second iteration to complete. Although this first iteration successfully met most if not all the success criteria, there are still many things I would like to add to further improve the program.

The first major improvement I would like to add is a GUI (graphics user interface). In its current state, the project can only be run within the editor — there is no way for the user to change any values or even explore the simulator. However, with the implementation of a GUI, all this would be possible and even more if I chose to do so. I've decided that this should be the main focus of the second iteration; creating a GUI.

Additionally, I found that the meshes for all the planets are very low quality. When zoomed in close to a planet in the editor, instead of a crisp and smooth circular edge, you can see jagged and polygon-like edges — especially when the planet is rotating. Also when the simulator is being run, for some reason the planets shake, or vibrate, quite a lot — I'm not too sure why as of yet.

Another improvement I would like to make is with light, specifically the sun which for some reason has a shadow?! The sun in my project was a static texture like all the other planets, so it was lit on one side but dark on the other — behaviour definitely not like the real sun. Also, the direction of the light never changed, which meant the lit side of each planet stayed on the same side, even when the planet orbited around and faced a different direction. To fix this, I would need to update the direction of the light so it is always coming from where the sun is placed. Although these issues are only cosmetic, I think it would be a great idea to fix these up in the future.

Also, another minor aspect I would like to improve on is the texture of the earth. At the moment, it looks very saturated and nothing like what it actually looks like from space — there are no clouds, no atmosphere, no lights, just a static texture slapped on a sphere mesh. I would like the earth to actually resemble what it looks like from space; so the simulator feels somewhat accurate rather than just some spheres rotating around.

All in all, this iteration went mostly to plan. Although there were some hiccups on the way, the program ended up quite nice. Despite all the improvements that could be made in the future, I think it's safe to say that this program succeeded in achieving its desired purpose.

Iteration 1: Stage 5 - Maintenance

Throughout this project, I have strived to use good intrinsic documentation (ie. variable names) and made sure to include numerous comments within my scripts. Not only does this make it easier for me to read, understand and fix my code, but it also allows other developers to extend my code. Hypothetically, if in the future someone decided to build further off the progress and coding I have made during this project, it will be much easier for them to understand and make the necessary changes with good code maintenance.

In addition, the project may need to be updated to newer versions of Unity when they get released further in the future. Eventually this version of Unity will become old and perhaps incompatible with newer technology, so it would be a wise decision to update which version of Unity the project is based around. However, this decision depends on many circumstances as upgrading to newer versions may break existing features within the program — which isn't good. To ensure this doesn't happen, the scripts and code within the program should be maintained with the current technologies available and adapted when old software becomes obsolete; prolonging the life of the program.

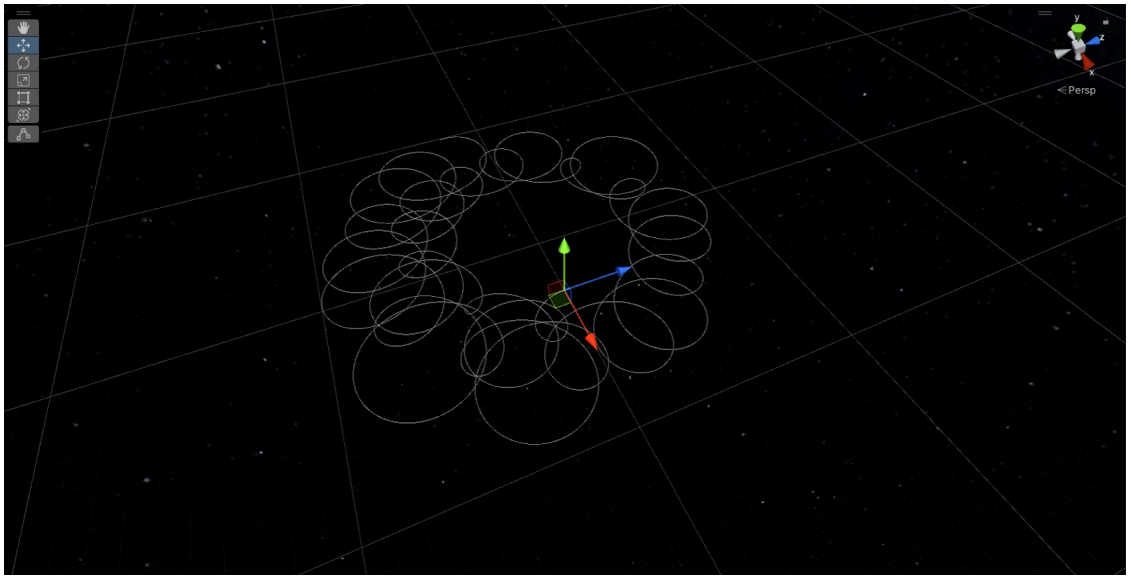
Iteration 1: Journal

Date & Title	Description
9/4/22 — Task Notification	Received task notification.
15/3/22 — Brainstorm Idea	Over the past few days, I have been thinking about the possible concepts I could use for my project. After discussing with my neighbour and mentor, James, I've come up with two ideas that I am quite interested in doing: 1. A 3D simulator of our solar system using real planetary data. 2. A machine learning algorithm for autonomous driving (ie. an algorithm that can analyse and detect hazards from a dashcam video). Tomorrow is the due date when I need to confirm my idea with my teacher, so tonight I will give it some more thought into which concept I would like to pursue.
16/3/22 — Idea Confirmation	Today I spoke with my software teacher and confirmed the idea for a 3D solar system simulator. Out of the two ideas, I was most eager with the solar system as I will be using knowledge I just learnt last from physics last week. I think the solar system idea will be a great choice for my project; despite being quite familiar with the concepts, implementing and adapting all the maths/physics will definitely challenge my coding ability.
17/3/22 — Preliminary Research into the n-body problem	After yesterday's discussion and throughout today I did some research into what a 3D solar system simulator entails. After some searching, I found some interesting pages that are very relevant to my idea for a simulator. It's called the n-body problem and the n-body simulation, which can be found using the links below: https://en.wikipedia.org/wiki/N-body_problem https://en.wikipedia.org/wiki/N-body_simulation Using this new information, I started to sketch out in my head what my concept for this first iteration is — jotting down some dot points.
18/3/22 — Concept + Planetary Data Table	Today I wrote out my concept and goals for this iteration — getting the fundamentals built and running. I also created a table with all the planetary data for the planets (and the moon) in our solar system. This will definitely come in handy later on when I need to input all the data into the program.
20/3/22 — Research into Equations	Over the past few days I did some research into the logistics of the project — that is, the equations I need to implement. After speaking with my physics teacher for some assistance, I realised that some of the equations I need to implement are actually different from what I've learnt in physics class. The

	equations I've learnt are for circular orbits, however in real life orbits aren't prefectually circular, rather elliptical. So with some guidance, I did some online research into what equations I actually need — I started to write these down in the stage 2 section.
21/3/22 — Success Criteria + Equations	Today I touched up on my research and documentation of the physics equations — specifically calculating acceleration and initial velocity. During my research, I found a really helpful webpage which explains the maths behind calculating the initial velocity of a planet. It's a webpage from the University of Georgia; here's the link: http://jwilson.coe.uga.edu/EMAT6680Fa05/Bacon/hohmanntransfers I also wrote the success criteria for the project — a general outline for what I want this first iteration to achieve.
23/3/22 — IPO Table + Equations	Today I continued to research and update my documentation for the physics equations. I decided to include a section where I explain what eccentricity and orbital inclination is as I am planning to use these in my program. Additionally, I wrote up a simple IPO table which outlines the inputs, processes and outputs of the program. Since I am planning to complete the UI in the second iteration, the inputs are from the program data (ie. data table from stage 1), not the user.
25/3/22 — Structure Diagram + Pseudocode for Calculating Acceleration	Today I started creating the structure diagram as well as writing up the pseudocode for the CalculateAcceleration function, which as the name suggests, calculates the acceleration for each object. I've added comments into the pseudocode to outline what's happening; which I have written in red font. Also, I continued further with researching the equations — specifically researching the methods of numerical integration, which I believe will allow me to implement the movement into the program — I'm not too sure, but I'll continue to do some more research into it.
26/3/22 — Research into Numerical Integration: Velocity Verlet	Today I continued to research the methods of numerical integration, such as researching web pages such as these: https://web.archive.org/web/20120713004111/http://wiki.vdrift.net:80/Numerical_Integration https://en.wikipedia.org/wiki/Numerical_integration https://en.wikipedia.org/wiki/Verlet_integration I discovered that I can't use normal integration methods because it's impossible to write one function for the movement of all the planets in the solar system. Rather, I need to use a numerical integration method to approximate the answer. I discovered that there are actually many different methods, each with their own pros and cons. But at the end of the day I decided to go with "Velocity Verlet" as it makes the most sense to me, as well as it has a mix of pros and cons which I think will be appropriate for my project. Later, I updated the documentation to include this.

29/3/22 — Finished Pseudocode + Data Dictionary	Over the past few days, I have been smashing out the pseudocode as well as keeping a data dictionary up to date. Rather than having one long algorithm, I broke the pseudocode up into their own distinct functions. There are 4 in total: - 'CalculateAcceleration' which calculates the acceleration for each planet in respect to all the other planets - 'CalculateInitialVelocity' which calculates the initial velocity for each planet. I've also made sure to factor in their respective orbital inclination and eccentricity (this part was pretty challenging and took quite a long time). - 'UpdateCelestialMovement' which updates the movement for a specific planet. - 'PlanetsController' which controls all the planets using the above 3 functions.
30/3/22 — Unity Project Creation + Planet Images	Today I created the Unity project files. However I soon realised that I didn't make/get the textures for the planets — so I went online to look for some free images. After some digging I found this great source: https://www.solarsystemscope.com/textures/ It has heaps of high quality textures based on NASA and other satellite imagery — some even in 8k resolution! I was pretty happy that I stumbled across this resource. I also made sure to take close notice at the licensing, which luckily is free to use. The exact wording for the licencing is: "Distributed under Attribution 4.0 International licence: You may use, adapt, and share these textures for any purpose, even commercially." With these textures I imported them into my unity project files. This took longer than expected for a number of reasons, such as the sheer number of images (I had to do lots of file management) as well as the fact that I was downloading multiple 8k images onto my computer (which using the school's wifi, does take quite a long time). Once the images were imported into my project, I worked on turning them into materials. This also took quite a bit, but the process was relatively simple.
31/3/22 — CelestialBody Script + Acceleration and Initial Velocity function	Today I worked on creating the 'CelestialBody' script. This script holds all the planetary data (from the table in stage 1) as well as the 'CalculateInitalVelocity' and 'CalculateAcceleration' functions. Instead of having all these functions in the 'PlanetsController' script like in the pseudocode plan, I decided to move them into this new script for one main reason, organisation (I think it's a better decision to make separate scripts which handle different tasks, not just one large script to do everything). Many compile errors later I decided to test to see if the functions work as intended. I used a Debug.Log statement to print out the results for each function, and then compared it to my own working out using a piece of

	paper and a calculator. The results matched exactly what I calculated, which I am quite happy about.
1/4/22 — PlanetsController Script using Velocity Verlet vs. RigidBodies	Today I worked on creating and implementing the 'PlanetsController' script. Instead of straight implementing the Velocity Verlet algorithm, I first tried to use rigid bodies to move the planets — a more simpler and easier approach to implement the movements of the planets. Everything was going smooth and well, but not for long — I hit a number of problems. Firstly, the unity rigid body component had a maximum mass amount, which was 1e+9 (or 1000000000). Since I was using the real mass of each planet (which were way over this number), the program capped each planet's mass at this number. However I did find a work around, which was scaling the mass of each planet in terms of earth's mass. This did work (since the mass of each planet still remains in the same ratio to each other), but there was still another major problem I encountered. To move the rigid bodies I was using rb.AddForce(), which adds a force to the rigid body using a Vector3. However, I soon realised after the first few runs that for each frame I was adding force, never subtracting, so the planets would instantly zoom out and off into oblivion. What I really intended was for the program to change the direction of the force, not add the force in the new direction. I tried looking into ways to work around this, but couldn't find any since you can't directly change the acceleration of a rigidbody, only its velocity and position. I eventually decided to scrap this idea and implement Velocity Verlet instead (as per the plan). Although the pseudocode was already written out, actually implementing it into c# took some time and required much debugging. Eventually I got it to work, and managed to get the earth to orbit around the sun! (at the moment only the earth and the sun are in the simulator, I will add the rest once I get everything else to work out first).
2/4/22 — Adding in another planet + cleaned up scripts + orbit paths script	Today I added another planet into the simulator, venus. I wanted to ensure that the simulator will work with multiple planets, which it succeeded in doing. I also spent quite a lot of time today developing a script that will show the orbit paths in the editor, without actually playing the simulator. I found starting the simulator to test it took way too long as well as the planets were far too hard to see (since they were to scale). So I created this script called 'OrbitDisplay' which uses Debug.drawline to draw a line where the planets will travel. Although I didn't initially plan to make this script, I thought it was a necessary decision as I think it will definitely help in the future. Although this is great and all, I soon discovered that this script takes quite a long time to compute. So I made the script only run when I change a value, not every frame as it previously was.
3/4/22 — Polished up orbit	Today I continued to work on the 'OrbitDisplay' script. Instead of using Debug.drawline, I decided to swap it with a line renderer. An issue I found

