# Fast Hexahedral Mesh Extraction from Locally Injective Integer-Grid Maps

**Bachelor thesis**
in the
**Computer Graphics Group**
**of the University of Bern**

*Author*
Tobias Kohler

*Supervisors*
Prof. Dr. David Bommes
Martin Heistermann

# Abstract

Integer-Grid Maps as boundary-aligned volume parametrizations of tetrahedral meshes onto a voxel grid have proven to be a valuable approach in the field of hex-meshing as they can induce highly structured hexahedral meshes. The decomposition of the map into multiple cell charts due to the presence of singularities and non-identity transitions between adjacent charts makes the process of extracting a hex-mesh from an IGM non-trivial. Additionally, inverted and degenerate charts in the parametrization may cause inconsistencies that need to be fixed in a postprocessing step. As most current methods work without such defects in the IGM, we present a specialized hex-extraction algorithm that expects all cell charts to have a strictly positive volume and aims to be as fast as possible. To achieve this, we apply common rasterization techniques to efficiently extract the hex-mesh's geometry and use a specifically designed data structure called propeller to extract its topology.

# Contents

# 1 Introduction

Parametrization based hex-meshing is a popular and actively researched topic. While tetrahedral meshes are easier to generate, hexahedral meshes are often preferred due to their structured nature and use in advanced numerical simulations (Reberol et al. 2023). Therefore, it is natural to want to transform a tetrahedral mesh into a hexahedral mesh. (Lyon et al. 2016) presented *HexEx*, a robust hex-extraction algorithm, with a tet-mesh and a parametrization as input, which looks for and resolves defects in the input such as degeneracies and foldovers. Due to this, optimizations that would be possible in the defect-free case are not applied.

Our method builds on HexEx and expects the input to be defect-free, namely locally injective. The preprocessing of HexEx which resolves floating-point inaccuracies is kept as-is.

The actual extraction consists of two phases which we aim to optimize: 1) Finding intersections of the parametrization with integer-grid points to extract hex-vertices (geometry extraction) and 2) tracing from each hex-vertex through potentially multiple charts of tet-cells in the parametrization along integer iso-lines to adjacent hex-vertices while also considering intersections of iso-surfaces with the parametrization that induce the hex-faces (topology extraction).

We reimplement these phases from scratch and optimize them in two main ways. First, where the original algorithm extracts hex-vertices by testing each point in the bounding box of an element, we use rasterization techniques. Second, we replace the costly dart data structure, which brings with it a lot of redundancy, with the more efficient propeller data structure.

Our version is implemented in C++ besides the original HexEx such that the user is able to easily switch between versions, in case, the original algorithm is desired. To use HexEx, there is both a command line application and an OpenFlipper (Möbius and Kobbelt 2012) Plugin.

# 2 Theoretical Foundation

## 2.1 Geometry

A subset of an Euclidean space is *convex* if each line segment given by two endpoints in the subset is in its entirety contained in the subset.
A point in a convex set is an *extreme point* if it does not lie on any open line segment given by two endpoints in the set.
The *convex hull* conv() of a subset of Euclidean space is the smallest convex set that contains the subset.
If not specified otherwise, we consider elements within the 3-dimensional Euclidean space $\mathbb{R}^3$.
A finite set of points $\{x_1, ..., x_d\}$ is *affinely dependent* if there exist real numbers $\lambda_1, ..., \lambda_d$ which satisfy $\lambda_1 + ... + \lambda_d = 0$ such that $\lambda_1 x_1 + ... + \lambda_d x_d = \mathbf{0}$. Otherwise it is called *affinely independent*.
The (affine) *dimensionality* of a set $X$ is $d \in \mathbb{N}_0$ if a maximal affinely independent subset of $X$ contains exactly $d + 1$ points (Grünbaum and Shephard 1969).
A (geometric and convex) *d-polytope* is the convex hull of finitely many points where $d$ denotes its dimensionality (Grünbaum and Shephard 1969). Then, each $1-$polytope has exactly two extreme points and each $d$-polytope, for $d > 1$, has at least $d + 1$ extreme points. $0-$polytopes are simply points, $1-$polytopes are line segments, $2-$polytopes are (convex) polygons and $3-$polytopes are (convex) polyhedra.

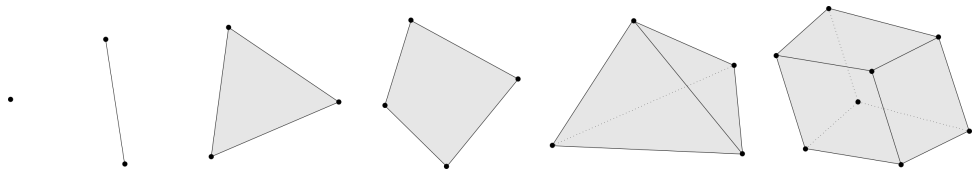


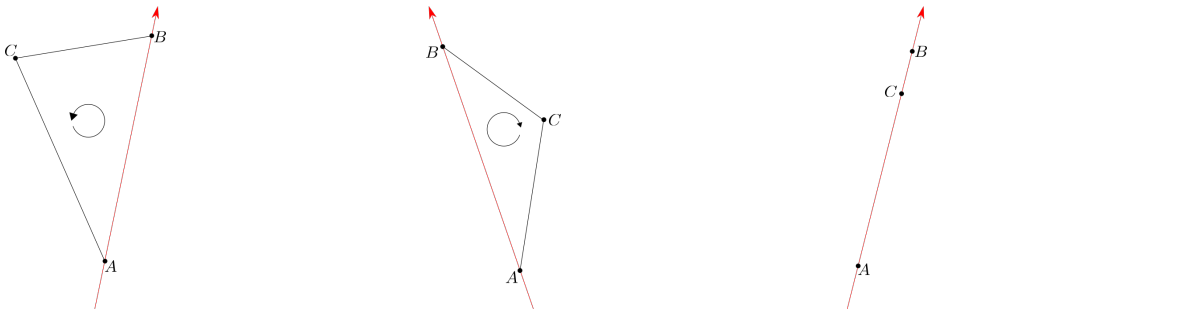
**Figure 1:** *From left to right: Polytopes with increasing dimensionality and extreme points: Point, line segment, triangle, quadrilateral, tetrahedron, hexahedron*

A *d-manifold* (with boundary) is a subset of the Euclidean space for which each internal point is locally homeomorphic to an open $d-$ball and each boundary point is locally homeomorphic to an open $d-$half-ball (Kremer et al. 2013).

## 2.2 Orientations

The *signed area* of a triangle given by three (ordered) points $A$, $B$, $C \in \mathbb{R}^2$ is

$$\text{area}(A, B, C) = \frac{1}{2} \cdot \begin{vmatrix} (A_x - C_x) & (A_y - C_y) \\ (B_x - C_x) & (B_y - C_y) \end{vmatrix} \in \mathbb{R}$$




**(a)** *ORI2D*$(A, B, C) > 0$ **(b)** *ORI2D*$(A, B, C) < 0$ **(c)** *ORI2D*$(A, B, C) = 0$

**Figure 2:** *Orientations of three points in 2D*

and their *orientation*

$$\text{ORI2D}(A,\, B,\, C) = \text{sign}(\text{area}(A,\, B,\, C)) \in \{-1,\, 0,\, 1\}$$

Geometrically, three points are oriented positively if they are ordered counterclockwise or $C$ lies to the left of $\vec{AB}$ (Fig. 2a). They are oriented negatively if they are ordered clockwise or equivalently $C$ lies to the right of $\vec{AB}$ (Fig. 2b). Their orientation is zero if $C$ lies on the line $\vec{AB}$ or equivalently the vectors $\vec{AB}$ and $\vec{BC}$ are collinear (Fig. 2c).

Similarly, in 3D space, the *signed volume* of a tetrahedron given by four (ordered) points $A,\, B,\, C,\, D \in \mathbb{R}^3$ is

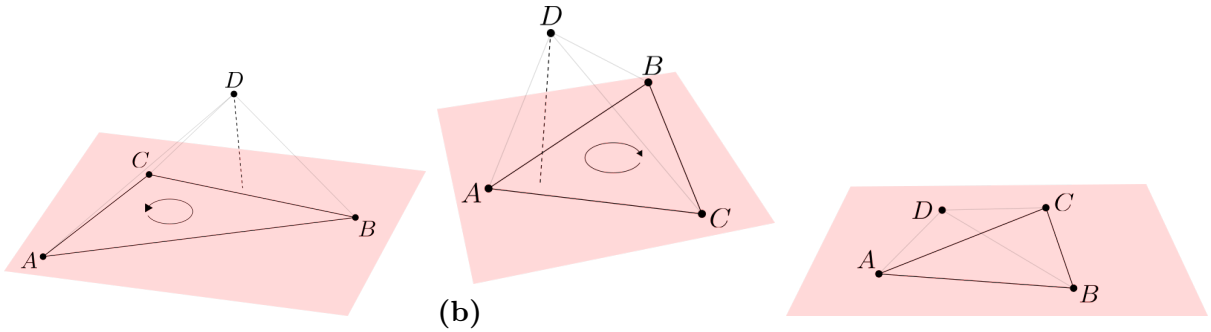
$$\text{volume}(A,\, B,\, C,\, D) = \frac{1}{6} \cdot \begin{vmatrix} (A_x - D_x) & (A_y - D_y) & (A_z - D_z) \\ (B_x - D_x) & (B_y - D_y) & (B_z - D_z) \\ (C_x - D_x) & (C_y - D_y) & (C_z - D_z) \end{vmatrix} \in \mathbb{R}$$

and their *orientation*

$$\text{ORI3D}(A,\, B,\, C,\, D) = \text{sign}(\text{volume}(A,\, B,\, C,\, D)) \in \{-1,\, 0,\, 1\}$$

Geometrically, four points are oriented positively if the first three are ordered counterclockwise when looking from $D$ (Fig. 3a). They are oriented negatively if the first three are ordered clockwise when looking from $D$ (Fig. 3b). Their orientation is zero if $D$ lies on the plane given by $A,\, B,\, C$ (Fig. 3c).

To efficiently and reliably compute these orientations with floating-point numbers, HexEx uses *exact predicates* provided by (Shewchuk 1996). From the orientation checks ORI2D and ORI3D, further checks are derived by (Lyon et al. 2016) which include ONLINE-SEGMENT, INTRIANGLE and INTETRAHEDRON.




**(a)** $ORI3D(A, B, C, D) > 0$ **(b)** $ORI3D(A, B, C, D) < 0$ **(c)** $ORI3D(A, B, C, D) = 0$

**Figure 3:** *Orientations of four points in 3D*

## 2.3 Mesh

A volumetric *mesh* is a 3-dimensional *CW complex* $\mathcal{M} = (V, E, F, C)$, consisting of *vertices* $V$ (0−elements), *edges* $E$ (1−elements), *faces* $F$ (2−elements) and *cells* $C$ (3−elements) (Pietroni et al. 2022). These $d$−elements are a generalization of geometric polytopes to topological entities.

An edge $e \in E$ is identified by two vertices $e = \{v_1, v_2\}$. Analogously, a face $f \in F$ is identified by a sequence of vertices $f = (v_1, ..., v_k)$ where $f$ is bounded by edges $e_i = (v_i, v_{i+1})$ for $0 < i < k$ and $e_k = (v_k, v_1)$ (Lyon et al. 2016). Analogously, a cell $c \in C$ is identified by a sequence of vertices $c = (v_1, ..., v_k)$[1] and bounded by faces.

If a $d$−element is entirely part of the boundary of a $d'$−element, $d \neq d'$, they are said to be *incident*. Two distinct $d$−elements, $1 \leq d \leq 3$ are *adjacent* if they share a common $(d-1)$−element on their boundary. Two distinct vertices are adjacent if they are incident to a common edge (Kremer et al. 2013; Lyon et al. 2016).

We denote incidency between two elements by "$\sim$" and write $x < y$ if the dimensionality of $x$ is smaller than the dimensionality of $y$.

A *boundary* face is a face that is incident to less than two cells and a boundary edge or vertex is an edge or vertex which is incident to at least one boundary face.

The geometry of the mesh is given by a *geometric embedding* $\boldsymbol{g} : V \to \mathbb{R}^3$ which maps each vertex to a position in Euclidean space (Lyon et al. 2016).

A *tetrahedral mesh* consists entirely of tetrahedral cells, referred to as *tets*, i.e. cells bounded by four triangular faces. Conventionally, the four vertices of a tet are oriented positively according to Section 2.2.

A *hexahedral mesh* consists entirely of hexahedral cells, referred to as *hexes*, i.e. cells bounded by six quadrilateral faces.

A mesh is a *manifold* if the union of its elements is a manifold.

We consider tetrahedral manifold meshes for which a $d$−element corresponds to a geometric convex $d$−polytope as defined in Section 2.1.

[1] The order of the vertices that define a cell are often not unique and depend on varying conventions.

### 2.3.1 Implementation

The data structure that is used to store our input mesh, which is a tetrahedral manifold mesh, is *OpenVolumeMesh (OVM)* as provided by (Kremer et al. 2013). OVM extends the concepts of OpenMesh (Botsch et al. 2002), like separating edges into two directed half-edges (Campagna et al. 1998), to the third dimension. Each face is split into a pair of half-faces with opposing orientations, each incident to up to one cell. If a half-face is incident to a tet-cell, we assume its vertices to be oriented positively with respect to the fourth vertex of the tet.
We denote the set of half-edges by $E^{\frac{1}{2}} = \{(v, e) \in V \times E : v \sim e\}$ and the set of half-faces by $F^{\frac{1}{2}} = \{(f, c) \in F \times C : f \sim c\}$.

## 2.4 Integer-Grid Map

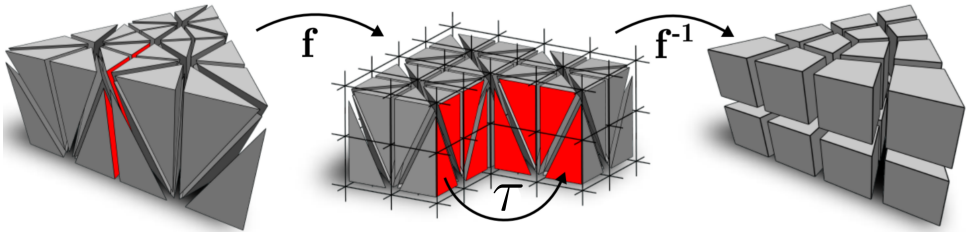



**Figure 4:** *A tet-mesh is mapped onto a 3D integer grid which implies the structure of the hex-mesh by the preimage of the map. To allow for a resulting singular hex-edge of valence 3 in the center, the mesh is cut open around it.*

*Source: (Lyon et al. 2016)*

A *3D integer-grid map (IGM)* $\boldsymbol{f}$ is the union of per-cell linear maps $\boldsymbol{f}_{c_i} : \mathbb{R}^3 \to \mathbb{R}^3$ which map the vertex positions $(\boldsymbol{p_i}, \boldsymbol{q_i}, \boldsymbol{r_i}, \boldsymbol{s_i})$ of each tet $c_i$ to *parameters* $(\boldsymbol{u_i}, \boldsymbol{v_i}, \boldsymbol{w_i}, \boldsymbol{x_i})$ in the parametric domain. For shorter notation, we write
$\boldsymbol{f}_c(v)$ for $\boldsymbol{f}_c(\boldsymbol{g}(v))$ and $\boldsymbol{f}_c(v_1, ..., v_k)$ for $(\boldsymbol{f}_c(v_1), ..., \boldsymbol{f}_c(v_k))$.
Given two adjacent tets $c_i$ and $c_j$, the parameter function $\boldsymbol{f}$ might have different values in both charts. They are related by the *transition functions* $\boldsymbol{\tau}_{ij} : \mathbb{R}^3 \to \mathbb{R}^3$ as illustrated in Fig. 4 which are automorphisms that map the chart of a tet $c_i$ to the chart of the adjacent tet $c_j$, i.e. $\boldsymbol{f}_{c_j}(v) = \boldsymbol{\tau}_{ij}(\boldsymbol{f}_{c_i}(v))$ for each of the three common vertices $v$. It follows that $\boldsymbol{\tau}_{ji} = \boldsymbol{\tau}_{ij}^{-1}$.
The sum of parametric dihedral face angles around an edge is always an integer multiple of $\frac{\pi}{2}$. If this angle is $2\pi$ around an inner edge or $\pi$ around a boundary edge, the edge is defined to be *regular*. Otherwise, the edge is called *singular* (Brückler et al. 2022).
A vertex is defined to be singular if it is incident to one, three or more singular edges (Lyon et al. 2016).

