D I P L O M A R B E I T

Tetrahedral Mesh Cleaving of Level Set Surfaces

ausgeführt zum Zwecke der Erlangung des akademischen Grades

Diplom-Ingenieur

in

Computational Science and Engineering

unter der Hauptbetreuung von

Associate Prof. Dr.techn. Lado Filipovic

und der Mitbetreuung von

Dr.techn. Xaver Klemenschits, MSc.

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik
von

Cedrik Balla, BSc.

Matrikelnummer: 01604862

Wien, im August 2023 _____

Abstract

In the field of microelectronics, device simulations require meshes which consist of good quality tetrahedra in order to achieve reliable results. Simulations of the fabrication of those devices are often carried out using process simulations which apply an implicit geometry representation, such as the level set (LS) method. Several algorithms exists which, from those implicit representations, generate tetrahedral meshes with various qualities, such as accurate surface representations or high element quality.

This work describes the implementation of an algorithm from the group of lattice-based meshing algorithms, where mesh generation is initiated on a background lattice with tetrahedra of known quality. Using a multi-material LS input, a graded tetrahedral mesh with good dihedral angles is generated.

The presented implementation adapts existing algorithms in order to create meshes from sparse multi-material LS grids, rather than a continuous signed distance function (SDF), or a dense LS grid. The input LSs are Manhattan normalized and use the so called LS wrapping approach, where an additive layer wrapping of the materials is used, in order to preserve thin features and make use of the LS sub grid accuracy during process simulation.

The implemented algorithm consists of modular and exchangeable parts. There are three main parts: the creation of an octree substructure, the creation of a background mesh, and the cleaving of the background mesh. The octree creation is composed of an initial loading of LS input data and a module for grading its interior. The background mesh is created using the octree by employing a crystal lattice stencil module. This should ease further development of the algorithm, by making it possible to develop drop-in replacements for key parts of the algorithm. Therefore the presented implementation allows for the used body centered cubic (BCC) background lattice to be replaced by another background lattice in the future. Using alternative background lattices can potentially increase the quality of the produced tetrahedra.

Kurzfassung

Im Bereich der Mikroelektronik brauchen Bauteilsimulationen Gitter aus guten Tetraedern um verlässliche Ergebnisse zu liefern. Simulation des Herstellungsprozesses dieser Bauteile werden oftmals durch Prozesssimulationen unter Anwendung impliziter Geometriedarstellungen wie der Level-Set (LS) Methode gemacht. Es existieren einige Algorithmen, welche von solchen impliziten Darstellungen tetraedrische Gitter mit unterschiedlichen Qualitäten erzeugen. Zu diesen Qualitäten gehören unter anderem eine akkurate Wiedergabe der Oberflächen oder Elemente von hoher Güte.

Die vorliegende Arbeit beschreibt die Implementierung eines solchen Algorithms, aus der Gruppe der Kristallgitter-basierten Gittererzeugungsalgorithmen. Diese Algorithmen initiieren die Gittererzeugung mit einem Hintergrundgitter basierend auf einem Kristallgitter, welches eine bekannte Güte aufweist. Ausgehend von multi-materiellen Level-Set Daten wird ein Gitter erzeugt, dessen Elemente im inneren größer werden und welches gute Diederwinkel aufweist.

Die präsentierte Implementierung adaptiert existierende Algorithmen, um Gitter ausgehen von multi-materiellen dünn besetzten LSs zu erzeugen, anstatt ausgehend von einer vorzeichenbehaftete Abstandsfunktion oder einem dicht besetzten LS. Die verwendeten LSs sind dabei Manhattan normalisiert, außerdem wird der sogenannte LS Wrapping Ansatz verwendet. Bei diesem Ansatz werden die einzelnen Materialschichten additiv umeinander gelegt, um dünne Schichten und Regionen zu bewahren und um Subgridgenauigkeit in der Prozesssimulation zu nutzen zu können.

Der implementierte Algorithmus ist modular gestaltet und aus einzelnen austauschbaren Teilen aufgebaut. Es gibt drei Hauptteile: das generieren einer Octree Substruktur, die Erzeugung einer Hintergrundgitters und das Mesh Cleaving des Hintergrundgitters. Die Generierung des Octrees setzt sich aus dem Laden der LS Eingangsdaten und einem Modul zu graduellen Füllung des Octree zusammen. Das Hintergrundgitter wird ausgehend von einem Octree und einem Modul zur Anwendung der Schablonen erzeugt. Die Schablone basiert dabei auf einem Kristallgitter. Der modulare Aufbau des Algorithmus sollte seine Weiterentwicklung begünstigen, in dem er es ermöglicht in der Zukunft einzelne Komponenten, unabhängig von

einander, leicht auszutauschen. Daher erlaubt die präsentierte Implementierung auch, das verwendete kubisch-raumzentrierte (BCC) Kristallgitter durch andere Kristallgitter zu ersetzen. Alternative Kristallgitter erhöhen eventuell die Güte der erzeugten Tetraeder.

Acknowledgement

First and foremost, I want to thank Associate Prof. Dr.techn. Lado Filipovic for agreeing to supervise this Thesis. I'm so grateful for the warm welcome and all the support I've received through him. He always had an open ear and was quick to reply. I especially also want to thank him for his tremendous support in regards to organization, when completion was near.

A big thank you to Dr. Xaver Klemenschits for the seemingly endless discussions about this project and C++ in general. Not only for all the valuable input, but also for his encouragement towards improvement in all regards, be it personal or professional. May his believe that sharing knowledge, teaching and motivating others should be paramount to all undertakings, inspire as many people as it possibly can.

A huge thank you to Global TCAD Solutions GmbH (GTS) for all the financial and technical support of this project. The work environment at GTS is brilliant and so I also want to thank the entirety of the GTS staff for their support, for all the talks both on- and off-topic, the laughs shared, their inspiration, their motivation. The way in which they uphold values like curiosity, mutual help, sharing of experiences and thoughts, and cooperation, is truly remarkable. Special mentions have to go out to Gerhard Rzepa, who is an everlasting source of good mood and motivation.

Thank you to my family, especially my parents Claudia and Markus Balla. Their endless support of all my endeavors, through all struggles and all great times shared, means the world to me.

Finally I would like to thank everyone involved in making all this possible, everyone lending motivation when times were tough, everyone who shared knowledge with me, or gave other much appreciated support during all my studies. Even though most of them have to remain nameless here, my deepest gratitude goes out to each and everyone of them.

This work was funded by Global TCAD Solutions GmbH (GTS).

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, im August 2023

Cedrik Balla

Contents

List of Figures

List of Tables

List of Abbreviations

2D	Two-dimensional
3D	Three-dimensional
BCC	Body centered cubic
FDM	Finite difference method
FEM	Finite element method
FET	Field-effect transistor
FVM	Finite volume method
GTS	Global TCAD Solutions GmbH
HRLE	Hierarchical run-length encoded
ID	Identifier
LS	Level set
LSF	Level set function
PC	Personal computer
PDE	Partial differential equation
RAM	Random-access memory
SDF	Signed distance function or signed distance field
TCAD	Technology computer aided design

Chapter 1

Introduction

For today's ever-increasing need for simulations, a vast amount of partial differential equations (PDEs) need to be solved. Due to the difficulty in solving PDEs analytically, in many cases they are solved numerically. Solving PDEs numerically, is often done by employing one of three schemes - the finite difference method (FDM), the finite element method (FEM), or the finite volume method (FVM). Each of these has distinct advantages and restrictions. The FDM can usually only be employed on rectilinear grids. Since this is not sufficient to explicitly represent complex microelectronic devices, it will not be further considered in regards to output meshes throughout this work. The other two methods, FEM and FVM, both require a mesh on which to operate. This mesh serves as a discretization of the domain on which the PDEs are to be numerically solved. The elements of such meshes can vary depending on the problem at hand and the used variations of the mentioned methods. The main aspects of the elements are their geometric shape, the points used within them, and the connectivity between the individual elements. One example of such elements are tetrahedra, which are the simplest three-dimensional (3D) complex and thus they are used in this work.

In microelectronics, the level set (LS) method is often employed for process simulation and emulation which aims to reproduce fabrication processes [1]–[4]. The LS method is based on implicit representations of material interfaces, while the mentioned techniques for solving PDEs numerically, FEM and FVM, are both executed on explicit volumetric representations of the to be simulated material domains. Therefore, in order to simulate a microelectronic device, which was designed using process emulation, one needs to convert the implicit interface representations to explicit volumetric material descriptions. This conversion should be computationally efficient while providing accurate explicit representations of the interfaces and retaining certain quality criteria for the mesh elements.

For microelectronic device simulation, the FVM is commonly applied [5]–[7], while also FEM simulations are sometimes used [8], [9]. For simulations employ-

ing the FVM to be reliable, it is essential that their meshes fulfill the so-called Delaunay criterion. The meshing technique that is used for this work's implementation was, however, conceptualized for creating high quality meshes for FEM simulations, whose mesh quality requirements are not as stringent as the Delaunay criterion. This works is therefore envisioned as a first steps in investigating the feasibility or better adaptability of the meshing techniques commonly used for FEM to be applied to meshing for FVM simulations in the microelectronics sector.

1.1 Motivation and Research Goals

This work aims to implement a meshing algorithm based on the existing mesh cleaving algorithm [10] - see Chapter 3 for more details on related works. Using a given LS input, the algorithm should create a conforming tetrahedral mesh, with good dihedral angles, meaning dihedral angles that are as close as possible to the ones of an equilateral tetrahedron. The algorithm should be implemented in *C++* with industry-orientation in mind, and be the first step in exploring the usability of the mentioned groups of meshing algorithms for the use-cases of technology computer aided design (TCAD) within the microelectronics industry.

The mentioned meshing algorithms offer, most notably, two potential advantages. Firstly, they create the explicit volumetric mesh directly, without creating an intermediate explicit representation of the implicitly-defined input interfaces first. This could be beneficial to runtime performance. The second advantage of those algorithms is that they offer guarantees on the bounds of the dihedral angles created in the mesh. Good dihedral angles usually lead to better and faster convergence in FEM applications [11]. However many microelectronic device simulations employ the FVM and not the FEM approach. Unfortunately, dihedral angles are not a sufficient quality measure for FVM.

To explore the feasibility of this meshing concept, for mesh generation in the mentioned microelectronics context, the first step is to create an implementation which works with input adhering to the LS definition used in process simulations. LSs used in process simulations have specific properties [2], which are considered in this work's implementation. The implementation of the mentioned meshing concept, is the topic of this Thesis. In the future, this implementation can then be used as a basis for evaluating and developing adaptions and extensions to the developed method, required to cater to the FVM and the broad microelectronics sector.

1.2 Outline of the Thesis

Theoretical concepts around the input data, underlying data structures, and the main reasons for the developed algorithm - the mesh quality - are briefly introduced in Chapter 2. This shall serve as a basis for topics covered throughout this work. Following those preliminaries, related works and ideas about meshing are presented in Chapter 3, where different tetrahedral meshing concepts, along with their key similarities and differences are discussed.

The algorithm implemented in this work is introduced in Chapter 4 by giving an overview of the three main components - the octree creation, the background mesh creation, and the mesh cleaving itself, In Section 4.3, Section 4.4, and Section 4.5 these steps are discussed in further detail.

Results and discussion of the performance are provided in Chapter 5, while Chapter 6 concludes the main matter with a quick summary and ideas for future work.

Chapter 2

Theory

This chapter introduces the reader to the basic concepts, underlying this work and their implementation. Firstly, the level set (LS) method (Section 2.1), which is the basis of the input data processed by the meshing algorithm. Secondly, octrees (Section 2.2) and tetrahedral mesh data structures (Section 2.3), used for storing data during the meshing process, and also for the resulting output mesh, are briefly covered. Finally, there is a quick discussion of quality measures for tetrahedral meshes (Section 2.4), which is important to evaluate the meshing algorithm.

2.1 Level Set Method

2.1.1 The Level Set

The level set (LS) method refers to an implicit representation of geometric objects using a scalar field given by a function $\phi : \mathbb{R}^d \to \mathbb{R}$. In literature, the generic symbol f is also often used instead of ϕ, and the function itself, is referred to with various names, such as the *defining function* [12], the *level set function (LSF)* [13], the *cut function* [14], or the *indicator function* [10]. In this work, the term LSF is used throughout to refer to the implicitly defining scalar function $\phi(\mathbf{x})$.

The represented objects, for example lines in two-dimensional (2D) space or surfaces in three-dimensional (3D) space, are implicitly defined, where the LSF $\phi(\mathbf{x})$ has a select constant value c, similar to contour lines or isolines found on topographical maps. By convention, this constant value is commonly selected to be zero, $c = 0$. A surface S is therefore given by the set of all points

$$S = \{\mathbf{x} | \phi(\mathbf{x}) = 0\}. \tag{2.1}$$

An exemplary 2D material domain is depicted in Fig. 2.1, together with contour lines of the LSF and the LSF field itself.

In order to characterize volumetric objects, a consideration must be made regarding what is inside and outside of a volumetric object or domain Ω. The normal vector $\mathbf{n}$, on the object's surface $S = \partial\Omega$ or boundary $\partial\Omega$, is selected to be pointing towards the outside, by convention. Additionally this inside-outside information must be included with the LS surface representation. This is done following the convention, that for any point $\mathbf{x}$ inside the object, the LSF $\phi(\mathbf{x})$ has to be negative. A volumetric object or material domain Ω, including its surface $\partial\Omega$, is therefore implicitly given by the set of points

$$\Omega = \{\mathbf{x} | \phi(\mathbf{x}) \leq 0\}. \tag{2.2}$$

Making the implicit representation even more clear, commonly a so called signed distance function (SDF) $f_{\text{SDF}}(\mathbf{x})$ is directly or indirectly used for the LSF $\phi(\mathbf{x})$. For any point $\mathbf{x}$, an SDF gives the shortest distance between the point $\mathbf{x}$ and the surface $S = \partial\Omega$: not in absolute values, however, but rather as a signed value, with the value's sign carrying the inside-outside information of the point $\mathbf{x}$. The choice of which sign, plus or minus, indicates whether the point $\mathbf{x}$ lies within the region Ω, is up to convention. Following the region Ω's definition in Eq. (2.2), negative values are selected to indicate points lying inside the region. The SDF is hence defined as

$$f_{\text{SDF}}(\mathbf{x}) = \begin{cases} -\min_{\mathbf{p}\in\partial\Omega} \|\mathbf{p} - \mathbf{x}\|, & \text{if } \mathbf{p} \in \Omega \\ +\min_{\mathbf{p}\in\partial\Omega} \|\mathbf{p} - \mathbf{x}\|, & \text{if } \mathbf{p} \notin \Omega \end{cases}. \tag{2.3}$$

Different norms $\|\cdot\|$ can be used to measure the closest distance to the surface, but the choice of the norm has implications beyond which explicit value is assigned to the distance between two points. An obvious choice would be the Euclidean norm or ℓ^2-norm on $\mathbb{R}^d$:

$$\|\mathbf{x}\|_2 := \sqrt{x_1^2 + \cdots + x_d^2} = \sqrt{\mathbf{x} \cdot \mathbf{x}} \tag{2.4}$$

The Manhattan norm or ℓ^1-norm

$$\|\mathbf{x}\|_1 := \sum_{i=1}^{d} |x_i| \tag{2.5}$$

may also be used, which provides algorithmic advantages, when calculating neighboring points in the discretization [2].

2.1.2 Discretization

Due to the complexity of the represented structures, it is inconvenient to represent the LSF analytically. Therefore, the LSF is stored at selected points in space.

The points at which the LSF is stored are commonly defined on a rectilinear grid and efficiently stored in hierarchical run-length encoded (HRLE) data structures [1], [2], on quadtrees, octrees (e.g. [15], [16]), or related structures (e.g. [17]). An example of such a discretization is shown in Fig. 2.1d. In this work, the LSF-values are stored on Cartesian grids - rectilinear grids using equal side-lengths. The side-length is referred to as the grid-spacing or grid-delta and will be denoted with Δg. All stored values $\phi(x)$ will be normalized to Δg [2]:

$$\phi(\mathbf{x}) = \frac{1}{\Delta g} f_{\text{SDF}}(\mathbf{x}) \tag{2.6}$$

To obtain an explicit representation out of this discretization, the interfaces represented using the LS are linearly approximated between points along the grid edges called cut points. Cut points are those points, where an interface intersects with an edge. Along such a grid edge, a cut point's location is determined using the LSF-values on the two incident grid points through linear interpolation. For an interface to be located on a grid edge, the two incident grid points have to be of opposite sign in their LSF-value. In the simplest case, the two values simply agree on a single location. This is the case when $|\phi(\mathbf{p}_a)| + |\phi(\mathbf{p}_b)| = 1$. An example for this case is depicted in Fig. 2.2 between grid points $\mathbf{p_5}$ and $\mathbf{p_2}$.

Since a grid point always stores a single value, the single shortest distance to the surface, it cannot not represent two different distances on two different edges simultaneously, as shown in Fig. 2.2 for the point $\mathbf{p_2}$ which represents the vertical and not horizontal distance. Therefore it is possible for two LSF-values of opposing sign, to not agree on a single location. This is shown in Fig. 2.2 between grid points $\mathbf{p_4}$ and $\mathbf{p_5}$. Then, the location of the interface is approximated by linear interpolation. From the two LSF-values a linear polynomial is built

$$\phi(t) = \phi(\mathbf{p}_a) + (\phi(\mathbf{p}_b) - \phi(\mathbf{p}_a)) \cdot t. \tag{2.7}$$

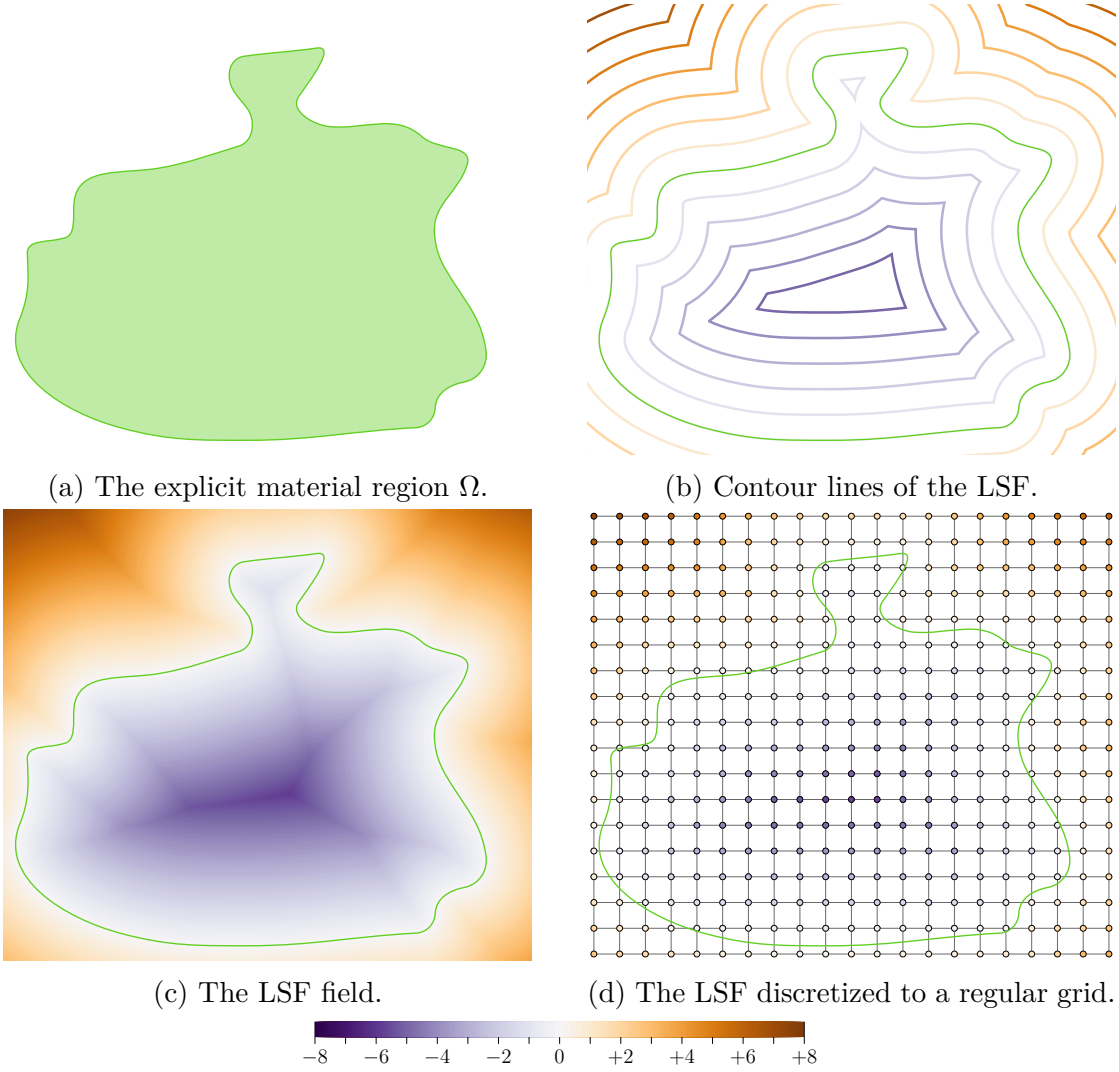
with $t \in [0, 1]$ being the position on the edge from $\phi(\mathbf{p}_a)$ to $\phi(\mathbf{p}_b)$, as seen from $\phi(\mathbf{p}_a)$. By definition, $\phi(t) = 0$, so the interpolated interface location is found at

$$t = \frac{-\phi(\mathbf{p}_a)}{\phi(\mathbf{p}_b) - \phi(\mathbf{p}_a)} = \frac{\phi(\mathbf{p}_a)}{\phi(\mathbf{p}_a) - \phi(\mathbf{p}_b)} \tag{2.8}$$

and in absolute coordinates the cut point is therefore at

$$\mathbf{p}_{\text{cut}} = \mathbf{p}_a + (\mathbf{p}_b - \mathbf{p}_a) \cdot t \tag{2.9}$$

Using the above LS definition, an interface can only ever occur on an edge, when the two incident grid points' LSF-values have opposing signs, or when a grid point has an LSF value of zero.



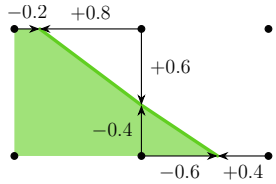
(a) The explicit material region Ω.

(b) Contour lines of the LSF.

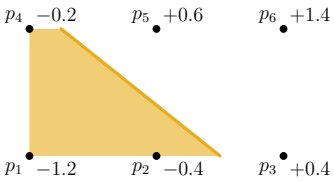
(c) The LSF field.

(d) The LSF discretized to a regular grid.

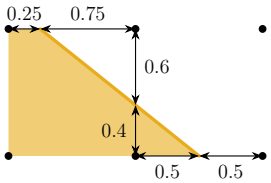
Figure 2.1: 2D example material region and different representations thereof, in regards to the LS method.



(a) LSF values determined using the signed distance (black arrows) to the explicit interface (green line).



(b) The stored LSF-values and the resulting interface interpolation (orange).



(c) The calculated distances (black arrows) between the grid points and the (explicit) interface interpolation.

Figure 2.2: 2D example of how an interface through a small set of LS grid points is interpolated and the relationship of stored LSF-values and signed distances.

2.1.2.1 Sparse Storage

Storing the full LSF grid, whereby all grid points in the volume Ω are stored, requires a considerable amount of memory. As points further away from the interface do not contribute significantly to the interface's representation, they are often not stored, referred to as the *narrow band method*. As its name suggests, the narrow band method only stores a narrow band of grid points, symmetrically around the implicitly represented interface. The thicker the (narrow) band, the more information is stored about the location of the surface, but also the more storage and computational time is required.

Since the distance of a point to the interface can be used as a measure for how important the point is for the surface's representation, the points can be classified, into layers around the interface (e.g. [1], [2], [18]). Since the used LSF is based on a SDF, a grid point's LSF-value can be used to determine whether the grid point is within the selected layer or not. In [1], [2] the layer classification is given by:

$$\mathcal{L}_i := \begin{cases} \{\mathbf{x} | i - \frac{1}{2} \leq \phi(\mathbf{x}) < i + \frac{1}{2}\}, & \text{if } i < 0 \\ \{\mathbf{x} | -\frac{1}{2} \leq \phi(\mathbf{x}) \leq +\frac{1}{2}\}, & \text{if } i = 0 \\ \{\mathbf{x} | i - \frac{1}{2} < \phi(\mathbf{x}) \leq i + \frac{1}{2}\}, & \text{if } i > 0 \end{cases} \tag{2.10}$$

The narrow band can now also be described in terms of the layers it includes. A narrow band of layer thickness n would include all layers $-n, \ldots, 0, \ldots, n$.

The narrowest band, while still representing the LS surface, is given by $|\phi(\mathbf{x})| \leq \frac{1}{2}$ [2] and is equivalent to the layer $\mathcal{L}_0$. The points in the layer $\mathcal{L}_0$ may then be called active (grid) points [2].

For the purpose of the implementation presented in this work, the sparseness will however not be permitted to be below $|\phi(\mathbf{x})| \leq 1$. By this sparseness definition, it is ensured that, for every edge which is cut by an interface, always both incident grid points store an LSF-value. This is required by the way the filling of the intermediate octree data structure is implemented. An example of such a set of points is given in Fig. 2.3.

The reduction of stored LS grid points leads to a reduction in stored information, as stated above. This reduction does not necessarily have to be lossy. Depending on the used LS definition, and in particular the norm used for the SDF underlying the LSF, the information can be recreated from the sparse data set. This is especially the case with the Manhattan norm used in this work. It allows for the easy (re-)creation of neighboring grid points, by simply adding (or subtracting) 1 to all neighboring points and keeping the value closest to zero for the new grid point. Whether to add or subtract 1 for the new point depends on the sign of the neighboring LSF-value. When the source LSF-value is positive, then 1 is added, when the LSF-value is negative, 1 is subtracted. This means that

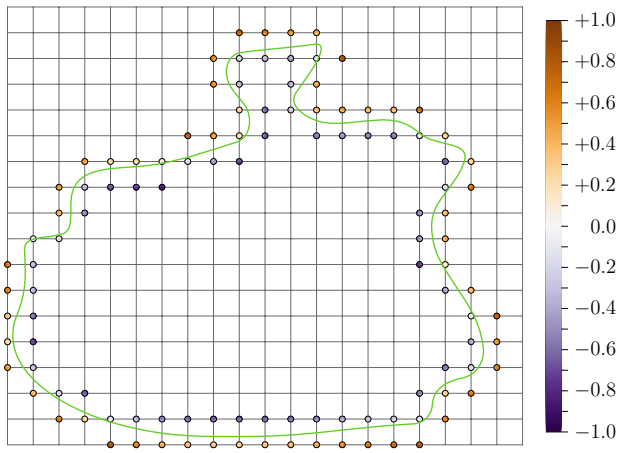


Figure 2.3: The 2D example material region, represented in by a LS stored as a sparse set of grid points. Only grid points with an LSF-value of $|\phi(\mathbf{x})| \leq 1$ are stored. This is as sparse as the LS is allowed to get to be used as an input to this work's implementation.

inside points try to create neighbors which are more inside than themselves, while outside points contribute LSF values which are even more outside.

The action of adding to or subtracting from a single LSF source value in a given direction and distance, will be termed *advancing* the said LSF-value, and is explained in detail in Section 2.1.2.2. A neighboring grid point will hence be calculated by advancing all neighboring grid points along a direct grid edge, and taking the value closest to zero as the advanced LSF value. This calculation of a neighboring grid point is commonly called *re-distancing* [2]. The term *advancing* was introduced in order to distinguish between simply using a single LSF source value, and the complete calculation of re-distancing, incorporating all neighboring grid points.

2.1.2.2 Advancing Level Set Values

In this work, *advancing* an LSF-value from point $\mathbf{p}_a$ with a known LSF-value to a point $\mathbf{p}_b$ in the positive or negative direction, means to add to the (source) LSF-value of $\mathbf{p}_a$, the distance between the two points, normalized to the grid spacing Δg, with the sign corresponding to the chosen direction.

$$\phi(\mathbf{p}_b) \approx \phi(\mathbf{p}_a) \pm \frac{\|\mathbf{p}_a - \mathbf{p}_b\|_1}{\Delta g}. \tag{2.11}$$

Note that this is possible because the Manhattan norm is being used for the LS representation.

