E.T.S. de Ingeniería Industrial,

Informática y de Telecomunicación

3 Dimensional N-Body Simulator

Grado en Ingeniería Informática

Trabajo Fin de Grado

Author: Juan Paños Basterra

Advisers: Jesús Palacián Subiela

Patricia Yanguas Sayas

Pamplona, 10/02/2022

ACKNOWLEDGEMENTS:

There are many people that I would like to thank for making this project possible.

Firstly, my good friends the “Programa2”, which have motivated and helped me throughout my university studies and never lost faith in me. They were the ones who convinced me to sign up to one of the projects the University offered, which is what lead me into this project, and I will always be thankful for that.

My family and many other friends have also been instrumental to the success of the project, helping me when I was losing hope and pushing me to achieve my goals.

To my good friend Franklin and my dad, who taught me not to look at my feet, but to instead look at the sky.

To all the professors at UPNA who made me interested in programming and developing software and helped me gain the skills with which I built this project.

To Mamen Peña, and the incredible work she did on a simulator for the N-body problem, which was of immense help and inspiration for this project.

And especially to professors Jesús Palacián and Patricia Yanguas, for entrusting me with this project, guiding me along the way and helping me learn so many new and different things, without their guidance and patience this project could not have been possible.

ABSTRACT:

This thesis is based on the development of a simulation software for the N-body problem, attempting to provide a modern program to improve upon existing simulators.

To achieve this, the N-body problem will be explained, along with other related concepts such as numerical integrators. The development of the simulator will also be discussed, studying the selected framework, design principles followed, program structure and in-depth code analysis.

Finally, the results of the project will be studied, analyzing aspects such as performance, resulting UI and several improvements will be proposed.

The results of this project are publicly available in **https://github.com/panosjuanis/3D-N-body-simulator**, where anybody can download the developed software to utilize it, read the documentation, or contribute to it in whichever way they like.

KEYWORDS:

N-body, Gravity, Unity, 3D, Simulator, Numerical Integrator, C#

Index

1. Introduction 7
1.1. Introduction to the N-body problem 7
1.2. Objective 8
1.3 Existing simulators 10
2. Equations of the N-body problem 10
3. Numerical integrators 15
3.1. Introduction 15
3.2. Euler 18
3.3. Leapfrog 19
3.4. Runge Kutta 4 20
3.5. Multi-step methods 22
3.5.1. Adams-Bashforth 22
3.5.2. Adams-Moulton 22
3.5.3. Adams-Bashforth-Moulton 23
4. Development of the program 24
4.1. Selection of framework 24
4.1.1. Matlab 24
4.1.2. First approximation in Python 26
4.1.3. Selection of Unity and C# 29
4.2. User Interface (UI) 30
4.2.1. Design philosophy 30
4.2.2. Resulting UI 33
4.2.2.1. Main Menu 33
4.2.2.2. Load Simulation 34

4.2.2.3. New Simulation 35

4.2.2.4. Simulation 37

4.3. Programming of the simulator 41

4.3.1. UI programming 41

4.3.1.1. Main menu 41

4.3.1.2. New Simulation 42

4.3.1.3. Load Simulation 43

4.3.2. Simulation programming 44

4.3.2.1. Main scripts 45

4.3.2.1.1. SimManager 45

4.3.2.1.2. CanvasManager 48

4.3.2.2. Other scripts 50

5. Performance and optimization 51

5.1. Computational costs 51

5.1.1. Effect of N 52

5.1.2. Visual settings 53

5.1.3. Numerical integrators 54

5.1.3. Energy/Angular momentum graph 55

5.1.4. Improving performance 56

5.2. Resulting performance 57

5.2.1. Effect of hardware 57

5.2.2. CPU, GPU and RAM 58

6. User guide 59

6.1. Requirements and installation 60

6.2. Instructions and tips 60

7. Examples 61
7.1. Simulation examples 61
7.1.1. Real world examples 61
7.1.1.1. External solar system 62
7.1.1.2. Trojan asteroids 63
7.1.2. Theoretical solutions 63
7.1.2.1. Figure 8 64
7.1.2.2. Broucke’s periodic solutions 65
8. Conclusions 66
8.1 Improvements 67
8.1.1 Optimizations 67
8.1.2 Extra features 68
8.1.3 New simulators 69
9. Bibliography 71

1. Introduction

1.1. Introduction to the N-body problem

In physics, the N-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally. Solving this problem has been motivated by the desire to understand the motions of the Sun, Moon, planets, and visible stars ever since humans were able to admire the cosmos.

The N-body problem has been studied since antiquity, where the interest was placed on the movement of the planets on the night sky. Different theories were proposed by the Greeks to explain their movement. Over the centuries the conception of the universe evolved until in the XVII century, Isaac Newton formulated the theory of universal gravitation. This established the gravitational attraction between two bodies, which will be the basis of our application.

Any discussion of planetary interactive forces has historically started with the two-body problem, which consists of predicting the motions of two bodies interacting with each other. This problem was solved analytically by Johan Bernoulli in the XVII century. However, for systems where $N > 2$ no analytical solution is possible, and therefore an approximation of the solution is the only viable option. The mathematical techniques used for approximating the solution are called numerical integrators, which differ in how they approximate the solution and in their quality and computational cost.

The N-body problem is perfectly suited to be solved by computers because even for systems where N is small, millions of calculations need to be done to come up with the solution, therefore rendering any human calculations impossible. Nevertheless, when determining a solution, computational power is not the only metric that ensures quality.

Therefore, the selection of quality numerical integrators is an essential part of obtaining a solution.

1.2. Objective

The purpose of this project is the development of a **simulator** for the N-body problem, which approximates the solution of a system of N bodies and renders them in a **3-dimensional** space where the user can interact with different parameters and observe the simulation in real time. The user shall be able to select the starting parameters, as well as the **numerical integrator** to be used, which will greatly vary the resulting simulation as will be discussed later.

The focus of the project will be on developing an application that is **easy to use**, with an intuitive **User Interface** (UI) to allow the user to interact with the different parameters to learn how they affect the simulation of the N-body problem, as well as experiment with a variety of pre-generated simulations and allowing for the creation of new ones. All being managed by a single application that is easy to download, install and utilize.

This thesis is divided in seven sections, firstly, the equations of motion that describe the N-body problem will be studied, and the need to approximate a solution will be explained.

The second part will cover the **numerical methods** used to approximate the solution of the general N-body problem, analyzing the difference between them, both in how they calculate an approximation and in their quality and computational complexity.

The following section will be devoted to the **development of the program**, we will cover topics such as the selection of the framework used to create the software, essential in

any software project, the design of the **User Interface** (UI) and the principles we have based our development on, and finally we will study the **programming** of the different elements, both the **UI** and the main **Simulation** codes that manage different aspects of our program.

In the fourth part **performance** will be discussed, analyzing metrics such as Frames Per Second (FPS), CPU, GPU and RAM. Several **optimizations** will also be studied, some that have been implemented as well as some that have been proposed to tackle performance issues.

After we have analyzed the obtained performance, a user guide will be proposed, explaining how the user can **install** the software as well as a brief set of **instructions** and **tips**, that the user should be aware of in order to have a satisfactory experience when using the program.

The sixth section will help visualize some **simulation examples** that can be executed by the simulator, studying both real world examples such as a solar system configuration and theoretical solutions that have been proposed, such as the renowned Figure 8 configuration.

The conclusions of the development of the project can be found in the last segment, together with a list of possible improvements proposed and the references used for the project.

1.3 Existing simulators

When considering the development of a N-body simulator, it is important to examine the existing software. While a more in-depth analysis will be performed in a later chapter, several simulators were tested and scrutinized and some of the detected issues common among them were:

- Lack of an intuitive User Interface.
- Lack of varied integration methods.
- High complexity to download and utilize
- Usage of only two dimensions
- Lack of user interactivity

In this project, we have attempted to solve most of these issues by designing an **intuitive UI**, providing the user with several **integration methods**, facilitating installation on **multiple platforms** with a streamlined application, allowing the user **3-dimensional** movement and providing different tools for the user to **interact** with the simulation.

2. Equations of the N-body problem

When a N-body system is discussed, interacting through their mutual gravitational attractions, and confined in a limited special region, we are referencing the appropriate mathematical model to represent some aspects of real systems, such as the Solar system and other galaxies.

Let us consider N(>1) point masses moving in an inertial reference system R^3, where the only forces acting on them are their mutual gravitational attractions.

If the ith particle has a position vector $\vec{q}_i$ and mass $m_i > 0$, because of the second law of Newton and the law of gravity, we have the motion equation for the ith particle:

$$m_i \ddot{\vec{q}}_i = \sum_{j=1}^{N} \frac{G m_i m_j (\vec{q}_j - \vec{q}_i)}{\left\| \vec{q}_i - \vec{q}_j \right\|^3}, (i \neq j),\ \ddot{\vec{q}}_i = \frac{d^2 \vec{q}_i}{dt^2},$$

Having:

$$G = 6.672^{-11} \frac{m^3}{kg * s^2}.$$

If the negative potential function, U is introduced:

$$U = \sum\nolimits_{1 \le i < j \le N} \frac{G m_i m_j}{\left\| \vec{q}_i - \vec{q}_j \right\|^3},$$

The motion equation can be rewritten:

$$m_i \ddot{\vec{q}}_i = \frac{\partial U}{\partial \vec{q}_i}.$$

The system of Ordinary Differential Equations (ODEs) defines the N-body problem. Since the initial positions (q), and velocities ($\dot{q}$) will be introduced as data, the problem corresponds to the category of **initial value problems** for ODEs. Furthermore, if we assert that the bodies move exclusively because of the effect of gravity from the other bodies, the problem conserves the **total energy**, and can be approached as a **Hamiltonian** problem.

The Hamiltonian formulation does not add anything to the physical laws, but it provides a very useful formalism [1]. The basic idea of the Hamiltonian formulation is to write the equations of motion in terms of positions and momenta, instead of positions and their derivatives.

We denote $\vec{q} = (\vec{q}_1, \vec{q}_2, \vec{q}_3, \dots, \vec{q}_N) \in R^3$ as the position vector, with this new nomenclature, the vectorial form of the motion equation is:

$$M * \ddot{\vec{q}} - \nabla U(\vec{q}) = 0,$$

Having $M = diag(m_1, m_1, m_1, \dots, m_N, m_N, m_N)$ as the mass vector and ∇ as the gradient operator.

Analyzing the last equation, the definition of U as the negative potential is clear. The potential function V is defined as:

$$\vec{F} = -\nabla V.$$

From this definition we find the following relation $U = -V$. Furthermore, by definition, since there exists a potential function, the system is conservative.

To use the Hamiltonian formulation, it is necessary to introduce momentum vectors.

We define $\vec{p} = (\vec{p}_1, \vec{p}_2, \dots, \vec{p}_N) \in R^{3N}$ as the lineal momentum vector, which verifies:

$$\vec{p} = M * \dot{\vec{q}},$$

Having $\vec{p}_i = m_i * \dot{\vec{q}}_i$ as the momentum of the ith particle.

With these definitions, the equations of motion can be written as:

$$\dot{\vec{q}} = \nabla H(\vec{p}) = M^{-1} * \vec{p},$$

$$\dot{\vec{p}} = -\nabla H(\vec{q}) = \nabla U(\vec{q}),$$

Or in components:

$$\dot{\vec{q_i}} = \frac{\partial H}{\partial \vec{p}_i} = \frac{\vec{p}_i}{m_i},$$

$$\dot{\vec{p_i}} = -\frac{\partial H}{\partial \vec{q}_i} = \frac{\partial U}{\partial \vec{q}_i} = \sum_{j=1}^{N} \frac{m_i m_j(\vec{q}_j - \vec{q}_i)}{\|\vec{q}_i - \vec{q}_j\|^3}.$$

where the Hamiltonian, or the total system energy of the particle system is:

$$H = \frac{1}{2}\vec{p}^T * M^{-1} * \vec{p} - U = \sum_{i=1}^{N} \frac{\|\vec{p}_i\|^2}{2m_i} - U,$$

With the first term of the equation being the **total kinetic energy**:

$$T = \frac{1}{2}\sum_{i=1}^{N} \frac{\|\vec{p}_i\|^2}{m_i},$$

And the second term being the **potential energy** of the system, $V = -U$.

The **Hamiltonian** is a constant of the considered equation system, since:

$$\frac{\partial H}{\partial t} = \frac{\partial H}{\partial \vec{p}}\,\dot{\vec{p}} + \frac{\partial H}{\partial \vec{q}}\,\dot{\vec{q}} = \dot{\vec{q}} * \dot{\vec{p}} + \left(-\dot{\vec{p}} * \dot{\vec{q}}\right) = 0,$$

The N-body problem is a system of 3N equations of second order $(q_{i_x}, q_{i_y}, q_{i_z})$ in the Newtonian formulation, and of 6N order equations $(q_{i_x}, q_{i_y}, q_{i_z}, p_{i_x}, p_{i_y}, p_{i_z})$ in the Hamiltonian formulation. To solve this problem, we need to know **6N independent first integrals**. However, $\forall N \geq 3$, only ten first integrals are known (In 1887, H.Burns proved that no more independent integrals exist [2]). One of them is the Hamiltonian, which has been demonstrated previously. The other nine remaining integrals are shown next:

The **center of mass** in the system is defined as:

$$\vec{C} = \frac{\sum_{i=1}^{N} m_i \vec{q}_i}{\sum_{i=1}^{N} m_i},$$

The **total lineal momentum** is:

$$\vec{L} = \sum_{i=1}^{N} p_i,$$

Deriving both vectors with respect to time we obtain:

$$\vec{C} = \sum_{i=1}^{N} m_i \dot{\vec{q}}_i = \vec{L} \, ,$$

$$\vec{L} = \sum_{i=1}^{N} \dot{\vec{p}}_i = \sum_{i=1}^{N} -\frac{\partial H}{\partial \vec{q}_i} = \sum_{i=1}^{N} \frac{\partial U}{\partial \vec{q}_i} = \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{m_i m_j (\vec{q}_j - \vec{q}_i)}{\left\| \vec{q}_i - \vec{q}_j \right\|^3} = 0 \, ,$$

Therefore, we have:

$$\ddot{\vec{C}} = \dot{\vec{L}} = 0 \, ,$$

If we integrate the previous equation, we obtain

$$\dot{\vec{C}} = \vec{L}_0 \qquad \vec{C} = \vec{L}_0 t + \vec{C}_0 \, .$$

As we can see, the center of mass vector, which has a uniform line movement, offers us three motion integrals, and the lineal momentum vector provides us with three more constants. Taking into account that the Hamiltonian is another motion integral, we only have three more first integrals to study, which come from the **total angular momentum**:

$$\vec{O} = \sum_{i=1}^{N} \vec{q}_i \times \vec{p}_i \, .$$

Applying these motion integrals, we can reduce the dimension of the system to 6N – 10. The ten constants of motion, $\vec{L}_0, \vec{C}_0, \vec{O}$ and H are functions of the positions and velocities of the N point masses and of time and their independence from each other can be demonstrated [3], through two additional transformations, we can obtain two more constants, bringing our total to twelve. Therefore, for any system where $N \geq 3$, we simply cannot meet $6N \leq 12$, therefore no analytical solution is possible.

3. Numerical integrators

3.1. Introduction

The integration of the N-body problem can be seen as an initial value problem for **Ordinary Differential Equations** (ODEs). The ODE that models the problem cannot be solved exactly. Therefore, different approximation methods, called **numerical integrators** or **numerical methods**, are used to attempt to solve the problem with minimal errors. Such methods do not produce a continuous solution, but instead generate approximations for specific discrete points in time.

