Cluster Comput (2017) 20:2591–2603
DOI 10.1007/s10586-017-0741-7

CrossMark

Parallel GPU-based collision detection of irregular vessel wall for massive particles

Binbin Yong[1] · Jun Shen[2] · Hongyu Sun[3] · Huaming Chen[2] · Qingguo Zhou[1]

Received: 11 September 2016 / Accepted: 5 January 2017 / Published online: 18 February 2017
© Springer Science+Business Media New York 2017

Abstract In this paper, we present a novel GPU-based limit space decomposition collision detection algorithm (LSDCD) for performing collision detection between a massive number of particles and irregular objects, which is used in the design of the Accelerator Driven Sub-Critical (ADS) system. Test results indicate that, the collisions between ten million particles and the vessel can be detected on a general personal computer in only 0.5 s per frame. With this algorithm, the collision detection of maximum sixty million particles are calculated in 3.488030 s. Experiment results show that our algorithm is promising for fast collision detection.

Keywords GPU-based · collision detection · irregular objects · space decomposition

✉ Qingguo Zhou
zhouqg@lzu.edu.cn

Binbin Yong
yongbb14@lzu.edu.cn

Jun Shen
jshen@uow.edu.au

Hongyu Sun
sunhy13@lzu.edu.cn

Huaming Chen
hc007@uowmail.edu.au

1 School of Information Science and Engineering, Lanzhou University, Lanzhou, China

2 School of Computing and Information Technology, University of Wollongong, Wollongong, Australia

3 Computer Science, Swinburne University of Technology, Melbourne, Australia

1 Introduction

Nowadays, nuclear power, as the primary peaceful utilization of nuclear energy, has been developing at a fast pace. Meanwhile, the quantity of nuclear waste also increases rapidly in some countries. In order to recycle and store the nuclear waste, the nuclear power industry strives to develop Accelerator Driven Sub-Critical (ADS) system to recycle nuclear waste.

In the research of reprocessing of nuclear waste, spallation target is an important part of ADS system. Using GPU (graphics processing units) technology to concurrently simulate the physical process in the spallation target is an effective way to help with development of the relevant spallation target. In particular, the simulation of collision between various particles and different objects is very important.

In our paper, we focus on the collision detection between same frames. A frame is the essential and smallest part of the simulation. The process of collision detection in a frame consists of two parts. The first part is the collisions between accelerated particles, which has been discussed sufficiently [1]. Hence, instead of the first part, we concentrate on the second part: the collisions between particles and the irregular objects, on which there is a scarcity of studies. The simulation is shown in Fig. 1, which is used in our project ("Strategic Technology Pilot Project").

Furthermore, as a limitation of the computing power of computers, traditional sequential algorithms could only simulate tens of thousands of particles. The reason is that the consumption of computing resources shows a linear relationship to the number of particles with sequential programs. Meanwhile, GPU is nowadays being widely used in simulation due to the continuous development of parallel simulation techniques. Therefore, it is natural to use GPU parallel computing technology in our issue. In this paper, we

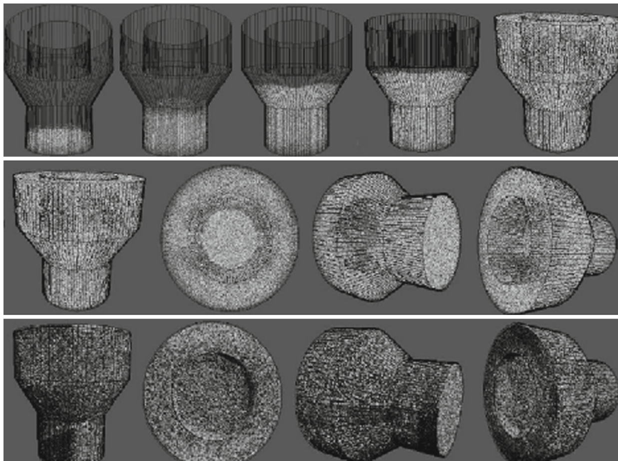


Fig. 1 The simulation of collision detection between particles and irregular models

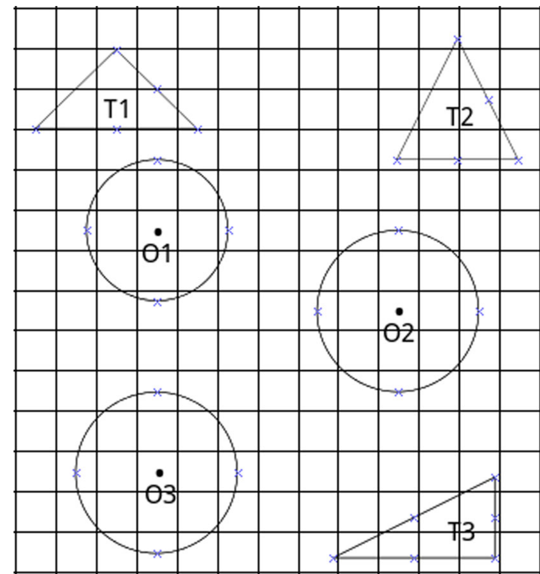


Fig. 2 The collision detection in three dimensional coordinate space. The circles represent the particles and the triangles stand for triangle meshes of the vessel wall

implement the Limit Space Decomposition Collision Detection (LSDCD) algorithm between the particles and irregular objects by dividing the collision space into infinitely small uniform grids and making full use of the parallel processing ability of GPU.

In the preprocessing stage, we calculate the nearest triangle meshes to the grid and sort the triangle meshes by distance to the grid in ascending order. The particles are assigned into these grids through the position of particle center. When the length of grid is tending to be infinitely small, the distance from particle center to the wall would be infinitely close to the distance from the grid to the wall. With the space data we get in the preprocessing stage, we are able to make collision detection easily. Consequently, experimental results demonstrate that our parallel algorithm is effective in accelerating the simulation process.

We list some terminologies used in this paper here. The term "irregular vessel wall" is the container used to store the particles to be accelerated and collided, which is equivalent to "irregular object", "object", "wall", and "model". Generally, a spatial model is often approximated as many triangle meshes. Hence, the wall model is made up of many triangle meshes. The triangle mesh has a collision area which is different from a general meaning of triangle consists of three lines. The particles are modeled as very small spheres whose radii are not the same. The collision space is a cube, whose center is located in the coordinate origin of the three-dimensional space. In general, the particles and the walls are just wrapped by the cube to reduce unnecessary computation. In the implementation of our algorithm, we split the cube into a number of uniform grids, as shown in Fig. 2. After defining these terminologies, the problem can be described as: perform collision detections between a larger number of spheres (particles) and triangles (walls).

