**On Orbital Mechanics:**

Premise:

I want to better understand the idea of orbital mechanics and how they apply to flight trajectories and space mission planning. The goal is to make several projects that demonstrate my understanding of orbital mechanics and how they apply to spacecraft. I hope to show that with enough research and through understanding of the physics principles, I can apply what I know to create physics simulations programs, with varying degrees of complexity. Furthermore, I hope that this series of projects will demonstrate my ability to apply what I know about physics to real life space-systems.

*Document guidelines:*

I have chosen a very specific way to lay out this document (this specific way may change as I fill it out, but you can always refer to the change logs of the guidelines to see what is different). Each project will be described initially with a general description, and different aims of the project, at varying levels of difficulty. This level system will serve as a rough guide to how I will go about creating the project and in what order I will complete things, but this is not a set in stone way of completing it. Additionally there will be a short description of how I intend to make the project, (style, language, engine, etc) and why I chose to do it this way.

Each project will also have a log of what I have found to be difficult, what I needed to research, the updates to the project, and anything else that I might see as relevant.

I will also include the physics behind the systems that I am building, to better understand how I will go about implementing the project.

Project 1: 2D Gravity simulation

For the first project I aim to create a solar system with fixed orbits, and realistic sizes and distances, so that I may simulate the trajectory of space craft through the solar system.

This project will be created in a Unity 2D project. I have chosen this engine to create the project, but I will not be relying too heavily on any built-in physics systems that would complete the work that I want to understand for myself. What I mean by this is all the built-in physics mechanics such as moving an object, applying velocity or force will be used, but any built-in components that would allow me to calculate a trajectory, or would otherwise take away from the exploratory aspect of this project, will not be used. A Unity project was primarily chosen to facilitate the rendering and graphical implementation of the project, so I can focus on the simulations and physics.

*Level 1 Aims:*

* To understand gravitational attraction and how it affects orbital trajectories.
* Simulate a space craft moving through the solar system to different planets

*Level 2 Aims:*

* Be able to accurately simulate a near-Earth orbit, and an Earth-Moon orbit
* Be able to adjust the orbit along the way, simulating propulsion burns and new trajectories mid-flight

*Level 3 Aims:*

* Re-create real-world space flights with accurate trajectories and rocket burns throughout the flight (such as Apollo 8 or Apollo 11)

*Level 4 Aims:*

* Simulate the landing and launching of a spacecraft from the surface of a terrestrial body (Moon landing, Mars landing etc)

*Logs:*

2/07/25

#1: Immediately I have run into a problem of scale, as is typical when working with such vast distances. The sizes of each of the planets are quite well scaled with each other, but it is their distance from the Sun that is causing an issue. The furthest plant, Neptune, sits at approximately 80,000 meters away from the sun at (0, 0) when using a 56,000:1km scale. Putting the sun at roughly 12.5m in radius.

This scale system does work, and is accurate, but I worry how it might affect the performance of the system if it must calculate large far-off numbers.

A screenshot of a video game

AI-generated content may be incorrect.



#2: The scale I am working with seems to be working well for the time being and I have set up a very basic, circular fixed orbit system using a basic point on circle equation, where:

A computer screen shot of a program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.To calculate the position of the planets as they orbit the Sun. Right now, you can just input the radius and angular velocity, but I want to be able to just input the radius and period of the orbit and calculate the velocity from there, (Kepler would be proud). Here is what the orbit script looks like now:

I now want to apply the updated version, but first I want to have the fixed orbit calculate the position around an ellipse, not a circle.

A screen shot of a computer program

AI-generated content may be incorrect.#3: The orbits of planets are now ellipses, and more realistic. I also calculate the orbit location for each planet by determining the major and minor axes of their orbit from their mass and orbital period.

