1. Personal information

Title: Solar system simulator

Name: Max Ulfves

Number: 732459

Program: Tietotekniikka

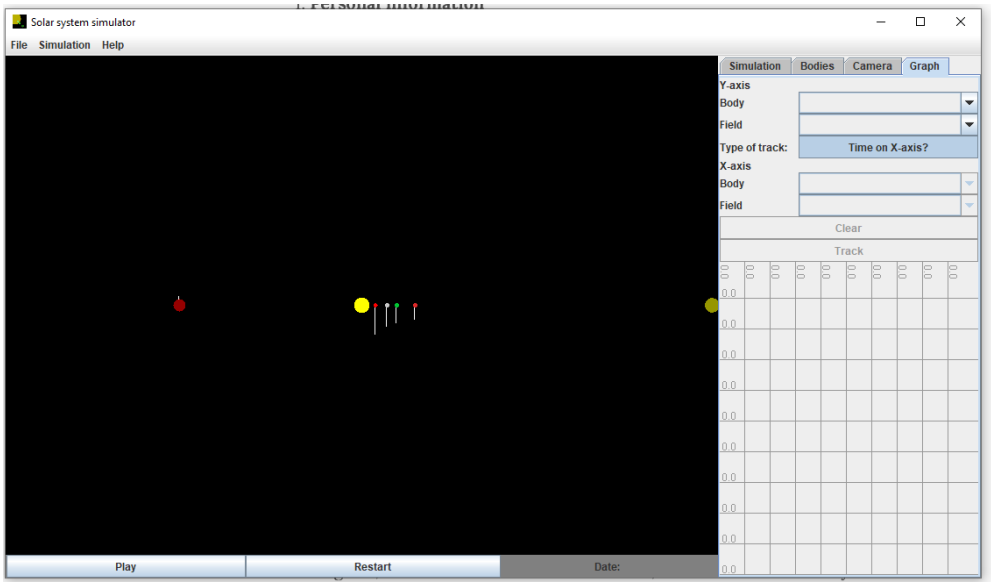
Year: 2019

Date: 23.04.2020

2. General description

I completed the project Solar system simulator. The program simulates the movement of bodies in a gravitational system, e.g. a solar system and displays them in a graphical user interface. The user is able to manipulate the properties of all bodies in the system, create new system with other bodies and store them In files for later use.

3. User interface



The program is started by running SimulatorApp.scala in the gui-package.

There is an instruction tab under Help > Instructions in the program with more detailed information, but to give an overview, in the bottom of the main window there is the time controller, which lets the user pause and play the simulation. The restart-button lets the user restart the simulation from the last save. On the right side of the controller is a label which displays the date.

The middle panel shows an image of the current state of the system, with distances in scale but the sizes exaggerated. The acceleration and velocity vectors are also displayed, but can be toggled off in the menu under Simulation > Toggle vectors. The user can rotate the system with the arrow- , punctuation- and comma keys to view it from different angles. The camera can be moved closer to the center and away with the scrollwheel. Make sure the centerpanel has focus before using keys or zooming. If the user uses a slow computer the stars can be turned off. They are meant to help the eye understand the directions though, but may be a bit resource intensive.

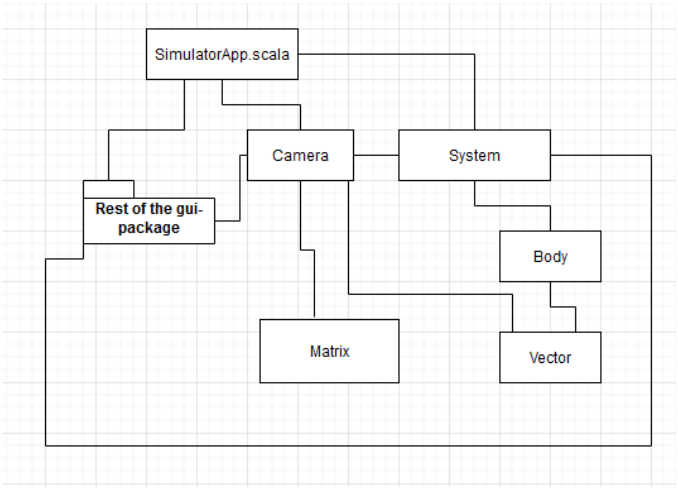
The simulation-tab displays general information about the simulation itself and is generally uninteresting. There you can also change the attempted simulations per second and frames per second. The default is 60. The Bodies-tab has a list of all bodies in the system and contains options to change the bodies properties or remove them completely. **Note that the simulation needs to be paused in order to make changes to text fields.** The same panel can also be used to view the properties of the bodies.

