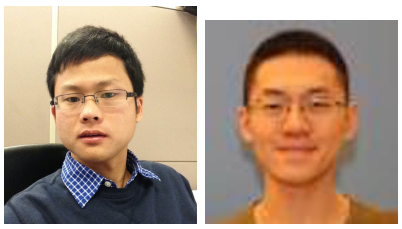
Implementing A Particle Collision Model:
A Report for CS 6491 Fall 2015 Project 1

Xiong Ding, Dingtian Zhang

Sep 10, 2015



1 Objective

To simulate the precise motion of a large number of balls in a cube in computer graphics.

2 Input

We are given the intial prositions and velocities of n balls. We have the following assumptions: All balls are in the cube, have the same radius and mass, and are pairwise disjoint. Balls starts in constant velocity motion. All collisions are elastic.

3 Implementation

3.1 Elastic collision law

Elastic collision preserves the total energy and momentum. So for collisions between ball and wall

$$V'_{\perp} = -V_{\perp}\,, \quad V'_{\parallel} = V_{\parallel} \tag{1}$$

and collision between ball and ball

$$V'_{1,\perp} = V_{1,\perp}\,, \quad V'_{2,\perp} = V_{2,\perp} \tag{2}$$

$$V'_{1,\parallel} = V_{2,\parallel}\,, \quad V'_{2,\parallel} = V_{1,\parallel} \tag{3}$$

Here, subscript $\perp$ denotes the direction perpendicular to the center-center line, and $\parallel$ denotes the direction along the center-center line.

3.2 Key Classes and Methods

We implement new classes and methods for *balls* class, some of which are:

3.2.1 class Colli

A class used to store collision object and time. It has two variables: p denotes the index of ball to collide, and t is the collision time.

3.2.2 Colli[] colli

An array to store next collisions of all balls with balls or walls (where walls are denoted by negative integers).

3.2.3 void initPointsOnGrid()

A modified version of ball collisions to help debugging.

3.2.4 float calB2BTime()

Calculate the collision time between two balls. If two balls cannot collide in positive finite time, the return -1. We solved the quadratic equation with regard to P_0, P_1, V_0, V_1, r, t to get the minimal collision time.

3.2.5 float calB2WTime()

Calculate the collision time between one ball and one wall. If they cannot collide in positive finite time, then return -1. We solve linear equation with regard to P_0, V_0, r, R, t to get the minimal collision time.

3.2.6 Colli calOneBallColli()

Calculate the smallest collision time between ball i and other balls and walls. This function checks all possibilities between one ball and all the other balls, and all six walls. A minimum collision time is maintained along with the next collision object (a ball or a wall). The complexity of this function is O(n), where n is the number of balls.

3.2.7 void initCollision()

Initialize the collision array by calculating next collision of all balls. This complexity is $O(n^2)$. Luckily we only need to do it once in initialization phase.

3.2.8 void bounceTwoBalls()

Solve two balls' velocities after collision. Normal compatible components (along the line conneting these two balls) will exchange.

3.2.9 void updateB2BColli()

We calculate the velocities of two balls after collisions, then update the next collisions of these two balls. We also need to update balls that "WERE going to collide into these two balls", and balls that "ARE supposed to collide into these two balls". All the rest do not need to recalculate. In the first case, since the initial velocities of all balls are randomized, the expected number of collisions for a ball with other balls is nP, n is the number of balls, and P is the probability of collision, which is proportional to r, where r is inversely proportional to n. Therefore, the number of collision is O(1). In the second case, it is clearly no more than two. Therefore, the complexity is $O(n)$.

3.2.10 void updateB2WColli()

We calculate the velocities of the ball after collision with a wall, then update the next collisions of this ball. We also need to update balls that "WERE going to collide into this ball", and the ball that "IS supposed to collide into this ball". All the rest do not need to recalculate. The complexity is $O(n)$.

3.2.11 void updateColli()

The main function to update the state whenever collision is about to happen. Update collision array using either one of the update functions.

3.3 General Picture

3.3.1 Initialization Phase: Before the graphic process begins

After adding balls to the scene, we call $initCollisions()$ to initialize collision array $colli[]$.

3.3.2 Update Phase: At each frame of the graphic process

We call updateColli() to update collision array if next collision is about to happen.

4 Complexity/Performance

$O(n^2)$ for pairwise comparison in the initialization phase, and $O(n)$ in the update phase.

5 Experimental results and analysis

5.1 Numerical Setup

`nbs`* `nbs`* `nbs` balls are initialized in a cubic frame with each side has `nbs` balls. Simulations are conducted with increasing `nbs` until the simulation time in each frame exceeds a single frame time (1/20 second). Sphere detail is set to 6 and sphere radius is given by formula

$$r = \frac{w}{(30\pi)^{1/3} * nbs}$$

All experimental data is taken every 5 frames with 50 samples for each `nbs`.

5.2 Maximal `nbs`

Fig 1 shows the average time for 4 different stages, among which, t_{12} is the time used to update the collision information (physical process) and t_{34} is the time to draw all balls (graphical process). As we can see, **the maximal `nbs` that our simulation can sustain is 11**. At this `nbs`, $t_{01} \simeq 23\%$ and $t_{34} \simeq 18\%$.

