OrbitCalc - A simplified version of the solar system

Tuesday 9th January, 2024 - 20:57

Henrik Klasen
University of Luxembourg
Email: henrik.klasen.001@student.uni.lu

This report has been produced under the supervision of:
Gabriel Garcia
University of Luxembourg
Email: gabriel.garcia@uni.lu

***Abstract*—With scientific endeavours becoming more cost intensive, there is a need to plan specific missions into minor details. In order of cutting cost to expensive missions, such as satellite launches and submarine drones, it is important to plan missions in high detail. Computational models can help to find the weak points of the mission, find extreme values of the devices and can help to set up a safe margin for the mission program. The development of one such computational model with a potential use case in space exploration is the topic of this Bachelor Semester Project. (5850)**

1. Introduction (190/400 words)

The importance of simulations in the scientific world is rising. Not only do they provide a less cost-effective way of testing real life scenarios in a virtual twin of the current model of the world. It is cheaper for example, to test new aerodynamics concepts in a virtual environment, where modularity is easier, instead of using wind tunnels. For these wind tunnel tests, the components have to be produced, the wind tunnel has to be powered on and a large amount of energy has to be used for powering the wind tunnel. Not only is this way of testing resource exhaustive, but also not very environmentally friendly, as a big portion of these prototypes will end up as scrap and can partly not even be used any more.
Simulations can provide near-real world environments, or at least emulate the behaviour of certain laws in physics. With these environments, we can reduce the amount of resources we use, by checking: is it the right way to achieve our goal? Can we optimize the behaviour of our test subject before producing it?

2. Project description ($\pm$ 10% of total words)

2.1. Domains

2.1.1. Scientific.

- *Computational Modelling*: This domain is all about breaking down real life environments and behaviours into equations and algorithms for simulating the environment as good as possible with certain simplifications.
- *Simulations*: Simulations are a computational approach to emulate real life behaviours of objects and environments on computers. They are used for example for optimizing the trajectories in space, simulating fluid dynamics or aerodynamics for optimizing the fuel efficiency of cars.
- *Scientific Computing*: Scientific Computing is an interdisciplinary discipline bound to develop approaches for algorithms, models and software. These are used for simulating real world behaviour as close as possible. This means, that computational modelling and simulations are both a subset of scientific computing, which is the broader scientific topic.

2.1.2. Technical.

- *Unity*: A state of the art game engine for three dimensional and two dimensional games. With its powerful, built-in physics engine, it is able to also create realistic simulations, such as digital twinning, fluid simulations, as well as large field gravitational simulations, which is the goal in this project.
- *Visual Studio*: A powerful code editor for all of the programming languages in the .NET Framework. This IDE has Unity integration, which simplifies the development of programs in Unity.
- *C#*: An object-oriented, higher level programming language in the .NET framework, which was heavily influenced by C++ and Java. C# is often used in cross-platform application development, game development and in Machine Learning applications.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The scientific part of this Bachelor Semester Project is about modelling a suitable,

simplified model of the solar system.
It will consist of three parts. The first part being about the fundamentals of computational modelling, which will take a look at the approach of breaking real world data down to a computable model. This model can be emulated by a computer program.
The second part is about modelling a model with some constraints and simplifications. This model will include the basics of orbital mechanics, as well as the fundamental concepts of Newtonian physics.
The scientific part will tackle the following scientific question: *How can scientists model a simplified simulation of the solar system?*
To find a satisfying answer to this question, it is necessary to answer the related questions:

- *What is a computational model?*: This question aims on answering the most basic and fundamental question of this project. The results of this question are required to later on build and model the computational model.
- *What environmental factors of the solar system have to be considered?*: This question aims on answering, *how* the model will look like. This includes the examination of the utilized simplifications, methodologies and an analysis of the factors influencing the selection of specific approaches.
- *How can the model be modelled?* : With this question, the goal is to find equations and algorithms that facilitate an accurate representation of the computational model. This part is meant to include all the orbital mechanics theory needed.

As the goal is to find satisfying answers to the questions above, a state-of-the-art study will be conducted. Based on this study, the subsequent resources are used to address the scientific question:

- [KaufmannFranzinMenegaisPozzer]: This publication is about the computational cost of large scale physics simulations. It will mainly be used in the section for the question "*What environmental factors of the solar system have to be considered?*", as it is about modelling the computational model itself.
- [MurinKompisKutis]: The main point in using this publication is to find out a way to properly model a computational model. It provides many examples for computational models for several different simulations, which might help when modelling the solar system later in the project.
- [HeisterRebholz]: This resource is about scientific computing. As its objective is to teach undergraduate students the fundamentals of scientific computing and numerical methods to model a simulation or whichever model needed, it is well suitable for this Bachelor Semester Project. It will be used for getting some insights on how scientific computing is used today and how it *can* be used in this project.

