Splitting Cubes: a Fast and Robust Technique for Representing Cuts with Mesh-free Methods

Anonymous

ABSTRACT

This paper presents a novel technique to represent cuts on deformable models implemented with mesh-free methods. Mesh-free methods have become a popular choice for modeling deformable objects. Unfortunately, among their many good properties, they do not explicitly represent the surface of the object and it seems quite a difficult task to enable real-time cutting on these models. Our method uses a regular decomposition of the space in cubic cells and encodes the surface using only the intersections with the edges of the cubes and handling implicitly all the changes on topology. Furthermore a new way to update the physical model is introduced which is easily implemented by the graphics hardware.

Keywords: Deformable Objects modeling, Cut simulation, Fracture simulation.

Index Terms: K.6.1 [Management of Computing and Information Systems]: Project and People Management—Life Cycle; K.7.m [The Computing Profession]: Miscellaneous—Ethics

1 INTRODUCTION

Real Time animation of deformable objects is a need in many areas of computer graphics animation, for example to generate realistic behavior of clothes, fluids,bodies on games or in other applicative areas such as surgery simulation. Especially in the latter case, the model should allow the user to interact with the object, possibly displaying topological changes during the simulation, e.g. if the object is cut with a scalpel or tore away.

If the deformable object is modeled with a mesh, either to represent the surface or the volume, several methods exists to handle the mesh so to implement contact, cuts and lacerations. In these cases, the visual representation of the surface of the object is provided by discretization used for the physical simulation. Less has been said about *mesh-free* methods, where the object is modeled as a set of samples, called *phyxels* that interact to implement the physical behavior of the object and whose properties (including position and velocity) implicitly describe its surface.

This paper presents an efficient and robust solution to represent cutting and tearing on mesh-free methods. The major contribution is twofold:

- the *Splitting Cubes* algorithm to provide a dynamic triangulation of an implicit surface under cuts and lacerations;
- a general solution to handle phyxel-phyxel interaction to correct the physical behavior of the object when cutting or tearing occur.

Although the ideas presented in this paper apply to the whole class of mesh-free methods, we detailed and implemented them for the model proposed in [13], for which some work on this problem already exists [18, 4].

The remaining of the papers proceeds as follows:in Section 2 we briefly review the approaches proposed so far related to the problem of cutting and tearing; Section 3 gives some formal definitions of the type of models to which our solution can be applied and explains the basic ideas of splitting cubesand transparency masks, which are detailed in 6.3 and Section 7.2, respectively. Results and future works are finally reported in Section 9.

2 PREVIOUS WORK

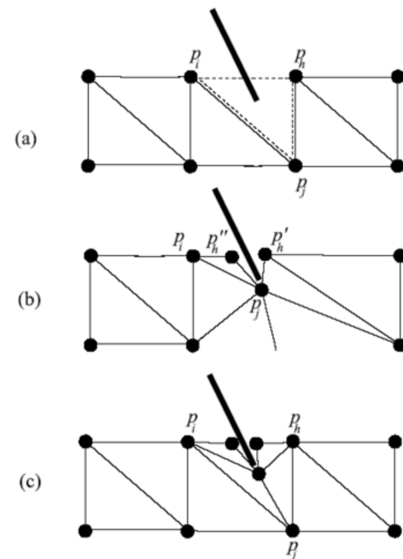

Figure 1: Different approaches to perform cuts on tetrahedral meshes: a)removing intersected primitives. b)Adapting the shape of the mesh by moving vertices c)Adapting the shape of the mesh by local re-meshing.

Most of the methods for real time interaction with deformable objects in literature use a *mesh-based* representation of the deformable body. Since the refresh rate of the physical system is linearly related to the number of primitives of the mesh and the stability of dynamic solvers is strongly influenced by the quality of the elements in the mesh, cutting/fracturing algorithms tries to avoid the creation of unnecessary or bad shaped primitives. Instead, many methods are focused on how to produce an accurate representation of the cut/laceration minimizing the number of primitive created and creating new primitives with good aspect ratios. Figure 1 illustrates three common types of approach:

1. Removing primitives;
2. Adapting the shape of the mesh to the cut by moving vertices;
3. Adapting the shape of the mesh to the cut by re-meshing intersected primitives.

The first method has been proposed in [2] for operating a cut with a scalpel on a tetrahedral complexes and it is quite straightforward: the tetrahedra intersected by the scalpel are removed from the mesh.

