Solar system simulator

Miska Tammenpää

729637

Tietotekniikan kandidaattiohjelma
1st year

27.4.2020

Description

The program created in this project is a simulator that calculates and visualizes the movements of celestial bodies based on a set of initial values. The user can either play with the default scenario, our solar system beginning at a certain reference point, or create an entirely new star system. These scenarios can be saved into and loaded from files. Bodies can be added and deleted at will and the simulation then displays the effects of gravity on the system.

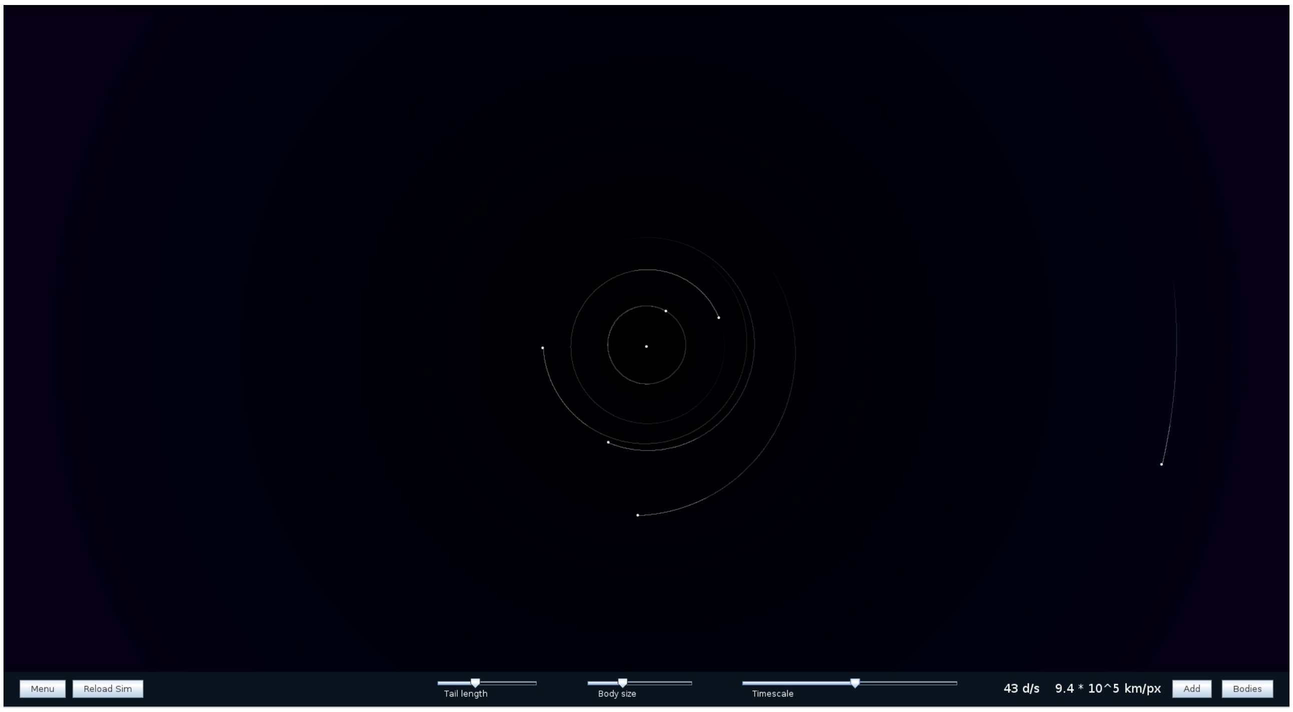


Figure 1: A snapshot of the interface in its normal state. The timestep of the calculation has been increased as can be seen from the third slider in the toolbar at the bottom

The simulated star systems are displayed in a graphical user interface which constantly draws the bodies' updated positions in relation to each other. Here the project deviates from the original plan as the interface is 2-dimensional instead of 3-dimensional like was planned. However, this doesn't have much impact on the outcome since the 3rd dimension of space can be simulated in a 2-dimensional format.

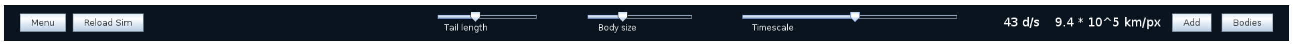
With the interface, the user can change parameters of the simulation such as its displayed scale and the timestep between each calculated state.

Considering the requirements defined in the project description, I think the implemented software is on the 'Demanding' level.

User interface

The software for using the simulator is packaged inside a .jar file in the simulator files so, with java installed, a user can run the program simply by running the jar file. Here it is worth noting that the program adapts to the user's screen resolution. However, it was designed for a resolution of 1920x1080 pixels, which it works optimally at. The software boots up to a visualization of our solar system at 18.04.2020, 00:00:00. Here the user can get around the star system by simply clicking and dragging the interface or zooming in closer on bodies with the mouse wheel.

The only UI element visible by default is the bottom toolbar.



It contains buttons with which the user can access the rest of the interface, number displays which conceptualize the simulation's timescale and it's physical scale and sliders for changing simulation parameters. These are the length of the 'tails' or orbits of the bodies as seen in figure 1, the displayed size of the bodies and the timescale. The buttons in the toolbar are, from left to right, the 'Menu' button, which opens a menu where the user can access file I/O and exit the program, the 'Reload sim' button, which reloads the most recently loaded simulation, the 'Add' button, with which the user can simulate more bodies and the 'Bodies' button which opens a panel on the right side of the screen for displaying information about the bodies.

