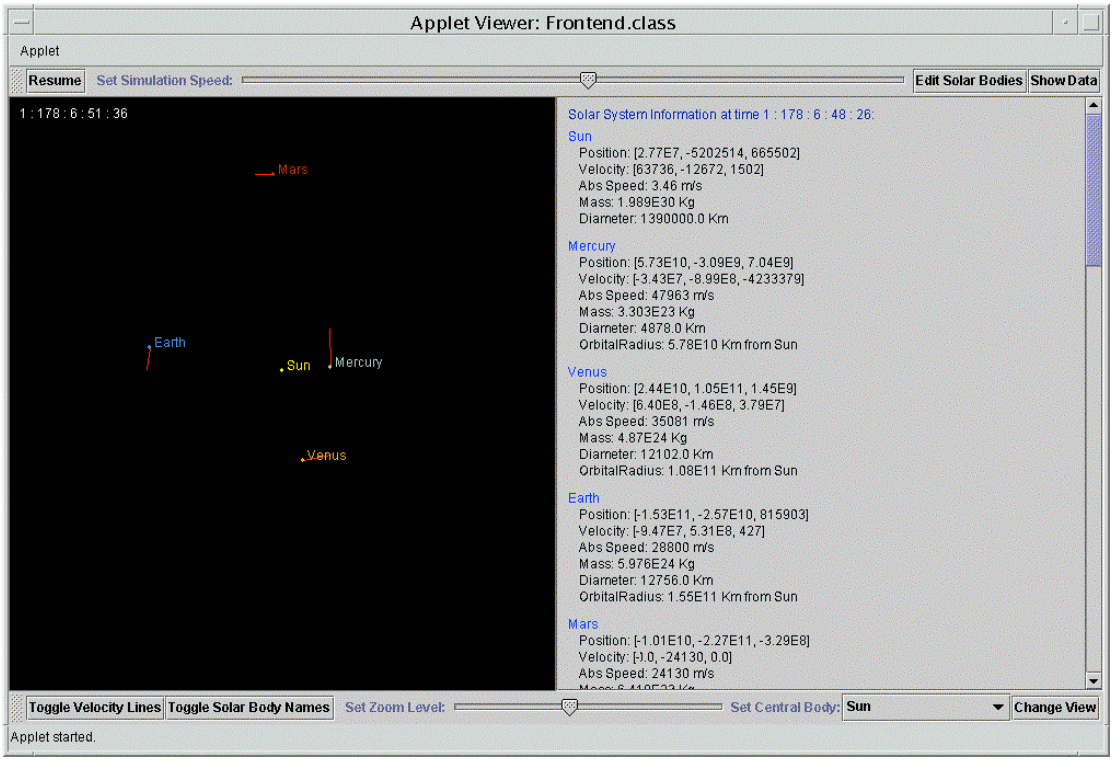
Summary

This project aims to produce a three-dimensional model of our Solar System, suitable for demonstrating the effects of gravity in an N-Body simulation. The movement of the Planets and Satellites is correct to a high degree of precision and there exist windows to view the state of the system both graphically, and in text-based form.

Points of user interaction include being able to manipulate the source of the view of the system; setting the display to follow a specific Planet, zoom in or out, or to change the angle from which the view is seen. Tools exist to allow any Star, Planet or Satellite to be changed in almost any way; changes to speed, direction of travel, absolute position, mass, diameter and even colour are all possible. It is even possible to create entire new Planets from scratch and see how they react to and affect the rest of the system.

In designing the system, I look into existing systems that achieve a similar task, and use their strengths and weaknesses to produce a full Requirements Analysis for my proposed system. This analysis is refined into a full design using methods from the Booch [1] Process and implemented as a Java Applet. By having members of my Target Audience fill out questionnaires, an appraisal of the success of the Product is achieved.

Screenshot of The Solar System Simulation



Contents

1. Statement of Originality

This report is submitted as part requirement for the degree of Computer Science and Artificial Intelligence at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed:

2. Acknowledgements

Many thanks to my project supervisor Julian Rathke, for his excellent guidance throughout this project.

Thanks to all people who participated in my questionnaire section to help assess the usability of my Program.

3. Introduction

There exist countless programs that model the gravitational physics of a set of Stars and Planets, some produce good realistic simulations while many others have much more marginal qualities. A common restriction of most of these is a distinct lack of user interaction; it can be nice to watch Planets orbiting each other but quickly becomes tiresome when there is no way to manipulate what is being seen.

This project aims to produce a three-dimensional model of our Solar System as a Java Applet, where a large degree of user interaction is encouraged. The movement of the Planets and Satellites in the simulation is correct to a high degree of precision and facilities must exist to allow the user to create new Solar Bodies, as well as to manipulate the existing ones to a large extent, including alterations to their movement and physical attributes. Key information about all the bodies in the simulation will be viewable both graphically and textually through two display windows.

3.1 Structure of this Report

In the first section I perform a requirements analysis; identifying the problem to be solved and what my product is intended to do. Analysing existing products that attack the problem provides an outline of the overall system and the best and worst aspects of these products are used to finalise what my end product aims to achieve. This list of properties is then be used to decide what is absolutely necessary to the completion of the project and what extras can then be added. Specific domain related problems are highlighted, even at this early stage, enabling solutions to be developed through the course of the project.

Following on from the analysis is a design section. In the design the overall architecture of the system is decided on and UML diagrams depict what classes exist and how they interact. A Main Class Diagram is included to show the overall architecture, while Collaboration and Sequence Diagrams show how each class interacts to achieve specific tasks. State Transition Diagrams demonstrate how some of the methods and variables change during the course of certain operations and a Use Case Diagram identifies what tasks the end user will be able to perform.

A full description of the system architecture and implementation is included to describe the classes, methods and interactions of the system, highlighting what problems were encountered during the course of development and what changes to the design were necessary, both for the completion of the project and to improve the system from the design.

Comprehensive testing of the Backend is included as this was the core part of the product; in testing this all the other classes used by the Backend were tested. I do not feel standard testing is appropriate for the graphical sections such as the Frontend, Display, Data Window and Solar Editor as the success of these is rather more subjective. Instead, members of my Target Audience use these parts and fill out questionnaires including an appraisal of what worked well and giving them an opportunity to point out any possible improvements.

The final conclusion section analyses the strengths of the product as well as any shortcomings or possible improvements that could be made to the system. An appraisal of how much of the design was finally implemented is also included.

4. Requirements Analysis

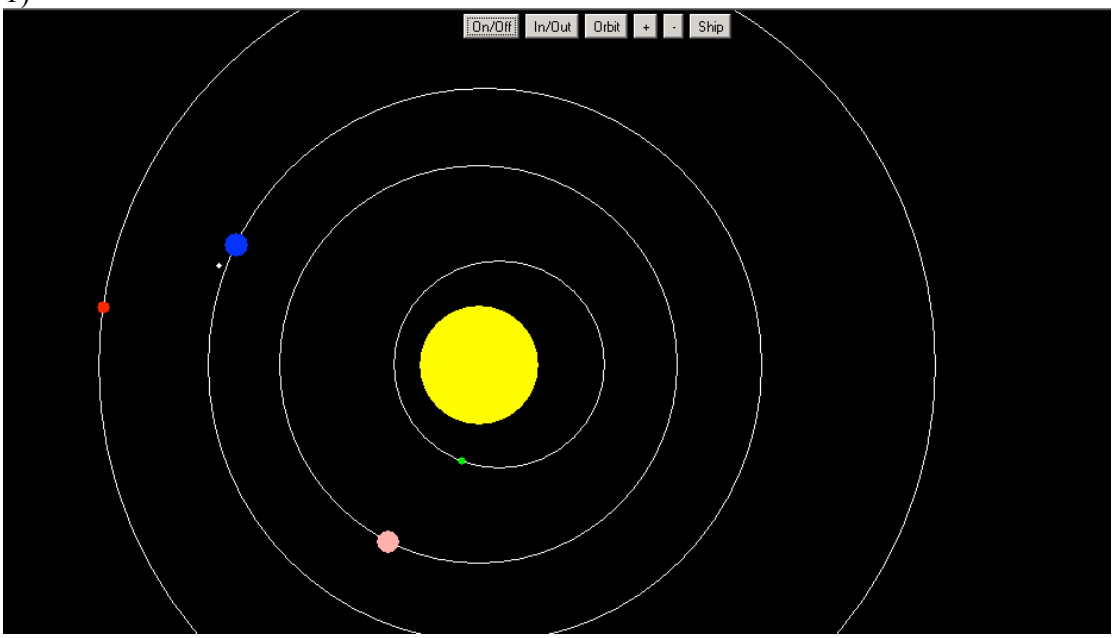
4.1 Product Definition

The final product is to be a working simulation of the major bodies of the Solar System, based on three-dimensional coordinates. The product is to be freely available over the Internet for ease of use in a classroom and as such will be built as a Java Applet. The target audience are pre-university students, such as those on an A-Level physics course.

4.2 Analysis of existing systems

In order to make the best possible system it is useful to see similar systems that other people have made in the past. What aspects of each program work well, were there any serious shortcomings, and what additions could be made to improve on each product? Fortunately there are many existing systems on the Internet that perform a similar task to that which I am trying to achieve. My goal is to take as many of the good aspects from these as possible and try and build on them, while striving to exclude as many of the shortcomings as possible. To this end I have performed an analysis of six systems that are currently on the Internet [2] - [7] as well as one commercial product [8]. The Internet-based systems should be more similar to what I am aiming for, but there are also likely to be many useful points to take from the commercial product.

1)

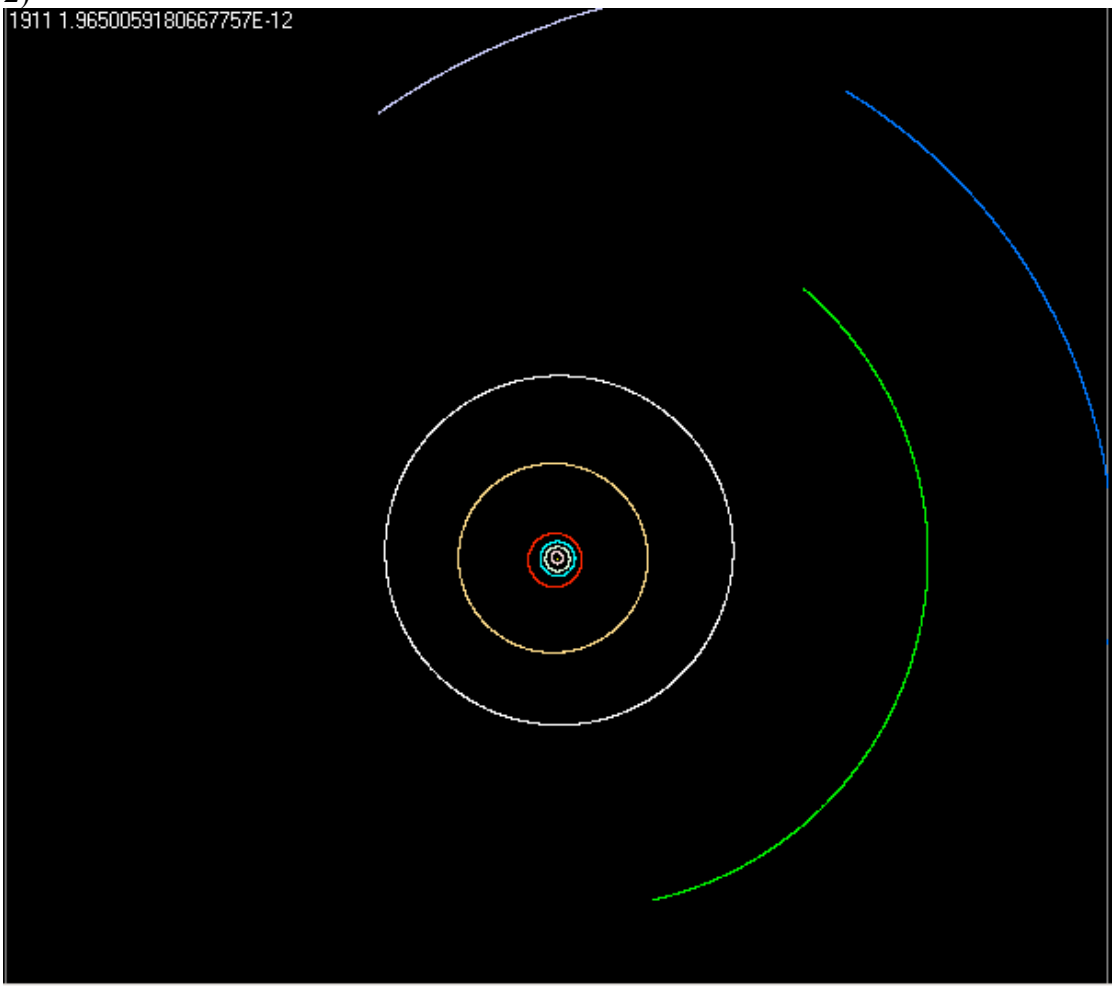


http://www.humnet.ucla.edu/humnet/french/faculty/gans/java/SolarApplet.html

This Applet contains all nine planets of the solar system plus our moon and the four largest satellites of Jupiter. There are '+' and '-' buttons that allow the user to resize the window so that the Applet can take a suitable proportion of their viewing area. Users can also select either the inner system (Mercury, Venus, Earth and Mars) or the outer system (Jupiter outwards) to view, preventing the Applet from being too cluttered while still maintaining a good sense of scale. Planetary orbits, velocities and sizes are approximately in scale with each other, which increases the sense of realism, but the planet scale has been made much larger than the orbit scale in an attempt to prevent the planets from being reduced to just single pixels and becoming hard to see. Additionally, there is a button to allow the user to toggle orbit lines on and off, however the orbit lines only exist for the Planets – Satellites do not have them.

One important problem is that there is no way of viewing all nine planets at once, which makes it more difficult for the user to appreciate the difference in scale between the inner system and the outer system. It would also be useful to have some actual data such as time scales, planetary masses and velocities so that there is more to learn from using the Applet. It is currently just Planets moving in circles at different speeds. Similarly there is no way for the user to manipulate the system, such as to change the speed or put in extra planets, which again limits what the user learns from the program and makes it less fun to use. There is also no way of viewing the system from another angle, so this program incorrectly gives the impression that all the planetary orbits are on exactly the same plane.

2)



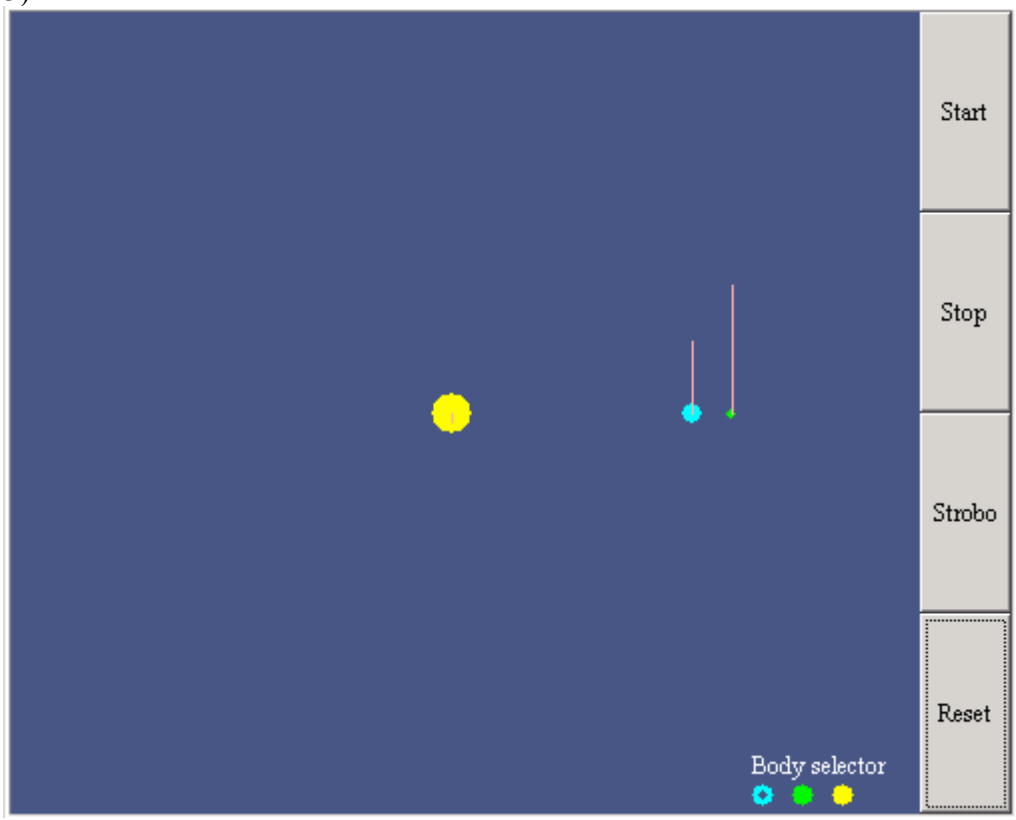
http://burtleburtle.net/bob/physics/solar.html

In this program, orbits are displayed close to their actual scale, which shows the great differences in scale between the inner and outer system. Unfortunately, due to this, the inner system is barely noticeable, limiting what the user can learn such as relative orbit times. Conversely, the actual planets are not shown, just their orbits, so the scale of the planets compared to each other cannot be appreciated.

The program does have a known 'real' start point (1 am, 1st January 1999), which reassures the user that what they are seeing is accurate. This is backed up by the fact there are real units of measurement mentioned on the website such as AU's, AU's/hr and lbs.

The user can rotate around the sun, giving a more three-dimensional feel to the system, although the controls for this are not friendly; the user has to click and drag in the middle of the applet, rather than use buttons. As in the previous program, there is no form of user interaction other than changing the view.

3)



http://www.particle.kth.se/~fmi/kurs/PhysicsSimulation/Lectures/13A/

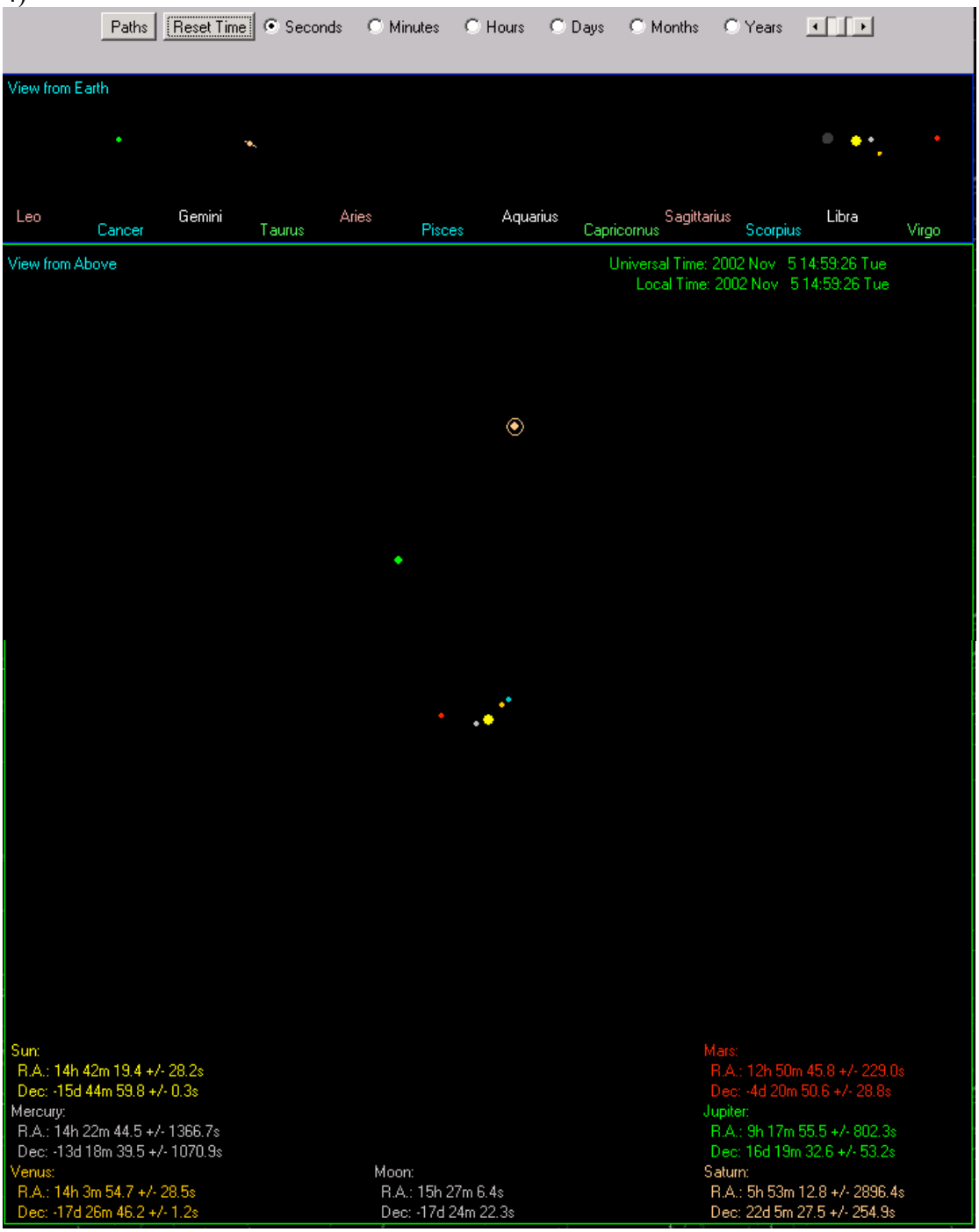
This was just a model of a Star, Planet and Moon. Although it is not of the Solar System, some of the interesting features still apply.

In this program velocities are marked in by pink lines (the longer the line, the higher the velocity), which allows the user to appreciate the huge changes in velocity that occur due to gravity.

The 'Strobo' button leaves the trail of each body so their previous movement can be seen, although the way this is implemented is not very neat and is just confusing. Each of the bodies can be manipulated by dragging them with the mouse, but this is tricky to do as the Planets move quickly and there is no way to slow the simulation down.

The Applet is of a nice scale and the menu panel is well separated from the viewer, although there is still no link to the real world – the bodies are not representations of actual Planets, no data such as size can be seen and there is no way of viewing from another angle. It is also not possible to make the planet crash into the sun, which could happen in reality.

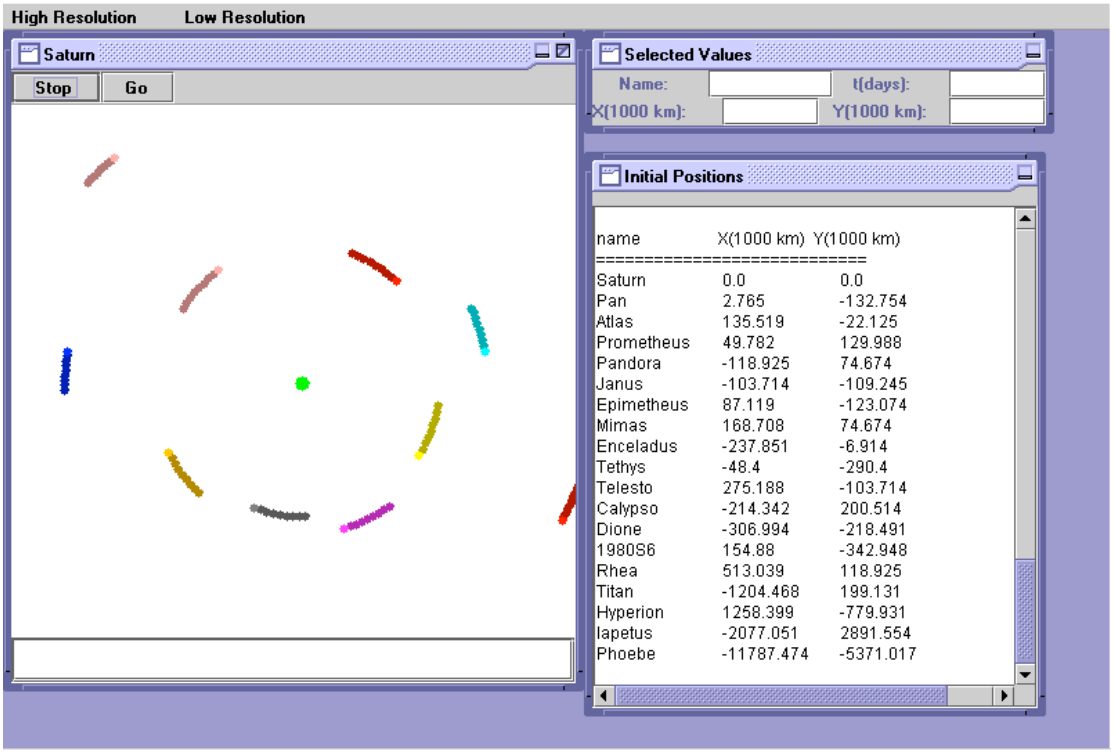
4)



http://jove.geol.niu.edu/faculty/stoddard/JAVA/luminaries.html

In this simulation, planetary sizes are (very approximately!) to scale, as are orbits and Saturn even has a ring! There is also a lot of data about orbit times and the system is built on real positions and times. The only form of user interaction is through changing the speed of the simulation but this is done in a way that is difficult to use and does not give enough precision. The system features two viewing angles, which gives a greater sense of three dimensions. Unfortunately the system stops at Saturn to attempt to keep a good scale, but this makes it feel incomplete.

5)

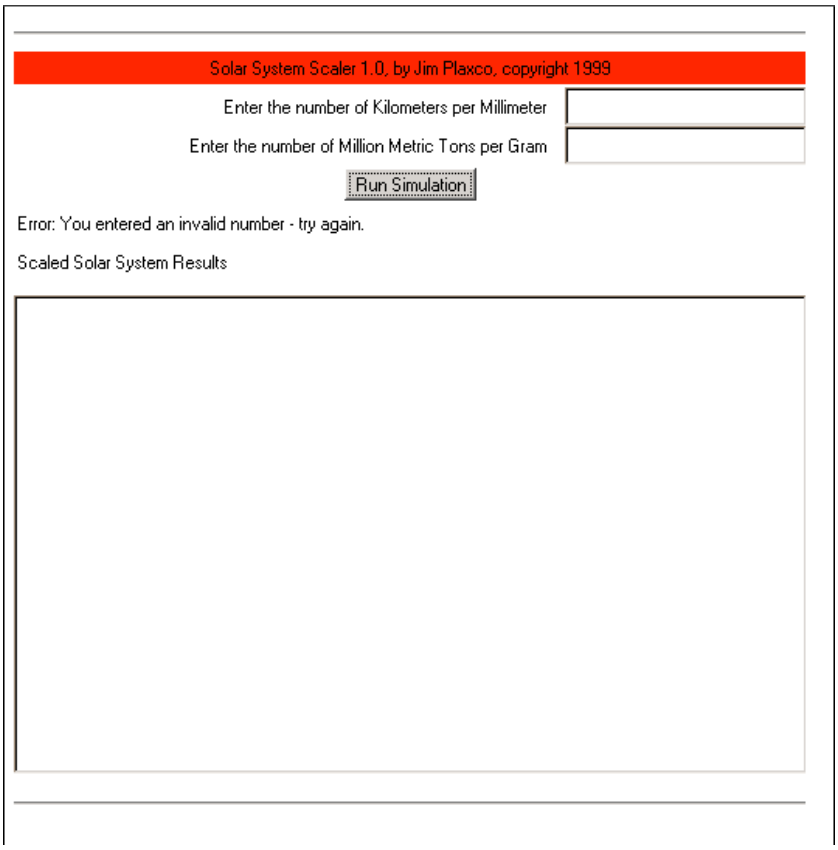


http://www.amherst.edu/~gsgreens/astro/Moon.html

The system only shows a single planet and its moons at any one time, but the general principle is similar to what I want from my system.

The program is very professional and organised looking, there is plenty of data shown and this is on a separate place in the applet making it look tidier. The orbits appear to be approximately to scale but all the moons are of equal size and not to scale with the orbits. There are also orbit lines, which are useful but cannot be turned off so the applet eventually looks like a green dot with lots stationary circles around it.
The applet also lacks a zoom function, and the viewing angle cannot be changed so it looks very 2-dimensional and there is no form of user interaction.

