# Solar Simulator

Name: Samuli Öhman

Student number: 907815

Study Program: Tietotekniiikka 2021

date: 23.04.2022

## General description

The application will load information about the planets from a file when starting. The file contains the names, masses, radii, colors, locations and velocities of the planets in the simulation. Next the simulation will start with the default time step size and length.

The user can interact with the simulation adding planets, changing the time step, changing the camera position, and saving the state of the simulation to a file. The project has been done in the most difficult version as it has both GUI, and the calculations are done accurately using the Runge-Kutta $4^{th}$ order algorithm.

## Operating instructions

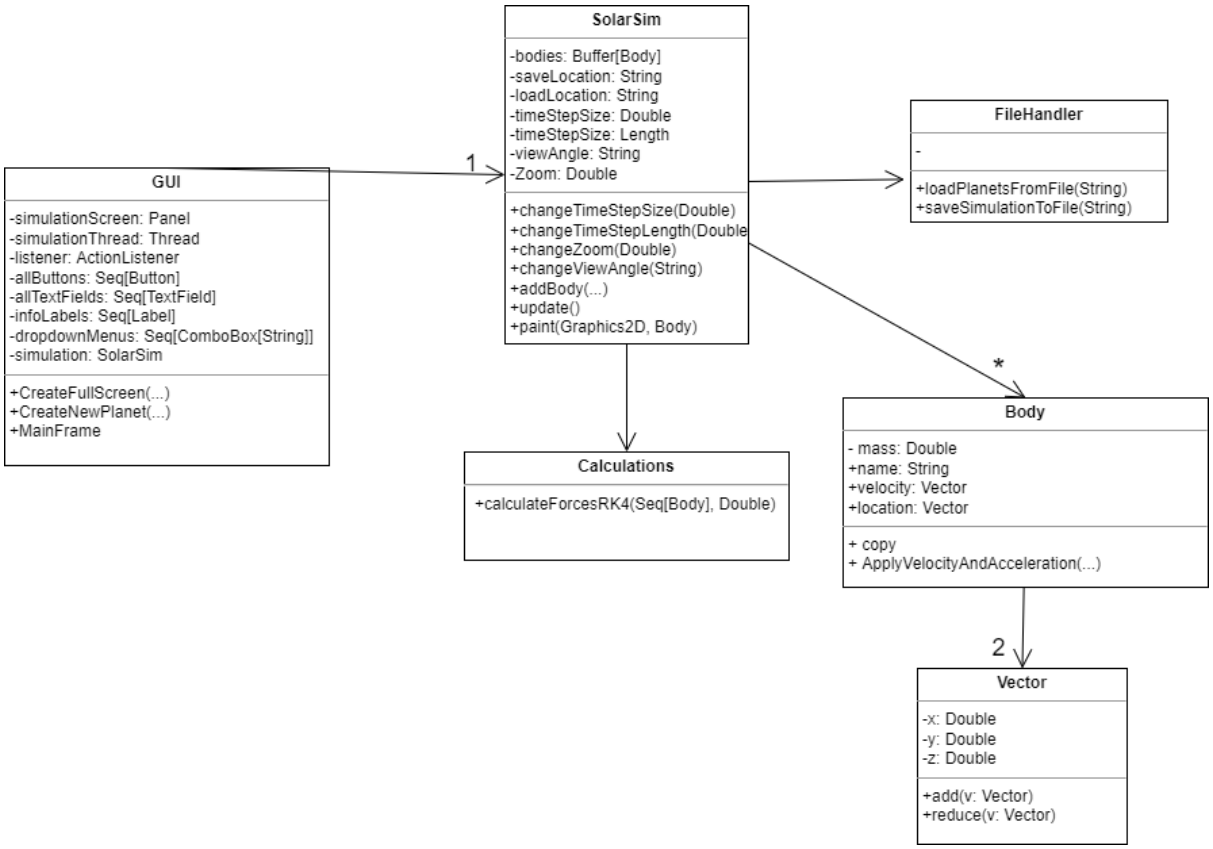
Before starting the simulation, the user can change the initial condition of the simulation. These include the time step size and length, save and load files' paths, zoom and view angle. After these are set to desired values the simulation can be started.

