# Interaktive Echtzeitmodellierung von biologischem Gewebe für Virtuelle Realitäten in der medizinischen Ausbildung

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Dipl.-Phys.
Johannes Grimm
aus Miltenberg

Mannheim, 2005

| | |
|---|---|
| Dekan: | Professor Dr. Matthias Krause, Universität Mannheim |
| Referent: | Professor Dr. Reinhard Männer, Universität Mannheim |
| Korreferent: | Professor Dr. Peter Fischer, Universität Mannheim |

Tag der mündlichen Prüfung: 19. Oktober 2005

*für meine Mutter*

# Zusammenfassung:

In der vorliegenden Arbeit wurde die interaktive Echtzeitmodellierung von biologischem Gewebe für Virtuelle Realitäten in der medizinischen Ausbildung diskutiert. Bestehende und neue Ansätze wurden vorgestellt, verglichen und bewertet. Die Implementierung der Algorithmen wurde dargestellt und evaluiert und die erstellten Anwendungen beschrieben.

## Modellierung von Deformationen:

Gängige Ansätze zur Modellierung von steifem Gewebe sind problematisch, wenn topologische Änderungen in Echtzeit modelliert werden müssen. Verfahren mit impliziter Integration benötigt für die Berücksichtigung der topologischen Änderungen zusätzliche Rechenzeit. Verfahren mit expliziter Integration konvergieren langsam und können bei der Simulation von steifem Gewebe numerisch instabil werden.

Der im Rahmen dieser Arbeit entwickelte Dragnet-Algorithmus ist ein numerisch stabiles, verschiebungsorientiertes Verfahren. Bei Dragnet sind jeweils zwei Knoten mit einem String-Element verbunden. Der String kontrolliert den Abstand der Knoten. Überschreitet der Abstand einen Grenzwert, wird er korrigiert. Die Korrektur erfolgt dabei gerichtet. Von den Interaktionsknoten ausgehend werden die Strings korrigiert. Die Reihenfolge der Korrektur wird durch die Überschreitung der maximalen String-Länge bestimmt.

Kombiniert mit einer Mass-Spring-Simulation erhöht Dragnet deren Konvergenzgeschwindigkeit und Stabilität. Verglichen mit anderen Ansätzen, die die Konvergenzgeschwindigkeit eines Mass-Spring-Ansatzes erhöhen, zeigt der Ansatz bei der Modellierung einer Membran den höchsten Geschwindigkeitszuwachs.

Der Dragnet-Algorithmus wurde unter Grimm, Wagner und Männer [2004] veröffentlicht.

## Modellierung von Reissen:

Das Modellieren von Reißen unter der Echtzeitrandbedingung ist ein wenig diskutiertes Thema in der wissenschaftlichen Literatur. Es wurde ein Algorithmus für ein flächenerhaltendes Reißen von Membranen vorgestellt.

Der Ansatz beruht auf einer Auswertung der Knotenspannungen. Um zu entscheiden, wo, wann und in welche Richtung gerissen wird, werden die Spannungen in den Knoten des Simulationsgitters ausgewertet. Hierbei wird zwischen drei Arten von Knoten unterschieden. Knoten innerhalb der Membran, am Rand der Membran und am Ende eines Risses haben unterschiedliche Schwellwerte für das Reißen. Dadurch ist es möglich, die Wahrscheinlichkeiten für Entstehung, Verzweigungen und Ausbreitung eines Risses zu beeinflussen. Die Modellierung der topologischen Änderungen wird durch

eine Trennung von Dreiecken realisiert. Die Rissausbreitung findet entlang der Dreieckskanten statt.

Der vorgestellte Algorithmus wird mit einem einfachen Reißalgorithmus, der auf dem Löschen von Federn in einem Mass-Spring-Gitter beruht, verglichen. Das flächenerhaltende Reißen zeigt vor allem bei groben Gittern ein deutlich realistischeres Verhalten. Dies wird allerdings mit einer etwas höheren Rechenzeit erkauft. Der vorgestellte Ansatz eignet sich im Besonderen, um das Reißen von steifen Membranen in Echtzeit zu simulieren. Durch die aufwendige Integration bei steifem Gewebe fällt die zusätzliche Rechenzeit nicht ins Gewicht. Bei der begrenzten Gitterauflösung, mit der steifes Gewebe in Echtzeit simuliert wird, ist der vorgestellte Algorithmus dem Reißen durch Löschen von Federn überlegen. Die Anpassbarkeit des Ansatzes ermöglicht sowohl die Modellierung des Reißverhaltens einer elastischen Membran sowie eines spröden Objekts.

Das flächenerhaltende Reißen wurde unter Grimm [2005b] veröffentlicht.

Software-Architektur:

Im Rahmen dieser Arbeit wurde eine Simulationsbibliothek erstellt. Die Bibliothek gliedert sich in eine Architektur ein, die als Basis für medizinische VR-Anwendungen dient. In der Implementierung repräsentiert sowohl eine Graph- als auch durch eine Arraystruktur die Simulationsdaten. Die Graphstruktur ermöglicht die schnelle Auswertung von Nachbarschaftsbeziehungen und die effiziente Berücksichtigung von topologischen Änderungen. Die der Graphstruktur zugrunde liegenden Graph-Elemente sorgen dafür, dass sich das Simulationsgitter immer in einem konsistenten Zustand befindet. Durch die Arraystruktur können Daten unter optimaler Ausnutzung des Prozessor-Caches abgearbeitet werden. Durchgeführte Messungen zeigen Geschwindigkeitsvorteile von bis zu 700% bei Integrationen über den Arrays im Vergleich zur Integration mit Zugriffen auf die Knoteneigenschaften des Simulationsgitters. Die erstellte Simulationsbibliothek ist Grundlagen der biomechanischen Modellierung in den VR-Simulatoren EYESI-*Vitreoretinal*, EYESI-*Cataract* und EndoSim.

Anwendungen:

Im Rahmen dieser Arbeit wurden vier Anwendungen erstellt.

Die erste Anwendung ist die Simulation der Ablösung der Inneren Grenzmembran im menschlichen Auge. Das Trainingsprogramm wurde auf Basis des ophtalmochirurgischen VR-Simulators EYESI-*Vitreoretinal* verwirklicht. Die Innere Grenzmembran wird mit einer Mass-Spring-Simulation modelliert. Der Dragnet-Algorithmus ermöglicht die Modellierung eines hinreichend steifen Gewebeverhaltens bei Interaktionen. Der flächenerhaltende Reißalgorithmus modelliert die Ausbreitung der Risse beim Ablösen der

Membran. Das Trainingsprogramm hat mittlerweile die Produktreife erreicht und wird bereits zu Ausbildung eingesetzt. Fachärzte bestätigen das realistische Gewebeverhalten der virtuellen Membran. Die Modellierung der Gewebedeformationen ist in Grimm et. al. [2004] beschrieben. Das verwendete Reißverfahren ist unter Grimm [2005b] erscheinen.

Ein weiteres Trainingsprogramm wurde für den EYESI-*Cataract* Simulator entwickelt. Das Programm ermöglicht das Training eines Capsulorhexis Eingriffs, bei dem aus der vorderen Linsenkapsel ein kreisförmiges Stück gerissen wird. Das Trainingsprogramm basiert auf einer Mass-Spring-Simulation. Die Rissausbreitung wird mit dem vorgestellten flächenerhaltenden Algorithmus modelliert. Die virtuelle Capsulorhexis befindet sich momentan in der letzten Phase des Prototypen-Stadiums und soll ab Mitte des Jahres als Modul von EYESI-*Cataract* zur Ausbildung eingesetzt werden. Fachärzte bewerten den virtuellen Eingriff als realistisch.

Für den Endoskopie-Simulator EndoSim wurde ein Modul entwickelt, mit dem die Entfernung eines Darmpolypen trainiert werden kann. Eine Finite-Elemente-Simulation modelliert die Deformationen des Polypen und der verwendeten Drahtschlinge. Eine Animationstechnik modelliert das Ein- und Ausfahren der Drahtschlinge. Das Abreißen des Polypen wird in einem Hintergrund-Thread vorbereitet, nachdem der virtuelle Polyp verödet wurde. Dadurch ist es möglich, das Abreißen des Polypen mit einem FEM-Ansatz in Echtzeit zu modellieren. Die Polypensimulation wird zurzeit in den Simulator EndoSim integriert. Eine Beschreibung des entwickelten Trainingsprogramms wurde unter Grimm [2005a] veröffentlicht.