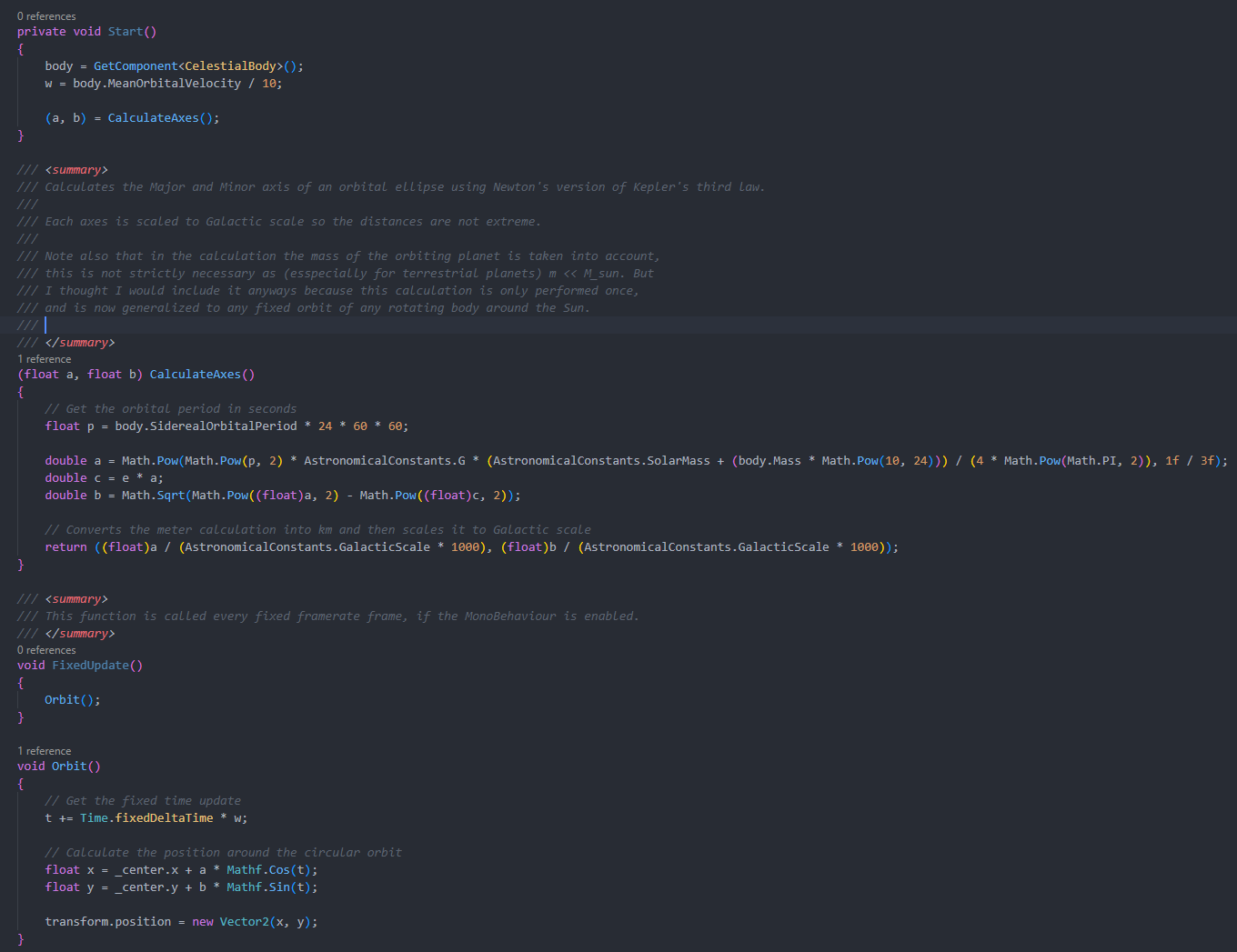
This is made easy with a simply celestial body class, that stores basic information about the celestial body:

Additionally, I added a astronomical constants static class, to store values of common constants that will be used throughout the project.

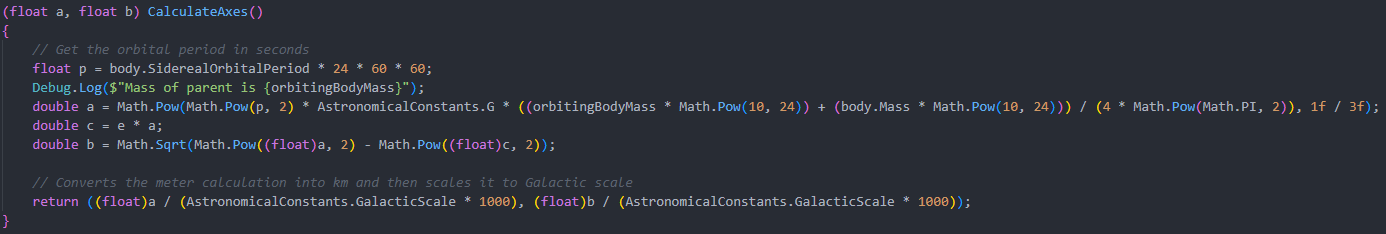
The updated ellipse equations now look like:

Where is the centre of the ellipse and are the major and minor axes respectively.

