HAL
open science

An Octree-based proxy for collision detection in large-scale particle systems

Wenshan Fan, Bin Wang, Jean-Claude Paul, Jiaguang Sun

To cite this version:

Wenshan Fan, Bin Wang, Jean-Claude Paul, Jiaguang Sun. An Octree-based proxy for collision detection in large-scale particle systems. Science in China Series F: Information Sciences, 2013, 56 (1), pp.1-10. 10.1007/s11432-012-4616-5 . hal-00920666

HAL Id: hal-00920666

https://inria.hal.science/hal-00920666v1

Submitted on 19 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An octree-based proxy for collision detection in large-scale particle systems

FAN WenShan[1,2,3,4,5]*, WANG Bin[1,4,5], PAUL Jean-Claude[1,5,6] & SUN JiaGuang [1,4,5]

[1]*School of Software, Tsinghua University, Beijing* 100084*, China;*
[2]*Beijing Aerospace Control Center, Beijing* 100094*, China;*
[3]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084*, China;*
[4]*Key Laboratory for Information System Security, Ministry of Education of China, Beijing* 100084*, China;*
[5]*Tsinghua National Laboratory for Information Science and Technology, Beijing* 100084*, China;*
[6]*Institut National de Recherche en Informatique et en Automatique (INRIA), France*

Abstract Particle systems are important building block for simulating vivid and detail-rich effects in virtual world. One of the most difficult aspects of particle systems has been detecting collisions between particles and mesh surface. Due to the huge computation, a variety of proxy-based approaches have been proposed recently to perform visually correct simulation. However, all either limit the complexity of the scene, fail to guarantee non-penetration, or are too slow for real-time use with many particles. In this paper, we propose a new octree-based proxy for colliding particles with meshes on the GPU. Our approach works by subdividing the scene mesh with an octree in which each leaf node associates with a representative normal corresponding to the normals of the triangles that intersect the node. We present a view-visible method, which is suitable for both closed and non-closed models, to label the empty leaf nodes adjacent to nonempty ones with appropriate back/front property, allowing particles to collide with both sides of the scene mesh. We show how collisions can be performed robustly on this proxy structure in place of the original mesh, and describe an extension that allows for fast traversal of the octree structure on the GPU. The experiments show that the proposed method is fast enough for real-time performance with millions of particles interacting with complex scenes.

Keywords particle systems, collision detection, octree-based proxy, GPU

1 Introduction

Particle systems [1,2] have long played an essential role in simulating natural effects such as fire, smoke, and water in virtual environments. Although great progress has been made in simulating particle systems, interaction between a complex scene with arbitrary topology and a large number of particles in real-time remains an issue that has not yet been adequately solved.

The computational cost of simulating particle collision in a scene consisting of triangle meshes can be expensive, with a naive approach requiring tests between every particle and every triangle to compute collisions in a single frame. However, the fact that each particle's interaction with the scene is essentially

*Corresponding author (email: fws06@mails.tsinghua.edu.cn)

independent would seem to make particle collisions suitable for computation on the GPU, on which many threads can independently calculate particle-scene collisions in parallel, greatly increasing performance.

Unfortunately, the traditional acceleration structures, such as grids, octrees, kd-trees and so on, that are typically used to accelerate performance on the CPU, do not naturally translate to the GPU context. The memory demand of these acceleration structures may impose too high of a cost for the GPU, and the non-uniform nature of some of these structures (e.g., the variable number of triangles in a leaf node of an octree) greatly harm the parallelism of SIMD GPUs, as all computation must wait for the worst-case number of tests required in any branch. Meanwhile, the efficiency is more important than physical accuracy when simulating particle-scene collisions in a lot of fields, these include computer games, movies, etc. Hence, various GPU-friendly proxies have been proposed to implement a fast and visually correct algorithm, such as texture-based depth maps [3], a lattice of planes [4], but these approaches either limit the input geometry or fail to guarantee that particles will not penetrate the environment.

In this paper, a new object-space proxy for complex scenes is proposed that can accelerate collision processing on the GPU. The proposed proxy is based on an octree subdivision of the original model. The octree bounds the original model and is refined until it converges closely enough to the shape of the model to satisfy a set of termination criteria. Then, for each nonempty leaf node, a single representative normal is computed that is simply the average of the normals of the triangles that intersect the node. In addition, to deal with the issue of arbitrary distribution of particles in practical scene, a general view-visible method for both closed and non-closed models is introduced to label empty leaves adjacent to nonempty ones with back/front property.

This proxy structure can then be uploaded to the GPU and used in place of the original mesh. During simulation, a particle that enters a nonempty leaf node of the proxy tests its velocity against the node's representative normal. If it is found that the particle is travelling in a direction opposed to the normal, then the particle is considered to be colliding with the model, and collision response is carried out based on the representative normal. Otherwise, the particle is allowed to continue on its path. Because a particle's velocity is always tested against a node's representative normal, this approach guarantees that the particle will not penetrate the surface.

