CMPS 101 - Programming assignment 2 - W13
Keeping Track of Elastic Balls with a Heap

Handed out Th 1-29-13 Due We 2-8-13 in locker

January 29, 2013

In this program you are to make a simulation of colliding elastic balls that bounce into each other. We will provide a JAVA program in

"http://classes.soe.ucsc.edu/cmps101/Winter13/hw/prog2/"

that has an $O(n^2)$ iteration time. You need to bring it down to $O(n \log n)$ using a smarter data structure (a heap).

Here n is the number of objects (balls + wall segments). For each pair $(\tilde{i}, \tilde{j})$, $1 \leq \tilde{i} \leq \tilde{j} \leq n$, the program keeps track of the collision time when object $\tilde{i}$ and $\tilde{j}$ collide. It then iteratively

1. **finds the pair (i, j) with the earliest collision time,**
2. simulates the graphics until that point,
3. updates the direction and velocities of the i and j using the laws of physics (walls stay put),
4. updates all collisions times involving i and j,
5. **update the data structure helping to compute the next** minimum.

Essentially, you are to modify the **last** and the **first** step. The implementation provided keeps track of the collision times in an array (upper triangle since $i < j$). For each row of the array, we keep track of the minimum and the minimum of the row minima is the global minimum.

You are to keep track of the collision times in a heap. So the entries (i, j) of the triangular array point into the heap. Now the collision times are updated using ChangeKey operations and the min collision time is found using the Minimum operation.

Your task:

1. Add the heap data structure.
2. Write the ChangeKey operation and modify the program.
3. Reason at a high level that the old program is $O(n^2)$ per iteration (worst case) and the your modification is $O(n \log n)$ (worst case).

Tricks:

1. Download the program, go through the code and run it with the simple example inputs provided.
2. Add a printout procedure for your heap.
3. First write the ChangeKey operation and check it.
4. Finally modify the program to work with your heap code. First make sure the program works with the simple examples we provide. We will provide u with more complex examples to work with.

Extra Credit:

- Run your new version against the one we provided on some larger problems and compare their running times on some interesting configurations.

1 Overview of the Code

There are four main classes in the code.

- Vector This class provides many operations needed for handling vectors. The code has been tuned to handle two dimensional vectors as those are the ones we need to handle in this assignment. The vectors we mostly use are for ball positions and ball velocities. The operations (e.g. multiply, dot product etc) are pretty straight forward.
- Wall This class provides abstraction for handling wall objects. We need to know where the walls are and how to handle collisions with balls. A wall is represented by a point on it and a normal vector. We simplify by assuming the locations of the walls (the four walls in the simulation). The most important operation in this class is computeCollisions(), which computes when a ball collides with the wall.

- Ball This class provides abstraction for handling ball object. A ball has radius, mass (proportional to square of radius), a position and a velocity. Other than usual set/get methods, the crucial operation is the computeCollisions() which computes when a collision occurs with another ball. Since the balls can have different mass, the computations are little more complex (law of conservation of momentum and kinetic energy come into play. The wikipedia page (http://en.wikipedia.org/wiki/Linear_momentum#Application_to_collisions) has some good information on these computations).

- CollidingBalls This class handle the GUI code and the actual simulation part.

  The main data structures in this class are

  - globalTime a counter that is updated in the main loop and acts as global time.
  - balls[] Keeps track of all the balls.
  - walls[] Keeps track of all the walls. There are just 4 of them for now, but anyways.
  - collisionTable[][] The main data structure that keeps track of the collisions between all objects. We only use the upper traingle of the 2d array as this is a symmetric matrix, i.e., collision time for ball i and ball j stored at collisionTable[i][j] is same as collision time for ball j and ball i at collisionTable[j][i]. This data structure is used along with rowMinimum[] array to efficiently compute the minimum collision time. The updates to this array occur primarily in computeCollisions().
  - rowMinimum[] keeps track of the minimum entry for ith row in collisionTable[][]. (This is not same as keeping track of minimum time for ith object).

  main functions in this class are

  - main() routine initializes the code, sets up GUI, updates the collisionTable[][] and calls computeCollisionTImes() to find the first collision event. The main loop sleeps for a bit and then keeps calling repaint(), which is where most of the work is done. Note that repaint() is part of the GUI event management.
  - doCollisionwithBall(), doCollisionwithWall() These routines handle the physics part of collisions.
  - DetectCollision() A simple loop which compares the current time with the time of next collision and when ready, calls the collision handler code.

- HandleCollisions() Handles the collision event by calling the collision handling code for the objects involved. And then calls the computeCollisions() to update the future collisions for the objects that just collided. Once all events are handled, call computeCollisionTImes() to find the time for next collision.
- computeCollisions()
  This routine computes all possible collisions for the input ball. This simply involves calling the computeCollisions() method on various objects and recording the return value in the collisionTable[][] array. Simultaneously the rowMinimum[] array is updated too. Because the collisionTable[][] array is maintained as a upper triangular matrix, the method to update this array is a bit complex. For ball i, we update all the row entries i+1 to max. We also update the ith column of all rows j, such that $j < i$. While doing this we also need to update the rowMinimum[] and because of this, the method has worst case of $O(n^2)$.
- computeCollisionTimes()
  In this method, we simply go through the rowMinimum[] array and find the minimum entry. If there are many entries with min value, we find all of them.

The input format is pretty simple and is described in the header comments for readData() method.