Central European Multimedia and Virtual Reality Conference (2006)
C. Sik Lányi, B.Oláh (Editors)

Global Mesh Partitioning for Surgical Planning

Philipp Fürnstahl[1], Bernhard Reitinger[†][1], Reinhard Beichel[1], Dieter Schmalstieg[1]

[1]Institute for Computer Graphics and Vision
Graz University of Technology, Austria

Abstract

We present a set of partitioning tools that classify a tetrahedral mesh into different regions of interest while preserving mesh consistency. These regions can then be individually visualized, repositioned, or combined for further analysis or processing. A partitioning operation, either defined analytically (by a formula) or geometrically (by a surface mesh), is applied globally to the model. A hierarchical data structure is used to store region information and consecutive partitioning operations: it ensures consistency between the specified regions of the volumetric mesh and the visualized surface mesh. Similar to volumetric cutting, subdivision is used to split the initial model into regions. Subdivision of tetrahedra that contain multiple intersection points per edge is a non-trivial task. An extension to existing subdivision methods is presented which handles the subdivision of such tetrahedra in an iterative way. Since the partitioning of a volumetric mesh is an important task in surgical planning, this paper finally shows that the presented algorithms can be successfully integrated in a virtual reality planning system.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: I.3.8 Applications

1. Introduction

Interactive tools, which partition the underlying volumetric model into regions of interest, give new opportunities in visualization and planning systems. These tools can be either applied progressively to the model (e.g. a scalpel cut specifies the partitioning surface) or globally, by interactively defined partitioning shapes. We aim at the second method: a partitioning operation is used to divide the entire mesh into disjoint regions. Consecutive operations can be used for more complex partitioning.

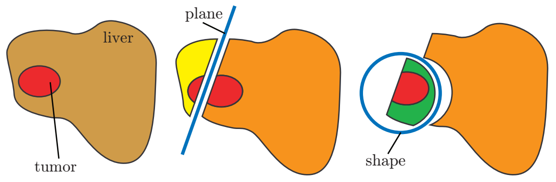
In our concept, the shape of the tool, which can be modified interactively, defines how the affected parts of the mesh will be separated into regions. Each specified region can then be treated as an individual sub-model. This improves interactive planning since quantitative analysis can be performed on particular regions (or a set of regions). Moreover, particular regions can be visualized, hidden, or modified for more accurate intervention planning. Regions, created during consecutive partitioning operations, are stored in a hierarchically organized data model based on a binary tree. The binary tree approach has the benefit that existing information is preserved and previously specified regions are still available. This feature is very important for planning environments but usually not considered by conventional cutting simulations where old information is discarded. The binary tree is also used to consistently combine regions for visualization (interface extraction) or analysis without node or face doubling.

In this paper, we introduce a new approach of global tetrahedral mesh partitioning (especially focusing on very large meshes) on the basis of three algorithms. In contrast to other approaches, the same scheme can be used for both, analytically *and* geometrically defined partitioning tools. Moreover, complex partitioning shapes (e.g. non-planar) can be used. Analytical partitioning is restricted to objects which can be described by a formula so it can be carried out efficiently. Geometrical partitioning shapes are represented by triangulated surface meshes and are used for more flexible partitioning. The developed partitioning algorithms operate on tetrahedral meshes. In contrast to other approaches, we designed the algorithms to work with high-resolution tetrahedral meshes. We accept higher computation times but

[†] contact: reitinger@tugraz.at

achieve highest quality for both meshes, the tetrahedral mesh and the visualized surface mesh.

Partitioning is especially useful for surgical environments where surgeons need to interactively classify an organ into healthy (benign) and diseased (malignant) tissue. For our target application, the intervention of a liver tumor resection must be planned using partitioning tools [Rei05]. Fig. 1 sketches an example of a liver model which will be partitioned into healthy and diseased regions. The partitioning procedure starts by locating a tumor in the initial mesh. Next, the dataset is partitioned using a plane in order to uncover the tumor. Two new regions are created, colored yellow and orange. Subsequently, the left part of the liver (yellow region) is hidden. A suitable shape is then used to apply a more specific partitioning operation for removing the residual. Finally, the binary tree is used to visualize and measure volumes of the healthy and diseased regions in order to obtain important quantitative indices.



(a) initial model (b) plane partitioning is applied (c) shape partitioning is applied

Figure 1: *A medical example showing the classification of a tetrahedral mesh into disjoint regions.*

2. Related Work

The partitioning operations presented in this paper are conceptually related to constructive solid geometry and virtual sculpting but share many technical aspects with existing volumetric cutting techniques.

Ganovelli et al. [GCMS01] presented a technique to enable cuts with a shape on multi-resolution representations. A lookup table is used to determine which basic subdivision case is necessary. Simplicial complex properties are always preserved. In contrast to our approach, this method is restricted to planar cut shapes but supports shape faces poking inside tetrahedra.

