European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS 2004
P. Neittaanmäki, T. Rossi, K. Majava, and O. Pironneau (eds.)
O. Nevanlinna and R. Rannacher (assoc. eds.)
Jyväskylä, 24–28 July 2004

IMAGE-BASED UNSTRUCTURED 3D MESH GENERATION FOR MEDICAL APPLICATIONS

Guntram Berti*

* C&C Research Laboratories, NEC Europe Ltd.,
Rathausallee 10, 53757 St. Augustin, Germany,
e-mail: berti@ccrl-nece.de

Key words: mesh generation, medical images, octrees, marching tetrahedra

Abstract. *We present a dimension-independent algorithm for non-uniform meshing of voxel geometries, e.g. stemming from segmented medical images.*

The basic algorithms uses spacetrees (generalized octrees for arbitrary dimension) to build a hierarchical representation of the image information. The resolution of the spacetree can depend on several parameters, such as homogeneity of the material, or application dependent criteria. Arbitrary Cartesian refinement rules can be used, in addition to the usual isotropic binary refinement. This is particularly important for images with anisotropic voxel sizes. Also, the balancing level can be freely chosen.

In a second stage, a non-uniform simplicial or hybrid mesh is generated with guaranteed element quality. Again, this part is completely dimension independent.

In order to achieve smooth material boundaries, we introduce the marching simplices algorithm for 2D and 3D. Marching simplices generalize the Cartesian volumetric marching tetrahedra method to general simplicial meshes. Our algorithm also handles cases where the separating surface passes through mesh vertices. This avoids degeneracies and allows one to shift vertices to the surface where appropriate, thus significantly reducing the number of cut simplices.

The implementation of the algorithmic components makes use of generic programming to maximize reuse, flexibility and composability. We strive for maximal independence of algorithmic components, thus paving the way toward a meshing toolbox which allows to build mesh generators adapted to particular applications. For example, in the context of maxillo-facial surgery simulation special mesh generation options can be applied to the vincinity of cuts.

Mesh generation for large models can require large computational resources. The algorithms described here lend themselves to parallelization, which is planned for the near future. Meshing of large geometries is a good candidate for a Grid service, which is currently under development.

*This work was supported by the European Union under research grants IST-1999-10378 and IST-2001-37153

1 INTRODUCTION

In mesh generation literature, it is common to assume an exact, unambiguous definition of the geometry which has to be represented or approximated with a mesh. This sounds like a reasonable precondition for input to mesh generation algorithms. However, in many fields, such exact geometry definitions are not a priori available. For example, in a medical context, images of different dimensions (2D, 3D, 4D) are the predominant source of geometric information, for example generated by computed tomography (CT), magnetic resonance imaging (MRI), cryosection, or ultrasound imaging. In general, these images do not contain an explicit and unambiguous description of the underlying geometry, and are thus not directly accessible for meshing. Instead, they contain continuous (albeit quantized) gray-scale or color values, typically describing the average of some physical quantity over small sampling volumes ("voxels").

These images are themselves only a (often noisy) approximation to the real geometry. But things get worse. The process of transferring the continuous information into discrete geometry information is called *segmentation:* Each voxel of the images gets assigned a unique material. Segmentation is a difficult task which in general still withstands complete automatization, and falls outside the scope of this paper, however relevant it is for our goal. Segmentation, on the one hand, adds some information by attributing material labels to geometric locations. On the other hand, the segmented image alone contains less geometric information than the original image, because a (more or less) continuous variable has been replaced by a discrete one. Therefore, the exact geometric locations given by such a segmented image have to be taken *cum grano salis*, they are only valid within some tolerance. Thus, a good meshing approach for the kind of geometry represented by segmented images should be open for any type of additional information which could help to make the results more accurate, and may contain other volumetric information like anisotropy direction which can be exploited.

Nevertheless, segmented images provide a good starting point for mesh generation. We will identify such images with cell-labeled Cartesian grids. On the level of segmented "images", it is no longer important where the image comes from, nor whether it is an "image" at all: The only important property is its finite number of regions (corresponding to "materials") defined as sets of cells (corresponding to "voxels") of a Cartesian grid.

Formulating algorithms on top of the Cartesian grid view relieves us from the details of image processing and opens a number of additional sources of geometric information, for instance isoranges of analytical functions or "voxelized" B-rep geometries [19]. The question where the boundary of the geometry "really" is will re-occur in Section 3 when we want to improve the approximation of the boundary.

The Cartesian structure of the geometry representation lends itself to exploitation. In the case of non-uniform mesh generation, octrees (or quadtrees in 2D) immediately impose themselves as appropriate intermediate data structures controlling mesh resolution. A closer look reveals that most issues related to a hierarchical Cartesian space partitioning can be formulated in a completely dimension independent way. Therefore, we prefer the term (Cartesian) *spacetree* [1] instead of octree/quadtree; we will try to give dimension-independent presentations whenever possible. In section 6 we will discuss issues related to dimension-independent implementation.

Our meshing approach consists of several stages, which we try to keep as independent as

possible in order to achieve maximal flexibility and extensibility. As a matter of fact, many applications e.g. in computational medicine require *ad-hoc* mesh manipulation, e.g. adding specific boundary conditions, cutting meshes in surgery simulation, or dealing with multi-physics phenomena requiring hybrid meshing approaches. These needs are best met by creating specialized mesh generators using a general meshing toolbox. In this paper, we describe some of the basic components of such a toolbox, and also briefly discuss a programming approach striving for truly reusable implementations of such meshing algorithms.

The first step in the meshing process is the creation of a spacetree representation from the Cartesian geometry (Sec. 2.1). After optional spacetree filtering such as balancing, it can be tessellated into various types of meshes — tetrahedral, hexahedral with hanging nodes, or hybrid, combining hexahedra, pyramids and tetrahedra (Sec. 2.3). The result is a non-uniform mesh with axis-parallel boundaries. Now, keeping in mind that the Cartesian geometry is just an approximation of a "real" geometry with presumably smooth boundaries, we can try to approximate these smooth (but unknown) boundaries somewhat better. In general, the only thing we know about these "real" boundaries is that they separate the voxel (centers) in the same way as the Cartesian approximation (assuming the segmentation is correct, which is a questionable assumption in practice). Therefore, the question naturally arises how to approximate these boundaries. The "best" answer depends on the specific situation, and on the amount of additional information available. A rather general assumption is that the interface between two materials can be modeled locally as the iso-surface of a continuous or even smooth *separating function*. For this case, we present in Sec. 3 a volumetric variant of the marching tetrahedra algorithm, which we call *marching simplices* in order to underline dimension independence of the basic idea — the dimension dependent parts are detailed for 2D and 3D. In a first attempt, we explore trilinear interpolation of the binary voxel values as separating function. This choice results in some vertices having exactly the iso-value 0. In order to deal with this situation, we extend the case analysis with patterns where the zero-level goes trough some of the tetrahedral vertices (Sec. 3.2). These cases are also handy in the general case of an arbitrary function because they allow to move vertices to the iso-surface instead of cutting (Sec. 3.3). The non-uniform tessellation algorithm and — in the case of a d-linear interpolation of a binary classification as separating function — also the marching simplices algorithm guarantee minimum cell quality (Sec. 4). Preliminary studies show that in the general case, the combined cutting and shifting approach also yields good element quality. In Sec. 5 we present some examples from medical applications. Issues related to implementing a meshing toolbox with maximal reusability and flexibility using generic programming and a generalized *pipe & filter* architecture are discussed in Sec. 6. Finally, we outline further plans for continuing this work and discuss possible future research directions.

