Contents

[Analysis 2](#_Toc129964006)

[The Problem 2](#_Toc129964007)

[Introduction 2](#_Toc129964008)

[Background to the Problem 3](#_Toc129964009)

[Clients and Other End Users 4](#_Toc129964010)

[Interview 4](#_Toc129964011)

[Limitations to Avoid and Acceptable Limitations 5](#_Toc129964012)

[Comparisons of Potential Models 6](#_Toc129964013)

[Time Increment Model 6](#_Toc129964014)

[Derived Model 9](#_Toc129964015)

[Overall 15](#_Toc129964016)

[Observations of an Existing System 15](#_Toc129964017)

[Objectives 17](#_Toc129964018)

[The Implemented Model 18](#_Toc129964019)

[Explanation of the Scientific Model 18](#_Toc129964020)

[Prototype 19](#_Toc129964021)

[Entity-Relationship Model 20](#_Toc129964022)

[Documented Design 21](#_Toc129964023)

[Data Flow Diagram 21](#_Toc129964024)

[User Interface Design 23](#_Toc129964025)

[Main Form: 23](#_Toc129964026)

[Add Body Form: 24](#_Toc129964027)

[Remove Body Form: 27](#_Toc129964028)

[Export Body’s Ephemeris Form: 27](#_Toc129964029)

[Controller Form: 27](#_Toc129964030)

[Information Form: 28](#_Toc129964031)

[Class Structure 28](#_Toc129964032)

[System Flowcharts 31](#_Toc129964033)

[Main Flowchart: 31](#_Toc129964034)

[Timer Tick Flowchart: 32](#_Toc129964035)

[Notes About Other Flowcharts 34](#_Toc129964036)

[Data Structures 34](#_Toc129964037)

[BodyTree 34](#_Toc129964038)

[Algorithms 34](#_Toc129964039)

[Technical Solution 39](#_Toc129964040)

[Foreword 39](#_Toc129964041)

[MainForm.cs 40](#_Toc129964042)

[Tree 53](#_Toc129964043)

[Body.cs 76](#_Toc129964044)

[Fixed Body.cs 76](#_Toc129964045)

[Moving Body.cs 77](#_Toc129964046)

[OrbitInformation.cs 102](#_Toc129964047)

[NotEnabledException.cs 103](#_Toc129964048)

[Matrix3x3.cs 103](#_Toc129964049)

[AddBodyForm.cs 104](#_Toc129964050)

[ControllerForm.cs 113](#_Toc129964051)

[ExportBodyEphemerisForm.cs 115](#_Toc129964052)

[InformationForm.cs 116](#_Toc129964053)

[RemoveBodyForm.cs 117](#_Toc129964054)

[AddBodyForm.Designer.cs 118](#_Toc129964055)

[ControllerForm.Designer.cs 139](#_Toc129964056)

[ExportBodysEphemerisForm.Designer.cs 142](#_Toc129964057)

[informationForm.Designer.cs 144](#_Toc129964058)

[MainForm.Designer.cs 148](#_Toc129964059)

[RemoveBodyForm.Designer.cs 152](#_Toc129964060)

[Testing 154](#_Toc129964061)

[Plan 154](#_Toc129964062)

[Tests of Objective Completeness 154](#_Toc129964063)

[Evaluation 168](#_Toc129964064)

[Objectives Completion 168](#_Toc129964065)

[Closing Discussion 169](#_Toc129964066)

# Analysis

## The Problem

### Introduction

My clients are interested in astrophysics and simulation games.

They are aware that in a lot of simulation games, especially that which simulate astrophysics and rocketry, many inaccuracies are made to make the games more approachable to new players. They however wish to simulate astrophysics in a more complicated way, such as they can input real world data and produce simulations which a lot of software simply don’t allow.

This project aims to provide an orbital motion simulation engine that avoids approximations, as well as giving a basic user interface to modify the simulation with.

An interview was done with the clients to obtain more information on their specific requirements for the project’s simulation capabilities (see ‘Interview’).

There are a few limitations which still must be made due to the complexities of astrophysics; the clients discussed the key limitations they wished to avoid as to guide how the final product would simulate (see ‘Limitations to Avoid and Acceptable Limitations).

To give more of a background to the problem as well as showing the derivation of particular used equations, both the disliked standard model and the new derived model are derived and discussed (see ‘Comparisons of Potential Models’).

### Background to the Problem

Orbital motion describes the motion of objects in space. Derived from celestial mechanics, which historically mostly referred to the motion of planets, it has been a popular field of science for centuries. It is often used to help predict the existence of large masses in space, the movement of spacecraft, and much more. In more modern-day applications, it has been subject to being simulated in various ways, due to it often being difficult to represent without complicated mechanical devices or precision drawing in previous years.

Simulation software is often used to represent high level physics and maths in a much more human-understandable way. It not only abstracts large amounts of maths from a user, but it also gives the results calculated a visual meaning.

In recent years, space and rocketry simulators have become a lot more popular in a mainstream audience. Games such as Universe Sandbox and Kerbal Space Program have become quite well known and allow a user without large understanding of astrophysics concepts to still model and play around with simulations.

Unfortunately, these popular games often have drawbacks. Often accuracy is sacrificed to make the simulations much more user friendly and simple; this is in part why they have gotten quite popular. This has however drawn a lot of attention away from developing the high level simulations that can still be used by a general user, and as such my clients felt they were looking for a product that doesn’t exist; they were looking for an orbit simulator that used Newtonian and Keplerian mechanics together to accurately model orbits instead of having to rely on somewhat inaccurate approximations provided by other software.

This project aims to create a simulation engine that provides a user to use higher level physics to get a more accurate model of orbital motion. This project should enable a user to use standard orbital elements to recreate real orbits as well as create their own new simulated orbits. It should focus specifically on stable orbits, allowing a user to project them into the far future and see what they are like. While this product is designed for my clients, it should be designed for any user with basic understanding of orbital mechanics to play around with creating simulations.

## Clients and Other End Users

I have two main clients:

* Alexis:
  + Alexis is my sister and likes to play many video games.
  + Some of those video games include Kerbal Space Program and Universe Sandbox.
  + She found that those games had some inaccuracies in how they ran.
  + They also were lacking in ability to properly define orbits using standard parameters.
* Ben:
  + Ben is a mathematician who has played many simulation games.
  + As he does a lot of high-level maths, he felt that simulation games often didn’t go for the more complex approaches to simulation despite them being more accurate.
    - He has derived non-compound approximative methods of simulating orbits which he also wanted testing out as an alternative to the standard methods.
  + As such, he wanted a more accurate orbit simulator, given his interest in rocketry and astrophysics.

Other potential end users include our friends, potentially physicists or astrophysics enthusiasts like me and Ben. Alexis has expressed a wish to send her simulations to her friends and colleagues, so any recipients of save data who wish to use it to view her simulations are also potential end users for this project.

## Interview

I interviewed both clients together with some very open-ended questions to gather more information about what kind of system they would want.

Questions:

1. What problems do you have with existing simulation software?
2. What features would you want an orbit simulator to have?
3. What kind of machine would you expect the simulator to run on?

1) Alexis said how a lot of simulation software was either too rigid or too free. Some software she said already did all the maths for the user but wouldn’t let her change the simulation in any way. She said these were often ‘real world’ simulators which were mostly designed to let a user explore a simulated version of reality. She complained about how the other kind of simulators were too far in the other direction, saying that they would rely too heavily on approximations and that would often cause a lot of glitches or ‘jank’. Ben added to this, saying how some software like Kerbal Space Program were somewhat infamous for being glitchy, although he did credit that partially to it being a sandbox game as well as a simulator.

Alexis also complained that in both simulation software she described, she’d never properly seen one allow for the standardised methods of defining orbits without ‘mods’ added to them. Ben agreed, saying that most simulators focused a lot more on the sandbox aspects as they were mostly video games first and scientific software second.

Ben went on to say that as he liked and often did a lot of high-level maths, he didn’t like their ‘wishy-washiness’. He said how simulations were too commonly inaccurate because of approximations and that, because of his own experiences with the maths, he knew the models behind the games could be a lot more well designed too not needlessly be inaccurate.

2) After discussing with them, they wrote a list with some reasons.

|  |  |
| --- | --- |
| Features | Reasons |
| The ability to add and remove orbiting bodies. | They want to be able to create objects which have orbits, not just orbits themselves. The idea is to simulate objects moving in space and not just ellipses. |
| The ability to define orbits using state vectors and Keplerian elements. | They originally said that objects should be definable by state vectors, but quickly remembered that Keplerian elements should also be allowed even if they are more intrinsic to orbits and not the bodies themselves. |
| The ability to view orbital statistics of bodies. | While they are the ones defining the objects, there is data that parameters don’t tell you without calculations such as orbital period which they want to be able to view. |
| The ability to traverse the simulation in at least 2d. | They wanted the orbits to be viewable as it would ‘be pretty cool’. They also said it would make the data a lot more meaningful if you could see it represented on the screen. |
| The ability to save and load simulations. | They want to be able to work and store simulations, and then open them up again later. Alexis wants also to be able to send them to her friends to run on their computers. |
| The ability to change the simulation speed. | They wanted to be able to run the simulation at various speeds, and to prioritise longer time scales. As they are objecting against compound approximations, longer time scales are important as that’s where the differences in the methods are much more prevalent. |

3) Alexis has quite a good computer which can run pretty much any game at high settings. She isn’t overly fussed about performance as such because her computer can run most things well. Ben’s computer is not as good, so he said it needs to be at least viewable running in real time without large amounts of lag between ticks. They agreed that the baseline is that it should be runnable on about a mediumly good pc. They also noted that adding lots of simulated bodies would slow down anyone’s computer, but if it could run a basic version of the solar system, it is probably fine.

## Limitations to Avoid and Acceptable Limitations

There are several simulation limitations that always must be made when creating a high-level physics simulation. With the help of the interview and other discussions with my clients, I have formalised limitations which this simulation software should specifically try and avoid:

* Compound approximations over time.
  + Any approximations made in the simulation should never impact later points in time.
  + While some approximations are allowed, they should mostly only affect how the data is represented in the GUI and not how the simulation functions.
* Using non-scientific rails.
  + Any body that’s trajectory is predefined (often referred to as being ‘on rails’) should only be on those rails if they are scientifically calculated.
  + They position at a given point in time should be equivalent to if it wasn’t predefined; the rails only should exist as a method of predetermining location instead of forcing location.
* Non-significant mass difference between a parent and child being handled as if the parent isn’t affected.
  + If a child body is particularly large, you cannot assume the centre of the parent is the centre of the parent-child system’s reference frame.
  + In other words, children should be at most a large moon of a parent and not something of overly similar size.

There are also some important limitations which, after conversing with my clients, have been decided to be acceptable:

* Orbiting bodies don’t have to be able to travel between hill spheres.
  + In other words, orbits don’t need to be able to leave their parent’s influence.
  + This is because Ben’s model doesn’t account for it, as well as the fact that it creates ambiguity if hill spheres ever overlap.
  + Furthermore, this is an orbit simulator, not strictly a space motion simulator.
    - Modelling bodies travel between parents, while useful, isn’t simulating orbits but generic trajectories instead.
* Objects don’t interact with any other object that their parent body.
  + Children will still move relative to their parents, but they won’t get pushed and pulled around by other bodies once they are created.
  + This was chosen because to solve this without compound approximations requires solving the n-body problem; this is a high-level astrodynamics problem which to this day hasn’t had an efficient solution.
* General relativity and other relating higher-level physics ideas do will not apply.
  + This simulation is made with Newtonian mechanics in mind and so will not have concepts suggested by relativity or any other more modern model implemented intentionally.
  + This is because the maths gets significantly harder and would likely require a degree to even begin working on.
* Movement doesn’t have to be continuous.
  + Some equations aren’t analytically solvable which means, unless an approximation is taken to calculate an answer at a given point, it is only possible to calculate certain values one way.
  + An example of this is distance at a given time; without difficult and time-consuming approximation it is not possible to calculate, but it is possible to calculate time at a given distance and use that to approximate distance at a given time.

## Comparisons of Potential Models

### Time Increment Model

The standard model used in most simulation software works around a few standard equations.

Where:

* is the acceleration experienced by a body
* is the mass of the body exacting a force
* is the distance between the two bodies
* is the gravitational constant
* is the change in distance between the objects caused by acceleration
* is the change in time that acceleration is being experienced over

This method can be implemented in a variety of ways to increase performance an accuracy, but the most basic principle works as such:

Diagram

Description automatically generated

Each double-headed arrow represents an acceleration experienced by a body towards another body, the magnitude of which are described by the first equation above. (In this diagram, nothing is to scale; this is purely for the purpose of explanation.)

Chart

Description automatically generated

These accelerations are resolved to provide an overall acceleration experienced at a given instance of time.

Chart, bubble chart

Description automatically generated

As shown in yellow, each body then moves a distance. This distance is as described by the direction and magnitude of the resolved accelerations multiplied by the time step squared, as described in the second equation above.

There are some benefits to this approach:

* Assuming performance isn’t an issue, you can simulate every body interacting with each other.
* It isn’t rigid; bodies can move however they will be simulated to instead of following a statically defined path.
* It is simple to understand.

There are however some glaring issues to this approach:

* Without accuracy-harming optimisation, the number of calculations needed to be ran per tick grows proportional to (number of bodies)!, as every body must interact with every other body.
  + To go with that issue, those equations look like with two bodies as their forces only act in straight lines to each other, allowing the vectors to be simplified down to scalars.
  + In reality, distance is the magnitude of the position vector, and acceleration is the second derivative of that vector.
* Time is continuous, but this approach assumes time moves in increments of .
  + This means that your position does not accurately change; there is a small inaccuracy between how far you move every tick and how far and in what direction you were supposed to move.
    - Over time this error builds up, making the simulation’s accuracy degrade.
    - There some methods of reducing this error, however none fix trying to speed up the simulation by having a greater interval and so trying to reach the far future would require either a very fast computer or a long time.
  + To fix this, you would need to integrate acceleration with respect to time.
  + Distance however is a function of time; in fact, acceleration is the second derivative of that function.
  + This means the method of solving this results in solving a second-order non-linear differential equation, which is very difficult.
    - Note that there are ways of solving some related equations to get distance as a function of time, one way of which is the derived method me and Ben created.
    - These methods are still technically approximative, but only ever because of reversing a non-analytic equation. This means their approximations are not compounded and so errors aren’t stacked as time passes.
* Orbits cannot be truly known nor defined.
  + Bodies will move in orbits if set up correctly, but that orbit cannot be predicted with guarantee because of the previously mentioned compounding errors.
  + This also means they cannot be defined through Keplerian elements; the best you could get is converting the Keplerian elements into state vectors and then approximating the rest of the orbit, which will eventually drift from the orbit defined by the Keplerian Elements.
  + Overall, this means that while objects moving in space can be roughly simulated, orbits themselves to some degree cannot.

For most space movement simulation software most of the highlighted issues aren’t overly major; time is probably on a human scale, accuracy can be lost due to not running over a long period of time and there probably not being a large number of bodies, orbits don’t need to be defined for objects to move; but for an orbit simulation software this is critical.

### Derived Model

There is a method of non-compound approximation orbit simulation that has been derived before. We did not use this method, but this is included to point out the existence of a potential scientific model. This is uses Kepler’s equation:

Where:

* is the mean anomaly
* is the eccentric anomaly
* is the orbital eccentricity

The eccentric anomaly can be used to calculate position around a body:

Where:

* and are as previously described
* is the semi-major axis
* is the semi-minor axis
* and are the x and y position around a body when reducing the axis from 3D to exist on the 2D plane the orbit exists on

Values such as the eccentricity, semi-major axis and semi-minor axis are all calculatable through standard orbital elements (state vectors and Keplerian elements) however this is not the case for the eccentric anomaly. Eccentric anomaly is calculated through Kepler’s equation, but as it is transcendental it means it cannot be analytically solved for; in other words, you can find it but with difficulty and non-compound approximation.

This does fit the limitations given by my clients, but there is one major problem; these equations do not determine position with respect to time, and so more derivation is required.

Me and Ben worked together to create some equations that link position and time. These equations are derived in an independent way to the derivation of Kepler’s equation:

Chart

Description automatically generated

As shown above, the velocity (shown by single headed arrows) can always be broken down into two parts; the velocity perpendicular to the position vector between the two bodies and the velocity normal to that.

Hence, as the normal velocity is defined to always be in the direction of the position vector, the distance between the two bodies is proportional to the normal velocity (specifically, it makes the integral of as shown below).

Chart

Description automatically generated

As shown by the diagram above, you can rearrange the velocity vectors to show that the total velocity is described by a right-angled triangle.

This means Pythagoras’s theorem applies, so .

Therefore:

Perpendicular velocity can be defined using angular momentum, a quantity conserved in an orbit making it useful to use:

This is rearranged to get:

Hence:

The squared magnitude of the velocity vector, , can be defined through conservation of energy:

Through replacing kinetic energy and gravitational potential energy with their standard equations you get the following:

This is rearranged to make the subject:

Hence if you substitute this into the previous equation for distance, you get:

Which is rearranged to get:

Or otherwise written as:

This is a first order differential equation with separable variables, and hence can be rearranged to:

Ben then solved this integral to get the final expression:

Where:

* is the gravitational constant
* is the mass of the body exacting a force
* is the specific total energy:
* is the distance between the two bodies
* A is the absolute value of the specific angular momentum:
* is the constant of integration
  + It denotes any offset the equation might be off based on the conditions of the equations use.

This equation does have a few boundaries that need to be set; in fact, 4 different variations are required depending on whether the object is moving away from or towards the periapsis and whether it is describing motion below or above the distance halfway between apoapsis and periapsis. (The plus or minus is caused by the latter, even though an offset also needs to be applied to account for the time taken to reach the midpoint).

Where:

* All previous constants are as previously defined.
* is the orbital period:
  + is the semimajor axis
    - It is not the acceleration as previously defined; both commonly use the letter a.
* is the ‘time offset’ caused by a body starting their orbit not at the periapsis
* is the ‘midpoint time’
  + This is used to correct the time outputted to handle the systematic error caused by not including the time taken travelling from periapsis up to the midpoint.
* The subscripts refer to where the equation is valid (the following are combined to define the equations properly).
  + Subscript of A refers to the equation being valid on approach towards the apoapsis.
  + Subscript of P refers to the equation being valid on approach towards the periapsis.
  + Subscript of L refers to the equation being valid in the lower portion of the orbit, such that it is under the midpoint, starting from the periapsis.
  + Subscript of H refers to the equation being valid in the higher portion of the orbit, such that it is further than the midpoint up the apoapsis.

This is equation, similarly to Kepler’s equation, isn’t analytically solvable for distance. This is however fine, as it still fits the limitations outlined by the clients.

This could be implemented in one of two ways:

* Using a non-compound approximation to solve for distance given a specific time.
  + Benefits:
    - This allows objects to always be updated to a new location every tick.
      * Movement hence would seem continuous to a user as ever frame the objects move.
  + Negatives:
    - The processing time to approximate the solution to that equation, given its unpleasant nature, would not be insignificant without skipping on accuracy.
    - All positions would have some inaccuracies as you could only approximate them.
* Using a pre-generated list of times created by a set of defined distances.
  + Benefits:
    - Performance will be better than the alternative, as bodies only must look up their next data point, and not calculate it as the simulation runs.
    - Data doesn’t have to be created in simulation time; instead, the orbits function on scientifically created rails.
      * This could allow for a user to export ephemerides of the entire orbit at once.
  + Negatives:
    - Depending on the time scale, size of an orbit, or number of pre-generated points, bodies may not move every tick. Hence, this means while their positions are accurate, they may not always be in those accurate positions at accurate times.
    - This takes more RAM space, depending on the number of stored nodes.

There are advantages and disadvantages to using these equations as a hole in comparison to the standard method:

* Advantages:
  + As stated previously, these equations do not rely on previous calculations to have been run, and so any approximations do not compound on each other.
    - An additional benefit to this is that you can tell the simulation to exist at a further point in time, and it can just throw that time into the method and return you a position for each body.
    - The alternative implementation requires the simulation to run continuously from the point you are starting at to the point specified to return those positions.
  + It is better for performance as it is specifically reducing from the n-body problem.
    - Only calculating its positions with respect to a parent means there is only one interaction needed to be calculated.
  + It allows for creation of orbits and not just trajectories.
    - When creating a body, you aren’t assuming the orbit will be produced over time.
    - Instead, you are directly creating an orbit which is followed.
      * This is obviously ideal for an orbit simulation software.
* Disadvantages:
  + Objects must only interact with a parent and no other body.
    - This isn’t surprising; the n-body problem is, as mentioned previously, not solved in any efficient way.
    - It does however reduce accuracy, especially in ‘crowded’ simulations.
  + There will be some inaccuracy in displaying a continuous simulation to the user.
    - As the equation linking position to time cannot be solved for position, either accuracy in time or position must be made.
    - As mentioned in the advantages, this isn’t overly a problem because these approximations are only in displaying to a user and not the physics itself, but it is still worth noting.

### Overall

The derived method clearly fits the limitations set out by the clients a lot better than the standard implementation. Using the equations provided as well as some key chosen limitations, a model has been defined (See ‘The Implemented Model’ for more information).

## Observations of an Existing System

The best current software that me and my clients are aware of is Kerbal Space Program. It is a rocket simulation game, allowing semi-accurate sandbox creation, flying, and crashing of rockets. The game can be purchased on steam or other platforms, but example screenshots are attached.

A picture containing text, electronics, loudspeaker

Description automatically generated

Viewing from the ‘tracking station’, an orbit-scale viewing window, you can see several pre-defined orbits of planets. These planets are ‘on rails’, meaning that their orbits are permanently fixed to travel along the paths shown.

These rails are not always based in scientific accuracy, as mentioned under ‘Natural Satellites’ of <https://wiki.kerbalspaceprogram.com/wiki/Jool>, which explains how the moons of the game’s gas giant Jool would in reality be incredibly unstable if not placed on rails as they are in game.

Graphical user interface

Description automatically generated

The game does allow for creation of orbits using the cheats menu; this is quite useful if you want to recreate an orbit using Keplerian Elements. Obviously, as the game is a rocket simulation game first and foremost, it makes sense this is in the cheats menu in KSP; this should be given straight as a feature user in this project’s implementation.

Unfortunately, there is no equivalent for state vectors. Furthermore, you are limited to moving spacecraft you created instead of being able to create bodies directly in space. You also cannot create any planets of your own through this menu.

Graphical user interface

Description automatically generated Graphical user interface

Description automatically generated

Sensibly, you can only time warp using the time-increment model up to 4x speed to not reduce accuracy. You can also place yourself on scientifically defined rails; these are made using orbital elements in a similar way to the derived method discussed in ‘Comparisons of Potential Models’.

Highlighted in this video, <https://www.youtube.com/watch?v=YcsE1LMWNIs>, the time-increment model can have certain issues if pushed. In that example, they abuse the fact a log graph is turned linear via the model to reach past the speed of light.

Overall, it has some good and bad elements to it’s use for simulating orbits:

* Good:
  + Allows creation of orbits with Keplerian elements.
  + Nice looking user interface.
  + It allows the user to speed up or slow down the simulation.
    - It also allows it to be paused by accessing the pause menu.
  + It combines both models nicely for a clean user experience of flying rockets.
* Bad:
  + Only small spacecraft can have their orbits created.
    - It is intended in the game for you to fly the rockets, so this design choice is understandable.
  + There are some errors in both models.
    - This is inevitable, but still a flaw.
    - Some errors are however intentionally overlooked, such as their Jool system’s design.
  + It lacks creation of orbits with state vectors.
  + You are unable to view orbital statistics of bodies.

Using these as pointers, I think some areas demand more focus in this project.

* Having clear and accessible orbit creation windows.
  + They should allow both state vectors and Keplerian elements to be used.
* The simulation should be able to change speed.
* Bodies of any reasonable size should be creatable, not just small spacecraft.
  + This should be in reason however, as shown by their Jool system being made of large bodies and having flaws from their lack of interactions.
* You should be able to view a body’s orbital statistics.

Note that while I credited a combined model as being quite good, I do not think it is worth pursuing for this simulation software as the physical scale is a lot larger than small rocket flying.

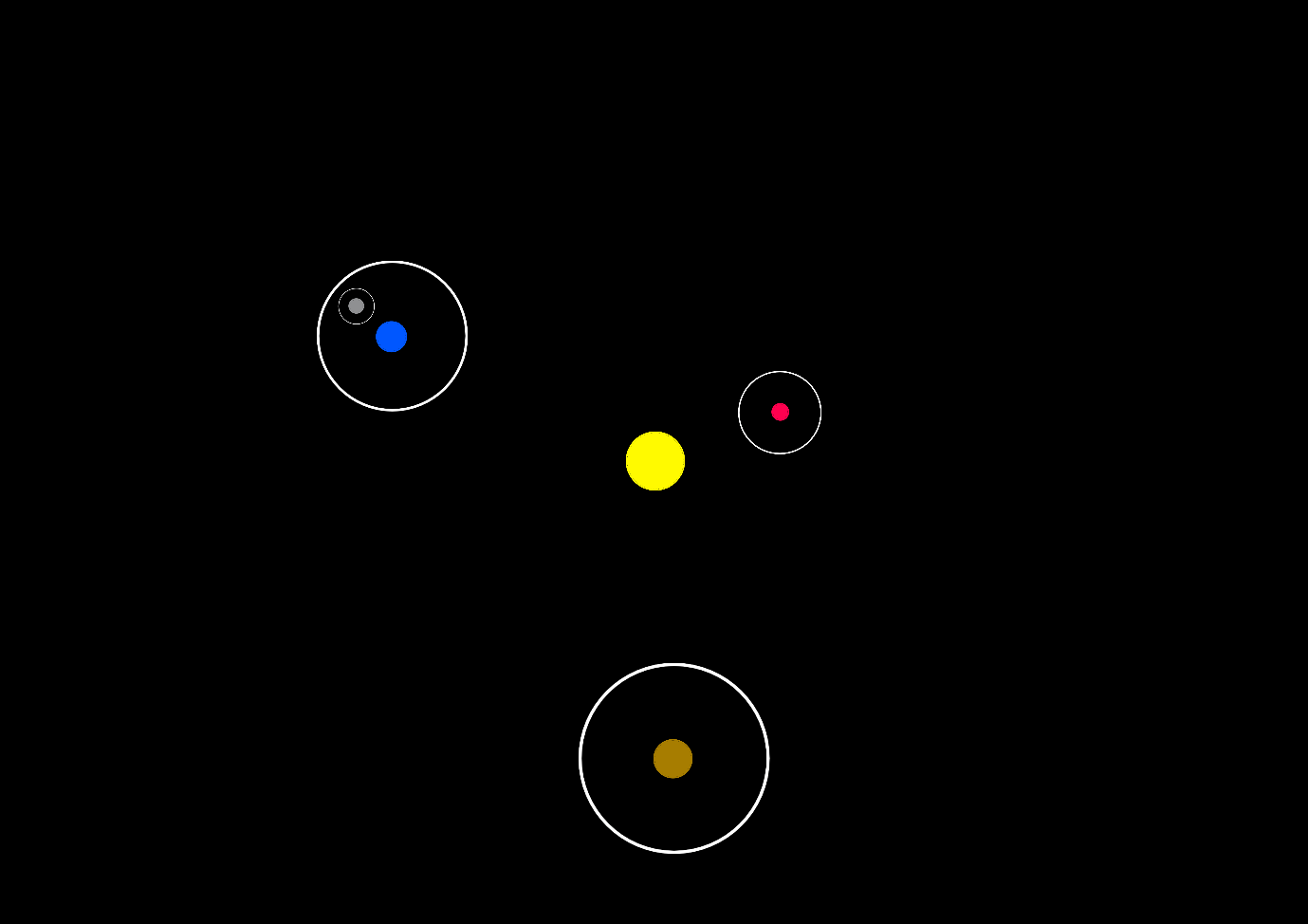
## Objectives

These objectives have been set based on talking with the clients as well as my own research. These bullet points have been confirmed with the clients as to being what they want from the project.

1. The system must calculate the motion of bodies in 3D space.
   1. The system should use a thoroughly designed physics model such that no approximations are compounded over time.
      1. The mathematical basis behind this model will use equations derived from both me and Ben, as per Ben’s request.
   2. The motion should be relative to a parent body.
      1. A simplified system will be used such that a body only relies on a single parent due to the complexities of the n body problem.
      2. Grandchild bodies (etc) are allowed but will not directly affect anything but their direct children.
2. The user should be able to create, remove, and read orbital data about orbiting bodies.
   1. An orbiting body must be able to be created through either state vectors or Keplerian elements.
      1. Orbits should only exist within a particular parent body’s dominant area. This should be implemented using the concept of a hill sphere.
         1. If two hill spheres overlap such that they both include each other’s centres, disallow it as such an orbital system cannot be accurately modelled without compound approximations, which this system does not implement.
      2. If a child body is significantly high in mass in comparison to its parent, warn the user.
   2. The user should be able to select a body and read calculated information at any time.
      1. The shown information should not interfere with the user’s ability to view and modify the simulation.
3. The user should be able to control the speed of the simulation.
   1. Any difference in accuracy or speed should not affect future information about orbiting bodies, but simply what is displayed to a user at any given time.
4. The user should be able to traverse the simulation.
   1. They should be able to view the simulation in real time at any position within a given bound.
   2. They should be able to change that position while the simulation runs.
5. The user should be able to save and load a given instance of the simulation.
   1. All necessary information about a scenario should be stored.
   2. These saves should be able to be used to recreate the instance that a simulation was in.
   3. Hence, the user should also be able to load the save.
6. A standard basic solar system should be given to the user.
   1. Only the main planets are required.