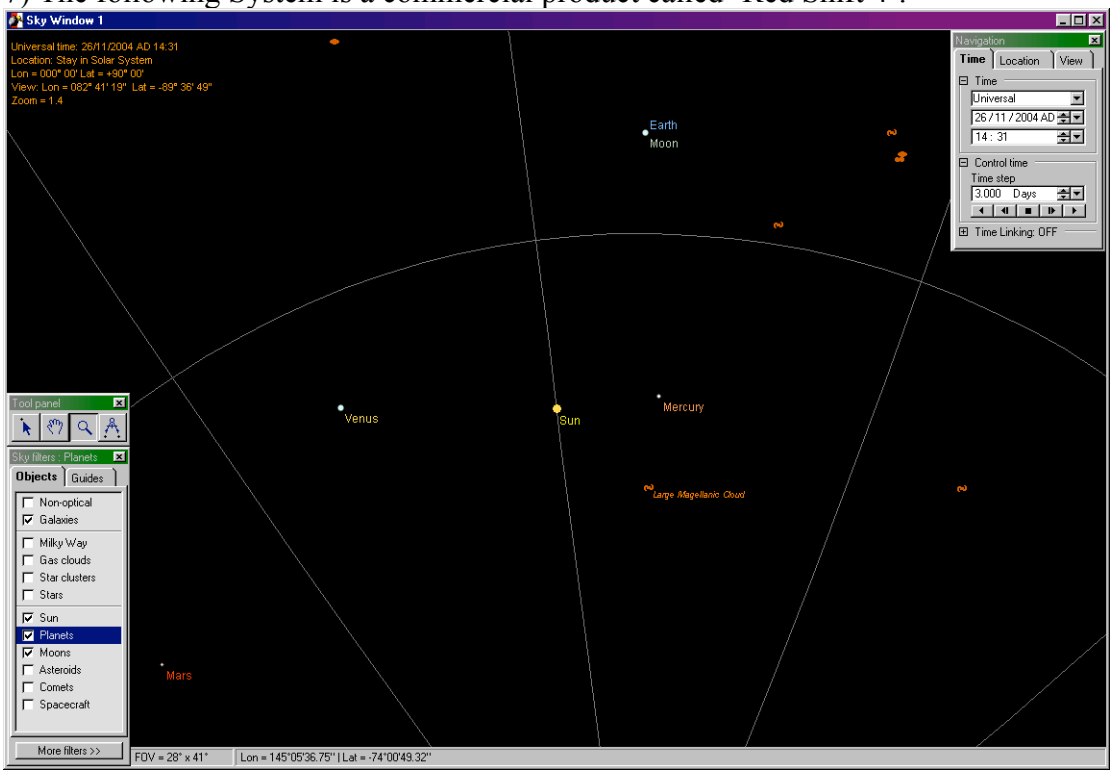
6)



http://www.astrodigital.org/astronomy/ss.html

This applet claims to use accurate scales, which I am unable to test because the applet is so confusing to use. This is a fundamental flaw, especially for my purpose, which is to be able to easily pick it up, use it, and then never see it again. One of the main things that made it hard to use was requiring text entry from the user without giving a clue of what sort of information was required. The error message was also completely useless, as it did not suggest how to fix the problem.

7) The following System is a commercial product called ‘Red Shift 4’.



This product includes many features that are not feasible for me to do on my time scale and is beyond the scope of a simple solar system simulation. Information on 70,000 Galaxies and 15,000 asteroids is great for the purpose of this product, just not necessary for a tool that will probably only be used briefly. As such it is also unsuitable as a product to just pick up and use as there could be a substantial learning period.

What it does include which is of a greater interest to me and my project is an accurate display of every major body in the solar system (including comets and asteroids) with tools to view these from any angle and at any zoom scale desirable.

As a commercial product with a larger budget it is far more professional looking than any of the applets I found on the Internet and contains far more information. It also requires a bit of time to work out all the controls and a much larger amount of time than I have to fully explore every feature and piece of data.

4.2.1 Summary of Existing Systems

An ideal system would be one that includes all the good points mentioned above with none of the bad points at all, as described below.

The ideal system should include all nine planets of our Solar System, The Sun and any other major bodies such as large moons. The scales of the objects should be representative of the scales in real life; for example Jupiter must be bigger than Earth, the Earth bigger than the moon, and the Sun bigger than all of them. The orbits should also be of a realistic scale, the inner planets far closer to each other than the outer ones. All the data regarding each solar body should be available in some sort of display, including time, planetary masses and velocities among other figures.

The user should also be able to manipulate the display in many ways; resizing the window, zooming in and out to see different detail levels and different planets, and changing the viewing angle.

Additional features include the following; orbit lines that can be toggled on and off to give a greater feel of where the planets move to over the course of time. Controls to let the user to manipulate the system by changing speed of time, change existing planets and create additional planets. The controls for all aspects of the system should be as clear and easy to use as possible.

4.3 Target Audience Requirements

There are five key requirements for this product to be appropriate for use by the intended target audience. It must be free and easy to get hold of, simple to use, informative, accurate and be able to hold the interest of the user.

To ensure that the product is easy to acquire, it is designed and built as a Java Applet. This means that it can be simply downloaded and ran straight off the Internet, so there is no need to install the program on lots of computers. As a Java Applet it can also be run through a Web Browser such as Internet Explorer or Netscape, which practically any computer in a school or college will come readily equipped with. In a classroom where each student has their own computer on a network, all the teacher has to do is tell the students the web address of the applet on the Internet. This part is imperative as a teacher may have in the region of thirty students and will not have time to go round to each student individually and sort out any problems they have with setting up the system.

It is very important for the Applet to be simple to use, as one teacher will not have time to run around thirty students showing them each and every aspect of the product. Additionally a key part of learning is discovering things yourself so this must be as easy to do as possible. When downloaded, the majority of the controls as well as the Display are arranged on a single frame, with the controls evenly split between two toolbars; the top one has general applet controls for manipulating the system and the bottom one has all the controls necessary for altering the viewing display. Whenever possible, controls such as slider bars and drop-down menus are used for user interaction, rather than text fields which would require the user to have too much knowledge of the workings of both the Solar System and the Applet itself making it

much clumsy to use. One of the major points of user-interaction is the Solar Editor Frame. Several pieces of information are required from the user at this point so the controls are friendly in the way described above and I also made sure I disabled the controls for any part that the user should not have access to at any point. A possible major difficulty for a user could be specifying a solar body from scratch, so example values are included for a solar body called Planet X, which the user could make use of as a template. Planet X is a planet believed to exist beyond the range of Pluto, so I included it as such.

An important aspect of the project is making the Applet informative while not overwhelming the user with too much information. The first step to reaching this goal is to include an appropriate number of Solar Bodies. Just having one star and one planet is not very useful as there are very few interactions involved in such a simplified universe; a student would not see how changing the Planet in any way affects any other solar bodies. Similarly, including all the Planets and all their satellites (which include over a dozen around each of Jupiter, Saturn and Uranus), would clutter the system up making it hard to distinguish important Solar Bodies from lumps of rock. To make a good compromise my simulation includes all Planets and any Satellites greater than 10^{22}Kg in mass. This eliminates the two tiny satellites of Mars as well as countless rocks around the largest Planets that have been identified as Satellites. Key information of orbital radius and speed as well as absolute positions and velocities of each Solar Body should be a sufficient amount of data for the user to view. Telling a student the individual forces of acceleration between every pair of solar bodies for example would just be confusing. The students could easily calculate information like this by using formulae for velocity and acceleration.

Accuracy of data is another key issue in this project as the entire goal is to teach students about the Solar System. Being taught incorrect information would be most unfortunate so I have endeavoured to ensure the data is correct. The data needed for each Planet such as mean orbital radius, mass and diameter has been checked against at least one other source. In most cases one of these sources is the commercial product RedShift4, and the other from one of various Internet sites. Additionally, my tests performed on the backend over an extended period of simulated time ensure that the data leads to realistic orbits and in doing so also ensures my formulae used in the Backend are accurate.

Ensuring that the simulation is interesting to use may not be as fundamental as some of the above points but it is still extremely important if the students are to get any real benefit from the simulation. If the program were dull, students would quickly lose interest and probably not bother looking at the program properly. The only way to ensure the students make good use of the program is to make sure it holds their interest. One thing that helps make the program interesting to use is by including as much user interaction as possible; be this by specifying new Planets, altering existing ones, viewing from different places or simply letting the user crash planets into the Sun, lots of user interaction should greatly aid the success of my product.

4.4 System Support

People running the simulation must have a Personal Computer with a recent web-browser such as Netscape 6. The program is a java Applet available over the Internet so java support in the web-browser must be turned on and access to the Internet available.

4.5 Core Functionality

The first and main task is to build a backend engine that drives the simulation based on 3-dimensional coordinates. This engine will make use of as much existing java functionality as possible, aim to be realistic and represent the planets as accurately as is feasible. There should be the Sun and at least two planets including the Earth.

On top of this, a Frontend will be is built to display the entire system as viewed from directly above the plane of the Earths orbit, with a time display and friendly controls to start and stop the simulation.

This will meet the minimal goal of showing how multiple Solar Bodies interact over a period of time.

The following key points define the core functionality:

1. Build an Engine that manipulates the positions, velocities and accelerations of Solar Bodies in three dimensions over a period of time.
2. Include the Sun and two Planets (minimum).
3. Build a display that renders the positions of each Solar Body over time.
4. Include a time display.
5. Include controls to start and stop the simulation.

4.6 Extended Functionality

If the existing functionality of Java does not give enough accuracy then increase the accuracy of the Backend by writing my own java functions to manipulate equations to a higher degree of precision. This is only useful if the functions are efficient enough to let the simulation run quickly. The Backend should also be extended to include other major bodies such as the rest of the planets and some of the more major moons.

The viewing system should be extended so that the user can zoom in and out to see different levels of detail, as well as have the ability to centre on a specific planet and follow it through its orbit. Within the display the planets should represent the difference in size of each planet. Orbit lines would be a useful feature to give more insight as to where Solar Bodies have been in the past.

There could also be a tool to let the user specify extra bodies to see how they would react to and affect the system. This tool could also let the user edit existing bodies in the system to see the effect of that, for example setting the mass of the Sun to 0Kg would effectively remove it from the system and the user could then see how everything else reacts.

A separate textual display of extra planetary information, such as mass, velocity, position and distance from sun should be included.

The following key points define the extended functionality:

1. Define a set of functions to manipulate equations to a greater degree of precision if the existing Java functionality is insufficient.
2. Include all nine planets and natural satellites above 10^{22}Kg in mass.
3. Include a zoom function.
4. Include a function to allow the user to centre the display on any solar body.
5. Drawn Solar Body sizes made representative of their actual sizes.
6. Allow Orbit lines to be toggled on and off.
7. Include a tool that can be used to specify new Solar Bodies.
8. Include a tool that allows the user to alter existing Solar Bodies.

4.7 Primary Function Points

Function points are a key part of the Booch Process [1] used during a requirements specification. Each point captures an element of functionality of the final product so the list is useful when implementing a design as it forms a good checklist to ensure the product does all it is intended to. Each function point is categorised into one of the four sets where each set contains related functionality. The first set is the core functionality, which is imperative to the working of the system and forms the basis of a prototype. The other sets form three levels of extended functionality; the first level being the most important and the third least important.

C = Core Functionality
E1 = First Level of Extended Functionality
E2 = Second Level of Extended Functionality
E3 = Third Level of Extended Functionality

- Calculate next time step. The Backend engine calculates the next positions of all the solar bodies based on their current position and the size of the time step. **(C)**

- Display the state of the system for a given point in time. The Display takes a snapshot of the Backend and renders the Solar Bodies based on this. **(C)**

- Pause simulation. The Backend is allowed to finish its calculations for the next time step and the Display refreshes to show this new state but the backend will not start calculations for next time step. **(C)**

- Resume simulation. The backend restarts from its last state and system behaviour resumes. **(C)**

- Change zoom level. On an increase, the Display shows a smaller area of the Solar System but in greater detail. On a decrease, the Display shows a larger area of the Solar System with less detail. The user, by means of the Frontend, controls the zoom level. **(E1)**

- Change time step. The time step used by the Backend is changed according to user specification through the Frontend. **(E1)**

- Display current system data. A separate window appears to show data on any object in the system, based on the current snapshot of the Backend. **(E1)**

- Change Viewing angle. The Display shows the system from a new position, which is set by the user through the Frontend. **(E2)**

- Change central viewing object. The Display centres on a solar body as specified by the user, by means of the Frontend. The zoom level and angle is not changed by this process **(E2)**

- Toggle orbit lines. The Display simply starts to draw lines following the previous orbit positions or remove them on request. **(E2)**

- Add new Solar Body. The Backend is modified to include a new Planet or Satellite with a specified mass, velocity and position as well as other attributes, which is then reflected in the Display. The user, through the Solar Editor activates the function. **(E3)**

- Alter Solar Body. The user alters the specified body using the Solar Editor. The Backend then reflects these changes by performing its calculations based on the new data and the Display will ultimately show a different effect. **(E3)**

4.8 Constraints

The main constraint here is time. With only one person and an allocated time of approximately 10 hours a week and 10 weeks in the spring term, there is only 100 man hours to produce the main part of the system, including coding and testing.

There is also a limit on the amount of information that can be put into this system, so I shall only concentrate on things within the Solar System and limit it to the major bodies above a certain size at that.

A third major constraint is ease of use, the product must be sufficiently simple to use that anyone with a minimal amount of computer experience can use it unsupervised.

5. Design

This section uses the information gathered in the Requirements Analysis to produce a concrete design of my intended system. I use UML Diagrams as used by Grady Booch [1] to describe the architecture of the system as they are a clean way of specifying precisely what each class does as well as how the classes interact. UML diagrams can also help to visualise what the system would actually be like.

Section 3.1 is the Main Class Diagram. Each class is defined together with the key attributes it will use and the operations it must perform. The relationships between each class are also described here; which classes use other classes to achieve their goals and where inheritance occurs to produce sub-classing. A textual description of each class is also included to precisely specify its role in the overall system.

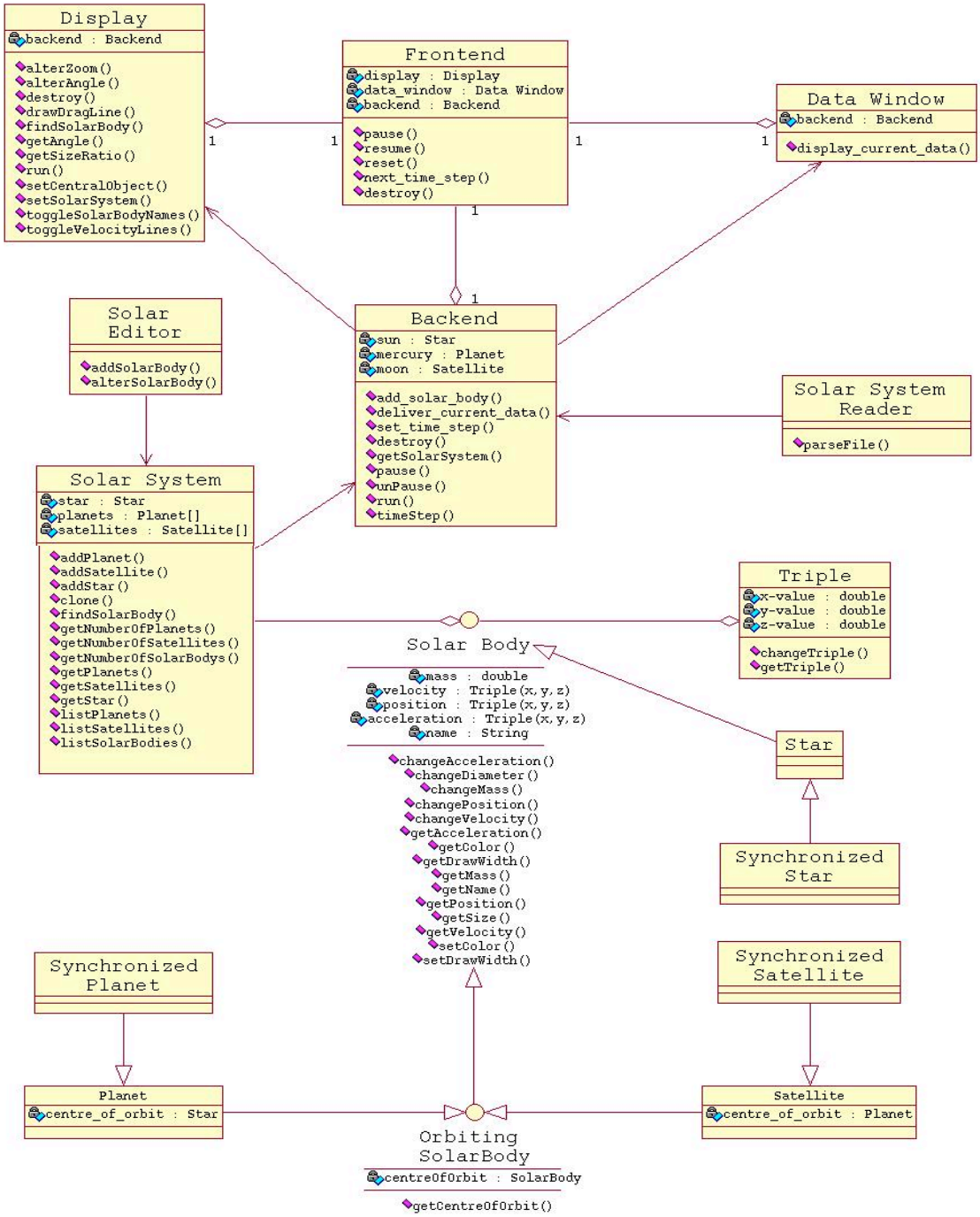
Section 3.2 contains a set of Collaboration Diagrams; each one describing how two classes interact to achieve an overall goal. In each case the first class calls the specified method in the second class and that method is what does the work to achieve the goal.

Section 3.3 shows a set of Sequence Diagrams; similarly to the Collaboration Diagrams, these show how classes interact to achieve a common goal. Methods are called across classes and objects can be created or passed back as a consequence.

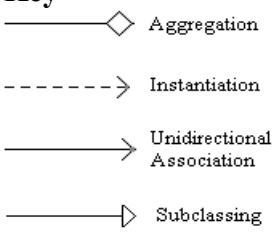
Section 3.4 is a set of State Transition Diagrams. These show how the state of the system or a part of the system changes when certain methods are called or variables changes.

Section 3.5 shows the Use Case Diagram for the system. Each Use Case is an extension of a single function point from the Requirements Analysis and captures a single task that the system must perform.

5.1 Main Class Diagram



Key



Backend
The Backend is the engine that drives the simulation. It stores and manipulates the positions, velocities and accelerations for every Solar Body in the simulation over time.

Data Window
This takes data for each Solar Body from the Backend at a specified time and renders it to screen in textual form.

Display
This represents the current state of the Backend graphically. Changes to the Backend are reflected in the Display for the user to see.

Frontend
The Frontend is the main source of interaction for the user as it is the Graphical User Interface. The Display and Data Window are shown in the Frontend together with controls to manipulate the running of the system as well as how it is displayed.

Orbiting Solar Body (*Interface*)
This Interface extends the Solar Body Interface and is implemented by every Planet and Satellite. It allows a 'Centre of Orbit' to be specified from which an Orbital Radius can be calculated.

Planet
This implements the Orbiting Solar Body Interface. It is used to bundle up all the information regarding a single Planet of the Solar System and contains methods for manipulating and accessing this data. An instance of the Planet class represents a single Planet to be manipulated by the Backend and viewed by the Display and Data Window.

Solar Editor
This is a Separate user interface used to specify alterations to the fundamental structure of any Solar Body in the system as well as allow the user to specify any extra Solar Bodies they may desire.

Solar Body (*Interface*)
This Interface specifies the basic methods any Solar Body in the simulation must implement. The polymorphism involved is key to relating all Stars, Planets and Satellites.

Solar System
This is used to bundle together all the Solar Bodies in the simulation as a single Object. Methods exist to allow new Solar Bodies to be added as well as to access existing Solar Bodies.

Solar System Reader
This is used when initialising the simulation to parse the permanent text file and produce all the initial Solar Bodies with the correct data.

Satellite
This is a sub-class of the Orbiting Solar Body interface. An instance of the Satellite class represents a single Satellite to be manipulated by the Backend and viewed by the Display and Data Window.

Star
This is a sub-class of the Solar Body interface. An instance of the Star class represents a single Star to be manipulated by the Backend and viewed by the Display and Data Window.

Synchronized Planet
This is a sub-class of the Planet class where the methods to manipulate its data are synchronized to prevent changes from occurring during a clone operation.

Synchronized Satellite
This is a sub-class of the Satellite class where the methods to manipulate its data are synchronized to prevent changes from occurring when cloning.

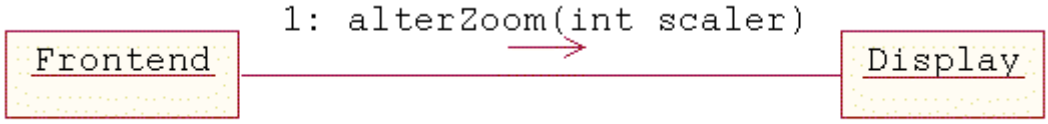
Synchronized Star
This is a sub-class of the Star class where the methods to manipulate its data are synchronized to prevent changes from occurring during cloning.

Triple
This is used to bundle up three java double variables as a single Object. Each instance represents the three-dimensional coordinates of Position, Velocity or Acceleration of any Solar Body in the simulation.

5.2 Collaboration Diagrams

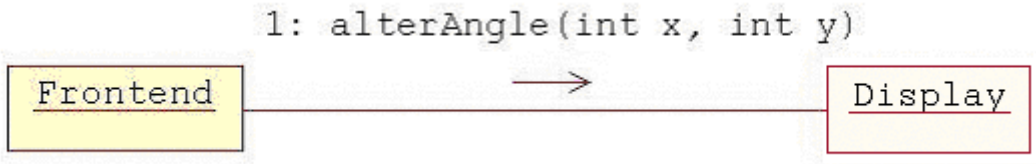
5.2.1 Control of how to alter the zoom of the Display

The Frontend calls the alterZoom method of the Display, passing in the new zoom level as the variable 'scaler'. The AlterZoom method then re-sizes everything to be drawn to reflect the new zoom level.



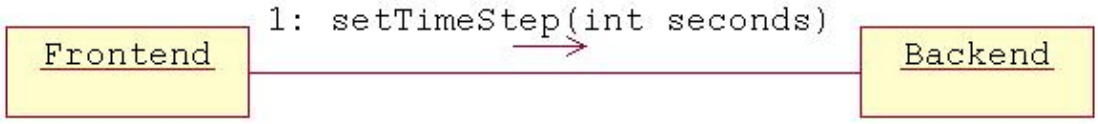
5.2.2 Control of how to alter the angle of the Display

The Frontend calls the alterAngle method of the Display, passing in values to determine which dimensions the Display should draw horizontally and vertically.



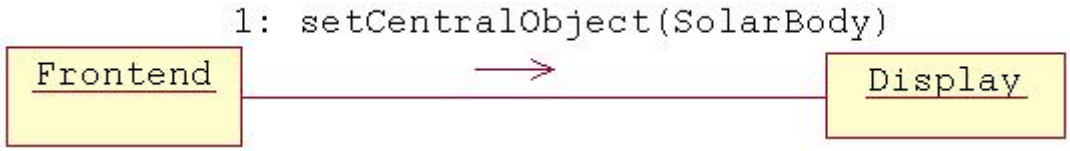
5.2.3 Setting a new rate of time

The Frontend calls the setTimeStep method of the Backend, passing in the number of seconds that is to pass between each time-step. The setTimeStep method also re-scales the current values for velocity and acceleration of each Solar Body to reflect this new time step size.



5.2.4 Centre on a Solar Body

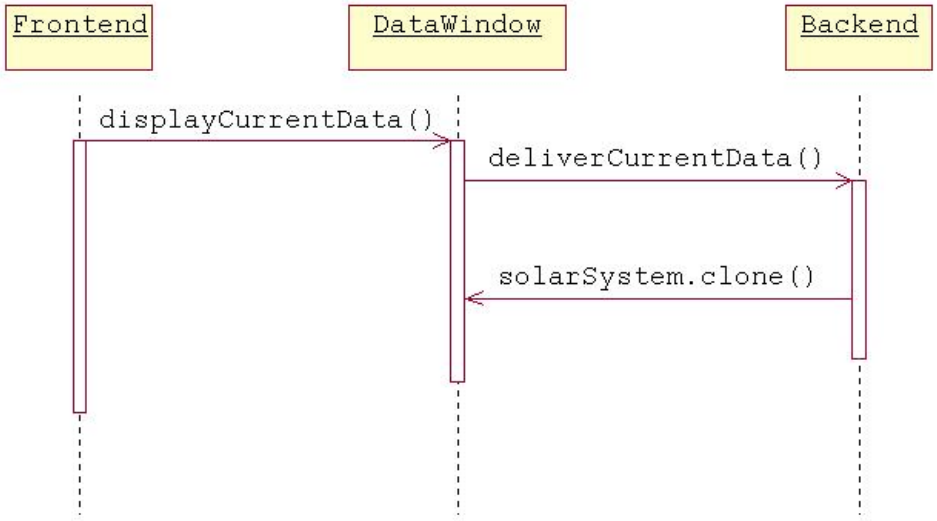
The Frontend calls the setCentralObject method of the Display, passing in the Solar Body that is to be centred on. Future iterations of the Display calculate draw positions relative to the specified Solar Body.



5.3 Sequence Diagrams

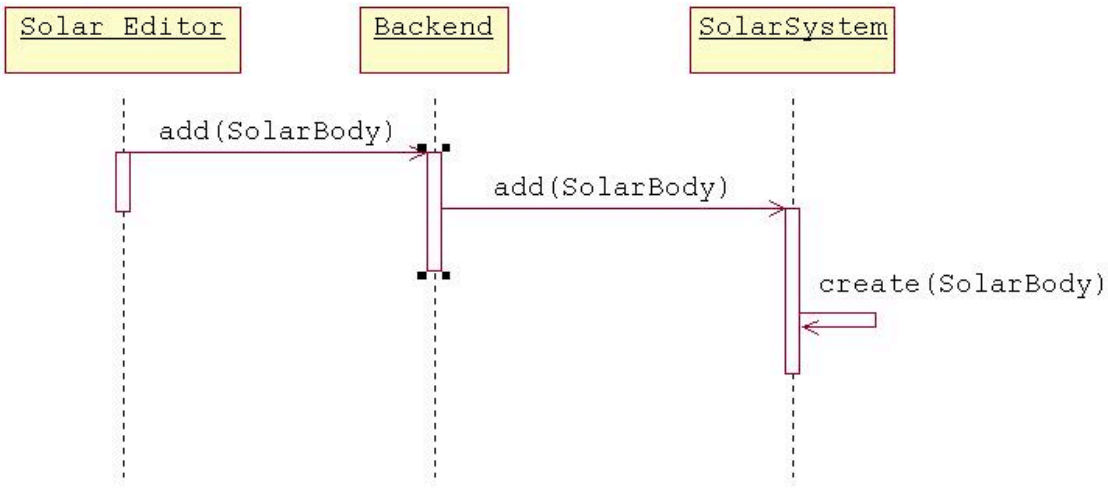
5.3.1 Display current system data as text

The Frontend notifies the Data Window that it is to redisplay the current data of the system by calling its displayCurrentData method. The Data Window calls the deliverCurrentData method of the Backend, which returns a clone of the Solar System in its current state. This clone is then safe from changes to the Backend and can be used by the Data Window to display the current textual information for each Solar Body in the System.



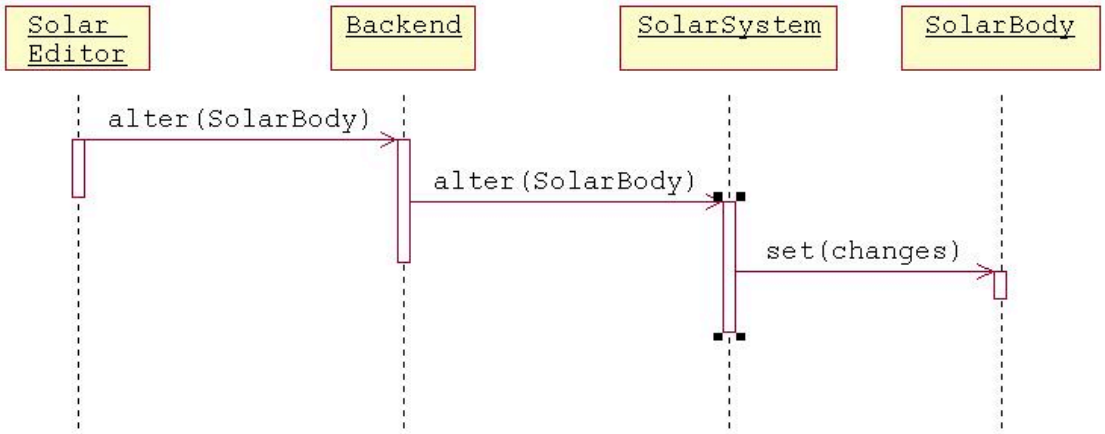
5.3.2 Create a new Solar Body

The Solar Editor calls the add method of the Backend, passing in the new Solar Body and the Backend then calls the add method of the Solar System class, again passing in the new Solar Body. The Solar System then puts the new Solar Body at the appropriate position in its variables. This new SolarBody will then be available to the Backend and as such used by any other classes that make use of the Backends' SolarSystem object.