1.1 Related work

Quadtrees and octrees have been used for mesh generation since about two decades [36, 25]. Buratynski [8] uses an octree-based approach similar to ours to mesh 3D CAD models with internal boundaries, cutting the octree and the geometric model *before* triangulation, and using more complicated case analysis afterwards. Schroeder and Shephard [23] use octrees and subsequent local Delaunay mesh generation for CAD data. In order to avoid degenerate cases and ensure consistency across octree cells, they employ a modified version of Watsons original

algorithm [34] with a sophisticated ordering of point insertion, suitable for balanced octrees. It is however not clear how to generalize the ordering to unbalanced octrees or different refinement patterns. Bern et al. [2] use quadtrees for guaranteed quality mesh generation of point clouds in 2D. Mitchell and Vavasis [17] use octrees for guaranteed-quality triangulation of 3D polyhedra with holes. In our opinion, the use of polyhedra as the representation of the geometry leads to a more complex case analysis than is really needed in the context of medical image data, as it does not exploit the approximate nature of the internal and external boundaries in this context.

In the context of mesh generation from medical images, Müller and Rüegsegger [18] use a uniform Cartesian grid in order to generate meshes of high-resolution CT images. They develop a volumetric marching cubes algorithm with appropriate extension of the case analysis of the original algorithm [16]. Nielson and Sung [20] use a regularly triangulated Cartesian grid, obtaining a volumetric marching tetrahedra algorithm to generate FEM meshes from medical image data. Both do not consider the zero-vertex case. Hartmann and Kruggel [13] use octrees to build a hierarchical representation of the images first. Simple octree cells (e.g., those with 8 nodes) are subdivided into tetrahedra, while complex cells (those with more than 8 nodes) are tesselated using a modified Delaunay algorithm on this cell. However, details are missing how a consistent tesselation across facets is achieved.

Another popular method is to first generate meshes of the material interfaces, and then use some sort of constrained (Delaunay or advancing front) meshing of the resulting polyhedral model [9, 29]. This has the drawback of being more time-consuming in practice, and imposes the generated surface mesh as an artifical, unnecessary constraint on the volume mesh generation, which moreover is limited to tetrahedra. Also, the link to the original volumetric representation of the geometry (image) is lost, which might contain additional information, such as material properties or anisotropy directions [35]. As the machinery needed for generating the surface mesh is very similar to that one presented here for volume meshes (which can easily be restricted to surface mesh generation and thus in principle also be used as preprocessing step for the surface-based volume mesh generation), the question arises why not generate volume meshes directly, and use the more complex approaches only to resolve special situations, e.g. triangulating a transition region for hybrid meshing. An advantage of the surface-based approach can be the superior quality of elements near the boundary. In our experience, using a combination of vertex snapping and smoothing also yields good quality elements near the boundary, whereas the quality of interior elements is always very good.

2 THE NON-UNIFORM MESHING ALGORITHM

The basic non-uniform meshing algorithm works in two main steps: First, a hierarchical representation of the geometry (i.e. image) is built, resulting in a spacetree, which determines mesh resolution. The leafs of this spacetree may already be regarded as a non-uniform mesh, having non-conforming d-cubes as cells (or alternatively, irregular cells having more vertices then a d-cube), see Fig. 1 (right). In a second step, this spacetree can be tessellated into a conforming mesh, resulting in either a triangulation, or a hybrid, hex-dominant mesh.

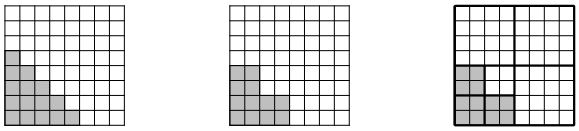


Figure 1: Spacetree generation in 2D. Left: Original image, middle: image subsampled to 2×2 voxels, which corresponds to finest resolution of the spacetree, right: generated spacetree. We can regard the leafs of this spacetree as a mesh in several ways: Either a mesh with axis-parallel quads as cells and two hanging nodes (non-conforming leaf grid), or as a conforming mesh containing two cells with five nodes (conforming leaf grid).

2.1 Building a hierarchical geometry representation (spacetree)

The geometry is assumed to be described by a d-dimensional segmented image, or equivalently, by a Cartesian grid $\mathcal{I}$ together with a characteristic function χ defined on the cells of $\mathcal{I}$ (which correspond to image voxels). χ maps each cell uniquely to a material type, i.e. a value in a small discrete set of labels, thus defining a partition of the set of Cartesian cells. Mesh generation will in general restricted to specific materials, forming a body of interest (BOI).

This base grid $\mathcal{I}$ is transformed into a grid hierarchy or spacetree $\mathcal{S} = (\mathcal{I}_0, \ldots, \mathcal{I}_k)$, which may or may not contain $\mathcal{I}$, depending on the resolution wanted — in Fig. 1, $\mathcal{I}$ is *not* contained in the spacetree. Initially, the spacetree consists of a grid $\mathcal{I}_0$ which must be a suitable coarsening of $\mathcal{I}$ with respect to some refinement pattern (cf. Fig. 2) — usually, one uses a 2^d pattern, such that a refinement step (split) bisects a cell in each coordinate direction, resulting in 2^d children. Unlike a classical octree, $\mathcal{I}_0$ can otherwise be arbitrary — in particular, it does not need to be a single cell.

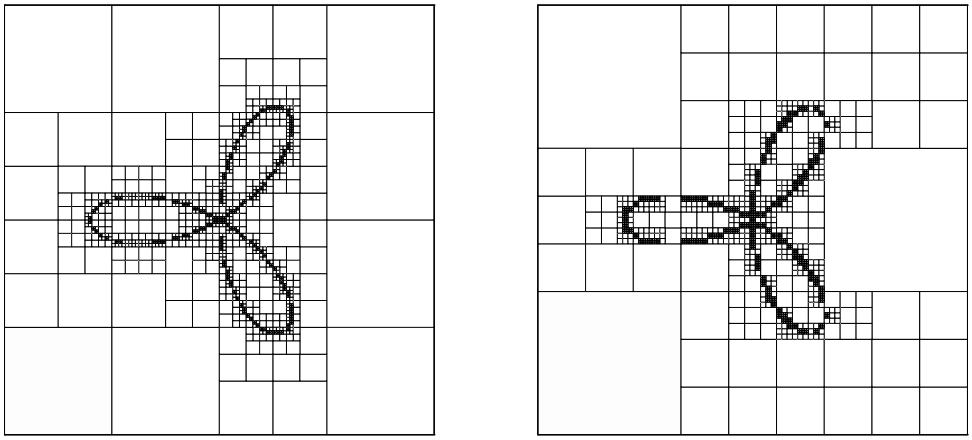


Figure 2: Left: Spacetree balanced to two levels maximal difference, using a 2x2 refinement pattern. Right: Unbalanced spacetree, 3x3 refinement pattern. In both cases, a binarized analytical function was used as a "virtual" image.

