Solar System Simulator

Meri Henell, 1022375

Bachelor's Degree in Science and Technology – Data Science, first year

27.04.2022

General description

Create a program that simulates the movements of various bodies (planets, moons, asteroids, satellites etc.) in space by calculating their movement paths. The simulation will last until either a collision happens or the simulated time frame ends. Minor factors not related to gravitational forces (atmospheric friction etc.) may be ignored in the calculations but all movements due to gravitation and Newton's laws should be simulated as accurately as possible by using, for example, the fourth order Runge-Kutta method. The space of all bodies must be handled together.

All bodies in a typical solar system will have a certain velocity and location, even the Sun, although its starting velocity can be set to zero. Other bodies must have sensible starting values so that they will not simply fall into the Sun or escape the system as soon as the simulation starts.

The idea is to study a satellite or multiple satellites that start from given three-dimensional points in space with given velocities (with a goal of, for example, getting to the Moon, Mars or on Earth's orbit etc.).

The user must be able to define the starting position, mass and velocity of the satellites, the length of the entire simulation and the individual time steps of the simulation through a graphical user interface. The user must also be able to control any other possible parameters that are deemed relevant to the simulation. The movement of the satellites and other bodies should be portrayed graphically with appropriately scaled velocity and acceleration vectors. The starting states of the bodies can be read from a file. Pay attention to the user interface of the application (try to make inputting the different starting parameters as easy and intuitive as possible from a general use and testing standpoint) as well as the simulation computation efficiency. Real units should be used in the input and output of all parameters (m, kg, etc.).

I think the program includes all the requirements for the demanding level as planned.

User interface

The user has the possibility to add satellites to the simulated solar system by defining the satellite’s mass (kg) and position (m) and velocity (m/s) in vector form in three dimensions. The satellite is added after pressing the “Add satellite” button. The program notifies the user if the inputs are in wrong format. The user has also the possibility to change the default duration (d), time step length (h), and the JSON file from which the initial states of the bodies are read. The simulation is started by pressing the “Start simulation” button. Here the user is also notified if the inputs are invalid.

A simple text-based user interface is also included. The user can basically do all the same things as with the GUI, meaning set the duration and time step length and add bodies and satellites, but only the coordinates, velocities, and accelerations of the bodies are printed on the console. The user has to manually advance the simulation by specifying the number of times the bodies should be moved. The user is notified is the given command is not recognized.

Program structure

Firstly, a class was needed to model the celestial objects within the solar system. The original idea was to have a common abstract superclass that would be implemented by subclasses modelling the different types of celestial objects like stars, planets, asteroids, etc., but it was not really necessary to differentiate between these different objects, and thus, in the end, only a single Body class was implemented. If needed, it would be possible to add a variable type to the class so that the different types of bodies could for example be displayed differently in the GUI. The most important methods in this class are probably the move method, which updates the position and velocity of the body, and the draw method, which draws the body with arrows representing its velocity and acceleration. There are also methods for computing the distance between two bodies and checking if two bodies have collided. The initial plan was to also have the method that determines an approximation for the acceleration of the body in the Body class itself, but implementing a separate class for calculating the acceleration allows better usability and extensibility as new algorithms can be added fairly easily.

In addition to the celestial objects, the three-dimensional coordinate system had to also somehow be modelled so that the positions of the objects and the directions of the velocities and accelerations can be clearly expressed. This was done with a Vector3D class that models three-dimensional vectors and includes methods for determining the length of a vector and other basic vector algebra.

A SolarSystem class was needed to model the solar system that contains and controls the bodies. This is the base for the user interfaces and handles adding satellites to the solar system and advancing the simulation by updating the positions and velocities of all the bodies in the solar system. The most important method within in this class is likely the advance method, which determines an approximation for the acceleration using one of the classes implementing the trait IntegrationMethod, Euler or RungeKutta, updates the accelerations of all the bodies, and calls the move method for each body. There is also a method for advancing the solar system a specified number of times, which is used by the TextUI. In addition, there are methods for adding bodies and satellites, checking if the simulation is over, executing commands for the TextUI, and drawing the bodies for the GUI.