2.1.2.3 Thin Features

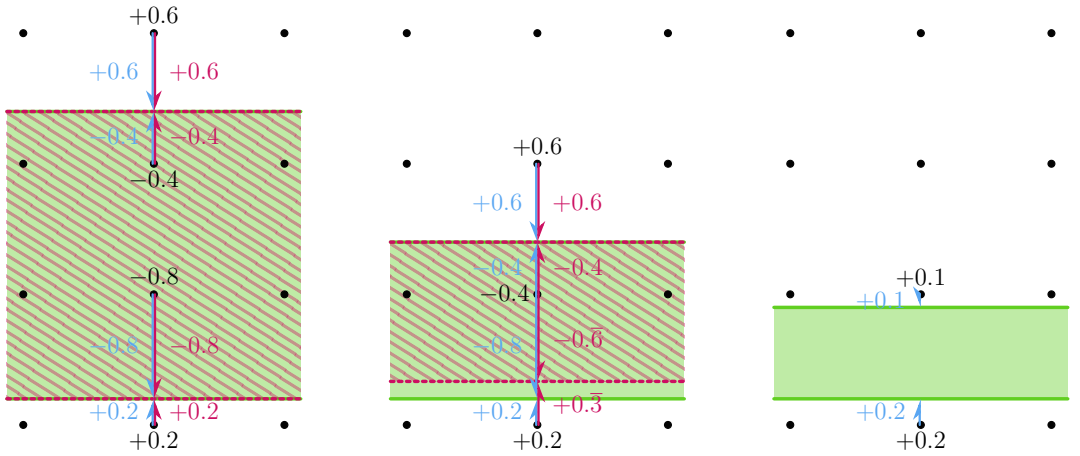
The LS definition described above, has the limitation that it cannot describe material regions smaller than $2\Delta g$ correctly. The problem comes from the previously mentioned fact, that any grid point only stores a single LSF, representing the smallest SDF to any surface. In a 3D grid, any point is, however, incident to 6 grid-edges, and therefore potentially 6 different distances to the surface, along those grid-edges. Therefore a grid point's stored LSF, which is the closest to zero, trumps over five other potential LSF-values which are larger in absolute value.

For the case of a thin layer with parallel surfaces, this is represented in Fig. 2.4. The feature in Fig. 2.4a is thicker than $2\Delta g$, so it is represented exactly, and not considered a thin feature in this context. As soon as a feature thickness falls below the $2\Delta g$ threshold, it is considered a thin feature, and can be affected by the LS representation. For such a thin feature it can be possible, that grid points within the feature should represent two different distances to the two individual surfaces. However, each grid point only stores one value. Therefore, as shown in Fig. 2.4b, the thin feature is made even thinner, by the previously discussed interface interpolation. In case the outside LSF-values below the thin feature in Fig. 2.4b were not present, like in a $\mathcal{L}_0$ layer representation, the feature would be made even thinner, symmetrically around the inner grid points. This also highlights an exception, in which the thin feature would still be correctly represented, namely in the case where the feature is already symmetric around the inner grid points. Note that there are also grid point constellations for thin features with a thickness smaller than $2\Delta g$ but larger or equal to one Δg, that result in an exact representation of the surface. However all features thinner than $2\Delta g$ shall be consider thin features within the scope of this thesis.

The feature becoming thinner is another implication of the interpolation discussed and shown in Fig. 2.2. When the thickness of Ω falls below Δg, however, the interpolated feature based on the LSF values disappears completely. This problem is depicted in Fig. 2.4c. The grid points both above and below the feature in Fig. 2.4c need to store positive LSF-values, as they lie outside of the feature. Since there are no opposite signed points, the interface cannot be represented properly.

It is important to note that what is considered a *thin* feature, is not based on absolute size, but purely based on the discretization made through the LS grid and its grid spacing Δg. The cases of Fig. 2.4b and Fig. 2.4c, could both be resolved by making the grid spacing Δg smaller, introducing more points, which could then represent each of the surfaces through neighboring grid points of opposing sign accurately. The downside of these additional points is an increased memory usage especially with high aspect ratio structures.

The case of the thin angled feature is shown in Fig. 2.5 in Section 2.1.3.1 together with a multi-material technique for resolving this issue.



(a) Normal feature: a feature which is thicker than $2\Delta g$.

(b) With a feature thinner than $2\Delta g$, but greater or equal to Δg, the feature can become even thinner by the interpolation.

(c) A thin feature, with a thickness below one grid spacing Δg, cannot be represented in the defined interpolation scheme anymore. The feature is entirely removed.

Figure 2.4: 2D examples showing how different thicknesses of thin features lead to a reduction in the feature representation, or to the complete disappearance of the feature. The green area indicates Ω. The distances to the grid points are marked in blue. The black numbers show what will be stored in the LS. The magenta lines show the interpolated region based on the stored LS values. *Images are adapted from* [2].

2.1.3 Multi-Material Representation

In microelectronic simulations, it is desirable to work with devices consisting of multiple materials M, placed into the void. The M materials and the void phase, give $M + 1$ distinct phases which need to be represented. The void phase is surrounding the object on the outside, and on any inside voids/holes the object might have.

In the LS method, representing the $M + 1$ distinct phases, has to be done by using M LSs, since a single LS only carries inside-outside information for a single material or two phases. In this work, the LS wrapping approach [1] is used and will be discussed in detail in the next section.

2.1.3.1 Wrapping Approach

The enclosing technique used in this work to model multiple materials [1] is referred to as the *wrapping approach* [2], or *additive LS approach* [4]. Using this technique a multi-material object with M materials and a void phase is stored using one LS dataset per material. Each material is assigned a material index $m \in \{0, \dots, M-1\}$. The domain represented by each material's LS wraps around, and therefore includes, the domains of all materials with smaller material index. Therefore the domain $\widetilde{\Omega}_m$ implicitly described by the SDF $\phi_m(\mathbf{x})$ associated with a material m, is not a representation of the modeled material domain Ω_m, but rather the union

$$\widetilde{\Omega}_m := \bigcup_{i=0}^{m} \Omega_i = \widetilde{\Omega}_{m-1} \cup \Omega_m \tag{2.12}$$

of the domains of all materials with lower i than m. The only material where $\widetilde{\Omega}_m$ and Ω_m coincide, is the material with $m = 0$.

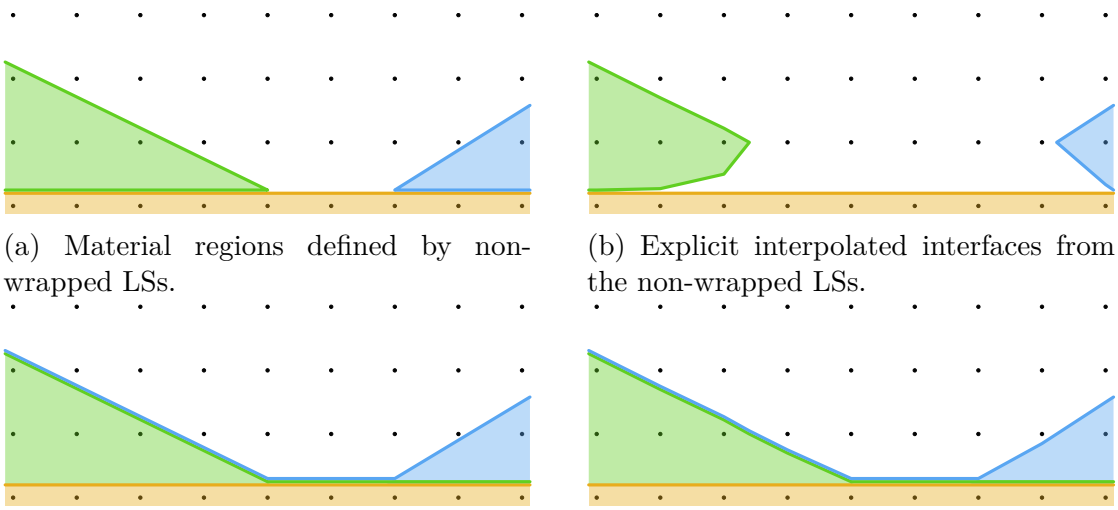
The material represented by a point $\mathbf{p}$, is determined by the lowest material identifier (ID) which still has an inside value, a negative LSF-value. In the case that even the material with the largest material ID has an outside value, meaning a positive LSF-value, the point $\mathbf{p}$ belongs to the special outside void phase, surrounding all actually represented materials. By this, the assigned material ID is always unique. With no tied cases, there is also no need for a deciding strategy, which is otherwise require when using the indicator function from [10], where a randomized push is employed to resolve ambiguous situations.

Using this LS wrapping approach, it becomes possible to resolve thin features as discussed in Section 2.1.2.3. In order to preserve a thin feature, it is wrapped onto a another adjacent material, whose underlying stored LS is not considered a thin feature. Therefore, multiple thin features can be stacked on top of each other, as long as the thin feature which is the lowest in the stack is wrapped around a material which is not a thin feature itself. This also highlights an important point in the LS wrapping approach - the order in which the materials are wrapped around each other. A poor ordering in the wrapping can also lead to the undesired removal of thin features, which would have otherwise been preserved through a different ordering.

The preservation of the thin features through wrapping works solely based on the fact that through the wrapping, the region represented by the material's LSF is not the thin material region itself, but rather it is the region of all material regions with lower ID. The individual non-overlapping material regions which were the basis for creating the wrapped multi-material LS, are recovered from the wrapped LS, simply by subtracting all lower material ID regions once again.

As discussed in Section 2.1.2.3, thin features with a thickness below a single grid spacing Δg cannot be represented without the LS wrapping approach. Hence the LS wrapping approach allows for *subgrid-accuracy*.

The LS wrapping approach's preservation of thin features works in the described way, for both the case of parallel thin layers and the small angled features. An example of a small angled thin feature is given in Fig. 2.5. The angles would become a rounded corner through the LS's interpolation - Fig. 2.5b. Wrapping it around an underlying bulk material domain, as in Fig. 2.5c, retains the sharp angles, as shown in Fig. 2.5d.



(a) Material regions defined by non-wrapped LSs.

(b) Explicit interpolated interfaces from the non-wrapped LSs.

(c) Material regions defined as wrapped LSs.

(d) Explicit interpolated interfaces from the wrapped LS input.

Figure 2.5: Example of how thin features are preserved using the LS wrapping approach. The material region which has to be represented is shown on the left. The right shows the interpolation based on the discretization to the LS grid. The top row depicts the case without the implementation of material wrapping, while the bottom row depicts the same geometry while using the LS wrapping approach. *Images are adapted from* [2].

2.1.4 Summary of Level Set Conventions

As a quick reference for readers who are familiar with the level set (LS) method, an overview of the applied conventions/definitions, regarding the LS method is provided here, without further explanation. For further explanation the reader is directed to the previous LS method sections, starting with Section 2.1.1.

- A normal vector $\mathbf{n}$ always points outwards from a material domain Ω.

- The level set function (LSF) ϕ is negative $\phi < 0$ inside of a material, positive $\phi > 0$ outside, and zero $\phi = 0$ as the interface itself.
- The LSF is stored on a Cartesian grid.
- The narrow band method is used, to make the LS grid storage sparse.
- The used LSF is a signed distance function (SDF) based on the Manhattan norm $\|\cdot\|_1$ and normalized to the grid spacing Δg.

2.2 Octree

An *octree* subdivides a given space into eight cubes, or usually cuboid sectors, and each of these cubes is again subdivided into eight parts, and so on. So an octree can be seen as a recursive subdivision of a given space. In two dimensions the space can analogously be divided into four sections (rectangles) instead of eight, hence in two-dimensional (2D) the structure is called *quadtree*. For the important case of the three-dimensional (3D) Cartesian space, an octree gives an intuitive way of subdividing the space.

Besides its subdividing nature, an octree is, as the name suggests, a tree data structure where each of the tree's nodes can have eight child nodes, or at most eight child nodes, depending on the implementation and sparsity rules of the octree.

2.2.1 Octree Terminology

Since there are different conventions and interpretations of terms commonly used with tree data structures, the following list contains key definitions, as they will be used in this work.

Root Is the only node which has no parent node. Each node of a tree may be considered the root node of a subtree formed by its children.

Parent Direct ancestor to a node.

Ancestor Any node going up the branch from the current node to the root node, e.g. the node's parent, the node's parent's parent, etc.

Child One of the (at most) eight direct descendants of a node.

Descendant Any node which has the current node as its ancestor, e.g. the node's children, the node's grandchildren, etc.

Sibling Nodes that share the same parent node. Any node in an octree can have at most seven siblings.

Leaf A node which has no children.

Neighbor A neighboring node is a node which shares a geometric interface with another node. Please note that this means that neighboring nodes can be

quite far apart, when seen from the tree datastructure, but geometrically they are close and share at least one of the entities of a cuboid - a face, an edge, or a corner.

Face-Neighbor A node which shares a face with the current node.

Edge-(Only-)Neighbor A node which shares **only** an edge with the current node. Note that by this definition face-neighbors which also shares an edge with the current node are not considered edge-neighbors, but face-neighbors, even though they geometrically also share an edge.

Corner-(Only-)Neighbor A node which shares **only** a corner with the current node. Note that by this definition neither face-neighbors, nor edge-(only-)-neighbors which also share a corner with the current node are considered corner-neighbors, even though they share a corner.

2.2.2 Storage Implementations

The following will give a quick introduction into the ways in which an octree data structure may be implemented and the most appropriate method, chosen for this work. There is the classical pointer-based octree, in which every node stores a pointer for each of its children. Then, there are approaches in which some form of container is used to store a node's children. For example in a child-array-based octree, each octree node only stores a single pointer to an array in which all of its children are stored. In a child-sibling-based approach, each node stores two pointers - one to its first child, the other to its sibling. So the child-sibling-based approach is similar to the child-array-based approach, but with the array exchanged for a linked list. In the child-sibling-based approach, each node also only stores a single pointer for all eight children.

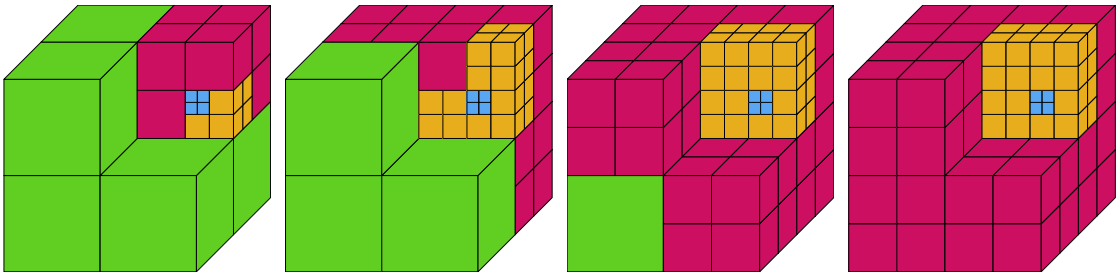
Following these implementation methods, all of which are based on pointers to various degrees, finally a very different approach is noteworthy - the one using a single contiguous array to store the entire octree. The access to individual nodes and relations between parents and siblings are calculated based on the ordering given to the nodes for the contiguous storage. For this ordering usually the so called *Morton encoding* or *Z-order* (or many other name-variation thereof) is used [19]–[22]. Other orderings exist, such as the Peano-Hilbert order [19]. The Morton ordering is given by the order in which the nodes are visited during a depth-first traversal of the full tree of a given depth. The encoding of a node's location within the tree, especially with such contiguous storage implementations, is also referred to as *locational code* [20].

The selection of the type of octree implementation approach depends on the use-case, with the main factors being memory consumption and sparsity, as well as memory access patterns. For example, a classical pointer-based octree allows for an arbitrarily sparse octree, as only those nodes which are stored must be allocated.

When a node's children are stored in a fixed size array, as in the example of the child-array-based approach, then the need for a single child of a node, leads to storage allocation for all eight child nodes. On the other hand, in order to access a sibling in the pointer-based octree, there are two pointer indirections - first going to the node's parent, then from the parent to the desired child. In the child-array-based approach, getting to any sibling is a simple matter of pointer arithmetic, as all siblings are contiguously stored in a single array.

2.2.3 Balancing

An octree is said to be balanced if neighboring leaf nodes do not exceed a certain size difference or ratio. A common ratio is the $2:1$ balancing. As there are different categories of neighboring relations, there are different categories of balancing as well. In a face-balanced octree, all face-neighbors do not exceed the given size difference. When all leaf nodes sharing an face or an edge do not exceed the given ratio, it is called edge-balanced, and when also all leaf nodes sharing a corner fulfill the criterion, then the octree is called corner-balanced (see Fig. 2.6).



(a) No balancing. (b) Face-balanced. (c) Edge-balanced. (d) Corner-balanced.

Figure 2.6: Example showing different kinds of $2:1$ balancing with octrees.

2.3 Mesh Data Structure

Mesh data structures store the individual elements of a mesh and their relations, such as the locations of the elements and their connectivities, as well as potentially additional data, such as assigned materials, or weightings needed for algorithms operating on the mesh. In this work, only the generation and manipulation of tetrahedral meshes will be discussed. Other types of meshes, including general meshes being able to store multiple element types, might have other implications with regards to their data structures.

Tetrahedral meshes consist of the four simplex types with dimensionality 0 to 3 - vertices, edges, triangular faces, and tetrahedra, respectively. The information

regarding their locations and connectivities has to be stored in the mesh data structure. Since any simplex within the mesh consists of simplices of lower order, only the positions of the vertices have to be stored explicitly. The locations of edges, faces and tetrahedra are then implicitly given through the relations of the lower order simplices from which they are composed. Additional data, however, still might need to be stored for the other types of simplices - as is the case with this work. Besides hierarchical incidence, information about adjacency may be required, such as which tetrahedron is adjacent to a given tetrahedron, by sharing a given face.

The following section will describe the data structure developed within the scope of this work to hold tetrahedral meshes and related data, such as the material of each tetrahedron or adjacency information.

2.3.1 Tetrahedral Mesh Data Structure

The data structure which is chosen for the tetrahedral mesh in this work was developed with a focus on simplicity. There are only two containers - one storing all vertices, the other storing the tetrahedra. The vertex container is a contiguous array of coordinates, whereby each element describes the location of a single vertex. The index of a vertex in this array is then used as the unique identifier (ID) of the vertex. The tetrahedra are represented in similar fashion, having a unique ID based on their index in the tetrahedra container, where each element consists of the four vertices forming the tetrahedron. This two-container concept would be enough to simply store a tetrahedral mesh, but it is not sufficient for this work's purposes. Unfortunately, querying neighboring tetrahedra is inefficient with this data structure and storing additional data for the simplices is challenging.

In order to gain more efficient incidence querying, another container is introduced. This container holds the inverse relation of the already stored one, so which tetrahedra are incident to a given vertex ID. Other than the entries in the tetrahedra container, which always hold four vertex IDs per entry, the number of tetrahedra incident to a single vertex can be arbitrarily large. Therefore, both dimensions of this third container need to have a dynamic size.

Vertices and tetrahedra received their unique ID through the index in the corresponding container. An edge, on the other hand, will be addressed by its edge id,

$$e_l := (v_i, v_j), \text{ with } i < j, \tag{2.13}$$

which will be a tuple built from the two incident vertices' vertex IDs v_i and v_j. The vertex IDs in an edge ID will always be ordered from lowest to highest.

For each face, a face ID will be constructed analogously from the incident vertex IDs v_i, v_j, v_k

$$f_l := (v_i, v_j, v_k), \text{ with } i < j < k. \tag{2.14}$$

In this way, all simplices in the mesh are addressable by a unique ID. The edge IDs and face IDs also store their incidence relation to the vertices in them.

To store the needed additional data, the vertex container will not only store a vertex's location, but also its additional data. The tetrahedra's additional data will be stored in another container, the tetrahedra-data container, which is indexed in the same way, since the tetrahedra container is also using the tetrahedra's unique IDs. For edges and faces each, a hash map storing the additional data to the edge ID or face id, is used. The hash map key is the edge ID or face ID respectively.

The combination of vertices and tetrahedra, with their incidence relations, define all of the simplices. By knowing which vertices belong to which tetrahedron, one can also infer which edges form a tetrahedron and which tetrahedra share an edge. Note that the mesh's simplices are only defined by the vertices-tetrahedra-relations, as this work assumes a conforming tetrahedral mesh. A conforming tetrahedral mesh does not allow for dangling edges or faces, hence it is not necessary to store edges or faces which are not related to any tetrahedron. Therefore the set of vertices, tetrahedra, and their relations will always be completely stored and kept up-to-date with any changes to the mesh's geometry. Since the implemented algorithm only requires edge and face information close to material interfaces, edges and faces are only generated and stored in these regions. Therefore, the memory footprint of the full mesh can be reduced drastically.

Using these additional data containers, the speed of the query of mesh elements can be improved drastically. If all edge IDs which are part of a given tetrahedron are required, the vertices of the tetrahedron are looked up from the tetrahedron container and the edge map is searched for each vertex combination forming the edges of the tetrahedron. Thus, each combination will result in a single edge ID.

2.4 Mesh Quality

Both the finite element method (FEM) and finite volume method (FVM) are used to numerically solve partial differential equations (PDEs) based on splitting the simulation domain into a finite number of elements composing a mesh. They have different constraints on the meshes and their elements. Elements of good quality lead to faster solving times and numerically more precise results. Elements of bad quality can make a problem unsolvable on the given mesh. The notion of what makes a mesh element be of good quality or bad quality can be quite complex. An introduction to the constraints and some of the different measures of quality

for FEM and FVM are given in Section 2.4.1 and Section 2.4.2, respectively. For a further source on element quality for FEM refer to [11].

Measures for the element quality can also be used to compare the outputs of different meshing approaches and implementations. Let it be noted however, that measures for element quality are not the only consideration with regards to the overall quality of a produced mesh. An example of an important mesh characteristic, not considered through such element quality measures, is the adherence to the given input surfaces/interfaces. Depending on the input geometry, precisely representing the input regions and having good quality elements might not be possible, and some form of trade-off between the two must be made.

2.4.1 Finite Element Method

For FEM, the optimal triangulation in both 2D and 3D are equilateral triangles and tetrahedra, respectively. While equilateral triangles are space tiling, this is not the case for equilateral tetrahedra. For tetrahedra, the dihedral angles are the quantity deciding whether they are space tiling or not. Equilateral tetrahedra however all have equal dihedral angles of $\arccos\frac{1}{3} \approx 1.231\,\text{rad} \approx 70.53°$. This dihedral angle is not equal to the dihedral angle needed for five tetrahedra around an edge to tile the space completely, which is $\frac{2\pi}{5} = 72°$ [23].

With exact equilateral triangulation being impossible, there are two general cases of angle constraints which are often considered when tiling space with triangles - the non-obtuse triangulation problem [24], [25] and the acute triangulation problem [23]. Both of these impose constraints on the permitted inner angles of the triangles in the triangulation. An angle θ is called acute if $0 < \theta < \frac{\pi}{2}$, and an angle is obtuse in the case $\frac{\pi}{2} < \theta < \pi$. In this sense, the triangulation $T_{\text{non-obtuse}}$ with non-obtuse angles includes right-angles and the constraint is $\forall\theta \in T_{\text{non-obtuse}} : 0 < \theta \leq \frac{\pi}{2}$. As an acute triangulation, T_{acute} only includes angles strictly less than $\frac{\pi}{2}$, the constraint $\forall\theta \in T_{\text{acute}} : 0 < \theta < \frac{\pi}{2}$ is stronger, than that for the case of the non-obtuse triangulation. Acute triangulation is therefore inherently more involved than non-obtuse triangulation, as show in [23].

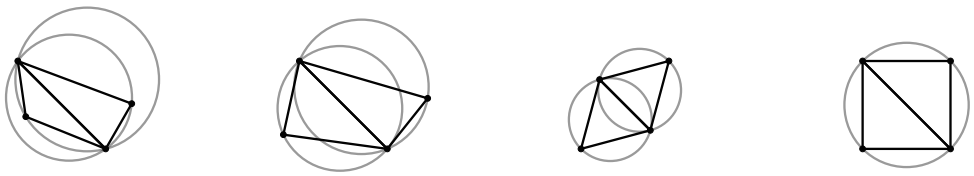
The quality of a mesh for FEM purposes is therefore often measured by the dihedral angles of the mesh' elements. The closer the dihedral angles are to the optimum of $\arccos\frac{1}{3} \approx 1.231\,\text{rad} \approx 70.53°$, the better the quality, even though this optimum is impossible to reach, as stated above.

2.4.2 Finite Volume Method

The FVM typically requires the Delaunay criterion to be fulfilled [26]. Other variants of the FVM exist, such as the cell-centered FVM, which does not require the

Delaunay property to be fulfilled [5]. Those other variants and their requirements on mesh quality are outside of the scope of this work.

The Delaunay criterion simply states that for any mesh consisting of d-dimensional simplices, no vertex of one simplex may lie withing the circumsphere of another simplex. In 2D, this means that the vertex of one triangle must not lie within the circumcircle of any other triangle in the mesh as shown in Fig. 2.7. Such a Delaunay triangulation is the dual of a Voronoi-diagram, with the circumspheres' centers being the points of the Voronoi-diagram. The Voronoi-diagram's cells, the so called *Voronoi cells* are the basis of the traditional FVM, compared to the previously mentioned cell-centered FVM.



(a) Violated Delaunay criterion.

(b) Fulfilled Delaunay criterion.

(c) Equilateral triangles.

(d) Special case with points on the circumcircle.

Figure 2.7: Different 2D example triangulations and information on whether they fulfill the Delaunay criterion.

The quality measure of a mesh for FVM purposes, is therefore whether or not the mesh fulfils the Delaunay criterion.

2.4.3 Selected Quality Measures

In 2D, the Delaunay property is connected to the angles of the triangles by maximizing their minimum angle [27]. Therefore, in 2D it is possible to obtain a quality measure directly representing both the quality for FEM and FVM.

In 3D, this is not the case as stated in [27], the conditions of non-obtuse dihedral angles and the Delaunay criterion are incomparable. This is due to the existence of tetrahedralizations which only consist of non-obtuse dihedral angles, but still are not conforming to the Delaunay criterion and vice versa. What is comparable also in 3D, as it holds for Delaunay triangulation of all dimensions, are the results derived by Rajan in [28].

For this work, the main considered element quality measure shall be focused on the tetrahedra's dihedral angles. This choice is made as it is used as the main measure for mesh quality in related works, especially previous works on the mesh cleaving algorithm (compare Chapter 3). For guarantees of meshing algorithms,

often their minimum and maximum values are given. This is not the case only due to the fact that guarantees are typically given as upper or lower bounds. In the case of mesh qualities, even a single bad element can impact the meshing method.

Chapter 3

Related Works

Due to the almost universal applicability of partial differential equation (PDE) solvers, much of recent research has been focused on different meshing algorithms and many variations exist to solve a vast range of problems.

In Section 3.1, different approaches to tetrahedral meshing, not employed in this work, are mentioned. The concepts used in this work will be discussed in great detail in the following chapters. Firstly, in Section 3.2, meshing of single-material isosurfaces will be discussed as the foundation for the multi-material case. Then, in Section 3.3, the single-material meshing will be extended to multiple materials.

3.1 General Tetrahedral Meshing Algorithms

Meshing algorithms are often categorized roughly by an idea or underlying structure they employ, but those categorizations are neither unique, nor does a universal convention exist. The best example for this are meshing algorithms which employ filling a lattice into an octree structure. Some works categorize those as octree-based methods [14], [29], while others refer to these algorithms as lattice-based methods [10], or even tessellation-based methods [30]. Therefore, this work tries to categorize algorithms based on the strongest conventions, but differences to other works may be found.

In advancing front methods (e.g. [31]), as the name suggest, a front or surfaces is moved in order to create a volumetric mesh from a surface mesh. When using implicit representations like a level set (LS) for the meshed regions, advancing front methods require the generation of a surface mesh from the implicit input data first. This can be done using the marching cubes algorithm [32] and several other methods.

Creating a Delaunay conforming mesh can be achieved through variants of Delaunay based methods [33], [34]. These methods attempt to triangulate a set

of given points, or previously constructing such a set of points, adhering to the mentioned Delaunay criterion (see Section 2.4.2). Delaunay methods can even be combined with the previously mentioned advancing front method [35]. Delaunay-based methods have, however, been repeatedly shown in literature to suffer from the creation of elements with bad angles, such as slivers. Hence, several studies have looked resolving these bad elements, with a prominent one being the sliver exudation [36].

