NTNU – Trondheim
Norwegian University of
Science and Technology

Procedural generation of multiple stable, small-scale solar systems using 3D N-Body simulation.

Joakim Hommeland

Master of Science in Computer Science

Submission date:	July 2015
Supervisor:	Anne Cathrine Elster, IDI
Co-supervisor:	Reiji Suda, University of Tokyo

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Implement a 3D Galaxy simulator comprised of multiple solar systems using an appropriate established algorithm(either Barnes-Hut, FMM, Particle Mesh, P3M etc. or a suitable combination of methods). The amount of planets per solar system should match what has been currently observed in our universe (up to nine). A central goal of the simulator is to procedurally generate galaxies that remains as stable as possible (that is, keeping the amount of colliding planets or runaway planets as low as possible), and use the implemented N-Body algorithm to test this. Investigate possible solutions to the initial value problem that appear when considering stability by using well-known methodologies (for example Monte Carlo). The Galaxy Simulator should feature 3D graphics and an intuitive user interface to show the results of the simulation, as well as take input from the user. Investigate and determine the hardware to run the Galaxy Simulator on, and justify this choice. For example in a multi-GPU environment, investigate the possibility of dividing workload between multiple GPUs.

Abstract

Owing to mankind's constant pursuit of knowledge, we have been seeking to understand the vast universe around us. Thanks to the field of GPU Computing, we have in the recent decade been able to do simulations on the universe with an efficiency like never before. This allows us to create simulation models of the universe, which we can use to learn more about it. In this thesis, we develop a galaxy generator that generates a small galaxy with a user-specified number of solar systems and internal bodies. One of the central goals of the galaxy generator is to generate galaxies that can remain in equilibrium for as long as possible. We do various tests with the galaxy generator in order to discover and understand what makes a galaxy stable. We find that the integration method and the type of N-Body algorithm has a significant impact on how the simulation is run, and choosing the correct one for the situation is crucial. We develop an approximate, specialized N-Body algorithm and integration method for both CPU and GPU that is able to calculate the movement of the bodies fast by using the center of mass as an approximation. Results show that the proposed N-Body algorithm is able to significantly speed up computation over the All-Pairs GPU N-Body algorithm. We also show that the proposed algorithm is suitable for GPU execution, as the GPU-based algorithm is several times faster than the CPU-based one. When analyzing the accuracy of the approximated algorithm, we found that the error becomes significant over time. We quantify the error and try to explain the reason for the observed error. Finally, we perform several tests on the proposed integration method, and find out that it shows promise as a way to integrate distant forces less frequently than nearby forces.

Sammendrag

På grunn av menneskehetens konstante vilje til å lære nye ting, har vi i lang tid prøvd å forstå universet rundt oss. Takket være GPU Beregning feltet, har vi i de siste ti årene kunnet utføre universesimulasjoner med høyere effektivitet. I denne masteroppgaven, utvikler vi en galakse generator som genererer små, bruker-definerte galakser med et gitt antall solsystem og med et gitt antall interne planeter i solsystemet. Et sentralt mål med generatoren er å generere galakser som er stabile så lenge som mulig. Vi utfører forskjellige typer tester for å utforske hva som gjør en galakse stabil. Vi ser at valg av integrasjonsmetode og N-Body algoritme har stor innvirkning på hvordan simulasjonen blir utført. Dette gjør det viktig å velge riktig metode til riktig tid. Vi utvikler en spesialisert N-Body algoritme og integrasjonsmetode for både CPU og GPU som kan utføre simulasjonen effektivt med å beregne massesenteret til et solsystem. Resultater viser at den foreslåtte N-Body algoritmen gir betydelige ytelsesfordeler i sammenligning med All-Pairs GPU N-Body algoritmen. Vi kom også fram til at den foreslåtte algoritmen er veldig egnet til å bli utført på en GPU, da den er flere ganger raskere enn CPU-versjonen. Når vi analyserer nøyaktigheten til den foreslåtte algoritmen, ser vi at den får store unøyaktigheter etter en tid. Vi kvantifiserer denne unøyaktigheten og prøver å forklare denne. Vi utfører også noen tester på den foreslåtte integrasjonsmetoden, og ser at den virker lovende some en måte å integrere fjerne krefter mindre ofte enn nære krefter.

Acknowledgements

I would like to thank my main advisor Associate Professor Anne C. Elster at the Norwegian University of Science and Technology for her support and advise during this thesis. I would also like to extend my heartfelt gratitude for the help and hospitality shown by Professor Reiji Suda at the University of Tokyo during my stay there. Finally, I would like to thank the Norwegian University of Science and Technology and NVIDIA for their support of the IDI HPC Lab through the GPU Research and Education Center program.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

For centuries, mankind has looked up to the sky in wonder and amazement. The celestial night sky must have been a mysterious and scary sight for the early humans who walked the earth. However, owing to mankind's constant pursuit of knowledge, we nevertheless have been seeking to conquer this scary abyss. While humans have continued to observe the other bodies in the Milky Way galaxy for centuries, it has not been until as recent as a couple of hundred years ago that we finally started to understand some of the intricacies of the universe. Leading mathematicians such as Johannes Kepler and Isaac Newton were instrumental in revolutionising the field around the beginning of the 17th Century. Kepler was the first who was able to explain Solar System Mechanics when he published his laws of planetary motion. Not long after this, Newton was able to verify Kepler's laws when he revolutionised how we understand the universe by publishing his Law's of Motion and his Law of Universal Gravitation.

GPU Computing is a field in Computer Science that has seen a large growth in recent years. In the beginning, GPUs were mainly used to render graphics to a display. However, since the beginning of this century, researchers have found new and exciting ways to use the GPUs for more general purpose computations. Owing to it's high level of parallelism, GPUs can compute certain problems much faster than traditional CPUs.

With the advent of the age of the computer, the field of Computational Science has grown to be a very important pillar of the scientific community. Various types of computations and simulations can now be performed on computers, making it easier than ever to verify previously analytically proven problems. One type of problem that has been widely implemented is the

N-Body algorithm. As the N-Body problem is very suited to run on the GPU, we will be focusing on implementing it to run on the GPU using NVIDIA's CUDA framework. Because of the complexity of the problem, it is not possible to analytically calculate how the bodies in an N-Body system will continue to move in relation to eachother.

However, by utilizing Newton's Law of Universal Gravitation, we are able to simulate the solution with the help of computational hardware. The N-body algorithm has been used many times in order to simulate the movement of bodies in a system. Most approaches focus on high level simulation, without individual planets. In addition to this, previous approaches mainly focus on simulating a universe in a state of chaos, where the goal of the simulation is to see how a state of relative equilibrium can be reached.

In this thesis we will be looking at the problem from a slightly different direction. We will use established theory in the field to generate a simplified galaxy which has already reached this state of relative equilibrium. However, even though a system is in equilibrium at a current time t, it is not possible to guarantee that it will remain so in the future. As such, one of the goals of the thesis is to simulate different types of small galaxy compositions and measure how stable they remain over time.

We will then analyze the results and try to understand why the generated test galaxy is able to remain stable. As we are interested in measuring equilibrium for a group of solar systems, it will also be important to perform simulations that takes into account individual planets. As such, we will also be exploring any issues that appear when working with such a fine-grained galaxy system in terms of simulation. The proposed galaxy generator will generate separate solar systems, each with a user-defined number of planets orbiting a star in a state of equilibrium at time $t = 0$. The solar systems will be distributed around the centre of a common black hole to form a galaxy.

Finally, we will test the generated galaxy, as well as different simulation methods, by utilizing various N-Body algorithms and integration methods running on both the CPU and the GPU. As part of the testing suite, we will consider the underlying data structure and use this to create a modified N-body algorithm and integration method that is tailored to the galaxy generator we have created. The results of the simulation can either be watched in real time through the use of a renderer, or by looking at test data and reports after the simulation has concluded.

1.1 Main Contributions

The following is a list of main contributions.

- We design and develop a galaxy generator that creates small solar systems that orbits a common black hole. Using the galaxy generator as a tool, we can analyze different galaxy compositions to determine which remain stable for the longest period of time.
- We develop a specialized N-Body algorithm that is tailored to the galaxies generated by the generator component. The algorithm approximates the total force exertion of distant solar systems using the center of mass. This allows us to speed up the computation at the cost of accuracy. We analyze and explain the error, and determine that the error grows with the number of bodies in the galaxy.
- Finally, we design a custom integration method that utilizes the specialized N-Body algorithm to integrate bodies in distant solar systems less frequently than bodies inside the same solar system. When testing the method, the initial results seem favorable, allowing us to choose a higher integration stepsize for distant forces.

1.2 Problem Formulation

In this section, we will explain the the problem in more detail and discuss goals and expectations for this thesis. The actual implementation of the application will be detailed in Chapter 3. We have divided the problem into three parts consisting of galaxy generation, N-Body simulation and integration.

1.2.1 Galaxy Generation

It is widely believed that the universe, with it's numerous galaxies, nebulas and other celestial constructs, is headed towards chaos. While it may seem like we are living in a state of relative equilibrium here from Earth, there are constantly millions of collisions going on in our own galaxy. Any one galaxy in the universe is in a constant state of flux, with millions of bodies dynamically moving in relation to each other.

In this thesis, we will study some of inherent attributes found in most galaxies. Since galaxies move in a dynamic way, it is inevitable that bodies collide or gets flung out in space by the sheer force of all the gravitational interactions. We want to test certain galaxy configurations and see how they develop over time in terms of stabilitity. We also want to determine what properties a stable galaxy has, in order to get insight in how these are formed.

With stability we mean that the system of bodies remain in a state that is close to it's current state, without diverging unrecoverably. In terms of celestial mechanics, this essentially means that systems of bodies move about each other with as few collisions and runaway bodies as possible. Using celestial mechanics, we want to explore how to achieve this stability in our simulations. This term is similar to, but not the same as numerical stability. With the term numerical stability, we talk about the desired property that rounding errors and other small numerical errors do not scale up significantly to create major errors. There are many factors that are at play, and correctly determine the cause of something can be difficult. It is important to distinguish between phenomena that happens because of a physical property and phenomena that happens because of a numerical error. As such, in the simulation part of the program, it will be crucial to develop clear and easy to understand tools in order to give the user the necessary information to determine the validity of the simulation.

A galaxy is comprised of billions of objects with varying mass and density. Scattered around in the galaxy we can observe massive constructs such as nebulas and dust clouds. While these galactical constructs are an integral part of what makes a galaxy, we will not focus on these aspects in this thesis. We will instead consider a simplied approach which focuses on the interaction between three important components inherent in most galaxies. According to recent research, it is widely believed that a supermassive black hole is present at the core of the galaxy [32]. This massive black hole is one of the major reasons that the galaxy can remain relatively stable. Orbiting this black hole are a wide range of other galactical objects. In this thesis, we will focus on designing and implementing a Galaxy Generator that procedurally generates a simplified galaxy comprised of a massive black hole in the center, with several solar systems orbiting it. Current observations suggest that a solar system can at most have up to nine planets[36]. During simulation, the solar systems in the generated galaxies should be kept stable such that the bodies in each solar system remain tightly coupled together for as long as possible. We plan to achieve this by using Celestial Mechanics in order to solve the initial velocity and position problem for all the bodies in the galaxy. The primary goals of the Galaxy Generator module can be summarized as follows.

1. Generate a galaxy comprised of a black hole with an abitrary number of solar systems orbiting it. The bodies in the galaxy should remain stable as long as possible.

2. After generation, send the necessary galaxy data over to the simulation module to perform N-Body Simulation.

3. Provide the user with ample feedback about the different properties about the generated system during simulation.

In the simulator, we will focus on observing how each of the solar systems continue to move in relation to each other. The general topology of the galaxy (how the solar systems are distributed around the black hole) will be simplified to facilitate such testing. In our universe, the Milky Way galaxy has a spiral topology, while the galaxies we will be generating will have simplified circular and spherical topologies.

1.2.2 N-Body Simulation

In order to calculate the exertion felt by a body at a time t, we need to employ an N-Body algorithm, which we will explain in more detail in Chapter 2. There are many approaches to the N-Body algorithm, ranging from the brute force All-Pairs approach to more sophisticated approaches that uses some form of heuristic to optimize the calculations. In this thesis we will focus on implementing GPU-based N-Body algorithms. However, for comparisons reasons we will also implement the CPU equivalent approaches.

An important goal of the Galaxy Simulator is to use various N-Body algorithms to test how the system remains stable. As each approach to the N-Body algorithm we will implement will be different, we wish to see if these changes have any major impact on the simulation itself. The results of the simulation should be sent to the renderer in order to render the results to a screen. As such, the information flow between all the component of the program to be implemented must be efficient.

1.2.3 Integration

Choosing the correct integration method and timestep for a given time will be crucial in order for the system to remain stable. If the timestep is too big, the simulation will be less accurate, which could potentially make an otherwise stable system become unstable. It is our goal to investigate this, and see if we can find a suitable integration method and timestep for a given situation. While a smaller timestep would always be more accurate, we need to find the correct trade-off between performance and accuracy.

1.3 Outline

The rest of this thesis is outlined as follows

Chapter 2	This chapter will describe the required background to understand the contents of this thesis and introduce related works.
Chapter 3	In this chapter, we will present the implemented application and explain the design decision that went into making it.
Chapter 4	This chapter will detail the results obtained when running the generator and simulator for various scenarios.
Chapter 5	In this chapter we will conclude on the work done and present ideas for future work.
Appendix A	This appendix details the installation and user guide in order to operate the implemented application.
Appendix B	This appendix presents relevant galactical constants and simulation variables used in the application.
Appendix C	This appendix contains the main code contributions from this thesis.

Chapter 2

Background and Related Works

In this chapter, we will briefly describe the theory that we consider prerequisite knowledge in order to properly understand the contents of this thesis. We will start off by covering Celestial Mechanics, a field made popular by Kepler's astronomic discoveries in the 17th century. We will then take a look at his successor, Sir Isaac Newton, and how he further improved the theory established by Kepler. By considering the contributions from both Kepler and Newton, we are then able to analytically solve the Two Body Problem. After that, we will talk about the N-body algorithm and how it can be used to calculate the movement of the bodies in a system and how this is tied together with Numerical Integration approaches. As GPU Computation is an important part of this thesis, we will also spent some time going through the core concepts of this field. The final section will briefly talk about how to render the output of the simulation to the screen and how to efficiently share information between the renderer and the simulation component. At the end of the chapter we will present other related works.

2.1 Celestial Mechanics

Celestial Mechanics is a field in astronomy that describes the motion of celestial objects, such as planets and stars. Using theory from this field, we can predict the motion of these bodies over time. When this field started to emerge in the 17th century, quantitative data obtained by observing the celestial bodies from Earth were analysed in order to create theory that matched the observations. A key person in advancing the field at that time, was the

German mathematician Johannes Kepler. Using data obtained by observing Mars from Earth, he was able to determine that the orbits of the planets were slightly elliptic and not a perfect circle as was previously assumed.

Johannes Kepler used this discovery and other empiric data available to him at the time to formulate his Three Laws of Planetary Motion, which he explains in his two famous publications, Astronomia Nova[17] and Harmonices Mundi[18].

1. Kepler's First Law: The planets move in ellipsis with the Sun at one of the two foci.

2. Kepler's Second Law: The line connecting the planet to the Sun sweeps out equal areas in equal times as it travels around the ellipse.

3. Kepler's Third Law: The square of the orbital period of a planet is proportional to the cube of its semi-major axis.

In the following sections, we will briefly talk about these laws and how they can be used to create our own solar systems where each planet is in orbit of a common star.

2.1.1 Kepler's First Law

Kepler's First Law states that "a planet that orbits the sun, will orbit it in an elliptic path where the Sun is located in one of the foci of the ellipse"[17]. An ellipse can be described as a flattened circle which has two foci, as opposed to a circle which has only one. While the Sun is located in one of the foci, the other is generally empty. At any point along the ellipsis, the sum of the distances from the point to each of the two foci is constant. This can be used to define an ellipse in terms of two distances d_1 and d_2 from the two foci. If we know the location of the two foci in addition to the two distances, we can draw an ellipse.

The eccentricity of an ellipse is a value that describes how flattened the ellipse is compared to a regular circle. When the eccentricity value is zero, we have the special case of a circle. For any other values approaching one, the ellipse will become more and more flattened, as the two foci moves farther and father away from each other. The ellipse is only defined for eccentricity values between zero up til one. For an eccentricity value of exactly one, parabolas are expressed. For values higher than one, the eccentricity describes the

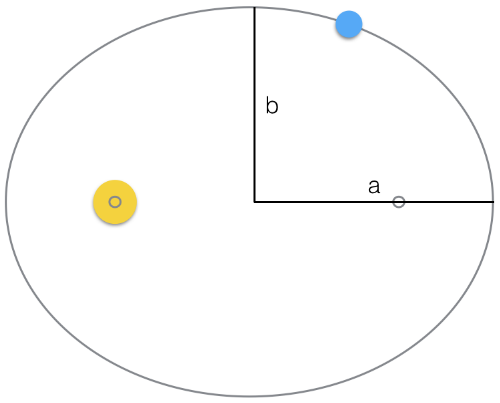


Figure 2.1: A planet orbiting the Sun in an elliptic orbit. a denotes the semi-major axis and b denotes the semi-minor axis of the ellipse.

shape of a hyperbola. While the planets orbiting the Sun all exhibit elliptic traits, they are not as pronounced as the ones shown in the figures above. In fact, most of the planets have orbits that have near circle shapes, with an eccentricity value close to zero. As such, the orbits are only slightly elliptic in nature. In the inner solar system, Mercury has the highest eccentricity value of 0,205. In Table 2.1, the eccentricity values of the four planets closest to the sun is shown. This information has been compiled from the NASA Planetary Factsheet[38]. The values for the remaining planets can be found in Appendix B1.

Planet	Eccentricity (ε)
Mercury	0,205
Venus	0,007
Earth	0,017
Mars	0,094

Table 2.1: Eccentricity values for planets of the inner solar system.

Since the ellipse is flattened, the length of the axes are not equal as it is with a circle. The long axis of an ellipse is called the major axis, while the short axis is called the minor axis. Equivalently, half of the major axis is called the semi-major axis (denoted by a) and half of the minor axis is called the semi-minor axis (denoted by b). In Section 2.3 we will be deriving the mathematical expression for finding the position of an orbiting body along an elliptic path, given an angle φ.

2.1.2 Kepler's Second Law

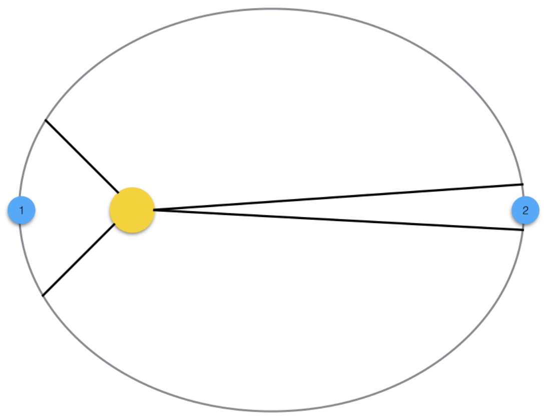


Figure 2.2: An illustration of Kepler's Second Law. Point 1 is located at the perihelion, while point 2 is located at the aphelion.

Kepler's Second Law states that "the line connecting the planet to the Sun will sweep out equal areas in an equal amount of time"[17]. The length of this line represents the distance from the Sun at a given point on the ellipse. The point along the orbit that is closest to the Sun is commonly referred to as the perihelion, while the point farthest from the Sun is known as the aphelion. In Figure 2.2, point 1 is at the perihelion and point 2 is located at the aphelion. As a consequence of Kepler's Second Law, we know that the velocity of a planet in orbit is faster the closer it is to the Sun. Consequently, the orbiting planet's velocity is the highest at the perihelion and the slowest at the aphelion.

2.1.3 Kepler's Third Law

Kepler's Third Law states that "the square of the orbital period of a planet is proportional to the cube of its semi-major axis"[18]. It can also be expressed mathematically with the following formula.

$$P^2/a^3 \tag{2.1}$$

Where P is the orbital period and a is the semi-major axis. Using Kepler's Third Law, we can see that planets that are farther away from the Sun will have a longer orbital period than planets that are closer.

2.2 Newton's Universal Law of Gravitation

Following the discoveries made my Kepler before him, Sir Isaac Newton went on to prove that a square force law was the origin of all motion in the solar system in his famous work Principia[29]. Although Kepler had previously been able to stipulate his famous laws based on observed data, he was unable to explain why the planets moved as they did. Using the results obtained by Kepler, Newton was able to come up with his Universal Law of Gravitation that explained why the planets move liked Kepler had observed. In Principia, Newton also famously stipulated his laws of motion. These laws detail how all bodies in the universe move in relation to each other.

1. Newton's First Law: Bodies will remain in a state of rest or constant motion in a straight line unless acted upon by a force.

2. Newton's Second Law: The force experienced by a body is equal to the rate of change of its momentum. ($F = ma$)

3. Newton's Third Law: To every action there is an equal and opposite reaction.

The core principle of The Universal Law of Gravitation states that "two bodies in the universe attract each other with a force proportional to the product of their masses, and inversely proportional to the square of the distance between them"[29]. What makes Newton's discoveries so revolutionary is that he was able to prove that this law is universally true for all objects in the universe. By combining the three laws of motion and the Law of Universal Gravitation, Newton was able to explain why Kepler's Laws of Planetary Motion behaved as they did. Newton's Law of Universal Gravitation can be expressed mathematically as

$$F = G\frac{m_1 m_2}{r^2} \tag{2.2}$$

where F denoted the force between the masses m_1 and m_2 and r is the distance between them. G denotes the Gravitational Constant, which is a proportionality constant needed to solve the problem for a specific set of units. The value of G was experimentally determined in 1798 by Lord Henry Canvendish to be $G = 6.673E^{-11}N\ m^2/kg^2$[5], assuming that distance is expressed in meters and masses are expressed in kilograms. It is also possible to modify this constant in order to express the force using different units.

Newton's Laws of motion significantly changed our understanding of the universe. Using these laws, we can predict the interactive movements of many objects in the universe with a high degree of accuracy. Although unknown at the time, Newtons discoveries were not the final solution to the problem. As it turns out, Einstein's theory of relativity[8] is able to even more accurately predict how objects move in relation to each other. However, as Newton's laws are a reasonable approximation for most cases, we will not take general relativity into account in this thesis.

2.3 The Two Body Problem

The Two Body Problem is the problem of determining how two celestial bodies interact and move with each other. The two bodies will exert a force on on the other body, causing them to move about each other. Although Kepler did not understand why the bodies moved like they did, his discoveries made a significant contribution to how we can analytically solve the movement of two bodies in relation to each other. The Two Body problem can be solved analytically, but because of the complexity of the problem, it is not possible to analytically determine how a general N-Body system will move in relation to each other. The solution to the Two Body Problem we will focus on is based on both Kepler's and Newton's Laws. The following formulas are based on their succinct explanation in the book "Solar System Dynamics" by C.D Murray and S.F Dermott[28].

We begin by deriving the equations of motion, which follows from Newton's Laws of Universal Gravitation. We consider a two body system with a smaller body with mass m_2 orbiting a larger body with mass m_1. The location of each body from the origin is determined by the respective vectors $\vec{r_1}$ and $\vec{r_2}$. In the mathematical formulas in this thesis, we will be using the vector notation $\vec{v}$ to distinguish between vectors and scalars. A third vector $\vec{r_{rel}} = \vec{r_2} - \vec{r_1}$ denotes the position of m_2 in relation to m_1. It's direction goes from the primary body, the Sun, to the orbiting planet.

Using Newton's formulas, we can find the gravitational force exerted on one body by the other.

$$\vec{F_1} = G\frac{m_1 m_2}{r^3}\vec{r} = m_1\ddot{\vec{r_1}} \tag{2.3}$$

$$\vec{F_2} = -G\frac{m_1 m_2}{r^3}\vec{r} = m_2\ddot{\vec{r_2}} \tag{2.4}$$

The above formulas depict the forces that each of the bodies exert on each other. Since we will be working with vectors, we also need to add the direction of force by multiplying a unit vector $\frac{\vec{r}}{\|\vec{r}\|}$ to Formula 2.2. The forces are in the opposite direction of each other. Using Newton's Second law $F = ma$, we can rewrite the forces as shown above. Variables with one or multiple dots over them depict the the time derivative. In this case, the variables represents the double time derivative of the position, which gives us the acceleration. Essentially, by determining the acceleration of the body, we can integrate this to the get motion over time.

We assume that $m_1 >> m_2$ such that the force exerted on the Sun by the planet is so small that the Sun will remain stationary. We can then simplify the problem since we will only need to find the movement of m_2. Since we know the position of m_1 at any one time, we can use the the double derivate of the relative position in order to find the relative acceleration $\ddot{\vec{r_{rel}}} = \ddot{\vec{r_2}} - \ddot{\vec{r_1}}$. Combining this formula with (2.3) and (2.4), we get

$$\frac{d^2\vec{r}}{dt^2} + \mu\frac{\vec{r}}{r^3} = 0 \tag{2.5}$$

where $\mu = G(m_1 + m_2)$. This equation gives us the relative motion between the two bodies. Since we will be working with the movement of one body in relation to another, the movement of the orbiting body will be restricted to a 2D orbiting plane. We convert to a 2D plane, where the Sun is located in the origin of the plane. Instead of using a cartesian coordinate system, we will instead be using a polar coordinate system to describe the motion of the orbiting body. By defining $\hat{\vec{r}}$ as the unit vector for the direction of $\vec{r}$ and $\hat{\vec{\varphi}}$ as the unit vector for the direction of $\dot{\vec{r}}$ (also known as the angular velocity), we can rewrite the position, velocity and acceleration in polar coordinates.

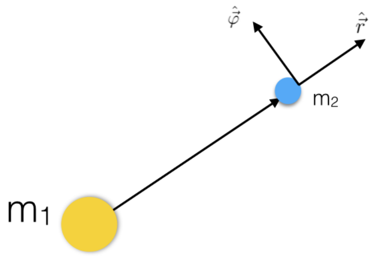


Figure 2.3: A picture showing the direction of the vectors in relation to each other

$$\vec{r} = r\hat{r}, \qquad \dot{\vec{r}} = \dot{r}\hat{\vec{r}} + r\dot{\varphi}\hat{\vec{\varphi}}, \qquad \ddot{\vec{r}} = (\ddot{r} - r\dot{\varphi^2})\hat{\vec{r}} + [\frac{1}{r}\frac{d}{dt}(r^2\dot{\varphi})]\dot{\vec{\varphi}} \tag{2.6}$$

We can now rewrite the relative motion equation (2.5) as a scalar equation by combining it with formula (2.6).

$$\ddot{r} - r\dot{\varphi^2} = -\frac{\mu}{r^2} \tag{2.7}$$

This gives us a non-linear differential equation, which is hard to solve. In order to be able to solve this equation, we need to express r as function of φ and not as a function of time. We do this by making the substitution $u = \frac{1}{r}$ and we eliminate time by introducing the constant $h = r^2\dot{\varphi}$.

