Hardwarebasiertes Volume Raycasting mit deformierbaren Tetraedernetzen

Diplomarbeit

im Fach Angewandte Informatik

Institut für Computergrafik und Multimediasysteme

Fachbereich 12 - Elektrotechnik und Informatik

Universität Siegen

Lisa Brückbauer

Matrikelnummer: 589864

Erstgutachter:	Dr. Christof Rezk-Salama
Zweitgutachter:	Prof. Dr. Andreas Kolb
Tag der Ausgabe:	1.7.2007
Tag der Abgabe:	31.12.2007

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Siegen, den 7. Dezember 2007

Abstract

The approach presented in this thesis allows volume data sets to be deformed in real-time. A tetrahedral grid is used as a proxy geometry for standard voxel datasets on uniform grids. Deformation is achieved by transforming the vertices of the tetrahedral grid while retaining the texture coordinates into the 3D-scalar field given at these vertices. Geometry and topology of the tetrahedral grid are stored completely in the local video memory of the graphics card. By this, high quality images can be rendered using GPU-based ray-casting. The implementation considerably takes advantage of the flexibility provided by the Unified Shader Model.

Danksagung

Ich möchte mich herzlich bedanken...

...bei meinem Betreuer Dr. Christof Rezk-Salama

Ich danke Dir für Deine Betreuung, für richtungsweisende Gedanken, für Motivation im richtigen Moment, für unendliche Geduld, gelegentliche Aufbauarbeit und nicht zuletzt für jede Menge Spaß bei der Arbeit!

...bei meinem Freund Michael

Ich danke Dir für Deine Liebe und für Deine Geduld und Nachsicht, die gelegentlich von mir auf die Probe gestellt wurden.

...bei meinen Eltern

Auf Eure Liebe und Unterstützung konnte ich immer bauen, auch manchen Umweg habt Ihr hingenommen und nie an mir gezweifelt. Dafür danke ich Euch von Herzen.

Lisa Brückbauer, Dezember 2007

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

Tabellenverzeichnis **vii**

Verzeichnis der Listings **viii**

1 Einleitung **1**

2 Grundlagen und bisherige Lösungen **3**

2.1 Raycasting im Volume Rendering 3

2.2 Tetraedergitter 4

2.3 Raycasting auf Tetraedergittern 6

2.4 Deformation 8

2.4.1 Oberflächendeformation 9

2.4.2 Volumendeformation 9

3 Die Theorie: GPU-basiertes Raycasting auf Tetraedergittern **11**

3.1 Überblick 11

3.1.1 Ausgangspunkt und Ziel 11

3.1.2 Der Weg zum Ziel 11

3.2 Datenstrukturen 13

3.3 Raycasting 15

3.3.1 Strahlverfolgung 16

3.3.2 Sampling und Compositing 21

3.3.3 Problematik nicht-konvexer Gitter 24

3.4 Deformation 25

3.5 Zusammenfassung 26

4 Implementierung **28**

4.1 Überblick 28

4.2 Datenstrukturen 28

4.2.1 Vom Mesh zum XML 28

4.2.2 Auslesen der Grid-Beschreibung 28

4.2.3 Speicherung der Geometrie und Topologie in Texturen 35

4.3 Raycasting 40

4.3.1 Eintrittspunkte 41

4.3.2 Strahlverfolgung 42

4.3.3 Sampling und Compositing 44

4.3.4 Bestimmung des Folgetetraeders 45

4.3.5 Early Ray Termination 47

4.3.6 Depth Peeling 47

4.4 Deformation 52

4.5 Zusammenfassung 55

5 Ergebnisse **57**

6 Zusammenfassung und Ausblick **61**

7 Anhang **62**

7.1 Vertexshaderprogramm: tetra_vert.cg 62

7.2 Fragmentshaderprogramm: depthpeeling_frag.cg 63

7.3 Vertexshaderprogramm: screen2D_vert.cg 64

7.4 Fragmentshaderprogramm: tetra2D_frag.cg 65

Literaturverzeichnis **68**

Abbildungsverzeichnis

1 Schematische Darstellung des Raycasting-Verfahrens 4
2 Durch Volume Raycasting erzeugte Bilder 5
3 Ein Sehstrahl schneidet ein Tetraedergitter 8
4 Volumendatensatz und Repräsentation durch ein Tetraedergitter 12
5 Überblick über die Texturen 15
6 Tetraedermodell 17
7 Berechnung von Eintritts- und Austrittspunkt 18
8 Baryzentrische Gewichte im Tetraeder 20
9 Emission und Absorption entlang des Sehstrahls 23
10 Schema des Raycasting bei nicht-konvexen Geometrien 25
11 Vertexkoordinaten und Texturkoordinaten 26
12 Zusammenhang zwischen der Geometrie und der XML-Beschreibung 29
13 Alle Listen auf einen Blick 34
14 Lookup der Vertexkoordinaten für einen Tetraeder 38
15 Lookup der Nachbartetraeder von $T_{current}$ 40
16 Vertexbuffer für eine statische Gridstruktur 41
17 Ablauf des Raycasting 42
18 Erster Render Pass. 48
19 Zweiter Render Pass. 49
20 Ablauf des Renderings mit integriertem *Depth Peeling* 50
21 Vertexbuffer für eine Gridstruktur mit Animation 53
22 Die vier Testgitter 57
23 Der Schädeldatensatz, Volumen und Tetraedergitter 59
24 Animation des Schädeldatensatzes 60

Tabellenverzeichnis

1 Namenskonventionen für die Texturen 35

2 Performance 58

3 Eigenschaften der verwendeten Gitter 58

Listings

1 Konstruktion der möglichen Dreiecksflächen 31
2 Festlegung der Tetraederzugehörigkeit für jede Fläche 33
3 Skalierung der Vertexkoordinaten 37
4 Ermittlung der Texturkoordinaten der Nachbarn an Face0 39
5 Die Geradengleichung des Sehstrahls 43
6 Berechnung der baryzentrischen Koordinaten im Tetraeder 44
7 Berechnung der 3D-Texturkoordinaten 44
8 Sampling und Anwendung einer Transferfunktion 44
9 Front-to-Back-Compositing 45
10 Voranschreiten entlang des Sehstrahls 45
11 Erfragen der Texturkoordinaten der Nachbartetraeder 46
12 Wahl des richtigen Folgetetraeders 46
13 Prüfung auf Austritt aus dem Volumen 47
14 Early Ray Termination 47
15 Auslesen des Tiefenwerts aus dem vorangegangenen Render Pass 51
16 Standardwerte der Fragmente vor dem Tiefentest 51
17 Vergleich zwischen dem aktuellen und dem vorherigen z-Wert des Pixels 52
18 Eine Vertexgruppe, zwei Vertexkonfigurationen 53
19 Vollständiger Vertexshadercode für die Animation 54
20 Interpolation zwischen den Vertices zweier Zeitschritte 55
21 Code des Vertexprogramms für die Geometrieinterpolation 62
22 Code des Fragmentprogramms für das Depth Peeling 63
23 Code des Vertexprogramms für die Transformation in den Screenspace 64
24 Code des Fragmentprogramms für das Raycasting 67

1 Einleitung

Die moderne Computergrafik macht es möglich, aus den Schichtaufnahmen von Computertomographen oder MRT-Scannern dreidimensionale Bilder zu konstruieren. Techniken der Volumenvisualisierung liefern hochqualitative Bilder, die beispielsweise Chirurgen bei der Planung schwieriger Operationen helfen können. Durch die sinnvolle Einstellung von Transferfunktionen kann die Darstellung so angepasst werden, dass nur die relevanten Bereiche (z.B. die Blutgefäße) sichtbar sind, während unwichtige Bereiche (z.B. Knochenstrukturen) des Datensatzes ausgeblendet werden. Nicht möglich war bisher die echtzeitfähige Deformation von Volumendaten (unter gleichzeitiger Beibehaltung einer hohen Qualität der Bilder). Die Deformation starrer Volumendaten kann sich allerdings bei chirurgischen Eingriffen bezahlt machen. Beispiele sind hier die Operationsplanung zum Einsatz künstlicher Hüftgelenke, Rekonstruktionsmaßnahmen im Bereich der Mund-, Gesichts- und Kieferchirurgie oder auch das sogenannte Brain Shift-Phänomen [5], welches bei Operationen am offenen Gehirn auftritt. Dabei verändert das Gehirn u.a. durch den Austritt von Flüssigkeit seine Lage und Form. Diese Deformation macht die Nutzung präoperativer Planungsdaten schwierig.

In dieser Arbeit wird gezeigt, dass die Deformation bzw. Animation von Volumendaten in Echtzeit auf einfache Weise möglich ist. Die Schwierigkeiten bei der Deformation von Volumendaten werden hauptsächlich durch die Unflexibilität der uniform-rectilinearen Gitterstrukturen verursacht, durch die die Datensätze diskretisiert werden. Wenn man aber nicht direkt auf den Volumendaten arbeitet, sondern auf einem Tetraedergitter, welches als Hilfsgeometrie im gegebenen, uniformen Datensatz liegt, wird die Deformation sehr einfach. Anders als bei bisherigen Ansätzen zur Volumendeformation bleibt außerdem die gewohnt hohe Qualität erhalten, die bei der direkten Volumenvisualisierung durch Raycasting erreicht wird.

Die Arbeit beginnt mit einem kurzen Überblick über die Grundlagen des Raycasting, über die Eigenschaften von Tetraedergittern und über die Möglichkeiten der Deformation von Oberflächen- und Volumendaten (Kapitel 2). Kapitel 3 widmet sich den theoretischen und mathematischen Grundlagen, die hinter der Idee dieses Ansatzes stehen. Auf dieser Basis wird in Kapitel 4 die Implementierung des Volume Raycasting und der Volumendeformation behandelt. Die Ergebnisse

werden in Kapitel 5 durch eine Performancemessung belegt. In Kapitel 6 werden die wichtigsten Erkenntnisse zusammengefasst.

2 Grundlagen und bisherige Lösungen

Ziel dieses Kapitels ist es, dem Leser einen kurzen Überblick über die Grundlagen zu geben, die für das Verständnis dieser Arbeit vorausgesetzt werden. Die folgenden Abschnitte sind daher relativ kurz gehalten. Zunächst geht es um Raycasting im Volume Rendering (2.1) und um die Eigenschaften von Tetraedergittern (2.2), bevor in Abschnitt 2.3 auf die Kombination von beidem, nämlich tetraederbasiertes Raycasting, und auf bisherige Ansätze diesbezüglich eingegangen wird. Das Kapitel endet mit einem Überblick über Möglichkeiten der Deformation, insbesondere der Volumendeformation (2.4.2).

2.1 Raycasting im Volume Rendering

Die Darstellung dreidimensionaler Skalarfelder, die durch bildgebende Verfahren wie die Computer- oder Kernspintomographie gewonnen werden oder mittels numerischer Simulationen erzeugt werden [15], wird als Volume Rendering bezeichnet. Im Volume Rendering unterscheidet man zwischen Bildraum- und Objektraumverfahren (*image-order* und *object-order approaches*). Die Bildraumverfahren, zu denen auch der hier verwendete Raycasting-Algorithmus gehört, haben den Startpunkt ihrer Berechnung immer in der 2D-Bildebene, während die Objektraumverfahren zunächst das 3D-Skalarfeld traversieren, bevor die Volumenelemente auf die Bildebene projiziert werden [2].
Beim Volume Raycasting wird für jedes Pixel der Bildebene ein Sehstrahl in das Volumen geschossen. Entlang eines jeden Sehstrahls wird durch Abtastung (*Sampling*) und Compositing der Beitrag des Volumens für diesen einen Pixel berechnet (siehe Abbildung 1). Einer der ersten Ansätze des Volume Raycasting geht zurück auf M. Levoy [11]. Gegeben ist in diesem Ansatz zunächst ein 3D-Skalarfeld, welches durch ein dreidimensionales rectilineares Gitter beschrieben wird. Das gewählte Modell des Lichttransports liefert die *rgb*-Farbwerte und die Opazität an jedem Voxel des Datensatzes. Für jeden Pixel der Bildebene wird nun ein Strahl in den Voxeldatensatz geschossen. Entlang dieses Strahls wird das Skalarfeld in gleichmäßigen Abständen abgetastet und die Farb- und Opazitätswerte an diesen Punkten mittels der acht umgeben-

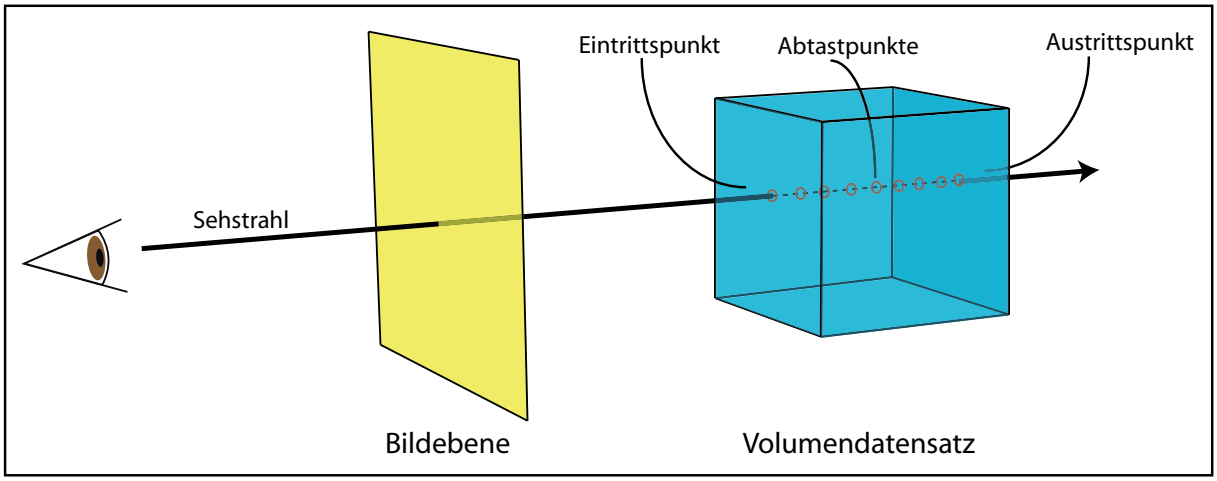

Abbildung 1: Schematische Darstellung des Raycasting-Verfahrens

den Gitterpunkte trilinear interpoliert. Der Hintergrund der Szene wird vollständig opak gesetzt. Für jeden Strahl wird nun durch back-to-front-Compositing aus den abgetasteten Farb- und Opazitätswerten eine resultierende Farbe berechnet, die in diesem Pixel angezeigt wird. Levoys Ansatz basiert auf uniformen Gittern. Es ist aber auch möglich, das Raycasting auf unstrukturierten Gittern ablaufen zu lassen. Eine Übersicht über verschiedene Verfahren des GPU-basierten Volume Renderings auf unstrukturierten Gittern bietet Silva [17]. Bevor erste Ansätze des tetraederbasierten Raycastings erläutert werden, sollen zunächst einige Grundlagen von Tetraedergittern untersucht werden.

2.2 Tetraedergitter

Bildgebende Verfahren wie die Magnetresonanztomographie oder die Computertomographie liefern Datensätze, die auf uniformen Gittern basieren. In medizinischen Volume Rendering-Anwendungen sind die uniformen Gitter daher am weitesten verbreitet. Dies liegt an der leichteren Handhabbarkeit, an der effizienteren Art der Speicherung und dem schnelleren Zugriff auf die Daten [2]. Die Effizienz besteht vor allem darin, dass für uniforme Gitter lediglich die Anzahl der Dimensionen, die Anzahl der Gitterpunkte in jeder Dimension sowie der Abstand zwischen

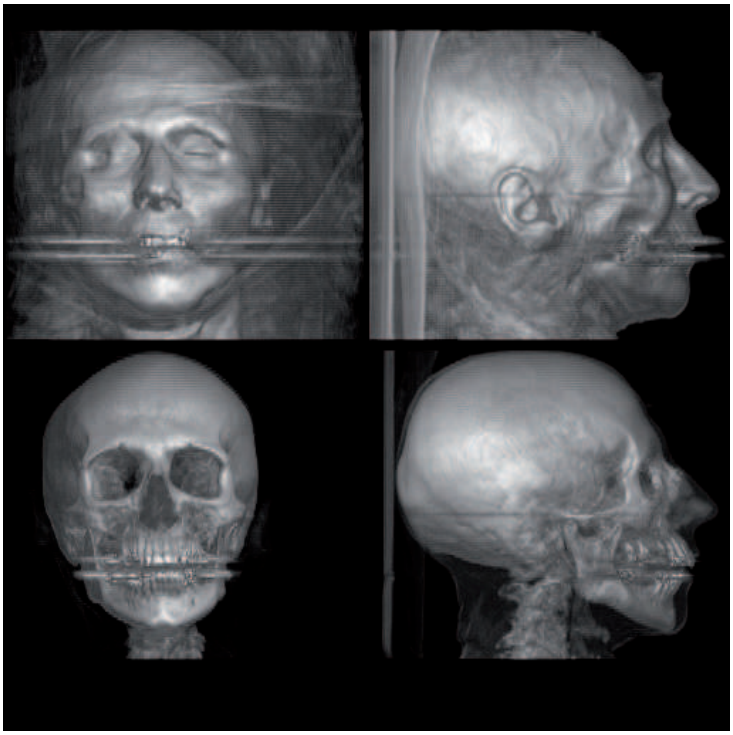

Abbildung 2: Durch Volume Raycasting erzeugte Bilder. Aus [11]

den Gitterpunkten bekannt sein muss, um auf beliebige Voxel zuzugreifen. Die Speicherung der Gitterstruktur erfolgt also implizit. Uniforme Gitter haben allerdings auch Nachteile. Einer davon ist die unflexible, starre Gitterstruktur ohne Möglichkeiten der lokalen Verfeinerung. Die Abtastrate eines Volumendatensatzes ist in einem uniformen Gitter überall konstant. Verfügt der Datensatz über Bereiche, an denen höhere Frequenzen im kontinuierlichen Skalarfeld vorliegen, so muss auch die Abtastrate erhöht werden, d.h. das Gitter muss global verfeinert werden. Unter Beibehaltung der impliziten Speicherung kann die Verfeinerung aber nicht lokal begrenzt werden. So kann es vorkommen, dass es große Bereiche im Datensatz gibt, die unangemessen hoch abgetastet werden, wodurch die Speichereffizienz wieder verloren geht. Ein weiterer Nachteil uniformer Gitter betrifft die Deformation: Verschiebt sich ein Vertex eines uniformen Gitters, geht die Uniformität verloren, und damit auch die Effizienz der Speicherung und die Einfachheit von Schnittpunktberechnungen zwischen Strahl und Ebene. Abhilfe schaffen hier unstrukturierte Tetraedergitter. Ihr Nachteil liegt vor allem in der Notwendigkeit, Geometrie und Topologie

explizit zu speichern. Demgegenüber ergeben sich jedoch einige Vorteile: Tetraedergitter können sehr genau an den Volumendatensatz angepasst werden, d.h. in Bereichen mit hohen Frequenzen kann ein unstrukturiertes Gitter sehr fein aufgelöst sein. In homogenen Bereichen kann die Anzahl der Zellen dagegen lokal reduziert werden [2]. Verglichen mit uniformen Gittern ermöglichen Tetraedergitter darüber hinaus bessere Deformationsmöglichkeiten. Auf die Deformation wird ausführlicher in Kap. 3.4 eingegangen.

Um auf Tetraedern basierende Gitter darzustellen kommen zwei Verfahren in Betracht: Raycasting (als Bildraumverfahren) und Cell Projection (als Objektraumverfahren). Traditionell wird beim tetraederbasierten Volume Rendering der Ansatz der Cell Projection angewendet. Cell Projection benötigt allerdings für das korrekte Compositing eine blickpunktabhängige Tiefensortierung der Tetraederzellen. Für dynamisch deformierbare Tetraedergitter muss diese Tiefensortierung ständig neu berechnet werden. Kommt es im Zuge der Deformation eines Tetraedernetzes zu sogenannten Visibility Cycles [7, 8], bei denen sich mehrere Tetraeder wechselweise teilweise überdecken, ist eine korrekte Tiefensortierung unmöglich. Raycasting dagegen hat den Vorteil, dass es keine Tiefensortierung der Tetraeder braucht. Um den Sehstrahl korrekt verfolgen zu können, müssen lediglich die Nachbarschaftsbeziehungen zwischen den Tetraedern bekannt sein. Sogar im Falle von Visibility Cycles, wo die Tiefensortierung fehlschlägt, funktionert Raycasting ohne Probleme. Ein weiteres Argument für die Verwendung des Raycasting besteht in der hohen Qualität der Bilder, die damit erzeugt werden können. Siehe dazu auch [1] und [16].

2.3 Raycasting auf Tetraedergittern

Im folgenden sollen einige vorhandene Ansätze für tetraederbasiertes Volume Raycasting vorgestellt werden. Hierbei sollte beachtet werden, dass in dieser Arbeit das Tetraedernetz als Proxygeometrie zum Zwecke der Deformation eines uniformen Datensatzes dient (siehe Kapitel 3). Das bedeutet, dass an den Vertices des Tetraedernetzes nicht direkt Skalarwerte gegeben sind, sondern Texturkoordinaten, die als Index in den uniformen Datensatz dienen. Für den Raycasting-Algorithmus ergeben sich aber daraus keine nennenswerten Unterschiede.
Das Prinzip des Volume Raycasting, wie von Levoy beschrieben (siehe 2.1), bleibt auch bei Te-

traedergittern erhalten. Für jeden Pixel wird ein Strahl durch das Volumen geschossen. Entlang des Sehstrahls werden Samplingwerte berechnet, die durch ein Compositingverfahren und die Anwendung einer Transferfunktion später die Farbe des Pixels ergeben. Allerdings ergeben sich bei Tetraedergittern einige Unterschiede:

- Beim Samplingschritt muss baryzentrisch im Tetraeder interpoliert werden, anstatt trilinear im Würfel
- Die Berechnung des Folgetetraeders ist aufwendiger als die Berechnung des nächsten Hexaeders. In der Regel ist bei Tetraedernetzen eine Zellsuche notwendig.