This this method avoids the creation of new primitives, but the visual feedback as well as the loss of volume are serious drawbacks. In the second method mesh is locally modified in order to have the faces that separate adjacent primitives in correspondence of the separation path. Finally nodes on the separation path are duplicated, creating the discontinuity in the mesh. In [11] and [16, 15] this method is implemented using a Finite Element model based on tetrahedral cells. This method, like the previous one, avoids the creation on new elements. The third approach groups a wide family of methods essentially based on the concept of local re-meshing. For each combinations of splitting of six edges of a tetrahedron a new set of tetrahedra in inserted to represent the cut. Since the intersection of edges can occur very close to the original mesh vertices, it is possible to create degenerate tetrahedra. This solution, first proposed in [1] for tetrahedral cells, offers the most pleasant visual results, even if the strategy used to find the new set of tetrahedra causes a high fragmentation of the mesh.
A wide kind of variations of this methods are proposed in literature: in [3] this method is extended using extra nodes on tetrahedrons faces in order to increase the degrees of freedom during the cut. In [10] propose to perform real-time simplification by edge collapse in order to erase degenerate tetrahedra. In [9] a multiresolution framework for tetrahedral meshes maintain constant the level of detail of the mesh in case of cuts or lacerations.
Another family of methods to simulate physically based deformable objects are *mesh-less* methods. Often present in simulation of fluids and viscous materials [6] [5] *mesh-less* methods models the interaction between a set of elementary particles,also called *phyxels*. In *mesh based* approaches, such as the Finite Element Method (FEM), the volume is divided into elements of finite size. In contrast, in *mesh free* methods the volume is sampled at a finite number of point locations without connectivity information and without the need of generating a volumetric mesh.

3 OVERVIEW

Our framework define a general solution for representing cuts on mesh free methods. A mesh free method represents the volume of the object with *phyxels*(see Figure 2.(a), which are particles with a set of physical properties to represent a portion of the volume during the animation: mass, position, velocity etc. Each phyxel *interacts* with the phyxels in its neighborhood, typically found as the phyxels closer than a given radius of influence. The interaction consists of reciprocally influence their physical properties (a straightforward example is the gravitational force among particles) and can be more or less complex depending on the specific model. Let x_0 and P_0 be the position of the point x and the position of the phyxels at rest shape, respectively. The position of the same point at time t, x_t is determined by the position of the phyxels P_t by some function $F(x_0,P_t)$. More specifically, the phyxels determining x_t are those including x_0 in their radius of influence, referred in the following as the *kernel* of x. The function F is used when the surface of the model is represented explicitly, for example with a triangle mesh or a surfelization, to compute the positions of the points of the surface in the current configuration. In these terms a cut is a function $Cut(cs,s,F) = s',F'$ where cs is the surface of cutting over which F becomes discontinuous (see Figure 2.(b)).

Representation of the surface

In the solution proposed in [18, 19] the surface is dynamically sampled with surfels represented with oriented elliptical splats. In order to show sharp features, the surfels overlapping a crease can be clipped against a plane lying on the other side of the crease [19]. This is crucial to represent cut because a cut always generates sharp features. In their model a crack is codified by sequence of phyxels (called *crack nodes* in the paper) which represent the propagation front of the crack. For cracks starting from the surface (like when a cut is being made), the first and last node of the sequence lie on the surface while for cracks generating inside the volume the front in circular. Every time a new crack node is added to the sequence, i.e. every time the front propagates, new surfels are added to represent the two new pieces of surface. This technique avoids the problems related to remeshing of the mesh based methods, i.e. fragmentation and degeneracies. On the other hand the crack fronts can split and merge and will ultimately reduce to a single point, and all these events need to be handled explicitly to maintain the topology of the object consistent. In [4], the surface is represented by a triangle mesh. When a cutting tool penetrates the object, the surface swept by the tool, called *cut surface* is triangulated and used to update the current object's surface with the new pieces of sheets. These new sheets are given by triangulating the portion of the splitting surface inside the volume, triangle by triangle. As for the previous solution, they have to handle explicitly the branching and merging of crack fronts. Compared to the point sampled method described above, the use of a triangle mesh may give some advantages in term of rendering speed, but, like similar approaches, if multiple cuts are executed in the same region, triangle fragmentation and degeneracies will occur, causing performance degradation and possibly instabilities of intersection tests, upon which depends the robustness of the approach.

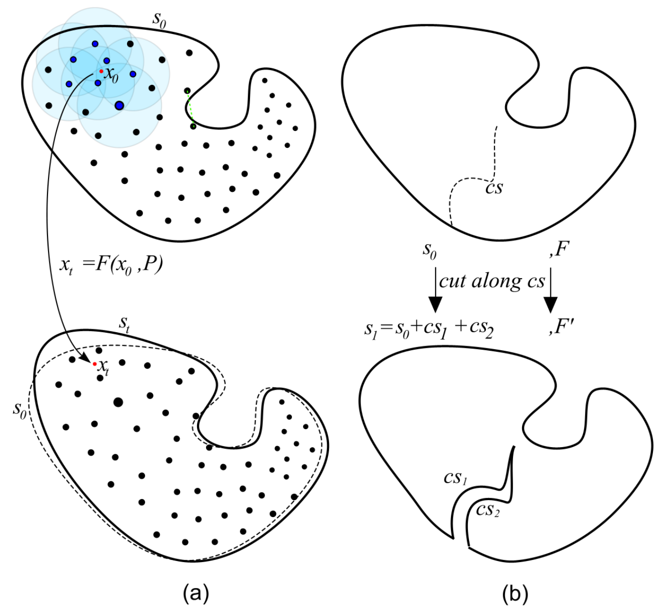

Figure 2: (a) Definition of an abstract mesh-free method (b) definition of a cut

Adapting the physical model

Both the approaches use the same way to update the function F by acting on the weights to the reciprocal influence of the phyxels and modifying such weights in term of the euclidean distance between phyxel, even if with different implementations.

