Master Thesis

Derivation and implementation of a hanging nodes resolution scheme for hexahedral non-conforming meshes in `t8code`

Tabea Leistikow

February 21, 2024

First Reviewer: Prof. Dr.-Ing. Gregor Gassner
Second Reviewer: Dr. Johannes Markert

Numerical Simulation
Division of Mathematics
University of Cologne

Faculty of Mathematics and Natural Sciences of
University of Cologne
and
German Aerospace Center

Universität zu Köln

Deutsches Zentrum für Luft- und Raumfahrt
German Aerospace Center

Contents

1 Introduction

Partial differential equations (PDEs) are often used to describe processes in the natural world [38]. To solve PDEs on geometries, an appropriate discretization is required. The discretization is usually given in the form of a mesh structure. There are several different methods in order to solve PDEs on meshes, for example, the finite difference (FD), finite element method (FEM), finite volume (FV), and the discontinuous Galerkin method (DG) [15, 31].

In order to discretize a 3-dimensional domain, it is very common to either use hexahedral or tetrahedral elements. The advantages and disadvantages of using hexahedral or tetrahedral elements are a much-discussed topic in the literature [42, 39, 12, 44]. In general, tetrahedral elements are more suitable to discretize complex geometries. Although, hexahedral elements need fewer elements to discretize the same geometry size.

However, depending on the geometry, it can also be advantageous to use tetrahedral and hexahedral elements for discretizing different parts of the domain. This results in a hybrid mesh. Thereby, it is possible to use the favorable properties of each element type. When using tetrahedral and hexahedral elements it may be necessary to use prisms and/or pyramids between those two element types [48, 34]. Naturally, it is also possible to use tetrahedral and prismatic elements like in [25], but this is rare.

Not only the element type is decisive in improving the quality of the discretization. The elements size is also an important property. The initial discretization may not be sufficient. One possible approach to enhance the discretization is to use smaller elements, called uniform mesh. With this approach, consequently, more elements are needed to discretize the whole domain resulting in extensive additional computational effort. Therefore, using smaller elements all over the underlying domain would decrease the numerical simulation's performance.

Another approach is to define particular regions, called refinement regions, where the initial discretization is insufficient, and thus, a more precise approximation is needed. This approach is called adaptive mesh refinement (AMR), see for instance [2, 14]. The use of error estimators is beneficial to identify the regions where a finer mesh is necessary [6, 2, 47]. Refining an element thereby means replacing it by mostly equally shaped, smaller elements. For example, refining a hexahedron means replacing it with eight smaller ones. Refining a pyramid, however, leads to smaller pyramids and tetrahedra, discussed in detail in [27]. Consequently, with AMR, it is possible to refine the mesh locally to discretize particular regions more precisely and leave the rest coarse.

Thus, the big advantage of AMR compared to a uniform mesh is the lower number of elements, smaller memory footprint, and a faster runtime of numerical solvers. However, a disadvantage of AMR is the accompanying mesh management. Especially when parallelizing the computations, a lot of additional tasks such as the random access of elements and the partition on different processes need to be realized. Due to the high complexity of this mesh management, there are AMR-specialized software libraries like `p4est` [10], `t8code` [20] that builds on `p4est` and `libMesh` [26].

Refining an element leads to parent-child relation and, therefore, to a tree structure. Considering a hexahedral mesh, one obtains an octree [40, 45].

One main task of AMR software is to identify each element uniquely. There are different ways of approaching this, for example, by using the concept of space-filling curves (SFCs) [19, 37, 3]. The idea of a SFC is that it maps each mesh element to a linear index, like numbering. This index is then a unique ID of the element. The AMR software `t8code` uses the Morton-SFC. Thus, its Morton index can uniquely address each element.

Most numerical PDE solvers, for example, those using the FV method, need neighbor-relations of elements to update their values over different time steps. Thus, they need a one-to-one correspondence between the faces of elements and their face-neighbors. Meshes that fulfill this one-to-one correspondence are called conformal. Conformal meshes are even a precondition for some solvers, see [49].

A mesh with AMR is, in general, not conformal. That means there are face-neighbored elements with different sizes. These neighbor-relations create so-called hanging faces. For example, if a hexahedral element has a hexahedral face-neighbor that is refined once, then there is a face with four face-neighbors, thus, there is no one-to-one correspondence.

There are different ways to handle hanging nodes [16, 36, 13, 50, 43, 39]. One approach is to identify hanging nodes and insert transition cells into the corresponding elements of the mesh in order to eliminate these hanging nodes. This approach is discussed in detail in this thesis.

In [5], Becker analyzed 2-dimensional quadrilateral meshes with hanging nodes. He developed different transition patterns and a `transition` algorithm that makes a mesh conformal there. This thesis builds on the work of Becker and extends this approach to 3-dimensional hexahedral meshes. The main challenge is that there are a lot of different cases that can cause hanging nodes leading to many different transition cells. Moreover, refining an element into a transition cell brings irregularities into the SFC. Additionally, a new face-neighboring algorithm needs to be developed.

All presented algorithms are implemented in a new transition-scheme in the open-source software `t8code`. The goal is to enable `transition` as a new key-feature of `t8code`.

This thesis is organized as follows: First, in Chapter 2, the fundamentals of AMR and SFCs are presented. After that, in Chapter 3, the problem of hanging nodes is discussed. Therefore, the overall problem of hanging nodes is discussed first in Chapter 3.1. After that, in Chapter 3.2, different approaches to solving the problem of hanging nodes are proposed. Afterward, in Chapter 3.3, the different transition cells are introduced. After that the modified SFC is presented in Chapter 3.4 for transition cells and Chapter 3.5 for forests. Chapter 4 covers the implementations in `t8code`. First, in Chapter 4.1, the high-level algorithms are introduced. After that, in Chapters 4.2 and 4.3, the requirements for implementing the `transition` algorithm are presented. In Chapter 4.4, the `transition` algorithm is presented. Chapter 5 discusses the problem of identifying face neighbors in a transitioned mesh. Chapter 5.1 focuses on identifying face-neighbors inside a forest, while Chapter 5.2 handles identifying face-neighbors inside a transition cell. Afterward, in Chapter 6, the impact of the transition cells on the amount of elements, Chapter 6.1, on the general runtime, Chapter 6.2, and the runtime of the neighbor-identifying algorithm of `t8code`, Chapter 6.3, are discussed. Then, in Chapter 6.4, two different quality metrics are discussed on the transitioned mesh.

One is jacobian-based in Chapter 6.4.1, and one is based on the aspect-ratio in Chapter 6.4.2. In the last part, Chapter 7, problems and remaining tasks of this approach are discussed. Therefore, the remaining task of edge-balancing is introduced in Chapter 7.1. Afterward, in Chapter 7.2, the problem of the occurrence of hanging edges inside the presented transition cells is discussed. A summary of all problems and solution strategies is given at the end of Chapter 7.

2 Theory

This chapter is guided by [20] and [27]. First, theoretical fundamentals of adaptive mesh refinement are given. After that, the concept of space-filling curves (SFC) is introduced. Afterward, the Morton index for hexahedral elements and the extension of SFC for forests are presented.

2.1 Adaptive Mesh Refinement

First, we need to describe what a mesh is and how to manage the elements in a mesh. To approximate the original geometry we use polygons as geometric elements. In this thesis, we mainly focus on hexahedral and pyramid-shaped elements. Therefore, we define a legal discretization as follows:

Definition 2.1.1. Let $\Omega \subset \mathbb{R}^d$ be bounded and with a polygonal boundary.
$A = \{A_i \mid i \in I,\ A_i \text{ is a Polygon}\}$ is a **legal discretization** of Ω with indices I, if

(i) $\overline{\Omega} \;=\; \bigcup_{i \in I} A_i$

(ii) If $A_i \cap A_j \neq \emptyset$, then $A_i \cap A_j$ is an $(d-k)$-dimensional sub-polygon of A_i and A_j with $0 \leq k \leq d$.

This means that the intersection of boundaries is either a common face in 3D, an edge, a vertex or empty. We call a legal discretization of Ω a *mesh*.

There are different ways to discretize an object. One possibility is to only us one type of polygons, such as quadrilaterals or triangles in 2D and hexahedra or tetrahedra in 3D. The mesh is called *hybrid* if different types of polygons are used in one mesh. If every element in a mesh has the same size, we call the mesh *uniform*. It always depends on the underlying geometry which type(s) of polygon(s) fit the best.

A wide range of discussions on the type of mesh can be found in the literature. For example, in [39], they discuss the differences between unstructured triangular/tetrahedral and structured quadrilateral/hexahedral meshes. In [12] they conduct a performance study of tetrahedral and hexahedral meshes. The result is that the tetrahedral and hexahedral are fairly equivalent in terms of CPU time and accuracy. In [39, 46], they show that a clear benefit of quadrilateral/hexahedral meshes is that we can use fewer elements to model the same domain size compared to a triangular/tetrahedral mesh.

With the initial discretization, details of the discretized object may get lost, or the solution error may be very high in a specific region. Then, it is necessary to improve the discretization. The idea of AMR is to refine the mesh only where it is needed. The finer the mesh, the smaller the computational error and the more accurate the discretization. Therefore, one can subdivide polygons of the mesh into smaller ones. In this way, we achieve a hierarchical structure of the elements. The smaller ones are called "children" and the bigger ones that have been refined are called "parent". The children have the same polygonal structure as their parent for some element types like hexahedra, triangles, or quadrilaterals. Refining a pyramid, however, leads to partly different polygonal structures of the children. In [27], pyramidal adaptive meshes are discussed in detail.

This thesis follows the idea of recursive refinement patterns. This means that elements are replaced by smaller ones via a repeated refinement process. Figure 2.1 shows a recursive $1:4$ refinement on triangles and a $1:8$ refinement on hexahedra. Note that every element thus has a specific refinement level which increases while refining.

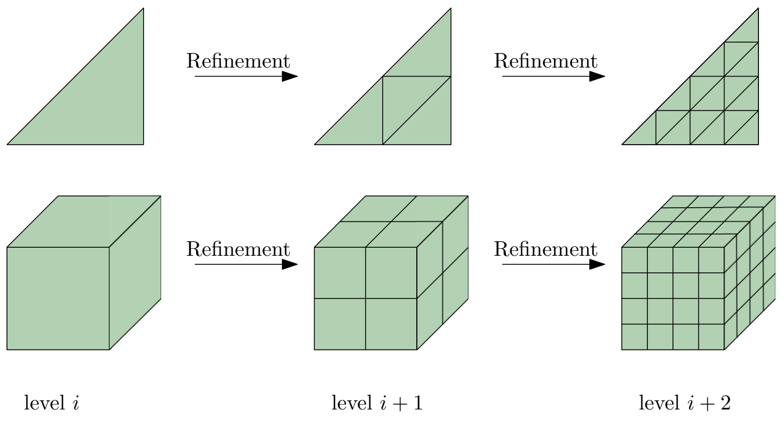


Figure 2.1: This figure illustrates recursively refined triangles in 2D (top) and hexahedral elements in 3D (bottom). The level of each element increases from the initial level i (left) to level $i+2$ (right).

AMR aims to achieve the same computational error with less elements and consequently with less memory use and a faster runtime for numerical solvers. Due to the use of elements with different levels and hence differing sizes the mesh management becomes more complicated compared to a uniform mesh [21]. In Figure 2.2, the difference between a uniform and an adaptive mesh is illustrated.

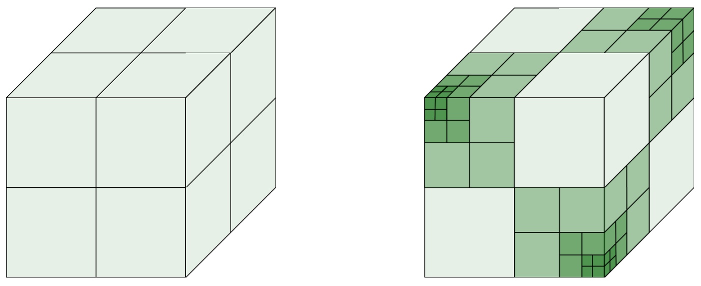


Figure 2.2: This figure shows a uniform mesh on the left and an adaptive mesh on the right. In the adaptive mesh, a higher level, thus a smaller element size, is depicted in a darker color.

There are different kinds of AMR like unstructured, block-structured or tree-based AMR. This thesis explores the topic of tree-based AMR. The core idea of tree-based AMR is to mesh a domain, for example, with a coarse hexahedron. Thus, the mesh contains only one element at first. This element is called the *root element*. Refining this element, i.e., replacing it with eight smaller hexahedra can be represented in a refinement tree. The leaves of the refinement tree represent the elements of the mesh. The refining operation can be repeated arbitrarily. The root element has level zero. Refining an element increases the element's level by

one.
A refinement space represents one specific mesh and holds all the possible mesh configurations.

Definition 2.1.2. A **refinement space** G is a triple $(\mathcal{S}, \ell, R)$, where

- Elements: $\mathcal{S}$ is a set of elements,
- Level: $\ell : \mathcal{S} \to \mathbb{N}_0$ is the level function
- Refinement maps: $R = \{R^l \mid l \in \mathbb{N}_0\}$ is a set of refinement maps $R^l : \mathcal{S}^l \to \mathcal{P}(\mathcal{S}^{l+1})$ with $\mathcal{S}^l = \ell^{-1}(l)$ and the power set $\mathcal{P}$, such that
  - (i) there exists exactly one element $E \in \mathcal{S}$ with $\ell(E) = 0$ (the root element) and
  - (ii) the image of R^l is a partition of $\mathcal{S}^{l+1}$:

$$\begin{aligned} R^l(E) \cap R^l(E') = \emptyset \text{ for } E \neq E' \in S^l \\ \bigcup_{E \in \mathcal{S}^l} R^l(E) = \mathcal{S}^{l+1} \end{aligned} \tag{1}$$

In the context of tree-based AMR, the level function ℓ of an element E determines its distance to the root element. The refinement map R^l specifies how to refine an element of level l into multiple smaller ones. In this thesis, only isotropic refinements are considered. This means that if we refine an element, we slice it along all dimensions. Using the example of a hexahedron, we slice it in x- y- and z-direction. In the case of anisotropic refinements, specific dimensions are needed to be determined for cutting. Thus, refining, for example, a regular-shaped hexahedron in an anisotropic way can result in children who are not regular-shaped hexahedra. Nevertheless, considering isotropic refinement of a regular-shaped hexahedron always results in regular-shaped hexahedra. To state that recursive refinements are valid, the following definition of a refinement is needed.

Definition 2.1.3. Let $\mathcal{S}$ be the set of elements of a refinement space $\mathcal{G}$. A **refinement** $\mathcal{K}$ is a subset $\mathcal{K} \in \mathcal{S}$ that is recursively defined by the following two rules:

(1) $\mathcal{K} = \mathcal{S}^0$ is a refinement and

(2) if $\mathcal{K}$ is a refinement and $E \in \mathcal{K}$, then $\mathcal{K} \setminus E \cup R^{\ell(E)}(E)$ is a refinement.

It follows that a refinement space containing elements at different levels can be valid. We say that a mesh fulfills a 2:1 **balance**-condition, if the level of face neighbors in the mesh differs at most by ± 1. If the balance-condition is not fulfilled, we call the mesh **unbalanced**. As already mentioned, the focus of this thesis is on hexahedral elements. Therefore, the following definition explains the faces and children of a hexahedron. Figure 2.3 illustrates the vertices, edges and faces of a regular hexahedron.

Definition 2.1.4. A hexahedral element E is defined by its eight vertices

$$E := [\vec{v_0}, \vec{v_1}, \vec{v_2}, \vec{v_3}, \vec{v_4}, \vec{v_5}, \vec{v_6}, \vec{v_7}], \tag{2}$$

where $f_0 := [\vec{v_0}, \vec{v_2}, \vec{v_4}, \vec{v_6}]$ defines the left, $f_1 := [\vec{v_1}, \vec{v_3}, \vec{v_5}, \vec{v_7}]$ defines the right, $f_2 := [\vec{v_0}, \vec{v_1}, \vec{v_4}, \vec{v_5}]$ defines the front, $f_3 := [\vec{v_2}, \vec{v_3}, \vec{v_6}, \vec{v_7}]$ defines the back, $f_4 := [\vec{v_0}, \vec{v_1}, \vec{v_2}, \vec{v_3}]$ defines the bottom and $f_5 := [\vec{v_4}, \vec{v_5}, \vec{v_6}, \vec{v_7}]$ defines the top face.
The eight hexahedral children $H_0, \dots, H_7$ of the hexahedron E are given by:

$$\begin{aligned}
H_0 &:= [\vec{v}_0, \vec{v}_{01}, \vec{v}_{02}, \vec{v}_{03}, \vec{v}_{04}, \vec{v}_{05}, \vec{v}_{06}, \vec{v}_{07}] \\
H_1 &:= [\vec{v}_{01}, \vec{v}_1, \vec{v}_{03}, \vec{v}_{13}, \vec{v}_{05}, \vec{v}_{15}, \vec{v}_{07}, \vec{v}_{17}] \\
H_2 &:= [\vec{v}_{02}, \vec{v}_{03}, \vec{v}_2, \vec{v}_{23}, \vec{v}_{06}, \vec{v}_{07}, \vec{v}_{26}, \vec{v}_{27}] \\
H_3 &:= [\vec{v}_{03}, \vec{v}_{13}, \vec{v}_{23}, \vec{v}_3, \vec{v}_{07}, \vec{v}_{17}, \vec{v}_{27}, \vec{v}_{37}] \\
H_4 &:= [\vec{v}_{04}, \vec{v}_{05}, \vec{v}_{06}, \vec{v}_{07}, \vec{v}_4, \vec{v}_{45}, \vec{v}_{46}, \vec{v}_{47}] \\
H_5 &:= [\vec{v}_{05}, \vec{v}_{15}, \vec{v}_{07}, \vec{v}_{17}, \vec{v}_{45}, \vec{v}_5, \vec{v}_{47}, \vec{v}_{57}] \\
H_6 &:= [\vec{v}_{06}, \vec{v}_{07}, \vec{v}_{26}, \vec{v}_{27}, \vec{v}_{46}, \vec{v}_{47}, \vec{v}_6, \vec{v}_{67}] \\
H_7 &:= [\vec{v}_{07}, \vec{v}_{17}, \vec{v}_{27}, \vec{v}_{37}, \vec{v}_{47}, \vec{v}_{57}, \vec{v}_{67}, \vec{v}_7],
\end{aligned} \tag{3}$$

where $\vec{v}_{ij} := \frac{\vec{v}_i + \vec{v}_j}{2}$.

Figure 2.4 shows the children of a regular hexahedron and its orientation. The given orientation is the underlying orientation of every element that will be introduced in this thesis.

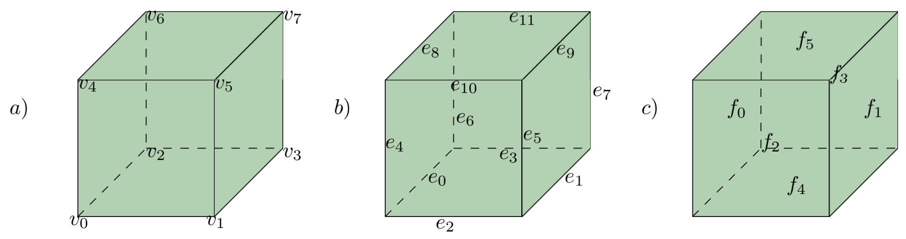


Figure 2.3: Illustration a) shows the vertices $v_0, \dots, v_7$ of a regular hexahedron. Part b) depicts the edges $e_0, \dots, e_{11}$ and c) shows the faces $f_0, \dots, f_5$, as defined in 2.1.4.

In the following we define different tree-based AMR specific expressions.

Definition 2.1.5. Let $\mathcal{G}$ be a refinement space as determined in Definition 2.1.2 and $E, E' \in \mathcal{S}$.

- E is named the **child** of E' if $E \in R^{\ell(E')}(E')$. E' is then called the **parent** of E.
- The set $C = \{C_0, \dots, C_k\} = R^{\ell(E)}(E)$ forms a **family** of **children** of E. From the point of view of a child C_i with $0 \le i \le k$, the elements in $C \setminus \{C_i\}$ are its **siblings**.
- Refinement maps can be concatenated via $R^{\ell(E')-1} \circ \dots \circ R^{\ell(E)}$ to construct a set D of elements with a higher refinement level than E. The set D

possesses the following property: In order to construct an element in D from the root element, one has to construct the element E at some point. Hence, if $E' \in R^{\ell(E)}(E)$ and $\ell(E') > \ell(E)$, then E' is a **descendant** of E and vice versa, E is an **ancestor** of E'.

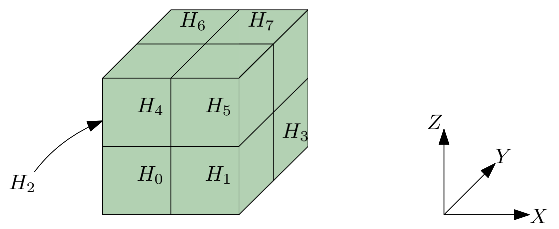


Figure 2.4: This figure shows the eight children $H_0, \ldots, H_7$ of a hexahedron with the underlying orientation.

2.2 Space-Filling Curves

This chapter discusses discrete space-filling curves (SFC) that lead to a unique enumeration of the elements in the mesh [20]. Due to the fact that a mesh can consist of a multitude of elements, a method to handle mesh management is needed. We use the concept of space-filling curves to address each element with a unique ID. Detailed discussions about different space-filling curves are given in [18, 3, 23]. In this thesis, the Morton SFC for hexahedral elements is discussed. The Morton curve, also called Z-curve because of its shape, was first published in [30] by Lebesgue for quadrilaterals in 2D and hexahedra in 3D. In Figure 2.5 we see some examples of the Morton curve on two quadrilaterals and hexahedra.

The following definition explains a space-filling curve in an analytical way.

Definition 2.2.1. Let $f : I \to \mathbb{E}^n$ be a continuous map, where $\mathbb{E}^n$ is the n-dimensional euclidean space, $n \geq 2$ and $\mathcal{J}_n(im(f)) > 0$, with the $n-$dimensional Jordan-volume $\mathcal{J}_n$ then $im(f)$ is a space-filling curve [37].

In [20] Holke worked on a discretized version of a SFC in order to manage a finite amount of elements of a mesh. Therefore, he defines a SFC index given in the following definition.

Definition 2.2.2. (Definition 3.13 in [20]) A **space-filling curve index** on a refinement space $\mathcal{G} = (\mathcal{S}, \ell, R)$, as determined in Definition 2.1.2, is a map

$$\mathcal{I} : \mathcal{S} \to \mathbb{N}_0 \tag{4}$$

that fulfills the following properties for any $E, E', \hat{E} \in \mathcal{S}$:

(i) The map $\mathcal{I} \times \ell : \mathcal{S} \to \mathbb{N}_0 \times \mathbb{N}_0$ is injective.

(ii) If E is an ancestor of E' then $\mathcal{I}(E) \leq \mathcal{I}(E')$.

(iii) If $\mathcal{I}(E) < \mathcal{I}(\hat{E})$ and $\hat{E}$ is not a descendant of E, then $\mathcal{I}(E) \leq \mathcal{I}(E') < \mathcal{I}(\hat{E})$.

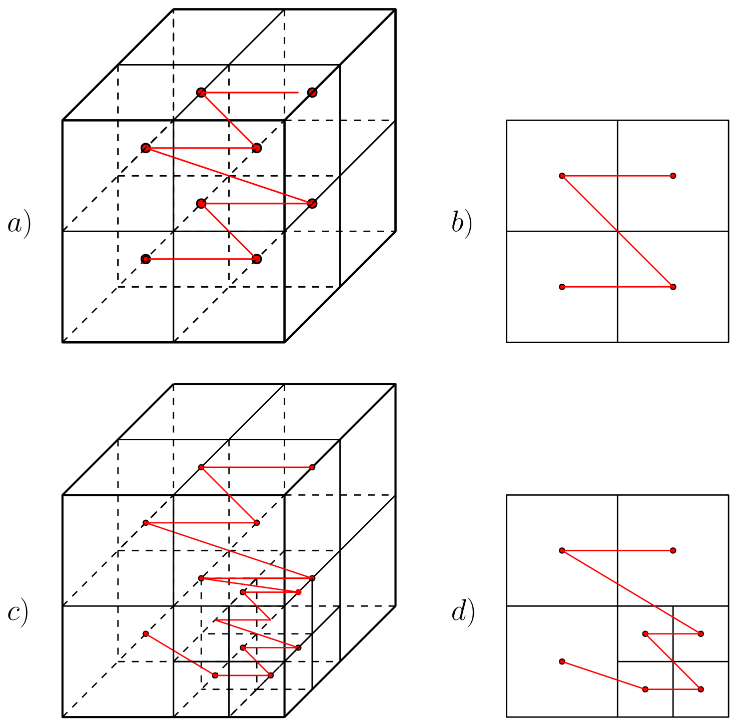


Figure 2.5: This figure shows the Morton curve on two quadrilaterals and hexahedra. The Morton curve in b) equals the front face of the hexahedron in a). The same applies to the Morton curve in d) and the front face in c).

Because of the one-to-one correspondence stated in the first point of Definition 2.2.2, an element of $\mathcal{S}$ can be uniquely identified with its index and level. Furthermore, because of point two, refining an element cannot lead to a smaller SFC index. Additionally, refining an element is a local operation, according to point three.

It is important to emphasize that if $\mathcal{K}$ is a refinement in a refinement space $\mathcal{S}$ with a SFC index $\mathcal{I}$, as determined in Definition 2.2.2, then two different elements of a mesh cannot have the same SFC index:

$$E \neq E' \in \mathcal{K} \Rightarrow \mathcal{I}(E) \neq \mathcal{I}(E') \tag{5}$$

A proof of this statement can be found in [20], Proposition 3.14.
Finally, we can define a discrete space-filling curve as an SFC index restricted to a certain refinement $\mathcal{S}$, $\mathcal{I}|_{\mathcal{S}} : \mathcal{S} \to \mathbb{N}_0$.

To identify an element in an adaptive mesh we only need its **anchor node** and its level. The anchor node of a hexahedral element is the left, front and lower node of the element.

