EG UK Theory and Practice of Computer Graphics (2007)
Ik Soo Lim, David Duce (Editors)

MTCut: GPU-based Marching Tetra Cuts

Eva Monclús, Isabel Navazo, Pere-Pau Vázquez

Modeling, Visualization, and Virtual Interaction Group
Universitat Politècnica de Catalunya, Barcelona
Email: emonclus@iri.upc.edu {isabel|ppau}@lsi.upc.edu

Abstract
Isosurface construction and rendering based on tetrahedral grids has been adequately implemented on programmable graphics hardware. In this paper we present MTCut: a volume cutting algorithm that is able to cut isosurfaces obtained by a Marching Tetrahedra algorithm on volume data. It does not require a tetrahedal representation and runs in real time for complex meshes of up to 1.8M triangles. Our algorithm takes as input the isosurface to be cut, and produces the cut geometry in response to the user interaction with a haptic device. The result is a watertight manifold model that can be interactively recovered back to CPU upon user request.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications I.3.4 [Computer Graphics]: Graphics UtilitiesVirtual device interfaces

1. Introduction

In computer-aided medical applications, data of the interior of the human body is obtained by systems such as Computer Tomography or Magnetic Resonance Imaging. These systems generate volume models consisting of scalar data sets over 3D uniform structured grids. The continuous improvements in capture devices allows sampling data at higher resolutions, which improves precision, but leads to a larger amount of information to be managed. Real time visualization of massive volumetric models has been an important challenge for many years. Recently, several approaches for real time GPU-based volume visualization algorithms have been proposed [EHK+06]. These are able to interactively analyze complex data sets, by using transparency or identifying the isosurface embedded in the volume. Some methods also focus on analyzing such data sets by using texture-based clipping algorithms. All these techniques have enabled the development of computer-aided clinical diagnose applications. However, in some applications such as planning, simulation, and surgical training, the rendering and manipulation of geometric models of anatomic structures is required. With the advent of new graphics hardware it has become possible to generate isosurface meshes on the fly from the volume models ([Pas04, RDS+04]).

Interactive cutting of volumetric models based on isosurfaces is of great interest in surgical planning (such as craniofacial surgery [CT96]). Most of the existing algorithms rely on the creation of a tetrahedral discretization in the volume bound by the isosurface. Later, according to a given cut, the topology of this tetrahedral mesh is modified (cut) and, if required, deformation computations are performed in order to determine the new position of the elements. The complexity of these operations makes it incompatible with real time rendering requirements [FDA05]. Some recent systems decouple the mechanical simulation and visualization. This allows modeling of fine surface detail without increasing the computational burden on the physical simulation, thus yielding a lower number of tetrahedra because they use a coarser tetrahedrization of the volume [SHGS06].

In this paper we present an approach for real time, progressive cutting of a complex isosurface model using a haptic device in a virtual reality environment without the need to pre-compute a tetrahedrization of the volume. We focus on the modelling of interaction with rigid objects such as the skull, represented with a manifold triangular surface mesh that has been obtained from the initial regular grid by means of a Marching Tetrahedra (MT) algorithm [DK91, YMDO93]. The cutting tool (blade) is represented by a segment. The cutting tool movement is tracked (only when the device is in contact with the mesh) and the blade path is approximated by a set of quads formed between two consecutive positions (segments) of the tool (it is a piecewise-bilinear surface). This is the approximation taken in the context of osteotomy applications such as the *LeFort I* [CT96] surgical operation for example.

Our application consists of two threads: The first one deals with the haptic device, from which it obtains a cutting path generated through the interaction. The second uses this cutting information in order to cut the geometry in real time by taking advantage of modern GPU capabilities. The proposed strategy for geometry cutting is based on the detection of the cells intersected by the tool path, their on-the-fly tetrahedization, and the reconstruction of the triangles based on the MT configuration of each tetrahedron after classifying its vertices according to both the isosurface and the cutting path. This approach has the advantages of generating a high-quality approximation and visualization of the cut path, it does not require any special preprocess, and guarantees the creation of manifold watertight meshes. Moreover, we can recover the cut manifold surface to CPU.

To sum up, the main contributions of our paper are:

- A real time, high quality algorithm for the visualization of interactive cuts of complex isosurfaces that does not require a tetrahedrization preprocess.
- Possibility of interactively undoing part of the cut without a cost penalty.
- Generation of manifold watertight triangle meshes.
- Recovery of the cut meshes from GPU to CPU.
- Maximum cut error bounded by the size of a tetrahedron in the initial regular grid.

