HexEx: Robust Hexahedral Mesh Extraction

Max Lyon* RWTH Aachen
David Bommes* RWTH Aachen
Leif Kobbelt* RWTH Aachen

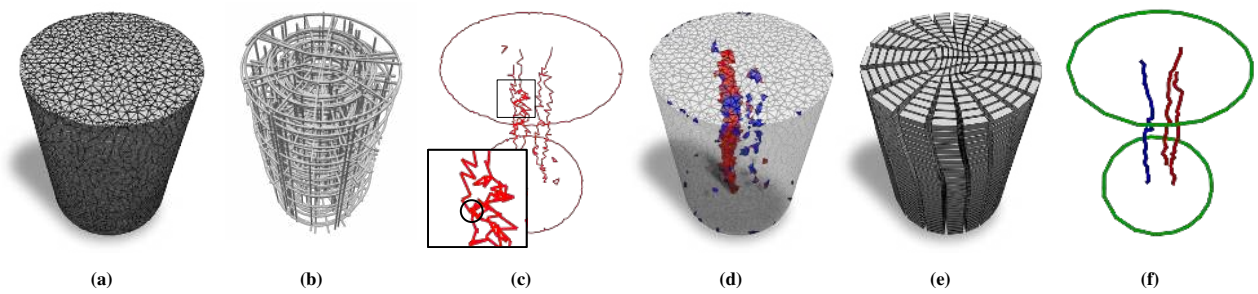


Figure 1: *Given the tetrahedral mesh* CYLINDER *as input in (a), the frame-field in (b) induces the singularity graph in (c). Note that having four valence-three singularities pass through the cylinder is a common way of parametrizing a cylindrical mesh. Unfortunately, the two left singularity curves meet at a common vertex (marked by a circle) in the middle of the cylinder, forcing both edge strips onto the same integer line in parametric space. This causes the parameter image of the many tetrahedra shown in (d) to degenerate (blue) or flip (red). While these imperfections of the parametrization are a difficult problem for naive mesh extraction algorithms, our algorithm is able to extract the sensible all-hexahedral mesh in (e). The singularity graph in (f) shows that the two singular valence-three curves were merged into a single valence-two curve.*

Abstract

State-of-the-art hex meshing algorithms consist of three steps: Frame-field design, parametrization generation, and mesh extraction. However, while the first two steps are usually discussed in detail, the last step is often not well studied. In this paper, we fully concentrate on reliable mesh extraction.

Parametrization methods employ computationally expensive countermeasures to avoid mapping input tetrahedra to degenerate or flipped tetrahedra in the parameter domain because such a parametrization does not define a proper hexahedral mesh. Nevertheless, there is no known technique that can guarantee the complete absence of such artifacts.

We tackle this problem from the other side by developing a mesh extraction algorithm which is extremely robust against typical imperfections in the parametrization. First, a sanitization process cleans up numerical inconsistencies of the parameter values caused by limited precision solvers and floating-point number representation. On the sanitized parametrization, we extract vertices and so-called darts based on intersections of the integer grid with the parametric image of the tetrahedral mesh. The darts are reliably interconnected by tracing within the parametrization and thus define the topology of the hexahedral mesh. In a postprocessing step, we let certain pairs of darts cancel each other, counteracting the effect of flipped regions of the parametrization. With this strategy, our algorithm is able to robustly extract hexahedral meshes from imperfect parametrizations which previously would have been considered defective. The algorithm will be published as an open source library [Lyon et al. 2016].

Keywords: hex meshing, parametrization, mesh extraction

Concepts: •**Applied computing** → *Computer-aided design;* •**Computing methodologies** → *Shape modeling;* Physical simulation;

*e-mail:{lyon,bommes,kobbelt}@cs.rwth-aachen.de

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
SIGGRAPH '16 Technical Papers, July 24 - 28, 2016, Anaheim, CA
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925976

1 Introduction

High-quality meshes are of great interest for various kinds of simulations. Finite element methods, for example, solve complicated problems such as partial differential equations (PDE) by discretizing a volume into a mesh consisting of many small cells. Since they are easy to generate, these cells are often tetrahedra. Hexahedral meshes, however, are better suited for these tasks since they typically require only 10–25% the number of elements of tetrahedral meshes to achieve the same accuracy [Shepherd and Johnson 2008]. They are suitable for a multilevel hierarchy of nested meshes which can enhance the speed and accuracy of PDE solvers significantly [Nieser et al. 2011].

Unfortunately, the generation of these meshes takes up a lot of time. Shimada [2006] reports that only 20% of the total time spent on modeling and simulation techniques is used for analysis, while setting up the problem takes up 80%. While the generation of tetrahedral meshes may only take hours or days, designing a hexahedral mesh can take several months [Shepherd and Johnson 2008] because they are often still constructed by hand to ensure the correct alignment to the current problem.

Recent developments, e. g. by Nieser et al. [2011] or Li et al. [2012], prove parametrization-based hex meshing algorithms to be a promising approach for automated hex mesh generation from given tetrahedral meshes. The general structure of these techniques consists of three steps [Nieser et al. 2011]:

1. Design of a *guiding frame-field*

2. Generation of a *parametrization* which aligns to that field
3. *Extraction* of the hexahedral mesh from the parametrization

While the first two steps are discussed in detail, most publications only mention the third step briefly. We show that, in practice, the extraction step is not trivial since parametrizations usually contain numerical inaccuracies and flipped elements. These imperfections cause simple extraction algorithms to generate inconsistent hexahedral meshes containing non-hex elements or entirely missing elements.

With *HexEx*, we provide a robust algorithm that is able to extract an all-hex mesh, even from a defective input parametrization. It consists of the following four phases:

1. Preprocessing: Transition functions are extracted and the parametrization is made numerically consistent.
2. Geometry extraction: Still containing flipped elements, the sanitized parametrization is used to extract vertices representing the output geometry.
3. Topology extraction: Topological information is extracted by interconnecting previously generated darts.
4. Postprocessing: Topological inconsistencies caused by degeneracies of the parametrization are resolved.

1.1 Related Work

With *CubeCover* Nieser et al. [2011] presented a first approach for automatic generation of hexahedral meshes using frame-fields to align the individual hexahedra. For their algorithm, the user designs a coarse hexahedral mesh which covers the whole tetrahedral input mesh. From this coarse mesh a frame is calculated for each tetrahedron. The parametrization that aligns the tetrahedra as well as possible to these frames is obtained by minimizing the energy

$$E(f) = \int_V ||\nabla f - X||^2 \mathrm{d}vol \ ,$$

where X is the frame-field, V is the set of all tetrahedra and f are piecewise linear functions mapping each tetrahedron into the parameter domain $\mathbb{R}^3$.

Li et al. [2012] propose a method for the automatic generation of guiding frame-fields. After the frame-field generation, a series of operations, such as edge collapses and tetrahedron splits is applied to obtain a *singularity-restricted field*. Nieser et al. [2011] define 24 types of singularities, 14 of which inevitably lead to zero volume tetrahedra in the parametrization. The singularity-restricted field only contains the 10 types of singularities that do not. However, flipped elements may still be present and need to be handled during mesh extraction. With our extraction algorithm, these mesh editing operations are not necessary, as it is able to robustly extract all-hex meshes even in the presence of degenerate tetrahedra.

Jiang et al. [2014] follow a similar approach as Li et al. For a given frame-field they present further operations to transform the parametrization. One example is moving a singularity that runs along two edges of a triangle to the other edge of that triangle, which prevents the degeneration of the triangle in the parameter domain that would have been caused by forcing the two edges to run along the same integer iso-line. Due to the robustness of our algorithm towards degenerate cells, it is not necessary to explicitly remove singular edge configurations that cause small regions of the parametrization to degenerate.

With *QEx*, Ebke et al. [2013] presented an algorithm to robustly extract quadrilateral meshes from an imperfect parametrization of the triangular input mesh. Like our algorithm, QEx applies a sanitization step to get rid of numerical errors in the parametrization, allowing the use of exact predicates for robustness in the following steps. In order to convert non-quad elements, initially extracted due to flips in the parametrization, QEx relies on a vertex merging procedure which merges vertices based on their local parameter. Unfortunately, since QEx only merges vertices with equal parameter coordinate on a per face basis, it is not able to reliably extract a quad mesh if larger areas are flipped, such that whole quads lie in the flipped region. Our more global iterative post processing procedure is able to handle such cases by letting flipped and non-flipped regions cancel each other out.

1.2 Contribution

Our two main contributions are a robust hexahedral mesh extraction algorithm which is able to handle most typical kinds of degeneracies in the input parametrization and an open source C++ reference implementation of our algorithm [Lyon et al. 2016].