Bisherige Ansätze für Tetraeder-Raycasting finden sich u.a. in [18, 19] und [17]. Weiler et al. [18] stellen einen tetraederbasierten Volume Raycasting-Algorithmus vor, der - so wie auch der Ansatz der vorliegenden Arbeit - die Geometrie und Topologie des Tetraedernetzes im lokalen Speicher der Grafikkarte hält sowie alle Berechnungen auf der GPU ausführt. Unterschiede zwischen [18] und dem vorliegenden Ansatz bestehen allerdings im Umgang mit der Strahlintegration und mit nicht konvexen Gittern. Weiler et al. benutzen *pre-integration tables* für ersteres, während in dieser Arbeit an jedem Samplingpunkt ein Lookup in einer 3D-Textur vollzogen wird. Nichtkonvexe Gitter werden in [18] durch Konvexifizierung behandelt, während hier ein Depth Peeling Schritt eingebaut wird. Weiler et al. räumen ein, dass aufgrund der Beschränkungen des Texturspeichers und der Probleme, die bei nicht-konvexen Meshes entstehen, die Vorteile ihres Ansatzes geschmälert werden. Um auch große Tetraedermeshes effizient darstellen zu können, wird in [19] eine Speicherung anhand von Tetrahedral Strips vorgeschlagen. Im Unterschied zu [18] werden nichtkonvexe Gitter wie auch hier mit einem Depth Peeling-Schritt behandelt.

Die Schwierigkeiten der oben genannten bestehenden Ansätzen waren bisher oft auf hardwareseitige Beschränkungen zurückzuführen. Aktuelle Grafikkarten, vor allem die NVidia GeForce 8 Serie, eröffnen heute aufgrund des schnellen lokalen Speichers und der flexiblen Parallelverarbeitung (Unified Shader Model) weitaus bessere Möglichkeiten. Ein neuerer Ansatz von Muigg et al. [12] stellt ein hybrides Verfahren für das Raycasting großer Volumendatensätze vor, bei dem

unstrukturierte Gitter (in wichtigen Bereichen) mit strukturierten Gittern (in weniger wichtigen Bereichen) kombiniert werden.

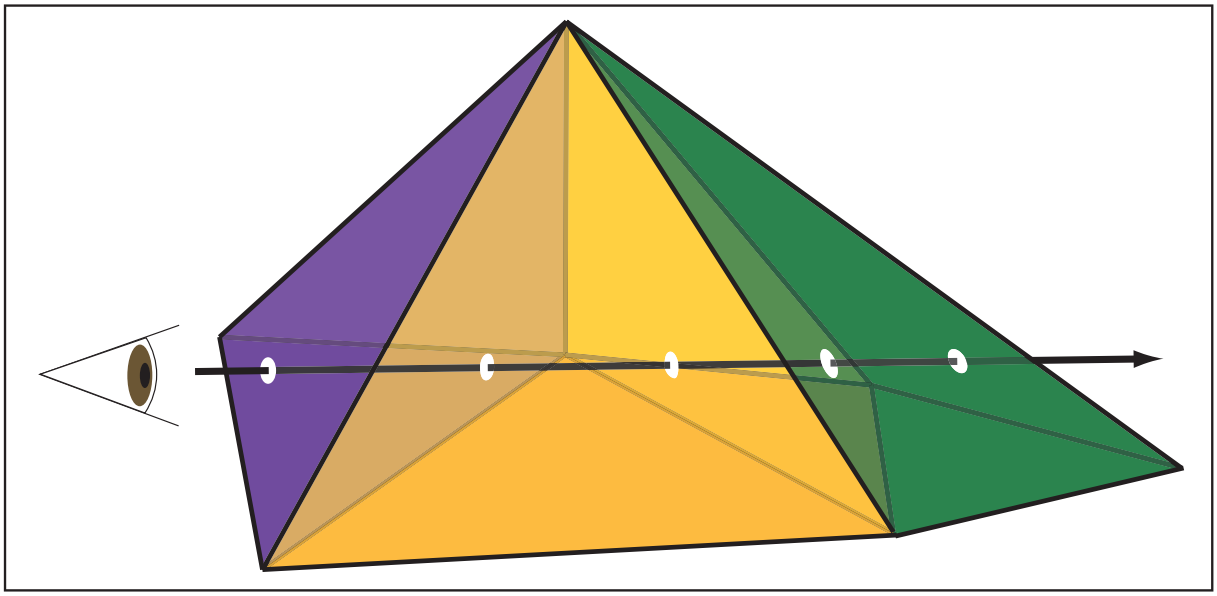

Abbildung 3: Ein Sehstrahl schneidet ein Tetraedergitter

2.4 Deformation

Ein wichtiger Aspekt dieser Arbeit ist die Deformation der Volumendaten. Wichtig ist die Volumendeformation zum Beispiel für die medizinische Operationsplanung. Vorstellbar wären hier chirurgische Eingriffe im Bereich der Gelenke. Auch im Hinblick auf die Simulation des Verhaltens von Weichgewebe ist es wichtig, Volumendaten möglichst realistisch deformieren zu können. Eine weitere interessante Anwendung ist die Möglichkeit, die eigentlich starren Volumenmodelle zu animieren, mit dem Ziel, biomechanische Abläufe oder, außerhalb des medizinischen Bereichs, technisch-mechanische Funktionen zu veranschaulichen. Darüberhinaus kann man sich gut vorstellen, dass die ein oder andere Anwendung auch im künstlerischen oder filmischen Bereich zu finden ist. Für die Deformation von Objekten gibt es zwei Herangehensweisen: Die Oberflächendeformation und die Volumendeformation.

2.4.1 Oberflächendeformation

Klassischerweise unterscheidet man in der Computergrafik zwischen Geometrie *(shape)* und optischen Eigenschaften *(appearance)* von Objekten oder Modellen [2]. Die Form eines Objekts entspricht der Geometrie, gegeben in Form von Punkten bzw. Dreiecken, die Oberfläche wird meist beschrieben durch eine Textur. Wird die Geometrie deformiert, d.h. wird die Lage des Eckpunktes eines Dreiecks verändert, so werden zwar die Raumkoordinaten des Punktes, nicht aber die Texturkoordinaten in diesem Punkt geändert. Die Textur passt sich der neuen Geometrie an [2], und als Ergebnis deformiert sich das Objekt.

2.4.2 Volumendeformation

Im Volume Rendering gibt es die klassische Trennung zwischen Geometrie und optischen Eigenschaften nicht. Ein Volumendatensatz beschreibt in Verbindung mit einer Transferfunktion gleichermaßen die Form und die Oberfläche des Modells. Die zugrundeliegende Gitterstruktur, meistens ein uniformes rectilineares Gitter, ist lediglich eine Hilfsgeometrie, die nicht die Form des Objekts, sondern die Diskretisierung des Datenraums beschreibt. Zur Deformation von Volumendatensätzen stehen zwei grundlegende Möglichkeiten zur Verfügung: Die Deformation der Hilfsgeometrie (d.h. im *model space*) einerseits und die Änderung der 3D-Texturparameter (im *texture space*) andererseits. Da im Hinblick auf die vorliegende Arbeit im wesentlichen das Verständnis der ersten Möglichkeit von Vorteil ist, wird im Folgenden lediglich auf die Deformation im *model space* eingegangen. Das grundlegende Verfahren wurde schon unter 2.4.1 beschrieben: Die Deformation erfolgt aufgrund der Verschiebung eines Vertices unter gleichzeitiger Beibehaltung der Texturparameter des Vertices. Nun verhält es sich jedoch so, dass bei der Verschiebung eines Vertices innerhalb eines uniform rectilinearen Gitters die Eigenschaft der Planarität verloren geht. Die vier Punkte des Rechtecks liegen u.U. nicht mehr innerhalb einer Ebene. Somit entstehen gekrümmte Flächen, und es wird schwierig, Schnittpunktberechnungen zwischen Strahl und Ebene, wie sie z.B. beim Volume Raycasting nötig sind, durchzuführen. Komplizierter wird auch die Interpolation, weil man für einen Punkt nicht mehr eindeutig bestimmen kann, in welchem Würfel er vor der Deformation lag. Bisherige Ansätze für die De-

formation von Volumendaten unterteilen das Volumen meist in Slices ([9], [10], [20] und [4]) und erreichen aufgrund der Notwendigkeit, die Slices in Echtzeit zu tesselieren, nur geringe Frameraten [2].

Abhilfe würde hier die Unterteilung der Würfel der Proxygeometrie in jeweils fünf Tetraeder schaffen [2]. Während die Verschiebung eines Vertex im Hexaedergitter einer nicht-linearen Transformation des Raumes entspricht, kann bei Tetraedern die Deformation durch eine einfache affine Transformation beschrieben werden. Allerdings müssten hier die Tetraeder innerhalb eines jeden Würfels tiefensortiert werden.
In dieser Arbeit wird eine Hybridlösung vorgestellt: Die durch Tomographiescanner erzeugten Bilder basieren auf uniformen Gittern. In diesen uniformen Datensatz wird ein Tetraedergitter als Hilfsgeometrie gelegt. Anhand des Tetraedergitters wird dann der Volumendatensatz deformiert. Optimal wäre dazu ein Tetraedergitter, welches die unterschiedlichen Strukturen des Datensatzes optimal beschreibt. Dabei könnte man durch Segmentierung unterschiedliche Gewebearten (in medizinischen Datensätzen) oder Materialeigenschaften (in technischen Datensätzen) klassifizieren, dann die unterschiedlichen Volumen, die sie beschreiben, tetraedrisieren und darauf aufbauend die Deformation/Animation vornehmen. Diese Arbeit behandelt jedoch nicht die Erzeugung solcher passenden Gitter, sondern liefert einen Proof of Concept der prinzipiellen Vorgehensweise für die Deformation.

3 Die Theorie: GPU-basiertes Raycasting auf Tetraedergittern

3.1 Überblick

In Kapitel 2.4.2 wurde deutlich, dass eine direkte Deformation von Volumendaten auf uniformen Gittern schwierig ist. Um also Bilddaten, so wie sie in der Praxis am häufigsten durch bildgebende Verfahren erzeugt werden, deformieren zu können, sollen die uniformen Gitter mit der Flexibilität tetraederbasierter Gitter kombiniert werden. Dieses Kapitel widmet sich den theoretischen Hintergründen, die hinter diesem Ansatz stehen.

3.1.1 Ausgangspunkt und Ziel

Im Folgenden wird davon ausgegangen, dass sich "innerhalb" eines durch ein rectilineares Gitter diskretisierten Datensatzes ein Tetraedergitter befindet, welches die Strukturen im Datensatz möglichst optimal repräsentiert (Abbildung 4). Das bedeutet, dass im besten Fall Segmentierungsdaten des Volumens existieren und die einzelnen Segmentierungsmasken jeweils durch ein Tetraedergitter approximiert werden[1]. Diese Tetraeder-Proxygeometrie dient als Grundlage für die effiziente Berechnung des Raycasting und der Deformation. Ziel ist es schließlich, einen Volumendatensatz durch Raycasting in hoher Qualität darzustellen und in Echtzeit deformieren zu können.

3.1.2 Der Weg zum Ziel

Der Weg vom Ausgangspunkt zum Ziel kann in folgende, implementierungsunabhängige Schritte unterteilt werden:

- Datenstrukturen erstellen

  In diesem Schritt (Kapitel 3.2) müssen die Mesh-Daten eingelesen werden. Im Ansatz die-

[1] Dieser Optimalfall angepasster Tetraedergitter ist allerdings keine Voraussetzung für das Funktionieren der Volumendeformation.

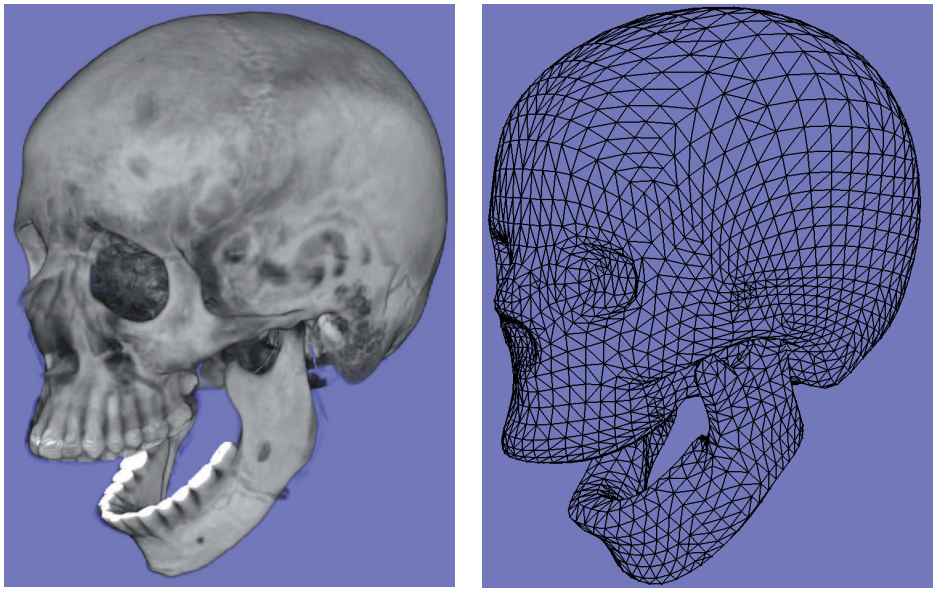

Abbildung 4: Links: Volumendatensatz, rechts: Repräsentation durch ein Tetraedergitter (Hidden Lines-Darstellung)

ser Arbeit wird das Tetraedergitter durch eine Liste von jeweils vier zusammenhängenden Vertices beschrieben. In welchem Format die Beschreibung des Tetraedernetzes vorliegt ist allerdings zweitrangig. Wichtig ist, dass auf der Basis der Geometriebeschreibung eine Rekonstruktion der Topologie des Tetraedernetzes erstellt werden kann. Da sämtliche Berechnungen auf der GPU erfolgen sollen, müssen alle in diesem Schritt gewonnenen Daten anschließend in den lokalen Speicher der Grafikkarte geladen werden.

- Raycasting

  Der Raycasting-Schritt (Kapitel 3.3) umfasst die Bestimmung der Eintrittspunkte der Sehstrahlen, die Strahlverfolgung (d.h. die Bestimmung der Tetraeder, die ein Strahl durchläuft) und die Bestimmung des Farbwerts für jedes Pixel durch Sampling und Compositing. Auch die Behandlung der Problematik nichtkonvexer Gitter wird diesem Schritt zugeordnet.

- Deformation

  Im Deformationsschritt (Kapitel 3.4) wird der Datensatz anhand mehrerer Vertexgruppen deformiert. Diese Vertexgruppen beschreiben die Vertices des Tetraedernetzes zu einem gegebenen Zeitpunkt. Die Deformation erfolgt dadurch, dass sich das Tetraedernetz deformiert, während die Texturkoordinaten für die 3D-Textur gleich bleiben.

Diese drei Schritte sollen nun genauer erläutert werden.

3.2 Datenstrukturen

Über das gegebene Tetraedergitter sind zu Beginn nur zwei Dinge bekannt: Die Koordinaten aller Vertices und die Zusammensetzung von jeweils vier Vertices zu einem Tetraeder[2]. Diese Informationen liegen in einer XML-Datei vor. Die Vertexkoordinaten (x, y, z) beschreiben die Geometrie des Tetraedergitters. Zur Realisierung einer *Shared Vertex*-Datenstruktur müssen die Vertexkoordinaten so gespeichert werden, dass sie über einen Index abrufbar sind. So reicht es, jeden Vertex nur einmal zu speichern und für einen gegebenen Tetraeder nur die Indizes seiner vier Vertices zu vermerken. Aus den beiden gegebenen Informationen, den Koordinaten und den Vierertupeln, können weitere Eigenschaften des Tetraedergitters abgeleitet werden und zum Beispiel die Frage beantwortet werden, welche drei Vertices ein Dreieck bilden. Dadurch kann wiederum festgestellt werden, zu welchen Tetraedern dieses Dreieck gehört. Anhand dieser Information können die Nachbarschaftsbeziehungen ermittelt werden, denn zwei benachbarte Tetraeder "teilen" sich das selbe Dreieck. Kurz gesagt: Anhand der gegebenen Informationen kann die komplette Geometrie und Topologie abgeleitet werden. Da die Anwendung zur Deformation echtzeitfähig sein soll, sollen alle notwendigen Daten im Grafikspeicher gehalten werden. Es werden daher folgende Informationen bzw. Datenstrukturen erzeugt und in Form von Texturen in den lokalen Speicher der Grafikhardware geladen:

[2]Dies gilt für diese Arbeit. Die Beschreibung des Gitters kann auch anders, z.B. durch eine Liste aller Faces vorliegen.

1. Informationen über die Vertices

   Für jeden Vertex sollen die *xyz*-Koordinaten gespeichert werden, und zwar so, dass man über einen Index i für den Vertex v_i die Koordinaten x, y und z erfragen kann (*Shared Vertex*-Datenstruktur).

2. Informationen über die Tetraeder

   Für jeden Tetraeder sollen die vier *Indizes* seiner Vertices gespeichert werden. Soll sich die Geometrie ändern, reicht es, die Vertexkoordinaten zu aktualisieren. Die Indizes behalten ihre Gültigkeit und die Nachbarschaftsbeziehungen bleiben erhalten.

3. Informationen über die Nachbarschaften der Tetraeder

   Für jeden Tetraeder muss gespeichert werden, welche Tetraeder benachbart sind. Falls ein Tetraeder keinen Nachbarn hat, weil er selbst ein Randtetraeder ist, muss dies ebenfalls gespeichert werden.

Für die Darstellung des Volumens mittels Raycasting ist die Kenntnis über die Nachbarschaftsbeziehungen immens wichtig. Für die Strahlverfolgung durch das Tetraedernetz muss lediglich der Eintrittspunkt in die Geometrie berechnet werden. Ist dieser bekannt, so kann auf einfache Weise ermittelt werden, welche Tetraeder der Sehstrahl nacheinander schneidet.

Die Beziehungen zwischen den Texturen, in denen die Informationen über die Vertices, die Tetraeder und die Nachbarschaften gespeichert werden, sind in Abbildung 5 dargestellt.

Die Kenntnis der Geometrie und Topologie des Tetraedernetzes ist natürlich nicht Selbstzweck. Der Sinn des Tetraedernetzes ist schließlich, als Proxygeometrie für das mittels Raycasting darzustellende und zu deformierende Volumen zu dienen. Insofern werden zwei weitere Texturen angelegt:

4. Der Volumendatensatz

   Der darzustellende und zu deformierende Datensatz wird in eine 3D-Textur geladen.

5. Die Texturkoordinaten der Vertices

   Für jeden Vertex des Tetraedergitters müssen die korrespondierenden uvw-Koordinaten

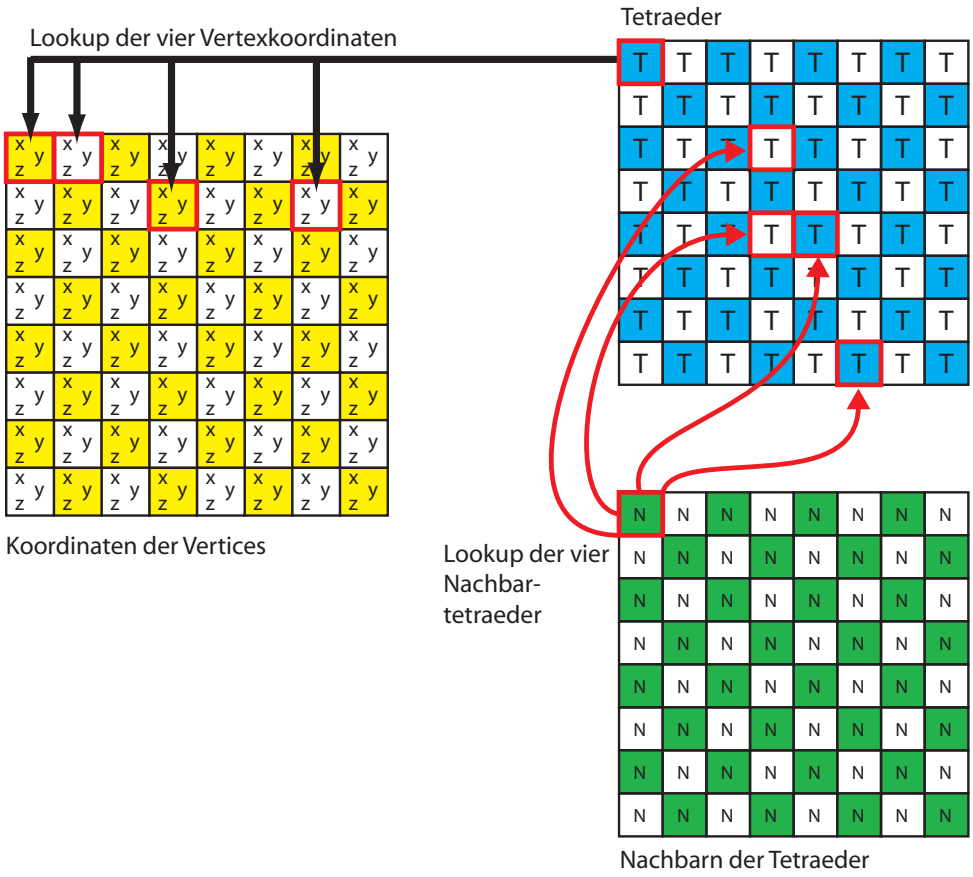

Abbildung 5: Überblick über die Texturen

bezüglich der Volumentextur (4) berechnet und gespeichert werden. Der Zusammenhang zwischen der Volumentextur und der Texturkoordinatentextur wird im Kapitel über die Deformation (3.4) noch genauer erläutert.

