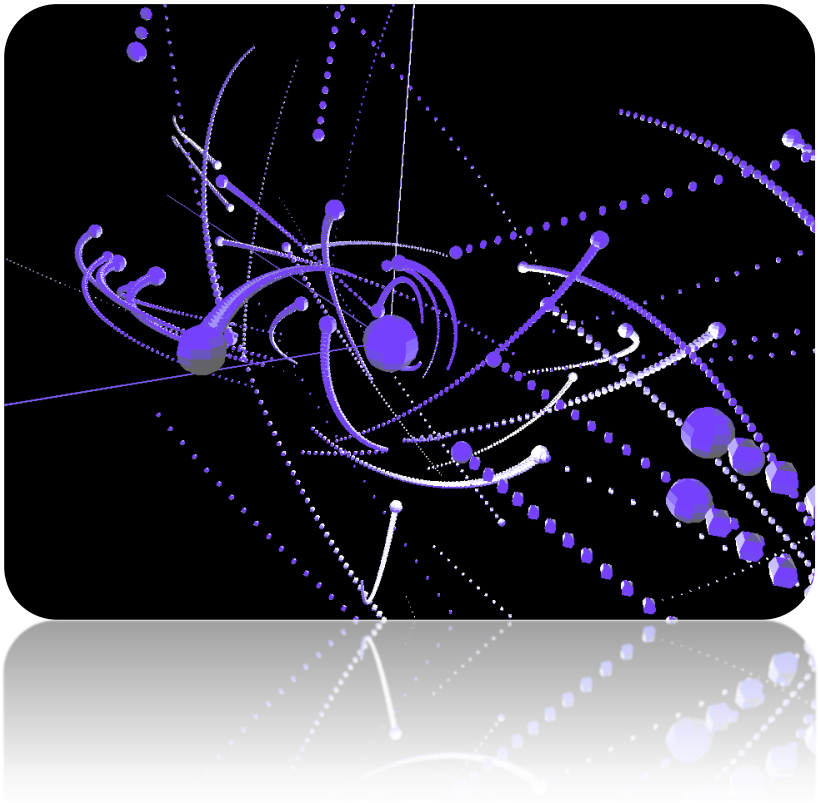
Peter Cottle

CS 184 Fall 2011

Final Project Report

GSim – An Interactive Gravity Simulator



Background:

Particle simulators have been implemented many times over the history of computing and have been used for a wide range of applications from animation generation, molecule analysis, galaxy formation research, videogames, and even some forms of art[1]. My final project "GSim" set out to create a gravity particle simulator that supported user interaction, file import / export, and intuitive navigation to deliver an enlightening (and fun) user experience.

Education in schools has recently been trending towards providing a more interactive, individualized experience for each student through the use of technology[2]. Gravity itself is a particular example where live interaction can really enhance student learning in the classroom, especially since all students come in with a pre-conceived notion of gravity as a constant force. If we can show our

students the mutual attraction nature of gravity through the use of a simple GUI application like GSim, we can both inspire students to pursue the sciences and enhance the learning experience.

Description of Project:

GSim is a gravity particle simulator that is capable of simulating dozens of particles while interacting with the user in an intuitive way. It uses a Verlet integration scheme to preserve the energy of the simulated system and employs a set of 3-2-1 Euler angles for camera orientation in space. User interaction for selecting particles is as natural as a mouse click, and the insertion procedure for particles only requires a few click-drags to specify position, mass, and velocity.

GSim simulates all particles in three full dimensions, unlike many web-based particle simulators that exist today. It's closest relative is Universe Sandbox® available on Steam, a commercially successful game that allows users to simulate universe creation and destruction. GSim aims to recreate this experience for free.

Implementation:

The implementation of this project was mainly divided into three categories: integration method, camera navigation, and user interaction in 3D. The rest of the implementation was fairly standard object-oriented programming with extra methods to render the scene, calculate particle trails, perform file input/output, and other miscellaneous tasks.

