Project Report 2004

solar: a Solar System Simulator

Author: Sam Morris

Supervisor: Dr. Richard Banach

Abstract

solar: a Solar System Simulator

Author: Sam Morris

The aim of this project is to create an interactive solar system simulator that allows the creation of arbitrary dynamic systems. A survey of other work in the field is carried out and a requirements list is formulated. Novel aspects of the program's design and implementation are described.
A tour of the finished program is presented, along with the results of testing and an analysis of the same. In the conclusion, proposals for improving the program are outlined.

Supervisor: Dr. Richard Banach

Acknowledgements

I would like to thank everyone who gave me help and encouragement in connection with my project. My family, my supervisor, my lecturers and my wonderful friends, you know who you are.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Proposal

The aim of the project was to create an interactive three-dimensional solar system simulator, akin to a “build your own” kit. The program was to be capable of simulating both real and artificial solar systems, as well as some interesting astronomical phenomena.

The work was carried out with regard to creating a program that could be adapted for teaching and demonstration; however the implementation described in this report does not go down that road itself. Instead care was taken to make the system extensible; for example, the use of an object oriented language made it easier to increase module cohesiveness and overall system flexibility.

The text of the original project proposal can be seen in Appendix A. It contains a list of preliminary project objectives and a draft plan consisting of a sequence of milestones and dates.

1.2 About the report

The report is divided up into chapters that follow the progression of the system from design to evaluation. The purpose of this chapter is to place the project in context and provide the rationale for undertaking it.

Chapter 2 gives a brief summary of the history behind the different models for the motion of the planets. It examines other projects in the field in order to gain an idea of the type of software already out there. Finally it specifies and justifies the platform upon which the software was developed and run.

Chapter 3 lists functional and non-functional requirements for the program, and continues with the design of the system. A method for calculating gravitational interactions between different bodies is presented. The system architecture is then developed, along with an overview of the design of the classes and modules that comprise the program.

Chapter 4 goes into detail about parts of the implementation that were interesting or challenging, including a more detailed overview of how gravitational forces were calculated and some notes about aspects of the program concerning user interaction.

Chapter 5 presents a short tour of the finished product, constructing a solar system consisting of a single sun and a planet. Some more interesting systems are then demonstrated, and the chapter ends with information about the batch mode feature of the program.

Chapter 6 contains information about the methods used to test the system to ensure that it operates correctly. Some test results are presented and analysed for accuracy and reliability.

Chapter 7 summarises what the project has achieved. The project is evaluated against the requirements stated in chapter 3, and recommendations for further work are put forward.

Chapter 2

Background information and research

2.1 Physics

It was generally believed that the Sun, Moon and other planets all orbited the Earth in concentric circles until the 16th century, when Nicolaus Copernicus published [Cop43]. This work put forward the idea that that the hitherto discovered celestial bodies, Earth included, orbited the Sun. Although this idea was not acceptable to the church of the day, it did explain certain quirks in the motion of the planets that the old model did not, such as the retrograde motion of the planets as they traversed the night sky.

Seventy years later, Johannes Kepler published a series of books in which he described three laws that he had derived from the observations of his colleague, Tycho Brahe. [Kep09] [Kep19] [Kep21] Briefly, his laws state that planets move in an ellipse with the Sun at one focal point, that they move faster when near the Sun's focal point and that planets closer to the sun orbit it at higher speeds.

Kepler's laws did a good job of explaining *what* was going on in the heavens, but because they were only derived from observational data, they could not explain *why* the planets moved in the way that they did. This would be done by another man: Isaac Newton.

In [New87], Newton defined the principles and laws of motion and Universal Gravitation—"Universal" because Newton realised that his laws applied equally to objects in the heavens (the moon orbiting the earth), and objects on the Earth (the famous apple falling from its tree). Until then, it was believed that the two domains were entirely separate and governed by different physical laws.

Newton's laws allowed the motions of the planets to be predicted to a hitherto unseen degree of accuracy. It is possible to plot orbits to take spacecraft to other planets and beyond, all with the aid of Newton's 18th century mathematics.

Of course, the story does not end there. Newton's laws do not account for the orbit of Mercury; it happens that Mercury's perihelion (point of closest approach) moves each time it orbits the sun.[Wud98] This was not explained until the advent of Einstein's Theory of General Relativity. The cycle continued with Relativity being usurped by String Theory/M-Theory[Gre00]. Nevertheless, Universal Gravitation remains the first scientifically accurate description of the motion of the planets and as such, is worth investigating.

2.2 Other projects

An overview of some projects related to this one:

2.2.1 Celestia

Celestia[cel] is a highly advanced planetarium; written for KDE in C++, it allows the user to visit and examine data for over 100,000 stars, planets and other interesting objects throughout the Milky Way. It comes with detailed texture maps of famous stellar objects and is capable of producing images of very high quality (see Figure 2.1).

Celestia is an extremely formidable project; however it does not use gravity to simulate the motion of stellar objects; as far as the physics are concerned, Celestia is an orrery—a mechanical model of the solar system.

2.2.2 Orrery

The ironically named Orrery[orr] is a gravitational simulation of a solar system; the user can play and pause the simulation, reversing time if desired; however time always runs at a fixed, rather slow rate. The gravitational constant (see page 15) is not the same as that of our own universe. This makes simulation of a "real" solar system difficult, although it does make the program more intuitive in some respects; for example, the unit of distance is the pixel (at the default zoom level). See Figure 2.2.

Orrery is a Java applet, and as such can be embedded in a web page. This makes it convenient to use as a teaching tool. However the quality of animation is quite poor as the frame rate is low. It also only calculates gravity in two dimensions.

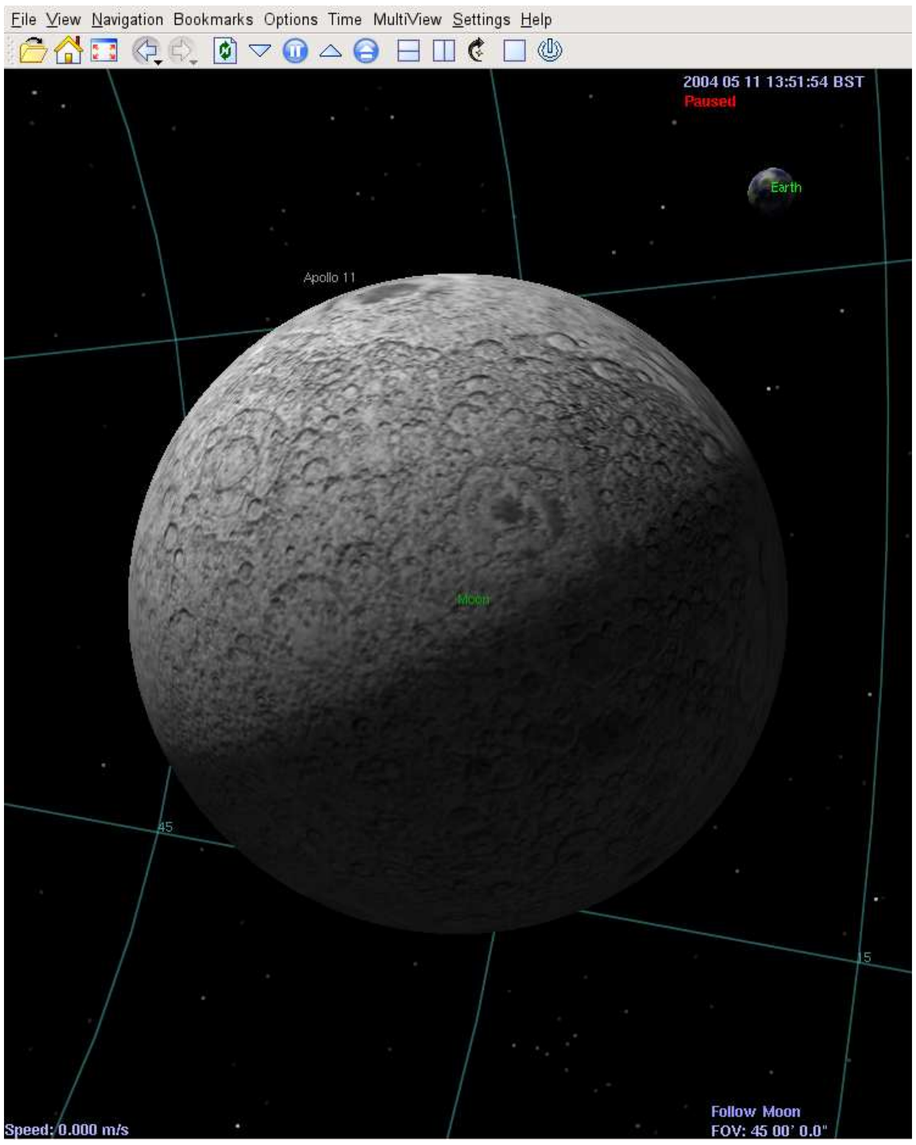


Figure 2.1: Celestia displaying the Moon, Apollo 11 and Earth.

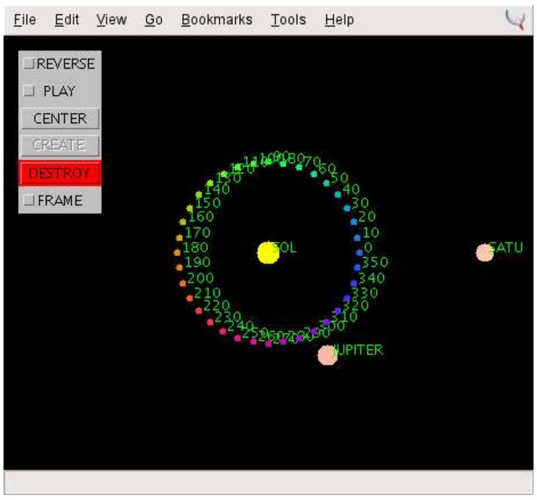
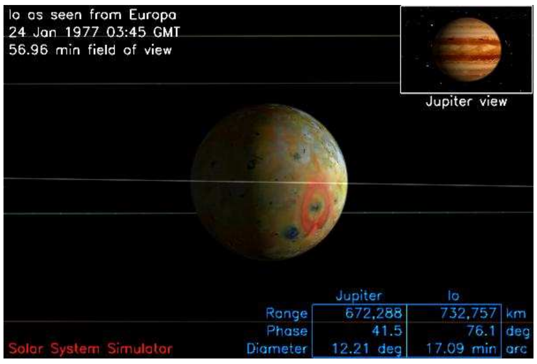


Figure 2.2: Orrery demonstrating an asteroid belt.