The menu:

Clicking the 'Menu' button in the toolbar, a popup menu is opened in the center of the screen. It displays the title of the program and four buttons.

The first of the buttons, the 'New' button creates a new empty simulation. The second and third, the 'Save' and 'Load' buttons open a dialog asking the user to input a file name to save or load respectively. If the current simulation has already been saved, the save button opens a dialog asking the user if they want to overwrite the saved simulation. The last button, 'Quit' simply exits the program. The menu, like all other popup interfaces in the program, can be exited by simply clicking outside the window.

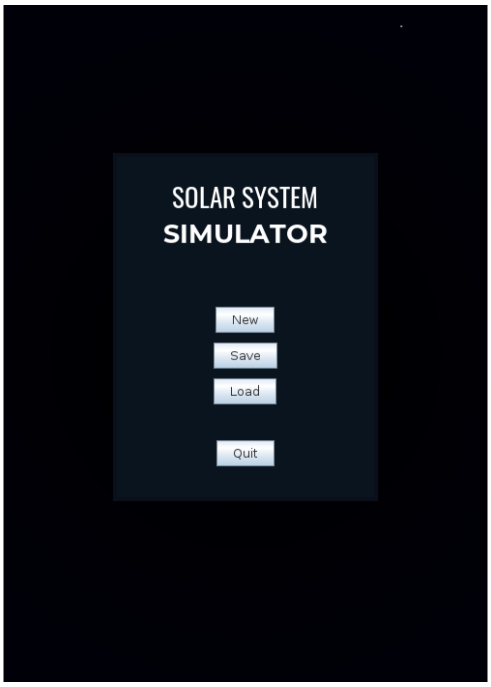
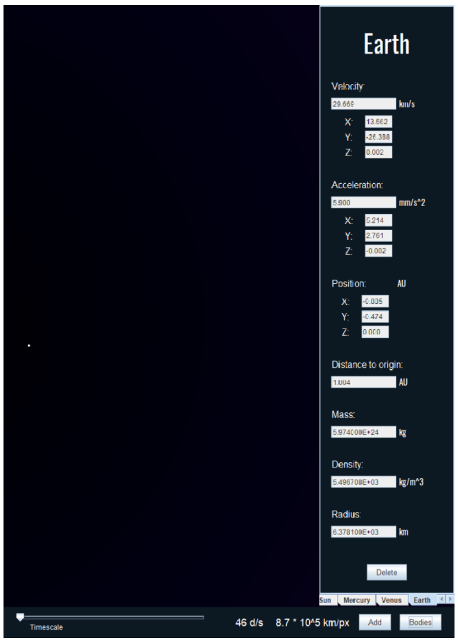


Figure 2: The main menu. From here the user can access file I/O or quit the program

The info page:

The ‘Bodies’ button in the toolbar brings up the info page of the bodies. It contains tabs for each of the currently simulated bodies. Each tab displays its respective body’s data updated constantly. The body’s velocity and acceleration are displayed both as the magnitude of the whole vector and the magnitudes of the vector components in three dimensions. The position is displayed as it’s 3-dimensional coordinates in a cartesian coordinate system. Also the body’s distance to the origin is shown. For Sol in its initial state, this means the body’s current distance to the sun. The last data pieces shown are the body’s mass, density and radius. At the bottom of the info page is the delete button, which, as expected, deletes the body.


Earth

Velocity: 29.669 km/s
X: 13.662
Y: -26.388
Z: 0.002

Acceleration: 5.900 mm/s^2
X: 5.214
Y: 2.761
Z: -0.002

Position: AU
X: -0.035
Y: -0.474
Z: 0.000

Distance to origin: 1.004 AU

Mass: 5.974000E+24 kg

Density: 5.496708E+03 kg/m^3

Radius: 6.378100E+03 km

46 d/s 8.7 * 10^5 km/px

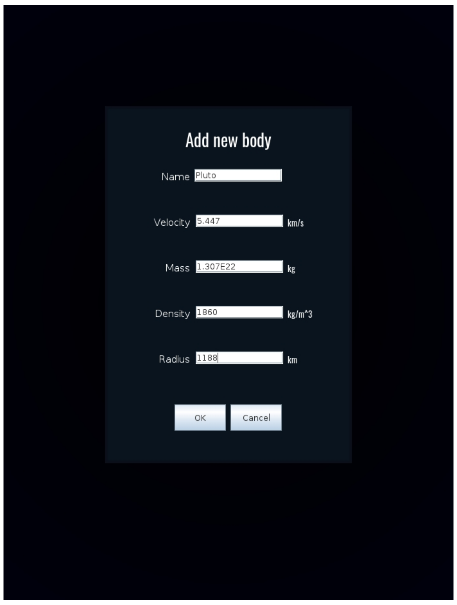
Figure 3: Earth's info page being displayed

Clicking the tabs below the page changes which body’s data is displayed. It also changes the body currently in focus in the displayed simulation.

Adding a body:

The user can start the process of adding a body by clicking the ‘Add’ button in the toolbar, bringing up an interface. This one contains several text fields and buttons ‘Ok’ and ‘Cancel’.

Each text field is linked to an attribute of the body as displayed by the labels next to the text fields. The unit of the attribute is shown on the right of the field. The user can input the values for a new body into these fields and then press the ‘Ok’ button to confirm. Pressing ‘Cancel’ or clicking outside the interface closes it. If the user inputs a value that is not valid, a message below the field with the invalid value pops up indicating that. Also clicking Ok either before inputting all values or when a value is invalid pops up an error message. If all values have been inputted correctly and the user clicks Ok the next step will begin.