By substituting, we can rewrite the differential equation to

$$u'' + u = \frac{\mu}{h^2} \tag{2.8}$$

We now have a second order linear differential equation which we can solve:

$$u = \frac{\mu}{h^2}[1 + e\ cos(\varphi - \varpi)] \tag{2.9}$$

We then substitute back for r and rearrange to get

$$r = \frac{p}{1 + e\ cos(\varphi - \varpi)} \tag{2.10}$$

The constant p is called the semilatus rectum, defined by the constant $p = \frac{h^2}{\varphi}$. For different values of the eccentricity e, the semilatus rectum can be rewritten to define different conic sections. In this thesis, we are interested in elliptical motion, which have an eccentricity value $0 < e < 1$. The corresponding semilatus rectum value is $p = a(1-e^2)$. By inserting this into the function above, we get the motion of the body on an elliptic path, where the ellipse is defined by the eccentricity e and the semi-major axis a. The variable ϖ refers to the longitude of the pericentre, the point which is closest to the Sun. We set $\varpi = 0$ in order to have the orbiting body located at the pericentre when $\varphi = 0$. We can thus write the equation as:

$$r = \frac{a(1-e^2)}{1 + e\ cos(\varphi)} \tag{2.11}$$

We can now express the movement of the body given an angle φ. Using the two variables (r,φ), we are able to express the distance from the Sun and the angle of the current position in the ellipse for the orbiting body. In this thesis, we will be using this formula in order to place bodies in an orbit at an arbitrary point along the elliptic path. Since we will be using a cartesian coordinate system, we first need to convert it back from the polar coordinate system. If we set the x axis to be direction from the Sun towards the pericentre, we can express it in the cartesian coordinate system with:

$$x = r\ cos\ \varphi \qquad and \qquad y = r\ sin\ \varphi \tag{2.12}$$

The next step is to find the velocities for a given angle φ such that we can assign the correct start values for any orbiting body with a start position along the elliptic path. In order to do this, we need to find an expression for the orbital period T. Using Kepler's Third Law, we can define the equation for the orbital period T as:

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{2.13}$$

We know that the angle φ will span from 0 to 2π as the body travels around the ellipse. We can use this to define the mean motion (the average velocity around the ellipse) as:

$$n = \frac{2\pi}{T} \tag{2.14}$$

We can use the equation for mean motion to derive the equations for the velocity a body has around the ellipse, given an angle φ. We combine equation (2.13) and (2.14) and solve for μ to get

$$\mu = n^2 a^3 \tag{2.15}$$

We can then use this equation to to rewrite h, defined by the semilatus rectum p on the previous page.

$$h = na^2\sqrt{1-e^2} \tag{2.16}$$

Next, we differentiate equation (2.11) to get an expression for the velocity of the body:

$$\dot{r} = \frac{r\ \dot{\varphi}\ e\ sin\ \varphi}{1 + e\ cos\ \varphi} \tag{2.17}$$

This equation can be rewritten by inserting the value for h found in equation (2.16) which also equals $h = r^2\varphi$. This gives us the following two equations:

$$\dot{r} = \frac{na}{\sqrt{1-e^2}} e\ sin\ \varphi \tag{2.18}$$

$$r\ \dot{\varphi} = \frac{na}{\sqrt{1-e^2}}(1 + e\ cos\ \varphi) \tag{2.19}$$

Finally, we find the x and y components of the velocity by differentiating equation (2.12) and combining them with equations (2.18) and (2.19).

$$\dot{x} = -\frac{na}{\sqrt{1-e^2}} sin\ \varphi \tag{2.20}$$

$$\dot{y} = \frac{na}{\sqrt{1-e^2}}(e\ + cos\ \varphi) \tag{2.21}$$

We can now create an arbitrary elliptic orbit and have an object orbit a Sun at an arbitrary point along the orbit. This solves the initial value problem for planetary orbits. However, we still need to find a way to express r as a function of time, and not as a function of φ. We can do this by introducing the mean anomaly M:

$$M = n(t - \tau) \tag{2.22}$$

where τ is the time when the pericentre was last passed(the angle φ is equal to 0). We can use this variable to find the position of a body given the current time. Unfortunately, since M assumes that the velocity around the orbit is constant, it is only valid when the eccentricity is equal to zero. In order to find the solution for the general case, we need to define a new variable, the eccentric anomaly E. By using the above formula for M, we are able to relate a value of M between 0 and 2π for a given time t. In order for us to get the angle at a given time for any eccentricity between 0 and 1, we need to use Kepler's Equation[35].

$$M = E - e\ sin\ E \tag{2.23}$$

As Kepler's Equation is transcendental, we cannot solve for E algebraically. Instead, we need to use numerical analysis to approximate the value of E. There are various ways to do the approximation as detailed in Danby (1988)[6]. In this thesis we will be using the Newton-Raphson method to approximate the eccentric anomaly E. The equation is:

$$E_{i+1} = E_i - \frac{f(E_i)}{f'(E_i)}, \qquad i = 0, 1, 2, ..., \tag{2.24}$$

Where $f(E_i) = E_i - e\ sin\ E_i - M$ and $f'(E_i) = 1 - e\ cos\ E_i$.

In order for the approximation to be as accurate as possible, we need to have a suitable starting value. In his book, Danby suggests the value of E_0 to be:

$$E_0 = M + sign(sin\ M)ke, \qquad 0 \leq k \leq 1 \tag{2.25}$$

Where the value of k is recommended to be $k = 0.85$. We can now find the approximated value of E with respects to time, and use this as the angle of the body in the elliptic orbit.

The last thing we need to do is find the formulas for the x and y coordinates that the orbiting body has given the eccentric anomaly E, for a given orbit around the Sun. In order to do so, we introduce the mathematical formula for the ellipse in the cartesian coordinate system.

$$\left(\frac{\bar{x}}{a}\right)^2 + \left(\frac{\bar{y}}{b}\right)^2 = 1 \tag{2.26}$$

We can express the x dimension of the ellipse in terms of E with the equation $\bar{x} = a\ cos\ E$. Inserting this into equation (2.26), we get $\bar{y}^2 = b^2 sin^2\ E$. We can then rewrite this equation in terms of the semi-major axis a, by using the relation $b^2 = a^2(1 - e^2)$, which gives us $\bar{y} = a\sqrt{1 - e^2}\ sin\ E$. We can thus find the equations for the coordinates

$$x = a(cos\ E - e) \qquad and \qquad y = a\sqrt{1 - e^2}\ sin\ E \tag{2.27}$$

Given an elliptic anomaly E, we can express the position of the orbiting body using the above formulas. We have thus solved the Two Body Problem for the case of a smaller body orbiting a larger one.

The formulas can be generalized to 3D using various methods. In the implementation chapter, we do this by rotating the XY-plane given by the above formula to the appropriate angle, and then place the rotated bodies in space given some offset from the origin.

2.4 N-Body Simulation

The N-Body problem is a classic problem in computational physics[1]. In a system of N bodies, the objective is to simulate how the celestial bodies continue to move in relation to each other. The simulation is performed by calculating the gravitational pull between all the bodies, using Newton's Law of Universal Gravitation. In a system with many bodies, this can be computationally expensive, where the brute-force method would need to perform N^2 computations. However, we can take advantage of the underlying structure of the problem to gain significant speedups by hardware acceleration. At any given time t, each body-to-body (or group of bodies) computation can be done independent of all the other bodies in the system. In addition, the ratio between computation and memory access for the N-Body problem is very high. This makes such a problem ideal for execution on accelerated hardware such as a Graphics Processing Unit.

In order to calculate the force exerted on a body i by all the other bodies, we need to apply the formula for Universal Gravitation *N-1* times, and sum

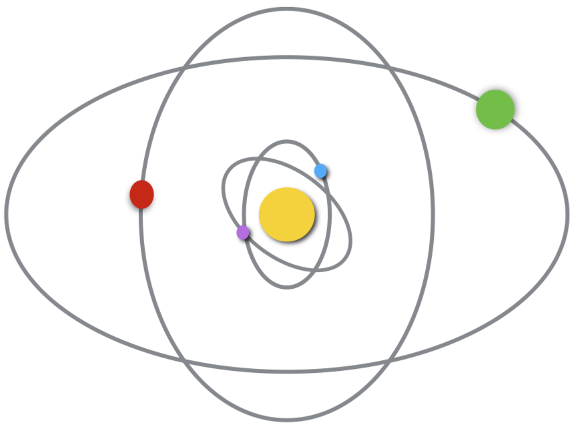


Figure 2.4: An illustrative picture of an N-Body simulation

the results

$$F_i = Gm_i \sum_{\substack{1 \leq i \leq N \\ i \neq j}} \frac{m_j}{r^2} \tag{2.28}$$

This approach is known as the All-pairs N-body algorithm, as each individual body-body pair is computed in order to get the result. While this is the most accurate of the N-body algorithm, it is also quite computational heavy and it does not scale well with a high amount of bodies, especially if done serially on the CPU.

After calculating the forces, the next step is to use Newton's Second Law to find the equivalent acceleration for a body i at time t. By using Newton's Second Law, we can derive the formula for the acceleration

$$F_i = m_i a_i \tag{2.29}$$

$$a_i = \frac{F_i}{m_i} \tag{2.30}$$

Since we will be doing N-Body computations in 3D space, we will need to do 3D vector computations. We can solve the N-body problem for 3D space by calculating the acceleration in each dimension of the vector independently. Each vector is three dimensional with axes x, y and z. Since we will do simulations in 3D, we need to include the direction of the vector. As such we multiply the magnitude of the force with a unit vector of $\vec{r_{ij}}$, which will give us the acceleration vector.

$$\vec{a_i} = G \sum_{\substack{1 \leq i \leq N \\ i \neq j}} \frac{m_j}{\left\|\vec{r_{ij}}^2\right\|} \frac{\vec{r_{ij}}}{\left\|\vec{r_{ij}}\right\|} \tag{2.31}$$

There exists several variations of the N-body algorithm, each aiming to improve the performance over the standard All-pairs approach. The Barnes-Hut[3] algorithm is an approximation that divides the bodies into an octree in three dimensional space. Using the octree, the algorithm is able to treat near bodies invididually while grouping up bodies that are far away in order to calculate an approximation of their combined force extertion. This well-known approach is popular in the litterature for being a good trade-off between performance and accuracy. While the Barnes-Hut approach is excellent for execution on the CPU, it is not well-suited for an GPU implementation. There have been successful attempts at tuning the algorithm for efficient execution on the GPU[4], however it requires a significant amount of effort to get it to run well.

The All-Pairs GPU N-Body (GEM3) algorithm developed by NVIDIA and presented in GPU Gems 3[31] is an example of an N-Body algorithm that has been efficiently and successfully implemented on the GPU. While the algorithm is $O(N^2)$, NVIDIA uses the power of the GPU in order to calculate the new acceleration of a body in an efficient parallel fashion. Because the acceleration exerted on a body by the other bodies in the system can be calculated in parallel, there is a lot of potential that the GPU can take advantage of. The GEM3 algorithm defines an NxN grid which represents the pairwise computation for the N bodies in the system to simulate. This grid is then divided into sub-tiles where each of the pairwise computations are calculated in parallel. This allows the GPU to take full advantage of the problem structure, maximizing efficiency.

2.5 Numerical Integration

By using the N-body algorithm, we are able to calculate the force exerted on a body by all other bodies in a system. We can then easily convert this to acceleration, which we can use to determine how the planets move over time.

After we have found the acceleration, we need to find the velocity change and positional change caused by the acceleration at a time *t*. This can be done using a numerical integration method[2] with a given timestep *dt*. There exists several numerical integration methods, each with different strengths and weaknesses. There are several examples in the litterature that suggests that choosing the correct one for the correct situation is crucial in order to get both a fast and accuracte answer. Therefore, it is essential also in our case to choose an integration method that suits the physical simulations we will be doing in this thesis. We will now take a look at two different methods and discuss them in order to find the most suitable one.

2.5.1 Explicit Integration

Using an explicit integration method, we are able to find the next state of a system at time *t+1* using only the state information we have about the system for the current time *t*. This is a common numerical integration method used in many applications. It's known for it's simplicity since we only use information we currently have access to, and do not need to do intermediary computations or approximations in order to complete the integration step. In our case, we already know the acceleration at a given time t by running the N-body algorithm. Using the current acceleration $\vec{a_t}$ and the current velocity $\vec{v_t}$, we can calculate the value for the velocity at time *t+1*

$$\vec{v_{t+1}} = \vec{v_t} + \vec{a_t}dt \tag{2.32}$$

We then use the current position $\vec{x_t}$ and the current velocity $\vec{v_t}$ to update the position of each of the bodies for time *t+1*

$$\vec{x_{t+1}} = \vec{x_t} + \vec{v_t}dt \tag{2.33}$$

The above equations are also known as the explicit Euler stepping equations. By inspection, we can see that they uphold the laws of motion as stipulated by Newton. Unfortunately, by using this method, the energy in the system will continue to increase steadily. Therefore, it does not uphold the laws of conservation of energy. As the simulations continue, the energy will continue to build and increase the errors as the simulation goes on. It is possible to mitigate this by lowering the timestep, however this will increase the computational cost significantly.

2.5.2 Semi-implicit Integration

The symplectic Euler integration method is likely to be more suitable in our case. It is commonly used in many types physics computations and simulation because it can better conserve the energy in the system. Using the symplectic Euler method, the error caused by energy increasing in the system will average out to a constant, instead of continuously increasing. Therefore, the symplectic Euler method is able to conserve energy on average. The symplectic Euler method uses a semi-implicit integration scheme. It uses a combination of the implicit and explicit method in order to calculate the new values for time *t+1*. We start off by calculating the new velocity as we did before

$$\vec{v_{t+1}} = \vec{v_t} + \vec{a_t}dt$$

However, unlike before, we use the new velocity at time *t+1* and the current position to update the bodies' position.

$$\vec{x_{t+1}} = \vec{x_t} + \vec{v_{t+1}}dt \tag{2.34}$$

The symplectic Euler method is a second order method. As such, by cutting the timestep in half, the accuracy of the method will go up by four times. When running the computations, it is imperative to find a timestep that suitably balances accuracy and computation time.

There are pros and cons for just about any numerical integration method in existence. When specifying the goals of the project, is it important to think about several factors when deciding the appropriate integrations scheme to choose. There does not exist a perfect integration method that guarantees the best conservation of energy, accuracy and fastest computational time for all situations. It is therefore an important question to consider when designing simulation software.

While we cannot find a perfect integration method for all occasions, there are alternative approaches that utilises multiple integration methods. In many simulations there is a considerable potential of improvement if we allow ourselves to dynamically switch both the integration method itself, but also the step size in the integration.

In this thesis, we will mainly focus on the Symplectic Euler integration method. For use with our generated galaxies, this method is relatively simple to implement both on the CPU and the GPU. In addition, as opposed to simpler integration methods, the Symplectic Euler method is able to conserve energy. When doing simulations with individual solar systems, this will be essential in order to get sufficient accuracy.

2.6 GPU Computing

Ever since the dawn of the age of computing, the Central Processing Unit(CPU for short) has been regarded as the de facto general purpose processing element of a computer. Although CPUs started out as single-core serial computing elements, the demand for energy efficiency and energy dissipations problems has led the CPU into becoming multi-core. Instead of having large and complex single-core CPUs which we would struggle to power on fully, smaller and simpler multi-core architectures have become popular. With the advent of multi-core architectures, it is the task of the programmers to learn how to develop and utilise this new architecture properly in applications.

Historically, the Graphics Processing Unit(GPU for short) has been used to render graphical output to a display. The GPUs were implemented with specialised rendering hardware in order to render pixels very fast. Taking inspiration from multi-core CPUs, researchers started to ask themselves if it were possible to use GPUs for more general purpose applications. This lead to the field of General Purpose GPU (GPGPU) computations.

At the beginning of the millennium the support for this was limited. The major manufacturers had not yet caught on to the recent trends and offered easy programmability for GPUs. Researchers were forced to use graphics libraries like OpenGL and Direct3D to trick the GPU into thinking it was working with graphics, when it was actually working with general purpose computations. The lack of support made programming for the GPU tedious, limitings its usefulness.

Around 2006, hardware manufacturers such as NVIDIA began to redesign their systems in order to support the recent GPGPU field. The previous architectures were redesigned for easier general purpose programmability. Programming languages like CUDA C were created to make it easier for programmers to utilise the parallel powers of the GPU. These languages were designed to coincide with the previous graphics libraries, making the GPU much more programmable for developers. From this, the field of GPU Computing was born.

2.6.1 The Graphics Processing Unit

The architecture of the GPU changed drastically when the push towards general purpose computing was made[19]. GPUs were originally made with a fixed graphics pipeline, designed to quickly rasterise computer graphics. By issuing commands to the GPU through a graphics API, data from the CPU were run through the fixed GPU graphics pipeline before being displayed on screen. However, with the advent of the field of GPU computing, recent GPU micro architectures has taken a different approach. Instead of having a fixed pipeline with limited programmability, GPUs have evolved into something aching to CPUs in terms of programmability. I will focus my efforts in explaining this new trend of GPU micro architecture.

At the heart of the modern GPU architecture lies the stream processor[33]. A stream processor processes data sets, also called streams, in a limited parallel fashion. By employing multiple processing elements, or cores, a stream processor is able to achieve parallelism. Under execution, several cores receive the same instruction from the scheduler. Logic specific to each core makes it possible to uniquely identify them in code. Parallelisation is achieved by assigning different data elements to each core. This approach to computation is known as SIMD (Single Instruction, Multiple Data).

Both NVIDIA and AMD has their own stream processor implementation. AMDs stream processor is named the ”SIMD engine”[9], after the parallel paradigm it follows. NVIDIA named their implementation the ”Streaming Multiprocessor”. While both share similarities, there are quite some differences in the implementations. In this thesis, we will be programming using NVIDIA’s CUDA programming language, so I will focus on explaining the microarchitecture of the Streaming Multiprocessor.

2.6.2 The Streaming Multiprocessor

The Streaming Multiprocessor contains and manages all of the GPUs processing elements. Each SM is able to schedule and execute instructions on its cores independently of the other SMs. Compared to a CPU, there are some significant differences. Where CPU usually have a small amount of physical processing elements, a single SM in the GPU can have tenfolds more. A common number for CPUs in recent years have been 8 cores, which pales in comparison with the current generation Maxwell SMs, which has 128 cores. Figure 2.5 shows a simplified diagram of the components commonly found in a streaming multiprocessor. The figure is a simplication of the GM204 SMM Diagram found in the GTX 980 Whitepaper[22], published by NVIDIA. Each core is labeled as a Stream Processor (SP). While the specifications found in the figure does not match any SM launched by NVIDIA, it serves as a tidy overview of all the important components of a Streaming Multiprocessor.

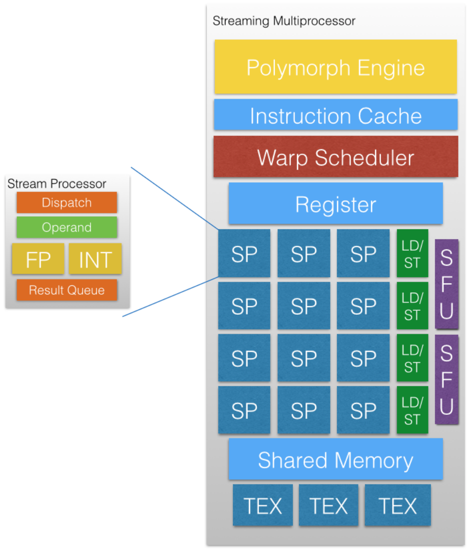


Figure 2.5: Example Streaming Multiprocessor Diagram

The SMs Stream Processors are lightweight processing elements. When the scheduler assign an instruction to the core, it is placed in the Dispatch Queue. When the SP is ready to process the request, the Operand Collector separates the operands from the instruction/opcode, and assigns it to an arithmetic unit. There are two kinds of arithmetic units in NVIDIAs stream processor, Floating Point Units and Integer Units. Based on the type of the operands, the corresponding arithmetic unit is chosen for the computation. When the arithmetic unit is finished, it places the result in the Result Queue where it can then be stored back to its original location and be used in further computations.

In addition to Stream Processors, the SM is also outfitted with several other components to help the stream processors. Load/Store(LD/ST) Units are used to help fetch and store data from the GPUs Global Memory. The Special Function Unit (SFU) are used to compute more complex mathematical functions such as reciprocal, square root and transcendental functions such as sine and cosine.

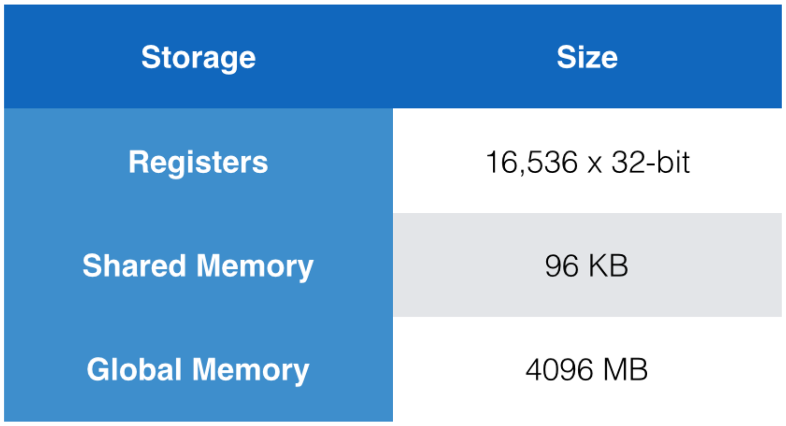
While the total amount of cores far exceed what is possible on the CPU, certain consideration need to be in place when working on GPU architecture. With the abundance of cores, creating an architecture that would allow each of them to work in a total independent fashion, as is common with CPUs, would be too complex and too expensive. Thus, the cores in an SM shares control flow logic between themselves. This essentially means that cores will have to share instructions in order for the GPU to be efficient. Following the SIMD paradigm closely, we can consider the streaming multiprocessor as a SIMT (Single Instruction, Multiple Threads) device.

On NVIDIA devices, SIMT execution is achieved by using a component called a Warp Scheduler to assign instructions to several Stream Processors at once. A Warp is a collection of 32 threads belonging to the same thread block that will all execute the same instruction. The Warp Scheduler fetches instructions from the Instruction Cache and can assign instructions to all the threads in the Warp at the same time. When working with SIMT execution, the programmer needs to be extra cautious when running conditional code. Since every thread in the warp executes the same instruction, threads with data that does not satisfy the conditional will simply halt for the duration of the code within the conditional. Using conditionals in warps causes Warp Divergency, which limits the parallelism of the application and will in the worst case lead to serial computation. Designing algorithms that takes this into account and avoids unnecessary conditionals is important in order to run code efficiently on the GPU.

The Streaming Multiprocessor is outfitted with several memory components to help with execution. Each core has a set of private registers, collectively situated in the Register File. The local registers is the fastest memory component available to the cores, and offers temporary storage for the cores local variables. The next level in the SMs memory hierarchy is the Shared Memory. Although it is slower than the private registers, Shared Memory is still considered very fast temporary storage. Data stored in the Shared Memory is available to all cores in the Streaming Multiprocessor. In order to create efficient applications, Shared Memory should be used whenever it is beneficial. Using Shared Memory, data from Global Memory need only be loaded once between all the cores in the SM, saving a significant amount of time. Typically, applications that does a lot of computations on a single data element will see speedups when using Shared Memory.

The slowest memory option available to the SM is Global Memory. This memory is located outside of the SM itself, and thus the slowest. Global Memory is shared among all the SMs on the GPU, which makes it possible to do inter-SM calculations. In order to speedup consecutive accesses to Global Memory, each SM has an L1 Cache it can use if necessary. Up until the Kepler micro architecture, the L1 Cache was combined in hardware with the Shared Memory Cache. Programmers are able to choose one of several storage space distribution configurations between Shared Memory and the L1 Cache, based on application needs.

Texture Memory is a special cache that can be used to load (texture) data faster from memory. Initially intended to store computer graphics textures, it is a read-only cache that can be used to store data that is known to be constant. Physically, the data is stored in Global Memory, however, special hardware is used to make transfers to and from the Texture Buffers faster. Table 2.2 summarises the memory options available to each core in a Streaming Multiprocessor on the Maxwell GM204 SMM[22].



Storage	Size
Registers	16,536 x 32-bit
Shared Memory	96 KB
Global Memory	4096 MB

Table 2.2: The Streaming Multiprocessor Memory types (Maxwell GM204)

Besides having hardware components to allow for general purpose calculations, the Streaming Multiprocessor also contains specialised hardware to allow the GPU to perform its rendering duties. NVIDIA introduced the PolyMorph Engine in their Fermi architecture. Before that, the rendering components were located outside of the SM, but with Fermi, NVIDIA went with a different approach. Each SM has their own PolyMorph Engine, allowing for fast distributed parallel graphics computations. The PolyMorph Engine contains five hardware components; Vertex Fetch, Tesselator, Viewport Transform, Attribute Setup and Stream Output. These components together perform the legacy rendering duties of the GPU.

2.6.3 High-Level GPU Architecture

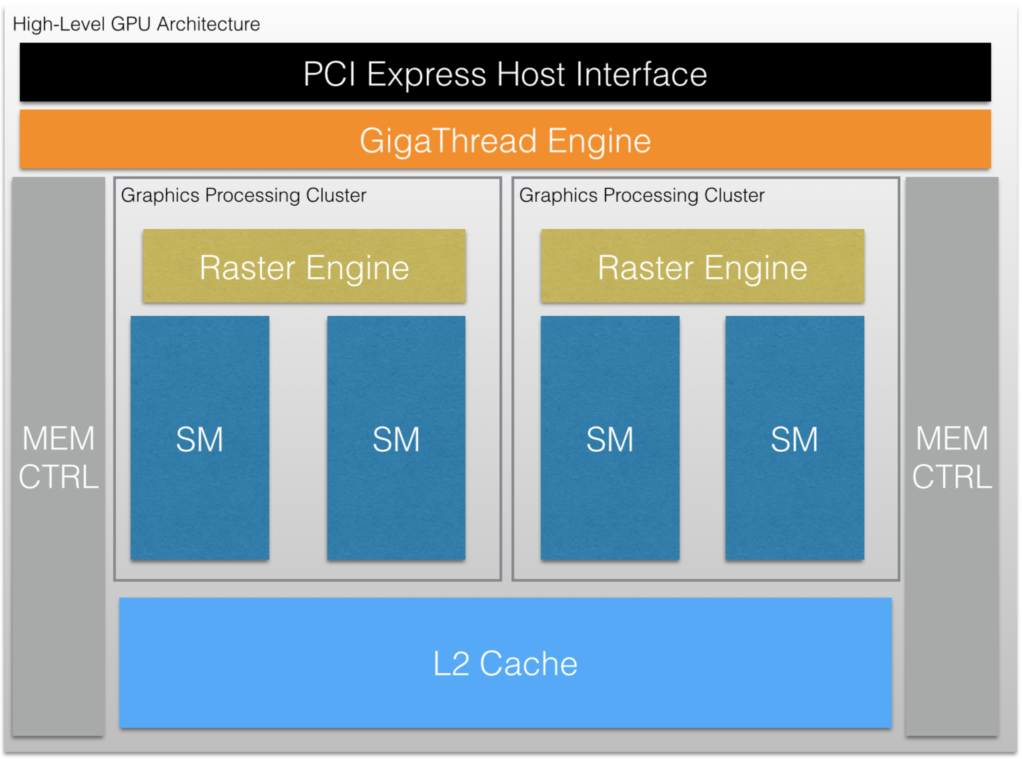

High-Level GPU Architecture
PCI Express Host Interface
GigaThread Engine
Graphics Processing Cluster
Graphics Processing Cluster
Raster Engine
Raster Engine
MEM CTRL
SM
SM
SM
SM
MEM CTRL
L2 Cache

Figure 2.6: High-level GPU Architecture Diagram

In order to help the SM in its execution, there exist important hardware logic outside of the SM. There is a PCI Express Host Interface, used to communicate with the CPU, known as the Host. When the Host wants to offload computations to the GPU, it sends a set of instructions, known as a "kernel", as well as the input data to the GPU. After the Host has sent the data, it is stored in the GPUs Global Memory for the duration of the computation. In order for the GPU to access its Global Memory, it is equipped with several Memory Controllers making it possible for several SMs to access the memory at the same time. This lets the GPU have a high memory bandwidth that can be used to quickly compute applications that are suitable for GPU Computation.

