Johannes Gutenberg-Universität Mainz
Fachbereich 08
Institut für Informatik
Arbeitsgruppe Computational Geometry

Arbeit zur Erlangung des akademischen Grades Bachelor of Science

GPU-beschleunigte Schnittvolumenberechnung auf Dreiecksnetzen

Justus Henneberg

1. Gutachter Prof. Dr. Elmar Schömer
Institut für Informatik
Johannes Gutenberg-Universität Mainz

2. Gutachter Dr. Christian Hundt
Institut für Informatik
Johannes Gutenberg-Universität Mainz

Betreuer Prof. Dr. Elmar Schömer

11. 09. 2018

Justus Henneberg

GPU-beschleunigte Schnittvolumenberechnung auf Dreiecksnetzen

Arbeit zur Erlangung des akademischen Grades Bachelor of Science, 11. 09. 2018

Gutachter: Prof. Dr. Elmar Schömer und Dr. Christian Hundt

Betreuer: Prof. Dr. Elmar Schömer

Johannes Gutenberg-Universität Mainz

Arbeitsgruppe Computational Geometry

Institut für Informatik

Fachbereich 08

Staudingerweg 9

55128 Mainz

Abstract

Mit einer speziellen Formel ist es möglich, das Schnittvolumen zweier Dreiecksnetze durch die unabhängige Auswertung einzelner Dreiecksschnitte zu berechnen. Wir beweisen die Gültigkeit dieser Formel und stellen danach zwei unterschiedliche Algorithmen vor, die potenziell für die Ausführung auf einem Grafikprozessor geeignet sind. Einer davon erlaubt die Anwendung einer räumlichen Datenstruktur, mit der Schnitte im Voraus ausgeschlossen werden können. Wir verwenden eine schwache Perturbation der Eingabedaten, um degenerierte Fälle mit hoher Wahrscheinlichkeit auszuschließen. Bereits mit einer schwach optimierten CUDA-Implementierung mit doppelter Fließkommagenauigkeit konnten wir Speedups von über 10 auf einer NVIDIA GeForce GTX 1080 mit einem relativen Fehler von 0.3% im Vergleich zur exakten Berechnung messen. Wir zeigen auch auf, wie die Berechnung weiter beschleunigt werden kann.

Abstract (English)

Using a particular formula one can compute the intersection volume of a pair of triangle meshes by independently evaluating intersecting triangles. After providing a formal proof for this formula, we propose two different intersection algorithms potentially suitable for running on a graphics processing unit, one of which allows early rejection tests by employing a spatial data structure. We slightly perturb the input vertices to eliminate degenerate cases. Using a weakly optimized CUDA implementation and double-precision floating point numbers, we were able to measure a speedup of more than 10 on an NVIDIA GeForce GTX 1080 with a relative error of 0.3% compared to the actual intersection volume. We also propose multiple ways for further speeding up the computation.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Inhalte dieser Arbeit 1

1.2 Danksagung 2

2 Verwandte Arbeiten **3**

3 Mathematische Fundierung **5**

3.1 Hinweise zur Notation 5

3.2 Grundkonzepte 5

3.2.1 Flächeninhalt eines Dreiecks 5

3.2.2 Zerlegung des Flächeninhalts 6

3.2.3 Zerlegung eines Polygons 7

3.2.4 Volumen eines Tetraeders 10

3.3 Formulierung der Volumenformel 10

3.3.1 Definition eines Polyeders 11

3.3.2 Repräsentation eines Polyeders 11

3.3.3 Volumen eines Polyeders 13

3.4 Beweis der Volumenformel 13

4 Umsetzung **21**

4.1 Eingabedatenformat 21

4.2 Volumenbestimmung eines einzelnen Dreiecksnetzes 22

4.3 Struktur des Schnittkörpers 23

4.4 Informationsgehalt eines Dreiecksschnitts 24

4.4.1 Berechnung der Schnittpunkte 24

4.4.2 Auswertung eines Dreiecksschnitts 25

4.5 Ablauf der Schnittvolumenberechnung 28

4.6 Bemerkungen zur Parallelisierung 33

4.6.1 Parallelisierungsansatz 33

4.6.2 Einschränkungen bei GPU-Parallelisierung 34

4.6.3 Umsetzung der GPU-Parallelisierung 35

4.6.4 Ungelöste Probleme 36

4.7 Degenerierte Fälle 37

4.8 Implementierungsdetails 39

4.9 Visualisierung der Szene 40
4.10 Ergebnisse 41
4.10.1 Korrektheit 41
4.10.2 Benchmarks 43

5 Zusammenfassung **47**
5.1 Beiträge dieser Arbeit 47
5.2 Ausblick 48

Anhang **51**

A Ausgelassene Beweise **51**
A.1 Besonderheiten von Skalarprodukt und Kreuzprodukt 51
A.2 Flächenbestimmung eines Dreiecks mit dem Kreuzprodukt 53
A.3 Flächeninhalt eines Polygons 54

Literatur **57**

1 Einführung

1.1 Inhalte dieser Arbeit

Die Berechnung des Schnittvolumens zweier Dreiecksnetze ist ein grundlegendes Verfahren in Anwendungen für Geometrie und 3D-Grafik. Abgesehen vom Nutzen der eigentlichen Schnittvolumenberechnung kann die Berechnung auch als Teilalgorithmus zur Lösung anderer Probleme eingesetzt werden. Mögliche Beispiele hierfür sind Packprobleme, bei denen der Algorithmus versucht, die Überlappung der zu packenden Gegenstände zu minimieren, oder in der Bauplanung, um bei Aushebungen eine schnelle Abschätzung des abzutragenden Volumens zu erhalten, etwa für den anschließenden Abtransport.

Die Schwierigkeit bei der Berechnung besteht in der korrekten Handhabung der Eingabenetze, denn diese können bis zu mehrere Millionen Seitenflächen besitzen. Ohne zusätzliche Einschränkungen muss hier davon ausgegangen werden, dass sich jedes Dreieck des einen Netzes mit jedem Dreieck des anderen Netzes schneiden kann. Für einen effizienten Algorithmus ist es also sowohl notwendig, möglichst viele Schnitte im Voraus auszuschließen, als auch die bestehenden Schnitte zwischen Dreiecken möglichst schnell auszuwerten und dabei auch für geometrische Sonderfälle korreke Ergebnisse zu liefern. Muss zusätzlich der Schnittkörper zur Volumenberechnung bestimmt werden, fällt außerdem Rechenaufwand für die Retriangulierung an, da der Schnittkörper die Flächen der Eingabenetze möglicherweise nur zu Teilen übernimmt.

Mit einer speziellen Volumenformel nach W. Randolph Franklin [9, 10] ist es möglich, das Volumen eines beliebigen Polyeders zu bestimmen, ohne den Schnittkörper explizit konstruieren zu müssen. Die allgemeine Gültigkeit der Formel wird jedoch an keiner Stelle formal bewiesen.

In dieser Arbeit erläutern wir die geometrischen Zusammenhänge, auf denen die Gültigkeit der Formel beruht, und führen einen formalen Beweis der Formel durch. Wir zeigen weiterhin, wie man zunächst sequenziell auf Basis dieser Formel das Schnittvolumen zweier allgemeiner Polyeder bestimmen kann, indem wir darstellen, welche Information über das Schnittvolumen aus dem Schnitt zweier einzelner Dreiecke extrahiert werden kann. Wir verbessern den Algorithmus anschließend so,

dass er die Verwendung räumlicher Datenstrukturen erlaubt. Zusätzlich behandeln wir, welche Sonderfälle in den Eingabedaten für unser Verfahren problematisch sind und wie wir diese umgehen können.

Seit einigen Jahren entwickelt sich das Konzept der *General Purpose Computation on Graphics Processing Units (GPGPU)* stetig voran. Hierbei wird versucht, nicht notwendigerweise grafikbezogene Berechnungen zunehmend auf einen oder mehrere Grafikprozessoren auszulagern, anstatt diese ausschließlich auf dem Hauptprozessor auszuführen. Damit eine Berechnung vom massiven Parallelismus moderner Grafikprozessoren profitieren kann, muss diese bestimmte Voraussetzungen an Datenabhängigkeiten und benötigte Speicherzugriffe erfüllen.

Wir werden in dieser Arbeit darauf eingehen, wie sich das zuvor beschriebene Berechnungsverfahren so formulieren lässt, dass es sich für die Berechnung auf einem Grafikprozessor eignet. Mit einer einfachen Implementierung dieses Verfahrens erreichen wir Speedups von ungefähr 13 gegenüber der sequenziellen Variante, wir gehen aber auch auf Optimierungen ein, mit denen das Verfahren weiter beschleunigt werden kann.

Diese Arbeit ist wie folgt aufgebaut:

Kapitel 2 fasst die für diese Arbeit relevanten Ergebnisse bisheriger Veröffentlichungen zur Berechnung von Schnitten zwischen Dreiecksnetzen zusammen.
In **Kapitel 3** führen wir die Formel ein, die als Grundlage der Schnittvolumenberechnung dient, und beweisen diese.
Kapitel 4 beschreibt die beiden Algorithmen zur Schnittvolumenberechnung, deren Parallelisierungsansatz und zusätzliche Details der Implementierung.
Kapitel 5 fasst die Kernaussagen und -beiträge dieser Arbeit zusammen und liefert einen Ausblick über mögliche Erweiterungen.
Im **Anhang** finden sich die Beweise zu den Sätzen, die in den ersten Kapiteln übergangen werden.

1.2 Danksagung

Ich möchte mich dieser Stelle bei Herrn Prof. Dr. Schömer für den Vorschlag dieses interessanten Themas und die weiterführenden Diskussionen bedanken. Weiterhin möchte ich der Arbeitsgruppe *High Performance Computing* der Universität Mainz danken, durch die ich erst das Potential von GPU-Programmierung zu schätzen gelernt habe.

2 Verwandte Arbeiten

Diese Arbeit basiert auf einer Veröffentlichung von Franklin und Kankanhalli[10], in der die Volumenformel eingesetzt wird, um das Schnittvolumen zwischen Paaren von Tetraedern zu berechnen. Franklin und Kankanhalli verwenden ein 2D-Gitter und berechnen nur Schnitte zwischen Flächen, die in dieselbe Gitterzelle projiziert werden.

Aktuelle Arbeiten gehen nicht mehr auf die Berechnung des Schnittvolumens ein, sondern beschäftigen sich mit der exakten Konstruktion des Schnittkörpers. Franklin und Magalhães[8, 18, 20] verwenden hierzu rationale Arithmetik und setzen eine Technik namens *Simulation of Simplicity*[4] gegen degenerierte Fälle ein, die in einer späteren Veröffentlichung[19] weiter verbessert wird. Der Schnittalgorithmus berechnet nur Schnitte zwischen Dreiecken, die sich eine Gitterzelle in einem 3D-Gitter teilen[7] und wird mit OpenMP[28] über die Gitterzellen parallelisiert.

Mei und Tipper[22] suchen für die Schnittberechnung nach geschlossenen Ringen aus Schnittkanten, um zusätzlich zur Berechnung des Schnittes weitere boolsche Operationen vornehmen zu können, etwa Vereinigung und Differenz zweier Dreiecksnetze. Der vorgestellte Algorithmus verwendet eine Octree-Datenstruktur zur Verwaltung der räumlichen Daten.

Bestehende Implementierungen von Schnittalgorithmen finden sich außerdem in Programmbibliotheken für 3D-Geometrie, beispielsweise CGAL[29] oder libigl[15]. Diese beiden Bibliotheken verwenden selbst keine Parallelisierung zur Beschleunigung der Berechnung. Zusätzlich dazu existieren nicht-quelloffene Implementierungen in Software für 3D-Modellierung wie AutoCAD[2] oder dem von uns zur Verifizierung der Ergebnisse eingesetzten Rhino3D[30].

3 Mathematische Fundierung

3.1 Hinweise zur Notation

Wir verwenden im Folgenden anstatt der in der Computergrafik gebräuchlichen Notation $\mathbf{u}^T\mathbf{v}$ die Notation $\langle \mathbf{u}, \mathbf{v} \rangle_2$ für das Standardskalarprodukt zweier Vektoren $\mathbf{u}$ und $\mathbf{v}$, um eine stärkere visuelle Trennung von skalaren und vektoriellen Größen zu erreichen. Wir schreiben analog $\|\mathbf{u}\|_2 = \sqrt{\langle \mathbf{u}, \mathbf{u} \rangle_2}$ für die euklidische Norm eines Vektors $\mathbf{u}$ und verwenden nur für Skalare Betragsstriche.

Wir bezeichnen mit $\mathrm{ar}(K)$ die Oberfläche eines dreidimensionalen Körpers K und mit $\mathrm{vol}(K)$ das Volumen von K. Wir verwenden die Notation $\mathrm{ar}(P)$ auch für die Fläche eines zweidimensionalen Polygons P.

3.2 Grundkonzepte

Wir nutzen diesen Abschnitt, um die Konzepte einzuführen, die für das Verständnis des eigentlichen Beweises wichtig sind. Dazu betrachten wir zunächst ein einzelnes Dreieck im euklidischen Raum.

3.2.1 Flächeninhalt eines Dreiecks

Der Flächeninhalt eines Dreiecks T lässt sich auf verschiedene Arten bestimmen. Hierzu wird häufig zunächst geometrisch argumentiert, dass der Flächeninhalt von T gerade der Hälfte der Fläche des Rechtecks mit der Länge der Grundseite g und der Höhe h des Dreiecks als Seitenlängen entspricht, oft geschrieben als

$$\mathrm{ar}(T) = \frac{1}{2}gh.$$

In grafischen Anwendungen sind selten g und h bekannt, sondern vielmehr die Koordinaten $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ der Eckpunkte. In diesem Fall können wir die Berechnung

von g und h umgehen, indem wir stattdessen den folgenden Zusammenhang nutzen:

$$\mathrm{ar}(T) = \frac{1}{2}\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|_2. \tag{3.2.1}$$

Einen Beweis für dessen Gültigkeit findet sich im Anhang. Die Gleichheit basiert darauf, dass die euklidische Norm des Kreuzprodukts zweier Vektoren $\mathbf{u} = \mathbf{b}-\mathbf{a}$ und $\mathbf{v} = \mathbf{c}-\mathbf{a}$ gerade der Fläche des von $\mathbf{u}$ und $\mathbf{v}$ aufgespannten Parallelogramms entspricht. Dieses Parallelogramm kann in exakt zweimal das von $\mathbf{u}$ und $\mathbf{v}$ aufgespannte Dreieck zerlegt werden.

Wir sehen im nächsten Abschnitt, dass wir auch das Dreieck T weiter zerlegen können.

3.2.2 Zerlegung des Flächeninhalts

Mit der Linearitätseigenschaft[6, S. 282] können wir das Kreuzprodukt aus Gleichung 3.2.1 weiter umformen, es gilt

$$\begin{aligned}(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a}) &= \mathbf{b}\times\mathbf{c} - \mathbf{b}\times\mathbf{a} - \mathbf{a}\times\mathbf{c} + \mathbf{a}\times\mathbf{a} \\ &= \mathbf{b}\times\mathbf{c} - \mathbf{b}\times\mathbf{a} - \mathbf{a}\times\mathbf{c} \\ &= \mathbf{a}\times\mathbf{b} + \mathbf{b}\times\mathbf{c} + \mathbf{c}\times\mathbf{a}\end{aligned} \tag{3.2.2}$$

und damit folgt

$$\begin{aligned}\mathrm{ar}(T) &= \frac{1}{2}\|\mathbf{a}\times\mathbf{b} + \mathbf{b}\times\mathbf{c} + \mathbf{c}\times\mathbf{a}\|_2 \\ &= \|\frac{1}{2}(\mathbf{a}\times\mathbf{b}) + \frac{1}{2}(\mathbf{b}\times\mathbf{c}) + \frac{1}{2}(\mathbf{c}\times\mathbf{a})\|_2.\end{aligned}$$

Was bedeutet dieser Zusammenhang geometrisch? Hierzu betrachten wir den Fall, dass T vollständig in der xy-Ebene liegt. Da der Kreuzproduktvektor immer senkrecht auf seinen Operanden steht (siehe Anhang), ist in diesem Fall nur die z-Komponente der drei Kreuzprodukten von Null verschieden und wir können die Flächenberechnung vereinfachen zu

$$\begin{aligned}\mathrm{ar}(T) &= \left|\frac{1}{2}[(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})]_z\right| \\ &= \left|\frac{1}{2}[\mathbf{a}\times\mathbf{b}]_z + \frac{1}{2}[\mathbf{b}\times\mathbf{c}]_z + \frac{1}{2}[\mathbf{c}\times\mathbf{a}]_z\right|.\end{aligned}$$

Mit der Notation $[\cdot]_z$ bezeichnen wir die z-Komponente eines Vektors. Zunächst stellen wir fest, dass diese Formel zunächst den vorzeichenbehafteten Flächenin-

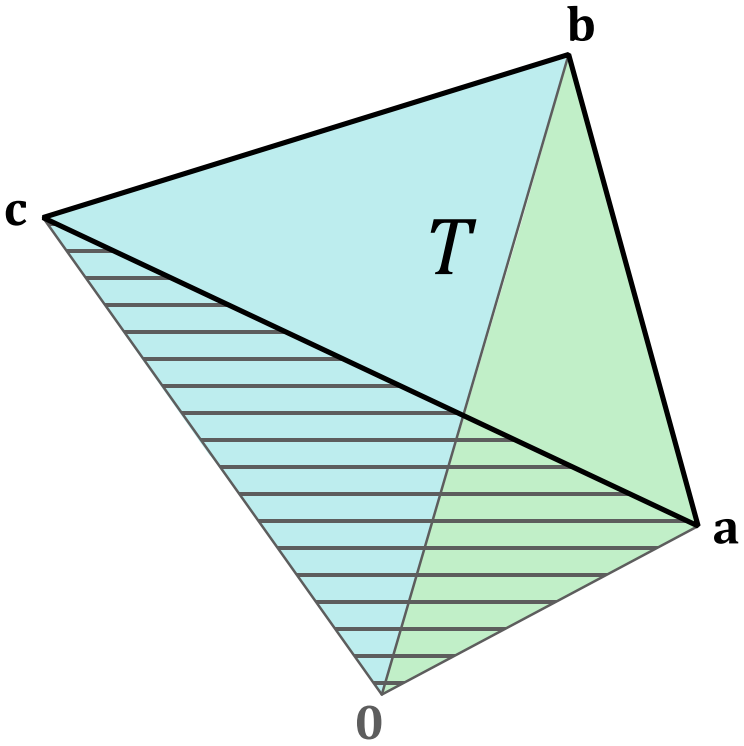


Abb. 3.1.: Zerlegung eines Dreiecks T mit den Eckpunkten a, b und c. Die Fläche des schraffierten Dreiecks eliminiert den Flächeninhalt, den das blaue und grüne Dreieck außerhalb von T abdecken, durch inverses Vorzeichen.

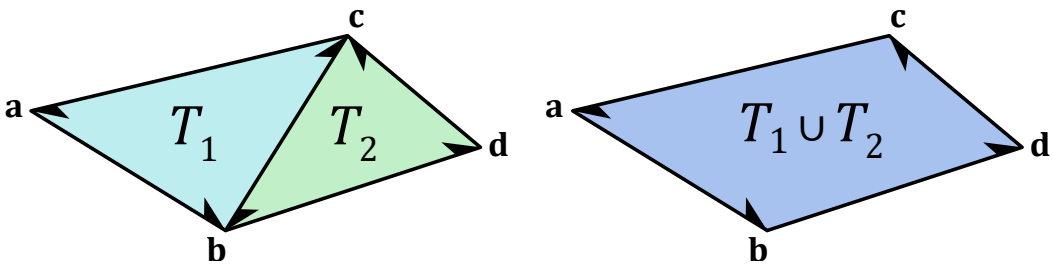
halt liefert und der Betrag hier lediglich sicherstellt, dass das Ergebnis positiv ist. Gleichzeitig gelten aber auch

$$\begin{aligned}\frac{1}{2}[\mathbf{a}\times\mathbf{b}]_z &= \frac{1}{2}[(\mathbf{a}-\mathbf{0})\times(\mathbf{b}-\mathbf{0})]_z,\\ \frac{1}{2}[\mathbf{b}\times\mathbf{c}]_z &= \frac{1}{2}[(\mathbf{b}-\mathbf{0})\times(\mathbf{c}-\mathbf{0})]_z,\\ \frac{1}{2}[\mathbf{c}\times\mathbf{a}]_z &= \frac{1}{2}[(\mathbf{c}-\mathbf{0})\times(\mathbf{a}-\mathbf{0})]_z,\end{aligned}$$

wir haben also die Berechnung des Flächeninhalts von T auf die Berechnung des Flächeninhalts dreier weiterer Dreiecke zurückgeführt, die jeweils einen Eckpunkt im Koordinatenursprung besitzen. Das Vorzeichen sorgt hierbei dafür, dass sich die Fläche außerhalb des Dreiecks zu Null summiert. Dieser Zusammenhang wird in Abbildung 3.1 visualisiert.