As presented similarly, for example, in (Brückler et al. 2022; Liu et al. 2018; Lyon et al. 2016), the 3D integer-grid map must satisfy the following four constraints:

(IGM1) *Conformity* The transition functions $\boldsymbol{\tau}_{ij}$ are of the form

$$\boldsymbol{\tau}_{ij}(\boldsymbol{u}) = \boldsymbol{R}_{ij}(\boldsymbol{u}) + \boldsymbol{t}_{ij}$$

where $\boldsymbol{R}_{ij} \in \mathrm{Oct}$ is one of the 24 orientation-preserving octahedral permutations (Solomon et al. 2017) and $\boldsymbol{t}_{ij} \in \mathbb{Z}^3$ is an integer translation. Note that $\boldsymbol{R}_{ji} = \boldsymbol{R}_{ij}^{-1}$ and $\boldsymbol{t}_{ji} = \boldsymbol{R}_{ji}(-\boldsymbol{t}_{ij})$.

(IGM2) *Boundary alignment* Boundary faces are mapped to regions on integer planes.

$$\forall f \in \partial F : \exists z \in \mathbb{Z},\, a,\, b,\, c \in \mathbb{R}^2,\, \boldsymbol{R} \in \mathrm{Oct} : \boldsymbol{f}(f) = (\boldsymbol{R} \begin{bmatrix} z \\ a \end{bmatrix},\, \boldsymbol{R} \begin{bmatrix} z \\ b \end{bmatrix},\, \boldsymbol{R} \begin{bmatrix} z \\ c \end{bmatrix})$$

(IGM3) *Singularities* Singularities are mapped to the integer grid. In particular, singular edges are mapped to segments on integer lines

$$\forall e \in S_E : \exists z \in \mathbb{Z}^2,\, a,\, b \in \mathbb{R},\, R \in \mathrm{Oct} : \boldsymbol{f}(e) = (\boldsymbol{R} \begin{bmatrix} z \\ a \end{bmatrix},\, \boldsymbol{R} \begin{bmatrix} z \\ b \end{bmatrix})$$

and singular vertices are mapped to integer points.

$$\forall v \in S_V : \boldsymbol{f}(v) \in \mathbb{Z}^3$$

(IGM4) *Local injectivity* The image of each tet has a positive volume.

$$\forall c_i \in C : \mathrm{ori3D}(\boldsymbol{u_i},\, \boldsymbol{v_i},\, \boldsymbol{w_i},\, \boldsymbol{x_i}) = 1$$

.

A parametrization which satisfies (IGM1)-(IGM3) approximately, due to the limited precision of floating point arithmetic, and disregards (IGM4) entirely is called a *relaxed integer-grid map* (Lyon et al. 2016). Similarly, we refer to a parametrization which satisfies (IGM1)-(IGM3) approximately, but ensures (IGM4) as a *positively relaxed integer-grid map*.

Using OVM, transition functions are stored per half-face where $\boldsymbol{\tau}_{(f,c)}$ refers to the transition from the chart of $c$ to the other cell incident to $f$. For boundary half-faces with no incident cell, the transition function is set to the identity.

# 3 Robust HexEx and Fast HexEx

From now on, to differentiate between the original and the optimized HexEx, we refer to the original algorithm, provided by (Lyon et al. 2016), as *Robust HexEx* and to the new algorithm, provided with this work, as *Fast HexEx*. If, in the context, there is no noteworthy distinction between the two, we refer to the algorithm simply as *HexEx*.
On a fundamental level, Robust HexEx and Fast HexEx differ in two ways. Firstly, Robust HexEx expects the IGM to be only relaxed, i.e. allows for flipped (negative volume) or degenerate (zero volume) tets in the parametrization. Fast HexEx is more strict and expects the IGM to be positively relaxed, meaning all parametrized tet-cells must have a positive volume (IGM4). Secondly, the two versions differ in their primary data structure. Whereas Robust HexEx uses darts (Kraemer et al. 2014), Fast HexEx makes use of a new data structure called *propeller* which we designed specifically for its use in the topology extraction phase of the algorithm. The two data structures are presented in Section 3.1 and Section 3.2.
Given a tuple $(\mathcal{M}_{tet}, \boldsymbol{f})$, consisting of a manifold tetrahedral mesh $\mathcal{M}_{tet}$ and a (positively) relaxed integer-grid map $\boldsymbol{f}$, HexEx extracts a boundary aligned hexahedral mesh $\mathcal{M}_{hex}$. The hexahedral mesh extraction consists of four stages:

1. *Preprocessing* The transition functions $\boldsymbol{\tau}_{ij}$ are extracted from the IGM and the parametrization is sanitized, meaning the exact fulfillment of constraints (IGM1)-(IGM3) is ensured.

2. *Geometry Extraction* For each integer-grid point intersecting with a parametrization of a tet-mesh-element, a hex-vertex is extracted.

3. *Topology Extraction* For each extracted hex-vertex, a list of darts/propellers, based on intersections of the integer grid with an infinitesimal neighborhood of the parametrization around the vertex, is extracted. Then, they are interconnected by navigating through the parametrized input mesh.

4. *Postprocessing* Inconsistencies due to flipped or degenerate tets are resolved.

In Fast HexEx, the preprocessing step remains untouched and is therefore not further discussed here[2]. What is important to note here is, that after the preprocessing phase, constraints (IGM1)-(IGM4) are all satisfied exactly and for any further geometric calculations, we use exact predicates.
Thanks to the constraint (IGM4) being satisfied, the postprocessing step can be completely omitted.
In Section 4 and Section 5 we will discuss the vertex- and topology extraction, which are the two stages we aim to optimize, in more detail.

[2]The ideas can be found in (Lyon et al. 2016) or in (Ebke et al. 2013) for the 2D equivalent.

## 3.1 The Dart Data Structure

Robust HexEx uses the *dart* data structure, based on (Kraemer et al. 2014), to define the topology of the resulting hex-mesh. A dart is a tuple of a vertex, an edge, a face and a cell which are all incident to one another. The unique set of all darts is denoted by $\mathcal{D} = \{(v, e, f, c) : v \sim e, e \sim f, f \sim c\} \subset V \times E \times F \times C$. The definition of darts is entirely combinatorial and does not depend on the geometry of the mesh.
Furthermore, for each dart, four connections $\alpha_0$, ..., $\alpha_3$ to other darts are stored where $\alpha_i(d)$ is the unique dart in $\mathcal{D}$ which shares all elements with $d$ except the $i-$element, as seen in Fig. 5. For boundary faces, the $\alpha_3$ connection is omitted.

$$\alpha_0(v, e, f, c) = (v', e, f, c) \in \mathcal{D}, v \neq v'$$
$$\alpha_1(v, e, f, c) = (v, e', f, c) \in \mathcal{D}, e \neq e'$$
$$\alpha_2(v, e, f, c) = (v, e, f', c) \in \mathcal{D}, f \neq f'$$
$$\alpha_3(v, e, f, c) = (v, e, f, c') \in \mathcal{D}, c \neq c'$$

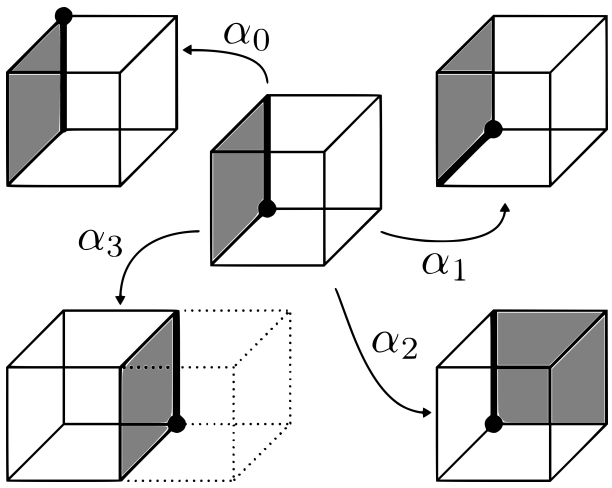



**Figure 5:** *Illustration of the four dart connections. The elements of the darts are highlighted.*

*Source: (Lyon et al. 2016)*

## 3.2 The Propeller Data Structure

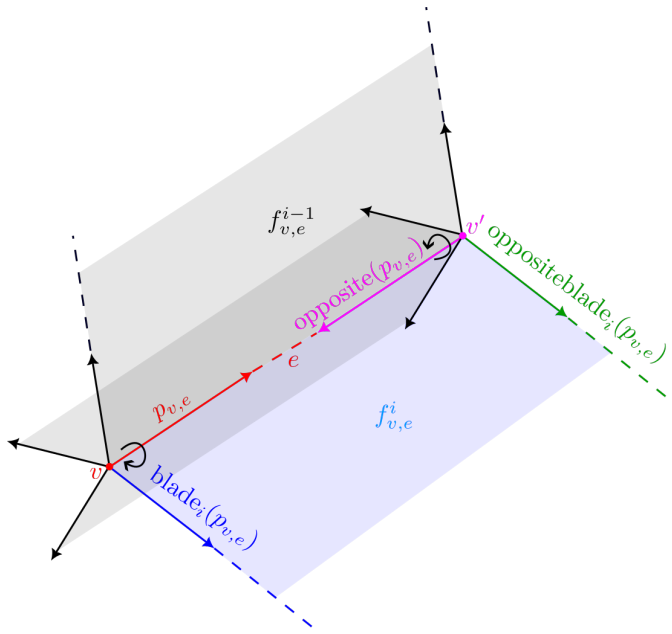



**Figure 6:** *A propeller $p_{v,e} = (v, e, f^0_{v,e}, f^1_{v,e}, f^2_{v,e}, f^3_{v,e})$ on a valence four edge $e$ is connected to the opposing propeller on the same edge. Its blades are ordered counter-clockwise around itself. The $i-$th blade and the corresponding opposite blade are incident to a common face $f^i_{v,e}$ and $f^{i-1}_{v,e}$ and $f^i_{v,e}$ are incident to a common cell.*

For a hexahedral manifold mesh $\mathcal{H} = (V, E, F, C)$ we define a set of $2|E|$ *propellers* as:

$$\mathcal{P} = \{(v, e, f^0_{v,e}, ..., f^{m_e-1}_{v,e}) : v \sim e, \forall i : e \sim f^i_{v,e}, \forall j \neq i : f^i_{v,e} \neq f^j_{v,e}\} \subset V \times E \times F^* \quad (1)$$

where $m_e = \text{valence}(e)$ is the number of faces in $F$ which are incident to $e \in E$.
Each propeller $p_{v,e} = (v, e, f^0_{v,e}, ..., f^{m_e-1}_{v,e}) \in \mathcal{P}$ then is a tuple corresponding to one unique half-edge $(v, e) \in E^{\frac{1}{2}}$ with all its incident faces $f^i_{v,e}$ around the edge[3].
The ordering of the faces is restricted by the following constraints:

(P1) Two consecutive faces are incident to a (unique) common cell.

$$\forall i \in \{0, ..., m_e - 2\} : \exists c \in C : f^i_{v,e} \sim c, f^{i+1}_{v,e} \sim c$$

(P2) The faces of two opposing propellers on the same edge are ordered conformingly which shall mean there exists a (unique) connection offset $l_e \in \{0, ..., m_e - 1\}$ per edge such that

$$\forall i \in \{0, ..., m_e - 1\} : f^i_{v,e} = f^{(l_e - i) \bmod m_e}_{v',e}$$

where $v' \neq v$ are adjacent vertices incident to the common edge $e$.

[3]Comparing it with OVM, a propeller is essentially a half-edge with an implicit half-edge-half-face-iterator.

For each propeller $p_{v,e} \in \mathcal{P}$ we store $m_e + 1$ connections to other propellers:

$$\begin{aligned} \text{opposite}(p_{v,e}) &= p_{v',e} \in \mathcal{P},\ v \neq v' \\ \text{blade}_i(p_{v,e}) &= p_{v,e'} \in \mathcal{P},\ e \neq e',\ e' \sim f_{v,e}^{i \bmod m_e} \end{aligned}$$

The opposite propeller $\text{opposite}(p_{v,e})$ is the unique propeller in $\mathcal{P}$ which lies on the opposing half-edge i.e. on the same edge but different vertex. The $i-$th blade $\text{blade}_i(p_{v,e})$ is the unique propeller on the same vertex but different edge such that both edges are incident to the same face $f_{v,e}^i$. An illustration is given in Fig. 6.
Additionally, for each edge connection between two opposing propellers $p_{v,e}$ and $p_{v',e}$, the connection offset $l_e$, as defined in (P2), is stored, as well as, whether or not the edge is part of the mesh's boundary.
If the edge is indeed a boundary edge then the last and first face, $f_{v,e}^{m_e-1}$ and $f_{v,e}^0$, are only incident to a common cell if $m_e = 2$. Otherwise, if the edge is not a boundary edge, (P1) implies that the last and first face are always incident to a common cell.
The propeller on the opposing vertex and different edge such that both edges are incident to the same face $f_{v,e}^i$ is implicitly given by:

$$\text{oppositeblade}_i(p_{v,e}) = \text{blade}_{(l_e - i)}(\text{opposite}(p_{v,e}))$$

Note that, if $\text{oppositeblade}_i(p_{v,e}) = \text{blade}_j(\text{opposite}(p_{v,e}))$, then the connection offset is $l_e = i + j$. Furthermore $\text{oppositeblade}_{i+k}(p_{v,e}) = \text{blade}_{j-k}(\text{opposite}(p_{v,e}))$ for every $k$.
For convenience, we define the next and previous blade of a propeller blade as

$$\text{nextblade}_i(p_{v,e}) = \begin{cases} \text{undefined}, & \text{if } e \text{ is boundary and } i = m_e - 1 \\ \text{blade}_{(i+1)}(p_{v,e}), & \text{else} \end{cases} \tag{2}$$

$$\text{prevblade}_i(p_{v,e}) = \begin{cases} \text{undefined}, & \text{if } e \text{ is boundary and } i = 0 \\ \text{blade}_{(i-1)}(p_{v,e}), & \text{else} \end{cases} \tag{3}$$

where they are undefined if the two consecutive blade faces are boundary faces of a different cell.
Whereas the set of darts is always unique for a given hex-mesh, this is not directly the case for the set of propellers based on the aforementioned definition as it allows blades to be ordered either clockwise or counterclockwise around the half-edge and the initial face for inner propellers is not well-defined. A canonical definition can be achieved by requiring blades to always be ordered in a counterclockwise manner around the half-edge, which directly ensures (P2), and the initial blade for inner propellers to be the one with the smallest index, assuming an index based structure.

## 3.3 Properties

Even though both constraints do not mention the geometry of the mesh, the propellers are only defined for manifold complexes. Only then is (P1) always feasible. However, we could easily store a boolean value per blade that determines if the blade and the next blade are incident to a common cell and adjust (P1) accordingly. This would allow the propellers to be used for non-manifolds. Moreover, nothing restricts propellers to only hexahedral meshes, any polyhedral mesh could be defined using this data structure.