The GUI was built using the Swing library and its UIElements and event listeners and handlers. It displays labels, text fields, and buttons in addition to the bodies in the solar system, accepts inputs and notifies the user if they are in wrong format, reacts to button clicks, and redraws the solar system at regular intervals once the simulation is started. Similar actions can also be executed in the text-based user interface except that the simulation has to advance manually. Both the GUI and TextUI use the FileReader object to read the bodies from a JSON file and add the bodies to the simulated solar system. A separate object allows methods for reading other file types to be added fairly easily. The class InvalidInputException is used to handle wrong inputs from the user. Furthermore, the TextUI uses the class Action through SolarSystem to execute the commands given by the user.

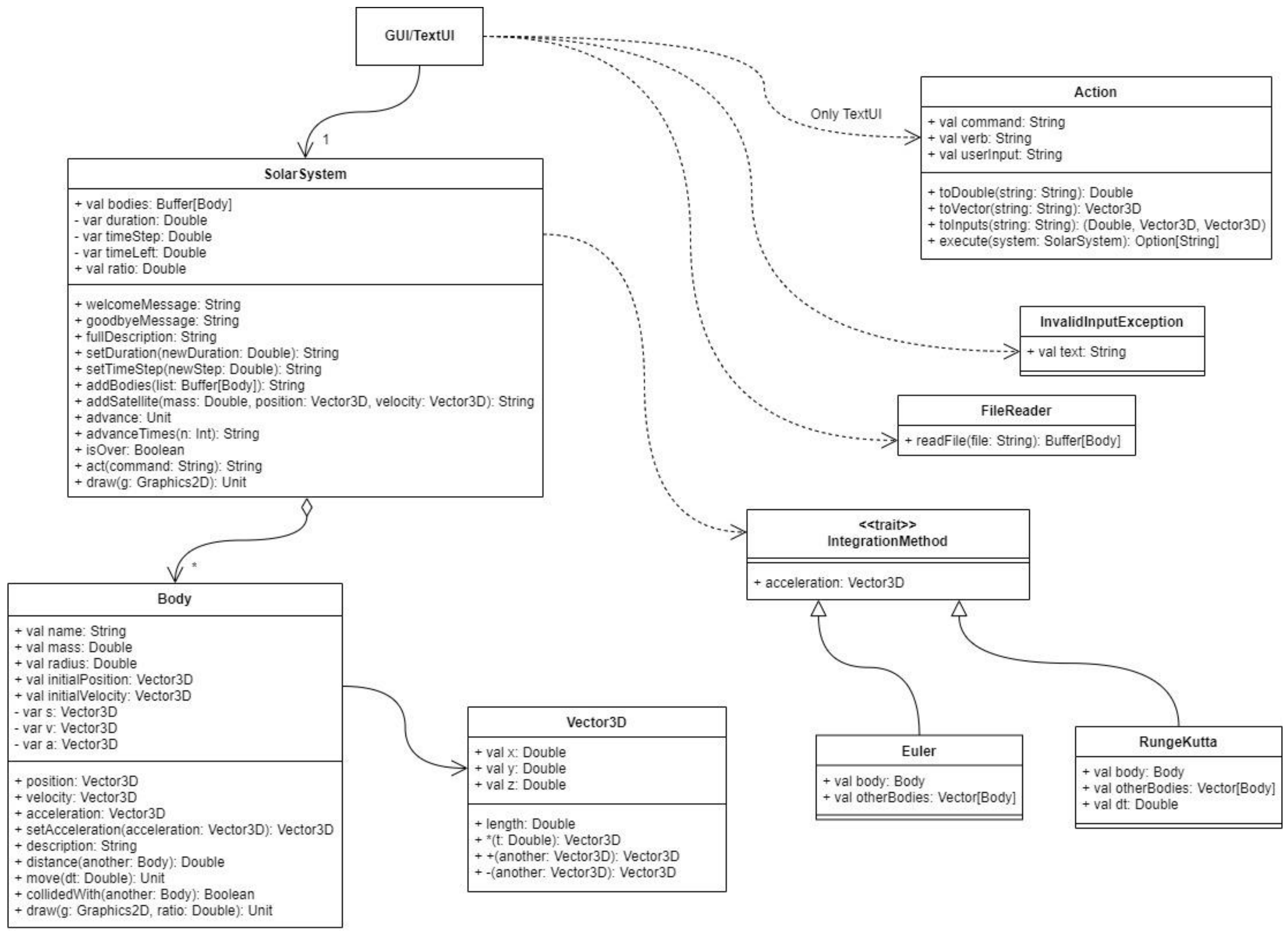

GUI/TextUI
Only TextUI
1
Action
+ val command: String
+ val verb: String
+ val userInput: String
+ toDouble(string: String): Double
+ toVector(string: String): Vector3D
+ toInputs(string: String): (Double, Vector3D, Vector3D)
+ execute(system: SolarSystem): Option[String]
SolarSystem
+ val bodies: Buffer[Body]
- var duration: Double
- var timeStep: Double
- var timeLeft: Double
+ val ratio: Double
+ welcomeMessage: String
+ goodbyeMessage: String
+ fullDescription: String
+ setDuration(newDuration: Double): String
+ setTimeStep(newStep: Double): String
+ addBodies(list: Buffer[Body]): String
+ addSatellite(mass: Double, position: Vector3D, velocity: Vector3D): String
+ advance: Unit
+ advanceTimes(n: Int): String
+ isOver: Boolean
+ act(command: String): String
+ draw(g: Graphics2D): Unit
InvalidInputException
+ val text: String
FileReader
+ readFile(file: String): Buffer[Body]
<<trait>>
IntegrationMethod
+ acceleration: Vector3D
*
Body
+ val name: String
+ val mass: Double
+ val radius: Double
+ val initialPosition: Vector3D
+ val initialVelocity: Vector3D
- var s: Vector3D
- var v: Vector3D
- var a: Vector3D
+ position: Vector3D
+ velocity: Vector3D
+ acceleration: Vector3D
+ setAcceleration(acceleration: Vector3D): Vector3D
+ description: String
+ distance(another: Body): Double
+ move(dt: Double): Unit
+ collidedWith(another: Body): Boolean
+ draw(g: Graphics2D, ratio: Double): Unit
Vector3D
+ val x: Double
+ val y: Double
+ val z: Double
+ length: Double
+ *(t: Double): Vector3D
+ +(another: Vector3D): Vector3D
+ -(another: Vector3D): Vector3D