The rest of the paper is organised as follows. In the next section, we introduce the basic concepts and related work. In Sect. 3, we present our design and implementation of the particles-irregular wall collision detection algorithm. The results of experiments are illustrated in Sect. 4. Meanwhile, the analysis of the effects of different parameters are discussed here. At last in Sect. 5, we conclude our work and address the future work.

2 Preliminary and related work

In general, to detect collision between a sphere and the wall, we need to calculate the shortest distance from the sphere center to the wall and compare it to the radius of the sphere. In most cases, we can get the distance by mathematical techniques directly. For example, to get the distance between a sphere and a plane wall, the distance equation from dot to plane is used.

In this paper, we mainly focus on the collision detection between irregular wall and particles. Without loss of generality, particles are considered as spheres. In computer graphics field, a model is represented by many triangle meshes. We represent the 3D irregular wall model by STL (STereo Lithograph) file format, which is developed by 3D SYSTEM company in 1988 [2]. The basic collision detection is based on space partition and the algorithm is implemented on GPU.

2.1 Collision detection

The technology of collision detection, which could study the problem of whether two or more objects would collide or

not, when and where the collisions would occur in the virtual scene, has been widely used in the field of computer games, physical simulation, virtual technology, computing and animation. In the past few decades, many algorithms have been proposed to tackle the challenge of collision detection. Many of them have been proved to be efficient. In general, there are two type of collision detection algorithms, spatial subdivision and bounding box. The bounding box method is more widely used than spatial subdivision method. The most widely used boxes are Axis-Aligned Bounding Boxes (AABB) [3], Sphere, Oriented Bounding Box (OBB) [4].

Recently, the algorithms based on bounding volume hierarchy (BVH) become widely used. This algorithm recursively constructs bounding volumes and treats each object as a leaf node. It makes self-collision detection within the same BVH to detect intersecting objects [5–7].

However, these algorithms mainly focus on the detection between a small number of objects [8], and it is not suitable for fast collision detection that contains massive objects, especially massive particles [9].

In the field of particle physics and nuclear technology, especially for spallation target in ADS system, the simulation is mainly related to the collision between particles and irregular wall. There are extensive researches on collisions among particles, such as uniform grid spatial subdivision [1]. Based on GPU technology, Xiong et al. [10] proposed an efficient implementation of three-dimensional gassolid DNS (Direct Numerical Simulation). In this paper, we implement the collision detection between particle and irregular wall based on GPU parallel technology. It should be noted that, while this algorithm is very efficient on GPU, it is very inefficient without the support of GPU.

2.2 GPU and CUDA

In recent years, big data computing is developing towards the CPU and GPU co-processing. In 2006, Nvidia introduced CUDA (Compute Unified Device Architecture), which has since become a general-purpose embedded GPU parallel computing platform [11]. CUDA supports C, C++, Fortran, and Python programing languages. A GPU with CUDA support contains a number of SMXs (Streaming Multiprocessors). The kernel function of the CUDA code is executed on these SMXs and the serial code is executed on CPU. When executing code on GPU, some blocks are allocated, each block contains a large number of threads, which can be executed in parallel. Hence, for some time-consuming tasks, like loop statement, it is able to reduce the overall run time through parallel programming. In general, CPU is more capable of data caching and flow control, while the GPU is specialized for highly parallel computation and data processing.

2.3 Related work

General Purpose GPU (GPGPU) is a relatively new research area. However, the idea of using GPU in the field of collision detection has been researched longer than the emergence of GPGPU. Purcell et al. [12] had researched light and triangle intersection test algorithm in ray tracing technology based on GPU. Baciu et al. [13] and Myszkowski et al. [14] researched convex body collision detection in the early days. They regarded pixels in each render buffer as a beam of light which was perpendicular to the visual plane and tested intersection between the light and objects. Govindaraju et al. [15] presented an algorithm for collision detection between multiple deformable objects in a large environment using GPU. Kipfer et al. [16] presented a particle system engine for real-time animation and rendering. The system rendered large particle sets using GPU and it implemented inter-particle collisions and visibility sorting. Zheng et al. [17] showed a contact detection algorithm based on GPU and they used the uniform grid method in detection. Based on the vector relation of point, line segment and rectangle, Shen et al. [18] implemented a rapid collision detection algorithm.

There are some other algorithms to optimize the computation of collision detection. Li and Suo [19] researched the application of particle swarm optimization in randomly collision detection algorithm and increased real-time capacity, compared to the classic Oriented Bounding Box (OBB) bounding box algorithm. Similarly, Qu et al. used parallel ant colony optimization algorithm in randomly collision detection algorithm to improve the real-time characteristic and precision in collision detection [20]. With spatial projection transformation method, Li and Tao mapped irregular objects from three dimensional space to regular two-dimensional objects to carry on collision detection [21]. Based on MPI (Message Passing Interface) and spatial subdivision algorithm, Huiyan et al. researched an advanced algorithm to improve the performance and accuracy of collision detection [22]. Tang et al. [23] proposed a GPU-based streaming algorithm to perform collision queries between deformable models by using hierarchical culling. Zhang et al. presented a parallel collision detection algorithm with many-core computation by CPUs or GPUs [24]. Wang et al. proposed an image-based optimization algorithm for collision detection [25].

Green et al. implemented a sample program about the simulation of a particle system documented in the white paper [1]. It uses a cube as the wall and concentrates on the parallelization of collision detection between particles. Our research differs from [1] and our focus is on the collision detection algorithm between particles and wall. Xu et al. [26] designed a G-Octree based fast collision detection for large-scale particle systems. Zou et al. [27] designed a collision detection algorithm based on GPGPU. Fan et al. [28]

explored the collision between two objects by finding intersections between a collection of line segments and a set of triangles. Although many of these collision detection algorithms based on GPU have been researched, the research on the collision between irregular walls and massive number of small objects (particles) is rarely discussed so far. Hence, in this paper, we design the LSDCD algorithm for this kind of scenario, and apply it to the collision detection between rigid bodies.

3 Collision detection algorithm

We will describe our parallel LSDCD algorithm and present the detailed steps of the algorithm in this section. It contains the data structure subsection, limit space decomposition and preprocessing subsection, collision detection and accuracy verification subsection, N-triangles method subsection and the extension subsection.

3.1 Overview

The basic idea of the algorithm is to divide the collision space into infinitely small uniform grids. The reason to use uniform grid is because it is simple and similar for each grid, hence, it is suitable for GPU implementation [9]. As shown in Fig. 2, for each grid, if the grid length is small enough, the grid will act as a point and the points in the grid are similar in distance feature to the triangles. For example, for all points in grid $O1$, the nearest triangle mesh is triangle $T1$, this is decided by their spatial features. Hence, when we calculate the shortest distance from a point to the wall, we can firstly get the triangle mesh that is closest to the grid. Then we can calculate the distance from the point to the triangle directly.