3.1: Integration Method

Euler integration, Runge-Kutta integration, Midpoint integration, and many other explicit integration schemes are widely popular in ODE-solving (like ode45 in Matlab) and are easy to implement. Their main drawback is that they do not preserve the mechanical energy of the system (in both potential and kinetic form). The mathematical proof of this is quite rigorous, but proof-by example is easy to demonstrate (and was covered in class as well). The summary of the shortcoming of these integration methods is that they assume the acceleration is constant over a given timestep, when acceleration is commonly a function of position (like in systems with springs or gravity). In these systems, the acceleration of a particle actually changes constantly throughout time, so an explicit integration method will

always over-estimate (or under estimate) the acceleration during the timestep and either add or subtract energy to the system. The simplest example of this is Euler integration with a simple spring-mass system, where the acceleration estimate will always be an overestimate on one side and an underestimate on the other. This leads to energy addition through time, causing the system to "blow up" or lose stability.

The Verlet integration scheme overcomes this simulation obstacle by making the calculation for the next velocity a function of current velocity, the current acceleration, and the acceleration at the *next* timestep forward. If acceleration is dependent on velocity (e.g. drag in fluid simulation), this requires the additional overhead of an implicit equation solver. If acceleration is instead simply a function of position, this integration scheme reduces to a simple "leapfrog" integration method.

Because of this acceleration calculation difference, Verlet integration actually maintains the energy of dynamic systems and leads to very stable behavior during simulation. In my implementation of GSim, you can both integrate forward in time and backwards in time with variable timesteps. Because of my choice of integration method, one can integrate forward for several minutes, pause, integrate backwards with a *different* timestep, and arrive at the exact same system configuration as before.

3.2: 3-2-1 Set of Euler Angles

Camera orientation and navigation represent a non-trivial problem to overcome with computer graphics simulation. An additional difficulty with this is that the OpenGL community recommends loading from identity for each frame draw (both to simplify drawing within the scene and to eliminate memory overflow problems with the matrix stack). Thus, a navigation and orientation method must be produced that allows intuitive translation and orientation in 3D space, has a constant memory overhead, performs in constant time, and is capable of orienting the camera from the original configuration on every frame draw.

The main challenge with intuitive interaction is that all changes in orientation and position are relative to the current reference frame. When a user presses the left or right arrow keys to yaw, he or she wants to yaw about whatever direction is "up" for the camera in the current configuration, not the actual positive z axis in the world space. This presents a considerable challenge, for the up "direction" can be any vector in 3D space (since the user can orient themselves to any configuration). In planar 2D motion (in 3D space) in applications like first-

person and third-person shooters, this problem becomes less challenging because the user is limited to two rotations (yaw and pitch) and two translations (within the plane). Thus, simple variables can keep track of the current orientation, and one rotation in yaw will never affect the pitch variable. This is not true for actual 3D motion, hence the challenge to overcome.

Fortunately for me I am concurrently enrolled in ME 175, the mechanical engineering class of Intermediate Dynamics taught by Professor Oliver O'Reilly (a distinguished professor and winner of four teaching awards at Berkeley). In class, Professor O'Reilly presented and discussed a paper on rocket navigation that uses one gyroscope and accelerometer for each of the three principle axes of the rigid body. One of the main results of the paper was the determination of the rate of change of each Euler angle given a measured angular velocity about each principle Euler basis vector. This result essentially relates the *local* angular velocity to the *global* Euler angle rate of change. This is the exact relation I needed, for I wanted to allow the user to specify *local* angular velocities to navigate while maintaining a *global* system of positioning. This relation is provided below:

$$\begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & sin(\phi)sec(\theta) & cos(\phi)sec(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}$$

Although this equation was used for a quite different application, it had direct use in my camera orientation and navigation method. The main computational flow for the navigation and orientation for the camera is described below:

1. Allocate a translation vector, three Euler angles, and three basis vectors into memory.
    a. The three basis vectors will be described as e1, e2, and e3 (as in common notation).
    b. For the camera, the e1 direction will be looking "forward" into the screen, the e2 direction will be the horizontal direction in the plane of the screen, and the e3 direction will be the vertical direction in the plane of the screen.

