Surface Models of Tube Trees

Petr Felkel
VRVis, Vienna, Austria
felkel@vrvis.at

Rainer Wegenkittl
VRVis, Vienna, Austria
wegenkittl@vrvis.at

Katja Bühler
VRVis, Vienna, Austria
buehler@vrvis.at

Abstract

This paper describes a new method for generating surfaces of branching tubular structures with given centerlines and radii. As the centerlines are not straight lines, the cross-sections are not parallel and well-known algorithms for surface tiling from parallel cross-sections cannot be used. Non-parallel cross-sections can be tiled by means of the maximal-disc interpolation method; special methods for branching-structures modeling by means of convolution surfaces produce excellent results, but these methods are more complex than our approach. The proposed method tiles non-parallel circular cross-sections and constructs a topologically-correct surface mesh. The method is not artifact-free, but it is fast and simple. The surface mesh serves as a data representation of a vessel tree suitable for real-time Virtual Reality operation planning and operation support within a medical application. Proposed method extracts a "classical" polygonal representation, which can be used in common surface-oriented graphic accelerators.

1. Introduction

This paper addresses the problem of generating a 2-manifold surface mesh of branching structures defined by central axis and radii. We apply this method in the preparation of surface models of liver vascular structures for Virtual Reality (VR) operation planning and Augmented Reality inter-operative support. The vessel tree is found in a pre-processing step by a vessel-tracking algorithm [11].

There are many algorithms for direct surface reconstruction from 3D datasets. The Marching Cubes approach [14] is useful for surfaces that can be defined by a threshold value. Deformable model approaches are applicable for more complex objects, performing the iterative reconstruction using balloons [13], simplex meshes [6], or level set methods [15], but with substantially higher time demands. We have chosen the surface reconstruction method from cross-sections, as it is simple, fast, and the cross-sections are already available.

Surface tiling from cross-sections has been intensively studied from the 1970's [12, 16]. There are well-known algorithms for reconstruction from *parallel cross-sections* of nearly any shape and any branching type [5, 17]. The input of the surface tiling algorithm is in our case formed by convex-shaped (circular) cross-sections, but these cross-sections are *not parallel*. They are distributed along a tree-shaped vessel centerline. If smooth transitions of the surface at joints are not desired, the tiling is simple via approximation by a zig-zag triangular pattern [20] or directly by generalized cylinders [10]. Both approaches are applicable if only one single tube segment is reconstructed [20], or if a surface of branching structures is constructed and a separate mesh created for each branching segment [10]. The ends of segments then intersect at the branching point.

An algorithm for constructing a surface from *non-parallel cross-sections* of an object was published by Treece et. al [19]. It uses a shape-based interpolation guided by maximal discs for real-time visualization of free-hand ultrasound. Previous work was done by Payne and Toga [18]. Two specialized algorithms for surface reconstruction of natural branching tree-like structures have been published by Bloomenthal. One used free form surfaces [3] for reconstruction of smooth transition surface in "ramiforms", another used convolution surfaces (form of implicit functions) for bulge-free blending in the branches [4]. The latter method generates the best surface approximation, but is rather complex.

We have developed a simpler recursive algorithm for surface construction of the branching tubular structures.

2. The mesh generator

The proposed algorithm uses generalized cylinders along the segments and constructs a transition surface at the joints. The algorithm solves n-furcations (n-times branching) and constructs a single 2-manifold mesh. The presented approach handles multiple branching in a unified way.

The algorithm uses a new recursive branching-construction procedure to construct an intermediate *base mesh* from the input vessel tree. This mesh serves as an in-

put to a well-known piecewise smooth subdivision surface generator based on the Catmull-Clark subdivision [7, 2], which produces a smooth vessel surface. We use the SUBDIVIDE 2.0 library for this purpose [2]. After subdivision of the base mesh we get a smooth surface, used for fast rendering in an augmented reality added surgery application [1].

2.1. Input data

The mesh generator presumes the input in the form of the *vessel-centerline tree*. The tree has the following structure (see Fig. 1 with an image of a simple branching vessel together with the tree data-structure): The tree *nodes* are located in the vessel start (root), in the branching points (joints) and in the end points (leaves).

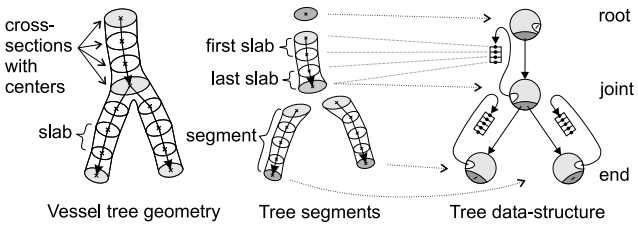

Figure 1. Vessel tree and its representation

Each node stores the incoming *segment* as a list of circular *cross-sections* with measured *radii*. The central vertices of the cross-sections lie on the *center-path* from the previous node to this node and the cross-sections are perpendicular to the center-path.