For a large number of particles, the collision detection is independent for each particle and it can be implemented in parallel based on GPU. The basic process of the collision algorithm is as follows: firstly, we get the position of these triangles from the model file (STL format). Secondly, the large number of particles are initialized. The next phase is limit space decomposition. We use a large quantity of small uniform grids to divide the collision space. In this phase, we should make sure that the length of grid is as small as possible to get a high-precision, and this is different from the traditional uniform grid. In reality, as we can see from the latter experimental part, the correct rate of collision detection is relevant to the grid length. Next we calculate the nearest triangles to each grid in parallel and store the information in array (named grid array). Finally, the collision for each particle are detected in threads on GPU-based code. This step is achieved by comparing the distance between particle and triangle to the radius of each particle.

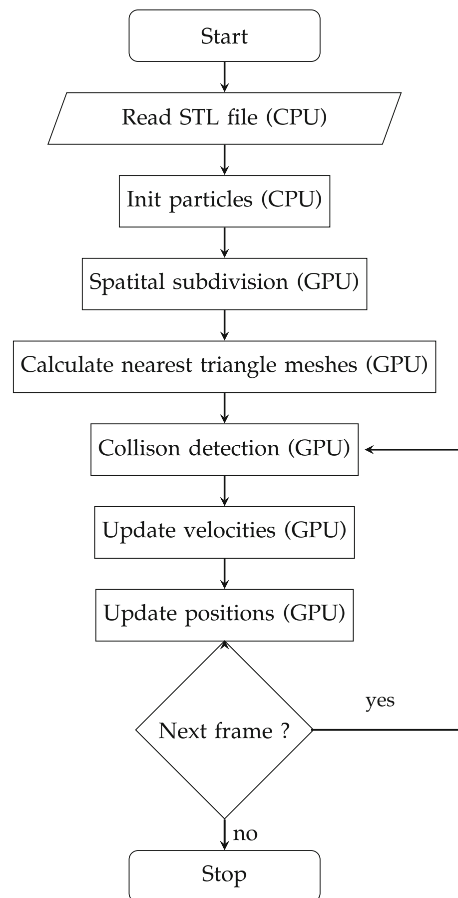


Fig. 3 The flow chart of simulation of collision based on LSDCD

The flow chart of the algorithm is shown in Fig. 3. In the whole process, the steps from 'Read STL file' to 'Init particles' is the initialization phase, which is executed on CPU only. The steps from 'Spatial subdivision' to 'Calculate nearest triangle meshes' is the preprocessing stage, which is executed mainly on GPU. And the steps between 'Collision detection' and 'Update positions' is the collision stage for a frame executed on GPU. For initialization phase, it is similar and simple for different computing platforms. Hence, we focus on the preprocessing stage and the collision stage, which are executed on GPU.

In the preprocessing stage, for each triangle we start up a thread on GPU, to find the several grids whose center are nearest to the triangle and write the grid numbers to array. In fact, we save the array to disk to execute preprocessing stage faster next time. In the collision stage, we open a thread for a sphere to find the nearest several triangles to the grid where the sphere is. And then we make collision detection and change the status (velocity and position) of the sphere according to Discrete Element Method (DEM) model. These two steps are implemented in parallel on GPU to reduce the mainly time cost by loop operation on CPU. The collision

between a sphere and the walls is simply as the interaction between a sphere and a triangle mesh directly without much iteration, what is the reason we divide the space into infinitely small uniform grid, and the difference from traditional uniform grid methods. Hence, for a massive number of particles and suitable GPU, the time cost is reduced greatly.

3.2 Data structure

In the implementation of the algorithm, the data structures are organized as array, which is efficient for data operations and replication. They include particles array (pa), triangles array (ta), and grid array (ga). These data structures are shown in the later pseudo-code function (1) and function (2). The particles array is organized every eight float data for each particle, including three position coordinate values, three velocity coordinate values, a radius value and the grid number where the particle is. The triangles array is organized every 12 float data for each triangle including three points (9 coordinate values) and a normal vector. The grid array contains N (cf. N-triangles method in subsection F) integer number that stand for N numbers of triangle meshes that are nearest to the grid center, and three coordinate values of the grid center. The collisions are detected based on GPU, where at the beginning, we copy these data structures from CPU memory to GPU memory.

To find the number of triangles that are nearest to every space region, we apply the limit space decomposition described in next section.

3.3 Limit space decomposition and preprocessing

The world space will be cut into small enough uniform grids to represent the distance feature to the triangle meshes of the wall as shown in Fig. 2. The number of grids in three dimensions is denoted by (NUM_x, NUM_y, NUM_z), the grid length is L, the lengths of the space in three dimensions are denoted by (LEN_x, LEN_y, LEN_z), the volume of the space is V, and the total number of grids is denoted by NUM. Simply, we get formula (1).

$$\begin{cases} NUM_x = \frac{LEN_x}{L} \\ NUM_y = \frac{LEN_y}{L} \\ NUM_z = \frac{LEN_z}{L} \\ V = LEN_x \times LEN_y \times LEN_z \\ NUM = NUM_x \times NUM_y \times NUM_z \end{cases} \quad (1)$$

From formula (1), we get formula (2).

$$NUM = \frac{V}{L^3} \quad (2)$$

The number of these grids will be arranged from number 0 to number (NUM-1) in grid array, hence, for grid coordinates (x, y, z), the grid number is calculated through formula (3).

$$index = z \times NUM_y \times NUM_x + y \times NUM_x + x \quad (3)$$

In our implementation, we use the grid center (set as P) as a representative feature of distance. Hence, as an important part in preprocessing, the nearest triangle mesh is derived by iteratively calculating the distance from point P to every triangle mesh and selecting the nearest several triangle number to fill in the grid array. As for the space shown in Fig. 2, after the preprocessing, the grid array is organized as Table 1. The first row stands for the index of grid from 0 to $NUM-1$. The second row stands for the nearest triangle number to this grid, and the third row stands for the next-closest triangle number and so on. For example, for points in grid $O3$, the nearest triangle mesh is $T3$, and the next-closest triangle mesh is $T1$, and $T2$ is the farthest triangle mesh from grid $O3$.

From formula (2) we can know that the number of grids, NUM, is inversely proportional to the cube of grid length L. So a decrease in grid length will obviously increase the computation and memory to be occupied. Here we designed a method to reduce the computation. The core idea is that, for the bigger grids that are too far from the wall that can not collide with any triangle even for the largest particles, we can filter them out. Based on this idea, as shown in Fig. 4, we

Table 1 The grid array after preprocessing

GridNum	O1	O2	O3
Tri1	T1	T2	T3
Tri2	T2	T3	T1
Tri3	T3	T1	T2
...	...	...	...