Add new body

Name: Pluto

Velocity: 5.447 km/s

Mass: 1.307E22 kg

Density: 1860 kg/m^3

Radius: 1188 km

Figure 4: A body representing Pluto being added to the simulation

The UI elements of the GUI disappear leaving only the graphical representation of the bodies visible. In this state the user can move the mouse around and left click to place the body where they want to. A text box next to the mouse cursor displays the coordinates of the mouse in astronomical units. Of course, moving the mouse only changes the body’s location in two dimensions. Changing the body’s location on the z-axis can

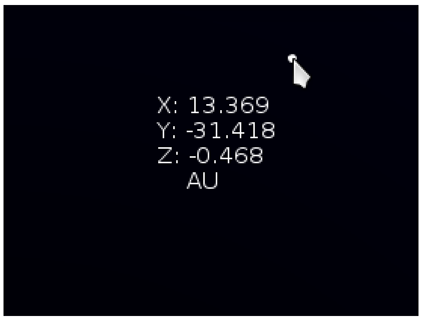

X: 13.369
Y: -31.418
Z: -0.468
AU

Figure 5: Pluto's position being chosen

be done by rotating the mouse wheel. If the user decides they don’t want to add a body after all, they can right click to cancel the process. If not, left clicking takes the user to the next step.

Now the body is in place but it’s velocity vector’s direction is unknown. In this step the user can aim the body’s velocity vector, again, by moving the mouse around. An arrow rooted at the body’s location and pointing towards the mouse shows the current direction. A text box next to it shows the velocity’s potential vector components in km/s. Again, the z-component of the vector brings up a slight issue. Here the program gets around it by imagining a set of unit circles around the body’s position defining a sort of hemisphere inside the xy-plane. The process is explained in more detail in the ‘Algorithms’ section.
Right-clicking here takes the user back to placing the body. This way they can change its place easily. Left-clicking again finalizes the process. After this, a message pops up telling if the body was added successfully. If it failed, a reason for the failure is included in the message.

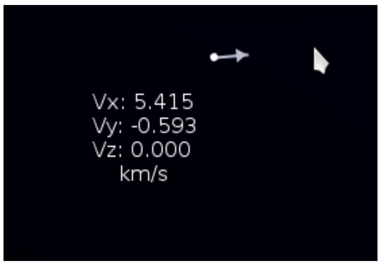
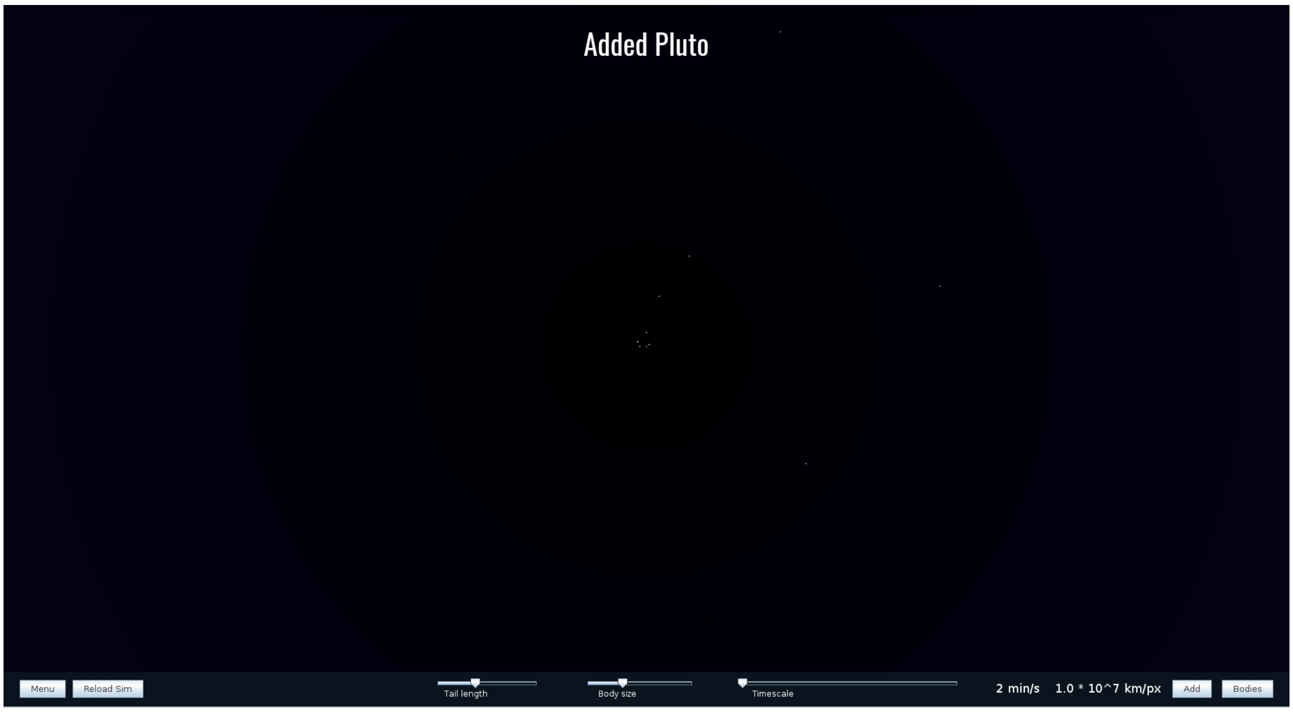

Vx: 5.415
Vy: -0.593
Vz: 0.000
km/s

Figure 6: Aiming Pluto's velocity vector


Added Pluto

2 min/s 1.0 * 10^7 km/px

Figure 7: Pluto successfully added to the simulation.