Bei der Unterleibssimulation ging es darum, einen Eindruck von der Spannungsverteilung in Unterleib und Beckenboden während des Stuhlgangs zu bekommen. Mit der Finiten-Elemente-Methode wurde ein Querschnitt des Unterleibs dynamisch modelliert. Lage und Form des Beckens, der Wirbelsäule und die Parameter des Gewebes lassen sich in der Anwendung frei definieren, um ihre Auswirkungen auf die Spannungsverteilung zu bewerten. Die Validierung erfolgte durch den visuellen Vergleich mit dynamischen MR-Aufnahmen. Die Ergebnisse der Simulation sind unter Grimm, Schill, Männer, Lienemann und Janssen [2002] veröffentlicht.

# Vorwort

> „Computers are useless. They can only give you answers.“
>
> —Pablo Picasso[1]

Meinen ersten Kontakt mit dem Gebiet der Simulation von biologischem Gewebe hatte ich in der Schule. Wir mussten damals in einem unserer Leistungskurse eine Facharbeit schreiben. Eines der vorgeschlagenen Physik-Themen lautete: „Computersimulation eines Raumfahrers, der in ein schwarzes Loch gezogen wird.“

Ich dachte mir, man könnte den Raumfahrer durch Massepunkte beschreiben, die durch Federn verbunden sind. Allerdings war mir nicht klar, wie ich aus dem Modell die Deformationen des Raumfahrers berechnen konnte. Die Kräfte des Systems hängen von den Deformationen ab und die Deformationen von den Kräften. Die Werkzeuge der Schulmathematik reichten nicht aus, um solche Probleme zu lösen. Ich hatte die Idee, die Kräfte zu einem Zeitpunkt zu berechnen. Anschließend könnte ich die Kräfte für eine kurze Zeit als konstant annehmen und über die Beschleunigung und Geschwindigkeit die Verschiebungen der Massepunkte ermitteln. Dann könnte ich die Kräfte neu berechnen und fortfahren. Das Ganze gefiel mir aber gar nicht. Die Kraftberechnungen basierten auf den nur näherungsweise bestimmten Verschiebungen und in jedem Zeitschritt kam eine neue Näherung dazu. Ich glaubte, mit meinem Ansatz keine brauchbaren Ergebnisse erzielen zu können und wählte ein anderes Thema. Heute, gut zehn Jahre später, schreibe ich meine Dissertation über die Arbeit mit Gewebedeformationen, die unter anderem die oben beschriebene Art von expliziter Integration verwenden.

Jahre nach meinem Abitur stieß ich wieder auf das Thema biomechanische Simulationen. Ich war auf der Suche nach einer Diplomarbeit zum Abschluss meines Physikstudiums. Ich erkundigte mich unter anderem bei der ViPA Arbeitsgruppe am Lehrstuhl für Technische Informatik V in Mannheim.

[1] Spanischer Künstler 1881–1973

Als mir der Gruppenleiter Markus Schill vorschlug, ein diffus schwellendes Gehirn mit der Methode der Finiten-Elemente zu simulieren, sagte ich sofort zu.

Nach Abschluss meiner Diplomarbeit hatte ich ein schwellendes Gehirn quasi-statisch simuliert. Obwohl die Arbeit an meiner Diplomarbeit sehr interessant war, blieben einige Fragen offen, die mich interessiert hatten. Ich wollte gerne dynamisch simulierten, um zu sehen, wie sich das Objekt verformt. Und ich fand die Vorstellung spannend, mit dem simulierten Objekt interagieren zu können. Aus diesem Grund war ich begeistert, als ich die Möglichkeit bekam, meine Kenntnisse über biomechanische Simulationen in einer Doktorarbeit zu vertiefen.

Jetzt am Ende meiner Dissertation habe ich das Gefühl viel gelernt, zu haben. Vielen Fragen haben sich beantwortet. Noch mehr Fragen sind neu aufgetaucht. Es gibt immer noch viel Interessantes, was auf dem Gebiet der biomechanischen Simulationen gemacht werden kann. Und ich habe immer noch keinen Raumfahrer simuliert, der in ein schwarzes Loch fällt.

An dieser Stelle möchte ich Prof. Dr. Reiner Männer danken, der mir ermöglichte, mich in den letzten Jahren mit dem spannenden Thema der Modellierung von biologischem Gewebe zu beschäftigen. Mein weiterer Dank gilt Prof. Dr. Peter Fischer, der sich als Zweitkorrektor zur Verfügung gestellt hat.

Weiter danke ich der den Mitgliedern der ViPA Arbeitsgruppe, in der ich mich sehr wohl gefühlt habe. Mein besonderer Dank geht dabei an Markus Schill den visionären Gruppenleiter, der immer die großen Ziele im Auge behalten hat. Weiter danke ich: Clemens Wagner für die vielen effektiven Diskussionen, Andreas Köpfle, der die Arbeitsgruppe immer über die neusten Entwicklungen in C++ auf dem Laufenden gehalten hat, Thomas Ruf, der auf jede technische Frage eine Antwort parat hat und Olaf Körner, für viele Anregungen und Hilfe bei der Entwicklung der Polyen-Simulation.

Außerdem danke ich der Firma VRmagic für die gute Zusammenarbeit bei der Entwicklung der Trainingsprogramme für den von ihr vertriebenen Simulator EYESI.

Ich danke Clemens Wagner, der die Arbeit Korrektur gelesen hat.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| BEM | Boundary-Element-Method |
| EYESI | Eye Surgery Simulation |
| FDM | Finite-Differenzen-Methode |
| FEM | Finite-Element-Methode |
| FPGA | Field-Programmable Gate Array |
| FVM | Finite-Volumen-Methode |
| GPU | Graphics Processing Unit |
| ILM | Innere Laminarmembran |
| KISMET | Kinematic Simulation, Monitoring and Off-Line Programming Environment for Telerobotics |
| LOD | Level-Of-Detail |
| MGE | Multigraphelement |
| OpenGL | Open Graphics Library |
| PHANToM | Personal HAptic iNTerface Mechanism |
| SOFA | Simulation Open Framework Architecture |
| VR | Virtuelle Realität |
| VRM | Virtuelle Realität in der Medizin |
| ViPA | Virtual Patient Analysis |

# 1

# Einleitung

‚I've come up with a set of rules that describe our reactions to technologies. Anything that is in the world when you're born is normal and ordinary and is just part of the way the world works. Anything that's invented between when you're fifteen and thirty five is new and exciting and revolutionary and you can probably get a career in it. Anything invented after you're thirty-five is against the natural order of things.'

—Douglas Adams[1]

## 1.1 Virtuelle Realitäten

Der griechische Philosoph Plato (427 v. Chr. – 347 v. Chr.) beschreibt im siebten Buche der Politeia mit dem Höhlengleichnis die vielleicht erste Beschreibung einer virtuellen Realität:

*Einige Menschen werden von Geburt an im Inneren einer Höhle so festgebunden, dass sie nur nach vorne in Richtung einer Höhlenwand blicken können. Hinter den Gefangenen ist ein Feuer, das Schatten auf die Höhlenwand wirft. Menschen tragen Formen, die der Silhouette von Tieren, Pflanzen und anderen Gegenständen entsprechen zwischen dem Feuer und den Gefangenen hin und her. Durch das Feuer werden die Formen auf die*

[1] Britischer Autor (1952–2001)

*Höhlenwand projiziert. Die Geräusche, die die Formenträger machen, werden von der Höhlenwand reflektiert, sodass die Gefangenen den Eindruck haben sie kämen von den sich bewegenden Schatten.*

Eine Virtuelle Realität in ihrer Idealform ist eine künstlich geschaffene Welt, die real wirkt. Der Benutzer bemerkt keinen wesentlichen Unterschied zu einer wirklichen Umgebung.

Gehen wir, anders als Plato, von einem Menschen aus, der in der realen Welt aufgewachsen ist, müssen die Sinne auf die gleiche Weiße stimuliert werden wie sie auch in der Realität werden.