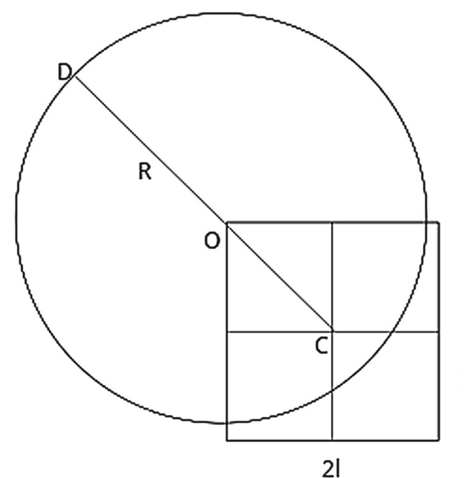


Fig. 4 The length to cull space. R is the maximum radius of particles

firstly divide the space with grids whose length are $2 * L$ and compute the distances from these triangles to the grid center. In reality, if the shortest distance is larger than CD, we can judge that the particles in this grid will make no collision with any triangle mesh. In the next step, when we divide the space into uniform grids whose length are L, it is not necessary to compute the grid in the bigger grid. The filter length CD can be calculated by formula (4):

$$FilterLength = R + \sqrt{3}l \tag{4}$$

Suppose the number of bigger grids is n, the computation cost of one grid is 1, the ratio of filtered grids is λ, the number of grids that are not filtered is $(1 - \lambda)n$. From formula (1), these non-filtered grids will be divided into $8(1 - \lambda)n$ grids. Hence, the total computation is $n + 8n(1 - \lambda)$. If we divide the grids into $8n$ grids directly, the computation is $8n$. By a simple computation we know that when λ is greater than 0.125, using this method will reduce overall computation cost. If λ reaches 0.5, it will reduce 37.5% computation.

In essence, this is a kind of ideal situation if considering only the computation for searching nearest triangle. It also indicates that if the triangles of collision wall is very few, the method will decrease computation significantly.

Although massive calculation is involved in this step, for each grid, it is parallel to get its triangle list ordered by distance and there is no influence among each other. Hence, with the help of concurrent computation based on GPU, this step could be achieved in a reasonable time frame even if the length of grid has a very small value.

The pseudo-code of kernel function for preprocessing is shown in function (1), which uses 1-triangle method simply.

3.4 Collision detection

Once we have finished preprocessing, the required data structures, such as particles array, triangles array and grid array are established correctly in memory of GPU. The particle-wall interactions can be detected easily. It should be noted that a few seconds may be needed in preprocessing stage. In the continuous simulation, the concrete physical collision model (Such as a DEM [29,30]) is considered. In our whole project, we used DEM soft ball model to simulate and calculate the process of collision. However, in this paper, we focus on the collision detection between particles and irregular vessel wall only. The way to simulate collision detection, as mentioned above, is mainly to calculate the distance from the particle center to the nearest triangle, which can be indexed directly and efficiency from the grid array. If the distance is smaller than the radius, then we assume that there is a collision between the particle and the triangle (wall).

Function (1)

Input:
1: *ng* number of grids,
2: *nt* number of triangles,
3: *ga* grid array with center point coordinates and nearest triangle numbers,
4: *ta* triangle array with vertex coordinates

Output: *ga* update nearest triangle numbers

```
5: function KernelPreprocessing(ng, nt, ga, ta)
6:     index ← threadId
7:     if index >= ng then return
8:     else
9:         P ← ga[index].center
10:        min ← FLOATMAX
11:        for i = 1 to nt do
12:            T ← ta[i].coordinates
13:            dis ← DISTANCE(P, T)
14:            if dis < min then
15:                min ← dis
16:                ga[index].num ← i
17:            end if
18:        end for
19:    end if
20:    return ga
21: end function
```

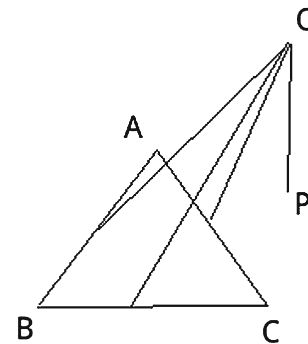


Fig. 5 The distance from sphere center O to triangle ABC

As shown in Fig. 5, the last issue is the calculation of the shortest distance between the sphere core (set as O) and the triangle (set as ABC). Our solution to this problem is as follows: when the point O is projected in the triangle, the shortest distance point must be the point of projection. If the projective point is out of the triangle, the shortest distance point must be in the edge of the triangle. Hence, we can simply get the shortest distance to each edge of the triangle and select the smallest one. Next, we only need to determine whether the projection is inside or outside of the triangle. It is based on the spatial geometry relationship between these points. If the projection is inside of the triangle, for point A, the angle between vector $\vec{AO}$ and vector $\vec{AC}$, vector $\vec{AO}$ and vector $\vec{AB}$ must be both acute angles. Concretely, this could be determined simply by the vector dot product:

$$\begin{cases} \vec{AO} \cdot \vec{AC} \geq 0 \\ \vec{AO} \cdot \vec{AB} \geq 0 \\ \vec{BO} \cdot \vec{BA} \geq 0 \\ \vec{BO} \cdot \vec{BC} \geq 0 \\ \vec{CO} \cdot \vec{CA} \geq 0 \\ \vec{CO} \cdot \vec{CB} \geq 0 \end{cases} \tag{5}$$

That is, if the point A, B, and C satisfy the formula 5, the projection of point O is projected in the triangle mesh ABC.

Some other approaches can be used to get the shortest distance between a point and a triangle. For example, Voronoi area method and vector calculus method were researched in [31]. This is beyond the scope of this paper.

The pseudo-code of kernel function for collision detection is shown in function (2).

3.5 Accuracy verification

In our work, the algorithm LSDCD is generally applicable to any regular or irregular wall. In some cases, we can not calculate the distance from a particle to the wall directly so as to make collision detection. As shown in Fig. 6, we herein choose a cylinder (which is located at the origin and is parallel to the Z axis) as the basic wall. Using such a model can help us to calculate the distance in mathematics method and calculate accuracy easily. Subsequently, it is more easily to verify the accuracy of collision. The function to calculate the distance from a point (x, y, z) to the cylinder is shown in function (3).