The initial value problem for a first ODE implies finding a function y(t) that satisfies:

$$\frac{dy(t)}{dt} = f\big(t, y(t)\big) \,,$$

together with the initial condition:

$$y(t_0) = y_0 \,,$$

A numerical solution for this problem generates a sequence of values for the independent variable: **t**, and the corresponding sequence of values for the dependent variable **y**. So for every value **i** we have:

$$i = 1, 2, 3..N \qquad\qquad y_i \approx y(t_i),$$

Where $\mathbf{y_i}$ is an approximation for the value of y in $\mathbf{t_i}$. Therefore we cannot obtain a continuous approximation to the solution y(t), but instead a series of approximations to that solution for a set of given points (i).

In the n-body problem we have that $f(t, y) = f(y)$, since the function is not dependent on time, only on the variables (y). We also need to consider that for the n-body problem:

$$f(y) = \frac{\partial H}{\partial \vec{p}} - \frac{\partial H}{\partial \vec{q}} \,.$$

Given a time interval [t_0, t_f], we call **h** (also referred to as **DT** in the simulator) the **timestep**, which is the length of time separating t_i from t_{i+1}. This parameter is essential when creating a simulation since, as we will see later, it directly impacts the way in which the solution is calculated.

When making an approximation for a value y_i, different techniques are used, the first is called **single-step**, this is because when approximating the solution for the value t_{i+1}, only the previous point (t_i) is used. The other type of methods we will study are called **multistep**, since when approximating a solution for t_{i+1}, several previous points (t_i , t_{i-1}, t_{i-2},... t_{i-n}) are used.

Multistep methods are generally regarded as obtaining better solutions because by using the information contained in previous steps, the predicted value is closer to the function. On the contrary, single-step methods only utilize the information of the previous calculated value, and generate less optimal results, though by eliminating these extra calculations they manage to decrease their computational costs. [4]

These methods can also be either **implicit** or **explicit**. **Explicit** integrators are based on formulas that predict the value for the function in t_{i+1} from the value of the function and its derivative in t_i and previous values (t_{i-1} , t_{i-2}...), resulting in:

$$y_{i+1} = f(y_i, y_{i-1}, \dots, t).$$

Meanwhile when we are talking about **implicit** integrators to calculate the value of y in t_{i+1} the derivate for that time point needs to be known, resulting in:

$$y_{i+1} = f(y_{i+1}, y_i , y_{i-1}, \dots, t).$$

Then we have come across a system of non-linear equations that need to be solved iteratively.

Explicit integrators are faster because they do not require solving an implicit equation with the Newton method, or with other techniques, however, implicit integrators offer approximations of higher quality than explicit ones.

Previously we have seen how the N-body problem can be solved through a second-order ODE and through two first-order ODEs, therefore numerical integrators that are either second-order or first-order will help us obtain a solution.

Since numerical methods produce approximations of varying quality, to compare the efficiency of different integrators, we use the **truncation error**, there are two kinds:

- **Local truncation error** (τ_n) is the error that our approximation function, A, causes during a single iteration, assuming perfect knowledge of the true solution at the previous iteration. It is computed from the difference between the left and the right-hand side of the equation for the increment:
$$y_n \approx y_{n-1} + hA(t_{n-1}, y_{n-1}, h, f),$$
$$\tau_n = y(t_n) - y(t_{n-1}) - hA(t_{n-1}, y(t_{n-1}), h, f),$$
We say that a numerical integrator has order $\boldsymbol{p}$ if the local truncation error is $O(h^{p+1})$, the higher the order, the better the integrator approximates the solution.
- **Global truncation error** is the accumulation of the local truncation error over all the iterations, assuming a perfectly accurate solution at the initial time step. The global truncation error e_n at time t_n is defined by:
$$e_n = y(t_n) - y_n,$$
$$e_n = y(t_n) - (y_0 + hA(t_0, y_0, h, f) + hA(t_1, y_1, h, f) + \cdots$$
$$+ hA(t_{n-1}, y_{n-1}, h, f).$$

Local and global truncation errors are closely related, and we always seek to minimize them when selecting our numerical integrators.

Next, we will analyze the different kinds of numerical integrators used to approximate the solution to the N-body problem that have been implemented in our simulator, as well as the differences between them in complexity, quality and computational costs.

3.2. Euler

One of the most basic **numerical integrators** is the **Euler** method, it does not have great utility when trying to obtain a reliable approximation of the solution, but it does serve as a good introduction to understand these methods.

The way the Euler numerical method approximates a solution is the following:

$$w_0 = \alpha ,$$

$$w_{i+1} = w_i + h * f(t_i, w_i) ,$$

$$\forall i = 0, 1, \dots N ,$$

Where $\boldsymbol{\alpha}$ is the initial condition (y_0) and $w_i \approx y(t_i)$ is the approximation of y for t_i.

After observing the method, we notice that it is **implicit** and **single-step**. The Euler method approximates the solution with its tangent as can be seen in image 1.

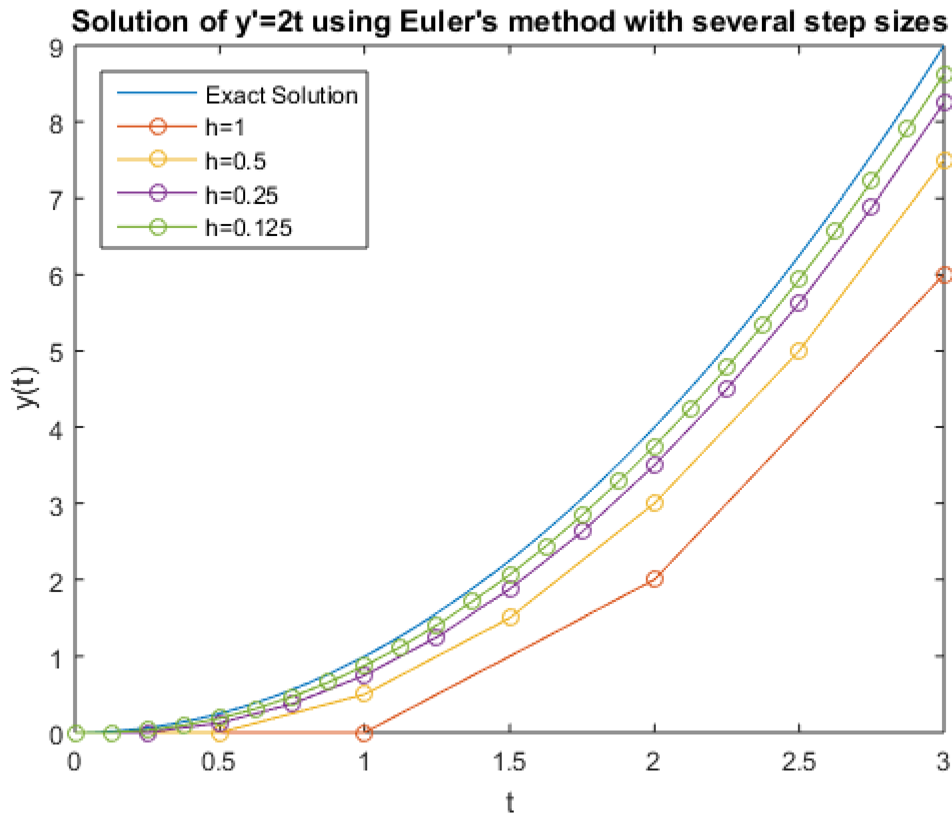


Image 1. Solution to the y' = 2t equation using Euler's method with several step sizes (**h**).

As we can observe, the smaller the value of **h**, the bigger the accuracy we obtain, this will be a recurring factor in the other integrators we are going to examine.

3.3. Leapfrog

The Leapfrog integrator is a **second-order** method with a similar velocity to **Verlet's** method, which is a variant of Verlet integration [5]. Leapfrog integration is equivalent to updating positions ***x(t)*** and velocities ***v(t)*** at interleaved time points, in such a way that they "leapfrog" over each other.

The leapfrog algorithm for updating position and velocity is the following:

$$a_i = A(x_i)\,,$$

$$v_{i+\frac{1}{2}} = v_i + a_i \frac{h}{2}\,,$$

$$x_{i+1} = x_i + h * v_{i+\frac{1}{2}},$$

$$v_{i+1} = v_{i+1/2} + a_{i+1}\frac{h}{2},$$

Where x_i is the **position** at step i, $v_{i+1/2}$ is the **velocity** (first derivative) of x at step $i + 1/2$, a_i is the **acceleration** (second derivative) of x, at step i and h is the **timestep**.

When it comes to mechanics problems, leapfrog integration has two major advantages. The first is the Leapfrog method's reversibility in time. To arrive at the same starting location, one can integrate forward n steps and then reverse the direction of integration and integrate n steps backwards. The second advantage is that it is **symplectic**, which means that it conserves the (modified) energy of the system. Many other integration schemes, such as the (order-4) Runge–Kutta method, do not conserve energy and allow the system to drift significantly over time.

3.4. Runge Kutta 4

The **Runge-Kutta** methods are **single-step** families of methods which can be either **explicit** or **implicit**. These methods can have varying orders (4, 8, 12…). One of the most common ones, and the one that has been implemented into the simulator, is the **Runge-Kutta 4** (RK4) method, which is **explicit**.

Its equations are the following:

$$w_0 = \alpha,$$

$$k_1 = h * f(t_i, w_i),$$

$$k_2 = h * f\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1\right),$$

$$k_3 = h * f\left(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2\right),$$

$$k_4 = h * f(t_{i+1}, w_i + k_3),$$

$$w_{i+1} = w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,,$$

$$\forall i = 1, 2, \dots N-1\,.$$

Here w_{i+1} is the RK4 approximation of y_{i+1}. It is determined by the previous value w_i plus the weighted average of four increments, where each increment is obtained by multiplying h with a slope estimating function f. When averaging the slopes, greater weight is given to the two middle slopes.

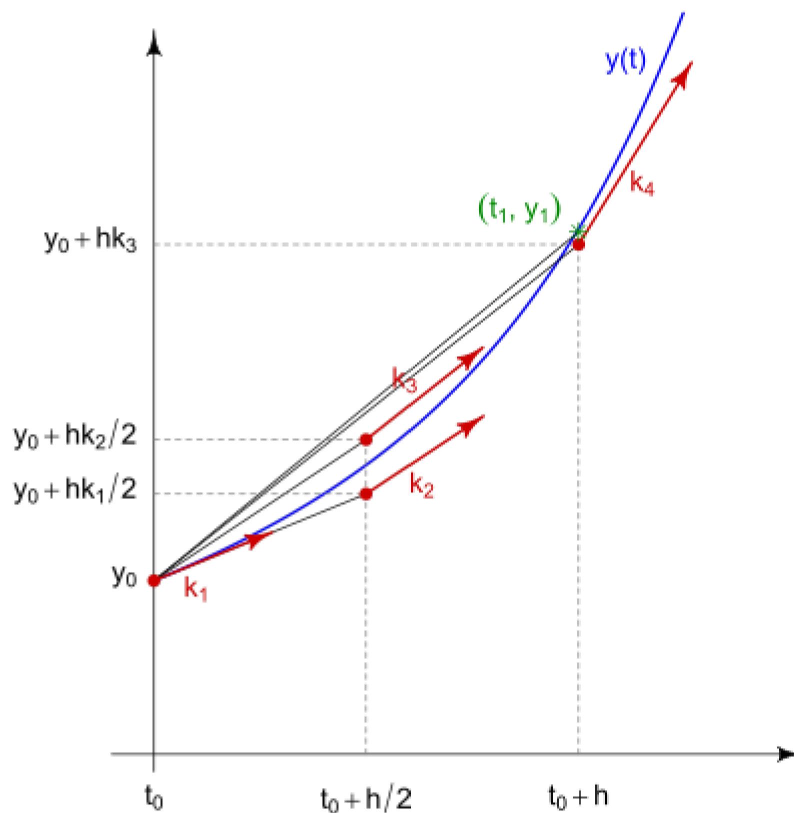


Image 2. Graph showing the approximation of **y(t)** using **Runge Kutta 4** method.

The bigger effort when applying the Runge-Kutta methods is the evaluation of f. The method RK4 requires four evaluations per step, and its local truncation error is $O(h^4)$ as long as the solution $y(t)$ has five continuous derivatives.

There are many Runge-Kutta methods, some are implicit and of higher order (their local and global truncation errors are smaller), but require more function evaluations and the solving of implicit equations with the Newton method.

3.5. Multi-step methods

The previous methods were **single-step** methods, where the approximation of w_{i+1} was made using only the previous value w_i. However, we know that many previous approximations have been calculated before w_i, and ignoring them results in the loss of valuable information, it is therefore reasonable to utilize methods that use more points previously calculated (w_i, $w_{i-1}, w_{i-2}, \dots w_0$).

3.5.1. Adams-Bashforth

This is an **explicit** method whose equations are the following:

$$w_0 = \alpha_0,\ w_1 = \alpha_1, w_2 = \alpha_2, w_3 = \alpha_3\ ,$$

$$w_{i+1} = w_i + \frac{h}{24}\left[55f(t_i, w_i) - 59f(t_{i-1}, w_{i-1}) + 37f(t_{i-2}, w_{i-2}) - 9f(t_{i-3}, w_{i-3})\right],$$

$$\forall i = 1, 2, \dots N-1.$$

This method involves 4 steps, as can be seen by the number of initial conditions required to execute it. In order to calculate these another method is used, for example a **Runge-Kutta 4**.

The local truncation error is $\tau_{i+1}(h) = \frac{251}{720}\, y^{(5)}(\mu_i)h^4$ for any $\mu_i\ \in (t_{i-3}, t_{i+1})$.

3.5.2. Adams-Moulton

An **implicit multi-step** method is the Adams-Moulton method in 3 steps. Its equations are the following:

$$w_0 = \alpha_0,\ w_1 = \alpha_1, w_2 = \alpha_2\ ,$$

$$w_{i+1} = w_i + \frac{h}{24}\left[9f(t_{i+1}, w_{i+1}) + 19f(t_i, w_i) - 5f(t_{i-1}, w_{i-1}) + f(t_{i-2}, w_{i-2})\right],$$

$$\forall i = 1, 2, \dots N - 1.$$

The local truncation error is $\tau_{i+1}(h) = \frac{-19}{720}\, y^{(5)}(\mu_i) h^4$ for any $\mu_i \in (t_{i-3}, t_{i+1})$.

As we can see, the truncation error for this method is smaller than for the **Adams-Bashforth** method, this is because generally **implicit** methods have smaller truncation errors, however they are commonly used to improve the approximations of **explicit** methods, as we are about to see.

3.5.3. Adams-Bashforth-Moulton

As we have mentioned before, **implicit** methods are used to improve the approximations of **explicit** methods, since in **implicit** methods the term w_{i+1} appears on both sides of the equation, we use the w_{i+1} calculated with an **explicit** method in the **implicit** method equation.

The combination of **implicit** and **explicit** methods is called a **predictor-corrector** method, normally executed in two steps, in the first one the **explicit** method approximates a solution, and in the second one the **implicit** method uses this approximation to generate a new, corrected approximation.

Since we are using the **Adams-Bashforth** method together with the **Adams-Moulton** method, this method is called the **Adams-Bashforth-Moulton** method.

In order to use this method, we need 4 initial values w_0, w_1, w_2, w_3 , then we proceed to calculate an approximation using the **Adams-Bashforth** method:

$$w_4^{(0)} = w_3 + \frac{h}{24}\left[55f(t_3, w_3) - 59f(t_2, w_2) + 37f(t_1, w_1) - 9f(t_0, w_0)\right],$$

After we have calculated this approximation, we proceed to correct it using the **Adams-Moulton** method:

$$w_4^{(1)} = w_3 + \frac{h}{24}\left[9f\left(t_4, w_4^{(0)}\right) + 19f(t_3, w_3) - 5f(t_2, w_2) + f(t_1, w_1)\right],$$

After achieving this, we have successfully improved our first approximation, the next iteration the **Adams-Bashforth** method will use the value $w_4^{(1)}$ as one of its initial conditions.