2 Terminology

2.1 Mesh

A *combinatorial 3-dimensional polytopal complex* is a mesh that consists of a set of conforming d-polytopes, $0 \leq d \leq 3$, with underlying incidence and adjacency relations [Kremer et al. 2012]. The 0-, 1-, 2- and 3-dimensional polytopes are called vertices, edges, faces and cells, respectively. For $d > 0$, each d-dimensional polytope is bounded by $(d-1)$-polytopes. Two polytopes are considered *incident* if one of them is entirely part of the others boundary. Two d-dimensional polytopes are called *adjacent* if they share a common $(d-1)$-polytope on their boundary. Two vertices are adjacent if they are incident to a common edge.

A mesh is then given as $\mathcal{M} = (V, E, F, C)$ where V, E, F and C denote the set of vertices, edges, faces and cells, respectively. In contrast to combinatorial polytopal complexes where vertices are abstract entities, vertices in a *geometric* polytopal complex have a *geometric embedding*, i. e. a function $\mathbf{g} : V \to \mathbb{R}^n$ [Kremer et al. 2012]. In our case $n = 3$. For a shorter notation, we use italic letters (typically p, q, r, s) for vertices and bold face letters ($\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}$) for their geometric embedding .

We identify an edge $e \in E$ by the two incident vertices, e. g. $e = (p, q)$. Analogously, we identify a face $f = (p_1, p_2, ..., p_k)$ by a sequence of vertices, where f is the face bounded by edges $e_i = (p_i, p_{i+1})$ and $e_k = (p_k, p_1)$, for $0 < i < k$.

A tetrahedral mesh consists entirely of tetrahedral cells, i. e. cells bounded by four triangular faces. A tetrahedral cell is also called tetrahedron or short *tet*. A tet $c = (p, q, r, s) \in C$ is bounded by the four faces that each are incident to three of the vertices. As a convention, the vertices of the tet are ordered such that the determinant $\det\begin{pmatrix}\mathbf{q}-\mathbf{p} & \mathbf{r}-\mathbf{p} & \mathbf{s}-\mathbf{p}\end{pmatrix}$ is positive.

A mesh is a *3-manifold* (with boundary) if every point is either locally homeomorphic to a sphere or a half sphere [Kremer et al. 2012]. The former are called inner, the latter boundary points. We require as input such a 3-manifold tetrahedral mesh.

2.2 Parametrization

In analogy to Ebke et al. [2013] and Bommes et al. [2013], a 3D integer-grid map $\mathbf{f}$ is the union of linear maps $\mathbf{f}_{c_i} : \mathbb{R}^3 \to \mathbb{R}^3$ that map each tet $c_i = (p_i, q_i, r_i, s_i) \in \mathcal{M}$ to a tet $(\mathbf{u}_i, \mathbf{v}_i, \mathbf{w}_i, \mathbf{x}_i) \in$

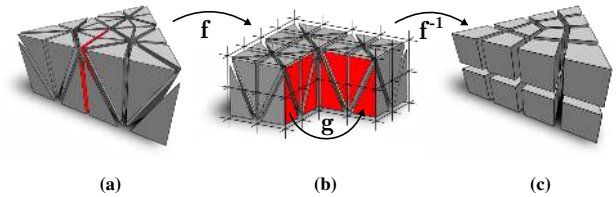


Figure 2: *(a) Input tetrahedral mesh. To allow a singular edge in the center, the mesh is cut open along the red faces. (b) Mesh in parametric space. (c) Output mesh defined by parametrization.*

$\mathbb{R}^{3\times4}$ in the parameter domain. The parametrization of two adjacent tets t_i and t_j is related by the transition function $\mathbf{g}_{ij}$. Given a closed loop of tets $(c_0, c_1, ..., c_k, c_0)$ around an edge e that starts in the cell c_0 incident to e and passes through all incident cells, the edge e is defined to be singular if the accumulated transition function $\mathbf{g}_e = \mathbf{g}_{k0} \circ ... \circ \mathbf{g}_{12} \circ \mathbf{g}_{01}$ is not the identity [Nieser et al. 2011]. Singular vertices are defined as vertices incident to other than two singular edges.

The 3D integer-grid map must satisfy the following constraints:

(A1) The transition functions $\mathbf{g}_{ij}$ mapping the chart of tet c_i to the chart of the adjacent tet c_j have to be 3D grid automorphisms, i. e. be of the form

$$\mathbf{g}_{ij}(\mathbf{u}) = \mathbf{\Pi}_{ij}\mathbf{u} + \mathbf{t}_{ij} \ ,$$

where $\mathbf{\Pi}_{ij}$ is an element of the chiral cubical symmetry group $\mathcal{G}$, containing the 24 orientation preserving transformations that map coordinate axes to coordinate axes [Nieser et al. 2011], and $\mathbf{t}_{ij} \in \mathbb{Z}^3$ is an integer translation.

(A2) Singular edges have to be mapped to segments on integer lines, i. e.

$$\mathbf{f}(p,q) = \left(\mathbf{\Pi}(a,b,c)^T, \mathbf{\Pi}(a,b,d)^T\right) \ \forall (p,q) \in S_e \ ,$$

for some $a, b \in \mathbb{Z}$, $c, d \in \mathbb{R}$ and $\mathbf{\Pi} \in \mathcal{G}$, where $S_e \subseteq E$ is the set of singular edges in $\mathcal{M}$.

(A3) Points incident to other than two singular edges have to be mapped to integer points, i. e.

$$\mathbf{f}(p) = \mathbf{u} \in \mathbb{Z}^3 \ \forall p \in S_v \ ,$$

where $S_v \subseteq V$ is the set of singular points in $\mathcal{M}$.

(A4) The image of each tet has to have a positive volume:

$$\det(\mathbf{v}_i - \mathbf{u}_i \quad \mathbf{w}_i - \mathbf{u}_i \quad \mathbf{x}_i - \mathbf{u}_i) > 0 \ \forall c_i \in C \ .$$

Figure 2 shows an example of a parametrization obeying these constraints. To allow a singular valence 3 edge in the output, the input mesh in Figure 2a is cut open along the highlighted faces and the central edge is mapped onto an integer grid line in Figure 2b. The regular integer grid then induces the hexahedral mesh in Figure 2c with the desired topology.

Computing a parametrization that satisfies Constraint (A4) is still an open problem and can even be infeasible for a given frame-field [Jiang et al. 2014]. Additionally, due to numerical inaccuracies with floating-point arithmetic, Constraints (A1) to (A3) are often only fulfilled approximately.

Therefore, we define relaxed 3D integer-grid maps in analogy to Ebke et al. [2013] to be parametrizations that satisfy Constraints (A1), (A2) and (A3) approximately and disregard Constraint (A4) entirely.

As input for our algorithm we require a tetrahedral mesh with such a relaxed 3D integer-grid map as parametrization.

3 Output Data Structure

The data structure used for our algorithm is based on that of Kraemer et al. [2014]. For a volumetric mesh $\mathcal{M} = (V, E, F, C)$, we store a set of vertices with a geometric embedding to describe its geometry. The topology is defined by a generalized map which consists of a set D of so-called darts and pointers α_i interconnecting the darts.

The set of darts $D \subset V \times E \times F \times C$ is defined as a set of tuples, such that

$$D = \{(v, e, f, c) \ : \ v \sim e, e \sim f, f \sim c\}$$

where $\sim$ means "incident to".

For each dart $d = (v, e, f, c) \in D$, four connections are stored to other darts, which are uniquely defined by:

$$\begin{aligned}\alpha_0(d) &= (v', e, f, c)\\ \alpha_1(d) &= (v, e', f, c)\\ \alpha_2(d) &= (v, e, f', c)\\ \alpha_3(d) &= (v, e, f, c')\end{aligned}$$

where $\alpha_i(d) \neq d$ and $\alpha_i(d) \in D$. For boundary faces there is no such $\alpha_3(d)$.

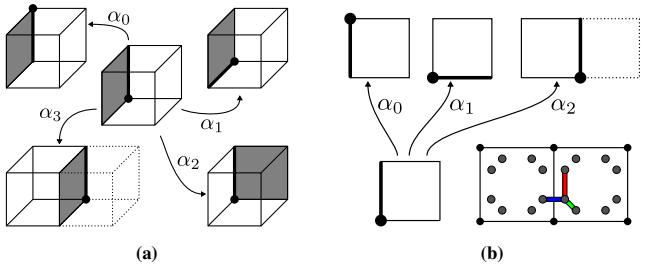


Figure 3: *(a) Illustration of α_i connections in 3D. Here, a dart $d = (v, e, f, c)$ is represented by the cell c where the vertex v, edge e and face f are highlighted. (b) α_i connections in 2D, where a dart $d = (v, e, f)$ is represented by the face f where the vertex v and edge e are highlighted. In further illustrations, we represent d as a small point that is closest to the corresponding entities (bottom right). α_0 connections are drawn in red, α_1 in green and α_2 in blue.*

In the data structure, only vertices are stored explicitly. Edges, faces and cells are stored implicitly:

Edges are bounded by the vertices reached via only α_0 pointers. Faces are bounded by edges reached via only α_0 and α_1 pointers. Cells are bounded by faces reached via all pointers except α_3 pointers. One connected component of a mesh consists of all cells reachable using any pointers.