3.2.3 Zerlegung eines Polygons

Wir wollen nun zeigen, dass sich dieser Zusammenhang auf beliebige Polygone verallgemeinern lässt. Dabei ist es gleichgültig, ob das Polygon konvex ist. Im letzten Abschnitt haben wir gesehen, dass Flächeninhalte vorzeichenbehaftet sein können.



(a) T_1 und T_2 mit gerichteten Kanten

(b) Das aus der Vereinigung von T_1 und T_2 entstehende Viereck mit gerichteten Kanten

Abb. 3.2.: Visualisierung der beteiligten Kanten bei der Berechnung von $\mathrm{ar}(T_1 \cup T_2)$. Die Kante zwischen $\mathbf{b}$ und $\mathbf{c}$ hat bei der Berechnung keinen Einfluss auf das Ergebnis.

Das Vorzeichen hängt hierbei von der Reihenfolge der Operanden des Kreuzprodukts ab. Es ist folglich sinnvoll, sich auf eine Reihenfolge festzulegen.

Wir nennen im Folgenden ein Polygon *positiv orientiert*, wenn seine Eckpunkte so angeordnet sind, dass Kanten zwischen aufeinanderfolgenden Eckpunkten gegen den Uhrzeigersinn gerichtet sind. Für Polygone im dreidimensionalen Raum müssen wir eine Blickrichtung fixieren, damit die Orientierung konsistent definiert ist. Für den Fall, dass das Polygon einen dreidimensionalen Körper begrenzt, legen wir die Blickrichtung so fest, dass wir von außen auf den Körper schauen.

Mit dieser Definition gilt für ein positiv orientiertes Dreieck T mit Eckpunkten $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ in der xy-Ebene sogar

$$\begin{aligned}\mathrm{ar}(T) &= \frac{1}{2}[(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})]_z \\ &= \frac{1}{2}[\mathbf{a} \times \mathbf{b}]_z + \frac{1}{2}[\mathbf{b} \times \mathbf{c}]_z + \frac{1}{2}[\mathbf{c} \times \mathbf{a}]_z,\end{aligned}$$

der Betrag fällt also weg.

Als Nächstes betrachten wir zwei nebeneinander angeordnete positiv orientierte Dreiecke T_1, T_2 mit Eckpunkten $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ und $\mathbf{d}$, $\mathbf{c}$, $\mathbf{b}$ in der xy-Ebene. T_1 und T_2 teilen sich die Kante zwischen $\mathbf{b}$ und $\mathbf{c}$. Dann gilt für ihre gemeinsame Fläche:

$$\begin{aligned}\mathrm{ar}(T_1 \cup T_2) &= \mathrm{ar}(T_1) + \mathrm{ar}(T_2) \\ &= \frac{1}{2}[(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})]_z + \frac{1}{2}[(\mathbf{c}-\mathbf{d}) \times (\mathbf{b}-\mathbf{d})]_z \\ &= \frac{1}{2}\Big([\mathbf{a} \times \mathbf{b}]_z + [\mathbf{b} \times \mathbf{c}]_z + [\mathbf{c} \times \mathbf{a}]_z\Big) \\ &\quad + \frac{1}{2}\Big([\mathbf{d} \times \mathbf{c}]_z + [\mathbf{c} \times \mathbf{b}]_z + [\mathbf{b} \times \mathbf{d}]_z\Big) \\ &= \frac{1}{2}\Big([\mathbf{a} \times \mathbf{b}]_z + [\mathbf{b} \times \mathbf{d}]_z + [\mathbf{d} \times \mathbf{c}]_z + [\mathbf{c} \times \mathbf{a}]_z + [\mathbf{b} \times \mathbf{c}]_z - [\mathbf{b} \times \mathbf{c}]_z\Big)\end{aligned}$$

$$= \frac{1}{2}\Big([\mathbf{a} \times \mathbf{b}]_z + [\mathbf{b} \times \mathbf{d}]_z + [\mathbf{d} \times \mathbf{c}]_z + [\mathbf{c} \times \mathbf{a}]_z\Big).$$

Die Terme, die zur Kante zwischen $\mathbf{b}$ und $\mathbf{c}$ gehören, löschen sich aus, es verbleiben nur solche entlang der Außenseiten. Dasselbe Argument gilt für allgemeine Polygone.

Wir repräsentieren ein positiv orientiertes Polygon über die Menge seiner gerichteten Kanten. Diese Repräsentation erlaubt es, dass Polygone aus mehreren Zusammenhangskomponenten bestehen können und Löcher enthalten dürfen, solange diese negativ orientiert sind. Für die Flächenberechnung müssen wir nur die Kanten einzeln betrachten. Analog dazu werden wir für die Volumenberechnung später die Seitenflächen eines Polyeders nur einzeln betrachten.

Wir können jedes positiv orientierte Polygon P mit seiner Menge gerichteter Kanten E_P in positiv orientierte Dreiecke $T_1, \ldots, T_M$ mit Eckpunkten $\mathbf{a}_i$, $\mathbf{b}_i$ und $\mathbf{c}_i$ und Kantenmenge E_{T_i} zerlegen. In der Praxis können hierfür Sweep-Verfahren[3] oder *ear clipping*[12] eingesetzt werden. Wir gehen in dieser Arbeit nicht weiter darauf ein.

Liegt P in der xy-Ebene, gilt schließlich

$$\begin{aligned}
\mathrm{ar}(P) &= \sum_{i=1}^{M} \mathrm{ar}(T_i) \\
&= \sum_{i=1}^{M} \frac{1}{2}[(\mathbf{b}_i - \mathbf{a}_i) \times (\mathbf{c}_i - \mathbf{a}_i)]_z \\
&= \frac{1}{2} \sum_{i=1}^{M} \Big([\mathbf{a}_i \times \mathbf{b}_i]_z + [\mathbf{b}_i \times \mathbf{c}_i]_z + [\mathbf{c}_i \times \mathbf{a}_i]_z\Big) \\
&= \frac{1}{2} \sum_{i=1}^{M} \sum_{(\mathbf{p},\mathbf{q}) \in E_{T_i}} [\mathbf{p} \times \mathbf{q}]_z \\
&= \frac{1}{2} \sum_{(\mathbf{p},\mathbf{q}) \in E_P} [\mathbf{p} \times \mathbf{q}]_z,
\end{aligned}$$

Mit der Beobachtung, dass Flächeninhalt invariant unter Translation und Rotation ist, können wir jedes Polygon unter Anwendung einer Rotation und einer Translation in die xy-Ebene legen. Im Anhang zeigen wir formal, wie sich Rotation und Translation in der Formel eliminieren, und wir erhalten

$$\mathrm{ar}(P) = \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{p} \times \mathbf{q}) \Big\|_2$$

für die Fläche von P.

Analog zur Zerlegung des Flächeninhalts werden wir in Abschnitt 3.4 ein Polyeder in Tetraeder zerlegen, um sein Volumen zu bestimmen. Dazu müssen wir aber zunächst das Volumen eines einzelnen Tetraeders berechnen können.

3.2.4 Volumen eines Tetraeders

In Abschnitt 3.2.1 haben wir den Flächeninhalt eines Dreiecks über den Flächeninhalt eines von zwei Vektoren aufgespannten Parallelogramms bestimmt. Analog dazu spannen 3 Vektoren das dreidimensionale Äquivalent eines Parallelogramms auf, ein sogenanntes Parallelepiped (auch *Spat* genannt). Für das Volumen eines von den Vektoren $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$ aufgespannten Parallelepipeds P gilt

$$\operatorname{vol}(P) = \langle \mathbf{u},\, \mathbf{v} \times \mathbf{w} \rangle_2 = \det \begin{bmatrix} | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} \\ | & | & | \end{bmatrix}.$$

Jedes Parallelepiped kann als das Bild einer linearen Transformation des Einheitswürfels angesehen werden. Die Gültigkeit der Formel lässt sich dann mit der mehrdimensionalen Integraltransformationsformel zeigen. Wir übergehen den Beweis an dieser Stelle. Den Zusammenhang

$$\langle \mathbf{u},\, \mathbf{v} \times \mathbf{w} \rangle_2 = \det \begin{bmatrix} | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} \\ | & | & | \end{bmatrix}$$

erhalten wir durch explizites Ausrechnen beider Seiten.

Das von $\mathbf{u}$, $\mathbf{v}$ und $\mathbf{w}$ aufgespannte Tetraeder T hat gerade ein Sechstel des Volumens von P. Es gilt also insgesamt

$$\operatorname{vol}(T) = \frac{1}{6} \langle \mathbf{u},\, \mathbf{v} \times \mathbf{w} \rangle_2.$$

Der Term $\langle \mathbf{u},\, \mathbf{v} \times \mathbf{w} \rangle_2$ wird aufrund seiner geometrischen Interpretation auch als *Spatprodukt* bezeichnet.

Wir können uns nun der eigentlichen Aussage des Kapitels zuwenden.

3.3 Formulierung der Volumenformel

Wir beginnen mit der Definition der dreidimensionalen Variante eines Polygons.

3.3.1 Definition eines Polyeders

Wir definieren ein Polyeder als eine Teilmenge des dreidimensionalen Raums, die ausschließlich von geraden Flächen begrenzt wird. Für unsere Zwecke muss diese Menge nicht zusammenhängen und darf Tunnel oder Aushöhlungen enthalten. Wichtig ist nur, dass die Begrenzungsflächen selbst keine Löcher aufweisen.

3.3.2 Repräsentation eines Polyeders

Üblicherweise genügt zur Repräsentation eines Polyeders eine Liste der Eckpunkte für jede Begrenzungsfläche. Das setzt allerdings voraus, dass wir diese Informationen über die Flächen des Polyeders haben. Bei der Berechnung des Schnittes zweier Polyeder liegt diese Information zunächst nicht vor. Für unsere Formel nutzen wir deshalb eine andere Darstellung, die nur sehr lokal die Struktur des Polyeders beschreibt.

Zur Berechnung des Volumens benötigen wir
- für jede Fläche F des Polyeders,
- für jeden Begrenzungspunkt $\mathbf{p} \in \mathbb{R}^3$ von F,
- für jeden Begrenzungspunkt $\mathbf{q} \in \mathbb{R}^3$ von F, der adjazent zu $\mathbf{p}$ ist,

das Quadrupel

$$(\mathbf{p}, \mathbf{t}_{pq}, \mathbf{u}_{pq}, \mathbf{n})$$

wobei gilt:

$\mathbf{n}$ ist der Normalenvektor der Fläche F und zeigt außerhalb des Polyeders. Wir fordern zusätzlich $\|\mathbf{n}\|_2 = 1$.

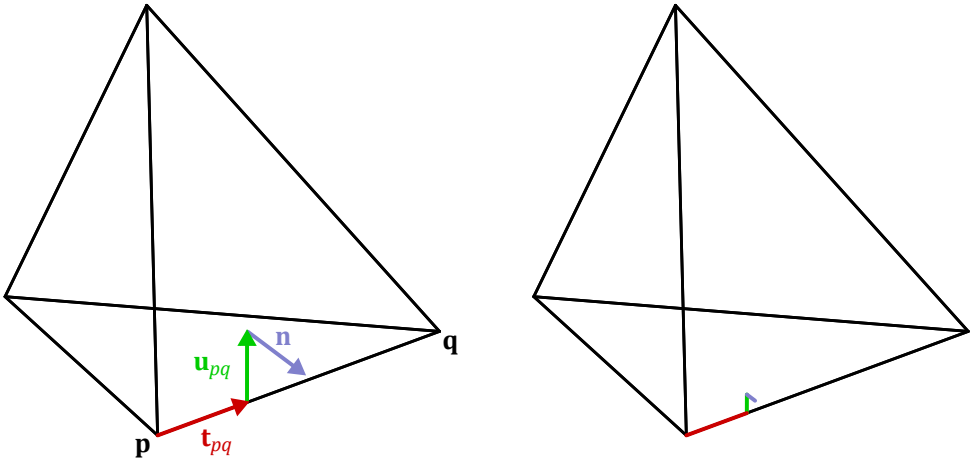
$\mathbf{t}_{pq}$ ist der normalisierte Vektor, der von $\mathbf{p}$ aus in Richtung $\mathbf{q}$ zeigt. Es gilt stets $\mathbf{t}_{pq} = -\mathbf{t}_{qp}$.

$\mathbf{u}_{pq}$ ist der normalisierte Vektor, der senkrecht auf $\mathbf{t}_{pq}$ steht, in der Ebene F liegt und ins Innere von F zeigt. Da die Innenseite von F unabhängig von der Reihenfolge der Begrenzungspunkte definiert ist, gilt hier stets $\mathbf{u}_{pq} = \mathbf{u}_{qp}$.

Wir bezeichnen im Folgenden die Menge aller solchen Quadrupel mit $\mathcal{Q}$. Die Menge $\mathcal{Q}$ ist für ein gegebenes Polyeder immer eindeutig. Abbildung 3.3 visualisiert die Quadrupel in $\mathcal{Q}$ für ein Tetraeder.

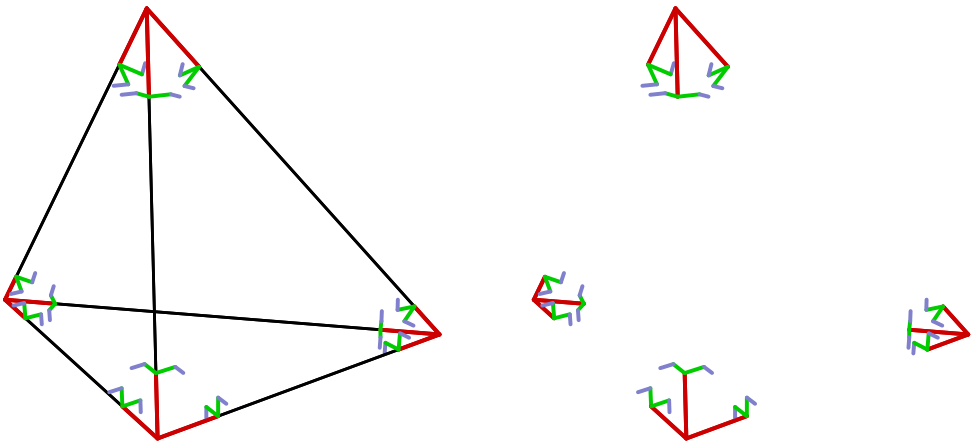
Zu gegebenen $\mathbf{p}, \mathbf{q}$ können wir immer $\mathbf{t}_{pq}$ mittels

$$\mathbf{t}_{pq} = \frac{\mathbf{q} - \mathbf{p}}{\|\mathbf{q} - \mathbf{p}\|_2} \tag{3.3.1}$$



(a) Tetraeder-Wireframe mit Visualisierung des Quadrupels $(\mathbf{p}, \mathbf{t}_{pq}, \mathbf{u}_{pq}, \mathbf{n}) \in \mathcal{Q}$

(b) Vereinfachte Darstellung von 3.3a



(c) Darstellung aller Quadrupel aus $\mathcal{Q}$ für dieses Tetraeder

(d) 3.3c nach Weglassen des Wireframes

Abb. 3.3.: Visualisierung der Quadrupeldarstellung eines Tetraeders. Benachbarte Flächen teilen sich für jeden Eckpunkt den Vektor auf der Begrenzungskante, deshalb wird dieser jeweils nur einmal dargestellt. 3.3d demonstriert, dass die Quadrupeldarstellung nur Informationen über die Richtung der Kanten, nicht aber deren Länge oder zweiten Endpunkt enthält.

berechnen. Kennen wir zusätzlich zu $\mathbf{p}$ und $\mathbf{q}$ einen weiteren Begrenzungspunkt $\mathbf{r}$ von F, so können wir auch den Vektor $\mathbf{n}$ bis auf sein Vorzeichen bestimmen durch

$$\mathbf{n} = \pm \frac{(\mathbf{r} - \mathbf{q}) \times (\mathbf{p} - \mathbf{q})}{\|(\mathbf{r} - \mathbf{q}) \times (\mathbf{p} - \mathbf{q})\|_2} \tag{3.3.2}$$

Sind die Begrenzungspunkte $\mathbf{p}, \mathbf{q}, \mathbf{r}$ positiv orientiert, so ist das Vorzeichen gerade positiv. In diesem Fall kann man $\mathbf{u}_{pq}$ über den Zusammenhang

$$\mathbf{u}_{pq} = \mathbf{n} \times \mathbf{t}_{pq}. \tag{3.3.3}$$

bestimmen.

3.3.3 Volumen eines Polyeders

Wir können nun den zentralen Satz dieses Kapitels formulieren:

Satz 3.3.1 (Volumenformel nach Franklin)**.** *Zu einem beliebigen Polyeder P sei $\mathcal{Q}$ wie oben beschrieben definiert. Dann gilt*

$$\mathrm{vol}(P) = \frac{1}{6} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} \langle \mathbf{p},\, \mathbf{t}\rangle_2 \,\langle \mathbf{p},\, \mathbf{u}\rangle_2 \,\langle \mathbf{p},\, \mathbf{n}\rangle_2.$$

Aus dem Beweis lässt sich zusätzlich folgende Aussage ableiten:

Satz 3.3.2. *Zu einem beliebigen Polyeder P sei wieder $\mathcal{Q}$ wie oben beschrieben definiert. Dann gilt*

$$\mathrm{ar}(P) = \frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} \langle \mathbf{p},\, \mathbf{t}\rangle_2 \,\langle \mathbf{p},\, \mathbf{u}\rangle_2$$

und

$$L = -\frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} \langle \mathbf{p},\, \mathbf{t}\rangle_2,$$

wobei L die Gesamtlänge aller Kanten von P beschreibt.

Den Beweis für beide Sätze führen wir im nächsten Abschnitt.

3.4 Beweis der Volumenformel

Für den Beweis gehen wir folgendermaßen vor: Wir zeigen zunächst, dass wir die Seitenflächen des Polyeders als Vereinigung von Dreiecken betrachten können, ohne die Gültigkeit der Formel einzuschränken. Wir beweisen im Anschluss, wie wir die Terme in der Formel so umformen können, dass bekannte Audrücke für geometrische Eigenschaften, etwa Länge und Flächeninhalt, entstehen. Zuletzt zeigen wir, dass ein Summand aus der klassischen Formel zur Volumenberechnung exakt mit der Summe mehrerer Summanden aus der Formel nach Franklin übereinstimmt.

Lemma 3.4.1. *Sei P ein Polyeder und F eine Seitenfläche von P. Seien $\mathbf{p}, \mathbf{q} \in \mathbb{R}^3$ zwei verschiedene Begrenzungspunkte von F, sodass die Strecke zwischen $\mathbf{p}$ und $\mathbf{q}$ vollständig in F liegt und $\mathbf{p}$ und $\mathbf{q}$ nicht bereits durch eine Kante von F verbunden sind. Dann können wir F durch Hinzunahme der Verbindungsstrecke zwischen $\mathbf{p}$ und $\mathbf{q}$ zerteilen, ohne dass sich der Wert der Formel aus Satz 3.3.1 verändert.*

Beweis. Durch die Hinzunahme der Strecke zwischen $\mathbf{p}$ und $\mathbf{q}$ kommen vier Quadrupel zu $\mathcal{Q}$ hinzu: Einerseits

$$(\mathbf{p}, \mathbf{t}_{pq}, \mathbf{u}_{pq}, \mathbf{n}) \text{ und } (\mathbf{q}, \mathbf{t}_{qp}, \mathbf{u}_{qp}, \mathbf{n})$$

für die angrenzende Fläche auf einer Seite der Strecke sowie

$$(\mathbf{p}, \mathbf{t}_{pq}, -\mathbf{u}_{pq}, \mathbf{n}) \text{ und } (\mathbf{q}, \mathbf{t}_{qp}, -\mathbf{u}_{qp}, \mathbf{n})$$

für die angrenzende Fläche auf der anderen Seite. Erzeugen wir aus diesen Quadrupeln die Summanden nach Satz 3.3.1, so ändert sich der Wert der Summe um

$$\begin{aligned}
&\langle \mathbf{p}, \mathbf{t}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{u}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{n}\rangle_2 + \langle \mathbf{q}, \mathbf{t}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{u}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{n}\rangle_2 \\
&\quad + \langle \mathbf{p}, \mathbf{t}_{pq}\rangle_2\langle \mathbf{p}, -\mathbf{u}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{n}\rangle_2 + \langle \mathbf{q}, \mathbf{t}_{qp}\rangle_2\langle \mathbf{q}, -\mathbf{u}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{n}\rangle_2 \\
&= \langle \mathbf{p}, \mathbf{t}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{u}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{n}\rangle_2 - \langle \mathbf{p}, \mathbf{t}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{u}_{pq}\rangle_2\langle \mathbf{p}, \mathbf{n}\rangle_2 \\
&\quad + \langle \mathbf{q}, \mathbf{t}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{u}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{n}\rangle_2 - \langle \mathbf{q}, \mathbf{t}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{u}_{qp}\rangle_2\langle \mathbf{q}, \mathbf{n}\rangle_2 \\
&= 0 + 0 \\
&= 0
\end{aligned}$$

und die Behauptung folgt. ∎

Da wir keine Konvexität unserer Polyeder fordern, ist mit Lemma 3.4.1 nicht gegeben, dass F durch die Zerlegung in zwei Flächen zerfällt. Dennoch können wir durch iteriertes Zerteilen eines gegebenen Polygons entlang einer Linie das Polygon vollständig in Dreiecke zerlegen.