Function (2)
Input:
1: *np* number of particles,
2: *pa* particle array with position coordinates, velocities, radii, and grid number
3: *ga* grid array with nearest triangle num,
4: *ta* triangle array with point coordinates
Output: *pa* update position coordinates and velocities

```
5: function KernelCollision(np, pa, ga, ta)
6:     index ← threadId
7:     if index >= np then return
8:     else
9:         P ← pa[index].coordinates
10:        num ← CalcGridNum(P)
11:        T ← ta[num].coordinates
12:        dis ← DISTANCE(P, T)
13:        if dis < pa[index].radius then
14:            pa[index].velocities ← UpdateVelocities()
15:        else
16:            pa[index].coordinates ← UpdatePosition()
17:        end if
18:    end if
19:    return ga
20: end function
```

Function (3)
Input: x, y, z spatial coordinates, r radius, h height
Output: d distance

```
1: function CalcDistance(x, y, z, r, h)
2:     d ← 0
3:     if z <= 0 then
4:         if x ∗ x + y ∗ y < r ∗ r then
5:             d ← −z
6:         else
7:             t ← sqrt(x ∗ x + y ∗ y) − r
8:             d ← sqrt(t ∗ t + z ∗ z)
9:         end if
10:    else
11:        if z >= h then
12:            if x ∗ x + y ∗ y < r ∗ r then
13:                d ← z − h
14:            else
15:                t ← sqrt(x ∗ x + y ∗ y) − r
16:                d ← sqrt(t ∗ t + (z − h) ∗ (z − h))
17:            end if
18:        else
19:            if x ∗ x + y ∗ y < r ∗ r then
20:                d ← min(r − sqrt(x ∗ x + y ∗ y), z)
21:                d ← min(d, h − z)
22:            else
23:                d ← sqrt(x ∗ x + y ∗ y) − r
24:            end if
25:        end if
26:    end if
27:    return d
28: end function
```

From function (1) we can calculate the collision statuses via mathematical method, and we get the accuracy of collision detection by comparing it with the results of our LSDCD algorithm. The accuracy formula is shown as the following formula (6).

$$accuracy = \frac{\sum_{i=1}^{n} S_i}{n} \tag{6}$$

Where n is the total number of particles, S_i is the status of the collision detection of the ith particle. If collision result of ith particle by our algorithm is consistent with the result by mathematical method, the value is set as 1, otherwise the value is set as 0.

It is also a matter of concern that the model represented by STL file format uses many triangles to approximately describe the cylinder. It is inevitable that a little error occurs between the STL model and the real cylinder model in distance calculation. Hence, the real accuracy of collision may be smaller than 1.0. In order to get the relative accuracy between our LSDCD algorithm and the real mathematical method, we define the error rate as formula (7).

$$error = \frac{best - accuracy}{best} \times 100\% \tag{7}$$

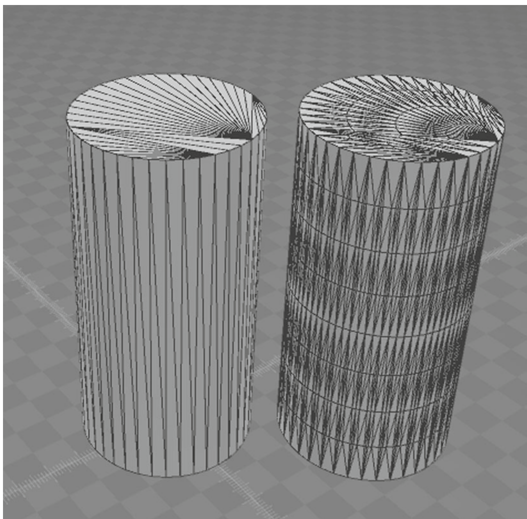
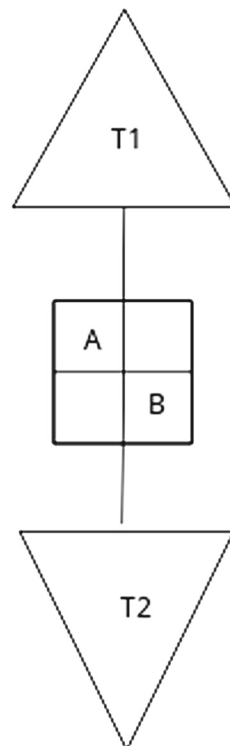


Fig. 6 The model of irregular wall and the divided model

Fig. 7 The possible problem in the algorithm



Where *best* is the maximum value of accuracy when using all triangles, and *accuracy* is the actual value of precision by our LSDCD algorithm.

3.6 N-triangles method

There is an issue when the grid is too small and the points in the grid are similar to all triangles with respect to the distance. As shown in Fig. 7, the center of the grid has the same distance to triangle $T1$ and triangle $T2$. For points in region A, they are nearer to triangle $T1$. However, for points in region B, they are nearer to triangle $T2$. To deal with this type of deviation, we use more than one nearer triangles to calculate the shortest distance. In other words, for points in region A and B, we calculate the distances to both triangle $T1$ and triangle $T2$ and select the smaller value. In fact, as can be seen from the experimental part, when we choose the nearest two or three triangles, the accuracy is almost the same as that when we use all triangles. If all triangles are used to iteratively obtain the minimum value of these distances, the accuracy is the real accuracy (it may be less than 1.0). The algorithm using only N triangles is denoted as N-triangles method. Essentially, our preprocessing process is designed to sort the space information respecting the principle that, the nearer triangles are more important to a grid. We select the nearest N triangles and cull those triangles that are too far away to collide.

3.7 Collision detection between rigid bodies

The idea of our algorithm can be widely applied to collision detection for various situations. The collision determination between near colliding objects is widely used in accurate collision detection system. We can also assume the objects as rigid bodies, and give an instance of collision detection between two rigid bodies with our LSDCD algorithm.

When simulating the collision between two rigid bodies, we still divide the space into infinitely small uniform grids. The difference herein is to regard one model (Fig. 6, usually the simpler one) as the irregular wall and divide another model (for example as a lion in Fig. 8) into very small triangles.

In the preprocessing stage, we divide the triangles of another rigid body model until the longest edge of these

Fig. 8 The model of lion

triangles are smaller than the length of the grids. The data structures of the triangles include one left triangle and one right triangle, that is the left and right subtrees of a binary tree. Firstly, we check if the longest edge of the triangle is larger than the length of grids. If true, we split the triangle into two smaller triangles from the midpoint of this edge and add the two triangles to the left tree and right tree. At last, we recursively divide the two subtrees. Fig. 6 also shows the divided model. The image in the left is the original model, and the right is the divided model.

The preprocessing of grid array is the same as mentioned in part .C. While making collision detection, we firstly get the center point of the triangle and assign the triangle into one grid by its center point. Then we get the grid index by formula (3). Next we get the nearest two or three triangles from grid array through index. At last, we make triangle-triangle intersection test between the triangle of the second model and the triangles of the wall. It should be noted that the number of triangles of the wall would only affect the time of preprocessing in our algorithm.

We implement triangle-triangle intersection test with the algorithm proposed by Devillers and Guigue [32]. This algorithm firstly calculates the determinant, which is composed of the vertices of the triangles. And then it judges the relative position between the points, lines and surfaces of the triangles by the signs of the determinant. At last, intersection of the two triangles is judged by the relative position. In fact, our LSDCD algorithm is applicable not only for rigid body. The principle is, we should make sure that one body is left relatively unchanged, so as to keep the grid array unchanged.

