Developing a 2D n-Body Engine

C. Van Nipper

CSC 499 – Dr. Andy Digh

Mercer University

**Table of Contents**

[Abstract 3](#_Toc184813275)

[Introduction 4](#_Toc184813276)

[Conceptual Foundations 5](#_Toc184813277)

[Development Process 8](#_Toc184813278)

[Results 12](#_Toc184813279)

[Future Directions 13](#_Toc184813280)

[Conclusion 14](#_Toc184813281)

[References 15](#_Toc184813282)

# Abstract

This paper describes the process used to design and develop an engine which simulates the motion of n bodies due to gravitation. The engine was developed from scratch using the Pygame library in Python. This project was motivated by my passion for physics and space exploration, and a desire to use fundamental equations I learned in school, such as the basic kinematic equations and Newton’s second law of motion, to simulate one of the real-world phenomena they apply to. The engine utilizes these equations to calculate the gravitational force each body exerts on each other body and update the velocities and positions accordingly. Because this algorithm requires a nested for-loop, it requires a runtime of . Through runtime analysis, it was determined that this means the maximum value of n before the program is unable to run at a constant 60 frames per second is 16. Future directions for this project might include utilizing more efficient algorithms for calculating forces or expanding the program to calculate physics in three dimensions. Overall, this project represents an intersection between physics and computer science and will serve as a foundation for further exploration in developing simulation engines.

# Introduction

## Background and Motivation

n-Body engines are crucial for the simulation of space flight. Research organizations utilize them to predict astronomical events including the position of planets at a given time, eclipses, etc. Historically, they have been especially useful for planning and testing the paths taken by spacecraft during interplanetary missions. I have been especially interested in developing an n-body engine because of my interest in space, as well as my love for programming. In many of my physics and engineering classes, I have learned equations of motion which can both be applied to real-world scenarios like the motion of planets and used in computer simulations.

## Project Scope and Objectives

I aimed to utilize both my knowledge of programming and the fundamental principles of physics, including Newton’s Second Law of Motion, to develop my own 2D n-body simulation engine. My goal for this project was to develop an n-body engine which could accurately simulate and predict the motion of n-bodies (for example, the earth, moon and sun where n = 3). The engine was be developed using Python libraries and tools like Pygame. The engine did not need to use the most computationally efficient algorithms for calculating the forces that were applied to the objects. However, it should have been able to run at 60 frames per second with a reasonable number of bodies, or in other words, it is able to complete all necessary calculations in seconds. The specific objectives for the functionality of this program were as follows:

* User-Customization: The program provides variety of options that allow the user to customize various parameters of bodies in the simulation including size, mass, color, initial position and velocity, etc.
* Time warp: Since gravity is the weakest of the four fundamental forces, the forces objects exhibit on one another are relatively weak, and motion can seem slow. For this reason, I implemented a feature to the program that enables the user to speed up time.
* Coordinate System: The program includes a clear coordinate system where one unit represents one kilometer. There is also a grid system with a key that shows the scale of distance which the user is observing.
* Body Tracking: So that the user can observe a body as it moves around the coordinate system, a feature was implemented to center the camera’s view about the center of whatever body the user clicks on.

# Conceptual Foundations

## Physics Properties of the n-Body Problem

The n-body problem is one that pioneers of physics began scrutinizing over centuries ago (Portillo, 2024). Many physicists worked on researching and developing methods to calculate the position and velocity of celestial objects at given times. Isaac Newton was one such physicist. One of the many significant scientific observations Newton made in his life was that the velocity of any object will remain constant unless a net external force is acted upon it. This means that if the velocity of an object is changing, there must be some force that is acting upon it, whether the object is as small as a ball rolling on the ground, or as large as a planet orbiting around the sun. Newton observed that planets move in an elliptical orbit around the sun, meaning that there must be some forces acting on it. Newton called this force gravity. He believed gravity it to be the very same force which accelerates humans towards the center of the earth. Today, we know by Einstein’s theory of general relativity that gravity is not a force, but a distortion of spacetime caused by immense amounts of concentrated mass (Tillman et. al, 2022). However, Newtonian gravity has shown to be exceptionally accurate for calculating the trajectories of planets in our solar system.

To calculate the effects of gravitational force on each body in the simulation, we utilized Newton’s second law of motion, which states that the acceleration of an object is equal to the net force applied to that object divided by its mass. This is commonly denoted by the equation . Newton observes that the strength of the gravitational force an object exerts on another object is inversely proportional to the square of the distance between the two objects () and is also multiplied by a constant (), which we will call the gravitational constant. This gives us the updated equation . Solving this new equation for acceleration using the original equation, we find . This equation is the basis for calculating the desired change in velocity of each body in the simulation.

