Solar system simulator

Miska Tammenpää

729637

Tietotekniikan kandidaattiohjelma
1st year

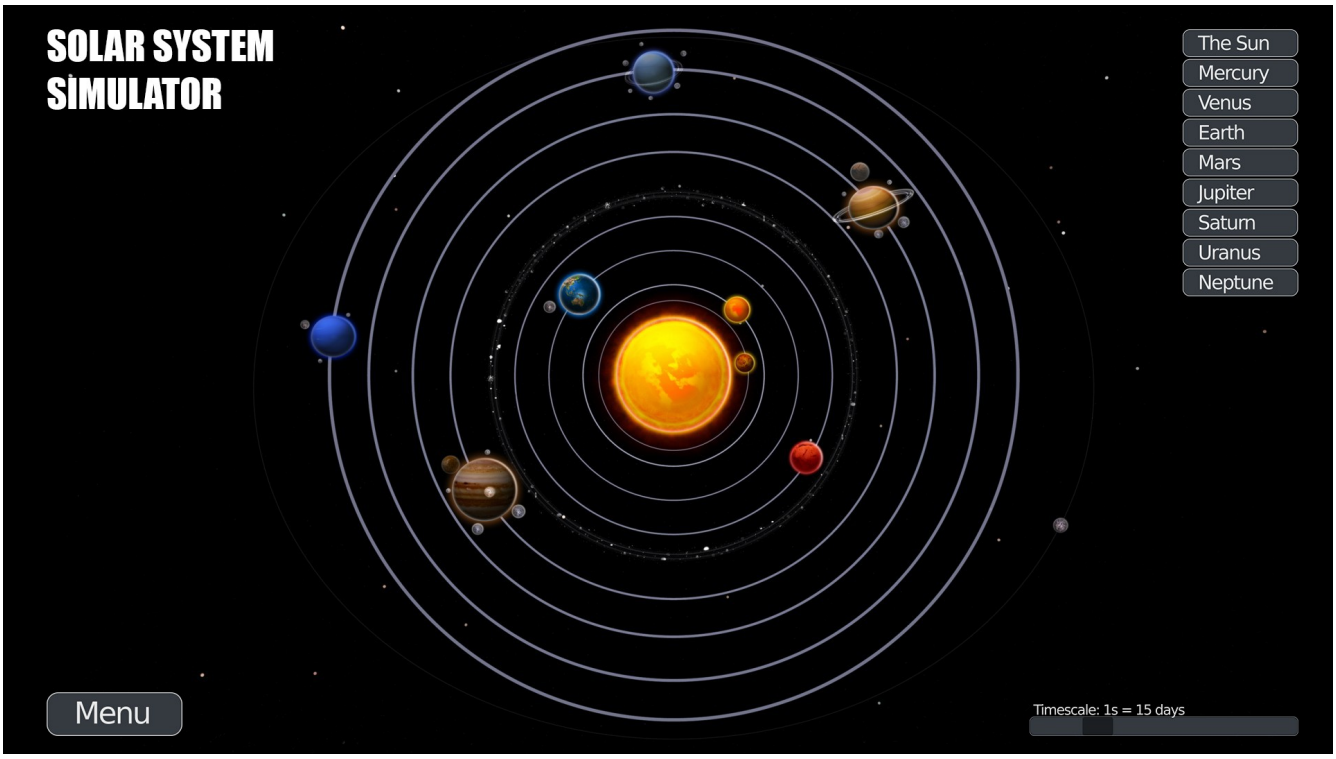
13.2.2020

General plan

Overview

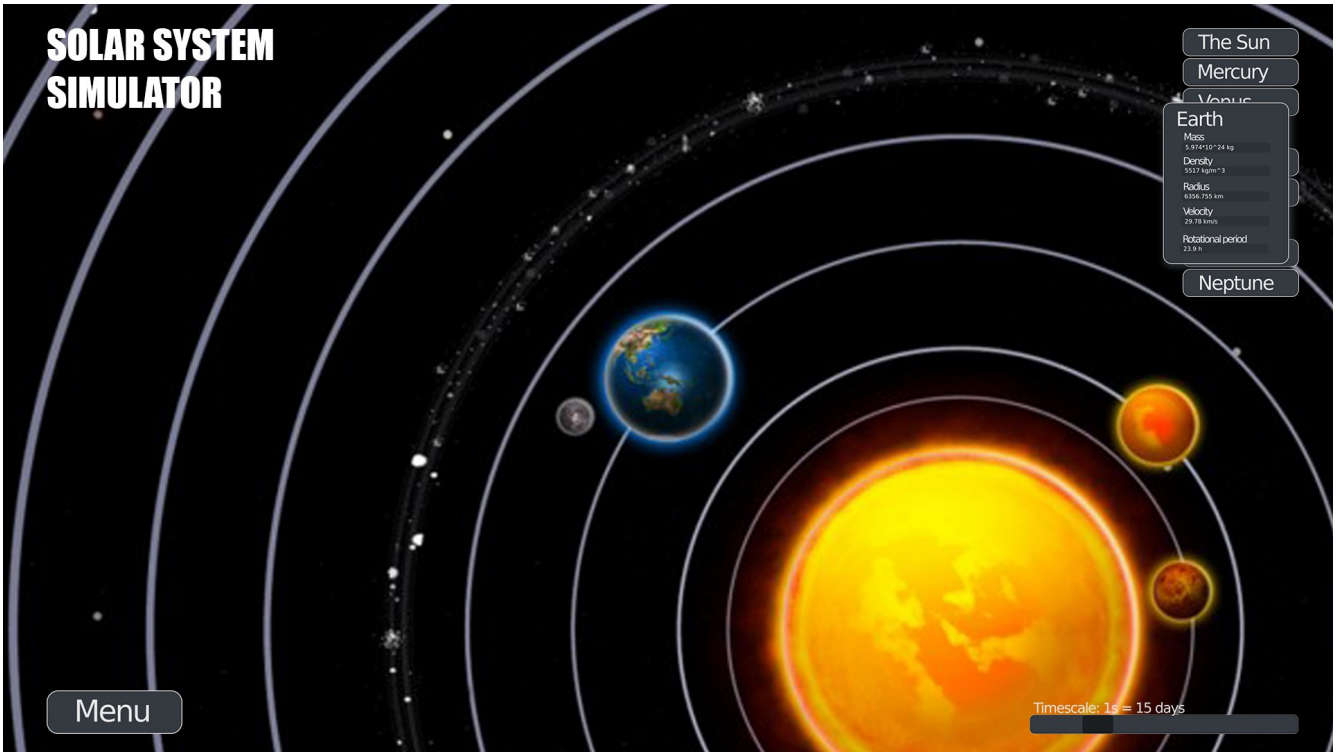
The idea of the project is to create a program that simulates the movements of different bodies in a solar system. The objects will be represented in a graphical user interface and the user will be able to create new objects and change their attributes. The simulation will then display how gravitation and Newton’s laws act on large bodies over time.

As well as changing the attributes of the objects, the user can change the timescale of the simulation to see longer term effects caused by gravitation.



(The scale of the bodies and their orbits in this draft is irrepresentative of what it will be in practice)

Selecting an object from the object menu on the right will display the object's current information and let the user change its values. Object selection will also highlight and/or zoom in closer on its visual representation.



The aim in the project is to meet the requirements for a "Demanding" difficulty project with a focus on the fluency of interaction between the user and the simulator.

Files and file formats

Under the menu button, the user will be able to create new solar systems from scratch (or from templates) and save or load solar systems to or from text files. The data stored in the files will be formatted roughly as follows:

```
#objectname
objectdata
-||-
-||-

#otherobjectname
objectdata
```

etc.

The simulation will then be able to collect all the objects and their information from these files.

Program testing

A key element to test in the simulator is the values given as parameters to the objects. Not only should the simulator not accept wrong kinds of inputs (letters in place of numbers, mass of zero, faster than light movement etc.) it should also not accept values that its not equipped to handle. For example giving the radius of a planet the same radius as the solar system and calculating the resulting behaviour afterwards is out of the scope of this project.