The Graph-panel contains an extra tool that I added. It's inspired by https://scatterplot.online/ in the sense that it can plot an individual field (eg. Sun's X-velocity) against time or a field of another body, including itself. When a simulation is running, fill in the fields that you wish to plot and press track. If you wish to have the time on the x-axis make sure the toggle button "time on x-axis" is selected. To clear a previous track, press clear. The cache is automatically cleared when significant changes are made to the system by the user.

To save the state of a system, press File > Save or File > Save as… depending on your intention. To load a system, use File > Load… and select a .gal file. Alternatively, to load an example file, use Simulation > Examples > and chose a file you wish to load. There are several example files demonstrating the functionality of the program. To make a new system the user uses File > New… which opens a dialog into which the user fills the name, starting date, simulation length and timestep.

4. Program structure

The image on the right is meant to show an idea of how the class/object-structure work. It's not meant as a complete UML, but simply an overview. Briefly, changes made in the GUI affect the camera and system classes, which alter their own intter states.



I'll start by describing the logical part and then describe how the GUI is structured. A simulation state is described by the System-class. It contains several bodies, described by the Body-class. It was originally planned to make Body abstract and let the classes Star, Planet, Satelite extend it, but the further I got the more I realized that it was unnecessary and would instead reduce the functionallity. A body can instead have any color, so the user is expected to identify them based on that. A body has the methods passTime(system) and updatePosVel(), of which the aforementioned calculates the change in velocity (using RK4) introduced by the given system and the latter updates the body's location parameter.

The RK4-calculations, the rendering and the UI are all their own separate threads, defined in the simulationCanvasPanel-object.

The system calls those methods (explained in more detail later). The system is in turn owned by the SimulationApp-object. The system has a property for the time difference, which is the time with which it updates itself when its method update() is called by the application. The application also stores a copy of the system as it was during the last save, which makes the user able to restart the simulation. The system's package also has a camera class, which is owned by the SimulationApp-object and is responsible for turing the a simulation-state into an image. It has several methods for rotation and zooming.

There is a Geometry (R3) package in the project, which contains some physical constants, a Vector-class, a Matrix class and a matrices-object which contains some rotation matrices. The vector-class is almost exactly as presented in the technical plan. The matrices object contains translation and rotation matrices and the matrix-class represents a matrix.

I'm generally quite satisfied with how that part turned out. The GUI on the other hand isn't as neat, but it has a logical structure as well. All objects exist in the package "gui". The SimulatorApp object is the main object of the application. It extends SimpleSwingApplication and has therefore naturally a

MainFrame-object which is in it's own separate file to make the code more readable. This object owns an object for every panel in the window, including popups. I thought it was clever to put them in their own files too. These are MenuBar.scala, HelpWindow.scala, MidPanel.scala, NewSystemDialog.scala, RightPanel.scala and TimeController.scala.

The most complex panel is the tabbed pane, or the rightside menu, which has several panels . If I were to expand the program any further than this I would subdivide this one as well, but it works fine at the moment. It currently has an object for every panel, which in turn have their own components.