The next step is to make the planets rotate on their axes. For the sake of simplicity and keeping the simulation simple, each planet will rotate only in the xy-plane.



#4: I have generalized the fixed orbit functionality so that any object in the solar system will be able to orbit a body, either the Sun or something else, like the Moon around the Earth.



#5: I have added basic sprite work to the planets. I have decided that although the sprite work will go unnoticed on the scale of the universe, I want to create a separate scene for each individual planetary system. This scene will have a larger smaller ratio so that the planets and their satellites can be more easily observed.

System scales are now separated in their own Galactic scale classes to accommodate for different scales in different systems.

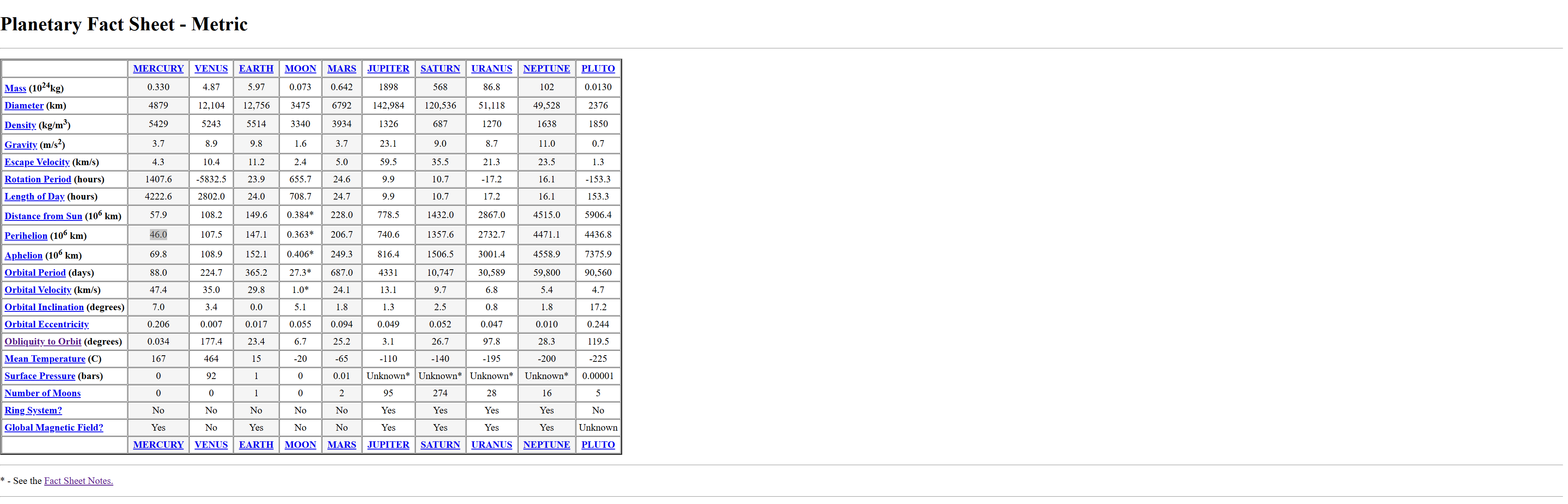
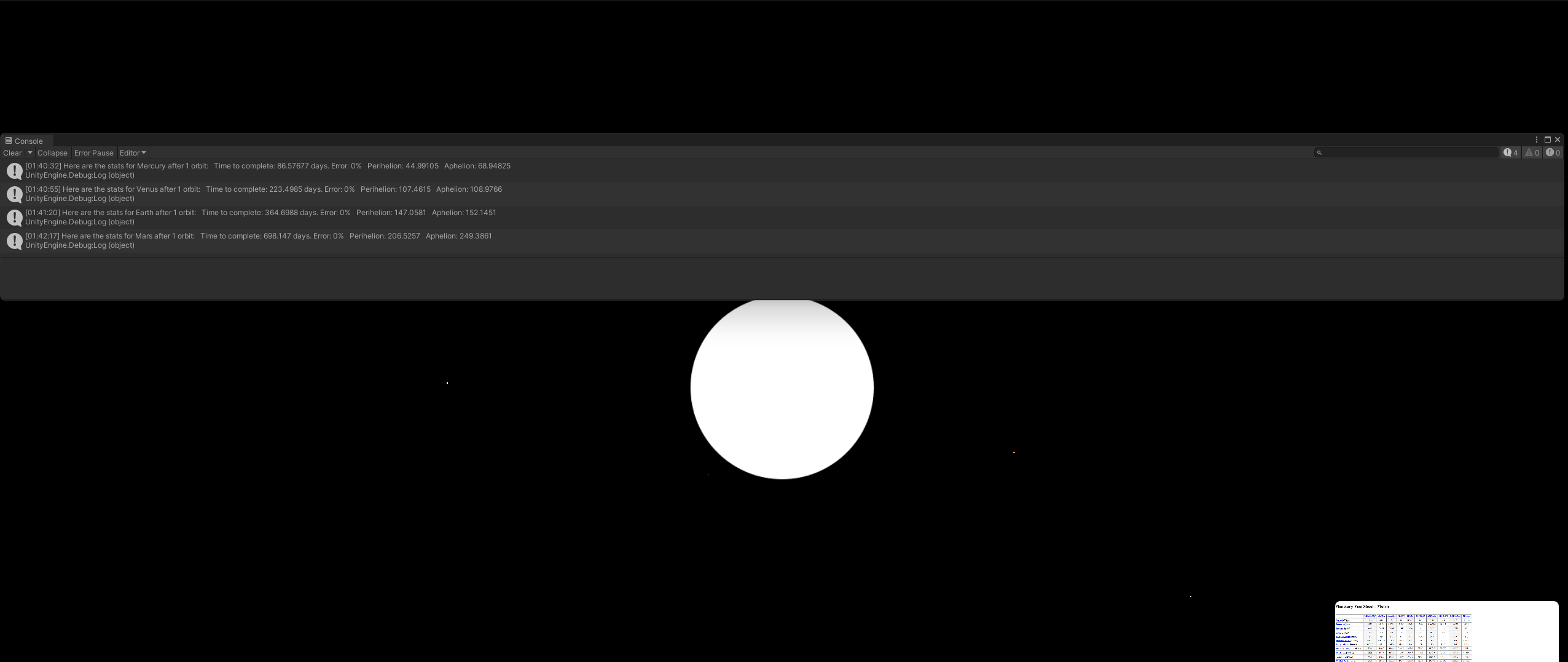
I have also corrected the orbital velocity of each planet, using the Semi-Major axis of each around the sun to calculate the appropriate angular velocity. This far more accurately reflects the actual speed they are travelling at any one time and isn’t obviously wrong by simply plugging in the orbital speed into the angular speed.