Collisions with both sides of the mesh are handled by labelling particles as they pass through empty nodes adjacent to nonempty ones, marking them with appropriate property according to the empty node's back/front property. Then, when a particle encounters a nonempty leaf node, it simply processes collisions based on either the default or flipped version of the node's representative normal depending on whether it is in front or at back of the surface, respectively. Note that this approach sidesteps the issue of labelling all points in the scene volume as being at back or in front of the surface, which may be difficult or impossible for meshes with holes; instead, it only requires labelling nodes immediately adjacent to the surface of the mesh, which can be done robustly even on open meshes.

Finally, we describe how a lookup table that records all the nodes of the octree at a certain depth can be used to optimize the octree traversal process by beginning at a node deeper than the root, further accelerating collision detection on the GPU using this proxy. The result is a fast, flexible, and robust proxy for particle system collisions with arbitrary scene geometry.

2 Related work

Particle systems. Reeves [1] first proposed the simulation of particle systems and the basic motion operations and data structures which are now familiar. It was quickly realized that the interactions between each particle and the scene are typically independent, making it possible to simulate each particle in parallel. Sims [5] presented a technique to animate and render a particle system running on a parallel supercomputer. More recently, GPGPU methods have been employed [6] to tackle the simulation of large particle systems. Latta [7] presented a full GPU implementation of a particle system, in which particles were encoded into textures on the GPU. Kipfer et al. [8] proposed an approach to implement particle simulations on the GPU that was able to eliminate all data transfer between the GPU and CPU at runtime, removing a performance bottleneck. However, both of these approaches could only handle

collisions between particles and a scene modeled by a height field.

Venetillo and Celes [9] proposed a way to represent a scene as a collection of basic primitives, such as planes, spheres, cones, and so on, to make collision processing simpler on the GPU. However, this approach does not support complex scenes.

In theory, a simplified (reduced polygon count) version of the input scene, as generated by any simplification method, could be employed as a proxy for the original complex mesh to generate approximate collision results. However, the deviation between the original mesh and the proxy tends to result in visual errors. Hence several more elaborate proxies have been presented.

Kolb et al. [3] proposed an implicit representation of the original mesh for collision detection in which a set of depth maps containing position and normal vectors at each point are precomputed and used to represent the outer boundary of the scene. However, their proxy is constrained to work only with closed models, and, as an image-based technique, has an accuracy that is dependent on the resolution of the depth map and the view direction.

Drone [4] partitioned the scene into a uniform three-dimensional lattice with each nonempty cell containing an important plane corresponding to the surface of original scene. However, the memory requirements for a uniform three-dimensional grid are prohibitive for complex scenes. Furthermore, this approach can result in missing some collisions, causing particles to penetrate the scene.

Octrees. A general introduction to the spatial acceleration structure can be found in [10–13]. In a classical approach, an octree representation of a mesh would record a variable number of triangles in each leaf node. Benson and Davis [14] and DeBry et al. [15] instead proposed storing a single representative color in each leaf node of an octree. Then, a fragment's position can be used as an index into the octree to directly determine the fragment's color, avoiding the complex problem of creating a 3D to 2D parameterization for texture mapping [16]. Lefebvre et al. [17,18] ported the resulting octree texture onto GPU to improve performance. However, in their approach the root node was always used as the entrance node for each access pass. Castro et al. [19] used a linear hash octree [20] as a representation and estimated deep entrance nodes using a cost function on the CPU to improve performance, but the accessing conflict of hash tables incurred a performance hit.

3 Constructing the octree-based proxy

The proposed proxy utilizes the convergence [21] of octree subdivision which conforms to the shape of the input mesh surface with the increase of the subdivision level. See Figure 1. However, different from general octree-based acceleration structure, in which each nonempty leaf node contains a set of triangles that intersect that node, we instead associate nonempty leaf nodes with a single representative normal. We also label empty nodes adjacent to nonempty ones with a back/front property recording whether they are at back or in front of the surface, which, as we later show, will enable us to support particles colliding with both sides of a mesh.

3.1 Octree subdivision

The construction of the basic octree structure proceeds in a typical fashion. The octree is first initialized to a single node consisting of a bounding box encompassing the entire scene, and a set of triangles that includes all the triangles in the scene. This node is then subdivided into eight child nodes, with each child node's bounding box being one-eighth of its parent's. An intersection test is then run between the parent's triangles and the child nodes' bounding boxes, and the triangles that intersect a child node's bounding box are assigned to that node.