To update the position and velocity of the bodies in the simulation given acceleration, initial velocities, and initial positions, we used the standard kinematic equations, which tell us how position, velocity and acceleration are related to each other with respect to time. The change in velocity () is given as , and the change in position () is given as . These two equations are the basis for updating velocity and position, respectively for each body in the simulation.

Computational Challenges

One significant issue with simulating n-body physics is runtime. Calculating the gravitational force that each body exerts on other bodies in the simulation requires a complexity of , which increases quite rapidly once n reaches high values. To combat against this, algorithms like the Barnes-Hut algorithm are often used. The Barnes-Hut algorithm uses rules to simplify groups of distant particles to a point mass and calculates the net gravitational force of that point mass. This reduces the complexity of the n-body problem from to (Portillo, 2024). We do not use the Barnes-Hut algorithm in this study, but applying it to this engine may be a good future direction.

Another computational challenge arises from the nature of the kinematic equations use to update position and velocity. In computer science, calculating integrals, or infinitely small slices of differential time to get the exact area under a curve, is only possible using special rules like the power and chain rules and depends on the function of acceleration with respect to time being given. This is not possible for the n-body problem, since the functions for the velocities and positions of all the bodies are differential equations which cannot be solved. Because of this, our kinematic equations needed to be discretized as follows and calculated each frame:

Note that the update in velocity and position is divided by the number of frames per second. This is because the unit for and are and and need to be converted to and , respectively. This discretization allows for the trajectories of each body to easily be calculated at the cost of minor inaccuracies.

# Development Process

Planning and Tools

To plan the creation of this engine, the process was broken into the following steps:

1. Coordinate System
2. Ease-of-Use Features (perspective dragging, focusing on planets, etc.)
3. Program Loop (includes calculating forces, velocities, positions, etc.)

Note that the use of this study’s foundational formulas is not considered until step 3. This is because the engine was designed from scratch, and it was important to create an interface that a user can traverse in a way that feels natural first, so that the debugging was easier once step 3 was reached.

Implementation

The first step in creating a functional simulation engine was to design a working coordinate system, as well as a way for the user to easily navigate through the coordinate system. Here, we define a few constants and variables. , or zoom is the scale at which the screen is displayed. Since the simulation is on such a large scale, we set the highest possible zoom as the value 1, where each pixel on the screen represents one square kilometer of space. This means if the zoom is 0.1, each pixel on the screen represents 100 square kilometers of space, and so on. We also define the constants and , the resolution of the user’s screen in pixels. This information is retrieved automatically using the Python *screeninfo* package. Finally, we will define anchor variables and and . Here it is very important to distinguish between screen pixels which range (0 –, 0 – ) and simulation pixels/units which can range infinitely. The variables and are the simulation coordinates at the pixel at the center of the screen (,). From now on, we will call the units within the simulation “anchor space” or “anchor pixels”, as opposed to “screen pixels” or “screen space”, the range of pixels on the screen. Because the pixels of the screen begin at (0, 0) on the top left corner and end at (, ) on the bottom right corner, we needed a way to easily transform from screen space to anchor space and vice versa for the rest of the study. We will call the transform from screen pixels and to anchor pixels the *Anchor Transform* and the transform from anchor pixels and to screen pixels the *Inverse Anchor Transform* . They are defined as follows:

Now that the user can easily traverse through a clear 2D coordinate system, a grid system is added to the program to give the user a clear perspective of their location in the coordinate system, as well as the ability to observe objects moving. To implement the grid, lines are drawn at a varying interval of pixels within the boundaries of the user’s screen. The location the lines are drawn at is dependent on anchor space, so it is clear when the user translates their perspective along the coordinate grid. After the grid system is added, a few quality-of-life features like dragging to translate the perspective, a bounding box for the simulation, hotkeys to allow for easier traversal and centering, an in-game clock, and more were added. Next, the data structure for the bodies was implemented. The bodies are represented by a Python class called “Planet”, as shown by the figure below:

A black screen with white text

Description automatically generated

Figure 1: Python class implementation of a body

A Planet object contains the following data:

* A string representing the body’s name.
* A float representing the body’s mass
* An integer representing the body’s diameter.
* Two integers representing the initial x and y coordinates of the body.
* Two integers representing the initial x and y components of the velocity of the body.
* A tuple of three integers representing the desired RGB values of the body once it is drawn on screen.

