Visualisierungsinstitut der Universität Stuttgart

Universität Stuttgart
Allmandring 19
D–70569 Stuttgart

Bachelorarbeit Nr. 36

Analyse poröser Medien auf Basis von Kristalliten

Alexander Straub

Studiengang:	Informatik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Dr. Guido Reina
	Dipl.-Inf. Daniel Kauker
Beginn am:	2012-12-03
Beendet am:	2013-06-04
CR-Nummer:	E.1, F.2.2, G.2.2, I.3.5

Kurzfassung

Eine Möglichkeit zur Simulation poröser Medien sind Modelle. Diese haben den Vorteil, dass sie computergestützt erstellt werden können und somit auch größere, aussagekräftigere Datensätze hervorbringen. In dieser Bachelor-Arbeit wird ein neuer Ansatz zur Bestimmung des Volumens und der Porösität vorgestellt, basierend auf einem Modell für Sandsteine. Dies war mit bisherigen Algorithmen nur unzureichend möglich, da durch Annäherung große Fehler entstehen.

Der neue hier vorgestellte Ansatz wird direkt auf der Geometrie des Modells ausgeführt, somit werden fast keine Rechenfehler begangen. Dieser Ansatz lässt sich zudem durch Aufteilen in Unterprobleme und durch die Benutzung von Heuristiken effizient auf große Datensätze anwenden. Die Messergebnisse und die Folgerungen für Effizienz und Effektivität des Algorithmus werden anschließend aufgeführt und erläutert.

Abstract

A possibility to simulate porous media are models. These models have the advantage of being computer-generated and thus, huge and more significant datasets can be produced. In this bachelor thesis, a new approach to calculate the volume and the porosity of such media is introduced, based on a model for sandstone. In this case, previous algorithmic approaches had the problem of producing too large errors using methods for approximation.

The newly introduced approach is directly executed on the geometry of the model. Thus, there is nearly no calculation error. Furthermore, this approach uses a divide-and-conquer strategy and additionaly uses heuristics to efficiently process large datasets. Measurement results and conclusions regarding efficiency and effectiveness of the algorithm are then given and discussed.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Die Erforschung poröser Medien ist ein aktuelles Thema in der Physik. Die Erkenntnisse die hier gewonnen werden können, beeinflussen wiederum andere Naturwissenschaften, wie die Biologie, Materialwissenschaften und Geowissenschaften. Durch diese Relevanz zur aktuellen Forschung ist es umso wichtiger Erkenntnisse und Informationen aus bereits vorhandenen Daten zu ziehen. Zu diesen Daten gehört auch ein computergestützt erstelltes Modell zur möglichst genauen Repräsentation poröser Medien.

Dieses Modell ist eine geometrische Darstellung poröser Medien, zum Beispiel von Sandsteinen. Es besteht aus vielen sich überschneidenden Polyedern. Das Verfahren zur Bestimmung des Stein-Volumens, welches Thema dieser Bachelor-Arbeit ist, ist somit ein geometrisches Problem. Dieses Problem soll hier mit Hilfe von BSP-Bäumen algorithmisch gelöst werden. Diese Baumstrukturen werden bereits vielfältig in den Bereichen der Computergrafik verwendet und sind dadurch schon großteils erforscht. Ein Algorithmus zum Erstellen räumlich getrennter Polyeder, wie er für diesen Zweck benötigt wird, gab es bisher jedoch noch nicht.

Somit verbindet dieses Thema die Interessen und Forschungsthemen von Physik und Informatik und ist für beide Bereiche relevant.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Related Works
: Gibt einen Überblick über relevante wissenschaftliche Arbeiten zum Thema der *Porösen Medien*

Problemstellung
: Beschreibt das benutzte Sandstein-Modell und die Idee für einen Ansatz zum Lösen des Problems

Datenstrukturen
: Beschreibt die Datenstrukturen, die in dieser Arbeit zum Einsatz kommen

Erstellen räumlich disjunkter Polyeder
: Führt die verwendeten Algorithmen ein, um aus je zwei konvexen Polyedern räumlich disjunkte konvexe Polyeder zu erstellen

Anwendung auf große Datensätze
Stellt Strategien und Heuristiken zur Lösung auf großen Datensätzen vor

Messwerte und Ergebnisse
Präsentiert Messergebnisse zur Effizienz und Effektivität der Algorithmen und der verwendeten Heuristiken

Zusammenfassung und Ausblick
Fasst die Ergebnisse der Arbeit zusammen und gibt entsprechend einen Ausblick

2 Related Works

In diesem Kapitel werden Informationen gegeben, um das Thema dieser Bachelorarbeit in ein konkretes Andwendungsgebiet einzuordnen, jedoch nicht, um es auf dieses zu beschränken. Dabei gelten die poröse Medien und deren algorithmische Erforschung als Motivation und Anlass für diese Arbeit.

2.1 Poröse Medien

Poröse Medien spielen eine große Rolle in der Physik und werden gerade im Gebiet der computergestützten Berechnungen genutzt, um mithilfe von Modellen Eigenschaften zu erforschen. Zu diesen Eigenschaften zählen unter anderem auch die Durchlässigkeit von Flüssigkeiten und Gasen und deren Einschlussvolumen. Um dabei gute und aussagekräftige Ergebnisse zu erhalten bedarf es ausreichend großer Datensätze. Dies ist jedoch problematisch, da für das Erstellen dieser Datensätze mit Röntgenaufnahmen und Computertomographie zum heutigen Stand der Technik Jahre benötigen würden.

An dieser Stelle kommt ein Modell ins Spiel, welches durch seine Einfachheit die Erforschung mithilfe von Algorithmen zulässt. Dieses Modell wurde erstmals von Latief et al. [LBFH] für Fontainebleau Sandsteine eingeführt. Auf dieses Modell und Teile des Datensatzes, von der Webseite des *Institute for Computational Physics* der Universität Stuttgart [HZL], wird sich diese Bachelor-Arbeit beziehen. In Abbildung 2.1 wird dieses Modell visualisiert dargestellt.

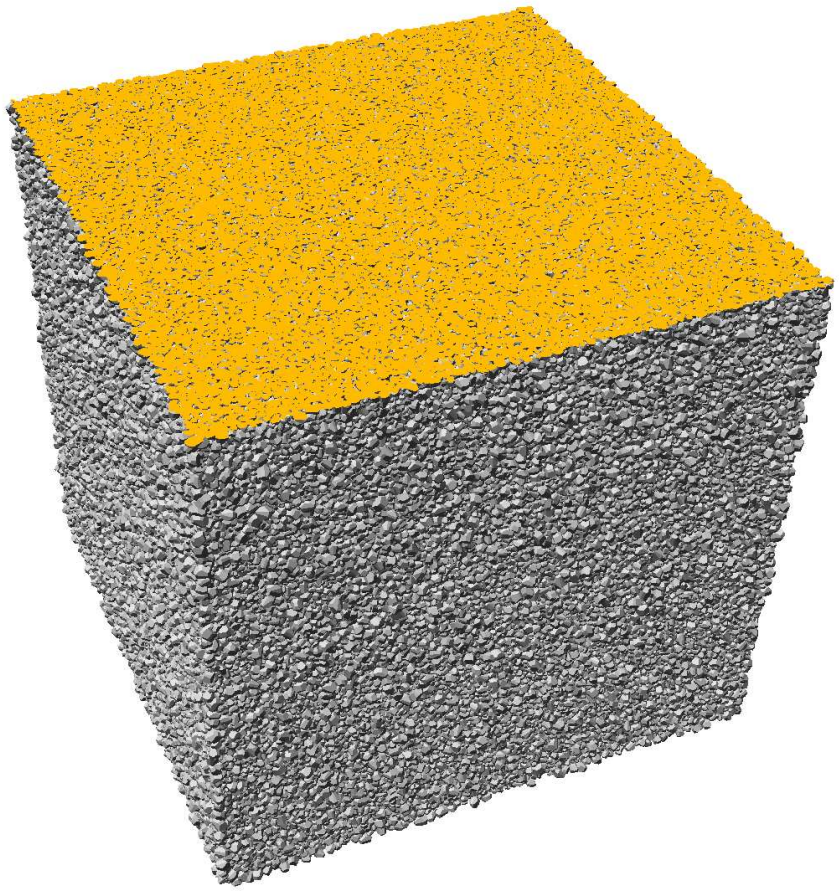

Abbildung 2.1: Modell des Fontainebleau Sandsteins [LBFH][HZL]

3 Problemstellung

Die Aufgabe ist es, das Volumen von porösen Medien zu berechnen. Für diese Medien existiert bereits ein Modell, jedoch noch kein effektiver Algorithmus, um das Volumen zu bestimmen. In den folgenden Kapiteln wird darum nach der Vorstellung dieses Modells die Idee für einen neuen Algorithmus vorgestellt.

3.1 Modell

Das Modell für poröse Medien, eingeführt von Latief et al. [LBFH], besteht aus einer Vielzahl an Kristallen. Diese Kristalle sind konvexe Polyeder und können auf mehrere Kristalldefinitionen zurückgeführt werden. Somit gibt es im Modell nur k Kristalltypen, welche durch Instanziierung gedreht, skaliert und verschoben werden. Dadurch erhält man n Kristalle mit $k << n$. In dieser Bachelorarbeit werden dabei alle Versuche mit Beispielmodellen gerechnet. In diesen gibt es $k = 100$ Kristalldefinitionen und im ersten Datensatz $n = 10.000$ Kristallinstanzen, im zweiten $n = 25.000$ Kristallinstanzen.

Das Problem des Volumenbestimmens ist nun jedoch, dass die Kristalle - im Folgenden als konvexe Polyeder bezeichnet - sich gegenseitig überschneiden können. Daraus folgt, dass das Gesamtvolumen nicht durch Aufsummieren der Volumen der Polyeder bestimmt werden kann:

$$V_{ges} \neq \sum_{i=1}^{n} V(K_i) \tag{3.1}$$

Dabei sind K_i die im Raum verteilten Kristalle, mit $i \in \{1, ..., n\}$.

In Abbildung 3.1 wird das Modell visualisiert dargestellt. Dabei kann man gut erkennen, dass sich die Polyeder meist mehrfach überschneiden.

3.2 Idee für neuen Ansatz

Gesucht ist ein Algorithmus, der die ursprünglichen Kristalle so aufteilt, dass diese räumlich disjunkt sind:

$$f : \{K_1, ..., K_n\} \rightarrow \{P_1, ..., P_m\} \tag{3.2}$$

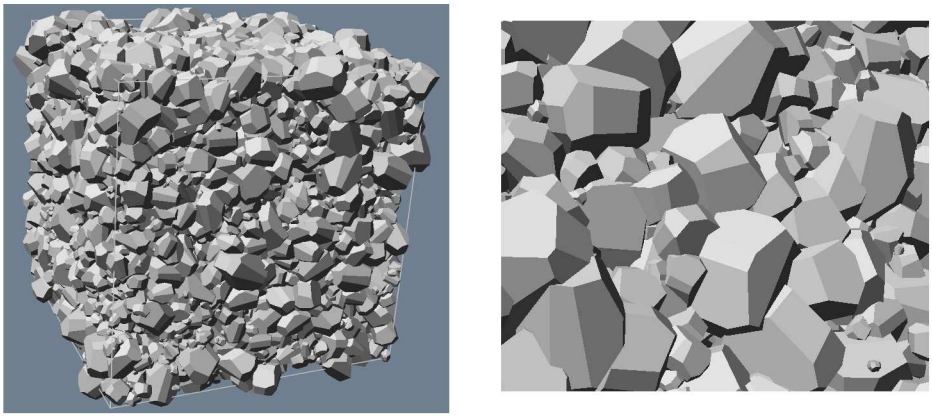

Abbildung 3.1: Modell mit 10.000 Kristallen aus 100 Kristalldefinitionen

wobei folgendes gilt

$$\{K_1, ..., K_n\} \cap \{P_1, ..., P_m\} = \{K_1, ..., K_n\} \qquad P_i \cap P_j = \emptyset, \forall i, j \in \{1, ..., m\}, i \neq j \tag{3.3}$$

Dabei bezieht sich der Schnitt und die leere Menge auf die räumliche Überschneidung, nicht auf Mengen.

Daraus folgt, dass

$$V_{ges} = \sum_{i=1}^{m} V(P_i) \tag{3.4}$$

Anschließend kann dann das Volumen berechnet werden, indem die Polyeder tetraedisiert werden. Die Berechnung des Volumen eines Tetraeders ist nun die folgende

$$V_{Tetraeder} = \frac{1}{3} Gh \tag{3.5}$$

mit Grundfläche G und Höhe h.

4 Datenstrukturen

Für die Verwendung der Algorithmen, die in den Kapiteln 5 und 6 eingeführt werden, bedarf es einiger Datenstrukturen. Diese werden hauptsächlich für die geometrische Repräsentation benötigt und werden nun vorgestellt.

4.1 Repräsentation der Geometrie

Für die einfache Darstellung konvexer Polygone und Polyeder werden Geraden und Ebenen benötigt. Diese müssen jedoch über ihre Normalen definiert werden, damit die in Abschnitt 4.2.2 auf Seite 17 eingeführte Datenstruktur der BSP-Bäume verwendet werden kann.

Daraus ergeben sich folgende Definitionen für

- 2D

  (4.1) Gerade $g = (\vec{p}, \vec{n})$ mit $\vec{p}$ beliebiger Punkt auf g und $\vec{n}$ Normale

  Das Polygon wird somit definiert durch eine Menge von Geraden, wobei Außen- und Innenseite durch die Richtung der Normalen festgelegt ist.

- 3D

  (4.2) Ebene $E = (\vec{p}, \vec{n})$ mit $\vec{p}$ beliebiger Punkt auf E und $\vec{n}$ Normale

  Das Polyeder wird definiert durch eine Menge von Ebenen, wobei wie im 2D-Fall die Außen- und Innenseite durch die Richtung der Normalen festgelegt wird.

4.2 Graphen

An mehreren Stellen in den Algorithmen wird auf Graphen zurückgegriffen. Dabei spielt gerade der BSP-Baum eine zentrale Rolle. Dieser soll hier, zusammen mit allgemeinen Graphen und Bäumen, eingeführt und erläutert werden.