Example 2.2.1. For example, we discuss a hexahedral elements $1:8$ refinement. Let $\mathcal{L}$ be the fixed maximum refinement level. Let $H = [0, 2^{\mathcal{L}}]^3$ be the scaled unit cube. The root element is given by $\mathcal{E} = [0, 2^{\mathcal{L}}]^3$. All further elements in the refinement space are given by refining a hexahedron into eight smaller hexahedra. This refining operation increases the level of the smaller hexahedra by one. The resulting refinement tree is an **octree**. Each vertex in the octree, if it is not a leaf and thus has level $l < \mathcal{L}$, has exactly eight children.
Let $\mathcal{G} = (\mathcal{S}, \ell, R)$ be the corresponding refinement space. We can describe the set

of elements $\mathcal{S}$ in terms of the level and the refinement maps. Starting with the set $\mathcal{S}^0 = \{\mathcal{E}\}$, each set $\mathcal{S}^l$ of level l can be recursively constructed from $\mathcal{S}^{l-1}$ by refining each element with the $1:8$ refinement rule. The union of all $\mathcal{S}^l$ with $0 \leq l \leq \mathcal{L}$ defines the set $\mathcal{S}$ of the refinement space. Thus, let $\mathcal{S}^l$ be the set of all sub hexahedra of $\mathcal{E}$ with side length $2^{\mathcal{L}-l}$. The coordinates of each sub hexahedron are an integer multiple of $2^{\mathcal{L}-l}$. Thus, their anchor node is given by the left, lower, front corner $\mathcal{A} = (r2^{\mathcal{L}-l}, s2^{\mathcal{L}-l}, t2^{\mathcal{L}-l})$ and its level l, with $r,s,t \in \mathbb{N}_0$, $0 \leq r,s,t < 2^l$. A graphical illustration of refinement maps is given in Figure 2.6. In terms of the corner coordinates, we describe the refinement maps R^l in the following way:

$$\begin{aligned}
R^l((r2^{\mathcal{L}-l}, s2^{\mathcal{L}-l}, t2^{\mathcal{L}-l})) = \{&(2r2^{\mathcal{L}-l-1}, 2s2^{\mathcal{L}-l-1}, 2t2^{\mathcal{L}-l-1}),\\
&(2(r+1)2^{\mathcal{L}-l-1}, 2s2^{\mathcal{L}-l-1}, 2t2^{\mathcal{L}-l-1}),\\
&(2r2^{\mathcal{L}-l-1}, 2(s+1)2^{\mathcal{L}-l-1}, 2t2^{\mathcal{L}-l-1}),\\
&(2(r+1)2^{\mathcal{L}-l-1}, 2(s+1)2^{\mathcal{L}-l-1}, 2t2^{\mathcal{L}-l-1}),\\
&(2r2^{\mathcal{L}-l-1}, 2s2^{\mathcal{L}-l-1}, 2(t+1)2^{\mathcal{L}-l-1}),\\
&(2(r+1)2^{\mathcal{L}-l-1}, 2s2^{\mathcal{L}-l-1}, 2(t+1)2^{\mathcal{L}-l-1}),\\
&(2r2^{\mathcal{L}-l-1}, 2(s+1)2^{\mathcal{L}-l-1}, 2(t+1)2^{\mathcal{L}-l-1}),\\
&(2(r+1)2^{\mathcal{L}-l-1}, 2(s+1)2^{\mathcal{L}-l-1}, 2(t+1)2^{\mathcal{L}-l-1})\}
\end{aligned} \tag{6}$$

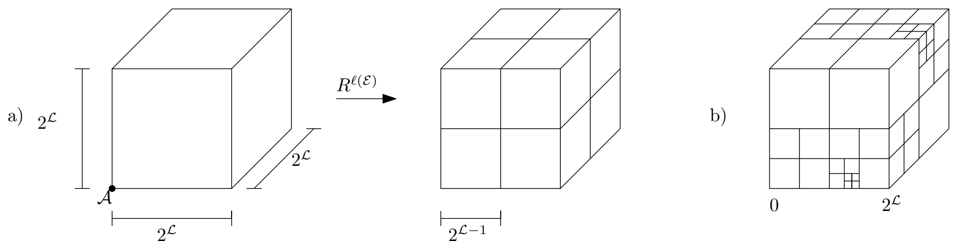


Figure 2.6: In a) on the left, we see the root element $\mathcal{E}$ with equal side length $2^{\mathcal{L}}$ and anchor node $\mathcal{A}$. In a) on the right, we can see the regular refinement archived with the refinement map $R^{\ell(\mathcal{E})}$. It is valid that $R^{\ell(\mathcal{E})} = R^0$ because the root element always has level zero. Illustration b) shows a possible configuration in the refinement space. The coordinates x, y, and z of each anchor node are integer multiples of $2^{\mathcal{L}-l}$ and lie within $[0, 2^{\mathcal{L}}]^3$

2.3 Morton Index For Hexahedral Elements

In [35], G. M. Morton describes the applications of data storage of the Z-curve (aka Morton curve). Defining the Morton index is generally possible in any space dimension n on the n- dimensional hypercube with a $1:2^n$ refinement. This thesis mainly discusses the 3-dimensional case with the $1:8$ refinement as shown in Definition 2.1.4.

An illustration of a space-filling curve index based on the Morton index, due to clarity for quadrilateral elements and thus in the case of a $1:4$ refinement, is given in Figure 2.7 a). Figure 2.7 b) shows the according representation by a refinement tree.

The following definition determines the Morton index.

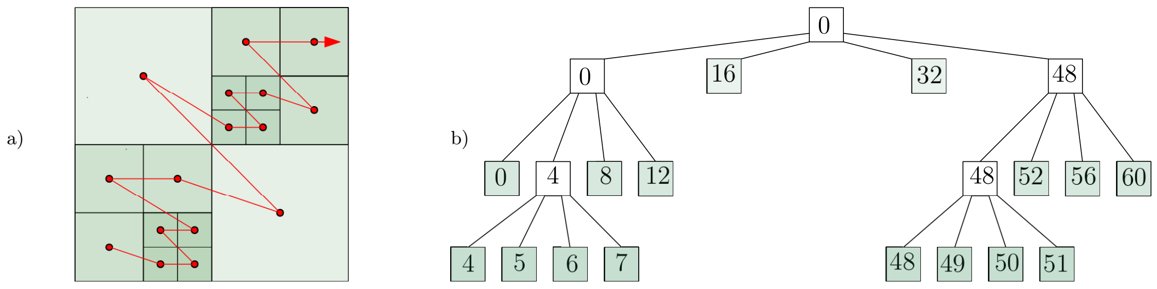


Figure 2.7: In a), we see the Morton SFC on a quadrilateral starting at the lower left element. In b), we see the corresponding refinement tree with the assigned Morton indexes. The root element always has a Morton index of 0. Every element is a leaf and is depicted in green. The parent elements are depicted in white.

Definition 2.3.1. Let H be a hexahedron with level $l < \mathcal{L}$ and anchor node $\mathcal{A} = (x, y, z) \in [0, 2^{\mathcal{L}}]^3 \cap \mathbb{N}_0$. The binary representation of x, y and z is given by:

$$x = \sum_{j=0}^{\mathcal{L}-1} x_j 2^j, \; y = \sum_{j=0}^{\mathcal{L}-1} y_j 2^j \text{ and } z = \sum_{j=0}^{\mathcal{L}-1} z_j 2^j. \tag{7}$$

With X, Y and Z we define the $\mathcal{L}$-tuples containing the binary digits of x, y and z respectively:

$$\begin{aligned} X &= X(H) = (x_{\mathcal{L}-1} x_{\mathcal{L}-2} \dots x_0), \\ Y &= Y(H) = (y_{\mathcal{L}-1} y_{\mathcal{L}-2} \dots y_0), \\ Z &= Z(H) = (z_{\mathcal{L}-1} z_{\mathcal{L}-2} \dots z_0), \end{aligned} \tag{8}$$

The following theorem states how to compute the Morton index of a hexahedron.

Theorem 2.1. Let $\mathcal{G} = (\mathcal{S}, l, R)$ be a hexahedral refinement space. Let $m : \mathcal{S} \to \mathbb{N}_0$ be the Morton index of a hexahedron $H \in \mathcal{S}$. $m(H)$ is defined by the bit-wise interleaving of the $\mathcal{L}$-tuples Z, Y and X:

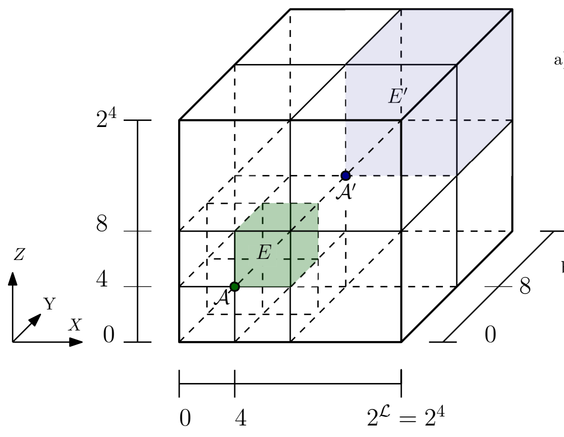
$$m(H) := Z \dot{\perp} Y \dot{\perp} X = (z_{\mathcal{L}-1} y_{\mathcal{L}-1} x_{\mathcal{L}-1} z_{\mathcal{L}-2} y_{\mathcal{L}-2} x_{\mathcal{L}-2} \dots z_0 y_0 x_0) \in [0, 2^{3\mathcal{L}}] \tag{9}$$

Because of bit-wise interleaving, computing the Morton index of an element is possible in constant time. As only the anchor node and level of an element are stored, the Morton index is very memory efficient. It is worth mentioning that the usage of the Morton index is not restricted to AMR applications. In [11], for example, the Morton index is used for 2D image encryption.

Figure 2.8 illustrates an exemplary computation of the Morton index. Once, the Morton index is computed based on bit-wise interleaving and in an alternative way using the corresponding refinement tree.

2.4 A SFC Index For Forests

In this chapter, we handle the topic of forests. We allow meshes to consist of multiple different refinement trees. The sum of all refinement trees in a mesh is called a forest. In 2.4.1, the definition of a forest is given. For a more detailed discussion about forests of adaptive meshes, see, for example, [4, 20].



a) $\mathcal{A}' = (x, y, z) = (8, 8, 8)_{10} = (1000, 1000, 1000)_2$

$\Rightarrow m(E') = (111000000000)_2 = (3584)_{10}$

$Alternative: m(E') = (7000)_8 = (3584)_{10}$

b) $\mathcal{A} = (x, y, z) = (4, 0, 4)_{10} = (0100, 0000, 0100)_2$

$\Rightarrow m(E) = (000101000000)_2 = (320)_{10}$

$Alternative: m(E) = (0500)_8 = (320)_{10}$

Figure 2.8: On the left, we see the underlying refined hexahedron. Example a) shows the calculation of the Morton index of E' with level 1. Example b) shows the calculation of the Morton index of E with level 2. In a) and b), two different methods of calculating the Morton index are given. First, the method defined in Definition 2.3.1 is shown. Secondly, the alternative method, with the help of the appropriate refinement tree, is presented. There, the octal system is used because of the underlying octree.

Definition 2.4.1. Let $\{\mathcal{S}_0, \ldots, \mathcal{S}_{S-1}\}$ be refinements of respective refinement spaces $\mathcal{G}_0, \ldots, \mathcal{G}_{S-1}$. Then, the **forest** $\mathcal{F}$ with trees $\{\mathcal{S}_s\}_{s<S}$ is the set of all leaves of the individual refinements paired with their tree number s:

$$\mathcal{F} := \bigcup_{s=0}^{S-1} \{s\} \times \mathcal{S}_s \tag{10}$$

The elements of $\mathcal{F}$ are the **leaves** of the forest.

It is worth mentioning that the element shapes of the refinement spaces $\{\mathcal{G}_0, \ldots, \mathcal{G}_{S-1}\}$ can differ from each other. For example, one tree builds a mesh with tetrahedral elements, and another tree of the forest builds a mesh with hexahedral elements. Then, this forest describes a hybrid mesh. It can be helpful to create hybrid meshes in order to approximate complex geometries more precisely. Generally, we define a maximum refinement level $\mathcal{L} \in \mathbb{N}$ for all elements in a refinement $\mathcal{S}$. Then it holds for all elements E that $\ell(E) \leq \mathcal{L}$.

It is apparent that if a maximum refinement level $\mathcal{L}$ exists, it holds $R^{\mathcal{L}}(E) = \emptyset$ for all $E \in \mathcal{S}$ and thus $\mathcal{S}^l = \emptyset$ for all $l > \mathcal{L}$.

To obtain the unique identification of each element in a forest, and not only in one single tree, the Morton SFC index must be extended. This extension is described in the following definition.

Definition 2.4.2. If for a forest $\mathcal{F}$ each refinement space $\mathcal{S}_s$ has an SFC index $\{\mathcal{I}_s\}$, then we extend these to an index $\mathcal{I}$ on the leaves of $\mathcal{F}$ by

$$\begin{aligned} \mathcal{I} &: \mathcal{F} \to \{0, \ldots, S-1\} \times \mathbb{N}_0 \\ (s, E) &\mapsto (s, \mathcal{I}_s(E)) \end{aligned} \tag{11}$$

with the order

$$(s, I) < (s', I') :\Leftrightarrow s < s' \text{ or } (s = s' \text{ and } I < I') \tag{12}$$

on $\{0,\ldots,S-1\}\times\mathbb{N}_0$, which extends the individual SFC orders across the trees. By extension of notation we call $\mathcal{I}$ an SFC index of forest $\mathcal{F}$. (Definition 3.20 [20])

However, the SFC in a forest has discontinuities when jumping from one tree to another. Moreover, if the forest holds a hybrid mesh, the connection of different trees interrupts the recursive structure of the SFC. Furthermore, this leads to challenges regarding identifying neighbor elements over tree boundaries. In Chapter 5 the problem of finding neighbors in one tree and inside transition cells is discussed in detail. In [20], Holke discusses the neighbor-relations of elements over tree boundaries in Chapter 6.

3 Transitioning

In this chapter a transition scheme to resolve hanging nodes on faces in a hexahedral adaptive mesh is presented. First, in Chapter 3.1, we introduce hanging nodes on faces in hexahedral meshes. After that, we discuss some possible approaches handling these hanging nodes. One possibility is to insert transition cells. This approach is the main focus in this thesis and is presented in Chapter 3.3. Afterward, the SFC index for transition cells is introduced in Chapter 3.4, and the extension to transitioned forests is given in Chapter 3.5.

3.1 Hanging Nodes On Hexahedral Faces

The following discusses which problems occur when two face-neighbored elements have different levels. We consider only face-balanced meshes. Consequently, the level of face-neighboring elements can only differ at most by ± 1. This chapter focuses on hanging nodes on faces while in Outlook, Chapter 7, the problem of hanging nodes on edges is discussed in detail.

Definition 3.1.1. Let $\mathcal{G} = (\mathcal{S}, \ell, R)$ be refinement space. Let $E, E' \in \mathcal{S}$ and f be a face of E and f' be a face of E'.

- E is called **face-neighbor** of E' at face f, if
  $$E \cap E' = f' \tag{13}$$
  is a sub-face of face f.
- If E is a face-neighbor of E', then E' is a face-neighbor of E and vice versa.
- E is called **hanging**, if it has at least one face f such that E has more than one face-neighbor at f.
- Let $\mathcal{N}_f = \{N_0, \ldots, N_k\}$ with $N_0, \ldots, N_k \in \mathcal{S}$ be the set that denotes all face-neighbors of E at f with $|\mathcal{N}_f| > 1$ and
  $$f = E \cap \Big(\bigcup_{N_i \in \mathcal{N}_f} N_i \Big), \tag{14}$$
  then f is a **hanging face** of E.
- Let $\mathcal{N}_e = \{N_0, \ldots, N_k\}$ with $N_0, \ldots N_k \in \mathcal{S}$ be the set that denotes all edge-neighbors of E at e with $|\mathcal{N}_e| > 1$ and
  $$e = E \cap \Big(\bigcup_{N_i \in \mathcal{N}_e} N_i \Big), \tag{15}$$
  then e is a **hanging edge** of E.

Therefore, we say that an element has no hanging nodes, if no nodes exist in their face centers. Consequently, if an element's corner node is a hanging node from the perspective of another element, we do not consider this node a hanging node. In Figure 3.1, hanging nodes of quadrilateral and hexahedral elements are illustrated in detail.

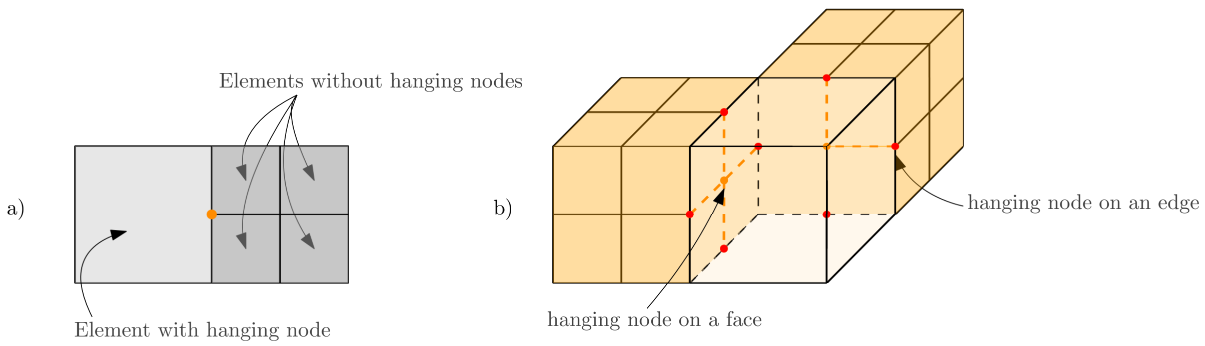


Figure 3.1: Illustration a) shows the quadrilateral case with one hanging node, presented in orange. Illustration b) presents the hexahedral case with, in total, two hanging nodes on faces (orange) and seven hanging nodes on edges (red).

The irregular index k denotes the maximum difference of refinement levels between face neighbored elements in the mesh and, consequently, the maximum number of hanging nodes [1]. In this thesis, we only consider 1-irregular meshes as a consequence of discussing balanced meshes. It is worth mentioning that all nodes that lie on the boundary of the mesh cannot be hanging.

Definition 3.1.2. A hexahedral element mesh is **conformal** if two distinct, non-disjoint elements intersect at common nodes, edges or faces only. (Definition 2 in [39])

Thus, a mesh is conformal if it does not contain any hanging nodes.

Definition 3.1.3. If f is a hanging face of an element E and $\mathcal{N}_f$ its set of face-neighbors with $|\mathcal{N}_f| > 1$, then we say that

$$\mathcal{H}_{E,f} := E \cap \left(\bigcup_{N_i, N_j \in \mathcal{N}_f, i \neq j} (N_i \cap N_j) \right) \tag{16}$$

is the **hanging set** of E at f. If f is no hanging face of E, then we define $\mathcal{H}_{E,f}$ as the empty set, $\mathcal{H}_{E,f} := \emptyset$. (Definition. 3.1.3 in [5])

Hence, $\mathcal{H}_{E,f}$ is a set containing vertices and edges in 3D. An exemplary set $\mathcal{H}_{E,f}$ can be found in Figure 3.1 in a) in orange and in b) in orange and red.

Especially for the case of a face-balanced mesh, it's worth mentioning that $\mathcal{H}_{E,f}$ does not always need to contain vertices and edges. There, it can also be the case, that $\mathcal{H}_{E,f}$ only contains vertices. These cases of hanging nodes on edges are discussed in Chapter 7. In the following, we assume that if $\mathcal{H}_{E,f}$ is not empty, it always contains vertices and edges.

3.2 What Do Other People Do To Solve Hanging Nodes

There are different ways to handle hanging nodes. Generally, there are two different overall approaches. Whether adapting the numerical solver or transitioning the mesh, the fundamental decision needs to be made. First, some approaches to adapting the numerical solver are presented.

One possible way is to adjust the shape functions in the sense of *constrained approximation* to ensure continuity. There, the hanging nodes are expressed in terms of their bigger face-neighbor. This approach is prevalent and is discussed in detail in [36, 13]. An algorithm using constrained shape functions can be found in [43].

An alternative approach is to not insert refined elements in the mesh but refine the region of interest by *superposing* it with a finer layer. This is an entirely different approach compared to the conventional approach of adaptive mesh refinement. In [50], the theory of this approach for 3D elements is given.

In the following, different techniques that modify the underlying mesh are presented. One possible method is the template-based approach among others shown by Schneiders in [39]. Schneiders presents transition patterns for a $1:9$ quadrilateral refinement in 2D and a 1:12 hexahedral refinement in 3D.

In [5], Becker derived another type of transition pattern for a $1:4$ quadrilateral refinement out of Schneiders idea and implemented it in `t8code`. An exemplary mesh consisting of transition cells presented by Becker can be found in Figure 3.2. The transition cells shown by Becker consist of triangles. These from Schneiders consist of regular and non-regular quadrilaterals. What Becker's and Schneiders' approach have in common is that they both insert additional nodes. In [24], they present transition patterns for a 2D quadrilateral mesh without inserting additional nodes. Evidently, these transition patterns contain different kinds of elements, such as regular and non-regular triangles and quadrilaterals.

Removing hanging nodes in a $1:8$ hexahedral mesh proves to be a more difficult task. Each of the six square faces of a hexahedron can contain hanging nodes. Furthermore, each of the twelve edges can contain either hanging nodes or not. This results in $\sum_{i=1}^{6}\binom{6}{i}+\sum_{i=1}^{12}\binom{12}{i}=4157$ different cases that can occur. However, with the removal of equivalent classes (due to rotation), 325 different cases remain. In [24], they also present templates for the 3D case without inserting additional nodes. However, one case exists where no template can be found without inserting an additional node.

Templates exist that solve the hanging-node problem completely. For example, in [32], over 325 templates are presented. These transition cells can consist of prisms, pyramids, hexahedra, or tetrahedra. In 260 of the 325, no additional nodes were inserted, and in the rest of the 65 patterns, a central node inside the transition pattern was inserted. This immense selection of different element types and the massive amount of different templates can lead to an overload of implementation and decrease the mesh quality.

In this thesis transition templates for a $1:8$ hexahedral refinement, derived from the transition patterns from Becker, will be presented. These transition patterns do not resolve hanging nodes completely. Nevertheless, after inserting the transition templates into the mesh there are no more hanging nodes on faces anymore. The transition cells presented in this thesis merely consist of pyramids, and there are "only" 63 different templates used, which is rather a difference compared to 325 as described above.

3.3 Transition Cells

We call a mesh with inserted transition templates a *transitioned* mesh. This thesis considers balanced meshes concerning faces and, hence, face-neighbored elements

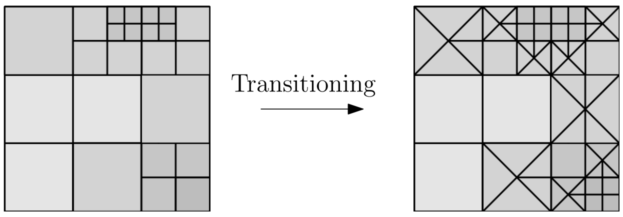


Figure 3.2: On the left, a non-conforming can be seen. The right side shows the mesh with inserted transition templates, introduced by Becker in [5]. Each transition cell adds a central node.

with hanging nodes on faces. Therefore, we end up with $2^6 = 64$ different transition cells. Figure 3.3 shows each equivalent class of the transition templates. This also includes the case, where each face of an element contains hanging nodes. In this particular case, no transition cell will be inserted due to the more extensive set of elements and one additional node in a transition cell compared to a regular refined hexahedron. Figure A.1 in Appendix A shows a complete list of all possible transition cells.

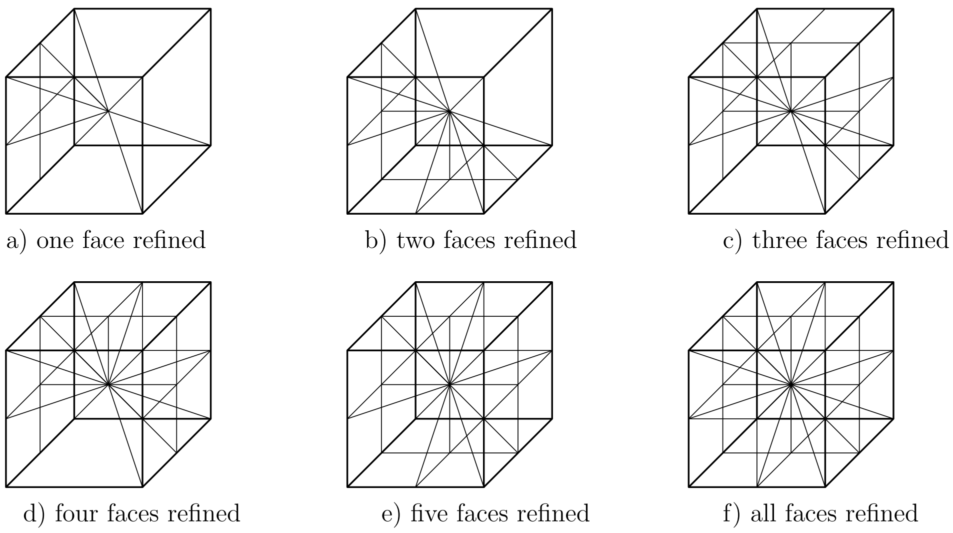


Figure 3.3: This figure shows the different equivalent classes of transition cells with hanging nodes on faces.

Definition 3.3.1. Let $E = [\vec{v_0}, \vec{v_1}, \vec{v_2}, \vec{v_3}, \vec{v_4}, \vec{v_5}, \vec{v_6}, \vec{v_7}]$ be a regular hexahedron. The transition cell T of E consists of a subset of set S consisting of 30 different pyramid shaped **subelement children**. The top of each pyramidal subelement is in the center, $\vec{v_{07}}$, of E. The square base sides of each pyramid are either given by the faces $f_0, \ldots, f_5$ of E, or by the corresponding refined square faces, that cause the hanging nodes.
Each transition cell of E can be constructed from this pyramid shaped subelements.

One subelement can be described by three indices i,j and k. The first index $i \in \{0,1,2,3,4,5\}$ equals the face of the hexahedron where the base side of the pyramid lies. The index $j \in \{0,1,2,3\}$ enumerates the pyramids of the hexahedral face f_i. If the face is split, j provides information about which pyramid of the four is possible according to the coordinate system given in 2.4. The index $k \in \{0,1\}$ thereby denotes whether the subelement is split ($k = 1$) or not ($k = 0$). In Figure 3.4, two subelements with their according describing indices inside a transition cell are shown.

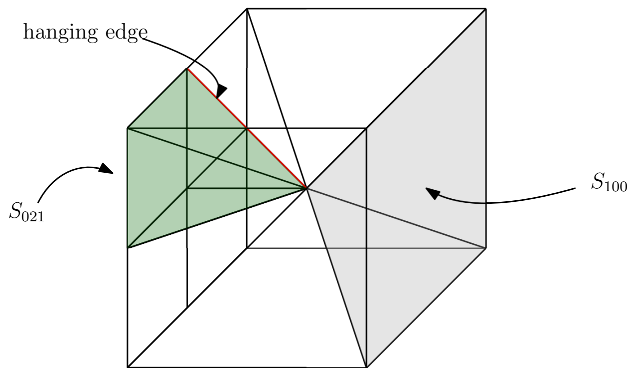


Figure 3.4: This figure shows a split subelement S_{021} in green on hexahedral face f_0 and a non-split subelement S_{100} in gray on hexahedral face f_1. One (of four) occurring hanging edge inside the transition is shown in red.