All memory accesses goes through the L2 Cache on this level. The only way to share data between SMs is to use the Global Memory. The L2 Cache helps speedup up data accesses to and from the Global Memory, ensuring that communication time is reduced. Data communication using Load/Store commands uses the L2 Cache to speedup consecutive lookups of data elements, similar to how it works on a CPU. Each Memory Controller has a L2 Cache associated with it, and all data going through each Memory Controller will be cached based on some caching policy.

The Streaming Multiprocessors in the GPU can be organised differently, depending on the main goal when designing the GPU. If the GPU has been designed to be able to render graphics efficiently, the SMs are grouped into several GPC(Graphics Processing Clusters). A GPC contains a number of SMs in addition to a Raster Engine, which is used in the rasterisation process. The GPCs can then be used to process graphics elements fast as part of the rendering process. Figure 2.6 shows a high-level architecture diagram where the SMs are organized into GPCs. The figure is a simplified diagram based on the GM204 Full-chip block diagram[22]. The GeForce GTX 980 is an example of a card where the SMs are grouped into GPCs. The GeForce Series GPUs are designed to deliver high-end graphics performance to gaming enthusiasts, and thus needs to be able to render graphics efficiently. However, when designing HPC-class cards the GPC grouping might not be the most efficient option. The HPC-class cards are designed to be fast at crunching numbers as a primary goal, and they are often not even connected to a screen. Certain rendering logic has thus been replaced in favour of more general purpose components. The total number of SMs in a GPU can vary. Generally, there are between 2 and 16 SM units. NVIDIA offers Graphics Cards at different price points, where the amount of SMs is one of the differing factors.

The final components on this level is called the GigaThread Engine, which is the component that handles scheduling of tasks received from the CPU.

This component receives kernel instructions from the CPU and schedules it onto one or multiple SMs, according to the Kernel Launch Configuration.

2.6.4 The GPU Programming Model

Programming for the GPU can be challenging. It is important for the programmer to correctly ascertain problems suited for the GPU. Generally, GPUs are well suited for applications that are computational heavy and have a high computation-to-communication ratio. The GPU sports a higher memory bandwidth than what is available on a GPU, which means that problems which deal with a large quantity of data and have the above-mentioned SIMD traits, could be suitable for GPU Computing. Matrix Multiplication is an example of a problem that is suitable for off-loading to the GPU. When multiplying matrices, elements needed to calculate a particular part of the matrix is loaded from memory before calculation starts. Since each data element loaded is used multiple times before being written back to memory, GPU Computing is able to efficiently multiply matrices. If Shared Memory is used to minimise the amount of lookup to Global Memory, computation can be performed even more efficiently. In addition to having a high computation-to-communication ratio, the problem also needs to exhibit high Data Parallelism, which is inherent in matrix multiplication problems. The GPU is also suitable for doing large signal processing problems. The Fast Fourier Transform (FFT) problem requires a large amount of bandwidth, which the GPU can provide. Conversely, problems that does not possess SIMD traits might not be suitable for the GPU. For instance, problems that are too unpredictable with a large amount of conditionals would cause too much Warp Divergency, which severely limits performance. In addition, problems sizes that are too small will not be able to benefit from the GPUs strengths, such as high memory bandwidth and the large amount of processing elements.

NVIDIA has implemented its own programming language extension, called CUDA for GPU programming. CUDA exists as extensions to languages such as C and Fortran, and gives developers access to the virtual instructions set of CUDA compatible GPUs. CUDA offers both the ease of use and the low-level access needed for developers to fully utilise the GPU. The CUDA extension gives the programmers a set of new tools needed to properly set up the GPU, transfer data and perform calculations.

When launching a GPU kernel, the programmer needs to supply a set of parameters (the Kernel Launch Configuration) which dictates how many

Streaming Multiprocessors the code is run on. The programming model in CUDA allows the SMs to be organised as the programmers wants to in software. This can be used to model which SMs handle what piece of data, based on the structure of the input data. In the CUDA programming model, all SIMT threads are organised into blocks, which can be 1D, 2D or 3D, based on the needs of the application. Blocks are further organised into grids, which can also be either 1D, 2D or 3D. In the kernel code, each thread can call CUDA functions which tells the thread its position in the block and grid. This can be used to calculate the address of the piece of data belonging to that specific thread. Figure 2.7 shows an example where the blocks are organised in to a 3D grid. Each block is further organised into a 3D collection of threads. This particular way of organising the threads is efficient when doing problems that uses 3D data, such as Matrix Multiplication.

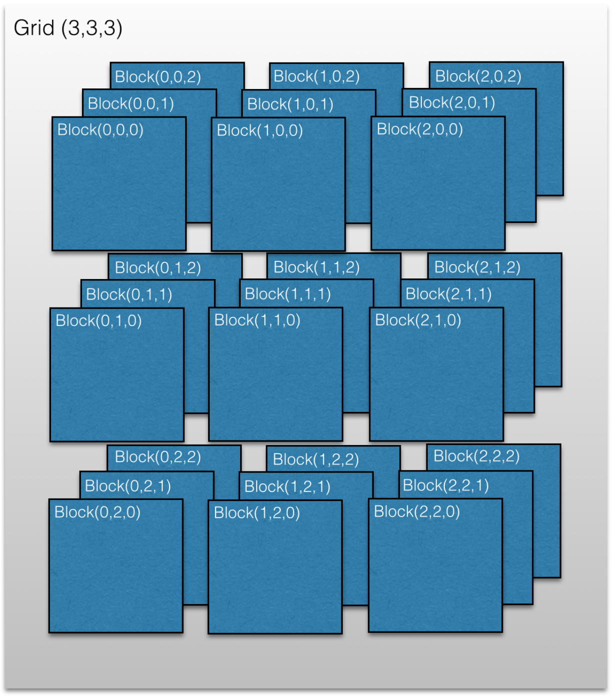


Figure 2.7: An illustration of how data can be divided into CUDA blocks on the GPU.

When launching the kernel, the programmer must specify the block size and grid size to be used in the calculation. This information is sent to the GigaThread Engine to be scheduled on one of the Streaming Multiprocessors. Each block in the grid can be independently assigned to one of the available SMs, which allows for a high level of parallelism. One of the tasks of the GigaThread Engine is to queue up the blocks and assign them to an SM as it becomes available. When assigned to an SM, all threads within a block is guaranteed to be run on the same SM. This allows for Shared Memory to be used when working with threads inside the same block.

When doing computations on the GPU, is is important to choose a block size and grid size that matches the underlying structure of the input data. Performance can vary between configurations, and choosing the best one is often a matter of trial and error.

2.6.5 Running Code on the GPU

This section will explain the steps needed to run code on NVIDIA GPUs using CUDA. To launch a simple compute kernel, there are five general steps:

1. Allocate Host and Device memory
2. Copy memory from Host to Device
3. Specify appropriate amount of threads-per-block and an appropriate grid size
4. Launch GPU kernel code
5. Copy memory back from Device to Host

The first step is to allocate the input data on the CPU. This is usually data taken directly from application input and consequently allocated using standard C malloc(...). In order to copy the Host data on to the Device, we first need to allocate the appropriate amount of space on the GPU. This is done using the special CUDA call cudaMalloc(...). After both of the allocations have been done, we invoke the cudaMemcpy(...) function to copy data from Host to Device.

The next step before execution is to determine the launch configuration of the kernel code. CUDA requires programmers to specify the amount of threads per block and an appropriate grid size, in addition to any function arguments. This is specified when invoking the kernel function, using special CUDA notation. After calling the kernel function, control is immediately returned to the CPU in order for the CPU to continue with something else while waiting for the GPU to finish. Proper synchronisation between the CPU and GPU is controlled implicitly when calling cudaMemcpy(...), which is a blocking function.

After the computation has been finished on the GPU side, it will signal that the data is ready. The CPU will then call cudaMemcpy(..) to retrieve the data from the GPU. When the transfer is done, the CPU is free to use the data for further computation.

2.7 Graphics Rendering

In this section, we will detail the rendering technology used to render the simulation output to the screen. In order to achieve this, we will be using OpenGL (Open Graphics Library)[27], which has been used for many years in various scientific applications. OpenGL is a powerful rendering library which can be used for many purposes. For many of it's features, it works like a state machine. Before drawing to the screen, we first need to 'bind' the data to the internal OpenGL state machine. As such, when drawing objects to a screen, several calls need to be made in preparation before the rendering call itself.

2.7.1 OpenGL Pipeline Overview

OpenGL is a powerful multi-platform rendering API maintained by the Khronos Group. Through this API, the CPU is able to send data over to the GPU to be rendered on a computer screen. The modern version of OpenGL has gone through several changes in recent years. With the recent increase in programmability on the GPU side, graphics libraries in general have adapted to a more dynamic approach, diverging from it's fixed pipeline history. While the general structure of this pipeline has remained intact, developers have been given much more freedom as the GPU started becoming more general purpose.

The goal of the OpenGL's graphics pipeline is to rasterize a 3D scene into a 2D imagine on the computer screen. Depending on the rendering goal, the process can be quite complex. This process is divided into several steps, each performed serially in a pipeline fashion. The data is sent to the GPU in the form of vertex data. A vertex is a geometric point that specifies the intersection of geometric shape. As such, the CPU also need to specify the geometric primitive it wishes to render. In OpenGL there exists various primitives such as points, triangles, lines etc. After the GPU receives vertex data from the CPU, the data goes through a transformation pipeline, where the vertices are transformed. In the transformation process, it is determined how each object should be placed in relation to each other during the rendering. After processing all the transformations, the vertex data is rasterized to the computer screen. In the rasterization process, vertex data is converted from a 3D scene to a specific pixel on the 2D computer screen.

In many graphics applications, the transformations are supplied by the CPU at the time of sending the initial vertex data to the GPU. The GPU will then apply these matrix transformations to each of the vertex in the data input. In the programmable graphics pipeline, the programmer is able to fully program how these transformations are applied through the vertex shader. A shader is a set of instructions very similar to a CUDA kernel that instructs the GPU in how to process each of the input vertices. The vertices are then processed by the GPU in an efficient and parallel fashion.

When doing transformations, there are generally three matrices that we need to multiply the input vertex data with.

1. Model Matrix

2. View Matrix

3. Projection Matrix

The model matrix transforms the vertex data from model coordinates (which specifies the geometric structure of the object itself) to world coordinates(where several objects are placed in relation to each other). The product of this is then multiplied with the view matrix, which puts the objects into camera coordinates. If we imagine the screen as a camera, this transformation puts the position of the camera in the origin of OpenGL's coordinate system, looking down the negative Z axis. The transformation then places the vertex data in relation to the camera's position and orientation around the origin.

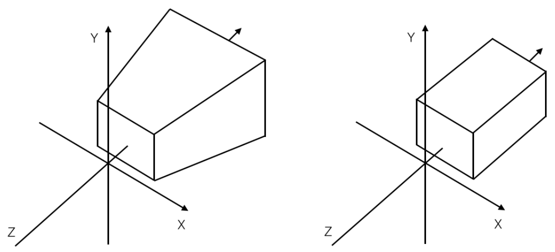


Figure 2.8: An illustration showing the difference between two common projections. Left: A perspective projection. Right: An orthographic projection

Finally, we multiply the results with a projection matrix. This final transformation puts the vertex data into clip space. In clip space, each vertex is evaluated based on its distance to the camera. Based on the projection matrix supplied, this process eliminates vertices which are either too far away or too close to the camera. Vertices which gets eliminated will not be rendered to the screen. When defining a 3D scene, programmers have the option of choosing several different projection matrices. In the picture above, we have included an example of two of them. With a perspective projection matrix, each object inside the clipping boundaries will be rendered. However, as the vertices gets farther and farther away from the camera, the clipping space in the XY-plane gets larger, causing the vertices to appear smaller on the screen. Using this approach, it is possible to realistically render a scene with a proper perspective, where near objects appear larger than far away objects. With an orthographic projection, the clipping space in the XY-plane remains the same size for all objects within its boundary. As such, objects rendered using this projection scheme will not become smaller as they move farther away from the camera.

By looking at how OpenGL works, it is clear that most of the work of moving the objects around is done in the vertex shader. However, since our own simulations are moving the objects around, we simply need to give the location data of the objects to OpenGL to render, without having do to more work in the shader. The problem, then, is how should we share the data between the renderer and the simulation in CUDA? In most cases, the data is inserted from the host side using commands to OpenGL. However, in the case of doing simulations on data already on the GPU, we end up with an unnecessary and costly round-trip. Luckily, NVIDIA has provided developers with a feature that allows us to by-pass this, and fetch the memory directly where its located in OpenGL's device-side memory.

2.7.2 OpenGL and Cuda Interoperability

OpenGL and Cuda can interoperate using a set of simple calls, as shown on NVIDIA Cuda documentation pages[23], and in more detail in various presentations held by NVIDIA[20][21]. The feature has gone through various revisions, and finding up to date information has been challenging. However, by using this interoperability, we are able to significantly speed up the time it takes to communicate between OpenGL and CUDA. Using CUDA API calls, we can register an OpenGL buffer to get its location in GPU memory, and use its data. As such, the data must first be allocated on the CPU side and sent to the GPU through OpenGL.

The process of registering the data to OpenGL and accessing it through CUDA is summarized below.

1. Allocate the vertex data to OpenGL in a Vertex Buffer Object using the glBufferData(...) API call.

2. Register the vertex data with CUDA by calling the cudaGraphics-GLRegisterBuffer(...) CUDA function.

3. Whenever CUDA wants to access the data it needs to call cudaGraph-icsMapResources(...) to map it for immediate use with CUDA.

4. In order to get an accessible pointer to the data that CUDA can use, we call the cudaGraphicsResourceGetMappedPointer(...) function.

5. When CUDA is done working with the data, we must call cudaGraph-icsUnmapResources(...) to unmap the data and make it ready for rendering again.

It is important to note that OpenGL is unable to access the buffer while CUDA has mapped the buffer for use in step 4 and 5. This is in order to prevent errors that would happen if CUDA were to write to the memory while OpenGL was rendering it. After step 5, OpenGL is again free to access the buffer data. This access pattern creates a rotation where CUDA has access to the memory when it's doing N-body calculations, while OpenGL has access to the memory during it's rendering process.

2.8 Other Related Works

This section will detail Related Works that touches upon topics that are similar to what we will explore in this thesis. Reading through these papers has provided valuable insight into the current state of N-Body simulation, and has provided a source for ideas to explore in this thesis.

In the paper Desell et al.[7], the authors seek to utilize N-Body simulations in order to determine the structure of the Milky Way galaxy's halo. This requires a significant amount of computational power, which they get from using volunteer computing. This paper uses the N-Body simulations in order to gain insight about how debris is formed around the halo of the Milky Way galaxy. This is done by using a modified Barnes-Hut algorithm, that takes advantage of distributed heterogeneous computing platform Milky-Way@Home. This paper is an example of using the N-Body algorithm to gain insight how real-world physics apply to a galaxy and see how the structure of the galaxy changes as the simulation goes on.

In the paper Noriega et al[30], the authors utilize an N-Body algorithm in order to do molecular dynamics simulations. This is done using GPU Computing on a CUDA enabled GPU. The results of the simulations are then rendered to a screen using the OpenGL framework. In order to achieve this, they design their own force calculations and integration scheme in order for the simulation to run sufficiently efficient. They also show that the pre-rendering work is more efficiently done in parallel on the GPU.

In the paper Gharakhani et al[34], the authors want to tailor an N-Body algorithm faster than $O(N^2)$ for execution on the GPU. They do this by implementing a GPU-accelerated multipole treecode solver that is able to calculate large and dynamic N-Body problems. The paper details the implementational details and how to repurpose the algorithm for execution on the GPU. When running the algorithm on a single GPU, they were ble to get 17 times the performance of an optimized dual-core implementation.

In the paper Hamada et al[13], the authors design and implement two hierarchical N-Body simulations for execution on a cluster with 256 GPUs. The first algorithm is a modified version of the Barnes-Hut treecode, which is used simulate a gravitational N-Body problem. The second algorithm is a periodic Fast Multipole Method, which they use to perform vortex particle calculations to simulate homogeneous isotropic turbulence. Using their implemented approaches, they gain better performance than previous implementations. The N-Body simulation was able to perform 42.15 TFlops consistently, while the vortex particle simulation had a consitent performance of 20.2 TFlops.

In the paper Hall[12], the author analyses how different integration methods perform in different scenarios. In order to meet performance and accuracy demands, he argues that it might be necessary to switch integration step size, as well as integration method whenever the previous methods become unsuitable. The paper shows and compares the error from two integration methods, the finite difference method and the Runge-Kutta method when used to integrate the movements of a body in orbit. In the paper, he shows an example of an instance where the finite difference method performs catastrophically, while the Runge-Kutta method remains accurate. Following these results, the author suggests and develops an integration scheme where one integration method is switched out when it approaches a state where it is deemed inaccurate.

Chapter 3

Our Implementation: Celestial Simulator

In this chapter we will take a detailed look at the implemented application, the Celestial Simulator. This application encompasses the three main components we outlined in Chapter 1.2. This application includes the galaxy generator, the N-Body simulator and the integration component. These component are highly dependant upon each other, which is why we opted for designing a larger application where each of the components can work efficienty together. The first section of this chapter will provide a general overview of the application. After this, each section will introduce each of the main components of the Celestial Simulator.

3.1 Implementation Overview

The implemented application has been named the The Celestial Simulator. It encompasses a galaxy generator, a renderer, as well as a simulator used to simulate the generated galaxy. The program was written in CUDA C++ targeting the newly released CUDA 7.0 runtime, running on a Ubuntu 14.04 operating system. A useful new feature in the latest CUDA version is the support for the new C++11 standard, giving us more flexible tools when writing software applications[15]. The Celestial Simulator has several main components which encompasses the functionality needed to create a galaxy generator, a renderer and a simulator using one of the N-body algorithms. The program has been implemented as a command line tool which takes

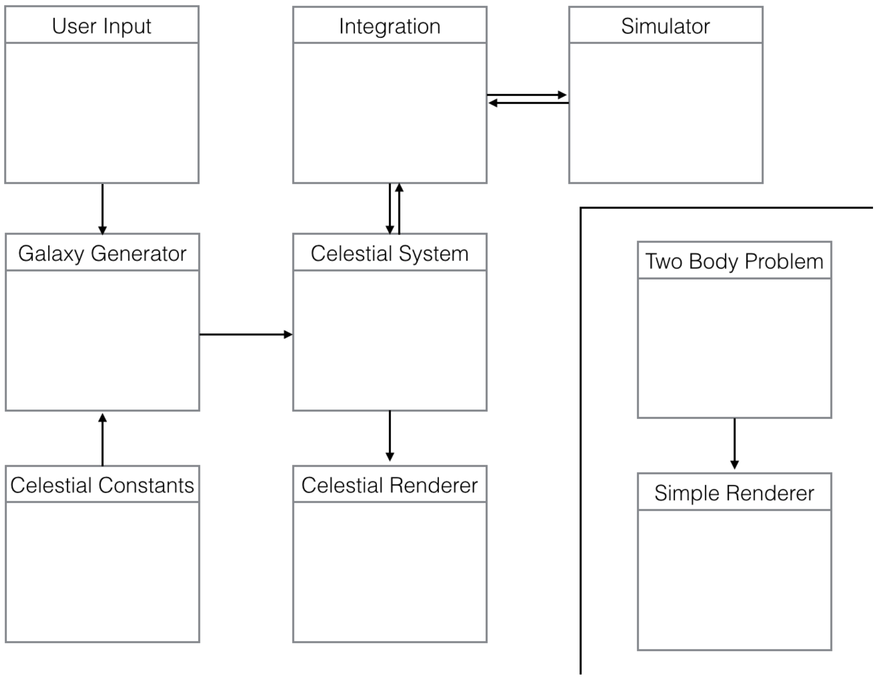


Figure 3.1: An schematic overview of the implemented solution. The arrows shows the direction of the flow of information

several arguments from the user at run-time. In the event the user requests the simulation to be rendered, a window will open to display the real-time simulation results. In addition to this, there are also several test cases and comparisons tests that can be run in order to compare the different N-body algorithms, which will produce various types of result reports after concluding the simulation.

The main components are:

- **Input** Serves as an input module for the Galaxy Generator. It combines User Input with the defined Celestial Constants in order to create a galaxy.
- **Galaxy Generator** Processes the input and uses celestial mechanics theory in order to generate the initial position and velocity for each body in the galaxy.
- **Celestial System** An object class that keeps track of all the data

needed for the simulation and rendering. All rendering and simulation calls go through this module.

- **Integration** Takes input from the user in order to decide the integration method and N-body algorithm to use. As the integration is closely tied to the N-body simulation itself, this module has the responsibility to also call the N-body module when needed.
- **Simulator** Takes in body information as input, and performs computations on it using an N-Body algorithm. As there are several algorithms, the one to use is determined by User Input at the time of program execution.
- **Celestial Renderer** Receives the simulated data from the Celestial System module and renders it to the screen based on simulation runtime input.

In any execution of the program, the first component that is invoked is the Input Module. This module analyses the command line arguments that were supplied and sets the desired values to the specified global variables, for use with the other components of the program. The second part of the Input Module defines the constants of various variables that are needed when creating solar systems and the galaxy. This class contains various information about the bodies in our solar system such as mass and distance from the Sun. These are used by the Galaxy Generator to create solar systems that mimics our own for use by the simulator.

In addition to the main components, there are also two modules that were used during development as experimentation tools

- **The Two Body System Generator** Used during development as a first implementation of the galaxy generator. This module generates two body systems as well as solar systems and tests how close the N-body simulation is to the analytical result of a two body system.
- **Simple Renderer** Used in conjuction with the initial Two Body System module to show the results of the comparison. Also works as an initial implementation of the Celestial Renderer.

Figure 3.1 shows all components of the implemented program and how they relate to each other. The flow of information through the program is shown by the directional arrows. As we can see, the Celestial System class is at the core of the program. It gets instantiated with the proper values through the galaxy generator and input module. After intialization, it stores the information and continously share it with the integration module when moving the bodies, and shares the values with the renderer if it is enabled. A full installation guide and user guide is available in Appendix A.

3.2 The Two Body System Generator

The Two Body System Generator is an experimental component used to test how the results from the N-body simulation differ from the analytical solution. The code is based on the laws of both Kepler and Newton as detailed in Chapter 2 and very succintly explained in the book ”Solar System Dynamics” by C.D Murray and S.F Dermott[28].

By using Kepler's Equation[35] to calculate the eccentric anomaly E from the mean anomaly M, it is possible to analytically calculate the movement of an orbiting body with respects to time. From equation (2.11), (2.20) and (2.21) derived in Chapter 2, it is possible to calculate the start position and start velocity that the body needs in order to move in a given orbit defined by the eccentricity e and the length of the semi-major axis a of the orbit's ellipse. This principle is used in order to solve the initial value problem for each of the bodies in the solar system. We then run the simulation with the generated start values and compare it to the analytical version, calculated from equation (2.27).

We can also build an entire solar system using this method. By creating multiple two body systems for each star-planet pair, we can approximate the orbit which the planets will have around the Sun. It is important to note that due to the complexity of all the forces working together in a system, that there does not exist an analytical solution for an N-body problem. When we create a solar system consisting of N bodies by chaining together multiple two body problem pairs, each orbit is only an approximation of it's actual value. While the Sun will exert the most amount of force on its orbiting planets, each individual planet will also exert a force on the other planets. This secondary force is known as a perturbation, which is a smaller force that affects the orbit only slightly compared to the primary force. In order to test the effect of this perturbation, we have designed a second test where we compare how the orbits of the planets of the inner solar systems behave differently when there are multiple planets in the system. Once again, we compare the simulated values to the analytically calculated values

3.2.1 Two Body Problem Test Results

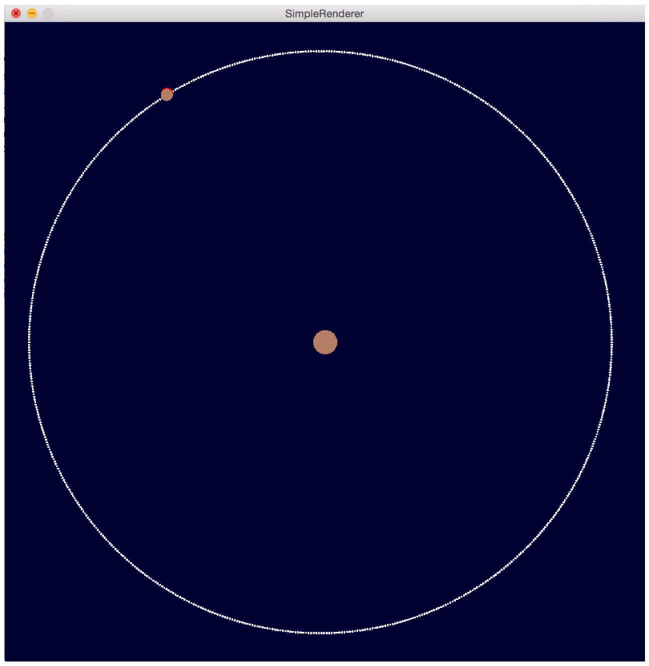


Figure 3.2: A picture of the Two Body test rendered using SimpleRenderer

In this section, we will present the results when comparing the analytical and simulated values of a system of two bodies. In Figure 3.2 we can see a screenshot of the simulation being run. We will be testing the Sun-Earth two body system, showing how the Earth will move about the Sun. The values used to specify the orbit such as the semi-major axis a and the eccentricity of the orbit's ellipse can be confirmed in Appendix B. The values for mass and distance have been normalized to Earth's values. The unit for mass is expressed in Earth Masses and the unit for distance is expressed in Astronomical Units (AU), which is the average distance from the Earth to the Sun. When doing simulation, we express the timestep in Earth days. A timestep of $dt = 1.0$ thus means that one day on Earth passes for each timestep. The gravitational constant G we use with the above units can be confirmed in Appendix B.

The brown orbiting planet is being moved by the analytical two body method, while the red body is moved using a CPU N-Body All-Pairs algorithm. For simplicity, the Sun is being kept in place. By observing the bodies over time, we can see that they move very close to each other. As such, the N-Body algorithm does a good job of simulating results similar to the analytical method. However, if we observe the simulation over several orbits, it slowly starts to move away from the analytical method. It appears that the simulated body moves slightly faster than the analytically calculated body. Despise this, the simulated body still moves without diverging from the elliptic orbit. While me might lose some accuracy over time, by using the N-Body algorithm with initial values generated by solving the two body problem, we are able to create a stable orbit for a single planet.