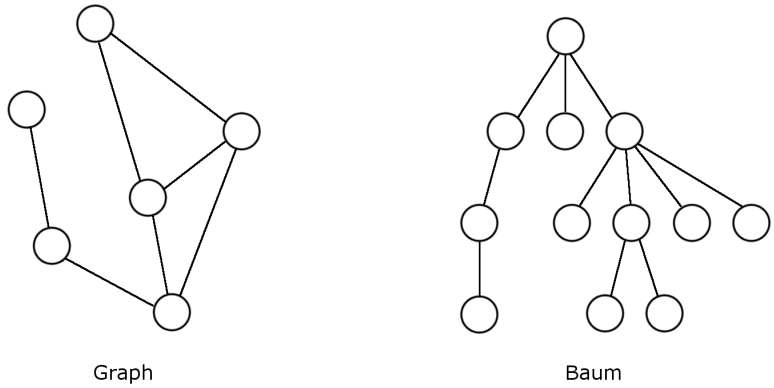

Abbildung 4.1: Beispiele für je einen einfachen Graphen und einen Baum

4.2.1 Allgemeine Graphen

Ein Graph G wird als Tupel seiner Knoten und Kanten definiert:

(4.3) $G = (V, E)$, mit V Menge der Knoten und E Menge der Kanten

Dabei repräsentieren die Knoten meist Objekte. Die Kanten, die jeweils zwei Knoten miteinander verbinden, repräsentieren die Relationen zwischen den Knoten. Diese Kanten können gerichtet oder ungerichtet sein. In dieser Bachelorarbeit werden jedoch nur ungerichtete Graphen benötigt, da die Relation zwischen zwei Knoten hier immer kommutativ ist.

4.2.2 Bäume

Ein Baum ist ein spezieller Graph, mit den Eigenschaften, dass dieser eine Wurzel hat und keine Zyklen. Bei Bäumen werden die Knoten unterteilt in Wurzel, innere Knoten und Blätter. Die Wurzel ist dabei das einzige Element, welches keinen Vaterknoten hat. Blätter hingegen sind die Knoten ohne Kinder. Alle anderen Knoten werden als innere Knoten bezeichnet, da diese sowohl einen Vaterknoten als auch Kinder haben. In Abbildung 4.1 wird zur Veranschaulichung je ein einfaches Beispiel für einen Graphen und einen Baum dargestellt.

Quad- und Octree

Quad- und Octrees werden meist zur räumlichen Aufteilung verwendet, um den Raum in gleich große Teile aufzuspalten, wie in Abbildung 4.2 gezeigt wird.

Hier kann man zwischen 2D und 3D unterscheiden:

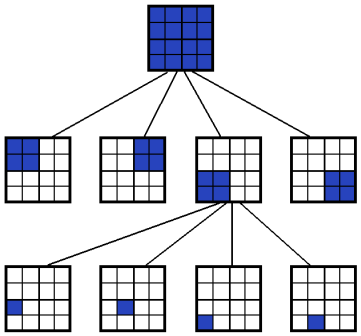

Abbildung 4.2: Raumaufteilung mit Hilfe eines Quadtree

- 2D
  Ein Baum mit maximal 4 Kindern pro Knoten; teilt einen endlichen zwei-dimensionalen Raum in vier Unterräume auf.
- 3D
  Ein Baum mit maximal 8 Kindern pro Knoten; teilt einen endlichen drei-dimensionalen Raum in acht Unterräume auf.

BSP-Bäume

Bei BSP-Bäumen handelt es sich um Binärbäume. Das heißt, dass ein Knoten maximal zwei Kinder hat. Im Gegensatz zu Quad- und Octrees wird der BSP-Baum jedoch nicht verwendet um den Raum in gleich große, am Raster ausgerichtete Unterräume aufzuteilen. Er wird dazu verwendet, um durch einen Partitionierer den Raum in zwei Halbräume zu unterteilen. Zudem wird hier nicht vorausgesetzt, dass der Raum begrenzt ist. Somit sind zwei Halbräume erstmal nur in eine Richtung begrenzt. Erst durch viele Unterteilungen und durch Unterscheidung zwischen innen und außen können geometrische Figuren entstehen.

Wiederum kann je nach Dimension unterschieden werden:

- 2D
  Aufteilung der Fläche durch Geraden
- 3D
  Aufteilung des Raums durch Ebenen

Um an dieser Stelle zwischen Innen- und Außenseite unterscheiden zu können, kommt die Geraden- bzw. Ebenendefinition aus Abschnitt 4.1 ins Spiel. Außerdem wird jetzt auch die

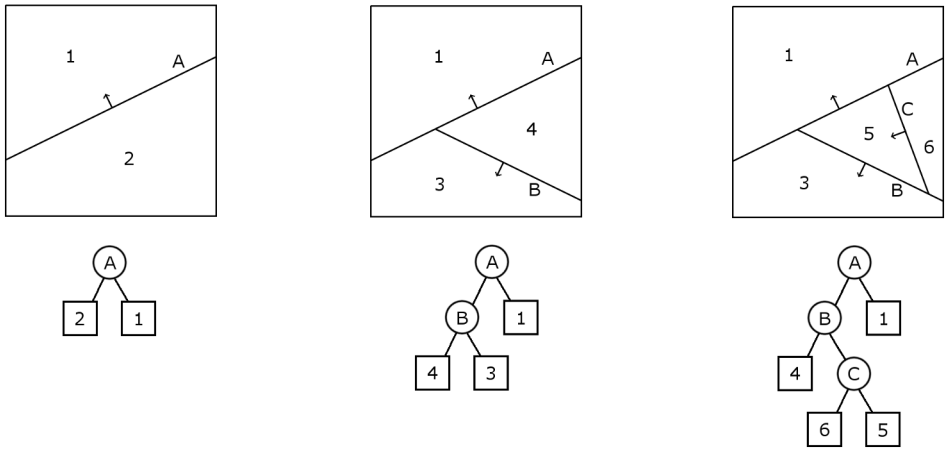

Abbildung 4.3: Einfaches Beispiel für einen BSP-Baum

Reihenfolge der Kanten entscheidend: Zusätzlich zu der Relation zwischen den Knoten muss zwischen linker und rechter Kante unterschieden werden, um eine Aussage über Innen- und Außenseite treffen zu können.

In dieser Bachelorarbeit wird dazu die Konvention eingeführt, dass die linke Kante die Innenseite repräsentiert und die rechte Kante entsprechend die Außenseite. Weiterhin wird immer angenommen, dass die Normalen der Geraden oder Ebenen nach außen zeigen.

In Abbildung 4.3 wird eine einfache Aufteilung des Raums im Zwei-Dimensionalen vorgenommen. Dabei gibt es keinen Unterschied zur Aufteilung im 3D-Raum, außer dass andere Partitionierer verwendet werden.
Im ersten Schritt wird durch die Gerade A der Raum in zwei Teile partitioniert. Der Halbraum mit der Bezeichnung 1 ist dabei außen, der Halbraum mit der Bezeichnung 2 ist innen. Dies folgt aus der oben eingeführten Konvention für die Normalenrichtung und der Kantenreihenfolge. Die nächsten Schritte laufen analog ab, bis alle Partitionierer abgearbeitet wurden.

Abbildung 4.4 zeigt hingegen eine Suche auf einem BSP-Baum. Dabei ist die Eingabe der blaue Punkt. Nun soll ermittelt werden, in welcher Fläche dieser Punkt liegt. Dazu wird bei der Wurzel beginnend abgestiegen und in jedem Schritt überprüft, ob der Punkt auf der Innen- oder auf der Außenseite des Partitionierers liegt. Dieser Schritt wird solange wiederholt bis ein Blatt erreicht wurde.
Ein Sonderfall wäre es jedoch, wenn der Punkt auf einem Partitionierer liegen würde. In diesem Fall wäre es nicht offensichtlich klar, welcher Seite der Punkt zugeordnet werden sollte.

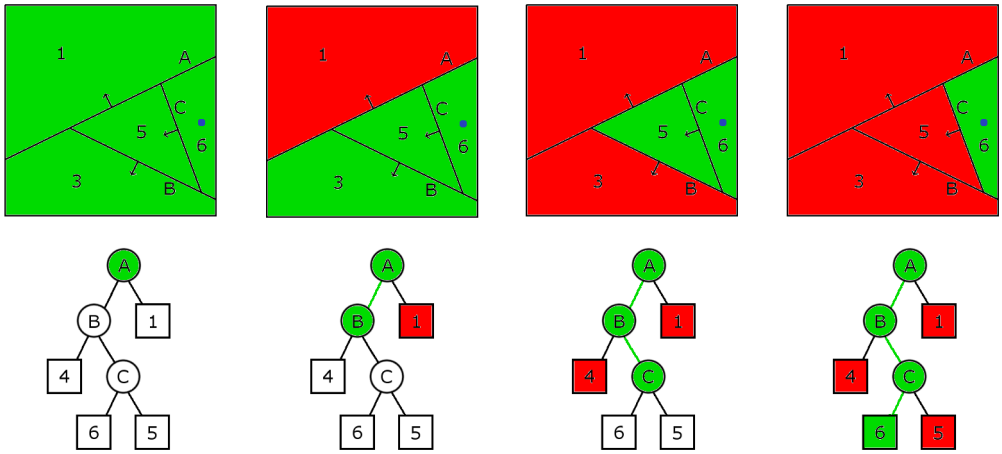

Abbildung 4.4: Suche auf einem BSP-Baum

Anwendungsgebiete BSP-Bäume werden häufig im Kontext von Computergrafik benutzt. Der Baum enthält dabei Informationen über den Raum, wie es zum Beispiel für Kollisionsabfragen bei Graphic Engines notwendig ist. Weiter können BSP-Bäume allerdings auch für boolsche Operationen zwischen zwei Objekten verwendet werden, wie es Naylor et al. [NAT] beschreiben.

5 Erstellen räumlich disjunkter Polyeder

Für die Berechnung des Gesamtvolumens müssen alle Polyeder räumlich disjunkt zueinander sein. Außerdem soll die gute Eigenschaft der Ausgangspolyeder beibehalten werden, nämlich dass diese konvex sind. Dass diese Eigenschaft wichtig ist für den Algorithmus wird mit dessen Einführung in Abschnitt 5.2.2 auf Seite 25 verdeutlicht. Im folgenden werden nun die notwendigen Algorithmen vorgestellt.

5.1 Problembeschreibung

Es soll ein Algorithmus gefunden werden, der zwei sich überschneidende konvexe Polyeder in räumlich disjunkte aufteilt. Da Überschneidungen von mehr als zwei Polyedern möglich sind, soll eine weitere Voraussetzung sein, dass die entstandenen Polyeder wiederum konvex sind. Abbildung 5.1 zeigt ein mögliches Vorgehen.

Dass das Aufteilen der Polyeder nicht trivial ist, verdeutlicht Abbildung 5.2. Der zweite, komplexere Fall zeigt hier deutlich, dass durch naives Aufteilen mehr Erzeugnisse entstehen können als unbedingt notwendig.

Aus diesem Grund, werden in den nächsten Abschnitten Algorithmen vorgestellt, durch die sich diese Aufgabe auf BSP-Bäumen lösen lässt.

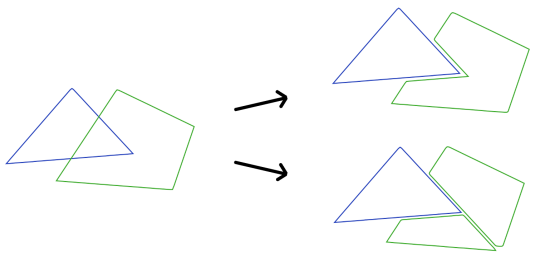

Abbildung 5.1: Ziel disjunkte Polyeder zu erhalten und wenn möglich konvexe

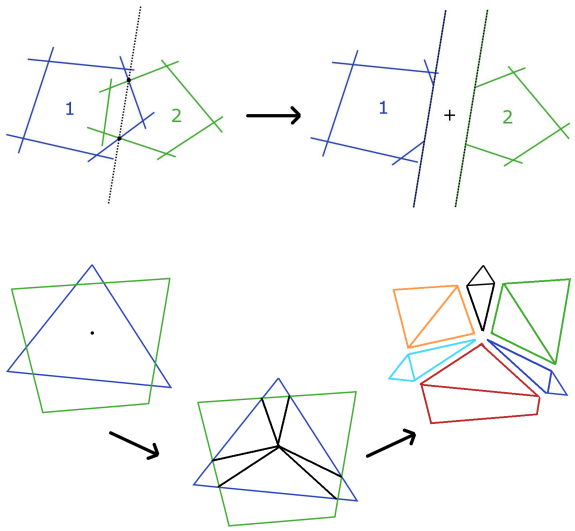

Abbildung 5.2: Einfacher Fall mit zwei Schnittpunkten und komplexer Fall mit vielen

5.2 Algorithmen auf BSP-Bäumen

Für BSP-Bäume gibt es viele Algorithmen. Die wichtigsten sind dabei die zum Ermitteln des Schnittbereichs zweier Objekte und des gemeinsamen Raums. Letzteres wird von [NAT] beschrieben und im folgenden Abschnitt eingeführt und erläutert.

5.2.1 Zusammenführung (Merge)

Ziel ist es zwei beliebige Polyeder zusammenzufügen, die jeweils durch einem BSP-Baum repräsentiert werden. Das neu entstandene Polyeder deckt also den Raum ab, der von mindestens einem der Ausgangsobjekte abgedeckt wird. Der Baum des zusammengesetzten Polyeders wird deshalb aus den beiden Bäumen der Ausgangspolyeder so erstellt, dass dieser für jeden Punkt, der sich in mindestens einem der beiden Polyeder befindet *IN* ausgibt, ansonsten *OUT*.

Nachfolgend wird der Algorithmus zum Zusammenführen zweier BSP-Bäume und damit zum Berechnen des gemeinsamen Raums als Pseudocode eingeführt. Siehe Algorithmus 5.1 auf der nächsten Seite. Dieser ist aus der Arbeit von Naylor, Amanatides und Thibault [NAT] entnommen. Der Algorithmus funktioniert dabei für beliebige Polyeder, insbesondere für konvexe. Zusätzlich zum Pseudocode wird ein Durchlauf des Algorithmus als Beispiel in Abbildung 5.3 gezeigt.

