Project Document

Solar System Simulator

Daniel Michaeli
909224
Tietotekniikka (year 1)
23. April 2023

General Description

The documentation describes a solar system simulator app. Assuming planar solar systems and circular orbits, as opposed to the more realistic elliptical case, the program calculates and updates new values for positions, velocities and accelerations of the celestial bodies using the laws of physics and numerical methods. Through the graphical user interface one is able to observe the movements of celestial bodies over time as they would in real life. The simulation runs until the specified time limit is reached or a collision between two bodies occur.

The program offers different options for editing the simulation as needed, and displaying different view elements for more information about the state of the bodies. The initial simulation data is read in from a .txt file, and is subsequently turned into a working simulation. While body radii are arbitrarily adjusted for maximum visibility all other values, such as masses and relative distances, can be put in as real values for actual planets in a solar system.

The topic has been implemented at the “demanding” difficulty level, which include features for adding celestial bodies mid-simulation, changing sun radii (or densities) and zooming out when a celestial body travels off the gui window.

User Interface

The program is launched through the IDE in the SolarSystemSimulatorApp file's app object. In order to launch the app a pre-specified simulation .txt file must be chosen, whose file name is written in as an argument on line 10 (domain.parseData(fileName)). The default input file is “theSolarSystem.txt”, a true-to-scale representation of our own solar system that includes the sun, the following six planets in order, and a satellite. Other preset simulation files include a slightly modified version of the previous system **solarTest0.txt,** a simulation for two of Jupiter’s Lagrangian points **JupiterLagrangePoints.txt** and two binary star systems **twoStarSystem.txt** and **twoStarSystemTwo.txt**.

Upon launch the simulation is set up and in pause-mode (assuming a properly structured input file, else an exception will be thrown). With adequately adjusted values one should see a semi-stable solar system with realistic planetary orbital motion, or perhaps a smaller body's trajectory through space affected by the gravitation of larger objects. With too extreme values the gravitational effects might be too small or large to get a comprehensive idea of what's going on (just like in real life). If a body exits the gui window the simulation reports it and zooms out to try and fit it into the picture again.

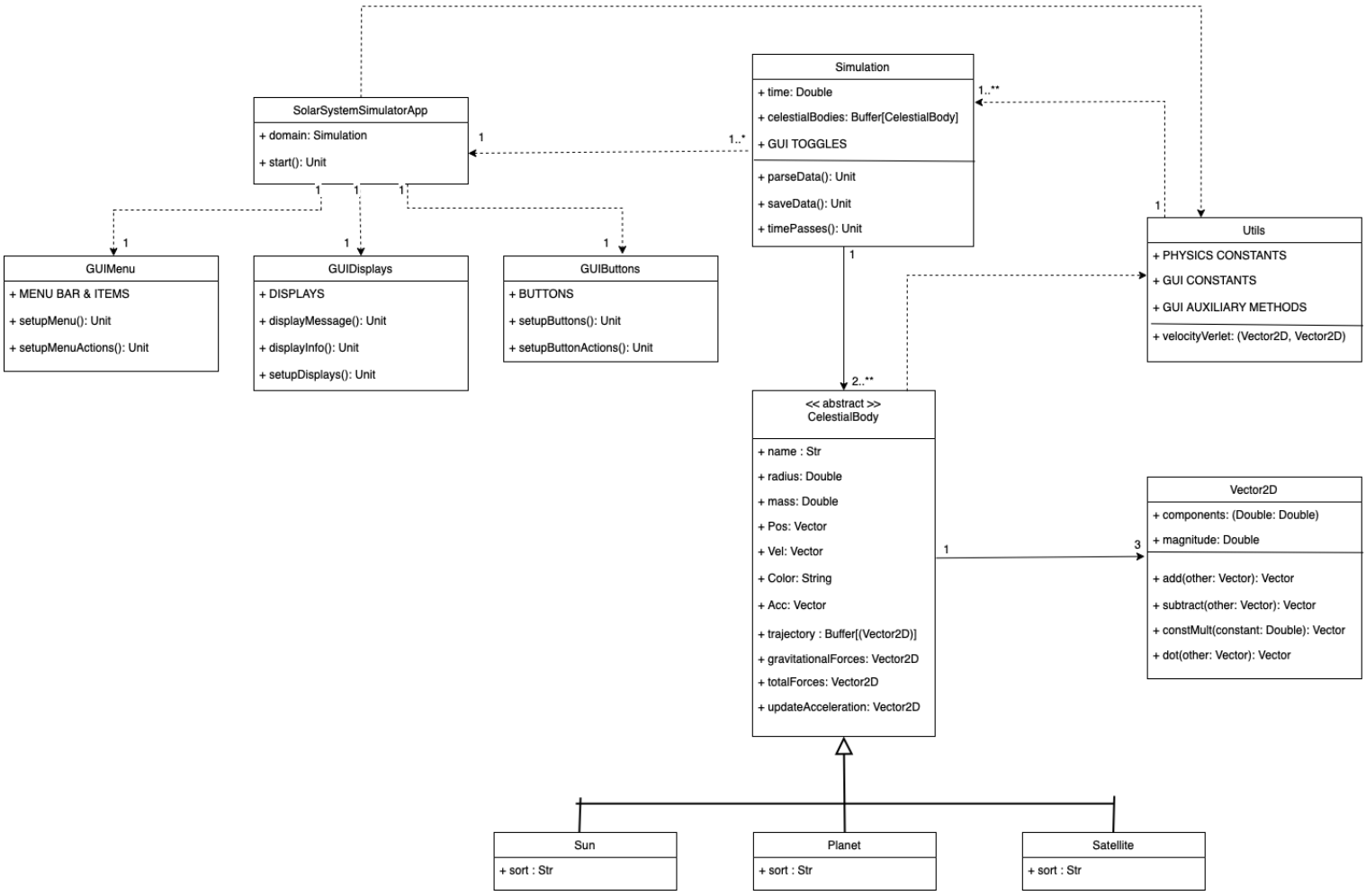
The bottom left corner contains buttons for controlling the simulation: a play/pause button, a reset button for returning to the initial state of the simulation and resetting the time, a time displayer, and a slider for adjusting the simulation time step. The slider will be set to the value specified in the input file, where the number on the slider represents how many days in the

simulation is equivalent to one second. For example, a value of 10 means that one second simulates ten days in simulation time. This can be verified by the fact that Earth's period of ≈ 365 days divided into 20-day intervals gives us a value of 18,25. If we adjust the slider to 20 in "theSolarSystem" we see that the time taken by Earth for one rotation around the sun is approximately 18 seconds.

The top left corner contains a menu bar with three menus. The first one, "File" contains features for starting a new simulation (i.e. parsing in a new .txt file), as well as for saving the current state of the simulation by overwriting the previous data or by saving as a completely new .txt file. The "Edit" menu enables the user to add a celestial body to the simulation by entering the values of the body in the same format as in the simulation text file. There is also a method for changing the radius of the sun, which with a constant mass effectively means a change in density.

Finally, the top right corner is reserved for an information box that is displayed when the user pauses the simulation, clicks on a celestial body with the mouse, and resumes the simulation. The box contains a summary of the body's name, type, position, radius velocity and distance from (the largest) sun. The user can hide the box by clicking again on the same celestial body while in pause mode. The box will continue displaying after having pressed play, with live updating values as the body moves through space and time.

Program Structure



Let us start by working our way up from the smallest pieces. We have first defined a Vector2D class representing a vector object. This greatly simplified calculations in all aspects - indeed a singel vector object is easier to manipulate and perform calculations with than two separate calculations, one in x-direction and one in y-direction. A Vector2D object contains an x- and y-component and has defined most standard mathematical operations for vectors (arithmetic, scalar multiplication, dot product, magnitude et.c.). For some methods we have two versions: one for returning a new vector and the other for modifying an existing vector (thus returning Unit).

The abstract class CelestialBody represents exactly that, an object in our simulator. The class contains data of the characteristics of each body, like name, radius, masses, initial velocities and positions and so forth. The most important part here are the methods for calculating the gravitational force the body experiences due to its surroundings, as this will is important information for correctly updating its velocity and position. Other methods are used for collision checks. This abstract class is then split into three extended subclasses: Sun, Planet and Satellite, who only differ by their sort string values. This is useful for identifying the types in other methods, but other differences can easily be added as well as a result of this separation.

The Simulation class represents a single simulation. This object is responsible for parsing the data from the .txt file, creating, storing and updating the state of the simulation. Each simulation contains a specific amount of celestial bodies stored in its buffer. Methods parseData() and writeData() are for opening and saving simulations, a collision detector and a method for updating the entire state of the simulation exist as well. This method updates positions of all bodies using the numerical algorithm and updates the time.

Every simulation object also has a group of “gui toggles” that trigger specific functions in the gui (like when to display vectors or trajectories).

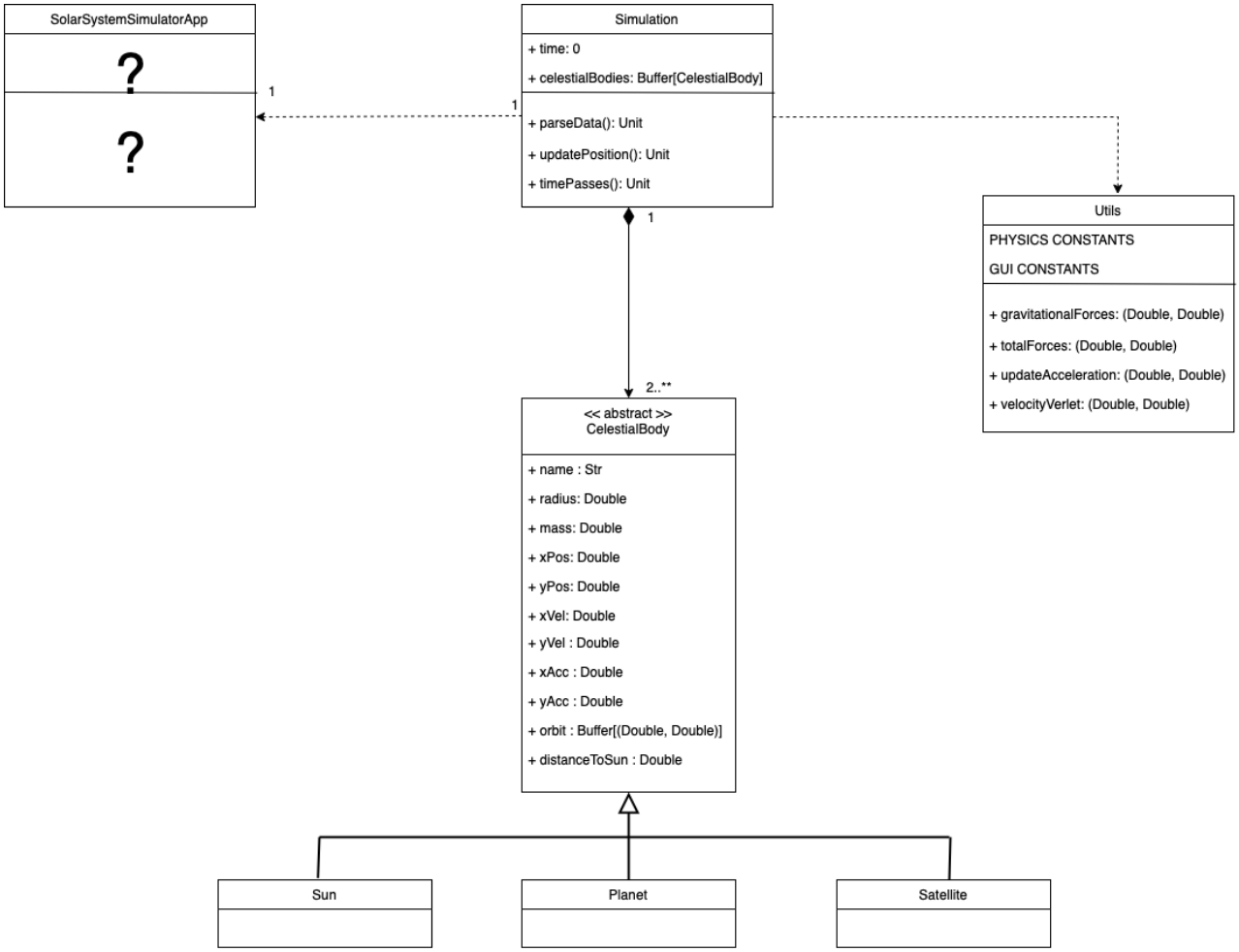
SolarSystemSimulatorApp is the main gui for the program, containing the app object for launching the entire thing. The object contains a start() function that sets up a stage and calls relevant methods for setting up buttons, displays, animation timers et.c.. All of the content in these methods were originally defined in the start() method itself, but that quickly became clunky and long so it was better to categorize and split the setup in parts, wrap the information around in methods and then let start() call those methods instead. These methods are in turn defined in their own files: GUIButtons handles all the buttons and their functionality, GUIMenu the menu buttons and GUIDisplays for the time and body information displayers. These three all share a similar structure: first they define the items, along with some functionality. Then the actual set up of these items on the gui window and their functionalities are wrapped in setup methods that will then be called by start() in SolarSystemSimulatorApp.

The final sub-part is the Utils file for anything auxiliary that didn’t logically fit in elsewhere. Gui and physics constants are defined here, as well as other gui auxiliary methods not related to the buttons, the menu or the displays. Examples would be all the methods responsible for drawing the bodies, trajectories and vectors onto the screen. But more importantly it hosts the Velocity Verlet algorithm, the heart of the entire system. More information about the algorithm below.

As for changes with regards to the original plan there weren’t too many. The biggest change was the splitting of the GUI into the different sub-parts, which made things much clearer, and even made the IDE smoother to

operate on my old(er) computer. The true first draft didn't implement a vector class but rather calcualted velocities, forces and accelerations in x and y components. This became quickly tedious, and after more consideration with the assistant a vector class was implemented.

Other smaller changes include the movement of methods from one part of the program to another. Some methods for updating forces and accelarations fit better in the CelestialBody class since they refer to values of a specific body. In contrast, velocityVerlet() is a general algorithm method not related to any specific instance, and is thus better suited in the Utils file. Some of the content in Utils could have further been split up into their own separate files, but I don't feel like it would have added much more clarity. For example the gui timer could instead be set up in a single file just like the buttons and menu of the gui. Here is the first draft UML for comparison:



Algorithms

There is really one main algorithm that drives everything forward: the Velocity Verlet numerical method. This is only one of several numerical methods used to approximately integrate Newton's laws of motion. A two-body problem is analytically solvable, but when more bodies are introduced to the system things quickly become complicated. The algorithm is one of many versions of an earlier “basic” Verlet algorithm.

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\,\Delta t + \tfrac{1}{2}\,\mathbf{a}(t)\Delta t^2,$$

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

The algorithm is split into two steps, one for finding the next position at t + Δt, and the other for finding the next velocity of the object at t + Δt. By analyzing the two steps we see that the required input is the current position, current velocity, current acceleration as well as the next acceleration at t + Δt. For this to work properly we need to make sure things are calculated in the correct order. Here is the main workflow for the updating of simulation state, using the algorithm as well as auxiliary methods:

1. At t = 0 set all accelerations to zero
2. Calculate the total gravitational force experienced by a body due to all other bodies in its surroundings, using Newton’s law of gravitation and vector addition
3. Use Newton's a = F/m to calculate the acceleration at t + Δt

4. Use the new acceleration, the current acceleration (0), along with current position and velocity (initial values specified) in the Velocity Verlet algorithm to update values for position and velocity at $t + \Delta t$.
5. Repeat for each body in the solar system to drive the entire simulation forward.

A nice trick used here is to call the updateAcceleration() method as an input parameter in the Velocity Verlet algorithm itself, thereby both updating the simulation correctly and assigning the new acceleration as the current acceleration.

Like mentioned there are other similar numerical methods that achieve the same purpose. Example include explicit / implicit Euler method, or a Runge-Kutta (RK) method. When making a choice one must reconcile with each method's strengths and weaknesses, like efficiency, accuracy, stability and complexity. The Velocity Verlet algorithm is fairly simple math-wise, had a good amount of information available on the internet, is not computationally intensive and provides somewhat accurate and stable systems. The algorithm has been used extensively to simulate both miniscule particles and massive celestial bodies, so overall a very save choice.

Data Structures

As for data structures things were kept as simple as possible. Granular data for celestial bodies and simulations are stored in simple variables, mutable or immutable depending on the purpose. As for larger collections of data there are only a few: a celestial body has a trajectory buffer for storing its position through time. This is used for drawing the trajectories. The other case is the simulation class itself, containing a celestialBodies buffer for containing all the bodies that are parsed in through the .txt file. Both buffers are intentionally mutable. The trajectories buffer is reset every time the draw toggle is switched on, meaning it will only start drawing from the body's current position as opposed to its initial position at t = 0. The celestialBodies buffer is also mutable so as to enable the user to add bodies mid-simulation.

There also exists a special bodyOnDisplay variable in the Simulation class to choose what body's info should be displayed in the top right corner. This is a special type of Option[CelestialBody] to enable not having a body selected as well.

Files and Internet Access

The program has no connection to the internet and only accesses local files on the user's computer. The program only supports .txt files that are correctly structured: the first two lines are numbers, specifying the simulation time and length of time step (the larger the faster but less stable). The following lines are each for providing parsing data for every celestial body: sort, name, radius, mass, x-pos, y-pos, x-vel, y-vel, color. If an incorrectly structured file is selected for parsing the program will show an error with some information. In some specific cases (like negative values for mass or radius) there will be more detailed information messages.

The program comes with a couple different pre-made files for testing the app, free to modify or use as a template for newer files. The more important ones for testing the program are mentioned above in the user interface section. Like mentioned there needs to be a pre-selected file for launching the app, but there also exists an option to open a new simulation file via the menu.

Testing

Testing was kept pretty simple and was mostly done via the gui. This partly because of my inexperience with unit testing, but also because the program is rather gui-reliant, and one can quite easily verify if stuff works as expected simply by using the interface.

Testing was partly implemented using skeleton classes of the celestial bodies, where only the most important features were added. Implementing the Velocity Verlet algorithm was mostly done blindly, and only tested and modified afterwards when a simple gui window had properly been set up. No separate unit tests were built, but features were tested in the gui by actually using the program as a user. I also always made sure to test for incorrect behavior to see if exception handling was working properly. I had originally planned on building some unit tests, also for my own learning benefit, but didn't have the time to get into it and chose the faster way in order to make progress on the program itself. The program now passes basically all requirements that were posted in the project information, with only minor inconveniences.

Known Bugs and Missing Features

All requirements are largely met, but some minor bugs have been found. For one, due to my non-familiarity with the ScalaFX gui library there exists some parts of my code that I can't fully modify and cause some minor annoyance. For example, when opening the app the console prints a small error message regarding some JavaFX configuration. Sometimes when using menu features, like adding a new celestial body mid-simulation, the program will print to the console some other strange error message that I can't seem to decipher. These messaged don't affect the program functionality in any way but would ideally not exist.

I also had some issues implementing the feature for zooming out when a celestial body is out of bounds. It seems to work the first time it happens, but later sometimes zooms out again, and in other cases zooms back in if a planer enters the original "space box". This could probably be fixed with a better understanding of the library tools and a smarter implementation, but I simply could not get it to work better and had to forfeit my attempt. The app does however report when a planet exits the original window, which I feel like is enough for most users to hit reset, since the main object of interest is the entirety of the solar system rather than some specific planet wandering off into eternity.

The buttons also seem to only be working using spacebar, since clicking on them with the mouse only highlights them. I never got to understand the underlying problem, since I use event listeners for all actions, not simply key presses. The menu bar buttons can only be accessed when the simulation is in pause as well. There might have been a way to fix this, but I figured it

would not be worth my time, as simply pausing the simulation is not too big an issue for the user.

There were also some more features I wanted to but never got around implementing. Examples include that the info displayer of a body would round the distance from the (largest) sun to two decimals. I never got it to work, so had to choose between rounding to the nearest integer vs keeping the entire long set of decimals. I chose the former alternative. I also wanted to implement more specific exception handling, since there now exists some cases where the error message is quite basic and not providing too much relevant information.

3 Strengths and 3 Weaknesses

+ Clean and simple gui: the buttons all make sense. Menus and buttons are logically placed on the screen so as not to disrupt the view of the simulation.

+ Extra features beyond what was requested. The Lagrange lines view option is visually pleasing and serves well to check bodies position relative to one another. Features for saving, saving as and opening new files make the program feel more robust and professional.

+ Message displayer that relays info of what is going on under the hood. The user is made aware of if the desired action was successful or failed, as well as the reason why the program behaves as it does at a certain point.

- While the code was refactored into a fairly clean and readable program I did not pay much attention to cohesion and coupling between the different parts of the program. As of now there are lots of methods in some parts of the program accessing completely different parts, and if one were to thoroughly analyze this it would probably be a quite messy structure in that sense.

- GUI-related code is sometimes a bit messier and not fully understood, therefore harder to refactor or improve functionality

- In rare cases the button actions don’t work at all. I have no idea why this is, but simply restarting the program has fixed the issue for me

Process and Schedule Deviations

The process started off according to plan and was not radically changed, but there were minor differences in the plan vs reality. I was stuck for quite a long time on getting a working GUI, and had originally planned to settle on the easy-intermediary level of the task. When I finally got things to work I became much more motivated and decided to push for the challenge.

As far as the schedule, while it did give a good rope to hang onto I must admit that the workload and frequency varied more than expected. Due to a heavy courseload there were weeks where I didn't touch development at all, and others where motivation was high and I worked for several days straight. The weeks did however match what I was working on quite well, except for week 11 and 12 that was supposed to be pure exception handling. Instead I implemented more features, and exception handling work blended more seamlessly in as opposed to being a standalone work block.

I did not feel bound by the set schedule in any way and had even forgotten about it at some point. It more so happened to be the case the the actual work done somehow matched the schedule fairly well. After this I realize that things seldom go according to plan, and that a good planner must also plan for these inevitable setbacks.

Final Evaluation

Overall I feel like the project was a success! The program is working as intended, satisfies all criteria more or less and even has some extra features. No significant shortcomings exist that drastically decreases the quality of the user experience. The program structure, while not inherently following high cohesion loose coupling, is clean and readable, and have made use of OOP principles. While the extended subclasses don't differ much from each other as of now features can easily be added in the future. Data structures make sense and the method

The program could be improved firstly by fixing the minor issues explained above (error messages, buttons) as well as by doing better refactoring to reduce coupling and increase cohesion between sub-parts. Buttons would ideally work also by mouse clicks, and even without having to press pause. New features could of course be added as well.

What I would do differently if I had to start over and had more time would be to implement unit tests and perhaps some mock/stub classes, simply to learn how that development process works.

References

GUI:

https://www.scalafx.org

https://javadoc.io/doc/org.scalafx/scalafx_2.13/latest/index.html

https://youtu.be/k5tPASiRd_Y (ScalaFX YouTube tutorial series)

Velocity Verlet:

https://en.wikipedia.org/wiki/Verlet_integration

https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2011/06/MA221-6.1.pdf

https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html

https://www.youtube.com/watch?v=g55QvpAev0I

https://www.youtube.com/watch?v=AZ8IGOHsjBk

https://www.youtube.com/watch?v=3HjO_RGIjCU&t=386s

General:

https://www.youtube.com/watch?v=7axImc1sxa0&t=287s

https://www.youtube.com/watch?v=WTLPmUHTPqo&t=916s

https://www.youtube.com/watch?v=4ycpvtIio-o

https://medium.com/analytics-vidhya/simulating-the-solar-system-with-under-100-lines-of-python-code-5c53b3039fc6

