ulm university universität

Entwicklung und Programmierung von Algorithmen zur klinisch relevanten Auswertung zeitabhängiger, dreidimensionaler Daten simulierter Frakturheilungsprozesse

Frank Niemeyer

Diplomarbeit an der Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Medieninformatik

und dem

Institut für Unfallchirurgische Forschung und Biomechanik (UFB)

sowie dem

Ulmer Zentrum für Wissenschaftliches Rechnen (UZWR)

Gutachter/Betreuer:
Prof. Dr. rer. nat. Alexander Keller
Dr.-Ing. Ulrich Simon

Ulm, 22. Juli 2007

Inhaltsverzeichnis

Danksagung

Auch wenn ich die vorliegende Arbeit selbständig verfaßt habe, heißt das nicht, daß sie ohne die Mithilfe und wertvollen Anregungen Dritter in dieser Form möglich gewesen wäre.

Herrn Prof. Dr. Dr. Widder, Ärztlicher Direktor der Klinik für Neurologie und Neurologische Rehabilitation des Bezirkskrankenhauses Günzburg, danke ich dafür, mich vor mehr als zwei Jahren auf die Idee gebracht zu haben, mich am UFB bei Prof. Claes als Praktikant zu bewerben.

Nur so war es überhaupt möglich, daß ich dort Herrn Dr. Ulrich Simon kennenlernte, der mich auch beim Erstellen dieser Arbeit betreute. Ihm möchte ich meinen Dank nicht nur für seine jederzeit zuvorkommende fachliche Unterstützung aussprechen, sondern insbesondere für seine motivierende, antreibende Art, mit Menschen umzugehen.

Herrn Prof. Dr. Keller danke ich für die stets freundliche und vor allem reibungslose Kommunikation. Die konstruktive Kritik seinerseits und das von ihm freundlicherweise zur Verfügung gestellte Material erwiesen sich als sehr nützlich bei der Erstellung der Arbeit.

Herrn Prof. Dr. Claes, Leiter des Instituts für Unfallchirurgische Forschung und Biomechanik (UFB), danke ich vor allem für eines: seine Geduld. Ich weiß nicht mehr genau, für wann ich ihm den Abschluß meines Studium prophezeit habe. Aber ich weiß nun, daß meine Glaskugel nicht funktioniert.

Den Kollegen am UFB und UZWR danke für ein tolles, kollegiales Arbeitsklima, um das man mich nur beneiden kann.

Der größte Dank aber gebührt ohne Frage meinen Eltern. Ohne sie, ohne ihre seelische, moralische und natürlich auch finanzielle Unterstützung, wäre ich heute nicht dort, wo ich nun mal glücklicherweise bin.

Danke.

Ulm, im Juli 2007

Abbildungsverzeichnis

1 Einleitung

Bei Knochen scheint es sich, oberflächlich betrachtet, um einen vergleichsweise statischen Gewebetyp zu handeln, der, einmal ausgebildet, im Laufe des Lebens keine wesentlichen Umbildungsprozesse mehr durchlebt. Dieser Schein trügt jedoch. Knochen kann man wohl guten Gewissens als einen der flexibelsten Bausteine des Bewegungs- und Stützapparats des Körpers bezeichnen, das in ständigem Umbau begriffen ist, um sich optimal an sich verändernde Bedingungen anzupassen.

Man denke zum Beispiel an Astronauten, deren muskolo-skeletales System in der Schwerelosigkeit auf Grund der fehlenden Anziehungskraft schnell degeneriert. Der Körper spart durch diesen Anpassungsprozeß Körpermasse und Energie, denn auch das Knochengewebe muß mit Blut versorgt und ernährt werden. Andererseits ist Knochengewebe auch zur Regeneration fähig, ja es ist sogar das einzige Gewebe des Bewegungsapparats, das nach der Heilung eines Defekts keinerlei Narbengewebe aufweist und die ursprüngliche Struktur und Funktion vollständig wiederherstellen kann [Ben91].

Der Heilungsprozeß von Knochenfrakturen ist ein physiologisch und biomechanisch äußerst komplexer Vorgang, der heute noch nicht vollständig verstanden und weiterhin Gegenstand aktiver Forschung ist. Von besonderem Interesse ist dabei, welchen Einfluß mechanische Stimuli auf die Gewebedifferenzierung, und damit auf den Heilungsprozeß selbst, ausüben. Wäre dieser Zusammenhang bekannt, ermöglichte dies auch eine bessere Behandlung von Knochenfrakturen, indem man die eingesetzten Behandlungsverfahren besser auf den jeweiligen Fall abstimmen könnte.

1.1 Grundbegriffe der Frakturheilung

Bevor wir uns dem eigentlichen Thema, der Simulation von Knochenheilungsprozessen, widmen können, folgen nun zunächst einige (kurze) Abschnitte, die wichtige Grundbegriffe erläutern, die zum weiteren Verständnis der Inhalte un-

bedingt erforderlich sind. Sofern nicht anders angegeben, basiert dieser Abschnitt inhaltlich auf den Ausführungen in [Hei98 Kapitel 1].

1.1.1 Knochengewebe

Knochengewebe (*Textus osseus*) gehört, wie auch das (eigentliche) Bindegewebe, Knorpel oder Sehnen und Bänder, zur Familie der *Binde- und Stützgewebe*, die einen Hauptteil der Körpermasse bilden. Diese Gewebe bestehen aus den für das jeweilige Gewebe charakteristischen Zellen sowie einer die Zellen untereinander verbindenden *extrazellulären Matrix*, deren Zusammensetzung und Struktur entscheidend für die mechanischen Eigenschaften des Gewebes ist. Die Knochengrundsubstanz besteht zu 50 % aus Mineralien, 25 % organischen Bestandteilen und weiteren 25 % Hydratationswasser.

Aus pluripotenten[1] mesenchymalen[2] *Vorläuferzellen* differenzieren sich *Osteoblasten*, die für die Knochenneubildung zuständig sind, indem sie die organischen Bestandteile der Knochensubstanz synthetisieren. Von Knochengrundsubstanz vollständig umgebene Osteoblasten bezeichnet man als *Osteozyten*. Sie sorgen für die Aufrechterhaltung des Knochengewebes. Die mehrkernigen, sich amöboid bewegenden *Osteoklasten* weisen eine enge Verwandtschaft mit den Markrophagen (Freßzellen) des Immunsystems auf und dienen der Resorption von Knochengewebe. Die Mineralien (insbesondere Kalzium) der phagozytierten Knochensubstanz gelangen dadurch in den Blutkreislauf.

Es existieren zwei verschiedene Grundtypen von Knochen. Bei jeder Knochenneubildung entsteht zunächst *Geflechtknochen*. Die Knochenzellen und Kollagenfasern in Geflechtknochen sind unregelmäßig verteilt und besitzen keine einheitliche Ausrichtung. Später ersetzt *Lamellenknochen* den ursprünglichen Geflechtknochen. Lamellenknochen besitzt im Gegensatz zum Geflechtknochen regelmäßige Strukturen und tritt in zwei Unterformen auf.

Die äußerste Knochenschicht (*Corticalis* oder *Substantia compacta*) setzt sich aus ineinander verschachtelten *Osteonen* (*Havers-Systeme*) zusammen (siehe Abbildung 1.1). Diese bestehen aus konzentrisch angeordneten Knochenlamellen aus parallel verlaufenden Kollagenfasern, in die über Kanäle verbundene Osteozyten eingebettet sind. In der Mitte jedes Osteons verläuft der sog. *Havers-Kanal*, in dem sich Blut- und Lymphgefäße sowie Nerven befinden, die der Versorgung des Knochengewebes dienen. Die Corticalis wird umhüllt vom *Periost*, der Knochenhaut, die zum einen für die Blutversorgung des Knochens von Bedeutung ist,

1 Pluripotenz ist die Fähigkeit einer Zelle, sich in jeden beliebigen Zelltyp differenzieren zu können.

2 Mesenchym: embryonales Bindegewebe

zum anderen Vorläuferzellen bereit hält, die sich bei Bedarf zu knochenbildenden Osteoblasten entwickeln können.

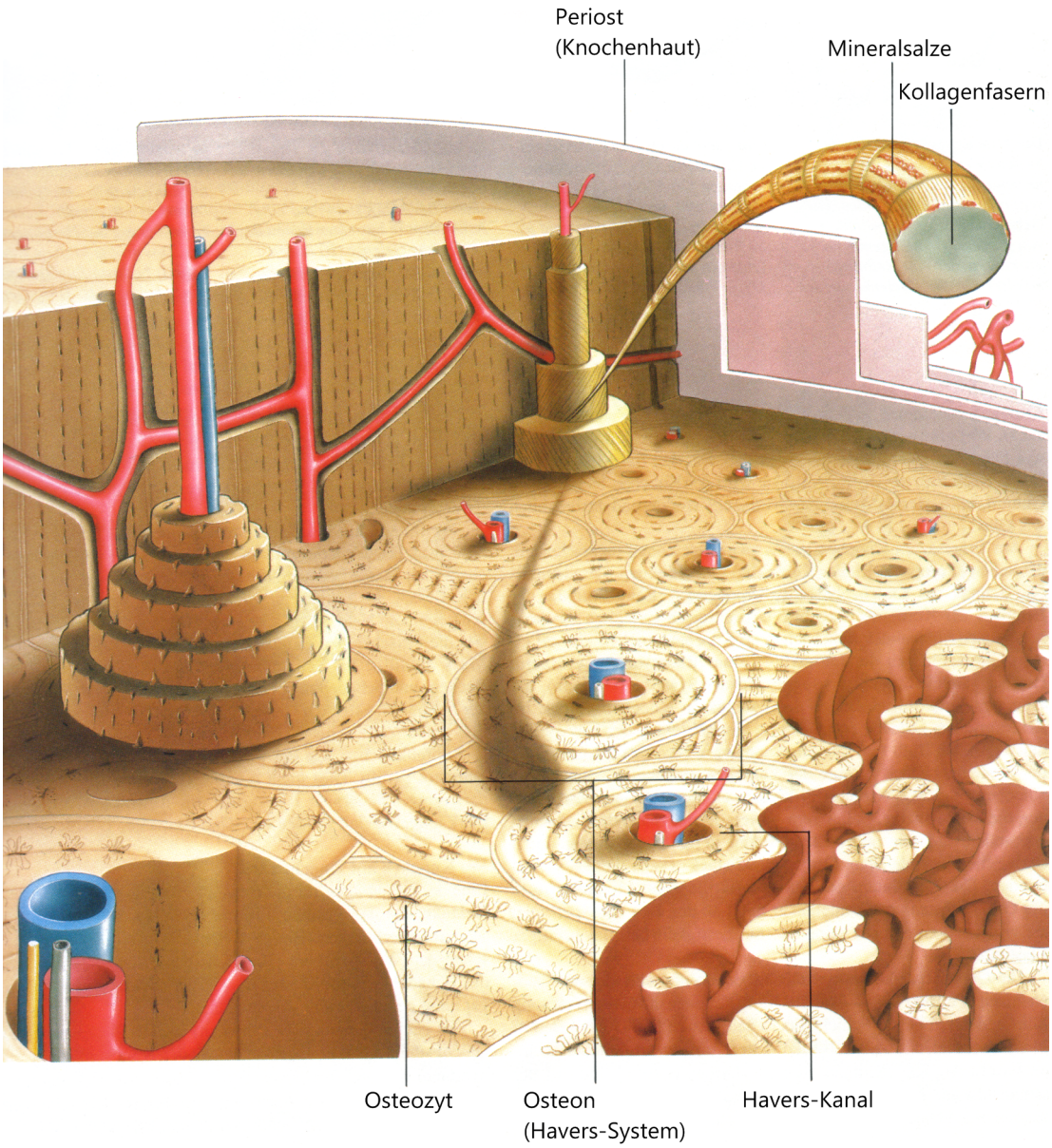


Abbildung 1.1: Aufbau von Lamellenknochen
(Quelle: [Ben91])

Im Inneren von kurzen, platten Knochen oder den Epiphysen (Schaft-Enden) einiger langer Röhrenknochen findet man hingegen schwammartige Strukturen, die *Substantia spongiosa*. Sie besteht aus einem feinen Geflecht von Knochenbälkchen, den *Trabekeln*, die entlang der Haupt-Zug- und Drucklinien des Knochens ausgerichtet sind. Diese Bauweise gewährleistet maximale Belastbarkeit bei minimalem Gewicht.

Abbildung 1.2: Substantia spongiosa (trabekulärer Knochen, vergrößert) (Quelle: [Ben91])

1.1.2 Ossifikation

Die Bildung von Knochengewebe (*Ossifikation*) findet entweder *desmal* oder *chondral* statt, das Ergebnis beider Vorgänge ist jedoch gleichwertig.

Bei der desmalen Ossifikation, auch *direkte Knochenbildung* genannt, gehen die knochenbildenden Osteoblasten direkt aus den pluripotenten Zellen des Mesenchyms (embryonales Bindegewebe) hervor. Die desmale Ossifikation kann nur

unter stabilen mechanischen Bedingungen und ausreichend guter Blutversorgung erfolgen.

Sind diese Randbedingungen nicht gegeben, erfolgt die Verknöcherung über einen Zwischenschritt, bei dem ein Knorpelmodell des zu bildenden Knochengewebes Schritt für Schritt durch Knochen ersetzt wird. Diese *indirekte Knochenbildung* bezeichnet man als *chondrale Ossifikation*. Die Interzellularsubstanz des Knorpels verkalkt mit der Zeit, die eingeschlossenen *Chondrozyten* (Knorpelzellen) sterben ab. *Chondroklasten* – das Analogon zu den Osteoklasten des Knochengewebes – öffnen die Knorpelhöhle, so daß Blutgefäße einwachsen können, die auch mesenchymale Vorläuferzellen mit sich bringen. Diese differenzieren sich zu die Knochengrundsubstanz (*Osteoid*) erzeugenden Osteoblasten. Das Osteoid füllt den von den abgestorbenen Chondrozyten und dem resorbierten Knorpelgewebe zurückgelassenen Raum, wodurch die charakteristischen Knochenbälkchen der Substantia spongiosa entstehen. Diese spezielle Form der chondralen Ossifikation nennt man *enchondrale Ossifikation*, weil der Knochen von innen heraus entsteht. Ersetzt er das Knorpelgewebe von außen her, spricht man von *perichondraler Ossifikation*. Die Substantia compacta entsteht durch perichondrale Ossifikation.

Grundsätzlich entsteht bei jeder Art der Ossifikation zunächst Geflechtknochen. Dieser kann später, abhängig von der Lastsituation, einen Umbau in Lamellenknochen erfahren.

1.1.3 Primäre Frakturheilung

Die Fraktur eines Knochens erfordert die Neubildung von Knochengewebe. Da die unterschiedlichen Ossifikationsprozesse (siehe den vorherigen Abschnitt) jeweils nur unter bestimmten mechanischen und physiologischen Bedingungen ablaufen können, beeinflussen diese auch die Art der Frakturheilung selbst.

Die *Primäre Frakturheilung* setzt absolute Stabilität und eine gute Durchblutung voraus, denn die Knochenbildung erfolgt hierbei ausschließlich desmal, also ohne knorpeliges Gerüst. Handelt es sich um eine Kontaktheilung, berühren sich also die beiden Frakturteile, findet die Überbrückung direkt mit Knochengewebe statt. Im Falle einer Spaltheilung, beispielsweise bei einer Verlängerungsosteotomie, füllt sich der Spalt innerhalb einiger Wochen mit Geflechtknochen, bevor parallel zum Schaft ausgerichtete Osteonen wachsen, die den neuen Lamellenknochen bilden.

1.1.4 Sekundäre Frakturheilung

Bei der sekundären Frakturheilung erfolgt die Heilung über mehrere Zwischenschritte. Bei Vorhandensein von Bewegungen im Frakturspalt (*IFM*[3]) entsteht nur entlang der steifen und gut durchbluteten Corticalis bereits wenige Tage später durch desmale Ossifikation ein Gerüst aus Geflechtknochen, das sich Richtung Frakturspalt ausbreitet [Ein95, CRD87, Stü87, Pau65]. Interfragmentär bildet sich, sofern die IFMs nicht zu groß sind, ein sogenannter *Kallus*. Er dient der Ruhigstellung der Frakturzone durch die Vergrößerung der Querschnittsfläche und Einlagerung von Gewebe zunehmender Steifigkeit, die eine knöcherne Überbrückung des Frakturspalts erst ermöglicht.

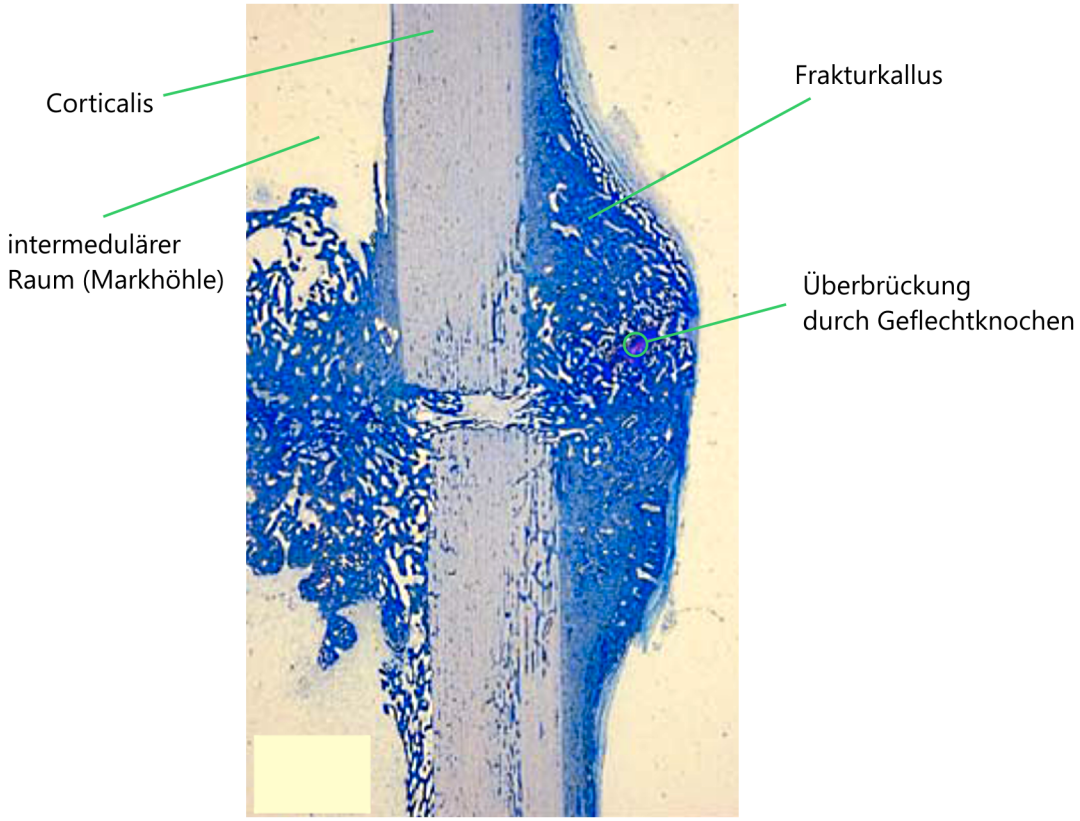


Abbildung 1.3: Schnitt durch einen Kallus einer Tibiafraktur beim Schaf (Quelle: [ABS03])

Der Kallus besteht zunächst aus einer Mischung aus Hämatom, Granulationsgewebe, fibrösem Gewebe, Knorpel und Knochen, die eine geringe Steifigkeit aufweist. Im Laufe des Heilungsprozesses wächst Knorpelgewebe, das später verkalkt und die interfragmentären Bewegungen weiter reduziert. Das Knorpelge-

3 IFM: *Interfragmentary Movement*

rüst verknöchert bei ausreichender Blutversorgung schließlich enchondral. Innerhalb einiger Monate bis Jahre baut sich der Kallus ab, die ursprüngliche Knochengeometrie wird wiederhergestellt [Ein95].

1.2 Heilungssimulation

Während man früher zur Behandlung diaphysärer[4] Frakturen auf eine möglichst starke Fixierung setzte, um die primäre Frakturheilung zu ermöglichen, kommen heute meist flexible Osteosynthesverfahren (Fixateur externe, Gibsschiene, Marknagel) zum Einsatz, die gewisse interfragmentäre Bewegungen erlauben und dadurch eine schnelle Kallusbildung und somit die sekundäre Frakturheilung fördern.

Wie wir bereits in den vorherigen Abschnitten gesehen haben, hängen Heilungsverlauf und -erfolg wesentlich von den herrschenden Umgebungsbedingungen und der mechanischen Stimulation der Zellen im Kallus ab. So wirkt sich einerseits ein zu großer Frakturspalt negativ auf die Heilung aus, während andererseits geringfügige interfragmentäre Bewegungen die Osteoblastenprofliferation und -aktivität stimulieren und den Heilungsprozeß dadurch beschleunigen.

Daher ist es gerade bei den heute eingesetzten flexiblen Osteosyntheseverfahren von enormer Bedeutung, die mechanischen Parameter, z. B. die Steifigkeit eines Fixateurs, „richtig“ zu wählen. Andernfalls kann sich die Heilung entweder deutlich verzögern oder es kommt gar zur Entwicklung einer Pseudarthrose[5], da die auftretenden IFMs zwar zu groß für eine primäre Frakturheilung, aber zu gering für die sekundäre Frakturheilung sind. Gesucht ist also ein optimaler mechanischer Stimulus, der die Dauer des Heilungsprozesses minimiert

Zwar erlauben in vitro Versuche mit Zellkulturen die Erforschung des Einflusses mechanischer Stimuli auf die Aktivität einzelner Zellen. Die komplexen lokalen mechanischen Bedingungen, die im Frakturkallus herrschen, können jedoch nicht in vivo gemessen werden. Wie sich eine Änderung der Fixateur-Parameter auf die Heilung auswirken, kann man folglich nicht direkt erfassen.

Um den Einfluß mechanischer Parameter besser verstehen zu können, geht man an diesem Punkt zu einer Simulation des Heilungsprozesses über. Die Simulation verknüpft die Erkenntnisse aus in vitro Experimenten und in vivo Tierexperimenten, indem sie einen Zusammenhang zwischen den gemessenen globalen IFMs und den mechanischen Stimuli auf zellulärer Ebene herstellt.

4 Diaphyse: Schaft eines Röhrenknochens

5 Pseudarthrose: „Falschgelenk“, d.h. die Fraktur wird nicht knöchern überbrückt

Die Simulation soll letztlich zu einem besseren Verständnis des Heilungsprozesses dienen, und somit die Voraussetzungen für die zuverlässige Vorhersage von Heilungsverläufen und die optimale Behandlung von Frakturen liefern [DFG99].

1.2.1 FEM in der Biomechanik

Die *Methode der finiten Elemente* (*FEM*) ist ein numerisches Verfahren, dessen heutige Form Mitte bis Ende der 1950er Jahren zur Lösung von Festkörperproblemen unter anderem von Forschern der UC Berkeley und bei Boeing entwickelt wurde. Die zugrundeliegende Idee ist es, ein geometrisches Modell, dessen mechanische Eigenschaften aufgrund seiner Komplexität nicht direkt berechenbar sind, in viele kleine, endlich-große (finite) Elemente zu zerlegen [CW99]. Diese Elemente können im zweidimensionalen Fall beispielsweise Dreiecke, im dreidimensionalen Tetra- oder Hexaeder sein. Bei der sogenannten Verschiebungsmethode erlaubt dieses elementweise Vorgehen die Approximation der Verschiebung und daraus wiederum die Berechnung der Verzerrungen und Spannungen, die ein Körper unter mechanischer Last in jedem Punkt erfährt. Verallgemeinerte Varianten ermöglichen die Anwendung der FEM auf Probleme der Thermodynamik, die Simulation elektromagnetischer Felder oder die Lösung von Strömungsproblemen (CFD[6]).

Für Problemstellungen der Biomechanik bietet sich die FEM damit geradezu an: Irreguläre Geometrien und komplexe nicht-lineare Materialeigenschaften sind hier allgegenwärtig. Die FEM in Kombination mit entsprechender Hardware ermöglicht die Analyse mechanischer Eigenschaften komplizierter Strukturen wie organischem Gewebe. In den letzten Jahrzehnten setzten verschiedene Forschergruppen auf die FEM zur Simulation des Knochenumbaus und der Gewebedifferenzierung oder auch zur Optimierung von Implantaten.

Frühe Heilungsmodelle nutzten einfache Gewebemodelle und waren rein statischer Natur [CBB88], d.h. die Analyse des Heilungsvorgangs erfolgte nur zu einigen wenigen festen Zeitpunkten. Bald erlaubten ausgefeiltere Gewebemodelle [CMD91, BCB89, DCH86] eine besser Vorhersage der Gewebedifferenzierung.

1.2.2 Dynamisches Frakturheilungsmodell

Um ein tiefgehendes Verständnis für den Heilungs*prozeß* zu gewinnen, ist jedoch zwingend eine *dynamische* Simulation von Nöten. Ament stellte dazu 1995 erstmals ein iteratives, dynamisches Heilungsmodell vor, bei dem lediglich der Anfangszustand vordefiniert ist und die Gewebetransformation auf einer Fuzzy-Lo-

6 CFD: *Computational Fluid Dynamics*

gic beruht, die gemäß gegebener Regeln den Umbau des Gewebes abhängig von den mechanischen Bedingungen steuert [AHA95].

Noch einen Schritt weiter geht das von Simon et. al entwickelte Modell [SAC04, SAC04a], das zusätzlich zu den mechanischen auch physiologische Faktoren, insbesondere die Durchblutung (*Vaskularität*), berücksichtigt. Das dreidimensionale Modell besteht aus einer unveränderlichen, axial-symmetrischen (Rotationssymmetrie entlang der Knochenlängsachse) FE-Geometrie, die die Corticalis und den Heilungsraum festlegt. Bei rein axialer Last nutzt das Modell zusätzlich die Spiegelsymmetrie zur Osteotomie-Ebene.

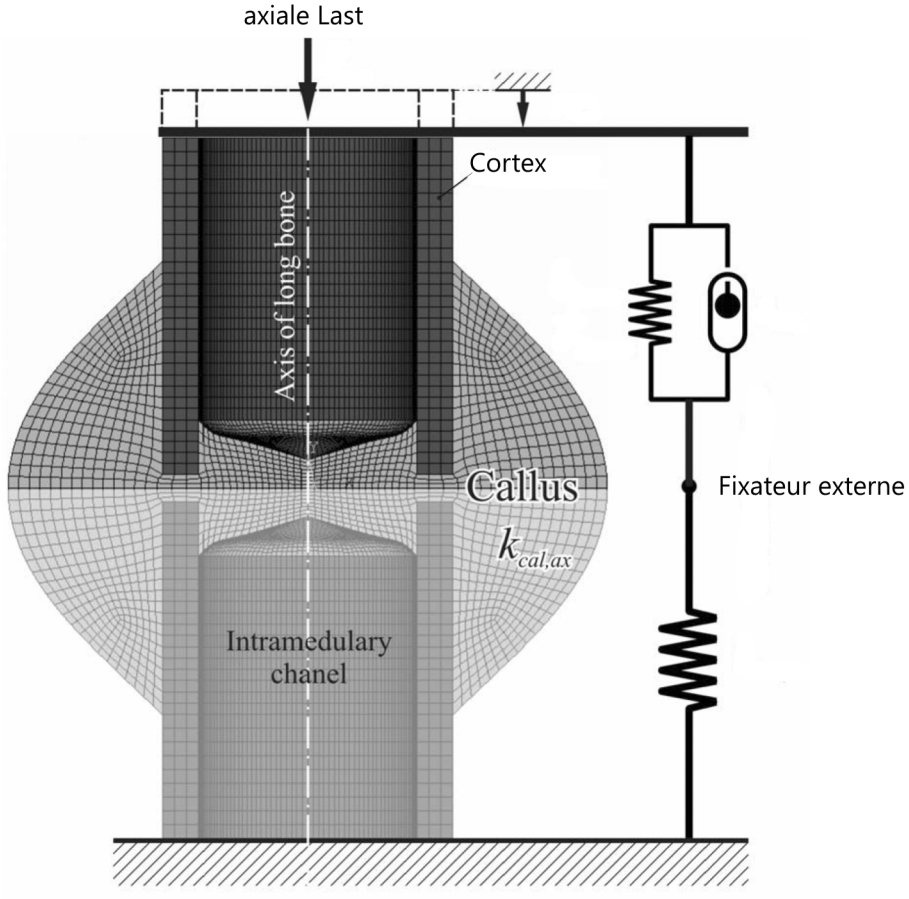


Abbildung 1.4: Finite-Elemente-Modell eines Frakturkallus unter axialer Last (Quelle: [SAC04])

Auch dieses Modell reguliert die Gewebedifferenzierung mit Hilfe von Fuzzy-Logic. Allerdings basieren die Fuzzy-Regeln in diesem Fall auf quantitativen Daten aus histologischen Untersuchungen von Zellkulturen und Tierexperimenten. Eine solche Regel könnte beispielsweise lauten: „Wenn in der Nähe bereits Knochen existiert und die Durchblutung gut ist und die mechanische Belastung ge-

ring ist, dann erhöhe die Knochenkonzentration.“ Wie man sieht, erlaubt die Fuzzy-Logic die Formulierung von „unscharfen“ Regeln (daher auch der Name), wie man sie in der Natur häufig antrifft. Die Zuordnung eines Elements zu einem Gewebetyp ist ebenfalls nicht binär, sondern jedes Element beinhaltet unterschiedliche Anteile (ausgedrückt als relative Konzentrationen) der Gewebetypen Bindegewebe, Knorpel oder Geflechtknochen. Innerhalb eines Elements setzt man eine konstante Gewebezusammensetzung voraus. Abbildung 1.5 verdeutlicht die Abfolge der einzelnen Simulationsschritte.

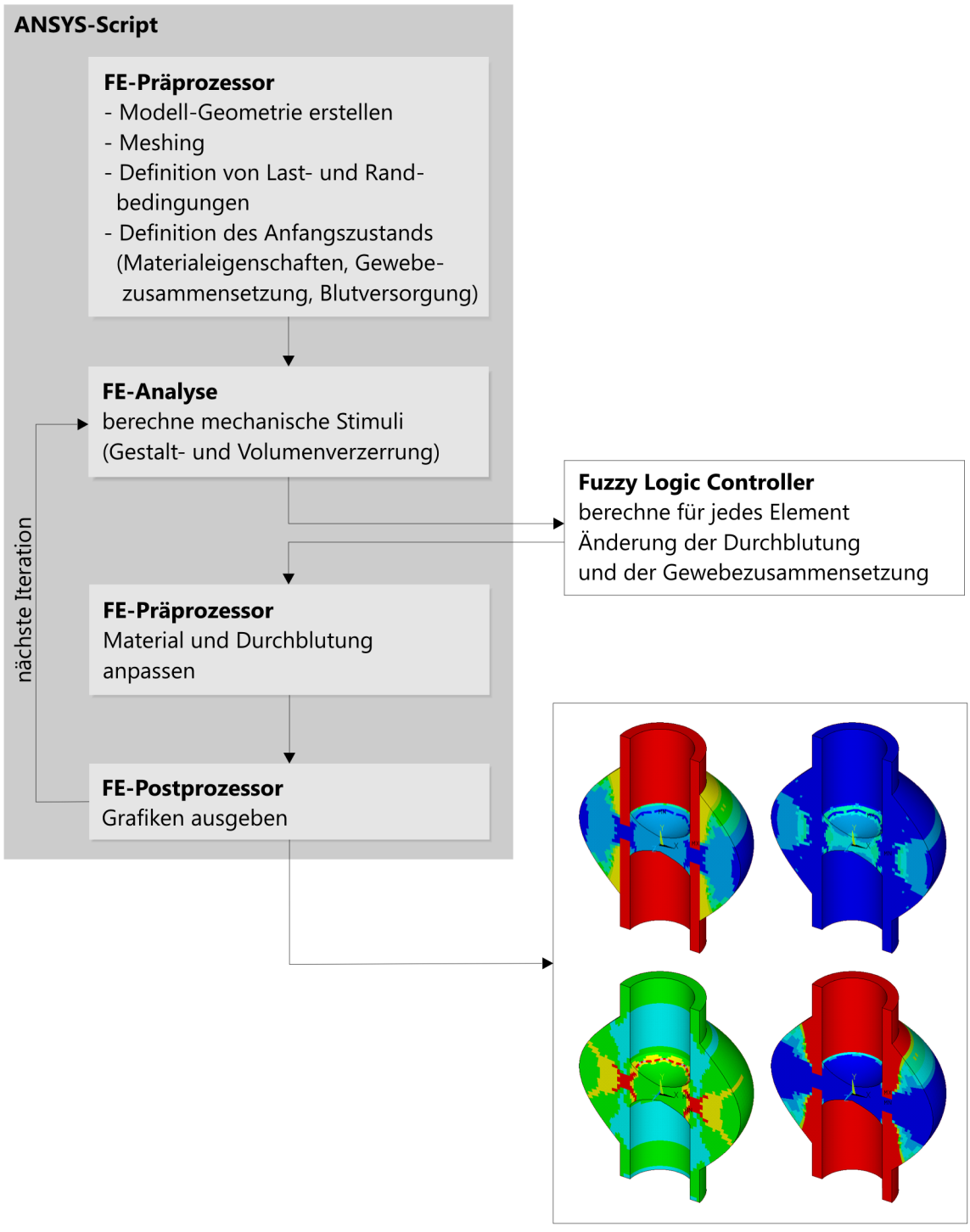


Abbildung 1.5: Schematischer Ablauf der Frakturheilungssimulation
Die Heilungssimulation ist in Form eines ANSYS-Scripts und eines externen Fuzzy-Logic-Controllers implementiert. Vor Beginn der Simulation gibt man die Zahl der zu berechnenden Iterationen vor.

1.3 Aufgabenstellung

Im Rahmen dieser Diplomarbeit soll die bestehende Frakturheilungssimulation um zwei weitere Aspekte erweitert werden.

Volumenvisualisierung der Simulationsergebnisse

Um die Überprüfbarkeit des Modells zu verbessern, soll ein Programm implementiert werden, das aus den Ergebnissen eines Simulationslaufs Röntgen- oder CT-ähnliche Bilder erzeugt. Die so synthetisierten „Röntgenaufnahmen“ können mit tierexperimentell gewonnenen Daten verglichen werden, was die visuelle Überprüfung der Gültigkeit des Modells erleichtert. Außerdem erlauben solche Bilder einen Blick „ins Innere“ des Kallus und somit einen Einblick in den simulierten Heilungsprozeß, den das FE-Programm ANSYS nicht bieten kann.

Automatisierte Überbrückungsdetektion

Eines der Ziele der Heilungssimulation ist es, Frakturen durch optimale Wahl der Designparameter der Fraktur-Fixation bestmöglich versorgen zu können und eine schnellstmögliche Heilung zu erreichen. Um eine solche Parameteroptimierung durchführen zu können, ist es zum einen notwendig, ein eindeutiges Abbruch-Kriterium für die Simulation zu finden, also zu definieren, wann eine Fraktur als geheilt gilt. Dieser Zeitpunkt muß automatisch erkannt und die ermittelte Heilungsdauer als skalarer Wert zurückgegeben werden, so daß man im folgenden numerische Verfahren anwenden kann, um die Fixateur-Parameter zu optimieren.

Des weiteren ist die Integration beider zu entwickelnden Programme in die bestehende Simulation erforderlich, die zur Zeit aus APDL-Scripten (FEA) und einer Matlab-Komponente für die Umsetzung der Fuzzy-Logic-Regeln besteht.

2 Volumenvisualisierung von FE-Modellen

Das in Abschnitt 1.2.2 beschriebene Programm zur Simulation von Frakturheilungsprozessen stellt mechanische und physiologische Eigenschaften des simulierter Gewebes in Form farbcodierter Grafiken dar (vgl. Abbildung 2.1). Dies hat zwei wesentliche Nachteile: Zum einen wird nur die Oberfläche der FEM[7]-Modelle visualisiert; die Strukturen und Eigenschaften im Inneren des Modells bleiben verborgen. Zum anderen ist dadurch ein Vergleich der Ergebnisse der Simulation mit klinischen oder tierexperimentell gewonnenen Daten nur in unzureichendem Maße möglich.

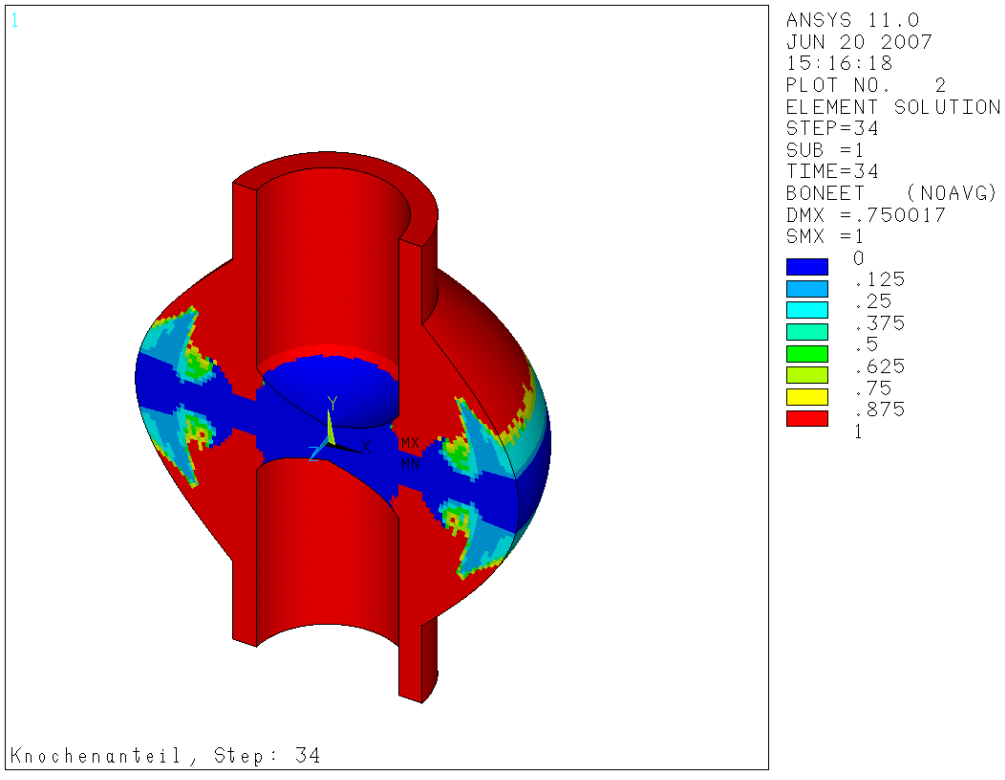


Abbildung 2.1: ANSYS-Plot der Knochenverteilung im FE-Frakturheilungsmodell

7 FEM: *Finite Elemente Methode*

Um diesen Vergleich in Zukunft zu vereinfachen, soll aus der Gittergeometrie des FE-Modells einerseits und der durch die Simulation ermittelten Verteilung der Knochendichte andererseits unter Zuhilfenahme von Verfahren der Volumenvisualisierung (*Volume Rendering*) eine röntgenbild-ähnliche Volumendarstellung synthetisiert werden, auf die sich schließlich auch die in 3.1 erläuterten üblichen klinischen Regeln zur Beurteilung des Heilungsverlaufes anwenden lassen können sollen.

2.1 Volume-Rendering-Verfahren im Überblick

Ob bildgebende Verfahren wie CT[8], PET[9] und MRT[10] oder FE-Simulationen: all diese Anwendungen arbeiten mit Volumendaten, die es in der Regel auch an einem Punkt der Verarbeitungskette zu visualisieren gilt. Während CT und MRT skalare Werte (Dichte, Wassergehalt, Temperatur, etc.) für diskrete Punkte (*Voxel*) innerhalb eines meist regulären, dreidimensionalen Gitters liefern, arbeiten FE-Simulationen mit Polyedern, am häufigsten Tetra- oder Hexaedern, die jeweils einen kleinen Ausschnitt des zu simulierenden Volumens repräsentieren. Die Gitter (*Mesh*) sind in diesem Fall normalerweise irregulär.