## The Implemented Model

### Explanation of the Scientific Model



The model decided upon combines the derived equations with a tree structure of bodies. As shown above, bodies will orbit a central fixed body, and will have an area that exist around them which are their zone of control. (These zones are outlined in white, but in the final UI will not be shown explicitly).

These zones will be defined using the concept of a hill sphere, which is defined as the region where a body dominates attraction of satellites. The radius of such a sphere is defined through this equation:

While this is approximate, hill spheres in reality are not as clean cut as a hard boundary, and so this equation is for all intents and purposes equal and not approximate.

The use of hill spheres allows for the orbits to be treated as nodes in a tree. The central fixed body is equivalent to the root node, and each child can without more children of their own. (In actual implementation, there would be a bodyless root node that the reference body is a child of to allow removal and creation of the reference body).

An example of this structure could be the solar system. The sun, Sol, acts as the central node with the planets all as its children. Their moons are their children, and so would be child nodes of the planets.

To try and reduce inaccurate simulation, when new bodies are created their hill sphere should not overlap with any pre-existing body.

### Prototype

I prototyped this model in 2D to make the maths simpler.

Diagram

Description automatically generated

The pink and red line traces the orbit as it swept across.

A picture containing diagram

Description automatically generated

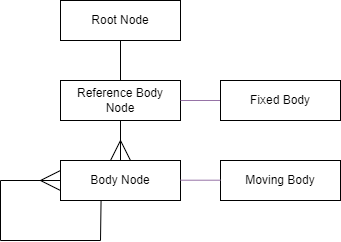
An example of the same orbit but fully traced.

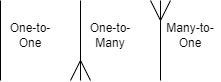
There is a lot of extra data on this prototype; this was used to debug the orbital motion. There were a lot of errors created when attempting to implement the maths and methods into c#, especially because the original derivation of the equations used was not as smooth as written out in the documentation. This meant I had to wrestle a lot with the physics to get a working 2D model before moving onto 3D.

Luckily, the transformation from 2D to 3D was reasonably simple; as orbits exist in 2D planes, all you need to do is create the orbit in 2D and then apply rotation matrices to the position and velocity vectors to transform it into the correct 3D orientation.

### Entity-Relationship Model

The general structure of the tree is as follows:





(The purple connections mean that the relationship must exist for either to exist in the tree)

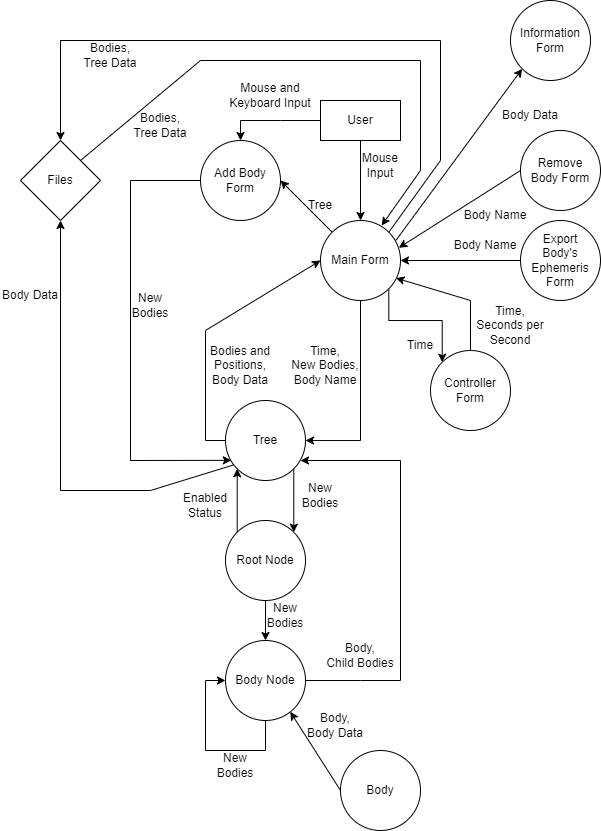
The root node contains no assigned body, as it exists as an anchor for the simulation to sit on top of instead of to stem from. Travelling down to the refence body node, you can see that it does in fact have a body anchored to it; this is a fixed body, which means it cannot have an orbit. The tree stems from it, shown by it being able to have many children. Those children are body nodes, which all must have a moving body attached to it. Each of these nodes can also have any number of orbiting children, hence having a one-to-many relationship with itself.

# Documented Design

## Data Flow Diagram

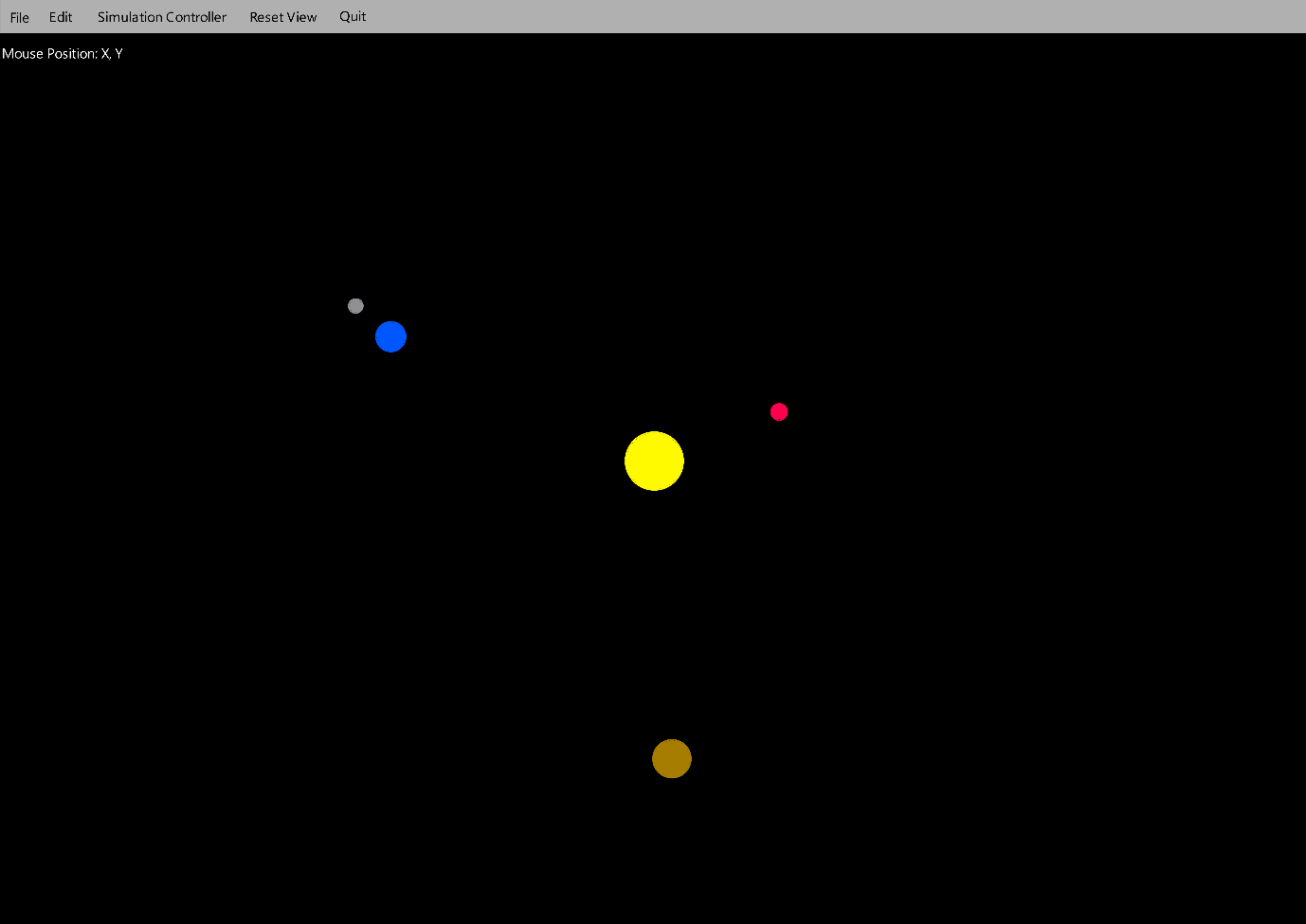
The data in the simulation is frequently passed between forms and data structures, so I have made a diagram which shows all the major data transfers between major objects.

Note that most forms are not connected to the user; in reality they are via keyboard and mouse input, but there was not enough room on the diagram to fit every connection. I prioritised Add Body Form and Main Form as their interactions are the most important/largest.



## User Interface Design

### Main Form:



This form most has the functionality of displaying the simulation to the user. They will be able to drag and zoom to change the view of the simulation (it will be a 2D projection of the 3D simulation) as well as right click on bodies to find orbital statistics.

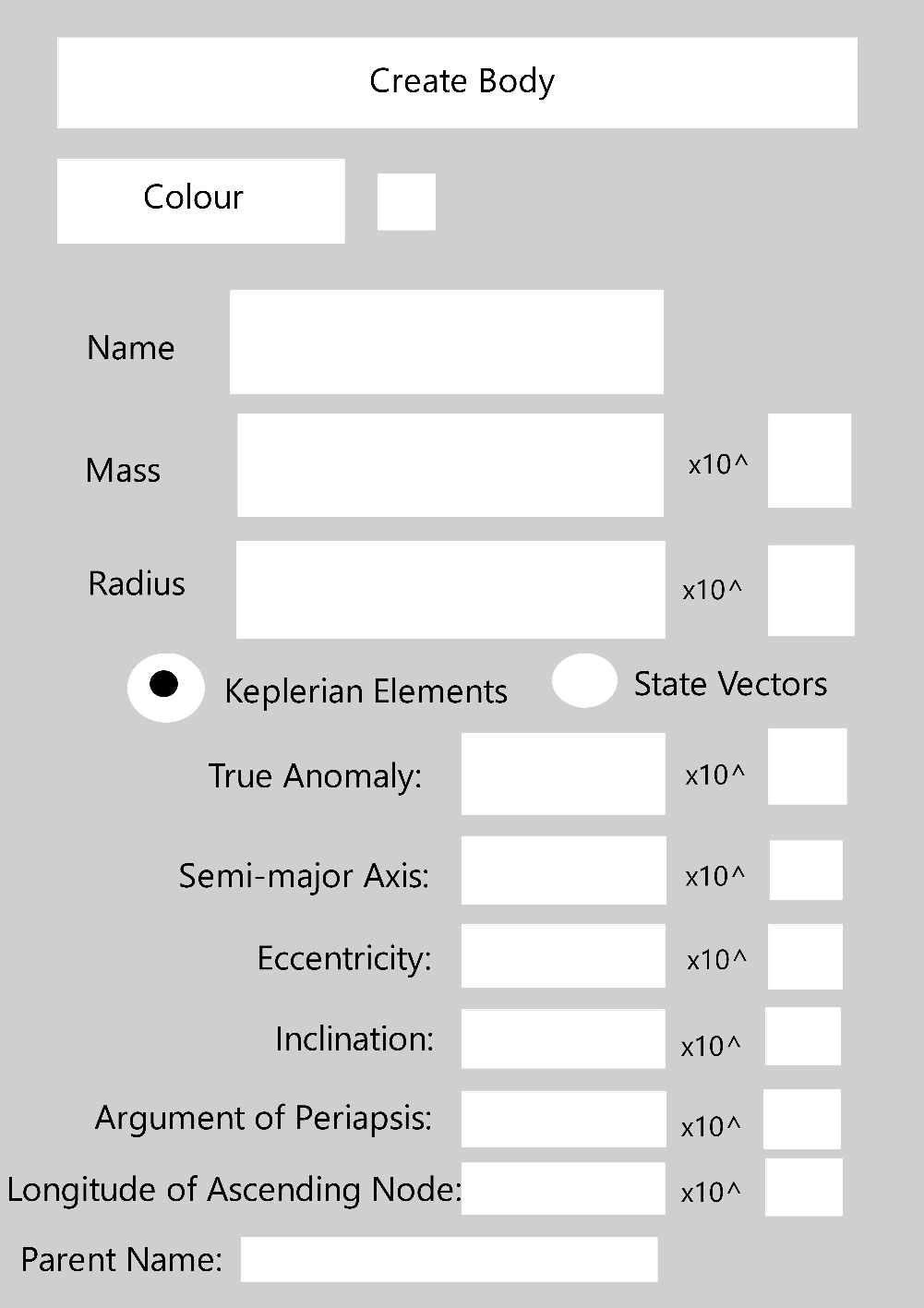
The bar at the top would have the options as follows:

* File
  + Save As
    - This allows a user to save their simulation.
  + Load
    - This allows a user to load a simulation from a save file.
  + Export A Body’s Ephemeris
    - This opens the Export Body’s Ephemeris Form.
* Edit
  + Add Body
    - This opens the Add Body Form.
  + Remove Body
    - This opens the Remove Body Form.
* Simulation Controller
  + This opens the Controller Form.
* Reset View
  + This resets the viewing parameters to the set standard.
* Quit
  + This closes the program.

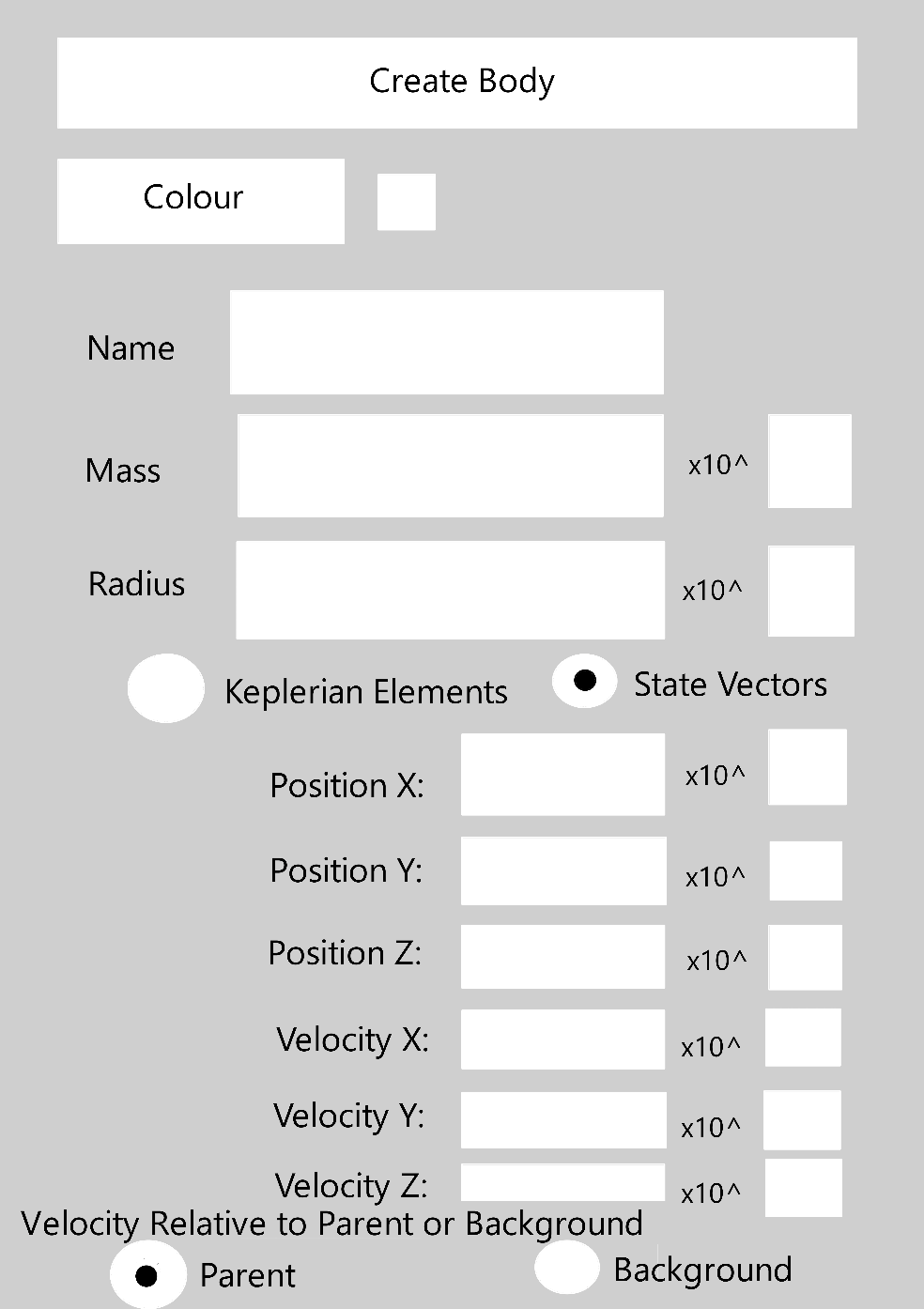
### Add Body Form:

There are 3 different ways this form can look.

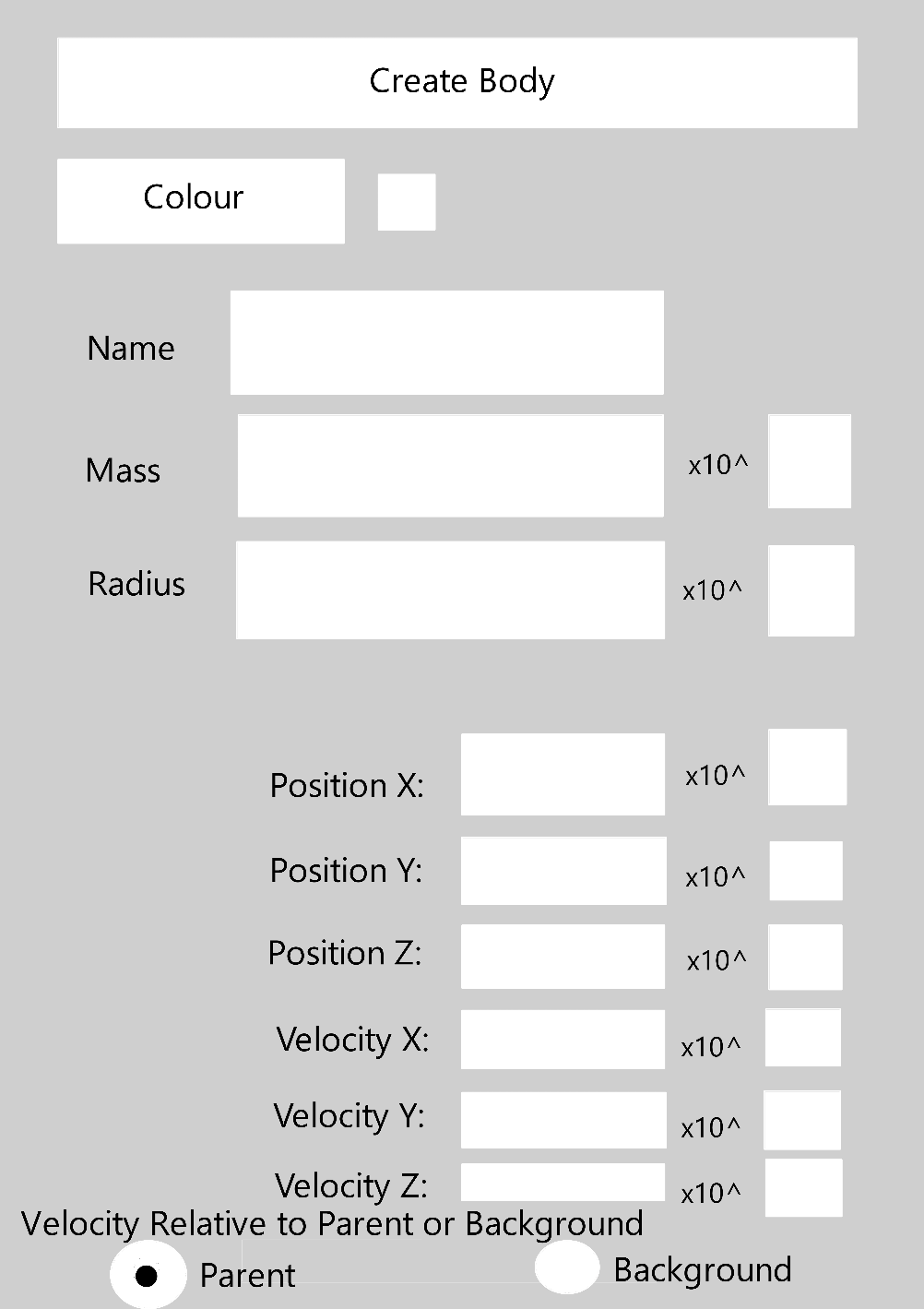
Adding a moving body with Keplerian elements:



Adding a moving body with state vectors:



Adding a fixed body:



These forms are designed to allow a user to input data to create bodies.

To get around the fact that a user needs to input large numbers, two textboxes are used to represent standard form.

The box next to the colour button is designed to change colour to the chosen colour to remind the user what colour they picked.

The first and second variations should be swapped between depending on whether Keplerian elements or state vectors is picked.

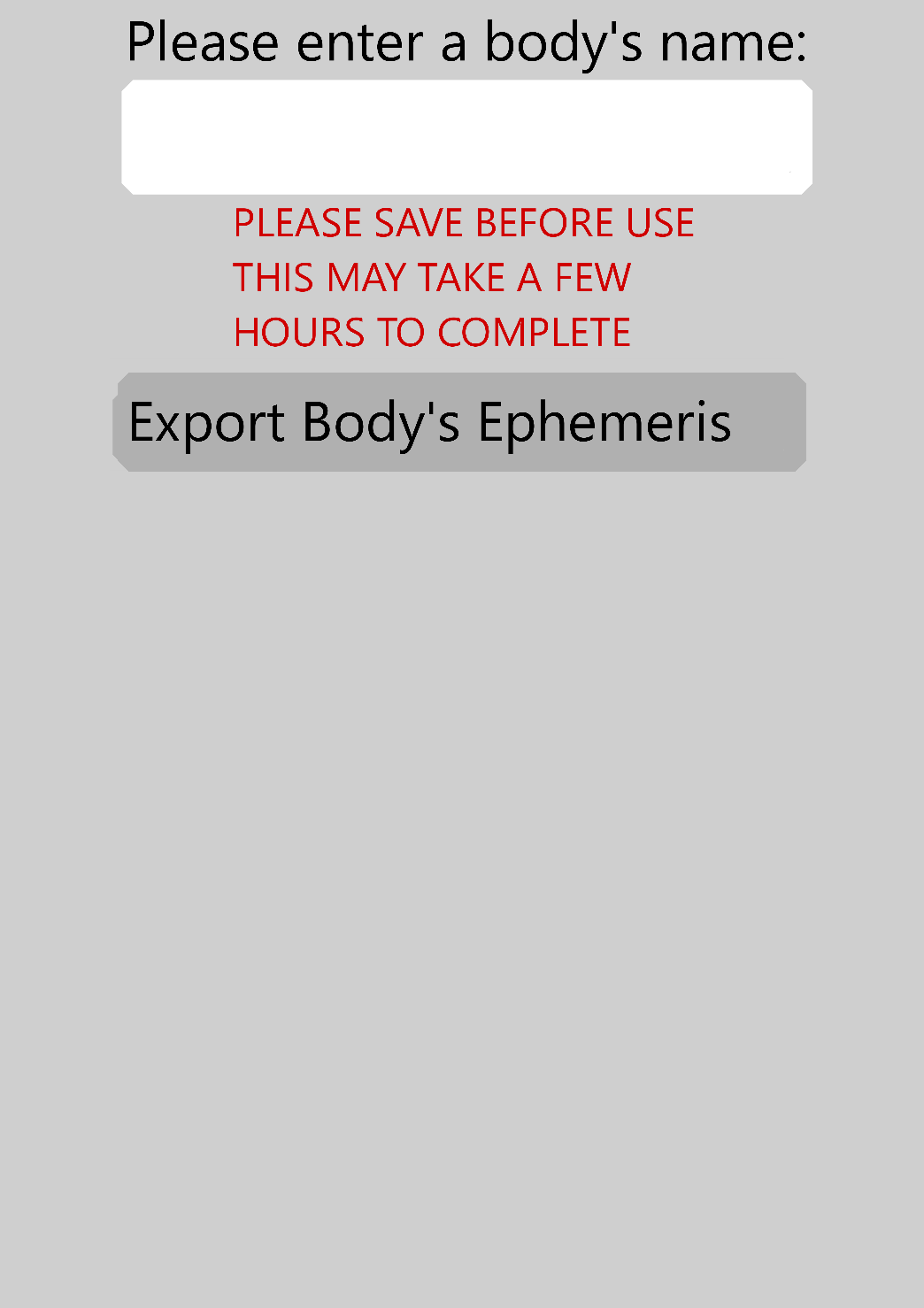
The third variation should only exist when no other bodies exist as it is used to create a new reference body.

### Remove Body Form:



This is a very simple form which takes in a body’s name for the purposes of removing the body with that name from the tree.

### Export Body’s Ephemeris Form:



This is a reasonably simple form which takes in a name for the purposes of allowing a user to save that body’s ephemeris of its orbit. There is a warning however as this takes a long time; there are 200000 data points attributed to each body and all of it has to be converted into text.

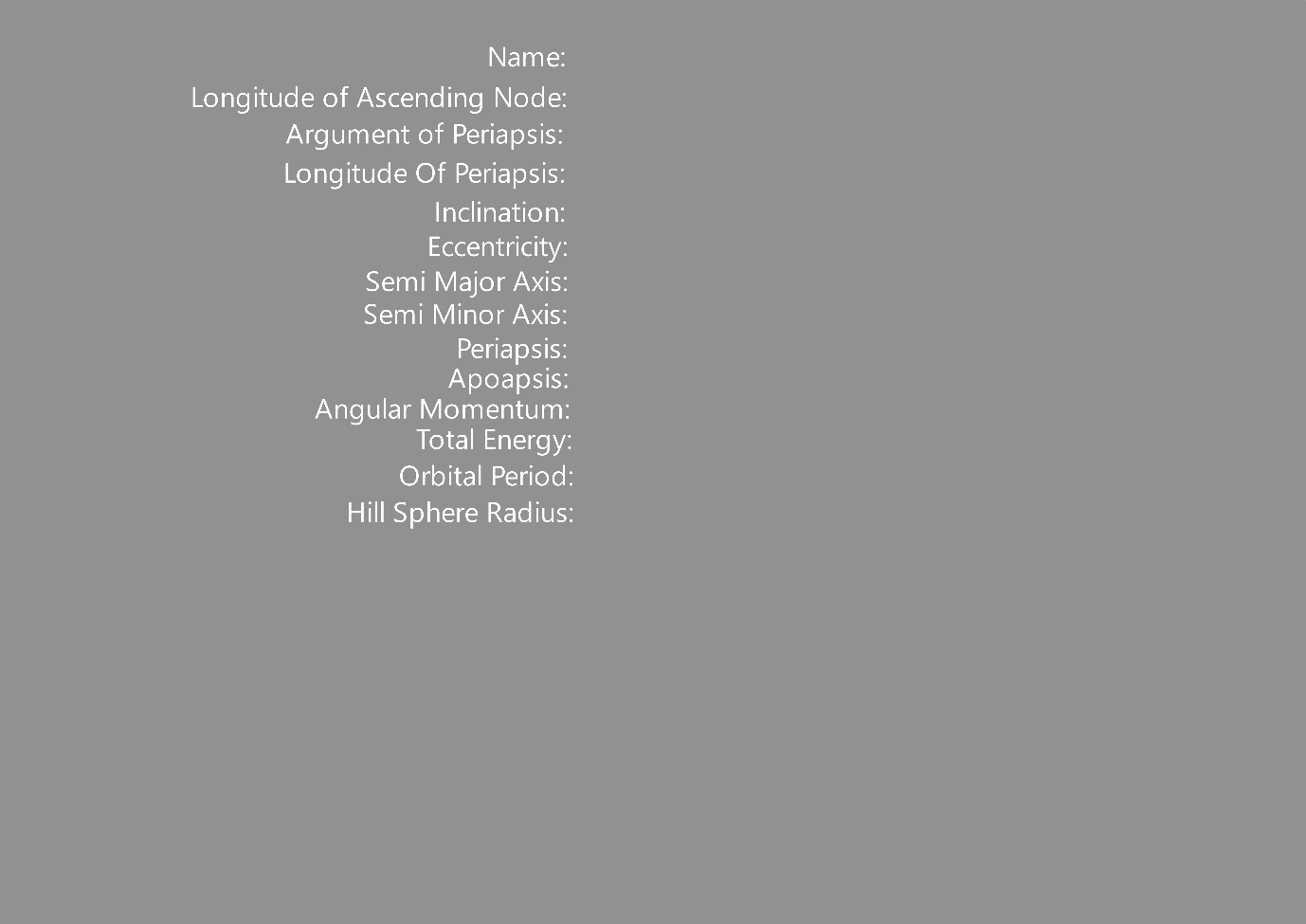
### Controller Form:



This form is responsible for controlling the speed and current time of the simulation.