Lemma 3.4.2. *Seien $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$ zwei verschiedene Punkte im Raum. Dann gilt:*

$$-\|\mathbf{b} - \mathbf{a}\|_2 = \langle \mathbf{a}, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b}, \mathbf{t}_{ba}\rangle_2$$

Beweis. Wir formen um:

$$\begin{aligned}
\langle \mathbf{a}, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b}, \mathbf{t}_{ba}\rangle_2 &= \langle \mathbf{a}, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b}, -\mathbf{t}_{ab}\rangle_2 \\
&= \langle \mathbf{a}, \mathbf{t}_{ab}\rangle_2 - \langle \mathbf{b}, \mathbf{t}_{ab}\rangle_2 && \text{(Linearität)} \\
&= \langle \mathbf{a} - \mathbf{b}, \mathbf{t}_{ab}\rangle_2 && \text{(Linearität)} \\
&= -\frac{\langle \mathbf{b} - \mathbf{a}, \mathbf{b} - \mathbf{a}\rangle_2}{\|\mathbf{b} - \mathbf{a}\|_2} && \text{(nach Definition von } \mathbf{t}_{ab}) \\
&= -\frac{\|\mathbf{b} - \mathbf{a}\|_2^2}{\|\mathbf{b} - \mathbf{a}\|_2} \\
&= -\|\mathbf{b} - \mathbf{a}\|_2
\end{aligned}$$

∎

$\|\mathbf{b}-\mathbf{a}\|_2$ ist offensichtlich gerade die euklidische Distanz der Punkte $\mathbf{a}$ und $\mathbf{b}$.

Lemma 3.4.3. *Sei T ein positiv orientiertes Dreieck mit Normalenvektor $\mathbf{n}$ und paarweise verschiedenen Eckpunkten $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$. Dann gilt:*

$$\begin{aligned}\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|_2 = \langle \mathbf{a}, \mathbf{t}_{ab}\rangle_2 \langle \mathbf{a}, \mathbf{u}_{ab}\rangle_2 + \langle \mathbf{b}, \mathbf{t}_{ba}\rangle_2 \langle \mathbf{b}, \mathbf{u}_{ba}\rangle_2 \\ + \langle \mathbf{a}, \mathbf{t}_{ac}\rangle_2 \langle \mathbf{a}, \mathbf{u}_{ac}\rangle_2 + \langle \mathbf{c}, \mathbf{t}_{ca}\rangle_2 \langle \mathbf{c}, \mathbf{u}_{ca}\rangle_2 \\ + \langle \mathbf{b}, \mathbf{t}_{bc}\rangle_2 \langle \mathbf{b}, \mathbf{u}_{bc}\rangle_2 + \langle \mathbf{c}, \mathbf{t}_{cb}\rangle_2 \langle \mathbf{c}, \mathbf{u}_{cb}\rangle_2\end{aligned}$$

Beweis. Zunächst beobachten wir:

$$\begin{aligned}\langle \mathbf{a}, \mathbf{u}_{ab}\rangle_2 &= \langle \mathbf{a}, \mathbf{n} \times \mathbf{t}_{ab}\rangle_2 && \text{(nach Gleichung 3.3.3)} \\ &= \frac{\langle \mathbf{a}, \mathbf{n} \times (\mathbf{b}-\mathbf{a})\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} && \text{(nach Definition von } \mathbf{t}_{ab}\text{)} \\ &= \frac{\langle \mathbf{a}, \mathbf{n} \times \mathbf{b} - \mathbf{n} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} && \text{(Linearität)} \\ &= \frac{\langle \mathbf{a}, \mathbf{n} \times \mathbf{b}\rangle_2 - \langle \mathbf{a}, \mathbf{n} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} && \text{(Linearität)} \\ &= \frac{\langle \mathbf{a}, \mathbf{n} \times \mathbf{b}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} && \text{(mit Lemma A.1.3)} \\ &= \frac{\langle \mathbf{n}, \mathbf{b} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} && \text{(mit Lemma A.1.4)}\end{aligned}$$

Und analog gilt:

$$\begin{aligned}\langle \mathbf{b}, \mathbf{u}_{ba}\rangle_2 &= \langle \mathbf{b}, \mathbf{u}_{ab}\rangle_2 && \text{(aus der Definition von } \mathbf{u}_{ab}\text{)} \\ &= \frac{\langle \mathbf{b}, \mathbf{n} \times \mathbf{b}\rangle_2 - \langle \mathbf{b}, \mathbf{n} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} \\ &= \frac{\langle \mathbf{b}, -\mathbf{n} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} \\ &= \frac{\langle \mathbf{b}, \mathbf{a} \times \mathbf{n}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} \\ &= \frac{\langle \mathbf{n}, \mathbf{b} \times \mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2}\end{aligned}$$

Daraus folgt

$$\langle \mathbf{a}, \mathbf{u}_{ab}\rangle_2 = \langle \mathbf{b}, \mathbf{u}_{ab}\rangle_2.$$

Geometrisch betrachtet sagt diese Gleichung aus, dass $\mathbf{a}$ und $\mathbf{b}$ zu einer Ebene mit Normalenvektor $\mathbf{u}_{ab}$ denselben Abstand haben. Das folgt aber auch unmittelbar aus der Definition von $\mathbf{u}_{ab}$, denn der Vektor $\mathbf{u}_{ab}$ ist so gewählt, dass er senkrecht auf der Geraden zwischen $\mathbf{a}$ und $\mathbf{b}$ steht, also muss die Gerade parallel zur Ebene verlaufen.

Die Gleichheit ermöglicht folgende Umformungen:

$$\begin{aligned}
\langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2\langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2\langle \mathbf{b},\, \mathbf{u}_{ba}\rangle_2 &= \Big(\langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2\Big)\langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2 \\
\langle \mathbf{a},\, \mathbf{t}_{ac}\rangle_2\langle \mathbf{a},\, \mathbf{u}_{ac}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{ca}\rangle_2\langle \mathbf{c},\, \mathbf{u}_{ca}\rangle_2 &= \Big(\langle \mathbf{a},\, \mathbf{t}_{ac}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{ca}\rangle_2\Big)\langle \mathbf{c},\, \mathbf{u}_{ca}\rangle_2 \\
\langle \mathbf{b},\, \mathbf{t}_{bc}\rangle_2\langle \mathbf{b},\, \mathbf{u}_{bc}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{cb}\rangle_2\langle \mathbf{c},\, \mathbf{u}_{cb}\rangle_2 &= \Big(\langle \mathbf{b},\, \mathbf{t}_{bc}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{cb}\rangle_2\Big)\langle \mathbf{c},\, \mathbf{u}_{bc}\rangle_2
\end{aligned}$$

Mit Lemma 3.4.2 erhalten wir dann

$$\begin{aligned}
\Big(\langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2\Big)\langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2 &= -\|\mathbf{b}-\mathbf{a}\|_2\langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2 \\
&= -\|\mathbf{b}-\mathbf{a}\|_2\frac{\langle \mathbf{n},\, \mathbf{b}\times\mathbf{a}\rangle_2}{\|\mathbf{b}-\mathbf{a}\|_2} \\
&= -\langle \mathbf{n},\, \mathbf{b}\times\mathbf{a}\rangle_2 \\
&= \langle \mathbf{n},\, \mathbf{a}\times\mathbf{b}\rangle_2
\end{aligned}$$

und analog dazu

$$\begin{aligned}
\Big(\langle \mathbf{b},\, \mathbf{t}_{bc}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{cb}\rangle_2\Big)\langle \mathbf{b},\, \mathbf{u}_{bc}\rangle_2 &= \langle \mathbf{n},\, \mathbf{b}\times\mathbf{c}\rangle_2 \\
\Big(\langle \mathbf{a},\, \mathbf{t}_{ac}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{ca}\rangle_2\Big)\langle \mathbf{a},\, \mathbf{u}_{ca}\rangle_2 &= \langle \mathbf{n},\, \mathbf{c}\times\mathbf{a}\rangle_2
\end{aligned}$$

Einsetzen liefert

$$\begin{aligned}
\langle \mathbf{n},\, \mathbf{a}\times\mathbf{b}\rangle_2 + \langle \mathbf{n},\, \mathbf{b}\times\mathbf{c}\rangle_2 + \langle \mathbf{n},\, \mathbf{c}\times\mathbf{a}\rangle_2 &= \langle \mathbf{n},\, \mathbf{a}\times\mathbf{b} + \mathbf{b}\times\mathbf{c} + \mathbf{c}\times\mathbf{a}\rangle_2 \\
&= \langle \mathbf{n},\, (\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\rangle_2 && \text{(mit Gleichung 3.2.2)} \\
&= \langle \mathbf{n},\, \|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2\mathbf{n}\rangle_2 && \text{(mit Gleichung 3.3.2)} \\
&= \|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2\langle \mathbf{n},\, \mathbf{n}\rangle_2 && \text{(Linearität)} \\
&= \|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2 && \text{(da } \|\mathbf{n}\|_2 = 1\text{)}
\end{aligned}$$

und die Behauptung folgt. ■

Der hier entstandene Term $\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2$ ist bereits aus der Einführung bekannt. Er entspricht dem doppelten Flächeninhalt des Dreiecks mit den Eckpunkten $\mathbf{a}$, $\mathbf{b}$ und $\mathbf{c}$.

Lemma 3.4.4. *Seien $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ paarweise verschiedene Eckpunkte eines Dreiecks. Dann gilt*

$$\begin{aligned}
\langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2 = {} & \langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2\langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2\langle \mathbf{a},\, \mathbf{n}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2\langle \mathbf{b},\, \mathbf{u}_{ba}\rangle_2\langle \mathbf{b},\, \mathbf{n}\rangle_2 \\
& + \langle \mathbf{a},\, \mathbf{t}_{ac}\rangle_2\langle \mathbf{a},\, \mathbf{u}_{ac}\rangle_2\langle \mathbf{a},\, \mathbf{n}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{ca}\rangle_2\langle \mathbf{c},\, \mathbf{u}_{ca}\rangle_2\langle \mathbf{c},\, \mathbf{n}\rangle_2 \\
& + \langle \mathbf{b},\, \mathbf{t}_{bc}\rangle_2\langle \mathbf{b},\, \mathbf{u}_{bc}\rangle_2\langle \mathbf{b},\, \mathbf{n}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{cb}\rangle_2\langle \mathbf{c},\, \mathbf{u}_{cb}\rangle_2\langle \mathbf{c},\, \mathbf{n}\rangle_2
\end{aligned}$$

Beweis. Wir berechnen

$$\begin{aligned}
\langle \mathbf{a},\, \mathbf{n}\rangle_2 &= \frac{\langle \mathbf{a},\, (\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\rangle_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2} \\
&= \frac{\langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2 + \langle \mathbf{a},\, \mathbf{a}\times\mathbf{b}\rangle_2 + \langle \mathbf{a},\, \mathbf{c}\times\mathbf{a}\rangle_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2} && \text{(mit Gleichung 3.2.2)} \\
&= \frac{\langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2} && \text{(mit Lemma A.1.3)}
\end{aligned}$$

und mit denselben Umformungen erhalten wir

$$\begin{aligned}
\langle \mathbf{b},\, \mathbf{n}\rangle_2 &= \frac{\langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2} \\
\langle \mathbf{c},\, \mathbf{n}\rangle_2 &= \frac{\langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2}.
\end{aligned}$$

Es folgt

$$\langle \mathbf{a},\, \mathbf{n}\rangle_2 = \langle \mathbf{b},\, \mathbf{n}\rangle_2 = \langle \mathbf{c},\, \mathbf{n}\rangle_2.$$

Dieser Ausdruck entspricht dem Abstand der durch $\mathbf{a}$, $\mathbf{b}$ und $\mathbf{c}$ verlaufenden Fläche zum Koordinatenursprung. Die Gleichheit ist somit auch geometrisch begründbar. Wir können nun die rechte Seite der Behauptung vereinfachen und den Beweis zu Ende führen:

$$\begin{aligned}
&\langle \mathbf{a},\, \mathbf{n}\rangle_2 \Big(\langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2 \langle \mathbf{a},\, \mathbf{u}_{ab}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2 \langle \mathbf{b},\, \mathbf{u}_{ba}\rangle_2 \\
&\qquad + \langle \mathbf{a},\, \mathbf{t}_{ac}\rangle_2 \langle \mathbf{a},\, \mathbf{u}_{ac}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{ca}\rangle_2 \langle \mathbf{c},\, \mathbf{u}_{ca}\rangle_2 \\
&\qquad + \langle \mathbf{b},\, \mathbf{t}_{bc}\rangle_2 \langle \mathbf{b},\, \mathbf{u}_{bc}\rangle_2 + \langle \mathbf{c},\, \mathbf{t}_{cb}\rangle_2 \langle \mathbf{c},\, \mathbf{u}_{cb}\rangle_2 \Big) \\
&= \langle \mathbf{a},\, \mathbf{n}\rangle_2 \, \|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2 && \text{(mit Lemma 3.4.3)} \\
&= \langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2 \, \frac{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2}{\|(\mathbf{b}-\mathbf{a})\times(\mathbf{c}-\mathbf{a})\|_2} \\
&= \langle \mathbf{a},\, \mathbf{b}\times\mathbf{c}\rangle_2 && \blacksquare
\end{aligned}$$

Wir zeigen nun die klassische Formel zur Bestimmung des Volumens eines Polyeders:

Lemma 3.4.5. *Sei P ein Polyeder, das von $N \in \mathbb{N}$ positiv orientierten und dreieckigen Seitenflächen begrenzt wird. Die Eckpunkte der Seitenflächen seien durch die Vektoren $\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i \in \mathbb{R}^3$ gegeben. Dann gilt*

$$\operatorname{vol}(P) = \frac{1}{6} \sum_{i=1}^{N} \langle \mathbf{a}_i,\, \mathbf{b}_i \times \mathbf{c}_i\rangle_2.$$

Beweis. Wir beginnen mit der Beobachtung, dass sich jedes Polyeder P in disjunkte Tetraeder $T_1, \ldots, T_M$ zerlegen lässt. Für jede Tetraederzerlegung gilt

$$\operatorname{vol}(P) = \sum_{i=1}^{M} \operatorname{vol}(T_i).$$

Sind die Eckpunkte von T_i durch die Vektoren $\mathbf{a}_i$, $\mathbf{b}_i$, $\mathbf{c}_i$, $\mathbf{d}_i$ gegeben und alle Flächen von T_i positiv orientiert, so gilt

$$\begin{aligned}
\operatorname{vol}(P) &= \sum_{i=1}^{M} \operatorname{vol}(T_i) \\
&= \sum_{i=1}^{M} \frac{1}{6} \langle \mathbf{b}_i - \mathbf{a}_i, (\mathbf{c}_i - \mathbf{a}_i) \times (\mathbf{d}_i - \mathbf{a}_i) \rangle_2 \\
&= \sum_{i=1}^{M} \frac{1}{6} \Big(\langle \mathbf{b}_i, \mathbf{c}_i \times \mathbf{d}_i \rangle_2 - \langle \mathbf{b}_i, \mathbf{c}_i \times \mathbf{a}_i \rangle_2 - \langle \mathbf{b}_i, \mathbf{a}_i \times \mathbf{d}_i \rangle_2 - \langle \mathbf{a}_i, \mathbf{c}_i \times \mathbf{d}_i \rangle_2 \Big) \\
&= \sum_{i=1}^{M} \frac{1}{6} \Big(\langle \mathbf{b}_i, \mathbf{c}_i \times \mathbf{d}_i \rangle_2 + \langle \mathbf{a}_i, \mathbf{c}_i \times \mathbf{b}_i \rangle_2 + \langle \mathbf{a}_i, \mathbf{b}_i \times \mathbf{d}_i \rangle_2 + \langle \mathbf{a}_i, \mathbf{d}_i \times \mathbf{c}_i \rangle_2 \Big)
\end{aligned} \tag{3.4.1}$$

Jedes der Spatprodukte involviert die Eckpunkte von jeweils einer Seitenfläche von T_i.

Wir wählen nun eine Zerlegung von P derartig, dass die äußeren Seitenflächen der Tetraeder genau mit den Seitenflächen von P übereinstimmen. Da jede Tetraederseite positiv orientiert ist, sind angrenzende Tetraederflächen immer entgegengesetzt orientiert. In diesem Fall gilt für die Eckpunkte $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^3$ der von zwei Tetraedern geteilten Fläche die Beziehung

$$\langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 + \langle \mathbf{w}, \mathbf{v} \times \mathbf{u} \rangle_2 = \langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 - \langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 = 0.$$

Damit eliminieren sich die Spatprodukte für innere Flächen aus Gleichung 3.4.1 vollständig. Die nicht eliminierten Terme liegen somit gerade an den Außenflächen des Polyeders und es folgt die Behauptung. ■

Aus dem Zusammenspiel der gerade bewiesenen Lemmata ergibt sich das gewünschte Ergebnis:

Korollar. *Die Aussage von Satz 3.3.1 und Satz 3.3.2 gilt.*

Beweis. Sei P ein beliebiges Polyeder. Wir zeigen zunächst die Gleichungen aus Satz 3.3.2. Dazu beobachten wir, dass wir die zu einer Kante zwischen $\mathbf{a}$ und $\mathbf{b}$ gehörenden Quadrupel

$$(\mathbf{a}, \mathbf{t}_{ab}, \mathbf{u}_{ab}, \mathbf{n}) \text{ und } (\mathbf{b}, \mathbf{t}_{ba}, \mathbf{u}_{ba}, \mathbf{n})$$

immer paarweise zusammenfassen können. Zu jeder Kante existiert dieses Paar zweimal in $\mathcal{Q}$, nämlich einmal für jede der zwei angrenzenden Flächen. Mit Lemma 3.4.2 erhalten wir

$$\begin{aligned} L &= \sum_{(\mathbf{a},\,\mathbf{b}) \text{ ist Kante von } P} \|\mathbf{b}-\mathbf{a}\|_2 \\ &= \sum_{(\mathbf{a},\,\mathbf{b}) \text{ ist Kante von } P} \Big(\langle \mathbf{a},\, \mathbf{t}_{ab}\rangle_2 + \langle \mathbf{b},\, \mathbf{t}_{ba}\rangle_2\Big) \\ &= \frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} -\langle \mathbf{p},\, \mathbf{t}\rangle_2 \\ &= -\frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} \langle \mathbf{p},\, \mathbf{t}\rangle_2. \end{aligned}$$

In der Definition von $\mathcal{Q}$ wurden die Quadrupel über die Flächen des Polyeders definiert. Wir können deshalb die Menge $\mathcal{Q}$ nach Flächenzugehörigkeit gruppieren und die Flächen einzeln auswerten. Lemma 3.4.1 erlaubt es uns, statt einer beliebigen Fläche F von P nur eine Triangulierung von F aus positiv orientierten Dreiecken T mit Eckpunkten $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ zu betrachten. Dann gilt

$$\begin{aligned} \operatorname{ar}(P) &= \sum_{T \text{ begrenzt } P} \operatorname{ar}(T) \\ &= \sum_{T \text{ begrenzt } P} \frac{1}{2}\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|_2 \\ &= \sum_{T \text{ begrenzt } P} \frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in \mathcal{Q}|_T} \langle \mathbf{p},\, \mathbf{t}\rangle_2 \langle \mathbf{p},\, \mathbf{u}\rangle_2 \qquad \text{(mit Lemma 3.4.3)} \\ &= \frac{1}{2} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n})\in\mathcal{Q}} \langle \mathbf{p},\, \mathbf{t}\rangle_2 \langle \mathbf{p},\, \mathbf{u}\rangle_2 \end{aligned}$$

wobei $\mathcal{Q}|_T$ für die Teilmenge von $\mathcal{Q}$ steht, die T zugeordnet ist.