Another approach for specifying virtual resections in liver surgery planning is described by Konrad-Verse et al. [KVPL04]. A deformable cutting plane is used to define the cut shape. Our algorithms are not limited to planes or deformable planes and can use other complex shapes as well. In addition, Konrad-Verse's algorithm is based on vertex displacement instead of subdivision yielding more approximate results. Unfortunately, neither the performance nor the accuracy of the algorithm are discussed.

A progressive cutting algorithm, using a state machine, was recently published by Bielser et al. [BGTG04]. They outlined a hierarchical subdivision method where already subdivided tetrahedra can further be split, based on the transition of the state machine. In addition, example applications are described where scalpel cutting is used to partition and visualize regions of interest.

Based on this method, Steinemann et al. [SHGS06] introduced a real-time cutting approach where the low-resolution mechanical model is decoupled from the high-resolution visualized model. This allows efficient real-time cutting, however, the visualized surface mesh is only an approximation of the tetrahedral mesh boundary.

Nienhuys and van der Stappen presented methods for cutting in deformable objects using the finite element method [NvdS00,NvdS02]. Their goal was to enable interactive cutting in volumetric meshes while preserving the finite element mesh model. To avoid the generation of degeneracies, a node snapping technique was described as well.

Several other techniques like volume sculpting or constructive volume geometry are related to our approach. Sculpting is done by moving voxel-based tools within the model (e.g. [FCG00]). In constructive volume geometry, introduced by Chen and Tucker [CT00], a set of algebraic operations is applied to scalar fields. However, direct volume rendering techniques rather than more efficient polygonal rendering techniques are used for visualization.

3. Data Representation and Visualization

For the proposed partitioning tools, we developed a two-level data model consisting of a *tetrahedral mesh* and a *binary tree*.

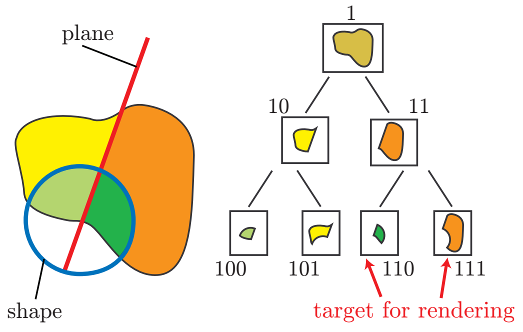
We call a tetrahedral mesh *consistent* if its set of tetrahedra is a 3-dimensional simplicial complex [Ede01]. The properties of simplicial complexes enables us to define regions (simplicial sub-complexes) and to consistently extract boundaries for visualization (the mesh boundary is inherently available).

In addition to the tetrahedral mesh, a binary tree is used for storing the actual partitioning results (regions). This section gives a basic overview of the binary tree representation, interfaces (region boundaries), and finally interface extraction and visualization.

Per definition, each partitioning operation splits the underlying mesh into two new, disjoint parts. Therefore, a binary tree can be used to store the relation of tetrahedra to certain regions. Each leaf node in the tree represents a region and stores the indices of associated tetrahedra (see Fig. 2). Initially, the root of the binary tree stores all tetrahedron indices of the mesh as one region. If partitioning operations are applied, additional tree levels are created by splitting each leaf node into two child nodes. Thus the path from the root to a given leaf node can be encoded into an index that represents a sequence of partitioning operations (similar to a location code). A binary representation, implemented as a bit vector, is used for encoding: each bit corresponds to either a left or right link in the tree.

To visualize a set of regions $\mathcal{S}$ of a tetrahedral mesh $\mathcal{M}$, an *interface* can be defined which represents the boundary to the remaining regions $\mathcal{M} \setminus \mathcal{S}$. By using the properties of simplicial complexes, each interface consists of a set of tetrahedron faces which can be always represented by a properly connected triangular mesh. These triangles are inherently available in the tetrahedral mesh: no additional geometry is generated, because the geometry of the parent model is used. This avoids duplicate vertices or faces and guarantees an overall consistent connection between the volumetric model and its visualized surface mesh.

The basic idea of the extraction algorithm is to traverse the binary tree for identifying the list of regions which are target for rendering (see Fig. 2). A rendering bit vector r specifies all regions which should be visualized. For each index i of a traversed node a bitwise XOR compare with r is performed (trimming i and r to equal length). If $r \otimes i = 0$, then the child nodes are rendering candidates and recursively processed. Finally, all visited leaf nodes and their lists of associated tetrahedra are examined for retrieving the boundary faces. A tetrahedron face is extracted if the neighboring tetrahedron does not belong to a region which should be rendered (defined by r).