4 Experimental results and analysis

The Fig. 1 showed the visualization of collision detection tests between different number of particles and the spallation target. The experimental environment and analysis for our algorithm are as discussed in this section.

4.1 Experimental environment

In the next subsections, we mainly focus on the experimental results and analysis. We tested the computational efficiency and accuracy, the influence of grid length, the difference between N-triangles methods. We further investigated the performance of LSDCD algorithm when running on different GPUs. The result of collision detection between rigid bodies would also be given. At last, we would analyze the best accuracy of collision detection.

The experimental environment is set as follows: a server with Intel Xeon E5-2620 processor and 8GB RAM and a Tesla K40 GPU. The operating system is selected as Ubuntu 15.04 with CUDA version 7.5 installed. The algorithm was

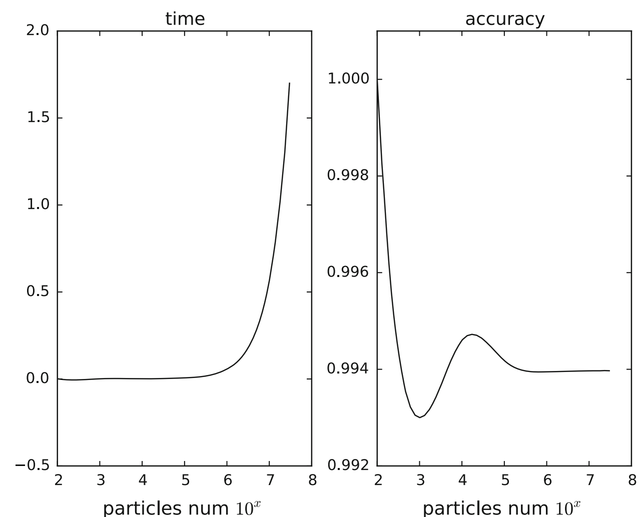


Fig. 9 Timings and accuracy of different number of particles

also executed on a personal computer with Intel Core i3 processor and 4GB RAM and a GTX480 GPU and Ubuntu 15.04 with CUDA version 7.5 installed, to verify the generalization of the algorithm.

We use a cylinder (STL format, height 100 and radius 50, as in Fig. 6) as the irregular collision wall drawn by a 3D modeling software. The rigid body is a lion whose size is about $25 \times 12 \times 25$, as Fig. 8. The particles are initialized in different size but the maximum radius of the particles is set as 1.

4.2 Computational efficiency and accuracy

We tested the computational efficiency and accuracy using a fixed grid length 0.2 and two-triangles method. As shown in Fig. 9, the number of particles is denoted as 10^x.

When the number of particles is less than one million, the time is at most several milliseconds (ms). When the number of particles is more than a million, the time is approximately proportional to the number of particles. Even so, when the number of particles reach ten million, we can see that the time of collision detection is only 564 ms. It indicates that our LSDCD is an efficient algorithm.

The accuracy of collision detection reduces a little with the increase of particles number, but it keeps constant (0.994) when the number of particle is more than a million. It indicates that, although it is an approximation algorithm, our LSDCD is sufficiently accurate for applications, especially for collision simulation of particles-irregular-wall.

4.3 The influence of grid length

We tested the influence of grid length by fixing the number of particles as 10^7 and using two-triangles method. As shown in Fig. 10, when the length of grid increases, the accuracy

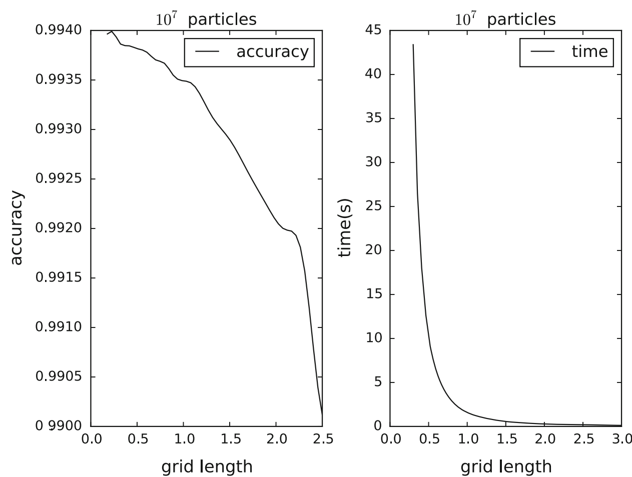


Fig. 10 The influence of grid length

begins to fall, which is consistent with our algorithm design. While we stress the importance of an infinitely small value of grid length, it has a high degree of accuracy, when the grid length is reduced to 0.2 only. On the other hand, as shown in the right of Fig. 10, with the decrease of the grid length, the time of preprocessing increases drastically. This is consistent with formula (2) where the time increases in inversely proportional scale to the cube of grid length. As we mentioned above, the whole process of collision detection contains two stages, preprocessing stage and collision detection stage. Although it takes a few seconds in preprocessing, the time of collision detection spent in each frame is reduced greatly and it takes about 0.5 second for collision detection for 10^7 particles regardless of the length of grid.

4.4 The accuracy improved by N-triangles methods

In order to test the difference between N-triangles methods, we experimented them based on one-triangle, two-triangles, three-triangles and all-triangles methods. We fixed the number of particles as 10^7, and the grid length is set as 0.2, 0.3, 0.4 and 0.5.

The experimental result is shown in Table 2. The label $accuracy1$ stands for the accuracy calculated by one-triangle method and $timing1$ stands for corresponding execution time. Similarity, $accuracy2$ and $accuracy3$ represent accuracy for two-triangles and three-triangles, $timing2$ and $timing3$ represent time used with two-triangles and three-triangles.

From Table 2 we can see that using more triangles will generally get higher accuracy. However, when using N-triangles method, the time cost will be N times of one-triangles. $accuracyN$ stands for the best accuracy when using all-triangles. We can see that the accuracy is the same value as $best = 0.994025$ even for the different values of grid length. The $timingN$ becomes 57.899280 seconds which is fairly unacceptable, and that is why we do not naively use the all-triangles method.