Algorithmus 5.1 Merge Algorithmus

```
Merge_Bspts : ( T1, T2 : Bspt ) -> Bspt
  Types
    PartitionedBspt : ( inNegHs, inPosHs : Bspt )

  Imports
    Merge_Tree_With_Cell : ( T1, T2 : Bspt ) -> Bspt
    Partition_Bspt : ( T : Bspt, P : Bp ) -> PartitionedBspt

  Definition
    IF T1.is_a_cell OR T2.is_a_cell
      VAL := Merge_Tree_With_Cell( T1, T2 )
    ELSE
      Partition_Bspt( T2, T1.root_region.bp ) -> T2_partitioned

      VAL.neg_subtree := Merge_Bspts( T1.neg_subtree, T2_partitioned.inNegHs )
      VAL.pos_subtree := Merge_Bspts( T1.pos_subtree, T2_partitioned.inPosHs )
      VAL.root_region := T1.root_region
    END IF

    RETURN VAL
END Merge_Bspts
```

Beschreibung des Algorithmus Für den Algorithmus werden anfangs zwei zusätzliche Funktionen *Merge_Tree_With_Cell* und *Partition_Bspt* definiert. Auf deren Funktionsweise wird später genauer eingegangen. Für den Algorithmus im Allgemeinen ist hier erstmal nur wichtig, dass *Merge_Tree_With_Cell* eine benutzerdefinierte Handhabung beim Erreichen des Abbruchkriteriums zulässt und *Partition_Bspt* anhand eines binären Partitionierers, im 2D-Fall einer Geraden und im 3D-Fall einer Ebenen, den als weiteren Parameter übergebenen Baum in linken und rechten Unterbaum aufteilt.

Der Algorithmus geht rekursiv vor. Als Abbruchkriterium nimmt er die Situation, wenn von einem der beiden Bäume ein Blatt erreicht wurde. Ansonsten wird der als zweiter Parameter übergebene Baum am Partitionierer des Wurzelknotens des ersten Baums aufgeteilt. Das Resultat dieses Vorgangs wird nun als Parameter für den rekursiven Aufruf verwendet. Dabei wird nun der linke Unterbaum des partitionierenden Baums mit dem linken Unterbaum des partitionierten Baums zusammengeführt, wie auch analog für die rechte Seite. Als Wert für den aktuellen Knoten wird der verwendete Partitionierer selbst eingesetzt.

Naylor et al. [NAT] schlagen zusätzlich vor, dass durch eine zusätzliche Funktion in jedem Schritt ermittelt werden kann, welcher der beiden Bäume partitioniert wird oder den Partitionierer stellt. In diesem Fall stellt immer der gleiche Baum den Partitionierer, um nachfolgend den Algorithmus zum Aufteilen des Baums anwenden zu können, welcher im Abschnitt 5.2.2 auf Seite 25 eingeführt wird.

Merge_Tree_With_Cell Durch Ausgliedern dieser Methode stellen Naylor et al. [NAT] die Möglichkeit zur Verfügung einen benutzerdefinierten Operator zu verwenden. In unserem

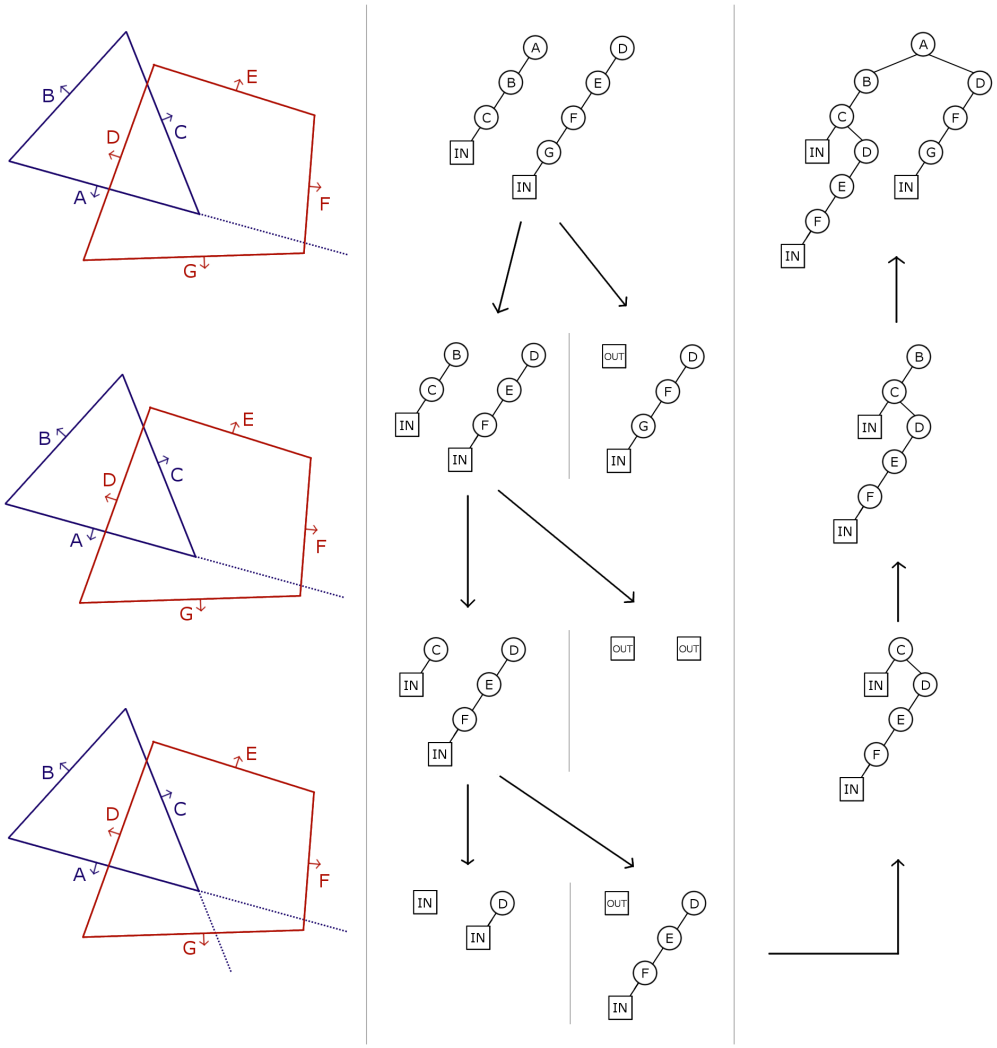

Abbildung 5.3: Durchlauf des Merge-Algorithmus für ein Beispiel

Fall ist dies jedoch nicht notwendig, da der Baum nur zur geometrischen Repräsentation verwendet wird, nicht zur logischen.

Partition_Bspt Diese Funktion liefert nach Übergabe eines BSP-Baums und eines Partitionierers einen rechten und einen linken Unterbaum zurück. Diese neuen Bäume werden durch Partitionieren des übergebenen BSP-Baums erstellt. Die Implementierung hängt dabei von den benutzten Datenstrukturen zur Repräsentation des Polyeders ab. Wenn beispielsweise der Partitionierer als Ebene in Punkt-Normalen-Form vorliegt und zu jeder Ebene im Baum die Eckpunkte bekannt sind, kann durch die Normale und die Position der Eckpunkte festgestellt werden, ob eine Fläche komplett innen, außen oder auf beiden Seiten liegt.

Algorithmus 5.2 Split Algorithmus

```
Split_Bspts : ( T : Bspt ) -> Bspt[]
  out_val : Bspt[]
  right_step_taken : bool := false

  DO
    right_step_taken := false
    currentNode : BsptNode := T.root
    out_val.Add( T.root )

    WHILE currentNode IS NO LEAF
      IF currentNode.hasPosSubtree AND currentNode.pos_subtree.notYetVisited
        out_val.last.InvertNormal()
        out_val.last.AppendNegSubtree( currentNode.pos_subtree.node )

        currentNode = currentNode.pos_subtree.node
        right_step_taken := true
      ELSE
        out_val.last.AppendNegSubtree( currentNode.neg_subtree.node )

        currentNode = currentNode.neg_subtree.node
      END IF
    END WHILE
  WHILE right_step_taken

  RETURN out_val
END Split_Bspts
```

5.2.2 Aufteilen (Split)

Die Funktion dieses Algorithmus ist es, einen zuvor zusammengefügten Polyeder in neue, konvexe Polyeder aufzuteilen. Der Algorithmus funktioniert dabei jedoch nur auf BSP-Bäumen, welche durch oben beschriebene Methode zusammengefügt wurden. Nur durch diese besondere Form des Baums, können daraus schnell konvexe Polyeder erstellt werden. Da wir wissen, dass in unserem Modell nur konvexe Polyeder vorhanden sind und durch diesen Schritt wiederum konvexe Polyeder entstehen, kann dieser Algorithmus hier angewendet werden.

Der Algorithmus wird im Folgenden in Algorithmus 5.2 als Pseudocode beschrieben und als Beispiel in Abbildung 5.4 veranschaulicht.

Beschreibung des Algorithmus Als Eingabe bekommt dieser Algorithmus einen BSP-Baum und gibt nach dem Aufteilen mehrere BSP-Bäume zurück. Das Aufteilen selber wird in einer Schleife vorgenommen. Diese wird so oft durchlaufen, wie neue BSP-Bäume und somit Polyeder entstehen. Am Anfang dieser Schleife wird die Abbruch-Variable *right_step_taken* wieder auf $false$ gesetzt. So wird nach jedem Durchlauf überprüft, ob es noch einen weiteren Baum gibt. Weiterhin wird der Liste der neuen BSP-Bäume ein neues Element hinzugefügt,

das bis dahin nur aus der Wurzel besteht, nämlich der gleichen Wurzel wie der des Eingabe-Baums.

In der inneren Schleife wird nun der Baum traversiert bis ein Blatt erreicht wurde. In der Schleife werden dabei folgende Unterscheidungen getroffen: der aktuelle Knoten hat ein noch nicht besuchtes rechtes Kind, oder es gibt keine Abzweigung oder das rechte Kind wurde bereits besucht. Im ersten Fall wird der letzte Knoten des neu erstellten Baums invertiert und als dessen linkes Kind das rechte Kind des aktuell traversierten Knotens angehängt. Zudem wird der Baum als nächstes an diesem rechten Knoten weiter traversiert und das Abbruchkriterium auf $false$ gesetzt (dazu wird die Variable *right_step_taken* auf *true* gesetzt). Zu diesem Zeitpunkt ist sicher, dass noch ein weiterer Baum erstellt werden kann, da erst im letzten Durchlauf ohne rechts ab zu zweigen der letzte mögliche Baum erstellt wird und somit das Abbruchkriterium *true* bleibt (*right_step_taken* bleibt $false$). In den anderen Fällen wird der linke Knoten des traversierten Baums dem neuen Baum hinzugefügt und dieser auch als nächstes Element traversiert.

Korrektheit Um zu zeigen, dass der Algorithmus korrekt ist, sind folgende drei Eigenschaften zu zeigen

1. die erstellten Polyeder sind räumlich disjunkt zueinander,
2. der gleiche Raum wie vom zusammengesetzten Polyeder wird abgedeckt und
3. die erstellten Polyeder sind konvex.

Die Eigenschaft der räumlichen Disjunktheit folgt aus der Definition des BSP-Baums. Zwei erstellte Polyeder A, B müssen demnach disjunkt sein, da sie einen gemeinsamen Knoten X haben dessen Kind Y zu A gehört und Kind Z zu B. Da diese zwei Unterräume darstellen, die durch X getrennt sind, sind diese beiden Räume disjunkt und somit auch die Polyeder.

Die vollständige Abdeckung folgt direkt aus der Korrektheit des Merge-Algorithmus, da der ganze Raum abgedeckt wird und aus dem BSP-Baum, da alle Blätter mit der Eigenschaft *IN* auch in den neuen BSP-Bäumen existieren und den selben Unterraum abdecken.

Wird ein konvexer Polyeder durch einen Partitionierer geteilt, sind die beiden resultierenden Polyeder wiederum konvex. Da der nächste Partitionierer dann nur noch auf diese Subprobleme angewendet wird, wird die konvexe Eigenschaft weitervererbt. Dadurch sind alle erstellten Polyeder konvex.

5.3 Veranschaulichung in 3D

In Abbildung 5.5 wird ein einfaches Beispiel für die Ausführung der beiden Algorithmen dargestellt. Dabei werden aus zwei sich überschneidenden Würfeln vier Polyeder erstellt. Einer dieser erstellten Polyedern entspricht dabei einem der Ausgangspolyeder. Alle erstellten

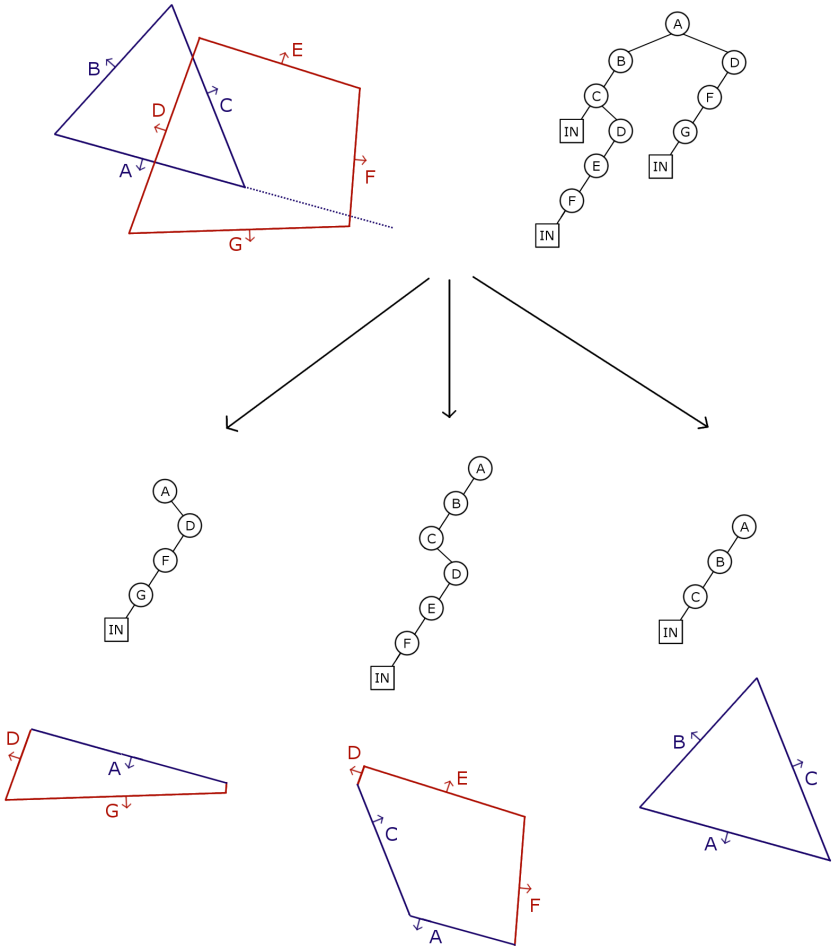

Abbildung 5.4: Durchlauf des Split-Algorithmus für ein Beispiel

Polyeder sind disjunkt zueinander und decken den ganzen auch zuvor abgedeckten Raum ab.

