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

Tom Smilack

June 16, 2008

Abstract

Physics simulations are often of single concepts or immune to user control. My project aimed to change that by allowing users to create a situation and then simulating the behavior of objects in that situation. Users can create objects rectangles and circles of any size through intuitive shape tools, then the program converts them to polymorphic objects and runs the simulation. The speed of the simulation can be controlled and users can add more shapes while it is running.

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 gave 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 objects. The objects are rectangles and circles of any size. I intended to add support for complex objects (simple objects connected by pins or axles), but other difficulties prevented this. Interactions include collisions, and my original plan was to implement friction and rolling, but those also were prevented by other problems. 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 Development

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 timer is started by a driver file which creates the window, adds the simulation panel to it, and builds a menu. The menu contains a few commands such as reset and exit. 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 Rectangle and Circle.

Circles are easy to draw, as they look the same no matter how they are rotated; one only needs to know the position and radius. 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 would have been similar in implementation to the rectangle, as I treat it more like a set of points than as a rectangle. In addition, the Graphics method fillPolygon is used to display it.

Complex shapes were to be implemented using pins and axles. Pins would connect two shapes so that they could stay together in the same position. To achieve this I intended to create a ComplexObject class that would have contained a list of shapes that combined to form it. It would have calculated collisions for every object in it and applied forces to each object so that they

would move in unison. Axles would be more complicated; I would have had 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[1]:

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

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

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

When two objects collide, there are two more equations, and the final one changes [1]:

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

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

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

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 was not sure how exactly to find it. I thought that I could look for what side of A the majority of B was on and then combine that with the lines I found for my collision detection. However, if I looked only at the side of A that B's protruding corner was 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.

I believe that part of my problem was that I vastly overestimated the effectiveness of current collision detection technology; I had assumed that it

was nearly perfect but I realized from reading an article about continuous and discrete collision detection that that is not the case. In addition, I underestimated the problems I would face in attempting to implement it. Estimating the time that it will take to complete something is one of the most important aspects of an extended project and I did not guess well with respect to collisions. When I finally finished them I thought I would be able to continue, but the problems with the normal line prevented that. Under fewer time restraints I probably would have been able to complete collisions but I spent too much time on detection.

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. (See Fig. 1) 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.

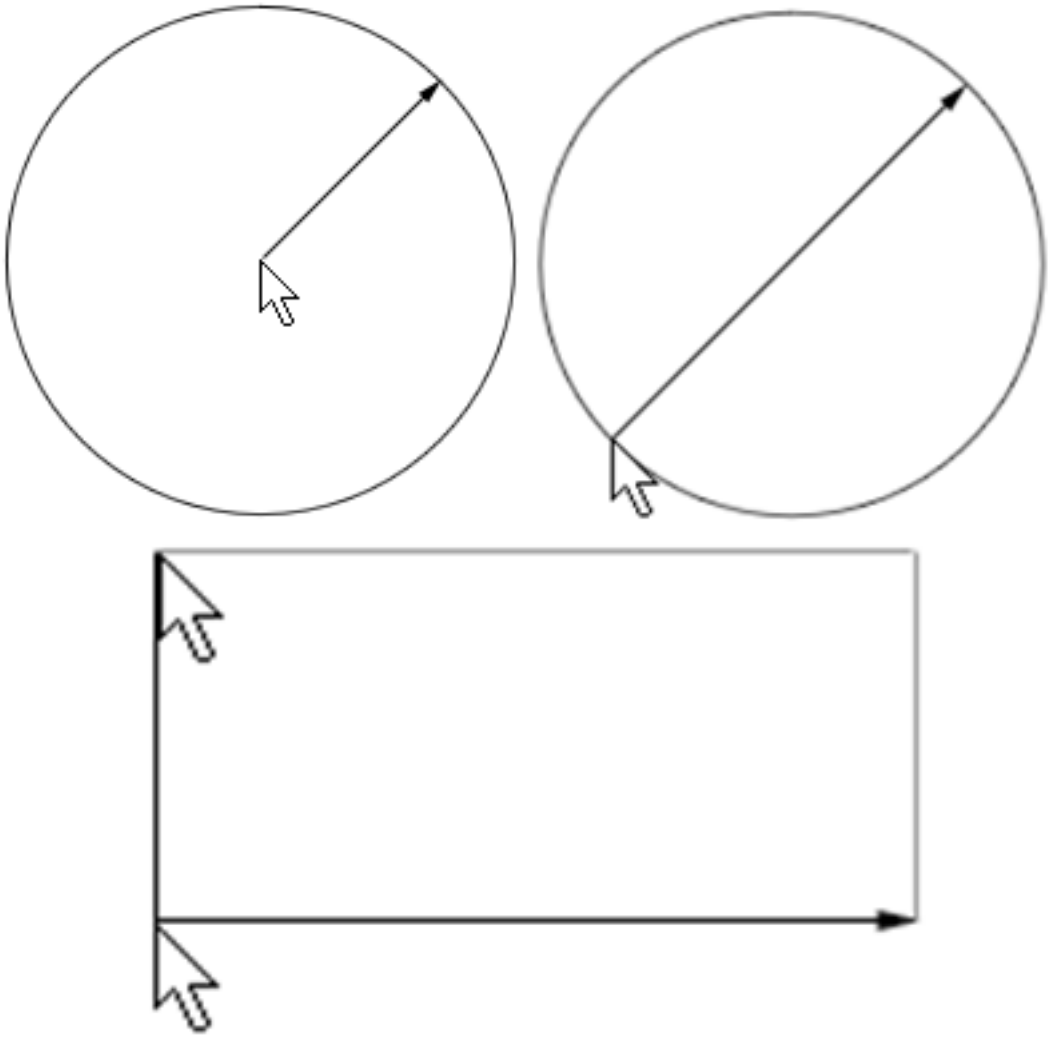

Figure 1: Three input methods. An image of a cursor represents a click and a line with an arrow represents a drag.

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.

Input methods are one major example of polymorphism in my project, the other being shapes. The shapes are descended from the SimObject class which defines getter and setter methods as well as the abstract methods draw, checkCollision and checkWallCollision. The subclasses, Rectangle and Circle, define these in different ways. The InputMethod class I designed to facilitate changing of input methods. InputMethod is an abstract class which implements the MouseListener and MouseMotionListener classes and overrides all of their abstract methods with blank methods. The different subclasses of InputMethod override only the methods that they need in order to work, as well as the abstract method draw which shows and progress on the shape being inputted, if it has not yet been completed. 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.

Anchors for the most part work well. 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. Objects can be manually anchored in the setup phase of the program, when the initial data is inputted, and they will act as they should. I created another InputMethod class for the anchor creation button. Rather than override several mouse methods it only uses mousePressed and borrows collision detection from the Rectangle class in order to determine if an object was clicked. After clicking, it cycles through every object and if the mouse is positioned over an object of class Rectangle, the method setFixed is called to stick it to the background. When an object is anchored, an image of an anchor appears over it and it does not move. However, there is a glitch such that when an object is made anchored in the

setup phase, it will not respond to changes made by the user. In every other respect they work, though. (See Fig. 2)

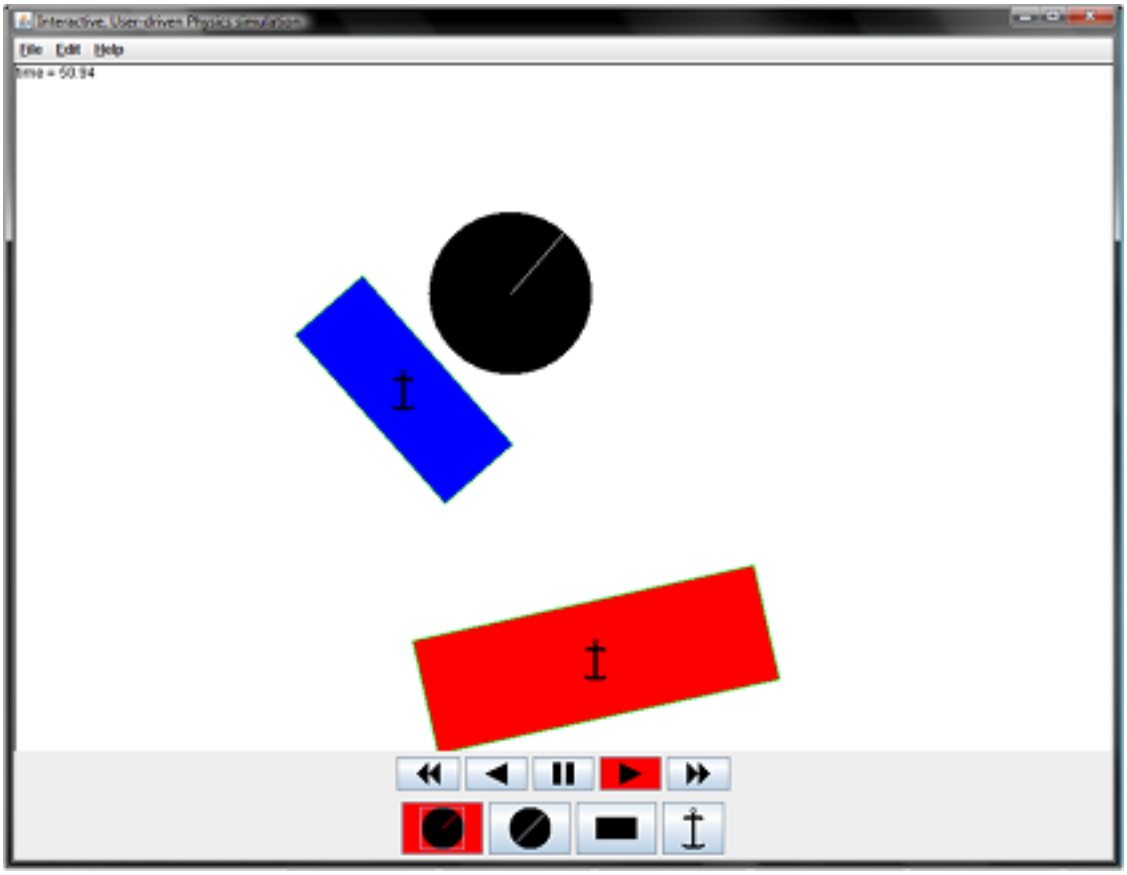

Figure 2: Two anchored rectangles and one free-falling circle.

4 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 particular, there is a glitch where objects will be able to move within the bounds of other objects, even after they have responded to a collision. The collisions also only work perfectly on one side, as well, because of the difficulty I had in determining the direction of the normal line from the collision point. Anchors serve their purpose, but the other problems combine to outweigh that. If my program had an accurate way of determining elasticity and friction, then it would seem much more realistic.

References

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

[2] Scheintaub, Hal. "Modeling modern methods in high school physics classes." 2006. <http://portal.acm.org/citation.cfm?id=1150034.1150210&coll=Portal&dl=ACM&CFID=62134220&CFTOKEN=55300109>

[3] Zhang, Xinyu et. al. "Continuous collision detection for articulated models using Taylor models and temporal culling." 2007. <http://portal.acm.org/citation.cfm?id=1275808.1276396&coll=Portal&dl=ACM&CFID=62134220&CFTOKEN=55300109>

[4] Lander, Jeff. "Trials and Tribulations of Tribology." 2000. <http://www.gamedevelopment.com/features/20000510/lander_pfv.htm>

[5] Yeh, Thomas et. al. "Enabling real-time physics simulation in future interactive entertainment." 2006. <http://portal.acm.org/citation.cfm?id=1183316.1183326&coll=Portal&dl=ACM&CFID=62134220&CFTOKEN=55300109>

[6] Boeing, Adrian et. al. "Evaluation of real-time physics simulation systems." 2007. <http://portal.acm.org/citation.cfm?id=1321261.1321312&coll=Portal&dl=ACM&CFID=62134220&CFTOKEN=55300109>