Why the computation time is super linear ? As we can see in Fig 1, t_{01} is not linear with respect to the number of balls (`nbs` * nbs * `nbs`). However, this phenomenon does not contradict our claim that the updating complexity is $O(n)$, because, with increasing `nbs`, the number of collisions per frame increases. Therefore, the work needed to update collision information in each frame is super linear.

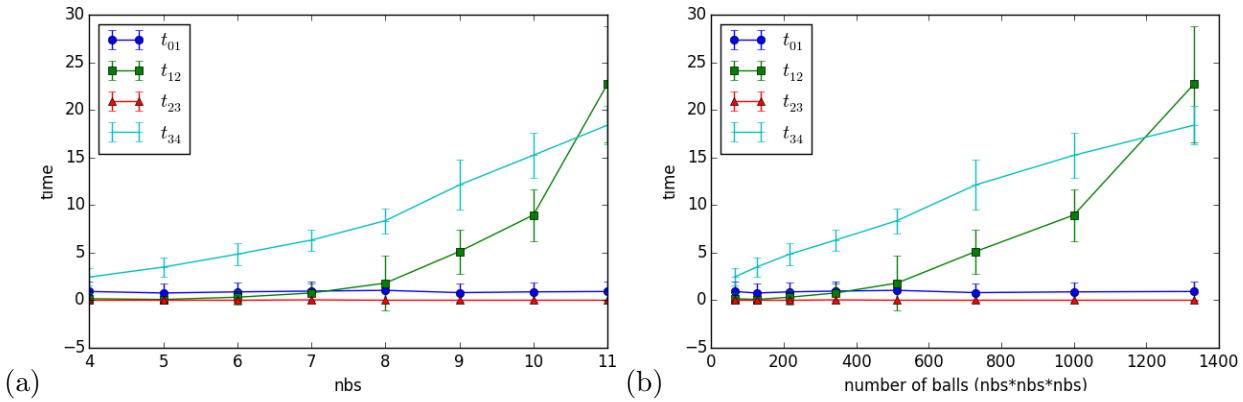


Figure 1: t_{01}, t_{12}, t_{23} and t_{34} with respect to `nbs`(a) and the number of balls (b) respectively.

5.3 Number of collisions per frame

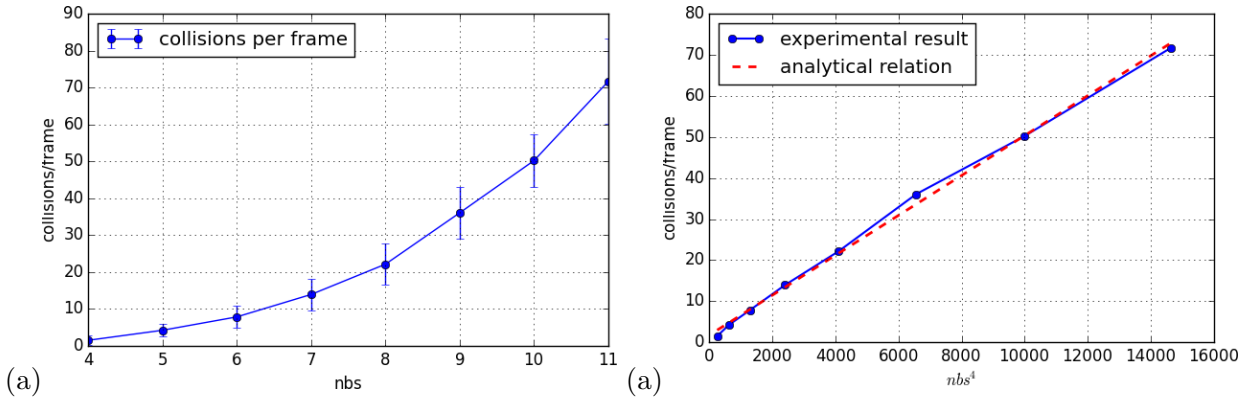


Figure 2: (a) Averaged number of collisions per frame for different `nbs`. (b) number of collision per frame vs `nbs`^{4}. The blue curve is experimental result. Red dashed line is the analytical relation formula (5) : $y = 0.00486\,x$.

First, we give a analytical formula for this relation. First, the average free flight time is estimated

as

$$t \propto \frac{w}{\texttt{nbs} * v}\,, \tag{4}$$

where, w is the cube length and v is the average velocity. Second, during this period, the number of collisions happened should be proportional to the number of balls, which is $\texttt{nbs}^3$. So, the number of collisions per frame is

$$\#collsions \propto \frac{v}{wf} * \texttt{nbs}^4 \tag{5}$$

Here, f is the frame rate. **Basically, the collision number per frame is proportional to $\texttt{nbs}^4$.**

Fig 2(a) shows relation between number of collisions per frame versus nbswith error bars. Panel (b) compares the experimental data with an analytical relation $y = 0.00486\,x$ (Note, the x axis is $\texttt{nbs}^4$ in (b)). We see that these two match every well.

6 Conclusion

We have simulated the collective motion of a large number of elastic balls in a cubic container. The maximal nbs that our simulation can support is 11. Also, we found the number of collisions per frame is proportional to $\texttt{nbs}^4$ both analytically and experimentally.