3.3 Raycasting

Um den Raycasting-Algorithmus für die gegebenen Daten zu implementieren, kann man sich zunächst das Prinzip verdeutlichen, nach dem *ein* Sehstrahl *einen* Tetraeder schneidet. Bekannt sind in diesem Schritt die Geradengleichung des Sehstrahls und die Koordinaten der Vertices des Tetraeders. Die Berechnung wird formal in Abschnitt 4.3.2 beschrieben. Sind Eintritts- und Austrittspunkt bekannt, können durch baryzentrische Interpolation die 3D-Texturkoordinaten

an den Schnittpunkten von Sehstrahl und Flächen bestimmt werden und daraufhin innerhalb des Tetraeders entlang des Sehstrahls abgetastet werden. Da aber der Sehstrahl nicht nur einen, sondern ein ganzes Netz von Tetraedern traversiert, muss bei der Schnittpunktberechnung mit den Randflächen des Gitters begonnen werden. Dann kann ermittelt werden, zu welchem Tetraeder eine Randfläche gehört, wo der Sehstrahl austritt und welcher Tetraeder als nächstes vom Sehstrahl geschnitten wird. Da die Nachbarschaften der Tetraeder bekannt sind, kann der an die Austrittsfläche angrenzende Tetraeder, dessen Parameter als Eingabe für die Interpolation im nächsten Schritt dienen, leicht ermittelt werden. Dies wird so lange fortgesetzt, bis der Strahl das Volumen verlässt. Die Strahlverfolgung wird in Abschnitt 3.3.1 erläutert. Durch Sampling und Compositing kann der Beitrag berechnet werden, den das Volumen für das untersuchte Pixel liefert (Abschnitt 3.3.2).

Probleme treten auf bei nicht-konvexen Gittern. Das oben beschriebene Verfahren funktioniert nur, solange der Sehstrahl nur einmal in das Volumen eintritt und einmal austritt. Bei nicht-konvexen Gittern kann es sein, dass der Sehstrahl, nachdem er das Gitter verlassen hat, erneut in einen Teil des Gitters eintritt, beispielsweise bei einem Torus. Die Problematik wird in Abschnitt 3.3.3 näher erläutert.

3.3.1 Strahlverfolgung

Der folgenden Beschreibung des Strahlverfolgungsalgorithmus liegt ein Tetraedermodell mit folgenden Bezeichnungen zugrunde (Abbildung 6): Die vier Eckpunkte des Tetraeders werden durch den Buchstaben v (engl. vertex) bezeichnet. Demnach ergeben sich für die vier Eckpunkte die Bezeichnungen $\vec{v_0}$, $\vec{v_1}$, $\vec{v_2}$ und $\vec{v_3}$. Die Fläche f_n liegt dabei genau dem Eckpunkt $\vec{v_n}$ gegenüber, so dass sich für die Flächen f_0, f_1, f_2 und f_3 folgendes Schema ergibt:

$$\begin{aligned} f_0 &: \vec{v_1} \rightarrow \vec{v_3} \rightarrow \vec{v_2} \\ f_1 &: \vec{v_0} \rightarrow \vec{v_2} \rightarrow \vec{v_3} \\ f_2 &: \vec{v_3} \rightarrow \vec{v_1} \rightarrow \vec{v_0} \\ f_3 &: \vec{v_2} \rightarrow \vec{v_0} \rightarrow \vec{v_1} \end{aligned} \tag{1}$$

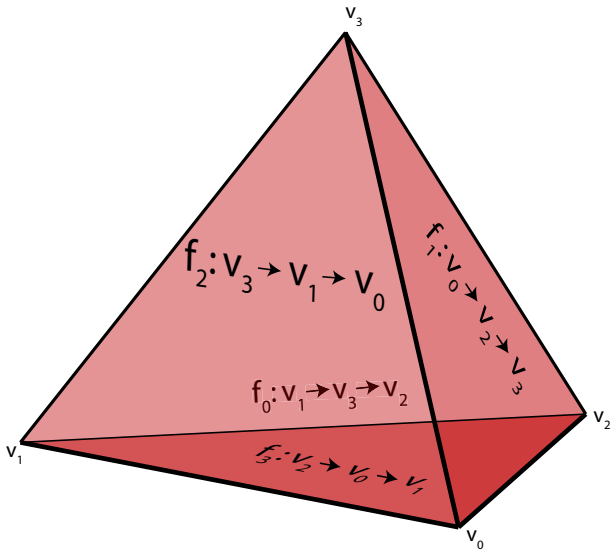

Abbildung 6: Tetraedermodell

Die Reihenfolge der Punkte wurde hier so gewählt, dass sich für jedes Dreieck eine Laufrichtung gegen den Uhrzeigersinn ergibt, wenn man von außen (d.h. von außerhalb des Tetraeders) auf die Fläche schaut.

Für jede Fläche des Tetraeders können die Normalen der Dreiecksflächen bestimmt werden:

$$\vec{n_0} = (\vec{v_1} - \vec{v_3}) \times (\vec{v_1} - \vec{v_2}) \tag{2}$$

$$\vec{n_1} = (\vec{v_0} - \vec{v_2}) \times (\vec{v_0} - \vec{v_3}) \tag{3}$$

$$\vec{n_2} = (\vec{v_3} - \vec{v_1}) \times (\vec{v_3} - \vec{v_0}) \tag{4}$$

$$\vec{n_3} = (\vec{v_2} - \vec{v_0}) \times (\vec{v_2} - \vec{v_1}) \tag{5}$$

Die Orientierung der Normalen zeigt hier nach außen (bezogen auf den Tetraeder).

Für jede Fläche lässt sich nun eine Ebenengleichung in der Hesse-Normalform aufstellen:

$$E_i : x \circ \vec{n_i} = d_i \tag{6}$$

Neben den Normalenvektoren $\vec{n_i}$wird dazu der Abstand zum Ursprung d_i benötigt. Dieser ergibt sich aus

$$d_0 = (\vec{n_0} \circ \vec{v_1}) \tag{7}$$

$$d_1 = (\vec{n_1} \circ \vec{v_0}) \tag{8}$$

$$d_2 = (\vec{n_2} \circ \vec{v_3}) \tag{9}$$

$$d_3 = (\vec{n_3} \circ \vec{v_2}) \tag{10}$$

Es sollen nun Eintritts- und Austrittspunkt eines Sehstrahls mit dem Tetraeder bestimmt werden. Der Sehstrahl sei gegeben durch eine Geradengleichung

$$r(\vec{\lambda}) = \vec{r_0} + \lambda \cdot \vec{t} \tag{11}$$

mit dem Augpunkt $\vec{r_0}$ und der Blickrichtung $\vec{t}$, siehe Abb. 7.

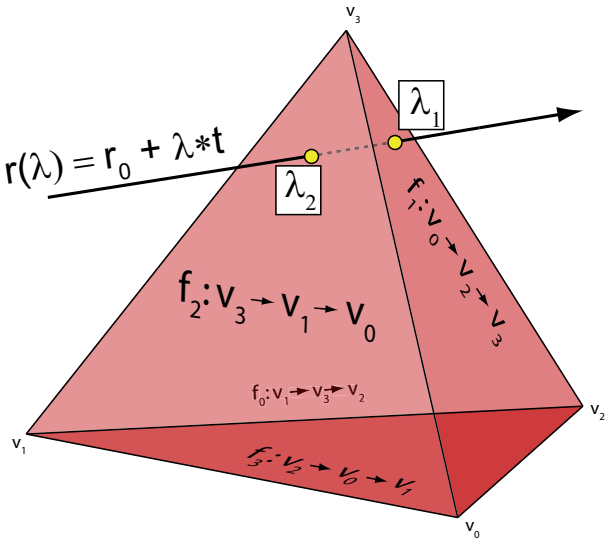

Abbildung 7: Berechnung von Eintritts- und Austrittspunkt

Zur Bestimmung des Austrittsdreiecks und damit des Folgetetraeders gibt es zwei Möglichkeiten. Die erste Möglichkeit ist, alle Schnittpunkte zwischen dem Strahl und den Ebenen zu berechnen und dann zu prüfen, welcher der Schnittpunkte innerhalb eines der Dreiecke liegt. Dieses

Dreieck ist dann das Austrittsdreieck. Die zweite Möglichkeit besteht darin, während des Samplingschrittes bei der baryzentrischen Interpolation so lange die vier Interpolationsgewichte zu betrachten bis eines davon negativ wird.

Exakte Bestimmung von Ein- und Austrittspunkt Zur Bestimmung der Schnittpunkte des Sehstrahls werden zunächst die Schnittpunkte mit den zu den Dreiecken gehörenden Ebenen bestimmt:

$$\vec{r}(\lambda_i) \circ \vec{n_i} = d_i \tag{12}$$

$$(\vec{r_0} + \lambda \cdot \vec{t}) \circ \vec{n_i} = d_i \tag{13}$$

$$(\vec{r_0} \circ \vec{n_i}) + \lambda_i(\vec{t} \circ \vec{n_i}) = d_i \tag{14}$$

$$\lambda_i = \frac{d_i - (\vec{r_0} \circ \vec{n_i})}{\vec{t} \circ \vec{n_i}} \tag{15}$$

Anschließend muss geprüft werden, ob der resultierende Punkt $\vec{x_i} = r(\lambda_i)$ innerhalb des jeweiligen Dreiecks liegt. Man kann dazu folgende Informationen ausnutzen:

- Da die Sehstrahlen schrittweise von einem Tetraeder in benachbarte Tetraeder verfolgt werden, kann man sicher sein, dass ein untersuchter Tetraeder vom Sehstrahl auch wirklich geschnitten wird.
- Wenn man entlang des Sehstrahls mit fester Schrittweite voranschreitet, ist bereits das λ_S für den aktuellen Punkt $\vec{r}(\lambda_S)$ bekannt, der entweder auf dem Rand oder innerhalb des Tetraeders liegt. Von den vier möglichen Schnittpunkten λ_i muss nur der kleinste herausgesucht werden, der größer als das λ des Eintrittspunktes ist[3]. Dies ist der Austrittspunkt aus dem aktuellen und somit der Eintrittspunkt des Sehstrahls in den nächsten Tetraeder.

Bestimmung des Folgetetraeders durch Beobachtung der baryzentrischen Koordinaten Für jeden Punkt auf dem Sehstrahl können die baryzentrischen Koordinaten bezüglich

[3]Dieser Vergleich führt allerdings in der Praxis aufgrund der Fließkommagenauigkeit zu numerischen Problemen.

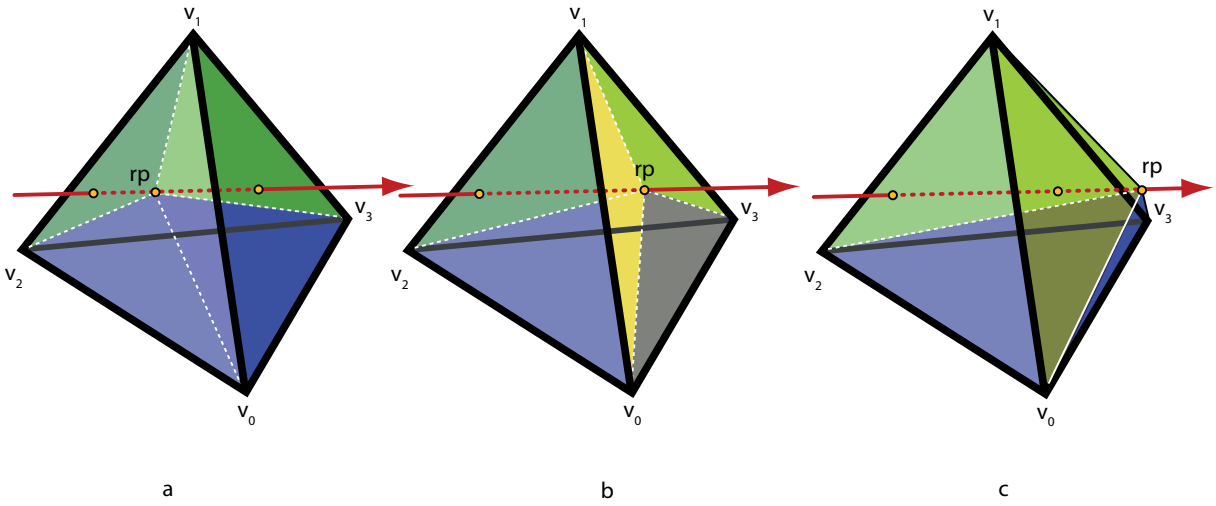

Abbildung 8: Baryzentrische Gewichte im Tetraeder. Befindet sich der Samplingpunkt nicht mehr im Tetraeder (c), wird eines der baryzentrischen Gewichte negativ.

des aktuellen Tetraeders bestimmt werden. Sie entsprechen den Volumen der vier Teiltetraeder, welche entstehen, wenn man die vier Dreiecke jeweils mit der aktuellen Strahlposition *rp (ray position)* als Spitze in neue Tetraeder unterteilt (siehe Abbildung 8). Wird der Betrag eines der Volumen dieser Tetraeder (vol_0 bis vol_3) negativ, kann man daraus ableiten, dass sich die Samplingposition auf dem Sehstrahl außerhalb des Tetraeders befindet. Außerdem weiß man dann, durch welches Dreieck der Sehstrahl den Tetraeder verlassen hat, nämlich durch die Grundfläche desjenigen Tetraeders, dessen Volumen negativ geworden ist. Damit ist das Austrittsdreieck bekannt, und über dieses Austrittsdreieck kann der Folgetetraeder bestimmt werden.

$$\begin{aligned}\vec{b_0} &= \vec{v_0} - r\vec{p} \\ \vec{b_1} &= \vec{v_1} - r\vec{p} \\ \vec{b_2} &= \vec{v_2} - r\vec{p} \\ \vec{b_3} &= \vec{v_3} - r\vec{p}\end{aligned} \tag{16}$$

$$
\begin{aligned}
vol_0 &= ((\vec{b}_2 \times \vec{b}_3) \circ \vec{b}_1) \\
vol_1 &= ((\vec{b}_3 \times \vec{b}_2) \circ \vec{b}_0) \\
vol_2 &= ((\vec{b}_0 \times \vec{b}_1) \circ \vec{b}_3) \\
vol_3 &= ((\vec{b}_1 \times \vec{b}_0) \circ \vec{b}_2)
\end{aligned} \tag{17}
$$

3.3.2 Sampling und Compositing

Durch die im vorangegangenen Kapitel beschriebene Strahlverfolgung kann bestimmt werden, durch welche Tetraeder der Sehstrahl verläuft. Für jeden Punkt auf diesem Sehstrahl ist bekannt, in welchem Tetraeder er sich befindet. Ziel ist es, für jeden Sehstrahl, d.h. für jeden Pixel der Bildebene, einen Farbwert zu bestimmen. Dazu wird der Volumendatensatz entlang des Sehstrahls abgetastet und für jeden Abtastpunkt der Beitrag bestimmt, den das Volumen an diesem Punkt zum Farbwert des Pixels leistet (*Sampling*). Die Beiträge aller Abtastpunkte werden zur Bestimmung des entgültigen Farbwerts im Pixel verrechnet (*Compositing*).

Sampling Jeder Punkt auf dem Sehstrahl kann nicht nur durch Einsetzen eines Lambdawertes in die Geradengleichung ausgedrückt werden, sondern auch durch baryzentrische Koordinaten bezüglich des Tetraeders, in dem er sich befindet. Zur Bestimmung der Werte für Farbe und Opaziät an diesem Punkt müssen an der entsprechenden Stelle Datenwerte aus der 3D-Textur entnommen werden. Dafür werden die entsprechenden Texturkoordinaten des Abtastpunktes benötigt. Farbe und Opazität werden dann letztendlich über eine Transferfunktion bestimmt. Texturkoordinaten für den Lookup in der Volumentextur liegen aber nur für die Eckpunkte der Tetraeder vor. Für beliebige Punkte auf dem Sehstrahl müssen sie mittels baryzentrischer Interpolation berechnet werden. Um die Texturkoordinaten P_{uvw} eines beliebigen Punktes P_{xyz} zu berechnen, müssen zunächst die baryzentrischen Gewichte für diesen Punkt im Tetraeder ermittelt werden. Die Berechnung der baryzentrischen Gewichte wurde schon im vorhergehenden Abschnitt in den Gleichungen 16 und 17 erläutert. vol_0 bis vol_3 beschreiben die Volumen der Teiltetraeder. Teilt man sie durch das Gesamtvolumen des Tetraeders erhält man die baryzen-

tischen Gewichte (siehe Kap. 3.3.1). Der Punkt P_{xyz} auf dem Sehstrahl kann dann ausgedrückt werden durch

$$P_{xyz} = \frac{vol_0 * \vec{v}_0 + vol_1 * \vec{v}_1 + vol_2 * \vec{v}_2 + vol_3 * \vec{v}_3}{vol_0 + vol_1 + vol_2 + vol_3} \tag{18}$$

Gesucht sind aber die Texturkoordinaten uvw_P in die Volumentextur. Sie ergeben sich aus der Linearkombination der baryzentrischen Gewichte mit den Texturkoordinaten der Eckpunkte des Tetraeders, geteilt durch das Gesamtvolumen des Tetraeders:

$$P_{uvw} = \frac{vol_0 * uvw_0 + vol_1 * uvw_1 + vol_2 * uvw_2 + vol_3 * uvw_3}{vol_0 + vol_1 + vol_2 + vol_3} \tag{19}$$

Für jeden Abtastpunkt kann mittels der so berechneten Texturkoordinaten der entsprechende Datenwert in der Volumentextur ermittelt werden. Durch die Anwendung einer Transferfunktion ergibt sich für jeden Samplingpunkt ein RGBA-Wert. Der endgültige Farbwert für das untersuchte Pixel ergibt sich jedoch erst aus der Integration über die Emissions- und Absorptionswerte aller Abtastpunkte beim Compositing, d.h. über die diskrete Lösung des Volume Rendering Integrals.

Compositing Im Compositingschritt muss für jedes Pixel das Volume Rendering Integral über alle abgetasteten Samplingwerte approximiert werden. Um den endgültigen Farbwert C in einem Pixel zu bestimmen, muss über die Farbwerte (Emission) und Opazitätswerte (Absorption) entlang des Sehstrahls $\vec{r}(\lambda)$ integriert werden, der für das Pixel in das Volumen geschossen wird. An jedem Punkt entlang des Sehstrahls wird durch das Sampling der 3D-Textur ein Skalarwert $s(\vec{r}(\lambda))$ ermittelt, für den (durch eine Transferfunktion) ein Absorptionskoeffizient $\kappa(s(\vec{r}(\lambda)))$ und ein Emissionswert (Farbwert) $c(s(\vec{r}(\lambda)))$ gegeben sind. Mit dem Volume Rendering Integral, gegeben durch Gleichung 20, kann nun die Farbe C des Pixels berechnet werden, indem über

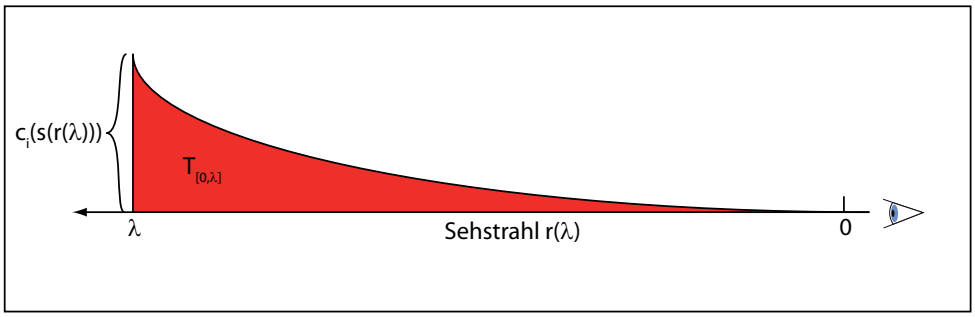

Abbildung 9: Emission und Absorption entlang des Sehstrahls

alle Emissions- und Absorptionswerte entlang des Sehstrahls bis zum Austrittspunkt D aus dem Volumen integriert wird (siehe [6]). Zur Veranschaulichung dient Abbildung 9.

$$C \;=\; I(D) \;=\; \int_0^D c(s(\vec{r}(\lambda)))\; e^{-\int_0^{\lambda}\; \kappa(s(\vec{r}(\lambda')))\; d\lambda'}\; d\lambda \tag{20}$$

Der Farbbeitrag an einem Punkt $\vec{r}(\lambda)$ besteht also aus der Emission c, die an dieser Stelle gegeben ist, multipliziert mit der bis zu diesem Punkt kumulierten Absorption. Das Volume Rendering Integral kann durch die Anwendung eines Compositingverfahrens numerisch berechnet werden. Die kumulierte Absorption bis zum Punkt $\vec{r}(\lambda)$ auf dem Sehstrahl,

$$T_{[0,\lambda]} \;=\; e^{-\int_0^{\lambda}\; \kappa(s(\vec{r}(\lambda')))\; d\lambda'} \quad , \tag{21}$$

kann diskretisiert werden durch

$$T_{[0,\lambda]} \;\approx\; e^{-\sum_{i=0}^{\lfloor \lambda/d \rfloor} \kappa(s(\vec{r}(i\cdot d)))\; \cdot\; d} \quad , \tag{22}$$

wobei d den Abstand zwischen zwei Samplingpunkten beschreibt. Gleichung 22 kann umgeformt werden zu

$$T_{[0,\lambda]} \;\approx\; \prod_{i=0}^{\lfloor \lambda/d \rfloor} e^{-\kappa(s(\vec{r}(i\cdot d)))\; \cdot\; d} \quad . \tag{23}$$

Mit der Opazität A, gegeben durch

$$A_i = 1 - e^{-\kappa(s(\vec{x}(i\cdot d)))\; \cdot\; d} \quad , \tag{24}$$

kann die Gleichung 23 umgeschrieben werden zu

$$T_{[0,t]} \approx \prod_{i=0}^{\lfloor t/d \rfloor} (1 - A_i) \tag{25}$$

Die Emission im i-ten Strahlsegment, kann analog ausgedrückt werden durch

$$C_i = c(s(\vec{r}(i \cdot d))) \cdot d \tag{26}$$

Damit kann das Volume Rendering Integral diskret approximiert werden durch

$$C_{approx} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{27}$$

Gleichung 27 kann nun iterativ durch Front-to-Back oder Back-to-Front Compositing gelöst werden. In der Implementierung soll jedoch die Optimierungsmöglichkeit durch Early Ray Termination genutzt werden. Hierbei wird die Berechnung entlang des Sehstrahls abgebrochen, sobald die akkumulierte Opazität A nahe 1.0 liegt. Early Ray Termination ist nur bei Front-to-Back-Compositing möglich. Die neuen Werte für die Farbe C'_i und die Opazität A'_i berechnen sich aus den Gleichungen 28 und 29

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{28}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{29}$$

wobei zu Beginn $C'_0 = 0$ und $A'_0 = 0$ gesetzt werden.