The rest of the paper is organized as follows: Section 2 reviews related work on model cutting and GPU-based isosurface extraction, Section 3 gives an overview of our system, Section 4 gives details on the implementation, and Section 5 discusses the results of our work. Finally, Section 6 points to some directions of future research.

2. Previous Work

Isosurface generation on the GPU has been addressed in previous papers. The presented methods are generally based on the generation of the triangles of the surface using a Marching Tetrahedra algorithm [DK91, YMDO93, GH95]. There are two main approaches: those who use the fragment shader and multiple render passes to obtain the 3D geometry, such as in Klein *et al.* [KSE04] and Kipfer and Westermann [KW05], and those who generate the isosurface in a single pass in the vertex shader, such as in Pascucci [Pas04] or Reck *et al.* [RDS$^+$04]. The key difference is that vertex-based approaches produce the corresponding geometry in a single pass, whereas fragment-based approaches build the geometry in fragment shaders (using the render to vertex array facility) and therefore require a second rendering pass that uses this information. Fragment-based approaches have been claimed to be better due to the fact that vertex texture fetch is often slower than fragment texture accesses, and, moreover, some graphics cards do not even allow this kind of access. On the other hand, with the advent of new graphics architectures, which unify processors that do either vertex or fragment processing in the same processor, this will not be true anymore. Concerning the isosurface cutting, there are two different approaches that depend on the input model: a triangle mesh or a tetrahedrization of the interior volume of a mesh. Zachow *et al.* [ZGSZ03] address computer-assisted 3D planning of arbitrarily shaped osteotomies on polygonal bone models. Their objective is cutting a mesh and they do not add the triangles of the surface of the mesh's cut. Therefore, they obtain open surfaces with borders. Pintilie and McInerney [PM03] focus on interaction techniques in order to determine the cut. Their proposal may produce watertight surfaces, but can only be extended to volume cutting if the cut is shallow. Previous approaches on tetrahedral mesh cutting typically use three different strategies: a) erasing intersected tetrahedra, b) restricting the incisions to be aligned with existing faces, and c) tetrahedra subdivision into smaller ones. The first require a larger tetrahedrization in order to eliminate small amounts of volume and get an acceptable visual quality [CDA00, FDA05]; the second [SHS01] produce a low extra discretization of the mesh, but also require dense meshes; the third approach has the drawback that a good cut quality requires excessive subdivision of the intersected nodes [GCMS00, BGTG04]. Recently, some approaches focus in decoupling the simulation and visualization domain [SHGS06], which allows for modelling of fine surface detail without incurring an increase in the computational cost of the physical simulation.

Our proposal does not require the initial tetrahedrization of all the volume. Instead, the tetrahedrization of the intersected cells of the grid is computed on-the-fly. In [SCM06] the authors analyze the different tetrahedrization schemes and the produced error in isosurface reconstruction with respect to the Marching Cubes algorithm. Although we can use any implicit subdivision, since we have high resolution models, we have chosen a minimal conformal voxel splitting configuration as in [ABE$^+$03] (see Figure 2) as it trades off between the subdivision degree and the produced error. Independently of the tetrahedrization scheme, it is possible to lose part of volume when reconstructing the interior of a cut tetrahedron. Therefore, our modification of the topology is closer to the first of the strategies listed above, but working with a higher precision and a higher quality of the cut.

3. System Overview

In this Section we overview the application's architecture and cutting algorithm (details are left for the next section).

3.1. System Architecture

Our system has two threads: the *haptic thread* which deals with the haptic device and the *geometry thread*, responsible of cutting the geometry, as depicted in Figure 1.