The GUI consists of the 'left bar', the 'top bar' and the 'simulation screen' (Image in attachments). Simulation screen occupies most of the GUI and it's the place the planets are drawn. The user cannot interact with the simulation screen. On the 'top bar' the user can change the view angle and the time step size and length. Each button that changes the view corresponds to an angle that views the plane specified in the button, for example 'xy' or 'yz'. The dimension that is not specified in the button (for example dimension z in button 'xy') corresponds to the depth dimension which cannot be seen on a 2-dimensional screen. On the right-hand side of the top bar, the user can change the time step size and length. Both inputs must be in double form. The time step size means seconds the simulation will pass each step and the time step length means millisecond between steps. When time step length is set between 1-10(ms), the user cannot notice the difference as the Thread.sleep method takes at least few

milliseconds to complete. When the time step is set to 0(ms) the simulation runs as fast as possible. The time step in previously mentioned scenario is around 0.125ms.

On upper part of the ‘left bar’ the user can select which planet’s location, velocity and mass are seen below the upper dropdown menu. The selected planet is also drawn in the center of the simulation screen and all other bodies are drawn relative to it. Below the information of the currently selected planet, the user can add a new planet/object to the simulation. The second dropdown menu on the left bar specifies relative to which the inputs are typed. For example, if an object is created in location (384000, 0, 0) and Earth is chosen on the dropdown menu, the object appears approximately the same distance away from Earth as the Moon is. The inputs for mass, location and velocities must be doubles and the units are km for location, km/s for velocity and kg for mass.

## Program’s structure




The program is divided into six different classes and objects: GUI, SolarSim, Calculations, Body, Vector3D and FileHandler. In addition, there is a folder which contains the different save files. The GUI is the object which runs when the program starts. When starting the program, the GUI objects creates an instance of the Solar Sim class, creates the screen with all the buttons and boxes, and

creates another thread for updating the simulation. The GUI objects is also responsible for listening to all the buttons and dropdown menus, updating the information on the screen, excluding the drawing of the planets which happens in the simulation class. The GUI class is divided into two parts. The upper part (around first 170 lines of code) contains the methods that are actively used when the program runs. The lower part (after the line 170) contain the creation of the GUI which is only used once when the program starts, and it means creating each individual element in the GUI and positioning it in the correct place.

The SolarSim class contains the variables that affect the simulation, and the methods for updating and drawing the simulation. For example, if time step size is pressed on the GUI, it calls a method on the SolarSim class which changes the variable that is also in the SolarSim class. Calculations class contains the Runge-Kutta $4^{th}$ order algorithm, and it is used to calculate new location and velocities for the planets. Calculations object's methods are only called in the SolarSim's update method. The methods in the SolarSim class are accessed by two different threads: drawing thread uses the draw method, and simulation thread uses the update method. This means that the status of the simulation can change in the middle of another thread's method call. For this reason, at the beginning of the draw method, the drawing thread creates a copy of all the planets and draws the copies in order that the locations of the planets don't change while drawing them. In addition, when new planets are added to the simulation, they must be added by the simulation thread because otherwise a new planet could be added in the middle of calculations which would cause errors.

The Body class represents the planets used in the simulation. Each instance of Body has the following attributes: name, mass, radiusReal (used for collision detection), radiusLarge (used for the size which is drawn on the screen), location, velocity and color. The case class Vector3D is used for storing information about the 3-dimensional world as it has attributes x, y and z. The Vector3D class also contain useful methods for adding, subtracting, multiplying and dividing vectors. Finally, The FileHandler object is used to read from and write into a file. The SolarSim class loads the initial state of the simulation from a file using the FileHandler when the program starts.

**Algorithms**

The most important algorithm used is the Runge-Kutta $4^{th}$ order. It calculates the velocities and accelerations of the planets four times. The calculations are done in three different times: now and in two different times in the "future". Each "future" calculation is done using the velocities and

accelerations calculated in the last step. Finally, the algorithm adds all the different results together using different weights for each.

$$k1 = evaluate(initial, 0, Derivative(0,0))$$

$$k2 = evaluate(initial, \frac{dt}{2}, k1)$$

$$k3 = evaluate(initial, \frac{dt}{2}, k2)$$

$$k4 = evaluate(initial, dt, k3)$$

$$Final\ Derivative = \frac{1}{6}(k1 + 2 * k2 + 2 * k3 + k4)$$

In these calculations the initial means the state of the simulation when starting the calculations and it stays constant during the calculations. The evaluate method returns the change in position (velocity) and the change in velocity (acceleration) as a Derivative (v, a). When calculating the k1 the Derivative (0,0) means that in the first time step the change in position and the change in velocity is zero. Finally, all the different derivates are added together forming the Final Derivative.