(a) liver model (b) e.g. render bit vector $r = 11$

Figure 2: *A plane and shape partitioning operation is applied to the tetrahedral mesh in Figure (a). Figure (b) shows the corresponding binary tree. The root node consists of one region. After plane partitioning, two child nodes are created, partitioning the domain into two pieces. If applying another operation, 4 new leaf nodes, storing the regions, are created. Interfaces can be extracted to visualize arbitrary regions.*

Since vertices of the mesh are shared by adjacent mesh primitives, a concept is required which allows the assignment of attributes to vertices. For each tetrahedron, belonging to a certain region, a unique set of attributes must be available (e.g. to render each region in a different color). Therefore, the *wedge concept* proposed by Hoppe [Hop98] is used. A wedge acts as a wrapper for vertices. It stores a vertex index and additional local parameters but is associated with exactly one tetrahedron. This has the effect of allocating multiple wedges each of which is indexing a similar vertex but storing different attributes.

4. Partitioning Algorithms

Global modification of tetrahedral meshes are computationally expensive and acceptable computation times require the design of efficient algorithms. We introduce a hierarchy of three partitioning algorithms which differ in the complexity of the partitioning shape and consequently in computation times. The first algorithm uses an analytically defined plane and is typically applied in order to perform coarse partitioning operations. Due to its simple shape, several algorithm parts can be specialized and optimized (e.g. subdivision). The second algorithm uses a generalized analytical shape. Contrary, the partitioning shape of last algorithm is specified by a triangular surface mesh for most flexible partitioning. In general, a partitioning algorithm consists of four essential parts:

1. **Collision detection:** tetrahedra which are intersected by the partitioning tool are detected.
2. **Intersection point calculations:** after testing if an intersection point is not yet calculated, it is stored in a hash table.
3. **Subdivision**: intersected tetrahedra must be split according to the intersection points for a unique region assignment.
4. **Assignment to regions and binary tree update:** all tetrahedra can now be assigned to one of the newly created regions. A new level is added to the binary tree and tetrahedron indices are stored in the corresponding leaf nodes.

According to these four steps, the following partitioning algorithms were developed.

4.1. Analytical Plane Partitioning

The plane partitioning algorithm starts with a traversal of the tetrahedral mesh and determines for each tetrahedron whether it is intersected by the plane. Tetrahedra which are not split, are immediately assigned to a region (corresponding to the side of the plane). A tetrahedron is intersected by the plane if not all of its vertices lie on the same side of the plane. This verification is done by calculating the signed distance between all tetrahedron vertices and the plane. The distance is computed by using robust orientation tests to avoid floating point inconsistency [She97]. If calculated distances have different signs, then the corresponding edges contain intersection points and the tetrahedron is target for subdivision. According to the signed distances of the vertices, a lookup table is used to determine the appropriate subdivision case (eight cases, one possible intersection point per edge).

Before subdivision, the intersection points are calculated using basic vector algebra. In order to keep the mesh consistent, shared intersection points of adjacent tetrahedra must be stored uniquely in the vertex buffer. Therefore, a hash table is used for efficient propagation of intersection points to adjacent tetrahedra. The index of each newly calculated intersection point is stored in the hash table. The corresponding tetrahedron edge uniquely defines the hash table's key value.

In case of plane partitioning only one intersection point per tetrahedron edge can exist. Therefore, an ordinary subdivision technique can be applied. We chose a subdivision method by Dompierre et al. [DLVC99] who splits quadrilateral faces by choosing the diagonal which contains the smallest vertex index in the face. This method is efficient, does not depend on floating point calculations, and guarantees that tetrahedra faces are always split uniquely.

Before adding newly created tetrahedra to the mesh, invalid orientations are detected and fixed. In addition, the binary tree is updated by adding the tetrahedron index to the corresponding node in tree (representing the correct region).

4.2. Analytical Shape Partitioning

Shape partitioning provides a more flexible way than plane partitioning, because more general shapes can be used to specify partitions. The method requires shapes, which can be defined by a formula but leads to an efficient algorithm. Since a partitioning operation must completely split each tetrahedron (required by the binary tree concept), partially split tetrahedra are not considered. In these cases the shape partitioning methods are approximative, but subsequently generate less tetrahedra and vertices. This has a positive effect on the mesh complexity. Since the partitioning shape can be specified analytically, the algorithm is efficient even for non-planar shapes.

An analytically defined partitioning shape $\mathcal{P}$ has to provide two functions. An intersection function $I(e_{\mathbf{t}})$ that calculates the intersection points between $\mathcal{P}$ and a tetrahedron edge $e_{\mathbf{t}}$, and a distance function $D(p_{\mathbf{t}})$ that determines the location of a point p_t relative to the partitioning shape. As an example, the distance function of a sphere is defined as $D(p) = (p_x - m_x)^2 + (p_y - m_y)^2 + (p_z - m_z)^2 - r^2$, where center point m and radius r specify the partitioning sphere. $I(e_{\mathbf{t}})$ is then an ordinary sphere – line segment intersection test.