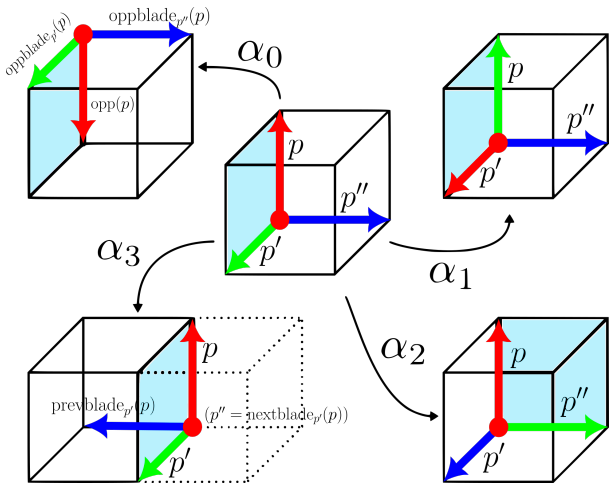



**Figure 7:** *Three ordered propellers (red, green, blue), which are blades among themselves, correspond to a dart. The $\alpha_i$ connections of the dart can then given by reordering ($\alpha_1$, $\alpha_2$) or by the propeller connections ($\alpha_0$, $\alpha_3$).*

We now compare the two data structures, darts and propellers, only considering hex-meshes.
In both, only vertices are stored explicitly while edges, faces and cells are stored implicitly by the interconnections.
If we define the set of all propeller triples that are incident to a common hex-cell and hex-vertex as

$$\mathcal{P}^3 = \{(p_{v,e},\, p_{v,e'},\, p_{v,e''}) : e \neq e',\, e \neq e'',\, e' \neq e'' \text{ and } \exists c \in C : e,\, e',\, e'' \sim c)\} \subset \mathcal{P} \times \mathcal{P} \times \mathcal{P}$$

then it follows that any two of the propellers are consecutive blades of the third propeller in the triple. A bijection between $\mathcal{P}^3$ and the set of darts $\mathcal{D}$ can then be defined as:

$$\text{dartify} : \mathcal{P}^3 \to \mathcal{D},\, (p_{v,e},\, p_{v,e'},\, p_{v,e''}) \mapsto (v,\, e,\, f,\, c) \text{ s.t. } e,\, e' \sim f \text{ and } e,\, e',\, e'' \sim c$$

If $\text{dartify}^{-1}(v,\, e,\, f,\, c) = (p_{v,e},\, p_{v,e'},\, p_{v,e''}) = (p_{v,e},\, \text{blade}_i(p_{v,e}),\, \text{nextblade}_i(p_{v,e}))$, the $\alpha_i$ pointers of the dart would then be given by

$$\begin{aligned}
\alpha_0(p_{v,e},\, p_{v,e'},\, p_{v,e''}) &= (\text{opposite}(p_{v,e}),\, \text{oppositeblade}_i(p_{v,e}),\, \text{oppositeblade}_{i+1}(p_{v,e})) \\
\alpha_1(p_{v,e},\, p_{v,e'},\, p_{v,e''}) &= (p_{v,e'},\, p_{v,e},\, p_{v,e''}) \\
\alpha_2(p_{v,e},\, p_{v,e'},\, p_{v,e''}) &= (p_{v,e},\, p_{v,e''},\, p_{v,e'}) \\
\alpha_3(p_{v,e},\, p_{v,e'},\, p_{v,e''}) &= (p_{v,e},\, p_{v,e'},\, \text{prevblade}_i(p_{v,e}))
\end{aligned}$$

as visualized in Fig. 7.

There are obviously $2|E|$ propellers in $\mathcal{P}$ as there is a one-to-one correspondence between the propellers and the half-edges.
This will always be significantly less than the number of darts in $\mathcal{D}$. Each hexahedral cell consists of six faces, each face consists of four edges and each edge consists of two vertices. This means, there are 48 darts per cell and $48|C|$ darts in total.
In the extreme and practically irrelevant case in which all cells of the mesh are isolated there are still twice as many darts as there are propellers. In practicality this number is higher (mostly around six or seven in our observations) which highlights an advantage of the propellers in our application compared to the darts.
Lastly, we consider the number of connections that need to be stored. Each dart will always store four connections, though one of them ($\alpha_3$) might not be defined. This brings the total number of stored connections to $4|\mathcal{D}| = 192|C|$.
The number of connections for each propeller is $m_e + 1$, depending on the edge valence. In total, there are therefore $2\sum_{e\in E}(m_e + 1) = 2\sum_{e\in E} m_e + 2|E| = 8|F| + 2|E|$ propeller connections. The last equation follows from the fact that each face consists of four edges. These observations are summarized in Table 1.

| Property | Darts $\mathcal{D}$ | Propellers $\mathcal{P}$ |
|---|---|---|
| Defined for non-manifolds | Yes | Yes |
| Defined for general polyhedral meshes | Yes | Yes |
| Well defined | Yes | Yes |
| Number of elements | $48\|C\|$ | $2\|E\|$ |
| Number of connections | $192\|C\|$ | $8\|F\| + 2\|E\|$ |

**Table 1:** *Comparison of the set of darts with the set of propellers in terms of some basic properties when used to define a hexahedral mesh.*

# 4 Vertex Extraction

Each integer-grid point $\boldsymbol{z} \in \mathbb{Z}^3$ intersecting a parametrized tet-element implies a vertex of the final hex-mesh or hex-vertex for short. The tet-element on which the hex-vertex is extracted is called the *generator* of the hex-vertex with the set of all hex-vertex-generators being denoted by $G \subset V \cup E \cup F \cup C$.
To avoid duplicate hex-vertices on shared boundary elements, the boundary is excluded. More formally, we define the *parametric volume* $\boldsymbol{F}_c$ of an element as its parametric image in the chart of a certain cell. The parametric volume of a $d-$element is then a $d-$polytope corresponding to the convex hull of its vertex parameters.

$$\boldsymbol{F}_c : V \cup E \cup F \cup C \to \mathbb{P}(\mathbb{R}^3),\ x \mapsto \text{conv}(\boldsymbol{f}_c(x))$$

Furthermore, we define the *parametric interior* $\mathring{\boldsymbol{F}}_c$ of an element as its parametric volume minus the parametric volume of its boundary.

$$\mathring{\boldsymbol{F}}_c(v) = \boldsymbol{F}_c(v) \tag{4}$$

$$\mathring{\boldsymbol{F}}_c(e) = \boldsymbol{F}_c(e) \setminus \bigcup_{v \sim e} \boldsymbol{F}_c(v) \tag{5}$$

$$\mathring{\boldsymbol{F}}_c(f) = \boldsymbol{F}_c(f) \setminus \bigcup_{e \sim f} \boldsymbol{F}_c(e) \tag{6}$$

$$\mathring{\boldsymbol{F}}_c(c) = \boldsymbol{F}_c(c) \setminus \bigcup_{f \sim c} \boldsymbol{F}_c(f) \tag{7}$$

A high-level algorithm for the whole extraction algorithm of hex-vertices from edges is given in Algorithm 1 with the concept for vertices, faces and cells being exactly the same. The position of an extracted hex-vertex is given by mapping the integer-grid parameter inside the parametric interior back to the input mesh domain. ${\boldsymbol{f}_c}^{-1}$ exists because $\boldsymbol{f}_c$ is injective (IGM4).
The problem boils down to finding $\mathring{\boldsymbol{F}}_c \cap \mathbb{Z}^3$ in an efficient manner.
In Section 2.2 it was stated that exact predicates ONLINESEGMENT, INTRIANGLE and INTETRAHEDRON are given to reliably perform the necessary checks. The check needed for vertices is trivial, one simply examines if its parameter is integer.
A naive approach and the one used by Robust HexEx for faces and cells, is to check all points inside the axis aligned bounding box of the parametric image. In general, multiple integer-grid points in the bounding box are not inside the polytope. Hence, a more efficient approach that aims to minimize the number of exact predicate calls is to apply common rasterization steps. The routines to extract integer-grid points from edges (line segments), faces (triangles) and cells (tetrahedra) are presented in the following.

**Algorithm 1** Vertex Extraction on Edges (High-Level)

```
for e ∈ E do
    pick any cell c ∈ C s.t. e ∼ c
    Z ← F̃_c(e) ∩ ℤ³
    if Z ≠ ∅ then
        G ← G ∪ {e}
        for z ∈ Z do
            generate hex-vertex with generator e, geometric embedding f_c⁻¹(z)
            and parameter z in the chart of c
```

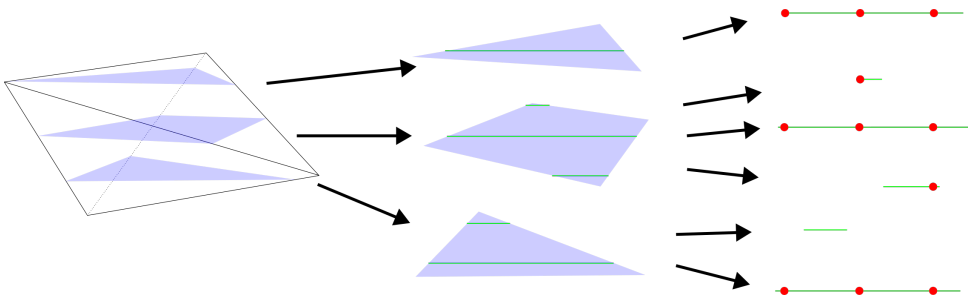


**Figure 8:** *Illustration of the tet rasterization hierarchy. From left to right: Rasterizing a tetrahedron means rasterizing multiple triangles and quadrilaterals which mean rasterizing multiple line segments.*

## 4.1 Rasterization

Commonly, rasterization refers to the process of converting a given shape into a series of pixels (2D) or voxels (3D). Here, we use the term to describe the process of enumerating all integer-grid points that are inside the parametric interior of a given polytope of a tet-mesh.
Our algorithms for rasterizing triangles, quadrilaterals and tetrahedra are derived from (Ruiz et al. 2004) and explained further in the following.

To simplify matters, we assume the coordinates of the points describing the rasterized polytope to be permuted such that the extents of the axis aligned bounding box of the polytope are sorted in descending order, i.e. if $S(A_1, ..., A_d) = \max_{i,j}(|A_i - A_j|)$ contains the three lengths of the bounding box, then

$$S_x \geq S_y \geq S_z \tag{8}$$

As we exclude the boundaries of our polytopes, it helps to define $\lceil x \rceil^* = \lfloor x \rfloor + 1$ and $\lfloor x \rfloor^* = \lceil x \rceil - 1$ as the smallest integer which is strictly larger than $x$ and the largest integer which is strictly smaller than $x$ respectively.
Figure 8 illustrates the rasterizations we are going to discuss. Each rasterization builds on the rasterizations of previous dimensions.

### 4.1.1 Line Segment

Let a line segment be given by its distinct endpoints $A$ and $B \in \mathbb{R}^3$ satisfying Eq. (8). We differentiate between the case in which the line segment lies parallel to or on an integer line, meaning $A_y = B_y$ and $A_z = B_z$, and the general 3D case. The former case is straightforward and does only require exact predicates when the line segment corresponds to a scanline of an enlarged triangle or tetrahedron as explained below. For the latter we sweep along one axis and test all nearest integer-grid points to the line segment, computed using linear interpolation and rounding, by evaluating the appropriate exact predicate. The axis of sweeping is chosen as $\arg\min_i(|A_i - B_i| > 0)$ (y or z) to minimize the number of points to check as illustrated in Fig. 9. The process is shown in Algorithm 2 where the sweeping axis is assumed to be the $y-$axis.

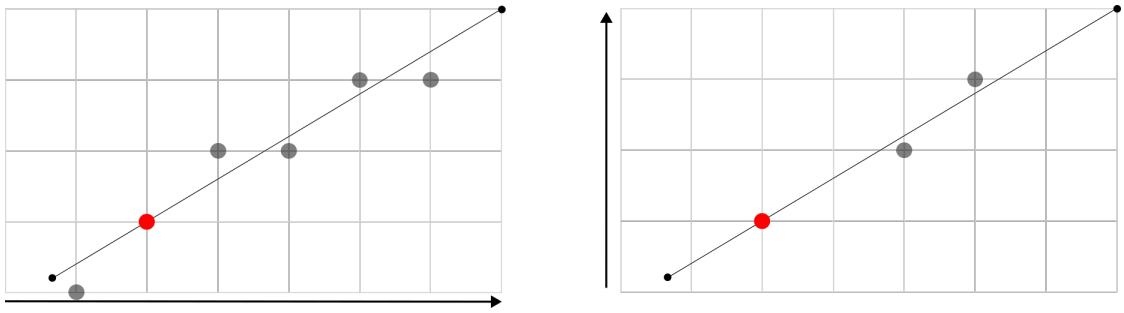


**Figure 9:** *Line Rasterization illustrated in 2D: Sweeping along the $x-$axis leads to* 6 *candidates (left) while sweeping along the y-axis leads to* 3*. The axis of sweeping is therefoe chosen as the $y-$axis. Marked red is the only integer-grid point on the line segment, the other candidates are gray.*

**Algorithm 2** Rasterization of a Line Segment

**Input**
$\boldsymbol{P} \in \{\textbf{ONLINESEGMENT}_{\boldsymbol{L}}, \textbf{INTRIANGLE}_{\boldsymbol{T}}, \textbf{INTETRAHEDRON}_{\boldsymbol{T}}\}$ Exact predicate
$\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^3$ Line segment s.t. (8)
**Output**
$\boldsymbol{L} \cap \mathbb{Z}^3$ Integer-grid points on the segment s.t. $P$ is satisfied

$Z \leftarrow \emptyset$
**if** $A_y = B_y$ and $A_z = B_z$ **then** *//1d case*
  **if** $(A_y, A_z) \in \mathbb{Z}^2$ **then** *//on integer-grid line?*
    $x1, x2 \leftarrow \lceil \min(A_x, B_x) \rceil^*, \lfloor \max(A_x, B_x) \rfloor^*$ *//1d interval integer endpoints*
    **if** $P \neq$ ONLINESEGMENT **then** *//subroutine of triangle or quad/tet?*
      $x1 \leftarrow \arg\min_{x1 \leq x \leq x2}(P(x, A_y, A_z))$ *//account for enlarged element*
      $x2 \leftarrow \arg\max_{x1 \leq x \leq x2}(P(x, A_y, A_z))$
    **for** $x = x1$ **upto** $x2$ **do** *//make use of convexity*
      $Z \leftarrow Z \cup \{(x, A_y, A_z)\}$
**else** *//3d case - rasterize along y-axis*
  $y1, y2 \leftarrow \min(A_y, B_y), \max(A_y, B_y)$ *//bottom and top*
  **for** $y = \lceil y1 \rceil^*$ **upto** $\lfloor y2 \rfloor^*$ **do**
    $t \leftarrow \frac{y - y1}{y2 - y1} \in (0, 1)$ *//linear interpolation*
    $\boldsymbol{z} \leftarrow \text{round}((1-t) \cdot A + t \cdot B) \in \mathbb{Z}^3$ *//nearest integer-grid point*
    **if** $P(\boldsymbol{z})$ **then** *//on line/in triangle/in tet?*
      $Z \leftarrow Z \cup \{\boldsymbol{z}\}$
**return** $Z$

### 4.1.2 Triangle

For a triangle, we use a common scanline approach like presented in (Ruiz et al. 2004) to enumerate all its integer-grid points.

With Eq. (8) being satisfied, the points ($A$, $B$, $C$) of the triangle are sorted in descending order with respect to the axis of sweeping which is chosen as one minimizing the number of scanlines. This is going to be the $y-$axis for integer-aligned triangles (where $S_z = 0$) and the $z-$axis otherwise. In the following, we assume it to be the $y-$axis.

$$A_y \geq B_y \geq C_y \tag{9}$$

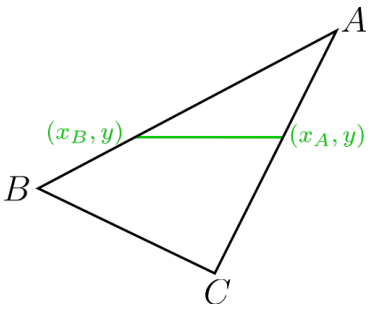



**Figure 10:** *Triangle Rasterization in 2D: With each iteration the scanline endpoints $(x_B, y)$ and $(x_A, y)$ are updated and the line segment is rasterized. Our rasterization in 3D works analogously.*

The level of the scanline is first initialized to $y = \lceil C_y \rceil^*$ and the scanline endpoints $P_A$ and $P_B$ are computed according to the slopes $m_{CA}$ and $m_{CB}$ and, with each iteration, they are updated by the slope of the edge they lie on. $P_B$ starts on the edge $CB$ and is incremented by the slope $m_{CB} = \frac{(B_x - C_x, B_z - C_z)}{B_y - C_y}$. Once it reaches the middle point $B$, it is adjusted to the edge $BA$ and from now on updated by $m_{BA}$. $P_A$ is always incremented by $m_{CA}$.

The core concept is illustrated in Fig. 10 and the implementation is given in Algorithm 3.