Ziel des Volume Rendering ist es, die dreidimensionalen Volumendaten auf ein zweidimensionales Bild zu projizieren. Je nach Anwendungsfall kann dabei die Darstellung der Volumenoberfläche genügen. Dazu werden polygonbasierte Approximationen der Isoflächen z. B. mittels des Marching-Cubes-Algorithmus generiert, die anschließend rasterisiert oder mittels Raytracing gerendert werden können (*Indirect Volume Rendering*) [MHBMC00]. Ein modernerer Ansatz ermöglicht das direkte Rendering von impliziten Isoflächen, eine Extraktion von Polygonoberflächen ist hierbei nicht mehr notwendig [WFMS05, Grü06]. Der größte Vorteil dieser Methode liegt darin, daß sie das interaktive Austesten verschiedener Iso-Werte und damit das „Browsen“ durch den Volumendatensatz ermöglicht.

Oft möchte man jedoch eine semi-transparente Darstellung des Volumens erreichen, um auch im Inneren verborgene Strukturen erkennen zu können. Das Ergebnis dieser als *Direct Volume Rendering* bezeichneten Verfahren – hier findet keine Konvertierung der Volumendaten in (Polygon-)Oberflächen statt – erinnert häufig stark an ein klassisches Röntgenbild, sofern keine farbcodierende Transferfunktion zum Einsatz kommt. Im folgenden möchte ich einen kurzen Über-

8 CT: *Computertomographie*

9 PET: *Positronen-Emissions-Tomographie*

10 MRI: *Magnetressonanztomographie, auch Kernspintomographie genannt*

blick über die Funktionsweise der wichtigsten Direct-Volume-Rendering-Verfahren geben, wie sie Marmitt, Friedrich und Slusallek in [MFS06] beschreiben. Die Qualität der erzeugten Bilder im Verhältnis zur Rendergeschwindigkeit der einzelnen Verfahren wird in [MHBMC00] näher diskutiert.

2.1.1 Volume Raycasting

Volume Raycasting basiert auf der Idee des *Raytracing* (Strahlverfolgung). Dabei wird der physikalische Prozeß des Lichttransports stark vereinfacht nachempfunden. Die Funktionsweise ähnelt prinzipiell der einer Lochkamera (siehe Abbildung 2.2). Beim Raytracing-/casting wird die Strahlrichtung allerdings umgekehrt, d.h. die Strahlen werden vom Augpunkt (entspricht dem Loch einer Lochkamera) aus durch die Bildebene in die Szene geschickt („Sehstrahlen“). So müssen nur die Strahlen berechnet werden, die auch tatsächlich zum Bild beitragen (siehe Abbildung 2.3).

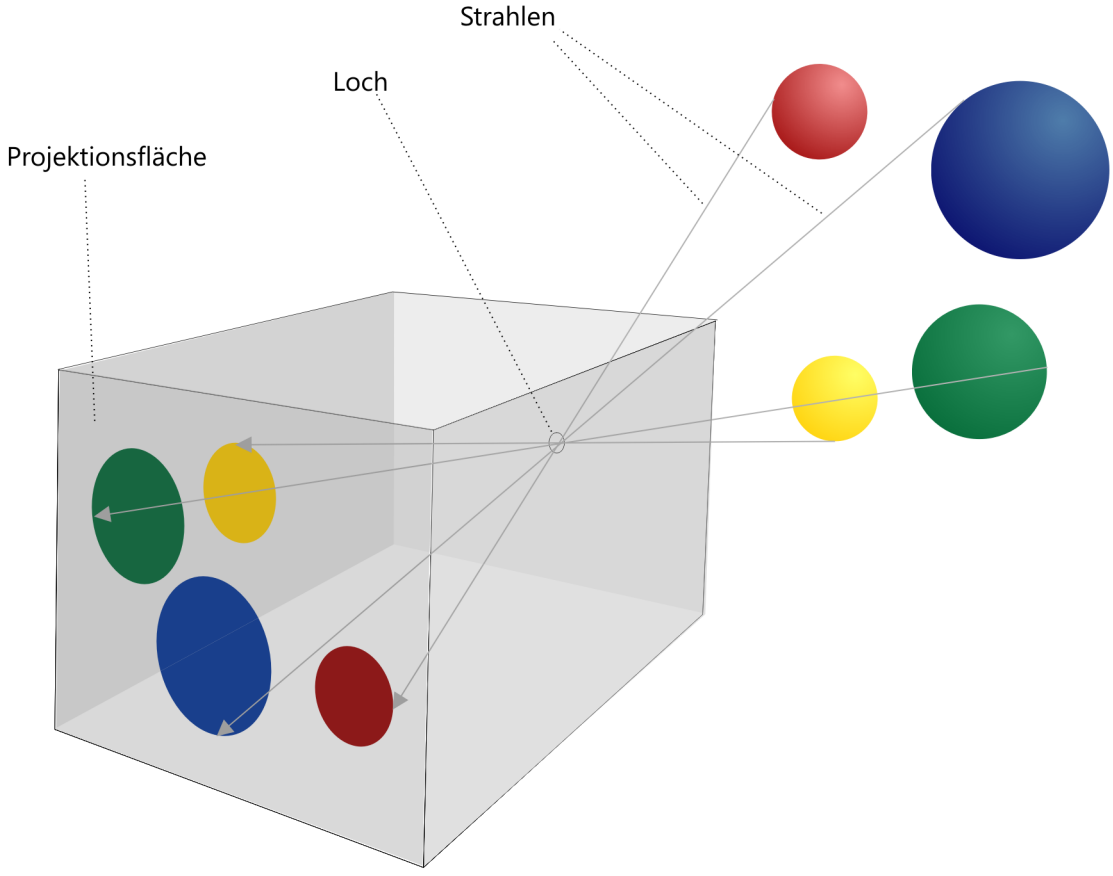


Abbildung 2.2: Funktionsprinzip einer Camera obscura (Lochkamera)
Das von den Objekten der Szene reflektierte oder emittierte Licht (Strahlen) fällt durch das Loch in das Innere der Kamera und erzeugt ein Bild auf der Rückwand (Projektionsfläche).

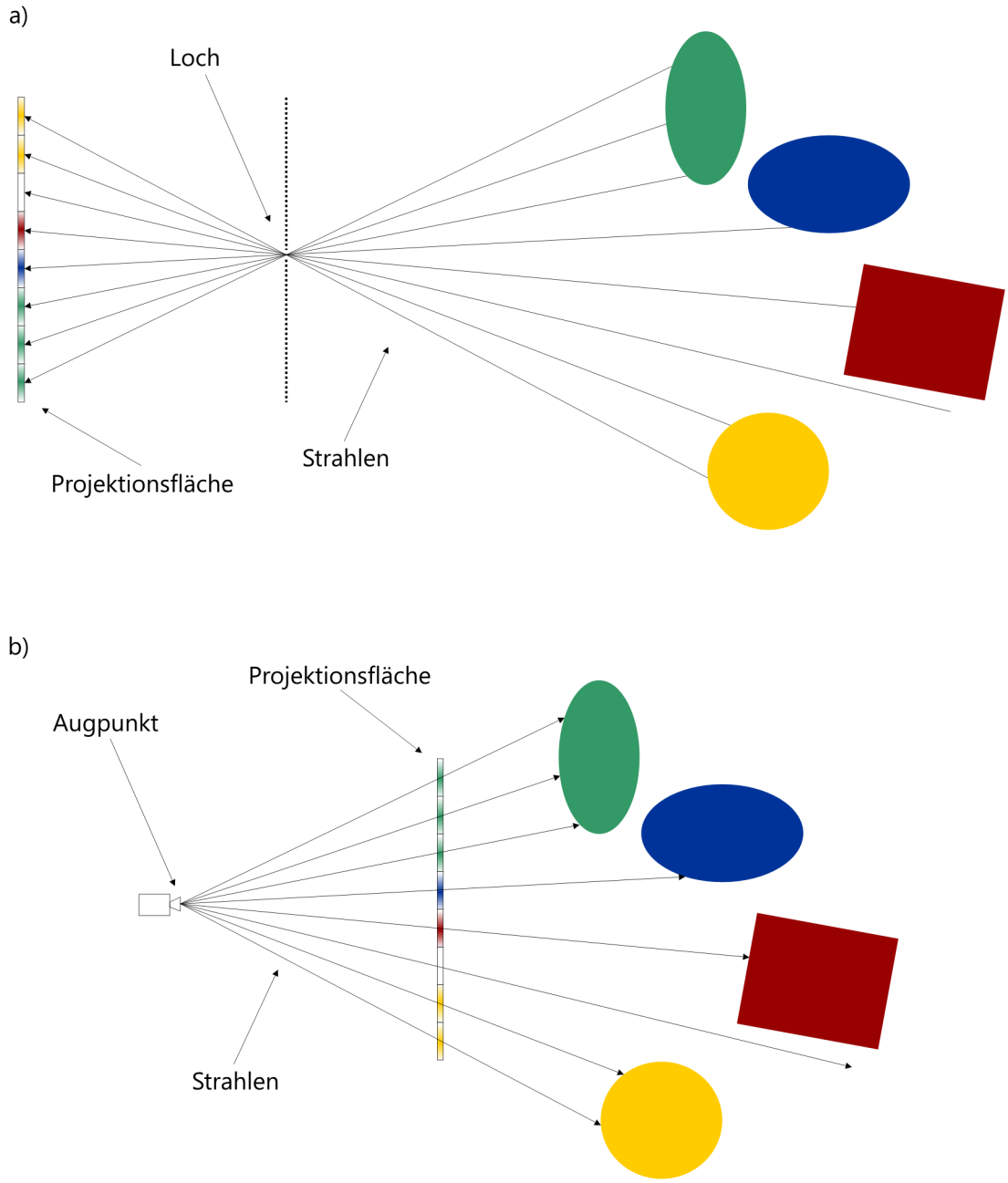


Abbildung 2.3: Prinzip des Raytracing
(a) zeigt das Funktionsprinzip einer Lochkamera, (b) die Grundidee des Raytracing: Strahlen verfolgt man hier vom Fokuspunkt (Augpunkt, entspricht dem Loch der Lochkamera) aus durch eine virtuelle Bildebene.

Nun muß der zum Strahlenursprung – also dem Betrachter, bzw. dem Augpunkt – nächstgelegene Schnittpunkt des jeweiligen Strahls mit einem Szenenobjekt (*Primitiv*) bestimmt werden. Abhängig von den Materialeigenschaften des so ermittelten Objekts und dem gewählten lokalen Beleuchtungsmodell, kann der

Farbwert für den Bildpunkt berechnet werden, der dem Schnittpunkt des ausgesendeten Strahls mit der Bildebene entspricht. Da als physikalisches Modell die seit der Antike bekannte geometrische Optik dient, werden Wellen- und Quanteneffekte vollständig vernachlässigt.

Im Gegensatz zu dieser einfachen Form des *Raycasting*, werden beim sog. *rekursiven Raytracing* neben den Primärstrahlen (*Primary Rays*) zusätzlich auch Sekundärstrahlen (*Secondary Rays*) und Schattenstrahlen (*Shadow Rays*) erzeugt, die ihrerseits mit Primitiven wechselwirken und so die Simulation komplexerer optischer Phänomene wie Reflexion, Brechung und Schattenwurf ermöglichen.

Um mit dieser Technik Volumendaten visualisieren zu können, muß der grundlegende Raycasting-Algorithmus geringfügig modifiziert werden, das Oberflächen-Beleuchtungsmodell wird durch ein Energieabsorptionsmodell ersetzt. Strahlen werden wie oben beschrieben vom Betrachter aus in die Szene gesendet. Liegen die Eingabedaten als Skalarfeld vor, z. B. in Form von CT-Schichtaufnahmen, werden entlang des Strahls aus den vorliegenden Samples, auch *Voxel* genannt, mittels trilinearer Interpolation Zwischenwerte ermittelt (*Resampling*), so daß aus diesen im *Compositing* genannten Verarbeitungsschritt die Dämpfung approximiert werden kann, die der Strahl erfährt, während er das Volumen durchquert (siehe Abbildung 2.4).

Im Vergleich zu anderen Volume-Rendering-Verfahren liefert Volume Raycasting die qualitativ hochwertigsten Ergebnisse und kommt den physikalischen Vorgängen am nächsten (vgl. Abschnitt 2.2). Volume Raycasting ist zwar ein relativ aufwendiges Verfahren, läßt sich andererseits jedoch sehr leicht parallelisieren, da die einzelnen Strahlen voneinander unabhängig sind.

Die zunehmende Programmierbarkeit moderner Graphikprozessoren ermöglicht es mittlerweile, Volume Raycasting auch hardwarebeschleunigt durchzuführen. Unter

http://www.vis.uni-stuttgart.de/ger/research/fields/current/spvolren

kann der Quelltext eines auf allen Grafikkarten, die Pixel-Shading gemäß Shader Model 3.0 oder höher unterstützen, lauffähigen Volume Rendering Framework bezogen werden [SSKE05].

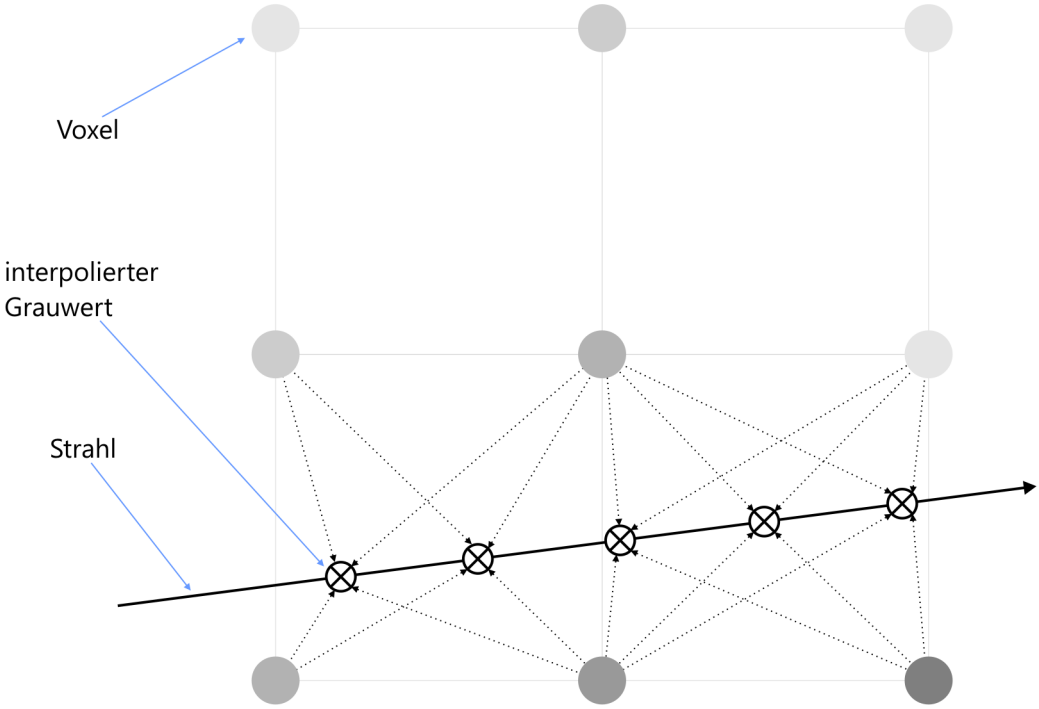


Abbildung 2.4: Volume Raycasting
Entlang des Strahls interpoliert man in regelmäßigen Abständen die Grauwerte aus den umliegenden Voxeln, die hier als Kreisscheiben dargestellt sind (Resampling). Die ermittelten Werte kombiniert man dann zu einem Grauwert (Compositing).

2.1.2 Splatting

Splatting gehört zu den Projektionsmethoden. Die Idee dahinter ist, Voxel auf die Bildebene zu „werfen“ und sie dort „zerplatzen“ zu lassen (Daher auch der Name der Methode[11]). Da die einzelnen Samples eines Volumendatensatzes nur für diskrete Punkte definiert sind, ordnet man den Samples zunächst Basisfunktionen bzw. Rekonstruktionskernel, in der Regel Gauß-Kerne, zu, um eine Annäherung für die Energieverteilung in der Umgebung des Samples zu erhalten. Die Basisfunktion wird mit dem jeweiligen Wert des Samples skaliert. Um die skalierten Basisfunktionen effizient auf die Bildebene projizieren zu können, werden die Basisfunktionen in einem Vorverarbeitungsschritt zunächst entlang der Betrachtungsrichtung integriert und der so gewonnene „Fußabdruck“ (*Footprint*) in einer Look-up-Tabelle gespeichert. Anschließend können die Werte aufsummiert und rasterisiert werden (siehe Abbildung 2.5).

11 engl. „splat“ bedeutet soviel wie „platsch“ – das Geräusch, das entsteht, wenn man „Farbbehälter“ (Voxel) gegen eine „Wand“ (Bildebene) wirft.

Mit fortgeschrittenen Splatting-Techniken wie EWA-Splatting [ZPBG01], kann eine ähnliche Bildqualität wie mit Raycasting erreicht werden. Perspektivische Projektionen sind jedoch etwas umständlicher zu realisieren, da hierfür die Basisfunktionen entsprechend transformiert werden müssen.

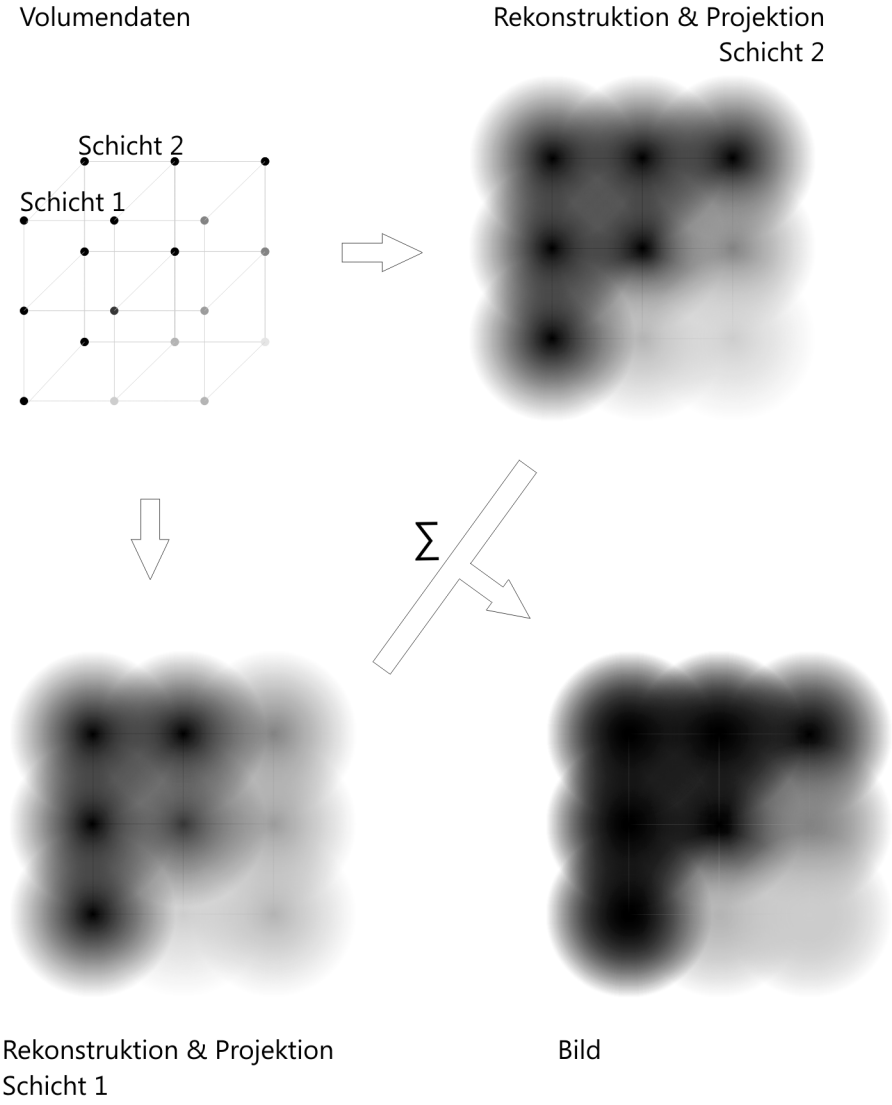


Abbildung 2.5: Splatting
Zunächst rekonstruiert man das Signal, indem man jedem Voxel des Volumendatensatzes eine entsprechend skalierte Basisfunktion zuordnet. Diese integriert man entlang der Betrachtungsrichtung und erhält so für jedes Voxel eine Projektion, die man Aufsummiert, um das endgültige Bild zu erhalten.

2.1.3 Shear-Warp-Faktorisierung

Der *Shear-Warp-Faktorisierung* liegt die Idee zugrunde, die Projektionsmatrix in einen 3D-Scher- und einen 2D-Warp-Anteil zu zerlegen. Zur Projektion und eigentlichen Bilderzeugung kommt auch hier der Raycasting-Algorithmus zum Einsatz.

Die Scher-Transformation richtet die Schichten senkrecht zur Blickrichtung aus. Im Gegensatz zum normalen Volume Raycasting, reicht daher zum Resampling eine bilineare Interpolation aus den Voxel-Daten jeweils einer Schicht aus. Nachteilig an dieser Vorgehensweise ist jedoch, daß die Sample-Rate entlang des Strahls somit von der Blickrichtung abhängig ist und als Folge Aliasing-Artefakte bei Unterschreiten der Nyquist-Frequenz auftreten können. Das so erzeugte, verzerrte Zwischenbild muß im Anschluß noch entzerrt und auf die Bildebene projiziert werden (*Warping*, siehe Abbildung 2.6).

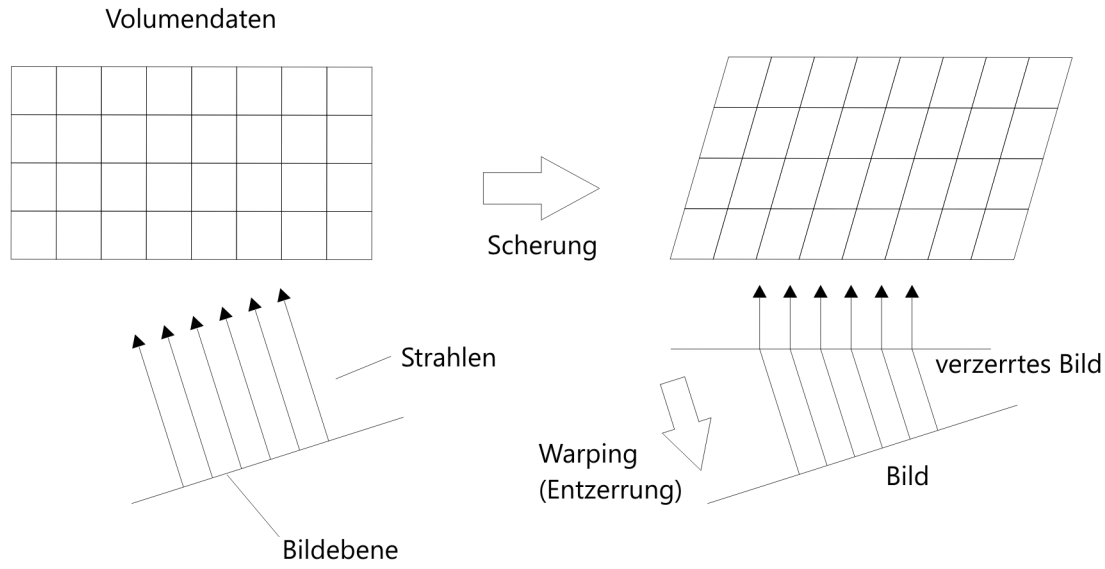


Abbildung 2.6: Shear-Warp-Faktorisierung

Shear-Warp gehört zu den schnellsten Möglichkeiten, Volumendaten zu visualisieren. Das Layout der Datenstrukturen erlaubt ein effektives Überspringen von opaken oder transparenten Regionen. Für eine perspektivische Projektion ist zusätzlich für jede Schicht eine unterschiedliche Skalierung notwendig. Weitere Details zu den Hintergründen der Shear-Warp-Faktorisierung enthält der Artikel [LL94], in dem Lacroute 1994 den Algorithmus vorstellte.

2.1.4 Texture Mapping

Texture Mapping ist eine Hardware-basierte Lösung, die auf die Fähigkeiten moderner GPUs[12] setzt. Es sind zwei Ansätze zu unterscheiden [Käh05]:

Beim *2D-Texture-Mapping* erzeugt man aus dem Volumendatensatz zunächst drei Sätze von Slices, die jeweils parallel zu einer der Koordinatenebenen ausgerichtet sind. Für jedes zu berechnende Pixel werden die Slices eines Stapels ineinandergeblendet, die am „senkrechtesten" zur Blickrichtung stehen. Innerhalb der Slices wird bilinear interpoliert. Das Blending erfolgt dabei back-to-front, ansonsten ist ein α-Accumulation-Buffer notwendig, der die α-Werte zwischenspeichert [KW03].

2D-Texture-Mapping ist sehr schnell und kann von jeder GPU durchgeführt werden, die Texture Mapping beherrscht. Zudem ist lediglich eine bilineare Interpolation nötig. Allerdings kann es bei einer Änderung der Blickrichtung durch den abrupten Wechsel des Slice-Satzes zu Artefakten kommen.

3D-Texture-Mapping modernerer Grafikhardware erlaubt es, Volumendaten direkt von der Hardware verarbeiten zu lassen. Auf entsprechend leistungsfähiger Hardware sind durchaus interaktive Frameraten zu erzielen. Der Datensatz wird als 3D-Textur in den lokalen Grafikspeicher der Grafikkarte geladen. Die GPU errechnet daraus on-the-fly zur Blickrichtung senkrechtstehende Slices. Beim Sampling wird trilineare Interpolation eingesetzt.

Durch die automatische Generierung von *Viewport-aligned* Slices, werden Artefakte durch einen plötzlichen Wechsel von einem Slice-Satz zum nächsten, wie sie beim 2D-Texture-Mapping auftreten, vermieden. Bei perspektivischer Projektion allerdings ist die Samplingrate nicht konstant (vgl. Abbildung 2.7). Bei nicht übertrieben groß gewähltem FOV[13] fällt dies jedoch kaum auf.

Die maximal sinnvolle Größe und Auflösung der visualisierten Volumen wird von der Größe des Grafikspeichers und den unterstützten Texturformaten begrenzt.

Die Qualität der erzeugten Bilder ist vor allem von der Auflösung des Framebuffers abhängig. Die lange Zeit gängigen 8-Bit-Framebuffer lösen maximal 256 Graustufen auf – zu wenig für professionelle Anwendungen. Es dürfte allerdings nur noch eine Frage der Zeit sein, bis Floating-Point-Framebuffer zum Standard werden. NVIDIA bietet z. B. bereits seit 2004/2005 mit dem NV40 Unterstützung für einen 64-Bit-Framebuffer; jede Farbkomponente[14] wird durch einen 16-

12 GPU steht für *Graphics Processing Unit*

13 FOV: *Field of View*

14 Jedes Pixel wird durch die drei Farbkomponenten Rot, Grün und Blau, sowie einem Alpha-Wert, der die Opazität bestimmt, beschrieben (RGBA).

Bit-Floating-Point-Wert repräsentiert. Die aktuelle Generation (G80) erlaubt bereits Floating-Point-Werte einfacher Genauigkeit (32 Bit) im Framebuffer.

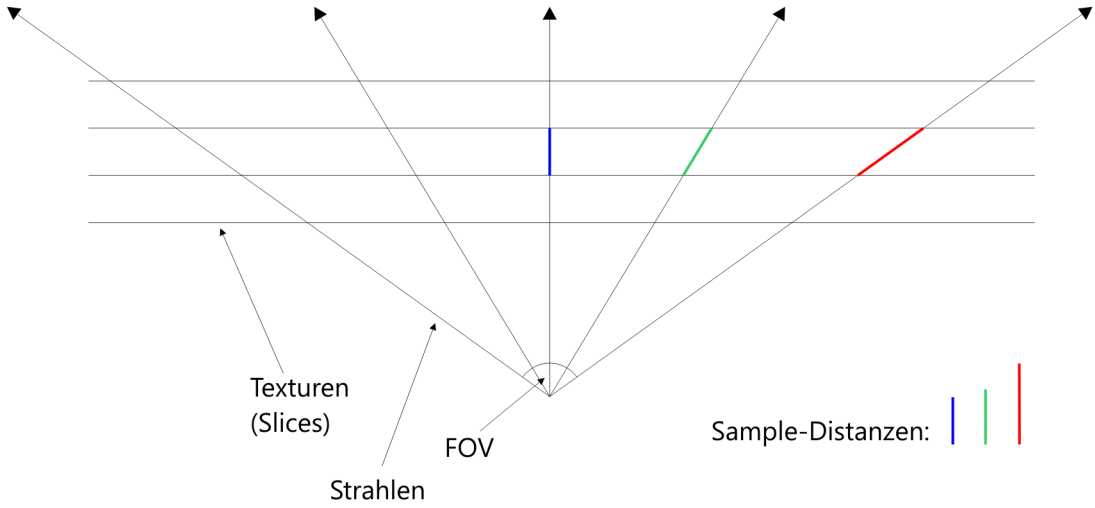


Abbildung 2.7: Abhängigkeit der Samplingrate von der Strahlrichtung (Texture Mapping)

2.2 Physikalische Grundlagen und Optische Modelle

Um einen Röntgensimulator entwerfen zu können, ist es zunächst notwendig, zu verstehen, welche physikalischen Effekte für die Absorption hochfrequenter elektromagnetischer Strahlung durch Materie verantwortlich sind. Auf dieser Grundlage lassen sich dann abstraktere optische Modelle entwerfen, die den Lichttransport innerhalb des zu visualisierenden Volumens allgemein beschreiben und als theoretisches Fundament für die Implementierung dienen können.

2.2.1 Absorption von Röntgenstrahlung

Die Photonenenergie von medizinisch genutzter Röntgenstrahlung liegt im Bereich von etwa 20 – 150 keV [KS88]. Gemäß dem Zusammenhang zwischen der Energie E, dem Planckschen Wirkungsquantum h und der Frequenz f eines Photons

$$E = hf \tag{2.1}$$

entspricht dieser Bereich Frequenzen in der Größenordnung von $4{,}8 \cdot 10^{18}$ bis $3{,}6 \cdot 10^{19}$ Hz. Photonen dieses Frequenzbereiches wechselwirken hauptsächlich auf zwei Arten mit Materie [KS88].

Bei der *Photoionisation*, auch als (atomarer) *photoelektrischer Effekt* bezeichnet, überträgt ein Röntgen-Photon seine Energie auf ein Elektron mit einer Bindungsenergie kleiner oder gleich der Energie des Photons. Gemäß der Einstein-Beziehung[15] [HH94], ist die kinetische Energie des so entstandenen Photoelektrons gleich der Differenz seiner Bindungsenergie und der Energie des absorbierten Röntgenphotons:

$$E_{\text{kin}} = \frac{1}{2} m v^2 = hf - E_{\text{B}} \tag{2.2}$$

Hierbei steht E_{B} für die Bindungsenergie des ionisierten Elektrons. Bei ausreichender kinetischer Energie, kann das Photoelektron das Atom verlassen und die so entstandene „Lücke“ wird durch ein Elektron mit höherer Bindungsenergie aufgefüllt. Die Energiedifferenz wird in Form eines niederenergetischen Photons abgegeben (*Fluoreszenz*).

Im Falle der *Compton-Streuung*, wechselwirkt ein Photon mit freien oder schwach gebundenen Elektronen [WP07]. Dabei ändern sich Richtung und Energie des gestreuten Photons [HH94]:

$$\Delta\lambda = \frac{h}{m_{\text{e}} c}(1 - \cos\theta) = \lambda_{\text{c}}(1 - \cos\theta) \tag{2.3}$$

λ_{C} ist die *Compton-Wellenlänge* eines Elektrons, m_{e} die Ruheenergie des Elektrons und $\Delta\lambda$ die Differenz der Wellenlängen des Photons vor und nach der Streuung. θ ist der Streuwinkel. Die Auswirkungen des Compton-Effekts sind erst für sehr hohe Frequenzen meßbar, da die Compton-Wellenlänge im niedrigen Picometer-Bereich liegt und $\Delta\lambda$ daher selbst für 180°-Streuwinkel entsprechend klein ist.

Für die Absorption monochromatischer Röntgen- und γ-Strahlen gilt, wie auch für sichtbares Licht, das *Lambert-Beersche-Gesetz* [PiN99], auch einfach Absorptionsgesetz genannt:

$$I(s) = I_0 \mathrm{e}^{-\mu s} \tag{2.4}$$

15 Albert Einstein erhielt für diese Erklährung des Photoeffekts 1921 den Nobelpreis für Physik.

I_0 ist die Intensität der in das Medium eingekoppelten Strahlung, $I(s)$ die Intensität der Strahlung nach einer im Medium zurückgelegten Strecke s. μ ist der material- und frequenzabhängige *Absorptions-* oder *Extinktionskoeffizient*. Da sowohl der Photoelektrische Effekt als auch die Compton-Streuung Wechselwirkungen mit Elektronen des Mediums darstellen, hängt der lineare Absorptionskoeffizient μ von der Energie (Wellenlänge λ) der Strahlung und der Zahl der Elektronen pro Volumeneinheit im Medium ab. Es gilt [PiN99]:

$$\mu \propto \frac{Z^4}{(hf)^3} \tag{2.5}$$

Die Ordnungszahl Z dient zur Abschätzung der Zahl der Elektronen im Medium. Hieran wird auch sofort klar, weshalb Knochen auf Röntgenaufnahmen besonders gut sichtbar ist: Während sich das umliegenden Gewebe vorwiegend aus Wasserstoff ($Z = 1$), Kohlenstoff ($Z = 6$) und Sauerstoff ($Z = 8$) zusammensetzt, besteht Knochen zu weiten Teilen aus Kalzium mit $Z = 20$.

Der lineare Absorptionskoeffizient μ ist nicht direkt meßbar. In der Literatur findet man daher stattdessen Werte für den *Masse-Absorptionskoeffizienten* μ_ρ, der von der Dichte ρ des Mediums sowie der Röntgenenergie abhängt [HS96]:

$$\mu_\rho = \frac{\mu}{\rho} \tag{2.6}$$

2.2.2 Das Volume-Rendering-Integral

Die *Volume-Rendering-Gleichung* [Kel06, Ish99]

$$I(s) = I_0 \exp\left(-\int_0^s \mu(t)\,\mathrm{d}t\right) + \int_0^s L(t) \exp\left(-\int_t^s \mu(l)\,\mathrm{d}l\right)\mathrm{d}t \tag{2.7}$$

beschreibt den Lichttransport in einem Medium entlang der Bahnkoordinate s (t und l: Substitutionen der Bahnkoordinate). Eine ausführliche Herleitung der Gleichung findet sich z. B. in [Mor04]. Das Medium, durch das sich das Licht bewegt, wird in diesem Modell als ein Volumen, gefüllt mit Partikeln, die Licht sowohl absorbieren als auch emittieren (z. B. durch diffuse Streuung), abstrahiert. Diese Integralgleichung ist in der Regel nicht analytisch lösbar und wird daher in der Praxis numerisch approximiert.

Eine erhebliche Vereinfachung erreicht man, indem man davon ausgeht, daß die Partikel Licht lediglich absorbieren, aber selbst kein Licht emittieren oder streuen. Der Lichttransport läßt sich dann folgendermaßen beschreiben:

$$I(s) = I_0 \exp\left(-\int_0^s \mu(t)\,\mathrm{d}t\right) \tag{2.8}$$

Dieses Modell entspricht dem Absorptionsterm in (2.7) bzw. dem Absorptionsgesetz (2.4), mit dem Unterschied, daß der Absorptionskoeffizient μ nun nicht mehr konstant sein muß.

Nimmt man hingegen an, daß die Partikel überwiegend Licht emittieren und der Absorptionsanteil vernachlässigbar klein ist (Beispiel: heißes Gas oder ein Plasma), bleibt nur der Emissions-Anteil aus (2.7) erhalten:

$$I(s) = I_0 + \int_0^s L(t)\,\mathrm{d}t \tag{2.9}$$

Problematisch an diesem Modell ist, daß $I(s)$ durch den Wegfall des Absorptionsterms nicht mehr nach oben beschränkt ist. Nach dem Absorptionsmodell (2.8) hingegen, liegt $I(s)$ immer zwischen I_0 und 0.

2.3 Lösungsansatz

In diesem Abschnitt möchte ich den grundlegenden Lösungsansatz für die Umsetzung des Röntgen-Simulators skizzieren. Ich verzichte bewußt auf einen zu hohen Detailgrad. Eine Diskussion ausgewählter interessanter Einzelaspekte der Implementierung findet in Abschnitt 2.4 statt.

2.3.1 Auswahl eines optischen Modells

Nachdem wir in 2.2 verschiedene Methoden der Volumenvisualisierung und deren theoretische Grundlagen kennengelernt haben, stellt sich die Frage, welche davon die Anforderungen an die Implementierung am besten erfüllt.

Für die Implementierung des Röntgen-Simulators, ist das reine Absorptionsmodell (2.8) das vorzuziehende Modell. Es birgt mehrere Vorteile.

Zum einen ist es das mathematisch am einfachsten handhabbare Modell; Diskretisierung und stückweise Integration sind sehr leicht umsetzbar. Auch striktes

back-to-front- oder front-to-back-Rendering ist unnötig, eine Sortierung der Polyeder ist daher ebenfalls nicht erforderlich (siehe 2.3.2).

Zum anderen beschreibt das Modell die physikalischen Vorgänge beim Anfertigen einer Röntgenaufnahme ausreichend genau [Max95]. Ein CT- oder Röntgen-Bild zeigt die Anzahl (Intensität) der Röntgen-Photonen, die den Detektor bzw. den Röntgen-Film erreichen. Da organisches Gewebe selbst keine Röntgen-Photonen emittiert, kann der Emissionsanteil vernachlässigt werden.

Die Restintensität $I(s_1)$ eines Strahls, nachdem er ein einzelnes finites Element (Strecke s_1) passiert hat, kann nach (2.8) folgendermaßen beschrieben werden:

$$I(s_1)=I_0\exp\left(-\int_0^{s_1}\mu(t)\,\mathrm{d}t\right) \tag{2.10}$$

Dabei ist zu beachten, daß wir an diesem Punkt noch keine speziellen Annahmen über $\mu(t)$ anstellen, insbesondere setzen wir an diesem Punkt noch keinen konstanten Absorptionskoeffizienten innerhalb eines finiten Elements voraus.

Durchquert er ein weiteres finites Element (Strecke $s_2 - s_1$), nimmt die Intensität entsprechend weiter ab:

$$\begin{aligned} I(s_2)&=I(s_1)\exp\left(-\int_{s_1}^{s_2}\mu(t)\,\mathrm{d}t\right)= \\ &=I_0\exp\left(-\int_0^{s_1}\mu(t)\,\mathrm{d}t\right)\exp\left(-\int_{s_1}^{s_2}\mu(t)\,\mathrm{d}t\right) \end{aligned} \tag{2.11}$$

Führt man dies rekursiv fort, gilt für die Intensität des Strahls, nachdem er das gesamte Volumen, also alle Elemente n, durchlaufen hat:

$$I(s_n)=I_0\prod_{i=0}^{n-1}\exp\left(-\int_{s_i}^{s_{i+1}}\mu(t)\,\mathrm{d}t\right) \tag{2.12}$$

2.3.2 Das Rendering-Verfahren

Ich habe mich aus mehreren Gründen für Volume Raycasting als Rendering-Verfahren für den Röntgen-Simulator entschieden. Das Verfahren ist seit Jahrzehnten Gegenstand aktiver Forschung, so daß inzwischen diverse Optimierungsverfahren bekannt sind; es liefert bestmögliche Bildqualität und ist gleichzeitig vergleichsweise einfach zu implementieren. Besonders positiv wirkt sich aus, daß das

Verfahren im Idealfall logarithmisch mit der Komplexität der Szene skaliert, sofern man die notwendigen Beschleunigungsdatenstrukturen einsetzt. Bedenkt man, das ein FE-Modell mittlerer Auflösung bereits aus über 100 000 Hexaeder-Elementen besteht und damit mehr als 1,2 Mio. Dreiecke pro Szene zu verarbeiten sind, ist diese Eigenschaft ein nicht zu unterschätzender Vorteil.

Grundsätzlich bieten sich zwei alternative Vorgehensweisen an: Möchte man „klassisches" Volume Raycasting betreiben, könnte man die Schwerpunkte der Hexaeder als Sample-Punkte interpretieren. Ordnet man diesen die jeweiligen Knochendichte-Werte zu, erhält man ein Voxel-Gitter, also ein diskretes, dreidimensionales Skalarfeld. Zum Resampling der Werte entlang eines Strahls, müßten Zwischenwerte wie üblich mittels Interpolation bestimmt werden (vgl. Abbildung 2.4).

