Modelling Arbitrary Spatial Objects under Gravity in a 2D-Plane

by

Edward Compton

1402754

supervised by

Associate Professor Mike Joy

Department of Computer Science

University of Warwick

2016–17

Abstract

This is a report to detail the projct to construct the application 'Gravity Lab'. The intention of this software is to construct an accessible environment to simulate and manipulate massive objects under the influence of gravity, with the aim to improve upon existing software available by increasing simulation accuracy and portability. In addition to enhancing existing simulation interfaces by constructing new features. As a part of this project, three numerical approaches to integration were implemented. The final project's simulation consisted of the BruteForce implementation of acceleration computation, and a Velocity Verlet implementation to the solution of the second order integral of the equations of motion. Additional features of the program included the prediction of future trajectories of masses, creation and editing of masses during the operation of the simulation and an inelastic rigid body collision handling system. The final program achieved all of its primary requirements and several of the extension requirements. A dicussion of the extent to which these were achieved is included at the end of the document.

Acknowledgements

I would like to express my gratitude to the people that assisted me in the completion of this project. Firstly, my supervisor Dr Mike Joy for his constructive suggestions and ideas throughout the duration of the project and for his willingness to so generously dedicate his time. Moreover, I would like to thank all those who took part in the user acceptance testing of the application, and their valuable insight into the effectiveness of the UI design.

Abbreviations

CB	Celestial Body
VV	Velocity Verlet
EE	Explicit Euler's Method
RK4	Runge Kutta's Fourth Order Method
ODE	Ordinary Differential Equation
UI	User Interface

Notations

$\dot{y}$	The first derivative of y.
$\boldsymbol{a}_i$ or $\boldsymbol{a}(t)$	The total instantanious acceleration acting upon an object i at some time t.
$\boldsymbol{v}_i$	The velocity of an object i.
$\boldsymbol{r}_i$	The position vector of an object i.
m_i	The mass of an object i.
R_i	The radius of an object i.
h	The size of the numerical step.
t	Time elapsed since the start of the simulation.

Contents

List of Figures

CHAPTER 1

Introduction

This document details the completion of the project 'Modelling Arbitrary Spatial Objects under Gravity in a 2D-Plane'. The primary focus of this project was to develop a piece of software eventually named 'Gravity Lab'. The purpose of this software is to produce a simulation environment for celestial masses that improves upon those that are already available. This document outlines the problem; details the methodology imployed to solve it; describes the processes undertaken to design and implement a solution; explains what objectives this project achieved and discusses how successful the project was and how it could be further improved. The rest of this chapter will introduce the problem of simulation software, and what was necessary to solve that problem.

1.1 Simulating

A simulation theory is the study of imitating the operation of real-world phenomena over time [3]. The behavior of these systems is defined by the development of a mathematical model and the problem of creating a mathematical model that most closely imitates the operations of reality is the fundamental issue that simulation theory tackles.

This project is particularly concerned with the simulation of a specific phenomena, gravity. Gravity is a universal property of mutual attraction between any particles that have mass [7]. Of the four fundamental forces it is by far the weakest [2]. In fact the phenomena is technically not a force at all [10], but such a discussion extends outside the scope of this project. For the purposes of this project, we consider Newtonian gravity, a minor simplification of our current understanding of gravitation, which considers gravity a force between two masses of magnitude described by Newton's law of Universal Gravitation [17].

Although on smaller scale the effects of gravity are weak, on larger scales such as the mass of stars, planets and even asteroids, it produces accelerations strong enough to hold a person to its surface or even a planet within its orbit.

The aims of this project are: to produce an effective simulation model of gravity, building upon existing techniques in the field; efficiently implement such a model and to wrap this model in an accessible interface to alter the parameters of the simulation.

1.2 Other Software

As a part of this project, a number of currently existing solutions were investigated to determine what this project's software should achieve. This section details the advantages and shortcomings of those pieces' of software.

The first program investigated was PHET's gravity and orbits. The program consists of several premade simulations of the earth, sun and a satellite, with modifiable mass, velocity and gravity. The user can click to load any of the four premade simulations and use its time menu to watch the planet's motion.

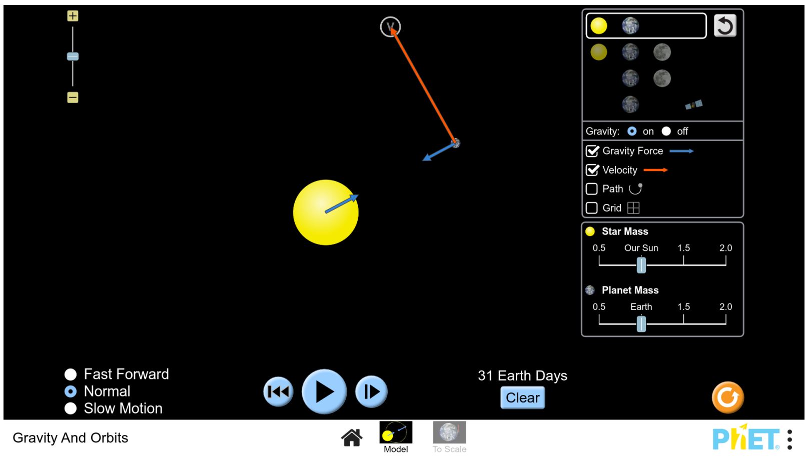


Figure 1.1: A screenshot of PHET's gravity and orbits.

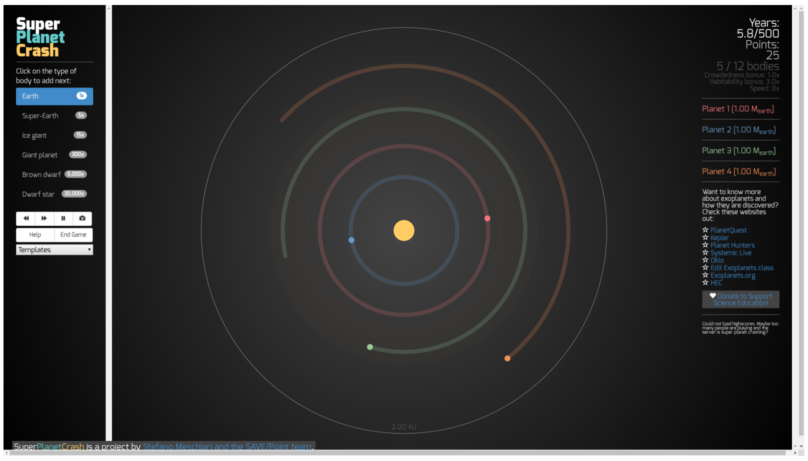


Figure 1.2: A screenshot of Stefano Meschiari's Super Planet Crash.

However, its interface is exceptionally large and clunky, covering most of the screen, this often covers the planets and makes it difficult to follow their motion. This is compounded by the fact that the program has no way to move the camera, so objects can quickly fall from the screen. Strangely, it only displays the force of gravity acting on an object and confuses the user as to why the central mass doesn't seem to be moving. To conclude, this piece of software, although somewhat easy to use, struggles with a poorly designed interface and a lack of features.

Another program found was Stefano Meschiari's Super Planet Crash. A video game styled application where you can create one of a set of masses by clicking on the screen. At first glance it appears to be very simplistic, clicking on the screen will produce a basic circular orbit. However, it does appear to have a full simulation operating in the background, by creating objects with larger masses, the orbits of all planet's appear to be perturbed. It has a simplistic collision detection algorithm, but no form of resolution, by ending the game when two masses collide.

Finally, there is a very wide selection of high level research simulations that require super computers to run, such as Caltech's recent simulation of the formation of the Milky Way. However, these programs are completely inaccessible to the general public due to their major computational requirements, making them ineffective at allowing a person equiped with just a laptop and little technical knowledge unable to make use of it.

1.3 Requirements

From this, we can infer appropriate requirements for this project to fulfil its aims. These have been broken down into simulation requirements and interface requirements.

Simulation Requirements

1. Accurately model the acceleration of one object to another due to gravity.
2. Support the creation of stable, unstable and perturbed orbits.
3. The ability to predict ahead of the simulation where planet's will be.
4. * Model basic collisions between celestial objects.

Item 1 in the simulation requirements is the most basic and necessary of these requirements. For any gravity simulator to be effective it must be able to accurate compute the motions of the planets. Item 2 is an extension to the previous requirement, that the simulation must support the addition of planets, and be accurate enough to represent the various different types of orbits that are possible in reality. This requirement will give the user access to creating a simulation with any path they want, a feature often missed in the applications inverstigated above. The third item will provide a prediction for the user, that they can use to decide how to position their planets, which reduces the problem of not knowing if your carefully crafted simulation will play out remotely how you expect before pressing play. Finally, a basic form of collision handling is necessary in this project, since without it a simulation cannot proceed effectively after a close encounter due to the Singularity problem which is discussed later in this document.

Interface Requirements

1. Visual display of the current state of the simulation.
2. Camera that can translate and zoom to better see the simulation.
3. Time menu to play, pause and adjust the speed of the simulation.
4. Tool to create new planets with specified Mass, Velocity and Position.
5. Predict the initial path of the body while selecting these values.
6. Tool to edit planet's values.
7. Information about each planet (velocity, acceleration acting upon them, etc) available by clicking.
8. * Distortion field below the plane of the Celestial Bodies.

The first requirement of the interface is a visual display of the planets, clearly it would not be an effective simulation if there was no way to see what was happening. Secondly, a camera allows the user to zoom in to see small details and drag the screen to follow any objects that might fall from it. Thirdly, a subtle menu to adjust the speed of the simulation is neccesary to start and stop the simulation. Additional faster speeds would also be a useful feature.

A key aspect of the freedom within this program is to specify exact position and velocities of the masses, so that the user is not restricted to artificially generated orbits. An expansion of this creation tool specified in interface requirement 3, is to predict the path of the planet during this process. A simple curve drawn by the renderer to predict where the object will travel would display whether or not the currently created mass would have a stable orbit once the simulation begins.

In addition to being able to create objects, it should also be possible to adjust their parameters after being created (requirement 6). This would include changes to mass and velocity, assuming collisions have been implemented. An adjuster for the radius would also be useful, since the radius on the planet will affect the collision distance.

The final two requirements are improvements to the visualisation of the simulation, the first being a display of the current acceleration acting of the planet, the acceleration providing richer information than the force might because it represents the magnitude of how its motion is changing. The second is a stretch goal, to visualise the distortions of gravity in some way beneath the plane. This could be achieved by some sort of lens effect, but it is the least vital of the requirements, as it adds very little in terms of features in comparison to the others.

CHAPTER 2

Research

As a part of this project a significant amount of research was required, the primary focus of this was in the mathematics of celestial motion and how it can be appropriately simulated.

2.1 Orbit Modelling

2.1.1 The 1-Body Problem

The 1-Body problem is the most basic form of gravitational simulation; its aim is to model the path of a single point A of negligible mass around a static point mass B. This problem was studied extensively by Johannes Kepler in the 16th century [13]. From his studies, Kepler deduced his equations of gravitational motion, showing that all movements of A around B would plot out a conic-section. He concluded that any orbit could be described by 6 constants (the Keplerian Orbital Elements) [14].

A modification of this solution can then be extended to the 2-Body problem, where both bodies in the system have non-negligible mass. This can be done by assuming that the combined centre of mass of both bodies is a static body from the first problem, then computing the paths of both bodies independently using the equations of gravitational motion from the 1-Body problem. Sadly, this cannot be extended to the 3-Body problem and the 3-Body problem in multiple dimensions remains unsolved to this day [5].

However, by ignoring the perturbations of other bodies, the 1-Body method can be used to model multiple orbits in a single system. If a single static mass is in the centre, then multiple masses of negligible comparative mass can be modelled to orbit around it, each using a Keplerian orbit [11].

Consequently, a simulation without perturbations is produced. The next step is to add the perturbations that were simplified away. Solutions to this problem come under the category of special perturbation methods, of which there are multiple, including Cowells Method, which was used to successfully predict the return of Halley's comet [9]. However, as we approach these methods the complexity of the algorithm significantly increases and the simulations still require a static central body. Perhaps a more direct approach would be more effective.

2.1.2 The N-Body Problem

The aim of a solution to the N-Body problem is to, instead of computing orbits, compute only the accelerations of gravity on the bodies. This approach can be broken down into two primary algorithms: one to compute the accelerations and another to integrate the equations of motion to find the new velocities and positions [1].

Acceleration Computation

In any instant of the simulation, every body i has a force acting upon it from every other body j according to Newton's law of gravitation shown in equation 2.1 [15].

$$\boldsymbol{F} = \frac{GMm}{|\boldsymbol{r}|^2}\hat{\boldsymbol{r}}$$

Figure 2.1: Newton's law of Universal Gravitation in Vector form.

We can equate this equation with Newtons second law of motion $\boldsymbol{F} = m\boldsymbol{a}$ to determine the acceleration $\boldsymbol{a}$ due to gravity on i from j.

$$m\boldsymbol{a} = \frac{GMm}{|\boldsymbol{r}|^2}\hat{\boldsymbol{r}}$$

$$\boldsymbol{a} = \frac{GM}{|\boldsymbol{r}|^2}\hat{\boldsymbol{r}}$$

Since the final acceleration acting on i is a sum of all other bodies in the system, the final equation becomes what is described in equation 2.2. Interestingly this final equation does not include the mass of i.

$$\boldsymbol{a}_i = \sum_{j=1}^{N} \frac{Gm_j}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3}(\boldsymbol{r}_j - \boldsymbol{r}_i)$$

Figure 2.2: Equation to find the total acceleration acting on body i.

Therefore, in each time step the value $\boldsymbol{a}_i$ must be computed for all $i < N$. Research into this project has found two methods of achieving this: Brute-Force and Barnes-Hutt. The former, just computing all values directly has time complexity $O(n^2)$ described in Algorithm 1 and the latters time complexity is actually improved to $O(nlog(n))$.

Algorithm 1 The Brute-Force method of acceleration computation.

```
for each Celestial-Body i do
    $\boldsymbol{a}_i \leftarrow (0, 0)$
    for each CelestialBody j do
        $\boldsymbol{a}_i \leftarrow pmba_i + G * m_j * (\boldsymbol{r}_j - \boldsymbol{r}_i)/|\boldsymbol{r}_j - \boldsymbol{r}_i|^3$
    end for
end for
```

Barnes-Hutt is a divide-and-conquer method that subdivides the simulation space into a quad-tree (shown in figure 2.3).The number of these quadrants is on average $\log(n)$, but this number can be significantly larger or smaller depending upon the clustering of bodies within the simulation (a simulation divided into a large number of well spread out clusters would achieve a small number of quadrants). Instead of using the centre of mass of each planet to calculate the accelerations (i.e. Brute-Force), it uses the centre of mass of the quadrants to compute the accelerations decreasing the total number of times Equation 2.2 is computed. However, this comes at a cost; by using the quadrants as the centre of mass it reduces the accuracy of the final acceleration. This compounds to the significantly increased algorithmic complexity of the program. It means that it is only appropriate in programs that can take full advantage of time-complexity. Considering that the end-user of this application will probably not be making simulations of much larger than 30 bodies, it is unlikely that the advantages of Barnes-Hutt will justify its significant time to implement and loss of accuracy [4].

2.2 Integration Methods

Once the accelerations have been computed, the new positions and velocities need to be calculated. The velocities are significant due to the conservation of momentum (a function of velocity). The total momentum of any closed system is preserved in the real world and as such should be preserved as closely as possible in the simulation.

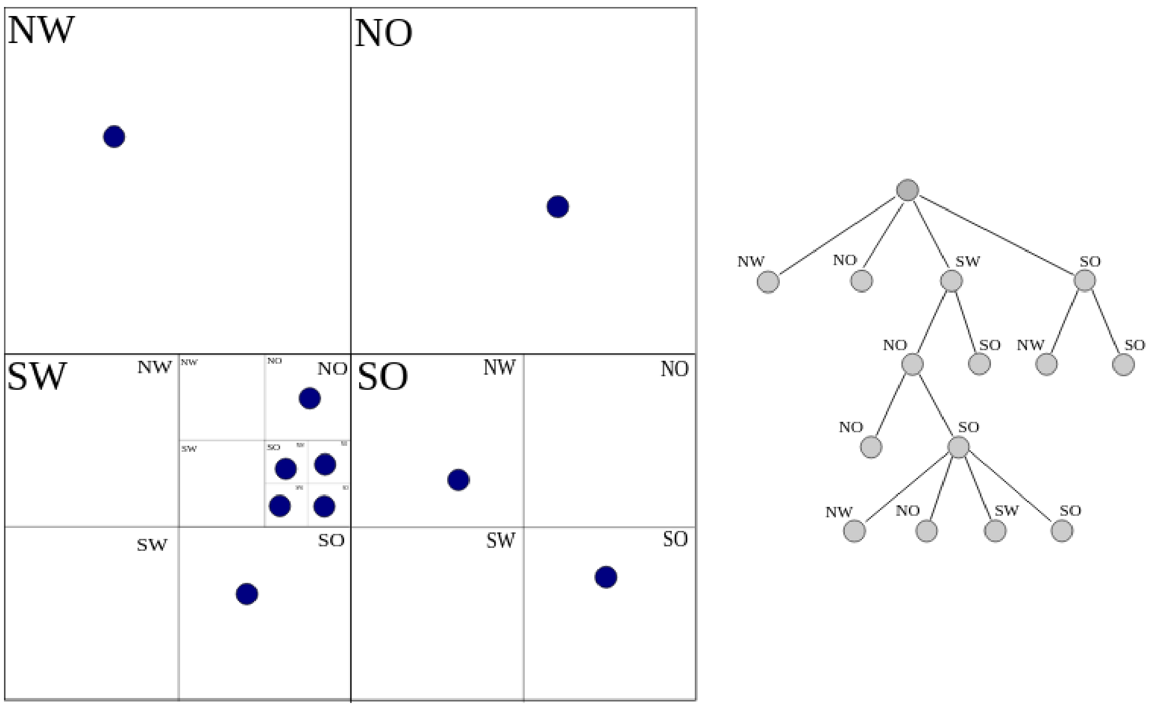


Figure 2.3: Diagram of a quadtree from the Barnes-Hut algorithm.

The new positions and velocities can be discovered by integrating to solve the equations of motion, shown below.

$$\boldsymbol{a} = \frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t}, \boldsymbol{a} = \frac{\mathrm{d}^2\boldsymbol{r}}{\mathrm{d}t^2}$$

These two equations are examples of Ordinary Differential Equations (ODEs), and since the initial starting values of the solutions $\boldsymbol{v}$ and $\boldsymbol{r}$ are known (before the simulation begins we know where the planets are and their velocity), solving these equations is an Initial Value Problem (IVF). The IVF is defined as an ODE of the form:

$$y'(t) = f(t, y(t)), y(t_0) = y_0$$

IVF solutions are typically designed to solve first order ODEs, which could be an issue because the ODE this program needs to solve is second order. However, any second order ODE $y''(t) = F(y)$ can be converted to two first order ODEs $y' = z, z' = F(y)$. Consequently, both first order and second order IVF solutions can be utilised.

Iterative integration schemes, also known as 'Numerical Methods' or 'Numerical Procedures', generate an approximation of the function by computing a set of y values for a set of values of t, with h sized steps between each value of t. They typically achieve this by working from an initial starting value and iterating along the values of t using the previous solution to compute the next one.

Such an integration scheme can be thought of as a recurrence relation, which provides a simple format to later be translated into code, as recurrence relations can be computed by iterating assignment operations. The next step from here is to find a suitable method of solving the integrals in the equations below.

$$\boldsymbol{v}_{t_0+h} = \boldsymbol{v}_{t_0} + \int_{t_0}^{t_0+h} \boldsymbol{a}(t)\mathrm{d}t$$

$$\boldsymbol{r}_{t_0+h} = \boldsymbol{r}_{t_0} + \iint_{t_0}^{t_0+h} \boldsymbol{a}(t)\mathrm{d}t$$

Three integration schemes will be applied in this project: Euler's Explicit method (often referred to as the most basic form of numerical integration); Velocity Verlet (a numerical method designed particularly for second order ODEs, to retain oscillatory stability); and Runge Kutta 4 (an expanded form of the

Euler method, that succeeds by dividing the time-steps into further segments). The following section briefly introduces the mathematics of each approach, so that they can be applied to this specific problem in the design and implementation section.