4. Development of the program

4.1. Selection of framework

The selection of a correct development framework is one of the keys to a project's success, creating software is a complex process, that requires multiple tasks such as coding, designing and testing to be performed correctly in order to achieve success. The simulator we have set out to build requires complex code that will need to be developed, the integration of 3-dimensional graphics and an interactive interface as well as ease of execution when the application is complete.

4.1.1. Matlab

The first option we considered was the use of the programming language MATLAB since this was the language the previous simulator was built on [6], the advantages it provides over other programming languages is the ease of programming large vector operations, which our simulator will utilize enormously as well as the existence of multiple numerical integrators already developed in Matlab libraries, the main drawback of choosing Matlab as our programming language was the inexistence of tools to develop a truly interactive 3 dimensional space and graphic user interface.

As it has been stated before, Matlab excels in matrix operations such as indexing, concatenation, transposing, reshaping and many more [7], this makes programming much easier for large vector operations, such as the ones undertaken in the programming for this project. This is not the case in other C-like languages, where native matrix operations do not exist and the use of libraries is almost necessary for development.

```
//Program to multiply a matrix by its transpose
--------------------MATLAB---------------------
A = [1 2 0; 2 5 -1; 4 10 -1]
B = A'
C = A * B

----------------------C-----------------------
#include<stdio.h>
int a[3][3]={1,2,0,2,5,-1,4,10,-1},b[3][3],mul[3][3],i,j,k;
int m = 3, n = 3;
for(i=1;i<=m;i++){
    for(j=1;j<=n;j++){
        b[j][i]=a[i][j];
    }
}
for(i=1;i<=m;i++){
    for(j=1;j<=m;j++){
        mul[i][j]=0;
    for(k=1;k<=n;k++)
        mul[i][j]=mul[i][j]+a[i][k]*b[k][j];
    }
}
```

Image 3. MATLAB and C programs to multiply a matrix by its transpose

The main downside of programming the simulator in Matlab is the difficulty to create a 3 dimensional graphical user interface that is interactive and allows for a proper and appealing visualization of the calculated body motions, while a rudimentary GUI would have been possible, and sufficient documentation can be found online [8]**,** Matlab is just not suited for complex interface development.

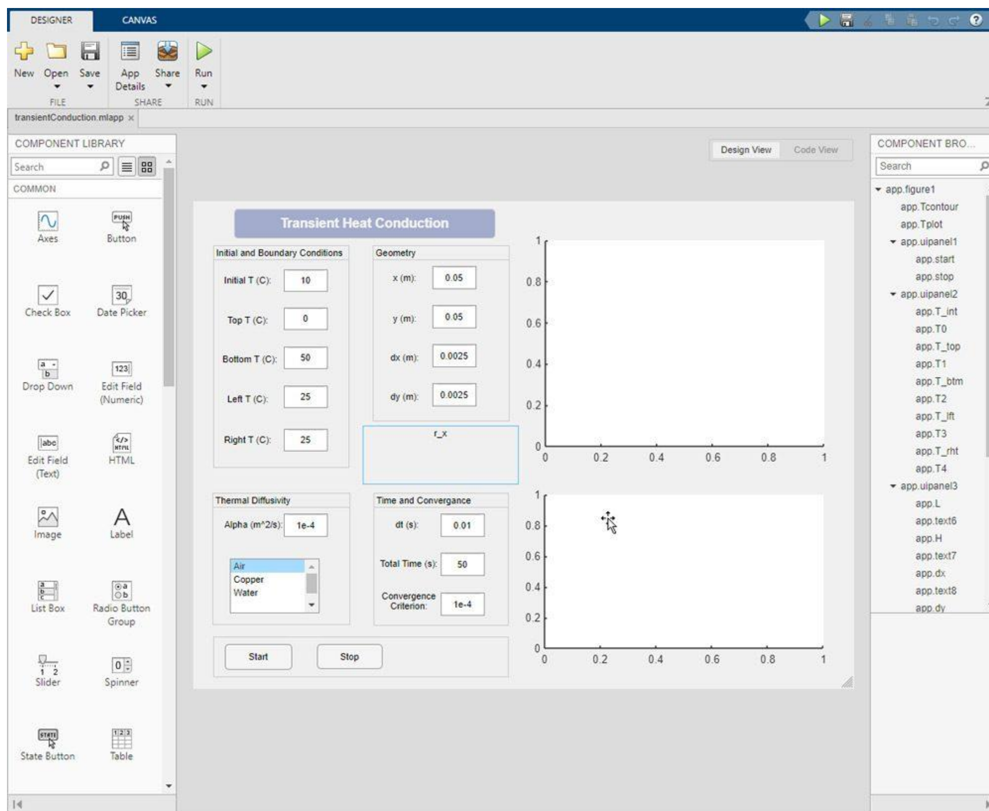


Image 4. Matlab App Designer Interface [9]

4.1.2. First approximation in Python

Python shares many of the advantages that Matlab has when developing the simulator, matrix operations are very easy to program and many numerical integrators are already developed for these languages, having said that, Python also suffers from the lack of quality 3 dimensional interactive GUIs.

Despite this, Python was chosen as the language to create a first approximation of the simulator, to see whether or not the programming in C#, which will be discussed later, could be undertaken.

In this first approximation, the main objectives were to do a small version of our simulator in order to ease the transition to C# and Unity for final development. The tasks

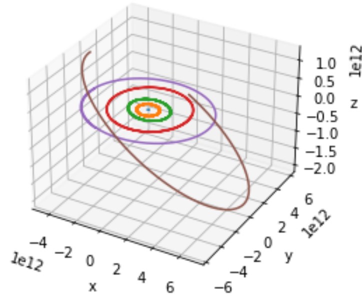
that were completed successfully were the programming of numerical methods, which was essential to the viability of the project since, as we will discuss later, this same programming would have to be accomplished in the C# language. Other tasks included the plotting of a reduced number of simulations to check the quality of the calculations and finally the realization of a small-scale energy graph.

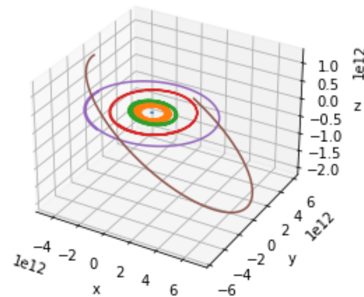
This small-scale simulator implemented some of the integrators described in the Matlab version [6], among them were Euler, Leapfrog, Runge-Kutta 4, Adams-Bashforth and Adams-Bashforth-Moulton. The programming was made easier by the use of the library Numpy [10] as well as several tutorials on numerical integrators [11].

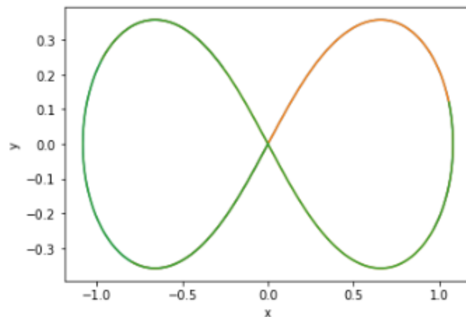
```
def runLeapFrog(T, dt, y0, masses, t0 = 0):
    nsteps = int((T-t0)/dt)
    history = np.empty((nsteps+1, len(y0)))
    energy = np.empty((nsteps+1, 2))

    history[0, :] = y0
    energy[0,:] = getEnergyAndAMomentum(y0, masses)

    t = t0
    for i in range(nsteps):
        history[i+1] = LeapFrog(t, dt, history[i,:])
        energy[i+1,:] = getEnergyAndAMomentum(history[i+1],masses)
        t += dt

    return history, energy
```

Image 5. Code to execute the simulation with LeapFrog integrator.

The development of the simulator was very useful to confirm that indeed it was possible to program the integrators manually and output the correct simulations despite them still being static. These codes were used as the first prototype in an iterative method of development and would serve as the stepping stone for the development in C#.







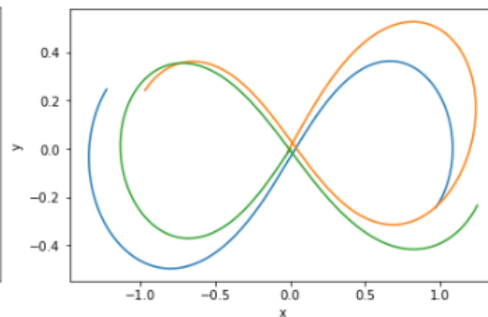


Image 6. 3d & 2d graphs showing Interior solar system and figure 8 simulations using different numerical integrators.

Despite this success, the next iterations of the simulator were not developed using Python due to the lack of tools to design and create an interactive 3-dimensional GUI, much like in Matlab, the tool selected to develop the final iterations was Unity, which will be discussed next.

4.1.3. Selection of Unity and C#

Unity is one of the most popular game engines in the market [12], used for creating cross-platform (Windows, iOS, Linux, Android…) 2D and 3D videogames and simulators. It was a very interesting option from the beginning for our project, due to the ease of developing and deploying an application which would be valid both for iOS and Windows platforms.

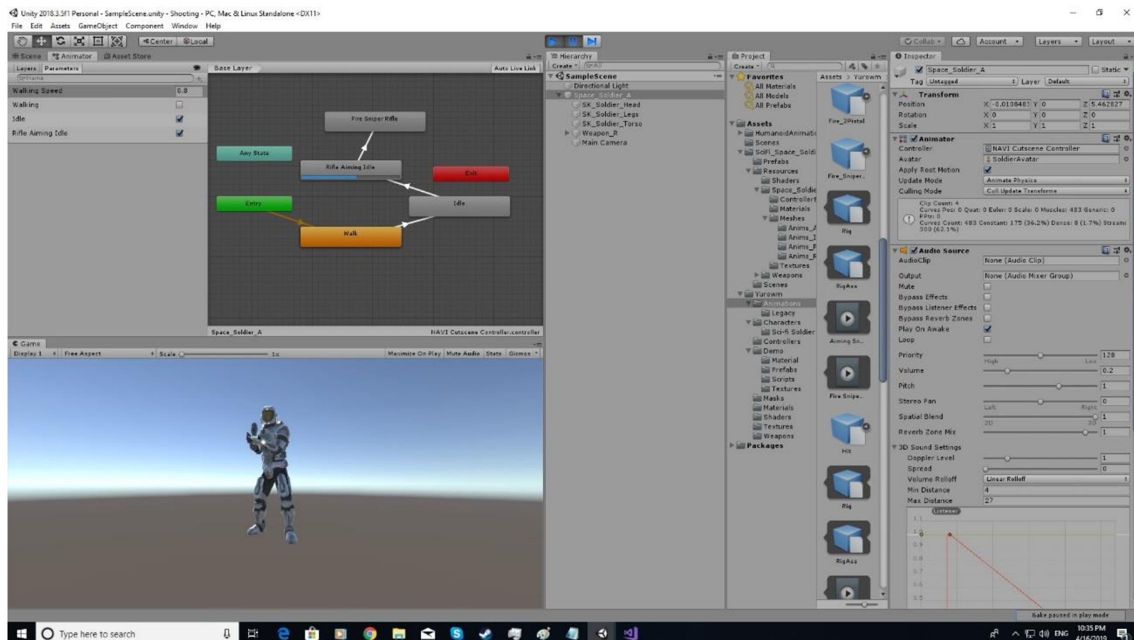


Image 7. Sample screenshot of a Unity project

The underlying programming language that is used in unity is C#, which as we explained earlier, is much more difficult for programming matrix operations and manipulation than Matlab or Python. To bridge this gap, some external libraries such as Numsharp [13] have been used to solve some of the intrinsic issues and ease the development of the solution.

In addition to this problem, the other main challenge for building the simulator in Unity is the lack of libraries from which to import numerical integrators, as C# is not a language normally used for these types of calculations and thus there is little supporting materials that can be found on the internet. Therefore, the majority of the numerical integrators

have to be manually programmed, requiring a lot of effort to correctly develop and optimize.

The main strength Unity has over Matlab and Python environments is the suitability to develop 3D software, this is due to the vast array of tools, technologies and ready to go assets available to developers, especially given how comparatively easier they are to utilize [14]. Unity offers 3D support, creation and optimization of custom graphics, GUI design as well as built in menus all completely integrated, rendered, and optimized by the game engine, leaving the developer to focus on the more technical scripts. To build upon this, the compilation of all the source code into a single executable file compatible with Windows and iOS makes the simulator very easy to access and utilize since the final user does not have to execute command lines or read through extensive documentation to open the program.

It is because of all the reasons mentioned above, as well as the familiarity I had with this environment, which would improve development speed and reduce errors that Unity was chosen as the framework to create this application.

4.2. User Interface (UI)

The User Interface [15], is the point at which human users interact with a computer website of application. The goal of an efficient UI is to make the user's experience easy, intuitive and enjoyable, requiring minimum effort on the user's part to receive the desired outcome all while avoiding undesired outcomes.

4.2.1. Design philosophy

The importance of a good UI cannot be overstated, to produce a satisfactory experience for the user, a UI needs to be clear, intuitive, not overbearing, and easy to use. The contrary is a UI that displays too much information, has a complex set of options, an

outdated design and is not easy to use [16]. All of this leads to confusing and negative user experience, our design philosophy has set out to avoid these mistakes.

As it was mentioned in the first chapter of this thesis, most of the existing simulator suffer from several flaws. Web-based simulators tend to be too simplistic, offering the user a very restricted simulation and rarely implementing the third dimension. In addition, the numerical methods used in the solution of the problems is rarely available, let alone modifiable by the user.

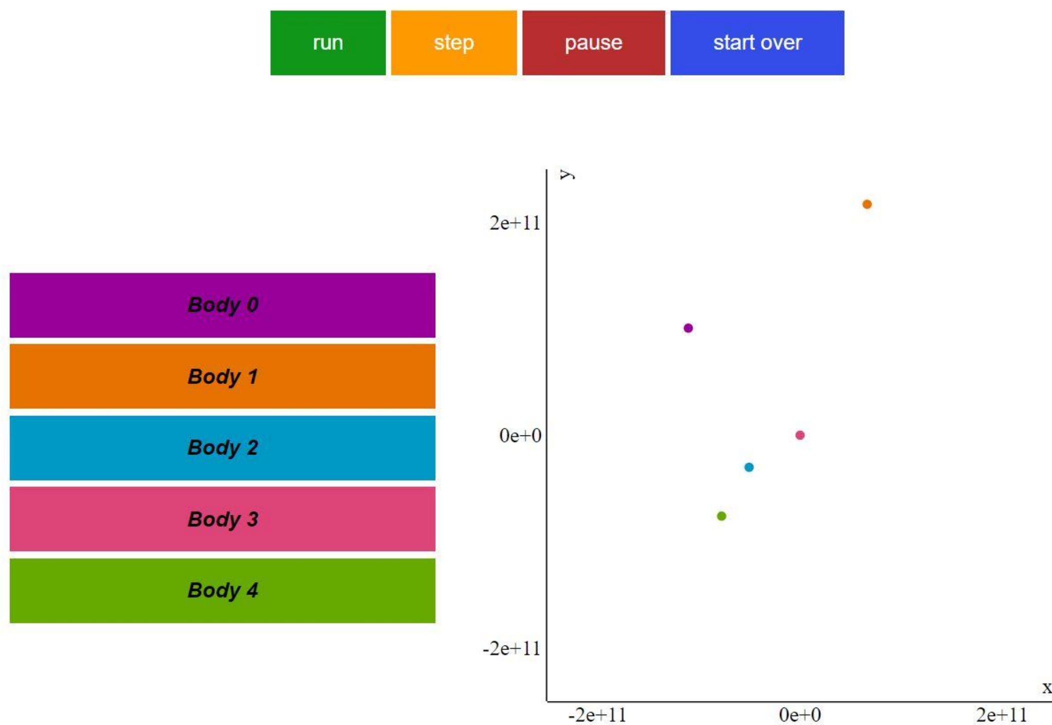


Image 8. Web-based simulator developed by Clark University (https://mathcs.clarku.edu/~jtrahan/subsites/nbody/)

Commercial simulators suffer from the opposite problems, overwhelming the user with too much unnecessary information, and occupying the UI with many complicated options and menus. The lack of interactivity is also prominent in these kinds of simulations, opting for a more static behaviour.

