Extensible Simulation of Planets and Comets

Natalie Wiser-Orozco
Department of Computer Science
and Engineering
CSU San Bernardino
San Bernardinio, California 92407-2393
Email: natalie@otsegoville.com

Keith Schubert
Department of Computer Science
and Engineering
CSU San Bernardino
San Bernardinio, California 92407-2393
Email: schubert@csci.csusb.edu

Ernesto Gomez
Department of Computer Science
and Engineering
CSU San Bernardino
San Bernardinio, California 92407-2393
Email: egomez@csci.csusb.edu

Richard Botting
Department of Computer Science
and Engineering
CSU San Bernardino
San Bernardinio, California 92407-2393
Email: rbotting@csusb.edu

***Abstract*—The research conducted is intended to enhance the way we view and study our solar system and others, by allowing scientists of astronomy, physics, and computer science to accurately simulate celestial systems. The Extensible Simulator organizes individual groups of celestial bodies, calculates their positions, graphically visualizes their movement using the computed positions, and finally, is extensible, which serves to accommodate additional numerical methods and gravitational functions, body shapes and behaviors, and camera views.**

I. Introduction

Accurate understanding the movement of objects in space is one of the oldest interests of mankind, and continues to be actively pursued. One reason we are interested in simulating solar systems stems from our interest in the survival of the life here on Earth. If an object is bound to impact Earth, we would like to know as soon as possible where and when it is going to hit, giving us sufficient time to prepare for the event. Another reason is such simulations allow us to examine the formation of the galaxies, and thus test hypotheses on dark matter, and other areas of modern physics. Further, these simulations allow us to plan space missions more accurately. Additionally, these simulations assist in the search for other solar systems with the hopes that they may be able to sustain life.

Extensive research has gone into how to solve the interactions of multiple bodies by inverse square laws, which is called the N-body problem. Sundman provided a solution to the three body problem in [12] and Qiu-Dong Wang [13] provided a global solution to the N-body problem (under certain conditions), although the solution is a series expansion and the convergence is extremely slow. The advent of computers has allowed the simulation of the N-body problem, which is faster but has its own challenges. The straightforward method is to calculate the interaction of each particle with every other particle (PP methods), which requires $O(n^2)$ time, where n is the number of particles, but gives good resolution. To address the complexity problem of PP methods, particle-mesh methods (PM methods), see [1], [7], were introduced, which approximates a field quantity on a mesh of size n_g resulting in a complexity of $O(n+n_g \log n_g)$. PM methods have problems with nearby particles due to the imposition of a mesh/-grid, so hybrid algorithms such as particle-particle/particle-mesh(P3M), see [11], and particle-multi-mesh(PM2), see [9], were developed. Another vein of research was in hierarchical methods or tree codes (TC), following the work of Appel in [3]. Probably best known of these was the work of Barnes and Hut, see [6], [5], and Greengard and Rokhlin, see [10], though many others utilized the tree approach, see [14], [15]. Tree codes are $O(n \log n)$ and do not have the mesh resolution problem, though PP methods can in theory be more accurate. Greengard's method is also called fast multipole [17], as it is usually cited to be $O(n)$, though this was disproved in [2].

There is no question that a plethora of simulators exist that simulate the movement of our solar system, though most of them focus on a static set of bodies that are very high in number[16], or that explore a single method or algorithm as a way of calculating such large systems[8][4]. It is spectacular to see the improvement in calculation power by employing more efficient algorithms and finding ways to more efficiently utilize processing power, however it would be nice to have a tool where as these new methods and algorithms are being forged, they could be easily added to one central framework, ready for visualization. This is precisely what separates our simulator from other simulators. While we do not explore any particular method or algorithm, we construct a framework that would easily allow multiple algorithms and methods to be used in order to arrive at a graphical simulation. To test the framework, however, we did employ the Runge-Kutta Fourth order method (RK4), along side Newton's Law of Gravitation adapted for n bodies. Using different technologies and techniques, we can easily adapt new numerical methods, gravitational functions, body shapes and behaviors, and camera views which will be discussed in the sections to follow. Throughout the process of its development, heuristics were employed to achieve maximum viewing pleasure.

In order to prove the accuracy of the simulator, we tested with four different sets of bodies, which will now and forever be referred to as 'projects'. These four projects were each run with different durations measured by earth years. To test the accuracy of the simulator, we compare the actual aphelion and perihelion values of the real planets used, with the ones obtained in the simulation.