3.3.3 Problematik nicht-konvexer Gitter

Die Strahlverfolgung, das Sampling und das Compositing, so wie sie oben beschrieben wurden, funktionieren auf diese Weise nur bei konvexen Geometrien. Bei nicht konvexen Gittern, d.h. Gittern mit Löchern oder Hohlräumen, tritt der Sehstrahl nicht nur einmal in die Geometrie ein- und aus, sondern mehrfach (siehe Abbildung 10). Die Berechnung des Farbwerts für das

betreffende Pixel soll natürlich erst abbrechen, wenn der Sehstrahl *endgültig* aus dem Volumen ausgetreten ist. Ob ein Austrittspunkt der letzte Austrittspunkt aus dem Volumen ist oder nur den Eintritt in einen Hohlraum markiert, kann aber mit dem bisherigen Ansatz nicht entschieden werden. Abhilfe schafft hier das Depth Peeling ([13], [3]), eine Technik, bei der das Volumen in mehreren Layers gerendert wird. Die Lösung der Problematik wird ausführlich in Kapitel 4.3.6 beschrieben.

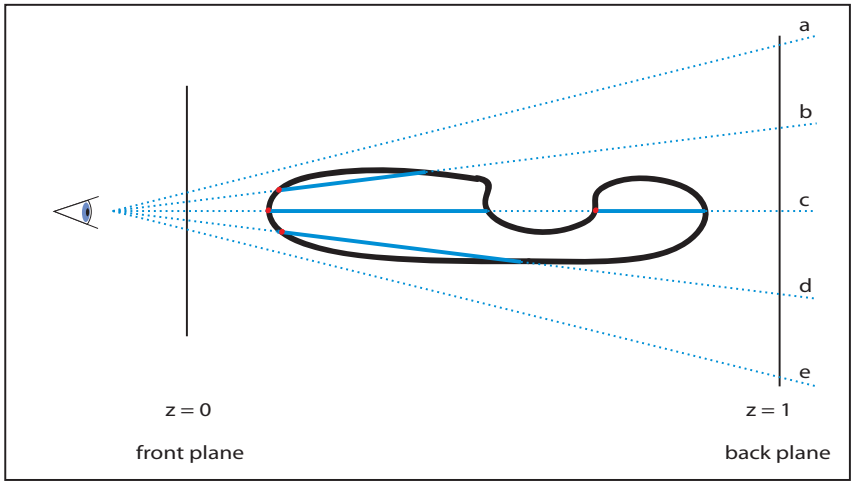

Abbildung 10: Schema des Raycasting bei nicht-konvexen Geometrien. Die Problematik wird beim Vergleich zwischen den Sehstrahlen b und d bzw. c deutlich: Während b und d jeweils nur einen Eintrittspunkt haben durchläuft Strahl c mehrfach die Geometrie.

3.4 Deformation

Im Kapitel 2.4 wurde schon angedeutet, welche Schwierigkeiten bei der Deformation von Volumendatensätzen bestehen, die durch uniforme rectilineare Gitter diskretisiert werden: Schnittpunktberechnung und Interpolation werden schwierig, weil durch die Verschiebung einzelner Gitterpunkte die Planarität verloren geht. Wenn ein Tetraedernetz als Proxygeometrie für die Deformation des uniform diskretisierten Datensatzes benutzt wird, bestehen diese Probleme nicht. Bei der Erstellung der nötigen Datenstrukturen müssen für jeden Vertex des Tetraedernetzes die entsprechenden Texturkoordinaten bezüglich des Volumendatensatzes berechnet und

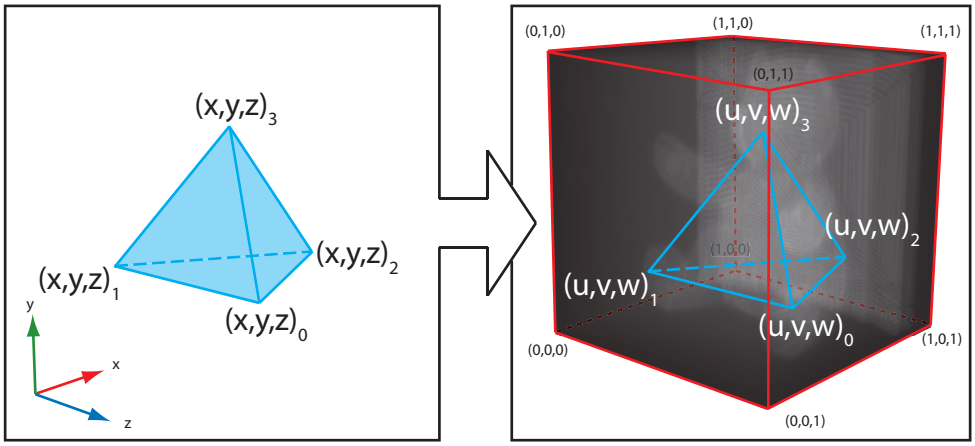

Abbildung 11: Zusammenhang zwischen den Vertexkoordinaten und den Texturkoordinaten

in einer separaten Textur abgelegt. Der Zusammenhang zwischen der Textur, die die Vertexkoordinaten hält und der entsprechenden Texturkoordinatentextur ist in Abbildung 11 dargestellt.

Werden nun die Vertices des Tetraedernetzes transformiert ohne dass die Texturkoordinaten aktualisiert werden, kann der auf dem uniformen Gitter basierende Volumendatensatz auf einfache Weise deformiert werden, ohne dass Anpassungen am Raycasting-Algorithmus vorgenommen werden müssen.

3.5 Zusammenfassung

Die wichtigsten Aspekte des Ansatzes zur Volumendarstellung und -deformation mittels Tetraedergittern wurden in diesem Kapitel theoretisch beleuchtet: Geometrie und Topologie des Tetraedernetzes werden als Offset-Texturen im lokalen Speicher der Grafikkarte gespeichert, um später eine schnelle Berechnung auf der GPU zu ermöglichen. Hinsichtlich des Raycasting-Algorithmus wurde erläutert, wie die Strahlverfolgung abläuft, d.h. wie für einen Sehstrahl die Tetraeder bestimmt werden, durch die er verläuft. Durch Sampling und Front-to-Back-Compositing entlang des Sehstrahls wird schließlich der resultierende Farbwert für das untersuchte Pixel berechnet. Diese Berechnung wird schwierig, sobald sich Hohlräume im Gitter befinden, da nicht bestimmt werden kann, ob der Sehstrahl endgültig oder nur temporär aus dem Volumen austritt. Eine

Lösung des Problems wird im folgenden Kapitel durch Depth Peeling implementiert.
Die Deformation des Datensatzes erfolgt über einen einfachen Mechanismus: Das Volumen wird deformiert, indem die *xyz*-Koordinaten der Vertices transformiert werden, die *uvw*-Texturkoordinaten an den Vertices aber gleich bleiben.

4 Implementierung

4.1 Überblick

Die Gliederung des Implementierungskapitels entspricht der Gliederung des vorangegangenen Theoriekapitels. In Kapitel 4.2 wird beschrieben, in welcher Form die Geometrie- und Topologiedaten des Tetraedermeshes ausgelesen und in Texturen abgelegt werden. Im darauffolgenden Abschnitt 4.3 geht es um die Bestandteile des Raycastingschrittes (Bestimmung der Eintrittspunkte, Strahlverfolgung, Sampling und Compositing) und Optimierungen des Ablaufs (Early Ray Termination, Depth Peeling). Abschnitt 4.4 handelt schließlich von der Implementierung der Deformation.

4.2 Datenstrukturen

4.2.1 Vom Mesh zum XML

Im Folgenden wird davon ausgegangen, dass für ein gegebenes Volumen ein Tetraedergitter existiert, das optimal an das Volumen angepasst ist. Optimal bedeutet hier, dass es in Bereichen hoher Frequenzen des Volumens fein aufgelöst ist und in Bereichen niedriger Frequenzen grob. Dieses Tetraedermesh liegt in einer XML-Beschreibung vor, die im darauf folgenden Schritt ausgelesen wird. Die Beschreibung enthält lediglich eine Liste aller Vertexkoordinaten des Gitters und für jeden Tetraeder die vier Indizes seiner Vertices in diese Liste, und zwar in der richtigen Reihenfolge (siehe Abschnitt 3.3.1). Die Zusammenhänge sind in Abbildung 12 verdeutlicht.

Die in dieser Arbeit verwendeten Gitter wurden mit der freien Software Gmsh [14] erstellt und in XML-Dateien konvertiert.

4.2.2 Auslesen der Grid-Beschreibung

In einer Initialisierungsphase werden die Daten aus der Grid-Datei ausgelesen, aus diesen Daten Listen gebildet und schließlich die Inhalte dieser Listen in Texturen geschrieben. Da die

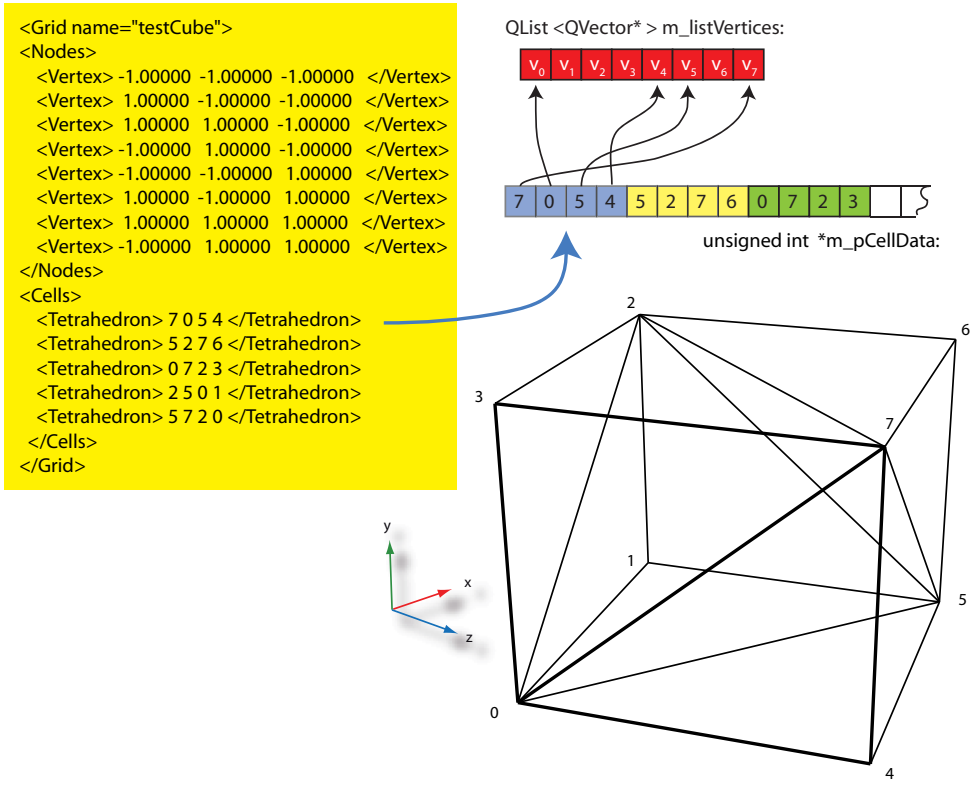

Abbildung 12: Zusammenhang zwischen der Geometrie und der XML-Beschreibung

Zusammenhänge zwischen den unterschiedlichen Listen anfangs schwierig zu durchschauen sein mögen, soll an dieser Stelle schon auf Abbildung 13 verwiesen werden, in der die Zusammenhänge zwischen den Listen verdeutlicht werden.

Auslesen der Vertex-Koordinaten und der Tetraeder-Daten

Die in XML vorliegende Beschreibung des Tetraedergitters wird in der Klasse `CTetraGrid` über die Funktion `parseXML` eingelesen. Wesentliches Ergebnis dieses Schrittes sind eine Vertexliste (`QList<CVector*> m_listVertices`) und eine Liste der Tetraederzelldaten (`unsigned int *m_pCellData`). In der Vertexliste sind für jeden Vertex die *xyz*-Koordinaten (als `CVector`) gespeichert. Die Vertices werden in der Reihenfolge gespeichert, in der sie aus der Grid-Datei ausgelesen werden. Für jeden Tetraeder werden in `m_pCellData` die vier Eckpunkte gespeichert, und zwar als Gruppen von vier `int`-Indizes in `m_listVertices`. Der Zusammenhang zwischen dem Tetraedernetz, der XML-Beschreibung und den beiden Datenstrukturen ist in Abbildung 12 anhand eines einfachen Kubus veranschaulicht, der in 5 Tetraeder unterteilt ist.

Generierung von Face-, Boundary Face- und Tetraederlisten

Die Bestimmung weiterer Datenstrukturen des - an dieser Stelle eigentlich schon ausreichend rekonstruierbaren - Tetraedernetzes in weiteren Datenstrukturen dient dem Ziel, den Raycasting-Algorithmus effizient zu implementieren.

Bis zu dieser Stelle liegt zwar eine ausreichende Beschreibung des Meshes vor, da durch die Koordinaten der Vertices und die Zusammensetzung von jeweils vier Vertices schon bekannt ist, welche Vertices einen Tetraeder bilden. Die beiden Listen `m_pCellData` und `m_listVertices` reichen aber nicht aus, um beim Raycasting effizient und schnell zu ermitteln, welche Tetraeder der Sehstrahl traversiert. Was gebraucht wird ist also eine Repräsentation des Meshes, die es ermöglicht, auf einfache Weise die Nachbarschaften der Tetraeder abzufragen. Zu diesem Zweck werden Listen der Dreiecksflächen und der Tetraeder benötigt. Diese beiden Listen können anhand der schon existierenden Vertex- und Zell-Liste erstellt werden. Zusätzlich dazu wird eine separate Liste der Randflächen gebildet, die für die spätere Rasterisierung notwendig ist.

```
for(int cell = 0; cell < m_nNumCells; ++cell)
{
    int basis = cell*4;
    //konstruiere die 4 möglichen faces
    newFace0 = new CTetraFace(m_pCellData[basis + 0],
                              m_pCellData[basis + 2],
                              m_pCellData[basis + 1]);
    newFace1 = new CTetraFace(m_pCellData[basis + 0],
                              m_pCellData[basis + 3],
                              m_pCellData[basis + 2]);
    newFace2 = new CTetraFace(m_pCellData[basis + 0],
                              m_pCellData[basis + 1],
                              m_pCellData[basis + 3]);
    newFace3 = new CTetraFace(m_pCellData[basis + 1],
                              m_pCellData[basis + 2],
                              m_pCellData[basis + 3]);
    // ...
}
```

Listing 1: Konstruktion der möglichen Dreiecksflächen

Die Dreiecksflächen

In der Funktion `void CTetraGrid::createFaceList()` werden die Listen der Dreiecksflächen und der Tetraederobjekte erstellt. Wichtig ist an dieser Stelle, den Unterschied zwischen der (schon vorhandenen) Liste `m_pCellData` und der (noch zu erstellenden) Liste der Tetraeder `m_listTetrahedra` zu verdeutlichen: `m_pCellData` speichert nur `int`-Werte als Indizes in die Vertexliste. Welche Vertices in welcher Orientierung eine Dreiecksfläche bilden, verrät `m_pCellData` nicht. `m_listTetrahedra` dagegen ist die Liste, die die tatsächlichen Tetraeder-*Objekte* aufnimmt. Zunächst wird die Liste aller Dreiecksflächen erstellt (`QList<CTetraFace* > m_listFaces`). Für jeden Tetraeder sind also aus den vier Vertices die vier möglichen Dreiecksflächen (`CTetraFace`) zu ermitteln. Dazu wird die Zellliste `m_pCellData` durchlaufen. Aus den Vertices 0 bis 3 einer Zelle werden die vier möglichen Dreiecke konstruiert. Die Liste der Flächen soll dabei jede Fläche nur einmal enthalten. Um dies zu erreichen, werden die drei Vertexindizes jeder Fläche der Größe nach sortiert. Dann werden die Flächen selbst

lexikographisch sortiert in die Liste eingefügt. So kann auf einfache Weise festgestellt werden, ob eine Dreiecksfläche schon in der Liste enthalten ist oder nicht.

Ob eine Fläche vor oder hinter einer anderen Fläche eingeordnet wird, ergibt sich aus dem paarweisen Vergleich der Vertexindizes. Eine Fläche ist größer/kleiner als eine andere, wenn ihr n-ter Vertexindex größer/kleiner als der n-te Vertexindex der Vergleichsfläche ist (für $n = 0...2$), wobei n nur bei Gleichheit hochgezählt wird. Wurden alle Dreiecksflächen des Tetraedernetzes in die Liste einsortiert, können darauf aufbauend die Tetraederobjekte konstruiert werden.

Die Tetraeder(-objekte)

Ein Tetraederobjekt soll nicht nur über die Informationen verfügen, aus welchen Flächen (*faces*) es besteht, sondern es muss auch die Orientierung seiner Flächen kennen. Die Orientierung einer Fläche muss bekannt sein, um zu ermitteln, zu welchem Tetraeder sie gehört, bzw. welcher Nachbartetraeder sich mit einem gegebenen Tetraeder eine Fläche "teilt". Für jedes neu zu erstellende Tetraederobjekt werden also die vier Indizes seiner Flächen in `m_listFaces` gesucht. Um diese zu ermitteln werden die Flächen für jede Zelle, wie im vorangegangenen Schritt, aus den Indizes von `m_pCellData` gebildet und die Indizes der Größe nach sortiert. Durch die Funktion `int CTetraGrid::findFace(CTetraFace *pNewFace)` kann dann diese Fläche in `m_listFaces` gesucht und ihr Index in `m_listFaces` ermittelt werden. Zusätzlich dazu muss aber auch die Orientierung jeder Fläche ermittelt werden. Dazu kann folgende Kenntnis ausgenutzt werden: Die Flächen werden zunächst so gebildet, dass die Normale vom Tetraeder aus betrachtet nach außen zeigt (siehe 4.2.2). Mussten die beiden hinteren Indizes vertauscht werden um die Vertex-Indizes der Größe nach zu sortieren, dann hat sich die Orientierung der Fläche geändert. Das heißt, dass somit die Normale nicht mehr nach außen, sondern nach innen zeigt. So kann für jede Fläche gesagt werden, in welcher Orientierung das `CTetraFace`-Objekt zum aktuell untersuchten Tetraeder gehört. Das Tetraederobjekt kann nun erstellt werden, indem dem Konstruktor von `CTetrahedron` die vier Indizes und die vier Orientierungen der Flächen übergeben werden. Das Tetraederobjekt wird abschließend in `QList<CTetrahedron *> m_listTetrahedra` eingefügt. Ein Tetraeder kann nun nach seinen vier Flächen "gefragt" werden. Wichtiger ist es aber, dass eine Fläche die beiden Tetraeder kennt, zu denen sie gehört. Durch die Funkti-

```
for(int cell = 0; cell < m_nNumCells; ++cell)
{
    //...
    m_listFaces.at(nFaceIdx0)->setNeighbour(nOrient0, cell);
    m_listFaces.at(nFaceIdx1)->setNeighbour(nOrient1, cell);
    m_listFaces.at(nFaceIdx2)->setNeighbour(nOrient2, cell);
    m_listFaces.at(nFaceIdx3)->setNeighbour(nOrient3, cell);
    //...
}
```

Listing 2: Jedem der vier Faces wird "mitgeteilt", in welcher Orientierung es zur gerade untersuchten Tetraederzelle gehört.

on `void CTetraFace::setNeighbour(FaceOrientation f, int nNeighbour)` muss deshalb noch jeder Fläche mitgeteilt werden, in welcher Orientierung sie zur aktuell untersuchten Zelle gehört: Somit kann für eine Fläche erfragt werden, in welcher Normalenrichtung sich welcher Tetraeder anschmiegt. Da bei der Erzeugung eines `CTetraFace`-Objekts die beiden möglichen Indizes der Nachbartetraeder initial mit -1 belegt werden und diese nur durch obenstehenden Code überschrieben werden, ergibt sich automatisch für die Randflächen der Nachbarindex -1, was bedeutet, dass sich dort kein Tetraeder mehr befindet.