The major contributions of this paper are:

1. A novel method for representing cut with fragmentation of the mesh with an implicit handling of topology, i.e. no need to know when and if to front merge.

2. A novel method to update the physical model efficiently and consistently.

The *Splitting Cubes* algorithm is a new technique to represent external surfaces of deformable bodies that implicitly represent topological modification like cuts or laceration. Each cubes represent a

part of volume of the deformable bodies and it can be considered as an independent entity that can represent the subpart of volume surface that it enclose.
We also propose a new technique,called *occlusion disk*, a simple way to to smoothly realize discontinuities on material's shape functions while new surface is being created.

4 MESHLESS SIMULATION OF DEFORMABLE BODIES

We give a brief introduction to the problem of animating deformable models and of the point based frameowrk used in this paper. For further details see [13], [8] or [20].
We call *rest space* the space represented by object volume at initial state when any deformation is introduced and total elastic energy is equal to zero. Given a solid in a 3-dimensional space let call call u a function that map each point x of the solid from *rest shape* to the corresponding position of *deformed shape* at world coordinate. Deformed shape is defined as $x+u$, where u is a 3D displacement vector field $u=(u,v,w)^T$. Shape deformation is quantified using *Green-Saint-Venant strain tensor*:

$$\varepsilon = \frac{(\nabla u + \nabla u^T + \nabla u \nabla u^T)}{2} \tag{1}$$

Elastic stress σ is linearly related to strain $\sigma = C\varepsilon$ where C is a rank four tensor that defines the constitutive law of the material. Strain energy density is defined as:

$$U = \frac{1}{2}(\varepsilon\sigma) \tag{2}$$

Now having energy definition we can derive the elastic force as:

$$f = -\sigma \nabla u \varepsilon \tag{3}$$

In mesh free methods, this equation is solved by sampling the volume in a finite set of elements. Each element called *phyxel* represents a portion of the volume and relative physical quantities as the position x_i, the displacements u_i, the velocities v_i, the density ρ_i, the strain ε_i, the stresses σ_i and the mass m_i. In order to express strain we must find a good estimation of first order derivative of displacement function (as states equation 1). To obtain a continuous estimation of displacement derivative the *moving least squares method* can be used (see [?] and [14] for details). Each phyxel ph_i has its *support radius* h_i, which is determined adaptively depending on the local density of nodes, and determines which other phyxels are taken into account to compute the derivative of the displacement function in ph_i. For each one of these *neighboring phyxels* n_j there is a weight value w_{ij} that is related to distance vector x_{ij} from phyxel ph_i to phyxel ph_j. Displacement derivative in the neighborhood of phyxel ph_i is finally estimated as:

$$\nabla u|x_i = A^{-1}(\sum_j (u_j - u_i) x_{ij} w_{ij}) \tag{4}$$

The matrix $A = \sum_j x_{ij} x_{ij}^T w_{ij})$ is a $3x3$ matrix that can be precomputed and inverted. Having displacement derivative he can estimate strain ε and stress σ and finally calculate elastic force, we also introduce in our framework strain state variables to model plasticity (see [13] and [7] for details). For each surfel is defined the initial position at rest shape x_{sfl} and a set of neighboring phyxels with relative position x_i, displacement vector u_i and displacement derivative ∇u_i estimated using MLS. Then surfel displacement u_{sfl} is updated each time step as:

$$u_{sfl} = \frac{1}{\sum_i w_i} \sum_i w_i (u_i + \nabla u_i^T (x_{sfl} - x_i)) \tag{5}$$

5 OUR APPROACH

The key idea of our approach is that the object surface is entirely defined by the intersection points of the object surface with the edges (and relative normals) of a regular decomposition of the space. Each cell with at least one edge crossed by the surface is associated with a small triangulation inside the cell. This resembles the Marching Cubes [12, 23] but our data structure, from now on referred to as the *Splitting Cubes*, holds the significant difference that the configuration of the cell is not given by the in/out sign of the nodes but directly by the intersections with the edges (in this sense it is more resembling of the Marching Intersections algorithm [21]). More important, the Splitting Cubes considers *all* the 2^{12} possible configurations and defines triangulations of the surface consistently. Every time the surface is updated, we also update the weights of the contribution of each phyxel to the state of its neighbors. Instead of estimating the shortest path connecting every couple of phyxels we estimate how visible they are to each other by taking advantage of the graphics hardware.