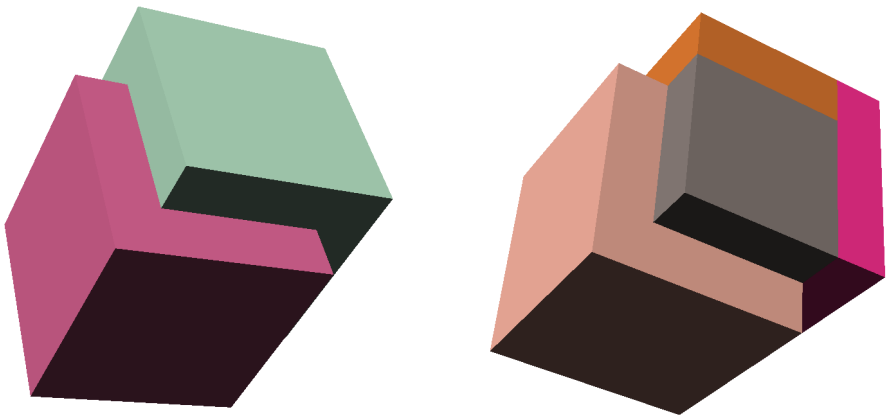

Abbildung 5.5: Einfaches Beispiel in 3D

6 Anwendung auf große Datensätze

Der Algorithmus in der Form wie er im vorherigen Kapitel eingeführt wurde, funktioniert angewendet auf zwei konvexe Polyeder. Für mehrere, sich mehrfach überschneidende Polyeder jedoch wird eine komplexere Vorgehensweise benötigt. Da in dem benutzten Testdatensatz 10.000 Polyeder verwendet werden bedarf es zusätzlich einer Strategie, die es erlaubt die Aufgaben parallel abzuarbeiten. Dies ist umso wichtiger, da das vollständige Modell, zu finden auf der Instituts-Homepage von Hilfer et al. [HZL], über 1 Million Polyeder enthält.

Um diese große Anzahl von Polyedern zu bewältigen, wird erst die Aufteilung in Gruppen beschrieben, wodurch Parallelisierbarkeit gewährleistet wird. Anschließend werden Strategien zur lokalen Bearbeitung dieser Gruppen gegeben, sowie Heuristiken um diese Bearbeitung weiter zu optimieren.

6.1 Aufteilung in Gruppen

Eine Aufteilung in geographische Gruppen ist aus mehreren Gründen ein sinnvolles Vorgehen. Einerseits wird dadurch ermöglicht diese Gruppen parallel abzuarbeiten. Andererseits müssten sonst alle Polyeder paarweise auf Überschneidungen überprüft werden. Das würde somit in quadratischer Laufzeit enden und auf ein Modell mit mehreren hunderttausend Polyedern nicht mehr anwendbar sein.

6.1.1 Schnitt am Gitternetz

Ein großer Vorteil des Modells ist es, dass die Polyeder in beliebig kleine Polyeder aufgeteilt werden dürfen, ohne dass dies die Eigenschaften und die Aussagekräftigkeit des Modells in Bezug auf das Volumen verletzt. Einzig zu beachten ist, dass durch alle Polyeder der gleiche Raum wie zuvor abgedeckt werden muss. Deshalb ist es möglich alle Polyeder an einem Gitter zu schneiden, um diese anschließend eindeutig den lokalen Gruppen zuweisen zu können. In Abbildung 6.1 wird dies verdeutlicht. Es ist gut zu sehen, dass die entstandenen acht Gruppen zueinander räumlich unabhängig sind.

Durch diesen Schnitt sind alle Polyeder betroffen, die ansonsten Raum auf beiden Seiten der Gitterebene belegen würden. Sie werden dadurch in zwei neue räumlich disjunkte Polyeder aufgeteilt. Dadurch, dass diese Aktion auf alle Polyeder angewendet wird, wird sichergestellt, dass die dadurch entstandenen Gruppen ebenfalls zueinander räumlich disjunkt sind. Somit fällt eine rechenintensive Nachbarschaftsbetrachtung weg und die lokalen Gruppen können

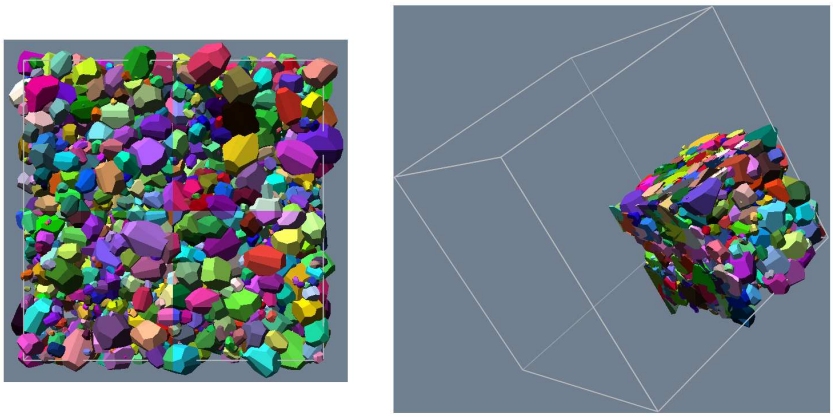

Abbildung 6.1: Schnitt des Modells am Gitternetz

parallel bearbeitet werden, ohne dass Informationen aus einer Nachbargruppe benötigt werden. Dies alles erleichtert die Implementation erheblich, da Rechnerknoten, auf denen die Gruppen abgearbeitet werden sollen nicht miteinander kommunizieren müssen und als Ergebnis nur das berechnete Volumen ihre Gruppe zurückgeben müssen.

6.1.2 Erstellen eines Rasters fester Größe

Ein naiver Ansatz zur Umsetzung des Schnitts am Gitter ist einfach alle Polyeder mit allen Gitterebenen zu schneiden. Dieser Ansatz hat jedoch zwei entscheidende Nachteile. Bei einer Aufteilung in kleine Gruppen werden viele Gitterebenen benötigt. Dadurch wächst die Laufzeit zur Erstellung der Gruppen auf $\mathcal{O}(k * n)$ mit k Anzahl der Gitterebenen und n Anzahl der Polyeder. Der andere Nachteil ist, dass durch stures Aufteilen keine Reaktion auf die Polyeder-Dichte vorgenommen werden kann. Gerade weniger dichte Außenbereiche werden somit weiter aufgeteilt als notwendig.

6.1.3 Dynamische Raumaufteilung durch Rekursion

Um die im vorherigen Abschnitt angesprochenen Nachteile zu verlieren, ist eine rekursive Raumaufteilung sinnvoll. Dabei kann durch geschickte Parameterwahl erreicht werden, dass dichte Regionen stärker aufgeteilt werden als weniger dichte. Zudem wird der Schnitt am Gitternetz nicht mehr auf alle Polyeder angewandt, sondern nur auf die lokalen Subprobleme. Außerdem lassen sich die Subprobleme parallelisieren, da diese einen räumlich disjunkten Bereich abdecken.

Die Laufzeit der dynamischen Rekursion ist somit

(6.1) $\mathcal{O}\left(2 * l * n\right)$

mit l maximaler Rekursionstiefe und n Anzahl der Polyeder.
Im Vergleich dazu die Laufzeit für die feste Aufteilung, bei gleich vielen Gitterebenen:

$$\mathcal{O}\left(\left(\sum_{i=1}^{l} 2^i\right) * n\right) \tag{6.2}$$

Octree

Zur Implementierung der dynamischen Aufteilung eignet sich die Datenstruktur des Octree. Auf ihm kann die Aufteilung einfach und effizient ausgeführt werden. Dabei sind die aufgeteilten Räume in den Blättern des Baums zu finden. Für die dynamische Aufteilung wird jedoch noch ein Abbruchkriterium benötigt. Im Folgenden wird deshalb nun ein mögliches Abbruchkriterium eingeführt.

Abbruchkriterium

Die Idee hinter dem Abbruchkriterium ist, dass abgebrochen wird, sobald die Dichte hinreichend gering ist, um den eigentlichen Algorithmus effizient anwenden zu können. Für dieses Modell bietet es sich an, die Anzahl der Überschneidungen der Polyeder in der Gruppe zu überprüfen. Übersteigt diese Anzahl einen Schwellwert, wird weiter aufgeteilt. Ansonsten wird die aktuelle Gruppe als Blatt dem Baum hinzugefügt. Da jedoch die Berechnung der Überschneidungen gerade in sehr großen Gruppen sehr viel Zeit kostet, weil alle Polyeder miteinander überprüft werden müssen, bietet es sich an, einen weiteren Schwellenwert für die Anzahl der Polyeder in der Gruppe einzuführen. Dieser ist zwar weniger aussagekräftig, kann allerdings als erstes überprüft werden und nur wenn die Anzahl der Polyeder geringer als der Schwellenwert ist, wird auf die Überschneidungen zurückgegriffen.

Parameterwahl Die Wahl der Parameter hängt dabei vom gegebenen Modell ab. In Tabelle 7.1 auf Seite 41 werden die Auswirkungen der Parameter auf die Erstellung des Octree dargestellt, in den darauf folgenden Schaubildern die Auswirkungen auf die Ausführung des Algorithmus. Das dazu verwendete Beispielmodell und die Folgerungen aus den Messergebnissen werden in Kapitel 7 näher erläutert.

Tiefenlimit Bei strenger Wahl der Schwellwerte kann es jedoch passieren, dass trotz weiterer Aufteilung der Wert nicht unter die Parameterwerte sinkt. Dadurch kann es zu sehr tiefen Stellen im Baum kommen. In diesem Fall würde die Erstellung des Baums viel zu lange dauern, es wäre also ineffizient. Darum kann es sinnvoll sein, ein Tiefenlimit für den Baum einzuführen. Die Auswirkungen dieses zusätzlichen Parameters werden ebenfalls in der Tabelle 7.1 auf Seite 41 aufgezeigt und in Kapitel 7 erläutert.

6.2 Bearbeitung in lokalen Gruppen

Der naive Vorgang zum Bearbeiten der nun erstellten lokalen Gruppen wäre, dass in jedem Schritt ein Polyeder aus der Liste genommen wird und mit allen anderen Polyedern in dieser Liste geschnitten wird. Das Ergebnis dieses Schnitts wird dann wiederum in die Liste aufgenommen. Der Algorithmus wäre dann fertig, wenn kein Polyeder mehr in der Liste ist. Dieser naive Ansatz ist jedoch ineffizient und führt zu sehr vielen vermeidbaren Abfragen. Zusätzlich wird es schwieriger auf so einer Datenstruktur mit Heuristiken zur Optimierung zu arbeiten. Deswegen werden im Folgenden Ansätze zur besseren Bearbeitung der Gruppen gezeigt.

6.2.1 Bearbeitung mit Überschneidungs-Graphen

Eine sehr effiziente Möglichkeit der Bearbeitung der lokalen Gruppen ist die Verwendung eines Graphen als Datenstruktur. In diesem Graphen entspricht jeder Knoten einem Polyeder und jede Kante zwischen zwei Knoten bedeutet, dass diese Polyeder sich höchstwahrscheinlich überschneiden. Dabei besteht die Möglichkeit, dass eine Kante zwischen zwei Knoten existiert, obwohl diese sich nicht überschneiden. Dies ist auf die Kantenvererbung im Algorithmus zurückzuführen, die in den nächsten Abschnitten erklärt wird. Jedoch kann nie vorkommen, dass sich zwei Polyeder überschneiden aber keine Kante dazwischen existiert. Die Bearbeitung der Gruppe wird somit über den Algorithmus 6.1 vorgenommen.

Im Pseudocode werden anfangs die beiden Algorithmen eingebunden, welche die eigentliche Arbeit des paarweise Aufteilens der Polyeder verrichten. Diese werden später aufgerufen.

Am Anfang des Algorithmus wird zunächst der Graph aufgebaut. Dies geschieht, indem alle Polyeder aus der lokalen Gruppe erst in einen Knoten verwandelt und anschließend in den Graphen hinzugefügt werden. Dabei wird angenommen, dass die *Add*-Funktion des Graphen die Aufgabe des Erstellens der Kanten übernimmt. Nach dem Hinzufügen aus der lokalen Gruppe, wird dieser Polyeder aus der Gruppe vorläufig entfernt, da die Gruppe später im Algorithmus zum Speichern der fertig bearbeiteten Polyeder verwendet wird.

In der Haupt-Schleife wird nun der jeweils nächste Knoten aus dem Graphen genommen und in die Ausgabeliste aufgenommen. Dies ist möglich, da durch die Ausführung des *Merge-And-Split*-Algorithmus dieser nicht verändert wird, sondern lediglich der jeweils andere Polyeder durch diesen geschnitten wird. Dieser Schnitt wird nun im nächsten Schritt ausgeführt, indem jeder über eine Kante erreichbarer Polyeder geschnitten wird. Das Ergebnis dieses Schnitts ersetzt nun den ursprünglichen, geschnittenen Polyeder. Hier wird zudem angenommen, dass die *Replace*-Funktion des Graphen alle Kanten des zu ersetzenden Polyeders löscht und diese allen seinen *Kindern* weitervererbt. Dadurch wird sichergestellt, dass die neuen Polyeder auch nur mit denen verglichen werden, die vom ursprünglichen ebenfalls geschnitten wurden. Anschließend wird der erste Polyeder aus dem Graphen entfernt und damit alle seine Kanten gelöscht, da diese bereits abgearbeitet wurden. Die

Algorithmus 6.1 Bearbeitung der Gruppen mit Graph

```
Intersect_Using_Graph : ( G : Group ) -> Group
  Types
    Node : ( Value : Polyhedra, Edges : PointerToNode[] )

  Imports
    Merge_Bspts : ( T1, T2 : Bspt ) -> Bspt
    Split_Bspts : ( T : Bspt ) -> Bspt[]

  Definition
    Graph : Graph_Structure

    FOR EACH P : Polyhedra IN G DO
      Graph.Add ( NEW Node ( P ))
      G.Remove ( P )
    END FOR

    FOR EACH N1 : Node IN Graph DO
      G.Add ( N1.Value )

      FOR EACH E : PointerToNode IN N1.Edges DO
        Graph.Replace ( E.Node, Split_Bspts ( Merge_Bspts ( N1.tree, E.Node.tree )))
      END FOR

      Graph.Remove ( N1 )
    END FOR

    RETURN G
END Intersect_Using_Graph
```

Schleife wird nun solange durchlaufen, bis der Graph leer ist. Dann wird die Ausgabegruppe zurückgegeben.