Program structure

The program is mostly defined by two parts, the interface package as the front-end and the simulator package as the back-end. The SolarSimApp swing application sits between them. It loads the solar system into the simulation and starts up the program.

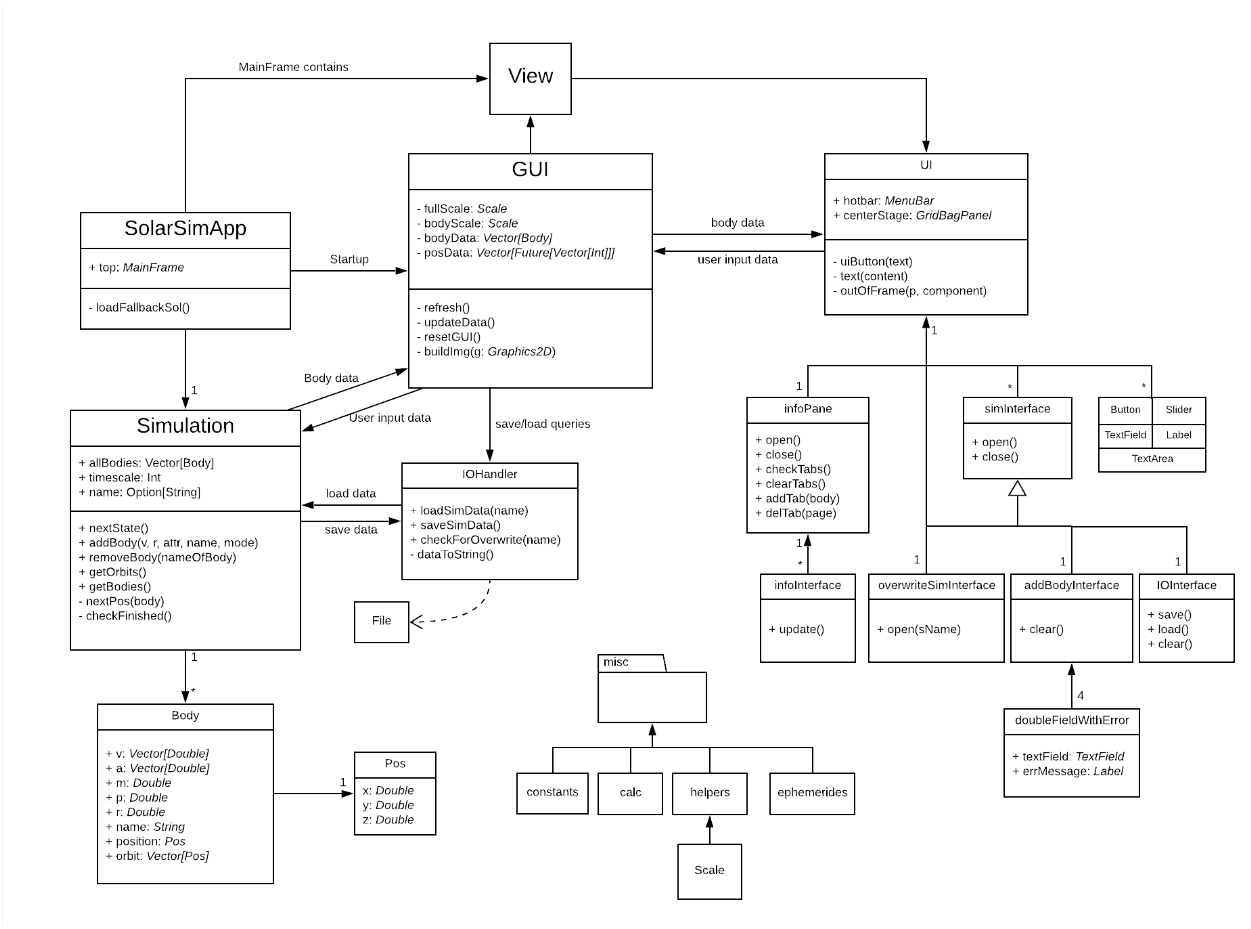


Figure 8: The UML diagram for the program

Simulator

The simulator package defines the bulk of the objects and methods used for calculation. Its key components are the Body class and the Simulation object.

Body is used for the construction of models of single celestial bodies. In practice, they're data structures containing attribute values and a few helper methods.

Simulation, as the name gives away, does the core calculations in the program. Its method nextPos calculates the next position for a single instance of Body given to it as a parameter. The positions of all bodies are calculated concurrently with the method nextState. It goes through the list of bodies being simulated, opening a new thread for each with a Scala Future, where nextPos is called for the body. The set of bodies is contained in the vector allBodies. There's no other variable pointing to the body objects so adding and deleting bodies is done by appending allBodies with a new instance of Body and filtering out a body respectively. The methods addBody and removeBody do these.

Simulation also has several helper methods and some setters and getters for communicating with the GUI. A key helper method in Simulation is the method checkFinished. It waits in a thread, separate from the calculation threads, for all the bodies' calculations to finish. Once finished it updates the bodies' internally stored positions. This way the bodies' concurrent calculations don't interfere with each other.