II. Managing Sets Of Celestial Bodies

As with many integrated development environments(IDEs), the idea of projects that contain sets of files is commonplace, which is the very reason this structure was used for the simulator. Projects contain 'body configuration files' that represent exactly one celestial body, whether it be a comet, planet, asteroid, etc. These files contain a plethora of information pertaining to the body itself. In particular, we ask for:

1) Name - The name of the celestial body.
2) Initial x-coordinate position - The x-coordinate of where the body will start its simulation(aphelion).
3) Initial y-coordinate position - The y-coordinate of where the body will start its simulation(aphelion).
4) Initial x-coordinate velocity - The initial velocity of the body in the x direction with meters/second as the units.
5) Initial y-coordinate velocity - The initial velocity of the body in the y direction with meters/second as the units.
6) Radius - The radius of the body in meters.
7) Mass - The mass of the body in kilograms.
8) Red color component - Values range from zero to one.
9) Green color component - See above.
10) Blue color component - See above.
11) Texture image - The filename of the image to use in texture mapping the body. If there is no plan to use texture mapping, a value of 'none' should be used in its place.
12) Parent body - The name of the body who is the parent to this body. If there is no parent body, as in the case of the Sun, a value of 'none' should be used in its place.
13) Aphelion from parent - The distance that is furthest away from the parent in meters.
14) Perihelion from parent - The distance that is the closest to the planet in meters.
15) Gravitational constant - This should remain the same for all planets in all solar systems, at 6.67259×10^{-11}.
16) Rotation angle on the xy plane - A degree value that you wish to rotate your initial conditions stated above.
17) Body class type - 'Body' is the base class type used for any celestial body.

You will notice by the above descriptions, that only x and y plotting information is asked for with regards to the actual position of the celestial body. At this stage, only two dimensional simulations are implemented. While the movement of the planets happens in two dimensions, the scene rendered during the simulation is in fact, three dimensional. This allows for scene navigation in three dimensions.

The main functions of managing projects are fairly straightforward:

1) Creating a new project
2) Opening an existing project
3) Creating a new body configuration file within a project
4) Editing a body configuration file within a project
5) Deleting a body configuration file
6) Create an exact copy of an existing project
7) Select numeric method to use for the calculation
8) Select gravitational function to use for the calculation
9) Perform the calculation
10) Launch the simulation
11) Exit the application

Aside from managing projects, we have the scene navigation of the actual simulation to explore. See Table I. You can see in the table that there are two groups of keystrokes. The first group are defined as a built in part of the simulator and will always remain. The second group, however, is dependent on which camera is used for the simulation. This will be discussed further in Section III. For now, just keep in mind that different cameras can be used as a part of the simulation.

TABLE I
Keystrokes for the Simulator.

Key	Description
+	Increments the body index.
-	Decrements the body index.
a	Scales the bodies so they are more visible.
n	Renders the bodies as their normal sizes.
e	Exit the application.
↑	Move in the positive y direction.
↓	Move in the negative y direction.
→	Move in the positive x direction.
←	Move in the negative x direction.
b	Move in the positive z direction (backwards).
f	Move in the negative z direction(forwards).
r	Resets the scene.

III. Extending the Simulator

In order to make this simulator extensible, an application programming interface (API) was implemented. Before creating the framework to accommodate the API, we determined the most interesting aspects of the simulator, and formed the API around them. Those four aspects include the drawing of the bodies, movement and functions of the cameras used to view the simulation, and perhaps the most useful, the ability to implement different numerical methods and gravitational functions. As mentioned in the beginning of this paper, we used different technologies and techniques to achieve the best possible results for the simulator as a whole. The technologies used include the Python programming language in conjunction with Scilab.

The API for Cameras and Bodies both have a similar structure. The camera and body base objects obviously have very different functions, but they interact with the main program in much the same way. Since there will be many different variations of each class in their own right, it was necessary to 'wrap' each of them in their respective wrapper classes,

a ‘body wrapper’ and a ‘camera wrapper’. This was done in order to load the many variations of cameras and bodies into general structures that would remain the same throughout the life of the Extensible Simulator, and be able to interact seamlessly with the other components of the program.