6.3 Heuristiken zur Optimierung

Durch das Wissen über das Modell können Heuristiken abgeleitet werden mit denen sich der Algorithmus weiter verbessern lässt. Die Ergebnisse zu der Wirkung dieser Heuristiken werden im Kapitel 7 dargelegt.

6.3.1 Überflüssige Kanten eliminieren

Die neu erstellten Polyeder bekommen jeweils alle Kanten von dem Polyeder vererbt, aus denen sie erstellt wurden. Dadurch kann es dazu kommen, dass Kanten eine Überschneidung anzeigen, die gar nicht existiert. Darum kann es sich lohnen schon früh solche Kanten zu eliminieren. In der Abbildung 6.2 wird so eine Situation verdeutlicht. Werden die Kanten nicht frühzeitig entfernt, kommt es nach mehreren Runden zu einer Vielzahl überflüssiger

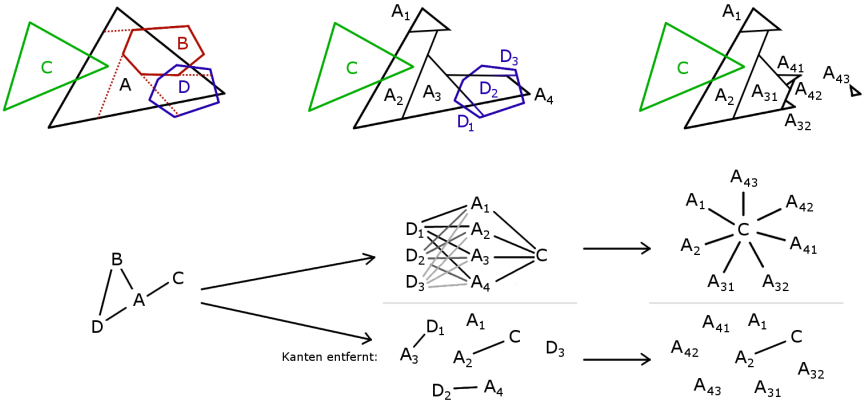

Abbildung 6.2: Beispiel für Optimierung durch Kanten-Elimination

Kanten und dadurch zu erhöhtem Aufwand. Der Eliminierungsprozess muss sich dabei nur auf die jeweils neuen Polyeder erstrecken, da alle anderen Polyeder spätestens im letzten Schritt genau so behandelt wurden und deshalb keine unnötigen Kanten haben können. Wie sich dieser Optimierungsschritt in den Algorithmus einfügt, wird in der erweiterten Fassung des Algorithmus im Pseudocode 6.2 gezeigt.

6.3.2 Knoten sortieren

Im Graphen gibt es mehrere Möglichkeiten nach denen Knoten sortiert werden können. Im Folgenden werden zwei Möglichkeiten vorgestellt, durch welche sich gewisse Vorteile erhoffen lassen. Dabei wird dieser Sortierschritt immer vor der nächsten Runde ausgeführt, sodass das *größte* bzw. *kleinste* Element als nächstes behandelt wird. Zusätzlich hilft es zu wissen, dass es nicht entscheidend ist, dass die ganze Liste sortiert ist. Benötigt wird jeweils nur das größte oder kleinste Element. Somit kann eine einfache Suche ausgeführt werden, die auf einer Liste angewendet in $\mathcal{O}(n)$ abläuft. Bei der Wahl großer lokaler Gruppen wäre es jedoch sinnvoller die Knoten in einer such-freundlichen Datenstruktur zu verwalten, um die Suche auf die Laufzeit $\mathcal{O}(\log n)$ zu verringern. Die Reihenfolge der paarweisen Anwendung des *Merge-And-Split*-Algorithmus auf den ausgewählten Polyeder und jeweils damit verbundenen spielt hier keine Rolle, da diese sich gegenseitig nicht beeinflussen.

Nach Anzahl der Kanten

Durch diesen Sortiervorgang werden die Knoten so angeordnet, dass diese absteigend nach der Anzahl ihrer Kanten sortiert sind. Von diesen Knoten wird dann der mit den meisten Kanten als nächstes Element ausgewählt. Durch diese Auswahl wird erreicht, dass in jedem Schritt möglichst viele Kanten abgebaut werden, die dadurch nicht mehr an mehrere Kinder

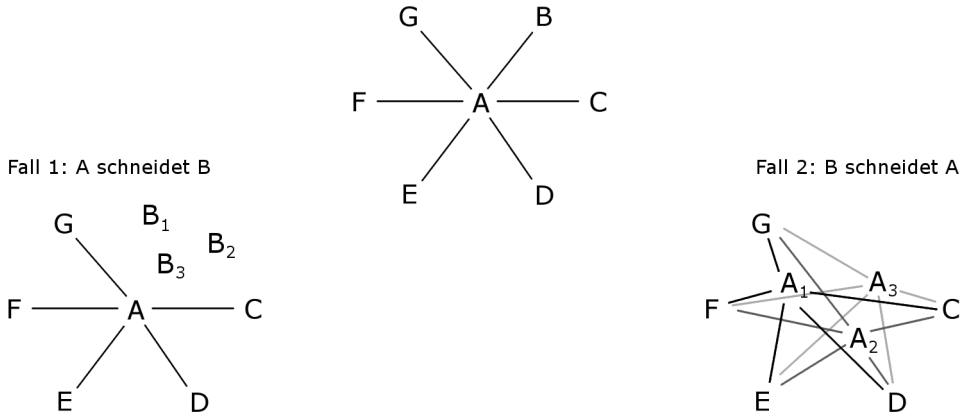

Abbildung 6.3: Problem der Kantenvererbung

vererbt werden können.

Ein extremes Beispiel wird in Abbildung 6.3 gezeigt. Im ersten Fall teilt A hier B auf. Da B jedoch nur diese Kante zu A hatte, haben alle resultierenden $B_1, B_2, ..., B_n$ auch keine Kanten. Diese gute Reihenfolge, dass erst A verarbeitet wird, führt zu einer Laufzeit von $\mathcal{O}(k)$ mit k Anzahl der Kanten in der Ausgangssituation. Im zweiten Fall jedoch wird A von B geschnitten und aufgeteilt. Nun werden alle übrigen Kanten von A an dessen Kinder vererbt, somit beläuft sich nach nur einem Schritt die Anzahl der Kanten auf $m * (k-1)$ mit m Anzahl der erstellten Kinder von A. Geht man nun von dem Worst-Case-Szenario aus, wird in jedem folgenden Schritt die schlechteste Wahl getroffen.

Für den schlechtesten Fall erhält man bei k Anzahl der ursprünglichen Kanten und m Anzahl der durchschnittlichen pro Schnitt erstellten neuen Polyedern demnach nach der i-ten Runde folgende Anzahl an verbleibenden Kanten:

(6.3) $m^i\,(k-i)$

Benötigte paarweise Schnitte sind insgesamt dann

(6.4) $\sum_{i=0}^{k-1} m^i \in \mathcal{O}(m^{k-1})$

Im Vergleich dazu die Anzahl der benötigten Schritte für die gut gewählte Reihenfolge

(6.5) $k \in \mathcal{O}(k)$

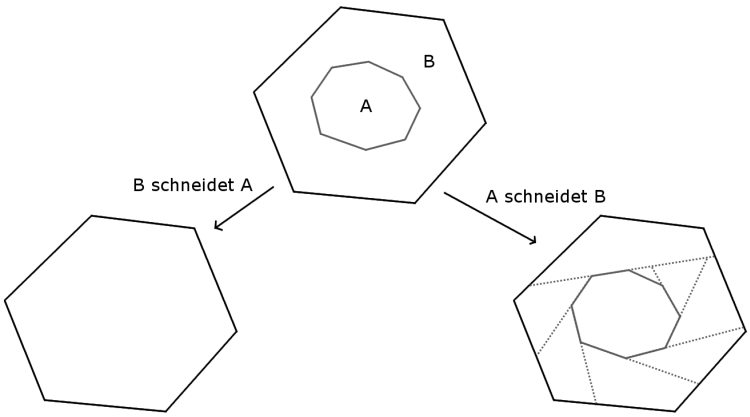

Abbildung 6.4: Optimierung durch Sortierung nach Größe des Polyeders

Größe des Polyeders

Eine andere Möglichkeit zu Sortieren ist nach der Größe des repräsentierten Polyeders des Knoten. Am einfachsten lässt sich diese Wahl anhand eines Beispiels erläutern. Dieses Beispiel in 2D, in Abbildung 6.4, zeigt wie aus einem Worst-Case ein Best-Case werden kann. In der unsortierten Liste käme es demnach zu der Möglichkeit, dass durch den Schnitt zweier Polyeder, wobei das kleinere ein n-Eck ist, n neue Polyeder entstehen können. In der sortierten Liste hingegen kommt es zum Best-Case, dass der kleinere Polyeder komplett überdeckt wird und somit kein neues Polyeder entsteht. Im allgemeinen Fall würde hingegen erreicht werden, dass in jedem Schritt durchschnittlich weniger neue Polyeder entstehen. Aus diesem Grund wird in jedem Zyklus der Knoten des größten Polyeders ausgewählt.

Für die Ermittlung der Größe eines Polyeders wird dabei auf Genauigkeit verzichtet, es reicht eine grobe Approximation. Dies ist begründet darin, dass zwei in etwa gleich große Polyeder beim Schnitt keinen erkennbaren Unterschied zulassen. Dafür ist der Aufwand für eine genauere Bestimmung der Polyedergröße nicht gerechtfertigt.

6.4 Zukünftige Arbeiten

Abschließend sollen weitere Ideen zur Anwendung der Algorithmen auf große Datensätze vorgestellt werden. Diese sind im Rahmen dieser Bachelor-Arbeit nicht implementiert und getestet worden, bieten aber eine Grundlage, um den Algorithmus weiter zu verbessern.

Algorithmus 6.2 Bearbeitung der Gruppen mit Graph und Optimierungsschritten

```
Intersect_Using_Graph_Optimized : ( G : Group ) -> Group
  Types
    Node : ( Value : Polyhedra, Edges : PointerToNode[] )

  Imports
    Merge_Bspts : ( T1, T2 : Bspt ) -> Bspt
    Split_Bspts : ( T : Bspt ) -> Bspt[]

  Definition
    Graph : Graph_Structure

    FOR EACH P : Polyhedra IN G DO
      Graph.Add ( NEW Node ( P ))
      G.Remove ( P )
    END FOR

    DO UNTIL Graph.IsEmpty
      N1 : Node := Graph.Best_Node
      G.Add ( N1.Value )

      FOR EACH E : PointerToNode IN N1.Edges DO
        Graph.Replace ( E.Node, Split_Bspts ( Merge_Bspts ( N1.tree, E.Node.tree )))
      END FOR

      Graph.Remove ( N1 )

      FOR EACH REPLACEMENT N1 : Node IN GRAPH DO
        FOR EACH E : PointerToNode IN N1.Edges DO
          IF NOT Intersects ( N1, E.Node ) THEN Remove_Edge ( N1, E.Node )
        END FOR
      END FOR
    END DO

    RETURN G
END Intersect_Using_Graph_Optimized
```

6.4.1 Optimierung: Auswahl des besten Ergebnis

Eine weitere Idee zum Verbessern des Algorithmus ist, dass beim paarweisen Schneiden der Polyeder beide Schnitt-Reihenfolgen durchgeführt werden. Somit würde einmal der erste den zweiten Polyeder schneiden und als nächstes der zweite den ersten. Aus diesen beiden Ergebnissen wird dann das vielversprechendere Ergebnis verwendet. Die Entscheidung nach der besseren Auswahl könnte dann nach der Anzahl der erstellten Polyeder geschehen, sodass wiederum weniger Kinder entstehen. Dies wäre eine Möglichkeit, die zur Anwendung kommen kann, wenn sich zwei in etwa gleich große Polyeder schneiden und das Ergebnis trotzdem stark variiert. Als weiteres Auswahlkriterium wäre möglich, dass die erstellten Polyeder auf deren Größe untersucht werden und die Gruppe weiter benutzt wird, durch die man sich eine höhere numerische Genauigkeit beim späteren Triangulieren und beim Berechnen des Volumens erhofft.

Die Vermutung ist jedoch, dass diese Heuristik trotz des Produzierens besserer Ergebnisse die Nachteile der doppelten Ausführung des Algorithmus nicht ausgleichen kann. Zumal durch die vorherig angeführten Optimierungsmöglichkeiten die relevanten Fälle bereits optimiert wurden.

7 Messwerte und Ergebnisse

Im Folgenden werden nun Messwerte und Ergebnisse dargestellt und erläutert. Dabei sind alle Messungen der ersten beiden Abschnitte *Messergebnisse des Testdatensatzes* und *Qualität der Volumenberechnung* auf Rechner-Knoten mit zwei Xeon E5620@2.4GHz (insgesamt 8 Kerne), 24 GB Arbeitsspeicher und Windows 7 64 Bit ausgeführt worden. Die Messungen im Abschnitt *Auswirkungen der Heuristiken* auf einem Intel(R) Core(TM)2 Duo CPU T6400 @ 2.00GHz (insgesamt 2 Kerne), 4 GB Arbeitsspeicher und Windows 7 64 Bit.