In Definition 2.1.1 a legal discretization is defined. Thus, when transitioning a mesh, we have to make sure that inserting the transition cells, presented in Figure 3.3, does not violate these conditions. Therefore, we have to ensure that a transition cell T of an element E fulfills the following properties:

(i) $E = \bigcup_{S_{ijk} \in T} S_{ijk}$

(ii) $S_{ijk} \neq S_{lmn} \ \forall \ S_{ijk}, S_{lmn} \in T$ with $ijk \neq lmn$

(iii) if $S_{ij0} \in T \Leftrightarrow S_{ij1} \notin T$

Regarding (i): Considering the construction of T as determined in Definition 3.3.1, condition (i) is trivially fulfilled. Condition (ii) is also trivial due to the construction of T, and condition (iii) is obviously valid because a subelement can not be split and not split at the same time.

Each transition cell is assigned to exactly one **transition type** t. The transition type is a six-digit binary number that defines how the transition cell is composed. Thereby denotes t_i with $0 \leq i \leq 5$ the i-th digit of t. If t_i equals 1, the face f_i of the corresponding hexahedral element E is a hanging face. Thus, there are four split pyramidal subelements with the base side on face f_i. The different equivalent classes of transition types are also shown in Figure 3.3.

Definition 3.3.2. A **hexahedral refinement map with transition cells** is a map $R_t^l(E) : S^l \to \mathcal{P}(S^{l+1})$ that maps an element $E \in \mathcal{S}$, with $\mathcal{S}$ as the set of elements of the underlying refinement space, of level l to a transition cell of type t with $0 \leq t \leq 63$. If $t = 0$, $R_t^l = R^l$,as in Definition 2.1.2, R_t^l fulfills the following properties:

(i) $R_t^l(E) = T$ if $t > 0$ and $R_t^l(E) = C$ if $t = 0$ and

(ii) if E is a subelement, then $R_t^l(E) = \emptyset$ for all t.

(c.f. Definition 3.3.4 in [5])

It is essential to say that because of property (ii), a subelement can not be refined any further. With the given refinement map, we are now able to define a refinement space for transitioned meshes.

Definition 3.3.3. The **hexahedral refinement space for transitioned meshes** $\hat{\mathcal{G}}$ is given by:

- The set $\mathcal{S} = \{E | E$ is a descendant of $\mathcal{E}$ with $0 \leq \ell(E) \leq \mathcal{L}\}$ of elements, that are either hexahedral elements or pyramidal subelements. $\mathcal{E}$ thereby denotes the root element.
- The level map $\ell : \mathcal{S} \to \mathbb{N}_0$, defined as the distance of E to the root element $\mathcal{E}$.
- The refinement maps $\mathcal{R} = \{R_t^l | l \in \mathbb{N}_0$ and $0 \leq t \leq 63\}$, where t denotes the transition type.

3.4 SFC In Transition Cells

Inserting transition cells brings irregularities. Therefore, one remaining task is to adjust the SFC index according to a transitioned mesh. The goal is to modify the Morton index in such a way that we achieve a unique enumeration of the elements and subelements. If element E is hanging and thus will be replaced by the corresponding transition cell T, the subelements of T need to follow a specific enumeration. Therefore, we define the subelement index in the following definition.

Definition 3.4.1. Let $\mathcal{S}$ be the set of elements of a refinement space $\hat{\mathcal{G}}$ as described in Definition 3.3.3. The **subelement index** is a mapping $s : \mathcal{S} \to \mathbb{N}_0$ with the following properties:

- If $\{S\}$ is the set of subelement of a transition cell T, then $s : \{S\} \to \{0, \ldots |T| - 1\}$ is bijective.
- If E is a no subelement, then $s(E) := 0$.

The subelement index is based on the orientation given by the coordinate system in Figure 2.4. Therefore, if S_{ijk} is a subelement of a transition cell T, the subelement index of S_{ijk} can be computed in the following way:

$$s(S_{ijk}) = \sum_{l=0}^{i} \Big(\chi_{l \neq i} \cdot \ (4\chi_{S_{01l} \in T} \ + \ \chi_{S_{00l} \in T})\Big) + \ j, \tag{17}$$

where χ denotes the indicator function. The calculation can be explained as follows: We have to calculate how many subelements exist in the transition cell *before* S_{ijk} itself. *Before* is meant in the context of the given orientation, in this case, from left to right, then from front to back and at least from the bottom to top. If $\chi_{l \neq i} = 1$ we know that we must sum up the elements on face l. It can either be the case that the face l is split, then $\chi_{S_{01l} \in T} = 1$ and $\chi_{S_{00l} \in T} = 0$ and four subelements have to be taken into account. Otherwise, face l is not split, and thus, there is one subelement with base side on face l, and thus, $\chi_{S_{01l} \in T} = 0$ and $\chi_{S_{00l} \in T} = 1$.

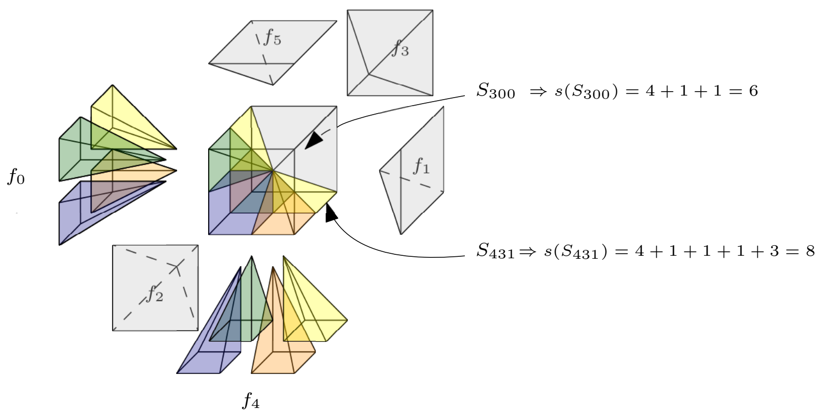

f_5
f_3
$S_{300} \Rightarrow s(S_{300}) = 4+1+1 = 6$
f_0
f_1
$S_{431} \Rightarrow s(S_{431}) = 4+1+1+1+3 = 8$
f_2
f_4

Figure 3.5: This figure shows an expanded view of a transition cell T with transition type $t = (100010)_2 = 34$. Exemplary two different subelement indices are computed. Subelement S_{431} is the third ($j = 3$) split subelement ($k = 1$) with base side on hexahedral face f_4 ($i = 4$). Due to t, we sum up all elements up to face f_3 (inclusive) and then add $j = 3$. Hence, S_{431} is the eighth element in T. The calculation follows analogously for subelement S_{300} that is non-split ($j = 0$ and $k = 0$) with the base side on the hexahedral face f_3 ($i = 3$). The color scheme of the split subelements is based on the underlying orientation.

The index j is only not equal to zero, if the face is split and hence only takes into account if $k = 1$. For clarification, see an example of an expanded view of a transition cell with transition type $100010 = 34$ in Figure 3.5.

The following required step is to define the Morton index for subelements. The key concept here is that if the element E got transitioned into a transition cell T, the Morton index of all subelements S of T is the same as the Morton index of E. That means if $S \in R_t^l(E)$ is a subelement out of a transition cell T with parent element E. Then with the Morton index m it applies that

$$m(S) = m(E). \tag{18}$$

It is important to keep in mind that if an element E gets transitioned, the anchor node of E keeps unchanged. Furthermore, it is valid that if two elements E, E' have the same Morton index, they are both subelements in the same transition cell. That means that we can not identify a subelement uniquely by the Morton index and level. Additionally we need the subelement index computed in equation (17).

Figure 3.6 shows exemplary how the Morton curve looks in a transition cell with transition type $t = 32$.

3.5 A SFC Index For Transitioned Forests

The property of the uniqueness of the Morton index, introduced in 2.3.1, is not fulfilled in transitioned forests because each subelement in a family has the same Morton index. Therefore, it is necessary to use the subelement index to obtain a global unique encoding of elements in a transitioned forest. The following definition describes the Morton index for transitioned forests.

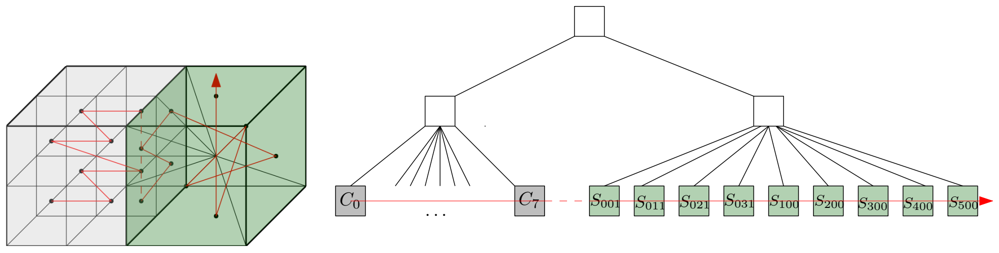


Figure 3.6: This figure shows the Morton curve inside a transition cell on the left and the corresponding refinement tree on the right. For clarity, the Morton curve does not go through the midpoint of each subelement but through the midpoint of each base side. Furthermore, for illustrative reasons, not every element is depicted in the refinement tree on the right.

Definition 3.5.1. Let $\{m_k\}_{k<K}$ be the Morton indices of refinement spaces $\{\mathcal{G}_0, \dots, \mathcal{G}_{K-1}\}$ within a forest $\mathcal{F}$. The Morton index for transitioned forests $\mathcal{I}_t$ is given by:

$$\begin{aligned} &\mathcal{I}_t : \mathcal{F} \to \{0, \dots, K-1\} \times \mathbb{N}_0 \times \mathbb{N}_0 \\ &(k, E) \mapsto (k, m_k(E), s_k(E)) \end{aligned} \tag{19}$$

with the order:

$$\begin{aligned} (k,m,s) < (k',m',s') \Leftrightarrow & k < k' \vee \\ & (k = k' \wedge m < m') \vee \\ & (k = k' \wedge m = m' \wedge s < s') \end{aligned} \tag{20}$$

on $\{0, \dots, K-1\} \times \mathbb{N}_0 \times \mathbb{N}_0$. This procedure extends the SFC index across transition cells. $\mathcal{I}_t$ is then called the **SFC index of the transitioned forest $\mathcal{F}$**

Proposition 3.5.1. Let $\mathcal{F}$ be a transitioned forest with SFC index $\mathcal{I}_t$ from Definition 3.5.1. Let N be a finite number of leaf elements in $\mathcal{F}$. Then, there exists a unique bijective map

$$\mathcal{I}_{t\mathcal{F}} : \mathcal{F} \to \{0, \dots, N-1\} \tag{21}$$

that is monotonous under $\mathcal{I}_t$, thus

$$(k, \mathcal{I}_t(k,E), s_k(E)) < (k', \mathcal{I}_t(k',E'), s_{k'}(E')) \Leftrightarrow \mathcal{I}_{t\mathcal{F}}(k,E) < \mathcal{I}_{t\mathcal{F}}(k',E') \tag{22}$$

([5])

A proof can be found in [5], Proposition 3.4.1.

4 Implementations In `t8code`

As mentioned in the Introduction of this thesis, we present an implementation of a transition scheme in the open-source software `t8code`. In this chapter, we give a brief overview of the `t8code` library. The algorithms of `t8code` are subdivided into high- and low-level algorithms. High-level algorithms are mesh-specific algorithms. That means they operate on the whole mesh, such as `new`, `adapt`, `balance`, and `partition` presented in the next chapters. Low-level algorithms, however, are element-specific algorithms. Every low-level algorithm is labeled with the prefix `t8_element`. This means that they operate on a specific element in the mesh. One advantage of `t8code` is that the high-level algorithms are detached from the low-level algorithms and vice versa.

4.1 High-level Algorithms Of `t8code`

In the following the high-level algorithms `new`, `adapt`, `balance`, `partition` are presented.

4.1.1 `New`

The starting point is always an initial uniform refinement of a mesh, the *coarse mesh*. This initial mesh can be computed via the `new` algorithm. This mesh contains all of the root elements of a forest. For the `new` routine, it is necessary to specify an initial level that determines the size of each root element.

4.1.2 `Adapt`

The `adapt` algorithm is the key-algorithm of `t8code`. In order to refine and coarsen different areas of a mesh, the `adapt` algorithm is used regarding a given refinement criterion. According to the SFC, the `adapt` routine iterates over every mesh element. The refinement criterion then decides whether the element should stay unchanged, be refined to the next higher level, or be coarsened to the next lower level. This refinement rule can be represented with the following refine function φ where $\mathcal{S}$ is the set of elements and $E \in \mathcal{S}$:

$$\begin{aligned} \varphi : \mathcal{S} &\to \mathbb{Z} \\ E &\mapsto \{-1, 0, 1\} \end{aligned} \tag{23}$$

The interpretation of the values $-1, 0$ and 1 is given as follows:

$$\varphi(E) = \begin{cases} -1 & E \text{ and its siblings should be coarsened to its parent} \\ 0 & E \text{ should stay unchanged} \\ 1 & E \text{ should be regularly refined into its children} \end{cases} \tag{24}$$

In the case of $\varphi(E) = -1$, the element and its n siblings, in terms of SFC so-called *successors*, have to be coarsened to their parent element. Thus, a whole family, which is the element itself and its siblings, will be coarsened into their parent element. In order to get the parent of an element, the function `t8_element_parent` is implemented. This example of a low-level function is discussed in detail in chapter 4.2. In applications, it is possible that different refinement rules are applied after another. In this context it is essential to note that the refinement rule is a user-provided function.

4.1.3 Balance

t8code offers the opportunity to *balance* the mesh with a 2:1 face-balance condition via the balance algorithm. That means that all face-neighbor levels differ at most by ± 1, see Chapter 2. It follows that in balanced meshes the number of face-neighbors is limited. In the case of a hexahedral balanced mesh, the number of possible face-neighbors is at most four. Hence, the number of possible occurring hanging faces is also restricted. Nevertheless, it is essential to mention that there are also applications that manage an arbitrary number of hanging nodes at elements with a differing refinement level greater than one, see [17] for example.

A balanced mesh can be beneficial for numerical applications because it simplifies the interpolation schemes and reduces the number of neighboring processes. The refinement criterion presented in chapter 4.1.2 is based on error estimation and/or geometric constraints. Therefore, the resulting meshes may not fulfill the balance condition. For this reason, the core algorithms t8_adapt and t8_balance are decoupled in t8code. First, the mesh is adapted, and afterwards, it is up to the user to call the balance algorithm. Thus, it depends on the application if a balanced mesh is valuable or not.

The input of balance is an arbitrary forest. The output is a forest that fulfills the balance property. To detect the level differences between face-neighbors, t8_balance uses similar low-level algorithms as t8_adapt. Hence, finding neighbors in a forest is a key feature of the balance algorithm. It is important to note that t8_balance never coarsens any elements to preserve the desired accuracy.

t8_balance iterates through every element of the starting forest $\mathcal{F}_0$ and verifies whether face-neighbors have a larger level that differs more than 1. If so, the children of the elements children are added into a new forest $\mathcal{F}_{i+1}$, which is created in every iteration. If not, the element itself is added into $\mathcal{F}_{i+1}$. These refinement steps are repeated until the current forest does not change anymore. The final forest is denoted by $\hat{\mathcal{F}}$. The pseudo-code of t8_balance is given in Algorithm 8.2.1 in [20]. Furthermore, the proof that the algorithm terminates and produces a balanced forest is given in Proposition 8.1 in [20]. In Figure 4.1, you can see an example of t8_balance on a hexahedral mesh.

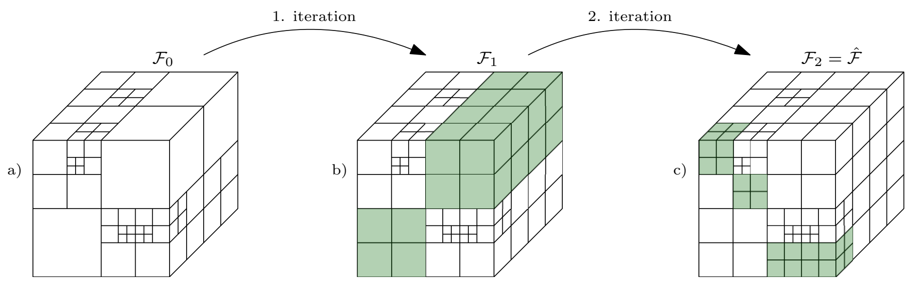


Figure 4.1: This figure shows two iterations of t8_balance. In a) and b), the mesh is unbalanced. The mesh shown in c) fulfills the face-balance condition. The elements that are refined in each iteration step are depicted in green.

4.1.4 `Partition`

`t8code` allows the distribution of the forest among multiple processes to ensure applications' scalability. To avoid an unbalanced distribution, `partition` spreads the forest elements evenly over the processes. An evenly distributed workload for each process is the goal of `partition`. In [9], the partition of a coarse mesh is discussed in detail. The following definition shows how `partition` divides N elements over P processes:

Definition 4.1.1. Let $\mathcal{F}$ be a forest with N leaves. Let $\mathcal{I}_\mathcal{F} \in \{0,\dots,N-1\}$ be the consecutive Morton index. Then the process $i \in \{0 \le i < P\}$ has the elements $\mathcal{F}(i)$:

$$\mathcal{F}(i) := \left\{(k,E) \in \mathcal{F} \;\middle|\; \left\lfloor \frac{N \cdot i}{P} \right\rfloor \le \mathcal{I}_\mathcal{F}(k,E) < \left\lfloor \frac{N \cdot (i+1)}{P} \right\rfloor \right\} \tag{25}$$

Thus, the SFC index induces a straight-forward allocation to load-balance the leaves of a tree across multiple processes.

Note that after adapting a forest the amount of elements changes and thus may not be partitioned equally anymore. Then, it might be necessary to migrate elements from one process to another. This procedure is called re-partitioning. In [20], Holke presents numerous numerical results. In [8, 22], the scalability of `t8code` is analyzed. The authors validated the scalability of thousands of processes of supercomputers such as, among others, JUQUEEN consisting of $28,675$ compute nodes, each with 16 IBM PowerPC-A2 cores at 1.6 GHz at the research center Jülich, Germany.

The last high-level algorithm of `t8code` is the `ghost` algorithm. `t8_ghost` enables the exchange of information about the boundary elements of different processes. `t8_ghost` will not be discussed further here because, at this time, transitioning forests on multiple processes is a future project and has not been implemented yet. The reader is referred to [20] where `t8_ghost` is discussed in detail in Section 7.

In Chapter 4.3, a new high-level algorithm `t8_transition` is presented.

4.2 Fundamentals For Transition Cells In `t8code`

In this chapter, some fundamentals of transition cells used by low-level algorithms to enable the high-level transitioning algorithm are presented. A selection of low-level algorithms, showing key principles of `t8code`, are shown here. Each low-level algorithm in `t8code` is labeled with the prefix `t8_element`. In Chapter 4.2.1, the binary encoding of transition cells is presented. After that, in Chapter 4.2.2, the subelement ID type is introduced. Finally, in Chapter 4.2.3, adjustments of the element structure are presented.

4.2.1 Transition Type

In order to remove hanging faces in a non-conformal hexahedral mesh, they must first be identified. Each element with hanging faces will be refined into the corresponding transition cell, consisting of a set of pyramids. These pyramids are called subelements. It is important to mention, that a subelement can not be refined again. Therefore, if, however, a transition cell is refined according to the refinement rule, it will be coarsened to its parent first. That means that all subelements will be removed in this operation. Then the parent element will be refined into its children. The function `t8_element_parent` maps an element to its parent. The reverse routine `t8_element_children` maps an element to its children. Consequently, a transition cell can never be a parent element. If `t8_element_children` is called with a subelement, it always returns the children of its parent element, thus regular hexahedral elements. In Figure 4.2, the functionalities of `t8_element_parent` and `t8_element_children` with a regular refinement and a transition cell are illustrated.

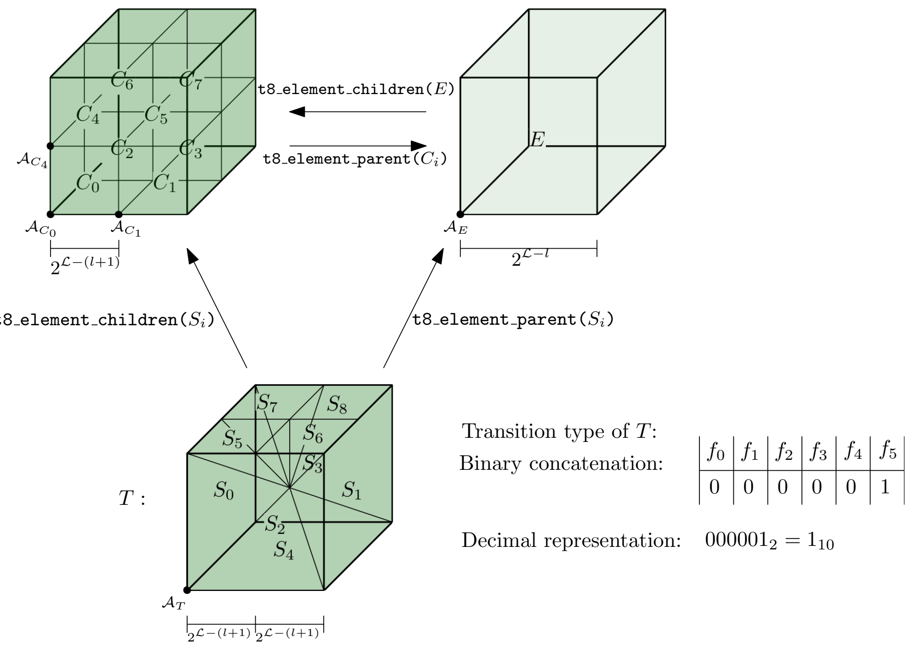


Figure 4.2: This figure shows how the functions `t8_element_children` and `t8_element_parent` work with regular refinement and transition cells. For reasons of clarity only two anchor nodes of the regular children C_i are depicted. Note that T only has one anchor node.

Definition 4.2.1. Let E be a 3D hexahedral element and $b_0, b_1, b_2, b_3, b_4, b_5 \in \{0,1\}$. Let $b_i, 0 \leq i \leq 5$ represent the boolean parameter, stating whether the face f_i of E is hanging or not. Then the binary concatenation $(b_0b_1b_2b_3b_4b_5)_2$ is called the **binary transition type** of E. We often refer to the decimal representation of this binary encoding in the following.

Therefore, there are $2^6 = 64$ different transition types possible. A selection of different transition types is shown in Figure 3.3. It is important to mention that the transition types $(111111)_2 = 63$ (shown in Figure 3.3 f)) and $(000000)_2 = 0$ are special cases. If all faces of a hexahedron are hanging we do not insert a transition cell. We refine the element with the regular $1:8$ hexahedral refinement. If the transition type equals zero, we do not insert any transition cell because the element is not hanging.

Hence, all geometrical properties that specify the transition cell can be derived from the transition type. To be more precise, the transition type implicitly determines every vertex, edge and face of the transition cell. Thus, the transition type determines the refinement map R_t^l that has to be applied to remove hanging faces.

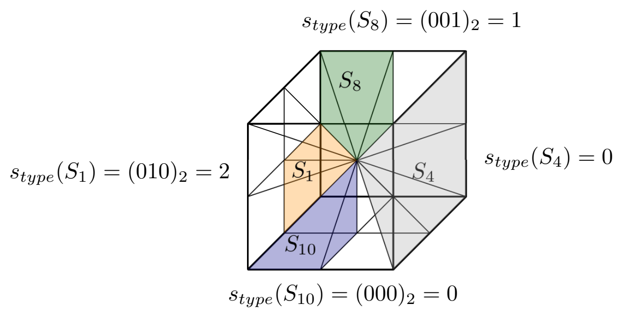
4.2.2 Subelement ID Type

This chapter focuses on the subelement ID type. The subelement ID type is a three-digit binary number derived from the transition type of a transition cell and the subelement ID of a subelement. The subelement ID type indicates for split subelements where the subelement is located in the transition cell. To be more precise, that means that the subelement ID type states whether the split subelement lies on the left/right side, on the front/back, or at the bottom/top of the four split subelements on this face. For non-split subelements, the subelement ID type is always zero.

Definition 4.2.2. Let E be a 3D hexahedral element and $b_0, b_1, b_2 \in \{0,1\}$. b_0 states whether the subelement base side lies on the left ($b_0 = 0$) or right ($b_0 = 1$) side of the hexahedral face. Analogously b_1 states whether the subelement base side lies on the front ($b_1 = 0$) or back ($b_1 = 1$) and b_2 states whether it lies on the bottom ($b_2 = 0$) or top ($b_2 = 1$). Then the binary concatenation $(b_0 b_1 b_2)_2 \in \{0,\dots,7\}$ is called the **binary subelement ID type** of E. We often refer to the decimal representation of this binary encoding in the following.

The binary subelement ID type can contain at most two ones. That means that the binary subelement ID type $(111)_2 = 7_{10}$ is not possible, because the base side of a subelement cannot lie on the right, back and top side simultaneously because it is a 2D quadrilateral face. Moreover, this implies that not every subelement ID type is possible for each hexahedral face f_i. For example, if face f_0 of a transition cell T is split, the sublements with base side on f_0 can either be in the front/back or on the bottom/top but trivially not on the left/right. Thus, the subelement ID type of face f_0 is between 0 (front and bottom) and 3 (back and top). The following $s_{type}(S_k)$ denotes the subelement ID type for subelement S_k. It is worth mentioning that a subelement ID type can be zero even if the subelement is split, but if it is not the type is always zero. For clarification, see Figure 4.3.

a)


$s_{type}(S_8) = (001)_2 = 1$
S_8
$s_{type}(S_1) = (010)_2 = 2$
S_1
S_4
$s_{type}(S_4) = 0$
S_{10}
$s_{type}(S_{10}) = (000)_2 = 0$

b)

face	f_0, f_1	f_2, f_3	f_4, f_5
s_{type}	$\{0,1,2,3\}$	$\{0,1,4,5\}$	$\{0,2,4,6\}$

Figure 4.3: Illustration a) shows the computation of different subelement ID types of a transition cell with transition type $t = (100110)_2 = 38$. $s_{type}(S_1) = 2$ because S_1 lies in the back and bottom. $s_{type}(S_4) = 0$ because this subelement is not split. $s_{type}(S_{10}) = 0$, thus, the subelement ID type can be zero even if the subelement is split. Part b) shows a table of all valid subelement ID types for the corresponding faces.

To compute the subelement ID type, the subelement ID is needed and the underlying transition type needs to be analyzed. The corresponding pseudo-code is given in Appendix A.

4.2.3 Adjustment Of The Element Data Structure

As described in Chapter 2, each element in `t8code` can be uniquely determined by its anchor node $\mathcal{A}$ and level l. Thus, for each element in `t8code` its level and anchor node are stored. However, inserting transition cells into a mesh, brings irregularities. Therefore, the element data structure has to be extended.
In a transitioned hexahedral mesh all routines have to manage hexahedral and pyramidal element types. Thus, we need an element data structure that differentiates the standard hexahedral elements between pyramidal subelements. In order to do so, the element data structure is extended by two additional parameters, the transition type (`transition_type`) and the subelement ID (`subelement_id`). The anchor node, level, transition type, and subelement ID offer all required information for implementing the transition algorithm in `t8code`. An overview of the new element data structure can be seen in Table 1.

