The Visual Computer
https://doi.org/10.1007/s00371-020-01856-y

ORIGINAL ARTICLE

Divided Voxels: an efficient algorithm for interactive cutting of deformable objects

Di Qi[1] · Nicholas Milef[2] · Suvranu De[1]

Abstract
Efficient algorithms that support dynamic topological updates are necessary for the simulation of progressive interactive cutting of deformable objects. Existing mesh-based techniques suffer from the generation of ill-shaped elements, whereas voxel grid-based methods require additional cut surfaces to be generated or the use of lookup tables for pre-computed cutting patterns. To overcome these limitations of existing methods, we propose a novel voxel-based topological operator, *divide*, which divides a voxel into two voxels identical to the original voxel's size by dynamically distributing its voxel elements (nodes, edges) into the newly divided voxels until the cutting of the original voxel is completed. The connectivity between the divided voxels and the neighbors of the original voxel is retained during the cut, and new connectivity between the adjacent divided voxels is generated to represent the continuity of the cut. As a result, the cut surface can be generated directly from the divided voxels on the fly, and the correspondence between the cut surface and the simulation voxels is maintained without any additional effort. We use several example problems to demonstrate the efficiency of our method and compare it with other existing approaches.

Keywords Interactive cutting · Progressive cutting · Voxel-based · Topological operator · Deformable objects

1 Introduction

The ability to interactively modify the geometry of soft deformable objects as a result of progressive cutting is a popular field of research in computer graphics [42]. There are numerous applications of interactive cutting including surgical procedure simulation, video games, and computer animations.

Cutting simulation is challenging due to its algorithmic and computational complexity. The simulation of cuts in a deformable object involves two major tasks: the modeling of the cut itself, which includes updating the geometrical and topological representation of the deformable object, and the simulation of the deformable body. Moreover, the capability of simulating progressive cutting of soft tissue is essential for many interactive applications such as virtual surgical simulation, where the user requires immediate feedback during a dissection operation. Specifically, the cut should occur progressively as the user moves a scalpel through the individual elements which need to be updated in real time and not after the cut is completed. This type of interactive cutting demands an efficient cutting algorithm that supports dynamic updates in deformable objects.

A significant volume of research focuses on mesh-based methods, in which cuts are modeled by splitting mesh elements along the cutting surfaces [3,5,23]. For instance, Bielser et al. presented a tetrahedral-based splitting method, where each cut tetrahedron is split into 17 smaller elements by inserting a vertex on each edge and face [5]. The subdivision algorithm is further improved to support progressive cutting using a state machine [3]. Although the cut surface can be accurately created, these remeshing methods lead to a large number of tetrahedra generated along the cut, which increases the computational complexity. Moreover, the mesh-based approaches are prone to generating ill-shaped elements, which lead to numerical inaccuracies

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00371-020-01856-y) contains supplementary material, which is available to authorized users.

✉ Di Qi
qid@rpi.edu; trudiqi@gmail.com

[1] Center for Modeling, Simulation and Imaging in Medicine, Rensselaer Polytechnic Institute, Troy, NY, USA

[2] Department of Computer Science and Engineering, Texas A&M University, College Station, TX, USA

Published online: 20 May 2020

[35] during deformation simulation [42] and visual artifacts [14]. To address this issue, the virtual node algorithm (VNA) [22] decouples the simulation domain and geometrical representation of the deformable object. Each cut tetrahedron is first subdivided into sub-elements to determine which portion is material. A triangulation of those sub-elements' boundary is then added into a triangle list for building the cut surface mesh; the triangulation across the neighboring sub-elements must be carefully checked to ensure consistency. For the simulation domain, a virtual copy of an element node (vertex) is created when the node has a "scoop" out of its one ring. Since at least one of the elements sharing the node must be completely cut, VNA therefore cannot handle partial cut and progressive cut within an element.

Recent works [2,7,11,20,21,27,30,33,34,41] address these computational complexity and stability issues by employing a structured voxel grid of hexahedral elements. The voxel grid structure allows more straightforward topological changes resulting from cuts while maintaining well-shaped elements. On the other hand, to render the surface of the deformable object including the additional surfaces generated by cutting, a surface mesh must be reconstructed from the voxels. In addition, to deform the surface according to the simulation performed on the underlying voxels, their correspondence needs to be established and maintained. Simulating cutting with this structure poses the problem of efficiently generating the cut surface and matching the new surface with the underlying voxel grid.

To resolve this, recent works [7,13,41] use "splitting cubes" [33] to model cuts in a deformable object by associating each voxel that is cut with a specific pre-defined cutting pattern of a voxel; the cutting surfaces are then reconstructed and mapped to the individual simulation voxels based on their cutting patterns recorded in a lookup table (LUT). However, in interactive cutting, each voxel is cut progressively and potentially can be cut in different ways, so the actual cutting pattern can only be confirmed when the cutting of the voxel is completed, contrary to what is necessary for these kinds of applications.

In this paper, we propose a novel voxel-based topological operator, *divide*, which can dynamically model a cut in a voxel to support partial cutting and progressive cutting. This operator divides a voxel into two voxels identical in size to the original voxel, by dynamically distributing the original voxel nodes and edges into the new divided voxels until the cutting of the original voxel is completed. The connectivity between the divided voxels and the neighbors of the original voxel is retained during the cut, and new connectivity between the adjacent divided voxels is constructed to represent the continuity of the cut. Since the topological change and connectivity of the cut are effectively represented by the new divided voxels, the cut surface can be generated directly from the divided voxels using the dual contouring algorithm [15], in which a surface mesh is constructed by connecting surface vertices in adjacent voxels sharing the edges with sign changes. The correspondence between the cut surface and the simulation voxels is retained without any additional effort.

To clearly focus our work on the efficient voxel-based cutting algorithm that can model partial cut and progressive cut within voxels, collision detection and response are not considered in this paper. However, any surface mesh-based collision detection method can be used and external force for collision response can be integrated into our algorithm.

The reminder of the paper is organized as follows. Related work is reviewed in Sect. 2. The voxel-based topological operator, *divide*, that enables progressive cutting is presented in Sect. 3. In Sect. 4, we describe cut surface meshing and updates during progressive cutting based on the divided voxels. Deformation simulation and the dynamic physics model updates are introduced in Sect. 5. Finally, some examples are presented to demonstrate the computational efficiency of the proposed technique in Sect. 6, followed by concluding remarks and future directions in Sect. 7.

2 Related work

Cutting of a deformable object involves updating the geometrical and topological representations of the object while computing its deformation at the same time. A detailed review of the field can be found in recent surveys [38,42]. In the following paragraphs, we briefly summarize the two major classes of algorithms for modeling cuts in deformable objects using mesh-based and voxel grid-based approaches.

2.1 Remeshing-based methods

Deformation simulation using a tetrahedral mesh has been widely employed. The rendered surface mesh can be directly obtained from the tetrahedral mesh by determining the triangle faces on the surface. Modeling of cuts in such tetrahedral meshes is primarily performed by mesh subdivision and remeshing operations. Simple and fast remeshing techniques such as element deletion [6] or splitting along element faces [28] avoid ill-shaped elements, but they result in jagged cutting surfaces. Cuts can be accurately represented by means of element refinement (i.e., subdividing a cut tetrahedron into several tetrahedra [3,4,23,31]), element snapping (i.e., snapping vertices to the cutting surface [29]), or a combination of both [37]. Nevertheless, it is necessary for these methods to prevent generation of ill-shaped elements with well-conditioned Jacobians [42].

To address this issue, the virtual node method [22] creates replicas of an element that is cut, and embeds each distinct component of the element into a unique replica. This method

is further improved by [36,39] for arbitrarily shaped and high-resolution cut surfaces. Although the virtual node algorithm avoids the creation of ill-shaped elements and can be processed in real time [12], it cannot address partial cuts which commonly occur during progressive cutting. Moreover, special attention must be paid to find a topologically consistent mapping between the high-resolution surface representation and the tetrahedral simulation domain.

Finite element methods (FEMs) are commonly employed in mesh-based approaches for simulating deformable bodies. In standard FEM, the displacement within an element is interpolated at the element's nodes based on shape functions, which cannot capture discontinuities introduced by cuts [42]. Polyhedral FEM and extended FEM (XFEM) are two FEM-based methods specialized for simulating cuts in deformable objects. Polyhedral FEM avoids the remeshing process in standard FEM by directly working on general convex and/or concave polyhedral elements, while shape functions are defined on the polyhedral domain [40]. XFEM models discontinuities introduced by cuts with introducing discontinuous enrichment functions and duplicating the degrees of freedom (DOFs) at the nodes of the original elements [10]. Recently, a robust cutting algorithm based on XFEM has been proposed [17]. This method presents the construction of specialized quadrature rules for each dissected element and solves the problem of ill-conditioned matrices using a method that constrains non-contributing DOFs, making it particularly suitable for fine structural cutting. However, this approach cannot simulate progressive cutting where the simulation elements are required to be completely dissected. Kaufmann et al. proposed enrichment textures for detailed cutting of shells [16]. They propose a harmonic enrichment method, which uses only one unified type of enrichment functions to deal with partial cuts and progressive cuts. In general, XFEM-based methods [10,16,24] focus on the updates of shape functions to model discontinuities for numerical simulation. When modeling a cut surface, they usually employ a standard remeshing method; this usually involves non-trivial mesh remeshing.

