The Visual Computer
https://doi.org/10.1007/s00371-019-01716-4

ORIGINAL ARTICLE

Using pseudo voxel octree to accelerate collision between cutting tool and deformable objects modeled as linked voxels

Shiyu Jia[1] · Weizhong Zhang[1] · Zhenkuan Pan[1] · Guodong Wang[1] · Xiaokang Yu[1]

Abstract
For deformable objects modeled as a uniform grid of voxels connected by links, an octree for the voxels is constructed. Cutting is performed by disconnecting links swept by the cutting tool and reconstructing cut surface mesh using the dual contour method. The cubes of the voxel octree are not directly used because their edges generally do not remain straight when the objects deform. Instead, the voxel octree is used to mark active voxels and links and is therefore called "pseudo." Voxels and links located in the interiors of voxel octree cubes are deactivated. For collision between the cutting tool and the deformable objects, only active voxels and links are considered. Then, voxel octree cubes with newly cut links on their boundaries are recursively subdivided, and new voxels and links are activated accordingly. These algorithms are implemented with multithreading techniques. Simulation tests show that when compared to previous methods using a uniform grid of voxels, our voxel octree method can increase cutting tool collision speed by 11–96% and can increase overall simulation speed by 7–43%.

Keywords Deformable object · Physically based modeling · Interactive cutting · Octree

1 Introduction

To simulate cutting of deformable objects, traditional methods split each component element swept by the cutting tool into several smaller elements. However, this process may create very small or degenerated elements that make deformation calculation numerically unstable. One type of methods to get around this problem embeds a uniform grid of voxels with fine resolution inside an octree mesh of hexahedral elements with coarse resolution. Deformation is only applied to the hexahedral elements. Adjacent voxels are connected by links. During cutting, links swept by the cutting tool trajectory are disconnected and related hexahedral elements are adaptively refined and duplicated. The cut surfaces are reconstructed from disconnected links using either the splitting cube method or the dual contour method.

For this type of methods, one of the major bottlenecks for simulation performance is the collision between the cutting tool and the deformable objects. The main contribution of this paper is using a voxel octree to accelerate this type of collision, while maintaining surface details and deformation accuracy of the deformable objects. The voxel octree is used to mark active voxels and links that will participate in the collision processing. Voxel octree cubes with newly cut links on their boundaries are recursively subdivided, and new voxels and links are activated accordingly. The voxel octree has no restriction on the level differences between adjacent cubes and is therefore easier to be constructed and subdivided than a restricted octree.

2 Previous work

Compared with simulating fractures in rigid bodies [1–3], simulating cuts in deformable objects is much more difficult, especially for simulations with real-time requirements, such as surgical simulations. Brief descriptions of previous deformable cutting methods are given in this section. Most details can be found in the survey by Wu et al. [4].

The traditional *element splitting method* [5, 6] splits each element swept by the cutting tool into smaller elements.

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00371-019-01716-4) contains supplementary material, which is available to authorized users.

✉ Weizhong Zhang
zhangwz_01@aliyun.com

[1] College of Computer Science and Technology, Qingdao University, Qingdao 266071, Shandong, People's Republic of China

Published online: 20 June 2019

This may result in degenerated or very small elements that decrease the numerical stability of deformation. To overcome this problem, several advanced cutting methods have been proposed.

The *virtual node method* [7–10] embeds cut fragments in virtual copies of the original uncut elements. The *extended finite element method* (*XFEM*) [11–13] uses enriched shape functions to model displacement discontinuities across the cut surfaces. The cutting method based on the *meshfree deformation method* (*MDM*) [14, 15] inserts new nodes around the cut surfaces and updates visibility between nodes occluded by the cut surfaces. The *position based dynamics* has been used to simulate deformable cutting in combination with voxel representations [16], or geometric metaballs and MDM [17].

The cutting method based on *adaptive octree mesh* uses an octree mesh of hexahedral elements for deformation. Hexahedral elements intersected by the cutting tool are recursively subdivided. Surface meshes for rendering and collision are embedded in the octree mesh. Jeřábková et al. [18] simply removed the finest level elements intersected by the cutting tool and reconstructed the cut surfaces using the level set method. Seiler et al. [19] embedded a tetrahedral mesh inside the octree mesh. Tetrahedrons intersected by the cutting tool were removed, and octree nodes were split according to tetrahedral mesh connectivity. The cut surfaces were formed using clipping and Delaunay triangulation. Dick et al. [20] embedded a uniform grid of voxels inside the octree mesh. Adjacent voxels are connected by links, and links swept by the cutting tool were disconnected. The cut surfaces were reconstructed from disconnected links using the splitting cube method [15]. Wu et al. [21] proposed an efficient collision detection method for the cutting method in [20] and changed the cut surface reconstruction method to the dual contour method [22]. Jia et al. [23] designed a parallel framework utilizing both CPU and GPU to accelerate deformable cutting simulation based on the methods proposed in [20] and [21].

3 Methods

Our deformable cutting method is based on the linked voxel model and the adaptive octree mesh proposed in [20] and [21]. The major performance bottlenecks for this type of methods are: deformation; inter-object collision and object self collision (referred to simply as "*object collision*" from now on); and the collision between the cutting tool and the deformable objects (referred to simply as "*cutting tool collision*" from now on). In this paper, a voxel octree is used to accelerate the cutting tool collision. The easiest way to utilize this voxel octree one may think of is to treat the edges of the voxel octree cubes as coarser resolution links and check collisions between the cutting tool and these edges. However, this will not work because these edges may not remain straight when the objects deform. Instead, our method uses the voxel octree to mark active voxels and links that will participate in the cutting tool collision. Voxel octree cubes with newly cut links on their boundaries will be recursively subdivided, and new voxels and links are activated accordingly. Therefore, we call this voxel octree "*pseudo*". To further increase the simulation performance, the parallel framework from [23] is modified to accommodate our new method.

The modified deformable cutting method will be described in Sect. 3.1, and the modified parallel framework will be described in Sect. 3.2.

3.1 Deformable cutting method using a pseudo voxel octree

Deformable objects are internally represented by voxels, with adjacent voxels connected by links (Fig. 1). These voxels are embedded in an octree of deformable hexahedral elements (Fig. 2). The most popular method for deformation is the *finite element method* (*FEM*). The original method proposed in [20] and [21] uses a uniform grid of voxels, as shown in Fig. 1a. Links intersected by the original object surface or the cutting tool trajectory are marked as disconnected. The object surface mesh and cut surface mesh are constructed using the intersection information of the disconnected links. Our method constructs an octree on top of this uniform voxel grid, as shown in Fig. 1b. The vertices of each octree cube are voxels and the edges of each octree cube consist of links. To avoid confusion, this octree will be referred to as the "*voxel octree*", while the octree of deformable hexahedral elements will be referred to as the "*deformation octree*".