5.3.3 Alter an existing Solar Body

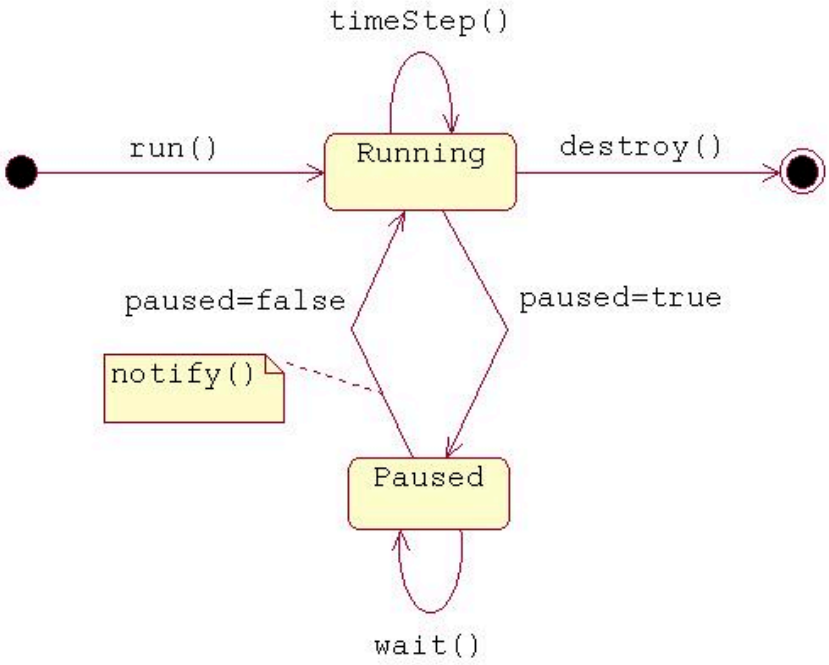
The Solar Editor calls the alter method of the Backend, passing in the changed Solar Body. The Backend calls the alter method of the Solar System object, again passing in the changed Solar Body. The Solar System then identifies which Solar Body is to be changed by matching the name. The changes to be made are then set in the Solar Body.



5.4 State Transition Diagrams

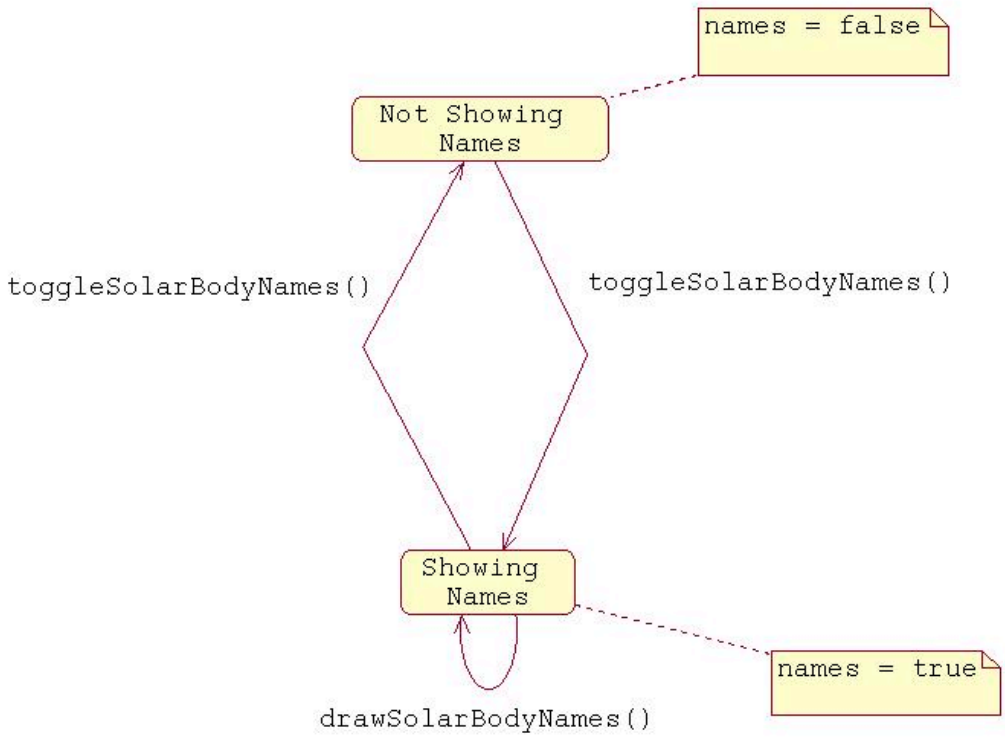
5.4.1 Pause the simulation

The simulation starts when the run method of the Backend is called. While it is in the running state, the timeStep method is repeatedly called to update the situation of the Backend. When the pause method is called the variable 'paused' is set to true and as a result the 'wait' method is invoked suspending the simulation. When the unPause method is called, the 'paused' variable is set to false and the 'notify' method is called to allow the Thread to continue. When destroy is called the simulation ends and can then only be restarted from the beginning.



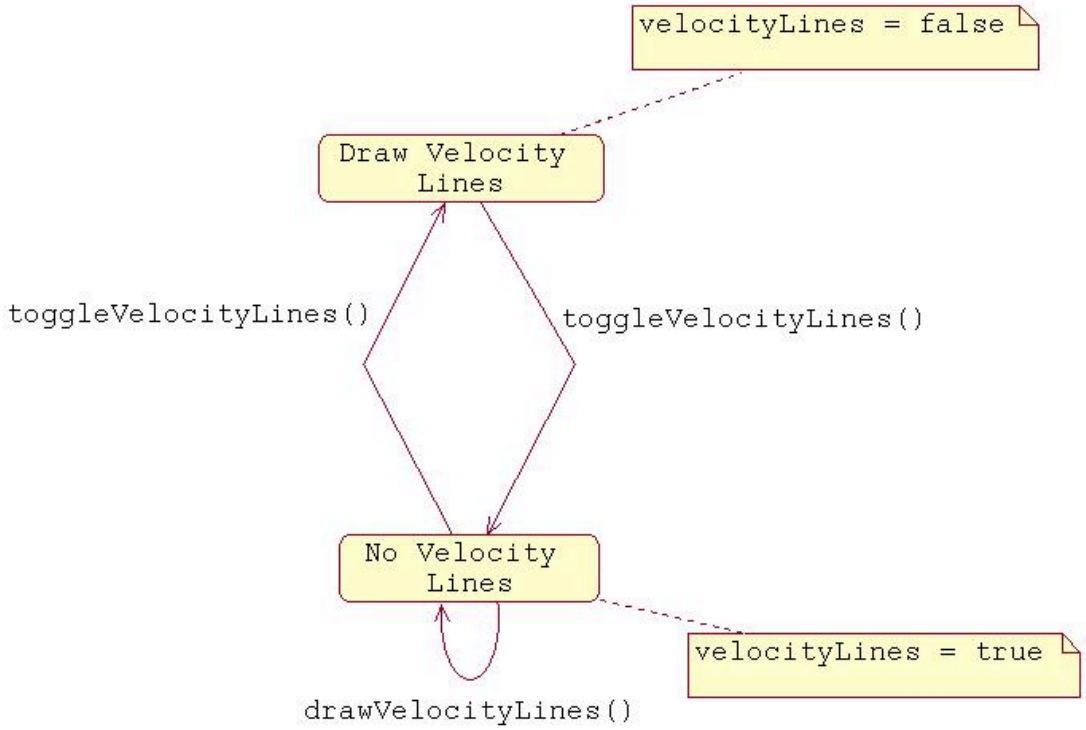
5.4.2 Toggle whether to display Solar Body names

When the System is not currently displaying the names of the Solar Bodies, the Boolean 'names' variable is false. Calling the method toggleSolarBodyNames sets the 'names' variable to true and the names are subsequently drawn each time the Display refreshes. Calling the method toggleSolarBodyNames again sets the 'names' variable back to false and the Solar Bodies names are no longer drawn when the Display refreshes.



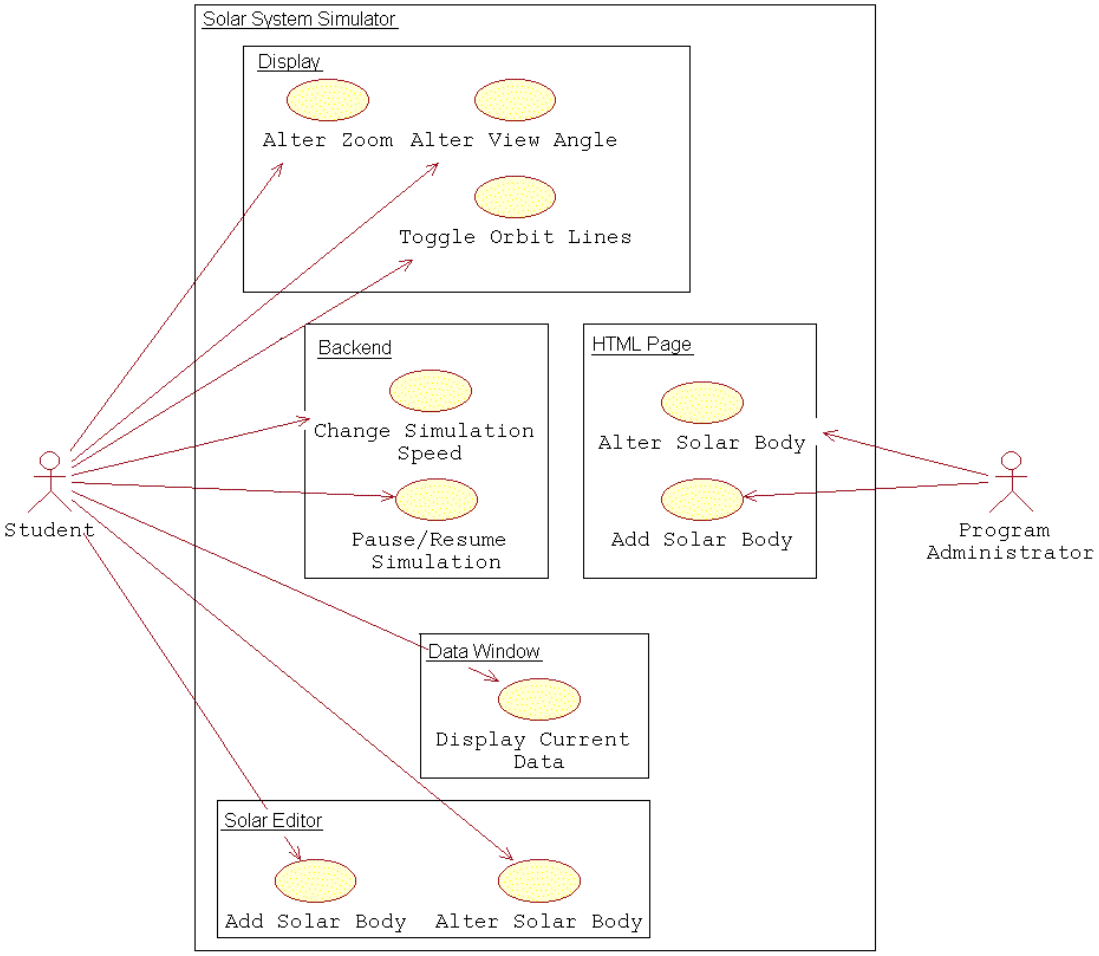
5.4.3 Toggle whether to display velocity lines

When the System is not currently displaying Velocity Lines for the Solar Bodies, the Boolean 'velocityLines' variable is false. Calling the method 'toggleVelocityLines' sets the 'velocityLines' variable to true and the Velocity Lines are subsequently drawn each time the Display refreshes. Calling the method 'toggleVelocityLines' again sets the 'velocityLines' variable back to false and the Velocity Lines are no longer drawn when the Display refreshes.



5.5 Use Case Diagram

The end user would typically be a Student running the simulation over the Internet and would manipulate the applet through the Frontend to perform the following use cases. The Program Administrator would be able to make permanent changes to the Solar Bodies my making alterations to the file that is read during the initialisation of the Applet.



6. Implementation

6.1 Overall Structure

The product is to be a visible simulation of the Solar System and as such the core of the program can be defined as two key parts, each implemented as its own class:

1) A backend engine that manipulates all the objects of the Solar System over a period of time.
2) A display that reflects the state of the backend as it moves through time.

Both parts run on their own Thread of execution independently of the other. When in execution, both run a cycle that continues uninterrupted until the program pauses or is terminated. Doing so allows the Display to refresh at regular time intervals, regardless of how many iterations the Backend has performed since the last refresh. This, combined with increasing the priority of the Display, means it stays smooth over a larger range of computers of varying performance; useful for a program that runs over the Internet.

6.2 The Thread Cycle of the Backend

For each iteration of the repeated loop of the Backend, the Boolean variable 'paused' is checked to see if a request has been made for it to pause execution. If it is paused then the wait method of the Java Thread class is invoked to suspend calculations and stop using the CPU. Otherwise, the Backend performs its necessary calculations for that time-step and then moves the time on the appropriate amount. On every iteration, the Backend is also told to sleep for a small period of time to prevent it from monopolising the computers processing power. Originally I did not include this small break in processing, but found that other programs as well as the other Threads of this program struggled to run at any reasonable speed.

On each time-step, the Backend calculates a new position, velocity and acceleration for every Solar Body in the simulation based on the existing values for these three variables. The formulae used are described as follows;

- $\mathbf{p_n}$ means position at time **n**.
- $\mathbf{v_n}$ means velocity at time **n**.
- $\mathbf{a_n}$ means acceleration at time **n**.

The formula used to calculate the next velocity is:

$$\mathbf{v_{n+1} = v_n + a_n}$$

The formula used to calculate the next position is:

$$\mathbf{p_{n+1} = p_n + v_{n+1}}$$

The formula used to calculate the acceleration of a Body **x** toward a Body **y** is:

$$\mathbf{a = (G \times M_y) / r^2}$$

where $\mathbf{G} = 6.673 \times 10^{-11}\ m^3\ kg^{-1}\ s^{-2}$
$\mathbf{M_y}$ = the Mass of Body **y** in Kg.
r = the distance between the centre of **x** and the centre of **y** in metres.

When calculating the acceleration of a Solar Body it is necessary take into account the mass of every Solar Body in the simulation rather than just one, so to get the overall acceleration for each Solar Body, the Backend calculates the sum of the accelerations towards every other Solar Body in the simulation.

6.3 The Thread Cycle of the Display

On each iteration, the display thread first re-draws to reflect the current state of the Backend, then sleeps for a short period. As with the Backend, the sleep is included to prevent the Display from using up too much CPU time. The sleep period is 100ms rather than 20 as used by the Backend, because the Display does not have to refresh as often as the Backend. The Display cannot be paused like the Backend because even when the system is paused it is sometimes necessary to re-draw the Display, for example if the zoom level or viewing angle is changed, or if the user tries to drag a Solar Body on screen.

Before re-drawing the new state, the paintComponent method of the Displays super-class is called to clear the previous state and a fresh copy of the Backend SolarSystem object is acquired as a clone so that any changes are reflected. By cloning the Solar System a 'snapshot' of a single state is used to prevent changes to the Backend affecting the Display while it is drawing. The process of cloning the Solar System includes individually cloning the data for every Solar Body. These clone methods are all synchronized with the Backend object so that the clones are made between iterations of the Backend to ensure that the data stays consistent.

Each Solar Body is drawn as a simple circle, in a position relative to the body that is central to the Display. Fields belonging to that specific Solar Body determine the colour and diameter of the circle. Boolean variables determine whether the name of the Solar Body is drawn adjacent to it as well as whether a red line representing its velocity appears.

As the simulation is based on 3D coordinates it is important that the Display maintains this. One thing taken into consideration is which Solar Bodies obscure others; if the system is being viewed from the side and the Earth goes past the far side of the sun it should not be visible but when it goes past the near side it should be. When the first dimension of a body's position shows its horizontal position in the Display and the second dimension shows its vertical position in the Display, the third dimension therefore determines how far back into the display the Body is. When rendering, anything drawn overwrites what was there before, so the last drawing of a planet in a certain position overwrites any previous drawings at that place. By drawing the furthest away Solar Bodies first, anything that should obscure them will do so when I draw it later. To achieve this, all the program does is set the order of when Solar Bodies are rendered to be the order of distance of each body's third dimension of position; the smallest value for the position first (furthest away) and the greatest value last.

The Display only renders Satellites when the zoom level is above ten percent of the maximum zoom. One reason for this is that below this zoom level, Satellites are barely distinguishable from the Planets they are orbiting and only serve to make the Display confusing. Additionally, the efficiency of the program is increased as it limits the number of bodies being drawn; when there are a large number of Planets on screen, the zoom level must be low and Satellites do not have to be drawn.

The Display inherits from a Swing component (a JPanel) and as such is double-buffered. This prevents flickering while rendering takes place, making the Simulation look smoother.

6.4 Defining Solar Bodies

Each Star, Planet and Satellite is its own Object, containing several items of data relevant only to itself, and a variety of methods, some to provide access to individual fields and others to manipulate those fields. It also seems sensible to distinguish between Stars, Planets and Satellites by putting them as separate sub-classes of an overall Structure. It would be entirely possible to have a single SolarBody class that any body in the Solar System would implement and just a field to say whether it was a Star, Planet or Satellite but this would not have worked so well; it would be much less Object-Oriented and less useful for any possible future enhanced version of the Program.

A key part of making this structure of separate Star, Planet and Satellite classes work is being able to relate them to each other. Many methods throughout the whole program act on Solar Bodies without necessarily knowing whether the body is a Star, Planet or Satellite, so they all need to inherit from a common super-class; the Solar Body interface. This Subtype Polymorphism allows Stars, Planets and Satellites to be passed into methods as the general SolarBody class and then cast into their relevant sub-class.

All the Solar Bodies in existence are bundled up into the SolarSystem class, which provides methods to access each body as well as to make changes to the system. Packaging it all together is a key part of making the program work as the whole Solar System object is needed regularly in many different classes including the Backend, Display and Data Window.

6.5 Use of 3D Coordinates

Some similar Applets on the Internet simulate orbits based on just two dimensions. This can produce a decent representation of the Planets of the Solar System as the planes of most of the planetary orbits are within a few degrees of that of the Earth. I however want a simulation that is as realistic as possible, so using three dimensions is very important to me. Additionally, in my program the Solar System can be viewed from multiple angles, which would be a pointless exercise if all the planets are rendered on a two-dimensional plane.

Basing the system on a set of three-dimensional coordinates means that it is not just positions that are stored in three dimensions; velocity and acceleration must also be stored in this way. Efficient storage of all these sets of coordinates is therefore desirable so I created a ‘Triple’ object that simply acts as a wrapper around three Java double values corresponding to X, Y and Z coordinates, and has methods to perform all the operations that I need on these values.

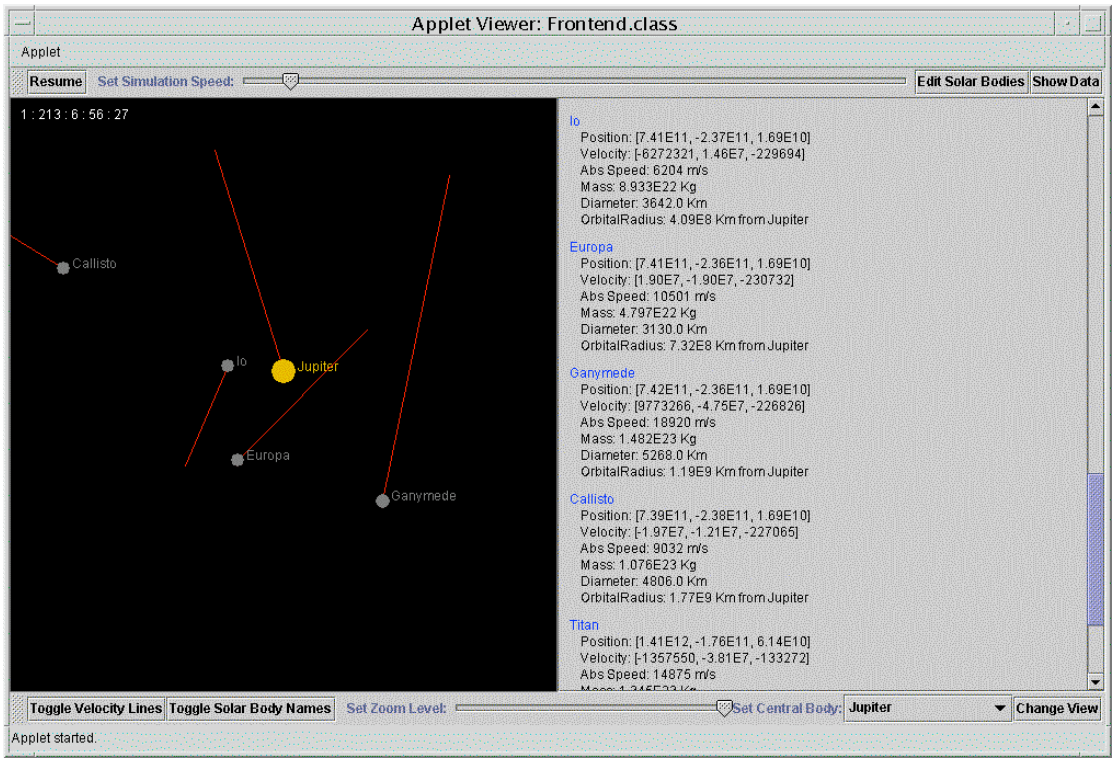
6.6 Graphical User Interface (GUI)

As previously stated, the goal of my project is to produce a program that will be used as a learning tool in the classroom and as such would probably only be used once by most people. Therefore the Graphical User Interface is critical; if it is difficult to understand or unappealing in any way then people will simply not be interested in using the program.

The GUI is structured with the goals of making the Applet look appealing, be easy to use and to give the user as much freedom as possible to interact with the system. The controls are all clearly marked including tool-tips to explain their function and are presented in a structured manner; general Applet controls appear in the top toolbar and Display controls on the bottom toolbar. Additionally I have minimised the use of text fields for user input as these require knowledge of what type of values are needed which the user may not know. Instead, slider bars and drop-down boxes are included where possible to give the user choices, which is much more friendly.

I could easily change the look and feel of the Applet by altering the colours and textures of the background and controls but did not see this as beneficial as it could easily make the Applet more daunting to a new user, and would distract attention away from the graphical display, which is supposed to be the main focus of user interaction. Using the standard Look and Feel of the Java Swing package keeps the Applet clean, clear and more professional looking.

Screenshot of the Frontend GUI



6.7 The Data Window Object

The Data Window is designed to show precise textual data about a large number of Solar Bodies and as such must be clear and easy to read. To this end, the data available is set out under sub-headings of each Solar Body. In order to prevent the window becoming too cluttered, it occupies the same amount of space on the GUI as the graphical display. As it is impossible to know in advance how many Solar Bodies may be incorporated at one time, a vertical scrollbar appears on the right-hand side when there are too many bodies to fit on a single screen.

Unlike the Display, the Data Window does not continuously update, it only does so when the user requests this by pressing the 'Show Data' button, which calls the displayCurrentData() method. It is therefore unnecessary to run the Data Window on its own Thread. The Data Window is implemented as a direct sub-class of the JPanel class, so data is written to it by overriding the paintComponent() method of the JPanel.

When the displayCurrentData() method is called, it first calls the Backend method deliverCurrentData(), which returns a clone of the SolarSystem object in its current state. A clone is required here for the same reason as for the Display; so that changes in the ongoing Backend thread do not affect the data being written by the Data Window. Once the new copy of the SolarSystem object is acquired, the paintComponent() method is called to write the data to the JPanel for the user to see.

The paintComponent() method works as follows; the single Star for the system is the first to be drawn. Its name in written in bright blue at an indentation of ten pixels horizontally and twenty vertically. The rest of the data regarding this body is written in black and with a horizontal indentation of twenty pixels. The Position is drawn as X, Y and Z coordinates in square brackets, fifteen pixels below the Name text. The Velocity too is drawn as X, Y and Z coordinates in square brackets, this time fifteen pixels below the Position text. The Absolute Speed is calculated by applying the pythag3D() method to the Velocity Triple, and then printed fifteen pixels below the Velocity text. The Mass is then written fifteen pixels below the absolute speed text and the Diameter written fifteen pixels below the mass text. The text for each Planet is then drawn underneath in turn and in the same way and then the same occurs for each Satellite. The Planets and Satellites all also print out one final piece of data which is their orbital radius which is calculated by applying the pythag3D method to the difference of its own Position triple compared to its centre-of-orbits' Position Triple.

The pythag3D method acts on a Triple according to the following mathematical formula;

$$\mathbf{A} = \sqrt{(\mathbf{X}^2 + \mathbf{Y}^2 + \mathbf{Z}^2)}$$

Where **A** = the value to be returned by the method.
X = the X component of the given Triple.
Y = the Y component of the given Triple.
Z = the Z component of the given Triple.

6.8 The Solar Editor

The Solar Editor would rarely be used, so does not warrant a permanent place within the Frontend GUI. I have therefore implemented it as a JFrame object that only appears when needed and can otherwise be hidden. Additionally it does not run on its own thread of execution as it will only be used when the simulation is not running so does not interfere with the Backend or Display.

This class potentially has the largest amount of user interaction in the system, as there are several fields of data input required from the user. In order to make this as user-friendly as possible, I used several techniques to simplify the process of using it;

- Whenever possible, Slider Bars and Drop-Down menus are included instead of Text Entry fields, as these require much less knowledge from the user about the type of data required.

- When certain fields should not be tampered with, they are de-activated to prevent errors. An example of this is not allowing the user to change the name of any Solar Body as these are what are used to identify Solar Bodies.

- When errors are made then a field on-screen is used to display any error messages, enabling the user to rectify the error before submitting the changes.

- It is possible for the user to change their mind about any changes and cancel them before the 'Commit' button is pressed by pressing the 'Reject' button.

- When specifying a new Planet from scratch, a set of example fields are included as a guide to help the user create a Planet with a stable orbit.

6.9 Key Problems and their Solutions

Several problems occurred during the implementation phase; the largest few are described here, together with my solutions.

6.9.1 Problem: Ensuring Accuracy of Initial Data

The proposed system is intended to be a learning tool so it is fundamental that anything that can be viewed is accurate and real. In order to have any chance of success I must be certain of the start positions of each planet and their initial velocities. These figures must also be to the highest precision that I can acquire and I must also be sure that they are correct.

6.9.1.1 Solution

I chose to derive the positions of the Planets and their Satellites, by calculating their distance from the Sun and the angle between the plane of their orbits and that of the Earths at some point, and then placing them in the simulation at a point that corresponds to these calculations. This does not give positions of every Solar Body correct for some single point in time but does still give one that is realistic and theoretically possible. It is then possible for test purposes to know where to expect the Solar Body to be after each full orbital period; that exact same position.

6.9.2 Problem: Scale of Display

A problem related to the Display, is the choice of scale for the planets and the inter-planetary distances. The scale of the planets is huge, but this part is no problem at all as I can just use a similarly huge scale.

One problem though is the huge difference in size between the smallest and largest planets. The smallest planet is Pluto, which has an equatorial radius of 1,150km whereas the largest, Jupiter, has a radius of 71,541km, which is more than 60 times the size. If I simulate Pluto as just a single pixel on the display, then Jupiter would still fill most of the rest of it. This does not even take into consideration the fact that the moon is smaller than Pluto and The Sun is many times the size of Jupiter.

Another problem is the enormous difference between the scale of the planets and the scale of the orbits. The best example of this is Pluto, which is 2,300km across but its mean distance from the sun is $5.9*10^9$km, over two and a half million times the distance. It is clear from this that planetary orbits must be on an entirely different scale to their physical sizes. The fact that I want the program to be as realistic as possible means that there cannot be any varying scale in the orbital distances; Mars must **look** twice as far from the Sun as the Earth is and then Jupiter must look twice as far as that.

6.9.2.1 Solution

To overcome the difference in planetary sizes, I use a varying scale where displayed sizes increase logarithmically. For example, using a base 2 log scale, if one planet were 2km in diameter it may be 1 (i.e. $\log_2(2)$) pixel on the display, then another planet that were 4km in diameter would take 2 ($\log_2(4)$) pixels on the same display. The huge difference in size of the planets means a higher log base of 10 is required.

To solve the second problem a zoom is necessary to allow the user to get a good view of the outer planets at one time, and the inner planets at a different time and a different zoom level. The maximum zoom level, which is used to display the Satellites around a single Planet, is five thousand times stronger than that needed to show the whole system.

6.9.3 Problem: Initialising program with the Data

The Backend of my system is based on a lot of data regarding each Solar Body. Hard-coding all this data into the Backend a bad idea, as future changes would be difficult to make, so my design allows this data to be stored in a single file and a class used to read this file whenever the program initialises. Unfortunately the Applet is must run over the internet, so the Java security manager rightly does not allow file read and write operations and the data is thus inaccessible when ran in a web-browser.

6.9.3.1 Solution

Java Applets have the ability to get data strings from their calling html page, through the java.Applet.getParameter() method. To this end, the contents of the data file is included as a String parameter in the html file and accessed using this method. In order for this to work, the class originally intended to read the file now simply parses the String. A possible improvement to this would be to have the html file read the data file and then pass that into the Applet as a parameter String, rather than putting the entire text file directly into the html.

6.10 Functionality implemented

All five points of the Core Functionality are captured by my final implementation.

- Point one is met, as the Engine works in all three dimensions and includes changes to position, velocity and acceleration. The accuracy of the Engine is tested in the next section although it is clear that stable orbits are achieved for all the Planets and Satellites.

- The second point of including the Sun and two Planets has been surpassed, as the Sun, all nine Planets and all Satellites above 10^{22}Kg in mass are included.

- The Display correctly renders all the Solar Bodies smoothly to meet requirement three. By running the Display on a separate Thread, it stays smooth even on slower computers.

- An accurate time-display is included as a Java GregorianCalendar class to meet point four; time is displayed in fields of years, days, hours, minutes and seconds.

- The simulation can be started and then paused and resumed at the press of a button any number of times, meeting requirement five. A reset control is unnecessary as the web-page can be re-loaded to achieve this effect.