2.2.1 Euler's Explicit Method

The Euler method is a first-order numerical process to solve ordinary differential equations (ODEs) with a given initial value. It is considered the most basic explicit method for solving differential equations numerically, and is equivalent to Runge-Kutta 1. Runge-Kutta's methods are discussed slightly further along in this chapter. The method is named after Leanhard Euler, who discussed it in his book 'Institutionum calculi integralis' [12].

The defining concept of Euler's method is that within a single step of the process, the slope of the function is assumed to be constant. In a step that starts from t_0 and ends at $t_0 + h$, it hopes to compute values of y where the slope of y is some known function F of y: $y' = F(y, t)$. To iterate across this time-step, we would assume that $y' = F(t_0, y(t_0))$ for the entirety of the time-step, despite the fact that in the actual function the slope might change. Consequently, the amount at which y changes over the duration of the time-step is $hF(t_0, y(t_0))$. This results in the recurrence relation:

$$y_{t_0+h} = y_0 + hF(t_0, y(t_0))$$

2.2.2 Velocity Verlet

Velocity Verlet is a more specific form of Verlet Integration, a method of solving second order differential equations. The technique has been rediscovered multiple times throughout history, but most recently by Loup Verlet in the 1960s to be used in molecular dynamics. It has the advantage of being numerically stable under oscillatory motion, which means that when being used to compute an oscillating function (e.g an orbit), it will continue oscillating indefinitely, whilst still preserving the same time complexity as the Explicit Euler Method. In particular, in this section we will discuss the LeapFrog method, which integrates second order differentials of the form $\ddot{y} = \frac{d^2y}{dx} = F(y)$. It achieves this by forming a recurrence relation of final points, where the first $\dot{y}$ and second order y solutions are computed at interleaving boundaries of x.

Since this method solves second order equations, it has to integrate twice to find the final answer. Instead of performing this calculation along the same boundary $[x, x + h]$ for both of the integrations, the two integrations are computed a half step out of phase, the first working on the original boundary and the second being computed with $[x + \frac{h}{2}, x + 3\frac{h}{2}]$.

To show this we start with the following equation in explicit Euler form:

$$y_{x+h} = y_x + \iint_x^{x+h} F(\ddot{y}(x))dx$$

To represent it with Verlet integration, we move the integral boundaries out of phase with a second variable:

$$\dot{y}_{x+\frac{h}{2}} = \dot{y}_x + \int_x^{x+h} F(\ddot{y}(x))dx$$

$$y_{x+h} = y_x + \int_{x+\frac{h}{2}}^{x+3\frac{h}{2}} F(\dot{y}(x))dx$$

Then, within these integrals it is assumed that y and $\dot{y}$ do not vary with x, but remain constant at their value for the lower boundary.

$$\dot{y}_{x+\frac{h}{2}} = \dot{y}_x + \int_x^{x+h} F(\ddot{y}_x)dx$$

$$y_{x+h} = y_x + \int_{x+\frac{h}{2}}^{x+3\frac{h}{2}} F(\dot{y}_{x+\frac{h}{2}})dx$$

This then integrates to:

$$\dot{y}_{x+\frac{h}{2}} = \dot{y}_x + xF(\ddot{y}_x)$$

$$y_{x+h} = y_x + xF(\dot{y}_{x+\frac{h}{2}})$$

From this form of the equation, we can begin the process of applying it to the N-Body problem, which is described in Section 5.1.2.

2.2.3 Runge-Kutta 4

Developed around 1900 by C. Runge and M. W. Kutta, the Runge-Kutta methods are a family of iterative integration methods. They consist of a multitude of solutions including implicit and explicit approaches. The connection between these methods is that they all build upon the Euler Method, to perform temporal discretization for the approximate solutions of ordinary differential equations, much like the two previous integrations schemes discussed. As such, any Runge Kutta method can be considered a modified form of Explicit Euler.

In particular, we shall discuss the most widely known Runge Kutta method, the fourth order solution, also known as 'RK4', 'the classical Runge Kutta method' or simply 'the Runge Kutta method'.

The Runge Kutta 4 method works by improving upon Explicit Euler, where Euler will assume that the derivitive of the equation is constant for the entire duration of the timestep, the Runge Kutta method assumes that the function is constant for the first quarter of the timestep and compute the function again, before assuming that the new solution is constant for a quarter of the timestep. It continues this until 4 solutions of the network are found, each with a solution to the new position. The final positions are then averaged, with a weighting against the central 2 quartiles, to produce the final solution.

Runge Kutta is used to solve first order differential equations, so we shall define $\dot{y} = f(t, y), y(t_0) = y_0$ for the purposes of the explanation. The aim of this procedure is to generate a sequence of y values for each value of t in h sized intervals.

The fourth order Runge-Kutta method is defined as follows:

$$y_{t+h} = y_t + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$k_1 = f(t, y_t)$$
$$k_2 = f(t + \frac{h}{2}, y_t + \frac{h}{2}k_1)$$
$$k_3 = f(t + \frac{h}{2}, y_t + \frac{h}{2}k_2)$$
$$k_4 = f(t + h, y_t + hk_3)$$

When enhancing the method to then solve a second order differential equation, as we must do in this problem, 2 sets of k values must be defined. In this example, we refer to the first order solution as k' and the second order solution as k. To solve $\ddot{y} = f(t, y)$.

$$\dot{y}_{t+h} = \dot{y}_t + \frac{h}{6}(k'_1 + 2k'_2 + 2k'_3 + k'_4)$$
$$y_{t+h} = y_t + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

2.3 Singularities

A significant issue, present in all N-Body simulations, is the singularity problem. It stems from the assumption that is made in N-Body simulations that all objects are point-masses. When in the real world, massive objects would be made up of a collection of atoms, each with their own gravitational disturbances. When two masses are reasonably far away, this assumption has very little effect. However, when the two masses get closer, the calculated acceleration's error rate dramatically increases. This is because Newton's law of universal gravitation divides by the absolute distances between the two masses $|\boldsymbol{r}|$, as $|\boldsymbol{r}|$ approaches 0 the magnitude of the acceleration approaches infinity $|\boldsymbol{a}| \to \infty$. As is demonstrated in figure 2.4, an object's acceleration will scale infinitely as it approaches a point mass.

Consequently, every object in a simulation of this kind will act as a singularity (or black hole), causing close encounters to scale the velocity of an object massively. This is especially prominent in simulations that do not handle collisions, because the objects can get closer than the sum of the radii of the planets, further exacerbating the problem.

One solution to this problem, that is often used, is called Dampening. This is when a constant ϵ is added to the sum of the radii between the two masses, slightly altering equation 2.2 to that shown in equation 2.5. Adding this constant means that the bottom of the quotient will never fall below ϵ, preventing $|\boldsymbol{a}|$ from becoming larger than $\frac{Gm_j}{\epsilon}$. The effects of the dampening constant can be seen in Figure 2.4, which shows that the function no longer scales to infinity, but it does introduce a constant error to the acceleration, causing it to always be very slightly lower than the true Newtonian acceleration.

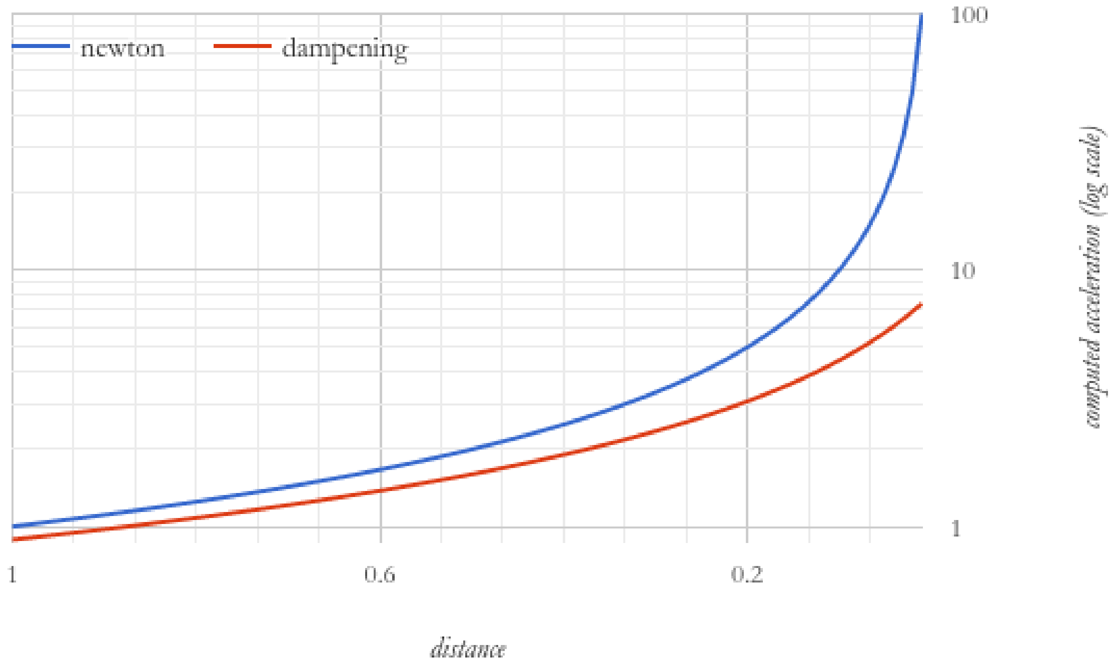


Figure 2.4: A graph to show the increase of acceleration as an object approaches a source of gravity.

$$\boldsymbol{a}_i = \sum_{j=1}^{N} \frac{Gm_j}{(\epsilon + |\boldsymbol{r}_j - \boldsymbol{i}_j|)^3}(\boldsymbol{r}_j - \boldsymbol{i}_j)$$

Figure 2.5: Acceleration equation with dampening.

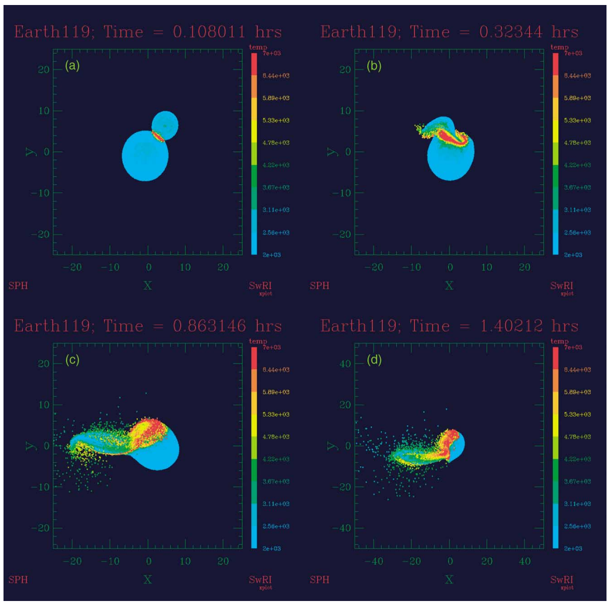


Figure 2.6: Renderings from a simulation of two large masses impacting one another. [6]

2.4 Collisions

Collisions is a stretch goal of the application, and the aim is to produce some basic collision handling. Collisions can be broken up into two components, the detection (identifying when a collision has occurred) and the resolution (computing the post collision state).

Since massive objects are clusters of atoms, in a perfect simulation between very massive objects some of these atoms break away in large clusters, thousands of new bodies to be simulated. As is seen in Figure 2.6, from Robin M. Canup's simulation of the moon formation [6]. However, even this simulation does not work on the atomic scale and it has still taken clusters of atoms working as point masses. For the purposes of this project, there is no access to large supercomputers to perform the calculations, it must be done in real time on low power machines. Therefore, further simplifications must be made.

Some examples of simplified approaches were discovered in the investigation of other applications. Firstly and most commonly the collision would not be handled at all, the collision would be detected, but not resolved, which would usually translate to the simulation finishing on a collision event. A second approach would be to assume that the resultant collided bodies merged entirely, with force due to gravity overwhelming all atoms within the system to produce a new combined mass. This approach is quite close to reality, as in these large scale collisions many of the two planet's fragments will merge to form a new mass, but it also ignores the impulse imparted from one body to another. Another approach is to assume the bodies are rigid, which also removes the need to compute fragments. This involves computing the tangential impulse vectors (computing the kinetic energy transferred in each dimension), a more complex, but also more interesting algorithmic solution. Using this method, bodies would bounce off one another, imparting energy on the collision.

2.5 Frameworks

This section details how effective the frameworks investigated are as a tool to create this piece of software. The frameworks compared are basic OpenGL, LibGDX and Unity.

Basic OpenGL has the advantage of being incredibly fast, reliable and well tested. However, it is not a game engine, but purely a graphical one. This means that it does not contain any packages that might be useful to development, putting it at a significant disadvantage compared to the other available frameworks. OpenGL is a widely adopted program, with almost all computers running a version. However, programs must be altered to work between Windows and Linux environments, and there isn't a simple deployment mechanism for web applications or Android. Although cross-platform functionality is not within the scope of this project, developing it in a framework that does support such functionality could make further work on the application in different projects much easier.

Figure 2.7: The three frameworks considered for the project.

Unity lays on the opposite end of the spectrum, it is a fully developed game engine, with numerous packages. It also incorporates multiple different languages (C#, UnityScript and Boo), although this combination of languages could increase development time. It is a proprietary application, operating on a monthly subscription service where customers can purchase access to additional features. Although there is a free tier, this could seriously hinder development if a feature required to complete the program was found to be gated off behind this tier system.

Finally, there is LibGDX, a well developed engine sitting on a java framework. The framework features cross platform deployment facilitated by its gradle wrapper, which allows it to be deployed as a desktop application, mobile app (iOS or Android), or web app. Although LibGDX's libraries are nowhere near as extensive as Unity, it consists of a wide variety of packages, including the Scene2D framework which can be used for events handling and UI rendering. Moreover, although it is written in Java, LibGDX utilises Java's new Dalvik compiler to avoid garbage collection.

CHAPTER 3

Methodology

This chapter describes the methodology employed in completing this project. It goes into detail about the project management techniques employed and breaks down the methodologies used to complete each objective.

3.1 Objectives

To facilitate the process of completing this project, it was necessary to produce a selection of objectives to complete. These objectives were broken down into Simulation, Interface and Deliverables to provide to the stakeholders of the project.

The Simulation objectives are objectives that pertain directly to the implementation of the researched simulation algorithms and the design of software in order to encapsulate them.

- Acceleration Computation
- Integration Methods
- Collisions

The approach to this collection of goals was primarily upon developing an accurate mathematical model to represent each simulated process. To develop these models, a significant amount of research was done into the underlying mathematics behind them. A focus of this research was to deduce the final model from as close to first principles as possible, in order to encourage the possibility of improvements to existing techniques. These mathematical models were developed with the knowledge that they would be implemented in a render loop based system, where the most appropriate equations would be ones that iterate upon existing values (recurrence relations), minimising the number of operations per iteration.

The interface objectives are components of the software that allow the user to be able to control the simulation and do not contain components of the simulation itself.

- Body Visualisation
- Object Creation
- Predictive Curve
- Time Menu
- Camera
- Object Editor

The focus of the completion of the interface goals was to develop isolated solutions to each of the problems. Which allows for modular development, where one item can be swapped out for a new one and they can be completed in any order. Each objective directly links to a requirement, to ensure requirements are fulfilled correctly, and none are missed during development.

The deliverables of the project consist of a sequence of reports and a presentation to document the progress and final conclusions of the project.

- Specification
- Progress Report
- Presentation
- Final Report

3.2 Legal, Social and Ethical Issues

Although this software does not contend with any major issues in these 3 fields, they have been considered. This section details the thought process of making sure that any issues are identified.

Legal Issues

There are very few possible legal issues involved in the completion of this task, all the software used to create it is open source and consists of licenses suitable for redistribution. The art assets used in the project are minimal, due to the minimalist approach to design. Those that are necessary have been created from scratch in open source software (such as GIMP), to prevent any possible legal issues.

Ethical Issues

When creating a system, that might be a user's sole source of knowledge into an educational topic, great care must be taken to make sure that this topic is not misrepresented. Consequently, significant testing into the accuracy of the simulation will be performed as a part of this project. Furthermore, the project intends to utilise user-acceptence testing to improve user experience. To reduce ethical concerns, these user acceptence tests will only be performed with close friends who have volunteered for the task.

Social Issues

Social issues concern situations where people involved in the project might have a negative experience because of it. Since the application is primarily concerned with representing strongly mathematical concepts, it is unlikely that such an event would occur to the end user. In terms of disability, colour blindness is a very common affliction that is often overlooked in development. The entire program will consist of a very limited color pallet, making sure to minimise colour based information. Moreover, it will avoid situations entirely where the distinguishment between red and green (the most common form of colourblindness) is necessary for the usage of the application.

CHAPTER 4

Project Management

In order to successfully complete a project of such a large scope, it is a necessity to utilise an effective project management strategy. The following section details how the project was managed using an appropriate software development methodology, how the schedule for the project was designed, and the management tools utilised to minimise risk.

4.1 Software Development Methodology

The opted software development methodology for this project was incremental software development. This method is from the AGILE family of software development methodologies and succeeds by being able to handle changing requirements. The incremental software development methodology in particular focuses on first creating a breakdown of all features that the final product wants to have, then ordering these features in terms of priority. The next step is to develop a working system, with the highest priority feature developed as quickly as possible. Once this product is deployable, development continues by taking a single or small group of features and adding them to the project, continuously referring to the stakeholders upon completion of each feature or small group of features.

This technique was selected for this project over other more structured approaches such as the Waterfall methodology, because the stakeholders were not fully aware of the capabilities of computation in this field when the project began. As the developer and superviser learn of what is and is not possible within N-Body simulations, the requirements must be flexible to accomodate for impossible items, but also be able to expand if new and relevant features are conceptualised.

4.2 Schedule

The most important requirement for this project is the implementation of the N-Body simulation. Although it can be expanded with different integration schemes, the initial most basic form requires a brute force algorithm, 1 integration scheme and some semblence of Body visualisation. This is the most immediate objective to be completed, and so was scheduled as such. The next step was a breakdown of the interface, ordered in priority of the usefulness of the feature (being able to create a Body is more important than editing it), before finally ending on the extension requirements, such as collisions and the distortion field. A full breakdown of the schedule in the form of the original project Gantt chart can be seen in Figure 4.1.

4.3 Development Tools

There are severel development tools that were heavily utilised within this project to lead to its success. In particular the text editor Atom, known for its strong customisability with a selection of community packages to tailor the editor to the current development environment. This made it a versatile tool

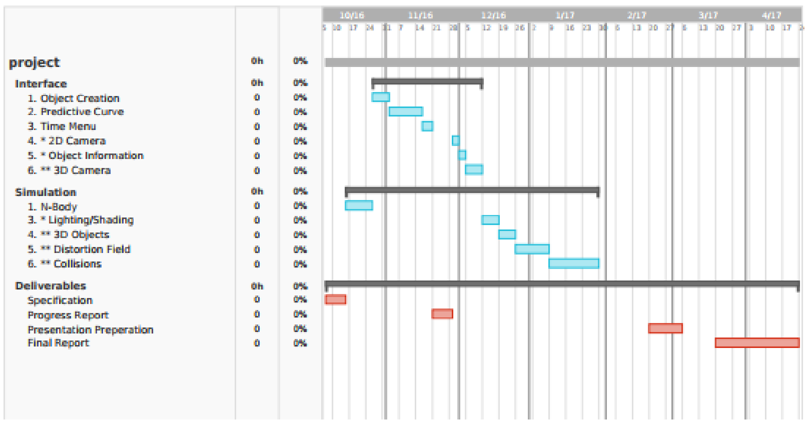


Figure 4.1: A Gantt chart displaying the planned schedule of the project's objectives.

for working in an AGILE development environment. The program itself utilised LibGDX and a Gradle wrapper as the basis of the program (discussion of LibGDX has can be found in the Frameworks section of the research). The gradle wrapper supports java dependancies through maven and can deploy the application to a minimised jar file for cross platform use. Finally, git is an excellent tool for source control, ensuring that consistant backups are maintained and provides the ability to roll back to a previous version upon code breakages.

CHAPTER 5

Design and Implementation

The overall design of this application strongly utilises LibGDX's listener interfaces. The following sections detail the completion of each objective specified in this section.