The main claimed advantage of the methods this work is based on, compared to above mentioned approaches, is that they offer to produce meshes with guarantees on the bounds of dihedral angles [14]. Bad elements in meshes can also be tackled by methods trying to improve an existing mesh, after its initial generation. Such approaches like mesh refinement methods, mesh smoothing methods, or even physics based methods (like mass-spring-systems [37]) employing PDEs themselves, are however considered out of the scope of this work.

3.2 Isosurface Stuffing

Isosurface stuffing describes the general idea of a meshing algorithm using a lattice to fill an implicit surface, generating a tetrahedralized explicit volume description, by conforming the lattice to the implicit surface.

3.2.1 Original Isosurface Stuffing

Labelle and Shewchuk presented in "Isosurface Stuffing: Fast Tetrahedral Meshes with Good Dihedral Angles" [14] their isosurface stuffing algorithm. This algorithm creates a tetrahedral mesh from a given LS input. The created mesh's tetrahedra have bounded dihedral angles, with the bounds varying based on the used parameters and sub-variants of the algorithm. Nonetheless the angle bounds given are stated to be guaranteed through a computer-assisted proof. It is claimed in [29] that [14] was the first of its kind to offer angle bounds.

The isosurface stuffing algorithm works in the following way: The space is filled with a body centered cubic (BCC) background lattice. This background lattice is implicitly filled with tetrahedra, resulting in a tetrahedral background grid. Next, the points where the background grid's edges intersect with the isosurface are determined. Those points are called cut-points. Simply cutting through the background grid tetrahedra wherever there are cut-points and tessellating the result, would lead to bad output tetrahedra in certain cases. Therefore the algorithm proceeds to check every found cut-point to assess whether the cut-point is too close to any of the background grid vertices. When a cut-point is too close to any vertex, the cut-point is marked, as it violates the vertex. Any found violation

is then resolved by moving the violated background grid vertex to the position of the violating cut-point, and removing the violating cut-point. These operations are called snap and warp. The cut-point is snapped to the vertex and the vertex is warped to the interface position marked by the snapped cut-point. When a vertex is warped due to a snap, any other cut-point lying on another edge, incident to the warped vertex, is also removed. Any vertex is warped at most once, in order to guarantee the bounds given for the output tetrahedra. Once all violating cuts have been resolved, for each background grid tetrahedron, a set of output tetrahedra is created using a set of stencils. The stencil which is selected for a given background grid tetrahedron depends on the signs of the level set function (LSF) values at the tetrahedron's vertices, the number and location of the remaining, non-violating cut-points, and some further rules resolving ambiguities. In this way, a consistent tessellation with high-quality tetrahedra is ensured.

Labelle and Shewchuk also presented a graded variant of the isosurface stuffing algorithm in [14]. This graded variant uses a balanced octree instead of the regular background grid. As the BCC lattice is not directly tileable across different sized cells of the octree, they introduce further types of background tetrahedra to bridge between those cells. Otherwise, the idea behind the algorithm is the same as for the regular background lattice.

3.2.2 Isosurface Stuffing Improved

The previously mentioned isosurface stuffing algorithm (Section 3.2.1) was then studied by Doran in [38]. Therein, the use of a different background lattice and feature matching to create sharp corners and edges are proposed.

The original isosurface stuffing algorithm uses a BCC background lattice. The BCC lattice consists of tetrahedra which are all similar to a single tetrahedron which has dihedral angles of 60° and 90°, which is advantageous for numerical applications (see Section 2.4.1). Therefore, Doran replaced the BCC lattice by an acute A15 lattice, whose tetrahedra have acute dihedral angles, in the range of 53° to 79° [38]. This is closer to the dihedral angle of $\arccos\left(\frac{1}{3}\right) \approx 70.529°$ of the regular tetrahedron, compared to the case of the BCC lattice. By using the acute A15 lattice tetrahedra, Doran reports better (lower) maximum dihedral angles in the output mesh. Additionally, in [38], the distribution of dihedral angles in the output mesh is reported to be more concentrated around the mentioned optimal dihedral angle. For the graded case, based on an octree substructure, Doran's algorithm could not improve upon the original BCC lattice results, even though a few different approaches were investigated.

The feature matching presented by Doran is used to create or recreate features of the input geometry which are lost due to the discussed implications of an implicit

representation, such as a LS. A common example of features which may be lost are sharp corners and edges, which become rounded.

An open source implementation of the non-graded version of the improved isosurface stuffing algorithm of [38], using the A15 lattice has been published under the name of *Quartet* [I] by Bridson and Dorian.

3.2.3 Other Similar Approaches

The same concept of the isosurface stuffing algorithm [14] is also used by Wang and Yu in [29], where they use the octree substructure to additionally refine a background BCC grid in regions of high curvature. Additionally the rules selecting the used stencil, after the cutting and warping, have been adjusted to always select the one resulting in the tessellation with better dihedral angles for the given tetrahedron. However, no details are provided about how mesh conformity is handled using such a selection method. Furthermore, they claim that their meshing algorithm achieves a provable minimum dihedral angle of 5.71°.

3.3 Mesh Cleaving

The concept behind the group of the isosurface stuffing algorithms of Section 3.2 was brought to the multi-material case with the *lattice cleaving* algorithm presented in [10] and further development of the *mesh cleaving* algorithm [39].

3.3.1 Lattice Cleaving

As the original isosurface stuffing algorithm (Section 3.2.1) only considers two-phase-interfaces (e.g. inside-outside), Bronson, Levine, and Whitaker presented their *lattice cleaving* algorithm which extends to the multi-material case in [40] and later in [10]. This lattice cleaving algorithm builds upon the idea of the isosurface stuffing algorithm in the way that they introduce other types of interface points, additionally to the cut-points on the edges. This way, interfaces between materials can be handled as well. The two additional interface points were named *triple* points and *quad* points, as a triple-point is located on a face where three materials meet, and a quad-point is located inside a tetrahedron, at the point where four materials meet. In order to cope with the larger number of stencils which would possibly be needed in order to resolve the multi-material case, they also introduced the concept of a single generalized stencil. This single generalized stencil is used instead of a set of stencils or a whole stencil table. This generalized stencil is designed for the most complex case and is also applied to all other less complex

cases through certain simplifications, like edge collapses. These simplifications are made possible through the concept of virtual interface points [10].

The group around the authors of the lattice cleaving method have also published an open source implementation of their algorithm, titled *Cleaver* [II].

3.3.2 Unstructured Mesh Cleaving

Instead of using fixed background lattices, it is also possible to generate an unstructured background mesh based on the characteristics of the implicit surface representation [39]. Although this approach may be promising, the added complexity of creating a fitting background mesh was considered to be outside of the scope of this work.

3.4 This Work's Contribution

This work implements a meshing algorithm based on the idea of the isosurface stuffing and mesh cleaving algorithms. The biggest influence is taken from the mesh cleaving algorithm [10], [40], as this work represents a further development of it.

The adaptions of the mesh cleaving algorithm and its implementation are necessary due to the different LS definition used for the input and its implications. The issues faced by working with the LS definition employed in this work (compare Section 2.1), are coming from the sparse nature of the input to the meshing algorithm, and the different understanding of the material regions by using the LS wrapping approach (see Section 2.1.3.1).

In order to create the substructure octree, an algorithm was devised which works on the the sparse LS input instead of a dense input, or a LSF given as a function which can be evaluated at arbitrary points. Both the *Quartet* [I] and the *Cleaver* [II] implementations require a full dense set of gird points.

The mesh cleaving algorithm, employed in this work, which adapts the background mesh to the implicit interfaces, was redesigned considerably in order to accommodate for the LS wrapping approach since it leads to strong implications for the location of interface points.

Chapter 4

Tetrahedral Meshing Algorithm

As mentioned in Chapter 3, the algorithm described throughout this work is an adaption of the so called *lattice cleaving* [10] algorithm, which is founded on the *isosurface stuffing* [14] algorithm. Both of these algorithms use a set of rules to modify a body centered cubic (BCC) background lattice in order to generate their output mesh which approximates the boundary of the meshed domain. The idea of changing the background grid was originally inspired by [24].

While certain parts of the mentioned algorithms use the fact that the employed background lattice is a BCC lattice, the main parts do not depend on it. The fact that the sets of rules are general enough to work with different background lattices has been shown for both the isosurface stuffing as well as the lattice cleaving algorithms. In [38], the isosurface stuffing algorithm was implemented using an acute A15 lattice instead of the BCC lattice. The lattice cleaving algorithm was even shown to work with unstructured background meshes in [39], and may therefore be described by the more general term *mesh cleaving* instead of *lattice cleaving*. Since there is no clear definition of the background mesh, there is no convention on what the terms mesh cleaving and isosurface stuffing encompass. They may refer to the entire procedure from the level set input to the output mesh, or only to the part of the algorithm adapting the background mesh to the domain interfaces. In this work, the term mesh cleaving will be used to describe the adaption of the background mesh to the implicit interfaces. This is reasoned by the fact that this technique for conforming the background mesh can be used on arbitrary background meshes, and that the authors of the mesh cleaving algorithm state, that algorithms like the isosurface stuffing or mesh cleaving can be thought of as what they termed *mesh processing algorithms* [39].

4.1 Structure

The meshing algorithm creates the output mesh based on the level set input in two parts - the generation of a background mesh, and the *mesh cleaving* of this background mesh, to conform it to the interfaces. The generation of the background mesh is further composed of two main steps - the creation of an octree and filling it with a lattice based background mesh.

In this way, the implementation can be summed up in three main steps:

1. Octree creation - Create and manipulate an octree based on the level set (LS) input.
2. Background mesh creation - Create a background tetrahedral mesh based on the octree substructure and a chosen lattice.
3. Mesh cleaving - Make the tetrahedra of the background mesh conform to the material interfaces of the domains described by the LS input.

The work's implementation was, by design, modularized into different parts, from the first reading of the LS input into an octree, to the creation of the final tetrahedra outputted as the generated mesh. This allows for a straight-forward exchange of individual key parts for a further development of this approach to the meshing problem.

The detailed explanation of the algorithm is structured in the same way, as it was modularized in the implementation. First the creation and filling of the octree is described in Section 4.3, then the lattice selection and generation of the background mesh is discussed in Section 4.4. Finally, the mesh cleaving is detailed in Section 4.5.

4.2 Input and Output

The input to the mesh processing algorithm consists of only one level set function (LSF) per material, taking into account the LS wrapping approach, where each LS must form a fully closed surface. A major goal of this work was to enable the use of sparse LSs as inputs. The octree-based background grid creation proposed in this work was improved to allow for sparse input, but is also compatible with dense LS input data. It is, however, restricted to LSs, whose LSF is based on distance measures in the Manhattan norm.

The output is a tetrahedral mesh, given in the form of the mesh data structure explained in Section 2.3. The mesh itself can be expected to be composed of good quality tetrahedra, but the explicit representation of the material interfaces should be expected to be only an approximation of the input's actual material interfaces.

While the algorithm offers a grading towards the inside of the individual material domains, there is currently no grading on the interfaces themselves.

For further details and implications regarding the output, refer to Chapter 5.

4.3 Octree Creation

The octree required for the second main step of the algorithm, the creation of the background mesh, is generated in two stages. First, the level set (LS) input is loaded into the octree as described in Section 4.3.2. Then, the octree object is filled in a graded manner, see Section 4.3.3, resulting in a 2 : 1 edge-balanced octree.

The octree which is created based on the LS input and manipulated afterwards, directly influences the background lattice or mesh, and in turn the output mesh. The octree's influence hereby lies mainly in the grading of the mesh, which also indirectly affects the tetrahedra quality. Based on the octree's position and rotation, the resulting alignment of the octree's nodes can be different, which affects the location and rotation of the generated background tetrahedra. As a result, the background mesh may be aligned to large parts of the surface, which may lead to better quality tetrahedra, created by the respective part of the algorithm.

Changing the octree's interior filling however, noticeably and predictably affects the mesh's grading. Hence, in order to allow for future improvements of the algorithm, the creation and filling of the octree was modularized, so each step is exchangeable. For example, one may specify a region of interest, in which smaller tetrahedra are required. Such a region could already be refined in the octree, instead of refining the tetrahedral mesh afterwards, making the process computationally more efficient.

4.3.1 Octree Implementation

For the implementation of the meshing algorithm, a child-array-based octree was chosen to increase performance due to a denser, more contiguous, memory layout. Additionally, aside from the region containing the outer-most interface (material with largest identifier (ID) to the outside), any node with children, is always filled with all eight children in this use-case.

Since the output mesh will be graded in size, the octree features leaf nodes of different sizes. This makes a balanced octree necessary for a few reasons. Firstly, as the main requirement is to generate a high quality mesh, the difference in size between neighboring leaf nodes of the octree cannot be arbitrarily large. Otherwise, the background grid tetrahedra bridging between different sizes of standard background-lattice tetrahedra will be of low quality. The quality of the bridging

tetrahedra, of course, does not solely depend on the size difference between the lattice cells being bridged, but also on the used lattice itself. Some lattices allow bridging with better quality tetrahedra than other lattices.

Secondly, the background lattice is placed into the nodes of the octree, so there is a limited number of tetrahedra for the lattice and the bridging between different sized neighboring nodes. However, the larger the balancing factor is, the larger this limited number of required tetrahedra becomes, which increases complexity of the mesh. Depending on the chosen lattice, the bridging of differently-sized elements may become more or less difficult.

Motivated especially by the quality of the bridging tetrahedra, the octree was implemented to be a $2:1$ edge-balanced octree. Corner-balancing of the octree is not necessary as a corner only consist of a single vertex which unlike edges or faces, cannot be refined, and therefore does not need special consideration. Labelle and Shewchuk [14] use a $2:1$ edge-balanced octree in the graded interior version of their isosurface stuffing algorithm. Bronson, Levine, and Whitaker [10] do not mention any balancing of their octree, but one can assume that they used $2:1$ edge-balancing, since they work with a body centered cubic (BCC) lattice. A simple $2:1$ face-balancing is not sufficient for the proposed way of filling the octree, as such an octree may lead to nodes sharing an edge having a $4:1$ factor between them.

In the graded interior version of their isosurface stuffing algorithm, Labelle and Shewchuk [14] use an octree, in which a node with children does not require all eight children. Additionally, Labelle and Shewchuk [14] design their bridging tetrahedra in such a way that they do not only bridge between lattice cells of twice/half the size, but also between any child node and its parent node. This bridging between children and parent nodes further reduces the number of created background tetrahedra and, in turn, reduces the number of output tetrahedra, which is why graded meshes are used in the first place. In this work, the octree was implemented such that not all eight children of a node need to be defined, but the bridging between children and their parents is not used. The reason for not using children-parent-bridging is that it might introduce challenges for certain background lattices, as there may not be a bridging configuration of good tetrahedral quality for complex lattices. Therefore, without loss of generality on the used background lattice, children-parent-bridging has not been implemented, although future implementations may introduce it as an optional feature.

Since not all eight children of the octree need to be allocated, parts of the octree which lie outside of the implicit material, do not need to be stored, as they can be excluded a priori, thus speeding up execution time and reducing memory requirements.

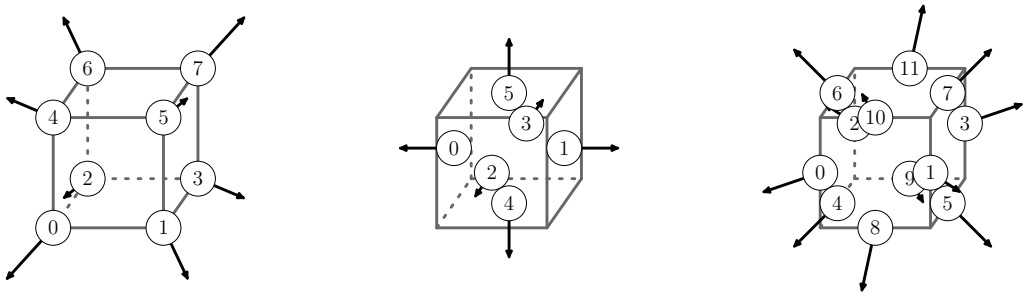
4.3.1.1 Indexing

In order to navigate the octree, this work uses an indexing scheme which allows for relations to be calculated by bit operations. The basis of this scheme is the location based on a grid, as well as numbering of children, corners, edges, faces, which are loosely based on binary representations of three-dimensional (3D) vectors.

The grid used by this work's octree implementation is a regular integer grid with a grid spacing of unity between two nodes of the smallest possible level. Therefore, any smallest possible node in this octree takes the space of a unity cube on the grid. The octree-internal grid only consists of the positive octant of the Cartesian 3D space. In order to represent an arbitrary domain, including negative coordinates, the octree-internal grid's origin must be set to the minimum point of the domain's bounding box. Since the input to the octree also lies on a grid, the octree stores a vector shifting between the input grid including any negative coordinates, and the - positive only - internal grid. A grid-index on the input grid, which will be termed *true grid-index* or *true location*, is transformed to an *internal grid-index* or *internal location* using the mentioned translation vector. One reason for using integers for both, the internal and input grids, is that locations given as integer vectors can be compared exactly. For floating point type comparisons to work reliably, a more elaborate comparison would have to be introduced.

The numbering of the children, or *octants*, is based on a binary representation of the corresponding direction starting from the center of the cube, referred to as corner-direction. The binary direction consists of one bit for each Cartesian direction, where 0 means towards the negative direction and 1 towards the positive. The least significant bit represents the x-direction, the next significant bit the y-direction, and so on. The corner-direction vectors and the resulting numbering can be found in Fig. 4.1a and Table 4.1a. While using one bit per dimension works ideally for the eight octants, the numbering of the 6 faces and the 12 edges, has to be performed differently. The idea is, however. The faces are numbered by going over the negative and positive directions, followed by the three dimensions, as can be seen in Fig. 4.1b and Table 4.1b. For the edges, the same is performed with two dimension at a time, compare Fig. 4.1c and Table 4.1c.

Using this numbering for the octants, the path to any point on the internal grid can be calculated simply from the grid point's coordinates. When calculating the path to an octree node, it is however important to understand that a point on the underlying grid is not equal to a unique octree node. This is the case because any node in the 0-octant and its parent, share the same location in the internal grid. In this way, the internal grid's origin hosts as many octree nodes, as the octree has levels. In order to calculate a path to a specific octree node, its location on the grid and its level in the octree is required. After shifting the bits of the grid point location to the right by the desired level, it is a simple matter of taking



(a) Corners or octants. (b) Faces. (c) Edges.

Figure 4.1: Depiction of the indexing of direction vectors used in the octree. For the exact values of the direction vectors see Table 4.1.

Id	Direction Vector
0	$(-1,-1,-1)^\mathsf{T}$
1	$(\ 1,-1,-1)^\mathsf{T}$
2	$(-1,\ 1,-1)^\mathsf{T}$
3	$(\ 1,\ 1,-1)^\mathsf{T}$
4	$(-1,-1,\ 1)^\mathsf{T}$
5	$(\ 1,-1,\ 1)^\mathsf{T}$
6	$(-1,\ 1,\ 1)^\mathsf{T}$
7	$(\ 1,\ 1,\ 1)^\mathsf{T}$

(a) Corners or octants.

Id	Direction Vector
0	$(-1,\ 0,\ 0)^\mathsf{T}$
1	$(\ 1,\ 0,\ 0)^\mathsf{T}$
2	$(\ 0,-1,\ 0)^\mathsf{T}$
3	$(\ 0,\ 1,\ 0)^\mathsf{T}$
4	$(\ 0,\ 0,-1)^\mathsf{T}$
5	$(\ 0,\ 0,\ 1)^\mathsf{T}$

(b) Faces.

Id	Direction Vector
0	$(-1,-1,\ 0)^\mathsf{T}$
1	$(\ 1,-1,\ 0)^\mathsf{T}$
2	$(-1,\ 1,\ 0)^\mathsf{T}$
3	$(\ 1,\ 1,\ 0)^\mathsf{T}$
4	$(-1,\ 0,-1)^\mathsf{T}$
5	$(\ 1,\ 0,-1)^\mathsf{T}$
6	$(-1,\ 0,\ 1)^\mathsf{T}$
7	$(\ 1,\ 0,\ 1)^\mathsf{T}$
8	$(\ 0,-1,-1)^\mathsf{T}$
9	$(\ 0,\ 1,-1)^\mathsf{T}$
10	$(\ 0,-1,\ 1)^\mathsf{T}$
11	$(\ 0,\ 1,\ 1)^\mathsf{T}$

(c) Edges.

Table 4.1: Indexing of direction vectors used in the octree shown in Fig. 4.1.

the bits, one by one, from each component (x, y, z) and arranging them in the mentioned way such that the x-component is the least significant bit. The list of 3-bit numbers resulting from that process contains the octant numbers which should be taken, starting from the root node, in order to end up at the desired octree node at the given location and level.

Point-data, like the stored level set function (LSF)-values, is about a single specific point in space. A single point is not uniquely provided by an octree node. In order to be consistent with the octree's internal grid, point data stored in an

octree node is always regraded to be about the node's grid location, or in other words, the node's 0-corner.

4.3.2 Loading of Level Set Input

The implemented octree has to be created with a fixed geometric size, a bounding box. This is required by the way the octree was implemented in this work, using the internal grid for referencing, as described in Section 4.3.1.1. Therefore, the LS input has to be analyzed for its geometric extent before inserting the first node into the octree. In order to find the bounding box of the the input, one only needs to go over the LSF dataset input of the material with the largest material ID. This is the case as the input has to adhere to the LS wrapping approach. The material with the largest ID wraps all other materials and therefore represents the total extent of all defined materials.

To find the bounding box of the input, the algorithm goes over every entry in the largest ID material's LSF dataset. For every entry, it checks the position vector in order to find the minimal and maximal value in each component (x, y, z) of all the position vectors in the dataset. Those component-wise minima and maxima are used to define the bounding box. The largest edge length of this bounding box is additionally padded by a user-parameter. The resulting size is then brought to the nearest larger power of two value, as the size of an octree has to be a power of two. The padding, provided by user input, is then used to reserve space to allow for the creation of additional octree nodes on the outside, in the case that potential future manipulations of the octree make this a necessity.

Once the octree data structure is set up with the determined size, the LS data is simply copied into the octree. From smallest to largest material id, for each material, the LSF dataset is loaded. Iterating over the LSF dataset, every entry is either created in the octree based on the entry's location, or simply updated, in the case that an octree node was already created at the entry's location by a material with a smaller ID.

For the creation of a new octree node, first the location stored in the LSF dataset's entry is brought to the integral grid indexing by dividing each component by the LS grid delta. The true location, in form of the grid index, is then shifted to the octree's internal grid index. This internal grid index is then used to determine the path or branching to the - yet to be created - octree node. When there is no octree node present at the determined branching, a new node will be created with the smallest leaf node size possible in the octree. Regardless of whether the node was freshly created or already present, the LSF-value of the current LSF dataset's entry is copied to the node's list of LSF-values, at the corresponding material ID.

Once the LSF data set of every material has been copied over, the octree can be viewed as representing the voxelized hulls of the individual material domains - of course including their wrapping.

As an additional note, it should be mentioned that during this initial loading of the LS input into the octree, the individual LSF are filtered by their absolute value using a threshold. In this way, a dense LS input does not fill up the complete octree with smallest sized leaf nodes, but only creates the mentioned material hulls, representing the wrapped material interfaces.

4.3.3 Graded Filling

After copying all the LSF-values as leaf nodes of the smallest node size into the octree in the previous step, the octree's leaf nodes only run along the material interfaces. The insides of those hulls of the different materials have to be filled without voids for the generation of the background grid later on. This filling of the domain hulls is done in a graded way, such that the resulting octree is already $2:1$ edge-balanced.

For the graded filling, the octree's leaf nodes are iterated over in level by level fashion, from the smallest leaf-nodes to the largest. Each octree level is iterated over twice in succession, before going to the next larger nodes. This is the case as two steps are performed for each level. In the first step, the parent nodes are *completed*; in the second step, some neighboring nodes of the next larger level are created. In this way the resulting octree is automatically $2:1$ edge-balanced, assuming that the input is a $2:1$ edge-balanced octree, like an octree stemming from the initial loading of LS data in Section 4.3.2.

4.3.3.1 Single-Material Case

For the first step, the *completion* of the parent nodes, the iteration goes over all leaf-nodes of the current level, ignoring all those which are outside of the LS. On each inside leaf-node, all face directions which are *direct-sibling-directions*, are checked for already existing neighboring leaf-nodes, which means nodes of the same parent. In case no neighbor is found, it is put on a list of to-be-created nodes. The neighbor's LSF-value is calculated by advancing the original node's LSF-value in the negative direction and distance (compare Section 2.1.2.2).

Direct sibling edge-neighbors are only put on the list of to be created nodes in case both of the corresponding face-neighbors were put on the list of to-be-created nodes. Similarly, the direct sibling corner-neighbor is only put on the list, if all three of the current node's direct sibling face-neighbors are to be created. Therefore, the current node's parent has been filled for all its children and thus dubbed *completed*. Completed, in this context, means that all children who are

inside the material have been defined. Children outside of the material, can be skipped, since they would be discarded later on.

After going over all leaf-nodes at the current level, the list of to-be-created neighboring nodes is iterated over to actually create those nodes. To-be-created nodes can be on the list multiple times, in case they have been put on the list by multiple different nodes. When such a node is encountered the first time, it is simply created. Any time it is encountered thereafter, the node already exists in the octree. Therefore, the existing LSF-value is compared to the to-be-inserted LSF-value of the current list entry. The value being closer zero (in absolute value) is kept, as required by the notion of the signed distance function (SDF) underlying the used LSF.

The reason for this two stage approach is that directly inserting new children would mean that they could be iterated over directly after being created, which would lead to overfilling. Additionally, as mentioned above, the final value is the smallest absolute value generated by all neighboring nodes, which can be compared more easily in the selected approach.

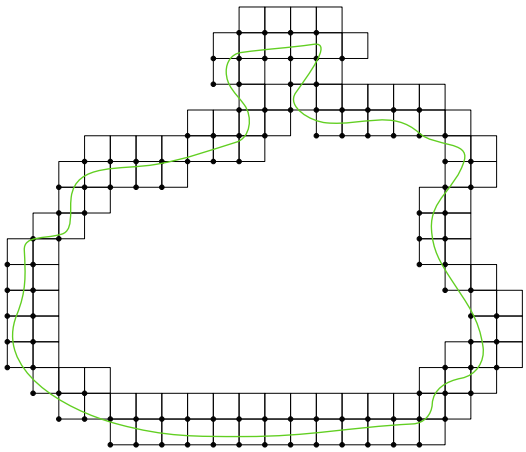
Following the first step, any holes to potential larger neighbors have been filled, as is shown for a two-dimensional (2D) example in Fig. 4.2c and Fig. 4.2e for two different node sizes, respectively. In the second step, those larger neighbors are created, as shown in Fig. 4.2d and Fig. 4.2f. One iterates over the same level leaf-nodes as in the first step. As the LS inside should be filled, only leaf-nodes inside the material are considered. Any considered leaf-node which is either in octant 0 or 7 checks its three directions which are not direct-sibling-directions. In similar fashion to the first step, in every direction, where no neighboring node is found, a node of the next-higher level - the parent level - is put on a list of to-be-created nodes. The LSF-value of the to-be-inserted node is again calculated by advancing the original node's LSF-value in the negative direction and distance. As the resulting octree must be $2:1$ edge-balanced, not only $2:1$ face-balanced, all edges also need to be checked for empty spaces on which neighboring nodes have to be created as well.

Once the iteration over the octree has completed, all nodes on the to-be-created nodes list are inserted in the same way as in the first step, once again comparing LSF-values and selecting the one closer to zero where needed.

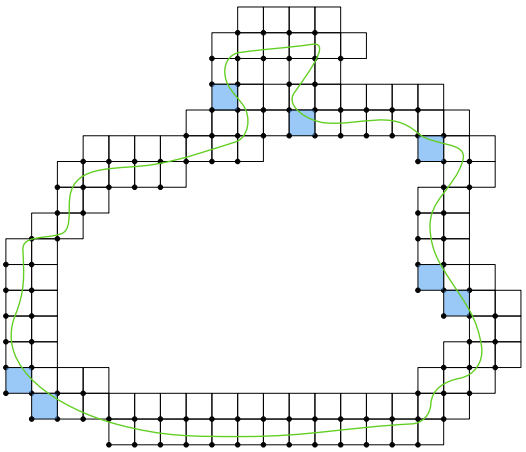
This two-step process is repeated, each time increasing the level. In this way all levels can be processed.