The simulator package also contains the definition for the cartesian coordinate system used when calculating positions. It is the Pos class, whose instances are points in 3-dimensions. They can be used for easily calculating the distance between two points by calling the 'd' method of a Pos instance with the other position as its parameter. The Pos class was implemented into the program fairly early in the project but later, as the calculation was done mostly with vectors, it became clear that using only vectors for the positions could've simplified the program. This, however, wasn't critical as the Pos instances are very simple.

Lastly, the I/O handler of the program is defined in simulator. All input and output of simulation files is done by the IOHandler object. Some resources are loaded independently by the GUI object as well, however.

Interface

The interface package consists of the GUI object and the UI package object. GUI is at the core of the program as it handles most of the communication between the program's different parts such as calling Simulation's nextState method to propagate the simulation forward. It also runs the timer that sets the pace for updating the program visually and does most of the program's event listening. The GUI isn't itself a part of the actual interface but is the backbone for it. It defines the methods for running the interface while getting necessary data from Simulation to tell the interface elements what to do. GUI draws the graphics of the program into an image object with it's method buildImg. GUI has a variable called 'view', the Scala Swing component at the heart of the user interface. The image created by GUI in buildImg is painted on the screen by 'view', after which all the program's UI elements are displayed on top of it.

The GUIs 'start' method is called by SolarSimApp when the program starts up. This activates the program's timer and gets things moving. In practice the timer is a java.util.Timer with a corresponding TimerTask. The task does two things every time the program ticks: calls GUIs refresh method which, in essence, propagates the simulation forward, and calls view's repaint method to create the image of the program on the screen. All of GUIs other methods can be seen as helpers to enable interfacing features described in the User interface section.

The UI package object is essentially the set of all UI elements in the program (except the earlier mentioned view, which is a special case). Generally, these are Scala Swing UI elements configured with certain values. UI does have some classes as well but these are just constructors extending the existing Swing elements to reduce the amount of repetition when implementing interface features. Take, for example, the simInterface class. An extension of Scala Swing's GridBagPanel, it defines most of the popup windows in the UI. Each instance has its own open and close methods along with a few preset configurations. This brings coherence to the implementation of the UI and has made creating the interface features far easier. Many of the individual UI elements have internal helper methods as well to reduce repetition in their definitions and improve code readability.

Miscellaneous

The program also contains the ‘misc’ package. It contains various constants, calculation methods and helper methods. Notably, the package object ‘calc’ defines functions for vector calculation and translating positions, and the package object ‘helpers’ defines class ‘Scale’, whose instances are used to scale the real physical sizes and distances to a level where they can be displayed on the screen. Different objects in the misc package are accessed by all other parts of the program.

The misc package also contains the ‘ephemerides’ package object. The data for the default solar system simulation is hard-coded into it. This way, if somewhy the default file is unable to be loaded, the data can be retrieved from the object.

Algorithms

The most important algorithm used in the program is, of course, the one that calculates the movement of the bodies. Originally, I planned to use and, in fact, did use Kepler’s laws for calculating the movements. They are easy to implement and very efficient but have the drawback of being inaccurate in situations where bodies are being attracted by several other bodies at the same time (n-body problem). The plan was to use Kepler’s laws for 2-body problems and numerical methods for three or more bodies. During the project I realized that the difference in performance between using Kepler’s laws and numerical methods is negligible. Thus, I decided to scrap using Kepler’s laws altogether and started using numerical integration in all cases.

The final algorithm uses Cowell’s method for calculating a function for the acceleration vector of the body.

$$\vec{a}(\vec{r}_i)=G\sum_{j\neq i}\frac{m_j(\vec{r}_j-\vec{r}_i)}{r_{ij}^3} \qquad (1)$$

Where:

$\mathbf{r}_i$ = the position vector of the current body (in relation to the barycenter)

$\mathbf{r}_j$ = the position vector of an attracting body (in relation to the barycenter)

r_{ij} = the distance between the two bodies

m_j = the mass of an attracting body

G = gravitational constant ($6.6743 \cdot 10^{-11}$ $m^3 kg^{-1} s^{-2}$)

Cowell’s method simply takes the vector sum of the acceleration vectors caused by the gravitational forces of the attracting bodies. Other popular orbit perturbation methods are VOP (variation of parameters) and Encke’s method. Cowell’s method is simple to implement and accurate, which is why I landed on using it. It can be argued that using VOP or Encke’s method would be more efficient (Roberto Barrio et. al., 2008). However, Cowell’s method works well for plain newtonian orbital mechanics. The perturbations caused by factors outside newtonian mechanics is outside the scope of this project so Cowell’s method is very suitable.

We can numerically integrate the result of equation (1) to first get the body’s velocity vector and then it’s position vector. Integration is done with the fourth order Runge-Kutta method (C. J. Voesenek, 2008) because of its efficiency and light performance requirements relative to its accuracy. As the body’s position vector depends on its velocity vector, we have to calculate them concurrently.