The evaluate method uses the vector form of the Newton's law of universal gravitation to calculate the acceleration for the planets. The formula used is following:

$$F = -G * \frac{m1 * m2}{\left|\underset{r2}{\rightarrow} - \underset{r1}{\rightarrow}\right|^3} * (\underset{r2}{\rightarrow} - \underset{r1}{\rightarrow})$$

$$F = -G * \frac{m1 * m2}{|rd|^3} * \underset{rc}{\rightarrow}$$

$$a = \frac{F}{m}$$

$$a = -G * \frac{m1 * m2}{m1 * |rd|^3} * \underset{rc}{\rightarrow}$$

$$a = -\frac{G * m2}{|rd|^3} * \underset{rc}{\rightarrow}$$

The evaluate method goes through each planet and sums together all the forces that other planets have on the current planet. Basically, the loop goes through N planets and calculates (N-1) forces that act on each planet. The evaluate method has a time complexity of O(n^2).

Other algorithms to mention are the algorithm used to calculate collision and the algorithm to draw the simulation from different angles. The collision algorithm calculates the distance between the center of the planets and compares it the sum of the planets' radii. If the distance is smaller than the sum of radii, the planets collide and the simulation stops. The algorithm used to calculate the pixel coordinates of the planets from different angles considers the width of simulation, the zoom and the angle and returns a pixel location that can be drawn. The planets are also drawn in correct order so that the planet "behind" another planet is drawn beneath the one who is on top.

## Data structures

The main collections used in the simulation are Buffer, Vector and Seq. Buffer is used where the collections size is not constant, for example planets are stored in a buffer because the user can add planets to the simulation and by doing so, change the collections size. Vectors and Seq -types are primarily used as local collections that are created only for the length of a method. In Calculations object Vectors are mainly used as they have a constant time for applying, updating, and appending. In calculations especially apply method is used often for the collections, and the calculations should be done as efficiently as possible. As a result, collection with constant time in methods is the best.

The Vector3D class uses Doubles to store information about planet's location and velocity. Doubles have a precision of 15 to 17 significant decimal digits which should be enough for the accuracy of our simulation. If the user would like to make the simulation even more precise, the data types should be changed from double to something more accurate. All the calculations for the velocities and accelerations are also done with double-precision.

## Data Files

Because of the small quantity of data that needs to be stored, I opted to use string format for easier readability and manual editing. This means that the data is not stored as efficiently, and it might take more storage space. In the end. it shouldn't matter meaningfully as we're only storing a few hundred characters.

The files in format in a way that each line of text creates a single planet to the simulation. A single line is formatted in the following way: (name, mass, radius, posX, posY, posZ, velX, velY, velZ, color). The name is a string, and the color is an int. Rest of the attributes are doubles and each attribute is separated by a comma.

## Testing

The first thing that I created in the program was the GUI. This enabled testing planets' locations, velocities and sizes visually on the GUI. A simpler version of the Calculations object was also created in the beginning to get an idea about how the simulation should work. The Runge-Kutta method was one of the final things added and it was tested both visually and by printing the forces and checking the forces calculated during each step manually.

When the program was still running single threaded, I noticed that most of the thread's time was used to refresh the GUI, not the calculations of physics. I then decided that trying to optimize the speed of the calculations' methods would be mostly meaningless if everything ran in the same thread as updating the screen would use most of the CPU's time. After adding multithreading, the framerate for simulations jumped from around 250 fps to around 5000 fps.

## Shortcomings and bugs

One of the features that was supposed to be added but wasn't in the end, is the appearance of new planets in the dropdown menus. The Scala Swing's default dropdown menus are immutable meaning that new elements cannot be added to them. The dropdown menu is also fixed to the screen meaning that creating a new immutable dropdown menu with different objects requires that the whole screen's layout would have to be created again in order to add the new dropdown menu to it. A fix to this problem would be to create a custom mutable dropdown menu which allows the addition of new elements.

Another feature that could be perfected even more is the size in which the planets are drawn. Because of the distances in space are so large compared to the radii of the planets, the planets must be magnified before drawn. I just made up a magnifying algorithm of my own which has a bug that

sometimes when zoomed in close enough, the planets seem to visually overlap when actually they might be far apart.