Der andere Ansatz arbeitet direkt mit der Hexaeder-Geometrie. Vereinfacht geht man folgendermaßen vor (Abbildung 2.8): Jeder Strahl wird darauf geprüft, welche Hexaeder er schneidet. Für jedes geschnittene Hexaeder wird der vom Strahl darin zurückgelegte Weg aus den Schnittpunkten (Ein- und Austrittspunkt) berechnet und mit der jeweiligen Knochendichte gewichtet. Daraus läßt sich mit (2.12) berechnen, welche Restenergie der gedämpfte Strahl nach Durchqueren des Gitters besitzt. Die Knochenkonzentration ist also in diesem Modell an allen Punkten definiert und wird innerhalb der Hexaeder als konstant angenommen.

Für den zweiten Ansatz spricht, daß auch die Knochenheilungssimulation mit stückweise konstanten Knochenkonzentrationen arbeitet (siehe 1.2.2). Außerdem kann so die Genauigkeit der Modellgeometrie bestmöglich ausgenutzt werden, eine Interpolation von Werten ist nicht notwendig. Der Röntgensimulator arbeitet daher nach dieser „geometrischen Methode".

Wendet man diese Überlegungen auf (2.12) an, folgt unmittelbar:

$$I_n = I_0 \prod_{i=1}^{n} \exp\left(-\mu_i s_i\right) = I_0 \exp\left(-\sum_{i=1}^{n} \mu_i s_i\right) \tag{2.13}$$

μ_i ist hier der innerhalb von Element i geltende, konstante lineare Absorptionskoeffizient, s_i die innerhalb dieses Elements zurückgelegte Strecke. Die Reihenfolge der Elemente spielt bei der Berechnung der Intensität offensichtlich keine Rolle[16].

16 Kommutativität von Addition und Multiplikation

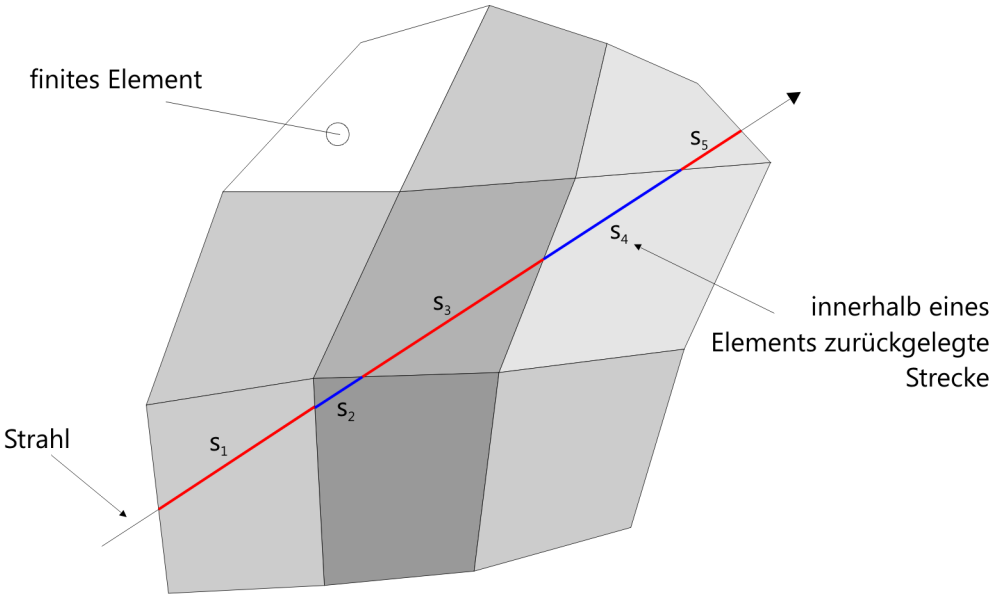


Abbildung 2.8: Semi-transparentes Volume Rendering von finiten Elementen mittels Raycasting/-tracing
Zur Berechnung der Restenergie eines Strahls nach dem Durchqueren eines Volumens berechnet man, welche Teilstrecke der Strahl innerhalb jedes finiten Elements zurücklegt (Strecken s_1 bis s_5). Diese gewichtet man mit der innerhalb des jeweiligen Elements geltenden konstanten Knochenkonzentration (hier durch Grautöne symbolisiert).

Diese Beziehung kann jedoch auch noch auf direkterem Wege erschlossen werden: Interpretiert man $\mu(t)$ als „Absorptionsfunktion“, die den Verlauf des Absorptionskoeffizienten entlang des gesamten Strahls – also nicht getrennt für jedes Element – beschreibt, die man elementeweise integriert, und legt man eine stückweise konstante Knochenkonzentration zugrunde, folgt aus (2.8) auch sofort (2.13):

$$I(s)=I_0\exp\left(-\int_0^s \mu(t)\,\mathrm{d}\,t\right)=I_0\exp\left(-\sum_{i=0}^{n-1}\int_{s_i}^{s_{i+1}} \mu(t)\,\mathrm{d}\,t\right)= \\ =I_0\exp\left(-\sum_{i=1}^{n}\mu_i s_i\right)=I_n \tag{2.14}$$

2.3.3 Berechnung des Absorptionskoeffizienten

Die Knochenheilungssimulation berechnet für jedes Element die relative Knochenkonzentration, also welchen Anteil das Knochengewebe am durch ein Element repräsentierten Volumen besitzt. Um (2.13) berechnen zu können, muß aus

diesem Wert der Absorptionskoeffizient des jeweiligen Elements hergeleitet werden.

Um den Zusammenhang zwischen der Knochenkonzentration und dem Absorptionskoeffizienten verstehen zu können, müssen wir das physikalische Modell, das dem aus dem Volume-Rendering-Integral hergeleitete Absorptionsmodell (2.8) zugrunde liegt, etwas näher betrachten [Max95, Mor04]:

Das von einem Strahl durchdrungene Volumen wird als dünner Zylinder, der mit Partikeln gefüllt ist, dargestellt. Die Partikeldichte ρ_P, also die Zahl der absorbierenden Teilchen pro Volumeneinheit, variiert nur über die Länge des Zylinders. Um die Dämpfung des Strahls entlang des Zylinders untersuchen zu können, unterteilt man den Zylinder in dünne Scheiben der Länge Δs. Für einen Zylinder mit Grundfläche A enthält eine Scheibe im Mittel $N = \Delta s A \rho_P$ Teilchen (Abbildung 2.9).

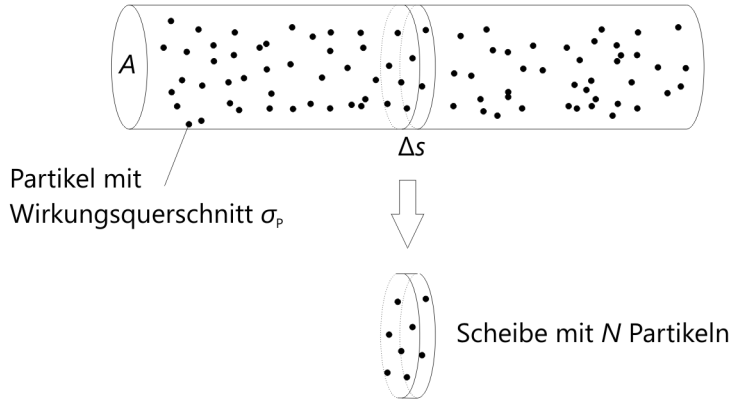


Abbildung 2.9: Partikelmodell eines Volumens

Beträgt der Wirkungsquerschnitt der einzelnen Partikel σ_P und geht die Dicke der Scheiben Δs gegen 0, so daß sich die Wirkungsquerschnitte der enthaltenen Partikel nicht mehr überlappen können, gilt für den Wirkungsquerschnitt einer Scheibe σ_{Scheibe}, also für die von Partikeln bedeckte Teilfläche einer Scheibe, somit:

$$\sigma_{\text{Scheibe}} = N \sigma_{\text{P}} \tag{2.15}$$

Wenn σ_{Scheibe} dem n-ten Teil des Zylinderquerschnitts entspricht, bedeutet dies auch, daß dieser Anteil des Lichts beim Durchqueren einer Scheibe der Dicke Δs absorbiert wird, da ein entsprechender Anteil der Photonen auf die in der Scheibe enthaltenen Partikel trifft:

$$Anteil_{\text{absorbiert}} = \frac{I_0 - I(\Delta s)}{I_0} = \frac{1}{n} = \frac{\sigma_{\text{Scheibe}}}{A} = \sigma_{\text{P}}\ \rho_{\text{P}}\ \Delta s \tag{2.16}$$

Der Anteil des absorbierten Lichts pro Längeneinheit beträgt folglich

$$\mu = \sigma_P \rho_P \ . \tag{2.17}$$

Dies entspricht auch der Definition des Extinktionskoeffizienten gemäß dem Lambert-Beerschen-Gesetz [WoP07]. Die Dichte ρ des Mediums hängt von der Teilchendichte ρ_P und der Masse eines Partikels m_P ab:

$$\rho = m_P \rho_P \tag{2.18}$$

Daraus folgt:

$$\mu \propto \rho_P \ \wedge \ \rho_P \propto \rho \ \Rightarrow \ \mu \propto \rho \tag{2.19}$$

Die Heilungssimulation beschreibt die Knochendichte als Konzentration des Gewebetyps Geflechtknochen mit der Dichte ρ_{Knochen}. Beträgt die relative Knochenkonzentration eines finiten Elements den Wert $C_{\text{Knochen}} \in [0 \ldots 1]$, bedeutet dies, daß der C_{Knochen}-te Teil des Volumens des Elements mit Geflechtknochen entsprechender Dichte gefüllt ist. Das restliche Volumen wird durch Bindegewebe oder Knorpel belegt; beide Gewebetypen werden jedoch bezüglich ihrer Röntgenschwächung vernachlässigt und das durch sie belegte Volumen als leerer Raum interpretiert.

Geht man nun noch davon aus, daß das Knochengewebe auf makroskopischer Skala im Mittel homogen über das Volumen verteilt ist, gilt für die mittlere Dichte des Knochengewebes innerhalb eines Elements mit Knochenkonzentration C_{Knochen}:

$$\rho_{FE} = \frac{m_{\text{Knochen}}}{V_{FE}} = \frac{C_{\text{Knochen}} \cdot V_{FE} \cdot \rho_{\text{Knochen}}}{V_{FE}} = C_{\text{Knochen}} \cdot \rho_{\text{Knochen}} \tag{2.20}$$

Da sich die Dichte ρ gemäß (2.20) linear zur Konzentration C_{Knochen} verhält und gemäß (2.19) μ proportional zu ρ ist, folgt somit für μ:

$$\mu \propto C_{\text{Knochen}} \tag{2.21}$$

μ verhält sich also direkt proportional zur Konzentration C_{Knochen} des absorbierenden Mediums.

2.3.4 Eingabedaten aus der Frakturheilungssimulation

Bei den zu visualisierenden Eingabedaten handelt es sich um FE-Modell von Knochenfrakturen, bestehend aus einem Gitter (Mesh) aus Hexaedern, die durch

ihre jeweils acht Eckpunkte (*Vertices*) definiert sind. Des weiteren ist für jedes dieser Hexaeder die relative Knochenkonzentration gegeben.

Um an die benötigten Modelldaten zu gelangen, bietet es sich an, die ANSYS--Scriptsprache *APDL*[17] zu verwenden. Da das Frakturheilungssimulationsprogramm zu weiten Teilen ebenfalls in dieser Sprache realisiert wurde, läßt sich eine entsprechende Export-Routine leicht in den Programmablauf integrieren. So werden zunächst zwei Textdateien erzeugt: eine listet die Koordinaten aller Vertices auf (im ANSYS-Jargon auch als „Nodes“ bezeichnet); eine weitere definiert die einzelnen Elemente bzw. Hexaeder, indem sie die Vertices aus der ersten Datei referenziert.

Nun wird noch die ermittelte relative Knochenkonzentration jedes Elements benötigt, um später die Absorption der Röntgenstrahlen berechnen zu können. Praktischerweise werden diese Daten bereits standardmäßig während jeder Iteration in eine Textdatei geschrieben, da sie als Input für den sich anschließenden Simulationsschritt notwendig sind (siehe 1.2.2).

Prinzipiell würden diese Daten ausreichen, um das Volumen rendern zu können. Allerdings läßt sich noch eine weitere Eigenheit des Simulationsprogramms nutzen: Die Heilungssimulation benötigt zwingend Informationen über die Nachbarschaftsbeziehungen der einzelnen Elemente zueinander. Auch diese Informationen werden standardmäßig in eine Textdatei ausgegeben, so daß sie vom externen Fuzzy-Logic-Controller verwendet werden können.

Wie oben beschrieben, werden die Hexaeder durch die Referenzierung von jeweils acht Vertices beschrieben. Schöner wäre es, wenn man die Seitenflächen der Hexaeder explizit exportieren könnte, um diese wiederum in der Hexaeder-Datei referenzieren zu können. Dadurch könnten die Seitenflächen, die mehr als ein Element begrenzen, automatisch wiederverwendet werden. Das würde zum einen Speicherplatz sparen, zum anderen erleichtert es das Traversieren des Volumens erheblich (siehe 2.5.4). Leider erlaubt ANSYS aber keinen direkten Zugriff auf die Vertexdaten der Seitenflächen. Um zu vermeiden, daß Seitenflächen, die zu zwei Elementen gehören doppelt angelegt werden müssen, was auch die Anzahl der notwendigen Schnittests erhöhen würde, rekonstruiert das Importfilter aus den Nachbarschaftsbeziehungen jedes Elements, an welche Seitenfläche des Elements welche Seitenfläche des jeweiligen Nachbarelementes grenzt (vgl. 2.4.4). Außerdem können die Informationen über die Nachbarschaftsbeziehungen der Elemente genutzt werden, um das Rendering erheblich zu beschleunigen (siehe 2.5.4).

Durch die Verwendung der bereits vorhandenen Nachbarschafsdaten, kann somit ein Vorverarbeitungsschritt, bei dem die jeweiligen Nachbarelemente aus den Vertexdaten ermittelt werden müßten, entfallen.

17 *APDL*: ANSYS Parametric Design Language

2.3.5 Bedienkonzept

Graphische Benutzerschnittstelle

Eine einfache *WinForms*-GUI erlaubt die intuitive Bedienung des Röntgensimulators (siehe Abbildung 2.10). Die Bedienelemente sollten weitgehend selbsterklärend sein: Im oberen Teil spezifiziert der Nutzer die Eingabedaten, die zur Generierung der Szene dienen sollen. Der mittlere Bereich erlaubt die Konfiguration verschiedener Kameraparameter, darunter auch die gewünschte Projektionsart. Links unten stellt man die gewünschte Auflösung ein. Außerdem kann man optional die Verwendung von Multithreading explizit deaktivieren.

Den eigentlichen Rendervorgang startet der Benutzer über den entsprechend beschrifteten Button. Der Button „Save image...“ erlaubt das Speichern des erzeugten Bildes unter einem beliebigen Namen.

Das „Trace Log“ zeigt standardmäßig Statusnachrichten und gegebenenfalls Fehlermeldungen an, so daß man den Fortschritt des Renderingvorgangs verfolgen kann. Ein Klick auf den Button „Hide log“ oder den Schließen-Button des Trace-Log-Fensters versteckt das Log-Fenster, ein weiterer Klick bringt das Fenster wieder zum Vorschein. Der Benutzer kann die von ihm vorgenommenen Einstellungen speichern, indem er auf den Button „Save settings“ klickt.

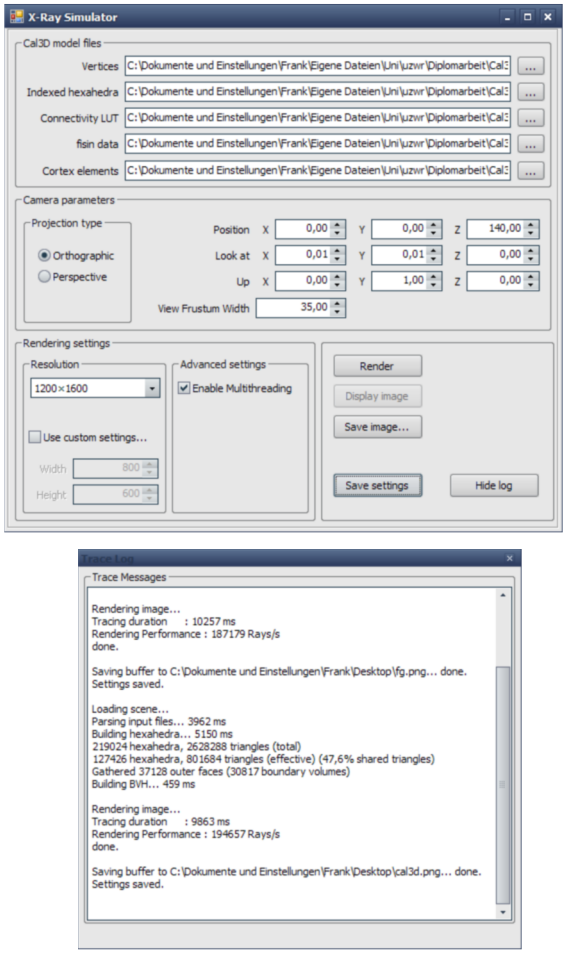


Abbildung 2.10: GUI des Röntgensimulators

Kommandozeilen-Interface

Ein Kommandozeilen-Interface erlaubt beispielsweise die scriptgesteuerte Erzeugung von Bildern mittels des Röntgensimulators. Zu allen Optionen der GUI kennt das CLI ein entsprechendes Pendant. Führt man das Programm ohne Parameter aus, informiert ein Hilfetext über die korrekte Bedienung des Röntgensimulators (siehe Abbildung 2.11).

Abbildung 2.11: Ausgabe des Kommandozeileninterface (CLI) bei Fehlbedienung

2.4 Implementierung

Ziel dieses Abschnittes ist es, einen genaueren Einblick in Design und Funktionsweise des Röntgensimulators zu geben. Dazu betrachten wir sowohl den grundlegenden Aufbau als auch interessante Einzelaspekte en Detail.

2.4.1 Zielplattformen

Die Software muß eine Vielzahl verschiedener Hard- und Softwareplattformen abdecken. Sowohl x86 und x64[18], als auch SPARC-CPUs sollten nach Möglichkeit unterstützt werden. Nicht weniger vielfältig ist die Liste der zu unterstützenden

18 x64 bezeichnet verschiedene Implementierungen von AMDs ursprünglich x86-64 genannter 64-Bit-ISA (AMD64, Intel 64)

Betriebssysteme: Das Programm muß zu Windows XP (in der x86- und x64-Variante), Linux, ebenfalls in beiden Varianten, sowie Solaris kompatibel sein.

Um diese Form der Kompatibilität zu erreichen, kann man nun entweder versuchen, möglichst hardware- und betriebssystemunabhängig zu programmieren, oder man paßt laufzeit-kritische Teile der Implementierung einzeln an die verschiedenen Plattformen an.

Eine weitere Möglichkeit besteht darin, eine zusätzliche Abstraktionsschicht zwischen Anwendung und Betriebssystem einzuziehen. Man führt letztendlich eine neue, rein softwarebasierte Plattform, eine sogenannte *Virtuelle Maschine (VM)*, ein, die als Laufzeitumgebung für Anwendungen dient, und diese vollständig vom darunter arbeitenden Betriebssystem und der Hardware abschirmt. Für die VM geschriebene und kompilierte Anwendungen sind auf allen Plattformen lauffähig, für die es eine Implementierung jener Laufzeitumgebung gibt.

Dieser Ansatz wird auch für den Röntgensimulator genutzt. Konkret kommt Microsofts *Common Language Infrastructure (CLI)*[19]-Architektur zum Einsatz. CLI-Laufzeitumgebungen (*Virtual Execution System*) existieren für diverse Betriebssysteme. Die wichtigsten sind zum einen Microsofts *Common Language Runtime (CLR)* für Windows, die als Bestandteil des .NET-Frameworks ausgeliefert wird, und zum anderen Novells *Mono*, das in Versionen für verschiedenste Unix-, Linux- und auch Windows-Versionen verfügbar ist.

2.4.2 Architekturüberblick

Der Aufbau des Röntgensimulators lehnt sich zum Teil an typischen Grundmustern üblicher Raytracer-Implementierungen an. Da jedoch nicht Oberflächen, sondern Volumen visualisiert werden müssen, ist eine spezielle Berücksichtigung von volumenbehafteten Objekten zumindest hilfreich, um so die Effizienz des Raycastings durch Ausnutzung bestimmter Eigenheiten der Eingabedaten verbessern zu können. Bei der Implementierung des Röntgensimulators habe ich mich bisher auf die Visualisierung von irregulären Hexaeder-Gittern beschränkt. Das Design wurde jedoch bewußt abstrakt gehalten, um eine problemlose Erweiterung, z. B. um eine Unterstützung von Tetraeder-Netzen oder evtl. auch anderer Raytracing/-casting-Methoden, zu ermöglichen. Sofern man ein entsprechendes Import-Plug-In schreibt, können problemlos auch Szenen gerendert werden, die keine oder nicht nur Daten aus der Frakturheilungssimulation enthalten.

Abstrakte Klassen, Interfaces und virtuelle Methoden verhelfen einer Architektur zwar zu hoher Flexibilität und ermöglichen ein elegantes Design. Andererseits wirkt sich ihr Einsatz meist negativ auf die Ausführungsgeschwindigkeit aus;

19 Standardisiert unter ECMA-335 und ISO/IEC 23271:2006

virtuelle Methoden können z. B. vom Compiler nicht (bzw. nur in bestimmten Ausnahmefällen) „geinlined“[20] werden. Das Design des Röntgensimulators ist daher letztlich ein Kompromiß zwischen Erweiterbarkeit und Effizienz. Die Klassenhierarchie sowie die Interaktion der einzelnen Klassen untereinander habe ich gegenüber frühen Prototypen teilweise deutlich vereinfacht, wobei mit Hilfe regelmäßig durchgeführter Mikrobenchmarks und Einsatz eines Profilers die Sinnhaftigkeit der jeweiligen Designentscheidung sichergestellt wurde.

Die folgenden Erläuterungen sollen lediglich einen kurzen Überblick über den Aufbau des Röntgensimulators geben (vgl. Abbildung 2.12) und wollen nicht als vollständige Dokumentation mißverstanden werden: Nur die Klassen, die für das Verständnis der in den Folgeabschnitten beschriebenen Abläufe zwingend notwendig sind, finden Erwähnung; andere lasse ich an dieser Stelle aus Gründen der Übersicht großzügig „unter den Tisch fallen“.

Primitive

Die abstrakte Klasse `Primitive` repräsentiert sämtliche darstellbaren Objekt-Typen, die eine Szene beinhalten kann. Durch einen Raycaster darstellbar sind alle Objekte, die von einem Strahl geschnitten werden können. Im wesentlichen definiert `Primitive` daher die abstrakte Funktion `Intersects(Ray, ref float)`, die den Abstand des Strahlenursprungs zum nächsten Schnittpunkt mit dem Primitiv als Gleitkommazahl einfacher Genauigkeit zurückliefert.

Die Methoden `GetMax/MinExtend()` dienen zum Erzeugen von *Bounding Volumes*. Jedes Primitiv verfügt außerdem über ein `Material`, das die physikalischen Eigenschaften des Primitivs definiert (Absorptionskoeffizient, diffuse Oberflächenfarbe, Reflektivität, etc.).

PlanarPrimitive

Diese Klasse leitet sich direkt von `Primitive` ab und ist die Basisklasse für alle planaren Szenen-Primitive, wie z. B. 2D-Ebenen oder Polygone, die kein Volumen besitzen. Jedem `PlanarPrimitive` ist genau eine Normale zugeordnet. Ein `PlanarPrimitive` zeichnet sich zudem dadurch aus, daß die darin durch einen Strahl zurückgelegte Strecke und somit auch die Absorptionsrate konstant 0 ist. Die Anzahl der möglichen Schnittpunkte ist 0 oder 1.

20 „Inlining“ bedeutet, daß der Compiler den vollständigen Code einer aufzurufenden Methode in die aufrufende Methode selbst kopiert. Dadurch vermeidet man sonst notwendige unbedingte Sprünge sowie diverse Stack-Operationen, was die Ausführungseffizienz verbessert.

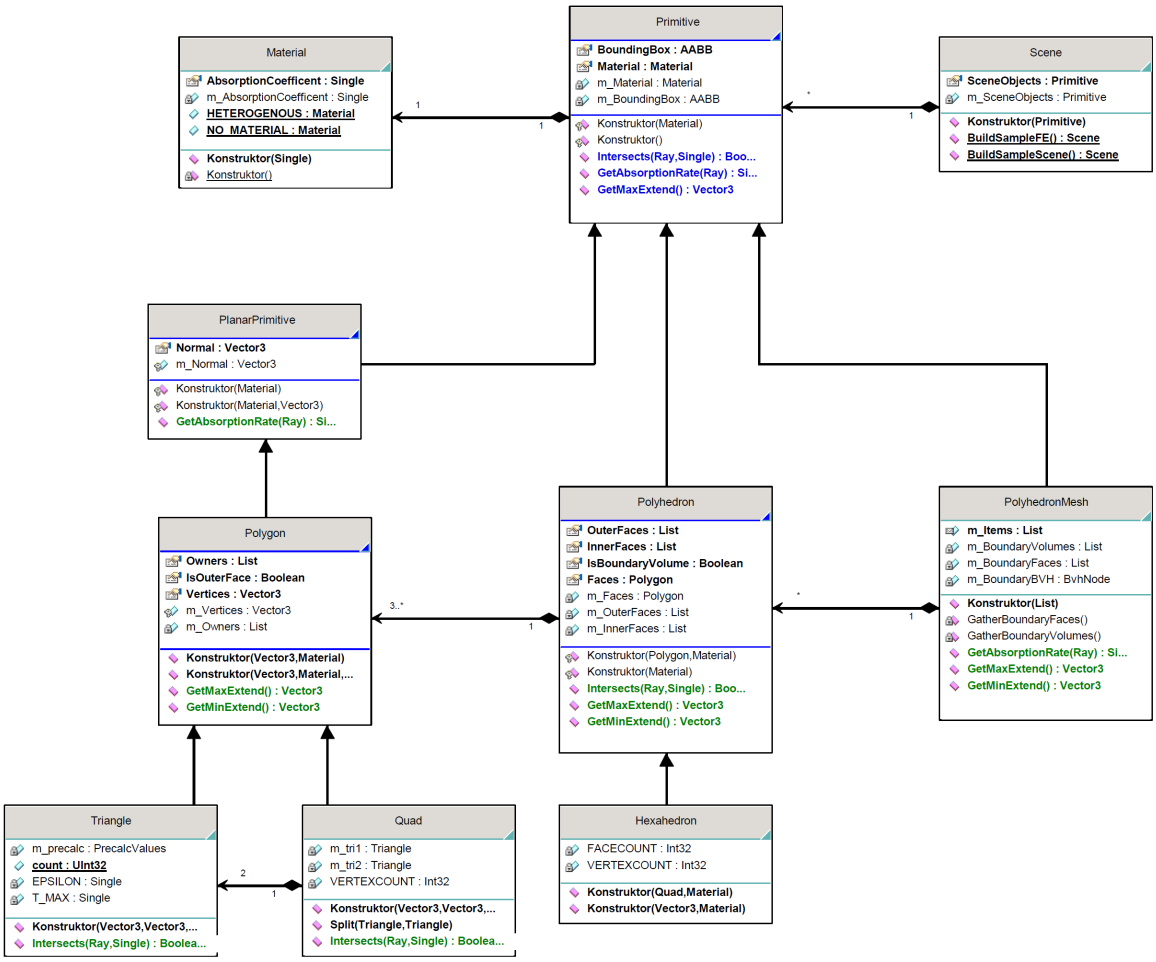


Abbildung 2.12: Klassen zur Modellierung einer Szene

Polygon

Polygone sind geometrische Objekte, die durch einen Satz komplanarer Vertices definiert werden. Die Property `IsOuterFace` besitzt den Wert `true`, falls das Polygon von keinem oder maximal einem Polyeder als Seitenfläche referenziert wird (vgl. `Owner`-Property).

Quad

`Quad` steht kurz für engl. „Quadrilateral“ und damit für allgemeine, aber ausschließlich konvexe, Vierecke. `Quad`-Objekte werden in je zwei Dreiecke zerlegt, um vom schnellen Schnittest der `Triangle`-Implementierung profitieren zu können.

Triangle

Dreiecke sind die Grundprimitive aller Szenen, die der Röntgensimulator in der aktuellen Version darstellen kann; nur sie verfügen über eine Implementierung der `Intersects`-Methode.

Polyhedron

Polyeder werden durch die Klasse `Polyhedron` repräsentiert. Jedes Polyeder referenziert eine Anzahl Polygone, deren `Owners`-Property wiederum eine Referenz auf das Polyeder enthält. Dies und die weiteren Felder und Properties dienen primär dem effizienten *Ray Traversal* durch Polyedernetze (siehe 2.4.6).

Hexahedron

Ein `Hexahedron` (Hexaeder) ist ein spezielles Polyeder mit sechs Quads als Seitenflächen.

PolyhedronMesh

Objekte dieses Typs repräsentieren die eigentlichen FEM-Modelle. Sie bestehen aus einer Menge zusammenhängender Polyeder. Im Falle der aktuellen Version des Röntgensimulators, handelt es sich dabei ausschließlich um Hexaeder-Netze.

Vector3

`Vector3` definiert einen 3D-Vektor bestehend aus drei `float`-Komponenten. Häufig genutzte Operationen sind mit Hilfe von Operator-Überladung implementiert.

Während die oben aufgeführten Klassen allesamt der geometrischen Modellierung der Szene dienen, existieren weitere wichtige Klassen, die die Szene an sich repräsentieren oder das eigentliche Raycasting implementieren (Abbildung 2.13).

Scene

Ein `Scene`-Objekt beinhaltet Referenzen auf sämtliche Objekte der Szene, die vom `Raytracer` gerendert werden sollen. Bei Oberflächen-Raytracern beinhaltet ein solches Szene-Objekt normalerweise auch eine Liste der Lichtquellen, aus deren Position zum ermittelten Schnittpunkt die jeweilige Farbe berechnet werden kann (*Shading*). Im Falle des Röntgensimulators entfällt dieser Teil der Implementierung.

Außerdem besteht eine Szene der Heilungssimulation in der Regel nur aus einem einzigen Primitiv vom Typ `PolyhedronMesh`. Die `Scene`-Implementierung unterstützt jedoch prinzipiell beliebig viele Objekte, und ermöglicht so auch die

Modellierung komplexerer Szenen, die sich möglicherweise aus mehreren Meshes oder anderen Primitiv-Typen zusammensetzen.

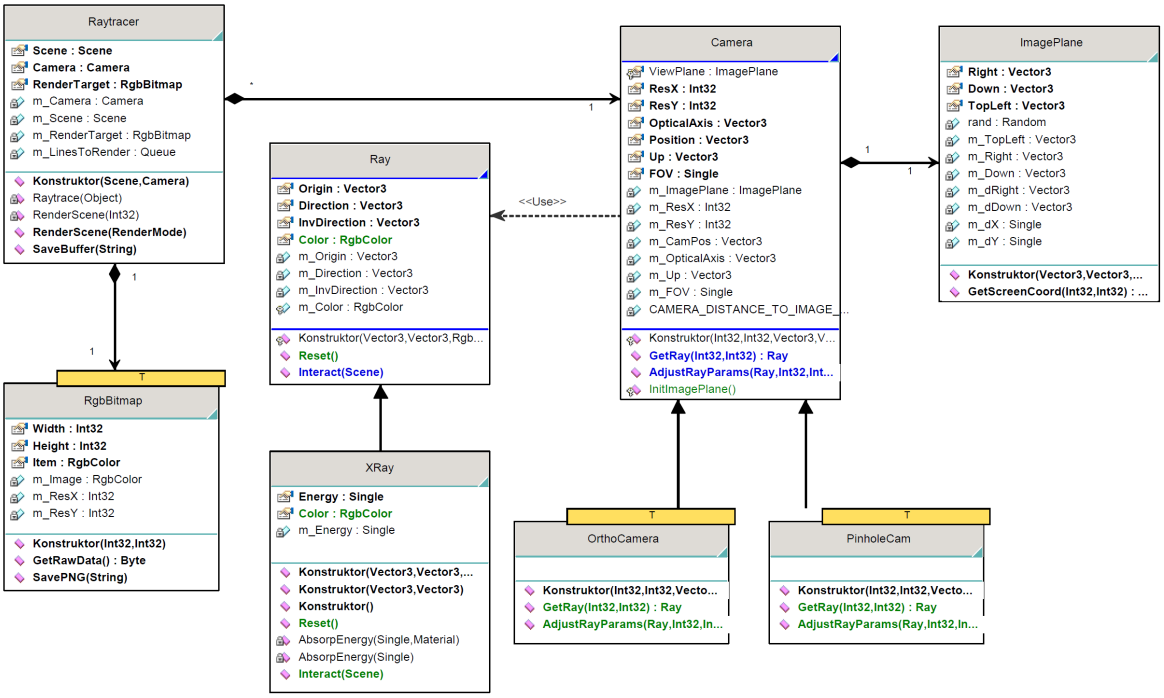


Abbildung 2.13: Ray- und Camera-Klassen

Camera

Der Augpunkt bzw. die virtuelle Kamera wird durch ein Objekt vom Typ `Camera` repräsentiert. Die Kamera-Parameter definieren Blickrichtung, das Blickfeld und die Position der Kamera. Für eine perspektivische Projektion findet die abgeleitete Klassen `PinholeCam<T>` Verwendung. Die abgeleitete Klasse `OrthoCamera<T>` leistet eine orthographische Projektion. Durch den Typparameter `T` spezifiziert man, welcher Strahltyp von der Kamera ausgesendet werden soll. Dieser dient dann als Rückgabetyp der Methode `GetRay(int, int)`, die einen Strahl durch die Bildebene für einen Bildpunkt (x, y) zurückliefert.

Um die Ausführungsgeschwindigkeit zu erhöhen, wurde zudem die Methode `AdjustRayParams(int, int, Ray)` eingeführt, die, abhängig von der Projektionsart, Ursprung oder Richtung eines Strahls so ändert, daß der Strahl den Bildpunkt (x, y) repräsentiert. Dadurch spart man sich das relativ teure Instanziieren eines neuen `Ray`-Objekts für jeden einzelnen Bildpunkt.

ImagePlane

`ImagePlane` stellt die virtuelle Bildebene dar, auf die die Szene projiziert wird. Sie wird von `Camera` verwendet, um die Parameter eines Strahls für einen Bildpunkt (x, y) zu gewinnen.

Ray

Strahlen werden durch drei Parameter definiert: Strahlursprung, -richtung und einen Farbwert. Im Falle von `XRay` wird der Wert des Feldes `Energy` vom Typ `float` auf einen Grauwert abgebildet. „Energie“ ist hier im Sinne von „Gesamtenergie aller durch den Strahl repräsentierten Photonen“ zu verstehen und daher direkt proportional zu dessen Intensität.

Die Implementierung der abstrakten Methode `Interact(Scene)` entscheidet wesentlich über das Ergebnis des Rendervorgangs. Ein Strahl vom Typ `XRay` berechnet hier beispielsweise ausschließlich die Schwächung seiner initialen Intensität durch die Objekte der übergebenen Szene. Eine einfache Implementierung eines „OpticRay“ hingegen könnte z. B. Oberflächenhelligkeiten berechnen.

Die Farbe eines Strahls wird in einem `RgbColor<byte>`-Struct gespeichert, das pro Kanal (Rot, Grün und Blau) eine Auflösung von 8 Bit bietet.

Raytracer

Hier findet das eigentliche Raycasting statt. Für jeden zu berechnenden Bildpunkt wird von der Kamera ein Strahl mit entsprechenden Parametern erzeugt und in den Szene geschickt. Der Strahl wechselwirkt daraufhin abhängig von der Implementierung der `Ray.Interact`-Methode mit den Objekten der Szene.

Der so ermittelte Farbwert des Strahls wird schließlich in einen Puffer (*Render Target*) vom Typ `RgbBitmap<byte>` geschrieben. Das Ergebnis kann nach dem Rendering im PNG-Format gespeichert werden (`SavePNG(string)`).

2.4.3 Generierung der Eingabedaten

Die notwendigen Eingabedaten werden direkt aus dem FE-Frakturmodell erzeugt. Dazu sind geringfügige Erweiterungen des Heilungssimulationsprogramms notwendig.

Vertex-Koordinaten

Das Exportieren der Vertexdaten erfordert wenige Zeilen APDL-Code (Listing 2.1). Das Ergebnis ist eine Textdatei, in der in jeder Zeile jeweils X-, Y- und Z-Komponente eines Vertices stehen.

```
*get, vertexCount, NODE, 0, COUNT
*dim, vertices, ARRAY, vertexCount, 3

*vget, vertices(1,1), NODE, 1, LOC, X
*vget, vertices(1,2), NODE, 1, LOC, Y
*vget, vertices(1,3), NODE, 1, LOC, Z

*mwrite, vertices, vertices.txt
%G  %G  %G
```

Listing 2.1: Export der Vertex-Koordinaten
Das APDL-Script schreibt die Koordinaten der Vertices (Knoten des FE-Modells) einmalig für einen Simulationslauf in ein zweidimensionales Array und gibt den Inhalt in eine Textdatei „vertices.txt" aus.

Hexaeder-Elemente

Eine weitere Textdatei beschreibt, wie diese Vertices zu Hexaedern zu verbinden sind. Für jedes Element werden die Vertices referenziert, aus denen das Element besteht. Im Falle von Hexaedern stehen also in jeder Zeile sechs Indizes, die den Zeilennummern in der Vertex-Datei entsprechen und auf die korrespondierenden Vertices verweisen. Abbildung 2.14 zeigt, wie und in welcher Reihenfolge die Vertices die Seitenflächen eines Hexaeders definieren.

Informationen über die Nachbarschaft von Elementen

Um das Raycasting zu beschleunigen, kann man die Nachbarschaftsbeziehungen der Hexaeder ausnutzen (siehe 2.5.4). Außerdem wird diese Information benötigt, damit sich benachbarte Elemente jeweils eine Seite teilen können (siehe dazu auch 2.3.4). Die Information, mit welchen anderen Hexaedern sich ein Element seine Seitenflächen teilt, wird ebenfalls als Eingabe für den Fuzzy-Controller benötigt. Auch hier liegen die Daten also bereits in einer Textdatei bereit, der Raycaster kann diese im Prinzip einfach mitbenutzen.

Die ursprünglich zu diesem Zweck verwendete APDL-Anweisung

```
*get, adjElemNo, ELEM, currentElemNo, ADJ, faceNo
```

liefert zwar die Elemente-Nummern (bzw. den Hexaeder-Index) abhängig von der Seitenfläche `faceNo`, die so gewonnenen Indizes der Nachbarelemente werden aber für die Verwendung durch den Fuzzy-Controller in absteigender Reihenfolge sortiert. Aus den resultierenden Daten kann nun nicht mehr direkt rekonstruiert werden, *welche* Seitenfläche sich benachbarte Elemente teilen. Daher

werden die Nachbarschafsinformationen nun zusätzlich unsortiert, d.h. in der ursprünglichen, von der Seitenfläche abhängigen Reihenfolge gespeichert.

Die ausgegebenen Daten (siehe Listing 2.2) sind folgendermaßen zu interpretieren: Jede Zeile *j* beschreibt die Nachbarschaftsbeziehungen eines Elements. Ein in Spalte *n* angegebener Index *i* bedeutet, daß sich das Element *j* die *n*-te Seitenfläche mit Element *i* teilt. Eine `0` bedeutet, daß an diese Seitenfläche kein Nachbarelement grenzt. Die Numerierung der Seitenflächen folgt der in ANSYS definierten Reihenfolge (vgl. Abbildung 2.14).