2.2.3 JPL Solar System Simulator

JSSS[jss] is a program for rendering pictures of the objects in our solar system. The user selects a position and target from a list of planets and moons, and specifies the date. The size of the picture, either in terms of the field of view, or the desired width of the target in the picture, is also entered. The program then calculates the positions of the planets at the specified date, and creates a picture such as seen in Figure 2.3.

JSS is not an interactive program; it is a batch system that is accessed via a CGI script on the NASA Jet Propulsion Laboratory's web page.



Io as seen from Europa
24 Jan 1977 03:45 GMT
56.96 min field of view

	Jupiter	Io	
Range	672,288	732,757	km
Phase	41.5	76.1	deg
Diameter	12.21 deg	17.09 min	arc

Figure 2.3: JPL Solar System Simulator showing a view of Io from Europa, Jan 1977.

2.3 Platform

The following decisions were made regarding the platform upon which the project would be developed and run.

Language: C++ C++ is a powerful, mature, object-oriented language. C++ code can be highly portable to different operating systems, if carefully written.

Development platform: Debian GNU/Linux Debian's[deb] Unix-like behaviour makes it a natural choice for software development. Automated package management and dependency handling make the process of experimenting with other libraries and software components simple.

Graphics: OpenGL A robust, efficient graphics library, OpenGL[ea03] allows the easy creation of sophisticated three-dimensional graphics on many different platforms.

User interface: GTKmm[gtkb] C++ bindings for the popular GTK+ widget toolkit[gtka]. Originally created for the X Window System, GTK+ has been ported to other platforms. It would also have been possible to use QT, as used by the KDE project[kde]—both toolkits have similar capabilities, and so selection came down to personal preference.

Data storage: libxml++ [xml] An efficient, cross platform library for the manipulation of eXtensible Markup Language files. The state of the simulation is regular and hierarchical, and so lends itself well to being serialised as XML. libxml++ also abstracts away the mundane and error-prone task of loading and parsing saved simulation states, replacing it with the simpler task of traversing a Document Object Model[ea98] tree.

These choices allowed the project to run on a wide variety of Unix-like operating systems, as well as Microsoft Windows.

Chapter 3

Design

3.1 Requirements

This section outlines the implementation requirements for the simulator. Since it was not expected for all the requirements to be completed due to time constraints, they were prioritised so that the important parts of the program could be written in the time available.

These features are summarised in Table 7.1 on Page 46.

3.1.1 Functional requirements

The functional requirements consist of specific features that should be present in the finished program.

- The aim of the project is to create a graphical program[R1] that simulates interactions between many bodies due to gravitational forces[R2]. The program should simulate these interactions in three dimensions[R3].
- The simulation must be interactive[R4], presenting the user with a graphical interface so that they can control what is going on. The user must be able to control the viewpoint of the simulation[R5], alter the flow of time[R6], and stop and start the simulation[R7].
- The program must provide visual aids[R8] for locating planets, since if everything was drawn to scale, it would be impossible to see anything (see Section 3.3.3).
- The user interface must allow the user to edit the universe[R9]; that is, allow them to create new planets and edit existing ones.

- It must be possible to save[R10] the state of the simulation to a file on disk and restore it[R11] in a subsequent run of the program. This will allow some interesting example solar systems[R12] to be distributed with the program.
- It should be possible to export data[R13] from the simulation. Scripts should be written to plot graphs[R14] or create animations[R15] of this data.
- The program should be cross platform[R16], that is it should work on both GNU/Linux and Microsoft Windows.

3.1.2 Non-functional requirements

The following are constraints on the behaviour of the program:

- The user interface must be relatively simple[R17]. The Gnome Human Interface Guidelines[CB02] will be consulted when designing the GUI.
- The quality of animation used when displaying the simulation should be high[R18]. This includes animating at a high frame rate.
- OpenGL provides a wide variety of functions that allow graphics created with it to be very pretty. The program should make use of some of these functions[R19].

3.2 What is gravity?

As far as this simulation is concerned, gravity is an attractive force that acts instantaneously between every particle of matter in the universe. The magnitude of the force is given by Newton's Law of Universal Gravitation:

$$F = \frac{GM_1M_2}{r^2} \tag{3.1}$$

M_1 and M_2 are the masses of the particles concerned, and r is the distance between them. G is the Gravitational Constant—a scaling factor that determines how strong the force of gravity is[1].

Once we know the overall force on a particle (remember, every particle in the universe is affected by every other particle) then we can combine it with

[1]In our universe, G is equal to $6.673 \times 10^{-11} Nm^2kg^{-1}$. This means that two masses of one kilogram each, placed one meter apart will attract each other with a force of $6.673 \times 10^{-11} N$.

Newton's Second Law of motion and two of the kinematic equations, where m is the mass of the particle, a is its acceleration, u is its initial speed, v is its final speed, t is the amount of time over which we are calculating the particle's movement and s is the final relative displacement of the particle:

$$F = ma \qquad v = u + at \qquad s = ut$$

Solving for s will give the position of the particle after t seconds have elapsed.

Note that since the distance term in Equation 3.1 is squared, it will dominate the overall result at great distances. This reveals a possible optimisation. Using the values from Table B.1 and Equation 3.1:

- The force between the Sun and Earth is $3.60 \times 10^{22} N$.
- The force between the Sun and Jupiter is $4.14 \times 10^{23} N$.
- The force between Earth and Jupiter is between $1.90 \times 10^{18} N$ and 8.75×10^{17}.

The force that Jupiter, the second most massy body in the solar system, exerts on the Earth is on average four orders of magnitude less than the force exerted upon the Earth by the Sun. The next massiest body, Saturn, has one third the mass of Jupiter, and is twice as far from the sun—it will have even less of an effect on the Earth. It seems that it is not strictly necessary to simulate the affect of every particle in the universe on every other particle after all. However, given the speed of today's computers there is no harm in doing so, especially since the planets can be treated as as point masses, instead of simulating each of their atoms.

3.3 Program architecture

A popular design pattern that occurs in object oriented software design is Model, View, Controller (MVC). Each module (or class) is designated as one of a Model, a View or a Controller. The separation of function helps to keep inter-class coupling low, and intra-class cohesiveness high. This is desirable because it reduces program complexity, which means that less time and effort are required to debug problems and add new features.

3.3.1 SI units

It was decided to use standard SI units throughout the program, in order to make creating realistic models of our, and other, solar systems easier. Distance is measured in meters, mass in kilograms, time in seconds and force in Newtons.

3.3.2 Model classes

The Model classes represent both intangible and concrete *things* in the simulation.

Vector

Vector (see Figure 3.1) moddled a collection of, and was constructed from, three floating point numbers. It was used to represent planet positions and velocities, the three components corresponding to the three spacial dimensions. A Vector could also be constructed from a libxml++ node; this was done when loading a saved simulation into memory.

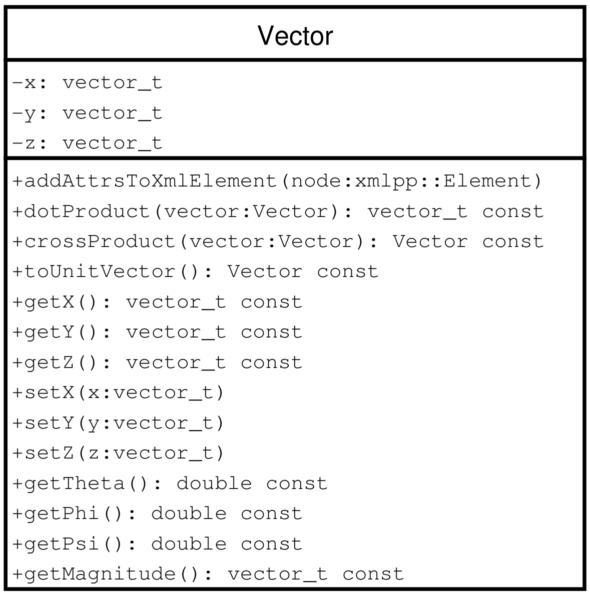

Vector
-x: vector_t
-y: vector_t
-z: vector_t
+addAttrsToXmlElement(node:xmlpp::Element)
+dotProduct(vector:Vector): vector_t const
+crossProduct(vector:Vector): Vector const
+toUnitVector(): Vector const
+getX(): vector_t const
+getY(): vector_t const
+getZ(): vector_t const
+setX(x:vector_t)
+setY(y:vector_t)
+setZ(z:vector_t)
+getTheta(): double const
+getPhi(): double const
+getPsi(): double const
+getMagnitude(): vector_t const

Figure 3.1: Vector class diagram.

Vector was also used to represent colours, since OpenGL considers a colour to be the standard triple of red, green and blue components.

It was decided to create a specialised class to represent a triple of floating point numbers, rather than to make use of C++'s STL template, std::vector, because there would be no need to model the situation in a higher number of dimensions and because of the additional complexity of template programming.

A Vector had accessor and mutator methods for each of its three components. Additional methods were defined to manipulate Vectors in ways that would be useful within the context of the simulation:

crossProduct Returned the cross, or vector, product of two vectors. The result was a third vector that is orthogonal/perpendicular to both vectors. It was useful for calculating camera placement values.

getMagnitude Returned the length of a vector. It was calculated by summing the squares of a vector's components, and taking the square of the result.

toUnitVector Returned a new Vector with a length of 1, that pointed in the same direction as the old one. This was done by dividing each component of the vector by the vector's magnitude.

dotProduct Returned the dot, or scalar, product of two unit vectors. This was the cosine of the angle that would have been made between them if they shared a common start point. It was useful for calculating the intersection of a ray and a planet (see page 21), for visibility calculations and for calculating the "absolute" angle components of a Vector.

getTheta, getPhi, getPsi These were used while determining the gravitational force between two Planets. They returned the angle between a Vector and the x, y and z axes, respectively.

C++'s operator overloading capabilities were used to provide an elegant way to add and subtract Vectors, and to multiply them by scalar values.