2.2 Voxel grid-based methods

Using a structured grid of voxels is an alternative to unstructured tetrahedral meshing; it allows topological changes introduced by cuts while maintaining well-shaped elements [34]. In addition, it is straightforward to create a mesh hierarchy from a voxel grid for coarsening and refinement of discretization [11]. However, to render the surface of the deformable object, including the cut surface, the grid structure requires a surface mesh representation. Moreover, to deform the surface in accordance with the underlying voxels, correspondence must be established and maintained between the two. When incorporating cuts into a deformable object represented by voxels, multiple surface vertices are inserted into the same voxel. The challenge is to connect these vertices to represent the cut surface and update the correspondence between surface vertices and voxel nodes.

To circumvent this, Jerabkova et al. [11] model a cut by voxel removal at the finest level, and the surface mesh along the cut is reconstructed using the marching cubes algorithm [19]. This method, however, makes it difficult to generate a surface that accurately aligns with the cut. Dick et al. [7], on the other hand, employ an octree grid that is adaptively refined along a cut. To construct the cut surface and compute its correspondence to the simulation grid, they employ the splitting cubes algorithm [33]. In this algorithm, a cut is modeled by associating each voxel that is cut with a specific pre-defined cutting pattern of a voxel, and the cut surface and its correspondence to the simulation voxels are constructed based on pre-defined cutting patterns of voxels. By extending this work, Wu et al. [41] further improve the quality of the surface mesh by employing the dual contouring method [15]. Based on that, Jia et al. [13] have recently proposed a parallel framework that makes use of both an octree and graphical processing units (GPU) to further enhance cutting performance. However, in interactive cutting each voxel is cut progressively and potentially in different manners; the actual cutting pattern can only be confirmed after the completion of cutting. Moreover, to achieve real-time performance, these cutting pattern-based methods need a LUT [33] with all cutting patterns pre-computed.

Another class of voxel-based methods [20,21,27,34] embed a detailed surface mesh into a voxel grid, and a cut is modeled by remeshing the surface mesh, updating the voxel connectivity, and carefully maintaining their correspondence. It is essential but challenging to find a consistent mapping between different representations [11]. Seiler [34] simulates a non-progressive cutting scenario, where the cut is modeled by removing all the material within a volumetric blade at once, and the boundary surface reconstruction is extremely simplified, but this process simulates stamping rather than cutting. Manteaux et al. [20] propose a novel method for simulating interactive detailed cutting in deformable thin sheets. Recently, Mitchell et al. [21] have simulated soft tissue cutting for plastic surgery by embedding a pre-cut surface mesh into a simulation grid, where the incision on the surface mesh is modeled by subtracting a thickened cutting path pre-defined by the user. This method is not applicable to progressive cutting.

3 The *Divided Voxels* algorithm

When incorporating cuts into a deformable object represented by voxels, previous approaches [7,13,41] employ splitting cubes algorithm [33] to model the cuts by associat-

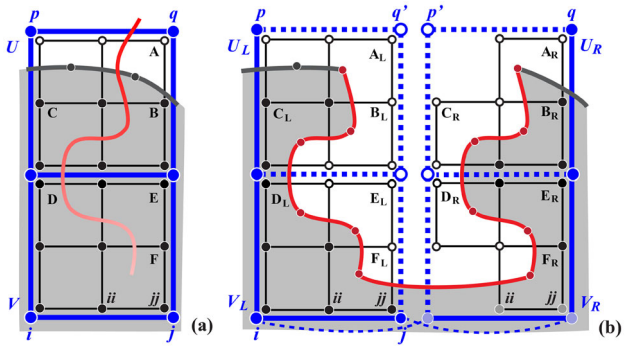


Fig. 1 PBD model construction and cutting updates. **a** Simulation voxels (blue) that are constructed from a coarse level of the initial voxel grid are associated with particles (blue circles) and constraints. Each simulation voxel contains $(2^l)^3$ voxels (black) sampled from the densest level of the grid, and each surface vertex corresponds to exact one voxel. **b** Simulation voxels being cut are divided with the *Divided Voxels* algorithm. The original stretching constraints are rendered with solid blue lines, while the newly added stretching constraints are rendered with dashed lines. When simulation voxels are not cut through, e.g., V, their divided simulation voxels, e.g., V_L and V_R, share the same particles (i and j) and stretching constraints

ing each voxel that has already been cut to a pre-defined cutting pattern in a LUT with all cutting patterns precomputed to realize real-time performance. However, in interactive cutting, each voxel is cut progressively, and the actual cutting pattern is finalized only after the cutting of the voxel is completed, not during the cutting process. This may be appropriate for very fine voxel grids, but it is unacceptable in real-time simulation with relatively coarser grids. To solve this problem, we propose a new voxel-based topological operator, *divide*, which can dynamically update the topology of the voxel being cut, while it is being cut to support progressive cutting.

In this work, a deformable object is represented by a set of edge-sharing voxels sampled from the densest level of a uniform grid, and simulation voxels are built from a coarse level, l, of the same initial grid. As illustrated in Fig. 1a, each simulation voxel (in blue) contains $(2^l)^3$ voxels (in black), where $l = 0$ denotes the finest level of the grid. For each voxel containing the boundary (including the cutting surface) of the deformable object, see Fig. 1b, one surface vertex can be generated based on the dual contouring algorithm and associated with the voxel via its barycentric coordinates. The correspondence between a surface vertex and its containing voxel and the correspondence between a voxel and its containing simulation voxel are retained automatically during cutting.

In the following, a non-manifold voxel grid structure facilitating topology changes is introduced in Sect. 3.1. The *divide* operator that enables progressive cutting in voxels is described in Sect. 3.2. The data structure of the non-manifold voxel grid is presented in Sect. 3.3. The cut surface generation and surface vertices updates are explained in Sect. 4. In Sect. 5, we discuss how to construct simulation voxels from a coarse level of the initial grid, how to deform the voxels and the surface mesh based on the simulation voxels, and how to update simulation voxels during cutting.

3.1 A non-manifold voxel grid

We employ a structured grid of hexahedral elements to simulate deformable objects similar to other methods [8,11,27]. However, due to the presence of disconnected components appearing in the same voxel resulted from a cut, a standard voxel grid is unable to present such topology using regular-shaped voxels, unless an additional surface part is included into the voxel such as in the case of splitting cubes [33]. Hence, various ways of cutting a voxel must be defined case by case in the splitting cubes-based approaches [7,13,41]. We thus adopted a non-manifold voxel grid aiming to represent connectivity of the disconnected components using regular voxels.

Non-manifold grids are often used in embedding simulations [20,21] to simulate objects with changing topology, where a separate surface mesh is embedded in a voxel grid. When a cut is made, the surface mesh is cut first, and each cut voxel is duplicated as many times as it contains disconnected parts as in the virtual node algorithm [22]. Each duplicate has a specific connectivity based on material continuity. Although the use of non-manifold grid improves the topological expressive ability of regular voxels, it is challenging for these methods to find a consistent mapping between two different representations. Moreover, the procedure of cutting the surface mesh and voxels separately and then maintaining their correspondence increases the computational complexity. The novelty of this work is that such structure is generated differently by using the *divide* operator.

As illustrated in Fig. 2a, a uniform Cartesian voxel grid is used to embed the deformable object. We distinguish between voxels that are inside the body and those that include the boundary of the body. Voxel nodes are labeled with two **signs**: *in* (black circle) and *out* (empty circle), indicating whether it is inside or outside of the body. We define *inner voxels* as those voxels all of those nodes are *in* (e.g., voxel C). Similarly, *boundary voxels* are voxels with nodes that are both *in* and *out* (e.g., voxel D). Boundary voxels contain the boundary surface including the new cut surface of the deformable object.

During progressive cutting, a partially cut voxel (e.g., C) is divided into two voxels (C_L, C_R), both of which are connected to the same edge (e) shared by voxel D; see Fig. 2b. This results in *non-manifold connectivity*. In our method, the cutting surface (contour in red) can be constructed directly based on our non-manifold grid using the dual contouring approach [15], retaining surface–voxel correspondence.