This subdivision is carried on recursively until the following termination criteria are met:

- The current node doesn't contain any geometry.
- The predefined maximum subdivision depth $d_{\max}$ is reached.
- The subdivision depth is greater than a predefined minimum depth $d_{\min}$ and the normals of the triangles in the current node vary by less than a specified tolerance ε. A revised $L^{2,1}$ metric [22] is def-

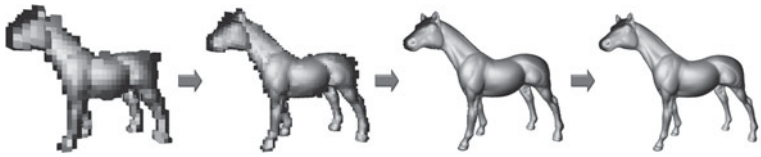

Figure 1 As the nodes of the octree are further subdivided, the nonempty leaf nodes (left three images) converge to the shape of the original model (rightmost image).

ined to estimate the normals' variance:

$$L_r^{2,1} = \sum \|\boldsymbol{n}_i - \boldsymbol{n}_{\text{avg}}\|^2, \tag{1}$$

where

$$\boldsymbol{n}_{\text{avg}} = \left\| \frac{\sum a_i \boldsymbol{n}_i}{\sum a_i} \right\|, \tag{2}$$

$\boldsymbol{n}_i$ is the normal of triangle i, and a_i is the area of triangle i.

The above subdivision process results in a generated octree, with each nonempty leaf node corresponding to a small patch of the input surface and containing a set of triangles with similar normals. At this point, rather than record all of the contained triangles, we simply record a single represent normal that approximates the normal of the surface over the corresponding patch by (2).

Because the position of the nonempty leaf nodes corresponds to the shape of the surface, and the nodes' representative normals correspond to the normals of the surface along its boundary, the resultant octree is a good approximation of the original mesh for the purpose of particle collision detection.

However, on the one hand, just a single representative normal is recorded in each nonempty leaf node, and on the other hand, the particles may end up at both sides of the model surface, either because they were initially placed there, or because the scene geometry was not closed and a particle moved through the hole. If a particle that is at back of the surface collides with the scene geometry, collision detection and response must be carried out with a flipped version of the representative normal. Hence, it is essential to track at which side of the surface the particles lie, so that the correct version of the representative normal can be retrieved to process collisions. We will describe a general view-visible method to deal with the issue.

3.2 Labelling empty nodes

The traditional ray casting based odd-even test [23] can work well to query if a point is inside or outside the closed model. However, the appropriate side may be poorly defined for non-closed models with it. For example, different results may be generated depending on whether or not the test ray passes through a hole with the odd-even test.

We solve this by observing that particles only need to be determined as being at back or in front of the surface immediately before a collision with the scene geometry, and that although these properties may be poorly defined over the scene as a whole, they are easily defined in nodes immediately adjacent to the surface, by testing whether the triangles visible to the node have normals pointing towards the empty node (indicating that the empty node is in front of the surface) or away from it (indicating that the node is at back of the surface).

We therefore label all empty nodes of the proxy adjacent to nonempty ones as being at back or in front of the surface using a general view-visible method described below. Each particle then records a back/front property which is set as the particle passes through these labelled empty nodes. Because particles must always pass through empty nodes before they encounter the surface (see Figure 2(a)), this is guaranteed to happen before we need to determine a collision. Then, when the labelled particle does encounter the surface, the default or flipped version of the surface node's representative normal can be retrieved as appropriate.

Our view-visible method proceeds by considering each nonempty leaf node in turn, and labelling its adjacent empty leaf nodes as being at back or in front of the surface. Although a single empty leaf node

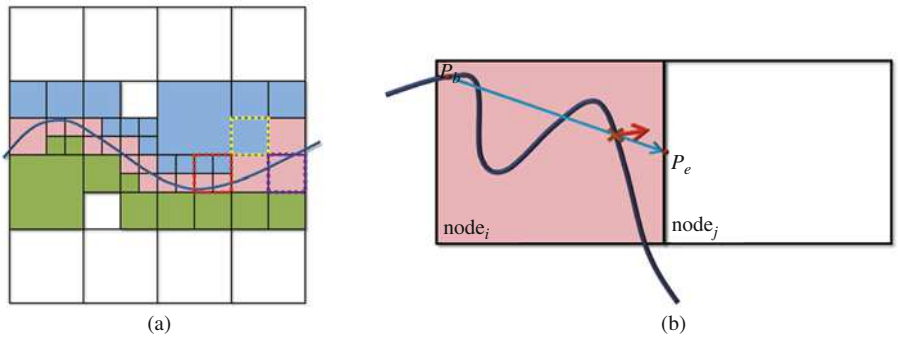

Figure 2 View-visible method for labelling empty nodes. (a) The empty leaf nodes, which are at back (in green) or in front (in blue) of the mesh surface, must be traversed first before the particles approach to the scene mesh. The dash lines with different colors highlight all the possible cases when labelling empty leaf nodes adjacent to surface; (b) a line segment is drawn from a point P_b on a triangle in the nonempty node to a point P_e shared by both nodes. The normal (in red) at the closest intersection (orange cross) of the line segment and the surface to P_e is used to determine the back/front property of the empty node.

may be adjacent to multiple nonempty leaf nodes, and therefore be labelled multiple times, all labellings should agree, making that not a problem.