Note, that, while we have given the definition of darts for 3-dimensional geometry, it is straightforward to generalize darts for n-dimensional geometry. In particular, we get a definition in 2D simply by omitting cells and α_3 pointers. In the following, we will often use 2D illustrations to explain concepts as they are both easier to sketch and easier to understand.

3.1 Properties

In this section, we list some interesting properties of the data structure described above.

Given a set $S \subseteq \{\alpha_0, ..., \alpha_3\}$ and a dart d, the orbit $\langle S \rangle(d)$ is defined as the set of darts reachable from d by following any combination of α_i's in S [Kraemer et al. 2014]. As a simplified notation, we drop the curly braces when specifying S explicitly, e. g. $\langle \alpha_0, \alpha_3 \rangle(d) = \langle \{\alpha_0, \alpha_3\} \rangle(d)$.

Two adjacent faces are connected over the whole shared edge:

$$\forall d \in D \ : \ \forall d' \in \langle \alpha_0 \rangle(d) \ : \ \alpha_2(d) \in \langle \alpha_0 \rangle(\alpha_2(d')) \quad (1)$$

Two adjacent cells are connected over the whole shared face:

$$\forall d \in D \ : \ \forall d' \in \langle \alpha_0, \alpha_1 \rangle(d) \ : \ \alpha_3(d) \in \langle \alpha_0, \alpha_1 \rangle(\alpha_3(d')) \quad (2)$$

If $\mathcal{M}$ is a hexahedral mesh, we can further formulate the following properties which hold for all $d \in D$ and :

For each dart $d \in D$:

Each face is a quad:

$$|\langle \alpha_0, \alpha_1 \rangle(d)| = 8 \quad (3)$$

Each corner is incident to three faces:

$$|\langle \alpha_1, \alpha_2 \rangle(d)| = 6 \quad (4)$$

Each quad strip within a cell consists of four quads

$$|\langle \alpha_0, \alpha_1 \circ \alpha_2 \circ \alpha_1 \rangle(d)| = 8 \quad (5)$$

Each cell consists of six half faces:

$$|\langle \alpha_0, \alpha_1, \alpha_2 \rangle(d)| = 48 \quad (6)$$

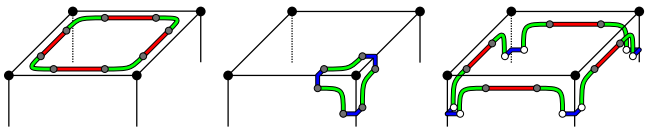


Figure 4: *Illustration of properties 3 to 5. Note, the white darts in (c) are only shown for orientation, they are not in the set* $\langle \alpha_0, \alpha_1 \circ \alpha_2 \circ \alpha_1 \rangle(d)$.

4 Algorithm

Our extraction algorithm can be divided into four phases described in the following sections. During the geometry extraction, the input tetrahedra are analyzed and checked for intersections with the integer grid points in parametric space, yielding the vertices of the hexahedral mesh which define its geometry (Section 4.2). In a similar fashion, we extract darts, which get connected to each other during the topology extraction, defining the topology of the hexahedral mesh (Section 4.3).

For a perfect parametrization, these two steps would already yield the desired hexahedral mesh. However, since we aim for a robust algorithm that is able to extract meaningful hexahedral meshes even for imperfect parametrizations, we employ two additional steps. The preprocessing step sanitizes the parametrization, compensating for the limited precision of numerical solvers and floating-point numbers, and thus enforcing Constraints (A1) to (A3) (Section 4.1). In the postprocessing phase, artifacts caused by flipped tetrahedra, such as duplicate vertices, are corrected (Section 4.4).

4.1 Preprocessing

Our preprocessing consists of two steps. First, we extract the transition functions $\mathbf{g}_{ij}$ that map the parameters from the chart of t_i to the chart of t_j. Then, during the sanitization step, we enforce the exact fulfillment of Constraints (A1) to (A3). The benefit of this is that all subsequent steps of the algorithm can employ exact predicates to check geometric properties, yielding reliable results and simplifying the implementation significantly.

4.1.1 Extracting the Transition Functions

As a first step, we extract from the parametrization the transition functions $\mathbf{g}_{ij}(u) = \mathbf{\Pi}_{ij} u + t_{ij}$ between all adjacent tets. Unfortunately, it is impossible to recover the matching $\mathbf{\Pi}_{ij}$ if the face f between the two tets t_i and t_j is degenerate. However, this is no limitation, since most parametrization techniques fix the matchings prior to the computations of the parametrization based on a frame-field. We therefore consider the matchings as given. The integer translation t_{ij} can then easily be calculated by

$$t_{ij} = \text{round}\left(\mathbf{g}_{ij}(u) - \mathbf{\Pi}_{ij} u\right)$$

4.1.2 Sanitizing the Parametrization

The purpose of the sanitization is to ensure the exact fulfillment of Constraints (A1) to (A3). While we made sure in the previous step that the translations $\mathbf{t}_{ij}$ of the transition functions are exact integers, it is still possible that for a vertex p its parameter $\mathbf{u} = \mathbf{f}_{c_i}(\mathbf{p})$ in cell c_i is not equal to the transformed parameter $\mathbf{g}_{ji}\left(\mathbf{f}_{c_j}(\mathbf{p})\right)$ from an incident cell c_j, due to the limited precision of floating point arithmetic.

To remedy this, we apply the same strategy as Ebke et al. [2013]: we pick for each vertex v an arbitrary incident cell c from which we propagate its parameter $\mathbf{u}$ into all other cells incident to v according to the respective transition functions. During this procedure, we have to ensure that Constraint (A1) is fulfilled exactly in both directions, i. e. rounding during any calculation must not occur. Rounding can happen when the floating-point representation of a parameter in one chart needs a larger exponent than in another chart, thus losing one significant digit in the mantissa. To prevent this, we find the largest exponent of all parameters of v in all incident cells. The parameter values of v in c are truncated accordingly before propagation. Additionally, in order to fulfill Constraints (A2) and (A3), we round each component of $\mathbf{u}$ to the closest integer if the distance is less than the maximum precision ε of the solver that was used to generate the parametrization.

After the sanitization, we only use exact predicates for all calculations in the following steps of the algorithm.

4.2 Geometry Extraction

Extracting the geometry of the hexahedral mesh consists mainly of finding all its vertices and their geometric embedding as described in the following section. We also describe the generation of darts here as this is, like the generation of vertices, a local operation that can be performed for each tet individually, whereas finding the connection between darts as explained in Section 4.3.1 involves several tets per dart.

4.2.1 Vertex Extraction

As already stated, each integer grid location intersecting with the parametrization defines a vertex of the hexahedral mesh, or short h-vertex. After the previous step, we can use exact predicates, such

as those provided by Shewchuk [1997], to reliably detect such locations. The algorithm is straightforward and analog to that of Ebke et al. [2013]. While it would be possible to enumerate all h-vertices by iterating over all tets and checking if an integer grid point intersects the parameter image of the tet, this would also lead to a lot of duplicate vertices, as integer grid locations intersecting a tet on the boundary may intersect several other tets on their boundary as well. We therefore iterate over all vertices, edges, faces and cells separately and exclude their respective boundaries. We call the entity intersecting with an integer grid point the generator of the h-vertex.

For each generated h-vertex, we compute a geometric embedding according to its barycentric coordinates with respect to its generator.

4.3 Topology Extraction

The topology extraction phase of our algorithm consists of two steps. First, we extract darts in a similar way we extracted vertices, based on intersections of the integer grid with the tet mesh in parametric space as discussed in Section 4.3.1. Then, these darts are interconnected by carefully navigating through the input mesh according to Section 4.3.2.