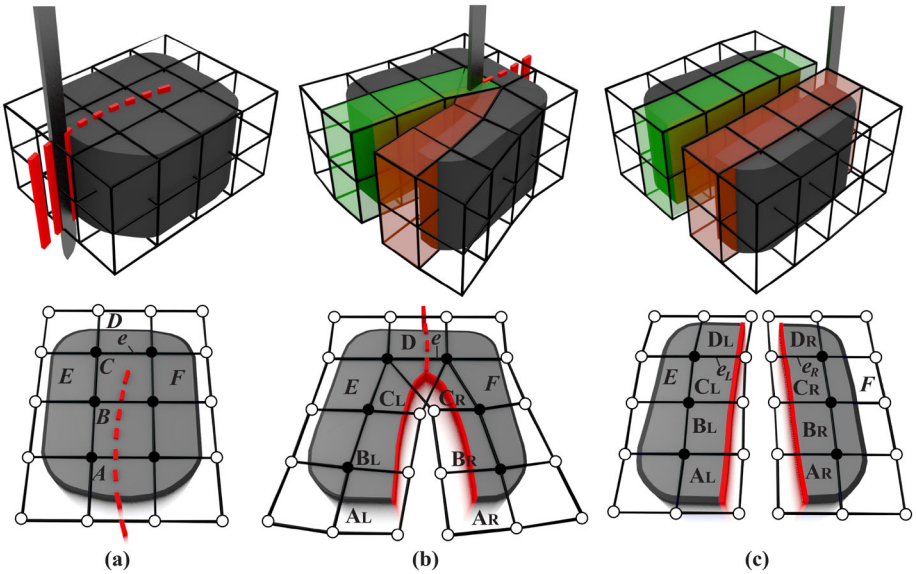


Fig. 2 *Divided Voxels* algorithm for progressive cutting of a deformable object. The top row of figures shows the 3D geometry, whereas the bottom row shows the top view. In **a** the object is shown being cut by a scalpel, with the red dashed line indicating the line of cut. For each voxel that is cut, see (**b**), two divided voxels (green and orange) are dynamically updated to represent the topology and connectivity of the progressing cut until the voxel is completely cut (**c**). The red solid curve in bottom figures indicates the cut surface constructed from the divided voxels

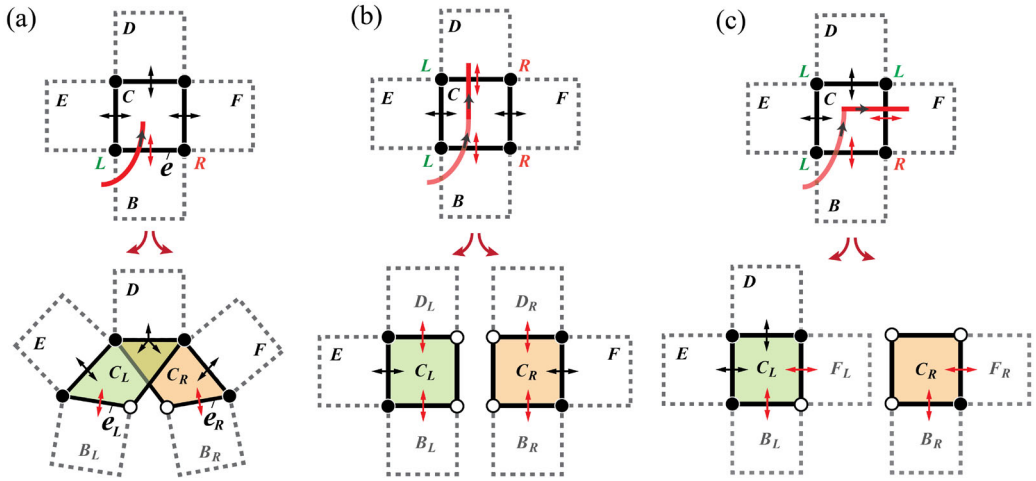


Fig. 3 2D illustration of dividing a voxel, C, in progressive cutting: (**a**) $\rightarrow$ (**b**) and (**a**) $\rightarrow$ (**c**). When a cut starts in a voxel, see (**a**), two topologically distinct ways of cutting through the voxel are depicted in **b** and **c**. In progressive cutting, the *divide* operator dynamically distributes the elements (nodes and edges) of the original voxel being cut into its two divided voxels, C_L and C_R, while updating their *edge-sharing connectivity* (indicated by double-head arrows), i.e., connections to their neighboring voxels (dashed quads), until the cutting of the voxel is completed. These two new divided voxels replace the original voxel to represent the new disconnected parts resulting from the cut. *Note* that no deformation is involved at this stage, the two divided voxels are stretched and separated just for display, and they share the same shape as the original voxel and entirely overlap one another

Edge-sharing connectivity (i.e., voxels connected to the same edges) is recorded explicitly in our data structure for modifying voxel connectivity to reflect topological changes during cutting.

As the cut continues, voxel C is cut through and edge e is split. C_L and C_R are completely disconnected and attached to new edges, e_L and e_R, respectively; see Fig. 2c. The *Divided Voxels algorithm* is detailed in the following section.

3.2 The *divide* operator

In this section, we introduce the voxel-based topological operator—*divide*—and how it dynamically updates the voxel topology to assist progressive cutting.

When the cutting is started in a voxel, we first create two voxels, identical in size to the original voxel, to represent the new disconnected parts resulted from the cut. These two voxels are called "*divided voxels*." As the cut in the voxel continues, to reflect the topological change, see voxel C in Fig. 3a, the *divide* operator dynamically distributes all

the elements (nodes, edges) of the original voxel into the divided voxels, C_L and C_R. At the same time, the *divide* operator updates the *edge-sharing connectivity* (double-head arrows) associated with the edges of the divided voxels until the cutting of the voxel is completed; see Fig. 3b, c for two topologically distinct ways of cutting. Therefore, the *divide* operator has two main functions: (1) voxel element distribution and (2) *edge-sharing connectivity* update.

- Voxel element distribution

Elements of a voxel include nodes and edges. As illustrated in Fig. 3a, the path of a progressive cut is represented by the red contour, whose marching direction is shown by single-head arrows in gray. Nodes of voxel edges that are intersected by the cut are divided into two sides along the cutting path, i.e., *left* (L) and *right* (R), and then distributed to the same locations of the divided voxels, C_L and C_R accordingly. Nodes whose edges are not intersected by the cut are distributed into both divided voxels.

Similarly, the uncut voxel edges are assigned to the divided voxels based on the L/R tags of their nodes. However, for each edge that is cut (e.g., e), its nodes are first duplicated, and each node replica is set to the opposite L/R tag of the original node and tagged "*out*" (empty circle)—indicating it lies outside the object. Two new edges, e_L and e_R, are then created by connecting one node with the replica of the other node and assigned to the new divided voxels C_L and C_R, respectively; see the bottom of Fig. 3a.

As the cut continues, more edges of the voxel are cut, and the same voxel element distribution method is utilized to update the divided voxels until the cutting of the voxel is completed; see Fig. 3b.

- *Edge-sharing connectivity* update

As shown in Fig. 3a, for each voxel (e.g., C) in our grid structure, its neighboring voxels (e.g., B) can be accessed through the *edge-sharing connectivity* (indicated as double-head arrows) associated with each voxel edge. When a voxel is cut, two divided voxels are created to replace the original one. To preserve the existing connectivity while representing the continuity of the new cut, the *edge-sharing connectivity* of the divided voxels needs to be updated during the same time as the voxel element distribution. When the edge distributed to the new divided voxel (e.g., C_L) is uncut, the existing connectivity (black double-head arrows) between the original voxel, C, and the other voxels (e.g., E) that share the same edge is retained by the divided voxel. In the meantime, new connections (red arrows) between the adjacent divided voxels (e.g., C_L and B_L) are constructed and associated with the new edges (e.g., e_L), which are split from the original edges (e.g., e) that are cut.

When the cut progresses, more edges are cut; see Fig. 3c. As new edges are then created and distributed into the divided voxels, new connectivity between these divided voxels (e.g., C_L) and the newly created divided voxels (e.g., F_L) are then constructed to extend the cut.

3.3 Non-manifold voxel data structure

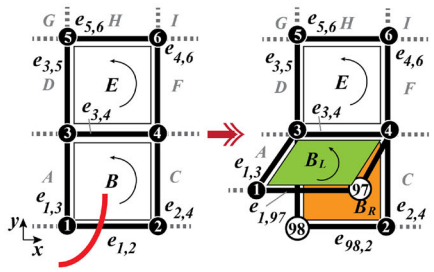
There are three basic elements in our data structure: voxel node, voxel edge, and voxel. Similar to mesh data structures [18], the connectivity between these voxel elements is stored explicitly in our data structure to facilitate topology changes during cutting. Each element contains multiple attributes:

- Voxel node stores x, y, z coordinates, a pointer to the new replica of the node resulting from cutting, an *in/out* sign according to the object, and a L/R tag with respect to the cutting plane;
- Voxel edge stores the indices of the 2 nodes forming it and *edge-sharing connectivity*, where more than 4 voxels can share the same edge resulting in a non-manifold connectivity;
- Voxel stores the indices of all the 8 nodes and 12 edges forming it. Moreover, each voxel also stores a pointer to the voxel from which it is divided (parent) and its *divided voxels*. This information can efficiently support voxel element distribution and *edge-sharing connectivity* updating from the *divide* operation during the progressive cutting of voxels.