The clock speed is a slider, but as sliders aren’t continuous, I will make each value of the slider correspond to a predefined speed.

### Information Form:



This form is designed to tell you orbital statistics of a right-clicked body.

There isn’t much to it apart from a lot of labels which display the information.

## Class Structure

I used a piece of free software called Software Ideas Modeller (available at <https://www.softwareideas.net/en/download>) to generate a class relationship diagram. It had a view bugs in displaying so I had to manually format parts of it.

A picture containing qr code

Description automatically generated

A picture containing qr code

Description automatically generated

(This has been split in 2 for readability purposes. The top image is the left, and the bottom image is the right.)

Some features of the design and of the above diagram are of note:

* All forms appear twice.
  + This is because of the nature of WinForms splitting forms into files containing code and files containing the form’s design.
  + Hence, when looking at the diagram, remember that just because a relationship isn’t shown on one of the two form appearances, it may still exist on the other.
* Some links that could exist are left blank, notably body.
  + Any object can instantiate a body, despite them supposed to only being permanently stored under the ownership of a node.
  + This is for good reason however as it allows bodies to be passed into the tree as well as created in the tree and not in the node.
  + I could in theory place the constructor for a body exclusively in the node classes, but I feel like they also have merit to a future coder who may want to define bodies outside of the tree structure, or potentially overhaul the model to remove the tree structure altogether.
  + Either way, the bodies can exist outside of the rigid tree structure so I still felt they should be allowed to be defined outside of the rigidity of the tree structure.
* Tree is struct because I want to store it on the stack as it needs to be accessed quickly and isn’t often written too.
  + This means most of the time MainForm is passed around instead of the tree itself as MainForm is a class, hence a reference type and so no data is copied in passing it as a variable.
  + If I were to pass tree, it would only pass a copy of the tree as it is a value type.
    - This is also problematic when I need to call to get live data from the tree or to modify the tree; potentially it could end up only modifying a copy.
    - This is unlikely to happen however as the data in the tree is mostly classes, which are reference types.
* There may be some errors that I have missed that formed with the auto generation of the diagram.

## System Flowcharts

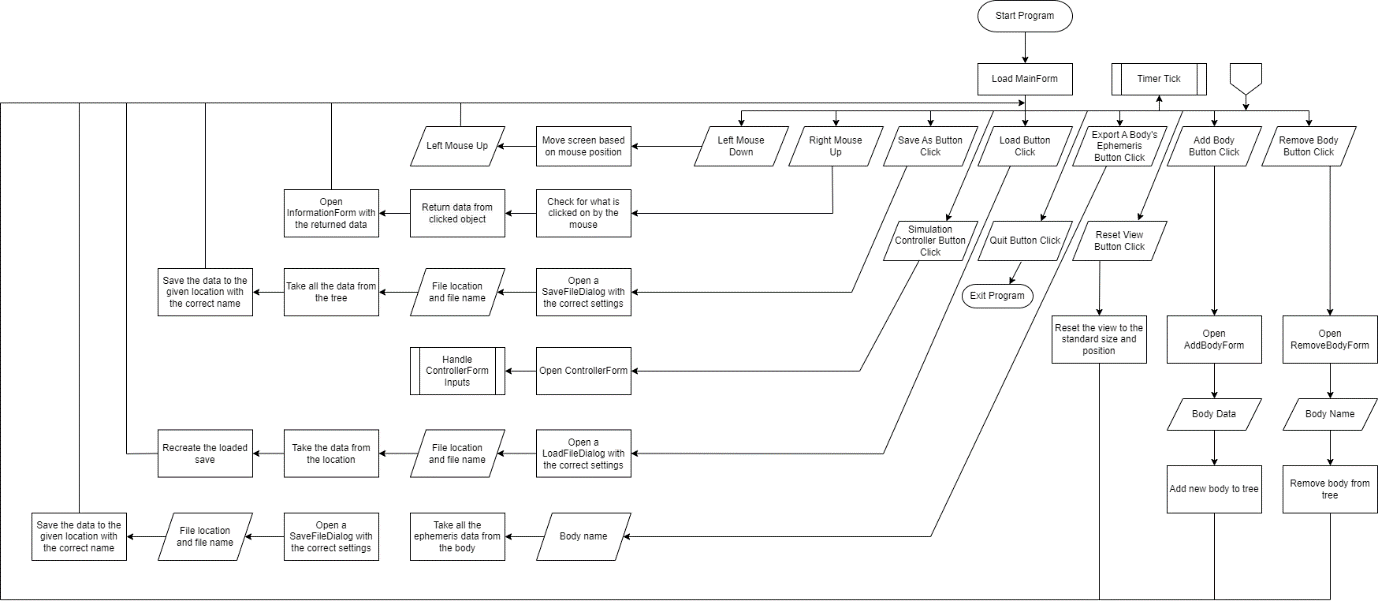
### Main Flowchart:

This flowchart shows how the program functions, with a mostly user-oriented focus.

Intentionally, it doesn’t talk about error handling, as it is implied to return to return through the off-page reference.

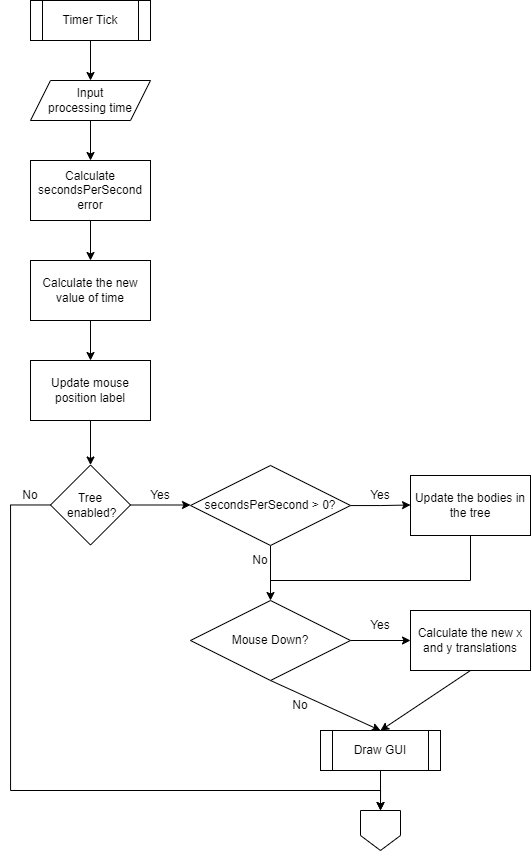
Any branch involving the timer has been abstracted, as they are best described separately given their continuous ticking doesn’t line up with the rest of the input waiting system. To be consistent, they return through the off-page reference.

The forms have their input cycle running asynchronously to each other, and they don’t close once used once in the final project. I found this hard to represent with a flowchart and decided I would note it here instead. This does mean this flow chart is inaccurate, however it does still show the overall processes each branch can and likely will take, which it was designed mostly around doing.



### Timer Tick Flowchart:

This flowchart shows what happens when the timer ticks; it technically connects with the previous flowchart however in reality they run asynchronously.



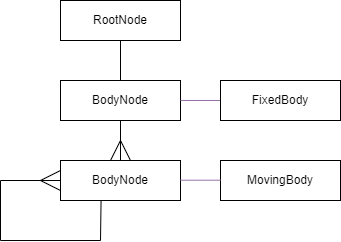
### Notes About Other Flowcharts

I believe the currently provided flowcharts give a good overview of how the code functions, and so I don’t believe making flowcharts about the currently omitted functions would give any benefit. Some will be highlighted in the following section however; see ‘Algorithms’.

## Data Structures

### BodyTree

The body tree functions as described in ‘Analysis, The Implemented Model, Entity-Relationship Model’. A version of the diagram has been attached here, with the names of the nodes and bodies being replaced with the data types that are used. The one part of the structure not previously mentioned was the fact that the one-to-many relationship is implemented using a list.



Diagram

Description automatically generated

(The purple connections mean that the relationship must exist for either to exist in the tree)

## Algorithms

|  |  |  |
| --- | --- | --- |
| Name | Where to Find | Explanation |
| Find and Populate Position | Tree.cs  The function AddToTree which the comments specifying state vectors | This is a recursive function used to traverse the tree. The reference body is originally passed in, along with the relative velocity and the current position relative to the reference body.  The function first creates a list:  This list is defined using a lambda function.  It sequentially takes every child of the passed in node and checks whether the distance between the new body and it is less than the child’s hill sphere radius.  If it is less, then according to the scientific model being used, that means the new body must exist in the child’s hands, or in one of their children’s hands, or there’s (and it goes on).  Hence, this child is added to the list.  Once all children are checked, there are 3 cases that could occur:   * There are 0 children in the list.   + This means the parent to the new node is the current node being evaluated, and so a new node can be created under the parent and be assigned the new body. The function can return up the chain and finish. * There is 1 child in the list.   + This child is passed back into the function, along with the position relative to the child.   + The velocity may also be changed to be relative if it was defined as to the static background reference frame.   + This is to check whether the new body is a child of this child, or maybe a grandchild or further. * There is more than 1 child.   + The children are ranked in terms of mass of the child / distance to the child^2, as gravity is proportional to M/r^2.   + Whichever has the highest score is then picked, and then treated as if there was only one item in the list. |
| Breadth First Traversal | Tree.cs | This function is a standard implementation of a breadth first traversal algorithm, but a function is passed in to allow evaluations of the children as they are traversed. This could be used for example to turn this from a breadth first traversal into a breadth first search. The function must return a Boolean to signify whether it wants to return from the BFT early.  The breadth first traversal algorithm in this situation is defined as follows in pseudocode:  QUEUE queue <- NEW QUEUE  LIST visitedNodes <- NEW LIST  DEF VOID bfs(startNode)  queue.ENQUEUE(startNode)  visited.ADD(startNode)  WHILE NOT queue.ISEMPTY()  node = queue.DEQUEUE()  IF (function(node)) THEN  RETURN  END IF  FOR EVERY childNode OF node  IF (childNode NOT IN visited)  queue.enqueue(childNode)  visited.add(childNode)  END IF  END FOR  END DEF |
| Depth First Search | Tree.cs  RemoveFromTree | A depth first search is used to delete bodies from the tree; it was chosen instead of breadth first search as it was judged unlikely for a user to wish to delete a full branch of the tree instead of just a single object. Single objects are located only at the bottom of the tree, and so depth first traversal makes sure to try and reach the bottom quickly.  The depth first search algorithm is defined as follows in pseudocode:  LIST visitedNodes <- NEW LIST  STRING removedBodyName  DEF BOOLEAN dfs(NODE parentNode)  visitedNodes.Add(parentNode)  FOR EVERY childNode OF parentNode  IF (childNode.assignedBody.Name = removedBodyName)  parentNode.childNodes.REMOVE(child)  RETURN True  END IF  IF (childNode NOT IN visited)  IF (dfs(child))  RETURN True  END IF  END IF  RETURN FALSE  END DEF |
| Load from Save File | Tree.cs  LoadTreeFromSaveableFormat | This function reads a save file and recreates the save described within it. When reading from a save file the format has to be decompiled back into useable quantities. They are stored in a modified csv style format, where instead of just commas separating the values its ‘,|’ instead.  The order of saved information is as follows:   * Heritage * Name * Mass * Radius * Colour * StartingTimeFromEpoch * StartingTrueAnomaly * orbitInformation   After reversing the saving back into data, orbits are reinitialised from the stored variables.  They are placed with the correct heritage by using a version of FindAndPopulatePosition. |
| Get bodies in a saveable format | Tree.cs  GetBodiesInSaveableFormat | This function takes a simulation and turns it into a save file. It does a depth first search to reach each member of the tree, keeping track of the heritage by adding and removing from a heritage list between each jump. |
| Register right click | Tree.cs  RegisterRightClick | This function takes the xy coordinates of the mouse and checks whether they intersect with a flattened version of the 3D orbit simulation being ran. It is based of the FindAndPopulatePosition method, but returns a body’s orbital statistics instead. |
| Return Ephemeris | Moving Body.cs  ReturnEphemeris | This function turns the 200000 data points of predetermined points and converts them into a csv.  The columns are as follows:   * Time * Angle of velocity to tangent * True Anomaly * Relative Position to Parent * Relative Velocity to Parent |
| Initialise Orbit Constants From State Vectors | Moving Body.cs  The function is a joined together version of the name on the left. | This function takes in state vectors and some characteristics of the parent to the new body and defines all orbital characteristics.  It first calculates a few orbital constants. One enough are defined to calculate the hill sphere radius, it proceeds to do that and then query whether any other objects exist in it’s hill sphere. If they do, the body will not be allowed to exist, as the simulation would be of very poor accuracy if allowed.  The periapsis is then checked to be greater than it’s parent’s radius, such that it means the new body doesn’t crash.  The hill sphere radius and apoapsis are summed and compared to the parent’s hill sphere radius; if it extends outside of the parent’s hill sphere radius the user is warned that the simulation has a small chance of not correctly inputting new bodies into the simulation due to the new body’s hill sphere extending outside of its comfortable area.  Finally, it is checked that the apoapsis also is within the bounds of the parent’s hill sphere as to make sure the orbit stays within their parent body’s reach. (As talked about in limitations, objects are not modelled to leave or enter parent body’s hill spheres).  More constants get defined. StartingPerpendicularAngularVelocity is derived through rotation the starting location in space using rotation matrices at to flatten the point onto the xy plane. Using this reduction, the angle the velocity makes to the tangent is now measurable and it then used to calculate the StartingPerpendicularAngularVelocity.  Finally, a few more constants are determined and the CreatePoints function is ran to create the orbit’s data points. |
| Initialise Orbit Constants From Keplerian Elements | Moving Body.cs  The function is a joined together version of the name on the left. | This function takes in Keplerian elements and some characteristics of the parent to the new body and defines all orbital characteristics.  This functions in a very similar way to with state vectors, so the full details won’t be reiterated. The one notable part of this method in comparison is the use of defining the orbit not by where the body is currently but where it should be at periapsis. That is then used to calculate a lot of the constants and to work backwards to calculating the state vectors of the body’s initial position. |
| Create Points | Moving Body.cs  CreatePoints | This function creates the 200000 data points that define an orbit in this simulation.  It starts off by defining some constants used in later maths. It then takes in the starting distance from the parent body of the new body to calculate the startPositionTimeOffset, which is an offset to the time equations to counteract not starting at the periapsis.  Then a for loop starts which runs 200000 times:   * First, a true anomaly is chosen.   + This method equally cuts the orbit into sections by using increments of angle (true anomaly). * A few variables are set. * Vectors like velocity and position have to be rotated using matrix transformations onto their orbital trajectory. * The time for that given point is then calculated to go with those calculated values. * Together they are all added to the predetermined points.   One they have finished, they are all ordered as it allows calculating the location of the body to be faster. |
| Update Current Point | Moving Body.cs  UpdateCurrentPoint | This calculates the closest point in time to current that is stored in the predetermined array.  The basic principle this runs on is that as time progresses you linearly travel the list of predetermined points. This means every tick it can progress forwards along the predetermined points until it reaches the point with the largest time smaller than the current time; this is usually the closest point to the true location at that given time located in the predetermined points array.  Every time the time passes the orbital period, a counter, timeWraps, counts how many times larger time currently is to the orbital period. This is used to make sure that it can update, as the actual benefit from updating timeWraps is that it simultaneously sets the current index back to 0. This means there won’t be an overflow error from the counter constantly going up outside of the predetermined points size, nor will the object get stuck at the end of its orbit because it fully traversed it’s array. |
| Update Current Points Using Array Search | Moving Body.cs  The function is a joined together version of the name on the left. | This calculates the closest point in time to current that is stored in the predetermined array.  This functions slightly differently to the other method of calculating the current point. This searches the full list for the closest point, which while is guaranteed to work it isn’t very efficient.  This method is more used to fix the other method in situations where it would fair, e.g., in changing the current time value manually. |

# Technical Solution

## Foreword

I used pascalcase typing to denote constants (variables not changed after a set of initialisation instructions) and camelcase to denote variables. Functions are also pascalcased. The main exception to this is pre-generated functions by WinForms, which default camelcase due to the form objects also defaulting their naming scheme as such, as well as said function parameters not being renamed according to constant and variable rules due to their pre-generated nature.

Unfortunately the formatting hasn’t been carried over to word.

## MainForm.cs

using System.Diagnostics;

using System.Numerics;

using System.Windows.Forms;

namespace \_3D\_Orbital\_Motion\_Simulation

{

public partial class MainForm : Form

{

public MainForm()

{

InitializeComponent();

tree = new BodyTree(new FixedBody(Color.Gold, "Sol", new Vector3(0, 0, 0), 1.989 \* Math.Pow(10, 30), 696340000));

time = 0;

resetViewToolStripMenuItem\_Click(new object(), new EventArgs());

}

// This is the tree structure that the simulation is built upon.

internal BodyTree tree { get; init; }

// Time is measured in seconds as the standard SI unit.

public decimal time { get; set; }

// This is the amount of time that occurs in the simulation every time a second occurs in real life.

public decimal secondsPerSecond { get; set; } = 1;

// This refers to how many metres in the simulation 1 pixel represents.

private float inverseScaleFactor { get; set; }

// The x and y translations are done with respect to the simulation's xy plane and not the viewing window. This is because it then stays consistent with scaling.

private float xTranslation { get; set; }

private float yTranslation { get; set; }

private InformationForm informationForm { get; set; }

private ExportBodysEphemerisForm exportBodysEphemerisForm { get; set; }

private RemoveBodyForm removeBodyForm { get; set; }

private AddBodyForm addBodyForm { get; set; }

private ControllerForm controllerForm { get; set; }

// A default reduced solar system is created on loading. It contains the sun, all major planets and the moon.

private void MainForm\_Load(object sender, EventArgs e)

{

double AU = 1.49597870700 \* Math.Pow(10, 11);

tree.AddToTree("Sol", Color.DarkGray, "Mercury", 3.285 \* Math.Pow(10, 23), 0, 0, 0.3871 \* AU, 0.20564, 0.122278, 0.843692, 0.50824, 2439700);

tree.AddToTree("Sol", Color.Salmon, "Venus", 4.867 \* Math.Pow(10, 24), 0, 0, 0.7233 \* AU, 0.00676, 0.059306, 1.338144, 0.961676, 6051800);

tree.AddToTree("Sol", Color.Blue, "Earth", 5.972 \* Math.Pow(10, 24), 0, 0, AU, 0.0167086, 0, 3.0525809, 5.0282936, 6378100);

tree.AddToTree("Earth", Color.Gray, "Luna", 7.347 \* Math.Pow(10, 22), 0, 0, 0.3844 \* Math.Pow(10, 9), 0.0549, 0.08979719, 0, 0, 1737400);

tree.AddToTree("Sol", Color.OrangeRed, "Mars", 6.39 \* Math.Pow(10, 23), 0, 0, 1.5237 \* AU, 0.09337, 0.032323, 0.867603, 4.998099, 3389500);

tree.AddToTree("Sol", Color.SandyBrown, "Jupiter", 1.898 \* Math.Pow(10, 27), 0, 0, 5.2025 \* AU, 0.04854, 0.022672, 1.750391, -1.50133, 69911000);

tree.AddToTree("Sol", Color.Beige, "Saturn", 5.683 \* Math.Pow(10, 26), 0, 0, 9.5415 \* AU, 0.05551, 0.043529, 1.983392, -0.36268, 58232000);

tree.AddToTree("Sol", Color.Azure, "Uranus", 8.681 \* Math.Pow(10, 25), 0, 0, 19.188 \* AU, 0.04686, 0.013491, 1.290846, 1.718626, 25362000);

tree.AddToTree("Sol", Color.Navy, "Neptune", 1.024 \* Math.Pow(10, 26), 0, 0, 30.07 \* AU, 0.00895, 0.030892, 2.300169, -1.48545, 24622000);

}

// Timers in winforms do not run asynchronously to the tick event function.

// This means that the time between intervals is as predefined, plus the amount of time it actually takes to run the code.

// To counteract this, the time taken to run the code is measured and used to correct the secondsPerSecond to make it still for the most part accurate.

Stopwatch watch = Stopwatch.StartNew();

private void simulationTimer\_Tick(object sender, EventArgs e)

{

watch.Stop();

double CorrectionFactor = (double)watch.ElapsedMilliseconds / simulationTimer.Interval;

if (CorrectionFactor > 0)

{

watch.Restart();

}

try

{

time += secondsPerSecond / Convert.ToDecimal(1000 / (simulationTimer.Interval \* CorrectionFactor));

}

catch (OverflowException)

{

time = 0;

MessageBox.Show("Time cannot increase as it has reached the decimal limit. Resetting time to 0.");

}

mousePositionLabel.Text = "Mouse Position (Metres): " + Convert.ToString(MousePosition.X \* inverseScaleFactor + xTranslation) + ", " + Convert.ToString((Height - MousePosition.Y) \* inverseScaleFactor + yTranslation);

if (tree.IsEnabled())

{

if (secondsPerSecond > 0)

{

tree.UpdateBodiesCurrentPositions(time);

}

if (mouseDown)

{

xTranslation = (mouseStartLocation.X - MousePosition.X) \* inverseScaleFactor;

yTranslation = (MousePosition.Y - mouseStartLocation.Y) \* inverseScaleFactor;

}

DrawGUI();

}

else

{

Graphics gr = CreateGraphics();

gr.Clear(BackColor);

return;

}

}

// Winforms graphics require previously drawn graphics to be drawn over back to the being the background colour, so a list of previously drawn locations is required.

private List<KeyValuePair<Vector3, double>> previousBodyLocations = new List<KeyValuePair<Vector3, double>> { new KeyValuePair<Vector3, double>(new Vector3(0, 0, 0), 0) };

private void DrawGUI()

{

List<KeyValuePair<Body, Vector3>> BodiesAndTheirPositions = tree.GetBodiesAndPositions();

List<Vector3> positions = new List<Vector3>();

foreach (KeyValuePair<Body, Vector3> body in BodiesAndTheirPositions)

{

positions.Add(body.Value);

}

Graphics gr = CreateGraphics();

Pen pen;

for (int i = 0; i < previousBodyLocations.Count; i++)

{

previousBodyLocations[i] = new KeyValuePair<Vector3, double>(previousBodyLocations[i].Key + new Vector3(xTranslation, yTranslation, 0), previousBodyLocations[i].Value);

}

foreach (KeyValuePair<Vector3, double> previousLocation in previousBodyLocations)

{

if (scrolled)

{

gr.Clear(BackColor);

scrolled = false;

break;

}

// To stop flashing of non-moved bodies on the screen, they are not drawn over with the background colour unless they specifically have moved.

if (positions.Contains(previousLocation.Key))

{

continue;

}

pen = new Pen(BackColor);

float CalculateDiameter()

{

float diameter = Convert.ToSingle(previousLocation.Value / inverseScaleFactor) \* 2;

if (diameter < 2)

{

diameter = 2;

}

return diameter;

}

float Diameter = CalculateDiameter();

gr.FillEllipse(pen.Brush, Convert.ToSingle((previousLocation.Key.X - xTranslation - previousLocation.Value) / inverseScaleFactor), Convert.ToSingle(Height - (previousLocation.Key.Y - yTranslation + previousLocation.Value) / inverseScaleFactor), Diameter, Diameter);

}

previousBodyLocations.Clear();

foreach (KeyValuePair<Body, Vector3> Body in BodiesAndTheirPositions)

{

pen = new Pen(Body.Key.Colour);

float CalculateDiameter()

{

float diameter = Convert.ToSingle(Body.Key.Radius / inverseScaleFactor) \* 2;

// To deal with the fact that planets are comparitively tiny in comparison to the distance between them, a minimum rendered size is defined.

if (diameter < 2)

{

diameter = 2;

}

return diameter;

}

float Diameter = CalculateDiameter();

try

{

gr.FillEllipse(pen.Brush, Convert.ToSingle((Body.Value.X - xTranslation - Body.Key.Radius) / inverseScaleFactor), Convert.ToSingle(Height - (Body.Value.Y - yTranslation + Body.Key.Radius) / inverseScaleFactor), Diameter, Diameter);

}

catch (OverflowException)

{

// If you zoom in too far, the FillEllipse function tries to calculate a massive ellipse which breaks the bounds of the number types used.

// The camera is reset and the user is told.

resetViewToolStripMenuItem\_Click(new object(), new EventArgs());

MessageBox.Show("Reached zoom limit. Resetting view.");

}

previousBodyLocations.Add(new KeyValuePair<Vector3, double>(Body.Value - new Vector3(xTranslation, yTranslation, 0), Body.Key.Radius));

}

}

// To allow dragging the camera around, the screen translation must be calculated while the user is dragging.

// This requires asynchronous running, which is done using a semaphore to send a message to the mouse down function to stop doing calculation once the user stops holding left click.

private SemaphoreSlim mouseUpSignal { get; set; } = new SemaphoreSlim(0, 1);

private bool mouseDown { get; set; } = false;

private Point mouseStartLocation { get; set; }

private async void MainForm\_MouseDown(object sender, MouseEventArgs e)

{

try

{

if (e.Button == MouseButtons.Left && tree.IsEnabled())

{

mouseDown = true;

mouseStartLocation = new Point(Convert.ToInt32(xTranslation / inverseScaleFactor + MousePosition.X), Convert.ToInt32(-yTranslation / inverseScaleFactor + MousePosition.Y));

await mouseUpSignal.WaitAsync();

mouseDown = false;

}

}

catch (OverflowException)

{

// If the user drags or zooms the camera to display space outside of float's range, the camera needs to be reset and the user told.

resetViewToolStripMenuItem\_Click(new object(), new EventArgs());

mouseDown = false;

MessageBox.Show("Attempted to click outside of the floating point maximum range. Resetting view.");

}

}

private void MainForm\_MouseUp(object sender, MouseEventArgs e)

{

if (e.Button == MouseButtons.Left && tree.IsEnabled())

{

mouseUpSignal.Release();

}

else if (e.Button == MouseButtons.Right && tree.IsEnabled())

{

Vector2 MouseLocation = new Vector2(MousePosition.X \* inverseScaleFactor + xTranslation, (Height - MousePosition.Y) \* inverseScaleFactor + yTranslation);

string name = "";

OrbitInformation OrbitInformation = tree.RegisterRightClick(MouseLocation, ref name, inverseScaleFactor);

if (Application.OpenForms.OfType<InformationForm>().Count() == 0)

{

informationForm = new InformationForm();

informationForm.SetLabels(name, OrbitInformation);

informationForm.Owner = this;

informationForm.Show();

}

else

{

informationForm.SetLabels(name, OrbitInformation);

informationForm.Focus();

}

}

}

// As the previous locations and sizes of the bodies on the screen are altered through zooming, the draw function must be told to compensate to still clear the previous locations correctly.

private bool scrolled { get; set; } = false;

private void MainForm\_MouseScroll(object sender, MouseEventArgs e)

{

if (tree.IsEnabled())

{

float scrollFactor;

if (e.Delta < 0)

{

scrollFactor = (float)6 / 5;

}

else

{

scrollFactor = (float)5 / 6;

}

xTranslation += MousePosition.X \* inverseScaleFactor - MousePosition.X \* inverseScaleFactor \* scrollFactor;

yTranslation += (Height - MousePosition.Y) \* inverseScaleFactor - (Height - MousePosition.Y) \* inverseScaleFactor \* scrollFactor;

inverseScaleFactor \*= scrollFactor;

scrolled = true;

}

}

// A custom file type is used here, but in essence it is simply a text file.