Planet

For the purposes of the simulation, all heavenly bodies were Planets (see Figure 3.2). A Planet had a mass, a radius, and a name. Vectors were used to represent a planet's position, velocity and colour. All these properties had corresponding accessors and mutators.

A Planet could be constructed from a libxml++ node. This was done while loading a saved simulation into memory.

Finally, the getGravity method returned a Vector representing the attractive force due to gravity, in Newtons, between two instances of Planet.

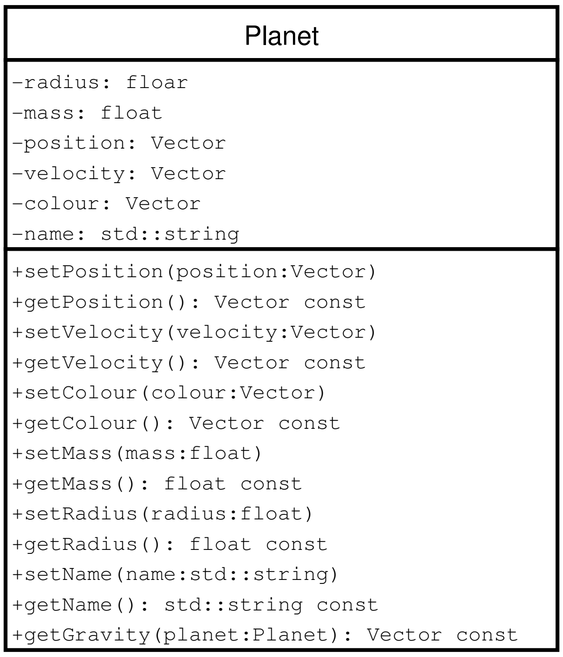

Planet
-radius: floar
-mass: float
-position: Vector
-velocity: Vector
-colour: Vector
-name: std::string
+setPosition(position:Vector)
+getPosition(): Vector const
+setVelocity(velocity:Vector)
+getVelocity(): Vector const
+setColour(colour:Vector)
+getColour(): Vector const
+setMass(mass:float)
+getMass(): float const
+setRadius(radius:float)
+getRadius(): float const
+setName(name:std::string)
+getName(): std::string const
+getGravity(planet:Planet): Vector const

Figure 3.2: Planet class diagram.

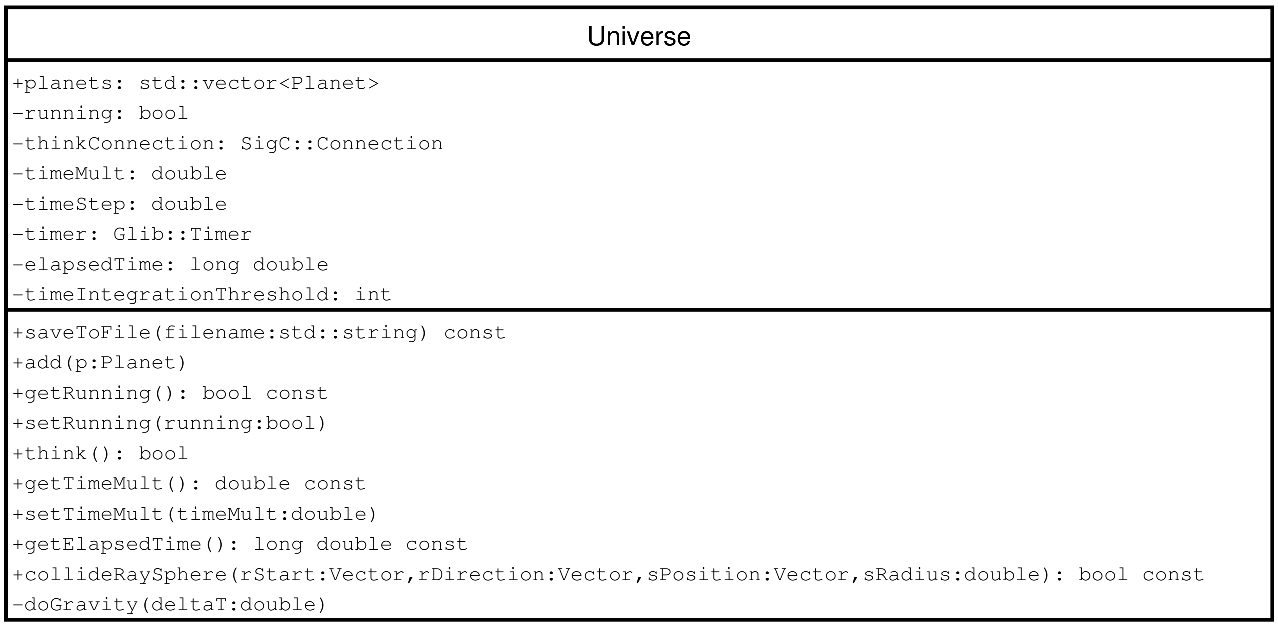
Universe

A Universe (Figure 3.3) was a collection of Planets. It was constructed from a filename which, if non-empty, caused a saved simulation state to be restored from disk.

A Universe ran in one of two modes. Both depended on the repeated calling of think, the method where the states of all the planets were examined and updated.

Interactive mode A SigC::Connection (see page 29) connected Glib's idle signal to the think method. Glib was then able to call this method repeatedly. The simulation could be paused and resumed by breaking and reconnecting thinkConnection. This was done from the setRunning method.

The difference in time between the current and previous calls to think was used as a basis for the rate at which time progressed in the simulation. This meant that the program would proceed at a speed independent from the speed of the computer it ran on. The time rate was multiplied by timeMult, which allowed the simulation to be sped up, slowed down and even reversed.



```
Universe
+planets: std::vector<Planet>
-running: bool
-thinkConnection: SigC::Connection
-timeMult: double
-timeStep: double
-timer: Glib::Timer
-elapsedTime: long double
-timeIntegrationThreshold: int
+saveToFile(filename:std::string) const
+add(p:Planet)
+getRunning(): bool const
+setRunning(running:bool)
+think(): bool
+getTimeMult(): double const
+setTimeMult(timeMult:double)
+getElapsedTime(): long double const
+collideRaySphere(rStart:Vector,rDirection:Vector,sPosition:Vector,sRadius:double): bool const
-doGravity(deltaT:double)
```

Figure 3.3: Universe class diagram.

Batch mode It is up to the program itself to call think repeatedly. Each time it is called, time progresses by a fixed amount. The smaller this amount, the more accurate the simulation, but the longer it takes for the simulation time to progress the same amount. The procedure calling think can read the positions from the planets vector after each iteration and output results in a useful format, for instance Gnuplot data as shown on page 37.

The constructor for Universe took a parameter, timeStep that determined which mode the Universe used. If this parameter was non-zero then the simulation ran in batch mode, and the parameter was used as the time advancement rate (per call to think, in seconds). Otherwise, the simulation ran in interactive mode.

The Planets themselves were stored in a std::vector because they were accessed more often than they were updated, and the vector data structure offered a constant-time access method, in exchange for insertion in linear time.

Planets were added to a Universe with the add method. This was called at the user's request, when adding a planet, but was mainly done when a simulation state was loaded.

Universe handled the task of saving a simulation state to disk by iterating over the contents of the planets collection, creating a node in the DOM tree for each one it found.

collideRaySphere calculated the intersection of a ray with a sphere. It was used to work out which planet the user had clicked on during the editing of the Universe.

getElapsedTime simply returned the total running time of the simulation, in simulated seconds. It was used by the MainWindow class to display the status of the simulation to the user.

3.3.3 View classes

Scene

A Scene (Figure 3.4) was a Gtk widget. That is, it could be placed in a Gtk window just like a button or textbox. A reference to the universe object was provided to it upon construction, so it was possible for a single window to create several Scenes and have them observe different aspects of the same simulation.

Scene provided the user with a view of the state of the Universe. It drew the planets and provided a number of methods for enhancing the clarity of the display.

For example, it can be seen from Table B.1 that the planets could not be drawn with their widths at the same scale as used to place them in their orbits; if the orbit of the Earth ($149,600,000km$) was drawn with a width of $20cm$, then an image of the Earth itself at the same scale would have a diameter of just $1.71 \times 10^{-8} meters$—far smaller than the width of a pixel. Therefore Scene has the ability to exaggerate the apparent sizes of the planets; exaggerateMode is an enumeration of possible methods to use.

on_idle was the method used to cause the Scene to redraw. It was connected to Glib's idle signal so that a Scene would be continually redrawn. See Page 29 for details.

zoomDistance stored the field of view for the scene. Its default value was 2.5×10^{11} m, which allowed a Scene to display all the planets in our inner solar system (the outermost being Mars) without the user having to zoom in or out.

A Scene could adjust its view so that it followed the progress of a particular planet. The methods followNextPlanet and followPreviousPlanet corresponded with the relevant buttons in the main window.