Die Außenflächen (*boundary faces*)

Die Liste der Randflächen ist wichtig, weil auf ihr (d.h. eigentlich nur auf der dem Betrachter zugewandten Teilmenge, den *front faces*), die Eintrittspunkte der Sehstrahlen für das Raycasting liegen. Um die Eintrittspunkte zu bestimmen werden also später nur die *boundary faces* rasterisiert. Gemäß der im vorigen Abschnitt erläuterten Möglichkeit, für jede Fläche ihre beiden Nachbarn zu erfragen, ist es nun ein Leichtes, die Liste aller Außenflächen zu erstellen. Es muss lediglich für jede Fläche geprüft werden, ob einer der beiden Nachbartetraeder einen Index von -1 hat. Falls dies so ist, handelt es sich um eine Randfläche, und die Fläche wird in die Liste `QList<CTetraFace* > m_listBoundaryFaces` aufgenommen.

An dieser Stelle soll eine kurze Zusammenfassung wieder den Blick auf das eigentliche Ziel, die Implementierung des Raycasting, richten: Mit den erzeugten Listen kann man nun einen

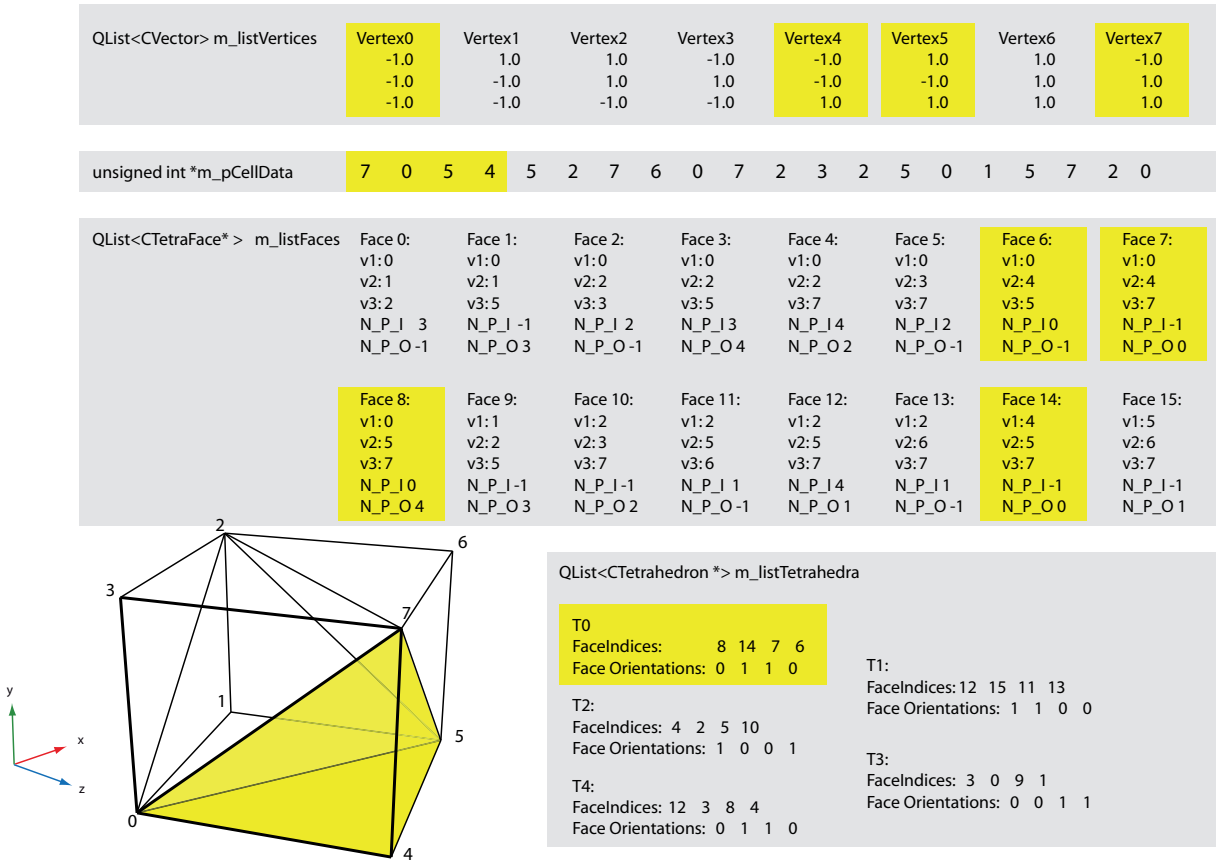

Abbildung 13: Alle Listen auf einen Blick. Anmerkung: N_P_I bedeutet NORMAL_POINTS_INWARDS, N_P_O entsprechend NORMAL_POINTS_OUTWARDS.

Sehstrahl, der in das Tetraedernetz eintritt und es an anderer Stelle wieder verlässt, von Tetraeder zu Tetraeder verfolgen. Ein Sehstrahl tritt über eine Randfläche in das Tetraedernetz ein. Über die Randfläche kann der zugehörige Tetraeder erfragt werden. Durch die im Theoriekapitel (3.3.1) erläuterten Möglichkeiten der Bestimmung des Austrittspunkts kann die Austrittsfläche bestimmt werden. Kennt man die Austrittsfläche, so kennt man den Tetraeder, in dem sich der Strahl im nächsten Schritt befindet. Dies setzt sich so lang fort, bis der Sehstrahl eine Fläche schneidet, an die kein weiterer Tetraeder angeschlossen ist, d.h. bis der Sehstrahl aus dem Volumen austritt.

Bis hierhin ist die Geometrie und Topologie allerdings rein im Hauptspeicher abgebildet. Das Ziel ist aber die Echtzeitfähigkeit, und so wird im nächsten Abschnitt beschrieben, wie die erstellten

Vertextextur	`VERTEXTEXTURE`	`2D RGB`
Tetraedertextur	`VERTEX_INDEX_TEXTURE0`	`2D RGBA`
Tetraedertextur	`VERTEX_INDEX_TEXTURE1`	`2D RGBA`
Nachbarschaftentextur	`NEIGHBOR_INDEX_TEXTURE0`	`2D RGBA`
Nachbarschaftentextur	`NEIGHBOR_INDEX_TEXTURE1`	`2D RGBA`
Texturkoordinatentextur	`TEXCOORDTEXTURE`	`2D RGB`
3D Volumentextur (Datensatz)	`VOLUMETEXTURE`	`3D LUMINANCE`
1D Transferfunktion	`TRANSFERFUNCTION`	`3D LUMINANCE`

Tabelle 1: Namenskonventionen für die Texturen

Listen in Texturen abgebildet werden, die in den lokalen Speicher der Grafikhardware geladen werden können.

4.2.3 Speicherung der Geometrie und Topologie in Texturen

Im vorigen Schritt wurden Geometrie und Topologie des Tetraedernetzes rekonstruiert und diese Daten in Listen abgelegt. Ein zentrales Konzept des vorliegenden Ansatzes und Voraussetzung für die Implementierung in Hardware ist die Speicherung der Geometrie und Topologie im lokalen Speicher der Grafikkarte. Diese Aufgabe übernimmt die Klasse `CTetraTextureHandler`. Um Verwirrungen zu vermeiden, werden im Folgenden die Namen der Texturen verwendet, die auch bei der Implementierung benutzt wurden. Sie sind in Tabelle 1 aufgeführt.

Die Vertextextur Zur Speicherung der Vertexkoordinaten des Tetraedernetzes dient eine zweidimensionale `RGB float`-Textur (`VERTEXTEXTURE`). In jedes Texel kann ein Vertex eingetragen werden, indem seine x, y und z-Werte in den R,G und B-Komponenten gespeichert werden. Die Größe der Textur wird in Abhängigkeit der Anzahl der Vertices *numVertices* des Meshes bestimmt. Als Breite bzw. Höhe der Textur wird die kleinstmögliche Zweierpotenz 2^n gewählt für die gilt:

$$numVertices \leq (2^n)^2, \quad n \in \mathbb{N} \tag{30}$$

Die Einfügereihenfolge der Vertices in die Textur entspricht der Reihenfolge der Vertices in `m_listVertices`. Die xyz-Werte der Vertices werden in

`void CTetraTextureHandler::generateVertexTexture(int numNodes)` der Reihe nach aus `m_listVertices` herausgelesen und in das Array `float *m_pTextureDataVertexData` geschrieben, welches der Vertextextur (`VERTEXTEXTURE`) übergeben wird. Ist nun der Index eines Vertices bekannt, können seine *xyz*-Koordinaten über einen Texture Lookup in der Vertextextur nachgesehen werden. Dazu muss lediglich der Index in Abhängigkeit der Texturdimensionen in die Texturkoordinaten *a*,*b* umgerechnet werden.

Die 3D-Texturkoordinaten Den Vertices des Tetraedernetzes (*xyz*) entsprechen bestimmte Voxel im Volumendatensatz, der dargestellt und deformiert werden soll. Deshalb müssen für jeden Vertex die entsprechenden *uvw*-Koordinaten bezüglich der 3D-Textur, die den Volumendatensatz enthält, berechnet werden, d.h. sie müssen auf den Bereich zwischen 0.0 und 1.0 abgebildet werden (zur Theorie dazu siehe Abbildung 11 in Kap. 3.2). Diese Texturkoordinaten werden der 2D-Textur `TEXCOORDTEXTURE` übergeben. Die Größe der `TEXCOORDTEXTURE` gleicht der Größe der im vorangegangenen Schritt erstellten `VERTEXTEXTURE`. Das hat den Vorteil, dass für einen gegebenen Vertexindex *i* nur einmal die Texturkoordinaten durch `Index_To_TexCoord()` berechnet werden müssen. Diese können dann sowohl zum Abfragen der *xyz*- als auch der *uvw*-Koordinaten genutzt werden, weil *xyz*- und *uvw*-Koordinaten an gleicher Stelle quasi deckungsgleich gespeichert sind.

Zur Berechnung von *uvw* werden zunächst im Vertexarray `m_listVertices` die Minimal- und die Maximalwerte für x, y und z berechnet. Die entsprechenden Texturkoordinaten werden dann in der Funktion `void CTetraTextureHandler::generateTexCoordTexture()` berechnet (Listing 4.2.3).

Wurden für alle Vertexkoordinaten die entsprechenden Texturkoordinaten in die `VOLUMETEXTURE` berechnet und in `m_pTexCoordTexture` eingefügt, wird `m_pTexCoordTexture` an die 2D-Textur `TEXCOORDTEXTURE` übergeben.

Die Tetraeder Sollen die Koordinaten der Vertices eines Tetraeders herausgefunden werden, muss lediglich unter den vier Indizes ein Lookup in der Vertextextur vorgenommen werden. Es

```
//...
for(it = m_listVertices->begin(); it != m_listVertices->end(); it++)
{
   CVector &vector = *it;
   double x, y, z;
   vector.get(x, y, z);
   fU = (x-xMin)/(xMax-xMin);
   fV = (y-yMin)/(yMax-yMin);
   fW = (z-zMin)/(zMax-zMin);
   m_pTexCoordTexture[count + 0] = fU ;
   m_pTexCoordTexture[count + 1] = fV ;
   m_pTexCoordTexture[count + 2] = fW ;
   count+=3;
}
//...
```

Listing 3: Berechnung der auf den Bereich zwischen 0.0 und 1.0 skalierten *uvw*-Texturkoordinaten aus den *xyz*-Vertexkoordinaten

soll aber nicht in jedem Schritt eine Umrechnung von Index zu Texturkoordinaten stattfinden müssen. Deshalb werden bei der Erstellung der Tetraedertextur(en) nicht die vier Vertexindizes eingetragen, sondern deren Texturkoordinaten. Für jeden Vertex v_n werden deshalb s_n, t_n bezüglich der `VERTEXTEXTURE` berechnet. Für jeden Tetraeder müssen also acht Werte gespeichert werden. Dazu werden zwei RGBA-float-Texturen erstellt, die man sich als übereinanderliegend vorstellen kann. Zwei übereinanderliegende Texel nehmen jeweils acht Werte, d.h. vier Texturkoordinatenpaare auf. Die Größe dieser Texturen ergibt sich aus der Anzahl der Tetraeder des Meshes. In der ersten der beiden Texturen können im RG-Kanal die Texturkoordinaten s,t für den ersten Vertex gespeichert werden, im BA-Kanal die Koordinaten s,t für den zweiten Vertex. Die Koordinaten für den dritten und vierten Vertex werden analog in der zweiten Textur im gleichen Texel gespeichert. Wenn für einen Tetraeder mit gegebenem Index die Vertexkoordinaten herausgefunden werden müssen, kann dadurch an der gleichen Stelle in zwei Texturen nachgeschaut werden und die vier Texturkoordinatenpaare seiner Vertices gefunden werden. Die Texturkoordinaten können dann benutzt werden, um die tatsächlichen vier *xyz*-Koordinatenpaare in der Vertex-Textur nachzusehen.

Der Lookup der vier Vertexkoordinaten ist in Abbildung 14 dargestellt. Im Hinblick auf den

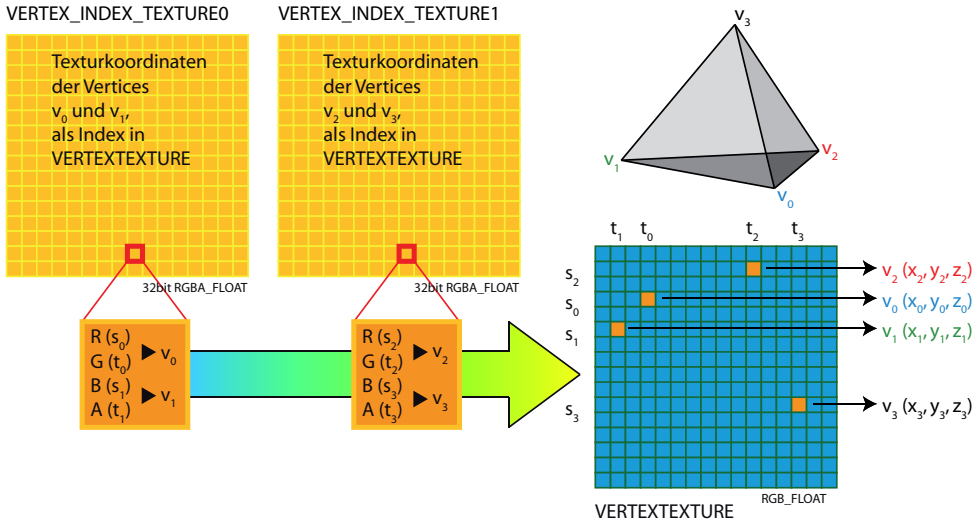

Abbildung 14: Lookup der Vertexkoordinaten für einen Tetraeder

Raycasting-Algorithmus muss nun noch dafür gesorgt werden, dass die Nachbarschaften ebenso effizient gespeichert werden, damit der Sehstrahl von einem Tetraeder zum nächsten Tetraeder allein durch Texture Lookups verfolgt werden kann.

Nachbarschaften Auch die Nachbarschaftsbeziehungen sollen in Texturen gespeichert werden. Dazu werden wieder zwei 2D-Texturen (`RGBA float`) erstellt. Die Texturen sind genauso groß wie die Tetraedertexturen. In den beiden Texturen können, analog zum Vorgehen bei den Tetraedervertices, vier Texturkoordinatenpaare gespeichert werden. Die Texturkoordinaten beziehen sich hier aber auf die Tetraedertextur. Beispielsweise werden die Texturkoordinaten s,t des Nachbartetraeders an Face0 von Tetraeder 6 als R und G in Texel 6 der ersten Textur gespeichert, Nachbar an Face1 in B und A des Texels. Die beiden anderen Nachbarn werden in der zweiten Textur analog gespeichert. In Abbildung 15 ist dieser Zusammenhang verdeutlicht. Um die Texturen mit den korrekten Texturkoordinatenpaaren zu füllen wird die Liste der Tetraeder `m_listTetrahedra` durchlaufen und für jeden Tetraeder ein Array seiner vier Faces `CTetraFace *pFaces[4]` erstellt. Dabei ist `pFaces[i]` dasjenige Dreieck, welches den Vertex v_i *nicht* enthält.

```
const int *pNeighbors0 = pFaces[0]->neighbors();
int neighborIndex0 = (pNeighbors0[0] == currentTetraIndex)?
                      pNeighbors0[1]:
                      pNeighbors0[0];
Index_To_TexCoord(neighborIndex0+1, m_nTexWidth, m_nTexHeight, nTet0);
m_pTextureDataNB01[p+0] = nTet0[0];
m_pTextureDataNB01[p+1] = nTet0[1];
```

Listing 4: Ermittlung der Texturkoordinaten der Nachbarn an Face0

Für jedes der vier Dreiecke des Tetraeders wird daraufhin erfragt, welche beiden Nachbarn sich anschmiegen. Einer der Nachbarn ist immer der untersuchte Tetraeder selbst. Für die Strahlverfolgung ist natürlich der andere Nachbar, der potenzielle Folgetetraeder, wichtig. Im Code in Listing 4 wird dies exemplarisch für `pFaces[0]` gezeigt. Tritt der Fall ein, dass ein Tetraeder an einem oder mehreren Faces keinen Nachbarn hat (indiziert durch den Nachbarindex -1), werden an der entsprechenden Stelle in der Textur die Texturkoordinaten für diese Nachbartetraeder auf $(0, 0)$ gesetzt. Die Stelle 0 in der Tetraedertextur markiert einen "nicht existenten Tetraeder". Zusammenfassend wurden folgende Daten als Texturen in den Grafikspeicher geladen:

1. Eine RGB-float Textur (`VERTEXTEXTURE`), die die *xyz*-Koordinaten der Vertices enthält.
2. Eine RGB-float Textur (`TEXCOORDTEXTURE`), die die *uvw*-Koordinaten der Vertices enthält, die sich auf den Volumendatensatz beziehen.
3. Zwei ("übereinanderliegende") `RGBA-float` Texturen (`VERTEX_INDEX_TEXTURE0` und `VERTEX_INDEX_TEXTURE1`), die in einem Texel für jeden Tetraeder die Texturkoordinaten für zwei seiner Vertices enthalten. Die Texturkoordinaten beziehen sich auf die Vertextextur.
4. Zwei ("übereinanderliegende") `RGBA-float` Texturen (`NEIGHBOR_INDEX_TEXTURE0` und `NEIGHBOR_INDEX_TEXTURE1`), die in einem Texel für jeden Tetraeder die Texturkoordinaten für zwei seiner Nachbartetraeder enthalten. Diese Texturkoordinaten beziehen sich auf die Tetraedertextur.

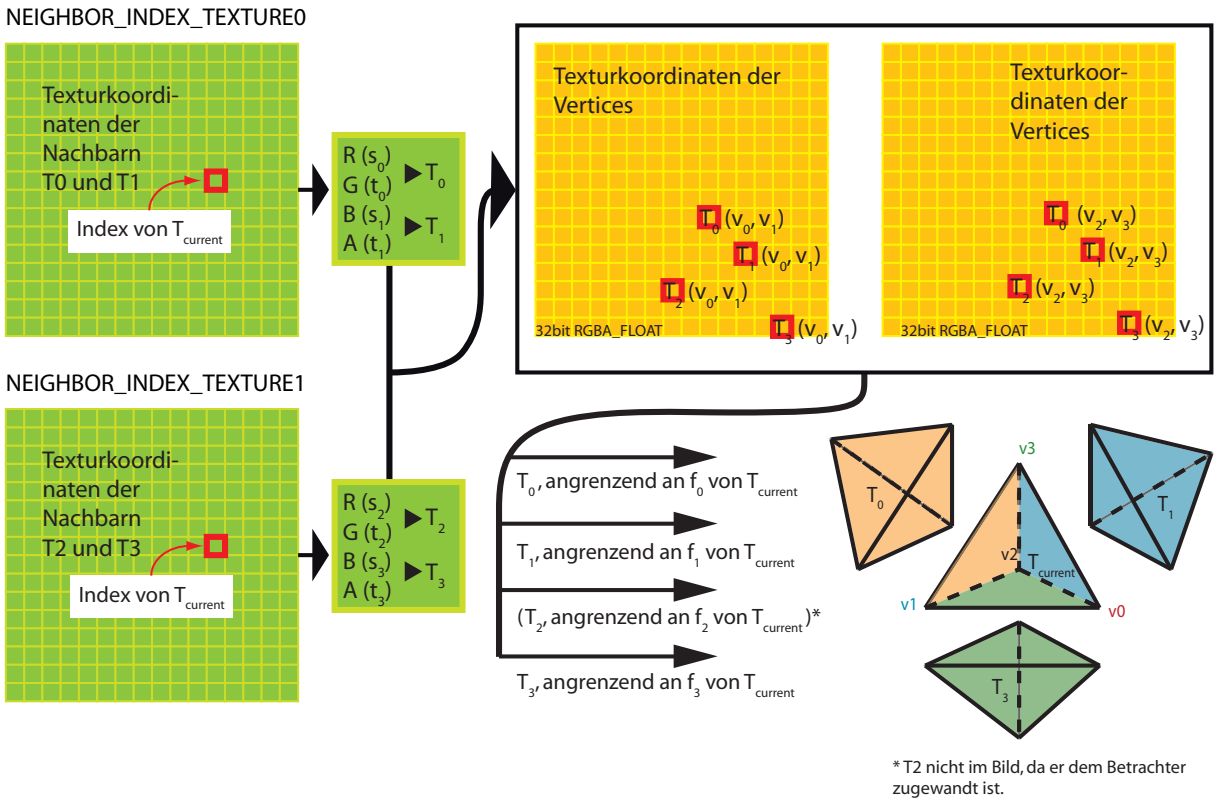

Abbildung 15: Lookup der Nachbartetraeder von $T_{current}$

4.3 Raycasting

Der Raycasting-Schritt umfasst folgende Teilschritte: Zunächst müssen die Eintrittspunkte der Sehstrahlen in die Geometrie bestimmt werden (4.3.1). Sind sie bekannt, kann der Sehstrahl durch das Tetraedernetz verfolgt werden und somit der eigentliche Raycasting-Algorithmus ablaufen (4.3.2). Durch Sampling entlang des Strahls und Front-to-Back-Compositing wird der Beitrag des Volumens für den aktuellen Pixel berechnet (4.3.3). Eine Optimierung erfährt der Algorithmus durch Early Ray Termination (4.3.5). Hierbei wird die Berechnung des Farbwerts abgebrochen, sobald beim Compositing ein vorgegebener Opazitätswert erreicht wurde. Die Probleme, die beim Raycasting mit nicht-konvexen Gittern auftreten, werden mit Hilfe eines Depth Peeling-Schrittes gelöst (4.3.6).