Zuletzt zeigen wir Satz 3.3.1. Genau wie eben betrachten wir wieder nur eine Triangulierung der Begrenzungsflächen. Dann gilt

$$\begin{aligned}
\operatorname{vol}(P) &= \frac{1}{6} \sum_{T \text{ begrenzt } P} \langle \mathbf{a},\, \mathbf{b} \times \mathbf{c} \rangle_2 && \text{(mit Lemma 3.4.5)} \\
&= \frac{1}{6} \sum_{T \text{ begrenzt } P} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n}) \in \mathcal{Q}|_T} \langle \mathbf{p},\, \mathbf{t} \rangle_2 \langle \mathbf{p},\, \mathbf{u} \rangle_2 \langle \mathbf{p},\, \mathbf{n} \rangle_2 && \text{(mit Lemma 3.4.4)} \\
&= \frac{1}{6} \sum_{(\mathbf{p},\mathbf{t},\mathbf{u},\mathbf{n}) \in \mathcal{Q}} \langle \mathbf{p},\, \mathbf{t} \rangle_2 \langle \mathbf{p},\, \mathbf{u} \rangle_2 \langle \mathbf{p},\, \mathbf{n} \rangle_2
\end{aligned}$$

Damit ist die zentrale Aussage dieses Kapitels gezeigt. ■

4 Umsetzung

Dieses Kapitel erklärt, wie wir die Formel aus Kapitel 3 einsetzen können, um das Schnittvolumen zweier Dreiecksnetze zu bestimmen.

4.1 Eingabedatenformat

Im letzten Kapitel haben wir allgemeine Polyeder betrachtet. Beschränken wir die zulässigen Begrenzungsflächen eines Polyeders auf Dreiecke, erhalten wir die in der Computergrafik üblichen Dreiecksnetze. Dreiecke haben hier den Vorteil, dass genau drei Eckpunkte nötig sind, um eine Ebene im dreidimensionalen Raum zu definieren. Nehmen wir mehr Eckpunkte hinzu, kann nicht mehr sichergestellt werden, dass alle Eckpunkte in derselben Ebene liegen. Jedes Polyeder kann durch Zerlegen der Seiten in Dreiecke in ein zulässiges Dreiecksnetz umgewandelt werden.

Zusätzlich gehen wir davon aus, dass alle Dreiecke in der Eingabe positiv orientiert sind. Das hat den Vorteil, dass wir an jeder Begrenzungsfläche berechnen können, auf welcher Seite sich die Innenseite des Dreiecksnetzes befindet. Diese Information ist auch für die korrekte Berechnung der Normalenrichtung und für Sichtbarkeitstests beim Darstellen der Fläche wichtig und ist deshalb in grafischen Anwendungen häufig bereits gegeben. Wir fordern für unsere Eingabenetze weiterhin, dass keine vier Eckpunkte koplanar liegen. Wir diskutieren diese Einschränkung im Abschnitt über Degeneriertheiten.

Für die Implementierung verwenden wir das STL-Dateiformat, das zu jedem Dreieck des Netzes die Koordinaten der Eckpunkte nacheinander aufzählt. Das STL-Dateiformat unterstützt als Begrenzungsflächen ausschließlich Dreiecke. Andere Dateiformate kommen ohne diese Restriktion aus, sind aber dafür mit größerem Aufwand beim Einlesen verbunden. Insbesondere müssten wir für unsere Zwecke alle nichtdreieckigen Flächen nachtriangulieren. Das STL-Dateiformat speichert außerdem alle Dreiecke unabhängig voneinander, an einem Dreieck sind weder angrenzende Flächen noch das umschlossene Volumen ablesbar.

Anstatt einfach für alle Seiten die Eckpunkte nacheinander aufzuzählen, ist es auch möglich, zuerst alle voneinander verschiedenen Eckpunkte aufzulisten und

dann für jede Fläche die Indizes der Begrenzungspunkte zu speichern. Diese häufig platzsparendere Repräsentation ist in einigen Dateiformaten bereits eingebaut. Für manche Algorithmen ist diese Repräsentation sogar Voraussetzung, wir werden gegen Ende dieses Kapitels weiter darauf eingehen.

4.2 Volumenbestimmung eines einzelnen Dreiecksnetzes

In Kapitel 3 wurden bereits mehrere Wege eingeführt, das Volumen eines Polyeders zu berechnen. Ein einfaches Verfahren ist durch Lemma 3.4.5 gegeben, indem man die Summanden durch Iteration über die Seitenflächen nacheinander auswertet.

Alternativ können wir hier die Formel aus Satz 3.3.1 anwenden, um das Volumen zu bestimmen. Da uns die Seitenflächen bereits separat vorliegen, müssen wir nur über die Kanten der einzelnen Flächen iterieren, um alle Quadrupel aus $\mathcal{Q}$ zu erhalten. Für die Volumenberechnung muss $\mathcal{Q}$ nie explizit bestimmt werden, es reicht aus, die Summanden aus Satz 3.3.1 direkt auszuwerten und alle Ergebnisse zu akkumulieren.

Diese Methode ist für einzelne Dreiecksnetze mit einem höheren Berechnungsaufwand verbunden, benötigt aber weniger Informationen über die Dreiecksnetze. Für einen Punkt, in dem ein Summand ausgewertet wird, müssen lediglich die Richtungen der benachbarten Punkte sowie die der angrenzenden Flächen bekannt sein. Informationen über die Flächen selbst werden nicht benötigt, es muss also nicht bekannt sein, welche Eckpunkte gemeinsam eine Fläche begrenzen oder welche Flächen sich Eckpunkte teilen.

Möchten wir das Volumen des Schnitts zweier Dreiecksnetze berechnen, besteht grundsätzlich die Möglichkeit, den Schnittkörper explizit zu konstruieren und anschließend über Lemma 3.4.5 das Volumen dieses Körpers zu berechnen. Das Problem besteht hier darin, dass die Seitenflächen des Schnittkörpers nicht notwendigerweise wieder Dreiecke sein müssen, sondern beliebige polygonale Form annehmen können. Für die Überführung in ein Dreiecksnetz fällt zusätzlicher Berechnungsaufwand für die Retriangulierung an. In Kapitel 2 haben wir mehrere Arbeiten aufgezeigt, die auf die Retriangulierung näher eingehen.

Wir können dieses Problem umgehen, indem wir aus den einzelnen Schnitten zwischen Dreiecken der Eingabenetze genug Informationen extrahieren, um mit der Formel aus Satz 3.3.1 das Schnittvolumen direkt zu berechnen. Diese Methode

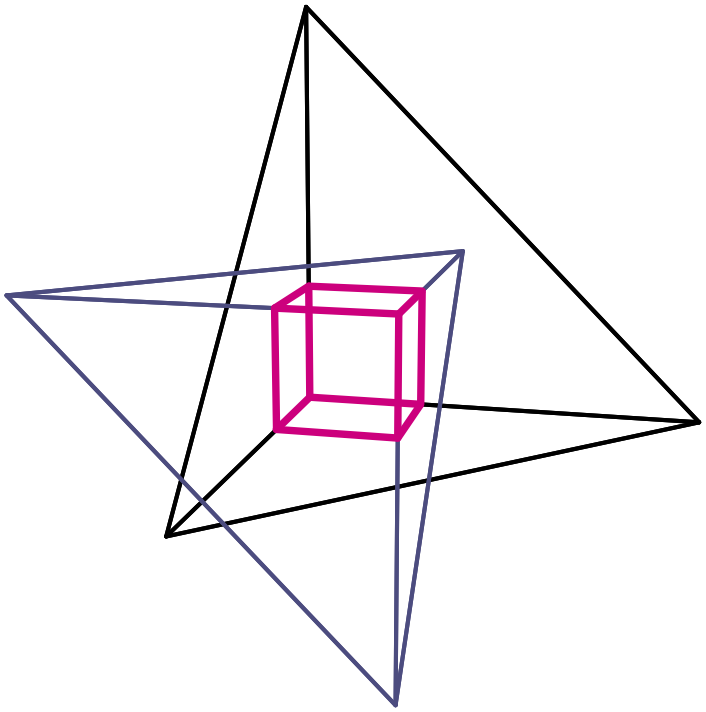


Abb. 4.1.: Zwei Tetraeder schneiden sich in einem Würfel. Zwei Eckpunkte des Würfels ergeben sich aus den Eckpunkten der Tetraeder, die im jeweils anderen Tetraeder liegen. Die übrigen sechs Ecken entstehen durch Schnitte von Kanten mit Flächen.

kommt vollständig ohne Retriangulierung aus. Wir zeigen in den nächsten Abschnitten, wie man die nötigen Informationen dafür gewinnt.

4.3 Struktur des Schnittkörpers

Obwohl wir den Schnittkörper nie explizit konstruieren, hilft es uns zunächst dennoch, dessen Struktur zu verstehen. Sei also K der Schnittkörper, der aus dem Schnitt der Polyeder P_1 und P_2 resultiert.

Zunächst stellen wir fest, dass die Begrenzungsflächen von K immer auch entweder P_1 oder P_2 begrenzen. Alle Begrenzungspunkte von K liegen also ausschließlich auf bereits bekannten Flächen.

In dem einfachen Fall, dass ein Polyeder das andere vollständig enthält, entspricht der Schnittkörper gerade dem Polyeder mit den kleineren Volumen.

Enthält ein Polyeder das andere Polyeder nicht vollständig, schneidet sich mindestens ein Dreieck von P_1 mit einem Dreieck von P_2. Die Schnittkante teilt dann beide Dreiecke in den Teil außerhalb des Schnittvolumens und den Teil innerhalb des Schnittvolumens und begrenzt somit den Schnittkörper K. Die Begrenzungspunkte

der Schnittkante fallen notwendigerweise mit den Kanten eines Dreiecks aus P_1 oder P_2 zusammen. Es reicht also aus, Schnitte zwischen Kanten des einen Polyeders mit Dreiecken des jeweils anderen Polyeders zu betrachten, um die Begrenzungspunkte des Schnittkörpers zu finden.

Die übrigen Eckpunkte von K sind durch die Punkte von P_1 gegeben, die innerhalb von P_2 liegen, und gleichermaßen durch die Punkte von P_2, die in P_1 liegen. Abbildung 4.1 veranschaulicht beide Fälle am Beispiel eines Tetraederschnitts.

4.4 Informationsgehalt eines Dreiecksschnitts

Wir zeigen nun, wie man einen Schnitt zwischen einer Dreieckskante und einem Dreieck berechnen kann und welche Informationen dieser über den Schnittkörper liefert.

4.4.1 Berechnung der Schnittpunkte

In diesem Abschnitt betrachten wir eine Dreieckskante zwischen den Punkten $\mathbf{s}$ und $\mathbf{e}$ des Dreiecks T_1, die wir auf einen Schnitt mit dem Dreieck T_2 mit den Eckpunkten $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ testen wollen.

Dazu benötigen wir die Ebene E durch die Punkte $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ und die Gerade g durch die Punkte $\mathbf{s}$ und $\mathbf{e}$, formal sind diese durch die Mengen

$$E = \{\, \mathbf{a} + x(\mathbf{b} - \mathbf{a}) + y(\mathbf{c} - \mathbf{a}) \mid x, y \in \mathbb{R} \,\}$$
$$g = \{\, \mathbf{s} + z(\mathbf{e} - \mathbf{s}) \mid z \in \mathbb{R} \,\}$$

gegeben. Das Dreieck $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ ist dann eine Teilmenge von E und die Kante zwischen $\mathbf{s}$ und $\mathbf{e}$ eine Teilmenge von g. Wenn E und g sich schneiden, existieren $x, y, z \in \mathbb{R}$ sodass

$$\mathbf{a} + x(\mathbf{b} - \mathbf{a}) + y(\mathbf{c} - \mathbf{a}) = \mathbf{s} + z(\mathbf{e} - \mathbf{s})$$

gilt. Schreiben wir alle von x, y, z unabhängigen Terme auf die rechte Seite, so gilt äquivalent dazu

$$x(\mathbf{b} - \mathbf{a}) + y(\mathbf{c} - \mathbf{a}) + z(\mathbf{s} - \mathbf{e}) = \mathbf{s} - \mathbf{a}.$$

Jetzt können wir dieses lineare Gleichungssystem als Matrix-Vektor-Produkt schreiben:

$$\begin{bmatrix} | & | & | \\ (\mathbf{b} - \mathbf{a}) & (\mathbf{c} - \mathbf{a}) & (\mathbf{s} - \mathbf{e}) \\ | & | & | \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \mathbf{s} - \mathbf{a}$$

Die Matrix ist genau dann invertierbar, wenn g und E nicht parallel zueinander liegen. In diesem Fall hat das Gleichungssystem die Lösung

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} | & | & | \\ (\mathbf{b}-\mathbf{a}) & (\mathbf{c}-\mathbf{a}) & (\mathbf{s}-\mathbf{e}) \\ | & | & | \end{bmatrix}^{-1} (\mathbf{s}-\mathbf{a}).$$

Für die Inverse einer 3×3-Matrix lässt sich über die adjunkte Matrix eine geschlossene Formel herleiten[17]. Diese ist üblicherweise in Programmbibliotheken für 3D-Geometrie bereits implementiert.

Der Schnittpunkt $\mathbf{p}$ ist gegeben durch

$$\mathbf{p} = \mathbf{s} + z(\mathbf{e}-\mathbf{s}).$$

Damit $\mathbf{p}$ innerhalb des Dreiecks $\mathbf{a}, \mathbf{b}, \mathbf{c}$ und auf der Strecke zwischen $\mathbf{s}$ und $\mathbf{e}$ liegt, müssen zusätzlich die Bedingungen

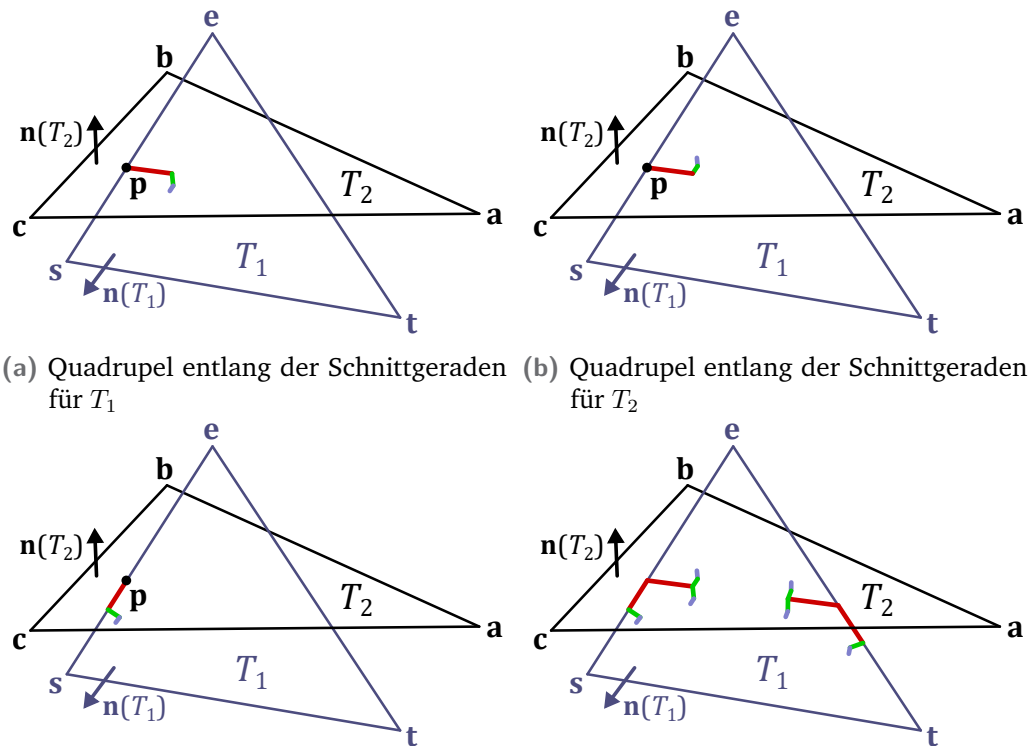
$$\begin{aligned} 0 \leq x,\, y \\ x + y \leq 1 \\ 0 \leq z \leq 1 \end{aligned}$$

erfüllt sein.

4.4.2 Auswertung eines Dreiecksschnitts

Aus der lokalen Umgebung von $\mathbf{p}$ können wir drei Quadrupel aus der Menge $\mathcal{Q}$ ableiten. Dazu betrachten wir die zwei Kanten des Schnittkörpers, von denen wir sicher wissen, dass sie durch $\mathbf{p}$ verlaufen.

Explizit gegeben ist die Kante zwischen $\mathbf{s}$ und $\mathbf{e}$, die auch eins der Polyeder begrenzt. Bewegen wir uns auf dieser Kante von $\mathbf{p}$ aus entgegen der Richtung des Normalenvektors von T_2, so bewegen wir uns ins Innere des Schnittvolumens. Da noch weitere Dreiecke die Kante zwischen $\mathbf{s}$ und $\mathbf{e}$ schneiden können, können wir weder eine Aussage darüber treffen, wie weit wir von $\mathbf{p}$ aus weitergehen können, bis wir auf die nächste Begrenzungsfläche treffen, noch können wir an der Normalenrichtung von T_2 entscheiden, ob $\mathbf{s}$ oder $\mathbf{e}$ auf der Innen- oder Außenseite des Schnittvolumens liegen. Für die Volumenformel ist diese Information nicht weiter nötig, dort benötigen wir nur die Richtung der Begrenzungskante.



(a) Quadrupel entlang der Schnittgeraden für T_1

(b) Quadrupel entlang der Schnittgeraden für T_2

(c) Quadrupel entlang der Strecke zwischen $\mathbf{s}$ und $\mathbf{e}$

(d) Alle Quadrupel für diesen Dreiecksschnitt

Abb. 4.2.: Darstellung der Quadrupel, die aus dem Schnitt von T_2 mit der Strecke zwischen $\mathbf{s}$ und $\mathbf{e}$ hervorgehen. 4.2d zeigt alle Quadrupel für den Schnitt von T_1 und T_2.

Zu der Kante zwischen $\mathbf{s}$ und $\mathbf{e}$ kennen wir lokal nur eine einzige angrenzende Fläche, nämlich gerade das Dreieck T_1 selbst mit seinem Normalenvektor $\mathbf{n}(T_1)$. Damit ergibt sich das erste Quadrupel bis auf noch unbekannte Vorzeichen zu

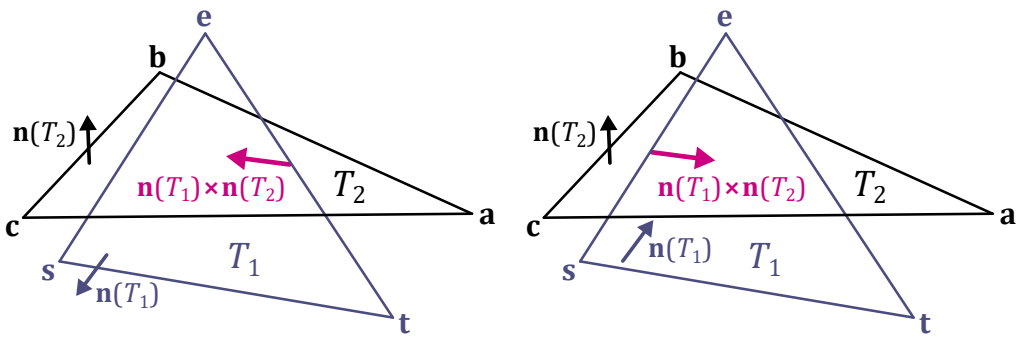
$$\left(\mathbf{p},\ \pm\frac{\mathbf{s}-\mathbf{e}}{\|\mathbf{s}-\mathbf{e}\|_2},\ \pm\frac{\mathbf{s}-\mathbf{e}}{\|\mathbf{s}-\mathbf{e}\|_2}\times\mathbf{n}(T_1),\ \mathbf{n}(T_1)\right). \tag{4.4.1}$$

Die zweite Kante ist implizit gegeben durch die Schnittkante der Dreiecke T_1 und T_2. Da die Schnittkante senkrecht auf beiden Dreiecken stehen muss, muss diese in Richtung des Vektors $\mathbf{n}(T_1)\times\mathbf{n}(T_2)$ zeigen. Zu dieser Schnittkante kennen wir zwei anliegende Flächen, T_1 und T_2. Analog zu oben finden wir das Quadrupel

$$\left(\mathbf{p},\ \pm\frac{\mathbf{n}(T_1)\times\mathbf{n}(T_2)}{\|\mathbf{n}(T_1)\times\mathbf{n}(T_2)\|_2},\ \pm\frac{\mathbf{n}(T_1)\times\mathbf{n}(T_2)}{\|\mathbf{n}(T_1)\times\mathbf{n}(T_2)\|_2}\times\mathbf{n}(T_1),\ \mathbf{n}(T_1)\right) \tag{4.4.2}$$

für T_1 und

$$\left(\mathbf{p},\ \pm\frac{\mathbf{n}(T_1)\times\mathbf{n}(T_2)}{\|\mathbf{n}(T_1)\times\mathbf{n}(T_2)\|_2},\ \pm\frac{\mathbf{n}(T_1)\times\mathbf{n}(T_2)}{\|\mathbf{n}(T_1)\times\mathbf{n}(T_2)\|_2}\times\mathbf{n}(T_2),\ \mathbf{n}(T_2)\right) \tag{4.4.3}$$



(a) Dreiecksschnitt aus Abbildung 4.2 (b) 4.3a mit umgekehrter Normalenrichtung

Abb. 4.3.: Orientierung des Vektors $\mathbf{n}(T_1) \times \mathbf{n}(T_2)$ mit unterschiedlichen Normalenrichtungen für T_1. Das Kreuzprodukt ist für jeweils eine Kante korrekt ins Innere von T_2 orientiert, für die andere Kante zeigt es außerhalb des Dreiecks.

für T_2. Wenn T_1 und T_2 koplanar sind, verschwindet $\mathbf{n}(T_1) \times \mathbf{n}(T_2)$. Wir gehen im Kapitel über Degeneriertheiten genauer auf diesen Fall ein. Die drei beschriebenen Quadrupel werden in Abbildung 4.2 visualisiert.