The complete single-material graded filling process is sketched in Fig. 4.2, including the step of creating additionally needed octree nodes, as discussed later on in Section 4.3.3.3.

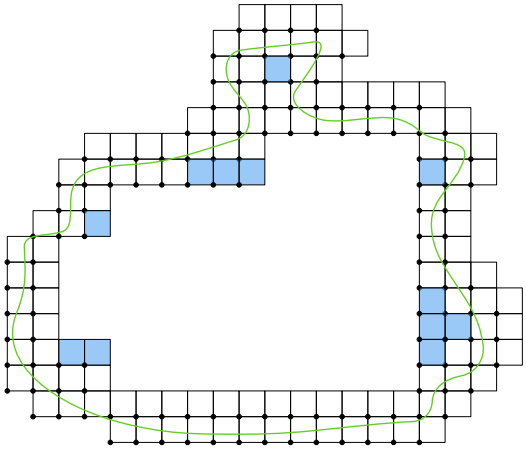
The single-material case explains the idea behind the implemented graded octree filling. The multi-material case, however, requires a few adaptions.



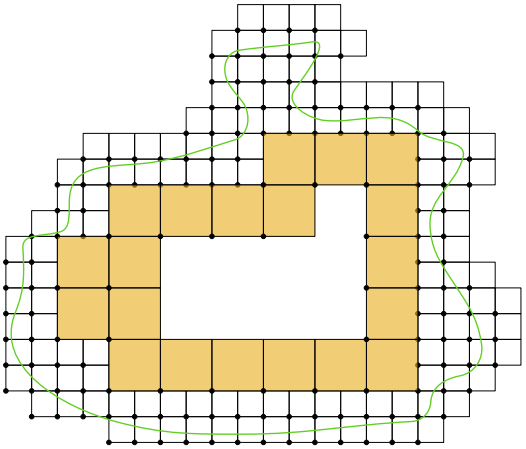
(a) Nodes after initial loading of LS data.



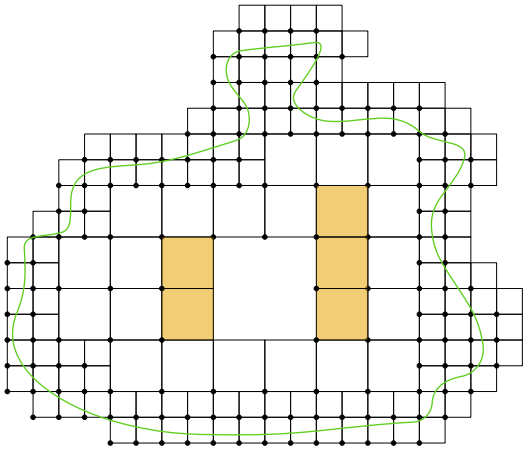
(b) Creating additional nodes at the interface.



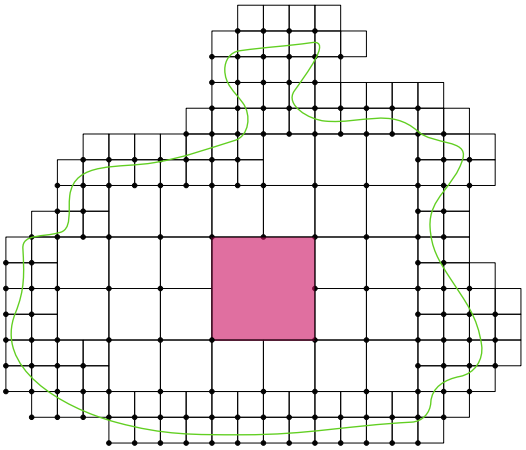
(c) Complete nodes at current level.



(d) Create neighboring nodes of next-larger size.



(e) Complete nodes at current level.



(f) Create neighboring nodes of next larger size.

Figure 4.2: 2D example of the octree's graded filling process, using a single material domain.

4.3.3.2 Multi-Material Case

In the multi-material setting, the concept of using the interfaces as a barrier and only filling the inside needs to be changed towards advancing the LS information in both directions from the interfaces. This is a direct consequence of using the LS wrapping approach and best exemplified by two concentric shapes. When advancing only towards the inside from every material interface, the resulting octree cannot be $2:1$ edge-balanced, excepted for a few special configurations. This is because advancing from the outer interface towards the inside produces larger and larger nodes, until they hit the small initial nodes of the inner interface, creating voids and thus an unbalanced octree.

Looking at a given level, in the first step of the two step process, all leaf-nodes which are inside any material need to be considered, and only leaf-nodes assigned to the outside void material are skipped over.

Whenever adding new neighbor nodes to a list of to-be-created nodes, the LSF-values are advanced in different directions per material. A material's LSF-value which is negative (inside) on the source node will be advanced in the negative direction towards further inside. A material's LSF-value which is positive (outside) will be advanced in the positive direction towards further outside.

This two-sided advancing creates the need for two different representations of the same material B in the LSF-values of a given node. On the one side, the material's inside-outside surface is advancing inwards, creating nodes with the classical representation using negative LSF-values for B. On the other side, the inside-outside interface of the next lower material A is advancing outwards, creating nodes which also have to represent the inside of B. However, the nodes created outwards from A's inside-outside interface, do not know how far they are within in the domain of material B, so they cannot set a negative value for B. Instead they simply advance their value for A in the positive direction, making it larger with each advancement. Therefore, the inside of B should both be represented by material B being the lowest material ID having a negative LSF-value - and material A being the largest material having a stored explicit LSF-value. Examples of the mentioned different material representations are depicted in Fig. 4.3.

In order to facilitate this additional sparsity in the data storage, the way the material is assigned to a point $\mathbf{p}$, is made a bit more involved in this work. There are two key aspects to this. Firstly, not all materials' LSF-values need to be stored at every point. Rather, only those LSF-values belonging to materials whose domain is near the point should be stored; it also may be sufficient to only store a single negative LSF-value of the material to which the point $\mathbf{p}$ belongs. The second aspect is that a point $\mathbf{p}$, being inside the region of material m, cannot only be indicated by a negative LSF-value for m, but also by a positive LSF-value of the material $m-1$.

Both of the two aspects for multi-material sparsity are bound to further constraints. Namely, for the first aspect, storing just a single material m's negative LSF-value to indicate that a point $\mathbf{p}$ belongs to m, requires that $\mathbf{p}$ is not also inside the region of material $m-1$. In other words, a single negative LSF-value is sufficient to indicate the material assigned to a point, as long as the negative value indeed belongs to the lowest material id, still having a negative value at said point. For the second aspect, the consideration is similar, since storing a material $m-1$'s positive LSF-value to indicate that a point $\mathbf{p}$ belongs to material m, requires that calculating material m's value at $\mathbf{p}$ would actually lead to a negative value, or an inside value. In other words, it is the opposite of the first case - it requires that the stored positive LSF-value indeed belongs to the largest material ID having a positive value at said point.

Figure 4.3: Example along a line showing different ways to represent multiple materials through the stored LSF-values.

4.3.3.3 Additional Octree Nodes

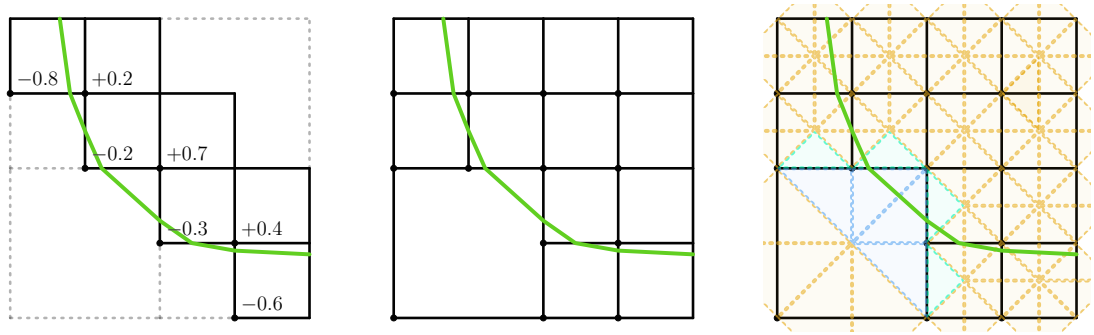
Filling the octree in the described graded way uses the nodes created by the initial loading of the LS like a barrier. Due to the geometric or volumetric difference between a single grid point of the LS input and an octree node, this barrier is sometimes not adequate or thick enough, so a few additional nodes might be necessary. The main reason behind the additional nodes can be considered to be the implemented lattice-based background mesh creation, which will be detailed in Section 4.4. This is concerned with suboptimal tetrahedra lying at or near interfaces, or even a complete lack of tetrahedra on the interface to the outside void.

4.3.3.4 Suboptimal Tetrahedra at Interfaces

The mesh cleaving section of the algorithm modifies only those background mesh tetrahedra which are cut by a material interface, or lie close enough by an interface such that an incident vertex is moved. The standard BCC lattice tetrahedra have better dihedral angles than all of the bridging tetrahedra (see the upcoming Section 4.4.1.1). Since, for the final mesh quality, it is favorable to modify background mesh tetrahedra which already start with better dihedral angles, it is consequently

better to modify only standard BCC lattice tetrahedra. This, in turn, means that near (or at least along) the interface, there should be only standard BCC lattice tetrahedra. To ensure this to be the case, both bridging tetrahedra coming from edge-only neighbors and face neighbors need to be considered.

The first kind of suboptimal tetrahedra at interfaces are as shown in Fig. 4.4. They are caused by edge-only-neighbors of different sizes. The difference in size leads to the creation of bridging tetrahedra in the edge-neighbor and the interface runs strait through them. Within the nodes initially loaded into the octree, in such a case also bridging tetrahedra have to be created. These bridging tetrahedra can also be affected by their close proximity to the interface.



(a) Set of interface nodes initially loaded into the octree.

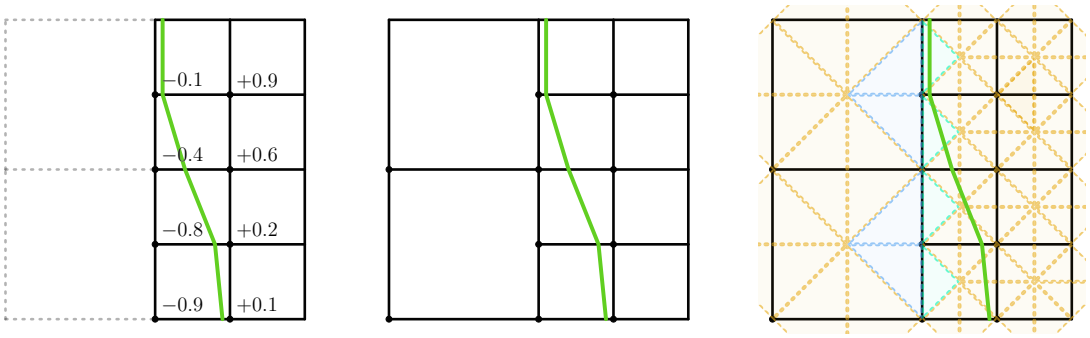
(b) Creation of larger neighbor nodes by the octree's graded filling, without resolving suboptimal tetrahedra at interfaces.

(c) Bridging tetrahedra are needed for the grading. The interface (green line) not only runs through the standard BCC tetrahedra (orange), but also through the different bridging tetrahedra types (blue and green) which is suboptimal.

Figure 4.4: 2D example of the first part of the problem with suboptimal tetrahedra at interfaces.

The same analysis can be made for the second case of the problem with suboptimal tetrahedra at interfaces. There, the bridging tetrahedra are created within the initial barrier octree nodes. An octree node which is incident to the interface, is only completely filled with standard BCC tetrahedra if none of its face-neighbors or edge-only-neighbors are of different size. Bridging tetrahedra in octree nodes which are within a barrier, however, only become affected by mesh cleaving if the interface runs through them or very close by, such that an incident vertex is warped. This depends on the configuration of the surrounding LSF-values and is exemplified in Fig. 4.5, where the interface runs through, or close to, a bridging

type tetrahedra in the upper three rows. The bottom row in Fig. 4.5 also contains bridging tetrahedra, but they are unaffected, as they are neither cut, nor close enough to the interface.



(a) Set of interface nodes initially loaded into the octree.

(b) Creation of larger neighbor nodes by the octree's graded filling, without resolving for suboptimal tetrahedra at interfaces.

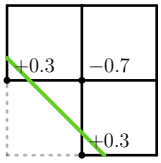
(c) Bridging tetrahedra are needed for the grading. The interface (green line) not only runs through the standard BCC tetrahedra (orange), but also through and near the bridging tetrahedra in the smaller nodes (green) which is suboptimal.

Figure 4.5: Second part of the problem with suboptimal tetrahedra at interfaces.

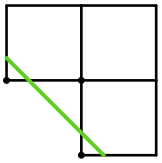
The problem of suboptimal tetrahedra at interfaces does not only arise with neighbors which differ in size, but also when there are no neighbors at all, which happens at the interface to the outside void phase. In this case, the tetrahedra are not suboptimal, in fact they are non-existent. Even though in the presented implementation, octree leaf nodes by default also create standard BCC tetrahedra reaching into the void regions next to them, whenever there is no neighbor in this direction (compare Section 4.4.2) this is still insufficient to cover every possible interface running through. This style of tetrahedra reaching into the void is depicted in Fig. 4.6, highlighting the issue of these cells.

It is unavoidable to create additional nodes for solving the problem of the non-existent tetrahedra on the interfaces towards the outside void phase, as shown in Fig. 4.6. For the two cases of the suboptimal tetrahedra in Fig. 4.4 and Fig. 4.5, it is favorable for element quality, but not mandatory for representing the material domains correctly.

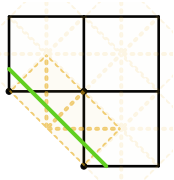
In [14], all three cases are solved with the creation of the initial octree nodes. The initial nodes are not simply loaded into the octree as is the case for this work,



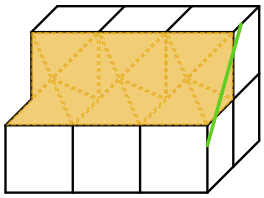
(a) Set of interface nodes initially loaded into the octree. The shown material interface, is the interface to the outside void, so it is the inside-outside-interface of the material with the largest material ID.



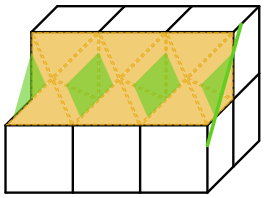
(b) Due to being the interface to the outside void, no neighbors are created, without resolving for suboptimal tetrahedra at interfaces.



(c) Standard BCC tetrahedra (orange) filled into the octree nodes for the background mesh, important tetrahedra for this example are highlighted.



(d) Reduced 3D view shows the gaps between the tetrahedra which are created into the empty void.



(e) 3D view exposes the insufficiency of the created tetrahedra as the interface surface (green) is not fully covered by them. Therefore, interfaces running along the edge/corner of the octree nodes are not represented correctly.

Figure 4.6: Example of how the interface towards the outside void can be insufficiently represented along edges and corners, without the creation of additional octree nodes on the outside.

but are found and created through what is called *continuation* with reference to [41]. Through the continuation approach and the *continuation condition* stated in [14], the initial interfaces nodes and the mentioned additional nodes are automatically created. Although the continuation condition is a simple solution, it could not be fully applied in this work. The first reason is that it requires quickly querying the LSF value at any point in space, which is not possible in the case of this work, due to the discretized input.

Secondly, all of the suboptimal tetrahedra are removed through additional nodes in [14] in order to maximize the mesh quality. The downside of this is that there is no refinement on the surface, but only towards the inside of the volume. Through the use of the lattice and the octree substructure, any refinement requires bridging tetrahedra, which is not permitted in the approach of [14]. Exactly this

refinement on the surface, however, is required for the planned technique to resolve thin features in the future. With thin features being thin regions, which were originally preserved through the LS wrapping, but are otherwise under-resolved by the mesh discretization. Therefore, this work's implementation always creates the mentioned mandatory additional nodes towards the outside void region, but the other additional nodes are not considered.

4.3.4 Discussion

When creating and filling the octree, as proposed in Section 4.3.2 and Section 4.3.3, the 2 : 1 edge-balancing occurs automatically whilst creating and filling the octree, instead of having to subsequently balance the octree. The octree balancing, therefore, also takes place at a known small number of iterations through the octree, without any cascading effects. The graded filling of the octree (Section 4.3.3) can thus happen in one iteration over the entire octree for creating needed diagonal nodes (compare Section 4.3.3.3), followed by two iterations over each level of the octree.

This implementation also allows for easy additional increments in the octree grading. The next larger nodes in Section 4.3.3 could, for example, be created every n-th round instead of every round. This leads to a shallower internal grading.

4.4 Background Mesh Creation

The next step of the mesh processing algorithm is the creation of the background mesh which can later be cleaved.

In order to avoid confusion between the *nodes* of an octree and the *vertices* in a mesh, the terms *node* and *vertex* will not be used interchangeably, but rather only in their respective mentioned context.

One of the core ideas behind algorithms like the isosurface stuffing [14] or mesh cleaving [10] is to take a good quality background mesh and only change it near material interfaces, such that the output mesh actually approximates the input material domains. A simple way to achieve a good quality tetrahedral background mesh is to use a crystal lattice. Hence, the mentioned algorithms can be categorized as *lattice based* meshing algorithms. The idea behind these is not limited to lattice based background meshes, it also works on unstructured meshes. This was shown for the case of the mesh cleaving algorithm, where the original lattice based *lattice cleaving* algorithm [10] was demonstrated to work with small changes as the *mesh cleaving* algorithm [39] also works on unstructured background meshes.

For this work's implementation of the background mesh generation, the lattice based approach was used, hence only this approach shall be discussed here.

4.4.1 Lattice Based Approach

In a lattice based approach, the background mesh is created by filling the space with the crystal lattice's cell or more accurately with a tetrahedralization thereof. Filling the lattice cell into a regular grid, leads to a regular background mesh. In order to generate a graded background mesh, which is important for many simulations, the lattice cell needs to be filled into some form of graded grid. Typically, an octree is used to create the grading for the lattice, as is the case in this work. In order to produce a conforming mesh using such a grading, it is however necessary to additionally treat the transition between different sized lattice cells, where a cell is coincident with an octree node, so these two terms will be used interchangeably here. These transitions are achieved using special tetrahedralisations called bridging cells.

The quality of a background mesh is determined by tetrahedral elements used for its creation. Besides the quality of the individual elements, also the quantity and distribution of different tetrahedra in the mesh can have an influence, depending on the selected quality measure. The quantity and distribution are influenced by the modeled domain and by the way this domain is represented through the underlying structure of the octree. The tetrahedron quality of the lattice cell's tetrahedralization is usually of higher quality than the tetrahedra in the bridging elements. This is the case for the body centered cubic (BCC) lattice and for the acute A15 lattice used in [38], which are discussed in Section 4.4.1.1 and Section 4.4.1.2 respectively.

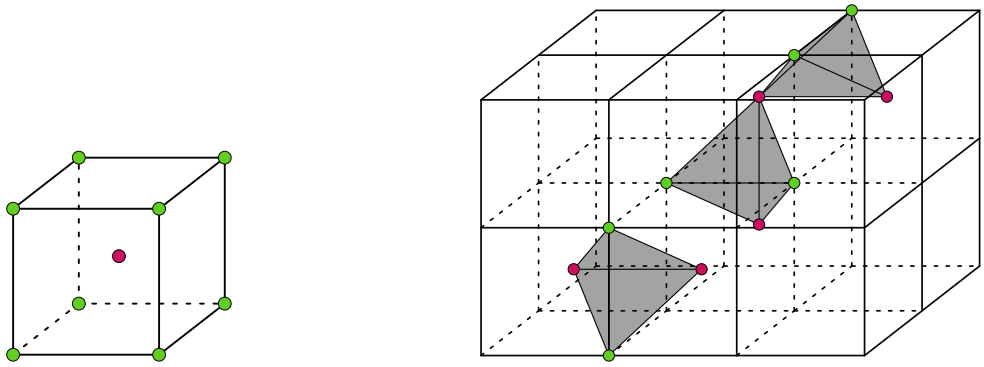
A lattice cell's tetrahedralization is not necessarily given directly by the lattice itself, as a crystal lattice cell only consists of a set of points. For a commonly used lattice, such as the BCC lattice, however, there is a well established tetrahedralization.

For the selection of a background lattice, there are a few key aspects to consider: The quality with which the lattice cell can be tetrahedralized, the quality with which the bridging regions can be tetrahedralized, and the complexity of both the cell and the bridging. Usually, defining consistent and high quality bridging elements is the most challenging of these aspects.

In this work, the BCC lattice was selected for the background mesh generation. Utilizing such a commonly used lattice brings more ease of implementation and a better comparability between the different methods and implementation. Especially regarding the mesh quality, fair comparisons between different implementations of the same type of algorithm are only possible on the same background mesh.

4.4.1.1 Body Centered Cubic Lattice

The BCC lattice's cell is based on a cube and consist of 9 vertices, the 8 corner-vertices and an additional vertex at the cube's body center, hence the name BCC. A single BCC cell is shown in Fig. 4.7a. Looking at space regularly filled with BCC cells, this can also be interpreted as two of the same cuboid grids with an offset of half a grid spacing in every direction. Its tetrahedralization is best looked at from the viewpoint of the two interlaced grids instead of a single lattice cell, as the tetrahedra are all between two cells each. Every BCC tetrahedron is composed of two body center vertices and two cube-corner-vertices, as depicted in Fig. 4.7b. Ignoring the neighboring cells in this way, a single BCC cell fans out 24 tetrahedra from its body center vertex.



(a) Single BCC lattice cell.

(b) Examples of standard BCC lattice tetrahedra.

Figure 4.7: The BCC lattice and the tetrahedra of the used tetrahedralization. Those tetrahedra are referred to as standard BCC lattice tetrahedra throughout this work.

For the 2 : 1 edge-balanced octree, three kinds of bridging tetrahedra will be used, which can be taken from literature [14], [23], [42] as there has been plenty of research on BCC tetrahedralization.

If two octree node share a face, each node uses a different cell to create a matching interface.

The first kind of bridging tetrahedra, which bridge smaller cells to larger ones, are always between the center, and three of the respective face's corner-vertices, as shown in Fig. 4.9. The corner-vertices are always selected such that the diagonal, created through tessellating the quadratic face in half, always runs from the larger cell's face-refinement vertex outwards.

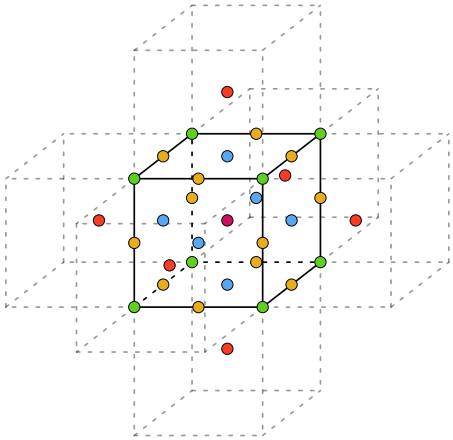
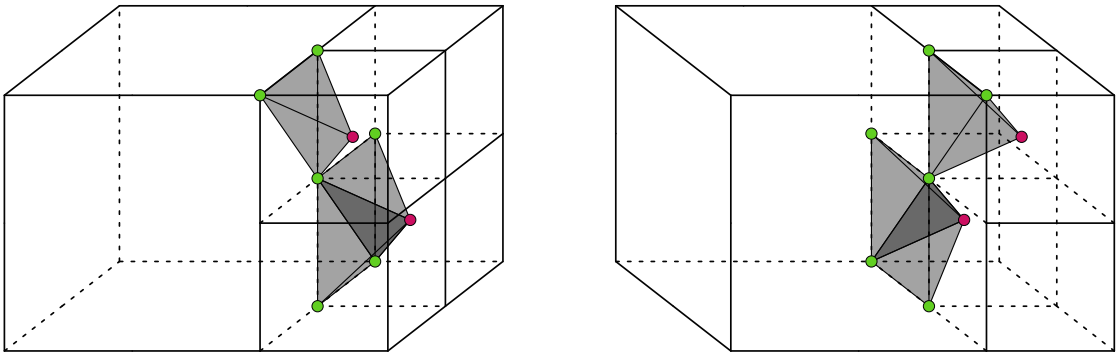


Figure 4.8: Different types of vertices considered for tetrahedralization of each BCC cell (including needed vertices from neighboring cells). Each node considers 8 ● corner-vertices, 1 ● center-vertex, 6 ● neighbor's center-vertices, 6 ● face-refinement-vertices, and 12 ● edge-refinement-vertices.

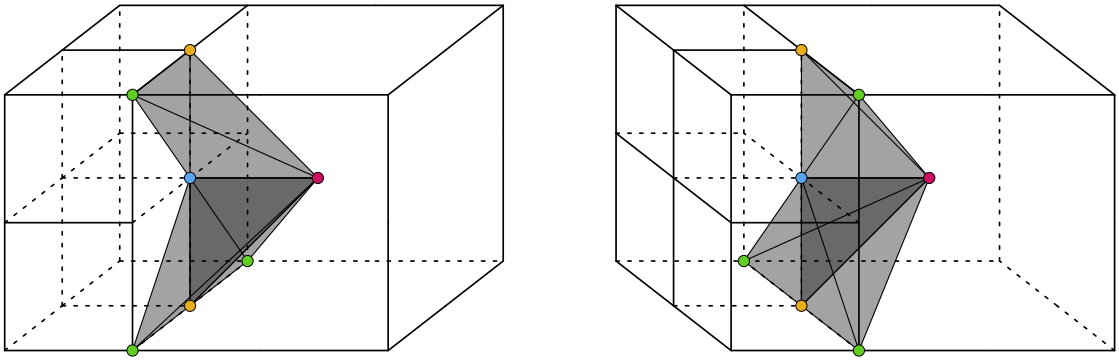


(a) View from the right. (b) View from the left.

Figure 4.9: Examples of the used BCC bridging tetrahedra of the first kind.

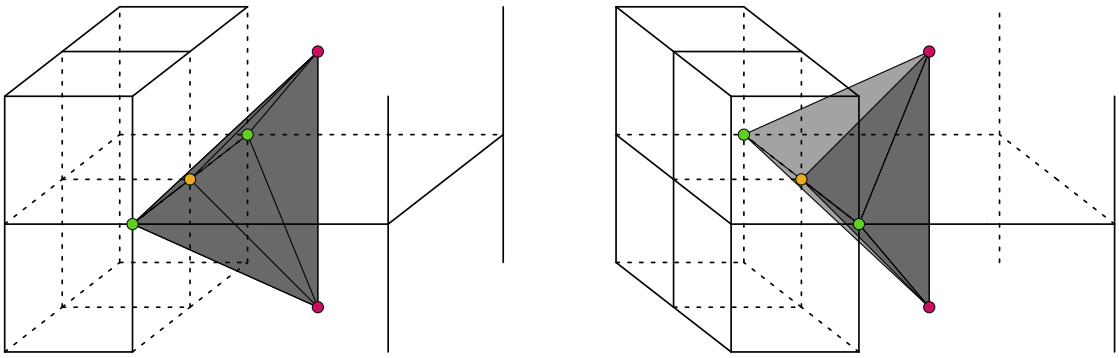
The second kind of bridging tetrahedra are on the opposite side of the face, in the larger cell. They are always between the center, a face-refinement, a corner and an adjacent edge-refinement vertex, as can be seen in Fig. 4.10.

The third kind of bridging tetrahedra is required, wherever a node has a face-neighbor of same size, but an edge incident to both is refined. In this case, the standard lattice tetrahedron is simply split into two halves, perpendicular to the refined edge, which is shown in Fig. 4.11.



(a) View from the right. (b) View from the left.

Figure 4.10: Examples of the used BCC bridging tetrahedra of the second kind.



(a) View from the right. (b) View from the left.

Figure 4.11: Examples of the used BCC bridging tetrahedra of the third kind.