// This is because the binarywriter can only store primitive data types, making saving the custom classes and structures significantly harder through such means.

private void saveAsToolStripMenuItem\_Click(object sender, EventArgs e)

{

SaveFileDialog saveFileDialog = new SaveFileDialog();

saveFileDialog.Filter = "Simulation Save File|\*.simsave";

saveFileDialog.Title = "Save the Current Simulation";

saveFileDialog.ShowDialog();

if (saveFileDialog.FileName != "")

{

File.WriteAllText(saveFileDialog.FileName, tree.GetBodiesInSaveableFormat());

}

}

// The simulation is saved only when the user tells it too, so they are reminded to save in case they had forgotton.

private void loadToolStripMenuItem\_Click(object sender, EventArgs e)

{

MessageBox.Show("Warning: the current simulation will not be saved. Make sure you manually save before loading a different simulation.");

OpenFileDialog openFileDialog = new OpenFileDialog();

openFileDialog.Filter = "Simulation Save File|\*.simsave";

openFileDialog.Title = "Load a Saved Simulation";

openFileDialog.ShowDialog();

if (openFileDialog.FileName != "")

{

try

{

tree.LoadTreeFromSaveableFormat(File.ReadAllText(openFileDialog.FileName));

tree.UpdateBodiesCurrentPositionsUsingArraySearch(time);

MessageBox.Show("Load successful.");

}

catch (Exception)

{

MessageBox.Show("Loading failed. File likely corrupted or not in the correct format.");

}

}

}

private void exportABodysEphemerisToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (Application.OpenForms.OfType<ExportBodysEphemerisForm>().Count() == 0)

{

exportBodysEphemerisForm = new ExportBodysEphemerisForm(this);

exportBodysEphemerisForm.Owner = this;

exportBodysEphemerisForm.Show();

}

else

{

exportBodysEphemerisForm.Focus();

}

}

private void addBodyToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (Application.OpenForms.OfType<AddBodyForm>().Count() == 0)

{

addBodyForm = new AddBodyForm(this);

addBodyForm.Owner = this;

addBodyForm.Show();

}

else

{

addBodyForm.Focus();

}

}

private void removeBodyToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (Application.OpenForms.OfType<RemoveBodyForm>().Count() == 0)

{

removeBodyForm = new RemoveBodyForm(this);

removeBodyForm.Owner = this;

removeBodyForm.Show();

}

else

{

removeBodyForm.Focus();

}

}

private void simulationControllerToolStripMenuItem\_Click(object sender, EventArgs e)

{

if (Application.OpenForms.OfType<ControllerForm>().Count() == 0)

{

controllerForm = new ControllerForm(this);

controllerForm.Owner = this;

controllerForm.Show();

}

else

{

controllerForm.Focus();

}

}

private void resetViewToolStripMenuItem\_Click(object sender, EventArgs e)

{

inverseScaleFactor = 10000000000;

xTranslation = -9600000000000;

yTranslation = -5400000000000;

scrolled = true;

}

private void quitToolStripMenuItem\_Click(object sender, EventArgs e)

{

Close();

}

}

}

## Tree

using System.Numerics;

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal struct BodyTree

{

internal abstract class Node

{

public List<BodyNode> childNodes { get; private set; } = new List<BodyNode>();

public void AddChild(BodyNode newNode)

{

childNodes.Add(newNode);

}

}

internal class RootNode : Node

{

public RootNode()

{

enabled = true;

}

internal bool enabled { get; set; }

}

internal class BodyNode : Node

{

public Body assignedBody { get; private set; }

public BodyNode(Body body)

{

assignedBody = body;

}

public BodyNode()

{

assignedBody = null;

}

}

// The root node is intentionally somewhat disingenuously added to the tree.

// It allows for storing whether the tree is enabled or disabled, as well as serving as an anchor for other nodes to be added and removed from.

private RootNode rootNode { get; set; }

public BodyTree(FixedBody ReferenceBody)

{

rootNode = new RootNode();

rootNode.childNodes.Add(new BodyNode(ReferenceBody));

}

public bool IsEnabled()

{

return rootNode.enabled;

}

// This is implimented as a safety net, not as a prefered method of checking whether a function is runnable.

// As most areas that rely on the tree being enabled have other surrounding calculations, it is more efficient to cut it off early by checking enabled independently to calling a function and handling the error.

private void IfNotEnabled()

{

if (!IsEnabled())

{

throw new NotEnabledException();

}

}

// Once the tree has regained nodes, it can be enabled once again.

public void Reenable(FixedBody ReferenceBody)

{

if (!IsEnabled())

{

rootNode.enabled = true;

rootNode.childNodes[0] = new BodyNode(ReferenceBody);

}

else

{

throw new Exception("This body is already enabled.");

}

}

// This generic breadth first traversal allows for a function to be ran on the children as defined in the parameters.

private bool BreadthFirstTraversal(Func<BodyNode, bool> ChildCheckFunction)

{

Queue<BodyNode> queue = new Queue<BodyNode>();

List<BodyNode> visitedNodes = new List<BodyNode>();

queue.Enqueue(rootNode.childNodes[0]);

visitedNodes.Add(rootNode.childNodes[0]);

while (queue.Count > 0)

{

BodyNode node = queue.Dequeue();

foreach (BodyNode child in node.childNodes)

{

if (ChildCheckFunction(child))

{

return true;

}

if (!visitedNodes.Contains(child))

{

queue.Enqueue(child);

visitedNodes.Add(child);

}

}

}

return false;

}

// This is made public as while it is only used internally this as a function could be useful to a coder outside of only a private context.

public bool AreObjectsWithinRadius(double Radius, Vector3 CurrentRelativeToBackgroundPosition)

{

IfNotEnabled();

Queue<KeyValuePair<BodyNode, Vector3>> queue = new Queue<KeyValuePair<BodyNode, Vector3>>();

queue.Enqueue(new KeyValuePair<BodyNode, Vector3>(rootNode.childNodes[0], ((FixedBody)rootNode.childNodes[0].assignedBody).Position));

bool CheckIfWithinRadiusAndCalculateBodyPositions(BodyNode Child)

{

Vector3 childRelativeToBackgroundPosition = ((MovingBody)Child.assignedBody).currentPoint.RelativePosition + queue.Peek().Value;

Vector3 RelativeVector = childRelativeToBackgroundPosition - CurrentRelativeToBackgroundPosition;

if (RelativeVector.Length() < Radius)

{

return true;

}

if (Child.childNodes.Count > 0)

{

queue.Enqueue(new KeyValuePair<BodyNode, Vector3>(Child, childRelativeToBackgroundPosition));

}

return false;

}

return BreadthFirstTraversal(CheckIfWithinRadiusAndCalculateBodyPositions);

}

// This adds specifically a moving body defined using state vectors.

public void AddToTree(Color Colour, string NewBodyName, double NewBodyMass, decimal Time, Vector3 CurrentRelativeVelocity, Vector3 CurrentRelativeToBackgroundPosition, double Radius, bool VelocityRelativeToBackground)

{

IfNotEnabled();

BodyTree thisTree = this;

Vector3 CurrentCalculatedPosition = CurrentRelativeToBackgroundPosition - ((FixedBody)rootNode.childNodes[0].assignedBody).Position;

bool IsObjectInTree(BodyNode Child)

{

if (Child.assignedBody.Name == NewBodyName)

{

return true;

}

else

{

return false;

}

}

if (!BreadthFirstTraversal(IsObjectInTree))

{

FindAndPopulatePosition(rootNode.childNodes[0], CurrentRelativeVelocity, CurrentCalculatedPosition);

}

else

{

throw new Exception("The name chosen is already taken by a body in the simulation.");

}

void FindAndPopulatePosition(BodyNode parentNode, Vector3 currentCalculatedVelocity, Vector3 currentCalculatedPosition)

{

List<BodyNode> ValidParents = parentNode.childNodes.FindAll(x => (currentCalculatedPosition - ((MovingBody)x.assignedBody).currentPoint.RelativePosition).Length() < ((MovingBody)x.assignedBody).orbitInformation.HillSphereRadius);

if (ValidParents.Count > 0)

{

BodyNode CalculateValidParent()

{

if (ValidParents.Count > 1)

{

// In case of a conflict, the default winner is that with the highest mass / r^2.

// This is because gravity is proportional to mass / r^2.

return ValidParents.MaxBy(new Func<BodyNode, double>(x => x.assignedBody.Mass / Math.Pow((currentCalculatedPosition - ((MovingBody)x.assignedBody).currentPoint.RelativePosition).Length(), 2)));

}

else

{

return ValidParents[0];

}

}

BodyNode ValidParent = CalculateValidParent();

if (VelocityRelativeToBackground)

{

currentCalculatedVelocity -= ((MovingBody)ValidParent.assignedBody).currentPoint.RelativeVelocity;

}

currentCalculatedPosition -= ((MovingBody)ValidParent.assignedBody).currentPoint.RelativePosition;

FindAndPopulatePosition(ValidParent, currentCalculatedVelocity, currentCalculatedPosition);

}

else

{

bool failed = false;

double CalculateHillSphereRadius()

{

if (parentNode.assignedBody.GetType() == typeof(MovingBody))

{

return ((MovingBody)parentNode.assignedBody).orbitInformation.HillSphereRadius;

}

else

{

return double.PositiveInfinity;

}

}

double parentHillSphereRadius = CalculateHillSphereRadius();

MovingBody NewBody = new MovingBody(ref failed, thisTree, Colour, NewBodyName, NewBodyMass, parentNode.assignedBody.Mass, parentHillSphereRadius, parentNode.assignedBody.Radius, Radius, Time, currentCalculatedPosition, CurrentRelativeVelocity, CurrentRelativeToBackgroundPosition);

if (!failed)

{

parentNode.AddChild(new BodyNode(NewBody));

}

}

}

}

// This adds specifically a moving body defined using keplerian elements.

public void AddToTree(string ParentBodyName, Color Colour, string NewBodyName, double NewBodyMass, decimal StartingTimeFromEpoch, double StartingTrueAnomaly, double SemiMajorAxis, double Eccentricity, double Inclination, double LongitudeOfAscendingNode, double ArgumentOfPeriapsis, double Radius)

{

IfNotEnabled();

// Instead of checking whether the name is unique followed by finding the parent's location, you can combine the tree traversal to save processing time.

BodyNode parentNode = new BodyNode();

bool parentNodeFound = false;

if (ParentBodyName == rootNode.childNodes[0].assignedBody.Name)

{

parentNode = rootNode.childNodes[0];

parentNodeFound = true;

}

bool CheckIfChildIsEitherParentOrSharingChildName(BodyNode Child)

{

if (Child.assignedBody.Name == NewBodyName)

{

throw new Exception("The name chosen is already taken by a body in the simulation.");

}

if (Child.assignedBody.Name == ParentBodyName)

{

parentNode = Child;

parentNodeFound = true;

}

return false;

}

BreadthFirstTraversal(CheckIfChildIsEitherParentOrSharingChildName);

if (!parentNodeFound)

{

throw new Exception("Chosen parent doesn't exist in the simulation.");

}

bool failed = false;

double parentHillSphereRadius;

Vector3 parentPosition;

if (parentNode.assignedBody.GetType() == typeof(MovingBody))

{

parentHillSphereRadius = ((MovingBody)parentNode.assignedBody).orbitInformation.HillSphereRadius;

parentPosition = ((MovingBody)parentNode.assignedBody).currentPoint.RelativePosition;

}

else

{

parentHillSphereRadius = double.PositiveInfinity;

parentPosition = ((FixedBody)parentNode.assignedBody).Position;

}

MovingBody NewBody = new MovingBody(ref failed, this, Colour, NewBodyName, NewBodyMass, parentNode.assignedBody.Mass, parentHillSphereRadius, parentPosition, parentNode.assignedBody.Radius, Radius, StartingTimeFromEpoch, StartingTrueAnomaly, SemiMajorAxis, Eccentricity, Inclination, LongitudeOfAscendingNode, ArgumentOfPeriapsis);

if (!failed)

{

parentNode.AddChild(new BodyNode(NewBody));

}

}

public void RemoveFromTree(string RemovedBodyName)

{

IfNotEnabled();

// The reference body isn't covered in the search function, so it is checked independently first.

// This is useful as a different procedure has to be ran if the reference body is removed.

if (RemovedBodyName == rootNode.childNodes[0].assignedBody.Name.ToLower())

{

rootNode.childNodes[0] = new BodyNode();

rootNode.enabled = false;

MessageBox.Show("Body removed succesfully.");

return;

}

List<BodyNode> visitedNodes = new List<BodyNode>();

if (DepthFirstSearch(rootNode.childNodes[0]))

{

MessageBox.Show("Body removed succesfully.");

}

else

{

MessageBox.Show("Body not found.");

}

bool DepthFirstSearch(BodyNode parent)

{

visitedNodes.Add(parent);

foreach (BodyNode child in parent.childNodes)

{

if (child.assignedBody.Name.ToLower() == RemovedBodyName)

{

parent.childNodes.Remove(child);

return true;

}

if (!visitedNodes.Contains(child))

{

if (DepthFirstSearch(child))

{

return true;

}

}

}

return false;

}

}

public void ExportBodysEphemeris(string ExportedBodyName)

{

IfNotEnabled();

BodyNode childNode = new BodyNode();

List<BodyNode> visitedNodes = new List<BodyNode>();

if (DepthFirstSearch(rootNode.childNodes[0]))

{

SaveFileDialog saveFileDialog = new SaveFileDialog();

saveFileDialog.Filter = "csv|\*.csv";

saveFileDialog.Title = "Export Body's Ephemeris";

saveFileDialog.ShowDialog();

if (saveFileDialog.FileName != "")

{

File.WriteAllText(saveFileDialog.FileName, ((MovingBody)childNode.assignedBody).ReturnEphemeris());

}

}

else

{

MessageBox.Show("Body not found.");

}

bool DepthFirstSearch(BodyNode parent)

{

visitedNodes.Add(parent);

foreach (BodyNode child in parent.childNodes)

{

if (child.assignedBody.Name.ToLower() == ExportedBodyName)

{

childNode = child;

return true;

}

if (!visitedNodes.Contains(child))

{

if (DepthFirstSearch(child))

{

return true;

}

}

}

return false;

}

}

public void LoadTreeFromSaveableFormat(string InputString)

{

Vector3 CreateVector3FromString(string InputString)

{

string[] processedInputStringAsArray = InputString.Substring(1, InputString.Length - 2).Split(", ");

return new Vector3(Convert.ToSingle(processedInputStringAsArray[0]), Convert.ToSingle(processedInputStringAsArray[1]), Convert.ToSingle(processedInputStringAsArray[2]));

}

MovingBody ConvertStringToMovingBody(string[] InputStringArray, double ParentMass)

{

string Name = InputStringArray[1];

double Mass = Convert.ToDouble(InputStringArray[2]);

double Radius = Convert.ToDouble(InputStringArray[3]);

Color Colour = Color.FromArgb(Convert.ToInt32(InputStringArray[4]));

decimal StartingTimeFromEpoch = Convert.ToDecimal(InputStringArray[5]);

double StartingTrueAnomaly = Convert.ToDouble(InputStringArray[6]);

string OrbitInformationAsString = InputStringArray[7].Substring(18, InputStringArray[7].Length - 20);

string[] orbitInformationAsStringArray = OrbitInformationAsString.Split(", ");

for (int i = 0; i < orbitInformationAsStringArray.Length; i++)

{

if (orbitInformationAsStringArray[i].Contains("="))

{

orbitInformationAsStringArray[i] = orbitInformationAsStringArray[i].Substring(orbitInformationAsStringArray[i].IndexOf('=') + 2);

}

}

double LongitudeOfAscendingNode = Convert.ToDouble(orbitInformationAsStringArray[0]);

double ArgumentOfPeriapsis = Convert.ToDouble(orbitInformationAsStringArray[1]);

double LongitudeOfPeriapsis = Convert.ToDouble(orbitInformationAsStringArray[2]);

double Inclination = Convert.ToDouble(orbitInformationAsStringArray[3]);

double Eccentricity = Convert.ToDouble(orbitInformationAsStringArray[4]);

double SemiMajorAxis = Convert.ToDouble(orbitInformationAsStringArray[5]);

double SemiMinorAxis = Convert.ToDouble(orbitInformationAsStringArray[6]);

double Periapsis = Convert.ToDouble(orbitInformationAsStringArray[7]);

double Apoapsis = Convert.ToDouble(orbitInformationAsStringArray[8]);

double AngularMomentum = Convert.ToDouble(orbitInformationAsStringArray[9]);

Vector3 SpecificAngularMomentum = new Vector3(Convert.ToSingle(orbitInformationAsStringArray[10]), Convert.ToSingle(orbitInformationAsStringArray[11]), Convert.ToSingle(orbitInformationAsStringArray[12]));

Vector3 EccentricityVector = new Vector3(Convert.ToSingle(orbitInformationAsStringArray[13]), Convert.ToSingle(orbitInformationAsStringArray[14]), Convert.ToSingle(orbitInformationAsStringArray[15]));

double TotalEnergy = Convert.ToDouble(orbitInformationAsStringArray[16]);

double HillSphereRadius = Convert.ToDouble(orbitInformationAsStringArray[17]);

decimal OrbitalPeriod = Convert.ToDecimal(orbitInformationAsStringArray[18]);

return new MovingBody(Colour, Name, Mass, Radius, StartingTimeFromEpoch, ParentMass, StartingTrueAnomaly, LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius);

}

void FindAndPopulatePosition(BodyNode parentNode, List<string> path, string[] referenceBodyAsAStringArray)

{

if (path.Count > 1)

{

int childNodePosition = parentNode.childNodes.FindIndex(child => child.assignedBody.Name == path[1]);

FindAndPopulatePosition(parentNode.childNodes[childNodePosition], path.Skip(1).ToList(), referenceBodyAsAStringArray);

}

else if (path.Count == 1)

{

parentNode.childNodes.Add(new BodyNode(ConvertStringToMovingBody(referenceBodyAsAStringArray, parentNode.assignedBody.Mass)));

}

}

rootNode.enabled = false;

rootNode.childNodes.Clear();

List<string> bodies = InputString.Split('\n').ToList();

string[] referenceBodyAsAStringArray = bodies[0].Split(",|");

FixedBody referenceBody = new FixedBody(Color.FromArgb(Convert.ToInt32(referenceBodyAsAStringArray[4])), referenceBodyAsAStringArray[1], CreateVector3FromString(referenceBodyAsAStringArray[5]), Convert.ToDouble(referenceBodyAsAStringArray[2]), Convert.ToDouble(referenceBodyAsAStringArray[3]));

rootNode.AddChild(new BodyNode(referenceBody));

bodies.Remove(bodies[0]);

while (bodies.Count > 0)

{

referenceBodyAsAStringArray = bodies[0].Split(",|");

if (referenceBodyAsAStringArray[0] != "")

{

List<string> heritage = referenceBodyAsAStringArray[0].Substring(1, referenceBodyAsAStringArray[0].Length - 2).Split(", ").ToList();

FindAndPopulatePosition(rootNode.childNodes[0], heritage.ToList(), referenceBodyAsAStringArray);

}

bodies.Remove(bodies[0]);

}

rootNode.enabled = true;

}

public string GetBodiesInSaveableFormat()

{

IfNotEnabled();

string output = "";

List<string> Heritage = new List<string>();

List<KeyValuePair<string[], BodyNode>> visitedNodes = new List<KeyValuePair<string[], BodyNode>>();

void DepthFirstTraversal(BodyNode parent)

{

visitedNodes.Add(new KeyValuePair<string[], BodyNode>(Heritage.ToArray(), parent));

Heritage.Add(parent.assignedBody.Name);

foreach (BodyNode child in parent.childNodes)

{

if (visitedNodes.FindAll(x => x.Value.assignedBody.Name == child.assignedBody.Name).Count() == 0)

{

if (child.childNodes.Count != 0)

{

DepthFirstTraversal(child);

if (parent.childNodes.Last() == child)

{

Heritage.Remove(Heritage.Last());

}

}

else

{

visitedNodes.Add(new KeyValuePair<string[], BodyNode>(Heritage.ToArray(), child));

}

}

}

Heritage.Remove(Heritage.Last());

}

DepthFirstTraversal(rootNode.childNodes[0]);

foreach (KeyValuePair<string[], BodyNode> HeritageAndNode in visitedNodes)

{

output += "[";

if (HeritageAndNode.Key.Length > 0)

{

string subOutput = "";

foreach (string bodyName in HeritageAndNode.Key)

{

subOutput += ", " + bodyName;

}

output += subOutput.Remove(0, 2);

}

output += "],|";

output += HeritageAndNode.Value.assignedBody.ToString();

}

return output;

}

// A breadth first search is done on the tree to gather all of their current locations for display.

public List<KeyValuePair<Body, Vector3>> GetBodiesAndPositions()

{

IfNotEnabled();

Queue<KeyValuePair<BodyNode, Vector3>> queue = new Queue<KeyValuePair<BodyNode, Vector3>>();

List<KeyValuePair<BodyNode, Vector3>> visitedNodes = new List<KeyValuePair<BodyNode, Vector3>>();

queue.Enqueue(new KeyValuePair<BodyNode, Vector3>(rootNode.childNodes[0], ((FixedBody)rootNode.childNodes[0].assignedBody).Position));

visitedNodes.Add(new KeyValuePair<BodyNode, Vector3>(rootNode.childNodes[0], ((FixedBody)rootNode.childNodes[0].assignedBody).Position));

bool UpdatePositions(BodyNode Child)

{

Vector3 ChildRelativeToBackgroundPosition = ((MovingBody)Child.assignedBody).currentPoint.RelativePosition + queue.Peek().Value;

if (!visitedNodes.Contains(new KeyValuePair<BodyNode, Vector3>(Child, ChildRelativeToBackgroundPosition)))

{

if (Child.childNodes.Count > 0)

{

queue.Enqueue(new KeyValuePair<BodyNode, Vector3>(Child, ChildRelativeToBackgroundPosition));

}

visitedNodes.Add(new KeyValuePair<BodyNode, Vector3>(Child, ChildRelativeToBackgroundPosition));

}

if (queue.Peek().Key.childNodes.Last().assignedBody.Name == Child.assignedBody.Name)

{

queue.Dequeue();

}

return false;

}

BreadthFirstTraversal(UpdatePositions);

List<KeyValuePair<Body, Vector3>> bodies = new List<KeyValuePair<Body, Vector3>>();

foreach (KeyValuePair<BodyNode, Vector3> nodeAndRelativeToBackgroundPosition in visitedNodes)

{

bodies.Add(new KeyValuePair<Body, Vector3>(nodeAndRelativeToBackgroundPosition.Key.assignedBody, nodeAndRelativeToBackgroundPosition.Value));

}

return bodies;

}

public void UpdateBodiesCurrentPositions(decimal Time)

{

IfNotEnabled();

bool UpdateChild(BodyNode child)

{

((MovingBody)child.assignedBody).UpdateCurrentPoint(Time);

return false;

}

BreadthFirstTraversal(UpdateChild);

}

public void UpdateBodiesCurrentPositionsUsingArraySearch(decimal Time)

{

IfNotEnabled();

bool UpdateChild(BodyNode child)

{

((MovingBody)child.assignedBody).UpdateCurrentPointUsingArraySearch(Time);

return false;

}

BreadthFirstTraversal(UpdateChild);

}

// To calculate what is being clicked on by the mouse, the tree has to be queried.

public OrbitInformation RegisterRightClick(Vector2 rightClickPosition, ref string name, float inverseScaleFactor)

{

IfNotEnabled();

BodyNode referenceBody = rootNode.childNodes[0];

return FindClickedObject(referenceBody, rightClickPosition, ref name);

OrbitInformation FindClickedObject(BodyNode parentNode, Vector2 currentCalculatedPosition, ref string name)

{

List<BodyNode> validParents = parentNode.childNodes.FindAll(x => (currentCalculatedPosition - new Vector2(((MovingBody)x.assignedBody).currentPoint.RelativePosition.X, ((MovingBody)x.assignedBody).currentPoint.RelativePosition.Y)).Length() < 20 \* inverseScaleFactor);

List<BodyNode> validParentsByRadius = parentNode.childNodes.FindAll(x => (currentCalculatedPosition - new Vector2(((MovingBody)x.assignedBody).currentPoint.RelativePosition.X, ((MovingBody)x.assignedBody).currentPoint.RelativePosition.Y)).Length() < x.assignedBody.Radius);

foreach (BodyNode validParentByRadius in validParentsByRadius)

{

if (!validParents.Contains(validParentByRadius))

{

validParents.Add(validParentByRadius);

}

}

if (validParents.Count > 0)

{

BodyNode CalculateValidParent()

{

if (validParents.Count > 1)

{

// Field strength is proportional to M / r^2, so I have made in the case of a conflict the decided body be that which provides the largest "proportionalised" field strength at the mouse Position.

return validParents.MaxBy(new Func<BodyNode, double>(x => x.assignedBody.Mass / Math.Pow((currentCalculatedPosition - new Vector2(((MovingBody)x.assignedBody).currentPoint.RelativePosition.X, ((MovingBody)x.assignedBody).currentPoint.RelativePosition.Y)).Length(), 2)));

}

else

{

return validParents[0];

}

}

BodyNode ValidParent = CalculateValidParent();

currentCalculatedPosition -= new Vector2(((MovingBody)ValidParent.assignedBody).currentPoint.RelativePosition.X, ((MovingBody)ValidParent.assignedBody).currentPoint.RelativePosition.Y);

return FindClickedObject(ValidParent, currentCalculatedPosition, ref name);

}

else

{

name = parentNode.assignedBody.Name;

if (parentNode.assignedBody.Name == referenceBody.assignedBody.Name)

{

return new OrbitInformation(double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, double.NaN, new Vector3(0, 0, 0), new Vector3(0, 0, 0), double.NaN, double.NaN, -1);

}

}

return ((MovingBody)parentNode.assignedBody).orbitInformation;

}

}

}

}

## Body.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal interface Body

{

public abstract string Name { get; init; }

public abstract double Mass { get; init; }

public abstract double Radius { get; init; }

public abstract Color Colour { get; init; }

public abstract string ToString();

}

}

## Fixed Body.cs

using System.Numerics;

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal class FixedBody : Body

{

public string Name { get; init; }

public Vector3 Position { get; init; }

public double Mass { get; init; }

public double Radius { get; init; }

public Color Colour { get; init; }

public FixedBody(Color Colour, string Name, Vector3 Position, double Mass, double Radius)

{

this.Name = Name;

this.Position = Position;

this.Mass = Mass;

this.Radius = Radius;

this.Colour = Colour;

}

// Instead of using commas, a comma and pipe symbol together are used to stop ambiguity in places such as converting a vector into a string.

public override string ToString()

{

string output = "";

output += Name + ",|";

output += Mass.ToString() + ",|";

output += Radius.ToString() + ",|";

output += Colour.ToArgb().ToString() + ",|";

output += Position.ToString() + "\n";

return output;

}

}

}

## Moving Body.cs

using System.Numerics;

using System.Web;

using System.Windows.Forms.Design;

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal class MovingBody : Body

{

// A record to store data about points in an orbit.

// While only relative position and time are used, the rest of the information was added to allow a future coder to implement features based on other orbital position statistics.

public record PointInformation(decimal Time, double AngleOfVelocityToTangent, double TrueAnomaly, Vector3 RelativePosition, Vector3 RelativeVelocity);

// The gravitational constant; used for some equations.

private const double G = 0.0000000000667;

// The body's name.

public string Name { get; init; }

// The body's mass.

public double Mass { get; init; }

// The radius of the body (all bodies are assumed to be spherical).

public double Radius { get; init; }

// The displayed colour of the body.

public Color Colour { get; init; }

// The time the body was created at; used for calculating Position at a given time.

private decimal StartingTimeFromEpoch { get; init; }

// The information about a body's orbit.

public OrbitInformation orbitInformation { get; private set; }

private double StartingTrueAnomaly { get; set; }

// The body's current point information.

public PointInformation currentPoint { get; private set; }

// The full orbit's precalculated data points; this is done on instantiation of the class as it is significantly better for performance and the memory tradeoff isn't particularly high.

// Unfortunately, the equation to calculate time is also non analytically solvable for distance between bodies, which means you can't calculate the current point directly using time, and so predetermine points at given times are used to be approximately equal.

private PointInformation[] predeterminedPoints { get; set; }

// This initialiser is specifically for initialising with state vectors.

public MovingBody(ref bool failed, BodyTree Tree, Color Colour, string Name, double Mass, double ParentMass, double ParentHillSphereRadius, double ParentRadius, double Radius, decimal StartingTimeFromEpoch, Vector3 StartingRelativePosition, Vector3 StartingRelativeVelocity, Vector3 StartingRelativeToBackgroundPosition)

{

this.Name = Name;

this.Mass = Mass;

this.StartingTimeFromEpoch = StartingTimeFromEpoch;

this.Radius = Radius;

this.Colour = Colour;

// An arbitrarily chosen parent:child mass ratio of 50:1 has been chosen as the smallest difference in mass before warning the user of innaccuracies.

// For example, the Earth:Moon ratio is about 81:1 for comparison; despite being accepted under the chosen ratio, the Earth:Moon barycentre is somewhat moved from Earth's center in more accurate simulations.