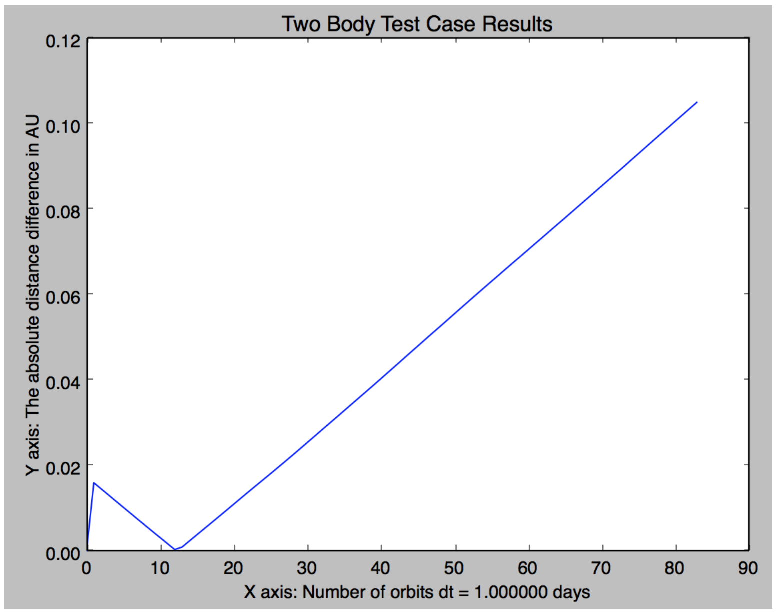


Figure 3.3: The difference between the analytical and the simulated results for the Sun-Earth two body system

By running the simulation, we see that the orbit time of the Earth is about one year. We run the simulation for about 80 orbits, where the absolute value of the error is found to be $\epsilon = 0.105$ AU. If we compare this to the total distance travelled by the Earth during that time, we get

$$E_{relative} = \frac{\epsilon}{80\ Orbits \times 6.3\ AU} = \frac{0.105\ AU}{504\ AU} = 2.08 \times 10^{-4} AU \qquad (3.1)$$

Even though the error is continously rising, the error rate is incredibly small compared to the distance traveled. This small difference will create small inaccuracies, and could propagate to larger errors over a significant amount of time, but it will not affect the stability of the orbit itself.

The reason for the difference between the two methods is most likely related to the timestep chosen for the test. When doing numerical integration, a small error relating to the size of the step size is introduced. In this test, we utilized the Symplectic Euler integration method introduced in Chapter 2 with a timestep of $dt = 1.0$ days. This integration method is of the second order, which means that by halving the timestep, the accuracy will increase four times. Thus, by reducing the timestep, the error we see in the simulation will decrease.

3.2.2 Solar System Test Results

We expand on the previous case by testing the planets of the inner solar system. The planets are as follows starting with the planet closest to the Sun; Mercury, Venus, Earth and Mars. The values used to generate the orbits can be confirmed in Appendix B. Just like the previous test, the Sun is kept in place while the brown orbiting bodies are moved by using the analytical method and the red bodies are moved using the CPU N-body All-Pairs algorithm. In this system, each planet will have a force exerted on it by not only the Sun, but the other planets as well. As such, the solution we are able calculate using two body problem analytics will only be an approximation. In this test, we wish to measure how accurate this approximation will be.

We again observe the bodies over time, and can see that the results are similar to the first test case. The N-Body simulation is able to closely simulate the analytical solution, however as time progresses small errors propagate and the results start to diverge little by little. However, Mercury has a significant amount of error, as high as $\epsilon = 0.8AU$ at around 70 orbits. This is most likely not because of additional pull from the other planets, but rather the constant speed changes in highly eccentric orbits. As explained in Chapter 2, the body will speed up as it gets closer to the Sun and slow down

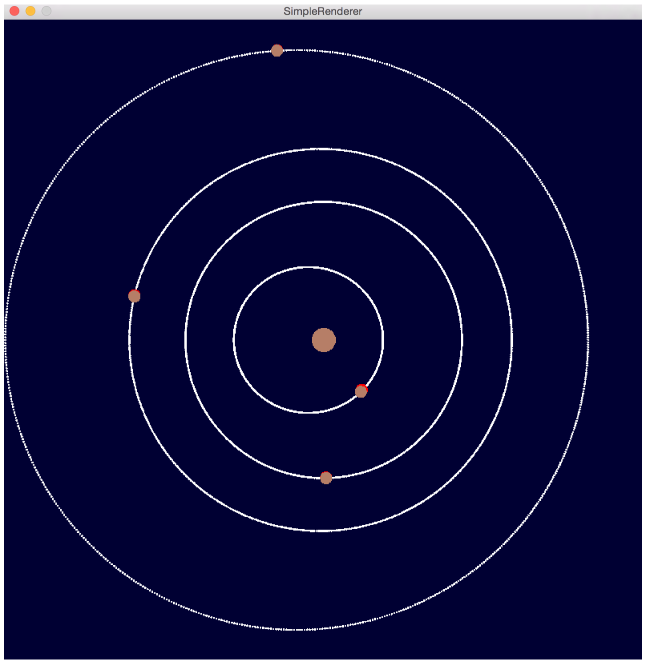


Figure 3.4: A picture of the Solar System test rendered using SimpleRenderer

as it gets farther away. For the other planets in the system, the velocity will not change as much because of their low eccentricity value. Combined with the fact that the distance is closer to the Sun, the orbit of Mercury is much more prone to rapid velocity changes. In order to simulate Mercury's orbit with an acceptable accuracy, we need to lower the timestep. This will allow the simulation to update it's velocity more frequently, thus matching the analytical solution much closer. These two examples goes to show the importance of choosing the correct integration method and timestep for the situation. It is also likely that the error observed for the other planets are related to the step size, thus lowering it would increase the accuracy of the simulation. However, in a real-time simulation environment we need to find a suitable trade-off between performance and accuracy. We will explore the timestep issue further in the Results chapter.

Overall, we see that a small error continue to accumulate during the simulation due to the integration step. The error can be made smaller by decreasing the timestep, but it is important to chose a suitable trade-off between performance and accuracy. In the case of errors caused by small

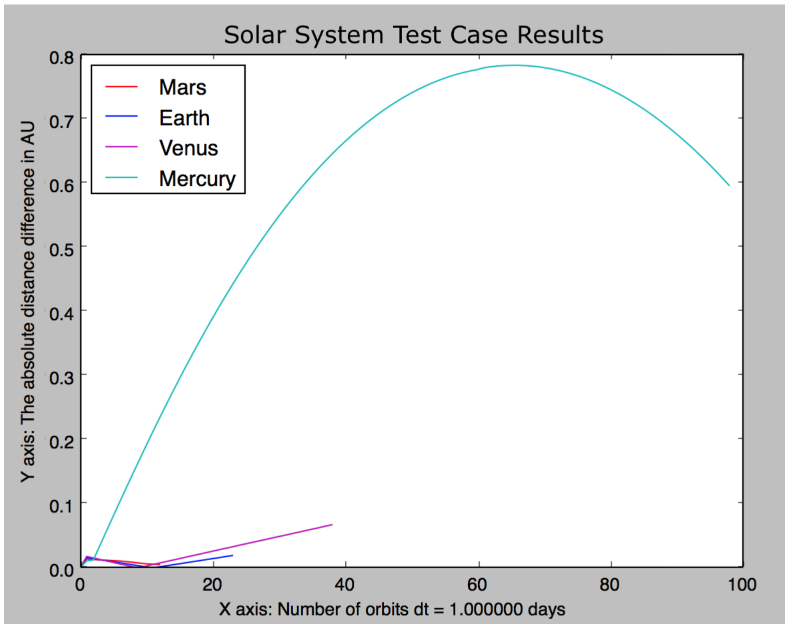


Figure 3.5: The difference between the analytical and the simulated results in the case of the Sun-Mercury-Venus-Earth-Mars system

perturbations by other planets, we can see from the result of Earth's orbit that it is affected very little by the other planets. This is expected, as the Sun is several thousand times more massive than the other planets, which means that the force contribution from the other planets are almost negligible compared to the Sun. However, very tiny perturbations could become significant over a very large period of time. As such, it is not possible to guarantee that the solar system will remain in the same stable orbit for eternity. However, using the method introduced in this section to create stable solar systems appear promising. In the next section, we will take a look at the second iteration of the generator, which expands on this implementation to enable us to generate whole galaxies.

3.3 The Galaxy Generator

The Galaxy Generator is tasked with the generation of a galaxy comprised of several solar systems with a fixed size. It is built upon the implementation featured in the previous section.

The Galaxy Generator is called by the Celestial System during initialization using the function createGalaxy(...). This function takes as input two integers specifying the number and size of the solar systems to generate. In the current implementation, every solar systems is fixed to the same size. However, by redesigning the generator to take in an array of different solar system sizes, it is possible to support different sized solar system. Note that the Celestial N-Body algorithm introduced later in the chapter must also be updated in order for the simulator to support dynamic sizing.

The createGalaxy(...) method does the following steps in order to generate the galaxy.

1. Generate a black hole in the origin of the galaxy with a given mass.

2. For each solar system,

   (a) Place the star in relation to the black hole based on a given inclination angle.

   (b) Find the star's initial velocity by calculating it's orbit velocity around the black hole.

   (c) Rotate each planet to the correct inclination angle, and place each planet in the star's orbit based on desired orbit values.

   (d) Find the combined initial velocities of the planets by adding together the star's orbiting velocity around the black hole and the planet's orbiting velocity around the star.

The values needed to determine the mass of the black hole, as well as the overall structure of the galaxy are read from the User Input module at run-time. In order to find the start position we make use of equation (2.11), which we derived in Chapter 2. Solar systems are distributed around the black hole in circular planes. Each plane has an inclination angle which determines how the plane is orientated with regards to the black hole, and each plane shares the same origin, located at the black hole. The azimuth angle determines the location of the solar system in the plane. Each plane has a corresponding distance r from the black hole associated with it. In order to avoid collisions at intersection between the planes, this distance will vary with each plane. At run-time, the user can specify the total number of solar systems, as well as how many planes the solar systems should be distributed on. To avoid too much clutter in the generated galaxies, we will limit the number of planes to a maximum of two.

We find the start position of the individual planets by once again utilizing equation (2.11). The positions will be spread around in a two dimensional plane. The next step is to use the inclination angle to rotate the plane to its correct start position and then use the location of planet's nearby star to offset it from the origin.

Similarly, the start velocity is calculated by first finding the orbital period and then use this to find the mean motion using equation (2.13) and (2.14). We then use the results of this to calculate the start velocity of the orbiting body in the orbiting plane using equation (2.20) and (2.21). After finding velocity values for the two dimensional plane, we once again rotate it to the correct position using the inclination angle, before adding the velocity of the planet's nearby star. This will create a plane of multiple solar systems orbiting a black hole at a given inclination angle. The simplified pseudo code used to generate the galaxy is shown below

```
for (int plane_i = 0; plane_i < planes; plane_i++) {

    float azimuth = plane_i * azimuth_d + startAngle;
    for (int solar_i = 0; solar_i < solarPerPlane; solar_i++) {

        float inclination = solar_i * inclination_d;

        mass[starIndex] = starMass;

        positions[starIndex] = findSolarStartPosition(distance,
            inclination, azimuth);

        findStartVelocity(0, starIndex, 0, distance, inclination
            , azimuth);

        setSolarSystemValues(starIndex, starIndex + solarBodies,
             azimuth);

        starIndex += solarBodies;

    }

    distance += planes_distance;

}
```

The values are stored in three float3 arrays specifying the current acceleration, velocity and position for each dimension. The mass values are stored in a single float array. The generated values are stored consecutively in memory in each array as shown in Figure 3.6.

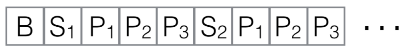


Figure 3.6: Galaxy Data Structure Overview

In the figure, B is the location of the values for the black hole, S_i is the location of the values for the star of solar system i, and the following P_i values are the values for the planets orbiting star S_i.

After successfully generating the values, the body information is made available to the Celestial System for use in simulation and rendering. If the simulation is to be run on the GPU, the generator will transfer the memory to the GPU before returning a pointer to the Celestial System. The full Galaxy Generator code is available in Appendix C.1.

3.4 The Celestial System

As the centerpiece of the implemented application, the Celestial System module is tasked with keeping track of the body information, as well as the current simulation and rendering state. It continosly stores the acceleration, velocity and position for the currrent time. During initialization, the Celestial System takes as input the desired amount of solar systems and the number of bodies within a solar system from the user. These variables are used with The Galaxy Generator to create a galaxy as specified by the user. As we will be working with both CPU -and GPU-based algorithms, the module also takes as input a boolean value specifying if the computations are to be run on CPU or the GPU. Based on this boolean value, the generated body information data is either allocated on the CPU or the GPU using either the standard C++ malloc(...) command or the CUDA equivalent cudaMalloc(...). At initialization, the Celestial System is thus associated with either the CPU or the GPU and will remain so for it's lifetime. As such, the way a host-allocated Celestial System will interact with the renderer is quite different to a device-allocated one.

```
void CelestialSystem::drawBodies() {

  if (renderingEnabled) {
    updateBufferCPU();
    glBindVertexArray(systemVAO);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glDrawArrays(GL_POINTS, 0, getN());
    glBindVertexArray(0);
  }

}
```

The drawBodies() function is called every frame by the Celestial Renderer. The drawing itself is done by issuing four important calls to OpenGL. The first call, glBindVertexArray(...) locates and binds the array which contains the vertex information for all the bodies in the system. By binding it, we tell OpenGL that we want to use this buffer when issuing the final rendering call glDrawArrays(...). The second glVertexAttribPointer(...) call specifies how the information is stored in the buffer and tells OpenGL how to interpret it. In this case, we specify to OpenGL that the data is bundled into groups of 3 single-presicion floating point variables. The final call unbinds the vertex buffer array that we just used, which is common practice in order to minimize any bugs associated with drawing from the wrong buffer. Now, in the case of running the simulation on the GPU using CUDA, the vertex buffer will continously be in GPU memory for both the drawing and the simulation. This allows us to quickly render and simulate without any memory transfer overhead. However, when running the simulation on the CPU, the most up-to-date memory will be located in the CPU during the simulation stage. This means that every time the CPU updates the positions of the bodies, the buffer located in the GPU will need to be updated. As this happens every frame, the memory transfer overhead is significant. Unfortunately, since the renderer must have the data in GPU memory in order to render, this overhead is unavoidable. The vertex buffer located on the GPU is updated using the glBufferData(...) call, which will initiate the transfer from CPU to GPU memory. The call takes as input the size of the data to be transfered and it's location in CPU memory. We can also specify a hint to OpenGL on how the data will be used. This will potentially allow OpenGL to make significant optimization on how it accesses the data. In our case, we will only draw the data one time before it gets outdated. As such, we issue the GL_STREAM_DRAW hint to tell this to OpenGL. The following code defines the updateBufferCPU() function used to update the buffer.

```
void CelestialSystem::updateBufferCPU() {

  if (renderingEnabled && !useGPU) {
    glBindBuffer(GL_ARRAY_BUFFER, systemVBO);
    glBufferData(GL_ARRAY_BUFFER, 3 * sizeof(float) * getN(),
        position,
         GL_STREAM_DRAW);
  }

}
```

The other important method defined in the Celestial System class is the moveBodies(...) function. As the name implies, it calls the simulation module in order to move the bodies a timestep *dt* forward.

```
void CelestialSystem::moveBodies(float dt) {

  mapResourcesCuda();

  performIntegration(dt);

  unmapResourcesCuda();

  updateCurrentTime(dt);

}
```

Before moving the bodies, in the case of the GPU, we need to map the resources for exclusive use by CUDA, as explained in Chapter 2. The methods mapResourceCUDA() and unmapResourcesCUDA() implements the functions needed to map the resource before execution, and unmap the resources after we are done moving the bodies. When running the simulation on the CPU, these methods simply do nothing.

The simulation itself is done by calling the performIntegration(...) function with the desired timestep *dt*. We will be using various different integration schemes, each which may use the current or past acceleration in different ways. As such, it is the duty of the integration module to call the N-Body system as appropriate, either before or after integrating the acceleration to get the velocity and position information.

The remaining functions defined in the Celestial System are used in order to retrieve data, debug the body information, or properly set up and tear down the system when running the simulation with a renderer. For instance, the setupForRendering() function prepares the system for rendering. If the user specifies that the simulation should be rendered, this function will be called in order to create the needed vertex buffer and update it with the body information. In addition, if the simulation is to be run using CUDA, we do the initial registration needed in order for OpenGL to communicate with CUDA here.

3.5 Celestial Renderer

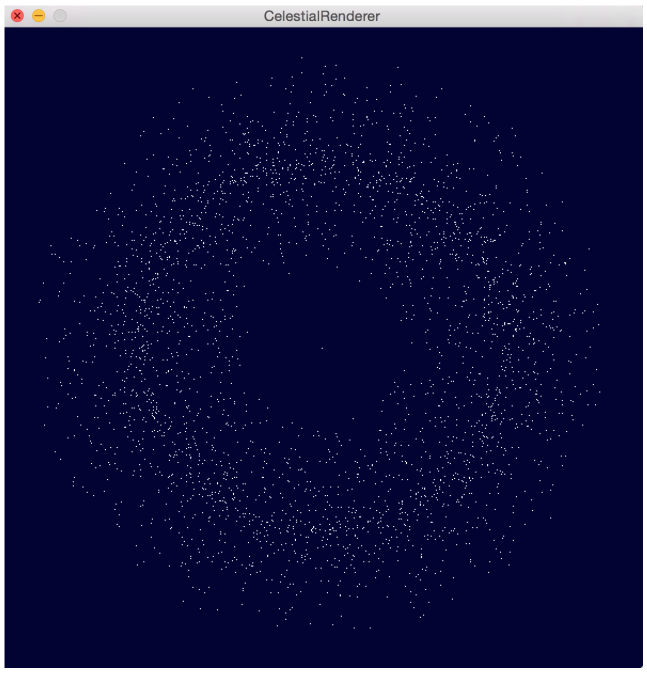


Figure 3.7: A picture of an example galaxy rendered by the Celestial Renderer

As speed of computation is an important factor in the application, we will focus on creating a renderer that is able to convey the state of galaxy using simple graphical elements in order to minimize rendering overhead.

The renderer was developed by carefully reading through the OpenGL 3.3 Reference Pages[26], and choosing the functions and features best suiting our needs.

While the fixed rendering pipeline is very simple and has been used many times in scientific applications, it's features are very old and close to being deprecated on many systems. As such, even though it adds a bit of complexity to the implementation, we have opted to use the programmable pipeline offered in OpenGL 3.3.

Before we can create an OpenGL context to render graphical objects to, we need to create a window from the operating system. As this is highly OS dependent, we will be using a cross platform development library called Simple DirectMedia Layer 2 (SDL2)[24]. Using SDL2 we are able to request a window from the operation system, which we can then use to render OpenGL elements through the use of an OpenGL context. In addition, using SDL2, we are also able to take run-time keyboard inputs from the user.

In our renderer, we will be using the OpenGL graphics primitive GL_POINTS. This primitive will render a single fixed-size point in 3D space for every vertex we supply the rendering function. As it is just a single point, it will be quick to render, ensuring that the renderer is as fast as possible.

As explained in Chapter 2, the positions of the points in relation to each other will be calculated by the simulation. Because of this, the only matrix transformation we need to apply is the perspective transformation, in order to transform the points into proper 3D space. However, since the GL_POINTS are defined with a fixed size, they will not be affected by a projection transform. For our purpose, it is better to use an orthographic transform, which will ignore perspective size differences for object that are farther away from the camera. By using this renderer approach, we will be able to easily convey the state of the bodies in the galaxy and how they move in relation to each other. The keyboard inputs to operate the Celestial Renderer has been included in Appendix A.

```
void CelestialRenderer::runRenderingLoop() {

  while (!shouldQuit) {

    handleInput();

    prepareToRender();

    renderBodies();
```

```

        moveBodies();

        SDL_GL_SwapWindow(window);

    }

    SDL_GL_DeleteContext(glContext);

    SDL_DestroyWindow(window);

    SDL_Quit();

}
```

The renderer class is also responsible for keeping the application running and updating based on simulation calculation and user input. This is done through the main rendering loop, as seen above. For every iteration of the main loop, the renderer has several calls to perform. The first call is handleInput(), which will query the keyboard and check if the user is currently pressing any buttons. Based on what the user is pressing, the function will adjust the appropriate variables to their new value. In the event that the user wants to quit the application, the handleInput() method detects this, and sets the shouldQuit flag to true. After exiting the loop, we delete the context and the window and call SDL_Quit(), which performs the necessary cleanup for the SDL2 framework. The next function to be called, prepareToRender() prepares the renderer for the next draw call to be made. During this process, it cleares the necessary buffers and resets the screen in order to render a new frame to it. In addition, it will prepare to send the current transformation matrix to the vertex shader. This needs to be done before every draw call in the event that the user has moved the camera position during the last frame. The drawing itself is done in renderBodies() by calling the drawBodies function for the associated Celestial System. Finally, in the moveBodies() function, the bodies are moved one timestep by calling the Celestial System's moveBodies() function with a given timestep *dt*. The last call is a SDL2 function which swaps the current framebuffer with the one we just drew to. This will present the new frame to the screen.

3.6 N-body Algorithms

Simulation is done by utilizing one of the implemented N-Body algorithms to calculate the acceleration exerted on a body by the other bodies in a system. We have implemented several different N-Body algorithms, both for the CPU and the GPU. The N-Body All-Pairs GPU algorithm used in this thesis is based on NVIDIA's GEM3 N-Body algorithm, as mentioned in Chapter 2. We have also implemented our own N-Body algorithm, The Celestial N-Body algorithm, which takes a different approach from GEM3. While GEM3 is a general algorithm that can compute any N-Body system with $O(N^2)$ complexity, our approach is a specialized algorithm that is designed to work with the generated galaxies specifically. In the next sections, we will describe each of the N-Body algorithms in more detail.

```
float3 computeBodyBody(float3 i_pos, float i_mass, float3 j_pos, float j_mass, float3 i_acc) {

  float3 distanceVector = calculateDistanceVector(i_pos, j_pos);
  if (distanceIsZero(distanceVector)) return i_acc;

  float absoluteDistance = calculateAbsoluteDistance(distanceVector);
  float absoluteDistanceCubed = absoluteDistance * absoluteDistance * absoluteDistance;
  float3 dA = calculateAccelerationChange(j_mass, absoluteDistanceCubed, distanceVector);
  i_acc = sumAccelerationChange(i_acc, dA);

  return i_acc;
}

float3 calculateAccelerationChange(float j_mass, float absoluteDistanceCubed,
    float3 distanceVector) {

  float3 dA;
  dA.x = G * j_mass / absoluteDistanceCubed * distanceVector.x;
  dA.y = G * j_mass / absoluteDistanceCubed * distanceVector.y;
  dA.z = G * j_mass / absoluteDistanceCubed * distanceVector.z;
  return dA;
}
```

A core part of the N-Body algorithm is how the pairwise body computations are done. The code above showns the simplfied psuedo code for the implementation of the pairwise body computations. It implements equation (2.31), discussed in Chapter 2. This piece of code will be used in all the N-Body algorithms which we will implement. While a lot of the code will remain the same, the different N-Body algorithm implementations will not call this function the same number of times. For instance, the Celestial N-Body algorithm will use a center of mass approximation in order to reduce the number of times this function is called. This allows us to speed up the overall computation time.

3.6.1 N-Body All-Pairs CPU

The CPU N-Body All-Pairs algorithm is a simple serial algorithm with a computational complexity of N^2. While this method is very simple to implement, it is a simple brute-force method which scales badly as the number of bodies increases.

```
void nbody_allpairs_cpu(float3 *acceleration, float3 *position,
    float *mass, int N) {

     for (int i = 0; i < N; i++) {

        float3 i_pos = position[i];
        float i_mass = mass[i];
        float3 i_acc = make_float3(0,0,0);
        for (int j = 0; j < N; j++) {

                    if (i == j) continue;
          float3 j_pos = position[j];
          float j_mass = mass[j];
          i_acc = computeBodyBody(i_pos, i_mass, j_pos, j_mass,
              i_acc);

        }

        acceleration[i] = i_acc;
     }
}
```

As this method computes the values with no optimizations, it is also expected to be the most accurate. Other approaches that are not All-Pairs will use some kind of heuristic in order to approximate the extertion felt by a given body. When comparing the results of the other algorithms, we will be using the CPU All-Pairs results as a base of comparison.

3.6.2 N-Body All-Pairs GPU (GEM3)

The GPU All-Pairs algorithm we will be using was first detailed in NVIDIA's GPU Gems 3[31], released in 2007. The book features an All-pairs N-Body algorithm that while still has a complexity of $O(N^2)$, uses the power of the GPU to efficiently calculate the pairwise forces in parallel. It does so by defining tiles of bodies to calculate and assign this tile to different Streaming Multiprocessor for quick parallel execution. This allows us to significantly reduce the computational time when running the N-Body problem. However, in order to maximize the occupancy we need to choose a suitable block size. The optimal size is dependent upon the GPU being used, however, NVIDIA recommends that the block size shoule be a multiple of 32 to coincide with the current warp size. The block size to use when running the GPU N-Body All-Pairs algorithm can be decided by the user at application run-time.

Since this method is also doing All-Pairs computation, we can expect the results to be close to the CPU All-Pairs algorithm. However, due to differences in how floating points operations are done on the GPU, the results may differ somewhat[37]. As each computation is based on the previous one, small errors can continously propagate to create much larger errors. This can become a significant problem, which is something that must be taken into account when comparing the algorithms

3.6.3 The Celestial N-Body Algorithm

By considering how the values from the Galaxy Generator are stored, we can create a specialized algorithm that takes advantage of this in order to speed up computations. We know from Figure 3.6 how each of the bodies inside a solar system are located in memory. At the time of generation, we also know that these bodies are grouped together in a solar system. The Celestial N-Body algorithm assumes that each of the bodies inside one solar system will continue to move together during the simulation. Since we know that the bodies inside a solar system are close to each other, we can approximate their combined exertion on a body outside the solar system given that the solar systems remain far apart from each other.

```
bool CelestialNbody::celestialNbody(float3 *acceleration, float3
    *position, float *mass, int solarSystems, int solarBodies) {

  float4 *massCenters = calculateInternalAcceleration(
      acceleration, position, mass, solarSystems, solarBodies);

  bool thresholdReached = calculateExternalAcceleration(
      acceleration, position, mass, massCenters, solarSystems,
      solarBodies);

  return thresholdReached;

}
```

The Celestial N-Body algorithm is divided into two functions that work together in order to approximate the total acceleration exerted on a body i by the other bodies in the galaxy. These functions are summarized in the code above. In order for us to accurately simulate the orbits of the planets within each solar system, we need to carefully calculate the body interactions among each member of a solar system. Therefore, when calculating the internal solar system accelerations of a body, we perform $O(N_{SB}^2)$ computations, where N_{SB}^2 is the number of bodies (one star and several planets) inside the solar systems.

We will be working with galaxies generated with a realistic number of solar system bodies. Current observations suggest that the highest number of planets inside one solar system is nine[36]. Because of the small number, we do not divide the computations into tiles, like in GEM3. Instead, we have implemented two versions of the internal acceleration algorithm that calculates the internal acceleration felt by a body slightly differently. In the standard version, we set the GPU Block Size equal to N_{SB}^2. This will allow each solar system to be calculated independently in parallel on separate CUDA cores. However, for small solar system sizes, the blocks will become very small, which does not utilize the GPU very efficiently. The Celestial N-Body Grouped algorithm fixes this by grouping together as many solar systems as possible into one block, where the block size is specified by the user. This approach will make the algorithm efficient even for small solar systems.

The acceleration felt by a body because of the external solar systems are calculated by using the center of mass of the external solar system. We compute the center of mass of each of the solar systems in the internal acceleration and use the computed mass centers to approximate the acceleration felt by distant bodies. This allows us to speed up computation significantly. The black hole is treated as a special case and computed directly. If we define the number of solar systems as N_{SS} and the total number of bodies as N, we can express the complexity of the algorithm with $O(N_{SS}N_{SB}^2 + NN_{SS})$. The full code for the Celestial N-Body algorithm is available in Appendix C.2, while the code for the Celestial N-Body Grouped algorithm is available in Appendix C.3.