Table 2 Experimental results of accuracy based on N-triangles methods of 10^7 particles

gridlen	0.2	0.3	0.4	0.5
accuracy1	0.963049	0.963002	0.961837	0.962520
timing1	0.362065	0.367259	0.372562	0.372499
accuracy2	0.993950	0.993887	0.993887	0.993817
timing2	0.564954	0.581134	0.591163	0.595089
accuracy3	0.994015	0.993997	0.993982	0.994011
timing3	0.937022	0.956412	0.970890	0.976490
accuracyN	0.994025	0.994025	0.994025	0.994025
timingN	57.899280	58.054784	58.001506	57.737152

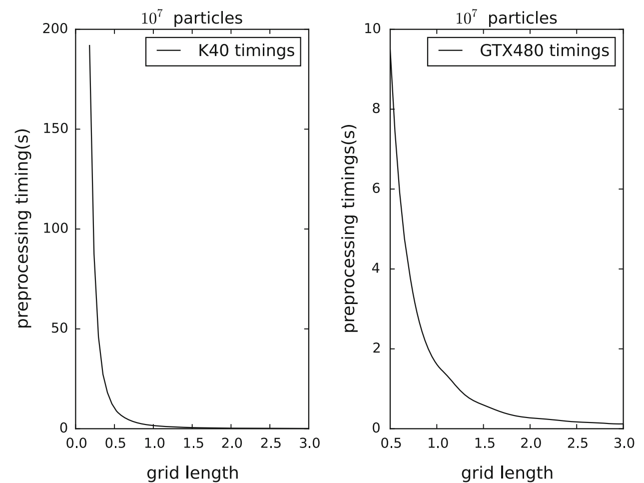


Fig. 11 The comparison of preprocessing time between different GPUs

In essence, the best accuracy is more related to the number of particles. We calculated error rate as formula (7). For example, when the grid length is 0.2, compared to using one-triangle, using two-triangles will improve accuracy from 0.963049 to 0.993950, and the error rate reduces from 3.11 to 0.0075%. When using 3 triangles, accuracy reaches 0.994015 and the error rate reduce to 0.001%. Considering the time factor, the execution time is proportional to the number of triangles. So basically, using two triangles is a better choice to achieve the trade-off between accuracy and computational efficiency. In general purpose, the two-triangles method is used by us as default.

4.5 On different GPUs

In order to test the applicability of the algorithm, we also tested the code on a low capacity personal computer with a GTX480 GPU. Because of the limitation of computation

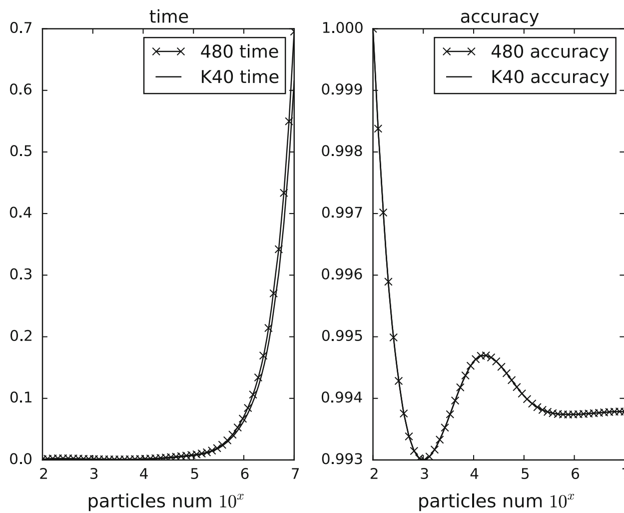


Fig. 12 The comparison between different GPUs

capacity, the minimum value of the grid length is set as 0.6. The time is composed of two parts by summing preprocessing time and collision detection time during one frame.

From Fig. 11, we know that the time spent in preprocessing stage is similar. For example, when the grid length is set as 0.5, the timings are both 9 s on two kinds of GPUs. However, with a more capable GPU we can even calculate the situation, where the length of grid is smaller, so as to improve overall accuracy. In fact, given the fixed number of particles randomly, the maximum accuracy is almost fixed and it is actually related to the degree of approximation of the model that is composed of triangles. For example, if we fixed the number of particles as 10^7, the maximum value of accuracy is 0.994025. When the grid length is reduced to 0.179 by running on a more powerful K40 GPU, the accuracy reaches 0.993847 and the error rate is only 0.0015%.

From Fig. 12, we know that a more powerful GPU will reduce a little time for collision detection. However, the accuracy is almost the same for the same grid length. This proves that our algorithm is stable.

4.6 Result of collision between rigid bodies

We also tested the collision between the wall and the lion model shown in Fig. 8. The model of lion is made up of 1.77M triangles after triangle division. The triangles are divided into smaller triangles, whose lengths of sides are less than 0.1.

As a benchmark, in [23], Tang et al. detected collision of 1.6M triangles in about 316.6 ms with GTX480. We used LSDCD for a similar collision detection and it needed 22.9 ms only to detect collisions of 1.77M triangles with GTX480. Although they are not the exactly same applicable environments and the focus is not directly comparable, we can see that our LSDCD algorithm is also very efficient to deal with a large number of collision detections.

5 Conclusion

In the field of physics simulation, it is meaningful to implement the collision between particles and irregular walls. In this paper, we implemented the LSDCD algorithm to deal with this type of collision based on the idea of limit space decomposition. We tested the efficiency and accuracy of the algorithm and discussed the influence of the length of grid. An improved N-triangles method was also experimented and we analyzed the influence of different N-triangles methods. We also explored the applicability of this algorithm on different GPUs. As a generalization of this algorithm, we finally performed the collision detection between rigid bodies.

The experimental results prove that our algorithm is feasible and efficient. In general, this method can be widely extended into the problems of large number of space computation. For example, this method can be used to compute the distances from massive points to an irregular surface or object efficiently. These promotional applications will be undertaken in the future.

Acknowledgements This work was supported by National Natural Science Foundation of China under Grant No. 61402210 and 60973137, Program for New Century Excellent Talents in University under Grant No. NCET-12-0250, Strategic Priority Research Program of the Chinese Academy of Sciences with Grant No. XDA03030100, Gansu Sci. and Tech. Program under Grant No. 1104GKCA049, 1204GKCA061 and 1304GKCA018, Google Research Awards and Google Faculty Award, China.

References

1. NVIDIA: Particle Simulation Using CUDA, 1st ed. NVIDIA (2013)
2. http://www.rapidtoday.com/stl-file-format.html
3. Bergen, G.: Efficient collision detection of complex deformable models using aabb trees. J. Graph.Tools **2**(4), 1–13 (1997)
4. Chang, J.-W., Wang, W., Kim, M.-S.: Efficient collision detection using a dual obb-sphere bounding volume hierarchy. Comput. Aided Des. **42**(1), 50–57 (2010)
5. Tang, M., Curtis, S., Yoon, S.E., Manocha, D.: ICcd: interactive continuous collision detection between deformable models using connectivity-based culling. IEEE Trans. Vis. Comput. Graph. **15**(4), 544–557 (2009)
6. Tang, M., Manocha, D., Tong, R.: Mccd: multi-core collision detection between deformable models using front-based decomposition. Graph. Model. **72**(2), 7–23 (2010)
7. Lauterbach, C., Mo, Q., Manocha, D.: Hierarchical gpu-based operations for collision and distance queries. In: Eurographics (2010)
8. Sulaiman, H.A. , Othman, M.A., Ismail, M.M., Said, M.A.M., Bade, A., Abdullah, M.H.: Methodology of performing narrow phase collision detection for virtual environment. In: 2014 International Symposium on Technology Management and Emerging Technologies (ISTMET), May 2014, pp. 511–515
9. Liu, F., Harada, T., Lee, Y., Kim, Y.J.: Real-time collision culling of a million bodies on graphics processing units. ACM Trans. Graph. **29**(6), 154 (2010)
10. Xiong, Q., Li, B., Xu, J., Wang, X., Wang, L., Ge, W.: Efficient 3d dns of gassolid flows on fermi gpgpu. Comput. Fluids **70**, 86–94 (2012)