## Best and worst features

- Best:
    - The ability to change camera angles
    - The framerate that multithreading enables
    - Intuitive design of the GUI
- Worst:
    - Planets are just colored circles, not images or 3D objects
    - Camera cannot follow new planets as they're not in the dropdown menus
    - The code in the GUI class is formatted in confusing way and some of the GUI's code could and should be in other classes

## Timetable

The timetable was mostly followed. The order in which the classes were created and perfected was done according to the timetable in the initial plan, excluding the addition of camera angles before the Runge-Kutta algorithm. In conclusion, the work that was required to create each class was quite accurately estimated in the initial timetable before the project had even started. The amount of work that was done also varied considerably week-by-week.

## Conclusion

In summary, the project was well planned and executed. I set out to do the difficult version of the project with the criterion of having a GUI and accurate calculations, and I succeeded doing both. I also had some additional features in mind before starting the project, for example different camera angles and zoom, and I also managed to add them. Some features which were not required, mainly multithreading, I decided to add during the project. An extra class, that was not in the original plan, was added to handle reading from and writing into files to make other classes more readable.

One improvement for the simulation could be redoing the visual appearance. During the project, I prioritized function over looks. Using Scala Swing, one can only do the visual appearance to a certain degree and that's why I decided that I wouldn't try to make my program the most beautiful it could be. Another improvement could be to use more precise data types for storing the data. This could improve the accuracy of the simulation in certain scenarios.

If I were to do the project again, I wouldn't change much. The biggest change would be the knowledge that the project would use multiple threads before starting. This would have made it easier to create certain methods and classes, knowing that the state of the instance could be changed by one thread in the middle of another thread's method call. Apart from that the classes were well designed and created in sensible order.