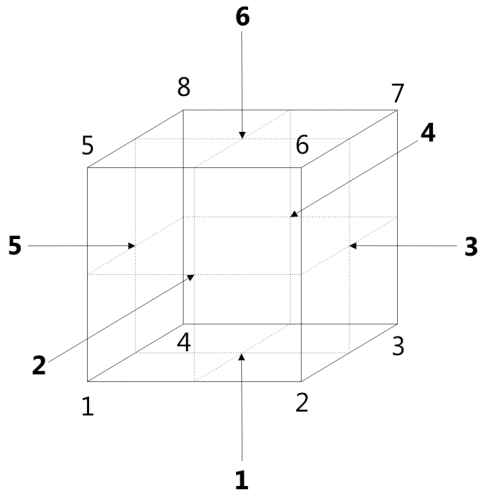


Abbildung 2.14: Nummerierung der Vertices und Seitenflächen der finiten Elemente in ANSYS 11.0 (Hexaeder)

Betrachten wir als Beispiel die beiden ersten Zeilen in Listing 2.2: Element Nr. 1 teilt sich seine dritte Seitenfläche mit Element Nr. 2. Die korrespondierende Seitenfläche von Element Nr. 2 ist dessen fünfte Seitenfläche, die es sich mit dem ersten Element teilt. Die dritte Seitenfläche des ersten Elements sowie die fünfte Seitenfläche des zweiten Elements sind somit identisch.

```
    0     0     2     4  6201    49
    0     0     3     5     1    50
    0     0   145     6     2    51
    0     1     5     7  6193    52
    ⋮     ⋮     ⋮     ⋮     ⋮     ⋮
    ⋮     ⋮     ⋮     ⋮     ⋮     ⋮
28607 28663 28672 22554 28670     0
28608 28664 28097 22553 28671     0
```

Listing 2.2: Ausschnitt aus dem Inhalt einer Nachbarschafts-Referenzdatei

Knochenkonzentrationen

Auch die Knochenkonzentrationen der Elemente finden sich in einer Textdatei. Diese wird während jeder Iteration erzeugt und dient als Eingabe für den Fuzzy-Controller (siehe 1.2.2). Für jedes Element werden hier zeilenweise physiologische und mechanische Eigenschaften eines Elements, wie die relative Knochenkonzentration, Durchblutung oder Knorpelanteil, aufgelistet. Für das Raycasting ist davon nur die Kochenkonzentration – dies ist die erste Spalte – relevant.

Materialarten

Die zu visualisierenden Heilungsmodelle beinhalten sowohl neu gebildeten Geflechtknochen als auch ursprünglichen kortikalen Knochen (*Compacta*). Beide Gewebetypen unterscheiden sich hinsichtlich ihrer Knochenmineraldichte. Kortikaler Knochen mit höherer Knochenmineraldichte absorbiert Röntgenstrahlung deutlich stärker.

Um dieser Eigenschaft Rechnung zu tragen, wird in der aktuellen Version des Röntgensimulators schlicht eine Liste der Elemente ausgegeben, die den ursprünglichen kortikalen Knochen repräsentieren. Diese Liste ändert sich im Verlauf der Simulation nicht. Die Absorptionsrate der betreffenden Elemente kann so beim Import entsprechen skaliert werden.

Diese Art, Materialien zu unterscheiden, ist sicher wenig flexibel oder gar elegant. Sie reicht für den momentanen Einsatzzweck des Raycasters jedoch aus.

2.4.4 Import der Simulationsdaten

Aus diesen Daten muß nun eine Szene rekonstruiert werden. Die Schnittstelle `IMeshInputFilter` definiert die Methode `GetMeshFromFile(string)`, die ein Polyeder-Mesh zurückliefert. Die einzige existierende Implementierung dieser Schnittstelle ist `Cal3DMeshReader`. Der Reader liest die im vorherigen Abschnitt erläuterten Simulationsdaten ein und erzeugt daraus ein Polyeder-Mesh. Dabei handelt es sich um einen mehrstufigen Prozeß.

Zunächst werden die Vertex-Daten eingelesen, sowie die Hexaeder-Index-Datei und die Nachbarschaftsreferenzdatei geparst. Schließlich werden noch die Knochenkonzentrationen der einzelnen Elemente eingelesen.

Im nächsten Schritt müssen diese Daten zueinander in Beziehung gesetzt und das Hexaeder-Netz aufgebaut werden. Alle Elemente, deren Knochenkonzentration kleiner gleich einem einstellbaren Schwellwert ist (Standardeinstellung: 0), werden komplett ignoriert. Ansonsten wird für jede Seitenfläche jedes Elements anhand der Konnektivitätsdaten überprüft, ob an diese Seitenfläche ein weiteres Element angrenzt. Ist dies der Fall und wurde das Element bereits instanziiert, wird dessen korrespondierende Seitenfläche referenziert. Andernfalls wird gemäß den Knoten-Indizes aus der Hexaeder-Datei eine neue Seitenfläche angelegt, wobei die spezifische Reihenfolge, in der ANSYS die Vertices jeder Seitenfläche anspricht und folglich auch exportiert, beachtet werden muß (vgl. 2.4.3 und Abbildung 2.14). Nach dem Anlegen des Hexaeders werden die `Owners`-Properties der Seitenflächen um eine Referenz auf das neue Hexaeder erweitert. Diese Eigenschaft findet später beim Ray-Traversal Verwendung.

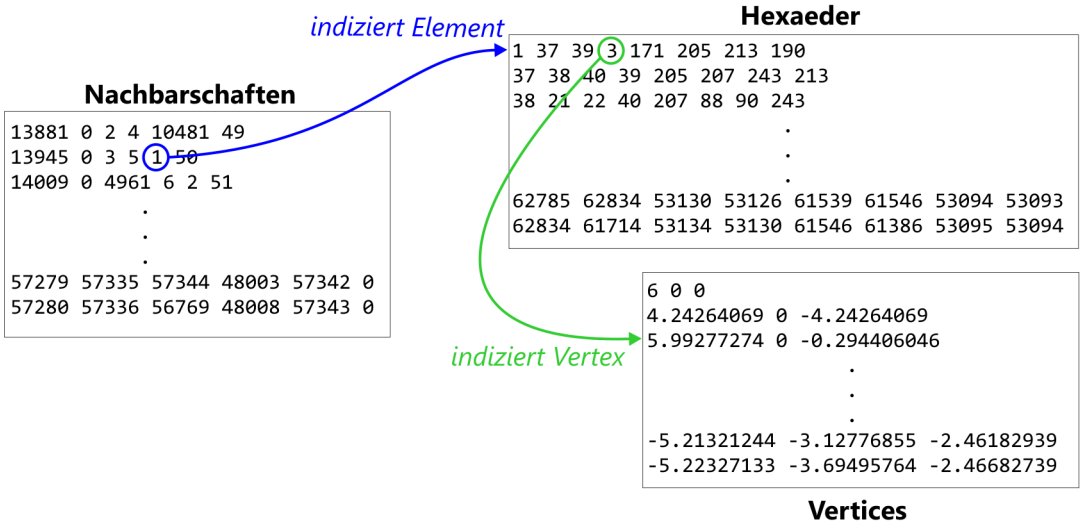


Abbildung 2.15: Zusammenhang der Nachbarschaftsdaten mit der Definition der Hexaeder-Elemente und den Vertex-Daten

Um das in 2.5.4 beschriebene Ray-Traversal-Verfahren zu beschleunigen, werden die außenliegenden Seitenflächen, also Seitenflächen, die nur von einem Polyeder referenziert werden, speziell markiert. Listing 2.3 faßt den groben Ablauf noch einmal kurz zusammen.

```
vertices ← read vertices from vertex_file
indexed hexahedra ← read indexed elements from hex_index_file
adjacencies ← read adjacencies from adjacencies_file

FOR ALL hexahedra IN indexed hexhedra
{
    IF bone_concentration(current hexahedron) >= THRESHOLD
    {
        FOR ALL faces OF current hexahedron
        {
            IF exists(adjacent) AND shares_face(current hexahedron,
                                                adjacent)
                current face ← get_shared_face(adjacent)

            ELSE
                current face ← create_face(vertices,
                                           indexed hexahedra)
        }

        hexahedron ← create_hexahedron(faces)
        set_owners_property(faces, hexahedron)
    }
}

mark_boundary_faces(hexahedra)

RETURN create_polyhedron_mesh(hexahedra)
```

Listing 2.3: Ablauf des Datenimports (Pseudocode)

2.4.5 Funktionsweise der virtuellen Kamera

Die Konfiguration der virtuellen Kamera entscheidet darüber, wie die Szene abgebildet wird. Sie bestimmt sowohl den Betrachterstandort als auch Blickrichtung, Blickwinkel und den Projektionstyp. Im Falle der `Camera`-Implementierungen `PinholeCam<T>` und `OrhtoCam<T>` definiert sie zudem, welcher Typ von Strahlen in die Szene geworfen wird.

Genaugenommen ist die Klasse `Camera` ein Konglomerat aus der eigentlichen Kamera und der Bildebene vom Typ `ImagePlane`. Der Grund dafür liegt darin,

daß die Bildebene in starker Abhängigkeit von den Kamera-Parametern konstruiert wird und auch nur von den `Camera`-Methoden verwendet wird.

Die Kamera benötigt als Parameter ihre Position, die gewünschte Blickrichtung, einen Up-Vektor, der die Neigung der Kamera bestimmt, sowie einen FOV-Parameter, der die Größe des Blickfeldes in Grad festlegt. Im Falle einer orthographischen Projektion wird der FOV-Parameter durch einen Parameter ersetzt, der die absolute Breite der Bildebene angibt.

Da der Parameter „Blickrichtung“ in der Regel intuitiv eher schwer zu bestimmen ist, implementiert `Camera` das Modell einer „Look-At“-Kamera: an Stelle der Blickrichtung übergibt man ihr als Parameter den Punkt, auf den sie fokussieren soll. Aus diesem Punkt und der Kameraposition kann dann unmittelbar die Blickrichtung berechnet werden (siehe Abbildung 2.16).

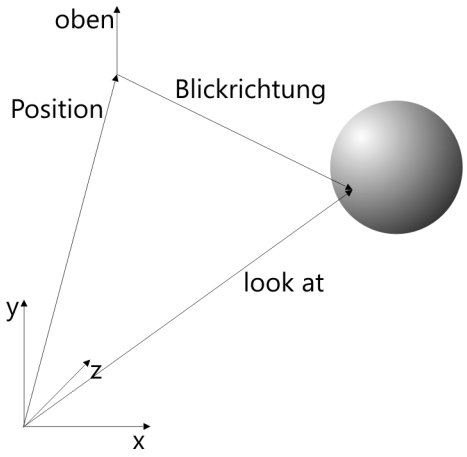


Abbildung 2.16: Parameter des Look-At-Kameramodells

Für das Raycasting werden Strahlen benötigt, die den Farbwert des jeweiligen zu rendernden Pixels repräsentieren. Die Methode `GetRay(int, int)` liefert einen Strahl für einen Bildpunkt mit den angegebenen Bildkoordinaten, indem sie den entsprechenden Punkt auf der Bildebene berechnet und dann, basierend auf der zugrundeliegenden Projektionsmethode, Ursprung und Richtung des Strahls festlegt. Bei der orthographischen Projektion dient als Ursprung immer der berechnete Punkt auf der Bildebene, während die Richtung immer gleich der Blickrichtung der Kamera ist. Um eine perspektivische Projektion zu erzielen, muß der Ursprung auf die Kameraposition gesetzt werden. Die Richtung errechnet sich aus den Ortsvektoren der Kameraposition und des Punktes auf der Bildebene (siehe Abbildung 2.17).

2.4.6 Raycasting

Die Klasse `Raytracer` setzt den eigentlichen Raycasting/-tracing-Vorgang um, indem sie für jeden Bildpunkt einen entsprechenden Strahl von der `Camera`-Instanz anfordert (`Camera.GetRay(x, y)`) bzw. die Parameter eines existierenden Strahls mittels `Camera.AdjustRayParams(Ray, x, y)`) anpaßt. Dann ruft der Raytracer die Methode `Ray.Interact(Scene)` des zurückgelieferten Strahls, woraufhin der Strahl, abhängig von seiner Implementierung, mit den Objekten der Szene wechselwirkt.

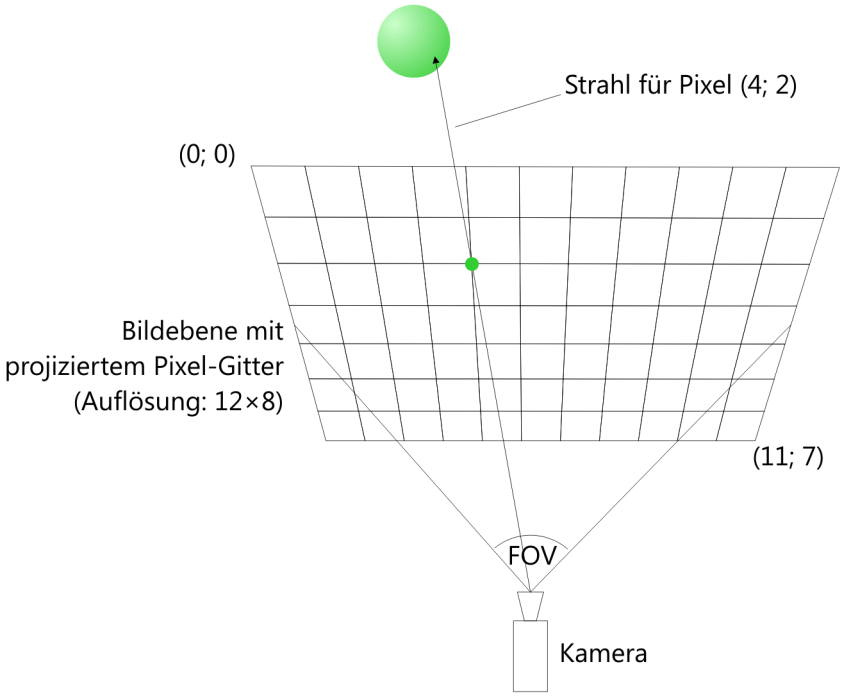


Abbildung 2.17: Projektion der Szene auf die Bildebene

Um Ergebnisse zu erzielen, die denen einer Röntgenaufnahme ähneln, wählt man als Typparameter der `Camera`-Implementierungen `PinholeCam<T>` oder `OrthCam<T>` die Klasse `XRay`. Das hat zur Folge, daß der Raytracer von der `Camera`-Instanz auch Strahlen vom Typ `XRay` erhält; beim Aufruf der `Interact`-Methode wird folglich die `Interact`-Methode der Klasse `XRay` gerufen.

Diese Methode ist sehr simple gehalten: sie frägt in einer Schleife von allen Objekten der Szene mittels `Primitve.GetAbsorptionRate(Ray)` die jeweilige Absorptionsrate ab. Dieser Wert entspricht dem Integral in (2.8). Bei Körpern mit konstantem Absorptionskoeffizienten ist die Absorptionsrate das Produkt aus Absorptionskoeffizient und der durch den Strahl innerhalb des Körpers zurückgelegten Wegstrecke. Da eine Szene im konkreten Anwendungsfall nur aus einem einzigen Polyeder-Mesh besteht, hat dieses „Brute-Force-Rendering“ auch keine negativen Auswirkungen auf die Rendergeschwindigkeit. Um Szenen mit sehr vielen Einzel-Primitiven effizient rendern zu können, müßten allerdings noch angemessene Beschleunigungsstrukturen in die `Scene`-Implementierung integriert werden.

Mit dem momentanen Wissen über die Arbeitsweise des Röntgensimulators erschließt sich vermutlich nicht sofort, weshalb man auf die Klasse `PolyhedronMesh` nicht komplett verzichtet und die einzelnen Polyeder bzw. Dreiecke statt dessen nicht einfach direkt als einzelne Objekte der Szene speichert. Der Grund für diese Kapselung ist, daß Objekte einer `Scene` ganz allgemeine Primitive re-

präsentieren, ob Kugeln, Dreiecke oder beliebige sonstige geometrische Einheiten. Speicherte man die Polyeder als Primitive einer Szene-Instanz, wäre der gezielte Zugriff auf die Seitenflächen der Polyeder nicht möglich. Das wiederum ist aber nötig, um das Ray-Traversal durch ein Polyeder-Netz durch Ausnutzung spezieller Eigenschaften eines Polyeder-Netzes effizienter zu gestalten (siehe 2.5.4).

Der Aufruf von `Primitve.GetAbsorptionRate(Ray)` in der `Interact(Scene)`-Methode von `XRay` führt also dazu, daß die `GetAbsorptionRate(Ray)`-Methode von `PolyhedronMesh` aufgerufen wird, da die Frakturheilungsszene, wie oben beschrieben, aus einem einzelnen Polyeder-Mesh besteht. Die einzelnen Poly- bzw. Hexaeder des Gitters sind zwar konvex, das Gitter selbst kann jedoch auch nichtkonvex sein. Der Absorptionskoeffizient ist nur innerhalb jeweils eines Elements konstant. Um die Absorptionsrate für einen Strahl, der das Polyeder-Mesh schneidet, zu berechnen, muß daher, gemäß (2.13) bzw. (2.14), folgendermaßen vorgegangen werden:

Für jedes Polyeder wird geprüft, ob es vom Strahl geschnitten wird. Falls Schnittpunkte mit genau zwei Seitenflächen existieren, wird daraus der innerhalb des Elements zurückgelegte Weg berechnet. Zusammen mit dem Absorptionskoeffizienten des Elements ergibt sich so die Absorptionsrate für dieses Element. Summiert man alle so berechneten Absorptionsraten auf, erhält man die gesamte Absorptionsrate für das Gitter.

Nachdem die Absorptionsrate ermittelt wurde, kann die XRay-Instanz seine Energie entsprechend modulieren, indem er gemäß (2.13) aus seiner aktuellen Intensität und der Absorptionsrate seine Restenergie berechnet. Um den Farb- bzw. Grauwert des gerenderten Pixels zu erhalten, wird die Restenergie des `XRay`-Objekts linear auf einen Grauwert abgebildet, der dann in einen Puffer (Render Target) geschrieben wird. Abbildung 2.18 faßt die Interaktion der einzelnen Objekte während des Renderings eines einzelnen Pixels nochmals zusammen.

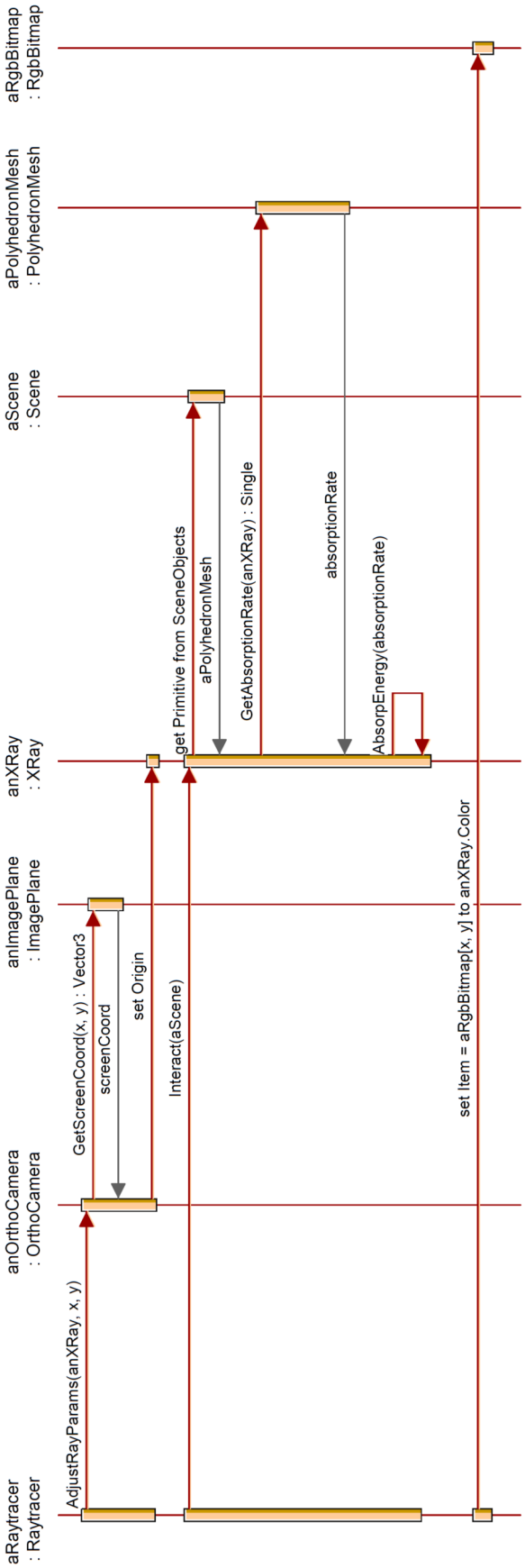


Abbildung 2.18: Arbeitsschritte zur Berechnung eines Pixelfarbwerts

2.4.7 Ausgabe des Ergebnisses

Der Röntgensimulator unterstützt das Speichern des gerenderten Bildes als PNG[21]-Datei. Dazu wird die freie Grafikbibliothek *DevIL* (*Developer's Image Library*, ehemals *OpenIL*) verwendet. Das *Tao Framwork* stellt die notwendigen Wrapper-APIs für C# (und alle anderen Sprachen, für die ein Compiler existiert, der CIL-Assemblies erzeugen kann) zur Verfügung. Das erzeugte Bild wird in einem Objekt des Typs `RgbBitmap<T>` gepuffert. Daraus wird zunächst mittels der DevIL-API `ilLoadDataL(byte[], int, int, int, int, byte)` eine für die DevIL-Routinen verständliche Repräsentation generiert, die dann anschließend mittels `ilSave(IL_PNG, string)` als PNG-Datei gespeichert werden kann.

Da DevIL auch viele weitere Bildformate wie z. B. TIFF, TGA oder auch OpenEXR unterstützt, wäre es problemlos möglich, diese Funktionalität auch durch den Röntgensimulator zu nutzen, sofern das User Interface entsprechende Optionen zur Formatwahl bietet.

2.5 Optimierungen

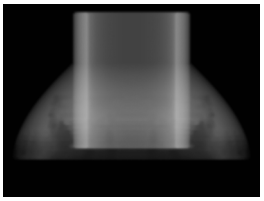


Abbildung 2.19: Erzeugnis eines frühen Prototypen

In den vorangegangenen Abschnitten haben wir gesehen, wie die Bilderzeugung aus den Ausgabedaten der Heilungssimulation prinzipiell ablaufen kann. In der Tat arbeiteten die ersten Entwürfe und Prototypen auch im wesentlichen auf diese Weise. An der Qualität der gerenderten Bilder, die diese frühen Versionen erzeugten, gab es zwar nur wenig auszusetzen (siehe Abbildung 2.19). Doch der Röntgensimulator war in dieser Form deutlich zu langsam, um praxistauglich aufgelöste Bilder in angemessener Zeit zu berechnen; nicht umsonst besitzt Abbildung 2.19 nur Briefmarkenformat. Die Raycasting-„Performance" der ersten Prototypen bewegt sich auf einem Niveau von lediglich etwa 25 Rays/s für ein Modell bestehend aus etwas mehr als 28 000 Hexaedern – für die praktische Anwendung schlicht untauglich. Im Falle von Abbildung 2.19 beträgt die Renderingdauer für ein Bild mit einer Auflösung von 200×150 Bildpunkten mehr als 18 Minuten[22].

21 PNG: *Portable Network Graphics* ist ein Grafikformat, das eine verlustlose Kompression von Bitmaps unterstützt

22 Sofern nicht anders angegeben, beziehen sich alle Messungen auf ein System mit AMD Athlon 64 X2 4200+ (2,2 GHz, 1 MiB L2-Cache) und 2 GiB RAM

Der beschriebene „Brute-Force"-Algorithmus skaliert linear mit der Auflösung des FE-Modells, da er stupide sämtliche Primitive auf einen Schnitt mit jedem Strahl testet. Während mit diesem Algorithmus Szenen aus knapp hundert Hexaedern innerhalb von einigen Sekunden gerendert werden können, dauert das Rendern eines – mit 28 000 Hexaedern noch relativ niedrig aufgelösten – FE-Modells bereits die rund 280-fache Rechenzeit bei gleicher Bildauflösung.

2.5.1 Multithreading

Die erste und einfachste Maßnahme, um einem Raytracer zu mehr Leistung zu verhelfen, ist, mehrere Strahlen parallel zu berechnen. Jeweils zwei Strahlen für zwei unterschiedliche Bildpunkte können völlig unabhängig voneinander verfolgt werden. *Multithreading* ist eine Methode, Berechnungen explizit-parallel auf mehreren CPUs durchzuführen. Die Verwendung dieser Methode drängt sich im Zeitalter von Dual-, Quad- und demnächst vielleicht auch „Multi-Core"-Prozessoren förmlich auf.

Aktivitäten in einem Betriebssystem werden durch *Prozesse* repräsentiert [NS01]. Prozesse bestehen aus drei Bestandteilen: einem *Adreßraum* (abstrakte Container für Programme und ihre Daten), einer *Handlungsvorschrift* (Programm-Code) und mindestens einem *Aktivitätsträger* (*Thread*), der einen sequentiellen Kontrollfluß beschreibt und eine Abstraktion des physischen Prozessors darstellt. Existiert in einem Adreßraum mehr als ein Thread, spricht man von *Multithreading*. Sofern genügend freie Systemressourcen (CPUs, Speicher, I/O) zur Verfügung stehen, können die einzelnen Threads parallel aktiv sein (*Nebenläufigkeit*).

Zur effizienten Parallelisierung einer Anwendung ist es zunächst notwendig, die zu erledigende Arbeit in unabhängig voneinander bearbeitbare Teile aufzuspalten. Auf einer Maschine, die die gleichzeitige Ausführung von *n* Threads in Hardware unterstützt, könnte man das zu rendernde Bild dazu einfach in *n* Teile unterteilen, die dann unabhängig voneinander berechnet würden. Solange man Brute-Force-Rendering betreibt, ist diese Vorgehensweise durchaus brauchbar, da hier die Berechnung jedes Pixel etwa gleich viel Zeit in Anspruch nimmt. Geht man jedoch zu verfeinerten Rendering-Methoden über (vgl. die folgenden Abschnitte), ist dies nicht mehr der Fall. Komplexe Bildteile benötigen dann wesentlich mehr Rechenzeit als Bildteile, auf denen nur der Hintergrund zu sehen ist. Während ein Thread seinen ihm zugeteilten Bereich längst gerendert hat, arbeitet ein anderer Thread noch an seiner komplexer zu berechnenden Aufgabe. Die CPU-Ressourcen, die der bereits beendete Thread belegt hatte, lägen in diesem Fall brach.

Aus diesem Grund arbeitet das Load-Balancing des Röntgensimulators mit einem flexibleren „Line-Pooling“-Mechanismus (Listing 2.4): Die zu rendernden Zeilen werden in einer Warteschlage gespeichert. Solange die Warteschlange nicht leer ist, beziehen die Threads die jeweils nächste zu rendernde Zeile aus der Warteschlange. Dadurch wird die CPU-Auslastung nahezu konstant bei 100 % gehalten.

Eine andere mögliche Load-Balancing-Methode bestünde darin, die zu rendernden Bildteile zwar statisch, aber dafür feingranularer, d.h. zeilen- oder gar pixelweise (Thread *n* berechnet *n*-te Zeile bzw. *n*-tes Pixel) zuzuteilen.

```
while (true)
{
    lock (linesToRender)
    {
        if (linesToRender.Count == 0)
            break;

        y = linesToRender.Dequeue();
    }

    for (int x = 0; x < Camera.ResX; ++x)
    {
        // get ray for point (x,y) on ImagePlane
        ray = Camera.GetRay(x, y);
        ray.Interact(Scene);
        renderTarget[x, y] = ray.Color;
    }
}

// signal main thread
threadState.threadFinished.Set();
```

Listing 2.4: Thread-Load-Balancing des Röntgensimulators

2.5.2 Schnellere Schnittests

Wie bekannt, führt der Primitiv-Raycaster für jeden Strahl und jedes Hexaeder je bis zu sechs Schnittests mit den Seitenflächen der Elemente durch. Um Abbildung 2.19 aus einem 28 000-Elemente-Modell zu erzeugen, sind somit maximal insgesamt 5 040 000 000 `Quad`-Schnittests von Nöten. Diese Zahl steigt noch dazu

linear mit der Zahl der Elemente und der zu erzeugenden Pixel. Daher ist es naheliegend, an genau diesem Punkt anzusetzen, möchte man die Leistung des Raycasters spürbar erhöhen.

Polygon/Strahl-Schnittest

Da die Elemente des Modells je sechs konvexe Vierecke als Seitenflächen besitzen, muß der Schnittest den Schnittpunkt eines Strahls mit einem Viereck bestimmen können.

Zur Berechnung von Schnittpunkten eines Strahls mit einem Polygon, wird zunächst getestet, ob der Strahl die Ebene schneidet, in der die Vertices des Polygons liegen. Unter Verwendung der Normalengleichung der Ebene und der Strahlparameter läßt sich die Distanz t des Schnittpunktes zum Strahlursprung bestimmt. Für den Fall $t \leq 0$ existiert kein Schnittpunkt mit der Ebene.

Nun muß überprüft werden, ob sich der berechnete Schnittpunkt auch innerhalb des Polygons befindet. Dazu projiziert man sowohl das Polygon als auch den Schnittpunkt mit der Ebene durch „Wegwerfen“ einer der drei Koordinatenachsen (wobei man hier aus Gründen der numerischen Stabilität die Dimension wählen sollte, in der das Polygon die geringste Ausdehnung besitzt) ins Zweidimensionale [Dodoo]. Um festzustellen, ob der projizierte Schnittpunkt P innerhalb des 2D-Polygons liegt, gibt es zwei populäre Methoden:

Beim *Odd-Even-Test* zählt man, wie oft eine Halbgerade, die ihren Ursprung im Punkt P hat, die Kanten des Polygons schneidet und erhält so die *Crossing Number*. Ist die Crossing Number eine gerade Zahl, befindet sich der Punkt außerhalb und der Strahl schneidet das (3D-)Polygon nicht. Eine ungerade Crossing Number bedeutet, daß der Strahl das Polygon trifft und somit ein Schnittpunkt existiert.

Eine andere Möglichkeit stellt der *Winding-Number-Test* dar. Beim Winding-Number-Test wird gezählt, wie oft der Punkt von den Seiten des Polygons „umwickelt“ wird. Diesen Wert kann man durch einfaches Aufsummieren der (vorzeichenbehafteten) Winkel, die je zwei Vertices mit dem Punkt P aufspannen, ermitteln; es existieren aber auch effizientere Verfahren [Hai94, Sun01].

Dreieck/Strahl-Schnittest

Die überwiegende Zahl der polygonbasierten Computergrafiksysteme setzt jedoch Dreiecke als Grundprimitive ein. Für diesen Polygontyp wurden in den letzten Jahren sehr effiziente Schnittests entwickelt. Um diese Algorithmen nutzen zu können, splittet der Röntgensimulator jedes Quad daher intern zunächst in zwei Dreiecke auf. Da beide Dreiecke aus dem gleichen Viereck erzeugt werden, liegen beide in einer Ebene; das bedeutet, daß ein Strahl maximal eines der beiden Dreiecke schneiden kann. Wurde ein Schnitt mit dem ersten getesteten

Dreieck festgestellt, kann auf einen Schnittest des zweiten Dreiecks verzichtet werden.

Der eingesetzte Algorithmus (siehe Listing 2.5) basiert auf dem in [Wal04] vorgestellten Verfahren, das seinerseits eine Abwandlung des *Möller-Trumbore-Tests* [MT97] ist. Wie in den oben angesprochenen Verfahren für Schnittests mit allgemeinen Polygonen, ermittelt man als erstes die Distanz t des Schnittpunktes des Strahls mit der Ebene, in der das Dreieck liegt, zum Strahlursprung. Befindet sich t in einem gültigen Intervall, wird getestet, ob sich der Schnittpunkt $P = O_{\text{Ray}} + t\,D_{\text{Ray}}$ innerhalb des Dreiecks befindet. Dazu werden die *baryzentrischen Koordinaten*[23] des Schnittpunkts berechnet. Trifft der Strahl das Dreieck, erfüllen die baryzentrischen Koordinaten bestimme Eigenschaften [MT97]. Durch die Projektion auf eine 2D-Koordinatenebene (entweder XY-, YZ- oder XZ-Ebene) kann die Berechnung der Koordinaten vereinfacht werden.

Wie Wald in [Wal04] zeigt, kann auch diese Projektionsmethode noch weiter optimiert werden, indem Parameter, die für jeden Strahl konstant sind, in einem Vorverarbeitungsschritt für alle Dreiecke berechnet und in einer entsprechenden Datenstruktur gespeichert werden. Auch der Schnittest des Röntgensimulators macht davon gebraucht (`PrecalcValues`-Struktur in Listing 2.5).

23 Für einen Punkt $P(u, v)$ in einem Dreieck $\Delta V_0 V_1 V_2$ gilt
$P(u, v) = (1-u-v)\,V_0 + uV_1 + vV_2$
(u, v) sind die *baryzentrischen Koordinaten* des Punktes. Falls P innerhalb des Dreiecks liegt, gilt $u \geq 0 \ \wedge \ v \geq 0 \ \wedge \ u + v \leq 1$ [MT97]

```
public override bool Intersects(Ray ray, ref float t)
{
    float d = 1.0f / (ray.Direction[precalc.k]
              + precalc.nu * ray.Direction[precalc.u]
              + precalc.nv * ray.Direction[precalc.v]);

    // distance to the intersection with the plane
    t = (precalc.nd - ray.Origin[precalc.k]
         - precalc.nu * ray.Origin[precalc.u]
         - precalc.nv * ray.Origin[precalc.v]) * d;

    if (!((EPSILON < t) && (t < T_MAX)))
        return false;

    // intersection point in the projected plane
    float pu = ray.Origin[precalc.u] + t * ray.Direction[precalc.u]
               - Vertices[0][precalc.u];
    float pv = ray.Origin[precalc.v] + t * ray.Direction[precalc.v]
               - Vertices[0][precalc.v];

    // calc barycentric coordinates
    float lambda = pv * precalc.b_nu + pu * precalc.b_nv;

    if (lambda < 0.0f)
        return false;

    float mu = pu * precalc.c_nu + pv * precalc.c_nv;

    if (mu < 0.0f)
        return false;

    if (lambda + mu > 1.0f)
        return false;

    return true;
}
```

Listing 2.5: Strahl-Dreieck-Schnittest

2.5.3 Bounding Volumes

Die oben vorgestellten Methoden zur Berechnung des Schnittpunkts eines Strahls mit einem Dreieck sind zwar bereits sehr performant. Es gibt jedoch geometrische Körper, für die noch wesentlich schnellere Schnittests bekannt sind. Verwendet man solche Körper als Hüllkörper für die eigentlich zu rendernden Primitive, bezeichnet man diese Hüllkörper auch als *Bounding Volumes*.

Ein Bounding Volume umschließt ein komplexeres Primitiv vollständig. Folglich muß ein Strahl, der das eingeschlossene Primitiv schneidet, auch zwingend dessen Bounding Volume schneiden. Bei Szenen, die aus vielen, relativ kleinen Primitiven bestehen, schneidet jeder Strahl nur eine kleine Untermenge der Szeneprimitive. Verfügen die Szene-Primitive über Bounding Volumes, genügt damit für den Großteil der Strahl-Primitiv-Schnittests ein Schnittest mit dem jeweiligen Bounding Volume, da dieser immer dann negativ ausfällt, wenn auch der Strahl-Dreieck-Schnittest negativ wäre. Die Umkehrung gilt jedoch nicht, d.h. aus dem positive Ergebnis eines Schnittests mit einem Bounding Volume kann man nicht die Existenz eines Schnittpunktes des Strahls mit dem eingeschlossenen Primitv folgern.

Als Bounding Volumes eignen sich verschiedene einfache, konvexe geometrische Körper unterschiedlich gut. Am häufigsten finden sogenannte *AABBs* (*Axis Aligned Bounding Box*) Verwendung. Zwar umschließt eine AABB das eingeschlossene Primitiv meist nicht optimal (d.h. möglichst „eng“, der konvexen Hülle möglichst nahekommend); diesen Nachteil machen AABBs durch den geringen Speicherverbrauch – es müssen lediglich die minimalen und maximalen Ausdehnungen der AABB hinterlegt werden – und den sehr schnellen Schnittest wieder wett.

Auch der Röntgensimulator setzt AABBs zur Beschleunigung bzw. Vermeidung überflüssiger Dreieck-Schittests ein. Als Algorithmus kommt das in [WBMS04] vorgestellte Verfahren nahezu unmodifiziert zum Einsatz. Allein die Verwendung von AABBs um die einzelnen Polyeder sorgt bereits für einen Speed-Up von Faktor 2,4 beim Rendering des 28 000-Hexaeder-Modells. In Anbetracht des geringen Implementierungsaufwandes, ist der Einsatz von Bounding Volumes ein offensichtlich lohnenswertes Unterfangen. In der aktuellen Version wird jede Seitenfläche (`Quad`) von einer AABB begrenzt (siehe auch 2.5.5).

2.5.4 Nutzen der Konnektivitätsinformation

Die zu rendernden Szenen aus den Daten der Frakturheilungssimulation bestehen, wie wir in den vorangegangenen Abschnitten erfahren haben, ausschließlich aus einem zusammenhängenden Netz aus Polyedern. Die Konnektivität der Poly-

eder kann man sich zu Nutze machen, um das Rendering effizienter zu gestalten. Der zugrundeliegende Gedanke ist folgender [Moro04, MS06]: Da ein Strahl, der ein von Nachbarn umgebenes Polyeder schneidet auch einen seiner Nachbarn schneiden muß, kann man sich an den jeweiligen Nachbarelementen „entlang hangeln“ und muß somit nicht sämtliche Polyeder des Gitters auf einen Schnitt mit dem Strahl testen.

Ermitteln der Boundary Faces

Damit man dieses Verfahren anwenden kann, benötigt man die Information darüber, wo ein Strahl in das Volumen eindringt. Da das Gitter in der Regel nichtkonvex ist, kann auch mehr als ein Eintrittspunkt existieren. Der Strahl kann in das Volumen nur über Volumen eindringen, die mindestens eine Seitenfläche besitzen, an die kein weiteres Polyeder grenzt. Diese Flächen bezeichnen wir im folgenden als *Boundary Faces*. In einem Vorverarbeitungsschritt können diese Flächen leicht an Hand der Zahl der auf sie verweisenden Polyeder (`Owners`-Property) erkannt und gesammelt werden. Die Zahl der Boundary Faces ist normalerweise deutlich geringer als die Gesamtzahl der Flächen in einem Polyeder-Mesh (vgl. Abbildung 2.20); entsprechend verringert sich die Zahl der Schnittests, die notwendig sind, um die Seitenflächen zu finden, über die der Strahl in das Volumen ein- bzw. austritt. Diese Flächen nennen wir im folgenden *Entry Faces*, auch wenn es sich dabei genaugenommen ebenso um „Exit Faces“ handeln kann.

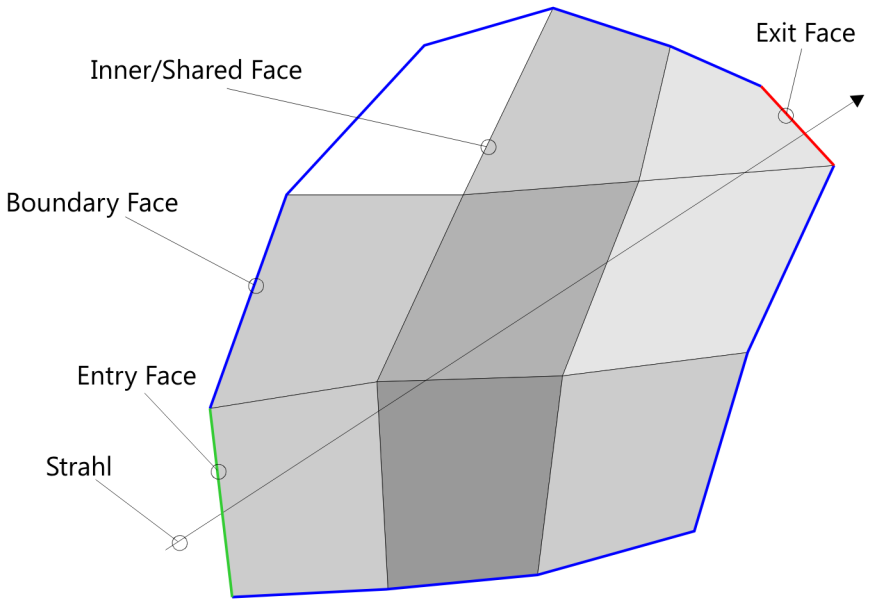


Abbildung 2.20: Ray-Traversal unter Berücksichtigung der Konnektivität der Elemente

Ray Traversal

Nun iteriert man über die so ermittelten Entry Faces. Für jede dieser Flächen wird das zugehörige Polyeder über die `Owners`-Property ermittelt. Der Strahl muß das Polyeder wieder über eine der restlichen Seitenflächen verlassen, also testet man diese auf einen Schnitt mit dem Strahl. So erhält man auch die im Polyeder zurückgelegte Strecke und kann die Absorptionsrate des Polyeders, in dem sich Strahl momentan befindet, berechnen.