6 THE SPLITTING CUBES ALGORITHM

The goal of a splitting cube is to represent the portion of object's surface crossing a cell *and* to encode the function F inside the cell. We first consider the case in 2 dimension where the cells are quadrilaterals.
Let us consider an *internal* cell, i.e. a cell with all nodes inside the volume and without edge intersection (see Figure 3).(A). In this case there is no surface and the function F is continuous inside the cell. We simply define F inside the cell by an interpolating its value at the cell nodes.
Now suppose that the cut surface crosses two opposite edges as in (D). We want to represent the fact that the cell volume has been split in two parts, so we create the two surfaces cs_0 and cs_1 in the same location as cs at rest shape. Now the function F is not continuous across cs: in particular its value will be interpolated only by the two left nodes in the left portion of the cell and by the two right nodes in the right part of the cell. Note that we are interested in the value of F only in the two surfaces since it will be used to move the vertices on their tessellations.
We define each surface with a simple triangulations and for each vertex we store which nodes and relative interpolation parameters are used to compute its position in the deformed shape. Figure 3) illustrates all the 6 configurations for a face. In the left column the cell with cut surface at rest shape, in the center column the same configuration in as hypothetical deformation where the dependence of the vertices on the nodes is illustrated by an arrow.

You may note that the vertex in the middle of a face its necessary only in the configuration (B). In all the other cases it could be avoided, however in the cases (E)(F) this would cause a loss of volume and in all cases a rougher approximation of the real cut surface.
The 3D version of this decomposition scheme derives directly from the 2D case. In fact, the configurations shown in Figure 3 are those used in the 6 faces of the cube, and the configuration of the cube depends un-ambiguously on the configuration of its faces. These 2^{12} configurations are procedurally found as follows. The cube is visited node by node. For each node, the three edges connected to the node are visited to check if they are intersected by the cut surface. If an edge is intersected a quad is built, whose vertices are the vertex at the intersection with the edge, the two vertices in the middle of the two faces sharing the edge and a *center vertex* inside the cube. The dependence of the face vertices is then found by visiting the face edges and accessing the corresponding configuration. To define the dependence of each instance of the central vertex, we visit all the nodes by starting from the node examined. The instance of the central vertex depends on all the nodes reachable without crossing the cut surface.

We do not use this procedural approach during the simulation, instead we run over all the possible permutations once for all and store the result, i.e. the surface created and dependence of its vertices, in a look up table.

For the sake of completeness we observe that there are cases where the splitting cubes algorithm produces non manifold surfaces, like in the unlikely situation shown in Figure 4 (bottom right) where the surface is not homomorphic to a disk in the center vertex. Fortunately, even when this happens there are no consequences on our framework.

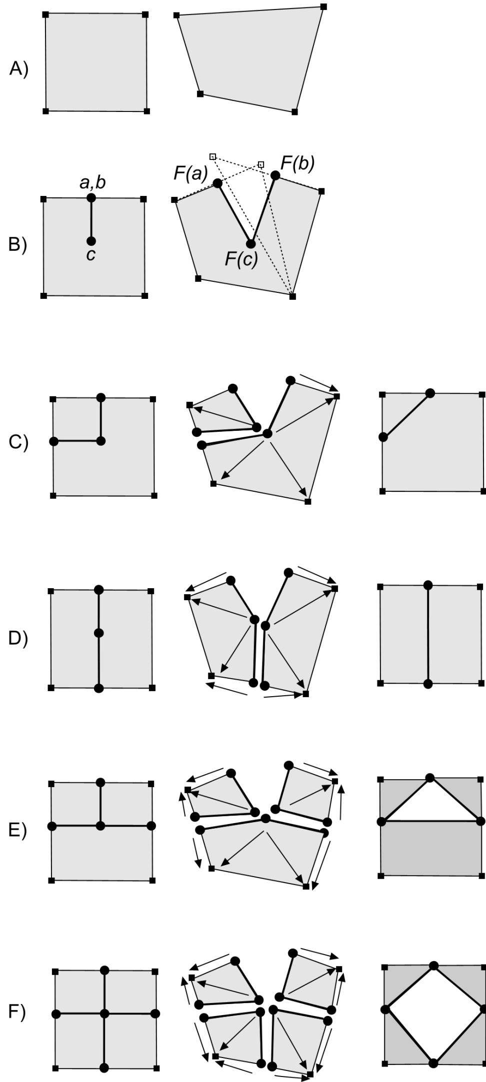

Figure 3: The 6 configurations for a face of the splitting cube. In the left column the configurations at rest shape; in the center column at a hypothetical deformed shape and in the right column the configuration is the face vertex was not used.