1:
$$\vec{k_{1v}}=\vec{a}(\vec{r_i})$$
$$\vec{k_{1r}}=\vec{v_i}$$

2:
$$\vec{k_{2v}}=\vec{a}(\vec{r_i}+\vec{k_{1r}}\frac{dt}{2})$$
$$\vec{k_{2r}}=\vec{v_i}\vec{k_{1v}}\frac{dt}{2}$$

(2)

3:
$$\vec{k_{3v}}=\vec{a}(\vec{r_i}+\vec{k_{2r}}\frac{dt}{2})$$
$$\vec{k_{3r}}=\vec{v_i}\vec{k_{2v}}\frac{dt}{2}$$

4:
$$\vec{k_{4v}}=\vec{a}(\vec{r_i}+\vec{k_{3r}}\,dt)$$
$$\vec{k_{4r}}=\vec{v_i}\vec{k_{3v}}\,dt$$

Where $\mathbf{v}_i$ is the current velocity vector and dt is the current timestep

Combining the results in (2) with necessary weights and the previous velocity and position vectors we get the new vectors

$$\vec{v_{i+1}}=\vec{v_i}+\frac{dt}{6}(\vec{k_{1v}}+2\vec{k_{2v}}+2\vec{k_{3v}}+\vec{k_{4v}}) \tag{3}$$

$$\vec{r_{i+1}}=\vec{r_i}+\frac{dt}{6}(\vec{k_{1r}}+2\vec{k_{2r}}+2\vec{k_{3r}}+\vec{k_{4r}}) \tag{4}$$

The bodies’ new position vectors are then relayed in Future wrappers to the GUI object which draws a corresponding circle on the screen immediately once calculation is finished.

The Simulation object also has a helper algorithm for calculating equation (1) which calculates the barycenter of the system.

$$\vec{r}_{BC} = \frac{\sum m_j \vec{r}_j}{M} \quad (5)$$

Where m_j is the mass of a body and $\mathbf{r}_j$ its position vector. M is the sum of the masses of all bodies.

As equation (1) takes into account the entire set of bodies currently simulated, it is only necessary to calculate the barycenter once for each state calculation.

The user interface also runs some algorithms. The most notable one of these is the algorithm which calculates the potential z-component of a body's velocity vector.

When the mouse is inside a certain range *dmax,* the radius of the hemisphere in figure 9, from the body to be added, the algorithm calculates an angle α from the mouse's location on the hemisphere. From this we can calculate the magnitude of the z-component with a sine function.

$$|z| = \sin\left(\frac{\pi}{2} - \frac{\pi \dot{d}}{2\, d\dot{m}ax}\right), d \le dmax \quad (6)$$

$$|z| = 0\,, d > dmax$$

Where d is the distance from the mouse cursor to the body.

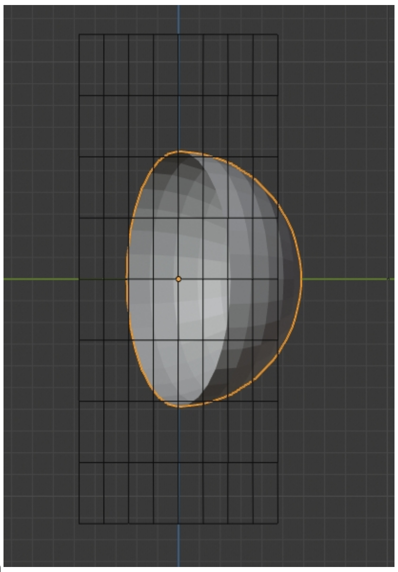


Figure 9: An illustration of the algorithm made in Blender. The lattice visualizes the xy-plane

As this only gets us the magnitude of the z-component, we need an external input for its direction. This is gotten from the rotation of the mouse wheel. The direction of the rotation defines the direction of the z-component on its axis, or the side of the lattice the hemisphere is on in figure 9.

The same feature could've also been implemented in the same way as the z-component of the body's position vector, by only rotating the mouse wheel. However, this solution felt like it's more intuitive to use for the user.

Data structures

The most prominent data structure in the program is the 'Body' class. Each instance has variables for storing all the values relevant for calculating their movements and displaying them. They also have a method for comparing themself to another body, which is done simply by comparing the two bodies' names, as they are unique. In theory, all you'd need to define a celestial body is a list of double-values. Thus, a vector of doubles would suffice. However, for the purposes of the program it is essential that the bodies have easily separable, unique identifiers, and having some of the body's values (velocity vector, position) in separate lists makes it easier to do calculations with the values. Considering these, it seems sensible to have a separate class for the creation and usage of the bodies.

A lot of the necessary calculation in the program is done with 3-dimensional vectors. These are implemented with immutable Scala Vectors. They are very fast to use for this purpose as changing the size of the vectors is unnecessary. After implementation, I realized I could have created a class extending the Vector class with the vector calculation methods implemented in the class itself. This would have improved usability and readabillity somewhat. However, it didn't seem worth the effort to convert to using a different class at the time of realizing this.

The Vector class is also used for containers for many of the data sets in use. There's no specific reason for this other than being in the habit of using Vector instances as containers.

Mutable Scala 'Queue' class instances also play a significant role in the program. They are used to store previous Body positions which are used to draw the tail, or orbit, of the body. They're also used for storing user inputs in the GUI object when the user is trying to do something that has destructive effects, such as deleting a body or loading a simulation from file. When the input is stored in a queue, it can be easily handled at a time when performing the action doesn't cause harmful effects. For example, if the user were to delete a body without these precautions while position calculation is taking place, it would likely lead to the program throwing an exception.

The reason for using queues specifically for these features is that the first-in-first-out nature of the queues makes it very easy and fast to add elements to the queues and remove them in the correct order. Consider the first use case mentioned. When storing previous positions of a body, all one needs to do is add it to the end of the list. If the list gets longer than is wanted, one only has to take out its first element.