With the use of these resources, the scientific part will focus on delivering the most useful and crucial information to develop a satisfying model of the solar system. The results of this part will later on be used in the technical part of this bachelor semester project, which is all about the implementation of a three dimensional solar system in Unity. In the last section of this part of the project, we will take a look at which factors cannot be simulated in the model developed and which parts could possibly be added in future extensions, but which exceed the scope of this project.
The overall goal of this deliverable is to ensure, that the reader can understand the design decisions in the technical part. The results of this section will be used as a fundamental basis of the technical deliverable and the implementation.

2.2.2. Technical deliverables. The technical part of this Bachelor Semester Project is about the implementation of the computational model developed in the scientific part. In fact, it is about the implementation of a virtual version of the solar system.
While the scientific part is more about the "*What do we want to simulate?*", this part deals with "*How do we implement what we want to simulate?*". The goal with this deliverable is, to get to know how to implement scientific computing models in a given environment.
As for the technical deliverable, there is not as much literature such as official scientific papers or publications. The main source of information will be the scientific deliverable, e.g. for the equations describing the gravitational law. Next to that, the official Unity Documentation (linked in [Unity]) will be used for any further documentation regarding Unitys API.
The technical deliverable will provide some key features:

- *Gravity*: According to the Newtonian gravitational law. Further details will be in the scientific part of the report.
- *Real scale*: The scale of the objects in the solar system will correspond to their actual scale in the real solar system. This will (hopefully) ensure a higher degree of realism and ease gravitational calculations, which are based on the distances.
- *Realistic lighting*: As the sun is more or less the only high intensity light source in the solar system, the simulation will use a point light source which will shine light from the suns position.
- *User controllable Satellite*: This will help the user experience navigating a satellite in a three dimensional solar system with objects influencing the trajectory of the space probe.

The program will eventually include a Graphical User Interface, for which some parameters are needed. So far it is planned to include the parameters for two objects, mainly x-y-z position of the objects. It will be determined in this section, which data is going to be displayed in a light-weight interface, which provides the user enough information for controlling a satellite and camera (exact number of cameras to be determined in the report). For example, the data, currently planned to be displayed are:

- Velocity of user controlled camera: Measure how fast the user is going in the solar system
- Velocity of the satellite: Enable the user to calculate where the space probe will be at which point in time.
- Distance of the satellite to the sun: Locate the position of the satellite
- Coordinates of the satellite: Locate the satellite in a three-dimensional space
- Coordinates of the camera: Locate the user controlled camera in three-dimensional space
- Distance camera – Sun: Locate the camera on a relative basis in the solar system
- Labels of the planets: Locate the planets relative to the camera
- Planet orbit paths: Locate the orbits of the planets
- Satellite trajectory: Display the points, the satellite already passed to estimate the further trajectory

As the user will be able to control the camera, it will be required for this project, to map keys on the keyboard to certain movements, to enable the user to experience the virtual solar system. As of now, the keys are mapped as follows:

- [W] – Forwards movement
- [A] – Left movement
- [S] – Backwards movement
- [D] – Right movement
- Left [⇧] – Upwards movement
- Left [ctrl] – Downwards movement
- [←] – Rotation Left (Z-Axis)
- [→] – Rotation Right (Z-Axis)
- [↑] – Rotation Upwards (X-Axis)
- [↓] – Rotation Downwards (X-Axis)

Furthermore the satellite will be controllable by the user, to ensure, that the user gets a better understanding of the complexity of navigating in a three-dimensional space under the influence of gravitational pull to other celestial bodies.

3. Pre-requisites

Although all used concepts will be explained in the report, it could be helpful, yet not mandatory, to understand the basic notions of the following topics. These are already known by the author before starting.

3.1. Scientific pre-requisites

Before starting the Bachelor Semester Project, the student had some experience in orbital mechanics, which includes the basics of (gravitational) force vectors.
Next to that, the student had some experience in constructing mathematical and physical models with certain simplifications for computational ease.

3.2. Technical pre-requisites

As the technical deliverable is about the implementation of a basic solar system simulation in Unity, the student already knows how to work with Unitys' scaling, transformation and rotation system and their script application programming interfaces (APIs) in C#. Also the basics of game and simulation development are known, as it was part of a small course in the Computer Science Summer Camp at Hochschule Trier.