11. Garland, M.: Parallel computing with CUDA (2010)
12. Purcell, T.J., Buck, I., Mark, W.R., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Trans. Graph. (TOG) **21**(3), 703–712 (2002)
13. Baciu, G., Wong, W.S.-K., Sun, H.: Recode: an image-based collision detection algorithm. In: Sixth Pacific Conference on Computer Graphics and Applications: Pacific Graphics' 98, pp. 125–133. IEEE (1998)
14. Myszkowski, K., Okunev, O.G., Kunii, T.L.: Fast collision detection between complex solids using rasterizing graphics hardware. Vis. Comput. **11**(9), 497–511 (1995)
15. Govindaraju, N.K., Redon, S., Lin, M.C., Manocha, D.: Cullide: interactive collision detection between complex models in large environments using graphics hardware. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 25–32. Eurographics Association (2003)
16. Kipfer, P., Segal, M., Westermann, R.: Uberflow: a gpu-based particle engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 115–122. ACM (2004)
17. Zheng, J., An, X., Huang, M.: Gpu-based parallel algorithm for particle contact detection and its application in self-compacting concrete flow simulations. Comput. Struct. **112**, 193–204 (2012)
18. Shen, Y., Jia, Q., Chen, G., Wang, Y., Sun, H.: Study of rapid collision detection algorithm for manipulator. In: 2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA), pp. 934–938 (2015)
19. Xue, S., Ji, Z.: Research of collision detection algorithm based on particle swarm optimization. Comput. Des. Appl. (ICCDA), **1** (2010)
20. Qu, H., Zhao, W.: Fast collision detection algorithm based on parallel ant. In: Virtual Reality and Visualization (ICVRV), pp. 261–264 (2013)
21. Xue-li, S., Tao, L.: Fast collision detection based on projection parallel algorithm. In: Future Computer and Communication (ICFCC), vol. 1 (2010)
22. Qu, H., Zhao, W.: Fast collision detection of space-time correlation. Comput. Sci. Electron. Eng. (ICCSEE) **3**, 567–571 (2012)
23. Tang, M., Manocha, D., Lin, J., Tong, R.: Collision-streams: Fast GPU-based collision detection for deformable models. In: I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, pp. 63–70 (2011)
24. Zhang, X., Kim, Y.J.: Scalable collision detection using p-partition fronts on many-core processors. IEEE Trans. Vis. Comput. Graph. **20**(3), 447–456 (2014)
25. Wang, L., Shi, Y., Li, R.: An image-based collision detection optimization algorithm. In: 2015 IEEE China Summit and International Conference on Signal and Information Processing (China SIP), pp. 220–224 (2015)
26. Xu, R., Kang, L., Tian, H.: A g-octree based fast collision detection for large-scale particle systems. Comput. Sci. Electron. Eng. (ICCSEE) **3**, 269–273 (2012)
27. Yisheng, Z., Xiaoli, Z., Guofu, D., Yong, H., Meiwei, J.: A GPGPU-based collision detection algorithm. In: Image and Graphics, pp. 938–942 (2009)
28. Fan, Z., Wan, H., Gao, S.: Streaming real time collision detection using programmable graphics hardware. J. Softw. **15**:1505–1513 (2004)
29. Karunasena, H., Senadeera, W., Gu, Y., Brown, R.: A coupled sph-dem model for fluid and solid mechanics of apple parenchyma cells during drying. In: 18th Australian Fluid Mechanics Conference. Australasian Fluid Mechanics Society Launceston, Australia (2012)
30. Rhodes, M., Wang, X.S., Nguyen, M., Stewart, P., Liffman, K.: Study of mixing in gas-fluidized beds using a dem model. Chem. Eng. Sci. **56**(8), 2859–2866 (2001)
31. Ericson, C.: Real-Time Collision Detection. Elsevier (2010)
32. Devillers, O., Guigue, P.: Faster triangle-triangle intersection tests. In: INRIA (2002)

Binbin Yong received his master's degree in Computer Science and Technology from Lanzhou University in 2012. Now he is a PhD candidate and studying in the School of Information Science and Engineering, Lanzhou University. He is researching in parallel computing of GPU, machine learning, and artificial neural network.

Dr. Jun Shen was awarded PhD in 2001 at Southeast University, China. He held positions at Swinburne University of Technology in Melbourne and University of South Australia in Adelaide before 2006. He is an Associate Professor in School of Computing and Information Technology at University of Wollongong in Wollongong, NSW of Australia, where he had been Head of Postgraduate Studies, and Chair of School Research Committee since 2014. He is a senior member of three institutions: IEEE, ACM and ACS. He has published more than 120 papers in journals and conferences in CS/IT areas. His expertise includes computational intelligence, Web services, Cloud computing and learning technologies including MOOC. He has been Editor, PC Chair, Guest Editor, PC Member for numerous journals and conferences published by IEEE, ACM, Elsevier and Springer. A/Prof Shen is also a current member of ACM/AIS Task Force on Curriculum MSIS 2016.

Hongyu Sun was born in Kaifeng, China, in 1989. He is currently studying for a doctor's degree in Computer Science, under the supervision of Dr. Qiang He, at Swinburne University of Technology, Melbourne, Australia. Before entering the doctor's program, he had received the master's degree in computer technology from Lanzhou University, Lanzhou, China, in 2013. He was a recipient of the 2011 national scholarship and 2012 national encouragement scholarship as an undergraduate student of Lanzhou University. His research focuses on blockchain application, geometric modeling and processing, parallel computing and mass data visualization.

Huaming Chen received the B.Eng. and M.Eng. degrees from Lanzhou University, China, in 2012 and 2015. He is currently a PhD candidate in University of Wollongong. His main research interests are bioinformatics, machine learning, neural network.

Qingguo Zhou received the BS and MS degrees in Physics from Lanzhou University in 1996 and 2001, respectively, and received PhD in Theoretical Physics from Lanzhou University in 2005. Now he is a professor of Lanzhou University and working in the School of Information Science and Engineering. He is also a Fellow of IET. He was a recipient of IBM Real-Time Innovation Award in 2007, a recipient of Google Faculty Award in 2011, and a recipient of Google Faculty Research Award in 2012. His research interests include safety-critical systems, embedded systems, and real-time systems.