4.3.1 Eintrittspunkte

Sind alle nötigen Datenstrukturen erzeugt, werden zur Bestimmung der Startpunkte für das Raycasting die Boundary Faces rasterisiert. Hierzu wurde die Liste aller Randflächen erstellt (4.2.2). Dem Vertex Shader werden jeweils die drei Vertices eines Boundary Face und der Tetraederindex, zu dem das Boundary Face gehört, als Vertex Buffer übergeben. Die Funktion `createVertexBuffers` erstellt den Buffer. In diesem Abschnitt wird der Einfachheit halber noch davon ausgegangen, dass die Geometrie nicht deformiert werden soll. Die Deformation und die Konsequenzen, die sich daraus ergeben, werden in Kapitel 4.4 behandelt. Für ein Tetraedernetz wird ein Vertexbuffer mit der in Abbildung 16 veranschaulichten Struktur erstellt und an das Vertexprogramm übergeben.

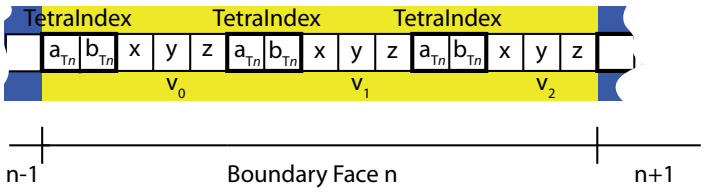

Abbildung 16: Vertexbuffer für eine statische Gridstruktur

Die Boundary Faces werden in der Funktion `drawBoundaryFaces` durch den Aufruf von `glDrawBuffers` gezeichnet.

Eingabeparameter für das Vertexprogramm sind also der aktuelle Vertex `float3 VertexInA` und die Texturkoordinaten `float2 TetraIndex` für den Tetraeder, zu dem der Vertex gehört. Das Vertexprogramm hat lediglich die Aufgabe, die Vertices in den Screenspace zu transformieren. Die Eintrittsposition `PositionIn` und die korrespondierenden Tetraeder-Texturkoordinaten `TetraIndex` werden dann dem Fragmentprogramm für das Raycasting übergegeben. Die Theorie der Strahlverfolgung wurde schon in Kapitel 3.3.1 erläutert. Im folgenden Kapitel wird die Implementierung der Strahlverfolgung beschrieben.

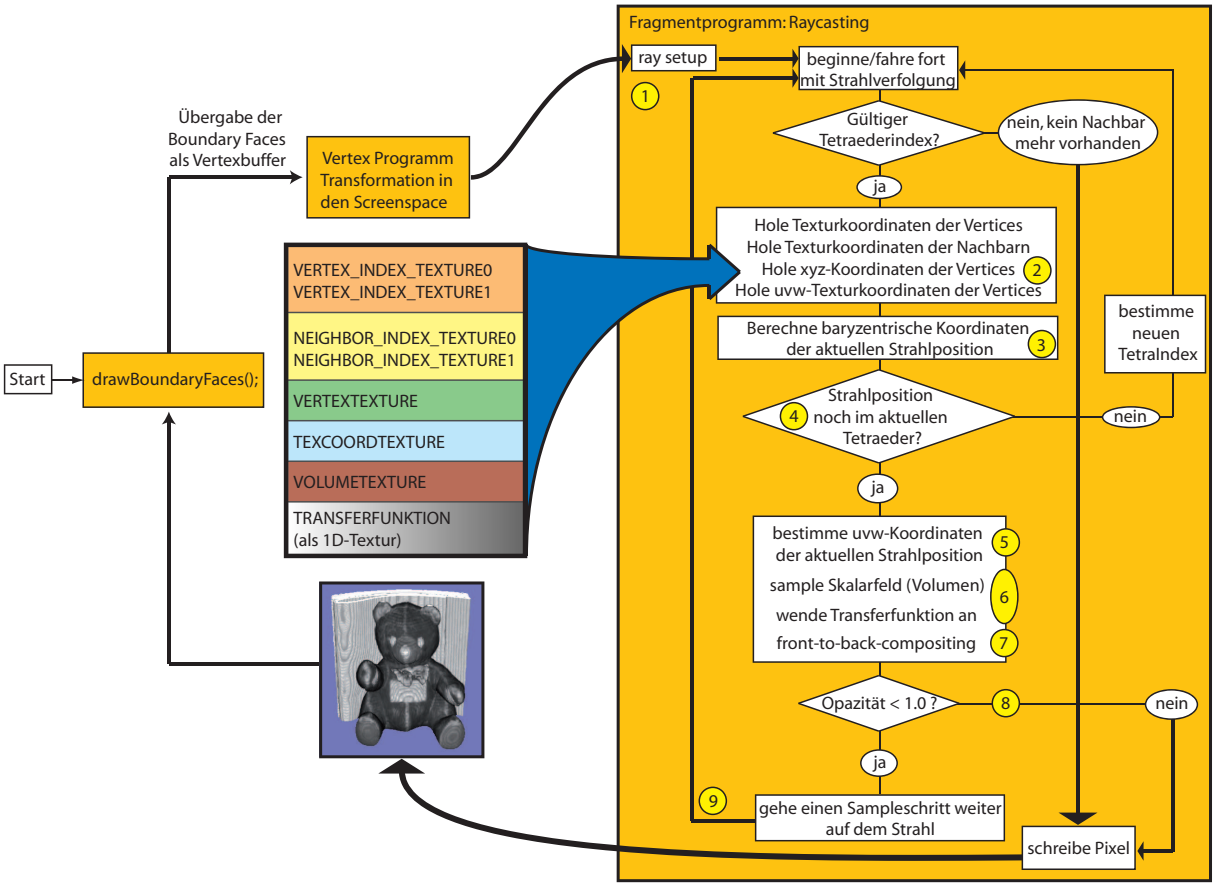

Abbildung 17: Ablauf des Raycasting

4.3.2 Strahlverfolgung

Dem Fragmentprogramm für das Raycasting stehen zu Beginn die Eintrittsposition des Sehstrahls in die Geometrie (`PositionIn`) und der Index des Tetraeders, auf dem der Eintrittspunkt liegt (`TetraIndex`), zur Verfügung. Zusätzlich dazu werden dem Fragmentprogramm die notwendigen Texturen übergeben: Die Vertextextur, die Texturkoordinatentextur, die Tetraedertextur(en), die Nachbarschaftstextur(en) und natürlich den darzustellenden Volumendatensatz als 3D-Textur `VolumeTexture`. Der Ablauf im Fragmentprogramm für das Raycasting ist in Abbildung 17 veranschaulicht. Die eingekreisten Ziffern im Text beziehen sich auf die gelb unterlegten Zahlen in dieser Abbildung.

Ray Setup ①

Zunächst kann anhand von Augpunkt und Eintrittspunkt die Richtung `rayDir` des Sehstrahls berechnet werden und so durch den Code in Listing 5 die Geradengleichung für den Sehstrahl aufgestellt werden. STEPSIZE ist die Schrittweite, um die die Strahlposition nach jedem Samp-

```
float3 rayPosition = PositionIn + STEPSIZE * rayDir;
```

Listing 5: Die Geradengleichung des Sehstrahls

lingschritt fortschreitet. Nach dieser Initialisierung des Sehstrahls kann mit der Strahlverfolgung begonnen werden.

Texture lookups ②

Über den `TetraIndex` können von `VERTEX_INDEX_TEXTURE0` und `VERTEX_INDEX_TEXTURE1` die Texturkoordinaten `s`,`t` für die vier Vertices des aktuellen Tetraeders erfragt werden. Im Anschluss daran kann `TetraIndex` für einen Lookup in den beiden Texturen `NEIGHBOR_INDEX_TEXTURE0` und `NEIGHBOR_INDEX_TEXTURE1` benutzt werden um die Texturkoordinatenpaare der vier möglichen Nachbartetraeder herauszufinden. Über die soeben abgefragten Texturkoordinaten für v_0 bis v_3 können aus `DataVertices` dann die *xyz*-Koordinaten der Vertices abgefragt werden, und aus `TEXCOORDTEXTURE` die *uvw*-Koordinaten für v_0 bis v_3. Auf diese Weise sind alle nötigen Informationen zusammengetragen worden, die für die baryzentrische Interpolation im Tetraeder notwendig sind.

Baryzentrische Interpolation ③

Die Interpolation muss durchgeführt werden, um an der korrekten Stelle in der `VolumeTexture` zu samplen. Für die Interpolation werden zunächst die Abstandsvektoren vec_0 bis vec_3 des aktuell untersuchten Punktes auf dem Strahl zu den Vertices des Tetraeders berechnet. Über das Spatprodukt können die Volumen der vier Tetraeder berechnet werden, die entstehen, wenn man die vier Eckpunkte des untersuchten Tetraeders mit der aktuellen `rayPosition` verbindet. Die Summe aller Volumen ergibt das Gesamtvolumen `area` des Tetraeders.

```
float area0 = dot(cross(vec2,vec3),vec1);
float area1 = dot(cross(vec3,vec2),vec0);
float area2 = dot(cross(vec0,vec1),vec3);
float area3 = dot(cross(vec1,vec0),vec2);
float area  = area0 + area1 + area2 + area3;
```

Listing 6: Berechnung der baryzentrischen Koordinaten im Tetraeder

4.3.3 Sampling und Compositing

Gesucht sind nun die Texturkoordinaten `rayTexCoord` bezüglich der Volume Texture für den aktuell untersuchten Punkt auf dem Sehstrahl. Sie ergeben sich für den aktuellen Tetraeder aus dem Code in Listing 7 (⑤).

```
float3 rayTexCoord  = uvw0 * area0 + uvw1 * area1 + uvw2 * area2 + uvw3 * area3;
       rayTexCoord /= area;
```

Listing 7: Berechnung der 3D-Texturkoordinaten in `VolumeTexture` für den aktuellen Samplingpunkt

Somit kann der Volumendatensatz an der entsprechenden Stelle abgetastet, der Skalarwert durch trilineare Interpolation ermittelt und ihm durch eine Transferfunktion ein entsprechender Farbwert zugewiesen werden (Listing 8, ⑥).

```
float  scalar      = tex3D(VolumeTexture, rayTexCoord).r ;
float4 sample      = tex1D(TransFunc, scalar);
       sample.rgb *= sample.a;
```

Listing 8: Sampling und Anwendung einer Transferfunktion

Der entgültige Farbwert `result` für das untersuchte Pixel, der vorher mit `float4 result = float4(0.0f,0.0f,0.0f,0.0f)` initialisiert wurde, ergibt sich durch Front-to-Back-Compositing über alle Samplingpunkte des Sehstrahls (Listing 9, ⑦).

```
result += (1.0 - result.a) * sample;
```

Listing 9: Front-to-Back-Compositing

Im Anschluss an die Berechnung des aktuellen Farbwerts für das untersuchte Pixel wird auf dem Sehstrahl um die Schrittweite `STEPSIZE` vorangeschritten (Listing 10, ⑨).

```
rayPosition += STEPSIZE * rayDir;
```

Listing 10: Voranschreiten entlang des Sehstrahls

4.3.4 Bestimmung des Folgetetraeders

Bisher unbeantwortet geblieben sind die Fragen, wie festgestellt werden kann, wann die aktuelle Position auf dem Sehstrahl außerhalb des aktuellen Tetraeders liegt und wie dann der Folgetetraeder bestimmt wird (④).
Wie im theoretischen Kapitel schon erläutert wurde gibt es zwei Möglichkeiten, das Problem zu handhaben: 1. Die *exakte Bestimmung von Ein- und Austrittspunkt* und 2. Die Bestimmung des Folgetetraeders durch *Beobachtung der baryzentrischen Koordinaten.*
Den Ein- und Austrittspunkt exakt zu bestimmen ist möglich, allerdings ergeben sich erhebliche numerische Probleme, die zu Artefakten im gerenderten Bild führen. Der Fehler passiert bei der Schnittpunktberechnung zwischen Sehstrahl und den vier Ebenen, in denen die Dreiecksflächen liegen. In Kapitel 3.3.1 wurde die Bestimmung des Schnittpunktes erläutert: Bei der Schnittpunktberechnung zwischen Sehstrahl und den vier Ebenen des Tetraeders ergeben sich vier mögliche Werte für den potenziellen Austrittspunkt. Von den vier möglichen Lambda-Werten muss der kleinste, der größer ist als 0, in die Geradengleichung eingesetzt werden um den Austrittspunkt zu bestimmen. Die numerischen Probleme entstehen dadurch, dass durch die begrenzte floating point-Genauigkeit keiner der Werte genau 0 ist und deshalb ein Test auf > 0 von mehr Werten bestanden wird als dies der Fall sein dürfte. Somit werden falsche Austrittspunkte berechnet.

Viel einfacher und eleganter ist die Lösung, den Austrittspunkt "nebenbei" zu detektieren: Da für die Feststellung der 3D-Texturkoordinaten der Strahlposition ohnehin in jedem Schritt die baryzentrischen Koordinaten im Tetraeder berechnet werden müssen genügt es, diese vier Volumen jeweils auf einen negativen Betrag zu untersuchen. So kann nicht nur herausgefunden werden, *wann* der Sehstrahl den Tetraeder verlassen hat, sondern auch durch welches Austrittsdreieck, denn durch den Code in Listing 4.3.2 wurde festgelegt, dass $area_n$ der Teiltetraeder ist, der als Grundfläche die Tetraederfläche f_n hat.

Wenn also das Spatprodukt $area_n$ negativ wird, dann muss der Sehstrahl den Tetraeder über die Fläche f_n verlassen haben. Es kann dann der Index des Nachbartetraeders bestimmt werden. Dem Fragmentprogramm wurden zu Beginn die Nachbartexturen übergeben und die Texturkoordinaten der vier möglichen Nachbartetraeder wurden erfragt:

```
//Texturkoordinaten der Nachbarn 0 und 1
float4 texcoord_NB01 = tex2D(DataNB01,TetraIndex);
//Texturkoordinaten der Nachbarn 2 und 3
float4 texcoord_NB23 = tex2D(DataNB23,TetraIndex);
```

Listing 11: Erfragen der Texturkoordinaten der Nachbartetraeder

Es müssen also lediglich die baryzentrischen Gewichte überprüft werden:

```
if (area0 < 0.0) {
TetraIndex = texcoord_NB01.rg; // Wähle Nachbarn an der Fläche 0
} else if (area1 < 0.0) {
TetraIndex = texcoord_NB01.ba; // Wähle Nachbarn an der Fläche 1
} else if (area2 < 0.0) {
TetraIndex = texcoord_NB23.rg; // Wähle Nachbarn an der Fläche 2
} else if (area3 < 0.0) {
TetraIndex = texcoord_NB23.ba; // Wähle Nachbarn an der Fläche 3
} else
//...dann befindet sich die Samplingposition noch
//im aktuellen Tetraeder.
```

Listing 12: Wahl des richtigen Folgetetraeders

Dies setzt sich so lange fort bis der Strahl aus dem Volumen austritt, d.h. bis festgestellt wird, dass kein Folgetetraeder mehr existiert (Listing 13). Es muss allerdings nicht jeder Sehstrahl

```
// Falls es keinen gültigen Tetraeder mehr gibt: Abbruch, Pixelfarbe schreiben.
if ((TetraIndex.x == 0.0) && (TetraIndex.y == 0.0)) {
   return result;
}
```

Listing 13: Prüfung auf Austritt aus dem Volumen

bis zum endgültigen Austritt aus der Geometrie abgetastet werden. Eine einfache Optimierung kann durch Early Ray Termination erreicht werden.

4.3.5 Early Ray Termination

Beim Front-To-Back-Compositing ist es möglich, den Sehstrahl nur so lange zu verfolgen, bis die akkumulierte Opazität des `result`-Werts bei annähernd 1.0 liegt. Jeder Beitrag des Volumens, der danach noch berechnet wird, wäre ohnehin nicht sichtbar. Deshalb wird die Strahlverfolgung sofort abgebrochen, wenn der Opazitätswert über $1 - \varepsilon$ liegt (⑧, Listing 14).

```
//Front to Back Compositing
result += (1.0 - result.a) * sample;
//Early Ray Termination
if (result.a > 0.9999)
{
    break;
}
```

Listing 14: Early Ray Termination

4.3.6 Depth Peeling

Der bisher beschriebene Ablauf des Raycasting funktioniert problemlos, solange die Geometrie konvex ist, d.h. solange sie keine Löcher oder Hohlräume aufweist, die zur Folge haben, dass ein

Sehstrahl mehrfach in die Geometrie eindringt und wieder austritt. Ein konvexes Gitter zeichnet sich dadurch aus, dass alle Front Faces sichtbar sind. Bei einem nichtkonvexen Gitter liegt aber eine Teilmenge der Front Faces hinter anderen Front Faces. Bei der bisherigen Implementierung würde das Sampling entlang des Sehstrahls nach dem ersten Austreten aus der Geometrie abgebrochen werden. Dies würde dazu führen, dass Bereiche, die vom Betrachter aus hinter Hohlräumen liegen, nicht dargestellt werden. Die Problematik wurde in der Theorie in Kapitel 3.3.3 behandelt. Um auch nichtkonvexe Tetraedergitter vollständig darstellen zu können, wird die Implementierung durch ein weiteres Fragmentprogramm erweitert, in dem ein zusätzlicher Depth Peeling-Schritt stattfindet.

Die Idee des Depth Peeling besteht darin, ein nichtkonvexes Gitter solange "schichtweise" zu rendern bis es keine verdeckten Front Faces mehr gibt, die gezeichnet werden müssen. In den Abbildungen 18 und 19 ist dies anhand einer Geometrie veranschaulicht, bei der zwei Schichten gerendert werden müssen.

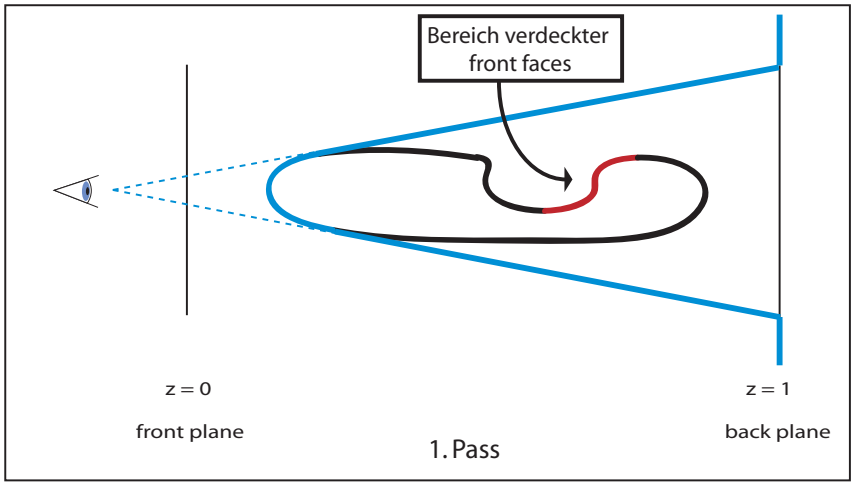

Abbildung 18: Erster Render Pass.

Um die unterschiedlichen Layers zu rendern, wird der bisherige Renderingalgorithmus zu einem Multipass-Rendering erweitert. In jedem Durchlauf werden zwar die gesamten Front Faces der Geometrie gerendert, jedoch nur eine Teilmenge davon zum Raycastingschritt weitergeleitet. Es werden nur diejenigen Front Faces (in einen off-screen buffer) gezeichnet, die sich hinter dem

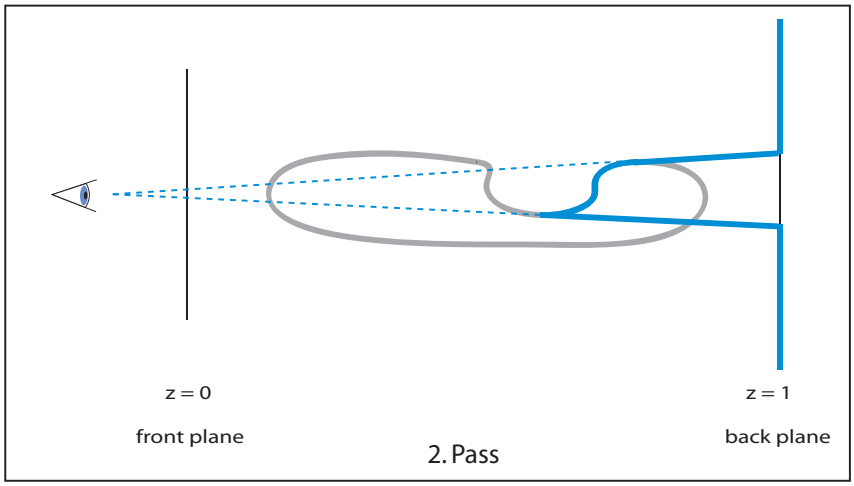

Abbildung 19: Zweiter Render Pass.

