Abstract

Isosurface construction and rendering based on tetrahedral grids has shown to be feasible 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 tetrahedral 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, slices it, and produces the cut geometry in response to the user interaction with a haptic device. The result is a watertight manifold that can be interactively recovered back to CPU in response to a user request.

1. Introduction

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

Interactive cutting of volumetric models based on isosurfaces is of great interest in osteotomy planning or mesh deformation computations. Most of the existing algorithms rely on the creation of a tetrahedral discretization in the interior of 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 to a lower amount of tetrahedra because they can use a coarser tetrahedrization of the volume [SHGS06].

In this paper we present an approach for fast, real time, progressive cutting of a complex isosurface model using a haptic device in virtual reality environment without the need of pre-computing a tetrahedrization of the volume. We limit ourselves to the modelling of interaction with rigid objects as the skull, represented with a manifold triangular surface mesh. This has been obtained from the initial regular grid by means of a Marching Tetrahedra algorithm [DK91, YMDO93]. The cutting tool (blade) is represented by a segment. The haptic system detects its intersection with the surface and indicates its intersection with the volume. The tool movement is tracked and the blade path is approximated by a set of quads formed between two tracked positions of the tool (it is a piecewise-linear surface). This kind of tool’s movements are usual in the context...
of osteotomy applications such as surgical operation LeFort I [CT96] among others.

Our application consists of two threads: the first one deals with the haptic device, from which it obtains a cutting path generated by the interaction with the haptic tool, 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 tetrahedrization, 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. Finally, the triangles are created according to this new configuration. This approach has the advantages of generating 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 final cut manifold isosurface to the CPU. However, in some cases we have the drawback of a bounded violation of the principle of mass conservation although it is not perceived in the rendering. Given the actual resolution of the medical images and that the upper bound of volume mass loss is the volume of a tetrahedron, we consider that the maximum loss is smaller than the mass the cutting tool may destroy in its real movement.

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 tetrahedrization preprocess.
• Possibility of interactively undoing part of the cut without cost increase.
• Generation of manifold watertight triangle meshes.
• Recovery of the cut meshes from GPU to CPU.
• Maximum cut error bounded by the size of a cell 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 some lines 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, YMD093, 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 do produce the corresponding geometry in a single pass, whereas fragment-based approaches do build the geometry in fragment shaders (using 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 access is often slower than fragment accesses, and, moreover, some graphics cards do not even enable 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 to isosurface cutting, there are two different approaches that depend on the input model: a) a triangle mesh or a tetrahedrization of the triangular 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 do not build the triangles of the surface of the mesh. Therefore, they obtain open, non manifold surfaces. Pintilie and McInerney [PM03] focus in interaction techniques in order to determine the cut. Their proposal may produce watertight surfaces, but can be only extended to volume cutting if the cut has no large depth.

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 still 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, BGTT04]. Recently, some approaches focus in decoupling the simulation and visualization domain [SHGS06], that allow for modelling of fine surface detail without incurring in increase in computational cost of 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, which permits us to work with dense tetrahedralized sets. Initial element subdivision is not generated, but it is possible to lose part of the interior volume when reconstructing the interior of a cut tetrahedron. Therefore, our modification of the topology is closer to the first of the presented techniques, but working with a higher precision, a higher quality of the cut (less zigzags), and without the need of a previous tetrahedrization.

A related problem, boolean operations on the GPU, has also been addressed. Weiskopf et al. [WEE03] use volume models encoded by 3D textures (instead of triangulated isosurfaces). Hable and Rossignac build a linear formulation of boolean expressions and use depth peeling for fast rendering.
in GPUs [HR05]. However, the goals and requirements of both works are completely different to ours.

3. System Overview

In this Section we overview the architecture of our application and outline the steps of the real time cutting algorithm. The details of this last algorithm are left for the next section.

Figure 1: Architecture of our system.

3.1. System Architecture

As already stated, our system has two threads: the haptic thread which deals with the haptic and the geometry thread, responsible of cutting the geometry, as depicted in Figure 1. The haptic thread samples the positions of the haptic tool and builds a set of sticks from them. 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 fourth vertex. This ensures that a voxel is cut by at most two planes. The voxels traversed by this surface are detected. The result is stored in a struct named 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-accelerated Marching Tetra-based algorithm (see Figure 4). In order to obtain the tetrahedralization of the affected voxels, we use a conformal voxel splitting configuration, shown in Figure 2, as in [ABE03]. When needed, the created geometry is recovered to CPU by using a rendering pass that encodes the information in a texture.

3.2. MTCut Cutting Algorithm

In order to perform the cut, we proceed in a similar way than the Marching Tetrahedra algorithm. In our case, we need, not only the isovalue that is used to create the isosurface, but also the cutting mesh which will be used build the surface of the cut. At voxel level, this is given by at most two planes (our cutting geometry is a set of connected quads whose size is larger than a voxel). Figure 3 shows how the surface creation is achieved for a single voxel. Note that this yields a different algorithm than that of the Marching Tetra.

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). 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 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 Marching Tetra pattern to obtain the final triangulation of the cut (see Figure 3d). This is done 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 both (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.