A note on LibGDX file structure: In order to support the plethora of LibGDX's supported platforms, the framework provides a comprehensive file system of classes. However, all of the primary source code is located within a single directory. The location of such code on the provided memory sticks is shown below, along with the location of a few tests discussed in the following chapter:

```
source/core/com/simulator
source/core/test/java/org/unittests/AlgorithmsTest.java
```

5.1 Simulation

Operating in an Object Oriented environment of Java, the first step of the Simulation is to define a class to describe each celestial body. Originally this was a class named CelestialBody, however, as is discussed later in this document, it was necessary to have more than one type of class to describe a body. Thus, the following interface was used:

```java
public interface Body {
    public Vector2 getR();
    public Vector2 getV();
    public float getM();
    public float getRadius();
    public void setR(Vector2 r);
    public void setV(Vector2 v);
    public void setM(float m);
    public void setA(Vector2 a);
    public Vector2 getA();
}
```

As shown in chapter 2, a point mass (body) requires 3 values to describe it in the simulation: Mass `m`, Position `r` and Velocity `v`. The acceleration vector `a` is also included, to keep track of the total acceleration acting upon a body before integrating its new position.

To encapsulate the algorithms required for the simulation, a class of static methods was created that stores the implementations. The following sections will detail the decisions made in terms of design and implementations of the methods described above.

5.1.1 Acceleration Computation

The aim of the acceleration algorithm is to compute and store each Body's acceleration to a variable `a`. The brute force method iterates through all bodies storing the solution to equation 2.2 in `a`.

```
public class Algorithm {
    static Array<? extends Body> BruteForce(Array<? extends Body> bodies);
    static Array<? extends Body> BruteForceI(Array<? extends Body> bodies, Body i);

    static Array<? extends Body> ExplicitEuler(Array<? extends Body> bodies, float dt);
    static Array<? extends Body> VelocityVerlet(Array<? extends Body> bodies, float dt);
    static Array<? extends Body> rk4(Array<? extends Body> bodies, float dt);

    static Array<? extends Body> detectCollisions(Array<? extends Body> bodies);
    static void resolveCollision(Body a, Body b);

    static float SumEnergy(Array<? extends Body> bodies);
}
```

Figure 5.1: The algorithms class, encapsulates simulation algorithms.

$$\boldsymbol{a}_i = \sum_{j=1}^{N} \frac{Gm_j}{\max(\epsilon, |\boldsymbol{r}_j - \boldsymbol{r}_i|)^3} (\boldsymbol{r}_j - \boldsymbol{r}_i)$$

Figure 5.2: Acceleration equation with improved epsilon modification.

Singularities were found to be a significant problem in the background research of this project, with a simple solution available to alleviate it. However, the method has a drawback, the distance between two planets are constantly slightly larger than it should be. Consequently, an improvement is proposed. Instead of adding ϵ to the distance, the distance could be maximum functioned against ϵ: $\max(\epsilon, |\boldsymbol{r}_j - \boldsymbol{r}_i|)$. This has the same effect as adding the epsilon constant, making the minimum distance ϵ, but with the value not being added when it is unnecessary at further distances. The resultant formula can be seen in equation 5.2.

However, another problem is also present. Both adding the ϵ constant and maxing it with the radius mean that the smallest possible distance is ϵ. As a consequence, the acceleration between any body and itself is non-zero, so another modification must be made to the equation so that the acceleration with a Body and itself is ignored from the summation. This is shown in equation 5.3. The effect of this modification can be seen in Figure 5.4. The blue line on the graph represents the true Newtonian acceleration, the red line represents the standard dampening technique, and the orange line represents the improved maxing technique. As is visible from the graph at all points, the maxing equation is closer to the true Newtonian curve, and never exceeds it. Therefore we can conclude that this is a better method of preventing the exponential increase of the equation.

Now that the final equation is complete, it can be converted into the brute force algorithm. This method is a simple traversal over all parameters for an array of Body objects. A listing of this code is shown in code snippet 5.5.

5.1.2 Integration Schemes

The next step after computing the accelerations is to integrate to find their new positions.

$$\boldsymbol{a}_i = \sum_{\substack{j=1 \\ i \neq j}}^{N} \frac{Gm_j}{\max(\epsilon, |\boldsymbol{r}_j - \boldsymbol{r}_i|)^3} (\boldsymbol{r}_j - \boldsymbol{r}_i)$$

Figure 5.3: Acceleration equation with improved epsilon modification and summation condition.

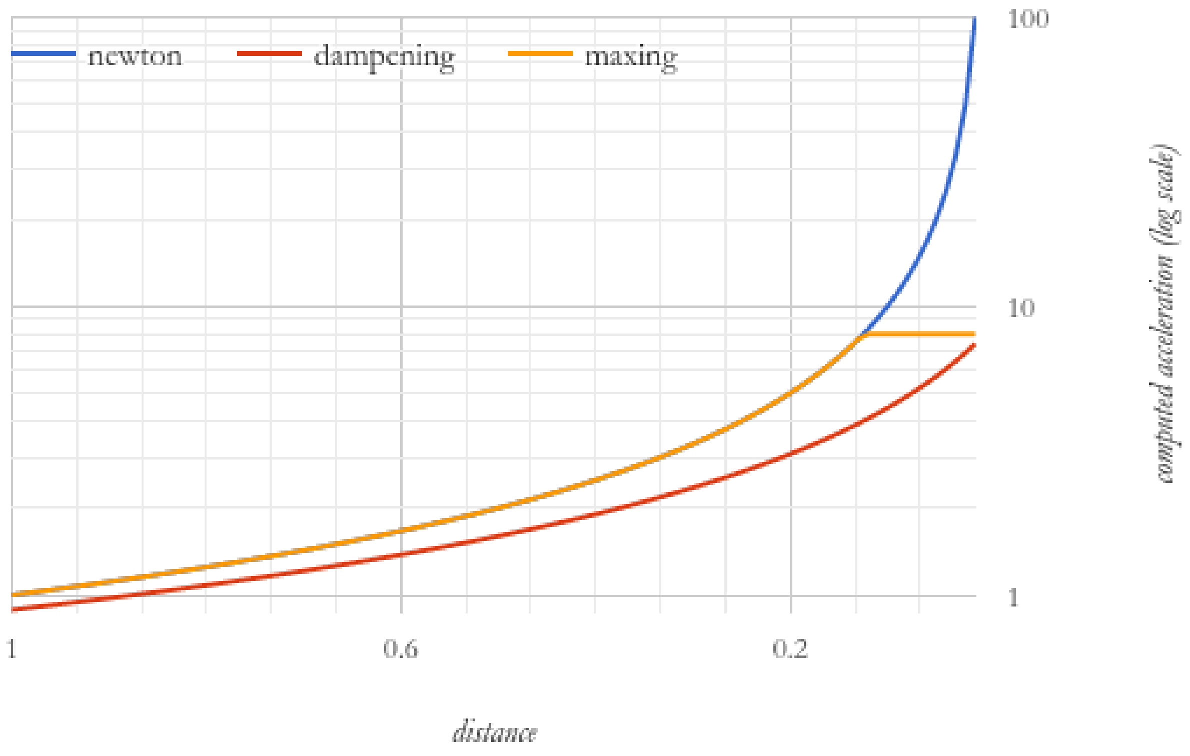


Figure 5.4: A graph to show the difference in the three acceleration equations: Newton, Dampening and Maxing.

```
static Array<? extends Body> BruteForce(Array<? extends Body> bodies) {
    for(Body i:bodies) {
        i.setA(new Vector2());
        for(int k = 0; k < bodies.size; k++) {
            Body j = bodies.get(k);
            if (!j.equals(i)) {
                Vector2 R = j.getR().sub(i.getR());
                float len = Math.max(R.len(),0.125f);
                float len3 = len * len * len;
                float s = j.getM() / len3;
                R.scl(s);
                i.setA(R);
            }
        }
    }
    return bodies;
}
```

Figure 5.5: Code snippet of the Brute Force method.

Explicit Euler

With a numerical method, as opposed to an analytic one, the solution is always an approximation to the actual value. The approximation in this instance of usage of Euler's Explicit method is that the acceleration does not depend upon time. By making this assumption the equations can then be integrated to:

$$\boldsymbol{v}_{t_0+h} = \boldsymbol{v}_{t_0} + \Big[t\boldsymbol{a}_0\Big]_{t_0}^{t_0+h}$$
$$\boldsymbol{r}_{t_0+h} = \boldsymbol{r}_{t_0} + \left[\frac{1}{2}t^2\boldsymbol{a}_0\right]_{t_0}^{t_0+h}$$

This can then be simplified to a form that can be converted into code.

$$\boldsymbol{v}_{t_0+h} = \boldsymbol{v}_{t_0} + h\boldsymbol{a}_0$$
$$\boldsymbol{r}_{t_0+h} = \boldsymbol{r}_{t_0} + \frac{1}{2}h^2\boldsymbol{a}_0$$

From these two equations, the Explicit Euler method can easily be converted to an algorithm operating in linear time. By taking in two parameters `Array<? extends Body> bodies` and `float dt`, calls the BruteForce method first to load up each `Body` with its acceleration vector $\boldsymbol{a}(t_0)$. It then proceeds to compute the Euler integral of the two velocities. The implementation of this in Java utilised the `Vector2` class's `add(Vector2 v)` (which sums the two vectors and loads the solution into the first) and `scl(float sf)` (which scales the vector by scale factor sf).

Velocity Verlet

Section 2.2.3 discussed the process of integrating a second order differential equation of the form $\ddot{y} = \frac{d^2y}{dx} = F(y)$. To find the equations in Velocity Verlet form see below:

$$\dot{y}_{x+\frac{h}{2}} = \dot{y}_x + xF(\ddot{y}_x)$$
$$y_{x+h} = y_x + xF(\dot{y}_{x+\frac{h}{2}})$$

This section will discuss how this form can be used to integrate the equations of motion within the confines of gravitational acceleration and describe the implementation of this procedure in the project. The equations to be integrated as with Euler's method are $\ddot{\boldsymbol{r}} = \boldsymbol{a}$ and $\dot{\boldsymbol{v}} = \boldsymbol{a}$. Unlike Euler's method, Velocity Verlet solves second order differential equations, so the integration process will only have to be performed once. The velocity $\boldsymbol{v}$, is the solution of the first integral of $\boldsymbol{v}$ and is produced on the way to computing $\boldsymbol{r}$.

To utilise VV, we shall determine what each variable represents in each equation, what modifications to the final equation need to be made and then substitute the values across. Firstly, the equations are to be integrated with respect to time t across h size intervals. The variable that is to be integrated is the acceleration $\boldsymbol{a}$, the first integral of which is the velocity $\boldsymbol{v}$ and the second integral is the position $\boldsymbol{r}$. The function upon the second integral of the primary variable $\boldsymbol{a}$ is the acceleration computation. We shall denote this function as $\boldsymbol{a}(\boldsymbol{r})$. As a consequence, the initial pair of integrals, with the time parameter phase shifted becomes:

$$\boldsymbol{v}_{t+\frac{h}{2}} = \boldsymbol{v}_t + \int_t^{t+h} \boldsymbol{v}(t)dt$$
$$\boldsymbol{r}_{t+h} = \boldsymbol{r}_t + \int_{t+\frac{h}{2}}^{t+3\frac{h}{2}} a(\boldsymbol{r}(x))dx$$

This is then integrated numerically to find the final two equations:

$$\boldsymbol{v}_{t+\frac{h}{2}} = \boldsymbol{v}_t + h\boldsymbol{a}(\boldsymbol{v}_t)$$
$$\boldsymbol{r}_{t+h} = \boldsymbol{r}_t + h\boldsymbol{a}(\boldsymbol{v}_{t+\frac{h}{2}})$$

To implement this, we must perform the operations with the positions updated a half step away from the velocity. Consequently, the accelerations are computed for the initial half step, and the velocity value is computed and loaded into the Bodies, followed by the acceleration being computed a second time so that the new positions can be generated using the second equation. The implementation of this can be found in Listing 5.6.

```
public static Array<? extends Body> VelocityVerlet(Array<? extends Body> bodies, float dt) {
  for(Body i:bodies) {
    BruteForceI(bodies, i);
    Vector2 acc = i.getA();
    i.setR(
      i.getR().add(i.getV().add(i.getA().scl(0.5f*dt)).scl(dt))
    );
    BruteForceI(bodies, i);
    i.setV(
      i.getV().add(i.getA().add(acc).scl(0.5f*dt))
    );
  }
  return bodies;
}
```

Figure 5.6: The implementation of Velocity Verlet.

Runge Kutta 4

The method discussed in the research section solves equations of the form $y' = F(y, x)$. We can see the N-Body problem in this format by considering:

$$r'' = v' = a(r,t) = \sum_{j=1}^{N} \frac{Gm_j}{|\boldsymbol{r}_j - \boldsymbol{r}_i|^3}(\boldsymbol{r}_j - \boldsymbol{r}_i)$$

Although $a(r,t)$ does not explicitly mention t, the set of values of $\boldsymbol{r}$ depend upon time and can only be found when the simulation is in state t, which provides an significant technical challenge to find any solution for calculation of $a(r,t)$ at a given t we require an collection of `Body` objects in that state to iterate through and determine the value. The second challenging aspect of this problem is the second order nature of the ODE, requiring a mechanism to perform both integrals simultaniously, as Runge Kutta approximates first order ODEs.

Following on from C.J. Voesenek's mathematics, to produce a Runge-Kutta 4 form of the equations of motion integral we have the following equations [18]:

K set of first integration (velocity):

$$\boldsymbol{K}_{1_{v_{t+h}}} = \boldsymbol{a}\boldsymbol{r}_i$$

$$\boldsymbol{K}_{2_{v_{t+h}}} = \boldsymbol{a}(\boldsymbol{r}_i + \boldsymbol{K}_{1_{r_{t+h}}}\frac{h}{2})$$

$$\boldsymbol{K}_{3_{v_{t+h}}} = \boldsymbol{a}(\boldsymbol{r}_i + \boldsymbol{K}_{2_{r_{t+h}}}\frac{h}{2})$$

$$\boldsymbol{K}_{4_{v_{t+h}}} = \boldsymbol{a}(\boldsymbol{r}_i + \boldsymbol{K}_{2_{r_{t+h}}}\frac{h}{2})$$

$$\boldsymbol{K}_{4_{v_{t+h}}} = \boldsymbol{a}(\boldsymbol{r}_i + \boldsymbol{K}_{2_{r_{t+h}}}h)$$

K set of Second integration (position):

$$\boldsymbol{K}_{1_{r_{t+h}}} = \boldsymbol{v}_i$$

$$\boldsymbol{K}_{2_{r_{t+h}}} = \boldsymbol{v}_i\boldsymbol{K}_{1_{v_{t+h}}}\frac{h}{2}$$

$$\boldsymbol{K}_{3_{r_{t+h}}} = \boldsymbol{v}_i\boldsymbol{K}_{2_{v_{t+h}}}\frac{h}{2}$$

$$\boldsymbol{K}_{4_{r_{t+h}}} = \boldsymbol{v}_i\boldsymbol{K}_{3_{v_{t+h}}}h$$

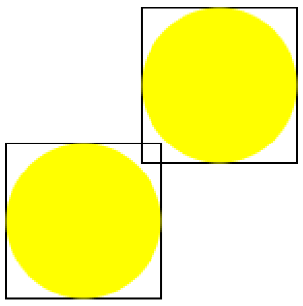


Figure 5.7: A diagram to show intersecting bounding boxes without a collision.

Final Values:

$$\boldsymbol{v}_{t+h} = \boldsymbol{v}_t + \frac{h}{6}(\boldsymbol{K}_{1_{v_{t+h}}} + 2\boldsymbol{K}_{2_{v_{t+h}}} + 2\boldsymbol{K}_{3_{v_{t+h}}} + \boldsymbol{K}_{4_{v_{t+h}}})$$

$$\boldsymbol{r}_{t+h} = \boldsymbol{r}_t + \frac{h}{6}(\boldsymbol{K}_{1_{r_{t+h}}} + 2\boldsymbol{K}_{2_{r_{t+h}}} + 2\boldsymbol{K}_{3_{r_{t+h}}} + \boldsymbol{K}_{4_{r_{t+h}}})$$

As can be seen from these equations, each subsequent K value depends upon the previous value in the opposite set (e.g. $\boldsymbol{K}_{3_{r_{t+h}}}$ for the positions requires $\boldsymbol{K}_{2_{v_{t+h}}}$ to be computed first). Consequently, the operations must be interlaced, computing each value in pairs. Furthermore, since the acceleration depends upon time through the state of the network, so as the new values are being computed, incrementing through the four quartiles ($\boldsymbol{K}$ values) of the time step, they must use a copy of the simulation to prevent the origin version becoming corrupted by the operations required upon the simulation. Such a copy will consist of an Array of SimpleBody clones of the CelestialBody objects in the actual simulation.

Moreover, not only must we compute these equations every time step, they must also be computed for every body in the simulation. Therefore each of the 8 $\boldsymbol{K}$ values will be stored as an array, which are computed in loops, pairs of $\boldsymbol{K}$ values at a time. Finally, they are each summed together to compute the final two values per object. A listing of this complete function can be found in the Appendices, it is not included here for brevity.

5.1.3 Collisions

To approach the problem of collisions, we must first divide the problem into two parts, detection and resolution. The former's role is to discern when a collision has occurred and the latter's concern is to determine how the collided objects should respond to such an event.

Detection

A typical method to approach collisions is to use bounding boxes. For this method all collidable objects have a rectangle drawn around them, and should any two of those rectangles intersect, a collision event would be triggered. Although this method can be very simply implemented within most programs, it suffers a problem that unless the collidable object is a rectangle, there will always be a scenario where a collision event is falsely triggered. This can be seen in Figure 5.7.

However, there is a more elegant approach that can be taken to solve this problem. We can assume within the simulation that all collidable objects are spherical, the definition of a circle being a region of points equidistant from a central coordinate. Any two circles are considered intersecting if the following inequality is satisfied:

$$|\boldsymbol{r}_A - \boldsymbol{r}_B| \leq R_A + R_B$$

```java
private static Array<? extends Body> detectCollisions(Array<? extends Body> bodies) {
  for(int i=0; i<bodies.size; i++) {
    Body current = bodies.get(i);
    for (int j=i+1; j<bodies.size; j++) {
      Body other = bodies.get(j);
      if (isIntersecting(current, other)) {
              resolveCollision(current, other);
      }
    }
  }
  return bodies;
}
```

Figure 5.8: Implementation of the Collision Detection algorithm.

```java
static boolean isIntersecting(<? extends Body> A, <? extends Body> B) {
  return A.getRadius() + B.getRadius() >= A.getR().sub(B.getR());
}
```

In our program where the central coordinate is $\boldsymbol{r}$, and the distance from that central point (the radius) is R. These are two values already stored in memory as a part of `Body` interface, thus eliminating the need for additional classes to be constructed to define the arbitrary objects such as bounding boxes.

Now that a condition has been developed that can determine if any two bodies are intersecting, there should be a method of iterating along all possible pairs of comparisons. For this we shall perform an exhaustive search upon the set of pairs of collidable objects. Due to the fact that the inequality test is symmetric $isIntersecting(A, B) \Leftrightarrow isIntersecting(B, A)$, the number of tests can be halved. Although this has no effect on algorithmic complexity, it can noticeably reduce the number of operations for a small number of iterations, as seen in this program. Listing 5.8 shows the implementation of this approach, to improve code clarity of readability it would have been helpful to use the Java iterator `for (Body a: bodies) {...}`, sadly this Java iterator syntax does not support nesting, so standard loops with an integer iterator were used.

Resolution

In the previous section a successful approach to the collision detection was implemented. Upon a collision event the `resolveCollision()` method is called. In this section, we shall discuss the design and implementation of the resolution of such a collision.

The chosen approach was to implement elastic rigid body collisions, as discussed in the research section. This collision handling technique was selected because although it does not perfectly represent how collisions occur in reality (as no simulation possibly could), it does an effective job. Another approach considered was the N-Body approach: This would involve converting each CB into multiple smaller CBs handling the collision just as the two objects shatter. However, the code and time complexity would have been an insurmountable issue for the amount of time available in this project. Another alternative approach considered was absorption, forming a single CB on all collision events. This is an approach taken by several other applications, but it was not chosen in this project, partly because it has already been done before in this context, but also because it doesn't show as much physics to the user. By handling collisions as them essentially bouncing off of one another, one can see the conservation of kinetic energy within collisions and the imparting of kinetic energy from one mass to another.