4.3.1 Dart Extraction

The parametrization implies the structure of the hexahedral mesh as the intersection of the parameter images of the tets with the regular grid. In Section 4.2.1, we already extracted h-vertices at intersections of integer grid points with the parameter image of the input mesh. In this section, we take this concept further in order to extract the darts that will later define the topology of the hexahedral mesh. We define the parametric volume $\mathcal{T}_{\text{et}}$ of a tet t with parameters $(\mathbf{u}, \mathbf{v}, \mathbf{w}, \mathbf{x})$ as:

$$\mathcal{T}_{\text{et}}(t) = \{\alpha\mathbf{u} + \beta\mathbf{v} + \gamma\mathbf{w} + (1-\alpha-\beta-\gamma)\mathbf{x} \ : \ 0 \leq \alpha, \beta, \gamma \leq 1\} \tag{7}$$

Furthermore, let $\mathbf{z} \in \mathbb{Z}^3$ be an integer grid point and $\vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2$ and $\vec{\mathbf{d}}_3$ be orthonormal and restricted to the six axis directions. We then define an integer grid edge as

$$\mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1) = \left\{\mathbf{z} + \alpha\vec{\mathbf{d}}_1 \ : \ 0 < \alpha < 1\right\} \ ,$$

an integer grid face as

$$\mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2) = \left\{\mathbf{z} + \alpha\vec{\mathbf{d}}_1 + \beta\vec{\mathbf{d}}_2 \ : \ 0 < \alpha, \beta < 1\right\} \ ,$$

and an integer grid cell as

$$\mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3) = \left\{\mathbf{z} + \alpha\vec{\mathbf{d}}_1 + \beta\vec{\mathbf{d}}_2 + \gamma\vec{\mathbf{d}}_3 \ : \ 0 < \alpha, \beta, \gamma < 1\right\} \ .$$

Note that we use $<$ rather than $\leq$ to define $\mathcal{E}, \mathcal{F}$ and $\mathcal{C}$ in order to exclude their boundaries.

We now extract a dart d for every vertex v, edge e, face f and cell c of the hexahedral mesh where $v \sim e$, $e \sim f$ and $f \sim c$. Thus, we have to find all tets t, integer grid points $\mathbf{z}$, and orthonormal vectors $\vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2$ and $\vec{\mathbf{d}}_3$ such that all the following conditions are fulfilled:

(B1) The integer grid point intersects the tet.

$$\{\mathbf{z}\} \cap \mathcal{T}_{\text{et}}(t) \neq \emptyset \tag{8}$$

(B2) The integer line starting at $\mathbf{z}$ going into direction $\vec{\mathbf{d}}_1$ intersects the tet.

$$\mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1) \cap \mathcal{T}_{\text{et}}(t) \neq \emptyset$$

Input: tetrahedral mesh (V, E, F, C), sanitized map $\mathbf{f}$,
1: **for each** h-vertex h with embedding $\mathbf{p}$ and generator g **do**
2: **for each** tet $t \in C$ incident to g **do**
3: $\mathbf{z} \leftarrow \mathbf{f}_t(\mathbf{p})$
4: **for each** orthonormal and axis aligned $\vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3$ **do**
5: **if** (B2) **and** (B3) **and** (B4) **then**
6: generate dart $d = (\mathbf{z}, \mathcal{E}, \mathcal{F}, \mathcal{C}, t)$

Algorithm 1: *Dart extraction*

(B3) The integer plane with one side being $\mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1)$ and extending into direction $\vec{\mathbf{d}}_2$ intersects the tet.

$$\mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2) \cap \mathcal{T}_{\text{et}}(t) \neq \emptyset \tag{9}$$

(B4) The integer cube with base $\mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2)$ extending into direction $\vec{\mathbf{d}}_3$ intersects the tet.

$$\mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3) \cap \mathcal{T}_{\text{et}}(t) \neq \emptyset \tag{10}$$

As an alternative notation for $d = (v, e, f, c)$, we will now use $d = (\mathbf{z}, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3), t)$ or short $d = (\mathbf{z}, \mathcal{E}, \mathcal{F}, \mathcal{C}, t)$ where $\mathbf{z}$, $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{C}$ are the parameter images of v, e, f and c, respectively, in the chart of t.

Algorithm 1 describes how to extract all darts. Since we already have extracted all vertices, we do not have to check for Condition (B1) anymore. Instead we can iterate over all extracted h-vertices and check for each tet that is incident to the generator of that h-vertex if Conditions (B2) to (B4) are fulfilled. We call darts extracted for flipped tets *anti darts*, and refer to all others as *regular darts*.

4.3.2 Connection Extraction

Having extracted the vertices and darts from the parametrization, the mesh data structure still lacks connectivity information. In the following step, we establish connectivity by assigning the four connecting darts indicated by α_i for every extracted dart d. In the easiest case, the other darts we want to connect d to were extracted in the same tet as d. We can then simply look them up in the list of darts extracted for t according to the following description. For dart $d = (\mathbf{z}, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3), t)$:

$$\begin{aligned}
\alpha_0^*(d) &= (\mathbf{z} + \vec{\mathbf{d}}_1, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3), t) \ , \\
\alpha_1^*(d) &= (\mathbf{z}, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_2), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3), t) \ , \\
\alpha_2^*(d) &= (\mathbf{z}, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_3), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, \vec{\mathbf{d}}_3), t) \ , \\
\alpha_3^*(d) &= (\mathbf{z}, \mathcal{E}(\mathbf{z}, \vec{\mathbf{d}}_1), \mathcal{F}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2), \mathcal{C}(\mathbf{z}, \vec{\mathbf{d}}_1, \vec{\mathbf{d}}_2, -\vec{\mathbf{d}}_3), t) \ .
\end{aligned}$$

For injective parametrizations which only contain trivial transitions between tets, the darts can be found similarly easily. One can search in the list of all extracted darts for a match of the above description, ignoring the tet the dart was extracted for. However, in most real-world examples, the parametrization will contain transitions which have to be considered. We therefore give the following conditions under which we connect two darts:

Two darts $d = (\mathbf{z}, \mathcal{E}, \mathcal{F}, \mathcal{C}, t)$ and $d' = (\mathbf{z}', \mathcal{E}', \mathcal{F}', \mathcal{C}', t')$ are connected via an α_i pointer if all following conditions are fulfilled:

(C1) There exists a chain of adjacent tets $\{t_0, ..., t_n\}$ with $t_0 = t$ and $t_n = t'$.

(C2) Let $\mathbf{g}_j$ be the transition function mapping from t_j to t_{j+1} and $\mathbf{g}_{jk} = \mathbf{g}_k \circ \mathbf{g}_{k-1} \circ ... \circ \mathbf{g}_{j+1} \circ \mathbf{g}_j$ for $j < k$.

$$\begin{aligned} \mathbf{g}_{0n}(\mathbf{z}) &= \mathbf{z}' && \text{or} && i = 0\ , \\ \mathbf{g}_{0n}(\mathcal{E}) &= \mathcal{E}' && \text{or} && i = 1\ , \\ \mathbf{g}_{0n}(\mathcal{F}) &= \mathcal{F}' && \text{or} && i = 2\ , \\ \mathbf{g}_{0n}(\mathcal{C}) &= \mathcal{C}' && \text{or} && i = 3\ . \end{aligned}$$

where i refers to α_i, i. e. depending on the type of connection one entity may be different.

(C3) For a face f with parameter image $(\mathbf{u}, \mathbf{v}, \mathbf{w})$, we define analogously to Equation (7)

$$\mathcal{T}_{\text{ri}}(f) = \{(1-\alpha-\beta)\mathbf{u} + \alpha\mathbf{v} + \beta\mathbf{w} \ : \ 0 \leq \alpha, \beta \leq 1\}\ .$$

Let f_k be the face shared by tets t_k and t_{k+1}. Each face has to intersect the parametric image of the three entities that are incident to both d and d':

$$\begin{aligned} \mathbf{g}_{0k}(\{\mathbf{z}\}) \cap \mathcal{T}_{\text{ri}}(f_k) \neq \emptyset && \text{or} && i = 0\ , \\ \mathbf{g}_{0k}(\mathcal{E}) \cap \mathcal{T}_{\text{ri}}(f_k) \neq \emptyset && \text{or} && i = 1\ , \\ \mathbf{g}_{0k}(\mathcal{F}) \cap \mathcal{T}_{\text{ri}}(f_k) \neq \emptyset && \text{or} && i = 2\ , \\ \mathbf{g}_{0k}(\mathcal{C}) \cap \mathcal{T}_{\text{ri}}(f_k) \neq \emptyset && \text{or} && i = 3\ . \end{aligned}$$

To give an intuition of the meaning of the these conditions, let us consider the α_0 connection. In this case, the chain of tets fulfilling the conditions corresponds to the result of an integer iso line tracing starting at $\mathbf{z}$ in t_0, going into direction $\vec{\mathbf{d}}_1$, passing through faces f_k and reaching $\mathbf{z} + \vec{\mathbf{d}}_1$ in tet t_n, all while considering the transition functions. During tracing, the next tet is entered through the face that is intersected by the integer iso line. In some cases however, the integer iso line may intersect two faces on their common edge or even three faces on their common vertex. For these cases, Condition (C3) provides a consistent decision which face to choose. Also, note that when tracing the iso line and switching from a tet that is flipped to one that is not, or vice versa, one has to change tracing direction, because the new cell simply does not continue into the old direction. In this case, the tracing stops when the start parameter $\mathbf{z}$ is reached again (under consideration of the transitions).