2. Initialize the Euler angles and translation vector to a default starting value in order to "point" the user at the origin.
3. Calculate the three Euler basis vectors based on the concatenation of the 3-2-1 Euler angle rotations. Essentially, combine the yaw, pitch, and roll rotation matrices into a combined rotation matrix.
4. Multiply this combined rotation matrix into the three axis vectors (X, Y, and Z) to obtain the Euler basis vectors.
5. In order to orient the camera, use the OpenGL utility "lookAt" command that takes in a camera position, a "look at" position, and a roll amount:
    a. Calculate the position by specifying the orientation vector
    b. Calculate the "lookat" position by adding the e1 "looking forward" vector to the translation vector to obtain a point forward from the camera.
    c. Specify the roll by providing the e3 vertical direction of the camera.
6. When a user specifies a translation by using the WASD, Q, and spacebar keys, add small fractions of the three basis vectors (one for each desired direction) to the translation vector to *move the camera but preserve orientation*.
7. To change the orientation and preserve translation, allow the user to specify a local yaw, local pitch, or local roll by pressing one of the arrow keys or the + and – keys.
    a. This keypress translates to inducing an angular velocity about one of the three Euler basis vectors.
    b. Take each one of these specified angular velocities and calculate the corresponding change in the three Euler angles as prescribed by the paper result.
    c. Combine these Euler angle changes together to get a net rate of Euler angle change.
    d. Integrate this change in time, and calculate the three new Euler angles.
8. Repeat part 3 until program exit.

The above implementation gives an orientation scheme that allows intuitive translation and orientation of the camera while preserving a global positioning system loading from identity on each frame, using constant memory, and performing in constant time.

Additionally, this orientation scheme retains the orthogonality of the rotation matrix (an issue for the method presented in class) in addition to avoiding numerical error buildup. The main drawback of Euler angles is the gimbal-lock scenario where two axes become aligned and a loss of a degree of freedom occurs. If a user navigates into gimbal-lock and then tries to rotate about a degree of freedom that has been lost, another entirely different motion will occur instead. For example, if one were to pitch all the way down to 90 degrees and then attempt to yaw, the camera would instead roll because those two axes are now co-linear and anti-parallel.

My orientation method overcomes this issue entirely **because the orientation is calculated from identity** for each frame draw. A 3-2-1 set of Euler angles can represent **any** orientation; the only problem arises when one tries to navigate away from a gimbal-lock scenario. Because my method never interpolates *between* Euler angle representations and instead recalculates the configuration every time, it avoids any gimbal lock issues and thus eliminates the main drawback of Euler angles.

Furthermore, the **key** feature of this orientation scheme is that it allows one variable (the psi angle) to describe the local yaw amount of the camera. Because the 3-2-1 set of Euler angles only yaws once (and it yaws before any other rotation occurs), the psi angle can be easily used to describe the amount of yaw in the current orientation. Only this amount of yaw affects the angle of the XZ plane to the camera, which is absolutely **critical** for the later ray-picking method for user interaction.

In conclusion, the orientation and navigation method was the main objective to overcome when programming GSim and my proudest result from this project. A video of this navigation is available on Youtube via the link given in the appendix.

3.3 Graphic interaction in 3D space for particle insertion, mass selection, and velocity.

Once the orientation and navigation method was implemented, I next needed a way to allow the user to interact with the simulation. This included three separate interactions: selecting a particle for deletion, launching a particle with a random velocity from the current position, and inserting a particle in 3D space.