paths script	was that with Debug.drawline, the lines would eventually fade and wouldn't stay there on the screen. However, with a line renderer, the lines can stay on screen forever, or until it is updated or turned off. I also added a feature where it can show the orbit lines relative to another body, which will definitely help when I eventually decide to add the moon in.
6/4/22 — Adding the moon + Fixed Velocity Verlet Implementation	Over the past few days I have been trying to implement the moon, which proved much more difficult than expected. I tried to add the moon in like any other planet, but I couldn't get it to orbit the earth. As much as I tried, the moon would drift close to the earth and then get slingshot way out of the simulator area. After much debugging and looking back at my code to see what I did wrong, I eventually found the problem — two actually. Firstly, I forgot to account for the initial velocity of the moon relative to the earth. The scripts had just been calculating the initial velocity of the moon relative to the sun, when it should have also added its relative initial velocity to the earth. This issue really stumped me, but after looking further into it, the solution was actually relatively straightforward. To fix it, all I had to do was add a few lines of code into the initial velocity function: `// Add initial velocity of orbited body (ignore if orbited body is the sun)` `if (orbitedBody.bodyName != "Sun")` `{` `    init_vel += orbitedBody.CalculateInitialVelocity(G);` `}` With this I could get the moon to orbit the earth, but it didn't look quite right. Here's an image of the moon's orbit path relative to the earth:  As shown in the image, the orbit paths are in spirals when it really should be a near perfect circle. This issue led me to discover another major flaw with my code, which was my Velocity Verlet implementation. I discovered that the issue lied with my looping. Here is the original code I had written:

```
for (int i = 0; i < bodies.Length; i++)
{
    // Get current montion data
    Vector3 acc = bodies[i].acceleration;
    Vector3 vel = bodies[i].velocity;
    Vector3 pos = bodies[i].position;

        // Calculate each planet's position, velocity and acceleration using Velocity Verlet
    Vector3 new_acc = CalculateAcceleration(bodies[i]);
    Vector3 new_vel = vel + 0.5f * (acc + new_acc) * timeStep;
    Vector3 new_pos = pos + vel * timeStep + 0.5f * acc * timeStep * timeStep;

    // Update motion data with new data
    bodies[i].acceleration = new_acc;
    bodies[i].velocity = new_vel;
    bodies[i].position = new_pos;
}
```

What was happening was that each planet's position was being updated directly before calculating their acceleration. On a surface level this seems fine, but diving deeper into the code I realised that this is actually a major flaw that I didn't pick up only until now.

The issue is that for the acceleration calculation to be correct, *ALL* planet's positions must be updated beforehand. This is as the acceleration calculation function works based on the positional data of all the other planets. So if the position data is being updated late, then in some loops the acceleration will be fed wrong position data. This wasn't really much of an issue with only two planets, but since adding the moon, this issue has really sprouted up.

To fix this issue I had to split the for loop into 2 or 3 separate loops. This way I can update all the positions *BEFORE* calculating the accelerations of each planet. Here is the updated code that does work (I also decided to go back and fix the faulty pseudocode which I had written in stage 2):

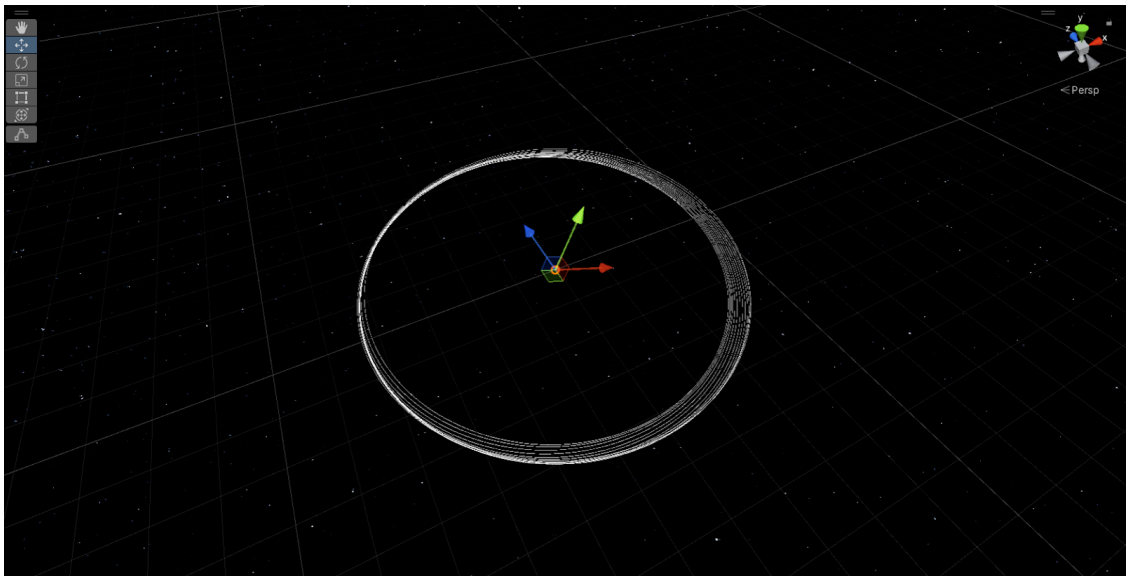
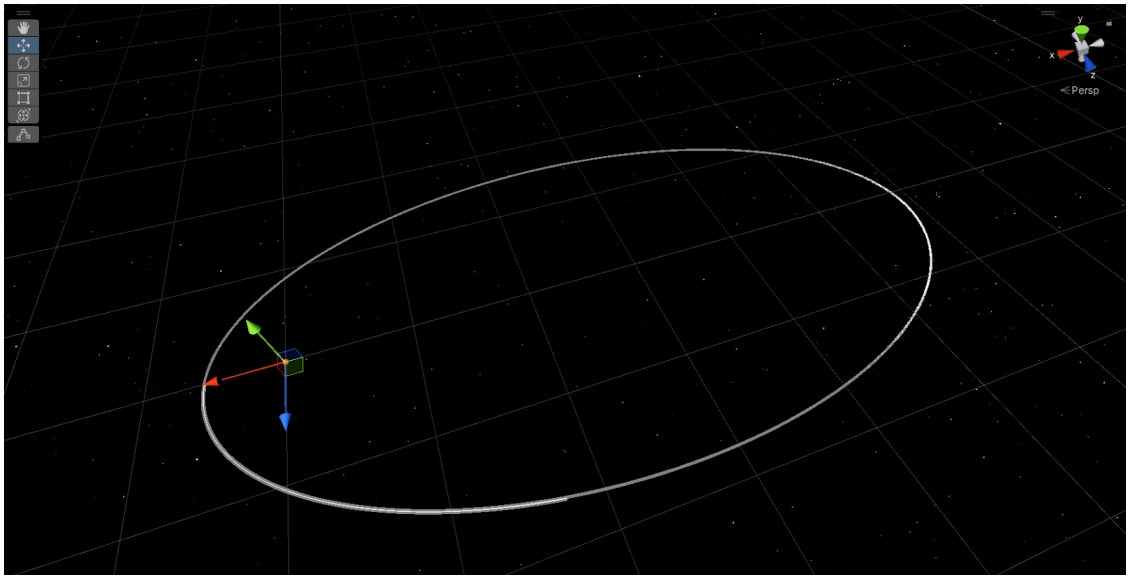
```
// Update pos using Velocity Verlet 1 time step ahead
for (int i = 0; i < bodies.Length; i++)
{
    bodies[i].position += bodies[i].velocity * dt + 0.5f * bodies[i].acceleration * dt * dt;
}

// Now caluclate vel and pos
var old_acc = new Vector3[bodies.Length];
for (int i = 0; i < bodies.Length; i++)
{
    // Save old acc before calculating new acc
    old_acc[i] = bodies[i].acceleration;

    // Calculate new acc at 1 time step ahead
    bodies[i].acceleration = bodies[i].CalculateAcceleration(bodies, dt, G);

    // Update vel using Velocity Verlet 1 time step ahead
    bodies[i].velocity += 0.5f * (old_acc[i] + bodies[i].acceleration) * dt;
}
```

After this fix, the moon orbited the earth much better and more circular (as shown in the image below; the three arrows are the earth):

	
8/4/22 — Orbital Inclination and Eccentricity	Today I implemented orbital inclination and eccentricity into the program. Despite being pretty complex, it was actually surprisingly straightforward to add into the program. In doing so, I also had to update the 'OrbitDisplay' script so it would also show the orbit paths with the new eccentricity and inclination angle. It surprisingly works very well, and you can make the orbits nearly any shape you want. Here is an example which I was playing with where I set the eccentricity very high (the three arrows is where the sun is): 
9/4/22 — Axial Tilt and Rotation	Today I added axial tilt and rotation into the program. It was pretty straightforward — nothing to write home about. I also added a feature where you can change the colour of each planet's orbit path.
11/4/22 — Testing	Today I tested the program using the success criteria I made in stage 2. It was quite interesting and I thoroughly enjoyed seeing what my program can and can't do. I also tweaked a few things, made some minor changes here and there, and updated my documentation (journal) so it is up to date.

14/4/22 — Evaluation	Today I finished writing the evaluation using the results from the testing I did a few days ago. It got me thinking about what I could do to improve the program, and quite a number of ideas came to me. But I think I will focus on implementing ideas that are actually possible in the time frame.
16/4/22 — Maintenance + Cleaned Up Documentation	Today I completed the maintenance stage and any unfinished parts in the documentation. Starting from tomorrow I will be moving into the next iteration, where I will begin to focus on creating a new GUI.

Iteration 2: Stage 1 - Define & Understand

Refined Goals - Iteration 2

Using the testing results from iteration 1, I have refined the goals and expectations for this iteration of the project. In this second iteration, the main goal will be to create the GUI of the simulator — that is all the visual components that can allow the user to interact with the simulator. As mentioned throughout the evaluation section of iteration 1, this iteration will focus on fixing the problems encountered, which are:

1. Fix the sun so it looks more like one;
2. Adding more detailed planet models (as well as Saturn's rings);
3. Improving the texture of the earth to reflect what it looks like from space;
4. Implement the GUI system to allow the user to control the simulator.

In order to achieve this, this iteration will make use of a number of CASE tools: Blender to create the new models, Photoshop to design the buttons, Cinemachine to implement camera movement, GitHub for project file management, and Unity for the creation and development of the project.

Reasons for Development

Personally, the first iteration has been an inspiration project that I have thoroughly enjoyed — which I hope will continue with this second iteration. The reason for the development of this iteration is not only to fix the bugs discovered and improve the features of the project, but also to serve as a useful tool that can be used to assist students (or anyone else) with understanding the physics of the planets surrounding us.