Tiefenprofil befinden, welches im vorigen Schritt erstellt wurde. Um im nächsten Schritt auf die Tiefeninformationen des vorigen Schrittes zugreifen zu können, werden zwei Frame Buffer Objects angelegt und diese abwechselnd verwendet ("Ping-Pong"-Schema). In jedem Render Pass wird der Depth Buffer eines der beiden geschrieben, der des anderen gelesen. Danach wird getauscht, so dass in Pass $n + 1$ der Depth Buffer des Frame Buffer Objects gelesen werden kann, der in Pass n geschrieben wurde. In jedem Durchgang werden die Fragmente des gerade gerenderten Layers gezählt. Sinkt die Anzahl der gezeichneten Fragmente unter einen anzugebenden Grenzwert, kann davon ausgegangen werden, dass alle zu zeichnenden Layers gezeichnet wurden. Der Multipass-Renderingschritt ist damit abgeschlossen.
Der Ablauf des kompletten Renderings mit integriertem Depth Peeling-Schritt ist in Abbildung 20 dargestellt.

Die Integration des Depth Peeling in den bestehenden Algorithmus und die Implementierung als Fragmentprogramm (siehe Listing 22 im Anhang) sollen nun noch genauer betrachtet werden. Die eingekreisten Zahlen im folgenden Text beziehen sich auf Abbildung 20.

Zu Beginn der `draw`-Routine wird einer der FBOs als zu lesender (read-FBO), der andere als zu schreibender Buffer (write-FBO) deklariert (①). Für den ersten Render Pass wird die Depth Texture des read-FBO mit Nullen gefüllt (②). Weil im ersten Render Pass alle Fragmente einen

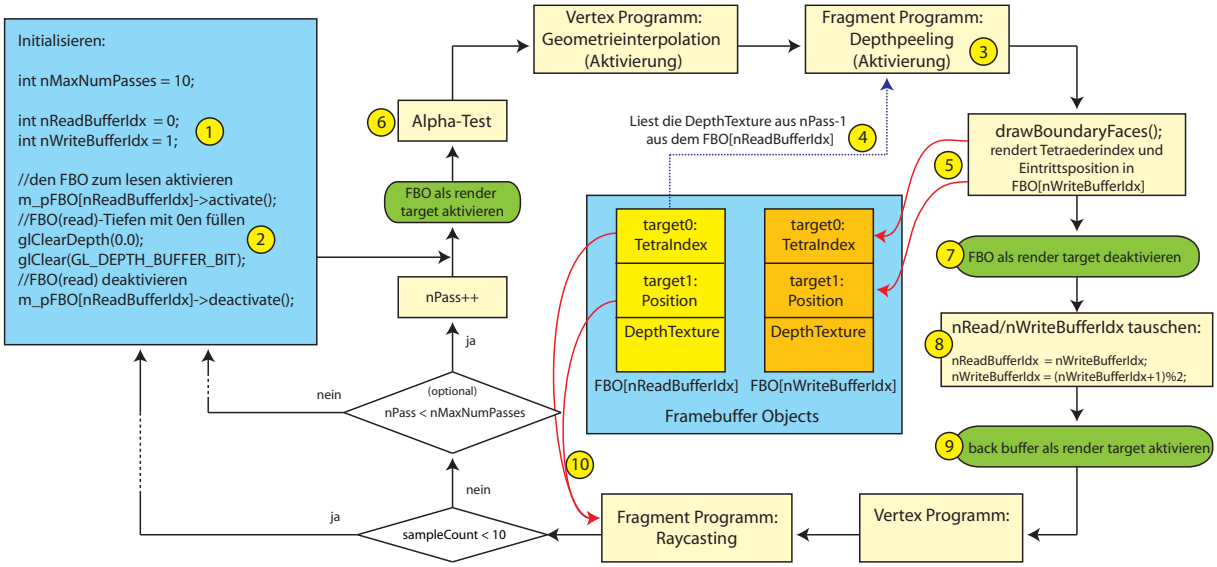

Abbildung 20: Ablauf des Renderings mit integriertem *Depth Peeling*

Tiefenwert größer als 0 haben, werden alle Front Faces den ersten Tiefenvergleich bestehen. In jedem Render Pass sollen nur die Fragmente gezeichnet werden, deren Tiefenwerte größer sind als die, die im vorangegangenen Schritt in den Depth Buffer geschrieben wurden. Der Depth Test sorgt in jedem Schritt dafür, dass die Fragmente, die auf verdeckten Front Faces liegen, verworfen werden. Im ersten Render Pass werden also alle Front Faces gezeichnet, die unmittelbar sichtbar sind. Ziel ist es nun, im nächsten Render Pass nur die Front Faces zu rendern, die auf dem zweiten Depth Layer liegen, im dritten Pass die Front Faces auf der dritten Schicht usw. Dafür sorgt das Fragmentprogramm für das Depth Peeling (③).

Dem Fragmentprogramm für das Depth Peeling stehen zu Beginn vier Informationen zu Verfügung: Die Eintrittsposition des Sehstrahls `PositionIn`, der zugehörige Tetraederindex `TetraIndex`, die Vertexposition im Screenspace `VPosUnitCube` und die Tiefentextur `DepthTexture` des read-FBO, welche die Tiefenwerte aus dem ersten bzw. dem vorherigen Render Pass enthält.

Zunächst wird für das untersuchte Fragment der Tiefenwert aus der Depth Texture des read-FBO ausgelesen (Listing 15, ④). Das Raycasting für diese Schicht, welches danach ausgeführt werden soll, benötigt für das untersuchte Pixel die Eintrittsposition und den Tetraederindex. Dazu wer-

```
float2 uv    = VPosUnitCube.xy;
// "bisherigen" Wert des Depth-Buffers lesen:
float  depth = tex2D(DepthTexture,uv).x;
```

Listing 15: Auslesen des Tiefenwerts aus dem vorangegangenen Render Pass

den zwei zusätzliche Render Targets des FBOs benutzt, sodass nicht nur die Tiefenwerte, sondern auch die Eintrittsposition (`value1`) und der entsprechende Tetraederindex (`value0`) für dieses Pixel in den write-FBO geschrieben werden. Diese beiden Werte müssen nun ermittelt werden. Um Fragmente zu verwerfen, die nicht mehr gezeichnet werden müssen, wird standardmäßig `value1`, also der Eintrittspunkt, mit einem Alphawert $a = 0$ initialisiert. Im nächsten Schritt würde dieses Fragment somit den Alpha-Test (⑥) nicht bestehen und verworfen werden. Auch `value0`, der Tetraederindex, wird mit 0 initialisiert. Es wird zunächst also davon ausgegangen, dass für das aktuell untersuchte Fragment kein weiterer Schnittpunkt mit der Geometrie existiert (Listing 16). Nun muss getestet werden, ob es nicht doch einen (weiteren) Schnittpunkt

```
//Zugehöriger Tetraederindex (default: 0, kein Schnittpunkt mit Tetraeder)
value0 = float4(0.0,0.0,0.0,0.0);
//Eintrittspunkt in die Geometrie (default: kein Eintrittspunkt)
value1 = float4(0.0,0.0,0.0,0.0);
```

Listing 16: Standardwerte der Fragmente vor dem Tiefentest. In dieser Form würde das Fragment im nächsten Render Pass den Alpha-Test nicht bestehen.

mit einem Front Face gibt: Wenn der aktuell ausgelesene Tiefenwert `VPosUnitCube.z` größer ist als der Tiefenwert für das Fragment aus dem vorherigen Schritt `depth` (Listing 17), dann bedeutet dies, dass der Sehstrahl ein weiteres Front Face geschnitten hat. Ist dies der Fall, müssen die zuvor gesetzten default-Werte für das Fragment überschrieben werden. `value0` erhält die Texturkoordinaten des Tetraeders, der geschnitten wird. In `value1` wird der Eintrittspunkt des Sehstrahls in die Geometrie als *rgba* geschrieben, wobei durch den Alphawert $a = 1$ sichergestellt wird, dass das Fragment im nächsten Pass den Alpha-Test besteht und nicht ignoriert wird. Die

beiden Werte, ob mit den default-Werten oder tatsächlichen Werten für Schnittpunkte belegt, werden in die beiden Render Targets geschrieben (⑤).

```
// nur das zeichnen,
// was hinter dem bisherigen Depth-buffer Wert liegt:
if (depth < VPosUnitCube.z)
{
   // hier ist A=1, der Alpha-Test schreibt dieses Fragment
   value0 = float4(TetraIndex.xy,0.0, 1.0); // target0 = TetraIndex
   value1 = float4(PositionIn,        1.0); // target1 = Position
}
```

Listing 17: Vergleich zwischen dem aktuellen und dem vorherigen z-Wert des Pixels

Nach dem Depth Peeling-Schritt werden die FBOs als Render Targets deaktiviert (⑦), die Indizes für read- und write-FBO getauscht (⑧) und wieder in den Back Buffer gerendert (⑨). Was folgt ist der schon beschriebene Raycasting-Schritt. Dem Fragmentprogramm für das Raycasting stehen jetzt durch Zugriff auf den read-FBO die soeben geschriebenen Werte `value0` und `value1` als Texturen `TetraIndexTex` und `PositionTex` zur Verfügung (⑩). In `PositionTex` wurde für jedes Pixel der erste Schnittpunkt mit der Geometrie gespeichert, der im Unterschied zum Raycasting ohne Depth Peeling nun auch auf einem ursprünglich verdeckten Front Face liegen kann. In `TetraIndexTex` stehen die Texturkoordinaten für den Tetraeder, der vom Sehstrahl getroffen wird. Dies ist alles, was das Raycasting-Fragmentprogramm benötigt, um die aktuelle Schicht, wie in den Abschnitten 4.3.2 bis 4.3.5 beschrieben, zu rendern. Abgebrochen wird das Multipass-Rendering, wenn die Anzahl der gezeichneten Fragmente unter eine bestimmte Grenze fällt oder (optional) eine angegebene Maximalzahl von Layers gerendert wurde.

4.4 Deformation

Das Prinzip der Deformation wurde schon im Kapitel 3.4 beschrieben: Die Vertices des Tetraedernetzes werden bei gleichzeitiger Beibehaltung der Texturkoordinaten in die VolumeTexture transformiert. Die Transformation der Vertices erfolgt in dieser Arbeit über die Definition mehrerer Vertexkonfigurationen. Jede dieser Konfigurationen beschreibt einen Keyframe, und zwischen

den Keyframes wird linear interpoliert. Diese sogenannten Time Nodes können aus der XML-Beschreibung ausgelesen werden. Listing 18 zeigt eine einfache Gruppe von vier Vertices, für die zu unterschiedlichen Zeitpunkten zwei Vertexkonfigurationen festgelegt sind.

```
<Nodes time="0.0">
    <Vertex> -1.00000 -1.00000 -1.00000 </Vertex>
    <Vertex>  1.00000 -1.00000 -1.00000 </Vertex>
    <Vertex>  1.00000  1.00000 -1.00000 </Vertex>
    <Vertex> -1.00000  1.00000 -1.00000 </Vertex>
</Nodes>
<Nodes time="1.0">
    <Vertex> -2.00000 -2.00000 -2.00000 </Vertex>
    <Vertex>  2.00000 -2.00000 -2.00000 </Vertex>
    <Vertex>  2.00000  2.00000 -2.00000 </Vertex>
    <Vertex> -2.00000  2.00000 -2.00000 </Vertex>
</Nodes>
```

Listing 18: Eine Vertexgruppe, zwei Vertexkonfigurationen

Damit der Volumendatensatz anhand einer solchen Beschreibung deformiert werden kann, muss die bisherige Implementierung erweitert werden:

1. Wenn die Vertices der Boundary Faces als Vertexbuffer zur Grafikkarte gesendet werden, müssen nun mehrere Vertexkonfigurationen als Vertexbuffer gespeichert werden. In Abbildung 16 in Kapitel 4.3.1 war die Struktur des Vertexbuffers für ein Grid ohne Deformation dargestellt. Für ein animierbares Grid werden jeweils korrespondierende Vertices hintereinander gespeichert. In Abbildung 21 ist dieser Fall für zwei Vertexkonfigurationen veranschaulicht. Die Vertexkoordinaten werden jeweils für beide Zeitschritte (t = 0 und t = 1) gespeichert. Das Vertexpro-

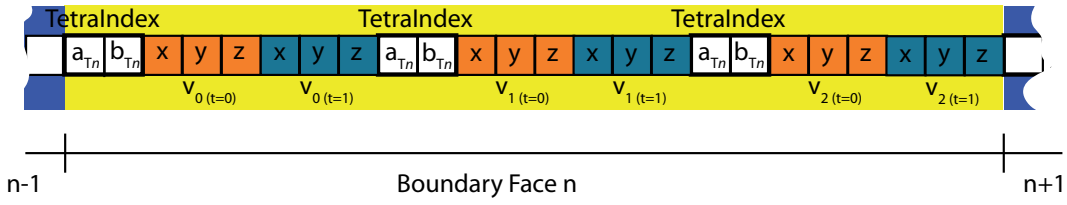

Abbildung 21: Vertexbuffer für eine Gridstruktur mit Animation

gramm `tetra_vert.cg` (Listing 19 erhält also zusätzlich die korrespondierenden Vertexkoordinaten `float3 VertexInB` des nächsten Zeitschritts. Zusätzlich dazu wird der aktuelle Zeitwert als Interpolationsgewicht `float time` als `uniform`-Parameter übergeben, der in jedem Animationsschritt neu berechnet wird. `tetra_vert.cg` hat also zusätzlich zur Screenspace-Transformation der Vertices nun noch die Aufgabe, zwischen den Zeitschritten zu interpolieren.

```
void main(float3 VertexInA    : POSITION,  //Vertexkoordinaten
          float2 TetraIndex   : TEXCOORD0, //Tetraederindex
          float3 VertexInB    : TEXCOORD1, //Vertexkoordinaten nächster Timenode
          uniform float4x4 ModelViewProj,
          uniform float4x4 ModelViewI,
          uniform float4x4 ModelViewIT,
          uniform float time,              //Lokaler Interpolationsschritt

          out float4 VertexOut     : POSITION,
          out float3 PositionOut   : TEXCOORD1,
          out float2 TetraIndexOut : TEXCOORD2,
          out float3 VPosUnitCube  : TEXCOORD5
          )
{
    // Vertex "entlang der Zeit" interpolieren
    // (bei statischen Netzen ist time = 0 und VertexInB irrelevant)
    float3 VertexIn = lerp(VertexInA, VertexInB, time);
    //Transformation in den screenspace.
    VertexOut     = mul(ModelViewProj, float4(VertexIn,1.0));
    VPosUnitCube  = (VertexOut.xyz/VertexOut.w +1.0)/2.0;
    PositionOut   = VertexIn;
    TetraIndexOut = TetraIndex;
}
```

Listing 19: Vollständiger Vertexshadercode für die Animation

2. Die zweite Änderung besteht darin, dass in jedem Zeitschritt die `VertexTexture` aktualisiert und auf die Grafikkarte geladen werden muss, damit im Fragment Shader für das Raycasting immer die aktuellen Vertexkoordinaten zu Verfügung stehen. Dies leistet die Funktion `void CTetraGrid::animate()`. Sie interpoliert linear zwischen den korrespondierenden Koordinaten zweier Vertexkonfigurationen (Listing 20) und veranlasst, dass die VertexTexture neu generiert

wird. Die Interpolation der Vertex Texture in Software bringt allerdings eine hohe Buslast mit

```
void CTetraGrid::animate()
{
    float u,v,w;
    CVector vSRC;
    CVector vDEST;
    CVector vDIR;
    CVector vSTEP;
    for(int k = 0; k < m_nNumVerticesSRC; k++)
    {
        vSRC = m_VerticesTimeNodeSRC.at(k);
        vDEST = m_VerticesTimeNodeDEST.at(k);
        vDIR = vDEST - vSRC;
        vSTEP = vSRC + m_fLocalStep * vDIR;
        m_pTextureHandler->getTexCoord(k,u,v,w);
        vSTEP.setTexCoord(u,v,w);
        m_listVertices.replace(k,vSTEP);
    }
    m_pTextureHandler->generateVertexTexture(m_nNumVertices);
}
```

Listing 20: Interpolation zwischen den Vertices zweier Zeitschritte

sich, weil in jedem Zeitschritt die Textur neu generiert werden muss. Eine weitere Möglichkeit besteht darin, die Time Node-Vertexkonfigurationen ebenfalls als Texturen in den Grafikspeicher zu laden und dann im Fragment Shader linear zwischen den Texturen zu interpolieren *(Blend Shapes)*.

4.5 Zusammenfassung

Aus der gegebenen Grid-Beschreibung wurden zu Beginn wichtige Informationen abgeleitet und als Texturen im Speicher der Grafikkarte abgelegt. Aus diesen Texturen können nicht nur die Vertexkoordinaten, die Zusammensetzung von Vertices zu Tetraedern, sondern auch die Nachbarschaftsbeziehungen zwischen den Tetraedern abgelesen werden. Diese Daten werden initial berechnet und dann einmalig über den Bus geschickt. Die Eintrittspunkte für das Raycasting ergeben sich automatisch aus der Rasterisierung der Boundary Faces. Für jeden Sehstrahl wird der

Beitrag berechnet, den das Volumen für dieses Pixel leistet. Im Raycasting-Fragment Programm werden durch baryzentrische Interpolation die Texturkoordinaten der aktuellen Strahlposition berechnet, um damit das 3D-Skalarfeld an entsprechender Stelle abzutasten. Verlässt der Sehstrahl einen Tetraeder, wird anhand der in den Texturen gespeicherten Nachbarschaftsbeziehungen der passende Nachbar ermittelt um die Texturkoordinaten weiterhin korrekt interpolieren zu können. Die Farb- und Opazitätswerte entlang des Sehstrahls werden durch Front-to-Back-Compositing zu einer resultierenden Farbe für das Pixel aufgerechnet. Durch Early Ray Termination kann die Berechnung vorzeitig abgebrochen werden, sobald die kumulierte Opazität nahezu bei 1.0 liegt. Um auch nichtkonvexe Gitter korrekt darstellen zu können, werden zwei Frame Buffer Objects benutzt und der Algorithmus zum Multipass-Rendering erweitert. Die Depth Textures der beiden FBOs dienen dazu, sich jeweils die Tiefeninformationen aus dem vorangegangenen Render Pass merken zu können, um nur dahinterliegende Bereiche zu zeichnen. Die Deformation des Volumens wird schließlich erreicht, indem unterschiedliche Vertexkonfigurationen für verschiedene Zeitpunkte festgelegt werden. Ein Vertexprogramm interpoliert zwischen den Listen, die die Boundary Faces zu verschiedenen Zeiten beschreiben. Die Interpolation zwischen den Konfigurationen der inneren Vertices findet noch in Software statt. Sie wäre jedoch leicht ebenfalls in einem Fragmentprogramm durch Blend Shapes zu implementieren (siehe [2]).