7.1 Messergebnisse des Testdatensatzes

Alle in diesem Abschnitt vorgelegten Messergebnisse wurden auf einem Testdatensatz mit 10.000 Polyedern erstellt. Von diesem ist jedoch nicht bekannt, welches Volumen zu erwarten ist. Darum wird vorwiegend auf die Effizienz des Algorithmus eingegangen. Erst im nächsten Abschnitt wird auf die Effektivität, also die Qualität der Ergebnisse eingegangen.

In Tabelle 7.1 wird die Dauer der Octree-Erstellung mit verschiedenen Parametern aufgezeigt. Wie auch in den Schaubildern werden die Parameter immer in folgender Form angegeben: Als erstes steht der Datensatz, gekennzeichnet durch die Anzahl der Polyeder, danach kommt der Parameter für die maximale Anzahl Polyeder pro Gruppe, anschließend der Parameter für die maximale Anzahl der Überschneidungen pro Gruppe. In dieser Tabelle ist die Angabe in den Spalten die jeweilige Ausführungsdauer in Sekunden bei angegebener maximalen Tiefe des Baums.

Diese gemessenen Zeiten zur Erstellung des Baums sind sehr gering, sodass auch die dreifache Erstelldauer für eine höhere Stufe nicht ins Gewicht fällt. Selbst bei der Octree-Erstellung für den Millionen Polyeder Datensatz, würde sich die Zeit auf weniger als eine Stunde belaufen.

Was weitaus mehr ins Gewicht fällt, ist die Anzahl der erstellten Polyeder durch das Aufteilen am Gitternetz. In Abbildung 7.1 werden diese in einem Schaubild dargestellt. Hier wird sehr gut deutlich, dass durch zu häufiges Aufteilen des Raums sehr viele Polyeder entstehen. Auf der anderen Seite heißt dies jedoch auch, dass die lokalen Gruppen weniger Polyeder enthalten und somit sich schneller bearbeiten lassen. Dadurch wird nämlich der Unabhängigkeits-Graph, oder eine ähnliche Datenstruktur die zur Verwaltung der Polyeder eingesetzt wird, weniger komplex. Dies lohnt sich jedoch nur bis zu einer bestimmten Stufe, wie in Abbildung 7.2 erkennbar ist. Anschließend ist die Tendenz, dass durch die stark steigende Gesamtzahl der Polyeder mehr Zeit in Anspruch genommen wird.

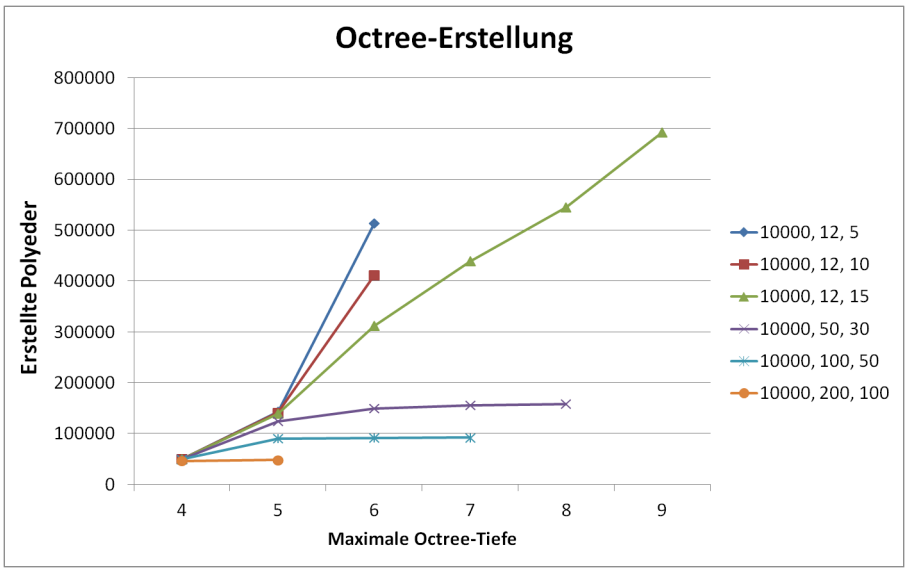

Abbildung 7.1: Anzahl der Polyeder nach der Raumaufteilung durch den Octree

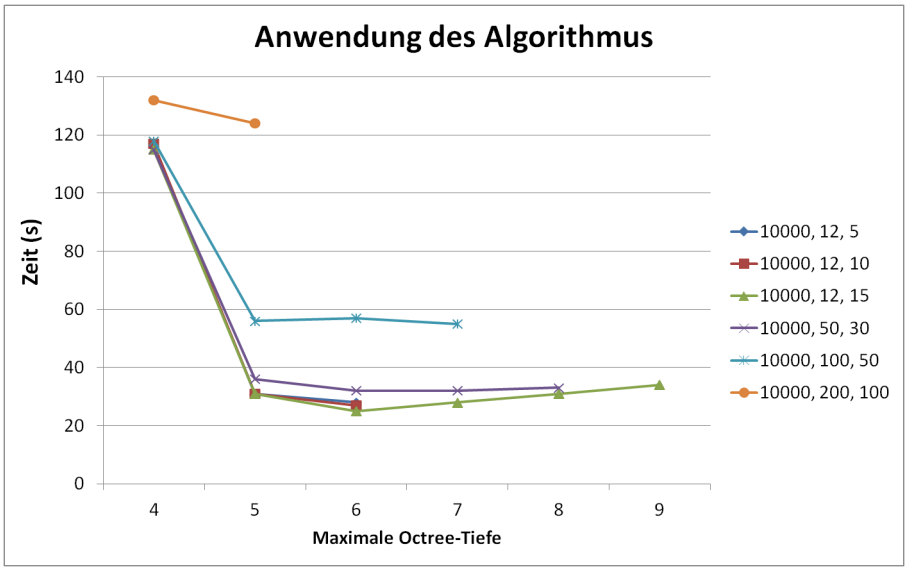

Abbildung 7.2: Dauer der Algorithmus-Ausführung

	Stufe 4	Stufe 5	Stufe 6	Stufe 7	Stufe 8	Stufe 9
10000, 12, 5	5	8	15			
10000, 12, 10	4	7	13			
10000, 12, 15	4	7	11	13	14	16
10000, 50, 30	5	9	10	9	9	
10000, 100, 50	7	10	10	9		
10000, 200, 100	8	8				

Tabelle 7.1: Dauer der Octree-Erstellung in Sekunden

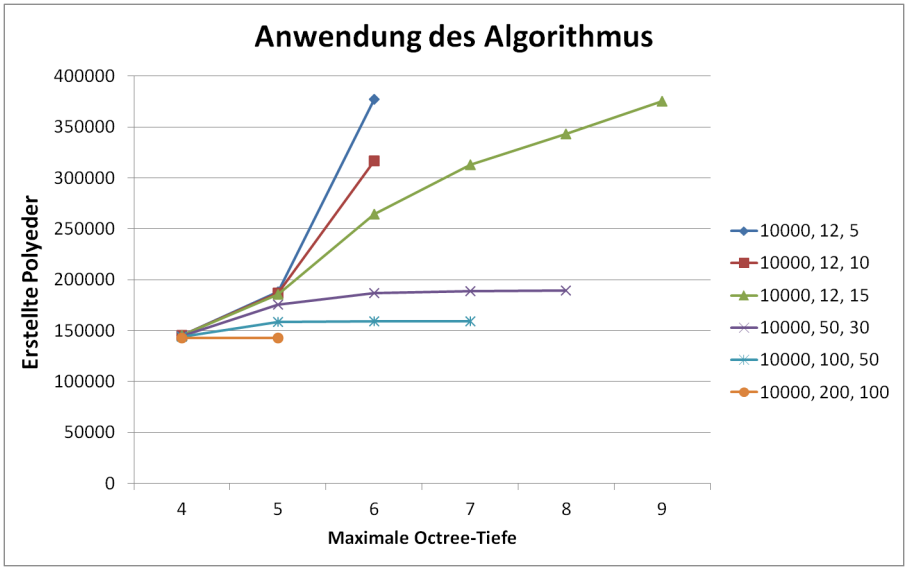

Abbildung 7.3: Anzahl Polyeder nach der Anwendung des Algorithmus

Eine weitere interessante Beobachtung ist, wenn man die Abbildungen 7.1 und 7.3 vergleicht. Hier kommt es zu dem Phänomen, dass nach Ausführung des Algorithmus weniger Polyeder vorhanden sind als davor. Dies lässt darauf schließen, dass durch vielmaliges Aufteilen des Raums auch sehr viele kleine Polyeder entstehen, die vollständig von anderen Polyedern umschlossen werden. Somit kommt es dazu, dass diese Polyeder einfach entfernt werden können. Trotz dieser Rückläufigkeit der Anzahl der Polyeder, ist diese Zahl weiterhin mit steigender Octree-Stufe signifikant höher. Dies wirkt sich somit negativ auf die Dauer der Tetraedisierung und die Volumenberechnung aus. Diese beiden Schritte sind in Abbildung 7.4 zusammengefasst. Vergleicht man diese Ergebnisse mit denen in Abbildung 7.3, wird dies deutlich. Gut zu sehen ist dies am Beispiel der Parameter *12, 5* (dunkelblau) bei Baumtiefe *6* und *12, 15* (grün) bei Tiefe *9*. Die Werte an diesen Stellen sind in beiden Diagrammen jeweils in etwa gleich.

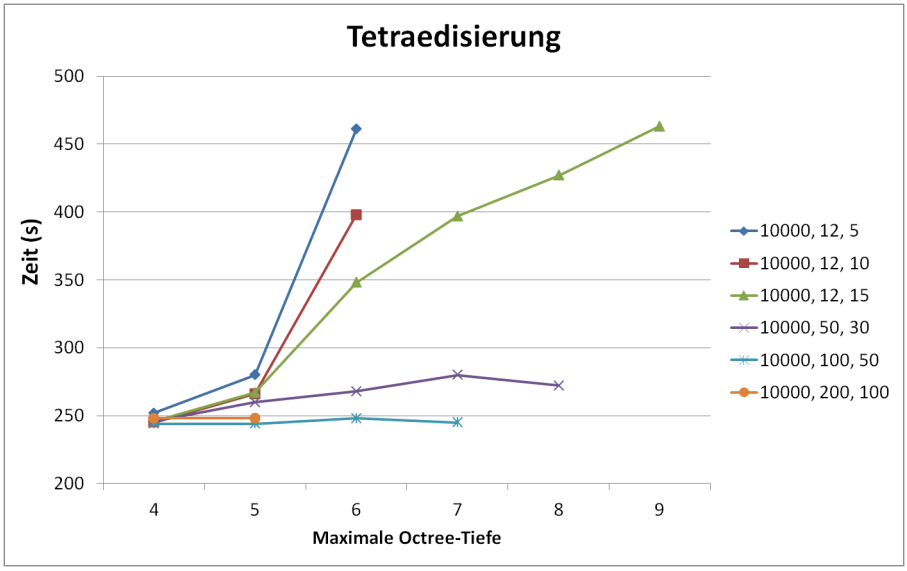

Abbildung 7.4: Dauer der Tetraedisierung und der Volumenberechnung

Diese Ergebnisse über die Dauer der Ausführung, werden in Abbildung 7.5 nochmals zusammengefasst. So ist für jeden Parameter die beste Baumtiefe offensichtlich gleich. Vergleicht man dieses Ergebnis jedoch mit dem berechneten Volumen in Abbildung 7.6, muss davon ausgegangen werden, dass das bessere Ergebnis jedoch schon bei einer niedrigeren Octree-Stufe gewonnen wird. Auf diese Theorie wird verstärkt im nächsten Abschnitt eingegangen.

7.2 Qualität der Volumenberechnung

In diesem Abschnitt wird nun auf die Qualität der Volumenberechnung eingegangen. Dazu wird ein Datensatz mit 25.000 Polyedern verwendet. Für diesen existiert ein Referenzwert für die Porösität. Um jedoch aus dem Volumen die Porösität berechnen zu können, muss eine Box um den Stein gesetzt werden. Somit ist das Einschlussvolumen des Steins

$$V_{Einschluss} = V_{Box} - V_{Stein}. \tag{7.1}$$

Dementsprechend ist die Porösität $\mathcal{P}$ das Verhältnis vom Einschlussvolumen zum Volumen der Box:

$$\mathcal{P} = \frac{V_{Einschluss}}{V_{Box}} = \frac{V_{Box} - V_{Stein}}{V_{Box}} \tag{7.2}$$

Die Ergebnisse für diesen Datensatz sind in den beiden Abbildungen 7.7 und 7.8 zusammengefasst. Zusätzlich ist der erwartete Wert eingezeichnet. Gut erkennbar ist, dass der