Besonders die Interaktion zwischen Mensch und virtueller Welt ist entscheidend für die Entstehung von Immersion[2]. Folgende Punkte sind bei Modellierung von Interaktionen zu beachten.

- Die modellierte Welt muss sich plausibel verhalten. Interaktion zwischen Benutzer und Weltmodell dürfen nicht im Widerspruch zu Erfahrungen aus der realen Welt stehen.
- Alle relevanten Sinne müssen adäquat stimuliert werden, um einen realen Eindruck zu hinterlassen. Räumliche und zeitliche Diskretisiertung müssen für jeden Wahrnehmungskanal ausreichend hoch sein und die vermittelten Eindrücke müssen plausibel erscheinen.
- Die Interaktionen zwischen Benutzer und virtueller Umgebung müssen ohne merkliche Latenz ablaufen.

## 1.2 Virtuelle Trainingssimulatoren für die medizinische Ausbildung

Mit dem medizinischen Fortschritt etablieren sich immer neue und komplexere Operationstechniken. Die neuen Eingriffe müssen erlernt und komplexe Operationen müssen praktiziert werden, um einen qualitativ hochwertigen medizinischen Standard zu etablieren und zu halten.

Praktisches Üben von Operationen kann an Tieren, Leichen oder Patienten erfolgen. Neben den ethischen und moralischen Fragstellungen gibt es bei allen drei Arten noch andere Probleme.

Tier und Mensch haben häufig in relevanten Bereichen unterschiedliche Anatomie und Physiologie. Bestimmte Pathologien kommen bei Tieren nicht oder nur selten vor. Die Züchtung von Tieren mit bestimmten Krankheiten wirft ethische Fragen auf.

[2]Immersion bedeutet Eintauchen in eine Virtuelle Realität

Bei Training mit totem Gewebe bleiben die physiologischen Reaktionen des Gewebes aus. Das Gewebe verhält sich physikalisch anderes als bei einem lebendigen Menschen.

Beim Üben am Patienten scheidet das Erlernen von neuen Methoden aus ethischen Gründen aus. Um die Übung bei einer schon erlernten Methode nicht zu verlieren, ist es am besten die Methode häufig zu praktizieren. Auf selten vorkommende anatomische Besonderheiten oder manche Komplikationen kann aber die Praxis nicht ausreichend vorbereiten. Seltene Eingriffe werden zu wenig trainiert.

Virtuelle Trainingssimulatoren haben diese Probleme prinzipbedingt nicht. Dass virtuelles Training die Leistungen im Operationssaal verbessert haben Seymour, Gallagher, Roman, O'Brien, Bansal, Andersen und Satava [2002] in einer Doppelblind-Studie nachgewiesen. Zusätzlich gibt es noch andere Gründe, die für den Einsatz von VR bei der medizinischen Ausbildung sprechen.

Pädagogisch ausgearbeitete Curricula bieten strukturierte Trainingkurse von den ersten Schritten beim Erlernen eines Eingriffs bis zur Behandlung von komplexen Komplikationen bei schwierigen anatomischen Besonderheiten. Die virtuellen Operationen werden objektiv bewertet. Daraus lassen sich Begabung und Qualifikation eines Trainierenden bemessen. So kann beispielsweise bereits im Vorfeld der Ausbildung die Eignung von Bewerbern getestet werden. Ein bestimmtes Grundniveau der Ausbildung kann durch eine objektive Abschlussprüfung am Simulator sichergestellt werden.

Das Training am Simulator kann ohne Aufsicht erfolgen. Das vermindert Ausbildungskosten und ermöglicht es dem Auszubildenden wann und so oft er will zu trainieren. Virtuelle Operationen können gespeichert werden, um sie anschließend auszuwerten oder von einem Experten analysieren zu lassen.

## 1.3 Schlüsseltechnologien für die Entwicklung medizinischer VR Simulatoren

Die Entwicklung von Virtuellen Realitätssimulatoren für die Medizin ist aktuelles Thema der Forschung. Forschungsgruppen weltweit beschäftigen sich mit der Entwicklung der dazu nötigen Schlüsseltechnologien.

**Benutzerschnittstelle** Die Benutzerschnittstellen unterteilt man in Eingabe- und Ausgabeschnittstellen. Die Eingabeschnittstelle ist häufig ein Trackingsystem, das Bewegungen des Benutzers erfasst und in die virtuelle Welt überträgt. Die Ausgabe der künstlichen Umgebung erfolgt in der

Regel visuell oder haptisch. Die virtuelle Welt wird computergrafisch aufbereitete, um einen fotorealistischen Eindruck zu machen. Häufig kommen auch 3D Ausgabegeräte wie Head-Mounted-Displays oder Stereobildschirme zum Einsatz. Force-Feedback-Vorrichtungen übertragen die virtuellen Kräfte der modellierten Welt auf den Benutzer.

**Kollisionserkennung und Kollisionsantwort** Anders als in unserer materiebehafteten aufgebauten Umgebung durchdringen sich virtuelle Objekte. Algorithmen zur Kollisionserkennung und Kollisionsantwort erkennen Durchdringungen und lösen sie auf.

**Gewebesimulation** Die Gewebesimulation bildet vielleicht den Hauptunterschied zwischen medizinischen und anderen VR-Simulatoren. Biologisches Gewebe ist relativ weich und elastisch deformierbar. Bei der Interaktion mit dem Gewebe kommt es zu Deformationen, zum Schneiden und zum Reißen von Gewebe. Simulationsalgorithmen beschreiben das Gewebeverhalten bei Interaktionen des Benutzers. Verschiebungen, Rotationen und Deformationen werden berechnet und der Visualisierung zur Verfügung gestellt. Die Algorithmen liefern außerdem die für das haptische Feedback nötigen Kräfte.

**Curriculum und Bewertung** Pädagogische ausgearbeitete Curricula garantieren eine kosten- und zeiteffiziente Ausbildung. Ein Bewertungssystem beurteilt Eignung von Qualifikation des Auszubildenden und sichert das Qualitätsniveau der Ausbildung.

**Validierung des Simulators** Die Validierung von Trainingsprogrammen stellt sicher, dass relevante Sachverhalte realistisch abbildet werden. Studien garantieren den positiven Trainingseffekt virtueller Eingriffe.

## 1.4 Zielsetzung und Schwerpunkte

Die vorliegende Arbeit beschäftigt sich mit dem Gebiet der Simulation von biologischem Gewebe.

Ziel der Arbeit war die Implementierung bestehender und die Entwicklung neuer Techniken zur interaktiven Echtzeitmodellierung von biologischem Gewebe für die medizinische Ausbildung an Virtuellen Realitätssimulatoren.

Das Augenmerk bei der Implementierung lag auf der universellen Einsetzbarkeit der Algorithmen und der Möglichkeit, die Ansätze ohne großen Aufwand beliebig kombinieren zu können.

Die entwickelte Bibliothek bildet den Simulationsteil der *VRM-Architektur* (siehe Wagner [2003]), auf der die VR-Simulatoren EYESI (siehe Schill, Wagner, Hennen, Bender und Männer [1999b]) und EndoSim (siehe Körner und Männer [2002]) basieren.

Bei allen Entwicklungen wurde auf Praxistauglichkeit geachtet. Im Rahmen der Arbeit wurden folgende vier Anwendungen entwickelt.

- Interaktive Echtzeitsimulation der Ablösung der obersten Schicht der Retina, der so genannten Inneren Grenzmembran auf Basis des VR-Trainingssimulators EYESI-*Vitreoretinal*.
- Virtueller Capsulorhexis Eingriff als Trainingsmodul für EYESI-*Cataract*. Die interaktive Echtzeitsimulation trainiert die Entfernung eines kreisförmigen Stücks des vorderen Kapselsacks.
- Entfernung eines Darmpolypen als interaktives Trainingsprogramm für den Endoskopiesimulator *EndoSim*.
- Prototyp einer Simulation, die die Spannungen im Unterleib während des Stuhlgangs abschätzt. Ziel ist, die Zusammenhänge zwischen Knochenanordnung und Belastungen des Beckenbodens zu untersuchen.

Neben den erstellten Anwendungen sind die Schwerpunkte der Arbeit der entwickelte *Dragnet*-Algorithmus und eine flächenerhaltenden Reißmethode.