4.4.1.2 Other Lattices

Another background mesh lattice option published in literature, is the acute A15 lattice employed in [38] for the *isosurface stuffing* type algorithm for the single material case. The modified A15 lattice of [38] offers better quality for the tetrahedralization of its lattice cell than that of the BCC lattice. The dihedral angles of the mentioned A15 lattice range from 53° to 79° [38]. However, creating feasible bridging elements for this lattice is a topic of ongoing research and therefor the A15 lattice was not considered in this work.

4.4.2 Implementation

Because the lattice based approach is used in this work's implementation, a background mesh is constructed from the previously-created 2 : 1 edge-balanced octree (see Section 4.3). The construction of the background mesh works by going over

the octree and filling each node using the previously defined stencils. Despite this name, before the placement of each stencil into an octree node, it has to be adapted according to a set of rules, described in the following.

When filling an octree node with a standard lattice cell or bridging cell, the operation is conceptually local to each individual node, once the correspond cell has been selected. The cell selection is based on its neighboring nodes and the resulting edge- and face-refinements to the node itself. Doing this in a local-to-the-node manner, however, would result in many duplicate vertices in the generated background mesh, e.g., a corner in an octree node can be incident to up to eight octree nodes and therefore up to eight stenciled cells. These duplicate points would subsequently have to be removed from the background mesh. As the vertex locations in the mesh data structure are represented by floating point type values, in contrast to the integral values used in the octree, this duplicate vertex removal would not only imply iterating over the entire created mesh, but would also be prone to floating point errors.

Therefore, this work's implementation resolves the duplicate vertices already during the stencil placement. All nodes are iterated over sequentially, whereby the stencil for the currently processed node is selected based on its neighbors' sizes and the resulting face and edge refinements. The selected cell type and its orientation dictate which vertices are required. The stenciling operation then checks the neighboring octree nodes for information about previously created vertices. Vertices which have already been created by the stenciling of other nodes are reused by their global vertex identifier (ID) in the background mesh. Any vertex still missing for the current node's stencil type is created and added to the background mesh, storing the information about its global vertex ID. In this way, any required vertex is created exactly once.

The level set function (LSF)-values stored in an octree node correspond to the coordinate of its 0-corner. A required stencil vertex may thus be placed at a coordinate for which the octree does not provide an LSF-value. In such a case, the LSF-values have to be approximated from values stored in surrounding octree nodes, or mesh vertices previously created in a similar fashion. The approximation depends on the configuration of the surrounding vertices' locations, which is, in turn based on the used lattice and tetrahedralization. An additional influence on the approximation of missing LSF-values is the configuration of known values and missing values at the set of surrounding vertices. For the case of the implemented BCC lattice tetrahedralization, the missing LSF-values are approximated using linear-, bilinear-, or trilinear interpolation, depending on the approximated vertex. For example, the LSF-values of face-refinement vertices can be approximated from the corner-vertices incident to the face using bilinear interpolation.

In total, there are 33 unique vertices which may be placed in a cell for all cell types, as shown in Fig. 4.8. Thus, any created cell uses a subset of these 33 vertices, which gives clear bounds on the maximum number of vertices created in the background mesh.

While only the discussed BCC lattice has been implemented in this work, in theory, the implementation of the background mesh creation algorithm allows for the use of different lattice stencils, through the inheritance of an abstract class representing the basic functionality required by a lattice stencil. Each such lattice stencil consists of a set of tetrahedralizations for the different cells - standard lattice cell and bridging cells - and all required vertices.However, an implemented lattice stencil also has to incorporate the rules for approximating the missing LSF-values at the required vertex locations, as those are heavily dependent on the geometric configuration of the selected lattice, and its stencil's tetrahedralizations.

4.4.2.1 Interpolating Level Set Values

The individual octree nodes in this work's implementation only store the LSF-values corresponding to their stored node-location - the node's zero corner (compare Section 4.3.1). The lattice stencil however needs vertices at additional locations not otherwise provided by the octree nodes. In the BCC lattice, these are the center of each octree node, the face- and edge-refinement vertices of the bridging elements, and vertices within the outside void phase near the interface. This extends also to other lattices, but in the specific case of the BCC lattice, where those additional vertices are the center of each octree node, and face- and edge-refinements in case of bridging elements. Also at the interfaces towards the outside void phase, additional nodes on the octree nodes boundary need to be approximated.

For each of these added mesh vertices, LSF-values have to be generated from nearby values. While this part lies in the context of the background mesh creation algorithm, the octree used as input to this background mesh creation can also impact the LSF-value calculations, as the constellation of the different sized octree nodes lead to easier or more complex ways of approximating the missing LSF-values.

Until now, new LSF-values have only been approximated by advancing (Section 2.1.2.2) the LSF-values of neighboring points. The concept of advancing is based on the previously-mentioned notion of having a barrier and therefore knowing whether the advancing is further inside the region or towards the outside. Therefore, the advancing does not work so easily at the interfaces. The calculation of in-between LSF-values is also required at the interfaces, for the calculation of the additional lattice vertices. In the background mesh creation algorithm, the required LSF-values are approximated by a combination of three different meth-

ods. The first one takes the LSF-values from an octree node at the corresponding location. The second one is the already used advancing of neighboring LSF-values, for example, from other corners of the same node. And the third way of approximating a vertex's LSF-values is by linear, bilinear, and trilinear interpolation, for center of edges, faces, or cubes (whole node), respectively. Due to the cuboid structure, this simply comes down to averaging over all corner values. The values gathered by these three means of approximation and from different sources, e.g., advancing from multiple neighboring corners, are then consolidated in the usual manner by keeping only the LSF-value closest to zero, for each of the individual materials.

4.4.2.2 Node Stenciling

The actual stenciling of the octree nodes is performed in breadth-first-order, meaning that the larger leaf nodes are treated first. This top-down approach aids in the approximation of the LSF-values. Each node can thus use the LSF-value approximations made by neighboring nodes which have already been stenciled for their own approximations. As the larger nodes can use interpolation as well, their approximations are expected to be improved. Therefore, processing the leaf nodes from largest to smallest enables the smaller nodes to reuse the more accurate approximations of their larger neighbors.

When stenciling a leaf node, three steps must be followed: First, the neighboring nodes are checked to find their refinement and vertices which they have already placed, leading to the type of stencil and the exact nodes which must be created. In the second step, all required vertices which have not been created by others are added to the background mesh and the LSF-values of all vertices are approximated and updated. In the final step, the vertices are used to form the selected stencil's tetrahedra, which are added to the background mesh.

4.5 Mesh Cleaving

The mesh cleaving algorithm converts a given tetrahedral background input mesh into a tetrahedral output mesh, which approximates the interfaces between the different materials.

4.5.1 Overview

Three main steps make up the mesh cleaving algorithm. First, material interfaces within the background mesh tetrahedra are sought and labeled. Second, constellations of background mesh vertices and labeled interface points that would lead

to bad quality output tetrahedra are resolved by modifying their location and labeling. Third, output tetrahedra are created based on the modified background mesh and its interface labeling.

Before looking in detail at the individual steps of the implementation of the mesh cleaving algorithm developed for this work, the general concept is introduced. In particular, the conforming of a tetrahedral background mesh to an interface and the generation of output tetrahedra will be discussed.

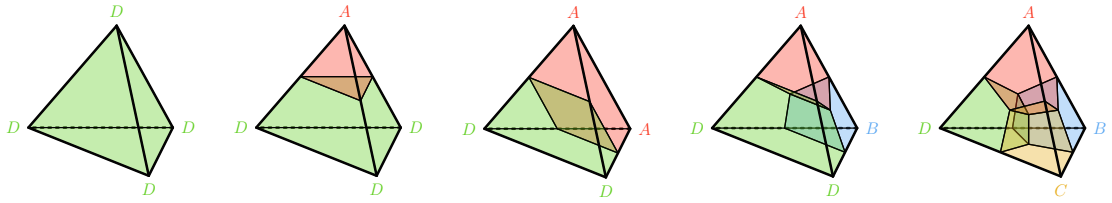
4.5.1.1 Interfaces

In the mesh cleaving algorithm, the interfaces between different materials are approximated in a linear fashion for the different simplices on which they can occur. Each vertex, or more explicitly each *corner-vertex*, of a tetrahedron is labeled with a single material identifier (ID) - throughout this work this is termed the material *assigned* to the corner-vertex. The simplex of next higher dimensionality is an edge. Each edge is incident to two corner-vertices, each with their own assigned material. In case the assigned materials of the two vertices are different, there must be an interface point on this edge, where the two assigned materials meet - this point is called a *cut-point* or *cut* for short. Any combination of three corner-vertices makes up one of the tetrahedron's faces. In similar fashion, at any such face, at most three materials are permitted to meet. If three materials meet at a face, it takes place in the form of a so called *triple-point* or *triple*. Finally, a tetrahedron itself can host at most four materials. If there are four different materials at the corner-vertices, the materials meet in a single point called a *quad-point* or *quad*.

This restriction of a single interface point per simplex approximates the actual boundaries in such a way that thin features are removed. For example, any material lying on an edge which is sandwiched by two other materials, is completely discarded, which is discussed in more detail in Section 4.5.3.1.

In certain cases, this removal of certain thin features is undesirable, especially when using the level set (LS) wrapping approach. In order to still adequately represent the thin features which were preserved by the LS wrapping, but would be removed by mesh cleaving, this work proposes that the creation of additional areas of finer resolution around thin features could be used to resolve the issue. Those areas of finer resolution are best incorporated during octree generation.

With the mesh cleaving's restriction to a single material assigned per corner-vertex within a tetrahedron, there are only five topological cases of material distributions possible, disregarding permutations. Those topological cases, or sometimes also stencil cases, are usually presented by the total number of cut-points in the tetrahedron, see for example [10], [30], [41] for the multi-material case. The mesh cleaving's five cut-cases are depicted in Fig. 4.12.



(a) 0-cut case. (b) 3-cut case. (c) 4-cut case. (d) 5-cut case. (e) 6-cut case.

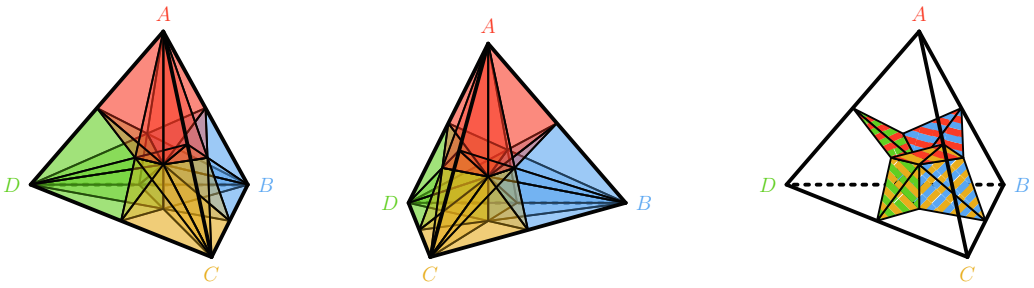
Figure 4.12: Examples of the different cut-cases - the material configurations based on the number of cuts. Depending on the case, up to four different materials A, B, C, and D are shown in red, blue, orange, and green, respectively. *Images are adapted from* [10].

Only the special case of the 0-cut, which does not represent any material interface, consists of a unique tessellation into tetrahedra. All other cases have at least one face of a material region which is a quadrilateral. A quadrilateral can be tessellated into triangles in two different ways and so there also cannot be a unique way of tessellating the other topological cases into tetrahedra. This means that tessellations of all the other topological cases must be created. In the case of the isosurface stuffing algorithm [14] this is done using a stencil table. This table consists of 12 stencils which may be rotated and reflected to fit the processed background grid tetrahedron. Algorithms considering only an isosurface, a single material interface, only require the topological cases of the 0-cut, the 3-cut and the 4-cut.

4.5.1.2 Generalized Stencil

The multi-material case calls for additional stencils for the 5-cut and 6-cut topological cases, which would make the stencil table larger. In order to avoid a larger stencil table, the mesh cleaving algorithm [10] uses a single generalized stencil instead. This stencil tessellates the topological case of the 6-cut, and is depicted in Fig. 4.13. The generalized stencil consists of 24 tetrahedra, which are referred to as *mini tetrahedra* throughout this work due to their small size compared to the octree grid. Each of the mini tetrahedra is created from a corner-vertex, a cut-point, a triple-point and the background mesh tetrahedron's quad-point. Therefore, if 4 materials meet at a single background tetrahedron, 6 mini tetrahedra will be created per material.

This, material-wise most complex, cut is directly used for all topological cases, by not only simply relabeling the materials assigned to the mini tetrahedra, but rather through edge collapses in the generalized stencil tetrahedron. In order to enable such edge collapses and actually generalizing the single stencil, each simplex



(a) Mini tetrahedra.

(b) Mini tetrahedra viewed from a different angle.

(c) All the possible material interfaces inside the stencil tetrahedron.

Figure 4.13: The mesh cleaving's generalized stencil. The four different materials A, B, C, and D are shown in red, blue, orange, and green, respectively.

needs to contain an interface point. Simplices which do not have an actual interface receive a so called *virtual* interface point. Hence, an edge with two vertices of the same material will contain a virtual cut-point. A virtual interface point is always snapped to and therefore co-located with an interface point of an incident simplex with lower dimensionality. In the example of an edge with a virtual cut-point, this virtual cut-point would be snapped to one of the edge's corner-vertices. The choice of interface point to snap to is sometimes arbitrary and will be discussed in detail in the following sections.

4.5.1.3 Virtual Interfaces

This virtuality of an interface point, as introduced the mesh cleaving algorithm of [10], can be understood in two ways:

1. The interface point is virtual because is it is snapped to, meaning co-located with another interface point. It does not need to be represented by its own vertex in a mesh data structure, e.g. a virtual triple-point is snapped to a cut-point and therefore always co-located with it. Whenever the cut-point moves, the triple-point is automatically moved there as well.

2. The interface point is virtual because not enough materials meet there to qualify for such an interface point - e.g., only one or two materials meet on a face, so the triple-point of that face can be considered virtual.

Since an interface point may satisfy either both or only the first condition, but not the second, there are two distinct types of virtual points. The former is called pure-virtual and the latter is dubbed semi-virtual, due to it not satisfying all

virtuality criteria. This distinction is another novelty of this implementation. An example of this would be a triple-point on a face where three materials meet, but the triple-point has been snapped to a cut-point on an edge incident to the face.

Having a non-virtual or virtual interface point for every simplex of a background mesh tetrahedron enables the generalized stencil to work in all mentioned topological cases. The before mentioned edge collapses in the generalized stencil now simply follow from virtual interface points being co-located with interface points of simplices of lower order. Of the 24 mini tetrahedra composing the most complex case, for simpler cases, only those mini tetrahedra are created which have four unique points or, in other words, not a single collapsed stencil edge.

A further consideration which must be made is that all topological cases, except for the 0-cut, contain at least one quadrilateral face patch. The edge collapses in the stencil enable the tessellation of the other topological cases, but they do not directly guarantee consistency across the faces of neighboring background mesh tetrahedra. Mesh consistency between different tetrahedra must be considered, especially when there are quadrilateral patches in the corresponding topological cases.

Notably, the only interface points which can be the cause of inconsistent tessellations are quad-points. This is explained by the fact that vertices, cut-points, and triple-points are geometric entities which are shared between different background mesh tetrahedra. Therefore, if the triple-point on a face of a tetrahedron is moved or snapped by said tetrahedron, it is the same for the tetrahedron on the other side of the face. This is the case, regardless of whether the corresponding geometric entity is stored once for the entire mesh, or individually for each of the tetrahedra it is incident to. In other words, cut-points and triple-points are not solely controlled by a single tetrahedron, but by all tetrahedra incident to the entity. Vertices, cuts, and triples alone would guarantee mesh consistency, since any change is always reflected in the other incident tetrahedra. Quad-points, however, are controlled only by the tetrahedron to which they belong. Furthermore, due to the design of the generalized stencil, a quad-point is included in every created mini tetrahedron. A virtual quad-point is then to be snapped to an interface point of lower hierarchy. The virtual quad-point has to be geometrically located on either a face, an edge, or a corner-vertex. Those are geometric entities potentially shared with other tetrahedra, but changes in the location of the quad-point are not reflected or noticed in any way by the neighboring tetrahedra. In the original mesh cleaving paper various geometric properties and their proofs are provided [10]. It is however not proven that the mesh cleaving algorithm cannot produce invalid tessellations.

4.5.1.4 Violations

A problem that the *isosurface stuffing* [14] and *mesh cleaving* [10] algorithms try to solve is the output of low quality tetrahedra, when conforming to a given interface. This problem, and their potential solutions, are discussed in the following. Consider only two-dimensional (2D) space for this example and the angles of triangles to be the selected quality measure. In this example, a material interface runs across a triangular face element. This cuts the face into two material regions and needs to be tessellated. When this tessellation is to be kept local to the element, only the corner-vertices and the cut-points can be used and no additional vertices are to be added to the triangle and its interior. Assume that this material interface now runs close by a corner-vertex, meaning that one of the interface's cut-points lies close to a corner-vertex on the same edge. This creates the problem that the tessellation of the element will always include a triangle containing a small angle. Because this results in bad output triangles, the cut-point is said to *violate* the corner-vertex. In order to resolve this violation, the mentioned algorithms move the corner-vertex to the violating cut-point and, therefore, onto the interface itself. This deforms the element, but avoids (or at least limits) the creation of elements with small angles, when conforming the entire background mesh to the interfaces.

The presented example is only one case of violation for the mesh cleaving algorithm and its generalized stencil. In total, there are three types of violation: corner-vertex violations, edge violations and face violations. The threshold for the proximity of when an interface violates a simplex depends on a threshold-parameter. This threshold-parameter is often denoted with α and understood as a fraction of an edge length in these types of algorithms. The violation-zones as defined in [10], in which an interface point is considered to be violating a given entity, are depicted in Fig. 4.14.

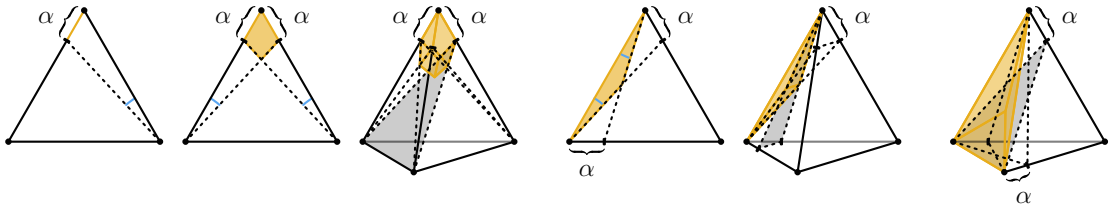


Figure 4.14: The different types of violations are vertex violations (a, b, c), edge violations (d, e), and face violations (f). The violation zone of the interface point is shown in orange. A few of the angles which potentially become small when ignoring the violations are shown in blue. *Images are adapted from* [10].

While a single theoretical parameter α explains the concept of the violations, for real-world applications one should think about the fact that there usually are different edge lengths occurring in the background mesh's tetrahedra. The different edge lengths can be subjected to different α values. Since the tetrahedra of a body centered cubic (BCC) background mesh features contain only two different edge lengths, it is useful to introduce two parameters [14]: α_{short} and α_{long}. In literature, different values for the two α-parameters can be found, as they are specific to the details of the used algorithm. In [14], an extensive table of values found by experimentation on the specific algorithm presented in [14] is provided, together with the corresponding dihedral angle bounds. The concept of the two α-parameters for the BCC based background mesh was also adopted by Bronson, Levine, and Whitaker [10] for the lattice cleaving algorithm, where the exact values were changed to accommodate for different conditions in the multi-material case.

Using a given set of different α-parameters for different edge lengths is a viable option, if the background mesh is based on a lattice consisting of a small number of different edge lengths. Optionally, when using a lattice consisting of a larger number of different edge lengths, or when using an unstructured background mesh, violation thresholds need to be generalized. Such a generalized violation threshold can be a fixed global α-value, such as in the isosurface stuffing implementation *Quartet* [I], which uses an A15 lattice [38]. The violation threshold may also be computed for each vertex, such that no flat, coplanar tetrahedra can occur, as shown for the unstructured background mesh in [39]. Here, only a single α threshold is used, since the presented implementation allows for interfaces to run through bridging tetrahedra, which do not have the same two edge lengths as the standard BCC tetrahedra.

4.5.2 Background Mesh Input

The input to the mesh cleaving algorithm must be conformal and store the level set function (LSF)-value at each vertex of the mesh. However, for any vertex, not all materials' LSF-values need to be stored. It is sufficient to store only the LSF-values of enough materials to make it possible to deduce which material is assigned to the vertex and to deduce the interfaces to other materials along edges, faces, and tetrahedra incident to said vertex.

4.5.3 Finding Interfaces

The first step of the cleaving algorithm is to find all of the interface points in the background mesh by linear approximation. At each edge, there may be multiple material interfaces, although not all of them can be represented in the final mesh. In order to find material interfaces, each material m is considered, and checked

whether there is a zero-crossing along the considered edge. If a zero-crossing is found, the material has an interface along the edge, either towards the next higher material or towards the outside void phase. Since the LS wrapping approach is used, there cannot be an interface to a lower material. The location of an interface of an individual material m on the edge defined by $\mathbf{v}_A$ and $\mathbf{v}_B$ is computed using

$$t_m = \frac{\phi_m(\mathbf{v_A})}{\phi_m(\mathbf{v_A}) - \phi_m(\mathbf{v_B})}. \tag{4.1}$$

In order to find interface points, the material assigned to each corner-vertex is considered. If all four corner-vertices are part of the same material, the tetrahedron may either lie completely inside a material, in which case it does not need to be changed, or it is entirely in the outside void, in which case it is simply removed. This pre-check of the tetrahedra was introduced in this work to increase the performance of the algorithm. In case at least two different materials are found in a tetrahedron, all the interface points, be they non-virtual or virtual, are set for this tetrahedron. The setting of the interface points takes place in hierarchical order, upwards from cut-points to triple-points, to the quad-point, explained in the upcoming Section 4.5.3.1, Section 4.5.3.2, and Section 4.5.3.3, respectively.

4.5.3.1 Finding Cut-Points

In case the materials assigned to the edge's vertices are the same, the edge's cut-point must be virtual. The choice of corner-vertex on which to create the virtual cut-point, is arbitrary. A virtual cut-point will, for simplicity of the rule, always be created on the corner-vertex with the lower vertex ID in the mesh data structure.

With two different materials m_A, m_B on the corner-vertices $\mathbf{v}_A$, $\mathbf{v}_B$, respectively, the edge's cut-point is non-virtually generated according to Eq. (4.1). If, however, there are several material interfaces cutting a single edge, only two materials can be represented, as there may only be one cut point which must be approximated. In this work, these approximations are performed based on the inside-outside-interfaces which lie on the edge in question. The largest material ID can never have an inside-outside-interface on the edge, when using the LS wrapping approach. Under the assumption of the lowest material being m_A and the highest m_B, and $m_A + 1 < m_B$, meaning at least three materials are represented, the edge contains the inside-outside-interfaces of the materials $m_A, \ldots, m_B - 1$. Four different approximations to the cut-point location were implemented in this work, as depicted in Fig. 4.15:

1. Average all material interfaces on the edge. This corresponds to averaging over all of the inside-outside-interfaces

$$s_{\text{avg. all}} = \frac{1}{m_B - m_A} \sum_{i=m_A}^{m_B-1} t_i. \tag{4.2}$$

2. Average only the interfaces to the materials m_A and m_B of the corner-vertices. This corresponds to

$$s_{\text{avg. } AB} = \frac{t_{m_A} + t_{m_B-1}}{2}. \tag{4.3}$$

3. Set the cut-point to the interface of material m_A. This corresponds to

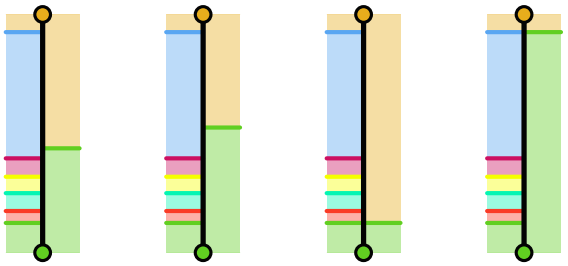
$$s_{\text{only } A} = t_{m_A}. \tag{4.4}$$

4. Set the cut-point to and interface towards material m_B. This corresponds to

$$s_{\text{only } B} = t_{m_B-1}. \tag{4.5}$$

The location of the cut-point is given as a factor s of the edge length, as seen from vertex $\mathbf{v}_A$, and t_m is the linear interpolation of Eq. (4.1). The actual cut-point coordinate is given by:

$$\mathbf{c} = \mathbf{v}_A + s(\mathbf{v}_B - \mathbf{v}_A). \tag{4.6}$$



(a) $s_{\text{avg. all}}$ (b) $s_{\text{avg. } AB}$ (c) $s_{\text{only } A}$ (d) $s_{\text{only } B}$

Figure 4.15: The four different implemented approximations to a cut point in case there are multiple material interfaces running through a single edge. On the left side of the edge all materials running through the edge are shown in different colors while on the right side the cut-point's location is shown.

Note that these presented forms of cut-point approximations are a direct consequence of the difference in LS definitions used between the original mesh cleaving [10] and this work, since the original mesh cleaving algorithm does not employ the LS wrapping approach.

4.5.3.2 Finding Triple-Points

As with the cut-points, first the number of unique materials assigned to the corner-vertices of a face are checked. In case there are less than three materials, the triple-point is created as a virtual triple-point. The selection of the cut-point, to which the virtual triple-point should be assigned, is made based on the following two rules. First, a non-virtual cut-point is preferred over a virtual one. Second, in case there are multiple options, the cut-point which lies on the edge with the lowest edge ID is selected. While the second rule is arbitrary and serves only to resolve ambiguities in the selection process, the first rule is important for tessellation.

For simplicity, we can consider only a 2D face and not the entire tetrahedron to which it belongs. Due to the design of the generalized stencil, the triple-point is part of every tessellation of a face. Hence, the triple-point must lie somewhere on a material interface in order to make it possible for the interface to be tessellated. This is not the case, when a virtual triple-point is assigned to a virtual cut-point, even though the face contains non-virtual cuts as well. Assigning a virtual triple-point to a virtual cut-point, would also mean that the virtual triple-point is consequently assigned to a corner-vertex.

When the face has three different materials, it receives either a non-virtual triple-point, with its own vertex in the mesh, or a semi-virtual triple-point, snapped to one of the three non-virtual cut-points. The type of triple-point is determined by the geometric approximation of the triple-point's location. If the approximated triple-point location is within the face, it becomes a non-virtual triple-point. If the location lies outside the face, the triple-point is snapped to the nearest cut-point, making it a semi-virtual triple-point. Note that all cut-points on such a face are non-virtual cut-points, as otherwise the face would have simply received a pure-virtual triple-point. Such snapping of a triple-point, during its creation, is another case of the algorithm removing thin features. In this case, small triangular shaped segments are removed.

For the LS wrapping approach used in this work, most interestingly the position of a triple-point on faces with three unique materials is predetermined and does not have to be computed. This is the case because the combination of the used LS definition and the mesh cleaving in its current form, does not allow for non-virtual triple-points. Any face which has three materials on it automatically receives a semi virtual triple-point assigned to the cut on the edge between the corner-vertices with the lowest and highest material ID. A similar observation is made for quad-points (see Section 4.5.3.3). A further explanation of this observation follows in Section 4.5.3.4.

4.5.3.3 Finding Quad-Points

Similar to all other interfaces, finding a quad-point for a given tetrahedron starts with a check of whether the are four distinct materials to make a non-virtual quad-point possible. In case there are less than four unique materials assigned to the corner-vertices, the quad-point must be pure-virtual. The selection of the interface point, to which to assign the virtual quad-point, follows similar rules as the selection of a pure-virtual triple-point. Following the hierarchical structure of the interface points, the virtual quad should be assigned to a triple-point. Once again, non-virtual interface points are preferred over virtual ones, in order to end up with a tessellation which is not only valid, but which approximates the interface given through the LS input.