**Algorithm 3** Rasterization of a Triangle

**Input**
$\boldsymbol{P \in \{\text{INTRIANGLE}_T, \text{INTETRAHEDRON}_T\}}$ Exact predicate
$\boldsymbol{A, B, C \subset \mathbb{R}^3}$ Triangle s.t. (8)
**Output**
$\boldsymbol{T \cap \mathbb{Z}^3}$ Integer-grid points in the triangle s.t. $P$ is satisfied

$Z \leftarrow \emptyset$
Sort $A$, $B$, $C$ s.t. $A_y \geq B_y \geq C_y$
$m_{CA} \leftarrow \frac{(A_x - C_x, A_z - C_z)}{A_y - C_y}$ *//slope of edge CA*
$m_{CB} \leftarrow \frac{(B_x - C_x, B_z - C_z)}{B_y - C_y}$ *//slope of edge CB*
$m_{BA} \leftarrow \frac{(A_x - B_x, A_z - B_z)}{A_y - B_y}$ *//slope of edge BA*
$y \leftarrow \lceil C_y \rceil^*$
$(x_A, z_A) \leftarrow (C_x, C_z) + (y - C_y) \cdot m_{CA}$; $i_A \leftarrow m_{CA}$ *//interval endpoint wandering from C to A and increment*
$(x_B, z_B) \leftarrow (C_x, C_z) + (y - C_y) \cdot m_{CB}$; $i_B \leftarrow m_{CB}$ *//interval endpoint wandering from C over B to A and increment*
**while** $y \leq \lfloor A_y \rfloor^*$ **do**
    **if** $y = \lceil B_y \rceil^*$ **then** *//reached middle vertex B?*
        $(x_B, z_B) \leftarrow (B_x, B_z) + (y - C_y) \cdot m_{BA}$; $i_B \leftarrow m_{BA}$ *//update appropriate point and increment*
    $Z \leftarrow Z \cup \text{rasterizeLineSegment}(P, (x_A, y, z_A), (x_B, y, z_B))$ *//Algorithm 2*
    $y \leftarrow y + 1$
    $(x_{A,B}, z_{A,B}) \leftarrow (x_{A,B}, z_{A,B}) + i_{A,B}$
**return** $Z$

### 4.1.3 Tetrahedron

Conceptually, rasterizing a tetrahedron functions like rasterizing a triangle (Fig. 11). Instead of scanlines, there are two-dimensional scanplanes corresponding to the intersection of an axis aligned plane with the tetrahedron. The axis of sweeping minimizing the number of scanplanes is always the $z-$axis because of Eq. (8). A scanplane here is either a triangle or a (convex) quadrilateral with the latter being the case if it lies between the two middle points of the tetrahedron.

It is common practice to rasterize a general polygon by separating it into individual triangles. However, as we consider only convex, two-dimensional[4] quads and our goal is to minimize the use of exact predicates, we present below a method to rasterize them directly. Algorithm 4 shows the tet-rasterization.

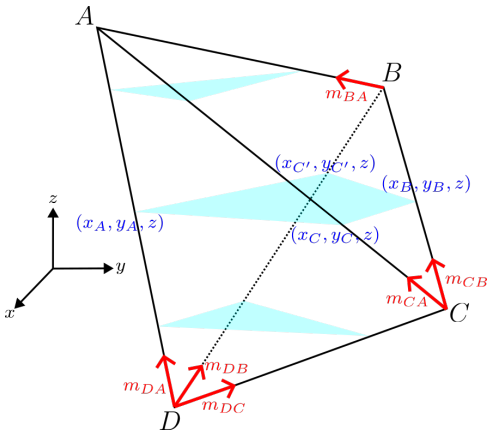



**Figure 11:** *Illustration of the rasterization of a tetrahedron. When intersecting the tet between $C$ and $B$, the intersection is a quadrilateral instead of a triangle.*

**Algorithm 4** Rasterization of a Tetrahedron

**Input**

$\boldsymbol{P = \text{INTETRAHEDRON}_T}$ Exact predicate

$\boldsymbol{A, B, C, D} \subset \mathbb{R}^3$ Tetrahedron s.t. (8)

**Output**

$\boldsymbol{T} \cap \mathbb{Z}^3$ Integer-grid-points in the tet s.t. $P$ is satisfied

$Z \leftarrow \emptyset$

Sort $A$, $B$, $C$, $D$ s.t. $A_z \geq B_z \geq C_z \geq D_z$

$m_{DA}, m_{DB}, m_{DC} \leftarrow \frac{(A_x - D_x, A_y - D_y)}{A_z - D_z}, \frac{(B_x - D_x, B_y - D_y)}{B_z - D_z}, \frac{(C_x - D_x, C_y - D_y)}{C_z - D_z}$ *//slopes*

$m_{CA}, m_{CB}, m_{BA} \leftarrow \frac{(A_x - C_x, A_y - C_y)}{A_z - C_z}, \frac{(B_x - C_x, B_y - C_y)}{B_z - C_z}, \frac{(A_x - B_x, A_y - B_y)}{A_z - B_z}$

$z \leftarrow \lceil D_z \rceil^*$

$(x_A, y_A) \leftarrow (D_x, D_y) + (z - D_z) \cdot m_{DA}$; $i_A \leftarrow m_{DA}$ *//scanplane points and increments*

$(x_B, y_B) \leftarrow (D_x, D_y) + (z - D_z) \cdot m_{DB}$; $i_B \leftarrow m_{DB}$

$(x_C, y_C) \leftarrow (D_x, D_y) + (z - D_z) \cdot m_{DC}$; $i_C \leftarrow m_{DC}$

$(x_{C'}, y_{C'}) \leftarrow (0, 0)$; $i_{C'} \leftarrow (0, 0)$

**while** $z \leq \lfloor A_z \rfloor^*$ **do**

  **if** $z = \lceil C_z \rceil^*$ **then** *//reached second lowest vertex $C$?*

    $(x_C, y_C) \leftarrow (C_x, C_y) + (z - C_z) \cdot m_{CA}$; $i_C \leftarrow m_{CA}$

    $(x_{C'}, y_{C'}) \leftarrow (C_x, C_y) + (z - C_z) \cdot m_{CB}$; $i_{C'} \leftarrow m_{CB}$

  **if** $z = \lceil B_z \rceil^*$ **then** *//reached third lowest vertex $B$?*

    $(x_B, y_B) \leftarrow (B_x, B_y) + (z - B_z) \cdot m_{BA}$; $i_B \leftarrow m_{BA}$

  **if** $C_z < z < B_z$ **then** *//Note that the ordering $A$, $B$, $C'$, $C$ always implies a convex, non-self-intersecting quad*

    $Z \leftarrow Z \cup \text{rasterizeQuadrilateral}(P, (x_A, y_A, z), (x_B, y_B, z), (x_{C'}, y_{C'}, z), (x_C, y_C, z))$ *//Algorithm 5*

  **else**

    $Z \leftarrow Z \cup \text{rasterizeTriangle}(P, (x_A, y_A, z), (x_B, y_B, z), (x_C, y_C, z))$ *//Algorithm 3*

  $z \leftarrow z + 1$

  $(x_{A,B,C,C'}, y_{A,B,C,C'}) \leftarrow (x_{A,B,C,C'}, y_{A,B,C,C'}) + i_{A,B,C,C'}$ *//increment*

**return** $Z$

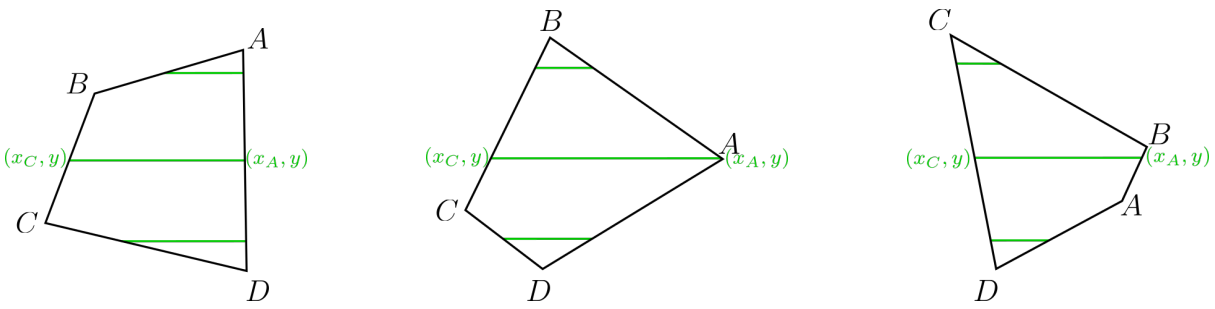
[4] A generalization to three dimensions is straightforward.

### 4.1.4 Quadrilateral

The idea for the rasterization of a quadrilateral given by $A$, $B$, $C$, $D \in \mathbb{R}^2$ remains the same. We assume the axis of sweeping to be the $y-$axis and

$$A_y,\, B_y,\, C_y \geq D_y \tag{10}$$

In analogy to a triangle, we start at the bottom point $D_y$ and update our scanline endpoints until the top point is reached. Unlike edges, triangles or tetrahedra, which are simplexes with each pair of vertices forming an edge, a quadrilateral is a $2-$polytope with 4 vertices and 4 edges. This means, $A$ can't be constrained to be the top point if $D$ is constrained to be the bottom one as that could imply a self-intersecting quad. With either $A$, $B$ or $C$ at the top, there are now three different configurations, each with its distinct way of updating the scanline endpoints as illustrated in Fig. 12. A detailed implementation is given in Algorithm 5.




**(a)** *$A$ is on top: $x_C$ moves through $C$ and $B$*

**(b)** *$B$ is on top: $x_C$ moves through $C$, $x_A$ through $A$*

**(c)** *$C$ is on top: $x_A$ moves through $A$ and $B$*

**Figure 12:** *With $D$ as the bottom point, depending on which point is on top, the scanline for a quadrilateral needs to be updated differently.*

**Algorithm 5** Rasterization of a Quadrilateral

**Input**
$\boldsymbol{P = \text{INTETRAHEDRON}_T}$ Exact predicate
$\boldsymbol{A, B, C, D} \subset \mathbb{R}^3$ Convex Quad s.t. (8) and $A_z = B_z = C_z = D_z \in \mathbb{Z}$
**Output**
$\boldsymbol{Q \cap \mathbb{Z}^3}$ Integer-grid points in the quad s.t. $P$ is satisfied

$Z \leftarrow \emptyset$
Shift $A$, $B$, $C$, $D$ s.t. $A_y$, $B_y$, $C_y \geq D_y$
$E \leftarrow \arg\max_{P \in \{A, B, C\}}(P_y)$ *//get top point*
$y \leftarrow \lceil D_y \rceil^*$
$m_{DA}$, $m_{DC}$, $m_{CB}$, $m_{AB} \leftarrow \frac{A_x - D_x}{A_y - D_y}$, $\frac{C_x - D_x}{C_y - D_y}$, $\frac{C_x - C_x}{B_y - C_y}$, $\frac{B_x - A_x}{B_y - A_y}$ *//slopes*
$x_A \leftarrow D_x + (y - D_y) \cdot m_{DA}$; $i_A \leftarrow m_{DA}$
$x_C \leftarrow D_x + (y - D_y) \cdot m_{DC}$; $i_C \leftarrow m_{DC}$ *//scanline endpoints and increments*
**while** $y \leq \lfloor E_y \rfloor^*$ **do**
  **if** $E = A$ **then** *//A is top?*
    **if** $y = \lceil B_y \rceil^*$ **then** *//reached B, second highest?*
      $x_C \leftarrow B_x + (y - B_y) \cdot m_{AB}$; $i_C \leftarrow m_{AB}$
    **else if** $y = \lceil C_y \rceil^*$ **then** *//reached C, second lowest?*
      $x_C \leftarrow C_x + (y - C_y) \cdot m_{CB}$; $i_C \leftarrow m_{CB}$
  **else if** $E = B$ **then** *//B is top?*
    **if** $y = \lceil B_y \rceil^*$ **then** *//reached A?*
      $x_A \leftarrow A_x + (y - A_y) \cdot m_{AB}$; $i_A \leftarrow m_{AB}$
    **if** $y = \lceil A_y \rceil^*$ **then** *//reached C?*
      $x_C \leftarrow C_x + (y - C_y) \cdot m_{CB}$; $i_C \leftarrow m_{CB}$
  **else** *//C is top?*
    **if** $y = \lceil B_y \rceil^*$ **then** *//reached B, second highest?*
      $x_A \leftarrow B_x + (y - B_y) \cdot m_{CB}$; $i_A \leftarrow m_{CB}$
    **else if** $y = \lceil A_y \rceil^*$ **then** *//reached A, second lowest?*
      $x_A \leftarrow A_x + (y - A_y) \cdot m_{AB}$; $i_A \leftarrow m_{AB}$
  $Z \leftarrow Z \cup \text{rasterizeLineSegment}(P, (x_A, y, A_z), (x_C, y, A_z))$ *//Algorithm 2*
  $y \leftarrow y + 1$
  $x_{A,C} \leftarrow x_{A,C} + i_{A,C}$
**return** $Z$

## 4.2 Robustness

While mathematically correct, these rasterization algorithms still need to consider floating-point errors which potentially cause inaccuracies like the one illustrated in Fig. 13 where the line segment in the triangle becomes too short, missing an integer point. To remedy this, before rasterizing, triangles and tetrahedra are enlarged by some small constant $\epsilon > 0$. For line segments, this is unnecessary since its endpoints are always given exactly. The rasterization steps are then performed with respect to the upscaled element whereas the exact predicates are evaluated with respect to the original element.

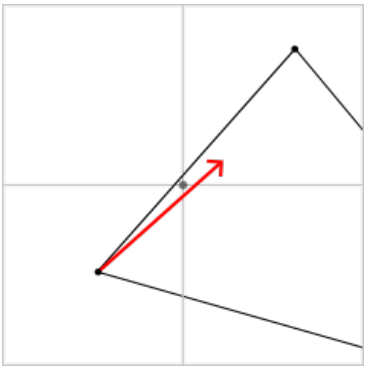


**Figure 13:** *The limited precision of floating point arithmetic leads to a point being considered outside the triangle, missing a hex-vertex.*

To achieve a constant distance between boundaries, given by $\epsilon > 0$, the points are scaled around the incenter $I$ by a factor $\frac{r+\epsilon}{r} > 1$ where $r > 0$ is the inradius (Fig. 14a).

For a tetrahedron given by points $A$, $B$, $C$, $D \in \mathbb{R}^3$, the *incenter* $I \in \mathbb{R}^3$ is given by $I = \frac{a \cdot A + b \cdot B + c \cdot C + d \cdot D}{a+b+c+d}$ where $a$, $b$, $c$, $d$ are the areas of the triangular faces and the *inradius* $r \in \mathbb{R}^{>0}$ is given by $r = \frac{3 \cdot |\text{volume}(A, B, C, D)|}{a+b+c+d}$. The incenter

and inradius of a triangle are given analogously. Each point $P$ defining the tet or triangle is then mapped to $P' = I + \frac{r+\epsilon}{r} \cdot (P - I)$.

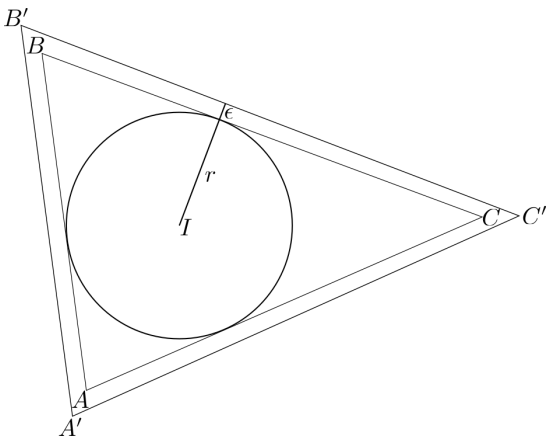
As different small values for $\epsilon$ did not have a noticeable effect on performance (Fig. 14b), it has been implemented as $\epsilon = 10^{-6}$ which is sufficiently larger than floating point inaccuracies[5].

The enlargement is accounted for as shown in Algorithm 2. When rasterizing an axis aligned integer line as a subroutine of rasterizing a triangular face or tetrahedral cell, one tests from both sides until the two endpoints inside the element are found. Then, due to convexity, all points in between do not require any exact predicates. All of the aforementioned calculations can be skipped if the bounding box of the element does not contain any integer-grid points, which is trivial to check.

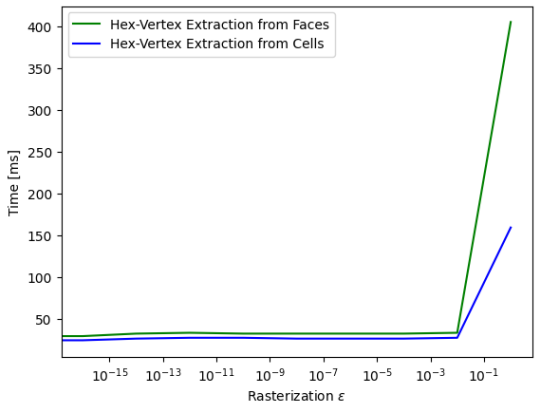
The vertex-extraction using the example of faces is shown in Algorithm 6.