4. A computational model

4.1. Requirements (122/150 words)

The requirement for this section is to explain with the given resources, the concept of a computational model. This includes the fundamental basics of how it came to be, as well as some insights on how they are usually developed and the scientific significance of them. For the explanation, it should be well understandable to any person, who understands the basics of computer science and is willing to read this bachelor semester project. This is to ensure that the author has well understood the topics of this section, as it will be the fundament of the upcoming sections and the technical part.
The results of this section will provide instructions on how to build a computational model of any kind.

4.2. Design (213/300 words)

As there are only few resources providing a general overview on computational models, this section of the report will mainly consist of information found in [Garrido].
Here, the following chapters will be used to find results to the above scientific question:

- Chapter 1: Problem Solving and Computing:
  - 1.1: This is useful for finding the motivation for computational models. It explains the fundamental issue and already hints at, what is needed to develop such a model.
  - 1.2: As it is also the goal of this report to find a step by step set of instructions for building a computational model, this section already provides a basic skeleton for this set of instructions.
    Regarding 4, the outline of steps will be used to construct a more detailed instruction set than given in the resource.
  - 1.3: This section will be used to find different ways to tackle different mathematical difficulties. It will help to set up a guideline for transforming the mathematical model with formulas into a computational model with algorithmic approaches.

Next to these information, the observations made during the first weeks of the semester in the development of the application will be used. This will not only provide meaningful results, but also show the real world development of the models.

4.3. Production (514/400 words)

A computational model is a mathematical and algorithmic construct describing real world behaviours in formulas and procedures. With this in mind, finding a computational model needs a few steps for developing the model. This process is called *computational modelling* and involves a numerous count of observations to be made.
But all in all, the steps for setting up a computational model are quite general, as they are usually applied universally and then tuned to the individual model depending on the models complexity.

4.3.1. Observation. The basis of modelling a computational model emulating an environment is observation. This enables to understand the challenges of the environment and to understand the problem. With these in mind, it is also useful to observe certain behaviours of different objects and note them down as much as possible, as it will help in 4.3.3 to find appropriate formulas to describe these behaviours.
Observation also enables to figure out factors with a minor impact, which might have to be left out for simplifying the model.

4.3.2. Problem Description. (acc. to [Garrido], Ch. 1 Section 2) The problem description is one of the most important steps of designing a computational model. By providing a detailed problem description, taking into account all the important aspects, the model is supposed to take into account. This helps to prioritize different observations for later considerations of which aspects are must-have and which ones are nice-to-have aspects.

4.3.3. Mathematical foundations to a model. To further advance in the model, it is usually required to find formulas describing the observed behaviours of the model.
As most models are used in interdisciplinary research, such as physics, mathematics, chemistry and biology, it often implies using existing formulas of this discipline to find a well-working environment for the computational model. The formulas for this have to be as complete as possible, as it will be the basis for later on designing the algorithmic approaches in the next step (4.3.4).

4.3.4. Algorithmic design. As the mathematical foundations are known from 4.3.3, this step is about the way of computing, i.e. the algorithm. As computers are usually resource and power limited, this step is very important. There are several factors to consider when designing an algorithm for a computational model, such as recursive and iterative approaches.
In general, iterative approaches are more favourable, as they usually have a lower space complexity while providing the same results as recursive methods or at least provide a similar degree of accuracy.
Designing the algorithm also implies to simplify the mathematical model where computationally needed to reduce the overall space and time complexity of the program.

4.3.5. Guideline for computational models. With all of these steps in mind, it is possible to reduce them to easy questions as a guideline:

1) What do we see? Which behaviour is observable?
2) What do we want to do? What is the problem we try to solve and which properties does the model need?
3) What formulas describe the behaviour of our model?
4) How do we implement the model in terms of algorithmic approach?

4.4. Assessment (0/150 words)

Provide any objective elements to assess that your deliverables do or do not satisfy the requirements described above.

5. Observation and Problem Definition

5.1. Requirements (121/150 words)

This section requires to execute steps 1-3 from 4.3.5 in order of finding a computational model for the solar system. It will be required to provide subsections for every step of the guide, similar to the sections of 4. These sections will contain the scientific findings and important points of the model.
Not only the scientific findings of this section will play a central role for this section, but also how they correlate to the steps found in 4.3.5 and *why* the results found, are important. Based on this, the report will deliver the most crucial formulas of the model in this part, which are a necessity in 6. Also this section will provide some general simplifications, such as negligible factors.