// Hence, I believe this to be a generous but reasonable compromise.

if (ParentMass / 50 < Mass)

{

MessageBox.Show("Orbiting body has a significantly high mass ratio to it's parent. The simulation may have increased inaccuracies.");

}

failed = InitialiseOrbitConstantsFromStateVectors(Tree, ParentMass, ParentHillSphereRadius, ParentRadius, StartingRelativePosition, StartingRelativeVelocity, StartingRelativeToBackgroundPosition);

}

// This initialiser is specifically for initialising with keplerian elements.

public MovingBody(ref bool failed, BodyTree Tree, Color Colour, string Name, double Mass, double ParentMass, double ParentHillSphereRadius, Vector3 ParentPosition, double ParentRadius, double Radius, decimal StartingTimeFromEpoch, double StartingTrueAnomaly, double SemiMajorAxis, double Eccentricity, double Inclination, double LongitudeOfAscendingNode, double ArgumentOfPeriapsis)

{

this.Name = Name;

this.Mass = Mass;

this.StartingTimeFromEpoch = StartingTimeFromEpoch;

this.Radius = Radius;

this.Colour = Colour;

this.StartingTrueAnomaly = StartingTrueAnomaly;

// An arbitrarily chosen parent:child mass ratio of 50:1 has been chosen as the smallest difference in mass before warning the user of innaccuracies.

// For example, the Earth:Moon ratio is about 81:1 for comparison; despite being accepted under the chosen ratio, the Earth:Moon barycentre is somewhat moved from Earth's center in more accurate simulations.

// Hence, I believe this to be a generous but reasonable compromise.

if (ParentMass / 50 < Mass)

{

MessageBox.Show("Orbiting body has a significantly high mass ratio to it's parent. The simulation may have increased inaccuracies.");

}

failed = InitialiseOrbitConstantsFromKeplerianElements(Tree, ParentMass, ParentHillSphereRadius, ParentPosition, ParentRadius, StartingTrueAnomaly, SemiMajorAxis, Eccentricity, Inclination, LongitudeOfAscendingNode, ArgumentOfPeriapsis);

}

// This initialiser is to be exclusively used in creating a tree from a save.

public MovingBody(Color Colour, string Name, double Mass, double Radius, decimal StartingTimeFromEpoch, double ParentMass, double StartingTrueAnomaly, double LongitudeOfAscendingNode, double ArgumentOfPeriapsis, double LongitudeOfPeriapsis, double Inclination, double Eccentricity, double SemiMajorAxis, double SemiMinorAxis, double Periapsis, double Apoapsis, double AngularMomentum, Vector3 SpecificAngularMomentum, Vector3 EccentricityVector, double TotalEnergy, double HillSphereRadius)

{

this.Name = Name;

this.Mass = Mass;

this.StartingTimeFromEpoch = StartingTimeFromEpoch;

this.Radius = Radius;

this.Colour = Colour;

ReinitialiseOrbitConstantsFromKeplerianElements(ParentMass, StartingTrueAnomaly, LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius);

}

private bool InitialiseOrbitConstantsFromStateVectors(BodyTree Tree, double ParentMass, double ParentHillsSphereRadius, double ParentRadius, Vector3 StartingRelativePosition, Vector3 StartingRelativeVelocity, Vector3 StartingRelativeToBackgroundPosition)

{

double StartingRelativeAbsPosition = StartingRelativePosition.Length();

double StartingRelativeAbsVelocity = StartingRelativeVelocity.Length();

// This is the vector perpendicular from the other 2 vectors which is a measure of angular momentum / mass.

Vector3 SpecificAngularMomentum = Vector3.Cross(StartingRelativePosition, StartingRelativeVelocity);

// e is the eccentricity vector, a vector with no unit that points from apoapsis to periapsis with magnitude equal to scalar eccentricity.

Vector3 EccentricityVector = Vector3.Divide(Vector3.Cross(StartingRelativeVelocity, SpecificAngularMomentum), Convert.ToSingle(G \* ParentMass)) - Vector3.Divide(StartingRelativePosition, Convert.ToSingle(StartingRelativeAbsPosition));

// Orbital eccentricity is a unitless (dimensionless) parameter which gives a measure of how far an orbit is from a perfect circle.

double Eccentricity = EccentricityVector.Length();

// This is the distance from the centre of the orbit and the apoapsis or periapsis (the longest diameter).

double SemiMajorAxis = 1 / (2 / StartingRelativeAbsPosition - Math.Pow(StartingRelativeAbsVelocity, 2) / (G \* ParentMass));

double HillSphereRadius = SemiMajorAxis \* (1 - Eccentricity) \* Math.Pow(Mass / (3 \* ParentMass), (double)1 / 3);

// Check that no other bodies already exist in that hill sphere.

if (Tree.AreObjectsWithinRadius(HillSphereRadius, StartingRelativeToBackgroundPosition))

{

return true;

}

// This is the shortest distance from the parent body.

// There is an acceptable error in this calculation added on to stop edge case calculation errors caused by rounding errors.

double Periapsis = SemiMajorAxis \* (1 - Eccentricity) + 0.0001;

if (Periapsis <= ParentRadius)

{

throw new Exception("The orbit mustn't collide with the parent body's surface.");

}

// This is the furthest distance from the parent body.

// There is an acceptable error in this calculation added on to stop edge case calculation errors caused by rounding errors.

double Apoapsis = SemiMajorAxis \* (1 + Eccentricity) - 0.0001;

if (Apoapsis + HillSphereRadius > ParentHillsSphereRadius)

{

MessageBox.Show("Any subsequently made bodies must lie in this body's parent's hill sphere before being accepted into this body’s gravitational influence. This is a consequence of the created simulation being potentially innaccurate.");

}

if (Apoapsis > ParentHillsSphereRadius)

{

throw new Exception("The orbit must lie within the parent's hill sphere.");

}

// The semi-minor axis is the shortest diameter of an ellipse. It is perpendicular to the semi-major axis through the centre point if drawn on a graph.

double SemiMinorAxis = SemiMajorAxis \* Math.Sqrt(1 - Math.Pow(Eccentricity, 2));

// The angle between the equatorial plane (in this case xy) and the plane of the orbit.

double Inclination = Math.Acos(SpecificAngularMomentum.Z / SpecificAngularMomentum.Length());

// N is the vector pointing towards the ascending node. If the ascending node doesn't exist (inclination = 0 or pi) it is assigned as a generic Vector3 as it isn't used. It is used for the following equations.

Vector3 CalculateN()

{

if (Inclination != 0 && Inclination != Math.PI)

{

return Vector3.Cross(new Vector3(0, 0, 1), SpecificAngularMomentum);

}

else

{

return new Vector3();

}

}

Vector3 N = CalculateN();

// The angle between the +x direction and the ascending node.

double CalculateLongitudeOfAscendingNode()

{

if (Inclination != 0 && Inclination != Math.PI)

{

if (N.Y >= 0)

{

return Math.Acos(N.X / N.Length());

}

else

{

return 2 \* Math.PI - Math.Acos(N.X / N.Length());

}

}

else

{

return 0;

}

}

double LongitudeOfAscendingNode = CalculateLongitudeOfAscendingNode();

// The angle between the +x direction and the periapsis.

double CalculateArgumentOfPeriapsis()

{

double argumentOfPeriapsis;

if (Inclination == 0 || Inclination == Math.PI)

{

argumentOfPeriapsis = Math.Atan2(EccentricityVector.Y, EccentricityVector.X);

if (Vector3.Cross(StartingRelativePosition, StartingRelativeVelocity).Z < 0)

{

argumentOfPeriapsis = 2 \* Math.PI - argumentOfPeriapsis;

}

}

else

{

argumentOfPeriapsis = Math.Acos(Vector3.Dot(N, EccentricityVector) / (N.Length() \* EccentricityVector.Length()));

if (EccentricityVector.Z < 0)

{

argumentOfPeriapsis = 2 \* Math.PI - argumentOfPeriapsis;

}

}

return argumentOfPeriapsis;

}

double ArgumentOfPeriapsis = CalculateArgumentOfPeriapsis();

// This is the longitude at which the periapsis would occur is the orbit's inclination was zero.

double LongitudeOfPeriapsis = LongitudeOfAscendingNode + ArgumentOfPeriapsis;

// This rotates the orbit such that its periapsis and apoapsis lie on the x axis, specifically such that the periapsis is furthest along the +x direction.

double AngleFromPeriapsis = 2 \* Math.PI - ArgumentOfPeriapsis;

TransformationMatrix3x3 RotationMatrixForPeriapsis = new TransformationMatrix3x3(new double[,] { { Math.Cos(AngleFromPeriapsis), Math.Sin(AngleFromPeriapsis), 0 }, { -Math.Sin(AngleFromPeriapsis), Math.Cos(AngleFromPeriapsis), 0 }, { 0, 0, 1 } });

double a;

double b;

if (Inclination != 0)

{

Vector3 RotateVector3OntoTheXYPlane(Vector3 vector3)

{

// This rotates the line between the ascending and decending node to be along the +x axis.

double AngleFromAscendingNode = 2 \* Math.PI - LongitudeOfAscendingNode;

TransformationMatrix3x3 RotationMatrixForAscendingNode = new TransformationMatrix3x3(new double[,] { { Math.Cos(AngleFromAscendingNode), Math.Sin(AngleFromAscendingNode), 0 }, { -Math.Sin(AngleFromAscendingNode), Math.Cos(AngleFromAscendingNode), 0 }, { 0, 0, 1 } });

// This rotates the orbit along it's node line (the x axis), hence giving a flat orbit.

double AngleFromInclination = 2 \* Math.PI - Inclination;

TransformationMatrix3x3 RotationMatrixForInclination = new TransformationMatrix3x3(new double[,] { { 1, 0, 0 }, { 0, Math.Cos(AngleFromInclination), Math.Sin(AngleFromInclination) }, { 0, -Math.Sin(AngleFromInclination), Math.Cos(AngleFromInclination) } });

vector3 = RotationMatrixForPeriapsis.Transform(vector3);

vector3 = RotationMatrixForInclination.Transform(vector3);

vector3 = RotationMatrixForAscendingNode.Transform(vector3);

return vector3;

}

Vector3 FlattenedRelativeStartPosition = RotateVector3OntoTheXYPlane(StartingRelativePosition);

Vector3 FlattenedRelativeStartVelocity = RotateVector3OntoTheXYPlane(StartingRelativeVelocity);

StartingTrueAnomaly = Math.Atan2(FlattenedRelativeStartPosition.X, FlattenedRelativeStartPosition.Y);

a = FlattenedRelativeStartPosition.Y / FlattenedRelativeStartPosition.X;

b = FlattenedRelativeStartVelocity.Y / FlattenedRelativeStartVelocity.X;

}

else

{

Vector3 RotatedPosition = RotationMatrixForPeriapsis.Transform(StartingRelativePosition);

StartingTrueAnomaly = Math.Atan2(RotatedPosition.X, RotatedPosition.Y);

a = StartingRelativePosition.Y / StartingRelativePosition.X;

b = StartingRelativeVelocity.Y / StartingRelativeVelocity.X;

}

double CosStartingAngleOfVelocity = (a / Math.Sqrt(Math.Pow(a, 2) + 1)) \* (1 / Math.Sqrt(Math.Pow(b, 2) + 1)) - (1 / Math.Sqrt(Math.Pow(a, 2) + 1)) \* (b / Math.Sqrt(Math.Pow(b, 2) + 1));

double StartingPerpendicularAbsVelocity = StartingRelativeAbsVelocity \* CosStartingAngleOfVelocity;

// This is the measure of momentum around an axis.

double AngularMomentum = Mass \* StartingPerpendicularAbsVelocity \* StartingRelativeAbsPosition;

double TotalEnergy = 0.5 \* Mass \* Math.Pow(StartingRelativeAbsVelocity, 2) + -1 \* G \* ParentMass \* Mass / StartingRelativeAbsPosition;

currentPoint = new PointInformation(StartingTimeFromEpoch, Math.Acos(CosStartingAngleOfVelocity), double.NaN, StartingRelativePosition, StartingRelativeVelocity);

// This is the amount of time taken to complete an orbit in seconds.

decimal orbitalPeriod;

// 200000 data points was chosen as a good trade between memory and performance.

predeterminedPoints = CreatePoints(new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius, 0), StartingTimeFromEpoch, 200000, ParentMass, StartingRelativeAbsPosition, out orbitalPeriod);

orbitInformation = new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius, orbitalPeriod);

return false;

}

private bool InitialiseOrbitConstantsFromKeplerianElements(BodyTree Tree, double ParentMass, double ParentHillsSphereRadius, Vector3 CurrentParentPosition, double ParentRadius, double StartingTrueAnomaly, double SemiMajorAxis, double Eccentricity, double Inclination, double LongitudeOfAscendingNode, double ArgumentOfPeriapsis)

{

double startingRelativeAbsPosition = (SemiMajorAxis \* (1 - Math.Pow(Eccentricity, 2))) / (1 + Eccentricity \* Math.Cos(StartingTrueAnomaly));

Vector3 startingRelativePosition = new Vector3(Convert.ToSingle(startingRelativeAbsPosition \* Math.Cos(StartingTrueAnomaly)), Convert.ToSingle(startingRelativeAbsPosition \* Math.Sin(StartingTrueAnomaly)), 0);

// This is the furthest distance from the parent body.

double Apoapsis = SemiMajorAxis \* (1 + Eccentricity);

// This is the shortest distance from the parent body.

double Periapsis = SemiMajorAxis \* (1 - Eccentricity);

if (Periapsis <= ParentRadius)

{

throw new Exception("The orbit mustn't collide with the parent body's surface.");

}

double TotalVelocitySquaredAtPeriapsis = G \* ParentMass \* (2 / Periapsis - 1 / SemiMajorAxis);

Vector3 RotateVector3OntoTheOrbitalPath(Vector3 vector3)

{

TransformationMatrix3x3 RotationMatrixForPeriapsis = new TransformationMatrix3x3(new double[,] { { Math.Cos(ArgumentOfPeriapsis), Math.Sin(ArgumentOfPeriapsis), 0 }, { -Math.Sin(ArgumentOfPeriapsis), Math.Cos(ArgumentOfPeriapsis), 0 }, { 0, 0, 1 } });

TransformationMatrix3x3 RotationMatrixForInclination = new TransformationMatrix3x3(new double[,] { { 1, 0, 0 }, { 0, Math.Cos(Inclination), Math.Sin(Inclination) }, { 0, -Math.Sin(Inclination), Math.Cos(Inclination) } });

TransformationMatrix3x3 RotationMatrixForAscendingNode = new TransformationMatrix3x3(new double[,] { { Math.Cos(LongitudeOfAscendingNode), Math.Sin(LongitudeOfAscendingNode), 0 }, { -Math.Sin(LongitudeOfAscendingNode), Math.Cos(LongitudeOfAscendingNode), 0 }, { 0, 0, 1 } });

vector3 = RotationMatrixForPeriapsis.Transform(vector3);

vector3 = RotationMatrixForInclination.Transform(vector3);

vector3 = RotationMatrixForAscendingNode.Transform(vector3);

return vector3;

}

Vector3 PeriapsisRelativePosition = RotateVector3OntoTheOrbitalPath(new Vector3(Convert.ToSingle(Periapsis), 0, 0));

Vector3 PeriapsisRelativeVelocity = RotateVector3OntoTheOrbitalPath(new Vector3(0, Convert.ToSingle(Math.Sqrt(TotalVelocitySquaredAtPeriapsis)), 0));

double TotalEnergy = (-G \* ParentMass \* Mass) / (Apoapsis + Periapsis);

double HillSphereRadius = SemiMajorAxis \* (1 - Eccentricity) \* Math.Pow(Mass / (3 \* ParentMass), (double)1 / 3);

Vector3 StartingRelativeToBackgroundPosition = startingRelativePosition + CurrentParentPosition;

// Check that no other bodies already exist in that hill sphere.

if (Tree.AreObjectsWithinRadius(HillSphereRadius, StartingRelativeToBackgroundPosition))

{

return true;

}

if (Apoapsis + HillSphereRadius > ParentHillsSphereRadius)

{

MessageBox.Show("Any subsequently made bodies must lie in this body's parent's hill sphere before being accepted into this body’s gravitational influence. This is a consequence of the created simulation being potentially innaccurate.");

}

if (Apoapsis > ParentHillsSphereRadius)

{

throw new Exception("The orbit must lie within the parent's hill sphere.");

}

// The semi-minor axis is the shortest diameter of an ellipse. It is perpendicular to the semi-major axis through the centre point if drawn on a graph.

double SemiMinorAxis = SemiMajorAxis \* Math.Sqrt(1 - Math.Pow(Eccentricity, 2));

// This is the vector perpendicular from the other 2 vectors which is a measure of angular momentum / mass.

Vector3 SpecificAngularMomentum = Vector3.Cross(PeriapsisRelativePosition, PeriapsisRelativeVelocity);

// e is the eccentricity vector, a vector with no unit that points from apoapsis to periapsis with magnitude equal to scalar eccentricity.

Vector3 EccentricityVector = Vector3.Divide(Vector3.Cross(PeriapsisRelativeVelocity, SpecificAngularMomentum), Convert.ToSingle(G \* ParentMass)) - Vector3.Divide(PeriapsisRelativePosition, PeriapsisRelativePosition.Length());

// This is the longitude at which the periapsis would occur is the orbit's inclination was zero.

double LongitudeOfPeriapsis = LongitudeOfAscendingNode + ArgumentOfPeriapsis;

double StartingPeriapsisAbsVelocity = PeriapsisRelativeVelocity.Length();

// This is the measure of momentum around an axis.

double AngularMomentum = Mass \* StartingPeriapsisAbsVelocity \* PeriapsisRelativePosition.Length();

double StartingRelativeAbsVelocitySquared = G \* ParentMass \* (2 / Periapsis - 1 / SemiMajorAxis);

double CalculateStartingPerpendicularVelocity()

{

double startingPerpendicularVelocity = AngularMomentum / (Mass \* startingRelativeAbsPosition);

if (StartingRelativeAbsVelocitySquared <= Periapsis)

{

startingPerpendicularVelocity += startingPerpendicularVelocity / 400000;

}

else if (StartingRelativeAbsVelocitySquared >= Apoapsis)

{

startingPerpendicularVelocity -= startingPerpendicularVelocity / 400000;

}

return startingPerpendicularVelocity;

}

double StartingPerpendicularVelocity = CalculateStartingPerpendicularVelocity();

Vector3 CalculateStartingRelativeVelocity()

{

Vector3 startingRelativeVelocity = new Vector3(Convert.ToSingle(StartingPerpendicularVelocity), Convert.ToSingle(Math.Sqrt(Math.Pow(StartingPerpendicularVelocity, 2) - StartingRelativeAbsVelocitySquared)), 0);

if (StartingTrueAnomaly > Math.PI || StartingTrueAnomaly < 0)

{

startingRelativeVelocity \*= -1;

}

return startingRelativeVelocity;

}

Vector3 StartingRelativeVelocity = RotateVector3OntoTheOrbitalPath(CalculateStartingRelativeVelocity());

currentPoint = new PointInformation(StartingTimeFromEpoch, double.NaN, double.NaN, startingRelativePosition, StartingRelativeVelocity);

// This is the amount of time taken to complete an orbit in seconds.

decimal orbitalPeriod;

// 200000 data points was chosen as a good trade between memory and performance.

predeterminedPoints = CreatePoints(new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum , EccentricityVector, TotalEnergy, HillSphereRadius, 0), StartingTimeFromEpoch, 200000, ParentMass, startingRelativeAbsPosition, out orbitalPeriod);

orbitInformation = new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius, orbitalPeriod);

return false;

}

private bool ReinitialiseOrbitConstantsFromKeplerianElements(double ParentMass, double StartingTrueAnomaly, double LongitudeOfAscendingNode, double ArgumentOfPeriapsis, double LongitudeOfPeriapsis, double Inclination, double Eccentricity, double SemiMajorAxis, double SemiMinorAxis, double Periapsis, double Apoapsis, double AngularMomentum, Vector3 SpecificAngularMomentum, Vector3 EccentricityVector, double TotalEnergy, double HillSphereRadius)

{

double startingRelativeAbsPosition = (SemiMajorAxis \* (1 - Math.Pow(Eccentricity, 2))) / (1 + Eccentricity \* Math.Cos(StartingTrueAnomaly));

// This is the amount of time taken to complete an orbit in seconds.

decimal orbitalPeriod;

predeterminedPoints = CreatePoints(new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius, 0), StartingTimeFromEpoch, 200000, ParentMass, startingRelativeAbsPosition, out orbitalPeriod);

orbitInformation = new OrbitInformation(LongitudeOfAscendingNode, ArgumentOfPeriapsis, LongitudeOfPeriapsis, Inclination, Eccentricity, SemiMajorAxis, SemiMinorAxis, Periapsis, Apoapsis, AngularMomentum, SpecificAngularMomentum, EccentricityVector, TotalEnergy, HillSphereRadius, orbitalPeriod);

return false;

}

private PointInformation[] CreatePoints(OrbitInformation OrbitInformation, decimal TimeOffset, int NumberOfIntervals, double ParentMass, double StartingDistance, out decimal OrbitalPeriod)

{

double SpecificTotalEnergy = OrbitInformation.TotalEnergy / Mass;

// This is derived using -2 \* V \* U / K = 0 (bare in mind U is the only important part as it contains AbsSpecificAngularMomentum).

// This is because of the "time" equation used later, which requires virtually 0 error in A, breaks when using the derivation from A = Angular Momentum / Mass due to the error being increased.

// The area that breaks is Math.Sqrt(-2 \* V \* U / K) hence needs to equal basically 0 as to stretch the time equation's bounds to be as close as possible to correct.

double AbsSpecificAngularMomentum = Math.Sqrt(2 \* SpecificTotalEnergy \* Math.Pow(OrbitInformation.Periapsis, 2) + 2 \* G \* ParentMass \* OrbitInformation.Periapsis);

if (AbsSpecificAngularMomentum > 0)

{

AbsSpecificAngularMomentum -= AbsSpecificAngularMomentum \* Math.Pow(10, -10);

}

else

{

AbsSpecificAngularMomentum += AbsSpecificAngularMomentum \* Math.Pow(10, -10);

}

// K is a constant used in the following equations.

double K = Math.Pow(G \* ParentMass, 2) + 2 \* SpecificTotalEnergy \* Math.Pow(AbsSpecificAngularMomentum, 2);

// U is a variable determined by r; MidpointAdjustedU is hence a variable determined by midpointDistance.

double MidpointAdjustedU = 2 \* SpecificTotalEnergy \* Math.Pow(OrbitInformation.SemiMajorAxis, 2) + 2 \* G \* ParentMass \* OrbitInformation.SemiMajorAxis - Math.Pow(AbsSpecificAngularMomentum, 2);

// MidpointTimeAdjustment is the time added to the second equation for time too adjust for the discontinuity in the graphs.

decimal MidpointTimeAdjustment = Convert.ToDecimal(-2 \* G \* ParentMass / (4 \* SpecificTotalEnergy) \* (Math.Sqrt(-2 / SpecificTotalEnergy) \* Math.Asin(Math.Round(Math.Sqrt(-2 \* SpecificTotalEnergy \* MidpointAdjustedU / K), 5))));

// OrbitalPeriod is the total time taken to complete one orbit in seconds.

OrbitalPeriod = Convert.ToDecimal(2 \* Math.PI \* Math.Sqrt(Math.Pow(OrbitInformation.SemiMajorAxis, 3) / (G \* ParentMass)));

// Given that the position the body starts at is not the apoapsis, there is an offset in the time from the natural time given from starting at apoapsis.

decimal startPositionTimeOffset = 0;

// If at the extremes errors occur right on those edges. This means a small offset is given to keep the edgecases from throwing errors.

if (StartingDistance <= OrbitInformation.Periapsis)

{

CalculatePoint(StartingDistance + StartingDistance / 400000, startPositionTimeOffset, OrbitalPeriod, true, out startPositionTimeOffset);

}

else if (StartingDistance >= OrbitInformation.Apoapsis)

{

CalculatePoint(StartingDistance - StartingDistance / 400000, startPositionTimeOffset, OrbitalPeriod, true, out startPositionTimeOffset);

}

else

{

CalculatePoint(StartingDistance, startPositionTimeOffset, OrbitalPeriod, true, out startPositionTimeOffset);

}

PointInformation[] predeterminedPoints = new PointInformation[NumberOfIntervals];

for (int i = 0; i < NumberOfIntervals; i++)

{

// Points lie both above and below the semi-major axis line, hence a positive and negative case need to be created.

double TrueAnomaly = i \* 2 \* Math.PI / NumberOfIntervals;

double RelativeAbsPosition = (OrbitInformation.SemiMajorAxis \* (1 - Math.Pow(OrbitInformation.Eccentricity, 2))) / (1 + OrbitInformation.Eccentricity \* Math.Cos(TrueAnomaly));

Vector3 relativePosition = new Vector3(Convert.ToSingle(RelativeAbsPosition \* Math.Cos(TrueAnomaly)), Convert.ToSingle(RelativeAbsPosition \* Math.Sin(TrueAnomaly)), 0);

decimal time;

bool CalculateIfPositive()

{

if (TrueAnomaly > Math.PI)

{

return false;

}

else

{

return true;

}

}

bool positive = CalculateIfPositive();

CalculatePoint(RelativeAbsPosition, startPositionTimeOffset, OrbitalPeriod, positive, out time);

double AngleOfVelocityToTangent = Math.Acos(OrbitInformation.AngularMomentum / Math.Sqrt(2 \* OrbitInformation.TotalEnergy \* Mass \* Math.Pow(RelativeAbsPosition, 2) + 2 \* G \* ParentMass \* Math.Pow(Mass, 2) \* RelativeAbsPosition));

double PerpendicularVelocity = OrbitInformation.AngularMomentum / (Mass \* RelativeAbsPosition);

double TotalVelocitySquared = 2 \* OrbitInformation.TotalEnergy / Mass + (2 \* G \* ParentMass) / RelativeAbsPosition;

Vector3 relativeVelocity = new Vector3(Convert.ToSingle(PerpendicularVelocity), Convert.ToSingle(Math.Sqrt(TotalVelocitySquared - Math.Pow(PerpendicularVelocity, 2))), 0);

RotatePoint(ref relativePosition, ref relativeVelocity);

predeterminedPoints[i] = new PointInformation(time, AngleOfVelocityToTangent, TrueAnomaly, relativePosition, relativeVelocity);

}

// This allows the efficient way of calculating which point the body is currently at to work.

predeterminedPoints = predeterminedPoints.OrderBy(new Func<PointInformation, decimal>(x => x.Time)).ToArray();

return predeterminedPoints;

void CalculatePoint(double DistanceToParent, decimal StartPositionTimeOffset, decimal OrbitalPeriod, bool Positive, out decimal time)

{

// U is a constant used in the following equations.

double U = 2 \* SpecificTotalEnergy \* DistanceToParent \* DistanceToParent + 2 \* G \* ParentMass \* DistanceToParent - AbsSpecificAngularMomentum \* AbsSpecificAngularMomentum;

if (DistanceToParent < OrbitInformation.SemiMajorAxis && Positive)

{

time = (Convert.ToDecimal(-G \* ParentMass / (4 \* SpecificTotalEnergy) \* (Math.Sqrt(-2 / SpecificTotalEnergy) \* Math.Asin(Math.Round(Math.Sqrt(-2 \* SpecificTotalEnergy \* U / K), 5))) + 1 / (2 \* SpecificTotalEnergy) \* Math.Sqrt(U)) + TimeOffset + OrbitalPeriod - StartPositionTimeOffset) % OrbitalPeriod;

}

else if (DistanceToParent > OrbitInformation.SemiMajorAxis && Positive)

{

time = (Convert.ToDecimal(G \* ParentMass / (4 \* SpecificTotalEnergy) \* (Math.Sqrt(-2 / SpecificTotalEnergy) \* Math.Asin(Math.Round(Math.Sqrt(-2 \* SpecificTotalEnergy \* U / K), 5))) + 1 / (2 \* SpecificTotalEnergy) \* Math.Sqrt(U)) + MidpointTimeAdjustment + TimeOffset + OrbitalPeriod - StartPositionTimeOffset) % OrbitalPeriod;

}

else if (DistanceToParent < OrbitInformation.SemiMajorAxis && !Positive)

{

time = OrbitalPeriod - (Convert.ToDecimal(-G \* ParentMass / (4 \* SpecificTotalEnergy) \* (Math.Sqrt(-2 / SpecificTotalEnergy) \* Math.Asin(Math.Round(Math.Sqrt(-2 \* SpecificTotalEnergy \* U / K), 5))) + 1 / (2 \* SpecificTotalEnergy) \* Math.Sqrt(U)) + TimeOffset + OrbitalPeriod + StartPositionTimeOffset) % OrbitalPeriod;

}

else if (DistanceToParent > OrbitInformation.SemiMajorAxis && !Positive)

{

time = OrbitalPeriod - (Convert.ToDecimal(G \* ParentMass / (4 \* SpecificTotalEnergy) \* (Math.Sqrt(-2 / SpecificTotalEnergy) \* Math.Asin(Math.Round(Math.Sqrt(-2 \* SpecificTotalEnergy \* U / K), 5))) + 1 / (2 \* SpecificTotalEnergy) \* Math.Sqrt(U)) + MidpointTimeAdjustment + TimeOffset + OrbitalPeriod + StartPositionTimeOffset) % OrbitalPeriod;

}

else

{

throw new Exception("Variable 'time' has failed to be defined.");

}

}

void RotatePoint(ref Vector3 relativePosition, ref Vector3 relativeVelocity)

{

// This rotates the orbit such that its periapsis is at the correct angle to the +x direction.