**Algorithm 6** Vertex Extraction on Faces

**for** $f \in F$ **do**
  pick any cell $c \in C$ s.t. $f \sim c$
  $T \leftarrow (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) = \boldsymbol{f}_c(f)$
  **if** BOUNDINGBOX$(T) \cap \mathbb{Z}^3 = \emptyset$ **then**
    **continue**
  $T' \leftarrow$ enlargeTriangle$(T, \epsilon)$
  Permute the $x$, $y$, $z$ coordinates of $T$ and $T'$ according to $\pi \in S_3$ s.t. Eq. (8)
  $P \leftarrow$ INTRIANGLE$_T$
  $Z \leftarrow$ rasterizeTriangle$(P, T')$ *//Algorithm 3*
  **if** $Z \neq \emptyset$ **then**
    $G \leftarrow G \cup \{f\}$ *//add face $f$ to list of hex-vertex generators*
    **for** $\boldsymbol{z} \in Z$ **do**
      generate hex-vertex with generator $f$, geometric embedding $\boldsymbol{f}_c{}^{-1}(\pi^{-1}(\boldsymbol{z}))$
      and parameter $\pi^{-1}(\boldsymbol{z})$ in the chart of $c$




**(a)** *Illustration of a triangle upscaling around its incenter by some amount $\epsilon$.*




**(b)** *Timings of the hex-vertex extraction from faces and cells using different rasterization $\epsilon$ (model: n03u_skijump).*

[5] In C++, DBL_EPSILON, which defines the difference between 1.0 and the next value representable by a double precision floating point, has the order of magnitude −16 (base 10).

# 5 Topology Extraction

The extracted hex-vertices define the geometry of the mesh, but topology information, i.e. hex-edges, -faces and -cells, is still missing.
For parametrizations which are globally injective and only contain identity transitions, this problem would be trivial. However, while in practical examples, most transitions are indeed trivial, non-identity transitions between the charts of adjacent cells, especially around singularities, need to be considered.
The topology extraction is separated into two parts. First, the local topology per hex-vertex generator is extracted, resulting in a list of propellers with blade connections but not yet any opposite connections. In the second part, the propellers per hex-vertex are traced through the parametrized mesh, in accordance with the transitions, until the opposing vertex and propeller is reached. The topology of the hex-mesh is then given as stated in Section 3.2. It is our goal to avoid extracting any redundant information that is already explicitly or implicitly provided.

## 5.1 Local Topology

During the vertex extraction a list of hex-vertex generators is created, consisting of all tet-elements which contain at least one integer-grid point in their parametric interior. Let us now discuss how we extract the local topology per generator and consequently per hex-vertex while thinking about possible local configurations depending on the type of generator.

### 5.1.1 Enumerating Propellers

First, we formalize the problem and define an *integer-grid edge* as an integer line segment starting at $\boldsymbol{z}$ going into direction $\vec{\boldsymbol{d}}$

$$\mathcal{E}(\boldsymbol{z}, \vec{\boldsymbol{d}}) = \{\boldsymbol{z} + t\vec{\boldsymbol{d}} : 0 < t < 1\} \tag{11}$$

and an *integer-grid face* as an integer plane segment starting at $\boldsymbol{z}$ extending into orthonormal directions $\vec{\boldsymbol{d}}_1$ and $\vec{\boldsymbol{d}}_2$

$$\mathcal{F}(\boldsymbol{z}, \vec{\boldsymbol{d}}_1, \vec{\boldsymbol{d}}_2) = \{\boldsymbol{z} + t_1\vec{\boldsymbol{d}}_1 + t_2\vec{\boldsymbol{d}}_2 : 0 < t_1, t_2 < 1\} \tag{12}$$

where $\boldsymbol{z} \in \mathbb{Z}^3$ is an integer parameter and $\vec{\boldsymbol{d}}, \vec{\boldsymbol{d}}_1, \vec{\boldsymbol{d}}_2 \in \{\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \vec{\boldsymbol{e}}_3, -\vec{\boldsymbol{e}}_1, -\vec{\boldsymbol{e}}_2, -\vec{\boldsymbol{e}}_3\}$ are axis aligned and unit length directions. Strict inequalities are used to exclude $\boldsymbol{z}$ from the integer-grid edge and integer-grid face.
Let $\boldsymbol{z} \in \mathring{\boldsymbol{F}}_c(x) \cap \mathbb{Z}^3$ be a point in the parametric interior of an element $x \in V \cup E \cup F \cup C$ and $\vec{\boldsymbol{d}}$ as above. Then we generate a propeller $p$, i.e. a halfedge of the hex-mesh, if

the corresponding integer-grid line intersects the parametric interior of an element in the local neighborhood.

$$\text{POINTSINTO}_c(x,\, y,\, \vec{\boldsymbol{d}}) \iff \mathcal{E}(\boldsymbol{z},\, \vec{\boldsymbol{d}}) \cap \mathring{\boldsymbol{F}}_c(y) \neq \emptyset \tag{13}$$

where $y \in E \cup F \cup C$ is either an incident element of $x$ of higher dimensionality or $x$ itself. POINTSINTO is carried over from the implementation of Robust HexEx by (Lyon et al. 2016). Note that we omit $\boldsymbol{z}$ as a parameter for the predicate because the local neighborhood of each point in the same parametric interior is identical. This observation is particularly beneficial for fine parametrizations in which multiple hex-vertices are extracted on the same generator since we only need to enumerate the local topology per generator, instead of per hex-vertex.

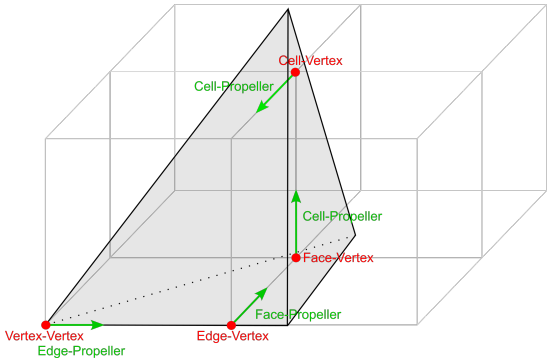



**Figure 15:** *Illustration of different propeller (and vertex) types. The type of a propeller is given by its holder and the type of a hex-vertex is given by its generator.*

Furthermore, since $\text{POINTSINTO}_c(c,\, c,\, \vec{\boldsymbol{d}})$ is always satisfied, the local topology extraction on cell-generators can be cut entirely. A cell-generator will always have six outgoing propellers, one in each of the axis aligned directions in the parametrization.
Another observation that further reduces the required use of exact predicates is that an integer-grid edge will never point from an element into an incident element in two opposing directions simultaneously with the opposite being true for directions from an element along itself.
Algorithm 7 shows the propeller root enumeration, i.e. the enumeration of "naked" propellers without yet any interconnections, for edge-generators. The process for vertex-generators is nearly identical, except that we need to iterate over all incident edges too, whereas it is a simplification for face-generators in which one checks the predicate for its one or two incident cells and the face itself. As explained, the process is skipped for cell-generators.

The element on whose parametric interior the propeller lies is referred to as the *holder* of the propeller, similarly to how we consider generators of hex-vertices. We refer to a propeller with a holder of type HTYPE as a HTYPE-propeller and when aiming to be more precise as a GTYPE-HTYPE-propeller where GTYPE is the type of the generator of the propeller. For example, an edge-cell-propeller is a propeller on a generator edge with a holder cell. Different types are illustrated in Fig. 15.

Propellers that got extracted on generator $g$ are stored in a list $\mathcal{P}_g$.
Due to non-identity transitions, propellers might have different images in different charts

of cells incident to their holder. All of them are explicitly stored and denoted by $\vec{\boldsymbol{d}}_c(p)$ for holder$[p] \sim c$. To find a propeller with a certain direction in a certain cell chart, like on Line 4 of Algorithm 10, an appropriate hash map is used.

**Algorithm 7** *enumeratePropellerRoots:* Propeller Root Enumeration on an edge-hex-vertex. We write $(g,\, h,\, c,\, \vec{\boldsymbol{d}})$ for a propeller on generator $g$ with holder $h$ and direction $\vec{\boldsymbol{d}}$ in the chart of cell $c$

**Input**
  $\boldsymbol{e \in G \cap E}$ Edge generator
1: $\mathcal{P}_e \leftarrow []$ *//list of propellers on e*
2: Pick any hex-vertex $v_h$ that got extracted on $e$.
3: **for each** incident cell $c \in C$ and $\vec{\boldsymbol{d}} \in \{\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \vec{\boldsymbol{e}}_3\}$ **do** *//Edge-Cell-Propellers*
4:  **if** POINTSINTO$_c(e,\, c,\, \vec{\boldsymbol{d}})$ **then**
5:  $\mathcal{P}_e \leftarrow \mathcal{P}_e + [(e,\, c,\, c,\, \vec{\boldsymbol{d}})]$
6:  **else if** POINTSINTO$_c(e,\, c,\, -\vec{\boldsymbol{d}})$ **then** *//Predicate cannot be true for two opposing directions simultaneously*
7:  $\mathcal{P}_e \leftarrow \mathcal{P}_e + [(e,\, c,\, c,\, -\vec{\boldsymbol{d}})]$
8: **for each** incident face $f \in F$ and $\vec{\boldsymbol{d}} \in \{\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \vec{\boldsymbol{e}}_3\}$ **do** *//Edge-Face-Propellers*
9:  Pick any cell $c \in C$ s.t. $f \sim c$.
10:  **if** POINTSINTO$_c(e,\, f,\, \vec{\boldsymbol{d}})$ **then**
11:  $\mathcal{P}_e \leftarrow \mathcal{P}_e + [(e,\, f,\, c,\, \vec{\boldsymbol{d}})]$
12:  **else if** POINTSINTO$_c(e,\, f,\, -\vec{\boldsymbol{d}})$ **then** *//Predicate cannot be true for two opposing directions simultaneously*
13:  $\mathcal{P}_e \leftarrow \mathcal{P}_e + [(e,\, f,\, c,\, -\vec{\boldsymbol{d}})]$
14: Pick any cell $c \in C$ s.t. $e \sim c$. *//Edge-Edge-Propellers*
15: **for each** $\vec{\boldsymbol{d}} \in \{\vec{\boldsymbol{e}}_1, \vec{\boldsymbol{e}}_2, \vec{\boldsymbol{e}}_3\}$ **do**
16:  **if** POINTSINTO$_c(e,\, e,\, \vec{\boldsymbol{d}})$ **then**
17:  $\mathcal{P}_e \leftarrow \mathcal{P}_e + [(e,\, e,\, c,\, \vec{\boldsymbol{d}}),\, (e,\, e,\, c,\, -\vec{\boldsymbol{d}})]$ *//Predicate for two opposing directions is the same*
18:  **break**

### 5.1.2 Connecting Propeller Blades

Analogously, we define the predicate POINTSINTO for two orthonormal directions to be true iff the integer-grid plane intersects the parametric interior.

$$\text{POINTSINTO}_c(x,\, y,\, z,\, \vec{\boldsymbol{d}}_1,\, \vec{\boldsymbol{d}}_2) \iff \mathcal{F}(\boldsymbol{z},\, \vec{\boldsymbol{d}}_1,\, \vec{\boldsymbol{d}}_2) \cap \mathring{\boldsymbol{F}}_c(z) \neq \emptyset$$

where $x \in V \cup E \cup F \cup C$ is a hex-vertex generator, $y \in E \cup F \cup C$ is a propeller holder and $z \in F \cup C$ is either $y$ itself or an incident element of higher dimension, $\vec{\boldsymbol{d}}_1$, $\vec{\boldsymbol{d}}_2$ are orthonormal and axis aligned, POINTSINTO$_c(x,\, y,\, \vec{\boldsymbol{d}}_1)$ is satisfied and $\boldsymbol{z}$ is any point in the parametric interior $\mathring{\boldsymbol{F}}_c(x)$. The implementation of POINTSINTO using exact predicates is relatively straightforward and not presented here.

First, we loop through each propeller on a generator and enumerate the orthonormal directions in counterclockwise order satisfying the predicate as seen in Algorithm 8 for edge-propellers. To achieve a counterclockwise ordering, one iterates over all incident tets $c \sim$ holder$[p]$ in CCW order and in each tet chart over all orthonormal directions to $\vec{\boldsymbol{d}}_2 \perp \vec{\boldsymbol{d}}_c(p)$ in CCW order. A consistent counterclockwise ordering guarantees $(P1)$ and $(P2)$. The process for face-propellers is a simplification in which one only iterates over the two cells incident to the holder where for one of the cells, POINTSINTO is evaluated against the parametric volume instead of the interior to include directions along the face

itself. For cell-propellers, there are always four blades in total, one in each of the four orthonormal directions, so no exact predicates are required.
Then, as seen in Algorithm 9 we loop through the propellers again and trace to the blade propeller by rotating on the integer-grid plane from $\vec{d}_c(p)$ into the direction previously enumerated.
The reason for iterating over the propellers twice instead of rotating to the blades directly is that the blade connections are symmetric and only one rotational tracing per connected propeller pair should be performed.
Algorithm 10 shows the rotational tracing along an integer-grid face through the parametrization mesh from a propeller to its blade. We start in the chart of a cell $c$ and, until the blade is found, exit the current cell into an adjacent one, through a face $f$ of $c$. The face through which the cell was entered is excluded. If the generator is a face, it must also be the transition face. Otherwise, it must be incident to the generator edge or vertex and intersect the integer-grid face. When rotating from a face-face- or edge-edge-propeller, we will always find the blade in the first cell chart, hence we do not need to consider these cases. Furthermore, we know that the transition face must be both incident to the generator and not equal to the previous transition face which leaves only one option for edge-generators, when the previous transition face is given. Figure 16 illustrates the rotational tracing and Algorithm 11 shows the implementation of pickNextHalfaceToBlade in detail.

### 5.1.3 Enumerating Hex-Corners

In preparation for the final step of the hex-mesh extraction, a list of *hex-corners* is generated. A *hex-corner* corresponds to a hex-vertex with an incident hex-cell and is represented as a triple of three distinct positively oriented propellers. Due to the counterclockwise ordering of the blades, each triple will be of the form ($p$, blades($p$)[$i$], blades($p$)[($i$+1) mod $m_p$]) where $m_p = \text{len}(\text{blades}(p))$. Only one triple is stored per hex-corner. If $p$ corresponds to an inner edge, the list of blades is considered cyclic whereas if it corresponds to a boundary hex-edge, meaning its holder is a boundary tet-element, there is no cell between the last and first blade (2). In this case, the triple ($p$, blades($p$)[$m_p - 1$], blades($p$)[0]) is disregarded.