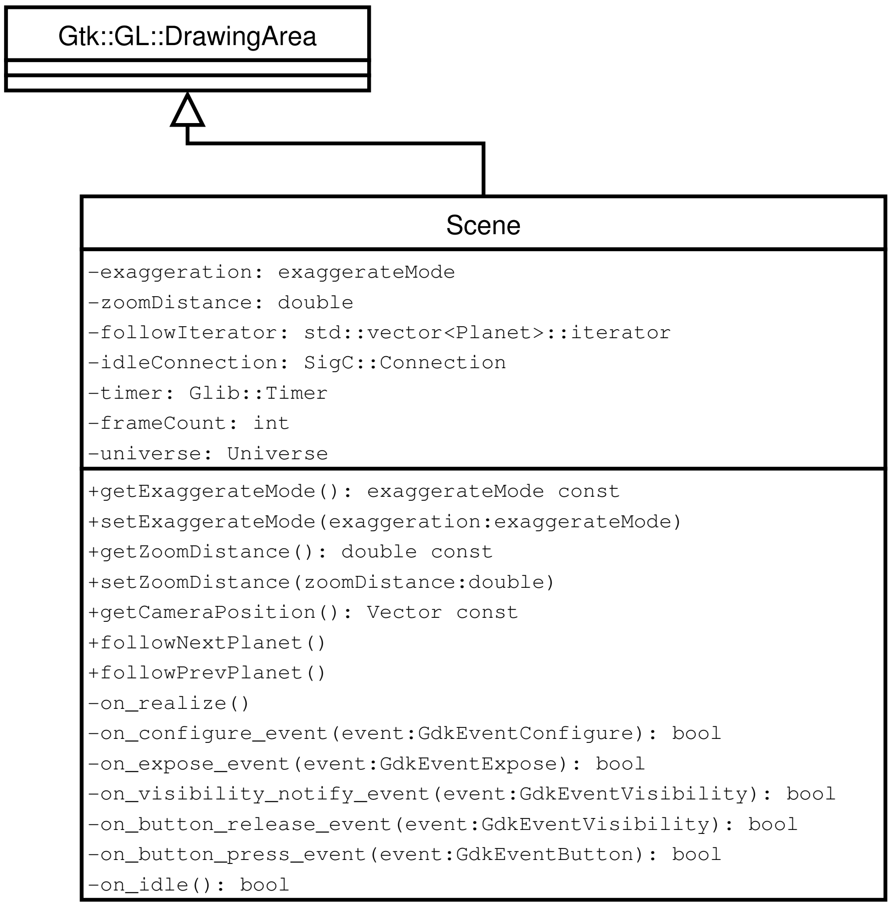

Gtk::GL::DrawingArea
Scene
-exaggeration: exaggerateMode
-zoomDistance: double
-followIterator: std::vector<Planet>::iterator
-idleConnection: SigC::Connection
-timer: Glib::Timer
-frameCount: int
-universe: Universe
+getExaggerateMode(): exaggerateMode const
+setExaggerateMode(exaggeration:exaggerateMode)
+getZoomDistance(): double const
+setZoomDistance(zoomDistance:double)
+getCameraPosition(): Vector const
+followNextPlanet()
+followPrevPlanet()
-on_realize()
-on_configure_event(event:GdkEventConfigure): bool
-on_expose_event(event:GdkEventExpose): bool
-on_visibility_notify_event(event:GdkEventVisibility): bool
-on_button_release_event(event:GdkEventVisibility): bool
-on_button_press_event(event:GdkEventButton): bool
-on_idle(): bool

Figure 3.4: Scene class diagram.

3.3.4 Controller classes

The interface mock ups in this section were created with GLADE, a GTK+ user interface design program.

MainWindow

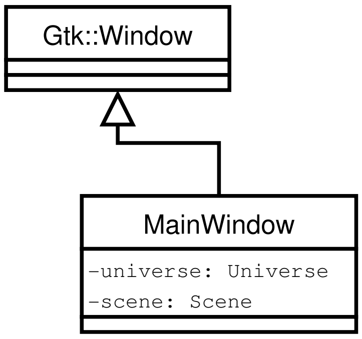


Figure 3.5: MainWindow class diagram.

MainWindow was a GTK+ window that handled interaction from the user, calling appropriate methods of its scene and universe such as setting the interactive simulation speed. See Figure 3.6.

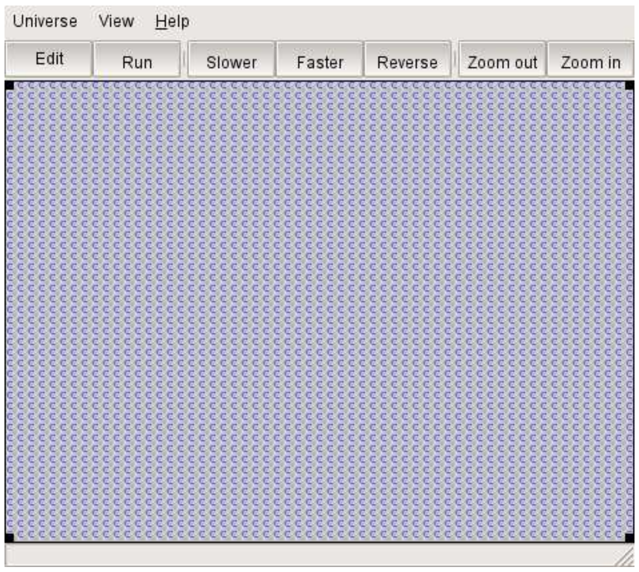


Figure 3.6: MainWindow mock up.

At the top of the window was a menu bar which allowed the user to save the state of universe to disk; add new planets to the universe; and adjust display parameters, such as the planet chosen for Scene to follow, and the method to use for the exaggeration of planet sizes.

Below this is a toolbar, giving quick and easy access to commonly used functions. The first two buttons on the toolbar were toggle buttons—only one can be pressed at a time. By changing the state of these buttons, the user could enter Edit or Run mode–this is the mechanism by which the simulation is stopped and started. The other buttons were to control the simulation time rate and the Scene zoom level.

Underneath was the scene control. When the program was in Edit mode, the user could click on a planet to print up a PlanetWindow that would let them edit the properties of their chosen planet. The scene control did not interact with the user in any other way.

Finally there was a status bar at the bottom of the window. This was used to display the simulation's elapsed time and current time rate, as a multiple of real time.

PlanetWindow

When the user clicked on a planet in Edit mode, he was presented with the PlanetWindow dialog. See Figure 3.7.

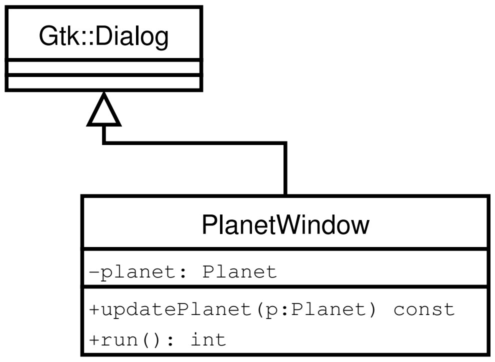
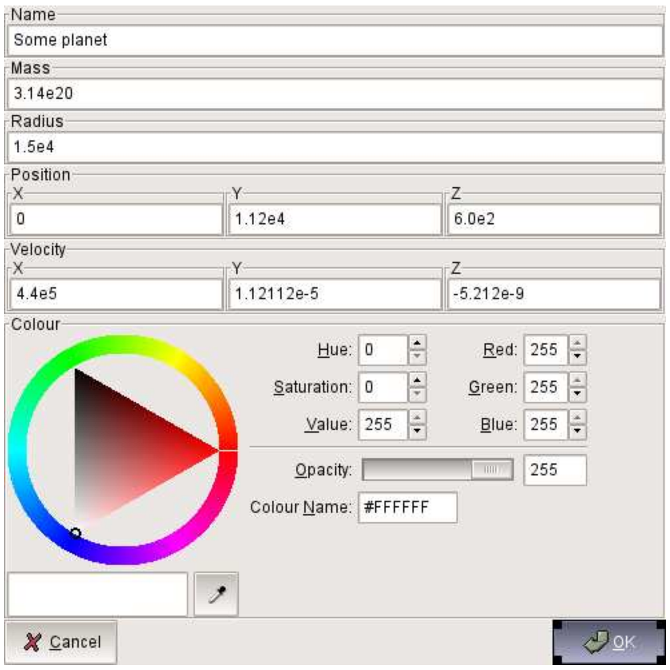


Figure 3.7: PlanetWindow class diagram.

PlanetWindow was constructed with a reference to a Planet. When it was displayed by calling the run method, it updated its controls to match the state of the selected Planet. The dialog would then be displayed and the user would be able to edit the properties of the planet.

Since the number of controls in the window was quite large, it was decided that making them all available to the caller, so that the caller would update



Name: Some planet
Mass: 3.14e20
Radius: 1.5e4
Position: X: 0 Y: 1.12e4 Z: 6.0e2
Velocity: X: 4.4e5 Y: 1.12112e-5 Z: -5.212e-9
Colour:
Hue: 0 Red: 255
Saturation: 0 Green: 255
Value: 255 Blue: 255
Opacity: 255
Colour Name: #FFFFFF

Figure 3.8: PlanetWindow mock up.

the Planet itself, would be tedious and error prone. Therefore, the decision was made to create an updatePlanet method, which would alter the properties of the specified Planet to match the state of the controls in the PlanetWindow. The caller would make this call conditional on run returning a non-zero value, signifying that the user pressed the "Ok" button.

3.3.5 Modules

The remaining modules were simple, being concerned with setting up the controller class in a normal run of the program, performing tests of the Model classes and converting various data types to strings for display.

main Entry point for the program. Also used to perform tests on the Model classes.

toString Template that converted a variable into a std::string.

Chapter 4

Implementation

4.1 Calculating gravity

4.1.1 The force between two planets

Planet::getGravity was an instance method that calculated the force of attraction between "this planet", p and another planet, q. It returned a Vector containing the x, y and z components of the force.

It began by subtracting the position of q from that of p. This means that $\vec{F}$ would ultimatly represent the effect of the other planet on "this" planet, rather than the other way around.

$$\vec{r} = \vec{p}_p - \vec{p}_q$$

Equation 3.1 was then used to calculate the magnitude of the force between the planets.

$$f = \frac{Gm_p m_q}{|r|^2}$$

Trigonometry was used to work out the components of the final answer in the direction of each of the world axes. For example, The x component is calculated as in Figure 4.1.

$$F_x = f \times \cos\theta \qquad F_x = f \times \cos\phi \qquad F_x = f \times \cos\psi$$

Where θ is the angle between $\vec{r}$ and the x axis, ϕ is the angle between $\vec{r}$ and the y axis and ψ is the angle between $\vec{r}$ and the z axis.

Values for $\cos\theta$, $\cos\phi$ and $\cos\psi$ were calculated by taking the dot product of $\widehat{r}$ and the corresponding axis vector (for example, $[1, 0, 0]$ for the x axis).