Iteration 2: Stage 2 - Plan & Design

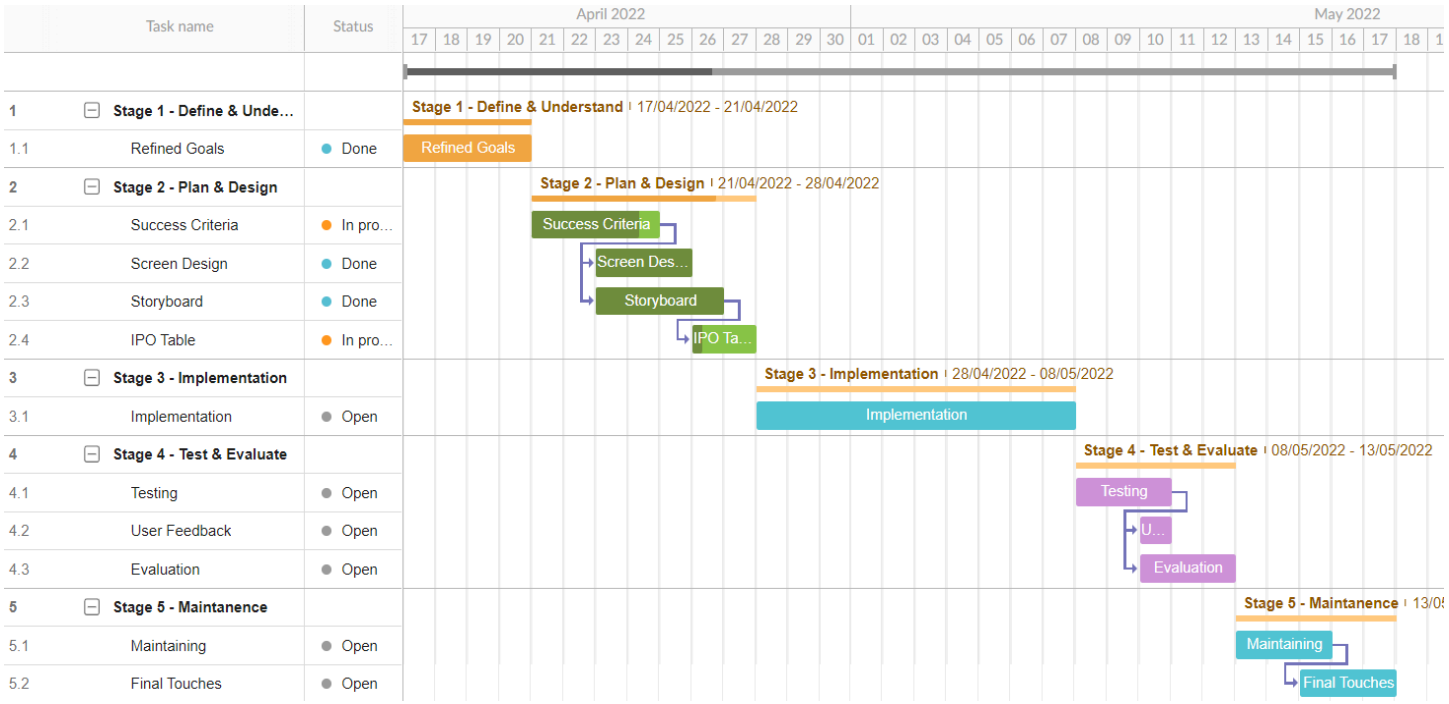
Success Criteria

For this second iteration to be considered successful, the program should meet the following criteria:

1. The program should include a graphics user interface (GUI) system that shows the user the location of all planets visible on their screen.
2. The GUI should be simple, intuitive and easy to navigate through and should also include a toolbar/settings menu to allow users to change the settings of the simulator.
3. The program should include a proper sun that emits light, which should be correctly shown on each planet.
4. The user should be able to control the camera by clicking and dragging the mouse.
5. The user should be able to zoom in/out using the scroll wheel or a slider.
6. The user should be able to click on a planet to move the camera there.

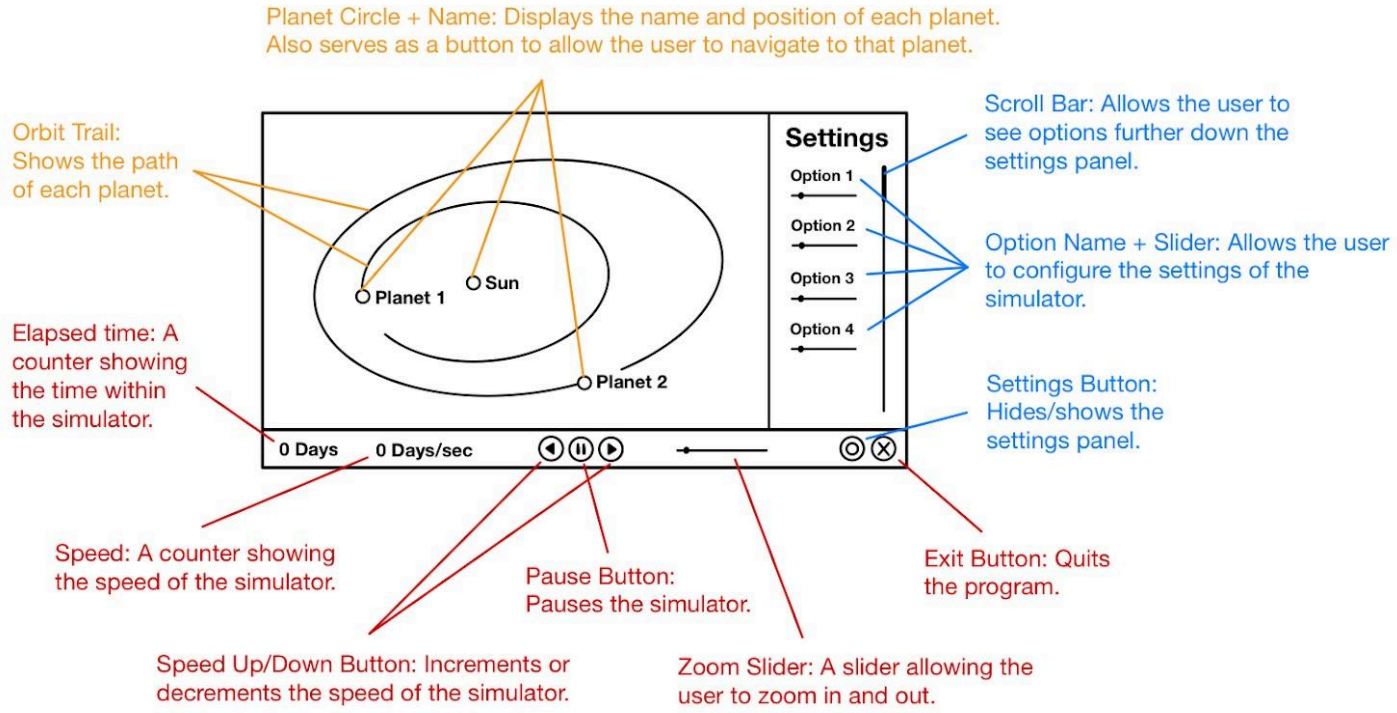
Gantt Chart

Here is the Gantt chart for this second iteration (made using a free trial from *GanttPRO*).

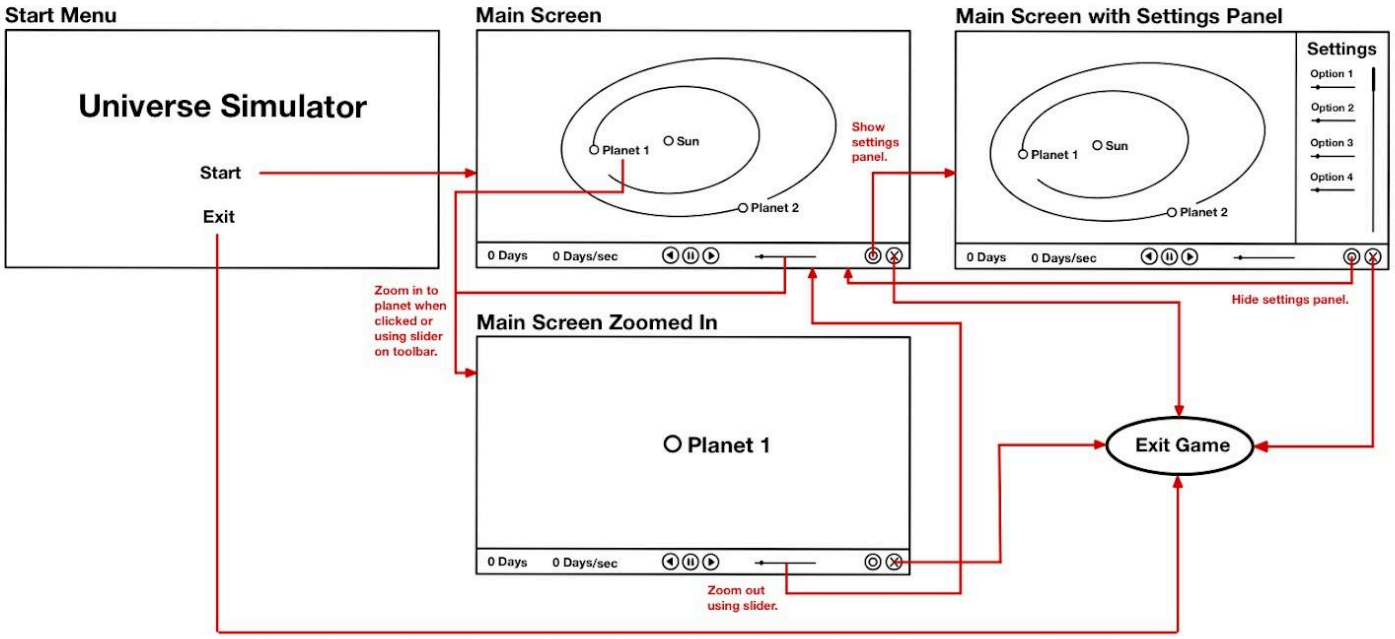


Screen Design

The following is a screen design for the main screen of the simulator (with the settings panel open).



Storyboard



IPO Table

The table below shows a very simple high-level overview of the program's inputs, processes and outputs.

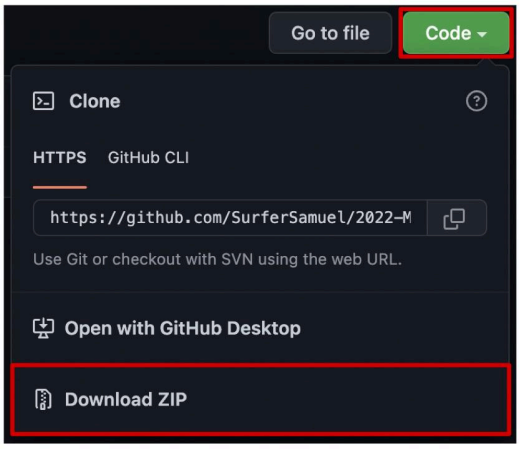
Input	Process	Output
User Clicked Start Button.	Load simulator main screen and hide start menu.	Display simulator.
User Clicked Exit Button.	Quit application.	Application closes.
User Clicked Planet.	Change camera target to the clicked planet. Zoom in camera to the clicked planet.	Smooth camera transition from current position to clicked planet.
User Cursor Position User Cursor Clicked	If cursor is clicked, rotate camera around the current planet using cursor position data.	Smooth camera movement around the current planet.
User Clicked Speed Up/Down Button.	Increment/decrement simulation speed variable.	Simulation appears faster/slower.
User Clicked Settings Button.	If settings panel is already open, close the panel; and vice versa.	Display/hide settings panel.

Iteration 2: Stage 3 - Implementation

Program Files

Please see the submitted ZIP files for the program (there are two, one for mac and another for windows). If for some reason you're unable to access it, I've provided some extra links down below — just in case.

If you're unfamiliar with GitHub, to download the ZIP file all you need to do is click the green "Code" dropdown and then click the "Download ZIP" button (as shown to the right).

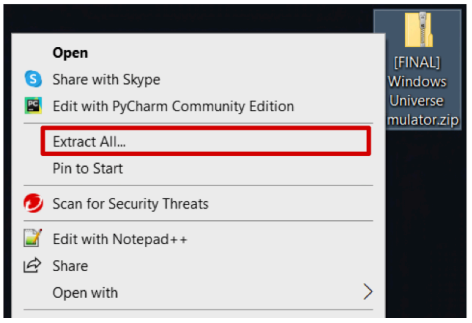


Windows Installation

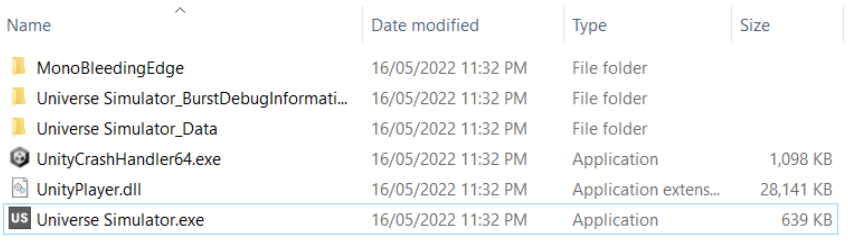
Google Drive ZIP File - Windows
https://drive.google.com/file/d/1-PiK1MD1GospJAjytFZjjwpv-6MMzH7l/view?usp=share_link

GitHub ZIP File - Windows
https://github.com/SurferSamuel/2022-Major-Project-Windows-Final-Iteration_2

Once downloaded, move the file to a suitable location (such as your desktop). Then right click the ZIP file and select "Extract All...". You will be prompted with a window to select the file destination, don't worry too much about it and click "Extract" near the bottom left corner. After this you should see that a new folder has been created — this is the build files. You can now, if you want, delete the ZIP file as it no longer needs to be used.



To run the program, simply open the new folder titled "[FINAL] Windows Universe Simulator" and click the "Universe Simulator.exe" file (as shown to the right).