To further aid in the management of these wrapper classes, respective management classes were also constructed. The ‘body manager’ and the ‘camera manager’ deal only with wrapper classes, making it easy to load all of the necessary sub classes that are extended from the base classes of the camera and body, and to also perform certain algorithms that require information from all of the loaded classes, enhancing the simulation. Figure 1 shows the structure of the framework, where the base camera and base body classes can be extended to include different or more refined functionality. The UML

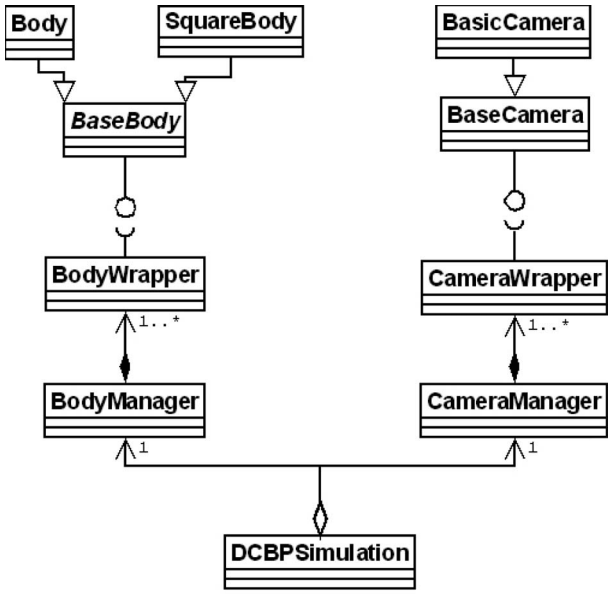

Fig. 1. Unified modeling language diagram of simulation components.

diagram above describes the business objects involved in the Python programming language only, which is why there are no references to gravitational functions or numerical methods. The gravitational functions and numerical methods are implemented as Scilab scripts that are called from within the python user interface. To extend the simulator to include different methods and functions, registering them with the user interface would make them accessible for calling in any simulation.

IV. Heuristics

While developing the Extensible Simulator some challenges arose, and what resulted were some interesting ways of solving these challenges. In the following section, an outline of the different problems and their solutions will be presented.

When a bare bones simulation was finally achieved in the early stages of the Extensible Simulator, it was apparent that we needed some way to scale the bodies. It was virtually impossible to see their small size given their actual dimensions. In order to preserve the integrity of the orbits of the planets, we couldn’t scale the bodies too much, as a larger body might completely swallow a smaller body, as in the case of the Earth and the Moon. What resulted was a heuristic that finds a scaling factor based on the shortest distance between two planets after evaluating all possible planet combinations. After the algorithm finds the two planets in the system with the shortest distance between them, it grows their radii until the distance between the surface of the two planets is no more than 30 times the smaller radius. The value of 30 was chosen by trial and error, because it rendered the best results with the Earth/Moon scenario. After getting the final results for the radii involved, the final scaling factor is found by how much of a percentage the smaller radius grew from its original value. This is then applied to all bodies involved in the simulation, and renders a scene where a majority of the planets can be seen in movement.

Yet another hurdle to overcome was zooming in and out while the camera is focused on a particular body. As the Extensible Simulator is constructed now, all the planets rotate about the z axis, and there is no account for the orbital inclination to the ecliptic, so all orbits lie in the same plane. It is for this reason that we keep the camera behind the xy plane on the positive z side. To zoom into any given planet, we subtracted a constant from our original z value to bring us closer to the origin. This worked fine for awhile, but in order to zoom very far away from the bodies, it would take awhile because we could only step back by that constant. It was then that the zoom heuristic was created. When we want to zoom far away from our system to get a birds eye view, we now double the current distance that we are from the planet in focus. To zoom in, we divide that distance by two. This logarithmic scale seemed to work much better because we could get that bird’s eye view more quickly, and as we zoomed in, we could get very close to the planet without going past it. This method posed its own challenges, where zooming out would cause all the bodies to disappear. This was happening due to the viewing volume not growing with the camera’s z viewing distance. In order to keep the planets in view, as we zoom out and zoom in, the viewing volume would increase and decrease accordingly, keeping all of the bodies contained and in view.

V. Miscellaneous

While putting together a test project, we gathered the necessary planetary data to simulate a set of planets. When it was time to set our initial conditions for velocity and position, we were setting the aphelion of every planet on the positive x axis, because frankly, it was the easiest. If we were to set it at a point not along one of the axes, we would have to do some elementary math to figure the x and y values for both the position and the velocity. An easy task to be sure, however why do it when you don’t have to? In order to alleviate that step, we have included Givens rotations, which allows an entry for the degrees to rotate the specified initial conditions for the body in question. In the Givens rotation, we account both for the position and the velocity properties of the body, and

by doing so, we also rotate the point at which the planet's aphelion occurs since our x and y initial conditions occur at the aphelion for each planet.