The size of an octree cube is 2^L times the size of a voxel. L is called the *level of the octree cube*. Two parameters for each material need to be specified to control the voxel octree structure: a *maximum internal level* and a *maximum inter-material boundary level*. The voxel octree is constructed iteratively from level 0 in level ascending order. For each potential octree cube with level L, if all $(2^L+1)^3$ voxels it contains exist and are inside, and all $3 \times 2^L \times (2^L+1)^2$ links it contains exist and are connected, then it is further tested for level restriction. Otherwise it is rejected. There are two cases for the level restriction test. If all voxels the cube contains belong to the same material, then the cube passes the test if L is no more than the maximum internal level of the material. If all voxels the cube contains belong to more than one material, then the cube passes the test if L is no more than the maximum inter-material boundary levels of all relevant materials. The octree cube is constructed if it passes the level restriction test. In the example shown in Fig. 1b, there are two materials. The

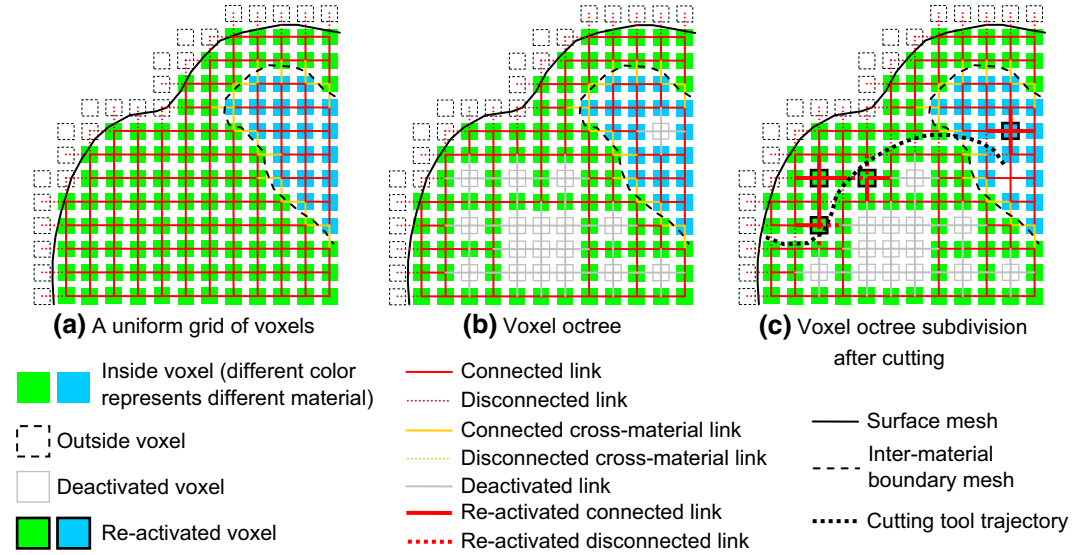

Fig. 1 Voxel representation of deformable objects

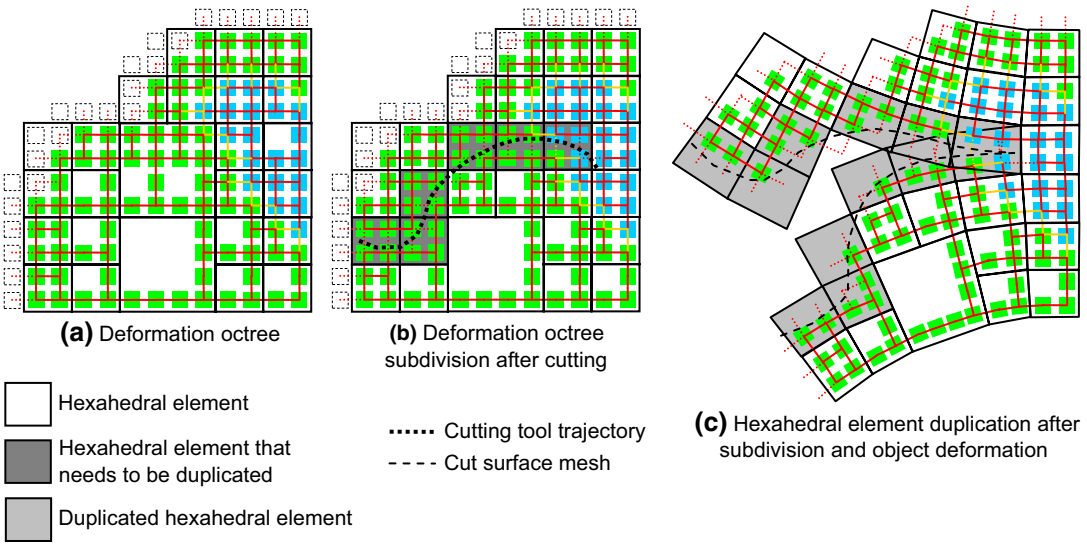

Fig. 2 Octree of hexahedral elements for deformation. (Symbols with the same meaning as in Fig. 1 are not annotated)

maximum internal level is 2 for the green material and 1 for the blue material. The maximum inter-material boundary level is 0 for both materials.

This voxel octree is used to mark active voxels and links. Deactivated voxels and links are ignored by the cutting tool collision process, but they are still used normally in deformation and object collision. Note that Fig. 1 shows 2D representations. For an actual 3D voxel octree, the process to mark active voxels and links consists of the following steps:

(1) Deactivate all voxels and links.
(2) For each connected link, if it is not completely surrounded by voxel octree cubes with levels higher than 0, then it is activated.
(3) Activate all links on edges of voxel octree cubes with levels higher than 0.
(4) Activate all voxels connected to at least one active link.

Note that, step (2) creates a "protective outer shell" around each material region, and step (3) creates a "protective frame" around each voxel octree cube with a level higher than 0. These are created to reduce the possibility of the cutting tool intersecting an inside inactive link without

intersecting any active links on the boundary, for either a material region or a voxel octree cube.

For the cutting tool collision, positions of active voxels are first calculated, and then, active links intersected by the cutting tool are detected and disconnected. To speed up the intersection detection, a spatial hash table is constructed for the deformation octree as proposed in [21]. Each hexahedral element in the deformation octree is added to hash table entries corresponding to the spatial cells its AABB covers. This spatial hash table is also used in the object collision, but that is beyond the scope of this paper. An approximate OBB for the cutting tool trajectory is constructed. All hexahedral elements in hash table entries corresponding to spatial cells covered by this OBB are marked. Then, all connected and active links with at least one connected voxel belonging to a marked hexahedral element are marked. Finally, these marked links are checked for intersection with the cutting tool.