TransformationMatrix3x3 rotationMatrixForPeriapsis = new TransformationMatrix3x3(new double[,] { { Math.Cos(OrbitInformation.ArgumentOfPeriapsis), Math.Sin(OrbitInformation.ArgumentOfPeriapsis), 0 }, { -Math.Sin(OrbitInformation.ArgumentOfPeriapsis), Math.Cos(OrbitInformation.ArgumentOfPeriapsis), 0 }, { 0, 0, 1 } });

relativePosition = rotationMatrixForPeriapsis.Transform(relativePosition);

relativeVelocity = rotationMatrixForPeriapsis.Transform(relativeVelocity);

// This rotates the orbit to be the correct inclination.

TransformationMatrix3x3 rotationMatrixForInclination = new TransformationMatrix3x3(new double[,] { { 1, 0, 0 }, { 0, Math.Cos(OrbitInformation.Inclination), Math.Sin(OrbitInformation.Inclination) }, { 0, -Math.Sin(OrbitInformation.Inclination), Math.Cos(OrbitInformation.Inclination) } });

relativePosition = rotationMatrixForInclination.Transform(relativePosition);

relativeVelocity = rotationMatrixForInclination.Transform(relativeVelocity);

// This rotates the orbit such that the ascending node is at the correct angle to the +x direction.

TransformationMatrix3x3 rotationMatrixForAscendingNode = new TransformationMatrix3x3(new double[,] { { Math.Cos(OrbitInformation.ArgumentOfPeriapsis), Math.Sin(OrbitInformation.ArgumentOfPeriapsis), 0 }, { -Math.Sin(OrbitInformation.ArgumentOfPeriapsis), Math.Cos(OrbitInformation.ArgumentOfPeriapsis), 0 }, { 0, 0, 1 } });

relativePosition = rotationMatrixForAscendingNode.Transform(relativePosition);

relativeVelocity = rotationMatrixForAscendingNode.Transform(relativeVelocity);

}

}

private int currentIndex { get; set; } = 0;

private ulong timeWraps = 0;

public void UpdateCurrentPoint(decimal time)

{

if ((ulong)time / (ulong)orbitInformation.OrbitalPeriod != timeWraps)

{

timeWraps = (ulong)time / (ulong)orbitInformation.OrbitalPeriod;

currentIndex = 0;

return;

}

time = (time - StartingTimeFromEpoch) % orbitInformation.OrbitalPeriod;

while (time > predeterminedPoints[(currentIndex + 1) % (predeterminedPoints.Length - 1)].Time)

{

currentIndex++;

}

currentPoint = predeterminedPoints[currentIndex];

}

public void UpdateCurrentPointUsingArraySearch(decimal time)

{

decimal[] predeterminedAdjustedTimes = new decimal[predeterminedPoints.Count()];

if ((ulong)time / (ulong)orbitInformation.OrbitalPeriod != timeWraps)

{

timeWraps = (ulong)time / (ulong)orbitInformation.OrbitalPeriod;

}

time = (time - StartingTimeFromEpoch) % orbitInformation.OrbitalPeriod;

int count = 0;

foreach (PointInformation point in predeterminedPoints)

{

predeterminedAdjustedTimes[count] = Math.Abs(point.Time - time);

count++;

}

currentIndex = Array.IndexOf(predeterminedAdjustedTimes, predeterminedAdjustedTimes.Min());

currentPoint = predeterminedPoints[currentIndex];

}

public string ReturnEphemeris()

{

MessageBox.Show("This may take up to a few hours as 200000 data points are having to be converted into a human readable form.");

string output = "Time, Angle of Velocity to Tangent, True Anomaly, Relative Position X, Relative Position Y, Relative Position Z, Relative Velocity X, Relative Velocity Y, Relative Velocity Z\n";

foreach (PointInformation point in predeterminedPoints)

{

output += point.Time + ",";

output += point.AngleOfVelocityToTangent + ",";

output += point.TrueAnomaly + ",";

output += point.RelativePosition.ToString().Substring(1, output.Length - 2) + ",";

output += point.RelativeVelocity.ToString().Substring(1, output.Length - 2) + "\n";

}

return output;

}

// Instead of using commas, a comma and pipe symbol together are used to stop ambiguity in places such as converting a vector into a string.

public override string ToString()

{

string output = "";

output += Name + ",|";

output += Mass.ToString() + ",|";

output += Radius.ToString() + ",|";

output += Colour.ToArgb().ToString() + ",|";

output += StartingTimeFromEpoch.ToString() + ",|";

output += StartingTrueAnomaly.ToString() + ",|";

output += orbitInformation.ToString() + "\n";

output = output.Replace("<", "");

output = output.Replace(">", "");

return output;

}

}

}

## OrbitInformation.cs

using System.Numerics;

namespace \_3D\_Orbital\_Motion\_Simulation

{

// This is defined independantly to any classes or structs as it is used in a multitude of locations.

internal record OrbitInformation(double LongitudeOfAscendingNode, double ArgumentOfPeriapsis, double LongitudeOfPeriapsis, double Inclination, double Eccentricity, double SemiMajorAxis, double SemiMinorAxis, double Periapsis, double Apoapsis, double AngularMomentum, Vector3 SpecificAngularMomentum, Vector3 EccentricityVector, double TotalEnergy, double HillSphereRadius, decimal OrbitalPeriod);

}

## NotEnabledException.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

[Serializable]

internal class NotEnabledException : Exception

{

// I decided to make this a custom exception as a future coder may find it useful to be able to handle specifically the tree not being enabled and not a generic exception in the code.

// It stops exception handling from dealing with 'too much'.

public NotEnabledException() : base("This tree is currently disabled. Try adding a new reference body.") { }

}

}

## Matrix3x3.cs

using System.Numerics;

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal struct TransformationMatrix3x3

{

private double[,] Contents { get; set; }

public TransformationMatrix3x3(double[,] Contents)

{

if (Contents.GetLength(0) != 3 || Contents.GetLength(1) != 3)

{

throw new Exception("Incorrect dimensions inputted when creating the matrix.");

}

this.Contents = Contents;

}

public Vector3 Transform(Vector3 Point)

{

Vector3 transformedPoint;

transformedPoint.X = Convert.ToSingle(Point.X \* Contents[0, 0] + Point.Y \* Contents[1, 0] + Point.Z \* Contents[2, 0]);

transformedPoint.Y = Convert.ToSingle(Point.X \* Contents[0, 1] + Point.Y \* Contents[1, 1] + Point.Z \* Contents[2, 1]);

transformedPoint.Z = Convert.ToSingle(Point.X \* Contents[0, 2] + Point.Y \* Contents[1, 2] + Point.Z \* Contents[2, 2]);

return transformedPoint;

}

}

}

## AddBodyForm.cs

using System.Numerics;

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal partial class AddBodyForm : Form

{

// The main form is passed into this form as it is a class.

// This means it is reference typed and not copied, allowing access to tree to always be up to date and not a one time copy.

private MainForm mainForm { get; init; }

public AddBodyForm(MainForm mainForm)

{

InitializeComponent();

this.mainForm = mainForm;

}

// I use the colour picker dialog to allow a user to input any colour they wish.

// To display their chosen colour, I use a label the user is unable to click on as it is both quicker and simpler than using the graphics functions.

private void colourPickButton\_Click(object sender, EventArgs e)

{

ColorDialog myDialog = new ColorDialog();

myDialog.AllowFullOpen = true;

myDialog.ShowHelp = true;

myDialog.Color = colourBox.BackColor;

if (myDialog.ShowDialog() == DialogResult.OK)

{

colourBox.BackColor = myDialog.Color;

}

}

// This sets the form to be designed for inputting a body defined by state vectors.

private void stateVectorsSelectionButton\_Click(object sender, EventArgs e)

{

sharedLabel1.Location = new Point(43, 214);

sharedLabel2.Location = new Point(43, 243);

sharedLabel3.Location = new Point(43, 272);

sharedLabel4.Location = new Point(45, 301);

sharedLabel5.Location = new Point(45, 330);

sharedLabel6.Location = new Point(45, 359);

extraLineLabel.Visible = false;

sharedLabel1.Text = "Position X:";

sharedLabel2.Text = "Position Y:";

sharedLabel3.Text = "Position Z:";

sharedLabel4.Text = "Velocity X:";

sharedLabel5.Text = "Velocity Y:";

sharedLabel6.Text = "Velocity Z:";

parentBodyInputTextBox.Visible = false;

parentBodyLabel.Visible = false;

relativeToParentOrBackgroundVelocityGroupBox.Visible = true;

Size = new Size(292, 474);

}

// This sets the form to be designed for inputting a body defined by keplerian elements.

private void keplerianElementsSelectionButton\_Click(object sender, EventArgs e)

{

sharedLabel1.Location = new Point(53, 214);

sharedLabel2.Location = new Point(39, 243);

sharedLabel3.Location = new Point(65, 272);

sharedLabel4.Location = new Point(70, 301);

sharedLabel5.Location = new Point(9, 330);

sharedLabel6.Location = new Point(39, 349);

extraLineLabel.Visible = true;

sharedLabel1.Text = "True Anomaly:";

sharedLabel2.Text = "Semi-major Axis:";

sharedLabel3.Text = "Eccentricity:";

sharedLabel4.Text = "Inclination:";

sharedLabel5.Text = "Argument of Periapsis:";

sharedLabel6.Text = "Longitude of";

parentBodyInputTextBox.Visible = true;

parentBodyLabel.Visible = true;

relativeToParentOrBackgroundVelocityGroupBox.Visible = false;

Size = new Size(292, 455);

}

// As there are many number input boxes, it made the code much easier to read and write by making a standard function for handling their data.

private double handleNumberInputs(TextBox BaseNumber, TextBox PowerOfTen)

{

return Convert.ToDouble(BaseNumber.Text) \* Math.Pow(10, Convert.ToDouble(PowerOfTen.Text));

}

private void createBodyButton\_Click(object sender, EventArgs e)

{

try

{

double Mass = handleNumberInputs(massInputTextBox, powerOfTenInputTextBoxMass);

double Radius = handleNumberInputs(radiusInputTextBox, powerOfTenInputTextBoxRadius);

double SharedInput1 = handleNumberInputs(sharedInputTextBox1, powerOfTenInputTextBox1);

double SharedInput2 = handleNumberInputs(sharedInputTextBox2, powerOfTenInputTextBox2);

double SharedInput3 = handleNumberInputs(sharedInputTextBox3, powerOfTenInputTextBox3);

if (mainForm.tree.IsEnabled())

{

double SharedInput4 = handleNumberInputs(sharedInputTextBox4, powerOfTenInputTextBox4);

double SharedInput5 = handleNumberInputs(sharedInputTextBox5, powerOfTenInputTextBox5);

double SharedInput6 = handleNumberInputs(sharedInputTextBox6, powerOfTenInputTextBox6);

if (keplerianElementsSelectionButton.Checked)

{

mainForm.tree.AddToTree(parentBodyInputTextBox.Text, colourBox.BackColor, nameInputTextBox.Text, Mass, mainForm.time, SharedInput1, SharedInput2, SharedInput3, SharedInput4, SharedInput5, SharedInput6, Radius);

}

else if (stateVectorsSelectionButton.Checked)

{

mainForm.tree.AddToTree(colourBox.BackColor, nameInputTextBox.Text, Mass, mainForm.time, new Vector3(Convert.ToSingle(SharedInput4), Convert.ToSingle(SharedInput5), Convert.ToSingle(SharedInput6)), new Vector3(Convert.ToSingle(SharedInput1), Convert.ToSingle(SharedInput2), Convert.ToSingle(SharedInput3)), Radius, relativeToBackgroundVelocityButton.Checked);

}

}

else

{

mainForm.tree.Reenable(new FixedBody(colourBox.BackColor, nameInputTextBox.Text, new Vector3(Convert.ToSingle(SharedInput1), Convert.ToSingle(SharedInput2), Convert.ToSingle(SharedInput3)), Mass, Radius));

}

}

catch (Exception)

{

MessageBox.Show("Failed to create body with the given parameters.");

}

}

// The form has to continuously check whether the tree is enabled so that as soon as a user deletes the reference body they aren't able to break the tree structure by attempting to add a moving body as the reference body.

private void checkIfTreeEnabledTimer\_Tick(object sender, EventArgs e)

{

if (!mainForm.tree.IsEnabled())

{

sharedLabel1.Location = new Point(43, 214);

sharedLabel2.Location = new Point(43, 243);

sharedLabel3.Location = new Point(43, 272);

sharedLabel1.Text = "Position X:";

sharedLabel2.Text = "Position Y:";

sharedLabel3.Text = "Position Z:";

if (keplerianElementsSelectionButton.Checked)

{

keplerianElementsSelectionButton.Checked = false;

createBodyButton.Text = "Create Reference Body";

sharedInputTextBox1.Text = "0";

sharedInputTextBox2.Text = "0";

sharedInputTextBox3.Text = "0";

powerOfTenInputTextBox1.Text = "0";

powerOfTenInputTextBox2.Text = "0";

powerOfTenInputTextBox3.Text = "0";

}

movingBodyDefinitionTypeGroupBox.Visible = false;

Size = new Size(292, 338);

}

else

{

if (!movingBodyDefinitionTypeGroupBox.Visible)

{

movingBodyDefinitionTypeGroupBox.Visible = true;

keplerianElementsSelectionButton.Checked = true;

keplerianElementsSelectionButton\_Click(sender, e);

createBodyButton.Text = "Create Body";

sharedInputTextBox1.Text = "";

sharedInputTextBox2.Text = "";

sharedInputTextBox3.Text = "";

sharedInputTextBox4.Text = "";

sharedInputTextBox5.Text = "";

sharedInputTextBox6.Text = "";

powerOfTenInputTextBox1.Text = "";

powerOfTenInputTextBox2.Text = "";

powerOfTenInputTextBox3.Text = "";

powerOfTenInputTextBox4.Text = "";

powerOfTenInputTextBox5.Text = "";

powerOfTenInputTextBox6.Text = "";

}

}

}

// Input validation for all standard form number's decimal part.

private void standardFormNumberInputTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes an arbitrary limit of 100 characters maximum allowed in the text box to stop string character limit overflows.

if (((TextBox)sender).Text.Length >= 100 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Only allows inputs of numbers, the '.' and '-' character as well as backspace and the delete key.

else if ((e.KeyChar > 47 && e.KeyChar < 58) || e.KeyChar == 45 || e.KeyChar == 8 || e.KeyChar == 127)

{

e.Handled = false;

}

else if (e.KeyChar == 46)

{

// If there is already a decimal point, don't allow another.

if (((TextBox)sender).Text.Contains('.'))

{

e.Handled = true;

}

else

{

e.Handled = false;

}

}

else if (e.KeyChar == 45)

{

// If there is already a minus, don't allow another.

if (((TextBox)sender).Text.Contains('-'))

{

e.Handled = true;

}

else

{

e.Handled = false;

}

}

else

{

e.Handled = true;

}

}

// Input validation for all standard form number's power of ten part.

private void powerOfTenInputTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes a maximum length of 2 characters given the largest possible float is of the order of magnitude of 10^38 and 38 is 2 characters long.

if (((TextBox)sender).Text.Length >= 2 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Only allows inputs of numbers as well as backspace and the delete key.

else if ((e.KeyChar > 47 && e.KeyChar < 58) || e.KeyChar == 45 || e.KeyChar == 8 || e.KeyChar == 127)

{

e.Handled = false;

}

else

{

e.Handled = true;

}

}

// Input validation for the name of the added or parent body.

private void nameInputTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes a basically arbitrary maximum length derived from the maximum number of characters for a name to still be displayed in full on the informationForm.

if (((TextBox)sender).Text.Length >= 17 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Allows all regular characters and punctuation, space, and backspace.

else if ((e.KeyChar > 31 && e.KeyChar < 128) || e.KeyChar == 8)

{

e.Handled = false;

}

else

{

e.Handled = true;

}

}

}

}

## ControllerForm.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

public partial class ControllerForm : Form

{

// The main form is passed into this form as it is a class.

// This means it is reference typed and not copied, allowing access to tree to always be up to date and not a one time copy.

// It also gives access to time and secondsPerSecond; these two variables are both displayed and changed by this form.

private MainForm mainForm { get; init; }

public ControllerForm(MainForm mainForm)

{

InitializeComponent();

this.mainForm = mainForm;

}

private void secondsPerSecondTrackbar\_Scroll(object sender, EventArgs e)

{

void SetSecondsPerSecondFromTrackbarValue(ulong secondsPerSecond, string labelText, int labelXPosition)

{

mainForm.secondsPerSecond = secondsPerSecond;

secondsPerSecondLabel.Text = labelText;

secondsPerSecondLabel.Location = new Point(labelXPosition, 55);

}

// It made a lot more sense to have fixed and human time intervals than allowing a psuedo continuous trackbar, so predefined lengths of time were chosed to correspond to the trackbar's discrete values.

switch (secondsPerSecondTrackbar.Value)

{

case 1:

SetSecondsPerSecondFromTrackbarValue(1, "1 second /s", 47);

break;

case 2:

SetSecondsPerSecondFromTrackbarValue(60, "1 minute /s", 47);

break;

case 3:

SetSecondsPerSecondFromTrackbarValue(3600, "1 hour /s", 54);

break;

case 4:

SetSecondsPerSecondFromTrackbarValue(86400, "1 day /s", 56);

break;

case 5:

SetSecondsPerSecondFromTrackbarValue(604800, "1 week /s", 52);

break;

case 6:

SetSecondsPerSecondFromTrackbarValue(2678400, "31 days /s", 50);

break;

case 7:

SetSecondsPerSecondFromTrackbarValue(15778800, "0.5 years /s", 49);

break;

case 8:

SetSecondsPerSecondFromTrackbarValue(31557600, "1 year /s", 55);

break;

case 9:

SetSecondsPerSecondFromTrackbarValue(315576000, "10 years /s", 49);

break;

case 10:

SetSecondsPerSecondFromTrackbarValue(3155760000, "100 years /s", 46);

break;

}

}

private void pausePlayButton\_Click(object sender, EventArgs e)

{

if (pausePlayButton.Text == "Pause")

{

pausePlayButton.Text = "Play";

secondsPerSecondTrackbar.Enabled = false;

mainForm.secondsPerSecond = 0;

}

else

{

secondsPerSecondTrackbar.Enabled = true;

pausePlayButton.Text = "Pause";

secondsPerSecondTrackbar\_Scroll(sender, e);

}

}

private void updateCurrentTimeTimer\_Tick(object sender, EventArgs e)

{

if (pausePlayButton.Text == "Pause")

{

// This is rounded as it makes it more readable to a human without lots of unnecesary decimal places.

currentTimeTextBox.Text = Convert.ToString(Math.Round(mainForm.time, 2));

}

}

private void setCurrentTimeButton\_Click(object sender, EventArgs e)

{

if (pausePlayButton.Text == "Play")

{

mainForm.time = Convert.ToDecimal(currentTimeTextBox.Text);

mainForm.tree.UpdateBodiesCurrentPositionsUsingArraySearch(Convert.ToDecimal(currentTimeTextBox.Text));

}

}

// Input validation for changing the current time.

private void currentTimeTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes an arbitrary limit of 100 characters maximum allowed in the text box to stop string character limit overflows.

if (((TextBox)sender).Text.Length >= 100 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Only allows inputs of numbers, the '.' character as well as backspace and the delete key.

else if ((e.KeyChar > 47 && e.KeyChar < 58) || e.KeyChar == 8 || e.KeyChar == 127)

{

e.Handled = false;

}

else if (e.KeyChar == 46)

{

// If there is already a decimal point, don't allow another.

if (((TextBox)sender).Text.Contains('.'))

{

e.Handled = true;

}

else

{

e.Handled = false;

}

}

else

{

e.Handled = true;

}

}

}

}

## ExportBodyEphemerisForm.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal partial class ExportBodysEphemerisForm : Form

{

// The main form is passed into this form as it is a class.

// This means it is reference typed and not copied, allowing access to tree to always be up to date and not a one time copy.

MainForm mainForm { get; init; }

public ExportBodysEphemerisForm(MainForm mainForm)

{

InitializeComponent();

this.mainForm = mainForm;

}

// Input validation for the name of the removed body.

private void pickBodyInputTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes a maximum length of the removed body name given the maximum length name an added body can have.

if (((TextBox)sender).Text.Length >= 17 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Allows all regular characters and punctuation, space, and backspace.

else if ((e.KeyChar > 31 && e.KeyChar < 128) || e.KeyChar == 8)

{

e.Handled = false;

}

else

{

e.Handled = true;

}

}

private void exportBodysEphemerisButton\_Click(object sender, EventArgs e)

{

if (mainForm.tree.IsEnabled() == true)

{

mainForm.tree.ExportBodysEphemeris(pickBodyInputTextBox.Text.ToLower());

}

else

{

MessageBox.Show("There are currently no bodies in the simulation.");

}

}

}

}

## InformationForm.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal partial class InformationForm : Form

{

internal InformationForm()

{

InitializeComponent();

}

internal void SetLabels(string name, OrbitInformation orbitInformation)

{

NameLabel.Text = "Name: " + name;

LongitudeOfAscendingNodeLabel.Text = "Longitude Of Ascending Node: " + Math.Round(orbitInformation.LongitudeOfAscendingNode, 3) + " Radians";

ArgumentOfPeriapsisLabel.Text = "Argument Of Periapsis: " + Math.Round(orbitInformation.ArgumentOfPeriapsis, 3) + " Radians";

LongitudeOfPeriapsisLabel.Text = "Longitude Of Periapsis: " + Math.Round(orbitInformation.LongitudeOfPeriapsis, 3) + " Radians";

InclinationLabel.Text = "Inclination: " + Math.Round(orbitInformation.Inclination, 3) + " Radians";

EccentricityLabel.Text = "Eccentricity: " + Math.Round(orbitInformation.Eccentricity, 3) + " Radians";

SemiMajorAxisLabel.Text = "Semi Major Axis: " + Math.Round(orbitInformation.SemiMajorAxis) + " Metres";

SemiMinorAxisLabel.Text = "Semi Minor Axis: " + Math.Round(orbitInformation.SemiMinorAxis) + " Metres";

PeriapsisLabel.Text = "Periapsis: " + Math.Round(orbitInformation.Periapsis) + " Metres";

ApoapsisLabel.Text = "Apoapsis: " + Math.Round(orbitInformation.Apoapsis) + " Metres";

AngularMomentumLabel.Text = "Angular Momentum: " + Math.Round(orbitInformation.AngularMomentum) + " KgM^2/s";

TotalEnergyLabel.Text = "Total Energy: " + Math.Round(orbitInformation.TotalEnergy) + " Joules";

OrbitalPeriodLabel.Text = "Orbital Period: " + Math.Round(orbitInformation.OrbitalPeriod) + " Seconds" ;

HillSphereRadiusLabel.Text = "Hill Sphere Radius: " + Math.Round(orbitInformation.HillSphereRadius) + " Metres";

}

}

}

## RemoveBodyForm.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

internal partial class RemoveBodyForm : Form

{

// The main form is passed into this form as it is a class.

// This means it is reference typed and not copied, allowing access to tree to always be up to date and not a one time copy.

MainForm mainForm { get; init; }

public RemoveBodyForm(MainForm mainForm)

{

InitializeComponent();

this.mainForm = mainForm;

}

// Input validation for the name of the removed body.

private void removeBodyInputTextBox\_KeyPress(object sender, KeyPressEventArgs e)

{

// Imposes a maximum length of the removed body name given the maximum length name an added body can have.

if (((TextBox)sender).Text.Length >= 17 && e.KeyChar != 8 && e.KeyChar != 127)

{

e.Handled = true;

}

// Allows all regular characters and punctuation, space, and backspace.

else if ((e.KeyChar > 31 && e.KeyChar < 128) || e.KeyChar == 8)

{

e.Handled = false;

}

else

{

e.Handled = true;

}

}

private void removeBodyButton\_Click(object sender, EventArgs e)

{

if (mainForm.tree.IsEnabled() == true)

{

mainForm.tree.RemoveFromTree(removeBodyInputTextBox.Text.ToLower());

}

else

{

MessageBox.Show("There are currently no bodies in the simulation.");

}

}

}

}

## AddBodyForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class AddBodyForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

components = new System.ComponentModel.Container();

colourPickButton = new Button();

colourBox = new Label();

nameLabel = new Label();

nameInputTextBox = new TextBox();

massInputTextBox = new TextBox();

massLabel = new Label();

radiusInputTextBox = new TextBox();

radiusLabel = new Label();

sharedInputTextBox2 = new TextBox();

sharedLabel2 = new Label();

sharedInputTextBox1 = new TextBox();

sharedLabel1 = new Label();

sharedInputTextBox5 = new TextBox();

sharedLabel5 = new Label();

sharedInputTextBox4 = new TextBox();

sharedLabel4 = new Label();

sharedInputTextBox3 = new TextBox();

sharedLabel3 = new Label();

sharedInputTextBox6 = new TextBox();

sharedLabel6 = new Label();

movingBodyDefinitionTypeGroupBox = new GroupBox();

stateVectorsSelectionButton = new RadioButton();

keplerianElementsSelectionButton = new RadioButton();

createBodyButton = new Button();

extraLineLabel = new Label();

parentBodyInputTextBox = new TextBox();

parentBodyLabel = new Label();

checkIfTreeEnabledTimer = new System.Windows.Forms.Timer(components);

powerOfTenInputTextBox1 = new TextBox();

powerOfTenInputTextBox2 = new TextBox();

powerOfTenInputTextBox3 = new TextBox();

powerOfTenInputTextBox4 = new TextBox();

powerOfTenInputTextBox5 = new TextBox();

powerOfTenInputTextBox6 = new TextBox();

powerOfTenLabel1 = new Label();

powerOfTenLabel2 = new Label();

powerOfTenLabel3 = new Label();

powerOfTenLabel4 = new Label();

powerOfTenLabel5 = new Label();

powerOfTenLabel6 = new Label();

powerOfTenLabelMass = new Label();

powerOfTenLabelRadius = new Label();

powerOfTenInputTextBoxMass = new TextBox();

powerOfTenInputTextBoxRadius = new TextBox();

relativeToParentOrBackgroundVelocityGroupBox = new GroupBox();

relativeToBackgroundVelocityButton = new RadioButton();

relativeToParentButton = new RadioButton();

movingBodyDefinitionTypeGroupBox.SuspendLayout();

relativeToParentOrBackgroundVelocityGroupBox.SuspendLayout();

SuspendLayout();

//

// colourPickButton

//

colourPickButton.Location = new Point(12, 42);

colourPickButton.Name = "colourPickButton";

colourPickButton.Size = new Size(52, 23);

colourPickButton.TabIndex = 0;

colourPickButton.Text = "Colour";

colourPickButton.UseVisualStyleBackColor = true;

colourPickButton.Click += colourPickButton\_Click;

//

// colourBox

//

colourBox.AutoSize = true;

colourBox.BackColor = Color.White;

colourBox.BorderStyle = BorderStyle.Fixed3D;

colourBox.Location = new Point(70, 46);

colourBox.Name = "colourBox";

colourBox.Size = new Size(18, 17);

colourBox.TabIndex = 1;

colourBox.Text = " ";

//

// nameLabel

//

nameLabel.AutoSize = true;

nameLabel.BackColor = Color.Transparent;

nameLabel.Location = new Point(15, 79);

nameLabel.Name = "nameLabel";

nameLabel.Size = new Size(42, 15);

nameLabel.TabIndex = 2;

nameLabel.Text = "Name:";

//

// nameInputTextBox

//

nameInputTextBox.Location = new Point(63, 74);

nameInputTextBox.Name = "nameInputTextBox";

nameInputTextBox.Size = new Size(136, 23);

nameInputTextBox.TabIndex = 3;

nameInputTextBox.KeyPress += nameInputTextBox\_KeyPress;

//

// massInputTextBox

//

massInputTextBox.Location = new Point(63, 103);

massInputTextBox.Name = "massInputTextBox";

massInputTextBox.Size = new Size(136, 23);

massInputTextBox.TabIndex = 5;

massInputTextBox.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// massLabel

//

massLabel.AutoSize = true;

massLabel.BackColor = Color.Transparent;

massLabel.Location = new Point(20, 106);

massLabel.Name = "massLabel";

massLabel.Size = new Size(37, 15);

massLabel.TabIndex = 4;

massLabel.Text = "Mass:";

//

// radiusInputTextBox

//

radiusInputTextBox.Location = new Point(63, 132);

radiusInputTextBox.Name = "radiusInputTextBox";

radiusInputTextBox.Size = new Size(136, 23);

radiusInputTextBox.TabIndex = 7;

radiusInputTextBox.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// radiusLabel