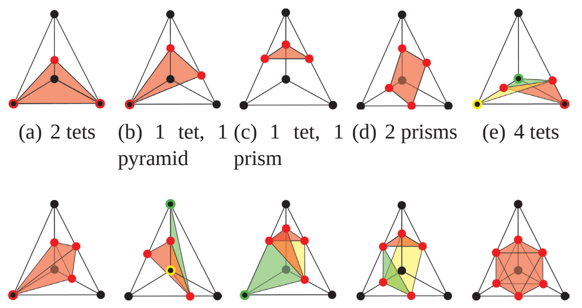
At first the distance function is calculated for each vertex p of the mesh by traversing the vertex buffer. Afterwards the mesh tetrahedra are traversed and the result of the $D(p)$ is stored in the corresponding wedges of each tetrahedron (see Section 3). Three states represent this result: *inside* ($D(p) < 0$), *intersection* ($D(p) = 0$), and *outside* ($D(p) > 0$). Since partially split tetrahedra are ignored, intersected tetrahedra can be identified corresponding to the sign of D: if $D(p_i) > 0$ and $D(p_j) < 0$ for tetrahedron vertices p_i and p_j, then this tetrahedron is target for subdivision. Otherwise it can be assigned immediately to the inside or outside region (according to the sign of the distance function).

Subsequent to the identification of an intersected tetrahedron $\mathbf{t}$, the exact intersection points are calculated. Similar to Section 4.1, a hash table stores the intersection information. In order to uniquely calculate the intersection points between $\mathbf{t}$ and $\mathcal{P}$, the hash table is examined for intersection points: for each edge of the current tetrahedron, existing intersection points are retrieved. If no entry for an edge is found, all its intersection points are calculated (and stored in the hash table) using the intersection function.

Intersecting tetrahedra with non-planer shapes can result in multiple intersection points per tetrahedron edge (e.g. a small sphere can intersect a tetrahedron edge twice). The iterative subdivision (described in the next paragraph) subdivides such tetrahedra according to the order of stored intersection points. Moreover, the used hash table approach must be extended. An ordinary hash table with the ability to store objects with equivalent key values is organized in buckets (usually arranged in a linked-list). These buckets are extended to so-called *sorted buckets* by storing all intersection points of a tetrahedron edge in a specific order. In our case, all intersection points of a tetrahedron edge are sorted based on the distance to the edge's start-point. The order represents the correct cut sequence and is required later in the subdivision process.

After all intersection points are calculated, each intersected tetrahedron $\mathbf{t}$ must be subdivided for a unique assignment to the corresponding region. Basically, 10 symmetrical different subdivision cases are required to split a tetrahedron into two parts (see Fig. 3). However, the subdivision of a tetrahedron that contains edges with multiple intersection points is a non-trivial task. We introduce an iterative approach: In the first iteration, the currently active intersection point of each tetrahedron edge is determined by choosing the first stored point in the corresponding sorted buckets of the hash table (see Fig. 4(a)). Subsequently, a lookup table is used to determine how to rotate or flip the vertices to fit the default orientation in order to perform the basic subdivision by Dompierre [DLVC99]. Active intersection points of the next iteration are retrieved by incrementing the position in the sorted buckets of the edges (Fig. 4(b)). Finally, the algorithm is iteratively applied to subdivided tetrahedra until all intersection points are processed (shown in Fig. 4(c)).

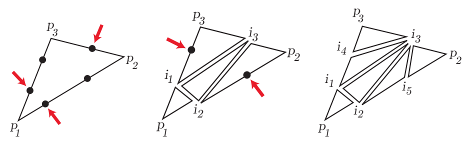
In the last step of the partitioning algorithm each subdivided tetrahedron is assigned to the corresponding region. It contains only vertices of the original tetrahedron or intersection points (labeled with *intersection* state). Thus the vertex states, which are stored in the wedge data structures,



(a) 2 tets (b) 1 tet, 1 pyramid (c) 1 tet, 1 prism (d) 2 prisms (e) 4 tets

(f) 4 tets (g) 1 tet, 2 pyramids (h) 2 tets, 2 pyramids (i) 2 tets, 1 pyramid, prism (j) 4 tets, 2 1 pyramids

Figure 3: *Partitioning cases of a tetrahedron and an arbitrary partitioning shape with subdivision into primitives. Intersection points are colored red; red-black points denote tetrahedron vertices lying on the shape. Red colored faces denote configurations which are required definitely. Either yellow or green colored faces are required in 4 cases to guarantee, that the tetrahedron is split completely. Yellow-black or green-black points must lie on the partitioning shape respectively.*



(a) initial state (b) 1^{st} iteration (c) 2^{nd} iteration

Figure 4: *Iterative tetrahedra subdivision with multiple intersection points per edge (simplified shown as a triangle subdivision). In each iteration an ordinary subdivision is performed, using active intersection points (red arrows).*

are then used to assign those tetrahedra to the correct region. If a tetrahedron does not contain a vertex with state *outside*, it is assigned to the inside region. Respectively, if a tetrahedron does not contain an *inside* vertex, it is assigned to the outside region. Tetrahedra, where no unique assignment is possible, are assigned to the outside.