Center-paths of incident segments redundantly store the joint cross-section. The simplification of the vessel cross-section to a circular shape is sufficient in this case, as the aim is a 3D visualization of the general run of the vessels.

Each pair of subsequent cross-sections forms a segment *slab* (see Fig. 1). The branching tree-segments are represented by links to the successive (child) nodes, and the joint is formed as the unification of the child-slabs.

In a preprocessing step, the segment centerline vertices (delivered in sub-pixel precision by the segmentation algorithm [9, 11]) are down-sampled to a lower resolution, such that the centerline vertex distance is comparable to the vessel diameter in the neighborhood. This yields a smoother input centerline and lower density of the generated mesh at an acceptable precision level.

2.2. Derived data

The input tree representation by centerlines and radii is compact. As the tiling algorithm also uses a derived information repeatedly, it is pre-computed in a pre-processing step. This derived information includes (definitions and details follow in this Section):

- Directions $\vec{dir}_i$ of slabs and normals $\vec{n}_i$ of the cross-section planes along segment centerline,
- average normals $\vec{n}_0^{avg}$ of all cross-section planes n_0 in the joints,
- classification of the child segments into so-called *forward* and *backward* classes, and
- selection of the *straightest* forward child segment.

Computation of the slab directions $\vec{dir}_i$ and cross-section normals $\vec{n}_i$ is straightforward (see Section 2.2.1), for computation of the average normal $\vec{n}_0^{avg}$ in the joints (Section 2.2.2) and for subsequent classification of *forward* and *backward* segments, a heuristic is applied (Section 2.2.3). Various criteria can be applied for the *straightest* segment selection preferring the desired vessel feature (Section 2.2.4).

2.2.1. Directions and normals along a segment. *Slab* i is between the cross-sections with indices i and $i+1$, and the slab direction $\vec{dir}_i$ is equal to the direction of the cross-section i. The computation of the slab directions $\vec{dir}_i$ and cross-section plane normals $\vec{n}_i$ along the segments is performed separately for each segment (see Fig. 2, where the vectors along the centerline $P_0, P_1, \ldots, P_{n-1}$ of one segment are shown). Vectors $\vec{dir}_i$ and $\vec{n}_i$ are positioned in the appropriate point P_i, with index i along the segment.

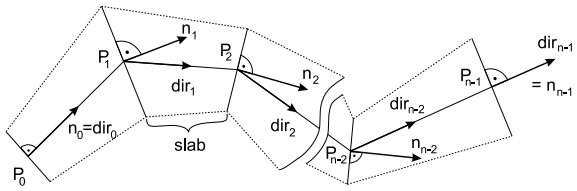

Figure 2. Directions and normals along the isolated segment

The directions $\vec{dir}_i$ of slab (P_i, P_{i+1}) are computed as normalized differences of the centerline points P_i (1).

$$\begin{aligned} \vec{dir}_i &= norm(P_{i+1} - P_i), \qquad i = 0, \ldots, n-2 \\ \vec{dir}_{n-1} &= \vec{dir}_{n-2}, \end{aligned} \tag{1}$$

where the function *norm*() represents a vector normalization, which scales a vector to unit length. It is defined as $norm(\vec{v}) = \frac{\vec{v}}{|\vec{v}|}$. The normals $\vec{n}_i$ of the cross-section planes along the tree segment are computed according to (2) (see also Fig. 2).

$$\begin{aligned} \vec{n}_0 &= \vec{dir}_0 \\ \vec{n}_i &= norm(\vec{dir}_{i-1} + \vec{dir}_i) \quad i = 1, \ldots, n-2 \\ \vec{n}_{n-1} &= \vec{dir}_{n-1} \end{aligned} \tag{2}$$

2.2.2. Joining of normals at joints. To enhance the smoothness of the generated surface at the joints, the common *average cross-section plane* is computed for all participating segments. The computed normal vector $\vec{n}_0^{avg}$ of the average plane replaces the stored normal in the last vertex of the incoming segment and the normals in the first vertices of the outgoing child segments.

In the following text, $\vec{n}^{in}$ represents the last normal of the *incoming segment*, i.e., $\vec{n}^{in} = \vec{n}_{n-1}$ of the incoming segment, where $\vec{n}_{n-1}$ is defined according to (2). Superscript index j represents the index of the child segment. The first cross-section planes of the followers take part in the n_0^{avg} computation. n_0^j represents the normal of the first cross-section plane of the j-th child segment.