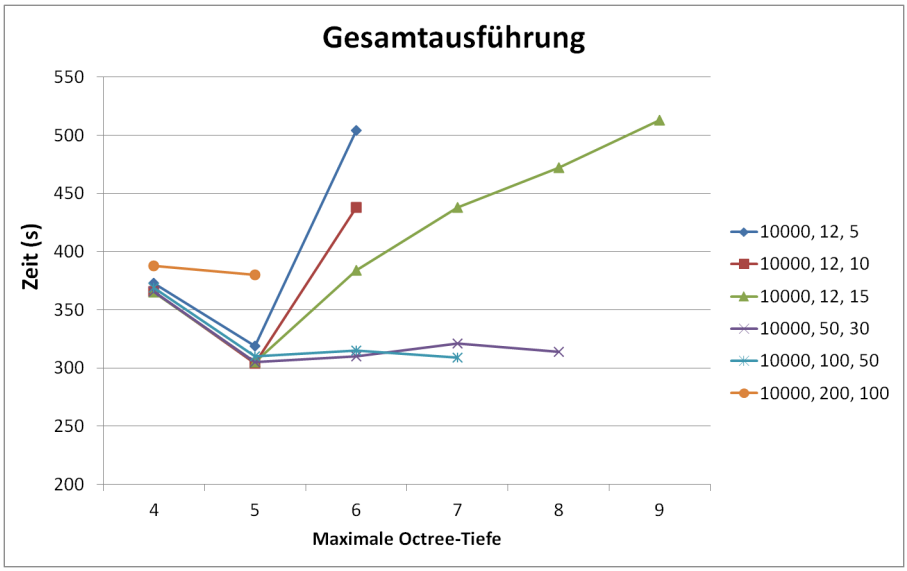

Abbildung 7.5: Dauer der Gesamtausführung

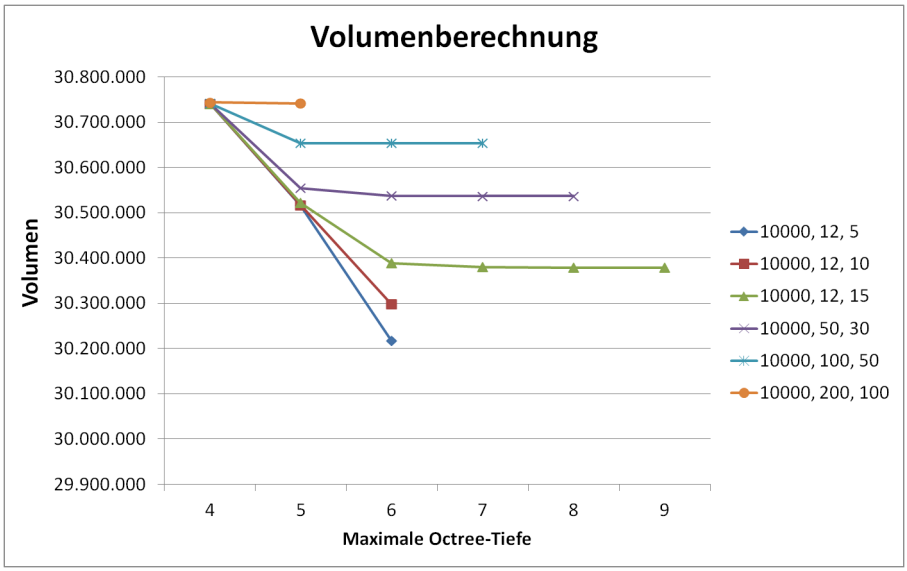

Abbildung 7.6: Berechnetes Volumen

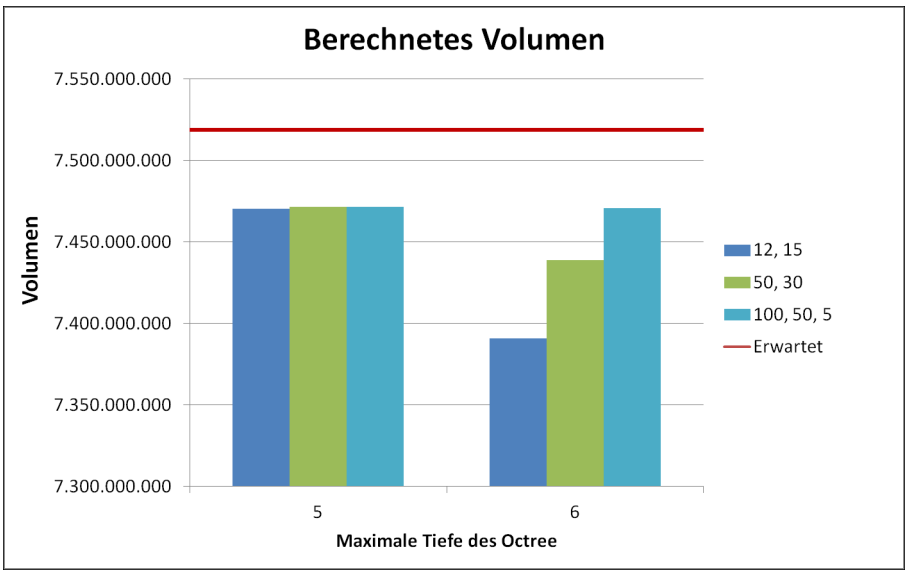

Abbildung 7.7: Berechnetes und erwartetes Volumen

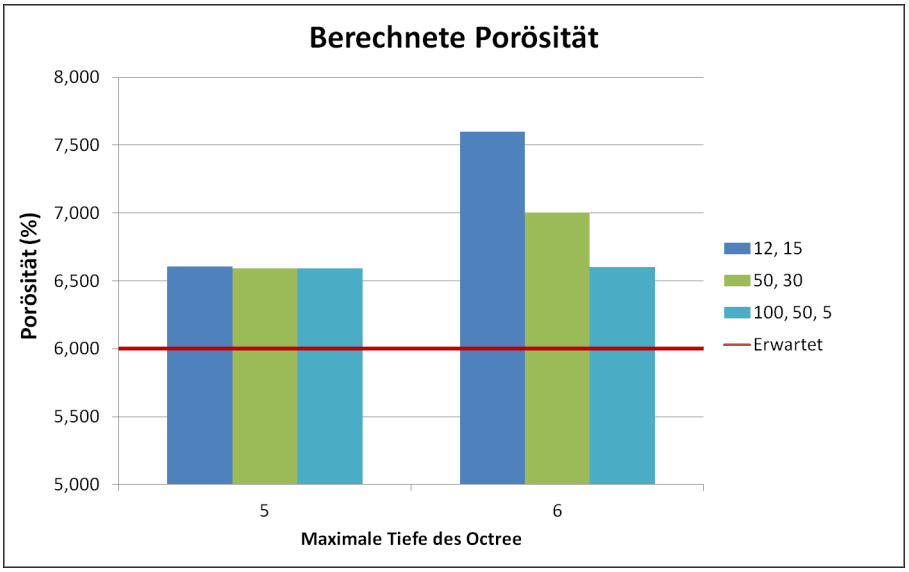

Abbildung 7.8: Berechnete und erwartete Porösität

berechnete Wert genauer ist, wenn das Volumen aus weniger und somit größeren Polyedern berechnet wird. Dies ist genau nicht der Fall, wenn der Octree stark weiter unterteilt wird, wie für die Parameter *12, 15* und *50, 30* bei Tiefe *6* und höher.

Nachfolgend werden die absoluten und relativen Fehler berechnet. Dazu werden die Werte aus der Tabelle 7.2 verwendet.

	Wert
Erwartete Porösität	6,00%
Berechnete Porösität	6,59%
Erwartetes Volumen	7518829527,04
Bestes berechnetes Volumen	7471519629,178760
Box-Volumen	7998754816,000000

Tabelle 7.2: Berechnete und erwartete Werte für den 25.000 Polyeder Datensatz

Für den besten berechneten Volumenwert ist der absolute Fehler

(7.3)
$$e^V_{absolut} = |V_{erwartet} - V_{berechnet}| = 7518829527,04 - 7471519629,178760 = 47309897,86124$$

Somit ist der relative Fehler:

$$e^V_{relativ} = \frac{e^V_{absolut}}{V_{erwartet}} = \frac{47309897,86124}{7518829527,04} \approx 0,00629 < 1\% \tag{7.4}$$

Für eine ungefähre Volumenberechnung wäre dies ein akzeptabler Fehler. Jedoch wirkt sich dieser Fehler sehr stark auf die berechnete Porösität aus.

$$\begin{aligned} e^{\mathcal{P}}_{absolut} &= |\left(\frac{V_{Box} - V_{Stein}}{V_{Box}}\right) - \left(\frac{V_{Box} - V^*_{Stein}}{V_{Box}}\right)| \\ e^{\mathcal{P}}_{absolut} * V_{Box} &= |V_{Box} - V_{Stein} - V_{Box} + V^*_{Stein}| \\ e^{\mathcal{P}}_{absolut} * V_{Box} &= |-(V_{Stein} - V^*_{Stein})| \\ e^{\mathcal{P}}_{absolut} * V_{Box} &= e^V_{absolut} \\ e^{\mathcal{P}}_{absolut} &= \frac{e^V_{absolut}}{V_{Box}} \end{aligned} \tag{7.5}$$

Es ergibt sich der absolute Fehler:

$$e^{\mathcal{P}}_{absolut} = \frac{e^V_{absolut}}{V_{Box}} = \frac{47309897,86124}{7998754816,000000} \approx 0,00591 \tag{7.6}$$

Somit ist der relative Fehler:

$$e^{\mathcal{P}}_{relativ} = \frac{e^{\mathcal{P}}_{absolut}}{\mathcal{P}_{erwartet}} \approx \frac{0,00591}{0,06} \approx 0,099 \approx 10\% \tag{7.7}$$

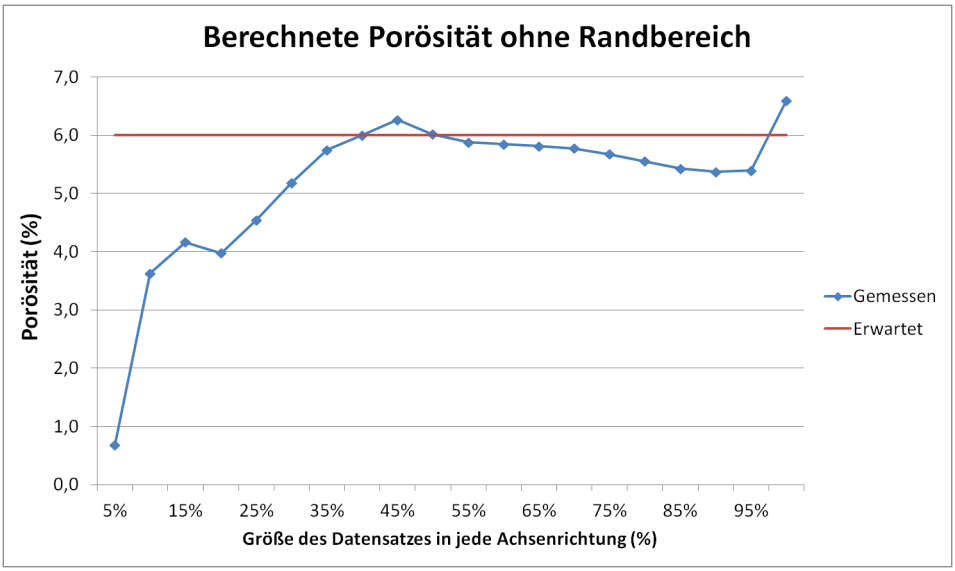

Abbildung 7.9: Berechnete und erwartete Porösität ohne Randbereich (letzter Wert mit Rand)

Jedoch kann der Fehler weiter reduziert werden. Was bisher nicht beachtet wurde, ist, dass der Randbereich eine geringere Dichte hat als der innere Bereich. Um ein genaueres Ergebnis zu erhalten, muss also der Randbereich weggeschnitten werden. Dazu wird wie oben eine Box erstellt, die etwas kleiner als der Stein ist. An dieser Box wird wie an einer Bounding-Box so geschnitten, dass die Polyeder an dieser Box aufgeteilt werden und nur alle Polyeder in die Ausführung des Algorithmus einfließen, die in der Box liegen.

Durch diese Modifikation erhält man die Ergebnisse, die in der Abbildung 7.9 und in der Tabelle 7.3 dargestellt werden. Die Ergebnisse wurden alle mit dem 25.000 Polyeder Datensatz erstellt, mit den Parametern *50*, *30*, bei maximaler Tiefe *5* des Baums.

Die berechneten Werte für die verschieden großen Würfel, die aus dem Gesamtdatensatz herausgeschnitten wurden, zeigen, dass es Stellen gibt an denen der Fehler sehr gering ist. Bei zu kleinen Würfeln kommt es jedoch zu großen Fehlern, da die Zufälligkeit der Dichte hier zu hoch ist. Auf der anderen Seite stellen sich bei großen Würfeln durch die hohe Anzahl der Polyeder auch wiederum größere Fehler ein.

Durch mehrere Durchläufe des Algorithmus mit unterschiedlichen Parametern für die Würfelgröße können also gute Ergebnisse erzielt werden.

7.3 Auswirkungen der Heuristiken

Die am Ende des letzten Kapitels (6.3 auf Seite 33) eingeführten Heuristiken werden im Folgenden auf ihre Auswirkung auf die Ausführung des Algorithmus hin untersucht. Darum

Größe des Ausschnitts (in Achsenrichtung)	Berechnetes Volumen	Erwartetes Volumen	Relativer Fehler
5%	993.092,78266	939.853,65394	0,056646
10%	7.708.909,11643	7.518.829,23149	0,025281
15%	25.871.972,81493	25.376.048,65629	0,019543
20%	61.448.804,96672	60.150.633,85195	0,021582
25%	119.302.660,64233	117.481.706,74209	0,015500
30%	204.779.139,47767	203.008.389,25032	0,008723
35%	323.252.543,66710	322.369.803,30028	0,002738
40%	481.222.044,71953	481.205.070,81558	0,000035
45%	683.253.352,02296	685.153.313,71984	0,002773
50%	939.735.256,68549	939.853.653,93668	0,000126
55%	1.252.560.426,35491	1.250.945.213,38972	0,001291
60%	1.626.751.979,92204	1.624.067.114,00258	0,001653
65%	2.069.053.893,39243	2.064.858.477,69888	0,002032
70%	2.585.197.234,91499	2.578.958.426,40224	0,002419
75%	3.183.076.081,46473	3.172.006.082,03629	0,003490
80%	3.867.985.774,72098	3.849.640.566,52463	0,004765
85%	4.645.901.799,03830	4.617.501.001,79090	0,006151
90%	5.517.970.205,99205	5.481.226.509,75872	0,006704
95%	6.488.315.106,89169	6.446.456.212,35168	0,006493
100% (mit Rand)	7.471.364.525,72715	7.518.829.527,04000	0,006313

Tabelle 7.3: Berechnetes und erwartetes Volumen, sowie Fehler bei abgeschnittenen Rand

gibt es für jeden Abschnitt in der Gesamtausführung, den die Heuristiken beeinflussen ein Schaubild mit Messergebnissen. Jedoch existieren keine konkreten Ergebnisse für die Ausführung nur mit Kantenentfernung und für die Ausführung ohne Heuristik. Diese Durchläufe sind so ineffizient, dass die Ausführungszeit ein Vielfaches von der Dauer der hier aufgeführten Messungen ist und nicht bis zum Ende gerechnet wurden.

In Abbildung 7.10 wird schon sehr deutlich, dass die Sortierung nach Polyedergröße im Vergleich zu den anderen Heuristiken sehr effizient ist und sogar in Kombination mit dem Entfernen der Kanten keine Verbesserungen mehr zulässt. Das Gegenteil ist der Fall, wie in Abbildung 7.11 aufgezeigt wird. Durch zusätzliches Kantenentfernen wird nur mehr Zeit in Anspruch genommen, aber kein Mehrwert erzielt. Im Gegensatz dazu lässt sich das Sortieren nach Anzahl der Kanten noch deutlich verbessern, indem unnötige Kanten frühzeitig entfernt werden.