Dragnet wurde entwickelt, um die unbefriedigende Konvergenzgeschwindigkeit bei Mass-Spring-Simulationen mit expliziter Integration zu verbessern.

Der Reißalgorithmus wurde für die EYESI-Trainingsprogramme entwickelt, um die Kontrolle der Rissausbreitung bei den jeweiligen Eingriffen virtuell üben zu können.

## 1.5 Überblick über die Arbeit

Die Arbeit gliedert sich in vier Kapitel.

Kapitel 2 beschäftigt sich mit der Simulation von deformierbaren Körpern. Bestehende Ansätze werden diskutiert und ein im Rahmen dieser Arbeit entwickelter Dragnet-Algorithmus vorgestellt. Die Eigenschaften des Dragnet Ansatzes werden untersucht. Ein Vergleich mit gängigen Ansätzen untersucht die Eignung von Dragnet zur Modellierung von steifem Gewebe bei großen lokalen Interaktionen.

In Kapitel 3 wird das Problem des Reißens von Gewebe diskutiert. Der Stand der Technik beschäftigt sich auch mit den verwandten Themen: dem Schneiden von Gewebe und der Animation von Brüchen. Ein flächenerhaltender Reißalgorithmus wird vorgestellt. Der Reißalgorithmus wird mit einem bestehenden Ansatz verglichen, bei dem das Reißen durch Löschen von Federn in einem Mass-Spring-Gitter modelliert wird.

Die Implementierung der Simulationsbibliothek wird in Kapitel 4 beschrieben. Neben der Architektur der Bibliothek wird hier besonders auf die duale Repräsentation der Simulationsdaten in einer Gitter- und einer Array-Struktur eingegangen.

Die im Rahmen der vorliegenden Arbeit entwickelten Anwendungen werden in Kapitel 5 vorgestellt. Abschnitt 5.1 beschreibt ein Trainingsprogramm, mit dem die Ablösung der obersten Schicht der Retina virtuell trainiert werden kann. Ein virtueller Capsulorhexis-Eingriff wird in Abschnitt 5.2 vorgestellt. Abschnitt 5.3 präsentiert eine Anwendung, um die Entfernung eines Darmpolypen zu trainieren. Die Simulation in Abschnitt 5.4 ist kein Trainingsprogramm, sondern dient dazu, die Spannungen im Unterleib beim Stuhlgang zu untersuchen.

Kapitel 6 diskutiert die Ergebnisse der vorliegenden Arbeit und gibt einen Ausblick auf zukünftige Entwicklungen.

Der Anhang beinhaltet einige zusätzliche Informationen, die keinem der obigen Kapitel zuzuordnen sind. Anhang A stellt einige der Grundlagen der Dynamik deformierbarer Körper zusammen. In Anhang B werden die verwendeten Integrationsverfahren zusammengefasst. Die Herleitung der implementierten Finiten-Elemente ist in Anhang C zu finden.

# 2

# Modellierung von Gewebedeformationen

„All science is either physics or stamp collecting.“[1]

—Ernest Rutherford[2]

Dieses Kapitel beschäftigt sich mit der Modellierung deformierbarer Körper. Insbesondere wird auf das Problem der Echtzeitmodellierung von steifen Objekten eingegangen, die während der Simulation topologisch verändert werden. Speziell wurde im Rahmen dieser Arbeit ein Algorithmus gesucht, mit dem sich Deformationen und topologische Änderungen von steifen Membranen modellieren lassen.

Zunächst werden in Abschnitt 2.1 einige Vorbemerkungen über verschiedene Klassen von Simulationen (Abschnitt 2.1.1), numerische Integration (Abschnitt 2.1.3) und über Simulationsgitter (Abschnitt 2.1.2) gemacht. Anschließend wird in Abschnitt 2.2 der Stand der Technik vorgestellt. Abschnitt 2.3 beschreibt den im Rahmen der vorliegenden Arbeit entwickelten Dragnet Algorithmus. Dragnet dient dazu, Interaktionen mit einer steifen Membrane in Echtzeit und unter Berücksichtigung topologischer Änderungen zu simulieren. Neben dem Konzept von Dragnet (Abschnitt 2.3.1) und dem Aufbau des Algorithmus (Abschnitt 2.3.2) werden in Abschnitt 2.3.4 die Ergebnisse des Ansatzes diskutiert. Hierbei wird auf die Konvergenz (Abschnitt 2.3.4) und die Geschwindigkeit (Abschnitt 2.3.4) eingegangen und Dragnet mit anderen Ansätzen aus dem Stand der Technik verglichen (Abschnitt 2.3.5).

[2] Britischer Physiker (1871–1937)

## 2.1 Vorbemerkungen

### 2.1.1 Klassifizierung der Ansätze

Es existiert eine Vielzahl verschiedener Ansätze, um Deformationen von biologischem Gewebe zu modellieren. Dieser Abschnitt gibt einen Überblick über verschiedene Kriterien, mit deren Hilfe sich Simulationsalgorithmen einteilen lassen.

#### Elastische und plastische Verformung

Modellierungsansätze kann man nach der Art der modellierten Verformung in elastische und plastische Deformationen unterscheiden. Wird ein Körper aufgrund von äußeren Zwängen plastisch verformt, bleiben die Deformationen auch nach Ende des Zwangs erhalten. Nach elastischen Verformungen relaxiert das Gewebe hingegen in seinen Ausgangszustand oder eine energetisch äquivalente Form. In der Realität kommen häufig Verformungen vor, die sowohl eine elastische als auch eine plastische Komponente haben. Der Köper relaxiert in Richtung seines Ausgangszustandes erreicht ihn aber nicht vollständig. Dieser *Hysterese-Effekt* tritt meist bei starker (Über-) Dehnung des Materials auf.

#### Herangehensweise bei der Modellerstellung

Schill, Wagner, Hennen, Jendritza, Knorz, Bender und Männer [1999a] unterscheiden aufgrund der Herangehensweise bei der Modellerstellung zwischen deskriptiver und physikalischer Modellierung.

**Physikalische Modellierung** Eine physikalische Modellierung basiert auf einer physikalischen Theorie. Beschrieben wird ein Problem in der Regel durch ein Differenzialgleichungssystem, das unter Berücksichtigung der geltenden Randbedingungen numerisch gelöst wird. Im Fall der Gewebedeformation bildet die Elastizitätstheorie, wie sie in Anhang A beschrieben wird, die Grundlage der Modellbildung. Die freien Parameter einer physikalischen Simulation entsprechen physikalischen Größen oder lassen sich aus diesen herleiten.

Die Parameter der Elastizitätstheorie sind das *Elastizitäts-* oder auch *Young's Modul*, die *Poisson-Zahl* und eventuell das *Schermodul*. Häufig wird in der Literatur auch die äquivalente Darstellung durch die *Lamé*-Koeffizienten verwendet. Die Parameter des Modells lassen sich prinzipiell durch Messungen bestimmen. In der Praxis findet man in der Literatur nur

bedingt die benötigten Messwerte. Ein Grund sind die technischen und ethischen Probleme bei Messungen an lebendem Gewebe. Messungen an totem Gewebe führen nicht zu den gewünschten Ergebnissen, weil das Gewebe degeneriert und physiologische Reaktionen ausbleiben. Anatomische Unterschiede bei Tieren macht die Übertragung physikalischer Parameter auf den Menschen schwierig. Davon abgesehen variieren die physikalischen Parameter individuell je nach Alter, genetischer Veranlagung und möglicherweise vorhandener Pathologie. Der komplizierte Aufbau von biologischem Gewebe führt zu einem Verhalten, das in der Regel nicht linear und anisotrop ist und Hysterese-Effekte aufweist.

Physikalische Modelle erreichen prinzipiell eine beliebige Genauigkeit, wenn Parameter und Randbedingungen genau genug bekannt sind. Große Genauigkeit ist allerdings mit einer Komplexität und demzufolge einen hohen Rechenaufwand verbunden. Für Echtzeitsimulationen geht man deshalb in der Regel von vereinfachten Annahmen aus, die die Komplexität verringern aber das Gewebeverhalten möglichst nicht relevant beeinflussen.

**Deskriptive Modelleierung** Bei der deskriptiven Modellierung beobachtet man das Verhalten des Gewebes bei allen vorkommenden Interaktionen. Anschließend wird ein Modell entwickelt, das die beobachteten Gewebereaktionen qualitativ abbildet.