The common normal vector $\vec{n}_0^{avg}$ is computed by means of the heuristic, as an "average" normal of the branching segment normals. To avoid unpredictable changes of the normal direction, only the "*positive*" normals are averaged. The decision positive/negative is based on the sign of the dot product with *incoming segment* normal $\vec{n}^{in}$. The normals whose included angle to the *incoming segment* normal $\vec{n}^{in}$ is less than 90° are classified as *positive*, the remaining as *negative*.

An example of the average normal $\vec{n}_0^{avg}$ computation can be found in Fig. 3, caption *i)*, where six branches are outgoing from the joint. The normals $\vec{n}_0^1$ and $\vec{n}_0^3$ are not used for the computation as they belong to the *negatives* (their included angle to the incoming segment normal $\vec{n}^{in}$ is greater than 90°). The normal $\vec{n}_0^5$ takes part in the $\vec{n}_0^{avg}$ computation, $\vec{n}_0^{avg}$ = average of $(\vec{n}^{in}, \vec{n}_0^2, \vec{n}_0^4, \vec{n}_0^0$, and $\vec{n}_0^5)$.

2.2.3. Forward and backward segments. The *average plane* defined by the average normal $\vec{n}_0^{avg}$ splits the outgoing vessels into two groups, called *backward* and *forward* (see an example in Fig. 3, caption *ii)*. They are handled separately, as will be discussed in steps 2a and 2b of the `TileTree()` procedure in Section 2.3.1. For an illustration of the role of forward and backward segments in the tiling process, see Figures 4 and 5.

In the example in Fig. 3, after the computation of the average normal $\vec{n}_0^{avg}$, the *positive* segment with normal $\vec{n}_0^5$ together with *negative* segments $(\vec{n}_0^1, \vec{n}_0^3)$ will belong to the segments directed *backward* in relation to the average plane. The segments with normals $\vec{n}_0^2, \vec{n}_0^4, \vec{n}_0^0$ will belong to the *forward* group.

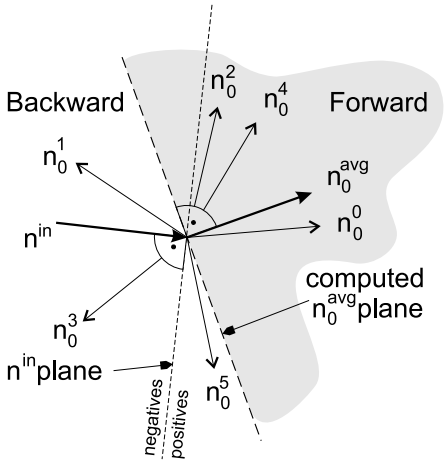

Figure 3. The normals $\vec{n}_0^j$ at starts of branching segments are classified twice: i) into *positives* and *negatives*, according to the angle to *incoming normal* $\vec{n}^{in}$ and ii) into *forward* and *backward* normals, according to the angle to the *average normal* $\vec{n}_0^{avg}$. The superscript index j represents the index of segment in the array of branching child-nodes, the zero subscript index the first layer of the segment

2.2.4. Straightest segment selection. The *straightest segment* is one of the *forward* segments. It has a privileged position, as the remaining *forward* segments are tiled to it (see Fig. 4).

The *straightest segment* is selected as a *forward* segment with the direction $\vec{dir}_0$ most similar to the direction of the average normal plane $n_0^{\vec{avg}}$. In Fig. 3, it is the segment with the superscript index 0 (its normal $\vec{n}_0^0 = \vec{dir}_0$, before being replaced by the average normal $n_0^{\vec{avg}}$).

If different vessel features are preferred for the algorithm other than strightness, other criteria for the straightest segment can be applied, such as the child vessel diameter or the vessel-sub-tree size (height or number of nodes).

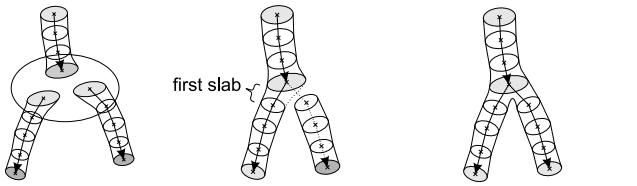

Figure 4. Tiling the *forward* segments

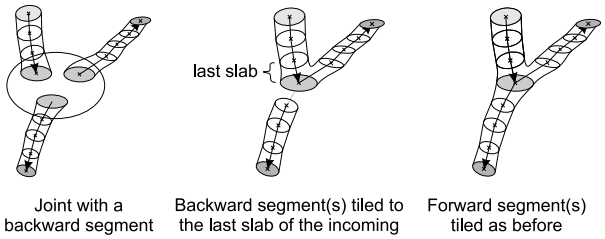

Figure 5. Tiling the *backward* segment

2.3 The tiling algorithm

As mentioned above, the algorithm first constructs a rough *base mesh* and then performs a Catmull-Clark subdivision scheme to achieve a surface of desired quality. The generated surface patches are quadrilaterals, as defined by the Catmull-Clark subdivision scheme [7].