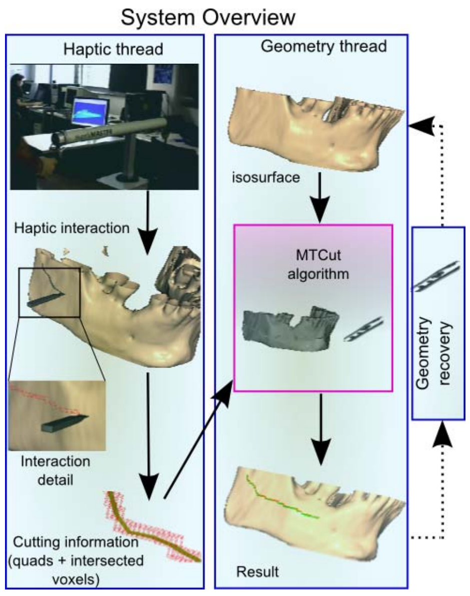


Figure 1: *Architecture of our system.*

The haptic thread samples the positions (and orientations) of the haptic tool and builds a set of sticks (segments) from them (with an assumed cutting tool length). Then, a quad mesh is built from those, by imposing the following conditions: only one stick is sampled inside each voxel, and coplanarity of every two consecutive sticks. This ensures that a voxel is cut by at most two planes. The voxels traversed by this surface are detected. The result (cut path and intersected voxels) is stored in a struct called *cutting information* that is passed to the subsequent rendering passes. The geometry thread executes a three step rendering algorithm. First, the intersected voxels are encoded in a texture. Then, a surface-based rendering algorithm draws the isosurface while a shader discards the fragments that belong to the intersected voxels according to the information obtained from the haptic thread. In the third step the new geometry, that is, the one affected by the cutting surface, is generated using our GPU MTCut algorithm (see Figure 4). In order to obtain the tetrahedrization of the affected voxels, we use a conformal voxel splitting configuration, shown in Figure 2, as in [ABE+03]. When needed, the created geometry is recovered to the CPU by using a rendering pass that encodes the new geometry in a texture.

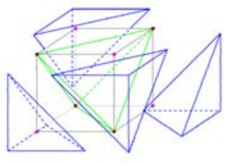


Figure 2: *Voxel splitting into 5 tetrahedra.*

3.2. Voxels cutting

MTCut algorithm has four steps: classification, pattern calculation, edge identification, and vertex generation. MT classifies vertices with respect to an isovalue. In our case, we classify the tetrahedron vertices according to an isovalue *and the cutting mesh* (see Figure 3). Once the edges are identified from the computed pattern, the vertex generation is also carried out in a different manner, as their positions are determined not only by the isovalue but also the cutting mesh.

For each tetrahedron being cut we use the classification of its vertices with respect to the isovalue (see Figure 3a) and the corresponding vertex classification with respect to the cutting surface (see Figure 3b). At voxel level, this is given by at most two planes. We reclassify the vertices of each tetrahedron setting values *inside* to those which are inside the final isosurface. As shown in Figure 3c, if the tetrahedron is cut by a single plane, points are classified as *inside* if they lie inside the isosurface and in the positive half space of the cutting plane. When we have two cutting planes, the classification is done according to the concavity/convexity that the cutting planes form with respect to the direction of the cut. This allows us to calculate the new vertex position to obtain the final triangulation of the cut (see Figure 3d). This is done in the following way: if the edge is cut only by the cutting plane, the vertex will be placed at the intersection of the plane with the edge. If the edge is traversed both by isosurface and cutting plane, we place the new vertex at the more interior of the two (except when the edges are shared with non cut voxels, where we keep the intersection with the isosurface). Notice that no cracks are generated between the newly created surface and the original isosurface. The reason is that the classification of all the vertices shared between cut voxels and non cut voxels is never changed. This generates the cutting geometry of the positive side of the cutting plane. By reversing its orientation we may reconstruct the remaining cut geometry.

4. MTCut Algorithm

In this Section we detail the implementation of the different steps of our algorithm, depicted in Figure 4.

4.1. Implementation details

The **first step** (*Texture coding*) determines the voxels that are traversed by the cutting tool. This information is encoded in a 2D texture. This is done in the following way: once we know the set of voxels that are cut by our cutting surface, we perform a first rendering pass where we render directly to a texture that encodes the voxels to be cut. Each texel (s,t) stores the information of the voxels whose coordinates are $(i = s, j = t, k = 0..depth)$, which means that the texel value is a bitmap that codifies the state of voxels: cut voxels take value 1 and non cut voxels take value 0. If the voxel (i, j, k) is cut, then, at texel $(s = i, t = j)$ the k-th bit will be set to 1. This is depicted in Figure 5. In order to do this, we only need to render a point for each cut voxel. We use blending function set to *GL_ADD* so, if we encode each different depth level in the color channels, we can generate a

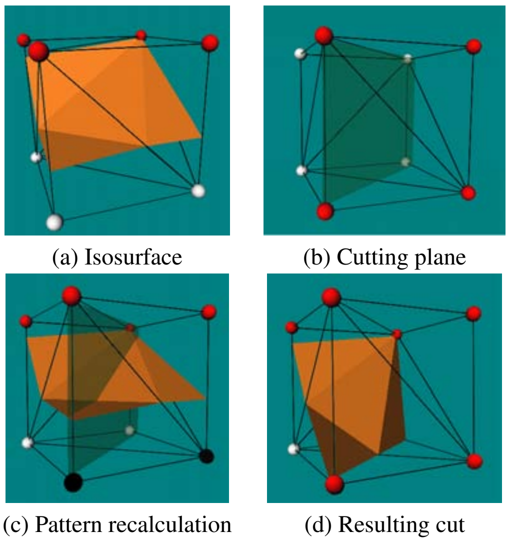

(a) Isosurface
(b) Cutting plane
(c) Pattern recalculation
(d) Resulting cut

Figure 3: *Cutting a voxel that contains isosurface by a plane. Red vertices indicate those that are* outside *the isosurface. White vertices are* inside *the isosurface, and the black ones, are the ones which were interior to the isosurface and now are extern to the plane and therefore are reclassified.*