4.3. Geometrical Shape Partitioning

More flexible partitioning can be achieved by using a triangulated surface mesh to define the partitioning shape. In order to divide the tetrahedral mesh into disjoint regions, the partitioning shape must be a closed mesh or an object homeomorphic to an open disc.

All tetrahedra, which are intersected by partitioning shape faces, must be identified. In addition, remaining tetrahedra have to be assigned to the inside or outside region as well. Both steps can be handled efficiently by using an octree, generated from the partitioning shape. Since the shape is usually modified interactively, the octree generation must be done on-the-fly.

For each vertex of the tetrahedral mesh, it is determined if a point lies inside or outside of the partitioning shape. Tetrahedron vertices, lying on a partitioning shape face, are identified later during exact intersection point calculations. A ray shooting approach, which can handle open surfaces, is used:

- A ray is shot, starting from each vertex of the volumetric mesh. The ray direction is specified by the barycenter of a certain partitioning shape face (e.g. primitive with index 0). This guarantees that the ray always intersects the partitioning shape.
- The intersected partitioning shape face with the minimal distance to the ray origin must be identified. Starting at the root of the octree, child nodes, which are hit by the ray, are traversed recursively in the order of the smallest distance to the ray origin. Once a leaf node with stored primitives is visited, an exact ray – triangle intersection test is performed for each stored primitive [MT97]. Subsequently, the algorithm terminates and returns the primitive index with the minimum distance.
- If the supporting plane's normal vector of the chosen primitive points in the same direction as the ray (computed in step one), then the tested vertex is located inside the partitioning shape.

In the next step, the tetrahedra of the mesh are traversed. According to the results of the ray shooting, the *inside* or *outside* state is set in the wedges of each tetrahedron. If all four vertices have the same state, the tetrahedron is immediately assigned to the corresponding region. Otherwise, it is intersected by the partitioning shape and the exact intersection points must be calculated. Each tetrahedron is tested against the octree (bounding box tests) to retrieve the list of overlapping partitioning shape faces. For each of these shape faces, the exact intersection points between the tetrahedron and the supporting plane of the shape face are calculated (similar to Section 4.1). Subsequently, it must be tested whether a calculated intersection point lies inside the partitioning shape face by projecting the point and the triangle into 2D: a robust 2D orientation test [She97] finally determines if the intersection point is located in the triangle and can be stored.

Analogue to analytical shape partitioning, the hash table approach is used for the propagation of intersection points. Since partitioning shape faces are tested in arbitrary order, the verification whether an intersection point is already calculated, is more complex. The following information must be stored in the hash table for each intersection point:

- The index of the intersection point p_k.
- The index of the partitioning shape face $\mathbf{f}$ which intersects the tetrahedron edge in p_k.
- The vertex indices of the edge of $\mathbf{f}$ if p_k lies on this edge.

- The vertex index of a point of **f** if p_k coincides with this point.

To ensure mesh consistency and avoid degeneracy, each intersection point must be inserted in the hash table only once. Three possible cases can arise:

1. If an intersection point p_k lies inside a single partitioning shape face **f**, then p_k is only shared by adjacent tetrahedra faces. A comparison of the identifier of **f** with stored hash table values is sufficient to verify that p_k was already calculated.
2. If p_k lies on an edge shared by two partitioning shape faces, then additionally the vertex indices of the affected partitioning shape edge must be compared with corresponding hash table values.
3. Otherwise, if p_k coincides with a vertex of **f**, then this vertex index is compared with stored hash table values as well.

Finally, the actual subdivision and the assignment of subdivided tetrahedra is performed according to Section 4.2.

5. Results

We evaluated the algorithms in 45 test cases for performance and quality measurements. In this test environment, high-quality tetrahedral meshes with sizes ranging between 10,000 and 280,000 tetrahedra were used. In case of geometrical shape partitioning, partitioning shapes were specified by $800 - 2,000$ triangular primitives.

The performance measurements were done on a Pentium 4 with 2 GHz. Experiments showed that in a typical partitioning operation $1,000 - 5,000$ tetrahedra were intersected by the partitioning tool. The graph of Figure 5 gives a runtime comparison of all algorithms. On average, a complete partitioning operation, applied to a $50,000$ tetrahedra mesh, took 75 ms, 320 ms, and 685 ms for plane, analytical shape, and geometrical plane partitioning. We carried out particular runtime measurements for collision detection, intersection point calculation, subdivision, and region assignment.