## Algorithm 8 *enumerateDirectionsToBlades*

**Input**

$\boldsymbol{p}$ Edge-propeller with holder $e \in E$ and generator $g \in V \cup E$

blades[$p$] $\leftarrow$ []; bladeDirs[$p$] $\leftarrow$ []
**for each** incident halfface ($f$, $c$) around the parametrized $p$ in CCW order **do**
  $\vec{d}_1 \leftarrow \vec{d}_c(p)$)
  $D \leftarrow$ each of the four axis aligned $\vec{d}_2 \perp \vec{d}_1$ in CCW order
  **for** $i = 0, 1, 2, 3$ **do**
    $F[i] \leftarrow$ POINTSINTO$_c(g, e, f, \vec{d}_1, D[i])$ *//integer-grid plane along face?*
    $C[i] \leftarrow$ POINTSINTO$_c(g, e, c, \vec{d}_1, D[i])$ *//integer-grid plane into cell?*
    $P[i] \leftarrow F[i]$ or $C[i]$ *//integer-grid plane into anything?*
  **if** $\forall i = 0, 1, 2, 3 : \neg P[i]$ **then** *//no blades in this cell chart?*
    **continue** *//move on with next cell*
  $r \leftarrow 3$ *//ensure CCW blades order, since $P[i]$ is only true for 1 to 3 directions per c.*
  **while** $P[r-1]$ or $\neg P[r]$ **do**
    $r \leftarrow r - 1$
  **for** $i = 0, 1, 2, 3$ **do**
    $i \leftarrow (r + i) \mod 4$
    $\vec{d}_2 \leftarrow D[i]$
    **if** $F[i]$ **then** *//The direction along a face is stored in both incident charts*
      $c', \boldsymbol{\tau} \leftarrow$ other cell incident to $f$ and transition function from $c$ to $c'$
      bladeDirs[$p$] $\leftarrow$ bladeDirs[$p$] + [[($c$, $\vec{d}_2$), ($c'$, $\boldsymbol{\tau}(\vec{d}_2)$)]]
    **else if** $C[i]$ **then** *//For a direction into a cell, there is only one relevant chart*
      bladeDirs[$p$] $\leftarrow$ bladeDirs[$p$] + [[($c$, $\vec{d}_2$)]]

## Algorithm 9 *connectPropellerBlades*

**Input**

$\boldsymbol{p}$ propeller with holder $g$

$m \leftarrow$ len(bladeDirs[$p$])
**for** $i = 0, ..., m - 1$ **do**
  **if** blades[$p$][$i$] is defined **then** *//blade connection was already set*
    **continue**
  $c, \vec{d}_2 \leftarrow$ bladeDirs[$p$][$i$]
  $\vec{d}_1 \leftarrow \vec{d}_c(p)$
  $p', c', \boldsymbol{\tau} \leftarrow$ rotateToBlade($p$, $c$, $\vec{d}_1$, $\vec{d}_2$)
  $j \leftarrow$ index s.t. $(c', \boldsymbol{\tau}(\vec{d}_1)) \in$ bladeDirs[$p'$][$j$]
  blades[$p$][$i$] $\leftarrow p'$; blades[$p'$][$j$] $\leftarrow p$

## Algorithm 10 *rotateToBlade*

**Input**

$\boldsymbol{p, c, \vec{d}_1, \vec{d}_2}$ propeller on generator $g$, initial cell, initial directions of root and blade

**Output**

$\boldsymbol{p', c', \tau'}$ blade propeller, final cell, transition from $c$ to $c'$

$p' \leftarrow$ propeller with generator $g$ s.t. $\vec{d}_c(p') = \vec{d}_2$
$\boldsymbol{\tau}' \leftarrow$ id
($f$, $c$) $\leftarrow$ invalid halfface
**while** $p' \notin \mathcal{P}_g$ **do**
  ($f$, $c$) $\leftarrow$ pickNextHalfaceToBlade($p$, ($f$, $c$), $c$, $\vec{d}_1$, $\vec{d}_2$)
  $\boldsymbol{\tau} \leftarrow \boldsymbol{\tau}_{(f,c)}$; $\boldsymbol{\tau}' \leftarrow \boldsymbol{\tau} \circ \boldsymbol{\tau}'$ *//transition function from the chart of $c$ through $f$ and accumulated transition*
  ($f$, $c$) $\leftarrow$ ($f$, $c$).opposite *//move into the chart of the adjacent cell*
  $\vec{d}_1, \vec{d}_2 \leftarrow \boldsymbol{\tau}(\vec{d}_1, \vec{d}_2)$
  $p' \leftarrow$ propeller with generator $g$ s.t. $\vec{d}_c(p') = \vec{d}_2$
**return** $p'$, $c$, $\boldsymbol{\tau}'$

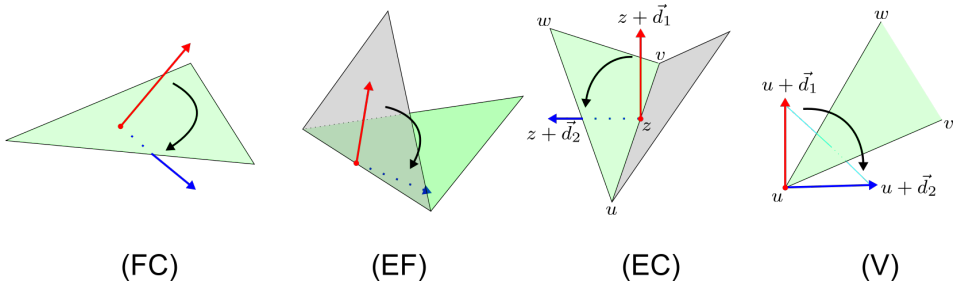



**Figure 16:** *Different cases to consider when rotationally tracing from a propeller root (red) to its blade (blue). (FC): From a face-cell-propeller, there is only one possible transition face. (EF): From an edge-face-propeller, the transition face must be incident to the generator but not equal to the holder. (EC): From an edge-cell-propeller, we pick the face on the correct side. (V) From a propeller with a vertex-generator, the rotation must go through the open ended triangle.*

**Algorithm 11** *pickNextHalffaceToBlade*: Note that this function will never be called for cell-cell propellers, face-face-propellers and edge-edge-propellers and up to once for face-cell propellers, making the other cases the only ones requiring exact predicates.

**Input**

$\boldsymbol{p}$ propeller on generator $g$

$\boldsymbol{(f, c)}$ entering halfface. Invalid or incident to $g$ and $c$

$\boldsymbol{c}$ current cell chart

$\boldsymbol{\vec{d}_1}, \boldsymbol{\vec{d}_2}$ propeller root and blade directions in the chart of $c$

**Output**

$\boldsymbol{(f', c)}$ exiting halfface incident to $c$ and not equal to $(f, c)$ which is intersected by the integer-grid face

**if** $g \in F$ **then** *//Face-cell-propeller*
  **return** $(g, c)$ *//(FC): only one option*
inFirstCell $\leftarrow (f, c)$ is invalid *//still in starting chart?*
**if** $g \in E$ **then**
  **if** ¬inFirstCell **then** *//Edge-generator, not inFirstCell*
    **return** unique $(f', c)$ s.t. $f' \neq f$, $g \sim f'$ *//only one option*
  **if** holder$[p] \in F$ **then** *//Edge-face-propeller, inFirstCell*
    **return** unique $(f', c)$ s.t. $f' \neq$ holder$[p]$, $g \sim f'$ *//(EF): only one option*
$\boldsymbol{z} \leftarrow \boldsymbol{f}_c(v_h)$ *//$v_h$ can be any hex-vertex with generator $g$*
**for** $(f', c) \sim c$ s.t. $f' \neq f$ and $g \sim f'$ **do**
  **if** $g \in E$ **then** *//(EC): Edge-cell-propeller, inFirstCell*
    $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \leftarrow \boldsymbol{f}_c((f', c))$ ordered s.t. $\boldsymbol{f}_c(g) = \{\boldsymbol{u}, \boldsymbol{v}\}$ *//halfface parameters*
    $\text{ori}_{d_2} \leftarrow \text{ORI3D}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{z} + \vec{\boldsymbol{d}}_1, \boldsymbol{z} + \vec{\boldsymbol{d}}_2)$
    **if** $\text{ori}_{d_2} = 0$ **then** *//face cuts through generator edge*
      **return** $(f', c)$ *//both faces incident to $g$ would be ok*
    $\text{ori}_w \leftarrow \text{ORI3D}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{z} + \vec{\boldsymbol{d}}_1, \boldsymbol{w})$
    **if** $\text{ori}_{d_2} = \text{ori}_w$ **then** *//face lies on correct side, see (EC)*
      **return** $(f', c)$
  **else** *//(V): Vertex-generator*
    $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \leftarrow \boldsymbol{f}_c((f', c))$ ordered s.t. $\boldsymbol{f}_c(g) = \boldsymbol{u}$ *//halfface parameters*
    $\text{ori}_1 \leftarrow \text{ORI3D}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{u} + \vec{\boldsymbol{d}}_1, \boldsymbol{u} + \vec{\boldsymbol{d}}_2)$
    $\text{ori}_2 \leftarrow \text{ORI3D}(\boldsymbol{w}, \boldsymbol{u}, \boldsymbol{u} + \vec{\boldsymbol{d}}_1, \boldsymbol{u} + \vec{\boldsymbol{d}}_2)$
    **if** $\text{ori}_1 = \text{ori}_2$ **then** *//Rotation goes through inner part of face triangle*
      **return** $(f', c)$ *//(both oris must be $< 0$)*
    **if** ($\text{ori}_1 = 0$ and $\text{ori}_2 < 0$) or ($\text{ori}_1 < 0$ and $\text{ori}_2 = 0$) **then** *//Rotation goes through edge $uv$ or $uw$*
      **return** $(f', c)$

## 5.2 Connecting Opposite Propellers

With the local topology having been extracted, the only thing left to do is to find the adjacency relations between the hex-vertices. This is achieved by tracing the propellers from each hex-vertex through potentially multiple cell charts in the parametrization mesh as seen in Algorithm 12, similarly to how the propellers were connected to their blades. To ensure finding the opposite blade for the connection offset according to Section 3.2, pickNextHalfaceToOpposite (Algorithm 13) picks a face $f$ of the current cell $c$, excluding the face through which $c$ was entered, which satisfies $\mathcal{E}(\boldsymbol{u}, \vec{\boldsymbol{d}}_1) \cap \boldsymbol{F}_c(f) \neq \emptyset$ and $\mathcal{F}(\boldsymbol{u}, \vec{\boldsymbol{d}}_1, \vec{\boldsymbol{d}}_2) \cap \boldsymbol{F}_c(f) \neq \emptyset$. To find a hex-vertex that has a given parameter in the chart of a given cell in constant time as required on Line 7 of Algorithm 12, a hash map is used.

**Algorithm 12** *connectPropellerOpposite:* Tracing from a hex-vertex and propeller to the opposite vertex and propeller

**Input**
  $\boldsymbol{v_h}$ Hex-vertex on some generator $g$
  $\boldsymbol{p_1}$ Propeller on the same generator

1: **if** opposite[$v_h$, $p$] is defined **then** *//p already connected?*
2:  **return**
3: $p_2 \leftarrow$ blades[$p_1$][0]
4: $c$, $\vec{\boldsymbol{d}}_2 \leftarrow$ bladeDirs[$p_1$][0][0]
5: $\vec{\boldsymbol{d}}_1 \leftarrow \vec{\boldsymbol{d}}_c(p_1)$
6: $\boldsymbol{z} \leftarrow \boldsymbol{f}_c(v_h)$
7: $v'_h \leftarrow$ hex-vertex s.t. $\boldsymbol{f}_c(v'_h) = \boldsymbol{z} + \vec{\boldsymbol{d}}_1$ *//look for opposite vertex in chart of c*
8: $(f, c) \leftarrow$ invalid halfface
9: **while** $v'_h$ does not exist **do**
10:  $(f, c) \leftarrow$ pickNextHalffaceToOpposite($(f, c)$, $c$, $\boldsymbol{z}$, $\vec{\boldsymbol{d}}_1$, $\vec{\boldsymbol{d}}_2$) *//transition face*
11:  **if** $f \in \partial F$ **then** *//traced into boundary?, happens if (IGM2) is not satisfied*
12:  **return** *//treat as error or simply leave ($v_h$, $p$) without opposite*
13:  $\boldsymbol{\tau} \leftarrow \boldsymbol{\tau}_{(f,\,c)}$ *//transition function from the chart of c through f*
14:  $\boldsymbol{z}$, $\vec{\boldsymbol{d}}_1$, $\vec{\boldsymbol{d}}_2 \leftarrow \boldsymbol{\tau}(\boldsymbol{z}, \vec{\boldsymbol{d}}_1, \vec{\boldsymbol{d}}_2)$ *//transition*
15:  $v'_h \leftarrow$ hex-vertex s.t. $\boldsymbol{f}_c(v'_h) = \boldsymbol{z} + \vec{\boldsymbol{d}}_1$ *//look for opposite vertex in chart of c*
16:  $(f, c) \leftarrow (f, c)$.opposite *//move into the chart of the adjacent cell*
17: $p' \leftarrow$ propeller on the generator of $v'_h$ s.t. $\vec{\boldsymbol{d}}_c(p') = -\vec{\boldsymbol{d}}_1$
18: $j \leftarrow$index s.t. $(c, \vec{\boldsymbol{d}}_2) \in$ bladeDirs[$p'$][$j$]
19: opposite[$v_h$, $p$] = ($v'_h$, $p'$, $j$) *//opposite vertex, opposite propeller, connection offset*
20: opposite[$v'_h$, $p'$] = ($v_h$, $p$, $j$)

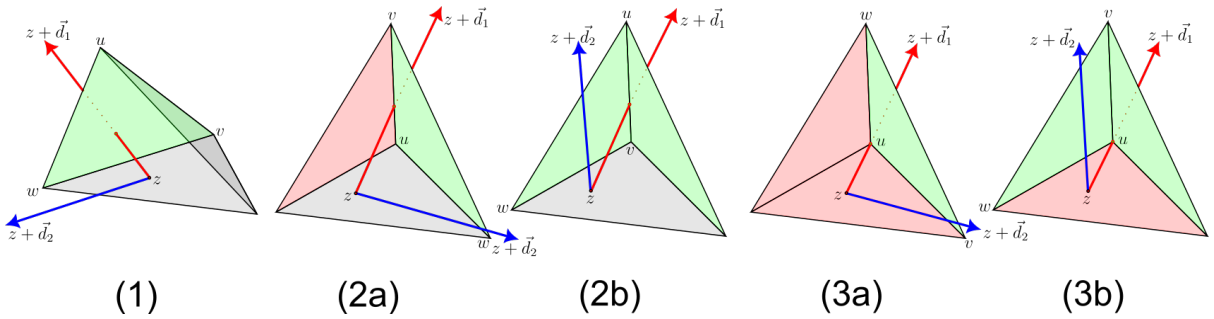



**Figure 17:** *The next tet-cell is entered through the face which intersects the integer-grid edge (red). If we trace through an edge or vertex, the secondary direction (blue) is considered (2,3).*

**Algorithm 13** *pickNextHalfffaceToOpposite*

**Input**

**$(f, c)$** entering halfface. Invalid or incident to $c$

**$c$** current cell chart

**$z$** hex-vertex parameter in the chart of $c$

**$\vec{d}_1, \vec{d}_2$** propeller root and blade directions in the chart of $c$

**Output**

**$(f', c)$** exiting halfface incident to $c$ and not equal to $(f, c)$ which is intersected by the integer-grid edge and by the integer-grid face

**for** $(f', c) \sim c$ s.t. $f' \neq f$ **do**
  $(u, v, w) \leftarrow f_c((f', c))$
  **if** ORI3D$(u, v, w, z) \leq 0$ or ORI3D$(u, v, w, z + \vec{d}_1) \geq 0$ **then**
    **continue** *//Root does not even cut through the face plane*
  $\text{ori}_{uv} \leftarrow$ ORI3D$(u, v, z, z + \vec{d}_1)$ *//get the orientations of the three*
  $\text{ori}_{vw} \leftarrow$ ORI3D$(v, w, z, z + \vec{d}_1)$ *//tets formed around $z$, $z + \vec{d}_1$*
  $\text{ori}_{wu} \leftarrow$ ORI3D$(w, u, z, z + \vec{d}_1)$
  **if** $\text{ori}_{uv} = \text{ori}_{vw} = \text{ori}_{wu}$ **then** *//all oris are $< 0$*
    **return** $(f', c)$ *//(1): Root does cut through inner part of face triangle*
  **if** any of these three oris is positive and any of them is negative **then**
    **continue** *//Root does not cut through the face triangle*
  *//Otherwise, the root cuts through the triangles boundary (edge or vertex)*
  $n \leftarrow (\text{ori}_{uv} = 0) + (\text{ori}_{vw} = 0) + (\text{ori}_{wu} = 0)$ *//Get number of edge intersections. 1 or 2*
  **if** $n = 1$ **then** *//cuts through edge*
    cycle $(u, v, w)$ s.t. $\text{ori}_{uv} = 0$ *//$u$, $v$ is intersected edge*
    ori $\leftarrow$ ORI3D$(u, v, z + \vec{d}_1, z + \vec{d}_2)$
    **if** ori$= 0$ **then** *//Integer-grid plane cuts through edge*
      **return** $(f', c)$ *//(2b): both faces incident to edge would be ok*
    **if** ori$=$ ORI3D$(u, v, z + \vec{d}_1, w')$ **then**
      **return** $(f', c)$ *//(2a): integer-grid face intersects triangle face*
    **else** *//(2a): integer-grid face intersects other incident triangle face*
      **return** unique $(f'', c)$ incident to intersected edge s.t. $f' \neq f''$
  **else** *//$n = 2$, (3): cuts through vertex*
    cycle $(u, v, w)$ s.t. $\text{ori}_{uv} = \text{ori}_{wu} = 0$ *//$u$ is intersected vertex*
    **if** ORI3D$(v, u, z, z + \vec{d}_2) < 0$ or ORI3D$(u, w, z, z + \vec{d}_2) < 0$ **then**
      **continue** *//$z$, $z + \vec{d}_2$ is not on correct side*
    **return** $(f', c)$ *//(3a), (3b)*

## 5.3 Extracting the Hex-Mesh

With the hex-vertices defined and all propellers interconnected, all is set for the final step: The extraction of the hex-cells. One iterates over all initially unvisited pairs of hex-vertices and hex-corners on a common generator, for each accessing the seven other hex-vertices and hex-corners of the hex-cell using the opposite and oppositeblade connections as illustrated in Fig. 18 and marking the eight pairs of hex-vertices and hex-corners as visited to not extract a hex-cell multiple times.