Another thing to test for is the simulator's performance. With many variables and long equations in play it will be crucial to know what kind of workload the simulator can take on. Knowing the simulator's limits, proper thresholds can be set for the objects' parameters.

The physical accuracy of the simulator can be tested as well. With a certain starting point for the simulation, the objects' values can be compared against known data of objects in space. This way, the simulator's accuracy can be honed down to a certain error threshold.

Technical plan

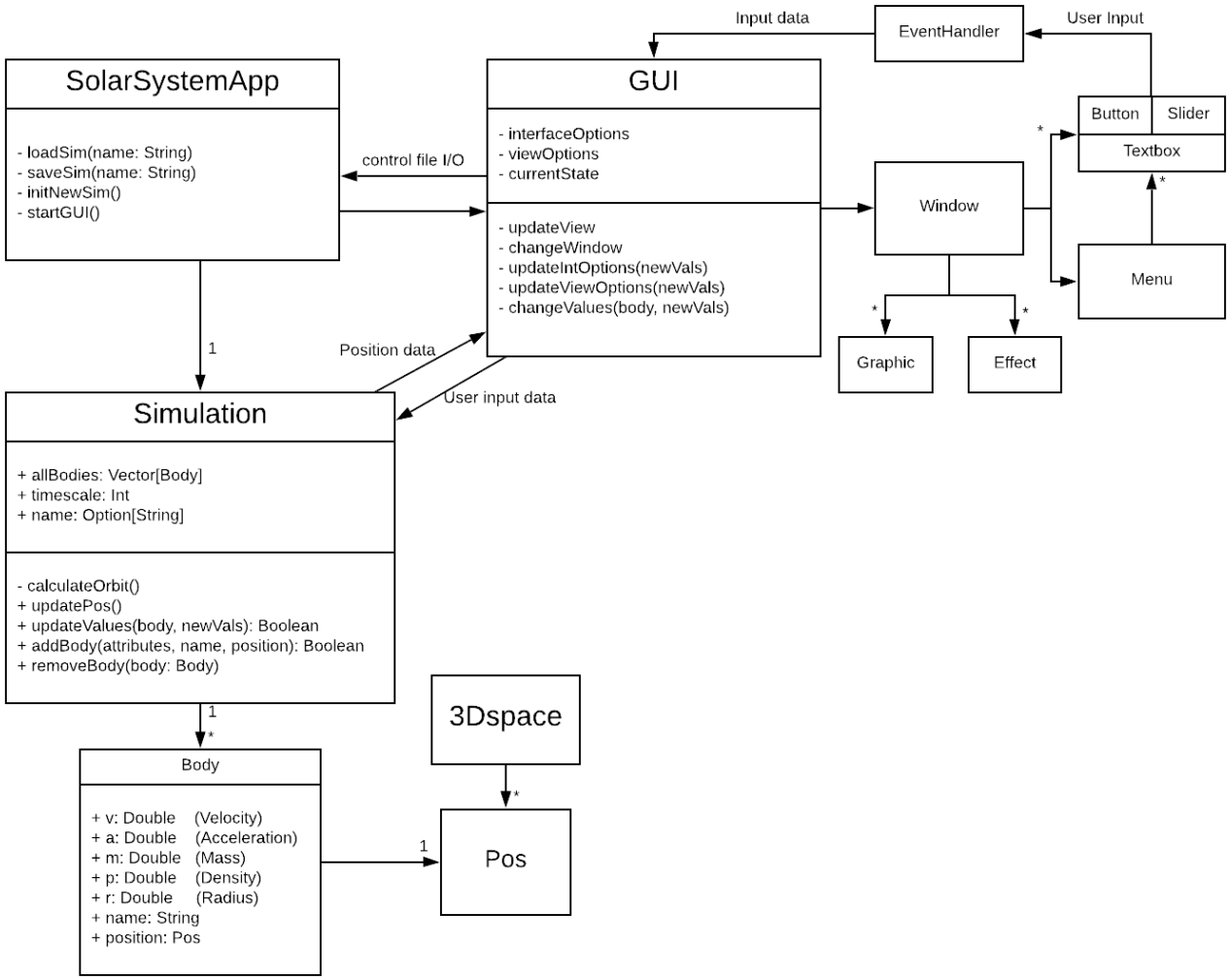
Structure

Stripped down to its core, a solar system simulator is a calculator for very specific equations. The other features, such as the graphical user interface, will be built around this. A GUI object will be rapidly updating a window instance by drawing a graphical representation of a solar system with UI elements like menus and buttons on top of it. The GUI only needs the position of the simulation's bodies in a coordinate system. It will query them from a "Simulation" object, which will take care of the calculation. It does this by reading information from instances of a "Body" class and computing the

positions of the bodies at any given moment. Really, the bodies here are just data structures since they don't have any functionality to themselves.

The user will interact with the simulator via instances of buttons, sliders and text boxes embedded into the UI. An event handler will then relay the data to the GUI, which will pass it forward where it needs to go. Generally, the cases of interaction will be changing the values of the instances of "Body" or creating and deleting the instances. Also they'll be used for changing GUI settings and initiating file I/O to save or load a state of a simulation.

A "SolarSystemApp" object will then tie all of this together by initiating the GUI and the simulation and handling the input and output of files.



Use case

The optimal usage of this simulator will be to conceptualize the movements of any groups of celestial bodies in relation to each other. While not accurate to the point where this information would be useful for, say, navigation in space, it is accurate enough to gain an understanding of the overall behaviour of the bodies. As a tool for education or even just an advertisement for astrophysics, I consider this highly valuable.