I specified eight key requirements for my Extended Functionality, of which six were fully implemented, one was amended and one disregarded as not being beneficial to the program.

- I disregarded the first point of the extended functionality, which was to create and use data types that had a higher degree of precision than the 'double' primitive of Java. Indeed, using my own functions would very likely slow the Backend down and perhaps require me to increase the step-size to accommodate this, resulting in a loss of accuracy.

- I fully implemented the second point of the extended functionality, which was to include all Planets and Satellites above 10^{22}Kg in mass. Indeed, due to my method of implementation, adding more Planets and Satellites requires no more programming at all, by simply adding additional data in the correct format to the data String in the html file an extra Solar Body can be generated.

- The third point of the Extended Functionality, namely the zoom function, was also included. Despite being easy to use via a Slider bar on the Frontend, it is still powerful enough to allow views of the entire Solar System on minimum zoom and good views of single Planets and their surrounding Satellites on maximum zoom.

- Point four of the Extended Functionality is met, as the Display can be centred on any Solar Body by selecting from a drop-down menu, and the Display continues to follow that Solar Body until another is specified.

- The displayed sizes of Solar Bodies are representative of their actual size, which meets Extended Functionality point five. Although one Solar Body actually being ten times the size of another does not mean it will appear ten times the size, it will be a small amount bigger. Accommodating the huge differences in size of each Solar Body makes it difficult to view large amounts of the Solar System at once so my program does not do this.

- Point six of the Extended Functionality is the only point that I made a significant change to. It states that the simulation should include Orbit Lines to track the previous positions of the Solar Bodies but I feel that due to the large number of Solar Bodies in the simulation, these would not show anything clearly; a large amount of overlapping would occur making it difficult to follow any lines. Instead, Velocity Lines are used to show the current direction of each Solar Body. The limited length of these and the fact that they are always straight prevents much overlapping from occurring.

- Point seven of enabling the user to create new Solar Bodies was achieved by the Solar Editor object. All the required fields for any given body can be used, and a sample set of data is included to aid the user in creating a Planet with a stable orbit.

- Point eight of allowing the user to alter existing bodies is also met by the Solar Editor Object. All key fields such as Position, Velocity, Mass and Colour can be changed as well as which Solar Body is being orbited. The Velocity of any existing Solar Body can also be manipulated by dragging that body within the Display.

The Data Window would often be needed at the same time as the Display, so it is a permanent feature within the Frontend, whereas the Solar Editor would only be needed occasionally and when the system was paused, so I it is a separate window that pops up only when required.

7. Testing

My testing strategy is split into two sections corresponding to the two sections of the implementation. The first section is a command line program to test the Backend; Planets are sent on orbits of the sun to see how well the simulation preserves the speed and positions of the Planets over time. The second section consists of questionnaires completed by members of my target audience and other individuals judging the Frontend of the Program; how good it looked, how easy it is to use and how interesting they found using the overall program.

7.1 Testing the Backend Engine

This is achieved using my TestBackend command-line program, which takes command-line arguments of the Planet being tested, the step-size to be used by the Backend and the time to run for in seconds. When ran, the test program first prints out the initial Position Triple, Velocity Triple, Orbital Radius and Absolute Speed of the specified Planet relative to the Sun. It then runs for the specified time with the specified time step before printing the new set of data for that Planet for after the full orbit, again relative to the Sun. The change in each piece of data is then calculated by subtracting the initial value for that piece of data from the final value.

For each Planet from Mercury to Saturn I ran the program for the time it should take to complete one orbit to see how the data regarding that Planet had changed compared to its initial data, for step-sizes of 360, 3600 and 36000 seconds. 36000 seconds being the maximum possible step-size for my Applet. Testing for Planets beyond Saturn, or for step-sizes less than 360 seconds is not included, as such tests would take a very long time to complete. I tested the following four statistics;

1) The change of the coordinate position relative to the Sun, to see the actual change in position. This is mainly useful to see how much of a complete orbit has been completed or if the Planet has gone slightly too far in its orbital period.
2) The change of the coordinate velocity relative to the Sun, to see the actual change in X, Y and Z velocity components.
3) The change in the straight-line distance from the Planet to the Sun (The Orbital Radius) to see whether the Planet has much of a tendency to float away from the Sun or to be dragged towards it.
4) The change in the absolute speed of the Planet to see if it has gained or lost any kinetic energy.

It is hard to set out any exact boundaries for what makes each test result successful; the smaller the changes relative to the initial values, the more successful that particular test result is. The criteria is that the system will stay realistic over a period of time, so as long as changes are only fractions of a percent of the initial values, they are perfectly acceptable. If changes are only hundredths or thousandths (or even less) of a percent then the simulation should last for hundreds of orbits without showing any visible changes; far longer than most students would use the system for.

My System should do a good job of conserving energy as it is built from proven physical formulae. If this is so then any increase in Orbital Radius (gain of Potential Energy), should be accompanied by a decrease in Absolute speed (loss of Kinetic Energy) and vice versa.

The test results in Appendix 2 show how the state of each Planet changes after one full orbital period of that Planet for each of the following step-sizes; 360 seconds, 3600 seconds and 36000 seconds. The full output is also in Appendix 2 and the test program itself in Appendix 1.

7.1.1 Analysis of the Results

As the tests were all carried out for a single orbit, the biggest changes would be expected to be for the Planets further out such as Jupiter and Saturn, simply because the orbits are much longer so there is more time for the errors to increase.

7.1.1.1 Changes in Absolute Speed

The initial speed of the Planets tested ranged from 9640ms^{-1} for Saturn, to 47890ms^{-1} for Mercury.

% Change in Absolute Speed after 1 Orbit.

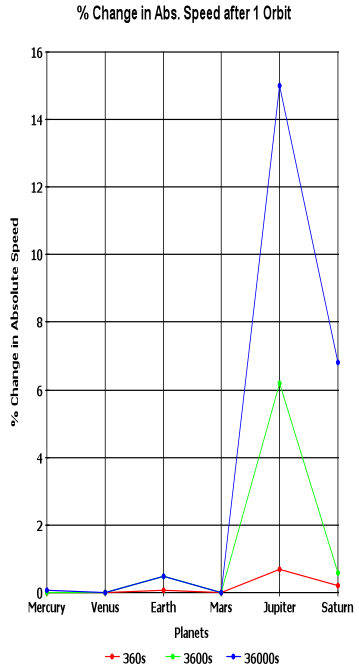
1	Mercury	0.00031	0.0011	0.062	
2	Venus	0.00014	0.00027	0.01	
3	Earth	0.084	0.48	0.49	
4	Mars	0.0092	0.0092	0.0083	
5	Jupiter	0.7	6.19	15	
6	Saturn	0.33	1.88	6.2	

When ran for a time step of 360 seconds, the biggest change in absolute speed for any Planet after a single orbit is a decrease of 91ms^{-1} for Jupiter. This compared to its initial speed of 13060ms^{-1} means a decrease by 0.69%. All of the other speed changes were a lot less, with the smallest change in speed being for Venus, which only increased in speed by 0.052ms^{-1}. This compared to its initial speed of 35030ms^{-1} means a miniscule increase of only 0.00015%. The Average change in speed is a decrease of 2.23ms^{-1} for Mars, meaning a decrease by 0.0092%, which is again tiny.

For a time step of 3600 seconds, the biggest change in absolute speed was again for Jupiter; a decrease by 808.8ms^{-1}, which is a 6.19% change; ten times more compared to the time-step of 360 seconds. The smallest change was again an increase for Venus by 0.128ms^{-1}, meaning a change of just 0.00037%; only twice the amount for a time-step of 360 seconds.

For a time step of 36000 seconds, the biggest change was still for Jupiter; this time a decrease in Speed by 1960.7ms^{-1}; a 15% change, which is definitely noticeable. The smallest change was by Mars; a decrease by 2.07ms^{-1}, which is a 0.0086% change.

From this I have to judge that increasing the time-step increases the errors in Planetary speeds, although this increase is only really noticeable for Planets whose initial errors are quite large anyway.



7.1.1.2 Changes in Orbital Radius

The initial orbital radii for the Planets tested ranges from 5.79×10^{10}m for Mercury to 1.43×10^{12}m for Saturn.

% Change in Orbital Radius after 1 Orbit.

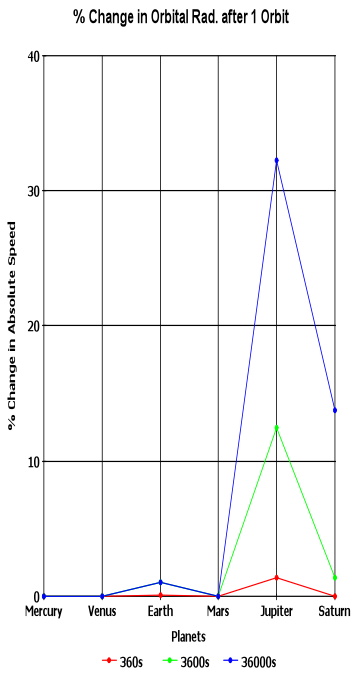
1	Mercury	0.00015	0.00078	0.017	
2	Venus	0.0012	0.0014	0.0044	
3	Earth	0.091	0.997	1.01	
4	Mars	0.0048	0.0048	0.0049	
5	Jupiter	1.34	12.5	32.3	
6	Saturn	0.034	1.25	12.6	

For a time step of 360 seconds, the biggest change in Orbital Radius for a Planet after a single orbit is an increase of 1.05×10^{10}m which, compared to its initial Orbital Radius of 1.43×10^{12}m, is a 0.73% change. As with the changes in absolute speed, the errors of Jupiter are far greater than for any other Planet tested. The second biggest change was for Saturn a 4.84×10^{8}m decrease, which is a 0.03% change compared to its initial Orbital Radius of 1.43×10^{12}m; negligible compared to the change for Jupiter. The smallest change in Orbital Radius is 8.87×10^{5}m for Mercury, which is an increase of only 0.0015% compared to its initial Orbital Radius of 5.79×10^{10}m.

For a time step of 3600 seconds, the biggest change in Orbital Radius is for Jupiter again; an increase of 9.73×10^{10}m, which is a 6.8% change. Again, the next biggest change is much smaller; an increase of 1.92×10^{10}m for Saturn, which is only a 1.34% change. The smallest change is a decrease of 4.5×10^{5}m for Mercury which is a change of only 0.0008%

For a time step of 36000 seconds, the biggest change in Orbital Radius is still for Jupiter; an increase of 2.52×10^{11}m, which is a very noticeable 17.6% change. The smallest change is a decrease of 4.74×10^{6}m for Venus, which is still just a 0.0044% change, despite the large step-size.

As with the changes in Absolute Speed, these changes increased with the step-size, but only noticeably for those Planets that had relatively large changes for small step-sizes.


% Change in Orbital Rad. after 1 Orbit
% Change in Absolute Speed
40
30
20
10
0
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Planets
360s
3600s
36000s

7.1.1.3 Changes in Position

As I separately measured fluctuations in Orbital Radius and determined that the backend does a good job of preserving distances correctly, I shall simply analyse the changes in position to determine whether the Backend has a tendency to move Planets around their Orbits too fast or too slow. If the Y-Position is 0 compared to the Sun, then the Planet has moved exactly 1 Orbit. A negative value indicates a tendency to move too quickly around the orbit and a positive value indicates a tendency to move too slowly around the orbit. The previous two sub-sections have already determined that Earth, Jupiter and Saturn do not follow such accurate orbits, so for the purposes of this test I will only judge Mercury, Venus and Mars. In order to have a common ground to judge on, I will calculate the angle, θ, of how far extra or short round the orbit the Planet has moved, which can be used to calculate the percentage change when compared to a full orbit of 360^o. The piece of Trigonometry used to calculate θ is:

Sin θ = Change in Y-Position / Final Orbital Radius

For the step-size of 360 seconds, Mercury has a Y-Value for positional change of 3.63×10^8m, giving a value for θ of 0.36^o, Venus has a Y-Value for positional change of 2.04×10^8m, giving a value for θ of 0.11^o and Mars has a Y-Value for positional change of -3.69×10^8m, giving a value for θ of -0.093^o. All of these angles are microscopic because the largest of these angles represents just 0.1% of an orbit.

For the step-size of 3600 seconds, Mercury has a Y-Value for change in position of 3.99×10^8m, giving a value for θ of 0.39^o, Venus has a Y-Value for positional change of 3.05×10^8m, giving a value for θ of 0.16^o and Mars has a Y-Value for change in position of -3.26×10^8m, giving a value for θ of -0.082^o. All of these angles are still microscopic despite the larger step-size, the largest being 0.39^o, which is just 0.11% of an orbit.

For the step-size of 36000 seconds, Mercury has a Y-Value for positional change of 7.98×10^8m, giving a value for θ of 0.79^o, Venus has a Y-Value for positional change of 6.22×10^8m, giving a value for θ of 0.33^o and Mars has a Y-Value for change in position of 2.96×10^8m, giving a value for θ of 0.074^o. Despite this being the maximum value for step-size and having limited accuracy, the angles are all still tiny as the largest is still only 0.79^o, which is just 0.22% of an orbit.

This all indicates that orbit times are very accurate providing there is nothing sending the Planets off their natural orbital paths, such as large Satellites.

7.1.2 Conclusions

The backend performs well with the smallest step size of 360 seconds, although it generally does nearly as well for the larger step sizes. The only times it performs poorly are on Jupiter and to a lesser extent Earth and Saturn, and only when the step size is particularly large. I have not analysed the changes in X, Y, and Z Velocity vectors as the Positions, Orbital Radii and Absolute Speeds prove the accuracy of the Backend.

7.1.2.1 Reasons for errors

One reason why the Backend performs less well on some Planets could simply be due to errors in the initial data, probably due to the limited known accuracy of those figures, leading to increases in error over time. Another more likely reason is that the three worst performers are the ones that have Satellites orbiting them and that the Satellites could affect the orbits enough to put them off track. This is also supported by the fact that the Planet with the highest mass of Satellites in my simulation, Jupiter, is also the one affected most. Running the simulation with no Satellites can easily test this. I will try it on Jupiter, with step-size of 36000 seconds, as this is the one that goes most out of position.

Changes for Jupiter with step-size 36000s and no Satellites:
Position: [1.239868851105957E8, -8.392075892194791E9, -1085165.9964141846]
Velocity: [-125.59261211764293, 5.370040102992789, -2.7193222242858575]
Orbital Radius: 1.691634745192871E8 Relative to Sun
Abs Speed: -4.7656363904206955 Relative to Sun

Here, the change in Orbital Radius is only 0.022% of its initial Orbital Radius, compared to 32.3% with satellites, meaning the accuracy has improved by about 1500 times! The Absolute Speed only changes by 0.036%, compared to 15% when Satellites were included, meaning the accuracy of this has improved by over 400 times. It is plain from this that the simulation is indeed much more accurate without the Satellites, the most probable reason being that the Satellites all start on the same side of Jupiter and are initially all pulling Jupiter away from the Sun. To improve the simulation, it would be a useful to start with the satellites more spread out around the orbit of Jupiter.

7.1.2.2 Running the Simulation with step-size of 1s

It would be interesting to know how well the simulation runs on maximum accuracy (minimum step size). To this end I have tested the simulation for a single orbit of Mercury with step-size of 1 second to see how much more accurate it is compared to step-size 360.

Changes for Mercury with step-size 1s for 1 Orbit:
Position : [-902788.4025115967, 3.487879169525821E8, -96719.44628810883]
Velocity: [286.352756962245, 1.062555104806961, 35.14099179781182]
Orbital Radius: 142521.67582702637
Abs Speed: -0.19354401388409315

Here, the change in Orbital Radius is by 0.00025%, compared to 0.00015% for step-size 360. Which is a slight decrease in accuracy but both values are so small it can easily be explained as errors in the starting figures. The change in Absolute speed is

by 0.00041%, compared to 0.00031% for step-size 360. Again the only difference between these two values is probably just due to initial errors rather than errors with the Backend.

7.1.2.3 Running the Simulation over a longer period of time

All the previous tests were just for a single orbit but this program is required to work well for a longer period than that. To this end I shall test it for 100 Orbits of Mercury at the maximum time-step of 36000s.

Changes for Mercury with step-size 36000s for 100 Orbits:
Position:[-2.848232318323985E10,4.925183681144157E10,-3.496537704341908E9]
Velocity: [41920.298042228904, 24163.287989386434, 5148.867761842838]
Orbital Radius: -6.45834200378006E8
Abs Speed: 553.5660331982435

The change in Orbital Radius is by 1.13% and the change in Absolute Speed just 1.16% despite the long time frame which shows my simulation is capable for working accurately for many orbits. I expect it to be quite unusual for someone to want to run it for anything like that long anyway, so the simulation is quite safe for any reasonable use and quite a bit beyond.

7.2 Testing the Frontend

Assessing the success of the Frontend is a rather more subjective task than for the Backend as there are no absolute values to compare and analyse. The success of the Frontend can be judged on the three requirements of being easy to use, informative and interesting to use. Members of my Target Audience can best judge these, so I compiled a short questionnaire and found six few individuals to fill out a copy while using my Simulation. The questionnaires can be found in Appendix 4.

7.2.1 Ease of Use

Questions 4, 5 and 6 of the Questionnaire regard the ease of use of the Applet. I ask each participant to rate the ease of use out of 5; 1 meaning very easy and 5 very hard. The scores given were as follows;

- 4 people gave it 1 (maximum score).
- 2 people gave it 2.

This averages out at 1.33, almost the highest possible score on this scale. The participants particularly liked the following features, or found them helpful in making the Program easy to use;

- Changing the Central Solar Body using the drop-down menu.
- Zooming in and out with the Slider bar.
- Tool-Tips for all the controls.
- Clear and realistic Time Display.
- Easy change of view via button.

I also asked for possible suggestions of improvements to make the simulation more easy to use. The responses included the following;

- 'Change View' button changed to a drop-down menu to make it more obvious which view is currently in use.
- A help feature to describe in more detail what can be achieved.
- Allow central Planet selection in a way similar to html hyperlinks via the Data Window.
- Enable view dragging using the mouse in the Display.

Implementing the view change for different angles as a drop-down menu rather than a button is a very minor change that can easily be achieved in a future version of the Applet.

Extra help features are potentially a major improvement to the system, but as such would take a large amount of time to fully implement. The current descriptions on the web-page, together with the tool-tips proved sufficient for all my test subjects to successfully use the program without needing guidance from me or anyone else.

The third suggestion of allowing Planet selection direct from the Data Window is a very exciting idea and would certainly be possible to implement in a future version. I believe the main steps involved are as follows;

- Enable mouse click events on the Data Window Object.
- Identify areas of the Data Window where click events should trigger a response.
- Produce the response of centring the Display on the identified Solar Body.

The fourth idea of enabling view change by dragging within the Display is probably much less feasible. One problem is that it is already possible to drag within the Display with the effect of altering the velocity of Solar Bodies, so it would become difficult to determine which task the user is trying to do when dragging. Another problem is that the Display has just three discrete viewing angles, but changing the view as a response to a drag-event seems much more appropriate for a continuous slide between views.

7.2.2 Look and Feel

Questions 7, 8 and 9 of the Questionnaire regard the Look and Feel of the Applet. I ask each participant to rate the Look and Feel of the Applet out of 5; 1 meaning very attractive and organized and five very unattractive with bad organization. The scores given are as follows;

- 1 person gave it 1 (maximum score).
- 3 people gave it 2.
- 2 people gave it 3 (average).

The average score is therefore 2.17 which is comfortably above an average score but still not as good as the score for ease of use.

The features of the Look and Feel that the participants particularly like are;

- The range of Colours used for the Planets, while still looking realistic.
- The organization of the Data Window, together with the scroll-bar.
- The extra information offered by the Velocity Lines
- Buttons and Toolbars are easily accessible.
- Ability to change the Velocity within the Window is neat.
- Professional looking with large display windows.

Again, I asked for suggestions of possible improvements that could be made in a future version. Suggestions include the following;

- More colourful presentation.
- Be able to zoom in on any region by clicking in the Display.
- Larger Display window to see more at once.
- Improve control layout.
- Put textures on Planets to improve 3D feel.

It is possible to make the presentation more colourful but is tricky to balance clean, clear organization against a colourful, fun presentation. Different peoples tastes affect which they feel is better, although the fact that this is aimed at College Students should guide the decision more towards colour and fun.

Enabling zooming by clicking in the Display is certainly possible, but this is another case where it is difficult to interpret whether a user clicking is attempting to drag a Solar Body, or to Zoom in on that location.

A larger Display is certainly possible but would probably involve re-designing the Frontend structure to fit the Data Window and all the other controls around it correctly.

Changes to the layout of the controls is certainly possible. They are currently separated into two toolbars as there are a large number of them but any changes could be suggested for other students to decide on. Implementing the changes is simply a matter of changing the GUI code within the Frontend class.

Adding textures to the Planets is also possible. One method is to acquire some GIF or JPG photos of each Planet, paste these onto Panels and then move the Panels around the Display instead of the circles I currently use. Swapping through a sequence of photos of each Planet, taken from different angles could give the impression of the Planets spinning.

7.2.3 Using the Solar Editor

As the Solar Editor has the potential for the most user interaction, ensuring it is easy to use is very important, so this section comprising the answers to questions 10, 11 and 12 concentrates on using the Solar Editor Frame. I specified a task for each participant to do using the Solar Editor and then had them rate how easy the Solar Editor made this task, 1 being very easy and 5 being very hard. The following results were yielded;

- 2 people gave it 1 (very easy).
- 3 people gave it 2.
- 1 person gave it 3 (average).

This means the average score is 1.83, which is again high. More importantly **none** of my participants said they found this particularly difficult to do.

The following features of the Solar Editor were identified by the participants as helpful in making it easy to use;

- Error messages displayed when mistakes were made in the input.
- Display to give an indication of the Colour created by the Sliders.
- Ability to reject changes.
- Slider bars for colours made it quick and easy to decide.
- Drop-down boxes were helpful where provided.
- Overall structure is well-organised.
- Putting on a separate window is useful due to the number of controls.

The following suggestions were made as possible improvements that could be made to the Solar Editor program;

- Help function specifically for this window.
- Warning message when ‘Commit’ button pressed to allow the user to change their mind.
- Make more obvious what the current colour is.
- Make it more clear what Scientific notation is and what values like 1E30 mean.

As described earlier, extra help features are a potential major improvement to the system, but would take a large amount of time to fully implement. The current descriptions on the web-page, together with the tool-tips were sufficient for all my test subjects to successfully use the program without needing any guidance at all.

Adding a small warning message when the ‘Commit’ button is pressed is another small enhancement that can be easily implemented in a future version of the Program.

A couple of the participants did not notice the small text field that represents the current colour, so this could easily be changed to be larger and thus more visible.

A description of what scientific notation is and how I use it in the simulation could easily be added to the web-site to make its use more clear to potential users.

8. Conclusion

The goal of the project was to produce a program suitable for use in a classroom as a learning tool about the effects of Gravity in an N-Body simulation. Important features of the system are an accurate engine to manipulate the Planets realistically in three dimensions, a clear graphical display capable of showing all the interactions within the Solar System, and a set of controls to allow the user to manipulate the simulation to a high degree.

8.1 Assessment of success

Assessing the amount of the Core and Extended functionality that has been achieved gives an idea of how complete the program is and helps to judge the coding part of the project. Additionally, assessing the program against the requirements of the audience gives a good assessment of the success of the design of the product, although it does also indicate the strength of the coding phase too.

The entire core, and most of the extended functionality have been implemented; a description of the level of Implementation is in section 6.10. The only two changes were made solely to improve the program from the design.

I originally identified five key requirements of the product for it to be appropriate for use by the intended target audience; being free and simple to get hold of, easy to use, informative, accurate and be able to hold the interest of the user.

1 Being free and simple to get hold of is a very clear-cut requirement, which can easily be met; it will not be charged for, so is free, and is an Applet available over the Internet so should be simple to get hold of. The success of this requirement can only be judged over time when it is clear whether many people use the Applet.

2 The questionnaires show that the sample of my target audience I tested found the Applet very easy to use (see section 7.2.1). There is no evidence to suggest that this view is not representative of the feelings of the majority of the Target Audience.

3 The requirement of being informative is much harder to judge; its performance over time would be the best judge of this. The fact that the data exists within the Applet means that information is readily available and it is certainly possible to learn from using the simulation.

4 The program is sufficiently accurate for its intended purpose of representing the distances, speeds and sizes of the Planets as well as demonstrating the effects of Gravity. I cannot claim that it shows the **exact** data from real life, as that would be impossible to achieve, however it is still sufficiently accurate to fulfil its purpose and for the simulation to last a long time before the orbits decay.

5 Being able to hold the interest of the user is another requirement that can only be judged over time when the program is being used by people. The fact that the members of my Target Audience who participated in the questionnaires liked the Look and Feel (see section 7.2.2) of the Applet is a positive indication.

8.2 Suggestions for extensions.

Like many programming projects it is never possible to say that this one is 'finished'. This one is simply at a stage where it fulfils all the main purposes, but there still exist many possible extensions and improvements.

8.2.1 More Solar Bodies

One very open ended extension for this is to improve the range and diversity of Solar Bodies included; there are over a dozen more satellites around each of Jupiter, Saturn and Uranus, plus others around Mars, Neptune and Pluto. There are many Comets and other bodies that follow Parabolic and Elliptical orbits within our Solar System that could easily be added too. Also, why stick to our Solar System – there is an almost infinite Universe that could be modelled if we had the resources and knowledge of what is out there. For any large extension of this kind it would probably be necessary to build in a tool that prompts the user to select which Solar Bodies they want in their simulation, otherwise the Display may get very cluttered or the added strain of calculating for so many bodies would slow the simulation down too much.

8.2.2 Improve Positions of Solar Bodies

When testing the Backend it became obvious that the Satellites had a dramatic effect on those Planets they were orbiting. The problem is not due to the inclusion of the Satellites as they have the same effect in reality, the problem is the initial positioning of them. The initial positions of all the Solar Bodies have been derived from the locations of what they are orbiting added to their distance from that Solar Body. This has led me to place all the Planets and Satellites in a straight line along the X-axis. The best way to improve this is to work out the exact coordinate positions and velocities of each Solar Body at a specific point in time and to use that as my start state, however this requires much research and calculation.

8.2.3 Extend the functionality of the Display

It is already possible to zoom in and out in the Display, select from three angles to view the system and have the Display track any Solar Body in the simulation. One extension is to allow the viewing angle be from a choice of more than three. Scroll-Bars are a possible means for the user to scroll **between** views, choosing any angle between those currently used.