#6: Scale continues to be an issue, so I have decided that every calculation will be accurate to scale, (distance and mass) so that each celestial body orbits exactly how it would in the system. However, each celestial body will be represented as being much larger than it’s actual size, to better visualize where each celestial body is in the system. This will only be implemented for the Solar System, where the distances are quite vast, each planet will be much larger than what it is, and the Sun much smaller, so we can see what we are doing.

Each system to account for scale will now be separated into a main body and its immediate subclass of satellites. For example, the solar system will have the main body be the Sun, then the immediate subclass of satellite would be every planet. Similarly, for any planet system the main body be the planet, and the immediate subclass would be each of its Moons.

#7: Added some basic camera controls and random starting positions of the system (so each body doesn’t just start in a line).

#8: I have corrected the position of the Sun in the ellipse of each orbit. I have also taken into account the longitude of perihelion of each of the planets. This ensures that every orbit and elliptical is correctly placed as it would be in the solar system (viewing it top down). I verified the time it takes for a body to complete an orbit, as well as its perihelion and aphelion with NASA’s planetary fact sheet to confirm everything was working and positioned correctly:



Using this simple tracker:

A screenshot of a computer program

AI-generated content may be incorrect.

This confirms that up until this point using just the mass of the Sun and the planets, and with the correct elliptical positioning, I can accurately simulate the orbital motions of the planets.