For tetrahedra with four unique materials, the quad-point must be either non-virtual or semi-virtual. Following the same ideas as for the triple-point, a non-virtual quad-point would only become a semi-virtual quad-point during its creation, in case the calculated quad-point location was outside of the tetrahedron. However, as previously mentioned, with the used LS definition and its wrapping approach, only semi-virtual quad-points are possible. Such a semi-virtual quad-point always lies on the non-virtual cut-point on the edge between the tetrahedron's vertices with the smallest and largest material ID. This cut hosts also two semi-virtual triple-points, so it is automatically consistent with the interface hierarchy. This observation is explained in detail in Section 4.5.3.4.

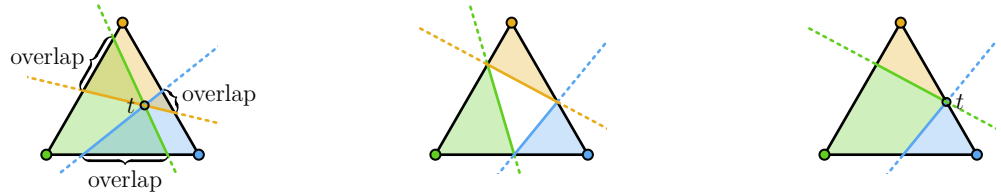
4.5.3.4 Level Set Wrapping Approach Implications

As mentioned earlier, it is important to note that using the LS wrapping approach, there cannot be any non-virtual triple- or quad-points. As a guide to understanding why this is the case, consider the problem of finding a non-virtual triple-point. Each of the face's corner-vertices has a unique material assigned to it. For each of the three materials, one has a LSF-value on every corner. Therefore, one can perform a linear interpolation of a material's LSF-values. The linear interpolation over the triangle is linear on each of the edges and thereby also the zero crossings of the interpolation function coincides with the inside-outside-interfaces of said material on each of the edges. The line between two such points is considered to be the linearized approximation to the inside-outside-interface. The problem of finding the position of the triple-point, therefore, reduces to looking at a constellation of the linearized inside-outside-interfaces - or in simple geometric terms, the intersection of three lines in 2D.

Three lines can either intersect at a single point or build an intersection triangle.

Consider the case when the lines intersect at a single point, which is the preferred case, as it leads to a unique triple-point location. In that case the different

material domains overlap on the edges, as depicted in Fig. 4.16a. As the LS is defined in this work to consist of non-overlapping material regions, this can never be the case. Non-overlapping material domains lead to a void region at the center, as depicted in Fig. 4.16b. To avoid such voids between materials the LS wrapping approach was used in the first place. With LS wrapping, however, the outermost material, the one with the largest material id, cannot have an inside-outside-interface within the triangle. This reduces the problem to an intersection of two lines instead of three lines, which is shown in Fig. 4.16c. Using the LS wrapping approach, an outer material completely has to wrap its inner materials, meaning there cannot be any crossings of linearized material interfaces within the triangle. Because of this, the triple-point location is always located on an edge, hence it is a semi-virtual triple-point.



(a) Using overlapping material domains.

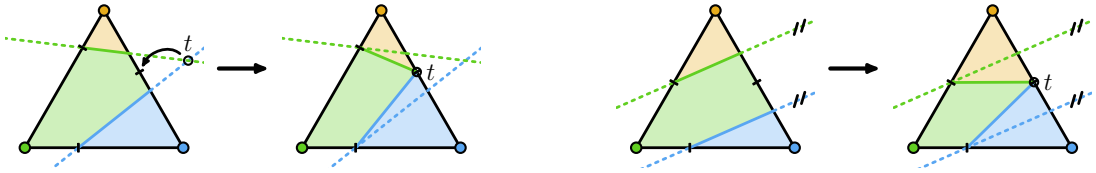
(b) Using non-overlapping material domains.

(c) Using the LS wrapping approach.

Figure 4.16: Material configurations for different LS definitions and how the LS wrapping approach can only lead to virtual triple-points.

Next, consider the imperfect case, where the middle material has a layer of finite thickness around the innermost material. In such a case, the three materials do not touch within the triangle based on the LSF-values, but the mesh cleaving algorithm removes such a thin feature by design. The intersection point of the two lines in the imperfect case would lie outside of the triangle and could simply be snapped as a semi-virtual triple-point to the nearest cut-point, removing the thin layer around the innermost material, as depicted in Fig. 4.17a. All these considerations finally lead to the observation already mentioned in Section 4.5.3.2, that a triple-point of a face with three distinct materials is always semi-virtual, and always lies on the cut-point on the edge between smallest and largest material ID. This quick and simple rule for setting the triple-point even takes care of the special case depicted in Fig. 4.17b, where the two interfaces are parallel to each other within the face. This would otherwise not allow for the calculation of an intersection point. The examples in Fig. 4.17 also show that in the imperfect case, the sandwiched material's region on the face gets smaller area wise, and that the innermost and outermost materials potentially gain in area on the face. The amount of region gained between the innermost and outermost material is

dependent on the geometric constellation and the type of approximation used to find the edge's cut-point, as explained in Section 4.5.3.1.



(a) Interface intersection outside of the triangle.

(b) Parallel interfaces.

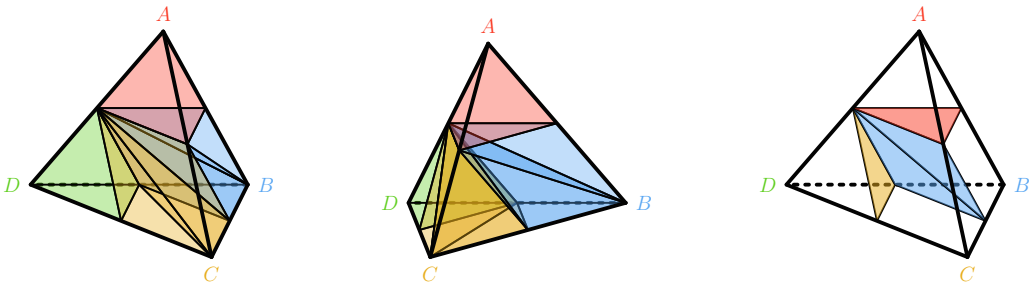
Figure 4.17: Cases of thin feature removal, when setting the triple-point using the LS wrapping approach.

For quad-points, the same considerations lead to analogous results. There are no non-virtual quad-points employing the LS definitions applied for this work and the LS wrapping approach. Any quad-point created on a tetrahedron with four distinct materials at its corner-vertices, will receive a semi-virtual quad-point lying on the cut-point on the edge between smallest and largest material ID. On the same cut-point both semi-virtual triple-points of the faces incident to the edge are placed.

4.5.3.5 Reduced Generalized Stencil

When applying the above-discussed implications of the LS wrapping approach to the mesh cleaving's generalized stencil, one arrives at a reduced generalized stencil. This reduced generalized stencil is depicted in Fig. 4.18. It consists of only 8 mini tetrahedra compared to the 24 of the original mesh cleaving stencil. The mini tetrahedra are also not equally distributed across the material regions. Assume material IDs $A < B < C < D$, which is also the case in Fig. 4.18. The materials with the smallest and the largest ID A and D, respectively, are each only represented by a single mini tetrahedron, while the two middle materials, B and C, are each assigned three mini tetrahedra. Similarly, the interfaces A to B and C to D consist of a single triangular patch, while the interface B to C consists of two connected triangular patches. This yields a total of 4 interface segments in the reduced generalized stencil, compared to the 12 interface segments of the normal generalized stencil.

This work's implementation still uses the normal generalized stencil, as the standard mesh cleaving operations work for the LS wrapping approach input as well. The reduced generalized stencil could however be exploited in order to increase the performance in future implementations - see Chapter 6.



(a) Mini tetrahedra.

(b) Mini tetrahedra shown in a rotated view.

(c) Material interfaces between the mini tetrahedra of the reduced generalized stencil.

Figure 4.18: The mesh cleaving's reduced generalized stencil specialized to the LS wrapping approach. The four different materials A, B, C, and D are shown in red, blue, orange, and green, respectively. The materials are wrapped in the given order, with D being the outermost material.

4.5.4 Resolving Violations

Once all of the interface points, virtual or not, of all tetrahedra with more than one material assigned to its corner-vertices, have been set, it needs to be checked whether any interface violates a geometric entity. The violations are checked and resolved depending on their type. First, violations of vertices, then edges, and finally faces are checked.

Resolving violated vertices in the first step has the advantage that these are the only types of violation which cause vertices to change location, which is called a *warp*.

All other violations are solved by simply *snapping* the corresponding interface point to another interface point of lower hierarchical order, but without changing the position of the interface point which is being snapped to. The interface point which is snapped, shall be labeled as a *snap-source* and the interface point which is snapped to shall be labeled as the *snap-target*.

Some of the snaps cause degenerate configurations of interfaces, which are discussed in Section 4.5.4.8. Those interface degeneracies are also dealt with by snapping other non-violating interface points appropriately. The non-violating interface points which are snapped in such a case are said to be *pulled into* the corresponding vertex. Therefore a *pull* is simply a special case of a *snap*.

When an interface point is snapped or pulled, all points hosted by said interface point need to *follow* the same snap. For example, a virtual triple-point which lies

on a cut-point, which was snapped to a corner-vertex, needs to follow its cut-point and also be snapped to the same corner-vertex.

4.5.4.1 Resolving Vertex-Violations

When a corner-vertex is violated, all interface points violating the corner-vertex are snapped to said corner-vertex and the corner-vertex is subsequently warped to a new location approximating the interface.

All non-violating cut-points which represent any of the material interfaces snapped to in the previous step need to be pulled into the corner-vertex, in order to resolve the interface degeneracies created by the snaps and the warp. Similarly, all triple-points on incident faces which represent an interface to two snapped materials need to be puled into the corner-vertex as well. For quad-points, the same needs to be performed in case all three of the other materials were snapped to the corner-vertex.

The new location to which the corner-vertex is warped to, is intended to be on the interface by the idea of the snaps and warps, but in some cases approximations must be made. In the simplest case, there is only one material, and a vertex violated by several cut-points can be moved to the closest cut-point [14]. This would form the closest approximation to the explicit interface.

In the multi-material case of the mesh cleaving algorithm, the selection of a location to which to warp the violated corner-vertex, is not so straight-forward. Several different interfaces can violate the same vertex. One solution would be to warp the corner-vertex to the center of mass of all violations (cuts, triple-points, and quad-points) [10]. Alternatively, higher order violations could be given priority, such that the corner-vertex is moved to a violating quad-point first, if only one exists, a violating triple-point if only one exists, or if none of the previous was possible, to the center of mass of all violations [II].

One advantage of the single material case is that independent of which violating vertex was selected, the corner-vertex always lies on the interface itself after the warp and an LSF-value of zero can directly be assigned to the only existing material. This is not so straight-forward when averaging all violations (taking their center of mass), as this is only a further approximation to the actual material interface. Even in the approach of the *Cleaver* implementation [II], where a single quad-point or triple-point might be selected as the new location, it would only be on the interface for four or three materials, respectively, compared to the potentially as many materials as there are adjacent to the warped vertex. In order to place the post-warp LSF-values of all materials correctly on the warped vertex, one would need to recalculate all the LSF-values for the new location. Although the LSF values of the warped vertex could be updated for increased accuracy, the benefit is quite insignificant, when compared to the computational effort required,

since the warping distances are short, so the recalculation of LSF-values may be skipped.

The multi-materiality of the mesh cleaving algorithm [10] introduces another complexity not otherwise present with the isosurface stuffing algorithm [14]. When considering only a single material, any non-violating cut-point being on an edge, incident to a warped vertex, will get pulled into the warp to resolve degeneracies, as mentioned above. With multiple materials, only those non-violating cut-points get pulled into the warp, which represent an interface to materials which was snapped to the warped vertex. Any non-virtual cut-point present after the snaps and pulls, still lies on an edge which changed due to the warped vertex. This is not only the case for cut-points, but also for the other interface points. Since the warp affects all incident simplices, the positions of all non-virtual interfaces, which are not pulled into the warped corner-vertex, need to be recalculated. Those interface points recalculations were introduced in the mesh cleaving algorithm [10], where they were named *projections*.

Virtual interface points do not need to be projected, as they are hosted by another interface point and do not have their own vertex in the mesh. However, in case the hosting interface point is snapped to another mesh-vertex, be that a corner-vertex or another interface point, the virtual interface point needs to follow. Therefore, any virtual triple whose cut-point is snapped during the resolution of a violation needs to follow its cut-point and, therefore, be snapped to the same mesh-vertex to which its cut-point was snapped. Similarly, any virtual quad-point whose triple-point was snapped, needs to follow its triple-point and be snapped in the same way. In this ordering, given through the hierarchy of the interface points, logically triples are checked before quads.

Because interfaces receive a new position, when they are projected, they can potentially be violating, based on their new position after the projection. When an interface point is violating a corner-vertex post projection, it can be one of three cases: it violates a corner-vertex which has already been warped, it violates a corner-vertex which has not been checked, it violates a corner-vertex which has been checked, but was not warped. When the vertex in violation has already been warped, the violating interface is simply snapped to the vertex it violates. When the violated vertex has not been checked, the violation will be resolved when the corresponding vertex is checked for violations. In case the violated corner-vertex has been checked already, it needs to be re-checked, otherwise, the violation will remain, resulting in low quality output tetrahedra. To manage this re-checking, a queue is introduced to which a corner-vertex is added in case it has already been checked, but is violated after some warp step. The vertices are therefore checked for violations in two main iterations. In the first iteration, every corner-vertex is visited once. When a corner-vertex is violated post warp, it is added to the queue

in case it has already been checked - if its ID is smaller than the ID of the vertex currently being warped. In the second main iteration, the queue is iterated over until no vertices are left in the queue. Any vertex from the queue is warped during its re-check, as the vertex was placed into the queue for being violated. This can potentially add new vertices to the queue. However, this time, as all vertices have been checked once already, any post warp violated vertices which have not been warped are added to the queue. Since all vertices from the queue are warped once they are re-checked, any vertex in the mesh is checked for violations through those two main iterations at most twice.

4.5.4.2 Projection of Interfaces

The *projection* of a non-virtual interface point means the recalculation of its position after changes to the simplex it resides on. In theory, this would simply mean doing the same calculations as for the interface point's initial position (see Sections 4.5.3.1 to 4.5.3.3). However, the fact that the warped vertices' LSF-values are not updated, and therefore do not correspond to their current positions but rather to their initial positions, prohibits recalculations in the same way as the initial calculation.

In contrast to the calculation of the initial position of an interface point, its post-warp position is approximated using the approximations to the interfaces rather than the corner-vertices' LSF-values. The material interfaces are surfaces approximated as a piecewise linear - connected patches of triangles and quadrilaterals. These are subsequently used to calculate points of intersection with, e.g., the post-warp location of an edge whose cut-point needs to be projected.

These approximations of the interfaces originally occur with the initial setting of the interface points in Section 4.5.3. With each warp, and with each snap, these approximations to the interfaces are modified. When using the interface approximation to project interface points, a decision should be made whether it is necessary to use the approximation in its current state, with all snaps and warps which happened up to the current point in the execution - as it is done in the *Cleaver* implementation [II]. Alternatively, it is possible to use the initial approximation, generated when finding the interfaces (Section 4.5.3). In this work, the latter is used, since it includes one fewer approximation to the interface being modeled. For example, consider the snapping of multiple violating cut-points to a single corner-vertex and the corner-vertex subsequently gets warped to the center of mass of the violations; then, the corner-vertex only approximately lies on all those different interfaces, as it was not snapped to a single violating cut-point, but rather to the center of mass. These snaps and warps are performed in order to create good quality output tetrahedra, but they do not add any benefit to the approximation of the interfaces.

Because the calculation of the initial position of an interface point does only rely on the LSF-values of the corner-vertices incident to the simplex, any tetrahedron incident to the simplex can be used to perform the calculation. For example, it does not matter which of the two tetrahedra sharing a face calculates the face's triple-point's location, as it is based on the same three corner-vertices' LSF-values. Note that when using the LS wrapping approach, there actually aren't any non-virtual triple-points, as explained in Section 4.5.3.4, but they serve as an easy way to understand the example.

When using intersections with the interface approximations, it does matter which incident tetrahedron is used to perform the calculation. As an example, once again consider a triple-point and its projection. The triple-point lies on the ray from the quad-point through the triple-point's initial position, as this is the line along which the three adjacent materials meet. Therefore the projection of the triple-point lies at the intersection of the new post-warp face and the quad-triple-ray. Each of the two incident tetrahedra has its own quad-triple-ray which point in different directions. Unless the warped vertex again lies on the pre-warp face, the two incident tetrahedra provide two different points of intersection, based on their two different rays. In order to resolve this ambiguity of intersection points, it appears reasonable to only select one of the two tetrahedra, to perform the triple-point projection. As interpolation is better than extrapolation, the tetrahedron whose intersection point lies within its pre-warp state, is selected.

Generally speaking, there are different possibilities of finding an appropriate tetrahedron on which to base the calculation of the intersection. For this work, a simple solution was devised using a metric which is going to be called *sum of edge length changes*. A tetrahedron's *sum of edge length changes* for a given warp is, as the name suggests, the sum of all changes in edge lengths in the given tetrahedron caused by the warp. The tetrahedron with the smallest sum of edge length changes, is the tetrahedron that which reduced the most in volume through the warp. For the selection of a tetrahedron for a given interface point projection, all tetrahedra incident to the simplex on which the interface point is located, are considered. Of all considered tetrahedra, the one with the smallest value for the sum of edge length changes is selected. Subsequently, the actual interface point projection is calculated using the selected tetrahedron. In case the interface point projection fails, the tetrahedron with the next bigger sum of edge length changes is sampled and so on.

The lattice cleaving paper [10] does not go into detail about the different projections and their order. Comparing the approach of this work to the source code of the *Cleaver* implementation [II], the ideas and concepts are similar, while the ordering is reversed. This means that the *Cleaver* implementation [II] projects quad-points first, then triples, and finally cut-points. In this work the cut-points

are projected first, then triples, and finally quads - when considering the implication of the LS wrapping approach (Section 4.5.3.4) only cut-points need to be projected. As the different types of projections can depend on the results of previous projections, there is no simple direct comparison between the two approaches for a given projection type.

The details of the different types of projections are given in the upcoming Sections 4.5.4.3 to 4.5.4.5 in the order in which they are dealt with in this work's implementation.

4.5.4.3 Projection of Cut-Points

The first projections which are calculated after a corner-vertex is warped are the cut-point projections on edges incident to the warped vertex. A cut-point always lies on the intersection of an edge and a material interface. In the mesh cleaving's interface approximation, interfaces between different materials are always represented by a triangular surface patch between a cut, a triple, and the quad-point of a given tetrahedron. This means that a new post-warp edge in a cut-point's projection can intersect with up to three, in a special case four, interfaces within a single tetrahedron. The special case arises when the new edge lies exactly on the edge between the two patches of the same material interface.

In case the new edge intersects with multiple interface patches, the projection point is not uniquely given. This happens especially when a cut-point which hosts a semi-virtual triple-point or a semi virtual quad-point must be projected. In this case of a non-unique intersection, a further approximation which results in a thin feature removal, must be made. While there are other approaches, in this work, the problem is resolved by averaging all found intersections between the new edge and the interface patches.

The complete process of projecting a cut-point thus consists of calculating the sum of edge length changes for all tetrahedra incident to the edge of the to-be-projected cut-point, and using the tetrahedron with the smallest sum of edge length changes, to calculate a projection position from the intersections. If the projection position's calculation fails in the selected tetrahedron, the tetrahedron with the next larger sum of edge length changes is attempted. If it fails again, the one with the next largest sum of edge length changes is attempted - and so on. Once a projection position has successfully been calculated, the results are checked for violations. This is done by examining the calculated position on the edge as a factor of the edge's length, as seen from the warped vertex. There are different cases to be considered, based on the edge factor λ.

The first case is when the cut-point's proposed projection position violates the warped vertex, $0 < \lambda < \alpha$, or lies outside of the edge's line segment with a negative factor $\lambda < 0$. The cut-point is therefore snapped to the warped vertex. The

material interface represented by the cut-point then has to be considered for pulls of other cut-points. This also holds true for cut-points which have already been projected and were not snapped. Therefore, all cut-points previously projected during this warp which remained non-virtual, need to be rechecked.

In the second case, the new position does not violate any of the edge's corner-vertices, $\alpha \leq \lambda \leq 1 - \alpha$. The cut-point is simply projected to the new position.

Similarly to the first case, in the third case, the edge's other corner-vertex is violated by the cut-point's proposed projection position, $1 - \alpha < \lambda \leq 1$. In case the violated vertex has been warped already, the cut-point is snapped to the edge's other corner-vertex. With the same consequences as for any snap to an already warped vertex, potential pulls and degeneracies caused by the snap need to be checked. If the other vertex has not yet been warped, the cut-point is projected to the new location. Depending on whether it is the first or second loop of resolving the vertex violations, the other vertex is added to the queue, assuming it wasn't already checked or added, respectively.

Depending on the approach used for the cut-point projection calculation, in theory, there may exist a need for a fourth case, where the proposed projection position lies beyond the edge's line-segment with $\lambda < 0$ or $\lambda > 1$. This case does not arise with the approach to cut-point projection calculation used in this work's implementation. As it was considered during development and for the sake of completeness, notes on the implications of this forth case are provided in Appendix B.1.

4.5.4.4 Projection of Triple-Points

The warp of a vertex also changes the incident faces. Virtual, semi-virtual, and pure-virtual triple-points simply follow the cut-point they are co-located with, but non-virtual triples need to be projected.

Through the use of the LS wrapping approach in this work, however, there is no need for a triple-point projection, as there are no non-virtual triple-points (compare Section 4.5.3.4). An approach to triple-point projection in the general case, disregarding the simplification accorded through the LS wrapping approach is described in Appendix B.2.

4.5.4.5 Projection of Quad-Points

Quad-points do not lie on intersections of material interfaces with geometric entities, like cuts and triples do, but rather they lie where four material interfaces meet. As the meeting point of the modeled interfaces does not change, when the geometric entities move, a quad-point should not move either. Therefore, the only check required of a quad-point, when one of the tetrahedron's vertices is warped,

is whether the quad-point ends up outside of the post-warp tetrahedron. The projection of a quad-point is still considered a projection, as the quad-point would need to be projected back into, or rather onto, the post-warp tetrahedron in this case.

As previously mentioned, calculating such a projection is not necessary when using the LS wrapping approach, nor in the general case. For the LS wrapping approach, the reason is that there are no non-virtual quad-points (compare Section 4.5.3.4). The general case is discussed in Appendix B.3.

4.5.4.6 Resolving Edge-Violations

Edges can, as discussed in Section 4.5.1.4, only be violated by triple points and quad points. Since those need to be non virtual to cause a violation, they are not considered, when using the LS wrapping approach. For an explanation on how the general case could be handled, see Appendix C.1.

4.5.4.7 Resolving Face-Violations

Similarly to resolving violated edges in Section 4.5.4.6, the violated face only need to be resolved in case there are indeed non-virtual quad points. Using the LS wrapping approach, this is not the case and so for an explanation of the more general case, refer to Appendix C.2.

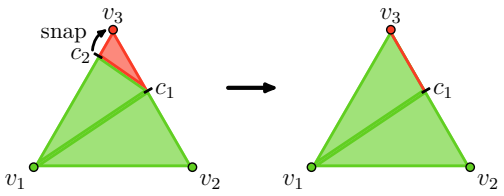
4.5.4.8 Resolving Degeneracies

Snaps, regardless of whether they are caused by a violation or another pull, can lead to degenerate configurations in terms of the interface constellations and degeneration in the interface point hierarchy. Although *violation* is probably a good term, it will be avoided as much as possible in the context of degeneracies, in order to not cause confusion with the interface point violations discussed up to now.

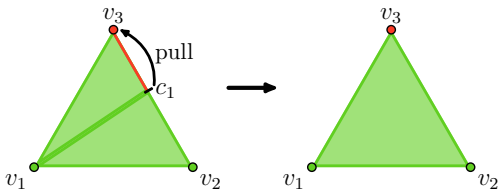
Most degeneracies can be viewed as being based on material constellations, caused by snaps of interface points to corner-vertices. They are resolved using pulls, according to on the corner-vertex's *snapped materials*. A snapped material is a material lying on the other side of an interface point which snapped to a corner-vertex. This is best explained using a cut-point. A cut-point represents the interface between the two different materials of an edge's corner-vertices. Assume the edge has the corner-vertices v_1 and v_1. In case the cut-point violates the corner-vertex v_1 and is therefore snapped to v_1, then the material assigned to the other corner-vertex v_2 shall be called a *snapped material* in this work. Similarly, when a triple-point is snapped to a corner-vertex, of the three materials meeting at the triple-point, the two materials not belonging to the snapped-to corner-vertex, are

considered to be snapped materials. Analogously, for a quad-point being snapped to a corner-vertex, the other three assigned materials are referred to as snapped materials. In theory, even the material assigned to the corner-vertex to which an interface point was snapped could be considered a snapped material. This does not yield any benefit, however, as there cannot be any non-virtual cut-point on an edge between two vertices having the same material, and therefore considering said material to be a snapped material would not result in any pulls.

In this context, a snap to a corner-vertex can potentially lead to degeneracies if there is an adjacent interface point representing the same interface material, and this interface point was not snapped. This is exemplified in 2D in Fig. 4.19 and Fig. 4.20, for the case of a cut-point and a triple-point snap, respectively. In Fig. 4.19a the cut-point c_2 is snapped to the corner-vertex v_3. With this snap the material assigned to the cut-point edge's other vertex, v_1, is added to v_3's list of snapped materials. The snap of c_2 however also leads to a degeneracy - on the edge between v_2 and v_3 there is a degenerated material interface along the segment v_3 to c_1. To resolve this degeneracy c_1 is pulled into v_3.



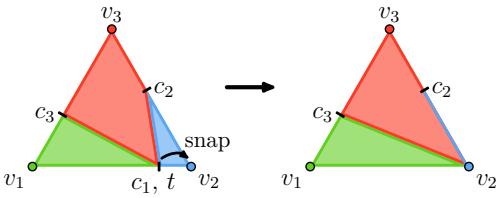
(a) The snapping of the cut-point c_2 causes a snapped material (green) and leads to a degeneracy on the edge between v_2 and v_3 since the existence of the cut point c_1 still indicates an interface. This interface exists only on the edge, which is not allowed.



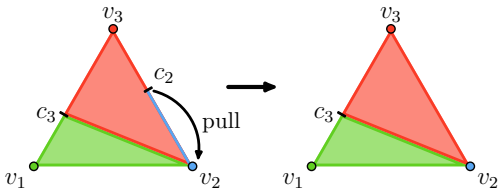
(b) The degeneracy is resolved by pulling the cut-point c_1 into v_3. The pull happens because v_2 is assigned a material which is on v_3's list of snapped materials.

Figure 4.19: 2D example of a degeneracy resolved by a *snapped material* based on a cut-point snap.

In Fig. 4.20a the cut-point c_1 is snapped to the corner-vertex v_2, so v_1's material is added to v_2's list of snapped materials. As the face's triple-point t is semi-virtual on c_1, it follows c_1 and is also snapped to v_2. With this triple-point snap, the materials of both v_1 and v_3 are added to v_2's list of snapped materials - v_1's material is of course not added as it is already on the list. The snap of the triple caused a degenerate configuration, as there is a degenerate interface on the segment v_2 to c_2. This degeneracy is resolved, as shown in Fig. 4.20b, by pulling c_2 into v_2 as c_2's other material, the one of v_3, is on v_2's list of snapped materials.



(a) The snapping of the cut-point c_1 and the triple-point t hosted by it, cause both materials (orange and green) to become snapped materials. The snap leads to a degeneracy on the edge between v_2 and v_3 since the existence of the cut point c_2 still indicates an interface between blue and red. This interface exists only on the edge, which is not allowed.



(b) The degeneracy is resolved by pulling the cut-point c_2 into v_2. The pull happens because v_2 is assigned one of the materials (green) marked as the snapped materials (orange and green).

Figure 4.20: 2D example of a degeneracy resolved by a *snapped material* based on a triple-point snap.

A snap always adds the corresponding materials to the snapped materials of the given corner-vertex, but it does not have to lead to a degeneracy. This is depicted in Fig. 4.21, where then snap of c_3 to v_3 adds v_1's material to v_3's list of snapped materials, but it does not result in any degenerate interface.