Euler
+ val body: Body
+ val otherBodies: Vector[Body]
RungeKutta
+ val body: Body
+ val otherBodies: Vector[Body]
+ val dt: Double

Figure 1. UML diagram of the class structure.

Algorithms

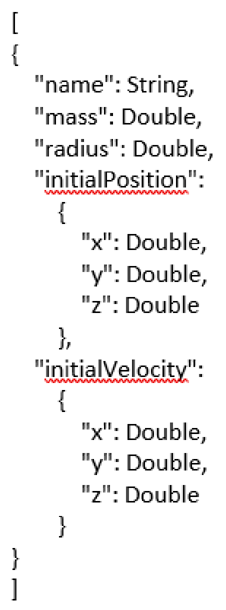
The fourth order Runge-Kutta method is used to update the positions and velocities of the bodies within the solar system. For each body in the solar system the idea is to calculate an approximation for the acceleration, which changes with position, by taking the weighted average of four different values for the acceleration due to each body and then adding the resulting vectors. The first value is calculated based on the mass of the body that causes the acceleration and the distance between the bodies ($a = \frac{GM}{r^2}$) and the three other values using the previous value for the acceleration. The weighted average is then taken, and the process is repeated so that accelerations due to all the bodies in the system are obtained. Finally the vectors are added to get the overall acceleration of the target body. This process is repeated for all objects as the targets. The velocity and position of all the bodies is then updated similarly as in the Euler method, which was also implemented as a starting point, but using the calculated approximation of the acceleration. The fourth order Runge-Kutta method is more accurate than most simpler methods, allowing for slightly longer time steps but still being computationally feasible.

Data structures

Buffers are mainly used to store the objects in a solar system because they are quite efficient and the collection has to be mutable to easily add new bodies to the solar system. Vector3D class modelling the three-dimensional coordinate system can also be seen as some kind of a data structure. The vector class stores three coordinates and has methods for computing the length of the vector and scaling, adding, and subtracting vectors.

Files and internet access

The sizes and initial locations and velocities of the objects in the solar system (except satellites added manually by the user) are read from a JSON file because it is human-readable and was covered quite well in the course materials.



```
[
{
   "name": String,
   "mass": Double,
   "radius": Double,
   "initialPosition":
      {
         "x": Double,
         "y": Double,
         "z": Double
      },
   "initialVelocity":
      {
         "x": Double,
         "y": Double,
         "z": Double
      }
}
]
```

Figure 2. File structure.