Plane partitioning can be performed very efficiently. Collision detection and region assignment was performed in 7 ms per 10,000 tetrahedra. In the remaining time, intersection points are consistently calculated and the subdivision is performed. For 1,000 tetrahedra, these operations together took 28 ms on average.

Analytical shape partitioning takes longer. Collision detection, intersection point calculations, and region assignment only depend on the complexity of the distance and intersection function. In case of a sphere, those steps took about 17 ms for 10,000 tetrahedra. In addition, iterative subdivision, which is more complex and thus has more overhead, contributed with 100 ms per 1,000 tetrahedra.

The additional runtime increase if using geometrical shapes is primarily caused by the collision detection (octree) and region assignment (ray shooting). We used an on-the-fly generated octree with depth 5 ($100 - 220$ ms generation time in experiments). The described ray shooting approach took about 40 ms for 1,000 mesh vertices. Finally, the consistent propagation of calculated intersection points is more runtime expensive since triangular faces are processed instead of evaluating analytical functions.

We also analyzed the quality of all tetrahedra, created after a partitioning operation, since degenerated tetrahedra are a known problem of subdivision. In our experiments, $4 - 7.1\%$ of subdivided tetrahedra were degenerated according to a quality ratio proposed by Nienhuys [NvdS02]. Node snapping was applied on-the-fly during subdivision [NvdS02], reducing the number of degeneracies by 52% on average.

Peer-to-peer comparisons to state-of-the-art techniques are difficult since most of them aim at local real-time cutting where small-sized tetrahedral meshes are used ($< 5,000$). Most related to *global partitioning* is a technique of Ganovelli et al. [GCMS01] where planar cut shapes are used to cut multi-resolution meshes with 37,000 tetrahedra. The runtime, given in their paper, consists of tetrahedra subdivision ($60 - 530$ ms) and data structure updates ($150 - 344$ ms). Ganovelli achieves a total runtime of $219 - 875$ ms (Pentium 2, 400 MHz).

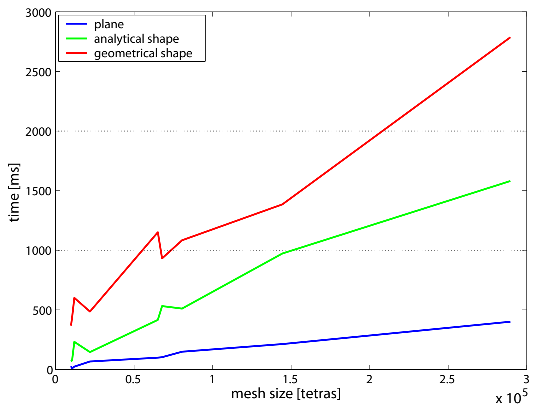


Figure 5: *Average runtime of the proposed partitioning algorithms in relation to the initial mesh size (P4, 2 GHz).*

5.1. Application

Several 3D applications require the ability to define and alter arbitrary regions of a volumetric mesh. In our case, the partitioning algorithms are integrated into a virtual liver surgery planning system [Rei05]. This system provides the possibility for surgeons to simulate atypical (non-segment oriented) tumor resections.

Figure 7 shows a medical scenario where partitioning is used to plan a surgical intervention. The partitioning concept is helpful in supporting physicians, who need to carefully consider a limited number of alternatives for the most promising outcome of an intervention. In this example, two tumors are target for resection (a). Inside the semitransparent liver the vessel structure can be seen. After investigation, the surgeon has several different possibilities for removing the diseased parts. In strategy 1, the dataset is partitioned using a plane in order to uncover the tumors (b). The left part of the liver is hidden after the partitioning operation (c). Subsequently, two options must be considered. In strategy 1a ((e) to (h)), the surgeon plans to remove (hide) each tumor separately. Therefore, two spheres are positioned for applying an analytically defined shape partitioning operation. In contrast, both tumors are removed at once with a single bigger sphere in strategy 1b ((i) and (j)). Another strategy simulates the tumor resection by removing a single tissue part: a geometrically defined deformable grid is applied in strategy 2 ((k) and (l)).

Since the hierarchically organized data structure of Section 3 is used, the different intervention strategies can be stored in a single data model. Moreover, it also enables to switch between those strategies (or combine them) in real-time, e.g. for measuring the volume of removed tissue.

Figure 6 shows two different scenarios of the virtual liver surgery planning where the presented partitioning tools are interactively used in a virtual reality setup. By using a tracked pencil and panel, partitioning tools can be specified and partitioning can be initiated intuitively.

An evaluation with a surgeon has shown that the introduced partitioning tools can be used successfully within a complete planning process (evaluations are detailed in [Rei05]). Surgeons can save time by using the presented atypical planning methods compared to traditional intervention planning. In addition, the performance of the partitioning algorithms was stated reasonable, since they are initiated in a separate thread without disturbing interactive rendering.

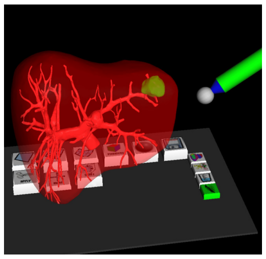
6. Conclusion and Future Work

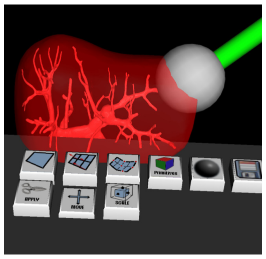
We have presented an entire concept for implementing partitioning tools into interactive visualization or planning environments. Based on the required complexity of the tool, one of three algorithms can be chosen. Evaluations have shown that the provided algorithms are feasible for surgical planning in interactive systems. The presented methods are not limited to surgical environments, thus partitioning has a broad field of application in volumetric visualization.

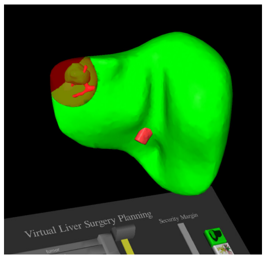
Nevertheless, several improvements are reasonable and addressed for future work. The algorithms can be further optimized, however, experiments indicate that the runtime is sufficient for interactive planning. In addition, situations may arise where a unique assignment of tetrahedra to the correct region is not possible. In this case, the methods are currently approximative. We intend to overcome this problem by performing on-the-fly mesh refinement.

Acknowledgments

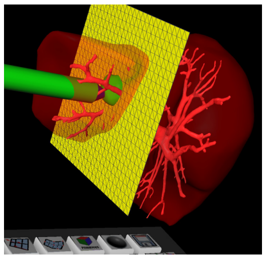
This work was funded by the Austrian Science Fund *FWF* under contract no. P17066.

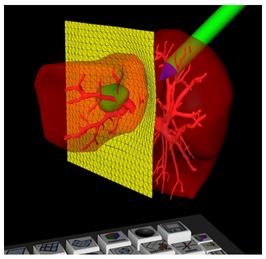

(a) The right part of the liver includes a tumor. An analytical shape partitioning is initialized, choosing a sphere primitive.


(b) The sphere is interactively scaled and positioned to cover the tumor completely.



(c) After partitioning, each region can be modified separately while the information is stored in one model. In this case, the left region is set to transparent.


(d) Another strategy is chosen where geometrical shape partitioning, using a deformable grid, is initialized.


(e) The grid is interactively deformed in such a way that the tumor is completely contained in the left liver part.

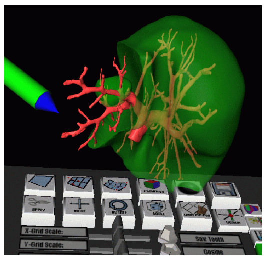

(f) Two regions are created after the operation. The left one is hidden in order to verify that the tumor was successfully resected.

Figure 6: *Partitioning methods embedded into a virtual reality-based liver surgery planning setup.*

References

[BGTG04] BIELSER D., GLARDON P., TESCHNER M., GROSS M.: A state machine for real-time cutting of tetrahedral meshes. *Graphical Models 66* (2004), 398–417.

[CT00] CHEN M., TUCKER J. V.: Constructive volume geometry. *Computer Graphics Forum 19*, 4 (2000), 281–293.

[DLVC99] DOMPIERRE J., LABBE P., VALLET M.-G., CAMARERO R.: How to subdivide pyramids, prisms, and hexahedra into tetrahedra. In *8th International Meshing Roundtable* (1999), pp. 195–204.

[Ede01] EDELSBRUNNER H.: *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, 2001.

[FCG00] FERLEY E., CANI M.-P., GASCUEL J.-D.: Practical volumetric sculpting. *Visual Computer 16*, 8 (2000), 469–480.

[GCMS01] GANOVELLI F., CIGNONI P., MONTANI C., SCOPIGNO R.: Enabling cuts on multiresolution representation. *Visual Computer 17*, 5 (2001), 274–286.

[Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Journal of Computers & Graphics 22*, 1 (1998), 27–36.

[KVPL04] KONRAD-VERSE O., PREIM B., LITTMANN A.: Virtual resection with a deformable cutting plane. In *Simulation und Visualisierung 2004* (2004), pp. 203–214.

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1 (1997), 21–28.

[NvdS00] NIENHUYS H.-W., VAN DER STAPPEN A.: Combining finite element deformation with cutting for surgery simulations. In *Proc. of Eurographics 2000* (2000), pp. 274–277.

[NvdS02] NIENHUYS H.-W., VAN DER STAPPEN A.: *Supporting cuts and finite element deformation in interactive surgery simulations*. Tech. Rep. UU-CS-2001-16, Univ. Utrecht, 2002.