5. Algorithms

The major algorithms used in the program is the algorithm for convering the system to an image and for updating the velocities in the system. I had a quite extensive geometry-package with lines, planes and points, but I realized during the final week that it was possible to replace them with two matrix operations so I did. Anyway, the matrixes first transforms the camera to be positioned at (0,0,0) and looking downwards and then transforms the rotation and position of all objects to be in relation to the camera. Then it projects them onto an image plane.

The algorithm is largely inspired by an opengl tutorial om matrices which I partly followed. The radius of the spheres on the other hand is calculated with an algoithm that I found in a forum and made some changes to. When the bodies postions on the screen is calculated they are drawn as filled ellipses with their color specified in the objects themselves.

This operation is performed in a foreach loop over all bodies in the system. The list of bodies is first sorted by the bodies z-distance to the image plane, starting with those that are the furthest away, so that closer bodies are drawn on top of them. Bodies that are behind the image plane are not drawn at all.

I decided not to implement panning, but it's possible.

The algorithm for updating the velocities is based on an implementation of Runge-Kutta 4 and is performed in the Body- and System-classes. It first calculates the new accelerations of the whole system. This is done with RK4 in the Body-objects. RK4 derives an acceleration by making a weighed average of four vectors that are found by Runge and Kutta's formula.

After finding the new velocities the program checks if it has any bodies within it's radius plus the magnitude of it's velocities unit vector times the change in time. If that's the case it checks if it's on a collision course with the object in which case the simulation is terminated. If there are no collisions it just repeats until the simulation time runs out.

There are some matrix [6][7] and vector algorithms that aren't originally made by me, namely Gauss-Jordan elimination and vector and matrix cross product. They are mostly taken from wikipedia.

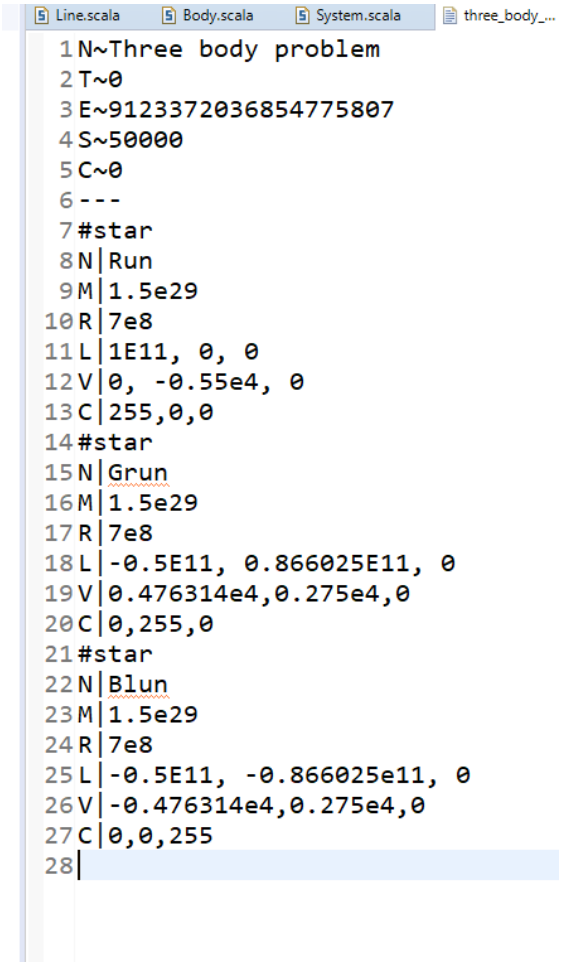
6. Data structures

I have described the body- and system structures quite throughly so far. I mainly use doubles in this project, but if serious calculations were to be made one should probably use BigDecimals instead as even the small rounding error of doubles are noticed and adds up over time. The system has a buffer for all it's bodies. This is because I wanted the user to add additional bodies during the simulation and remove them.

During matrix-operations I mainly use vectors and arrays as they are faster in the long run. The matrix itself is a sequence. I haven't really done any own data structures except the vectors which depends on doubles.