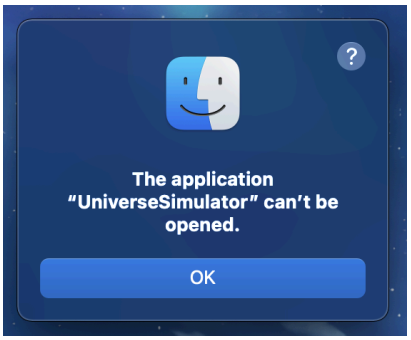
MacOS Installation

Google Drive ZIP File - MacOS
https://drive.google.com/file/d/19X62IzrPmhuectRL9YrxIeSjKjtaW7dG/view?usp=share_link

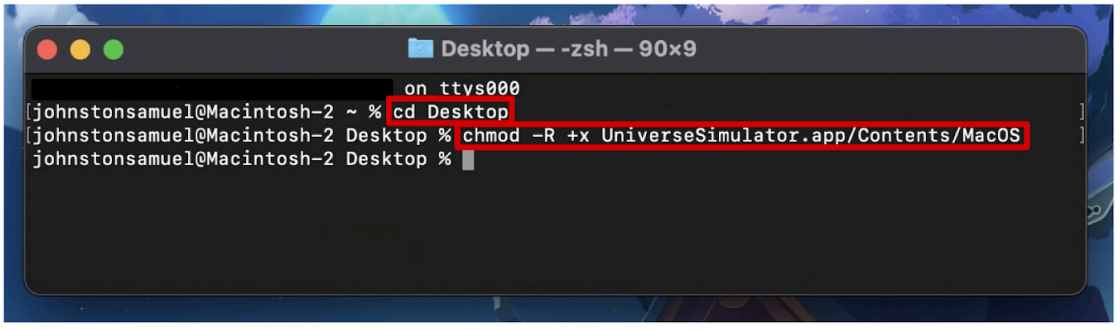
GitHub ZIP File - MacOS
https://github.com/SurferSamuel/2022-Major-Project-MacOS-Final-Iteration_2

Since the program was developed and built on a Windows device, quite a few steps are required to run the program on MacOS devices. The reason behind this is that Windows credentials don't carry over to MacOS, so we need to manually authenticate the program.

Once downloaded, move the file to a suitable location (such as your desktop). Then simply double click to unzip the file — this should create the application. However, you won't be able to immediately open the application (as shown to the right).



To get the application to open, we need to type some commands into terminal. You can navigate to terminal by pressing command + spacebar, then typing 'terminal' into the search bar. Once open, we need to navigate to the file location. This can be done by typing "*cd <file location>*" (eg. "*cd Desktop*" if the files are located on your desktop). After this, we now need to manually authenticate the program, which can be done by typing the following command: *"chmod -R +x UniverseSimulator.app/Contents/MacOS"* (as shown below).



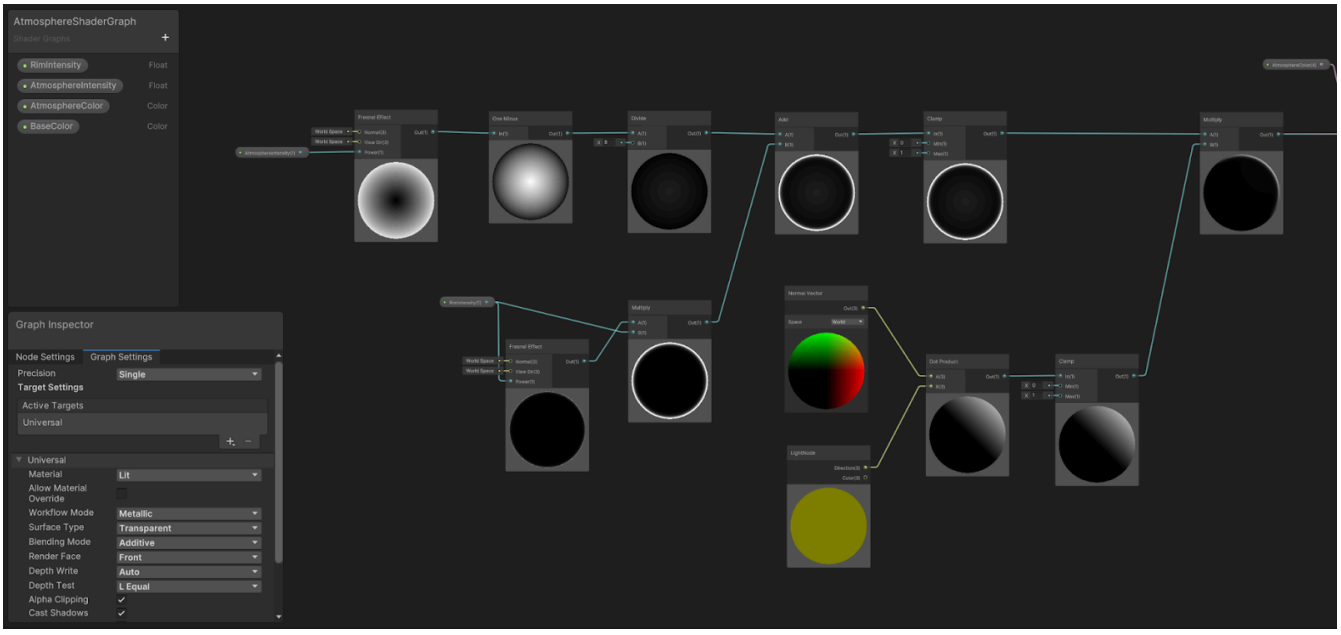
Once the commands have been entered in, you can now open the program. However, if this is the first time the files are opened, instead you need to right click the application and select open (you can't straight double click the application). You will then be prompted with a window saying that Apple can't check the files for malicious software (this project doesn't have anything of that sort). Select the open option and then the program should run like normal.

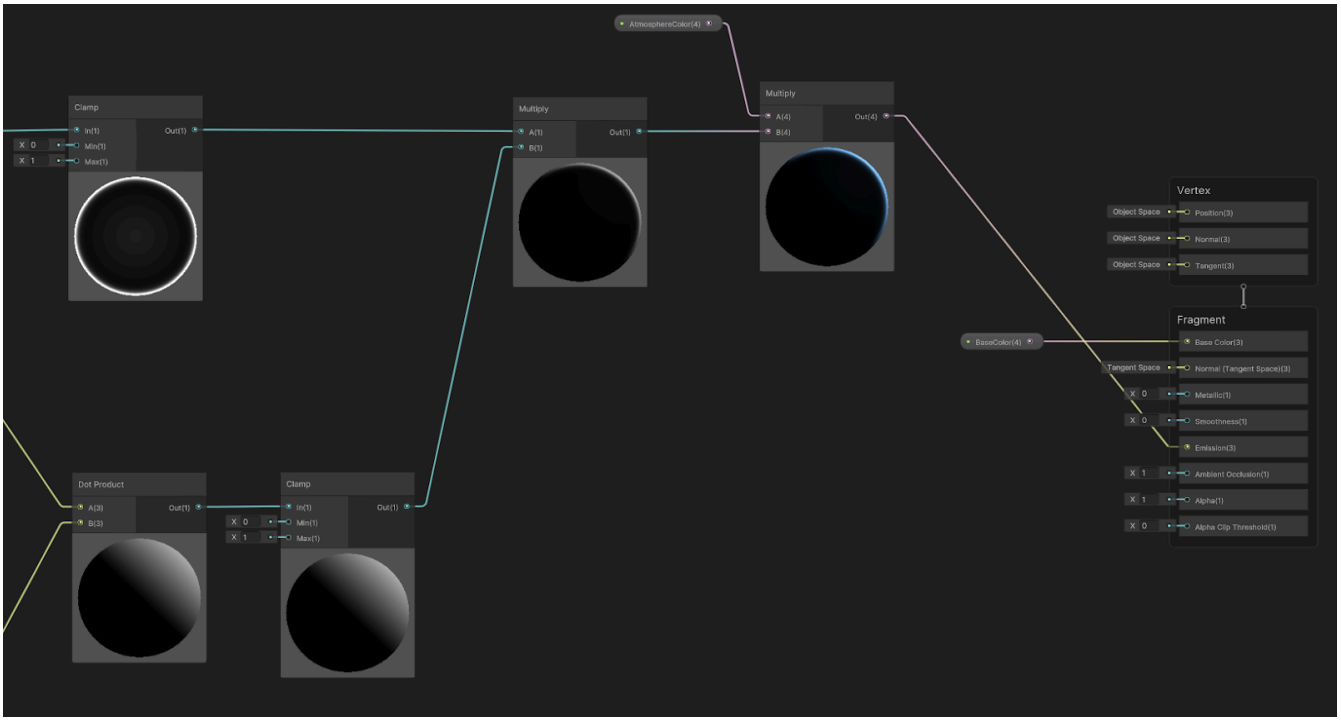
Project Scripts

Here's a folder for all the scripts I have developed:
https://drive.google.com/drive/folders/1V1xAEYwJCQ2wrb8SDwxv1Vs76AgpbtOV?usp=share_link

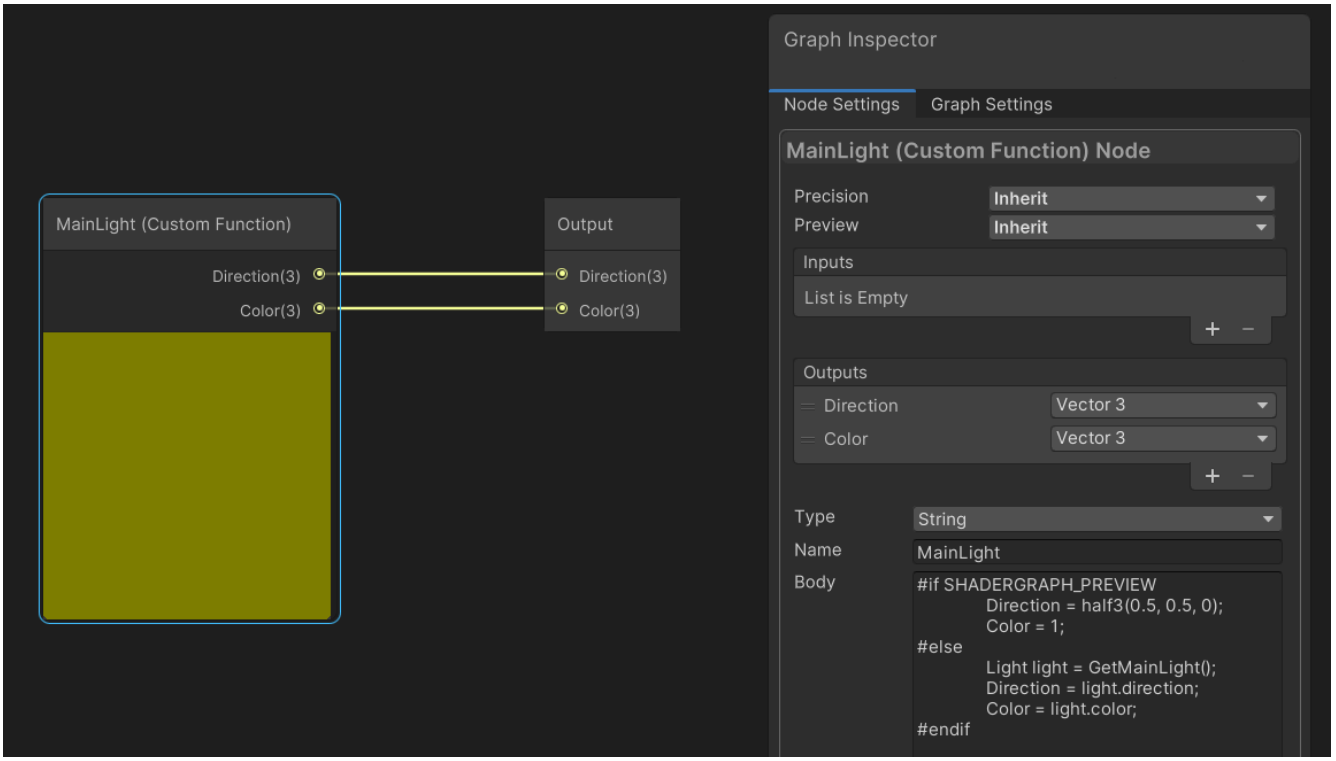
The folder also contains the shader graphs and meshes I have created, although they probably won't be able to be viewed — so I added some images below for each one (some are pretty hard to view because their so long)

AtmosphereShaderGraph:

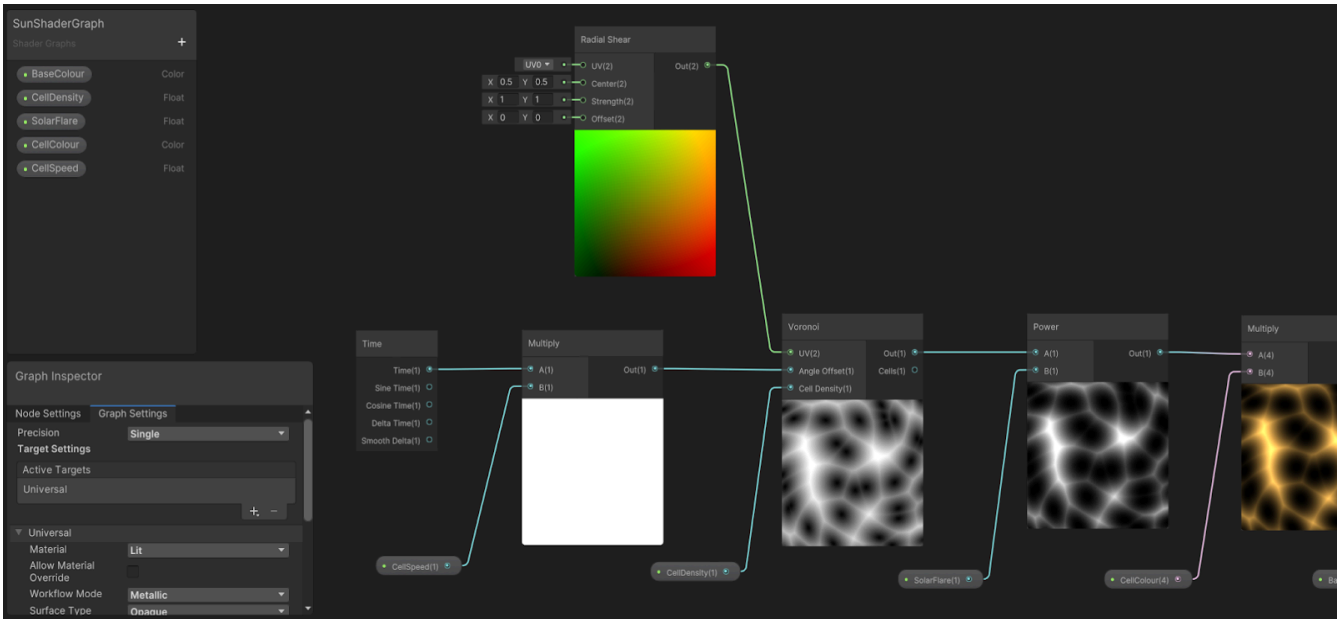




LightNode - Sub-Shader Graph:



SunShaderGraph:



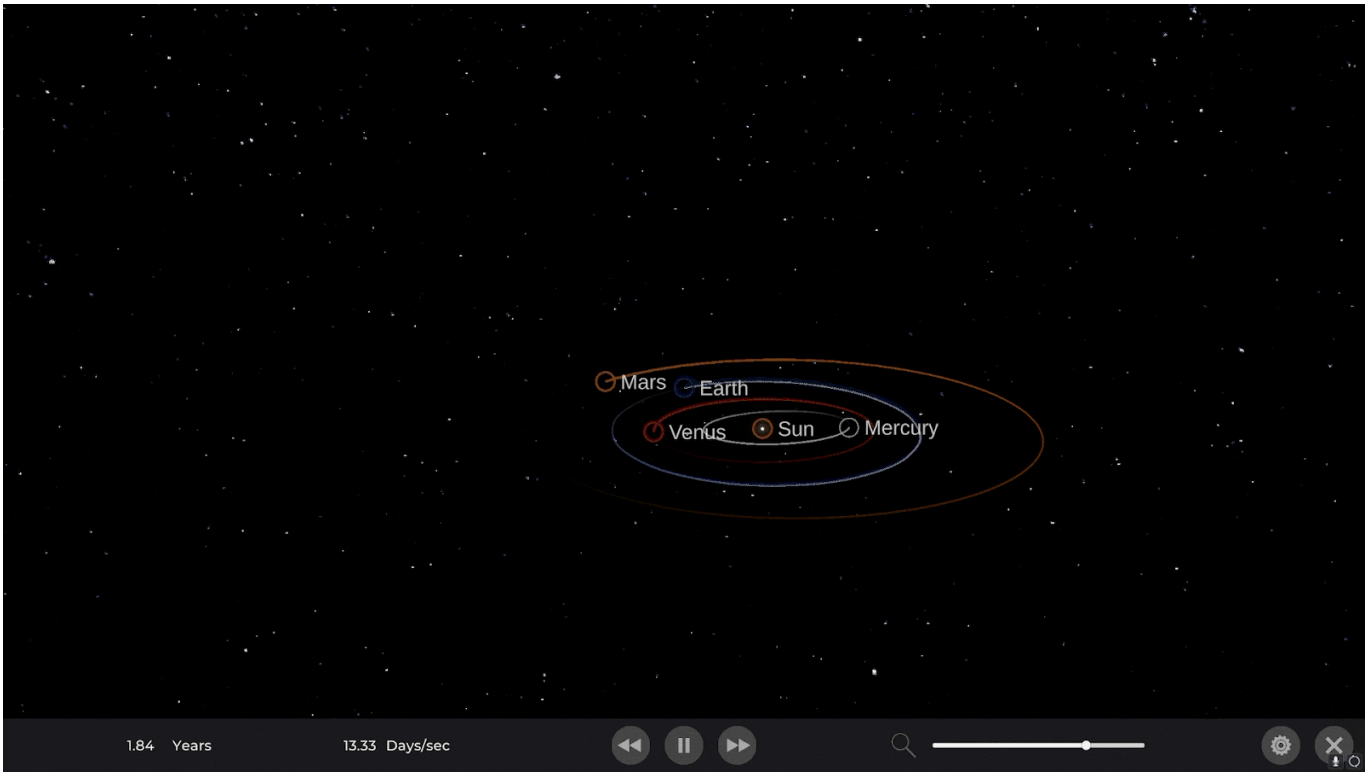
Iteration 2: Stage 4 - Test & Evaluate

As referenced in stage 2, for the project to be deemed successful the program should meet all aspects of the success criteria. Without a doubt I can say that this second iteration has hit its targets; perhaps even exceeding what was previously expected.

The main focus of this testing stage is to check and ensure the program meets all its requirements (as per the success criteria on stage 2). Here is the synopsis:

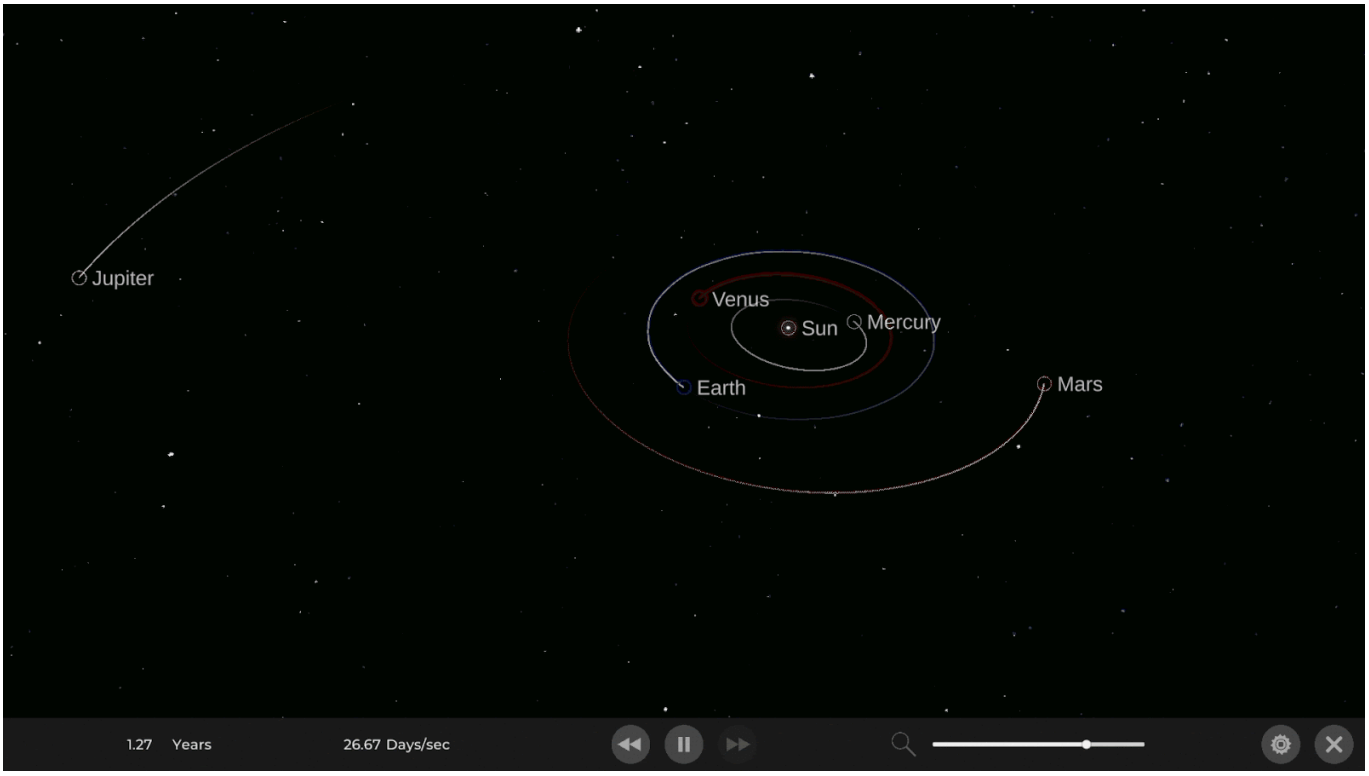
1. **The program should include a graphics user interface (GUI) system that shows the user the location of all planets visible on their screen.**

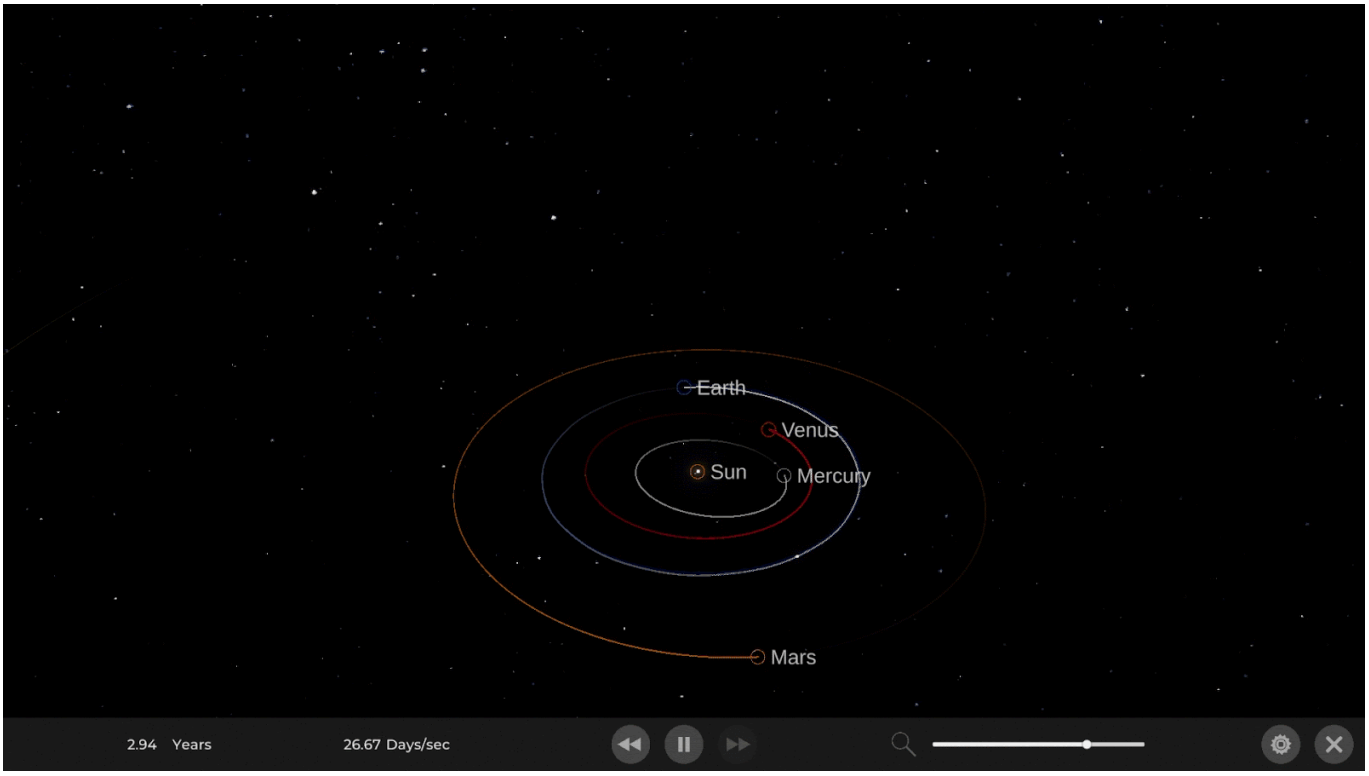
Yes, the program does include a GUI system that highlights where each planet is located — achieved by showing a circle and the name of the planet next to it. It is constantly updated so the GUI elements are always hovering above the planet; although fast movements may cause the GUI elements to fall a bit behind. It also fades the UI elements to prevent the text from stacking. Here is a GIF of it in action (Note: some of the GIF's may take some time to load):