Aber erst im Tetraedisierungs-Schritt wird die Verbesserung durch die Heuristiken sehr deutlich. In Abbildung 7.12 sieht man dabei den Einfluss der Anzahl der Polyeder auf

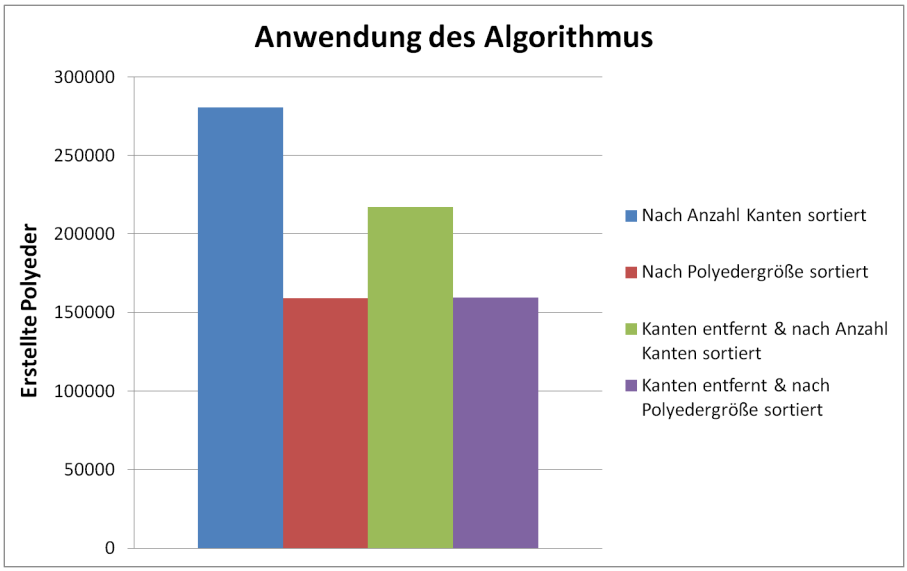

Abbildung 7.10: Auswirkung der Heuristiken auf die Algorithmus-Ausführung (Anzahl Polyeder)

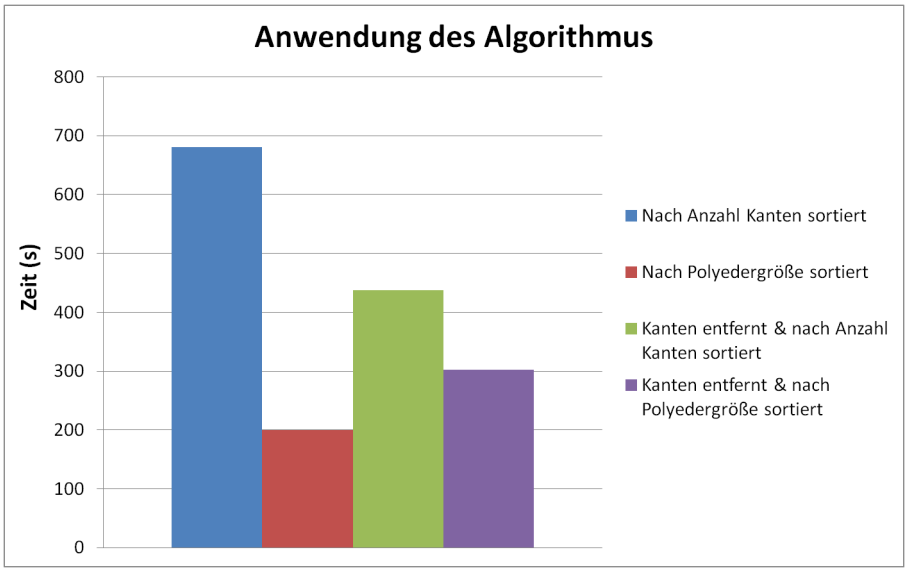

Abbildung 7.11: Auswirkung der Heuristiken auf die Algorithmus-Ausführung (Zeit)

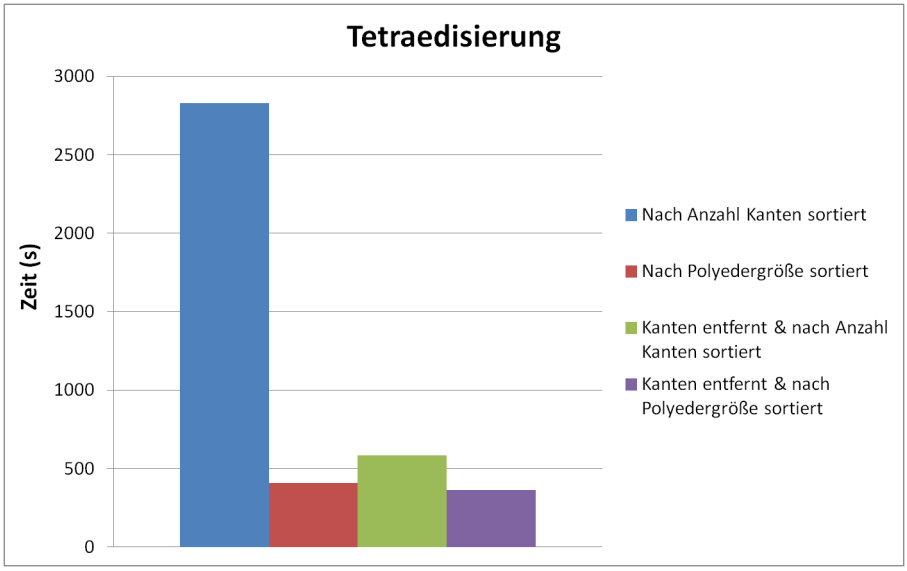

Abbildung 7.12: Auswirkung der Heuristiken auf die Dauer der Tetraedisierung

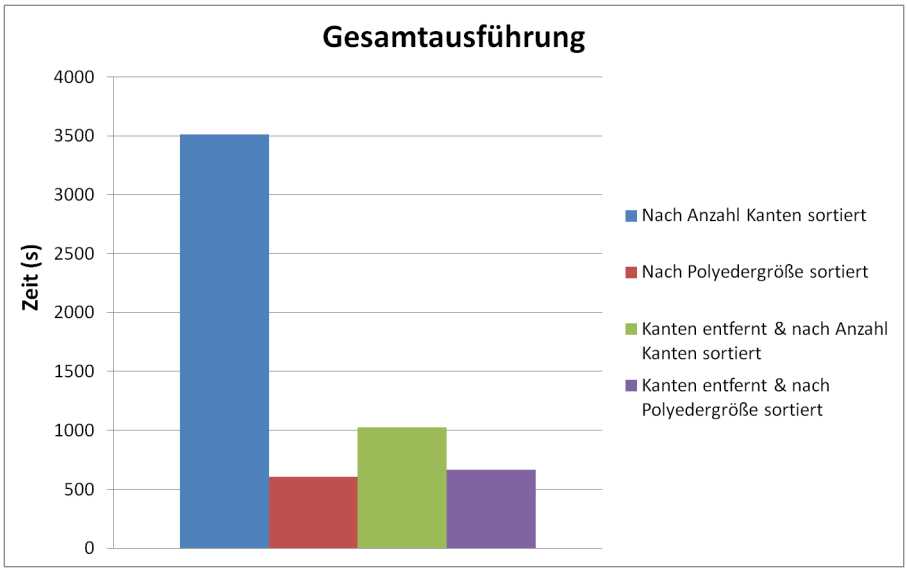

Abbildung 7.13: Auswirkung der Heuristiken auf die Gesamtdauer

die Dauer der Tetraedisierung. Da dieser Schritt die längste Zeit im Gesamtdurchlauf in Anspruch nimmt, Vergleich dazu Abbildung 7.13, wird bewusst, dass nur nach Anzahl der Kanten zu sortieren nicht effizient genug ist.

Weiterhin wird durch das berechnete Volumen in Abbildung 7.14 gezeigt, dass es großen Einfluss auf das Ergebnis gibt, ob kleine Polyeder entfernt werden oder diese verwendet

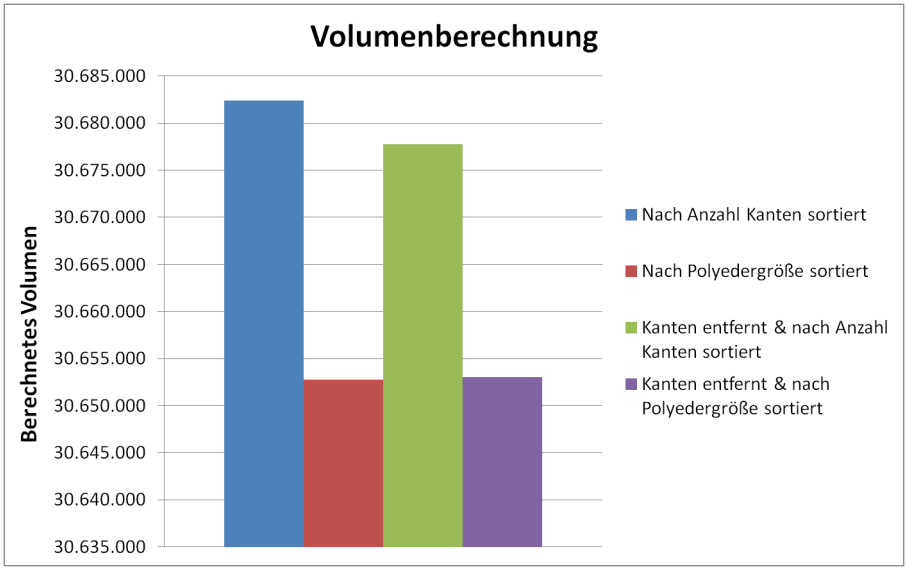

Abbildung 7.14: Auswirkung der Heuristiken auf das berechnete Volumen

werden um größere Polyeder aufzuteilen. Ersteres ist öfter der Fall, wenn nach Größe der Polyeder sortiert wird und nicht nach Anzahl der Kanten. Da jedoch nicht bekannt ist, welcher der erzielten Werte näher am tatsächlichen Wert liegt, kann hier wiederum nur Auskunft über die Effizienz, jedoch nicht über die Effektivität gegeben werden. Nur unter Berücksichtigung der Effizienz ist zumindest deutlich, dass die Ausführung mit dem Sortieren nach der Polyedergröße am sinnvollsten ist.

8 Zusammenfassung und Ausblick

In dieser Bachelor-Arbeit wurde ein neuer Ansatz zur Berechnung des Volumens sich überschneidender konvexer Polyeder vorgestellt. Dazu wurde ein neuer Algorithmus für BSP-Bäume eingeführt. Durch diesen ist es möglich zwei im vorangegangenem Schritt zusammengefügte Polyeder so aufzuteilen, dass diese neu erstellten Polyeder nicht den selben Raum abdecken und ebenfalls konvex sind. Dieser Algorithmus funktioniert jedoch immer nur paarweise auf konvexen Polyedern.

Aus diesem Grund wurden Methoden zur Anwendung auf großen Datensätzen vorgestellt. Diese erlauben es durch geschickte Aufteilung des Raums und Verwalten der Polyeder in Unabhängigkeitsgraphen, dass die Algorithmen auch auf großen Datensätzen einsetzbar sind. Jedoch wird die Ausführung erst richtig effizient, wenn zudem Heuristiken eingesetzt werden. Durch diese Heuristiken lässt sich die Anzahl der erstellten Polyeder weiter senken. Somit wird die Laufzeit stark reduziert und das Ergebnis genauer.

Bei diesem Ansatz ist der numerische Fehler sehr gering, da die meisten Aktionen auf den BSP-Bäumen ablaufen. Diese Aktionen sind dabei meist nur, dass Ebenen zu einem Polyeder hinzugefügt oder von ihm entfernt werden. Dabei kann kein Fehler entstehen. Lediglich beim Zusammenführen der Bäume ist ein kleiner Fehler möglich, da hier berechnete Eckpunkte des Polyeders in Relation zu einer Ebene überprüft werden müssen.

Größere numerische Fehler treten somit erst bei der Tetraedisierung der Polyeder auf. Jedoch trotz dieser Fehler ist das Ergebnis der Volumenberechnung sehr gut. Lediglich bei der Bestimmung der Porösität macht sich dieser Fehler stärker bemerkbar, lässt sich aber durch Ausführung mit verschiedenen Parametern handhaben.

Ausblick

Das Verfahren kann durch mehrere Modifikationen und durch das Hinzufügen weiterer Heuristiken verbessert werden. Eine dieser möglichen Heuristiken wird bereits in Kapitel 6, Abschnitt 6.4 beschrieben, wurde aber bisher noch nicht getestet.

Die wichtigste Verbesserung ist jedoch eine bessere Methode zur Berechnung des Volumens. Diese kann in diesem Verfahren problemlos die Tetraedisierung ersetzen. Dadurch kann die Qualität des Verfahrens weiter erhöht werden und auch die Berechnung der Porösität weiter verbessert werden.

Literaturverzeichnis

[LBFH] F. D. E. Latief, B. Biswal, U. Fauzi, R. Hilfer. Continuum reconstruction of the pore scale microstructure for Fontainebleau sandstone. *Physica A: Statistical Mechanics and its Applications*, 389(8):1607–1618, 2010. doi:10.1016/j.physa.2009.12.006. URL http://www.sciencedirect.com/science/article/pii/S0378437109010024. (Zitiert auf den Seiten 7, 11, 12 und 13)

[HZL] R. Hilfer, T. Zauner, A. Lemmer. Worldwide largest threedimensional strongly correlated microstructure. *Institute for Computational Physics*, 2011. URL http://www.icp.uni-stuttgart.de/microct/info.php. (Zitiert auf den Seiten 7, 11, 12 und 29)

[CS] J. L. D. Comba, C. T. Silva. *Automatic Convexification of Space using BSP-trees*.

[NAT] B. Naylor, J. Amanatides, W. Thibault. Merging BSP trees yields polyhedral set operations. *SIGGRAPH Comput. Graph.*, 24(4):115–124, 1990. doi:10.1145/97880.97892. URL http://doi.acm.org/10.1145/97880.97892. (Zitiert auf den Seiten 19, 22 und 23)

Alle URLs wurden zuletzt am 23. 05. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift