Solar system simulation project

CS-C2120

Personal information

Ukko Miettinen, 793414

Aalto SCI, Computer Science, 1st year Bachelor's programme

5.5.2021

1. **General description**

The program is a crude simulation of our solar system. It contains a graphical view of the system animated in real-time with definible time steps. The program's ui also consists of a tab-section where the user may review some data about the bodies of the system, change the view options of the graph and finally add a custom body into the system with user-defined values. In this tab there is also the option of launching a satellite (also with user-defined values) from an arbitrary body, f.e. Mars or another already launched satellite/station. One can also set the time step reasonably high and observer the movement of the bodies while eating breakfast.

The implementation had different possible difficulty-levels, and I think my level was the moderate one – an implementation with a graphical user interface with vectors. I did try to implement a higher order algorithm for calculating the movements of the bodies, though in the end I didn't quite manage to get any to work properly without being too consuming on the cpu.

2. **User interface**

After starting the program, the user is greeted with a view of the system with the sun in the center. The user can then move about the graph by using the WASD-keys. By using these keys, the graph is moved by a set amount of pixels. This amount can be changed in the view-tab ('Change pan-level'). The user can also zoom outward or inward by using the up- and down-keys. If the user ever gets lost in space trying to search their ever-lost first satellite, they can always press the 'Reset view' -button in the View-tab, which will move to user back to the sun. If the keys don't work, try clicking on the graph to regain focus.

The tab-section consists of three tabs: data, view and add/launch. The data-tab is filled with the data of the bodies; mass, distance from sun, acceleration and velocity. The view-tab consists of all kinds of options. There is a pause/play-button for the simulation, the mentioned reset view -button and a reset time -button for resetting the simulations count of days. Three checkboxes are also included - 'Show names', 'Show acceleration vectors' and 'Show velocity vectors' – with obvious functions. Then there are three input-fields where the user can type in a value and set it by pressing the fields' respective 'Set'-Button. These are where the user can change the graphs time step, pan-level and zoom-multiplier. The time step -input changes the timeStep-variable of the Space-object, which affects how fast the

objects move in real-time or f.e. how fast a year goes by. The pan-level-input changes the amount of pixels the graph is moved when pressing WASD-keys. The zoom-level-input changes the zoomGain-variable that multiplies the current zoom-level of the graph. The top left corner of the graph shows the current zoom-level (meters converted to pixels), pan-level and time (in days). The final toggle button changes whether or not satellites draw trails behind them.

Finally, the add/launch-tab consists of the different variables the user can set before adding a body or launching a satellite. For adding a body, the user must first define a name, mass (kg), radius(m), drawn radius (pixels, this is the circle drawn into the graph), distance from sun, direction from sun and velocity, which is to be inputted in vector form. These inputs are quite strict, so in case of a typo, f.e. in the velocity-field, the body simply won't be added. The direction from sun -input will be also given in vector-form and will be normalized in the process. This direction joined with the distance from the sun will give the body a location in space. The drawn radius will define the radius of the drawn circle, while mass will only affect gravity. Furthermore, no two planets shall own identical names, since the process works with ids that use the bodies' names. The same goes for satellites. By clicking on the 'Add body' -button the body will take the input-values and the program will add it into the space aswell as the graph. One can easily check the data-tab for distances and masses of the other planets for some directional guidance on the variables. The direction will work the same way it does on a regular gui, that is +x is to the right and +y is downward. The GUI will only use z-values of the bodies by deciding which bodies will be drawn on top or below eachother. Though nothing is stopping the user from adding a body that has an orbital rotation on the z-axis, it will just not look as pretty and its orbit harder to inspect.

Launching a satellite is similar to adding a body, though some inputs are different. Satellites are launched from bodies, which is why one input will ask the user where from the satellite will be launched. This input will take a body's name, f.e. Earth or Saturn, and set the satellite to be launched from that body's surface. The direction input works the same way, and will define the direction the satellites velocity. Finally, the velocity will not be inputted as a vector this time, as the direction of the satellite is already known. The input is just a simple value in m/s. Be sure to input a high enough velocity for the satellite (depending on the body), since this simulation doesn't have external thrusters on the satellites, which means the initial velocity is all the power the satellite will get.