In order to better visualize the trajectory of the bodies, we added a tail that would trail the body as it moves along it's path. The length of this tail, when first implemented, would follow the planet continually, no matter how long the planet would orbit it's parent. We experienced increased sluggishness during the duration of the simulations. Since it was wasteful to keep tracing the orbit as the body moved around due to overlapping, and with the simulation becoming slower and slower, we decided to shorten the tail to increase simulation speed. We ended up drawing the tail for only the most current 1,000 time steps. The time steps themselves are determined by a combination of the numerical method used, and the duration the simulation. Here, using the number 1,000 as the constant was chosen by trial and error, because this number was high enough to convey the trail of the body, but short enough to keep the simulation from becoming sluggish.

VI. Results

Error analysis was performed on the accuracy of the measured aphelion and perihelion versus their true values, and also the angle at which aphelion and perihelion occurs versus their starting angle. Specifically, I looked at the relative error between the measured and true values, which can be defined as

$$\epsilon = \frac{|v_{approx} - v_{true}|}{|v_{true}|},$$

where ϵ is the resulting error, which portrays the digits of accuracy between the two values, v_{approx} is the measured value, and v_{true} is the expected value.

While running the initial tests to see how the RK4 ODE solver was handling the data, I realized that after a simulation time of 40 years, the error of the Moon was sky-rocketing. The step size in the solver was so large, that the calculated positions for the Moon became very inaccurate towards the end of a 40 year simulation. In fact, the RK4 solver could not finish integrating.

After finding this, I decided that negotiating step-size was essential. I experimented with a few different values, and came up with a step-size that varied based on simulation duration. Using the new step sizes I was able to get a handle on the error for the Moon. This was the only body that had a problem with the errors due to it being a few orders of magnitude smaller than the other planets, and having a much faster period as well. This poses a problem, because with a much larger step size, one step could be a large portion of the Moon's orbit, thus giving us a highly inaccurate value for its position.

The other factor to consider is the time it takes to calculate a simulation. With a larger step-size, the time to calculate would be small, because there would be less calculations to perform. With a smaller step-size, we would have more calculations to perform, resulting in a longer calculation time. My goal was to find a happy medium, that would give us accurate results in the shortest amount of time possible. This is why, for every Earth year, I multiply by a constant of 500 to get an optimal step size. This particular number yielded the most accurate solutions in the shortest amount of time.

At a duration of 100 Earth years, we are at an average of 2 digits of accuracy for the aphelion and perihelion of all the planets with the exception of the Moon. Our errors aren't too bad, but aren't great either if we adopt the general rule that 3 digits of accuracy is a fair result. This could be due to the fact that our system contains a mere 5 bodies compared to the eight that are in our solar system. We are missing Jupiter, which has a very large mass, and could certainly have an affect on the system as a whole.

We ran the same tests again with Mars and Jupiter, and with their inclusion, we don't see much of a difference in the significant digits of our measurements based on the relative error. Some factors that could result in less than favorable results are the angles of the true planets' orbits themselves. For these projects, we picked arbitrary angles that didn't necessarily reflect the measured angles at which they occur in our actual solar system. With the exception of the angles of the Moon's orbit, however we still get fair results at about 2 digits of accuracy. What really throws off our overall averaged errors are the angles at which the Moon's perihelion and aphelion occur. This could be due to a minor distance change in aphelion/perihelion that greatly affects the angle at which it occurs. Our step size in the RK4 algorithm may still be too large for the Moon's orbit, which may be skipping over a significant enough area, which could skip right over a true aphelion/perihelion point resulting in a significant error at the angles at which they occur. This can perhaps be avoided by employing a numerical method that treats systems containing varying periods.

VII. Conclusions and Future Work

The aim of this research was to create an extensible simulator that will not only calculate the trajectories of planets and comets using a numerical method, but be able to graphically simulate them. This will be particularly useful to colleagues in the fields of astronomy, physics, and computer science, by being able to convey ideas easily to one another with confidence that the results are accurate to within an average of two digits with the exception of the Moon.