5.2. Design (148/300 words)

As section 4 already contains a detailed look at computational models and computational modelling, this section will use the results of section 4 and combine it with the formulas describing the solar system. This will then provide a mathematical/computational model of the solar system, for which the goal is to model algorithmic approaches to emulate the behaviour of the solar system. This modelling will be step by step according to the guideline found in 4 and will illustrate the use of it on this example.
Additionally, next to the chapters used in the preceding section, this section will use the following sections of Chapter 1 of [Garrido]:

- 1.4: As this section is a more detailed description of Computational modelling with more details on the development process, which are not needed in 4.
- 1.8: Section 1.8 provides a software development approach to the problem of computational models.

5.3. Production (400/400 words)

With the plan of how to build a computational model, we can now start constructing the basic framework of the model for this project.

5.3.1. Observable behaviour. Let us assume, we want to model the computational model for a three-dimensional space. Our solar system has 10 major objects, namely the sun and all planets. These objects have different behaviours each:
Sun: The sun is the centre of our model, as it also is the centre of the solar system. This object, although in reality not static, stays in more or less the same place in the closed system (if we assume, that the solar system is a closed system). It is also very important to note, that the sun has a mass and a size.
Planets: The planets revolve around a common centre of mass. Each orbit has an inclination α, a major semi-axis a, a minor semi-axis b and an eccentricity ϵ. These orbits are determined by the attraction between all bodies. As the attraction between bodies is determined by mass m and distance r, each planet has a mass, a size and a position. It is also observable, that the orbits do indeed change slightly over time, due to effects, such as orbital decay or certain constellations pulling the body more into a certain direction, thus impacting the orbit.
Satellite: The satellite has a mass, a position and a velocity or acceleration. With these values, it is able to revolve around an other celestial object. All other celestial bodies are *theoretically* also attracted to it, but since the mass of the satellite object is negligible, it can be left out without loss of generality.

5.3.2. Problem definition. The model we want to develop will have to simulate the behaviour of the solar system. The main focus will have to be the satellite objects trajectory, such that it will suffice to simplify the behaviours of the remaining bodies in the solar system, such as the sun and planets. For the celestial bodies, simplifications which do not interfere with the accuracy have to be found. Furthermore it will be necessary to minimize heavy weight calculations to save on compute power to be able to run the simulation on a lower end machine.
Since it is easier to see odd behaviour, the simulation should also be displaying all calculated values in a three dimensional space, i.e. the movement should be visible.

5.4. Assessment (0/150 words)

Provide any objective elements to assess that your deliverables do or do not satisfy the requirements described above.

6. Mathematical approach and simplifications

6.1. Requirements (107/150 words)

This section will find formulas for the model described in 5.3.2. It is required to explain every formula and to reason about the specific model. This will finally lead to the model completion and finalize design of the model to be implemented in 7 and later sections. Furthermore will this section give details like scale of the model and assumptions to simplify the computations to ensure ease of computation. It will be required to give the specific formulas and from these, to elaborate, why this formula is usable or not useful for this model. Then from results in 5.3.1 it will be necessary to deduct useful simplifications.

6.2. Design (95/300 words)

This section will take a look at the scale of the numbers used in the model developed in 5 and point out floating point and precision issues. Furthermore it will simplify some equations for ease of computation and also simplify some behaviour into constants.
Specifically, this section will use Chapter 1 of [Garrido]. Section 1.11 of this chapter is especially important, as it provides the necessary information regarding floating point precision, errors and accuracy.
To find issues in precision, the sections 4 and 5 will be used to show the errors or complex calculation made.

6.3. Production (600/400 words)

To build a mathematical model of the simulated environment, we need to look at the physics behind the environment. This involves finding formulas for the observed behaviour. To structure this in a similar way to the Observable behaviour section (5.3.1), we will take a look at the model object for object.

6.3.1. Sun.
Theoretical aspects: The centre of mass, or also known as barycentre, of the solar system is located close to the sun. This means that the sun also revolves around this barycentre, which changes over time. So in theory, the sun is not a fixed object in the solar system, which means, we would need to calculate the position of the sun in every single moment.
Practical aspects: Since vectorial computations are rather resource heavy and are quite complex to compute, we would like to simplify the model here. Let us therefore assume, that the position of the sun is fixed and it represents the centre of mass of the solar system.

6.3.2. Planets.
Theoretical aspects: We know from 5.3.1, that the planets revolve around the sun. Their orbits are approximately elliptic, though not fully elliptic, since the focus of the ellipsis is the barycentre of the system, which changes over time.

