Adaptive Verfeinerung von Prismen

David Knapp

Geboren am [redacted]

30. August 2017

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Carsten Burstedde

Zweitgutachter: Prof. Dr. Michael Griebel

Institut Für Numerische Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich direkt oder indirekt bei der Fertigstellung dieser Arbeit unterstützt haben.

Hierbei gilt mein besonderer Dank Herrn Prof. Dr. Burstedde und Johannes Holke. Zum einen für die Zuteilung dieses spannenden Themas und zum anderen für die freundliche Unterstützung und Hilfsbereitschaft bei Fragen jeglicher Art. Zudem bedanke ich mich ausdrücklich bei Herrn Prof. Dr. Griebel für die Begutachtung der Arbeit.

Weiterhin bedanke ich mich bei meinen Freunden und Kommilitonen. Ich habe die Stunden, die wir gemeinsam mit Lernen und kniffligen Übungsblättern verbracht haben, sehr genossen und freue mich darauf, noch mehr davon mit euch erleben zu dürfen.

Mein herzlichster Dank gilt natürlich meiner Familie für die außerordentliche Unterstützung, die ich über die Jahre hinweg erhalten habe. Vielen Dank für die lieben Carepakete, die in den Klausurphasen oft meine letzte Rettung waren und vor allem vielen Dank für die Hilfe beim Korrekturlesen dieser Arbeit.

Mein letzter, aber weitem nicht mindester, Dank gilt meiner ehemaligen Mathematiklehrerin Claudia R[redacted] die meine Begeisterung für die Mathematik in der Schulzeit entflammt hat. Ihretwegen habe ich mich für ein Studium der Mathematik entschieden und dafür bedanke ich mich von ganzem Herzen.

Inhaltsverzeichnis

1 Einleitung

Viele Probleme in der numerischen Mathematik benötigen Gitter, damit sie gelöst werden können. Bekannte Probleme dafür sind das Lösen von partiellen Differentialgleichungen (PDG) oder gewöhnlichen Differentialgleichungen (GDG), um zum Beispiel das Verhalten von Fluiden zu simulieren [15], aber auch in der Visualisierung von Grafiken am Computer (zum Beispiel für das Programm Paraview) werden Gitter benötigt [1,14]. Diese Probleme werden in der Praxis zumeist auf einem 1-, 2- oder 3-dimensionalen Gebiet gelöst. Dazu wird das Gebiet durch ein Gitter approximiert, welches aus Polygonen besteht. Hierfür werden meistens Dreiecke und Rechtecke in zwei Dimensionen, beziehungsweise Tetraeder und Hexaeder in drei Dimensionen verwendet [1–3]. Jedoch sind auch weitere Formen denkbar.
Ein Vorteil von Rechtecken und Hexaedern gegenüber Dreiecken und Tetraeder ist, dass sich durch sie eine einfache Struktur ergibt, anhand derer sich zum Beispiel Löser für PDG oder GDG einfach implementieren lassen [3]. Zudem benötigt es weniger Rechtecke und Hexaeder um ein Gebiet mit der gleichen Genauigkeit, die sich durch Dreiecke und Tetraeder ergibt, zu diskretisieren. Außerdem lassen sich durch Dreiecke und Tetraeder komplexere Strukturen besser approximieren. Um die Vorteile beider Polygone (Rechtecke und Dreiecke in 2D, Tetraeder und Hexaeder in 3D) zu nutzen bietet es sich an, diese zu kombinieren. Somit werden an weniger komplexen Stellen eines Gebietes Rechtecke, beziehungsweise Hexaeder verwendet, an komplexen Stellen jedoch Dreiecke, beziehungsweise Tetraeder. Dies ist in 2D direkt möglich, jedoch werden Prismen und Pyramiden benötigt um Tetraeder und Hexaeder zu kombinieren da ein Tetraeder nur Dreiecke und ein Hexaeder nur Vierecke als Oberflächen besitzt. Ein Dreiecksprisma kombiniert die Flächen von Tetraedern und Hexaedern, und lässt Verbindungen zwischen ihnen zu. Hierbei können jedoch Lücken entstehen, welche nicht durch Tetraeder oder Hexaeder, aber durch Pyramiden gefüllt werden können. Die Prinzipien zur Approximation durch Prismen werden in dieser Arbeit erarbeitet.
Eine gegebene Diskretisierung eines Gebietes ist meistens nicht fein genug, um eine zufriedenstellend genaue Lösung zu berechnen. Dies passiert zum Beispiel, wenn die Lösung einer PDG Singularitäten aufweist. Um trotzdem ausreichend exakte Ergebnisse zu erhalten, verfeinert man das Gitter, indem man die anfangs gegebenen Polygone verfeinert, also durch kleinere gleiche Polygone ersetzt. Um die Beziehung zwischen dem ursprünglichen Polygon und den von ihm aus verfeinerten Polygon zu verdeutlichen, werden sie Eltern und Kinder genannt. So entsehen Generationen von verfeinerten Polygonen [4]. Gebiete, welche in gleich große Polygone unterteilt sind, heißen uniform verfeinert.
Im Kontrast dazu ist ein möglicher Ansatz zur Optimierung des Rechenaufwandes, oder auch um das Gebiet nur lokal genauer aufzulösen, das Gebiet

nur an den Stellen zu verfeineren, an denen eine gegebene Fehlerschranke überschritten wird [14, 19]. Es wird also nur noch adaptiv verfeinert anstatt uniform. Umgekehrt kann es auch Stellen geben, an denen viel zu genau gerechnet wird und ein gröberes Gitter ausreichend ist um zufriedenstellende Ergebnisse zu bekommen. An solchen Umgebungen kann das diskretisierte Gebiet wieder vergröbert werden. Wie stark oder wie oft verfeinert worden ist, wird Level eines Polygons genannt. Eine der erfolgreichsten Methoden zum Diskretisieren eines Gebiets ist die konforme adaptive Gitterverfeinerung für Simplizes (Dreiecke und Tetraeder), da sie eine gute Möglichkeit bietet, komplexe Gebiete zu approximieren [3, 4, 14].
Zur weiteren Beschleunigung der Berechnungen lassen sich Programme parallelisieren. Das Problem soll in kleinere Probleme zerteilt werden und dann parallel auf mehreren Prozessen laufen. Dies bedeutet für das hier vorgestellte Problem, dass jeder Prozess eine Anzahl an Polygonen erhält um dort die entsprechenden auszuführenden Operationen abzuarbeiten. Dies nennt man eine Partitionierung des Gitters [3,14,16]. Dabei ist es wichtig, die Partionen sinnvoll zu wählen, sodass unter anderem die Partitionen zusammenhängend sind. Das Partitionieren von Gittern wird oftmals mittels graphenbasierten Algorithmen gelöst [7]. Jedoch handelt es sich dabei um ein NP-hartes Problem [18], weswegen eine Verbesserung der Skalierung, der Laufzeit und des Speicherverbrauchs eine Herausforderung bleibt.
Ein alternativer Ansatz sind hierbei raumfüllende Kurven (RFK), welche auch als space-filling curves bekannt sind. Die RFK soll hierbei jedes Poygon einmal durchlaufen und jedem Polygon wird anhand der Reihenfolge, in der die RFK sie durchläuft, eine Zahl zur eindeutigen Identifizierung (ID) zugewiesen. Danach wird die RFK in gleich lange Teile geteilt und die Arbeit, die für Objekte in einem Teil zu tun ist, dem gleichen Prozess zugewiesen. Gleich lang bedeutet in diesem Fall, dass der Arbeitsaufwand, der in jedem Abschnitt zu tun ist, gleich groß ist. Im Gegensatz zum NP-harten Problem kann man mittles RFK das Partitionierungsproblem in linearer Zeit approximierend lösen [1, 3, 14, 16].
Jedes Polygon wird beim Verfeinern zerteilt, die dabei entstehende Eltern-Kind Beziehungen entsprechen einem Baum, wobei jedes initial gegebene Element ein Baum ist. Wird ein Kind noch einmal verfeinert, so entsteht auch im Baum für jedes neue Kind ein neuer Knoten, welcher vom entsprechenden Elternknoten abgeht. Für jedes weitere Level entsteht somit eine Generation in dem Baum. Soll dann der Index eines Elementes berechnet werden, so wird nur noch eine Schleife über das Verfeinerungslevel benötigt. Es ist also äquivalent dazu, einem Weg im Baum zu folgen, um den Index eines Objektes zu finden [2, 14, 17].
Diese Arbeit beschäftigt sich mit der Ausarbeitung von Prismen, welche durch eine Übertragung des tetraedischen Morton Indexes [1] auf Prismen indiziert und verfeinert werden sollen. Darüber hinaus erarbeiten wir das adaptive Verfeinern von Linien, welches für die Übertragung des tetraedi-

schen Morton Indexes auf Prismen nötig ist. Zudem beweist die Arbeit, dass sich wichtige Eigenschaften des Index für Tetraeder auch auf den Index für Prismen übertragen. Abschließend zeigen Beispiele, dass die erarbeiteten Prinzipien effizient sind und ideal skalieren.

Umsetzung

Zunächst wird in dieser Arbeit das Prinzip einer RFK für Rechtecke und Dreiecke vorgestellt. In dieser Arbeit werden Prismen als ein Kreuzprodukt zwischen einem Dreieck und einer Linie betrachtet, sodass der schon bestehenden Index und die eins zu vier Verfeinerung von Dreiecken verwendet werden können. Eine Linie soll durch eine Halbierung verfeinert werden, die Indizierung der Linien soll linear sein. Somit soll ein Prisma in acht weitere Prismen unterteilt werden und durch eine Kombination der Linien- und Dreiecksindizierung indiziert werden. Zudem wird am Ende eine Implementierung der Prismen in der AMR Bibliothek `t8code` [8] vorgestellt, welche das Problem unter anderem schon für Dreiecke löst. Die Bibliothek selber wird ebenfalls vorgestellt. Hier soll das Ziel sein, möglichst auf schon bestehende Algorithmen zurückzugreifen, um die Kreuzproduktstruktur auch im Code zu verdeutlichen. Darüber hinaus wird die Implementierung auf ihre Skalierbarkeit getestet. Sämtliche Beispiele und Tests wurden mit Hilfe der `t8code`-Bibliothek implementiert.

Konventionen

In dieser Arbeit werden, sofern nicht explizit anders beschrieben, folgende Konventionen genutzt:
Ein beliebiges Dreiecksprisma wird P genannt. Zudem wird jeder Form ein maximales Verfeinerungslevel zugeteilt. Zum Beispiel kann ein Rechteck ein anderes maximales Verfeinerungslevel haben als ein Dreieck.
Anhand der RFK wird jedem Element eine ID zugeteilt. Da das Maximallevel und die ID eines Objektes zueinander im Bezug stehen (siehe Kapitel 4 und 5) und wir sicherstellen wollen, dass wir die ID immer speichern können, muss es dementsprechend gewählt werden. Darüber hinaus hängt das Maximallevel davon ab, in wie viele Kinder sich ein Objekt teilt. Für die ID wurde ein 64 bit Integer verwendet und die Maximallevel wie folgt festgelegt:

Elementtyp	Bezeichnung des Maximallevel	Kinder	gesetztes Maxlevel
LINE	$\mathcal{L}_1$	2	30
TRI	$\mathcal{L}_2$	4	30
QUAD	$\mathcal{L}_3$	4	30
PRISM	$\mathcal{L}_4$	8	21

Tabelle 1: Die Elemente und ihre im `t8code` gesetzten Maximallevel

Wird das Maximallevel nicht weiter spezifiziert angegeben, so wird es $\mathcal{L}$

genannt. l steht für das aktuelle Verfeinerungslevel, $l(P)$ gibt das Verfeinerungslevel bezüglich eines Objektes an. Außerdem wird von jedem Objekt ein ausgezeichneter Knoten betrachtet, der Ankerknoten. Dieser soll die Eigenschaft haben, möglichst kleine Koordinaten zu haben. Im weiteren Verlauf der Arbeit wird klar, dass durch das Level und den Ankerknoten sich alle weiteren Eckkoordinaten eines Objektes berechnen lassen. Die Menge aller möglichen Koordinaten für Ankerknoten wird definiert als

$$\mathbb{L} = \{[0, 2^{\mathcal{L}}) \cap \mathbb{Z} | 0 \leq x\} \tag{1}$$

$$\mathbb{L}^2 = \{[0, 2^{\mathcal{L}})^2 \cap \mathbb{Z}^2 | 0 \leq y \leq x\} \tag{2}$$

$$\mathbb{L}^3 = \{[0, 2^{\mathcal{L}})^3 \cap \mathbb{Z}^3 | 0 \leq y \leq z \leq x\} \tag{3}$$

[1]

Darüber hinaus werden die Objekte in unterschiedlich viele Kinder zerteilt. In dieser Arbeit wird eine Möglichkeit für die Teilung vorgestellt, jedoch sind auch andere denkbar. Um die Allgemeinheit beizubehalten, werden in den Algorithmen die Anzahl der Kinder für Prismen mit $\mathcal{PC}$, für Dreiecke mit $\mathcal{TC}$ und für Linien mit $\mathcal{LC}$ bezeichnet. In den Algorithmen werden *and* und *or* als logisches *und* bzw. *oder* verwendet. Das bitweise *und* wird mit & und das *oder* mit | beschrieben. Zudem wird in den Algorithmen der Ausdruck A ? EXPR0 : EXPR1 benutzt. Dieser steht für den Wert in EXPR0, falls A wahr ist, und für EXPR1, falls A falsch ist. Darüber hinaus wird mit „%“ die Modulo Rechnung beschrieben.

2 Theoretische Grundlagen

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen der adaptiven Gitterverfeinerung. Die wichtigsten Resultate zu diesem Thema sollen dem Leser ins Gedächtnis gerufen werden, es bietet jedoch keine vollständige Einführung. Die vorgestellten Methoden sind hauptsächlich aus [1, 10, 14] entnommen.

2.1 Gitter

Gewöhnliche Differentialgleichungen (GDG) und partielle Differentialgleichungen (PDG) werden auf einem Gebiet $\Omega \subset \mathbb{R}^d$ betrachtet. Da dies jedoch nur in seltenen Fällen eine einfach zu diskretisierende geometrische Form ist, ist es notwendig Ω zu approximieren und in ein Gitter zu zerlegen um die diskretisierte Gleichung zu lösen. Dazu wird Ω trianguliert, also in kleinere, einfachere Teilgebiete zerlegt. Hierbei ist darauf zu achten, dass sich die Teilgebiete nicht überschneiden. Dazu werden Polygone gewählt.

Definition 1. *Sei $\Omega \subset \mathbb{R}^d$ beschränkt und polygonal berandet.*
$T = \{T_i | i \in I,\ T_i \text{ Polygon}\}$ *heißt zulässige Triangulierung von Ω, falls:*

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

(ii) $T_i \cap T_j \neq \emptyset \Rightarrow T_i \cap T_j$ *ist $(n-k)$ - dimensionales Unterpolygon von T_i und T_j $(0 \leq k \leq d)$*

[4, 19]

Wir werden Tetraeder, Hexaeder und Prismen als Polygone in drei Dimensionen betrachten. Wie in der Einleitung schon beschrieben, ist es oftmals sinnvoll, eine gegebene Triangulierung weiter zu verfeinern. Dazu wird ein einzelnes Element in kleinere Elemente zerlegt. Um den globalen Fehler der diskreten Lösung an die exakte Lösung zu verringern, genügt es, den lokalen Fehler zu verringern [4]. Dazu werden die Elemente, an denen der lokale Fehler noch zu groß ist, in ihre Kinder-Elemente zerlegt. Die Basis eines Prismas, also ein Dreieck, soll durch eine Red-Verfeinerung zerteilt und nach Beys Verfeinerungsregel nummeriert werden (siehe Def. 3 und Abb. 1). Dieses Idee wird auf Prismen übertragen.

Definition 2. *Ein 2-Simplex (Dreieck) T oder ein Prisma P wird durch seine drei bzw. sechs Ecken definiert. Diese werden durch Vektoren angegeben.*

$$T := [\vec{x}_0, \vec{x}_1, \vec{x}_2] \tag{4}$$
$$P := [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \vec{x}_5] \tag{5}$$

Definition 3. *Sei T ein 2-Simplex gegeben durch $T = [\vec{x}_0, \vec{x}_1, \vec{x}_2]$, so wird es nach Beys Verfeinerungsregel durch Abschneiden von Untersimplizes in den Ecken verfeinert. Die vier neuen Dreiecke sind:*

$$T_0 \coloneqq [\vec{x}_0, \vec{x}_{01}, \vec{x}_2], \qquad T_1 \coloneqq [\vec{x}_{01}, \vec{x}_1, \vec{x}_{12}] \tag{6}$$

$$T_2 \coloneqq [\vec{x}_{02}, \vec{x}_{12}, \vec{x}_2], \qquad T_3 \coloneqq [\vec{x}_{01}, \vec{x}_{02}, \vec{x}_{12}] \tag{7}$$

Wobei $\vec{x}_{ij} \coloneqq \frac{1}{2}(\vec{x}_i + \vec{x}_j)$. [1]

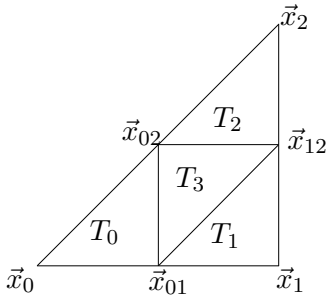

Abbildung 1: Die Unterteilung eines Dreiecks $T = [\vec{x}_0, \vec{x}_1, \vec{x}_2]$ in seine vier Kinderdreiecke $T_0, \ldots, T_3$

Definition 4. *Sei P ein Prisma gegeben durch $P = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \vec{x}_5]$. Wird es verfeinert so soll es in acht Prismen geteilt werden. In der Höhe wird es halbiert, die Grundfläche des Prismas soll wie in Definition 3 geteilt werden. Die acht neuen Prismen sind:*

$$P_0 \coloneqq [\vec{x}_0, \vec{x}_{01}, \vec{x}_{02}, \vec{x}_{03}, \vec{x}_{04}, \vec{x}_{05}], \quad P_1 \coloneqq [\vec{x}_{01}, \vec{x}_1, \vec{x}_{12}, \vec{x}_{04}, \vec{x}_{14}, \vec{x}_{15}] \tag{8}$$

$$P_2 \coloneqq [\vec{x}_{02}, \vec{x}_{12}, \vec{x}_2, \vec{x}_{05}, \vec{x}_{15}, \vec{x}_{25}], \quad P_3 \coloneqq [\vec{x}_{01}, \vec{x}_{02}, \vec{x}_{12}, \vec{x}_{04}, \vec{x}_{05}, \vec{x}_{15}] \tag{9}$$

$$P_4 \coloneqq [\vec{x}_{03}, \vec{x}_{04}, \vec{x}_{05}, \vec{x}_3, \vec{x}_{34}, \vec{x}_{35}], \quad P_5 \coloneqq [\vec{x}_{04}, \vec{x}_{14}, \vec{x}_{15}, \vec{x}_{34}, \vec{x}_4, \vec{x}_{45}] \tag{10}$$

$$P_6 \coloneqq [\vec{x}_{05}, \vec{x}_{15}, \vec{x}_{25}, \vec{x}_{35}, \vec{x}_{45}, \vec{x}_5], \quad P_7 \coloneqq [\vec{x}_{04}, \vec{x}_{05}, \vec{x}_{15}, \vec{x}_{34}, \vec{x}_{35}, \vec{x}_{45}] \tag{11}$$

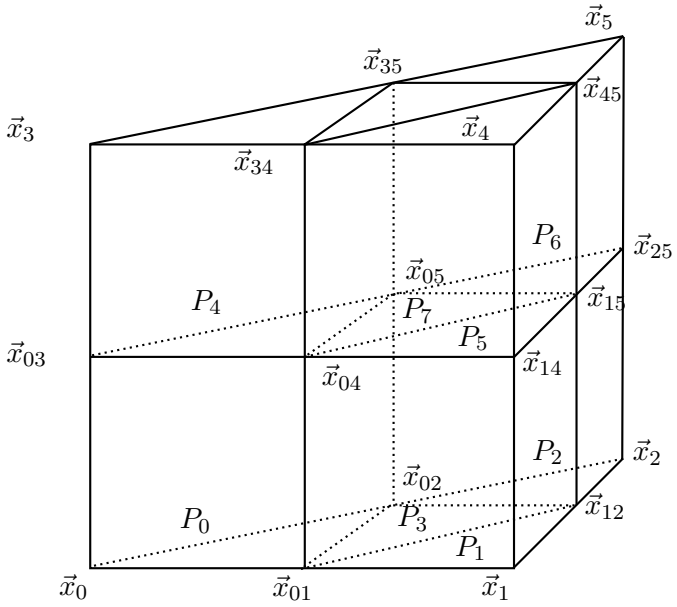

Abbildung 2: Die Unterteilung eines Prismas $P = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \vec{x}_5]$ in seine acht Kinderprismen $P_0, \ldots, P_7$.

Es wird auch ein Index für Quadrate kurz vorgestellt, weswegen hier eine Verfeinerungsregel für Quadrate präsentiert wird.

Definition 5. *Ein Quadrat* $Q = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3]$ *teilt sich in seine vier Kinder* Q_0, Q_1, Q_2, Q_3 *wie folgt auf:*

$$Q_0 \coloneqq [\vec{x}_0, \vec{x}_{01}, \vec{x}_{02}, \vec{x}_{03}] \qquad Q_1 \coloneqq [\vec{x}_{01}, \vec{x}_1, \vec{x}_{03}, \vec{x}_{13}] \tag{12}$$

$$Q_2 \coloneqq [\vec{x}_{02}, \vec{x}_{13}, \vec{x}_2, \vec{x}_{23}] \qquad Q_3 \coloneqq [\vec{x}_{03}, \vec{x}_{13}, \vec{x}_{23}, \vec{x}_3] \tag{13}$$

2.2 Raumfüllende Kurven

Im Jahr 1878 beobachtete Georg Cantor, dass zwei beliebige endlichdimensionale Mannigfaltigkeiten die gleiche Kardinalität haben, unabhängig von ihrer Dimension [10]. Dies impliziert, dass es eine bijektive Abbildung zwischen dem Intervall $[0, 1]$ und $[0, 1]^2$ gibt, beziehungsweise im dreidimensionalen eine bijektive Abbildung zwischen $[0, 1]$ und dem Würfel $[0, 1]^3$. Dies wirft allerdings die Frage auf, ob so eine Abbildung stetig sein kann. Nachdem dies von E. Netto wiederlegt wurde, entdeckte G. Peano jedoch eine Abbildung, welche raumfüllend ist, also jeden Punkt im Quadrat $[0, 1]^2$ berührt und einen positiven Jordaninhalt hat. Weitere Beispiele folgten durch D. Hilbert [5], E.H. Moore [6], H. Lebesque [12], W. Sierpinski [20] und viele weitere [10].

Definition 6. *Sei* $\mathcal{I} = [0, 1]$ *und* $\mathbb{E}^n$ *der n-dimensionale euklidische Raum (*$\mathbb{R}^n$ *mit der durch die euklidischen Metrik definierten Norm). Falls*

$f : \mathcal{I} \to \mathbb{E}^n$ stetig ist, so ist im(f) eine Kurve. $f(0)$ ist der Startpunkt und $f(1)$ der Endpunkt der Kurve [10].

Definition 7. *Sei $f : \mathcal{I} \to \mathbb{E}^n$, $n \geq 2$ eine Kurve, stetig und $\mathcal{J}_n(im(f)) > 0$, dann ist im(f) eine raumfüllende Kurve. Hierbei bezeichnet $\mathcal{J}_n$ den n-dimensionalen Jordaninhalt [10].*

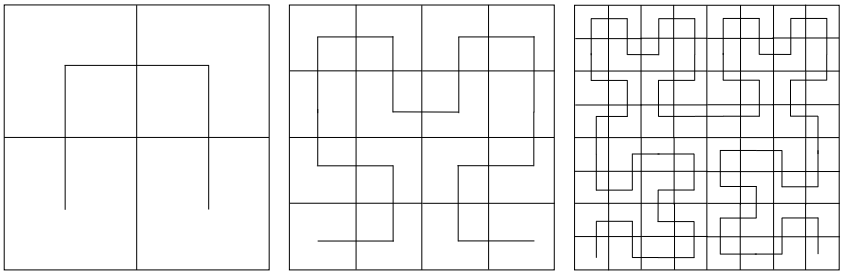

Abbildung 3: Ein Beispiel für eine raumfüllende Kurve, die Hilbertkurve. Zu sehen sind die ersten drei Iterationen. Bildet man den Limes nach dem dargestellen Prinzip, füllt diese Kurve den gesamten Raum des äußeren Quadrates.

2.2.1 Raumfüllende Kurven in der numerischen Mathematik

Hat man nun die Diskretisierung eines Gebiets $\Omega \subset \mathbb{R}^d$ gegeben und verfeinert es, so stellt sich die Frage, wie die neu entstehenden Elemente gespeichert werden sollen. Dabei sollen Elemente, die in Ω nah beieinander liegen auch im Speicher nah beieinander liegen. Optimal wäre es, wenn die Nachbarschaftsbeziehungen auf dem Gebiet auch im Speicher erhalten blieben. Wenn man sich nun den Speicher als 1-dimensionales diskretisiertes Intervall vorstellt, so ist es also wieder die Aufgabe, eine Beziehung zwischen dem Intervall und dem d-dimensionalen Gebiet Ω herzustellen. Somit ist schon klar, dass die Nachbarschaftsbeziehungen nicht direkt übertragen werden können, da ein Element im Speicher nur zwei Nachbarn besitzt, in höheren Dimensionen Elemente aber mehr als nur zwei Nachbarn haben. Zudem ist zu bemerken, dass es nicht nötig ist, die RFK unendlich oft zu iterieren, da schon nach endlich vielen Iterationen sämtliche Elemente durchlaufen werden, und es nicht mehr Ziel ist, den kompletten Raum mit einer Kurve zu füllen. Darüber hinaus ist es sinnvoll, nicht gleichmäßig zu iterieren, also in jedem Schritt an jeder Stelle, sondern nur an den Stellen, an denen es neue, noch nicht durchlaufene Objekte gibt. Somit soll analog zum adaptiven Verfeinern des Gebietes auch die Kurve nur adaptiv verfeinert werden können. Anhand der Reihenfolge, in der die RFK die Objekte durchläuft, sollen die Objekte indiziert werden. Sie sollen eine ID erhalten. Das Indizieren soll möglichst schnell und effizient geschehen, somit ist ein Anspruch an die RFK, dass sich der Index schnell ermitteln lässt [14].

2.2.2 Eine raumfüllende Kurve für Linien

Eine raumfüllende Kurve für eine Linie, Beziehungsweise das Einheitsintervall zu finden, soll hier als ein einsteigendes Beispiel besprochen werden, findet aber auch im weiteren Verlauf der Arbeit noch Anwendung, da die Höhe eines Prismas wie eine Linie verfeinert werden soll. Der Algorithmus, der im weiteren Verlauf noch vorgestellt wird, unterstützte zum Beginn der Arbeit noch keine Linien, weswegen das Konzept ebenfalls erarbeitet wurde. Zudem ist das Konzept der RFK für Prismen eine Kombination aus dem Prinzip der RFK für Linien und für Dreiecke. Die Konstruktion der RFK für eine Linie erfolgt rekursiv und ähnelt einer Generation in einem Binärbaum:

(i) Das zu verfeinernde Intervall, am Anfang das Einheitsintervall, wird in zwei gleich lange Intervalle geteilt.

(ii) Finde eine RFK für jedes Unterintervall.

Iteriert man diese Vorschrift nun unendlich oft, so wird einer Zahl $x \in [0, 1]$ auch x im Bildraum zugeordnet. Man erhält die Identität, welche eine RFK ist.

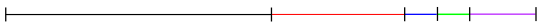

Abbildung 4: Eine adaptiv verfeinerte Linie. Die Färbung dient zur Darstellung der Intervalle.

2.2.3 Der Morton Index

Als einführendes Beispiel, und auch um einige Konzepte vorzustellen, die in späteren Kapiteln wichtig sind, soll hier der Morton Index für Quadrate vorgestellt werden. Ein zwei dimensionales Gebiet Ω soll durch Vierecke approximiert werden. Um das Problem zu vereinfachen, werden die Elemente, durch die Ω approximiert wird, auf einem Referenzquadrat betrachtet. Anstatt klassischerweise das Einheitsquadrat zu benutzen, wird es hier skaliert und das Quadrat $[0, 2^{\mathcal{L}}]^2$ als Referenzquadrat benutzt. Wird ein Quadrat verfeinert, so soll es in vier gleich große Quadrate geteilt werden, wie in Definition 5 beschrieben. Damit wird erreicht, dass bis zum maximalen Verfeinerungslevel alle Nachfahren nur ganzzahlige Koordinaten an ihren Eckpunkten besitzen.
Zur Berechnung des Morton Index wird ein ausgezeichneter Knoten des Quadrates benötigt: der Ankerknoten (siehe Abb. 5). Dies ist der linke untere Eckpunkt, also der PUnkt, mit den kleinsten x- und y-Koordinaten. Da sich die Kantenlängen im Referenzwürfel mithilfe des Levels berechnen lassen, genügt es, den Ankerknoten zu kennen. Wird ein Element nur durch Halbieren seiner Kanten verfeinert, so ist die Länge einer Kante eines Elementes

E von Level ℓ gegeben durch

$$\text{length}(E) \coloneqq 2^{\mathcal{L}-\ell} \tag{14}$$

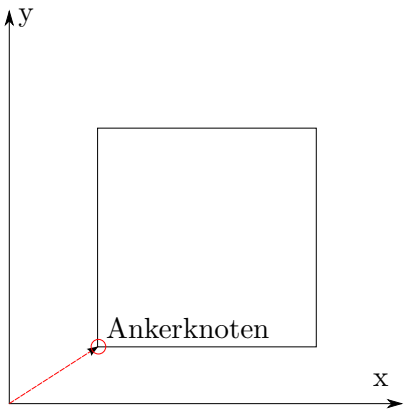

Abbildung 5: Ein Würfel und sein Ankerknoten. Der rot gezeichnete Pfeil ist der zum Ankerknoten gehörige Vektor.

Der Kern des Morton Index liegt im Überlagern der Ankerkoordinaten. Diese Operation wird benötigt, um den Index eines Elementes zu berechnen.

Definition 8. *Wir definieren das Überlagern $a \dot{\perp} b$ zweier n-Tupel $a = (a_{n-1}, \ldots, a_0)$ und $b = (b_{n-1}, \ldots, b_0)$ als das 2n-Tupel, welches durch Alternieren der Einträge von a und b erhalten wird:*

$$a \dot{\perp} b \coloneqq (a_{n-1}, b_{n-1}, \ldots, a_0, b_0) \tag{15}$$

Das Überlagern von mehr als zwei n-Tupeln ist analog als mn-Tupel definiert:

$$a^1 \dot{\perp} a^2 \dot{\perp} \ldots a^m \coloneqq (a^1_{n-1}, a^2_{n-1}, \ldots, a^m_{n-1}, a^1_{n-2}, \ldots, a^{m-1}_0, a^m_0) \tag{16}$$

[1]

Essentiel zur Berechnung des Mortonindex ist, dass die Koordinaten des Ankerknoten in Binärdarstellung gegeben sind. Vorbereitend für spätere Kapitel wird hier auch schon ein Tupel Z definiert, welches für Ankerknoten in 3 Dimensionen relevant ist.

Definition 9. *Sei Q ein Quadrat (ein Prisma) von Verfeinerungslevel $\ell \leq \mathcal{L}$ mit Ankerknoten $\vec{x}_0 = (x, y)^T \in \mathbb{L}^2$ ($\vec{x}_0 = (x, y, z)^T \in \mathbb{L}^3$). Da $x, y, (z) \in \mathbb{N}_0$ mit $0 \leq x, y, (z) \leq 2^{\mathcal{L}}$, können sie auch als Binärzahl mit $\mathcal{L}$ Stellen geschrieben werden:*

$$x = \sum_{j=0}^{\mathcal{L}-1} x_j 2^j, \quad y = \sum_{j=0}^{\mathcal{L}-1} y_j 2^j, \quad \left(z = \sum_{j=0}^{\mathcal{L}-1} z_j 2^j \right) \tag{17}$$

Somit lassen sich die $\mathcal{L}-Tupel$ X und Y, bestehend aus den binären Stellen von x und y definieren:

$$X = X(T) \coloneqq (x_{\mathcal{L}-1}, \ldots, x_0), \tag{18}$$
$$Y = Y(T) \coloneqq (y_{\mathcal{L}-1}, \ldots, y_0), \tag{19}$$
$$(Z = Z(T) \coloneqq (z_{\mathcal{L}-1}, \ldots, z_0)) \tag{20}$$

[1]

Mit Hilfe dieser Definitionen lässt sich der Morton Index für Rechtecke definieren. Er lässt sich jedoch auch noch weiter verallgemeinern und auf d-dimensionale Würfel übertragen.

Definiton 10. *Der Morton Index eines Rechtecks Q ist definiert als Überlagerung von den $\mathcal{L}$-Tupeln Y und X:*

$$\tilde{m} \coloneqq Y \dot{\perp} X \tag{21}$$

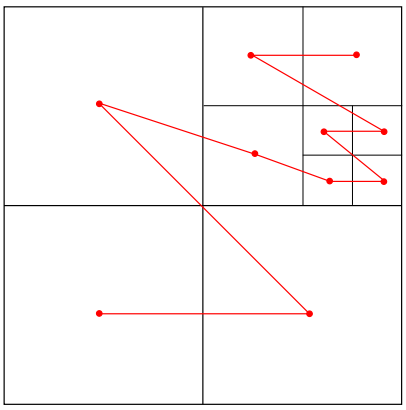

Abbildung 6: Ein Quadrat und die raumfüllende Kurve nach dem Morton Index.

2.2.4 Der tetraedische Morton Index

Die Algorithmen zur Berechnung des Index für Prismen basieren auf dem tetraedischen Morton Index (TM-Index). Darum soll zunächst dieser betrachtet werden. Für diese Arbeit ist vor allem der 2-dimensionale Fall des Index interessant, da wir hier Dreiecke betrachten. Im 3-dimensionalem indiziert der TM-Index Tetraeder. Zur genaueren Erarbeitung des TM-Index empfiehlt sich die Arbeit „A Tetrahedral Space-Filling Curve for nonconforming adaptive meshes“ [1], und dementsprechend sind die hier vorgestellten Methoden hauptsächlich aus [1] entnommen. Der erste Schritt zur Erarbeitung des TM-Index ist, dass wir Dreiecke auf einem Referenzdreieck betrachten,

welches wiederum in einem Referenzquadrat liegt (siehe Abb. 7). Anstatt den Einheitswürfel $[0,1]^2$ zu betrachten, wird ein skaliertes Quadrat $[0,2^{\mathcal{L}}]^2$ betrachtet. Dies hat den Vorteil, dass die Koordinaten aller Kinder ganzzahlige Binärdarstellungen besitzen, wenn das Verfeinern nur durch Halbieren der Kanten und nicht über das Maximallevel hinaus durchgeführt wird. Dieses Referenz-d-simplex wird T_d^0 genannt. Für Dreiecke nennen wir es S_2^0 und für Prismen P_3^0. Somit reicht es also aus, RFK auf T_d^0 zu betrachten. Sei nun T_d die Menge aller möglichen Nachfahren des d-Simplex T_d^0, also

$$T_d \coloneqq \left\{ T \mid T \text{ ist ein Nachfahre von } T_d^0, \text{ mit } 0 \leq l(T) \leq \mathcal{L} \right\} \tag{22}$$

Der TM-Index basiert auf dem Morton Index für Vierecke und Hexaeder, weswegen einige Methoden auf Dreiecke übertragen werden sollen.
Zur Berechnung des TM-Index wird zunächst jedem Dreieck eine eindeutige ID zugeordnet, welche auf dem Typ und dem Ankerknoten des Dreiecks basiert. Da zwei aneinander liegende Dreiecke ein Rechteck bilden, gibt es ein zugrunde liegendes kubisches Gitter. Wie im Abschnitt zum Morton-Code wird dann als Ankerknoten der Dreiecke die Ecke mit einer möglichst kleinen x- und y-Koordinate gewählt. Somit entspricht der Ankerknoten des Würfels dem Ankerknoten des Dreiecks.

Definition 11. *Jedes 2-Simplex $T \in T_2$ von Level ℓ liegt in einem Quadrat des kubischen Gitters, welches Teil der gleichmäßigen Verfeinerung von Level ℓ von $[0,2^{\mathcal{L}}]^2$ ist. Dieses Quadrat wird zu T assoziiert und Q_T genannt. Jedes 2-Simplex T ist eine skalierte und verschobene Version von genau einem der zwei Dreiecke S_i und wir definieren den Typ von T als diese Zahl,*

$$\text{type}(T) \coloneqq i. \tag{23}$$

[1]

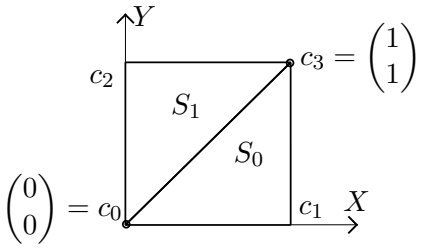

Abbildung 7: Die zwei Typen von Dreiecken in einem Referenzquadrat. Zudem erkennt man, dass der Knoten c_0 der Ankerknoten für beide Dreiecke ist. Durch den Typ unterscheiden sie sich. Die Zeichung wurde entnommen aus [1] mit Erlaubnis der Autoren.

Die Länge einer Kante ist wieder durch das Level eines Dreiecks bestimmt, die Richtung der Kanten wird durch den Typ des Dreiecks fest-

gelegt. Somit lässt sich jedes Dreieck durch die Wahl eines Levels ℓ, eines Würfels von Level ℓ und eines Typs bestimmen.

Definition 12. *Für $T = [\vec{x_0}, \ldots, \vec{x_d}] \in T_d$ definieren wir die Tet-ID von T als das Tupel aus Ankerknoten und Typ von T:*

$$\text{Tet-ID}(T) \coloneqq (\vec{x}_0, \text{type}(T)) \tag{24}$$

[1]

Korollar 13. *Seien $T, T' \in T_d$. Dann ist $T = T'$ genau dann, wenn ihre Tet-ID's und Level gleich sind. [1]*

Ein weiter Schritt zur Erarbeitung des TM-Index ist, dass beim Berechnen des Index nicht nur der aktuelle Typ eines Dreiecks betrachtet wird, sondern auch die Typen aller Vorfahren eines Dreieckes. Diese sollen dann mit den x- und y- Koordinaten des Ankerknotens überlagert werden.

Definition 14. *Für ein $T \in T_d$ von Level ℓ und jedem $0 \leq j \leq \ell$ sei T^j der (eindeutige) Vorgänger von T von Level j. Wir definieren $B(T)$ als das $\mathcal{L}-$Tupel mit den Typen von T's Vorgängern in den ersten ℓ Einträgen, beginnend mit T^1. Die letzen $\mathcal{L} - \ell$ Einträge von $B(T)$ sind Nullen:*

$$B = B(T) \coloneqq (\text{type}(T^1), \text{type}(T^2), \ldots, \text{type}(T^\ell), 0, \ldots, 0). \tag{25}$$

Da wir für die Stellen in B nur 0 und 1 zulassen, besitzt B direkt eine binäre Darstellung. Somit ergibt sich beim Überlagern der Koordinaten und B wieder eine binäre Zahl, der TM-Index.

Definition 15. *Der Tetraedische Morton Index (TM-Index) eines d-Simplex $T \in T_d$ ist definiert als Überlagerung von den $\mathcal{L}$-Tupeln Y, X und B:*

$$m(T) \coloneqq Y \dot{\perp} X \dot{\perp} B \tag{26}$$

Der Morton Index erfüllt drei wichtige Eigenschaften, die sich auf den TM-Index übertragen lassen. Sie sichern zu, dass beim Verfeinern eines Würfels nur lokale Änderungen der Indizierung, gegeben durch den Morton Index passieren. Für beliebige d-Simplizes $T \neq S \in T_d$ erfüllt der TM-Index m folgende Eigenschaften:

(i) Wenn S ein Nachfahre von T ist, so ist auch sein TM-Index größer oder gleich dem von S ($m(T) \leq m(S)$).

(ii) Wenn das Level von T kleiner als das Level von S ist, so ist $m(T)$ ein Präfix von $m(S)$ genau dann, wenn S ein Nachfahre von T ist.

(iii) Verfeinern ändert die RFK nur lokal. Wenn $m(T) < m(S)$ und S kein Nachfahre von T ist, dann gilt für jeden Nachfahre T' von T, dass $m(T) \leq m(T') < m(S)$ ist.

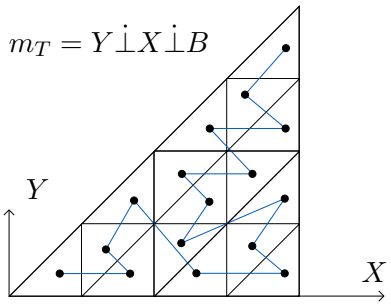

Abbildung 8: Die raumfüllende Kurve nach dem TM-Index. Es wird deutlich, dass zuerst alle Kinder durchlaufen werden, bevor die Kurve ein Dreieck des selben Levels durchläuft. Das Bild ist entnommen aus [1] mit Erlaubnis der Autoren.

3 Der t8code

Die hier vorgestellten Algorithmen sind Teil der t8code-Bibliothek [8]. Dies ist eine Bibliothek, die Algorithmen zur Manipulation von Gittern zur Verfügung stellt. Die Bibliothek kann Gitter mit verschiedenen Elementen approximieren. Die Struktur der globalen Algorithmen (High-Level Algorithmen) hängt nicht von der konkreten Implementierung der lokalen Elementalgorithmen (Low-Level Algorithmen) ab. In den Elementalgorithmen werden unter anderem die Elternelemente, Kinderelemente, Nachbarn, oder die ID berechnet. Im Gegensatz zu den High-Level Algorithmen sind die Low-Level Algorithmen also abhängig vom Element. Der High-Level Algorithmus gibt unabhängig vom dem zu verfeinernden Element vor, wann welcher Algorithmus des jeweiligen Elementes gebraucht wird. Zudem ist zu beachten, dass das „Level“ in High- und Low-Level nichts mit dem Level eines Objektes zu tun hat. Die Implementierung der Prismen und Linien ist Gegenstand dieser Arbeit. Es existieren bereits die Low-Level Algorithmen für Dreiecke, Rechtecke, Tetraeder und Hexaeder, somit lassen sich auch hybride Gitter erzeugen. Dieser Abschnitt beruht im Wesentlichen auf der Arbeit „A Tetrahedral Space-Filling Curve for nonconforming adaptive meshes“ [1]. Wir stellen nun die High-Level Algorithmen `New`,`Adapt`,`Partition` und `Ghost` vor.

3.1 New

Der `New`-Algorithmus bekommt ein konformes Gitter von Elementen übergeben, welche jeweils als Wurzel eines Verfeinerungsbaumes sind. Zudem werden sie initial bis zu einem Level ℓ verfeinert und partitioniert. Als erstes wird für jeden Prozess der erste und der letzte Element berechnet, somit lässt sich ermitteln, welche Bäume zu welchem Prozess gehören. Für die Bäume wird der Index des ersten und des letzten Element berechnet und das erste Element wird erstellt. Da das wiederholte Erstellen von Elementen anhand ihrer Indizes sehr viel Rechenzeit kostet, werden alle weiteren Elemente durch eine Berchnung des Nachfolgers erstellt. Dies wird dann solange iteriert, bis das letzte Element eines Baums erreicht wird, bzw. bis alle Bäume erstellt worden sind.

3.2 Adapt

Wie in der Einleitung schon beschrieben, kann eine bessere Approximation an die exakte Lösung einer PDG erreicht werden, wenn das initiale Gitter adaptiv verfeinert wird. Dies unterstützt auch der `t8code`. Ein gegebenes Gitter kann bezüglich eines vorgegebenen Kriteriums verfeinert oder vergröbert werden. Hierzu werden die Elemente eines Gitters entlang der RFK, anhand derer sie indiziert sind, durchlaufen. Für jeden d-Simplex wird eine

Funktion aufgerufen, die bestimmt, ob es verfeinert werden soll. Haben der aktuelle d-Simplex und seine $2^d - 1$ Nachfolger den gleiche Elternsimplex, so bilden sie eine Familie und werden komplett an die Funktion übergeben. Diese liefert 0 zurück, falls der erste übergebene Simplex verfeinert werden soll und somit werden dessen 2^d Kinder in Reihenfolge der RFK dem neuen Wald übergeben. Wird eine Familie übergeben und ist der Rückgabewert kleiner als 0, so wird die Familie vergröbert, also durch den Elternsimplex ersetzt.

3.3 `Partition`

Für weitere Arbeiten mit dem Gitter ist es sinnvoll, dieses in Abschnitte zu partitionieren und somit gleichmäßig auf mehrere Prozesse zu verteilen (siehe Abb. 9). Dadurch soll erreicht werden, dass alle benutzten Prozesse gleich stark belastet sind und es keine großen Unterschiede im Aufwand pro Prozess gibt. Im `t8code` wird zum gleichmäßigen Verteilen von N Elementen auf P Prozesse folgende Formel verwendet:

Definition 16. *Wenn N Elemente auf P Prozesse aufgeteilt werden und wir jedem Element eine eindeutige Nummer entsprechend einer RFK in $\{0, \ldots, N-1\}$ zuordnen, dann werdem dem Prozess $i \in \{0 \leq i < P\}$ die Elemente S_i*

$$S_i \in \left\{ \left\lfloor \frac{Ni}{P} \right\rfloor, \ldots, \left\lfloor \frac{N(i+1)}{P} \right\rfloor - 1 \right\} \tag{27}$$

zugeteilt.

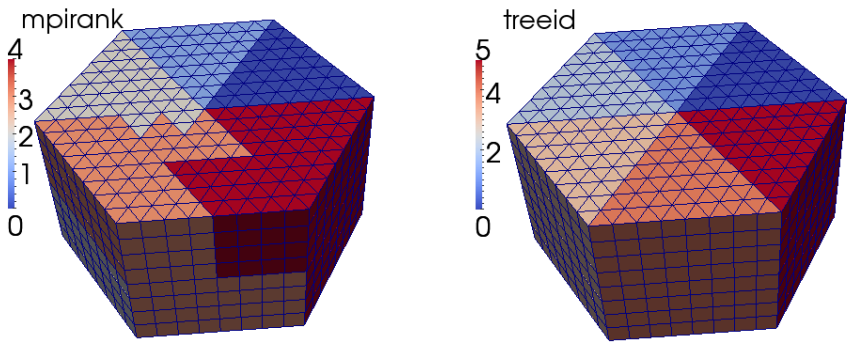

Abbildung 9: Eine Verfeinerung von sechs Prismen bis Level 3, nach einem Aufruf von `New`. Links: Die Aufteilung der Elemente auf fünf Prozesse. Jedem Prozess wird hierbei eine Farbe zugeordnet. Rechts: Die sechs Prismen und ihre uniforme Verfeinerung. Jedem initial gegebenem Prisma wird hierbei eine Farbe zugeordnet.

3.4 `Ghost`

Wenn auf dem Gitter diskretisierte GDG's und PDG's gelöst werden, wird oftmals ein Differenzenschema auf allen Punkten des Gitters angewandt [4]. Dieses bezieht auch Werte benachbarter Elemente mit ein. Da ein Prozess nicht die Informationen außerhalb seiner Partition kennt, müssen die Informationen der Randelemente mit den benachbarten Prozessen geteilt werden. Diese Randelemente werden Ghost genannt. Bis jetzt werden in den hier vorgestellen Algorithmen als Ghost Elemente nur Elemente berechnet, die sich eine gemeinsame Fläche teilen (siehe Abb. 10). Jedoch ist auch eine Verallgemeinerung auf Elemente, die eine gemeinsame Kante oder Ecke haben, möglich [3]. Wir beschränken uns in dieser Arbeit auf eine Version, welche nur mit balancierten Gittern arbeitet. Ein Gitter ist balanciert wenn das Verfeinerungslevel zweier Simplizes sich nicht um mehr als eins unterscheidet [11]. `Ghost` ermittelt für jeden Prozess alle Elemente, deren Nachbarn auf einem anderen Prozess liegen. Insgesamt gilt es jedoch eine starke Kommunikation zwischen den Prozessen über die Ghosts zu vermeiden, da diese sich sehr negativ auf die Laufzeit auswirkt [14]. Hierbei kann ein hoher Kommunikationsaufwand schon durch eine optimale Wahl der Grenzen der Partitionen verhindert werden [14]. Dies wird hier approximativ gelöst.

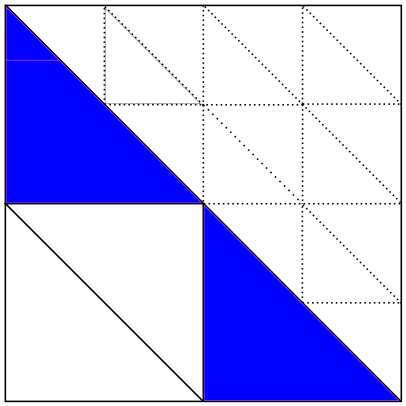

Abbildung 10: Ghost für Dreiecke für zwei Prozesse in einem balancierten Gitter. In durchgehenden Linien die Dreiecke für den 0-ten Prozess, in gestrichelten Linien die für den 1-ten Prozess. In blau die Ghosts von Prozess 1.

4 Low-level Algorithmen für Linien

Der Algorithmus zur Verfeinerung von Prismen basiert auf dem `t8_code` [8]. Der High-level Algorithmus, also das Erstellen des Gitters ist schon gegeben, jedoch müssen für neue Elementtypen die Low-level Algorithmen, welche steuern, wie verfeinert wird, wie die Kinder berechnet werden etc., konkret implementiert werden. Da in dieser Arbeit Prismen als Kreuzprodukt zwischen einer Linie und einem Dreieck aufgefasst werden, werden zunächst die Algorithmen für Linien implementiert. Hier werden die wichtigsten Algorithmen vorgestellt und vor allem die Theorie dazu erklärt. Im Folgenden besteht eine Linie L aus seiner Ankerkoordinate $L.\texttt{x}$ und einem Level $L.\texttt{level}$. Der Ankerknoten ist der Knoten, mit der kleinsten x-Koordinate der Linie. Analog zu Dreiecken und Rechtecken betrachten wir Linien auf der Referenzlinie $[0, 2^{\mathcal{L}}]$.

4.1 Initialiserung durch Übergabe der linearen ID

Die ID einer Linie L steht im direkten Zusammenhang mit der x-Koordinate von L. Da die Größe der ID durch das Level ℓ begrenzt ist, und durch die Binärdarstellung der ID immer gegeben ist, in welcher Hälfte die Linie sich nach einem Verfeinerungsschritt befindet, reicht es diese um $\mathcal{L}_1 - \ell$ Stellen zu bitshiften. Man kann anhand der Binärdarstellung also direkt den Weg im Baum zu L erkennen.

Algorithm 1 `t8_dline_init_linear_id`(Line L, int level, int id)

Ensure: $0 \leq level \text{ and } level \leq \mathcal{L}_1$
Ensure: $0 \leq id \text{ and } id \leq (1 \ll level)$
1: L.level $\leftarrow level$
2: L.x $\leftarrow id \ll (\mathcal{L}_1 - level)$

Analog dazu wird, um aus einer Linie ihre ID zu berechnen, in die andere Richtung geshiftet. Damit ergibt sich für eine Linie L und dem Level ℓ die Formel

$$\texttt{t8\_dline\_linear\_id} = L.\text{x}/2^{\mathcal{L}_1 - \ell} \tag{28}$$

4.2 Nachfolger

Als Nachfolger einer Linie wird die Linie vom selben Level mit dem nächsthöheren Index definiert. Auf einer uniform verfeinerten Referenzlinie ist es somit die nächste Linie rechts von der betrachteten Linie, da links mit dem Index 0 begonnen wird. Um die nächste Linie K einer Linie L mit Level ℓ zu berechnen, werden alle bits der Koordinate von L bis zum Level ℓ mit einer 0 überschrieben. Um die neue Koordinate zu berechnen, wird auf die x- Koordinate von L die Länge einer Linie vom Level ℓ hinzugerechnet. Bei

einer uniformen Verfeinerung vom Level ℓ sind die Koordinaten immer um 2^ℓ voneinander entfernt. Auch das Level von K muss am Ende aktualisiert werden.

Algorithm 2 `t8_dline_succesor`(Line L, Line succ, int level)

Ensure: $1 \leq level \; and \; level \leq L.\text{level}$

1: $h \leftarrow (1 \ll \mathcal{L} - level) - 1$
2: $succ.\text{x} \leftarrow L.\text{x}\&\neg h$
3: $succ.\text{x}+ = 2^{\mathcal{L}_1 - level}$
4: $succ.\text{level} \leftarrow level$

4.3 Berechnung der lokalen ID und Eltern

Zur Berechnung des Kindes wird eine lokale ID an die Funktion zur Berechnung des Kindes übergeben. Die lokale ID legt fest, ob eine Linie L das rechte oder das linke Kind ist. Ist es das linke Kind, so steht in der Binärdarstellung der Koordinate $L.\text{x}$ an der ℓ-ten Stelle eine 0, ansonsten eine 1. Dementsprechend genügt eine Abfrage des bits an der Stelle $L.\text{level}$ um die lokale ID zu berechnen.

$$\texttt{t8\_dline\_childid} = (L.\text{x} \gg (\mathcal{L}_1 - L.\text{level}))\&1 \tag{29}$$

Analog dazu wird zu Berechnung der Elternlinie der zur Kinderkoordinate gehörige Teil gelöscht und das Level neu gesetzt.

Algorithm 3 `t8_dline_parent`(Line L, Line parent)

Ensure: $L.\text{level} > 0$

1: $h \leftarrow 2^{\mathcal{L}_1 - L.\text{level}}$
2: $parent.\text{x} \leftarrow L.\text{x}\&\neg h$
3: $parent.\text{level} \leftarrow (L.\text{level} - 1)$

4.4 Berechnung der Kinder

Wird eine Linie verfeinert, so wird sie in zwei Kinder zerteilt. Anhand einer lokalen ID kann bestimmt werden, welches der Kinder berechnet werden soll. Ist es das linke Kind, so bleibt die Ankerkoordinate dieselbe, ist es das rechte Kind, so muss die Ankerkoordinate um die Länge einer Linie vom neuen Level verschoben werden.

Algorithm 4 `t8_dline_child`(Line L, int childid, Line child)

Ensure: $L.\text{level} < \mathcal{L}_1$
Ensure: $childid = 0$ *or* $childid = 1$
1: $h \leftarrow 2^{\mathcal{L}-L.\text{level}+1}$
2: $child.\text{x} \leftarrow L.\text{x} + (childid = 0 \ ? \ 0 : h)$
3: $child.\text{level} \leftarrow L.\text{level} + 1$

4.5 Überprüfung, ob zwei Linien zur selben Familie gehören

Es soll überprüft werden, ob zwei Linien L und K dieselbe Elternlinie haben und somit zur selben Familie gehören. Dazu müssen zunächst einmal das Level beider Linien identisch und die Koordinaten aufeinanderfolgende sein. Zudem sollen es keine Linien sein, die diese Bedingungen erfüllen, jedoch das 0. und 1. Kind zweier benachbarter Linien eines höheren Levels sind, da sie dann verschiedene Elternlinien hätten.

Algorithm 5 `t8_dline_is_familypv`(Line f[2])

1: **if** $f[0].\text{level} = 0$ *or* $f[0].\text{level} \neq f[1].\text{level}$ **then**
2: **return** 0
3: **else if** $f[0].\text{x} \gg (\mathcal{L}_1 - f[0].\text{level} + 1) \neq$
$f[1].\text{x} \gg (\mathcal{L}_1 - f[1].\text{level} + 1)$ **then**
4: **return** 0
5: **end if**
6: **return** $(f[0].\text{x} + len = f[1].\text{x})$

4.6 Zwei Linien vergleichen

Wir legen zunächst fest, wann eine Line L kleiner ist als eine Linie K. Dazu wird fest gelegt, wann allgemein ein Element im Gitter kleiner ist.

Definition 17. *Ein Element e ist kleiner als ein Element f, welches vom gleichen Elementtyp wie e ist, wenn bezüglich eines Levels die lineare ID von e kleiner als die von f ist. Ist sie gleich, so ist e kleiner, genau dann wenn e ein kleineres Level als f hat. Sie sind gleich, wenn e eine Kopie von f ist.*

Korollar 1. *Zwei Linien oder Prismen sind genau dann gleich, wenn sie die gleiche lineare ID und das gleiche Level haben.*

Aus dieser Definition lässt sich direkt der folgende Algorithmus ableiten.

Algorithm 6 `t8_dline_compare`(Line L_1, Line L_2)

1: $maxlvl \leftarrow max(L_1.\text{level}, L_2.\text{level})$
2: $id1 \leftarrow L_1.\text{x} \gg (\mathcal{L}_1 - maxlvl)$
3: $id2 \leftarrow L_2.\text{x} \gg (\mathcal{L}_1 - maxlvl)$
4: **if** $id1 = id2$ **then**
5: **return** $L_1.\text{level} - L_2.\text{level}$
6: **end if**
7: **return** $id2 - id1$

5 Konzept und Low-Level-Algorithmen für Prismen

Ein wesentlicher Teil der Arbeit besteht darin, das Konzept der RFK von Dreiecken auf Prismen zu übertragen und die Logik der Algorithmen für Prismen zu erklären. Als Ziel gilt, sich die schon bestehende Logik für Dreiecke und Linien zunutze zu machen und so einen gut verständlichen, aber auch schnellen Code zu schreiben. Der hier erarbeitete Code ergänzt die `t8code`-Bibliothek und ist in [9] zu finden. Ein Prisma besteht hierbei aus einem Dreieck *p.tri*, in dem x- und y- Koordinaten gespeichert werden, und einer Linie *p.line*, in welcher die z-Koordinate gespeichert wird. Diese drei Koordinaten sind jeweils Ankerknoten des Dreiecks oder der Linie und somit auch Ankerknoten des Prismas. Zudem wird das Level gespeichert. Darüber hinaus wird der Code so implementiert, dass eine Übertragung auf allgemeine Prismen einfach und praktikabel und möglich ist. Dies wird in dieser Arbeit jedoch nicht untersucht, alle Resultate wurden für Dreiecksprismen, die nach obigem Schema verfeinert werden, entwickelt und geprüft (siehe Def. 4 und Abb. 2). Die hier nicht vorgestellten, aber verwendeten Algorithmen für Dreiecke haben die gleiche Funktion wie die analogen Algorithmen für Prismen und sind in [1] zu finden.

5.1 Eine raumfüllende Kurve für Prismen

Beim Konstruieren der raumfüllenden Kurve für Prismen werden die schon bestehenden RFK für Dreiecke und Linien mitgenutzt. So wird zum Druchlaufen der Grundfläche eines Prismas eine Kurve für Dreiecke genutzt, zum Durchlaufen der Höhe eine Kurve für Linien. Wie auch in Kapitel 2.2.2 erfolgt die Konstruktion rekursiv und ähnelt einer Generation in einem Oktalbaum:

(i) Das zu verfeinernde Prisma, am Anfang das Einheitsprisma, wird in acht gleich große Prismen geteilt. Dazu wird das Basisdreieck geviertelt und das Prisma in der Höhe halbiert. Das Prisma soll also wie nach der Verfeinerungsregel in Definition 4 unterteilt werden.

(ii) Finde eine RFK für jedes Unterprisma.

Auch hier betrachten wir die Prismen auf einem Referenzprisma, welches im skalierten Würfel $[0, 2^{\mathcal{L}}]^3$ liegt (siehe Abb. 11). Wie bei der Berechnung des TM-Index ordnen wir jedem Prisma eine eindeutige ID zu, welche den Typ und den Ankerknoten des Prismas mit einbezieht.

Definition 18. *Jedes Prisma $P \in T_3$ von Level ℓ liegt in einem Würfel des kubischen Gitters, welches Teil der gleichmäßigen Verfeinerung von Level ℓ von $[0, 2^{\mathcal{L}}]^3$ ist. Dieser Würfel wird zu P assoziiert und W_P genannt. Jedes*

Prisma ist eine skalierte und verschobene Version von genau einem der zwei Prismen P_i aus dem Einheitswürfel und wir definieren den Typ von P als diese Zahl,

$$\text{type}(P) \coloneqq i. \tag{30}$$

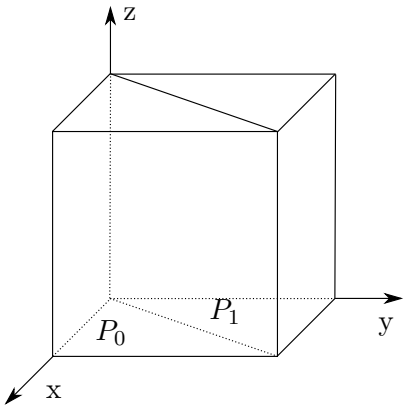

Abbildung 11: Die zwei Typen von Prismen, eingebettet in einem Würfel. Betrachten wir nur die Grundfläche des Prismas, so entsprechen die Typen des Prismas denen des Dreiecks.

Analog zu Dreiecken verwenden wir nicht nur den aktuellen Typ eines Prismas, sonder auch die Typen aller seiner Vorfahren.

Definition 19. *Für ein Prisma $P \in T_3$ von Level ℓ und jedem $0 \leq j \leq \ell$ sei P^j der (eindeutige) Vorgänger von P von Level j. Wir definieren $B(P)$ als das $\mathcal{L}-$Tupel mit den Typen von P's Vorgängern in den ersten ℓ Einträgen, beginnend mit P^1. Die letzen $\mathcal{L} - \ell$ Einträge von $B(P)$ sind Nullen:*

$$B = B(P) \coloneqq (\text{type}(P^1), \text{type}(P^2), \ldots, \text{type}(P^\ell), 0, \ldots, 0). \tag{31}$$

Der Ankerknoten eines Prismas ist analog zum Dreieck der Ankerknoten des Würfels, indem das Prisma liegt. Damit gibt es eine eindeutige Darstellung eines Prismas durch die Wahl eines Levels ℓ, einem Würfel von Level ℓ und des Types des Prismas.

Definition 20. *Für $P = [\vec{x}_0, \ldots, \vec{x}_5] \in T_3$ definieren wir die Prism-ID von P als das Tupel aus Ankerknoten und Typ von P:*

$$\text{Prism-ID} \coloneqq (\vec{x}_0, \text{type}(P)) \tag{32}$$

Korollar 21. *Seien $P,\ P' \in T_3$. Dann ist $P = P'$ genau dann, wenn ihre Prism-ID und Level gleich sind.*

Beweis. Sei $P = P'$. Damit ist ihr Level, ihr Typ und ihr Ankerknoten derselbe. Somit ist auch ihre Prism-ID gleich. Seien P, P' zwei Prismen mit der gleichen Prism-ID und gleichem Level. Da ihre Prism-ID die gleiche ist, stimmen ihr Typ und ihr Ankerknoten überein. Die einzige Eigenschaft, in der sie sich noch unterscheiden können, ist die Größe, also die Position ihrer Ecken bzw. in deren Koordinaten. Diese sind schon durch das Level gegeben, welches aber dasselbe ist. Somit gilt $P = P'$. □

Wie auch bei dem Index für Dreiecke geschieht der entscheidene Schritt zum berechnen des Index im bitweisen Überlagern der x-, y- und z-Koordinaten des Ankerknotens, sowie der Typen aller Vorfahren. Damit lässt sich auch sofort erkennen, dass die RFK das Prisma schichtweise durchläuft, also Ebene für Ebene. Jede einzelne Ebene wird wie ein Dreieck durchlaufen, da solange, wie die z-Koordinate unverändert bleibt, die RFK für Prismen der RFK für Dreiecke entspricht (siehe Abb. 12).

Definition 22. *Der Dreiecksprismen Index (Prism-Index) eines Dreiecksprismas $P \in T_3$ ist definiert als Überlagerung von den $\mathcal{L}-$Tupeln Z, Y, X, und B:*

$$m_p(P) \coloneqq Z \dot{\perp} Y \dot{\perp} X \dot{\perp} B \tag{33}$$

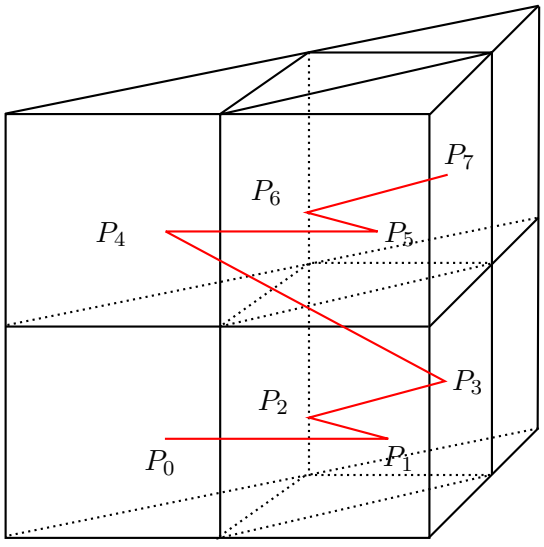

Abbildung 12: Die raumfüllende Kurve eines Prismas, welches um ein Level verfeinert worden ist. Zudem heben wir die lokale ID i der Prismen P_i hervor.

Schon hier erkennen wir, dass der Aufbau der RFK für Prismen sehr dem der RFK für Dreiecke oder Tetraeder ähnelt. Auch wenn sich das Muster stark von der des Tetraeders unterscheidet, so sind sie sich in der Theorie

ähnlich. Zudem besitzt auch die RFK für Prismen die Eigenschaft, dass die Verfeinerung eines Elementes die Kurve nur lokal ändert.

Theorem 23. *Für beliebige Dreiecksprismen $P \neq Q \in T_3$ erfüllt der Prism-Index m_p folgende Bedingungen:*

(i) *Wenn Q ein Nachfahre von P ist, dann gilt $m_p(P) \leq m_p(Q)$.*

(ii) *Wenn $l(P) < l(Q)$, dann ist $m_p(P)$ ein Präfix von $m_p(Q)$ genau dann, wenn Q ein Nachfahre von P ist.*

(iii) *Wenn $m_p(P) < m_p(Q)$ und Q kein Nachfahre von P ist, dann gilt für jeden Nachfahren P' von P :*
$m_p(P) \leq m_p(P') < m_p(Q)$.

Der Beweis dieses Theorems benötigt zuvor einige andere Resultate. Wie im Beweis der Eigenschaften für Tetraeder [1], zeigen wir eine Einbettung des Prism-Index in den Morton-Index für 4-dimensionale Würfel. Die Eigenschaften (i)-(iii) gelten für 4D-Würfel und somit gelten sie auch für Prismen. Da die Typen eines Prismas nur 0 und 1 sein können, können wir B als eine Binärzahl interpretieren. B besitzt insgesamt $\mathcal{L}$ Stellen, somit liegt B in $0 \leq B < 2^{\mathcal{L}}$. Sei nun $\mathcal{Q}$ die Menge aller 4D Würfel, welche Kind von $Q_0 \coloneqq [0, 2^{\mathcal{L}}]^4$ sind:

$$\mathcal{Q} = \{Q | Q \text{ ist ein Nachfahre von } Q_0 \text{ mit level } 0 \leq l \leq \mathcal{L}\} \tag{34}$$

Ein Würfel $Q \in \mathcal{Q}$ ist eindeutig definiert durch seine vier Koordinaten $\{x_0, \dots x_3\}$ seines Ankerknotens und sein Level ℓ. Somit lässt er sich schreiben als $Q = Q_{(x_0,\dots,x_3),l}$. Mit Hilfe dieser Darstellung lässt sich der Morton Index für Würfel als bitweises Überlagern der Binärdarstellung der Koordinaten des Ankerknoten definieren:

$$\tilde{m}(Q) = X^3 \dot{\perp} X^2 \dot{\perp} X^1 \dot{\perp} X^0. \tag{35}$$

Wir zeigen nun, dass der Prism-Index die Prismen eindeutig bestimmt, um als nächstes zu zeigen, dass der Prism-Index in den Morton-Index eingebettet werden kann.

Proposition 24. *Zusammen mit dem Verfeinerungslevel ℓ bestimmt der Prism-Index $m_p(P)$ eindeutig ein Prisma in T_3.*

Beweis. Für $\ell = 0$ existiert nur ein Prisma in T_3, nämlich P_3^0. Seien nun $\ell > 0$ und $m = m_p(P)$ gegeben wie in (31) und sei ℓ das Level von P. Durch m lassen sich die x-, y- und z-Koordinaten des zu P assoziierten Würfels sowie der Typ des Prismas bestimmen. Dazu muss die Überlagerungsoperation umgekehrt werden. Nach dem Überlagern von 4 n-Tupeln entsteht ein 4n-Tupel. An jeder vierten Stelle steht also eine Stelle der x-, y- und z-Koordinaten, sowie des Typen. Nach Korollar 15 reicht dies aus, um ein Prisma eindeutig zu bestimmen. □

Proposition 25. *Die Abbildung*

$$\Phi : T_3 \to \mathcal{Q} \tag{36}$$

$$P \mapsto Q_{(B(P),x(P),y(P),z(P)),l(P)} \tag{37}$$

ist injektiv und erfüllt

$$\tilde{m}(\Phi(P)) = m_p(P). \tag{38}$$

Zudem erfüllt sie die Eigenschaft, dass P' ein Kind von P ist, genau dann wenn, $\Phi(P')$ ein Kind von $\Phi(P)$ ist.

Beweis. Die Gleichung $m_p(P) = \tilde{m}(\Phi(P))$ folgt aus der Definition des Prism-Indexes für T_3 und des Morton-Indexes für $\mathcal{Q}$. Durch Proposition 23 lässt sich schließen, dass Φ injektiv ist. Sei nun $P, P' \in T_3$, wobei P' ein Kind von P ist und ℓ das Level von P. Zusätzlich wissen wir, dass $Q' \coloneqq \Phi(P')$genau dann ein Kind von $Q \coloneqq \Phi(P)$ ist, wenn für jedes $i \in \{0, \dots, 3\}$

$$x_i(Q') \in \{x_i(Q), x_i(Q) + 2^{\mathcal{L}-(\ell+1)}\} \tag{39}$$

gilt. Die Koordinaten des Ankerknoten dürfen also nicht kleiner sein, als die des Elternelements, aber auch nur genauso groß wie es das aktuelle Level zulässt. Da auch den Prismen eine Würfelstruktur zugrunde liegt, folgt für die x-Koordinate des Ankerknotens von P', dass

$$X(P') \in \{x(P), x(P) + 2^{\mathcal{L}-(\ell+1)}\} \tag{40}$$

und analoges für $Y(P')$ und $Z(P')$. Damit gilt die Eigenschaft (40) schon für $i = 1, 2, 3$. Für das Prisma P mit Level ℓ und dem Kind P' gilt, dass die ersten ℓ Stellen von $B(P) = B(P')$ sind. Alle Kinder von P, die am Rand liegen, haben den gleichen Typ wie P. Hat P' die lokale ID 2 oder 6 liegt es in der Mitte und ist vom anderen Typ. Die nächste Stelle von B wird also wieder eine 0 oder eine 1 sein, also ist $B(P')$ gleich denen von $B(P)$, oder $B(P') = B(P) + 2^{\mathcal{L}-(\ell+1)}$.
Damit gilt die Eigenschaft (39) weiterhin und somit auch für $i = 0$. Also ist $\Phi(P')$ ein Kind von $\Phi(P)$.
Wir nehmen an, dass $\Phi(P')$ ein Kind von $\Phi(P)$ ist, um die restlichen Folgerungen zu zeigen. Da $l(P') = l(\Phi(P')) > 0$, hat P' einen Elter, den wir R nennen. Im vorherigen Argument wurde gezeigt, dass $\Phi(R)$ der Vorfahre von $\Phi(P')$ ist, und da jeder Würfel einen eindeutigen Vorfahre hat, muss $\Phi(R) = \Phi(P)$ gelten. Mit der Injektivität von Φ gilt dann, dass $R = P$ und P' ist ein Kind von P. □

Durch ein induktives Argument kann gezeigt werden, dass P' ein Nachfahre von P ist, genau dann wenn $\Phi(P')$ ein Nachfahre von $\Phi(P)$ ist. Somit folgt Theorem 22, da die Eigenschaften (i) bis (iii) für den Morton Index für Würfel gelten.

5.2 Initialisierung durch Übergabe der linearen ID

Um anhand der linearen ID ein Prisma P zu initialisieren, berechnen wir zunächst einmal die entsprechenden linearen IDs des Dreiecks $p.tri$ und der Linie $p.line$. Dazu betrachten wir in beiden Fällen die lokalen IDs aller Elternprismen. Analog zur Linie lesen wir die Koordinaten von P über eine Oktaldarstellung der ID ab. Dabei zeigt jede Stelle der Darstellung die lokale Lage des Prismas in einem Prisma dem der Stelle entsprechenden Level an. Da immer zwei übereinander liegenden Prismen dasselbe Dreieck zugeordnet wird, lässt sich direkt auch die lokale Dreiecks-ID ablesen. Da in jedem Verfeinerungsschritt die Dreiecke geviertelt werden, wird die lokale ID mit 4^{level} für das gerade betrachtete Level multipliziert. Zur Berechnung der Linien-ID wird geprüft, ob das Prisma P im gerade betrachteten Level in der oberen oder unteren Hälfte des Elternprismas liegt. Falls dies der Falls ist, so wird 2^{level} zu der Linien-ID addiert.

Algorithm 7 `t8_dprism_init_linear_id`(Prism p, int id)

Ensure: $id < 2^{3p.\text{level}}$

1: $tri\_id \leftarrow 0, line\_id \leftarrow 0, triangles \leftarrow 1$
2: **for** $i = 0$ **to** $i = level$ **do**
3: $\quad tri\_id+ = (id \ \% \ \mathcal{TC}) \cdot triangles$
4: $\quad line\_id+ = (id \ \% \ \mathcal{PC}) \ / \ \mathcal{TC} \ \cdot \mathcal{LC}^i$
5: $\quad id \ / = \ \mathcal{PC}$
6: $\quad triangles \cdot = \ \mathcal{TC}$
7: **end for**
8: `t8_dtri_init_linear_id`(p.tri, $tri\_id, level$)
9: `t8_dline_init_linear_id`(p.line, $line\_id, level$)

5.3 Berechnung der linearen ID

Wie in 5.2 verwenden wir auch hier zur Berechnung der linearen ID die schon bestehenden Algorithmen für Dreiecke und Linien. So berechnen wir zuerst die zugehörige Linien- und Dreiecks-ID eines Prismas. Durch die Dreiecks-ID lässt berechnet sich die ID eines Prismas, welches in der untersten Ebene eines mit Level ℓ verfeinerten Prismas liegt. Hierzu bestimmen wir wieder die lokale Lage des Dreiecks in jedem Level und multiplizieren sie mit 8^ℓ. Durch die Linien-ID soll die ID des Prismas aus der passenden Ebene bestimmt werden, also der oberen oder der unteren Ebene des Prismas. Dazu wird für jedes Level geprüft, ob das Prisma in der oberen oder unteren Hälfte des Prismas liegt. In einer Hälfte des einen Prismas mit Level ℓ liegen $4 \cdot 8^{\ell-1}$ Prismen, somit wird dies zur Prismen-ID addiert, wenn das Prisma in der oberen Hälfte liegt. Dies stellen wir in Algorithmus 8 dar.

Algorithm 8 `t8_dprism_linear_id`(Prism p, int level)

Ensure: $0 \leq level \text{ and } level \leq \mathcal{L}_4$

1: $tri\_id \leftarrow$ `t8_dtri_linear_id`$(p.\text{tri}, level)$,
$line\_id \leftarrow$ `t8_dline_linear_id`$(p.\text{line}, level)$
$line\_level \leftarrow \mathcal{LC}^{\texttt{level}-1}$,
$prism\_shift \leftarrow \mathcal{PC}/\mathcal{LC} \cdot \mathcal{PC}^{\texttt{level}-1}$
$prisms \leftarrow 1,\ id \leftarrow 0$
2: **for** $i = 0$ **to** $i = level - 1$ **do**
3: $id+ = (tri\_id \ \% \ \mathcal{TC})prisms$
4: $tri\_id/ = \mathcal{TC}$
5: $prisms\cdot = \mathcal{PC}$
6: **end for**
7: **for** $i = level - 1$ **to** $i = 0$ **do**
8: $id+ = (line\_id/line\_level)prism\_shift$
9: $line\_id \leftarrow line\_id \ \% \ line\_level$
10: $prism\_shift/ = \mathcal{PC}$
11: $line\_level/ = \mathcal{LC}$
12: **end for**
13: **return** id

5.4 Nachfolger

Um den Nachfolger Q eines Prismas P zu berechnen, wird zwischen drei Fällen unterschieden.

Fall 1: Die lokale ID von P ist 7. Das Prisma Q ist das erste Kind des nächsten Prismas mit Level $\ell - 1$. Dazu setzen wir Q auf das nächste Prisma mit Level $\ell - 1$ und aktualisiert das Level auf ℓ.

Fall 2: Die lokale ID von P ist 3. Das nächste Prisma ist das vierte Kind des Elternprismas.

Fall 3: Die lokale ID von P ist nicht 3 oder 7. Hier kann einfach der Nachfolger des Dreiecks berechnet werden, da die Linie des Prismas Q dieselbe bleibt.

In Fall 1 bestimmen wir das nachfolgende Prisma rekursiv, da es möglich ist, dass das Elternprisma auch die lokale ID 7 hat. Für eine allgemeinere Version bestimmen wir im Fall 1 also das letzte Prisma, in Fall 2 das letzte Prisma einer Ebene (siehe Abb. 13).

Algorithm 9 `t8_dprism_successor`(Prism p, Prism succ, int level)

Ensure: $1 \leq level \text{ and } level \leq \mathcal{L}$

1: $prism_child_id = $ `t8_dprism_child_id`(p)
2: **if** $prism\_child\_id = \mathcal{PC} - 1$ **then**
3: `t8_dprism_successor`$(p, succ, level - 1)$
4: $succ$.level $\leftarrow level$
5: **else if** $(prism\_child\_id + 1)\%\mathcal{TC} = 0$ **then**
6: `t8_dprism_parent`$(p, succ)$
7: `t8_dprism_child`$(succ, prism\_child\_id + 1, succ)$
8: **else**
9: `t8_dtri_successor`$(p$.tri$, succ$.tri$, level)$
10: **end if**

Der Nachfolger eines Prismas wird im Durchschnitt in konstanter Laufzeit berechnet und das trotz des rekursiven Aufrufs. Jede Operation des Algorithmus kann in konstanter Zeit ausgeführt werden. Die durchschnittliche Laufzeit ist hierbei nc, wobei c eine Konstante (unabhängig von $\mathcal{L}$) ist und $n - 1$ ist die durchschnittliche Anzahl an Rekursionsstufen. Da die Rekursion nur für das Prisma mit der lokalen ID 7 aufgerufen wird, lässt sich daraus schließen, dass die Rekursion auch nur in jedem 2^d-ten Schritt aufgerufen wird. Mit einem Argument über die geometrische Reihe folgt, dass im Durchschnitt eine konstante Laufzeit erreicht wird.

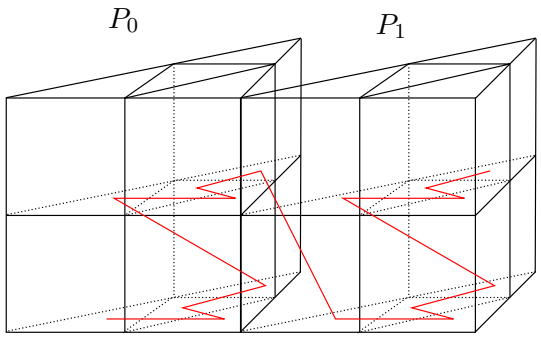

Abbildung 13: Die Kinder P_0 und P_1 eines um zwei Level verfeinerten Prismas, zur Verdeutlichung des Nachfolgers des Prismas mit der lokalen ID 7 von P_0. Außerdem sieht man den Nachfolger der Prismen mit lokaler ID 3 der Prismen P_0 und P_1. Diese müssen jedoch nicht unbedingt so aufeinander folgen, wie hier eingezeichnet.

5.5 Die lokale ID berechnen

Um eine direkte Beziehung zwischen einem Prisma und seinen Kindern herzustellen, gibt es die lokale ID. Sie ordnet den Kindern eines Prismas die ID $0, \cdots, 7$ anhand der Reihenfolge, wie die RFK die Kinder durchläuft zu.

Da sich übereinanderliegende Prismen dieselbe lokale Dreiecks-ID teilen und sich in diesem Fall die lokale Prismen-ID nur durch die lokale Linien-ID von der lokalen Dreiecks-ID unterscheidet, benutzen wir beide lokale IDs um die lokale ID eines Prismas zu berechnen.

Definition 26. *Für ein Prisma P und die dazu gehörigen lokalen IDs von P.*tri *und P.*line *ergibt sich für die lokale ID von P:*

$$\text{prism\_child\_id}(P) = \text{tri\_child\_id}(P.\text{tri}) + \text{line\_child\_id}(P.\text{line})\mathcal{TC} \tag{41}$$

5.6 Berechnung der Kinder

Für die Berechnung eines Kindes K eines Prismas P übergeben wir die lokale ID von K. Anhand dieser lassen sich die entsprechenden lokalen IDs für Dreiecke und Linien berechnen. So gilt für die entsprechenden IDs

$$tri\_id \;=\; \text{ID } \%\ 4 \tag{42}$$
$$line\_id \;=\; \text{ID } /\ 4 \tag{43}$$

Die Berechnung von K wird dann durch einen Aufruf der entsprechenden Kinderfunktionen für Linien und Dreiecke mit der entsprechenden ID durchgeführt.

5.7 Überprüfen, ob Prismen eine Familie bilden

Diesem Algorithmus werden acht Prismen übergeben, welche nach dem Prism-Index sortiert sind. Bilden sie eine Familie, so wird eine Zahl ungleich Null zurückgegeben. Sie bilden eine Familie wenn die Basisdreiecke auf jeder Ebene eine Familie bilden und alle Linien von Prismen, die übereinander liegen, eine Familie bilden. Zudem sind die Dreiecke von übereinander liegenden Prismen die gleichen sein.

Algorithm 10 `t8_dprism_is_family`(Prism[$\mathcal{PC}$] fam)

1: $is\_family = 1$
2: **for** $i = 0$ TO $i < \mathcal{LC}$ **do**
3: $\quad is\_family = is\_family$ *and* `t8_dtri_is_family`($fam[i\mathcal{TC}]$.tri)
4: **end for**
5: **for** $i = 0$ TO $i < \mathcal{TC}$ **do**
6: $\quad is\_family = is\_family$ *and* `t8_dline_is_family`($fam[i\mathcal{LC}]$.line)
7: $\quad is\_family = is\_family$ *and* `t8_tri_is_equal`($fam[i]$.tri, $fam[i + \mathcal{TC}]$.tri)
8: **end for**
9: **return** is_family

5.8 Zwei Prismen vergleichen

Dieser Algorithmus basiert analog zum Algorithmus 4.6 auf der Definition 17. Ob ein Prisma P größer als ein Prisma P' ist, hängt abermals von der ID und dem Level der Prismen ab. Dementsprechend ist der Algorithmus bis auf die Berechnung der ID identisch zu Algorithmus 6 und wird deswegen nicht noch einmal explizit vorgestellt.

5.9 Erster und letzter Nachfahre

Der erste/letzte Nachfahre eines Prismas ist der erste/letzte Nachfahre der zugrunde liegenden Linie und des Dreiecks. Das zu berechnende Prisma speichern wir in S, das übergebene Prisma heißt P.

Algorithm 11 `t8_dprism_first_descendant`(Prism P, Prism S, int level)

1: `t8_dtri_first_descendant`(P.tri, S.tri, level)
2: `t8_dline_first_descendant`(P.line, S.line, level)

Der Algorithmus für den letzten Nachfahren ist analog.

5.10 Vom Prisma zur Randfläche

Mit diesem Algorithmus wird zu einem gegebenen Prisma eine dazugehörige Randfläche berechnet. Wir benotigen dies, um die Flächennachbarn eines Prismas zu berechnen. Dies ist notwendig, damit wir die Ghostelemente berechnen können. Je nachdem welcher Rand des Prismas berechnet wird, wird ein Dreieck oder ein Viereck berechnet. Jedoch entspricht das Koordinatensystem in der Fläche nicht dem des Prismas und wir müssen es somit überführen. Allerdings stimmen Teile der Koordinatenachsen überein, somit müssen nur noch die Längen des Vierecks an die ursprüngliche Länge des Prismas angepasst werden. So entspricht für ein Rechteck in der zweiten Randfläche im Prisma die x-Achse des Rechtecks der x-Achse im Prisma, jedoch enspricht die y-Achse der z-Achse im Prisma. Bei der Dreiecksfläche ist dies jedoch nicht nötig, da ein Prisma ein Dreieck direkt mit speichert und wir hier nur die Koordinaten übertragen (siehe Abb. 14). Um schnell zu wissen, welche Fläche eines Prismas gemeint ist, ordnen wir jeder Fläche des Prismas eine eindeutige Zahl nach folgendem Schema zu (siehe Abb. 14).

Definition 27. *Die fünf Oberflächen eines Prismas* $P = [\vec{x}_0, \vec{x}_1, \vec{x}_2, \vec{x}_3, \vec{x}_4, \vec{x}_5]$ *sind gegeben durch:*

$$\text{face } 0 \coloneqq [\vec{x}_1, \vec{x}_2, \vec{x}_4, \vec{x}_5], \qquad \text{face } 1 \coloneqq [\vec{x}_0, \vec{x}_2, \vec{x}_3, \vec{x}_5] \tag{44}$$

$$\text{face } 2 \coloneqq [\vec{x}_0, \vec{x}_1, \vec{x}_3, \vec{x}_4], \qquad \text{face } 3 \coloneqq [\vec{x}_0, \vec{x}_1, \vec{x}_2] \tag{45}$$

$$\text{face } 4 \coloneqq [\vec{x}_3, \vec{x}_4, \vec{x}_5] \tag{46}$$

Die Vektoren beschreiben das durch sie aufgespannte Dreieck oder Rechteck.

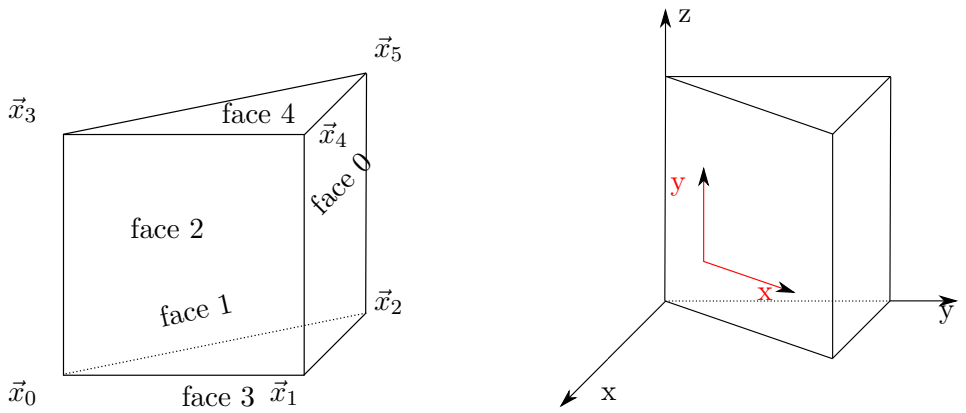

Abbildung 14: Links: Die Numerierung der Flächen eines Prismas, nach obiger Definition. Rechts: Das Koordinatensystem des Prismas und in rot das Koordinatensystem der rechteckigen Randfläche.

Im Algorithmus wurde dem Objekt boundary noch kein genauer Typ zugeordnet, da es entweder ein Dreieck, oder ein Rechteck wird.

Algorithm 12 `t8_dprism_boundary_face`(Prism p, int face, boundary)

Ensure: $0 \leq face < 5$
1: **if** $face \geq 3$ **then**
2: $t8\_dtri\_copy(p.\text{tri}, boundary)$
3: **return**
4: **end if**
5: **switch** $(face)$
6: **case** 0**:**
7: $boundary.x = 2^{\mathcal{L}_3 - \mathcal{L}_2} p.\text{tri.y}$
8: $boundary.y = 2^{\mathcal{L}_3 - \mathcal{L}_1} p.\text{line.x}$
9: **case** 1**:**
10: $boundary.x = 2^{\mathcal{L}_3 - \mathcal{L}_2} p.\text{tri.x}$
11: $boundary.y = 2^{\mathcal{L}_3 - \mathcal{L}_1} p.\text{line.x}$
12: **case** 2**:**
13: $boundary.x = 2^{\mathcal{L}_3 - \mathcal{L}_2} p.\text{tri.x}$
14: $boundary.y = 2^{\mathcal{L}_3 - \mathcal{L}_1} p.\text{line.x}$
15: **case** default**:**
16: ABORT
17: **end switch**

5.11 Von der Randfläche zum Prisma

Analog zum Algorithmus t8_dprism_boundary_face wird hier aus einer gegebenen Randfläche ein Prisma berechnet. Wir brauchen ihn ebenfalls, um die Ghostelemente zu berechnen Dazu wird abermals darauf geachtet, dass die

Koordinatensysteme unterschiedlich gesetzt sind. Hierbei muss jedoch das Koordinatensystem der Randfläche in das Koordinatensystem des Prismas transformiert werden. Es entspricht also einer Umkehrfunktion zum Algorithmus 12. Da der Algorithmus 12 das Prinzip schon verdeutlicht, werden hier nur noch die entsprechenden Werte für die Koordinaten gezeigt.

	face 0	face 1	face 2	face 3	face 4
x	$2^{\mathcal{L}_2} - 2^{\mathcal{L}_2 - p.\text{level}}$	$2^{\mathcal{L}_2 - \mathcal{L}_3} face.\text{x}$	$2^{\mathcal{L}_2 - \mathcal{L}_3} face.\text{x}$	$face.\text{x}$	$face.\text{x}$
y	$2^{\mathcal{L}_2 - \mathcal{L}_3} face.\text{x}$	$2^{\mathcal{L}_2 - \mathcal{L}_3} face.\text{x}$	0	$face.\text{y}$	$face.\text{y}$
z	$2^{\mathcal{L}_1 - \mathcal{L}_3} face.\text{y}$	$2^{\mathcal{L}_1 - \mathcal{L}_3} face.\text{y}$	$2^{\mathcal{L}_1 - \mathcal{L}_3} face.\text{y}$	0	$2^{\mathcal{L}_1} - 2^{\mathcal{L}_1 - p.\text{level}}$

Tabelle 2: Die Koordinaten und ihre entsprechenden Werte, um von einer Randfläche auf das dazugehörige Prisma zu schließen.

5.12 Visualisierung der Prismen

Am Ende des Programms ist es oftmals wünschenswert, auch eine visuelle Darstellung der Ergebnisse zu haben. Der `t8code` bietet eine Möglichkeit, mit Hilfe des Programms Paraview, die Ergebnisse zu visualisieren. Hierzu werden Punkte auf dem Prisma interpoliert. Dazu bekommt der Algorithmus die Koordinaten aller Ecken des Prismas übergeben. Die Interpolation geschieht dann in zwei Schritten. Zunächst wird über die Höhe interpoliert. Somit entsprechen die noch nicht miteinbezogenen Werte einem Dreieck, welches in dem Prisma liegt. Im zweiten Schritt wird über genau dieses interpoliert. Zudem muss gleichzeitig von den Koordinaten des Referenzprismas auf die Koordinaten des ursprünglichen Prismas transformiert werden. Im Pseudocode entspricht corner_coords den Koordinaten einer Ecke im Referenzprisma, prism_coords den Koordinaten der Ecken des Prismas und corner_num der Nummer einer Ecke im Prisma. Damit lassen sich die 9 Koordinaten der drei Ecken des Dreiecks durch

$$\begin{aligned}\text{tri\_vertices}[i] = 2^{-\mathcal{L}_4} \cdot (&\text{prism\_coords}[9+i] - \text{prism\_coords}[i]) \\ &\cdot \text{corner\_coords}[2] + \text{prism\_coords}[i]\end{aligned} \tag{47}$$

berechnen, wobei $i \in \{0, 1, \cdots 8\}$ ist. Mithilfe der Dreieckskoordinaten gilt dann für die Koordinaten coords des zu berechnenden Punktes:

$$\begin{aligned}\text{coords}[i] = &\ 2^{-\mathcal{L}_4} \cdot (\text{tri\_vertices}[3+i] - \text{tri\_vertices}[i])\text{corner\_coords}[0] \\ &+ 2^{-\mathcal{L}_4} \cdot (\text{tri\_vertices}[6+i] - \text{tri\_vertices}[3+i]) \\ &\cdot \text{corner\_coords}[1] + \text{tri\_vertices}[i]\end{aligned} \tag{48}$$

6 Laufzeittests für Prismen

Neben dem Ziel, dass die Implementierung des Prism-Indexes auf dem Code von Linien und Dreiecken basiert, ist auch wichtig, dass die Implementierung schnell und parallel läuft. Es werden ähnlich schnelle Laufzeiten für Prismen wie für Tetraeder gemessen. Zudem erwarten wir bei der hier vorgestellten Implementierung, dass die Laufzeit nahezu proportional zur Anzahl der Elemente und unabhängig vom Verfeinerungslevel ist. Alle vorgestellten Tests wurden mit der Version [9] des `t8code` getestet. Hierzu werden Laufzeit- und Skalierungstests für `New` und `Adapt` auf dem JUQUEEN Supercomputer des Forschungszentrums Jülich getestet [13]. JUQUEEN ist ein IBM BlueGene/Q System mit 28,672 Knoten, welche jeweils aus 16 IBM PowerPC A2@1.6 GHz und 16 GB Ram pro Knoten bestehen. Um einen Vergleich zur Laufzeit und Skaliebarkeit der Tetraeder möglich zu machen, testen wir mit einem analogen Test. So wird zuerst die starke Skalierbarkeit (bis 131072 Prozesse) und die Laufzeit für `New` getestet. Dies wird auf einem Grobgitter mit 512 Prismen getestet. Die dabei maximal erreichte Anzahl an Prismen ist ca $6.8 \cdot 10^{10}$ mit $5 \cdot 10^5$ Elementen pro Prozess und 131072 Prozessen. Die Zeit des `New` Algorithmus wird für das Input Level 7, (bzw. Level 10 für eine größere Anzahl an Prozessen) getestet. Zudem soll die Laufzeit mit der von Tetraedern verglichen werden, welche sich ebenfalls in acht Kinder teilen. Somit werden bei beiden Tests gleichviele Tetraeder und Prismen verfeinert. Im Unterschied zu Prismen werden jedoch keine schon bestehenden Algorithmen mitbenutzt. Außerdem haben Tetraeder nur 4, anstatt 6 Ecken, sie sind somit kleiner. Es ist also zu erwarten, dass der `t8code` für Tetraeder schneller läuft als für Prismen. Die Laufzeittests für Tetraeder wurden nicht aus [1] entnommen sondern noch einmal berechnet, da sich der `t8code` seit der Veröffentlichung des Artikels stark erweitert und verändert hat. Die Prinzipien und Ideen sind jedoch dieselben geblieben. In Abb. 6 lässt sich erkennen, dass die Skalierung fast gar nicht von der idealen Skalierung abweicht.

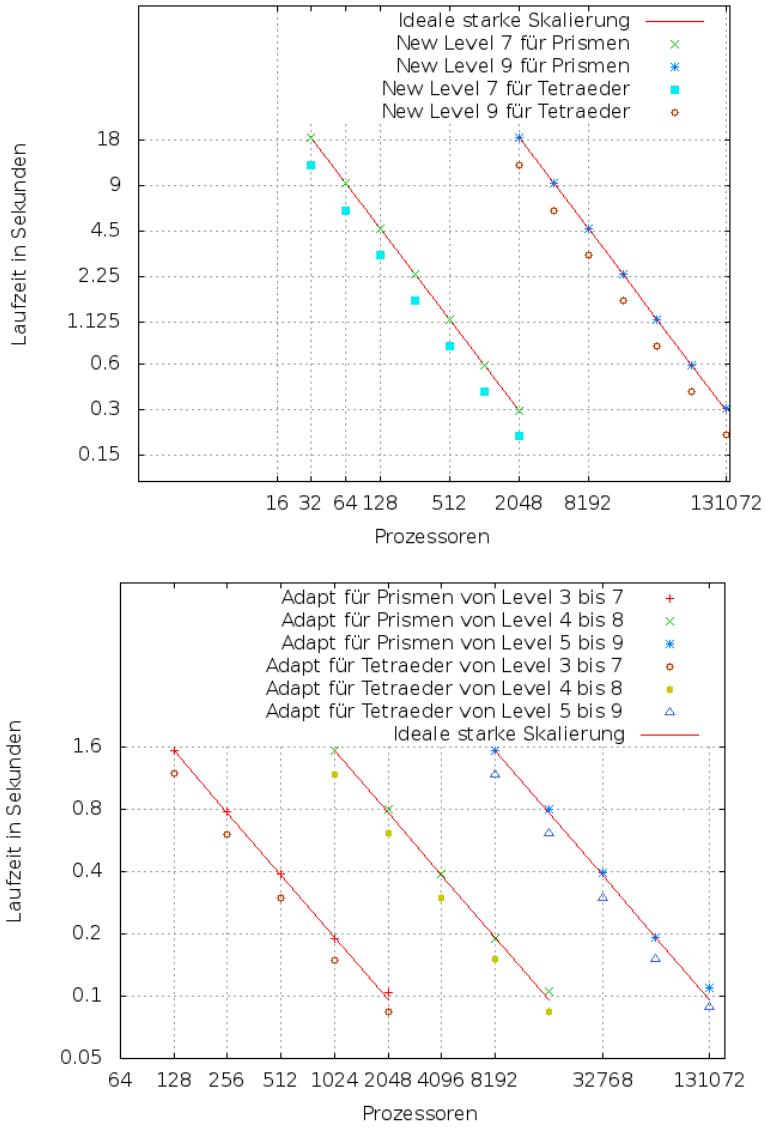

Abbildung 15: Laufzeittest auf JUQUEEN. In beiden Tests wurde `New` und `Adapt` auf einem Gitter mit 512 Prismen getestet. Für `Adapt` wurden alle Prismen von Typ 0 mit initialem Level `k` auf das Level `k+4` verfeinert. Auch für Tetraeder wurde in beiden Tests ein Gitter mit 512 Tetraedern gebaut. Beim Verfeinern eines Tetraeders entstehen sechs verschiedene Typen von Tetraedern, für `Adapt` wurden alle Tetraeder vom Typ 0, 2 und 4 verfeinert. Somit entstehen bei `New` für Prismen und Tetraeder maximal $6,8 \cdot 10^{10}$ Elemente. Einem Prozess werden hierbei max. $8 \cdot 10^6$ Elemente zugeordnet. Die exakten Messwerte für `New` können im Anhang in Tabelle 3 nachgeschaut werden, für `Adapt` in Tabelle 4 und 5.

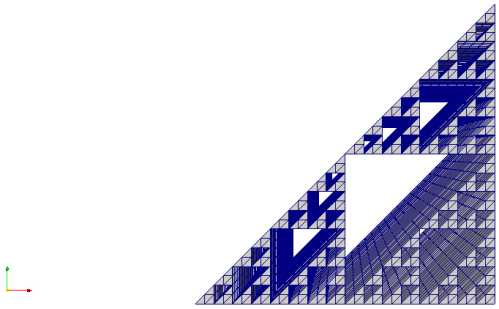

Abbildung 16: Eine Verfeinerung eines Prismas nach dem fraktalen Muster mit initialem Level 0 und Ziellevel 4. Die Unterprismen von Typ 1 bis Level 3 sind nicht eingezeichnet. Die Prismen von Typ 1 mit Level 4 sind weiterhin zu sehen da das Ziellevel dort erreicht wurde. Die y-Achse des Koordinatensystems zeigt zum Betrachter. Zudem ist es eine perspektivische Abbildung, in der die Kamera links oben positioniert ist.

In einem zweiten Test testen wir die Laufzeit für `Adapt`. Hierzu wird das gleiche Gitter wie im vorherigen Test verwendet, jedoch wird das Gitter nach dem Aufruf von `New` noch einmal um vier Level adaptiv verfeinert. Bei Prismen werden nur solche vom Elementtyp 0, bei Tetraedern nur solche vom Typ 0,2 und 4 verfeinert. Das dabei entstehende Muster entspricht einer Übertragung des Sierpinski-Dreiecks auf ein Prisma (siehe Abb. 16). Auch hier wird eine ideale Skalierung erreicht, welches sich ebenfalls in Abbildung 14 erkennen lässt.
Bei dem dritten Test werden die Laufzeiten von `Partiton`, `Adapt` und `Ghost` getestet. Dafür wird ein Zylinder durch 511 Prismen approximiert. Um das Experiment komplexer zu gestalten, wird ein besonderes Verfeinerungsschema gewählt.
Der Zylinder wird immer dort verfeinert, wo eine schräg gestellte Wand von einer beliebig gewählten Dicke den Zylinder schneidet. Zudem durchwandert die Wand den Zylinder, sie geht vier Zeitschritte lang eine bestimmte Schrittweite weiter. Dies hat zur Folge, dass Elemente nicht nur verfeinert werden müssen, sondern auch wieder vergröbert. Wird das Level um k höher gesetzt, muss darauf geachtet werden, dass die Schrittweite, der Start- und Endpunkt der Wand und das Zeitinterval so gewählt werden, dass nur 8^k mehr Elemente in der Wand liegen.

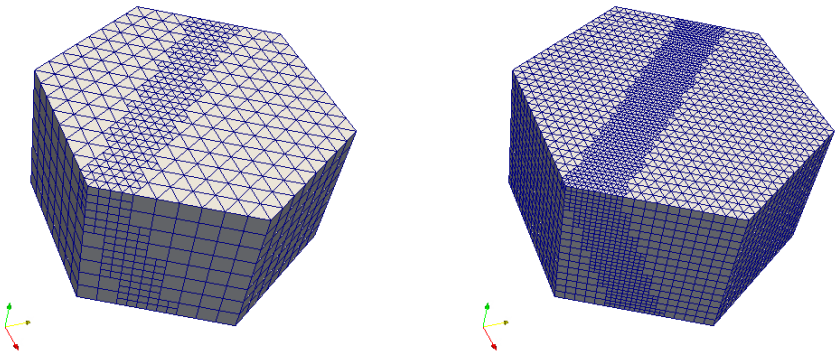

Abbildung 17: Ein Zylinder, approximiert durch sechs Prismenbäume, wird mit einer schrägen Wand geschnitten und an der Schnittstelle verfeinert. Links ist das initiale Level 3, rechts ist es 4. Um das Gitter durchgehend balanciert zu halten, wird nur um ein Level verfeinert.

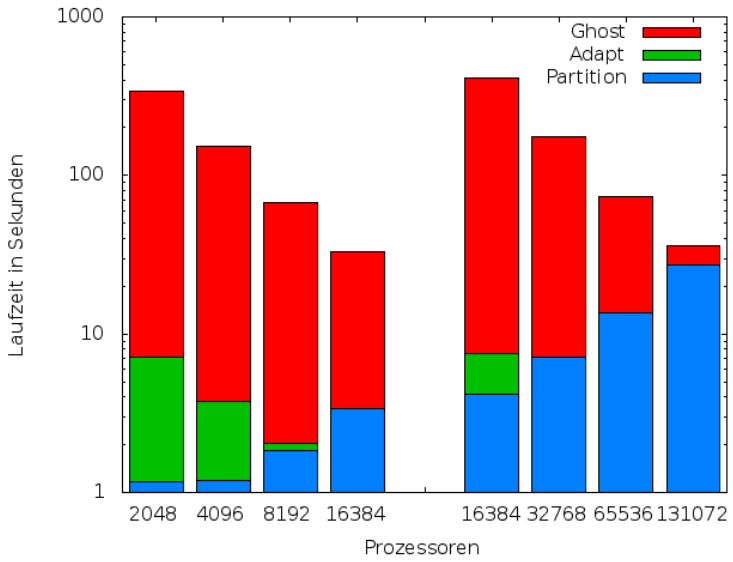

Abbildung 18: Die Ergebnisse des dritten Laufzeittests. Der Test wurde zuerst auf bis zu 16384 Prozessen für ein initiales Gitter mit 511 Prismen und einem anfänglichem Verfeinerungslevel 5 durchgeführt. Die Wand wurde für vier Zeitschritte verschoben. Die letzten vier Säulen stehen für denselben Test, jedoch wurden die Prismen am Anfang um 6 Level verfeinert und es wurde auf mindestens 16384 Prozessen getestet. Die exakten Messwerte können im Anhang in Tabelle 6 nachgeschaut werden. Für eine Verfeinerung von Level 5 entstehen somit ca. $16 \cdot 10^6$ und für Level 6 $134 \cdot 10^6$ Prismen.

Auch im dritten Test wird eine sehr gute Skalierung erreicht, eine Abweichung zur idealen Skalierung lässt sich mitunter dadurch erklären, dass die Anzahl der Ghostelemente nicht linear mit dem Level ansteigt da sie von der Oberfläche der Partition abhängt und nicht vom Volumen. Damit die Ergebnisse reproduziert werden können, geben wir die Parameter des Programms an. Für eine initiale Verfeinerung von Level 5, wählen, wir als x-Koordinate der Wand 0.09 und ein Zeitschritt wird 0.02 lang gewählt. In beiden Test wird die Zeitspanne viermal so lang wie der Zeitschritt gewählt. Bei einer initialen Verfeinerung von Level 6 ist die x-Koordinate der Wand 0.105 und ein Zeitschritt 0.01. Die Wand ist in beiden Test 0.3 Einheiten dick.

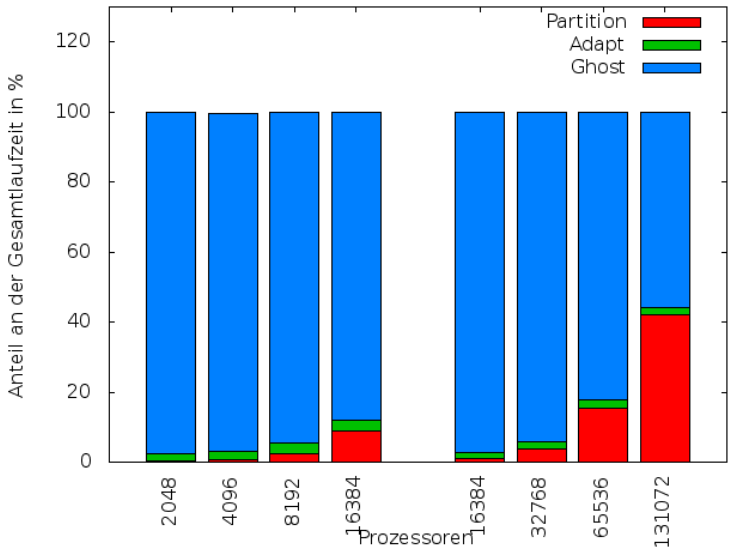

Abbildung 19: Der relative Anteil an der Gesamtlaufzeit der einzelnen High-Level Algorithmen zu den Testergebnissen in Abbildung 17.