Schneidet der Strahl ein weiteres Boundary Face, verläßt der Strahl das Volumen durch diese Fläche wieder. Die aktuelle Iteration ist damit zu Ende. Das geschnittene Boundary Face wird aus der Liste der Entry Faces entfernt, so daß nicht mehrmals der gleiche Weg verfolgt werden kann. Der Algorithmus fährt mit dem nächsten geschnittenen Entry Face fort.

Handelt es sich bei der geschnittenen Fläche jedoch um eine innere Fläche (*Shared Face*, *Inner Face*), grenzt also ein weiteres Polyeder an diese Seite, wird dieses Nachbar-Polyeder als nächstes Element ausgewählt. Dieses wird wiederum auf einen Schnitt mit seinen Seitenflächen geprüft und dessen Absorptionsrate kumuliert. Dies setzt man solange fort, bis der Strahl das Volumen durch ein Boundary Face wieder verläßt.

Falls man für die Strahlverfolgung die Reihenfolge der Entry Faces beachten muß (z. B. für Front-to-Back Rendering), müssen diese zum einen nach deren Distanz zum Strahlursprung sortiert werden. Zum anderen muß explizit zwischen Flächen, durch die der Strahl in das Volumen eintritt und solche, durch die er es wieder verläßt, unterschieden werden, um den Strahl der „richtigen“ Richtung entlang durch das Volumen zu verfolgen. Die Unterscheidung von Entry und Exit Faces ist relativ leicht umsetzbar: im Falle eines Entry Faces liegt der Schnittpunkt eines Strahls mit diesem näher am Strahlursprung, als der Schnitt mit der nächsten inneren Fläche des Polyeders; bei Exit Faces verhält es sich genau umgekehrt.

Im Gegensatz zu dieser in [Mor04] vorgestellten Variante, kommt der Röntgensimulator mit einer vereinfachten Version aus. Die Richtung der Strahlverfolgung ist hier irrelevant (siehe 2.3.2); damit ist auch eine Unterscheidung zwischen Entry und Exit Faces nicht von Nöten.

2.5.5 Bounding Volume Hierarchy

Bereits eine naive, unoptimierte Implementierung aller bisherigen beschriebenen Optimierungsmöglichkeiten, drückt die für das Rendering der 28 000-Hexaeder-Szene benötigte Zeit um rund den Faktor 40 auf unter eine halbe Minute. Es

existiert jedoch noch ein weiterer Flaschenhals, den es zu beseitigen gilt: Bei der Suche nach den Entry Faces für das Ray Traversal, müssen weiterhin alle Boundary Faces getestet werden; die Rendering-Dauer skaliert somit linear mit der Zahl der äußeren Flächen.

Die Grundidee

Wie in 2.5.3 besprochen, verwendet der Röntgensimulator Bounding Volumes, um die Zahl der unnötigerweise durchgeführten Dreieck-Schnittests zu senken. Diese Idee läßt sich aber noch weitertreiben: Faßt man mehrere Primitive bzw. deren Bounding Box jeweils zu einem „Über"-Bounding-Volume zusammen, muß man für Strahlen, die keines der eingeschlossenen Primitive/Bounding Volumes trifft, nur einen Schnittest mit dem übergeordneten Bounding Volume durchführen. Diese Konstruktion übergeordneter Bounding Volumes durch Aggregation kleinerer Volumen führt man rekursiv fort, bis die ganze Szene von einer einzigen Bounding Box umschlossen wird. So erhält man schließlich eine Baumstruktur aus Bounding Volumes, deren Blätter Referenzen auf die eigentlichen Primitive besitzen. Die so entstandene Beschleunigungsstruktur trägt den bezeichnenden Namen *Bounding Volume Hierarchy*, kurz *BVH*. Die BVH kann, wie beschrieben, entweder *Bottom-Up* durch Zusammenfassung kleinerer Volumen oder *Top-Down* durch sukzessives Aufteilen übergeordneter Volumen konstruiert werden [Hav04, WBS07].

Mit Hilfe einer solchen BVH läuft das Raycasting wesentlich effizienter ab. Verfehlt ein Strahl bereits die Bounding Box der Wurzel der BVH, sind keinerlei weiteren Schnittests notwendig. Nur bei einem Treffer müssen beide Kinder-Knoten auf einen Schnitt mit dem Strahl getestet werden. Ist der Test negativ, können alle tieferliegenden Knoten der Hierarchie ignoriert werden. Bei einem positiven Ergebnis hingegen, fährt man rekursiv fort. Erreicht man einen Blatt-Knoten, sind die durch dessen Bounding Box eingeschlossene Primitive Kandidaten für einen Treffer durch den Strahl.

Konstruktion

Zur Konstruktion der BVH verfolgt die Implementierung einen vergleichsweise simplen Top-Down-Ansatz, der mit den zu rendernden Szenen aber recht gut harmoniert. Nachdem die Boundary Faces des Modells gesammelt wurden, berechnet man die Bounding Box, die diese Primitive umschließt. Von diesem Wurzelknoten ausgehend erzeugt der Konstruktionsalgorithmus den Baum. Jeder Knoten muß in zwei Kind-Knoten unterteilt werden. Dazu ermittelt man die Dimension, in der die zu unterteilende Bounding Box die größte Ausdehnung besitzt. Senkrecht zu dieser Achse wird der Knoten dann in der Mitte unterteilt.

Jetzt müssen die Primitive des Elternknotens noch auf die neu entstandenen Knoten verteilt werden. Die Primitive werden dabei jeweils dem Knoten zugeordnet, zu dessen Schwerpunkt der Schwerpunkt ihrer eigenen AABB die geringste Distanz verfügt. Da dieses Vorgehen nicht garantiert, daß alle zu einem Knoten zugeordneten Objekte von dessen Bounding Box umschlossen werden, wird das Bounding Volume nach der Zuordnung der Primitive an deren maximale Ausdehnungen angepaßt. Somit überlappen sich die Bounding Volumes einzelner Knoten mitunter.

Der Konstruktionsalgorithmus an sich ist zwar von rekursiver Natur. Implementiert ist er jedoch als iterative Funktion die eine explizite Stack-Datenstruktur verwendet. Der Grund hierfür liegt darin, daß der Ausführungsstack nicht groß genug ist, um einer rekursiv arbeitenden Methode den Aufbau eines tiefen Baumes zu erlauben. Die Verwendung einer rekursiven Methode würde unweigerlich zum Stack Overflow führen.

BVH-Traversierung

Der Raycasting-Algorithmus setzt die so erzeugte BVH ein, um effizienter die Entry Faces aus der Menge der äußeren Seitenflächen zu finden. Das Durchlaufen des Baumes erfolgt wie bei der Tiefensuche (*Depth First Search, DFS*) in einem Graphen in *Pre-Order* bzw. *Depth-First Order* (siehe unten) [Smi98, Hav04]. Die Suche darf allerdings nicht abgebrochen werden, sobald ein „Ziel“ (Primitiv, das vom Strahl geschnitten wird) gefunden wurde, denn die Anzahl der Entry Faces ist vor dem Ende der Traversierung nicht bekannt.

Um einen Binärbaum in Pre-Order zu durchlaufen, betrachtet man zunächst den Wurzelknoten. Wird dieser vom Strahl geschnitten, setzt man die Suche im linken Kind-Knoten/Teilbaum fort, bis man entweder einen Treffer (ein Primitiv, daß vom Strahl getroffen wird), oder einen Knoten findet, dessen Bounding Volume nicht getroffen wird. Damit ist der Abstieg in den Baum für diesen Teilbaum beendet. Die Suche wird dort fortgesetzt, wo der letzte Abstieg in den linken Teilbaum begann (*Backtracking, Trial-and-Error-Prinzip)*, und der verbliebene rechte Teilbaum wird nach derselben Methode durchsucht (Abbildung 2.21).

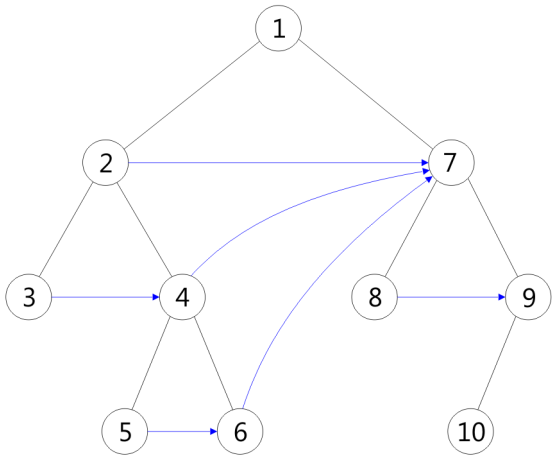


Abbildung 2.21: Traversierung eines Binärbaums in Pre-Order
Die Knoten werden in der Reihenfolge ihrer Numerierung durchlaufen. Blaue Pfeile stellen Skip-Pointer dar.

Die BVH-Implementierung des Röntgensimulators nutzt explizite *Skip-Pointer* (siehe Abbildung 2.21), die eine effizientere, iterative Traversierung des Baumes ermöglichen [Smi98]. Falls der aktuelle Knoten nicht getroffen wird, verweist der Skip-Pointer gegebenenfalls auf den nächsten zu untersuchenden Knoten.

Der Einsatz einer BVH zum Auffinden der Entry Faces verhilft dem Röntgensimulator erwartungsgemäß zu einem spürbaren Leistungsschub. Abbildung 2.19 rendert er in einigen hundert Millisekunden – inklusive Aufbau der BVH. Bei einer praxistauglicheren Bildauflösung von 1600×1200 Bildpunkten erreicht er eine Renderingleistung von etwa 280 kRays/s und ist somit um mehr als vier Größenordnungen schneller, als die naive Implementierung.

2.5.6 Weitere Optimierungsmöglichkeiten

Zwar verhelfen die hier vorgestellten Optimierungen des naiven Raycasting-Algorithmus zu brauchbarer Performance. Die Möglichkeiten für weitere Verbesserungen bezüglich der Geschwindigkeit sind damit jedoch noch nicht erschöpft.

Implizite BVH

Die BVH ist in der aktuellen Version als expliziter Baum implementiert. Die einzelnen Knoten besitzen Pointer bzw. Referenzen auf ihre AABB, ihre Kind-Knoten oder, im Falle eines Blatt-Knotens, auf eine Liste von Quads. Die eigentlichen Daten liegen folglich irgendwo verstreut auf dem (managed) Heap. Traversiert man den Baum, ergibt sich dadurch ein sehr unregelmäßiges Speicherzugriffsmuster, das praktisch keine Referenzlokalität aufweist; der Prefetcher der CPU kann nicht vorhersehen, auf welchen Speicherbereich als nächstes zugegriffen werden wird. Entsprechend verringert sich auch die Wahrscheinlichkeit, daß ein benötigtes Datum bereits im Cache zur Verfügung steht.

Dieses Problem läßt sich relativ einfach dadurch umgehen, daß man die BVH als implizite Datenstruktur in einem Array speichert [Smi98]. Allerdings muß hier eine Eigenheit des *Common Type Systems* (CTS) beachtet werden, die auch die Sprache C# betrifft [MSDN07]: Grundsätzlich unterscheidet das CTS zwischen Referenzdatentypen (*Reference Types*) und Wertedatentypen (*Value Types*). Klassen sind Referenzdatentypen, Instanzen davon landen immer auf dem managed Heap. Dem stehen `structs` gegenüber, die Wertedatentypen darstellen und immer „in-place“, d.h. je nach Kontext, in der die Allokation stattfindet, auf dem Stack (`struct` als lokale Variable) oder dem Heap (`struct` als Feld einer Instanz eines Referenztyps) angelegt werden. Da die BVH-Knoten (`BvhNode`) als Referenzdatentyp implementiert sind, könnte man nur die Referenzen selbst in einem

Array speichern. Die Knoten und deren Daten liegen weiterhin an unkorrelierten Stellen auf dem Heap. Vielmehr müßte man sowohl die Knoten als auch die von ihnen referenzierten Daten als Wertedatentypen (`struct`) reimplementieren, um die Daten sequentiell im für das Array reservierten Speicherbereich bereitstellen zu können.

Anders als in C++ besitzen C#-Structs jedoch gegenüber Klassen einige Einschränkungen. Von Seiten der Sprachdefinition ist die wichtigste davon, daß sie zwar Interfaces implementieren können, jedoch keine Vererbung unterstützen. Für die Performance einer Anwendung viel schlimmer wiegt jedoch die Tatsache, daß der aktuelle JIT-Compiler, der aus dem CIL-Code x86-Code erzeugt, offenbar Methoden, die einen `struct`-Parameter entgegennehmen, grundsätzlich nicht inlinen kann [Not04]. Dieses Problem konnte ich durch eigene Experimente mit der ursprünglichen Struct-Version von `Vector3` und dem Debugger „CorDbg" ebenfalls nachvollziehen. Aus diesem Grund ist selbst der Datentyp `Vector3` als Referenzdatentyp implementiert. Man kann nur hoffen, daß Microsoft dieses Problem mit neuen Versionen des x86-JIT-Compilers behebt.

Weitere Beschleunigungsdatenstrukturen

Eine BVH ist nicht die einzige Datenstruktur, die eine starke Beschleunigung des Rendervorgangs ermöglicht. Sehr populär sind BSP-Strukturen (*Binary Space Partitioning*); insbesondere der Spezialfall namens *kD-Tree* gilt als sehr effizient [Wal04, WFMS05]. Im Gegensatz zur BVH handelt es sich bei einem kD-Tree jedoch nicht um eine Hierarchie der Objekte einer Szene. Vielmehr wird der Raum selbst in eine hierarchisch organisierte Menge von disjunkten Unterräumen unterteilt. Raumunterteilungsverfahren wie BSP bieten für klassische Raytracer den Vorteil, daß die Traversierung der Struktur abgebrochen werden kann, sobald ein Schnitt des Strahls mit einem Szenenobjekt festgestellt wurde (*Early Ray Termination*). Da die einzelnen Unterräume nach Distanz zum Strahlursprung sortiert sind, kann folglich kein weiterer, näher am Strahlursprung gelegener Schnittpunkt existieren.

Dieser Vorteil zahlt sich jedoch nur für Front-to-Back-Rendering aus. Für das Rendering semi-transparenter Volumen kann diese Eigenschaft nicht sinnvoll genutzt werden; schließlich müssen alle Schnittpunkte des Strahls mit den Objekten der Szene bestimmt werden und nicht nur der Schnittpunkt mit der geringsten Distanz zum Strahlenursprung.

Ein weiteres Problem von BSP-Strukturen ist, daß oftmals mehrere Blatt-Knoten der Hierarchie ein und dasselbe Primitiv referenzieren. Um mehrfache Schnittests mit demselben Primitiv vermeiden zu können, benötigt man einen Mechanismus, der feststellen kann, ob ein Schnittest mit einem Primitiv bereits durchgeführt wurde (*Mailboxing*). Eine andere Möglichkeit besteht darin, die be-

troffenen Primitive zu zerlegen, was aber den Speicherverbrauch entsprechend erhöht.

Die *Bounding Interval Hierarchy* (BIH) hat nicht mit diesem Problem zu kämpfen [WK06]. Jedes Primitiv wird – wie auch bei einer BVH – genau einmal referenziert. Wie bei der klassischen BVH muß ein Schnittest grundsätzlich für alle Kind-Knoten eines vom Strahl geschnittenen Knoten durchgeführt werden, da sich die repräsentierten Volumina auch in einer BIH überlappen können. Ist jedoch bereits mindestens ein Schnittpunkt bekannt, ist es möglich, einzelne Unterbäume auszulassen, sofern diese Teile der Szene umschließen, die weiter entfernt als der berechnete Schnittpunkt sind. Inwiefern der Einsatz einer BIH dem Röntgensimulator zu mehr Leistung verhelfen könnte, läßt sich vermutlich nur durch eine testweise Implementierung klären.

Strahl-Kohärenz und SIMD

Bei einem Raytracer vermutet man intuitiv, daß dessen Geschwindigkeit in erster Linie von der Prozessor-, genauer der FPU-Leistung, abhängt. Schließlich sind Schnittests mit AABBs und Dreiecken, die sicherlich eine hohe Dichte an FPU-Code aufweisen, mit die teuersten Operationen, die ein Raytracer durchführt. Diese intuitive Einschätzung erweist sich jedoch als falsch. Tatsächlich ist die Leistungsfähigkeit von Raytracern auf modernen Rechnern in erster Linie durch die verfügbare Spcicherbandbreite und die Zugriffslatenz beschränkt [Wal04]. Speicherzugriffe erfolgen beim Raytracing chaotisch und quasi-zufällig.

Die Idee beim *Packet Tracing* ist es, ein ganzes Bündel kohärenter (im Sinne von: „direkt benachbart“, „schneiden vermutlich die gleichen Primitive“) Strahlen auf einmal zu verfolgen, anstatt jeden Strahl einzeln für sich. Durch die Kohärenz der Strahlen ergeben sind meist sehr ähnliche oder gar die gleichen Speicherzugriffe. Sobald einmal auf die benötigten Daten zugegriffen wurde, kann man für alle folgenden Zugriffe den direkteren und wesentlich schnelleren Weg über den Cache gehen, um an die Daten zu gelangen. Die Verwendung dieser Maßnahme dämmt die Abhängigkeit von der Hauptspeicherbandbreite und der Speicherlatenz enorm ein.

Bei Software-Raytracern, die die *SIMD*[24]-Einheiten eines Prozessors wie z. B. *SSE*[25] einsetzen, findet man häufig Paketgrößen von 2×2 Strahlen. Dies hängt primär mit der Funktionsweise von SSE zusammen, dessen 128-Bit-breite Register jeweils genau vier Single-Precision-Werte aufnehmen können. Auch größere Pakete sind machbar, erhöhen aber auch den Overhead.

24 SIMD: *Single Instruction, Multiple Data*; SIMD-Einheiten wenden dieselbe Operation gleichzeitig auf mehrere Daten an

25 SSE: *Streaming SIMD Extensions*; ein von Intel eingeführter SIMD-Befehlssatz für x86-/x64-Prozessoren

Während Packet Tracing direkt in C# umsetzbar ist, muß man, um SSE verwenden zu können, zwingend „Native Code“ schreiben. Zwar ist es möglich, aus managed Code heraus unmanaged Code zu rufen. Das geschieht entweder direkt mittels des *Plattform-Invoke*-Mechanismus (*P/Invoke*) oder mittels eines in C++/CLI implementierten Wrappers. Der durch das zwingend notwendige Marshalling entstehende Overhead ist jedoch nicht zu unterschätzen. Erst bei umfangreicheren Operationen lohnt sich ein solcher Umweg.

2.6 Ergebnisse

In diesem Abschnitt geht es darum, die durch den Röntgensimulator erzeugbaren Bilder qualitativ zu bewerten. Das Rendering der synthetischen Röntgenbilder übernahm wiederum ein Athlon 64 X2 mit 2 GiB RAM.

2.6.1 Vergleich mit konventionellen Visualisierungsmethoden

Zunächst wollen wir die aus den Simulationsdaten berechneten Volumenbilder mit ANSYS-typischen Plots vergleichen, um die Qualität und Gültigkeit der erzeugten Bilder beurteilen zu können.

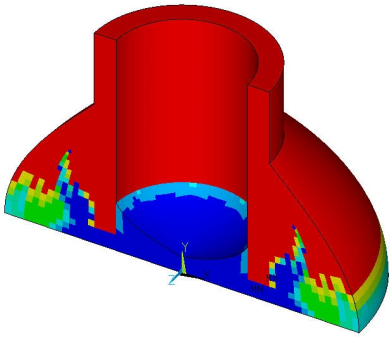
Während der frühen Phase der Entwicklung des Röntgensimulators kam zu Testzwecken in erster Linie ein Frakturmodell zum Einsatz, das zusätzlich zur Rotationssymmetrie auch die Spiegelsymmetrie entlang der Osteotomie-Ebene ausnutzt. Es handelt sich also um ein „Viertel-Modell“. Mit einer durchschnittlichen Kantenlänge der Hexaeder-Elemente von 0,8 mm ist es zudem relativ grob aufgelöst; die resultierenden 28 672 Elemente genügen jedoch für eine erste Einschätzung der Funktionstüchtigkeit des entwickelten Programms.

Zu Vergleichszwecken habe ich versucht, eine ähnliche Perspektive zu wählen, wie sie auch die Heilungssimulation zur Ausgabe der Oberflächenbilder verwendet. Offensichtlich funktioniert die Abbildung der Knochenkonzentrationen auf Röntgenschwächungskoeffizienten bzw. Grauwerte wie gewünscht (siehe Abbildung 2.22).

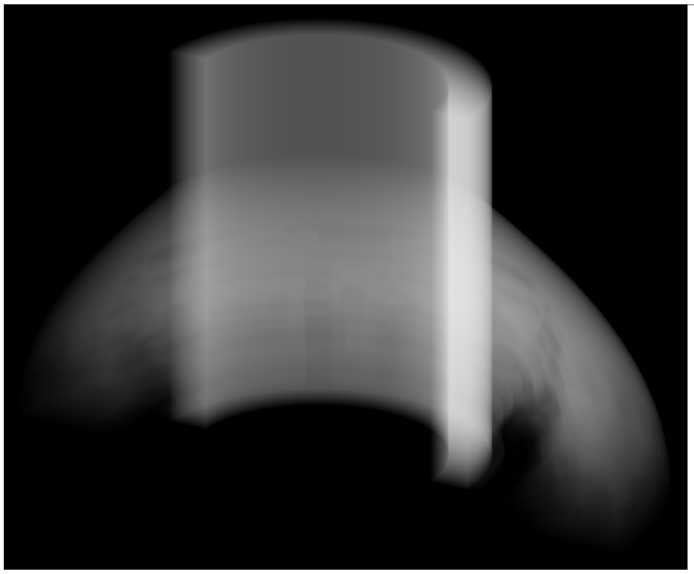
Abbildung 2.23 vergleicht Röntgenbilder unterschiedlicher Perspektive mit ANSYS-Plots (a) der Knochenkonzentrationen zu verschiedenen Zeitpunkten der Heilung (Simulationsschritte 1, 31 und 42, axiale Last). Das Modell ist nun jedoch ein „Halb-Modell“, da es nur die Axialsymmetrie ausnutzt. Die Auflösung des FE-Meshes liegt im Vergleich zum FE-Modell aus Abbildung 2.22 mit einer durchschnittlichen Kantenlänge von 0,5 mm deutlich höher, die Anzahl der Elemente beträgt bei diesem Modell 219 024 Hexaeder-Elemente. Drei Röntgenbil-

der zeigen den jeweiligen Heilungszustand aus drei verschiedenen Perspektiven: (b) gewährt einen leicht versetzten Blick, (c) stellt das Frakturmodell von einer seitlichen Ansicht dar und (d) zeigt die schon aus Abbildung 2.22 bekannte Frontalansicht.

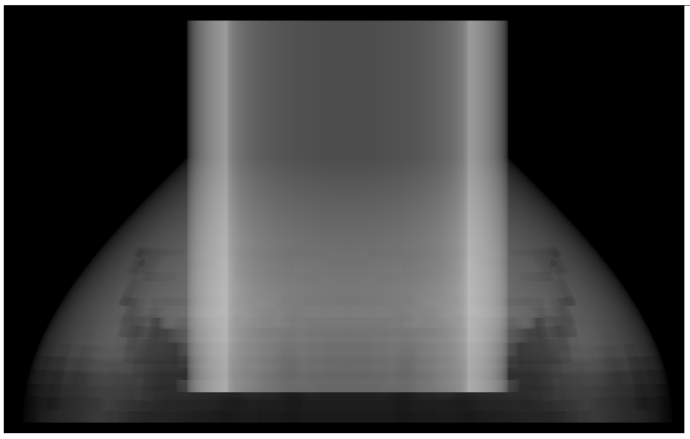
Auch auf diese Bilder zeigt sich die erwartete Übereinstimmung mit den ANSYS-Grafiken. Insbesondere auf den Bildern der 42. Iteration ist sehr schön die entstandene Überbrückung zwischen proximaler und distaler Kallushälfte zu erkennen.



ANSYS-Plot der Knochenkonzentrationen, Heilungsschritt 35, Elementgröße 0,8 mm



synthetisches Röntgenbild, ANSYS-ähnliche Perspektive



synthetisches Röntgenbild, Frontalansicht

Abbildung 2.22: Röntgenbild des Viertel-Modells
Blaue Stellen im ANSYS-Plot sind als 0 %-ige Knochenkonzentration zu interpretieren, rot steht für Knochenkonzentrationen von mindestens 87,5 %. Bei den Röntgenbildern stehen helle Bereiche für Orte hoher Röntgenabsorption.

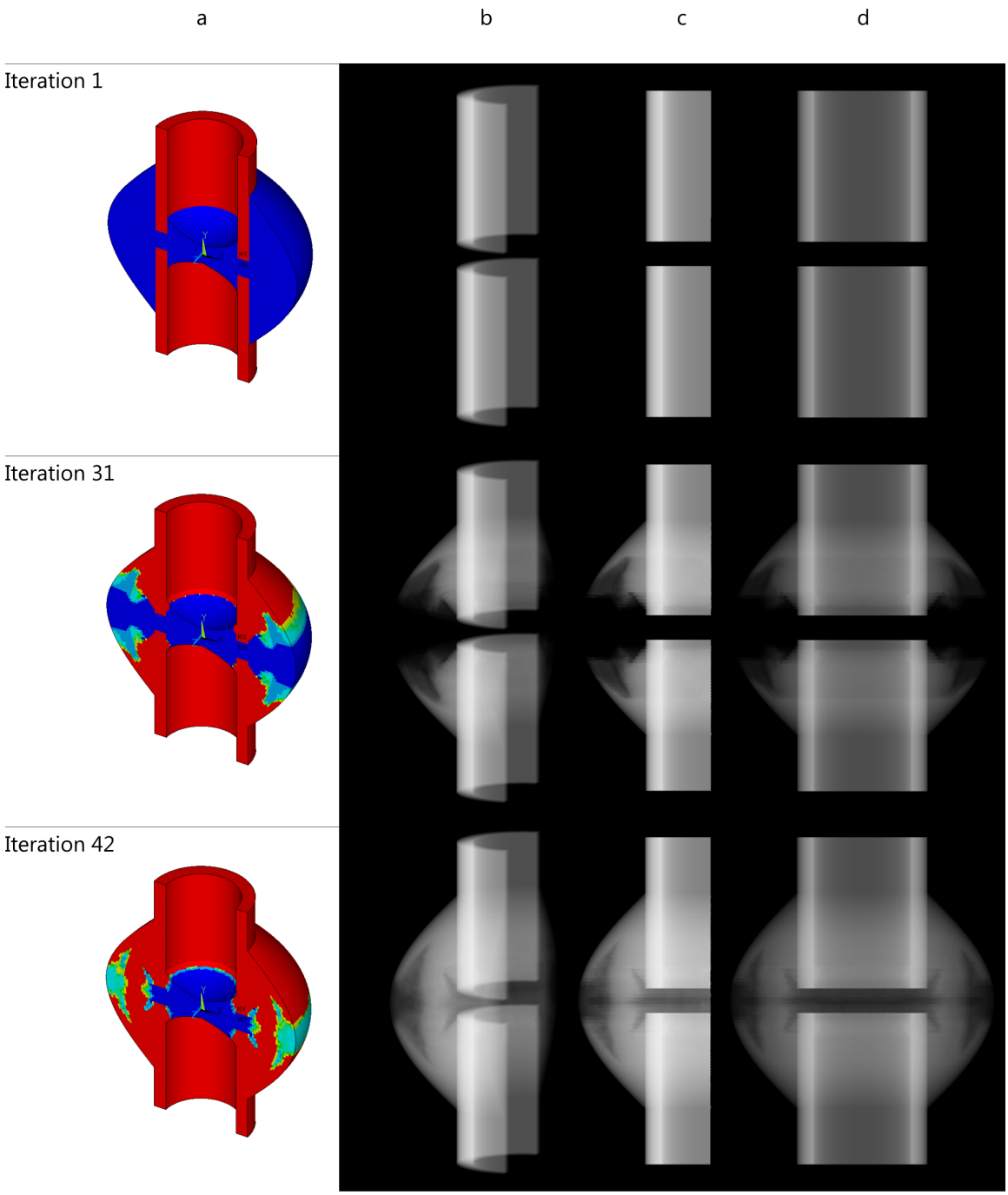


Abbildung 2.23: Röntgenbilder aus je drei Perspektiven zu drei Heilungszeitpunkten
Die ANSYS-Plots (a) stellen die durch die Heilungssimulation berechneten Knochenkonzentrationen der einzelnen Elemente zu den Simulationsschritten 1, 31 und 42 dar. (b), (c) und (d) zeigen Röntgenbilder der jeweiligen Heilungszustandes aus drei verschiedenen Perspektiven.

2.6.2 Vergleich mit realen Röntgenaufnahmen

Die mittels des Röntgensimulators gerenderten Bilder sollen den Vergleich des Heilungsmodells mit tierexperimentell gewonnenen Daten erlauben (vgl. 1.3). Um dieses Kriterium bewerten zu können, stellen wir einen qualitativen Vergleich mit Röntgenaufnahmen einer Osteotomie einer Schafstibia an. Die Aufnahmen wurden während des Heilungsvorgangs in wöchentlichem Abstand angefertigt. Dabei ist zu beachten, daß der Heilungsverlauf, den die realen Bilder zeigen, *nicht* mit denen der simulierten Heilung vergleichbar ist. Während in der Simulation eine rein axiale Last auf die Fraktur wirkt, sind die mechanischen Bedingungen des Tierexperiments nicht bekannt.

Man kann jedoch die Größe des Frakturspalts in Relation zur Dicke der Corticalis abschätzen. Der Spalt im Tierexperiment mißt etwa 1,25 Corticalis-Dicken, in der Simulation liegt das Verältnis von Corticalis-Dicke zu Frakturspalt bei ca. 1:1,6. Schon allein diese Abweichung genügt, um deutliche Unterschiede im Heilungsverlauf und insbesondere der Heilungsdauer im Vergleich zum simulierten Vorgang hervorzurufen.

In Abbildung 2.24 sieht man einen Vergleich zwischen Seitenansichten, die der Röntgensimulator aus den Daten der der Heilungssimulation erzeugt hat (Bilder jeweils rechts außen) und realen Aufnahmen einer Tibiafraktur eines Schafs (links). Die Bilder in der Mitte basieren auf den Renderings des Röntgensimulators, wurden jedoch mit Gaußschem Weichzeichner nachbearbeitet und um gleichverteiltes Bildrauschen ergänzt, um einen etwas wirklichkeitsgetreueren „Look“ zu erzeugen.

Vergleicht man das gerenderte Bild, das auf den Daten von Iteration 20 basiert, mit der Aufnahme der Schafstibia, lassen sich gewisse Gemeinsamkeiten feststellen: Neuer Knochen lagert sich an der Corticalis an (A) und wächst von dort Richtung Frakturspalt. Interfragmentär ist zu diesem frühen Zeitpunkt noch kein Knochengewebe zu finden (B).

Zu einem späteren Zeitpunkt (Iteration 42 der Simulation) kann man, zumindest auf den gerenderten Bildern, eine erste Überbrückung der beiden Kallushälften (D) erkennen. Beide, Simulation und reale Röntgenaufnahme, zeigen im lateralen Bereich des Kallus noch keine oder nur geringfügige Verknöcherung (C).

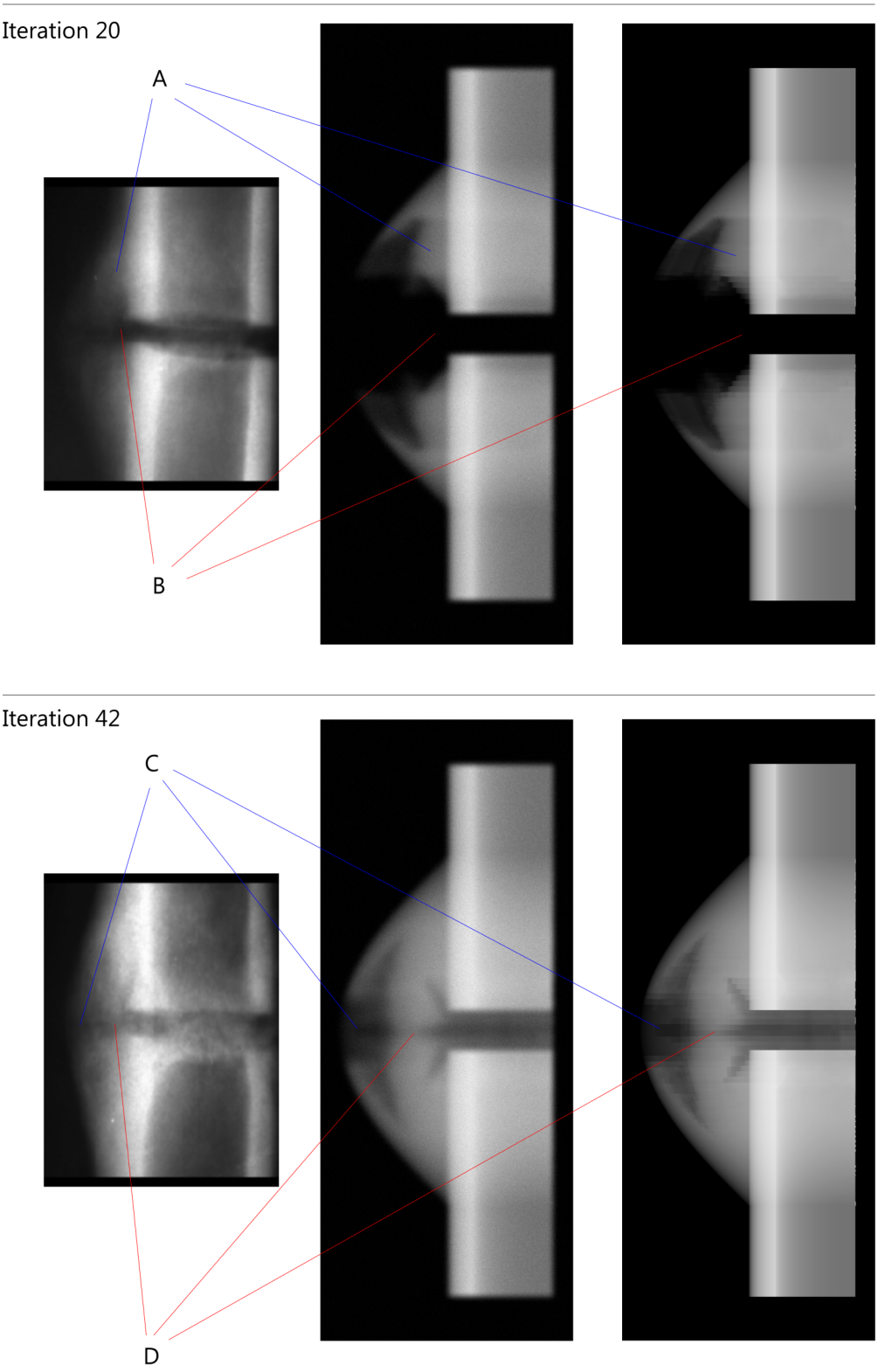


Abbildung 2.24: Synthetische Röntgenbilder im Vergleich zu Röntgenaufnahmen einer Tibiafraktur eines Schafs

2.6.3 Performance-Betrachtungen

In 2.5 habe ich verschiedene Optimierungen vorgestellt, die den Röntgensimulator zu besserer Leistung verhelfen und die Generierung von Bildern in kürzerer Zeit ermöglichen sollen. Zunächst möchte ich eine kurze Analyse wagen, inwiefern diese Anstrengungen gefruchtet haben. Als Maßeinheit für die Renderingleistung dient im folgenden die Einheit kRays/s (1000 Strahlen pro Sekunde).

Während der Entwicklung kam das Viertel-Frakturmodell als Benchmark zum Einsatz. So lassen sich im Nachhinein die durch die in den verschiedenen Versionen vorgenommenen Maßnahmen erzielten Effekte beurteilen (vgl. Abbildung 2.25). Der erste Leistungssprung gegenüber der „Ur-Version“ (Versionen vor v0010, allerdings bereits mit optimiertem Schnittest) erzielt man durch den Einsatz des in 2.5.4 beschriebenen „Nachbarschaftstraversals“. In v0010 kommt allerdings noch eine vereinfachte Version dieses Algorithmus zum Einsatz. Dennoch ist der Speedup von etwa Faktor 10 bereits nicht zu verachten.

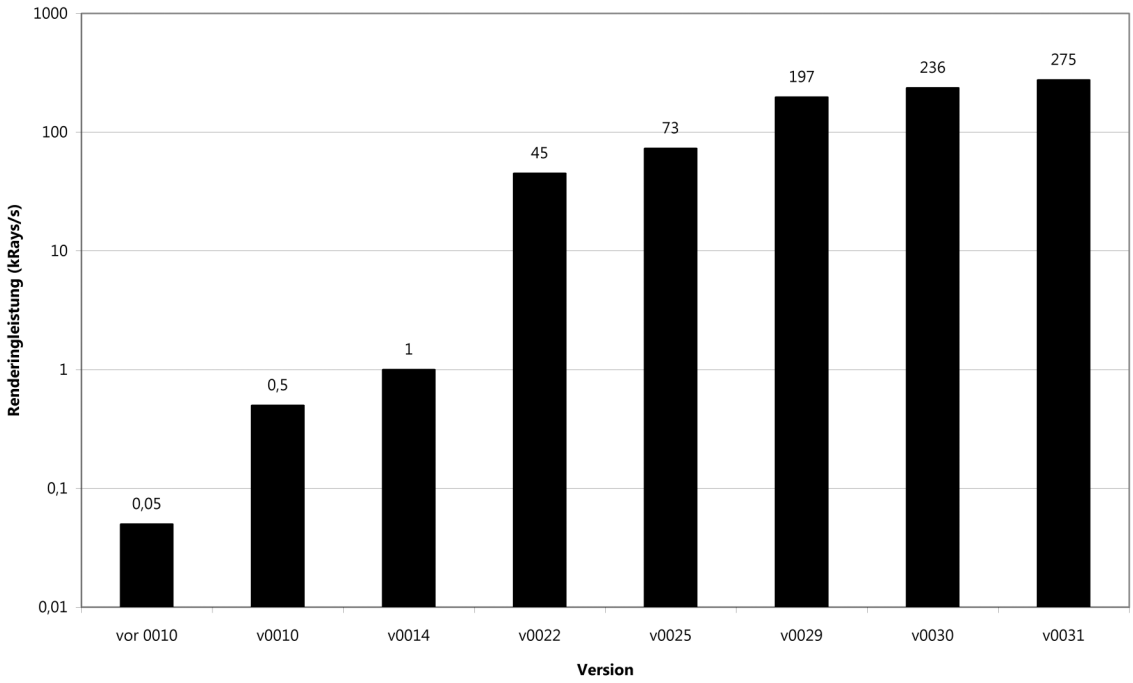


Abbildung 2.25: Auswirkungen der Optimierungen
Die in Abschnitt 2.5 beschriebenen Maßnahmen führen gegenüber der unoptimierten Version (vor v0010) zu einer Leistungssteigerung von mehr als drei Größenordnungen.

Version v0014 führt erstmals AABBs als Bounding Volumes ein und kann die Leistung dadurch nochmals verdoppeln. Den größten Speedup erfährt der Röntgensimulator durch die Implementierung der BVH (v0022). Diese Maßnahme beschleunigt das Rendering der Testszene um Faktor 45. Weitere 62 % Leistung ge-

winnt man durch die Verwendung von gemeinsamen Seitenflächen für Flächen, die sich zwei Hexaeder teilen (v0025). In Version v0029 sind diverse kleinere Optimierungen zusammengefaßt, die sich aber dennoch zu einer ansehnlichen Leistungssteigerung von fast Faktor 2,7 gegenüber v0025 kumulieren. v0030 nutzt Skip-Pointer zur Traversierung der BVH (+20 %).

v0031 führt ein optimiertes Handling der `Ray`-Objekte ein, dessen Auswirkungen jedoch nur auf Prozessoren mit Intels „Netburst"- und „Core"-Prozessorarchitekturen wirklich deutlich zu erkennen sind: Durch das Wegfallen der Heap-Allokationen für jeden neuen Strahl, erfährt der Röntgensimulator auf diesen Architekturen gegenüber älteren Versionen eine Leistungssteigerung um Faktor 10 (Pentium 4 mit Netburst-µArchitektur) bzw. Faktor 5 (Core 2 Duo mit Core-µArchitektur). Alles in allem führen die Optimierungen zu einer Leistungssteigerung um den Faktor 5500 gegenüber der Ur-Version. Verglichen mit ersten Prototypen, die noch nicht den optimierten Strahl-Dreieck-Schnittest verwenden, liegt die erzielte Beschleunigung gar rund doppelt so hoch (ca. Faktor 11000).

Eine weitere noch ausstehende Frage ist, wie gut der Röntgensimulator mit der Komplexität der Szene skaliert. Im Diagramm (Abbildung 2.26) ist die Anzahl der gerenderten Dreiecke gegenüber der erreichten Renderingleistung (Median aus vier Messungen) abgetragen.

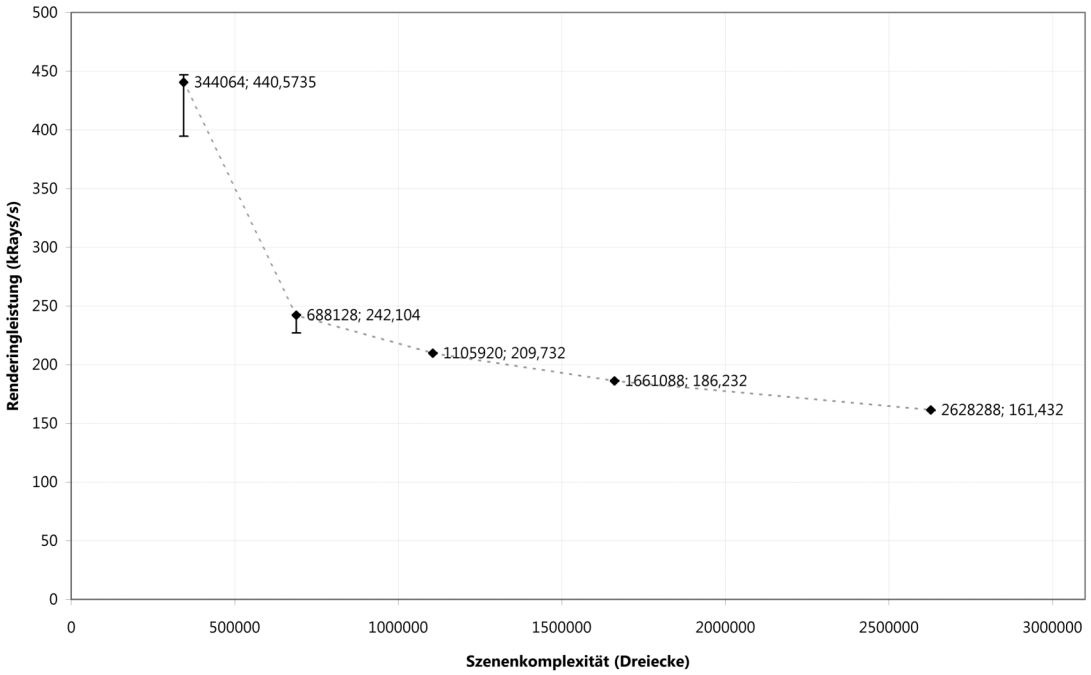


Abbildung 2.26: Skalierung des Renderinggeschwindigkeit mit der Szenenkomplexität
Dargestellt ist der Median aus vier Messungen.

Die Auflösung der Bilder lag bei jeweils 1200×1600 Bildpunkten, es waren also je Bild 1 920 000 (Primär-)Strahlen zu verfolgen. Die Perspektive entspricht der aus den Abbildungen 2.22 und 2.23 bekannten Frontalansicht. Die erste Szene arbeitet mit dem Viertel-Modell und Elementen von 0,8 mm Kantenlänge. Alle weiteren Messungen wurden mit dem halben (axialsymmetrischen) Modell durchgeführt, die Steigerung der Szenenkomplexität geht allein auf eine Schrittweise Verkleinerung der finiten Elemente zurück (0,8, 0,7, 0,6 und 0,5 mm Kantenlänge).

Zwar läßt sich aus den Meßdaten nicht zwangsläufig eine logarithmische Skalierung der Rendergeschwindigkeit mit der Zahl der Dreiecke in der Szene ableiten. Man kann aber ein gewisses asymptotisches Verhalten erkennen; auf jeden Fall bleiben die Geschwindigkeitseinbußen durch eine Erhöhung der Szenenkomplexität im Rahmen des Erwarteten und praktisch Nutzbaren: Die Szene mit 2 628 288 Dreiecken weist gegenüber der zweiten Szene mit 688 128 Dreiecken eine um den Faktor 3,8 höhere Komplexität auf. Das Rendern der komplexen Szene dauert aber mit ca. 11,9 s nur 4 s länger als das der einfachen Szene (7,9 s). Man könnte auch sagen: Die vierfache Szenenkomplexität reduziert die Renderingleistung um rund ein Drittel von 242 kRay/s auf 161 kRays/s. Dies ist eine deutliche Verbesserung gegenüber dem naiven Brute-Force-Ansatz, bei dem die Leistung auf ein Viertel des ursprünglichen Wertes einbrechen würde.

2.7 Ausblick

Die aktuelle Version des Röntgensimulators weist noch einigen Spielraum für Verbesserungen auf. Neben den in Abschnitt 2.5.6 genannten möglichen Maßnahmen zur weiteren Steigerung der Rendergeschwindigkeit, bietet sich auch die ein oder andere funktionale Erweiterung an.

Das momentan zum Einsatz kommende minimalistische Materialmodell ist auf die Modellierung von ANSYS-Frakturheilungsszenen spezialisiert. Es unterscheidet lediglich zwei Materialien, nämlich kortikalen Knochen und Geflechtknochen. Es existiert nur ein einziger Parameter, der die Darstellung beeinflußt: der Absorptionskoeffizient. Oberflächeneigenschaften (Farbe, Textur, Reflektivität, etc.) kennt das Modell nicht, da sie für das Rendering röntgenähnlicher Bilder unerheblich sind. Sollte allerdings irgendwann der Wunsch aufkommen, den Röntgensimulator für das Rendering anderer Szenen erweitern zu wollen, wäre es folglich zwingend nötig, das vorhandene Materialmodell zu überarbeiten und auszubauen. Eine Möglichkeit wäre die Implementierung eines Shader-Modells, mit dem sich die Materialeigenschaften jedes Primitivs flexibel definieren ließen.

Auch die Importfunktionalität der Implementierung ist auf die Bedürfnisse der Frakturheilungssimulation bzw. deren Ausgabedaten zugeschnitten. Soll eine andere Quelle als die Frakturheilungssimulation als Datenquelle dienen, erfordert das die Entwicklung eines entsprechenden Import-Plug-Ins. Eventuell könnte man auch über ein allgemeineres Szenenbeschreibungsformat nachdenken, das die Geometrie in einem von der Quelle unabhängigen Format speichert und stärker Rücksicht auf die Anforderungen nimmt, die ein Raytracer an die Szenenbeschreibung stellt.

Neben diesen, die Kernfunktionalität betreffenden, Verbesserungsmöglichkeiten, ist das Benutzerinterface ein weiterer Kandidat für zukünftige Erweiterungen. Für die Kamerapositionierung könnte sich eine schnelle, evtl. interaktive Vorschaufunktion als nützlich erweisen. Zur komfortableren Generierung von „Heilungsanimationen“ ohne zusätzliche externe Scripte und Hilfsprogramme, könnte eine überarbeitete Version des Interface die direkte Eingabe eines ganzen Satzes an Dateien, die die Materialeigenschaften der Elemente beschreiben, ermöglichen.

3 Automatisierte Überbrückungsdetektion

Eines Tages Empfehlungen für die optimale, patientenspezifische Behandlung von Knochenfrakturen geben zu können, die auf den Ergebnissen einer Simulation des Heilungsvorgangs basieren, ist eine der Visionen, die die Entwicklung der Frakturheilungssimulation maßgeblich vorangetrieben haben. Um dieses Fernziel in erreichbare Nähe zu rücken, ist ein Verständnis dafür, wie sich eine Veränderung der zahlreichen Parameter auf den (simulierten) Heilungsvorgang auswirkt, unabdingbar. Von besonderem Interesse ist hier die Beeinflussung der Heilungsdauer selbst. Darunter versteht man die Zeitspanne, bis die Fraktur im klinischen Sinne als „geheilt“ gilt.

Sowohl die tiefgehende Analyse der Zusammenhänge als auch die daraufhin folgende numerische Optimierung der Simulationsparameter, um die Heilungsdauer zu minimieren, erfordert eine eindeutige, mathematisch faßbare Definition des Zielzustandes der Simulation. Zudem benötigen Optimierungsverfahren wie Gradientenabstieg oder Simulated Annealing eine große Menge an Stichproben („Samples“), in diesem Fall die Ergebnisse von Simulationsläufen mit variierten Parametern. Dies macht eine automatisierte Bestimmung der Heilungsdauer notwendig.

In den ersten Abschnitten dieses Kapitels stelle ich zuerst eine mögliche Formalisierung der klinischen Definition für eine geheilte Fraktur vor. Die Umsetzung in ein lauffähiges Programm, die Integration des Programms in die bestehende Simulation, sowie die Betrachtung der Ergebnisse, sind Thema der darauf folgenden Abschnitte.

3.1 Formalisierte Definition des Heilungszustandes

Eine Fraktur gilt im klinischen Sinne als geheilt, wenn die beiden Cortex-Enden über das im Kallus neugebildete Knochengewebe ausreichend stark verbunden

sind. Wann eine knöcherne Überbrückung als „ausreichend stark“ gilt, hängt in der Praxis von ihrem Erscheinungsbild auf Röntgenaufnahmen ab. In der Klinik gilt die Regel, daß auf zwei Röntgenbildern, deren Abbildungsebene orthogonal zueinander steht, drei von vier Cortices überbrückt sein müssen, um eine Fraktur als geheilt bezeichnen zu können[26].

Die Umsetzung dieses Kriteriums bei der manuellen Analyse der Simulationsergebnisse hinsichtlich der prognostizierten Heilungsdauer erfolgte bisher schlicht durch Betrachten der von der Heilungssimulation für jede Iteration ausgegeben Grafiken, die die Verteilung der Knochenkonzentrationen veranschaulichen. Sobald eine Überbrückung der beiden Kallushälften durch Elemente einer bestimmten, willkürlich festgelegten Knochenkonzentration (derzeit 88,9 % im 2D-Modell bzw. 87,5 % im 3D-Fall) hergestellt ist, gilt die Fraktur als geheilt (vgl. Abbildung 3.1).

Für eine automatisierte Überbrückungserkennung ist diese Vorgehensweise zu ungenau. Die Frage ist, wie das Kriterium der Überbrückung der beiden kortikalen Enden in eine eindeutige Definition zu überführen ist, die auch durch ein Computerprogramm umsetzbar ist, das die Heilungsdauer in Form eines skalaren Wertes bestimmen und an die Simulation zurückliefern kann.

Die Information über die Verteilung der Knochenkonzentration ist unmittelbar für jedes Element erreichbar. Zur Ermittlung des Heilungszustandes ist der Umweg über eine (z. B. simulierte) Röntgenaufnahme, aus der man mittels Bildanalyse wiederum auf Knochendichten schließen könnte, daher nicht erforderlich. Sehr wohl erfordert die Analyse des Heilungszustandes, den das Modell repräsentiert, eine geeignete Abstraktion dieses Modells.

26 Persönliche Mitteilung von Prof. Claes (03. Juli 2007)

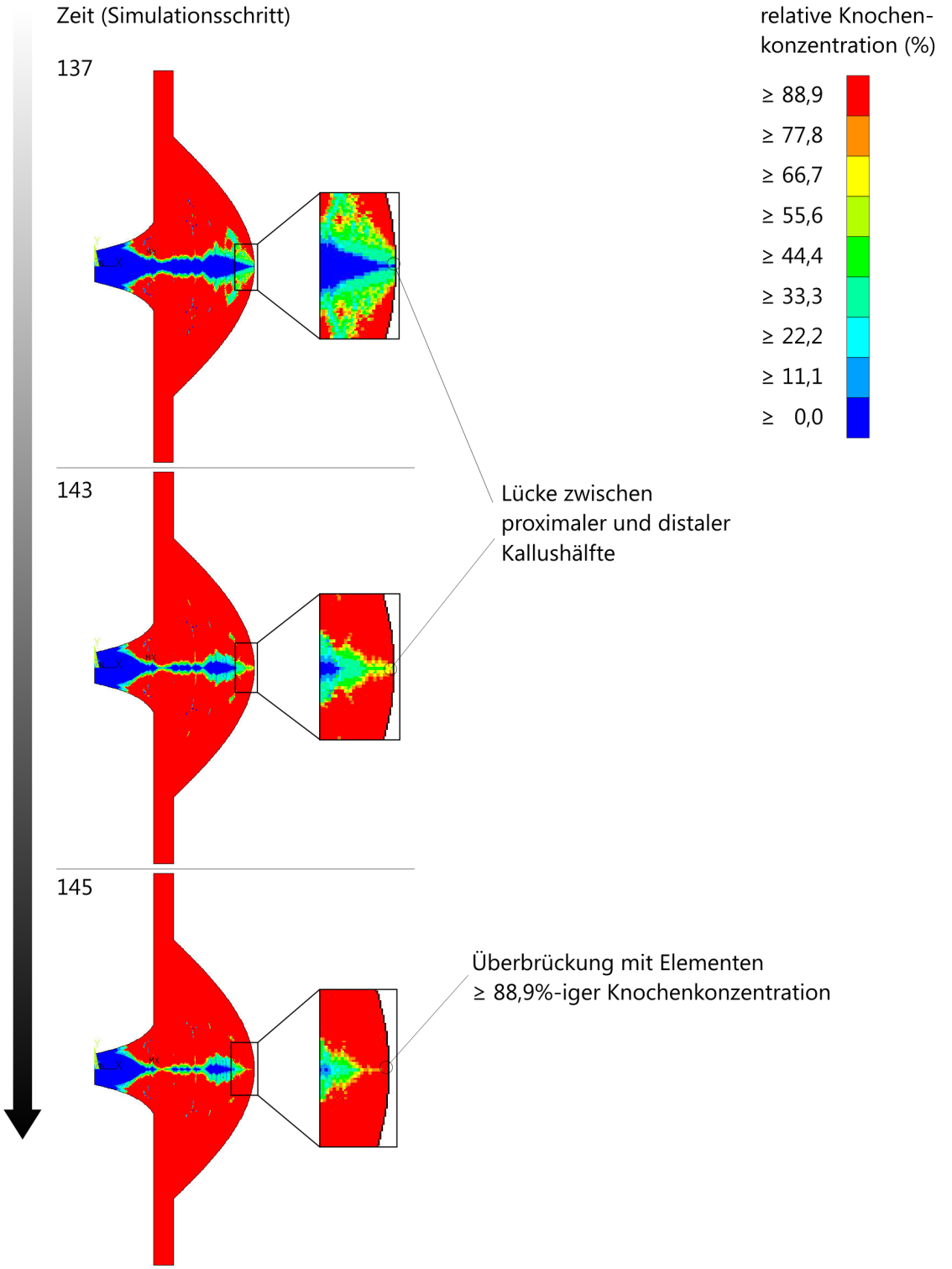


Abbildung 3.1: Visuelle Beurteilung des Heilungszustandes (2D-Frakturmodell)
Ab Iteration 145 kann man eine Überbrückung der Kallushälften mit Elementen hoher Knochenkonzentration (rot) erkennen. Dieser Zustand gilt als „geheilt".

3.1.1 Das Frakturheilungsmodell als Graph

Bei der Bestimmung des Heilungszustandes der Fraktur interessiert uns letztlich nur der Zusammenhang zwischen zwei Komponenten des Modells. Die exakte Geometrie der finiten Elemente bzw. der des Modells ist dafür unerheblich. Von Interesse sind die Beziehungen der Elemente zueinander.

Ein *Graph* ist eine abstrakte Datenstruktur, die prädestiniert ist für die Modellierung von Relationen aller Art zwischen einer beliebigen Zahl von Objekten. Die folgenden Erläuterungen zu ausgewählten Grundbegriffen der Graphentheorie basieren, sofern nicht anders angeben, auf den einleitenden Ausführungen in [ST02] (Kapitel 3, S. 85ff), [Die06] (Kapitel 0.1, 0.3 und 0.4) sowie [BM76] (Kapitel 1).

Grundbegriffe der Graphentheorie

Gemäß der Graphentheorie besteht ein Graph aus zwei disjunkten Teilmengen: Die Menge der *Knoten* (auch *Vertices* oder *Ecken* genannt) repräsentiert beliebige Objekte, die in irgendeiner Form in Zusammenhang stehen können. Den Zusammenhang der Knoten untereinander modelliert die Menge der *Kanten*. Mathematisch gesehen ist eine Kante eines ungerichteten Graphen ein ungeordnetes Paar von Knoten. Handelt es sich bei dem Knotenpaar um ein Tupel, spielt also die Reihenfolge der Knoten für die Beziehung eine Rolle, spricht man von einem *gerichteten Graphen*. Die beiden Knoten der Kante müssen in diesem Fall nicht notwendigerweise verschieden sein. Die beiden Knoten, die eine Kante definieren, nennt man *benachbart* oder *adjazent*, sie sind *Nachbarn*. Ordnet man jeder Kante zusätzlich ein *Kantengewicht* zu, handelt es sich um einen *gewichteten Graphen*.

Ein *Untergraph G'* eines Graphen *G* beinhaltet eine Teilmenge der Knoten des *Obergraphen G*, sowie sämtliche auch in *G* enthaltenen Kanten zwischen den Knoten der Teilmenge. Bei einem *Teilgraphen* hingegen können zusätzlich beliebige auf der Knoten-Teilmenge definierte Kanten entfallen.

Eine Folge von paarweise verschiedenen und verbundenen Knoten bildet einen *Weg*. Existiert ein Weg zwischen zwei Knoten *s* und *g* sagt man auch, *s* ist von *g* aus *erreichbar*, beide Knoten sind *verbunden*. Ein *Pfad* ist ein Weg, in dessen Knotenfolge jeder Knoten und jede Kante maximal einmal auftaucht.

Schon die recht anschauliche Terminologie zeigt, daß es sich bei diesem Thema um ein verhältnismäßig intuitiv verständliches Konzept handelt. Auch die oftmals sehr hilfreiche Visualisierung eines Graphen fällt leicht. Knoten stellt man in der Regel als Punkte oder Kreise dar, verbindende Linien symbolisieren die Kanten (siehe Abbildung 3.2).

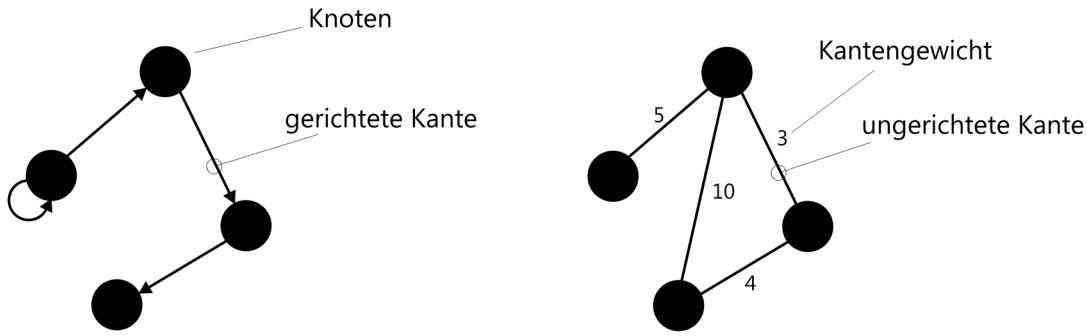


Abbildung 3.2: Visualiserung von Graphen
Links: gerichteter Graph aus vier Knoten und vier Kanten
Rechts: ungerichteter Graph mit gewichteten Kanten

Interpretation des Frakturmodells als abstrakter Graph

Auf ebenso natürliche Weise läßt sich das Konzept des Graphen auf das Frakturmodell übertragen (siehe Abbildung 3.3 und Abbildung 3.4). Die Knoten des Graphen repräsentieren die einzelnen finiten Elemente und deren physiologische und mechanische Parameter, während Kanten zwischen den Knoten die Nachbarschaftsbeziehungen der Elemente im FE-Modell widerspiegelt. Außerdem erhalten die Kanten ein Gewicht, das sich aus dem Euklidischen Abstand der Schwerpunkte der entsprechenden Elemente errechnet. Der Zweck dieser Gewichtung wird später ersichtlich werden.

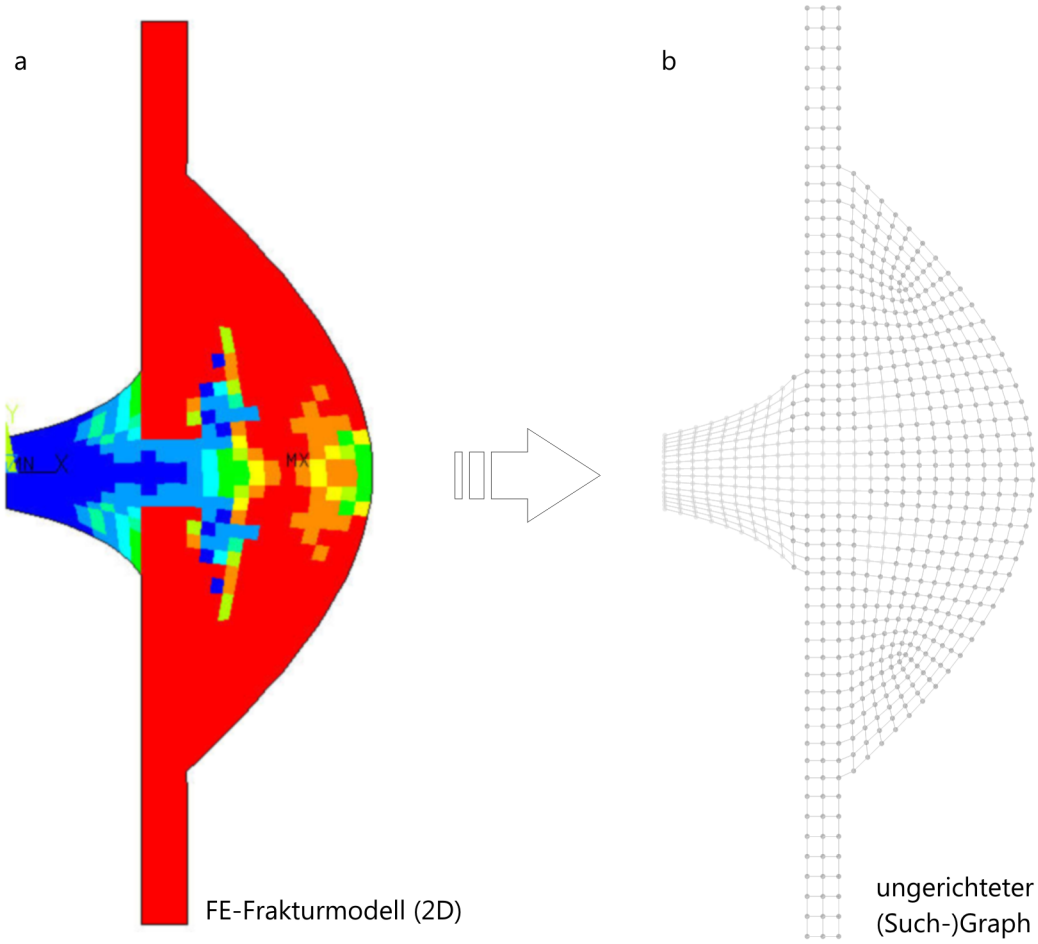


Abbildung 3.3: Modellierung des 2D-Frakturmodells als Graph
Das (hier zur besseren Übersichtlichkeit niedrig aufgelöste) zweidimensionale Frakturheilungsmodell (a) wird in einen ungerichteten Graphen (b) überführt. Dunkelgrau gefärbte Knoten repräsentieren Elemente hoher Knochenkonzentration. Die Nachbarschaften der Knoten sind durch ungerichtete Kanten dargestellt.

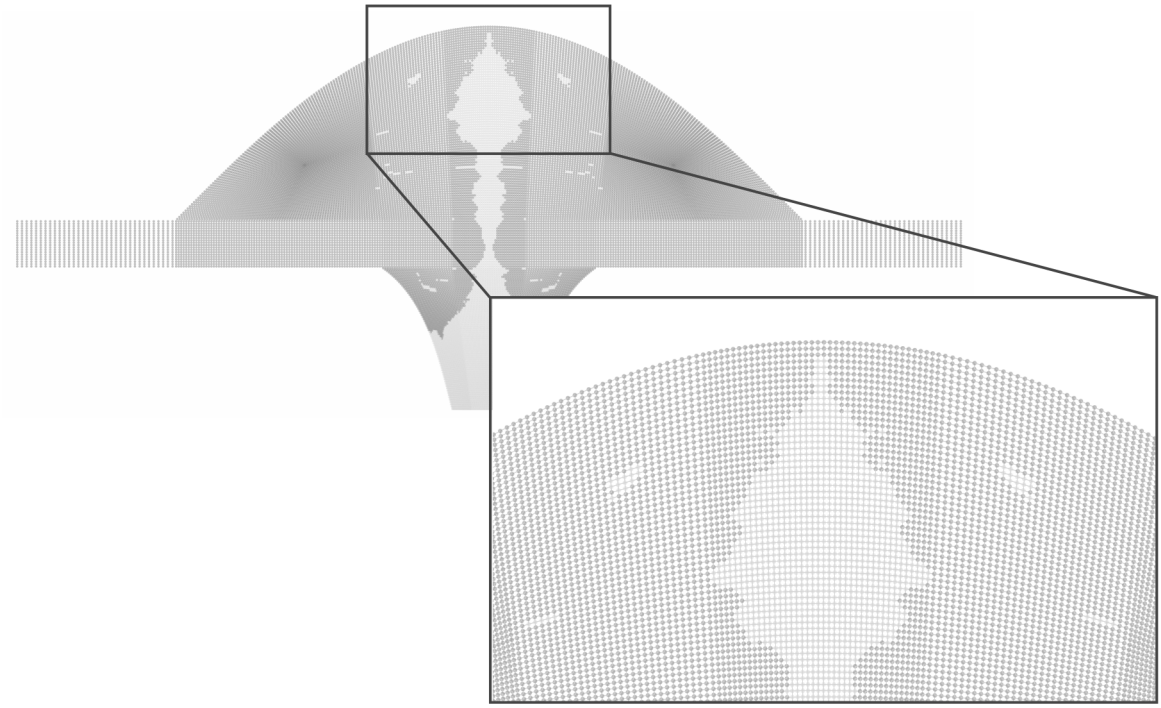


Abbildung 3.4: Graph eines hochauflösenden 2D-Modells
Im vergrößerten Ausschnitt (eingerahmt) erkennt man die Kanten zwischen den Knoten.

3.1.2 Überbrückung im Kontext des Graphenmodells

Die Fraktur gilt als geheilt, wenn eine knöcherne Überbrückung die beiden kortikalen Enden verbindet. Übertagen auf das Graphenmodell der Fraktur bedeutet dies, daß der Untergraph C, dessen Knoten die Cortex-Elemente des Frakturmodells darstellen, *zusammenhängend* sein muß. Zusammenhang im Sinne der Graphentheorie bedeutet, daß jeweils zwei verschiedene Knoten aus C über einen Pfad verbunden sein müssen. Sofern keine Überbrückung existiert, besteht der Untergraph aus zwei nicht zusammenhängenden Komponenten. Im Gegensatz zum Gesamtgraphen G, verfügen alle Knoten von C per Definition über eine initiale Knochenkonzentration von 100 %, während die Kallus-Knoten zu Beginn der Simulation eine 0 %-ige Kochenkonzentration aufweisen.

Die beiden Cortex-Komponenten für sich sind aufgrund der Modellierung der Fraktur inhärent zusammenhängend. Somit ist der gesamte Untergraph C zusammenhängend, sofern mindestens ein Pfad zwischen zwei beliebigem Knoten s und g der beiden Cortex-Hälften existiert. Damit läßt sich das Zusammenhangsproblem auf ein Wegfindeproblem in einem Graphen reduzieren, für das diverse effiziente Algorithmen bekannt sind (siehe 3.2).

Basierend auf ihrer Knochenkonzentration, markiert man dazu die Knoten des Graphen als *begehbar* (in den Abbildungen des Graphen dunkelgrau) oder *nicht begehbar* (hellgrau) und extrahiert so die Knoten, die Teil eines Pfades bzw. Überbrückung im Kontext der Frakturheilung sein können. So erhält man einen weiteren Untergraphen B. Nun wählt man aus jeder der Cortex-Hälften einen Knoten aus. Davon legt man für die Wegsuche in B einen der gewählten Knoten als Start-, den anderen als Zielknoten fest. Findet der Wegsuchealgorithmus einen Pfad zwischen den Knoten, müssen die Komponenten von C über Knoten aus B zusammenhängend sein: Die Fraktur ist geheilt (Abbildung 3.5).

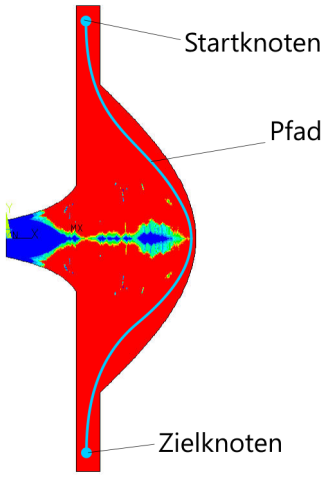


Abbildung 3.5: Erkennung des Heilungszustandes durch Wegsuche im Graphen

Zu beachten ist, daß B selbst zu diesem Zeitpunkt jedoch noch nicht zwingend zusammenhängend sein muß. Theoretisch können sich, zumindest über einen kurzen Zeitraum, abgeschlossene Inseln bilden, die nicht mit dem restlichen Untergraphen zusammenhängen. Die Existenz eines Spannbaumes für B ist kein zwingendes Kriterium für den Zusammenhang der Cortex-Elemente.

3.2 Pfadsuche in Graphen

Die folgenden Abschnitte geben einen Überblick über die grundlegenden Prinzipien der wichtigsten Suchalgorithmen für die Pfadsuche in Graphen, so daß der Leser die Entscheidung für oder wider einen der Algorithmen für die Implementierung nachvollziehen kann.

3.2.1 Bewertungskriterien

Zuvor möchte ich jedoch noch einige wenige Worte zu den Begriffen *Optimalität*, *Vollständigkeit* und (Zeit- und Platz-)*Komplexität* von Suchalgorithmen verlieren, da wir sie später für die Bewertung der Algorithmen benötigen werden.

Ein Suchalgorithmus heißt *vollständig*, wenn er, sofern mindestens ein Weg zwischen zwei Punkten existiert, auch mindestens einen Weg findet. Ist der Algorithmus zusätzlich *optimal*, ist garantiert, daß es sich bei dem gefundenen Weg um den – im Sinne der verwendeten Metrik – *kürzesten Pfad* handelt.

Das asymptotische Laufzeitverhalten eines Algorithmus schätzt man mit der Landau-Notation ab [WP07a]. Weist ein Algorithmus eine Laufzeitkomplexität von $O(g)$ auf, bedeutet dies, daß das Wachstum der Laufzeit durch das Wachstum der Funktion g beschränkt ist [ST02 S. 65]. Beispiel: die Laufzeit eines Algorithmus mit Laufzeitkomplexität $O(n^2)$ wächst maximal quadratisch mit der Größe eines veränderbaren Parameters; n könnte beispielsweise für die Zahl der Knoten in einem Graphen stehen. Analoges gilt für die Speicherplatzkomplexität.

In der Theorie sind konstante Faktoren zu vernachlässigen, d.h. die Ausdrücke $O(a \cdot f(n))$ und $O(b \cdot f(n))$ mit $a \neq b$ beschreiben äquivalente Mengen von Funktionen. Gemäß dem *linearen Speedup-Theorem* ist diese Abstraktion auch sinnvoll: Jeder Algorithmus kann durch entsprechend schnellere Hardware um einen linearen Faktor beschleunigt werden; die Skalierung des Algorithmus mit der Problemgröße ändert sich dadurch jedoch nicht. Für die praktische Anwendung hingegen können konstante Faktoren durchaus von Bedeutung sein. Auch wenn ein Algorithmus der Komplexität $O(n)$ nicht besser mit n skaliert als einer der Komplexität $O(2n)$ ist ersterer doch für jede Eingabe immer doppelt so schnell und in der Praxis ist es in der Regel von Bedeutung, ob man auf das Ergebnisse einer Berechnung eine oder doch zwei Stunden warten muß. Und noch ein Aspekt muß bei der Wahl eines Verfahrens Berücksichtigung finden: Ist der Grundaufwand, den ein Algorithmus betreibt sehr hoch, kann eine schlechter skalierender Algorithmus für kleine Problemgrößen dennoch im Vorteil sein.

3.2.2 Tiefensuche

Bei der *Tiefensuche* (*Depth First Search*) handelt es sich um ein klassisches, uninformiertes Suchverfahren. Uniformiert bedeutet, daß der Algorithmus über keinerlei Wissen über die Topologie des Graphen oder die Position des Zieles verfügt; man bezeichnet solche Suchverfahren auch als „blind“ [AI03].

Zunächst wählt man einen Startknoten. Die Tiefensuche expandiert rekursiv jeweils einen Nachfolger des aktuellen Knotens. Handelt es sich bei dem Nachfolgerknoten um das Ziel, ist die Suche beendet. Besitzt der aktuelle Knoten keinen weiteren noch nicht untersuchten Nachbarknoten, fährt die Suche mit dem nächsten Knoten auf dem Stack fort, der weitere, noch nicht untersuchte Nachbarknoten besitzt (*Backtracking*).

Wie sich die Tiefensuche verhält, ist in einem starken Maße von der Struktur des Graphen abhängig. Mitunter wirkt die Wegwahl völlig zufällig, was aufgrund der fehlenden Richtungsinformation aber nicht weiter verwundert (siehe Abbildung 3.6). Für endliche Graphen ist die Tiefensuche vollständig, sofern Zyklen durch einen entsprechenden Test erkannt werden. In diesen Fällen findet die Tie-

fensuche immer eine Lösung, falls mindestens eine solche existiert. Beim gefundenen Pfad handelt es sich jedoch in der Regel nicht um einen optimalen Pfad.

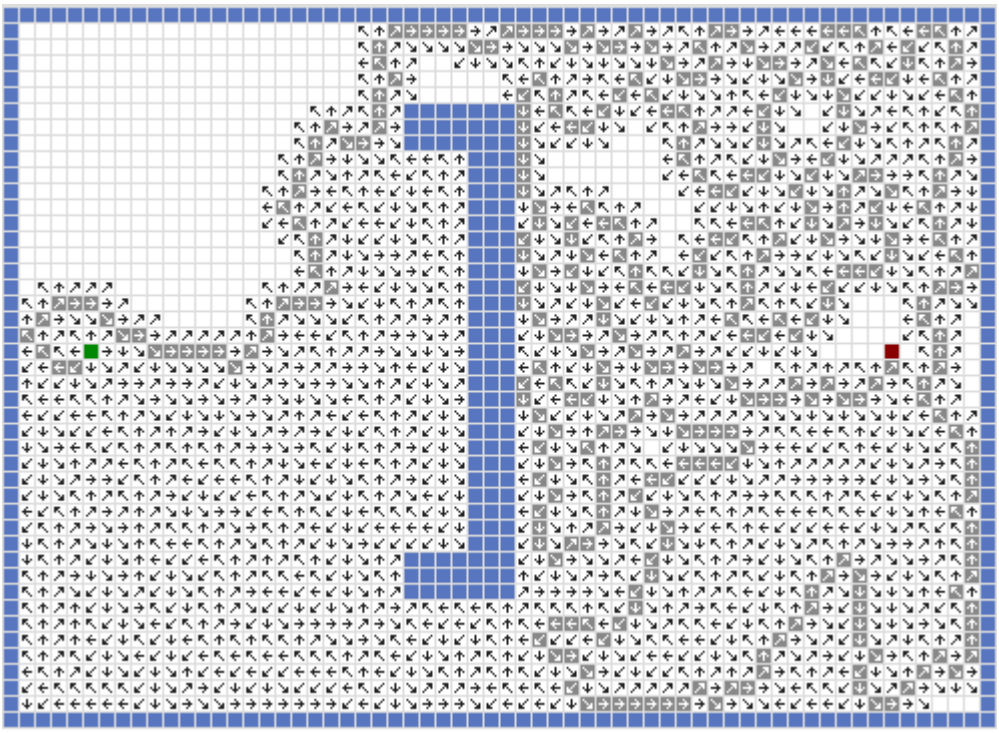


Abbildung 3.6: Depth First Search
Der Startknoten ist grün, das Ziel rot dargestellt, Hindernisse sind in Blau gehalten.
Die grau hinterlegten Felder repräsentieren Knoten des gefundenen Pfads.
(Quelle: http://www.stefan-baur.de/cs.web.mashup.pathfinding.html)

Die Laufzeitkomplexität liegt im *worst case* bei $O(b^m)$. Hierbei steht b für den maximalen Verzweigungsgrad (*Branching Factor*) des Suchbaumes, der im Graphen der maximalen Anzahl an Nachfolgeknoten entspricht; m bezeichnet die maximale Tiefe des erzeugten Suchbaums. Die Platzkomplexität skaliert linear mit $O(bm)$ [RN02]. Die Tiefensuche eignet sich somit insbesondere für Situationen, in denen es auf sparsamen Umgang mit den Speicherressourcen ankommt.

3.2.3 Breitensuche

Auch die *Breitensuche* (*Breadth First Search*) bedient sich keinerlei Information über den Suchraum. Sie hat daher starke Ähnlichkeiten mit der Tiefensuche, verwendet jedoch als zentrale Datenstruktur, um noch nicht untersuchte Knoten zu speichern, eine *FIFO-Queue* (Warteschlange).

Die Suche funktioniert nach folgendem Schema [RN02]: Der Startknoten wird als aktueller Knoten ausgewählt und in die Warteschlange eingereiht. Solange die

Warteschlange nicht leer ist, wiederholt man folgende Schritte: Man entfernt den nächsten Knoten aus der Warteschlange und prüft, ob es sich um das Ziel handelt. In diesem Fall ist die Suche zu Ende und ein Pfad existiert. Andernfalls fügt man die Nachbarknoten des aktuellen Knotens in die Warteschlange ein.

Als problematisch an der Breitensuche erweist sich insbesondere die Platzkomplexität von $O(b^d)$. d gibt hierbei die Länge (Anzahl der Knoten, über die der Pfad führt) des ersten gefundenen Pfades zwischen Start- und Zielknoten an. Für große Suchräume ist die Breitensuche somit ungeeignet [RN02]. Sie ist jedoch in jedem Fall vollständig und liefert als Ergebnis, falls ein Pfad existiert, immer den Pfad mit der geringst-möglichen Anzahl an Schritten. Sofern die Kosten von einem Knoten zu einem Nachbarknoten für alle Kanten gleich und konstant sind, entspricht dieser Pfad auch dem optimalen Pfad. Andernfalls ist dies nicht zwingend der Fall.