Booting up the program, the user is met with the program's default simulation: "Sol", our own solar system. From here the stage is set for the user. They can simply observe how the planets in our solar system move around or, for example, decide to change the mass of Earth while locking its radius. An event handler catches that and sends data to the GUI. The GUI then tells the Simulation to change values under variables m and p (mass and density) of the instance of "Body" named "Earth". Noting that the values of a body have been changed, the Simulation then calculates a new orbit for the body and any other bodies whose orbits are affected by the change to a high enough degree. The GUI, of course, notices none of this and keeps updating the view of the solar system according to the coordinates given to it by the simulation.

Having made this change and observing it's effects, the user is content and decides it's worthwhile to save the the current state of the simulation. They press the "Menu" button and under it the "Save" button. They are then prompted for a name to give to the save state and they respond with "Sol2" and press the "Submit" button. The event handler has been busy sending all these requests to the GUI and now catches the submitted data. It sends it to the GUI, which forwards it to the "SolarSystemApp". An IOStream is opened and the data of the simulation is written into a text file titled "Sol2.txt". The IOStream is then closed. Lastly the user again presses the menu button and under it "Exit", which shuts down the program.

Algorithms

The balance between accuracy and performance in the simulation is a hard one to strike. Assuming a relatively "sol-like" solar system (the planets are far from each other), the orbits of singular bodies around a static massive body are easily calculated with

Kepler’s and Newton’s laws. This is due to the other planets having very small effects on a planet’s orbit because the effect of distance on gravitational forces is greater than the effect of the planets’ masses.

Most of the calculations done by this simulation will be concerning simple two-body problems. However, in a situation where we want to calculate the movements of the satellites of a planet (e.g. the Moon orbiting the Earth) we need to consider it a three-body problem. In these cases, both the planet and whatever body the planet is orbiting around have an effect on the satellite’s orbit.

For three-body problems we’ll have to use differential equations. Which equation to choose depends on the exact prioritization between performance and accuracy. Fourth-order Runge-Kutta (RK4) and Semi-Implicit Euler are both good candidates. RK4 is very accurate and fairly fast where Semi-Implicit Euler is less accurate but faster. Both methods will be tested in use after which I will decide which one to stick with.