This leads to small variations in the orbits.
Practical aspects: As we already simplified the sun as a static object, we can further simplify this system by assuming, that the planetary orbits do not change over time. The variations in the orbits are related to the variations of the barycentre of the system. As we already cancelled these variations out in the simplification of the sun, we can also do this here.
Leaving out these small variations leads to a formula for orbits describing elliptical shapes with a certain angle α.
$$\begin{pmatrix} f_x + (a_{mj} \cdot cos(0.005 \cdot v) \\ 0 \\ f_z + (a_{min} \cdot sin(0.005 \cdot v) \end{pmatrix} \cdot \begin{pmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

6.3.3. Satellite.

Theoretical aspect: As mentioned in 5.3.1, the satellite has a mass m_{sat}. It is under the gravitational pull of all celestial bodies, such as sun and planets. This means, that it is a necessity to calculate the distance r_i to all objects O_i. Therefore let the celestial bodies O_i be numbered from 1 to 10 and their masses be m_1 ... m_{10} with distance r_1 ... r_{10}. Then the forces influencing the satellite would be:

$$\vec{F} = \sum_{i=1}^{10} \gamma \cdot \frac{m_i \cdot m_{sat}}{\vec{r_i}^2}$$

This formula can be simplified, as γ and m_{sat} are all factors of every portion of the sum, so they can be seen as a constant factor. This means, that the simplified formula would be:

$$\vec{F} = \gamma \cdot m_{sat} \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}$$

Since the goal is to find a formula for $\vec{v}$, we need to transform $\vec{F}$ to $\vec{a}$. Assuming that $\vec{a} = \frac{\vec{F}}{m_{sat}}$, which leads to:

$$\vec{a} = \frac{\vec{F}}{m_{sat}} = \frac{\gamma \cdot m_{sat} \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}}{m_{sat}} = \gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}$$

Acceleration $\vec{a}$ can be transformed to velocity $\vec{v}$ with the formula:

$$\vec{v}(t) = \vec{a} \cdot t^2 + \vec{v_0}$$

Practical aspects: Even though we want to minimize vectorial calculations, we cannot simulate the system without them. But to make it more efficient in terms of calculations, we have to see, where we can use iterative approaches.
Since our final formula for the vectorial velocity $\vec{v}$ has $\vec{a}$ inside, we can simply replace $\vec{a}$ by the formula for $\vec{a}$, such that we do not introduce a new variable for $\vec{a}$:

$$\vec{v}(t) = (\gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}) \cdot t^2 + \vec{v_0}$$

For transforming this formula into a usable formula for the calculations later on, we need to replace $\vec{v_0}$ by something already known or calculated. Since we already know, that $\vec{v_0}$ is the starting velocity at a point t_0, we can reformulate the formula as:

$$\vec{v_n}(t) = (\gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}) \cdot (\Delta t)^2 + \vec{v}_{n-1}$$

Where Δt is the time difference between the current and the previous calculation and $\vec{v}_{n-1}$ is the velocity calculated in the previous iterative step.

6.4. Assessment (0/150 words)

Provide any objective elements to assess that your deliverables do or do not satisfy the requirements described above.

7. The solar system

7.1. Requirements (105/150 words)

In this section it is required to find explanations and pseudocode for the implementation of the solar system. This section will focus on the implementation of the "stable orbit script" (SOS) which is mainly responsible for the elliptical orbits in this system. It will also be required to look at the results from 5 and 6 as they provide the *theoretical* aspects of this part. With these as the main points covered, it will also be necessary to take a look at the game objects and the addition of a script to such an object. This will include a detailed glimpse at Unitys scripting API.

7.2. Design (97/300 words)

This section will mainly take results from 6.3.1 and 6.3.2 to find code snippets corresponding to the formulas. It will be strongly oriented along [Unity], as it will be the environment, in which the simulation is going to be written in.
Not only is one requirement to find the pseudocode, but also to explain it, which will also be done using very simple syntax and clear explanations. This section represents Step 4 of the Guide from 4.3.5.
Since this guide is also derived from [Garrido], it also represents the "Algorithmic design"-Step in Chapter 1.3 of the book.

7.3. Production (490/400 words)

To start constructing the model, as defined in 6, we need to find corresponding code for the behaviour described in 5 and the formulas found in 6.3.1 and 6.3.2.
Let us therefore start with the Sun, as it is quite straight forward.