We now show how to do this for a single nonempty leaf node node$_i$. The first thing that must be done is to determine which nodes are adjacent to node$_i$ and can be labelled. This task would be trivial if dealing with a regular lattice, but in an octree, a node may be adjacent to nodes that are of the same size, larger, smaller, or a combination. In our approach, rather than treat every possible configuration separately, we assume that there is a full set of 26 neighbors of the same size (as there would be in a regular lattice) and calculate what the centers of those nodes would be. We then take each center point and find the node at our depth or less that contains the given point, and store it as node$_j$. At this point, we are dealing with a single adjacent node, and have just three cases to deal with: node$_j$ is a nonempty leaf node (highlight by purple dash line in Figure 2(a)), in which case it does not need to be processed; node$_j$ is a nonempty non-leaf (highlight by red dash line in Figure 2(a)), in which case it can be left for another neighbor of the node at its own depth (which must exist) to process; or node$_j$ is an empty node (highlight by yellow dash line in Figure 2(a)), either at the same depth or shallower, in which case it can be labelled using the method described below.

We now consider a nonempty node node$_i$ and an adjacent empty node node$_j$. Our goal is to find a triangle in node$_i$ of which node$_j$ has an unobstructed "view" and determine node$_j$'s back/front property based on the normal of that triangle. This is done by first constructing a line from a beginning point P_b that is any point laying on a triangle in node$_i$'s triangle set and is within the bounding box of node$_i$ to an end point P_e that lies on the boundary shared by both the bounding boxes of node$_i$ and node$_j$. We then find all the triangles in node$_i$ that intersect this line, and take the one that is with the closest intersection to P_e as the triangle that will determine node$_j$'s back/front property (see Figure 2(b)). The dot product of this triangle's normal and the direction of the line determine whether the triangle is "facing" node$_j$, and therefore whether node$_j$ is then at back or in front of the surface: if the dot product is positive (the normal and the line point in the same direction), node$_j$ is labelled as being in front of the surface; if it is negative, node$_j$ is labelled as being at back of the surface.

Algorithm 1 illustrates a high level framework of the proposed view-visible method.

Figure 3 shows an example of a scene in which particles are colliding with both sides of a mesh, and how our approach supports non-closed meshes as well.

4 Collision detection between particles and the scene

We now describe how our proxy can be used to detect collisions between particles and the scene. We leave the details of the implementation of the particle system (such as whether explicit or implicit integration is used) unspecified, as our approach will work with any particle motion scheme.

Algorithm 1 Framework of view-visible method.

Root of octree The back/front properties for all empty leaf nodes that are adjacent to nonempty ones
Traverse octree to retrieve a nonempty leaf node node$_i$;
$k \leftarrow 1$ 26 Calculate center position p of adjacent node with equal size to that of node$_i$;
Search p in the octree to find node$_j$ which contains p with depth less or equal to that of node$_i$;
node$_j$ (*[h]Do not need to label)nonempty leaf node break;
(*[h] Do not label at this level)internal node break;
empty leaf node Initialize a line segment P_bP_e;
Find a triangle T_v which intersects P_bP_e with the closest intersection to P_e;
frontorback = dot(T_v.normal, P_bP_e);
frontorback>0 Label node$_j$ with front property;
Label node$_j$ with back property;
break.

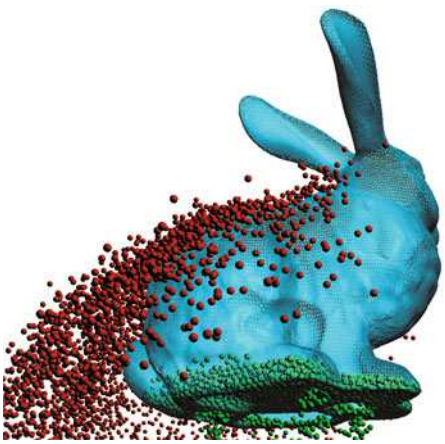

Figure 3 A scene demonstrating collisions with both sides of a mesh and support for non-closed meshes. The green particles were created inside the Bunny model and are interacting with the backside of the mesh surface and falling out of holes in the base of the mesh.