5 Ergebnisse

Die Implementierung unter Verwendung von C++, OpenGL und Cg wurde auf einem System mit Intel Core Duo 6700 Prozessor (2,66 GHz), 3 GB Hauptspeicher und einer Geforce 8800 GTX mit 768 MB Grafikspeicher getestet. Die Anwendung ist auch auf Grafikboards der Geforce 7-Familie lauffähig. Die Performance ist dort aber erheblich geringer, weil die vorliegende Implementierung sehr vom Unified Shader Model profitiert und dieses erst seit der Geforce 8-Serie realisiert wurde. Der Algorithmus wurde an vier Gittern mit unterschiedlicher Anzahl von Tetra-

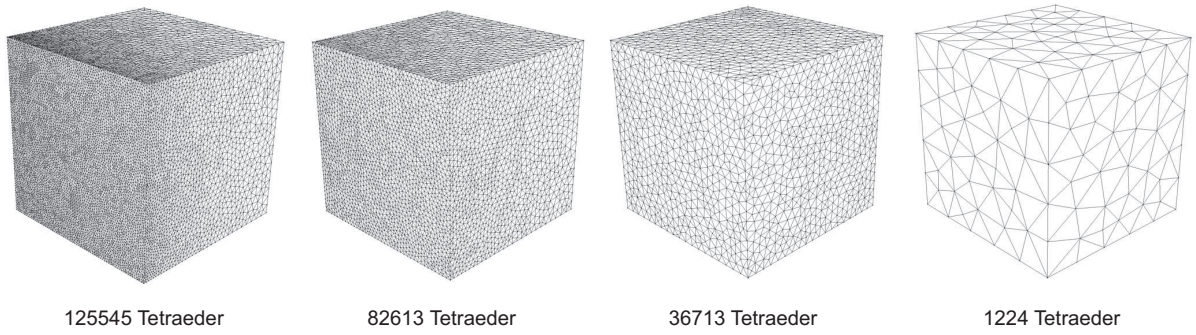

Abbildung 22: Die vier Testgitter

edern getestet (1224, 36713, 82613 und 125545 Tetraeder, siehe Abbildung 22). Drei verschiedene Volumendatensätze wurden dargestellt ($128 \times 128 \times 62$ Voxel (8 bit), $256 \times 256 \times 256$ (16 bit) und $512 \times 512 \times 512$ (16 bit) Texturen). In Tabelle 3 sind die Eigenschaften der verwendeten Gitter aufgelistet. Gemessen wurden die Frames pro Sekunde (siehe Tabelle 2), jeweils mit einer hohen und einer niedrigen Opazität. Die transparente Transferfunktion wurde so gewählt, dass fast alle Strahlen bis zum Austritt durchlaufen können. Eine solch hohe Transparenz wird allerdings in der praktischen Anwendung kaum vorkommen. Die sehr hohen Frameraten beim 8 bit ($128 \times 128 \times 62$)-Datensatz ergeben sich aus der geringen benötigten Bandbreite beim Transfer zwischen GPU und lokalem Videospeicher. Bei den beiden 16-bit Volumen verringert sich die Framerate deshalb entsprechend. Auffallend ist, dass sich die Unterschiede in der Tetraederanzahl bei der Anwendung einer opaken Transferfunktion nicht in der Framerate niederschlägt. Der Grund dafür ist in allen Fällen das frühe Abbrechen durch Early Ray Termination (ERT),

Volumengröße	**Anzahl Tetraeder**	**ERT**	**fps**
$128 \times 128 \times 62$	1224	opak	60
$128 \times 128 \times 62$	36713	opak	60
$128 \times 128 \times 62$	82613	opak	60
$128 \times 128 \times 62$	125545	opak	60
$128 \times 128 \times 62$	1224	transparent	10
$128 \times 128 \times 62$	36713	transparent	8
$128 \times 128 \times 62$	82613	transparent	7
$128 \times 128 \times 62$	125545	transparent	7
$256 \times 256 \times 256$	1224	opak	30
$256 \times 256 \times 256$	36713	opak	20
$256 \times 256 \times 256$	82613	opak	20
$256 \times 256 \times 256$	125545	opak	20
$256 \times 256 \times 256$	1224	transparent	9
$256 \times 256 \times 256$	36713	transparent	7
$256 \times 256 \times 256$	82613	transparent	6
$256 \times 256 \times 256$	125545	transparent	5
$512 \times 512 \times 512$	1224	opak	30
$512 \times 512 \times 512$	36713	opak	20
$512 \times 512 \times 512$	82613	opak	20
$512 \times 512 \times 512$	125545	opak	20
$512 \times 512 \times 512$	1224	transparent	9
$512 \times 512 \times 512$	36713	transparent	7
$512 \times 512 \times 512$	82613	transparent	6
$512 \times 512 \times 512$	125545	transparent	5

Tabelle 2: Performance

Datensatz	**Vertices**	**Dreiecksflächen**	**Tetraeder**
Cube_1224	321	2665	1224
Cube_36713	7210	76314	36713
Cube_82613	16337	171860	82613
Cube_125545	28206	264431	125545
Skull_55950	11017	114910	55950

Tabelle 3: Eigenschaften der verwendeten Gitter

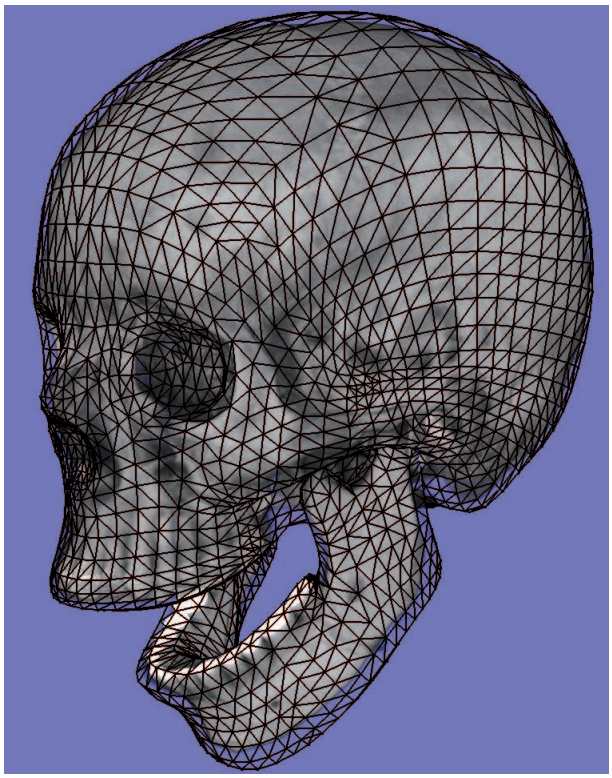

Abbildung 23: Der Schädeldatensatz, Volumen und Tetraedergitter

unabhängig vom zugrundeliegenden Gitter.

Die Animation wurde anhand eines 16 bit $512 \times 512 \times 333$ CT-Datensatzes eines Schädels getestet (Abbildung 23). Ziel war es, den Unterkiefer zu animieren (Abbildung 24). Das eigens dafür erstellte Gitter besteht aus 55950 Tetraedern. Zudem handelt es sich dabei um ein nichtkonvexes Gitter, so dass je nach Blickpunkt zwischen 4 und 7 Depth Layers gerendert werden mussten. Der Performancetest hat ergeben, dass die Animation, d.h. die Aktualisierung der Vertex-Daten, keine erhebliche Rolle bezüglich der Performance spielt. Sowohl die animierte als auch die nicht animierte Darstellung der nichtkonvexen Geometrie erreichte Frameraten von 10 fps. Insofern profitiert der Ansatz dieser Arbeit enorm vom Unified Shader Model, weil die Anzahl der Vertexoperationen im Vergleich zu den Fragmentoperationen sehr viel geringer ist und die Last entsprechend verteilt werden kann.

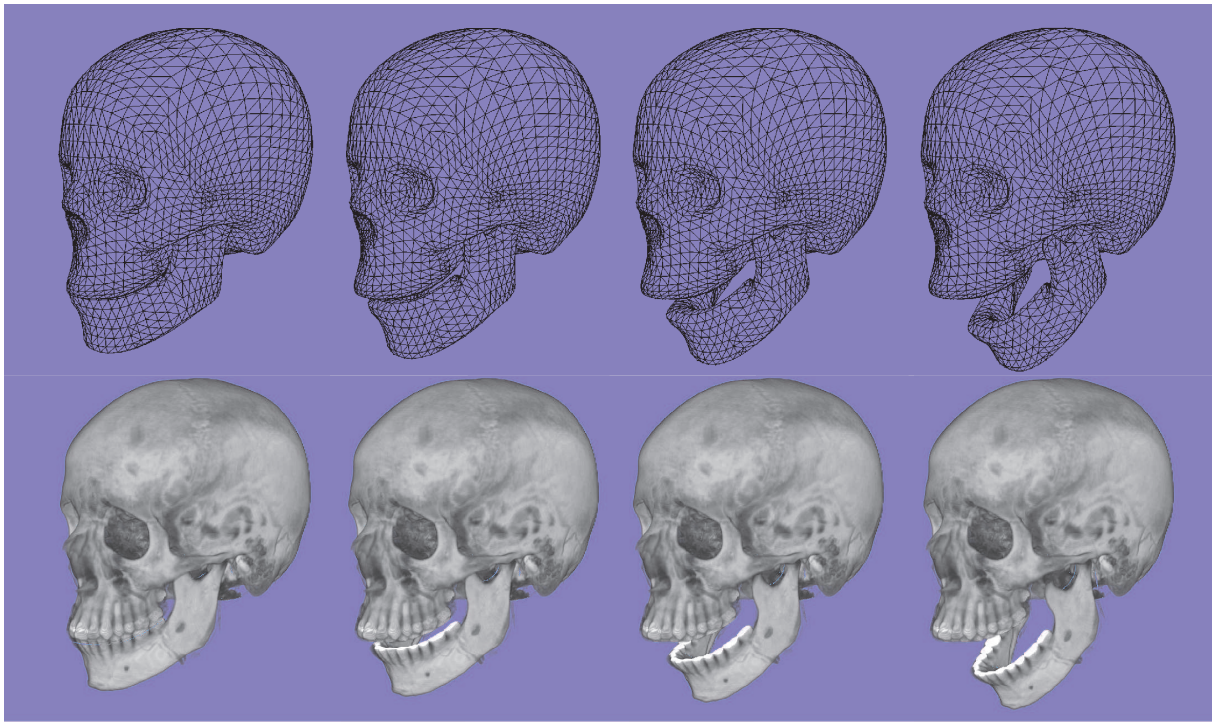

Abbildung 24: Animation des Schädeldatensatzes

6 Zusammenfassung und Ausblick

Der Ansatz, der in dieser Arbeit vorgestellt und implementiert wurde, ermöglicht die Echtzeitdeformation von Volumendaten, die auf uniformen Gittern basieren. Es wurde gezeigt, dass die Deformation durch die Verwendung eines Tetraedergitters als Proxygeometrie realisiert werden kann, indem bei gleichzeitiger Beibehaltung der Texturkoordinaten an den Vertices die Geometrie geändert wird. Die Geometrie und Topologie des Tetraedernetzes können vollständig im lokalen Speicher der Grafikkarte gehalten werden. Dadurch können hochqualitative Bilder durch GPU-basiertes Raycasting erzeugt werden. Die Implementierung profitiert dabei erheblich von der Flexibilität des Unified Shader Model.

Um die vorgestellte Methode der Volumendeformation praxistauglicher zu machen, ist es notwendig, die Tetraedergitter für zu deformierende Bereiche automatisch generieren zu können. Das in dieser Arbeit verwendete, an den Schädeldatensatz angepasste Tetraedergitter wurde aufwändig und langwierig von Hand erstellt. Für die klinische Praxis ist dieses Vorgehen natürlich untauglich. Hier besteht weiterer Forschungsbedarf, damit anhand gegebener Volumen- bzw. Segmentierungsdaten angepasste Gitter erzeugt werden können.

Weitere Herausforderungen finden sich auf dem Gebiet der Weichgewebedeformation. Die Deformation bzw. Animation von starren Strukturen wie dem Unterkieferknochen stellt keine große Herausforderung dar. Schwieriger ist es, Weichgewebe zu deformieren oder die Interaktionen zwischen verschiedenen Gewebearten korrekt darzustellen. Mit Hilfe der Finite Element-Methode oder Masse-Feder-Modellen könnten hier sicher sehr anschauliche Ergebnisse erzielt werden.

7 Anhang

7.1 Vertexshaderprogramm: tetra_vert.cg

```
void main(float3 VertexInA    : POSITION,  //Vertexkoordinaten der 3 Vertices
          float2 TetraIndex   : TEXCOORD0, //Tetraederindizes
          float3 VertexInB    : TEXCOORD1, //Vertexkoord. nächster Timenode
          uniform float4x4 ModelViewProj,
          uniform float4x4 ModelViewI,
          uniform float4x4 ModelViewIT,
          uniform float time,              //lokaler Interpolationsschritt

          out float4 VertexOut     : POSITION,
          out float3 PositionOut   : TEXCOORD1,
          out float2 TetraIndexOut : TEXCOORD2,
          out float3 VPosUnitCube  : TEXCOORD5
          )
{
    // interpoliere den Vertex "entlang der Zeit"
    // (bei statischen Netzen ist time=0 und VertexInB irrelevant)
    float3 VertexIn = lerp(VertexInA, VertexInB, time);
    //transformation in den Screenspace
    VertexOut     = mul(ModelViewProj, float4(VertexIn,1.0));
    VPosUnitCube  = (VertexOut.xyz/VertexOut.w +1.0)/2.0;
    PositionOut   = VertexIn;
    TetraIndexOut = TetraIndex;
}
```

Listing 21: Code des Vertexprogramms für die Geometrieinterpolation und die Transformation in den Screenspace

7.2 Fragmentshaderprogramm: depthpeeling_frag.cg

```
#define EPSILON (0.01)
void main(
          float3 PositionIn     : TEXCOORD1,
          float3 TetraIndex     : TEXCOORD2,
          float3 VPosUnitCube   : TEXCOORD5,
          uniform sampler2D DepthTexture,
          out float4 value0 : COLOR0,
          out float4 value1 : COLOR1
  )
{
   float2 uv    = VPosUnitCube.xy;
   // "bisherigen" Wert des Depth-Buffers lesen
   float depth = tex2D(DepthTexture,uv).x;
   value0 =  float4(0.0,0.0,0.0,0.0);
   value1 =  float4(0.0,0.0,0.0,0.0);
   // nur das zeichnen,
   // was hinter dem bisherigen Depth-buffer Wert liegt:
   if (depth+EPSILON < VPosUnitCube.z)
   {
      // hier ist A=1, der Alpha-Test schreibt dieses Fragment
      value0 = float4(TetraIndex.xy,0.0, 1.0); // target0 = TetraIndex
      value1 = float4(PositionIn,        1.0); // target1 = Position
   }
}
```

Listing 22: Code des Fragmentprogramms für das Depth Peeling

7.3 Vertexshaderprogramm: screen2D_vert.cg

```
void main(float4 VertexIn     : POSITION,
          float2 TexCoordIn   : TEXCOORD0,
          uniform float4x4 ModelViewI,

          out float4 VertexOut     : POSITION,
          out float2 TexCoordOut   : TEXCOORD0,
          out float3 EyePosOut     : TEXCOORD1
          )
{
   VertexOut   = VertexIn;
   TexCoordOut = TexCoordIn;
   EyePosOut   = mul(ModelViewI,    float4(0.0,0.0,0.0,1.0));
}
```

Listing 23: Code des Vertexprogramms für die Transformation in den Screenspace

7.4 Fragmentshaderprogramm: tetra2D_frag.cg

```
#define STEPSIZE (0.01)
float4 main(
            float3 screenUV     : TEXCOORD0,
            float3 EyePosition  : TEXCOORD1, // Konstant für die gesamte Szene

            uniform sampler2D DataV0V1,
            uniform sampler2D DataV2V3,
            uniform sampler2D DataNB01,
            uniform sampler2D DataNB23,
            uniform sampler2D DataVertices,
            uniform sampler2D TexCoords,
            uniform sampler3D VolumeTexture,
            uniform sampler1D TransFunc,
            uniform sampler2D PositionTex,
            uniform sampler2D TetraIndexTex
            ) : COLOR
{
   float4 temp = tex2D(PositionTex, screenUV);
   if (temp.a <= 0.0)
   {
      // Der Strahl schneidet das Volumen überhaupt nicht
      discard;
   }
   // Index des ersten Tetraeders, den der Sehstrahl schneidet
   float2 TetraIndex = tex2D(TetraIndexTex, screenUV);
   // Die Position des Schnittpunkts in Weltkoordinaten
   float3 PositionIn = temp.xyz;
   float3 rayDir = normalize(PositionIn - EyePosition);
   //Strahlberechnung: Startpunkt und Strahlrichtung

[Fortsetzung siehe nächste Seite]
```

```
   float3 rayPosition = PositionIn + STEPSIZE * rayDir;
   float3 rayTexCoord;
   float4 result = float4(0.0f,0.0f,0.0f,0.0f);
   float bBreak = 0.0;
   for(int w = 0; w < 10; w++) {
      for(int r = 0; r < 300; r++)
      {
         // Falls wir keinen gültigen Tetraeder mehr haben, brechen wir ab.
         if ((TetraIndex.x == 0.0) && (TetraIndex.y == 0.0)) {
            return result;
         }
         //Texturkoordinaten von v0, v1, v2, v3
         float4 texcoord_v0v1 = tex2D(DataV0V1,TetraIndex);
         float4 texcoord_v2v3 = tex2D(DataV2V3,TetraIndex);
         //Texturkoordinaten der Nachbarn
         float4 texcoord_NB01 = tex2D(DataNB01,TetraIndex);
         float4 texcoord_NB23 = tex2D(DataNB23,TetraIndex);

         //Koordinaten der Tetraedervertices v0-v3
         float3 v0 = tex2D(DataVertices,texcoord_v0v1.rg).xyz;
         float3 v1 = tex2D(DataVertices,texcoord_v0v1.ba).xyz;
         float3 v2 = tex2D(DataVertices,texcoord_v2v3.rg).xyz;
         float3 v3 = tex2D(DataVertices,texcoord_v2v3.ba).xyz;

         //uvw0-uvw3 3D-Texturkoordinaten holen
         float3 uvw0 = tex2D(TexCoords,texcoord_v0v1.rg);
         float3 uvw1 = tex2D(TexCoords,texcoord_v0v1.ba);
         float3 uvw2 = tex2D(TexCoords,texcoord_v2v3.rg);
         float3 uvw3 = tex2D(TexCoords,texcoord_v2v3.ba);

         // Berechnung der baryzentrischen Koordinaten der
         // aktuellen Position (rayPosition) entlang des Sehstrahls
         // bezüglich des aktuell geschnittenen Tetraeders
         float3 vec0 = v0-rayPosition;
         float3 vec1 = v1-rayPosition;
         float3 vec2 = v2-rayPosition;
         float3 vec3 = v3-rayPosition;
         float area0 = dot(cross(vec2,vec3),vec1);
         float area1 = dot(cross(vec3,vec2),vec0);
         float area2 = dot(cross(vec0,vec1),vec3);
         float area3 = dot(cross(vec1,vec0),vec2);
         float  area = area0 + area1 + area2 + area3;
```

[Fortsetzung siehe nächste Seite]

```
        // Wenn das Volumen des Tetraeders zu klein wird,
          // wird die Berechnung numerisch instabil.
          // In diesem Fall wird abgebrochen.
          if (area < 1.0e-6) {
             return result;
          }
          if (area0 < 0.0) {
             // Wähle Nachbarn an der Fläche 0
             TetraIndex = texcoord_NB01.rg;
          }
          else if (area1 < 0.0) {
             // Wähle Nachbarn an der Fläche 1
             TetraIndex = texcoord_NB01.ba;
          }
          else if (area2 < 0.0) {
             // Wähle Nachbarn an der Fläche 2
             TetraIndex = texcoord_NB23.rg;
          }
          else if (area3 < 0.0) {
             // Wähle Nachbarn an der Fläche 3
             TetraIndex = texcoord_NB23.ba;
          }
          else
          {
             rayTexCoord = uvw0 * area0 + uvw1 * area1 +
                           uvw2 * area2 + uvw3 * area3;
             rayTexCoord /= area;
             // Skalarfeld samplen
             float scalar =  tex3D(VolumeTexture, rayTexCoord).r ;
             float4 sample = tex1D(TransFunc, scalar);
             sample.rgb *= sample.a;
             //front to back compositing
             result += (1.0 - result.a) * sample;
             if (result.a > 0.9999) {
                bBreak = 1.0; break;
             }
             //Strahlintegration
             rayPosition += STEPSIZE * rayDir;
          }
       }
    }
    return result;
}
```

Listing 24: Code des Fragmentprogramms für das Raycasting

Literaturverzeichnis

Literatur

[1] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. *Proc. SIGGRAPH '88, Computer Graphics*, 22(22):65–74, 1988.

[2] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf. *Real-Time Volume Graphics*. A K Peters, Ltd, 2006.

[3] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA OpenGL Applications Engineering, 2001.

[4] S. Fang, R. Srinivasan, S. Huang, and R. Raghavan. Deformable volume rendering by 3D texture mapping and octree encoding. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 73–80, 1996.

[5] P. Hastreiter, C. Rezk-Salama, G. Soza, G.Greiner, R. Fahlbusch, O. Ganslandt, and C. Nimsky. Strategies for Brain Shift Evaluation. *Medical Imaging*, 8(4):447–464, 2004.

[6] J. Kniss, G. Kindlmann, M. Hadwiger, C. Rezk-Salama, and R. Westermann. High-Quality Volume Graphics on Consumer PC Hardware. ACM SIGGRAPH Course Notes 42, 2002.

[7] Martin Kraus. *Direct Volume Visualization of Geometrically Unpleasant Meshes*. PhD thesis, Universität Stuttgart, 2003.

[8] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 215–222, Washington, DC, USA, 2001. IEEE Computer Society.

[9] Yair Kurzion and Roni Yagel. Space deformation using ray deflectors. In *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pages 21–30, New York, 1995. Springer-Verlag.

[10] Yair Kurzion and Roni Yagel. Interactive space deformation with hardware-assisted rendering. *IEEE Computer Graphics and Applications*, 17(5):66–77, /1997.

[11] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[12] P. Muigg, M. Hadwiger, H. Doleisch, and H. Hauser. Scalable hybrid unstructured and structured grid raycasting. In *IEEE Visualization 2007*, pages 1592–1599, 2007.

[13] Zoltan Nagy and Reinhard Klein. Depth-peeling for texture-based volume rendering.

[14] Christophe Geuzaine (University of Liège) and Jean-François Remacle (Catholic University of Louvain). Gmsh. GNU General Public License (GPL).

[15] Christof Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware*. PhD thesis, Universität Erlangen-Nürnberg, 2002.

[16] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Proc. SIGGRAPH '88, Computer Graphics*, 24(24):51–58, 1988.

[17] C. Silva, J. Comba, S. Callahan, and F. Bernardon. A Survey of GPU-Based Volume Rendering of Unstructured Grids. *Brazilian Journal of Theoretic and Applied Computing (RITA) 12:2*, pages 9–29, 2005.

[18] Manfred Weiler, Martin Kraus, Merkus Merz, and Thomas Ertl. Hardware-Based Ray Casting for Tetrahedral Meshes. *Proceedings of IEEE Visualization 2003*, pages 333–340, 2003.

[19] Manfred Weiler, Paula N. Mallón, Martin Kraus, and Thomas Ertl. Texture-Encoded Tetrahedral Strips. *Proceedings of IEEE Symposium on Volume Visualization*, pages 71–78, 2004.

[20] Rüdiger Westermann and Christof Rezk-Salama. Real-time volume deformation. In *Proceedings of Eurographics*, Aire-la-Ville, Switzerland, 2001. Eurographics Association.