9. References

1. Grady Booch. (1994). *Object-Oriented Analysis and Design with Applications*. 2nd Edn. Addison-Wesley.

2. http://www.humnet.ucla.edu/humnet/french/faculty/gans/java/SolarApplet.html

3. http://burtleburtle.net/bob/physics/solar.html

4. http://www.particle.kth.se/~fmi/kurs/PhysicsSimulation/Lectures/13A/

5. http://jove.geol.niu.edu/faculty/stoddard/JAVA/luminaries.html

6. http://www.amherst.edu/~gsgreens/astro/Moon.html

7. http://www.astrodigital.org/astronomy/ss.html

8. Focus Multimedia. (2000). *RedShift 4*. (CD-Rom).

Appendix 1 - Code Listing

Backend.java

```
/**
 * The Backend class holds all the data for the system, including the positions,
 * velocities and accelerations of each Planet, Star and Satellite.  Methods exist
 * to alter each Solar Body's data in such a way as to simulate the moving orbits
 * of each.
 *
 * @author Adam Haigh
 * @version 1.2  21/1/03
 */

import java.io.*;
import java.util.*;

public class Backend extends Thread{

    private boolean isAlive;
    private boolean paused;
    private int scaler;     // The time to scale the Acceleration and Velocity by in
seconds
    private GregorianCalendar time;
    private Display display;
    private SynchronizedStar star;
    private SynchronizedPlanet[] planets;
    private SynchronizedSatellite[] satellites;
    private SolarSystem solar;
    private final double G = 0.00000000006673;  // Universal Gravitational Constant
    // = 6.673 * 10^-11 m^3 kg^-1 s^-2

    /**
     * Units of measurement will be Kg, Metres, Seconds.
     *
     * @param text The data String to produce the Solar System from.
     */
    public Backend(String text){

        try{
            // Set up the Solar System
            solar = SolarSystemReader.parseString(text);
        }
        catch(SolarSystemIOException e){

            System.out.println(e.getMessage());
        }

        // Set up the time of the system.
        time = new GregorianCalendar();
        time.set(Calendar.YEAR, 1);
        time.set(Calendar.DAY_OF_YEAR, 1);
        time.set(Calendar.HOUR, 0);
        time.set(Calendar.MINUTE, 0);
        time.set(Calendar.SECOND, 0);

        scaler = 1; // Set time-step to 1 second

        star = solar.getStar();
        planets = solar.getPlanets();
        satellites = solar.getSatellites();

        // Set running off pause.
        isAlive = true;
        paused = false;
    }

    /**
     * Called by the Owner of this Thread.
     * Starts execution
     */
    public void run(){
```

```
        // Keep looping until told to stop
        while(isAlive){

            if(!paused){
                // Do the calculations
                timeStep();
                time.add(Calendar.SECOND,scaler);
            }
            else{
                // Suspend calculations
                try{

                    synchronized(this){

                        this.wait();
                    }
                }
                catch(InterruptedException e){ }
            }

            try{
                // Sleep during each loop to free up processor time.
                this.sleep(20);
            }
            catch(InterruptedException e){ }
        }
    }

    /**
     * Destroys the Backend Thread by allowing the run() method to terminate
naturally.
     */
    public void destroy(){

        isAlive = false;
        unPause();
    }

    /**
     * Suspend calculations until told to resume.
     */
    public void pause(){

        paused = true;
    }

    /**
     * Resume all calculations.
     */
    public void unPause(){

        paused = false;

        synchronized(this){

            this.notify();
        }
    }

    /**
     * Add this SolarBody to the Solar System.
     *
     * @param body The SolarBody to add to the System.
     */
    public synchronized void addSolarBody(SolarBody body){

        // Set the acceleration of the Solar Body.
        body.changeAcceleration(totalAcceleration(body));

        // Add the Solar body to the correct part of the Solar System object.
        if(body instanceof Star){

            star = (SynchronizedStar) body;
            solar.addStar(star);
        }
        else if(body instanceof Planet){
```

```
            solar.addPlanet((SynchronizedPlanet) body);
            planets = solar.getPlanets();
        }
        else if(body instanceof Satellite){

            solar.addSatellite((SynchronizedSatellite) body);
            satellites = solar.getSatellites();
        }
    }

    /**
     * Calculate the new Position, Velocity and Acceleration of each SolarBody
     * based on the previous time step.
     */
    public synchronized void timeStep(){

        // Set new values for the star.
        recalculateData(star);
        star.changeAcceleration(totalAcceleration(star));

        // Set new values for the Planets
        for(int i=0; i<planets.length; i++){

            recalculateData(planets[i]);
            planets[i].changeAcceleration(totalAcceleration(planets[i]));
        }

        // Set new values for the Satellites
        for(int i=0; i<satellites.length; i++){

            recalculateData(satellites[i]);
            satellites[i].changeAcceleration(totalAcceleration(satellites[i]));
        }
    }

    /*
     * Recalculate the Position and Velocity of the given SolarBody.
     *
     * @param body The SolarBody to be recalculated.
     */
    private synchronized void recalculateData(SolarBody body){

        // Find the last Position, Velocity and Acceleration of the SolarBody
        Triple oldPos = body.getPosition();
        Triple oldVel = body.getVelocity();
        Triple oldAcc = body.getAcceleration();

        Triple newVel = addTriples(oldVel, oldAcc);

        // Calculate a new Velocity and Position
        body.changePosition(addTriples(oldPos, newVel));
        body.changeVelocity(newVel);
    }

    /*
     * Recalculate the total vector acceleration of the given body by
     * adding the individual accelerations towards all other Solar Bodies.
     *
     * @param body The SolarBody to be recalculated.
     * @return The new acceleration for the given SolarBody.
     */
    private synchronized Triple totalAcceleration(SolarBody body){

        // Create a new Acceleration Triple.
        Triple total = new Triple(0,0,0);


        // Calculate acceleration to all solar bodies other than itself

        if(!body.equals(star)) total = calculateAcceleration(body, star);

        for(int i=0; i<planets.length; i++){

            if(!body.equals(planets[i]))
                total = addTriples(total, calculateAcceleration(body, planets[i]));
        }
```

```
        for(int i=0; i<satellites.length; i++){

              if(!body.equals(satellites[i]))
                total = addTriples(total, calculateAcceleration(body, satellites[i]));
        }

        return total;
    }

    /*
     * Recalculate the Acceleration of body1 towards body2.
     *
     * @param body1 The SolarBody that is accelerating.
     * @param body2 The SolarBody being accelerated to.
     *
     * @return The Acceleration of body1 towards body2 as a Triple
     */
    private synchronized Triple calculateAcceleration(SolarBody body1, SolarBody
body2){

        // Calculate the Straight line distance between the 2 SolarBodys
        double orbitalRadius = calculateRadius(body1.getPosition(),
body2.getPosition());

        // Calculate the Gravitational Strength between the 2 SolarBodys
        double gravStrength =
            (Math.pow(scaler,2.0) * G  * (body2.getMass())) / (Math.pow(orbitalRadius,
2.0));

        double[] distanceFractions =
            getDistanceFractions(body1.getPosition(), body2.getPosition());

        // Calculate the new Acceleration for the Planet.
        return new Triple(gravStrength*distanceFractions[0],
gravStrength*distanceFractions[1], gravStrength*distanceFractions[2]);
    }


    /*
     * Takes two Triples, each corresponding component is added and returned as
     * part of a new Triple
     *
     * @param first The first Triple to be added.
     * @param second The second Triple to be added.
     *
     * @return The Triple whose values are equal to the sum of the corresponding
values in first and second
     */
    private synchronized Triple addTriples(Triple first, Triple second){

        double[] f = first.getTriple();
        double[] s = second.getTriple();

        for(int i=0; i<3; i++)
            f[i] = f[i] + s[i];

        return new Triple(f);
    }


    /**
     * Given 2 Triples of Coordinates, calculate the straight line distance
     * between the 2.
     *
     * @param t1 The First Triple being added.
     * @param t2 The Second Triple being added.
     *
     * @return The distance as a double.
     */
    public synchronized double calculateRadius(Triple t1, Triple t2){

        double x = t1.getX() - t2.getX();
        double y = t1.getY() - t2.getY();
        double z = t1.getZ() - t2.getZ();

        return(Math.sqrt(Math.pow(x, 2.0) + Math.pow(y, 2.0) + Math.pow(z, 2.0)));
```

```
    }


    /*
     * Work out the distance between two Triples as a fraction of the total distance.
     *
     * @param t1 The first Triple being added.
     * @param t2 The second Triple being added.
     *
     * @return Three doubles in an array corresponding to the distance fractions.
     */
    private synchronized double[] getDistanceFractions(Triple t1, Triple t2){

        double x = t2.getX() - t1.getX();
        double y = t2.getY() - t1.getY();
        double z = t2.getZ() - t1.getZ();

        double total = Math.sqrt(Math.pow(x, 2.0) + Math.pow(y, 2.0) + Math.pow(z,
2.0));
        double[] answer = new double[3];
        answer[0] = x/total;
        answer[1] = y/total;
        answer[2] = z/total;

        return answer;
    }

    /*
     * Multiply each field of the Triple by the value of scaler.
     *
     * @param t The Triple to be multiplied.
     * @param scaler How much to multiply each element of the Triple by.
     *
     * @return The multiplied Triple.
     */
    private synchronized Triple multiplyTriple(Triple t, double scaler){

        double[] d = t.getTriple();

        for(int i=0; i<3; i++) d[i] = d[i] * scaler;

        return new Triple(d);
    }

    /**
     * Creates a 'snapshot' of the System in its current state.
     *
     * @return A clone of the system rather than just a pointer to the original.
     */
    public synchronized SolarSystem deliverCurrentData(){

        return (SolarSystem) solar.clone();
    }


    /**
     * Set the length of each timestep to be stepSize.
     *
     * @param stepSize The number of seconds jumped in each timestep
     */
    public synchronized void setTimeStep(int stepSize){

        double newStep = (double) stepSize/scaler;

        // Scale the velocity of the Star to the amount specified by scaler
        star.changeVelocity(multiplyTriple(star.getVelocity(), newStep));
        star.changeAcceleration(multiplyTriple(star.getAcceleration(),
Math.pow(newStep,2.0)));

        // Scale the velocity of the Planets to the amount specified by scaler
        for(int i=0; i<planets.length; i++){
            planets[i].changeVelocity(multiplyTriple(planets[i].getVelocity(),
newStep));
            planets[i].changeAcceleration(multiplyTriple(planets[i].getAcceleration(),
Math.pow(newStep,2.0)));
        }
```

```
        // Scale the velocity of the Satellites to the amount specified by scaler
        for(int i=0; i<satellites.length; i++){
            satellites[i].changeVelocity(multiplyTriple(satellites[i].getVelocity(),
newStep));

satellites[i].changeAcceleration(multiplyTriple(satellites[i].getAcceleration(),
Math.pow(newStep,2.0)));
        }

        scaler = stepSize;
    }

    /**
     * Get the size of each timestep in seconds.
     *
     * @return The current timestep in seconds.
     */
    public double getTimeStep(){

        return scaler;
    }

    /**
     * Get the simulated time that the system has been running for.
     *
     * @return A string representation of the Time Calendar.
     */
    public String getTime(){

        return time.get(Calendar.YEAR) + " : " + time.get(Calendar.DAY_OF_YEAR) + " : "
+ time.get(Calendar.HOUR) + " : " + time.get(Calendar.MINUTE) +  " : " +
time.get(Calendar.SECOND);
    }

    /**
     * Get a pointer to the SolarSystem.  This is not a clone.
     *
     * @return A pointer to the SolarSystem being used by the Backend.
     */
    public SolarSystem getSolarSystem(){

        return solar;
    }
}
```

DataWindow.java

```
/**
 * The DataWindow class is the window that shows the current
 * state of the Solar System in text form as a Panel in the Applet
 *
 * @author Adam Haigh
 * @version 1.1 10/02/2003
 */

import javax.swing.*;
import java.awt.*;

public class DataWindow extends JPanel{
    private SolarSystem system;
    private Backend backend;
    private int width;
    private int height;

    /**
     * @param b The backend used to find data to be displayed.
     */
    public DataWindow(Backend b){

        backend = b;
        width = 450;
        height = 20 + backend.getSolarSystem().getNumberOfSolarBodys() * 115;
        setPreferredSize(new Dimension(width, height));
    }

    /**
     * Gets a copy of the current state from the Backend and
     * shows this textually.
     */
    public void displayCurrentData(){

        system = backend.deliverCurrentData();
        this.repaint();
        this.setVisible(true);
    }

    /**
     * Display text according to the current SolarSystem state.
     * Should only ever be called by the repaint() method.
     *
     * @param g The current Graphics context.
     */
    public void paintComponent(Graphics g){

        // Removes everything drawn by last call to repaint()
        super.paintComponent(g);

        int posY = 20;

        g.setColor(Color.blue.darker());
        g.drawString("Solar System Information at time " + backend.getTime() + ":", 10,
posY);
        posY += 20;

        try{
            // Go through each SolarBody in turn drawing its data.
            posY = bodyData(system.getStar(), g, posY);

            SynchronizedPlanet[] planets = system.getPlanets();
            for(int i=0; i<planets.length; i++)
                posY = bodyData(planets[i], g, posY);

            SynchronizedSatellite[] satellites = system.getSatellites();
            for(int i=0; i<satellites.length; i++)
                posY = bodyData(satellites[i], g, posY);

        }
        catch(NullPointerException e){

            System.out.println(e.getMessage());
```

```
        }
    }

    /*
     * Draw data for a specified SolarBody into the window.
     *
     * @param body The SolarBody whose data is to be drawn
     * @param g The current Graphics context being used.
     * @param posY The Number of pixels down the Window to start drawing.
     * @return The Number of pixels down the window to draw the next SolarBody.
     */
    private int bodyData(SolarBody body, Graphics g, int posY){

        // Draw the SolarBodys Name
        g.setColor(Color.blue.brighter());
        g.drawString(body.getName(), 10, posY);
        posY += 15;

        // Draw the SolarBodys Position
        g.setColor(Color.black);
        g.drawString("Position: [" + toShortString(body.getPosition().getX()) + ", " +
toShortString(body.getPosition().getY()) + ", " +
toShortString(body.getPosition().getZ()) + "]", 20, posY);
        posY += 15;

        // Draw the SolarBodys Velocity
        g.drawString("Velocity: [" + toShortString(body.getVelocity().getX()) + ", " +
toShortString(body.getVelocity().getY()) + ", " +
toShortString(body.getVelocity().getZ()) + "]", 20, posY);
        posY += 15;

        // Draw the absolute speed of the SolarBody
        g.drawString("Abs Speed: " +
toShortString(pythag3D(body.getVelocity())/backend.getTimeStep()) + " m/s", 20, posY);
        posY += 15;

        // Draw the SolarBodys Mass
        g.drawString("Mass: " + body.getMass() + " Kg", 20, posY);
        posY += 15;

        // Draw the SolarBodys Diameter
        g.drawString("Diameter: " + body.getSize()/1000 + " Km", 20, posY);
        posY += 15;

        // Draws the SolarBodys OrbitalRadius
        try{
            OrbitingSolarBody orbitingBody = (OrbitingSolarBody) body;
            SolarBody centre = orbitingBody.getCentreOfOrbit();
            g.drawString("OrbitalRadius: " +
toShortString(backend.calculateRadius(orbitingBody.getPosition(),
centre.getPosition())) + " Km from " + centre.getName(), 20, posY);
            posY += 15;

        }
        catch(ClassCastException e){ }

        return posY + 10;
    }

    /*
     * Find the straight line length from [0, 0, 0] to t
     *
     * @param t The Triple whose value is to be used.
     * @return The straight line lingth as a double.
     */
    private double pythag3D(Triple t){

        return Math.sqrt(Math.pow(t.getX(),2) + Math.pow(t.getY(),2) +
Math.pow(t.getZ(),2));
    }

    /*
     * Multiply each element of a Triple by the value of scaler.
     *
     * @param t The Triple whose values need to be multiplied.
     * @param scaler The amount to multiply each element of the Triple by.
     * @return The new Triple corresponding to t * scaler.
```

```
     */
    private Triple multiplyTriple(Triple t, double scaler){

        double[] d = t.getTriple();

        for(int i=0; i<3; i++) d[i] = d[i] * scaler;

        return new Triple(d);
    }


    /*
     * Converts a double format number to one with less sig figs suitable for
printing.
     */
    private String toShortString(double d){

        char[] chars = Double.toString(d).toCharArray();
        StringBuffer sb = new StringBuffer();
        int i = 0;

        if(chars[i] == '-') sb.append(chars[i++]);

        sb.append(chars[i++]);
        sb.append(chars[i++]);

        // if its a fraction or in index form
        if(chars[i-1] == '.'){

            sb.append(chars[i++]);

            try{

                if(chars[i] != 'E') sb.append(chars[i++]);

                while(chars[i] != 'E') i++;

                for(int j=i; j<chars.length; j++) sb.append(chars[j]);
            }
            catch(ArrayIndexOutOfBoundsException e){ }

        }
        else{

            for(int j = i; j<chars.length; j++){

                if(chars[j] == '.') break;

                sb.append(chars[j]);
            }
        }

        return sb.toString();
    }
}
```

Display.java

```
/**
 * The Display class is the Window that visually shows the current state
 * of the Solar System as a Panel in the Applet.
 *
 * @author Adam Haigh
 * @version 1.1 10/1/03
 */
import javax.swing.*;
import java.awt.*;

public class Display extends JPanel implements Runnable{
    private boolean isAlive;
    private boolean velocityLines;
    private boolean solarBodyNames;
    private Backend backend;
    private SolarSystem solar;
    private double halfSize; // half the size of the solar system in metres
    private int halfWindow = 250; // half the size of the Display window in Pixels
    private double halfDisplay; // half the size of the displayable solar system in
metres
    private double sizeRatio; // Ratio between window size and Solar System size
    private boolean drawSatellites = false;

    public static final double solarBodySizeScaler = 30000;
    public static final double velocityLineScaler = 50;
    public static final double maximumZoom = 5000;
    public static final double zoomSkew = 6;

    private double centreDim1;
    private double centreDim2;
    private SolarBody centralBody;

    private int dimension1; // The Horizontal draw direction
    private int dimension2; // The Vertical draw direction
    private int dimension3; // The direction of depth into the Display

    // Values used to draw a drag line when moving planets.
    private boolean isDragLine = false;
    private int startDragX = 0;
    private int startDragY = 0;
    private int endDragX = 0;
    private int endDragY = 0;

    /**
     * @param b The Backend used to drive the display.
     * @param s Half the size of the Solar System in metres.
     * @param d1 The first dimension to draw in.
     * @param d2 The second dimension to draw in.
     */
    public Display(Backend b, double s, int d1, int d2){

        backend = b;
        setSolarSystem(backend.getSolarSystem());
        halfSize = s;
        halfDisplay = halfSize;
        centralBody = solar.getStar();
        dimension1 = d1;
        dimension2 = d2;

        // Work out the 3rd dimension from the 2 given dimensions.
        if(dimension1 == 1 && dimension2 == 2) dimension3 = 3;

        else if(dimension1 == 1 && dimension2 == 3) dimension3 = 2;

        else dimension3 = 1;


        centreDim1 = halfSize + solar.getStar().getPosition().getDim(dimension1);
        centreDim2 = halfSize + solar.getStar().getPosition().getDim(dimension2);
        sizeRatio = (double)halfWindow/s;
        setBackground(Color.black);
        setPreferredSize(new Dimension(halfWindow * 2, halfWindow * 2));
        setSolarBodyDrawWidths(1/solarBodySizeScaler);
```

```
        velocityLines = false;
        solarBodyNames = false;
        isAlive = true;
    }

    /**
     * Continuously run and repaint every time the Thread wakes up
     */
    public void run(){

        try{
            // Keep looping until told to stop
            while(isAlive){

                this.repaint();
                Thread.sleep(100); // Pause for 100 ms between each draw
            }
        }
        catch(InterruptedException e){

            System.err.println(e.getMessage());
            System.exit(-1);
        }
    }

    /**
     * Destroys this Thread by allowing the program to drop out of the while loop
     * in the run() method and the Thread terminates naturally.
     */
    public void destroy(){

        isAlive = false;
    }

    /**
     * Draw each current SolarBody in its current location.
     * Should only ever be called by repaint().
     *
     * @param g The current Graphics context to draw in.
     */
    public void paintComponent(Graphics g){

        // Called to remove all previous drawings.
        super.paintComponent(g);

        // Get a snapshot of the Backends current state.
        SolarSystem solar = backend.deliverCurrentData();

        // Find the position that is to be the centre of the display
        //SolarBody body = solar.findSolarBody(centralBody);
        if(centralBody != null){

            centreDim1 = halfDisplay - centralBody.getPosition().getDim(dimension1);
            centreDim2 = halfDisplay - centralBody.getPosition().getDim(dimension2);
        }

        // Get all SolarBodys into a single array ready to draw
        SolarBody[] bodys;

        if(drawSatellites){
            // Only draw the satellites if zoomed in enough to distinguish from
Planets.
            bodys = new SolarBody[solar.getNumberOfSolarBodys()];
            SolarBody[] temp = solar.getSatellites();
            System.arraycopy(temp, 0, bodys, bodys.length - temp.length, temp.length);
        }
        else{

            bodys = new SolarBody[1 + solar.getNumberOfPlanets()];
        }

        bodys[0] = solar.getStar();
        SolarBody[] temp = solar.getPlanets();
        System.arraycopy(temp, 0, bodys, 1, temp.length);

        // Draw furthest SolarBodies first
```

```
        orderByDistance(bodys);

        for(int i = 0; i < bodys.length ; i++){
            // Draw each SolarBody.
            drawBody(bodys[i], g);
        }

        // Draw the current time elapsed.
        g.setColor(Color.white);
        g.drawString(backend.getTime(), 10, 20);
    }

    /*
     * Draws A given SolarBody according to its Position and size.
     *
     * @param body The SolarBody to be drawn.
     * @param g The current Graphics context being used.
     */
    private void drawBody(SolarBody body, Graphics g){

        Triple pos = body.getPosition();

        int radius = (int) body.getDrawWidth()/2;

        int positionX = (int)((centreDim1 + pos.getDim(dimension1)) * sizeRatio);
        int positionY = (int)((centreDim2 + pos.getDim(dimension2)) * sizeRatio);

        // Only draw the Solar Body if it falls within the Display.
        if((positionX + radius > 0) && (positionY + radius > 0) && (positionX - radius
< getWidth()) && (positionY - radius < getHeight())){

            if(velocityLines) drawBodyVelocityLine(body, g);

            g.setColor(body.getColor());

            g.fillOval(positionX - radius, positionY - radius, body.getDrawWidth(),
body.getDrawWidth());
        }

        // Draw the Solar Body names.
        if(solarBodyNames) g.drawString(body.getName(), positionX + radius + 2,
positionY);

        // Draw a line corresponding to any current mousedrag.
        if(isDragLine){

            g.setColor(Color.orange);
            g.drawLine(startDragX, startDragY, endDragX, endDragY);
        }
    }

    /*
     * Draws a line representing Velocity for the given SolarBody.
     *
     * @param body The SolarBody that needs a Velocity Line drawn for.
     * @param g The current Graphics context being used.
     */
    private void drawBodyVelocityLine(SolarBody body, Graphics g){

        Triple position = body.getPosition();

        Triple velocity = body.getVelocity();

        int x1 = (int) ((centreDim1 + position.getDim(dimension1)) * sizeRatio);
        int y1 = (int) ((centreDim2 + position.getDim(dimension2)) * sizeRatio);
        int x2 = x1 + (int) (velocity.getDim(dimension1) * velocityLineScaler *
sizeRatio);
        int y2 = y1 + (int) (velocity.getDim(dimension2) * velocityLineScaler *
sizeRatio);
        g.setColor(Color.red);
        g.drawLine(x1, y1, x2, y2);
    }

    /*
     * Tells each SolarBody in the system how many pixels wide it
     * should be drawn on the screen.
     *
```

```
     * @param scalingFactor The amount to scale the SolarBody sizes by.
     */
    private void setSolarBodyDrawWidths(double scalingFactor){

        solar.getStar().setDrawWidth(scalingFactor);

        Planet[] planets = solar.getPlanets();

        for(int i = 0; i < planets.length; i++){

            planets[i].setDrawWidth(scalingFactor);
        }

        Satellite[] satellites = solar.getSatellites();

        for(int i = 0; i < satellites.length; i++){

            satellites[i].setDrawWidth(scalingFactor);
        }
    }

    /*
     * Re-order the array of SolarBodys in order of their depth into
     * the Display.  First element will be the furthest from the Camera.
     *
     * @param bodys The Array of SolarBodies being drawn.
     */
    private void orderByDistance(SolarBody[] bodys){

        for(int i=0; i<bodys.length-1; i++){

            int max = i;
            for(int j=i+1; j<bodys.length; j++){

                if(bodys[j].getPosition().getDim(dimension3) <
bodys[max].getPosition().getDim(dimension3)){

                    max = j;
                }
                SolarBody temp = bodys[i];
                bodys[i] = bodys[max];
                bodys[max] = temp;
            }
        }
    }

    /**
     * Toggle whether Velocity Lines are shown
     */
    public void toggleVelocityLines(){

        velocityLines = ! velocityLines;
    }

    /**
     * Toggle whether SolarBody Names are shown
     */
    public void toggleSolarBodyNames(){

        solarBodyNames = ! solarBodyNames;
    }

    /**
     * Set the zoom level of the Display.
     * 1.0 is Standard zoom.
     * maximumZoom is Maximum.
     *
     * @param scale The level to zoom in by.
     * @exception SolarSystemException If the amount specified by scale exceeds the
minimum of maximum allowed Zoom
     */
    public void alterZoom(double scale) throws SolarSystemException{

        scale = scale/Math.pow(maximumZoom, zoomSkew -1) + 1;

        if(scale < 1.0) throw new SolarSystemException
            ("Zoom level too low; must be between 1.0 and " + maximumZoom + ".");
```

```
        if(scale > maximumZoom + 1) throw new SolarSystemException
            ("Zoom level too high; must be between 1.0 and " + maximumZoom + ".");


        // Reset the Ratios between displayable Size and Window size
        halfDisplay = halfSize/scale;
        sizeRatio = (double) halfWindow / halfDisplay;
        setSolarBodyDrawWidths(scale/solarBodySizeScaler);

        // Only draw satellites if the current zoom is at least 1/10th of the max zoom.
        drawSatellites = (scale > 0.1 * maximumZoom);
    }

    /**
     * Sets the draw angle.
     *
     * @param d1 The Horizontal draw direction.
     * @param d2 The Vertical draw direction.
     */
    public void alterAngle(int d1, int d2){

        dimension1 = d1;
        dimension2 = d2;

        // Work out the third dimension.
        if(dimension1 == 1 && dimension2 == 2) dimension3 = 3;

        else if(dimension1 == 1 && dimension2 == 3) dimension3 = 2;

        else dimension3 = 1;
    }

    /**
     * Find out what the current draw angle is.
     *
     * @return A Dimension whose first element corresponds to the Horizontal Value and
whose second element corresponds to the Vertical Value.
     */
    public Dimension getAngle(){

        return new Dimension(dimension1, dimension2);
    }

    /**
     * Sets the Display to centre on the given SolarBody.
     *
     * @param body The SolarBody to be centred on.
     */
    public void setCentralObject(SolarBody body){

        centralBody = body;
    }

    /**
     * Finds any SolarBody that is within a few pixels of position (x, y).
     * Will give priority to Stars, then Planets, then Satellites.
     *
     * @param x The x-coord of the position to search for.
     * @param y The y-coord of the position to search for.
     * @return A SolarBody that is close to position (x, y) or null if none are close.
     */
    public SolarBody findSolarBody(int x, int y){

        SolarSystem system =  backend.getSolarSystem();

        // Check if the Star is close to (x, y)
        SynchronizedStar star = system.getStar();

        if(isSpecifiedBody(star, x, y)) return star;


        // check if any Planet is close to (x,y)
        SynchronizedPlanet[] planets = system.getPlanets();

        for(int i=0; i<planets.length; i++)
            if(isSpecifiedBody(planets[i], x, y)) return planets[i];
```

```
        // check if any Satellite is close to (x,y)
        SynchronizedSatellite[] satellites = system.getSatellites();

        for(int i=0; i<satellites.length; i++)
            if(isSpecifiedBody(satellites[i], x, y)) return satellites[i];


        // gets here if no SolarBody is close enough to (x,y)
        return null;
    }

    /*
     * Check if SolarBody body is close to position (x,y)
     *
     * @param body The SolarBody to be checked for.
     * @param x The x-coord of the position to search for.
     * @param y The y-coord of the position to search for.
     * @return true if the body is close to position (x, y).  false otherwise.
     */
    private boolean isSpecifiedBody(SolarBody body, int x, int y){

        int bodyX = (int) ((centreDim1 + body.getPosition().getDim(dimension1)) *
sizeRatio);
        int bodyY = (int) ((centreDim2 + body.getPosition().getDim(dimension2)) *
sizeRatio);

        return ((Math.abs(bodyX-x) < 10) && (Math.abs(bodyY-y) < 10));
    }

    /**
     * Sets the position to draw the DragLine of the Planet dragger.
     *
     * @param x1 The Horizontal start position
     * @param y1 The Vertical start position
     * @param x2 The Horizontal end position
     * @param y2 The Vertical end position
     */
    public void drawDragLine(int x1, int y1, int x2, int y2){

        startDragX = x1;
        startDragY = y1;
        endDragX = x2;
        endDragY = y2;
    }

    /**
     * Sets whether the DragLine should be drawn
     *
     * @param draw true if the line should be drawn. false otherwise.
     */
    public void setDragLine(boolean draw){

        isDragLine = draw;
    }

    /**
     * Find the ratio between the Window Size and Solar System size.
     *
     * @return The number of times bigger the Solar System is compared to the window,
     */
    public double getSizeRatio(){

        return sizeRatio;
    }

    /**
     * Set the SolarSystem being drawn to be system
     *
     * @param system The SolarSystem to be drawn.
     */
    public void setSolarSystem(SolarSystem system){

        solar = system;
    }
}
```