Compared to previous methods with a uniform grid of voxels, our method needs an additional step after the intersection detection of links: recursive subdivision of voxel octree cubes affected by cutting. This recursive subdivision step is crucial in making both geometry accuracy and deformation accuracy of the cut surfaces the same as those from previous methods. As shown in Fig. 1c, each voxel octree cube with at least one newly disconnected link on its boundary (including edges and faces) and with a level higher than 0 is recursively subdivided until no cube has newly disconnected links on its boundary or level 0 is reached. During the subdivision process, the positions of voxels located on boundaries between sub-cubes are calculated, and links located on boundaries between sub-cubes are checked for intersection with the cutting tool. After the subdivision process finishes, voxels and links located on the edges of newly created voxel octree cubes are activated if they are currently inactive.

After the cutting tool collision, hexahedral elements in the deformation octree containing newly disconnected links are recursively subdivided (Fig. 2b). Then, hexahedral elements containing more than one connected voxel parts are duplicated, with each connected voxel part distributed to one duplicate (Fig. 2c).

The deformation octree is restricted, meaning that the level differences between adjacent octree cubes are no more than one. This makes it easier to deal with constraints imposed on octree vertices shared between octree cubes with different levels. The voxel octree has no such problem and is therefore not restricted.

This paper has no new contribution to the deformation algorithm. Figure 2 shows that the edges of voxel octree cubes with levels higher than 0 generally do not remain straight when the objects deform and therefore cannot be treated as line segments for intersection test with the cutting tool. This is the reason why our method uses the voxel octree indirectly to mark active voxels and links. For hexahedral elements using a linear interpolation function, the edges of a voxel octree cube can indeed remain straight if the cube is entirely contained in a hexahedral element. However, it is impossible for every voxel octree cube to be contained in a hexahedral element, because the voxel octree and the deformation octree are defined on two grids that are off by half a voxel.

3.2 Parallel implementations

Our deformable cutting method is implemented in a parallel framework shown in Fig. 3. This framework is derived from the framework designed in [23] with several modifications. As in [23], both CPU and GPU are utilized in the framework, and a single CPU thread is used specifically for dispatching GPU commands.

The explicit time integration method in [23] is now replaced with an implicit backward Euler time integration method. The resulting differential equations are solved using the preconditioned conjugate gradient method. Simulation tests show that the equation solution times are approximately doubled, but the time steps are nearly quadrupled compared to [23].

In [23], the voxel position update step is placed in the deformation stage, after velocities and positions of the deformation octree vertices are updated. In our framework, this step is moved to the collision stage, before intersection test between active links and the cutting tool, and in parallel to GPU's object collision step. The reason for this is that the voxel positions are only used in checking intersections between links and the cutting tool. Making the voxel position update step to run in parallel to GPU can potentially reduce the total simulation time.

The voxel octree data structure is only accessed by the CPU. It is only used for the cutting tool collision and remains solely in the CPU memory. As mentioned in Sect. 3.1, after intersection test between active links and the cutting tool, an additional step performing voxel octree subdivision is added. This step can potentially consume a large amount of time and its parallel implementation needs to be carefully designed.

Since the voxel octree is not restricted, subdivision of each voxel octree cube can almost be performed independently, except for shared boundaries (edges and faces). Therefore, our implementation first processes voxels and links on shared boundaries and then subdivides voxel octree cubes in parallel. The processing sub-steps are:

(1) Mark affected voxel octree cubes. ("Affected" means affected by cutting.) Quit if there are no affected voxel octree cubes.

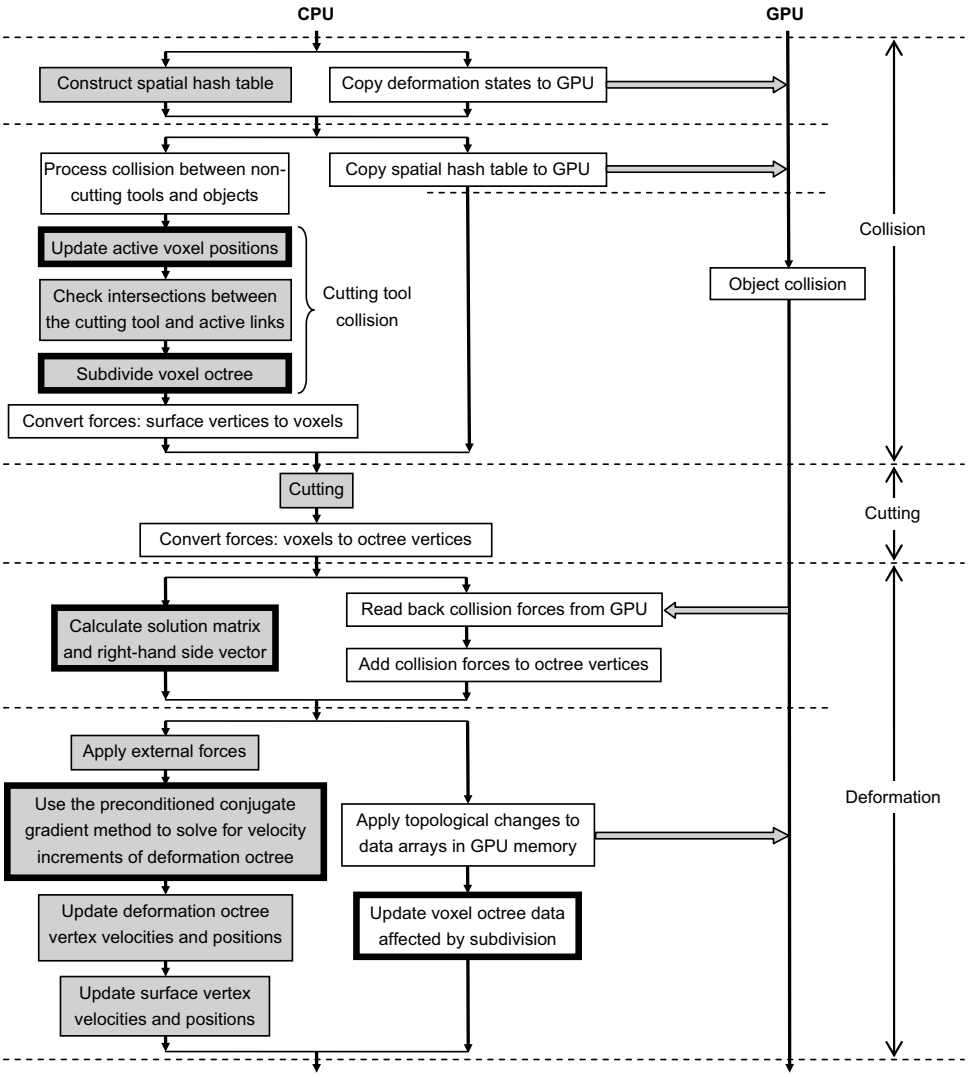

Fig. 3 Parallel framework of a single simulation frame. Dashed lines represent thread synchronization points. Text boxes represent processing steps. Steps in grey boxes are implemented using multi-threading techniques. Boxes with thick edges represent steps different from the previous method

(2) Mark voxels and links on the boundaries of affected voxel octree cubes.
(3) Calculate positions of inactive voxels that are also marked in sub-step (2).
(4) For each connected link that is also marked in sub-step (2), check if it intersects the cutting tool.
(5) Recursively subdivide each affected voxel octree cube.
(6) Go to sub-step (1).

Sub-steps (1) to (4) are implemented using lock-free multi-threading techniques.

Sub-step (1) processes each active and newly cut link and marks all voxel octree cubes surrounding the link with levels higher than 0. Each voxel octree cube has a special affected flag indicating if it is marked. Note that different threads processing different links may mark the same voxel octree cube multiple times, but there is no racing condition here because raising an already raised affected flag does not create any state conflicts.

Sub-step (2) processes each voxel octree cube marked in sub-step (1) and marks voxels and links on the voxel octree cube's boundary. Each voxel and link also has a special affected flag indicating if it is marked. Again, there is no racing condition here for the same reason as for sub-step (1).

For sub-step (5), each thread uses a mutex to get the next affected voxel octree cube it needs to subdivide. The subdivision itself has the following sub-steps:

(5-1) Calculate positions of voxels located on boundaries between sub-cubes.
(5-2) Check collisions between the cutting tool and links located on boundaries between sub-cubes.
(5-3) If current voxel octree cube has level 1, then activate all 27 voxels and 54 links it contains, mark current voxel octree cube as deleted, and return.
(5-4) Allocate storage space for seven new voxel octree cubes. Subdivide current voxel octree cube into eight

Table 1 Test models

Model	Voxel resolution	# Voxels	# Links	# Surface vertices	# Surface triangles	# Deformation octree vertices	# Deformation octree cubes	Time step (ms)
Bunny	101×78×100	242,930	635,744	70,810	141,616	999	557 (level 3) 11 (level 4)	4
Liver with tumor	100×62×88	185,259	483,596	55,736	111,448	820	498 (level 3) 2 (level 4)	5
Starfish	180×166×30	207,594	506,610	88,526	177,048	1056	574 (level 3)	3

sub-cubes and store them in the original storage space and the newly allocated storage space.

(5-5) Check if any of the eight sub-cubes need to be further subdivided. A sub-cube needs to be subdivided if any links on its boundaries are disconnected.

(5-6) Recursively subdivide sub-cubes that need to be subdivided.

After the voxel octree subdivision step, some related data structures need to be updated. Some of these data structures are not needed in the following cutting and deformation stages of the same simulation frame, and therefore, their updates can be delayed. These updates include the following:

(a) *Compaction of the voxel octree cube array* Our simulation system does not store level 0 voxel octree cubes, since they have no contribution to inactive voxels and links. Voxel octree cubes with levels higher than 0 are stored in a linear array. When a level 1 voxel octree cube is subdivided, it is temporarily marked with a deleted flag. After subdivision, the voxel octree cube array needs to be compacted to actually remove the voxel octree cubes marked with deleted flags.

(b) *Activation of voxels and links on edges of newly created voxel octree cubes* As mentioned in Sect. 3.1, this creates a "protective frame" around each newly created voxel octree cube.

(c) *Construction of the active voxel index array* The indices of active voxels are stored in a linear array to facilitate parallel implementations. This array needs to be reconstructed after voxel octree subdivision.

Notice that while deformation calculation is being performed, the GPU command dispatch thread is updating topological changes to GPU data structures. Simulation tests show that the former usually takes much longer time than the latter. Therefore, to increase the simulation performance, the task to update the data structures that are affected by voxel octree subdivision but not needed by cutting and deformation is assigned to the GPU command dispatch thread, after the step that updates topological changes to GPU data structure, and in parallel to the deformation steps.

4 Simulation results and analyses

Our simulation software is written in C++ for CPU and OpenCL for GPU. It runs on a PC with an Intel Core i5-3450 CPU (four cores, 3.1 GHz, Max Turbo 3.5 GHz), 8 GB RAM, an AMD Radeon R9 380 GPU with 4 GB Video RAM, one Phantom Premium 6DOF haptic device and one Phantom Desktop haptic device. The operating system is Windows 7 Ultimate 64bit. A file containing incomplete source codes just enough to illustrate the algorithms in this paper is provided in the supplemental materials.

Table 1 shows the three models used for our simulation tests. Time steps for the implicit backward Euler time integration are also included in the table.

The maximum inter-material boundary level for the voxel octree is set to 0 for all models, while the maximum internal level for the voxel octree varies from 1 to the maximum allowable value for each model. The numbers of voxel octree cubes with different maximum internal levels are shown in Table 2. For the rest of this paper, unless specifically stated, the maximum internal level of each model is set to the maximum allowable value (4 for the bunny and the liver with tumor, 3 for the starfish).

4.1 Single-frame cutting tests

In this suit of tests, the simulation is only run for a single frame. The cutting tool is positioned at one side of the model at the beginning of the frame and moves instantaneously to the other side of the model at the end of the frame. The cut surfaces resulted from these single-frame cutting tests are shown in Fig. 4.

This test is run for ten times for each model, and the execution times of simulation steps related to the cutting tool collision are averaged. These steps are: voxel position update, cutting tool intersection test and voxel octree

Table 2 Number of voxel octree cubes under different maximum internal levels for the test models

Model	Max internal level 1	Max internal level 2	Max internal level 3	Max internal level 4
Bunny	21,580 (level 1)	3580 (level 1), 2250 (level 2)	3580 (level 1), 722 (level 2), 191 (level 3)	3580 (level 1), 722 (level 2), 103 (level 3), 11 (level 4)
Liver with tumor	15,002 (level 1)	3754 (level 1), 1406 (level 2)	3754 (level 1), 662 (level 2), 93 (level 3)	3754 (level 1), 662 (level 2), 77 (level 3), 2 (level 4)
Starfish	15,248 (level 1)	4952 (level 1), 1287 (level 2)	4952 (level 1), 1015 (level 2), 34 (level 3)	

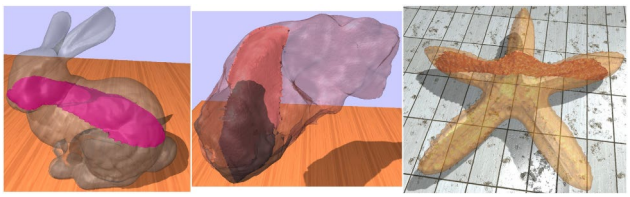

Fig. 4 Model cut surfaces for the single-frame cutting tests

The cutting tool collision total includes all three steps for our method, and only the voxel position update step and the cutting tool intersection test step for the previous method. Although the speedup from using the voxel octree is impressive for the voxel position update time and the cutting tool intersection test time, it is somewhat limited for the total time, ranging from 1.1 for the starfish to 1.96 for the bunny. The reason is that the previous method does not have the voxel octree subdivision step. Therefore, the speedup gained from the voxel position update and the cutting tool intersection test is offset by the addition of the voxel octree subdivision. Compared to the other two models, the starfish has a lower maximum internal level, and therefore, its ratio between the number of voxel octree cubes affected by cutting and the number of links cut along the cutting tool trajectory is higher. This results in a higher ratio between the voxel subdivision. Table 3 shows the averaged execution times of these steps under different numbers of threads. The numbers in the parentheses are speedup factors relative to single-threaded implementations. The thread scaling is good for all three steps.