For more details, a 2D example is provided to explain the non-manifold voxel grid data structure and its update during cutting in Fig. 4.

4 Cut surface meshing and updates

Dual contouring [15] is a well-known mesh generation approach based on voxels. In this work, dual contouring is utilized for cut surface mesh generation due to its capability of preserving sharp features, such as the sharp corners created by cuts, using fewer triangles compared to the marching cubes method [19]. In marching cubes, surface vertices are located on the voxel edges that intersects the object boundary, and the surface mesh is generated by directly connecting those edge vertices, which usually leads to the loss of sharp features [15]; in dual contouring, surface vertices are distributed inside the voxels, where the sharp features exist, and surface mesh is generated by connecting the surface vertices of the voxels sharing the same edges with sign changes. Since both voxel node sign change and *edge-sharing connectivity*



Node	1	⋯	98
x, y, z	⋯	⋯	⋯
in/out	*in*	⋯	*out*
L/R	*null* → *L*	⋯	*R*
Replica	98	⋯	*null*

Edge	$e_{1,2}$	$e_{3,4}$	⋯	$e_{98,2}$
Nodes	1,2	3,4	⋯	98,2
Voxels	*B*	$B, E \rightarrow B_L, B_R, E$	⋯	B_R
Parent	*null*	*null*	⋯	$e_{1,2}$
DE:L	*null* → $e_{1,97}$	*null*	⋯	*null*
DE:R	*null* → $e_{98,2}$	*null*	⋯	*null*

Voxel	*B*	⋯	B_L	B_R
Nodes	1,2,4,3	⋯	1,97,4,3	98,2,4,3
Edges	$e_{1,2}, e_{2,4}, e_{3,4}, e_{1,3}$	⋯	$e_{1,97}, e_{97,4}, e_{3,4}, e_{1,3}$	$e_{98,2}, e_{2,4}, e_{3,4}, e_{98,3}$
Parent	*null*	⋯	*B*	*B*
DV:L	*null* → B_L	⋯	*null*	*null*
DV:R	*null* → B_R	⋯	*null*	*null*

Fig. 4 Data structure of the non-manifold voxel grid and updates during cutting. Left: 2D illustration of our non-manifold voxel grid being cut. Right: The data structure consists of three basic elements: voxel node, voxel edge, and voxel; the connectivity between these elements is stored explicitly. Each element also has a pointer to its parent and the voxels/edges that are divided from it (DV/DE). The cut elements (e.g., voxel B) are shown in red; the elements whose attributes are updated due to the cut (e.g., edge e_{34}) are shown in yellow; the newly created elements (e.g., B_L) are illustrated in blue

are preserved in the divided voxels during cutting, dual contouring can be employed for cut surface generation in our method.

In this section, we focus on mesh generation from a partial-cut voxel, which is not considered in the original dual contouring algorithm. We also explain how to update the displacement of each surface vertex to reflect a progressive cut and partial cut within each voxel (Sect. 4.1). We then introduce surface vertex displacement during deformation, especially from those voxels that are partially cut (Sect. 4.2).

4.1 Cut surface meshing during progressive cut

When the voxel is completed cut, the original dual contouring algorithm can be directly applied to the divided voxels; the position of the newly created surface vertex (see v in Fig. 5b) within each divided voxel is calculated by minimizing a quadratic function based on the intersections between voxel edges and the cutting surface. For each voxel edge with a sign change, a quad is formed by connecting the surface vertices of the four voxels adjacent to that edge to present the cut surface. For more details of the dual contouring algorithm, we refer the reader to the original work of Ju et al. [15].

In this section, we focus on mesh generation from partially cut divided voxels using dual contouring algorithm, as such voxels are not considered in the original dual contouring work.

4.1.1 Cut surface vertex generation and update

In progressive cutting, voxels are cut gradually by the cutting surface. However, in the dual contouring method, the position of a cut surface vertex, v, within a voxel is calculated based on the intersections between the voxel edges and the cutting surface, namely *edge vertices*; see circles in green in Fig. 5; the position of the surface vertex can only be updated

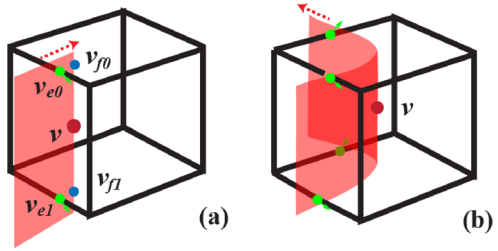


Fig. 5 Surface vertex update during progressive cut. **a** When the voxel is partially cut, the cut surface vertex, v, is positioned in the average position among the face vertices (e.g., v_{f0}, v_{f1}, circles in blue). It is constantly updated to reflect progressive cut within each voxel. **b** Once the voxel is completely cut, the position of v is then determined by minimizing a quadratic function based on the edge vertices and their normals (in green) according to the dual contouring algorithm

when the cutting surface intersects new edges of the voxel. To represent a progressive cut within the voxel, as shown in Fig. 5a, in this work cutting is started in a voxel when at least one edge is cut, and the position of the surface vertex, v, is constantly updated to the average position among the *face vertices*, v_{f0} and v_{f1} (circles in blue), i.e., the most front intersection points between the voxel faces and the cutting surface with respect to the cut marching direction (dashed arrow). Once the voxel is completely cut, see Fig. 5b, the position of the surface vertex v is then updated to the location minimizing a quadratic function based on all the edge vertices and their normals (in green) according to the original dual contouring method.

4.1.2 Mesh generation for partial-cut voxels

As depicted in Fig. 6, when dividing a cut voxel, (e.g., A, into two voxels, A_L and A_R), all the edge vertices of the original voxel, p_0 and p_1, are distributed to these divided voxels along with the edges being distributed. The edge vertices (e.g., q) that are created by cutting are distributed to both divided voxels. For each divided voxel (see A_R in Fig. 6b), a surface

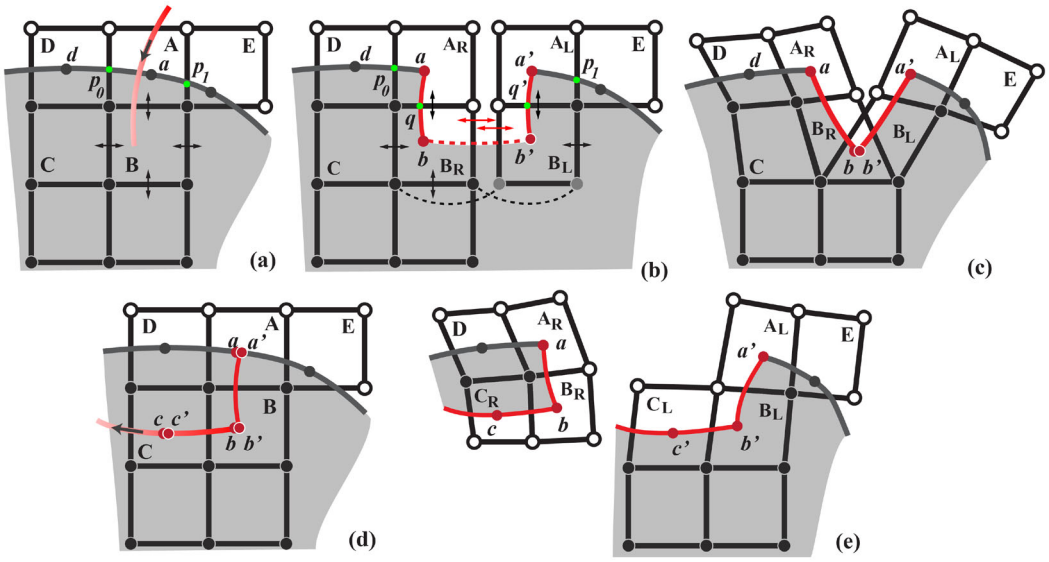


Fig. 6 2D illustration of cut surface mesh generation and update during progressive cut based on divided voxels. **a** The cutting surface and its marching direction are shown with a red curve with an arrow, and voxel A is completely cut, while voxel B is partially cut. **b** For a complete-cut voxel, e.g., A, the cut surface is generated directly based on dual contouring, while for a partial-cut voxel, e.g., B, the *edge-sharing connectivity* (red double-head arrows) is built between the two divided voxels, B_R and B_L, to generate a fully connected surface mesh. *Note* that B_R and B_L share the same edge and entirely overlap one another before the deformation; to clearly demonstrate the connectivity of the new cut surface mesh, they are placed separately in this example. **c** Displacements of surface vertices within partial-cut voxels, e.g., b and b' are synchronized based on both divided voxels to render the singularity. **d**, **e** For a partial-cut voxel, e.g., B, the cut surfaces within its divided voxels are kept updating with more edges being cut until the cutting of the voxel is completed

vertex is created and updated constantly based on the cutting surface as described in the above section. In accordance with the dual contouring algorithm, for each edge of a divided voxel with a sign change, a quad is formed by connecting the surface vertices of the four voxels adjacent to that edge to represent the cut surface. As illustrated in Fig. 6b, two edges of voxel A_R are intersected and exhibit a sign change, and A_R's surface vertex a is connected to the surface vertices, d and b, of the voxels sharing those edges, the newly created cut surface mesh (contour between a and b) is therefore connected to the existing surface mesh (contour between a and d).