The final hex-mesh is then stored in the .MESH format where only vertices and cells are stored explicitly.

A complete overview of the pipeline of Fast HexEx is given in Algorithm 14.

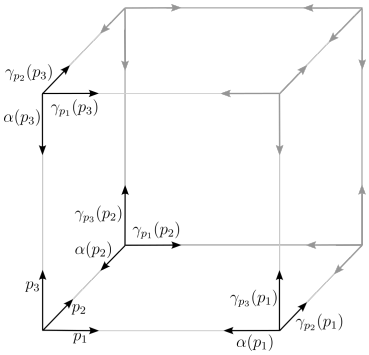



**Figure 18:** *From a corner $(p_1, p_2, p_3)$, the adjacent corners can be accessed via the opposite ($\alpha$) and oppositeblade ($\gamma$) connections.*

**Algorithm 14** Complete Hex Extraction Pipeline

**Input**
$\boldsymbol{\mathcal{M} = (V, E, F, C)}$ Manifold tetrahedral mesh
$\boldsymbol{f}$ Positively relaxed integer-grid-map
**Output**
$\boldsymbol{\mathcal{M}_h = (V_h, C_h)}$ Manifold hexahedral mesh

```
extractTransitionFunctions()
sanitizeParametrization()

G ← ∅
for x ∈ V ∪ E ∪ F ∪ C do                                   //order: V, E, F, C
    extractHexVertices(x)                                   //Algorithm 1

for g ∈ G \ C do                                           //implicit order: V, E, F
    enumeratePropellerRoots(g)                              //Algorithm 7
    for p ∈ P_g do
        enumerateDirectionsToBlades(p)                      //Algorithm 8
    for p ∈ P_g do
        connectPropellerBlades(p)                           //Algorithm 9
    enumerateHexCorners(g)                                  //Section 5.1.3

for v_h ∈ V_h do
    g ← generator of v_h
    for p ∈ P_g do
        connectPropellerOpposite(v_h, p)                    //Algorithm 12

extractHexCells()                                          //Fig. 18
```

## 5.4 Implementation

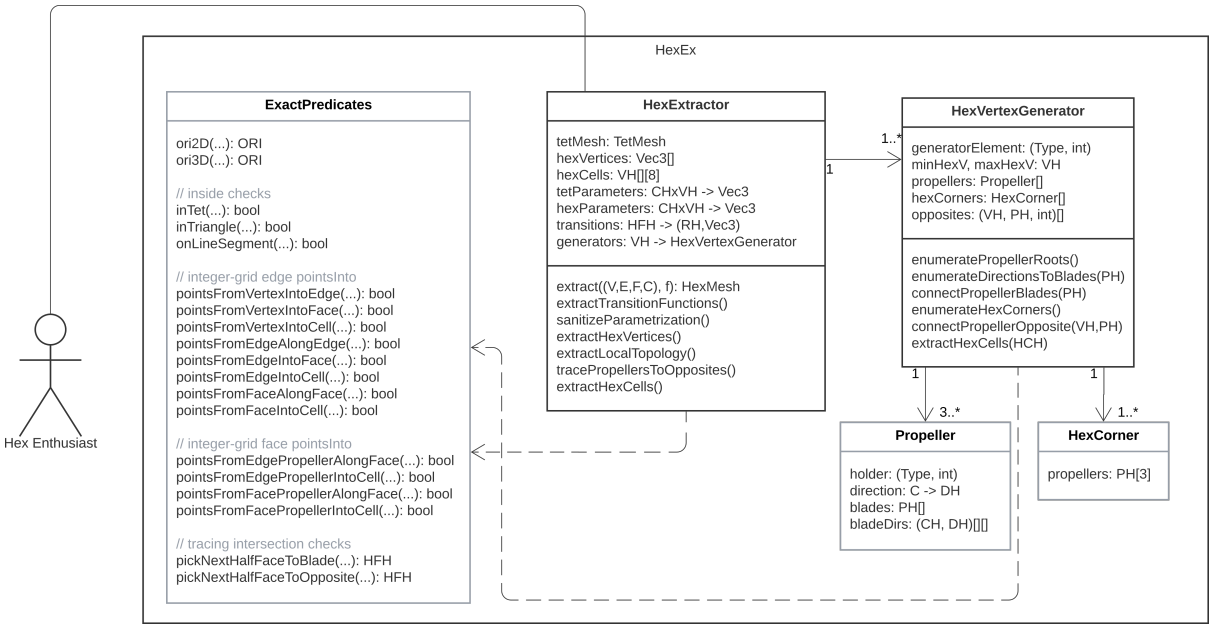



**Figure 19:** *Simplified UML class diagram of Fast HexEx. To access entities, type specific handles are used which include: VH (Vertex), HFH (HalfFace), CH (Cell), RH (Rotation), DH (Direction), PH (Propeller), HCH (HexCorner).*

The vertex- and topology-extraction algorithms have been presented in detail, and we now briefly discuss how some of the data structures are stored and accessed. Like the entities of OVM (vertices, (half-)edges, (half-)faces and cells), our propellers and hex-corners are index based which means, they are stored in an array and accessing them is achieved via special handle objects which are just integers, with additional type safety, corresponding to the index of the object in the respective array. This means that, for example, instead of opposite and blade connections of propellers being pointers or references, they are propeller handles.

Similarly, the six possible coordinate permutations which are used in the vertex-extraction, the six axis aligned directions and the twenty-four orientation preserving rotations are globally stored and their respective handle objects are single bytes. For example, a direction handle object with index 0 refers to $\vec{e}_1$. Their inverses and opposites are hard coded and globally stored for quick access.

To access hex-vertices with a certain parameter in the chart of a certain cell, as required during tracing, we use hash maps, one per tet-cell. The hash map of a certain cell is quickly accessed using a cell-handle and the hex-vertex in the chart is accessed by first mapping our integer parameter onto a non-negative integer, based on a global bounding box around the whole parametrized mesh.

More specifically, each $(x,\, y,\, z) \in [x_{min},\, x_{max}] \times [y_{min},\, y_{max}] \times [z_{min},\, z_{max}] \subset \mathbb{Z}^3$
is mapped to the unique $(z - z_{min}) \cdot S_y \cdot S_x + (y - y_{min}) \cdot S_x + (x - x_{min}) \in \mathbb{N}_0$

where $S = (x_{max} - x_{min} + 1,\ y_{max} - y_{min} + 1,\ z_{max} - z_{min} + 1)$ are the extents of the parameter bounding box.

This allows us to use the trivial hash of integers instead of relying on the hash of 3d vectors. Arrays that are known to be small, like the array of propellers per generator, are linearly looked through.

An illustration of the structure of Fast HexEx is presented with Fig. 19.

The opposite of a propeller is stored in a separate array of the generator rather than in the propeller object itself since the propellers are only explicitly stored for one hex-vertex on the generator as explained before. Consider some generator with $n$ extracted hex-vertices (with indices from $v_{min}$ to $v_{max}$, meaning $n = v_{max} - v_{min} + 1$) and $m$ propellers per hex-vertex. Then, only $m$ propellers are enumerated (with indices from 0 to $m - 1$), even though there are $m \cdot n$ propellers in total. The same applies to the array of hex-corners. Propeller 0 then contains the common data of the actual $n$ propellers on the $n$ hex-vertices which all point in the same direction in an incident cell chart. However, unlike the directions and blade indices, the opposite propeller differs per hex-vertex, so it is stored separately. The opposite propeller of the propeller with index $i_p \in \{0, ..., m-1\}$ on the hex-vertex with index $i_v \in \{v_{min}, ..., v_{max}\}$ is then the $(n \cdot i_p + (i_v - v_{min}))$-th element in the opposites array.

For cell-generators, the array of propellers stays empty and the propeller handle with index $i \in \{0, ..., 5\}$ implicitly refers to a propeller pointing in the direction given by the same index $i$.

This system leads to our goal of minimal data redundancy.

# 6 Results

Now, we present and discuss some results of our algorithm on a range of examples of the HexMe dataset (Beaufort et al. 2022) shown in Fig. 22 and compare it to the original method.
The performance was evaluated using the command line tool of HexEx. Note that the timings of reading and writing input and output files are excluded. A small performance increase in the preprocessing was achieved by preliminary assuming that (IGM4) is satisfied.

## 6.1 Timings

| Model | #T | #H | $\frac{\#H}{\#T}$ | Robust | Fast | $\frac{\text{Fast}}{\text{Robust}}$ |
|---|---|---|---|---|---|---|
| s09u_bridge | 100130 | 4590 | 5% | 2.04s | 0.91s | 44% |
| s10u_cuthemicylinder | 28217 | 4986 | 18% | 1.26s | 0.33s | 26% |
| s13u_roundcube | 41179 | 5312 | 13% | 1.40s | 0.44s | 31% |
| s15c_cylinder | 28246 | 5238 | 19% | 1.19s | 0.30s | 25% |
| s17c_sphere | 28028 | 4608 | 16% | 1.07s | 0.28s | 26% |
| n03u_skijump | 8782 | 103218 | 1175% | 14.38s | 0.47s | 3% |
| n04c_prism | 7122 | 4233 | 59% | 0.84s | 0.11s | 13% |
| n10u_qtorus | 133239 | 6511 | 5% | 3.03s | 1.31s | 43% |
| i01c_m1 | 137233 | 49520 | 36% | 10.92 | 2.04 | 19% |
| i09u_m9 | 793884 | 7188 | 1% | 12.66s | 8.44s | 67% |
| i11u_s1 | 265131 | 4810 | 2% | 4.30s | 2.53s | 59% |
| i14c_s7 | 47107 | 5691 | 12% | 1.68s | 0.55s | 33% |
| i18c_s22 | 73118 | 5465 | 7% | 1.99s | 0.78s | 39% |
| i22c_s27 | 77162 | 6248 | 8% | 2.22s | 0.89s | 40% |

**Table 2:** *Comparison of the old method of Robust HexEx and our new method of Fast HexEx. #T and #H are the number of tetrahedral cells in the input mesh and the number of hexahedral cells in the output mesh respectively. The higher the ratio of hex-elements to tet-elements, the better our method compares to the original.*

Fast HexEx outperformed Robust HexEx in all examples and optimized tasks. The results are shown in Table 2 and in Fig. 20 which also shows the impacts of the individual tasks. The most noticeable difference can be observed in the propeller tracing, i.e. the extraction of hex-vertex adjacencies. As we do not create an OpenVolumeMesh, which would require a lot of expensive topology checks, and instead represent the extracted hex-mesh by a list of vertex positions and cells, the cell extraction is practically negligible. Furthermore, as explained before, the postprocessing can be omitted entirely. As a result, our new method is consistently faster than the original one, especially for parametrizations that result in a hex-mesh with a lot of elements. In coarse examples, the preprocessing phase is dominant, restricting the possible speedup.

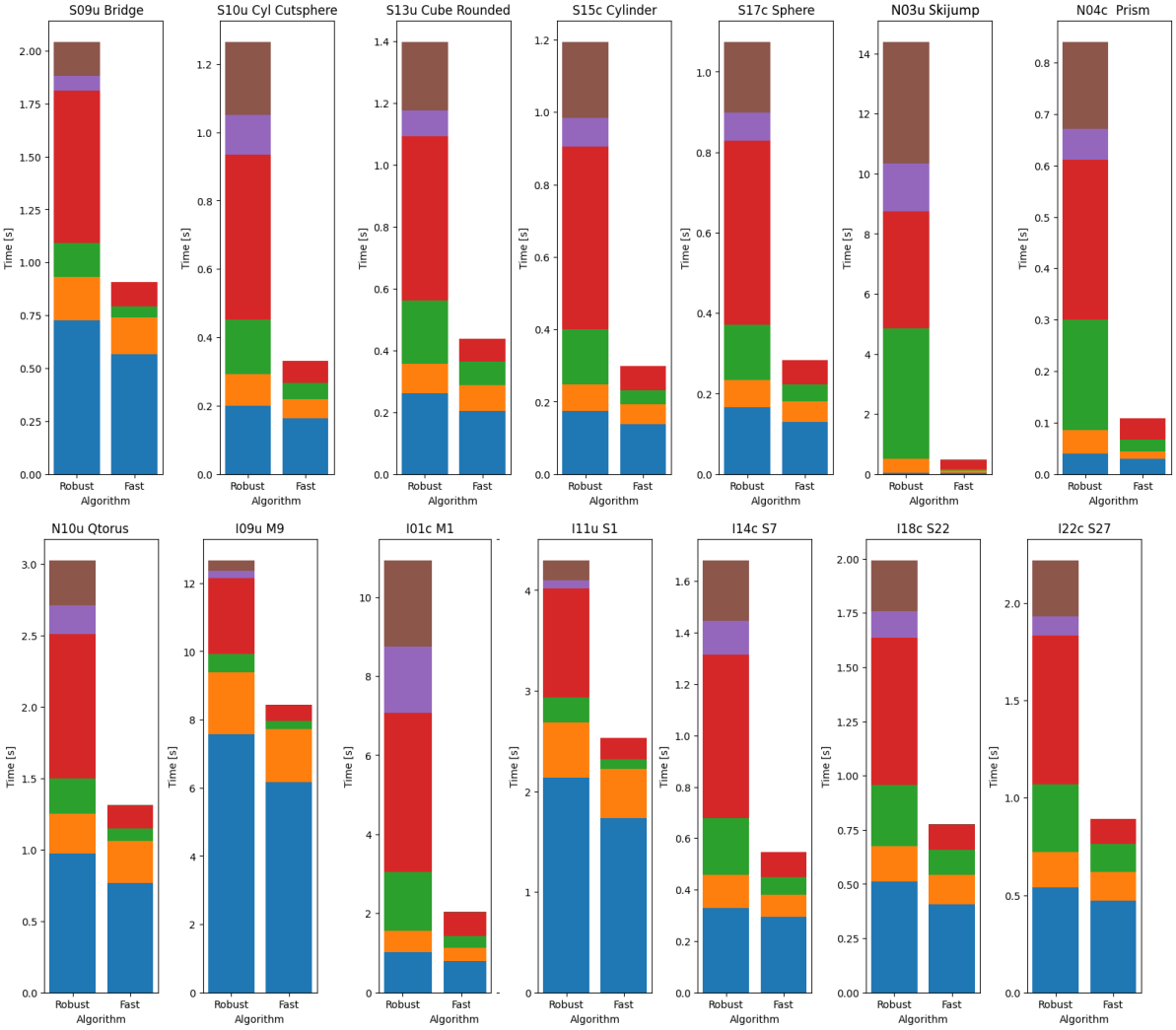



**Figure 20:** *Performance comparison of the original Robust HexEx and the new Fast HexEx on the meshes shown in Fig. 22. The individual tasks are marked and stated in Table 3.*

| Plot color | Task | Robust HexEx | Fast HexEx |
|---|---|---|---|
| | Preprocessing | extract transitions, sanitize IGM | extract transitions, sanitize IGM |
| | Vertex Extraction | hex-vertex extraction (bounding box) | hex-vertex extraction (rasterization) |
| | Local Connections | enumerate darts, connect $\alpha_1$, $\alpha_2$, $\alpha_3$ | enumerate propellers, connect blades |
| | Tracing Connections | connect $\alpha_0$ | connect opposites |
| | Postprocessing | dart annihilation, vertex merging | - |
| | Cell Extraction | add hex-cells to OVM | create list of hex-cells |

**Table 3:** *The common tasks of Robust HexEx and Fast HexEx.*

## 6.2 Complexity Scaling

There are two main factors that contribute to the performance of the algorithm, namely, the number of tet-elements and the number of hex-elements induced by the IGM. Unsurprisingly, the impact of the former remains linear, however, Fast HexEx is especially efficient for inputs where the latter is relatively large. The more refined the parametrization, the more we profit from the rasterization in the hex-vertex extraction and the local topology extraction per generator instead of per hex-vertex. This is not surprising as in such parametrizations, parametric volumes of tets contain multiple integer-grid points. Conversely, for coarse parametrizations where most parametrized tets contain few or no integer-grid points, these optimizations are less impactful and most time is spent in preprocessing.