Testing

In the initial plan, the most important parts to be tested were determined to be advancing the solar system and calculating the new coordinates and velocities of the bodies, and the GUI in general, especially with incorrect input values. The original plan was also to implement some unit tests for testing the individual classes and methods, but in the end most of the simpler classes, such as Vector3D, FileReader, Body, and SolarSystem, and their functionalities were tested manually in REPL checking that the methods return correct values in the most common situations. Some problems were found and fixed and now all the classes seem to work properly. The GUI and

TextUI have been tested with different input values and files trying all the different functionalities and they both seem to work at least in the most common cases. Error handling has been tested with missing and wrong input types and formats as well as invalid file names and formats. The simulation itself has been tested with a simple solar system with a satellite at a Lagrangian point, with several planets, as well as with a system where a collision happens before the simulated time frame ends.

Known bugs and missing features

While testing, some bugs were found mainly in the GUI. Overall, the program is perhaps little too dependent on reasonable user inputs because there are no restrictions on the actual input values although the program notifies the user whenever the inputs are of invalid format. There are for example some problems with the view if the input values for the satellite position/velocity are somehow unreasonable, especially if they are very small. The simulation also becomes very slow if the time step length is set to be unreasonably small because no threading was implemented. On the other hand, the user also has to make sure the time step length is not too large as it decreases the accuracy of the calculations and might cause a collision to go undetected if the acceleration is very high.

Some problems also arise because of the way the distances are scaled down while drawing the solar system. For example, a satellite might be displayed on top of a planet instead of next to it if it is orbiting very close to the planet because of the way the radii are scaled. Also, because collisions are checked using actual units it might seem like two bodies collided on the GUI even though they actually did not. This could perhaps be somehow solved by displaying the actual coordinates on the window as well. Currently it is also possible to press the “Start simulation” button again while the simulation is running, which reads the given file again and adds the bodies to the solar system, causing some problems in the view if the bodies are added on top of each other. It is also possible to add a satellite while the simulation is in progress, which might cause some unexpected problems. These problems could possibly be solved by disabling the buttons once the simulation is started.

3 best sides/strengths and 3 weaknesses

I think overall the problem domain was modelled quite well. The structure of the classes SolarSystem, Body, RungeKutta, and Vector3D, and their relationships were planned and implemented so that they can easily be generalized and extended. The GUI was maybe the hardest part of the program to implement so I feel quite proud of the end result, even though there is still room for improvement. I think all the exceptions are also handled quite well by the user interfaces so that the user is notified whenever the input is of wrong format.

One weakness of the program would probably be scaling down the large values so that the bodies can be drawn in the GUI. Because it is basically impossible to display the distances, radii, and velocity and acceleration vectors to scale relative to each other, I had to scale the values down by different factors and create for example bounds for the radii of the bodies. There are most likely better ways to do this with less arbitrary constants. The input formats are also not perhaps the most intuitive for the user although the program instructs and notifies the user if they are invalid. One way to fix this might be for example to add an option to add a satellite by only giving its orbit radius and speed and general direction and let the program figure out how to represent these as vectors. There is also some redundancy in the code because both a text-based user interface and a graphical user interface were implemented.

Deviations from the plan, realized process and schedule

The initial plan was to start with the classes Vector3D and Body, then implement the SolarSystem and FileReader, and finally the GUI. The realized process followed this plan quite well although some additional classes were implemented in the process because the structure of the program was modified slightly from the original plan by implementing the calculations in a separate class, adding a class for exceptions, and another one for handling commands for the TextUI. The first two weeks were used to implement Vector3D and Body, as planned, as well as Euler and RungeKutta extending the trait IntegrationMethod, which took bit less time than originally estimated. The next two weeks were used implementing the class SolarSystem and object FileReader and testing some of the simpler classes, which also took bit less time than estimated. The last four two weeks were used building the GUI and TextUI and making some changes to the other classes as well, which took about the same time as originally estimated. The order followed the plan quite accurately but overall the project was always perhaps bit behind the original plan and thus a lot of work was left for the last three or two weeks. I think the time estimate was actually quite accurate, but it was still little surprising how long it took to implement all the features for the GUI. I also had to make quite a lot of changes to the original model of the program along the way. I learned that it is good to start to program quite early so that if I face some problems or get stuck with something there is time to put the project aside for some time and then continue working because that seems to be the best way to solve the problems.