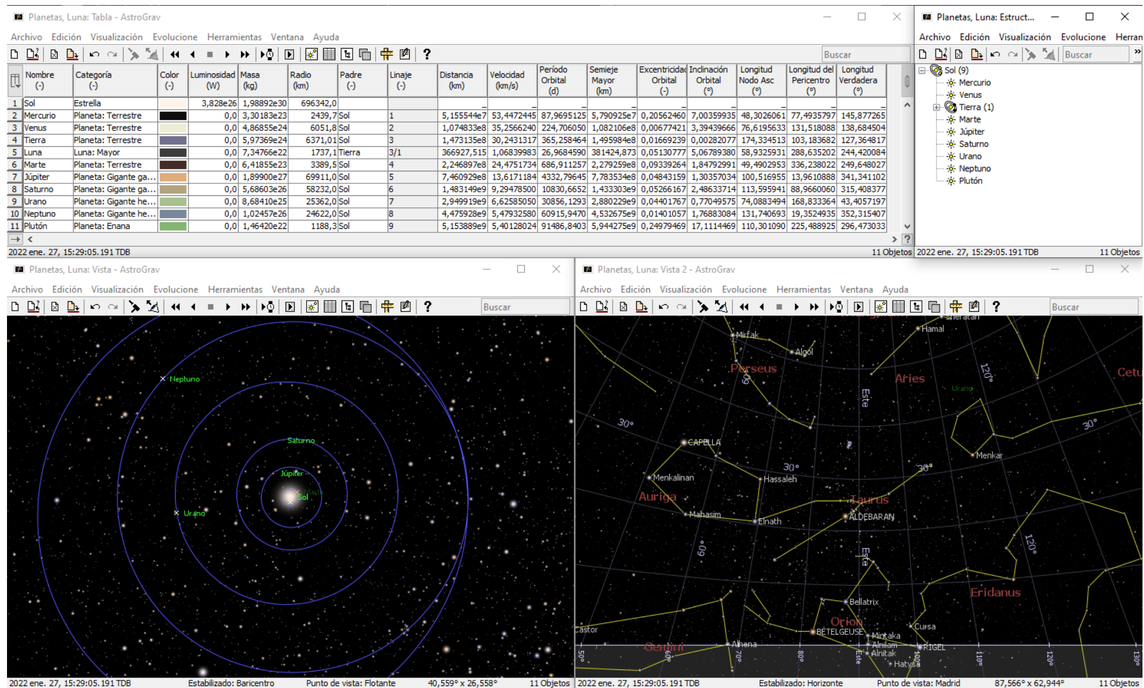


Image 9. Screenshot from first execution of AstroGrav simulator. [17]

In order to improve upon the existing simulators, the design for our UI has been based on the following principles:

- **Intuitiveness**: The simulator UI should be easy to use for the first time user as well as for an experienced user, in order to achieve this a set of instructions is provided on the first screen, and in every menu, the actions to perform are clearly displayed so that the user can select them without any confusion. Apart from this, all buttons, toggles, graphs and other interactable items should offer information into what their function is so that the user can get the expected functionality from them.
- **Usability**: The simulator UI should provide a satisfactory experience for the user, this is achieved by reducing user burden, for example being able to use the Tab key to move between input fields the user has to introduce, minimizing loading times and making mistakes easy to fix.

- **Simplicity**: The simulator should have a simple interface, displaying the essential information to the user, and allowing for options to minimize cognitive burden for the user when receiving too much information all at once.

4.2.2. Resulting UI

Each of the following menus that are about to be explained are created through an object in Unity called Scene [18], these Scenes contain the different objects that allow the user to interact with it, such as Buttons, Sliders and the scripts that manage the different menu logic.

Next, we will see the functionalities the menus that have been created offer the user, as well as how they interact with each other.

4.2.2.1. Main Menu

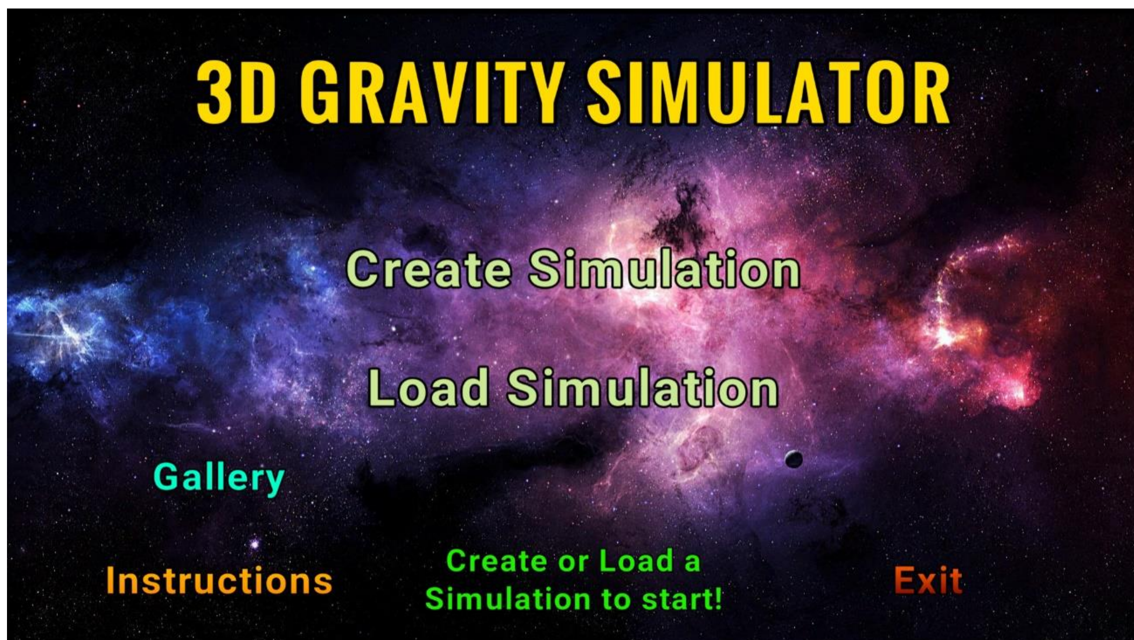


Image 10. Main menu loaded when starting the application.

The main menu that can be seen on image 10 is loaded when executing the simulator, this menu has different functionalities. For the first time users of the program, a flashing green text in the bottom indicates them to “Create or Load a simulation to start”, pointing them in a path to take. A set of instructions can be accessed through the “Instructions” button, where users can learn about how to create, execute or edit a simulation, how to navigate through the program or receive tips on how to increase performance and speed.

The options the user can choose are the following:

- **Instructions**: Obtain set of instructions for the program.
- **Gallery**: Open a file explorer and look through screenshots taken with the app.
- **Create Simulation:** Open the **Create simulation**, explained next.
- **Load Simulation:** Open the **Load simulation** menu, explained next.
- **Exit:** Quit the program and return to desktop.

4.2.2.2. Load Simulation

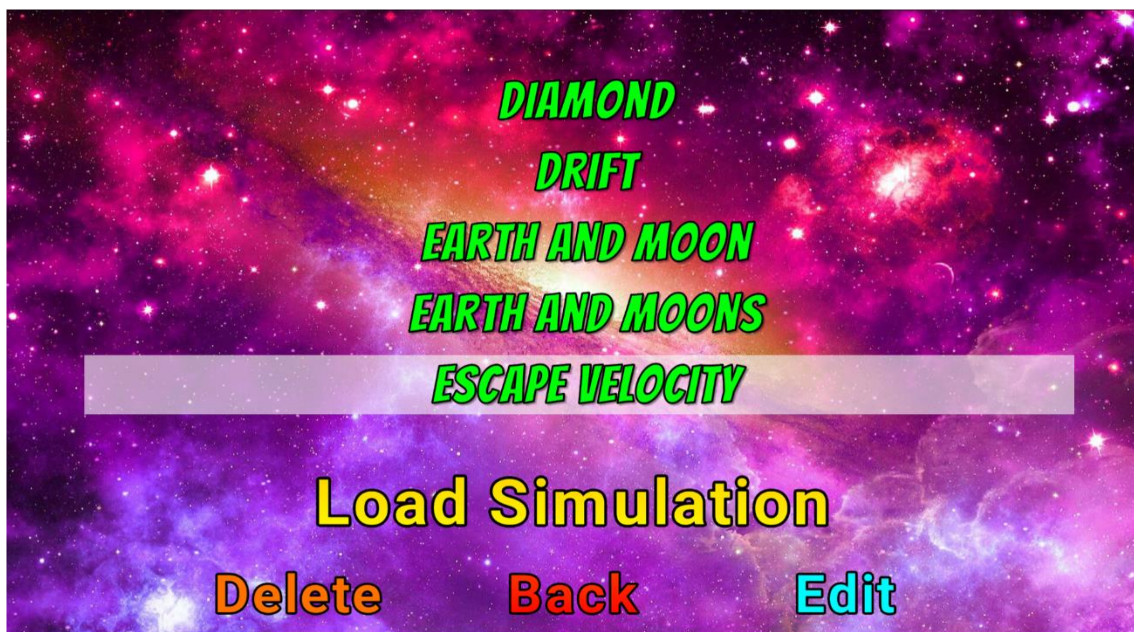


Image 11. Load simulation menu layout.

This menu, which can be seen in image 11, has several elements that are of interest to the user, the first of them is the big list of simulations we can see in **green**, after selecting the desired simulation, the user now has 4 options open to them:

- **Load simulation:** Loads the selected simulation in the **Simulation** scene, which will be explained later.
- **Delete:** Deletes the file of the selected simulation, removing it from the application folder that stores simulations.
- **Back:** Loads the **Main menu**.
- **Edit:** Opens the **New Simulation** menu on edit mode, explained next.

4.2.2.3. New Simulation

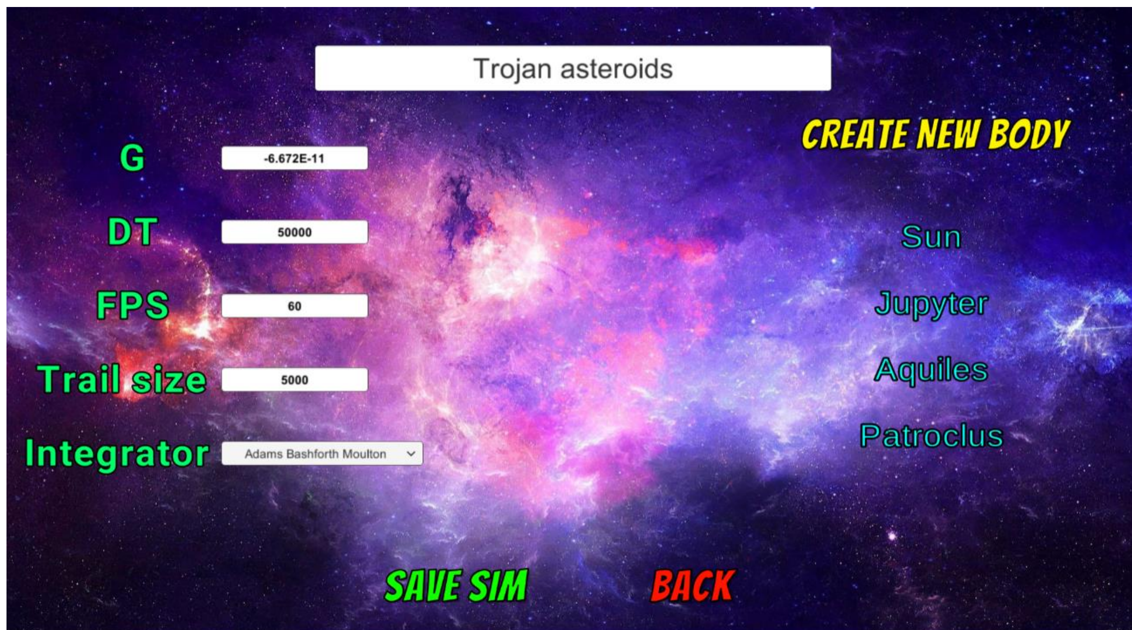


Image 12. New Simulation menu.

This menu allows for the creation of a new simulation, as well as the editing of existing ones, two paths lead to this menu, through the “Create Simulation” button in the **Main menu** and through the “Edit” button in the **Load simulation** menu.

The user can input different simulation parameters:

- **G**: Gravitational constant, default value is **6.672e^{-11} m^{3}kg^{-1}s^{-2}.**
- **DT**: Timestep, also referred to as **h**, measures the length of time elapsed between each iteration.
- **FPS**: Frames Per Second (**FPS**) which the simulator will target executing.
- **Trail Size**: Number of points to draw in a body's orbit.
- **Integrator**: Integrator to use in the simulation, this parameter will greatly impact the simulation and allows the user to compare between the different ones that have been implemented.

The user can also modify the simulation name and select **Create new body**, which opens a pop up menu where the user enters the **name**, **mass**, **position** and **velocity** of the body, placeholder values are provided, so the user can see the format that is expected and the recommended values for each input.

Once the parameters are set correctly, the user can press **Save sim**, which will save the new simulation, overwriting an existing the previous one if it was editing it and will start the **Simulation** menu, if the user presses the **Back** button, the program will load the **Main menu**.

4.2.2.4. Simulation

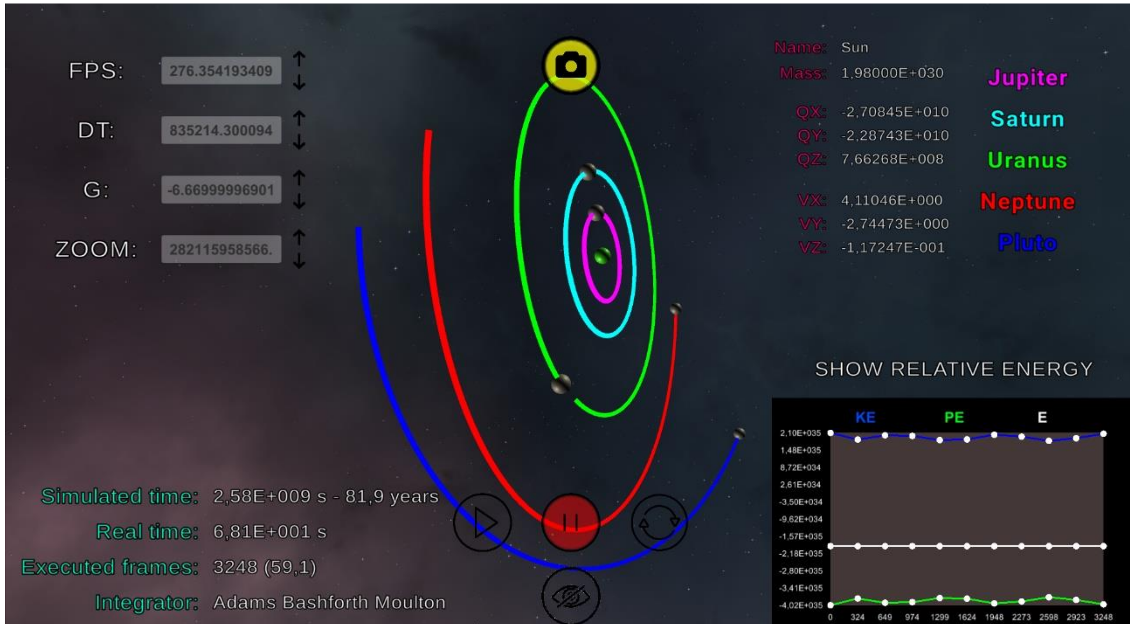


Image 13. View of the Simulation menu while executing the “Solar system external” simulation

This menu is the soul of the application, it displays the evolution of the simulation parameters that the user has input over time and allows for editing in real time as well as the visualization of a multitude of parameters. The UI is divided into several parts:

- Non-Interactable parameters:

Image 14. Non-interactable parameters in a simulation.