Component	Declaration
$\mathcal{A}$	The anchor node $\mathcal{A} \in \{0,1,\dots,2^{19}\}^3$ defining the left, front, lower corner of the hexahedral element, given by $E.\mathcal{A}$.
l	The refinement level $l \in \{0,\dots,19\}$, $E.l$. In `t8code` the maximum refinement level for all hexahedral elements is $\mathcal{L} = 19$.
`transition_type`	If $E = S_{ijk}$ is a subelement in a transition cell T, E. `transition_type` describes the transition type of E in the sense of Definition 4.2.1. Thus, if E is no subelement E.`transition_type` is equal to zero.
`subelement_id`	If $E = S_{ijk}$ is a subelement in a transition cell T, E.`subelement_id` describes its location within the transition cell T in the sense of Definition 3.4.1. If E is no subelement E.`subelement_id` is equal to zero.

Table 1: This table shows the element data structure within a transitioned mesh extended by the transition type and the subelement ID.

It is worth mentioning, that the Morton index is not stored explicitly in the element struct but in the subelement ID. It is important to bear in mind, that when an element gets refined into a transition cell, the resulting subelement takes over the anchor node and the level of its hexahedral parent element.

Remark 1. Compared to Becker's work [5], we do not store the flag `is_sub` to minimize the memory effort. The flag `is_sub` is true if the element is a subelement and false otherwise. We do not store this flag because we can derive its informative value through the transition type. If the transition type is greater than zero, we know that the element is a subelement and the flag `is_sub` is no longer needed.

4.3 Implementing Transition Cells In `t8code`

To finally implement `t8_transition`, we need to introduce some essential algorithms. `t8_transition` iterates through a given forest and exchanges elements with hanging faces with suitable transition cells.

Therefore, we first need to implement a function that computes the transition type t. Thus, this function specifies a refinement criterion whether, and if yes, which transition cell needs to be inserted. This function is called `t8_element_compute_transition_type` and gets as input an element E and the corresponding forest $\mathcal{F}$ and returns the underlying transition type.

4.3.1 `t8_element_compute_transition_type`

`t8_element_compute_transition_type`, shown in Algorithm 1, uses the function `t8_element_face_neighbor`, provided by `t8code`, that returns a virtual face-neighbor of an element at a specific face. This virtual face-neighbor has the same level as the element itself. The idea of `t8_element_compute_transition_type` is to check whether a descendant of this virtual face-neighbor does exist in the mesh. The function `t8_element_has_leaf_desc` can do this. If this function returns true, a hanging face is identified, and thus, a bit-shift by one is performed.

Algorithm 1: Compute the transition type t of an element E

```
Data: A forest $\mathcal{F}$ and an element $E$
Result: Transition type $t \in \{0,\dots,64\}$
t8_element_compute_transition_type($\mathcal{F}$, $E$)
begin
    $t \leftarrow 0$
    // iterate through every face
    for $0 \le f <$ P8EST_FACES do
        // create virtual face-neighbor
        $N_n \leftarrow$ t8_element_face_neighbor($\mathcal{F}, E, f$)
        // Check if descendant of virtual face-neighbor exists
        if t8_element_has_leaf_desc($\mathcal{F}, N_n$) then
            $t \leftarrow t + 1 << (($P8EST_FACES$-1) - f)$
        end
    end
    // If $t > 0$ increase it by one to avoid $t = 1$
    if $t > 0$ then
        $t \leftarrow t + 1$
    end
    return $t$
end
```

Looking at the definition of the refinement rule in equation (23), we see that if $\varphi(E) = 1$, we interpret it as the element E should be refined into its regular children. Therefore, we do not want to allow the transition type 1, computed in Algorithm 1 and increase it by one if its greater than zero.

Remark 2. It is important to mention, that `t8code` uses some implementations of the `p4est` library by Carsten Burstedde from the University of Bonn [10]. The `p4est` library contains implementations for quadrilateral refinement in 2D and hexahedral refinement in 3D. The data structure for the 3D hexahedral elements, called `p8est` arises from the two-dimensional quadrilateral structure by extending with a third coordinate z. The macro `P8EST_FACES` defines the number of faces of a hexahedron. Thus, it equals 6. Analogously, the macro `P4EST_FACES` defines the number of faces of a quadrilateral which is 4.

4.3.2 `t8_element_num_subelements`

If the computed transition type t of an element E is greater than zero, to be more precise, if $t > 1$, E needs to be refined into a transition cell, and thus, a family of subelements arises. Therefore, the data structure of E needs to be adjusted, like in Table 1.

The number of subelements, and thus the number of siblings in a transition cell that form a family, depends on its transition type. The function `t8_element_num_subelements` gets as input data a transition type t and returns how many subelements in the corresponding transition cell exist, see Algorithm 2.

Algorithm 2: Counts the number of subelements in a transition cell with transition type t

```
Data: A transition type $t$
Result: Number of subelements $num\_subs \in \{0,\dots,24\}$
t8_element_num_subelements($t$)
begin
    $num\_subs \leftarrow 0$
    $\mathcal{H}_E \leftarrow 0$
    // Iterate through every face of element $E$
    for $0 \leq i <$ P8EST_FACES do
        // Count hanging faces of $E$
        $\mathcal{H}_E \leftarrow \mathcal{H}_E + \left(t \text{ and } (1 << i)\right) >> i$
    end
    $num\_subs \leftarrow$ P8EST_FACES $+ \mathcal{H}_E \cdot 3$
    return $num\_subs$
end
```

`t8_element_num_subelements` iterates through the faces of the element and, thus, through the binary representation of the given transition type t. While iterating t `and` $(1 << i)$ checks whether the i-th position of t equals one or zero. The operator `and` means the bit-wise and-operation here. The right-bit-shift executed afterward transforms the received number to one if it's not equal to zero. The amount of subelements is stored in *num_subs* and the amount of hanging faces is stored in $\mathcal{H}_E$. The amount of subelements is then the amount of faces of E, six, plus the amount of hanging faces times three, because one element of each face is already counted.

4.3.3 `t8_element_initialize_transition_cell`

The following Algorithm 3 shows the initialization of a family of subelements that forms a transition cell. The function is called `t8_element_initialize_transition_cell`. The amount of subelements in a family can differ and is computed by `t8_element_num_subelements`, shown in Algorithm 2.

While initializing, the anchor node and level of the parent element E are handed over to each subelement and remain unchanged. Naturally, keeping the level unchanged does not fit the definition of transitioned refinement maps. However, it is necessary because it simplifies the implementation in some parts, for example, when identifying neighbors in the transitioned forest. Additionally, keeping the anchor node unchanged does not match the underlying idea but it also simplifies the implementation in many places.

Moreover, the unique identification of an element is also given in the transitioned forest because of the assigned subelement ID and transition type of each element. Therefore, the enumeration of subelements follows the SFC index for transitioned forests from Definition 2.4.2. In Figure 4.4, the initialization of a transition cell is illustrated.

Algorithm 3: Initialize a family of subelements that form a transition cell of an element E.

```
Data: A hexahedral element $E$ and its transition type $t$
t8_element_initialize_transition_cell($E$,$t$)
begin
    for $0 \leq i <$ t8_element_num_subelements($t$) do
        S[$i$].$\mathcal{A} \leftarrow E.\mathcal{A}$
        S[$i$].$l \leftarrow E.l$
        S[$i$].transition_type $\leftarrow t$
        S[$i$].subelement_id $\leftarrow i$
    end
    return S[]
end
```

The general refinement function φ, introduced in equation (23), maps each element E either to -1, which means E shall be coarsened, to 0, which means E shall remain unchanged or to 1, which means that E shall be regularly refined. In a transitioned forest, we may not want to refine all elements regularly but irregularly into a transition cell. Therefore, we extend the general refinement function to φ_t to allow returning values higher than one, in particularly to the transition types.

$$\begin{aligned} \varphi_t \ &: \mathcal{S} \to \mathbb{Z} \\ E \ &\mapsto \{-1,0,1,2,\ldots\} \end{aligned} \tag{26}$$

with

$$\varphi_t(E) = \begin{cases} -1 & E \text{ and its siblings should be coarsened to its parent} \\ 0 & E \text{ should stay unchanged} \\ 1 & E \text{ should be regulary refined into its children} \\ > 1 & E \text{ should be refined into a transition cell} \end{cases} \tag{27}$$

where $\varphi_t(E) = k > 1$ indicates that the element E has to be refined into the transition cell with transition type $k-1$. The definition of φ_t justifies why `t8_element_compute_transition_type`, shown in Algorithm 1, does not return transition type 1 because then it would be ambiguous whether the element should be refined regularly or into a transition cell.

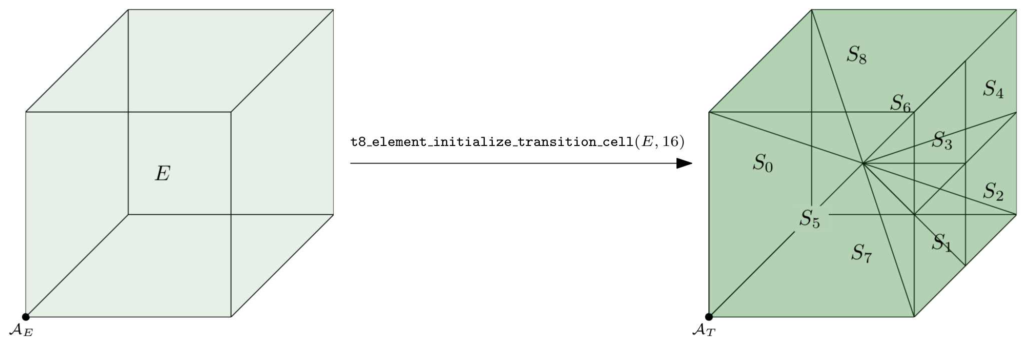


Figure 4.4: This figure shows an example of the function `t8_element_initialize_transition_cell` of a transition cell T with transition type 16, that means only the right face, f_1 of E, is hanging. It applies that `t8_element_num_subelements`$(E) = 9, \mathcal{A}_E = \mathcal{A}_T, \ell(E) = \ell(T)$ and `t8_element_compute_transition_type`$(\mathcal{F}, E) = (010000)_2 = 16$.

4.4 Transition

Now, the presented algorithms can be merged into a general routine, called `t8_transition`, that removes hanging faces out of a balanced forest. `t8_transition` works similarly to the `balance` algorithm presented in Chapter 4.1.3.

Let $\mathcal{F}$ be a forest that got modified by `t8_balance`. `t8_transition` copies the forest $\mathcal{F}$ to $\mathcal{F}^*$ and iterates through the elements of $\mathcal{F}^*$. If `t8_transition` identifies an element E with at least one hanging face, it exchanges it with the corresponding transition cell T, according to the rule $\mathcal{F}^* \leftarrow \mathcal{F}^* \backslash E \cup T$. In Algorithm 4 `t8_transition` is presented .

Remark 3. All elements of a tree $\mathcal{K}_i$ in a forest $\mathcal{F}^*$ are stored in an element array. In Algorithm 4 the array $\mathcal{K}_i$.`elements` stores every element of the tree $\mathcal{K}_i$. Thus, all elements of $\mathcal{F}^*$ are stored in an element array of the corresponding tree. The elements in $\mathcal{K}_i$.`elements` are stored in Morton-SFC order.

Algorithm 4: Remove all hanging faces of a given forest $\mathcal{F}$ and return the transitioned forest $\mathcal{F}^*$

```
Data: A balanced forest $\mathcal{F}$
Result: A transitioned forest $\mathcal{F}^*$
t8_transition($\mathcal{F}$)
begin
    // Copy $\mathcal{F}$ to $\mathcal{F}^*$
    $\mathcal{F}^* \leftarrow \mathcal{F}$
    // Iterate through the trees of $\mathcal{F}^*$
    for $\mathcal{K}_i \in \mathcal{F}^*$.trees do
        // Iterate through the elements of tree $\mathcal{K}_i$
        for $0 \leq j < |\mathcal{K}_i|$ do
            // Get the j-th element of $\mathcal{K}_i$ and store it in $E$
            $E \leftarrow \mathcal{K}_i$.elements$[j]$
            // Compute the transition type
            $t \leftarrow$ t8_element_compute_transition_type($\mathcal{F}^*, E$)
            // Check whether $E$ needs to be transitioned or not
            if $t > 0$ then
                // Get the corrected transition type
                $t \leftarrow t - 1$
                // Exchange $E$ by its corresponding transition cell
                $\mathcal{K}_i$.elements$[j] \leftarrow$
                  $\mathcal{K}_i$.elements$[j] \setminus E \cup$ t8_element_transition($E, t$)
            end
        end
    end
    return $\mathcal{F}^*$
end
```

More low-level functions are needed to implement the `t8_transition` in `t8code`. One crucial algorithm is, of course, computing the coordinates of the subelements. This is achieved with the function `t8_element_vertex_coords_of_subelement`. The coordinates of the subelements are given in Definition 3.3.1.

Figure 4.7 shows the different call-pipelines of high-level algorithms in `t8code` with enabling transition cells on the right and without them on the left. It is essential to bear in mind, that the `balance` algorithm is optional in the non-transitioning case but mandatory if the user wants to transition its mesh.

Furthermore, we see that if the high-level algorithm `t8_transition` is called before `t8_adapt`, then `t8_adapt` is likely to deal with a transitioned forest, thus with a mesh containing regular hexahedra and subelements in pyramid shape. Consequently, we must consider how `t8_adapt` works on a transitioned forest. Thus, if an element E shall be coarsened into its parent E, all of its siblings shall be coarsened, too.

Figure 4.5 depicts an exemplary adapted and then transitioned mesh.

If E is a subelement, the number of siblings in the transition cell has to be determined. This is done by the function `t8_element_num_siblings` presented in the following Algorithm 5 that uses the function `t8_element_num_subelements` shown in Algorithm 2.

Algorithm 5: Return the number of siblings of an element E

```
Data: An element $E$
Result: Number of siblings of $E$

t8_element_num_siblings($E$) begin
    // Check if $E$ is a subelement
    if $E$.transition_type $> 0$ then
        return t8_element_num_subelements($E$.transition_type)
    else
        return P8EST_CHILDREN
    end
end
```

`t8_element_num_subelements` first examines whether the input element E is a subelement or not. If it is a subelement, its transition type is greater than zero and the amount of subelements gives the number of siblings in the corresponding transition cell. If E is a regular hexahedron, the number of siblings is always eight, which equals `P8EST_CHILDREN`.

Now, a function can be constructed that states whether a set of elements forms a family. The pseudo-code of this function `t8_element_is_family` is presented in Algorithm 6.

The presented algorithm has a performance advantage compared to the algorithm presented in Becker's work [5]. Becker constructs the parent element of the first element of the element set $E[]$. That means if $E[0]$ is a subelement, the transition type and the subelement ID of $E[0]$ are set to zero. Then, he builds the parent element's transition cell according to the transition type of $E[0]$. If E is no subelement the function `t8_element_parent`, shown in Figure 4.2, is used, and the children of the parent element are constructed via `t8_element_children`. After that, he inspects if, in the case of a subelement, the transition cell or in case of no subelement, the set of regular children is equal to the input set of elements. If true, they form a family; if not, the elements are not siblings. Analyzing this approach, it is apparent that in every scenario, a parent element is constructed, and then the resulting children of this parent element. It follows that at least eight children and one parent element are built, in every scenario. Additionally, two sets of elements need to be compared.

In the approach presented in Algorithm 6 there is an initial screening to see if element $E[0]$ is a subelement or not. If true, we already know that the following *#num_siblings*, which equals the amount of elements in $E[]$, are its siblings and therefore we can quit the function prematurely without constructing additional elements.

Algorithm 6: Check if a set of elements $E[]$ form a family or not

```
Data: A set of elements $E[]$
Result: True if $E[]$ forms a family, false if not
t8_element_is_family($E[]$) begin
    // Check if $E[0]$ is a subelement we assume to be the following
       num_siblings its siblings
    if $E[0]$.transition_type $> 0$ then
        return TRUE
    // If the first element is no subelement we check the following seven
       elements
    else
        // If any of the following elements is a subelement, $E[]$ can not
           form a family
        if $E[i]$.transition_type $> 0$ for any $1 \leq i \leq 7$ then
            return FALSE
        else
            return p8est_quadrant_is_family($E[]$)
        end
    end
end
```

The pseudo-code of how to transition an adapted forest is given in [5], Algorithm 7. Therefore, we only sketch the procedure here. Since we have to iterate through all elements of the input forest $\mathcal{F}$ to obtain the transitioned and adapted forest $\mathcal{F}^*$, we first copy $\mathcal{F}$ into $\mathcal{F}^*$. Afterward, we iterate through all trees and then through all elements of the tree. Subsequently, we compute the refinement value of each element according to the user-defined refinement rule. After that, we check whether the element is a subelement. If it is a subelement, it first has to be coarsened into its parent. Thus, the refine value gets updated to -1, which means coarsen; see Figure 4.2 for explanation. Afterward, an inspection is required to see whether the refine value is equal to minus zero (Case 1), equal to zero (Case 2), equal to one (Case 3), or greater than one (Case 4).

Case 1: The element and its siblings should be coarsened into its parent. That means the number of siblings has to be determined. This is done by `t8_element_num_siblings` shown in Algorithm 5. Then a family check, see Algorithm 6, is done, and if the family is identified, all children are replaced by its parent element in the new forest $\mathcal{F}^*$. Afterward, the element-iterator increases by the number of siblings. If the family check is not successful, nothing happens, and the element-iterator increases by one.

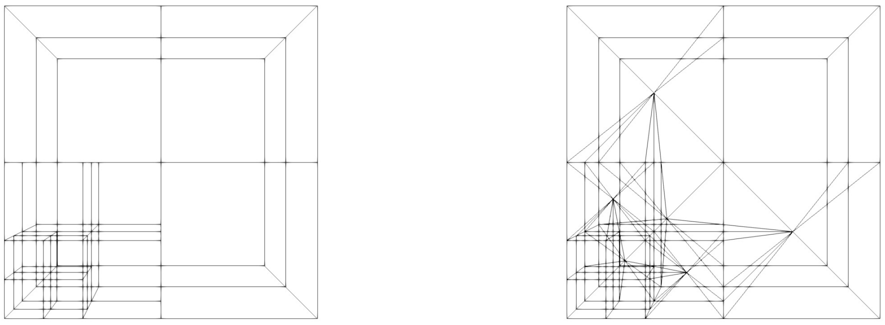


Figure 4.5: On the left, we see a two-times adapted mesh. In every adaptation step, the element with element ID zero was refined. `t8_balance` does not insert any elements because the mesh is already balanced. On the right, we see the transitioned mesh with inserted transition cells.

Case 2: In this case the element has to be inserted in the new forest $\mathcal{F}^*$, and the element-iterator increases by one.

Case 3: The element should be regularly refined into its eight hexahedral children. The parent element has to be excluded and the children elements inserted in the new forest $\mathcal{F}^*$. Because the element-iterator iterates over the "old" forest $\mathcal{F}$, it increases by one.

Case 4: This case is the transitioning case. This case can not occur if the user does not set the transitioning flag to one. First of all the refinement value has to be decreased by one, see Algorithm 1 for explanation. The transition type then equals the refine value needed as input data to initialize the transition cell via `t8_element_initialize_transition_cell`.

In the following remark, the changes/differences to the implementation of Becker in [5] are presented.

Remark 4. Case 1: Naturally, elements can only be coarsened to its parent if their level is greater than zero. This must be confirmed first.
Case 3: Analogue, the refinement level has to be inspected. If it is already at maximum level (in the hexahedral case 19) the element can not be refined any further and gets inserted unchanged into the new forest.

The workflow of the high-level algorithms of a regular `t8code` application with and without transitioning can be seen in Figure 4.7.

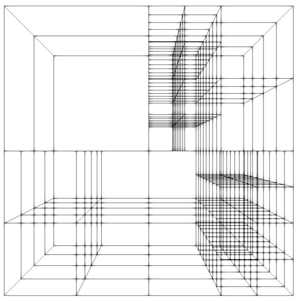
In [39], Schneiders defines a numerical stable refinement, given in the following definition.

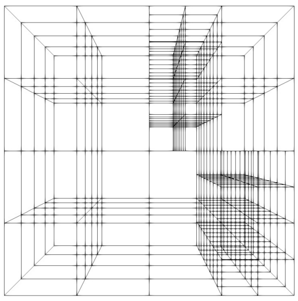
Definition 4.4.1. A refinement is **stable** if the minimum angle α_{min} of any element of a refined mesh does not depend on the refinement level l Thus,

$$\alpha_{min} \geq \gamma > 0 \tag{28}$$

(Definition 4 in [39])

Naturally, the presented transitioning algorithm does not depend on the underlying refinement level because of the decomposition of each subelement before re-adapting the mesh. Therefore, the transition algorithm is numerically stable.


(a) `t8_adapt`


(b) `t8_balance`

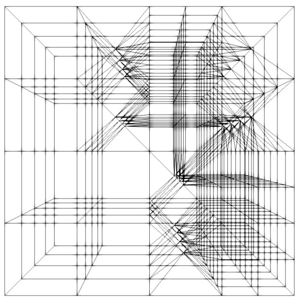
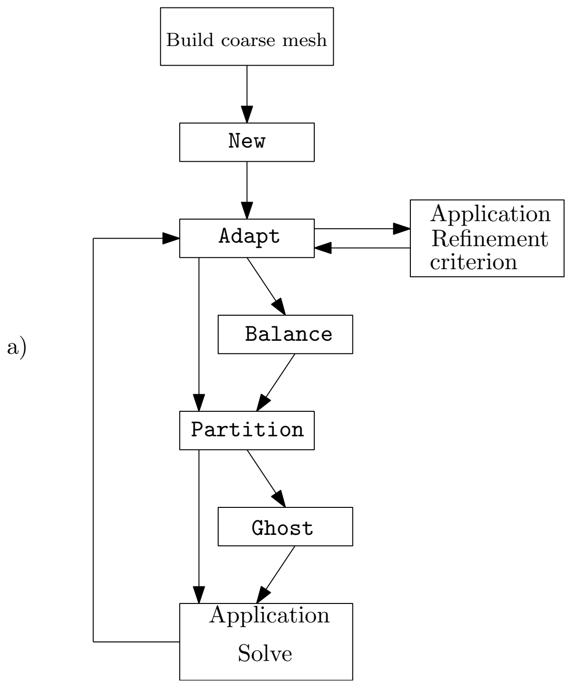

(c) `t8_transition`

Figure 4.6: Illustration (a) shows a mesh that is only adapted via `t8_adapt`. Figure (b) shows this mesh with a fulfilled face-balancing condition via `t8_balance`. Illustration c) shows the transitioned mesh via `t8_transition`.



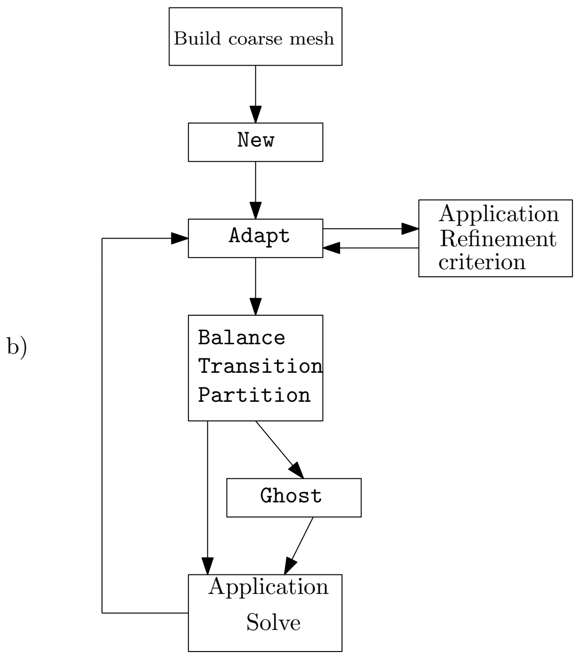


Figure 4.7: Figure a) illustrates the workflow of a regular `t8code` application. We see that only partitioning the mesh is a must. The algorithms `t8_balance` (and `t8_ghost`) are optional. Figure b) shows an application workflow with the option of transitioning the mesh. We see that `t8_balance` is not optional anymore.

5 Identifying Face-Neighbors

Identifying face-neighbors is a crucial task because most numerical PDE solvers need information about the connectivity of elements to update a given element's numerical values. Therefore, it is necessary that face-neighbors can be identified in an efficient way. In Chapter 4.3 a SFC index based on the Morton index and the subelement ID was introduced in order to identify elements in a transitioned forest. The low-level algorithm `t8_element_leaf_face_neighbors`, implemented in `t8code`, computes face-neighbor elements. In this chapter, this algorithm will be extended in order to identify face-neighbors in a hexahedral transitioned forest.

Therefore, we first discuss the case when a face-neighbor of an element is not its sibling in Chapter 5.1. Afterward, in Chapter 5.1.1, an algorithm is presented that computes the exact location of a subelement inside a transition cell. Following this, we discuss how to identify a face-neighbor inside a transition cell in Chapter 5.2.

The key logic of `t8_element_leaf_face_neighbors` for a transitioned forest is the use of virtual face-neighbors and the binary search with the use of the extended Morton SFC index.

A distinction is made between finding face-neighbors inside a transition cell, thus, the neighbor is a sibling, too, and finding face-neighbors of outer faces of a transition cell or regular hexahedron.

In the presented approach, there is not always a one-to-one correspondence between elements faces and their neighbors. Because of the occurring hanging edges, shown in Figure 3.4, there are cases where one subelement has two face-neighbors, which are siblings. In Chapter 5.2, we discuss this case in detail. In all of the other cases, there exists a one-to-one correspondence. That means that every element has exactly one face-neighbor at one face.

5.1 Face-Neighbors In Transitioned Forests

This chapter discusses the face-neighbor relations of subelements if the face-neighbor is not a sibling. If the face-neighbor of E is not a sibling but E is a subelement it follows that we are looking for the face-neighbor at face f_4 of E because f_4 is always pointing outwards to its transition cell. This situation is illustrated in Figure 5.1 below.

The face-neighbor of a subelement at face f_4 can either be a hexahedron or a subelement. The following definition describes the coordinates of a hexahedral face-neighbor of a subelement.