6.1 Position of the vertices

The surface created by the splitting cubes algorithm has three type of vertices: edge vertices, face vertices and center vertices. The position of these vertices should be chosen so to approximate the cut surface. For the edge vertices the position is obviously the intersection point of the surface with the edge. For the other two type there are many choices, none of which leads to best result for all cases. Our solution is illustrated in Figure 5. For the case (B)

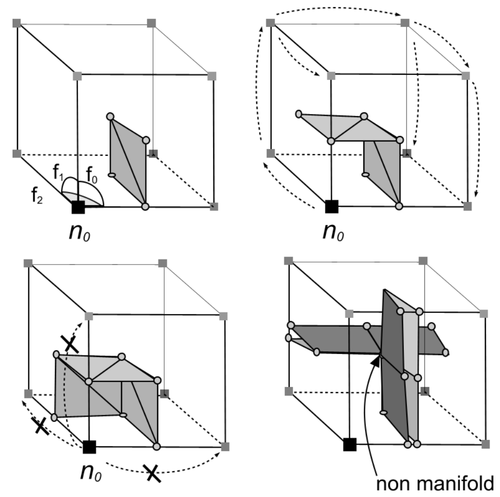

Figure 4: The three quads (6 triangles) that can be created processing thenode n_0. In the top right configuration the visit of the nodes to find out the dependences of the central vertex is shown with dashed arrows. In the bottom right a configuration resulting in a non manifold surface.

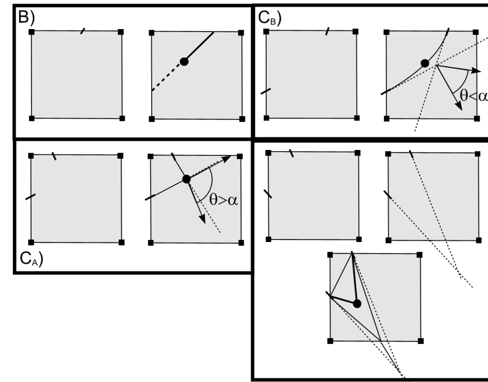

Figure 5: How the face vertex is derived by the edges vertices.

the face point is placed at the middle way from the corresponding edge vertex to the intersection of the tangent with another edge of the cube. For the case (C) and (D) (the most common) we first compute the intersection between the two tangent lines: if the intersection of the two tangents is outside the cube we clamp the intersection point inside the cube and then move it in the barycenter of the triangle formed by the clamped point with the two edge vertices. Once the point is calculated we distinguish two cases: if the normals form an angle θ greater than a predefined α then we decide that the cut surface has a sharp angle and place the vertex at the computed point; otherwise we use the two edge vertices and the calculated point to define a Bezier curve and we place the vertex in the middle of the curve. For the case (E) we solve for the two opposite edge vertices as in case (D) and then if the tangent line of the remaining edge vertex intersects the surface generated so far, such intersection point is taken. Finally for the case (F) we use the intersection point of the two segments connecting opposite vertices.

6.2 Data structures for the Splitting Cubes

The data structure for the Splitting Cubes is all in the grid nodes. Every node stores 6 couples (scalar,normal), one for each edge leaving the node on directions $-x, x, -y, y, -z, z$. If the scalar value is

the special value -1 there is no intersection with the relative edge, otherwise the couple stores the barycentric coordinate of the intersection point and relative normal of the cut surface at that point. The value of the normal is not necessary to find out the proper configuration but, as explained before, it is used to place the vertices on the faces of the cubes and it will be used for shading the surface. Furthermore every node stores, for each of the 8 cells that share it, the piece of surface created for that cell (at most 6 triangles per cell).

In order to find out the dependence of the vertices, we need to visit the cell walking on the edges, therefore for each node we need to find its 6 adjacent nodes. Since that the subdivision is regular the connectivity is implicit and the neighbors of the node (i, j, k) are the nodes at distance 1 in the grid. However a constant time random access to the grid would require to store all the nodes, while we need only those having at least one incident edge intersected by the cut surface. Therefore we use a hash table containing only the nodes instantiated and use a hash function of their position on the grid as suggested in [22].

6.3 The algorithm

The Splitting Cubes algorithm consists of processing a single type of event, that is, an edge is intersected by the cut surface. When this happens, the two nodes at the extremes of the intersected edge are processed to generate the respective surfaces, and all the other nodes in the 4 cells sharing the edge are reprocessed to recompute the dependence of the vertices of their surfaces. As previously mentioned, we use a look-up table to store the triangulations and the dependences of the vertices. So the look-up table has one entry for each configuration and for each node of the cell ($2^{12} * 8$ entries).