Zur Auswertung der Terme aus Satz 3.3.1 müssen natürlich alle Vorzeichen bekannt sein. Abbildung 4.3 zeigt am Beispiel des Vektors $\mathbf{n}(T_1) \times \mathbf{n}(T_2)$, dass eine feste Wahl eines Vorzeichens zu einer falschen Orientierung der Vektoren führen kann. Stattdessen müssen wir die Vektoren einzeln eigenhändig orientieren.

Der erste hiervon betroffene Vektor ist der Tangentialterm für die Strecke zwischen $\mathbf{s}$ und $\mathbf{e}$ in 4.4.1. Dieser muss ins Innere des Schnittvolumens zeigen, also entgegen der Normalenrichtung von sowohl T_1 als auch T_2. Da der Vektor bereits koplanar zu T_1 ist, müssen wir diese Bedingung nur für T_2 überprüfen: Gilt

$$\left\langle -\mathbf{n}(T_2), \frac{\mathbf{s}-\mathbf{e}}{\|\mathbf{s}-\mathbf{e}\|_2} \right\rangle_2 < 0,$$

müssen wir den Vektor umdrehen, ansonsten zeigt dieser bereits in die korrekte Richtung.

Der andere Vektor von 4.4.1 muss so orientiert sein, dass er ins Innere des Dreiecks T_1 zeigt. Zeigt die Kante zwischen $\mathbf{s}$ und $\mathbf{e}$ gegen den Uhrzeigersinn, ist

$$\frac{\mathbf{s}-\mathbf{e}}{\|\mathbf{s}-\mathbf{e}\|_2} \times \mathbf{n}(T_1) \tag{4.4.4}$$

bereits auf die Innenseite des Dreiecks gerichtet, andernfalls drehen wir diese um.

In 4.4.2 und 4.4.3 muss der Tangentialvektor in beiden Fällen ins Innere von T_1 zeigen, denn dort verläuft die Schnittkante. Hierbei ist wichtig, dass dieser Vektor nicht notwendigerweise wie 4.4.4 senkrecht auf der Dreieckskante steht. Wir nutzen

zur Orientierung aus, dass der Vektor ein positives Skalarprodukt mit 4.4.4 haben muss, damit er ins Innere von T_1 zeigt.

Die verbleibenden Vektoren müssen so gerichtet sein, dass sie jeweils ins Innere des Schnittvolumens zeigen. Wir können hier die Normalenrichtung von T_1 nutzen, um den Vektor zu orientieren, der in T_2 liegt, und andersherum.

Wir halten den Algorithmus zur Feststellung eines Schnitts als Pseudocode-Funktion fest, um diesen später referenzieren zu können. Die Referenzimplementierung orientiert sich zu großen Teilen an dieser Formulierung.

In allen Pseudocode-Funktionen in dieser Arbeit bezeichnet T ein Dreieck mit den Eckpunkten $T.\mathbf{a}$, $T.\mathbf{b}$, $T.\mathbf{c}$ und dem Normalenvektor $T.\mathbf{n}$. Für Linien verwenden wir L mit den Attributen $L.\mathbf{s}$, $L.\mathbf{e}$ für Start- und Endpunkt. Auf die Komponenten eines Vektors $\mathbf{v}$ greifen wir im Pseudocode mit $\mathbf{v}.x$, $\mathbf{v}.y$ und $\mathbf{v}.z$ zu.

Funktion solve_intersection(T, L)

$$\mathbf{M} \leftarrow \begin{bmatrix} | & | & | \\ (T.\mathbf{b} - T.\mathbf{a}) & (T.\mathbf{c} - T.\mathbf{a}) & (L.\mathbf{s} - L.\mathbf{e}) \\ | & | & | \end{bmatrix}$$

if $|\det(\mathbf{M})| < \epsilon$ **then**
    return nil
else
    $\mathbf{b} \leftarrow L.\mathbf{s} - T.\mathbf{a}$
    $\mathbf{u} \leftarrow \mathbf{M}^{-1}\mathbf{b}$
    if $\mathbf{u}.x < 0$ **or** $\mathbf{u}.y < 0$ **or** $\mathbf{u}.x + \mathbf{u}.y > 1$ **then**
        return nil
    else
        return $\mathbf{u}.z$

4.5 Ablauf der Schnittvolumenberechnung

Bereits in Abschnitt 4.3 wurde bemerkt, dass zusätzlich zu den eben behandelten Dreiecksschnitten auch innere Punkte das Schnittvolumen begrenzen und deshalb in der Berechnung berücksichtigt werden müssen. Zusätzlich zeigt Abschnitt 4.4, dass sich die Eckpunkte einer Kante nicht durch Betrachtung eines einzelnen Schnitts klassifizieren lassen. Wir zeigen jetzt, wie wir zu zwei Eckpunkten $\mathbf{p}$ und $\mathbf{q}$ einer Kante von (ohne Einschränkung) P_1 herausfinden, ob diese in P_2 liegen.

Da wir keine Konvexität unseres Netzes fordern, nutzen wir einen Strahltest, um innere Punkte zu identifizieren. Dafür betrachten wir einen Strahl von $\mathbf{p}$ in eine beliebige Richtung. Schneidet der Strahl eine ungerade Anzahl von Dreiecken in P_2, so liegt $\mathbf{p}$ innerhalb von P_2, bei einer geraden Anzahl außerhalb. Wir richten den Strahl so, dass er durch $\mathbf{q}$ verläuft, und können auf diese Weise $\mathbf{p}$ und $\mathbf{q}$ gleichzeitig klassifizieren.

Für konvexe Netze existieren einfachere Tests, um innere Punkte zu identifizieren. Hier wird ausgenutzt, dass ein Punkt nicht im Inneren des Netzes sein kann, wenn er für mindestens eine Außenseite in Richtung des Normalenvektors liegt. Dieses Verfahren ist vor allem für Tetraeder sinnvoll, da diese notwendigerweise konvex sein müssen.

Wir integrieren den Strahltest in die Berechnung der Dreiecksschnitte, indem wir bei der Schnittpunktbestimmung zwischen Geraden und Flächen aus Abschnitt 4.4 für jede Kante mitzählen, wie viele Schnitte auf der Kante selbst liegen und wie viele Dreiecke geschnitten werden, wenn wir die Kante einseitig durch einen Strahl fortsetzen.

Da ein Eckpunkt immer mindestens zwei ausgehende Kanten hat, wird er für jede dieser Kanten unabhängig klassifiziert. Die Klassifikation mit dem Strahltest liefert konsistente Ergebnisse, solange beide Netze keine Löcher in der Triangulierung aufweisen. Diese Voraussetzung haben wir bereits in Kapitel 3 gefordert.

Der vollständige Algorithmus zur Schnittvolumenberechnung ist in Abbildung 4.4 als Pseudocode dargestellt.

An diesem Punkt sollte explizit erwähnt werden, dass durch die Normierung der Vektoren irrationale Zahlen entstehen können, die wir mit endlicher Präzision nicht exakt speichern können. Das gilt sogar für den Fall, dass alle Koordinaten durch rationale Zahlen gegeben sind und folglich auch das Schnittvolumen am Ende rational sein muss. Wir können das Schnittvolumen allerdings beliebig gut annähern. Für unsere Zwecke fixieren wir die Anzahl der signifikanten Stellen und geben uns mit einer Approximation des tatsächlichen Volumens zufrieden.

Der Pseudocode zeigt, dass die Berechnung der Dreiecksschnitte einen signifikanten Anteil der Berechnung ausmacht. Intuitiv ist es allerdings unwahrscheinlich, dass sich jedes Paar von Dreiecken tatsächlich schneidet. Wenn wir eine räumliche Datenstruktur einsetzen, kann eine potenziell große Teilmenge der zu testenden Schnitte im Voraus ausgeschlossen werden.

Funktion intersection_volume(P_1, P_2)

```
acc ← 0                                                    › Akkumulatorvariable
foreach triangle T_1 of P_1 do
    foreach triangle side L of T_1 from p to q do
        co ← 0                                             › zählt Schnitte auf der Kante
        cb ← 0                                             › zählt Schnitte vor der Kante
        foreach triangle T_2 in P_2 do
            s ← solve_intersection(T_2, L)
            if s = nil then                                › kein Schnitt
                continue
            else
                if s > 1 then                              › Schnitt nach der Kante
                    continue
                if s < 0 then                              › Schnitt vor der Kante
                    cb ← cb + 1
                    continue
                co ← co + 1                                › Schnitt auf der Kante
                p ← (1 − s) · L.s + s · L.e
                evaluate the intersection in p and add the result to acc
        if cb mod 2 = 1 then                               › Startpunkt im Volumen
            evaluate inner point L.s and add the result to acc
        if (cb + co) mod 2 = 1 then                        › Endpunkt im Volumen
            evaluate inner point L.e and add the result to acc
repeat all above steps once with P_1 and P_2 swapped
return acc
```

Abb. 4.4.: Algorithmus zur Schnittvolumenberechnung

Wir können hierzu wie Franklin und Magalhães[7] den Raum mithilfe eines uniformen 3D-Gitters in Zellen unterteilen und nur Dreiecke miteinander schneiden, die in derselben Gitterzelle liegen. Die Idee ist hier, dass in jede Gitterzelle im Erwartungswert eine konstante Anzahl von Dreiecken fällt und somit die quadratische Laufzeit auf lineare Laufzeit reduziert werden kann.

Alternativ können mithilfe von Octrees[21] benachbarte Dreiecke schnell aufgefunden werden. Wir vermuten allerdings, dass sich baumartige Datenstrukturen aufgrund ihrer unvorhersehbaren und unzusammenhängenden Speicherzugriffe nicht für GPU-basierte Parallelisierung eignen.

Andere Ansätze partitionieren nicht den Raum, sondern die Dreiecksnetze selbst. Wir können hier für eine Teilmenge von Dreiecken einen Körper finden, der die Dreiecke vollständig einschließt und einen einfachen geometrischen Schnitttest erlaubt. Die offensichtlichste Variante hiervon sind Quader, deren Kanten parallel zu den Koordinatenachsen gerichtet sind. Mit einer Hierarchie dieser Hüllkörper können wir zu einen gegebenen Dreieck alle Dreiecke für einen Schnitt ausschließen, wenn das gegebene Dreieck den Hüllkörper nicht berührt.

Im Allgemeinen wird nicht garantiert, dass Hüllkörper disjunkter Dreiecksmengen ebenfalls disjunkt sind. Die Wahl der umhüllten Teilmenge hat einen erheblichen Einfluss auf die Größe des Hüllkörpers, die Konstruktion einer geeigneten Hüllkörperhierarchie ist folglich ein selbstständiges Problem. Betrachten wir nur die Eckpunkte eines Netzes, sind auch Lösungen über Clustering-Verfahren denkbar. Wir gehen in dieser Arbeit nicht weiter darauf ein. Alle Datenstrukturen, die von den Dreiecksnetzen und nicht vom umgebenden Raum abhängen, haben den Vorteil, dass wir diese für ein gegebenes Netz vorberechnen können.

In unserer ursprünglichen Formulierung des Algorithmus verwenden wir zur Identifizierung innerer Punkte einen Strahltest. Damit dieser zuverlässig funktioniert, dürfen wir entlang des Strahls kein Dreieck auslassen, wodurch der Gewinn durch räumliche Datenstrukturen eingeschränkt wird. Wir stellen deshalb ein komplexeres Verfahren für die Klassifikation innerer Punkte vor, das aber nur die Schnitte benötigt, die tatsächlich auf der Dreieckskante stattfinden.

Wir gehen jetzt davon aus, dass wir zu einer Kante mit Eckpunkten $\mathbf{p}$ und $\mathbf{q}$ alle Dreiecke kennen, die die Kante schneiden. Finden wir das Dreieck T, dessen Schnittpunkt mit der Kante am nähesten an $\mathbf{p}$ liegt, können wir an der Normalenrichtung von T ablesen, ob $\mathbf{p}$ im Schnittvolumen liegt. Gleiches gilt für $\mathbf{q}$. Schneidet kein Dreieck die Kante zwischen $\mathbf{p}$ und $\mathbf{q}$, können wir keine Aussage darüber treffen, ob $\mathbf{p}$ und $\mathbf{q}$ innere Punkte sind.

```
Funktion localized_intersection_volume(P_1, P_2)
acc ← 0                                                  › Akkumulatorvariable
foreach triangle T_1 of P_1 do
    foreach triangle side L of T_1 from p to q do
        co ← 0                                           › zählt Schnitte auf der Kante
        mndst ← ∞                                        › kleinste Distanz zum nächsten Dreieck
        ins ← nil                                        › speichert, ob Startpunkt innen liegt
        foreach triangle T_2 of P_2 intersecting L do
            s ← solve_intersection(T_2, L)
            co ← co + 1
            p ← (1 − s) · L.s + s · L.e
            if s < mndst then
                › T_2 liegt näher an L.s als alle Dreiecke davor
                mndst ← s
                › prüfe, ob L.s in Normalenrichtung von T_2 liegt
                if ⟨L.s − p, T_2.n⟩_2 > 0 then
                    ins ← true
                else
                    ins ← false
            evaluate the intersection in p and add the result to acc
        if ins then
            evaluate inner point L.s and add the result to acc
        › bei gerader Anzahl Kantenschnitte liegen L.s und L.t auf derselben
          Seite von P_2
        if ins xor co mod 2 = 0 then
            evaluate inner point L.e and add the result to acc
› Spezialfälle
if there was no intersection then
    if P_1 fully contains P_2 then
        return vol(P_2)
    else if P_2 fully contains P_1 then
        return vol(P_1)
    else
        return 0
else
    use depth-first-search to find all remaining inner points
    evaluate all remaining inner points and add the result to acc
    repeat all above steps once with P_1 and P_2 swapped
    return acc
```

Abb. 4.5.: Verbesserter Algorithmus zur Schnittvolumenberechnung

Wir passen den Algorithmus so an, dass nur noch die Schnitte gespeichert werden, die den beiden Punkten liegen. Zusätzlich merken wir uns die kleinste Distanz von $\mathbf{p}$ zu einem Dreieck T, das die Kante schneidet, und zusätzlich, ob $\mathbf{p}$ auf der Innen- oder Außenseite von T liegt. Dieses Verfahren ist mit dem Tiefentest in Grafikkarten vergleichbar, die neue Information wird auch hier nur geschrieben, wenn sie näher am Betrachter liegt. Über die Anzahl der Schnitte, die auf der Kante selbst liegen, können wir am Ende auch $\mathbf{q}$ klassifizieren.

Alle Punkte, die nicht an eine geschnittene Kante angrenzen, sind zu diesem Zeitpunkt noch nicht klassifiziert. Liegt aber ein unklassifizierter Punkt neben einem inneren Punkt, muss dieser auch ein innerer Punkt sein. Wir können mit dieser Regel unter Verwendung einer Tiefen- oder Breitensuche auf dem durch das Dreiecksnetz gegebenen Graphen alle verbleiben Punkte klassifizieren.

Es kann passieren, dass sich die beiden Netze in keinem Dreieck schneiden. In diesem Fall bleiben alle Eckpunkte unklassifiziert. Wir führen dann je einen Strahltest für beide Netze durch, um herausfinden, ob ein Netz vollständig im anderen enthalten ist oder die eingeschlossenen Volumen disjunkt sind.

Abbildung 4.5 zeigt den verbesserten Algorithmus wieder als Pseudocode.

Diese beiden Algorithmen sind das zentrale Ergebnis dieses Kapitels. Wir gehen in den nächsten Abschnitten auf Details der Implementierung ein.

4.6 Bemerkungen zur Parallelisierung

4.6.1 Parallelisierungsansatz

Für eine CPU-basierte Shared-Memory-Parallelisierung reicht es aus, die Schleife über die Kanten eines Netzes auf die verfügbaren Prozessorkerne aufzuteilen. Die Teilergebnisse können mit logarithmischem Aufwand innerhalb des geteilten Hauptspeichers reduziert werden. Die Zählung und Auswertung der Dreiecksschnitte auf einer Kante erfolgt dann jeweils vollständig auf derselben Recheneinheit.

Unsere Implementierung enthält eine einfache, nicht weiter optimierte Umsetzung dieses Konzepts auf Basis von OpenMP[28]. Wir erreichen hiermit auf einem Dual-Core-System eine ungefähre Halbierung der Laufzeit. Da dieser Parallelisierungsansatz bis auf die Reduktion am Ende nicht auf Kommunikation oder Synchronisation zwischen den Prozessorkernen angewiesen ist, erwarten wir, dass unsere Implemen-

tierung auch auf CPUs mit mehr als zwei Kernen entsprechende Speedups erfährt. Unsere Messungen in Abschnitt 4.10 bestätgen diese Annahme.

Für die Implementierung auf der GPU verwenden wir CUDA[24]. Die hier präsentierten Konzepte sollten sich aber problemlos auf andere Schnittstellen zur GPU-Programmierung wie OpenCL[16] oder AMDs ROCm[1] übertragen lassen. Unabhängig von der verwendeten Schnittstelle teilen sich alle Grafikprozessoren die Einschränkung, dass CPU-Speicher und GPU-Speicher physisch getrennt sind, alle von der GPU benötigten Daten müssen also irgendwann an diese übertragen werden.

4.6.2 Einschränkungen bei GPU-Parallelisierung

CUDA-fähige Grafikprozessoren verwenden im Gegensatz zum Hauptprozessor wenige hundert bis mehrere tausend Rechenkerne, die aber dafür einen kleineren Instruktionssatz haben. Einige Generationen spezialisieren die Rechenkerne zudem auf eine bestimmte Fließkommapräzision. Die Rechenkerne werden dann in *streaming multiprocessors (SMs)* zusammengefasst. Ein SM enthält eine feste Anzahl FP32-Kerne, üblicherweise 32 oder 64, eine feste Anzahl FP64-Kerne, die über die GPU-Generationen stark variiert, und zusätzlich dazu eine große Anzahl an Registern sowie einen eigenen Cache und einen kleinen nicht mit anderen SMs geteilten Speicherbereich. SMs arbeiten bis zur Pascal-Generation strikt nach dem *SIMT*-Prinzip (Single Instruction, Multiple Threads), was in den meisten Aspekten mit Vektorregistern nach dem SIMD-Prinzip auf der CPU vergleichbar ist. Konkret bedeutet das, dass jede Recheneinheit im SM entweder dieselbe Instruktion ausführt oder nichts tut[32]. Anders als bei SIMD-Instruktionen, bei denen immer nur zusammenhängende Speicherbereiche bearbeitet werden können, hat jeder Rechenkern im SM theoretisch die Möglichkeit, einen beliebigen Speicherbereich anzufordern und zu bearbeiten. In dem Fall muss dann für jeden Rechenkern ein eigener Speicherzugriff stattfinden, was die Ausführung deutlich verlangsamen kann. Greifen die Rechenkerne auf aufeinanderfolgende Speicherbereiche oder alle auf dieselbe Adresse zu, kann die Anfrage mit einem einzigen zusammenhängenden Speicherzugriff bearbeitet werden.

Das CUDA-Programmiermodell basiert analog zur CPU-Parallelisierung auf Threads. GPU-Threads unterscheiden sich insofern von CPU-Threads, als dass diese schneller erzeugt und schneller vom Scheduler ausgetauscht werden können[32]. Threads werden in Blöcke gruppiert, die jeweils einem SM auf der GPU zugewiesen werden. Die Größe und Anzahl der Blöcke kann vom Programmierer festgelegt werden, aber die maximal verfügbare Blockgröße und -anzahl hängen von der Generation der GPU ab[25]. CUDA erlaubt Indizierung der Blöcke in bis zu drei Dimensionen, um das

Arbeiten mit mehrdimensionalen Datenstrukturen zu vereinfachen. Innerhalb eines Blocks existieren Befehle zur Synchronisation, außerdem teilen sich alle Threads eines Blocks den SM-eigenen Speicherbereich. Zugriffe auf diesen Speicherbereich sind schneller als Zugriffe auf den globalen Speicher der GPU und können zum Datenaustausch zwischen Threads innerhalb eines Blocks eingesetzt werden. Kommunikation und Synchronisation zwischen Blöcken ist im ursprünglichen Modell nicht vorgesehen, um Skalierbarkeit zu gewährleisten, aber seit CUDA 9 mittels *cooperative groups* möglich. 32 aufeinanderfolgende Threads werden zu einem *Warp* zusammengefasst. Die 32 Threads in einem Warp werden auf die Rechenkerne eines SMs verteilt und immer gleichzeitig ausgeführt. Für bestmögliche Nutzung der SMs sollten Warpgrenzen mit Blockgrenzen zusammenfallen, ein Vielfaches von 32 ist für die Blockgröße optimal.