Note that the Jovian planets were originally not evaluated for correctness, as the margin of error would be larger for planets that take a longer time to orbit the Sun. However, for good measure here they are:

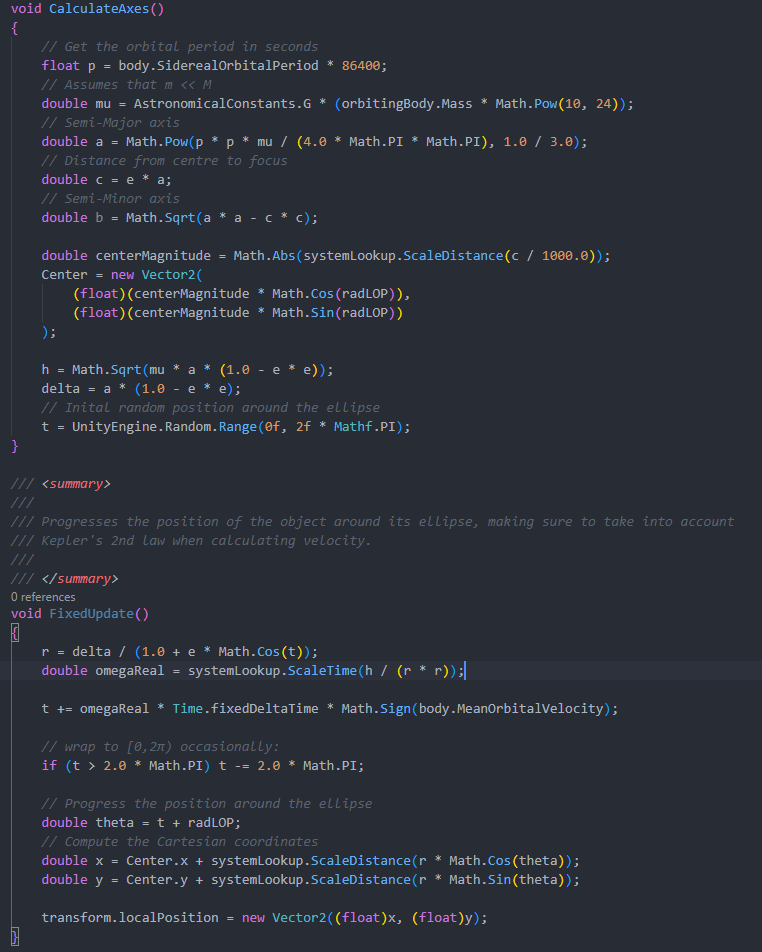
A screenshot of a computer

AI-generated content may be incorrect.

Still very accurate, I stand corrected.

The next step will be to create the Earth-Moon system, and place an object between the two and calculate the magnitude and direction of the gravitational attraction of each body on that object. This will be easy enough with the Earth and the Moon; however I also need to consider the position of the Sun in this system, which will be more difficult.

I still need to implement planetary axel rotation, but I will tackle that with the Earth-Moon system too, as it is the first one where the rotation will be noticed.

#9: The first quality of life fix was to adjust the way fixed orbits were calculated. I wanted to consider the ellipses longitude of perihelion when creating the ellipse, as well as applying Kepler’s 2nd law of orbital motion and increasing the velocity of the orbiting body when it got closer to the planet. To accomplish this, I decided to rewrite some of the fixed orbit script to still calculate the full ellipse and then progress the body through the ellipse through radians alone. This makes it easier to determine the position of the body on the ellipse, instead of worrying too much about scaling issues and more complex distance and position calculations.

This new method also means that I can effectively consider the angular velocity of the body and calculate the angular velocity directly from the position on the elliptical.

#10: I have created an Earth-Moon system, to test out gravitational attraction and orbits. So far I have added a method in the systems script to determine the acceleration of a body at any point within the system. This is accomplished by having the system controller store each body in the system, which automatically stores their mass and position. Then I iterate over each body and add their masses like so:

Where we sum the mass of each body over the distance to that body at any point within the system.

A computer screen shot of a program

AI-generated content may be incorrect.

The acceleration is stored as a double2 to increase the precision of the calculations and mitigate the need to determine the angle of acceleration by separating the x and y components. The acceleration is scaled to our distance and time scale by realising the following:

So, by taking our acceleration to the product of , we convert our acceleration into our time and distance scale. Note here that and are our scales for time in seconds and distance in meters respectively (I use a scaling system where is in but for clarity it is in meters here, also something I want to change later).

As stated at the start in the project guidelines I did not want to use any built in Unity physics systems like Vector2.AddForce and others. This was made to fully develop my understanding of the equations and systems of orbital mechanics without relying on the built in physics engine.

We don’t need the force directly when calculating the gravitational movement of the body, to translate the force of gravity into motion we can do the following:

Where denotes the acceleration at some time in the program. Once we have the acceleration and it is scaled, we can progress the position of the transform with:

Like so:

