TJHSST Computer Systems Lab Senior Research Project
An Interactive, User-driven Physics Simulator
2007-2008

Tom Smilack

May 23, 2008

Abstract

Physics simulations are often of single concepts or immune to user control. My project aims to change that by allowing users to create a situation and then simulating the behavior of objects in that situation. Users will create objects either through shape tools, then the program will convert them to polymorphic objects and run the simulation. Objects varying from the simple to complex will be modeled: single shapes or multiple shapes connected statically or with axles.

Keywords: physics, simulation, interactive, ASSIST

1 Introduction

The majority of my research was in physics simulation: how to do it accurately, what equations to use, and how to implement them. Using the equations and properties that I give objects, the program determines and shows the way that the objects behave. I started with basic equations and added more complex ones as the year progressed.

This project models projectile motion and interaction between simple and complex objects. I define simple objects as rectangles or circles, and complex objects as multiple simple objects connected by pins or axles. Interactions include collisions, friction, and rolling. Objects can also be anchored to the background to provide platforms or obstacles.

My goal was to create a program, usable by anyone, that would help the user to gain a better understanding of physical interactions by inputting any situation using an intuitive input system and viewing the behavior of the system. The process of creating the program would also help me to gain a better understanding of physics.

2 Background

A team from MIT created ASSIST: A Shrewd Sketch Interpretation and Simulation Tool which inspired this project. The program was created in order to give engineers a way to model systems in the early stages of design, when only an idea exists, before a traditional CAD program, which requires precision and planning, would be appropriate. The user draws a mechanical system on a smartboard, including an arrow for gravity. The "sketchpad" system then interprets the drawing. Certain symbols have special meanings: an x is an anchor, a small circle is a pivot. Finally, the interpreted drawing is fed into a commercial simulator. My project was inspired by ASSIST and aimed to be similar but with more focus on the physics rather than the sketching.

3 Testing and Analysis

The two main sections of my program are the simulation and the objects. The simulation is implemented

both in the main program file - that is, the file containing the main timer - and the objects themselves. The main file contains an ArrayList of SimObjects and at each timer iteration it calls the step and draw functions of every object. Each object is an instance of a subclass of the abstract class SimObject. SimObject defines step, which updates the object's position and velocity when it is passed a double value dt. It also includes a signature for the abstract method draw, which is implemented differently in each subclass. The subclasses are currently Rectangle and Circle.

Circles are easy to draw, but rectangles are more complicated because their rotation changes the way they must be displayed. When the rectangle is created, I determine the angle from the center to each corner. When drawing the rectangle, I add its rotation to the angles already found and multiply the sine and cosine of those by the distance from center to corner to determine where to draw the points. Other polygons should be similar in implementation to the rectangle, as I treat it more like a set of points than as a rectangle. The method fillPolygon is used to display it.

Complex shapes will be implemented using pins and axles. Pins will connect two shapes so that they stay together in the same position. To achieve this I will create a ComplexObject class that will contain a list of shapes that combine to form it. It will calculate collisions for every object in it and apply forces to each object so that they move in unison. Axles will be more complicated; I will have to give each object independent motion while still keeping them attached to each other.

Collision detection has been the most complicated part of the project so far. It is easy to find when something is past a wall - check every corner to see if the x and y values are within an acceptable range. Determining whether an object is in another is more difficult. For circles, one must check if the distance from the center C to the point P is less than or equal to the radius:

$$\sqrt{(C_x - P_x)^2 + (C_y - P_y)^2} \leq r \tag{1}$$

For rectangles, one must treat each edge as a line and determine whether the point P is inside the area enclosed by each line. The equation of each line is the point-slope equation with y isolated on the left. If the topmost point is T, the leftmost is L, the bottommost is B, and the rightmost is R:

$$\overline{LT}(x) = \frac{T_y - L_y}{T_x - L_x}(x - L_x) + L_y \tag{2}$$

$$\overline{LB}(x) = \frac{B_y - L_y}{B_x - L_x}(x - L_x) + L_y \tag{3}$$

$$\overline{TR}(x) = \frac{R_y - T_y}{R_x - T_x}(x - T_x) + T_y \tag{4}$$

$$\overline{BR}(x) = \frac{R_y - B_y}{R_x - B_x}(x - B_x) + B_y \tag{5}$$

There is a collision when the following conditions are satisfied:

$$L_x < P_x < R_x \tag{6}$$

$$P_y < \overline{LT}(P_x) \tag{7}$$

$$P_y < \overline{TR}(P_x) \tag{8}$$

$$\overline{LB}(P_x) < P_y \tag{9}$$

$$\overline{BR}(P_x) < P_y \tag{10}$$

In the event that the rectangle is straight up or to a side - in other words, $\theta \ \% \ \pi/2$ is 0, then T, L, B, and R are sides rather than points, and the equations become simpler:

$$L < P_x < R \tag{11}$$

$$B < P_y < T \tag{12}$$

Wall collisions and object collisions are both resolved using similar equations. When an object collides with a wall[**?**]:

$$\overline{v}_{a2} = \overline{v}_{a1} + \frac{j}{m_a}\overline{n} \tag{13}$$

$$\omega_{a2} = \omega_{a1} + \frac{(\overline{r}_{ap} \times j\overline{n})}{I_a} \tag{14}$$

$$j = \frac{-(1+e)\overline{v}_{ap1} \cdot \overline{n}}{1/m_a + (\overline{r}_{ap} \times \overline{n})^2/I_a}, e = elasticity \tag{15}$$

When two objects collide, there are two more equations, and the final one changes [**?**]:

$$\overline{v}_{b2} = \overline{v}_{b1} - \frac{j}{m_b}\overline{n} \tag{16}$$

$$\omega_{b2} = \omega_{b1} - \frac{(\overline{r}_{bp} \times j\overline{n})}{I_b} \tag{17}$$

$$j = \frac{-(1+e)\overline{v}_{ap1} \cdot \overline{n}}{1/m_a + 1/m_b + (\overline{r}_{ap} \times \overline{n})^2/I_a + (\overline{r}_{bp} \times \overline{n})^2/I_b} \tag{18}$$

In order to prevent any possible glitches with resolving collisions more than once, and because the process must be done at the same time to both objects, I have created a Collider class, whose method, collide, is called whenever there is a collision between two objects. The Collider takes both objects as arguments and resolves the collision between them, although collisions are not working perfectly. To resolve a collision, it is necessary to know the direction of the normal vector that protrudes from the object (A) into which the other object (B) travels. This would be easy to determine if the side through which the B passed were known, but I am not sure how exactly to find it. Currently I think that I could look for what side of A the majority of B is on and then combine that with the lines I found for my collision detection. If I looked only at the side of A that B's protruding corner is closest to, then there could be a problem if the corner went past the middle of A or if it were especially close to one of A's corners; the result could be ambiguous.

There are currently three ways to create objects - two for circles and one for rectangles. I tried to come up with as intuitive a way as possible so as to make working with my program easy and fluid. One way of inputting circles is to click where the center will be and drag to create a radius. The other way is to click an edge and drag the diameter. I implemented both because I think that the second is easier, but I have seen the first used before. It was harder to figure out a way to create rectangles because there are more variables than with circles. To create a rectangle, one clicks where a corner will be, then clicks again for another corner and drags to finish the rectangle. After creating a shape, a dialog box appears to ask for the velocity and color of the object. I would like to create something that does not interrupt the flow as much, but I am not sure how to do so.

Input method selection is part of the GUI. The GUI is manifested in a menu bar with the ubiquitous File, Edit, and Help menus, and two rows of buttons along the bottom. Although File, Edit, and Help are not the most descriptive names for menus, considering what my project does, I chose them because psychologically it would probably be more difficult or distressing for a user to have unfamiliar menus. They contain exit, reset, help, and about commands. The rows of buttons along the bottom are speed controls and input controls. The speed controls are fast rewind, rewind, pause, play, and fast forward. They work except when collisions are involved, but I can fix this by reversing parts of the equations. The input method controls are the three I already mentioned, and will eventually include anchors, pins, axles, and other polygons.

I have made a class called InputMethod to facilitate changing of input methods. InputMethod is an abstract class which implements the MouseListener and MouseMotionListener classes. Whenever the buttons controlling input are pressed, the PhysSim class removes the current InputMethod and adds the new one to itself as a MouseListener and a MouseMotionListener.

Work on anchors has been started. The specific purpose of an anchor is to lock an object to the background, enabling it to act as another wall with which objects can interact. Currently, objects can be manually anchored in the setup phase of the program, and they will act as they are supposed to. It should not be hard to create the input method class for anchoring objects because it will consist of checking if the mouse is over an object, an extension of collision detection, and if it is clicked, toggling a boolean value held by the object.

4 Preliminary Results

My program accurately represents projectile motion and collisions with walls without regard to friction, and with an elasticity of one. While running, it may seem that it is not accurate, but that is because people are judging it with respect to their experiences, which take place in the real world, which has many more forces than my program currently simulates. Rudimentary collisions between moving objects can

be seen, but are not completed. In addition, anchors are almost complete. Once I implement friction and find a good way to determine the elasticity value for each collision, my simulations will seem much more realistic.

References

[1] Neumann, Eric. "Rigid Body Collisions." 2004. <http://www.myphysicslab.com/collision.html>