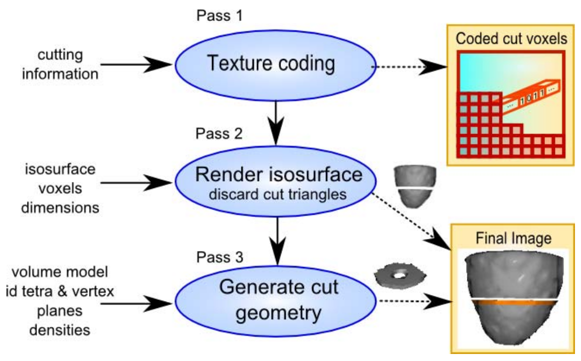


Figure 4: *Overview of MTCut: First step creates a texture with information of cut voxels, second renders the isosurface by discarding the triangles inside cut voxels, and third creates the cut geometry. Dashed lines indicate generated information.*

texture of dimensions 512 × 512 and 64 depth values. Note that, although our textures could hold up to 32 bits per color channel, blending operations are internally computed with only 16 bits. This reduces our available set to 64 different depth values in total. This is large enough for most applications. For larger volume models, we can use several alternatives. The first one is to increase the size of the texture we are projecting to. A size of 1024 × 1024 would allow us to encode up to 256 depth values. We can also use several textures. This possibility is especially interesting due to the fact that current GPUs are able to cope with several textures, and the *multiple render target* facility permits writing them in a single pass. A third possibility is to codify the color in CPU and pass it to GPU in order to generate this texture. Computing this texture in CPU slightly slows down framerates due to texture update and transfer, as the cut is defined interactively.

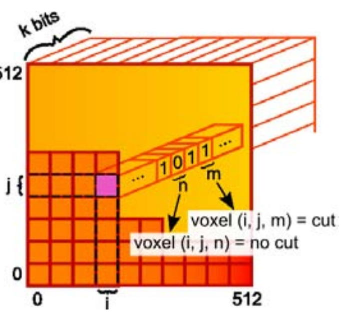


Figure 5: *Texture coding: Texel (s, t) stores the information of voxels (i = s, j = t, k). If voxel (i, j, k) is cut, then k-th bit of texel (s, t) will be set to 1, else it will be 0.*

The **second step** (*Render isosurface*) is responsible for drawing in the final image the isosurface that is not affected by the cut. This is carried out by using a *vertex buffer object* that stores the geometry of the original isosurface and a fragment shader that discards the triangles of the isosurface that lie inside the cut voxels. Note that this is not possible at vertex shader level because our isosurface representation does not repeat vertices for triangles that share the same vertex. Therefore, discarding a vertex (that is, degenerating the corresponding triangle) would eliminate all triangles that share that vertex, including those placed outside the voxels pierced by the cut. Despite that, it is important to remark that this can theoretically be done on the so-called *geometry shader* step of the new GPU architectures such as the NVidia GeForce 8800, which would also improve performance.

The original isosurface of the model remains unmodified throughout the whole process, as real time modification is not possible. This is the reason for the first step. Discarding triangles on the CPU is not an option because we would need to rebuild all vertex arrays every frame or use immediate mode (which slows the frame rate to half the fps we obtain with our method). Thus, we discard the fragments corresponding to these affected triangles on the GPU, which can be performed in real time. This strategy has another advantage: it affords the progressive cutting of the model together with the implementation of an interactive *undo* operation. Furthermore, we do not have artifacts on the boundary of the two geometries because both are generated using the same principles of the MT algorithm. This algorithm may incur in a small violation of the principle of mass conservation (not visually detectable) in special cases of slicing surfaces. Notwithstanding, given the actual resolution of the medical images, we postulate that the lost mass is smaller than the volume the blade may destroy in its real movement, and actually the upper bound is the volume of a tetrahedron.

Finally, all that remains is to create the geometry of the cut, which is done at the **third step** (*Generate cut geometry*). In order to do so, we have designed a MT-based algorithm that is executed in a vertex shader (cf. Section 3.2). It is similar to Pascucci's method [Pas04] in the sense that we are also sending a quad per each tetrahedron (belonging to the cut voxels), but we take advantage of the fact that we have a regular voxelization, and therefore the information we need to provide is different, as explained in Section 4.2.