The result was the combination of the components:

$$\vec{F} = [F_x, F_y, F_z]$$

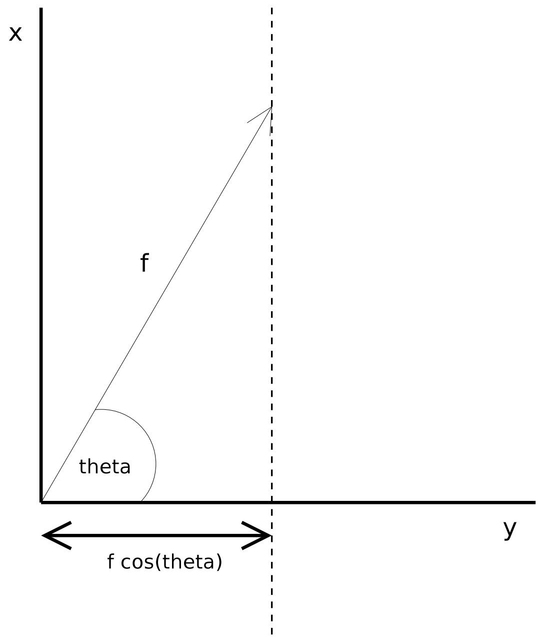


Figure 4.1: Calculating the x component of a vector.

4.1.2 Moving the planets; updating the universe

The Universe::think method was used to calculate the combined effect of all the planets on each other, using this information to calculate the planet's new positions. It worked like this:

1. If running in batch mode, set *deltaT* to the fixed time step value (e.g. 1 hour). Otherwise, set *deltaT* to the time elapsed between this call to think and the last one, multiplied by the time rate factor.
2. Create an empty vector, c' to hold copies of updated planets.
3. For each planet in the universe, p...
    (a) Set the accumulated force on p, $\vec{F} = [0, 0, 0]$.
    (b) Set m to the mass of p.
    (c) For each planet in the universe, q...
        i. If $p \neq q$ then add p.getGravity(q) to $\vec{F}$.
    (d) Set $\vec{a}$ (acceleration) to equal $\frac{\vec{F}}{m}$.
    (e) Copy p into c' to create p'.

   (f) Update the velocity of p', $\vec{v}$ to equal $\vec{a} \times deltaT$.

   (g) Update the position of p' to equal its old value added to $\vec{v} \times deltaT$.

4. Overwrite the old planets collection, c with the contents of c'.

5. Iterate through c, subtracting the position of the first planet in c from the position of each planet found. This is done in order to ensure that the solar system remains centred about the world origin.

This approach worked very well as long as the value of *deltaT* did not become too large; this would happen when the user set the simulation to run at 1,000,000 its normal rate. It broke down at this point because the time step became to coarse for the above method to generate an accurate result.

It was decided to devise an algorithm that divided up large chunks of time into many smaller chunks, processing a set of gravity calculations for each one. The chunk size was known as the *time integration threshold*, because the algorithm approximated the process of integrating *deltaT* down to an infinitely small size for each set of calculations.

The algorithm ensured that no matter how high the user set the time rate multiplier, the universe's internal time step would never rise above the value of the *time integration threshold*. The threshold was set to 12 hours by default.

1. At the start of think, set *partialDeltaT* to the *time integration threshold*.

2. While $deltaT > partialDeltaT$

   (a) Subtract *partialDeltaT* from *deltaT*

   (b) Perform universe gravity calculations as above, using *partialDelta* as the time step.

3. Perform the above gravity calculations, using *deltaT* as the time step.

This optimisation was not performed when the simulation is in batch mode, because the purpose of batch mode was to produce more accurate results, at the price of a slower running simulation.

4.2 Interaction

4.2.1 Making the display interesting

As explained in Section 3.3.3, if all the objects in a typical solar system were drawn to scale, the scene would seem very empty. It was necessary to provide visual aids so that the user would be able to see the planets, even while at a zoom level at which they would normally be invisible.

Four methods were implemented:

Follow planet The `Scene` class maintained a reference to a member of the `planets` collection. Before each frame was drawn, `Scene` would move the camera position so that the user's view of the universe was centred on a particular planet. This gave the illusion of the camera following a planet as it travelled through the universe.

Exaggerate planet radius `Scene` would dramatically increase the radius of all the planets it drew. While this did made the planets visible from large zoom distances, it resulted in a Sun that appeared to engulf the entire solar system. The gas giants were also far too big.

A logarithmic scaling factor was added, so that planets that were already large would have their sizes increased by a (much) smaller amount. The final result was that every planet in our solar system was visible at the default zoom level of the orbital radius of of Mars.

Planet bounding box If exaggeration was disabled, a box was drawn surrounding each planet as an indicator of the planet's position. The box remained at a constant on-screen size of $\frac{1}{128}$ of the width of the window.

Velocity indicator A line was drawn extending from the centre of each planet with a constant on-screen length of $\frac{1}{16}$ of the window width. This line pointed in the direction in which the planet was moving.

4.2.2 Fake multithreading

GTK+ depends on Glib, a low-level library that encapsulates some tedious C and C++ programming techniques into a pleasant API. One of the features used by the program was that of Signals.

Signals allow GTK+ objects to communicate with the program and with each other. For example, when a button is pressed, its `clicked` signal is fired, and any methods currently "connected" to the signal will be called.

Buried deep in the bowls of GTK+ lies the event loop, which is run continuously whenever a GTK+ window is open. The event loop is responsible for waiting on incoming events and dispatching messages that cause an object's signal to be fired. Glib provides an idle signal, which is fired whenever the event loop has no other pending messages to deal with.

This signal was used to make the program appear perform several tasks at once (updating user interface, running the simulation and redrawing scene), without having to resort to using, thereby and dealing with the enormous complexities of, multithreading. This is because methods connected to a signal are called synchronously.

The think method of Universe and the on_idle method of Scene were both connected to the idle signal.

4.2.3 User interaction with Scene

While the program was in Edit mode, the user was able to click on a planet in order to alter its properties. This was an interesting problem because it involved the calculation of the intersection of a ray and a sphere.

Since it inherited from Gtk::GL::DrawingArea, which is ultimatly a GTK+ control, Scene had the ability to detect mouse presses within its on-screen area. When a Scene was clicked on, the button_clicked signal was fired and on_button_release was called.

The first thing this method did was convert the x and y co-ordinates of the click, measured in screen pixels, into the units used by Universe. This was done by scaling the values from the range $[0 \rightarrow sceneWidth]$ to $[-zoomWidth \rightarrow zoomWidth]$[1].

The values of x and y then represented an infinitely long ray that extended through the Universe. The method called Universe::collideRaySphere once for each planet in the universe, to determine which planets, if any, the ray passed through. This was done as follows[Hub]:

Define the ray as a starting point, and a unit vector to represent the direction:

$$R_0 = [X_0, Y_0, Z_0] \qquad R_d = [X_d, \widehat{Y_d}, Z_d]$$

then we can work out all points in the ray as follows, where $t > 0$.

$$R_t = R_0 + R_d \times t$$

[1]$zoomWidth$ is the distance from the centre of the universe to the edges of the scene.

A sphere is defined by a centre point and radius.

$$S_c = [X_c, Y_c, Z_c] \qquad S_r$$

The sphere has the following set of points on its surface.

$$[X_s, X_y, X_z]$$

The following equation implicitly defines all points on the surface of the sphere.

$$(S_r)^2 = (X_s - X_c)^2 + (Y_s - Y_c)^2 + (Z_s - Z_c)^2$$

We can express the ray as $X = X_0 + X_d t$, $Y = Y_0 + Y_d t$, $Z = Z_0 + Z_d t$ and replace X_s, X_y, X_z in the equation for the sphere with these values.

$$(S_r)^2 = ((X_0 + X_d t) - X_c)^2 + ((Y_0 + Y_d t) - Y_c)^2 + ((Z_0 + Z_d t) - Z_c)^2$$

If we then express this equation in terms of t, we get $At^2 + Bt + C = 0$, where:

$$A = (X_d)^2 + (Y_d)^2 + (Z_d)^2 = 1$$

$$B = 2(X_d(X_0 - X_c) + Y_d(Y_0 - Y_c) + Z_d(Z_0 - Z_c))$$

$$C = (X_0 - Xc)^2 + (Y_0 - Y_c)^2 + (Z_0 - Z_c)^2 - (S_r)^2$$

$A = 1$, because the ray direction vector R_d is a unit vector.

Solving the quadratic equation would reveal the values of t that cause the ray to intersect the sphere; however since we are only interested in *whether* the two intersect, we only need to calculate the *discriminant* of the quadratic equation. If $b^2 - 4ac \geq 0$, then the ray intersects with the sphere in at least one place.

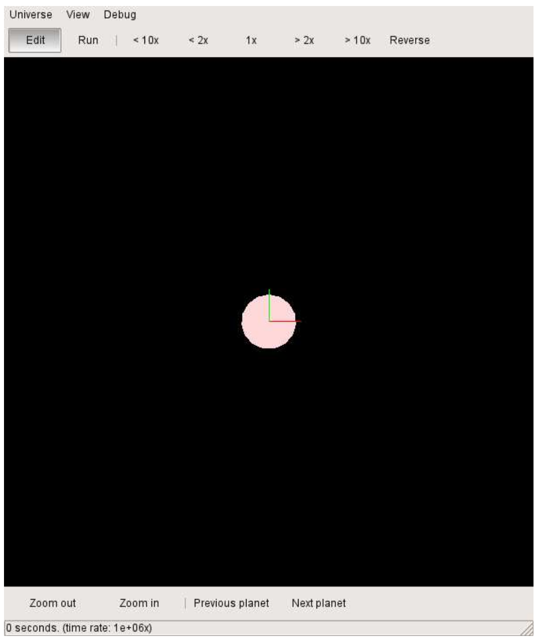
on_button_release would then display a PlanetWindow for every planet that the ray passes through,

Chapter 5

Results

5.1 A simple solar system

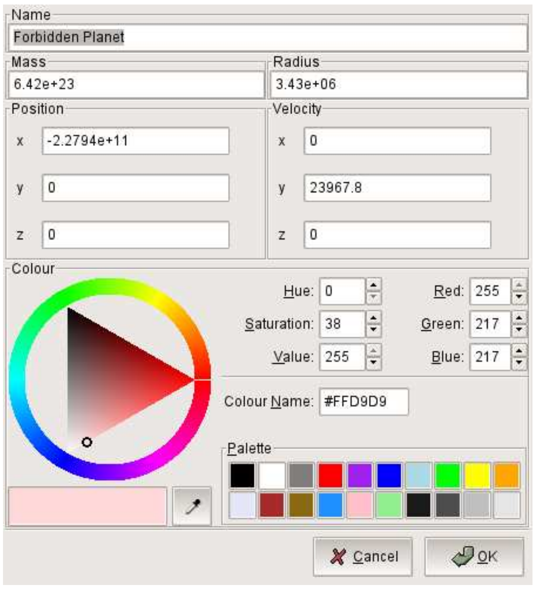
When the program is started, the user is presented with an empty universe. After selecting Add planet from the Universe menu, a default planet is placed in the centre of the universe.