File access

The main use for file access in the program is saving and loading the files defining a simulation. These contain the name of the simulation (which is also the name of the file) as the header. After that each line in the files contains the relevant data for a single body. The format in the lines is as follows:

Name;Vx;Vy;Vz;mass;density;radius;Rx;Ry;Rz

Where Vx, Vy and Vz define the velocity vector and Rx, Ry and Rz define the position vector. The units of the attributes are SI units. The acceleration vector or the orbit of the body are not stored in the files as these are not needed to continue simulating the body (technically, neither is density but it's a single number so it's easy to save and load).

Thus, the format for a single file is:

Simname
Name;Vx;Vy;Vz;mass;density;radius;Rx;Ry;R
Name;Vx;Vy;Vz;mass;density;radius;Rx;Ry;Rz
…
Etc. for as many lines as there are bodies in the simulation. For example, an Earth-Sun simulation's file might look like this:

```
Earth-Sun
Sun;0.0;0.0;0.0;1.989E30;1409.0;6.98E8;0.0;0.0;0.0
Earth;13562.03;-26387.70;1.71;5.974E24;5517.0;6378100.0;-1.32E11;-7.09E10;3976044.11
```

The actual saved files, of course, have the double values stored with regular double precision. All the files are saved into and loaded from the ‘simdata’ folder in the project files. The names of the files are in format <simname>.ssf. The example file’s name would then be *Earth-Sun.ssf*.

The GUI also loads some font files from the project’s ‘resources’ folder at startup.

Testing

Most of the testing was done during the programs implementation by trying different features and different combinations of features in the user interface. The seamless operation of the interface was an important element for me so it was necessary to make sure that using the program is very simple and intuitive. For example, when adding invalid values, the program should inform the user about the problem as soon as possible and never let those values be used to initialize an instance of Body.

The general testing of the program was done as intended. As the program leans heavily on the interface, it was easy to use it for testing different features. Also testing the behaviour of the simulation was made easy with the interface.

An essential thing that could and should have been tested is the exact accuracy of the calculations. The accuracy was tested by comparing the known data of the solar system gotten from the Jet Propulsion Labratory’s planetary ephemerides (NASA, 2020) with the data displayed by the interface. However, it could have been tested much more rigorously by creating unit tests for different scenarios with different timesteps. Especially since the API for getting the data is very easy to use. This way the change in relative error as a function of the timestep or the nature of the scenario could have been documented. The plan was to test the program this way but in the end, the time was spent on other parts of the program.

Known bugs and missing features

A, sort of intended, missing feature is the inaccuracy of Mercury’s orbit in the default simulation. The inaccuracy is caused by the fact that Newton’s laws don’t account for Mercury’s total orbital precession over time (wikipedia.com, 2020). As this project’s scope was studying only the effect of Newton’s laws on bodies, this isn’t a crucial issue. If the scope was broadened, the simulation could fairly easily account for this as well, as the total precession and its causes are well documented.

Another missing feature is the ability to edit the values of bodies. This was a planned feature since the beginning and is, in fact, partly implemented already as the Simulation object has a method for it. However, it never became relevant enough to implement fully. Also, whenever I was testing the program, I didn’t feel I even needed to edit the bodies.

Thus, the feature wasn’t implemented in the final product, even though doing so would be fairly easy. If the program’s development was carried out further, this would be among the first implemented features.

The detection of collisions is missing as well. Collisions technically do happen but as the bodies are just points of mass in a coordinate system, the way the collisions present themselves is by an orbiting body getting closer and closer to the attracting body until it reaches esape velocity and is slinged off its orbit. This could be fixed by hard-coding this behaviour to be interpreted as a collision. It wouldn’t be a difficult thing to do but it would require some tinkering with the position calculations. If neither body was deleted after collision, the distance between the collided bodies would be zero and this would lead to dividing by zero in Cowell’s method.

A sort of bug/unintended behaviour is seen when zooming in on the view in the simulation. The zooming works by changing the rate by which the distances of the bodies compared to the origin are scaled. Thus, when zooming in closer, the middle point of the view always approaches the origin in the coordinate system since dividing a number by larger values makes the number approach zero. So if the user has changed the offset of the view by clicking and dragging, zooming in doesn’t zoom in directly toward the location in the middle of the screen but towards the origin.

Another bug related to zooming is a slight graphical glitch which occurs when a body is in focus. When zooming in or out, the body in focus is sometimes drawn with a slight offset, whose direction depends on the direction of zooming. This is clearly caused by the the bodies being drawn concurrently and some overlap between the the offset, the scaling and the drawing of the body. The tail of the body was previously behaving in the same way but this was fixed by making the changing of the scale happen only before starting to draw the bodies. I thought the graphical glitch of the body itself would be fixed as well with this change but it seems not.

Strong and weak points

A definite strong point in the program is the robustness of the calculations. You can throw a lot of different situations at the simulation and it handles them gracefully. It takes some effort to even reduce the program’s performance significantly. A weak point, however, related to the calculations is handling more delicate situations. For example, simulating the movements of the Earth-Moon system around the Sun works fairly well when the timestep is small, but when increasing the timestep, the Moon is very easily thrown off its orbit around the Earth. This isn’t a serious issue, though, since adjusting the timestep on the go is very easy.