The purpose of these parameters is to give information to the user about the overall simulation, these are:

- **Simulated time**: measures the time that has been simulated in seconds.
- **Real time**: represents real time that has elapsed since the beginning of the simulation
- **Executed frames**: represents total number of frames executed since the beginning of the simulation and the real fps in parenthesis and lastly the **Integrator**: lets the user know which numerical integrator is being used.

- Interactive parameters:

Image 15. Interactable parameters in simulation.

The parameters that are shown in image 15 are the ones the user can interact with in real time, both by entering data through the keyboard and by selecting the arrows next to the value which will update it in real time. With these controls the user can set the desired parameters for frames per second (**FPS**), **dt**, change the gravitational constant (**G**) to see the effect on the simulation and **zoom** in/out.

- Body data:

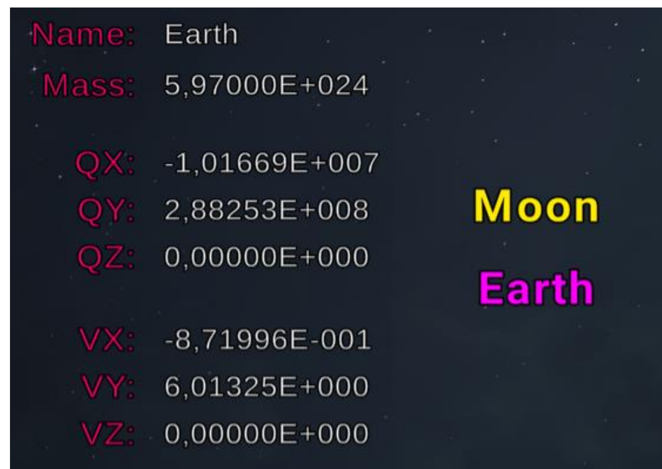


Image 16. Body parameters in the simulation.

Information about the evolution of each individual body data (**Position-Q** and **Velocity-V**) can be found on the top right section of the screen, where the user can scroll through the list of bodies in the simulation and select the one they are interested in to visualize how its parameters change in real time.

- Energy/Momentum graph:

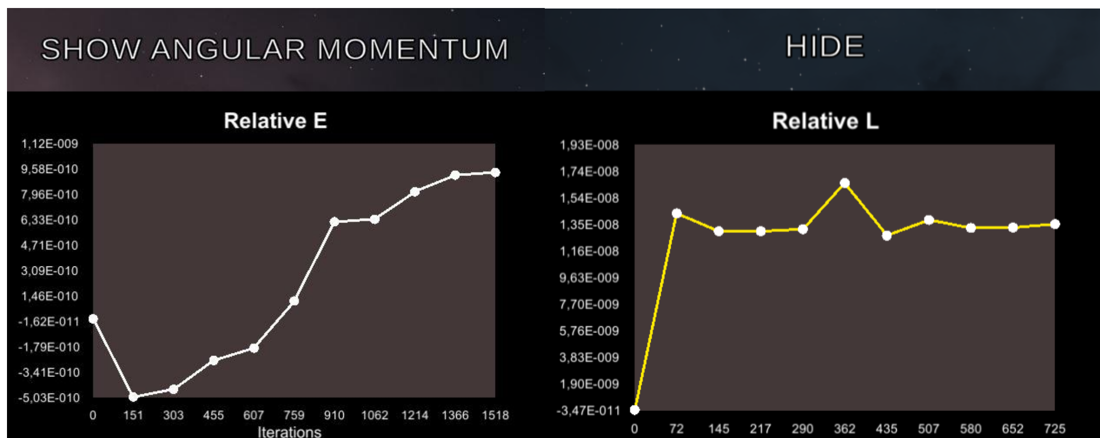


Image 17. Different modes of the graph showing **relative energy** and **relative angular momentum.**

This section of the UI consists of a graph that is updated in real time which can display 4 different types of data:

- Kinetic energy (KE), Potential energy (PE), **Total Energy** (**E**).
- **Relative energy.**
- Angular momentum (L).
- **Relative angular momentum.**

The **y axis** displays the values for the selected parameters and the **x axis** displays the iterations.

The user can cycle through the different graphs and can also choose to **hide** the graph from rendering, thus increasing overall performance.

- Control buttons:

Image 18. Control buttons located on the bottom part of the screen

The control buttons that can be seen on the image 18 allow the user to perform the following actions:

- **Play/Pause:** These two buttons allow the player to pause and resume the simulation, they can be toggled both by mouse and by pressing the **space** key.
- **Restart:** This button resets the simulation, removing any modifications made by the user to the simulation parameters.
- **Hide UI:** This button allows the user to hide the UI elements, this is very useful when visualization of the simulation is the main objective and the accompanying data is not required, UI can be reenabled by pressing the same button.

Image 19. Simulation before and after pressing the **Hide UI** button.

- **Screenshot:** This button allows for the user to take a screenshot of the current simulation, this is very interesting to keep track of how it evolves over time, or to save some data parameters without having to execute

the simulation multiple times. These screenshots can be accessed through the **Gallery** button in the Main menu.

Image 20. Screenshot button.

4.3. Programming of the simulator

This section will cover how this simulator has been programmed, the structure of the codes and how the encountered problems have been solved.

4.3.1. UI programming

The programming of the **UI** is radically different to the programming for the simulation. While the latter involves more mathematical codes, requiring the calculation of new positions, updating the graphics and setting the trails, the former is more modularized, managing the execution of small, more diverse set of tasks. We will now take a closer look at some of the scripts that are run by the **UI.**

4.3.1.1. Main menu

The scripts that manage the functionality of the **Main Menu** are simple and are all managed by a single script, **MainMenuManager.cs**. Some of the functionalities of this script are:

- **Loading** either the **New Simulation** scene or the **Load Simulation** scene.

```
//Functions to load scenes
1 referencia
void createSimulation() {
    SceneManager.LoadScene("CreateSimulation");
}

1 referencia
void loadSimulation() {
    SceneManager.LoadScene("LoadSimulation");
}
```

Image 21. Functions in **MainMenuManager.cs** which load the selected scenes.

- **Accessing** the file system to open an instance of the file manager, this section of the code needs to check the operative system to act accordingly, as files are not opened the same way in **iOS** and in **Windows.**
- **Displaying** the instructions, a set of pop-up menus can be iterated through by the user.

```
void nextInstructions() {
    hidePopUpMenus();
    popUpMenus[popUpMenuIndex++].SetActive(true);
    if (popUpMenuIndex >= popUpMenus.Length)
        popUpMenuIndex = 0;
}
```

Image 22. Function in **MainMenuManager.cs** to iterate through the instruction menus.

4.3.1.2. New Simulation

The complexity of the scripts increases as the tasks go from trivial ones, such as loading scenes and files, to more complex ones, such as editing a file of simulation parameters. This menu allows the user to **create** a new simulation as well as, through the Load Simulation menu, **edit** an existing simulation. In order to achieve this, several codes have to be implemented, the most important of them are:

- **NewSimManager:** The main tasks of this script are managing the events of buttons, loading the existing simulation when the user chooses to edit it and sending the simulation information to the **DataManager.cs** to save it into the file system and then load it in the **Simulation** scene.

```
1 referencia
public void editSimulation() {
    editingMode = true;
    importSimParameters();
    setBodies();
    setInputFields();
}

1 referencia
void saveSim() {
    if (!inputFieldsEmpty() && panelManager.hasBodies()) {
        dataManager.saveSimulationParameters(editingMode);
        editingMode = false;
        SceneManager.LoadScene("Simulation");
    }
    else
        blinkText.blink(4);
}
```

Image 23. Functions to edit and save a simulation in **NewSimManager.cs**

- **DataManager:** Once all the parameters have been correctly set, this script **parses** all the input parameters to their desired type, **calculates** the correct zoom for every simulation based on the maximum distance and creates a **json** object to then store the data as a .json file.

```
{
    "integrator":"Euler",
    "G":-6.66999996901119e-11,
    "dt":3600.0,
    "scaleFactor":51200000.0,
    "fps":30.0,"trailSize":250,
    "nBodies":2,
    "y0":[384000000.0,0.0,0.0,0.0,1000.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0],
    "masses":[7.33999999932568e22,5.96999991261919e24],
    "radii":[1.0,1.0],
    "names":["Moon","Earth"]
}
```

Image 24. **Earth and moon** simulation .json file containing the simulation parameters.

4.3.1.3. Load Simulation

This menu, and its corresponding scripts, **LoadSimManager** and **LoadSimPanelManager** fulfill several tasks:

- **Loading** all the simulations contained in the **\Examples** folder and sorting them in order to make searches easier for the user.

```
private void readFiles() {
    //Add all json files to a list and sort it
    files = new List<string>();
    path = Application.streamingAssetsPath + "/Examples/";
    foreach (string file in System.IO.Directory.GetFiles(path)) {
        if (file.EndsWith(".json"))
            files.Add(file);

    }
    files.Sort();
}
```

Image 25. **readFiles** function in **LoadSimPanelManager** which reads all .json files in the Examples folder.

- **Generate** the UI panel containing the list of all simulations, where each name is a button with a given file path to a simulation, when the button is pressed it sets a variable from which the loading, editing or deleting information is extracted.

```
private void generateCells() {
    //If cells != null -> it is being called after edit/deletion of file ->
    //we must delete the old cells and re create them
    if(cells != null) {
        foreach (GameObject cell in cells)
            Destroy(cell);
    }

    cells = new List<GameObject>();
    foreach (string file in files) {
        GameObject cell = Instantiate(cellPrefab);
        cell.transform.SetParent(this.gameObject.transform, false);
        cell.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text = Path.GetFileNameWithoutExtension(file);
        cell.GetComponent<Button>().onClick.AddListener(() => ButtonClicked(file));
        cells.Add(cell);
    }

    //Set the first element of the list at the top
    this.gameObject.GetComponent<RectTransform>().position = new Vector3(981.7f, -50000, 0);
}
```

Image 26. Function to generate list of simulations available to load/edit/delete in **LoadSimulation** menu.

4.3.2. Simulation programming

When studying the structure of the codes that run the simulation, it is critical to discuss the most essential parts: **loading** the simulation, executing the **iterations** and **updating** the UI. This is because the simulation the user experiences is just a sequence of different iterations, which are all run in the same way and are executed from a series of parameters that are loaded from files.

4.3.2.1. Main scripts

It is important to note that a wide of array of scripts manage the main simulation code, some manage the calculations using numerical methods, some control the canvas and how the user interacts with it and others oversee calculating and rendering the graph data, as well as many other auxiliary codes, here are the main ones:

- **SimManager:** This is the heart of the simulation, it is the script that controls all others and stores the most important data, its functions are **Loading** the simulation parameters, calculating the **Next Iteration** and **Updating** the graphics.
- **Canvas Manager:** This script manages all of the UI (excluding the graph), it is in charge of **Updating** certain simulation parameters which are editable by the user, such as **G**, **DT**, **Zoom**, and **FPS**. As well as **Receiving** information from the SimManager in order to display it in the UI.

4.3.2.1.1. SimManager

As its name states, the Sim(ulation)Manager manages the simulation, it acts like a "brain", which controls a "body" of codes, it is divided mostly into three sections, **Start**, **Update**, and **public methods**. This script also contains some of the most important variables in the whole simulation, some of them are:

- **y0:** This is one of several parameters from the simulation data, but it is the most important, it is the code representation of the initial conditions of the bodies, take the following configuration:

	Earth	Moon
Positions	X:0m Y:0m Z:0m	X:384400000m Y:0m Z:0m
Velocities	X:0m/s Y:0m/s Z:0m/s	X:0m/s Y:10000m/s Z:0m/s

Image 27. Initial configuration of the **Earth and moon** simulation.

- **History:** List of arrays of type double which store every **y** vector that has been calculated each frame, if we have a simulation with 6 bodies that has run for 100 frames, the size of **history** will be 100x36.

```
public List<double[]> history = new List<double[]>();
```

Image 28. Declaration of **history** in SimManager.cs.

- **Bodies:** Array of **GameObjects** [19], these are the graphical representation of our bodies, containing both the **Sphere** and the **Line Renderer** (orbit) for each of our bodies.

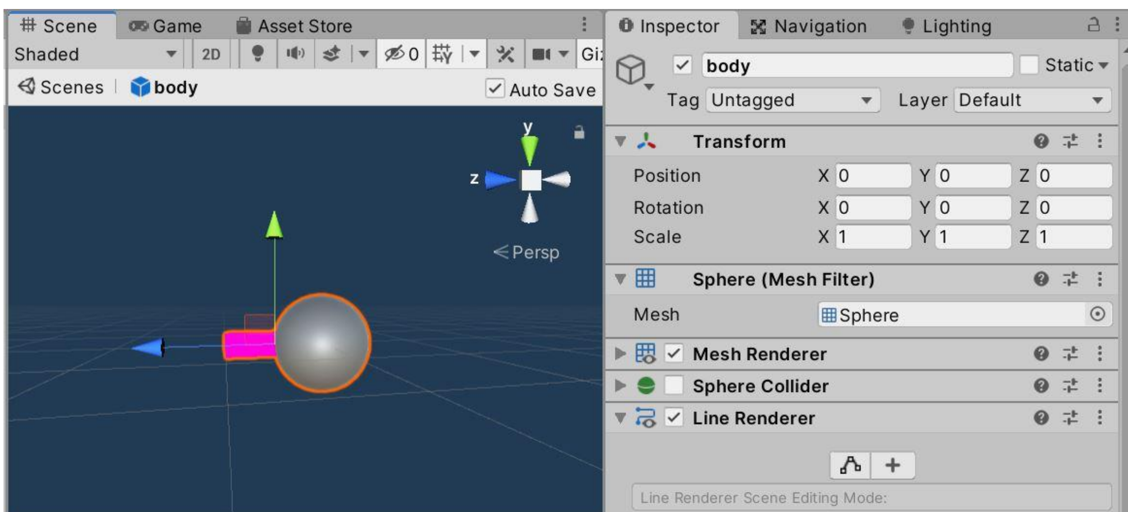


Image 29. View of a body **GameObject** with its components in the Unity Editor.

Now we will take a look at the main sections in which the code is divided:

- **Start:** This function defined by Unity [20] executes when the simulation is first loaded, in it the SimManager performs the following actions:
  - **Loading** the simulation data from the file selected by the user.
  - **Instantiating** (Creating) the 3D bodies with their parameters.
  - **Initializing the history** depending on the integrator used as some require more than 1 initial condition.