Frontend.java

```
/**
 * The Frontend is the GUI for the Solar System simulation.
 * It contains controls to let the user manipulate the system
 * as well as 2 windows.  The first window displays all the
 * current data regarding the Solar Bodies such as position,
 * velocity, mass etc.  The second window shows a graphical
 * view of the Solar System as it currently stands.
 *
 * @author Adam Haigh
 * @version 1.1  9/1/03
 */

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.text.*;

public class Frontend extends JApplet implements ActionListener{
    private boolean paused;
    private boolean executing;
    private Display display;
    private Thread displayThread;
    private DataWindow dataWindow;
    private SolarEditor editor;
    private Backend backend;
    private static final double halfSolarSystemSize = 10000000000000.0; // Half the
size of the solar system in metres

    private int mousePressedX;
    private int mousePressedY;

    JToolBar mainToolBar;
    JToolBar displayToolBar;
    JPanel viewingArea;
    JScrollPane dataWindowScroll;

    JButton pause;
    JButton displayData;
    JButton toggleEditor;
    JButton velocityLines;
    JButton solarBodyNames;

    JButton toggleView;

    JLabel zoomLevelLabel;
    JSlider zoomLevel;

    JLabel stepSizeLabel;
    JSlider stepSize;

    JComboBox centralBodyCombo;
    JLabel centralBodyLabel;

    /**
     * Set up the Applet including initialising the Backend.
     */
    public void init(){

        paused = false;
        executing = false;
        backend = new Backend(getParameter("SolarSystem"));
        backend.setPriority(Thread.MIN_PRIORITY);

        display = new Display(backend, halfSolarSystemSize, 1, 2);
        displayThread = new Thread(display);
        displayThread.setPriority(Thread.MAX_PRIORITY);
        displayThread.start();

        dataWindow = new DataWindow(backend);
        dataWindow.displayCurrentData();
        dataWindowScroll = new JScrollPane(dataWindow,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED, JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
```

```
        editor = new SolarEditor(backend, this, "Solar System Editor");

        mainToolBar = new JToolBar(JToolBar.HORIZONTAL);
        displayToolBar = new JToolBar(JToolBar.HORIZONTAL);

        viewingArea = new JPanel();
        viewingArea.setLayout(new GridLayout());

        pause = new JButton("Start");
        displayData = new JButton("Show Data");
        toggleEditor = new JButton("Edit Solar Bodies");
        velocityLines = new JButton("Toggle Velocity Lines");
        solarBodyNames = new JButton("Toggle Solar Body Names");

        toggleView = new JButton("Change View");

        zoomLevelLabel = new JLabel("Set Zoom Level:");
        zoomLevel = new JSlider(100, (int)display.maximumZoom * 100, 100);

        stepSizeLabel = new JLabel("Set Simulation Speed:");
        stepSize = new JSlider(1, 36000, 1);

        // Initialise Central Body to be the Star
        centralBodyLabel = new JLabel("Set Central Body:");
        centralBodyCombo = new JComboBox(backend.getSolarSystem().listSolarBodys());

        // Set the ToolTipTexts for all the JButtons, JLabels and JSliders
        pause.setToolTipText("Pause or Resume the Simulation");
        zoomLevel.setToolTipText("Increase zoom by dragging bar to the right");
        zoomLevelLabel.setToolTipText("Increase zoom by dragging bar to the right");
        stepSize.setToolTipText("Increase Simulation speed by dragging bar to the
right");
        stepSizeLabel.setToolTipText("Increase Simulation speed by dragging bar to the
right");
        centralBodyCombo.setToolTipText("Select the Solar Body to centre on");
        displayData.setToolTipText("Display the current Solar System info.");
        toggleEditor.setToolTipText("Bring up an Editor to create and alter Solar
Bodys");
        velocityLines.setToolTipText("Toggle whether Velocity lines are shown");
        solarBodyNames.setToolTipText("Toggle whether Solar Body names are shown");
        toggleView.setToolTipText("Change the view between 3 different angles");

        pause.addActionListener(this);
        displayData.addActionListener(this);
        toggleEditor.addActionListener(this);
        velocityLines.addActionListener(this);
        solarBodyNames.addActionListener(this);
        toggleView.addActionListener(this);

        /**
         * Listener for JSlider setting the zoom level of the display.
         */
        zoomLevel.addChangeListener(new ChangeListener(){

                /**
                 * set the zoom level to be the current value of the slider
                 */
                public void stateChanged(ChangeEvent e){

                    try{
                        display.alterZoom(Math.pow((double)zoomLevel.getValue() / 100,
display.zoomSkew));
                    }
                    catch(SolarSystemException ex){
                        System.out.println(ex.getMessage());
                    }
                }
            });

        /**
         *  Listener for JSlider setting the size of each time step in the backend
         */
        stepSize.addChangeListener(new ChangeListener(){

                /**
                 * set the size of each time step to be the current value of slider
```

```
                 */
                public void stateChanged(ChangeEvent e){

                    backend.setTimeStep(stepSize.getValue());
                }
            });

        /**
         * Listener for Combobox selecting items
         */
        centralBodyCombo.addItemListener(new ItemListener(){

                /**
                 * Set the display to centre on the object selected in the combo box
                 */
                public void itemStateChanged(ItemEvent e){
                    if(e.getSource().equals(centralBodyCombo)){
                        SolarBody body = backend.getSolarSystem().findSolarBody((String)
centralBodyCombo.getSelectedItem());
                        display.setCentralObject(body);
                    }
                }
            });


        /**
         * Listener for Mouse Events on the display.
         */
        display.addMouseListener(new MouseAdapter(){

                SolarBody body;

                /**
                 * On a Mouse Press, pause the Simulation and try to locate the
                 * SolarBody being clicked on.
                 */
                public void mousePressed(MouseEvent e){

                    if(! paused) pause();

                    body = display.findSolarBody(e.getX(), e.getY());

                    mousePressedX = e.getX();
                    mousePressedY = e.getY();

                    if(body != null) display.setDragLine(true);
                }

                /**
                 * On a mouse release set the Solar Body that was clicked on
                 * to have a new velocity determined by the length and direction of the
line.
                 */
                public void mouseReleased(MouseEvent e){

                    if(body != null){

                        double changeX = (e.getX() -
mousePressedX)/display.getSizeRatio();
                        double changeY = (e.getY() -
mousePressedY)/display.getSizeRatio();

                        body = backend.getSolarSystem().findSolarBody(body.getName());
                        Triple t = body.getVelocity();
                        Dimension d = display.getAngle();
                        t.changeDim((int) d.getWidth(),
changeX/display.velocityLineScaler);
                        t.changeDim((int) d.getHeight(),
changeY/display.velocityLineScaler);
                    }
                    display.setDragLine(false);
                    display.drawDragLine(0,0,0,0);
                    resume();
                }
            });

        /**
```

```
         * Listener for MouseMotionEvents on the Display
         */
        display.addMouseMotionListener(new MouseMotionAdapter(){

                /**
                 * Draw the Drag line on the display.
                 */
                public void mouseDragged(MouseEvent e){

                    display.drawDragLine(mousePressedX, mousePressedY, e.getX(),
e.getY());
                }
            });

        mainToolBar.add(pause);
        mainToolBar.add(new JToolBar.Separator());
        mainToolBar.add(stepSizeLabel);
        mainToolBar.add(stepSize);
        mainToolBar.add(toggleEditor);
        mainToolBar.add(displayData);

        displayToolBar.add(velocityLines);
        displayToolBar.add(solarBodyNames);
        displayToolBar.add(new JToolBar.Separator());
        displayToolBar.add(zoomLevelLabel);
        displayToolBar.add(zoomLevel);
        displayToolBar.add(centralBodyLabel);
        displayToolBar.add(centralBodyCombo);
        displayToolBar.add(toggleView);

        viewingArea.add(display);
        viewingArea.add(dataWindowScroll);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(mainToolBar, BorderLayout.NORTH);
        getContentPane().add(displayToolBar, BorderLayout.SOUTH);
        getContentPane().add(viewingArea, BorderLayout.CENTER);
        display.setBorder(LineBorder.createGrayLineBorder());
    }

    /**
     * Find which button was pressed and determine what action
     * should be taken
     */
    public void actionPerformed(ActionEvent e){

        JButton button = (JButton) e.getSource();

        // Pause Button
        if(button.equals(pause)){

            if(! executing) go();

            else{

                if(paused) resume();

                else pause();
            }
        }
        // DisplayData Button
        else if(button.equals(displayData)){

            dataWindow.displayCurrentData();
        }
        // VelocityLine Button
        else if(button.equals(velocityLines)){

            display.toggleVelocityLines();
        }
        // SolarBodyNames Button
        else if(button.equals(solarBodyNames)){

            display.toggleSolarBodyNames();
        }
        // SolarEditor Button
        else if(button.equals(toggleEditor)){
```

```
            editor.show();
        }
        //Toggle View Button
        else if(button.equals(toggleView)){

            Dimension currentAngle = display.getAngle();

            // Get the next of the 3 possible angles
            if(currentAngle.equals(new Dimension(1,2)))
                display.alterAngle(1,3);
            else if(currentAngle.equals(new Dimension(1,3)))
                display.alterAngle(2,3);
            else if(currentAngle.equals(new Dimension(2,3)))
                display.alterAngle(1,2);
        }
    }

    /*
     * Pause the Backend
     */
    private void pause(){

        paused = true;
        pause.setText("Resume");

        backend.pause();
    }

    /*
     * Tell the Backend to resume
     */
    private void resume(){

        paused = false;

        backend.unPause();

        pause.setText("Pause");
    }

    /*
     * Start execution
     */
    private void go(){
        executing = true;
        pause.setText("Pause");
        backend.start();
    }
}
```

OrbitingSolarBody.java

```
/**
 * The OrbitingSolarBody interface is an extension of the SolarBody interface where
 * the Object implementing this must also implement the getCentreOfOrbit() method
 * and so have some SolarBody that it is Orbiting.
 *
 * @author Adam Haigh
 * @version 1.1  23/2/03
 */

public interface OrbitingSolarBody extends SolarBody{

    public SolarBody getCentreOfOrbit();
}
```

Planet.java

```
/**
 * The Planet class implements the SolarBody interface to provide methods
 * that allow any given Planet to alter itself in any way.  The original system
 * will instantiate the Planet class 9 times; once for each of the 9 major
 * Planets of the Solar System.
 *
 * @author Adam Haigh
 * @version 1.2  26/1/03
 */

import java.awt.Color;

public class Planet implements OrbitingSolarBody, Cloneable{

    private String name;
    private double mass;  // Measured in Kg
    private double size;  // Diameter measured in metres
    private double drawScaler; // current scaler used to draw the planet.
    private Star centreOfOrbit;
    private Triple position;
    private Triple velocity;
    private Triple acceleration;
    private int drawWidth;
    private Color color;

    /**
     * Create a new Planet with a full specification.
     *
     * @param n The Name of the Planet
     * @param m The Mass of the Planet
     * @param si The Diameter of the Planet
     * @param p The Position of the Planet. [0, 0, 0] is the centre of the Solar
System
     * @param v The Velocity of the Planet. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Planet. [0, 0, 0] Shows no change in Velocity.
     * @param c The Color of the Planet.
     * @param s The Star that the Planet is Orbitting.
     */
    public Planet(String n, double m, double si, Triple p, Triple v, Triple a, Color
c, Star s){
        name = n;
        mass = m;
        size = si;
        centreOfOrbit = s;
        position = p;
        velocity = v;
        acceleration = a;
        color = c;

        drawWidth = 1;
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public void changeMass(double newMass){

        mass = newMass;
    }

    /**
     * Sets the diameter to be newDiameter and resets the draw Width.
     *
     * @param newDiameter The new Diameter of the Planet.
     */
    public void changeDiameter(double newDiameter){

        size = newDiameter;
        setDrawWidth(drawScaler);
    }
```

```
/**
 * Sets a new Position for the Planet.
 *
 * @param t The new position of the Planet.
 */
public void changePosition(Triple t){

    position = t;
}

/**
 * Sets a new Velocity for the Planet.
 *
 * @param t The new Velocity of the Planet.
 */
public void changeVelocity(Triple t){

    velocity = t;
}

/**
 * Sets a new Acceleration for the Planet.
 *
 * @param t The new Acceleratio of the Planet.
 */
public void changeAcceleration(Triple t){

    acceleration = t;
}

/**
 * Sets the size of the Planet to be drawn in pixels.
 *
 * @param scaler The new draw width of the planet in pixels.
 */
public void setDrawWidth(double scaler){

    drawScaler = scaler;
    int w =  (int) Math.pow(size, scaler);

    if(w<4)
        drawWidth = 4;
    else
        drawWidth = w;
}

/**
 * Specifies the Color of the Planet with RGB values in range 0-255
 *
 * @param r The value for red.
 * @param g The value for green.
 * @param b The value for blue.
 */
public void setColor(int r, int g, int b){

    color = new Color(r, g, b);
}

/**
 * Gets the current Position of the Planet
 *
 * @return The current Position of the Planet as a Triple.
 */
public Triple getPosition(){

    return position;
}

/**
 * Gets the current Velocity of the Planet
 *
 * @return The current Velocity of the Planet as a Triple.
 */
public Triple getVelocity(){

    return velocity;
}
```

```
    /**
     * Gets the current Acceleration of the Planet
     *
     * @return The current Acceleration of the Planet as a Triple.
     */
    public Triple getAcceleration(){

        return acceleration;
    }

    /**
     * Gets the Mass of the Planet
     *
     * @return The mass of the Planet as a double.
     */
    public double getMass(){

        return mass;
    }

    /**
     * Gets the Star that this Planet is orbiting.
     *
     * @return The Star that the Planet is orbiting as a general SolarBody.
     */
    public SolarBody getCentreOfOrbit(){

        return centreOfOrbit;
    }

    /**
     * Gets the diameter of the Planet in metres.
     *
     * @return The diameter of the planet as a double.
     */
    public double getSize(){

        return size;
    }

    /**
     * Gets the Name of the Planet.
     *
     * @return The Name of the Planet as a String.
     */
    public String getName(){

        return name;
    }

    /**
     * Gets the size of the Planet to be drawn in pixels
     *
     * @return The number of pixels wide the Planet should be drawn.
     */
    public int getDrawWidth(){

        return drawWidth;
    }

    /**
     * Gets the Color of the Planet
     *
     * @return The Color of the Planet.
     */
    public Color getColor(){

        return color;
    }

    /**
     * Gets a String representation of the Planets Position,
     * Velocity and Acceleration.
     *
     * @return The String representation of the Planets Position, Velocity and
Acceleration.
```

```
     */
    public String toString(){

        return name + position.toString() + velocity.toString() +
acceleration.toString();
    }

    /**
     * Performs a deep clone of the Planet Object.
     *
     * @return An exact copy of the Planet.
     */
    public Object clone(){

        Planet p  = null;

        try{
            p = (Planet) super.clone();
        }
        catch(CloneNotSupportedException e){
            System.err.println("Planet can't clone");
        }

        p.position = (Triple) position.clone();
        p.velocity = (Triple) velocity.clone();
        p.acceleration = (Triple) acceleration.clone();

        return p;
    }

    /**
     * Checks equality by checking that the Names match.
     *
     * @param body The SolarBody being compared to this one.
     * @return true if The name of body matches that of this Planet.
     */
    public boolean equals(SolarBody body){

        return name.equals(body.getName());
    }
}
```

Satellite.java

```
/**
 * The Satellite class implements the SolarBody interface to provide methods
 * that allow any given Satellite to alter itself in any way.  The original system
 * will instantiate the satellite class once; for the moon Orbiting the Earth
 *
 * @author Adam Haigh
 * @version 1.2  26/1/03
 */
import java.awt.Color;

public class Satellite implements OrbitingSolarBody, Cloneable{
    private String name;
    private double mass;  // Measured in Kg
    private double size;  // Diameter measured in metres
    private double drawScaler; // The amount to re-scale the Satellite by when drawing
    private Planet centreOfOrbit;
    private Triple position;
    private Triple velocity;
    private Triple acceleration;
    private int drawWidth;
    private Color color;

    /**
     * Create a new Satellite with a full specification.
     *
     * @param n The Name of the Satellite
     * @param m The Mass of the Satellite
     * @param si The Diameter of the Satellite
     * @param p The Position of the Satellite. [0, 0, 0] is the centre of the Solar
System
     * @param v The Velocity of the Satellite. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Satellite. [0, 0, 0] Shows no change in
Velocity.
     * @param c The Color of the Satellite.
     * @param pl The Planet that the Satellite is Orbitting.
     */
    public Satellite(String n, double m, double si, Triple p, Triple v, Triple a,
Color c, Planet pl){
        name = n;
        mass = m;
        size = si;
        centreOfOrbit = pl;
        position = p;
        velocity = v;
        acceleration = a;
        color = c;

        drawWidth = 1;
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public void changeMass(double newMass){

        mass = newMass;
    }

    /**
     * Sets the diameter to be newDiameter and resets the draw Width.
     *
     * @param newDiameter The new Diameter of the Planet.
     */
    public void changeDiameter(double newDiameter){

        size = newDiameter;
        setDrawWidth(drawScaler);
    }

    /**
     * Sets a new Position for the Satellite.
```

```
     *
     * @param t The new position of the Satellite.
     */
    public void changePosition(Triple t){

        position = t;
    }

    /**
     * Sets a new Velocity for the Satellite.
     *
     * @param t The new Velocity of the Satelite.
     */
    public void changeVelocity(Triple t){

        velocity = t;
    }

    /**
     * Sets a new Acceleration for the Satelite.
     *
     * @param t The new Acceleratio of the Satellite.
     */
    public void changeAcceleration(Triple t){

        acceleration = t;
    }

    /**
     * Sets the size of the Satellite to be drawn in pixels.
     *
     * @param scaler The new draw width of the Satellite in pixels.
     */
    public void setDrawWidth(double scaler){

        drawScaler = scaler;
        int w =  (int) Math.pow(size, scaler);

        if(w<4) drawWidth = 4;

        else drawWidth = w;
    }

    /**
     * Specifies the Color of the Satellite with RGB values in range 0-255
     *
     * @param r The value for red.
     * @param g The value for green.
     * @param b The value for blue.
     */
    public void setColor(int r, int g, int b){

        color = new Color(r, g, b);
    }

    /**
     * Gets the current Position of the Satellite
     *
     * @return The current Position of the Satellite as a Triple.
     */
    public Triple getPosition(){

        return position;
    }

    /**
     * Gets the current Velocity of the Satellite
     *
     * @return The current Velocity of the Satellite as a Triple.
     */
    public Triple getVelocity(){

        return velocity;
    }

    /**
     * Gets the current Acceleration of the Satellite
```

```
     *
     * @return The current Acceleration of the Satellite as a Triple.
     */
    public Triple getAcceleration(){

        return acceleration;
    }

    /**
     * Gets the Mass of the Satellite
     *
     * @return The mass of the Satellite as a double.
     */
    public double getMass(){

        return mass;
    }

       /**
     * Gets the Planet that this Satellite is orbiting.
     *
     * @return The Planet that the Satellite is orbiting as a general SolarBody.
     */
    public SolarBody getCentreOfOrbit(){

        return centreOfOrbit;
    }

    /**
     * Gets the diameter of the Satellite in metres.
     *
     * @return The diameter of the Satellite as a double.
     */
    public double getSize(){

        return size;
    }

    /**
     * Gets the Name of the Satellite.
     *
     * @return The Name of the Satellite as a String.
     */
    public String getName(){

        return name;
    }

    /**
     * Gets the size of the Satellite to be drawn in pixels
     *
     * @return The number of pixels wide the Satellite should be drawn.
     */
    public int getDrawWidth(){

        return drawWidth;
    }

    /**
     * Gets the Color of the Satellite
     *
     * @return The Color of the Satellite.
     */
    public Color getColor(){

        return color;
    }

    /**
     * Gets a String representation of the Satelites Position,
     * Velocity and Acceleration.
     *
     * @return The String representation of the satellites Position, Velocity and
Acceleration.
     */
    public String toString(){
```

```
        return name + position.toString() + velocity.toString() +
acceleration.toString();
    }

    /**
     * Performs a deep clone of the Satellite Object.
     *
     * @return An exact copy of the Satellite.
     */
    public Object clone(){

        Satellite s  = null;

        try{
            s = (Satellite) super.clone();
        }
        catch(CloneNotSupportedException e){
            System.err.println("Satellite can't clone");
        }
        s.position = (Triple) position.clone();
        s.velocity = (Triple) velocity.clone();
        s.acceleration = (Triple) acceleration.clone();

        return s;
    }

    /**
     * Checks equality by checking that the Names match.
     *
     * @param body The SolarBody being compared to this one.
     * @return true if The name of body matches that of this Satellite.
     */
    public boolean equals(SolarBody body){

        return name.equals(body.getName());
    }
}
```

SolarBody.java

```
/**
 * The Solar Body interface is used as a common template for any moving
 * object within the simulation, whether it be a Star, Planet or Satellite.
 * Each of the 'change' methods returns true if the change was successfully
 * completed.
 *
 * @author Adam Haigh
 * @version 1.2  26/1/03
 */

import java.awt.Color;

public interface SolarBody{

    /**
     * Sets the mass to be newMass.
     */
    public void changeMass(double newMass);

    /**
     * Sets the diameter to be newDiameter and resets the draw Width.
     */
    public void changeDiameter(double newDiameter);

    /**
     * Sets the Position to be t.
     */
    public void changePosition(Triple t);

    /**
     * Sets the Velocity to be t.
     */
    public void changeVelocity(Triple t);

    /**
     * Sets the Acceleration to be t.
     */
    public void changeAcceleration(Triple t);

    /**
     * Sets the Color of the SolarBody to c
     */
    public void setColor(int r, int g, int b);

    /**
     * Sets the size of the Planet to be drawn in pixels
     */
    public void setDrawWidth(double scaler);

    /**
     * Gets the Name of this SolarBody
     */
    public String getName();

    /**
     * Gets the current Position of the SolarBody
     */
    public Triple getPosition();

    /**
     * Gets the current Velocity of the SolarBody
     */
    public Triple getVelocity();

    /**
     * Gets the current Acceleration of the SolarBody
     */
    public Triple getAcceleration();

    /**
     * Gets the Mass of the SolarBody
     */
    public double getMass();
```

```
    /**
     * Gets the diameter of the SolarBody in metres.
     */
    public double getSize();

    /**
     * Gets the size of the Planet to be drawn in pixels
     */
    public int getDrawWidth();

    /**
     * Returns the current Color of the SolarBody
     */
    public Color getColor();

    /**
     * Returns a String representation of the SolarBodys Position,
     * Velocity and Acceleration.
     */
    public String toString();

    /**
     * Checks equality by checking that the Names match.
     */
    public boolean equals(SolarBody body);
}
```