To resolve a rigid body collision, we must first discuss momentum. Every moving object with mass has the property of momentum p according to $p = m\boldsymbol{v}$. The total momentum in each dimension between two objects in a collision is also preserved in elastic collisions (elastic referring to the fact that energy is not lost to heat or sound). This means that if we have two objects colliding with one another the following equation holds:

$$m_a\boldsymbol{v}_a + m_b\boldsymbol{v}_b = m_a\boldsymbol{v}'_a + m_b\boldsymbol{v}'_b$$

Where $\boldsymbol{v}_i$ is the velocity before the collision and $\boldsymbol{v}'_i$ is the velocity after. Using this method we can devise a suitable method of testing the system.

The change in velocity, or the imparting of kinetic energy in a collision, is referred to as the impulse. Notably, the impulse will always act in the direction of the collision. Using the following equation

```
public static void resolveCollision(Body a, Body b) {
  Vector2 r_a = a.getR();
  Vector2 r_b = b.getR();
  Vector2 v_a = a.getV();
  Vector2 v_b = b.getV();

  float m_a = a.getM();
  float m_b = b.getM();
  float radius_a = a.getRadius();
  float radius_b = b.getRadius();

  float m_a_impulse;
  if (m_b != 0) m_a_impulse = 2*m_b/(m_a + m_b) / new Vector2(r_a).dst2(r_b);
  else m_a_impulse = 0;

  float m_b_impulse;
  if (m_a != 0) m_b_impulse = 2*m_a/(m_a + m_b) / new Vector2(r_b).dst2(r_a);
  else m_b_impulse = 0;

  Vector2 vel_diff_a = new Vector2(v_a).sub(v_b);
  Vector2 pos_diff_a = new Vector2(r_a).sub(r_b);

  Vector2 vel_diff_b = new Vector2(vel_diff_a).scl(-1);
  Vector2 pos_diff_b = new Vector2(pos_diff_a).scl(-1);

  float coeff_a = m_a_impulse * new Vector2(vel_diff_a).dot(pos_diff_a);
  float coeff_b = m_b_impulse * new Vector2(vel_diff_b).dot(pos_diff_b);

  Vector2 V_a = new Vector2(v_a).sub(new Vector2(pos_diff_a).scl(coeff_a));
  Vector2 V_b = new Vector2(v_b).sub(new Vector2(pos_diff_b).scl(coeff_b));

  a.setV(V_a);
  b.setV(V_b);
}
```

Figure 5.9: The collision handling implementation.

from [16], we can find the change in velocity of the collision, where $\boldsymbol{r}_{ab}$ and $\boldsymbol{v}_{ab}$ are relative position and velocity vectors respectively.

$$\delta \boldsymbol{v}_a = -\frac{2m_b}{m_1 + m_2} \frac{\boldsymbol{r}_{ab} . \boldsymbol{v}_{ab}}{||\boldsymbol{r}_{ab}||^2} \boldsymbol{r}_{ab}$$

Since the mass is preserved in the collision, and the program already handles the movement of objects through the integration methods, the only parameter of the CB that needs updating is the velocity. As a consequence, from this single equation we can completely handle the collision. Furthermore, since the collisions are called with pointers to both CelestialBodies, no further calls are needed. The implementation of this equation can be found in Figure 5.9, where the implementation has been broken down into smaller statements to improve efficiency and readability. One edge case that it is necessary to handle is that of the case where one of the objects has a mass of zero. An object of zero-mass would not pass any impulse to any object that it collided with. For this purpose and to prevent zero divisions, a check is made on the calculation of the impulse to assert that the impulse has no magnitude if the object creating it has no mass.

5.1.4 Predictive Curve

The next requirement was to compute the predictive curve of the path of the `Body` objects in the simulation. This section will detail design and implemention of code that produces a sequence of points to represent the future path of the `Body`.

```
private static Array<Vector2> predictCurve(Array<CelestialBody> bodies, float length, float dt) {
  Array<SimpleBody> pseudoBodies = new Array<SimpleBody>();
  Array<Vector2> predictiveCurve = new Array<Vector2>();
  for (int i = 0; i<bodies.size; i++) {
    pseudoBodies.add(new SimpleBody(bodies.get(i)));
  }
  for (int i = 0; i<1000; i++) {
    predictiveCurve.add(new Vector2(pseudoBodies.get(pseudoBodies.size - 1).getR()));
    Algorithms.compute(pseudoBodies,0.01f);
  }
  return predictiveCurve;
}
```

Figure 5.10: The implementation of Predictive Curve generation.

The approach to this problem, was to generate a list of position vectors that we can confirm will be almost identical to that of the simulation. So the most obvious solution is to use the previously implemented algorithm to generate points in the future, this algorithm is accessed through `computeC(Array<? extends Body> bodies, float dt)`. The algorithm modifies each `Body` in `bodies` to produce the next iteration of the system.

Using this function, we can compute the next 1000 positions of the `Body`. However, if the main container of the objects is used `bodies`, it will change the positions and velocities of all the other objects in the simulation and there would need to be a mechanism to reset them to a previous value. So the solution is to create a copy of `bodies` to a new array entitled `psuedobodies`. This array must consist of new body objects, but these objects will never be rendered. Therefore, we can use a new minified implementation of `Body`. This new version will only need to implement the functions contained in the interface, and can completely avoid containing LibGDX render objects (many of these objects can consume a lot of memory, and take time to instantiate). The implementation of this class, named `SimpleBody`, consists entirely of getter and setter methods, and a constructor to produce a copy of an object that implements `Body`.

Consequently, the code that performs this process can now be constructed. On each request for a predictive curve, it will need to create a copy of the current state of the simulation. Then, update the simulation a given number of times `length` (the length of the curve), with a given timestep size `dt`, before returning an arraylist of the most recently added object's position on each update. Such an arraylist can then be used to construct the curve which is discussed in Section 5.2.2. A listing of the function developed here can be seen in Figure 5.10.

5.2 Interface

In the previous section, we discussed the implementation and encapsulation of the simulation algorithms. In this section, we will discuss the implementation of the surrounding interface as specified in the requirements. We will start with the first of these requirements, which was the display of the Celestial Bodies. Then we will walk through the rest, adding each additional feature as they were completed during the project.

But first a brief introduction to LibGDX's application system: The primary class is an object that implements `ApplicationListener` and is called by the appropriate launcher of the program depending on the run-time environment (Desktop/Android/etc...). This object, which we shall refer to as the application listener, implements a number of important methods, two of which are `create()`, called on the launch of the program, and `render()`, called on every screen render. From these two methods, the vast majority of the program's logic is called, `create()` is used to initialise all your game logic and renderable components and `render()` is used to render them to the screen.

Rendering in LibGDX can be achieved from one of a number of libraries, including many OpenGL commands. A key library in developing 2D applications is the `Scene2D` system. `Scene2D` is a tree based rendering path, where every renderable object (object that extends the `Actor` interface) is a child of a group or the Stage (the root node). Every node in this tree implements the `Actor` interface, which means it must have a method to `render()` and to `act()` (run before each render cycle to evaluate game logic). Events in Scene2D are handled through the use of listeners that are attached to Actors. A listener will

```
@Override
public void render() {
  main.act(Gdx.graphics.getDeltaTime());
  main.draw();

  ui.act(Gdx.graphics.getDeltaTime());
  ui.draw();
}
```

Figure 5.11: Simulator's render function.

consist of a collection of methods to run on each type of event. For example, a `ClickListener` might listen for left clicks on an Actor. The event would be called only if a left click occurred while the mouse was over the Actors coordinates.

From here we can specify an appropriate design for the application's interface. At the top level, the ApplicationListener is typically named after the application. At this point in design the program did not have a distinct title, so the class was simply named `Simulator`. `Simulator` creates the program window, and loads two Stages: main and UI.Simulation based objects will be attached to main (such as masses and predictive paths), while various controls will be rendered in the UI section. The UI will always be drawn on top of the rest of the program, this can be seen in `Simulator`'s render function shown in listing 5.11.

5.2.1 Body Visualisation

This section will discuss the rendering of the masses in the simulation. As discussed in the previous section, all renderable bodies must extend the `Actor` class. This means that we shall need to create an implementation of the `Body` interface that extends the actor class. Here in lies the reasoning for specifying the `Body` as an interface instead of an implemented class, because by extending the `Actor` class each body would significantly increase its memory usage, and not all bodies need to be rendered. Therefore, it is appropriate to use an interface so that a smaller, simplified version of the object can also be used when necessary. However, for the purposes of this section we are only interested in the larger, renderable class which will be referred to as `CelestialBody`. This class implements the `Body` interface and extends the `Actor` class.

At this point, we must discuss a minor flaw in the construction of the Scene2D library. As all objects are rendered in the tree, a Batch object is passed down to perform the rendering. This saves computation as the `Batch.start()` function is known to be a very expensive operation. However, this batch object only supports the rendering of pixel maps, whilst at the same time the `glBegin()` command from OpenGL has been wrapped in LibGDX to the `ShapeRenderer` class. The `ShapeRenderer` class contains methods to draw non-rasterised shapes, which prevents shapes becoming pixelated when zoomed in. Consequently, the batch passing process of scene2d is unable to render non-rasterised circles, So it is necessary to keep a ShapeRenderer inside the `CelestialBody`, which further increases the size of the object in memory. This supports the design decision of using an interface instead of a class to represent the object.

The core logic of the simulation required a separate space to be encapsulated, for this purpose the `Group` class was extended to produce the `Simulation` class. As well as encapsulating simulation logic, it can also be used to point to the `CelestialBody` instances to trigger their `render()` methods.

Tails

To improve the visual fidelity of the motion of the planets, a tail has been added to the motion of the planets. This was implemented using a queue in the CelestialBody class. The `setR()` method was altered, so that it would push new positions to the queue. Then, once the queue has more than 50 items, it pops off the oldest item as the new positions are added. In this method, each celestial body keeps a record of the last 50 coordinates that it has been in. The method is shown in listing 5.12.

To render this history of previous positions, we utilise the `rectLine()` function within the shape renderer. This function will draw lines of a specified width and colour to the screen. Prior to the rendering of the planet itself (so that the tail is not drawn over the top of the planet), a for loop iterates through each item in the list (except the last), drawing a line from that position to the next position in

```
public void setR(Vector2 r) {
  /* add new position to trail, remove previous position if trail is too long */
  trail.addFirst(new Vector2(r));
  if (trail.size > 100) trail.removeLast();
  this.r = r;
}
```

Figure 5.12: The set position method from CelestialBody

the stack, each time decreasing the width and brightness of the line by a scale factor of 0.95. This can be seen in the listing below:

```
float grey = 1.0f;
for (int j = 0; j<trail.size - 1; j++) {
    render.setColor(1.0f,1.0f,1.0f,grey);
    grey *= 0.95f;
    render.rectLine(trail.get(j),trail.get(j+1),0.012f*grey );
}
```

Visualising Vectors

An additional feature to the visualisation of the bodies within the simulation is to display their acceleration and velocity vectors. This can be achieved by drawing the vectors to the screen with their origins on the CelestialBody, vectors are typically drawn as lines with an arrow head at the end to indicate its direction. The two line segments that are to be drawn can be seen in parametric form below:

$$Line_v = \boldsymbol{r} + t\boldsymbol{v}, 0 \leq t \leq 1$$

$$Line_a = \boldsymbol{r} + t\boldsymbol{a}, 0 \leq t \leq 1$$

However, this feature should not be constantly drawn for all CelestialBodies as that would significantly clutter the screen, so it should only be shown on a currently selected CelestialBody. The implementation of Body selection is described in Section 5.2.5, so for the purpose of this section we shall assume it already exists, can be accessed through a CelestialBody variable in the Simulation entitled `selected`, and is `null` when there is no selected item.

The render loop needs to draw two items, the line and the arrow head. These can be drawn using the shaperenderer within Simulation. Lines are drawn in LibGDX with the start and endpoints of the line, which can be obtained from the parametric equations by taken $t = 0$ and $t = 1$ results respectively. Since the lines should only be shown when a CelestialBody is selected, these lines should only be drawn when `selected` is non-null.

The next step is to produce a ArrowHead, this issue would be quite simple if it were not for the necessity for it to be an actor object in order to complete the EditBody menu (discussed in Section 5.2.5).Consequently, a simple actor was implemented `ArrowHead`. This class is constructed with the origin coordinate $\boldsymbol{r}$ as its position and either the acceleration $\boldsymbol{a}$ or the velocity $\boldsymbol{v}$ as its vector. Using these values, it determines exactly where and how rotated the triangle should be rendered. Two of these ArrowHeads are created on instantiation of the Simulation and their positions are updated or it is made hidden, depending upon the state of the `selected` variable.

5.2.2 Object Creation

The next requirement was to develop a tool that could be used to add new planet's to the simulation. Developing user tools creates an interesting problem in an event driven model such as LibGDX, as changes to the system only occur when a user event is triggered. Therefore, the system must have some method of retaining what to do under certain circumstances. Interestingly, this is a state-machine based problem. We can solve it by developing a set of state transitions (methods to call upon whilst in a specific state).

When creating a new planet, several pieces of information are required from the user: Location, Mass, and velocity. The first step is to decide the most appropriate way to capture these values. The location is simple, the Body should be created exactly where the user clicks. The velocity (a vector) can then

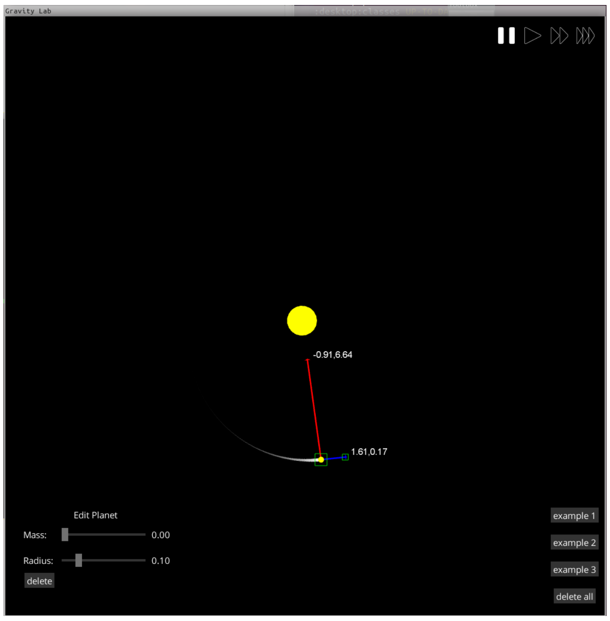


Figure 5.13: A screenshot of the edit menu and visualisation vectors.

be captured by drawing a vector from the the body to the current position of the mouse pointer. By updating this vector while the mouse moves, the user can draw their own velocity vector to give as input to the planet creation tool. Finally, the mass is more tricky. It could be achieved by adding a slider to the screen, or letting the user type a number, but this would feel clunky and out of place, since all other components of the planet creation only rely upon clicking with the mouse. The solution is to take a similar approach to the velocity and represent the magnitude of the mass as the distance from the centre of the mass to the mouse pointer. To further highlight this to the user, a circle was added to the display to indicate that the user's next click would input some data to the program.

From here, we can begin to tie the various user inputs together. The first input would be location, as the most intuitive way to create something would be to click in where you would like to place it, then alter its parameters later. It should be possible to create new planets as long as the simulation is not running. This provides two states for the first parameter: user can create a new object and user cannot create a new object (because the simulation is running). These two state will be referred to as `EDIT_MODE` and `SIMULATING` respectively. The alternation between these two states would be operated by the time menu, which is discussed later in the document.

From `EDIT_MODE` the location input transition can be called, this is the first user action that implies the user wants to create a new object. The transition serves two purposes: we know the user wants a new object and we know the location they want that object to be in. Next we, could either accept the velocity or the mass as input, but a feature required later on is the predictive curve of the object's path, this would be provided while the user is selecting the velocity. The mass of the object is going to effect the path of this curve, because an object with mass will move the other objects in the system, thus effecting the magnitude and direction of the acceleration experienced by this one. Therefore, it is appropriate to select the mass before we select the velocity, so that the user can decide the path of the planet as the final item. The location input transition will travel from the `EDIT_MODE` state, to the state where we are waiting on a mass input `GET_MASS`. From there, the only transition available should be to accept the mass (which means that we cannot transition to simulating, so the time menu will have to handle this). Such a transition would then change to `GET_VELOCITY`, from which we can return to `EDIT_MODE` once the velocity has been established. A complete state transition diagram of this process can be seen in figure 5.14.

To implement any state based system, a collection of states and state transitions are required. For this purpose in java we shall use the java state design pattern as outlined in James W. Cooper's Java Design Patterns [8].

The state machine design pattern constructs a wrapper class, which adjusts its behavior depending upon its own state. The behaviors (methods), are what it should do under specific circumstances (events). The wrapper's behaviors are typically encapsulated into subsidiary classes each with a special variable

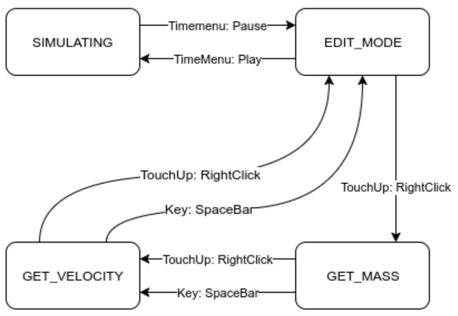


Figure 5.14: The state transition diagram of the create body tool.

```
public class State {
    SIMULATING, EDIT_MODE, GET_MASS, GET_VELOCITY
}
```

Figure 5.15: A listing of the state enum.

that stores a pointer to the wrapper. Methods within the subsidiary classes that change the state of the wrapper are referred to as state transition functions, to further the analogy of a state machine.

In this program we shall have to modify this model slightly to accommodate for the event driven listener system.

The class that is to be the state model, the primary class, in this context is `Simulation`. As well as all the items described earlier, it contains an instance of the `State` enum (shown in listing 5.15). Attached to `Simulation` using the `addListener()` method from `Actor`, is `SimulationListener`, a class that extends LibGDX's `InputListener` to register mouse events. The actions to perform on `Simulation` during each state are specified in `SimulationListener`'s mouse event methods. And `SimulationListener` is constructed with a pointer to the instance of `Simulation` from which it is attached.

Now that the overarching structure is ready, the next step is to describe the state transitions and interface logic (this is represented in the implementation by the contents of the switch case statements). The `InputListener` has 3 different types of events that it captures: `touchDown()` is called as a mouse/-touchpad button is pressed, `touchUp()` is called as it is lifted again and `mouseMoved()` as the pointer is moved. The `touchUp()` event (which is only called after a succesful `touchDown()` event), is used for all state transitions in this section. Since we want the new object to be created on a right-click (so that the left click is still available for camera requirement), all of these transitions act only if the event is a right-click.

We will now discuss exactly what happens on each transition. The system starts in `EDIT_MODE`, where there are two options available, `SIMULATION` or `GET_MASS`. If a right-click event is received, then the listening will record the location of the event and create a new CelestialBody in this position. At this

```
class SimulationListener extends InputListener {
    private Simulation simulation;
    ...

    public SimulationListener(Simulation simulation) {...}

    public boolean touchDown(...) {...}
    public void touchUp(...) {...}
    public boolean mouseMoved(...) {...}
}
```

Figure 5.16: A summary listing of the SimulationListener class.

time the mass and velocity are set to zero. It then changes the state to `GET_MASS` and finishes the event. Whilst in the `GET_MASS` mode, `mouseMoved()` events are handled to record the current distance from the mouse to the new CelestialBody. The radius is passed to the `Simulation`, which uses it to draw a circle around the Celestialbody, so that the mouse appears as though it is drawing a circle. On another right click event, the radius is stored as the mass of planet and a transition is made to the `GET_VELOCTY` state, which on mouse moved events records the current position of the mouse and passes it to the `Simulation` to draw the vector between the body and the mouse. A final right-click event will transition back to `EDIT_MODE` after storing the vector as the new velocity of the object.