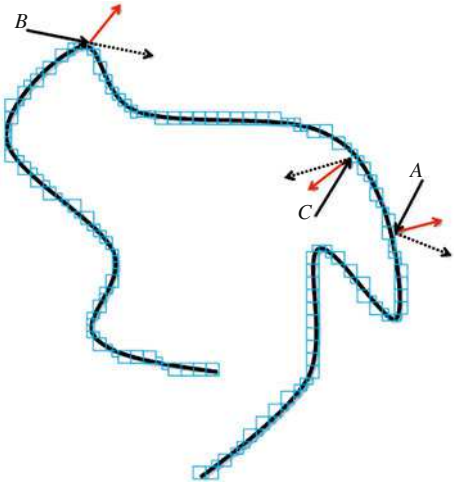

Figure 4 A 2D illustration of the interaction between particles and scene geometry. When a particle encounters a nonempty leaf node, it tests its velocity direction (black line) against the correct version of the node's representative normal (red line). If the particle is moving counter to the normal, a collision is detected and processed (particles A and C); if not, it continues on its path (particle B).

For each particle, we find the current position of the particle x^t and the intended new position computed by the particle system dynamics x^{t+h}. We then consider each of the nodes that the particle's path passes through in sequence. This process starts by setting the current node to the octree leaf node that contains the particle's start position. For each node, we run the test below, potentially update the planned path of the particle, find the point at which the particle exits the current node's bounding box (if it does) and use that position to query the proxy to find the next node. This repeats until the node containing the particle's end position is reached.

The test for each node is as follows:

• If the node is empty, the particle is allowed to continue on its path. If the empty node is flagged with a back/front property, we stamp the particle with the corresponding property.

• If the node is nonempty, we first retrieve either the default or flipped version of the node's representative normal, depending on whether the particle is stamped as being in front or at back of the surface, respectively, and label it $\boldsymbol{n}$. We then calculate the dot-product $\boldsymbol{n} \cdot \boldsymbol{v}$, where $\boldsymbol{v}$ is the velocity of the particle. If $\boldsymbol{n} \cdot \boldsymbol{v} \geqslant 0$, the particle is considered to be travelling away from the surface of the scene in this node, and the motion is unchanged. However, if $\boldsymbol{n} \cdot \boldsymbol{v} < 0$, this is considered a collision, and we perform

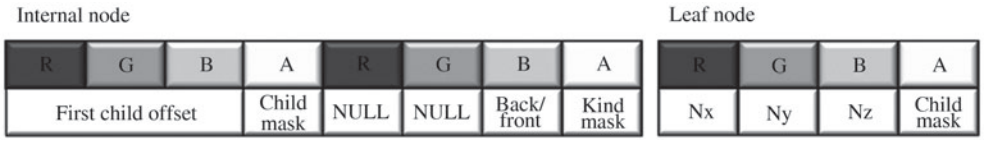

Figure 5 The encoding format of octree nodes. First Child Offset defines how far the parent node is from its first child in the texture, N_x, N_y and N_z define the representative normal. Child Mask identifies if the node has children and Kind Mask identifies the children as internal or leaf nodes if existing. Back/Front mask defines if the associated children are labelled as being at back or in front of the surface. Please see [24] for the detailed encoding scheme.

collision response at the point at which the particle entered the node and update the position and velocity as appropriate, which are used for further traversal.

Figure 4 illustrates the interaction between particles and scene geometry in our approach.

5 Querying the proxy on the GPU

In the previous section, we referred to accessing the leaf node of the octree that contains a desired query point. This operation is the core operation employed when using our proxy. In this section, we will describe an extension to speed up the operation on the GPU.

The first task we face is encoding the pointer-based octree into a flat texture which can be uploaded to the GPU to exploit the efficiency of texture fetching. For this, we use the encoding scheme proposed by Lacoste et al. [24] with modification by adding a back/front mask to define the associated empty children whether they are flagged as being at back or in front of the surface. Figure 5 illustrates the data that are written for each type of node used in our method.

Then, different from the usual querying method which always proceeds from the root node, we implement an optimized scheme that allows the octree traversal to begin at a deep entrance node, reducing the number of indirect accessing.

Our optimization works by constructing a lookup table for all the nodes of the octree at a single depth d. The lookup table is simply a 3D matrix of pointers $D(i, j, k)$ with dimensions $2^d \times 2^d \times 2^d$ that maps index (i, j, k) to the position in the texture containing node data at which the octree node at index (i, j, k) and depth d is recorded. If octree subdivision at that position stopped at a depth shallower than d, the mapping simply points to the leaf node ancestor of the node that would exist at that position. The lookup table also records the depth of the pointed-to node, to enable the calculation of the correct-sized bounding box for the node when querying the proxy.

Once this lookup table has been constructed and uploaded to the GPU, octree traversal can start at depth d (or at a shallower leaf node if no node at that position and depth d exists) instead of at the root by looking up the node containing the query position at depth d in the lookup table.

In constructing this lookup table, there is a trade-off between speed and table size, as a lookup table at a deeper depth will save more of the traversal, but require more space. We use a depth of 5 for all of our experiments.

In a test scene composed of 1024 k particles with inter-collisions falling onto the Bunny model, we observed a speedup of 53% with the lookup table relative to without.

6 Results

Model statistics, proxy generation settings and algorithm timings are provided in Table 1. The performance of our proxy determines the collision detection time, which is given in milliseconds per frame. All timings were generated on an Intel Core 2 processor running at 2.8 GHz with 3 GB of main memory and an NVIDIA GeForce 260 GTX graphics card with 896 MB of video memory. The particle system was implemented using NVIDIA's CUDA framework [25]. The method presented by Green [26] was employed in our system for inter-collision test among particles. As Table 1 indicates that the proposed proxy is able to support large-scale particle system in real-time. Images of particle systems that we are able to

Table 1 Model statistics, octree proxy generation parameters and timings of our algorithm for detect collisions between particles and models

Models	# Triangles	ε	$d_{\min}$	$d_{\max}$	Size of proxy (MB)	Timings (ms)		
						1 M Particles	2 M Particles	3 M Particles
Horse	96966	0.01	4	6	0.06	17.5–26.3	21.9–33.1	22.4–40.1
Angel	474048	0.02	5	6	0.05	19.4–25.3	21.1–33.4	22.5–39.8
Pegasus	667474	0.01	5	6	0.11	19.5–24.5	22.1–36.1	23.7–40.7
Dragon	871414	0.02	5	7	0.41	20.0–23.6	21.7–34.8	23.3–41.3
Buddha	1087716	0.01	5	7	0.31	20.6–23.5	24.9–36.7	30.9–50.5
Neptune	4007872	0.02	6	8	0.67	22.3–29.7	26.3–43.5	37.6–56.7

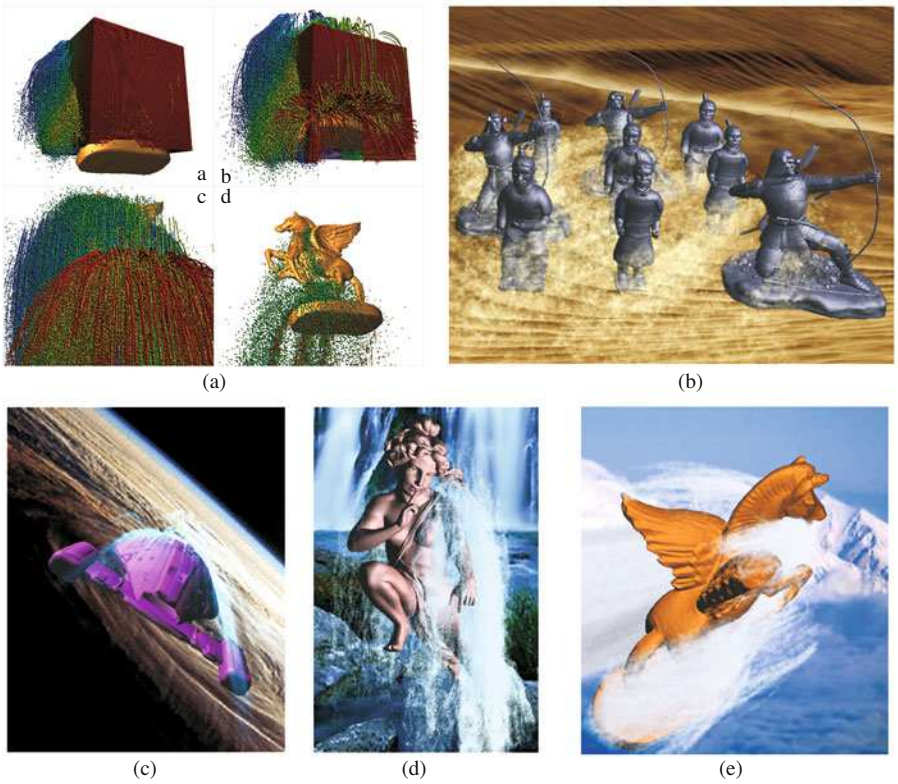

Figure 6 Particle systems that can be simulated with our approach. (a) 1 M particles are colliding with the Pegasus model, shown in progression (a–d); (b) warriors stand in the sand. Instead of building a proxy per model, just a single octree proxy is necessary to detect the collisions between particles and geometry in this kind of scene that consists of multiple objects; (c) a spaceship is landing. The collisions, which result in the heat and light between the atmosphere and high speed aircraft, are calculated by the proposed octree proxy; (d) the Angel is in the waterfall. The octree proxy is used to simulate the interaction between the water and the model; (e) pegasus is flying in the cloud sea. The octree proxy is used to calculate the collisions between the cloud and the model.

simulate with the proposed proxy are given in Figure 6. Note that the construction of the proposed octree proxy takes all the triangles in the scene as a whole collection, hence just a single octree proxy is necessary when simulating a particle system. The advantage makes the complexity of our algorithm irrelevant to the number of models consisting of the scene, saving the additional rendering passes when processing complex system composed of multiple objects, such as Figure 6(b), as compared with the style of building proxy per model.