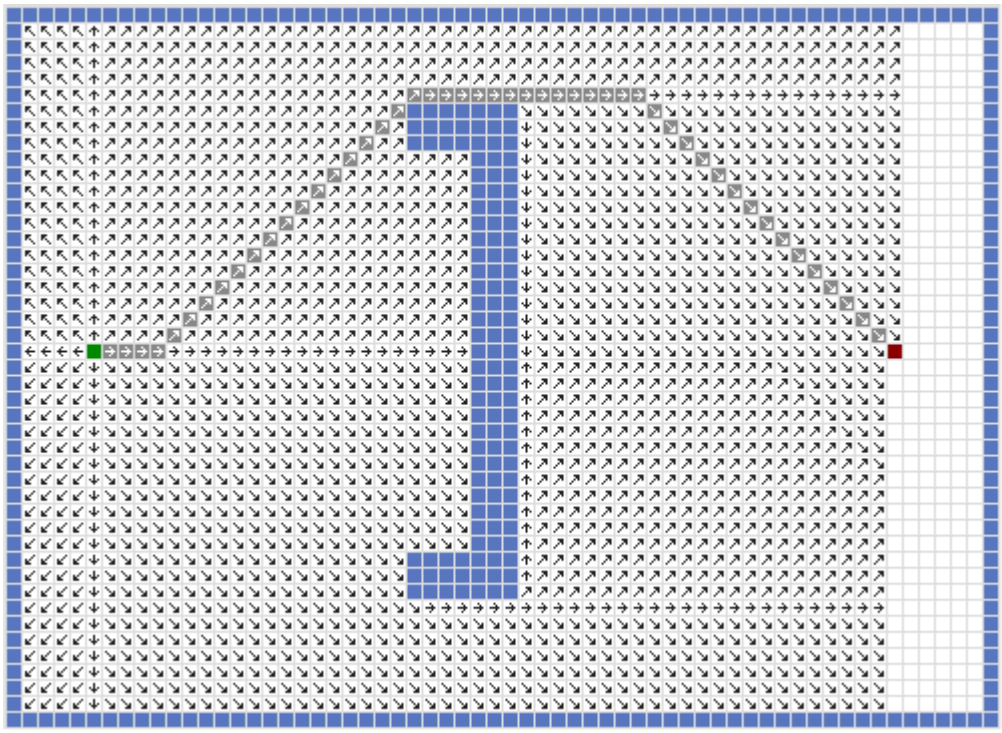


Abbildung 3.7: Breadth First Search
(Quelle: http://www.stefan-baur.de/cs.web.mashup.pathfinding.html)

3.2.4 Uniforme-Kosten-Suche und der Algorithmus von Dijkstra

Kommen wir nun zu den intelligenter vorgehenden Suchverfahren. Auch wenn im folgenden nur von der *Uniforme-Kosten-Suche*[27] (*Uniform-Cost Search, UCS*) die Rede ist, treffen alle folgenden Erklärungen auch auf den *Algorithmus von Dijkstra* zu, der eine Verallgemeinerung der Uniforme-Kosten-Suche darstellt: Während die Uniforme-Kosten-Suche terminiert, sobald sie den Zielknoten entdeckt, berechnet Dijkstras Algorithmus für sämtliche Knoten des Graphen den jeweils kürzesten Pfad vom Startknoten aus und löst damit das *Single Source Shortest Path Problem*.

Bei der Uniforme-Kosten-Suche kommt eine Kostenfunktion zum Einsatz, die jedem Knoten n einen Wert $g(n)$ zuordnet; $g(n)$ drückt die absolute Distanz eines Knoten zum Startknoten aus. Dieser Wert bestimmt, mit welcher Priorität der Knoten zu untersuchen ist. Folglich sortiert man die zu untersuchenden Knoten des Graphen in eine Prioritätswarteschlange (*Priority Queue*) ein, die die Knoten nach ihrem *f*-Wert sortiert. Zu Beginn sind die Distanzen noch unbekannt und werden mit $+\infty$ initialisiert [WP07b].

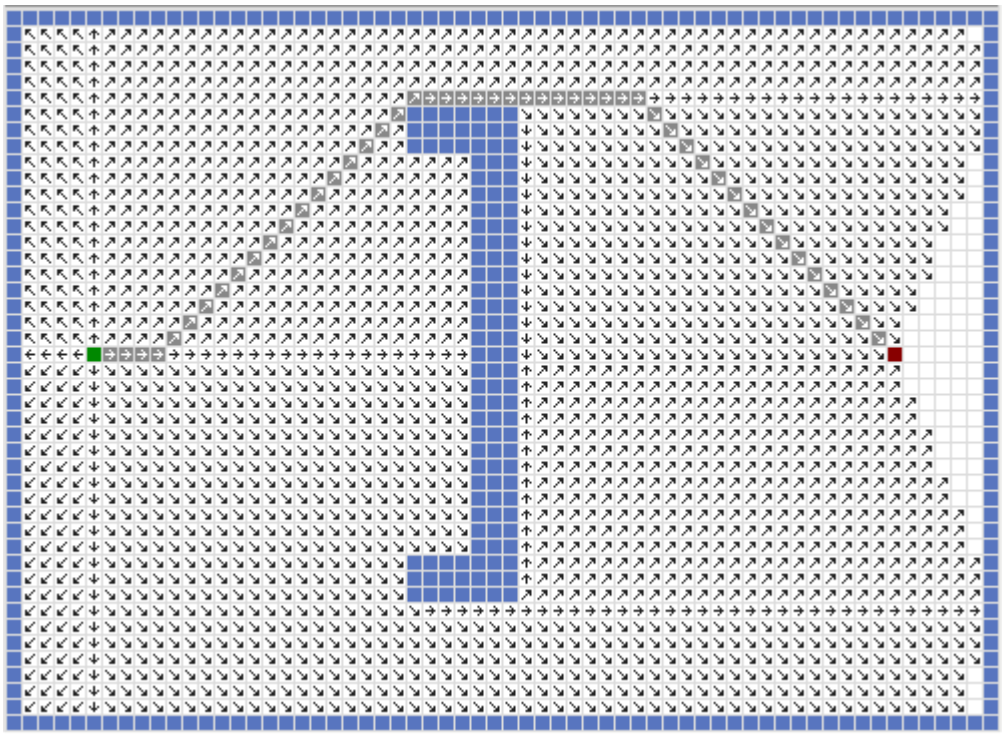


Abbildung 3.8: Uniforme-Kosten-Suche (Uniform-Cost Search)
(Quelle: http://www.stefan-baur.de/cs.web.mashup.pathfinding.html)

27 meist falsch als *„uniforme Kostensuche"* übersetzt

Der Ablauf ist ähnlich wie bei der Breitensuche. Allerdings wird nun immer der Knoten v als nächstes expandiert, der die geringsten absoluten Kosten zum Ziel aufweist, d.h. in der Prioritätswarteschlange an erster Stelle steht. Für alle Nachbarknoten v' des aktuell besuchten Knoten v prüft der Algorithmus, ob sich deren Distanz zum Ziel verringert, wenn der Weg vom Start zu v' über v führt. Falls dies zutrifft, wird der g-Wert von v' entsprechend modifiziert und der Vorgänger-Zeiger (darüber kann im Rekonstruktionsschritt der Pfad rekonstruiert werden) des Knoten v' auf den Vorgängerknoten v gesetzt.

Da die Uniforme-Kosten-Suche immer den Knoten mit den geringsten absoluten Pfadkosten expandiert, garantiert sie die Optimalität des gefundenen Pfades. Die Laufzeit- und Platzkomplexität beträgt jeweils $O(b^d)$ [RN02]. Der Algorithmus untersucht alle Knoten n, für die die Kostenfunktion $g(n)$ kleiner oder gleich der Pfadlänge des bzw. eines kürzesten Pfades ist.

Eine gewisse Ähnlichkeit zu diesem Suchverfahren weist auch *Lees Maze-Routing-Algorithmus* auf, der das Problem löst, die kürzeste Verbindung zwischen zwei Lötpunkten auf einer Leiterplatine zu finden. Die Leiterplatine ist hierbei als Raster gegeben. Vom Startpunkt aus startet man eine „Flutwelle". Die Wellenfront breitet sich aus, indem man in jeder Iteration jeweils die Vierer-Nachbarschaft der aktuellen Zelle im Raster untersucht und den Nachbarzellen die Zahl der aktuellen Iteration zuordnet. Je weiter eine Zelle vom Startpunkt entfernt ist (im Sinne der Manhatten-Metrik), desto höher ist der ihr zugeordnete Wert. Erreicht man das Ziel, rekonstruiert man den kürzesten Weg, indem man wiederum vom Ziel aus jeweils aus der Vierer-Nachbarschaft einer Zelle die Zelle mit dem geringsten Wert (=Wegkosten) auswählt, bis man den Startpunkt errreicht [Kel05].

3.2.5 Greedy Best-First Search

Während der im vorherigen Abschnitt beschriebene Algorithmus jeweils den Knoten mit der geringsten Distanz zum Startknoten expandiert, orientiert sich die Wahl des nächsten zu untersuchenden Knotens bei der *Greedy Best-First Search*[28] (GBFS) ausschließlich an der geschätzten Distanz eines Knotens zum Ziel $h(n)$. Da dem Algorithmus die Position des Ziels bekannt ist, und die Heuristik somit die Distanz eines Knotens zum Ziel abschätzen kann, nennt man Suchverfahren dieser Art auch *informiert*.

28 In der Literatur ist die Verwendung des Namens „Best-First Search" nicht ganz eindeutig. Manchmal bezeichnet er das hier beschriebene greedy Verfahren (das manchmal auch einfach als „Greedy Search" bezeichnet wird), während dieser Begriff an anderer Stelle als Überbegriff für alle Suchalgorithmen dient, die eine Bewertungsfunktion einsetzen. Ich halte mich hier an die Terminologie wie sie Russel und Norvig in [RN02] verwenden.

Der Ablauf erfolgt weitgehend analog zu dem der im vorherigen Abschnitt beschriebenen UCS, mit dem Unterschied, daß die Prioritätswarteschlange die Knoten nach deren *h*-Wert sortiert und somit immer der Knoten expandiert wird, der die geringste Distanz zum Ziel aufweist.

Zwar suggeriert die Worst-Case-Komplexität von $O(b^m)$ ein ähnliches Laufzeitverhalten wie das der Tiefensuche [RN02]. Da der GBFS-Algorithmus durch die Heuristik jedoch zielgerichtet sucht, muß das Verfahren in der Regel wesentlich weniger Knoten erforschen, als dies bei der blinden Tiefensuche der Fall ist [AP07] (vgl. Abbildung 3.9). In zyklischen bzw. ungerichteten Graphen muß eine Überprüfung stattfinden, ob ein Knoten bereits besucht wurde, um die Vollständigkeit des Algorithmus gewährleisten zu können. Außerdem ist der Algorithmus nicht dazu in der Lage, einen optimalen Pfad zu bestimmen, da er die Gesamtkosten des Pfades nicht berücksichtigt.

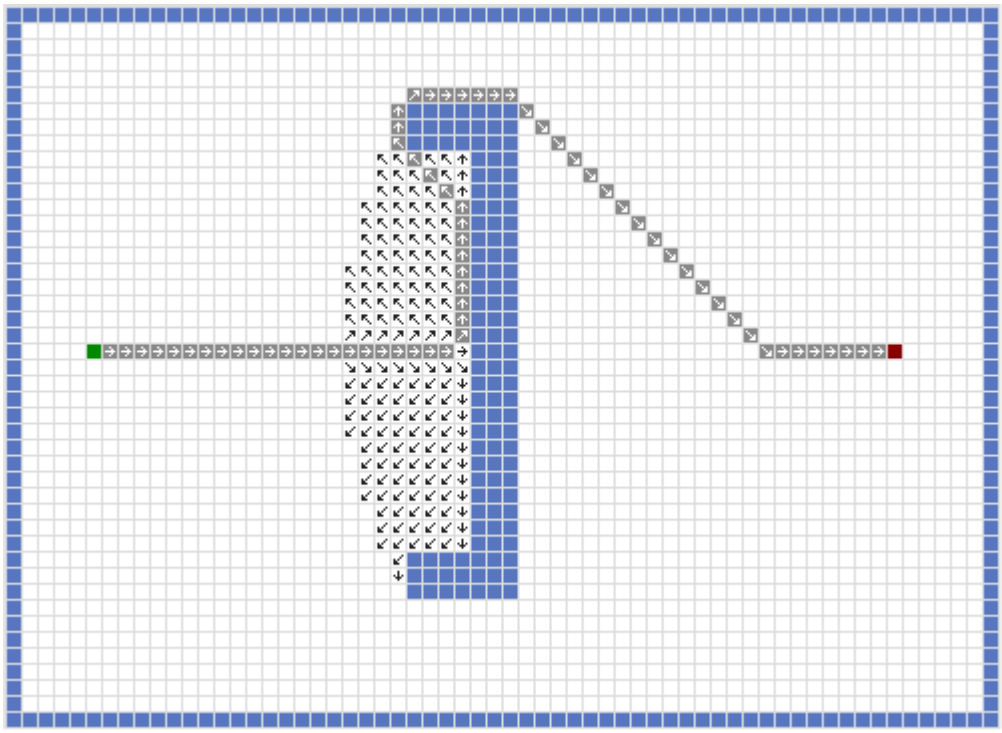


Abbildung 3.9: Greedy Best-First Search
(Quelle: http://www.stefan-baur.de/cs.web.mashup.pathfinding.html)

3.2.6 A*-Algorithmus

Dieses Manko beseitigt schließlich der *A*-Algorithmus*. Je nach Interpretation kann man den A*-Algorithmus entweder als einen Hybriden aus Uniforme-Kos-

ten-Suche und Greedy Best-First Search oder aber letztere als Spezialfälle des A*-Algorithmus auffassen [AP07a]. Der A*-Algorithmus führt eine Bewertungsfunktion $f(n)$ ein, die sowohl die Kosten $g(n)$ vom Start zu einem Knoten n als auch die geschätzten Restkosten $h(n)$ von n zum Zielknoten in die Bewertung der Knoten mit einbezieht:

$$f(n) = g(n) + h(n) \tag{3.1}$$

Damit der A*-Algorithmus einen optimalen Pfad findet, muß es sich bei h um eine *zulässige* d.h. optimistische Heuristik handeln: Sie darf die tatsächlichen Kosten zum Ziel zwar unter-, niemals aber überschätzen. Übliche Heuristiken sind beispielsweise die Manhatten-Distanz oder die Euklidische Metrik [AP07a]. Ist die Heuristik monoton, überschätzt sie zudem die Wegkosten von n zu einem Nachbarknoten n' nicht; $f(n)$ wächst somit entlang eines Pfades monoton.

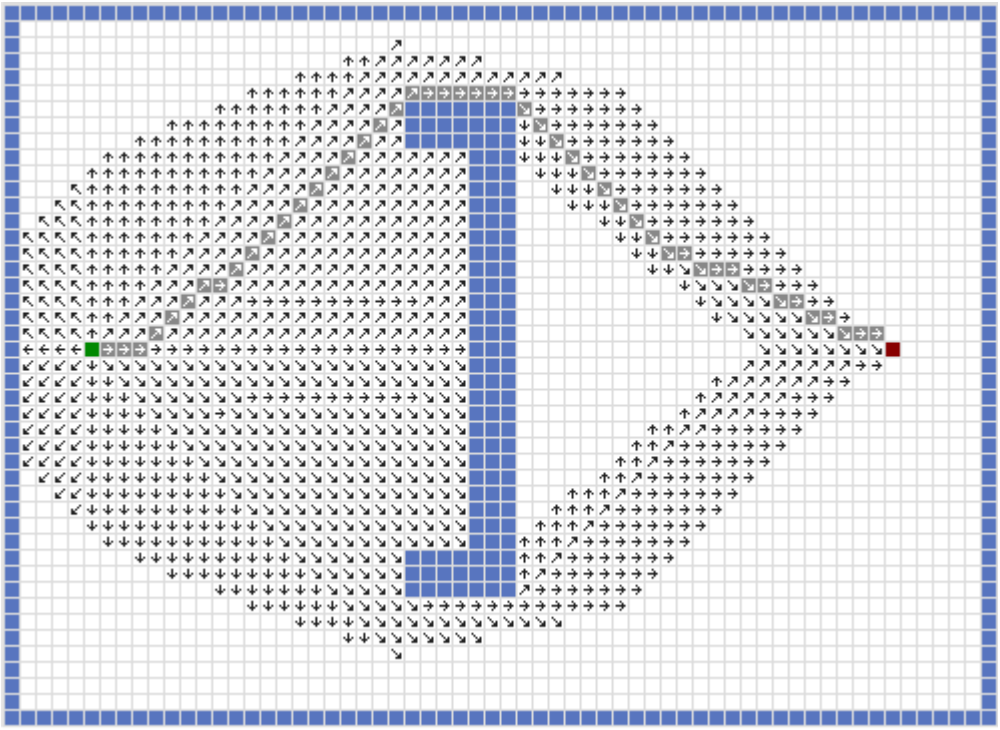


Abbildung 3.10: A*-Suche
(Quelle: http://www.stefan-baur.de/cs.web.mashup.pathfinding.html)

Wie die Uniforme-Kosten-Suche, überprüft der A*-Algorithmus für jeden expandierten Knoten die Distanz zum Start $g(n)$ und paßt diese und den Eltern-Zeiger des Knotens gegebenenfalls an (vgl. 3.2.4). Die Prioritätswarteschlange ist jedoch nach $f(n)$ sortiert, also nach der geschätzten Gesamtdistanz eines Weges, der über einen Knoten n führt. Daher erkundet der Algorithmus immer den Knoten als

nächstes, der für einen Pfad zum Ziel die vermutlich geringsten Gesamtkosten verspricht. Mögliche Pfade, die bereits hohe Gesamtkosten aufweisen, werden somit mit niedriger Priorität behandelt, während die Uniforme-Kosten-Suche in alle Richtungen mit gleicher Priorität sucht.

Wählt man als Heuristik $h(n) \equiv 0$, degeneriert das Verfahren zur Uniforme-Kosten-Suche bzw. Dijkstras Algorithmus. Ignoriert man hingegen $g(n)$, verhält sich die Suche äquivalent zur Greedy Best-First Search. Die Laufzeit wächst zwar exponentiell mit Länge des kürzesten Pfades. Die Wahl einer guten Heuristik kann in der Praxis jedoch zu einem deutlichen Geschwindigkeitsvorteil gegenüber uninformierten Methoden führen. Man kann zeigen, daß A* mit einer monotonen Heuristik optimal effizient ist, d.h. es existiert kein anderer optimaler Suchalgorithmus mit geringerer Laufzeitkomplexität als A* [RN02].

3.3 Implementierung und Integration

Basierend auf den Überlegungen der Abschnitte 3.1 und 3.2, demonstriere ich nun die Umsetzung dieser abstrakten Konzepte in eine praxistaugliche Anwendung zur automatisierten Überbrückungsdetektion.

3.3.1 Grundlegende Vorgehensweise

Die kortikalen Enden sind zusammenhängend, wenn sie mindestens ein Weg aus verknöcherten Elementen verbindet (3.1.2). Um einen der zuvor vorgestellten Suchalgorithmen auf dieses Problem anwenden zu können, benötigt man einen definierten Start- und Zielknoten als Eingabeparameter für die Suche. Dazu spezifiziert der Benutzer die Koordinaten des Schwerpunktes der Elemente, die als Start- bzw. Zielknoten dienen sollen. Da sich die exakten Koordinaten in Abhängigkeit von der Modellauflösung ändern können, wählt der „Bridge Detector“ (so der Name der Implementierung) jeweils die Elemente mit der geringsten Distanz zu den vorgegebenen Koordinaten.

Als Grundlage für die Konstruktion des Graphen dienen zum einen die Schwerpunkte der finiten Elemente, zum anderen die auch vom Fuzzy-Controller der Heilungssimulation genutzten Nachbarschaftstabelle und eine Auflistung der Materialeigenschaften der Elemente; darunter befinden sich auch die Knochenkonzentrationen (siehe 2.4.3). Zwischen den Simulationsschritten ändern sich zwar die Knochenkonzentrationen, die Relationen zwischen den Knoten und die Gewichte der Kanten bleiben jedoch bestehen. Der Benutzer erzeugt den Graphen daher nur einmal nach der Initialisierung des FE-Modells; der Bridge De-

tector paßt in folgenden Aufrufen nur die jeweils aktuellen Knochenkonzentrationen an.

Falls der Suchalgorithmus nach einem Heilungsschritt eine Überbrückung feststellt, muß der Bridge Detector die Heilungssimulation darüber informieren. APDL-Skripte unterstützen leider keinerlei leistungsfähige IPC-Mechanismen[29], so daß man auch hierfür wiederum auf die Kommunikation via Text-Datei ausweichen muß. Optional kann der Bridge Detector den Zeitpunkt der frühesten Überbrückung aus den Knochenkonzentrationen des vorangegangenen und des aktuellen Iterationsschrittes interpolieren.

3.3.2 Architektur

Der Aufbau des Bridge Detectors ist, im Vergleich zum Röntgensimulator, eher einfach gehalten.

Die wesentliche Funktionalität beherbergen die Klassen `Node` und `Graph`, die den Graphen und seine Knoten modellieren. Die Kanten sind in Form einer Liste von Nachbarknoten realisiert, die jeder Knoten verwaltet. Die Klasse Graph bietet auch die Suchfunktionalität an. Daneben existieren noch einige Utility-Klassen, die z. B. der Berechnung von Distanzen (`Vector`) oder dem Parsen der Eingabedaten dienen (`AnsysOutputParser`). Die Implementierung erlaubt das Suchen in beliebig-dimensionalen FE-Modellen.

Die zentrale Funktionalität ist als .NET-Klassenbibliothek (DLL) implementiert, die vom Benutzerinterface verwendet wird (siehe Abbildung 3.11). Das primäre Interface ist kommandozeilenbasiert (CLI, siehe Folgeabschnitt). Es existiert zwar auch ein graphisches Interface (VisualBridgeDetector); dieses dient jedoch primär Debugging-Zwecken und zur visuellen Kontrolle des Verhaltens der Suchalgorithmen.

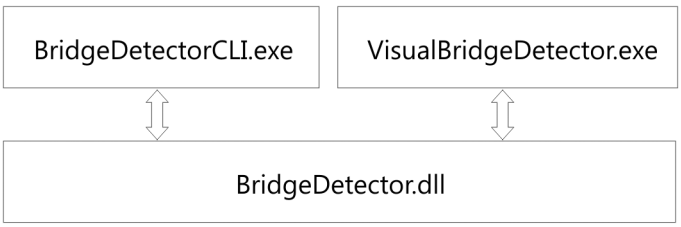


Abbildung 3.11: Architektur des BridgeDetectors

3.3.3 Bedienkonzept

Die Benutzung des Bridge Detectors erfolgt fast ausschließlich nicht-interaktiv. Vielmehr muß er sich möglichst leicht in das bestehende APDL-Script der Hei-

29 IPC: *Inter-Process Communication;* Verfahren, die es ermöglichen, über Adressraumgrenzen hinweg Daten zwischen Prozessen auszutauschen

lungssimulation integrieren lassen. Die primäre Benutzerschnittstelle ist deswegen kommandozeilenbasiert.

Der Bridge Detector kennt drei „Betriebsmodi“. Welcher davon aktiv ist, hängt von der Kombination der Kommandozeilenparameter ab.

Erzeugen eines neuen Graphen

Die Parameterfolge

```
-c <Koordinatendatei> -a <Nachbarschaftstabelle> ↵
-s <Koordinaten des Startknotens> -g <Koordinaten des Ziels> ↵
-o <zu erzeugende Graph-Datei>
```

erzeugt einen Graphen, indem es die Dateien `<Koordinatendatei>` und `<Nachbarschaftstabelle>` parst. Die mittels `-s` und `-g` angegebenen Koordinaten legen die gewünschten Start- und Zielknoten für eine Suche im Graphen fest. Der „`-o`“-Parameter schließlich gibt an, unter welchem Namen der erzeugte Graph gespeichert werden soll.

Suche in einem bestehenden Graphen

Den so erstellten Graph verwendet man in Kombination mit den jeweils aktuellen Knochenkonzentrationen, um dann die Überbrückungsdetektion durchführen zu können:

```
-u <Graph-Datei> ↵
-fnow <aktuelle Konzentrationen> ↵
-fbefore <Konzentrationen der vorangegangenen Iteration> ↵
-iht <Datei für interpolierten Heilungszeit-Offset> ↵
-o <Pfad-Datei>
```

Findet der Bridge Detector einen Pfad zwischen Start- und Zielknoten, gibt er ihn auf `stdout` und in die Datei `<Pfad-Datei>` aus. Die Verwendung der Konzentrationen aus der vorangegangenen Iteration erlaubt es, den Zeitpunkt der Überbrückung durch Interpolation genauer zu bestimmen (siehe 3.3.6). Das Ergebnis der Interpolation wird als Offset in der angegebenen Datei gespeichert. Um das APDL-Script darüber zu informieren, daß eine Überbrückung existiert, erstellt der Bridge Detector eine Datei `path-exists.txt`, deren Existenz das APDL-Script prüft und die Simulation dann gegebenenfalls abbricht.

Erzeugung und Suche „On-the-Fly“

Es ist nicht zwingend notwendig, den Graphen vorab zu erzeugen, und ihn später wieder zu laden. Übergibt man dem Bridge Detector alle notwendigen Parameter, führt er die Suche auf einem zur Laufzeit erzeugten Graphen durch.

```
-c <Koordinatendatei> -a <Nachbarschaftstabelle> ↵
-s <Koordinaten des Startknotens> -g <Koordinaten des Ziels> ↵
-fnow <aktuelle Konzentrationen> ↵
-fbefore <Konzentrationen der vorangegangenen Iteration> ↵
-iht <Datei für interpolierten Heilungszeit-Offset> ↵
-o <Pfad-Datei>
```

Wie man sieht, handelt es sich um eine Kombination der beiden zuvor beschriebenen Betriebsmodi.

3.3.4 Datengenerierung und -import

Neben den Koordinaten der Start- und Zielelemente, benötigt der Bridge Detector die Schwerpunkte der Elemente, ihre Nachbarschaftsbeziehungen und die jeweilige Knochenkonzentration.

Element-Schwerpunkte

Der Export der Koordinaten der Schwerpunkte der Elemente verläuft analog zur Ausgabe der Vertex-Koordinaten für den Röntgensimulator (siehe 2.4.3). Die Koordinaten werden auch in diesem Fall in eine Textdatei ausgegeben.

Nachbarschaften

Die Beziehungen der Elemente zueinander sind unverzichtbar, um ein Graphenmodell des Frakturmodells generieren zu können. Auch der Fuzzy-Controller der Heilungssimulation und der Röntgensimulator benötigen diese Daten. Im Gegensatz zu den Anforderungen des Röntgensimulators, kann der Bridge Detector die Eingabedaten des Fuzzy-Controllers ohne Änderung mitverwenden, da für die Konstruktion des Graphen nicht von Belang ist, welche Seitenfläche des Nachbarn an welche eigene Seitenfläche grenzt. Die Kanten zwischen den Knoten bilden nur die Nachbarschaftsbeziehung selbst ab.

Die Nachbarschaftsreferenzdatei, die auch der Fuzzy-Controller verwendet, listet die Nachbarschaften jedes Elements in folgendem Format auf:

```
<Anzahl Nachbarelemente> <Element-Nummer> … <Element-Nummer>
```

Knochenkonzentrationen

Der Bridge Detector verwendet hierfür die gleichen Eingabedaten, wie der Röntgensimulator und der Fuzzy-Controller. Daher möchte ich an dieser Stelle auf meine Ausführungen in 2.4.3 verweisen.

Datenimport und Generierung des Graphen

Die Methoden der statischen Klasse `AnsysOutputParser` sind für das Einlesen der Eingabedaten zuständig. Dafür parst sie zunächst die Datei mit den Schwerpunkten der Elemente und erzeugt die Knoten des Graphen. Abhängig von den Knochenkonzentrationen, setzt sie das `IsPassable`-Flag der Knoten, das angibt, ob ein Knoten als passierbar gilt und somit auch Teil eines Pfades sein kann. Im letzten Schritt erzeugt eine Methode der Klasse die Verknüpfungen zwischen den Knoten, basierend auf den Daten aus der Nachbarschaftsreferenzdatei.

3.3.5 Pfadsuche

Der Wegsuche-Algorithmus, den der Bridge Detector zur Überbrückungsdetektion einsetzt, basiert auf dem in Abschnitt 3.2.6 vorgestellten A*-Algorithmus. Dieser besitzt die praktische Eigenschaft, daß man dessen Verhalten allein durch eine andere Gewichtung der Parameter g und h der Bewertungsfunktion „tunen“ kann. Im Extremfall erzielt man so zu Dijkstras Algorithmus (Gewichtung von $h(n)$ mit 0) oder zur Greedy Best-First Search (Gewichtung von $g(n)$ mit 0) äquivalente Ergebnisse, ohne den Algorithmus selbst modifizieren zu müssen. Die zu verwendende Heuristik übergibt man dem Graphen-Objekt in Form eines Delegaten[30]; die Heuristiken `EuclideanDistance` und `ManhattenDistance` sind bereits vordefiniert.

Die naive Implementierung des A*-Algorithmus arbeitet mit zwei Listen: Die Open-Liste beinhaltet Knoten, die entdeckt, aber noch nicht untersucht wurden und ist nach dem f-Wert der Knoten sortiert. Die Closed-Listet verzeichnet die Knoten, die der Algorithmus bereits besucht hat. Sie dient dazu, Zyklen zu erkennen und die Suche gegebenenfalls abbrechen zu können.

In der Initialisierungsphase setzt man den g-Wert des Startknotens auf 0, berechnet seine geschätzte Distanz zum Ziel (h-Wert) und fügt ihn in die Open-List ein. Die folgenden Schritte wiederholen sich so lange, bis die Open-Liste leer ist, ein Pfad gefunden wird, oder ein bereits besuchter Knoten erneut geöffnet wird (vgl. Abbildung 3.12).

Der Algorithmus betrachtet nun den Knoten der Open-Liste, der den geringsten f-Wert besitzt. Diesen entfernt er zunächst von der Open-Liste. Handelt es sich bei dem Knoten um den gesuchten Zielknoten, kann man den Pfad durch Zurückverfolgen der `Parent`-Referenzen der Knoten rekonstruieren. Falls der Knoten bereits zuvor besucht wurde und somit auch in der Closed-List enthalten ist, existiert kein Pfad und die Suche terminiert.

30 Ein .NET-*Delegat* (engl. *delegate*) ist, vereinfacht gesagt, ein typisierter Funktionszeiger.

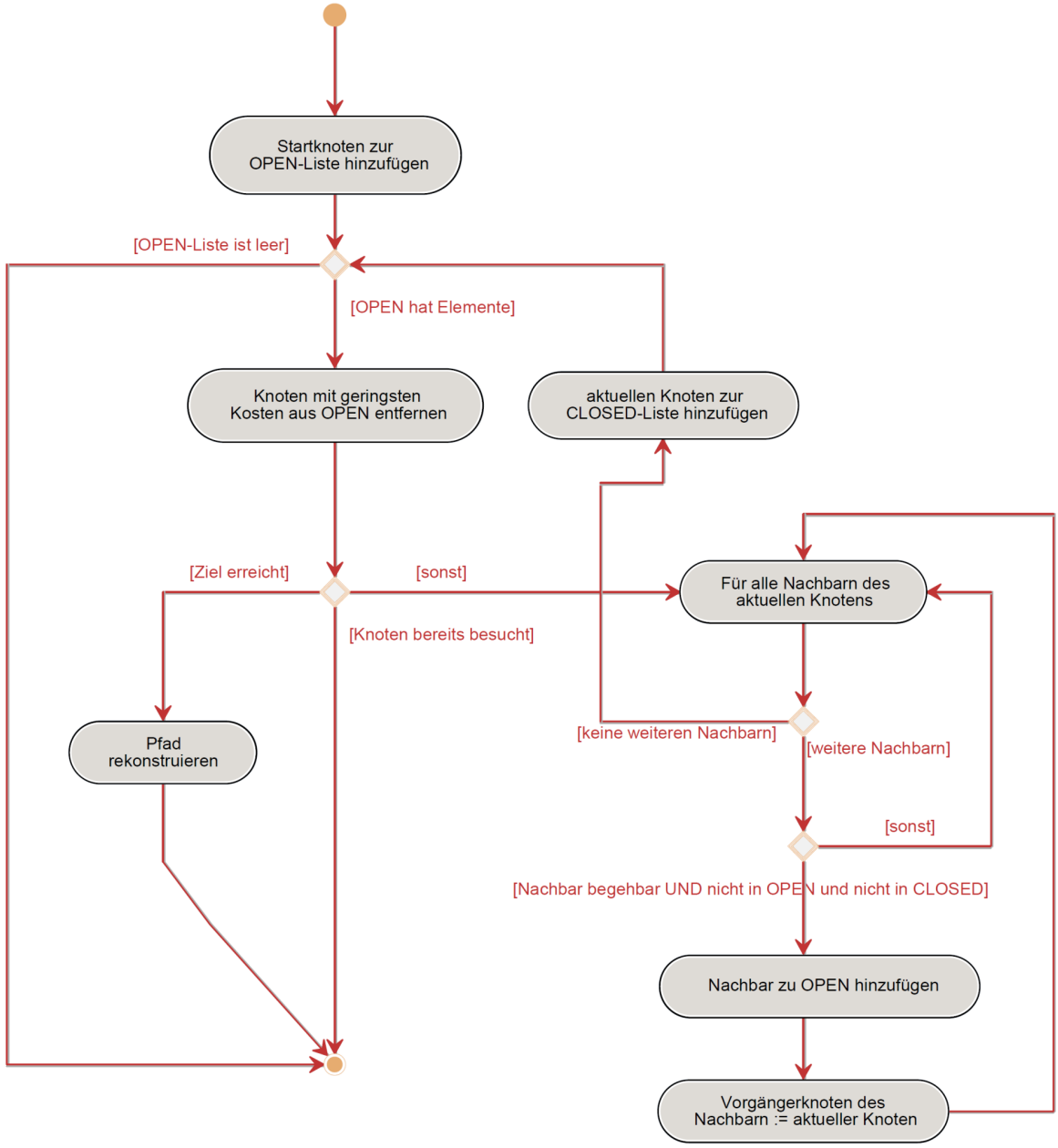


Abbildung 3.12: Schematischer Ablauf der A*-Pfadsuche

Andernfalls folgt die Untersuchung der als begehbar markierten Nachbarknoten des aktuell geöffneten Knotens. Verringert der Weg über den aktuell geöffneten Knoten die Distanz eines der Nachbarknoten zum Startknoten, setzt man die `Parent`-Referenz des Nachbarknoten auf den aktuellen Knoten und seinen g-Wert auf die neu berechnete Distanz $g(n_{\text{Neighbour}}) = g(n_{\text{Current}}) + \text{dist}(n_{\text{Current}}, n_{\text{Neighbour}})$. Da sich seine geschätzte Gesamtdistanz zum Ziel durch den neuen g-Wert geändert hat, muß der Algorithmus den Knoten eventuell nochmals untersuchen. So-

fern sich der Nachbarknoten auf der Closed-Liste befindet, entfernt man ihn von dieser. Außerdem muß man ihn zur Open-Liste hinzufügen, falls der Nachbarknoten dort noch nicht verzeichnet ist. Hat man alle Nachbarn untersucht, fügt man den aktuellen Knoten in die Closed-Liste ein und fährt mit dem nächsten Knoten der Open-Liste fort.

Findet die Suche einen Weg zwischen Start- und Zielknoten, kann man den Pfad auf einfache Weise mittels einer rekursiven Funktion aus den `Parent`-Referenzen der Knoten rekonstruieren. Dieser wird zwar für die eigentliche Funktionalität („Wann tritt eine Überbrückung auf?“) nicht benötigt; es erleichtert aber das Debugging und ermöglicht die Überprüfung des durch den Algorithmus entdeckten Pfades.

3.3.6 Interpolation des Heilungszeitpunktes

Als Ergebnis der bisher beschriebenen Funktionalität erhält man als Maß für die Heilungsdauer die Anzahl der Iterationen *i*, bis eine erste Überbrückung auftritt. Verfügt man über die Knochenkonzentrationen der vorangegangenen Iteration *i*-1, ist es möglich, Zwischenzustände des Graphen zu interpolieren und somit eine genauere Näherung für die Heilungsdauer zu ermitteln. Das angewendete Verfahren arbeitet in mehreren Schritten:

Ermittlung kritischer Knoten

Für die Interpolation von Zwischenzuständen sind nur die Knoten von Bedeutung, deren Knochenkonzentration mit der aktuellen Iteration einen kritischen Wert überschritten hat. Alle anderen Knoten symbolisieren entweder bereits in Iteration *i*-1 verknöchertes Material oder können zur Überbrückung nicht beitragen, da ihre Knochenkonzentration auch in der aktuellen Iteration noch zu gering ist.

Es ist zu beachten, daß es nicht genügt, nur die kritischen Knoten des entdeckten Pfades zu extrahieren. Zum Zeitpunkt *i* existiert typischerweise mehr als genau ein Pfad zwischen Start- und Zielknoten. Es ist daher nicht unwahrscheinlich, das einer dieser Pfade bereits früher ausgebildet war, als der vom Suchalgorithmus entdeckte Pfad.

Berechnung möglicher Überbrückungszeitpunkte

Für alle diese „kritischen Knoten“ berechnet man mittels linearer Interpolation den Zeitpunkt, zu dem die Knochenkonzentration des jeweils repräsentierten Elements einen willkürlich gewählten Schwellwert überschritten hat (hier:

88,9 %). So erhält man eine Liste möglicher Zeitpunkte, zu denen es zu einer Überbrückung gekommen sein kann.

Suche nach dem Zeitpunkt frühester Überbrückung

Interpoliert man den Zustand des Graphen für jeden der berechneten möglichen Heilungszeitpunkte und führt auf den so erzeugten Graphen jeweils eine Pfadsuche durch, kann man den frühesten Zeitpunkt zwischen den Iterationen i-1 und i ermitteln, zu dem eine Überbrückung bestand. Um diesen Suchvorgang zu beschleunigen, setzt der Bridge Detector eine angepaßte binäre Suche ein. Durch diese Maßnahme sinkt die mittlere Laufzeitkomplexität der Heilungszeit-Interpolation von $O(n)$ auf $O(\log(n))$ (n: Anzahl der berechneten Zeitpunkte).

Das Ergebnis der Interpolation ist ein Offset, der angibt, um welchen Bruchteil einer Iteration die Überbrückung bereits vor der aktuellen Iteration existiert hat. Befindet man sich also momentan in Iteration i und gibt die Interpolation einen Offset von o zurück, trat die erste Überbrückung zum Zeitpunkt $n - o$ auf.

3.3.7 Integration in die Heilungssimulation

Neben der Erzeugung und der Ausgabe von Daten (siehe 3.3.4), muß man die Heilungssimulation noch um drei weitere Punkte erweitern, um eine Integration der Überbrückungsdetektion zu erreichen.

Nach dem Datenexport erzeugt man wie in 3.3.3 beschrieben ein Graphenmodell. In den Folgeiterationen wird dieser Graph zur Überbrückungsdetektion wiederverwendet, da sich an der Geometrie und dem Zusammenhang der Elemente während der Simulation nichts ändert. Lediglich die Knochenkonzentrationen paßt man dem aktuellen Zustand an.

Entdeckt der Bridge Detector eine Überbrückung, muß die Heilungssimulation davon in Kenntnis gesetzt werden, um den Simulationsvorgang abbrechen zu können. Dazu erzeugt der Bridge Detector eine Datei `path-exists.txt` mit dem schlichten Inhalt „`1`“. Die Heilungssimulation prüft, ob eine solche Datei vorhanden ist und beendet die Simulation gegebenenfalls (siehe Abbildung 3.13).