**References**

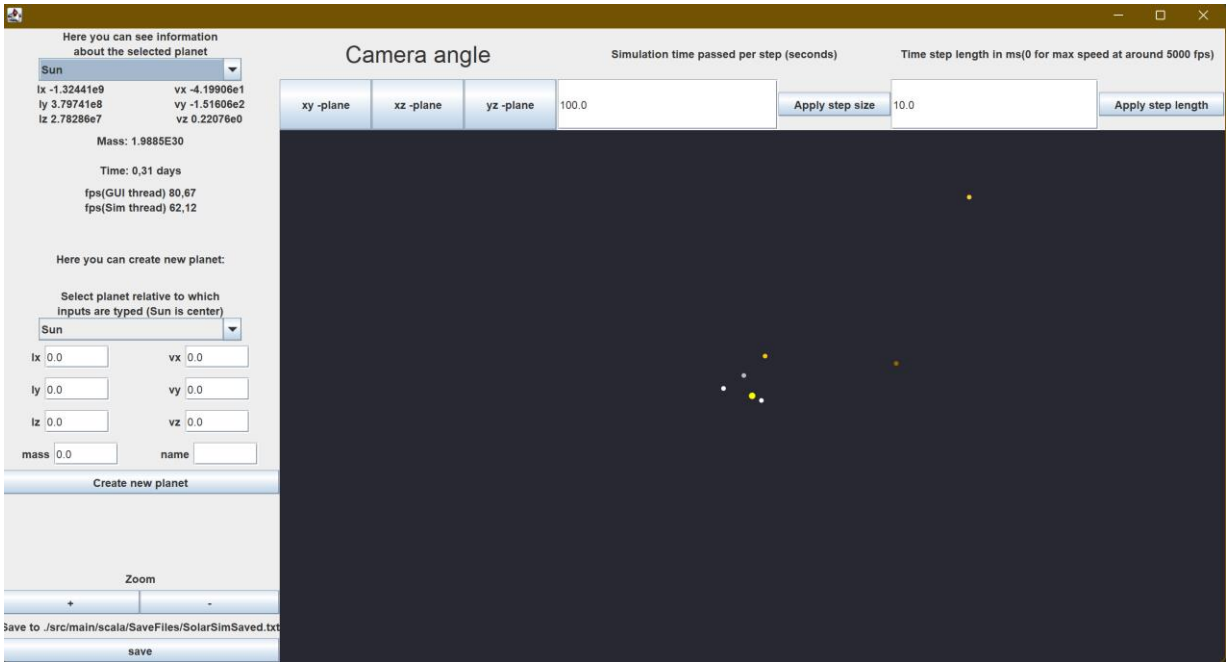
https://ssd.jpl.nasa.gov/horizons/

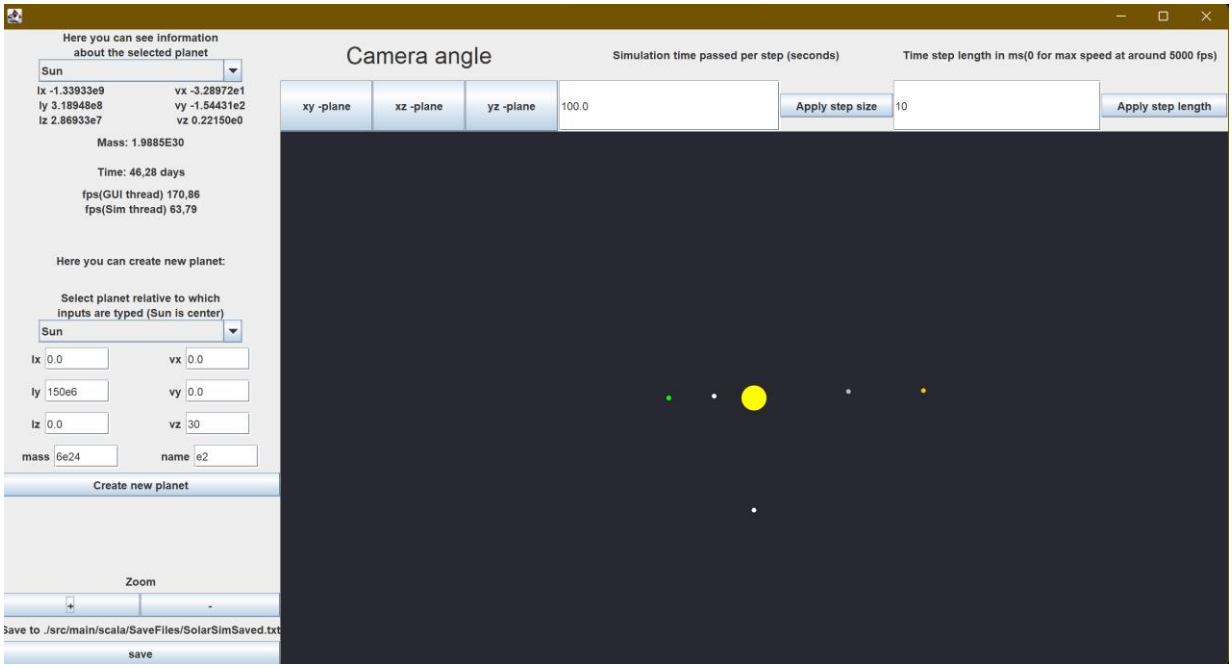
https://gafferongames.com/post/integration_basics/

https://docs.scala-lang.org/overviews/collections/performance-characteristics.html

**Attachments**

Images of GUI:


Here you can see information about the selected planet
Sun
lx -1.32441e9
ly 3.79741e8
lz 2.78286e7
vx -4.19906e1
vy -1.51606e2
vz 0.22076e0
Mass: 1.9885E30
Time: 0,31 days
fps(GUI thread) 80,67
fps(Sim thread) 62,12
Here you can create new planet:
Select planet relative to which inputs are typed (Sun is center)
Sun
lx 0.0
ly 0.0
lz 0.0
vx 0.0
vy 0.0
vz 0.0
mass 0.0
name
Create new planet
Zoom
+
-
Save to ./src/main/scala/SaveFiles/SolarSimSaved.txt
save
Camera angle
xy -plane
xz -plane
yz -plane
Simulation time passed per step (seconds)
100.0
Apply step size
Time step length in ms(0 for max speed at around 5000 fps)
10.0
Apply step length


Here you can see information about the selected planet
Sun
lx -1.33933e9
ly 3.18948e8
lz 2.86933e7
vx -3.28972e1
vy -1.54431e2
vz 0.22150e0
Mass: 1.9885E30
Time: 46,28 days
fps(GUI thread) 170,86
fps(Sim thread) 63,79
Here you can create new planet:
Select planet relative to which inputs are typed (Sun is center)
Sun
lx 0.0
ly 150e6
lz 0.0
vx 0.0
vy 0.0
vz 30
mass 6e24
name e2
Create new planet
Zoom
+
-
Save to ./src/main/scala/SaveFiles/SolarSimSaved.txt
save
Camera angle
xy -plane
xz -plane
yz -plane
Simulation time passed per step (seconds)
100.0
Apply step size
Time step length in ms(0 for max speed at around 5000 fps)
10
Apply step length