7.3.1. Sun - Algorithmic approach and implementation.
As defined in 6.3.1, the sun should be modelled as a static object. This means, that we create a sphere in the simulation environment. As the sun is also a light source in the solar system, it is also a light source in the simulation.
Given, that the sun does not provide a constant light, but some variations in its intensity, we can model this as a particle effect with large particles to hide a part of the light emission in one direction.
As we also defined, that the sun has a mass, it gets the property of a rigid body, which simply adds a floating point number indicating the mass of the body. With the size of the star being the X-Y-Z dimensions, we can also check this point of the list of properties.

7.3.2. Planet - Algorithmic approach and implementation.
Similar to the sun, a planet has a mass and a size. This implies, that the planet object also has the property of a rigid body object. As we know from 6.3.2, we want to simplify the orbits of the planets to an ellipsis with the sun in one of the foci. The formula for an ellipsis in three-dimensional space is also given in 6.3.2 as:

$$\begin{pmatrix} f_x + (a_{mj} \cdot cos(0.005 \cdot v) \\ 0 \\ f_z + (a_{min} \cdot sin(0.005 \cdot v) \end{pmatrix} \cdot \begin{pmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Given the fact, that calculating with matrices in programming languages such as C#, is difficult, we will use the predefined data structure **Vector3D** from Unity. This data structure is of the type 3-tuple of type float, in the form (X-Coordinates — Y-Coordinates — Z-Coordinates). For manipulating the position of the planet object (a sphere), it is necessary to use the **transform.position(Vector3D)** method, which takes a Vector3D as input and adds it to the objects current position. This leads to a the position to change coordinates, which ultimately means, that the object changes coordinates.
The algorithm
As already mentioned, Unity provides methods and classes for three dimensional space. For us, the following classes and methods are important:

- Quaternion: The class of rotation matrices in Unity.
  - Euler: Takes an angle as input and outputs the corresponding rotation matrix for this angle. The angle is measured in degree
- Vector3(D): The Vector3 or Vector3D class is the implementation of three-dimensional vectors in the Unity game engine.
  - **transform**: The transform attribute describes the position, rotation and the scale of the game object. To manipulate the position of the object, transform.position has to be called. Usually, the transform attribute is associated with the game object, on which the script applies, i.e. script applies on Mercury, then it takes the position of Mercury etc.

Translated to pseudocode, this would imply the following script.

Algorithm 1 Planet orbits on ellipsis

```
1: function FRAME UPDATE
2:     α ← |v⃗|    ▷ Calculate how much the object moved
3:     x ← focus.x + (mj_semiaxis · cos(alpha · 0.005)
4:     y ← focus.y + (min_semiaxis · sin(alpha · 0.005)
5:     p⃗ ← (x, 0, y)        ▷ Let p be the vector of position without rotation
6:     Q ← α                 ▷ Let Q be the rotation matrix
7:     position ← p⃗ · Q
8: end function
```

7.4. Assessment (0/150 words)

8. Movement in three dimensions and time control

8.1. Requirements (75/150 words)

This section will be mostly about the camera, camera controller and the time controller scripts of the Unity project. It will focus on the coding aspects as well as some mathematical background from Linear Algebra, such as Vectors and point transformations. Since this section is not based on physics, it does not rely on results from 5 and 6. Instead, it will illustrate based on pseudocode the algorithmic approach of movement in three dimensional space.

8.2. Design (85/300 words)

This section mostly relies on [Unity] and Linear Algebra for coming up with formulas and code. Next to this, simple formulas from mechanical physics are used to show the relation between time and movement. This will result in a formula based on a time t and a time factor f, such that all functions, such as $s(t), v(t), a(t)$ are dependent on $f \cdot t$. To conclude, these formulas will then be translated into code, understandable by a computer using the Unity Programming Interface.

8.3. Production (575/400 words)

8.3.1. Modelling the camera.
As the camera object is not part of the solar system, let us assume, that the forces present in the solar system do not affect the camera. This implies that camera movement has a constant velocity v. As v is constant, we know from basic mechanical physics, that

$$s(t) = v \cdot t$$

Since we want to move the object in three dimensions, we need to transform all variables, except the time t into three-dimensional vectors. This implies, that in order of transforming the given formula for translation into a three-dimensional formula for translation, we will only have to use the vector $\vec{v}$, since t can be seen as a scalar. This means, that our formula is:

$$\vec{s}(t) = \vec{v} \cdot t + \vec{s_0}$$

where s_0 is the previous position of the camera.
Given, that we can also rotate the camera, we need to introduce a rotation matrix Q, which is given by the two rotation matrices around the x-axis and the y-axis. Let Θ be the rotation around the x-axis and ϕ be the rotation around the y-axis.