2. **The GUI should be simple, intuitive and easy to navigate through and should also include a toolbar/settings menu to allow users to change the settings of the simulator.**

Yes, the program actually has both of these (a toolbar and a settings menu). The toolbar acts as a quick and easy way to change the most prominent settings, and the settings panel/menu allows for more detailed and specific changes. It also has a settings button on the start menu which allows for the resolution settings to be changed. Here is a GIF of the toolbar and another of the settings menu (Note: some of the GIF's may take some time to load):



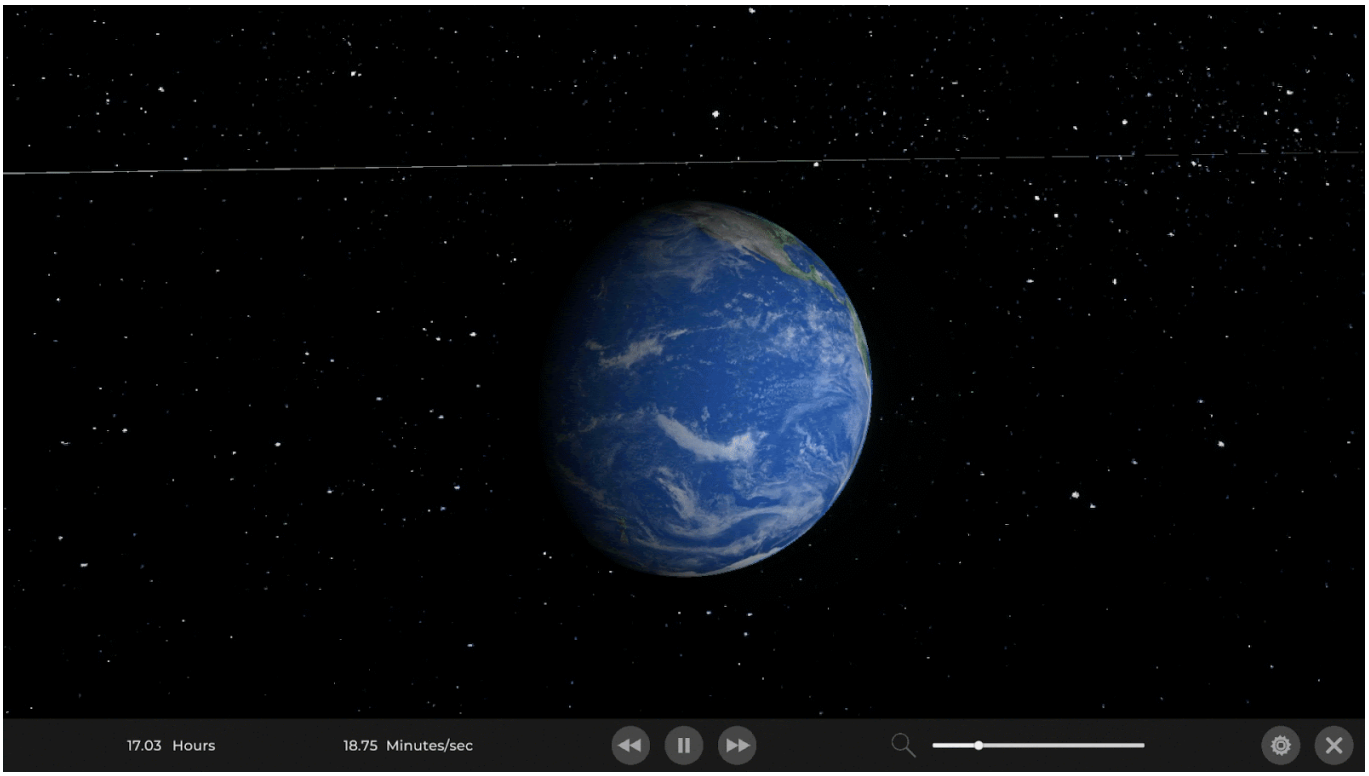


3. **The program should include a proper sun that emits light, which should be correctly shown on each planet.**

Yes, the program does include a proper sun now. It emits light and even has solar flares (or bursts of light that look like one) coded onto it. The light from the sun is also fixed so the correct side of each planet is lit up. Here is an image of the new and improved sun:

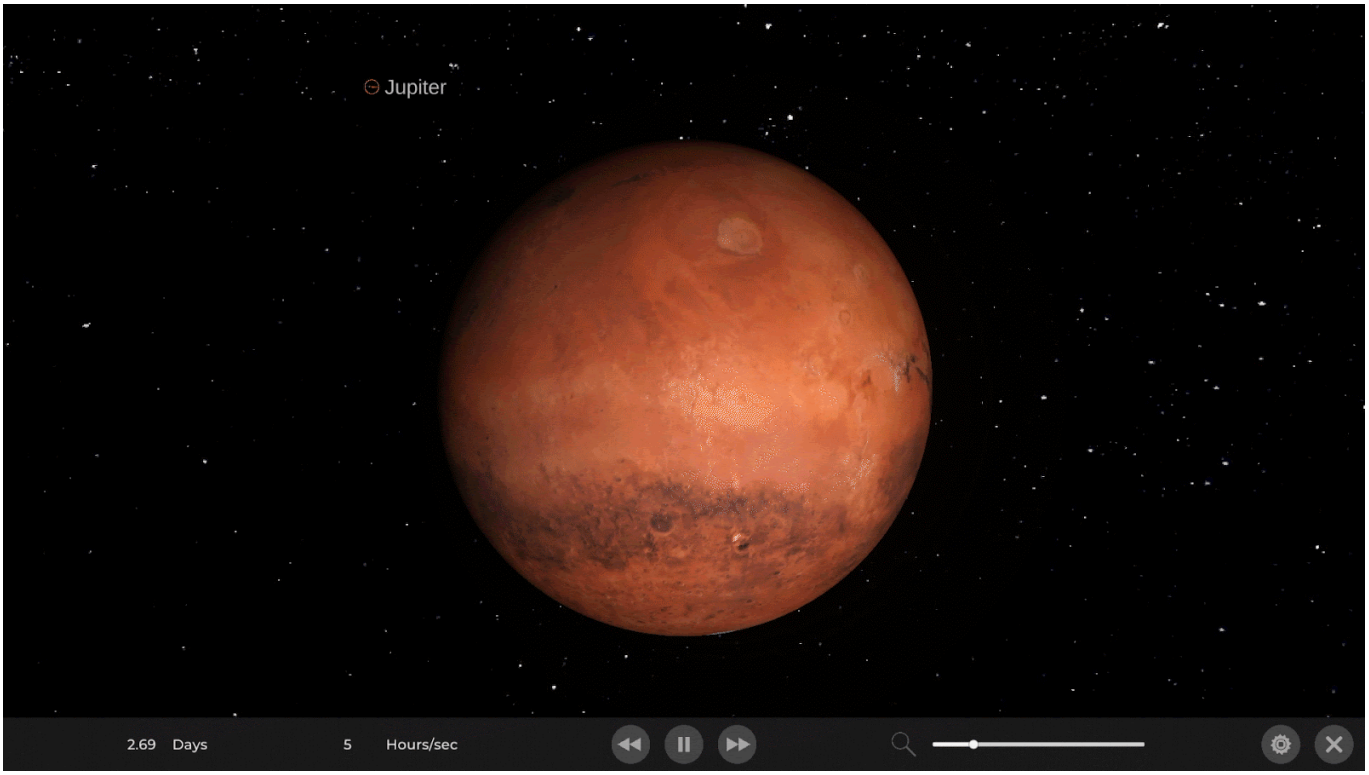
4. **The user should be able to control the camera by clicking and dragging the mouse.**

Yes, the user can click and drag the mouse to rotate the camera around the target planet. However, the sensitivity can be a little bit too sensitive (which was some feedback — more on this further down). Here is a GIF of the camera being rotated around earth (which is also much more ‘earth’ like, unlike in the first iteration) on the page below (Note: some of the GIF’s may take some time to load):



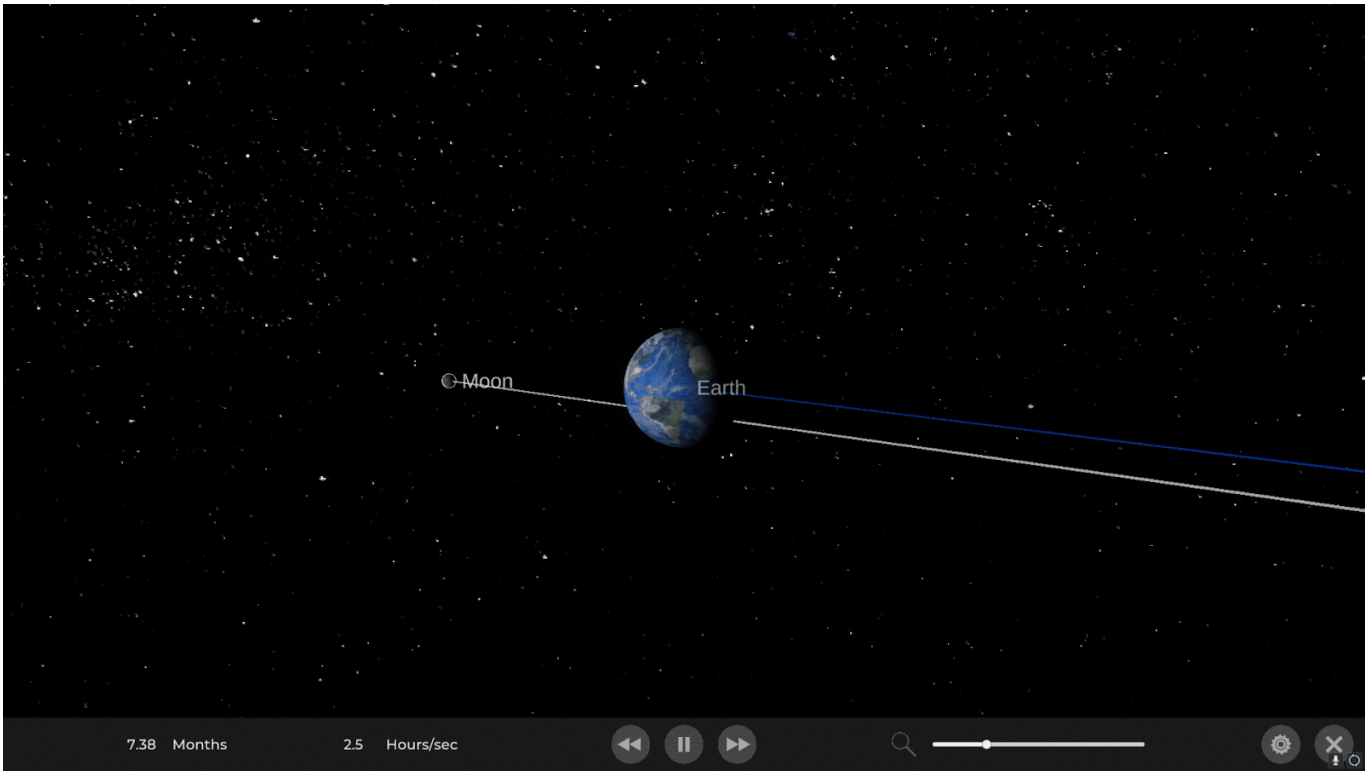
5. **The user should be able to zoom in/out using the scroll wheel or a slider.**

Yes, the user can both zoom in and out using their mouse scroll wheel and the slider on the bottom of the toolbar. There is also an option in the settings panel to change the sensitivity of the zoom. Here is a GIF of mars being zoomed in and out (Note: some of the GIF's may take some time to load):



6. **The user should be able to click on a planet to move the camera there.**

Yes, the program has implemented camera transitions to other planets (which is my personal favourite part of this project). By clicking on a planet's name or circle GUI element, that camera will transition to that selected planet. During this transition, both moving the camera and zooming in/out is disabled — and also some text is displayed showing which planet the user is travelling to. However, clicking another planet mid-transition is not disabled, which is something that will need to be fixed in future iterations. Here is a GIF of the transition between earth and the moon (Note: some of the GIF's may take some time to load):



User Feedback / Suggestions

I have reached out to a few of my friends for their feedback on my finalised program. Below is a table showing what each of their contributions were:

Name	Feedback / Suggestions
Nate	- Would be a great idea to add some sort of notification that tells you when an event happens within the simulator (such as a solar eclipse, etc.).