7 Physical response to cutting

In the previous section we introduced the Splitting Cubes algorithm, which enables the representation of cuts and the discontinuity in the shape function F inside a cell. This is a sort of *small scale* discontinuity since that we did not modify the function F at the nodes of the grid. This further step is more dependant on the specific model adopted. If, for example, Free Form Deformations was used to animate the object then the splitting cubes would be enough as it is, while is the grid was a set of masses connected by springs, we should remove the springs corresponding to intersected edges and so on. The case of mesh free method is slightly more complex because the function F (both at the grid nodes and at the phyxels) is determined by a group of phyxels.

Surfels - Phyxels bounds
The kernel of a surfel consists of all those phyxels within a given radius and such that the surfel is visible, i.e. the segment between a phyxel in the kernel and the surfel does not intersect the surface. We simply keep the connection updated performing the intersection tests with the new piece of surface created by the splitting cubes and creating the kernel for new surfels as in [19, 4]. This updating only concerns the surfels closer than r to the intersected edge, where r is the maximum radius among all the surfels of the object.

Phyxels - Phyxels bounds
As shown in [19], the visibility criterion introduced in [17], i.e. to eliminate the bound among two phyxels is they cannot see each other because of the surface, leads to excessive discontinuities that can compromise the stability of the system. Instead they use the method of transparency weights, which consists of weighting the bound between two phyxels with an approximation of the Euclidean distance between them. The same method with a more approximated estimation is presented in [4]. In this case the distance between two phyxels is computed on the connectivity graph, i.e. a graph with one edge for each couple neighbor phyxels,

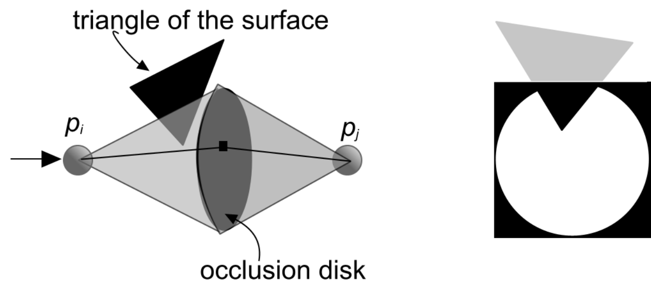

Figure 6: Left) the occlusion disk is sampled to see how many rays intersect the surface. The intersection test is left to the graphics hardware by rendering the surface twice and making the frustum equal to each of the cones. Right) an example of rendering from p_i

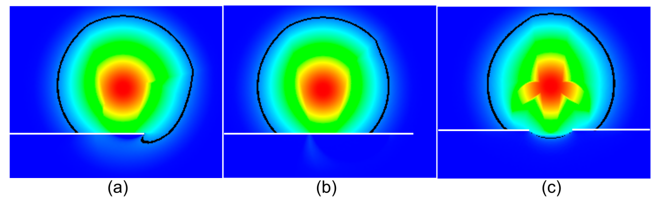

Figure 7: (a) the weight function when a cut surface (white line) is introduced (b) the cut surface is complete (c) with two cut surfaces

as the length of the path between the two phyxels such that no edge intersects the cut surface

We found these techniques not very practical for real time interaction, because the intrinsic cost of computing the Euclidean distance, and not free from tricky situations, like for example two phyxels that can actually see each other through a little hole in the cut surface.

7.1 Occlusion Disks

We chose to use a modified version of the visibility criterion, obtained by replacing the segment between two phyxels with two cones as shown in Figure 6. Consider the common base of these two cones, that we call *occlusion disk*. From each point on this disk we trace two rays, one for each phyxel and say that the point is obscured if at least one of these two rays intersect the cut surface. Then, assuming to sample the disk with n points, the weight is obtained as:

$$w'(p_i, p_j) = w(p_i, p_j) \frac{\#\dot{obscured}}{n}$$

The intuition behind this choice is straightforward: we replace the "if" with "how much" in the visibility criterion. Figure 7 shows the variation of the weight around a phyxel close to a cut surface. Note that, although it exhibits some discontinuities in the region very close to the phyxel, it happens to be fairly smooth at the average distance at which the phyxels are placed (we emphasized the isoline corresponding to such distance). In Figure 7.(c) the tricky situation above mentioned is reproduced by putting two cut surfaces without preventing the phyxels to directly see each other. Although the discontinuities around the phyxel are more obvious, the weight behind the hole is greatly reduced.

7.2 Implementing Occlusion Disks

The good thing of this method is that it can be entirely implemented on the GPU. Let cs the cut surface potentially occluding the mask between p_i and p_j and consider the smallest square enclosing the

occlusion disk. We associate a small single-channel texture to the square, and therefore to the occlusion disk, that stores how many samples of the disk have been obscured.
To update the texture we render *cs* twice: one from p_i towards p_j and one in the opposite direction, always setting the far plane of the projection in the mid way and setting the projection so that each sample in the disk project on the same pixel for the two renderings. In a fragment shader, we discard those fragments projecting on a pixel already written, so all the fragments that are written into the buffer correspond to newly obscured samples. By using the hardware occlusion query, we count these fragments and update the weight as in the formulae.
The size of the texture influence both smoothness and performance. A large texture would require a longer rasterization time but will provide a smoother change of the weight and viceversa. Another aspect to keep in mind is that we need one texture for each neighbor phyxels. Considering an average number of neighbors of 10 we need 10 by the number of phyxels textures.