Die freien Parameter in einem deskriptiven Modell entsprechen keinen physikalischen Größen und können auch nicht direkt gemessen werden. In der Regel werden die Parameter durch gezieltes Ausprobieren angepasst. Kann das gewünschte Verhalten durch eine Anpassung der Parameter nicht erreicht werden, muss das Modell modifiziert werden. Deskriptive Modellierung ist meist ein iterativer Prozess, wobei die Komplexität des Modells tendenziell mit jedem Schritt zunimmt, bis das Modell die Beobachtungen widerspiegelt. Die Übertragung eines deskriptiven Modells von dem beobachteten Vorgang auf einen anderen Vorgang ist nur bedingt möglich. Extrapolationen des Gewebeverhaltens führen meist zu unbrauchbaren Ergebnissen.

### Grundlage der Modellierung

Es existieren zwei Ansätze, auf deren Basis die Verformungen eines Körpers berechnet werden können.

**Kraftbasierter Ansatz** Grundlage der kraftbasierten Ansätze sind die inneren und äußeren Kräfte, die auf ein Objekt wirken. Die inneren Kräfte sind eine Folge elastischer Verformungen und hängen von den Spannungen innerhalb des Gewebes ab. Äußere Kräfte entstehen durch Interaktion eines

Objektes mit seiner Umgebung oder dem Benutzer. Um von den Kräften auf die gesuchten Verschiebungen zu kommen, ist ein Integrationsschritt erforderlich. Abschnitt 2.1.3 gibt einen Einblick in die Probleme der numerischen Integration. Einige relevante Integrationsverfahren sind in Anhang B beschrieben. An dieser Stelle sei nur erwähnt, dass nicht alle Integrationsverfahren für beliebige Objekte numerisch stabil sind.

**Verschiebungsorientierter Ansatz** Neben der kraftbasierten gibt es verschiebungsorientierte Verfahren. Interaktionen werden durch Verschiebungen beschrieben, die direkt zu einer Verformung des Objekts führen. Kräfte werden dabei nicht berechnet und können auch nicht direkt berücksichtigt oder ausgewertet werden. Verschiebungsorientierter Ansätze benötigen keinen Integrationsschritt und sind deshalb in der Regel unbedingt stabil.

### Echtzeitverhalten

Eine weitere Möglichkeit der Klassifizierung von Simulationsmethoden ist ihre Eignung für Echtzeitanwendung. Ein Virtuelles Realitätssystem erfüllt die Echtzeit-Randbedingung, wenn der Benutzer des Systems den Eindruck hat, das alle Veränderungen der Umgebung, mit der er interagiert kontinuierlich und ohne Verzögerung ablaufen.

Das menschliche Auge besitzt eine Abtastrate im Bereich von ungefähr 50 Hz. Damit kann es nach dem Abtasttheorem von *Nyquist* Frequenzen bis zu 25 Hz auflösen. Das bedeutet für ein Virtuelles Realitätssystem, dass mindestens alle 40 ms ein neu berechneter Zeitschritt visualisiert werden muss, damit der Benutzer die Bewegung als kontinuierlich wahrnimmt. Wie viele Rechenoperationen einer Simulation unter Echtzeitrandbedingung zur Verfügung stehen, hängt entscheidend von dem System ab, auf dem die Simulation läuft. Entscheidend ist zum einen die Geschwindigkeit der zugrunde liegenden Hardware, zum anderen die Software des Systems, die die begrenzte Rechenzeit pro Zeitschritt auf verschiedene Aufgaben aufteilen muss. Je nach System benötigen Computergrafik für die Visualisierung, Force-Feedback für das haptische Interface und Tracking für die Eingabeschnittstelle einen großen Teil der verfügbaren Rechenzeit. Für die Simulation ist neben der Berechnung der Deformationen, Rotationen und Translationen der Objekte auch noch ein zeitaufwendiger Kollisionserkennungs-Schritt nötig. Für die eigentliche Gewebesimulation stehen so oft nur 10 bis 20 ms an Rechenzeit pro Zeitschritt für Verfügung.

Der Tastsinn des Menschen besitzt eine weitaus höhere zeitliche Auflösung als der visuelle Wahrnehmungskanal. Er kann Frequenzen im kHz Bereich

auflösen. Deshalb muss das haptische Rendering mit wenigstens 1000 Hz erfolgen.

Wagner [2003] beschäftigt sich ausführlich mit dem Thema Latenz und kommt zu dem Schluss, dass die Latenz-Obergrenze bei wenigen 10 ms liegen muss, um nicht wahrgenommen zu werden.

Für Simulationen, bei denen die Rechenzeit von der aktuellen Konfiguration des simulierten Objektes abhängt, ist das Zeitverhalten im *Worst Case* entscheidend.

Es kann nicht direkt von einem Simulationsansatz auf dessen Echtzeitfähigkeit geschlossen werden, da die benötigte Rechenzeit für einen Zeitschritt in erster Linie von der verwendeten Diskretisierung abhängt. Trotzdem sind einige Ansätze besser für Echtzeitanwendungen geeignet als andere, weil bei ihnen die Rechenzeit bei der Erhöhung der Diskretisierung langsamer ansteigt.

### Berücksichtigung topologischer Änderungen

Während virtuellen Eingriffen müssen die simulierten Objekte oft topologisch verändert werden. Der Operateur schneidet oder zerreißt Gewebe und Organe, sägt und bohrt in und durch Knochen und vernäht oder verklebt Wunden und Öffnungen. Kapitel 3 geht auf die Modellierung von Schneiden, Brechen und Reißen ein. Dabei geht es aber in erster Linie um die Bestimmung, wann, wo und in welche Richtung das Gewebe reißt und wie Risse und Schnittlinien allgemein modelliert werden. Die Frage, mit welchen Simulationsalgorithmen sich topologische Veränderungen ohne großen zusätzlichen Rechenaufwand berücksichtigen lassen, wird in Abschnitt 2.2 diskutiert.

### Simulation und Animation

Logan, Wills und Avis [1994] unterscheiden zwischen physikalischen Simulationen und Animationen. Animationen basieren dabei auf Regelsystemen oder einer direkten Bewegungskontrolle wie bei der Keyframe Technik. Diese Unterscheidung hat sich in der Literatur allerdings nicht durchgesetzt. Im Bereich der deskriptiven Modelle werden Animation und Simulation oft synonym verwendet. Selbst kontinuumsmechanische Ansätze werden im Bereich der Computergrafik häufig als Animation bezeichnet und für manche verschiebungsorientierte Methode hat sich die Bezeichnung Simulation eingebürgert. Auch die Echtzeitfähigkeit kann nicht als Unterscheidungskriterium herangezogen werden. Im Folgenden wird deshalb keine Unterscheidung zwischen Simulation und Animation gemacht.

### 2.1.2 Der Gitter-Begriff

Bei den meisten medizinischen Simulationen wird das simulierte Objekt durch ein *Gitter* aus Knoten und Verbindungen repräsentiert. Der deutsche Begriff Gitter suggeriert eine regelmäßige und starre Gitterstruktur wie sie beispielsweise bei Strömungsberechnungen (siehe Bathe [1990]) verwendet werden. Deformationssimulationen verwenden in Regel ein Gitter, das unregelmäßig ist und sich während der Simulation verformt. Die englische Bezeichnung *Mesh* (Netz, Masche) ist deshalb der intuitivere Begriff.

Häufig bestehen Gitter aus Massepunkten, die mit Simulationselementen verbunden sind, die die physikalischen Wirkungen zwischen den Knoten untereinander beschreiben. Diese Elemente können beispielsweise Federn, virtuelle Kettenglieder oder elastische Volumenelemente sein. Meist sind Knotenpunkte an der Oberfläche der simulierten Objekte zusätzlich mit Dreiecken verbunden, die für die Visualisierung benötigt werden.