7. Files and internet access

The program doesn't use internet. There is one filetype, .gal which is a serialized System-object. The program isn't planned to support the user making his own files in a text editor, which is why it has fairly little error handling when reading files. It is possible though and in some cases easier if one understands the syntax. Example files can be found in ./Resources/Examples.



On the right hand side is a solution to the three body problem. Everything above "---" on line 6 describes the simulation itself. On the first line, "N~" means that what comes after ~ will be the name of the simulation.

T~0 means that current system time is 0 in unix time.
E~9123…007 means that the simulation will end when T is more than E
S~50000 the stepsize of the calculation in seconds. I've found that anything between an hour and a day is quite nice to look at, but it depends on the desired accuracy.
C~0 I was thinking about making a checksum, but never got to it.
--- Tells the program that what follows is the bodies

#star A relic from before I decided to not make bodies abstract. It wasn't necessary for the functionallity of the program to change it to anything else, so I decided to use the time other things. It separates the bodies from each other.
N|Run Name of the body. In this case "Run"
M|1.5e29 Mass of the body. Here 1.5 * 10^29 meters. Can also be written with numbers.
R|7e8 Radius of the object. Here 700000 km. Used to calculate collisions
L|1E11, 0, 0 The position coordinates expressed as X,Y,Z
V|[X],[Y],[Z] The velocity as meters per second
C|[R],[G],[B] The RGB-color of the object. Between 0 and 255.

Then it just repeats. There is no need for a blank line at the end. I want to reiterate that it doesn't handle faulty files.

8. Testing

I've done a few unit tests for the geometry package, but didn't really come up with a good way to do it for the rest of the classes and objects so I went with Facebook's motto and moved fast and broke things. I wrote a few examples such as collision.gal, which can be found in the examples folder. The solar system example represents the closest planets until Saturn in our system. With the mean velocity and distance to the sun. I concluded that they had the right orbital period given the slightly inaccurate data. I did try to find more exact position data to compare with, but I found that such data mostly uses angles to present positions.

All functionality like rotation and zooming has been tested to work as intended.

The RK4 method was tested when I first implemented it by predicting the movements of a sine-wave and comparing it to the real values, which it managed quite well. I haven't done a similar test after adapting it to calculate vectors though.

The overall performance has been tested on Windows 10, and Ubuntu 18.04 over an remote connection. When the application is working as supposed the initial example simulation should look completely smooth when opening the program.

9. Known bugs and missing features

I've added all the required features to the project, except the text UI which I'm told is unnecessary given a working GUI. In my original project plan I had planned to make a simulation file which would let the user "render" his simulation and then replay it at another speed. I didn't really have time to add it in the end, but it wasn't required either.

- The input fields work as described, but may be sensitive to some data. I have tried to put try-catch blocks around all textinputs, but I may very well have missed something. Also some silly variable values may be possible such as giving the sun a negative mass. Furthermore, I'm not sure what happens as certain vector values approaches the maximum value of a double. (e.g. when zooming). It should break something but I haven't found it.
- As mentioned earlier, the filereader is strict with the input data and doesn't handle faulty data well.
- The graph-tool has some trouble recognizing that its fields have been changed and the displayed fields may at some points be impaired with the actual data. This happens particularly when the bodies-buffer in the system-class is altered, either by restarting the system or adding/removing bodies. It's usually solveable by selecting and deselecting the fields and clicking around a few times. The bug is apparent in the end of the example_execution video.
- The program attemts to render vectors whose endpoints are behind the camera. This is partly because the render recongnize when bodies are behind a camera, but not vector endpoints. Usually they wouldn't be drawn anyway, but there may be some white or red flashes for a few frames or weird lines that become apparent when moving the focalpoint close to the center.
- I made quite a lot of late changes to the camera class, by implemeting matrices, so there may be a few unused variables that I've missed.