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\Theta) & -sin(\Theta) \\ 0 & sin(\Theta) & cos(\Theta) \end{pmatrix} + \begin{pmatrix} cos(\phi) & 0 & sin(\phi) \\ 0 & 1 & 0 \\ -sin(\phi) & 0 & cos(\phi) \end{pmatrix}$$

To compute the position of the object after the rotation of the camera, it is necessary to multiply $\vec{v}$ with Q. This ensures, that the movement is relative to the rotation of the camera and not absolute, which means, that it would only move in the direction relative to the coordinate system (i.e. instead of moving forward on press of W with a 90° rotation relative to origin, it would move sidewards relative to the camera). This implies, that the final formula, with Q dependent on Θ and ϕ, would be:

$$\vec{s}(t) = Q(\Theta, \phi) \cdot \vec{v} \cdot t$$

To simplify the above formula in words: Instead of moving in an absolute coordinate system, relative to the origin, we define the camera as the origin of a new coordinate system, where x, y and z are always forwards/backwards, sidewards and up-/downwards. This means, that controlling the camera is a bit more natural, as it is always going exactly where the user is expecting it to move.

8.3.2. Time warp.

The idea behind time warp in this project is, to speed up the simulation, while still having the same physics apply to them. This basically means, that any formula in dependence on Δt (time-difference) will have to have a factor f to apply the time warp.
The main intention behind this is, to minimize the time of observation of some behaviour, such as planetary constellations which might affect the satellite orbit.
The only object, on which time warp does not apply is the main camera, as it does not have physical properties, because it is only a tool of visualization of the program.
To control the time warp factor, there is a script which controls every script dependent on time. For the planet orbit algorithm for example this would mean, that $\alpha = |\vec{v}| \cdot f$ instead of $\alpha = |\vec{v}|$. This implies, that the planet will move further each frame, which eventually results in a higher velocity (in the model it stays the same, as all objects speed up by the same factor). In the end, this means, that all physical objects (i.e. celestial bodies and satellite) are affected by this, such that it does not introduce errors.
The keys for time warp are:

- 1 - Decrease time warp factor
- 2 - Increase time warp factor

8.4. Assessment (0/150 words)

9. Satellite and gravity

9.1. Requirements (79/150 words)

For this section it is required to provide the most important explanations on how gravity and the satellite object are implemented in the editor. This includes (Pseudo-)Code snippets for the most important functions. Also section 8 has to be taken into account, as the satellite object has to move in three dimensions.
Most importantly, this section takes results from 5 and 6 to provide an answer to the question: *How is satellite movement implemented in this program?*

9.2. Design (87/300 words)

This section will focus on the technical aspects of the scientific findings from 6.3.3. Next to the algorithmic approach (as explained in 4.3.4), this section will take the time-warp feature into account, as well as satellite controls (prograde and retrograde). To conclude the technical part of the project, we will take a look at how the forces are calculated (with the formulas from 6.3.3) and how they apply to the object. The main resource for this section will be the previous sections and [Unity] for elaborating how the algorithm works.

9.3. Production (672/400 words)

As already discussed in 6.3.3, the satellite cannot be simplified to static orbits, as it is very much influenced by the bigger celestial bodies. With this in mind, there is a three step approach to coming up with the algorithm for the satellite object.

1) Find all celestial bodies in the enclosed system, access the relevant data and plug it into the formula.
2) Find a velocity v_0 and a gravitational constant γ, which enable the satellite to have a stable orbit (i.e. to not crash into the sun).
3) Implement Prograde and Retrograde burning by scaling the vectors and time warp.

To recall the relevant formula:

$$\vec{v_n}(t) = (\gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}) \cdot (\Delta t)^2 + \vec{v}_{n-1}$$

Given this formula, we can observe, that we require each celestial bodies position for calculating the distance r_i and the celestial bodies mass m_i, as well as the satellites previous velocity vector $\vec{v}_{n-1}$.

9.3.1. Determining celestial bodies, fetching data and finding constants.
Our model has ten celestial bodies (the sun, all planets and Pluto). Since we want to simplify our algorithm as much as possible, we can save all celestial bodies in a game object array, over which we can iterate.
As it is necessary to access this array every frame update, it is the most resource efficient, to initialize this game object array upon startup of the program. It is possible to do this, as the game objects do not change, but only the attributes get updated to different values.
Let us look at the important attribute classes for an example body. Given, that we want to calculate the distance from the satellite to the celestial body, we need the position.

Figure 1: Transform properties of an example celestial body.