Final evaluation

Overall, I am quite satisfied with the outcome of the project especially because it was my first bigger programming project. I am very proud of the way the GUI turned out although it could still be improved by making the input formats more intuitive and the program less dependent on reasonable user inputs. The way the actual coordinates and distances are converted to coordinates within the GUI could also likely be improved and simplified. I also feel like the code could be cleaner and simpler in some places. I think the problem domain was modelled quite successfully and the class structure works well. There are many possibilities for changing and extending the code by for example defining the type of the bodies, adding new algorithms for calculating the acceleration, and methods for reading other file formats. However, it is highly likely that there are more efficient ways of modelling the problem and implementing the functionalities and relationships between the classes that I simply do not know about yet.

I think if I could start over, I would start working earlier and pay more attention to time management. I would probably try to complete everything two weeks before the deadline so there would be more time for fixing unexpected problems that tend to emerge towards the end. I think I might also use more time panning the program structure and thinking about the class relations because it makes implementing them much easier and faster.

References

Fiedler, G. (2004). Integration Basics. Gaffer On Games. Retrieved from https://gafferongames.com/post/integration_basics/.

Voesenek, C.J. (2008). Implementing a Fourth Order Runge-Kutta Method for Orbit Simulation. Retrieved from http://spiff.rit.edu/richmond/nbody/OrbitRungeKutta4.pdf.

Docs – Scala 2: Collections. Retrieved from https://docs.scala-lang.org/overviews/collections-2.13/introduction.html.

Scala Swing Library Docs. Retrieved from https://www.scala-lang.org/api/2.12.0/scala-swing/scala/swing/index.html.