8 Results

The approach presented in this paper was implemented on a Windows XP platform using C++ and OpenGL; the GPU code was written in GLSL. The tests shown in the accompanying videos have been run on a laptop Intel Centrino 1.7GH, 512 RAM equipped with a nVidia GeForge 6800 GO. Figure **??** shows the data about the performance of our method acquired during the video ”cut2.avi”. The model consists of 800 phyxels and 2400 splitting cubes intersected by the initial surface. The texture size for each occlusion disks is 16×16. The initial surface is obtained by the splitting cube algorithm, by taking a (watertight) mesh and considering it as a big cut surface. Once the cut is done, the surface bounding the empty space is simply thrown away.
It can be seen that during the cutting the frame rate drops down from 40 to 20, because of the time taken to generate the new triangulations and to update the occlusion disks. The total number of triangles grow linearly during the action from 8000 to 10000 and does not affect the frame rate. Looking at the videos you can spot the changing of the configuration of the splitting cube. As afore mentioned, this depends on the way the vertices are placed in the cell and, more important, by the size of the cell. There are two ways out of this artifact: to reduce the cell size or to make the cell more ”informative”.
In the first case we should speed up the rendering phase, which is currently much slower than we could make. This depends on the need for updating the vertices position by interpolation from the cell nodes which is done in the CPU. A possibility could be to pass to a vertex shader all the data for interpolation, but it is not obvious that it would result in a substantial speed up, since much of the work is retrieving the data for the interpolation and that should be done anyway. On the other side we could derive a more sophisticated look-up table for each configuration. When an edge is intersected, and a surface exists in one or more of the 4 cells sharing that edge, we should avoid that the existent surface moves as it does in the current version. This would requires a re-meshing algorithm to apply cell by cell, which should compose the new cut surface with the existent one.

9 Conclusions

In this paper we presented a novel solution for representing cuts on mesh free methods named Splitting Cubes Algorithm. In contrast with the previous solutions to this problem, the splitting cubes handles implicity all the changes of topology due to the cuts and it is extremely robust and conceptually simple. The splitting cubes algorithm always provides a well formed, watertight mesh which cannot be fragmented by repeated cuts in the same region. The second contribution of this paper is a novel scheme for a fast updating of the

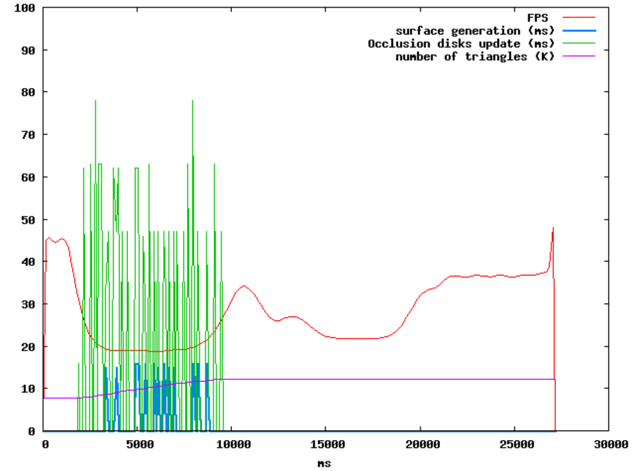

Figure 8: Data collected during video ”Cut2.avi”.

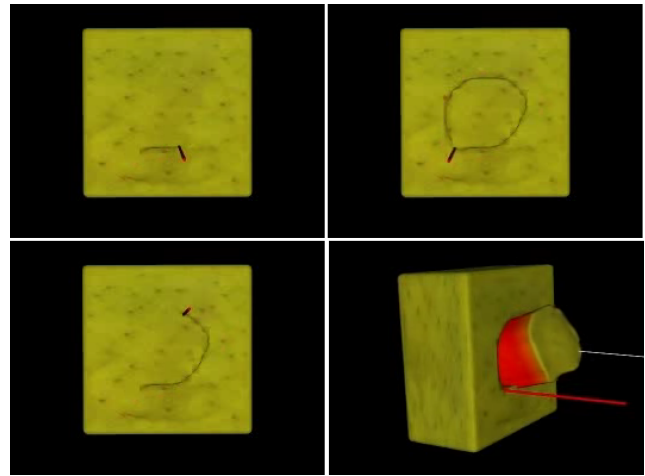

Figure 9: Some snapshots of a sequence where a cylinder is cut out of a cube like object.

physical system by introducing the occlusion disks, which provides a good approximation of the correct change of shape function and can be implemented entirely in the GPU.