Hoch aufgelöste Gitter können ein Objekt besser beschreiben als eine grobe Gitterstruktur. Auf der anderen Seite steigt die benötigte Rechenzeit mit der Anzahl der Knoten und Elemente an. Eine Gitter-Diskretisierung ist daher immer ein Kompromiss zwischen Genauigkeit und Geschwindigkeit. Um eine ausreichende Feinheit unter der Echtzeitrandbedingung zu erreichen, passen Debunne, Desbrun, Barr und Cani [1999] die Auflösung des Gitters bei ihrer Gewebesimulation lokal während der Laufzeit an. Bereiche des Gitters, die nach einer höheren Genauigkeit verlangen, werden verfeinert. Feine Bereiche werden wieder vergröbert, wenn eine hohe Genauigkeit nicht mehr erforderlich ist. Der Abstand zwischen Nachbarknoten wird ausgewertet, um zu entscheiden, welche Bereiche verfeinert oder vergröbert werden. Bei Debunne, Desbrun, Cani und Barr [2000] und Wu, Downes und Gotektekin [2001] wird als Kriterium der Fehler der Simulation benutzt.

Die gitterlosen Verfahren, die in Abschnitt 2.2.3 beschrieben werden, basieren ebenfalls auf Knotenpunkten. Es gibt aber keine starren Verbindungen zwischen den Knoten. Die physikalischen Eigenschaften werden für eine Knotenumgebung festgelegt.

### 2.1.3 Numerische Integration

Im Folgenden soll kurz auf die Begriffe *Numerische Integration* und *Numerische Stabilität* eingingen werden. Eine ausführliche Behandlung des Themas würde den Rahmen dieser Arbeit sprengen. Einige Integrationsverfahren, die häufig bei Gewebesimulation zum Einsatz kommen, sind in Anhang B aufgelistet. An dieser Stelle sollen spezielle Aspekte der numerischen Integration plausibel gemacht werden, ohne exakte Herleitungen anzugeben.

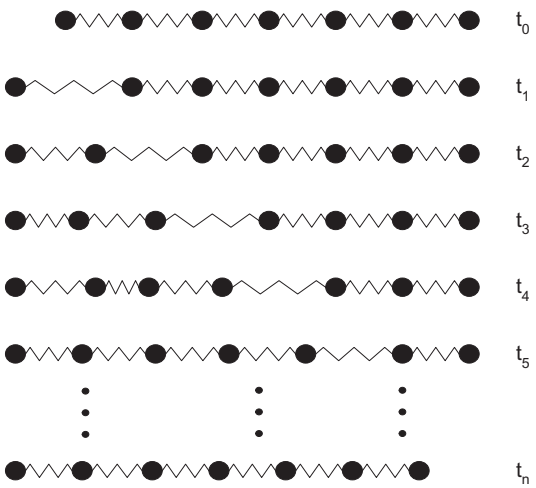


**Abbildung 2.1:** Der linke Knoten wird zum Zeitpunkt $t_1$ nach links verschoben. Bei einer einfachen expliziten Integration breiten sich lokale Störungen von Zeitschritt zu Zeitschritt nur um jeweils einen Knoten aus.

Bei der Modellierung Virtueller Realitäten wird die simulierte Welt räumlich und zeitlich diskretisiert. Durch die Diskretisierung entstehen Fehler. Verstärken sich die Fehler von Iterationsschritt zu Iterationsschritt, wird das System numerisch instabil. Werden die Fehler verringert, ist das System numerisch stabil.

Verschiebungsorientierte Verfahren verwenden keinen Integrationsschritt. Sie sind in der Regel numerisch stabil. Auftretende Fehler haben im nächsten Simulationsschritt keine verstärkende Wirkung.

Bei kraftbasierten Modellen hängt die numerische Stabilität vom simulierten System und dem verwendeten Integrationsverfahren ab.

**Explizite Integration** Bei der *expliziten Integration* werden aufgrund der Kräfte zum Zeitpunkt $t$ die Verschiebungen zum Zeitpunkte $t + \Delta t$ ermittelt. Fehler in den Verschiebungen zum Zeitpunkt $t$ führen zu Fehlern bei den inneren Kräften zum Zeitpunkt $t$ und damit zu Fehlern bei den Verschiebungen zum Zeitpunkt $t + \Delta t$. Das System ist nur unter der Bedingung stabil, dass die Fehler zu einem Zeitpunkt nicht zu größeren Fehlern im Folgezeitpunkt führen.

Damit ein System numerisch stabil bleibt, muss der verwendete Integrationszeitschritt kleiner als der kritische Zeitschritt $\Delta t_c$ sein. $\Delta t_c$ hängt vom verwendeten Integrationsalgorithmus ab und ist nach Wu und Tendick [2004] proportional zur Elementgröße und indirekt proportional zur

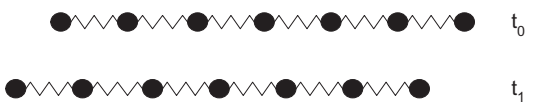


**Abbildung 2.2:** Bei einer impliziten quasi-statischen Integration breitet sich die Störung instantan über den gesamten simulierten Körper aus. Der Gleichgewichtszustand ist bereits nach einem Zeitschritt erreicht.

Wurzel der Materialsteifigkeit.

Ein weiteres Problem der expliziten Integration ist die Ausbreitung von lokalen Störungen durch den simulierten Körper. Wird beispielsweise die Position eines einzelnen Interaktionsknoten verändert, ändern sich die inneren Kräfte im nächsten Zeitschritt nur für die unmittelbaren Nachbarknoten. Ihre Position wird durch den Integrationsschritt verändert. Knoten, die nicht direkt mit dem Interaktionsknoten verbunden sind, werden von der Veränderung nicht betroffen. Ihre Position bleibt konstant. Erst im darauf folgenden Zeitschritt wirkt die Veränderung der Positionen der Nachbarknoten auf deren Nachbarn und so weiter. Eine Störung bereitet sich folglich in jedem Zeitschritt um eine Knotenentfernung aus. Im realen Körper geschieht die Störungsausbreitung mit der Schallgeschwindigkeit des Materials. Abbildung 2.1 veranschaulicht die Ausbreitung einer lokalen Störung bei einer expliziten Integration.

**Algorithmus 2.1** Explizite Integration

```
1: for all n_i ∈ N do
2:     CalculateForces(n_i)
3: for all n_i ∈ N do
4:     DoIntegration(n_i)
```

**Implizite Integration** Bei der *Impliziten Integrationen* wie sie Baraff und Witkin [1998] im Bereich der Animation eingeführt haben, werden die Verschiebungen zu einem Zeitpunkt $t+\Delta t$ unter Berücksichtigung der Kräfte zum Zeitpunkt $t+\Delta t$ berechnet. Das Verfahren ist numerisch unbedingt stabil.

Lokale Störungen wirken sich unverzüglich auf alle Knoten des Gitters aus, da von den Kräften zum Zeitpunkt $t+\Delta t$ ausgegangen wird. Implizite Integrationen können auch ohne Berücksichtigung der Masseneffekte durchgeführt werden. Bei diesem quasi-statischen Fall wird der Gleichgewichtszustand bereits nach einem Zeitschritt erreicht. Eine Vernachlässigung der Masseneffekte ist bei einer expliziten Integration nicht möglich. Nachteil bei der impliziten Integration ist der meist hohe Rechenaufwand.

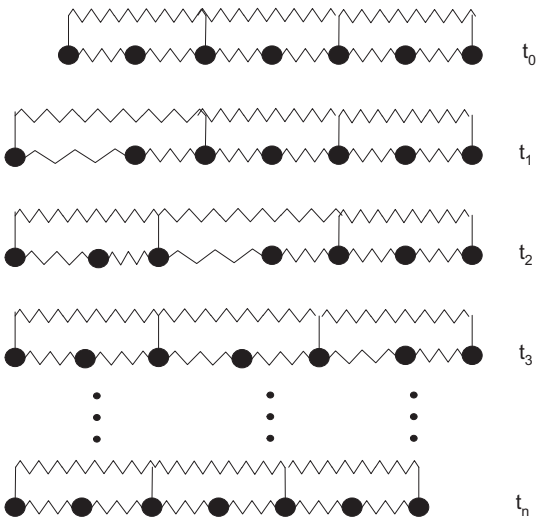


**Abbildung 2.3:** Bei einem Mehrgitterverfahren sorgt ein zusätzliches grobes Gitter für eine bessere Konvergenzgeschwindigkeit und erhöhte die numerische Stabilität.