The cutting vertex shader performs the following steps:

1. Calculate the coordinates of the four vertices of the tetrahedron given its index inside the voxel (see Section 4.2) and the origin and dimensions of the voxels model.
2. Create the marching pattern of the current tetrahedron using the scalar data at each vertex and the classification values of the vertices with respect to the cutting planes.
3. Generate the coordinates of the new vertex given the computed pattern classification.
4. Compute the normal associated to the new vertex (see the method in Section 5).

Figure 6 depicts the intermediate results of our algorithm for a simple model of a heart traversed by a quad. Note that we also create new geometry for all the voxels interior to the isosurface which are affected by the cut. For the examples in this paper, we will only reconstruct the isosurface that is located in one of the sides of the cut. Rendering both parts does not imply an important penalty, but the resulting images would not show the produced geometry because the caps share the cutting mesh. Both second (*Render isosurface*) and third (*Generate cut geometry*) steps render onto the same final image, therefore, the result is the combination of both: the first one renders the surface not affected by the cut and discards all the geometry close to the cutting mesh, and the final step creates the new geometry.

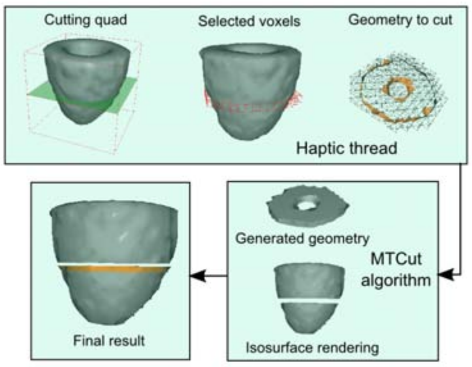


Figure 6: *Cut of a heart model. Orange triangles correspond to the newly created geometry.*

4.2. Data organization

The information required for the first two steps of our interactive cutting algorithm has already been discussed. The third step is the one that executes the actual cut. As already stated, each voxel can be pierced by a maximum of two cutting planes. We need the following information:

- Tetrahedron information: as we have a regular voxel grid, the information concerning the vertices of a tetrahedron can be determined by knowing in which voxel we are at (given by a triplet (i, j, k)) and the number of the tetrahedron: an integer value in the range $[0..4]$.
- Vertex index in the quad: an integer value in $[0..3]$.
- Cutting planes: they can be stored using 8 float values, represented by the signed distance of the vertices of the tetrahedron to the planes.
- Classification of each vertex with respect to the isosurface, encoded in one float value per vertex.

Since the set of cut voxels changes dynamically at each render step as the user moves the cutting tool, we use OpenGL immediate mode to pass this information to the GPU. Therefore, it will be important to reduce as much as possible the amount of information passed for each primitive (GL_QUAD). We will use the following vertex attributes:

- Position: 4 float values.
- Color: 4 float values.
- Texture coordinates: 4 float values.
- Normal: 3 float values.

Vertex attribute	Information contents
glVertex4f	Signed distances to isosurface
glMultiTexCoord4f	Signed distances to first plane
glNormal	Signed distances to plane 2 (fourth component in glColor)
glColor4f	Voxel coordinates, tetra index, vertex id, and distance to plane 2

Figure 7: *Data encoding for the third rendering pass.*

Unfortunately, the information we need to encode amounts a total of 17 values (5 per vertex, 8 plane coordinates and 4 values for the pattern configuration), and we have only 15 variables to store. We can rearrange the information in order to fit into this space. The information arrangement is shown in Figure 7. We take advantage of the fact that the voxel information consists of integer values and so the number of available bits is larger than we need. First, we store the classification of the vertex into the *glVertex* coordinates (x, y, z, w). Then, we encode the information of the tetrahedron and the current vertex in the color components (r, g, b). Red and green channels hold the i and j components of the voxel, while the blue channel encodes the k component together with the number of the tetrahedron and the vertex number of the associated quad. This is possible due to the fact that all these values are integers and relatively small, thus we can set blue to $k * 16 + tetra * 4 + vertex$. The alpha channel will be used for the second plane information. The first plane is encoded by storing the distances of each vertex to the plane in *glMultiTexCoord* coordinates. Finally, the second plane is encoded in *glNormal* values plus the remaining alpha channel of the color values. With this information we can determine the rest of the data necessary to perform the cut in a vertex shader. Note that in the third step we also send the 3D texture of the volume model in order to calculate the illumination in the fragment shader, as explained later.