10. 3 best sides and weaknesses

Input management is probably the greatest weakness. It should tell the user better when invalid data is entered. The camera is both a strength and a weakness. It was a nice excersise to do it in 3d, despite not being a requirement, and I'm also quite happy with the result – it's interesting to look at. But it also opened up for bugs that wouldn't have occurred had I made a simple orthographic representation.

The geometric classes are also a nice implementation that saved me a lot of time and made the final code more pleasant for the eye.

The GUI is completely written manually which was quite tedious and I'm quite happy that I managed to make it look half-decent despite that. Despite that it could probably have been structured better and the comments are lacking and I had to mix javax.swing with scala.swing which I realized quite late into the project.

11. Deviations from the plan, realized process and schedule

During the first to weeks I had already a working model of the system with bodies rotating as they should and a 2d implementation of the graphics. My implementation only used implicit euler integration, so converting to 3d and studying numerical integration took a few more weeks at which point I had already deviated from the plan. So my original estimate matched reallity quite poorly.

12. Final evaluation

I am generally happy with the final result. It is scaleable and follows a clear design structure. The graphics are simple, but portrays what they are supposed to in a resource efficient way. Of course certain values could be tweaked, and some variables could be made into methods but overall I think I've learned a lot about 3d and numerical integration which I hope to put to use in the future. It was also nice to get some use of the matrices-course.

13. References

I was asked in the project plan to reference to the text. Most sources are not specifically mentioned in this repport but have been either used for studying a concept or idea, which is why I'll only include a description of those sources.

3d projection

[1] https://www.gamedev.net/forums/topic/545837-calculating-radius-of-projected-sphere/

[2] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

The most useful source in 3d graphics. Contains a very clear description on matrix transformations. So clear that when I found it I managed to remove five classes with around 500 lines of code and spead up the execution time ~10 times.

[3] https://math.stackexchange.com/questions/633181/formula-to-project-a-vector-onto-a-plane

[4] https://en.wikipedia.org/wiki/3D_projection

[5] https://math.stackexchange.com/questions/2305792/3d-projection-on-a-2d-plane-weak-maths-ressources

[6] https://fi.wikipedia.org/wiki/Gaussin_algoritmi

Contained a simple pseudocode example of the gaussian algorithm.

[7] https://en.wikipedia.org/wiki/Cross_product

Matrix crossproduct.

Swing & GUIs

[8] https://github.com/scala/scala-swing

The scala swing library, which I used for the main graphical structure.

[9] The course-textbook

There were some swing functions that I used in the beginning, but I've written over all of them I believe. Particularly the lama project were useful

[10] https://docs.oracle.com/javase/tutorial/uiswing/components/spinner.html

Java tutorial on spinners

[11] https://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html

Java tutorial on dialogs

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

Java documentation for swing

Gravity and numerical integration

[13] https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

[14] https://en.wikipedia.org/wiki/Backward_Euler_method

The above articles were mainly used to get a general idea of numerical integration.

[15] https://spiff.rit.edu/richmond/nbody/OrbitRungeKutta4.pdf

A good explaination of RK4.

[16] https://gafferongames.com/post/integration_basics/

https://gafferongames.com/post/fix_your_timestep/

https://gafferongames.com/post/physics_in_3d/

A ebook linked in the project description. I used the three aforementioned chapters as help in implementing the gravity methods in the Body-class.

[17] Walker, Jearl, Principles of Physics, Cleveland State University,2008,9thedition, Chapters:9, 13

Had some conent on optics and gravity that was useful.

Astronomy

The following articles were used to get some example data for testing.

[18] https://en.wikipedia.org/wiki/Lagrangian_point

[19] https://en.wikipedia.org/wiki/Three-body_problem

[20] https://evgenii.com/blog/three-body-problem-simulator/

[21] Maols tabeller 2011

14. Appendix

The project files are included in a compressed folder in this project. There is also a video showing an example execution. For some reason my screen recorder didn’t catch popup windows, so keep that in mind when viewing.