Wir beschreiben jetzt, wie mit diesen Einschränkungen die Algorithmen aus Abschnitt 4.5 auf einer GPU parallelisiert werden können. Wir gehen auch kurz auf die von uns implementierten Optimierungen ein, bemerken aber an dieser Stelle, dass noch viele weitere Optimierungen möglich sind.

4.6.3 Umsetzung der GPU-Parallelisierung

Damit die GPU optimal ausgelastet ist, weisen wir jedem einzelnen Thread genau einen Schnitt zwischen einer Kante und einem Dreieck zu. Findet ein Schnitt statt, wertet der Thread diesen direkt aus. Wir verwenden eine zweidimensionale Indizierung der Threadblöcke und zählen die Kanten entlang der x-Dimension und die Dreiecke entlang der y-Dimension auf. Aufeinanderfolgende Threads greifen auf aufeinanderfolgende Daten zu. Anders als bei der CPU-Parallelisierung müssen wir jetzt auch die Schnittzähler entlang einer Kante reduzieren. Hierzu wäre es theoretisch denkbar, dass jeder Thread sein Ergebnis in den Speicher schreibt und dann eine gewöhnliche Reduktion ausgeführt wird. In der Praxis steht für ausreichend große Eingaben dafür nicht genug Speicher zur Verfügung. Mit der Beobachtung, dass ein großer Anteil aller möglichen Schnitte zwischen Kanten und Dreiecken nicht stattfindet, verwenden wir stattdessen einen atomaren Zähler für die Reduktion. Für die Reduktion der Schnittzahlen entlang der Kanten verwenden wir ein Array mit einem Zähler für jede Kante, der ebenfalls atomar inkrementiert wird. Bei mehrdimensionalen Threadblöcken fasst CUDA die Threads entlang der x-Dimension zu Warps zusammen, also entlang der Kanten. Dadurch bearbeitet jeder Thread im Warp eine andere Kante und innerhalb eines Warps wird nie zweimal auf denselben atomaren Zähler zugegriffen. Konflikte mit anderen zur gleichen Zeit ausgeführten Blöcken lassen sich jedoch nicht ausschließen.

Für die Auswertung der Kantenschnitte müssen wir davon ausgehen, dass jede Auswertung ein von Null verschiedenes Ergebnis liefern kann. Das ist etwa dann der Fall, wenn ein Netz vollständig im anderen Netz enthalten ist. Wir gehen hier in zwei Stufen vor. CUDA erlaubt begrenzten Datenaustausch der Threads innerhalb eines Warps durch sogenannte *Warp-Shuffles*. Wir können also mit Warp-Shuffles die Ergebnisse zunächst innerhalb eines Warps so austauschen, dass ein Thread die Summe der Ergebnisse speichert. Diese werden in einem zweiten Schnitt in den globalen Speicher geschrieben und dort unter Verwendung der Thrust-Bibliothek[26] blockübergreifend summiert. Die Summierung mit Warp-Shuffles sorgt dafür, dass wir nach diesem Schritt die Zwischenergebnisse bereits um einen Faktor von 32 reduziert haben, ohne einen zeitaufwendigen Speicherzugriff auszuführen. Die Auswertung der Kantenschnitte hat nur lineare Laufzeit, könnte also auch problemlos sequenziell auf der CPU ausgeführt werden. Die Nutzung der GPU erspart es uns an dieser Stelle, das Array aller Schnittzähler in den Hauptspeicher zu übertragen.

Bisher sind wir davon ausgegangen, dass die verwendete GPU immer die benötigte Anzahl an Threads bereitstellen kann. Für die x-Dimension ist das Blocklimit[25] von $2^{31} - 1$ für alle realistischen Eingaben effektiv unerreichbar, für die y-Dimension ist das derzeitige Limit von 65535 deutlich einfacher zu erreichen. Bei einer Blockgröße von 32×32 reicht hier ein Netz mit ungefähr 2.1 Millionen Dreiecken aus. Damit das Programm auch für diese Eingaben funktioniert, bearbeiten wir in diesem Fall mehrere Dreiecke in einem Thread.

Da für Anwendungen in der 3D-Grafik üblicherweise mit einfacher Präzision statt doppelter Präzision gerechnet wird, unterstützen ältere GPUs einfache Präzision deutlich besser. Atomare Addition mit doppelter Präzision wird auf NVIDIA-Grafikkarten vor Compute-Capability-Version 6.x nicht unterstützt. Wir verwenden in diesem Fall stattdessen einen Compare-and-Swap-Loop. Neuere GPUs können problemlos mit doppelter Präzision umgehen, alle vorgestellten Konzepte gelten hier analog.

4.6.4 Ungelöste Probleme

Für die Parallelisierung des zweiten Algorithmus können wir ähnlich vorgehen, allerdings wirken die Einschränkungen von oben hier deutlich stärker.

Die Klassifikation der verbleibenden Eckpunkte nehmen wir nicht auf der GPU vor. Abgesehen davon, dass sich Tiefen- und Breitensuche in allgemeinen Graphen nur schwer parallelisieren lassen, hat dieser Teil der Berechnung nur lineare Komplexität in der Größe der Eingabenetze, wirkt sich also asymptotisch nicht auf die Laufzeit aus.

Weiterhin müssen wir durch die Verwendung einer räumlichen Datenstruktur weniger Schnitte zwischen Dreiecken überprüfen, dafür steigt aber die Wahrscheinlichkeit, dass die verbleibenden Schnitte tatsächlich stattfinden. Unsere Reduktion der Teilergebnisse über atomare Zähler, die darauf basiert, dass ein Großteil der Schnitte zu Null evaluiert wird, ist hier also nicht sinnvoll. Sie ist auch überhaupt nicht mehr möglich, da die Reduktion diesmal nicht nur aus einer Summation der Teilergebnisse besteht.

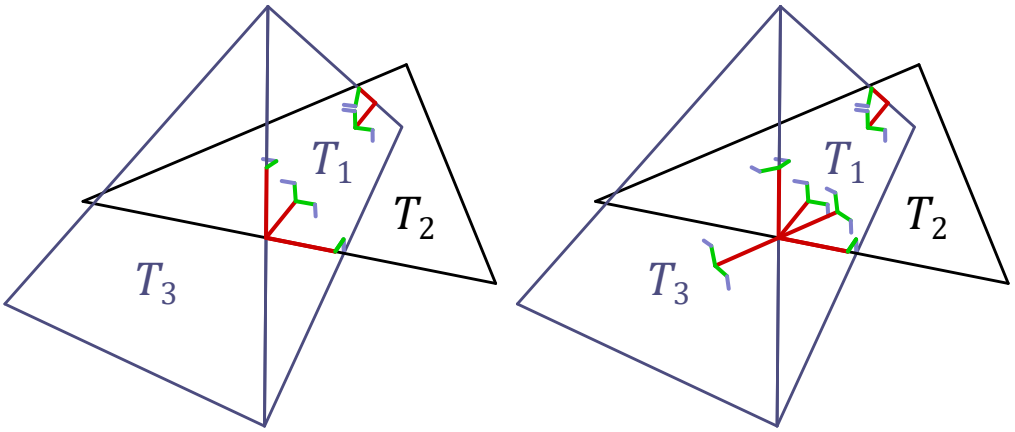
Wir können zur Lösung dieses Problems einerseits die Reduktion mit expliziter Synchronisation im globalen Speicher ausführen. Dadurch werden automatisch komplexere Reduktionsoperationen ermöglicht. Zum anderen können wir stattdessen die Parallelisierung vergröbern und wie bei der CPU-Parallelisierung jedem Thread eine Dreieckskante zuweisen. Bei ausreichend großen Eingaben wäre hiermit immer noch gewährleistet, dass die Anzahl der Threads die Anzahl der Rechenkerne um ein Vielfaches übersteigt.

Für beide Lösungsansätze ist noch nicht klar, wie gut diese Art der Parallelisierung mit räumlichen Datenstrukturen auf der GPU zusammenspielt. Wir gehen davon aus, dass umfangreiche Versuche notwendig sind, um eine geeignete Kombination aus einer GPU-fähigen Datenstruktur und einem vollständig damit kompatiblen Algorithmus zu finden. Aus diesem Grund sehen wir dieses Problem als noch ungelöst an.

4.7 Degenerierte Fälle

Wir gehen nun darauf ein, wieso wir die am Anfang des Kapitels gestellte Forderung, dass keine vier Punkte koplanar liegen dürfen, für ein korrektes Ergebnis voraussetzen und wie wir sicherstellen, dass das Netz diese erfüllt. Wir betrachten dafür drei Fälle von Degeneriertheiten, die bei der Schnittvolumenberechnung auftreten können.

I. Zusammenfallende Eckpunkte Nach der mathematischen Definition eines Polyeders kommt jeder Eckpunkt exakt einmal vor. Die Repräsentation als Dreiecksnetz erlaubt es aber, dass zwei Eckpunkte eines Dreiecks zusammenfallen können. Hier ist eine Tangentialrichtung nicht sinnvoll definierbar, in Gleichung 3.3.1 wird in diesem Fall durch Null dividiert. Dieses Problem lässt sich verhältnismäßig leicht beheben, indem wir degenerierte Dreiecke in einzelnen Meshes als Teil eines Preprocessing-Schritts entfernen.



(a) Alle Quadrupel, die aus dem Schnitt der Dreiecke abgeleitet werden können

(b) Alle Quadrupel, die durch den Algorithmus erzeugt werden

Abb. 4.6.: Zwei Dreiecksnetze schneiden sich exakt an einer Kante. Die Abbildung zeigt drei der vier in der Schnittberechnung für diese Kante involvierten Dreiecke. Die beiden blauen Dreiecke sind nicht koplanar.

II. Exakter Schnitt zweier Kanten Abbildung 4.6 visualisiert den Fall, dass sich zwei Dreieckskanten exakt treffen. Das hat zur Folge, dass, weil alle Dreiecke separat gespeichert werden, jede Kante immer zwei Kanten auf einmal schneidet und die Endpunkte der Kante fehlerhaft klassifiziert werden. Wir können zumindest für dieses Teilproblem nach Feito und Torres[5] vorgehen und für beide Kanten nur einen halben Kantenschnitt zählen. Da Schnittpunkt und Schnittgerade zwischen T_1 und T_2 exakt gleich liegen, werden auch die Quadrupel entlang der Schnittkante doppelt generiert. An der Abbildung lässt sich weiterhin ablesen, dass zwischen T_2 und T_3 kein tatsächlicher Schnitt stattfindet. T_3 berührt T_2 nur an einer Kante und schneidet stattdessen das unmittelbar an T_2 angrenzende Dreieck. Der Algorithmus kann diese Situation aber lokal nicht erkennen und wertet den Schnitt der Dreiecksseiten wie einen gewöhnlichen Dreiecksschnitt aus. Nehmen wir das an T_2 angrenzende Dreieck hinzu, werden noch weitere Quadrupel analog dazu inkorrekt erzeugt.

III. Koplanare Dreiecke Liegen zwei Dreiecke aus verschiedenen Netzen koplanar, ist der Schnitt möglicherweise nicht mehr durch nur eine Gerade darstellbar. Die Klassifikation der Schnittebene führt viele Sonderfälle ein[31] und erfordert in einigen dieser Fälle die Auswertung von sechs Quadrupeln für nur diesen Schnitt. Koplanare Dreiecke erlauben zudem einen degenerierten Schnittkörper, der nur aus Seitenflächen besteht.

Alle drei Fälle haben gemeinsam, dass jeweils mindestens vier Eckpunkte koplanar liegen. Wenn wir diesen Fall im Voraus ausschließen können, kann keine der diskutieren Degeneriertheiten eintreten. Offensichtlich erfüllen nicht alle möglichen Eingabenetze diese Voraussetzung. Das ergibt sich insbesondere daraus, dass es Kör-

per gibt, die über Koplanarität ihrer Eckpunkte definiert sind, zum Beispiel Würfel oder die in Kapitel 3 angesprochenen Parallelepipede.

Wir lösen dieses Problem durch eine numerische Perturbation der Eckpunkte. Wir verrauschen jeden Eckpunkt ab einer fest gewählten Nachkommastelle k durch Addition einer zufällig gleichverteilt gezogenen Zahl in der entsprechenden Größenordnung. Dadurch tritt der Fall, dass vier Punkte koplanar liegen, nur noch mit einer sehr geringen Wahrscheinlichkeit ein. Für die meisten Tests haben wir bereits mit $k = 10$ korrekte Ergebnisse erhalten. In besonders degenerierten Fällen wie dem Schnitt eines Netzes mit sich selbst wählen wir $k = 5$.

Um den Einfluss der Perturbation auf das Ergebnis zu minimieren, zentrieren wir die Netze vor der Schnittvolumenberechnung um den Koordinatenursprung, indem wir den Durchschnitt aller Eckpunkte von jedem Eckpunkt abziehen. Dieses Verfahren lässt noch einige Verbesserungen zu, zum Beispiel stellen wir die Robustheit gegen Ausreißer in Frage. Wir gehen aber davon aus, dass sich für einen hohen Anteil der möglichen Eingaben das Ergebnis nicht verschlechtert.

Franklin und Magalhães[8, 18, 20] behandeln für die exakte Berechnung des Schnitts nicht jeden Sonderfall einzeln, sondern verwenden ein symbolisches Perturbationsverfahren namens *Simulation of Simplicity*[4]. Tritt ein degenerierter Fall ein, werden die betroffenen Eckpunkte symbolisch durch speziell gewählte Infinitesimale verrauscht. Die Infinitesimale sind immer größer als Null, aber kleiner als jede reelle Zahl. Damit auch Abstufungen innerhalb der Infinitesimale möglich sind, werden verschiedene Ordnungen von Infinitesimalen definiert. Die Verwendung von Infinitesimalen hat zur Folge, dass analog zur numerischen Perturbation exakte Schnitte konsistent zugunsten einer Seite entschieden werden können.

Symbolische Perturbation ist aufgrund der Eigenschaften von Infinitesimalen nicht auf inexakte Rechnungen übertragbar. Wir sehen deshalb unser Vorgehen als die bestmögliche Annäherung des exakten Umgangs mit Degeneriertheiten an.

4.8 Implementierungsdetails

Die Referenzimplementierung ist in der Programmiersprache C++ geschrieben und verwendet GLM[11] als Bibliothek für Vektorarithmetik. Für unsere Berechnung verwenden wir für alle Berechnungen doppelte Präzision, da sich einfache Präzision in den ersten Tests als unzureichend genau herausgestellt hat. Die Implementierung lässt sich aber durch Definition eines entsprechenden Makros vollständig auf einfache Präzision umstellen.

Wir fassen die drei Eckpunkte eines Dreiecks nach dem Einlesen einer STL-Datei in einem `struct` zusammen. Da wir im Verlauf der Berechnung häufig auf den Normalenvektoren der Dreiecke zugreifen, berechnen wir diese vor und fügen sie dem Dreieck-`struct` hinzu. Wir speichern alle so gewonnenen `structs` hintereinander in einer Form von Array, beispielsweise `std::vector`. Für die Speicherverwaltung auf der GPU nutzen wir `thrust::device_vector` aus der mit CUDA mitgelieferten Thrust-Bibliothek[26], speichern die Daten aber im gleichen Format. Wir gehen allerdings davon aus, dass effizientere Speicherzugriffe möglich sind, wenn wir die Koordinaten in separate Arrays aufteilen (SOA-Format).

Da wir Eckpunkte redundant speichern, können wir diese nicht problemlos perturbieren, da sonst derselbe Eckpunkt auf unterschiedliche Art verschoben werden könnte. Wir müssen deshalb für die Anwendung der Perturbation gleiche Eckpunkte zusammenfassen. Wir nutzen hierzu eine Hashtabelle, verwenden für die Hashfunktion aber nur die ersten paar Nachkommastellen, um Rauschen in der Eingabe zu kompensieren. Nach der Unifizierung perturbieren wir die Eckpunkt wie in Abschnitt 4.7 beschrieben und trennen diese anschließend wieder in die einzelnen Dreiecke auf.

Die Implementierung ist so strukturiert, dass sich die Codebasis sowohl mit einem gewöhnlichen `C++`-Compiler als auch mit dem CUDA-Compiler erstellen lässt. Wird ein CUDA-Compiler erkannt, so wird automatisch die GPU-beschleunigte Version des Schnittalgorithmus kompiliert. Beide Versionen verwenden intern dieselben Funktionen zur Auswertung von Dreiecksschnitten, der Unterschied besteht nur darin, wie die Auswertung auf die verfügbaren Recheneinheiten aufgeteilt wird.

4.9 Visualisierung der Szene

Um eine schnelle, informelle Überprüfung der generierten Quadrupel zu ermöglichen, stellen wir zusätzlich zur eigentlichen Implementierung eine statische Programmbibliothek zur Visualisierung der Quadrupel auf Basis von OpenGL[33] und dem Qt-Framework[34] bereit. Dieses Werkzeug ermöglicht es, die 3D-Szene mithilfe einer First-Person-Kamerasicht zu navigieren.

Das Visualisierungsprogramm rendert Dreiecksnetze als Wireframes, damit die Dreiecke nicht als opake Flächen die Sicht auf die Innenseite der Netze einschränken. Ohne diese Anpassung wäre es schwieriger, die korrekte Behandlung innerer Punkte zu überprüfen.

Die Quadrupel werden als Punktwolke mit Attributen gerendert. Ein Geometry-Shader transformiert die Attribute während des Rendervorgangs in farbige Linienprimitive. Mit dieser Methode können die Linien, die die Tangenten-, Flächen- und Normalenrichtung anzeigen, auch nachträglich skaliert werden, ohne dass die Daten erneut von der CPU auf die GPU übertragen werden müssen.

Zusätzlich erlaubt das Visualisierungsprogramm das Exportieren der aktuellen Kamerasicht im PNG- und SVG-Format sowie das Abspeichern und Wiederherstellen der Kameraausrichtung. Viele der in dieser Arbeit verwendeten Vektorgrafiken wurden mithilfe dieses Programms erzeugt.

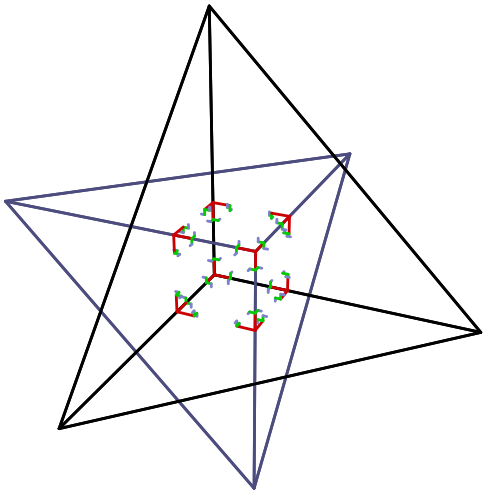
4.10 Ergebnisse

4.10.1 Korrektheit

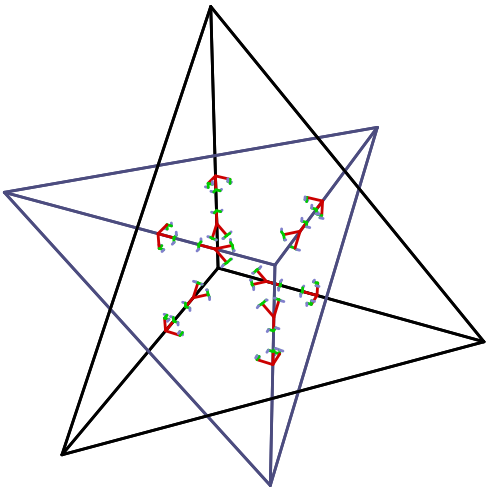
Wir testen unsere Implementierung zunächst in einer einfachen 3D-Szene, in der sich zwei durch Rotation auseinander hervorgehende Tetraeder schneiden. Wir verifizieren die Ergebnisse mit Rhino3D[30] und durch visuelle Inspektion der generierten Quadrupel.

Wir können die Tetraeder so gegeneinander verschieben, dass jeder Schnitt ein exakter Kantenschnitt ist. Abbildung 4.7c zeigt, wie die Quadrupel in diesem Fall aussehen müssten. Der Algorithmus erzeugt aber die in 4.7d visualisierten Quadrupel, da jeder Schnitt degeneriert ist. Insbesondere wird hier sichtbar, dass der Algorithmus einige der äußeren Punkte fälschlicherweise als innere Punkte klassifiziert und diese entsprechend zusätzlich auswertet. Abbildung 4.7e zeigt den Schnitt nach Anwendung einer Perturbation auf die Eckpunkte. Einige Quadrupel entlang der Kanten scheinen hier ins Äußere des Schnitts zu zeigen. Das ist ein Artefakt der Darstellung und entsteht dadurch, dass die Kanten die jeweils andere Kante knapp verfehlen und stattdessen die zwei angrenzenden Flächen schneiden. In der Berechnung löschen sich diese Quadrupel mit denen ihnen entgegengesetzten Quadrupeln aus.