In order for the external approximation to be accurate, we need to assume two things. The first assumption is that each of the bodies inside a solar system remains tightly coupled together in orbit around a common star. The second assumption is that the solar systems are far away from each other. Since the Celestial N-Body algorithm require these assumptions to be valid, the approximation will create large errors if the assumptions were to become invalid. If this happpens, we need to switch the algorithm over to a more general one (for example the All-Pairs algorithm) in order to maintain the accuracy of the simulation. This is done by checking the distance between each of the solar systems and notifying the calling function if the solar systems are too close to each other, determined by some threshold. The threshold is checked and a boolean value is returned when calling the calculateExternalAcceleration(...) function.

3.7 Integration

Choosing an appropriate integration method will be crucial in order to ensure the accuracy of the simulator. Several types of different integration schemes have been implemented in order to compare which ones are the most suitable for our simulator. This section will detail the implementation of the two integration methods explained in Chapter 2 and introduce a custom integration scheme.

3.7.1 Explicit Integration Implementation

```
For (i = 1; i < N; i++) {

  velocity_next = velocity[i] + acceleration[i] * dt
  position_next = position[i] + velocity[i] * dt

  velocity[i] = velocity_next
  position[i] = position_next

}
```

This integration method implements the integration formulas (2.32) and (2.33) explained in Chapter 2. The code above shows the pseudo code for the calculation steps. For all bodies except the black hole, which we will keep in place, we perform the above operations. Since we are working with vectors, in the code above, we calculate the results of the formula for each dimension independently. The code above shows the pseudo code for the CPU version. The GPU version computes the same calculations, but instead launches several blocks of threads in order to go through all N-1 elements in parallel.

3.7.2 Symplectic Euler Implementation

```
For (i = 1; i < N; i++) {

  velocity_next = velocity[i] + acceleration[i] * dt
  position_next = position[i] + velocity_next[i] * dt

  velocity[i] = velocity_next
  position[i] = position_next

}
```

The pseudo code above implements the integration formulas (2.32) and (2.34) which we discussed in Chapter 2. The calculations are very similar to the previous approach, however this time we use the next velocity at time $t+1$ to calculate the next position. This implements a symplectic integration method where one variable is calculated using the current value at time t and the other is calculated using the updated value for time $t+1$. Since we already know the acceleration at time t, we know how to compute both the Symplectic and Explicit Euler integration method without doing expensive intermediary computations.

3.7.3 Celestial Integration Implementation

In addition to the two integration methods mentioned above, we have also implemented a custom integration scheme, called Celestial Integration. This is a custom approach specialized to the type of simulation we are doing in this project. It has been built to support the Celestial N-Body algorithm and as such can only be utilized together with either the grouped or standard Celestial N-Body algorithm. The integration scheme itself is used in conjuction with one of the two integration methods detailed above, which can be decided at application run-time.

The Celestial Integration scheme was implemented with both precision and performance in mind. By considering the fact that the acceleration exerted by distant forces on a body will change much less frequently than bodies within the solar system, we can implement an integration scheme that takes advantage of this.

```
performCelestialIntegration(int steps, float dt) {

  float4* massCenters = calculateMassCenterForSolarSystemsGPU(
      position, mass);
  float3* acceleration_ext = calculateExternalAccelerationGPU(
      acceleration_ext, position, mass, massCenters);

  float internal_dt = dt / steps;

  for (int i = 0 ; i < steps ; i++) {

    calculateInternalAccelerationGPU(acceleration, position,
        mass);

    //Explicit or Symplectic Integration possible
    symplecticIntegrationKernel(acceleration, acceleration_ext,
        velocity, position, internal_dt);

  }

}
```

The first step of the algorithm is to calculate the center of masses of each of the solar systems in the galaxy. We then use this to compute the acceleration exerted on a body i by all the external bodies. The assumption we rely on in this integration method is that the external acceleration felt by a body will not change very much over a small period of time. For each timestep *dt*, we calculate the external acceleration once and use this to calculate the internal acceleration *steps* amount of times with a timestep of $\frac{dt}{steps}$ for some user defined value of *steps*. With this integration scheme, we recognize that the internal force needs to be calculated several times more often than the external forces and use this to approximate the external forces for a small period of time. In the Results section we will investigate how this integration methods performs over the other approaches.

3.8 Implementation Discussion

In this section we will discuss some of the design choices we did with regards to the implemented application.

3.8.1 Targeted Hardware Configuration

The current version of the program has been designed to be run in a hardware environment with one available GPU. As such, it does not currently support a dual-GPU configuration. While it may be possible to add such support, I believe there will not be much to gain by doing so at the moment.

When considering developing for a multi-GPU environment, it is important to pay careful attention to the origin and the flow of information. There are two use cases worth discussing where it might be useful to use multiple GPUs in the Celestial Simulator application. In the first use case, one GPU is delegated to doing computations related to the simulation, such as integration and running the N-Body algorithm while the other GPU performs the rendering. Another potential use case is to perform the N-Body algorithm on one GPU, while delegating the integration and rendering to the second GPU. This approach has the advantage of evenly distributing computations among GPUs. Since the N-Body is the most computationally expensive algorithm, having one GPU only focus on that, while the other takes care of the leftover work could potentially be an efficient setup. Despite this, the reason for not pursuing the multi-GPU approach is because of the significant overhead of transferring between GPUs. For every timestep, we would have to transfer acceleration, velocity and positional information between GPUs. After integration, we would also have to send back the updated velocity and posotional information, which leads to an expensive data transfer round-trip. As transfers are currently done using the PCI bus, this will be very expensive.

In 2016, NVIDIA is scheduled to launch their new line of GPU's utilizing their new Pascal microarchitecture[10]. Together with Pascal, NVIDIA will introduce their new NVLink technology[11], which can be used to send data from GPU to GPU much faster than current approaches. In the above use cases, there is potential for a form of pipelining parallelism, where each component continously receives data and sends it to the next stage in the pipeline. This scheme would allow us to do calculations at the same time as we are transferring data between GPUs. By utilizing the increased speed of data transfer using NVLink, such a multi-GPU approach could be interesting.

Another important design choice that I faced when designing the application was to decide whether to focus on single or double precision floating point operations. Disregarding the few high-end HPC-class GPU's, most NVIDIA GPU's currently have a small number of double precision floating point arithmetic units. As such, double precision computations can be very slow for many applications[16]. As real-time rendering is an important feature of the application, a decision was made to focus on single-precision floats, which has ample hardware support on NVIDIA's GPUs.

Chapter 4

Results and Discussion

In this chapter, we will present the result achieved during testing of the implemented application. As there are infinitely many possible test cases, we will focus on a select few to test the capabilities of the application as a whole. In the first few sections, we will briefly overview the test goals, test methodology and the hardware we will be running the simulation on.

4.1 Testing Overview

During testing, we will for the most part focus on measuring the performance and accuracy of each component of the application. As this can be difficult for simulation systems, we will be utilizing the analytical two-body solution whenever possible. When testing the the accuracy of the approximated Celestial N-Body algorithm, we will compare it to the All-Pairs N-Body CPU algorithm. In section 5.6, we will perform various performance tests to determine the performance of each of the algorithms. In these tests, we will test both the CPU and GPU version of the proposed Celestial N-Body algorithm, and compare the results to previous CUDA implementations, such as the All-Pairs N-Body GPU algorithm (GEM3), developed by NVIDIA[31].

4.1.1 Simulation Hardware Specification

The tests done in this chapter were performed on a machine running Ubuntu 14.04 and CUDA 7.0 with the following specifications:

1. **HPC-Lab24** 4x Intel Core i5-3470 CPU @ 3.20GHz with 16GB of Main Memory and a NVIDIA GeForce GTX 980 GPU

4.1.2 Performance Test Methodology

In the last section of this chapter, we will do performance comparison tests to measure the time taken to move the galaxy one timestep, for both the CPU and GPU-based algorithms. On the CPU, we will measure performance by using the gettimeofday(...) function to get the time in milliseconds.

```
double Utilities::get_wtime() {

    struct timeval t;
    gettimeofday(&t, NULL);
    double time_in_mill = (t.tv_sec) * 1000 + (t.tv_usec) /
        1000.0;
    return time_in_mill;

}
```

The GPU performance will be measured in milliseconds by utilizing available API calls from the CUDA framework. The code used to measure GPU performance is presented below:

```
void CudaTimer::cudaTimerStart() {

    cudaEventCreate(&timerStart);
    cudaEventCreate(&timerStop);

    cudaEventRecord(timerStart, 0);

}

float CudaTimer::cudaTimerStop() {

    float elapsedTime;

    cudaEventRecord(timerStop, 0);
    cudaEventSynchronize(timerStop);

    cudaEventElapsedTime(&elapsedTime, timerStart, timerStop);

    cudaEventDestroy(timerStart);
    cudaEventDestroy(timerStop);

    return elapsedTime;

}
```

Using these functions we are able to record a cudaEvent timestamp in our code, and use this to compare the timestamps before and after executing the specific code we want to measure. By calling the cudaEventSynchronize(cudaEvent) method, the CPU will block until the specified event has been recorded. The output measurement value has a resolution of about one half microsecond as detailed by NVIDIA[14].

4.2 Galaxy Generator Results

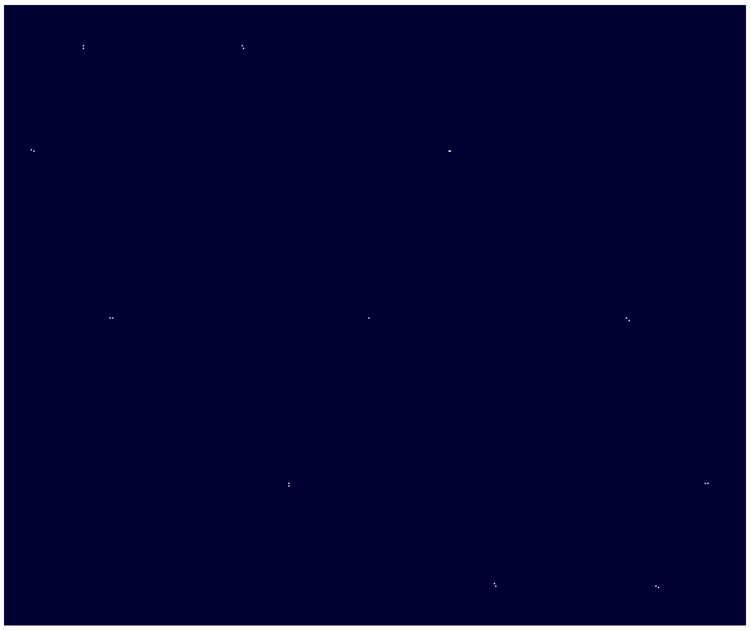


Figure 4.1: A circular galaxy with ten solar systems each with one star and one planet.

In this section, we will perform simulations tests on the galaxies created by the generator component. The current version of the galaxy generator is able to create simplified circular galaxies divided on one or two "planes", as detailed in the previous chapter. We opted to create small and simplistic galaxies with a few solar systems in order to be able to more easily observe how the galaxy is affected when changing different parameters such as the mass of the black hole.

We start off by determining a suitable timestep dt on the galactic scale. In our initial test, we will also confirm our assumption that the Symplectic Euler integration method is more suitable than the Explicit Euler integration method when doing simulation. We will run the simulation with a small circular galaxy where each of the solar systems are at the same distance $r = 100\ AU$ from the black hole. The mass of the black hole is ten times the mass of the stars in the solar systems. The galaxy will contain ten solar systems, each with one Earth-like planet orbiting a star with the same mass as the Sun. Figure 4.1 shows the test galaxy observed from an angle.

For each of the two integration methods, we will be testing different timesteps starting at $dt = 10\ days$, and incrementing by 10 $days$ for each test. As it can be difficult to analytically determine when the solar system no longer remain tightly coupled and become unstable, we will be empirically observing this by utilizing the Celestial Renderer. In order to ensure the error remains as low as possible, we will perform the test using the All-Pairs CPU N-Body algorithm.

During the simulation, the solar systems will orbit the black hole while each of the planets in the solar system will orbit it's star. As each of the stars start to pull on each other, they start to move closer and closer to the black hole. After reaching a point closer to the black hole, the solar systems start to move apart again. This pattern is repeated twice, before the solar systems gain too much momentum and escapes the black hole's orbit. This particular pattern can be observed because of the relative small mass of the black hole. As it is just ten times more massive than the stars, it is not able to keep the solar systems in orbit for a long period of time. This can be mitigated by increasing the mass of the black hole.

For all simulations performed in this test, the general movement of the stars remain unchanged. However, the movement of the orbiting planets greatly change between each test case. These results are presented in Table 4.1. We can see from the table that when using the Symplectic Euler integration method that timesteps up to 40 days is sufficient in order to keep both the star and its orbiting planet in a stable orbit around the black hole. At a timestep of 50 days, the timestep is simply too big and inaccurate, causing the planet to go out of orbit after 25 years. When using the Explicit Euler

Timestep (in days)	Time to instability (Explicit Euler)	Time to instability (Symplectic Euler)
dt = 10	~27 years	~330 years
dt = 20	~8 years	~330 years
dt = 30	~5 years	~330 years
dt = 40	0 years	~330 years
dt = 50	0 years	~30 years

Table 4.1: Integration Stability Results

integration method, none of the timesteps tested are small enough for the planets to remain in orbit. This is most likely because of the fact that the Explicit Euler method is not able to converse energy very well. While it is possible to mask this error by choosing a very small step size, we can see that it is much more viable to use the Symplectic Euler method instead.

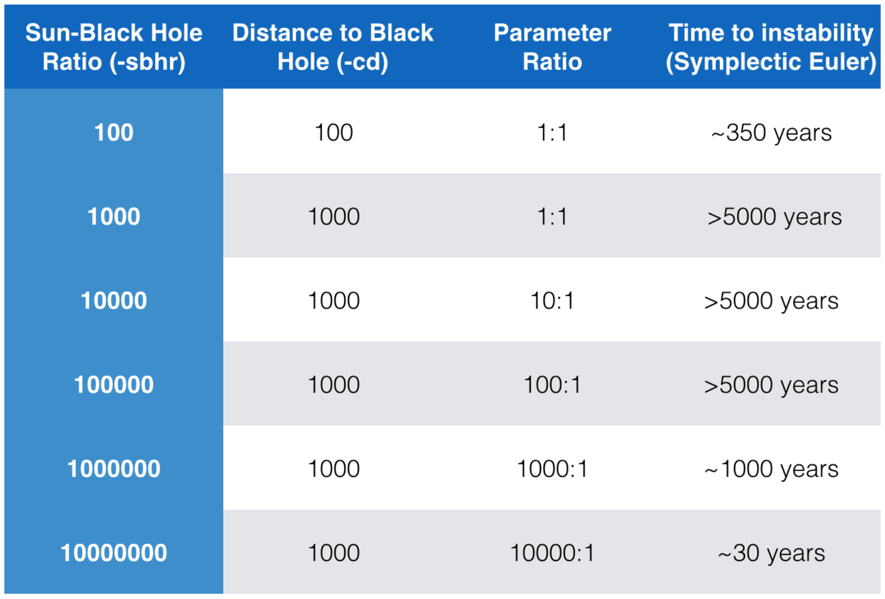
We can expand on this galaxy generator test by analyzing how changing different parameters changes the stability of the system. We set the timestep at 40 days, which from the previous result is a decent trade-off between performance and accuracy. We can generate different types of galaxies by changing parameters such as distance and mass. By changing these parameters, the stability of the system will also change. The previous test remained stable for a time, but after about 350 years, the combined force of the other solar systems caused the galaxy to become unstable. By increasing the mass of the black hole, the dynamics of the galaxy will change significantly.

Table 4.2 shows the result when changing the mass of the black hole. The parameter *sbhr* defines the ratio between the mass of the black hole and the mass of the stars in the galaxy (the default value is $sbhr = 10$). Finally, Table 4.3 shows additional results when increasing the mass when the distance is

Sun-Black Hole Ratio (-sbhr)	Time to instability (Symplectic Euler)
1	~190 years
10	~330 years
100	~330 years
1000	~780 years
10000	~12 years

Table 4.2: Black Hole Mass Stability Results

set to $r = 1000\ AU$. The results show various situations where the stability of the galaxy behaves differently. We can see that by increasing the distance and the mass of the black hole, the stability is maintained for a longer period of time. This is not surprising, as changing these parameters makes the black hole the dominant force in the galaxy. In the previous test case, the mass of the black hole was only 10 times bigger than a star. In addition, the distances between the solar systems were also much smaller. This causes each of the individual solar system to make a significant perturbation on the orbit of the other solar systems around the black hole, eventually making the system unstable. However, as we increase either of the two parameters above, the other forces in the system diminish in comparison to the black hole. This causes the system to be stable. While the results appear correct with regards to the inverse square law and Newton's Law of Universal Gravitation, they are also well-known. As such, we will be focusing our energy in utilizing the galaxy generator to test the capabilities of the simulator part of the application, starting with the integration.



Sun-Black Hole Ratio (-sbhr)	Distance to Black Hole (-cd)	Parameter Ratio	Time to instability (Symplectic Euler)
100	100	1:1	~350 years
1000	1000	1:1	>5000 years
10000	1000	10:1	>5000 years
100000	1000	100:1	>5000 years
1000000	1000	1000:1	~1000 years
10000000	1000	10000:1	~30 years

Table 4.3: Stability Results when changing black hole mass in relation to distance

4.3 Integration Results

In this section we will take a closer look at how the choice of integration step size and integration method affects the overall simulation. In the previous test, we determined that a timestep of 40 days was a reasonable trade-off because it was able to keep the entire galaxy stable for as long as physically possible. Lowering the timestep below 40 did not change how long the galaxy was stable, which suggests that the instability is because of a physical property of the galaxy and not that of an integration error. While the galaxy appear stable on a galactic level, we still need to confirm how well this timestep performs in terms of accuracy inside the invidual solar system. We will test this by once again running the Two Body Test component introduced in the implementation chapter. Since the analytical solution is not affected by the integration, we can compare the simulated version to this in order to get an idea how much error is introduced with a given timestep.

We run the Sun-Earth Two Body system for 50 orbits with a varying timestep of 1, 10, 20, 30 and 40. The results are compiled in Figure 4.2. The graphs shows the result of the comparison given the timestep with the lowest timestep 1 at the top and the largest timestep 40 at the bottom right. The first graph shows the results when running the comparison with a timestep of 1. This is the same test that we did in the implementation chapter. We can see that the difference is steadily increasing, however very slowly. As such, with a timestep of 1, the comparison remain very close for atleast 50 years(50 Earth orbits). As we increase the timestep, this changes significantly. At a timestep of 10, we see that the the error is as high as 2 AU after about 25 Orbits. This suggests that the simulated body is at the opposite side of the Earth orbit, which is a signficant error. After peaking at 25 orbits, the error keeps falling until it is almost negligible again. This test suggests that the simulated body moves much faster than the analytical body. At an interval of about 50 years, the simulated body catches up to the analytical body by an extra orbit, causing the error to sink as they get close, and rise again afterwards. This same pattern can also be observed for higher timesteps as the velocity difference between the two methods are increases about 4 times every 10 days increase in timestep. Additionally, if we take a look at the renderer when comparing for higher timesteps, we can see that the simulated body moves so fast that it does not manage to keep the same elliptic orbit as the analytical body. As we increase the timesteps, the errors in the elliptic path also significantly increases.

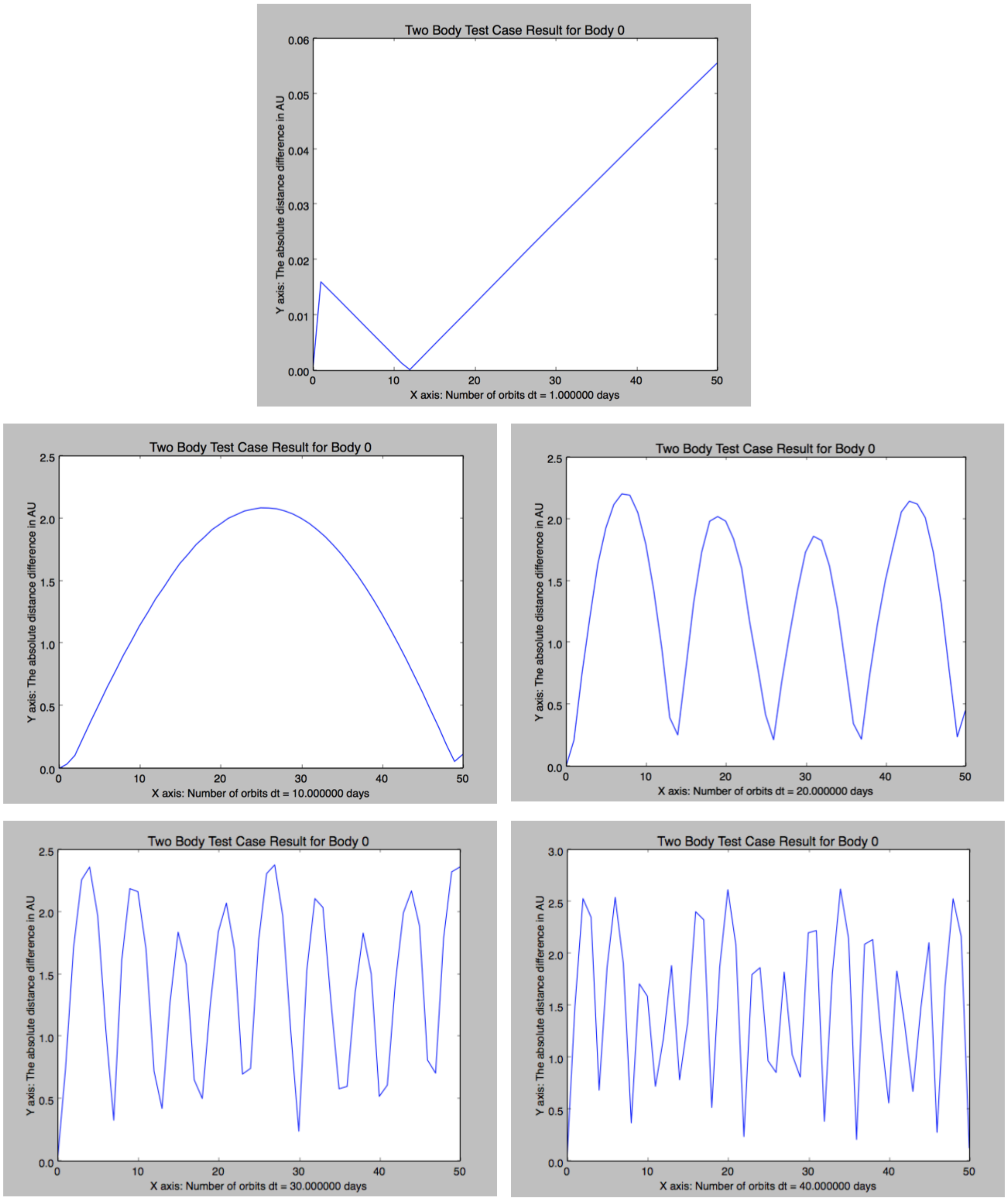


Figure 4.2: Planetary Stability Results in terms of integration stepsize

4.4 Celestial Integration Results

Timestep (-dt)	Integration Ratio (-cir)	Internal timestep	Time to instability
dt = 40	40	dt = 1	~330 years
dt = 80	40	dt = 2	~330 years
dt = 120	40	dt = 3	~330 years
dt = 160	40	dt = 4	~330 years

Table 4.4: Celestial Integration Results

It becomes clear that a timestep that is suitable on a galactic scale, can cause significant errors when utilized on solar systems. Inside each solar systems, bodies are much closer to each other, requiring much smaller timesteps to accurately simulate their force interactions. In order to fix this, we will utilize the Celestial Symplectic Integration method, which calculates distant forces less frequently than the forces that are closer. In order to achieve this, the integration method relies on using one of the implemented Celestial N-Body algorithms to calculate the external and internal forces exerted on each solar system.

In this next test, we run several simulations with the same galaxy composition as before. For each test case, we set a different time step and Celestial Integration Ratio, and observe the simulation. The results are summarized in Table 4.4. The tests were first run on the GPU and then compared to the equivalent Celestial N-Body CPU algorithm to ensure that they produce the same results. The comparison was done using the comparison component of the application, with the following terminal call

```
./CelestialSimulator -compare -ss 10 -sb 2 -set_nbody
    celestial_cpu celestial_gpu -set_integration
```

```
celestialSymplectic celestialSymplectic −dt dt −cir cir −
totalTime 120450 −log
```

with varying timestep *dt* and integration ratio *cir*. The comparison is run for 120450 days, which is almost 330 years.

We can see from the results that the Celestial Integration approach allows us to select a much higher integration stepsize as we can adjust the integration ratio such that the internal forces are integrated often enough to keep the solar system sufficiently accurate as well. As we have seen before, as long as the internal timestep of the solar system remains under 40 days, the system will remain stable. However, as this causes very inaccurate planetary orbits, it is much better to choose a timestep around 1 day. The Celestial Integration method allows us to do this without wasting a lot of computational power on recalculating external forces that does not change as often as the internal forces.

Using the Celestial Integration in conjunction with the Celestial N-Body algorithm, we are able to speedup the simulation by approximating the external forces for a number of steps determined by the integration ratio. When choosing the timestep and integration ratio we need to be cautious of two potential integration errors. The first error is caused by choosing a timestep that is high enough to cause significant integration errors even on a galactical scale. The second error is choosing an integration ratio that is too big. As the integration ratio increases, the reuse of the external acceleration causes the approximation to become less accurate. The upper threshold for both these values are highly dependent upon the galaxy composition. For any galactical composition, the first error can be determined experimentally by testing various timesteps and analyze the results. By assuming that lower timesteps are more accurate, we can determine the point where the error starts getting too high, based on how accurate we want the simulation to be. However, determining the second error is much more complicated. Since the Celestial Integration method is built upon the Celestial N-Body method, which is in itself an approximation, it is hard to analyze which errors are because of the N-Body approximation and which errors are because of the integration approximation. In order to better understand this, the two components needs to be decoupled such that they can be analyzed independently. This is out of the scope for this thesis, and we present it as future work in Chapter 5.

4.5 Celestial N-Body Accuracy Test

In this section, we will compare the implemented Celestial N-Body algorithm with the All-Pairs N-Body algorithm. We want to measure how good of an approximation the Celestial N-Body is. The test will be run on the same system as before. We assume that the Celestial N-Body algorithm will be able to give a good approximation as long as the galaxy remains stable. The approximation will not be sufficient after this point, as solar systems will start to collide with each other. We first compare the two N-Body algorithms using the Symplectic Euler Integration method. The comparison is done by running the two algorithms on two instances of the same galaxy. We run the simulation for 400 years, and for each timestep we compare the two galaxies by calculating and summing the absolute distance value for each of the corresponding bodies in the galaxies. This gives us the total absolute distance between the two galaxies for the 400 years time-span, which we graph using Python.