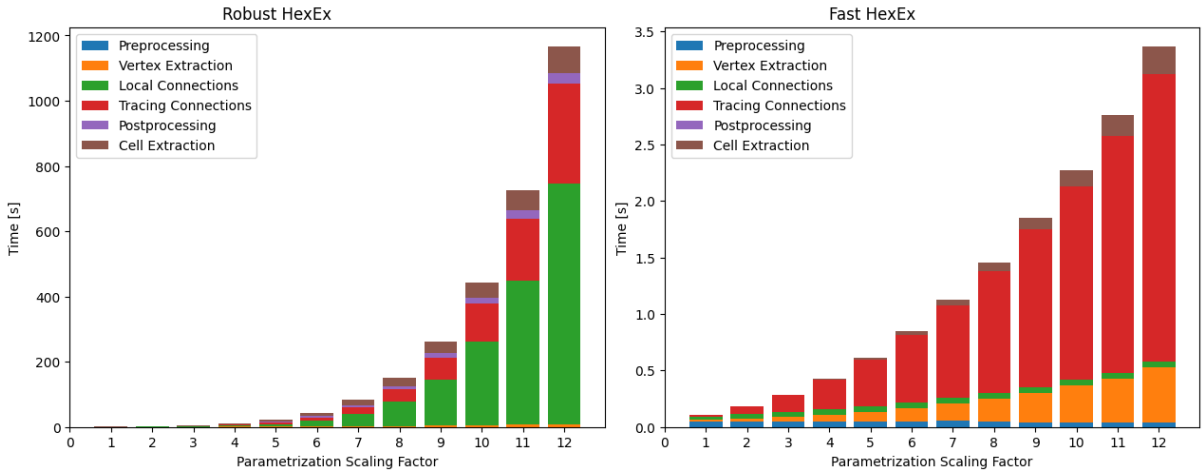



**Figure 21:** *Comparison of the runtimes for a model with increasingly more hex-elements. The original model contains 8782 tet-cells. With an unscaled parametrization (scaling factor 1), the resulting hex-mesh contains 1256 hex-cells. A scaling factor of 12 leads to a hex-mesh with* $12^3 \cdot 1256 = 2170368$ *hex-cells. Note the significant difference in scale.*

Figure 21 highlights our methods capability of handling inputs with increasingly more hex-elements per tet-element. For the initial parametrization ($\frac{\#\mathrm{H}}{\#\mathrm{T}} = 14\%$), Robust HexEx took 0.4 seconds and Fast HexEx took 0.1 seconds. Then, for the upscaled parametrization with $12^3$ as many hex-elements ($\frac{\#\mathrm{H}}{\#\mathrm{T}} = 24714\%$), Robust HexEx took nearly 20 minutes whereas Fast HexEx took 3.3 seconds.

Even when ignoring such extreme cases, the inherent benefits of the propeller data structure compared to the dart data structure (less propellers than darts and only two types of connections instead of four) are advantages applicable to all valid inputs.

## 6.3 Runtime Analysis

Table 4 shows the (asymptotic) runtimes of the newly implemented tasks in detail. Let $V$, $E$, $F$, $C$ be the number of input vertices, edges, faces and cells respectively and $M = V + E + F + C$ the number of tet-elements in total. Furthermore, we use $L = \sqrt[3]{\frac{C_h}{C}}$ as a measure of the parametrization fineness where $C_h$ is the number of resulting hex-cells. While the runtimes of the hex-vertex extraction from vertices and edges remain unchanged[6], the rasterization of the faces and cells results in a reduced runtime by a factor of $L$ compared to checking every point in the bounding box[7].
As propellers are enumerated only on a single hex-vertex per generator, providing most propellers implicitly, both the propeller enumeration and the connecting of blades are constant in $L$. On the other hand, all darts are explicitly enumerated, so the runtime is cubic in $L$. Moreover, to find a specific dart in the chart of a tet-cell, the list of all darts in the chart is iterated linearly, i.e. cubic in $L$, compared to our use of a hash map, which is why all the connection algorithms for the darts are in $\mathcal{O}(ML^3 \cdot L^3)$.

| Task | Robust HexEx | Fast HexEx |
|---|---|---|
| extract hex vertices (vertices) | $\mathcal{O}(V)$ | $\mathcal{O}(V)$ |
| extract hex vertices (edges) | $\mathcal{O}(EL)$ | $\mathcal{O}(EL)$ |
| extract hex vertices (faces) | $\mathcal{O}(FL^3)$ | $\mathcal{O}(FL^2)$ |
| extract hex vertices (cells) | $\mathcal{O}(CL^3)$ | $\mathcal{O}(CL^2)$ |
| enumerate darts/propellers | $\mathcal{O}(ML^3)$ | $\mathcal{O}(M)$ |
| connect $\alpha_1$, $\alpha_2$, $\alpha_3$/blades | $\mathcal{O}(ML^6)$ | $\mathcal{O}(M)$ |
| connect $\alpha_0$/opposites | $\mathcal{O}(ML^6)$ | $\mathcal{O}(ML^3)$ |

**Table 4:** *Asymptotic runtimes of the individual tasks of Robust HexEx and Fast HexEx.*

[6]If an edge is integer-aligned, the runtime is practically constant in $L$ as no exact predicates are required.

[7]If a face is integer-aligned, the runtime is reduced by a factor of $L^2$ instead of $L$ because all scanlines are integer-aligned.

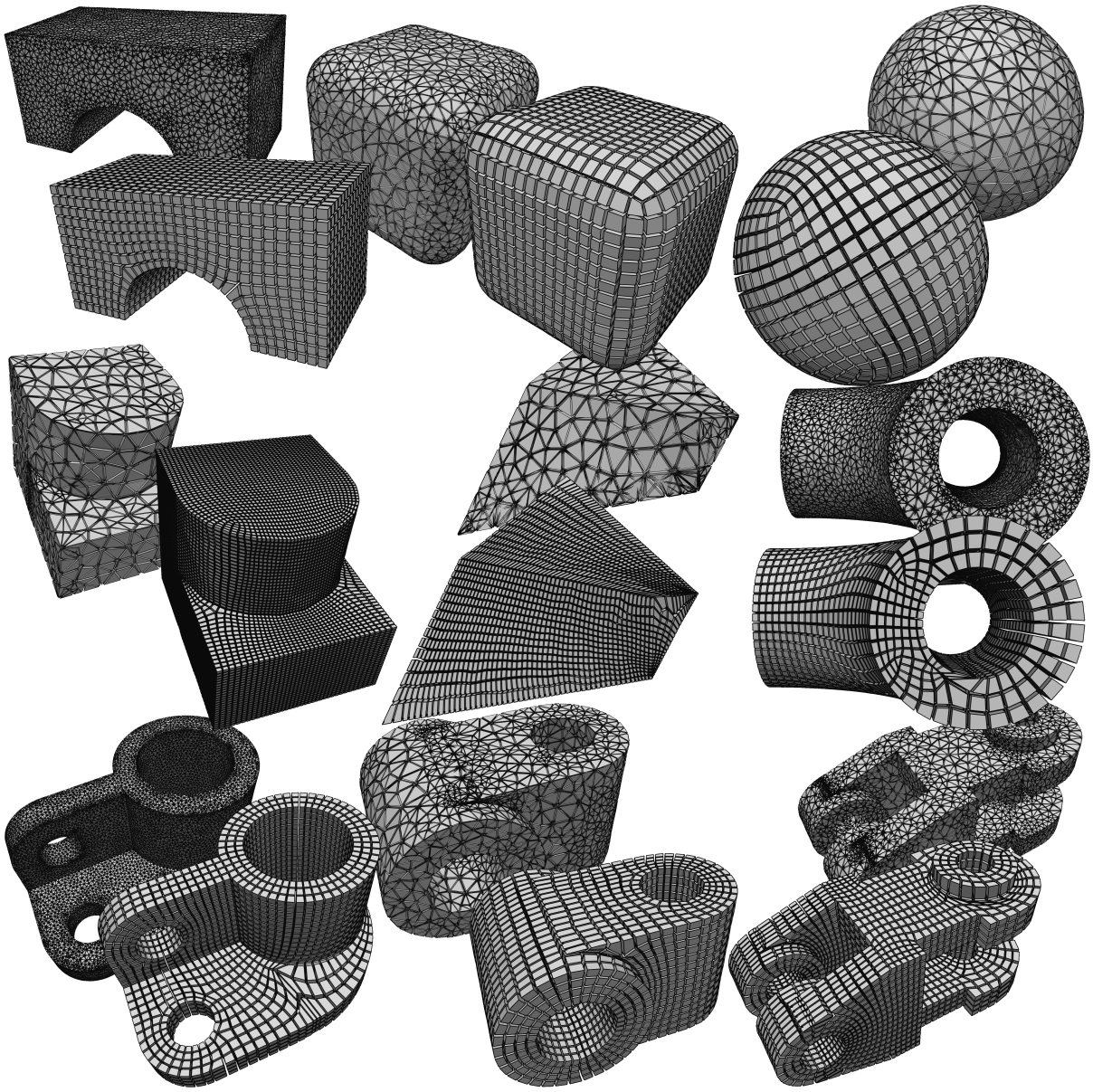


**Figure 22:** *Example tet-meshes of the HexMe dataset and their respective hex-meshes produced by Fast HexEx, visualized in OpenFlipper. From top left to bottom right: s09u_bridge, s13u_roundcube, s17c_sphere, n03u_skijump, n04c_prism, n10u_qtorus, i11u_s1, i14c_s7, i22c_s27.*

# 7 Conclusion

We presented Fast HexEx, an optimization of Robust HexEx by (Lyon et al. 2016) for reliably extracting hex-meshes from tet-meshes with positively relaxed integer-grid maps. Using rasterization techniques to find integer-grid points in the parametrization and the new propeller data structure to define the resulting topology, we achieved a noticeable performance improvement of HexEx, particularly for fine parametrizations.

## 7.1 Discussion

Lastly, we present a few ideas on how to further improve Fast HexEx:
1) Currently, we iterate all elements separately to avoid duplicate hex-vertices. Instead, when evaluating the normal ORI3D checks for a tetrahedron, we could already determine if the hex-vertex lies inside the tet, a face, an edge or a vertex. To avoid duplicates, the property system of OVM might be used to store simple flags. This would allow us to extract hex-vertices by only iterating the cells which could significantly improve the runtime of the geometry extraction.
2) Certain tasks like the hex-vertex extraction or the local topology extraction could profit from parallelization as they consist of multiple independent routines.
3) Non-identity transitions between charts of adjacent cells in the parametrization cause the topology extraction to be considerably more complex than for a globally injective parametrization with only identity transitions. However, as in practice, non-identity transitions predominantly occur around singularities, with them making up only a small portion of all transitions (mostly $\approx 5\%$), it might be worth detecting and storing trivial regions in the form of blocks of adjacent cells in the parametrization beforehand and then operate on the charts of the whole regions instead of the individual cells. This would both trivialize the local topology extraction of hex-vertices in such regions and simplify tracing while rasterizing them would be challenging.

# References


Beaufort, Pierre-Alexandre, Maxence Reberol, D. Kalmykov, H. Liu, Franck Ledoux, and D. Bommes (Oct. 2022). "Hex Me If You Can". In: *Computer Graphics Forum* 41, pp. 125–134. DOI: 10.1111/cgf.14608.

Botsch, Mario, Stefan Steinberg, Stefan Bischoff, and Leif Kobbelt (Feb. 2002). "OpenMesh: A Generic and Efficient Polygon Mesh Data Structure". In: URL: https://api.semanticscholar.org/CorpusID:15631664.

Brückler, Hendrik, David Bommes, and Marcel Campen (July 2022). "Volume parametrization quantization for hexahedral meshing". In: *ACM Transactions on Graphics* 41, pp. 1–19. DOI: 10.1145/3528223.3530123.

Campagna, Swen, Leif Kobbelt, and Hans-Peter Seidel (1998). "Directed edges—a scalable representation for triangle meshes". In: *Journal of Graphics tools* 3.4, pp. 1–11.

Ebke, Hans-Christian, David Bommes, Marcel Campen, and Leif Kobbelt (2013). "QEx: Robust quad mesh extraction". In: *ACM Transactions on Graphics (TOG)* 32.6, pp. 1–10.

Grünbaum, Branko and Geoffrey C Shephard (1969). "Convex polytopes". In: *Bulletin of the London Mathematical Society* 1.3, pp. 257–300.

Kraemer, Pierre, Lionel Untereiner, Thomas Jund, Sylvain Thery, and David Cazier (Jan. 2014). "CGoGN: N-dimensional Meshes with Combinatorial Maps". In: ISBN: 978-3-319-02334-2. DOI: 10.1007/978-3-319-02335-9_27.

Kremer, Michael, David Bommes, and Leif Kobbelt (2013). "OpenVolumeMesh–A versatile index-based data structure for 3D polytopal complexes". In: *Proceedings of the 21st International Meshing Roundtable*. Springer, pp. 531–548.

Liu, Heng, Paul Zhang, Edward Chien, Justin Solomon, and David Bommes (July 2018). "Singularity-Constrained Octahedral Fields for Hexahedral Meshing". In: *ACM Trans. Graph.* 37.4. ISSN: 0730-0301. DOI: 10.1145/3197517.3201344. URL: https://doi.org/10.1145/3197517.3201344.

Lyon, Max, David Bommes, and Leif Kobbelt (July 2016). "HexEx: Robust Hexahedral Mesh Extraction". In: *ACM Trans. Graph.* 35.4. ISSN: 0730-0301. DOI: 10.1145/2897824.2925976. URL: https://doi.org/10.1145/2897824.2925976.

Möbius, Jan and Leif Kobbelt (2012). "OpenFlipper: An Open Source Geometry Processing and Rendering Framework". In: *Curves and Surfaces*. Ed. by Jean-Daniel Boissonnat, Patrick Chenin, Albert Cohen, Christian Gout, Tom Lyche, Marie-Laurence Mazure, and Larry Schumaker. Vol. 6920. Lecture Notes in Computer Science. 10.1007/978-3-642-27413-8_31. Springer Berlin / Heidelberg, pp. 488–500. ISBN: 978-3-642-27412-1. URL: http://dx.doi.org/10.1007/978-3-642-27413-8_31.

Pietroni, Nico et al. (Oct. 2022). "Hex-Mesh Generation and Processing: A Survey". In: *ACM Transactions on Graphics* 42.2, pp. 1–44. DOI: 10.1145/3554920. URL: https://doi.org/10.1145%2F3554920.

Reberol, Maxence, Kilian Verhetsel, François Henrotte, David Bommes, and Jean-François Remacle (Mar. 2023). "Robust Topological Construction of All-Hexahedral Boundary Layer Meshes". In: *ACM Trans. Math. Softw.* 49.1. ISSN: 0098-3500. DOI: 10.1145/3577196. URL: https://doi.org/10.1145/3577196.

Ruiz, Antonio J. Rueda, Rafael Jesús Segura, Francisco R. Feito-Higueruela, Juan Ruiz de Miras, and Carlos Javier Ogáyar (2004). "Voxelization of Solids Using Simplicial Coverings". In: *International Conference in Central Europe on Computer Graphics and Visualization*. URL: https://api.semanticscholar.org/CorpusID:15347116.

Shewchuk, Jonathan (Mar. 1996). "Robust Adaptive Floating-Point Geometric Predicates". In: *Proceedings of the Annual Symposium on Computational Geometry*. DOI: 10.1145/237218.237337.

Solomon, Justin, Amir Vaxman, and David Bommes (July 2017). "Boundary Element Octahedral Fields in Volumes". In: *ACM Trans. Graph.* 36.4. ISSN: 0730-0301. DOI: 10.1145/3072959.3065254. URL: https://doi.org/10.1145/3072959.3065254.