Definition 5.1.1. Let $S_{ijk} = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4]$ be a subelement with indices as given in Definition 3.3.1. The hexahedral face-neighbors $N_{S_{ijk}}$ of S_{ijk} are given by

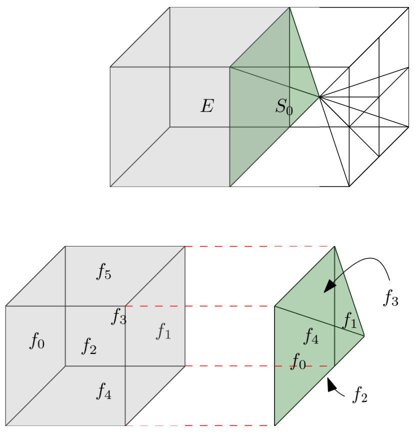


Figure 5.1: This figure shows a possible scenario where the face-neighbor of a subelement S_0 at face f_4 is a regular hexahedron E on face f_1. All other face-neighbors of S_0 are siblings.

the following coordinates:

$$\begin{aligned}
N_{S_{ij0}} &:= [\vec{x}_0 - x,\ \vec{x}_0,\ \vec{x}_1 - x,\ \vec{x}_1,\ \vec{x}_2 - x,\ \vec{x}_2,\ \vec{x}_3 - x,\ \vec{x}_3] \\
N_{S_{ij1}} &:= [\vec{x}_0,\ \vec{x}_0 + x,\ \vec{x}_1,\ \vec{x}_1 + x,\ \vec{x}_2,\ \vec{x}_2 + x,\ \vec{x}_3,\ \vec{x}_3 + x] \\
N_{S_{ij2}} &:= [\vec{x}_0 - y,\ \vec{x}_1 - y,\ \vec{x}_0,\ \vec{x}_1,\ \vec{x}_2 - y,\ \vec{x}_3 - y,\ \vec{x}_2,\ \vec{x}_3] \\
N_{S_{ij3}} &:= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_0 + y,\ \vec{x}_1 + y,\ \vec{x}_2,\ \vec{x}_3,\ \vec{x}_2 + y,\ \vec{x}_3 + y] \\
N_{S_{ij4}} &:= [\vec{x}_0 - z,\ \vec{x}_1 - z,\ \vec{x}_2 - z,\ \vec{x}_3 - z,\ \vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3] \\
N_{S_{ij5}} &:= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \vec{x}_0 + z,\ \vec{x}_1 + z,\ \vec{x}_2 + z,\ \vec{x}_3 + z].
\end{aligned} \tag{29}$$

Thereby x,y and z denote the shift value in x-,y- and z-direction, which is determined by the side length of the parent of S_{ijk}. To be more precise, $x,y,z = |\vec{x}_0 - \vec{x}_1|$.
If S_{ijk} has a pyramid-shaped face-neighbor at face f_4, thus its neighbor is a subelement, its coordinates are given by:

$$\begin{aligned}
N_{S_{ij0}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} - \frac{x}{2}] \\
N_{S_{ij1}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} + \frac{x}{2}] \\
N_{S_{ij2}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} - \frac{y}{2}] \\
N_{S_{ij3}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} + \frac{y}{2}] \\
N_{S_{ij4}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} - \frac{z}{2}] \\
N_{S_{ij5}} &= [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} + \frac{z}{2}]
\end{aligned} \tag{30}$$

or shorter, with $s = \frac{|\vec{x}_0 - \vec{x}_1|}{2}$:

$$N_{S_{ijk}} = [\vec{x}_0,\ \vec{x}_1,\ \vec{x}_2,\ \vec{x}_3,\ \frac{\vec{x}_3 - \vec{x}_0}{2} + (-1)^{1+k} \cdot s] \tag{31}$$

Notice that the face-neighbor of a subelement can either have the same level as the subelement or one level higher but, of course, never lower. An illustration is given in Figure 5.2.

The `t8code` function `t8_element_face_neighbor` computes a virtual face-neighbor inside the root tree of a given element E and a face f. Notice that this function does not compute a neighbor inside a transition cell but accepts a subelement as an input element. From this, it follows that if the input element is a subelement it is only valid to call this function if the face equals f_4 because this is the only outward-pointing face of a subelement.

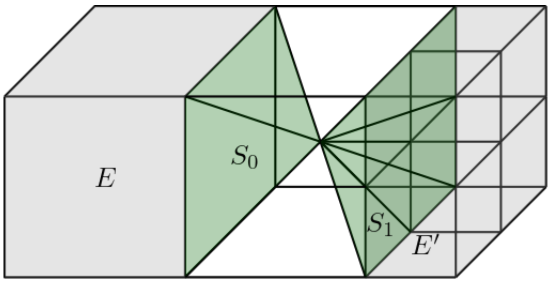


Figure 5.2: This figure shows neighborhood relations of subelements to regular hexahedra. The level of E equals the level of S_0, and the level of E' is one higher than that of S_1.

The function `t8_element_face_neighbor` computes the coordinates of the face-neighbor according to Definition 5.1.1. To compute the face-neighbor of f_4 of a subelement, `t8_element_face_neighbor` needs to know where exactly the subelement lies in the transition cell. For that reason, I implemented the function `t8_element_get_location_of_subelement`. This function is discussed in detail in the following Chapter 5.1.1.

5.1.1 `t8_element_get_location_of_subelement`

In order to determine the exact location of a subelement in a transition cell, the function `t8_element_get_location_of_subelement` is needed. The aim is to create a location array L with the length of three unsigned integers. The location array is constructed as follows: $L = [\texttt{face}_H, \texttt{split}, \texttt{subelement\_id\_type}]$.
The first entry $\texttt{face}_P$ determines the face of the parent element P where the face f_4 of the subelement lies on, thus $\texttt{face}_P \in \{0,1,2,3,4,5\}$. The second entry $\texttt{split} \in \{0,1\}$ specifies whether the subelement is split ($\texttt{split} = 1$) or not ($\texttt{split} = 0$). The third entry states the `subelement_id_type` as determined in Definition 4.2.2. In Figure 5.3, the location array is calculated for three subelements as an example.

The `split`-value and the `subelement_id_type` can both be derived from the transition type. Therefore, the transition type must be converted into the bitwise-form stored in an array. This procedure is implemented in the function `transition_type_to_binary_array` with transition type t as input data.

Furthermore, a cumulative array derived from the binary array, which is six digits long and stores the cumulative amount of subelements for each face, must be calculated. That means, if the binary array is equal to zero/one at position i, we derive that there are one/four more subelements to add up at face f_i. So, we

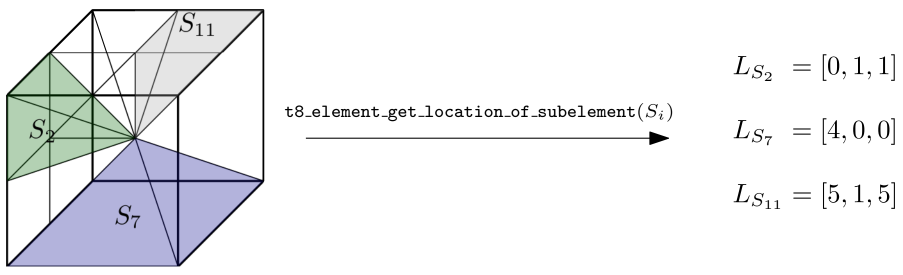


Figure 5.3: This figure shows three exemplary location arrays for three different subelements. The function `t8_element_get_location_of_subelement` generates the location arrays. The first entry of the location array L_{S_2} is equal to zero because the face f_4 lies on the face f_0 of the parent element, see Figure 2.3.

can derive the amount of subelements up to each single face from the cumulative array. The input data for this function is the binary array of the transition type computed by the function `transition_type_to_binary_array`.

For reasons of readability, the algorithm of `transition_type_to_binary_array` and `compute_cumulative_array` are shown in Algorithm 8 and Algorithm 9 in Appendix A.

The function `t8_element_get_location_of_subelement` works as follows: First, the corresponding hex-face, where the face f_4 of the subelement lies, is determined. This is done by comparing the subelement ID with the entries of the cumulative array. If the subelement ID is greater or equal to the i-th entry and smaller than the $i+1$-th entry, we know that face f_{i+1} of the parent element is the corresponding hex-face.
Then, the `split`-value is determined by verifying whether the binary array is equal to one at the corresponding hex-face. If so, the `split`-value equals one.

If the subelement is split, the subelement ID type must be specified. This is done by comparing the subelement ID with the entry of the cumulative array at the position of the `face`-value. In doing so, different cases that occur for the different faces of the parent element must be distinguished. A list of all possible subelement ID types and the corresponding hex-faces is given in Figure 4.3 b).

Note that all subelements in a transition cell do have the same anchor node. Thus, the question remains to identify the right subelement-neighbor if only the Morton index of the transition cell is given. Therefore, we must determine the exact location via the algorithm `t8_element_get_location_of_subelement` presented above.

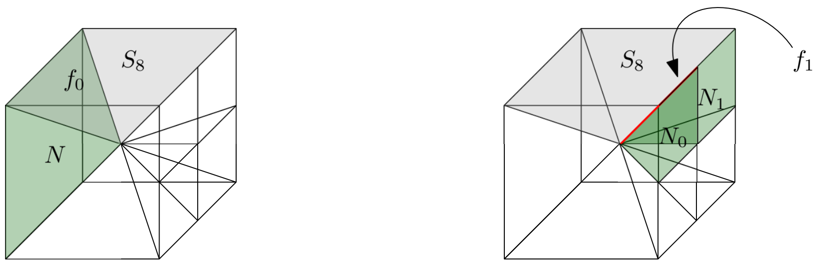
5.2 Finding Face-Neighbors In Transition Cells

In this chapter the essential low-level function `t8_element_get_sibling_neighbor-`

`_in_transition_cell` is introduced. The pivotal question is:

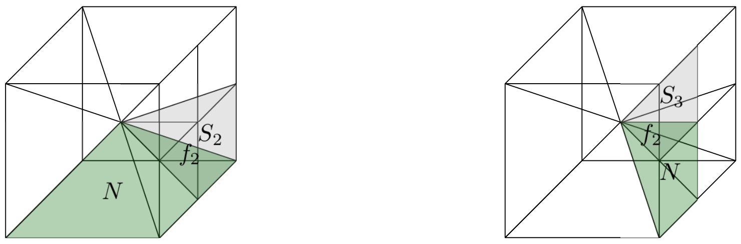
If S is a subelement, what is the sibling neighbor N of S at face $f_0, \ldots, f_3$?

Notice that the face-neighbor of f_4 of a subelement can not be a sibling neighbor. This case is discussed in detail in Chapter 5.1.



Case 1) Subelement S_8 is not split, and so is its neighbor at face f_0.

Case 2) Subelement S_8 is not split, but so is its neighbor at face f_1.



Case 3) Subelement S_2 is split, but its neighbor at face f_2 is not split.

Case 4) Subelement S_3 is split, and so is its neighbor at face f_2.

Figure 5.4: Four cases can occur when identifying a sibling neighbor in a transition cell. The presented transition cell has transition type 16. Notice that in case 2) we identify two sibling neighbors at one face. This is because of the hanging edge, colored in red.

We have to distinguish between four cases that can occur:

1. The subelement S is not split, so is its face-neighbor N.
2. The subelement S is not split, but its face-neighbor N is split.
3. The subelement S is split, but its face-neighbor N is not split.
4. The subelement S is split and so is its face-neighbor N.

All of these different cases are illustrated in Figure 5.4. Notice that if the subelement S is not split, but its neighbor is split, it follows that S has two neighbors at this face. This occurs because of the hanging edges in the transition cell. This problem and how to solve it is discussed in Chapter 7. The algorithm `t8_element_get_num_sibling_neighbors_at_face` returns the amount of siblings inside a transition cell by comparing the own `split`-value with the neighbor's one. Thus, `t8_element_get_num_sibling_neighbors_at_face` returns either two if the own `split`-value is equal to zero and the neighbor's one is equal to one or one otherwise.

In order to identify a sibling neighbor in a transition cell, a relevant concept is *dual faces* described in the following definition.

Definition 5.2.1. Let E and E' be two face-neighbored elements, that intersect at the face F_E of E and the face $f_{E'}$ of E' respectively. We say that the index of

f_E is the **dual face** of $f_{E'}$, from the perspective of E'. Vice versa, $f_{E'}$ is the dual face of f_E from the perspective of E.

The face duals of a regular hexahedral can be found in Appendix in Definition A.0.1.

Additionally, we distinguish between subelement face duals and dual subelements to a face of a subelement. The following two look-up tables show the relations.

a)

S / P	f_0	f_1	f_2	f_3	f_4
f_0	0	0	0	0	4
f_1	1	1	1	1	4
f_2	0	2	0	2	4
f_3	1	3	3	3	4
f_4	2	2	2	2	4
f_5	3	3	3	3	4

b)

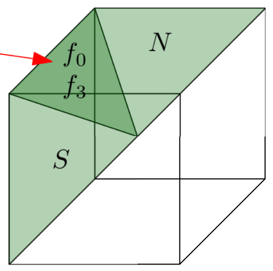


Figure 5.5: a) shows the look-up table `subelement_face_dual`[6][5]. Illustration b) shows an example at entry `subelement_face_dual`[0][3]: The face-dual of subelement S at face f_3 is face f_0 from subelement N.

The look-up table `subelement_face_dual`, shown in Table 5.5, describes the relationship between a subelement and its face neighbors. The rows determine which subelement is considered by assigning the face of the parent hexahedron to the base side of the subelement. This means that row zero shows the face-dual relations of a subelement with the basal face on face f_0 of its parent. Naturally, face f_4 does not have a face-dual in its transition cell. Thus, we establish the rule that face f_4 is dual to itself. A complete list of all non-split subelement face labels can be found in Appendix A.2.

a)

S / P	f_0	f_1	f_2	f_3	f_4
f_{P0}	2	3	4	5	-1
f_{P1}	2	3	4	5	-1
f_{P2}	0	1	4	5	-1
f_{P3}	0	1	4	5	-1
f_{P4}	0	1	2	3	-1
f_{P5}	0	1	2	3	-1

b)

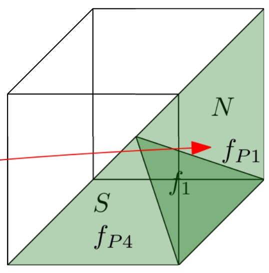


Figure 5.6: a) shows the look-up table `subelement_face_to_dual_subelement`[6][5]. Illustration b) shows an exemplary entry at `subelement_face_to_dual_subelement`[4][1]: For clarity, the faces of the parent P are declared with f_{Pi} to distinguish them from the subelement faces. Thus, the subelement S, with basal face f_{P4}, dual on face f_1 is subelement N with base side f_{P1}.

In addition, we store the relation between a subelement's face and its neighboring subelement in a second look-up table `subelement_face_to_dual_subelement`, shown in Table 5.6. Again, the rows determine which subelement is considered, and the columns state the basal face of the face-neighbored subelement. Notice the convention to assign f_4 to -1 because f_4 has no face-neighboring siblings.

The advantage of look-up tables is that they reduce run-time computations, and the code is relatively easy to maintain with good documentation. The disadvantage is that the RAM is relatively slow. Thus, many RAM accesses due to look-up tables can decrease the performance. But, the presented look-up tables do not store much data and thus, they fit into the caches. Therefore, the performance is hardly negatively affected by the look-up tables. Nevertheless, if simple calculations can replace look-up tables, they should be preferred.

All of the key algorithms, needed to implement the function `t8_element_get_sibling_neighbor_in_transition_cell_hex` are introduced. The input data consists of a subelement S and a face f. The returning data is an array filled with one or two face-neighboring subelements and the corresponding dual face/faces. For a more detailed description of the implementation, the pseudo-code of `t8_element_get _sibling_neighbor_in_transition_cell_hex` is presented Algorithm 7 in Appendix A.

`t8_element_get_sibling_neighbor_in_transition_cell_hex` works as follows: First of all, the location array L of the subelement is created by calling the function `t8_element_get_location_of_subelement(`S`)`. Then, the face-index of the neighboring subelement is identified by checking the `subelement_face_dual` look-up table in row f and column $L[0]$. Afterward, the algorithm checks which case is the correct one. Therefore, the algorithm distinguishes whether the subelement S is split or not. Then, the subelement ID of the face-neighbor/face-neighbors is calculated by counting the subelements up to the sought-after.

It is worth mentioning that there are some special cases that can occur if the subelement S is split: First, if the subelement is split, a distinction must be made as to whether the face-neighbored subelement base side lies on the same parental face or not. This can be done with a helper variable, which covers all cases where the face-neighbored subelement lies on the same parental face. The helper variable equals one if the neighbor-subelement lies on the same face, and thus, the corresponding scenario is case 4. Otherwise, the binary representation of the transition type needs to be checked at the position of the dual subelement, stored in the look-up table `subelement_face_to_dual_subelement`, shown in Table 5.6.

Regarding case 4, there is one exception in calculating the subelement ID of the face-neighbor stated in the following remark.

Remark 5. If the face-neighbored element's base side is on the same parental face as the subelement itself, the calculation of the subelement ID of the face-neighbored subelement differs. In that scenario, the subelement ID gets calculated by analyzing the subelement ID type and subelement ID of S to derive the subelement ID of the face-neighbor.

6 Analysis Of The Influence Of `t8_transition`

In this chapter, we discuss the advantages and disadvantages of transition cells. Certainly, inserting transition cells leads to an additional programming effort and computation time. Thus, it is important to discuss whether this is in proportion to the advantages of a transitioned, respectively conformal mesh.

First, the amount of elements in transitioned meshes is analyzed compared to non-transitioned meshes in Chapter 6.1. The amount of elements depends on the amount of subelements, and thus, it also depends on the different transition types of the used transition cells. Certainly, with an increasing amount of hanging faces of an element, the number of subelements also increases. Then, some runtime measurements are done by comparing a transitioned mesh to a non-transitioned mesh in Chapter 6.2. After that, in Chapter 6.3, some runtimes of identifying a leaf-face-neighbor via the function `t8_element_leaf_face_neighbor` are presented and discussed. Finally, the influence of transitioning according to the mesh quality is discussed in Chapter 6.4.

6.1 Measurements Of The Amount Of Elements

First, we look at the number of regular hexahedra compared to the amount of subelements. Therefore, we need to determine a refinement criterion and an initial level of the coarse mesh.

Different refinement regions are considered to analyze the impact of subelements in a transitioned mesh. First, two different static refinement regions are discussed. After that, a dynamic refinement region is considered.

As the coarse mesh, we always take a regular cube. The first considered refinement area is a sphere in the middle of the coarse mesh with a centroid at $(0.5, 0.5, 0.5)$. The radius of the sphere is 0.25. All elements inside of this sphere will be refined at each adaptation step. In order to measure the distance between the element and the sphere, the midpoint of each element is calculated. The initial refinement level is 3. Thus, the initial amount of hexahedra is $8^3 = 512$. Six adaptation steps are conducted. The refinement region is called static because it does not change over time. The results are shown in a histogram in Figure 6.1.

In Figure 6.1, we see that the ratio of subelements to regular hexahedral elements increases while refining more elements. In the first adapt step the ratio of subelements to regular elements is $\approx 29.6\%$. After adaptation step 6, it increases to a ratio of $\approx 59.5\%$. Thus, the amount of subelements makes up a large part, namely over half. So, the proportion of subelements dominates.

Naturally, increasing the initial refinement level leads to a decreasing part of the subelements. To prove that, the presented scenario above is conducted for the initial refinement levels $3, 4, 5$ and 6. The results are presented in Figure 6.2. It is worth mentioning that a higher initial refinement level leads to a more accurate approximation of the geometry and thus can be useful.

We clearly see that the ratio of subelements decreases sharply when increasing

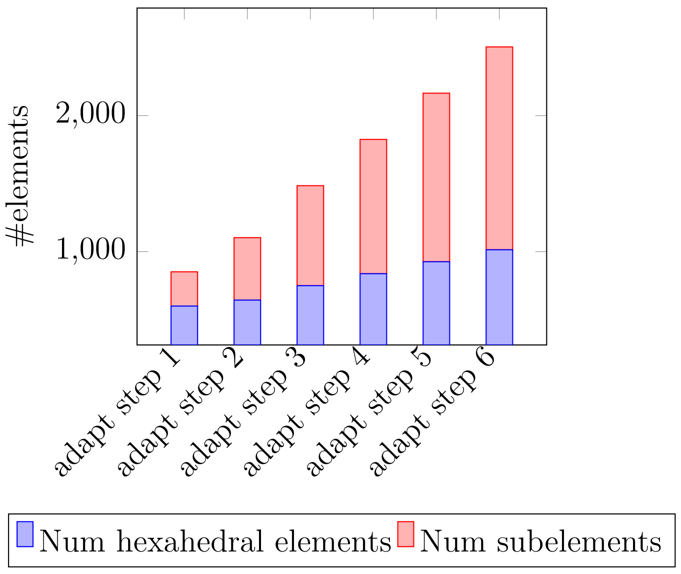


Figure 6.1: The histogram shows the amount of subelements and regular hexahedral elements. The refinement criterion is a sphere $\mathcal{S}_r(0.5, 0.5, 0.5)$ with radius $r = 0.25$ and centroid at $(0.5, 0.5, 0.5)$. All inner elements of the sphere are refined in each adaptation step.

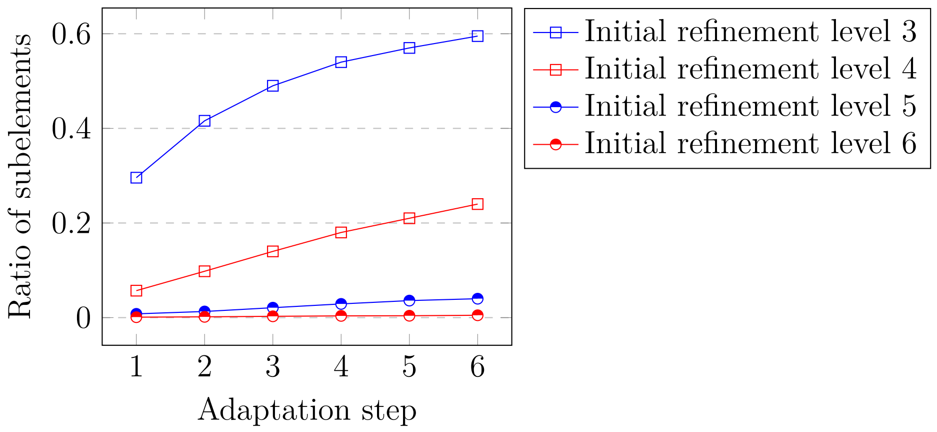


Figure 6.2: This figure shows the ratio of subelements to regular hexahedral elements when refining inside a sphere $\mathcal{S}_r(0.5, 0.5, 0.5)$ with radius $r = 0.25$ and centroid at $(0.5, 0.5, 0.5)$.

the initial refinement level from 3 to 4 and from 4 to 5. The ratio of subelements with initial level 4 after the sixth adaptation step is equal to $24,5\%$, and with initial refinement level 6, it is only 0.056%. The difference between the ratio of subelements in a mesh with initial level 6 does not differ as much from a mesh with initial refinement level 5—the proportion there is in both cases deficient. In general, the impact of subelements decreases strongly. Furthermore, we can see that the graph of initial refinement level 3 resembles a logarithmic function. In contrast, all other graphs look more linearly, if not even constant, for initial refinement level 6. We can conclude from this that the impact of subelements decreases when the initial refinement level increases.

Next, a different static refinement region will be considered. Every element on

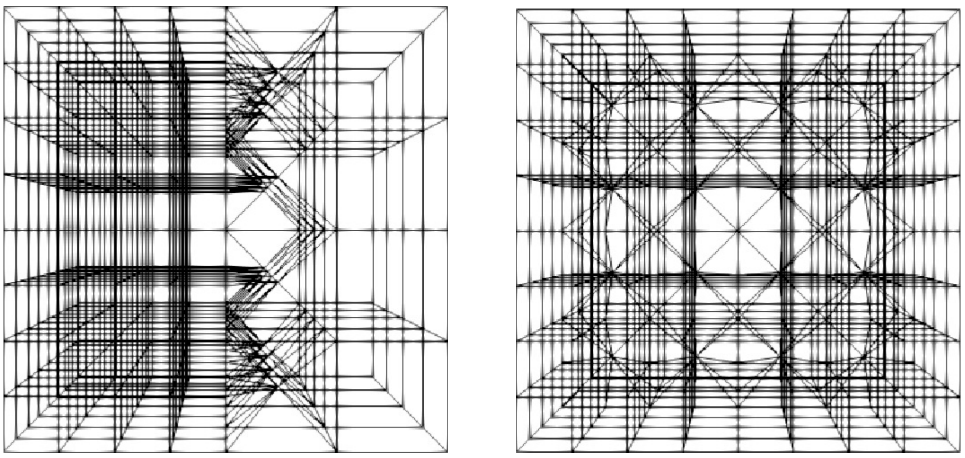

(a) front side (b) top side

Figure 6.3: This figure shows a refinement of one-half side of a cube. The elements on the left side have level 3, and those on the right side have level 2.

the cube's left side is being refined. That means all elements with an x-coordinate of their anchor node smaller than 0.5 are refined. The resulting mesh with initial refinement level 2 (due to the better perceptibility) from two different perspectives can be found in Figure 6.3.
The measurements are conducted with initial level 3. The resulting mesh then consists of one transition layer. The additional amount of subelements is 576, but every transition cell has transition type 32 and thus consists of "only" 9 subelements, which is the smallest amount of subelements a transition cell can consist of. In total, there are $2,816$ elements, and therefore, the ratio of subelements is approximately 20.45%. Compared to the results of 6.2, this ratio is quite small for initial level 3. In Figure 6.2, we see that already after the first adaptation step the ratio of subelements was about 30%. Additionally, the ratio of subelements is also compared to the results of Figure 6.1, with 20.45% relatively small. This indicates that the share of subelements depends on the underlying refinement criterion.
The amount of subelements in a transitioned mesh is also related to the amount of nodes in a mesh. We discuss the additional amount of inserted nodes, which equals $576/9 = 64$. This is $\approx 2\%$ of the total amount of nodes, which is $2,624$, and thus relatively small.

The now-considered refinement region is dynamic. Thus, it changes from step to step. We consider a sphere with an initial radius equal to zero and a band around this sphere with a bandwidth of 2.0. Every element within the band will be refined in each adaptation step. If an element lies outside of the band, it gets coarsened. After each adaptation step, the radius of the sphere increases by 0.05. Six adaptation steps are conducted. After the sixth adaptation step, the refinement area reaches the boundary of the cube. Figure 6.4 shows the amount of elements. A distinction is made between a mesh that is transitioned, a mesh that is adapted and balanced and a mesh that is just adapted according to the refinement criterion.
We see that the additional amount of elements after transitioning the mesh in step 6 is $72,718$. The additional amount of elements caused to `t8_balance` after step 6

is 16,919. That means that `t8_transition` increases the amount of elements by factor ≈ 4.15, and `t8_balance` demonstrates a smaller increase of the amount by factor ≈ 1.595. That shows that the impact of transitioning, in the sense of additional elements, in this case, is about 2.6 times as high as the influence of balancing the mesh. The ratio of subelements after step 6 is quite high, with approximately 60%. After transitioning, the mesh consists of 6,955 transition cells. That results in an additional 6,955 nodes. The average amount of subelements in one transition cell is 10.24 and thus relatively small (9 subelements is the minimum amount per transition cell). That means that, on average, a transition cell has between 2 and 3 hanging faces.