The performance of our collision detection approach relative to a general octree acceleration scheme is given in Figure 7. The variable number of triangles in a leaf of the general octree imposes significant time and memory access penalties in the SIMD GPUs context, while our approach does not suffer from these effects. The result is a significant speed increase when using our proxy, especially with many particles, with only a minimal visual difference.

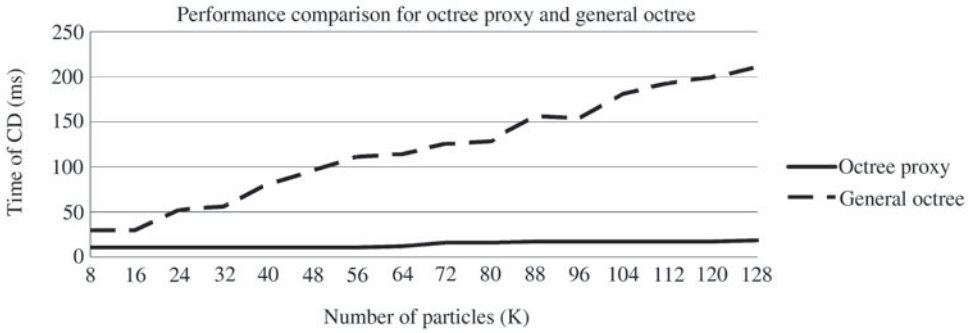

Figure 7 A comparison of the performance of our octree proxy based approach relative to that of a general octree accelerate scheme. The particles are tested colliding against the Horse model.

Comparing performance with other optimized methods for particle system collision detection is more difficult, as they vary in the restrictions placed on scene geometry, correctness guarantees, computation platform, and so on.

Castro et al. [19] presented an optimized method for octree traversal on the CPU that provided exact collision with the triangle mesh, but was far slower, achieving 4.1 FPS for a single particle interacting with the Buddha model.

Early GPU-based collision detection methods [7,8] employed heightmaps to represent scenes, restricting their methods can only handle simple geometry, such as terrain. Kolb et al. [3] performed collision detection with an implicit representation. In order to approximate the boundary of scene geometry, they reconstructed a set of depth maps, encoding the position and normal vectors, from the original triangle mesh with an intricate method. Compared with their method, the construction of our octree-based proxy is simple. In addition, their implicit representation based algorithm cannot process non-closed models.

Drone [4] proposed a uniform grid based method to detect collisions between particles and scene. They stored an important plane in each nonempty cell to approximate the mesh surface. Since only the triangles intersecting current cell are considered when building their important plane, the resultant proxy cannot ensure correct continuity among all cells, causing particles penetrate the scene in their method. Different from that, the converged octree structure is employed in our method to approximate the boundary of the scene geometry, removing the above issue, and the proposed optimized octree traversal scheme ensures good performance of our method.

Venetillo and Celes [9], meanwhile, presented a system capable of simulating 1024 K particles on the GPU at 6 FPS, but they limit the input geometry to a scene constructed by combining simple geometric primitives. To make a fair comparison, we tested our approach on a GPU of similar speed (an NVIDIA GeForce 8800 GT, relative to their slightly faster 8800 GTX), and were able to achieve 11.2 FPS on average with 1024 K particles on the Pegasus model.

7 Conclusion and future work

We have presented a novel proxy for complex meshes that can be used to perform collision detection for a large-scale particle system. We have shown how this proxy is suitable to being stored and run entirely on the GPU, allowing the real-time computation of collision detection for millions of particles. We have shown how our proxy can support collisions with both the sides of the geometry surface, and presented an optimization that increases the speed of octree traversal when computing collisions with our proxy.

As with all existing geometry proxies, our proxy performs only approximate collision detection with the scene geometry. In our case, the accuracy of our proxy is limited by the maximum subdivision level of the proxy's octree structure. Although the deviation between our proxy and actual scene geometry can be reduced by increasing the maximum subdivision level, both the memory requirement and the access overhead will increase. As a result, the problem may arise when processing models with some extreme cases, such as thin-plate and self-collision, in practical applications.