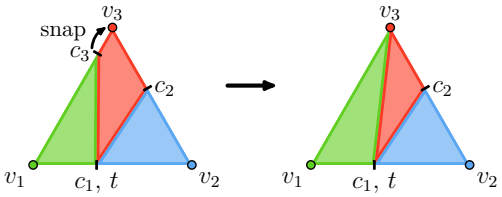


Figure 4.21: Snap not causing a degeneracy. The material is still a *snapped material*.

Even though in both examples, Fig. 4.19 and Fig. 4.20, the degeneracy is a segment along one of the edges, this is not the only possible degeneracy. The other possible degeneracies are related to non-virtual triple and non-virtual quad-points. These do not need to be considered for the LS wrapping approach, and only come into play when using the general implementation; hence, further discussion of the other possible degeneracies can be found in Appendix D.

4.5.4.9 Resolution Cycle

Any snap can lead to a degeneracy. Hence, after each snap, all interface points on simplices incident to the snap-target need to be checked for potential degeneracies. A degeneracy is always resolved through a snap. Therefore, a snap can lead to a degeneracy, which leads to a snap, which can lead to a degeneracy and so on. This snap-degeneracy cycle can cascade through the background mesh. The cycle can, however, only be kept alive by degeneracies involving corner-vertices - this excludes the special case mentioned in Appendix D. Therefore, it can only occur around a single corner-vertex at a time. This means that a single cascade can only run over all tetrahedra incident to the same corner-vertex, which cause the cycle.

Note that the above is a simplified way of looking at the degeneracy for descriptive purposes. Checking for degeneracies does need to be performed directly after each individual snap. For example, when checking a corner-vertex for violations, all interface points violating said corner-vertex are snapped to it without checking for degeneracies after each individual snap. In this work's implementation, the degeneracies are instead examined after the corner-vertex has been warped, while going over the remaining interface points to determine required projections. Any remaining interface point being subject to a degeneracy is directly snapped based on this degeneracy, instead of calculating its projection first.

4.5.5 Output of Tetrahedra

The final step of the mesh cleaving algorithm is the creation of the resulting mesh and its output tetrahedra. This work's implementation reuses and modifies the background mesh. Therefore, the resulting output tetrahedra are created in two ways, depending on whether they lie at the interface or not.

The background mesh tetrahedra, which lie completely inside a single material, meaning that all four corner-vertices have the same material assigned to them, have been treated at this point in the algorithm already. In the part, where the interface points were set (see Section 4.5.3), the said tetrahedra were either deleted in case they belong to the outside void phase, or kept unchanged in case they were fully inside a material.

At this point in the algorithm, only those background mesh tetrahedra need to be considered, which lie on a material interface and therefore were assigned interface points at the beginning. The tetrahedra at the interfaces need to be turned into mini tetrahedra, in order to create an output mesh which approximates the material interfaces. Mini tetrahedra are the tetrahedra which are based on the generalized stencil, as introduced in Section 4.5.1.2. For every such background mesh tetrahedron at an interface, the following steps are executed: All combinations made of a corner-vertex, a cut-point, a triple-point and the quad-point are

examined. All combinations consisting of four distinct points are added as tetrahedra to the resulting mesh. After being checked, the original background mesh tetrahedron is removed from the mesh, as its volume is completely tessellated by mini tetrahedra.

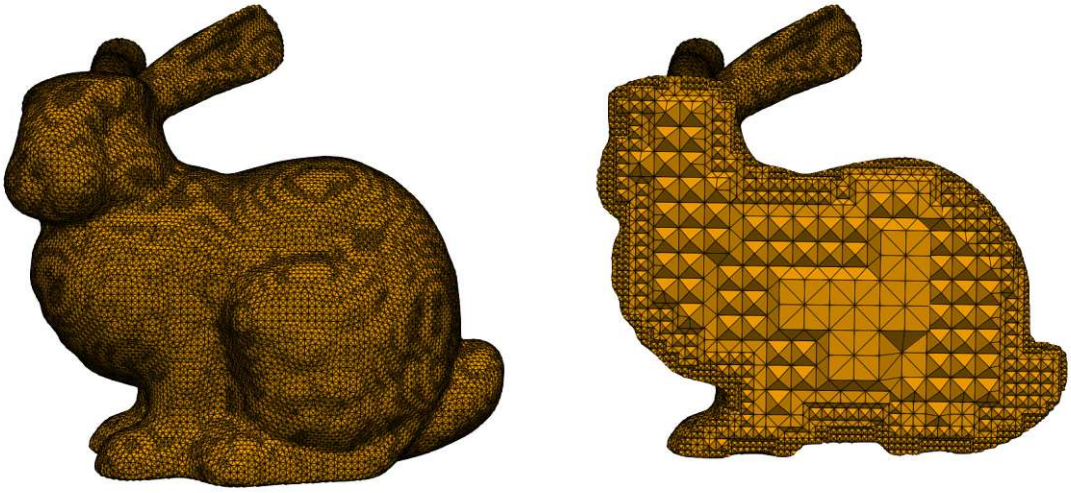
The resulting mesh is thereby completed and the mesh cleaving algorithm is concluded.

Chapter 5

Discussion

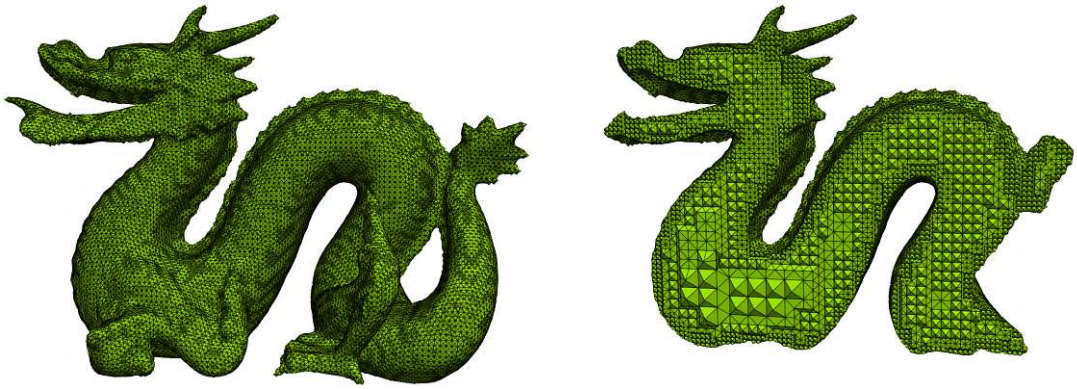
5.1 Generated Meshes

In order to evaluate the implemented meshing algorithm, it was tested on several model inputs. The single material case was developed first, for which a dodecahedron model was built using the level set (LS) Framework *ViennaLS* [III] and comparable in-house tools at Global TCAD Solutions GmbH (GTS). Although there are common meshing test models for single material meshing, such as the *Stanford Bunny* [IV] or the *Stanford Dragon* [V], there are no such standardized models for multi-material meshing. This makes comparisons between different implementations quite difficult. This work is focused on the multi-material case; however, the mentioned classical models were meshed as well. Images of the produced meshes can be seen in Fig. 5.1 and Fig. 5.2, for the Stanford Bunny and the Stanford Dragon respectively. The two single material models were also used to produce additional results presented later on in this chapter.



(a) View from the outside. (b) Cut through the mesh.

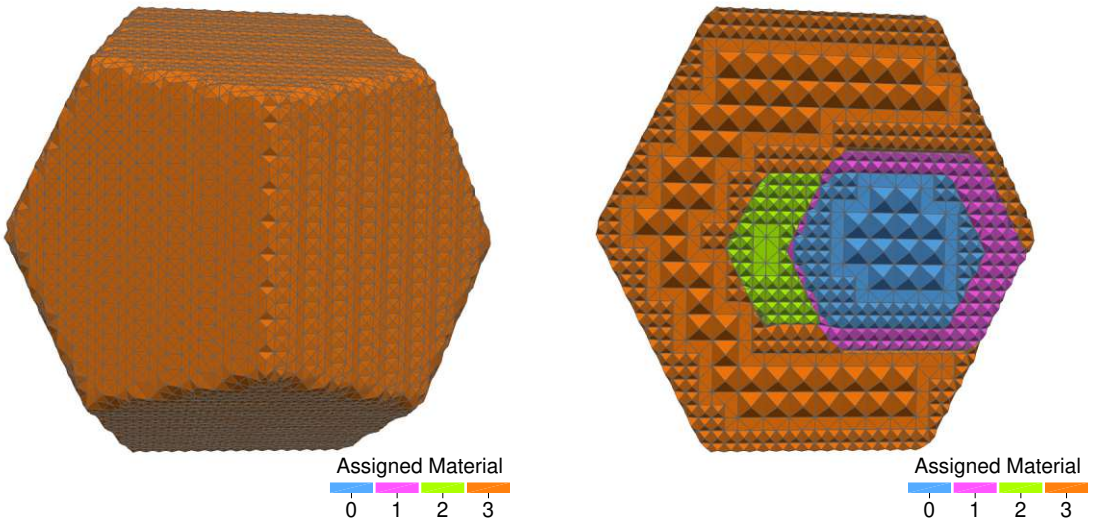
Figure 5.1: Mesh generated for the famous *Stanford Bunny* [IV]; however, the original model was not used, as it contains holes, making it difficult to turn it into a LS. Instead, a watertight version [VI] was used. The input LS data has a grid delta of $\Delta g = 2.0$ and the violation threshold in the meshing algorithm was set to $\alpha = 0.225$. The produced mesh consists of 447 558 tetrahedra and 98 331 vertices.



(a) View from the outside. (b) Cut through the mesh.

Figure 5.2: Generated mesh for the famous *Stanford Dragon* [V] model. The input LS data has a grid delta of $\Delta g = 0.002$ and the violation threshold in the meshing algorithm was set to $\alpha = 0.225$. The produced mesh consists of 485 314 tetrahedra and 110 067 vertices.

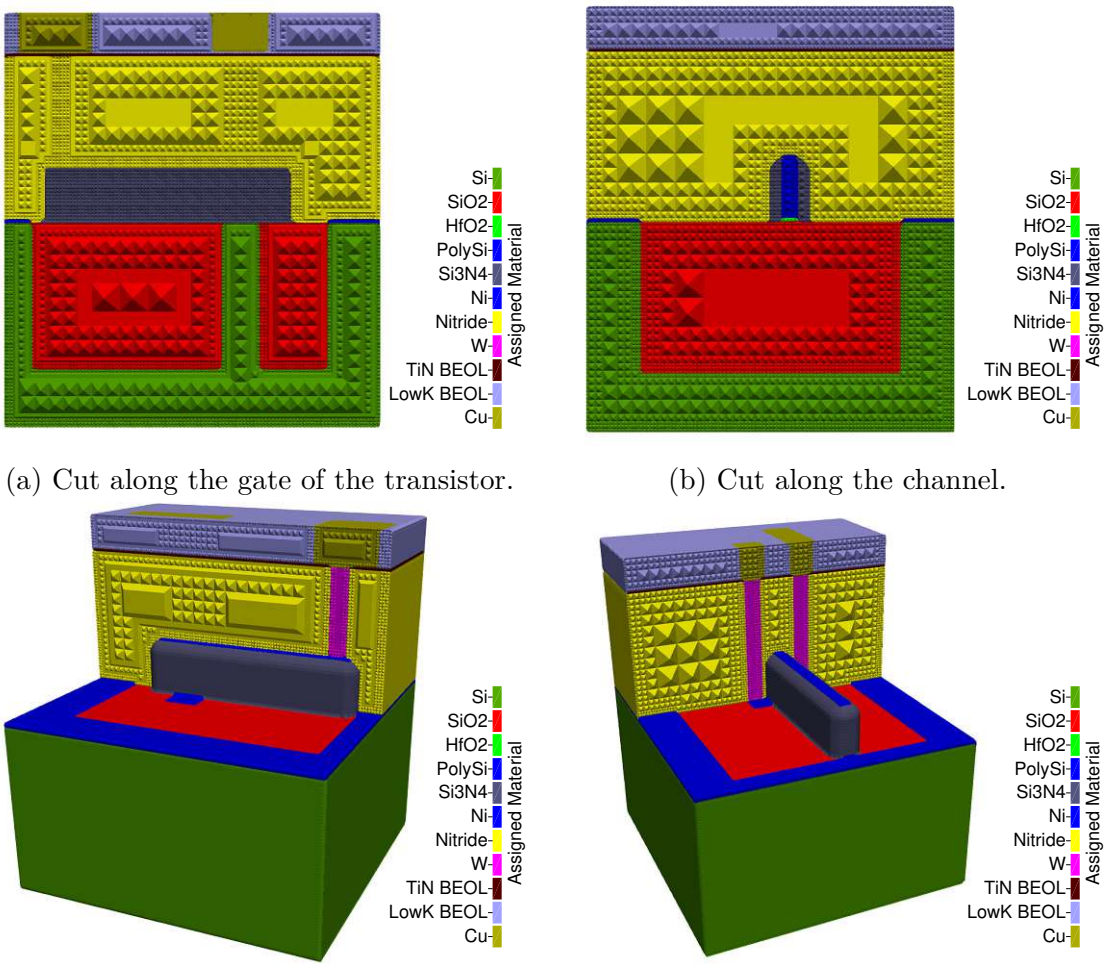
The single-material dodecahedron example was later on extended to the multi-material case again, by placing four different dodecahedra, such that they contain or intersect each other. The region of each dodecahedron was labeled with a separate material. A resulting mesh produced with this work's implementation is provided in Fig. 5.3. In Fig. 5.3b a thin material region assigned to material 1 between materials 0 and 2 can be seen. This thin region varies in thickness which is an artifact produced by the algorithm, due to the region being too thin to be resolved. Similar artifacts are were also found in [10] and highlight the need for a thin feature resolution in the implementation. These artifacts are caused by the vertex added to the center of each octree node and are the topic future research.



(a) View from the outside. (b) Cut through the mesh.

Figure 5.3: Generated mesh for a multi-material test structure of four different dodecahedra intersecting each other. Each dodecahedron represents a single material. The input LS data has a grid delta of $\Delta g = 0.15$ and the violation threshold in the meshing algorithm was set to $\alpha = 0.285$. The produced mesh consists of 176 741 tetrahedra and 35 425 vertices. The thin region of material 1 shows artifacts which are cause by the algorithm being unable to resolve the thin region.

As this work is intended to be the first step in an evaluation of the used meshing concept for the microelectronics sector, the implementation was also tested on a planar field-effect transistor (FET) model. The model was kindly provided by Xaver Klemenschits, courtesy GTS. A mesh generated for the planar FET model is shown in Fig. 5.4.



(a) Cut along the gate of the transistor.

(b) Cut along the channel.

(c) Open cut view along the gate.

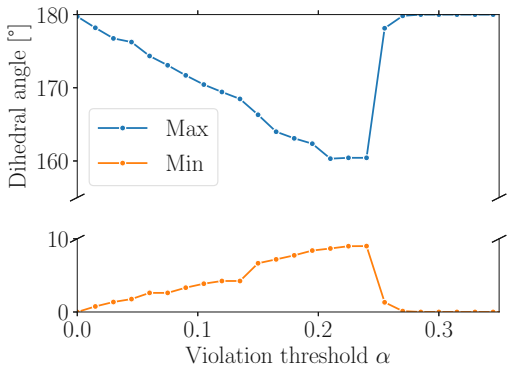
(d) Open view cut along the channel.

Figure 5.4: Mesh generated by the implemented algorithm, for a planar FET model with 11 different materials, based on the chemical composition of the transistor. The input LS data has a grid delta of $\Delta g = 0.006$ and the violation threshold in the meshing algorithm was set to $\alpha = 0.225$. The produced mesh consist of 4 939 376 tetrahedra and 936 524 vertices. The input model was kindly supplied by Xaver Klemenschits, courtesy of GTS.

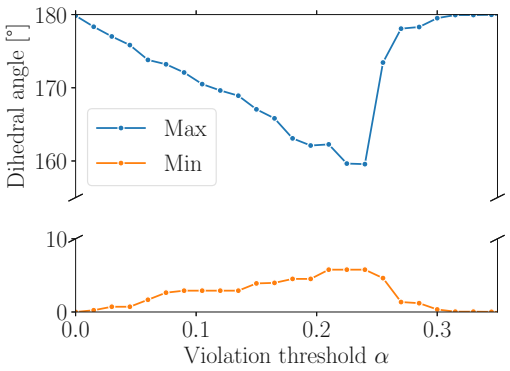
5.2 Violation Threshold

Because the implemented meshing algorithm depends on user input, namely the violation threshold, its impact on the quality of the produced mesh was examined. The violation threshold influences how many cuts are resolved by warping, and how many are left to be tessellated. As stated in [10], setting the threshold to $\alpha = 0$ will lead to all interface points being left as they are. This means that no warping is done at all, by which one yields the best geometric approximation to the interfaces [10]. Turning the threshold up to its potential maximum of $\alpha = 0.5$ will lead to all cuts being warped which, in turn, also leads to the generation of degenerate elements. Since, in this work, a structured mesh was used, no special determination of the threshold at every vertex was implemented, as suggested in [39]. Without such further considerations, the value at which degenerate tetrahedra are created depends on the tetrahedra which make up the background mesh, and will actually be lower than the upper bound of $\alpha = 0.5$. An example is shown in Fig. 5.5a or Fig. 5.5b, where the minimum dihedral angle in the last view data points drops to zero. In both plots, the optimum value is around 0.2, it was not deemed necessary to determine this value geometrically.

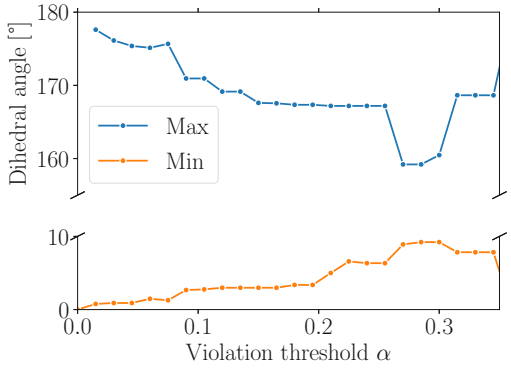
Through the resulting dihedral angles of the different tested models, the optimal violation threshold for the implemented algorithm is experimentally found to be $\alpha = 0.225$. This shall be reasoned by the fact that the optimal α value for three of the four examined models is at $\alpha = 0.225$ - see Fig. 5.5a, Fig. 5.5b, Fig. 5.5d, compared to Fig. 5.5c. Repeating the experiments with a denser set of thresholds would increase the precision of the found optimal value. The mentioned results also show that with an increasing minimum dihedral angle, also the maximum dihedral angle decreases, meaning that the overall range of dihedral angles improves for these cases. Whether or not there is a feasible way of predicting or pre-calculating the optimal threshold for a given background mesh remains to be explored. Related works also determined the optimal threshold by experimentation [10], [14].



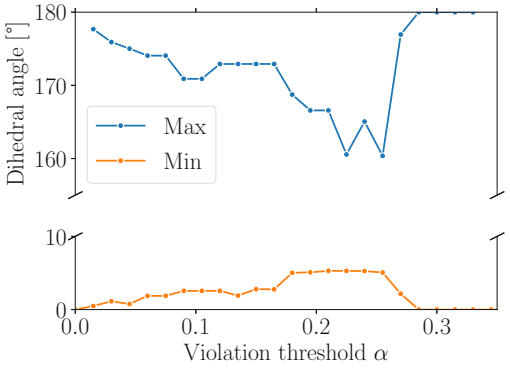
(a) Stanford Bunny model [IV], [VI] with $\Delta g = 2.0$.



(b) Stanford Dragon model [V] with $\Delta g = 0.002$.



(c) Four-material dodecahedron model with $\Delta g = 0.05$.



(d) Planar FET model with $\Delta g = 0.018$.

Figure 5.5: Resulting minimum and maximum dihedral angles using different violation threshold values on the same input.

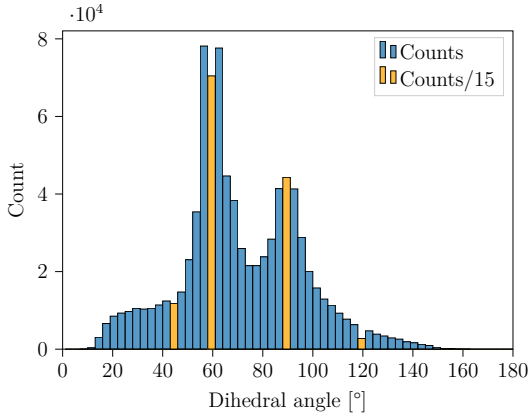
5.3 Dihedral Angles

Histograms of the resulting dihedral angles of the four presented meshes, Fig. 5.1, Fig. 5.2, Fig. 5.3, and Fig. 5.4, can be found in Fig. 5.6a, Fig. 5.6b, Fig. 5.6c, and Fig. 5.6d, respectively.

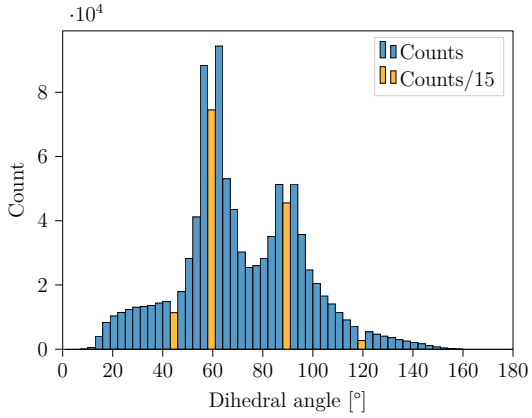
The dihedral angles found in the background mesh tetrahedra, both standard and bridging ones, were scaled down in the histogram plots. The scaling is however not equal between the four presented plots, as the ratio of these special dihedral angles to all other dihedral angles depend on the model itself. The larger the features within the model, the more unmodified background mesh tetrahedra there are.

The histogram plots Fig. 5.6a, Fig. 5.6b, Fig. 5.6c, and Fig. 5.6d, show a decrease in dihedral angles around the optimal value of $\arccos\frac{1}{3} \approx 1.231\,\text{rad} \approx 70.53°$. This is caused by the choice of background mesh lattice. The body centered cubic (BCC) lattice only contains the dihedral angles shown in orange, which are not at the optimal dihedral angle.

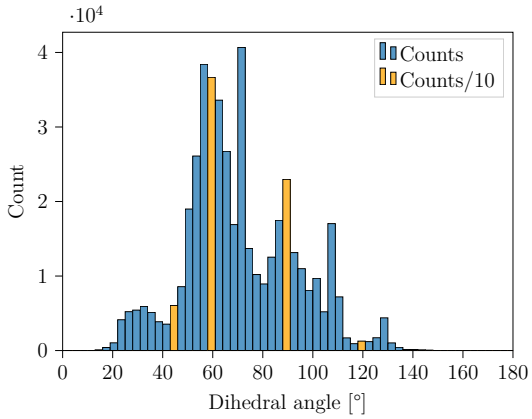
Additionally, the presented histograms suggest that the smoother models also lead to a smoother distribution in resulting dihedral angles. This would also mean that the approximation to the interfaces is working as intended, because smooth input interfaces have a smoother distribution and a wider range of angles, to which the output tetrahedra are conformed. For non-smooth inputs with sharp edges and corners, on the other hand, the resulting dihedral angles should be limited to the smaller number of angles of the input interfaces.



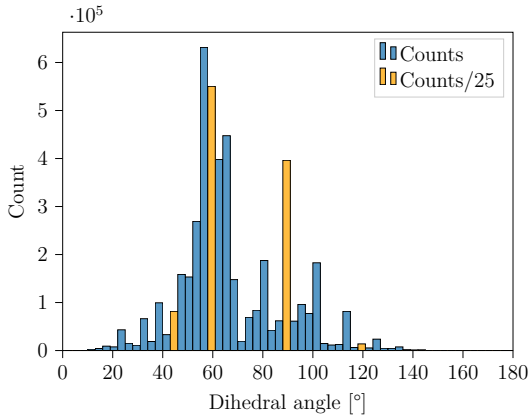
(a) Stanford Bunny model [IV], [VI] of Fig. 5.1. Dihedral angle range: $[8.99°, 160.43°]$.



(b) Stanford Dragon model [V] of Fig. 5.2. Dihedral angle range: $[5.78°, 159.64°]$.



(c) Four material dodecahedron test model of Fig. 5.3. Dihedral angle range: $[11.66°, 153.18°]$.



(d) Planar FET model of Fig. 5.4. Dihedral angle range: $[6.98°, 160.54°]$.

Figure 5.6: Histograms of dihedral angles of the generated meshes for different models. The background mesh only consists of the dihedral angles shown in orange. Since only the tetrahedra at the interfaces get modified by the mesh cleaving, the counts of the dihedral angles shown in orange were scaled down.

5.4 Runtime

Even though the implemented algorithm can work with single-material inputs, it is not optimized for it. Therefore, the performance in the single material case can be expected to be worse, when compared to dedicated single-material algorithms like the *isosurface stuffing*(-like) algorithms (compare Section 3.2). Worse runtime performance is expected due to the additional multi-material overhead, such as assigning virtual interface points or checking for degeneracies.

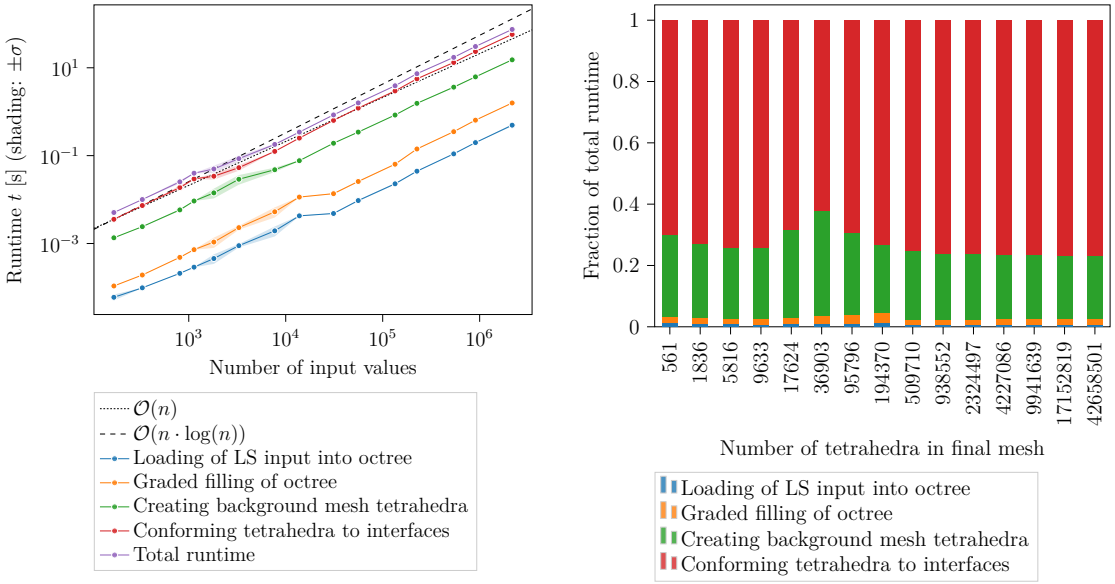
The runtime measurements were collected on a desktop personal computer (PC) with a AMD Ryzen™9 5950X 16-Core Processor and 32 GB of random-access memory (RAM). Runtimes were collected by running the same configuration five times and then averaging the gathered results. No results were excluded, since there were no obvious outliers. The average and standard deviation σ were calculated from this set of data. Ranges of a single standard deviation around the mean are indicated in this work by $\pm\sigma$.

The scaling of the algorithm sections and the total runtime can be seen in Fig. 5.7a and Fig. 5.8a, for the to presented multi-material models. Figure 5.7b and Fig. 5.8b, on the other hand, convey how the total time required to generate a mesh is composed of each part of the algorithm.

The creation of the octree substructure takes the least amount of time. Its two parts, the initial loading of the LS input data into the octree, and the graded filling procedure are similar in runtime, taking only about as long to fill the octree, as it does to copy the input data. This is, however, probably also caused by the dynamic allocation of the octree structure. Creating the first nodes in the octree requires repeated dynamic allocation of memory. When creating neighbors of those initial octree nodes, most of the octree's structure is already built. Therefore, only a small number of new nodes, and potentially their parents, need to be allocated.

All operations on the octree benefit from the implemented octree's fast neighbor access. Additionally, in this implementation only the direct neighbors around a node are required most of the time. For example, corner-only-neighbors are only necessary, when the required information on created vertices could not already be acquired from face or edge-only neighbors.