The transform property is given in every game object and determines its position, scale and rotation. The algorithm will access this property to fetch the position and calculate the difference vector. The resulting vector is $\vec{r_i}$.

Figure 2: Rigid body class determining mass.

The rigid body class has the public attribute mass. Instead of defining it on our own, which would not have been an issue since it is just a floating point number, we access this attribute. In doing so, it might enable more future features if needed. The mass stored in the rigid body attribute will become the value m_i for plugging into the formula.
Now, that all necessary data is fetched from the game objects after iterating over the array of celestial bodies, it is time to determine the constants $\vec{v_0}$ and γ. As the scale of the system is 1:1, but the mass-size proportion is not 1:1, γ acts as a correcting factor. Instead of using the normal gravitational constant of $6.6743 \cdot 10^{-11}$, which is a very small number resulting in very small amount of force, the factor will be 66.743. We simply multiply by 10^12. The value of this was found in the trial and error method. Single precision floating point numbers are precise enough to represent this.
For v_0, we have the condition, that it enables the satellite object to have a stable orbit around the sun. If v_0 is too small, the satellite will have a collision course with the sun. If it is too large, it is going on an escape trajectory, eventually reaching a space, where the gravitational pull of the sun and all other celestial bodies become negligible.

9.3.2. Prograde and retrograde orbit manipulation.
A spacecraft can alter its orbit by altering the vector of the resulting force. For prograde and retrograde orbit manipulation it is done by scaling the velocity vector by a scalar factor p. For this the following rules apply:

- If $p < 1$: orbital velocity gets smaller, resulting in a lower orbit. This is a retrograde orbit manipulation.
- If $p > 1$: The orbital velocity gets higher, resulting in a higher orbit apoapsis. This is a prograde manipulation.

To apply this in the formula, we need to introduce p.

$$\vec{v_n}(t) = p \cdot ((\gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}) \cdot (\Delta t)^2 + \vec{v}_{n-1})$$

As we did not yet introduce time warp in this formula, it is necessary to change Δt to $f \cdot \Delta t$.

$$\vec{v_n}(t) = p \cdot ((\gamma \cdot \sum_{i=1}^{10} \frac{m_i}{\vec{r_i}^2}) \cdot (f \cdot \Delta t)^2 + \vec{v}_{n-1})$$

9.4. Assessment (0/150 words)

Acknowledgment

The author would like to thank his tutor for helping and guidance in the project and the Hochschule Trier for the Computer Science Summer Camps 2019 and 2021 which guided him to Computer Science and simulations.

10. Conclusion

The conclusion goes here.

11. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:

    1) Not putting quotation marks around a quote from another person's work

2) Pretending to paraphrase while in fact quoting
3) Citing incorrectly or incompletely
4) Failing to cite the source of a quoted or paraphrased work
5) Copying/reproducing sections of another person's work without acknowledging the source
6) Paraphrasing another person's work without acknowledging the source
7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
8) Using another person's unpublished work without attribution and permission ('stealing')
9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

[Armstrong and Green(2017)] J Scott Armstrong and Kesten C Green. Guidelines for science: Evidence and checklists. *Scholarly Commons*, pages 1–24, 2017. https://repository.upenn.edu/marketing_papers/181/

[BiCS(2021)] BiCS Bachelor Semester Project Report Template. https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2021).

[BiCS(2021)] Bachelor in Computer Science: BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2021)

[Armstrong and Green(2017)] J Scott Armstrong and Kesten C Green. Guidelines for science: Evidence and checklists. *Scholarly Commons*, pages 1–24, 2017. https://repository.upenn.edu/marketing_papers/181/

[KaufmannFranzinMenegaisPozzer] Lorenzo Schwertner Kaufmann, Flavio Paulus Franzin, Roberto Menegais, Cesar Tadeu Pozzer. Accurate Real-Time Physics Simulation for Large Worlds. *Universidade Federal de Santa Maria, Santa Maria, Brazil*, 2021.

[MurinKompisKutis] Justin Murin, Vladimir Kompis, Vladimir Kutis. Computational Modelling and Advanced Simulations. *Springer*, 978-94-007-0317-9-1, 2011.

[HeisterRebholz] Timo Heister, Leo G. Rebholz. Scientific Computing for Scientists and Engineers. *De Gruyter*, 2023.

[Garrido] Jose M. Garrido Introduction to Computational Models with Python *Chapman and Hall/CRC*, 2016.

[Unity] https://docs.unity3d.com/Manual/index.html

12. Appendix

All images and additional material go there.

12.1. Source Code