[Rei05] REITINGER B.: *Virtual Liver Surgery Planning: Simulation of Resections using Virtual Reality Techniques*. PhD thesis, Graz University of Technology, 2005.

[She97] SHEWCHUK J. R.: Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry 18*, 3 (october 1997), 305–363.

[SHGS06] STEINEMANN D., HARDERS M., GROSS M., SZEKELY G.: Hybrid cutting of deformable solids. In *IEEE Virtual Reality 2006* (2006), pp. 35–42.

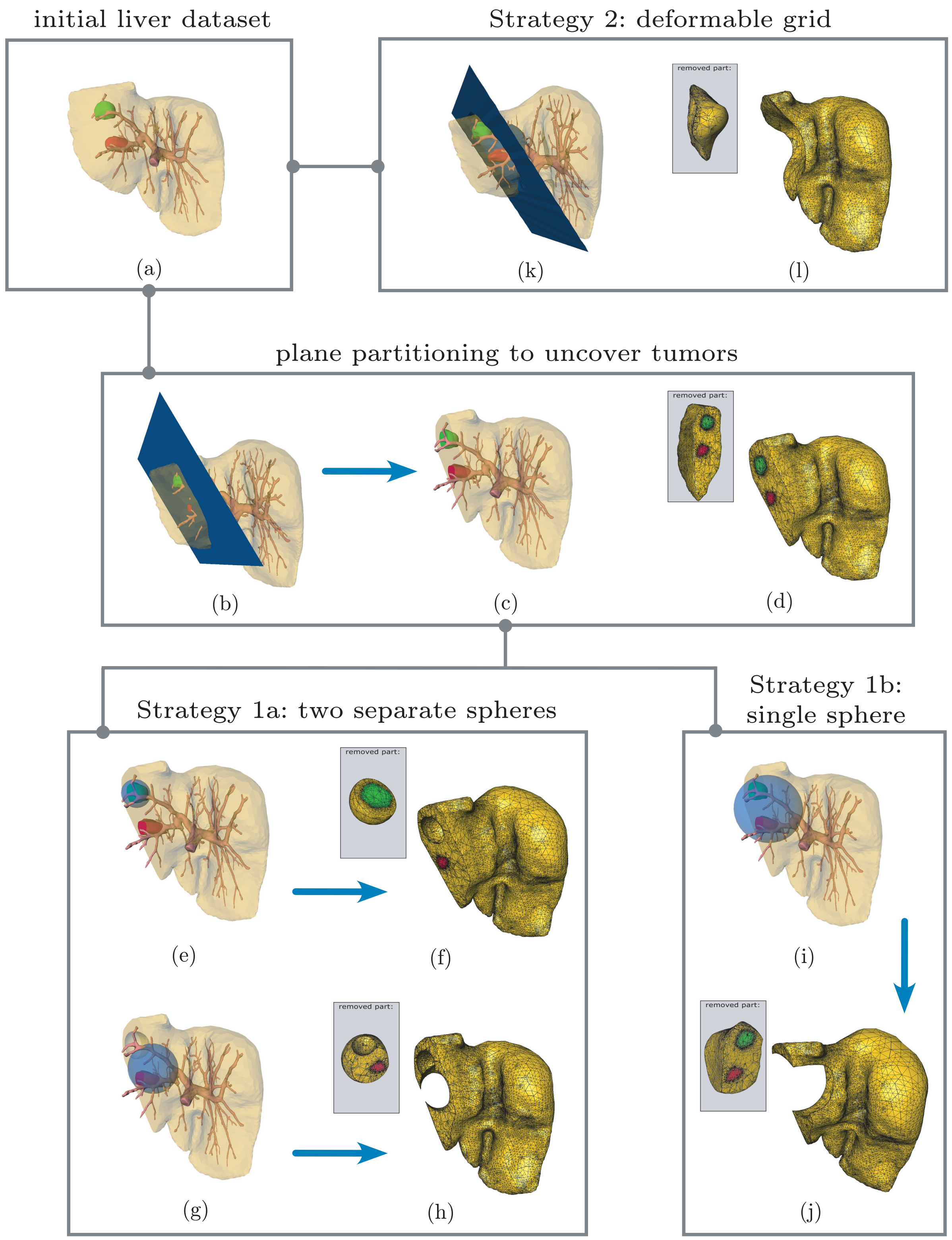

initial liver dataset
(a)
Strategy 2: deformable grid
removed part:
(k)
(l)
plane partitioning to uncover tumors
removed part:
(b)
(c)
(d)
Strategy 1a: two separate spheres
removed part:
(e)
(f)
removed part:
(g)
(h)
Strategy 1b: single sphere
(i)
removed part:
(j)

Figure 7: *Medical scenario showing different possibilities for the removal of liver tumors. Either plane partitioning and analytical shape partitioning (strategy 1, 1a, 1b), or geometrical shape partitioning (strategy 2) are used two simulate different intervention strategies.*