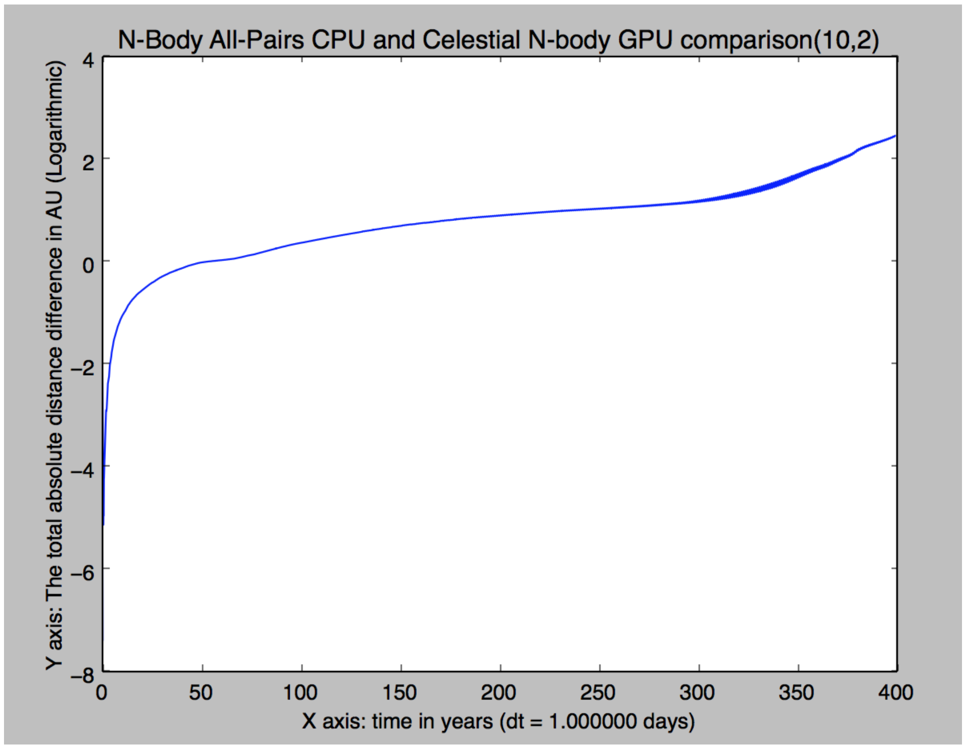


Figure 4.3: Accuracy Test 1 (distance = 100 AU)

The results of the comparison between the two N-body algorithms are a bit surprising. For the current test galaxy, we are simulating using real-life values for the solar systems. As the sun is around 300000 times more massive than the Earth, the force contribution from the single planet in the solar system should be negligible. As such, the center of mass for each solar system should be located very close to the star, and the Celestial N-Body algorithm should provide a good approximation. However, as we can see from Figure 4.3, The exponent of the total absolute difference is continously increasing before flatting out at around 1. We see another increase from around 330 years, however this is expected as beyond this point the system is no longer stable and the Celestial N-Body algorithm is no longer a viable approximation.

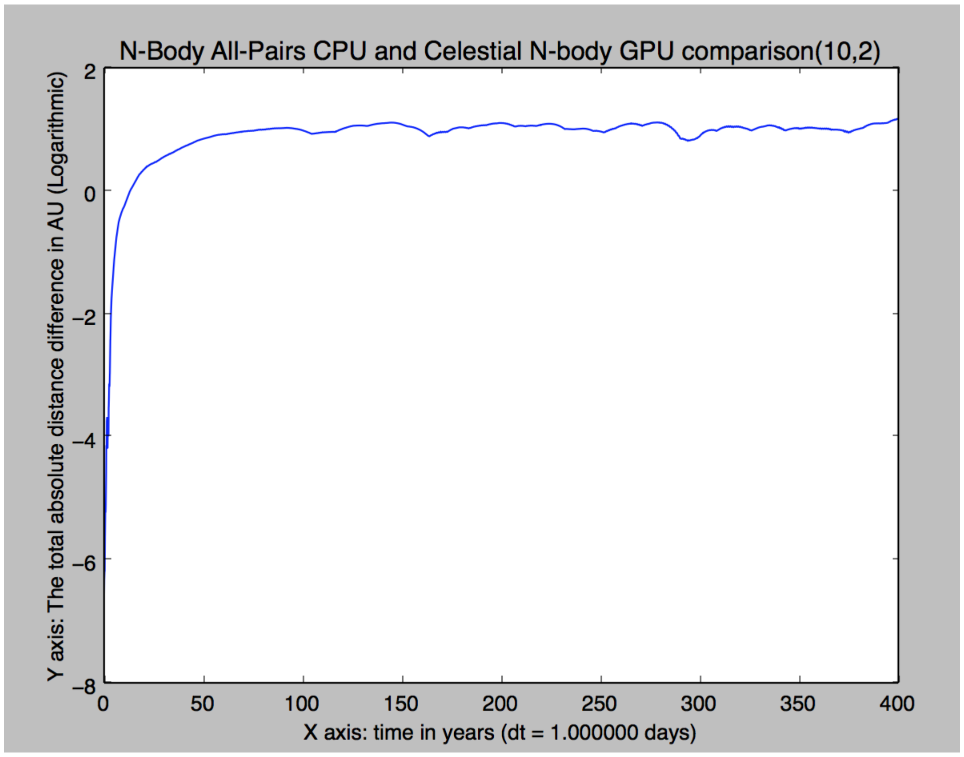


Figure 4.4: Accuracy Test 2 (distance = 1000 AU)

We investigate the source of the error further by increasing the distance from the black hole. Increasing this distance will increase the circumference of the circle the solar systems are located on, thus increasing the distance

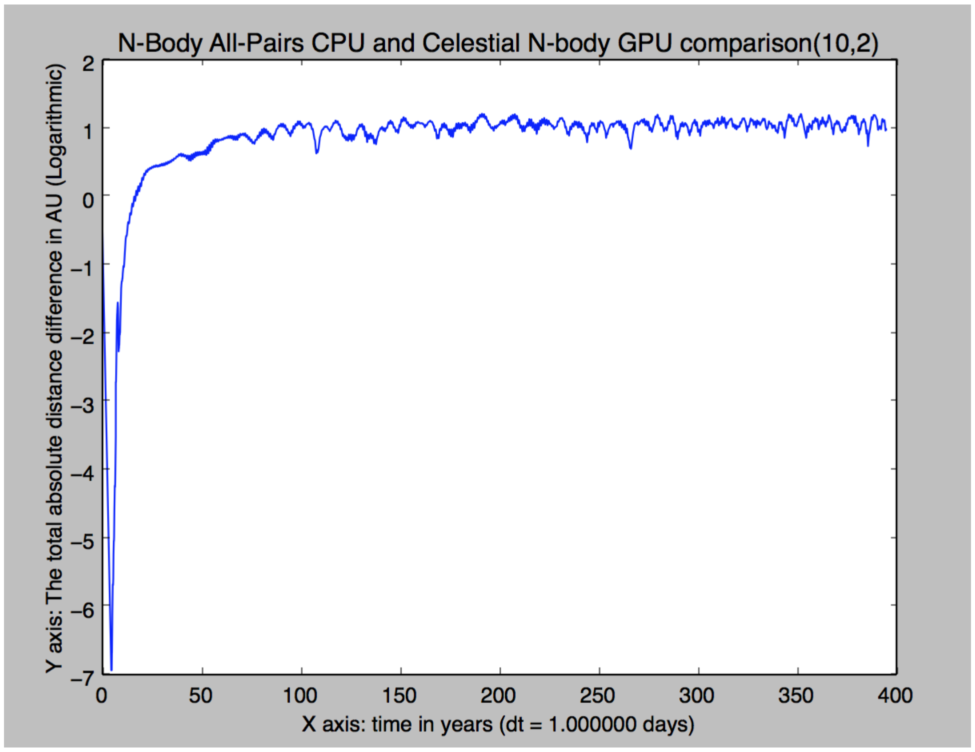


Figure 4.5: Accuracy Test 3 (distance = 10000 AU)

between the solar systems as well. This test will show if the solar systems were too close to each other in the previous tests. We perform two new tests with a new distance of 1000 AU and 10000 AU, respectively. The results can be seen in Figure 4.4 and 4.5. By looking at the figures, we can see that the problem still persists for larger distances. For both tests the exponent of the total absolute error flats out at around 1. These results suggest that the distance between the solar systems is not the source of the error.

In the next series of tests, we will increase the number of bodies in the galaxy, to see how this affects the error. In the first test, we will significantly increase the number of solar systems to 100, at a distance of 10000 AU. In the second test, we will increase the number of bodies within each solar system to 8 at the same distance of 10000 AU. The results are presented in Figure 4.6 and 4.7. We can see from the results that the error still flats out as the simulation goes on. However, as we have increased the number of bodies, the magnitude of the exponent has flattened out to a higher value of around 2,

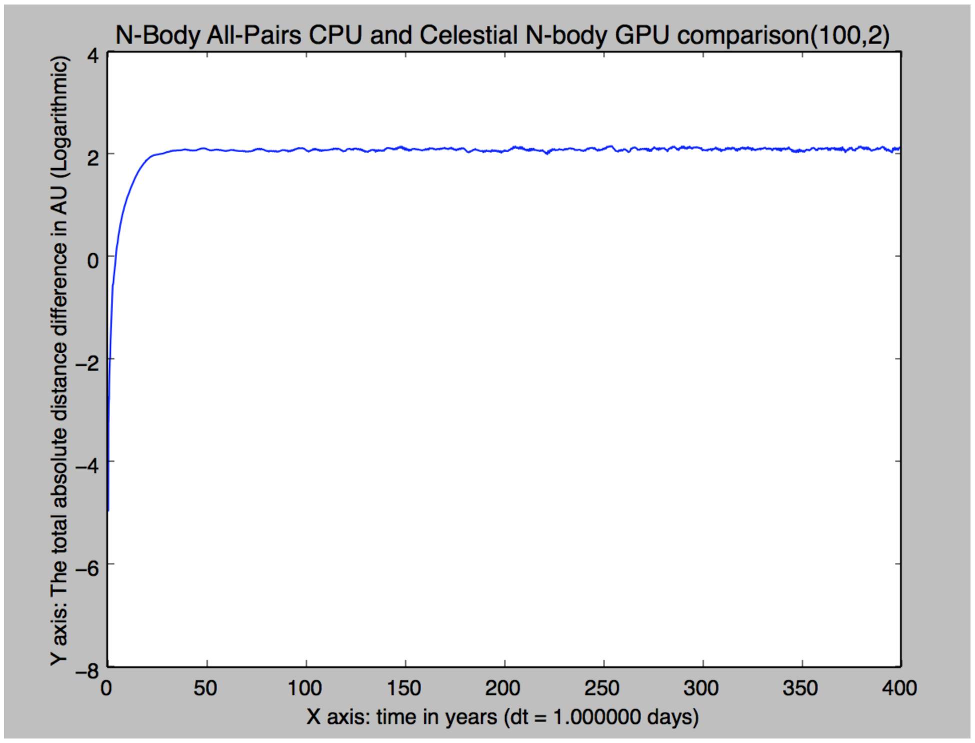


Figure 4.6: Accuracy Test 4 (distance = 10000 AU)

which is a significantly higher error than before. We can thus surmise that the error is related to the number of bodies within the galaxy. Considering this, the error is most likely caused by a numerical error when computing the center of mass. When approximating the external forces, the center of mass calculation causes tiny rounding errors which propagates to create larger errors. While the error increases both when increasing the number of solar systems and the number of solar bodies, the error increases more rapidly in the latter case. This is reasonable, as the number of computations when calculating each center of mass increases, thus increasing the error. When increasing the number of solar systems, the approximated center of mass is used more in order to approximate distant forces. Using this approximated value does also generate some error, although in smaller quantities. Increasing the number of solar systems by 10 times, increased the exponent from 1 to 2. However, increasing the number of solar bodies by four times, increases the error to around 2,2, which is a higher error even though there are fewer

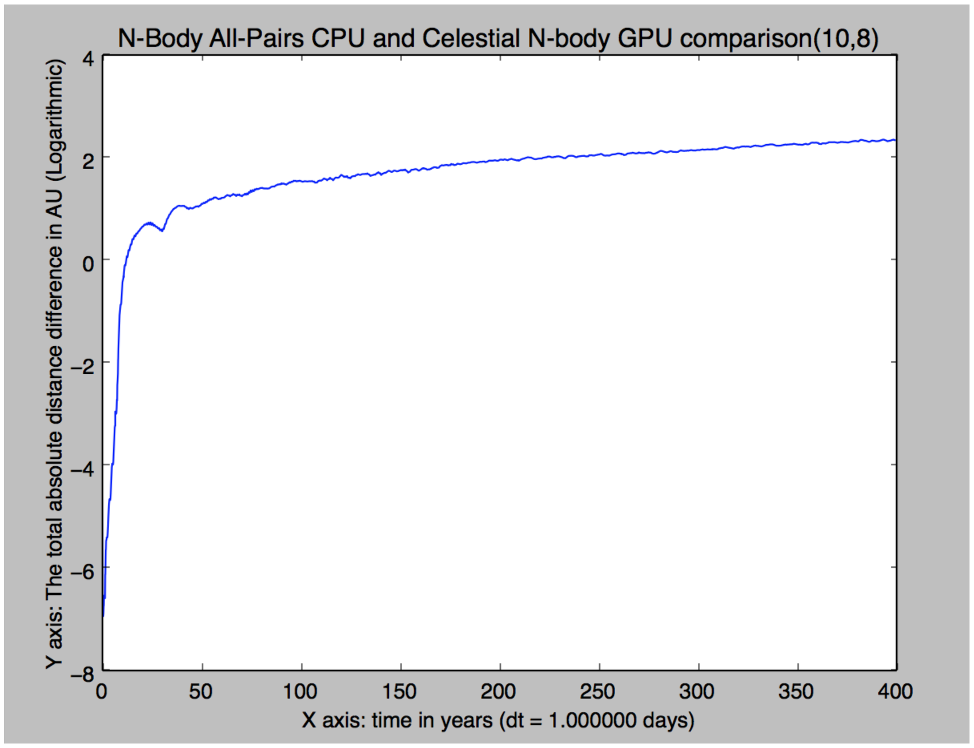


Figure 4.7: Accuracy Test 5 (distance = 10000 AU)

bodies overall in the galaxy.

In the last test, we set the solar systems to 100 and the solar bodies to 8, for a total of 801 bodies. The result is displayed in Figure 4.8. We can see that with this galaxy composition, the exponent of the total error is about 3,2. Overall, the error generated by using the center of mass as a means of approximation is surprisingly high. However, it is possible to explain it when considering that approximations with small errors are used in many further computations, which can cascade and end up with causing very large errors overall in the simulation.

4.5.1 Celestial N-Body Threshold Testing

In this section, we wish to test the threshold switching capabilities of the Celestial N-Body algorithm, as described in the implementation chapter. We run several tests with a galaxy consisting of ten solar systems with two

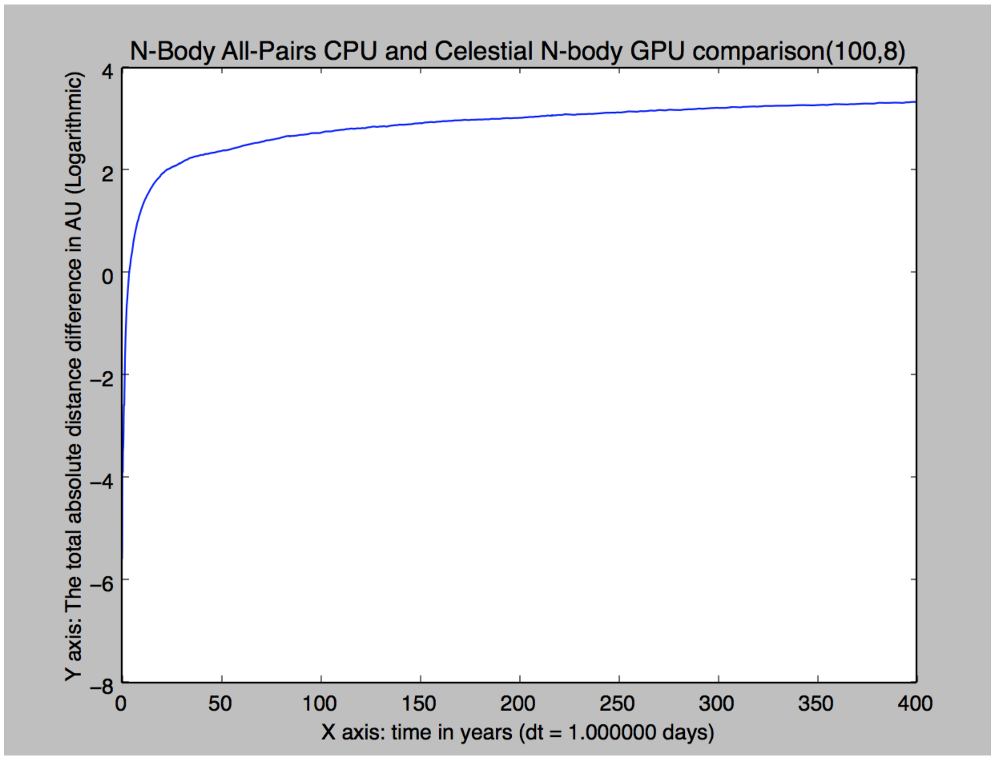


Figure 4.8: Accuracy Test 6 (distance = 10000 AU)

bodies each. Figure 4.9 presents one of these test cases, where the threshold is set at 50 AU. This threshold is reached after about 61 years. We compare Figure 4.9 with Figure 4.3, which is the same test run without the threshold.

When comparing the two figures, we can see that the results are very similar. While there are minute changes, the threshold switching does not reduce the error as was intended. The goal was for the threshold switching to prevent large-scale errors when systems start to collide with each other. However, since the switch occurs after several computations of the approximated algorithm, the state of the two comparison systems are slightly different. After switching, the same algorithm is used on both systems, but the different state of the systems will cause bodies to collide with each other slightly differently, which can propagate to large errors.

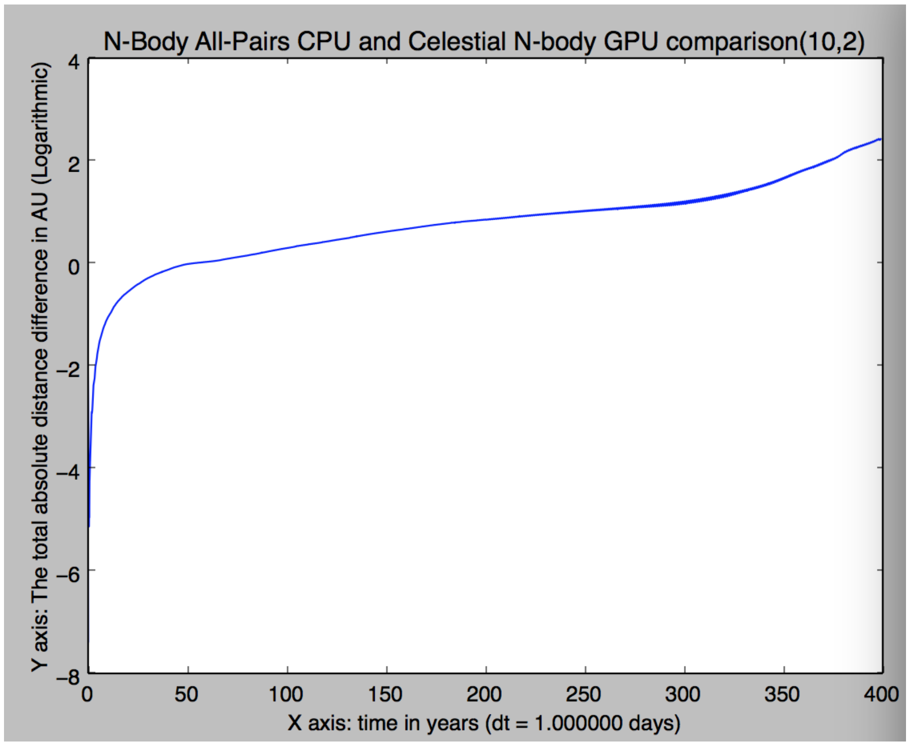


Figure 4.9: Threshold Test (Threshold = 50 AU, Distance = 100 AU)

4.6 N-Body Performance Comparison Test

In this last section, we will perform a series of performance tests to measure the performance between the implemented N-Body algorithms. In these tests, we will keep the number of bodies constant at 8, while increasing the number of solar systems between each test. The results are presented in Figure 4.10 and 4.11 and 4.12. In Figure 4.10, we have tested several small-scale galaxies with solar systems ranging from 100 to 500. In figure 4.11, we test larger scale galaxies with a solar system count ranging from 1000 to 5000. In the second test, we have excluded the All-Pairs algorithm, as it becomes too slow for these large cases. In Figure 4.12, we present how the execution time increases as the number of solar systems increases for the Celestial N-Body GPU algorithm.

We can see from the results in Figure 4.10, that the Celestial N-Body algorithms perform significantly better than the All-Pairs CPU approach, which is not surprising. Even for relative small numbers of solar systems,

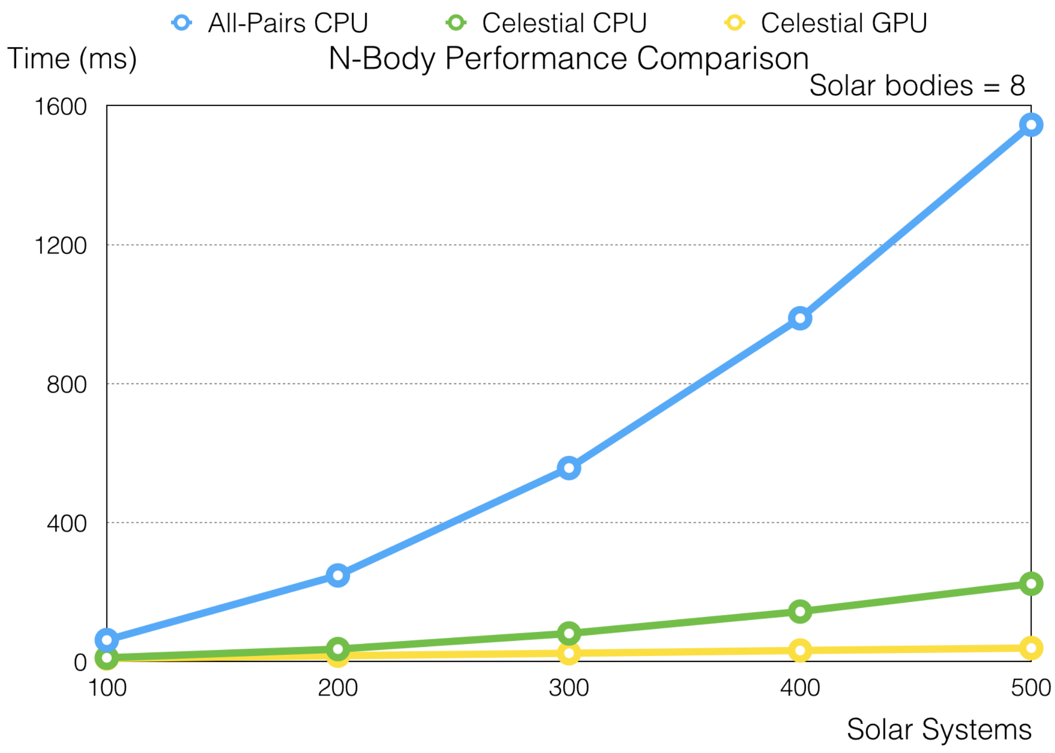


Figure 4.10: N-Body Performance Comparison

the Celestial N-Body algorithms are several times faster than the All-Pairs CPU algorithm. For smaller values, the Celestial N-Body CPU and GPU algorithms perform very similarly, however when approaching larger solar system numbers, the GPU-based algorithm is faster. At 5000 solar systems, the elapsed time of the GPU-based algorithm is just 6% the time compared to the CPU-based algorithm. Based on these results, we can see that the implemented algorithm gains significant performance benefits when utilizing GPU hardware. When comparing the Celestial N-Body GPU algorithm with the All-Pairs GPU(GEM3) algorithm (with a block size of 32) we see that it consistently performs better. By looking at Figure 4.11, we can see that the execution time is around 90% faster than the All-Pairs GPU N-Body algorithm when the number of solar systems is set to 5000 for a total of 40000 bodies.

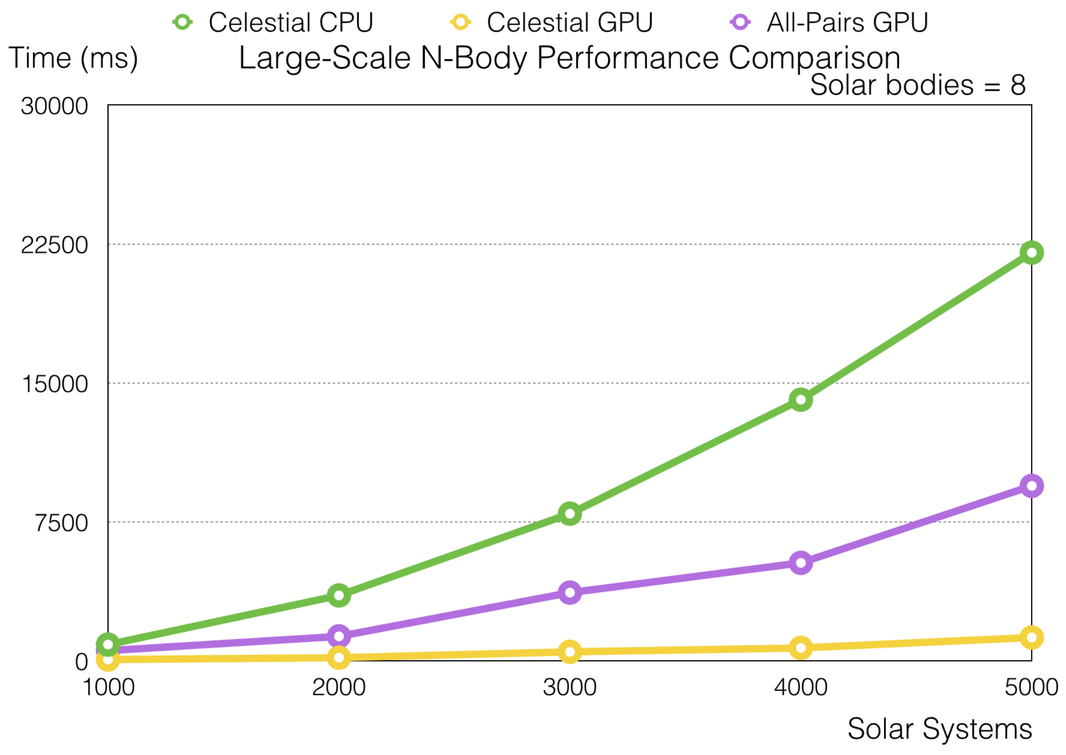


Figure 4.11: Large-scale N-Body Performance Comparison

4.7 Further Discussion

By looking at the results of the performance test we see that the the current version of the Celestial N-Body algorithm is very fast. The GPU is able to efficiently calculate the center of mass in parallel and apply that to the external solar systems. Using the center of mass as an approximation is very viable in terms of computational speed. However, by looking at the accuracy tests we see that the Celestial N-Body algorithms lose accuracy very fast as the number of bodies rises. As such, the current version of the algorithm is suitable when we want to do fast simulations that does not require pinpoint accuracy. If the goal of the simulation is purely accuracy, the Celestial N-Body simulation algorithm will not be sufficient. From the performance test, we can see that the Celestial N-Body algorithm is significantly faster than the All-Pairs GPU algorithm featured in NVIDIA's GPU Gems 3 book.