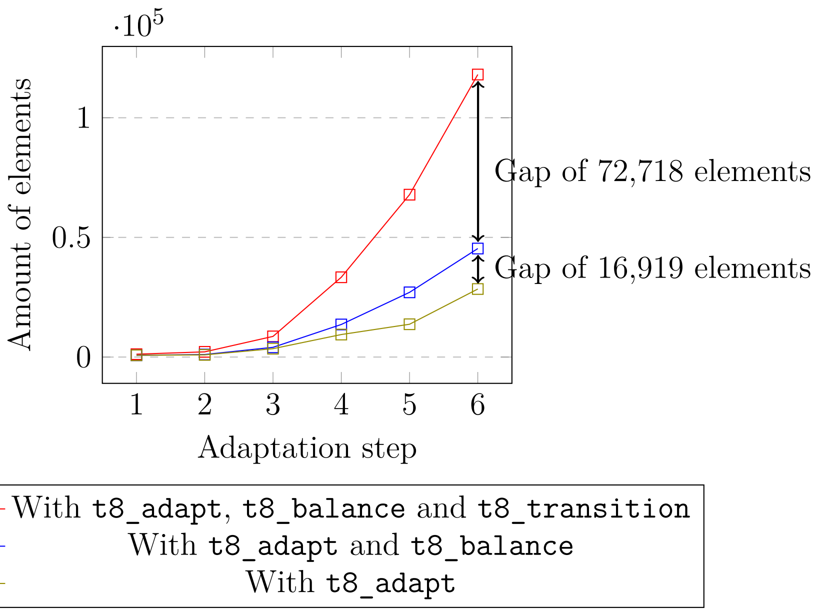


Figure 6.4: This figure shows the amount of elements in a transitioned mesh (red), a non-transitioned but balanced mesh (blue), and a mesh that is only adapted (green). The underlying refinement criterion is a band of width 2.0 around a sphere with an initial radius of 0.0 in the center of a unit cube. After each adaptation step, the radius of the sphere increases by 0.05. Every element inside the band gets refined. If an element and its siblings lie outside the band, they will be coarsened. The initial level is equal to 3.

6.2 Runtime Measurements Of `t8_transition`

Now, various runtime measurements of `t8_transition` will be discussed. As the first refinement criterion, we consider a static refinement of one-half of a unit cube, as shown in Figure 6.3. The initial level varies from 2 to 5. The measurements are conducted with and without `t8_transition`. Notice that the `balance` algorithm does not insert extra elements because the resulting mesh is already balanced. Nevertheless, `t8_balance` needs time to verify the necessary balance condition for transitioning. Therefore, the runtime measurements are also conducted with `t8_balance` to clarify the impact. The results are shown in Figure 6.5.

We see that the difference between a non-transitioned and balanced mesh is minimal until initial level 4. With initial level 4 the difference is 0.16 seconds.

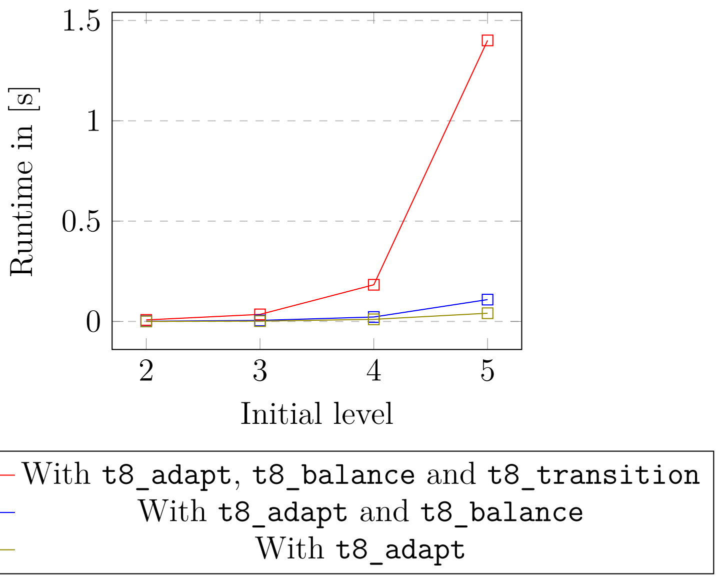


Figure 6.5: This figure shows the runtime of `t8_adapt`, `t8_balance`, and `t8_transition` in (red), the runtime of `t8_adapt` and `t8_balance` in (blue), and the runtime of just adapting the mesh via `t8_adapt` in (green). The underlying refinement region is one-half side of a unit cube. To be more precise, every element with an x-coordinate of its anchor node smaller than 0.5 is refined.

In relation to that result, it must be noted that the balance algorithm checks the balance condition and does not insert any elements for the reason that the balance condition is already fulfilled, as explained above. Whereby `t8_transition` inserts 2, 304 subelements, which equals 11.25% of the total subelements.

With initial level 5, the difference is more significant. There, the difference between transitioning and balancing the mesh amounts to 1.29 seconds. The difference between transitioning and adapting the mesh is 1.36 seconds. That means that transitioning and balancing increase the commit time by an increasing factor of 23.9 with initial level 2 to 34.2 with initial level 5. Balancing increases from factor 4.3 to 4.9 while transitioning increases from factor 4.6 to 7.6. In summary, the additional runtime of `t8_transition` is quite low up to initial level 5.

Now we consider the dynamic refinement region with a fixed band of width 2.0 around a sphere with an initial radius of zero, as presented in Chapter 6.1.

After each adaptation step, the radius increases by 0.05. Six adaptation steps are conducted. All of the elements that are entirely inside the band will be refined. All elements outside the band will be coarsened until the predefined minimum level is reached. The minimum level equals the initial level, which is 3. The runtime measurements are conducted separately for the adapt, balance, and transition algorithms. The results are shown in Figure 6.6.

We see that the most significant part of the runtime is caused by balancing the mesh with about 60.9%. Nevertheless, transitioning makes about 35.8% after step 6. It is noticeable that whilst a significant number of computations are done, compared, for instance, to the static refinement of the half side of the cube as explained above. For example, the `t8_adapt` algorithm needs to compute the centroid of each element to compute the distance of the element to the center of

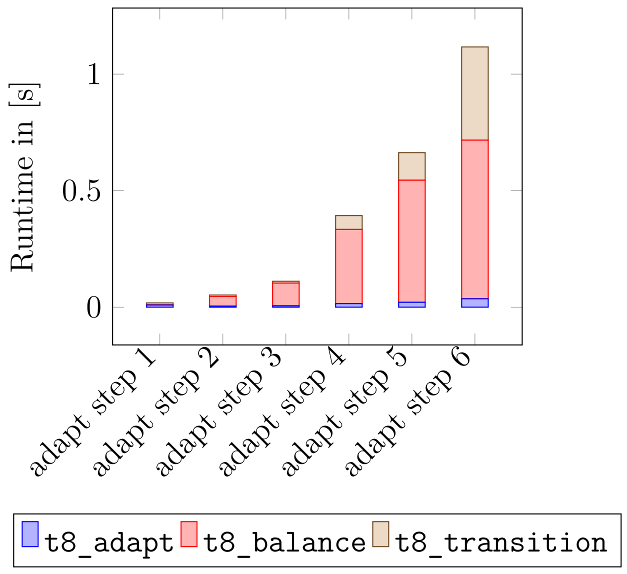


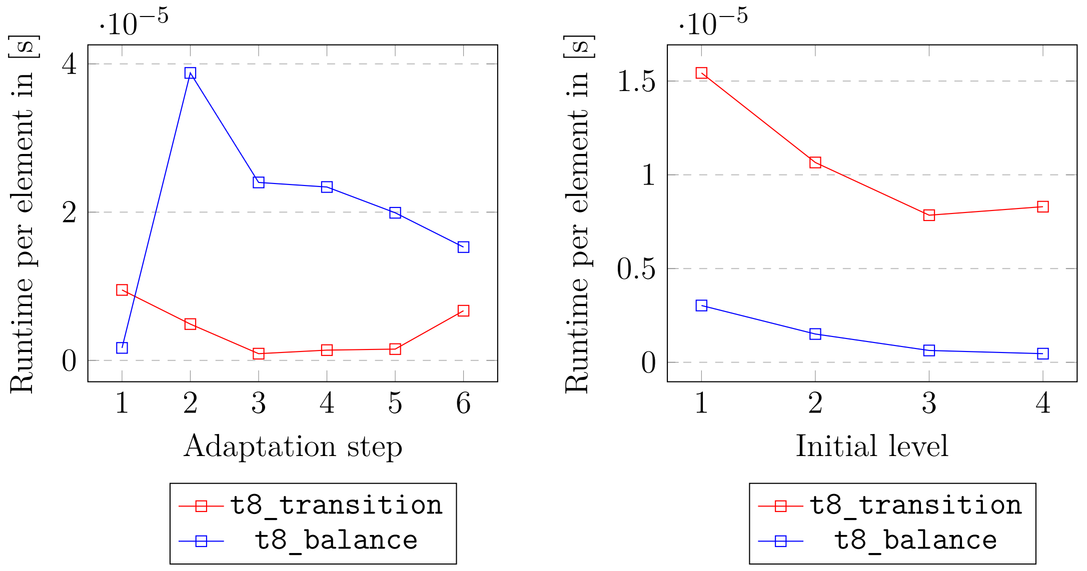
Figure 6.6: The histogram shows the individual runtimes of adapting, balancing, and transitioning the mesh. The underlying refinement criterion is a band of width 2.0 around a sphere with an initial radius of 0.0 in the center of a unit cube at (0.5,0.5,0.5). After each adaptation step, the radius of the sphere increases by 0.05. Every element inside the band will be refined. If an element and its siblings lie outside the band, they will be coarsened. The initial level is equal to 3.

the sphere. Furthermore, it computes the size of each element, with the help of an approximate diameter. Finally, it has to inspect whether the element is within the band. And, of course, `adapt` needs to refine elements and coarsen them if it is the case. Thus, it may seem questionable that the influence is so small. However, there is a major difference to `t8_balance` and `t8_transition`. `t8_adapt` does not need to check any face-neighbor relations. The significant influence of finding face-neighbors is discussed in detail in Chapter 6.3.

Furthermore, it is interesting to see how long, on average, the `balance` and `transition` algorithm needs per element to run through the mesh. It is apparent that the transitioned mesh contains more elements than a non-transitioned mesh, see Figure 6.4. To determine the runtime per element of `t8_balance` and `t8_transition`, the individual runtimes are put in relation to the amount of elements of the mesh. To make this observation more general, we look at the static half-side refinement and at the dynamic refined, as explained above. The results are shown in the following Figure 6.7.

In the dynamic case, the balancing algorithm takes relatively more time per element than transitioning. One reason is that `t8_balance` needs several times to completely fulfill the balance condition. Meanwhile, `t8_transition` always needs one round, thus one visit per element. In the static refinement, the opposite case is valid. However, it is worth mentioning that the balancing algorithm examines the balancing condition by neighboring relations and does not insert any elements. With this in mind, the impact of transitioning is low.

To sum up, transitioning always increases the amount of elements and runtimes in a significant way. Nevertheless, some specific refinement criteria may lead to a relatively small influence. A distinction must be made between the relative



(a) Dynamic refinement (b) Static refinement

Figure 6.7: This figure shows the relative runtime per element of `t8_transition` in (red) and `t8_balance` in (blue) of a dynamic refinement. The refinement criterion is a band of width 2.0 around a sphere with an initial radius of 0.0 in the center of a unit cube at (0.5,0.5,0.5). After each adaptation step, the radius of the sphere increases by 0.05. Every element inside the band will be refined. If an element and its siblings lie outside the band, they will be coarsened.

influence of transitioning on the runtime/amount of elements and the absolute. For example, in the case of the presented dynamic refinement region, the relative influence on the amount of elements and runtime is relatively small (the smallest of all presented scenarios). In the case of the half-side refinement the total time is quite fast and thus the total influence of transitioning is quite small there. Nevertheless, in relative terms, the influence is immense. Thus, checking the refinement region before transitioning can be very useful in order to not increase the amount of elements and runtime in an inefficient way.

6.3 Runtime Measurements `t8_element_leaf_face_neighbor`

This chapter discusses the runtime of finding neighbors in transitioned meshes. Therefore, we examine different refinement regions and compute each element's face-neighbors. This is done by iterating through the elements and their faces and call the `t8_element_leaf_face_neighbor` function, short `LFN`. That means that `LFN` is called six times per hexahedral element and five times per subelement. We examine the total runtime and the relative runtime per call of `LFN`.

First, we look at the total runtime of `t8_balance` and `t8_transition` at the dynamic refinement region with six adaptation steps and an increasing radius of 0.05 at each step. The initial level increases from 2 to 5. The results are shown in Figure 6.8. For the sake of clarity, the results are split into two figures.

One immediately observes that in all four cases `t8_transition` and `t8_balance` are very close. However, with a closer look, one can see that from adaptation step 2 at the latest, `t8_transition` takes more time than `t8_balance`. From a previ-

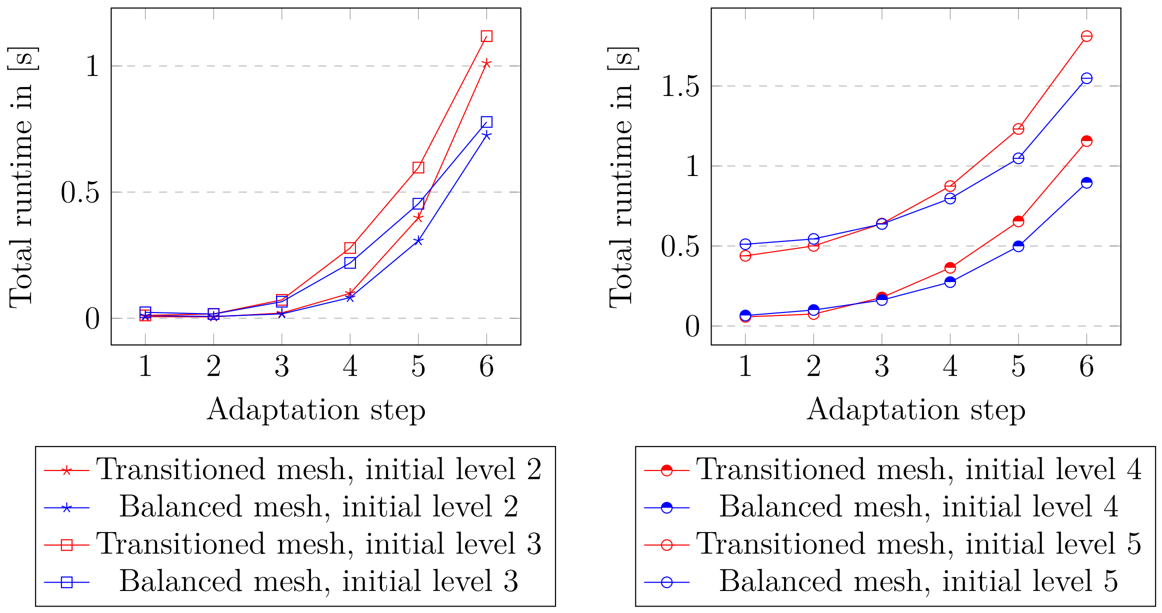


Figure 6.8: This figure shows the total runtime of `LFN` in a transitioned mesh (red) and a balanced mesh (blue). The underlying refinement criterion is a band of width 2.0 around a sphere with an initial radius of 0.0 in the center of a unit cube at (0.5,0.5,0.5). After each adaptation step, the radius of the sphere increases by 0.05. Every element inside the band will be refined. If an element and its siblings lie outside the band, they will be coarsened.

ous chapter, we know that the ratio of subelements increases with each step with a maximum of $\approx 60\%$ at step 6 with initial level 3. Figure 6.4 shows the amount of elements. Because of this, examining the relative runtime per call of `LFN` is interesting. The relative runtimes are shown in Figure 6.9. The results are also shown in two figures to obtain a better overview.

In Figure 6.9, we see that in all cases, `LFN` takes relatively less time in a transitioned mesh than in a balanced one. This can be an indicator for a relatively good performance of `LFN` in a transitioned mesh. However, of course, we must mention that balancing the mesh may take several rounds while transitioning always visits each element only once. For example, with initial level 4 `t8_balance` needs four rounds to balance the whole mesh in the sixth adaptation step, which is quite time-consuming.

Now, the already presented static refinement where one-half of the coarse mesh gets refined will be inspected. Here, we also examine the effect of different initial levels on the runtime of `LFN`. We also analyze `LFN`'s runtimes on a transitioned mesh and balanced mesh. The absolute and relative runtimes per call are shown in Figure 6.10.

In Figure 6.10, we see that the runtime of `LFN` is smaller in absolute and relative terms in the transitioned case. Compared to Figure 6.8, where the absolute runtime of a dynamic refinement region is shown, it appears that the presented static refinement of the cube might be more practical than the dynamic one. In relative terms, the runtime measurements of the static refinement are pretty high, compared to the results of the dynamic refinement region shown in Figure 6.9.

Naturally, it is worth mentioning that `LFN` can identify at most two neighbors

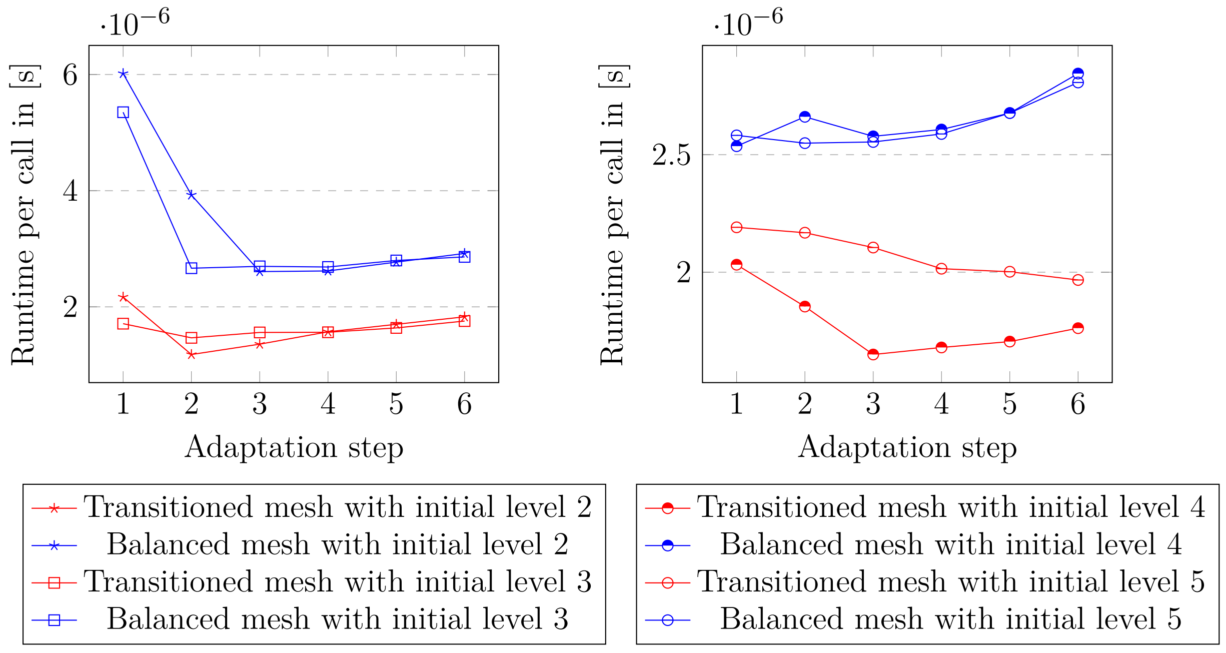


Figure 6.9: This figure shows the relative runtime per element of `t8_transition` in (red) and `t8_balance` in (blue). The underlying refinement criterion is a band of width 2.0 around a sphere with an initial radius of 0.0 in the center of a unit cube at (0.5,0.5,0.5). After each adaptation step, the radius of the sphere increases by 0.05. Every element inside the band will be refined. If an element and its siblings lie outside the band, they will be coarsened.

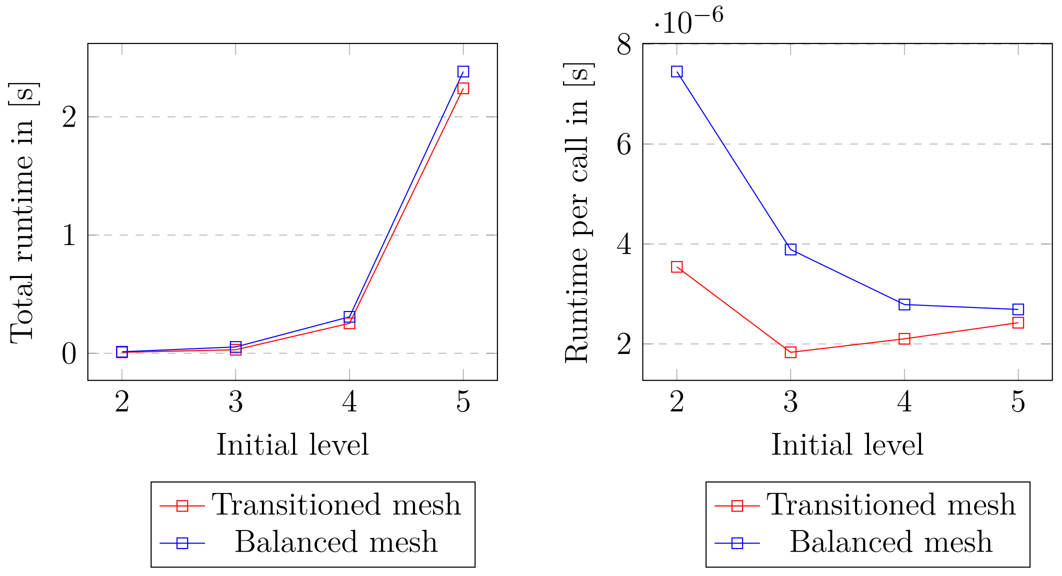


Figure 6.10: This figure shows `LFN` on a transitioned mesh in (red) and on a balanced mesh in (blue). The left side depicts the absolute runtimes. The right side shows the relative runtimes per call. The underlying refinement region is one-half side of a unit cube. To be more precise, every element with an x-coordinate of its anchor node smaller than 0.5 is refined.

in a transitioned mesh and four neighbors at most in a balanced mesh.

To summarize, the runtime of `LFN` in a transitioned mesh is comparable to the runtime of `LFN` in a balanced mesh. The relative runtime per call is even lower in a transitioned mesh.

6.4 Parameters For Mesh Quality

In this chapter, some mesh quality parameters for the transitioned mesh are discussed. A good mesh quality can be a crucial factor in the sense of accuracy and efficiency for solving PDE's. Nevertheless, many other factors influence the accuracy, for example, the type of geometry being discretized or details of the solution. However, a good mesh quality is fundamental for a good simulation. In [28], Knupp defines mesh quality for PDE applications as follows:

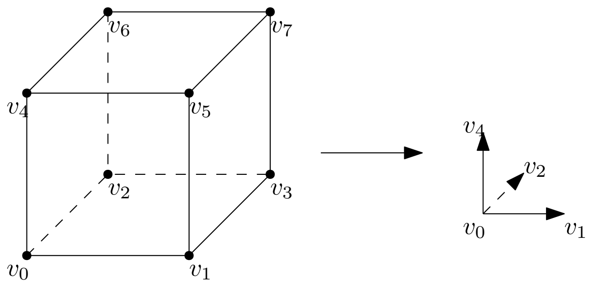
"Mesh quality concerns the characteristics of a mesh that permit a particular numerical PDE simulation to be efficiently performed, with fidelity to the underlying physics, and with the accuracy required for the problem."

Due to the dependence of mesh quality on the underlying simulation, no generally applicable parameter determines whether the elements of a mesh are of good or poor quality.

In literature, there are several mesh quality parameters for hexahedral and tetrahedral meshes but there are very poor for pyramids and prism. In the Verdict Geometry Quality Library [29], a number of quality metrics, like the aspect–ratio and the Jacobian, are given for triangles, quadrilaterals, tetrahedra, and hexahedra. For pyramids, there is only one simple volume metric given.

6.4.1 Jacobian-Based Quality Metrics

First, we introduce the Jacobian (J) and the scaled Jacobian (J_S) for hexahedra. For computation, the determinant of edge vectors of the hexahedron needs to be taken. Figure 6.11 illustrates the computation of J and J_S of a hexahedron as in the Verdict library of mesh quality metrics [29]. If $J > 0$, respectively $J_S > 0$, the element is valid due to positive volume. If $J < 0$ ($J_S < 0$), the element is inverted and therefore invalid.



$$det(A_0) = \overrightarrow{v_1v_0} \cdot (\overrightarrow{v_2v_0} \times \overrightarrow{v_4v_0})$$
$$J = min\{det(A_i), i = 0, \ldots, 7\}$$

$$det(\hat{A}_0) = \frac{\overrightarrow{v_1v_0}}{\|\overrightarrow{v_1v_0}\|} \cdot \left(\frac{\overrightarrow{v_2v_0}}{\|\overrightarrow{v_2v_0}\|} \times \frac{\overrightarrow{v_4v_0}}{\|\overrightarrow{v_4v_0}\|} \right)$$
$$J_S = min\{det(\hat{A}_i), i = 0, \ldots, 7\}$$

Figure 6.11: Computation of the Jacobian J and the scaled Jacobian J_S by the example of a regular hexahedron. A_i with $i = 0, \ldots, 7$ is called the *Jacobian matrix.*

In Figure 6.11, we see that $J \in [-\infty, \infty]$ and $J_S \in [-1, 1]$. Thus, J depends on the element's size (level respectively). This means, for example, that a regular hexahedron with level l has a greater Jacobian than a regular hexahedron with level $l+1$. Thus, J states the validity of an element, but it does not necessarily replicate the quality of an element. Therefore, it makes sense to consider the scaled Jacobian J_S. If $J_S = 1$ the nodes are in optimal position regarding their direct node-neighbors. In [41], the scaled Jacobian is discussed in detail and it is stated that if $J_S < 0.2$, the quality of the hexahedron is questionable.

In [7] they introduced a general approach to computing J_S for elements where every node has three neighbors. Notice that if the underlying element is in a pyramid shape the Jacobian matrix for the apex is not a square matrix. The apex is the only node in a pyramid that has four node-neighbors. Therefore, the computation of J_S is not as straightforward as for a tetrahedron, for example.

In [33], they introduced the element normalized scaled Jacobian J_{ENS} that is suitable for measuring the quality of tetrahedra, pyramids, prisms, and hexahedra. J_{ENS} ranges in $[-1, 1]$, where a negative value states that the element is invalid. The following equation 32, shows the definition of J_{ENS} of a node i.

$$J^i_{ENS} = \begin{cases} (1+k^e) - J^i_S & \text{if } J^i_S > k^e \\ J^i_S / k^e & \text{if } -k^e \le J^i_S \le k^e \\ -(1+k^e) - J^i_S & \text{if } J^i_S < -k^e \end{cases} \tag{32}$$

The value of constant k^e depends on the element type under consideration. To obtain k^e, the theory of perfectly shaped elements needs to be introduced.

It may be intuitive to think that equilateral elements are always the perfect shaped elements. In fact, this is true for tetrahedra and hexahedra but not for prism and pyramids as shown in [33].