Table 3 Averaged execution times (in milliseconds) under different numbers of threads for the single-frame cutting tests

Model	Voxel position update			Cutting tool intersection test			Voxel octree subdivision		
	1 *T*	2 *T*s	3 *T*s	1 *T*	2 *T*s	3 *T*s	1 *T*	2 *T*s	3 *T*s
Bunny	3.95	2.13 (1.85×)	1.66 (2.38×)	78.0	41.1 (1.90×)	28.2 (2.77×)	87.5	50.5 (1.73×)	37.4 (2.34×)
Liver with tumor	4.03	2.24 (1.80×)	1.68 (2.40×)	107	56.1 (1.91×)	37.9 (2.82×)	63.7	36.8 (1.73×)	28.0 (2.28×)
Starfish	4.66	2.63 (1.77×)	1.94 (2.40×)	38.9	20.7 (1.88×)	14.7 (2.65×)	37.3	21.4 (1.74×)	16.9 (2.21×)

"*T*" represents "thread." The numbers in the parentheses are unitless speedup factors relative to single-threaded implementations

Table 4 Averaged execution times (in milliseconds) with and without the voxel octree for the single-frame cutting tests

Model	Voxel position update			Cutting tool intersection test			Cutting tool collision total		
	Without voxel octree	With voxel octree	Speed ratio	Without voxel octree	With voxel octree	Speed ratio	Without voxel octree	With voxel octree	Speed ratio
Bunny	4.96	1.66	2.99	127	28.2	4.50	132	67.2	1.96
Liver with tumor	3.62	1.68	2.15	104	37.9	2.74	108	67.6	1.60
Starfish	3.65	1.94	1.88	33.5	14.7	2.28	37.2	33.6	1.11

Three threads are used for all simulations. Speed ratio is between simulation with the voxel octree and simulation without the voxel octree

Table 4 shows comparisons between the three-threaded implementation of our method using the voxel octree and that of the previous method [23] not using the voxel octree.

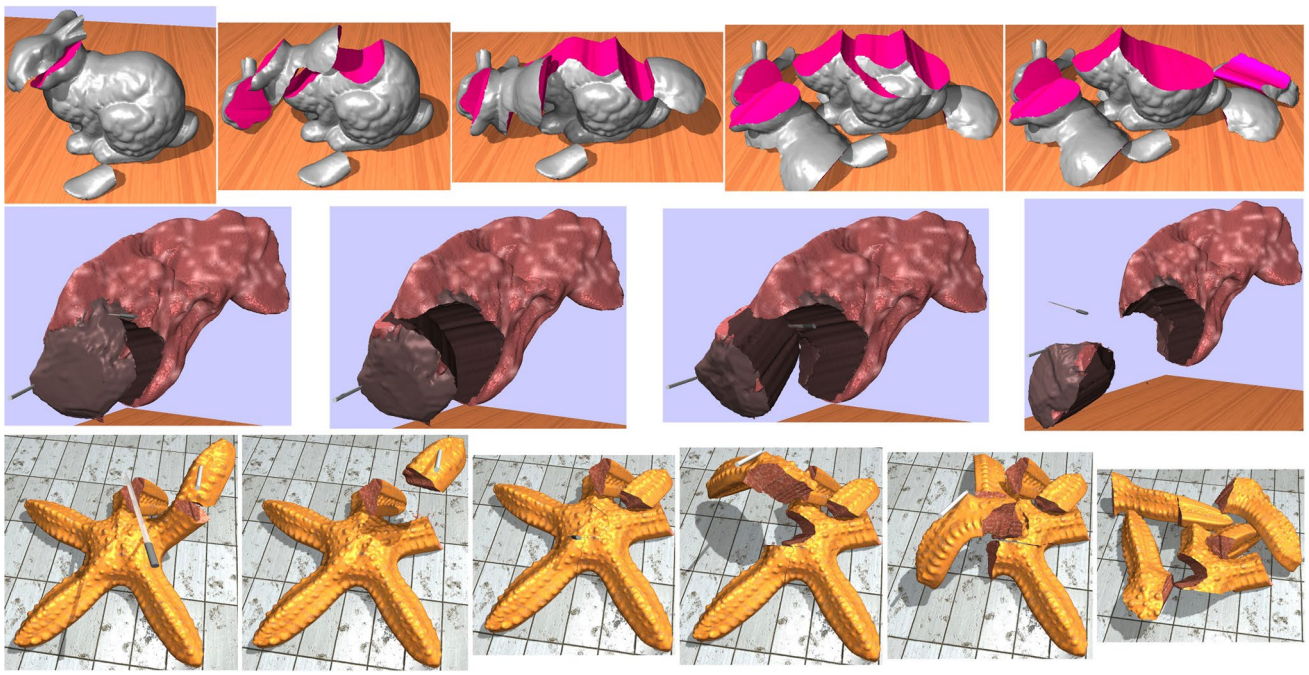

Fig. 5 Complex simulation tests for the bunny, the liver with tumor and the starfish models

octree subdivision time and the total time of the previous method, and in turn a smaller speedup for the total time.

4.2 Complex cutting and deformation tests

In this suit of tests, each model undergoes a complex simulation test scenario while user interacts with it using both a poking tool and a cutting tool. Screen captures of these simulation tests are shown in Fig. 5. Video files showing the entire simulation sequences are provided in the supplemental materials. For each model, movements of the interaction tools are recorded once and later played back for each test.

Figure 6 shows the simulation FPS (frames per second) as functions of the simulation time under different numbers of threads assigned to the cutting tool collision. The numbers of threads assigned to other simulation steps (object collision, cutting and deformation) remain three for all cases. The results using the previous method without the voxel octree are also shown for comparison.