For partial-cut voxels, however, since there is only one edge being cut (see voxel B in Fig. 6a) the cut surface mesh generated from its divided voxels is not closed based on the original dual contouring algorithm. To address this issue, we further build *edge-sharing connectivity* between the two divided voxels derived from the same voxel; see red double-head arrows between B_L, B_R in Fig. 6b. The surface vertices, b and b', of the two divided voxels are then connected to one another based on dual contouring, and the cut surface meshes within those two voxels are therefore seamlessly connected.

As the cut continues and intersects with more edges of the partial-cut voxel, e.g., B in Fig. 6d, the voxel elements (e.g., *in/out* of voxel nodes) and *edge-sharing connectivity* of its divided voxels (e.g., B_L and B_R) are kept updating by the *divide* operator until the cutting of the voxel is completed. With more edges are cut, the surface vertex connectivity of the cut surfaces within the divided voxels are updated, while the new connectivity between the surface vertices of these divided voxels and the newly created divided voxels are constructed to extend the cut surface. For instance, surface vertex b of voxel B_R is connected to b' of B_L when voxel B is partially cut, but it is connected to surface vortex c of voxel C_R, when the cut passes through voxel B and extends to voxel C; see Fig. 6e.

4.2 Surface vertex displacement during deformation

In this work, the surface mesh constructed based on the voxels is used to represent the boundary of a deformable object. The surface mesh is deformed by updating the displacement of each surface vertex based on the deformation of its corresponding voxel.

To deform the surface mesh, each surface vertex is bound to one voxel via its barycentric coordinates. For each voxel that is cut, only one surface vertex is created for each divided voxel, and the correspondence between each surface vertex and its underlying voxel is still retained; see the surface vertex a of voxel A is assigned to one of its divided voxels, A_R in Fig. 6b. The displacement of each surface vertex is updated based on the voxel nodes using trilinear interpolation (bilin-

ear in 2D); see a and a' in Fig. 6c. The deformation of each voxel is discussed in Sect. 5.

In partial-cut voxels, the cut surface meshes join to the same points. To display this singularity, the displacements of the two surface vertices within the divided voxels should be synchronized during the deformation. In the current implementation, we first calculate the new position of the surface vertex within each divided voxel using trilinear interpolation. The surface vertices are then positioned to the averaged location among their interpolated locations, see b and b' in Fig. 6c, and their barycentric coordinates are then updated, respectively.

5 Deformation simulation and dynamic model updates

The position-based dynamic (PBD) method [25] has been used to simulate the deformable objects in this paper, though other approaches including the finite element methods could be used. We briefly discuss the extension to the extended finite element method in Sect. 7 to demonstrate that. The PBD model for simulation is constructed based on the voxels of a coarse level of the initial voxel grid to represent the deformable object. To simulate the deformation during cutting, the PBD model needs to be updated simultaneously as the voxels are being cut. Although a previous work [30] also implements a PBD approach that accommodates the topology modifications, whenever a cut is made, the mapping between surface mesh and simulation mesh needs to be carefully reestablished before the deformation starts. In our method, each surface vertex always corresponds to one voxel, and each voxel always corresponds to one simulation voxel. These correspondences are retained automatically during cutting.

In this section, we first discuss how to construct a PBD model based on the voxels sampled from a coarse level of the initial grid and how to deform the voxels and the surface mesh based on the simulation voxels (Sect. 5.1). We then explain the update of the PBD model during the progressive cutting using the same *Divided Voxels* algorithm (Sect. 5.2).

5.1 PBD model construction and deformation

In PBD, the physics model is represented by a set of particles and constraints. Each particle is associated with a set of constraints, and its new position can be obtained by solving all the constraints for each time step.

In this work, a deformable object is represented by a set of edge-sharing voxels sampled from the densest level of a uniform grid, and simulation voxels are built from a coarse level, l, of the initial grid. As illustrated in Fig. 1a, each simulation voxel (in blue) contains $(2^l)^3$ voxels (in black), where $l = 0$ denotes the finest level of the grid. For each voxel containing the boundary of the deformable object, see Fig. 1b, a surface vertex is created and associated with the voxel via its barycentric coordinates, as explained in Sect. 4.1. To construct the PBD model, we assign particles (blue circles) to voxel nodes (black circles) and apply two types of constraints to each particle: stretching constraints [25] and shape matching constraint [26] bound to eight nodes of each simulation voxel. Each simulation voxel is therefore made of eight particles, one shape matching constraint, and twelve stretching constraints, which are shared by other simulation voxels. Simulation voxels are deformed when applying PBD to all the particles and constraints of the PBD model. For more details of the deformation, we refer the reader to the original PBD work. For each simulation voxel, the nodes of the voxels that are contained within the simulation voxel are updated via trilinear interpolation based on the particles associated with that simulation voxel; each surface vertex within each voxel is then updated by interpolating those voxel nodes during the deformation.

5.2 PBD cutting updates

When voxels are being cut, their corresponding simulation voxels are updated simultaneously. A simulation voxel starts to be cut when at least one of its containing voxels are being cut. As shown in Fig. 1b, we first create two new simulation voxels by following a similar manner of dividing a voxel, and the old simulation voxel is then removed from the dynamic system. The stretching constraints (solid blue lines) that are not cut are directly distributed to the new simulation voxels. For each stretching constraint that is split, the two particles connected to the constraint are first duplicated, and two new stretching constraints (dashed blue lines) are created by associating the original particles (solid blue circles) with the duplicated particles (empty blue circles), and then distributed to the new simulation voxels, respectively; see p and q' of voxel U_L in Fig. 1b. In the meantime, a new shape matching constraint is constructed by including all the particles mapped to each new simulation voxel. As the cut continues within a simulation voxel, its containing voxels and their divided voxels are distributed into the corresponding divided simulation voxels of the same L/R; see voxels A_R, B_R, and C_R are assigned to simulation voxel U_R in Fig. 1b.

When the simulation voxel is not cut through, the newly created simulation voxels share the same particles and stretching constraints, e.g., V_L and V_R share the same stretching constraints connecting particle i and j in Fig. 1b. As the cut progresses, more stretching constraints are split, and these simulation voxels are updated dynamically by following the same procedure as described above. When the simulation

voxel is completely cut, the two new simulation voxels are entirely separated.

When dividing a simulation voxel, all the nodes (particles) are duplicated and distributed into the new simulation voxels. In this work, when dividing a simulation voxel, we split the mass of each original node, e.g., p in Fig. 1a, equally to the node and its duplicate, p and p' in Fig. 1b, as they are distributed to the new divided voxels to achieve to mass conservation.

6 Results

In this section, we conduct a series of experiments to analyze the performance of our approach for interactive cutting of deformable objects. We further compare our work with other real-time cutting methods [30,41]. Lastly, we conduct another experiment to analyze the mesh quality after cutting and compare our work with other methods based on internal angle metrics. The *Divided Voxels* algorithm was implemented in C++. All the experiments were run on a standard desktop with Intel i7-6850K, 3.6 GHz CPU, and 16.0 GB RAM.

6.1 Experiment results and performance analysis

To analyze the performance of our cutting algorithm, we present two types of interactive cutting experiments: (1) cutting deformable objects represented by increasing voxel resolution and (2) cutting the same deformable object with increasing length of cut.

Experiment 1: Interactive cutting with increasing voxel resolution. Table 1 shows the data of cutting simulation time for three example models, including a Stanford bunny, armadillo, and a liver model (see Fig. 7), represented by different numbers of voxels (second column). For each model, two levels of uniform voxel grid are used: the finest level of voxels for cutting and a coarse level for deformation simulation. For all the following examples, the simulation voxels are sampled from the same coarse level, i.e., $16 \times 16 \times 16$, of the initial grid. The third column shows the number of simulation voxels for each example, which is also the number of shape matching constraints. The number of simulation voxel edges, which is also the number of stretching constraints, is listed in the fourth column. The fifth column shows the number of triangles of the surface mesh that are reconstructed from the voxels.