0 seconds. (time rate: 1e+06x)

Figure 5.1: A single, default planet.

Clicking on the planet opens up a dialog box that allows the user to adjust the planet's name, mass and other properties (see Figure 5.2).


Name: Forbidden Planet

Mass: 6.42e+23

Radius: 3.43e+06

Position: x -2.2794e+11, y 0, z 0

Velocity: x 0, y 23967.8, z 0

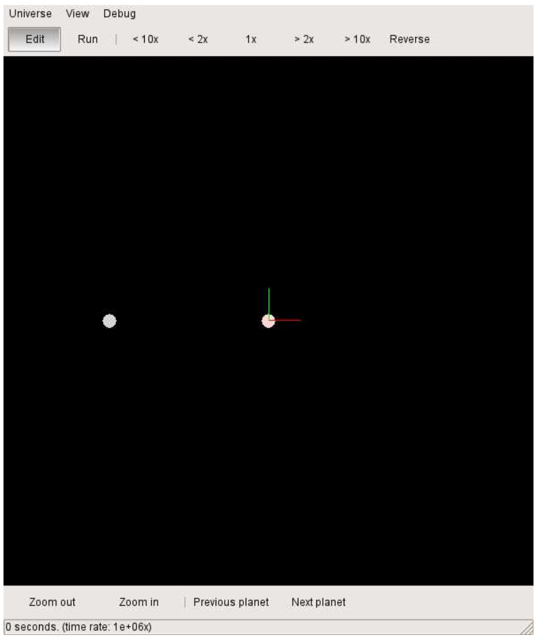
Colour

Hue: 0
Saturation: 38
Value: 255
Red: 255
Green: 217
Blue: 217

Colour Name: #FFD9D9

Figure 5.2: Editing the properties of the planet.

Another planet is then added.



0 seconds. (time rate: 1e+06x)

Figure 5.3: A new planet is added.

This planet is made into the system's sun. The simulation does not distinguish between suns, planets, moons and other heavenly bodies.

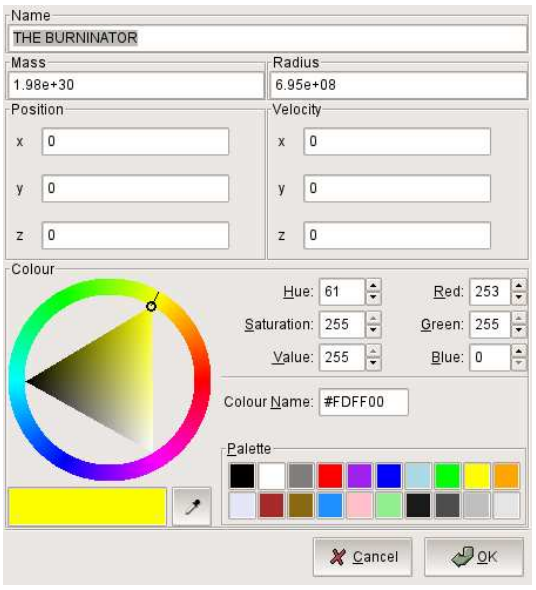

Name: THE BURNINATOR
Mass: 1.98e+30
Radius: 6.95e+08
Position: x 0, y 0, z 0
Velocity: x 0, y 0, z 0
Colour:
Hue: 61
Saturation: 255
Value: 255
Red: 253
Green: 255
Blue: 0
Colour Name: #FDFF00

Figure 5.4: The planet becomes a sun.

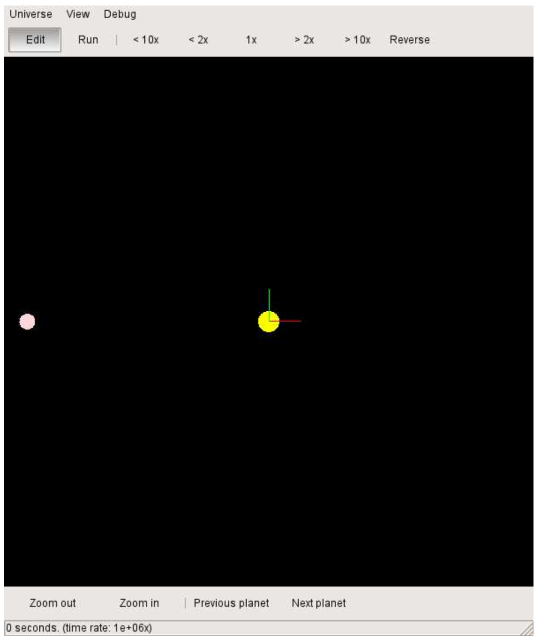
The velocity of the planet ($24000 m/s$) was calculated using the formula for the velocity of a circular orbit: $v = \sqrt{\frac{GM}{r}}$. The system is now ready to be run! (Figure 5.5)

The simulation is set into motion upon the click of the Run button. The time rate can be adjusted in factors of 2 or 10 by pressing the appropriate toolbar buttons.

After a few years of simulation time, it looks like Figure 5.6. Note the line sticking out of the planet. This indicates the direction it is heading in.

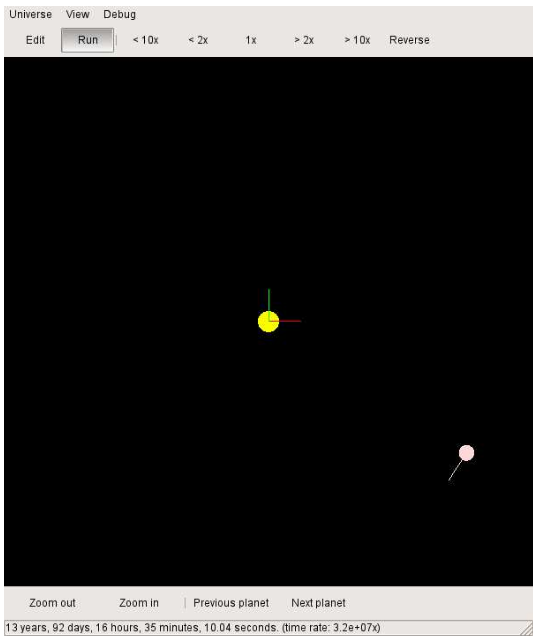
The Save item in the Universe menu allows the universe to be written to a file. The universe is serialised to XML, like this:

```
<?xml version="1.0"?>
<solar>
    <planets>
        <planet name="THE BURNINATOR" radius="6.95e+08" mass="1.98e+30">
                <colour x="0.991272" y="1" z="0"/>
                <position x="0" y="0" z="0"/>
                <velocity x="0" y="0" z="0"/>
        </planet>
```



0 seconds. (time rate: 1e+06x)

Figure 5.5: The system has been set up.



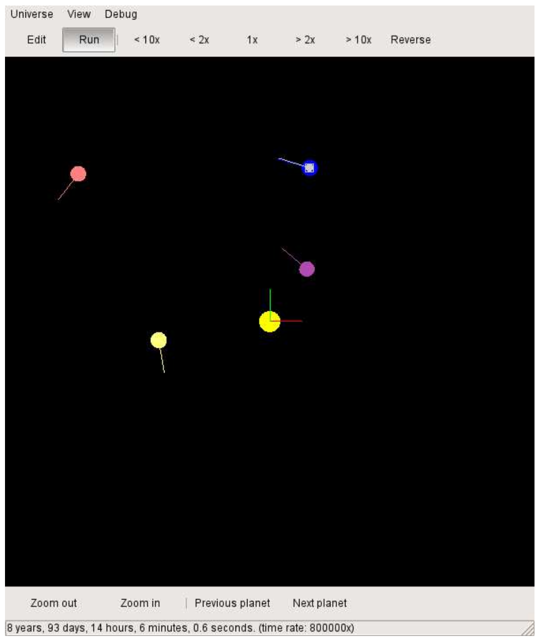
13 years, 92 days, 16 hours, 35 minutes, 10.04 seconds. (time rate: 3.2e+07x)

Figure 5.6: The system, a few simulated years later.

```
        <planet name="Forbidden Planet" radius="3.43e+06" mass="6.42e+23">
                <colour x="1" y="0.850828" z="0.850828"/>
                <position x="-2.2794e+11" y="0" z="0"/>
                <velocity x="0" y="23967.8" z="0"/>
        </planet>
    </planets>
</solar>
```

5.2 More complex systems

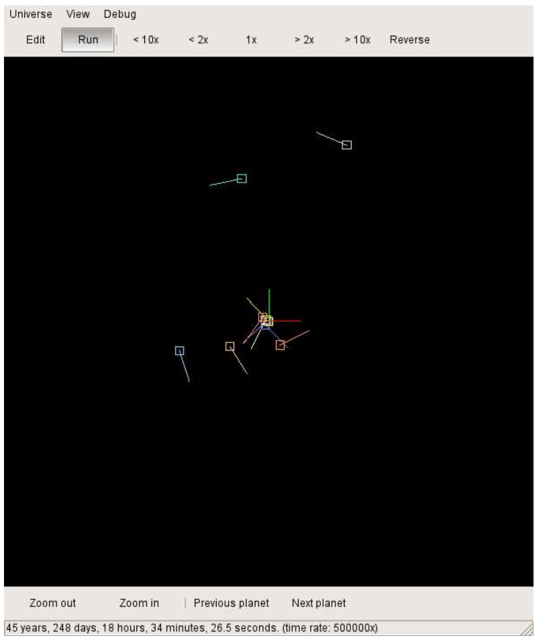
It is possible to create much more complicated solar systems. Figure 5.7 shows our inner solar system. Note that the Earth appears to have engulfed the Moon. This is because the positions of the planets are drawn to scale, but their sizes have been exaggerated.



8 years, 93 days, 14 hours, 6 minutes, 0.6 seconds. (time rate: 800000x)

Figure 5.7: The four innermost planets of our own solar system.