Now we look at each cell of the currently finest level of $\mathcal{S}$, and see whether it is too large and has to be split. The decision can depend e.g. on geometric location, material or material boundaries contained within the cell, and other, application specific criteria. The splitting may lead to cells which are even finer than the original cells of $\mathcal{I}$, if one uses a special geometric criterion for refining a particular region.

2.2 Balancing the spacetree

If needed, the spacetree may be balanced by prescribing a maximal difference in the size of two adjacent spacetree cells. Typical is a difference of 1. Higher values allow for smaller spacetrees, but result in worse element shapes. In our implementation, the balance level can be set to any value (cf. Fig. 2), and future versions will support a much finer *local* control of balancing, depending on material and geometric location.

2.3 Tessellating the spacetree cells

Although the spacetree itself constitutes a valid mesh which can be used e.g. for speeding up visualization or even for finite element simulations (see [1] for an approach to handling hanging nodes), one often wants a conforming mesh.

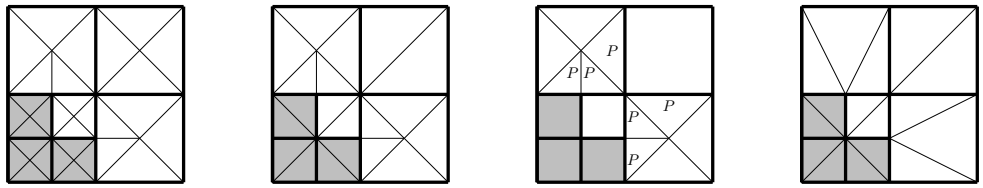


Figure 3: Spacetree tessellation in 2D. (a): Pure midpoint insertion algorithm. (b): Template variant. The additional nodes on the boundary of the left upper and right lower cell trigger the insertion of midpoints in these cells. For the cells in the lower left and upper right, either template subdivisions are used for a simplicial mesh (b), or they are left as is for a hybrid mesh (c). Cells marked with P are pyramids. (d): Extended template variant: Also some non-regular cells are meshed by a template.

Therefore, the d-cubes which are spacetree leaf cells have to be tessellated in a consistent way, that is, tessellations must match across spacetree facets. This is simple in the case of a uniform spacetree, i.e. all cells have the same size. In this case, we can either output the cubes themselves, or adopt a simple scheme for subdividing the spacetree cells into simplices (see Figure 4).

In the case of a non-uniform spacetree, the situation is more complicated. Some possibilities are:

1. Keep only the spacetree vertices and perform a global Delaunay triangulation, eliminating simplices outside the BOI afterwards

2. Apply a Delaunay triangulation to each cube and its vertices separately, trying to ensure matching edges across cube faces afterwards

3. Perform a direct triangulation of each cube which guarantees conforming meshes

We believe that the first two approaches using global or local Delaunay triangulations have serious problems: The first approach guarantees a conforming mesh, but some work has to be invested to remove superfluous simplices outside the domain, and to make sure that the boundary matches. Also, there is the risk of producing very flat cells called *slivers*, which in the concrete case at hand may even be degenerate, given the highly regular structure of the

input. These problems make a robust implementation more difficult. The second approach has the additional problem of having to assure conformity across cube facets, which is not trivial [23]. Prescribing the triangulation on the facets before triangulating the cubes would be possible, but it is not clear whether the resulting polytopes are triangulable without Steiner points.

Because of these problems, the third alternative has been pursued. We developed a direct triangulation algorithm similar to that described by Bern et al. for point clouds [3] with the following useful properties:

- It runs completely locally, that is, it considers only the nodes of the current spacetree cell, and is therefore suited for parallelization, and easy to restrict to arbitrary regions
- It guarantees consistency across spacetree cell boundaries
- It works for any number of maximal level difference of the spacetree, in particular for unbalanced trees
- It guarantees a minimum quality of the resulting tetrahedra, the concrete bounds depending on the balancing level of the spacetree
- It is purely combinatorial, therefore independent of possible geometric transformations and absolutely stable
- It is very efficient (linear in the number of nodes)
- It can easily be extended to create hybrid (cube-dominant) meshes
- It works for any dimension

The main idea of the tessellation algorithm is recursive: If we have a (simplicial) subdivision of the boundary facets of a spacetree cell, we get a (simplicial) subdivision of the spacetree cell itself by connecting its midpoint with the boundary simplices (or more generally, the boundary $(d-1)$-cells), see Figure 3 for the 2D case (upper left and lower right cell). For the boundary facets themselves, we can apply the same idea recursively. This tessellation is evidently consistent across facets, because the subdivision of a facet depends only on the data of the facet, namely, its boundary nodes.

A hybrid subdivision which tries to retain as many d-cubes as possible while producing a conforming mesh is obtained, if each k-cube is tesselated if and only if at least one of its $(k-1)$-facets has been subdivided. In this case, the midpoint of the k-cube is added, and it is connected to all boundary $(k-1)$-facets. This approach creates d different element types in d dimensions: squares and triangles in 2D, cubes, pyramids and tetrahedra in 3D, and in general elements combinatorially equivalent to the polytopes

$$P_k = \operatorname{conv}\left(\{0\} \cup \Big\{ \sum_{j=1}^{k} \epsilon_j e_j \mid \epsilon_j \in \{\pm 1\}\Big\} \cup \bigcup_{j=k+1}^{d} \{e_j\}\right) \quad 1 \le k \le d \tag{1}$$

with the unit vectors $e_i \in \mathbb{R}^d$. Here, P_d is equivalent to the d-cube, and P_1 is equivalent to the d-simplex.

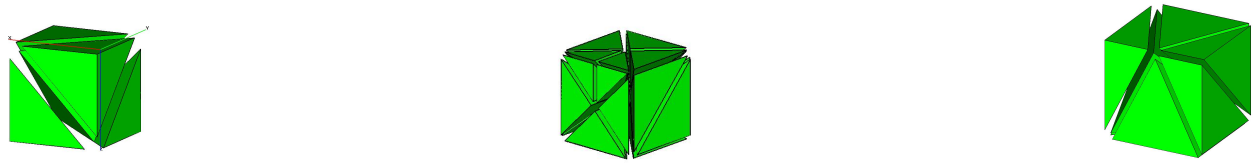


Figure 4: Left:Template subdivision of a cube (regular spacetree cell) into 5 tetrahedra. Middle: Midpoint subdivision of a non-regular spacetree cell. The left facet is subdivided according to the midpoint rule, whereas the right and top facets' subdivisions use a template. Right: Extended template, needing no additional node.

2.4 Reducing the number of inserted midpoints

For simplicial tessellations, it would be a waste to insert midpoints in every case. Therefore, the rule can be relaxed as follows: If a spacetree cell is regular (i.e. there are no additional nodes on its boundary), then use a template subdivision without new midpoint node, else use the midpoint rule. This relaxed rule is applied recursively to the boundary of cells: A facet of a cell is only subdivided by the midpoint rule if there are additional nodes on its boundary. In 3D this means that a single additional node on an edge first triggers midpoint insertion of the (not yet subdivided) incident facets and then that of the incident cells, see Figs. 4 middle and 5.

The template subdivision of both spacetree cells and facets can (at least in 2D and 3D) be chosen in such a way that a consistent tessellation on both sides of an spacetree facet is guaranteed. It is a generalization of the subdivision of a 3D Cartesian grid, where each cell is subdivided into 5 tetrahedra in an alternating, checkerboard way. This subdivision of the Cartesian cells also defines a subdivision of the Cartesian facets, in a way that depends only on the Cartesian (integer) coordinates of the facet. As each level of the spacetree is a Cartesian grid, we can apply the subdivision rule implied by this Cartesian grid to the spacetree facets of the appropriate level. Thus, each spacetree facet has a uniquely defined triangulation, which is consistent with the template subdivision of incident spacetree cells. This is important if cells with template and with midpoint subdivision are adjacent, see Figs. 4 and 5 right (the rear lower cell is subdivided by a template).

We can drive the template approach further by introducing additional templates for frequently occuring patterns with few additional nodes, see Figs. 4 right and 3 (d). However, the additional logic tends to get more and more complicated, whereas the additional benefit tends to zero.

The template approach can probably be extended to higher dimensions, but this has not been investigated by the author. In any case, it is possible to reuse already developed lower-dimensional templates for cube boundaries. If we always use midpoints to tesselate d-cubes, we may stop spacetree refinement at one level coarser than necessary, because the finer level is effectively created via midpoint insertion anyhow.

3 THE GENERALIZED VOLUMETRIC MARCHING SIMPLICES ALGORITHM

The basic algorithm for non-uniform mesh generation creates grids with 'staircase' boundaries, parallel to coordinate planes. While — strictly speaking — the segmented input voxel image does not provide more geometric information, the underlying geometry typically has

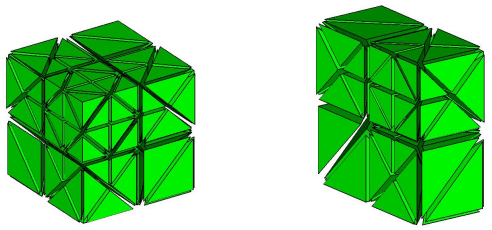


Figure 5: Non-uniform subdivisions in 3D. On the right, the rear half of the mesh is shown. The vertex on the middle of the upper middle spacetree edge triggers insertion of midpoints in the incident spacetree facets and cells, e.g. the rear upper spacetree cell. Here, 15 additional nodes are inserted. If the extended template approach for facets is used, only 6 additional nodes (cell centers) have to be inserted; if in addition the 3D template of Fig. 4 right is used, only 3 nodes have to be added. (Images have been generated by GMV [21], following a cell shrink filter [6].)

smooth boundaries. In order to achieve a better representation of such smooth boundaries, a generalized variant of the volumetric marching tetrahedra algorithm [20] has been designed and implemented, working for arbitrary triangulations.

The basic idea of this algorithm is to cut tetrahedra at the "smooth" interface between two materials. The core algorithm can cope with arbitrary triangulations, in particular those arising from the non-uniform tessellation algorithm. Thus, it can be used to post-process any given triangulation in 2D or 3D, for example to cut out a given geometry. Also, the algorithm allows cuts to go through existing mesh vertices, which is very useful for avoiding small features and coping with "degenerate" situations (which now are not degenerate any more).

We have restricted ourselves to the case of exactly 2 materials (regions) M_1 and M_2 which have to be separated. In this case, the boundary between the two materials can be modeled as the zero-surface of a continuous function f. For the multi-material case, the approach has to be extended, for instance by using a vector-valued probabilities for affiliation of vertices to regions, along the lines of [7] or [14, 28, 27]. This is work in progress.

Several problems have to be solved, in order to implement the simple idea of cutting simplices:

1. Starting from the binary cell classification, we have to define a smooth separating function f, with $f(x) < 0 \Leftrightarrow x \in M_1$, $f(x) > 0 \Leftrightarrow x \in M_2$, and $f(x) = 0 \Leftrightarrow x \in \overline{M}_1 \cap \overline{M}_2$.

2. Given the function f, simplices have to be cut where $f = 0$, and both halves of cut simplices have to be triangulated in a way consistent with their neighbors.

3. The cut surface must be in the interior of the mesh, that is, in the case of smoothing the outer mesh boundary, we have to create an additional outside layer of cells

4. A minimum quality of the resulting simplices should be guaranteed

Item 3 can be guaranteed in the spacetree framework by a simply tessellating an additional boundary layer of cells outside the BOI. For arbitrary triangulations, it is more difficult. One can try to extrude the mesh towards the outside, but care has to be taken not to create self-penetration of the mesh. In the following, we will address the remaining issues.

3.1 Constructing a separating function

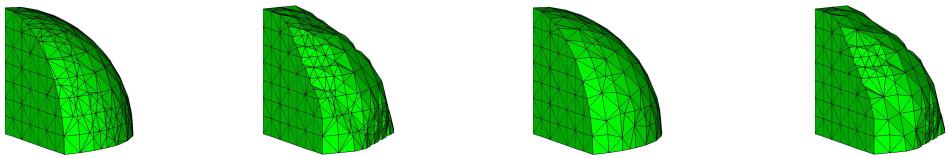
The choice of a suitable separating function f is not straightforward. If we stick to the "literal" information in the segmented image, we would have to define a function whose zero-level is exactly the interface $\overline{M}_1 \cap \overline{M}_2$, which would of course result in the same staircase mesh as before. Incidentally, in this case, our algorithm would indeed not cut a single cell, thanks to the handling of zero-vertices.

A simple way to get a separating function is to average cell values on their incident vertices by using d-linear interpolation. Defining $\tilde{f}$ on $\mathcal{I}_0$ on cell centers c by $\tilde{f}(c) = -1$ if $c \in M_1$ and $\tilde{f}(c) = +1$ if $c \in M_2$, we can use the dual grid with nodes at the cell centers to (d-linearly) interpolate these values on the vertices of $\mathcal{I}_0$, which include those of the triangulation, and hence have arrived at a continuous separating function.

The d-linear approach is certainly far from optimal for smooth data, and alternatives are investigated. Smoothing of the binary voxel data by using e.g. a Gauss filter may cause small structures to disappear or not have the desired effect [12].

Other choices might be to use higher-order interpolation, or to try to fit analytical functions to the data, for example minimizing some curvature measure. See Fig. 6 for the difference this may make. An analytical function provides us with a gradient field normal to the surface, which can be used e.g. to project vertices to the surface (see Section 3.3).

If the geometry comes from a continuous gray-scale image, we could also try the original voxel values as function values. Due to imaging inhomogenities, there is often no single global threshold value separating a material from the rest of the geometry. Thus, additional information on the *local* threshold must be obtained, for instance from the segmentation algorithms. If such information is available, it can be used to achieve sub-voxel accuray of the boundary location [31].



(a) Analytical (b) Trilinear (c) Analytical, snap 0.3 (d) Trilinear, snap 0.3

Figure 6: Analytic vs. trilinear separating function for meshing a 3D ball using volumetric marching simplices, with and without vertex snapping (see Sec. 3.3). Trilinear separating function based on binary voxel data does not result in satisfactory approximations for smooth underlying surfaces

3.2 Cutting the tetrahedra

Given a separating function f, we can classify each vertex as (a) positive (> 0, above threshold), (b) negative or (c) zero, and each edge as either (a) positive (all vertices ≥ 0), (b) negative, (c) cut (one vertex > 0, one < 0), or (d) zero (both vertices $= 0$). Only cut edges (case (c)) are split by inserting a *cut vertex*.

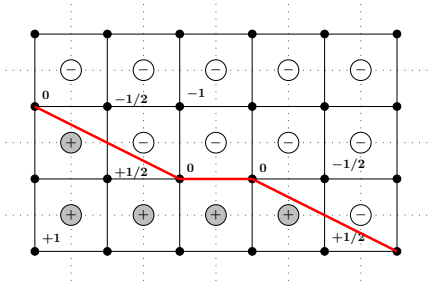
The case where vertices have a value exactly zero may seem to occur with probability zero. This is only true if the underlying function f is "generic". If f is derived via interpolation from

α	0.0	0.1	0.2	0.3	0.5
10x10x10 grid					
edges cut	303	192	108	45	0
total cells	6485	5960	5531	5210	5000
new cells	1485	960	531	210	0
surf. facets	532	388	264	186	135

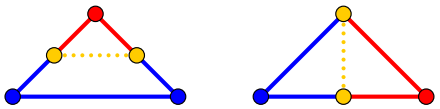
α	0.0	0.1	0.2	0.3	0.5
20x20x20 grid					
edges cut	1212	858	303	177	0
total cells	46369	44446	41479	40894	40000
new cells	6369	4446	1479	894	0
surf. facets	2278	1726	892	738	552

Table 1: Differences in additional elements for different vertex snapping parameters α (see Sec. 3.3), using an analytical separating function on two grid sizes (cf. Fig. 6 left column). $\alpha = 0$ corresponds to no shifting at all, and $\alpha = 0.5$ corresponds to shifting for each edge (and cutting no simplex). Shifting vertices to the boundary does reduce considerably the number of cells, and for values of α up to approx. 0.35 also significantly improves cell quality, see table 2.

a binary labeling of underlying Cartesian cells, these cases *will* occur (in fact, there will only be a small number of different possible vertex values). Thus, it is indispensable to consider this situation up-front. Handling these situation can also be used to avoid very small edges in the general case, by declaring a vertex 'zero' below a certain threshold, see Sec. 3.3.



(a) Using d-linear functions as separating function. Shown are cell values (+/−), interpolated vertex values and cut surface (red, thick line).Dotted lines are dual grid used for interpolation of cell values on vertices.



(b) The 2 essential cases for 2-dimensional marching simplices (blue positive, red negative, yellow zero vertex).

Figure 7: Marching simplices in 2D

Given the vertex classification, each simplex can be classified by a tuple (n_+, n_0, n_-), where n_+, n_0, n_- are the number of positive, zero, and negative vertices, respectively. In 2D, the situation is rather simple, leading to 2 essential cases needing subdivision (Fig. 7(b)). In 3D, we can distinguish 5 essential cases (counting symmetric ones only once), 4 of which need splitting (see also figure 8):

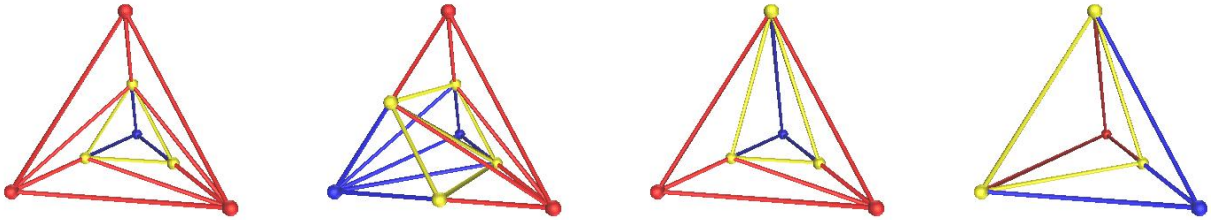
1. $n_+ = 0$ or $n_- = 0$:
   (a) $n_0 < 4$: This is the trivial case, the tet does not need splitting
   (b) $n_0 = 4$: Still we assume no splitting is needed, but we cannot deduce from the vertex values which material the tet has. In our case, we use the material label of the underlying spacetree cell. This situation is typical for a tet spanning a small "finger" of the material. However, if subsequent smoothing is planned (see Sec. 3.4), we better avoid such a situation by refining the simplex.

2. $(3,0,1),(1,0,3)$: The isosurface cuts off one corner of the tet, resulting in one tet and one prism.

3. $(2,0,2)$: The isosurface separates 2 opposite edges, one positive, one negative. The cut results in two prisms.

4. $(2,1,1),(1,1,2)$: The isosurface cuts off a tet through the two cut-vertices and the zero vertex, resulting in one tet and one pyramid.

5. $(1,2,1)$: The isosurface passes through the two zero vertices and the cut vertex, dividing the tet into two new tets.



case 2: $(1,0,3)/(3,0,1)$ case 3: $(2,0,2)$ case 4: $(1,1,2)/(2,1,1)$ case 5: $(1,2,1)$

Figure 8: Essential cases for vertex signs in 3D: Blue (dark) is positive, red (gray) is negative, yellow (light) is zero. The new triangulation edges are also shown, assuming the "smallest" vertex is always the right front one.

In the case of prisms or pyramids, which must be subdivided if we want a purely tetrahedral mesh, we have to ensure that the neighboring cell (which will also be a result of a subdivided tet) is subdivided in a compatible way. This is guaranteed if the the subdivision of facets is determined by the properties of the facet itself, without referring to the incident cells. As some subdivisions of a prism surface exclude tetrahedrization without Steiner points, we must find such a facet-based subdivision rule which still allows tetrahedrization of the incident cells. Fortunately, there is such a rule [20]: If we impose an (arbitrary) total order on the original vertices of the mesh, we choose in a quadrilateral face always that diagonal which is incident to the smallest vertex. The smallest vertex of the whole prism will thus have two diagonals, a configuration which always allows tetrahedrization. We can also use the smallest-vertex diagonal in the case of a pyramid, as either choice of the diagonal in the quadrilateral face allows tetrahedrization.

3.3 Avoiding cuts: Vertex snapping

If using an arbitrary function, intersections of edges may be arbitrarily close to an existing vertex, and thus yield very thin elements. In order to avoid this, we can declare an intersection to go through an existing vertex if any edge incident to it is intersected "too close" to the vertex. In this case, the vertex is moved to the cutting surface, and no incident edge is intersected any more, reducing the cutting of simplices incident to this vertex. We should however avoid to move all vertices of a simplex to the surface, as subsequent smoothing may lead to degeneration.

The exact meaning of "close" can be defined in an application-dependent way. A practical way is to take the ratio of edge length and the minimal distance of intersection point to an endpoint. If this is too small, the corresponding vertex should be moved to the cutting surface. Threshold values α of about 0.3 result in quite good triangulations, cf. Fig. 6. A value of 0.5 (together with some tie-break rule which vertex to move) avoids cutting at all, but may lead to ugly distortions. The rather simple threshold rule can be improved by deciding on a case-by-case basis whether cutting or shifting is more appropriate.

α	0.01	0.1	0.2	0.33	0.37	0.4	0.5
$n = 2$	1.97	1.97	1.97	1.97	1.66	1.66	1.66
3	8.31	3.73	2.67	2.67	2.67	3.40	8.05
5	14.13	5.03	2.89	1.96	1.96	2.74	4.20
10	8.71	7.66	4.01	2.63	2.32	13.96	11.21
20	21.62	8.42	4.06	2.84	2.59	29.69	1.6e6
40	37.89	9.02	5.08	3.51	52.97	79.55	1.1e5
60	54.54	10.71	5.13	3.60	81.30	83.75	2.0e6

Table 2: Worst measured element quality for different vertex snapping parameters for the example of Fig. 6 (a) & (c), using different grids sizes with n^3 cells. The quality measure is the condition number of the linear transformation to the equilateral tetrahedron [15], which thus gets quality 1.0. As an example, a tet spanned by the scaled unit vectors e_1, e_2 and ae_3 gets quality values 2.5 ($a = 0.25$) and 5.8 ($a = 0.1$). Values for α of approx. 1/3 yield good results.

3.4 Smoothing the mesh at boundaries

The surface vertices may be shifted in order get a smoother surface (assuming the underlying geometry is smooth). Laplacian smoothing is often used [29]: Iteratively move each surface towards the centroid of its adjacent surface vertices, controlled by a damping factor. However, Laplacian smoothing results in an overall shrinking of the geometry. Strategies which avoid or reduce shrinkage are found in [30, 33]. The SurfaceNets approach described in [10] combines Laplacian smoothing, projection onto an smooth iso-surface and constraining vertex relocation.

Other strategies are related to getting smoother separating surface, such as fitting a smooth analytical surface or using the gray values of the original image, and have been discussed in section 3.1.

4 MESH QUALITY

Some of the algorithms described before guarantee a minimum quality of the resulting tetrahedra.

The non-uniform meshing algorithm does generate only a finite number of different tetrahedra, the worst-case quality depending essentially on the balancing level of the spacetree, cf. figure 9.

If in the d-dimensional marching simplices algorithm, we choose the d-linear interpolation as separating function f, it can take only values in the finite set

$$\{i/2^{d-1} \mid -2^{d-1} \leq i \leq 2^{d-1}\} \tag{2}$$

Thus, in this case, the marching simplices algorithm does generate only a finite number of configurations, too. While a analytical investigation of the occurring cases has not been per-

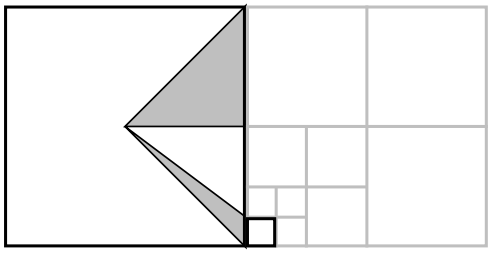


Figure 9: Worst cases in 2D for different spacetree balancing level L (gray triangles): $L = 1$ (top) and $L = 3$ (bottom). A balance level $L = 1$ results in perfect elements.

formed, preliminary numerical studies show that the quality of the resulting elements is mostly acceptable, but may also be enhanced by snapping.

Using arbitrary separating functions, we can no longer distinguish a finite number of cases. Yet, using a vertex snapping extension with a suitable threshold parameter α, e.g. $\alpha = 1/3$, seems to largely avoid bad elements in practice, see Table 2. More detailed analysis and experiments are required here, especially to see how the selection of the projection to the surface influences mesh quality, and whether a case-by-case choice between cutting and snapping further improves the situation.

Shifting the surface vertices in order to obtain smoother surfaces might be detrimental to volumetric element quality. Subsequent volumetric Laplacian smoothing in the vicinity (leaving the boundary unchanged) has proven to be useful.

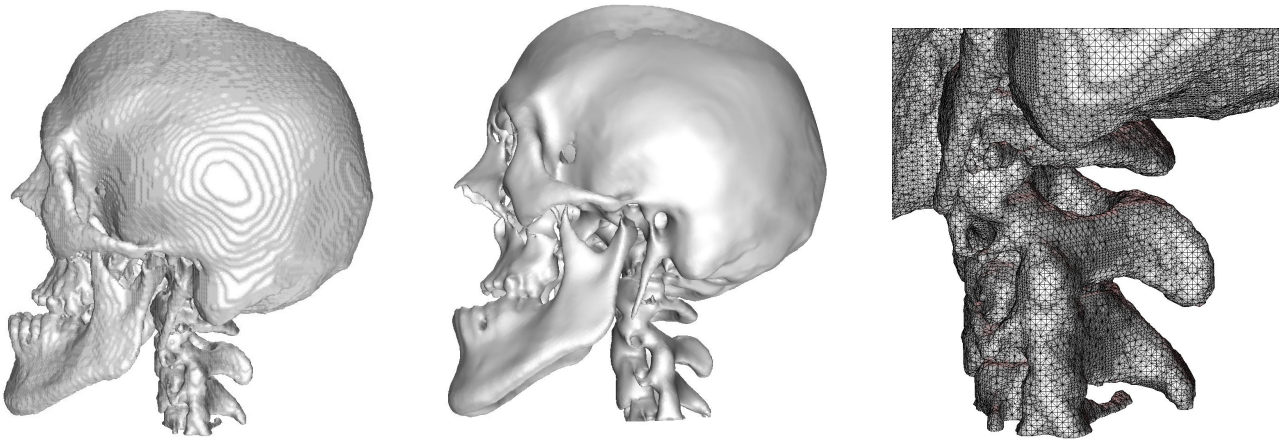


Figure 10: Left: Mesh of human skull - finest resolution (1mm voxel size). The mesh has 4M cells, and was generated using the volumetric marching tetrahedra algorithm with a trilinear separating function. Middle: Same, after 20 iterations of surface smoothing. Right: Zoom of the spinal region (no smoothing).

5 EXAMPLES

The examples in this section stem from maxillofacial surgery support, a task occurring within the EU IST projects SimBio [26] and GEMSS [11]. Maxillofacial surgery is trying to correct inborn deformations in the mid-face (cf. Fig. 10) by cutting some bones and pulling face parts into the "right" position. The ultimate aim is to be able to predict the outcome of this process in

a reliable way, in order to develop a treatment plan [22]. Therefore, an accurate representation of individual patients head geometry is crucial.

The data sets used can be quite large. A modern CT scanner produces images of patient head with a resolution of about $0.5 \times 0.5 \times 1.0\,\mathrm{mm}$, resulting in ca. 125 million voxels, a substantial part of which represent actual material. It is out of question to produce meshes at this global level of detail. However, for some regions and some applications, the finer the mesh, the better. Thus, useful mesh resolution is only limited by available hardware, and a fast and stable mesh generation process is crucial. In this respect, the approach we have presented here is very efficient. A mesh with about one million cells using the marching simplices algorithm is generated in well less than a minute on a standard PC. The limiting factor is rather main memory and IO. Therefore, we investigate parallel mesh generation and distributed mesh storage.

6 GENERIC IMPLEMENTATION OF A MESHING TOOLBOX

The old `vgrid` mesh generation software [32] used in the SimBio project [26] is currently being replaced. A new mesh generation toolkit is designed as a generic library based on the Grid Algorithms Library GrAL [5], which allows to develop reusable mesh based algorithms independently of the concrete mesh data structures. Its aim is to provide general-purpose meshing tools, which can be combined to solve application-specific mesh generation problems. Thus, we can enjoy the best of two seemingly mutally exclusive approaches: On the one hand very general algorithmic components with a broad range of application, and on the other hand, tailor-made tools which allow to exploit the complete available set of additional, problem specific information. More examples for this line of thought can be found in [4, 6].

The toolkit encourages development of mesh generators using a generalized *pipe & filter* architecture: Typically, we are dealing with a chain of different data representations, from raw images to segmented images to spacetrees to meshes (cf. Fig. 11). Some filters transform data from one representation to the next (e.g. spacetree tessellation transforms a spacetree into a mesh), while other filters act on a single representation, e.g. spacetree balancing, or marching simplices.

In contrast to the pure pipe & filter approach [24], we can access all earlier stages at any time; for instance, gray scale images can be used to improve the accuracy of the marching simplices algorithm. Vice versa, we may fulfill requirements of later stages, e.g. by letting the decision whether to refine a spacetree cell be driven by an estimate of how well a boundary may be approximated by the marching simplices algorithm on the current level.

By using the generic programming paradigm promoted by GrAL, we achieve maximal reusability — for instance, the marching simplices algorithm can be used for any (simplicial) grid. Also, we achieve dimension-independent code as far as possible: Spacetree data structures and algorithms are implemented completely dimension-independent; thus, spacetrees can be built e.g. for 4D images (time series of 3D images). Where dimension-dependent algorithmic parts exist (e.g. the case analysis of marching simplices), they are hidden behind specialization code which enables their transparent use from dimension-independent higher level code.

We make algorithms as flexible as possible by parameterizing their changeable parts. For example, spacetrees may be instantiated with Cartesian refinement patterns of any size or user-defined refinement criteria, or marching simplices may use arbitrary separating functions, or

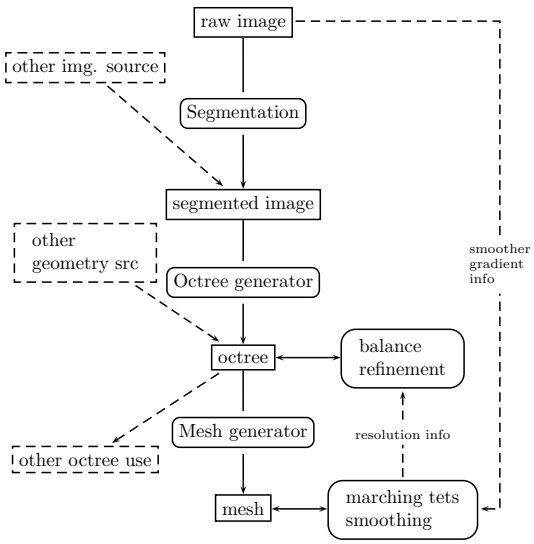


Figure 11: Outline of the generalized pipe & filter design of the meshing toolbox

switch between volume and surface element generation.

Genericity also makes restriction of algorithms to substructures transparent, thus rendering the implementation of hybrid algorithms practical: Use hexahedral meshes with hanging nodes in one place, smooth boundaries using the marching simplices at a particular material boundary, and a hybrid mesh to interface both. The free composability of algorithms can be used to implement problem-specific behavior in a specialized mesh generator, for example interactive support for cutting bones, which will operate on a partially constructed mesh, being able to access all intermediate structures like spacetrees, but without polluting the generic algorithmic components.

7 CONCLUSION AND OUTLOOK

We have presented the basis for a general and flexible mesh generation system for voxel based geometries, which are predominant in fields like medical simulation. We introduced a non-uniform, spacetree-based meshing algorithm for arbitrary dimensions with guaranteed element quality, which is capable of generating both simplicial and hybrid meshes. Also, a generalized volumetric marching simplices algorithm for arbitrary triangulations was presented providing extended case analysis including zero-vertex cases, which allows to avoid cutting in many cases. Thus, much less and better elements are created.

The algorithmic mesh generation toolbox outlined in Section 6 will be greatly enlarged in future work. We plan to fit analytical separating functions to the data in order to achieve better (smoother) approximations to material boundaries. We also plan to use the original images where available for achieving sub-voxel accuracy. The toolbox will allow to combine different algorithms at will, tailoring the best suited approach for each concrete situation.

The case analysis for the marching simplices algorithm will be extended to support more then 2 materials, which is crucial for multi-material simulations. More work is needed to investigate the effect of vertex snapping on the mesh quality. The choice between vertex shifting and simplex cutting could also be based on the achievable element quality. From our experience,

vertex shifting has quite some potential for reducing the number of surface cells in pure surface meshing, where it produces less and better elements.

As the generated meshes for real medical applications may contain several million cells or more, parallel mesh generation becomes a necessity. We plan to parallelize our meshing toolbox, exploiting the data-parallel nature of the basic algorithms.

Acknowledgements

I thank my current and former colleagues at NEC C&C Research Laboriatories, Jochen Fingberg, Jens Georg Schmidt and Ulrich Hartmann, for their ongoing support and constructive discussions.

The original version of the `vgrid` mesh generator was implemented by U. Hartmann and F. Kruggel at the Max-Planck-Institute of Cognitive Neuroscience at Leipzig, Germany, and continued by U. Hartmann at C&C Research Laboratories. It was subsequently extended by the author by adding the algorithms described in Secs. 2.3 and 3, and is available at [32].

A CT image used for the meshes displayed in Fig. 10 was kindly provided by Dr. Hierl, Department of Oral and Maxillofacial Surgery, University Clinic Leipzig, and was segmented using software from F. Kruggel, MPI CNS, Leipzig.

REFERENCES

[1] M. Bader, H.-J. Bungartz, A. Frank, and R. Mundani. Space tree structures for pde solution. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Proceedings of ICCS 2002, part 3*, volume 2331 of *LNCS*. Springer, 2002.

[2] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci.*, pages 231–241, 1990.

[3] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48:384–409, 1994.

[4] G. Berti. A generic toolbox for the grid craftsman. In W. Hackbusch and U. Langer, editors, *Proceedings of the 17th GAMM Seminar on Construction of Grid Generation Algorithms*. Online proceedings at `http://www.mis.mpg.de/conferences/gamm/2001/`, 2001.

[5] G. Berti. GrAL – the Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral`, 2001.

[6] G. Berti. Generic programming for mesh algorithms: Towards universally usable geometric components. In H. A. Mang, F. G. Rammerstorfer, and J. Eberhardsteiner, editors, *Proceedings of the Fifth World Congress on Computational Mechanics (WCCM V)*. IACM, Vienna University of Technology, July 7-12 2002.

[7] K. S. Bonnell, D. R. Schikore, K. I. Joy, M. Duchaineau, and B. Hamann. Constructing material interfaces from data sets with volume-fraction information. In *Proceedings Visualization 2000*, pages 367–372. IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[8] E. K. Buratynski. A fully automatic three-dimensional mesh generator for complex geometries. *Internat. J. Numer. Methods Eng.*, 30:931–952, 1990.

[9] J. Cebral and R. Löhner. From medical images to cfd meshes. In *Proceedings of the 8th International Meshing Roundtable*, pages 321–331, 1999.

[10] P. W. de Bruin, F. Vos, F. H. Post, S. F. F. Gibson, and A. M. Vossepoel. Improving mesh quality of extracted surfaces with surfacenets. In *MICCAI*, pages 804–813, 2000.

[11] The GEMSS project: Grid-enabled medical simulation services. `http://www.gemss.de`, 2002. EU IST project IST-2001-37153, 2002–2005.

[12] S. F. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 23–30. ACM Press, 1998.

[13] U. Hartmann and F. Kruggel. A fast algorithm for generating large tetrahedral 3d finite element meshes from magnetic resonance tomograms. In *Proceedings of IEEE Workshop on Biomedical Image Analysis*, pages 184–192, Santa Barbara, California, June 26–28 1998.

[14] H.-C. Hege, M. Seebaß, D. Stalling, and M. Zöckler. A generalized marching cubes algorithm based on non-binary classifications. Technical Report SC-97-05, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), 1997.

[15] P. M. Knupp. Matrix norms & the condition number: A general framework to improve mesh quality via node-movement. In *Proceedings of 8th International Meshing RoundTable*, pages 13–22, Lake Tahoe, 1999.

[16] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[17] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 212–221, 1992.

[18] R. Müller and P. Rüegsegger. Three-dimensional finite element modelling of non-invasively assessed trabecular bone structure. *Med. Eng. Phys*, 17(2):126–133, 1995.

[19] G. M. Nielson. Volume modelling. In M. Chen, A. Kaufman, and R. Yagel, editors, *Volume Graphics*, pages 29–50. Springer, 1999.

[20] G. M. Nielson and J. Sung. Interval volume tetrahedrization. In *IEEE Visualization '97 (VIS '97)*, pages 221–228, Washington - Brussels - Tokyo, Oct. 1997. IEEE.

[21] F. Ortega. General mesh viewer (GMV) homepage. `http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html`, 1996.

[22] J. G. Schmidt, G. Berti, J. Fingberg, G. Wollny, and J. Cao. A finite-element based tool chain for the planning and simulation of maxillo-facial surgery. In P. Neittaanmäki, T. Rossi, K. Majava, and O. Pironneau, editors, *European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS 2004*, to appear.

[23] W. J. Schroeder and M. S. Shephard. A combined octree Delaunay method for fully-automatic 3-D mesh generation. *Internat. J. Numer. Methods Eng.*, 29(1):37–55, 1990.

[24] M. Shaw and D. Garlan. *Software architecture: Perspectives on an emerging discipline.* Prentice Hall, 1996.

[25] M. S. Shephard, H. L. de Cougny, R. M. O'Bara, and M. W. Beall. *Handbook of grid generation*, chapter Automatic grid generation using spatially based trees. CRC Press, 1999.

[26] The Simbio Project. `http://www.simbio.de`, 2000. EU IST project IST-1999-10378, 2000–2003.

[27] D. Stalling, M. Zöckler, and H.-C. Hege. Segmentation of 3D medical images with subvoxel accuracy. In H. U. Lemke, K. Inamura, M. W. Vannier, and A. G. Farman, editors, *Proceedings of CAR'98. Computer Assisted Radiology and Surgery*, pages 137–142, June 1998.

[28] D. Stalling, M. Zöckler, O. Sander, and H.-C. Hege. Weighted labels for 3D image segmentation. Technical report, Konrad-Zuse-Zentrum für Informationstechnik (ZIB), Preprint SC 98-39, 1998.

[29] J. M. Sullivan Jr., Z. Wu, and A. Kulkarni. 3d volume mesh generation of human organs using surface geometries created from the visible human data set. In R. A. Banvard, editor, *The Third Visible Human Project Conference Proceedings*. National Institutes of Health, Oct.5–6 2000.

[30] G. Taubin. Geometric signal processing on polygonal meshes. In *Eurographics 2000 State of the Art Report*, 2000.

[31] U. Tiede, T. Schiemann, and K. H. Höhne. High quality rendering of attributed volume data. In D. Ebert, H. Hagen, and H. Rushmeier, editors, *IEEE Visualization '98*, pages 255–262. IEEE, 1998.

[32] The SimBio-Vgrid mesh generator homepage. `http://www.ccrl-nece.de/vgrid`, 2004.

[33] J. Vollmer, R. Mencl, and H. Müller. Improved Laplacian smoothing of noisy surface meshes. In *Proc. 20th Conf. Eur. Assoc. for Computer Graphics (EuroGraphics '99)*, volume 18. Computer Graphics Forum, 1999.

[34] D. F. Watson. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Comput. J.*, 24(2):167–172, 1981.

[35] C. H. Wolters, M. Kuhn, A. Anwander, and S. Reitzinger. A parallel algebraic multigrid solver for finite element method based source localization in the human brain. *Computing and Visualization in Science*, 5(3):165–177, 2002.

[36] M. A. Yerry and M. S. Shephard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, 3(1):39–46, Jan. — Feb. 1983.