//

radiusLabel.AutoSize = true;

radiusLabel.BackColor = Color.Transparent;

radiusLabel.Location = new Point(12, 135);

radiusLabel.Name = "radiusLabel";

radiusLabel.Size = new Size(45, 15);

radiusLabel.TabIndex = 6;

radiusLabel.Text = "Radius:";

//

// sharedInputTextBox2

//

sharedInputTextBox2.Location = new Point(134, 240);

sharedInputTextBox2.Name = "sharedInputTextBox2";

sharedInputTextBox2.Size = new Size(65, 23);

sharedInputTextBox2.TabIndex = 13;

sharedInputTextBox2.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel2

//

sharedLabel2.AutoSize = true;

sharedLabel2.BackColor = Color.Transparent;

sharedLabel2.Location = new Point(39, 243);

sharedLabel2.Name = "sharedLabel2";

sharedLabel2.Size = new Size(97, 15);

sharedLabel2.TabIndex = 12;

sharedLabel2.Text = "Semi-major Axis:";

//

// sharedInputTextBox1

//

sharedInputTextBox1.Location = new Point(134, 211);

sharedInputTextBox1.Name = "sharedInputTextBox1";

sharedInputTextBox1.Size = new Size(65, 23);

sharedInputTextBox1.TabIndex = 11;

sharedInputTextBox1.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel1

//

sharedLabel1.AutoSize = true;

sharedLabel1.BackColor = Color.Transparent;

sharedLabel1.Location = new Point(53, 214);

sharedLabel1.Name = "sharedLabel1";

sharedLabel1.Size = new Size(83, 15);

sharedLabel1.TabIndex = 10;

sharedLabel1.Text = "True Anomaly:";

//

// sharedInputTextBox5

//

sharedInputTextBox5.Location = new Point(134, 327);

sharedInputTextBox5.Name = "sharedInputTextBox5";

sharedInputTextBox5.Size = new Size(65, 23);

sharedInputTextBox5.TabIndex = 19;

sharedInputTextBox5.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel5

//

sharedLabel5.AutoSize = true;

sharedLabel5.BackColor = Color.Transparent;

sharedLabel5.Location = new Point(9, 330);

sharedLabel5.Name = "sharedLabel5";

sharedLabel5.Size = new Size(127, 15);

sharedLabel5.TabIndex = 18;

sharedLabel5.Text = "Argument of Periapsis:";

//

// sharedInputTextBox4

//

sharedInputTextBox4.Location = new Point(134, 298);

sharedInputTextBox4.Name = "sharedInputTextBox4";

sharedInputTextBox4.Size = new Size(65, 23);

sharedInputTextBox4.TabIndex = 17;

sharedInputTextBox4.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel4

//

sharedLabel4.AutoSize = true;

sharedLabel4.BackColor = Color.Transparent;

sharedLabel4.Location = new Point(70, 301);

sharedLabel4.Name = "sharedLabel4";

sharedLabel4.Size = new Size(66, 15);

sharedLabel4.TabIndex = 16;

sharedLabel4.Text = "Inclination:";

//

// sharedInputTextBox3

//

sharedInputTextBox3.Location = new Point(134, 269);

sharedInputTextBox3.Name = "sharedInputTextBox3";

sharedInputTextBox3.Size = new Size(65, 23);

sharedInputTextBox3.TabIndex = 15;

sharedInputTextBox3.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel3

//

sharedLabel3.AutoSize = true;

sharedLabel3.BackColor = Color.Transparent;

sharedLabel3.Location = new Point(65, 272);

sharedLabel3.Name = "sharedLabel3";

sharedLabel3.Size = new Size(71, 15);

sharedLabel3.TabIndex = 14;

sharedLabel3.Text = "Eccentricity:";

//

// sharedInputTextBox6

//

sharedInputTextBox6.Location = new Point(134, 356);

sharedInputTextBox6.Name = "sharedInputTextBox6";

sharedInputTextBox6.Size = new Size(65, 23);

sharedInputTextBox6.TabIndex = 21;

sharedInputTextBox6.KeyPress += standardFormNumberInputTextBox\_KeyPress;

//

// sharedLabel6

//

sharedLabel6.AutoSize = true;

sharedLabel6.BackColor = Color.Transparent;

sharedLabel6.Location = new Point(39, 349);

sharedLabel6.Name = "sharedLabel6";

sharedLabel6.Size = new Size(75, 15);

sharedLabel6.TabIndex = 20;

sharedLabel6.Text = "Longitude of";

//

// movingBodyDefinitionTypeGroupBox

//

movingBodyDefinitionTypeGroupBox.Controls.Add(stateVectorsSelectionButton);

movingBodyDefinitionTypeGroupBox.Controls.Add(keplerianElementsSelectionButton);

movingBodyDefinitionTypeGroupBox.Location = new Point(15, 161);

movingBodyDefinitionTypeGroupBox.Name = "movingBodyDefinitionTypeGroupBox";

movingBodyDefinitionTypeGroupBox.Size = new Size(250, 42);

movingBodyDefinitionTypeGroupBox.TabIndex = 22;

movingBodyDefinitionTypeGroupBox.TabStop = false;

movingBodyDefinitionTypeGroupBox.Text = "Definition Type";

//

// stateVectorsSelectionButton

//

stateVectorsSelectionButton.AutoSize = true;

stateVectorsSelectionButton.Location = new Point(137, 17);

stateVectorsSelectionButton.Name = "stateVectorsSelectionButton";

stateVectorsSelectionButton.Size = new Size(92, 19);

stateVectorsSelectionButton.TabIndex = 1;

stateVectorsSelectionButton.Text = "State Vectors";

stateVectorsSelectionButton.UseVisualStyleBackColor = true;

stateVectorsSelectionButton.Click += stateVectorsSelectionButton\_Click;

//

// keplerianElementsSelectionButton

//

keplerianElementsSelectionButton.AutoSize = true;

keplerianElementsSelectionButton.Checked = true;

keplerianElementsSelectionButton.Location = new Point(6, 17);

keplerianElementsSelectionButton.Name = "keplerianElementsSelectionButton";

keplerianElementsSelectionButton.Size = new Size(125, 19);

keplerianElementsSelectionButton.TabIndex = 0;

keplerianElementsSelectionButton.TabStop = true;

keplerianElementsSelectionButton.Text = "Keplerian Elements";

keplerianElementsSelectionButton.UseVisualStyleBackColor = true;

keplerianElementsSelectionButton.Click += keplerianElementsSelectionButton\_Click;

//

// createBodyButton

//

createBodyButton.Location = new Point(12, 12);

createBodyButton.Name = "createBodyButton";

createBodyButton.Size = new Size(253, 24);

createBodyButton.TabIndex = 23;

createBodyButton.Text = "Create Body";

createBodyButton.UseVisualStyleBackColor = true;

createBodyButton.Click += createBodyButton\_Click;

//

// extraLineLabel

//

extraLineLabel.AutoSize = true;

extraLineLabel.Location = new Point(18, 364);

extraLineLabel.Name = "extraLineLabel";

extraLineLabel.Size = new Size(118, 15);

extraLineLabel.TabIndex = 24;

extraLineLabel.Text = "the Ascending Node:";

//

// parentBodyInputTextBox

//

parentBodyInputTextBox.Location = new Point(133, 385);

parentBodyInputTextBox.Name = "parentBodyInputTextBox";

parentBodyInputTextBox.Size = new Size(132, 23);

parentBodyInputTextBox.TabIndex = 26;

parentBodyInputTextBox.KeyPress += nameInputTextBox\_KeyPress;

//

// parentBodyLabel

//

parentBodyLabel.AutoSize = true;

parentBodyLabel.BackColor = Color.Transparent;

parentBodyLabel.Location = new Point(27, 388);

parentBodyLabel.Name = "parentBodyLabel";

parentBodyLabel.Size = new Size(109, 15);

parentBodyLabel.TabIndex = 25;

parentBodyLabel.Text = "Parent Body Name:";

//

// checkIfTreeEnabledTimer

//

checkIfTreeEnabledTimer.Enabled = true;

checkIfTreeEnabledTimer.Tick += checkIfTreeEnabledTimer\_Tick;

//

// powerOfTenInputTextBox1

//

powerOfTenInputTextBox1.Location = new Point(241, 211);

powerOfTenInputTextBox1.Name = "powerOfTenInputTextBox1";

powerOfTenInputTextBox1.Size = new Size(24, 23);

powerOfTenInputTextBox1.TabIndex = 27;

powerOfTenInputTextBox1.Text = "0";

powerOfTenInputTextBox1.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox1.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBox2

//

powerOfTenInputTextBox2.Location = new Point(241, 240);

powerOfTenInputTextBox2.Name = "powerOfTenInputTextBox2";

powerOfTenInputTextBox2.Size = new Size(24, 23);

powerOfTenInputTextBox2.TabIndex = 28;

powerOfTenInputTextBox2.Text = "0";

powerOfTenInputTextBox2.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox2.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBox3

//

powerOfTenInputTextBox3.Location = new Point(241, 269);

powerOfTenInputTextBox3.Name = "powerOfTenInputTextBox3";

powerOfTenInputTextBox3.Size = new Size(24, 23);

powerOfTenInputTextBox3.TabIndex = 29;

powerOfTenInputTextBox3.Text = "0";

powerOfTenInputTextBox3.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox3.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBox4

//

powerOfTenInputTextBox4.Location = new Point(241, 298);

powerOfTenInputTextBox4.Name = "powerOfTenInputTextBox4";

powerOfTenInputTextBox4.Size = new Size(24, 23);

powerOfTenInputTextBox4.TabIndex = 30;

powerOfTenInputTextBox4.Text = "0";

powerOfTenInputTextBox4.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox4.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBox5

//

powerOfTenInputTextBox5.Location = new Point(241, 327);

powerOfTenInputTextBox5.Name = "powerOfTenInputTextBox5";

powerOfTenInputTextBox5.Size = new Size(24, 23);

powerOfTenInputTextBox5.TabIndex = 31;

powerOfTenInputTextBox5.Text = "0";

powerOfTenInputTextBox5.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox5.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBox6

//

powerOfTenInputTextBox6.Location = new Point(241, 356);

powerOfTenInputTextBox6.Name = "powerOfTenInputTextBox6";

powerOfTenInputTextBox6.Size = new Size(24, 23);

powerOfTenInputTextBox6.TabIndex = 32;

powerOfTenInputTextBox6.Text = "0";

powerOfTenInputTextBox6.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBox6.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenLabel1

//

powerOfTenLabel1.AutoSize = true;

powerOfTenLabel1.BackColor = Color.Transparent;

powerOfTenLabel1.Location = new Point(205, 214);

powerOfTenLabel1.Name = "powerOfTenLabel1";

powerOfTenLabel1.Size = new Size(33, 15);

powerOfTenLabel1.TabIndex = 33;

powerOfTenLabel1.Text = "x10^";

//

// powerOfTenLabel2

//

powerOfTenLabel2.AutoSize = true;

powerOfTenLabel2.BackColor = Color.Transparent;

powerOfTenLabel2.Location = new Point(205, 243);

powerOfTenLabel2.Name = "powerOfTenLabel2";

powerOfTenLabel2.Size = new Size(33, 15);

powerOfTenLabel2.TabIndex = 34;

powerOfTenLabel2.Text = "x10^";

//

// powerOfTenLabel3

//

powerOfTenLabel3.AutoSize = true;

powerOfTenLabel3.BackColor = Color.Transparent;

powerOfTenLabel3.Location = new Point(205, 272);

powerOfTenLabel3.Name = "powerOfTenLabel3";

powerOfTenLabel3.Size = new Size(33, 15);

powerOfTenLabel3.TabIndex = 35;

powerOfTenLabel3.Text = "x10^";

//

// powerOfTenLabel4

//

powerOfTenLabel4.AutoSize = true;

powerOfTenLabel4.BackColor = Color.Transparent;

powerOfTenLabel4.Location = new Point(205, 301);

powerOfTenLabel4.Name = "powerOfTenLabel4";

powerOfTenLabel4.Size = new Size(33, 15);

powerOfTenLabel4.TabIndex = 36;

powerOfTenLabel4.Text = "x10^";

//

// powerOfTenLabel5

//

powerOfTenLabel5.AutoSize = true;

powerOfTenLabel5.BackColor = Color.Transparent;

powerOfTenLabel5.Location = new Point(205, 330);

powerOfTenLabel5.Name = "powerOfTenLabel5";

powerOfTenLabel5.Size = new Size(33, 15);

powerOfTenLabel5.TabIndex = 37;

powerOfTenLabel5.Text = "x10^";

//

// powerOfTenLabel6

//

powerOfTenLabel6.AutoSize = true;

powerOfTenLabel6.BackColor = Color.Transparent;

powerOfTenLabel6.Location = new Point(205, 359);

powerOfTenLabel6.Name = "powerOfTenLabel6";

powerOfTenLabel6.Size = new Size(33, 15);

powerOfTenLabel6.TabIndex = 38;

powerOfTenLabel6.Text = "x10^";

//

// powerOfTenLabelMass

//

powerOfTenLabelMass.AutoSize = true;

powerOfTenLabelMass.BackColor = Color.Transparent;

powerOfTenLabelMass.Location = new Point(205, 106);

powerOfTenLabelMass.Name = "powerOfTenLabelMass";

powerOfTenLabelMass.Size = new Size(33, 15);

powerOfTenLabelMass.TabIndex = 39;

powerOfTenLabelMass.Text = "x10^";

//

// powerOfTenLabelRadius

//

powerOfTenLabelRadius.AutoSize = true;

powerOfTenLabelRadius.BackColor = Color.Transparent;

powerOfTenLabelRadius.Location = new Point(205, 135);

powerOfTenLabelRadius.Name = "powerOfTenLabelRadius";

powerOfTenLabelRadius.Size = new Size(33, 15);

powerOfTenLabelRadius.TabIndex = 40;

powerOfTenLabelRadius.Text = "x10^";

//

// powerOfTenInputTextBoxMass

//

powerOfTenInputTextBoxMass.Location = new Point(241, 103);

powerOfTenInputTextBoxMass.Name = "powerOfTenInputTextBoxMass";

powerOfTenInputTextBoxMass.Size = new Size(24, 23);

powerOfTenInputTextBoxMass.TabIndex = 41;

powerOfTenInputTextBoxMass.Text = "0";

powerOfTenInputTextBoxMass.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBoxMass.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// powerOfTenInputTextBoxRadius

//

powerOfTenInputTextBoxRadius.Location = new Point(241, 132);

powerOfTenInputTextBoxRadius.Name = "powerOfTenInputTextBoxRadius";

powerOfTenInputTextBoxRadius.Size = new Size(24, 23);

powerOfTenInputTextBoxRadius.TabIndex = 42;

powerOfTenInputTextBoxRadius.Text = "0";

powerOfTenInputTextBoxRadius.TextAlign = HorizontalAlignment.Center;

powerOfTenInputTextBoxRadius.KeyPress += powerOfTenInputTextBox\_KeyPress;

//

// relativeToParentOrBackgroundVelocityGroupBox

//

relativeToParentOrBackgroundVelocityGroupBox.Controls.Add(relativeToBackgroundVelocityButton);

relativeToParentOrBackgroundVelocityGroupBox.Controls.Add(relativeToParentButton);

relativeToParentOrBackgroundVelocityGroupBox.Location = new Point(15, 385);

relativeToParentOrBackgroundVelocityGroupBox.Name = "relativeToParentOrBackgroundVelocityGroupBox";

relativeToParentOrBackgroundVelocityGroupBox.Size = new Size(250, 42);

relativeToParentOrBackgroundVelocityGroupBox.TabIndex = 23;

relativeToParentOrBackgroundVelocityGroupBox.TabStop = false;

relativeToParentOrBackgroundVelocityGroupBox.Text = "Velocity Relative to Parent or Background";

relativeToParentOrBackgroundVelocityGroupBox.Visible = false;

//

// relativeToBackgroundVelocityButton

//

relativeToBackgroundVelocityButton.AutoSize = true;

relativeToBackgroundVelocityButton.Location = new Point(134, 17);

relativeToBackgroundVelocityButton.Name = "relativeToBackgroundVelocityButton";

relativeToBackgroundVelocityButton.Size = new Size(89, 19);

relativeToBackgroundVelocityButton.TabIndex = 1;

relativeToBackgroundVelocityButton.Text = "Background";

relativeToBackgroundVelocityButton.UseVisualStyleBackColor = true;

//

// relativeToParentButton

//

relativeToParentButton.AutoSize = true;

relativeToParentButton.Checked = true;

relativeToParentButton.Location = new Point(34, 17);

relativeToParentButton.Name = "relativeToParentButton";

relativeToParentButton.Size = new Size(62, 19);

relativeToParentButton.TabIndex = 0;

relativeToParentButton.TabStop = true;

relativeToParentButton.Text = "Parent ";

relativeToParentButton.UseVisualStyleBackColor = true;

//

// AddBodyForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

ClientSize = new Size(276, 416);

Controls.Add(relativeToParentOrBackgroundVelocityGroupBox);

Controls.Add(powerOfTenInputTextBoxRadius);

Controls.Add(powerOfTenInputTextBoxMass);

Controls.Add(powerOfTenLabelRadius);

Controls.Add(powerOfTenLabelMass);

Controls.Add(powerOfTenLabel6);

Controls.Add(powerOfTenLabel5);

Controls.Add(powerOfTenLabel4);

Controls.Add(powerOfTenLabel3);

Controls.Add(powerOfTenLabel2);

Controls.Add(powerOfTenLabel1);

Controls.Add(powerOfTenInputTextBox6);

Controls.Add(powerOfTenInputTextBox5);

Controls.Add(powerOfTenInputTextBox4);

Controls.Add(powerOfTenInputTextBox3);

Controls.Add(powerOfTenInputTextBox2);

Controls.Add(powerOfTenInputTextBox1);

Controls.Add(parentBodyInputTextBox);

Controls.Add(parentBodyLabel);

Controls.Add(extraLineLabel);

Controls.Add(createBodyButton);

Controls.Add(movingBodyDefinitionTypeGroupBox);

Controls.Add(sharedInputTextBox6);

Controls.Add(sharedLabel6);

Controls.Add(sharedInputTextBox5);

Controls.Add(sharedLabel5);

Controls.Add(sharedInputTextBox4);

Controls.Add(sharedLabel4);

Controls.Add(sharedInputTextBox3);

Controls.Add(sharedLabel3);

Controls.Add(sharedInputTextBox2);

Controls.Add(sharedLabel2);

Controls.Add(sharedInputTextBox1);

Controls.Add(sharedLabel1);

Controls.Add(radiusInputTextBox);

Controls.Add(radiusLabel);

Controls.Add(massInputTextBox);

Controls.Add(massLabel);

Controls.Add(nameInputTextBox);

Controls.Add(nameLabel);

Controls.Add(colourBox);

Controls.Add(colourPickButton);

FormBorderStyle = FormBorderStyle.FixedToolWindow;

Margin = new Padding(2);

Name = "AddBodyForm";

Text = "Add a Body";

movingBodyDefinitionTypeGroupBox.ResumeLayout(false);

movingBodyDefinitionTypeGroupBox.PerformLayout();

relativeToParentOrBackgroundVelocityGroupBox.ResumeLayout(false);

relativeToParentOrBackgroundVelocityGroupBox.PerformLayout();

ResumeLayout(false);

PerformLayout();

}

#endregion

private Button colourPickButton;

private Label colourBox;

private Label nameLabel;

private TextBox nameInputTextBox;

private TextBox massInputTextBox;

private Label massLabel;

private TextBox radiusInputTextBox;

private Label radiusLabel;

private TextBox sharedInputTextBox2;

private Label sharedLabel2;

private TextBox sharedInputTextBox1;

private Label sharedLabel1;

private TextBox sharedInputTextBox5;

private Label sharedLabel5;

private TextBox sharedInputTextBox4;

private Label sharedLabel4;

private TextBox sharedInputTextBox3;

private Label sharedLabel3;

private TextBox sharedInputTextBox6;

private Label sharedLabel6;

private GroupBox movingBodyDefinitionTypeGroupBox;

private RadioButton stateVectorsSelectionButton;

private RadioButton keplerianElementsSelectionButton;

private Button createBodyButton;

private Label extraLineLabel;

private TextBox parentBodyInputTextBox;

private Label parentBodyLabel;

private System.Windows.Forms.Timer checkIfTreeEnabledTimer;

private TextBox powerOfTenInputTextBox1;

private TextBox powerOfTenInputTextBox2;

private TextBox powerOfTenInputTextBox3;

private TextBox powerOfTenInputTextBox4;

private TextBox powerOfTenInputTextBox5;

private TextBox powerOfTenInputTextBox6;

private Label powerOfTenLabel1;

private Label powerOfTenLabel2;

private Label powerOfTenLabel3;

private Label powerOfTenLabel4;

private Label powerOfTenLabel5;

private Label powerOfTenLabel6;

private Label powerOfTenLabelMass;

private Label powerOfTenLabelRadius;

private TextBox powerOfTenInputTextBoxMass;

private TextBox powerOfTenInputTextBoxRadius;

private GroupBox relativeToParentOrBackgroundVelocityGroupBox;

private RadioButton relativeToBackgroundVelocityButton;

private RadioButton relativeToParentButton;

}

}

## ControllerForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class ControllerForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

components = new System.ComponentModel.Container();

pausePlayButton = new Button();

clockSpeedTitleLabel = new Label();

secondsPerSecondTrackbar = new TrackBar();

secondsPerSecondLabel = new Label();

currentTimeLabel = new Label();

currentTimeTextBox = new TextBox();

setCurrentTimeButton = new Button();

updateCurrentTimeTimer = new System.Windows.Forms.Timer(components);

((System.ComponentModel.ISupportInitialize)secondsPerSecondTrackbar).BeginInit();

SuspendLayout();

//

// pausePlayButton

//

pausePlayButton.Location = new Point(8, 72);

pausePlayButton.Margin = new Padding(2);

pausePlayButton.Name = "pausePlayButton";

pausePlayButton.Size = new Size(146, 25);

pausePlayButton.TabIndex = 0;

pausePlayButton.Text = "Pause";

pausePlayButton.UseVisualStyleBackColor = true;

pausePlayButton.Click += pausePlayButton\_Click;

//

// clockSpeedTitleLabel

//

clockSpeedTitleLabel.AutoSize = true;

clockSpeedTitleLabel.Font = new Font("Segoe UI", 9F, FontStyle.Underline, GraphicsUnit.Point);

clockSpeedTitleLabel.Location = new Point(45, 8);

clockSpeedTitleLabel.Margin = new Padding(2, 0, 2, 0);

clockSpeedTitleLabel.Name = "clockSpeedTitleLabel";

clockSpeedTitleLabel.Size = new Size(72, 15);

clockSpeedTitleLabel.TabIndex = 1;

clockSpeedTitleLabel.Text = "Clock Speed";

//

// secondsPerSecondTrackbar

//

secondsPerSecondTrackbar.LargeChange = 1;

secondsPerSecondTrackbar.Location = new Point(8, 25);

secondsPerSecondTrackbar.Margin = new Padding(2);

secondsPerSecondTrackbar.Minimum = 1;

secondsPerSecondTrackbar.Name = "secondsPerSecondTrackbar";

secondsPerSecondTrackbar.Size = new Size(146, 45);

secondsPerSecondTrackbar.TabIndex = 2;

secondsPerSecondTrackbar.Value = 1;

secondsPerSecondTrackbar.Scroll += secondsPerSecondTrackbar\_Scroll;

//

// secondsPerSecondLabel

//

secondsPerSecondLabel.AutoSize = true;

secondsPerSecondLabel.Location = new Point(47, 55);

secondsPerSecondLabel.Name = "secondsPerSecondLabel";

secondsPerSecondLabel.Size = new Size(67, 15);

secondsPerSecondLabel.TabIndex = 4;

secondsPerSecondLabel.Text = "1 second /s";

//

// currentTimeLabel

//

currentTimeLabel.AutoSize = true;

currentTimeLabel.Font = new Font("Segoe UI", 9F, FontStyle.Underline, GraphicsUnit.Point);

currentTimeLabel.Location = new Point(43, 110);

currentTimeLabel.Name = "currentTimeLabel";

currentTimeLabel.Size = new Size(76, 15);

currentTimeLabel.TabIndex = 5;

currentTimeLabel.Text = "Current Time";

//

// currentTimeTextBox

//

currentTimeTextBox.Location = new Point(8, 128);

currentTimeTextBox.Name = "currentTimeTextBox";

currentTimeTextBox.Size = new Size(146, 23);

currentTimeTextBox.TabIndex = 6;

currentTimeTextBox.KeyPress += currentTimeTextBox\_KeyPress;

//

// setCurrentTimeButton

//

setCurrentTimeButton.Location = new Point(8, 157);

setCurrentTimeButton.Name = "setCurrentTimeButton";

setCurrentTimeButton.Size = new Size(146, 23);

setCurrentTimeButton.TabIndex = 7;

setCurrentTimeButton.Text = "Set Time";

setCurrentTimeButton.UseVisualStyleBackColor = true;

setCurrentTimeButton.Click += setCurrentTimeButton\_Click;

//

// updateCurrentTimeTimer

//

updateCurrentTimeTimer.Enabled = true;

updateCurrentTimeTimer.Tick += updateCurrentTimeTimer\_Tick;

//

// ControllerForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

ClientSize = new Size(163, 190);

Controls.Add(setCurrentTimeButton);

Controls.Add(currentTimeTextBox);

Controls.Add(currentTimeLabel);

Controls.Add(secondsPerSecondLabel);

Controls.Add(secondsPerSecondTrackbar);

Controls.Add(clockSpeedTitleLabel);

Controls.Add(pausePlayButton);

FormBorderStyle = FormBorderStyle.FixedToolWindow;

Margin = new Padding(2);

Name = "ControllerForm";

Text = "Simulation Settings";

((System.ComponentModel.ISupportInitialize)secondsPerSecondTrackbar).EndInit();

ResumeLayout(false);

PerformLayout();

}

#endregion

private Button pausePlayButton;

private Label clockSpeedTitleLabel;

private TrackBar secondsPerSecondTrackbar;

private Label secondsPerSecondLabel;

private Label currentTimeLabel;

private TextBox currentTimeTextBox;

private Button setCurrentTimeButton;

private System.Windows.Forms.Timer updateCurrentTimeTimer;

}

}

## ExportBodysEphemerisForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class ExportBodysEphemerisForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

exportBodysEphemerisButton = new Button();

pickBodyInputTextBox = new TextBox();

removeBodyPromptLabel = new Label();

warningLabel1 = new Label();

warningLabel2 = new Label();

warningLabel3 = new Label();

SuspendLayout();

//

// exportBodysEphemerisButton

//

exportBodysEphemerisButton.Location = new Point(9, 95);

exportBodysEphemerisButton.Margin = new Padding(2);

exportBodysEphemerisButton.Name = "exportBodysEphemerisButton";

exportBodysEphemerisButton.Size = new Size(163, 26);

exportBodysEphemerisButton.TabIndex = 0;

exportBodysEphemerisButton.Text = "Export Body's Ephemeris";

exportBodysEphemerisButton.UseVisualStyleBackColor = true;

exportBodysEphemerisButton.Click += exportBodysEphemerisButton\_Click;

//

// pickBodyInputTextBox

//

pickBodyInputTextBox.Location = new Point(9, 22);

pickBodyInputTextBox.Margin = new Padding(2);

pickBodyInputTextBox.Name = "pickBodyInputTextBox";

pickBodyInputTextBox.Size = new Size(163, 23);

pickBodyInputTextBox.TabIndex = 1;

pickBodyInputTextBox.KeyPress += pickBodyInputTextBox\_KeyPress;

//

// removeBodyPromptLabel

//

removeBodyPromptLabel.AutoSize = true;

removeBodyPromptLabel.Location = new Point(15, 5);

removeBodyPromptLabel.Margin = new Padding(2, 0, 2, 0);

removeBodyPromptLabel.Name = "removeBodyPromptLabel";

removeBodyPromptLabel.Size = new Size(153, 15);

removeBodyPromptLabel.TabIndex = 2;

removeBodyPromptLabel.Text = "Please enter a body's name:";

//

// warningLabel1

//

warningLabel1.AutoSize = true;

warningLabel1.Font = new Font("Segoe UI", 9F, FontStyle.Bold, GraphicsUnit.Point);

warningLabel1.ForeColor = Color.Red;

warningLabel1.Location = new Point(16, 47);

warningLabel1.Margin = new Padding(2, 0, 2, 0);

warningLabel1.Name = "warningLabel1";

warningLabel1.Size = new Size(149, 15);

warningLabel1.TabIndex = 3;

warningLabel1.Text = "PLEASE SAVE BEFORE USE";

//

// warningLabel2

//

warningLabel2.AutoSize = true;

warningLabel2.Font = new Font("Segoe UI", 9F, FontStyle.Bold, GraphicsUnit.Point);

warningLabel2.ForeColor = Color.Red;

warningLabel2.Location = new Point(23, 62);

warningLabel2.Margin = new Padding(2, 0, 2, 0);

warningLabel2.Name = "warningLabel2";

warningLabel2.Size = new Size(131, 15);

warningLabel2.TabIndex = 4;

warningLabel2.Text = "THIS MAY TAKE A FEW";