**Mehrgitterverfahren** Um die Ausbreitung lokaler Störungen bei expliziter Integration zu beschleunigen, werden so genannte Mehrgitterverfahren eingesetzt. Dabei wird das simulierte Objekt durch mehrere Gitter verschiedener Auflösung beschrieben. Auf dem groben Gitter breiten sich Störungen schnell aus. Das feine Gitter ist besser geeignet, um kleine Strukturen zu beschreiben. Die Kopplung beider Gitter ermöglicht schnelle Ausbreitung lokaler Störungen und detaillierte Abbildung kleiner Strukturen. Wu und Tendick [2004] beispielsweise benutzen vier Gitter verschiedener Auflösungen, um die Stabilität und Konvergenzgeschwindigkeit ihrer Gewebesimulation zu beschleunigen.

**Integration von einem Interaktionspunkt ausgehend** Brown, Sorkin, Bruyns, Latombe und Stephanides [2001a] beschleunigen die Konvergenz der expliziten Integration bei großen lokalen Störungen, wie sie durch Interaktionen hervorgerufen werden. Dazu integrieren Brown et. al. knotenweise ausgehend von dem Interaktionspunkt. Bei einer normalen expliziten Integration werden die Kräfte für alle Knotenpunkte zu einem Zeitpunkt $t$ bestimmt. Danach erfolgt der Integrationsschritt bei dem für alle Knoten die Position zum Zeitpunkt $t + \Delta t$ bestimmt wird (siehe Algorithmus 2.1).

Brown et. al. berechnen die Kräfte und führen die Integration zunächst für den Interaktionsknoten aus. Danach wird die Kraftberechnung und Integration für die Nachbarknoten ausgeführt. Anschließend kommen die Nachbarn der Nachbarn an die Reihe usw. Die Integration ist abgeschlossen, wenn al-

le Knoten abgearbeitet wurden. Algorithmus 2.2 zeigt die Implementierung einer Integration, die von einem Interaktionspunkt ausgeht.

**Algorithmus 2.2** Integration von Interaktionspunkt($n_{interaction}$)

```
1: SortNodeListFrom(n_interaction)
2: for all n_i ∈ N do
3:     CalculateForces(n_i)
4:     DoIntegration(n_i)
```

Der Algorithmus beschleunigt die Ausbreitung lokaler Störungen, da bei der Berechnung der Knotenkräfte im Gegensatz zur normalen expliziten Integration die Verschiebungen aller Koten, die näher am Interaktionsknoten liegen bereits berücksichtigt werden. Dadurch kann sich eine Störung bereits in einem Integrationsschritt auf alle Knoten des Gitters auswirken. Bei der normalen expliziten Integration wirkt die Störung nur auf direkt benachbarte Knoten (siehe oben). Nachteil des Verfahrens ist der erhöhte Rechenaufwand für die Sortierung der Knotenliste.

## 2.2 Stand der Technik

Eine ganze Reihe zum Teil höchst unterschiedlicher Modelle wird eingesetzt, um Deformationen von Weichgewebe in medizinischen VR-Simulatoren zu berechnen.

### 2.2.1 Finite-Elemente-Methode

Die *Finite-Elemente-Methode* (FEM) ist eine Methode zur numerischen Lösung eines partiellen Differenzialgleichungssystems, die auf dem Variationsprinzip beruht.

Bei der Simulation von Weichgewebe mit der Methode der Finiten-Elemente geht man von einer physikalischen Beschreibung des modellierten Objektes aus. Einen Einblick in die Elastizitätstheorie vermittelt Anhang A. Die Lösung erfolgt durch Aufteilung des simulierten Gebietes in geometrisch einfache Finite-Elemente. Innerhalb dieser wird die Lösung durch eine kontinuierliche *Ansatzfunktion* angenähert, die in Abhängigkeit von den so genannten *Knotenvariablen* dargestellt wird. Im einfachsten Fall sind die Knotenvariablen die Koordinaten der Knotenpunkte. Die Ansätze für die Lösungen innerhalb der Finiten-Elemente werden anschließend zu einem Gesamtgleichungssystem zusammengesetzt das gelöst werden kann. Für eine ausführliche Beschreibung der Methode der Finiten-Elemente wird auf Bathe [1990] verwiesen.

Durch die Diskretisierung des Gebietes in Finite-Elemente wird die Bewegungsgleichung eines deformierbaren Körpers A.1 zu

$$M\ddot{U} \quad + \quad D\dot{U} \quad + F_{int}(U) \quad = \quad F_{ext} \tag{2.1}$$

Dabei ist $U$ der Vektor der Änderung der Knotenvariablen. $M$ und $D$ sind die *Massen-* und die *Dämpfungsmatrix*. $F_{int}(U)$ und $F_{ext}$ sind die Vektoren der internen und externen Kräfte, die auf die Knotenvariablen wirken.

Vernachlässigt man die Masseneffekte, kommt man zum *quasi-statischen* Fall.

$$F_{int}(U) \quad = \quad F_{ext} \tag{2.2}$$

Geht man näherungsweise von der Annahme einer linearen Elastizität aus, so lassen sich die inneren Kräfte in den Gleichungen 2.1 und 2.2 durch Multiplikation der Verschiebungen mit einer *Steifigkeitsmatrix* $K$ berechnen.

$$F_{int}(U) = \quad KU \tag{2.3}$$

Wie in Abschnitt A.3 näher beschrieben wird, hat der lineare Ansatz zwei Vereinfachungen zur Folge. Die geometrische Linearität bricht mit der Rotationsinvarianz des Ansatzes. Rotationen von Finiten-Elementen führen zu einer unrealistischen Vergrößerung des Volumens. Maurel, Wu, Magnenat-Thalmann und Thalmann [1998] nehmen an, dass ab einer Auslenkung von über 10% die Näherung einer geometrischen Linearität für chirurgische Simulatoren nicht mehr ausreichend ist.

Die zweite Vereinfachung ist die Annahme eines Hooke'schen Materials mit einer linearen Beziehung zwischen Verzerrung und Spannung. Biologisches Gewebe ist in der Regel kein Hooke'sches Material.

Die Bestimmung der Lösung von 2.1 oder 2.2 erfolgt durch ein Integrationsverfahren. In Abschnitt 2.1.3 wurden bereits spezielle Aspekte der Integrationsverfahren diskutiert. Anhang B stellt einige Integrationsverfahren vor.

Unbedingt stabile implizite Integrationsverfahren setzen in der Regel eine Invertierung einer Matrix voraus, deren Dimension $N$ von der Anzahl der Knotenvariablen entspricht. Matrixinvertierungen sind aufwendig und benötigen in der Regel eine Laufzeit von $O(N^3)$. Nach der Invertierung können die Verschiebungen durch Multiplikation einer Matrix mit einem Vektor in $O(N^2)$ berechnet werden. Bei einer dynamischen Simulation muss die Invertierung nur zu Beginn und nach Änderungen an der Gitterstruktur vorgenommen werden. Bei den anderen Zeitschritten fällt nur die Multiplikation an.

Die inneren Kräfte sind in den meisten Modellen kurzreichweitig und wirken nur zwischen Nachbarknoten. Das hat zur Folge, dass die Matrizen sehr dünn besetzt sind. Berücksichtigt man das bei der Speicherung und den Berechnungen, kann ein Großteil des Speicherbedarfs und der Rechenzeit eingespart werden.

Somit lässt sich ein Materialverhalten prinzipiell beliebig genau modellieren. Die hohe Komplexität führt aber zu einem hohen Rechenaufwand. Bei numerisch stabiler impliziter Integration kostet die Berücksichtigung von topologischen Änderungen viel Rechenzeit. Das Problem beim Ansetzen einer linearen Elastizität ist die Rotationsvarianz der Elemente.

Um das Problem der geometrischen Linearität bei Rotationen eines Körpers zu umgehen, schlagen Terzopoulos und Waters [1990] einen hybriden Simulationsansatz vor. Eine *Starre-Körper* (engl. *Rigid-Body*) Simulation modelliert Translationen und Rotationen eines Körpers. Die Finite-Elemente-Methode berechnet die Deformationen des Objekts.