Analogously, for the other α_i, the chains correspond to a generalized form of tracing. For α_1, one traces rotationally within the current face towards the other edge. For α_2, one traces rotationally around the current edge. In both cases, the direction of rotation is uniquely defined by having to stay in the same integer grid cell. For α_3, the other dart is almost always found in the original tet, or, if a face of the tet aligns with the integer grid face, in the neighboring tet. If the chain contains more than two tets, all but the first and last one are degenerate.

Algorithm 2 finds for each dart the four darts to connect to according to the previously stated rules. Starting with a dart d, we can easily generate the expected partner using α_i^*. If this dart was generated during the geometry extraction we can connect the two darts. If not, we keep looking in the next adjacent tet that we reach over the face fulfilling Condition (C3). Note that most of the times, there are two faces that fulfill this condition. Therefore, we ignore the face that was used to enter the current dart, as this face would bring us back into a tet that was already checked. Should the other face be a boundary face, we leave the dart unconnected.

When entering the next cell, we have to consider the transition function and update our dart descriptions accordingly. Also, we have to check whether the flippedness of the tet changed. Whenever we pass from a flipped region into a regular one, or vice versa, we swap the descriptions of the start dart d_s and target dart d_t. We repeat this process until we find the target dart, or we run into the boundary of the input mesh.

Input: tet mesh (V, E, F, C), sanitized map $\mathbf{f}$, darts D
for each dart $d \in D$ **do**
  for each $i \in \{0, 1, 2, 3\}$ **do**
    $d_s \leftarrow d$
    $d_e \leftarrow \alpha_i^*(d_s)$
    $f_{\text{last}} \leftarrow \emptyset$
    $s \leftarrow$ flippedness of t
    while $d_e \notin D$ **do**
      let $d_s = (\mathbf{z}, \mathcal{E}, \mathcal{F}, \mathcal{C}, t)$
      for each face f' of tet t with $f' \neq f_{\text{last}}$ **do**
        if f' fulfills (C3) **then**
          $f \leftarrow f'$
      if f boundary **then**
        abort, leave dart d without connection
      let $\mathbf{g}$ be the transition function over f
      $t \leftarrow$ tet on other side of f
      $d_s \leftarrow (\mathbf{g}(\mathbf{z}), \mathbf{g}(\mathcal{E}), \mathbf{g}(\mathcal{F}), \mathbf{g}(\mathcal{C}), t)$
      $d_e \leftarrow \alpha_i^*(d_s)$
      if $s \neq$ flippedness of t **then**
        swap d_s and d_t
        $s \leftarrow$ flippedness of t
      $f_{\text{last}} \leftarrow f$
    connect dart d to dart in $d' \in D$ with $d' = d_e$

Algorithm 2: *Connection extraction*

4.4 Postprocessing

For perfect parametrizations which do not contain any flipped or degenerate tets, the algorithm has already obtained the final hex mesh at this point. However, flipped tets can lead to several kinds of inconsistencies.

In such cases, properties 3, 5, and 6 are not fulfilled. We observed, however, that the following, weaker properties were fulfilled in all our tests:

Let $\langle S \rangle^+(d) = \{d \in \langle S \rangle(d) \ : \ d \text{ proper}\}$ and $\langle S \rangle^-(d) = \{d \in \langle S \rangle(d) : d \text{ anti}\}$

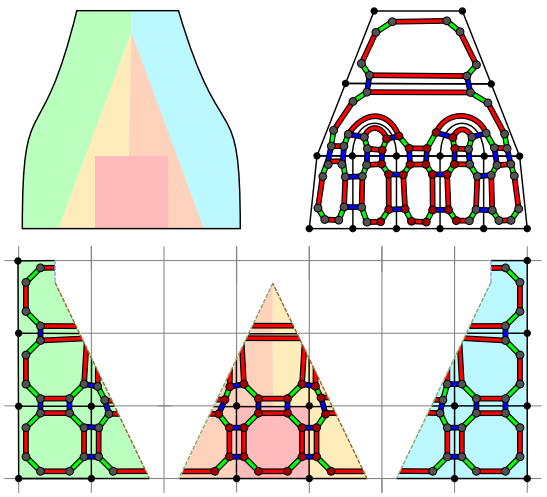


Figure 5: *Top left: Input mesh. Colors are for visual orientation. Bottom: Parametrization. Note that the triangular area is flipped. Top right: extracted darts and α_is.*

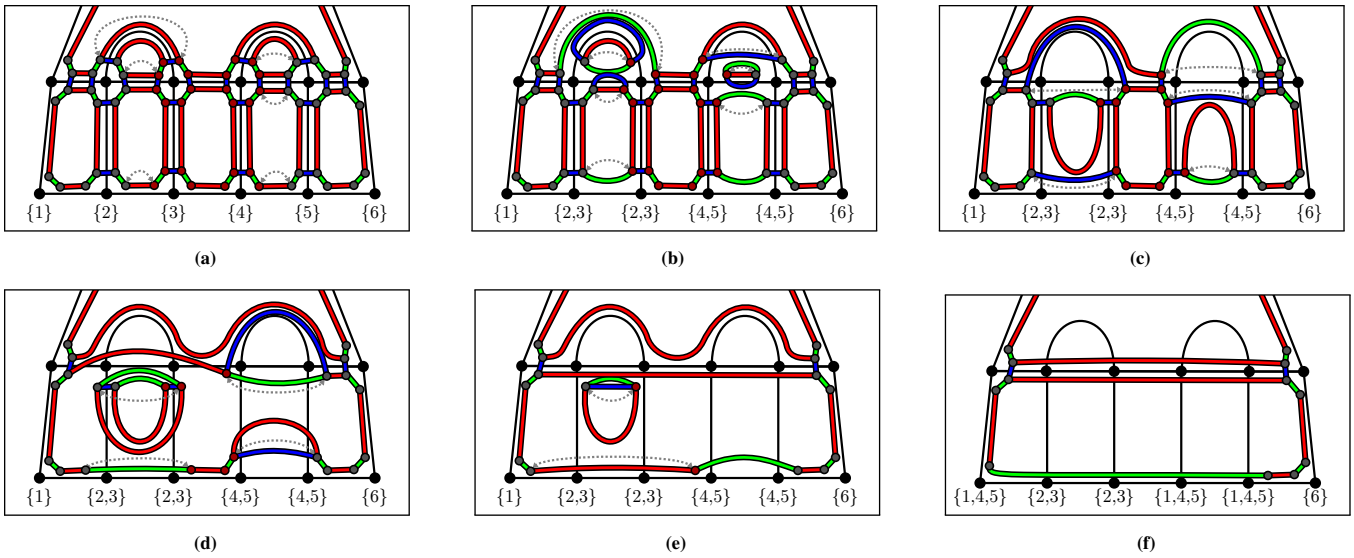


Figure 6: *Dart annihilation process of the mesh from Figure 5. The gray dotted arrow identifies a pair of dart and anti dart that is annihilated in the next step. The sets on the bottom track the equivalence classes of the vertices in the bottom row (cf. section 4.4.2)*

For each dart $d \in D$:

$$|\langle \alpha_0, \alpha_1 \rangle^+(d)| - |\langle \alpha_0, \alpha_1 \rangle^-(d)| = z \cdot \ 8\ , \tag{11}$$
$$|\langle \alpha_1, \alpha_2 \rangle^+(d)| - |\langle \alpha_1, \alpha_2 \rangle^-(d)| = z \cdot \ 6\ , \tag{12}$$
$$|\langle \alpha_0, \alpha_{121} \rangle^+(d)| - |\langle \alpha_0, \alpha_{121} \rangle^-(d)| = z \cdot \ 8\ , \tag{13}$$
$$|\langle \alpha_0, \alpha_1, \alpha_2 \rangle^+(d)| - |\langle \alpha_0, \alpha_1, \alpha_2 \rangle^-(d)| = z \cdot 48\ , \tag{14}$$

with $z \in \mathbb{Z}$ and $\alpha_{121} = \alpha_1 \circ \alpha_2 \circ \alpha_1$.

A 2D example of an extracted mesh is given in Figure 5. While Equations (12) to (14) do not apply in 2D, the extracted mesh contains faces fulfilling Equation (11) for z equal to -1 (bottom middle), 0 (left and right of bottom middle) and 1 (rest). With the postprocessing step, we remove all anti darts and try to make the extracted mesh fulfill Equations (11) to (14) for $z = 1$.

4.4.1 Dart Annihilation

The key idea of dart annihilation is that flipped regions of the parametrization and adjacent regular regions should cancel each other out.