## Program Loop

Since all necessary components of the simulation have been initialized, working on the program loop is next. Up until this point, we had been laying the groundwork for the program loop and creating the variables and functions that the program uses. The program loop contains all the tasks which should be completed each frame. The following are each of the tasks in the program loop:

1. Event handling: Each possible input is checked to see if the user has interacted with the simulation in any way. If they have, the proper action is taken. For example, if the user drags their mouse to the left, a value proportional to the distance the user dragged is added to the x component of the anchor coordinate.
2. Update of In-Game Clock: The values of the clock in game which shows the current date and time are updated. Naturally, more time is added to the clock if the warp speed is higher than .
3. Screen Logic, Grid, and UI: The screen is initialized, the gridlines are drawn in the proper locations, and all UI elements like the in-game clock, the warp slider, and the coordinate display are drawn on screen.
4. Motion Update: A nested for loop runs through each of the bodies and calculates the force each other body exerts on it. This is added to the body’s velocity. Each body’s position is then updated after all changes in velocities have been calculated.
5. Game Tick Update: At the end of the loop, the program is told to halt for ~ s. To ensure the program runs at exactly 60 frames per second, the following equation is applied to determine exactly how long the program should halt:

Where is the amount of time the program is told to halt at the end of the program loop and is the total amount of time it takes the program loop to run once. If is negative, a value of zero will be used instead.

# Results

Because this study is an independent learning project, there are not many results to observe other than the working product itself. However, I thought it would be interesting to examine two things: The performance as the value of n increases, and the maximum value of n that the engine can still run at 60 FPS. To measure performance, we used the variable , the time it takes the program loop to run once. The lower the value of is, the better the performance. The last value of n that gives , or , is the maximum value of n we are looking for. Testing the program with increasing values of n gave the following data:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 1 | 5 | 10 | 15 | 16 | 17 | 20 | 25 | 30 |
| (s) | 0.0092 | 0.0096 | 0.0128 | 0.0138 | 0.0148 | 0.0176 | 0.0198 | 0.0254 | 0.0298 |

Table 1: n versus to the run time of one program loop

A graph with a line going up

Description automatically generated

Figure 2: Table 1 represented graphically

Note that the change of is small at first but increases more rapidly as n increases. This is due to the complexity of the algorithm used to compute changes in velocity and acceleration for the bodies. Also, note that the last value of n which is 16.

# Future Directions

There are several directions a potential continuation of this study could follow. For one, the implementation of the engine was not as optimized as it could have been. As mentioned above, the algorithm used to calculate the required changes in position and velocity had a complexity of . While this is decent, a proper n-body engine used by professional organizations must be able to calculate changes for thousands or millions of individual particles. In a continuation of this study, I could utilize the Barnes-Hut algorithm, which has a runtime of , and study the effects using this algorithm has on as the value of n increase.

2D physics engines are easy to implement and provide a reasonably accurate representation of how particles interact with each other, however since we live in a world with three dimensions of space, 2D physics engines don’t capture the entire picture of everything that is going on. Another potential continuation of this study could be to implement this physics engine in 3D. Doing this would be simple in terms of the physics-based calculations, however it would come with many graphical challenges. Rendering 3D environments is much more mathematically rigorous than 2D environments. Though challenging, expanding the n-body engine to 3D would be an excellent way to continue this study in the future.

# Conclusion

In this study, I have learned a great deal about physics, programming and particle simulation. Specifically, I have learned about ways optimize CPU time and memory usage through performance analysis. By carefully examining the algorithms I was using and doing some research, I was almost always able to utilize more optimal ways of completing tasks. I have also learned about rendering graphics within game engines. Since I coded this project from scratch, I had to develop my own processes of rendering. This required some mathematical intuition, especially for developing the Anchor and Inverse Anchor transforms. Finally, I was most excited to get the opportunity to use the mathematical principles I have learned throughout school to simulate real-world physics digitally. This helped me broaden my understanding of the applications in which equations like the basic kinematic equations and Newton’s second law of motion are used. Overall, this study was enlightening, and I look forward to continuing my work on this project.

# References

Eastwood, J. W. & Hockney, R. W. *Computer Simulation Using Particles.* Taylor and Francis Group, 1988.

Millington, Ian. Game Physics Engine Development 2nd ed. CRC Press, Jan 2010.

Portillo, S. (2024, December 5). *N-body building*. NBody Building. http://portillo.ca/nbody/the-n-body-problem/

Szauer, Gabor. Game Physics Cookbook: Discover Over 100 Easy-to-follow Recipes to Help You Implement Efficient Game Physics and Collision Detection in Your Games. United Kingdom, Packt Publishing, 2017.

Tillman, N. T., Bartels, M., & Dutfield, S. (2022, January 5). *What is the theory of general relativity? understanding Einstein’s space-time revolution*. Space.com. https://www.space.com/17661-theory-general-relativity.html