COMS W4130 Project Milestone #3

Varun Ravishankar, vr2263 and Nathaniel Clinger, nc2411

11/18/11

This milestone consisted of parallelizing our existing animation code. The animation itself simulates particles colliding and outputs each particle's velocity and position at every time step during the duration of the simulation. The bulk of the project lies in simulating collisions reliably and efficiently for large numbers of particles. Our sequential code detected collisions not by looking for intersecting particles throughout the simulation, but by instead applying a slight repulsive force that is strongest when the distance between the particles is less than the sum of their radii and diminishes as the particles grow further and further apart.

1 Determining potential speedups in the sequential code

We first determined spots where our algorithm itself could be parallelized. Unfortunately, since the code itself simply used a force to resolve collisions and the code has to be run in a specific order for the integrator to simulate the particles efficiently, we had to determine a better algorithm that could be more easily parallelized. Other areas of potential speedup included optimizing the matrix operations we were performing, such as multiply matrices or computing the transpose or inverse, and optimizing reading a file in. Unfortunately, while disk I/O might take a while, it would be not create a significant speedup in comparison to the time it would take to compute the simulation, even assuming the file was 10,000 lines long. Optimizing the matrix operations could also lead to some performance improvements, but most of the operations we were performing were on vectors of length 2 or matrices of size 2x2. Our best bet then, would be to find a better way of doing collision detection.

2 Hash table-based Collision Detection

To improve the collision detection, we developed a hash table-based implementation with a hash table of size n, where n is the number of particles. Each particle gets hashed based upon its position in the x and y directions, and then gets added to the hash table. The code then iterates through the hash table and finds collisions in the hash table; these are particles that hashed to the same cell and so have a high probability of being close to each other, assuming our hash function is good and does not generate many collisions for particles that are far apart. The code then looks through these colliding particles and generates pairs from them, which in turn get added to a set storing the collisions for each particle. The replusive penalty force is then applied to each pair of colliding particles, as opposed to checking every pair of particles to see if it is colliding or simply applying the force in a matrix operation. To parallelize the code, we then determined opportunities where an individual thread could add particles to the hash table, or could resolve colliding particles in a single cell in the hash table.

3 Safety and Determinacy

Our code is safe and determinate. Each thread operates on the same hash table, but is operating on a different cell in the hash table and thus does not affect other cells. When we then resolve the collisions in the hash table and determine the actual particle collisions, the threads simply append to a set, which is a safe operation. Since we do not alter any shared mutable variables, the code is safe and since it leaves the heap in the same state after the collision detection is completed (all colliding pairs are in the set of particle-particle collisions and no outside variables like positions or velocities are changed), the code is determinate as well.