Appendix

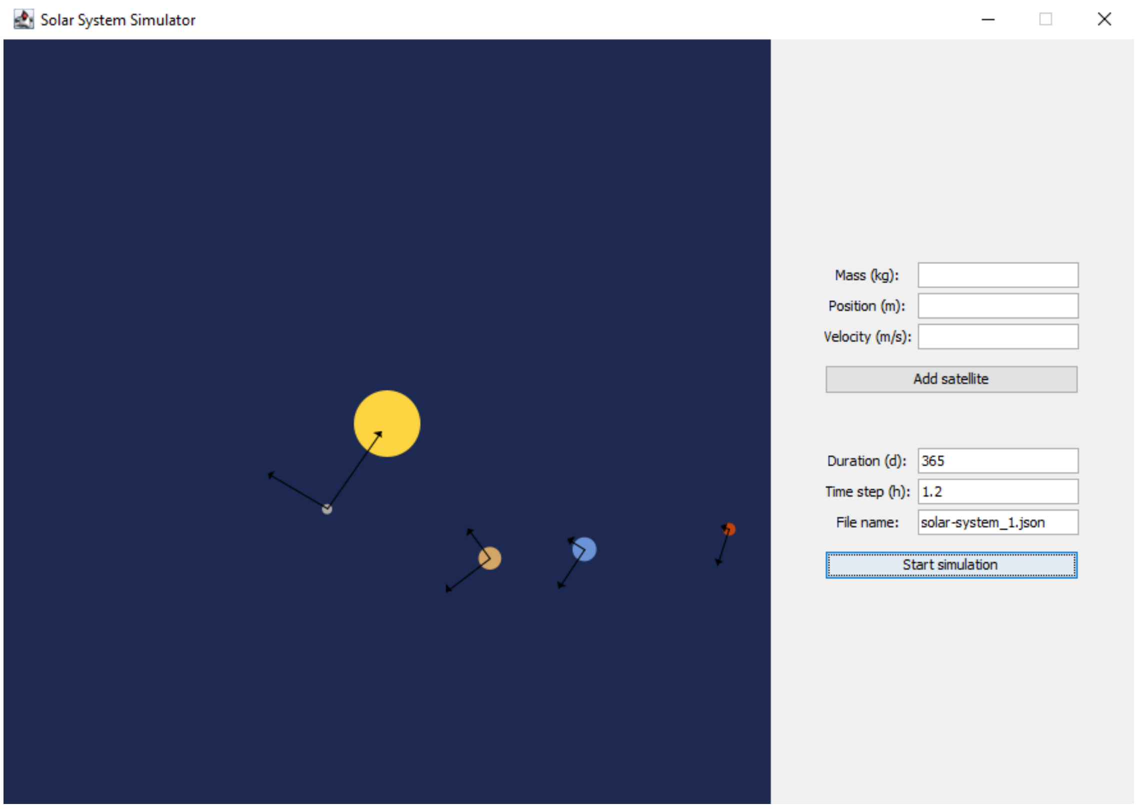
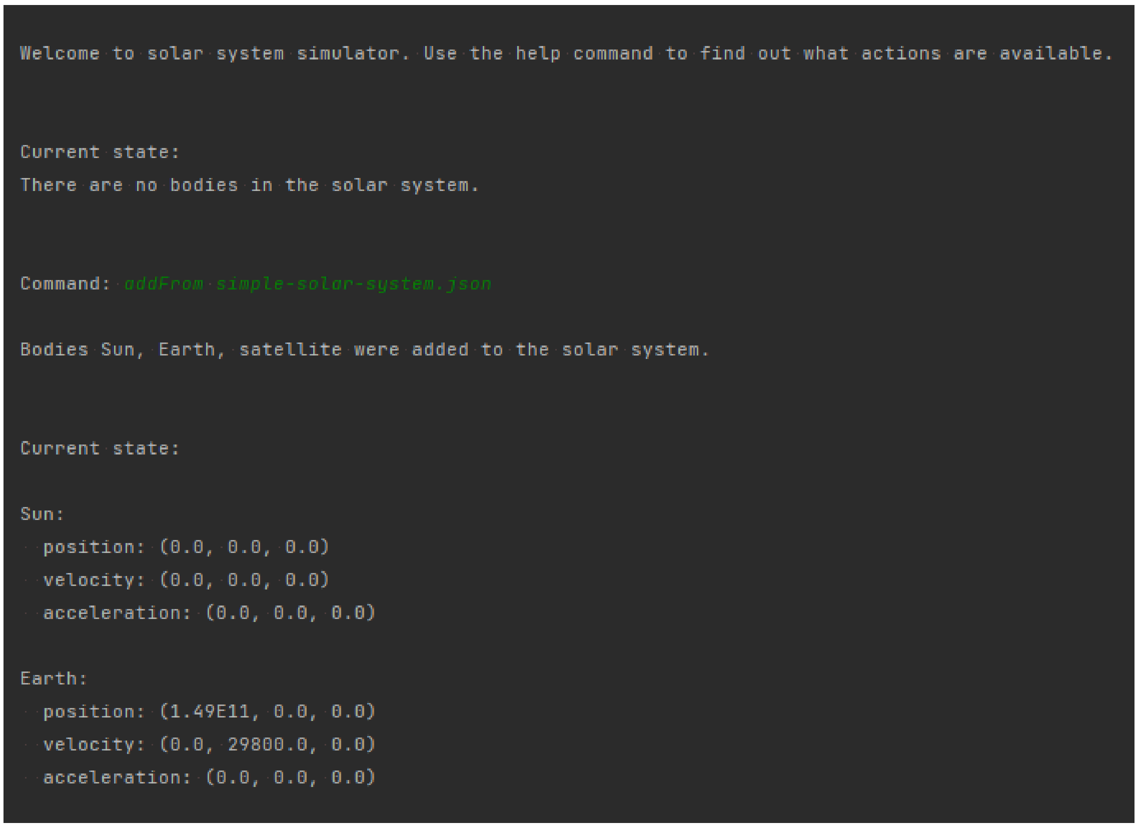


Figure 3. The graphical user interface.



```
Welcome to solar system simulator. Use the help command to find out what actions are available.


Current state:
There are no bodies in the solar system.


Command: addFrom simple-solar-system.json


Bodies Sun, Earth, satellite were added to the solar system.


Current state:

Sun:
  position: (0.0, 0.0, 0.0)
  velocity: (0.0, 0.0, 0.0)
  acceleration: (0.0, 0.0, 0.0)

Earth:
  position: (1.49E11, 0.0, 0.0)
  velocity: (0.0, 29800.0, 0.0)
  acceleration: (0.0, 0.0, 0.0)
```

Figure 4. The text-based user interface.