4.3. Geometry recovery

So far, the presented algorithm is able to generate, in a frame-based fashion, the new geometry that represents the surface of the cut. Nevertheless, we have also developed an algorithm for the recovery of this newly created geometry by exploiting the capabilities of modern GPUs.

The main idea is to build a pair of textures where each texel represents the position (in the first texture) and the normal (in the second) of a vertex. The texels will be generated sequentially from the input information. In order to do so, instead of rendering quads, we issue a point (with the GL_POINTS primitive) per vertex. We will also pass to the vertex shader the texel position. The rest of the work is straightforward: the vertex shader will compute the actual position and pass it to the fragment shader and this will write both position and normal in two auxiliary buffers. Furthermore, it also detects if the generated geometry is a triangle or a quad and flags it in the alpha channel so the recovery process at the CPU is simplified. This process can be carried out quite fast in a modern GPU. The cost depends on the size of the texture to be captured back to the CPU, which depends on the number of traversed voxels. Our experiments show that textures of up to 512×512 can be read back to main memory and processed in order to obtain the encoded geometry in 100 to 120 ms. Once the new geometry has been recovered to CPU, we can fuse it with the current isosurface (by substituting the triangles in the cut voxels by the new triangles and updating the neighborhood relations). Then, a flooding algorithm may detect the present shells in the resulting geometry, which allows to separate the models.

5. Results

In Figure 8a we can see an example of a cut in the jaw model extracted from a volume model of $512 \times 512 \times 40$. The correct matching between new and original geometry may be visually inferred from the continuity of shading (Figure8a bottom center). This is also illustrated if the new geometry is shown, as it can be appreciated in Figure 9, where a close-up of of a cut of the heart model ($64 \times 64 \times 22$ voxels) is shown in wireframe for comparison purposes.

In order to obtain a good illumination quality, in all the examples we use a Phong fragment shader implementation. This requires the computation of a normal per pixel, which we do in three different ways:

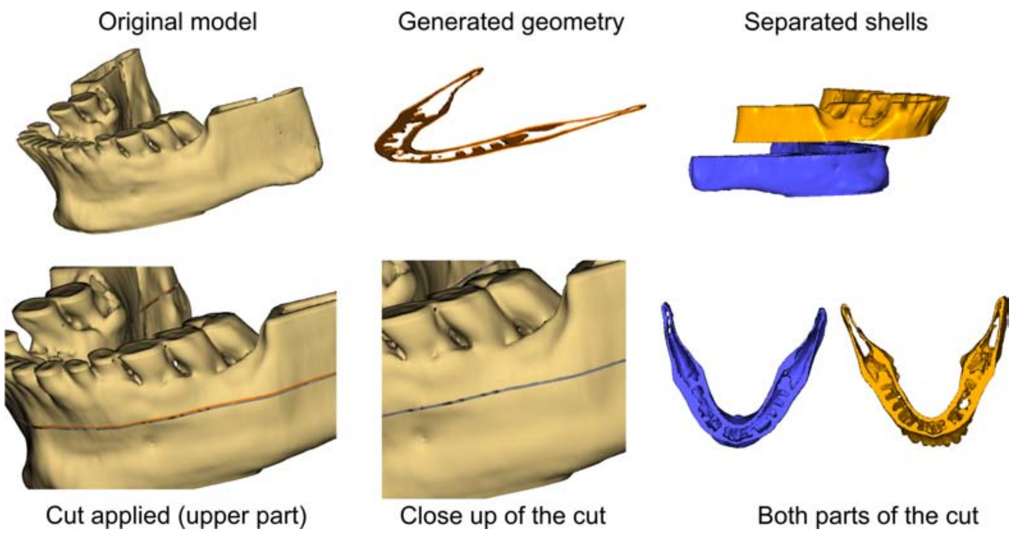
- For the triangles of the initial isosurface, we interpolate the normals at their vertices (calculated using the gradient formulation).
- For new triangles of the cut that belong to tetrahedra that contain isosurface, we approximate the normal from the gradient at this point using the volume information stored in the 3D texture. This is performed in the fragment shader because 3D texture access at vertex shader level is extremely slow.
- For new triangles that belong to tetrahedra that only contain cutting planes (they are interior to the isosurface), we calculate the average of the normal of the cutting planes (computed using the distance of the vertex's tetrahedron to the plane as in [ABE+03]). This has to be done in the vertex shader.