Reddit

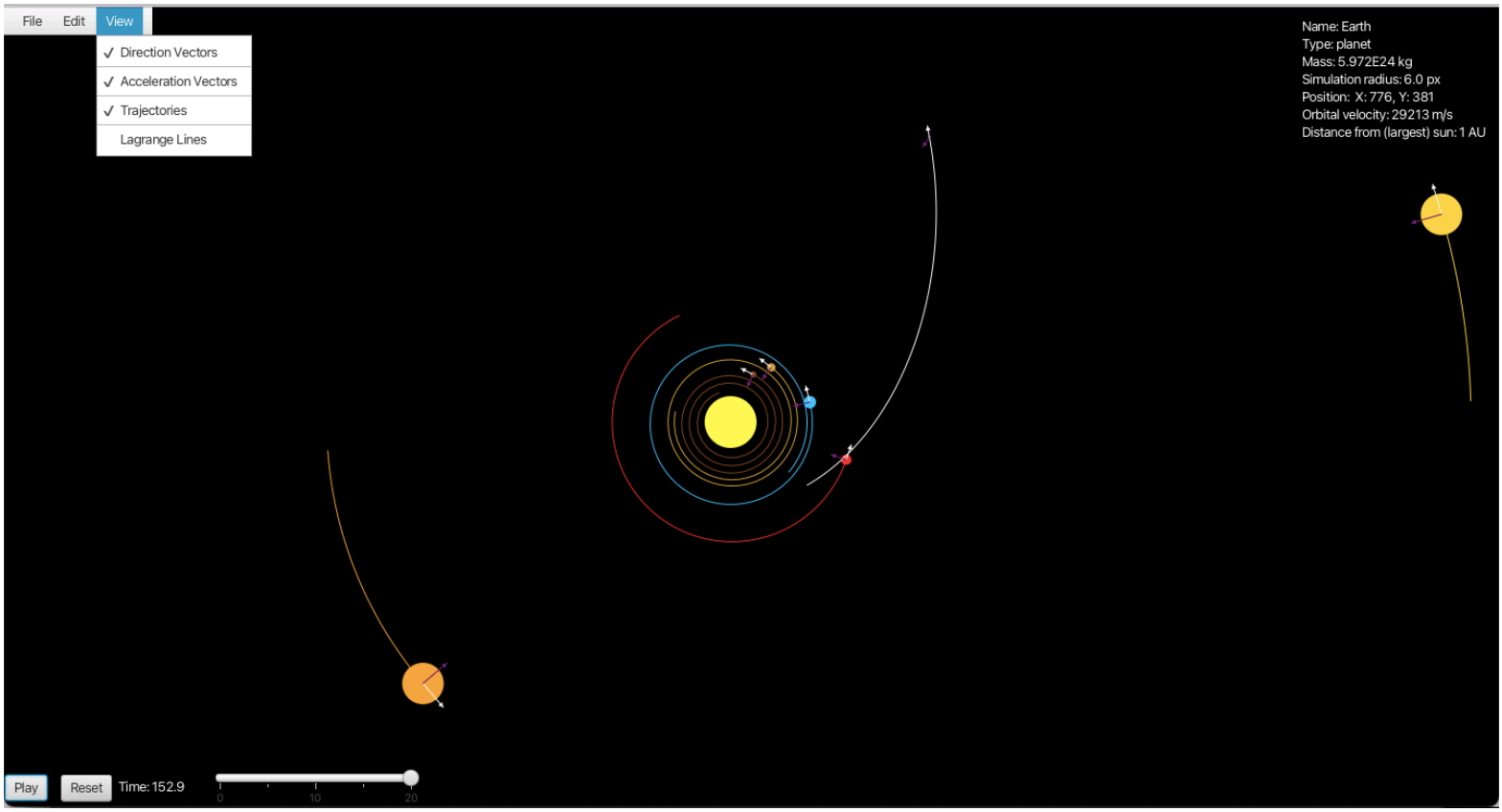
Stack Overflow

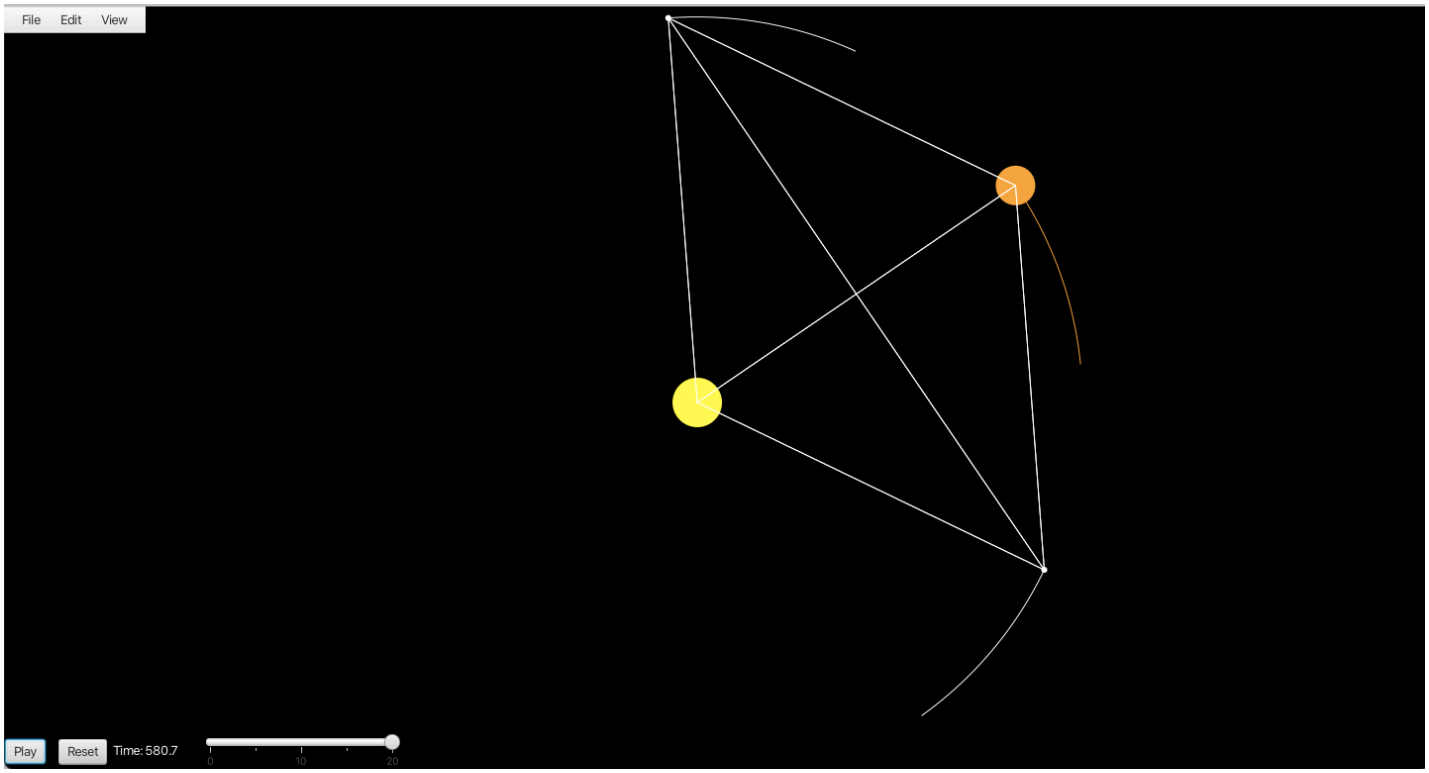
Appendixes

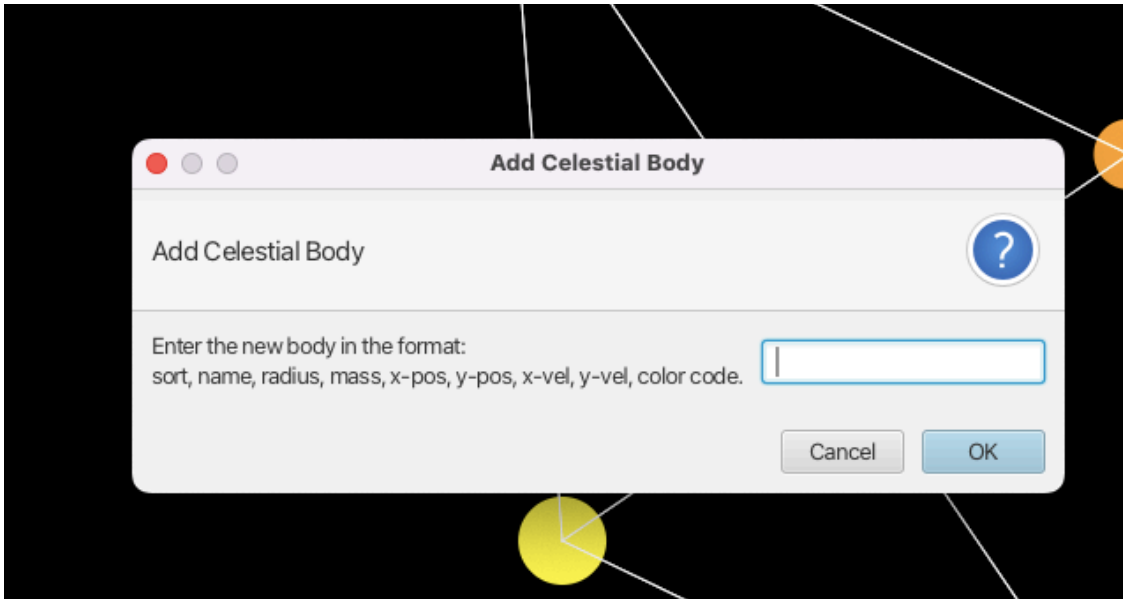
Link to public GitLab repo w/ source code:

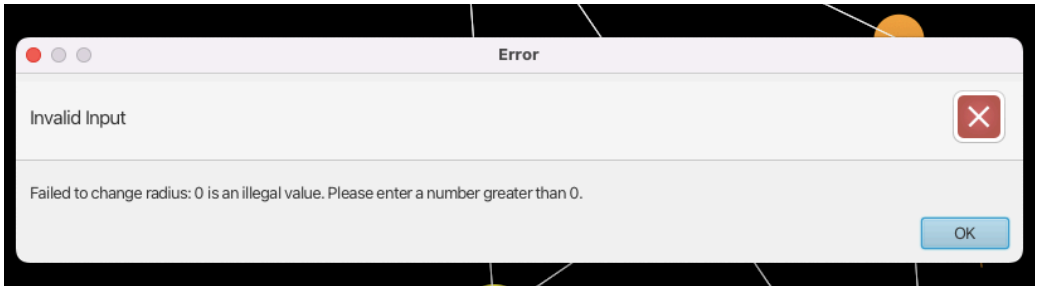
https://version.aalto.fi/gitlab/michaed1/os2-solar-system-simulator

Screenshots of program:




File
Edit
View
Play
Reset
Time: 580.7


Add Celestial Body
Add Celestial Body
Enter the new body in the format:
sort, name, radius, mass, x-pos, y-pos, x-vel, y-vel, color code.
Cancel
OK


Error
Invalid Input
Failed to change radius: 0 is an illegal value. Please enter a number greater than 0.
OK