Adding a body or a satellite will also add their respective data into the data-tab aswell, so the user can for example check how far their fast satellite has travelled.

3. Program structure

The program consists of three main classes and an application object. The structure of the physics is divided into these three classes: Space, AstralBody and Vector3. Vector3-class is a class for a three-dimensional vector and has two important methods: magnitude and normalize, which are both needed to calculate the magnitudes of accelerations and velocities and their directions.

AstralBody-class describes a body. It keeps all the important values of a body and has methods that update its values. UpdateDirection, updateAcceleration, updateVelocity and updatePos are called respectively to update the bodies values with the help of the Space-class.

The Space-class describes the universe of the bodies. The class contains universal values such as the gravitational constant and time-values. It keeps track of all the bodies in it as a Vector[AstralBody]. Space also contains the other key-methods for calculating the gravitational infrastructure between bodies. Using the methods distanceBetweenBodies, directionAtoB, gravityVectorFromAtoB the class can calculate all the gravities for all the bodies in it, which it does by calling the method updateGravitiesForBodies, which in turn updates the AstralBody-object's allGravities array. Finally the method advanceStepInTime will call all the bodies' updating methods and therefore change their space.

The GUI-application object 'App' will of course implement all this. It will draw all the bodies as circles and satellites as small dots. The application uses the added AstralBody-objects' values to calculate their position in space and keep track of their data (printing it in the data-tab). The app also uses an AnimationTimer-class to animate the graph, in which the Space-object's method for updating everything is called during every tick.

I think the three class -system for calculating the physics was the best choice, since it kept things in some sort of order while not cranking too many lines into the same class. The Vector3-class is pretty small, though I do not regret it being kept separate from the others as an own class, since it is a completely different function of a class after all. ScalaFX also has a Point3D-class though it lacks all the necessary methods I needed.

4. **Algorithms**

I mentioned that I didn't include a high-order calculation method for updating the positions and velocities. This was because I couldn't get them to either work properly, or they were too processor-consuming. This was why I ended up with using the simple Euler method:

(v_i+1 = v_i + dt * f(x_i, t))

The basic order of calculating change in space:

updateGravitiesForBodies() >> updateDirection() >> updateAcceleration() >> updateVelocity() >> updatePos()

All of the updating method are done for each body when the time is increased by the time step.

The Eule- method is a simple and effective/fast way of calculating the changing velocity/position of a body. The negative side is that it is not as accurate as the higher order methods, and with super-high time steps the results will become heavily inaccurate, even changing the bodies' orbits drastically. Though no problem is had if the time steps are kept reasonable.

Other calculations done in the AstralBody-, Space- or Vector3-class all use basic matrix/vector-calculations (addition/multiplication). The Space-class uses Newton's gravitational law to calculate the forces/gravities.

The GUI's zooming-function uses the user-defined variables, and divides the distances of the bodies by this variable. The bodies' pictures are also divided/multiplied by this factor.

5. **Data structures**

   I used mainly Vector-types of arrays to store data. There is no particular reason since my program doesn't require to use the full size of these structures.

6. **Files and Internet access**

   The program doesn't use Internet nor external files.

7. **Testing**

   The testing was done mainly by unit testing and by GUI. Unit tests were used to calculate the accuracy of the physics and to check if they were right in different simple scenarions, that I could first calculate and solve by hand. I mainly used these tests when I tried to implement higher order methods, and I included a lot of println()s into the code to see where everything would vary.

   After that I tested the movements in the test-GUI to see how well the bodies would move (f.e. their speed on the screen with different distances/time steps etc.). I used this GUI also to build first tests of the actual application's GUI. This was exactly how I originally planned to test the program, and there was no sudden uphill on this part, since the actual calculations are not that exotic.

8. **Known bugs and missing features**

   While adding or launching a body/satellite, the inputs are not actually checked to make sure they are of the correct type. The button will just not simply work if the inputs are wrong. I think in the future I might implement some sort of a pop-up that informs the user that some input was wrong. Also the vector-inputs could be solved a bit differently, so f.e. that there's no need for whitespace.