Further work can be done on this approach, including incorporating state of the art solutions for the physical simulation, as re-sampling the volume with new phyxels whenever a cut is done. Without re-sampling, the object cannot be freely cut because phyxels that remain isolated easily leads to instabilities.

References

[1] D. Bielser, V. Maiwald, and M. Gross. Interactive cuts through 3-dimensional soft tissue. *Computer Graphics Forum (Eurographics'99 Proc.)*, 18(3):C31–C38, Sept. 1999.

[2] H. D. S. Cotin and N. Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. In *CAS99 Proceedings*, pages 70–81, May 1999.

[3] B. D. and G. M. Interactive simulaiton of surgical cuts. In *Proceedings of the Pacific Graphics*, pages 116–125, 2000.

[4] M. G. Denis Steinemann, Miguel A. Otaduy. Fast arbitrary splitting of deforming objects. In *Eurographics/SIGGRAPH Symposium on Computer Animation (to appear)*, 2006.

[5] M. Desbrun and M.-P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In R. Boulic and G. Hegron, edi-

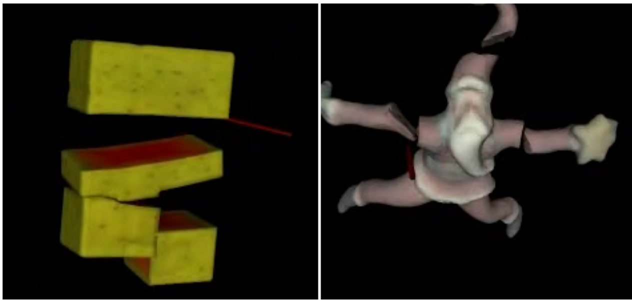

Figure 10: Left: multiple cuts on a cube Right: torturing Santa Claus

tors, *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, pages 61–76. Springer-Verlag, Aug 1996. Published under the name Marie-Paule Gascuel.

[6] M. Desbrun and M. Gascuel. Animating soft substances with implicit surfaces. In R. Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 287–290. ACM SIGGRAPH, Addison Wesley, Aug. 1995. held in Los Angeles, California, 06-11 August 1995.

[7] O. J. F., B. A. W., and H. J. K. Graphical modeling and animation of ductile fracture. In *Proceedings of SIGGRAPH*, pages 291–294, Aug 2002.

[8] M. H. G. Fries T.-P. Classification and overview of meshfree methods. Technical report, 2003.

[9] F. Ganovelli, P. Cignoni, C. Montani, and R. Scopigno. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum*, 19(3), 2000.

[10] F. Ganovelli and C. O'Sullivan. Animating cuts with on-the-fly remeshing. *EuroGraphics Short Presentations, 2001. (J. C. Roberts, editor)*, 2001.

[11] A. Lamouret, M. Gascuel, and J. Gascuel. Combining physically-based simulation of colliding objects with trajectory control. *The Journal of Visualization and Computer Animation*, 6(2):71–90, Apr.–June 1995.

[12] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM Computer Graphics (SIGGRAPH 87 Proceedings)*, volume 21, pages 163–170, 1987.

[13] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point based animation of elastic, plastic and melting objects. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, Aug 2004.

[14] A. Nealen. An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation. Technical report, 2004.

[15] H. Nienhuys. *Cutting in Deformable Objects*. Ph.d. thesis, Utrecht University, 2003.

[16] H.-W. Nienhuys and A. F. van der Stappen. Supporting cuts and finite element deformation in interactive surgery simulation. Technical report, Utrecht University, Institute for Information and Computing Sciences, PO Box 80.089, 3508 TB, The Netherlands, June 2001.

[17] T. T. Organ D., Fleming M. and B. T. 1996. Continuous meshless approximations for nonconvex bodies by diffraction and transparency. In *Computational Mechanics*, volume 18, 1996.

[18] M. Pauly, R. Keiser, B. Adams, P. Dutr;, M. Gross, and L. J. Guibas. Meshless animation of fracturing solids. *ACM Trans. Graph.*, 24(3):957–964, 2005.

[19] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 22(3):641–650, 2003.

[20] L. G. R. Mesh-free methods. Technical report, 2002.

[21] M. Tarini, M. Callieri, C. Montani, C. Rocchini, K. Olsson, and T. Persson. Marching intersections: An efficient approach to shape-from-silhouette. In *7th Int.l Fall Workshop on Vision, Modeling, and Visualization 2002*, pages 283–290, Erlangen (D), Nov. 20 - 22 2002. IOS Press.

[22] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, and R. Westermann, editors, *Proceedings of the Conference on Vision, Modeling and Visualization 2003 (VMV-03)*, pages 47–54, Berlin, Nov. 19–21 2003. Aka GmbH.

[23] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.