Data structures

The most important data structure used in the simulation will be the “Body” class. It will contain all the relevant information of a celestial body. Since most of this information is decimal numbers, an array of Double type values would also suffice. Though, in this case, readability and usability would decrease. Additionally, the difference of performance between these alternatives is trivial since the amount of bodies in a system at a time is fairly low.

An argument could be made for using integers in place of doubles for the bodies’ attributes since, in the case of celestial bodies, the numbers are generally large (not much decimals in use) and heavy calculation is done with them. However, in the case of the simulation, the advantage gained in accuracy is comparatively larger when using doubles than the advantage in performance would be when using integers.

The data structure used for the input and output of files will be simple text documents. The amount of data stored is fairly low and easily written and read so anything more complicated is unnecessary.

Schedule

Deadline: 24.4. → ~9 weeks
The goal is to get some part of the project done every week. This way it will be easier to focus on the part at hand and finish the project without having to push myself to work extra hours near the deadline.

Week 1 (24.2.- 1.3.):

The top priority in the beginning is to get the calculator running. At the end of the week I should have a working calculator for at least two-body problems, preferably three-body problems as well. This should be done in 5-10 hours.

Week 2:

Finishing up the calculator if not yet finished. A low-level interface for testing the calculator. Running tests with gnuplot or something similar for a rough idea on the accuracy of the simulation. ~5 hours

Week 3:

Graphical interface. Displaying things in at least 2D, perhaps starting on 3D as well. 5-10 hours.

Week 4:

Catching up on the schedule so far if running behind. If not, working on displaying the bodies and orbits in 3D. 5-10 hours.

Week 5:

More work on 3D. Fluent user-simulation interaction and testing it. Perhaps file I/O as well. 5-10 hours.

Week 6:

Polishing user interaction if needed and finishing up file I/O. Testing file I/O ~5 hours.

Week 7:

At this point, the necessities are hopefully out of the way. This means it’s time for flashy things. 5-10 hours

Week 8:

Overall testing of the simulator. Fixing broken things and breaking more things to then fix. 10 hours

Week 9:

More testing just to be sure. Polishing the product. However many hours it takes to feel satisfied.

Unit testing

When the calculator is ready, I will have to test it extensively with many different combinations of bodies to make sure that the equations work as intended. This is also the time to choose which method of calculation to stick with. This is what the whole project is built around so it’s crucial to get it right early. Testing the simulation with the bodies of our solar system against the data we already have on it will be especially important as that way the accuracy of the simulation will be easy to determine.

User input is another important part to test mostly to understand the simulation’s limits. A user shouldn’t be able to create systems whose computation is too much to handle for the simulation. Some stress-testing can be done to see what types of situations (e.g. extremely large or small values, too many bodies at once) are the most difficult to compute. Then limitations can be set so that the user can’t reach those situations. Here the problem is that the computationally difficult situations might not be as simple as “large values”.

File I/O has to be tested so that the program knows how to respond to the wrong kinds of files and corrupted files. This is fairly easy to do as the files in use are normal text files. Giving the program random text files should yield no response. The same goes for text files where the headers look right but the data under them does not. For example the program might be looking for a double type value and find a string in its place.

References and links

Info on RK4 and Semi-implicit Euler:
https://en.wikipedia.org/wiki/Semi-implicit_Euler_method
https://en.wikipedia.org/wiki/Runge-Kutta_methods
https://arxiv.org/pdf/1609.00915.pdf
https://gafferongames.com/post/integration_basics/
https://physics.stackexchange.com/questions/456808/how-do-computers-solve-the-three-body-problem

Overall celestial mechanics:
https://encyclopedia2.thefreedictionary.com/Orbit+of+a+Celestial+Body
https://en.wikipedia.org/wiki/Celestial_mechanics

Libraries:
https://rosettacode.org/wiki/Runge-Kutta_method#Scala
https://www.scala-lang.org/api/current/

Image used in the GUI draft:
https://i.pinimg.com/originals/d9/d3/22/d9d322e44852e87e8489b2991abbb33c.jpg

Inspiration:
http://universesandbox.com/