Whenever we find a pair of regular dart d and anti dart d' which are connected via any α_i, we remove these two darts after adjusting the connections of all other darts connected to either d or d' as follows:

$$\alpha_i(\alpha_i(d)) = \alpha_i(d')\ ,$$
$$\alpha_i(\alpha_i(d')) = \alpha_i(d)$$

for all i where $\alpha_i(d) \neq d'$.

After all pairs of regular darts and anti darts are annihilated, each connected component consists of only proper darts or only anti darts. In practice, the only darts left are typically regular ones. Only if the majority of the parametrization is flipped will there be more anti darts than regular darts. Since annihilation removes one dart of each type, only anti darts will be left in this case. While this is no problem for our algorithm, it is probably a sign that the input parametrization is erroneous. For the remainder, we will assume that only regular darts are left. Thus, $|\langle S \rangle^-(d)| = 0$ for any S and d. Therefore, Equations (11) to (14) become:

$$|\langle \alpha_0, \alpha_1 \rangle(d)| = n \cdot \ 8\ , \tag{15}$$
$$|\langle \alpha_1, \alpha_2 \rangle(d)| = n \cdot \ 6\ , \tag{16}$$
$$|\langle \alpha_0, \alpha_{121} \rangle(d)| = n \cdot \ 8\ , \tag{17}$$
$$|\langle \alpha_0, \alpha_1, \alpha_2 \rangle(d)| = n \cdot 48\ , \tag{18}$$

with $n \in \mathbb{N}$ and $n > 0$ since $d \in \langle S \rangle(d)$ for any S and d.

In Figure 6, the dart annihilation process is demonstrated for the mesh of Figure 5.

4.4.2 Vertex Merging

As the last step of our algorithm, we refine the geometric embedding of the output mesh. As already mentioned above, flipped and degenerate tets cause several vertices to be extracted for the same integer grid point. We want to merge those vertices such that only one representative is left which is assigned a new geometric embedding based on the originally extracted vertices.

For the merging step, we first set up equivalence classes for the vertices. Initially, each vertex is in its own class. Then, we check for each vertex which is incident to a degenerate tet t whether a vertex was extracted for the same integer grid location on another face, edge or vertex of t. If so, we merge the equivalence classes of the two vertices.

Additionally, to eliminate duplicate vertices caused by flipped tets, we perform an additional check during the dart annihilation step. Whenever we find a pair of dart and anti dart connected via an α_0, we merge the two equivalence classes of the vertices referenced by the two darts.

For the final equivalence classes we create a new vertex for each class replacing its original vertices. If all vertices are inner vertices, we simply choose their center of gravity as the geometric embedding of the newly created representative. If there are boundary vertices in the equivalence class, i. e. vertices generated from a boundary vertex, edge or face, we use a different strategy only considering these vertices. For each boundary vertex, we collect all boundary

faces incident to the generating entity. We calculate a new position as the point closest to all planes defined by these boundary faces. Since this new position may be far off the original positions, we chose the position of the boundary vertex of the equivalence class which is closest to the calculated point. Thus, we ensure that the geometric embedding of the new representative lies on a meaningful position on the input mesh's boundary.

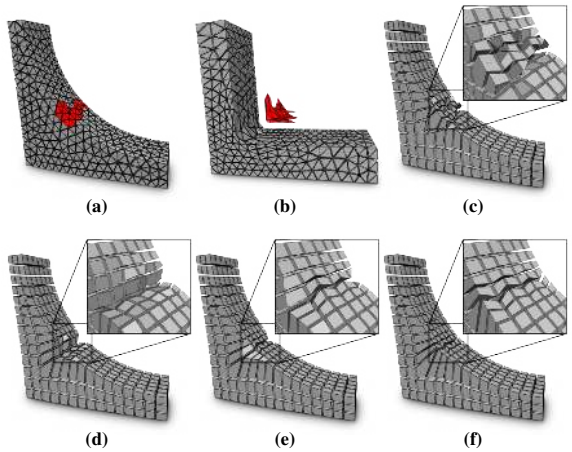


Figure 7: *Comparison of different merging positions. (a): Input mesh* FAN PART. *(b): Parametrization. Flipped tets are highlighted in red. (c): Extracted mesh before post processing. (d)–(f): Merged vertices positioned at center of gravity, center of gravity of boundary vertices only, and closest point to all incident boundary planes.*

5 Results

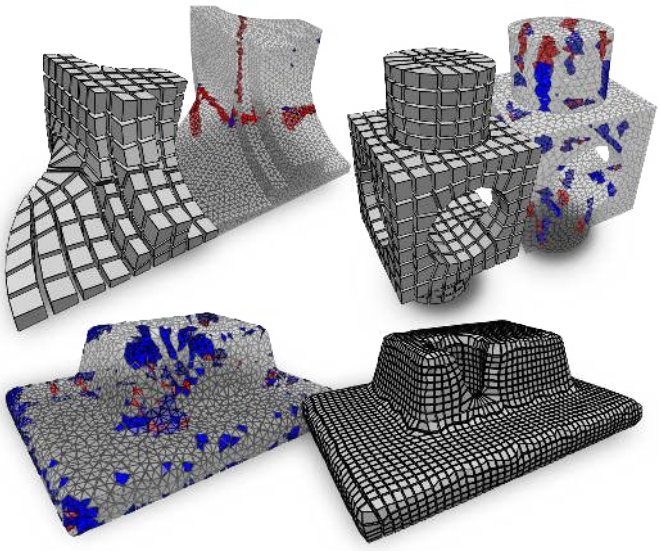


Figure 8: *Results of HexEx for* FANDISK, BLOCK *and* DRILLED HOLE. *Tets that are flipped (red) or degenerate (blue) in parameter domain are highlighted.*

We use our algorithm to extract hex meshes on a variety of different meshes and parametrizations that we obtained from an implementation based on work by Nieser et al. [2011] and Ray and Sokolov [2015] and summarize the results in this section. Some statistics for these meshes are given in Table 1. Even though all parametrizations contained imperfections, our algorithm was able to extract meshes free of flipped hexahedra and non-hexahedral cells in most cases. Typically, a hex mesh optimization algorithm, such as that by Livesu et al. [2015], is applied on an extracted hex mesh to optimize its element quality. Unless stated otherwise, we did not apply any such optimization on the extracted hex mesh and show our results as they are produced by our algorithm.

Table 1: *Statistics of example models: Number of tets in input mesh (# T), number of tets with flipped parametrization (# Flip), number of tets with degenerate parametrization (# Deg), number of hexahedral cells in output mesh (# H), and timings (t) on an i5 CPU @ 3.3GHz.* BUNNY 1 *and* 2 *refer to the initial and randomized parametrizations, respectively.*

Model	# T	# Flip	# Deg	# H	t
CYLINDER	90371	558	376	5160	6.8s
FAN PART	4137	48	0	468	0.5s
FANDISK	75309	470	21	244	2.2s
BLOCK	38779	210	580	648	1.7s
DRILLED HOLE	17299	56	1021	11347	7.9s
SPHERE	61299	30	196	490	2.1s
TETRAHEDRON	32768	23	44	192	0.9s
TORUS	333824	7200	825	124412	99s
BUNNY 1	65766	44	28	16612	14s
BUNNY 2	65766	29286	28	16612	36s
ELK	233930	38001	9962	100706	102s

In Figure 9, we show a common artifact also described by Jiang et al. [2014]. Often, curves of the singularity graph passing along the boundary occasionally sink into the inside leading to degenerate tets and thus cause dents in the surface of the extracted mesh. Our vertex merging strategy creates a new vertex for the duplicated vertices that occur in this scenario and ensures that its position lies on the surface of the original mesh leading to a faithful boundary approximation

When two parallel curves of the singularity graph intersect at a vertex, they are forced to collapse onto each other in parametric space. This leads to many clustered flipped tetrahedral cells between the two curves as seen in Figures 1 and 12. Even for these complicated cases, our algorithm extracts a meaningful mesh in which the two singular curves are merged into a single one.