//

// warningLabel3

//

warningLabel3.AutoSize = true;

warningLabel3.Font = new Font("Segoe UI", 9F, FontStyle.Bold, GraphicsUnit.Point);

warningLabel3.ForeColor = Color.Red;

warningLabel3.Location = new Point(24, 77);

warningLabel3.Margin = new Padding(2, 0, 2, 0);

warningLabel3.Name = "warningLabel3";

warningLabel3.Size = new Size(130, 15);

warningLabel3.TabIndex = 5;

warningLabel3.Text = "HOURS TO COMPLETE";

//

// ExportBodysEphemerisForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

ClientSize = new Size(179, 132);

Controls.Add(warningLabel3);

Controls.Add(warningLabel2);

Controls.Add(warningLabel1);

Controls.Add(removeBodyPromptLabel);

Controls.Add(pickBodyInputTextBox);

Controls.Add(exportBodysEphemerisButton);

FormBorderStyle = FormBorderStyle.FixedToolWindow;

Margin = new Padding(2);

Name = "ExportBodysEphemeris";

Text = "Export a Body's Ephemeris";

ResumeLayout(false);

PerformLayout();

}

#endregion

private Button exportBodysEphemerisButton;

private TextBox pickBodyInputTextBox;

private Label removeBodyPromptLabel;

private Label warningLabel1;

private Label warningLabel2;

private Label warningLabel3;

}

}

## informationForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class InformationForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

LongitudeOfAscendingNodeLabel = new Label();

ArgumentOfPeriapsisLabel = new Label();

LongitudeOfPeriapsisLabel = new Label();

InclinationLabel = new Label();

EccentricityLabel = new Label();

SemiMajorAxisLabel = new Label();

SemiMinorAxisLabel = new Label();

PeriapsisLabel = new Label();

ApoapsisLabel = new Label();

AngularMomentumLabel = new Label();

TotalEnergyLabel = new Label();

OrbitalPeriodLabel = new Label();

HillSphereRadiusLabel = new Label();

NameLabel = new Label();

SuspendLayout();

//

// LongitudeOfAscendingNodeLabel

//

LongitudeOfAscendingNodeLabel.AutoSize = true;

LongitudeOfAscendingNodeLabel.Font = new Font("Segoe UI", 9F, FontStyle.Regular, GraphicsUnit.Point);

LongitudeOfAscendingNodeLabel.ForeColor = Color.White;

LongitudeOfAscendingNodeLabel.Location = new Point(12, 19);

LongitudeOfAscendingNodeLabel.Name = "LongitudeOfAscendingNodeLabel";

LongitudeOfAscendingNodeLabel.Size = new Size(171, 15);

LongitudeOfAscendingNodeLabel.TabIndex = 0;

LongitudeOfAscendingNodeLabel.Text = "Longitude Of Ascending Node:";

//

// ArgumentOfPeriapsisLabel

//

ArgumentOfPeriapsisLabel.AutoSize = true;

ArgumentOfPeriapsisLabel.Font = new Font("Segoe UI", 9F, FontStyle.Regular, GraphicsUnit.Point);

ArgumentOfPeriapsisLabel.ForeColor = Color.White;

ArgumentOfPeriapsisLabel.Location = new Point(54, 34);

ArgumentOfPeriapsisLabel.Name = "ArgumentOfPeriapsisLabel";

ArgumentOfPeriapsisLabel.Size = new Size(129, 15);

ArgumentOfPeriapsisLabel.TabIndex = 1;

ArgumentOfPeriapsisLabel.Text = "Argument Of Periapsis:";

//

// LongitudeOfPeriapsisLabel

//

LongitudeOfPeriapsisLabel.AutoSize = true;

LongitudeOfPeriapsisLabel.ForeColor = Color.White;

LongitudeOfPeriapsisLabel.Location = new Point(54, 49);

LongitudeOfPeriapsisLabel.Name = "LongitudeOfPeriapsisLabel";

LongitudeOfPeriapsisLabel.Size = new Size(129, 15);

LongitudeOfPeriapsisLabel.TabIndex = 2;

LongitudeOfPeriapsisLabel.Text = "Longitude Of Periapsis:";

//

// InclinationLabel

//

InclinationLabel.AutoSize = true;

InclinationLabel.ForeColor = Color.White;

InclinationLabel.Location = new Point(117, 64);

InclinationLabel.Name = "InclinationLabel";

InclinationLabel.Size = new Size(66, 15);

InclinationLabel.TabIndex = 3;

InclinationLabel.Text = "Inclination:";

//

// EccentricityLabel

//

EccentricityLabel.AutoSize = true;

EccentricityLabel.ForeColor = Color.White;

EccentricityLabel.Location = new Point(112, 79);

EccentricityLabel.Name = "EccentricityLabel";

EccentricityLabel.Size = new Size(71, 15);

EccentricityLabel.TabIndex = 4;

EccentricityLabel.Text = "Eccentricity:";

//

// SemiMajorAxisLabel

//

SemiMajorAxisLabel.AutoSize = true;

SemiMajorAxisLabel.ForeColor = Color.White;

SemiMajorAxisLabel.Location = new Point(88, 94);

SemiMajorAxisLabel.Name = "SemiMajorAxisLabel";

SemiMajorAxisLabel.Size = new Size(95, 15);

SemiMajorAxisLabel.TabIndex = 5;

SemiMajorAxisLabel.Text = "Semi Major Axis:";

//

// SemiMinorAxisLabel

//

SemiMinorAxisLabel.AutoSize = true;

SemiMinorAxisLabel.ForeColor = Color.White;

SemiMinorAxisLabel.Location = new Point(87, 109);

SemiMinorAxisLabel.Name = "SemiMinorAxisLabel";

SemiMinorAxisLabel.Size = new Size(96, 15);

SemiMinorAxisLabel.TabIndex = 6;

SemiMinorAxisLabel.Text = "Semi Minor Axis:";

//

// PeriapsisLabel

//

PeriapsisLabel.AutoSize = true;

PeriapsisLabel.ForeColor = Color.White;

PeriapsisLabel.Location = new Point(127, 124);

PeriapsisLabel.Name = "PeriapsisLabel";

PeriapsisLabel.Size = new Size(56, 15);

PeriapsisLabel.TabIndex = 7;

PeriapsisLabel.Text = "Periapsis:";

//

// ApoapsisLabel

//

ApoapsisLabel.AutoSize = true;

ApoapsisLabel.ForeColor = Color.White;

ApoapsisLabel.Location = new Point(125, 139);

ApoapsisLabel.Name = "ApoapsisLabel";

ApoapsisLabel.Size = new Size(58, 15);

ApoapsisLabel.TabIndex = 8;

ApoapsisLabel.Text = "Apoapsis:";

//

// AngularMomentumLabel

//

AngularMomentumLabel.AutoSize = true;

AngularMomentumLabel.ForeColor = Color.White;

AngularMomentumLabel.Location = new Point(64, 154);

AngularMomentumLabel.Name = "AngularMomentumLabel";

AngularMomentumLabel.Size = new Size(119, 15);

AngularMomentumLabel.TabIndex = 9;

AngularMomentumLabel.Text = "Angular Momentum:";

//

// TotalEnergyLabel

//

TotalEnergyLabel.AutoSize = true;

TotalEnergyLabel.ForeColor = Color.White;

TotalEnergyLabel.Location = new Point(109, 169);

TotalEnergyLabel.Name = "TotalEnergyLabel";

TotalEnergyLabel.Size = new Size(74, 15);

TotalEnergyLabel.TabIndex = 11;

TotalEnergyLabel.Text = "Total Energy:";

//

// OrbitalPeriodLabel

//

OrbitalPeriodLabel.AutoSize = true;

OrbitalPeriodLabel.ForeColor = Color.White;

OrbitalPeriodLabel.Location = new Point(100, 184);

OrbitalPeriodLabel.Name = "OrbitalPeriodLabel";

OrbitalPeriodLabel.Size = new Size(83, 15);

OrbitalPeriodLabel.TabIndex = 12;

OrbitalPeriodLabel.Text = "Orbital Period:";

//

// HillSphereRadiusLabel

//

HillSphereRadiusLabel.AutoSize = true;

HillSphereRadiusLabel.ForeColor = Color.White;

HillSphereRadiusLabel.Location = new Point(78, 199);

HillSphereRadiusLabel.Name = "HillSphereRadiusLabel";

HillSphereRadiusLabel.Size = new Size(105, 15);

HillSphereRadiusLabel.TabIndex = 13;

HillSphereRadiusLabel.Text = "Hill Sphere Radius:";

//

// NameLabel

//

NameLabel.AutoSize = true;

NameLabel.Font = new Font("Segoe UI", 9F, FontStyle.Regular, GraphicsUnit.Point);

NameLabel.ForeColor = Color.White;

NameLabel.Location = new Point(141, 4);

NameLabel.Name = "NameLabel";

NameLabel.Size = new Size(42, 15);

NameLabel.TabIndex = 14;

NameLabel.Text = "Name:";

//

// InformationForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

BackColor = Color.DimGray;

ClientSize = new Size(374, 221);

Controls.Add(NameLabel);

Controls.Add(HillSphereRadiusLabel);

Controls.Add(OrbitalPeriodLabel);

Controls.Add(TotalEnergyLabel);

Controls.Add(AngularMomentumLabel);

Controls.Add(ApoapsisLabel);

Controls.Add(PeriapsisLabel);

Controls.Add(SemiMinorAxisLabel);

Controls.Add(SemiMajorAxisLabel);

Controls.Add(EccentricityLabel);

Controls.Add(InclinationLabel);

Controls.Add(LongitudeOfPeriapsisLabel);

Controls.Add(ArgumentOfPeriapsisLabel);

Controls.Add(LongitudeOfAscendingNodeLabel);

FormBorderStyle = FormBorderStyle.FixedToolWindow;

Name = "InformationForm";

ResumeLayout(false);

PerformLayout();

}

#endregion

private Label LongitudeOfAscendingNodeLabel;

private Label ArgumentOfPeriapsisLabel;

private Label LongitudeOfPeriapsisLabel;

private Label InclinationLabel;

private Label EccentricityLabel;

private Label SemiMajorAxisLabel;

private Label SemiMinorAxisLabel;

private Label PeriapsisLabel;

private Label ApoapsisLabel;

private Label AngularMomentumLabel;

private Label TotalEnergyLabel;

private Label OrbitalPeriodLabel;

private Label HillSphereRadiusLabel;

private Label NameLabel;

}

}

## MainForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class MainForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

components = new System.ComponentModel.Container();

simulationTimer = new System.Windows.Forms.Timer(components);

menuStrip = new MenuStrip();

fileToolStripMenuItem = new ToolStripMenuItem();

saveAsToolStripMenuItem = new ToolStripMenuItem();

loadToolStripMenuItem = new ToolStripMenuItem();

exportABodysEphemerisToolStripMenuItem = new ToolStripMenuItem();

editToolStripMenuItem = new ToolStripMenuItem();

addBodyToolStripMenuItem = new ToolStripMenuItem();

removeBodyToolStripMenuItem = new ToolStripMenuItem();

simulationControllerToolStripMenuItem = new ToolStripMenuItem();

resetViewToolStripMenuItem = new ToolStripMenuItem();

quitToolStripMenuItem = new ToolStripMenuItem();

mousePositionLabel = new Label();

menuStrip.SuspendLayout();

SuspendLayout();

//

// simulationTimer

//

simulationTimer.Enabled = true;

simulationTimer.Interval = 50;

simulationTimer.Tick += simulationTimer\_Tick;

//

// menuStrip

//

menuStrip.ImageScalingSize = new Size(24, 24);

menuStrip.Items.AddRange(new ToolStripItem[] { fileToolStripMenuItem, editToolStripMenuItem, simulationControllerToolStripMenuItem, resetViewToolStripMenuItem, quitToolStripMenuItem });

menuStrip.LayoutStyle = ToolStripLayoutStyle.HorizontalStackWithOverflow;

menuStrip.Location = new Point(0, 0);

menuStrip.Name = "menuStrip";

menuStrip.Padding = new Padding(4, 1, 0, 1);

menuStrip.Size = new Size(685, 24);

menuStrip.TabIndex = 0;

menuStrip.Text = "menuStrip";

//

// fileToolStripMenuItem

//

fileToolStripMenuItem.DropDownItems.AddRange(new ToolStripItem[] { saveAsToolStripMenuItem, loadToolStripMenuItem, exportABodysEphemerisToolStripMenuItem });

fileToolStripMenuItem.Name = "fileToolStripMenuItem";

fileToolStripMenuItem.Size = new Size(37, 22);

fileToolStripMenuItem.Text = "File";

//

// saveAsToolStripMenuItem

//

saveAsToolStripMenuItem.Name = "saveAsToolStripMenuItem";

saveAsToolStripMenuItem.Size = new Size(215, 22);

saveAsToolStripMenuItem.Text = "Save As";

saveAsToolStripMenuItem.Click += saveAsToolStripMenuItem\_Click;

//

// loadToolStripMenuItem

//

loadToolStripMenuItem.Name = "loadToolStripMenuItem";

loadToolStripMenuItem.Size = new Size(215, 22);

loadToolStripMenuItem.Text = "Load";

loadToolStripMenuItem.Click += loadToolStripMenuItem\_Click;

//

// exportABodysEphemerisToolStripMenuItem

//

exportABodysEphemerisToolStripMenuItem.Name = "exportABodysEphemerisToolStripMenuItem";

exportABodysEphemerisToolStripMenuItem.Size = new Size(215, 22);

exportABodysEphemerisToolStripMenuItem.Text = "Export A Body's Ephemeris";

exportABodysEphemerisToolStripMenuItem.Click += exportABodysEphemerisToolStripMenuItem\_Click;

//

// editToolStripMenuItem

//

editToolStripMenuItem.DropDownItems.AddRange(new ToolStripItem[] { addBodyToolStripMenuItem, removeBodyToolStripMenuItem });

editToolStripMenuItem.Name = "editToolStripMenuItem";

editToolStripMenuItem.Size = new Size(39, 22);

editToolStripMenuItem.Text = "Edit";

//

// addBodyToolStripMenuItem

//

addBodyToolStripMenuItem.Name = "addBodyToolStripMenuItem";

addBodyToolStripMenuItem.Size = new Size(147, 22);

addBodyToolStripMenuItem.Text = "Add Body";

addBodyToolStripMenuItem.Click += addBodyToolStripMenuItem\_Click;

//

// removeBodyToolStripMenuItem

//

removeBodyToolStripMenuItem.Name = "removeBodyToolStripMenuItem";

removeBodyToolStripMenuItem.Size = new Size(147, 22);

removeBodyToolStripMenuItem.Text = "Remove Body";

removeBodyToolStripMenuItem.Click += removeBodyToolStripMenuItem\_Click;

//

// simulationControllerToolStripMenuItem

//

simulationControllerToolStripMenuItem.Name = "simulationControllerToolStripMenuItem";

simulationControllerToolStripMenuItem.Size = new Size(132, 22);

simulationControllerToolStripMenuItem.Text = "Simulation Controller";

simulationControllerToolStripMenuItem.Click += simulationControllerToolStripMenuItem\_Click;

//

// resetViewToolStripMenuItem

//

resetViewToolStripMenuItem.Name = "resetViewToolStripMenuItem";

resetViewToolStripMenuItem.Size = new Size(75, 22);

resetViewToolStripMenuItem.Text = "Reset View";

resetViewToolStripMenuItem.Click += resetViewToolStripMenuItem\_Click;

//

// quitToolStripMenuItem

//

quitToolStripMenuItem.Name = "quitToolStripMenuItem";

quitToolStripMenuItem.Size = new Size(42, 22);

quitToolStripMenuItem.Text = "Quit";

quitToolStripMenuItem.Click += quitToolStripMenuItem\_Click;

//

// mousePositionLabel

//

mousePositionLabel.AutoSize = true;

mousePositionLabel.ForeColor = Color.White;

mousePositionLabel.Location = new Point(0, 24);

mousePositionLabel.Name = "mousePositionLabel";

mousePositionLabel.Size = new Size(92, 15);

mousePositionLabel.TabIndex = 1;

mousePositionLabel.Text = "Mouse Position:";

//

// MainForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

BackColor = Color.Black;

ClientSize = new Size(685, 566);

Controls.Add(mousePositionLabel);

Controls.Add(menuStrip);

DoubleBuffered = true;

FormBorderStyle = FormBorderStyle.None;

MainMenuStrip = menuStrip;

Name = "MainForm";

Text = "Orbital Motion";

WindowState = FormWindowState.Maximized;

Load += MainForm\_Load;

MouseDown += MainForm\_MouseDown;

MouseUp += MainForm\_MouseUp;

MouseWheel += MainForm\_MouseScroll;

menuStrip.ResumeLayout(false);

menuStrip.PerformLayout();

ResumeLayout(false);

PerformLayout();

}

#endregion

private System.Windows.Forms.Timer simulationTimer;

private MenuStrip menuStrip;

private ToolStripMenuItem fileToolStripMenuItem;

private ToolStripMenuItem saveAsToolStripMenuItem;

private ToolStripMenuItem loadToolStripMenuItem;

private ToolStripMenuItem simulationControllerToolStripMenuItem;

private ToolStripMenuItem editToolStripMenuItem;

private ToolStripMenuItem addBodyToolStripMenuItem;

private ToolStripMenuItem removeBodyToolStripMenuItem;

private Label mousePositionLabel;

private ToolStripMenuItem resetViewToolStripMenuItem;

private ToolStripMenuItem quitToolStripMenuItem;

private ToolStripMenuItem exportABodysEphemerisToolStripMenuItem;

}

}

## RemoveBodyForm.Designer.cs

namespace \_3D\_Orbital\_Motion\_Simulation

{

partial class RemoveBodyForm

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the Contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

removeBodyButton = new Button();

removeBodyInputTextBox = new TextBox();

removeBodyPromptLabel = new Label();

SuspendLayout();

//

// removeBodyButton

//

removeBodyButton.Location = new Point(9, 44);

removeBodyButton.Margin = new Padding(2);

removeBodyButton.Name = "removeBodyButton";

removeBodyButton.Size = new Size(163, 26);

removeBodyButton.TabIndex = 0;

removeBodyButton.Text = "Remove Body And Children";

removeBodyButton.UseVisualStyleBackColor = true;

removeBodyButton.Click += removeBodyButton\_Click;

//

// removeBodyInputTextBox

//

removeBodyInputTextBox.Location = new Point(9, 22);

removeBodyInputTextBox.Margin = new Padding(2);

removeBodyInputTextBox.Name = "removeBodyInputTextBox";

removeBodyInputTextBox.Size = new Size(163, 23);

removeBodyInputTextBox.TabIndex = 1;

removeBodyInputTextBox.KeyPress += removeBodyInputTextBox\_KeyPress;

//

// removeBodyPromptLabel

//

removeBodyPromptLabel.AutoSize = true;

removeBodyPromptLabel.Location = new Point(15, 5);

removeBodyPromptLabel.Margin = new Padding(2, 0, 2, 0);

removeBodyPromptLabel.Name = "removeBodyPromptLabel";

removeBodyPromptLabel.Size = new Size(153, 15);

removeBodyPromptLabel.TabIndex = 2;

removeBodyPromptLabel.Text = "Please enter a body's name:";

//

// RemoveBodyForm

//

AutoScaleDimensions = new SizeF(7F, 15F);

AutoScaleMode = AutoScaleMode.Font;

ClientSize = new Size(179, 81);

Controls.Add(removeBodyPromptLabel);

Controls.Add(removeBodyInputTextBox);

Controls.Add(removeBodyButton);

FormBorderStyle = FormBorderStyle.FixedToolWindow;

Margin = new Padding(2);

Name = "RemoveBodyForm";

Text = "Remove a Body";

ResumeLayout(false);

PerformLayout();

}

#endregion

private Button removeBodyButton;

private TextBox removeBodyInputTextBox;

private Label removeBodyPromptLabel;

}

}

# Testing

## Plan

My plan and final test documentation will merge, as it will be made from tables with empty results columns waiting to be filled in.

I plan to test both whether the project is complete, as well as whether it handles poor inputs that would cause errors to occur.

## Tests of Objective Completeness

To ensure objective completeness, a test shall be ran to make sure each objective has been met. The ‘test code’ refers to which objective is being tested, as it refers to the numbering used for the objectives.

Some tests will not be tests on running but instead validation of code written, e.g., 1.a.i cannot be tested, but a pointer to where in the code there is proof of implementation would suffice.

Some tests will not cover the full extend of what a point is asking if the other elements of that statement have already been proven true, e.g., 1.b.ii does not require testing that bodies don’t interact with objects bar their parents because 1.b.i proves that they do not; hence, you can only test that grandchildren are creatable as that is the only part of the statement not tested.

|  |  |
| --- | --- |
| Test Code | Description |
| 1.a.i | Does the code use the equations derived by both me and Ben? |
| Normal, Erroneous or Extreme | Test Data |
| N.A | N.A. |
| Expected Outcome | Actual Outcome |
| Yes | See Moving Body.cs  Function CreatePoint’s local function CalculatePoint |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 1.b.i | Do bodies orbit one parent body and that alone affects trajectory? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Using the default created orbits, I manually activated the paths. |
| Expected Outcome | Actual Outcome |
| Orbits should not drift over time as that would imply, so we should see a consistent orbit. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 1.b.ii | Can grandchild bodies exist? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Using the default created orbits, one with and one without paths. |
| Expected Outcome | Actual Outcome |
| The moon should exist and orbit the earth as it moves around the sun. (The moon is a grandchild to the sun.)  If the moon follows the earth, when paths are enabled the earth-moon trajectory should have both blue and grey parts showing both the Earth and Moon. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.a **1** | Can orbits be created using state vectors? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Name: New Object  Mass: 1 x10^ 1  Radius: 1 x10^ 1  Position X: 1 x10^ 10  Position Y: 1 x10^ 10  Position Z: 1 x10^ 10  Velocity X: 3 x10^ 4  Velocity Y: -7.5 x10^ 3  Velocity Z: 0 |
| Expected Outcome | Actual Outcome |
| An object to appear which will orbit the sun. |  |
| Potentially | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.a **2** | Can orbits be created using Keplerian elements? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Name: New  Mass: 1  Radius: 1  True Anomaly: 0  Semi-Major Axis: 1 x10^ 7  Eccentricity: 0.2  Inclination: 0.5  Argument of Periapsis: 0.2  Longitude of Ascending Node: 0  Parent Body Name: Mars |
| Expected Outcome | Actual Outcome |
| A body should be created, and over time with trails it should show movement. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.a.i | Can an orbit exist pushing outside of their parent’s hill sphere radius? |
| Normal, Erroneous or Extreme | Test Data |
| Erroneous | Eccentricity: 0.97  True Anomaly: 3.14 |
| Expected Outcome | Actual Outcome |
| With a high eccentricity the apoapsis is forced higher. This means this should fail. |  |
| Probable success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.a.i.A | Can a body be created where its hill sphere extends over a pre-existing body? |
| Normal, Erroneous or Extreme | Test Data |
| Erroneous | The moon’s data after the moon has slightly moved. |
| Expected Outcome | Actual Outcome |
| As the moon will overlap with the moon when placed next to each other, this should fail. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.a.ii | Is a user warned about inaccuracies when trying to make a high mass object orbit another object of high mass? |
| Normal, Erroneous or Extreme | Test Data |
| Extreme: This tests the step between being of insignificant mass to just before being too significant where the hill sphere extends over the parent. | Using any orbital parameters but the mass needs to be significantly high. |
| Expected Outcome | Actual Outcome |
| The user is warned about the inaccuracies. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.b | Can a user view orbital statistics of a selected body? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Right click on a body. |
| Expected Outcome | Actual Outcome |
| An information window should appear. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 2.b.i | Does the information window stop access to other features? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Open the information window and attempt to open and other windows. Use one of the form’s functions to show they also work. |
| Expected Outcome | Actual Outcome |
| Other windows should open and be useable. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 3 | Can a user alter the simulation speed? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | The slider in the controller form should be moved. |
| Expected Outcome | Actual Outcome |
| The simulation speed should change. This is hard to represent through screenshots, however showing a larger time displayed as a result of the time moving faster should imply that time has been sped up.  The simulation should also be pause-able which would grey out the ability to change how fast time is moving because time isn’t moving. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 3.a | After increasing the speed for a short period of time, does lowering the speed reveal inaccuracies caused by changing the time period? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Standard scenario with trails to show if any inaccuracies occur. The time needs to get far into the future by speeding up time, and then slowed much more. |
| Expected Outcome | Actual Outcome |
| The same orbits without variation. | (Note that it was left for a while on lower than a year, but I sped it up at the end because 1 year is still 100x less fast than the 100 years per second I had it running originally on.) |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 4.a | Can a user translate their viewing window? |
| Normal, Erroneous or Extreme | Test Data |
| Normal: Regular movement  Extreme: Reaching the edge of the boundary  Erroneous: Pushing outside of the boundary | Normal: Move the viewing window significantly from the centre.  Extreme: Reach the floating-point limit.  Erroneous: Exceeding the floating-point limit. |
| Expected Outcome | Actual Outcome |
| Normal: Should work fine.  Extreme: Should be reset.  Erroneous: Should be reset | Normal:  Uranus is just about visible on the edge, but the screen can be translated to not include it.  Extreme:  Erroneous: |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 4.b | Can a user translate their viewing window while the simulation updates? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Make sure the simulation is running while dragging. |
| Expected Outcome | Actual Outcome |
| It should work. | This is hard to provide proof for, but it worked. |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 5 **1** | Can a user save the simulation? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Save the standard Solar System provided. |
| Expected Outcome | Actual Outcome |
| The user should be able to save the data. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 5 **2** | Can a user load the simulation? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Remove Sol and then load the Solar System back |
| Expected Outcome | Actual Outcome |
| The solar system should come back and exist as it would have been defined for the time it currently is in the simulation. |  |
| Success | |

|  |  |
| --- | --- |
| Test Code | Description |
| 6 | Is a standard solar system available to the user? |
| Normal, Erroneous or Extreme | Test Data |
| Normal | Load the simulation. |
| Expected Outcome | Actual Outcome |
| It should be provided on loading. |  |
| Success | |

# Evaluation

## Objectives Completion

|  |  |
| --- | --- |
| Objective Code: | Personal score out of 10 in how well it was achieved: |
| 1 | 7/10 |
| 1.a | 9/10 |
| 1.a.i | 8/10 |
| 1.b | 8/10 |
| 1.b.i | 8/10 |
| 1.b.ii | 6.5/10 |
| 2 | 7/10 |
| 2.a | 7/10 |
| 2.a.i | 7/10 |
| 2.a.i.A | 5/10 |
| 2.a.ii | 7/10 |
| 2.b | 6/10 |
| 2.b.i | 8/10 |
| 3 | 6.5/10 |
| 3.a | 7/10 |
| 4 | 6.5/10 |
| 4.a | 6.5/10 |
| 4.b | 7/10 |
| 5 | 8/10 |
| 5.a | 7/10 |
| 5.b | 7/10 |
| 5.c | 8/10 |
| 6 | 7/10 |
| 6.a | 9/10 |

## Closing Discussion

Personally, I think I achieved most, if not all of the objectives that I set out to do. While the testing could certainly have been more rigorous and clearer (especially because a lot of cases were too ambiguous to definitively prove something worked or not), I still think that all features set out to be achieved have been achieved. Furthermore, there were some extra things I added on top of the objectives, such as the ability to export ephemeris data and the ability to zoom.

The system uses 3D vector and matrix mathematics to properly define static orbits, as originally set out in the objectives and by my clients. I think the UI could certainly improve as currently it is not very attractive. Furthermore, it lacks viewing the Z axis or any other viewing angle than a bird’s eye on the xy plane. If I had more time, I certainly would have pivoted from WinForms directly after the prototype and moved into something like unity, especially because the rendering would be done for me without my own personal creation of making things appear on the screen. I am however still happy about my rendering attempts and giving the user an easer drag and zoom option to view trajectories in more detail.

After speaking with Ben and Alexis after the completion of the project, they said that it was a good product to satisfy what they were asking for. They agreed that it fit most of the objective but commented also about the lack of third dimension in the viewing window. They were happy with the adhering to the limitations set out at the beginning; Ben was especially happy to see his maths work in a visual way. Since giving the product to the two, Alexis has attempted to add as many real-world bodies as possible in the solar system to push the code to it’s limits; she was happy to say the performance still help up despite that. Also, given her game development background, she has expressed a will to take the physics heavy back end and create her own front end for it, potentially allowing a 3d viewing of the whole simulation.

Overall, I think the project still has a lot more room for potential improvement and growth into a much more interactive and accurate simulation software. Despite this, I think the product is satisfactory for the direct reasons at too its existence, such as modelling the physics with more accuracy and using Ben’s equation. Hence, I think it turned out well.