As shown in Fig. 6, the simulation FPS with three threads are significantly higher than those with one thread, showing the effectiveness of our multi-threaded implementations. However, when compared to the results using the previous method without the voxel octree, the simulation FPS with three threads have only moderate increases at the beginning of the simulation tests, about 43% for the bunny model, 24% for the liver with tumor model and 15% for the starfish model. At the end of the simulation tests, the increases are changed to about 17% for the bunny model, 29% for the liver with tumor model and 7% for the starfish model. As more cuts are performed on the deformable objects, more voxel octree cubes are subdivided and more voxels and links are activated. This in turn lowers the advantages of our method over the previous method. In the extreme case, all voxels and links are activated and our method is essentially the same as the previous method. This explains the lowered increases for the bunny and the starfish models at the end. For the liver with tumor model, the FPS increase at the end is actually a little higher. This anomaly can be attributed to the cutting tool trajectory. In this case, the cutting tool mostly cuts into the boundary regions between the liver and the tumor. Since the maximum inter-material boundary level is set to 0, the number of voxel octree cubes with levels higher than 1 is small in the boundary regions; thus, the number of newly activated voxels and links due to cutting is small. Therefore, the performance reduction in the cutting tool collision does not outweigh the performance reduction in an entire simulation frame.

Figure 7 shows the simulation FPS as functions of the simulation time under different maximum internal levels for the voxel octree. The numbers of threads are three for all cases. Again, the results using the previous method without the voxel octree are shown for comparison.

As shown in Fig. 7, the simulation FPS have significant increases when going from not using the voxel octree to using a voxel octree with maximum internal level 1. This is also true when going from maximum internal level 1 to maximum internal level 2. After that, increasing the maximum

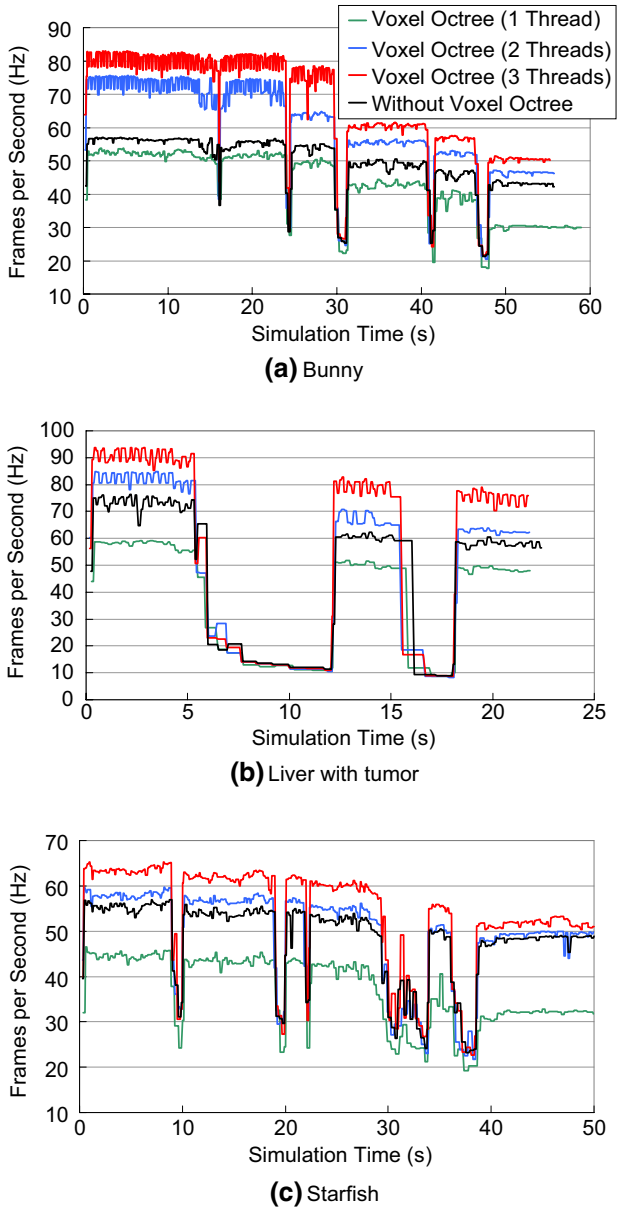

Fig. 6 Simulation FPS as functions of the simulation time under different numbers of threads

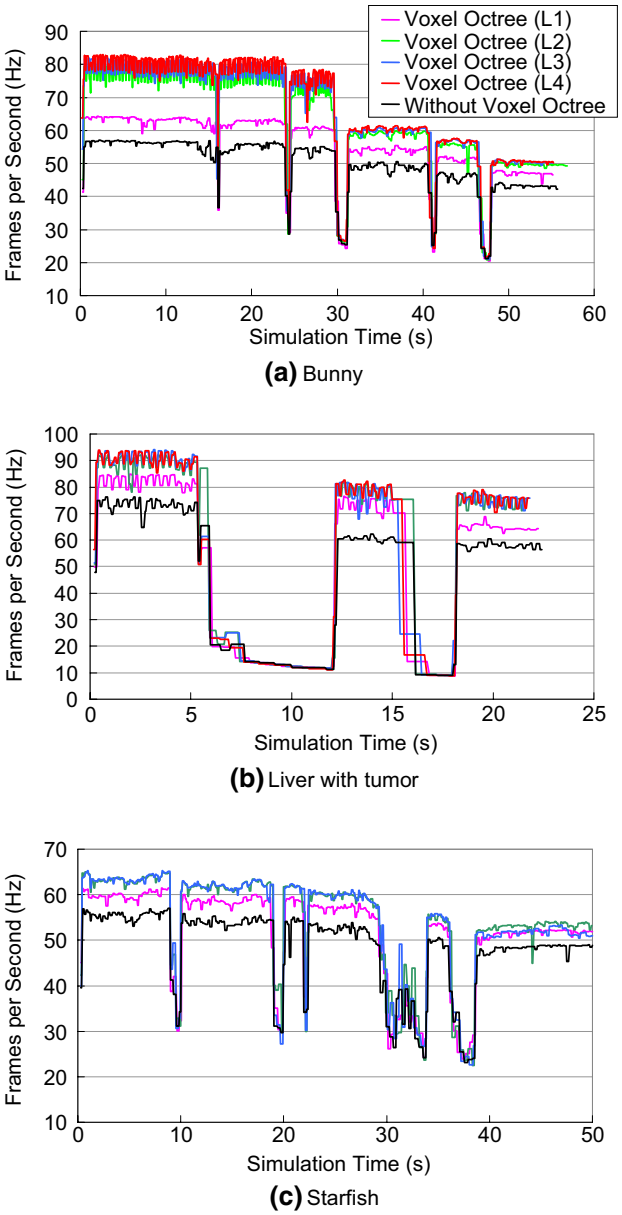

Fig. 7 Simulation FPS as functions of the simulation time under different maximum internal levels for the voxel octree. "L#" represents "Maximum internal level #"

internal level only increases the simulation FPS slightly, and in some cases even decreases the simulation FPS slightly. This is because the number of voxel octree cubes in a certain level drops as the level increases (see Table 2); thus, diminishing returns kick in when a certain level is reached.