   The panning and zooming will also remove the trail of a satellite. This was because adding the trail into the panning- and zooming-functions was more troublesome for me (for some reason) than just simply cleaning it out.

   The pictures of the bodies will also disappear for a frame when pausing and resetting the view. Also panning and zooming is not possible while having the simulation paused, since they use the AnimationTimer.

   The bodies will also hover over the view-tab and add/launch-tab. Perhaps this is a feature…

   The GUI also misses a feature that updates the satellites when they collision with another body. At the moment they just gain a huge velocity from the increased acceleration and explode into the abyss. (Kind of cool?) Perhaps in the future I will implement a 'Destroyed'-status that is updated once the satellites collide with something.

   As I mentioned before, the problem of bodies having the same name exists. Though I presume this will only affect those who forget to change the name when they add another body/satellite.

9. **3 best sides and 3 weaknesses**

Weaknesses:

1. I think the biggest weakness of the project is that it uses only the simple Euler-method for updating data. This will become a familiar problem once the user figures to test unimaginable time steps (for fun: test setting the time step to 1000000).
2. For now, the unexperienced user will mostly have to guess appropriate values for satellites/bodies when adding/launching them. Perhaps in the future I might add a feature where the user can drag the body into the graph, which will then draw the calculated path of the body, so that the user can see how well/badly they've guessed some values. (User could see if the body would be sucked by the sun instantly or not.)
3. The panning and zooming might not be enough, and I would personally really appreciate a function of locking onto a body and following it, instead of constantly having to pan all around. (I tried to implement this but failed ultimately.)

Positives:

1. Personally I'm really happy with how the UI turned out in the end, as it does represent how I pictured it from the start. The tabs and the overall view of the graph is elegantly simple, in my humble opinion.
2. The zoom-function was a headache for me for some unknown reason, but once I was finally able to implement it I think it really paid off. I really like the way the user can change the parameters for zooming or panning.
3. I think I managed the launching of a satellite / adding a body -part of the program reasonably well, and I do like the system that the user can launch the satellite from a certain body. I think the satellites flying out of a body with the included trail looks nice. The other tabs also worked out pretty great.

2. **Deviations from the plan, realized process and schedule**

Everything about the program's plans went as I originally figured except two things: my own schedule got ruined for a while and the implementation of a 3D graph. The first one is partly because of my own personal happenings and some stuff I couldn't really affect, though mostly because my priorities were on different tasks.

The 3D graph was a real headache, though. At first I couldn't find a great library I could use to implement the picture of the program I had in my mind. Then I found a java library that I might be able to implement (jzy3d), which didn't really do what I originally wanted (granted, my imagination might've been too grand). I tried to learn to use it, but I couldn't figure out how to, since the documentation for it was outdated and quite small of quantity. In the end I also thought that the whole 3D graph -thing might be too processor-consuming, and that in the end I couldn't even really use it. This is why I decided to ditch the whole idea and just focus on the 2D view. And to be clear, I'm mostly annoyed about the time I spent on the idea, and in the end I'm glad I focused on the 2D part more.

Otherwise the program looks exactly how I planned from the beginning.

3. **Final evaluation**

   I've mentioned all the shortcomings I can remember/tell in earlier sections, aswell as the ways I could improve on them. Though what I didn't mention that one way I could add into the projects quality would be to include a text file that lists all kinds of bodies of near our system, that would then be read from it and added when launching the application. This way one wouldn't have to manually add all the bodies in the code, and one big list would suffice (granted, one that is consistent in its format).

   All together, I think I've managed to create a fun little simulation program. One that I can improve later with various quality of life improvements, and maybe a working implementation of a high order calculation method for super increased accuracy.

   I think if I could start from the beginning, I would instantly ditch the 3D idea, and try to focus on the quality of life improvements on the 2D graph.

4. **References**

   I mostly just read the JavaFX and ScalaFX APIs.

5. **Appendixes**

   Source code will be added.