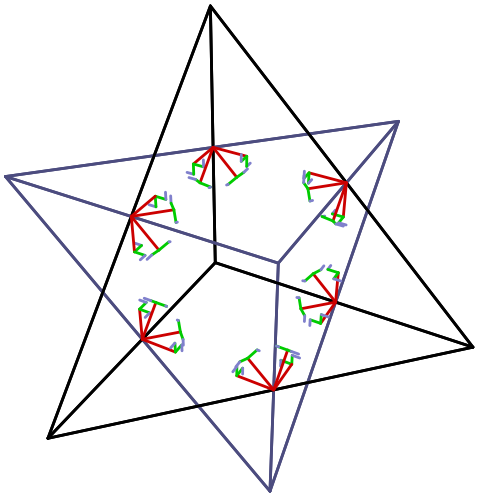
Abbildungen 4.7b und 4.7f zeigen einen Schnittkörper, der alle Seitenflächen der Eingabenetze zu einem Teil enthält. Jede mögliche Triangulierung der beiden Körper enhält mehr Dreiecke als beide Eingabenetze zusammen. Da unser Verfahren aber keine Triangulierung des Schnittkörpers benötigt, hat diese Tatsache keinen Einfluss auf das Verhalten des Algorithmus. Einer der Schnittkörper wird in Abbildung 4.8 zur Veranschaulichung aus unterschiedlichen Perspektiven dargestellt.



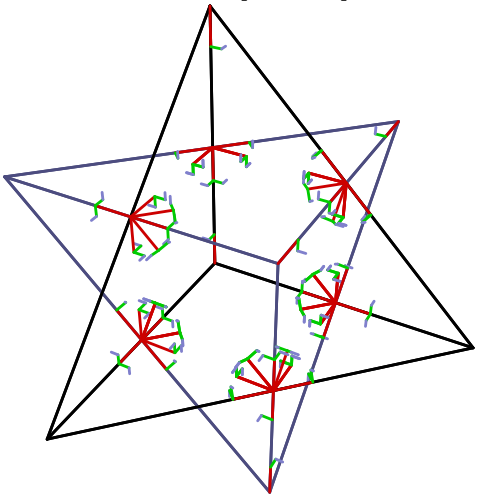
(a) Verschiebung um $\begin{bmatrix}3 & 3 & 3\end{bmatrix}$



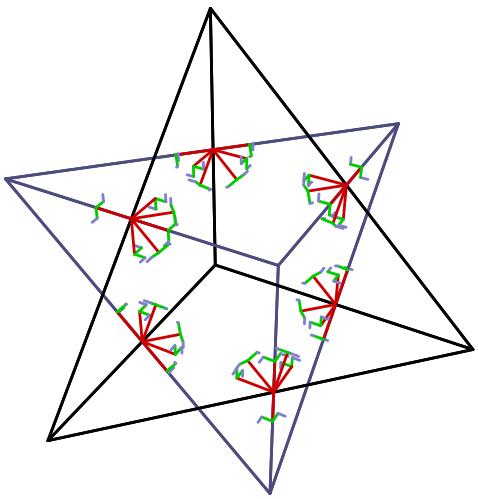
(b) Verschiebung um $\begin{bmatrix}4 & 4 & 4\end{bmatrix}$



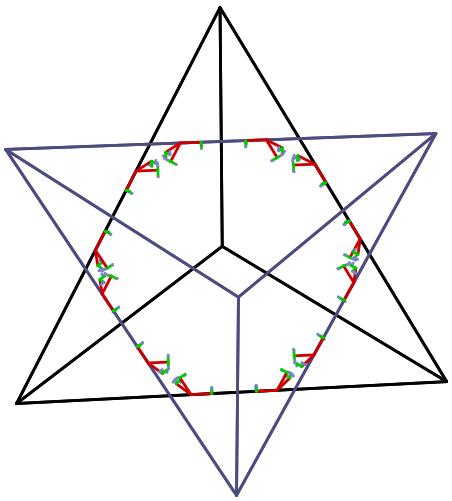
(c) Verschiebung um $\begin{bmatrix}5 & 5 & 5\end{bmatrix}$ mit exakter Berechnung



(d) Verschiebung um $\begin{bmatrix}5 & 5 & 5\end{bmatrix}$ ohne Perturbation der Eckpunkte

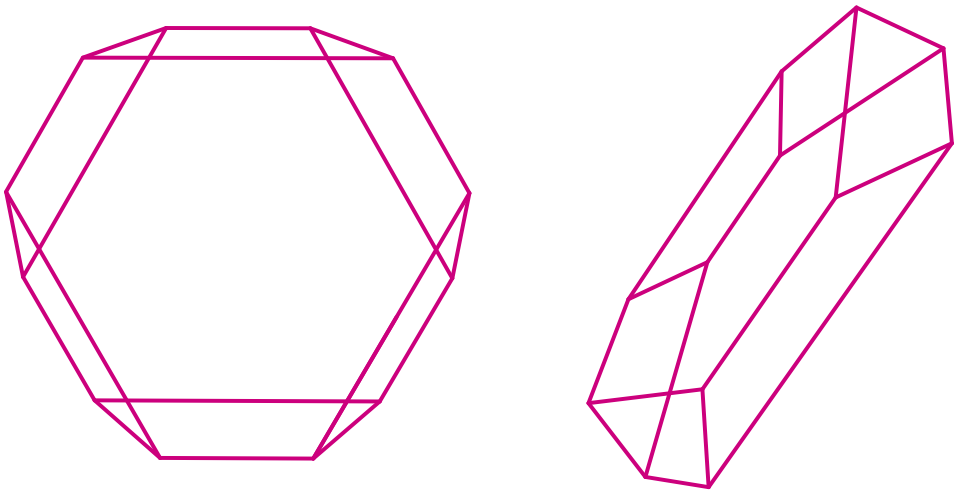


(e) Verschiebung um $\begin{bmatrix}5 & 5 & 5\end{bmatrix}$ mit Perturbation der Eckpunkte



(f) Verschiebung um $\begin{bmatrix}6 & 6 & 6\end{bmatrix}$

Abb. 4.7.: Visualisierung der Quadrupel bei verschiedenen Schnitten zweier Tetraeder mit Seitenlänge 10. Die Tetraederspitzen wurden jeweils relativ zueinander verschoben.



(a) Frontalansicht (b) Seitenansicht

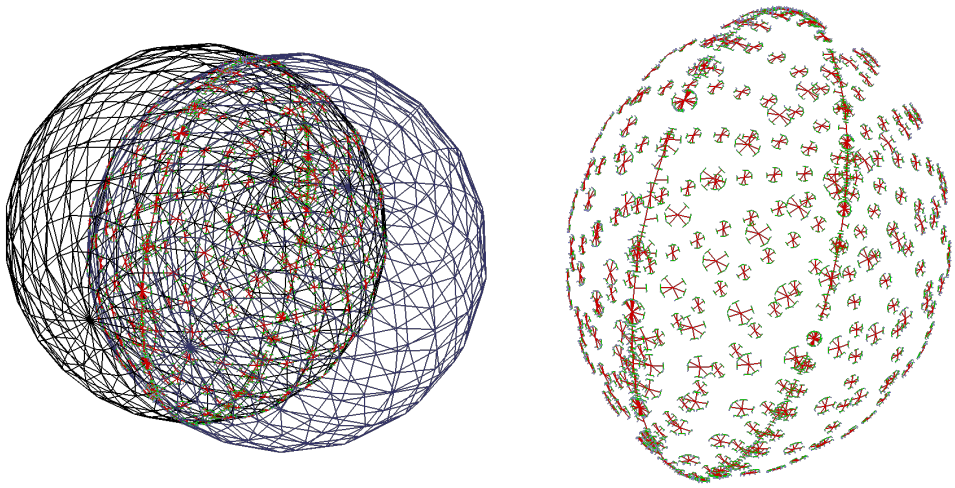
Abb. 4.8.: Wireframe-Darstellung des Schnittkörpers aus 4.7f. Der Schnittkörper wird durch zwei Sechsecke und sechs Vierecke begrenzt. Eine Triangulierung dieses Körpers würde mindestens 20 Dreiecke benötigen, also mehr Dreiecke als die beiden Eingabenetze zusammen.

	Laufzeit			**relativer Fehler**	
Dreiecke	sequenziell	OpenMP	CUDA	Rhino3D	Formel
180	5.2 ms	3.1 ms	11.5 ms	3.9%	7.3%
760	42.3 ms	14.6 ms	12.8 ms	$< 0.001\%$	2.8%
1740	224.5 ms	50.0 ms	25.9 ms	0.018%	1.2%
4900	1729.1 ms	339.5 ms	126.3 ms	0.3%	0.8%
19800	23.3 s	3.7 s	1.8 s	7.1%	7.1%
79600	222.6 s	35.1 s	22.2 s	-	88.8%

Tab. 4.1.: Benötigte Zeit für die Schnittvolumenberechnung mit einem Abstand von 0.5 zwischen den Kugelmittelpunkten. Das exakte Schnittvolumen beträgt 2.6507.

4.10.2 Benchmarks

Für unsere Benchmarks berechnen wir das Schnittvolumen zwischen zwei identischen Tessellierungen der Einheitskugel bei unterschiedlichen Abständen der Mittelpunkte. Dadurch können wir die Anzahl der beteiligten Dreiecke durch die Qualität der Tessellierung kontrollieren. Wir verifizieren die Ergebnisse wieder mit Rhino3D. Wir können außerdem die Ergebnisse zusätzlich damit verifizieren, dass das Ergebnis mit zunehmender Qualität der Tessellierung gegen das Schnittvolumen der beiden Einheitskugeln konvergieren muss, für das sich eine geschlossene Formel herleiten lässt. Abbildung 4.9 visualisiert eine mögliche Testanordnung. Für die Abstände der Mittelpunkte wählen wir 0.5 (Tabelle 4.1) und 1.5 (Tabelle 4.2). Die Testfälle unterscheiden sich dadurch insbesondere in der Anzahl der Punkte, die im jeweils anderen Netz enthalten sind.



(a) Schnitt zweier Einheitskugeln mit jeweils 760 Seitenflächen

(b) Darstellung der für die Schnittberechnung benötigten Quadrupel

Abb. 4.9.: Veranschaulichung des für die Benchmarks verwendeten Kugelschnitts

	Laufzeit			relativer Fehler	
Dreiecke	sequenziell	OpenMP	CUDA	Rhino3D	Formel
180	3.4 ms	7.6 ms	9.8 ms	35 %	4.7%
760	47.7 ms	15.0 ms	14.2 ms	< 0.001%	6.0%
1740	221.9 ms	44.9 ms	25.7 ms	0.07%	3.6%
4900	1682.9 ms	306.5 ms	124.9 ms	0.008%	1.3%
19800	23.3 s	3.7 s	1.8 s	70.8%	70.9%

Tab. 4.2.: Benötigte Zeit für die Schnittvolumenberechnung mit einem Abstand von 1.5 zwischen den Kugelmittelpunkten. Das exakte Schnittvolumen beträgt 0.3599.

Wir testen die Implementierung mit einem Intel i7 6800K als CPU und einer NVIDIA GeForce GTX 1080 als GPU. Die Zeitmessung wird mit den Funktionen aus `std::chrono` realisiert und beinhaltet nur die Ausführung der Funktion zur Schnittvolumenberechnung. Das Einlesen der Daten wird also nicht mitgemessen, dafür aber die Zeit für die Allokierung und Initialisierung benötigter Hilfsdatenstrukturen sowie die Kernel-Aufrufe für den CUDA-Teil. Wird das Programm als Teil eines größeren Verfahrens eingesetzt, fällt dieser zusätzliche Overhead normalerweise nur einmal an, wir messen hier also eine obere Schranke für die Laufzeit.

Zusäzlich zur Laufzeit bestimmen wir den relativen Fehler, indem wir die Differenz von unserem Ergebnis und dem Referenzergebnis berechnen und diese durch das Referenzergebnis teilen. Wir mitteln alle Ergebnisse jeweils über 10 konsekutive Ausführungen.

Aus den Testergebnissen lässt sich zunächst die quadratische Komplexität des unoptimierten Algorithmus erkennen. Wir können auch sehen, wie sich das Schnittvolumen der beiden Netze immer weiter dem tatsächlichen Schnittvolumen annähert, aber ab

einer bestimmten Dreieckszahl der relative Fehler zunimmt. Da dieser Fehler auch bei stärkerer Perturbation der Eckpunkte nicht zurückgeht, führen wir das falsche Ergebnis nicht auf mögliche Degeneriertheiten, sondern auf numerische Instabilität der Berechnung zurück.

Betrachten wir nicht das tatsächliche Schnittvolumen, sondern das von Rhino3D ebenfalls durch Schnitte von Dreiecksnetzen berechnete Volumen als Referenzergebnis, fluktuiert der relative Fehler deutlich stärker. Da ohne Perturbation der Eingaben beide Berechnungen dasselbe Ergebnis liefern müssten, gehen wir hier davon aus, dass der Fehler durch die Perturbation entsteht. Der fehlende Wert in Tabelle 4.1 kommt daher, dass die Berechnung des Schnittkörpers in Rhino3D für diese Eingabe fehlgeschlagen ist.

Für die jeweils letzten korrekten Ergebnisse der beiden Messreihen erreichen wir ein Speedup zwischen 13 und 14 gegenüber dem sequenziellen Algorithmus. Wir konnten somit erfolgreich zeigen, dass GPU-Parallelisierung für unser Verfahren zur Schnittvolumenberechnung in Frage kommt. Unter Betrachtung der Tatsache, dass eine GTX 1080 über 2560 CUDA-fähige Rechenkerne verfügt, stellen wir aber auch fest, dass deutlich höhere Speedups möglich sind. Die weitere Optimierung der Implementierung ist somit ein Thema zukünftiger Nachforschungen.

5 Zusammenfassung

5.1 Beiträge dieser Arbeit

In dieser Arbeit haben wir die Gültigkeit der Volumenformel nach Franklin bewiesen, indem wir die Gleichheit mit der klassischen Volumenformel termweise gezeigt haben. Aus der Formel konnten wir ableiten, dass sich aus dem Schnitt einer Dreieckskante mit einem Dreieck drei Summanden der Volumenformel ergeben. Die übrigen Summanden resultieren aus den Punkten, die in beiden Netzen enthalten sind. Durch sequenzielles oder paralleles Abarbeiten aller möglichen Schnitte zwischen Kanten und Dreiecken ergibt sich ein einfaches Berechnungsverfahren, das sich auch leicht auf einer GPU implementieren lässt. Für die Reduktionen verwenden wir atomare Operationen, wenn das Teilergebnis mit hoher Wahrscheinlichkeit Null ist.

Darüber hinaus haben wir einen weiteren Algorithmus vorgestellt, der für die Entscheidung, ob die Endpunkte einer Kante im anderen Volumen enthalten sind, auch ohne Strahltest auskommt. Das bildet die Grundlage für die Verwendung räumlicher Datenstrukturen, mit denen wir einen Teil der Schnitte im Voraus ausschließen können. Wir perturbieren außerdem die Eckpunkte der Eingabenetze, damit während der Schnittberechnung keine geometrischen Degeneriertheiten eintreten. Zuletzt zeigen wir eine Möglichkeit auf, wie die zur Volumenberechnung notwendigen Terme visualisiert werden können.

Zum Zeitpunkt der Erstellung dieser Arbeit existiert nach unserem Wissen keine Veröffentlichung, die die Volumenformel nach Franklin zur Beschleunigung der Berechnung auf einer GPU einsetzt. Viele Anwendungen im CAD-Bereich erfordern exakte Berechnungen oder eine explizite Konstruktion des Schnittkörpers und schränken deshalb den Einsatzbereich inexakter Berechnungsverfahren auf Grundlage dieser Formel ein. Wenn eine Anwendung aber nur eine Approximation des Schnittvolumens benötigt, haben wir gezeigt, dass die nötige Berechnung durch Verwendung eines Grafikprozessors beschleunigt werden kann. Der Quellcode unserer Implementierung steht unter `https://github.com/slyphiX/MeshIntersectionVolume` zur Verfügung. Wir gehen davon aus, dass für die Berechnung des Schnittvolumens mit den entsprechenden Optimierungen und unter Verwendung aktueller Grafikprozessoren Speedups über 100 durchaus möglich sind.

5.2 Ausblick

Schnitte mehrerer Dreiecksnetzpaare Wir gehen in Kapitel 4 nur auf den Fall ein, dass genau zwei Dreiecksnetze gegeben sind. Das Programm kann so erweitert werden, dass zu einer beliebigen Anzahl von Dreiecksnetzen die Summe aller paarweisen Schnittvolumen berechnet wird. Der einzige Unterschied besteht hierbei darin, dass wir die Netze für die gesamte Schnittberechnung nur einmal auf die GPU übertragen müssen.

Datenstrukturen auf der GPU In Kapitel 4 zählen wir einige räumliche Datenstrukturen auf und bemerken, dass sich möglicherweise nicht alle davon für die Verwendung auf Grafikprozessoren eignen. Es steht noch aus, diese Vermutung experimentell zu überprüfen. Wir halten es für realistisch, dass die Suche nach GPU-optimierten Datenstrukturen (analog zur Enwicklung sekundärspeicheroptimierter Datenstrukturen wie B-Bäume) in der Zukunft an Relevanz gewinnt.

Erhöhte Präzision Wir können für unsere Berechnung eine höhere Anzahl an signifikanten Stellen erreichen, indem wir mehr Bits zur Repräsentation der Operanden wählen. Der IEEE-Standard[14] für Fließkommaarithmetik sieht zwar einen 128-bit-Datentyp vor, dieser ist aber bisher nicht in Hardware, sondern nur als Compilererweiterung[35] implementiert. Stattdessen können wir eine unevaluierte Summe von zwei oder vier Zahlen doppelter Genauigkeit nutzen, um zumindest einen größeren Wertebereich für die Mantisse zu erhalten[13]. Wir erwarten, dass Rechnungen in dieser Darstellung aufgrund der konstanten Größe des Datentyps immer noch zu merklichen Speedups auf Grafikprozessoren führen. Auch für arbitrary-precision-Arithmetik stehen mittlerweile Bibliotheken mit GPU-Beschleunigung zur Verfügung[23].

Distributed-Memory-Parallelisierung In dieser Arbeit sind wir nur auf Parallelisierung auf einem einzelnen Rechner eingegangen. Das Verfahren zur Schnittvolumenberechnung funktioniert analog auf mehreren Rechnern, der Unterschied besteht hier darin, dass Speicherbereiche nicht implizit geteilt sind, sondern explizit zwischen Rechnern ausgetauscht werden müssen. Da für die Kommunikation zusätzlicher Aufwand anfällt, lohnt sich Distributed-Memory-Parallelisierung nur dann, wenn die Berechnung weitaus mehr Zeit als der nötige Datenaustausch beansprucht. Es steht noch offen, ob Schnittvolumenberechnung von Distributed-Memory-Parallelisierung profitiert.

Parallelisierung auf mehreren GPUs Mehrere GPUs bilden einen Spezialfall der Distributed-Memory-Parallelisierung, da sich diese den physischen Speicher im Nor-

malfall nicht teilen. Aktuelle Technologien wie NVLINK[27] erlauben hier allerdings Kommunikationsgeschwindigkeiten von bis zu $300\,\mathrm{GB/s}$ zwischen zwei Grafikprozessoren ohne Beteiligung der CPU. Wir können auf diese Weise die Berechnung unabhängig von der Leistung des Hauptprozessors oder dem damit verbundenen Netzwerk durchführen.

Lokale Änderungen Ist die Berechnung des Schnittvolumens Teil einer größeren Berechnung, die eine lokale Verschiebung oder Verformung eines oder mehrerer Netze beinhaltet, können möglicherweise zusätzliche Optimierungen vorgenommen werden. Wenn wir wissen, welche Summanden der Volumenformel sich verändern, können wir die Differenz der vorigen Summanden mit den neuen Summanden berechnen und damit das Volumen adjustieren, ohne alle anderen Schnitte erneut auswerten zu müssen.

Netze ohne vollständige Begrenzungsfläche Für unsere Eingabedaten haben wir gefordert, dass die Dreiecksnetze keine Löcher in der Triangulierung aufweisen dürfen. Eigentlich müssen wir diese Bedingung nur an den Teil der Dreiecksnetze stellen, der Teil des Schnittkörpers ist. Damit ist es möglich, das Programm so zu erweitern, dass beispielsweise Schnitte mit einer Ebene im Raum berechnet werden können. In diesem Fall müsste ein neues Verfahren gefunden werden, um innere Punkte zu identifizieren. In Kapitel 4 haben wir festgestellt, dass der Strahltest in diesem Fall keine konsistenten Ergebnisse liefert.