Looking at Figs. 6 and 7 again, we can see that the simulation FPS increase in our method relative to the previous method only exist in the non-cutting periods, i.e., the time periods during which the cutting tool does not cut into the deformable objects. During cutting periods, represented by valleys in Figs. 6 and 7, the simulation FPS of our method are about the same as those of the previous method. This problem needs further analyses.

Figure 8 shows the execution times of the cutting tool collision total, the cutting tool collision total minus the voxel octree subdivision (the same as the voxel position update plus the cutting tool intersection test), and the voxel octree subdivision under different numbers of threads for the bunny model. The results using the previous method without the voxel octree are yet again shown for comparison.

As shown in Fig. 8, the combined execution time of the voxel position update and the cutting tool intersection test

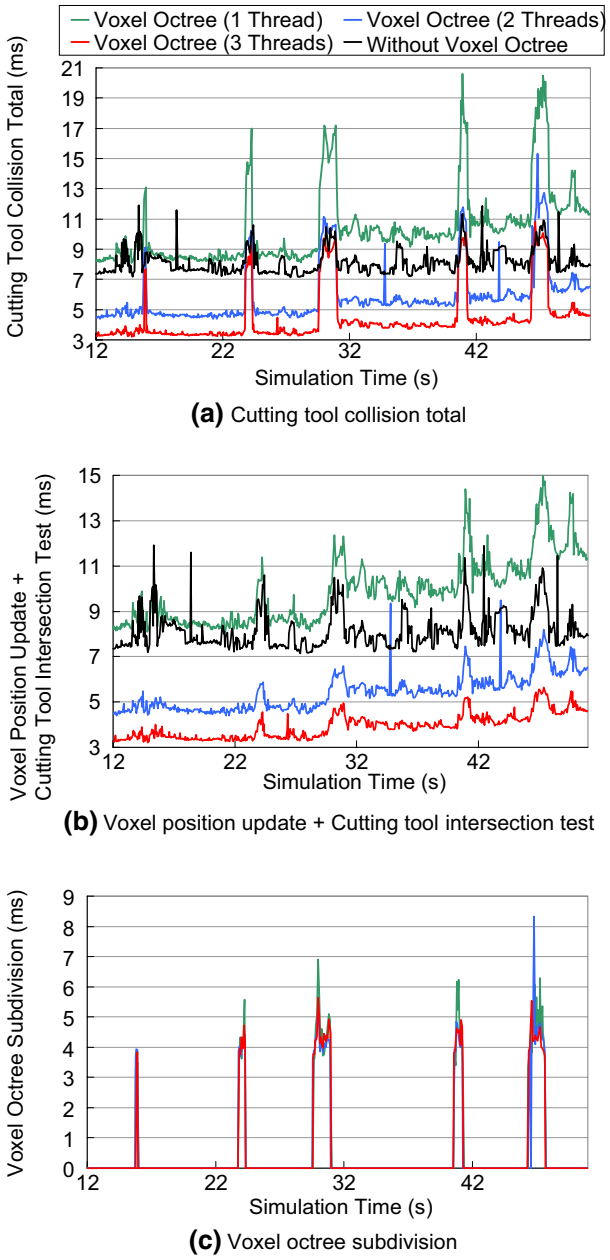

Fig. 8 Execution times of cutting tool collision steps as functions of the simulation time under different numbers of threads for the bunny model

has good thread scaling, in both non-cutting periods and cutting periods. With three threads, it is also significantly lower than that of the previous method. However, the voxel octree subdivision time has poor thread scaling in cutting periods, although it is zero in non-cutting periods. Adding the voxel octree subdivision time nearly cancels out the time reduction from the voxel position update and the cutting tool intersection test relative to the previous method. This is why the simulation FPS of our method are almost the same as those of the previous method in cutting periods.

However, the single-frame cutting tests show that the voxel octree subdivision has good thread scaling. Why is its thread scaling poor in the complex cutting tests?

In the single-frame cutting tests, the cutting tool moves from one side of the model to the other side in a single frame. In the complex cutting tests, the cutting tool is controlled by the user and moves much slower. Therefore, the number of voxel octree cubes to be subdivided in a single frame in the complex cutting tests is much smaller than that of the single-frame cutting tests. Our multi-threaded implementation of the voxel octree subdivision has significant overheads. It needs to first process voxels and links on shared boundaries before it can subdivide the voxel octree cubes in parallel. When the number of voxel octree cubes to be subdivided is low, the time saved due to parallel subdivision cannot offset the overheads. This is the reason why the thread scaling for the voxel octree subdivision is good in the single-frame cutting tests, but poor in the complex cutting tests. We also constructed figures similar to Fig. 8 for the liver with tumor model and the starfish model and came to the same conclusion. To save space, these figures are not shown in this paper.

5 Conclusions and future work

In this paper, a pseudo voxel octree is used to accelerate collision between a cutting tool and deformable objects modeled as voxels connected by links. Voxels and links in the interiors of voxel octree cubes are deactivated, and only active voxels and links are considered. Voxel octree cubes with newly cut links on their boundaries are recursively subdivided, and new voxels and links are activated accordingly. Due to this recursive subdivision, the cut surface details are the same as those using previous methods without the voxel octree. Multi-threading techniques are used to implement our method.

In the single-frame cutting tests, our method shows good thread scaling for all three steps of the cutting tool collision: voxel position update, cutting tool intersection test and voxel octree subdivision. Compared to previous methods without the voxel octree, which do not have a voxel octree subdivision step, our method is still 11–96% faster.

In the complex cutting tests, our method has good thread scaling for the voxel position update step and the cutting tool intersection test step, and the overall simulation speed is 7–43% faster than those using previous methods without the voxel octree in non-cutting periods. However, in cutting periods, the thread scaling is almost non-existent for the voxel octree subdivision step due to overheads and low

parallel workload. Thus, the overall simulation speed in cutting periods has no improvements over those using previous methods without the voxel octree.

In the future, we plan to turn the pseudo voxel octree into a real octree, i.e., making edges of voxel octree cubes straight during deformation so that they can be treated as coarser resolution links. Theoretically, this can make the cutting tool collision even faster. However, this requires aligning the voxel octree and the deformation octree in the same grid and necessitates a complete redesign of current deformable object models as well as deformation, cutting and collision algorithms. We also plan to use XFEM for deformation octree cubes containing cut surfaces to achieve better deformation accuracy near cut surfaces.