Drawing the Predictive Curve

Once the object creation tool was established, the next step was to draw out the predictive curve. This was completed much in the same way as the vector outline and the circle drawing, however it had a few more steps to complete. Using the process outlined in the simulation section, an ArrayList of future positions can be produced. This would have to be updated on every mouse movement during the `GET_VELOCITY` state, so this is an appropriate place to store such code. This could also have been placed in the render section, but by placing it here, we can reduce the total number of times the predictive curve is generated, especially since it has a very high time complexity. Now that we can assume that the contents of the ArrayList will always contain a predictive curve in the `GET_VELOCITY` state, the render section of the simulation should draw it while in that state. The proposed method of drawing it is to use the `rectline()` function, in a similar method to the CelestialBody trails, but instead of drawing lines to join up the previous positions of the CelestialBody, in this instance we draw the contents of the ArrayList, this being the future positions of the planet.

Help Text

Although the right click method of creating planets is very simple to use, there is no visual cue to indicate what it does. Consequently, some initial help text was proposed. The help text should walk the user through the steps to create their first planet, but it shouldn't further interfere with them after that and it shouldn't be too intrusive as to be distracting upon launch. So 3 items of text were proposed:

1. 'Rightclick to make your first planet'
2. 'Rightclick to set its mass'
3. 'Rightclick to set its velocity'

Each item should appear until the user performs the action. Since we can assume that any time the user launches this program they will always want to create a planet, the final piece of text should disappear very early on in program usage.

To make the text a bit more interesting to look at, a simple typing animation should be included. Such an animation would add a new character every $\frac{1}{4}$ of a second until the sentence is complete.

To implement this, a boolean value was added to the ApplicationListener. This boolean initialises to true on launch. While it is true, the program draws help text 1 in `EDIT_MODE`, help text 2 in `GET_MASS` and help text 3 in `GET_VELOCITY` and the boolean is disabled on the transition from `GET_VELOCTY` to `EDIT_MODE`.

The initial help text is initialised by setting two parameters in the ApplicationListener, the string of the help text and an index (instantiated to 0). On each render loop, if the help text boolean is enabled, [0,index] substring is rendered to the screen as a label.

To create an interesting animation for the program, several methods were defined to manipulate the index of this substring: `incrementHelpText()`, `decrementHelpText()` and `changeHelpText()`. Using these methods we use two states for the program, incrementing and decrementing. This state is represented by another boolean `incrementhelptext` and is initialised to `true`. When in the incrementing state, the index of the helptext will be incremented every 50ms until it reaches its maximum value (the length of the string) and when in the decrementing state the help text index is decremented every 10ms until the index is at its minimum position 0, afterwards the text is changed to the new help text string and the state is switched to incrementing. On a call to `changeHelpText(String text)`, the state is changed to decrementing and the new helptext parameter is loaded with a new string (the new string won't be used until the switch of decrementing to incrementing).

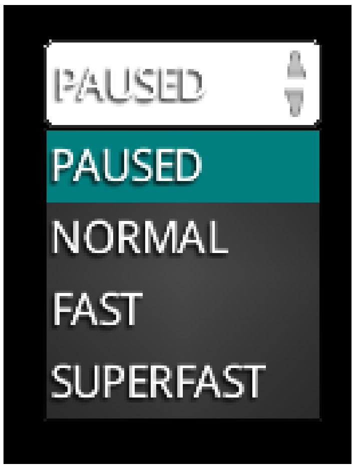


Figure 5.17: A screenshot of the time menu in select box form.

This has the effect of a string being appeared to be typed out in the incrementing state (typing 1 character every 50ms) and the effect of the previous line being deleted (one character every 10ms), before typing the new line.

5.2.3 Time Menu

This section will discuss the design and implementation of the time menu requirement. The problem consists of two aspects, starting and stopping the simulation algorithms and creating a user friendly interface to access it.

Firstly the interface, the Scene2D library within LibGDX comes with a strong collection of UI tools that can be used to make this work easier. One of these classes is the SelectBox, a drop down box where you can select one of a list of items. This could be an effective way of switching between the EDIT_MODE and SIMULATING states. However, it is necessary that the button is disabled whilst not in one of these two states. Helpfully, there is a function within Button (an abstract class that SelectBox extends) that already performs this action. So, on changes from EDIT_MODE to GET_MASS, this button should be disabled and on changes from GET_VELOCITY to EDIT_MODE it should be enabled. The planet creation tool already handles its disabling while in SIMULATING by implicitly ignoring inputs while in that state. Although the SelectBox object has all the features of a working button, it does not have a position on the screen. For this we require a Table object, which will act as the container for all future buttons that appear on the screen.

So the structure for the timemenu is as follows: Within the ApplicationListener `Simulator`, a table is created and attached to the `ui` stage. The program then calls `createUI()` a function to load all the buttons and insert them into the Table in the appropriate order to achieve a suitable layout on the screen. One of these buttons is the SelectBox `timeMenu`, with the list `[PAUSE, PLAY, FASTER, FASTEST]` as its items. Attached to `timeMenu` is an anonymous extension `ChangeListener`. Which is called whenever the timeMenu's value has been changed, the `Simulation` object is then set to the new state on each event. `Play`, `Faster` and `Fastest` are all considered versions of `SIMULATING`, while `PAUSE` in this context represents `EDIT_MODE`. The SelectBox timeMenu can be seen in figure 5.17.

After getting feedback from use testing with a small number of participants, it was found that the SelectBox solution was not very effective. Having a SelectBox was not an obvious method of changing the time settings of the simulation, often users would require prompts before realising how to start the simulation. A proposed solution to this problem was to use mediabuttons. Media Buttons are a ubiquitous standard for changing time. We see them in music players, and videos all the time. This implementation will feature 4 options, as in the previous menu. Each represented by a standard media button symbol, with very obvious connotations of rate of time. The images for this program were created in GIMP, an image editing program and are pictured in figure 5.18. Each symbol requires two pictures, each for selected and unselected. Both of which need to be visible on the black background of the simulation.

The `ImageButton` class in LibGDX supports several textures to be applied to it for different states: up (unselected and not depressed), down (touchdown event has occurred over the button but a touchup

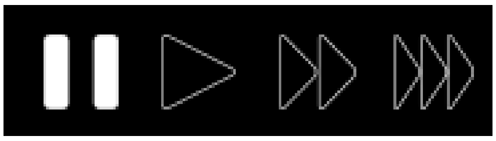


Figure 5.18: A screenshot of the time menu in media button form.

event has not occurred yet) and selected (shown when the button is selected and not depressed). The white (bolder) images will be used for down and selected states, while the black images will be used for the up state. To prevent more than one button being activated at once, the ButtonGroup class was used. This class automatically deselects and selects buttons to preserve the minimum and maximum number of buttons selected in its collection. This instance of ButtonGroup has the minimum and maximum parameters both set to 1 and all 4 imagebuttons added to it. So only one button may be selected at once. When using the SelectBox version, it was very simple to send the Time enum value to the simulation to update the new state, because the selectbox had a method to return the current state of the menu. To achieve this with the ImageButton class, a new class `ValuedImageButton` was created that extended Image button, but had a new public variable to store its value. Sadly, change events do not occur to ButtonGroups, so prevent the need to attach listeners to all 4 buttons, which would be unnecessarily complex, a small check was added in the render loop to update the simulation with the current time menu value.

```
Time t = mediabuttons.getChecked().value;
simulation.setSpeed(t);
```

Up until this point the two parts of the program have been quite separate, the algorithms are encapsulated into a collection of methods. To achieve this goal, the render loop should seamlessly switch between including the algorithms into the render loop and excluding them. Now that the TimeMenu has been implemented and sets a parameter in the simulation to indicate what state it should be in. The simulation needs to adjust its behaviour in its render loop accordingly. This was added through the `act()` method from the `Actor` class, a switch statement determines how to call the computation to increment the simulation. This method is called as a part of the LibGDX stage render loop, where a delta time value is passed between down to all act methods. Delta time is the amount of real world time that passed between the current render and the previous render, as measured by the system clock. This is a more robust way of simulating physics than the alternative, where the time is a constant in all physics calculation. Such a method would cause a disparity between the passage of time in the simulation and reality. As the number of frames rendered per section fluctuates (as it does in all simulations due to changes in processor load), the rate of the simulation would change. This problem is avoided in the delta time system, because as the processor takes longer to render a frame, the simulation assumes more time has passed and vice versa.

```
public void act(float delta) {
  switch (speed){
    case PAUSED:
      break;
    case PLAY:
      Algorithms.compute(bodies, delta);
      break;
    case FASTER:
      Algorithms.compute(bodies, delta*5);
      break;
    case FASTEST:
      Algorithms.compute(bodies, delta*10);
      break;
  }
}
```

5.2.4 Camera

The two features required in the camera, is to drag the position of the camera, and adjust the zoom of the camera. To achieve this requires access to the `gluLookAt()` function in OpenGL, which in LibGDX is encapsulated to Camera interface, in particular the OrthographicCamera implementation of the Camera interface will be used as this simulation is in 2D, so an orthographic projection of the program is required. Perspective projects would cause distortions in the zooming of the program.

The default instance of the Stage class will generate its own Camera, but since this program requires access to its own instance, it creates one and launches the stage with that camera in its constructor. Holding a pointer to the camera as a variable stored in the ApplicationListener.

The next step was to attach a listener to the camera object to modify its zoom parameter and position. This was achieved the the CameraInputController, a specialised listener to handle camera events. In particular it handles scrolled and dragged events of the mouse. Firstly the zoom parameter is adjusted as a proportion of the amount scrolled in the event. This zoom parameter must then be maxed with 0.01 to prevent the user from zooming through the simulation plane, making the entire screen black.

Secondly, dragging, is a slightly more complex issue. Drag events are generated in LibGDX for every mouse moved event that is detected after a touchDown event, but prior to a touchUp event. To implement this, the program saves the initial coordinates of the touchDown event and upon each subsequent touchDragged event it find the coordinates difference to determine how much to adjust the position of the camera by. This process can be seen in the Listing below:

```java
CameraInputController camListener = new CameraInputController(cam){
  private int dragX, dragY;

  @Override
  public boolean touchDown(int x, int y, int pointer, int button){
    if (button == Buttons.LEFT) {
      dragX = x;
      dragY = y;
      return true;
    }
    return false;
  }

  @Override
  public boolean touchDragged(int x, int y, int pointer) {
    float dX = (float)(dragX-x)/(float)Gdx.graphics.getWidth();
    float dY = (float)(y-dragY)/(float)Gdx.graphics.getHeight();
    dragX = x;
    dragY = y;
    camera.position.add(dX * 1000f*zoom, dY * 1000f*zoom, 0f);
    camera.update();
    return true;
  }

  @Override
  public boolean scrolled(int amount) {
    zoom += (float)amount/10;
    zoom = Math.max(zoom, 0.01f);
    cam.zoom = zoom;
    return true;
  }
};
```

5.2.5 Object Editor

The final requirement to implement was the object editor. This editor should provide the ability to adjust the radius and mass of any CelestialBody in the simulation. This can be broken up until several sub-objectives: the ability to select a planet, an onscreen menu to adjust the values and delete planets, and an interface with the simulation so that the actions of this menu effects the simulation.

Object Selection

The aim of this feature is to have a simple left click to select a CelestialMass object, there must be an obvious highlight to see which mass is currently selected and there may only be 1 mass selected at any one time.

The click event was handled by using a listener on the CelestialBody object. This listener was an extension of ClickListener and was entitled `BodyListener`. It captures a single event, a click, triggered when the user clicks within the bounds of the CelestialBody. To make sure that this event is properly triggered each CelestialBody was required to have its bounding box be constantly updated with its current position. This was achieved by adding a `setPosition()` call into the CelestialBody `render()` method. There is a method in the Actor class entitled debug(), which toggles whether or the bounding box is drawn to the screen in the render loop. Interestingly this can solve our problem of highlighting which item is selected quite successfully, by drawing a nice square around the CelestialBody as it is rendered to the screen. This was used to save having to code additional drawing information.

The Simulation was to handle which CelestialBody was currently selected at once, since it stores all the CelestialBodies within an ArrayList, giving it access to their calls. And by implementing the BodyListener as a nested class within Simulation, it gives instances of the listener access to Simulation calls. On a click event of any CelestialBody within the simulation the program will have to check several cases that the simulation could currently be in:

1. There is no CelestialBody currently selected
2. There is one CelestialBody currently selected and it is not the clicked CelestialBody
3. There is one CelesitalBody currently selected and it is the clicked CelestialBody

In the first case, the clicked CelestialBody is added to the selected variable stored in Simulation and its debug toggle is called. In the second case the currently selected CelestialBody's debug toggle is called (to remove the debug highlight), the clicked CelestialBody's debug toggle is called and the selected variable is set to the new CelestialBody. And in the final case the currently selected and clicked CelestialBody's debug toggle is called (to remove the highlight) and the selected variable is set to null. The implementation of this can be seen below, with additional code to implement the adjustable velocity as discussed in subsequent sections.

```java
@Override
public void clicked(InputEvent event, float x, float y) {
  if (selected == null) {
    selected = c;
    selected.setDebug(true);
    event.handle();
    addActor(accHead);
    velHead = new ArrowHead(selected.getV(), selected.getR(),Color.BLUE, selected);
    addActor(velHead);
    return;
  } else if (selected.equals(c)) {
    selected.setDebug(false);
    selected = null;
    event.handle();
    removeActor(accHead);
    removeActor(velHead);
    return;
  } else {
    selected.setDebug(false);
    selected = c;
    selected.setDebug(true);
    event.handle();
    removeActor(velHead);
    velHead = new ArrowHead(selected.getV(), selected.getR(),Color.BLUE, selected);
    addActor(velHead);
    return;
  }
}
```

Editing Methods

The parameters required to be changed were the radius, velocity, mass and existence (deleting) of a CelestialBody. This would require access to these parameters in the CelestialBody class and a method within the Simulation class to delete a CelestialBody cleanly.

The velocity and mass already had their own getter methods, as was necessary for the algorithms discussed in Section 5.1. And the velocity already had its own setter. However, for this objective to be completed there needed to be getters and setters for all three of these parameters. The CelestialBody class was therefore modified to include them.

In order to cleanly delete a CelestialBody, it needed to be removed from several aspects of the simulation. The ArrayList that is iterated through to perform the algorithms in the Simulation. The render group that the simulation extends (so that the CelestialBody is no longer rendered). And it needed to be unselected if it was currently selected CelestialBody (which we can assume it always is, since the only access to the delete button will be to the currently selected CelestialBody). So a method is proposed `remove(CelestialBody c)` within Simulation. It will take a CelestialBody as input, and remove it from the program as specified above. The implementation of this can be seen below (Note the method contains other code that is discussed in Section 5.2.5).

```
public void remove(CelestialBody c) {
  bodies.removeValue(c, false);
  this.removeActor(c);
  selected = null;
  removeActor(accHead);
  removeActor(velHead);
}
```

Edit Menu

The next goal was to create a menu that can adjust the radius, velocity, mass and delete a CelestialBody. The radius and mass are both scalar parameters, so an effective way of adjusting them is through use of a slider. Scene2D UI contains a class entitled Slider that can achieve this purpose. The mass slider scales from 0 to 10 in 0.1 increments. As this was found to be an appropriate range of possible masses to produce interesting simulations. The radius will scale from 0.001 to 0.05 in 0.0001 increments, although these numbers do not have any specific units, due to the modification the gravitational constant, the maximum radius here covers the majority of the screen (any larger would be quite ineffective for simulating because the entire screen would be a single colour) and the minimum values is small enough so that any smaller would not be visible on the screen.

The delete button can be achieved quite simply with a text button, this button calls the `remove()` method as specified in the previous section with the currently selected CelestialBody. This will remove that body cleanly from the Simulation.

The velocity on the other hand is significantly greater challenge. Since it is not an action to perform, and it is not a scalar parameter. It is a vector parameter, one solution could be to implement it with two sliders, but that interface would feel clunky to the user and it would be difficult to gauge how much adjustment is necessary.

A more suitable solution is take advantage of the already implemented Velocity arrow on the screen during CelestialBody selection. Giving the user the ability to drag this velocity arrow would provide a simple interface to adjusting the velocity of the CelestialBody. To achieve this a modified version of the ArrowHead class was produced. This modified version includes a variable to store the CelestialBody that the arrowhead is for and a listener to track drag events and displace the velocity by the same amount as is dragged. This has been implemented much in the same way as the camera dragging was implemented and can be seen in a Listing below.

```
addListener(new DragListener() {
  private float dragX, dragY;
```

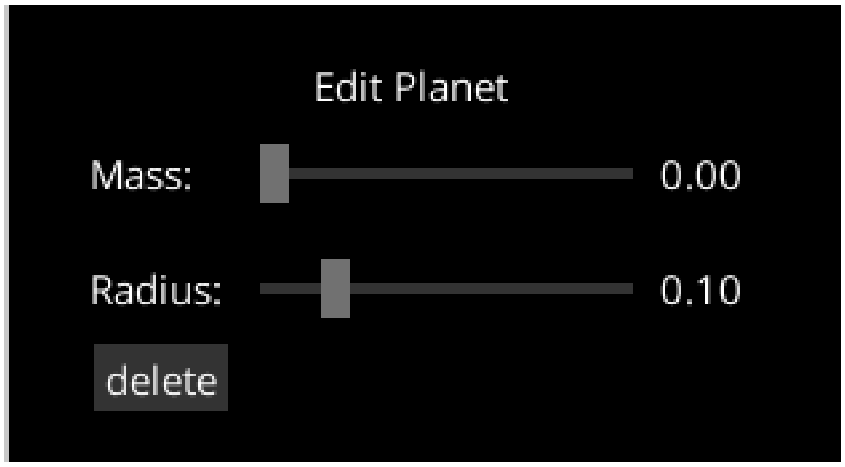


Figure 5.19: A screenshot of the edit menu.

```
  @Override
  public boolean touchDown(InputEvent event, float x, float y, int pointer, int button){
    if (button == Buttons.LEFT) {
      dragX = x;
      dragY = y;
      event.handle();
      return true;
    }
    return false;
  }

  @Override
  public void touchDragged(InputEvent event, float x, float y, int pointer)  {
    float dX = (float)(x-dragX)/(float)Gdx.graphics.getWidth();
    float dY = (float)(y-dragY)/(float)Gdx.graphics.getHeight();
    dragX = x;
    dragY = y;
    increment(dX,dY);
    event.handle();
  }
});
```

With all buttons now implemented the next step is render them in an appropriate way to the screen. This was done through an additional Table to collate all buttons. Using the table's structuring methods a suitable layout was produced with each of the sliders having labels added to them indicating their purpose and a second label at the end of each slider indicating its current value. A graphic of the edit menu is shown in Figure 5.19.

The entire table is made made non-visible when no CelestialBody is selected, to prevent the editor from covering too much screen in the simulation. This is achieved by adding a check in the render loop as a part of the `updateIU()` call.

CHAPTER 6

Testing and Results

The following chapter details the process undertaken to ensure that the final product completed all of the objectives as intended. Due to the incremental software development methodology, the program is clearly divided between a set of features. On this basis, the testing focused upon ensuring each feature operated effectively and in all circumstances. Along with the rest of this document, the features have been categorised into 'Simulation' and 'Interface'. Within these two categories, each individual feature has been tested by the most appropriate method. These testing techniques include: algorithmic correctness, through conservation of the laws of motion; manual unit testing; and use acceptance testing.

6.1 Simulation

Testing to ensure the correctness of the simulation model is a particularly difficult task, because it is not possible to construct a real world equivalent to test against. Therefore various laws of physics, that are known to be constant, were utilised to test the accuracy of the simulation, in addition to unit tests manually implemented by the developer.

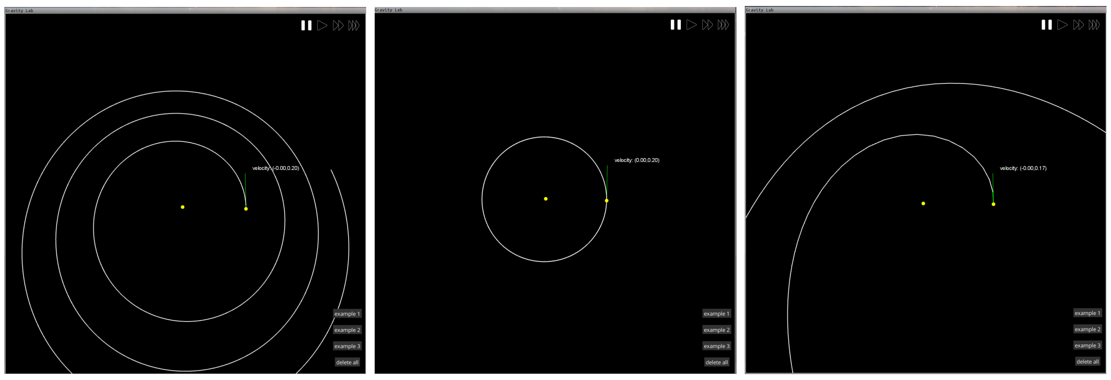
6.1.1 Oscillatory Stability

A test of oscillatory stability determines if an integration scheme will preserve circular motion indefinitely. Such a test will also reveal any system that will clearly never be able to conserve the amount of energy in a system on a global scale. The test for such stability would normally be quite challenging mathematically, however, because the simulation has a complete visualisation mechanism in place at this point we can determine whether or not an integration scheme is stable under oscillating motion.