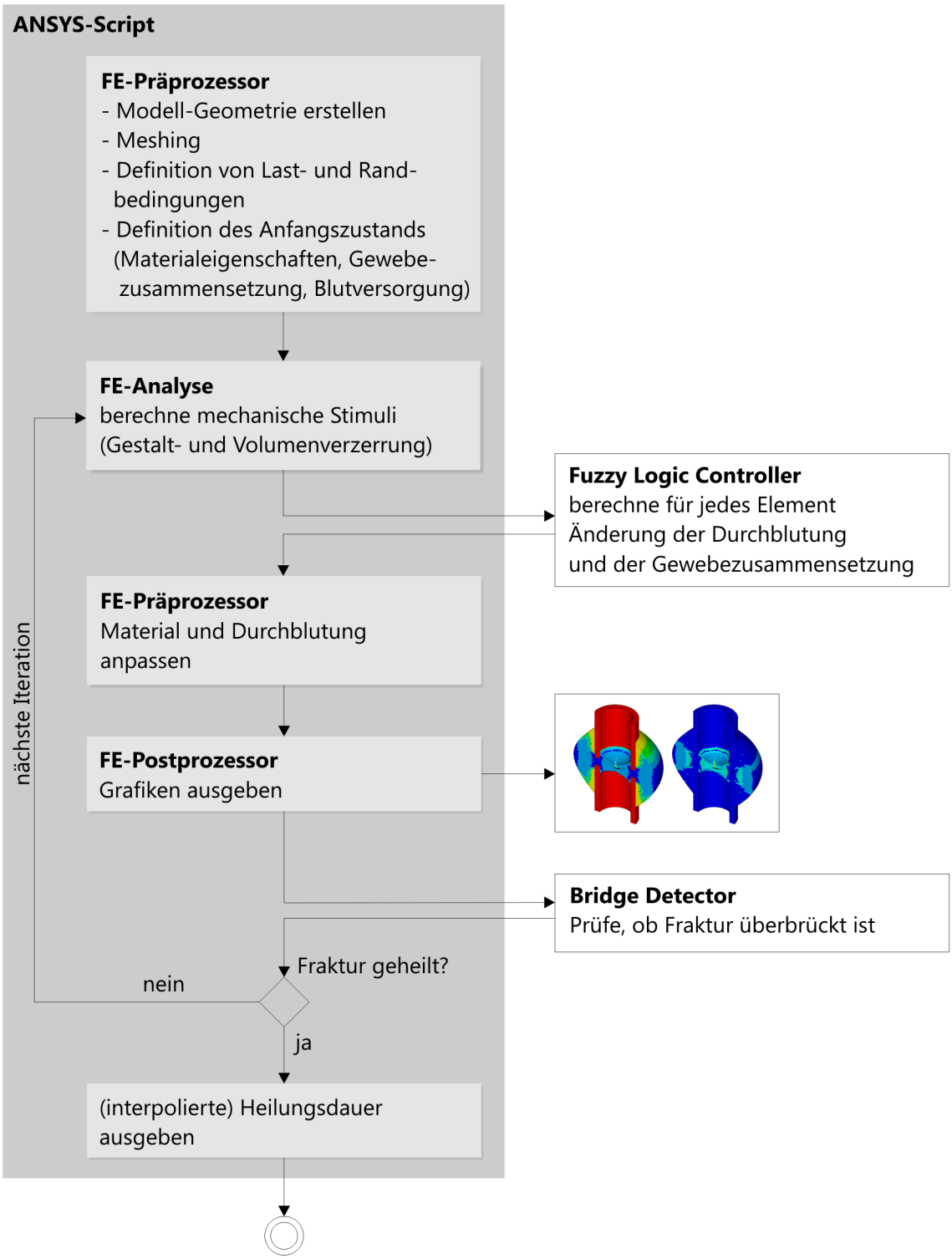


Abbildung 3.13: Schema der um die Überbrückungsdetektion erweiterten Heilungssimulation

3.4 Optimierungen

Die Überbrückungsdetektion prüft nach jedem Heilungsschritt den Zustand der simulierten Fraktur. Daher ist es wichtig die Laufzeit, die der Bridge Detector pro Iteration beansprucht, möglichst gering zu halten. Im folgenden erörtere ich die zwei wichtigsten, algorithmischen Optimierungen, die der Überbrückungsdetektion zu mehr Geschwindigkeit verhelfen.

3.4.1 Optimierte Datenstrukturen

Der Suchalgorithmus verwendet in seiner naiven Implementierung zwei Listen (Open und Closed, vgl. 3.3.5) vom Typ `List<Node>`. Die Open-Liste muß die referenzierten Knoten nach dem *f*-Wert sortiert zurückliefern.

Analyse des bestehenden Algorithmus

Der A*-Algorithmus fordert von der Open-Liste den Knoten mit der geringsten geschätzten Gesamtdistanz an. Dazu sortiert er die Liste vor der Abfrage zunächst durch einen Aufruf der `List<T>.Sort()`-Methode. Diese Methode implementiert den Quicksort-Algorithmus, der eine mittlere Laufzeitkomplexität von $O(n \log(n))$ aufweist [MSDN07a]. Die Sortierung erfolgt erst zu diesem Zeitpunkt, da der Algorithmus wesentlich häufiger Knoten zur Liste hinzugefügt als entnimmt. Dann greift er mittels des Index-Operators auf den ersten Knoten in der nun sortierten Liste zu ($O(1)$). Die Entfernung dieses Knotens aus der Liste besitzt eine Komplexität von $O(n)$, da `List<T>` intern als Array implementiert ist und das Entfernen des ersten Knotens in einem Array ein Umkopieren aller n Knoten nach sich zieht [MSDN07a].

Um Zyklen erkennen zu können, prüft man, ob der aktuell geöffnete Knoten bereits in der Closed-Liste verzeichnet ist. Die Methode `List<T>.Contains(T)` führt eine lineare Suche durch und erzielt somit eine Komplexität von $O(n)$.

Bei der Untersuchung der Nachbarknoten, muß der Suchalgorithmus prüfen, ob sich der aktuelle Nachbarknoten in der Open- oder Closed-Liste befindet, falls der neu berechnete Weg kürzer als der alte ist (siehe 3.3.5). Sowohl für die Closed- als auch für die Open-Liste ruft man wiederum zunächst die `List<T>.Contains(T)`-Methode auf ($O(n)$, siehe oben). Befindet sich der Nachbarknoten auf der Closed-Liste, entfernt man ihn mittels `Remove(T)` ($O(n)$). Das Hinzufügen des Knotens zur Open-Liste mittels `List<T>.Add(T)` erfordert $O(1)$, sofern das interne Array nicht vergrößert werden muß (sonst: $O(n)$). Schließlich fügt man den gerade untersuchten Knoten der Closed-Liste hinzu ($O(1)$).

Einige der linearen Suchoperationen kann man einsparen, indem man an Stelle der `Contains(T)`-Methode direkt den Index *i* eines Elements der Liste mittels `IndexOf(T)` ermittelt; der Zugriff auf das Element bzw. das Entfernen kann über diesen Index erfolgen. Da es aber auch bei `RemoveAt(int)` vonnöten ist, zumindest Teile des Arrays umzukopieren, weist auch diese Methode noch immer eine Komplexität von $O(n-i)$ auf [MSDN07a]; der Vorteil dieser Vorgehensweise hält sich daher meist in engen Grenzen.

Entwicklung einer Hybrid-Datenstruktur

Um insbesondere die zahlreichen linearen Suchoperationen zu vermeiden, die insbesondere bei einer großen Zahl von Knoten schnell zum Flaschenhals werden, bietet sich eine *Hashtabelle* (*Hash Table*, *Hash Map*, *Dictionary*) als Speicherstruktur an. Sie ordnet jedem Wert (*Value*) einen eindeutigen Schlüssel (*Key*) zu, über dessen Hash-Wert sie die Speicheradresse eines gesuchten Wertes direkt ermitteln kann: Der Aufwand ein Element zu finden sinkt dadurch auf $O(1)$, sofern die Hashtabelle ausreichend groß dimensioniert wurde und es nicht zu Kollisionen kommt.

Dem gegenüber steht die Anforderung von informierten Suchalgorithmen, als nächstes immer den „besten“ (z.B. geringste geschätzte Gesamtkosten) noch nicht untersuchten Knoten zu untersuchen zu müssen. Um den besten Knoten schnell ermitteln zu können, benötigt man folglich eine Datenstruktur, die die Knoten in einer geeigneten Ordnung speichert. In einer Hashtable jedoch gibt es keine solche Ordnung, im Gegenteil: die Hash-Funktion sorgt für eine möglichst gleichverteilte Streuung der zu speichernden Werte, um Kollisionen zu vermeiden.

Für den Bridge Detector habe ich daher eine speziell angepaßte Hybrid-Datenstruktur entwickelt, die gezielt die Operationen beschleunigt, die vom Suchalgorithmus am häufigsten benötigt werden (insbesondere das Auffinden von Elementen), andererseits aber auch für die benötigte Ordnung auf den gespeicherten Daten sorgt. Der auf den Namen `SortableDictionary<TKey, TValue>` getaufte generische Container verwendet intern zwei getrennte Datenstrukturen zum Abspeichern der Knoten: eine sortierte Liste von Knoten und eine Hashtabelle vom Typ `Dictionary<int, Node>`. Als Schlüssel für die Hashtabelle dient eine eindeutige Elementnummer, die jedes finite Element besitzt. Die Sortierreihenfolge bestimmt die dem `SortableDictionary` übergeben Instanz einer Implementierung von `IComparer<TValue>`.

Während die (seltener auftretenden) `Remove`-Operationen ihre Komplexität von $O(n)$ beibehalten, kann man zumindest die Kosten aller `Contains`-Aufrufe durch den Einsatz der Hashtabelle auf $O(1)$ reduzieren. Gegenüber der naiven Implementierung erzielt man so bereits bei einem relativ kleinen Graphen mit 30 000 Knoten einen Speed-Up von mehr als Faktor 60.

3.4.2 Anpassung des Suchalgorithmus

Betrachtet man die Ausgabe des Debugging-Tools „Visual BridgeDetector“ in Abbildung 3.14, die einen Suchlauf in einem Graphen mit 31826 Elementen (generiert aus einem 2D-FE-Modell) visualisiert, fällt sofort auf, daß der Suchalgorithmus sehr viel mehr Knoten untersucht als notwendig wäre, um einen Zusammenhang zwischen Start- und Zielknoten feststellen zu können. Der Grund für dieses Verhalten ist in der Optimalität des A*-Algorithmus zu suchen: Um mit Sicherheit einen kürzesten Pfad liefern zu können, muß der Algorithmus alle Wege untersuchen, deren geschätzte Wegkosten kleiner sind wie die des aktuell besten bekannten Weges. Dieses Verhalten macht sich besonders dann negativ bemerkbar, wenn weite Teile des Graphen — wie auch im Beispiel — nicht begehbar sind. Befindet sich hingegen kein „Hindernis“ zwischen Start und Ziel, kann auch der A*-Algorithmus direkt in Richtung Ziel expandieren, ohne eine große Zahl weiterer Kandidaten berücksichtigen zu müssen.

Glücklicherweise ist es im Rahmen der Überbrückungsdetektion lediglich notwendig, die Existenz *irgendeines* Weges zwischen den beiden als Start- bzw. Zielknoten festgelegten Elementen zu überprüfen. Ob es sich dabei um einen kürzesten Pfad handelt oder nicht, ist für diese Anwendung nicht von Belang. Für diese Aufgabe eignet sich die in Abschnitt 3.2.5 vorgestellte Greedy Best-First Search (siehe Listing 3.1), zumal es sich dabei im wesentlichen um einen Spezialfall von A* handelt, der die Kosten vom Start zum aktuellen Knoten ($g(n)$) nicht berücksichtigt — entsprechend leicht fällt die Integration in den Bridge Detector (vgl. Abbildung 3.14).

Durch den kompakteren Code und die deutlich geringere Zahl expandierter Knoten, terminiert der neue Suchalgorithmus oftmals deutlich schneller als A*: Für das dargestellte 2D-Modell mit 31826 Elementen ergibt sich durch diese Modifikation eine Beschleunigung um etwa den Faktor drei gegenüber A*.

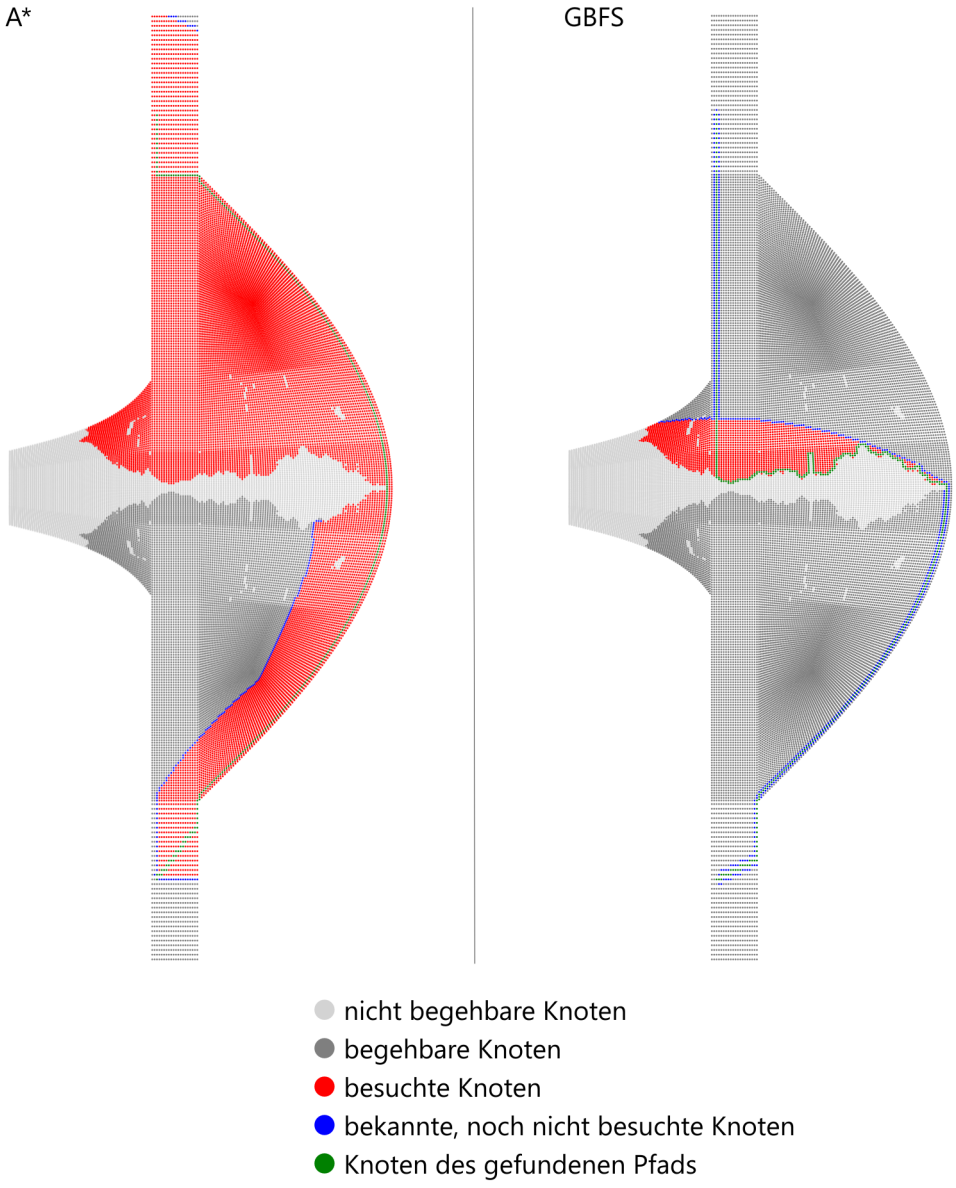


Abbildung 3.14: A*-Suche im Graphen eines 2D-FE-Modells im Vergleich zur Greedy Best-First Search (GBFS)

```
open : List
closed : List
current node : Node

add start node to open

WHILE count(open) > 0
{
    current node ← remove top node from open

    IF current node == goal node
        RETURN back_track_path(current node, start node)

    IF closed CONTAINS current node
        RETURN NULL

    FOR EACH neighbour OF current node
    {
        IF is_passable(neighbour) AND NOT
        (open CONTAINS neighbour OR closed CONTAINS neighbour)
        {
            parent(neighbour) ← current node
            add neighbour to open
        }
    }

    add current node to closed
}
```

Listing 3.1: Greedy Best-First Search (Pseudocode)

3.5 Ergebnisse

Zur Überprüfung der vom BridgeDetector zurückgelieferten Heilungszeiten betrachten wir die mit 3D-FE-Modellen verschiedener Auflösung simulierten Heilungsverläufe. Vergleiche mit ANSYS-Plots (Verteilung der Knochenkonzentration) und gerenderten Röntgenbildern stellen die Plausibilität der vom BridgeDetector berechneten Heilungsdauer sicher.

In Abbildung 3.15 sieht man links jeweils Grafiken der Knochenkonzentrationen pro Element und in der Mitte das zugehörige Röntgenbild. Die Daten basie-

ren auf einem Modell aus Elementen mit 0,8 mm Kantenlänge, die Elementanzahl beträgt 57 344. Nach Simulationsschritt 30 (erste Reihe) ist auf der ANSYS-Grafik noch keine Überbrückung der Kallushälften erkennbar, wendet man zur Beurteilung die bisher übliche manuelle bzw. visuelle Methode an (vgl. 3.1). Auch der BridgeDetector kann noch keine Überbrückung feststellen.

Nach der Folgeiteration (31) kann man schließlich eine knöcherne Überbrückung ausmachen, die auch der BridgeDetector findet. Der berechnete Pfad vom Start- zum Zielknoten/-element ist im Röntgenbild rechts unten als Folge rein-weißer Elemente gekennzeichnet. Als Resultat wird folglich der Wert „31“ ausgegeben. Berücksichtigt man das Ergebnis der Interpolation der Heilungszeit (siehe 3.3.6), lautet der Zeitpunkt der Überbrückung (gerundet):

31,0 – 0,843 = 30,16

D.h.: Die Überbrückung der Kallushälften fand nach rund 30,16 Simulationsschritten statt.

Abbildung 3.16 zeigt ein deutlich höher aufgelöstes Modell: Die mittlere Kantenlänge beträgt 0,5 mm, die Anzahl der Elemente liegt bei 219 024. In der oberen Hälfte sieht man die Plots der Knochenkonzentrationen nach Schritt 41 (links) und 42 (rechts). Wie man unschwer erkennt, findet die Überbrückung in Iteration 42 statt. Auch der BridgeDetector erkennt am Ende dieser Iteration die Fraktur als geheilt. Der gefundene Pfad ist diesmal etwas komplizierter, wie man an den drei „Röntgenaufnahmen“ aus drei verschiedenen Perspektiven erkennen kann.

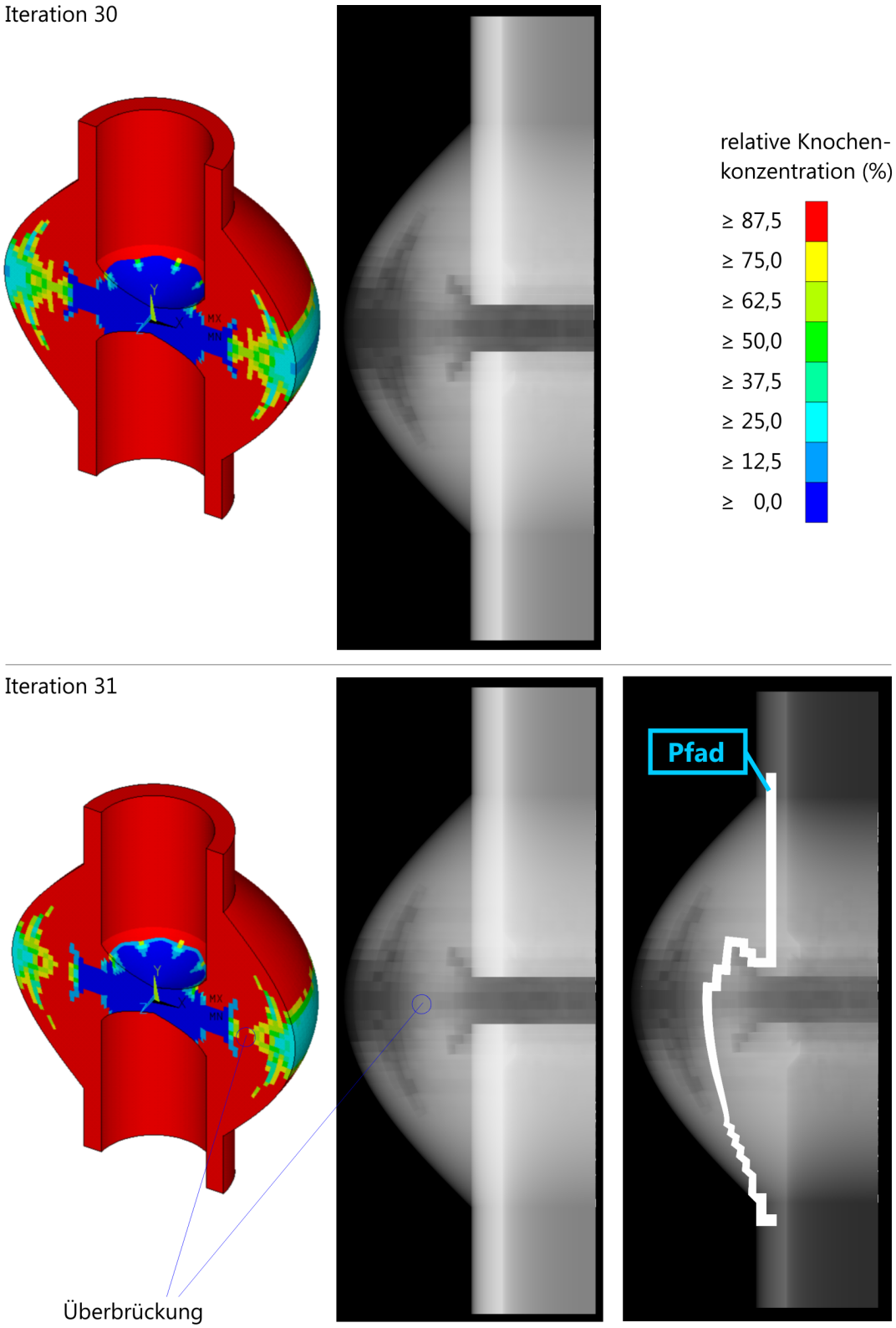


Abbildung 3.15: Überbrückungsdetektion in niedrig aufgelöstem 3D-Modell
Links sind Grafiken der Verteilung der Knochenkonzentration im Modell zu sehen, in der Mitte das korrespondierende Röntgenbild (Ansicht von lateral). Nach Simulationsschritt 31 erkennt der BridgeDetector eine Überbrückung (rein weiße Elemente im Röntgenbild rechts unten).

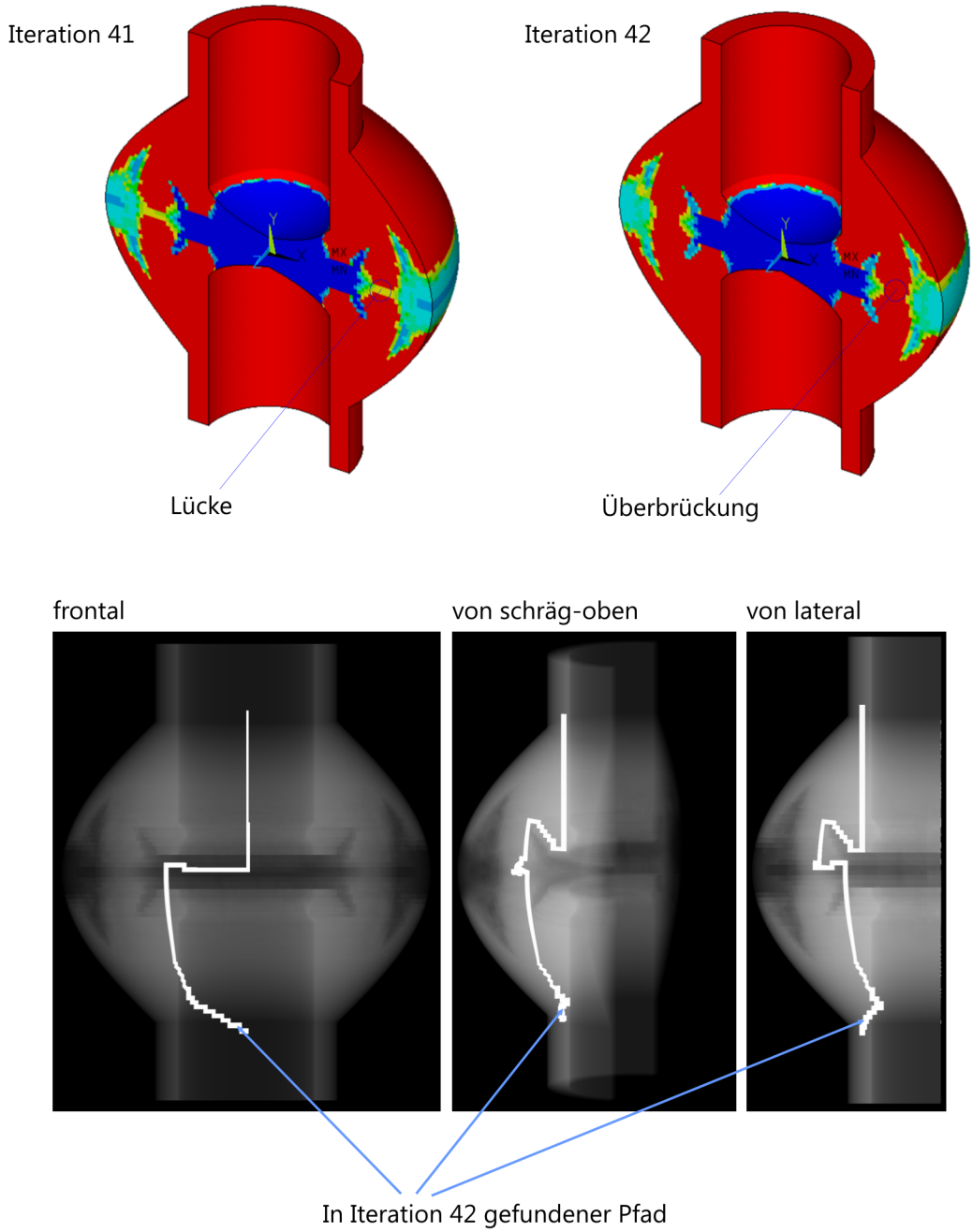


Abbildung 3.16: Überbrückungsdetektion in hoch aufgelöstem Frakturmodell

4 Fazit

In Kapitel zwei habe ich eine Möglichkeit zur Volumenvisualisierung von Finite-Elemente-Modellen aufgezeigt. Das vorgestellte Verfahren erlaubt die Generierung röntgen- bzw. CT-ähnlicher Bilder aus der Modellgeometrie und den zugeordneten Materialeigenschaften. Insgesamt verbessert sich dadurch die Vergleichbarkeit der Ergebnisse der Simulation mit den tierexperimentell gewonnenen Erkentnissen. Das im Rahmen dieser Arbeit entwickelte Visualisierungstool „X-Ray-Simulator“ kann somit wie geplant in Zukunft der Weiterentwicklung, Verbesserung und Überprüfung des bestehenden Frakturheilungsmodells dienen.

Die automatisierte Überbrückungsdetektion in Form des „BridgeDetectors“ entdeckt Überbrückungen des Frakturkallus zuverlässig. Im Rahmen eines Projekts[31] zur Optimierung der Fixationsparameter befindet sich das Tool bereits im produktiven Einsatz und hat sich dort als zuverlässiges Werkzeug bewährt.

31 „Simulationsbasierte Optimierung von Fraktur-Fixateuren“ (U. Simon, D. Nolte, F. Niemeyer, T. Wehner, K. Urban, L. Claes), Landesschwerpunkt-Förderung Baden-Württemberg

Literaturverzeichnis

[ABS03] Peter Augat, Johannes Burger, Sandra Schorlemmer, Thomas Henke, Manfred Peraus, Lutz Claes: „Shear Movement at the Fracture Site Delays Healing in a Diaphyseal Fracture Model". In: Journal of Orthopaedic Research, Vol. 21, Issue 6, pp. 1011–1017. Orthopaedic Research Society, 2003.

[AHA95] C. Ament, E.P. Hofer, P. Augat, L. Claes: „Fuzzy Logic as a Method to Describe Tissue Adaption in Fracture Healing". In: Transactions of the 41st Annual Meeting of the Orthopaedic Research Society, pp. 228-238. 1995.

[AI03] „Beginners Guide to Pathfinding Algorithms – Blind Search". Artificial Intelligence Depot, 2003. http://ai-depot.com/Tutorial/PathFinding-Blind.html (accessed June 10, 2007)

[AP07] Amit J. Patel: „Dijkstra's Algorithm and Best-First-Search". Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/index.html (accessed June 11, 2007)

[AP07a] Amit J. Patel: „Heuristics". Amit's A* Pages, http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (accessed June 11, 2007)

[BCB89] Patricia R. Blenman, Dennis R. Carter, Gary S. Beaupré: „Role of Mechanical Loading in the Progressive Ossification of a Fracture Callus". In: Journal of Orthopaedic Research, Vol. 7, Issue 3, pp. 398–407. Orthopaedic Research Society, 1989.

[Ben91] Klaus-Uwe Benner: „Der Körper des Menschen". Weltbild Verlag GmbH, Augsburg, 1991. ISBN: 3-89350-273-4

[BM76] J.A. Bondy, U.S.R. Murty: „Graph Theory with Applications". Elsevier Science Publishing Co., Inc., New York 1976. ISBN: 0-444-19451-7

[CBB88] Dennis R. Carter, Patricia R. Blenman, Gary S. Beaupré: „Correlations between Mechanical Stress History and Tissue Differentiation in Initial Fracture Healing". In: Journal of Orthopaedic Research, Vol. 6, Issue 5, pp. 736–748. Orthopaedic Research Society, 1988.

[CMD91] J. Cheal, K.A. Mansmann, A.M. Di Gioia III, W.C. Hayes, S.M. Perren: „Role of Interfragmentary Strain in Fracture Healing: Ovine Model of a Healing Osteotomy". In: Journal of Orthopaedic Research, Vol. 9, Issue 1, pp. 131–142. Orthopaedic Research Society, 1991.

[CRD87] L. Claes, J. Reinmüller, L. Dürselen: „Experimentelle Untersuchungen zum Einfluß der interfragmentären Bewegungen auf die Knochenheilung". Aus: Hefte zur Unfallheilkunde, Ausgabe 189, S. 53–57. Springer Verlag, Berlin, Heidelberg, New York, 1987.

[CW99] Ray W. Clough, Edward L. Wilson: „Early Finite Element Research at Berkeley". 5th U.S. National Conference on Computational Mechanics, 1999. http://www.edwilson.org/History/fe-history.pdf

[DCH86] A.M. Di Gioia, E.J. Cheal, W.C. Hayes: „Three-Dimensional Strain Fields in a Uniform Osteotomy Gap". In: Journal of Biomechanical Engineering, Vol. 108, pp. 273–280. 1986.

[Die06] Reinhard Diestel: „Graphentheorie". 3. Auflage. Springer-Verlag, Heidelberg, 2006. ISBN: 3-540-21391-0

[DFG99] Neuantrag an die Deutsche Forschungsgemeinschaft auf Gewährung einer Sachbeihilfe. Thema: „Simulation des Knochenheilungsprozesses". Antragssteller: Lutz Claes. Ulm, 1999

[Dod00] Neil A. Dodgson: „Some Mathematical Elements of Graphics". University of Cambridge, 1999 – 2000. http://www.cl.cam.ac.uk/teaching/2000/AGraphHCI/SMEG

[Ein95] Thomas A. Einhorn: „Enhancement of Fracture-Healing". In: Journal of Bone & Joint Surgery. Issue 77/1995, pp. 940–956. Needham, MA, 1995.

[Grü06] Leonhard Grünschloß: „Experiences with Implementing Implicit kD-Trees for Isosurface Ray Tracing“. (unpublished) Universität Ulm, 2006.

[Hai94] Eric Haines: „Point in Polygon Strategies“. In: Graphics Gems IV, ed. Paul S. Heckbert, pp. 24-46. Academic Press, Boston, 1994. ISBN: 0-12-336155-9

[Hav04] Herman J. Haverkort: „Introduction to Bounding Volume Hierarchies“. Department of Mathematics and Computer Science, Technische Universiteit Eindhoven. Eindhoven, 2004. Available online: http://www.win.tue.nl/~hermanh/stack/bvh.pdf

[Hei98] Christa Heigele: „Finite-Elemente-Analysen zur Untersuchung des Gewebedifferenzierungsprozesses in der sekundären Frakturheilung“. Dissertation an der Medizinischen Fakultät der Universität Ulm. Ulm, 1998.

[HH94] A. Hammer, H. Hammer, K. Hammer: „Physikalische Formeln und Tabellen“. 6. Auflage. J. Lindauer Verlag (Schaefer). München, 1994. ISBN: 3-87488-064-8

[HS96] J. H. Hubbell, S. M. Seltzer: „Tables of X-Ray Mass Attenuation Coefficients and Mass Energy-Absorption Coefficients“. Ionizing Radiation Division, Physics Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, 1996 (Last Update: 2004), http://physics.nist.gov/PhysRefData/XrayMassCoef/cover.html

[Ish99] Akira Ishimaru: „Wave Propagation and Scattering in Random Media“. IEEE/Oxford University Press Series on Electromagnetic Wave Theory. Wiley-IEEE Press, 1999. ISBN: 978-0-7803-4717-5

[Käh05] Ralf Kähler: „Accelerated Volume Rendering on Structured Adaptive Meshes“. Dissertation am Fachbereich Mathematik und Informatik der Freien Universität Berlin. Berlin, September 2005

[Kel05] Alexander Keller: „Lee’s Maze Router Algorithm“. Aus: Scriptum zur Vorlesung „Praktische Informatik I“, S. 69. Universität Ulm, Wintersemester 2005/2006.

[Kel06] Alexander Keller: „Lichttransport“. Aus: Scriptum zur Vorlesung „Monte-Carlo- und Quasi-Monte-Carlo-Methoden in der photorealistischen Computergraphik“, S. 89–138. Universität Ulm, Wintersemester 2006/2007.

[KS88] A. C. Kak, M. Slaney: „Principles of Computerized Tomographic Imaging“. Originally published by IEEE Press, New York, 1988

[KW03] J. Krüger, R. Westermann: „Acceleration Techniques for GPU-based Volume Rendering“. In: Proceedings of the 14th IEEE Visualization 2003 (VIS ’03). IEEE Computer Society, Washington DC, 2003. ISBN: 0-7695-2030-8

[LL94] Philippe Lacroute, Marc Levoy: „Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation“. In: Proceedings of SIGGRAPH ’94, pp. 451–458. ACM Press, New York, July 1994. ISBN: 0-89791-667-0

[Max95] Nelson Max: „Optical Models for Direct Volume Rendering“. In: IEEE Transactions on Visualization and Computer Graphics. Volume 1, Issue 2, pp. 99–108. IEEE Educational Activities Department, Piscataway, 1995. ISSN: 1077-2626

[MFS06] Gerd Marmitt, Heiko Friedrich, Philipp Slusallek: „Interactive Volume Rendering with Ray Tracing“. Eurographics State-of-the-Art Report. Wien, September 2006

[MHBMC00] M. Meißner, J. Huang, D. Bartz, K. Mueller, R. Crawfis: „A Practical Evaluation of Popular Volume Rendering Algorithms“. In: Proceedings of the 2000 Volume Visualization Symposium, pp. 81-89. Salt Lake City, 2000

[Mor04] Moreland, K.D.: „Fast High Accuracy Volume Rendering“. PhD Thesis. University of New Mexico. Albuquerque, New York, Juli 2004

[MS06] Gerd Marmitt, Philipp Slusallek: „Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering“. In: Proceedings of the Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS) 2006. Lissabon, 2006

[MSDN07] „Common Type System Overview“. In: .NET Framework Developer’s Guide, MSDN Library. Microsoft Corporation, 2007. http://msdn2.microsoft.com/en-us/library/2hf02550(vs.80).aspx

[MSDN07a] .NET Framework SDK 2.0 Class Library Reference: System.Collections.Generic Namespace. MSDN Library. Microsoft Corporation, 2007. http://msdn2.microsoft.com/en-us/library/system.collections.generic(vs.80).aspx

[MT97] Tomas Möller, Ben Trumbore: „Fast, Minimum Storage Ray/Triangle Intersection“. In: Journal of Graphics Tools, Volume 2, Issue 1, pp. 21–28. AK Peters Ltd., Natick, 1997. ISSN: 1086-7651

[Not04] David Notario: „JIT Optimizations: Inlining (II)“. In: David Notario’s WebLog, MSDN. Microsoft Corporation, 2005. http://blogs.msdn.com/davidnotario/archive/2004/11/01/250398.aspx

[NS01] Jürgen Nehmer, Peter Sturm: „Systemsoftware. Grundlagen moderner Betriebssysteme“. 2. aktualisierte Auflage. dpunkt-Verlag, Heidelberg, 2001. ISBN: 3-89864-115-5

[Pau65] F. Pauwels: „Grundriß einer Biomechanik der Frakturheilung“. Aus: Gesammelte Abhandlungen zur funktionellen Anatomie des Bewegungsapparats, ed. F. Pauwels, pp. 139–182. Springer Verlag, Berlin, 1965.

[PiN99] „Visible Light and X-rays“. In: Patterns in Nature (Web-Course). Arizona State University. http://acept.la.asu.edu/PiN/rdg/visnxray/visnxray.shtml (accessed April 15, 2007).

[RN02] Stuart J. Russel, Peter Norvig: „Artificial Intelligence: A Modern Approach“. 2nd Edition. Prentice Hall International, 2002. ISBN: 978-0137903955

[SAC04] Ulrich Simon, Peter Augat, Lutz Claes: „3D Fracture Healing Model Can Help to Explain Delayed Healing with Interfragmentary Shear Movement Compared to Axial Movement“. In: Proceedings of the 6th International Symposium on Computer Methods in Biomechanics & Biomedical Engineering. Madrid, 2004.

[SAC04a] Ulrich Simon, Peter Augat, Lutz Claes: „Dealayed Healing of Fractures with Interfragmentary Shear Movement Can Be Explained Using a 3D Computer Model“. In: Proceedings of the 13th Conference of the European Society of Biomechanics. ’s-Hertogenbosch, Netherlands, 2004.

[Smi98] Brian Smits: „Efficiency Issues for Ray Tracing“. In: Journal of Graphics Tools, Volume 3, Issue 2, pp. 1–14. AK Peters Ltd., Natick, 1998. ISSN: 1086-7651

[SSKE05] Simon Stegmaier, Magnus Strengert, Thomas Klein, Thomas Ertl: „A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting“. In: Proceedings of the International Workshop on Volume Graphics ’05, pp. 187–195. Stony Brook, New York, 2005. ISBN: 3-905673-26-6, ISSN: 1727-8376

[ST02] Uwe Schöning, Jacobo Torán: „Berechenbarkeit und Komplexität“. Vorlesungsskript zur Vorlesung „Theoretische Informatik II“. Universität Ulm, 2002

[Stü87] K.M. Stürmer: „Histomorphologie der Frakturheilung im Vergleich der Fixationsverfahren am Tibiaschaft“. Aus: Die Tibiaschaftfraktur beim Erwachsenen, S. 23–49. Springer Verlag, Berlin, 1987.

[Sun01] Dan Sunday: „Fast Winding Number Inclusion of a Point in a Polygon“. http://softsurfer.com/Archive/algorithm_0103 (accessed May 23, 2007)

[Wal04] Ingo Wald: „Realtime Raytracing and Interactive Global Illumination“. PhD Thesis. Computer Graphics Group, Saarland University. Saarbrücken, 2004

[WBMS04] Amy Williams, Steve Barrus, R. Keith Morley, Peter Shirley: „An Efficient and Robust Ray-Box Intersection Algorithm". In: Journal of Graphics Tools. Volume 10, Issue 1, pp. 49–54. AK Peters Ltd., Natick, 2004

[WBS07] Ingo Wald, Solomon Boulos, Peter Shirley: „Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies". In: ACM Transactions on Graphics, Volume 26, Issue 1, Article No. 6. ACM Press, New York, 2007. ISSN: 0730-0301

[WFMS05] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, Hans-Peter Seidel: „Faster Isosurface Raytracing Using Implicit KD-Trees". In: IEEE Transactions on Visualization and Computer Graphics, Volume 11, Issue 5, pp. 562–572. IEEE Educational Activities Department, Piscataway, 2005. ISSN: 1077-2626

[WK06] Carsten Wächter, Alexander Keller: „Instant Ray Tracing: The Bounding Interval Hierarchy". In: Proceedings of Rendering Techniques 2006, Eurographics Symposium on Rendering. Eurographics Association, 2006. ISBN: 3-905673-35-5, ISSN: 1727-3463

[WoP07] Eric W. Weisstein: „Extinction Coefficient". In: Eric Weisstein's World of Physics, http://scienceworld.wolfram.com/physics/ExtinctionCoefficient.html (accessed May 15, 2007)

[WP07] „Compton scattering". In: Wikipedia, The Free Encyclopedia, Wikimedia Foundation, 2007. http://en.wikipedia.org/w/index.php?title=Compton_scattering&oldid=122778371 (accessed April 15, 2007)

[WP07a] „Komplexitätstheorie". In: Wikipedia, The Free Encyclopedia, Wikimedia Foundation, 2007. http://de.wikipedia.org/w/index.php?title=Komplexit%C3%A4tstheorie&oldid=31758724 (accessed June 9, 2007)

[WP07b] „Djikstra's Algorithm". In: Wikipedia, The Free Encyclopedia, Wikimedia Foundation 2007. http://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=136340402 (accessed June 11, 2007)

[ZPBG01] M. Zwicker, H. Pfister, J. van Baar, M. Gross: „EWA Volume Splatting“. In: Proceedings of the conference on Visualization ’01 , pp. 29–36. IEEE Computer Society Press, San Diego, 2001. ISSN: 1070-2385

Index

T

V

W

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, daß ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind angegeben.

Diese Arbeit wurde bisher weder veröffentlicht, noch einer anderen Prüfungskommision in gleicher oder ähnlicher Form vorgelegt.

Datum

Frank Niemeyer