Additionally, we structured this program in such a way to make it easily extensible by providing an API to allow the extending of cameras, bodies, gravitational functions and numerical methods. These features are also very useful, because they allow scientists to add more features as needed, with a great deal of ease. If one wanted to define a different camera that follows a specific path throughout the simulation, that could be easily achieved. If one wanted to define a body of a different shape, or add rotation to the shape itself, they can do so through the body portion of the API. The gravitational functions and numerical methods are employed as scripts that can be added and used easily. These factors give this program

the potential to become a very robust simulator, capable of many different and intricate simulations.

The research conducted here, along with the created program can also facilitate research in the following ways:

1) Add the third dimension to the calculation of the trajectory taking into account each body's inclination to the ecliptic.
2) Dynamically speed up the simulation while the simulation is running.
3) Add a GUI to the actual simulation screen that allows easier ways of switching between cameras and determining which body the camera is looking at.
4) Add more numeric methods to the simulation to give the user different options. For instance, methods that treat different periods differently. This could aid greatly with the inconsistency of the Moon's results compared to the other bodies' results.
5) In the existing GUI, add the ability to edit project files and body configuration files more easily, instead of having to edit them by hand. Adding in the ability to input values in scientific notation would also improve the usability tremendously.
6) Programmatic video capture of a simulation would be of great interest, allowing one to capture a specific simulation, and be able to re-play it anywhere, eliminating the need to have a machine that contains the software needed to run the Extensible Simulator. This would aid greatly in presentations of specific scenarios.
7) Automatically calculate the necessary velocities at aphelion and perihelion for the user.
8) Develop an algorithm to determine an ideal step size based on the magnitude of the bodies, their orbits, and numerical method used.

In closing, this tool can help us understand a bit more, how our universe really moves. It enables communication over a broad spectrum of backgrounds and experiences from those in highly technical or scientific fields to those who are not, and can, but is certainly not limited to, helping us detect disasters that may threaten our existence on Earth, or aiding in the prediction of stable systems elsewhere.

REFERENCES

[1] E. Saar I. Suisalu A. Klypin A. Melott, J. Einasto and S. Shandarin. Cluster analysis of the nonlinear evolution of large scale structure in an axion/gravitino/photino dominated universe. *Physical Review Letters*, (51):935, 1983.

[2] Srinivas Aluru. Greengard's n-body algorithm is not order n. *SIAM Journal on Scientific Computing*, 17(3), May 1996.

[3] A.W. Appel. An efficient program for many- body simulation. *SIAM J. Sci. Stat. Comput.*, (6):85–103, 1985.

[4] J. S. Bagla. Cosmological N-Body simulation: Techniques, Scope and Status. *Current Science*, 88:1088–1100, April 2005.

[5] J. Barnes. A modified tree code: Don't laugh; it runs. *J. Comput. Phys.*, (87):161–170, 1990.

[6] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, (324):446–449, 1986.

[7] M. Davis, G. Efstathiou, C. Frenk, and S.D.M. White. The evoloution of large-scale structure in a universe dominated by cold dark matter. *ApJ*, (292):371–394, 1985.

[8] D. J. D. Earn and J. A. Sellwood. The Optimal N-Body Method for Stability Studies of Galaxies. *The Astrophysical Journal*, 451:533–+, October 1995.

[9] Sergio Gelato, David F. Chernoff, and Ira Wasserman. An adaptive hierarchical particle-mesh code with isolated boundary conditions. *ApJ*, May 1997.

[10] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, (73):325–348, 1987.

[11] Randall Splinter. A nested grid particle-mesh code for high resolution simulations of gravitational instability in cosmology. *MNRAS*, (281), 1996.

[12] K.F. Sundman. Mémoire sur le problème des trois corps. *Acta Math.*, (36):105–179, 1913.

[13] Qiu-Dong Wang. The global solution of the n-body problem. *Celestial Mechanics and Dynamical Astronomy*, 50(1):73–88, 1991.

[14] M.S. Warren and J.K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proc. Supercomputing '92*, pages 570– 576, 1992.

[15] M.S. Warren and J.K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proc. Supercomputing '93*, pages 1–12, 1993.

[16] Dr. Bernd Wirsing. Supercomputer simulations explain the formation of galaxies and quasars in the universe, June 2005.

[17] F. Zhao and L. Johnsson. The parallel multipole method on the connection machine. *SIAM. J. Sci. Stat. Comput.*, (12):1420–1437, 1991.