However, the perfectly shaped pyramid should be inside the cone containing the equilateral tetrahedron. The same goes for a perfectly shaped prism with the cylinder that contains the equilateral hexahedron. This cone is $h = \frac{\sqrt{6}}{3}$ high and has diameter of $d = \frac{\sqrt{12}}{3}$. With the use of trigonometry, it can be deduced that the perfectly shaped pyramid has a basal square face with edge size $b = \frac{\sqrt{6}}{3}$, which is equal to the height h. The edges connected to the apex are all the same size $s = 1$.

With the kind permission of the author C. Lobos of [33] a perfectly shaped tetrahedron, pyramid, hexahedron and prism is shown in Figure 6.12.

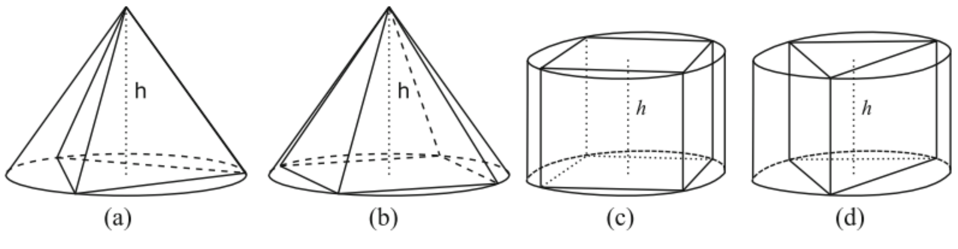


Figure 6.12: This figure shows a perfectly shaped tetrahedron in (a) and a pyramid in (b) inside a cone of height $h = \frac{\sqrt{6}}{3}$ and diameter $d = \frac{\sqrt{12}}{3}$. The perfectly shaped hexahedron in (c) and prism in (d) are shown inside a cylinder with height $h = 1$ and diameter $d = \sqrt{2}$. Source: C. Lobos Figure 3.11 in [33]

In order to compute the constant k^e of equation (32) we need to calculate the J_S of each node of the perfectly shaped element of the corresponding type. In this chapter, we only consider pyramid-shaped elements because of the composition of transition cells. Computing J_S for pyramids is more complex than for the other types because of the node connectivity of the apex. The apex is connected to all four nodes of the basal square face. Thus, J_S cannot be directly computed for the apex. The approach of [33] is to decompose the perfectly shaped pyramid into four virtual tetrahedra, each with the apex and three basal nodes. Figure 6.13 shows this decomposition. Then, J_S needs to be computed for all four decompositions. However, the chosen J_S will be the smallest positive value of all four. If all four values of J_S for the apex are negative, the whole pyramid is inverted.

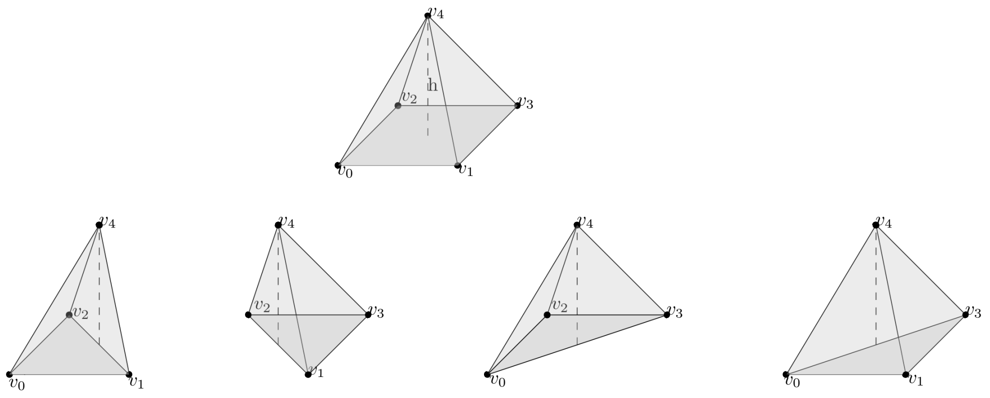


Figure 6.13: Decomposition of a pyramid into four different tetrahedra.

The resulting values of J_S are $K^B = \frac{\sqrt{6}}{3}$ for the basal nodes and $K^A = \frac{2\sqrt{6}}{9}$ for the apex. The quality of a pyramid-shaped element P_q is given in the following definition:

Definition 6.4.1. Let P be a pyramid shaped element. The quality of P is defined as follows:

$$P_q = \begin{cases} min_i\{J^i_{ENS}\} & \text{if } \forall\ i: \quad J^i_{ENS} > 0 \\ max_i\{J^i_{ENS}\} : J^i_{ENS} < 0 & \text{if } \exists\ i: \quad J^i_{ENS} < 0 \end{cases} \tag{33}$$

J^i_{ENS} thereby denotes the element normalized scaled Jacobian as shown in equation (32) with constant $k^e = \frac{\sqrt{6}}{3}$ if i is a basal node and $k^e = \frac{2\sqrt{6}}{9}$ if node i is the apex of the pyramid.

Now, we compute the quality of the two different shaped pyramids inside a transition cell based on Definition 6.4.1.

First, we start with the pyramid type whose basal face equals the face of its parent element. This type of pyramid is called *type 1*, in the following. The pyramids whose basal face is at the face with a hanging node are called *type 2*. The two types of pyramids inside a transition cell are shown in Figure 6.14.

The size of basal edges of *type 1* pyramids is then, of course, $b_{\text{type 1}} = 1$. The apex is of height $h_{\text{type 1}} = 0.5$ because it is the midpoint of the parent element. The edges that are connected to the apex are of size $s_{\text{type 1}} = \frac{\sqrt{3}}{2}$. Computing J^i_S for all basal nodes results in:

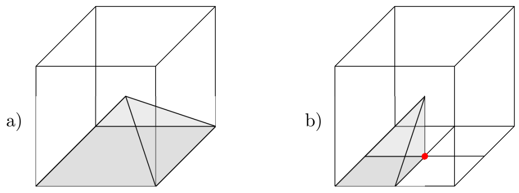


Figure 6.14: Illustration a) shows a pyramid of *type 1*. Its base face equals the face f_4 of its parent element. Illustration b) shows a pyramid of *type 2*. The hanging node on the face is marked in red. One can directly see the differing sizes of edges connected to the apex.

$$\begin{aligned} J_S^i &= \frac{\sqrt{3}}{3} \text{ for } i = 0, \dots, 3 \\ \Rightarrow J_{ENS}^i &= \frac{J_S^i}{k^e} = \frac{\frac{\sqrt{3}}{3}}{\frac{\sqrt{6}}{3}} = \frac{\sqrt{2}}{2}. \end{aligned} \tag{34}$$

For the apex we get the following results:

$$\begin{aligned} J_S^4 &= \frac{4\sqrt{3}}{9} \\ \Rightarrow J_{ENS}^4 &= \left(1 + \frac{2\sqrt{6}}{9}\right) - \frac{4\sqrt{3}}{9} = 1 + \frac{2\sqrt{2} - 4}{3\sqrt{3}} \end{aligned} \tag{35}$$

Therefore, the quality of pyramid *type 1* is:

$$P_q^{type\ 1} = \frac{1}{\sqrt{2}} \approx 0.71 \tag{36}$$

Now, we compute the quality of *type 2* pyramids. The basal edges are of size $b_{type\ 2} = 0.5$. The height is again $h_{type\ 2} = 0.5$. The edges connected to the apex are not all of the same size. The shortest edge equals the height of the pyramid, and the longest edge equals the size of apex-connected edges of *type 1* pyramids. The two other edges are of size $\frac{1}{\sqrt{2}}$. Analogous to the above calculation, we get the following results:

$$\begin{aligned} J_S^i &= \frac{\sqrt{3}}{3} \Rightarrow J_{ENS}^i = \frac{\sqrt{2}}{2} \text{ for } i = 1, \dots, 3 \\ J_S^4 &= \frac{\sqrt{3}}{6} \Rightarrow J_{ENS}^4 = \frac{\frac{\sqrt{3}}{6}}{\frac{2\sqrt{6}}{9}} = \frac{3\sqrt{2}}{6} \end{aligned} \tag{37}$$

Thus the quality of pyramid *type 2* is given by:

$$P_q^{type\ 2} = \frac{3\sqrt{2}}{6} \approx 0.53 \tag{38}$$

We see that the quality of *type 2* pyramids is not as good as that of *type 1* pyramids. Therefore, it might be advantageous to minimize the use of transition cells with a high number of *type 2* pyramids.

6.4.2 Aspect-Ratio Based Quality Metrics

In this chapter, we look at aspect-ratio (AR) metrics. The most common AR for hexahedra is the quotient between minimum and maximum edge length $AR = \frac{e_{min}}{e_{max}}$. This quality metric can be used to examine whether the hexahedron is stretched or flattened. In order to analyze the validity of a hexahedron, the AR might fail to detect a poor-quality element because it does not take the volume into account, compared to Jacobian-based quality metrics introduced in the previous Chapter 6.4.1. Imagine an equilateral hexahedron that is completely flat. The AR would not be very bad but the element is invalid because of its empty volume. According to [33], AR can be a good metric for measuring pyramid and prism quality. Nevertheless, it is worth mentioning that even if the AR is equal to one, it does not describe the perfectly shaped pyramid/prism, see Figure 6.12 b) and d). So, the AR might be useful for detecting stretched or flattened elements but does not reproduce their validity.

In the previous chapter, we examined the validity of the two types of subelements. In the next step the AR will be computed. For this, we first calculate the standard AR. After that, we compare it with the Element Normalized Aspect Ratio (AR_{EN}) introduced in [33].

The standard AR for a pyramid of *type 1* is:

$$AR^{\textit{type 1}} = \frac{\frac{\sqrt{3}}{2}}{1} \approx 0.866 \tag{39}$$

Compared to this, the standard AR for pyramid of *type 2* is:

$$AR^{\textit{type 2}} = \frac{\frac{1}{2}}{\frac{\sqrt{3}}{2}} = \frac{\sqrt{3}}{3} \approx 0.577 \tag{40}$$

We see that the results confirm the quality measurement of the Jacobian-based metrics. The pyramid of *type 2* is more distorted than the pyramid of *type 1.*

Now, we present the Element Normalized Aspect Ratio (AR_{EN}).

$$AR_{EN} = \begin{cases} (1+k^e) - AR & \text{if } AR > k^e \\ \frac{AR}{k^e} & \text{if } AR \leq k^e \end{cases} \tag{41}$$

For pyramids the constant $k^e = \frac{\sqrt{6}}{3}$ as derived in the previous chapter. So, the AR_{EN} for pyramids of *type 1* is:

$$AR_{EN}^{\textit{type 1}} = 1 + \frac{\sqrt{6}}{3} - \frac{\sqrt{3}}{2} \approx 0.95 \tag{42}$$

and the AR_{EN} for pyramids of *type 2* is:

$$AR_{EN}^{\textit{type 2}} = \frac{\frac{\sqrt{3}}{3}}{\frac{\sqrt{6}}{3}} = \frac{\sqrt{2}}{2} \approx 0.707 \tag{43}$$

We see that the AR_{EN} is greater for the pyramid of *type 1* than for *type 2.* For *type 1*, it is nearly perfect, which suggests that the distortion is very small compared to a perfectly shaped pyramid.

In summary, one can say that with the Jacobian-based metrics and the presented aspect-ratio metrics the quality of *type 1* pyramids seems to be better. However, neither pyramid is perfectly shaped. Therefore, transitioning a mesh always decreases the quality of the mesh. Nevertheless, if the amount of *type 2* pyramids is minimized the entire quality loss is minimized, too.

7 Outlook

In this chapter, possible improvements of `t8_transition` are discussed. Thus far, we only focused on hanging nodes on faces. Unfortunately, this is not the only hanging node type in hexahedral meshes. Another type of hanging nodes occurs if a mesh only fulfills the face-balance condition but is not edge-balanced. That means there are hanging nodes on edges, another yet to be discussed kind of hanging node. These particular types of hanging nodes are discussed in Chapter 7.1.

Then, in Chapter 7.2, the problem of hanging edges that occur inside the presented transition cells will be considered. There, we present and discuss a possible solution strategy, based on the results from Chapter 7.1, to adjust the whole transition process in order to receive a conformal mesh with no hanging nodes and edges.

In the last chapter, 7.3, all results and findings pertaining to this thesis will be summarized.

7.1 Edge-Balancing

This chapter will focus on the topic of edge-balancing. When introducing the balance condition in Chapter 2, we only looked at the face-balance condition. That means that the levels of two elements that are each other's face-neighbor differ at most by ± 1. But in a hexahedral mesh there are three different possible types of neighborhoods (face, edge, and node), see Figure 7.1.

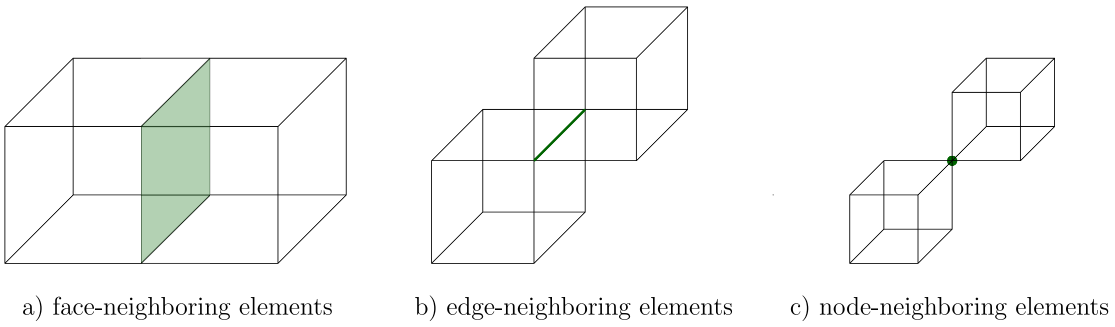

a) face-neighboring elements
b) edge-neighboring elements
c) node-neighboring elements

Figure 7.1: This figure shows the three different possible neighborhood types in an octree.

We see that not only face-neighboring but also edge-neighboring elements can cause hanging nodes. So, even if the face-balance condition is fulfilled, two scenarios with hanging nodes that are not covered by the presented transition cells can occur. The reason for this is that the level of a face-neighbor of an element's face-neighbor can differ at most by ± 2. For an illustration of this, see Figure 7.2.

One possible way to handle edge-hanging nodes is to create a lot of more different transition cells that can deal with all these possible scenarios. However, this results in over 8000 different cases that must be covered. Therefore, it appears important to restrict the amount of possible scenarios with edge-hanging nodes. So, due to this enormous overload of transition cells and a probably high loss of mesh quality, we discuss one possible way to solve the edge-balance condition in `t8code`. If the edge-balance condition is fulfilled scenarios like those shown in

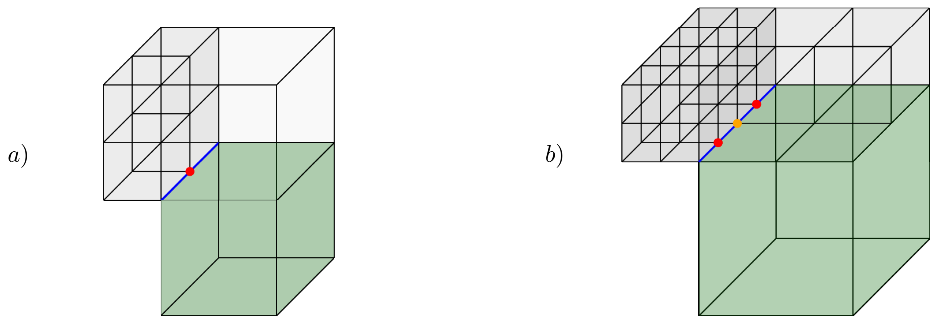


Figure 7.2: This figure shows two scenarios that can occur in a face-balanced mesh. In a), we see a hanging node in red on a blue edge of the green hexahedron. This hanging node occurs on the edge of the green hexahedron and is thus ignored by `t8_transition`. Notice that the elements in scenario a) fulfill the edge-balance condition. In b), we see three hanging nodes on one blue edge of the green hexahedron. The one in the middle, labeled in orange, is the kind of hanging node we know how to deal with. The other two hanging nodes depicted in red still need to be dealt with.

Figure 7.2 b) are not possible anymore. That eliminates already $\sum_{i=1}^{12} \binom{12}{i} = 4094$ different cases.

In order to adjust `t8_balance` to create edge-balance, the first thing that needs to be done is to implement a method that identifies edge-neighbors.

In [24], an edge structure for octrees is introduced that stores neighbor information of each edge in a five integers long array. Thereby, the first integer states whether the edge is split or not, and the following four integers describe the potential neighbors in terms of their unique ID; in `t8code`, it would be the Morton index. For more details, see [24]. Naturally, edge-balancing leads to an additional amount of hexahedral elements.

So, we see that the edge-balancing condition is a basic requirement in order to limit the amount of transition cells so that the impact of transitioning is minimized. Nevertheless, remember that scenario a) in Figure 7.2 also leads to an additional amount of 4094 different cases. Therefore, it seems reasonable to restrict further hanging-node-producing scenarios but edge-balancing. Some possible restrictions are discussed in the following Chapter 7.2. There, we assume the underlying mesh to fulfill the edge-balance condition.

7.2 Hanging Edges

In this chapter one disadvantage of the presented approach to solving hanging faces in a hexahedral mesh, the so-called hanging edges, is discussed.

At this point, it is worth mentioning that there are numerical solvers that are not concerned with hanging edges. They might be concerned only with the conformal node-connectivity.

In the presented transition cells, hanging edges occur when split subelements are face-neighbored with a non-split subelement, see Figure 3.4. That means there are edges that lie in the face of another element. This problem can only be solved by changing the structure of the transition cell. In the following, this modified transition cell is called *transition cell A*.

In order to keep the additional amount and complexity of elements inside a transition cell as small as possible, one idea is to limit the possible scenarios with hanging nodes to transition cells with only one hanging face or one hanging edge. Notice that for this restriction, the `balance` algorithm needs to be modified. Furthermore, this restriction will also, as the edge-balance condition, lead to an increasing amount of hexahedral elements and, thus, additional computational effort.

So, in the following discussion, it will be assumed that the mesh is edge-balanced and modified in such a way, that an element has at most one hanging node at a face (which induces four hanging edges) or one hanging node at an edge.

In the following, the modification of the transition cell is explained. The aim is to translate the apex of the opposite pyramid, which corresponds to the node in the center, to the hanging node inside the face. This type of pyramid is called a *type 1* pyramid, in the following. The apexes of the smaller pyramids, called *type 2* pyramids are translated to an opposite-lying node. The remaining parts of the transition cell is filled with tetrahedra. This results in a transition cell containing five pyramids and four tetrahedra. Thus, we do not increase the amount of elements inside a transition cell of this specific type. Furthermore, we do not add an additional node. To make this idea clear, see Figure 7.3.

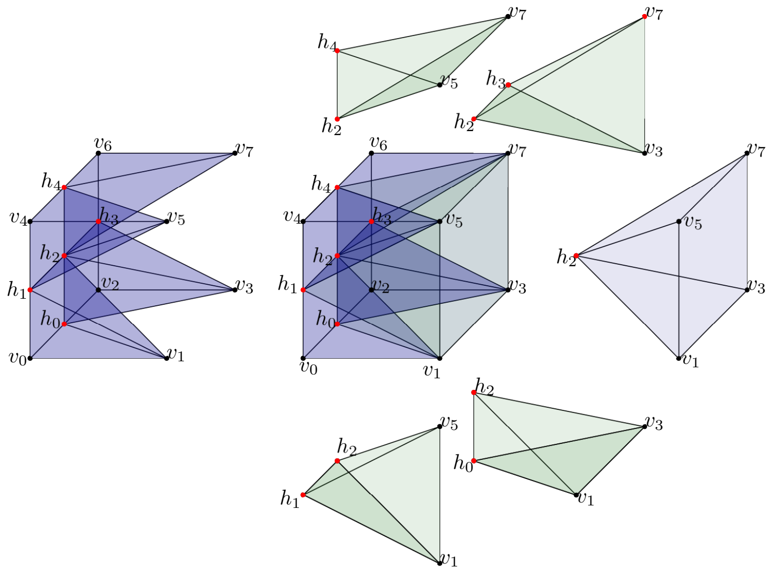


Figure 7.3: This figure shows an exploded view of *transition cell A*. *Transition cell A* is a modified transition cell with no hanging edges inside. It consists of five pyramids, shown in dark blue and blue, and four tetrahedra, shown in green. The hanging nodes are marked in red.

Notice that the modified transition cell shown in Figure 7.3 does not contain hanging edges inside the transition cell but may produce hanging edges to the face of neighboring elements. Therefore, we need to transition the face-neighbored element as well. In Chapter 7.1 an approach to store edge-data information to identify leaf-edge-neighbors was presented. With this hypothetical function, one could modify the `transition` algorithm in the sense of not only detecting face-hanging nodes but also edge-hanging nodes, as shown in Figure 7.2 a). Therefore, a second type of transition cell needs to be introduced. We call this type of transition cell *transition cell B*. An exploded view of this transition cell can be seen in Figure 7.4. This transition cell consists of four pyramids with their apex at the hanging node.

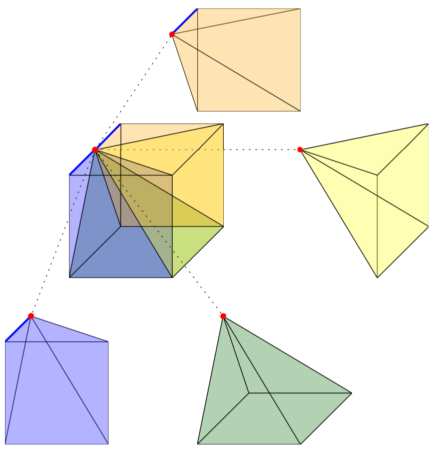


Figure 7.4: This figure shows an exploded view of *transition cell B*. *Transition cell B* consists of four pyramids with their apex at the edge-hanging node.

We see that both, the transition cell that eliminates hanging nodes at one face, *transition cell A*, and the transition cell that eliminates a hanging node at one edge, *transition cell B*, generate edges on their outer faces. However, due to the construction, these edges match with the other transition cell. To make this clear: If an element consists of one hanging node on an edge, it follows that the faces that are adjacent to this edge are the outer faces of a transition cell. The edges inside these outer faces of *transition cell A* then match the edges of *transition cell B*.

Thus, the presented restrictions allow only the following three scenarios:

(i) One element has five hanging nodes that are all located at one face.
(ii) One element has one hanging node in total at one edge. (44)
(iii) One element has no hanging nodes.

Thus, for a mesh restricted to the scenarios given in (44), the *transition cells A* and *B* would be sufficient to make this modified mesh conformal. Another advantage of that kind of modified mesh is that a lot of functions, like the `LFN`-function could be implemented in a more hard-coded way because fewer options are possible. This would significantly accelerate the runtime of transitioning the mesh.

Furthermore, looking at the refinement region, discussed in Chapter 6, where one-half of a cube is refined, the use of *transition cell A* would be sufficient to make the mesh conformal. So, this could result in a very low implementation effort and a manageable additional amount of elements.

7.3 Conclusion

We have seen that many possible scenarios can cause hanging nodes. In order to keep the expenditure as small as possible, potential restrictions were presented.

Foremost, enabling edge-balancing would be an essential task for the future. Moreover, additional restrictions can be useful in order to keep the amount of transition cells as small as possible. For example, the presented restrictions in the previous Chapter 7.2, only require two different transition cells in order to make a mesh conformal.

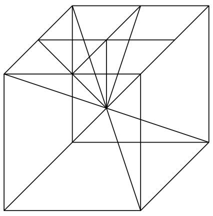
Therefore, checking the underlying refinement regions before transitioning the mesh can be useful. If there are clear boundaries between refined regions and non-refined regions, transitioning the mesh with transition cells can be very useful. For example, the presented refinement region in Figure 6.3, where just one-half of a cube is refined would be quite simple to transition. There is only one type of transition cell needed. This might lead to a very low impact of transitioning.

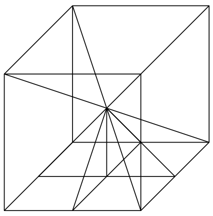
If the refinement region is curved, more different types of transition patterns may be needed. It still seems useful to restrict the amount to minimize the impact of transition. This can exemplary be done by "straightening" the refinement region to receive straight boundaries. In terms of the dynamic refinement region presented in Chapter 6, it may be useful to modify the sphere into a cube.

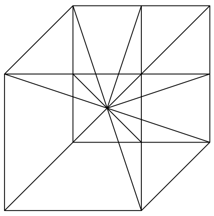
Naturally, restricting possible scenarios for occurring hanging nodes always leads to an increasing amount of hexahedral elements. Consequently, it is always a trade-off between modifying the mesh to restrict possible scenarios of hanging nodes with the cost of additional elements and a manageable number of transition patterns in order to make the mesh conformal. Therefore, not only the additional amount of elements must be considered but also a potential loss of quality as analyzed in Chapter 6.4.

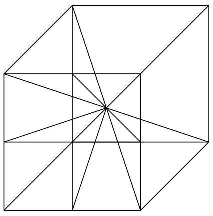
The pictures in this theses were made by the open source software IPE `https://ipe.otfried.org`.