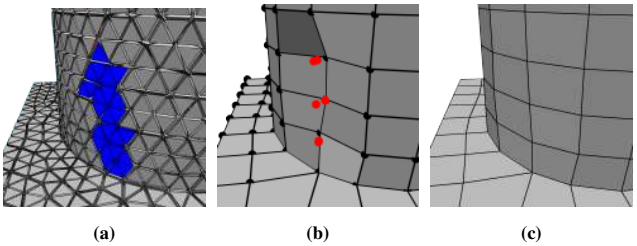


Figure 9: *Close-up of* BLOCK. *Marked tets in (a) have a degenerate parameter image. This leads to poor boundary alignment in (b). After merging with the h-vertices marked red, the boundary of the output mesh aligns perfectly with that of the input mesh in (c).*

5.1 Stress test

In order to test the robustness of our algorithm, we set up the following stress test. Given the initial parametrization of BUNNY shown

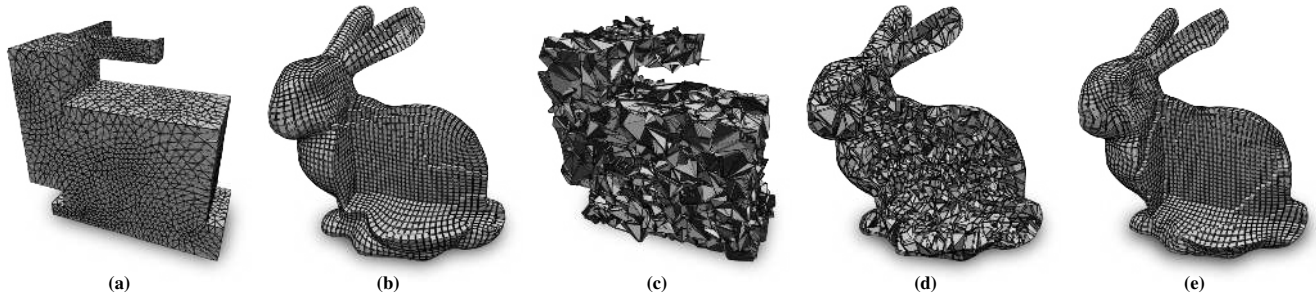


Figure 10: *(a) Polycube parametrization of* BUNNY. *(b) Resulting hex mesh. (c) Randomized parametrization constructed by moving vertex parameters by a random value between -2 and 2 in each direction while preserving the original polycube boundary constraints. (d) Resulting hex mesh shows a lot of distortion but the same topology as (b). (e) A few smoothing steps reveal the correct topology.*

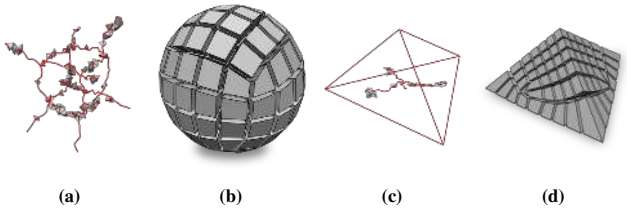


Figure 11: *(a) and (c) The singularity graph and flipped tets of* SPHERE *and* TETRAHEDRON. *(b) and (d) The extracted all-hex meshes.*

Figure 12: *(a) Input mesh* TORUS *for which the singularity graph and all flipped or degenerate tets are shown. (b) Our algorithm extracted an all-hex mesh where the touching singular curves are combined into a single twisted valence two singularity. We applied 5 iterations of setting each vertex to the center of gravity of its incident vertices to untangle the inner cells.*

in Figure 10a, we perturbed each vertex parameter by a random value between -2 and 2 in each direction while fulfilling the original boundary conditions, i. e. boundary vertices remained either in a plane or a line (the intersection of two planes) or a vertex (intersection of three planes). This causes about half of the tetrahedra to flip and leads to the extraction of many more darts than for the original parametrization. However, after the annihilation process the number of remaining darts was identical to that of the undistorted case. Even though the geometric embedding is quite distorted due to the randomization the extracted mesh has the correct topology as seen in Figure 10e.

We observed the same robustness for all parametrizations without inner singularities. This suggests that our algorithm is perfectly suited for polycube like parametrizations [Gregson et al. 2011; Livesu et al. 2013; Huang et al. 2014].

On meshes with inner singularities, our algorithm is not as robust towards such random distortions. Performing the stress test on SPHERE (Figure 11) led to the extraction of non-hex elements if the displacement was larger than 0.4 units. We discuss these failures in the following section.

5.2 Failure cases

Unfortunately, our algorithm is not able to handle all possible configurations of flipped tetrahedra. We identified two problematic cases.

Split vertices Further challenges appear, when Equations (15) to (18) are not fulfilled for $n = 1$. This can happen when the tets around an edge are flipped in such a way that the sum of dihedral angles (where the dihedral angle of flipped tets is considered negative) around that edge is 2π less than originally intended. This may change, for example, an inner valence 5 edge to a valence 1 edge. In this case, our algorithm extracts non-hexahedral cells as shown in Figure 13a.

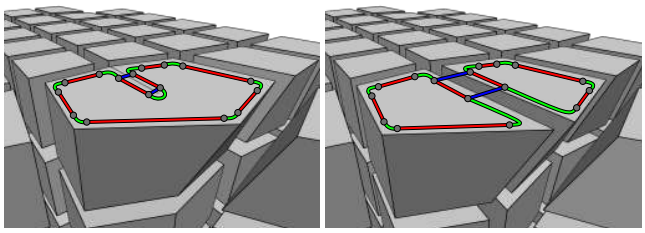
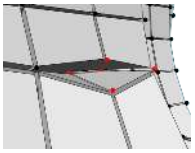


Figure 13: *(a) Non-hexahedral cell caused by inner valence 1 edge. (b) Reconnecting darts resolves the problem.*

Ebke et al. [2013] observed the same behavior in the 2D case and we refer to Figure 5 in their paper for an example parametrization leading to this artifact. They call this phenomenon “lost q-ports”, where a q-port is similar to a 2D dart, because of the 5 expected edge intersections with the input mesh, only 1 is present, while the others are “lost”. We note, however, that the extracted darts allow the representation of a valid all-hex mesh by changing some of the α pointers (cf. Figure 13b where we reconnected some pointers by hand). Therefore, we would like to reinterpret this behavior as a “split vertex”, where some of the darts are extracted at one place and the rest at another. This assumption is further strengthened by the results of the stress test. For all randomized parametrizations, the number of darts left after annihilation was constant. We therefore conjecture that in these cases, the darts can always be reconnected in a way which yields a valid all-hex configuration. Finding a robust and general reconnection strategy is an interesting topic for future research.

Partial connectivity Currently, our algorithm does not guarantee that Equation (1) and Equation (2) are fulfilled. Figure 14 shows an example where a bad parametrization of the brown and purple boundaries leads to two faces only being partially connected. Similar configurations are possible in 3D and lead to partially connected cells shown in the inset.



Both examples can easily be fixed. One has to find the unconnected darts, connect them, and merge the two vertices. Unfortunately, there are configurations which are not trivial to fix. Consider again the example in Figure 5 but with the red area removed in both input and parametrization. The initial extraction shown in Figure 15a is similar to the original but the eight anti darts extracted for the red area are missing. Without these darts, the annihilation process converges with more darts left than before (Figure 15b) and leads to a mesh with invalid connectivity, since both lower faces are partially connected to the upper face. Obviously, the upper face does not have enough darts to be fully connected to both lower faces. Instead of trying to connect the unconnected darts, an easy way to fulfill Equation (1) could be to disconnect the pairs of darts that form the partial connection. However, this may be problematic for the application using the hex mesh. Alternative solutions might incorporate splitting faces or cells to add more darts which then allow to complete the partial connection.

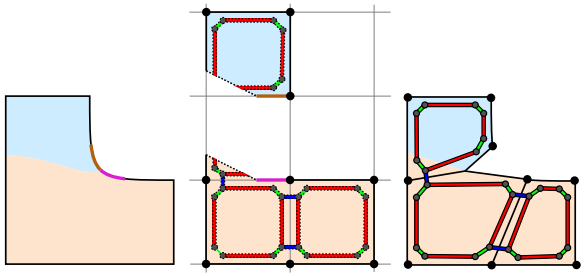


Figure 14: *Configuration with partial adjacency. (a) Input mesh. (b) Parametrization. (c) Invalid output mesh.*

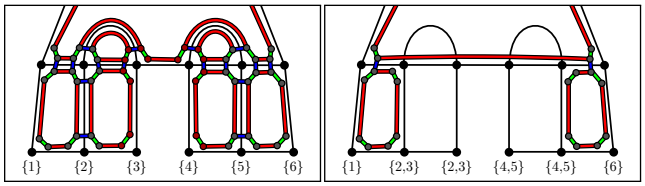


Figure 15: *(a) Initially extracted dart structure for input from Figure 5 with the red areas removed. (b) Darts after annihilation. Note that both lower remaining faces are only connected via one of the two darts of the upper edge.*

Volume Collapses As already discussed earlier, undesired intersections between arcs of the singularity graph force entire regions of the mesh onto the same parameter, causing many tetrahedra to flip or degenerate in the parametric domain. While our algorithm often is able to handle the resulting parametrization, this is not the case if a complete volume between two boundaries collapses. In Figure 16, we show an example where the antlers and the back left wheel of ELK are collapsed into a sheet and a point, respectively. In such cases, all darts in these regions are eliminated during the annihilation process and no hexahedral cells remain. If, however, only a subvolume collapses, while the boundary around it remains on reasonable parameter locations, our algorithm is able to extract a meaningful mesh, e. g. the front left wheel of the elk.