All of these interactions use a simple ray-picking method that utilizes the OpenGL “unProject” utility function. A mouse coordinate is mapped into 3D space on both the near and far drawing planes. These two coordinates then form a ray (with a start location and direction). This ray is then used for the three interactions.

The first “selection” interaction uses the ray sphere intersection test to find the closest sphere along this way. Once this sphere is found, it is “selected” and this selection is represented by a rotating torus around its body. The user then can either delete this particle with the delete key or de-select it by clicking it again.

The second “launch random particle” interaction is also easy, for it uses the ray origin position as the position for a new particle. This is then supplemented by a simple random velocity.

The third particle insertion interaction is far more complicated. It involves specifying a position, a velocity, and a mass for a given particle. This represents seven different variables that have to be calculated from a minimal amount of user interaction. Additionally, the difficulty with mouse interaction in 3D space is that there is no way of easily determining how “far” away the click was meant to be.

My implementation of this was inspired by Google Sketchup which assumes a default “intersection plane” depending on orientation. I intersect all mouse clicks with the XZ or YZ plane that intersects the origin. Additionally, I choose between the XZ or YZ plane depending on the psi “yaw” variable from the orientation scheme, allowing for a very easy way to determine which plane the user wants depending on their orientation. Because both intersect planes are co-planar with the yaw vector, these two planes are easily differentiated with a single variable.

Position, mass, and velocity each have a graphical representation during the insertion process. Position is represented by the position on the screen (obviously) and also a large torus centered at the origin that gives a sense of scale to the radius. The mass is relative to the volume of the particle, and a cone represents the velocity. The orientation of the cone describes the direction of the velocity and the width describes the magnitude. With these three elements and the plane intersection with the ray picking, user interaction with GSim is natural and frictionless.

Results:

After implementation was complete, I had a gravity simulator that was quite fun to play with and also educational about the laws of gravitational attraction. I tested the user interaction by asking several of my friends to insert a particle with no further instruction. After the first click, people realize a position is being modified and then move to where they want the particle to be. The second two clicks are equally natural for mass and velocity, and then the particle fires off once it’s completely selected. It brought a smile to my face to see 3D interaction being so intuitive in a program I wrote. I am also offering up the source of this project on

Github in hopes that someone will fork the project and expand on it (or perhaps give it to a teacher to use in class).

In conclusion, I am pleased with my final project and specifically pleased with the ways I overcame implementation challenges.

References:

[1] "Nanotechnology Now - Press Release: "Linux Labs Announces Completion of Key Development Milestone for Its Super Computer Operating System, NimbusOs; This Milestone Will Expand the Available Markets Its NimbusOs Products Can Be Sold To, Such as the Nanotechnology Market "" *Nanotechnology*. Searchlight Solutions, 12 Nov. 2011. Web. 04 Dec. 2011. http://www.nanotech-now.com/news.cgi?story_id=43850.

[2] Glum, Julia. "More Colleges Using Video Games as Educational Tools - The Independent Florida Alligator: Campus." *The Independent Florida Alligator: We Inform. You Decide.* 2 Dec. 2011. Web. 04 Dec. 2011. http://www.alligator.org/news/campus/article_94efe934-1ca9-11e1-808f-001871e3ce6c.html

[3] Pritchard, Carolyn. "How the Physical Distribution of Digital Goods Impacts the Environment — Cleantech News and Analysis." *GigaOM — Tech News, Analysis and Trends*. 16 Aug. 2007. Web. 04 Dec. 2011. http://gigaom.com/cleantech/how-the-physical-distribution-of-digital-goods-impacts-the-environment/

Appendix:

It is estimated that an entire kilogram of carbon dioxide is produced for the manufacturing and delivery of a compact disc[3]. In order to strive for a carbon-neutral educational environment, I have made the video that accompanies this project report available online at Youtube via the link below:

http://youtu.be/r_68uPETMJk

Google's data centers are industry leaders in Green technology, and it is my hope that this video can serve wider audiences while maintaining a minimum carbon footprint.