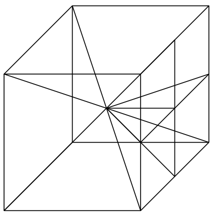
A Appendix

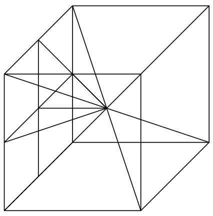

$t = 1$

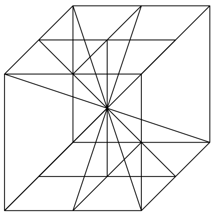

$t = 2$

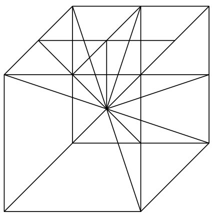

$t = 4$

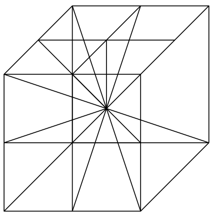

$t = 8$

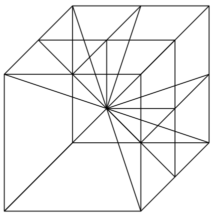

$t = 16$

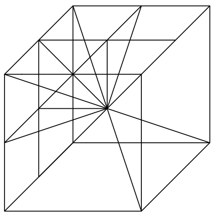

$t = 32$

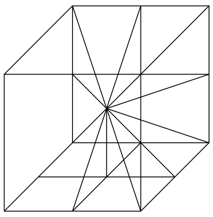

$t = 3$

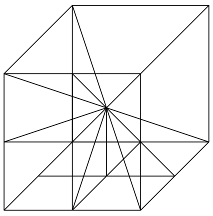

$t = 5$

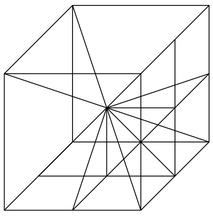

$t = 9$

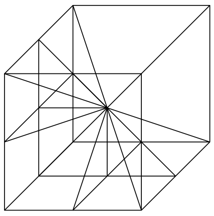

$t = 17$

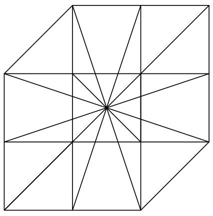

$t = 33$

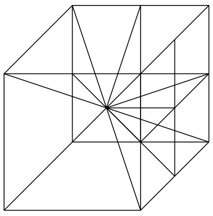

$t = 6$

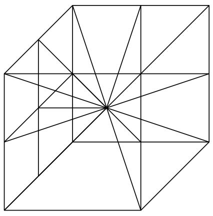

$t = 10$

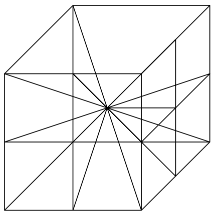

$t = 18$

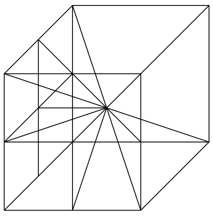

$t = 34$

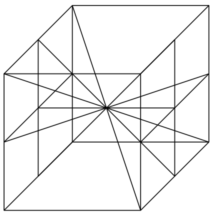

$t = 12$

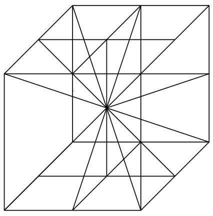

$t = 20$

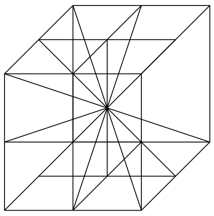

$t = 36$

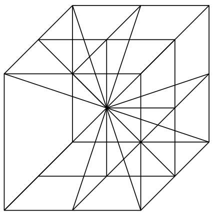

$t = 18$

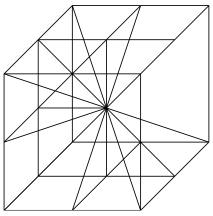

$t = 34$

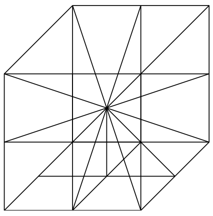

$t = 48$

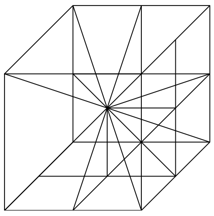

$t = 7$

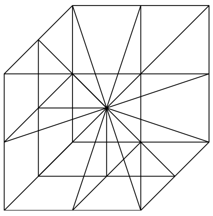

$t = 11$

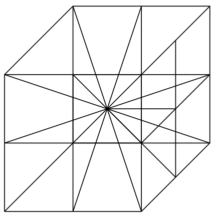
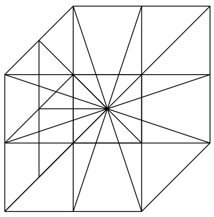
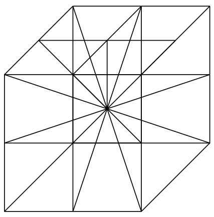
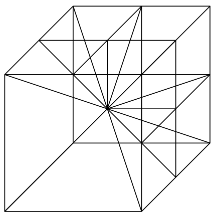
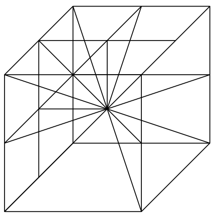
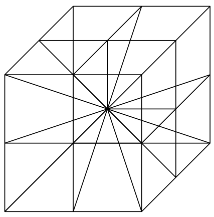
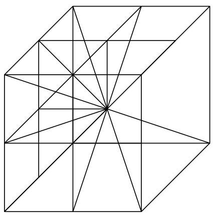
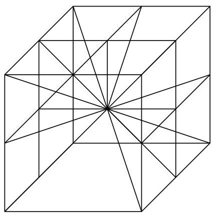
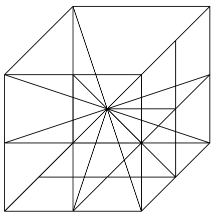
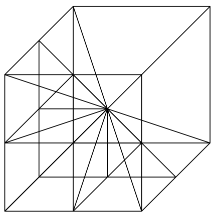
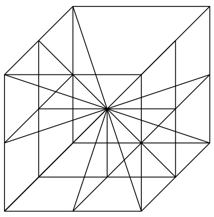
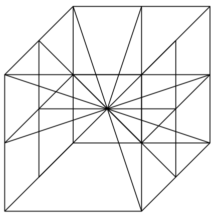
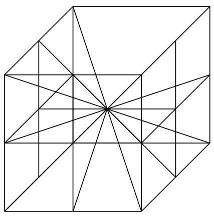
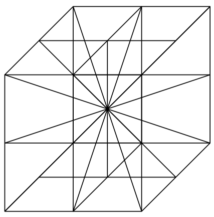
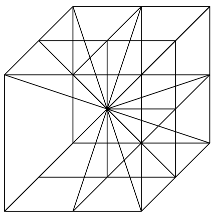
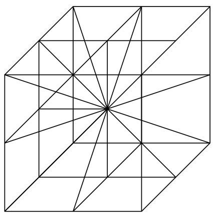
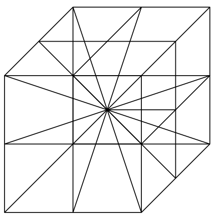
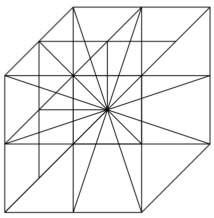
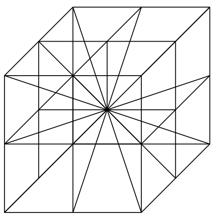
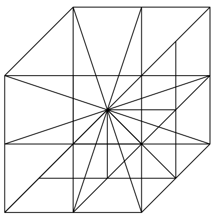
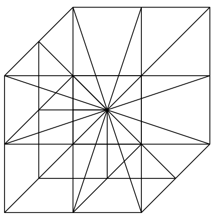
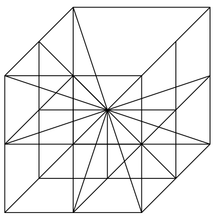
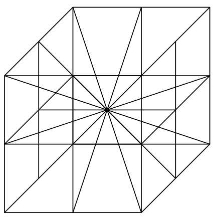
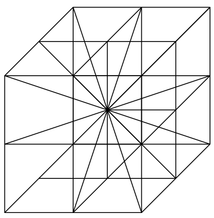

$t = 19$


$t = 35$


$t = 14$


$t = 22$


$t = 38$

$t = 28$

$t = 44$

$t = 13$

$t = 21$

$t = 37$

$t = 25$

$t = 41$

$t = 49$

$t = 26$

$t = 42$

$t = 50$

$t = 52$

$t = 56$

$t = 15$

$t = 23$

$t = 39$

$t = 29$

$t = 45$

$t = 57$

$t = 30$

$t = 46$

$t = 58$

$t = 60$

$t = 31$


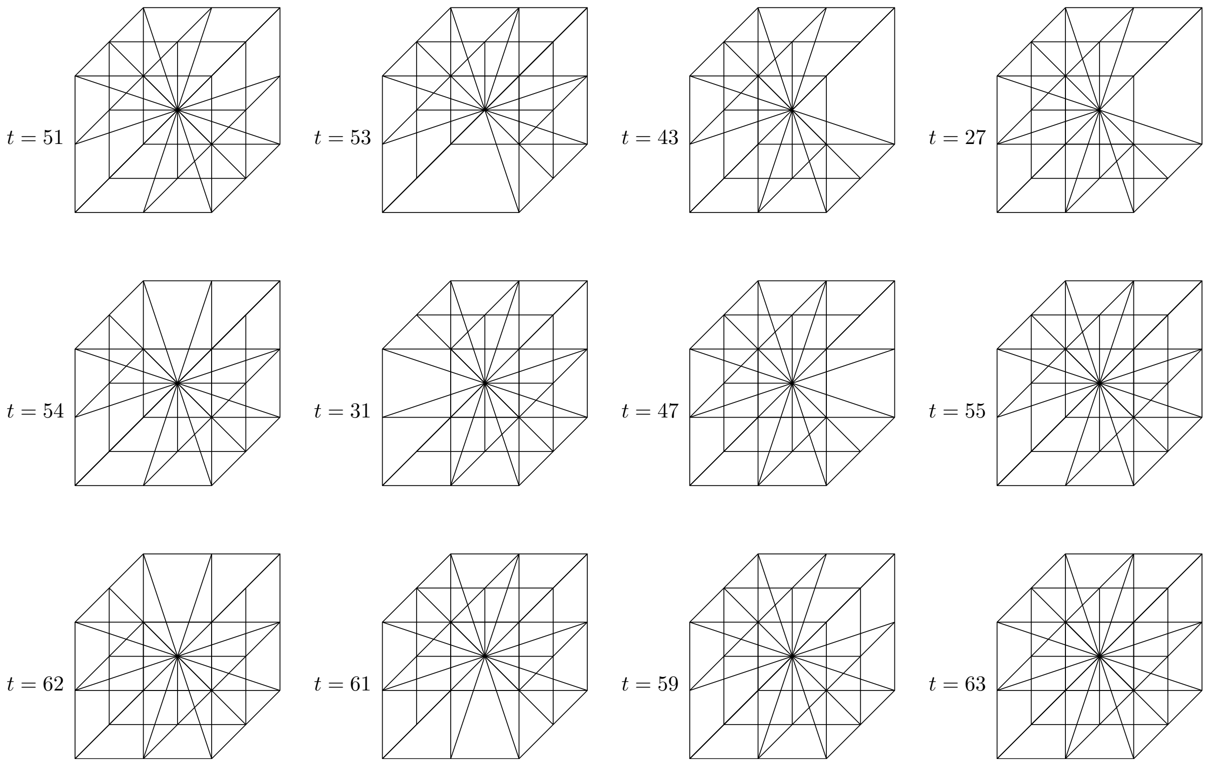


Figure A.1: A complete list of all possible transition types $t \in \{1, \ldots, 63\}$

Definition A.0.1. The face duals of a regular hexahedron with faces $f_0, \ldots, f_5$ are given by:

$$f_0 \xrightarrow{dual} f_1,\ f_1 \xrightarrow{dual} f_0,\ f_2 \xrightarrow{dual} f_3,\ f_3 \xrightarrow{dual} f_2,\ f_4 \xrightarrow{dual} f_5,\ f_5 \xrightarrow{dual} f_4 \quad (45)$$

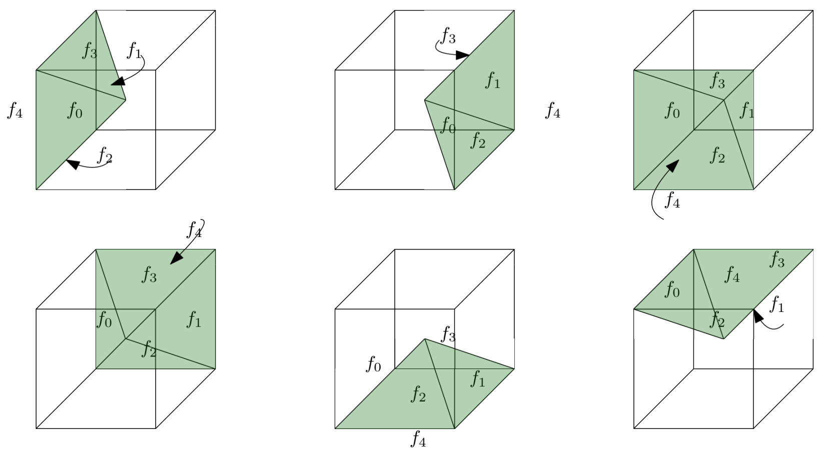


Figure A.2: This figure shows the face labels of a non-split subelement.

Algorithm 7: Compute the location array of a subelement S

```
Data: A subelement $S$
Result: Location array $L$ of subelement $S$
t8_element_get_location_of_sub($S$) Begin
    // Compute binary array of transition type
    $b[] \leftarrow$ transition_type_to_binary_array($S$.transition_type)
    // Compute cumulative array of the binary array
    $b_{\text{cum}}[] \leftarrow$ compute_cumulative_array($b[]$)
    // Create array for the subelement_id_type and initialize with zero
    $sub\_id\_type\_array[] = \{0,0,0\}$
    // Use $b_{\text{cum}}[]$ to determine the face number of the parent on which
       the subelement lies
    if $S$.subelement_id $< b_{cum}[0]$ then
        face $= 0$
    else
        for $0 \leq$ P8EST_FACES $- 1$ do
            if $b_{cum}[i] \leq S$.subelement_id $< b_{cum}[i+1]$ then
                face $= i+1$
                break
            end
        end
    end
    // Determine whether the subelement is split or not
    if $b$[face] $= 0$ then
        split $= 0$
    else
        split $= 1$
    end
    // Determine the subelement_id_type, if split $= 1$
    if split $= 1$ then
        // Check whether the subelement is on the right or left side
        // This can only be the case for faces $f_2, \ldots, f_5$
        if face $> 1$ then
            if $S$.subelement_id $+ 1 =$
             $b_{cum}$[face] OR $S$.subelement_id $+ 3 = b_{cum}$[face] then
                $sub\_id\_type\_array[0] = 1$
            end
        end
        // Check whether the subelement is on the front or back
        if face $\leq 1$ then
            if $S$.subelement_id $+ 1 =$
             $b_{cum}$[face] OR $S$.subelement_id $+ 3 = b_{cum}$[face] then
                $sub\_id\_type\_array[1] = 1$
            end
        else
            if face $\geq 4$ then
                if $S$.subelement_id $+ 1 =$
                 $b_{cum}$[face] OR $S$.subelement_id $+ 2 = b_{cum}$[face] then
                    $sub\_id\_type\_array[1] = 1$
                end
            end
        end
    end
    // Check whether the subelement is on the top or bottom
    if face $< 4$ then
        if $S$.subelement_id $+ 1 = b_{cum}$[face] OR $S$.subelement_id $+ 2 =$
         $b_{cum}$[face] then
            $sub\_id\_type\_array[2] = 1$
        end
    end
    // Calculate the subelement id type out of the sub_id_type_array
    for $0 \leq i < 3$ do
        if $sub\_id\_type\_array[i] = 1$ then
            $sub\_id\_type \mathrel{+}= 2^{2-i}$
        end
    end
end
```

Algorithm 8: Convert transition type t to a binary array $b[]$

```
Data: Transition type $t$
Result: Binary array $b$ of transition type $t$
1 transition_type_to_binary_array($transition\_type$) begin
2     for $0 \leq i <$ P8EST_FACES do
3         $b[(\texttt{P8EST\_FACES} - 1) - i] \leftarrow (transition\_type$ and $(1 << i))\ >> i$
4     end
5 end
```

Algorithm 9: Compute cumulative array $b_{cum}[]$ of the binary array $b[]$

```
Data: Binary array $b[]$
Result: Cumulative array $b_{cum}$ of binary array $b[]$
1 compute_cumulative_array($b[]$) begin
2     $b_{\mathrm{cum}}[0] = b[0] * 3 + 1$
3     for $0 \leq i <$ P8EST_FACES do
4         $b_{\mathrm{cum}} \leftarrow b_{\mathrm{cum}}[i-1] + b[i] + 1$
5     end
6 end
```

Declaration of Authorship

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Ort, Datum Unterschrift

References

[1] Ivo Babuska and Werner C Rheinboldt. “Reliable error estimation and mesh adaptation for the finite element method”. In: *Computational methods in nonlinear mechanics* (1980), pp. 67–108.

[2] I Babuvška and Werner C Rheinboldt. “Error estimates for adaptive finite element computations”. In: *SIAM Journal on Numerical Analysis* 15.4 (1978), pp. 736–754.

[3] Michael Bader. *Space-Filling Curves*. Springer Berlin, Heidelberg, Oct. 2012. ISBN: 978-3-642-31046-1. DOI: `https://doi.org/10.1007/978-3-642-31046-1`.

[4] Santiago Badia et al. “A generic finite element framework on parallel tree-based adaptive meshes”. In: *CoRR* abs/1907.03709 (2019). arXiv: `1907.03709`. URL: `http://arxiv.org/abs/1907.03709`.

[5] Florian Becker. “Removing hanging faces from tree-based adaptive meshes for numerical simulations”. Erstgutachter: Prof. Dr. Gregor Gassner, Zweitgutachter: Dr. Johannes Holke. MA thesis. Universität zu Köln, Dec. 2021. URL: `https://elib.dlr.de/187499/`.

[6] Dietrich Braess and Rüdiger Verfürth. “A posteriori error estimators for the Raviart–Thomas element”. In: *SIAM Journal on Numerical Analysis* 33.6 (1996), pp. 2431–2444.

[7] Marek Bucki et al. “Jacobian-based repair method for finite element meshes after registration”. In: *Engineering with Computers* 27 (July 2011). DOI: `10.1007/s00366-010-0198-2`.

[8] Carsten Burstedde and Johannes Holke. “A Tetrahedral Space-Filling Curve for Nonconforming Adaptive Meshes”. In: *SIAM Journal of Scientific Computing* 38 (2016), pp. C471–C503. ISSN: 1064-8275. DOI: `10.1137/15m1040049`.

[9] Carsten Burstedde and Johannes Holke. “Coarse Mesh Partitioning for Tree-Based AMR”. In: *SIAM Journal on Scientific Computing* 39.5 (Jan. 2017), pp. C364–C392. DOI: `10.1137/16m1103518`.

[10] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. “`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees”. In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: `10.1137/100791634`.

[11] Henry Ker-Chang Chang and Jiang-Long Liu. “A linear quadtree compression scheme for image encryption”. In: *Signal Processing: Image Communication* 10.4 (1997), pp. 279–290. ISSN: 0923-5965. DOI: `https://doi.org/10.1016/S0923-5965(96)00025-2`. URL: `https://www.sciencedirect.com/science/article/pii/S0923596596000252`.

[12] A.O. Cifuentes and A. Kalbag. “A performance study of tetrahedral and hexahedral elements in 3-D finite element structural analysis”. In: *Finite Elements in Analysis and Design* 12.3 (1992), pp. 313–318. ISSN: 0168-874X. DOI: `https://doi.org/10.1016/0168-874X(92)90040-J`. URL: `https://www.sciencedirect.com/science/article/pii/0168874X9290040J`.

[13] L. Demkowicz et al. “Toward a universal h-p adaptive finite element strategy, part 1. Constrained approximation and data structure”. In: *Computer Methods in Applied Mechanics and Engineering* 77.1 (1989), pp. 79–112. ISSN: 0045-7825. DOI: https://doi.org/10.1016/0045-7825(89)90129-1. URL: https://www.sciencedirect.com/science/article/pii/0045782589901291.

[14] Willy Dörfler. “A convergent adaptive algorithm for Poisson’s equation”. In: *SIAM Journal on Numerical Analysis* 33.3 (1996), pp. 1106–1124.

[15] Luca Formaggia, Fausto Saleri, and Alessandro Veneziani. *Solving numerical PDEs: problems, applications, exercises*. Springer Science & Business Media, 2012.

[16] Thomas-Peter Fries et al. “Hanging nodes and XFEM”. In: *International Journal for Numerical Methods in Engineering* 86.4-5 (2011), pp. 404–430.

[17] Arthur Guittet, Maxime Theillard, and Frédéric Gibou. “A stable projection method for the incompressible Navier–Stokes equations on arbitrary geometries and adaptive Quad/Octrees”. In: *Journal of Computational Physics* 292 (2015), pp. 215–238. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2015.03.024. URL: https://www.sciencedirect.com/science/article/pii/S0021999115001710.

[18] Herman Haverkort. *Sixteen space-filling curves and traversals for d-dimensional cubes and simplices*. 2018. arXiv: 1711.04473 [cs.CG].

[19] Herman Haverkort and Freek van Walderveen. “Locality and bounding-box quality of two-dimensional space-filling curves”. In: *Computational Geometry* 43.2 (2010), pp. 131–147.

[20] Johannes Holke. “Scalable Algorithms for Parallel Tree-based Adaptive Mesh Refinement with General Element Types”. In: *CoRR* abs/1803.04970 (2018). arXiv: 1803.04970. URL: http://arxiv.org/abs/1803.04970.

[21] Johannes Holke and Carsten Burstedde. “Scalable algorithms for adaptive mesh refinement with arbitrary element types”. In: *Scalable algorithms for adaptive mesh refinement with arbitrary element types*. Nov. 2018. URL: https://elib.dlr.de/124383/.

[22] Johannes Holke, David Knapp, and Carsten Burstedde. “An Optimized, Parallel Computation of the Ghost Layer for Adaptive Hybrid Forest Meshes”. In: *SIAM Journal on Scientific Computing* 43.6 (Nov. 2021). Ed. by Jan Hesthaven, pp. 359–385. URL: https://elib.dlr.de/130166/.

[23] Jan Hungershöfer and Jens-Michael Wierum. “On the Quality of Partitions Based on Space-Filling Curves”. In: *Computational Science — ICCS 2002*. Ed. by Peter M. A. Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 36–45. ISBN: 978-3-540-47789-1.

[24] Fabrice Jaillet and Claudio Lobos. “Fast Quadtree/Octree adaptive meshing and re-meshing with linear mixed elements”. In: *Engineering with Computers* 38.4 (Mar. 2022), pp. 3399–3416. ISSN: 0177-0667. DOI: 10.1007/s00366-021-01330-w.

[25] Aly Khawaja and Yannis Kallinderis. “Hybrid grid generation for turbomachinery and aerospace applications”. In: *International Journal for Numerical Methods in Engineering* 49.1-2 (2000), pp. 145–166.

[26] Benjamin S Kirk et al. “libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations”. In: *Engineering with Computers* 22 (2006), pp. 237–254.

[27] David Knapp. “A space-filling curve for pyramidal adaptive mesh refinement”. Erstgutachter: Prof. Dr. Reinhard Klein, Zweitgutachter: Dr. Johannes Holke. MA thesis. Rheinische Friedrich-Wilhems-Universität Bonn, May 2022, p. 107205. DOI: `10.1016/j.tws.2020.107205`. URL: `https://elib.dlr.de/138982/`.

[28] Patrick M. Knupp. “Remarks on Mesh Quality.” In: 2007. URL: `https://api.semanticscholar.org/CorpusID:35978553`.

[29] Patrick M. Knupp et al. “The verdict geometric quality library.” In: 2006. URL: `https://api.semanticscholar.org/CorpusID:124174134`.

[30] Henry Lebesgue. “Leçons sur l’intégration et la recherche des fonctions primitives”. In: *Monatshefte für Mathematik und Physik* 15.1 (Dec. 1904), A46–A47. DOI: `10.1007/bf01692367`.

[31] Randall J LeVeque. *Finite volume methods for hyperbolic problems.* Vol. 31. Cambridge university press, 2002.

[32] Claudio Lobos. *A set of mixed-element transition patterns for adaptive 3d meshing.* Tech. rep. June 2015. DOI: `10.13140/RG.2.1.3367.4400`.

[33] Claudio Lobos et al. “Measuring geometrical quality of different 3D linear element types”. In: *Numerical Algorithms* 90 (2022). DOI: `10.1007/s11075-021-01193-8`.

[34] David Martineau et al. “Anisotropic hybrid mesh generation for industrial RANS applications”. In: *44th AIAA Aerospace Sciences Meeting and Exhibit.* 2006, p. 534.

[35] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing.* International Business Machines Company, 1966.

[36] W. Rachowicz and L. Demkowicz. “An hp-adaptive finite element method for electromagnetics: Part 1: Data structure and constrained approximation”. In: *Computer Methods in Applied Mechanics and Engineering* 187.1 (2000), pp. 307–335. ISSN: 0045-7825. DOI: `https://doi.org/10.1016/S0045-7825(99)00137-1`. URL: `https://www.sciencedirect.com/science/article/pii/S0045782599001371`.

[37] Hans Sagan. *Space-Filling Curve.* New York, NY: Springer New York, 1994. ISBN: 978-1-4612-0871-6. DOI: `10.1007/978-1-4612-0871-6_5`. URL: `https://doi.org/10.1007/978-1-4612-0871-6_5`.

[38] Sandro Salsa and Gianmaria Verzini. *Partial differential equations in action: from modelling to theory.* Vol. 147. Springer Nature, 2022.

[39] Teseo Schneider et al. “A Large-Scale Comparison of Tetrahedral and Hexahedral Elements for Solving Elliptic PDEs with the Finite Element Method”. In: *ACM Trans. Graph.* 41.3 (Mar. 2022). ISSN: 0730-0301. DOI: 10.1145/3508372. URL: https://doi.org/10.1145/3508372.

[40] Robert Schneiders. “Octree-based hexahedral mesh generation”. In: *International Journal of Computational Geometry & Applications* 10.04 (2000), pp. 383–398.

[41] Jason F. Shepherd and Chris R. Johnson. “Hexahedral mesh generation for biomedical models in SCIRun”. In: *Engineering with Computers* (2009). DOI: 10.1007/s00366-008-0108-z.

[42] Hang Si and A TetGen. “A quality tetrahedral mesh generator and three-dimensional delaunay triangulator”. In: *Weierstrass Institute for Applied Analysis and Stochastic, Berlin, Germany* 81 (2006), p. 12.

[43] Pavel Šolín, Jakub Červený, and Ivo Doležel. “Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM”. In: *Mathematics and Computers in Simulation* 77.1 (2008), pp. 117–132. ISSN: 0378-4754. DOI: https://doi.org/10.1016/j.matcom.2007.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0378475407001504.

[44] Srinivas C Tadepalli, Ahmet Erdemir, and Peter R Cavanagh. “Comparison of hexahedral and tetrahedral elements in finite element analysis of the foot and footwear”. In: *Journal of biomechanics* 44.12 (2011), pp. 2337–2343.

[45] Tiankai Tu, David R O’Hallaron, and Omar Ghattas. “Scalable parallel octree meshing for terascale applications”. In: *SC’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE. 2005, pp. 4–4.

[46] TIANKAI TU, DAVID R. O’HALLARON, and OMAR GHATTAS. “Scalable Parallel Octree Meshing for TeraScale Applications”. In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC ’05. USA: IEEE Computer Society, 2005, p. 4. ISBN: 1595930612. DOI: 10.1109/SC.2005.61. URL: https://doi.org/10.1109/SC.2005.61.

[47] David A Venditti and David L Darmofal. “Adjoint error estimation and grid adaptation for functional outputs: Application to quasi-one-dimensional flow”. In: *Journal of Computational Physics* 164.1 (2000), pp. 204–227.

[48] Samir Vinchurkar and P Worth Longest. “Evaluation of hexahedral, prismatic and hybrid mesh styles for simulating respiratory aerosol dynamics”. In: *Computers & Fluids* 37.3 (2008), pp. 317–331.

[49] Nils Zander et al. “Multi-level hp-adaptivity: high-order mesh adaptivity without the difficulties of constraining hanging nodes”. In: *Computational Mechanics* 55.3 (2015), pp. 499–517.

[50] Nils Zander et al. “The multi-level hp-method for three-dimensional problems: Dynamically changing high-order mesh refinement with arbitrary hanging nodes”. In: *Computer Methods in Applied Mechanics and Engineering* 310 (2016), pp. 252–277. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2016.07.007. URL: https://www.sciencedirect.com/science/article/pii/S0045782516307289.