Schnitte identischer Netze In unserer Implementierung verwenden wir getrennte Buffer, um die Dreiecksnetze zu speichern, selbst dann, wenn alle Netze identisch sind. Für den Fall, dass alle verwendeten Netze durch eine affin-lineare Abbildung aus einem einzelnen Netz hervorgehen, bietet es sich möglicherweise an, das Netz nur einmal zu speichern und die Koordinaten der Eckpunkte jeweils durch Anwendung einer Transformationsmatrix zu errechnen, ähnlich der Verwendung einer Model-Matrix in der Computergrafik. Dieser Ansatz kann zusätzlich auf allgemeine Abbildungen ausgeweitet werden.

Ausgelassene Beweise

A

A.1 Besonderheiten von Skalarprodukt und Kreuzprodukt

Die Definitionen und Eigenschaften des Kreuzprodukts und Standardskalarprodukts setzen wir hier als bekannt voraus (im Zweifel verweisen wir auf das Buch von Fischer[6]). Wir zeigen in diesem Abschnitt einige Eigenschaften dieser Operationen auf, die nicht unmittelbar aus der Definition ablesbar sind.

Lemma A.1.1. *Sei* $\mathbf{u} \in \mathbb{R}^N$. *Dann gilt* $\langle \mathbf{u}, \mathbf{0} \rangle_2 = \langle \mathbf{0}, \mathbf{u} \rangle_2 = 0$.

Beweis. Sei $\mathbf{v} \in \mathbb{R}^N$ beliebig. Es gilt

$$\begin{aligned} 0 &= \langle \mathbf{u}, \mathbf{v} \rangle_2 - \langle \mathbf{u}, \mathbf{v} \rangle_2 \\ &= \langle \mathbf{u}, \mathbf{v} - \mathbf{v} \rangle_2 && \text{(Linearität)} \\ &= \langle \mathbf{u}, \mathbf{0} \rangle_2 \\ &= \langle \mathbf{0}, \mathbf{u} \rangle_2 && \text{(Symmetrie)} \end{aligned}$$

■

Lemma A.1.2. *Sei* $\mathbf{u} \in \mathbb{R}^3$. *Dann gilt* $\mathbf{u} \times \mathbf{0} = \mathbf{0} \times \mathbf{u} = \mathbf{0}$ *und* $\mathbf{u} \times \mathbf{u} = \mathbf{0}$.

Beweis. Sei $\mathbf{v} \in \mathbb{R}^3$ beliebig. Dann gilt

$$\begin{aligned} \mathbf{0} &= \mathbf{u} \times \mathbf{v} - \mathbf{u} \times \mathbf{v} \\ &= \mathbf{u} \times (\mathbf{v} - \mathbf{v}) && \text{(Linearität)} \\ &= \mathbf{u} \times \mathbf{0} \\ &= -(\mathbf{0} \times \mathbf{u}) && \text{(Antisymmetrie)} \\ &= (-\mathbf{0}) \times \mathbf{u} && \text{(Linearität)} \\ &= \mathbf{0} \times \mathbf{u}. \end{aligned}$$

Mit der Antisymmetrie des Kreuzprodukts folgt außerdem

$$\mathbf{u} \times \mathbf{u} = -(\mathbf{u} \times \mathbf{u})$$

und die Behauptung folgt. ■

Lemma A.1.3. *Seien $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^3$. Ist mindestens ein Vektor von $\mathbf{u}, \mathbf{v}, \mathbf{w}$ der Nullvektor oder sind mindestens zwei Vektoren von $\mathbf{u}, \mathbf{v}, \mathbf{w}$ gleich, so gilt $\langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 = 0$.*

Beweis. Wir nutzen wieder den Zusammenhang

$$\langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 = \det \begin{bmatrix} | & | & | \\ \mathbf{u} & \mathbf{v} & \mathbf{w} \\ | & | & | \end{bmatrix} =: \det(\mathbf{u}, \mathbf{v}, \mathbf{w})$$

und zeigen zuerst den zweiten Fall. Sei o.E. $\mathbf{u} = \mathbf{v}$, dann gilt

$$\det(\mathbf{u}, \mathbf{v}, \mathbf{w}) = \det(\mathbf{v}, \mathbf{u}, \mathbf{w}).$$

Da die Determinante eine in den Spalten alternierende Abbildung ist, gilt aber auch

$$\begin{aligned} \det(\mathbf{u}, \mathbf{v}, \mathbf{w}) &= -\det(\mathbf{v}, \mathbf{u}, \mathbf{w}) \\ &= -\det(\mathbf{u}, \mathbf{v}, \mathbf{w}) \end{aligned}$$

und damit muss

$$\det(\mathbf{u}, \mathbf{v}, \mathbf{w}) = 0$$

sein. Wir nutzen jetzt die zweite Aussage, um die erste Aussage zu beweisen. Sei o.E. $\mathbf{u} = 0$, dann gilt

$$\begin{aligned} \det(\mathbf{u}, \mathbf{v}, \mathbf{w}) &= \det(\mathbf{0}, \mathbf{v}, \mathbf{w}) \\ &= 0 + \det(\mathbf{0}, \mathbf{v}, \mathbf{w}) \\ &= \det(\mathbf{v}, \mathbf{v}, \mathbf{w}) + \det(\mathbf{0}, \mathbf{v}, \mathbf{w}) && \text{(zweite Aussage)} \\ &= \det(\mathbf{v} + \mathbf{0}, \mathbf{v}, \mathbf{w}) && \text{(Multilinearität)} \\ &= \det(\mathbf{v}, \mathbf{v}, \mathbf{w}) \\ &= 0 && \text{(zweite Aussage)} \end{aligned}$$

und es folgt die Behauptung. ■

Das bedeutet mit der geometrischen Interpretation des Skalarprodukts insbesondere, dass das Kreuzprodukt zweier Vektoren immer senkrecht auf den Operanden steht.

Lemma A.1.4. *Seien $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^3$. Dann gilt $\langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 = \langle \mathbf{v}, \mathbf{w} \times \mathbf{u} \rangle_2$.*

Beweis. Wir berechnen

$$\begin{aligned} 0 &= \langle \mathbf{u} - \mathbf{v}, (\mathbf{u} - \mathbf{v}) \times \mathbf{w} \rangle_2 && \text{(mit Lemma A.1.3)} \\ &= \langle \mathbf{u}, \mathbf{u} \times \mathbf{w} \rangle_2 - \langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 - \langle \mathbf{v}, \mathbf{u} \times \mathbf{w} \rangle_2 + \langle \mathbf{v}, \mathbf{v} \times \mathbf{w} \rangle_2 && \text{(Linearität)} \\ &= -\langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 - \langle \mathbf{v}, \mathbf{u} \times \mathbf{w} \rangle_2 && \text{(mit Lemma A.1.3)} \\ &= \langle \mathbf{v}, \mathbf{w} \times \mathbf{u} \rangle_2 - \langle \mathbf{u}, \mathbf{v} \times \mathbf{w} \rangle_2 && \text{(Antisymmetrie)} \end{aligned}$$

und mit Umstellen folgt die Behauptung. ■

A.2 Flächenbestimmung eines Dreiecks mit dem Kreuzprodukt

Lemma A.2.1. *Sei T ein Dreieck mit Eckpunkten $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$. Dann gilt*

$$\operatorname{ar}(T) = \frac{1}{2}\|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})\|_2.$$

Beweis. Wir zeigen zunächst, dass für die Fläche des von $\mathbf{u} = \mathbf{b} - \mathbf{a}$ und $\mathbf{v} = \mathbf{c} - \mathbf{a}$ aufgespannten Parallelogramms P

$$\operatorname{ar}(P) = \|\mathbf{u} \times \mathbf{v}\|_2$$

gilt. Extrudieren wir ein Parallelogramm entlang seiner Normalenrichtung, erhalten wir ein Parallelepiped. Das Volumen dieses Parallelepipeds ist dann durch das Produkt des Grundflächeninhalts mit der Extrusionshöhe gegeben. Wir spannen also durch $\mathbf{u}$, $\mathbf{v}$ und einen dritten Vektor $\mathbf{w} = \mathbf{u} \times \mathbf{v}$ ein Parallelepiped $P_{\mathbf{w}}$ auf und berechnen dessen Volumen mit der Formel aus Abschnitt 3.2.4:

$$\begin{aligned}
\operatorname{ar}(P)\,\|\mathbf{w}\|_2 &= \operatorname{vol}(P_{\mathbf{w}}) \\
&= \langle \mathbf{w},\, \mathbf{u} \times \mathbf{v} \rangle_2 \\
&= \langle \mathbf{u} \times \mathbf{v},\, \mathbf{u} \times \mathbf{v} \rangle_2 \\
&= \|\mathbf{u} \times \mathbf{v}\|_2^2
\end{aligned}$$

Ist $\|\mathbf{w}\|_2 = 0$, so sind wir fertig. Andernfalls können wir beide Seiten durch $\|\mathbf{w}\|_2$ dividieren und erhalten

$$\begin{aligned}
\operatorname{ar}(P) &= \frac{\|\mathbf{u} \times \mathbf{v}\|_2^2}{\|\mathbf{w}\|_2} \\
&= \frac{\|\mathbf{u} \times \mathbf{v}\|_2^2}{\|\mathbf{u} \times \mathbf{v}\|_2} \\
&= \|\mathbf{u} \times \mathbf{v}\|_2
\end{aligned}$$

Das von $\mathbf{u}$ und $\mathbf{v}$ aufgespannte Dreieck T hat gerade den halben Flächeninhalt von P. Damit folgt

$$\begin{aligned}
\operatorname{ar}(T) &= \frac{1}{2}\operatorname{ar}(P) \\
&= \frac{1}{2}\|\mathbf{u} \times \mathbf{v}\|_2
\end{aligned}$$

$$= \frac{1}{2}\|(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a})\|_2.$$

und wir sind fertig. ■

A.3 Flächeninhalt eines Polygons

Mit der Beobachtung, dass Flächeninhalt unabhängig von seinem Vorzeichen invariant unter Translation und Rotation ist, kann man jedes Polygon unter Anwendung einer Rotation $\mathbf{R}$ und einer Translation $\mathbf{t}$ in die xy-Ebene legen.

Lemma A.3.1. *Die Fläche eines Polygons P mit gerichteten Kanten E_P ist gegeben durch*

$$\operatorname{ar}(P) = \frac{1}{2}\Big\| \sum_{(\mathbf{p},\mathbf{q})\in E_P} (\mathbf{p}\times\mathbf{q}) \Big\|_2.$$

Beweis. Für den Beweis benötigen wir zwei Hilfsaussagen. Für jede orthogonale Matrix $\mathbf{M}$ und jeden Vektor $\mathbf{v}$ gilt:

$$\begin{aligned}
\|\mathbf{M}\mathbf{v}\|_2 &= \sqrt{\langle \mathbf{M}\mathbf{v},\, \mathbf{M}\mathbf{v}\rangle_2} && \text{(nach Definition)}\\
&= \sqrt{\langle \mathbf{M}^T\mathbf{M}\mathbf{v},\, \mathbf{v}\rangle_2} && \text{(Eigenschaft des Skalarprodukts)}\\
&= \sqrt{\langle \mathbf{M}^{-1}\mathbf{M}\mathbf{v},\, \mathbf{v}\rangle_2} && \text{(da } \mathbf{M} \text{ orthogonal)}\\
&= \sqrt{\langle \mathbf{v},\, \mathbf{v}\rangle_2}\\
&= \|\mathbf{v}\|_2 && \text{(nach Definition)}
\end{aligned}$$

Wir können außerdem durch Nachrechnen verifizieren, dass für zwei Vektoren $\mathbf{u}$ und $\mathbf{v}$ und eine Rotation $\mathbf{R}$ stets

$$\mathbf{R}(\mathbf{u}\times\mathbf{v}) = \mathbf{R}\mathbf{u}\times\mathbf{R}\mathbf{v}$$

erfüllt ist.

Wir setzen nun die Herleitung aus Abschnitt 3.2.3 fort. Seien eine Rotation $\mathbf{R}$ und eine Translation $\mathbf{t}$ so gewählt, dass P nach Anwendung der Rotation und Translation in der xy-Ebene liegt. Dann gilt

$$\begin{aligned}
\operatorname{ar}(P) &= \operatorname{ar}(\mathbf{R}\cdot P + \mathbf{t})\\
&= \frac{1}{2}\Big[\sum_{(\mathbf{p},\mathbf{q})\in E_P} (\mathbf{R}\mathbf{p}+\mathbf{t})\times(\mathbf{R}\mathbf{q}+\mathbf{t})\Big]_z\\
&= \frac{1}{2}\Big\| \sum_{(\mathbf{p},\mathbf{q})\in E_P} (\mathbf{R}\mathbf{p}+\mathbf{t})\times(\mathbf{R}\mathbf{q}+\mathbf{t}) \Big\|_2
\end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{Rp} \times \mathbf{Rq} + \mathbf{Rp} \times \mathbf{t} + \mathbf{t} \times \mathbf{Rq} + \mathbf{t} \times \mathbf{t}) \Big\|_2 \\
&= \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{Rp} \times \mathbf{Rq}) + \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{Rp} \times \mathbf{t}) - \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{Rq} \times \mathbf{t}) \Big\|_2 \\
&= \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{Rp} \times \mathbf{Rq}) \Big\|_2 \\
&= \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} \mathbf{R}(\mathbf{p} \times \mathbf{q}) \Big\|_2 \\
&= \frac{1}{2} \Big\| \mathbf{R} \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{p} \times \mathbf{q}) \Big\|_2 \\
&= \frac{1}{2} \Big\| \sum_{(\mathbf{p},\mathbf{q}) \in E_P} (\mathbf{p} \times \mathbf{q}) \Big\|_2
\end{aligned}
$$

und die Behauptung folgt. ■

Literatur

[1]AMD Corporation. *ROCm: Platform for GPU-Enabled HPC and Ultrascale Computing*. 2016. URL: https://rocm.github.io (besucht am 8. Sep. 2018) (zitiert auf Seite 34).

[2]Autodesk, Inc. *AutoCAD*. URL: https://www.autodesk.de/products/autocad/overview (besucht am 5. Sep. 2018) (zitiert auf Seite 3).

[3]Mark de Berg, Otfried Cheong, Marc van Kreveld und Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. Kap. 3 (zitiert auf Seite 9).

[4]Herbert Edelsbrunner und Ernst Peter Mücke. „Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms“. In: *ACM Trans. Graph.* 9.1 (Jan. 1990), S. 66–104 (zitiert auf den Seiten 3, 39).

[5]F.R. Feito und J.C. Torres. „Inclusion test for general polyhedra“. In: *Computers & Graphics* 21.1 (1997), S. 23 –30 (zitiert auf Seite 38).

[6]Gerd Fischer. *Lineare Algebra: Eine Einführung für Studienanfänger*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, S. 1–31 (zitiert auf den Seiten 6, 51).

[7]W. Randolph Franklin und Salles V. G. Magalhães. „Parallel intersection detection in massive sets of cubes“. In: *Proceedings of BigSpatial'17: 6th ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. Los Angeles Area, CA, USA, 2017 (zitiert auf den Seiten 3, 31).

[8]W. Randolph Franklin, Salles V. G. Magalhães und Marcus V. A. Andrade. „Exact fast parallel intersection of large 3-D triangular meshes“. In: *27th International Meshing Roundtable*. (to appear). Alberqueque, New Mexico, 2018 (zitiert auf den Seiten 3, 39).

[9]Wm. Randolph Franklin. *Volume of a Polyhedron*. 1997. URL: https://wrf.ecse.rpi.edu/Research/Short_Notes/volume.html (besucht am 27. Aug. 2018) (zitiert auf Seite 1).

[10]Wm Randolph Franklin und Mohan S. Kankanhalli. „Volumes From Overlaying 3-D Triangulations in Parallel“. In: *Advances in Spatial Databases: Third Intl. Symp., SSD'93*. Hrsg. von D. Abel und B.C. Ooi. Bd. 692. Lecture Notes in Computer Science. Springer-Verlag, 1993, S. 477–489 (zitiert auf den Seiten 1, 3).

[11]G-Truc Creation. *OpenGL Mathematics*. 2017. URL: https://glm.g-truc.net/ (besucht am 31. Aug. 2018) (zitiert auf Seite 39).

[12]M. Held. „FIST: Fast Industrial-Strength Triangulation of Polygons“. In: *Algorithmica* 30.4 (2001), S. 563–596 (zitiert auf Seite 9).

[13] Yozo Hida, Xiaoye S Li und David H Bailey. „Library for double-double and quad-double arithmetic“. In: (2007) (zitiert auf Seite 48).

[14] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2008* (2008), S. 1–70 (zitiert auf Seite 48).

[15] Alec Jacobson, Daniele Panozzo et al. *libigl Tutorial: Boolean operations on meshes*. 2018. URL: https://libigl.github.io/libigl/tutorial/#boolean-operations-on-meshes (besucht am 5. Sep. 2018) (zitiert auf Seite 3).

[16] Khronos OpenCL Working Group. *The OpenCL Specification*. 2018. URL: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf (besucht am 8. Sep. 2018) (zitiert auf Seite 34).

[17] E. Lengyel. *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*. E-libro. Delmar Cengage Learning, 2012 (zitiert auf Seite 25).

[18] Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, Wenli Li und Maurício Gouvêa Gruppi. „Exact intersection of 3D geometric models“. In: *Geoinfo 2016, XVII Brazilian Symposium on GeoInformatics*. Campos do Jordão, SP, Brazil, 2016 (zitiert auf den Seiten 3, 39).

[19] Salles V. G. Magalhães, W. Randolph Franklin und Marcus V. A. Andrade. „Fast exact parallel 3D mesh intersection algorithm using only orientation predicates“. In: *25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2017)*. Los Angeles Area, CA, USA, 2017 (zitiert auf Seite 3).

[20] Salles Viana Gomes de Magalhães. „Exact and parallel intersection of 3D triangular meshes“. Diss. Rensselaer Polytechnic Institute, 2017 (zitiert auf den Seiten 3, 39).

[21] Donald Meagher. „Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer“. In: (Okt. 1980) (zitiert auf Seite 31).

[22] Gang Mei und John C. Tipper. „Simple and Robust Boolean Operations for Triangulated Surfaces“. In: *CoRR* abs/1308.4434 (2013). arXiv: 1308.4434 (zitiert auf Seite 3).

[23] Takatoshi Nakayama und Daisuke Takahashi. „Implementation of Multiple-Precision Floating-Point Arithmetic Library for GPU Computing“. In: (Dez. 2011) (zitiert auf Seite 48).

[24] NVIDIA Corporation. *CUDA*. 2018. URL: https://developer.nvidia.com/cuda-zone (besucht am 6. Sep. 2018) (zitiert auf Seite 34).

[25] NVIDIA Corporation. *CUDA Toolkit Documentation: CUDA C Programming Guide*. 2018. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (besucht am 8. Sep. 2018) (zitiert auf den Seiten 34, 36).

[26] NVIDIA Corporation. *CUDA Toolkit Documentation: Thrust*. URL: https://docs.nvidia.com/cuda/thrust/index.html (besucht am 3. Sep. 2018) (zitiert auf den Seiten 36, 40).

[27] NVIDIA Corporation. *NVLink Fabric Multi-GPU Processing*. URL: https://www.nvidia.com/en-us/data-center/nvlink/ (besucht am 3. Sep. 2018) (zitiert auf Seite 49).

[28] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. 2018. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (besucht am 6. Sep. 2018) (zitiert auf den Seiten 3, 33).

[29]The CGAL Project. *CGAL 4.12.1 - Polygon Mesh Processing: Corefinement and Boolean Operations*. 2018. URL: https://doc.cgal.org/latest/Polygon_mesh_processing/group__PMP__corefinement__grp.html#ga240e5df984c7d44741a7031e38203dc3 (besucht am 5. Sep. 2018) (zitiert auf Seite 3).

[30]Robert McNeel & Associates. *Rhino 6*. URL: https://www.rhino3d.com (besucht am 3. Sep. 2018) (zitiert auf den Seiten 3, 41).

[31]Chaman L Sabharwal, Jennifer L Leopold und Douglas McGeehan. „Triangle-triangle intersection determination and classification to support qualitative spatial reasoning“. In: *Polibits* 48 (2013), S. 13–22 (zitiert auf Seite 38).

[32]Bertil Schmidt, Jorge González-Domínguez, Christian Hundt und Moritz Schlarb. *Parallel Programming: Concepts and Practice*. Dez. 2017 (zitiert auf Seite 34).

[33]Mark Segal und Kurt Akeley. *The OpenGL Graphics System: A Specification*. 2014. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec44.core.pdf (besucht am 27. Aug. 2018) (zitiert auf Seite 40).

[34]The Qt Company. *Qt for Application Development*. URL: https://www.qt.io/qt-for-application-development/ (besucht am 31. Aug. 2018) (zitiert auf Seite 40).

[35]*Using the GNU Compiler Collection (GCC): Floating Types*. 2018. URL: https://gcc.gnu.org/onlinedocs/gcc/Floating-Types.html (besucht am 4. Sep. 2018) (zitiert auf Seite 48).

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Mainz, 11. 09. 2018

Justus Henneberg