The input circular cross-sections are approximated by described squares during the base mesh generation (see Fig. 6). Four vertices V_0, V_1, V_2, V_3 of each such square-shaped cross-section approximation are fully defined by equal quadrants, originated in the cross-section center, where the vertex V_0 is in the direction of the $\vec{up}$ vector. Propagation of the $\vec{up}$ vectors along the segments and in the tree joints is described in Section 2.3.4.

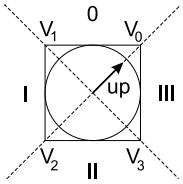

Figure 6. Circular cross-section and vertices inserted into the mesh. V_0 (in direction of the up-vector) defines the quadrants (0, I, II, III)

The *base mesh* generation is done recursively, respecting the recursive structure of the input vessel-tree data. The recursion is in the highest level handled by a `TileTree()`, and in the lower level by `TileJoint()` and `TileTrivially()` routines. `TileTree()` processes one tree segment and its followers, `TileJoint()` constructs the surface in the branching node, and `TileTrivially()` handles the straight, non-branching parts of the tree. These routines are described below.

2.3.1. Surface generation for one segment. The `TileTree(`S`)` controls the process of the tree surface generation starting in segment S. It is executed recursively for each segment of the tree, starting in the root node. It handles the straight tubular parts of the segment and the joining surface parts separately by executing `TileTrivially()` and `TileJoint()` routines, respectively.

For straight parts of the vessel tree (non-branching vessels), it executes the `TileTrivially()` routine for each slab, placing four surface quadrilateral patches into the base mesh per one execution. Patches for the joint surface are generated by executing the `TileJoint()` routine. Each execution processes one child node and places one quadrilateral surface patch into the base mesh.

While processing the joints, the outgoing segments are classified into *forward* and *backward* segments, according to their outgoing direction (for details see Section 2.2.3). Each execution of `TileTree(`S`)` consists of three tasks:

1. The *tubular part* of S is tiled from the second slab to the slab preceding the last one by means of a simple `TileTrivially()` procedure. It assumes, that the first slab was already tiled in the previous steps.
2. The *surface of the joint* generated by means of the `TileJoint()`.
   (a) First, the first slabs of the *backward* segments are recursively connected to the last slab of S.
   (b) Second, the *straightest forward segment* is selected and the first slabs of the remaining *forward* segments are recursively connected to its first slab.

   After this step, the surface mesh contains the first slabs of the child segments of S.
3. Finally, `TileTree()` is recursively executed for all *backward* and *forward* child segments of S.

2.3.2. Trivial tiling of one slab. The low-level surface tiling procedures `TileTrivially()` and `TileJoint()` work on the level of segment slabs, defined between pairs of subsequent vessel cross-sections (as described in Section 2.1). The `TileTrivially(`S`,` i`)` routine directly tiles one non-branching slab i of the segment S, and outputs four quadrilateral patches, one patch for each quadrant. The non-branching slab is the first slab of the root segment. The slabs between joints, where the segments are formed as single tubes, are also non-branching (Step 1 of `TileTree()`).

2.3.3. Tiling of the joint surface. The recursive procedure `TileJoint(`G`,` *dir*`,` *from*`)` is responsible for tiling the surface of the joint in one "depth level" around the selected segment slab. It forms the crucial step of the algorithm. The generated patches can be classified into two types: A *transition* patch between a pair of branches, and a surface *closing* patch. `TileJoint()` takes three parameters: the set of child segments G to be connected in this level, the direction dir of the processed slab, to which the set of segments G will be tiled, and the quadrant number *from*, from which the tiling was executed (for quadrant numbers see Fig. 6).

`TileJoint(`G, *dir*, *from*`)` handles these two tasks:

1. The segments in G are classified into subsets Q_k according to quadrants k. Quadrants are defined by the $\vec{up}$ vector in the processed cross-section.
2. The subsets Q_k are processed in the quadrants, skipping the quadrant in direction towards the caller ($k = \{0, I, II, III\} \setminus \{from\}$):
   (a) If the set of segments Q_k waiting for tiling is empty, the *closing patch* is created and the procedure returns.
   (b) A segment $C \in Q_k$ closest to the currently tiled segment is found by means of a maximal dot product $|\texttt{dirOf}(C) \,.\, dir| \,.\, dir$
   (c) C is removed from Q_k.
   (d) A *transition patch* is created to the segment C.
   (e) Processing continues by tiling C: `TileJoint(`Q_k, `dirOf(`C`)`, `modify(`k`))`.

The function `modify(`k`)` converts the quadrant number k of the call to the quadrant the call came from.
For *forward* segments: `modify`$(k) = (k + 2)$ *modulo* 4, for *backward* segments: `modify`$(k) = (5 - k)$ *modulo* 4.

The function `dirOf(`S`)` returns direction $\vec{dir}_0$ of the first slab of segment S.

To be more precise about the entire procedure, the `TileJoint()` procedure takes four references to the generated mesh nodes of the tiled quadrant as its additional parameters. During the first execution, four mesh vertices related to the quadrant of the calling segment slab are used. Later on, the selected mesh vertices belonging to different segments are passed, reflecting the topology of the surface.

Tiling the surface of the joint is executed in Step 2 of `TileTree()`, after classifying the outgoing segments into *forward* and *backward* subsets. The first executions in Step 2 of `TileTree()`, with quadrant number *from* undefined, have the following shape:

2(a) `TileJoint(`*backward*, *lastSecDir*, undef`)`, where the *lastSecDir* represents the direction of the last slab of the current incoming segment,

2(b) `TileJoint(`*forward*, *straightestDir*, undef`)`, where the *straightestDir* represents $\vec{dir}_0$ of the first slab of the straightest segment in the *forward* subset.

An example of the tiling of three different situations is shown in Fig 7, where the generated patches are drawn as filled quadrilaterals. The *closing* patches are grey and the *transition* ones are grey with a letter *T* in the center, pointing to the recursively executed segment.

If the quadrant has no successors, the recursive procedure `TileJoint()` tiles it directly by a *closing* quadrilateral patch (Fig. 7a).

For each quadrant with successors (one in Fig. 7b and two in Fig 7c) the closest branching segment C is found.

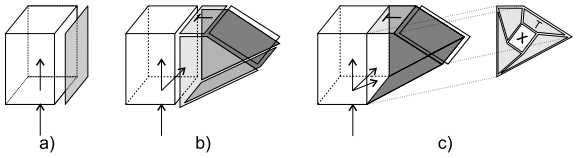

Figure 7. The principle of the recursive branching procedure in one quadrant direction: a) straight-through segment without branching, b) bifurcation, and c) trifurcation

As there is no other segment between the current segment and the nearest segment C, they are directly connected by a *transition* quadrilateral patch (T). The recursive procedure `TileJoint()` is then executed for the remaining three quadrants of the branching segment C, while the direction of C is used as reference.

In Fig. 7b, the case with one branching segment (bifurcation) is shown. The recursive procedure `TileJoint()` creates the *transition* quadrilateral patch (the top one), then recursively executes itself in the remaining three directions. As there are no following branches, each of the three quadrant directions is filled with a *closing* quadrilateral patch (the front, back and bottom one in Fig. 7b).

In Fig. 7c, a trifurcation in one quadrant direction occurs. The recursive executions of `TileJoint()` create the first *transition* patch (top one in the left part of Fig. 7c) and two *closing* quadrilateral patches (back and bottom ones) equally as in the case in Fig. 7b. The quadrant of the front quadrilateral has a successor (marked by *X* in the right part of Fig. 7c). This successor is recursively connected to the remaining front quadrilateral. The *transition* patch (in the right part of Fig. 7c) is created directly, the remaining three *closing* patches are created during the three successive executions of `TileJoint()`.

2.3.4. Up-vector propagation. To generate a minimally twisted surface, the concept of propagation of so-called *up-vectors* is used. The base mesh is generated by tiling the square-shaped cross-sections and the up-vector points to the vertex V_0 of the square (see Fig. 6). By defining the up-vectors as similarly as possible along the segments and carefully rotating in the segment joints in the branches, the squares in neighboring sections are mutually minimally rotated and the surface is minimally twisted. Also the branching itself is simplified, since the edges of branch-squares are mostly parallel to each other. For details refer to [8].

2.4. Operation and memory complexity

The amount of mesh vertices and quads generated by the algorithm is first discussed, followed by the number of recursive calls and the memory complexity. For details about the derivation of the presented equations refer to [8].

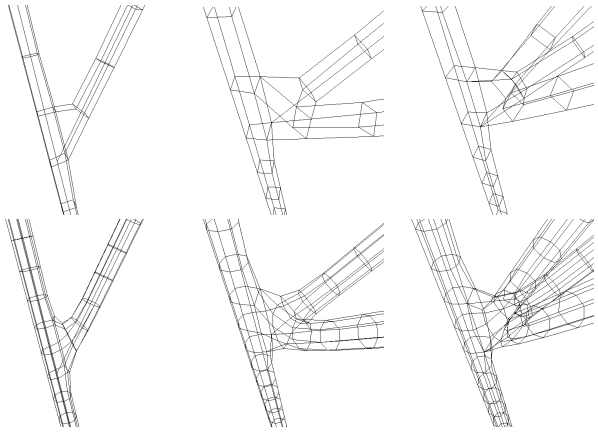

Figure 8. Simulation of branching in the direction of one quadrant used for tests. The first row shows the base mesh, the second row the surface after one subdivision step

2.4.1. Number of generated vertices and quads. Let n be the number of processed segments, m_i the number of vertices of the down-sampled i-th segment centerline, V the number of generated mesh vertices (mesh nodes), and Q the number of generated quadrilaterals. Let S be the number of the input slabs equal to the sum of slabs of the individual input segments $S = \sum_{i=0}^{n-1}(m_i - 1)$.

In general, the number of generated vertices (mesh nodes) is equal to $V = 4(\sum_{i=0}^{n-1} m_i - n + 1) = 4S + 4$, and the number of generated quadrilaterals lies between $Q_{min} = 4(\sum_{i=0}^{n-1} m_i - 1) - (n - 2) = 4S - n + 2$ and $Q_{max} = 4(\sum_{i=0}^{n-1} m_i - 1) = 4S$, where Q_{min} represents the branching of $n - 1$ segments from one, and Q_{max} a chain of consecutive segments.

2.4.2. Operation complexity. The number of operations depends on the structure of the tree. As the number of tree segments n is typically much smaller than the number of input cross-sections S ($n << S$), the overall operation complexity of the proposed tiling algorithm can be approximated as $O(S + n) = O(S)$.

2.4.3. Memory complexity. The algorithm stores the down-sampled input tree with pre-processed directions and normals, and the generated base mesh. This part of memory is fixed, given by the input, and can be approximated as n(tree node size)+ $(S + n)$(tree vertex size) + Q_{max}(mesh quad size)+V(mesh vertex size) $= O(S)$.

The dynamically allocated temporary variables are used in the recursion steps. The maximal recursive depth of the tiling procedures `TileTree()` and `TileJoint()` is $O(n)$. Therefore, the overall memory complexity can be approximated as $O(S + n) = O(S)$.

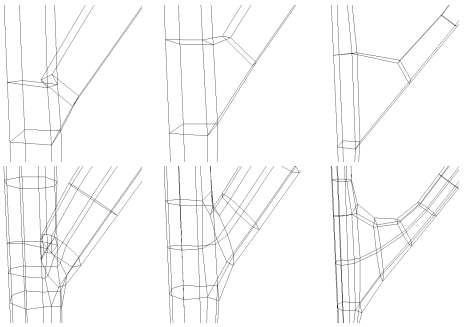

Figure 9. Influence of the sampling step size to the quality of the generated branch surface

3. Tests and results

Extensive testing on simulated datasets (see some of the examples in Fig. 8 and 9) and on real data (see an example in Fig. 10) were performed to test the algorithm qualities.

The 2-manifold property of the algorithm was verified by simulating different versions of bifurcations, tri-furcations and n-furcations in both directions of the branching segments (*forward* and *backward*) and with different amounts of segments branching simultaneously from one quadrant. For "natural" datasets with up to two branches in one segment, the generated surface also looks natural. If more segments branch from one quadrant, the surface in the joint may shrink and narrowings appear.

The algorithm presumes consistent input, where no parts of the centerlines overlap and where cross-sections and parts of the separately reconstructed segment surface do not intersect, aside from segment ends at joints.

In real vessel-tree datasets, if the joints are not precisely located, the centerlines, which originate from the joints, can overlap. Parts of the centerlines will then share the same space between the wrong joint and the true branch. Intersection of the generated surface can be also caused by an incorrect vessel diameter detection. But such cases would be classified as errors of the data segmentation tool applied, and would be assigned a low priority.

For the correctly-segmented data appropriate sampling in the joints is necessary. The simulation of variations of the cross-section sampling step size in the branching point is shown in Fig. 9. The left panel in Fig. 9 shows a step that is too small in relation to the tube diameter, the middle panel an appropriate step, and the right panel a relative sampling step that is too-large. Both problems are discussed in Section 4 below.

To verify the number of vertices and patches of the generated mesh and the complexities derived in section 2.4, six datasets with a different number of segments and with a different branching structure were measured (PC, Pentium III, 600MHz, WinNT, VC++6.0). The results are in Table 1

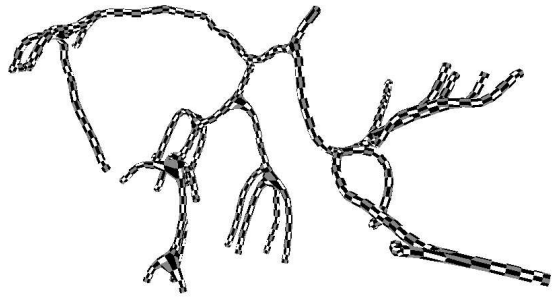

Figure 10. Example of a liver vessel-tree surface (for greater clearity with differently colored patches and a constant radius)

and Figs. 11 and 12. In the table, the input tree segments are classified according to the joint topology: *Chain* is the single follower, *branch* is the branching segment tiled to the *straightest* segment. The table and both graphs show the linear complexity of the algorithm as derived in Section 2.4.2.

We closely cooperate with physicians, and they were satisfied with the speed of the algorithm and with the quality of the generated 3D mesh.

4. Discussion

The proposed method is simple, fast, memory efficient, and produces a topologically correct surface mesh. The mesh is "watertight", which means there are no cracks or holes in the surface. The application of subdivision surfaces for this task is new.

The generated surface is smooth, and, thanks to the up-vector propagation method, it contains minimum twists. Twists can deform the surface, which can be misinterpreted as vessel narrowings (false stenoses).

The reconstruction is sensitive to the input data. The algorithm presumes non-overlapping cross-sections along the vessel segments and in the places of joints (except for where the segment starts and ends). The first property is guaranteed in this application by the nature of the datasets and by the preceding segmentation and down-sampling algorithm. The second property can be achieved by a carefully designed pre-processing step. This preprocessing step is under development, but in our case, the model is used for navigation only and not for diagnostic purposes, and inaccurate surface details do not cause a real problem.

The branching is sometimes enlarged, if the cross-section sampling step is too big, which can be misinterpreted as an aneurism (see the joints in the lower left part of Fig. 10 and an example in the right panel in Fig. 9). Again, this is not a severe problem in this case, as the model is not used for diagnostic purposes, but for surgical navigation and identification of the liver segments.

The resulting surface may self-intersect, if the constellation of branching angle and the cross-section sampling step is set incorrectly (see left panel in Fig. 9). This error can be avoided by skipping cross-sections that would overlap. For small outgoing angles, such cross-sections have the centerlines' distances less than the sum of their radii. As stated above, this preprocessing step is under development and its description will be published soon.

The 2-manifold property in the local vertex neighborhood is given by the way the algorithm constructs a surface. A patch for each edge is generated maximally twice (once at the mesh border in the tree root and in the leaves). By one processing step, the quadrilateral patch is either directly created if no branching segment continues in the processed direction, or the recursive call (in the appropriate quadrant of a slab) joins the edge by a quadrilateral patch with the branching segment.

5. Conclusions and future work

The algorithm presented here constructs a topologically correct surface mesh of branching tubular structures (e.g. a vessel tree) defined by their centerlines and radii. It handles multiple branching via a new recursive tiling scheme. The surface tiling is done in two steps: first, the "quadratic" base mesh is generated. This mesh is then subdivided by means of the Catmull-Clark subdivision scheme, which results in a smooth, topologically correct 2-manifold mesh.

The generated mesh will be used in the Augmented Reality Aided Surgery [1] system, developed in cooperation with the BR1 research group in the VRVis Research Center in Austria. Extended clinical tests will follow.

The input centerline is down-sampled to a lower resolution. An adaptive sampling of the centerline according to the local vessel curvature is still to be tested. The precise positioning of the branching points and modifications of segment starts after the branching (skipping of cross-sections) will also be addressed to avoid overlapping segment parts, causing the constructed surface to self-intersect. Finally, we will compare our method with the convolution surfaces [4].

6. Acknowledgments

This work was funded by the VRVis Research Center, Vienna, and by TIANI Medgraph AG, Austria. The datasets used for tests in this paper are courtesy of Dr. Erich Sorantin and Dr. Georg Werkgartner, LKH Graz.

References

[1] ARAS–Augmented Reality Aided Surgery. http://www.vrvis.at/br1/aras/.

DS	Segments in the input tree: total	classified (root excluded): chain	branch	straightest	Input tree: vertices	sections	Mesh: nodes	patches	Cals of `TileJoint()`	Running time
	n				$\sum_{i=0}^{n-1} m_i$	S	V	Q		[ms]
1	3	0	1	1	15	12	52	47	7	1.80
2	8	2	4	1	58	50	204	196	24	7.01
3	12	4	5	2	63	51	208	199	39	7.51
4	20	3	11	5	111	91	368	353	65	13.52
5	32	5	18	8	177	145	584	562	106	19.53
6	44	5	26	12	263	219	880	850	146	30.84

Table 1. Timings for input trees of different structure and size

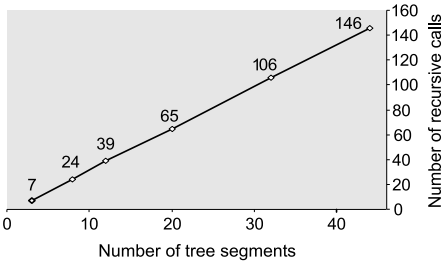

Figure 11. Number of recursive executions of `TileJoint()` in relation to the number of input tree segments n

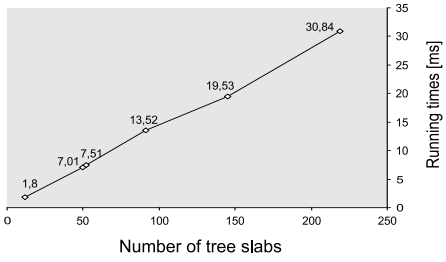

Figure 12. Running times of the mesh generation algorithm in relation to the number of input tree slabs S

[2] H. Biermann and D. Zorin. Subdivide 2.0. Media Research Lab, NYU, Feb. 2001.

[3] J. Bloomenthal. Modeling the Mighty Maple. In *SIGGRAPH'85. Proceedings*, pages 305–311, ACM Press 1985.

[4] J. Bloomenthal. *Skeletal Design of Natural Forms*. PhD thesis, University of Calgary, 1995.

[5] J.-D. Boissonnat and B. Geiger. Three Dimensional Reconstruction of Complex Shapes Based on the Delaunay Triangulation. RR No 1697, INRIA, Sophia Antipolis, Apr. 1992.

[6] H. Delingette. General Object Reconstruction Based on Simplex Meshes. *Int. J. Comp. Vision*, 32(2):111–146, 1999.

[7] T. DeRose, M. Kass, and T. Truong. Subdivision Surfaces in Character Animation. In *SIGGRAPH'1998, Proceedings*, pages 85–94, Orlando, Florida, 1998. ACM Press.

[8] P. Felkel, A. Kanitsar, A. L. Fuhrmann, and R. Wegenkittl. SMART—Surface Models from by Axis- and Radius-defined Tubes. TR-VRVis-2002-008, VRVis Research Center, Vienna, Austria, *www.vrvis.at*, 2002.

[9] P. Felkel, R. Wegenkittl, and A. Kanitsar. Vessel Tracking in Peripheral CTA Datasets – An Overview. In R. Ďurikovič and S. Czanner, editors, *SCCG 2001. Proceedings.*, pages 232–239, Budmerice, Slovakia, 2001. IEEE Comp. Society.

[10] K. K. Hahn, B. Preim, D. Selle, and H.-O. Peitgen. Visualization and Interaction Techniques for the Exploration of Vascular Structures. In T. Ertl, K. Joy, and A. Varshney, editors, *IEEE Visualization 2001*, pages 395–402, 2001. IEEE.

[11] A. Kanitsar, R. Wegenkittl, P. Felkel, D. Fleischmann, D. Sandner, and E. Gröller. CTA: A Case Study of Peripheral Vessel Investigation. In T. Ertl, K. Joy, and A. Varshney, editors, *IEEE Visualization 2001*, pages 477–480, 2001. IEEE.

[12] E. Keppel. Approx. Complex Surfaces by Triangulation of Contour Lines. *IBM J. Res. and Devel.*, 19:2–11, 1975.

[13] H. M. Ladak, J. S. Milner, and S. D. A. Rapid 3D Segmentation of the Carotid Bifurcation from Serial MR Images. *Journal of Biomechanical Engineering*, 122:96–99, 2000.

[14] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH '87. Proceedings*, pages 163–169, ACM Press 1987.

[15] R. Malladi and J. A. Sethian. Level Set Methods for Curvature Flow, Image Enhancement, and Shape Recovery in Medical Images. In *Proceedings of Conference on Visualization and Mathematics*, pages 329–345. Berlin, Germany, Springer-Verlag, Heidelberg, Germany, June 1995.

[16] D. Meyers, S. Skinner, and K. Sloan. Surfaces from Contours. *ACM Trans. on Graphics*, 11(3):228–258, July 1992.

[17] J.-M. Oliva, M. Perrin, and S. Coquillart. 3D Reconstruction of Complex Polyhedral Shapes from Contours using a Simplified Generalized Voronoï Diagram. *Computer Graphics Forum*, 15(3):C–397–C–408, 1996.

[18] B. Payne and A. Toga. Surface Reconstruction by Multiaxial Triangulation. *IEEE CG&A*, 14(6):28–35, Nov. 1994.

[19] G. M. Treece, R. W. Prager, A. H. Gee, and L. Berman. Surface Interpolation from Sparse Cross-Sections Using Region Correspondence. Technical Report CUED/F-INFENG/TR 342, Cambridge University, Dept. of Engineering, 1999.

[20] A. Wahle, S. C. Mitchell, S. D. Ramaswamy, K. B. Chandran, and M. Sonka. Four-dimensional coronary morphology and computational hemodynamics. In M. Sonka and K. M. Hanson, editors, *Medical Imaging 2001: Image Processing*, volume 4322, pages 743–754, Bellingham WA, Feb. 2001. SPIE Proceedings.