- **Initializing** the Canvas (UI).

```
void Start() //SimManager.cs
{
    importVariables();
    instantiateBodies(y0);
    setBodiesParameters();
    initHistory();
    initCanvas();
}
```

Image 30. Start() function in SimManager.cs

- **Update:** This function is also defined by Unity [21] and it executes each frame, the most important actions it performs are:
  - **Check** that the simulation is playing, in order to advance a frame, this is done by checking if the Boolean variable **playToggle.isOn** has the value **true.**
  - **Control** the current FPS, since the **Update** method is called on every Unity frame, there is a difference between the desired **FPS** value and the **FPS** value unity is trying to achieve, this is controlled by a set of variables that keep track of time and only allow the desired number of frames to be executed per second.
  - **Calculate** the next iteration, the selected numerical integrator is used to obtain the following approximation of the simulation parameters.
  - **Check** that no collisions have occurred, if they have, merge the two bodies by removing them and creating a new body which will be located in the halfway point between the two collided bodies.
  - **Update** the positions of the in-game spheres representing the bodies by scaling them to in-game zoom.
  - **Update** the materials used to color the spheres based on which body the user is monitoring.
  - **Update** the graph variables and render the new result, this is done through the variable **graphManager**.

```
void Update()
{
    if (playToggle.isOn) {
        realTime += Time.deltaTime;
        tfps += Time.deltaTime;
        if (tfps < 1 / fps) {
            return;
        }
        //Frame is executed
        nExecutedFrames++;
        simTime += dt;
        deltaTime = tfps;
        tfps = 0;

        nextIter(integrator);
        checkCollisions();
        updatePositions();
        updateTrails();
        updateMaterials();
        graphManager.updateGraph();
    }
}
```

Image 31. Update() function in SimManager.cs

4.3.2.1.2. CanvasManager

The **Canvas** [22] is the Unity object in which we design and program the **UI**, as in any **UI**, in order to perform the actions the user desires, there needs to be a script which manages the **UI**, that is the function of our CanvasManager. Some of the main variables are:

- **Buttons:** The **Button** [23] object allows for actions to be performed when pressed, many of them are used in our Canvas, and they have different functions; allowing the user to control the simulation, increasing or decreasing simulation parameters, switching the graph view or allowing the user to inspect the parameters of a body.

```
public Button GUp;
public Button GDown;
public Button DTUp;
public Button DTDown;
public Button FPSUp;
public Button FPSDown;
public Button ZoomUp;
public Button ZoomDown;
```

Image 32. Button variables in CanvasManager.cs

- **Texts** and **InputFields:** To display information on screen, the object **TextMeshProGUI** [24] allows for the manipulation of the content of the text in real time. Most of these objects are only modified by the CanvasManager, for user manipulation, the object **InputField** [25] is used.

```
//User can interact with these objects
public InputField GIF;
public InputField dtIF;
public InputField fpsIF;
public InputField zoomIF;

//User cannot interact with these objects
public TextMeshProUGUI simTimeText;
public TextMeshProUGUI realTimeText;
public TextMeshProUGUI nFramesText;
public TextMeshProUGUI integratorText;
```

Image 33. InputField and TextMeshProGUI variable declaration in CanvasManager.cs

The most important part of the script however, are the functions it performs to update and modify the data that is being displayed in the **UI**, as well as modify the simulation parameters if the user chooses to do so, here are the most important functions:

- **Update:** Much like in SimManager.cs, this function also executes each frame, to improve efficiency, it is only executed when the Canvas is **not** hidden. The main actions it performs are:
    - **Update** the texts that display the selected Body parameters.
    - **Update** the simulation variables values, taking them from the **InputFields** and sending them to the SimManager.cs script while also updating the displayed text.

- **Update** the remaining texts the user cannot interact with.

```
void Update()
{
    if (!hiddenCanvas) {
        updateBodyTexts();
        updateInputFieldValues();
        updateInputFieldTexts();
        updateStaticTexts();
    }
}
```

Image 34. Update function for the CanvasManager.cs script.

4.3.2.2. Other scripts

While the **SimManager** and **CanvasManager** are the main scripts tasked with running and displaying the simulation, they rely on a wide array of auxiliary codes to perform parallel calculations, variable management and other auxiliary tasks. These are some of the scripts:

- **ELGraphManager:** This script is tasked with calculating the Energy, both KE and PE as well as the angular momentum, and rendering this data in a custom graph that is created with the **WindowGraph** script since Unity does not offer native graph support.

```
public void updateGraph() {
    updatePositions();
    calculateEnergy();
    calculateMomentum();
    testPrints();
    if (graph.mode != Window_Graph.HIDE && KE.Count > (graph.Ndots + 1))
        graph.ShowGraph(KE, PE, E, L, Erelative, Lrelative);
    else
        graph.disableGraph();
}
```

Image 35. **updateGraph()** function in ELGraphManager.cs script, this function is executed every frame.

- **Screensaver:** This script is in charge of capturing the screen when the user selects the screenshot button, as well as navigating in the file system and creating a new file in which to store the image.

```
1 referencia
void saveScreenshot() {
    string filepath = buildFilePath();
    ScreenCapture.CaptureScreenshot(filepath);
}

1 referencia
string buildFilePath() {
    string simName = SimManager.selectedFilePath.Split('/')[SimManager.selectedFilePath.Split('/').Length - 1];
    simName = simName.Replace(".json", "");
    string simFolder = destinationFolder + simName + "/";

    //check if directory doesn't exit
    if (!Directory.Exists(simFolder)) {
        //if it doesn't, create it
        Directory.CreateDirectory(simFolder);
    }
    return simFolder + System.DateTime.UtcNow.ToString("dd MMMM HH_mm_ss") + ".png";
}
```

Image 36. Main functions of the **Screensaver.cs** script, these functions take a screenshot and save it to the file system.

5. Performance and optimization

This section is dedicated to the study of the computational complexity of the N-body problem and the performance obtained by the simulator that has been developed. Several measures of performance will be taken into account, the most widely used is **Frames Per Second** (**FPS**), because it is the one that impacts user experience the most [26], 30 FPS is an industry standard considered optimal, with 60 FPS considered a more premium target. Other metrics that will be considered are **CPU**, **GPU**, and **RAM** usage. The main computational costs of the simulator will be analyzed, and the final performance of the simulator will be discussed.

5.1. Computational costs

The computational complexity of the N-body problem has been studied for decades, especially for the so called "many-bodies" where the large magnitude of **N** (usually > 10.000) implies that direct methods such as the ones used by our simulator are impossible, and special optimization techniques need to be implemented [27].

Next, we will review some aspects that affect the computational cost of calculating the evolution of any simulation, such as the size of **N**, the **visual settings**, the **numerical integrator** being used, and other calculations.

5.1.1. Effect of N

Because every body exerts a gravitational force on every other body in the simulation, the complexity of obtaining the total force that a body experiments in a given moment in time is of order $O(n^2)$ [28]**.** This is the biggest limiting factor in our simulator, since the number of calculations needed is exponentially bigger when **N** increases, the number of **Frames Per Second** that we can achieve decreases exponentially.

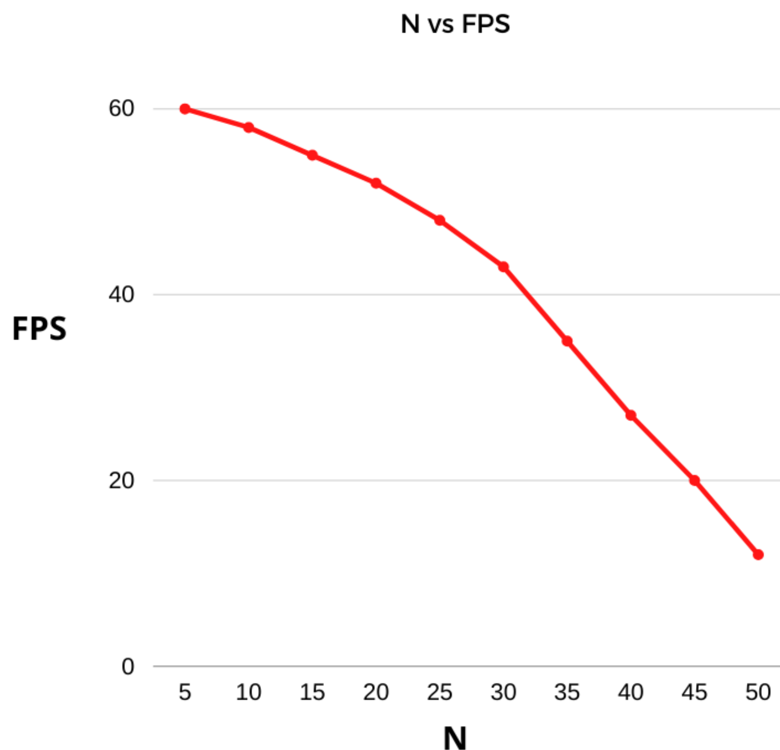


Image 37. Graph showing the drop in **FPS** value as **N** increases. Using Euler integrator and trail size = 250.

5.1.2. Visual settings

Graphical components are also a big part of computational costs, the elements in the scene in Unity, mainly the UI and the bodies need to be rendered [29], and the computational costs of rendering these elements needs to be carefully monitored**.**

The element that most affects graphic performance is the **Line Renderer** [30] , this component takes an array of two or more points in 3D space, and draws a straight line between each one. This is used to visualize the orbits of the planets, the length of which is referred to as **Trail size**.

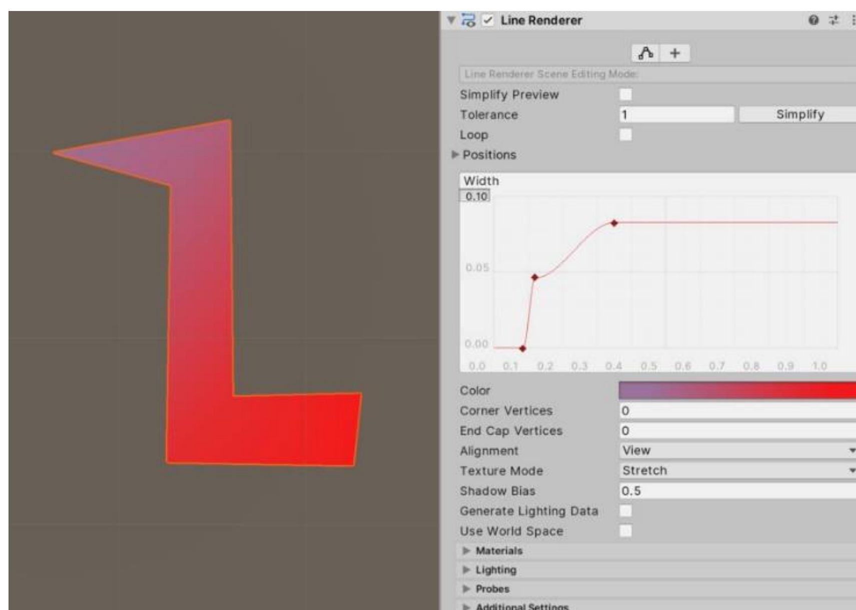


Image 38. Line renderer component in the Unity editor.

The main challenge comes when the selected Trail size is > 10.000, as the engine needs to calculate and render every point in the line for every body each frame, which causes the simulator to slow down considerably.

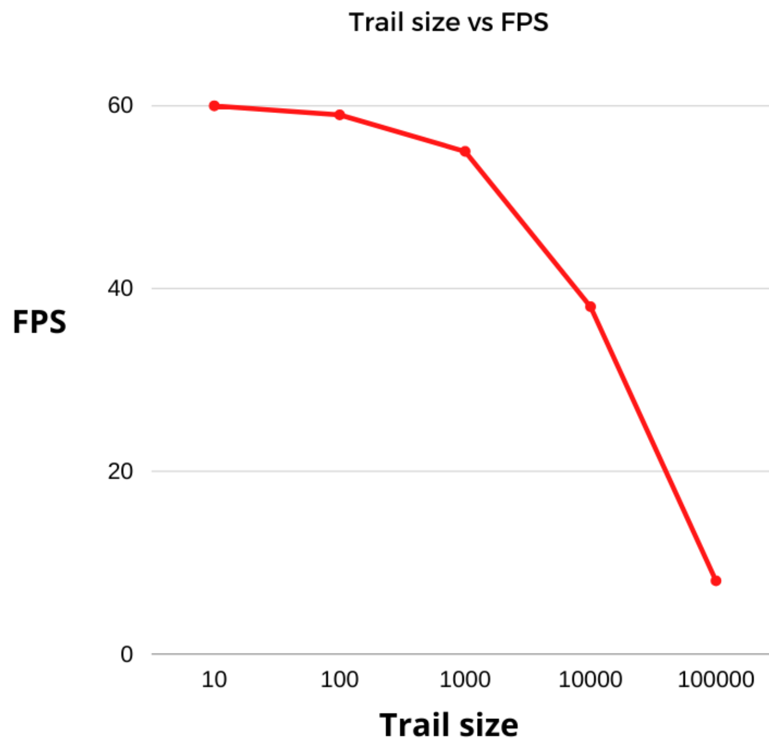


Image 39. Graph showing the impact of trail size on performance.

5.1.3. Numerical integrators

Numerical methods provide different ways of approximating the solution to any given N-body configuration, each approximation varies in its equations, error and complexity. As we have seen, integrating a simulation with the Euler method requires only one function evaluation, whereas in the Adams-Bashforth-Moulton method we need to combine two other methods each with several function evaluations.

Since one of the main objectives of the simulator is to contrast and compare different numerical integrators, observing the differences in the quality of the solutions and the error obtained, it is very interesting to also contrast the effect different integrators have on performance. In order to do this, the same simulation has been observed while varying the integrators and CPU usage has been measured.

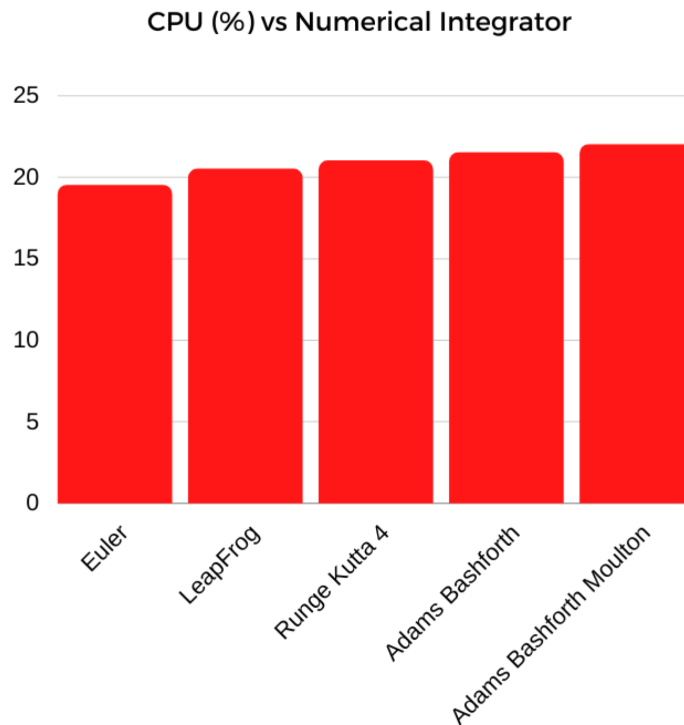


Image 40. CPU use of different numerical integrators in the simulation “Solar System Asteroids”, at the 1000th frame.

Surprisingly, the CPU usage is similar for all integrators, this could be due to the relative simplicity of the methods that have been implemented, as more complex integrators could have a more severe impact on CPU usage. From these measurements we observe that numerical integrators **do not** have a significant effect on the program’s performance.

5.1.3. Energy/Angular momentum graph

Other computational costs come from the calculation and rendering of the **Energy/Angular momentum graph**, this is because for each frame the program needs to calculate the new values for the **KE**, **PE**, **E** and **L**, update their Maximum and Minimum values, and render the graph. These calculations are not computationally significant, in contrast, when first developing this graph, inspired by [31], the rendering of the graph affected the performance massively, due to errors when batching the UI elements [32].

Once these issues were fixed through rendering optimizations [32] the effect of the graph calculations and rendering on the **FPS** was reduced significantly.