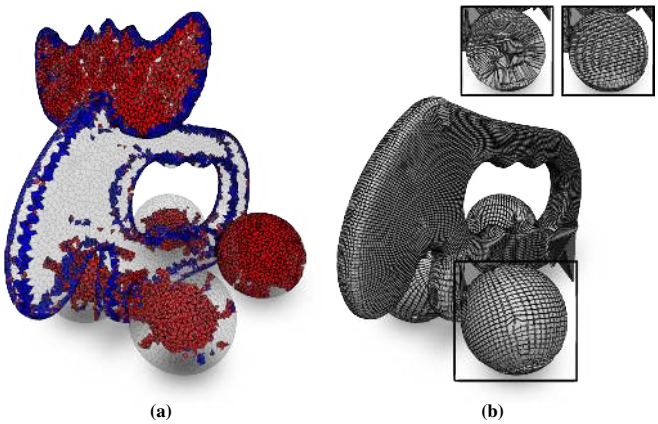


Figure 16: *(a) Tetrahedral input mesh* ELK. *Tets which are degenerate in the parametrization are shown in blue, flipped ones in red. (b) Extracted mesh with some parts missing due to their complete collapse in the parametric domain. Top boxes show an inside view of the front wheel before (left) and after (right) smoothing.*

6 Discussion

While we cannot give any formal proofs and guarantees we empirically found that our algorithm can robustly handle the following cases.

- A "perfect" parametrization (which may contain numerical errors on double precision, but no flips or degeneracies) always leads to a valid hex mesh due to our robust tracing using exact predicates only.
- The very common case of small regions of degenerate and inverted tets caused by incompatible singularity types of different edges of the same tet as described by Li et al. [2012] or zigzags as described by Jiang et al. [2014] are handled well without the need of explicitly editing the singularity graph or modifying the mesh (Figure 11). The same is true for singularity curves touching the surface as described by Jiang et al. [2014] (Figure 9).
- Polycube parametrizations, which do not contain inner singularities, succeed even when many defects are present (Figure 10).
- Larger regions of degenerate or flipped tets caused by touching curves in the singularity graph such as CYLINDER (Figure 1) or TORUS (Figure 12) are handled correctly unless one of the following problematic cases occurs.

We identified three scenarios which prevent our algorithm from extracting a valid hex mesh.

- If the extracted data structure contains darts which satisfy Equations (11) to (14) for a k other than -1, 0 or 1, or not at all[1], our algorithm will extract non-hex cells (Figure 13). For these cases, we conjecture that a reconnection of darts is always possible to form a valid all hex-mesh, and we hope such a reconnection strategy can be found in the future.

[1] Figure 5c in [Ebke et al. 2013] shows such a parametrization in 2D.

- Flipped tets at the boundary may cause a partial connectivity between adjacent hexahedral cells. Often, these issues can easily be resolved as described in 5.2. While we have artificially constructed a parametrization which does not allow an easy fix, we have not encountered such a case in practice.

- When integer or alignment constraints induced by boundaries and singularities force large regions of the parametrization to invert or to collapse (cf. Figure 16), these regions will sometimes not create any hexahedra and thus leave parts of the input model uncovered. While this behavior is suboptimal from a meshing point of view, it is clear that the resulting mesh is actually consistent with the parametrization. Hence, such large collapses need to be addressed during the earlier frame field and parametrization stages. Repairing them during the mesh extraction phase would in general require global changes and thus deliver a mesh that is no longer consistent with valid parts of the parametrization.

7 Conclusion

We presented a definition for generalized maps based on tuples of vertices, edges, faces and cells, bridging the gap between explicit and implicit mesh data structures. This allowed us to provide an intuitive criterion for when to extract a dart for a given parametrization and how to build the connections between those darts. Based on these concepts, we presented a mesh extraction algorithm which is extremely robust towards imperfections in the parametrization such as flipped or degenerate tetrahedra. While we motivated the algorithm in 3D, due to the n-dimensional nature of generalized maps the presented concepts are easily transferable to 2D, as well as to higher dimensions.

By providing a publicly available C++ implementation of HexEx we hope to accelerate the progress of developing new parametrization algorithms by relieving researchers from one of the involved tasks of hex meshing. Thanks to the robustness of our algorithm, a larger set of parametrizations is now applicable to hex meshing. It might even facilitate the development of parametrization algorithms that instead of trying to prevent flipped elements purposefully place them to achieve a desired effect. For example, Ebke et al. were able to exploit the robustness of QEx to extract high quality quad-meshes from noisy input data [Ebke et al. 2014]. Similar applications are imaginable for HexEx.

Our main goal for future research is to identify a set of constraints that can be added on top of Constraints (A1) to (A3), yet are not as strict as Constraint (A4), such that a valid hex mesh can be guaranteed by our algorithm.

Acknowledgments

This research has received funding from the European Research Council under the European Unions Seventh Framework Programme (ERC grant agreement 340884 “ACROSS”) and the German Research Foundation (DFG, Gottfried-Wilhelm-Leibniz Programm, and grant GSC 111, Aachen Institute for Advanced Study in Computational Engineering Science). We would like to thank Jan Möbius for creating and maintaining the geometry processing framework *OpenFlipper* [Möbius and Kobbelt 2012], Hans-Christian Ebke and Marcel Campen for fruitful discussions, as well as the reviewers for their helpful feedback. Some of the models are courtesy of the AIM@SHAPE Repository.

References

BOMMES, D., CAMPEN, M., EBKE, H.-C., ALLIEZ, P., AND KOBBELT, L. 2013. Integer-grid maps for reliable quad meshing. *ACM Transactions on Graphics 32*, 4 (July).

EBKE, H.-C., BOMMES, D., CAMPEN, M., AND KOBBELT, L. 2013. QEx: Robust quad mesh extraction. *ACM Transactions on Graphics 32*, 6 (Nov.).

EBKE, H.-C., CAMPEN, M., BOMMES, D., AND KOBBELT, L. 2014. Level-of-detail quad meshing. *ACM Transactions on Graphics 33*, 6 (Dec.).

GREGSON, J., SHEFFER, A., AND ZHANG, E. 2011. All-hex mesh generation via volumetric polycube deformation. *Computer Graphics Forum 30*, 5.

HUANG, J., JIANG, T., SHI, Z., TONG, Y., BAO, H., AND DESBRUN, M. 2014. ℓ1-based construction of polycube maps from complex shapes. *ACM Trans. Graph. 33*, 3 (June).

JIANG, T., HUANG, J., WANG, Y., TONG, Y., AND BAO, H. 2014. Frame field singularity correction for automatic hexahedralization. *IEEE Transactions on Visualization and Computer Graphics 20*, 8.

KRAEMER, P., UNTEREINER, L., JUND, T., THERY, S., AND CAZIER, D. 2014. CGoGN: n-dimensional meshes with combinatorial maps. In *Proceedings of the 22nd International Meshing Roundtable*.

KREMER, M., BOMMES, D., AND KOBBELT, L. 2012. OpenVolumeMesh – a versatile index-based data structure for 3d polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable*.

LI, Y., LIU, Y., XU, W., WANG, W., AND GUO, B. 2012. All-hex meshing using singularity-restricted field. *ACM Transactions on Graphics 31*, 6 (Nov.).

LIVESU, M., VINING, N., SHEFFER, A., GREGSON, J., AND SCATENI, R. 2013. PolyCut: Monotone graph-cuts for polycube base-complex construction. *Transactions on Graphics (Proc. SIGGRAPH ASIA 2013) 32*, 6.

LIVESU, M., SHEFFER, A., VINING, N., AND TARINI, M. 2015. Practical hex-mesh optimization via edge-cone rectification. *ACM Trans. Graph. 34*, 4 (July).

LYON, M., BOMMES, D., AND KOBBELT, L., 2016. libHexEx: a robust hexahedral mesh extraction library. http://www.graphics.rwth-aachen.de/software/libHexEx.

MÖBIUS, J., AND KOBBELT, L. 2012. OpenFlipper: An open source geometry processing and rendering framework. In *Curves and Surfaces*, vol. 6920 of *Lecture Notes in Computer Science*.

NIESER, M., REITEBUCH, U., AND POLTHIER, K. 2011. CubeCover – parameterization of 3D volumes. *Comp. Graph. Forum*.

RAY, N., AND SOKOLOV, D. 2015. On smooth 3D frame field design. *Computing Research Repository*.

SHEPHERD, J. F., AND JOHNSON, C. R. 2008. Hexahedral mesh generation constraints. *Engineering with Computers 24*, 3.

SHEWCHUK, J. R. 1997. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18*, 3 (Oct.).

SHIMADA, K. 2006. Current trends and issues in automatic mesh generation. *Computer-Aided Design & Applications 3*, 6.