SolarEditor.java

```
/**
 * The SolarEditor Class is used both to create new SolarBodys for
 * the SolarSystem Applet and to edit existing ones as the user sees fit.
 * Changes through this tool are not permanent.
 * Closing the Applet will permanently lose all changes made through this tool.
 * Permanent changes and additions must be made directly to the
 * SolarSystem text file.
 *
 * @author Adam Haigh
 * @version 1.1 7/3/2003
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;

public class SolarEditor extends JFrame implements ActionListener{

    private Backend backend;
    private Frontend frontend;
    private SolarSystem solar;
    private SolarBody current;

    JPanel bodySelectorPanel;
    JLabel solarBodyLabel;
    JComboBox solarBodySelector;
    JButton edit;
    JButton makeNew;
    JButton hide;

    private boolean editable; // Whether the JFrame is in edit/create mode or not
    private boolean existing; // Whether the SolarBody being edited is a new or
existing one

    JPanel centre;
    JLabel nameLabel;
    JLabel bodyTypeLabel;
    JLabel orbitCentreLabel;
    JLabel massLabel;
    JLabel diameterLabel;
    JLabel positionXLabel;
    JLabel positionYLabel;
    JLabel positionZLabel;
    JLabel velocityXLabel;
    JLabel velocityYLabel;
    JLabel velocityZLabel;
    JLabel red;
    JLabel green;
    JLabel blue;
    JLabel messageLabel;

    JTextField nameText;
    JComboBox bodyTypeCombo;
    JComboBox orbitCentreCombo;
    JTextField massText;
    JTextField diameterText;
    JTextField positionXText;
    JTextField positionYText;
    JTextField positionZText;
    JTextField velocityXText;
    JTextField velocityYText;
    JTextField velocityZText;
    JSlider redSlider;
    JSlider greenSlider;
    JSlider blueSlider;
    JTextField messageText;

    // Strength of rgb values of current SolarBody
    private int redness;
    private int greenness;
    private int blueness;
```

```
    JPanel commitPanel;
    JButton commit;
    JButton reject;
    JLabel currentColor;

    /**
     * Create a new SolarEditor that acts on Backend b with the String title as the
Title.
     *
     * @param b The Backend which will be modified by the Editor.
     * @param f The Frontend that called the Editor.
     * @param title The Text that will appear in the Title of the Editor.
     */
    public SolarEditor(Backend b, Frontend f, String title){

        super(title);

        frontend = f;
        backend = b;
        solar = backend.getSolarSystem();
        current = solar.getStar();

        setSize(new Dimension(450,400));

        bodySelectorPanel = new JPanel();
        solarBodyLabel = new JLabel("Select Solar Body:");
        solarBodySelector = new JComboBox(solar.listSolarBodys());
        edit = new JButton("Edit");
        makeNew = new JButton("New");
        hide = new JButton("Hide This");

        bodySelectorPanel.add(solarBodySelector);
        bodySelectorPanel.add(edit);
        bodySelectorPanel.add(hide);

        centre = new JPanel();
        nameLabel = new JLabel("Name: ");
        bodyTypeLabel = new JLabel("Object Type: ");
        orbitCentreLabel = new JLabel("Centre of Orbit: ");
        massLabel = new JLabel("Mass: ");
        diameterLabel = new JLabel("Diameter: ");
        positionXLabel = new JLabel("X-Position: ");
        positionYLabel = new JLabel("Y-Position: ");
        positionZLabel = new JLabel("Z-Position: ");
        velocityXLabel = new JLabel("X-Velocity: ");
        velocityYLabel = new JLabel("Y-Velocity: ");
        velocityZLabel = new JLabel("Z-Velocity: ");
        red = new JLabel("Red Strength: ");
        green = new JLabel("Green Strength: ");
        blue = new JLabel("Blue Strength: ");
        messageLabel = new JLabel("Messages: ");

        Star star = solar.getStar();
        nameText = new JTextField(star.getName());
        String[] bodyTypes = new String[3];
        bodyTypes[0] = "Star";
        bodyTypes[1] = "Planet";
        bodyTypes[2] = "Satellite";
        bodyTypeCombo = new JComboBox(bodyTypes);
        bodyTypeCombo.setSelectedIndex(0);
        orbitCentreCombo = new JComboBox();//(solar.listSolarBodys());
        orbitCentreCombo.addItem("None");
        orbitCentreCombo.setSelectedIndex(0);
        massText = new JTextField(Double.toString(star.getMass()));
        diameterText = new JTextField(Double.toString(star.getSize()));
        positionXText = new JTextField(Double.toString(star.getPosition().getX()));
        positionYText = new JTextField(Double.toString(star.getPosition().getY()));
        positionZText = new JTextField(Double.toString(star.getPosition().getZ()));
        velocityXText = new JTextField(Double.toString(star.getVelocity().getX()));
        velocityYText = new JTextField(Double.toString(star.getVelocity().getY()));
        velocityZText = new JTextField(Double.toString(star.getVelocity().getZ()));
        Color color = star.getColor();
        redness = color.getRed();
        greenness = color.getGreen();
        blueness = color.getBlue();
        redSlider = new JSlider(0, 255, redness);
        greenSlider = new JSlider(0, 255, greenness);
```

```
        blueSlider = new JSlider(0, 255, blueness);
        messageText = new JTextField("None");

        commitPanel = new JPanel();
        commit = new JButton("Commit");
        reject = new JButton("Reject");
        currentColor = new JLabel("Solar Body Color");
        currentColor.setForeground(color);

        // Set the Tool Tips of all the JButtons, JLabels and JTextFields
        nameLabel.setToolTipText("Set the name of the Solar Body.");
        bodyTypeLabel.setToolTipText("Set whether the Solar Body is a Star, Planet or
Satellite.");
        orbitCentreLabel.setToolTipText("Set which Solar Body this Solar Body is
orbiting.");
        massLabel.setToolTipText("Set the mass of the Solar Body.");
        diameterLabel.setToolTipText("Set the diameter of the Solar Body.");
        positionXLabel.setToolTipText("Set the X-Component of the Position.");
        positionYLabel.setToolTipText("Set the Y-Component of the Position.");
        positionZLabel.setToolTipText("Set the Z-Component of the Position.");
        velocityXLabel.setToolTipText("Set the X-Component of the Velocity.");
        velocityYLabel.setToolTipText("Set the Y-Component of the Velocity.");
        velocityZLabel.setToolTipText("Set the Z-Component of the Velocity.");
        red.setToolTipText("Set the Red-Strength of the Color.");
        green.setToolTipText("Set the Green-Strength of the Color.");
        blue.setToolTipText("Set the Blue-Strength of the Color.");
        messageLabel.setToolTipText("Shows any error messages.");

        nameText.setToolTipText("Set the name of the Solar Body.");
        bodyTypeCombo.setToolTipText("Set whether the Solar Body is a Star, Planet or
Satellite.");
        orbitCentreCombo.setToolTipText("Set which Solar Body this Solar Body is
orbiting.");
        massText.setToolTipText("Set the mass of the Solar Body.");
        diameterText.setToolTipText("Set the diameter of the Solar Body.");
        positionXText.setToolTipText("Set the X-Component of the Position.");
        positionYText.setToolTipText("Set the Y-Component of the Position.");
        positionZText.setToolTipText("Set the Z-Component of the Position.");
        velocityXText.setToolTipText("Set the X-Component of the Velocity.");
        velocityYText.setToolTipText("Set the Y-Component of the Velocity.");
        velocityZText.setToolTipText("Set the Z-Component of the Velocity.");
        redSlider.setToolTipText("Set the Red-Strength of the Color.");
        greenSlider.setToolTipText("Set the Green-Strength of the Color.");
        blueSlider.setToolTipText("Set the Blue-Strength of the Color.");
        messageText.setToolTipText("Shows any error messages.");

        // Set up all the ActionListeners for the Buttons
        edit.addActionListener(this);
        makeNew.addActionListener(this);
        hide.addActionListener(this);
        commit.addActionListener(this);
        reject.addActionListener(this);

        /**
         * Listener for JSlider setting the red level of the Solar Body.
         */
        redSlider.addChangeListener(new ChangeListener(){

                /**
                 * Set the Red level to be the current value of the Slider
                 */
                public void stateChanged(ChangeEvent e){

                    redness = redSlider.getValue();
                    currentColor.setForeground(new Color(redness, greenness,
blueness));
                }
            });


        /**
         * Listener for JSlider setting the green level of the Solar Body.
         */
        greenSlider.addChangeListener(new ChangeListener(){

                /**
                 * Set the Green level to be the current value of the Slider
```

```
                 */
                public void stateChanged(ChangeEvent e){

                    greenness = greenSlider.getValue();
                    currentColor.setForeground(new Color(redness, greenness,
blueness));
                }
            });


        /**
         * Listener for JSlider setting the blue level of the Solar Body.
         */
        blueSlider.addChangeListener(new ChangeListener(){

                /**
                 * Set the Blue level to be the current value of the Slider
                 */
                public void stateChanged(ChangeEvent e){

                    blueness = blueSlider.getValue();
                    currentColor.setForeground(new Color(redness, greenness,
blueness));
                }
            });


        /**
         * Listener for Combobox selecting Solar Bodies
         */
        solarBodySelector.addItemListener(new ItemListener(){

                /**
                 *
                 */
                public void itemStateChanged(ItemEvent e){

                    current = solar.findSolarBody((String)
solarBodySelector.getSelectedItem());
                    // Set the contents of the orbitCentreCombo to include
                    // the appropriate SolarBodies.
                    if(current instanceof Star){

                        orbitCentreCombo.removeAllItems();
                        orbitCentreCombo.addItem("None");
                    }
                    else if(current instanceof Planet){

                        orbitCentreCombo.removeAllItems();
                        orbitCentreCombo.addItem(solar.getStar().getName());
                    }
                    else if(current instanceof Satellite){

                        orbitCentreCombo.removeAllItems();
                        String[] s = solar.listPlanets();
                        for(int i=0; i<s.length; i++){

                            orbitCentreCombo.addItem(s[i]);
                        }
                    }


                    restoreFields();
                }
            });

        /**
         * Listener for ComboBox selecting which type of Solar Body is being edited.
         */
        bodyTypeCombo.addItemListener(new ItemListener(){

                /**
                 *
                 */
                public void itemStateChanged(ItemEvent e){

                    if(bodyTypeCombo.getSelectedItem().equals("Star")){
```

```
                    orbitCentreCombo.removeAllItems();
                    orbitCentreCombo.addItem("None");
                }
                else if(bodyTypeCombo.getSelectedItem().equals("Planet")){

                    orbitCentreCombo.removeAllItems();
                    orbitCentreCombo.addItem(solar.getStar().getName());
                }
                else if(bodyTypeCombo.getSelectedItem().equals("Satellite")){

                    orbitCentreCombo.removeAllItems();
                    String[] s = solar.listPlanets();
                    for(int i=0; i<s.length; i++){

                        orbitCentreCombo.addItem(s[i]);
                    }
                }
            }
        });

    bodySelectorPanel.add(solarBodyLabel);
    bodySelectorPanel.add(solarBodySelector);
    bodySelectorPanel.add(edit);
    bodySelectorPanel.add(makeNew);
    bodySelectorPanel.add(hide);

    centre.setLayout(new GridLayout(15,2));
    centre.add(nameLabel);
    centre.add(nameText);
    centre.add(bodyTypeLabel);
    centre.add(bodyTypeCombo);
    centre.add(orbitCentreLabel);
    centre.add(orbitCentreCombo);
    centre.add(massLabel);
    centre.add(massText);
    centre.add(diameterLabel);
    centre.add(diameterText);
    centre.add(positionXLabel);
    centre.add(positionXText);
    centre.add(positionYLabel);
    centre.add(positionYText);
    centre.add(positionZLabel);
    centre.add(positionZText);
    centre.add(velocityXLabel);
    centre.add(velocityXText);
    centre.add(velocityYLabel);
    centre.add(velocityYText);
    centre.add(velocityZLabel);
    centre.add(velocityZText);
    centre.add(red);
    centre.add(redSlider);
    centre.add(green);
    centre.add(greenSlider);
    centre.add(blue);
    centre.add(blueSlider);
    centre.add(messageLabel);
    centre.add(messageText);

    commitPanel.add(commit);
    commitPanel.add(reject);
    commitPanel.add(currentColor);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(bodySelectorPanel, BorderLayout.NORTH);
    getContentPane().add(centre, BorderLayout.CENTER);
    getContentPane().add(commitPanel, BorderLayout.SOUTH);

    editable = true; // next line sets editable to not(editable)
    toggleEditable();
    nameText.setEnabled(false);
    diameterText.setEnabled(false);
    messageText.setEnabled(false);
}

/**
 * Find which button was pressed and determine what action
```

```
     * should be taken
     */
    public void actionPerformed(ActionEvent e){

        JButton button = (JButton) e.getSource();


        //Edit the currently selected SolarBody
        if(button.equals(edit)){

            backend.pause();
            restoreFields();
            existing = true;
            toggleEditable();
        }
        // Commit the changes that have been made
        else if(button.equals(commit)){

            if(commitChanges()){

                backend.unPause();
                toggleEditable();
            }
        }
        // Discard changes made and exit edit mode
        else if(button.equals(reject)){

            backend.unPause();
            toggleEditable();
            restoreFields();
        }
        // Hide this JFrame
        else if(button.equals(hide)){

            hide();
        }
        // Start edit mode for creating a new SolarBody
        else if(button.equals(makeNew)){

            backend.pause();
            existing = false;
            exampleFields();
            toggleEditable();
        }
    }

    /*
     * Set the Backend to reflect the changes that have been made in the Editor.
     *
     * @return true if the commit was successful.
     */
    private boolean commitChanges(){

        messageText.setText("None");

        synchronized(backend){

            double diameter;
            double mass;
            Triple pos;
            Triple vel;

            String errorField = "Diameter";
            try{
                diameter = Double.parseDouble(diameterText.getText());
                errorField = "Mass";
                mass = Double.parseDouble(massText.getText());
                errorField = "Position";
                pos = new Triple(Double.parseDouble(positionXText.getText()),
Double.parseDouble(positionYText.getText()),
Double.parseDouble(positionZText.getText()));
                errorField = "Velocity";
                vel = new Triple(Double.parseDouble(velocityXText.getText()),
Double.parseDouble(velocityYText.getText()),
Double.parseDouble(velocityZText.getText()));
            }
            catch(NumberFormatException e){
```

```
                messageText.setText(errorField + " must be Numerical.");
                return false;
            }

            if(existing){
                // Make changes to the Selected SolarBody
                current = solar.findSolarBody(current.getName());
                current.changeMass(mass);
                current.changeDiameter(diameter);
                current.changePosition(pos);
                current.changeVelocity(vel);
                current.setColor(redness, greenness, blueness);
            }
            else{
                // Create a whole new SolarBody
                Color col = new Color(redness, greenness, blueness);

                String name = nameText.getText();

                if(solar.findSolarBody(name) != null){

                    messageText.setText("This name already exists.");
                    return false;
                }

                // Add if its a Star
                if(bodyTypeCombo.getSelectedItem().equals("Star")){

                    backend.addSolarBody(new SynchronizedStar(name, mass, diameter,
pos, vel, new Triple(0,0,0), col));
                }

                // Add if its a Planet
                else if(bodyTypeCombo.getSelectedItem().equals("Planet")){

                    Star star = (Star) solar.findSolarBody((String)
orbitCentreCombo.getSelectedItem());
                    backend.addSolarBody(new SynchronizedPlanet(name, mass, diameter,
pos, vel, new Triple(0,0,0), col, star));
                }

                // Add if its a Planet
                else if(bodyTypeCombo.getSelectedItem().equals("Satellite")){

                    Planet planet = (Planet) solar.findSolarBody((String)
orbitCentreCombo.getSelectedItem());
                    backend.addSolarBody(new SynchronizedSatellite(name, mass,
diameter, pos, vel, new Triple(0,0,0), col, planet));

                }

                // Add The new SolarBody to the Combo Boxes
                solarBodySelector.addItem(name);
                frontend.centralBodyCombo.addItem(name);
            }
        }
        return true;
    }

    /*
     * Set the Fields back to the values determined by the currently selected
SolarBody
     */
    private void restoreFields(){

        messageText.setText("None");

        if(current instanceof Star){
            bodyTypeCombo.setSelectedIndex(0);
        }
        else{

            if(current instanceof Planet){

                bodyTypeCombo.setSelectedIndex(1);
            }
```

```
            else if(current instanceof Satellite){

                bodyTypeCombo.setSelectedIndex(2);
            }

orbitCentreCombo.setSelectedItem(((OrbitingSolarBody)current).getCentreOfOrbit().getNa
me());
        }
        nameText.setText(current.getName());
        massText.setText(Double.toString(current.getMass()));
        diameterText.setText(Double.toString(current.getSize()));
        positionXText.setText(Double.toString(current.getPosition().getX()));
        positionYText.setText(Double.toString(current.getPosition().getY()));
        positionZText.setText(Double.toString(current.getPosition().getZ()));
        velocityXText.setText(Double.toString(current.getVelocity().getX()));
        velocityYText.setText(Double.toString(current.getVelocity().getY()));
        velocityZText.setText(Double.toString(current.getVelocity().getZ()));
        Color color = current.getColor();
        redSlider.setValue(color.getRed());
        greenSlider.setValue(color.getGreen());
        blueSlider.setValue(color.getBlue());
    }

    /*
     * Set all text fields ready for the user to specify a new SolarBody.
     * Initiates them all to examples that work well in the system
     */
    private void exampleFields(){

        nameText.setText("Planet X");
        bodyTypeCombo.setSelectedIndex(1);
        massText.setText("2E23");
        diameterText.setText("4E7");
        positionXText.setText("8E12");
        positionYText.setText("0");
        positionZText.setText("0");
        velocityXText.setText("0");
        velocityYText.setText("-3000");
        velocityZText.setText("0");
        redSlider.setValue(128);
        greenSlider.setValue(128);
        blueSlider.setValue(256);
    }

    /*
     * Toggle which JComponents are editable according and whether this JFrame is
     * in Edit mode.
     */
    private void toggleEditable(){

        editable = !editable;

        commit.setEnabled(editable);
        reject.setEnabled(editable);
        nameText.setEnabled(editable && !existing);
        bodyTypeCombo.setEnabled(editable && !existing);
        orbitCentreCombo.setEnabled(editable && !existing);
        massText.setEnabled(editable);
        diameterText.setEnabled(editable);
        positionXText.setEnabled(editable);
        positionYText.setEnabled(editable);
        positionZText.setEnabled(editable);
        velocityXText.setEnabled(editable);
        velocityYText.setEnabled(editable);
        velocityZText.setEnabled(editable);
        redSlider.setEnabled(editable);
        greenSlider.setEnabled(editable);
        blueSlider.setEnabled(editable);

        solarBodySelector.setEnabled(!editable);
        edit.setEnabled(!editable);
        makeNew.setEnabled(!editable);
        hide.setEnabled(!editable);
    }
}
```

SolarSystem.java

```
/**
 * This is used to Bundle up every SolarBody in the System as one object.
 *
 * @author Adam Haigh
 * @version 1.1  07/02/2003
 */

public class SolarSystem implements Cloneable{
    private SynchronizedStar star;
    private SynchronizedPlanet[] planets;
    private SynchronizedSatellite[] satellites;
    private int numPlanets = 0;
    private int numSatellites = 0;

    /**
     * Creates a new empty Solar System.
     */
    public SolarSystem(){
        planets = new SynchronizedPlanet[numPlanets];
        satellites = new SynchronizedSatellite[numSatellites];
    }

    /**
     * Set the Solar Systems Star to be s.
     *
     * @param s The new Star for thew SolarSystem.
     */
    public synchronized void addStar(SynchronizedStar s){

        star = s;
    }

    /*
     * Put a new Planet in the Solar System bundle.
     *
     * @param p The Planet being added to the list of Planets.
     */
    public synchronized void addPlanet(SynchronizedPlanet p){

        SynchronizedPlanet[] temp = new SynchronizedPlanet[numPlanets];
        temp = planets;
        planets = new SynchronizedPlanet[numPlanets + 1];
        System.arraycopy(temp, 0, planets, 0, temp.length);
        planets[numPlanets] = p;
        numPlanets++;
    }

    /**
     * Put a new Satellite in the Solar System bundle.
     *
     * @param s The Satellite being added to the list of Satellites.
     */
    public synchronized void addSatellite(SynchronizedSatellite s){

        SynchronizedSatellite[] temp = new SynchronizedSatellite[numSatellites];
        temp = satellites;
        satellites = new SynchronizedSatellite[numSatellites + 1];
        System.arraycopy(temp, 0, satellites, 0, temp.length);
        satellites[numSatellites] = s;
        numSatellites++;
    }

    /* *
     * Get the current Star.
     *
     * @return A pointer to the Star.
     */
    public SynchronizedStar getStar()
{
        return star;
    }

    /**
```

```
     * Gets the Set of current Planets
     *
     *@return An array of all the Planets.
     */
    public SynchronizedPlanet[] getPlanets(){

        return planets;
    }

    /**
     * Get the Set of current Satellites
     *
     * @return An array of all the Satellites.
     */
    public SynchronizedSatellite[] getSatellites(){

        return satellites;
    }

    /**
     * Get a String representation of the whole Solar System.
     *
     * @return The String corresponding to the Positions, Velocities and and
Accelerations of all SolarBodies.
     */
    public String toString(){

        String s = star.toString() + "\n";

        for(int i=0; i<numPlanets; i++)
            s = s + planets[i].toString() + "\n";


        for(int i=0; i<numSatellites; i++)
            s = s + satellites[i].toString() + "\n";

        return s;
    }

    /**
     * Gets the Names of all SolarBodys as an array of Strings.
     *
     * @return An Array of Strings, each element being the name of a SolarBody.
     */
    public String[] listSolarBodys(){

        String[] s = new String[planets.length + satellites.length + 1];

        s[0] = star.getName();

        for(int i=1; i < planets.length + 1; i++) s[i] = planets[i-1].getName();

        for(int i=planets.length+1; i<s.length; i++) s[i] = satellites[i-
planets.length-1].getName();

        return s;
    }

    /**
     * Gets the Names if all planets as an array of Strings.
     *
     * @return An Array of Strings, each element being the name of a Planet.
     */
    public String[] listPlanets(){

        String[] s = new String[planets.length];

        for(int i=0; i < planets.length; i++) s[i] = planets[i].getName();

        return s;
    }

    /**
     * Gets the Names if all satellitess as an array of Strings.
     *
     * @return An Array of Strings, each element being the name of a Satellite.
     */
```

```
public String[] listSatellites(){

    String[] s = new String[satellites.length];

    for(int i=0; i < satellites.length; i++) s[i] = satellites[i].getName();

    return s;
}

/**
 * Gets the total number of SolarBodies in the SolarSystem.
 *
 * @return The number of SolarBodies.
 */
public int getNumberOfSolarBodys(){

    return 1 + planets.length + satellites.length;
}

/**
 * Gets the number of Planets in the SolarSystem
 *
 * @returns The number of Planets.
 */
public int getNumberOfPlanets(){

    return planets.length;
}

/**
 * Gets the number of Satellitess in the SolarSystem
 *
 * @returns The number of Satellites.
 */
public int getNumberOfSatellites(){

    return satellites.length;
}

/**
 * Searches for the first SolarBody whose name is equal to name.
 * returns null if such a SolarBody does not exist.
 *
 * @param name The name of the SolarBody being searched for.
 * @return The SolarBody that was found or null if none were found.
 */
public SolarBody findSolarBody(String name){

    if(name.equals(star.getName())) return star;

    for(int i=0; i< planets.length; i++)
        if(name.equals(planets[i].getName())) return planets[i];

    for(int i=0; i<satellites.length; i++)
        if(name.equals(satellites[i].getName())) return satellites[i];

    return null;
}

/**
 * Create a Deep Clone of the Solar System object
 *
 * @return an exact copy of the Solar System.
 */
public synchronized Object clone(){

    SolarSystem s = null;

    try{
        s = (SolarSystem) super.clone();
    }
    catch(CloneNotSupportedException e){
        System.err.println("SolarSystem can't clone");
    }

    s.star = (SynchronizedStar) star.clone();
```

```
        for(int i = 0 ; i < numPlanets ; i++){
            s.planets[i] = (SynchronizedPlanet) planets[i].clone();
        }

        for(int i = 0 ; i < numSatellites ; i++){
            s.satellites[i] = (SynchronizedSatellite) satellites[i].clone();
        }

        return s;
    }
}
```

SolarSystemException.java

```
/**
 * A general Exception generated for anything to do with the SolarSystem Applet
 * Includes a line number for the Exception.
 *
 * @author Adam Haigh
 * @version 1.1 10/02/2003
 */

public class SolarSystemException extends Exception{

    private String error;

    /**
     * Create a new SolarSystemException with a given message.
     *
     * @param s The message carried by the Exception.
     */
    public SolarSystemException(String s){

        error = s;
    }

    /**
     * Create a new SolarSystemException with no message.
     */
    public SolarSystemException(){

        error = "";
    }

    /**
     * Get the message carried by this exception.
     *
     * @return The String corresponding to the error message.
     */
    public String getMessage(){

        return "SolarSystemException: " + error;
    }
}
```

SolarSystemIOException.java

```
/**
 * A specific Exception generated for IOExceptions when reading data
 * for the Solar System Applet.
 * Includes a line number for the Exception.
 *
 * @author Adam Haigh
 * @version 1.1 10/02/2003
 */


public class SolarSystemIOException extends SolarSystemException{
    private String error;
    private int line;

    /**
     * Create a new SolarSystemIOException with a given error message and line number.
     *
     * @param s The message carried by this Exception.
     * @param i The line number in the file where the Exception occurred.
     */
    public SolarSystemIOException(String s, int i){
        error = s;
        line = i;
    }

    /**
     * Gets the message carried by this Exception.
     *
     * @return A String representation of the error message and line number it
occurred on.
     */
    public String getMessage(){
        return "SolarSystemIOException on line " + line +": " + error;
    }
}
```

SolarSystemReader.java

```
/**
 * The SolarSystemReader class Is used to parse files and attempt
 * to Create a SolarSystem object from the text.
 * All SolarBodys have to have different names.
 *
 * @author Adam Haigh
 * @version 1.1 07/02/2003
 */

import java.text.*;
import java.awt.Color;

public class SolarSystemReader{
    private static char ch;
    private static int index;
    private static String text;
    private static NumberFormat nf;
    private static int lineNumber = 1;

    /**
     * Attempts to parse the given String and use it to create a Valid Solar System
Object.
     *
     * @param t The String representing the SolarSystem to be Parsed.
     * @return The SolarSystem Object corresponding to the String.
     * @exception SolarSystemIOException If part of the String is not parseable to
create a SolarBody.
     */
    public static SolarSystem parseString(String t) throws SolarSystemIOException{

        index = 0;
        text = t;
        nf = NumberFormat.getInstance();

        // All previously generated Names
        String[] names = new String[10];
        int namePointer = 0;

        SolarSystem system = new SolarSystem();

        ch = text.charAt(index);
        index++;

        // Loop until <EOF>
        while(ch != '$'){

            skipSpace();
            // Ignore Lines that begin with a '#' key and empty lines.
            if(ch == '#' || ch == '\n') skipLine();

            else{
                // Read all Attributes for a SolarBody
                String s = readString();
                String name = readString();

                // Ensure name is not a duplicate of a previous Name
                for(int i=0; i<namePointer; i++){

                    if(name.equals(names[i])) throw new SolarSystemIOException
                        ("Duplicate Name Declaration of SolarBody.", lineNumber);
                }

                if(namePointer == names.length){
                    String[] temp = names;
                    names = new String[namePointer + 10];
                    System.arraycopy(temp, 0, names, 0, namePointer);
                }

                names[namePointer] = name;
                namePointer++;

                double mass = readDouble();
                double diameter = readDouble();
```

```
                Triple position = readTriple();
                Triple velocity = readTriple();
                Triple acceleration = new Triple(0,0,0);//readTriple();
                Color color = readColor();

                // Create a Star
                if(s.equalsIgnoreCase("Star")){
                    system.addStar(new SynchronizedStar
                        (name, mass, diameter, position, velocity, acceleration,
color));
                }
                // Create a Planet
                else if(s.equalsIgnoreCase("Planet")){

                    String centreOfOrbit = readString();
                    SynchronizedStar star = system.getStar();

                    if(star.getName().equals(centreOfOrbit)) system.addPlanet(new
SynchronizedPlanet
                        (name, mass, diameter, position, velocity, acceleration, color,
star));

                    else throw new SolarSystemIOException
                        ("Star name for Centre of orbit does not match system Star.",
lineNumber);

                }
                // Create a Satellite
                else if(s.equalsIgnoreCase("Satellite")){

                    String centreOfOrbit = readString();
                    SynchronizedPlanet[] planets = system.getPlanets();

                    for(int i=0; i<planets.length; i++){

                        if(planets[i] == null) throw new SolarSystemIOException
                            ("Planet name for Centre of orbit does not match any system
Planet.", lineNumber);

                        if(planets[i].getName().equals(centreOfOrbit)){
                            system.addSatellite(new SynchronizedSatellite
                                (name, mass, diameter, position, velocity, acceleration,
color, planets[i]));

                            break;
                        }
                    }

                }
                else throw new SolarSystemIOException
                    ("Incorrect SolarBody type.", lineNumber);
            }

            lineNumber++;
        }

        return system;
    }

    /*
     * Read the next bit of text as a String and parse it and return as a Triple.
     *
     * @return The Triple corresponding to the current bit of text.
     * @exception SolarSystemIOException If part of the String is not parseable to
create a Triple.
     */
    private static Triple readTriple() throws SolarSystemIOException{

        String[] s = new String[3];

        for(int i=0; i<3; i++) s[i] = readString();

        Number[] n = new Number[3];

        try{
            for(int i=0; i<3; i++) n[i] = nf.parse(s[i]);
        }
```

```
        catch(ParseException e){

            throw new SolarSystemIOException
                ("Incorrect number format for double", lineNumber);
        }

        double[] d = new double[3];

        for(int i=0; i<3; i++) d[i] = n[i].doubleValue();

        return new Triple(d);
    }

    /*
     * Read the next bit of text as a String and parse it and return as a Color.
     *
     * @return The Color corresponding to the current bit of text.
     * @exception SolarSystemIOException If part of the String is not parseable to
create a Color
     */
    private static Color readColor() throws SolarSystemIOException{

        String[] s = new String[3];

        for(int i=0; i<3; i++) s[i] = readString();

        Number[] n = new Number[3];

        try{
            for(int i=0; i<3; i++) n[i] = nf.parse(s[i]);
        }
        catch(ParseException e){
            throw new SolarSystemIOException
                ("Incorrect number format for integer", lineNumber);
        }

        int r = n[0].intValue();
        int g = n[1].intValue();
        int b = n[2].intValue();

        return new Color(r, g, b);
    }

    /*
     * Read the number as a String and then parse and return it as a double.
     *
     * @return The double corresponding to the current bit of text.
     * @exception SolarSystemIOException If part of the String is not parseable to
create a double.
     */
    private static double readDouble() throws SolarSystemIOException{

        try{
            String s = readString();

            Number n = nf.parse(s);

            return n.doubleValue();
        }
        catch(ParseException e){
            throw new SolarSystemIOException
                ("Incorrect number format for double", lineNumber);
        }

    }

    /*
     * Read Characters until the end of the word and return that word as a String.
     *
     * @return The current part of the String up until a ';' or ','
     */
    private static String readString(){

        skipSpace();

        String s = "";
```

```
        while(ch != ';' && ch != ','){
            s = s + ch;
            ch = text.charAt(index);
            index++;
        }

        ch = text.charAt(index);
        index++;

        return s;
    }

    /*
     * Read file until non-white-space is encountered.
     */
    private static void skipSpace(){

        // Check for '\n' or ' ' or '\t'
        while(ch == ' ' | ch == '\t'){
            ch = text.charAt(index);
            index++;
        }
    }

    /*
     * Move on to the start of the next line
     */
    private static void skipLine(){

        while(ch != '\n'){
            ch = text.charAt(index);
            index++;
        }

        ch = text.charAt(index);
        index++;
    }
}
```

Star.java

```
/**
 * The Star class implements the SolarBody interface to provide methods
 * that allow any given Star to alter itself in any way.  The original system
 * will instantiate the Star class once; representing the Sun.
 *
 * @author Adam Haigh
 * @version 1.2  26/1/03
 */
import java.awt.Color;

public class Star implements SolarBody, Cloneable{

    private String name;
    private double mass; // measured in Kg
    private double size; // Diameter measured in metres
    private double drawScaler; // The amount to rescale the Star by when re-drawing
    private Triple position;
    private Triple velocity;
    private Triple acceleration;
    private Color color;

    private int drawWidth;

    /**
     * Create a new Star with a full specification.
     *
     * @param n The Name of the Star
     * @param m The Mass of the Star
     * @param si The Diameter of the Star
     * @param p The Position of the Star. [0, 0, 0] is the centre of the Solar System
     * @param v The Velocity of the Star. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Star. [0, 0, 0] Shows no change in Velocity.
     * @param c The Color of the Star.
     */
    public Star(String n, double m, double s, Triple p, Triple v, Triple a, Color c){
        name = n;
        mass = m;
        size = s;
        position = p;
        velocity = v;
        acceleration = a;
        color = c;

        drawWidth = 1;
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public void changeMass(double newMass){

        mass = newMass;
    }

    /**
     * Sets the diameter to be newDiameter and resets the draw Width.
     *
     * @param newDiameter The new Diameter of the Star.
     */
    public void changeDiameter(double newDiameter){

        size = newDiameter;
        setDrawWidth(drawScaler);
    }

    /**
     * Sets a new Position for the Star.
     *
     * @param t The new position of the Star.
     */
    public void changePosition(Triple t){
```

```
    position = t;
}

/**
 * Sets a new Velocity for the Star.
 *
 * @param t The new Velocity of the Star.
 */
public void changeVelocity(Triple t){

    velocity = t;
}

/**
 * Sets a new Acceleration for the Star.
 *
 * @param t The new Acceleratio of the Star.
 */
public void changeAcceleration(Triple t){

    acceleration = t;
}

/**
 * Sets the size of the Star to be drawn in pixels.
 *
 * @param scaler The new draw width of the Star in pixels.
 */
public void setDrawWidth(double scaler){

    drawScaler = scaler;
    int w =  (int) Math.pow(size, scaler);

    if(w<4) drawWidth = 4;

    else drawWidth = w;
}

/**
 * Specifies the Color of the Star with RGB values in range 0-255
 *
 * @param r The value for red.
 * @param g The value for green.
 * @param b The value for blue.
 */
public void setColor(int r, int g, int b){

    color = new Color(r, g, b);
}

/**
 * Gets the current Position of the Star.
 *
 * @return The current Position of the Star as a Triple.
 */
public Triple getPosition(){

    return position;
}

/**
 * Gets the current Velocity of the Star
 *
 * @return The current Velocity of the Star as a Triple.
 */
public Triple getVelocity(){

    return velocity;
}

/**
 * Gets the current Acceleration of the Star
 *
 * @return The current Acceleration of the Star as a Triple.
 */
public Triple getAcceleration(){
```

```
        return acceleration;
    }

    /**
     * Gets the Mass of the Star.
     *
     * @return The mass of the Star as a double.
     */
    public double getMass(){

        return mass;
    }

    /**
     * Gets the diameter of the Star in metres.
     *
     * @return The diameter of the Star as a double.
     */
    public double getSize(){

        return size;
    }

    /**
     * Gets the Name of the Star.
     *
     * @return The Name of the Star as a String.
     */
    public String getName(){

        return name;
    }

    /**
     * Gets the size of the Star to be drawn in pixels
     *
     * @return The number of pixels wide the Star should be drawn.
     */
    public int getDrawWidth(){

        return drawWidth;
    }

    /**
     * Gets the Color of the Star.
     *
     * @return The Color of the Star.
     */
    public Color getColor(){

        return color;
    }

    /**
     * Gets a String representation of the Stars Position,
     * Velocity and Acceleration.
     *
     * @return The String representation of the Stars Position, Velocity and
Acceleration.
     */
    public String toString(){

        return name + position.toString() + velocity.toString() +
acceleration.toString();
    }

    /**
     * Performs a deep clone of the Star Object.
     *
     * @return An exact copy of the Star.
     */
    public Object clone(){

        Star s = null;

        try{
```

```
            s = (Star) super.clone();
        }
        catch(CloneNotSupportedException e){
            System.err.println("Star can't clone");
        }
        s.position = (Triple) position.clone();
        s.velocity = (Triple) velocity.clone();
        s.acceleration = (Triple) acceleration.clone();

        return s;
    }

    /**
     * Checks equality by checking that the Names match.
     *
     * @param body The SolarBody being compared to this one.
     * @return true if The name of body matches that of this Star.
     */
    public boolean equals(SolarBody body){

        return name.equals(body.getName());
    }
}
```

SynchronizedPlanet.java

```
/*
 * The SynchronizedPlanet implements all the same methods as the Planet
 * class but the methods that alter the attributes of the Planet are
 * synchronized so that alterations are not made during Cloning.
 *
 * @author Adam Haigh
 * @version 1.1  10/02/2003
 */

import java.awt.Color;

public class SynchronizedPlanet extends Planet{

    /**
     * Create a new SynchronizedPlanet with a full specification.
     *
     * @param n The Name of the Planet
     * @param m The Mass of the Planet
     * @param si The Diameter of the Planet
     * @param p The Position of the Planet. [0, 0, 0] is the centre of the Solar
System
     * @param v The Velocity of the Planet. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Planet. [0, 0, 0] Shows no change in Velocity.
     * @param c The Color of the Planet.
     * @param s The Star that the Planet is Orbitting.
     */
    public SynchronizedPlanet(String n, double m, double si, Triple p, Triple v,
Triple a, Color c, Star s){
        super(n, m, si, p, v, a, c, s);
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public synchronized void changeMass(double newMass){
        super.changeMass(newMass);
    }

    /**
     * Sets a new Position for the Planet.
     *
     * @param t The new position of the Planet.
     */
    public synchronized void changePosition(Triple t){
        super.changePosition(t);
    }

    /**
     * Sets a new Velocity for the Planet.
     *
     * @param t The new Velocity of the Planet.
     */
    public synchronized void changeVelocity(Triple t){
        super.changeVelocity(t);
    }

    /**
     * Sets a new Acceleration for the Planet.
     *
     * @param t The new Acceleratio of the Planet.
     */
    public synchronized void changeAcceleration(Triple t){
        super.changeAcceleration(t);
    }

    /**
     * Performs a deep clone of the Planet Object.
     *
     * @return An exact copy of the Planet.
     */
    public synchronized Object clone(){
```

```
        return super.clone();
    }
}
```