Fig. 7 Example problems **a** Stanford bunny, **b** armadillo, and **c** liver model

To investigate the interactive cutting performance by increasing the number of voxels that are used to represent a model, the cutting length remains the same for each example model. Since cutting is always performed progressively, in a sense that in every frame the scalpel is moved only a small distance through the model, we measure computation time by averaging over simulation frames where cutting occurs. The average time (in milliseconds, ms) spent on cutting, deformation, and their sums (total) are listed in the last three columns of Table 1. The time for *cutting* indicates the computation time spent on both cutting of voxels and reconstructing the surface mesh. The *deformation* includes both performing PBD on simulation voxels and deforming the surface mesh.

To maintain 30 frames per second (FPS) real-time rendering speed, the whole computation needs to be done in 33.3 ms. The data indicate that the proposed cutting method is, therefore, suitable for real-time applications even when a high voxel resolution ($256 \times 128 \times 128$) is utilized. This is clearly shown in Fig. 8, where the times for cutting the armadillo model (Fig. 7b) is plotted with an increase in the number of voxels, with the cutting length remaining the same. It can be observed that by increasing the number of voxels for an object, all the curves of cutting, deformation, and total time of these two show an overall linear growth. Compared to the cutting time, the deformation time only increases slightly as the initial resolution of the simulation voxels for all examples is the same, and the increase in the number of voxel constraints that participate in the simulation is relatively small.

Experiment 2: Interactive cutting with progressive increase in cut length. As shown in Fig. 9, we conduct another experiment by cutting the same model with a fixed voxel resolution ($64 \times 64 \times 64$), but increasing the length of the cutting path, which leads to an increase in the number of cut voxels. Both the bunny and the armadillo are chosen for this experiment; their initial voxel resolutions are $64 \times 64 \times 64$ and $128 \times 128 \times 128$, respectively.

We plot the cutting time with an increase in the number of cutting voxels in Fig. 10, which shows that although different resolutions are adopted, the cutting times for both models show an overall linear growth with approximately the same slope. This demonstrates that the cutting time of the *Divided Voxels* algorithm is proportional to the number of voxels that are cut.

To demonstrate our algorithm's ability to handling more complex cutting scenarios, we include another two results in Fig. 11, where multiple cuts and a helix-shaped cut are shown.

Table 1 Simulation time for different models with different numbers of voxels (#voxels)*

Model	#Voxels	#Sim Voxels	#Sim vox-Edges/nodes	#Triangles (k)	Time (ms) Cutting	Deformation	Total
Bunny	16 × 16 × 16	1928	15.0k/7.0k	2	0.3	6.8	7.1
	32 × 32 × 16			5.4	0.5	7	7.5
	64 × 64 × 32			22.2	1.9	7.2	9.1
	128 × 64 × 64			56.8	3	8.2	11.2
Armadillo	128 × 128 × 64	1823	14.6k/6.8k	71.4	6.1	9.1	15.2
	128 × 128 × 128			111.4	11.2	11.1	22.3
	256 × 128 × 128			184.1	17.1	14.3	31.4
Liver	64 × 64 × 64	2037	16.3k/7.7k	35.1	1	7.1	8.1
	32 × 32 × 32			8.6	2	7.5	9.5

*Simulation voxels for all examples are sampled from the same level, i.e., 16 × 16 × 16, of the initial voxel grid

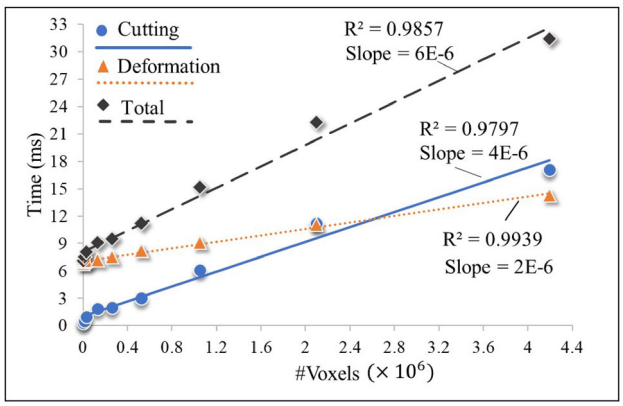


Fig. 8 Cutting, deformation, and total times for cutting the armadillo model while increasing the number of voxels

6.2 Comparison with existing techniques

We also compare our approach with other real-time cutting methods: a voxel-based composite FEM approach proposed by Wu et al. [41] and Pan et al.'s tetrahedra-based remeshing method that also employs PBD [30], by performing cutting on the same examples represented by the approximately same number of elements used in their papers.

Comparisons of simulation performance with these two methods are shown in Tables 2 and 3, respectively. It can be observed that our cutting method demonstrates a higher overall performance compared to the other two. Specifically, compared with Wu et al., where example models are represented by voxels with higher resolutions varying from 100 to 5000 k, the speedup of our method is about 3–5x, while the speedup of our method is about 2x compared to Pan et al., where examples are modeled with low-resolution tetrahedra.

We further compare the cutting time (not including the deformation time) of these three works. In comparison with Wu et al.'s work, the cutting time we listed in Table 2 includes both cutting of voxels and reconstructing the surface mesh;

Fig. 9 Cutting time with the increase in the number of cut voxels

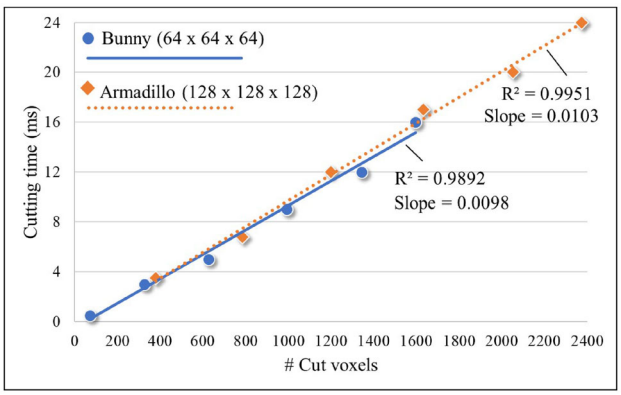


Fig. 10 Increasing the number of cut voxels with the same voxel resolution

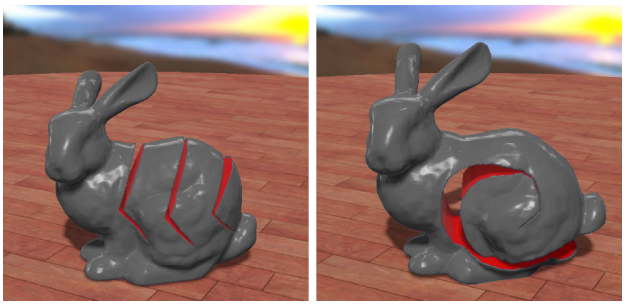


Fig. 11 Examples of multiple cuts and helix-shaped cut

the speedup of our *Divided Voxels* algorithm is about 3–4x for all examples. For Pan et al.'s tetrahedra-based cutting approach, the cutting time shown in Table 3 is composed of both tetrahedra decomposition and surface mesh subdivision; our method is significantly faster than Pan et al.'s work about 40x for all examples.

For deformation, the PBD-based methods that are used by both this and Pan et al., are faster than composite FEM employed by Wu et al., although the latter is more physically accurate. Most notably, in addition, to be the fastest, our cutting method shows better scaling with respect to the increasing number of voxels.

The visual comparison of the cutting results between our work and Wu et al.'s work is shown in Fig. 12, where the same model that is sampled with the same numbers of voxels ($\sim 260k$) is cut. It can be observed that the cutting results and deformation of both works look similar, while there are some triangles missing along the cut in Wu et al.' result.

6.3 Cut mesh quality comparison

Ill-shaped mesh elements can cause visual artifacts and compromise simulation accuracies [42]. Internal angles provide a commonly used metric to measure triangular mesh quality [32] with equilateral triangles generally preferred [14]. To assess the mesh quality of our algorithm and compare with other cutting methods such as remeshing-based (i.e., element refinement) and Wu et al.'s voxel-based approaches, we calculate the minimum and maximum internal angles for each triangle using the same cutting example; see Fig. 13, where the same input mesh (130k triangles) and cutting plane are used for all the methods. The result of the remeshing-based cutting is obtained using Blender's Boolean difference operation [1], where the triangles being cut are further refined to small triangles. The voxel resolution used to model the cut in Wu et al.'s method is similar to ours ($\sim 600k$). We then calculate [9] the minimum angle $0 < \theta_{min} \leq 60$ and maximum angle ($60 \leq \theta_{max} < 180$) [32] to determine the number of triangles with extreme angels, such that $\theta_{min} \leq \theta_{threshold}$ or $\theta_{max} \geq \theta_{threshold}$, after cutting for all the three methods. As Fig. 13 shows, our algorithm produces higher-quality mesh than both remeshing-based and Wu et al.'s methods by creating fewer ill-shaped triangles with extreme internal angles after cutting.