In einem letzten Test wurden die Prismen im Zusammenspiel mit anderen Elementen getestet. Dazu wurde ein Gitter in Form eines Tors, bestehend aus zwei Tetraedern, einem Würfel und zwei Prismen, getestet. Das initiale Level ist drei und es wurde wieder anhand einer gedachten Wand um ein Level verfeinert, ebenso wie im vorherigen Test.

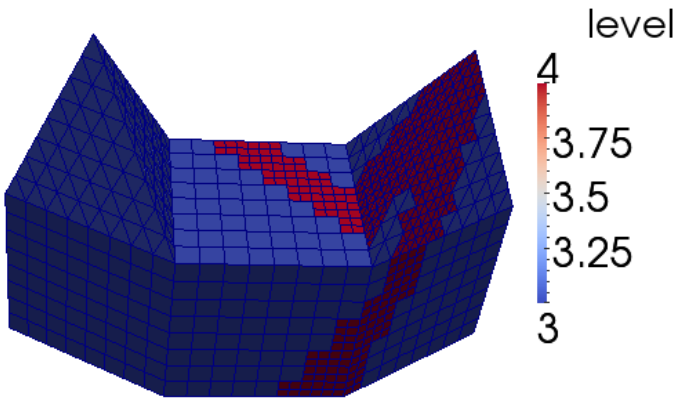

Abbildung 20: Ein hybrides Gitter mit initialem Level 3. Wieder wurde wie in Abb. 16 anhand einer schräg gestellten Wand um ein Level verfeinert. Somit gibt es Stellen, an denen Tetraeder, Würfel und Prismen gleichzeitig verfeinert und vergröbert werden müssen.

7 Zusammenfassung und Ausblick

Zusammenfassung
Ziel der Arbeit war es, das Prinzip des TM-Index auf Prismen zu übertragen und so eine Möglichkeit zur adaptiven Verfeinerung von Prismen zu schaffen. Dazu wurde zunächst ein Index für Linien ausgearbeitet um anschließend ein Prisma als Kreuzprodukt zwischen einem Dreieck und einer Linie auffassen zu können. Hierbei wurde herausgefunden, dass sich die Konzepte der RFK für Dreiecke und Linien auf Prismen übertragen lassen und sich diese somit unter Benutzung bereits bestehender Werkzeuge adaptiv verfeinern lassen. Die Verfeinerung basiert auf der Verfeinerungsregel von Bey und die Prismen lassen sich in zwei Typen einteilen. Damit lassen sich Prismen nur durch ihren Ankerknoten und ihren Typ speichern. Darüber hinaus wurden Methoden implementiert, die notwendig sind um Ghost Elemente für Prismen zu berechnen. Es wurde gezeigt, dass ein Nachfolger eines Prismas im Durchschnitt in konstanter Zeit berechnet wird. Zudem wurde die Skalierbarkeit der vorgestellten Algorithmen getestet und eine ideale starke Skalierung beobachtet. Im Vergleich zu der Implementierung von Tetraedern erzielen die Algorithmen ähnlich gute Laufzeiten. Zudem wurde festgestellt, dass sich mit Prismen auch in Kombination mit Tetraedern und Würfeln hybride Gitter erzeugen und manipulieren lassen. Die vorgestelle Methode besteht also die ersten Tests, einige weiterführende Fragestellungen sollten dennoch untersucht werden.

Ausblick
Die in dieser Arbeit implementierten Algorithmen bieten einen guten Ausgang um verschiedene Fragestellungen zu vertiefen. Zum einen wurde eine allgemeinere Version der für `Ghost` notwendigen Algorithmen nicht im Rahmen dieser Arbeit behandelt. Diese ist aber sicherlich sinnvoll, da somit die Einschränkung auf balancierte Gitter aufgehoben werden kann und auch in einem beliebig verfeinerten Gitter die Randelemente berechnet werden können.
Zudem ist ein Vergleich mit einer Implementierung, die die Kreuzproduktstruktur nicht ausnutzt und in den Low-level Algorithmen nicht auf schon bestehende Algorithmen für Dreiecke und Linien zurückgreift, interessant. Es stellt sich die Frage, ob so die Laufzeit noch weiter verbessert werden kann und ob solche Algorithmen genauso gut skalieren, wie die hier vorgestellten.
In dieser Arbeit wurde nur das Manipulieren von Gittern behandelt, weswegen es nun interessant ist Problemstellungen zu betrachten, die Gitter benutzen. Hierfür kann man einen Löser für eine PDG oder GDG

auf einem durch Prismen verfeinertes Gitter betrachten.
Ausführliche Tests der Prismen in hybriden Gitter und dazu gehörigen Lösern sollten zudem auch untersucht werden, da die hybriden Gitter Motivation zur Erarbeitung der Prismen sind.

8 Anhang

8.1 Alle notwendigen Low-Level Algorithmen

Die in Kapitel 4 und 5 vorgestellten ALgorithmen sind nicht ausreichend, um alle Funktionen des `t8code` auszuführen. Jedoch sind sie die wichtigsten und interessantesten. Im folgenden werden alle Low-Level-Algorithmen für Prismen sehr kurz erklärt um einen vollständigen Überblick zu bieten.

- `t8_dprism_get_level` (Prism p): Berechne das Level eine Prismas.

- `t8_dprism_copy` (Prism p, Prism dest): Kopiere alle Werte von p nach dest.

- `t8_dprism_compare` (Prism p_1, Prism p_2): TTeste die Rangfolge der Prismen.

- `t8_dprism_init_linear_id` (Prism p, int level, uint64_t id): Initialisiere ein Prisma anhand seiner ID.

- `t8_dprism_linear_id` (Prism p, int level): Berechne die lineare ID eines Prismas.

- `t8_dprism_successor` (Prism p, Prism succ, int level): Berechne den Nachfolger eines Prismas.

- `t8_dprism_first_descendant` (Prism p, Prism desc, int level): Berechne den ersten Nachfolger desc von Level level eines Prismas p.

- `t8_dprism_last_descendant` (Prism p, Prism s, int level): Berechne den letzen Nachfolger s von Level level eines Prismas p.

- `t8_dprism_child_id` (Prisma p): Berechne die lokale ID eines Prismas.

- `t8_dprism_child` (Prism p, int childid, Prism child): Berechne das zur childid entsprechende Kind von p.

- `t8_dprism_parent` (Prism p, Prism parent): Berechne das Elternprisma.

- `t8_dprism_is_familypv` (Prism fam[8]): Überprüfe, ob die Prismen eine Famile bilden.

- `t8_dprism_childrenpv` (Prism p,int length, Prism c[8]): Berechne alle Kinder eines Prismas.

- `t8_dprism_boundary_face` (Prism p, int face, Element boundary): Konstruiere die Randfläche an einer gegebenen Oberfläche (Gegenstück zu extrude_face).

- `t8_dprism_is_root_boundary` (Prism p, int face): Berechne, ob p sich eine Fläche mit seinem Baum teilt.

- `t8_dprism_is_inside_root` (Prism p): Prüfe, ob ein Prisma innerhalb der Wurzel liegt.

- `t8_dprism_num_face_children` (Prism p,int face): Berechne die Anzahl an Kindern, die an einer Randfläche liegen.

- `t8_dprism_face_neighbour` (Prism p, int face, Prisma neigh): Berechne den Nachbar zu einer bestimmten Fläche von p.

- `t8_dprism_children_at_face` (Prism p, int face, Prism children, int num_children): Berechne alle Kinder, die eine bestimmte Fläche berühren.

- `t8_dprism_face_child_face` (Prism elem, int face, int face_child): Für eine Fläche eines Prismas und eine lokale ID eines Kindes an dieser Fläche, gib die Flächennummer zurück, mit der das Kind die Fläche von p berührt.

- `t8_dprism_tree_face` (Prism p, int face): Wenn p die Baumgrenzen berührt, gib die Flächennummer des Baumes zurück.

- `t8_dprism_extrude_face` (Element face, Prism elem, int root_face): Baue zu einer gegebenen Oberfläche das entsprechende Prisma.

- `t8_dprism_vertex_coords` (Prisma p,int vertex, int coords[]): Berechne die Koordinaten einer Ecke von P.

8.2 Ergebnisse der Tests

Procs	Lev. 7	Fakt.	Lev. 9	Fakt.	Lev. 7	Fakt.	Lev. 9	Fakt.
32	18.730		-	-	12.467	-	-	-
64	9.366	1.999	-	-	6.235	1.999	-	-
128	4.683	2	-	-	3.120	1.998	-	-
256	2.343	1.998	-	-	1.561	1.998	-	-
512	1.173	1.997	-	-	0.783	1.993	-	-
1024	0.589	1.991	-	-	0.393	1.992	-	-
2048	0.295	1.996	18.733	-	0.199	1.974	12.471	-
4096	-	-	9.369	1.999	-	-	6.235	2.000
8192	-	-	4.685	1.999	-	-	3.119	1.999
16384	-	-	2.344	1.998	-	-	1.562	1.996
32768	-	-	1.175	1.994	-	-	0.783	1.994
65536	-	-	0.591	1.988	-	-	0.396	1.977
131072	-	-	0.302	1.956	-	-	0.204	1.941

Tabelle 3: Ergebnisse der Laufzeittests für `New` für Prismen in den Spalten 2 bis 5, für Tetraeder in den Spalten 6 bis 9. Die Zeiten sind in Sekunden angegeben. Der Faktor vergleicht hier ein Testergebnis mit der vorher gemessenen Laufzeit. Bei einer idealen starken Skalierung ist er 2.

Procs	Level 3-7	Faktor	Level 4-8	Faktor	Level 5-9	Faktor	
128	1.548	-	-	-	-	-	
256	0.776	1.994	-	-	-	-	
512	0.391	1.984	-	-	-	-	
1024	0.188	2.076	1.540	-	-	-	
2048	0.104	1.807	0.799	1.927	-	-	
4096	-	-	0.392	2.038	-	-	
8192	-	-	0.189	2.074	1.541	-	
16384	-	-	0.105	1.8	0.804	1.916	
32768	-	-	-	-	0.395	2.035	
65536	-	-	-	-	0.191	2.068	
131072	-	-	-	-	0.109	1.752	

Tabelle 4: Ergebnisse der Laufzeittests für `Adapt` für Prismen. Die Zeiten sind in Sekunden angegeben. Der Faktor vergleicht hier ein Testergebnis mit der vorher gemessenen Laufzeit. Bei einer idealen starken Skalierung ist er 2.

Procs	Level 3-7	Faktor	Level 4-8	Faktor	Level 5-9	Faktor	
128	1.203		-	-	-	-	
256	0.604	1.991	-	-	-	-	
512	0.299	2.020	-	-	-	-	
1024	0.149	2.006	1.182	-	-	-	
2048	0.084	1.773	0.614	1.925	-	-	
4096	-	-	0.299	2.053	-	-	
8192	-	-	0.150	1.993	1.181	-	
16384	-	-	0.084	1.785	0.616	1.917	
32768	-		-	-	0.300	2.053	
65536	-		-	-	0.152	1.973	
131072	-		-	-	0.089	1.707	

Tabelle 5: Ergebnisse der Laufzeittests für `Adapt` für Tetraeder. Die Zeiten sind in Sekunden angegeben. Der Faktor vergleicht hier ein Testergebnis mit der vorher gemessenen Laufzeit. Bei einer idealen starken Skalierung ist er 2.

Procs	Adapt	Partition	Ghost
2048	7.093	1.168	342.266
4096	3.74	1.193	152.814
8192	2.034	1.828	67.391
16384	1.141	3.378	32.804
16384	7.495	4.141	408.259
32768	3.919	7.133	174.419
65536	2.089	13.702	73.482
131072	1.22	27.156	35.919

Tabelle 6: Ergebnisse des Laufzeittests für `Adapt`, `Partition` und `Ghost` für ein Gitter aus 511 Prismen, die einen Zylinder approximieren, welcher an den Stellen verfeinert wird, an denen eine gedachte Wand den Zylinder schneidet. Zudem wird die Wand viermal verschoben. In der oberen Hälfte der Tabelle sieht man die Ergebnisse für eine anfängliche Verfeinerung von Level 5, in der unteren für eine anfängliche Verfeinerung von Level 6. Die Zeiten sind in Sekunden angegeben.

Literatur

[1] C. Burstedde, J. Holke, A Tetrahedral Space-Filling Curve for nonconforming adaptive Meshes, veröffentlicht im SIAM Journal on Scientific Computing 38 no. 5 (2016) Seiten 471-503

[2] C. Burstedde und J. Holke, Coarse Mesh Partitioning for tree-based AMR, akezptiert zur Veröffentlichung im SIAM Journal on Scientific Computing. ArXiv e-prints 2016, arXiv:1611.02929 [cs.DC]

[3] C. Burstedde, Lucas, C. Wilcox und Omar Ghatte, Scalable Algorithms for parallel adaptive Mesh refinement on forests of octrees, veröffentlicht im SIAM Journal on Scientific Computin 33 no. 3 (2011) Seiten 1103-1133

[4] D. Braess, Finite Elemente, Springer Verlag, 1997

[5] D. Hilbert, Über die stetige Abbildung einer Linie auf ein Flächenstück, Math. Annln., 38:459-460 (1891)

[6] E. H. Moore, On certain crinkly curves, Trans. Amer. Math. Soc., 1:72-90 (1900)

[7] George Karypis und Vipin Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering im Journal of Parallel and Distributed Computing, 48:71-95, 1998

[8] https://github.com/holke/t8code/releases/tag/v0.2

[9] https://github.com/Davknapp/t8code/releases/tag/v0.21

[10] Hans Sagan, Space-Filling Curves, Springer-Science+Business Media, 1994

[11] H. Sundar, R. S. Sampath und G. Biros, Bottom-up construction and 2:1 balance refinement of linear octrees in parallel, SIAM Journal on Scientific Computing, 30 (2008) Seiten 2675-2708

[12] H. Lebesque, Leçons sur L'Intégration et la Rechereche des Fonctions Primitives, Gauthier-Villars, Seiten 44-45 (1904)

[13] J. S. Centre, JUQUEEN: IBM Blue Gene/Q supercomputer im Jülich Supercomputing Centre, J. Large-Sacele Res. Facilities

[14] M. Bader, Space-Filling Curves - An Introduction with Applications in Scientific Computing, Springer Verlag, 2013

[15] M. Burkow und M. Griebel, A full three dimensional numerical simulation of the sediment transport and the scouring at a rectangular obstacle, Computer and Fluids, 125:1-10, 2016

[16] M. Griebel und G. Zumbusch, Hash based adaptive parallel multilevel methods with space-filling curves, NIC Symposium 2001 volume 9 der NIC Series (2002), Seiten 479-492

[17] R. S. Sampath und G. Biros A parallel geometric multigrid method for finite elements on octree meshes, SIAM Journal on Scientific Computing, 32 (2010), Seiten 1361-1392

[18] T.N. Bui und C.Jones, Finding good approximate vertex and edge partitions is NP-hard, Inform. Process. Lett., 42:153-159 (1992)

[19] Vorlesung "Wissenschaftliches Rechnen 1", gehalten von Prof. Dr. M. Rumpf, Universtiät Bonn, WS 2016/17 Das Vorlesungsskript ist zu finden unter http://numod.ins.uni-bonn.de/teaching/ws16/WissRechI/, letzter Zugriff 27.07.2017

[20] W. Sierpinski, Sur une nouvelle courbe continue qui remplit toute une aire plane, Bull. Acad. Sci. de Cracovie, Seiten 462-478 (1912)