We have tested our algorithm in a P-IV PC equipped with 1Gb of RAM memory and a GeForce 6800 Ultra graphics card with 256Mb of memory. Our haptic device is a HapticMASTER from FCS. Table 1 shows the results obtained with different models, namely: the heart model, the jaw (with two different resolutions), the hip, and the head (see Figure 8b and c). For all of them, different cuts have been performed, the number of voxels pierced by the cuts is shown in the fourth column. The remaining columns show the framerates obtained by: using Phong shading rendering of the isosurface (sixth column), executing the first and second steps of our algorithm (texture creation and isosurface rendering with collapsation of the affected voxels), and finally, the last column shows the framerates of the overall process. Note that, as expected, our algorithm is sensitive to the number of voxels traversed by the cutting geometry. This can be used to recover geometry when part of the cut is considered to be stable, which will lead us to framerates equivalent to those of the sixth column. It is important to remark that the number of triangles we originate is linear with the cutting surface, and, in this sense, it is not higher than the ones that would be generated by a MT of the cut surface. Some videos can be seen in *www.lsi.upc.edu/∼moving/MTCut.xml*

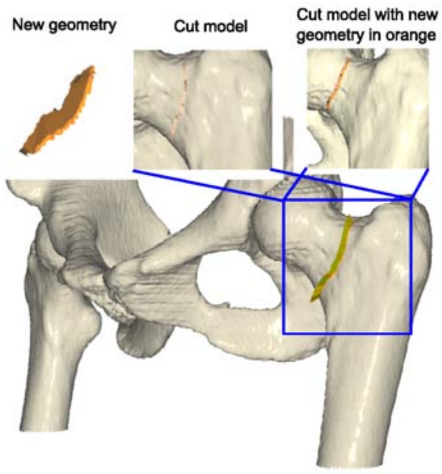
It is not possible to perform the whole process (complete GPU-based isosurface generation and isosurface cutting) at the same frame rates. We have implemented the algorithm by Pascucci [Pas04] and only the generation of the isosurface with a model like the jaw yields only 7 fps for the middle model ($256 \times 256 \times 40$) and approximately 3 fps for the complete model ($512 \times 512 \times 40$).

6. Conclusions and Future Work

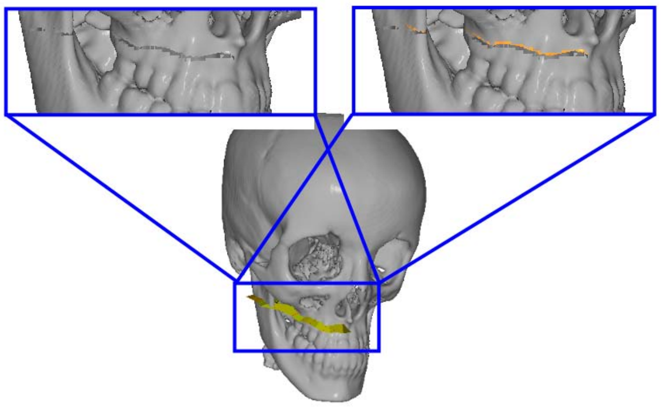
We have presented an interactive cutting algorithm that performs cuts on volume models. Our rendering method combines the information of a previously built isosurface and interactively created cut surface from the volume model, creating a manifold watertight surface. In the future we want to work on improving the framerates by using the new GPU architectures. Furthermore, we want to add force feedback to the haptic interaction (currently, the haptic tool is only



(a) A cut performed in the jaw model. Top left shows the original model and bottom left the cut model. We can see the generated geometry in the top middle image and, at its bottom, a close-up of the cut region that shows the continuity in shading when using the same color. The right column shows the jaw separated in two different shells.



(b) Hip model cutting



(c) Skull cut

Figure 8: *Several snapshots of the interactive cutting process. The first two rows show the cutting of the jaw model. The last row shows a hip and a skull cutting.*

used as an input device). We also want to enhance the visualization in order to help the surgeon visualize the volume information interior to the isosurface, as this reveals noble structures (vessels, nerves...), that should not be cut.

Acknowledgments

The authors thank Iban Lozano and Héctor Yela for their help with some parts of the implementation. Work supported by TIN2004-08065-C02-01 of Spanish Government.

References

[ABE+03] C. Andujar, P. Brunet, J. Esteve, E. Monclús, I. Navazo, and A. Vinacua. Robust recovery for hybrid surfaces visualization. In *8th International Fall Workshop on Vision, Modeling, and Visualization, VMV'03*, pages 215–222, Munich, Germany, 2003. 2, 3, 6

[BGTG04] D. Bielser, P. Glardon, M. Teschner, and M. Gross. A state machine for real-time cutting of tetrahedral meshes. *Journal of Graphical Models*, 66(6):398–417, 2004. 2