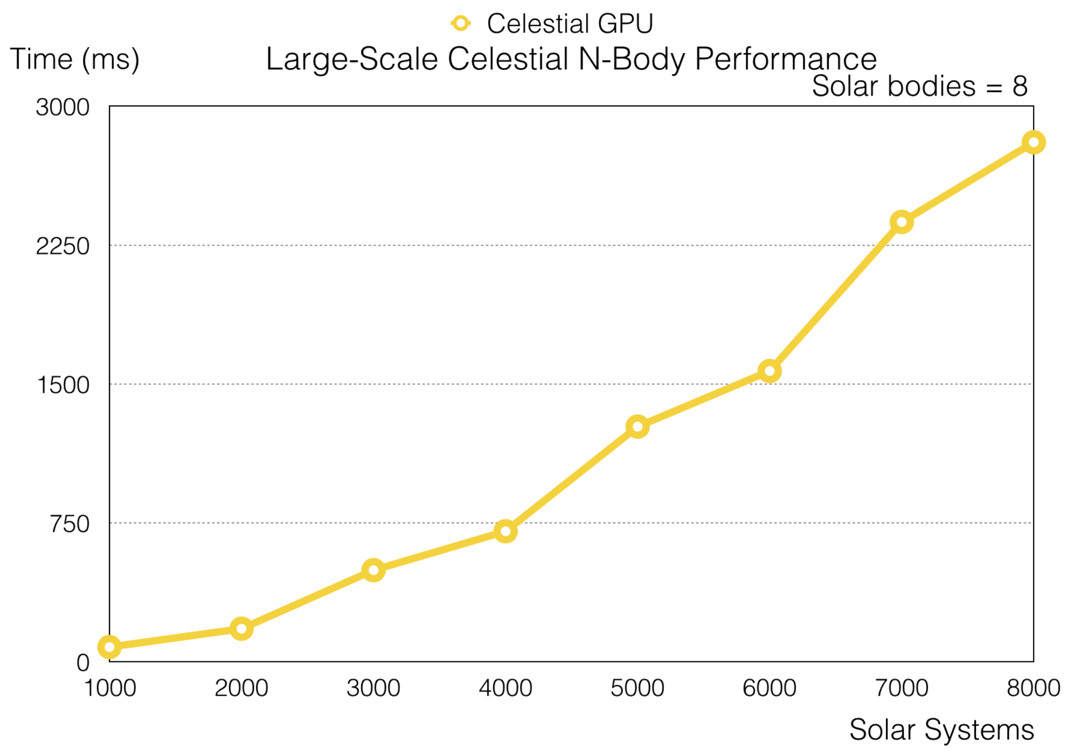


Figure 4.12: Large-scale Celestial N-Body GPU Performance

However, NVIDIA's approach is much more general than our algorithm, as the Celestial N-Body algorithm currently requires certain assumptions to be met in order to be useful.

Chapter 5

Conclusion and Future Work

In this thesis we have looked at a couple of topics relating to galaxy generation and simulation. In the original thesis draft, the plan was to focus more on the generator component of the implemented application. However, as work progressed it became clear that while such an application is able to fulfill the expected goals, the results obtained by running the simulator on the generated galaxies are already widely explored and well-known. Because of this, the simulation component of the application was expanded in order to implement a specialized N-Body algorithm and a specialized integration method.

The galaxy generator is currently able to generate simplified galaxies divided into one or two planes. Stability in the galaxy is achieved by letting the smaller bodies orbit the larger, more massive bodies. With the correct mass values and distance between the solar systems, the system will remain stable for quite some time. It is important for the primary bodies(such as the black hole and the stars), to be the dominant force for it's secondary orbiting bodies. If the mass ratio between the primary body and the orbiting body is too small, the galaxy created will have multiple bodies which can signicantly affect smaller bodies. This creates large perturbations which will affect the stability of the system aversely. It becomes clear that these factors are the most important to ensure the stability of our generated galaxies.

Tests performed on the simulator component of the application, show that the specialized Celestial N-Body algorithm runs much faster than the All-Pairs approach, both on the CPU and the GPU. In addition, the GPU implementation is able to perform significantly better than the CPU version, especially for a large number of bodies. When we ran a test with 5000 solar systems with 8 solar bodies each (for a total of 40000 bodies, excluding the black hole), the execution time was over 90% faster. This shows that the Celestial Algorithm gains much from GPU execution. Accuracy testing on the specialized N-Body algorithm shows that there are currently significant errors in the approximation, which is most likely caused by numerical errors when calculating the center of mass. This error grows as the number of bodies in the galaxy increases. The current version of the Celestial N-Body algorithm works well for doing fast simulations that does not require pin-point accuracy. However, it is not suited for simulation work that needs to guarantee the highest possible accuracy.

In any physical simulation, the integration of the acceleration into velocity and positional values is very important. In this thesis, we have explored a couple of integration methods for use with the simulator component. For instance, we were able to determine that the Explicit Euler integration method is not suitable for use when integrating planets orbiting a star in a solar system, as the energy of the system is not conserved very well. We have instead opted to use the Symplectic Euler integration method, which was able to integrate the values without any issues.

Adding to this, we have implemented and briefly tested a specialized integration method, which utilizes the Celestial N-Body algorithm in order to approximate the external forces for a period of time. This essentially allows us to calculate the internal forces with a lower timestep than the external forces. The initial results look promising, however more research must be done in order to determine the error when using this approximated integration method.

5.1 Future Work

In this section we will briefly talk about ideas for future work. The following ideas were things that came to mind during the development of the application that I believe could be interesting to look at.

- The Celestial Simulator is a tool capable of creating galaxies and simulating their movement in time. By expanding the type of galaxies the generator is able to create, there might be interesting things to learn from analyzing the simulation results.
- The current version of the Celestial N-Body algorithm shows promise

because of it's speed of computation. However, accuracy still leaves something to be desired. We have determined that the source of this inaccuracy most likely is in the computation of the center of mass. It would be interesting to look at other ways to approximate the external exertion besides the center of mass, and see if this improves the accuracy.

- As mentioned in Chapter 3.8, the current version of the Celestial Simulator is designed to be run on a single GPU, due to the significant amount of data that needs to be transfered in a multi-GPU environment. As NVIDIA's new NVLink is set to launch together with Pascal in 2016, this transfer protocol could make a multi-GPU approach much more viable.Implementing a multi-GPU approach using NVLink and comparing it to the current version would be an interesting case study.

- The communication capabilities between CUDA and OpenGL is crucial in order to ensure the efficiency of the application. It would be interesting to evaluate how the CUDA and OpenGL interoperability measure up to other approaches. In particular, OpenGL offers the OpenGL Compute Shader[25] that allows us to do general purpose computations using the glsl shader language. Not unlike CUDA, this allows us to do various types of general purpose computations on the GPU kernels. Using the OpenGL compute shader in conjunction with the OpenGL renderer might lead to performance improvements, as both features are part of the same OpenGL framework.

- In Chapter 4.4, we tested the Celestial Integration algorithm, and saw some promising results with regards to having dynamic integration timesteps. However, as this method is an approximation and is also based on the approximate Celestial N-Body algorithm, doing meaningful analysis of this method can be challenging. By finding a way to decouple the integration method from the N-Body algorithm, there might be a way to better analyse the error of this approximate integration method.

Bibliography

[1] Sverre J. Aarseth. Gravitational n-body simulations, tools and algorithms. Cambridge University Press, 2003, 2003.

[2] Kendall A. Atkinson. An introduction to numerical analysis, 2nd edition. New York, John Wiley and Sons, 1989.

[3] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, December 1986.

[4] Martin Burtscher and Keshav Pingali. Chapter 6 an efficient cuda implementation of the tree-based barnes hut n-body algorithm.

[5] Henry Cavendish. Experiments to determine the density of the earth. Philosophical Transactions of the Royal Society of London, Vol. 88 (1798), 469-526, 1798.

[6] John M. A. Danby. Fundamentals of celestial mechanics, 2nd revised and enlarged edition, September 1988.

[7] Travis Desell, Malik Magdon-ismail, Boleslaw Szymanski, Carlos A. Varela, Benjamin A. Willett, Matthew Arsenault, and Heidi Newberg. Evolving n-body simulations to determine the origin and structure of the milky way galaxy's halo using volunteer computing. 2011 IEEE International Parallel and Distributed Processing Symposium, 2011.

[8] Albert Einstein. Relativity: The special and general theory (translation 1920). H. Holt and Company, 1920.

[9] Mark Fowler. ATI Radeon HD5000 Series: An Inside View - Unleashing the Power of Parallel Compute! High Performance Graphics 2010 `http://www.highperformancegraphics.org/previous/www_`

2010/media/Hot3D/HPG2010_Hot3D_AMD.pdf, 2010. [Online; accessed 30-June-2015].

[10] Sumit Gupta. NVIDIA Updates GPU Roadmap; Announces Pascal. http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/, March 2014. [Online; accessed 05-January-2015].

[11] Sumit Gupta. What is NVLink? And How Will It Make the World's Fastest Computers Possible? http://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink/, November 2014. [Online; accessed 05-January-2015].

[12] J.C Hall. A multipropagator approach to real-time orbit simulation. Aerospace Conference Proceedings, 2002. IEEE (Volume:7), 2002.

[13] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[14] Mark Harris. How to Implement Performance Metrics in CUDA C/C++. NVIDIA Developer Zone http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/, November 2012. [Online; accessed 30-June-2015].

[15] Mark Harris. The Power of C++11 in CUDA 7. NVIDIA Developer Zone http://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/, March 2015. [Online; accessed 30-June-2015].

[16] Joakim Hommeland. A Qualitative Performance Comparison of Kepler and Maxwell GPUs through Benchmarking, January 2015. Pre-Master Project at the Norwegian University of Science and Technology. Advisor: Anne C. Elster.

[17] Johannes Kepler. Astronomia nova, 1609.

[18] Johannes Kepler. Harmonices mundi, 1619.

[19] NVIDIA Corporation. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. `www.nvidia.com/object/IO_37100.html`, November 2006. [Online; accessed 30-June-2015].

[20] NVIDIA Corporation. What Every CUDA Programmer Should Know About OpenGL. GPU Technology Conference `http://www.nvidia.com/content/gtc/documents/1055_gtc09.pdf`, October 2009. [Online; accessed 30-June-2015].

[21] NVIDIA Corporation. Optimizing CUDA - Part II. GPU Technology Conference `http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_Further_CUDA_Optimization.pdf`, 2011. [Online; accessed 30-June-2015].

[22] NVIDIA Corporation. NVIDIA GeForce GTX 980 Whitepaper. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`, 2014. [Online; accessed 05-January-2015].

[23] NVIDIA Corporation. OpenGL Interoperability. The CUDA Toolkit v7.0 Documentation `http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__OPENGL.html#group__CUDART__OPENGL`, March 2015. [Online; accessed 30-June-2015].

[24] SDL Community. Simple Directmedia Layer. SDL2 Website `https://www.libsdl.org/index.php`. [Online; accessed 30-June-2015].

[25] The Khronos Group. Compute Shader. OpenGL Website `https://www.opengl.org/wiki/Compute_Shader`. [Online; accessed 30-June-2015].

[26] The Khronos Group. OpenGL 3.3 Reference Pages. OpenGL Website `https://www.opengl.org/sdk/docs/man3/`. [Online; accessed 30-June-2015].

[27] The Khronos Group. OpenGL Overview. OpenGL Website `https://www.opengl.org/about/`. [Online; accessed 30-June-2015].

[28] C. D. Murray and Dermott S. F. Solar System Dynamics, 1999.

[29] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica*. The Royal Society, 1687.

[30] Edgar Josafat Martinez Noriega and Tetsu Narumi. High performance n-body simulation and visualization through cuda architecture. Bulletin of The University of Electro-Communications 24-1, pp. 59-64, 2012.

[31] M. Nyland, L. Harris and Prins J. Fast N-Body Simulation with CUDA. GPU Gems 3 Chapter 31, 2007.

[32] Dennis Overbye. Black hole hunters. The New York Times `http://www.nytimes.com/2015/06/09/science/black-hole-event-horizon-telescope.html`, June 2015. [Online; accessed 30-June-2015].

[33] Ashu Rege. An Introduction to Modern GPU Architecture. `ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`, 2012. [Online; accessed 30-June-2015].

[34] Mark J. Stock and Adrin Gharakhani. Toward efficient gpu-accelerated n-body simulation. 46th AIAA Aerospace Sciences Meeting and Exhibit, January 2008.

[35] N.M Swerdlow. Notes: Kepler's iterative solution to kepler's equation. Journal for the History of Astronomy, p.339, September 2000.

[36] Mikko Tuomi. Evidence for 9 planets in the hd 10180 system. Astronomy and Astrophysics Volume 543, July 2012.

[37] Nathan Whitehead and Alex Fit-Florea. Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs, 2011.

[38] Dr. David R. Williams. Planetary fact sheet - metric. `http://nssdc.gsfc.nasa.gov/planetary/factsheet/`, 2014. [Online; accessed 30-June-2015].

[39] Dr. David R. Williams. Planetary fact sheet. `http://nssdc.gsfc.nasa.gov/planetary/planetfact.html`, 2015. [Online; accessed 30-June-2015].

[40] Dr. David R. Williams. Planetary fact sheet - ratio to earth values. `http://nssdc.gsfc.nasa.gov/planetary/factsheet/planet_table_ratio.html`, 2015. [Online; accessed 30-June-2015].

Appendix A

Celestial Simulator Guides

This appendix details the installation and user guide needed to use the implemented application.

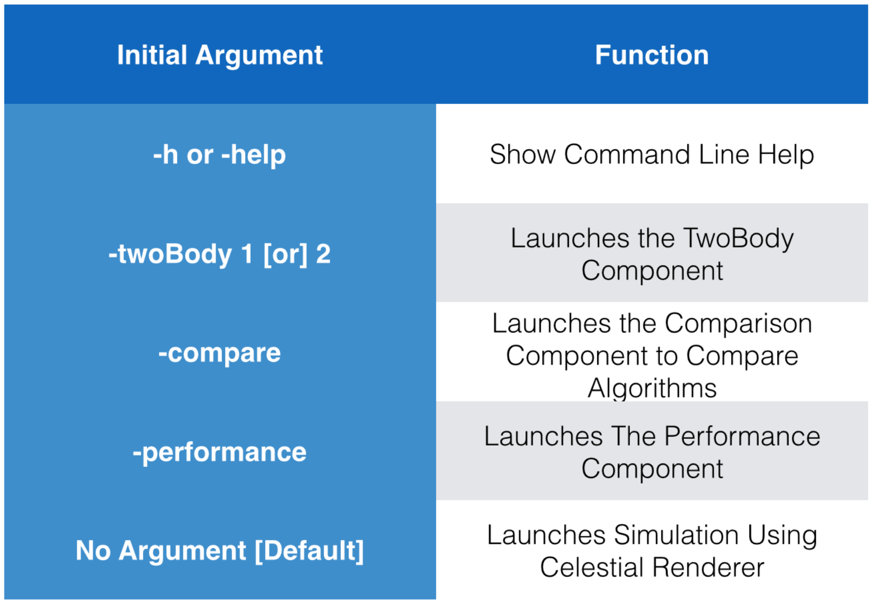
A.1 Installation Guide

The Following instructions were tested on an Ubuntu 14.04 64-bit machine.

1. Before compiling the application the following dependencies must be installed.
   (a) NVIDIA CUDA Toolkit 7.0 (Install from NVIDIA's CUDA download page)
   (b) OpenGL Mesa (Install the libgles2-mesa package using apt-get)
   (c) GLEW (Install the libglew-dev package using apt-get)
   (d) SDL2 (Install the libsdl2-dev package using apt-get)
   (e) In order to graph the results, Python 2.7 must be installed, along with the necessary libraries (Install python-dev, python-numpy, python-scipy and python-matplotlib using apt-get)
2. After all of the dependencies have been resolved, compile the application by running the make command in the install folder.
3. The application is terminal-based. After installation, run the application in the terminal with the desired arguments. The arguments the application takes as input can be confirmed in the next section.

4. In the case the renderer is used, commands for the renderer can be confirmed in Usage Guide.

A.2 Usage Guide



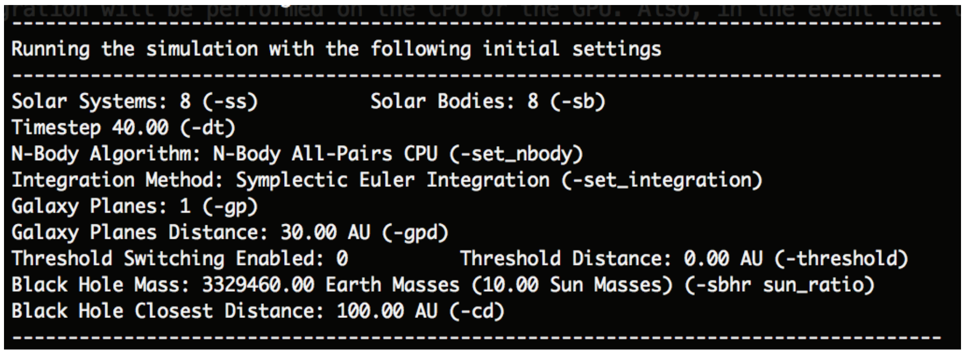
Initial Argument	Function
-h or -help	Show Command Line Help
-twoBody 1 [or] 2	Launches the TwoBody Component
-compare	Launches the Comparison Component to Compare Algorithms
-performance	Launches The Performance Component
No Argument [Default]	Launches Simulation Using Celestial Renderer

Table A.1: Initial Commands for the application

The application is launched from the terminal using run-time arguments based on the simulation/test the user wants to do. There are four main modes that can be executed, depending on the first argument when launching the application. In addition, to this depending on each mode, there are several optional arguments that can be specified to change the simulation. The available commands can be confirmed in the table below. Integration method and N-Body algorithm can be set using the ”Set Integration” and ”Set N-Body” arguments. For the default component using the simulation, these arguments takes in one additional parameter specifying the algorithm and the integration method. However, pelase note that for the comparison component, these arguments expect two parameters. This is used to compare the difference between various algorithm and integration method pairs. In the event that Celestial Integration is chosen as the integration method, as it exclusively utilize the Celestial N-Body algorithm, the "Set N-Body" argument will not be used. However the N-Body algorithm set will determine if the Celestial Integration will be performed on the CPU or the GPU. Also, in the event that threshold switching is enabled, the set N-Body algorithm will be used as the fallback algorithm. Most of the default values for the application can be confirmed in Figure A.1. The default values for the GPU block sizes have all been set to 32.

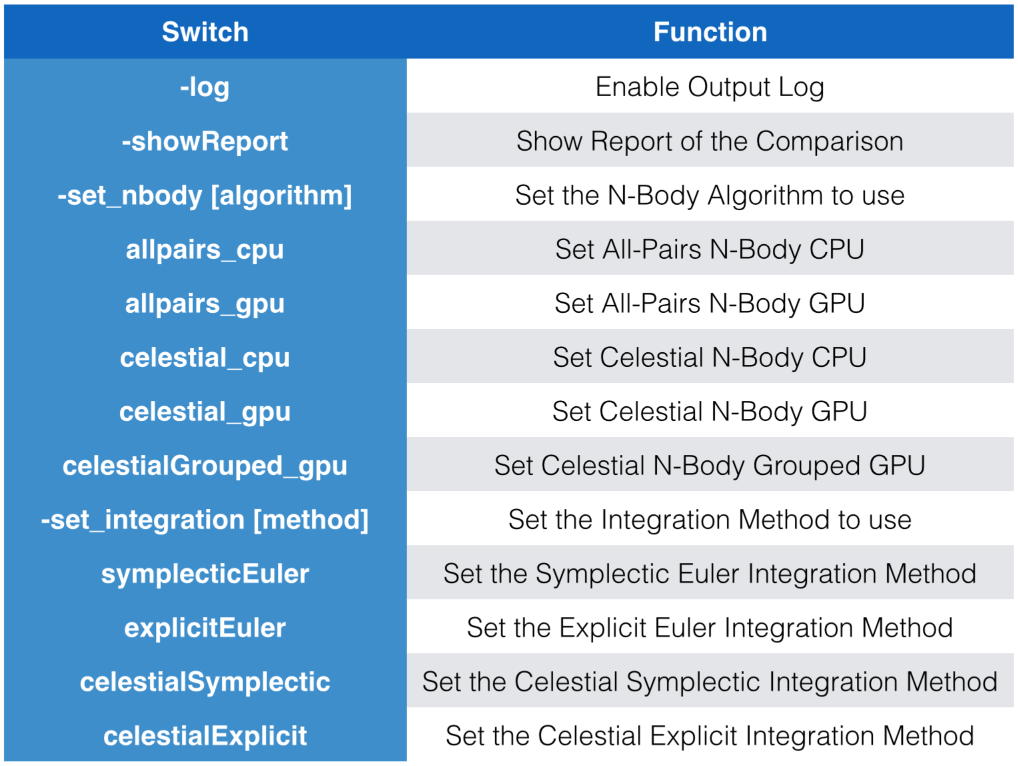
Switch	Function
-log	Enable Output Log
-showReport	Show Report of the Comparison
-set_nbody [algorithm]	Set the N-Body Algorithm to use
allpairs_cpu	Set All-Pairs N-Body CPU
allpairs_gpu	Set All-Pairs N-Body GPU
celestial_cpu	Set Celestial N-Body CPU
celestial_gpu	Set Celestial N-Body GPU
celestialGrouped_gpu	Set Celestial N-Body Grouped GPU
-set_integration [method]	Set the Integration Method to use
symplecticEuler	Set the Symplectic Euler Integration Method
explicitEuler	Set the Explicit Euler Integration Method
celestialSymplectic	Set the Celestial Symplectic Integration Method
celestialExplicit	Set the Celestial Explicit Integration Method

Table A.2: Run-time arguments flags for the Celestial Simulator Application



```
ration will be performed on the CPU or the GPU. Also, in the event that
--------------------------------------------------------------------------------
Running the simulation with the following initial settings
--------------------------------------------------------------------------------
Solar Systems: 8 (-ss)          Solar Bodies: 8 (-sb)
Timestep 40.00 (-dt)
N-Body Algorithm: N-Body All-Pairs CPU (-set_nbody)
Integration Method: Symplectic Euler Integration (-set_integration)
Galaxy Planes: 1 (-gp)
Galaxy Planes Distance: 30.00 AU (-gpd)
Threshold Switching Enabled: 0          Threshold Distance: 0.00 AU (-threshold)
Black Hole Mass: 3329460.00 Earth Masses (10.00 Sun Masses) (-sbhr sun_ratio)
Black Hole Closest Distance: 100.00 AU (-cd)
--------------------------------------------------------------------------------
```

Figure A.1: Default Values for the Celestial Simulator Application



Switch	Function
-log	Enable Output Log
-showReport	Show Report of the Comparison
-set_nbody [algorithm]	Set the N-Body Algorithm to use
allpairs_cpu	Set All-Pairs N-Body CPU
allpairs_gpu	Set All-Pairs N-Body GPU
celestial_cpu	Set Celestial N-Body CPU
celestial_gpu	Set Celestial N-Body GPU
celestialGrouped_gpu	Set Celestial N-Body Grouped GPU
-set_integration [method]	Set the Integration Method to use
symplecticEuler	Set the Symplectic Euler Integration Method
explicitEuler	Set the Explicit Euler Integration Method
celestialSymplectic	Set the Celestial Symplectic Integration Method
celestialExplicit	Set the Celestial Explicit Integration Method

Table A.3: Run-time flags for the Celestial Simulator Application

Switch	Usage	Function
-ss	-ss [Integer]	Number of Solar Systems
-sb	-sb [Integer]	Number of Solar Bodies
-ibs	-ibs [Integer]	Integration GPU Block Size
-cbs	-cbs [Integer]	Celestial N-Body Block Size
-abs	-abs [Integer]	All-Pairs GPU Block Size
-cir	-cir [Integer]	Celestial Integration Ratio
-cd	-cd [float]	Set Distance to Black Hole
-sbhr	-sbhr [float]	Set Star-Black Hole Mass Ratio
-dt	-dt [float]	Set Timestep in Days
-gp	-gp 1 [or] 2	Set Number of Galaxy Planes
-gpd	-gpd [float]	Set Distance between Planes
-threshold	-threshold [float]	Set Threshold For Celestial N-body

Table A.4: Run-time arguments for the Celestial Simulator Application

Keyboard Input	Function
Esc	Close the window and quit the application
Directional Key Up	Zoom In (Decrease FOV)
Directional Key Down	Zoom Out (Increase FOV)
Directional Key Left	Rotate Left
Directional Key Right	Rotate Right
Space	Start/Pause Simulation
R	Reset Camera
TAB	Invert Camera
F	Reset FOV
C	Turn Celestial Integration On/Off
V	Turn V-Sync On/Off
.	Print Current Timestep to Console
L	Turn Logging On/Off
P	Turn Performance Measure On/Off

Table A.5: Keyboard Controls for the Celestial Simulator Application

Appendix B

Planetary and Simulation values

This appendix shows an overview of the values used when generating galaxies and during simulation.

B.1 Determining the Gravitational Constant G

We want to determine the gravitational constant G based on the fact that our mass values are expressed in earth masses and distance is expressed using the astronomical unit (AU). We want the timestep to be expressed in days. We can thus convert the gravitational constant G found by Henry Canvendish to

$$8.883E^{-10} Days^{-2} M_{\oplus}^{-1} AU^{3} \tag{B.1}$$

B.2 Planetary Values

On the following pages are an overview of the different planetary values we have used when creating the orbits in this thesis. The following information has been compiled from the NASA planetary factsheet[39][38][40] and converted to the appropriate unit based on the Gravitational Constant derived in the previous section.

Planet	Eccentricity (ε)
Mercury	0,205
Venus	0,007
Earth	0,017
Mars	0,055
Jupiter	0,049
Saturn	0,057
Uranus	0,046
Neptune	0,011

Table B.1: Eccentricity values for the planets orbiting the Sun

Planet	Mass	Unit
Sun	332946	M⊕
Mercury	0,0553	M⊕
Venus	0,815	M⊕
Earth	1.0	M⊕
Mars	0,107	M⊕
Jupiter	317,8	M⊕
Saturn	95,2	M⊕
Uranus	14,5	M⊕
Neptune	17,1	M⊕

Table B.2: Mass values for the planets in our solar system.

Planet	Semi-Major axis a in AU
Mercury	0,387
Venus	0,723
Earth	1,0
Mars	1,524
Jupiter	5,204
Saturn	9,582
Uranus	19,201
Neptune	30,047

Table B.3: Semi-major axis values for the planets orbiting the Sun in Astronomical Units

Appendix C

Code Contributions

The following appendix include the important code contributions from this thesis. Please note that the code in this appendix has been design as part of a larger application. For the full code, please take a look at the source code accompanying this thesis.