Will	- Add more planets and moons (such as the moons of Saturn or Jupiter).
Tom	- Create a function that lets the user launch asteroids into the simulator, and see what would happen.
Alex	- Camera movement sensitivity was quite high. - Add an option in the settings panel to change the camera sensitivity.

Although I would love to implement each of their suggestions, due to the timely nature of the task I don't think it will be possible for me to complete them in time. However I will consider their suggestions and perhaps that could be something improved in future iterations.

Evaluation

Overall I am very satisfied with the progress I have made during this second iteration, it exceeded my initial expectations. However, this doesn't mean there aren't any bugs and improvements to be made.

As touched on in the testing section, there are a number of bugs which were discovered. Some small minor ones were fixed quite easily, however some of the larger ones couldn't be fixed in time (although these larger ones were still quite minor in nature). The bugs that have been discovered and are not fixed are as follows:

1. When the simulator is paused by clicking the pause button, increasing/decreasing the speed doesn't update the simulation speed counter in the bottom left toolbar — it is only once the simulation is resumed then does the counter update.
2. During the transition to one planet you can still click another planet and travel to it. The camera snaps to it and hence loses its overall fluidity/smoothness.
3. When increasing the speed, the orbital trails of the planets may temporarily disappear. Also when decreasing the speed, the orbital trails may temporarily overlap each other.

In the future it would be great to fix these bugs as well as implement some of the user feedback collected during testing, maybe even publish the simulator — the possibilities are endless.

All in all, this project has been a great experience and a rollercoaster of a ride. Not only did it challenge my coding skills, but it also extended my knowledge on various software (Unity, Blender, etc.) and taught valuable skills in regards to organisation practices. I think it's more than safe to say that this project has succeeded in achieving its desired purpose.

Iteration 2: Stage 5 - Maintenance

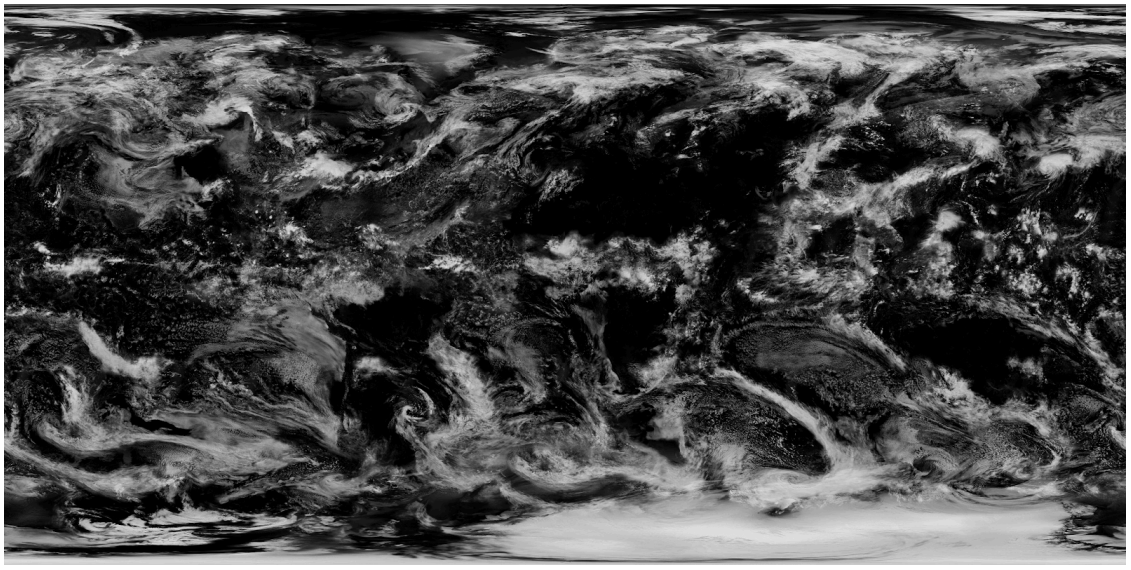
In addition to the tests performed in the previous stage (which were tested on a Windows device), the program was tested on a MacOS device; my school laptop. When testing the program on a MacOS device, the bottom toolbar, the side settings panel, and the GUI circle were out-of-proportion and their positions did not match the original. Hence, some additional tweaking of canvas anchoring was required. To ensure the program is maintained properly in the future, the program should have regular tests and updates to ensure it works on current operating systems (such as Windows 10, MacOS, Linux, or if a new main-stream operating system is released).

Additionally, the relevancy of the project also needs to be considered during maintenance. Since technology is always evolving, our knowledge of the solar system and the universe around us may change — such as the discovery of new planets or more detailed data on each planet. Hence, the program should be maintained and updated to match the current data available. However, this doesn't not completely future-proof the program as new and better software could make this program obsolete. An example of this is NASA's solar system map, which is very similar to the design of my program — except it is done to a much better degree using satellite imaging and detailed 3D models. Thus, the implementation of new features and improvements (such as the few outlined in the evaluation section) serve as an essential role to ensure the program is maintained relevant with future technology and future society.

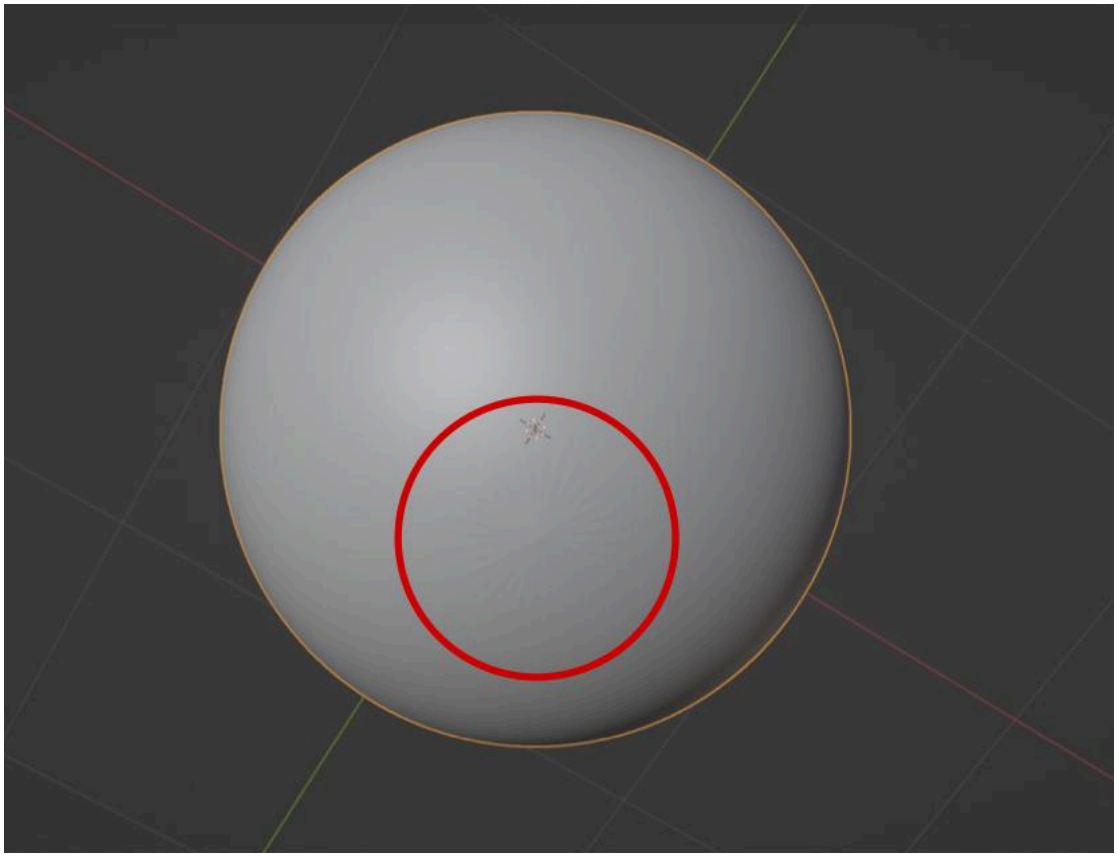
Iteration 2: Journal

Date & Title	Description
17/4/22 — Started Iteration 2	Yesterday marked the end of iteration 1, and today marks the start of iteration 2. Most of my time spent today was reading over what I wrote in iteration 1 and thinking of ways to improve and build off it.
20/4/22 — Refined Goals	Today I finished writing the refined goals for this iteration. I made sure to use the work done in iteration 1 to guide the goals for this iteration.
22/4/22 — Success Criteria	Today I wrote up the success criteria for the project based on the last iteration — it's a general outline for what I want this second iteration to achieve. Unfortunately, I did leave starting the second iteration a little late so there isn't as much time as I hoped.
24/4/22 — Screen Design + Storyboard	Today I drew up a detailed screen design as well as a storyboard using my Ipad. The screen design has the toolbar at the bottom, and a sliding settings panel on the right hand side. I also drew up a very simple main menu, but this will probably change later down the path.
25/4/22 — IPO Table	Today I created a very simple IPO table for all the user inputs of the program. Many of the processes are straightforward, but some will require some more thought put into it.
26/4/22 — Start Implementation + Cinemachine Cameras	Today I started to implement the refined changes into the program (as per the success criteria) which started off with me downloading all the necessary software. I downloaded Blender onto my computer and imported the Cinemachine package into the Unity project. I also created the cameras (Free-Look Cameras) for each planet and tweaked with its settings until it was as I liked. Although there was already a built-in method/function for the user to move the camera, unfortunately there wasn't one for zooming in/out — so I had to code it myself. I had to do quite a lot of research into the documentation of Cinemachine as I am not very familiar with the package, especially with changing the variables within one of its components.
28/4/22 — Sun Shader + Earth Shader + Blender Meshes + Bloom Effect	Throughout yesterday and today I have been working on creating a shader for the sun and also for the earth. Since I am very unfamily with shaders and especially making shaders, I did quite a bit of research online to better understand how they work. Similar to Blender's shader graphs, Unity has shader graphs which are represented using visual components, called nodes. Using these nodes I can connect and create the desired effects for the sun and for the earth's atmosphere — which I followed from these videos: https://www.youtube.com/watch?v=ykwvCCqdcCs https://www.youtube.com/watch?v=bFWM8vJZQ_0 In addition to these shaders, yesterday I added a cloud texture for the

earth. Since Unity's shader graphs don't have a transparent BSDF node, I had to use Photoshop to manually mask out the black background. Here is the source image of the clouds texture:



Also, today I created the models/meshes for the planets (which is just a sphere) and the rings for saturn (a disk with a hole in the middle). During this process, I ran into an issue when adding further detail to the sphere mesh — specifically a stretching issue at the top and bottom of the mesh. Here is an image of the stretching, circled in red (it's a bit hard to see):

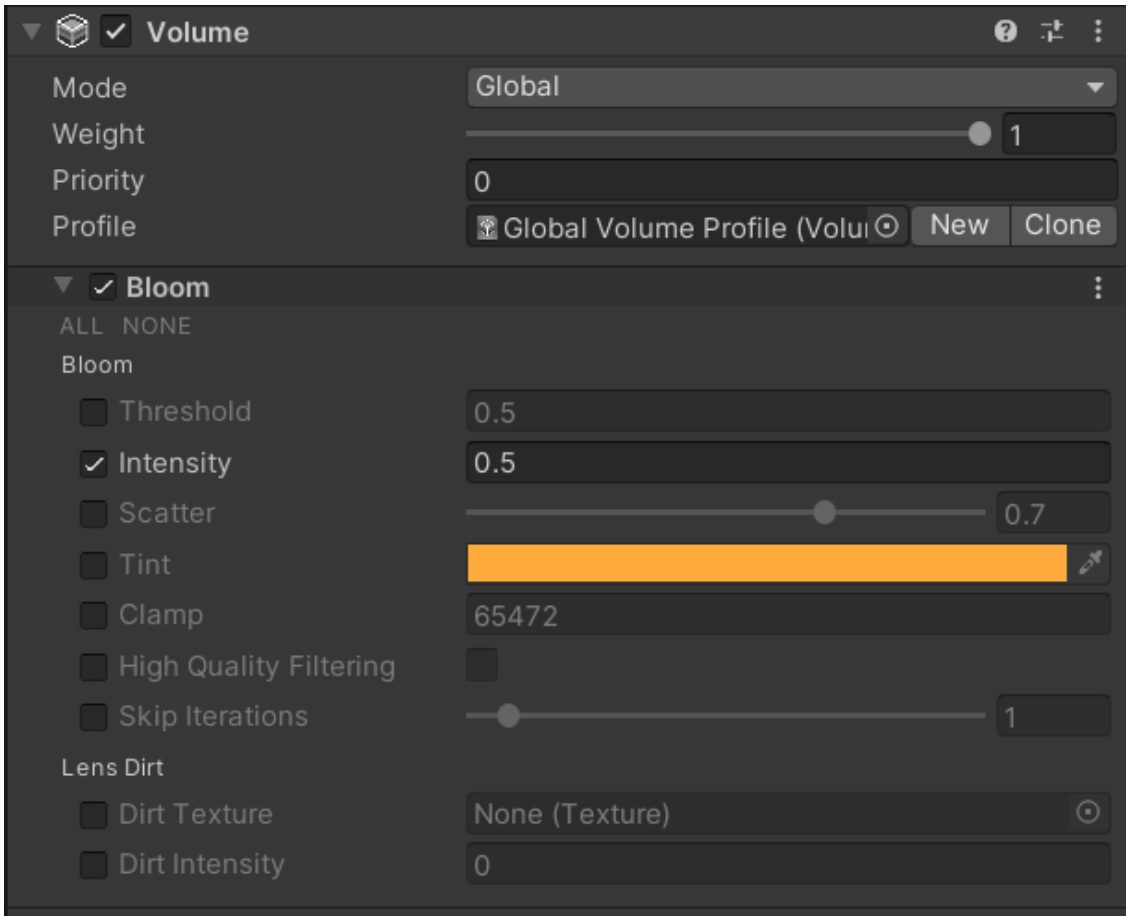


To fix this, instead of using a subdivision surface component and then smoothing it, all I had to do was manually change the number of segments and rings the sphere had.