Figure 5.8 shows a model of our entire solar system. The exaggeration of planet sizes has been disabled from the View menu, so the boxes used to indicate planet positions are visible.



45 years, 248 days, 18 hours, 34 minutes, 26.5 seconds. (time rate: 500000x)

Figure 5.8: Our complete solar system.

5.3 Batch mode

When running the program with the `-b` option, the specified universe file is loaded and run in batch mode for a simulated ten years at a time step of six hours. The output is of the following form:

```
alpha -1.49998e+11 5.54342e+08 1.27088e-10
beta 1.49994e+11 -1.10868e+09 -1.1438e-09
alpha -1.49994e+11 1.10868e+09 3.81265e-10
beta 1.49988e+11 -1.663e+09 -2.2876e-09
alpha -1.49988e+11 1.663e+09 7.62534e-10
beta 1.49979e+11 -2.21729e+09 -3.81269e-09
```

This output can easily be processed by a shell script and the results plotted by Gnuplot[gnu].

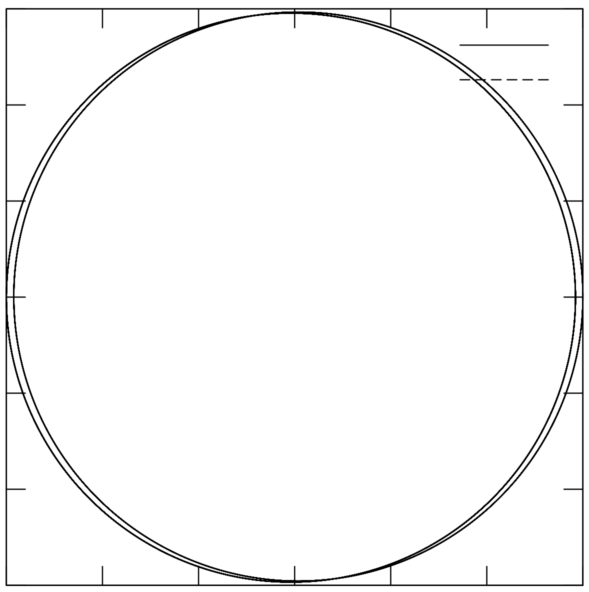


Figure 5.9: Plot of the orbits of a binary star system.

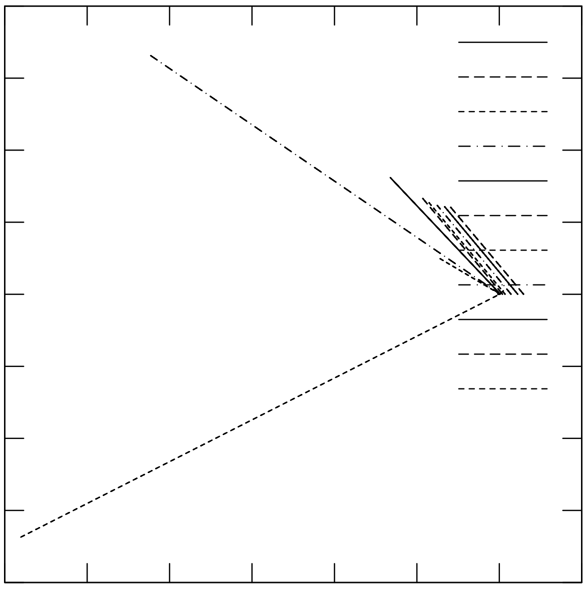


Figure 5.10: A rogue planet with a large mass enters our solar system. The extra velocity it gives to the planets allow them to escape the pull of the Sun's gravity. The interloper continues almost unaffected on its south-westerly journey, whilst one of the planets in our solar system was accelerated to escape velocity.

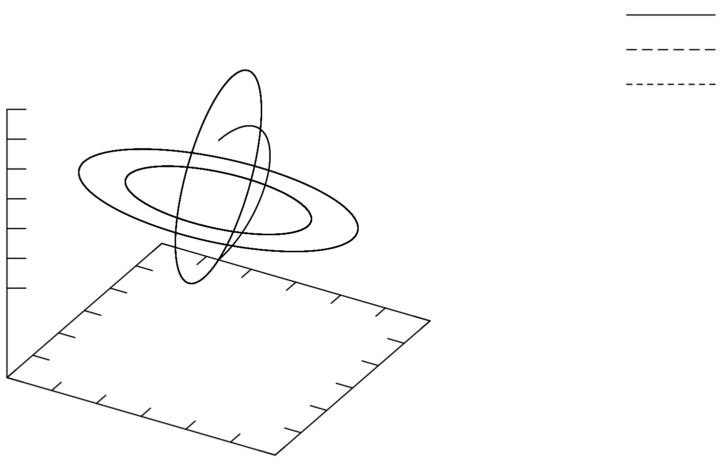


Figure 5.11: Three dimensional plot of a highly ornamental system. Three sets of Earth-like masses orbit a sun on two planes.

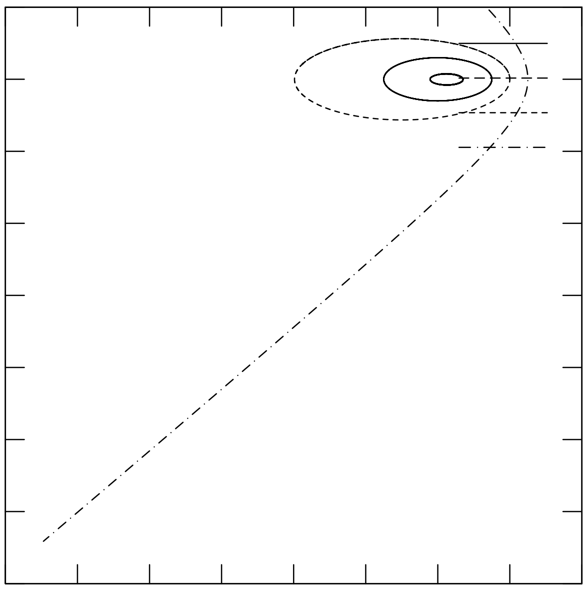


Figure 5.12: A demonstration of elliptical, circular and parabolic orbits. The outermost planet has attained escape velocity by a small margin.

Chapter 6

Testing and evaluation

6.1 Strategy

The majority of the testing that the program underwent occurred during development. The classes described in Chapter 3 were developed separate from one another. A test harness was constructed for each class, which was used to perform a number of test suites when implementing new features. This methodology made it possible to be reasonably certain that the individual components of the program were working correctly before they were integrated together.

If a bug condition was discovered during integration, a new test was added to the test suite for that class, so that any future regressions of the bug fix would be detected and repaired quickly.

6.2 Test suite output

This is a portion of the output from the test suite for Vector::getPhi. A series of vectors are created, anchored to the origin but rotating about the y axis at ten degree intervals. The method is called for each of these objects. The correct result would be angle values starting at 0 degrees and going up to 350 degrees before going back to 0.

```
{0, 0, 10}; psi = 0 degrees
{1.73648, 0, 9.84808}; psi = 10 degrees
{3.4202, 0, 9.39693}; psi = 20 degrees
{5, 0, 8.66025}; psi = 30 degrees
{6.42788, 0, 7.66044}; psi = 40 degrees
{7.66044, 0, 6.42788}; psi = 50 degrees
```

```
{8.66025, 0, 5}; psi = 60 degrees
{9.39693, 0, 3.4202}; psi = 70 degrees
{9.84808, 0, 1.73648}; psi = 80 degrees
{10, 0, 6.12303e-16}; psi = 90 degrees
...
```

The following is the test output for the Planet class.

```
Planet.
Sol {mass=1.99e+30, radius=7.1492e+07, position={0, 0, 0},
        velocity={0, 0, 0}, colour={0.5, 0.5, 0.5}}
Jupiter {mass=1.9e+27, radius=7.15e+07, position={7.78e+11, 0, 0},
        velocity={0, 0, 0}, colour={0.5, 0.5, 0.5}}
Earth {mass=5.98e+24, radius=6.38e+06, position={1.496e+11, 0, 0},
        velocity={0, 0, 0}, colour={0.5, 0.5, 0.5}}

Sol.gravity(Earth) = {3.54823e+22, 2.17259e+06, 2.17259e+06}
        |F| = 3.54823e+22
Earth.gravity(Sol) = {-3.54823e+22, -6.51778e+06, -6.51778e+06}
        |F| = 3.54823e+22
Sol <> Jupiter = {4.16839e+23, 2.55232e+07, 2.55232e+07}
        |F| = 4.16839e+23
Earth <> Jupiter = {1.92001e+18, 117.563, 117.563}
        |F| = 1.92001e+18
```

The results of the calls to getGravity were compared to answers worked out manually.

6.3 Informal tests

Once the components of the program had been integrated into a single program, a model of the inner solar system was created, featuring the Sun, Mercury, Venus, Earth, the Moon and Mars. The correct velocity for the moon to maintain a steady orbit around the Earth was calculated with the formula on page 34 and the system was set in motion. The Moon stayed in its orbit around the Earth. By increasing the time multiplication rate to 1^7, it was even possible to see that the moon circled the earth 13 times for every simulation year that passed. This corresponds to the Moon's actual 28-day orbital period.

These results are far from perfect; the procedure could have been improved by running batch simulations of a model of the solar system over night and comparing it observed astronomical data. In some cases this would not be sufficient, however, as Newtonian Gravitation cannot account for some of the phenomena in our solar system (eg, the precession of the perihelion of Mercury on page 10).

6.4 User interface tests

Finally, the user interface was tested.

Start program without specifying a file to open. Expected: Window opens, displaying empty universe. Pass.

Start program specifying the path to a sample solar system. Expected: Window opens, displaying a populated system. Pass.

Select "Universe/Add planet" menu item. Expected: Default planet created at centre of universe. Pass.

Select "Universe/Save As" menu item. Expected: a file chooser dialog appears and the universe is written out to the selected file. Pass.

Click on planet in edit mode. Expected: Planet property dialog to appear when planet is clicked. Pass.

Make changes and press OK. Expected: Edited planet has attributes update to match the controls. Pass.

Make changes and press Cancel. Expected: changes are discared, edited planet is unchanged. Pass.

Run simulation. Expected: planets begin to move. Pass.

Click on planet in run mode. Expected: nothing happens. Pass.