Funding This study was funded by the National Key Technology Support Program of China during the Twelfth Five-year Plan Period (Grant Number 2013BAI01B03).

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Zhu, Y., Bridson, R., Greif, C.: Simulating rigid body fracture with surface meshes. ACM Trans. Gr. **34**(4), 150 (2015)
2. Hahn, D., Wojtan, C.: High-resolution brittle fracture simulation with boundary elements. ACM Trans. Gr. **34**(4), 151 (2015)
3. Hahn, D., Wojtan, C.: Fast approximations for boundary element based brittle fracture simulation. ACM Trans. Gr. **35**(4), 104 (2016)
4. Wu, J., Westermann, R., Dick, C.: A survey of physically based simulation of cuts in deformable bodies. Comput. Gr. Forum **34**(6), 161–187 (2015)
5. Courtecuisse, H., Allard, J., Kerfriden, P., Bordas, S.P.A., Cotin, S., Duriez, C.: Real-time simulation of contact and cutting of heterogeneous soft-tissues. Med. Image Anal. **18**(2), 394–410 (2014)
6. Paulus, C.J., Untereiner, L., Courtecuisse, H., Cotin, S., Cazier, D.: Virtual cutting of deformable objects based on efficient topological operations. Vis. Comput. **31**(6), 831–841 (2015)
7. Molino, N., Bao, Z., Fedkiw, R.: A virtual node algorithm for changing mesh topology during simulation. ACM Trans. Gr. **23**(3), 385–392 (2004)
8. Sifakis E, Der K G, Fedkiw R. Arbitrary cutting of deformable tetrahedralized objects. In: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, August 2007, pp. 73–80
9. Wang Y, Jiang C, Schroeder C, Teran J. An adaptive virtual node algorithm with robust mesh cutting. In: Proceedings of the 2014 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, July 2014, pp. 77–85
10. Jia, S., Zhang, W., Yu, X., Pan, Z.: CPU-GPU mixed implementation of virtual node method for real-time interactive cutting of deformable objects using OpenCL. Int. J. Comput. Assist. Radiol. Surg. **10**(9), 1477–1491 (2015)
11. Jeřábková, L., Kuhlen, T.: Stable cutting of deformable objects in virtual environments using xfem. IEEE Comput. Gr. Appl. **29**(2), 61–71 (2009)
12. Kaufmann, P., Martin, S., Botsch, M., Grinspun, E., Gross, M.: Enrichment textures for detailed cutting of shells. ACM Trans. Gr. **28**(3), 50 (2009)
13. Turkiyyah, G.M., Karam, W.B., Ajami, Z., Nasri, A.: Mesh cutting during real-time physical simulation. Comput. Aided Des. **43**(7), 809–819 (2011)
14. Steinemann, D., Otaduy, M.A., Gross, M.: Splitting meshless deforming objects with explicit surface tracking. Gr. Models **71**(6), 209–220 (2009)
15. Pietroni, N., Ganovelli, F., Cignoni, P., Scopigno, R.: Splitting cubes—A fast and robust technique for virtual cutting. Vis. Comput. **25**(3), 227–239 (2009)
16. Berndt, I., Torchelsen, R., Maciel, A.: Efficient surgical cutting with position-based dynamics. IEEE Comput. Gr. Appl. **38**(3), 24–31 (2017)
17. Pan, J., Yan, S., Qin, H., Hao, A.: Real-time dissection of organs via hybrid coupling of geometric metaballs and physics-centric mesh-free method. Vis. Comput. **34**(1), 105–116 (2018)
18. Jeřábková, L., Bousquet, G., Barbier, S., Faure, F., Allard, J.: Volumetric modeling and interactive cutting of deformable bodies. Prog. Biophys. Mol. Biol. **103**(2/3), 217–224 (2010)
19. Seiler, M., Steinemann, D., Spillmann, J., Harders, M.: Robust interactive cutting based on an adaptive octree simulation mesh. Vis. Comput. **27**(6/8), 519–529 (2011)
20. Dick, C., Georgii, J., Westermann, R.: A hexahedral multigrid approach for simulating cuts in deformable objects. IEEE Trans. Vis. Comput. Gr. **17**(11), 1663–1675 (2011)
21. Wu, J., Dick, C., Westermann, R.: Efficient collision detection for composite finite element simulation of cuts in deformable bodies. Vis. Comput. **29**(6/8), 739–749 (2013)
22. Ju, T., Losasso, F., Schaefer, S., Warren, J.: Dual contouring of hermite data. ACM Trans. Gr. **21**(3), 339–346 (2002)
23. Jia, S., Zhang, W., Yu, X., Pan, Z.: CPU-GPU parallel framework for real-time interactive cutting of adaptive octree-based deformable objects. Comput. Gr. Forum **37**(1), 45–59 (2018)

Shiyu Jia received his Ph.D. in Mechanical Engineering from Yale University in 2001. His Ph.D. research was related to finite element analysis and computer simulation of composite material properties. He began working in College of Computer Science and Technology (former College of Information Engineering) of Qingdao University in 2003 and was appointed to associate professor in 2004. He has been working on deformation, collision, cutting simulation and haptic interaction of deformable objects since 2005. His research interests are computer graphics, haptic interaction and surgical simulation.

Weizhong Zhang received his Ph.D. in Computer Science from College of Mechanical and Electrical Engineering, Nanjing University of Aeronautics and Astronautics in 2007. He began working in College of Computer Science and Technology (former College of Information Engineering) of Qingdao University as professor in 2005. His research interests are computer vision, computer graphics, image processing and 3D image reconstruction.

Zhenkuan Pan received his Ph.D. in Dynamics and Control from Shanghai Jiao Tong University in 1992. Thereafter, he began working in Qingdao University until today. He served as associate head of College of Automation from 1992 to 1996, head of computing center from 1996 to 2001, assistant dean of College of Information Engineering from 1998 to 2002 and dean of College of Information Engineering (now College of Computer Science and Technology) from 2002 until today. He was appointed to professor in 1996. His research interests are multi-body system dynamics and control, surgical simulation, image processing and numerical analysis.

Guodong Wang received his B.Sc. and M.Sc. degrees in control technology and control engineering from Qingdao Science and Technology University, PR China, in 2001 and 2004, respectively, and his Ph.D. degree in pattern recognition and intelligent systems from Huazhong University of Science and Technology, in 2008. He is an Associate Professor in College of Computer Science and Technology at Qingdao University, China. His research interests include biometrics, image processing, intelligent video monitoring and analysis.

Xiaokang Yu received his Ph.D. from School of Computer Science and Technology, Shangdong University in 2012. He began working in College of Computer Science and Technology (former College of Information Engineering) of Qingdao University as lecturer in 2012. His research interests are computational geometry and computer graphics.