Currently, our octree-based proxy is initially constructed on the CPU as an offline procedure. However, it has been shown that octree construction can be performed on the GPU [27,28]; therefore, we would like to implement a dynamic constructing algorithm for our proxy on the GPU in order to support deformable models. We are also interested in exploring the possibility of employing the proposed view-visible method to solve other related issues, such as reconstruction on point cloud [29] for non-closed models.

Acknowledgements

This work was supported by National Basic Research Program of China (Grant No. 2010CB328001), National Natural Science Foundation of China (Grant Nos. 61003096, 51021140004), and National High-tech R&D Program of China (Grant No. 2009AA045201). We thank Alec Rivers for his helpful suggestions and ideas and editing work on the paper. The Bunny, Dragon and Buddha are courtesy of Stanford 3D Scanning Repository. The Pegasus is courtesy of AIM@SHAPE Repository. The Horse is courtesy of 3DScanCo. The Angle is courtesy of Cyberware, Inc. The Spaceship is courtesy of INRIA Gamma team research database and the Warrior is courtesy of Evermotion.

References

1 Reeves W T. Particle systems—a technique for modeling a class of fuzzy objects. ACM Trans Graphic, 1983, 2: 91–108
2 Witkin A, Baraff D. Physically based modeling: principles and practice. In: Proceedings of SIGGRAPH'97 Course notes. 1997
3 Kolb A, Latta L, Rezk-Salama C. Hardware-based simulation and collision detection for large particle systems. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. 2004. 123–131
4 Drone S. Real-time particle systems on the GPU in dynamic environments. In: Proceedings of ACM SIGGRAPH 2007 Courses. San Diego, 2007. 80–96
5 Sims K. Particle animation and rendering using data parallel computation. In: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques. Dallas, 1990. 405–413
6 Owens J D, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware. Comput Graph Forum, 2007, 26: 80–113
7 Latta L. Building a million particle system. Lecture at the GDC, 2004
8 Kipfer P, Segal M, Westermann R. UberFlow: a GPU-based particle engine. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. 2004. 115–122
9 Venetillo J S, Celes W. GPU-based particle simulation with inter-collisions. Visual Comput, 2007, 23: 851–860
10 Glasner A S. Space subdivision for fast ray tracing. IEEE Comput Graph, 1984, 4: 15–22
11 MacDonald D J, Booth K S. Heuristics for ray tracing using space subdivision. Visual Comput, 1990, 6: 153–166
12 Samet H. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS. Addison-Wesley, 1990
13 Scherson I D, Caspary E. Data structures and the time complexity of ray tracing. Visual Comput, 1987, 3: 201–213
14 Benson D, Davis J. Octree textures. ACM Trans Graphic, 2002, 21: 785–790
15 DeBry D, Gibbs J, Petty D D, et al. Painting and rendering textures on unparameterized models. ACM Trans Graphic, 2002, 21: 763–768
16 Hormann K, Polthier K, Sheffer A. Mesh parameterization: theory and practice. In: Proceedings of ACM SIGGRAPH ASIA 2008 Courses. Singapore, 2008. 1–87
17 Lefebvre S, Dachsbacher C. TileTrees. In: Proceedings of the 2007 symposium on Interactive 3D Graphics and Games. Seattle, 2007. 25–31
18 Lefebvre S, Hornus S, Neyret F. GPU Gems 2: Octree Textures on the GPU. Addison-Wesley, 2005
19 Castro R, Lewiner T, Lopes H, et al. Statistical optimization of octree searches. Comput Graph Forum, 2008, 27: 1557–1566
20 Warren M S, Salmon J K. A parallel hashed octree N-Body algorithm. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. Portland, 1993. 12–21
21 Bloomenthal J. Polygonization of implicit surfaces. Comput Aided Geom D, 1988, 5: 341–355
22 Cohen-Steiner D, Alliez P, Desbrun M. Variational shape approximation. ACM Trans Graph, 2004, 23: 905–914
23 Haines E. Point in polygon strategies. Graphics Gems IV, 1994: 24–46
24 Lacoste J, Boubekeur T, Jobard B, et al. Appearance preserving octree-textures. In: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia, 2007. 87–93
25 NVIDIA. NVIDIA CUDA Programming Guide 1.1. 2007
26 Green S. Particle Simulation using CUDA. NVIDIA Whitepaper, 2010
27 Zhou K, Gong M M, Huang X, et al. Data-parallel octrees for surface reconstruction. IEEE Trans Vis Comput Gr, 2011, 17: 669–681
28 Deng H, Zhang L Q, Mao X C, et al. Fast and dynamic generation of linear octrees for geological bodies under hardware acceleration. Sci China Earth Sci, 2010, 53: 113–119
29 Yang Z W, Seo Y H, Kim T W. Adaptive triangular-mesh reconstruction by mean-curvature-based refinement from point clouds using a moving parabolic approximation. Comput Aid Des, 2010, 42: 2–17