3D Physics Simulation

Steven Durant

June 8, 2006

Abstract

The creation of any physics simulation is simply applying the laws of physics into a virtual environment on a computer. All physical laws are taken into account for the interactions between the various particles and results can be seen on screen. My simulation takes into account gravity, elastic collisions and conservation of momentum among other things. These physical rules result in a visual display via OpenGL. Various cases can be programmed in and then the particles can be observed as they interact with one another.

1 Background

Historically the purpose of the physics simulation is to create a realistic three dimensional environment in which bodies can interact. These simulations are hard to create realistically because of all of the things that factor in; however, the maker can choose exactly how realistic he wants his specific simulation to be. For example a simulation could be successful without factoring in torque and spinning of objects, without these the simulation still works. Once the backbone behind a simulation is created the author can edit it to add or remove various effects such as torque, collisions, gravity, elasticity, etc. Typically physics simulations can be found on the Internet for gravity, springs, collisions, or anything physics related at all.

2 Description

The purpose of writing a simple physics simulation is to see what happens when various test cases are loaded into the simulator. For example if the simulation is known to run properly, we could theoretically assign millions of random points in space and see what happened over thousands of years. All of the units would be physically accurate and the time scale would be realistic. This way the simulation would actually predict what would happen in a real life situation. When used to place millions of particles randomly in space and see what happens, time would be sped up to millions of times faster than actual time and the user could see what would theoretically happen in the given situation. The solar system is predicted to have been created by the big bang for example and an experiment like this could show whether or not this is realistic at all. My simulation is currently somewhat simplistic. The user can define any number of spheres to be loaded into the simulator and the program will randomly distribute the spheres around a sphere of user-defined radius. The radius of each sphere is randomized along with its position and small starting velocity in three dimensions. Each sphere also has a density function which is used to calculate the mass of the sphere based on the radius. Thus when the user enters a larger and smaller sphere the collisions are accurate based on density and size. The gravity is completely realistic and is used to change the accelerations, velocities, and ultimately positions of the spheres. Collision detection is implemented when the spheres move too close to one another, they will exchange momentum and bounce off in opposite directions. The simulation can be restarted and I am currently working on making a function that will randomize a solar-system type orbit where any number of small spheres are orbiting a much larger one.

3 Methodology

The simulation uses basic laws of physics in order to determine what the interactions between the various spheres are. Force is given by the equation

and is calculated between every pair of spheres. The total force on each sphere is added up between every time step and then used to calculate the resulting acceleration of the sphere, which is given by [a = F / M]. Since my simulation takes place in three dimensions each sphere is represented by a structure that includes a position array with (x,y,z) values, a velocity array with (x,y,z) values, and an acceleration array with (x,y,z) values. In order to calculate the force more simply a unit vector is found between the two spheres based on their position vectors. The magnitude of this is taken, which is the distance between the two objects. Force is calculated based on the distance and then the Force scalar is multiplied by the unit vector along the line between the two spheres. This Force vector is then divided by mass in order to create the three dimensional acceleration vector for the sphere. This process is applied to all of the spheres which results in an O(n squared) efficiency for this part of the code. Unfortunately there really isn't any way to make this run more efficiently unless I negate the effects of small enough spheres. After the acceleration vectors are created for each of the spheres I used Euler's Method to calculate the resulting velocity of each sphere, which is in turn used to calculate the resulting position of each sphere. Euler's Method is [Vx' = Vx + Ax * deltaT] for velocity and [Px' = Px + Vx' * deltaT] for position. Namely, the new velocity will be the old one plus the acceleration times the time step for the calculation. In order to use Euler's Method for velocity and position I implemented a time step variable which is modifiable via keyboard input. The 'A' key will accelerate time and 'Z' will decelerate time. The new positions are calculated for every sphere based on the calculated time and then all of the spheres are moved and the program moves on to the next time step. Also, every time step yields a collision check for every given sphere against each other sphere. This could be more efficient but it takes very complicated math to determine whether or not any given sphere should be ignored or checked for collision so I have not currently implemented this in my simulation. Basic collision detection while using Euler's

Method also brings about a complication with a loss of energy. What happens is that the position calculations do not check in advance whether or not the new positions are taken or open. After all of the spheres move they end up being drawn overlapping each other in some cases. The collision detection assigns a new velocity to each sphere based on conservation of momentum, namely... [Vx1' = Vx2 * M2 / M1] and [Vx2' = Vx1 * M1 / M2]. These are due to conservation of momentum and are physically accurate but the spheres still lose energy due to the overlapping of the spheres. Eventually after a number of collisions the spheres will lose all energy and collapse on top of each other. The way to get around this is very complicated, the time steps have to be calculated in advance for all of the spheres and then checked to see whether two spheres will collide with each other. The next collision to occur has to be found based on the distance between the spheres and the velocities that the spheres are moving at. After going through all possible collisions and finding the next one the time to collision is calculated, next the spheres will do a fraction of the time step until they are exactly touching, and then the rest of the time step is calculated with their new velocities based off of the collision. This method allows for perfect collision detection and no loss of energy; however it is very processor intensive, I am currently trying to figure out a way to prune this somehow and make it run more efficiently.

4 Analysis

Despite the collision detection not being completely accurate at this time most simulations I run with my program are innately inaccurate; however if the particles do not collide during the execution of the simulation the results are accurate. Upon randomizing small starting velocities and masses of two particles approximately one out of five simulations results in one particle orbiting the other. Observing the resulting orbit of the two particles is enough to show that the physics is accurate for a non-collision stand point. The orbit will continue indefinitely as it should and the particles will never collide. Also the orbiting particle is always visibly smaller than the one being orbited and the orbit traces out an elliptical path much like that of the Earth around the Sun. Kepler's laws also seem to be kept accurate although I do not have a way of being completely certain. The orbiting particle most definitely speeds up when closer to the larger particle and slows down when further away. Speed changes are more drastic in more elliptical orbits. After I have fixed the collision detection I can go back and calculate the area swept out by the orbiting particle and display it on the screen.

5 Conclusion and Future Plans

The simulation is currently mostly physically accurate but in order to test any real situations I need to get the collision detection perfectly accurate. As shown in the Methods section this will take substantial mathematics and time and I am going to attempt to get this working once I have finished updating my paperwork. Once the collision detection is accurate I can perform substantial tests on my program such as printing out the total energy and momentum of each of the particles and the system as a whole and making sure that these values are constant. In addition I can implement inelastic collisions where the particles will stick when they hit each other and then gravitational forces will cause torques on the combined particles. Currently my physics simulation is perfectly fine for orbiting particles and I am working on a method to randomize particles such that they are all orbiting one significantly more massive central particle. I have run into a few mathematical problems here as I cannot figure out how to randomize a vector that is perpendicular to the position vector of the orbiting particle from the center of the system. The problem is that I could find a perpendicular vector to the central mass, but not a random vector on the plane that is perpendicular to the position vector. I am hoping to get this method working soon so that I can perform testing on the orbiting condition which does not involve any collision at all.

References

[1] Tipler, Physics For Scientists and Engineers : Vol. 1: Mechanics, Oscillations and Waves, Thermodynamics. W. H. Freeman; 4th edition, 1998.

[2] Molofee, Jeff. NeHe Productions. 18 Apr. 2006. ¡http://nehe.gamedev.net¿.