Table 2 Cutting simulation performance comparison with Wu et al. [41]*

Model	#Voxels (k)	Total time (cutting time) (ms)		
		Wu et al.	*Divided Voxels*	Ratio
Armadillo	~600	43.62 (15.41)	11.29 (3.3)	3.86 (4.67)
	~5000	156.92 (50.09)	31.52 (16.65)	4.99 (3.0)
Bunny	~100	26.87 (5.93)	8.54 (1.34)	3.15 (4.43)
	~800	85.75 (20.13)	14.96 (5.92)	5.73 (3.4)

* Wu et al. used the finest level of voxels for cutting, but a coarser level for simulation. The second column shows the number of voxels used for cutting. The maximum number of simulation voxels used by Wu et al. is around 1540, but we used about 2000 simulation voxels (with approximately 2000 shape matching constraints and 15k stretching constraints) for all examples in this table. The time listed in parentheses is the cutting time. Wu et al.'s statistics were obtained under the desktop with Intel Xeon-X5560, 2.80 GHz CPU, 8 GB RAM

Table 3 Cutting simulation performance comparison with Pan et al. [30]*

Model	#Tets/Voxels(k)	Total time (cutting time) (ms)		
		Pan et al.	Divided Voxels	Ratio
Liver	~4	19.3 (15.2)	6.8 (0.35)	2.84 (43.43)
Spleen	~2	8.1 (5.2)	4.14 (0.14)	1.96 (37.14)
Bunny	~2	7.7 (5.1)	4.13 (0.13)	1.86 (39.23)

* To compare with Pan et al.'s work, where the same number of tetrahedra was utilized for both cutting and deformation, we used a similar number of voxels for both also. The time listed in *parentheses* is the cutting time. Pan et al.'s statistics were obtained using the desktop with Intel Xeon, 2.53 GHz, 12 GB RAM

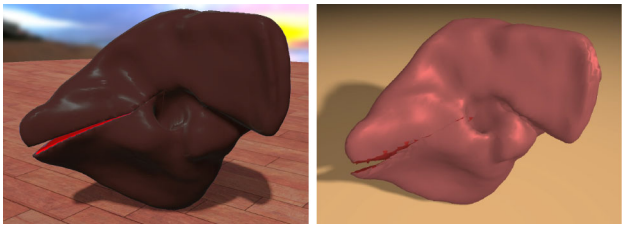


Fig. 12 Visual comparison between this (left) and Wu et al.'s work (right)

7 Conclusion and future work

In this paper, we introduce an efficient cutting algorithm—*Divided Voxels*—for interactive progressive cutting in deformable objects. We propose a novel voxel-based topological operator, *divide*, which can model a cut in a voxel as it is being cut. This operator divides a voxel that is cut into two divided voxels, which can effectively represent the topological change and connectivity of the cut. The cut surface can therefore be generated directly from the divided voxels on the fly, and the correspondence between the cut surface and the simulation voxels is maintained without the need for pre-computation. Using several test cases, we show that our cutting algorithm can perform at interactive rates also scales linearly with an increase in the number of voxels that represent the deformable object. Moreover, our experiment also shows that the proposed cutting algorithm produces higher-quality mesh with fewer ill-shaped elements after cutting.

There are several directions to further improve or extend the *Divided Voxels* algorithm:

Cutting existing cuts. The major focus of this paper is to propose an efficient voxel-based cutting algorithm that can model partial cut and progressive cut within voxels based on a novel topological operator, *divide*. More complex cutting scenarios such as cutting an existing cut are not discussed in this work, but can be handled by our algorithm. Cutting an existing cut such as cut intersection requires to further dividing the voxels that have already been divided. Specifically, when cutting a voxel that is already cut, the number of divided voxels created from the original voxel depends on the number of edges being cut. For simplicity, let us take a 2D voxel for example. If the voxel is cut by two intersecting cutting paths and all its edges are cut, then four divided voxels are created from the original voxel, and the same *divide* operator can be used to distribute the cut voxel elements into the new divided voxels to model the cut surfaces. This belongs to the scope of our future work.

Complex cutting surface. Similar to those voxel-based mesh generation methods (marching cubes, dual contouring) that rely on sign change of voxel nodes to generate surfaces, when the size of the mesh to be constructed is smaller than the width of that voxel, in our case, the cutting path intersects with a voxel edge more than once, or the cutting path does not intersect with the voxel edge at all, this would result in no sign change of the voxel nodes, and the surface mesh may not be constructed properly from the voxels. To represent such complex cutting surface, in the future implementation such

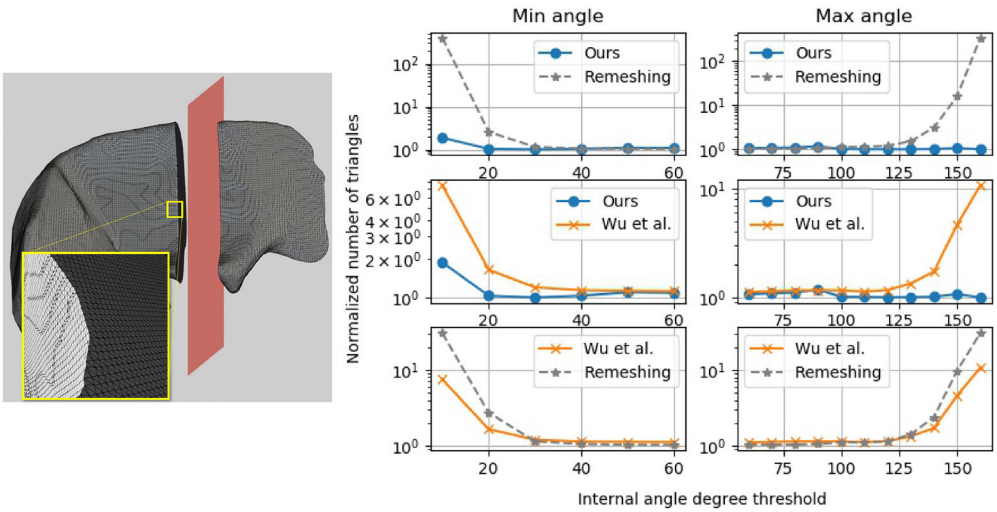


Fig. 13 Left: example model and cutting plane; the detailed view (yellow square) depicts that the triangles of the cut mesh produced by our work are well shaped. Right: comparing mesh quality between our work, remeshing-based cutting, and Wu et al.'s work by measuring θ_{min} and θ_{max} of triangles after cutting, results show that our algorithm produces less triangles with extreme internal angles. Since Wu et al.'s output mesh is not exactly same as ours, we normalize the number of triangles with extreme internal angels by dividing the number of triangles after cutting by the number of triangles before cutting, where each triangle satisfies one of the following conditions: $\theta_{min} \leq \theta_{threshold}$ or $\theta_{max} \geq \theta_{threshold}$, for all three works

voxels can be further refined using a regular 1:8 subdivision until each voxel edge is intersected at most once [15]; we then apply the *divide* operator to the refined voxels to model the cut.

Memory consumption. To effectively represent the topology change in the voxels being cut, we employ a non-manifold voxel grid, where voxel connectivity is stored explicitly in our data structure. Such a grid structure, on the other hand, has a much higher memory footprint than a regular Cartesian grid. To address the above problems more effectively, we plan to exploit adaptive voxels in our future work, where a coarse uniform grid can be used to represent the deformable object in the beginning; we then refine the voxels along the boundary of the object or the places where cuts occur.

Physics-based deformation. Although PBD is implemented in this work for simplicity and speed, techniques such as the extended finite element method (XFEM) may be directly applied to the voxels if they are treated as voxel finite elements and additional degrees of freedom are introduced corresponding to the cut. On the other hand, XFEM or similar algorithms employing level set methods may be computationally more demanding due to the need to perform numerical integration of the weak form.