[CDA00] S. Cotin, H. Delingette, and N. Ayache. A hy-

Model	Dimensions	Triangles	Intersected voxels	fps original	Steps 1 and 2	Steps 1 to 3
Heart	$64 \times 64 \times 22$	5956	182/251	125	125	125/125
Jaw	$256 \times 256 \times 40$	549942	550/3495	100	69	62/40
Jaw	$512 \times 512 \times 40$	1851016	419/3638/11716	40	30	30/23/15
Hip	$512 \times 512 \times 100$	1184544	132/598/1401/4116	64	45	44/43/39/29
Head	$256 \times 256 \times 196$	1428084	171/744/1641	46	31	32/30/28

Table 1: *Timings of different models. Columns 1 to 3 show the model and the resolution, The next column shows the number of intersected voxels in different cuts, and the following columns the framerates obtained. "*fps original*" shows the results of isosurface rendering, without any cut. Column "*Steps 1 and 2*" shows the results of the first two steps of our algorithm: texture creation and voxels collapsing. Finally, the last column shows the fps of the whole algorithm.*

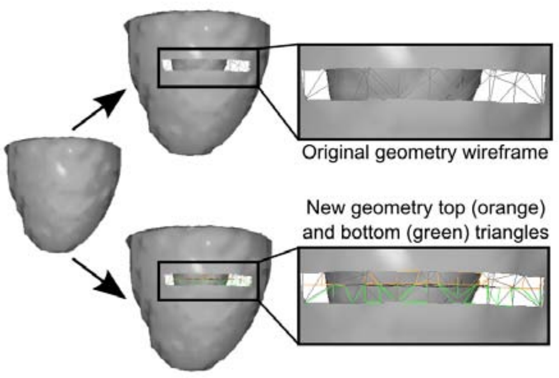

Original geometry wireframe
New geometry top (orange) and bottom (green) triangles

Figure 9: *A zoom-in of the heart model, note the continuity between original (top) and new (bottom) geometry.*

brid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. *The Visual Computer*, 16(8):437–452, 2000. 2

[CT96] Bookstein, F. L. Cutting, C. B. and Taylor, R. H. Applications of simulation, morphometrics, and robotics in craniofacial surgery. pages 641–662. 1996. 1, 2

[DK91] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Transactions*, 74(1):214–224, 1991. 1, 2

[EHK$^+$06] K. Engel, M. Hadwiger, J.M. Kniss, Ch. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006. 1

[FDA05] C. Forest, H. Delingette, , and N. Ayache. Removing tetrahedra from manifold tetrahedralisation : application to real-time surgical simulation. *Medical Image Analysis*, 9(2):113–122, 2005. 1, 2

[GCMS00] F. Ganovelli, P. Cignoni, C. Montani, and R. Scopigno. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum (EG 00)*, 19(3):271–282, 2000. 2

[GH95] A. Guéziec and R. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Trans. on Visualization and Computer Graphics*, 1(4):328–342, 1995. 2

[KSE04] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004. 2

[KW05] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In *Proc. Vision, Modeling and Visualization*, pages 241–248. IOS Press, infix, 2005. 2

[Pas04] V. Pascucci. Isosurface computation made simple. In *VisSym*, pages 293–300, 2004. 1, 2, 5, 6

[PM03] G. Pintilie and T. McInerney. Interactive cutting of the skull for craniofacial surgical planning. In *IASTED International Conference Biomedical Engineering (BioMed2003)*. Acta Press, 2003. 2

[RDS$^+$04] F. Reck, C. Dachsbacher, M. Stamminger, G. Greiner, and R. Grosso. Realtime isosurface extraction with graphics hardware. In M. Alexa and E. Galin, editors, *Eurographics Short Papers*, pages 33–36. Blackwell Publishers, 2004. 1, 2

[SCM06] J. Snoeyink, H. Carr, and T. Moller. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, 2006. 2

[SHGS06] D. Steinemann, M. Harders, M. Gross, and G. Szekely. Hybrid cutting of deformable solids. In *Proc. of IEEE Conference on Virtual Reality 2006)*. IEEE, 2006. 1, 2

[SHS01] D. Serby, M. Harders, and G. Szekely. A new approach to cutting into finite element models. In *Proc. of Int. Conf. on Medical Image*, pages 425–433, 2001. 2

[YMDO93] R. Yoshida, T. Miyazawa, A. Doi, and T. Otsuki. Clinical planning support system-clipss. *IEEE Comput. Graph. Appl.*, 13(6):76–84, 1993. 1, 2

[ZGSZ03] S. Zachow, E. Gladilin, R. Sader, and H.-F. Zeilhofer. Draw and cut: intuitive 3d osteotomy planning on polygonal bone models. In *CARS*, pages 362–369, 2003. 2