SynchronizedSatellite.java

```
/*
 * The SynchronizedSatellite implements all the same methods as the Satellite
 * class but the methods that alter the attributes of the Satellite are
 * synchronized so that alterations are not made during Cloning.
 *
 * @author Adam Haigh
 * @version 1.1  10/02/2003
 */
import java.awt.Color;

public class SynchronizedSatellite extends Satellite{

    /**
     * Create a new SynchronizedSatellite with a full specification.
     *
     * @param n The Name of the Satellite
     * @param m The Mass of the Satellite
     * @param si The Diameter of the Satellite
     * @param p The Position of the Satellite. [0, 0, 0] is the centre of the Solar
System
     * @param v The Velocity of the Satellite. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Satellite. [0, 0, 0] Shows no change in
Velocity.
     * @param c The Color of the Satellite.
     * @param pl The Planet that the Satellite is Orbitting.
     */
    public SynchronizedSatellite(String n, double m, double si, Triple p, Triple v,
Triple a, Color c, Planet pl){
        super(n, m, si, p, v, a, c, pl);
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public synchronized void changeMass(double newMass){
        super.changeMass(newMass);
    }

    /**
     * Sets a new Position for the Satellite.
     *
     * @param t The new position of the Satellite.
     */
    public synchronized void changePosition(Triple t){
        super.changePosition(t);
    }

    /**
     * Sets a new Velocity for the Satellite.
     *
     * @param t The new Velocity of the Satelite.
     */
    public synchronized void changeVelocity(Triple t){
        super.changeVelocity(t);
    }

    /**
     * Sets a new Acceleration for the Satelite.
     *
     * @param t The new Acceleratio of the Satellite.
     */
    public synchronized void changeAcceleration(Triple t){
        super.changeAcceleration(t);
    }

    /**
     * Performs a deep clone of the Satellite Object.
     *
     * @return An exact copy of the Satellite.
     */
    public synchronized Object clone(){
```

```
        return super.clone();
    }
}
```

SynchronizedStar.java

```
/*
 * The SynchronizedStar implements all the same methods as the Star
 * class but the methods that alter the attributes of the Star are
 * synchronized so that alterations are not made during Cloning.
 *
 * @author Adam Haigh
 * @version 1.1  10/02/2003
 */
import java.awt.Color;

public class SynchronizedStar extends Star{

    /**
     * Create a new SynchronizedStar with a full specification.
     *
     * @param n The Name of the Star
     * @param m The Mass of the Star
     * @param si The Diameter of the Star
     * @param p The Position of the Star. [0, 0, 0] is the centre of the Solar System
     * @param v The Velocity of the Star. [0, 0, 0] Shows no movement.
     * @param a The Acceleration of the Star. [0, 0, 0] Shows no change in Velocity.
     * @param c The Color of the Star.
     */
    public SynchronizedStar(String n, double m, double s, Triple p, Triple v, Triple
a, Color c){
        super(n, m, s, p, v, a, c);
    }

    /**
     * Sets a new value for the Mass.
     *
     * @param newMass The new Value for Mass.
     */
    public synchronized void changeMass(double newMass){
        super.changeMass(newMass);
    }

    /**
     * Sets a new Position for the Star.
     *
     * @param t The new position of the Star.
     */
    public synchronized void changePosition(Triple t){
        super.changePosition(t);
    }

    /**
     * Sets a new Velocity for the Star.
     *
     * @param t The new Velocity of the Star.
     */
    public synchronized void changeVelocity(Triple t){
        super.changeVelocity(t);
    }

    /**
     * Sets a new Acceleration for the Star.
     *
     * @param t The new Acceleratio of the Star.
     */
    public synchronized void changeAcceleration(Triple t){
        super.changeAcceleration(t);
    }

    /**
     * Performs a deep clone of the Star Object.
     *
     * @return An exact copy of the Star.
     */
    public synchronized Object clone(){
        return super.clone();
    }
}
```

Triple.java

```
/**
 * The Triple Class is used by each instance of a Solar Body three times.
 * Once to hold 3D position coordinates, once to hold 3D Velocity coordinates,
 * and once to hold 3D Acceleration Coordinates.
 * The underlying Data Structure is simply 3 doubles, each holds a value
 * corresponding to x, y, or z coordinates.
 *
 * @author Adam Haigh
 * @version 1.1  6/1/03
 */
public class Triple implements Cloneable{
    private double[] triple;

    /**
     * Create a new Triple object with values x, y and z.
     *
     * @param x The X-Value for the new Triple.
     * @param y The Y-Value for the new Triple.
     * @param z The Z_Value for the new Triple.
     */
    public Triple(double x, double y, double z){

       triple = new double[3];
       triple[0] = x;
       triple[1] = y;
       triple[2] = z;
    }

    /**
     * Create a new Triple object with x, y and z values taken from the array.
     *
     * @param t The 3-Length array to create the Triple from.
     */
    public Triple(double[] t){

       triple = t;
    }

    /**
     * Sets the Triple to be a new set of values.
     *
     * @param x The first value for the Triple.
     * @param y The second value for the Triple.
     * @param z The Third value for the Triple.
     */
    public synchronized void changeTriple(double x, double y, double z){

       triple[0] = x;
       triple[1] = y;
       triple[2] = z;
    }

    /**
     * Sets the value to position determined by dimension
     * dimension = 1 means x
     * dimension = 2 means y
     * dimension = 3 means z
     *
     * @param dimension Which element of the Triple to change.
     * @param value The new Value for the specified element of the Triple.
     */
    public synchronized void changeDim(int dimension, double value){

       triple[dimension-1] = value;
    }

    /**
     * Get the coordinates of the Triple.
     *
     * @return An array of length 3 representing the Triple.
     */
    public double[] getTriple(){

       return triple;
    }
```

```
    /**
     * Get the X-coordinate of the Triple.
     *
     * @return The value for the first element of the Triple.
     */
    public double getX(){

        return triple[0];
    }

    /**
     * Get the Y-coordinate of the Triple.
     *
     * @return The value for the second element of the Triple.
     */
    public double getY(){

        return triple[1];
    }

    /**
     * Get the Z-coordinate of the Triple.
     *
     * @return The value for the third element of the Triple.
     */
    public double getZ(){

        return triple[2];
    }

    /**
     * Get the coordinate of the Triple specified by dim.
     * dim = 1 means x
     * dim = 2 means y
     * dim = 3 means z
     *
     * @param dim Which element of the Triple to return.
     * @return The value of the element specified by dim.
     */
    public double getDim(int dim){

        return triple[dim-1];
    }

    /**
     * Get a String representation of the Triple
     *
     * @return The three elements of the Triple as a String.
     */
    public String toString(){
        return " [ " + triple[0] + ", " + triple[1] + ", " + triple[2] + " ] ";
    }

    /**
     * Create a Clone of the Triple
     *
     * @return A new copy of the Triple.
     */
    public synchronized Object clone(){

        Triple t = null;

        try{
            t = (Triple) super.clone();
        }
        catch(CloneNotSupportedException e){

            System.err.println("Triple can't clone");
        }
        t.triple[0] = getX();
        t.triple[1] = getY();
        t.triple[2] = getZ();

        return t;
    }
}
```

Solar.html

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
  <head>
    <title>
      A simulation of the Solar System
    </title>
    <meta name = "keywords" content = "Solar, Solar System, simulation, orbit, java,
applet">
  </head>

  <body bgcolor = "000000" text = "ffffff" link = "00ffff">

    <p>
      This applet is a scale model of the Solar System.
      I have adopted the following standards for Units:-
    <UL>
      <LI>
       Numbers are written in scientific notation.
      <LI>
       Masses are in Kg,
      <LI>
       Distances are in Metres,
      <LI>
       G = 6.673x10<sup>11</sup>m<sup>3</sup>kg<sup>-1</sup>s<sup>-2</sup>,
      <LI>
       Time format: [Years: Days: Hours: Mins: Secs]
    </UL>
  </p>

    <p>
      <font color="ff0000">
       To start the simulation Press Start and slide the top Scroll Bar to increase
the Speed.
       The bottom scrollbar can be used to adjust the Zoom level of the display.
      </font>
    </p>

    <center>
      <applet code = Frontend.class width = 1000 height = 600>
       <param name=SolarSystem value="Star; Sun; 1.989E30; 1.39E9; 0,0,0; 0,0,0; 255,
255, 0; Planet; Mercury; 3.303E23; 4.878E6; 5.7478E10,0,7.0615E9; 0, -47890, 0; 170,
195, 195; Sun; Planet; Venus; 4.870E24; 1.2102E7; 1.0801E11,0,6.4056E9; 0, -35030,0;
225, 180, 30; Sun; Planet; Earth; 5.976E24; 1.2756E7; 1.496E11,0,0; 0,-29790,0; 60,
135, 255; Sun;Planet; Mars; 6.418E23; 6.786E6; 2.2782E11,0,7.3586E9; 0,-24130,0; 195,
50, 10; Sun;Planet; Jupiter; 1.9E27; 1.43082E8; 7.7813E11,0,1.7767E10; 0,-13060,0;
230, 190, 0; Sun;Planet; Saturn; 5.688E26; 1.20536E8; 1.42563E12,0,6.1945E10; 0,-
9640,0; 255, 190, 0; Sun; Planet; Uranus; 8.684E25; 5.1118E7; 2.87073E12,0,3.87826E10;
0,-6810,0; 170, 170, 170; Sun; Planet; Neptune; 1.024E26; 4.9528E7;
4.49499E12,0,1.36863E11; 0,-5430,0; 0, 100, 255; Sun; Planet; Pluto; 1.31E22; 2.3E6;
5.65064E12,0,1.74355E12; 0,-4740,0; 80, 80, 130; Sun; Satellite; Moon; 7.35E22;
3476000; 1.496E11,3.844E8,0; 1023.1573,-29790,0; 127, 127, 127; Earth;Satellite; Io;
8.933E22; 3642000; 7.7813E11, 4.216E8, 1.7767E10; 17331.6, -13060, 0; 127, 127, 127;
Jupiter; Satellite; Europa; 4.797E22; 3130000; 7.7813E11, 6.709E8, 1.7767E10; 13739.6,
-13060, 0; 127, 127, 127; Jupiter; Satellite; Ganymede; 1.482E23; 5268000; 7.7813E11,
1.07E9, 1.7767E10; 10875.3, -13060, 0; 127, 127, 127; Jupiter; Satellite; Callisto;
1.076E23; 4806000; 7.7813E11, 1.883E9, 1.7767E10; 8205.1, -13060, 0; 127, 127, 127;
Jupiter; Satellite; Titan; 1.345E23; 5150000; 1.42563E12, 1.221830E9, 6.1945E10;
5572.5, -9640, 0; 127, 127, 127; Saturn; Satellite; Triton; 2.147E22; 2704000;
4.49499E12, 3.54760E8, 1.36863E11; 4389.8, -5430, 0; 127, 127, 127; Neptune;$">
      </applet>
    </center>

    <p>
      This Applet has the following features:
    <UL>
      <LI>
       All nine Planets of the Solar System, plus all Moons above 10<sup>22</sup>Kg in
Mass.
      <LI>
       Complete control of the Speed of the simulation.
      <LI>
       Almost any view desirable through a combination of Zoom, 3 angles and ability
to centre on different Planets.
      <LI>
       Manipulate the Planets directly through the Display.
```

```
      <LI>
        Seperate textual representation of the system.
      <LI>
        Editor module to allow complete control over existing Planets and ability to
define new ones.
    </UL>
  </p>

    <p>
      There are many other examples of Solar System Simulations on the web:
    <UL>
      <LI>
        <a
href="http://www.humnet.ucla.edu/humnet/french/faculty/gans/java/SolarApplet.html">A
Basic simulation including all Planets and some moons.</a>
      <LI>
        <a href="http://jove.geol.niu.edu/faculty/stoddard/JAVA/luminaries.html">A
simulation with 2 views. </a>
    </UL>
  </p>


    <HR>
    <H4>
      <CENTER>
        <a href="http://www.cogs.susx.ac.uk">Cogs Homepage</a>
        |
        <a href="http://www.cogs.susx.ac.uk/courses/csaiproj/">CSAI Projects</a>
      </CENTER>
    </H4>
    <HR>
  </body>
</html>
```

TestBackend.java

```
/**
 * The TestBackend class is a Command-Line Program used to test the accuracy
 * of the Backend.
 * When ran, 3 arguments should be specified;
 *   1) The Name of the Planet being Tested.
 *   2) The Step-size to be used by the Backend, in seconds.
 *   3) The total run-time to be tested, in seconds.
 *
 * The expectation is that the specified run-time is to be one complete orbital period
 * of the Specified Planet, or a multiple thereof.
 *
 * @author Adam Haigh
 * @version 1.1 3/4/2003
 */

public class TestBackend{

    public static void main(String[] args){

        // Ensure that the correct number of arguments are used.
        if(args.length != 3){

            usage();
            return;
        }

        // Create a Backend with the String parameter describing the Solar System.
        Backend backend = new Backend("Star; Sun; 1.989E30; 1.39E9; 0,0,0; 0,0,0; 255,
255, 0;\nPlanet; Mercury; 3.303E23; 4.878E6; 5.7478E10,0,7.0615E9; 0, -47890, 0; 170,
195, 195; Sun;\nPlanet; Venus; 4.870E24; 1.2102E7; 1.0801E11,0,6.4056E9; 0, -35030,0;
225, 180, 30; Sun; \nPlanet; Earth; 5.976E24; 1.2756E7; 1.496E11,0,0; 0,-29790,0; 60,
135, 255; Sun;\nPlanet; Mars; 6.418E23; 6.786E6; 2.2782E11,0,7.3586E9; 0,-24130,0;
195, 50, 10; Sun;\nPlanet; Jupiter; 1.9E27; 1.43082E8; 7.7813E11,0,1.7767E10; 0,-
13060,0; 230, 190, 0; Sun;\nPlanet; Saturn; 5.688E26; 1.20536E8;
1.42563E12,0,6.1945E10; 0,-9640,0; 255, 190, 0; Sun;\nPlanet; Uranus; 8.684E25;
5.1118E7; 2.87073E12,0,3.87826E10; 0,-6810,0; 170, 170, 170; Sun;\nPlanet; Neptune;
1.024E26; 4.9528E7; 4.49499E12,0,1.36863E11; 0,-5430,0; 0, 100, 255; Sun;\nPlanet;
Pluto; 1.31E22; 2.3E6; 5.65064E12,0,1.74355E12; 0,-4740,0; 80, 80, 130;
Sun;\nSatellite; Moon; 7.35E22; 3476000; 1.496E11,3.844E8,0; 1023.1573,-29790,0; 127,
127, 127; Earth;\nSatellite; Io; 8.933E22; 3642000; 7.7813E11, 4.216E8, 1.7767E10;
17331.6, -13060, 0; 127, 127, 127; Jupiter; \nSatellite; Europa; 4.797E22; 3130000;
7.7813E11, 6.709E8, 1.7767E10; 13739.6, -13060, 0; 127, 127, 127; Jupiter;
\nSatellite; Ganymede; 1.482E23; 5268000; 7.7813E11, 1.07E9, 1.7767E10; 10875.3, -
13060, 0; 127, 127, 127; Jupiter; \nSatellite; Callisto; 1.076E23; 4806000; 7.7813E11,
1.883E9, 1.7767E10; 8205.1, -13060, 0; 127, 127, 127; Jupiter; \nSatellite; Titan;
1.345E23; 5150000; 1.42563E12, 1.221830E9, 6.1945E10; 5572.5, -9640, 0; 127, 127, 127;
Saturn; \nSatellite; Triton; 2.147E22; 2704000; 4.49499E12, 3.54760E8, 1.36863E11;
4389.8, -5430, 0; 127, 127, 127; Neptune; \n$");

        // Set up variables used by Test Program.
        SolarSystem solar = backend.getSolarSystem();
        Star star = solar.getStar();
        Planet planet = (Planet) solar.findSolarBody(args[0]);
        int scaler = Integer.parseInt(args[1]); // Size of Time-step.
        int executions = Integer.parseInt(args[2])/scaler; // number of iterations
required.

        backend.setTimeStep(scaler);

        // Find and Print the initial Data.
        Triple tmp = subtractTriples(planet.getPosition(), star.getPosition());
        Triple initPos = new Triple(tmp.getX(), tmp.getY(), tmp.getZ());
        tmp = subtractTriples(planet.getVelocity(), star.getVelocity());
        Triple initVel = new Triple(tmp.getX()/scaler, tmp.getY()/scaler,
tmp.getZ()/scaler);
        double initOrbRad = pythag3D(initPos);
        double initAbsSpeed = pythag3D(initVel);

        System.out.println("Init Pos of " + planet.getName() + ": " +
initPos.toString() + " Relative to " + star.getName());

        System.out.println("Init Vel of " + planet.getName() + ": " +
initVel.toString() + " Relative to " + star.getName());
```

```
        System.out.println("Init Orbital Radius of " + planet.getName() + ": " +
initOrbRad + " Relative to " + star.getName());

        System.out.println("Init Abs Speed of " + planet.getName() + ": " +
initAbsSpeed + " Relative to " + star.getName());


        // Run the simulation
        for(int i=0; i<executions; i++) backend.timeStep();

        // Find and print the final Data.
        tmp = subtractTriples(planet.getPosition(), star.getPosition());
        Triple finalPos = new Triple(tmp.getX(), tmp.getY(), tmp.getZ());
        tmp = subtractTriples(planet.getVelocity(), star.getVelocity());
        Triple finalVel = new Triple(tmp.getX()/scaler, tmp.getY()/scaler,
tmp.getZ()/scaler);
        double finalOrbRad = pythag3D(finalPos);
        double finalAbsSpeed = pythag3D(finalVel);

        System.out.println("\nFinal Pos of " + planet.getName() + ": " +
finalPos.toString() + " Relative to " + star.getName());

        System.out.println("Final Vel of " + planet.getName() + ": " +
finalVel.toString() + " Relative to " + star.getName());

        System.out.println("Final Orbital Radius of " + planet.getName() + ": " +
finalOrbRad + " Relative to " + star.getName());

        System.out.println("Final Abs Speed of " + planet.getName() + ": " +
finalAbsSpeed + " Relative to " + star.getName());


        // Find and print the Difference between the initial and final values. ie/ The
Change.
        System.out.println("\nChange in Position of " + planet.getName() + ": " +
subtractTriples(finalPos, initPos).toString() + " Relative to " + star.getName());

        System.out.println("Change in Velocity of " + planet.getName() + ": " +
subtractTriples(finalVel, initVel).toString() + " Relative to " + star.getName());

        System.out.println("Change in Orbital Radius of " + planet.getName() + ": " +
(finalOrbRad - initOrbRad) + " Relative to " + star.getName());

        System.out.println("Change in Abs Speed of " + planet.getName() + ": " +
(finalAbsSpeed - initAbsSpeed) + " Relative to " + star.getName());
    }

    // Describe the arguments to be used.
    private static void usage(){

        System.out.println("Usage: java TestBackend <Planet> <StepSize> <Runtime>");
    }

    // Subtract the fields of the dirst Triple from their corresponding fields in the
second Triple.
    private static Triple subtractTriples(Triple first, Triple second){

        double[] f = first.getTriple();
        double[] s = second.getTriple();

        for(int i=0; i<3; i++)
            f[i] = f[i] - s[i];

        return new Triple(f);
    }

    // Perform the 3D-Pythagorous equation on the Triple.
    // Ans = Sq. Root(X^2 + Y^2 + Z^2)
    private static double pythag3D(Triple t){

        return Math.sqrt(Math.pow(t.getX(),2) + Math.pow(t.getY(),2) +
Math.pow(t.getZ(),2));
    }
}
```

Appendix 2 - Test results, taken from Test Output

Step-size of 360 seconds

Mercury:
Position: [-1040382.4129943848, 3.626022341350182E8, -113609.3933801651]
Velocity: [311.78634397643714, 1.1772287723506452, 38.26566548370468]
Orbital Radius: 88747.49773406982
Absolute Speed: -0.1469891637971159

Venus:
Position: [-1531109.1682128906, 2.0364143418143266E8, 47832.77289581299]
Velocity: [70.94096430153752, 0.020495748743996955, 4.093213345725181]
Orbital Radius: -1333953.791732788
Absolute Speed: 0.05157636903459206

Earth:
Position: [1.3211924736669922E8, 1.0135075009672309E9, -152839.36058544042]
Velocity: [208.23203191726523, 25.62072568027361, 0.09931157635650396]
Orbital Radius: 1.3554932542077637E8
Absolute Speed: -24.892337259509077

Mars:
Position: [1.0727008868560791E7, -3.6936567066884536E8, 563265.1195392609]
Velocity: [-38.996199431831755, 2.263050267829385, -1.305824422589075]
Orbital Radius: 1.1038858456176758E7
Absolute Speed: -2.231501354737702

Jupiter:
Position:[9.499373313398071E9, 4.011788415145029E10, 3.4374392607299805E7]
Velocity: [670.1149577268869, 108.72744787958436, 15.233377136661753]
Orbital Radius: 1.0518483462658081E10
Absolute Speed: -91.39379764903424

Saturn:
Position:[-2.1233918537963867E9,-6.843742745001351E10,-1.14220944519752E8]
Velocity: [-464.51865120284333, 29.631266887497986, -20.561845890514764]
Orbital Radius: -4.837238275800781E8
Absolute Speed: -18.389555110426954

Step-size of 3600 seconds

Mercury:
Position: [-1814941.8819198608, 3.9933995295359874E8, -208711.65135860443]
Velocity: [469.19659854329296, 1.8252003355883062, 57.60449594999235]
Orbital Radius: -449925.9807739258
Absolute Speed: 0.5079255717646447

Venus:
Position: [-1934595.315612793, 3.051705956359236E8, 24317.092664718628]
Velocity: [140.40954773996182, 0.15435015995899448, 8.213323008063316]
Orbital Radius: -1499398.4714813232
Absolute Speed: 0.1280122038515401

Earth:
Position: [1.3113943627606506E9, 7.361755868989283E9, -175970.66821660986]
Velocity: [1463.7710701136862, 179.90902611807905, 0.0965125285483317]
Orbital Radius: 1.490848161026062E9
Absolute Speed: -143.75043556807577

Mars:
Position: [1.07579357449646E7, -3.261779107859837E8, 565093.8459367752]
Velocity: [-26.132381879445013, 2.2477695513043727, -0.8903817594406432]
Orbital Radius: 1.1003938783660889E7
Absolute Speed: -2.2336013446038123

Jupiter:
Position: [-1.6058757762168945E10, 4.3093493025866E11,-1.858169275508175E9]
Velocity: [6019.966061769408, 2390.6124554811686, 126.9265795459684]
Orbital Radius: 9.728730350993494E10
Absolute Speed: -808.7987398547593

Saturn:
Position:[1.893803173325439E10,3.0257102790017124E10,-2.92290660579529E7]
Velocity: [148.27275890688884, 57.12555737344519, 4.436499830377921]
Orbital Radius: 1.923571015889258E10
Absolute Speed: -55.97751063209398

Step-size of 36000 seconds

Mercury:
Position: [-1.5904965929199219E7, 7.979662839784356E8, -1939146.255249977]
Velocity: [2069.1277097755383, 15.559393679504865, 254.165979357299]
Orbital Radius: -1.0523731106010437E7
Absolute Speed: 29.80752457727067

Venus:
Position: [-6528725.915008545, 6.221146243516858E8, -247836.60442733765]
Velocity: [609.5007716733381, 1.7810398902220186, 36.034933020208506]
Orbital Radius: -4743372.352462769
Absolute Speed: 3.539832915288571

Earth:
Position:[-1.6443107014578247E9,-6.2108282342952385E9,-129653.43420123123]
Velocity: [-1039.0362473911157, -129.3773854597821, 0.10116436041907007]
Orbital Radius: -1.5140101725369873E9
Absolute Speed: 147.4137070520701

Mars:
Position: [1.0913199780426025E7, 2.9647092448170745E8, 566095.3626203537]
Velocity: [122.48266421239606, 2.3823044012206083, 3.910471899067008]
Orbital Radius: 1.1118581511169434E7
Absolute Speed: -2.0711009047772677

Jupiter:
Position:[-3.265327247347602E11,9.255943507302981E11,-9.305118709659847E9]
Velocity: [10026.098433859239, 8306.644248012963, 278.51026885141073]
Orbital Radius: 2.515878687310382E11
Absolute Speed: -1960.6933243450458

Saturn:
Position:[-5.215940730274707E10,8.657932774388147E11,-8.075396964848572E9]
Velocity: [4755.375898368609, 2020.9202063069306, 189.73318680627534]
Orbital Radius: 1.9750058112649316E11
Absolute Speed: -656.6834899563546

Appendix 3 - Log

Autumn Term Week 4 - 31st October 2002 - 1 hour
Wrote Project proposal

Week 5 1st - 7th November 2002 - 5 hours
Made notes of existing systems found on the internet.
Made notes of core and extended functionality for my system based on the existing systems.

Week 6 8th - 14th November 2002 - 4 hours
Wrote up Analysis of existing systems.
Wrote up notes for requirement analysis including what I intend to implement and what is not possible from the existing systems.

Week 7 15th - 21st November 2002 - 2 hours
Wrote analysis of Existing System 'Red Shift 4'
Did a Grammatical parse for objects and behaviours.

Week 8 22nd - 28th November 2002 - 8 hours
Wrote up existing work into a single report.
Wrote a list of function points.
Started on UML Class diagram.
Created Collaboration and Sequence diagrams for most function points

Week 9 29th November - 5th December 2000 - 5 hours
Finished UML Class, Collaboration and Sequence Diagrams.
Produced a domain analysis of key issues to overcome.
Produced a draft interim report.

Week 10 6th - 12th December 2002 - 4 hours
Refined domain analysis.
Produced Use Case Diagrams.
Finalised and the interim report.

· **Major Milestone - Handed in Interim Report**

Spring Term Week 1 6th-12th January 2003 - 4 hours
Started Prototyping.
First version of SolarBody, Planet, Star and Triple.java
Start of Backend. Basic version where only interaction of a Solar Body is with the Body it is orbitting.

Week 2 13th-19th January 2003 - 7 hours
Completed Backend to first executable version.
Created short Test programs: TestTriple and TestBackend

Week 3 20th-26th Jauary 2003 - 8 hours
Created basic Frontend GUI.
Created Display and incorperated into Frontend
Put Backend on its own thread to be started within Frontend

Week 4 27th Jan - 2nd February 2003 - 6 hours
Worked on Display and Backend to iron out bugs.

Week 5 3rd-9th February 2003 - 8 hours
Adapted Backend to allow alterations to the size of the time-step
Put Display on its own Thread.
Created a basic Data Window and incorporated into Frontend

Week 6 10th - 16th February 2003 - 7 hours
General bug fixing within Backend and Display.
Presented basic working Prototype including Sun and Earth.

Week 7 17th - 23rd February 2003 - 12 hours
Extended Backend so that accelerations occur between all Solar Bodies.
Extended to include Moon orbitting Earth.
Refinements to Data Window.
Included functions to draw Solar Body Names and Velocity Lines

Week 8 24th Feb - 2nd March 2003 - 9 hours
Extended to include all 9 Planets plus all Satellites above 10^{22}Kg in mass.
Increased power of zoom to 5001 times magnification.
Altered draw sizes of Planets to allow good viewing at any zoom level.

Week 9 3rd - 9th March 2003 - 11 hours
Wrote functions to allow dragging of Solar Bodies within the Display.
Started work on the Solar Editor.

Week 10 10th - 16th March 2003 - 12 hours
Extended Solar Editor.
Wrote function to allow Viewing angle to be changed.
Presented a good working second version.

Easter Week 1 - 17th - 23rd March 2003 - 12 hours
Worked on Solar Editor to improve usability.
General Code tidying and improvements to comments.
Added extras such as Tooltips.

Easter Week 3 - 31st March - 6th April 2003 - 20 hours
Wrote basic html page for displaying Applet on the web.
Produced Testing for Backend.
Started Final Report, main work to Analysis and Design sections.

Week 4 7th - 13th April 2003 - 15 hours
Refinements to existing sections of report.
Wrote up System Architecture and Implementation sections.
Produced Tables and Graphs of Testing Section.

Week 5 14th - 20th April 2003 - 15 hours
Wrote up Testing and Conclusion sections.
Finalised and Submitted Draft Report.

Summer Term 29th April - 6th May - 30 hours
Made changes to Draft Report.
Added User testing from questionaires
Finalised and Submitted Final Report

· **Major Milestone - Handed in Final Report**