A different weak point with the calculation is the difficulty to conduct ‘scientifically relevant’, simulations. By this I mean scenarios where you would, for example, want to simulate launching a satellite from the earth to orbit the moon. However, I feel this is more of a weak point from the perspective of the initial project description than from my own perspective. During the project, for me, it felt more important to visualize the broader, longer term effects of gravity than single scenarios of astrodynamics. This could be changed by making different scenarios more clearly separate by, say, limiting the working area of a scenario to the confines of the area relevant to the scenario. For example, in an Earth-Moon scenario the program could be limited to only displaying the Earth-Moon system and going beyond that would require creating a new scenario.

Another strong point, deservedly so, is the usage of the interface. All the features of the program are, at most, a few clicks away and using them is intuitive. Especially adjusting the view of the simulation still feels surprisingly easy.

The fluidity of file input and output is very much a strong point as well. Saving and loading simulation files works smoothly and doesn’t break down if there’s something wrong with the files.

A slightly weak point is the graphics. They definitely are not bad but as someone keen on the visual side of things, there are improvements I’d make if there was extra time for that. Especially some of the default UI elements, such as the buttons and sliders, have a very crude look to them.

Deviations from the plan, realized process and schedule

In the beginning, the progress was nowhere near as fast as was planned. The first version of the calculator was, indeed, implemented fairly early (still not in the first two weeks like was planned) but it was more of a proof of concept than anything even close to the final version. As stated before, it was implemented based on Kepler’s laws, and was scrapped later as I realized that it’s redundant. What made implementing the calculator difficult was that a lot of reading about celestial mechanics and astrodynamics had to be done to get an understanding of what I’m trying to implement.

Work on the GUI started fairly fast. However, as time went on, it became evident that the GUI would not be 3-dimensional. It would probably look very neat that way, but honestly, it’s very irrelevant to the actual usage of the program. Still, a lot more time was spent on the GUI than was perhaps intended at first. This is largely due to having to learn to use the Scala Swing library without much online guides. Retrospectively, going with another library for the user interface would probably have been smarter. Most of the GUI work was done between week 4 and week 7. File I/O was implemented during this period as well, which is in line with the plan.

Reaching week 7, I finally realized I’d have to overhaul the calculation system. I bit the bullet and did some more reading on astrodynamics. Week 7 was spent changing the calculation to being vector based and in the form it now has. Week 8 and 9 were largely as they were planned.

In the end, there wasn’t much deviation in the order of implementing features other than the late finalization of the calculator. However, a large deviation from the plan was made in the amount of time that was used in the project. The schedule laid out in the plan would’ve amounted to a maximum of a little over 100 hours. The real amount of time used was probably somewhere between 200 and 250 hours.

Looking back, it seems that, had I used more time early on in the project simply reading about celestial mechanics and astrodynamics, I could've saved a lot of time by not doing things that were scrapped later. Of course it's hard to say whether that would have been the case since a lot of the understanding of the subject matter came while doing the things that were scrapped.

Final evaluation

The end product, I would say, is a well-working simulator with a polished feel. Some obvious issues emerge if it was tried to be used for scientific purposes. However, the direction of the program was always more of a visual and pedagogical one instead of a scientific one. Of course, this doesn't close away the possibility of the simulator being more scientifically accurate. That's a direction the possible future improvements of the program could take.

The overall structure of the program is solid. Still, some improvements could be made in terms of individual parts of it. For exampel, as mentioned, a new data structure could be made extending the Vector class with the basic methods of vector calculation. This would make it easier to read the code and expand the program overall. Perhaps the Pos classes could then be scrapped as well and all calculation could be done with the new vectors. Also the UI package object could be made more coherent by generalizing constructor classes.

Further development of the program could be done easily. Since the start of the project, I've had the idea of continuing its development after the course. This is why the program was designed with room to expand. The methods in use were mostly built to handle general cases instead of specific ones. Also the interfaces between different parts of the code are very clear which makes adding features simpler.

Starting the project over, I'd plan a lot more specifically how I'd implement the calculation in the simulation. Perhaps, generally speaking, I'd spend more time during the planning phase and not jump right into the 'fun' parts.

References

[1]: Comparison between methods for orbital perturbation
https://www.sciencedirect.com/science/article/pii/S0895717707003433

[2]: Using fourth-order Runge-Kutta for orbital modeling
http://spiff.rit.edu/richmond/nbody/OrbitRungeKutta4.pdf

[3]: Solar system data from JPL:s planetary ephemerides:
https://ssd.jpl.nasa.gov/horizons.cgi

[4]: The precession of Mercury's orbit
https://en.wikipedia.org/wiki/
Tests_of_general_relativity#Perihelion_precession_of_Mercury

Useful links

Fourth-order Runge-Kutta:
https://en.wikipedia.org/wiki/Runge-Kutta_methods
https://gafferongames.com/post/integration_basics/
https://www.haroldserrano.com/blog/visualizing-the-runge-kutta-method

Celestial and orbital mechanics:
https://arxiv.org/pdf/1609.00915.pdf
https://encyclopedia2.thefreedictionary.com/Orbit+of+a+Celestial+Body

https://en.wikipedia.org/wiki/Celestial_mechanics
https://en.wikipedia.org/wiki/Orbital_mechanics

https://physics.stackexchange.com/questions/456808/how-do-computers-solve-the-three-body-problem

Libraries used:

Scala standard library:
https://www.scala-lang.org/api/current/

Scala swing:
https://www.javadoc.io/doc/org.scala-lang.modules/scala-swing_2.13/latest/index.html

Java.awt:
https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html

Javax.swing:
https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

Inspiration:
http://universesandbox.com/