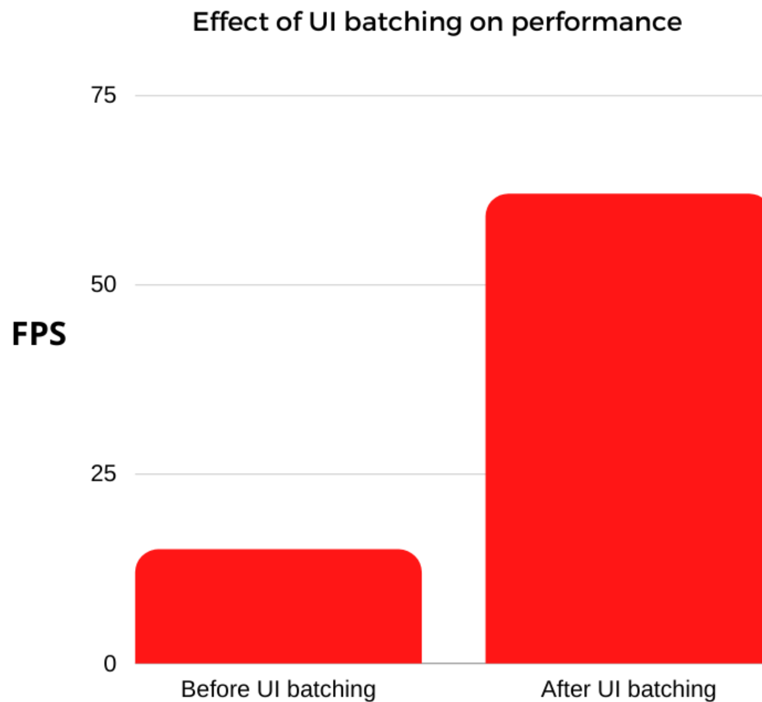


Image 41. Effect of UI batching on graph performance in simulation "Asteroids".

5.1.4. Improving performance

After visualizing how these parameters affect performance, we can draw some conclusions on how to improve the performance of the program. If the user suffers from low **FPS**, performing some of the following actions will help:

- Reducing **Trail size**: As we have seen, this is the parameter that least affects the simulation because it is only visual, and it is the one that impacts **FPS** the most, therefore it should always be kept at a minimum.
- Reducing **N**: This is not desirable, as it involves changing the simulation drastically by eliminating bodies until the **FPS** count reaches the desired target.

- Changing **Numerical integrators:** This is also not desirable, as generally changing the current numerical integrator for one with less computational costs involves losing quality in the simulation results, but it can be done if necessary.
- **Hiding E/L graph:** As we have discussed in the previous section, after some optimizations, hiding the graph does not significantly affect performance, but it might help speed up the simulator by a few **FPS.**

If the user wishes to improve the performance of the simulator, these are the internal options that are available to them, other external ways of increasing performance could involve shutting down other running applications or upgrading the hardware of the machine.

5.2. Resulting performance

We have already seen how different parameters such as **N**, the **Trail size** or the **Numerical integrator** used have a profound impact on the simulator's performance. However, other metrics are also used in the videogame industry [33] to evaluate the performance of an application, mainly **CPU**, **GPU** and **RAM** use, these metrics will also be useful to analyze how our simulator performs.

5.2.1. Effect of hardware

The measurements that about to be analyzed have been taken on a computer with the following specifications:

- **CPU:** Intel Core i7-6700 CPU 3.4GHz
- **GPU:** NVIDIA GeForce GTX 1060 6GB
- **RAM:** 16 GB RAM

This corresponds to a mid-upper range hardware configuration, performance can increase or decrease based on the user’s hardware specifications, we will also study which of these three parameters is the most important when trying to optimize performance.

5.2.2. CPU, GPU and RAM

To study the evolution of these parameters, the simulation “Orbit Layers” with **N** = 11, **Trail size** = 25000 and **Integrator** = Runge Kutta 4 was executed, and the following data was extracted.

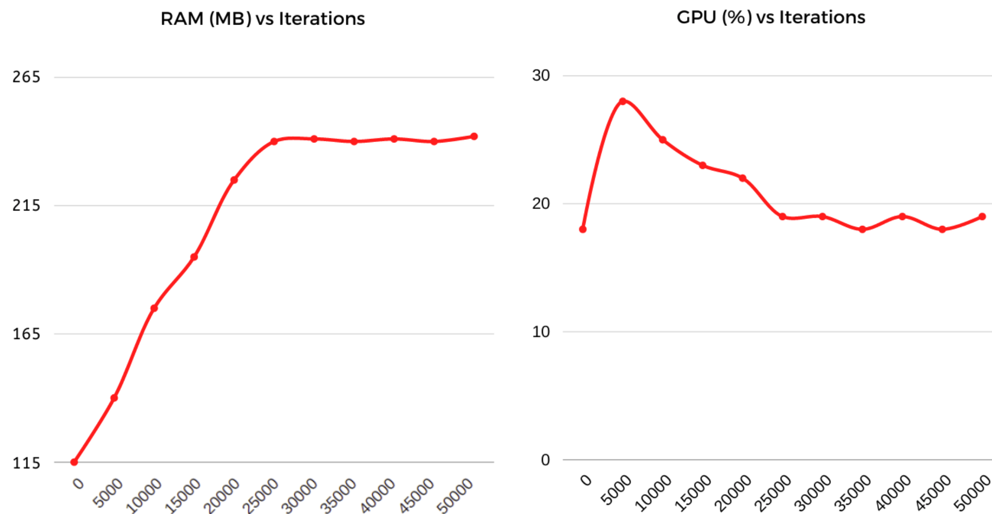


Image 42. Graph showing RAM vs Iterations.

Image 43. Graph showing GPU vs Iterations.

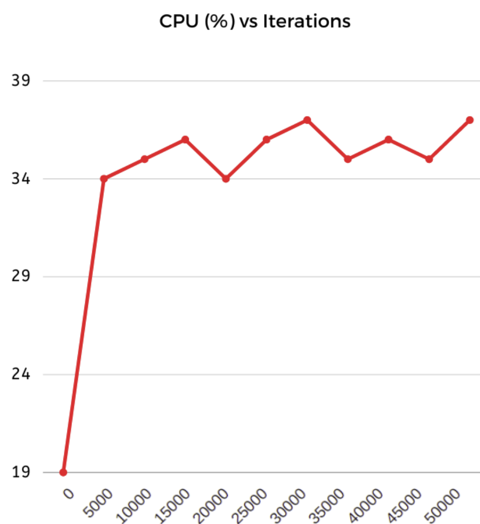


Image 44. Graph showing CPU vs Iterations.

The results from these measurements are very interesting but require a few explanations. As we can see, once again the **Trail size** is the parameter that has the biggest impact on performance.

It is apparent that until the **25000** Trail size limit is reached, the simulator keeps increasing the RAM allocated to it. However, the CPU usage does not increase, as the iteration of the orbit array does not appear to be parallelized by the processor, this means that one processor core needs to do more tasks each iteration, therefore decreasing the number of FPS achieved, as seen in 4.1.2.

Interestingly, GPU usage is inversely proportional to **Trail size**, since less frames are calculated by the processor, less frames need to be rendered graphically by the GPU, therefore lowering its consumption.

One solution to this problem would be to speed up the iteration of the orbit array by parallelizing this task by using technique such as parallelization. This consists of allocating multiple CPU cores to perform separate tasks, effectively “breaking” up the main task into many smaller ones that different CPU cores can perform separately and allowing for the calculations required to perform an iteration to be sped up [34].

6. User guide

This section will provide the guidelines on how to install, execute and utilize the program, explaining both the technical requirements for installation as well as instructions on its use and best practices to obtain the best experience when utilizing the simulator.

6.1. Requirements and installation

The technical requirements required to run the simulator are inherited from the Unity framework [35] **,** they are the following:

- **Operating system:** Windows 7+ or macOS 10.12+
- **CPU:** SSE2 Instruction set support
- **GPU:** Graphics card with DX10 (shader model 4.0) capabilities

These requirements are met by most modern machines, therefore most people who are interested in this simulator should be able to run it.

To install the program, the user simply needs to open **https://github.com/panosjuanis/3D-N-body-simulator** and download the version for the operating system they are interested in, after this is complete, simply copying the folder to the desired destination and executing either the .exe (Windows) or the .app (Mac) files will start the program.

In order to uninstall the program, simply removing the folders from the file system will delete the program and all its auxiliar files.

6.2. Instructions and tips

The instructions on how to use the program are explained in the main menu so that the user can understand concepts both regarding the calculation of the simulation, such as **DT** and **numerical integrators**, and of the graphical aspect of the program, such as movement with the camera and how to properly manage it.

Other aspects are also covered, such as managing the simulation files, both editing, creating, and deleting existing simulations and taking screenshots and visualizing them. The UI elements that have been described previously are also explained here so that the user can have a correct understanding of them and knows how to utilize them properly.

Lastly the user is given some tips on how to reduce the main problem challenging a premium user experience: performance. This has been discussed in an earlier chapter and the user is given a brief summary of these issues and how to address them, by reducing trail size, n, selecting the correct integrator and turning off the UI.

7. Examples

7.1. Simulation examples

One of the main goals of developing this simulator is the creation of a didactical tool to learn about the law of gravitation, numerical integration and visually understand the different celestial configurations that often are studied in theory.

To achieve this, and for the user to be able to experiment with different scenarios, dozens of pre-loaded simulations can be accessed by the user when entering the application, these are mainly divided into two parts, real world examples and theoretical solutions of the n-body problem.

7.1.1. Real world examples

There are hundreds of known configurations for the N-body problem, and it would be impossible to try to encompass all of them, but we have strived to introduce a variety of real world configurations focusing on the solar system, which is the system most people are familiar with, we will take closer look at two of them.

7.1.1.1. External solar system

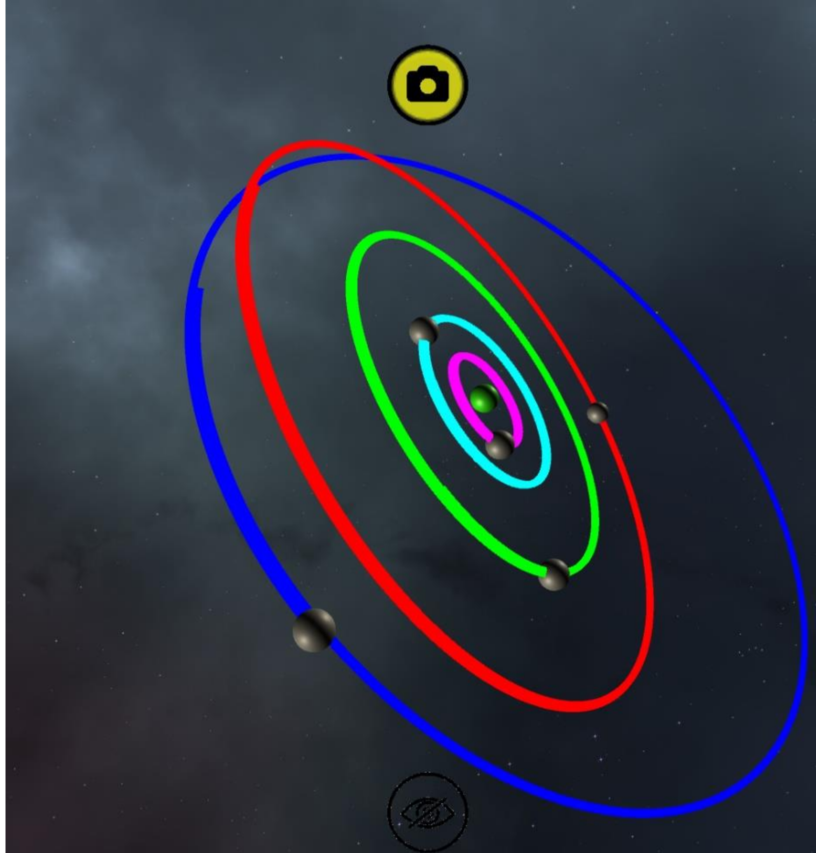


Image 45. “External solar system” simulation over a period of 260 years using **Adams-Bashforth-Moulton** integrator.

The initial conditions for this configuration were obtained through NASA’s database [36] and Mamen Peña’s simulator examples [6] and have been saved in the **“Solar system external”** file which can be accessed and edited by the user. The simulation that can be observed in the figure has evolved for a 260 year period, and the bodies that are represented are: **The Sun** (**Green sphere**), **Jupiter**, **Saturn**, **Uranus**, **Neptune**, and **Pluto**.

The user can move around the 3-Dimensional space in the simulator and observe the evolution of the system from any angle they desire, in this simulation the orbit described by **Pluto** is especially interesting, as it has a tilt that the other bodies do not have.

7.1.1.2. Trojan asteroids

Image 46. "Trojan asteroids" simulation over a period of 8 years using **Runge Kutta 4** integrator.

This is another real world configuration, this time relating to **Trojan asteroids** [37], which are a group of asteroids that share the planet Jupiter's orbit around the Sun first discovered around 1904 by E.E.Bernard. In this configuration, we will focus on the asteroids **Aquilles** and **Patroclus** in their obits around the **Sun**, the user can observe both asteroids share **Jupiter's** orbit around the Sun.

7.1.2. Theoretical solutions

Another type of configurations that can be found in the files of the simulator are the theoretical simulations, this is a very well established field of the N-body simulations, and hundreds of people have come up with configurations that achieve stable orbits,

due to the way the bodies move in the simulations, they are called choreographies as the bodies appear to be "dancing".

Next, we will take a closer look at two of these choreographies but this is a fascinating topic and new choreographies are discovered constantly, for more information see [38].

7.1.2.1. Figure 8

One of the most famous of these kinds of solutions is the so called **Figure 8**, because the motions of the bodies look like the number 8. It was discovered by Chris Moore [39], since this is one of the most famous theoretical configurations, we thought it would be interesting to visualize it with our program.

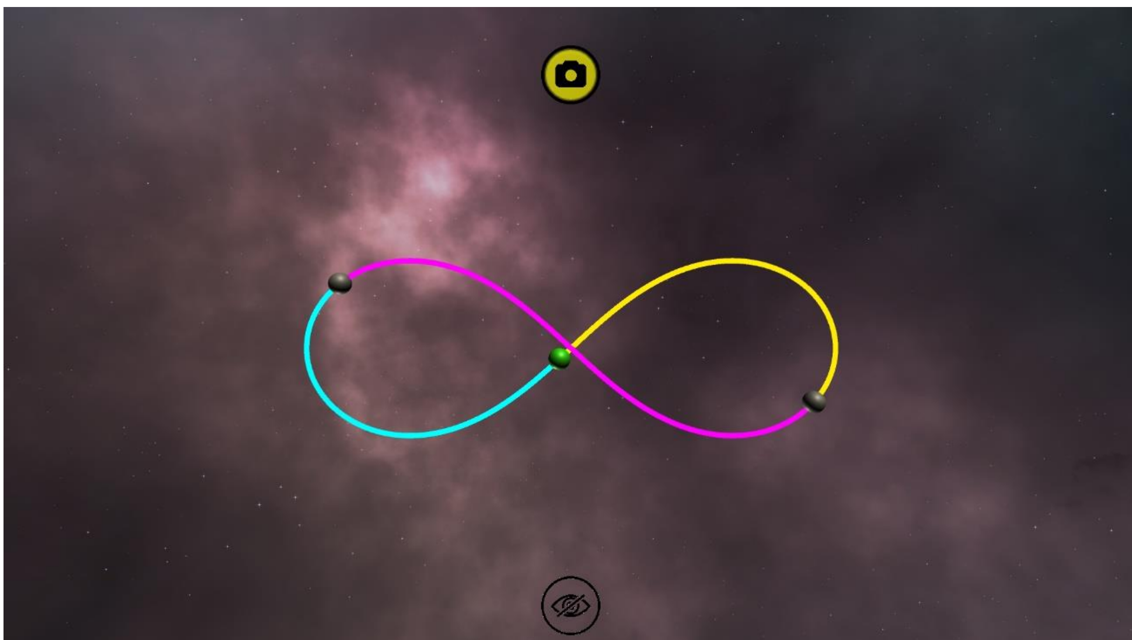


Image 47. Figure 8 simulation with the **Adams-Bashforth** method over a period of 9 seconds.

As we can see, the bodies perfectly follow the motions that are expected from them and display a beautiful **8**. Other variations of this configuration also exist, Figure 8 simulations can be created with different **N** values which result in more complex patterns, but always in line with the original solution.