Element access in the used octree can be estimated to be of order $\mathcal{O}(\log(n))$, due to the recursive subdivision. This does not only apply to direct element access, but also to neighbor access. Loading the initial data therefore happens in order $\mathcal{O}(n \cdot \log(n))$ in the worst case, as each of the n input elements needs to be loaded and placed into the octree with $\mathcal{O}(\log(n))$. The octree's graded filling and the stenciling of the background mesh also happen in $\mathcal{O}(n \cdot \log(n))$ by similar considerations. The mesh cleaving needs to go over all tetrahedra of the mesh, only accessing incident elements, which leads to an overall linear time complexity of $\mathcal{O}(n)$. As the the background mesh depends on the initial LS input by $\mathcal{O}(\log(n))$,



(a) Runtimes shown are for individual components and not stacked. Lines for $\mathcal{O}(n)$ and $\mathcal{O}(n \cdot \log(n))$ are provided for visual aid.

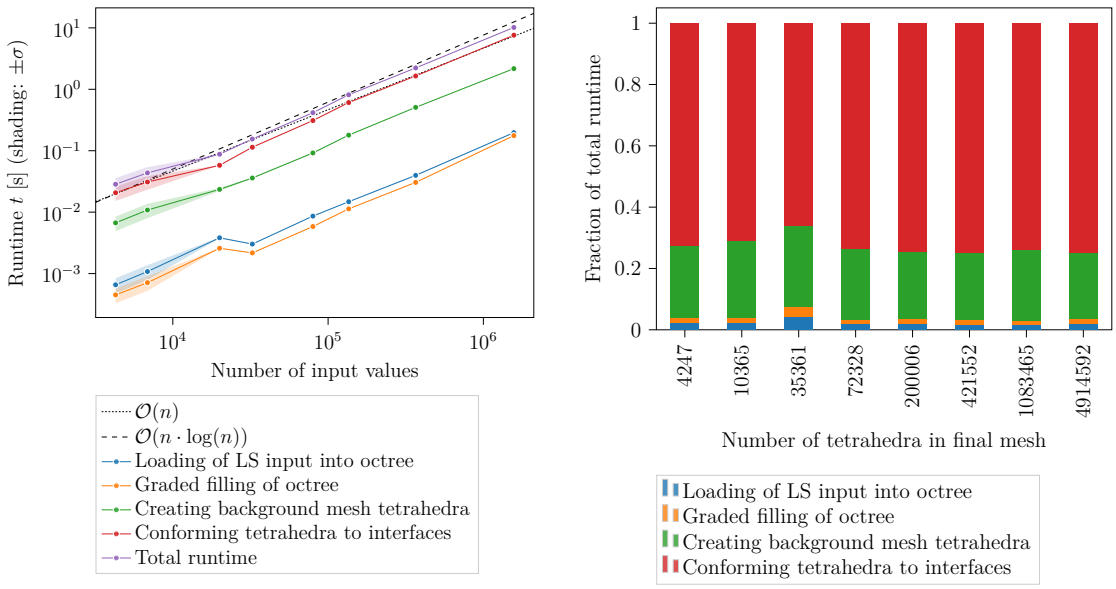
(b) Components contribution to total runtime.

Figure 5.7: Comparing the runtimes of different input sizes. The model used is the same, but the LS input has been generated with different grid spacings, resulting in different input and output data sizes. The model used in this figure is the four-material dodecahedron test model, run with the meshing algorithm set to a violation threshold of $\alpha = 0.225$.

and the mesh cleaving is of order $\mathcal{O}(n)$, the total algorithm has a runtime complexity of $\mathcal{O}(n \cdot \log(n))$. Figure 5.7a and Fig. 5.8a show that the reported mesh generation runtimes are slightly below $\mathcal{O}(n \cdot \log(n))$ which is expected, based on the fact that not all access in the octree happens in $\mathcal{O}(\log(n))$. For example neighbor access to direct siblings happens in constant time $\mathcal{O}(1)$, making the estimate of $\mathcal{O}(n \cdot \log(n))$ for the meshing only in the worst case, as mentioned above.

The creation of the background mesh takes considerably more time than the creation of the octree substructure. This is caused by the larger number of elements required, as every octree node is filled with multiple tetrahedra, each consisting of four vertices. Additionally, the operation of creating the background mesh tetrahedra, requires that the additional level set function (LSF) values are approximated.

Most of the mesh cleaving's operations are local to the individual elements, with only a few exceptions, such as projections of interfaces (see Section 4.5.4.2). Furthermore, the mesh cleaving only has to work on the tetrahedra on or near



(a) Runtimes shown are for individual components and not stacked. Lines for $\mathcal{O}(n)$ and $\mathcal{O}(n \cdot \log(n))$ are provided for visual aid.

(b) Components contribution to total runtime.

Figure 5.8: Comparing the runtimes of different input sizes. Runtimes of the algorithm's components in a are not stacked. To increase the number of input values, the model used is kept the same, while the LS input is generated with different grid spacings, thereby resulting in different input and output data sizes. The model used to generate this figure is the planar FET, executed with the meshing algorithm set to a violation threshold of $\alpha = 0.285$.

the interfaces. Therefore, the performance of the mesh cleaving part is tied to the surface area and the number of surface elements, rather than the total number of elements. Elements which are not located at an interface are visited once and directly pushed to the output. Based on the number of output tetrahedra, also the mesh cleaving scales well, even though it takes up more time than all of the preceding steps in the implemented meshing algorithm.

Chapter 6

Summary and Outlook

In this work, an implementation of a meshing algorithm based on the mesh cleaving concept [10], [39], [40] was presented. The algorithm was created to work with input data in the form of sparse Manhattan normalized level sets (LSs) adhering to the LS wrapping approach [1], [2].

6.1 Future Work

First and foremost, the feasibility of this algorithm outside of the finite element method (FEM) setting, for which it was originally conceptualized, needs to be investigated. This is important, since in the microelectronics sector, the finite volume method (FVM) is frequently applied. The FVM relies on the stringent Delaunay quality criteria, and so it remains to be seen, whether the meshes generated by this algorithm can be adjusted to fulfill the Delaunay criterion.

The second most import future research topic is the preservation of thin features which are represented in the input LS. These thin features cannot be handled by the mesh cleaving algorithm in its current form. The proposed method of retaining such thin features in the mesh is by further refining the octree substructure in areas, where thin features are detected, based on their level set function (LSF)-values. This would lead to more elements in the output mesh, but representing those thin features is critical for adequate representation and proper further simulation of microelectronic devices. Such a further refinement of the octree would however also lead to a revision of the octree balancing, as the graded filling procedure relies on the octree not only being in a balanced configuration, but also in a configuration which allows it to be filled in a balanced way, by only adding new nodes and without further refining existing nodes.

As the LS method leads to rounded corners and edges, it would be useful to implement a feature matching similar to that presented in the improved isosurface

stuffing algorithm in [38]. Even if the sharp corners and edges have no impact on the simulation, this would lead to a clearer representation of the simulated device, also for visualization reasons.

Another important point of future research should be an improvement in the creation of the background mesh. Both the feasibility of other crystal lattices, as well as different methods of creating a high quality, graded background mesh, without using a crystal lattice or an octree, or perhaps by using unstructured meshes (compare [39]). This topic of further research could benefit both the mesh quality and the reduction of tetrahedra in the output mesh, reducing the load put on the solvers of the respectively used partial differential equation (PDE).

In order to further decrease the time needed to generate a mesh, ways of parallelizing it could be investigated. Since the mesh cleaving part consumes the most time, parallelization of this step should be explored first. During the mesh cleaving, changes to vertices and interface points need to be reflected in all incident tetrahedra, which can lead to race conditions and makes parallelization highly challenging.

Appendix A

Additional Notes on Finding Interfaces

In the following sections the finding of interfaces point locations when using a more general level set (LS) definition, without the LS wrapping approach, is briefly explained.

A.1 Finding Triple-Points Irrespective of Level Set Wrapping

When using a more general approach to the LS definition without employing the LS wrapping approach, a triple-points initial location can be found by intersecting the linear approximations of the three materials. Based on the cut cases, as discussed in Section 4.5.1.1, each material must have exactly two cuts with the triangle face. Therefore the linear approximation of each material is uniquely defined. Using those three linear approximations finding the initial triple-point location simply comes down to calculating the intersection of the three lines. Of course the intersection of three lines in two-dimensional (2D) space is usually not unique, but can be one of three cases - a single point, a line, or a triangle. For the case of the intersection being a line or a triangle an additional approximation must be made. A typically choice would be simply taking the barycenter.

The final step of finding a triple-point location is checking whether the approximated initial triple-point location lies within the face's triangle. In case it does not, it is snapped to the nearest cut point and the triple-point becomes semi-virtual.

A.2 Finding Quad-Points Irrespective of Level Set Wrapping

To find the location of a quad-point, when not using the LS wrapping approach, a similar method can be applied to the finding of a triple-point in such a case, as is explained in Appendix A.1. For a quad-point, the considerations are the same, but the space which must be considered is three-dimensional (3D) instead of 2D. The quad-point's location can, therefore, be approximated to be at the intersection of the four materials linear approximations in 3D. Each material interface is either of the 3-cut, and the 4-cut, mentioned in Section 4.5.1.1. For the 3-cut this give a uniquely defined plane. In the 4-cut case, it can happen that the four points do not lie on a single plane in which the plane could be approximated through via fitting. Intersecting the four planes can result in a number of cases, which do not constitute a single point. In such a case again an barycentric approximation could be used to determine the quad-point's initial location.

Similar to the triple-point, it must be verified that the quad-point's initial location is within the tetrahedron, since otherwise it must be snapped to the nearest triple-point. In case the quad-point is snapped, it is created as semi-virtual instead of non-virtual.

Appendix B

Additional Notes on the Projection of Interfaces

The sections below give additional information regarding interface point projections, like projection procedures in the case of a more general level set (LS) definition which does not employ the LS wrapping approach.

B.1 Projection of Cut-Points Beyond the Edge

This section explains an approach to dealing with the forth case of Section 4.5.4.3, where the proposed location of a cut-point during its projection is beyond the new post-warp edge's line segment. This can only happen if the approach used for the calculation of the proposed cut-point position allows for a location beyond the line segment. The proposed location, given as a factor λ of the edge's length as seen from the warped vertex is then either $\lambda < 0$ or $\lambda > 1$.

In the case of the proposed position being beyond the edge, the cut-point is always snapped to the edge's other vertex. For $\lambda < 0$ this is the same as the first case of Section 4.5.4.3. For $\lambda > 1$,however, this is handled differently from the third case in Section 4.5.4.3, in order to avoid potentially skewing the other vertex's center of mass of violations calculation. The reason behind this is that a cut-point can, by its definition in the algorithm, only be located somewhere on the edge. When it is not directly snapped, the cut-point would have to be relocated onto the edge, for example, to the position of the other vertex. In case the other vertex was not already warped, the cut-point's position influences the location to which the other vertex is warped. Assuming that the cut-point's projection was relocated to stay on the edge and the cut-point was not directly snapped, its position potentially affects the center of mass, moving it away from the interface's actual position. This results in the other vertex's warp to be a worse

approximation to the interface. Hence, when a cut-point is beyond the edge, the cut is not relocated onto the edge, but rather it is directly snapped to the vertex.

As this direct snap of the forth case does not lead to any violation of the other vertex, the vertex does not need to be added to the queue of to-be-re-checked vertices.

B.2 Projection of Triple-Points Irrespective of Level Set Wrapping

A triple-point always lies on a ray from the the tetrahedron's original quad-point position through the triple-point's original position. The triple-point's position on the ray is where the ray intersects with the face on which the triple-point lies. Only non-virtual triple-points need to be projected, and non-virtual triple-points can only arise in the the 5-cut and the the 6-cut cases (compare Section 4.5.1.1). In the 5-cut case, there can only ever be a virtual quad-point and in the 6-cut case the quad-point can potentially be virtual from the beginning, by being created as a semi-virtual quad-point. Such a virtual quad-point can be co-located with the to-be-projected triple-point. In such a case, the original positions of the quad-point and the triple-point would be at the same point is space, leading to a degenerate ray for calculating the intersection. To resolve such a degenerate ray, one of the other triple-points, preferably one that has been created as a non-virtual one can be selected to emulate the quad-point for establishing the ray. Once the ray has been set up, it is a simple matter of calculating the intersection point of the ray and the new post-warp face.

As a triple-point is tied to a face and a face can be incident to two tetrahedra, one must also consider how to select the tetrahedron to perform the triple-point projection calculation. The only case in which it does not matter is if the rays constructed by each of the two tetrahedra are co-axial. For all other cases it is performed as discussed in Section 4.5.4.2, trying the incident tetrahedron with the smaller *sum of edge length changes* first, and only in case that this fails, the alternative.

The projected triple-point could be violating an incident vertex or edge. Since the edges are checked for violations later on on the mesh cleaving algorithm, they do not need to be checked during the triple-point projection. The vertices, however, need to be checked. As in the case where the projected triple-point violates a corner-vertex which has already been warped, it needs to be snapped to the corner-vertex. For a corner-vertex which has not been warped, the corner-vertex is based on its identifier (ID) potentially queue for re-checking, depending on the main iteration in which the algorithm is currently found (compare Section 4.5.4.1).

B.3 Projection of Quad-Points Irrespective of Level Set Wrapping

As explain in Section 4.5.4.5, the quad-point should represent a fixed point in space, and therefore the projection of a quad-point is mainly considered with the question whether the quad-point is still within the tetrahedron post warp. When the quad-point is outside of the post-warp tetrahedron, it should be mapped back into, or rather onto, the tetrahedron by the general idea of the mesh cleaving algorithm. However, even in the general case, disregarding the simplifications through the LS wrapping approach, there is no need to calculate an actual new location for a quad-point projection. The reason behind this is that any such projection would always automatically lead to a violation. Therefore, this step of calculating a projection point is omitted in this work.

In the general version of this implementation, the quad-point projection works in the following way. Check whether the quad-point is still within the tetrahedron post warp. If it is not, snap it to the geometrically closest interface point of any type.

In case the quad is still within the tetrahedron, it remains to be checked whether the quad violates any of the four corner-vertices. All other types of violations are checked thereafter, once all the warping has been completed. A violation of corner-vertex could have taken place prior to the warp with a corner-vertex which simply has not been checked yet. However, a corner-vertex violation can also have been caused by the warp, as with the warp also the violation zones which are measured in a factor of the edge length change, due to changes in the edges' lengths. When a corner-vertex is indeed violated by the quad-point post warp, the same strategy is applied as with all other post-projection corner-vertex violation checks - in case the corner-vertex was already warped, the quad-point is simply snapped to the vertex; if it has not been warped yet, the vertex is potentially queued for re-checking (compare Section 4.5.4.3 and Appendix B.2).

Appendix C

Additional Notes on Resolving Violations

In the following sections a brief explanation of resolving different types of violation in the general case is given. The general case, means that the violation resolution here is considered to not use the simplifications otherwise permitted by the level set (LS) wrapping approach.

C.1 Resolving Edge-Violations

Since triple-points and quad-points can violate an edge, every edge is checked in succession. For each edge, all of the incident faces and tetrahedra are collected. On each such face, the triple-point is examined, whether it is virtual or not. In case the triple-point is virtual, it is simply skipped as only non-virtual triples can cause a violation. When the triple-point violates the current edge, it is snapped to the edge's cut point. Through this snap, the triple-point can either becomes semi-virtual or pure-virtual. This depends on the cut point it is snapped to. In case the cut point is non-virtual the triple-point becomes semi-virtual, as it still represents a point where three materials meet. When the edge's cut point was already snapped itself and is therefore virtual, the triple-point also has to be labeled pure-virtual.

After checking all triple-points incident to the current edge, the incident quad-points are also checked. In case a quad-point is virtual and its hosting triple-point was snapped to the edge's cut point, the virtual quad-point needs to follow this snap and also be snapped to the edge's cut point. Non-virtual quad-points can violate the edge themselves. If this is the case, they are also snapped to the edge's cut point. The same rules as for the triple-points apply in regards to the virtuality type - a snap to a non-virtual cut point means that the quad-point becomes semi-

virtual, while snapping to a virtual cut point means the quad-point is turning pure-virtual.

In case any triple- or quad-point was snapped to the edge's cut point, all incident interface points need to be checked for potential degeneracies caused by the snap. Those degeneracies need to be resolved as discussed in Section 4.5.4.8 and, in this case, also the special degeneracy mentioned in Appendix D needs to be considered.

C.2 Resolving Face-Violations

Violations of faces can only be caused by non-virtual quad-points. Therefore, checking for face violation simply means to go over all tetrahedra, or better only the tetrahedra near the interface, to examine whether a non-virtual quad-point lies too close to any of the incident faces. When this is the case the quad-point is simply snapped to the corresponding face's triple-point and becomes a semi-virtual quad-point. This snap does not cause any further implications, as there is no higher tier interface point which needs to follow nor can this snap cause any degeneracies.

Appendix D

Additional Notes on Degeneracies

If two cut-points of a face snap to the same corner-vertex, a non-virtual triple-point on that face would also constitute a degeneracy, as after the snaps only two materials meet there.

Many arising degeneracies can also be viewed as being based on breaches of the interface point hierarchy. Such a breach happens, for example, when a non-virtual triple-point which is snapped to a corner-vertex, and the cut-points on the edges incident to the corner-vertex are non-virtual. This example also shows that this is indeed just another view on one of the already discussed degeneracies.

There is, however, a single special case of such a degeneracy caused by a breach of hierarchy, which is not covered yet. This special case is when a non-virtual quad-point is snapped to a non-virtual cut-point, but the triple-points on the faces incident to the cut-point's edge, are still non-virtual.

Bibliography

[1] O. Ertl, "Numerical methods for topography simulation", Doctoral dissertation, 2010. DOI: 10.34726/HSS.2010.001.

[2] X. Klemenschits, "Emulation and simulation of microelectronic fabrication processes", Doctoral dissertation, 2022. DOI: 10.34726/HSS.2022.89324.

[3] X. Klemenschits, S. Selberherr, and L. Filipovic, "Geometric advection algorithm for process emulation", in *Proc. International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, 2020, pp. 59–62. DOI: 10.23919/sispad49475.2020.9241678.

[4] M. Quell, "Parallel velocity extension and load-balanced re-distancing on hierarchical grids for high performance process tcad", Doctoral dissertation, 2022. DOI: 10.34726/HSS.2022.97084.

[5] K. Rupp, M. Bina, Y. Wimmer, A. Jungel, and T. Crasser, "Cell-centered finite volume schemes for semiconductor device simulation", in *Proc. International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, 2014, pp. 365–368. DOI: 10.1109/sispad.2014.6931639.

[6] Z. Stanojevic, M. Karner, K. Schnass, C. Kernstock, O. Baumgartner, and H. Kosina, "A versatile finite volume simulator for the analysis of electronic properties of nanostructures", in *Proc. International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, 2011, pp. 143–146. DOI: 10.1109/sispad.2011.6035089.

[7] J. E. Sanchez and Q. Chen, "Element edge based discretization for TCAD device simulation", *IEEE Transactions on Electron Devices*, vol. 68, no. 11, pp. 5414–5420, 2021. DOI: 10.1109/ted.2021.3094776.

[8] S. Micheletti, "Stabilized finite elements for semiconductor device simulation", *Computing and Visualization in Science*, vol. 3, no. 4, pp. 177–183, 2001. DOI: 10.1007/s007910000046.

[9] J. J. H. Miller, W. H. A. Schilders, and S. Wang, "Application of finite element methods to the simulation of semiconductor devices", *Reports on Progress in Physics*, vol. 62, no. 3, pp. 277–353, 1999. DOI: 10.1088/0034-4885/62/3/001.

[10] J. Bronson, J. A. Levine, and R. Whitaker, "Lattice cleaving: A multimaterial tetrahedral meshing algorithm with guarantees", *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 2, pp. 223–237, 2014. DOI: 10.1109/tvcg.2013.115.

[11] J. R. Shewchuk, "What is a good linear element? Interpolation, conditioning, and quality measures", in *Proc. International Meshing Roundtable (IMR)*, 2002, pp. 115–126. [Online]. Available: https://people.eecs.berkeley.edu/~jrs/papers/elem.pdf (visited on 07/24/2023).

[12] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: Concepts, implementation and applications", *The Visual Computer*, vol. 11, no. 1, pp. 429–446, 1995. DOI: 10.1007/BF02464333.

[13] J. Hicken and S. Kaur, "An explicit level-set formula to approximate geometries", in *Proc. AIAA Science and Technology Forum*, 2022, pp. 1–17. DOI: 10.2514/6.2022-1862.

[14] F. Labelle and J. R. Shewchuk, "Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles", in *Proc. ACM Special Interest Group on Computer Graphics (ACM SIGGRAPH)*, 2007, pp. 57–66. DOI: 10.1145/1275808.1276448.

[15] J. Duriez and C. Galusinsky, "Level set representation on octree for granular material with arbitrary grain shape", in *Proc. Topical Problems of Fluid Mechanics*, 2020, pp. 64–71. DOI: 10.14311/tpfm.2020.009.

[16] C. Min and F. Gibou, "A second order accurate level set method on non-graded adaptive cartesian grids", *Journal of Computational Physics*, vol. 225, no. 1, pp. 300–321, 2007. DOI: 10.1016/j.jcp.2006.11.034.

[17] S. Kambampati, C. Jauregui, K. Museth, and H. A. Kim, "Large-scale level set topology optimization for elasticity and feat conduction", *Structural and Multidisciplinary Optimization*, vol. 61, no. 1, pp. 19–38, 2019. DOI: 10.1007/s00158-019-02440-2.

[18] S. Saien, H. A. Moghaddam, and M. Fathian, "A unified methodology based on sparse field level sets and boosting algorithms for false positives reduction in lung nodules detection", *International Journal of Computer Assisted Radiology and Surgery*, vol. 13, no. 3, pp. 397–409, 2017. DOI: 10.1007/s11548-017-1656-8.

[19] M. Warren and J. Salmon, "A parallel hashed oct-tree n-body algorithm", in *Proc. ACM/IEEE Supercomputing*, 1993, pp. 12–21. DOI: 10.1145/169627.169640.

[20] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel", *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008. DOI: 10.1137/070681727.

[21] T. Isaac, C. Burstedde, and O. Ghattas, "Low-cost parallel algorithms for 2:1 octree balance", in *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012, pp. 426–437. DOI: 10.1109/ipdps.2012.47.

[22] J. Baert, A. Lagae, and P. Dutré, "Out-of-core construction of sparse voxel octrees", in *Proc. High-Performance Graphics Conference*, 2013, pp. 27–32. DOI: 10.1145/2492045.2492048.

[23] D. Eppstein, J. M. Sullivan, and A. Üngör, "Tiling space and slabs with acute tetrahedra", *Computational Geometry*, vol. 27, no. 3, pp. 237–255, 2004. DOI: 10.1016/j.comgeo.2003.11.003.

[24] M. Bern, D. Eppstein, and J. Gilbert, "Provably good mesh generation", *Journal of Computer and System Sciences*, vol. 48, no. 3, pp. 384–409, 1994. DOI: 10.1016/s0022-0000(05)80059-5.

[25] M. Bern, L. Chew, D. Eppstein, and J. Ruppert, "Dihedral bounds for mesh generation in high dimensions.", in *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1995, pp. 189–196. DOI: 10.5555/313651.313694.

[26] P. Fleischmann, "Mesh generation for technology CAD in three dimensions", Doctoral dissertation, 1999. DOI: 10.34726/HSS.1999.03012084.

[27] H. Edelsbrunner, "Spatial triangulations with dihedral angle conditions", in *Proc. International Workshop on Discrete Algorithms and Complexity*, 1989, pp. 83–89. [Online]. Available: http://id.nii.ac.jp/1001/00032670/ (visited on 07/20/2023).

[28] V. T. Rajan, "Optimality of the Delaunay triangulation in $\mathbb{R}^d$", *Discrete & Computational Geometry*, vol. 12, no. 2, pp. 189–202, Dec. 1994. [Online]. Available: https://doi.org/10.1007/BF02574375.

[29] J. Wang and Z. Yu, "Feature-sensitive tetrahedral mesh generation with guaranteed quality", *Computer-Aided Design*, vol. 44, no. 5, pp. 400–412, 2012. DOI: 10.1016/j.cad.2012.01.002.

[30] B. Bagley, S. P. Sastry, and R. T. Whitaker, "A marching-tetrahedra algorithm for feature-preserving meshing of piecewise-smooth implicit surfaces", *Procedia Engineering*, vol. 163, no. 1, pp. 162–174, 2016. DOI: 10.1016/j.proeng.2016.11.042.

[31] J. Schöberl, "NETGEN an advancing front 2D/3D-mesh generator based on abstract rules", *Computing and Visualization in Science*, vol. 1, no. 1, pp. 41–52, 1997. DOI: 10.1007/s007910050004.

[32] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm", *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987. DOI: 10.1145/37402.37422.

[33] L. P. Chew, "Guaranteed-quality triangular meshes", Tech. Rep., Apr. 1989. DOI: 10.21236/ada210101.

[34] X.-Y. Li, "Generating well-shaped d-dimensional Delaunay meshes", *Theoretical Computer Science*, vol. 296, no. 1, pp. 145–165, 2003. DOI: 10.1016/s0304-3975(02)00437-1.

[35] D. L. Marcum and N. P. Weatherill, "Unstructured grid generation using iterative point insertion and local reconnection", *AIAA Journal*, vol. 33, no. 9, pp. 1619–1625, 1995. DOI: 10.2514/3.12701.

[36] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng, "Sliver exudation", in *Proc. Annual symposium on Computational Geometry*, 1999, pp. 883–904. DOI: 10.1145/304893.304894.

[37] J. Teran, N. Molino, R. Fedkiw, and R. Bridson, "Adaptive physics based tetrahedral mesh generation using level sets", *Engineering With Computers*, vol. 21, no. 1, pp. 2–18, 2005. DOI: 10.1007/s00366-005-0308-8.

[38] C. Doran, "Isosurface stuffing improved: Acute lattices and feature matching", M.S. thesis, 2013. DOI: 10.14288/1.0052176.

[39] J. Bronson, S. Sastry, J. Levine, and R. Whitaker, "Adaptive and unstructured mesh cleaving", *Procedia Engineering*, vol. 82, no. 1, pp. 266–278, 2014. DOI: 10.1016/j.proeng.2014.10.389.

[40] J. Bronson, J. Levine, and R. Whitaker, "Lattice cleaving: Conforming tetrahedral meshes of multimaterial domains with bounded quality", in *Proc. International Meshing Roundtable*, 2012, pp. 191–209. DOI: 10.1007/978-3-642-33573-0_12.

[41] J. Bloomenthal, "An implicit surface polygonizer", in *Graphics Gems*, 1994, pp. 324–349. DOI: 10.1016/b978-0-12-336156-1.50040-9.

[42] D. M. Y. Sommerville, "Space-filling tetrahedra in Euclidean space", in *Proc. Edinburgh Mathematical Society*, 1922, pp. 49–57. DOI: 10.1017/s001309150007783x.

Software and Model References

[I] R. Bridson and C. Dorian, "Quartet", 2014, Software. [Online]. Available: https://github.com/crawforddoran/quartet (visited on 04/18/2023).

[II] CIBC - Scientific Computing and Imaging Institute (SCI), "Cleaver, A multimaterial tetrahedral meshing library and application", version 2.4, 2015, Software. [Online]. Available: https://www.sci.utah.edu/software/cleaver.html (visited on 06/08/2023).

[III] Institute for Microelectronics at TU Wien, "ViennaLS", version 2.1.0, Software. [Online]. Available: https://github.com/ViennaTools/ViennaLS (visited on 11/21/2022).

[IV] Stanford University Computer Graphics Laboratory, "Stanford Bunny", 1996, 3D Model. [Online]. Available: https://graphics.stanford.edu/data/3Dscanrep/ (visited on 08/11/2023).

[V] Stanford University Computer Graphics Laboratory, "Stanford Dragon", 1996, 3D Model. [Online]. Available: https://graphics.stanford.edu/data/3Dscanrep/ (visited on 08/11/2023).

[VI] C. Batty, "Watertight Stanford Bunny", 3D Model. [Online]. Available: https://cs.uwaterloo.ca/~c2batty/ (visited on 08/11/2023).