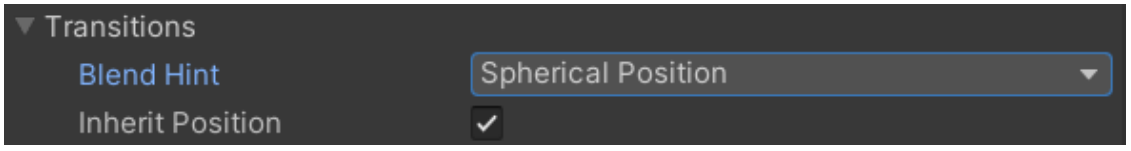
For ring mesh, I decided to follow an online tutorial video. Not only am I quite new to blender, but also my focus should be on the program and the coding itself. Here is the tutorial video:

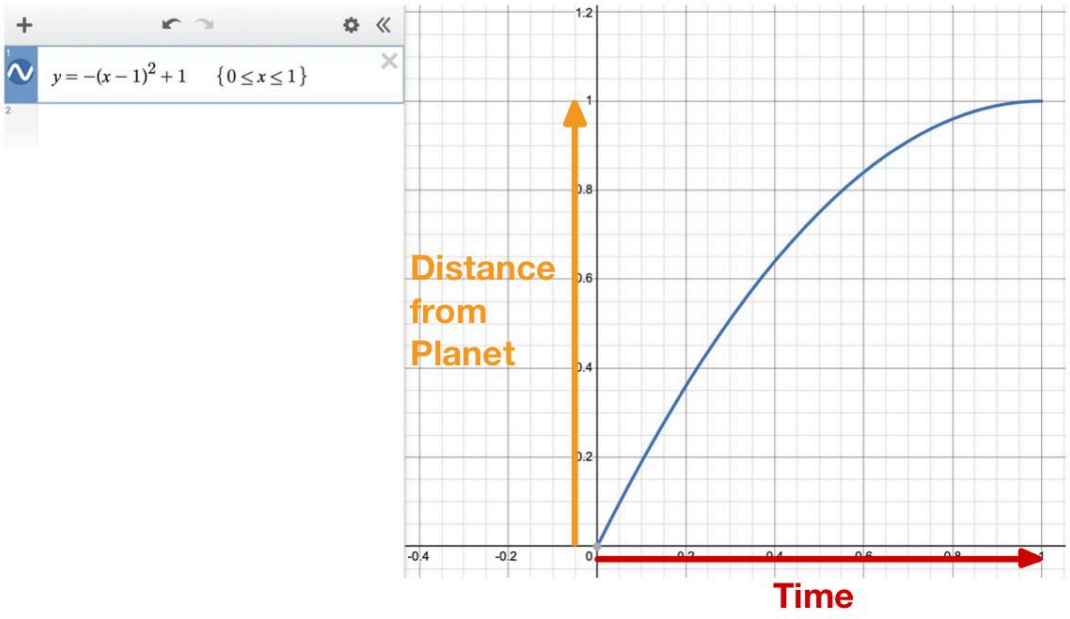
https://www.youtube.com/watch?v=coylMcM-WJQ

I also implemented bloom into the Unity project — but I ran into a few problems. To create the bloom effect I needed to add a Global Volume — however, as the name 'Global' suggests, it applies the bloom effect to all objects in the program. I only wanted the sun to have the bloom effect, not all the other planets. So as a work around, I increased the light intensity of the sun and then decreased the intensity of bloom effect. This change meant the sun still had the bloom effect, but the other planets did not.



29/4/22 — GUI Circles + Zoom In/Out Tweaks	Today I started to implement the GUI system — specifically I added the circles around the planets. Using Photoshop I made my own simple circle sprite, and then imported it into the Unity project. The circles also update to hover over each planet's position, which I am quite happy about. I also made some minor tweaks to the zoom in/out script — I used Mathf.Lerp() to smooth out the camera movements.
30/4/22 — GUI Text + Trails Improvements + Anti-Stacking	Today I added the text next to each planet's GUI circle. It uses the 'bodyName' variable in each planet's 'CelestialBody' script to update its text contents. I also moved the trail function from the 'PlanetsController' script to the 'PlanetsUI' script. I think it's a good decision to organise all the UI functions in this one central script. I also made a minor change to the 'PlanetsController' script — specifically I changed the function Update() to LateUpdate() as I had some issues with the GUI elements updating slowly; this change fixed it. In addition to this, I also implemented an anti-stacking fade function. What I found after running the program was that the text from the GUI elements can stack on top of each other — making the text impossible to read. Today I wrote a function that checks the distance between each element and fades them if they are too close. I also set a priority to each one, so the Sun element will be displayed before other planets, and the Earth element will be displayed before the Moon element.
2/5/22 — GUI Close Fade Out + Fixed Axial Tilt + Fixed Trails	Over the past few days I have been working on adding a function to fade out the GUI elements when zoomed in to a planet. Previously, the GUI circle and text would appear on the screen even if the camera is zoomed all the way in. However now after adding this function, it will fade out when the camera is close. I also fixed each planet's axial tilt. Yesterday, I found that the planet's axial tilt axis doesn't stay constant — when it really should. For some reason the planet's axis would drift around in a circular motion — I arrived at the conclusion that this was due to my implementation of the axial tilt processes. Originally it was placed in the FixedUpdate() function where I used the sin and cos function to rotate the planet. Here is the original code: `// Update planet rotation` `var axialTiltRadians = Mathf.Abs(bodies[i].axialTilt) * (Mathf.PI / 180f);` `var rotationAnglePerDt = 360f / (bodies[i].rotationPeriod / dt);` `bodies[i].transform.Rotate(rotationAnglePerDt * Mathf.Sin(axialTiltRadians),` `rotationAnglePerDt * Mathf.Cos(axialTiltRadians), 0f, Space.World);` To fix this, I moved the processes into the LateUpdate() function (which is called much more often than FixedUpdate). I also swapped transform.Rotate to transform.RotateAround as shown below. I think the reason for this weird behaviour was due to floating point inaccuracies when using Mathf sin and cos. However, with transform.RotateAround this behaviour doesn't happen anymore. Here is the new code:

	`` // Update each planet's rotation float axialTiltRadians = Mathf.Abs(bodies[i].axialTilt) * (Mathf.PI / 180f); float rotationAnglePerFrame = 360f * (simTimeSinceLastFrame / bodies[i].rotationPeriod); bodies[i].transform.RotateAround(bodies[i].transform.position, bodies[i].transform.forward, rotationAnglePerFrame); ``` I also fixed another issue I have recently seen, which was when the trails were turned off and back on. When it was turned off, the trails were no longer updating — so when they were turned back on they would 'jump' to their new location, creating a straight line towards it. To fix this, instead of turning on/off the script, I changed the alpha value of the trail colour (1 = seen; 0 = hidden). This way the trails are still updating even if they are hidden to the user.
3/5/22 — Added Clickable GUI + Camera Transition	Today I fixed up the GUI element grouping as well as implemented a click and highlight feature. When the mouse is hovering over a planet's GUI element (circle and text), it becomes a bit more bright so the user knows which planet is currently being selected. Also, when the user clicks on it, the camera transitions to the respective new planet. This was achieved by creating an invisible button over the GUI elements, which when clicked transitioned the camera. Transitioning the camera was pretty easy — all I had to do was code a function that changes the priority of the free look cameras; Cinemachine does all the rest.
5/5/22 — Fixed Camera Transition Smoothness + Zoom In to Current Planet + Pop-up Travel Text	Today I fixed the camera transition smoothness. I found that when selecting and transitioning to another planet near the edges of the screen, the camera would snap straight towards it — which isn't very smooth at all. After some research, I found that simply changing the blend hint from 'None' to 'Spherical Position' fixed this issue up.  I also added in a feature where you can zoom back in to the current planet by clicking on the GUI elements again. For example, when viewing the earth the user can click the "Earth" GUI element to zoom back in automatically. Unlike the other Cinemachine transitions, I coded this one in myself. To replicate the smoothness that the other transitions have, I used a maths equation to scale the distance over time. Here is an image of the equation below, where the bottom axis is time (0 = start, 1 = end) and the vertical axis is the distance from the planet.

	 I also added a pop-up text that tells the user what planet they are navigating/transiting to. It was pretty easy to add, all I had to do was make the text fade in at the start of the transition, and fade out at the end of the transition — with the name of the planet updated as well. I also imported a font into the project — I found the default text Unity uses to be quite bland and robotic. The new font is called Montserrat and can be downloaded for free off the google fonts webpage: https://fonts.google.com/specimen/Montserrat
7/5/22 — Toolbar + Started Settings Panel	Today I implemented the bottom toolbar, which includes the pause button, the elapsed time and simulator speed counter. The buttons and the toolbar were pretty easy to implement, however for the counters I had to code my own function to change the units next to the value. With this, the counters are always showing the relevant units, so something like 3.154e+8 seconds appears as 1 decade. I also started to make the settings panel and the toggle button — but it is currently blank.
8/5/22 — Settings Panel	Today I finished off creating the settings panel. Using an animator, I made the panel smoothly slide in and out of frame when the toggle button is clicked. I also decided to include a short description underneath each option in the settings menu, allowing the reader to understand what each option does. Since the number of options exceeded the space on screen for the panel, I added a scrollable feature where the user can scroll down the list of options.
10/5/22 — Main Menu + Resolution Options	Today I added the main menu into the program. In the main menu, I coded in 3 buttons: start, settings and exit. With the settings button, I added another menu where the user can change the resolution of the program. I also decided to add an earth in so the start menu doesn't look bland.
11/5/22 — Testing	Today I built and exported the project for testing. During the testing, I found

+ Built Project	an issue with the program — the exit button didn't work. Instead of closing immediately once the button is clicked, the program freezes and stops responding. I did some research into this and found that this issue can be fixed quite easily — instead of writing Application.Quit(), the program can use the internal systems to force quit the program. Here is the new line of code: `System.Diagnostics.Process.GetCurrentProcess().Kill();` Although this does fix the issue, I need to be careful when testing the program in the editor. Unlike Application.Quit() which doesn't work in the editor, this new code does work in the editor — so I need to make sure to save the project before pressing the button; however since this is near the end of the implementation phase, this won't be much of an issue.
12/5/22 — User Feedback	Today I gave my friends the program to test and get feedback on. I sent the program over to them and they each had their own suggestion/feedback for how the program can be improved in the future. I wrote their contributions in the user feedback section above.
14/5/22 — Evaluation	Throughout yesterday and today I have been working on completing the evaluation — which I have since finished today. In the evaluation I made sure to address some of the issues I encountered throughout this project iteration.
16/5/22 — Maintenance	Today I finished writing the maintenance phase. I also found that running the program on MacOS devices had some issues with it — specifically the anchoring of the canvas elements. I have since fixed it and re-built the program.
17/5/22 — Final Touch Ups	Today I went through all my documentation and fixed up anything that wasn't complete or up to scratch.
18/5/22 — Hand In Date	Submitted the project.