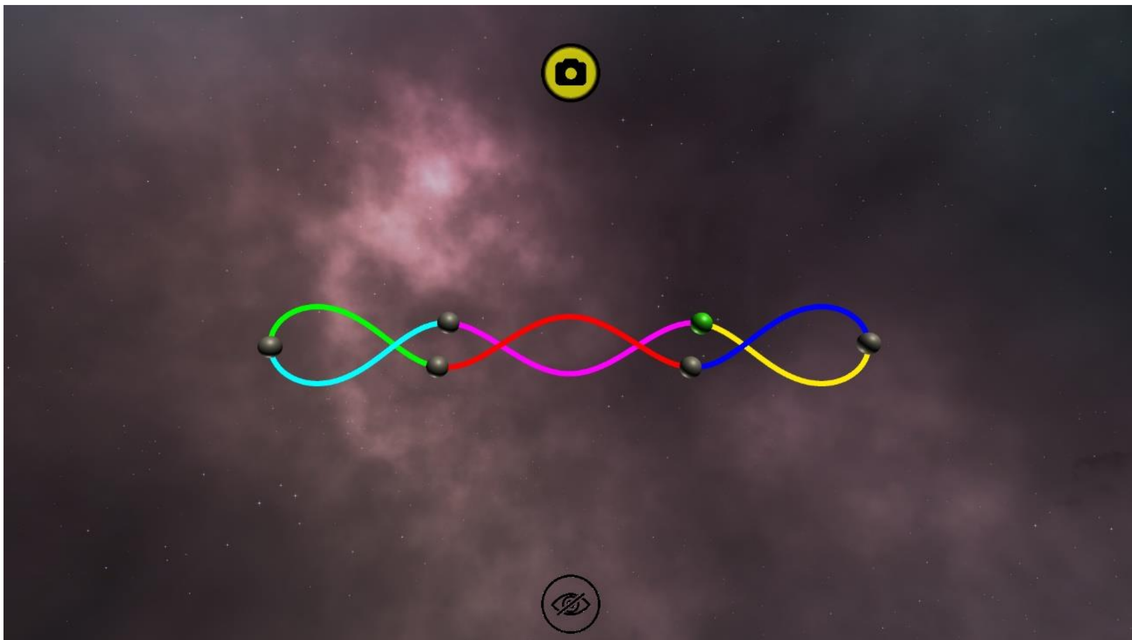


Image 48. Figure 8 simulation with **Adams-Bashforth** method over a period of 16 seconds with N = 6.

7.1.2.2. Broucke's periodic solutions

Another family of solutions was discovered by Roger A. Broucke and Michel Hénon in the 1970 [40]. In this family the three objects all have the same mass and in some of these solutions, two of the bodies follow the same path.

Several solutions from this family have been included into the simulator so that the user can visualize them and experiment with them.

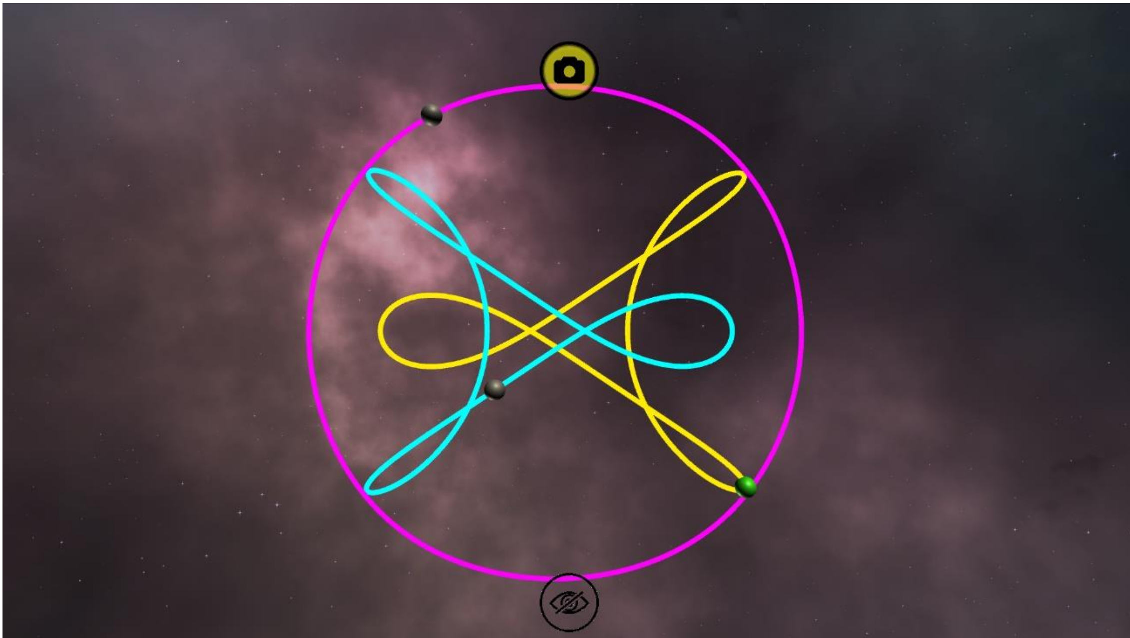


Image 49. Periodic Broucke solution 4, simulated with the **LeapFrog** method over 35 seconds.

8. Conclusions

The initial aim when starting this project was the development of a simulator for the N-body problem that was easy to use, representing the motions of the bodies in a 3-dimensional space and allowed the interaction of the user with different parameters and numerical methods.

These objectives were successfully completed, as the developed simulator can be installed and utilized by users with great ease, allowing them to experiment with different configurations of the N-body, visualizing or modifying different parameters in real time. The user can also move around the 3-dimensional space inside the simulation and monitor elements such as the energy of the system, its angular momentum, or the different body parameters.

Though other N-body simulators exist, the one that has been developed in this project excels in the visualization of the solution and the ease of use that it provides. Its UI is

tailored towards first comers and people that are familiar with the N-body problem alike, allowing the user to intuitively use the program, which is not the standard practice in the N-body simulators.

This makes it especially interesting for pedagogical uses, such as university lectures or any teaching of Newton's law of gravity. Researchers of the N-body problem, be it amateur or professional, can also find the simulator interesting, as the installation, camera movement and parameter options it provides are not easily found in other simulators, which tend to be simpler and not allow for 3-dimensional motion.

The development of this thesis has facilitated learning from a diverse range of topics, from the N-body problem formulation, the numerical methods used to resolve it, developing in different programming languages to UI design and many other interesting topics.

8.1 Improvements

Despite the success of the project in the development of a 3-dimensional N-body simulator, there are several ways in which this application could be improved both by optimizing the program and by adding some extra features, now we will cover some of them.

8.1.1 Optimizations

When it comes to optimizations, these should always be targeted at the issues that affect performance the most, in our case like we have seen that the **Trail size** is the parameter that has the biggest impact in simulator performance, to solve this issue, two solutions are proposed:

- Changing the way the orbit points are assigned by using the **SetPositions()** function in the **LineRenderer** component [41]. This allows for a more rapid assignment of the points that make up the orbit, which needs to happen every frame. This change has not been implemented because it would involve structural changes in the code and in the way camera **zoom** is calculated, and this issue was detected in the final stages of development.
- Since the optimization proposed above might not solve this issue, an alternative would be to speed up the iteration of the orbit setting in the **LineRenderer** with the techniques described in [42].

Many other optimizations could be done to improve code efficiency, graphical rendering, and many other aspects of the simulator, but these are all secondary changes, since the simulator performs satisfactorily except when using big Trail sizes.

8.1.2 Extra features

In these types of programs, hundreds of different features can be implemented, with varying degrees of importance and interest for the user. In our application we have implemented the features that we thought most important, such as a 3-Dimensional interactive space, real time rendering and others. Some features that were studied with interest but have not been implemented are:

- Fixed interval simulation, instead of letting the user run the simulation for as long as they want, and update it in real time, another option would be to allow the user to select a start and finish times and display the resulting simulation results.
- Allowing the user to export data at any given point in the simulation, this is very interesting for the user if, after experimenting with a simulation, they find an evolution that they would like to save.
- Allowing for video recordings to be made inside the application, this would enable the user to save a small video of the evolution of a simulation to be used in presentations, projects…

One of the main purposes of this simulator was to allow the user to compare and contrast different **numerical integrators**, to achieve this, a wide range of numerical integrators of different types, single-step/multi-step, implicit/explicit… , should have been developed. However, due to time and complexity constraints, only a small set of numerical methods has been implemented, and the programming of more methods would greatly improve the options the user can choose from.

8.1.3 New simulators

During the final stages of the development for this project, a new simulator for the N-body problem has been discovered, it is called "Harmony of the spheres" [43]. It is very interesting because it is a web-based simulator and contains several of the improvements over the existing simulators that this project has tried to implement.

Some of its features are a wide array of **numerical integrators**, smooth **3-dimensional** camera movement, a vast number of **existing simulations** and many other features. Though we cannot compete with this project, since it has been developed over a span of several years and many experienced people are working on it, it does serve as a benchmark for future projects of this nature since it is a piece of software of very high quality.

Despite its many polished features, in some aspects the simulator that has been developed in this project offers some improvements, some of them are:

- Free camera movement, as opposed to camera movement restricted to be anchored around a body.
- Stable orbit paths, in the "Harmony of the spheres" simulator, orbits will become incorrect when altering several parameters drastically.

- Direct control over some simulation parameters, as opposed to restricted changes in parameters.
- Visualization of metrics such as **Energy** and **Angular momentum**.

9. Bibliography

[1] K. R. Meyer, Periodic solutions of the N-Body Problem., Berlin: Ed Springer, 1999.

[2] B. H., Uber die Integrale des Vielkörper-Problems, Acta Math, 1887.

[3] A. Wintner, The analytical foundations of Celestial Mechanics, Princeton Univ. Press, 1947.

[4] J. C. Butcher, Numerical Methods for Ordinary Differential Equations, Wiley, 2003.

[5] E. Hairer, C. Lubich and G. Wanner, Geometric numerical integration illustrated by the Störmer/Verlet method, Acta Numerica, 2003.

[6] M. d. C. Peña Moreno, "Construcción de un simulador del sistema solar mediante la integración del problema de n-cuerpos," 2005. [Online]. Available: https://academica-e.unavarra.es/handle/2454/761.

[7] Mathworks, "Matrix operations in Matlab," [Online]. Available: https://es.mathdowks.com/help/matlab/math/basic-matrix-operations.html?lang=en.

[8] Mathworks, "Creation of a User Interface in Matlab," [Online]. Available: https://es.mathworks.com/videos/creating-a-gui-with-guide-68979.html.

[9] Mathworks, "Matlab GUI," [Online]. Available: https://es.mathworks.com/discovery/matlab-gui.html.

[10] NumPy.org, "Numpy Documentation," [Online]. Available: https://numpy.org/doc/.

[11] T. C. Omelette, "Python N-Body orbit simulation," [Online]. Available: http://www.cyber-omelette.com/2016/11/python-n-body-orbital-simulation.html.

[12] Nanalyze, "Unity Technologies – The World's Leading Game Engine," October 2017. [Online]. Available: https://www.nanalyze.com/2017/10/unity-technologies-leading-game-engine/.

[13] NumSharp, "NumSharp: The C# Numpy port," [Online]. Available: https://github.com/SciSharp/NumSharp.

[14] S. Axon, "Unity at 10: For better—or worse—game development has never been easier," 27 9 2016. [Online]. Available: https://arstechnica.com/gaming/2016/09/unity-at-10-for-better-or-worse-game-development-has-never-been-easier/.

[15] Indeed Editorial Team, "What is a User Interface?," 17 9 2021. [Online]. Available: https://www.indeed.com/career-advice/career-development/user-interface#:~:text=The%20user%20interface%20(UI)%20is,to%20receive%20maximum%20desired%20outcome.

[16] L. Stanley, "Signs that point to a poor user experience," [Online]. Available: https://www.resourcetechniques.co.uk/news/web-design/signs-that-point-to-poor-user-experience-101495.

[17] AstroGrav, "AstroGrav Astronomy Software," [Online]. Available: http://www.astrograv.co.uk/.

[18] Unity Technologies, "Scenes," 19 1 2022. [Online]. Available: https://docs.unity3d.com/Manual/CreatingScenes.html.

[19] Unity Technologies, "GameObject," 19 1 2022. [Online]. Available: https://docs.unity3d.com/ScriptReference/GameObject.html.

[20] Unity Technologies, "MonoBehaviour.Start()," [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html.

[21] Unity Technologies, "MonoBehaviour.Update()," [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html.

[22] Unity Technologies, "Canvas," [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html#:~:text=The%20Canvas%20is%20the%20area%20that%20all%20UI%20elements%20should%20be%20inside.&text=Creating%20a%20new%20UI%20element,a%20child%20to%20this%20Canvas.

[23] Unity Technologies, "Button," [Online]. Available: https://docs.unity3d.com/es/2018.4/ScriptReference/UI.Button.html.

[24] Unity Technologies, "Text Mesh Pro," [Online]. Available: https://docs.unity3d.com/Manual/com.unity.textmeshpro.html.

[25] Unity Technologies, "Input Field," [Online]. Available: https://docs.unity3d.com/2019.1/Documentation/ScriptReference/UI.InputField.html.

[26] GameBench Staff, "Game performance that matters," [Online]. Available: https://blog.gamebench.net/game-performance-metrics-that-matter.

[27] F. Diacu, "The solution of the n-body problem," 1996. [Online]. Available: https://www.math.uvic.ca/faculty/diacu/diacuNbody.pdf.

[28] S. R. T. John H. Reif, The Complexity of N-Body Simulation, Springer-Verlag, 1993.

[29] T. Akenine-Möller y E. Haines, Real-time rendering, A K Peters/CRC Press, 2004.

[30] Unity Technologies, "Line Renderer," [Online]. Available: https://docs.unity3d.com/Manual/class-LineRenderer.html.

[31] C. Monkey, "Unity tutorial - Create a graph," [Online]. Available: https://www.youtube.com/watch?v=CmU5-v-v1Qo&list=PLzDRvYVwl53v5ur4GluoabyckImZz3TVQ&index=3.

[32] Unity Technologies, "Optimizing Unity UI," [Online]. Available: https://learn.unity.com/tutorial/optimizing-unity-ui.

[33] J. Dobbin, "GPU vs CPU: What matters most for PC Gaming?," 2019. [Online]. Available: https://www.hp.com/us-en/shop/tech-takes/gpu-vs-cpu-for-pc-gaming.

[34] B. Barney, Introduction to parallel computing, Lawrence Livermore National Laboratory.

[35] Unity Technologies, "Requirements to run Unity 2019.1f," [Online]. Available: https://docs.unity3d.com/2019.1/Documentation/Manual/system-requirements.html.

[36] NASA, "Small-Body database," [Online]. Available: https://ssd.jpl.nasa.gov/tools/sbdb_lookup.html#/.

[37] S. B. Nicholson, The Trojan Asteroids, Astronomical Society of the Pacific Leaflets, 1961.

[38] J. Montaldi, "N-Body Choreographies," [Online]. Available: https://personalpages.manchester.ac.uk/staff/j.montaldi/Choreographies/about.html.

[39] C. Moore, Figure eight orbit, Science direct, 1993.

[40] D. B. R. Broucke, Periodic orbits in the Planar General Three-Body Problem, Celest. Mech., 1975.

[41] Unity Technologies, "LineRenderer SetPositions()," [Online]. Available: https://docs.unity3d.com/ScriptReference/LineRenderer.SetPosition.html.

[42] Microsoft Inc, "System Threading," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.threading.parameterizedthreadstart?redirectedfrom=MSDN&view=net-6.0.

[43] H. G. P. W. J. V. V. Darell A. Huffman, «Harmony of the spheres,» [En línea]. Available: https://gravitysimulator.org/.