C.1 Galaxy Generator

```
#include "GalaxyGenerator.h"
#include "math.h"
#include "../input/CelestialConstants.h"
#include <stdio.h>
#include "../utility/RandomNumberGenerator.h"
#include "../utility/CudaErrorCheck.h"
#include "../input/UserInput.h"

#define ToRadian(x) ((x) * M_PI / 180.0f)
#define ToDegree(x) ((x) * 180.0f / M_PI)

void GalaxyGenerator::createGalaxy(int solarSystems, int
    solarBodies) { //Currently only supports 2 planes (the angle
    is wrong)

  float blackHoleMass = UserInput::blackHoleMass;
  float starMass = CelestialConstants::sun_mass;
  float startAngle = ToRadian(45);

  //Set Galaxy Center properties
  velocities[0] = make_float3(0, 0, 0);
```

```
positions[0] = make_float3(0, 0, 0);
mass[0] = blackHoleMass;

float distance = UserInput::blackHoleClosestDistance;
float planes_distance = UserInput::galaxyPlanesDistance;

int planes = UserInput::galaxyPlanes;

int solarPerPlane = solarSystems / planes;
float inclination_d =  2*M_PI / solarPerPlane;
float azimuth_d = M_PI / planes;

int starIndex = 1;
for (int plane_i = 0; plane_i < planes; plane_i++) {

  float azimuth = plane_i * azimuth_d + startAngle;
  for (int solar_i = 0; solar_i < solarPerPlane; solar_i++) {

    float inclination = solar_i * inclination_d;

    //Set mass of the sun
    mass[starIndex] = starMass;

    //Step 1: Find the Star's start position in relation to
        the Black Hole
    positions[starIndex] = findSolarStartPosition(distance,
        inclination, azimuth);

    //Step 2: Find the Star's star velocity needed in order to
         orbit the Black Hole
    findStartVelocity(0, starIndex, 0, distance, inclination,
        azimuth);

    //Step 3: Find Solar System bodies start position and
        velocity in relation to star
    setSolarSystemValues(starIndex, starIndex + solarBodies,
        azimuth);

    starIndex += solarBodies;

  }

  distance += planes_distance;

}
```

```

}

void GalaxyGenerator::setSolarSystemValues(int startIndex, int
    endIndex, float azimuth) {

  float planetMass = CelestialConstants::earth_mass;
  float e = CelestialConstants::earth_e;
  float a = CelestialConstants::earth_a;

  for (int i = startIndex + 1; i < endIndex; i++) {

    //Set mass of the planet
    mass[i] = planetMass;

    float theta = RandomNumberGenerator::getRand(0, 2*M_PI);

    //Find Start Position
    findPlanetStartPosition(startIndex, i, e, a, theta, azimuth)
        ;

    //Find Start Velocity
    findStartVelocity(startIndex, i, e, a, theta, azimuth);

    a += CelestialConstants::earth_a;

  }

}

float3 GalaxyGenerator::findSolarStartPosition(float r, float
    inclination, float azimuth) {

  return make_float3(r * sin(inclination) * cos(azimuth), r *
      sin(inclination) * sin(azimuth), r * cos(inclination));

}

void GalaxyGenerator::findPlanetStartPosition(int primaryIndex,
    int secondaryIndex, float e, float a, float theta, float
    azimuth) {

  float3 primaryPosition = positions[primaryIndex];
  float3 blackholePosition = positions[0];

  float3 ss_vec = make_float3(blackholePosition.x -
      primaryPosition.x, blackholePosition.y - primaryPosition.y,
       blackholePosition.z - primaryPosition.z);

  //Find position
  float r = a * (1 - e * e) / (1 + e * cos(theta));
  float3 positionZY = make_float3(0, r * sin(theta), r * cos(
      theta));

  //Rotate to match the current plane
  float3 position;
  position.x = positionZY.x * cos(2 * M_PI - azimuth) -
      positionZY.y * sin(2*M_PI - azimuth);
  position.y = positionZY.x * sin(2 * M_PI - azimuth) +
      positionZY.y * cos(2*M_PI - azimuth);
  position.z = positionZY.z;

  position.x -= ss_vec.x;
  position.y -= ss_vec.y;
  position.z -= ss_vec.z;

  positions[secondaryIndex] = position;

}

void GalaxyGenerator::findStartVelocity2D(int primaryIndex, int
    secondaryIndex, float e, float a, float theta) {

  float primaryMass = mass[primaryIndex];
  float secondaryMass = mass[secondaryIndex];

  //Find phi
  float phi = G * (primaryMass + secondaryMass);

  //Find Orbital Period
  float T = sqrt((4 * M_PI * M_PI / phi) * a * a * a);

  //Find Mean Motion
  float n = 2 * M_PI / T;

  //Find start velocity
  float startX = (-1) * (n * a / sqrt(1 - e * e)) * sin(theta);
  float startY = (n * a / sqrt(1 - e * e)) * (e + cos(theta));



  float3 startVelocity = make_float3(startX, startY, 0);

  velocities[secondaryIndex] = make_float3(startVelocity.x +
      velocities[primaryIndex].x,
       startVelocity.y + velocities[primaryIndex].y,
       startVelocity.z + velocities[primaryIndex].z);

}

void GalaxyGenerator::findStartPosition2D(int primaryIndex, int
    secondaryIndex, float e, float a, float theta) {

  //Find position
  float r = a * (1 - e * e) / (1 + e * cos(theta));
  float3 startPos = make_float3(r * cos(theta), r * sin(theta),
      0);

  positions[secondaryIndex] = make_float3(startPos.x + positions
      [primaryIndex].x,
       startPos.y + positions[primaryIndex].y,
       startPos.z + positions[primaryIndex].z);

}

void GalaxyGenerator::findStartVelocity(int primaryIndex, int
    secondaryIndex,
     float e, float a, float theta, float azimuth) {

  float primaryMass = mass[primaryIndex];
  float secondaryMass = mass[secondaryIndex];

  //Find phi
  float phi = G * (primaryMass + secondaryMass);

  //Find Orbital Period
  float T = sqrt((4 * M_PI * M_PI / phi) * a * a * a);

  //Find Mean Motion
  float n = 2 * M_PI / T;

  //Find start velocity
  float startX = (-1) * (n * a / sqrt(1 - e * e)) * sin(theta);
  float startY = (n * a / sqrt(1 - e * e)) * (e + cos(theta));
  float3 velocityZY = make_float3(0, startY, startX);

  //Rotate it to match the plane
  float3 velocity;
  velocity.x = velocityZY.x * cos(2 * M_PI - azimuth) -
      velocityZY.y * sin(2 * M_PI - azimuth);
  velocity.y = velocityZY.x * sin(2 * M_PI - azimuth) +
      velocityZY.y * cos(2 * M_PI - azimuth);
  velocity.z = velocityZY.z;

  velocity.x += velocities[primaryIndex].x;
  velocity.y += velocities[primaryIndex].y;
  velocity.z += velocities[primaryIndex].z;

  velocities[secondaryIndex] = velocity;

}

GalaxyGenerator::GalaxyGenerator(int solarSystems, int
    solarBodies) {

  this->solarSystems = solarSystems;
  this->solarBodies = solarBodies;
  this->N = 1 + solarSystems * solarBodies;

  allocateVariables(N);

  createGalaxy(solarSystems, solarBodies);

}

void GalaxyGenerator::allocateVariables(int N) {

  positions = (float3*) calloc(N, sizeof(float3));
  accelerations = (float3*) calloc(N, sizeof(float3));
  velocities = (float3*) calloc(N, sizeof(float3));
  mass = (float*) calloc(N, sizeof(float));

  //Set acceleration to zero
  for (int i=0;i<N;i++) {
    accelerations[i] = make_float3(0,0,0);
  }

}

float3 *GalaxyGenerator::getPositionsGPU() {

  float3 *positions_d;

  CudaSafeCall(cudaMalloc(&positions_d, sizeof(float3) * N));

  CudaSafeCall(
      cudaMemcpy(positions_d, positions, sizeof(float3) * N,
          cudaMemcpyHostToDevice));

  free(positions);

  return positions_d;

}

float3 *GalaxyGenerator::getVelocitiesGPU() {

  float3 *velocities_d;

  CudaSafeCall(cudaMalloc(&velocities_d, sizeof(float3) * N));

  CudaSafeCall(
      cudaMemcpy(velocities_d, velocities, sizeof(float3) * N,
          cudaMemcpyHostToDevice));


  free(velocities);

  return velocities_d;

}

float3 *GalaxyGenerator::getAccelerationsGPU() {

  float3* accelerations_d;

  CudaSafeCall(cudaMalloc(&accelerations_d, sizeof(float3) * N))
      ;

  CudaSafeCall(
      cudaMemcpy(accelerations_d, accelerations,
          sizeof(float3) * N, cudaMemcpyHostToDevice));

  free(accelerations);

  return accelerations_d;

}

float *GalaxyGenerator::getMassesGPU() {

  float *masses_d;

  CudaSafeCall(cudaMalloc(&masses_d, sizeof(float) * N));

  CudaSafeCall(
      cudaMemcpy(masses_d, mass, sizeof(float) * N,
          cudaMemcpyHostToDevice));

  free(mass);

  return masses_d;

}

float3* GalaxyGenerator::getPositions() {

  return positions;

}

float3* GalaxyGenerator::getVelocities() {

  return velocities;

}

float3 *GalaxyGenerator::getAccelerations() {

  return accelerations;

}
float *GalaxyGenerator::getMasses() {

  return mass;

}

int GalaxyGenerator::getN() {

  return N;

}

int GalaxyGenerator::getSolarSystems() {

  return solarSystems;
}

int GalaxyGenerator::getSolarBodies() {

  return solarBodies;

}
```

C.2 Celestial N-Body

```
#include "CelestialNbody.h"
#include "NbodyInteractions.h"
#include "../utility/CudaErrorCheck.h"
#include "../input/UserInput.h"
#include <stdio.h>

//CPU
bool CelestialNbody::celestialNbodyCPU(float3 *acceleration,
    float3 *position, float *mass, int solarSystems, int
    solarBodies, bool enableThresholdSwitching) {

  if (UserInput::logSimulation) {
    printf("[Log] Simulation Algorithm: Celestial N-Body CPU
        Algorithm\n");
  }

  float4 *massCenters = calculateInternalAccelerationCPU(
      acceleration, position, mass, solarSystems, solarBodies);
  bool thresholdReached = calculateExternalAccelerationCPU(
      acceleration, position, mass, massCenters, solarSystems,
      solarBodies, true, UserInput::distanceThreshold);

  free(massCenters);

  return enableThresholdSwitching && thresholdReached;

}

float4 *CelestialNbody::calculateInternalAccelerationCPU(float3
    *acceleration, float3 *position, float *mass, int
    solarSystems, int solarBodies) {

  float4 *massCenters = (float4*) calloc(solarSystems, sizeof(
      float4));

  for (int solarIndex=0;solarIndex<solarSystems;solarIndex++) {

    int start = (solarIndex * solarBodies) + 1;
    int end = start + solarBodies;

    float totalMass = 0.0;
    float xSum = 0.0;
    float ySum = 0.0;
    float zSum = 0.0;

    for (int i=start;i<end;i++) {

      float3 i_acc = make_float3(0,0,0);

      for (int j=start;j<end;j++) {

        if (i == j) {
          continue;
        }

        i_acc = computeBodyBody(position[i], mass[i], position[j
            ], mass[j], i_acc);

      }

      acceleration[i] = i_acc;

      totalMass += mass[i];
      xSum += mass[i] * position[i].x;
      ySum += mass[i] * position[i].y;
      zSum += mass[i] * position[i].z;
    }

    //Calculate Center of mass
    massCenters[solarIndex] = make_float4(xSum / totalMass, ySum
        / totalMass, zSum / totalMass, totalMass);

  }

  return massCenters;

}

bool CelestialNbody::calculateExternalAccelerationCPU(float3 *
    acceleration, float3 *position, float *mass, float4 *
    massCenters,
    int solarSystems, int solarBodies, bool sumAcc, float
        distanceThreshold) {

  int N = 1 + solarBodies * solarSystems;

  bool thresholdReached = false;

  for (int i=1;i<N;i++) {

    int solarIndex = (i - 1) / solarBodies;

    float3 i_acc;
    if (sumAcc) {
      i_acc = acceleration[i];
    } else {
      i_acc = make_float3(0,0,0);
    }

    i_acc = computeBodyBody(position[i], mass[i], position[0],
        mass[0], i_acc);

    for (int j=0;j<solarSystems;j++) {

      float3 j_pos = make_float3(massCenters[j].x, massCenters[j
          ].y, massCenters[j].z);
      float j_mass = massCenters[j].w;

      if (j != solarIndex) {
        i_acc = computeBodyBody(position[i], mass[i], j_pos,
            j_mass, i_acc);

        if (distanceThreshold > 0) { //This can be optimized by
            putting this code in a separate loop.

          float3 i_com = make_float3(massCenters[solarIndex].x,
              massCenters[solarIndex].y, massCenters[solarIndex
                  ].z);

          float3 distanceVector = calculateDistanceVector(i_com,
              j_pos);
          float absoluteDistance = calculateAbsoluteDistance(
              distanceVector);

          if (absoluteDistance <= distanceThreshold) {
            thresholdReached = true;
          }

        }
      }

    }

```

```
        acceleration[i] = i_acc;

    }

    return thresholdReached;

}

bool CelestialNbody::launchCelestialNbodyKernel(float3 *
    acceleration_d, float3 *position_d, float *mass_d, int
    solarSystems, int solarBodies, bool enableThresholdSwitching)
    {

    if (UserInput::logSimulation) {
        printf("[Log] Simulation Algorithm: Celestial N-Body GPU
            Algorithm\n");
    }

    int N = 1 + solarBodies * solarSystems;

    int block_size = solarBodies;
    int num_blocks = solarSystems;
    int sharedMemSize = block_size * sizeof(float3) + block_size *
        sizeof(float);

    //Allocate memory in cuda kernel
    float4 *massCenters;
    CudaSafeCall( cudaMalloc(&massCenters, sizeof(float) * 4 *
        solarSystems) );

    bool *thresholdReachedGPU;
    CudaSafeCall( cudaMalloc(&thresholdReachedGPU, sizeof(bool)) )
        ;

    //Step 1: Calculate Internal Accelerations
    calculateInternalAccelerationGPU<<<num_blocks, block_size,
        sharedMemSize>>>(
        acceleration_d, position_d, mass_d, massCenters, N);
    CudaCheckError();

    int ext_blockSize = UserInput::allPairsBlockSize;
    int ext_numBlocks = (N-1) / ext_blockSize + (((N-1) %
        ext_blockSize == 0) ? 0 : 1);

    //Step 2: Calculate External Accelerations
```

```
  calculateExternalAccelerationGPU<<<ext_numBlocks,
      ext_blockSize>>>(acceleration_d, position_d, mass_d,
       massCenters, solarSystems, solarBodies, true, UserInput::
           distanceThreshold, thresholdReachedGPU);
  CudaCheckError();

  bool thresholdReached_output = false;
  if (enableThresholdSwitching) {
    bool *thresholdReached = (bool*) malloc(sizeof(bool));
    CudaSafeCall(
        cudaMemcpy(thresholdReached, thresholdReachedGPU, sizeof
            (bool),
            cudaMemcpyDeviceToHost));
    thresholdReached_output = *thresholdReached;
  }

  //Free Memory
  CudaSafeCall( cudaFree(massCenters) );
  CudaSafeCall( cudaFree(thresholdReachedGPU) );

  return thresholdReached_output;

}

//GPU Kernels

__global__ void calculateInternalAccelerationGPU(float3 *
    acceleration_d, float3 *position_d, float *mass_d, float4 *
    output_com, int N) {

  //Find the index of this planet in the array
  int planetIndex = blockIdx.x * blockDim.x + threadIdx.x + 1;
  int solarGlobalIndex = blockIdx.x;
  int solarBodies = blockDim.x;

  //Define shared memory
  extern __shared__ float sharedMemory[];

  //Divide into float3 position and float mass
  float *massShared = sharedMemory;
  float3 *positionShared = (float3*) &massShared[solarBodies];

  float i_mass;
  float3 i_pos;

  if (planetIndex < N) {

    //Load position values for body i
    i_pos = position_d[planetIndex];

    //Load mass value for body i
    i_mass = mass_d[planetIndex];

    //Load data into the block's shared memory
    positionShared[threadIdx.x] = position_d[planetIndex];
    massShared[threadIdx.x] = mass_d[planetIndex];

  }

  __syncthreads();

  float3 i_acc = make_float3(0,0,0);

  if (planetIndex < N) {

    float3 j_pos;
    float j_mass;

    float totalMass = 0.0;
    float xSum = 0.0;
    float ySum = 0.0;
    float zSum = 0.0;

    for (int j = 0; j < solarBodies; j++) {

      j_pos = positionShared[j];
      j_mass = massShared[j];

      i_acc = computeBodyBody(i_pos, i_mass, j_pos, j_mass,
          i_acc);

      if (output_com) {
        totalMass += j_mass;
        xSum += j_mass * j_pos.x;
        ySum += j_mass * j_pos.y;
        zSum += j_mass * j_pos.z;
      }

    }

    if (threadIdx.x == 0 && output_com) {
      output_com[solarGlobalIndex] = make_float4(xSum /
          totalMass, ySum / totalMass, zSum / totalMass,
           totalMass);
    }

  }

  __syncthreads();

  if (planetIndex < N) {
    acceleration_d[planetIndex] = i_acc;
  }


}

__global__ void calculateExternalAccelerationGPU(float3 *
    acceleration_d, float3 *position_d, float *mass_d, float4 *
    massCenters,
     int solarSystems, int solarBodies, bool sumAcc, float
         distanceThreshold, bool *thresholdReached) {

  int globalIndex = blockIdx.x * blockDim.x + threadIdx.x + 1;
  int N = 1 + solarSystems * solarBodies;
  int solarGlobalIndex = (globalIndex - 1) / solarBodies;

  float3 i_acc; //returned value

  if (globalIndex < N) {

    float3 i_pos = position_d[globalIndex];
    float i_mass = mass_d[globalIndex];

    if (sumAcc) {
      i_acc = acceleration_d[globalIndex];
    } else {
      i_acc = make_float3(0,0,0);
    }

    //The acceleration from the black hole
    i_acc = computeBodyBody(i_pos, i_mass, position_d[0], mass_d
        [0], i_acc);

    //The acceleration from each distant solar system
    float3 j_pos;
    float j_mass;


    if (distanceThreshold > 0 && globalIndex == 1) {
      *thresholdReached = false;
    }

    for (int j = 0; j < solarSystems; j++) {

      if (j != solarGlobalIndex) {
        j_pos = make_float3(massCenters[j].x, massCenters[j].y,
            massCenters[j].z);
        j_mass = massCenters[j].w;

        i_acc = computeBodyBody(i_pos, i_mass, j_pos, j_mass,
            i_acc);

        if (distanceThreshold > 0 && globalIndex == 1) {

          float3 i_com = make_float3(massCenters[
              solarGlobalIndex].x,
               massCenters[solarGlobalIndex].y, massCenters[
                  solarGlobalIndex].z);

          float3 distanceVector = calculateDistanceVector(i_com,
               j_pos);
          float absoluteDistance = calculateAbsoluteDistance(
              distanceVector);

          if (absoluteDistance <= distanceThreshold) {
            *thresholdReached = true;
          }

        }

      }

    }

  }

  __syncthreads();

  if (globalIndex < N) {
      acceleration_d[globalIndex] = i_acc;
   }

}
```

C.3 Celestial N-Body Grouped

```
#include "CelestialNbodyGrouped.h"
#include "NbodyInteractions.h"
#include "../utility/CudaErrorCheck.h"
#include "../input/UserInput.h"

//Kernel Launcher
bool CelestialNbodyGrouped::launchCelestialNbodyKernel(float3 *
    acceleration_d, float3 *position_d, float *mass_d, int
    solarSystems, int solarBodies, bool enableThresholdSwitching)
    {

  if (UserInput::logSimulation) {
    printf("[Log] Simulation Algorithm: Celestial N-Body GPU
        Grouped Algorithm\n");
  }

  int N = 1 + solarBodies * solarSystems;

  int block_size = UserInput::celestialNbodyBlockSize;
  int num_blocks = (N - 1) / block_size + (((N - 1) % block_size
      == 0) ? 0 : 1);
  int systems_per_block = block_size / solarBodies;
  int sharedMemSize = systems_per_block * solarBodies * sizeof(
      float3) + solarBodies * systems_per_block * sizeof(float);

  //Allocate memory in cuda kernel
  float4 *massCenters;
  CudaSafeCall( cudaMalloc(&massCenters, sizeof(float) * 4 *
      solarSystems) );

  bool *thresholdReachedGPU;
  CudaSafeCall( cudaMalloc(&thresholdReachedGPU, sizeof(bool)) )
      ;

  //Step 1: Calculate Internal Accelerations
  calculateInternalAccelerationGroupedGPU<<<num_blocks,
      block_size, sharedMemSize>>>(
       acceleration_d, position_d, mass_d, massCenters,
           solarSystems, solarBodies, systems_per_block);
  CudaCheckError();

  int ext_blockSize = UserInput::allPairsBlockSize;
  int ext_numBlocks = (N-1) / ext_blockSize + (((N-1) %
      ext_blockSize == 0) ? 0 : 1);

  //Step 2: Calculate External Accelerations
  calculateExternalAccelerationGPU<<<ext_numBlocks,
      ext_blockSize>>>(acceleration_d, position_d,
       mass_d, massCenters, solarSystems, solarBodies, true,
           UserInput::distanceThreshold, thresholdReachedGPU);
  CudaCheckError();

  bool thresholdReached_output = false;
  if (enableThresholdSwitching) {
    bool *thresholdReached = (bool*) malloc(sizeof(bool));
    CudaSafeCall(
        cudaMemcpy(thresholdReached, thresholdReachedGPU, sizeof
            (bool),
            cudaMemcpyDeviceToHost));
    thresholdReached_output = *thresholdReached;
  }

  //Free Memory
  CudaSafeCall( cudaFree(massCenters) );
  CudaSafeCall( cudaFree(thresholdReachedGPU) );

  return thresholdReached_output;

}

__global__ void calculateInternalAccelerationGroupedGPU(float3 *
    acceleration_d, float3 *position_d, float *mass_d, float4 *
    output_com, int solarSystems, int solarBodies, int
    SystemsPerBlock) {

  //Total Size of array
  int N = 1 + solarSystems * solarBodies;

  //Group size
  int groupSize = SystemsPerBlock * solarBodies;

  //Find Indices
  int globalIndex = blockIdx.x * blockDim.x + threadIdx.x + 1;
      //Global Index in array from 1 to N-1
  int localIndex = threadIdx.x; //Index within group of solar
      systems from 0 to systemsPerBlock * solarBodies - 1
```

```
int solarGlobalIndex = (globalIndex - 1) / solarBodies; //
    Index within solar systems from 0 to solarSystems-1
int solarLocalIndex = localIndex / solarBodies; //Solar system
     index within current block

//Define shared memory
extern __shared__ float sharedMemory[];

//Divide into float3 position and float mass
float *massShared = sharedMemory;
float3 *positionShared = (float3*) &massShared[groupSize];

float i_mass;
float3 i_pos;

if (localIndex < groupSize && globalIndex < N) {

  //Load position values for body i
  i_pos = position_d[globalIndex];

  //Load mass value for body i
  i_mass = mass_d[globalIndex];

  //Load data into the block's shared memory
  positionShared[threadIdx.x] = position_d[globalIndex];
  massShared[threadIdx.x] = mass_d[globalIndex];

}

__syncthreads();

float3 i_acc = make_float3(0,0,0);

if (localIndex < groupSize && globalIndex < N) {

  float3 j_pos;
  float j_mass;

  float totalMass = 0.0;
  float xSum = 0.0;
  float ySum = 0.0;
  float zSum = 0.0;


```

```
    int j_start = solarLocalIndex * solarBodies; //Shared Memory
        Index
    int j_end = j_start + solarBodies;

    for (int j = j_start; j < j_end; j++) {

      j_pos = positionShared[j];
      j_mass = massShared[j];

      i_acc = computeBodyBody(i_pos, i_mass, j_pos, j_mass,
          i_acc);

      //Calculate components for Center of mass
      if (output_com) {
        totalMass += j_mass;
        xSum += j_mass * j_pos.x;
        ySum += j_mass * j_pos.y;
        zSum += j_mass * j_pos.z;
      }

    }

    if (threadIdx.x % solarBodies == 0 && output_com) {
      output_com[solarGlobalIndex] = make_float4(xSum /
          totalMass, ySum / totalMass, zSum / totalMass,
           totalMass);
    }

  }

  __syncthreads();

  if (localIndex < groupSize && globalIndex < N) {
    acceleration_d[globalIndex] = i_acc;
  }

}
```

C.4 Celestial Integration

```
bool SymplecticEuler::performCelestialIntegrationCPU(
    CelestialSystem* system, float dt) {

  if (UserInput::logSimulation) {
    printf("[Log] Celestial Integration CPU will be performed!
        Ratio: %f\n", UserInput::celestialIntegrationRatio);
  }

  float3 *velocity = system->getVelocities();
  float3 *acceleration = system->getAccelerations();
  float3 *position = system->getPositions();
  float *mass = system->getMasses();
  int N = system->getN();
  int solarSystems = system->getSolarSystems();
  int solarBodies = system->getSolarBodies();

  float internal_steps = UserInput::celestialIntegrationRatio;
  float external_dt = dt;
  float internal_dt = external_dt / internal_steps;

  float4 *massCenters = calculateMassCenterForSolarSystemsCPU(
      position, mass, solarSystems, solarBodies);
  float3 *acceleration_ext = (float3*) calloc(N, sizeof(float3))
      ;
  bool thresholdReached = CelestialNbody::
      calculateExternalAccelerationCPU(acceleration_ext, position
      , mass, massCenters,
       solarSystems, solarBodies, false, UserInput::
           distanceThreshold);
  free(massCenters);

  for (int i=0;i<(int)internal_steps;i++) {

    CelestialNbody::calculateInternalAccelerationCPU(
        acceleration, position, mass, solarSystems, solarBodies);
    SymplecticEuler::performSymplecticIntegrationCPU(system,
        acceleration_ext, internal_dt);

  }

  free(acceleration_ext);

  return thresholdReached;
```

```

}


bool SymplecticEuler::performCelestialIntegrationGPU(
    CelestialSystem* system, float dt) {

  if (UserInput::logSimulation) {
    printf("[Log] Celestial Integration GPU will be performed!
        Ratio: %f\n", UserInput::celestialIntegrationRatio);
  }

  float3 *velocity = system->getVelocities();
  float3 *acceleration = system->getAccelerations();
  float3 *position = system->getPositions();
  float *mass = system->getMasses();
  int N = system->getN();
  int solarSystems = system->getSolarSystems();
  int solarBodies = system->getSolarBodies();

  int ibs = UserInput::integrationBlockSize;
  int inb = N / ibs + ((N % ibs == 0) ? 0 : 1);

  int cbs = solarBodies;
  int cnb = solarSystems;
  int sharedMemSize = ibs * sizeof(float3) + ibs * sizeof(float)
      ;

  float internal_steps = UserInput::celestialIntegrationRatio;
  float external_dt = dt;
  float internal_dt = external_dt / internal_steps;

  float4* massCenters;
  CudaSafeCall( cudaMalloc(&massCenters, sizeof(float) * 4 *
      solarSystems) );
  calculateMassCenterForSolarSystems<<<solarSystems, solarBodies
      >>>(position, mass, massCenters);
  CudaCheckError();

  bool *thresholdReachedGPU;
  CudaSafeCall( cudaMalloc(&thresholdReachedGPU, sizeof(bool)) )
      ;
  CudaCheckError();

  //Save acceleration for external systems
  float3* acceleration_ext;
  CudaSafeCall( cudaMalloc(&acceleration_ext, sizeof(float) * 3
      * N) );
  calculateExternalAccelerationGPU<<<inb, ibs>>>(
      acceleration_ext, position, mass, massCenters,
       solarSystems, solarBodies, false, UserInput::
           distanceThreshold, thresholdReachedGPU);

  //Perform Internal calculation internal_steps-1 times
  for (int i=0;i<(int)internal_steps;i++) {

    calculateInternalAccelerationGPU<<<cnb, cbs, sharedMemSize
        >>>(acceleration, position, mass,  0, N);
    CudaCheckError();

    symplecticIntegrationKernel<<<inb, ibs>>>(acceleration,
        acceleration_ext, velocity, position, N, internal_dt);
    CudaCheckError();

  }

  bool thresholdReached_output = system->thresholdIsReached();
  if (system->thresholdSwitchingIsEnabled()) {
    bool *thresholdReached = (bool*) malloc(sizeof(bool));
    CudaSafeCall(
        cudaMemcpy(thresholdReached, thresholdReachedGPU, sizeof
            (bool),
             cudaMemcpyDeviceToHost));
    thresholdReached_output = *thresholdReached;
  }

  CudaSafeCall( cudaFree(massCenters) );
  CudaSafeCall( cudaFree(thresholdReachedGPU) );
  CudaSafeCall( cudaFree(acceleration_ext) );

  return thresholdReached_output;

}
```