Press "¡ 10x" button. Expected: planets slow down by a factor of 10. Time rate in status bar also decreases. Pass.

Press "¡ 2x" button. Expected: planets slow down by a factor of 2. Time rate in status bar also decreases. Pass.

Press "¿ 10x" button. Expected: planets speed up by a factor of 10. Time rate in status bar also increases. Pass.

Press "¿ 2x" button. Expected: planets speed up by a factor of 2. Time rate in status bar also increases. Pass.

Press Reverse button. Expected: planets reverse direction. Time in status bar counts backwards. Pass.

Zoom in button. Expected: view zooms in by a factor of 2. Pass.

Zoom out button. Expected: view zooms out by a factor of 2. Pass.

Previous planet. Expected: camera viewpoint changes to another planet. Pass.

Next planet. Expected: camera viewpoint changes to the original planet. Pass.

Enable "View/Don't exaggerate planet sizes" menu item. Expected: planet discs disappear, bounding boxes become visible. Pass.

Disable "View/Don't exaggerate planet sizes" menu item. Expected. planet discs reappear. Pass.

Chapter 7

Conclusions

7.1 Requirements reprise

ref	Description	Priority	Attempted
R1	Graphical simulation	High	Yes
R2	Gravity simulated	High	Yes
R3	Gravity done in three dimensions	High	Yes
R4	Interactive simulation	High	Yes
R5	Controllable viewpoint	Medium	Yes
R6	Controllable time flow	Medium	Yes
R7	Pauseable simulation	Medium	Yes
R8	Visual aids	High	Yes
R9	Editable universe	Medium	Yes
R10	Save simulation state to disk	Medium	Yes
R11	Restore simulation from disk	Medium	Yes
R12	Create interesting sample systems	Low	Yes
R13	Export data from simulation	Low	Yes
R14	Script to plot graphs of data	Low	Yes
R15	Script to create animations of data	Low	No
R16	Cross platform	Low	Yes
R17	Simple user interface	High	Yes
R18	Smooth animation	Medium	Yes
R19	Pretty graphics	Low	No

Table 7.1: Compliance with requirements.

The final program fulfilled all the original high and medium priority requirements. The two low priority requirements that were dropped were [R15] and [R19]. Although the creating the facility to export animations would have been an interesting challenge, it was not a core function of the program and so it was thought that time would be better spent elsewhere. Making the graphics pretty was not attempted for the same reason.

Although the progress of simulating gravity worked in three dimensions, the Scene only presented a top-down, orthographic projection of the universe. If time had allowed it would have been interesting to switch to a 3d perspective projection.

7.2 Learning

The process of designing and implementing the program was very informative. I have learned how to create GUI programs using the GTK+ toolkit in combination with OpenGL, and how to simulate a physics system at a speed independent from that of the computer it runs on. It was enjoyable to develop the universe processing algorithm without going in to the messy business of solving integrals and constructing differential equations.

7.3 Project plan reprise

Due to the boycott on assessment carried out by the AUT, it was necessary to push back the demonstrations by four weeks. Although this should not have affected my progress I must admit that I did let things slide by about two weeks, the result of which being that I did not have as much time to work on the unfinished requirements or this report as I had originally intended.

7.4 Suggestions for further work

Apart from the two incomplete requirements mentioned above, there is plenty of scope to improve the project:

- A more sophisticated interface for editing the universe could be provided. It might allow the user to edit the position of planets by dragging them around the window. It would also be interesting to allow the value of the Gravitational Constant to be altered. This, and other scene-related values could be written out when the simulation state is saved.

- The visual aids could benefit from text labels beside the planets. The GTK+ HTML viewer control could even be used to display information about a selected planet; the data could be transformed from the saved XML with an XSLT style sheet.

- A feature to calculate the necessary velocity of a planet for it to maintain a circular orbit, using the formula on page 34 would be a very useful addition for those trying to create their own solar systems.

Bibliography

[CB02] Seth Nickell Colin Z. Robertson Calum Benson, Adam Elman. Gnome human interface guidelines 1.0, 2002. http://developer.gnome.org/projects/gup/hig/1.0/.

[cel] Celestia: A 3d space simulator. http://www.shatters.net/celestia/.

[Cop43] Nicolaus Copernicus. De revolutionibus orbium celestium, 1543.

[deb] Debian gnu/linux – the universal operating system. http://www.debian.org/.

[Don92] W. Donahue. *Johannes Kepler New Astronomy.* Cambridge Universite Press, 1992.

[ea98] W3C DOM Working Group et. al. Document object model (dom) level 1 specification. 1998. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[ea03] OpenGL Architecture Review Board et. al. *OpenGL Programming Guide.* Addison-Wesley Pub. Co., fourth edition, 2003.

[gnu] Gnuplot homepage. http://www.gnuplot.info/.

[Gre00] Brian Greene. *The Elegant Universe: Superstrings, Hidden Dimensions and the Quest for the Ultimate Theory.* Random House, 2000.

[gtka] Gtk+ - the gimp toolkit. http://www.gtk.org/.

[gtkb] gtmm - the c++ interface for gtk+. http://www.gtkmm.org/.

[Ham] Calvin J. Hamilton. Views of the solar system. http://www.phy.bg.ac.yu/web_projects/solar/eng/datafact.htm.

[Hub] Roger Hubbold.

[jss] Jpl solar system simulator. http://space.jpl.nasa.gov/.

[kde] Kde desktop environment. http://www.kde.org/.

[Kep09] Johannes Kepler. Astronimia nova, 1609.

[Kep19] Johannes Kepler. Harmonices mundi, 1619.

[Kep21] Johannes Kepler. Epitome astronomiae, 1621.

[New87] Isaac Newton. Principia mathematica philosophiae naturalis, 1687.

[orr] Orrery: Solar system simulator. http://www.cuug.ab.ca/~kmcclary/ORRERY/index.html.

[Sag88] Carl Sagan. *Cosmos*. Random House, 1988.

[Wud98] Jose Wudka. Precession of the perihelion of mercury, 1998. http://phyun5.ucr.edu/ wudka/Physics7/Notes_www/node98.html.

[xml] The xml c parser and toolkit of gnome. http://www.xmlsoft.org/.

Appendix A

Project plan

This is the original project plan, written in October 2003.

A.1 Description and Objectives

The aim of my project is to create a "Build your own solar system" kit that lets the user create a star system out of a number of heavenly bodies, including a star and planets. The user can then set the simulation in motion to see how their star system develops (or collapses!) over time, according to the gravitational effects that the bodies exert on each another.

I will implement my project in C++, using the OpenGL library to present the interactive simulation to the user. I have not had much experience with programming in C++, so I am treating the project as a good way to learn the language. I have had experience with using OpenGL both in the second year graphics course and in my own studies, however I intend to learn how to use advanced features, such as texture mapping, to make the simulation look pretty.

Finally, I intend to learn how to use a revision control system such as CVS to manage the source code that I produce.

A.2 Features

A.2.1 Desired features

- A graphical program that simulates the interactions between many bodies due to the effects of gravity.
- Interactivity in the program; the user should be able to pause the

simulation and alter parameters at any point, as well as adjusting the rate at which time flows.

- The user should be able to save the state of the simulation to disk and restore it later. This should allow for some interesting example star systems to be distributed with the program, such as a realistic model of our inner solar system, or a model of a planet orbited by a moon, itself being orbited by another moon. Such examples could also be used to demonstrate astronomical concepts such as a binary star system, the path of a comet, or a slingshot manoeuvre.

A.2.2 Wishlist

- The user could be able to go back in time to adjust a parameter, then continue the simulation from that point for the generation of "what if?" cases.
- It would be nice to be able to export animations generated from the results of the simulation.
- The use of texture mapping to produce interesting graphics, such as the use of a real "map" of the sky, as seen from our solar system.
- The finished program should be cross-platform, at least between Linux and Windows. The use of C++, OpenGL and other cross-platform libraries such as "libsdl" and "libpng" should make this possible.

A.3 Reading list

A.4 Project plan

Proposal October 15th. This document! Contains an overview of the project and an outline of the project plan.

Research Performed before and during the design and implementation stages as required. Since it is easier to learn by doing rather than reading, I hope to knock up a few prototypes as I work out how best to simulate the interaction between planets.

Design I would like to have the overall system design finished by the 27th of October. Details can be filled in as necessary after this date, but most aspects of the operation of the system (as outlined under Features, above) should be fixed by this date in a Specification document.

Poster This leaves two weeks to prepare materials for the project poster, which is due in on November 14th…

Seminar …and the seminar, which will be between November 17th and 28th. It would be nice to have a working prototype of the simulation to present at this time.

Coding System implementation, using the prototype as a basis, will take place from around the 17th of November, up until the end of the first semester. I expect development to trail off near the end of the semester so that I can revise for my exams.

Report: first draft Should contain background information about the project, my research, and be reasonably complete with respect to events occurring in semester 1. Should be ready about the 7-15th of December.

Testing Coding will continue after the start of the second semester, which is when I intend to start formal testing of the program. Testing and coding will overlap to some extent. I will put a provisional date of early February for the end of the major part of the implementation. This leaves a month to hunt down the remaining bugs and polish the program off.

Demonstration I can also use this time to prepare for the formal demonstration of the project, which will occur around the 10-24th of March.

Final report After the demonstration I will have about three weeks to complete my report, including the analysis of how the system operates compared to the original specification. The final report is due on the 5th of May.

Appendix B

Planetary data

The following data were used to construct the example solar systems used to demonstrate the project. They were sourced from [Ham].

Name	Orbital radius ($10^3 km$)	Radius (km)	Mass (kg)
Sun	0	695,000	1.98×10^{30}
Mercury	57,910	2,440	43.30×10^{23}
Venus	108,200	6,052	4.87×10^{24}
Earth	149,600	6,378	5.98×10^{24}
Moon	384	1,737	7.35×10^{22}
Mars	227,940	3,397	6.42×10^{23}
Jupiter	778,330	71,492	1.9×10^{27}
Saturn	1,429,400	60,268	5.69×10^{26}
Uranus	2,870,990	25,559	8.69×10^{25}
Neptune	4,504,300	24,746	1.02×10^{26}
Pluto	5,913,520	1,160	1.29×10^{22}

Table B.1: Planetary data for the Sol system.