The reasoning behind desiring oscillatory motion, is that orbits are a form of this motion. Therefore, a quick and simple test is to determine if it is possible to produce a circle or ellipse with the predictivecurve. Such a path is impossible in a system that does not preserve oscillatory motion, because the increasing energy will always produce a spiral either inwards or outwards. As can be seen from Figure 6.1 the results show that of the three implemented integration schemes, only one (Velocity Verlet) passed the oscillatory stability test. Hence the reasoning behind VV being the only integration scheme included in the final version of the program.

6.1.2 Conservation Tests

The total energy in any closed system is always conserved. In this simulation there are two forms of energy that exist: kinetic energy and gravitational potential energy. Every object in the simulation has its own quantity of these two energy types and can be computed for any one of them with the following equation:



(a) Explicit Euler (b) Velocity Verlet (c) Runge Kutta 4

Figure 6.1: Oscillatory Stability Tests

$$E_i = \frac{1}{2}m_i||\boldsymbol{v}_i||^2 - \sum_{i\neq j}^{N} \frac{Gm_i m_j}{||r_j - r_i||}$$

The total momentum in the simulation can then be represented as:

$$E_{total} = \sum_{i=0}^{N}(\frac{1}{2}m_i||\boldsymbol{v}_i||^2 - \sum_{i\neq j}^{N} \frac{Gm_i m_j}{||r_j - r_i||})$$

The total momentum along each dimension in any closed system is also conserved. The momentum vector can therefore be computed as:

$$\boldsymbol{p}_i = m_i\boldsymbol{v}_i$$

Consequently, the total momentum vector of the simulation is:

$$\boldsymbol{p}_{total} = \sum_{i=0}^{N}(m_i\boldsymbol{v}_i)$$

Therefore, the change in these two values from the beginning of the simulation will indicate the accuracy of the simulation. Using these equations, implemented in JUnit, the accuracy of the simulation was tested.

Firstly, the BruteForce implementation. Although the acceleration computation should not interfere with the positions or the velocity of any object in the simulation, it was found during the implementation of this objective that very minor errors could lead to inadvertantly adjusting those variables. Therefore, a JUnit test was constructed to instantiate a pair of bodies, both with mass, to compute their accelerations. It then tested that the energy and momentum before and after a run of the BruteForce algorithm were identical. This test was passed.

Secondly the integration method. The chosen integration method for the final product was Velocity Verlet. Like all methods of numerical analysis, this approach is not perfectly accurate. Therefore, we cannot expect it to perfectly conserve the energy within the system. However, to what extent these laws are conserved is a strong indicator of their effectiveness. The test performed against Velocity Verlet consisted of the same procedure as the BruteForce implementation, and Velocity Verlet produced an error of less than $10^{-9}J$ for energy and $10^{-9}ms^{-1}$ for each dimension of the momentum.

Finally in this section is the collision handling system. This part can only detect conservation of energy and so shall be tested more thoroughly later in this section. Rather impressively, the change in energy and momentum per dimension post collision resolution between two objects was less than $10^{-20}J$ and $10^{-9}ms^{-20}$ respectively.

6.1.3 Integration Tests

Although the conservation of energy and momentum provides a good indicator of numerical accuracy, it does not prove that the motions are realistic in their paths. For this, manual integration tests of the simulation system were employed. All performed tests passed. In particular, within Test 5, the test's purpose was to see that two bodies travelling directly towards each other would bounce off directly away from each other. Not only did they repel along the correct axis, the objects returned to a visibly indistinguishable point where they started, which implies that the GPE, that they started with, was converted to KE and back to GPE with negligible loss. A complete listing of the manual integration tests can be found in the appendices.

6.2 Interface

The testing of the interface consisted of two techniques: manual unit tests, individually testing the funcionality of the components and use case testing, placing the application in front of a user to observe any usage difficulties and obtain feedback as to the feel of the application.

6.2.1 Unit Tests

The unit tests consisted of listing out all possible use cases for several of the key features. Since the features have already been discussed so far in this document, the Unit tests have been renegated to the appendices. All unit tests performed passed, showing functionality of the interface is as expected.

Noteably, one bug was discovered in the duration of the testing, which the developer was unable to resolve. Which is, if the screen is dragged too far, the program will no longer register create planet events. The bug was inverstigated, and found to be a systematic problem in the LibGDX platform, actor objects cannot be larger than the window. So when the actor that recognises the create planet event, *Simulation*, is dragged off of the screen, click events can no longer be registered. Sadly, the systemic nature of the bug made it impossible to correct in the given time span. The bug itself is quite an obscure edge case and it was never observed during use case testing.

6.2.2 Use Case Testing

Use case testing against the interface was performed at multiple points during development. As the program incrementally gained in features a distributable jar file was circulated to those who were interested in the development of the project. This provided feedback that was utilised for early improvements within the project. Notable changes that occured due to use case feedback of the system included:

- Arrowheads on the end of the rendered vectors, to indicate that they were vectors.
- Changes from the drop down list format time menu, to the media buttons based time menu
- The rendered circle, and text whilst selecting the mass of the body.

These changes were already discussed in the Design and Implementation chapter, so they have only been mentioned here.

CHAPTER 7

Conclusions

In this final section, there will be discussion of the effectiveness to which the program implemented its requirements, through analysis of the complete application.

7.1 Evaluation of Requirements

Simulation Requirements

Accurate Model Acceleration between planets: The first requirement of the simulation was to accurately model the acceleration between planets, using the Velocity Verlet integration method and BruteForce gravity algorithm. The error in the conservation of the total energy in the system was found to be less than $10^{-9}ms^{-1}$. However, the use of the BruteForce method suffers from $O(n^2)$ time complexity, placing limitations on the number of objects that the simulation can handle on low power machines. Fortunately, these limitations are not significant, since no noticeable drop in framerate has been observed so far on up to 30 objects.

Support the creation of stable, unstable and perturbed orbits: The program is capable of supporting stable orbits, and unstable orbits. Furthermore, those orbits are perturbed by other masses in the system.

Predict where planets will be ahead of time: Whilst selecting a new object's velocity, a curve is drawn ahead of the object that predicts with identical accuracy (as it utilises the same code) to that of the simulation itself. Although this does achieve the requirement, it would have been a helpful addition for the program to predict the path of curves in more locations than just the velocity input (such as whilst an object is selected). Another issue present is the time complexity of this computation, because it relies on iterating the BruteForce method multiple times. The final time complexity is $O(nl)$, where n is the number of planets and l is the length of the curve. Reduction in framerates can be observed on as low as 10 planets with a 5000 step line.

*** Model Basic collisions between objects**: The program will detect and resolve collisions between any two objects as rigid body elastic collisions. It achieves this in $O(n^2)$ time complexity, and although this no effect on the simulation time complexity of the running of the simulation (since BruteForce is run in the same loop and also has $O(n^2)$), were the BruteForce algorithm ever replaced with a more efficient method, this would be the limiting factor of the simulation. However, it succesfully solves this requirement, in a non-limiting time complexity for the current state of the program.

Another concern with the collisions, is the choice of modelling them as elastic rigid body collisions. Although they do rather effectively display a very natural bouncing motion, it might have been more appropriate to follow in the footsteps of other software, by assuming the two masses combine to a single mass. But, this comes down to a matter of personal preference as to which more closely matches reality, as they both represent a different aspect of true collisions on that scale.

Interface Requirements

Visualisation of the Simulation: The visulisation of the simulation focused on a minimalist approach, and has widely received strong praise from user acceptance testing. In particular the trails that follow the objects as they move has a visually appealling aesthetic. Overall this requirement was completed very sucesfully.

Translatable and zoomable Camera: The camera was general implemented quite well, with appropriate controls. However, the lack of indication that it exists was a slight issue in user acceptance testing. And it suffers from a slight bug, that prevents the creation of planets after being dragged too far. This bug has generally not been noticed by users of the program, but it is still present and would have liked to have been fixed. Sadly, the issue is with LibGDX itself, and cannot be worked around with the current structure of the program.

Time Menu to play, pause and adjust speed: The time menu also received wide praise from user acceptance testing, the changes made from the initial version has produced a very obvious interface to adjust the time, and all those who use the application have not required any form of prompt to understand how to use it.

Tool to create new objects, with specified mass, velocity and position: Creating the bodies generally has succeeded to reasonable effect. The interface works entirely as specified, however it could benefit from some improvements. Firstly, rightclicking on each selection is not entirely obvious (the help text had to be implemented to make it obvious). Moreover, one could argue the create body tool itself is over engineered. Once the edit body menu had been implemented, it could be more effective to only create the object, then immediately select the body automatically, so the use can edit it from there. However, this would require moving the path prediction logic into the edit body menu, for the when the velocity is dragged.

Prediction of the initial path of the body while selecting these values: The predictive curve interface received quite a lot of praise through user acceptance testing. It was found to be a very useful feature and one that many users did not expect to exists, because no other software available implements it. Although the time complexity of the computation of the predictive curve leaves more to be desired, when the predictive curve does slow down, it does not make the program unusable, it merely lowers the framerate of rendering and most users have been comfortable with using it, even when it is updating slowly in circumstances where the velocity is being selected after more than 10 bodies exist.

Tool to edit objects: The editor requirement was to produce a menu that could be accessed for each object, with tools to adjust the velocity, mass and radius of the objects and also to delete that object. This requirement is satisfied by the selection model, that causes the edit menu to appear once clicking upon a planet. The sliders were quite effective at adjusting mass and radius, although the maximum and minimum sizes of the mass and radius could be considered a limitation. An improvement to this might be to add a number box next to the sliders for alternate values. The velocity adjustment mechanism (with dragging of the vector head), was considered a wide success. It allows for simple manipulation of the direction of the object, and can even be used while the simulation is running.

Information about each Body: Provided whilst an object is selected, the acceleration and velocity vectors accurately display exactly the properties of the object. This requirement worked quite closely with the previous requirement discussed here. Through utilising the velocity vector of this system to both display its current value and allow changes to new values. One issue that could be considered in this section is that of colours, the only differentiation between the acceleration and the velocity is that they are red and blue respectively. Although these colours were chosen to avoid colourblindness issues, it would have been nice to also include some other indicator of which one is which, and a person who has little experience of the two vectors, might have difficulty distinguishing them through the context of their positions.

Distortion field below the plane: This was one of the few stretch goals that did not eventually get implemented. The reasoning behind it not being implemented, was that it was a very time consuming feature to preduce, with very little payoff in the final product. Although it still would have been nice to have been implemented. But looking back on the project, the developer feels it would not have been possible to implement within the time constaints of the project, and the application does not suffer due to its lack of existance.

7.2 Summary

This project has successfully completed all of its primary objectives, and it is reasonable to conclude that it has also achieved its aims through this set of requirements. With a complete, distributable application that can succesfully simulate the motions of celestial-like objects and an interface that those who have experienced it, have found aesthetically pleasing and reasonably user friendly. The project was completed largely on schedule and implemented several of its stretch goals. The potential for expansion to the application is discussed in the following section, for items that can be completed to further enhance this application outside the scope of the project.

7.3 Further Work

In this final section we shall discuss the potential of further work being completed in this software. The program itself consists of implementations of numerous different concepts of simulation theory and interface design, so there are many ways in which this program can be expanded in the future. What follows is a list of suggestions as to how to expand this application:

This application could be deployed as a java applet to a suitable website, removing the barrier of downloading from the end user. This process would probably not be particularly complex as the LibGDX/Gradle framework has built in support for such deployment.

The acceleration computation algorithm could be upgraded through use of Barnes-Hutt, an algorithm that can reduce the complexity down to $O(nlog(n))$, it was avoided in the original version of the program due to its significantly higher algorithmic complexity. But an implementation of it could potentially reduce the overall runtime of the simulation, and therefore reduce the impact of the issues faced with the predictive curve.

The potential of a Velocity Verlet Runge-Kutta combination integration scheme is intriguing, and several papers have been discovered as to other methods to introduce oscillatory stability into Runge Kutta. Such a scheme would provide significantly higher accuracies in simulation, and is worth investigating.

The collisions handling system could be improved through two ways: firstly by using an algorithm that detects collisions within $O(n)$ time (which is possible through techniques such as horizontal sweeping). Secondly, a more realistic collisions handling system could be implemented, perhaps one that allows the shattering of objects into multiple pieces at high speeds and the merging of objects at low speeds.

Bibliography

[1] S. J. Aarseth. *Gravitational N-Body Simulations.* D. Reidel Publishing Company, 2003.

[2] N. Arkani-Hamed, L. Motl, A. Nicolis, and C. Vafa. The string landscape, black holes and gravity as the weakest force. *Journal of High Energy Physics*, 2007(06):060, 2007.

[3] J. Banks. *Discrete-event system simulation.* Prentice Hall, 2001.

[4] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 1324:446–448, 1986.

[5] F. Calogero. Solution of a three-body problem in one dimension. *Journal of Mathematical Physics*, 10(12):2191–2196, 1969.

[6] R. M. Canup. Simulations of a late lunar-forming impact. *Icarus*, 168(2):433–456, 2004.

[7] A. Cook. Gravity. `https://www.britannica.com/science/gravity-physics`. Accessed: April 22, 2017.

[8] J. W. Cooper. *Java design patterns: a tutorial.* Addison-Wesley Professional, 2000.

[9] C. P. H. . C. A. C. D. Investigation of the motion of halley's comet from 1759 to 1910. *Greenwich Observations in Astronomy Magnetism and Meteorology made at the Royal Observatory*, 71:2 – 10, 1910.

[10] A. Einstein. On the electrodynamics of moving bodies. *Special Theory of Relativity*, page 187218, 1970.

[11] F. A. El-Salam, S. A. El-Bar, M. Rasem, and S. Alamri. New formulation of the two body problem using a continued fractional potential. *Astrophysics and Space Science*, 350(2):507–515, 2014.

[12] L. Euler. *Institutionum calculi integralis*, volume 1. imp. Acad. imp. Saènt., 1768.

[13] J. Kepler, E. J. Aiton, A. M. Duncan, and J. V. Field. *The harmony of the world*, volume 209. American Philosophical Society, 1997.

[14] J. Kovalevsky. *Introduction to Celestial Mechanics*, volume 7. Press Syndicate of the University of Cambridge, 1967.

[15] R. Spurzem. Direct n-body simulations. *Journal of Computational and Applied Mathematics*, 109:10, 1999.

[16] R. J. Stephenson. *Mechanics and properties of matter.* John Wiley & Sons, 1969.

[17] E. Verlinde. On the origin of gravity and the laws of newton. *Journal of High Energy Physics*, 2011(4):1–27, 2011.

[18] C. Voesenek. Implementing a fourth order runge-kutta method for orbit simulation. pages 1–3, 2008.

Appendices

APPENDIX A

Previous Documents

A.1 Specification

2D Gravitational Simulation of Celestial Objects

Specification

Edward Compton 1402754

October 2016

1 Problem Statement

The study of celestial mechanics can be quite challenging for students, because it relies on the comprehensive understanding of multiple challenging concepts. Unlike most other forms of mechanics, students who are studying it cannot perform experiments to assist in their learning and performing experiments is a fundamental method of problem solving. It gives students the opportunity to play with the components in the system, understanding how they work.

Celestial mechanics is the study of the way in which massive objects interact through gravity, and the interesting patterns of motion that gravity can produce. The aim of this project is to produce a tool that will allow students to experiment with a few of the topics within celestial mechanics:

- Basic Gravity: Massive objects accelerate towards each other.
- Orbits: Circular motion around a massive body, including stable and unstable orbits.
- Perturbations: Multiple orbits in a single system can alter the paths of each other.
- Other various intricacies involved in a gravitational system that are captured by an N-Body simulation.

If there is sufficient time towards the end of the project, the simulation can be expanded to simulate simplified versions of the following topics:

- The Gravitational Field: The concept that gravity is not actually a force but the bending of the space that massive objects occupy to cause an effect similar to a force.
- Collisions: Resolution of the way massive objects collide into one another with gravity.

2 Objectives

This tool will allow users to create a customisable simulation of massive objects. Development will be divided into two sections, the interface that the user uses to create the simulation and the program that runs the simulation. The objectives are divided into 3 categories:

- Core-Objective: Necessary for the solution of the problem.
- Sub-Objective: Not necessary for solution but will improve usability of the program. Marked with (*).
- Extension-Objective: Additional feature that will be added if the project has enough time. Marked with (**).

2.1 Interface

The requirements for the interface are as follows:

1. An editing menu that allows users to create objects with a selected start position, mass and velocity.
2. A predictive curve of the initial motion of the planet displayed while selecting velocity of planet on creation.
3. A menu with buttons to start, pause and alter speed of the simulation.
4. * A camera that be translated and zoomed for better view of the simulation.
5. * Information about each planet (velocity, acceleration acting upon them, etc) available by clicking.
6. ** Upgrade the camera to be able to rotate the simulation in all 3 axes.

2.2 Simulation

The requirements for the simulation are as follows:

1. Implement a 2D N-body simulation that iterates the new locations of planets for each time-step by calculating the experienced acceleration of gravity.
2. * The N-Body Simulation should be able to simulate approximately 100 objects on consumer grade hardware.
3. * Apply lighting, shading and textures to the objects to make it more appealing to the user.
4. ** Upgrade 2D representative circles to 3D spheres representing Stars/ Planets/ Asteroids/ Comets.

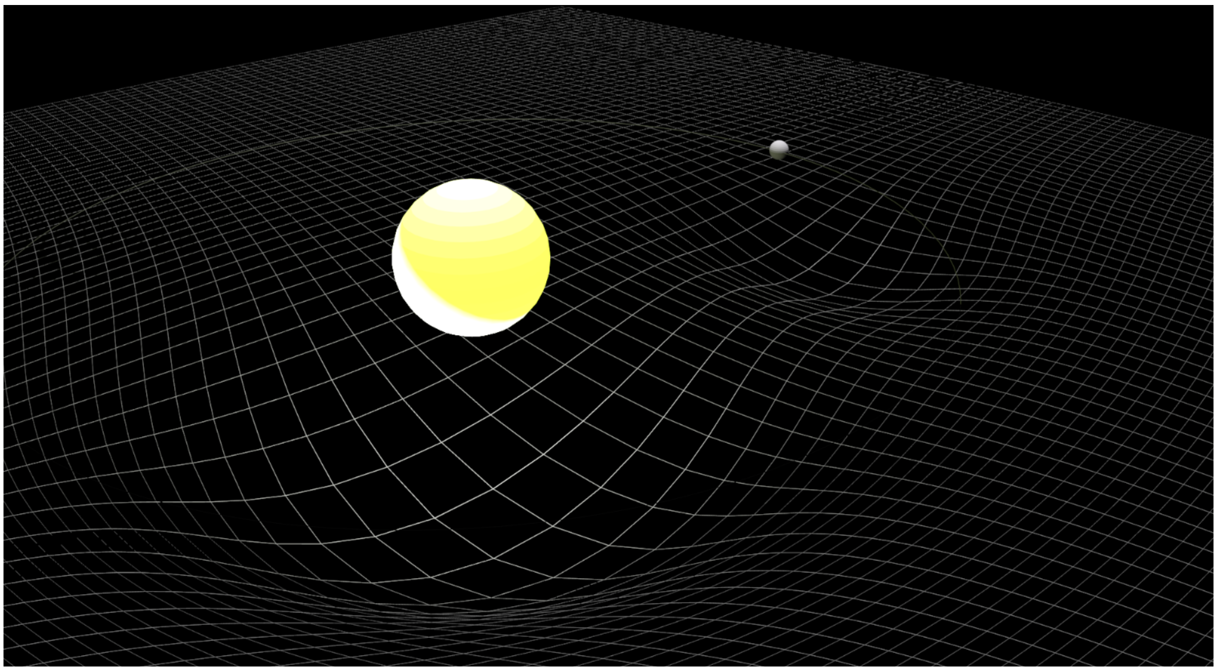


Figure 1: Visual Representation of Distortion Field

5. ** A field beneath the system representing the distortion of space due to gravity.
6. ** A collision event resolution system, that resolves collisions assuming that they are rigid and elastic.

3 Methodology

This project shall be using the iterative software development model to produce new versions, because each of the requirements have isolated into distinct objectives. In each cycle of development a new version of the program will be completed with an additional feature. The new program will then be tested and corrected to remove any bugs before moving onto the next objective.

4 Timeline

The N-Body algorithm is a core part of this program and also one of the most complex, so it shall be tackled first for about 2 weeks. After that, the project will shift focus to the interface, creating all of the necessary menus for the user to interact with the program. This is with the hope that the program is essentially usable by the progress report and the remaining time can be spent on improvements and other additional features.

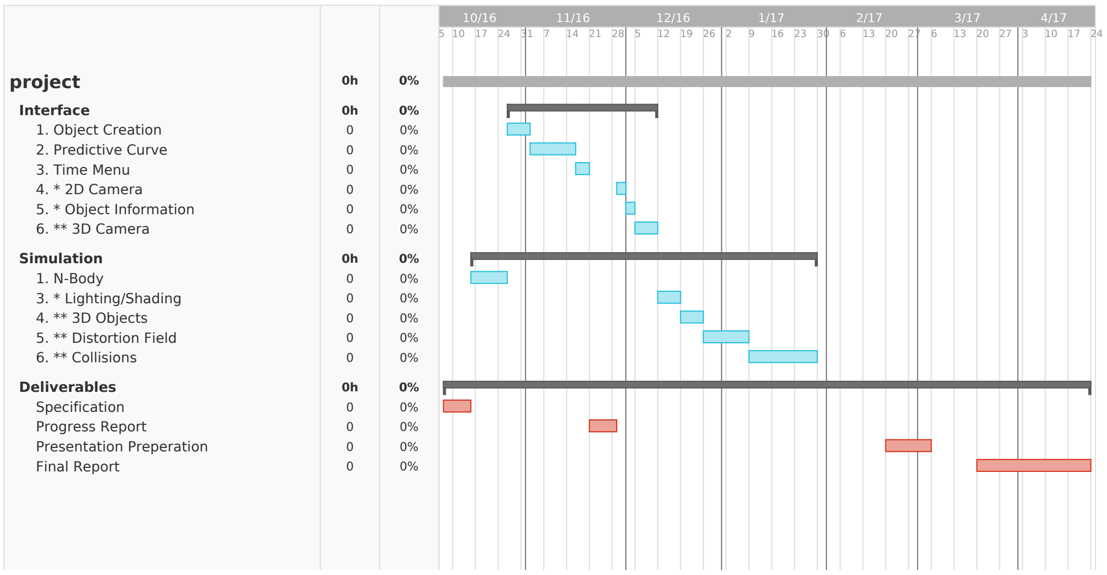


Figure 2: Gantt chart representing the timeline of the project

However if unforeseen technical problems arise during the first term, this gives plenty of time during the second half of the project to complete the basic objectives.

5 Risks

Risk 1: Data lost due to hardware faults
Solution: The Software will be backed up regularly to multiple devices and stored on a github repository.
Risk 2: The project is too taxing on hardware to properly run.
Solution: The program can be thoroughly optimised using code re-factoring techniques and additional taxing features can be toned down or removed. Since there is significant research already done on the topic, there should be available resources to improve the running of the code, including a research project on the optimisation of N-Body systems done by a student at Warwick last year.
Risk 3: The mathematics involved in some of the extension goals prove to be too complicated for the time given in this project and my technical abilities.
Solution: I could ask to interview researchers in the physics department for guidance in the production of these features, or in the worst case scenario: the features can be simplified or not incorporated into the final product if they are too difficult to produce.

6 Resources

The program will be written in Java using OpenGL and the LibGDX framework to support vector calculations, rendering and event handling.
Version control and regular backups will be made to Github to preserve previous versions of the program and as recovery in the event of data loss.
The underlying mathematics of the system can be researched through online articles, books available in the library and interviewing specialists at the university if necessary.

A.2 Progress Report

Developing a Visualisation of Gravitational Motion

by

Edward Compton

Department of Computer Science

University of Warwick

2016–17

supervisor

Dr Mike Joy

Abstract

This document is an analysis of the current state of the Project, it describes: the research carried out, the objectives that have so far been completed and any alterations that may be necessary to the project timeline to ensure that all objectives are complete. The overall aim of the project is to build a sandbox style simulator of solar system scale celestial objects.

The latex templates used to make this document were created by Matthew Leeke, University of Warwick Computer Science Department.

I would like to thank Matthew Leeke and John Pennycook for allowing me to use his Bruteforce N-Body implementation as a reference point to help me to understand how N-Body algorithms work.

Glossary

Celestial Body - A massive object with a size between a small asteroid and the largest of stars.

Body - A simulated representation of a celestial body.

Celestial Mechanics - The study of the motion of celestial bodies.

Sandbox Style - A category of video game with little to no form of structured play.

Orbit - A gravitationally curved path of an object about a point in space.

Stable Orbit - An orbit whose motion is stable enough to maintain itself indefinitely unless outside interference occurs (e.g. a collision).

Unstable Orbit - An orbit that will not maintain itself indefinitely even without external interference.

Perturbation - The alteration of the path of a celestial body's orbit around a central mass by a third massive object.

Conic-section - A curve that is either parabolic, elliptical or circular.

Time Step - The simulation time between the last set of plotted coordinates and the next set of plotted coordinates.

Context Menu - A small pop-up menu that appears next to the mouse.

Notations

N	is the number of bodies in the simulation.
a_i	is the acceleration vector acting upon the i^{th} body.
v_i	is the velocity vector of the i^{th} body.
r_i	is the position vector of the i^{th} body.
m_i	is the mass of the i^{th} body.
G	is the gravitational constant.

Contents

List of Figures

CHAPTER 1

Introduction

The study of celestial mechanics can be quite challenging for students, because it relies on the comprehensive understanding of multiple challenging concepts. Unlike most other forms of mechanics, students who are studying it cannot perform experiments to assist in their learning and performing experiments is a fundamental method of problem solving. It gives students the opportunity to play with the components in the system, understanding how they work.

Although the real components of celestial mechanics cannot be experimented upon, we can develop approximate simulations that can be. A simulation for students just starting to learn the very basics of celestial mechanics could provide very useful insight into the way these objects move.

1.1 Project Aims

The aim of this project is to produce a sandbox style simulation of celestial bodies. This problem can be split up into two main categories: the simulation and the encapsulating interface.

Since the time of this project is limited some requirements are listed as stretch goals. These are highlighted with an *. The stretch goals are items that will improve the tool but are not necessary for its completion.

The requirements for the simulation:

1. Accurately model the acceleration of one object to another due to gravity.
2. Support the creation of stable, unstable and perturbed orbits.
3. Model basic collisions between celestial objects.

The requirements for the interface:

1. Visual display of the current state of the simulation.
2. Camera that can translate and zoom to better see the simulation.
3. Time menu to play, pause and adjust the speed of the simulation.
4. Tool to create new planets with specified Mass, Velocity and Position.
5. Predict the initial path of the body while selecting these values.
6. Tool to edit planets values.
7. Possible graphical improvements:
   - Setting to print the vectors of the bodies as the simulation runs.
   - Settings on each body to make it a star, planet or possibly black hole.
   - Animations for the camera.
8. Distortion field below the plane of the Celestial Bodies (note this requires the simulation to be in 2D) .

At this point in the project (7 weeks in) most of the primary requirements have now been completed, notably the Simulation. The rest of this document details the research, design and implementation that was done and how what needs to be done moving forward.

CHAPTER 2

Research

2.1 Orbit Modelling

The fundamental part of this project is orbit modelling. It is the process of simulating the motions of Celestial Bodies. As a part of this project, background research has been undertaken into the underlying mathematics.

2.2 The 2-Body Problem

The 2-Body problem is a simulation of 2 celestial bodies, usually with one orbiting the other. Using Keplers laws of motion, the 2-Body problem can be modelled. Kepler deduced that using just 6 orbital elements (constant values) an orbit can be described as a conic-section. All orbits within the 2-body problem are conic-sections (a circle, ellipse or parabolic curve)[6].

This solution could be very efficient to compute, since the simulation would just have to compute the 6 orbital elements at the beginning of the simulation and then iterate the planet through the path of the orbit. Then a function could determine the angular velocity of the planet at each point in the curve.

2.3 The N-Body Problem

Since the intent of this project is to produce a simulation that works for more than just 2 bodies, an option would be to have a single fixed body in the centre of the simulation and model multiple Keplarian orbits around the fixed body. This can been used for more basic simulations, but it has the problem that perturbations are not taken into account.

Perturbations are when the number of celestial bodies with non-zero mass is greater than 2. Consider an orbit where there is one central mass A and two orbiting masses B & C. As B orbits the central mass A, its motion can be influenced by the gravitational attraction of C. The vector function representing this

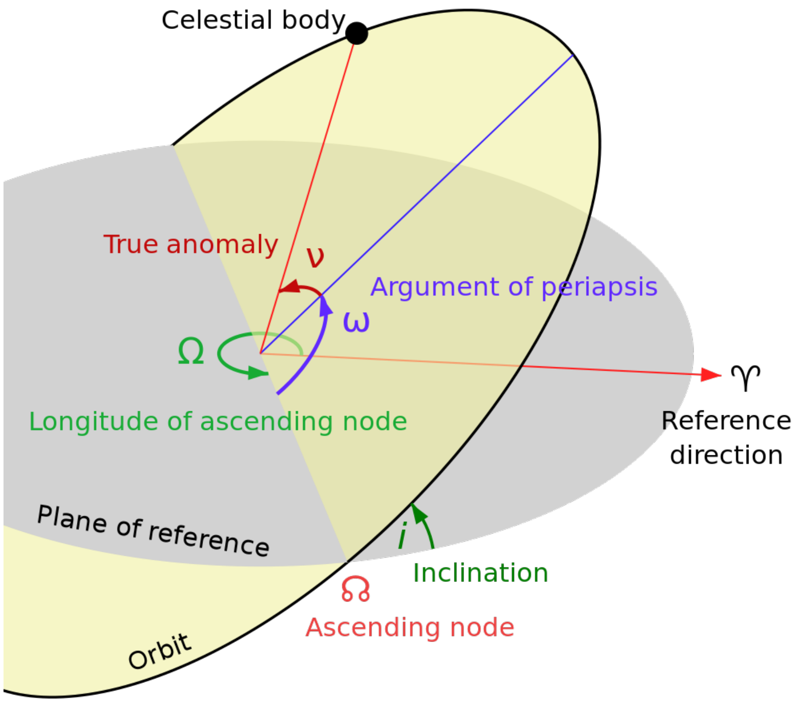


Figure 2.1: The orbital elements within a two body problem. [7]

change from the standard Keplarian orbit is referred to as the perturbation of the orbit. Without computing the perturbation of each orbit, the simulation can become drastically different from reality. As such, the ability to calculate perturbations has been added to the requirements of this project and an algorithm must be found that takes them into account.

Therefore a different approach is necessary: instead of computing the orbit itself, one considers the acceleration of gravity acting upon each celestial body and determines how the properties of the object has changed. The properties in this case required for each CB are [1]:

- m_i : The mass of the CB
- r_i : the last position of the CB
- v_i : the last velocity of the CB

Using this set of values between each timestep, the next set values can be determined. There are two possible algorithms to compute all of the new positions.

The simpler, but also the less efficient of the methods is the direct method (often referred to as Cowells formulation, named as such for Phillip H Cowell who used this formulation to predict the return of Halleys comet)[4]. Calculating the acceleration of gravity on each body by using the equation described in [8].

$$\mathbf{a}_i = \sum_{j=1}^{N} \frac{Gm_j(\mathbf{r}_j - \mathbf{r}_i)}{|\mathbf{r}_j - \mathbf{r}_i|^3} \tag{2.1}$$

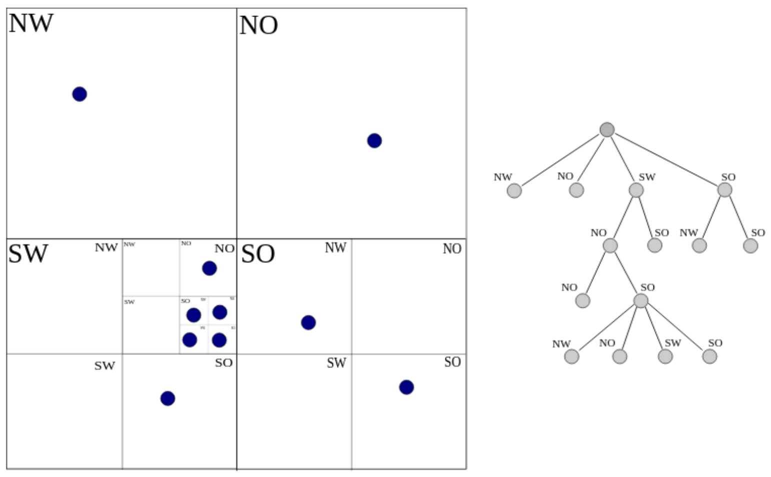


Figure 2.2: Diagram of a quadtree from the Barnes-Hut algorithm.[3]

```
for each CelestialBody i do
  for each CelestialBody j do
    acc = G * j.m * (j.r - i.r)
  end for
end for
```

This formula has a time complexity of $O(n^2)$. However the second method, treecode, has an efficiency of $O(nlog(n))$ [2], but it is more complex to implement. The algorithm uses the following steps:

1. Divide the simulation space into a quadtree by splitting the space into 4 sections. Every quadrant that contains more than 1 body is subdivided into another 4 sections. Each quadrant is recursively subdivided until every square contains just 1 or 0 bodies.

2. Then the tree is recursively traversed, computing the total mass and centre of mass for every node (at this point empty leaf nodes are pruned from the tree).

3. Finally, for every body the forces acted upon it are approximated by traversing the tree, where if a nodes centre of mass is more than a certain distance from the particle, then that nodes centre of mass and mass are used to calculate the force instead of its children. [2]

The reduction in time complexity mainly comes from part 3, where the fact that very distant clusters of stars can be assumed to be a single point mass at the centre of mass of the cluster. However this algorithm is very complex to implement and requires that there are a large number of bodies in the system (in the region of thousands of bodies). Since the simulator this project aims to create is merely at the solar system scale, with a maximum of about 30 bodies (although the program will allow for as much as the computer can handle), there is no need to implement it and the Brute-force method will suffice.

2.4 Integration Methods

The previous algorithms were both methods of determining the forces acting on each body. The next step is to use the force to calculate the new velocity and position. This, at first glance seems simple. However time, in reality, is not split into discrete chunks. As the particles move through the system their acceleration continuously changes, which is a problem when computing over timesteps because it assumes that the acceleration is constant for the duration of the timestep. Due to the coninuous change in the acceleration, the equations of motion will need to be numerically integrated.

$$\mathbf{s} = \mathbf{u}dt + 0.5\mathbf{a}t^2 \tag{2.2}$$

There are many possible integration methods, the simplest of which is Eulers explicit integration.

```
Velocity += acceleration * dt
Position += velocity * dt
```

This allows the position to be updated from the most recently integrated value of velocity instead of computing it directly from the equation of motion.

Other more accurate methods including Implicit Euler and Runge-Kutta also exist, and if the testing of the simulation proves this method to be too inaccurate they will be implemented instead.[5]

2.5 Singularities

An issue present in all collisionless n-body simulations (unless it is specifically handled) is the singularity problem. As bodies approach each other, the Newtons force equation scales to a disproportionately large size (because the particles are point masses they can get much closer to each other than they should be able to). This causes bodies to increase to much higher velocities than they are supposed to be able to, breaking the conservation of energy. [10][9]

One solution to this problem is called dampening. A small value (about the width of the average planet) is added to the distance between the two bodies just before the force is calculated. This removes some of the much large possible forces experienced by a planet, but adds inaccuracy to the system. For now the dampening solution will be implemented, however a more effective solution is to create an accurate collision event handler. This is a stretch goal of the project and will only be implemented if there is enough time.

2.6 Frameworks

The LibGDX framework is used in this project. It is programmed in Java and based on OpenGL with multiple useful libraries to assist in the creation of the project (like text and shape drawing object). The heart of libGDX is the event handler. It resolves user and system events to change the state of the program.

Notable methods include create(), which is run when the program is launched, and render() which is run at the frame rate of the program times per second (the target frame rate can be set in a config file).

CHAPTER 3

Design and Implementation

3.1 Completed Objectives

3.1.1 Simulator

The first step in implementing any n-body algorithm within an OO language is to create a class for each of the Bodies. The CelestialBodies object is therefore proposed:

```
class CelestialBody {
    private float m;
    private Vector2 r;
    private Vector2 v;
    private Vector2 a;
}
```

This object requires an acceleration, velocity, position and mass to be stored. The first three are 2D vectors, so the $Vector2$ object provided by LibGDX is adequate. The mass is just a decimal so a $float$ will suffice.

The next step is to decide how to store the objects. Since they will almost always be called sequentially, but the additional objects could be added at any time, the ArrayList java object will be adequate. The collection will be created and populated with CelestialBody objects in the Simulator class.

During the algorithm, each planet must compute its new velocity and position by calculating all of the accelerations upon it, therefore the $compute()$ method is proposed, that takes the time-step and collection of planets as inputs before calculating its own new position. This avoids the necessity of making the global variables of the class public.

```
public void compute(ArrayList<CelestialBody> bodies, float dt)
{...}
```

Once this function has been implemented, it is simply a matter of getting the Simulator class to run it. Consequently, another *compute*() method is included in the Simulator class, that iterates through all the Bodies running their respective *compute*() functions.

```
public void compute(float dt) {...}
```

This fulfills requirements 1 and 2 of the simulation, as described in the project aims.

3.1.2 Interface

With the mathematical side of the simulator completed, the next task was to display it on the screen. The LibGDX ShapeRenderer object can draw circles on the screen so it is adequate for this task. Each body is printed as a circle, where the radius of the circle is proportional to its mass and the position of circle is scaled to the viewscreen coordinates.

To improve the visual fidelity of the motion of the planets, a tail has been added to the motion of the planets. This is a simple queue that stores the last 50 coordinates of each bodies position. Then the rendering function cycles through each queue drawing lines between each subsequent coordinate with slightly lower alpha and line thickness to create a tail that follows the body. Due to the way the tail is computed, its length is proportional to the speed of the body.

Having useable tools in the program creates an interesting problem in the event driven model; the program must be in certain states during the usage of each tool (e.g. once a body has been created it the user needs to give it a mass). A way to handle this is to model the program as a State Machine, where user events cause state transitions.

This graph will almost certainly be expanded as more features are added, but for now it has 4 states. Switch operators have been added to every event that changes depending on the state.

Body Creation Tool:

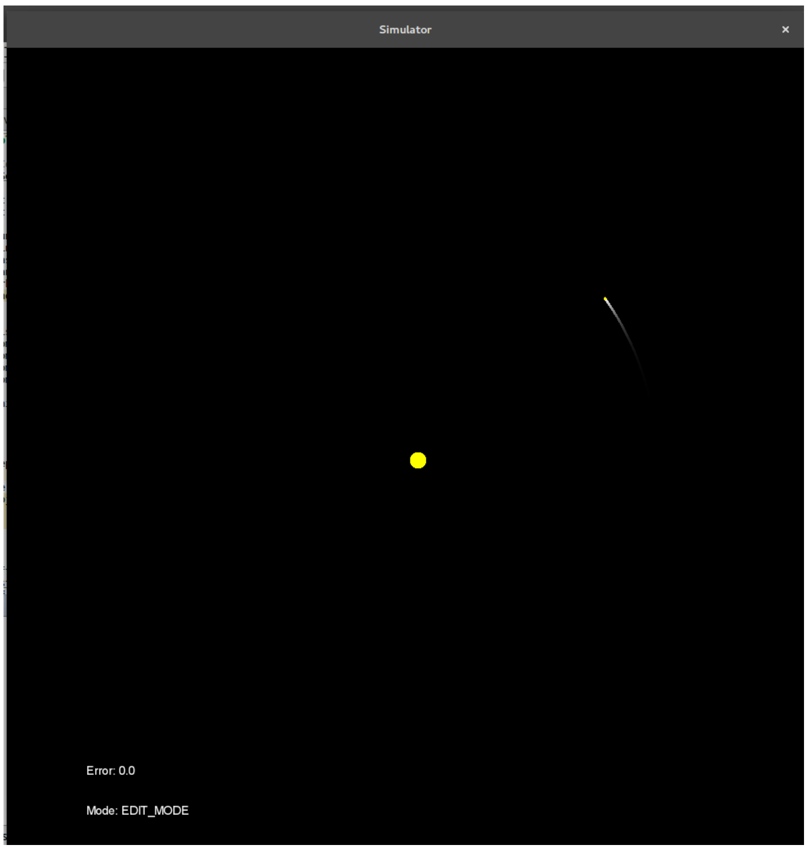
1. Starts in Editmode, mouseclick creates new body in location that it was clicked. Changes to Get_mass

2. While in get_mass mouse-moved events change the mass of body to the distance of the mouse from the body. Mouseclick events fix this mass and switch to get_velocity.

3. While in get_velocity mode, mouse-moved events change the velocity to the vector of the body to the mouse and draw predictive curves. Mouseclick events fix this velocity and switch to back to edit mode.

Part of the body creation tool, but significant enough to merit its own requirement, is the predictive curves calculation. This is a curve predicting the future path of the new planet. It is calculated while the velocity is being selected, so at this point the mass is already chosen, but the velocity is changing on every mouse movement. Which means that a new predictive curve must be calculated on every mouse movement.

In order to calculate it two new arrays are proposed, contained within Simulator:



Error: 0.0

Mode: EDIT_MODE

Figure 3.1: Screenshot of the program displaying the rendered bodies, one with a position tail.

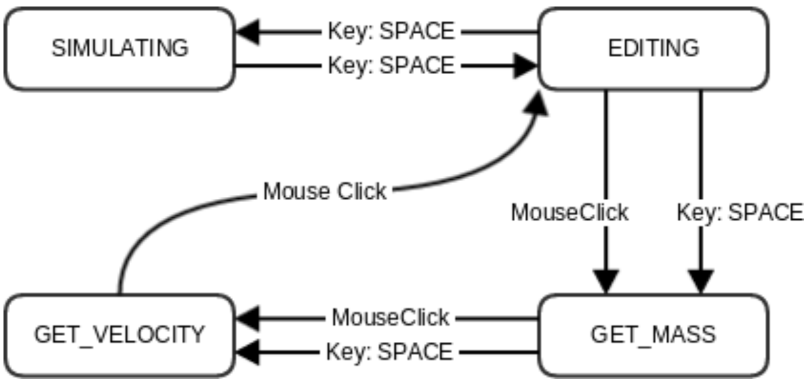


Figure 3.2: A statemachine representing the way tools are handled.

```
    private ArrayList<CelestialBody> psuedobodies
    private ArrayList<Vector2> predictivecurve
```

On every mouse movement whilst in the get_velocity state:

1. The psuedobodies array is populated with the bodies array
2. The CelestialBodies are incremented over 5000 timesteps, on each step adding the position vector of the body being created to the end of the predictivecurve Array.

The render method draws a sequence of line segments connecting the 5000 position vectors together.

Since the brute-force method of computing new coordinates has time complexity of $O(n^2)$, this method of calculating the predictive curve has time complexity of $O(mn^2)$. Testing on physical machines has found this to be perfectly fine with about $N = 20$. However mousemove events in libgdx stack along every pixel that the mouse has touched, so when the mouse is dragged across the screen over a large number of pixels the algorithm then becomes $O(omn^2)$ where m is the number of timesteps (5000) and o is the number of pixels traversed. This becomes a problem and drops the framerate to almost nothing until the machine recovers (which is only a fraction of a second). This should be changed later on in the project, but for now is functional.

This satisfies the body creation tool requirement. However if the additional features to the Bodys are to be implemented later on such as texturing this system might change to a context menu. Especially once the editing of planets has been implemented, they could be combined to a single context-menu.

Time Menu: So far, only the play/pause functionality has been added to the program. This is simply a hotkey that switches between simulating and edit-mode using the statemachine (disabled while creating planets). This will probably be fully implemented with Play (Normal Speed), Play (Faster Speed), and Pause buttons on the screen, later on in the project.

3.2 Future Objectives

3.2.1 Time Menu

The next stage for the time menu is to support different speeds of simulation. This could be done by adjusting the dt value stored in the Simulator object. Following this buttons could be created, possibly of a similar design to the game Mini Metro seen in figure /reffig:mini.

3.2.2 Camera

This task should be able to be completed just by using inbuilt features of libgdx. However the most natural way to drag (translate) the view is usually by using the left click on the mouse, so this will require changes to the controls for the planet creation tool, either by switching between the dragging tool and the creation tool or by setting the creation tool to something like right-click.

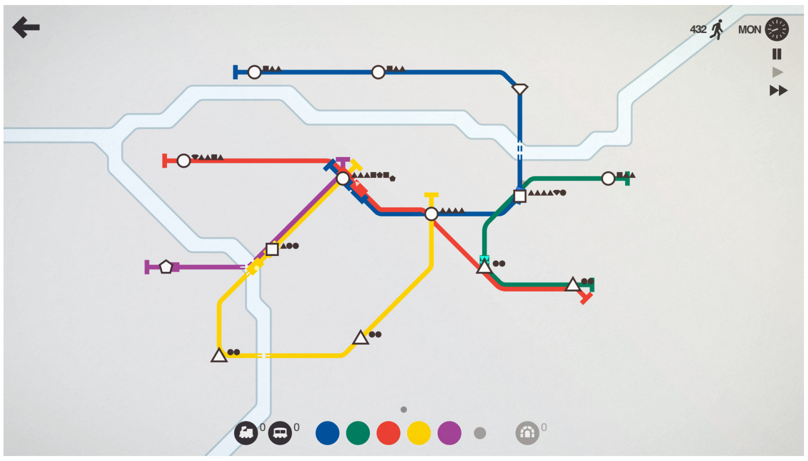


Figure 3.3: Example of a simple minimalist time menu from the game Mini-Metro

3.2.3 Edit Body Context menu

The new context menu to edit planets also wrestles control for the right-click/left-click on the mouse. It will also necessitate additional parameters to the Celestial Body object such as texture (from a preset list of textures). Adjusting the velocity of the object can be done in the same way as it is done in the creation tool, however there might be a more user friendly way to select the mass as it is currently not particularly user friendly (perhaps a slider within the context menu).

3.2.4 Graphical Improvements

By this point the project will be functionally complete, but there are other improvements to consider. For example, a setting to print the vectors, which could be selected in a settings menu. A settings menu could also include resolution options, and graphical settings if lighting and shading is included when the different texturing options are added (e.g. Light produced by a star). Then it would also be useful include animated camera transitions to zoom in on bodies when they are being edited and a button to zoom to see all bodies at once.

3.2.5 Collisions

Detecting and resolving collisions is one of the more ambitious goals. That said the Box2D tool included within Libgdx could be useful. Research would have to be done into how other N-Body simulations handle collisions.

3.2.6 Distortion Field

Creating the distortion field could, in theory, be quite simple, drawing a mesh below the solar system that is distorted proportional to the amount of gravity experienced in that area. Drawing lines as an altitude graph could provide a good solution to this problem. However it will require the camera to also rotate about the center of the simulation and significant research will need to be done to show that this mesh is actually representative of the way gravity distorts spacetime in a relativistic environment. Although, at this scale, relativistic effects are not significant enough to alter the simulation.

CHAPTER 4

Project Management

So far, the project has progressed quite well. All objectives have been met on the previous timeline except one (the time-menu), which was the last objective and will be implemented as soon as the progress review has been submitted. One problem identified is that there was not any testing, particularly within the simulation, specifically planned at the beginning of the project. However there are simple methods to do this (such as summing the energy and angular momentum within the system and testing that they are constant). Moving forward, there will be Unit testing through JUnit for all future implemented code, and Unit testing will need to be done on previous tasks as well. Another identified problem is that I personally struggle to write large documents within short time spans (E.g. the week allocated to create this document). In the future it will be a requirement for each objective to have written supporting documentation ready to be incorporated into the final submission and presentation.

There are 13 weeks left until the presentations start. Consequently the objectives have been divided into weekly chunks between now and then to free up the entire time between the presentations and the final report submission for report writing. A week buffer gap has also been added to deal with any delays to the project. To conclude, planned objectives are outlined in figure 4.1 and all of these objectives must be Unit tested with supporting documentation written for them to count as completed.

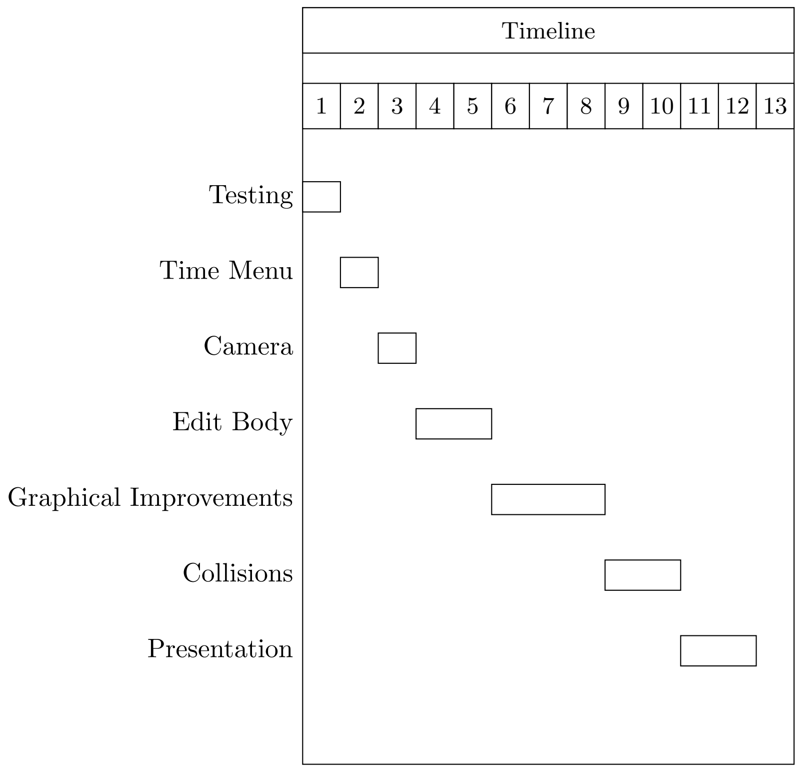


Figure 4.1: Gantt Chart to represent the future timeline of the project.

Bibliography

[1] S. J. Aarseth. *Gravitational N-Body Simulations.* D. Reidel Publishing Company, 2003.

[2] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 1324:446–448, 1986.

[3] I. Berg. The barnes-hut galaxy simulator. `http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator`, 2016. Accessed: 2016-11-27.

[4] C. P. H. . C. A. C. D. Investigation of the motion of halley's comet from 1759 to 1910. *Greenwich Observations in Astronomy Magnetism and Meteorology made at the Royal Observatory*, 71:2 – 10, 1910.

[5] G. Fiedler. Integration basics. `http://gafferongames.com/game-physics/integration-basics/`, 2006. Accessed: 2016-11-27.

[6] J. Kovalevsky. *Introduction to Celestial Mechanics*, volume 7. Press Syndicate of the University of Cambridge, 1967.

[7] Lasunncty. Digram illustrating and explaining various terms in relation to orbits of celestial bodies. `https://commons.wikimedia.org/wiki/File:Orbit1.svg`, 2007. Accessed: 2016-11-27.

[8] R. Spurzem. Direct n-body simulations. *Journal of Computational and Applied Mathematics*, 109:10, 1999.

[9] D. Tu. Noncollision singularities in the n-body problem. *Department of Mathematics, Princeton University*, 2013.

[10] Z. Xia. The existance of noncollision singularities in newtonian systems. *The Annals of Mathematics, Second Series*, 135:413–467, 1992.

APPENDIX B

Listings

B.1 Runge Kutta 4 Implementation

```
public static Array<? extends Body> rk4(Array<? extends Body> bodies, float dt) {
  Array<Vector2> K_1_V = new Array<Vector2>(bodies.size);
  Array<Vector2> K_1_R = new Array<Vector2>(bodies.size);
  Array<Vector2> K_2_V = new Array<Vector2>(bodies.size);
  Array<Vector2> K_2_R = new Array<Vector2>(bodies.size);
  Array<Vector2> K_3_V = new Array<Vector2>(bodies.size);
  Array<Vector2> K_3_R = new Array<Vector2>(bodies.size);
  Array<SimpleBody> psuedoBodies =  new Array<SimpleBody>(bodies.size);

  BruteForce(bodies);
  for (int i =0; i< bodies.size; i++) {
    Vector2 r = bodies.get(i).getR();
    Vector2 v = bodies.get(i).getV();
    Vector2 a = bodies.get(i).getA();
    Vector2 K_1_v = new Vector2(a).scl(dt/2);
    Vector2 K_1_r = new Vector2(v).scl(dt/2);
    K_1_V.add(K_1_v);
    K_1_R.add(K_1_r);
    SimpleBody c = new SimpleBody(bodies.get(i));
    c.setV(v.add(K_1_v));
    c.setR(r.add(K_1_r));
    psuedoBodies.add(c);
  }
  BruteForce(psuedoBodies);
  for (int i =0; i< bodies.size; i++) {
    Vector2 r = bodies.get(i).getR();
    Vector2 v = bodies.get(i).getV();
    Vector2 a = bodies.get(i).getA();

    Vector2 K_1_r = K_1_R.get(i);
    Vector2 K_1_v = K_1_V.get(i);

    Vector2 K_2_v = new Vector2(a).scl(dt/2);
    Vector2 K_2_r = new Vector2(v).add(K_1_v).scl(dt/2);

    K_2_V.add(K_2_v);
    K_2_R.add(K_2_r);
    SimpleBody c = new SimpleBody(bodies.get(i));
    c.setV(v.add(K_2_v));
    c.setR(r.add(K_2_r));
    psuedoBodies.set(i,c);
  }
  BruteForce(psuedoBodies);
  for (int i =0; i< bodies.size; i++) {
    Vector2 r = bodies.get(i).getR();
    Vector2 v = bodies.get(i).getV();
    Vector2 a = bodies.get(i).getA();
    Vector2 K_2_r = K_2_R.get(i);
    Vector2 K_2_v = K_2_V.get(i);

    Vector2 K_3_v = new Vector2(a).scl(dt);
    Vector2 K_3_r = new Vector2(v).add(K_2_v).scl(dt);

    K_3_V.add(K_3_v);
    K_3_R.add(K_3_r);

    SimpleBody c = new SimpleBody(bodies.get(i));
    c.setV(v.add(K_3_v));
    c.setR(r.add(K_3_r));

    psuedoBodies.set(i,c);
  }
  BruteForce(psuedoBodies);
  for (int i =0; i< bodies.size; i++) {
    Vector2 r = bodies.get(i).getR();
    Vector2 v = bodies.get(i).getV();
    Vector2 a = bodies.get(i).getA();
    Vector2 K_1_v = K_1_V.get(i);
    Vector2 K_1_r = K_1_R.get(i);
    Vector2 K_2_v = K_2_V.get(i);
    Vector2 K_2_r = K_2_R.get(i);
    Vector2 K_3_v = K_3_V.get(i);
    Vector2 K_3_r = K_3_R.get(i);
    Vector2 K_4_v = new Vector2(a).scl(dt/2);
    Vector2 K_4_r = new Vector2(v).add(K_3_v).scl(dt/2);

    Vector2 v_diff = new Vector2(K_1_v)
      .mulAdd(new Vector2(K_2_v),2)
      .mulAdd(new Vector2 (K_3_v),2)
      .add(new Vector2(K_4_v));
    Vector2 r_diff = new Vector2(K_1_r)
      .mulAdd(new Vector2(K_2_r),2)
      .mulAdd(new Vector2 (K_3_r),2)
      .add(new Vector2(K_4_r));

    Vector2 V = v.mulAdd(v_diff,1);
    Vector2 R = r.mulAdd(r_diff,1);
    bodies.get(i).setV(V);
    bodies.get(i).setR(R);
  }
  return bodies;
}
```

APPENDIX C

Test Results

Simulation Tests

Test #	Description	Input	Expected Output	Actual Output	Result
1	Acceleration not computed in zero-mass case.	3 bodies created with no mass and no velocity	No movement	No movement was observed in either body.	PASS
2	Bodies will move towards each other when both have mass.	2 bodies, both with equivalent non-zero mass.	Both bodies move towards the midpoint between them.	The two bodies travelled towards each other gradually gaining in speed along the bisecting line.	PASS
3	Oscillatory Stability	1 body with non zero mass, another body with zero mass and perpendicular velocity.	Predictive curve will draw a perfect circle.	A perfect circle was witnessed with the predictive curve.	PASS
4	Perturbations	Start with stable orbit. Add additional object with non zero mass.	The orbit will be distorted by the new object.	The path of the orbiting body was distorted/perturbed by the new object.	PASS
5	Collision Axis preserved	Two stationary masses created.	The masses will eventually collide and bounce off of one another along the line that intersects them.	Observed 6 successive collisions all along the same line of intersection, returning to the same distance apart.	PASS
6	Multiple Collisions Register in a short timeframe	Loaded example 3 with 7 different objects of varying mass.	At no point should two objects pass through one another.	No object was witnessed passing through another object.	PASS

Object Visualisation Tests

Test #	Description	Input	Expected Output	Actual Output	Result
1	Circle Drawn to the screen.	Launch simulation with objects added to the simulation at (0.5,0.5) (-0.5,-0.5) positions	Two circles appear on the screen, in the top right and bottom left.	Circles were rendered and yellow, in the top right and bottom left of the screen.	PASS
2	Moves when simulation is running.	Begin simulation with an object, with an initial velocity.	Object moves across the screen in the direction of the velocity.	Object was rendered to the screen and could be seen travelling in the direction of the velocity when simulation was active.	PASS
3	Is appropriate color.	Create a body	Yellow circle rendered to screen	Yellow circle seen rendered to screen.	PASS
4	Size varies with radius.	Add three objects of differing radius, at positions (-0.5,-0.5) smallest, (0,0) medium, (0.5,0.5) largest.	Three masses rendered to the screen, smallest in the bottom left, largest in the top right.	Three masses were seen rendered to the screen, smallest in the bottom right, medium in the centre and largest in the top right.	PASS

Object Creation Tests

Test #	Description	Input	Expected Output	Actual Output	Result
1	Object created on centre of mouse pointer.	Launch application, right click in blank space.	Yellow circle produced, centered on mouse.	A new celestial body object was drawn centred on the mouse.	PASS
2	Object continues from same point when simulation is running.	Create object, with velocity. Press play.	Circle does not jump to a different location in the screen upon play event.	Circle did jump to a different section of the screen. It travelled normally.	PASS
3	Velocity arrow, loads value into object.	Create object, set velocity. Press play.	Circle should move in the direction of velocity.	Circle moved in the direction of the body.	PASS
4	On Spacebar event velocity is set to zero.	Create single object, select mass. Press spacebar. Run simulation.	Circle should not move.	Circle did not move.	PASS
5	Mass circle, drawn and moves with mouse.	Create object, move mouse.	Should be a second outline of a circle, where the mouse is always intersecting the edge.	Mouse was always intersecting the edge of the second circle.	PASS
6	Mass circle, loads value into object.	Create two objects, one with mass. Press play	Object with no mass moves towards object with mass.	The object with no mass moved towards the object with mass.	PASS
7	On Spacebar event mass is set to zero.	Create two objects by pressing spacebar, select zero velocity. Then press play.	Spacebar event will switch state to velocity input, on play event neither object will move.	Neither object moved, and spacebar events caused the velocity input to display.	PASS
8	Resets to edit mode after mass is set.	Create object, then right click on blank space.	Final right click should produce a new object.	Final right click produced a new object.	PASS
9	Cannot create object whilst simulation is running.	Press play, right click on screen.	No change to program.	No change to the program was observed.	PASS