4. MTCut Algorithm

In this Section we detail the implementation of the different steps of our algorithm, depicted in Figure 4.
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 external 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 cut voxels, and third creates the cut geometry. Dashed lines indicate generated information.

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 which is 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 \((i = s, j = t, k = 0 \ldots \text{depth})\), 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 texture of dimensions \(512 \times 512\) 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 big enough for a high number of applications. For larger volume models, we can do several different approaches. The first one is to increase the size of the texture we are projecting to. A size of \(1024 \times 1024\) 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 to issue multiple results per pixel position. 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 cut is defined interactively.

The second step (Render isosurface) is responsible of 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 correspondent triangle) would cause to wipe out 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. Furthermore, due to the incremental and interactive generation of cut geometry, it is easy to implement the undo operation.

Finally, it is only necessary 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 Marching Tetra-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 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. Since discarding triangles on the CPU is not an option, we discard the fragments corresponding to these affected triangles on the GPU. This can be performed in real time. This strategy has another advantage: it allows us for the progressive cutting of the model together with the easy implementation of an 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 Marching Tetra algorithm. This algorithm may incur in a small violation of the principle of mass conservation, though not visually detectable, in special cases of slicing surfaces. Notwithstanding, the lost mass can be considered smaller than the volume the blade may destroy in its real movement, and actually the upper bound is the volume of a tetrahedron.

The cutting vertex shader performs the following steps:
1. Calculate the coordinates of the four vertices of the tetrahedron given its tetrahedron index and the rest of the incoming data.
2. Create the marching pattern of the current tetrahedron using the scalar data of 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 planes, and the final step creates the new geometry.

4.2. Data organization

As explained, our interactive cutting algorithm involves three rendering steps. The information required for the first two has already been explained. The third step is the one that executes the actual cut. In order to generate correct geometry (that matches the rest of the isosurface), we have implemented a version of Marching Tetra algorithm in a vertex shader. Each voxel can be pierced by a maximum of two cutting planes. We need the following information:

- Tetrahedron information: as we have a regular voxels world, the information concerning to the vertices of a tetrahedron can be determined by knowing which voxel are we placed 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 tetrahedra to the planes.
- Classification of each vertex respect to the isosurface, encoded in 4 float values.

As 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.

Unfortunately, the information we need to encode amounts a total of 17 values (5 per vertex, 8 plane coordi-

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{Vertex attribute} & \textbf{Information contents} \\
\hline
\texttt{glVertex4f} & Signed distances to isosurface \\
\texttt{glColor4f} & Voxel coordinates, tetra index, vertex id, and distance to plane 2 \\
\texttt{glMultiTexCoord4f} & Signed distances to first plane \\
\texttt{glNormal} & Signed distances to plane 2 (fourth component en glColor) \\
\hline
\end{tabular}
\end{table}
mates and 4 values for the pattern configuration), and we have only 15 values to store. Nonetheless, we can rearrange the information in order to fit the values 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 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 \(k + 64 + tetra * 4 + vertex\). As we will see, 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.

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 represent 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 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 CPU, and this depends on the number of traversed voxels. Our experiments show that textures of up to \(512 \times 512\) can be read back to main memory and processed in order to obtain the encoded geometry in 100 to 120 ms.

5. Results

In Figure 8a we can see an example of a complex cut in the jaw model extracted from a volume model of \(512 \times 512 \times 40\). The path was originated with a haptic tool. Figure 8b shows the model together with the voxels that have been determined as pierced by the cut (in red). In Figure 8c the geometry not affected by the cut has been rendered. Figure 8d shows the generated geometry for both sides of the cut (green and orange). Finally, in Figure 8e we can see the positive side of the cut, and in Figure 8f we can see a close-up of the new geometry. The correct matching between new and original geometry may be visually inferred from the continuity of shading. This can be better seen in Figure 9, where a close-up of the heart model \((64 \times 64 \times 22\) voxels) created geometry is shown in wireframe for comparison purposes.

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

- For the triangles of the initial isosurface, we interpolate the normals at its 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 with the distance to the plane using the algorithm in [ABET03]. This has to be done in the vertex shader.

We have tested our algorithm in a PIV 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 coxis, and the head. 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 collapse 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 progressively 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 marching tetra of the cut surface.

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...
Figure 8: Several snapshots of the interactive cutting process. The first two rows show the cutting of the jaw model. Note that the cut trajectory is not realistic in the sense that surgical cuts are mostly straight in order to allow for posterior displacement of the cut structures, as in LeFort I (see [CT96]). The last row shows a coxis and a skull cutting.

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 previously built isosurface and interactively created cut surface from the volume model by creating a manifold watertight surface.

In future we want to work in improving the framerates and in the use of the recovered generated geometry for further processing, such as parts manipulation. We also want to enhance the visualization in order to help the surgeon in order to show the volume information interior to the isosurface, as this reveals noble structures (vessels, nerves, and so on). These should be not affected by the cut. To do this we will develop a hybrid visualization approach that combines volume and isosurface rendering.

References


[BGTG04] D. Bielser, P. Glardon, M. Teschner, and
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 obtained framerates. fps original holds 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.

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