Acknowledgements Research reported in this article was supported by NIBIB under Award Number R01EB005807, R01EB010037, R01EB009362, R01EB014305, R01EB025241; NHLBI under Award Number R01HL119248; NCI under Award Number R01CA197491. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

References

1. Blender: Open-source 3D computer graphics software toolset. www.blender.org
2. Berndt, I., Torchelsen, R., Maciel, A.: Applications efficient surgical cutting with position-based dynamics. IEEE Comput. Graphi. Appl. **38**, 24–31 (2017)
3. Bielser, D., Glardon, P., Teschner, M., Gross, M.: A state machine for real-time cutting of tetrahedral meshes. Graph. Models **66**, 398–417 (2004). https://doi.org/10.1016/j.gmod.2004.05.009
4. Bielser, D., Gross, M.H.: Interactive simulation of surgical cuts. In: Proceedings-Pacific Conference on Computer Graphics and Applications, vol. 2000-Jan (2000). https://doi.org/10.1109/PCCGA.2000.883933
5. Bielser, D., Maiwald, V.A., Gross, M.H.: Interactive cuts through 3-dimensional soft tissue. Comput. Graph. Forum **18**(3), 31–38 (1999). https://doi.org/10.1111/1467-8659.00325
6. Cotin, S., Delingette, H., Ayache, N.: A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation (2000). https://doi.org/10.1007/PL00007215
7. Dick, C., Georgii, J., Westermann, R.: A hexahedral multigrid approach for simulating cuts in deformable objects. IEEE Trans. Vis. Comput. Graph. **17**(11), 1663–1675 (2011). https://doi.org/10.1109/TVCG.2010.268
8. Frisken-Gibson, S.F.: Using linked volumes to model object collisions, deformation, cutting, carving, and joining. IEEE Trans. Vis. Comput. Graph. **5**(4), 333–348 (1999). https://doi.org/10.1109/2945.817350
9. Jacobson, A., Panozzo, D.: Others: Libigl: A Simple C++ Geometry Processing Library (2018). https://libigl.github.io/
10. Jeřábková, L., Kuhlen, T.: Stable cutting of deformable objects in virtual environments using XFEM. IEEE Comput. Graph. Appl. **29**(2), 61–71 (2009). https://doi.org/10.1109/MCG.2009.32
11. Jerabkova, L., Bousquet, G., Barbier, S., Faure, F., Allard, J.: Volumetric modeling and interactive cutting of deformable bodies. Progress Biophys. Mol. Biol. **103**(2–3), 217–224 (2010). https://doi.org/10.1016/j.pbiomolbio.2010.09.012
12. 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). https://doi.org/10.1007/s11548-014-1147-0
13. Jia, S., Zhang, W., Yu, X., Pan, Z.: CPU–GPU parallel framework for real-time interactive cutting of adaptive octree-based deformable objects. Comput. Graph. Forum **37**(1), 45–59 (2017). https://doi.org/10.1111/cgf.13162
14. Jong, B.S., Chiang, C.H., Lee, P.F., Lin, T.W.: High quality surface remeshing with equilateral triangle grid. Vis. Comput. (2010). https://doi.org/10.1007/s00371-009-0392-7
15. Ju, T., Losasso, F., Schaefer, S., Warren, J.: Dual contouring of hermite data. ACM Trans. Graph. **21**(3), 339–346 (2002). https://doi.org/10.1145/566654.566586
16. Kaufmann, P., Martin, S., Botsch, M., Grinspun, E., Gross, M.: Enrichment textures for detailed cutting of shells. ACM Trans. Graph. (2009). https://doi.org/10.1145/1531326.1531356
17. Koschier, D., Bender, J., Thuerey, N.: Robust eXtended finite elements for complex cutting of deformables. ACM Trans. Graph. **36**(4), 1–13 (2017). https://doi.org/10.1145/3072959.3073666
18. Kremer, M., Bommes, D., Kobbelt, L.: OpenVolumeMesh: a versatile index-based data structure for 3D polytopal complexes. In: Jiao, X., Weill, J.C. (eds.) Proceedings of the 21st International Meshing Roundtable, pp. 531–548. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-33573-0_31
19. Lorensen, W.E., Cline, H.E.: Marching cubes: a high resolution 3D surface construction algorithm. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques—SIGGRAPH '87, pp. 163–169 (1987). https://doi.org/10.1145/37401.37422
20. Manteaux, P.L., Sun, W.L., Faure, F., Cani, M.P., O'Brien, J.F.: Interactive detailed cutting of thin sheets. In: Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games, MIG '15, pp. 125–132. ACM, New York (2015). https://doi.org/10.1145/2822013.2822018
21. Mitchell, N., Cutting, C., Sifakis, E.: GRIDiron: an interactive authoring and cognitive training foundation for reconstructive plastic surgery procedures. ACM Trans. Graph. **34**(4), 43:1–43:12 (2015). https://doi.org/10.1145/2766918
22. Molino, N., Bao, Z., Fedkiw, R.: A virtual node algorithm for changing mesh topology during simulation. ACM Trans. Graph. **23**(3), 385 (2004). https://doi.org/10.1145/1015706.1015734
23. Mor, A.B., Kanade, T.: Modifying soft tissue models: progressive cutting with minimal new element creation. In: Medical Image Computing and Computer-Assisted Intervention—Miccai 2000 **1935**, 598–607 (2000). https://doi.org/10.1007/978-3-540-40899-4_61
24. Mousavi, S.E., Grinspun, E., Sukumar, N.: Harmonic enrichment functions: a unified treatment of multiple, intersecting and branched cracks in the extended finite element method. Int. J. Numer. Methods Eng. (2011). https://doi.org/10.1002/nme.3020

25. Muller, M., Heidelberger, B., Hennix, M., Ratcliff, J.: Position based dynamics. J. Vis. Commun. Image Represent. **18**(2), 109–118 (2007). https://doi.org/10.1016/j.jvcir.2007.01.005
26. Müller, M., Heidelberger, B., Teschner, M., Gross, M.: Meshless deformations based on shape matching. ACM Trans. Graph. **24**(3), 471 (2005). https://doi.org/10.1145/1073204.1073216
27. Muller, M., Teschner, M., Gross, M.: Physically-based simulation of objects represented by surface meshes. In: Proceedings Computer Graphics International **2004**, 0-7 (2004). https://doi.org/10.1109/CGI.2004.1309189
28. Nienhuys, H.: Combining finite element deformation with cutting for surgery simulations. EuroGraphics short presentations (2000)
29. Nienhuys, H.W., van der Stappe, A.F.: A surgery simulation supporting cuts and finite element deformation. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **2208**, 145–152 (2001). https://doi.org/10.1007/3-540-45468-3_18
30. Pan, J., Bai, J., Zhao, X., Hao, A., Qin, H.: Real-time haptic manipulation and cutting of hybrid soft tissue models by extended position-based dynamics. Computer Animation And Virtual Worlds **19**(August), 271–281 (2015). https://doi.org/10.1002/cav
31. 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). https://doi.org/10.1007/s00371-015-1123-x
32. Pébay, P.P., Baker, T.J.: Analysis of triangle quality measures. Mathematics of Computation (2003). https://doi.org/10.1090/s0025-5718-03-01485-6
33. 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). https://doi.org/10.1007/s00371-008-0216-1
34. 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). https://doi.org/10.1007/s00371-011-0561-3
35. Shewchuk, J.R.: What is a good linear finite element? Interpolation, conditioning, anisotropy, and quality measures. Tech. rep. In: Proceedings of the 11th International Meshing Roundtable (2002)
36. Sifakis, E., Der, K., Fedkiw, R.: Arbitrary cutting of deformable tetrahedralized objects. In: ACM SIGGRAPH/Eurographics Symposium on Computer Animation pp. 73–80 (2007). https://doi.org/10.2312/SCA/SCA07/073-080
37. Steinemann, D., Harders, M., Gross, M., Szekely, G.: Hybrid cutting of deformable solids. In: Proceedings IEEE Virtual Reality, vol. 2006, p. 5 (2006). https://doi.org/10.1109/VR.2006.74
38. Wang, M., Ma, Y.: A review of virtual cutting methods and technology in deformable objects. Int J. Med. Robot. Comput. Assist. Surg. e1923 (2018). https://doi.org/10.1002/rcs.1923
39. Wang, Y., Jiang, C., Schroeder, C., Teran, J.: An adaptive virtual node algorithm with robust mesh cutting. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation (2014)
40. Wicke, M., Botsch, M., Gross, M.: A finite element method on convex polyhedra. Comput. Graph. Forum **26**(3), 355–364 (2007). https://doi.org/10.1111/j.1467-8659.2007.01058.x
41. Wu, J., Dick, C., Westermann, R.: Interactive high-resolution boundary surfaces for deformable bodies with changing topology. In: VRIPHYS 2011—8th Workshop on Virtual Reality Interactions and Physical Simulations, pp. 29–38 (2011). https://doi.org/10.2312/PE/vriphys/vriphys11/029-038
42. Wu, J., Westermann, R., Dick, C.: A survey of physically based simulation of cuts in deformable bodies. Comput. Graph. Forum **34**(6), 161–187 (2015). https://doi.org/10.1111/cgf.12528

Di Qi is a Research Scientist with Center for Modeling, Simulation and Imaging in Medicine at Rensselaer Polytechnic Institute. She received her Ph.D. in Mechanical Engineering from the Hong Kong University of Science and Technology in 2015. Her research interests include computer graphics, real-time surgical simulation, virtual reality, and machine learning.

Nicholas Milef is currently a Ph.D. student at Texas A&M University in the Department of Computer Science & Engineering. Previously, he was a software engineer at the Center for Modeling, Simulation, and Imaging at Rensselaer Polytechnic Institute. His research interests include virtual reality, computer graphics, and deep learning.

Suvranu De is a full professor and Head of the Department of Mechanical, Aerospace and Nuclear Engineering and Director of the Center for Modeling, Simulation and Imaging in Medicine at Rensselaer Polytechnic Institute. He received his Sc.D. in Mechanical Engineering from MIT in 2001. His research interests include computational mechanics and medical and biological engineering.