Bro-Nielsen und Cotin [1996] verwenden eine als Kondensation bezeichnete Technik, um die lineare Finite-Elemente-Methode mit impliziter Integration zu beschleunigen. Die Kondensation geht von der Annahme aus, dass nur die Verschiebungen der Oberflächenknoten für die Ausgabe benötigt werden und dass keine externen Kräfte auf innere Knoten wirken. Unter dieser Voraussetzung lässt sich Gleichung 2.3 so umformen, dass nur noch die Knotenvariablen der Oberfläche berechnet werden. Das Verhalten der Oberflächenknoten wird durch die Umformung nicht verändert. Die Dimension des zu lösenden Systems reduziert sich auf die Anzahl der Knotenvariablen der äußeren Knoten. Allerdings verliert die Steifigkeitsmatrix ihre geringe Dichte, was zu einer Verlängerung der Laufzeit für die Matrixinvertierung bei topologischen Änderungen führt.

Delingette, Cotin und Ayache [1999] entwickeln eine hybride Simulation, um Operationen mit topologischen Änderungen in Echtzeit zu simulieren. Eine quasi-statische FEM-Simulation berechnet mögliche globale Deformationen in einem Vorverarbeitungsschritt. Zur Laufzeit werden globale Verformungen aus den vorberechneten Daten interpoliert. Lokale Deformationen im Interaktionsgebiet und topologische Änderungen werden von ihrem Tensor-Mass-Ansatz modelliert, der in Abschnitt 2.2.6 beschrieben wird. Die Kopplung beider Modelle ermöglicht die lokale Interaktion mit einer sich global verformenden virtuellen Leber.

Zhuang und Canny [1999] benutzen einen nicht-linearen FEM-Ansatz, der rotationsinvariant ist. Um knotenweise explizit integrieren zu können, entkoppeln sie Gleichungssystem 2.1 durch so genanntes *Mass-Lumping*. Beim Mass-Lumping wird die Massenmatrix $M$ durch eine Diagonalmatrix $\tilde{M}$

angenähert, die sich wie folgt berechnet.

$$\tilde{M}_{i,j} = \delta_{ij} \sum_{k=1}^{N} M_{ik} \tag{2.4}$$

Zhuang und Canny gehen im Folgenden von *Rayleigh-Dämpfung* aus.

$$D = \alpha M + \beta K \tag{2.5}$$

Sie wählen den Parameter in Gleichung 2.5 $\beta = 0$, um eine diagonale Dämpfungsmatrix zu erhalten. Damit ist ihr Gleichungssystem nach der Berechnung der Kräfte entkoppelt und es kann mit $O$(Anzahl der Knoten) knotenweise explizit integriert werden. Um die nichtlineare Kraftberechnung zu beschleunigen, reduzieren Zhuang und Canny die Anzahl der Knotenpunkte und finiten Elemente, indem sie das Gitter im Inneren gröber auflösen als in den Randbereichen. Zur Integration verwenden sie eine explizite Version der *Newmark* Methode (siehe Abschnitt B.2.1). In Zhuang und Canny [2000] wird der Ansatz auch für eine haptische Ausgabe benutzt.

Ähnlich wie Terzopoulos und Waters [1990] verwenden Müller, McMillan, Dorsey und Jagnow [2001] eine Starre-Körper-Simulation, um Translationen und Rotationen eines simulierten Objekts zu modellieren. Treten Kollisionen auf, wird ein quasi-statischer Finite-Element-Ansatz verwendet, um Deformationen und Brüche des Objekts zu modellieren.

Müller, Dorsey, McMillan, Jagnow und Cutler [2002] verwenden eine lineare Finite-Elemente-Simulation und erweitern diese, um Rotationsinvarianz zu erreichen. In jedem Zeitschritt berechnen sie ein Tensorfeld, das die Rotation aller Knotenpunkte beschreibt. Sie benutzen das Tensorfeld um die Elemente in ihre Ausgangslage, zu rotieren. Die Deformationsberechnung läuft dann im nicht rotierten Referenzsystem ab. Müller et. al. nennen ihre Technik *Stiffness Warping*.

Wu und Tendick [2004] verwenden einen nicht linearen Finiten-Elemente-Ansatz um interaktiv Weichgewebe zu simulieren. Wie Zhuang und Canny [1999] benutzen sie Mass-Lumping, Rayleigh-Dämpfung und einen expliziten Integrationsansatz. Neu an ihrer Idee ist Verwendung eines Mehrgitteransatzes (siehe Abschnitt 2.1.3), um die langsame Störungsausbreitung und die numerische Stabilität des expliziten Ansatzes zu verbessern.

Vigneron, Verly und Warfield [2004] schlagen die *Extended Finite Element Method* (*XFEM*) von Moes, Dolbow und Belytschko [1999] zur Modellierung von topologischen Änderungen während einer FEM-Simulation vor. Die Ansatzfunktion der XFEM ist nicht notwendigerweise kontinuierlich, sondern

erlaubt Diskontinuitäten. Dadurch lassen sich topologischen Veränderungen modellieren, ohne das Simulationsgitter anzupassen. Allerdings erhöht sich die Anzahl der Knotenvariablen an den Elementen, die geschnitten werden, was einen größeren Rechenaufwand zur Folge hat. Vigneron et. al. [2004] verwenden XFEM für eine zweidimensionale Proof-of-Concept-Simulation des Gehirns.

Die Methode der Finiten-Elemente eignet sich besonders für genaue offline Simulationen. Echtzeitsimulationen sind wegen des hohen Rechenaufwands auf grobe Gitter beschränkt. Topologische Änderungen benötigen zusätzliche Rechenzeit oder setzten einen numerisch nur bedingt stabilen, expliziten Integrationsalgorithmus voraus. Besonders bei der Modellierung von steifem Gewebe oder lokalen Rotationen (nichtlinearer Ansatz) können bei expliziter Integration Stabilitätsprobleme auftreten.

### 2.2.2 Andere gitterbasierte Verfahren

Im Folgenden soll noch kurz auf einige andere Ansätze zum Lösen des Differenzialgleichungssystems A.1 eingegangen werden. Die mit der Methode der Finiten-Elemente verwandten Ansätze spielen bei der Simulation von Weichgewebe allerdings eine eher untergeordnete Rolle.

Bei der *Finiten-Differenzen-Methode* (*FDM*) werden die Differentialoperatoren durch Differenzen angenähert. Die Lösung wird für diskrete Werte ermittelt. Es gibt keine kontinuierliche Ansatzfunktion wie bei der Finiten-Elemente-Methode. Terzopoulos, Platt, Barr und Fleischer [1987] benutzen einen Finite-Differenzen-Ansatz zur Simulation einfacher Körper. Probleme bereitet die FDM bei unregelmäßigen Gittern, wie sie oft eingesetzt werden, um biologisches Gewebe zu beschreiben.

Teran, Blemker, Hing und Fedkiw [2003] verwenden die *Finite-Volumen-Methode* (*FVM*) zur Modellierung von Muskelkontraktionen. Die Finite-Volumen-Methode beruht auf einer geometrischen Beschreibung. Die Variablen werden nicht an den Knoten des Meshs gespeichert, sondern innerhalb der Elemente. Teran et. al. bezeichnen ihren FVM-Ansatz als intuitiver als die Methode der Finiten-Elemente. Sie verweisen darauf, dass er weniger rechenintensiv ist als der FEM-Ansatz von Müller et. al. [2001].

Bei der *Randelementmethode BEM* (von engl. *Boundary-Element-Method*) reduziert sich die Dimension des zu lösenden Gleichungssystems, da nur die Oberfläche des simulierten Körpers parametrisiert wird. Die entstehende Matrix ist allerdings dicht besetzt, sodass sich der Geschwindigkeitsvorteil relativiert. Prinzipbedingt können mit der Randelementmethode keine Inhomogenitäten und Anisotropien innerhalb des simulierten Körpers modelliert werden. James und Pai [1999] zeigen, dass sich die BEM für Echtzeitdeformationen einsetzen lässt.

### 2.2.3 Gitterlose Verfahren

Neu im Bereich der Gewebesimulation sind die *Gitterlosen Verfahren*. Müller, Teschner und Gross [2004] verwenden einen kontinuumsmechanischen Ansatz. Anders als bei der Methode der Finiten-Elemente werden die physikalischen Eigenschaften nicht durch die Verbindungen modelliert, sondern durch die Knoten selbst. Müller et. al. benutzen in diesem Zusammenhang den Term *Phyxel* für *physical element*. Die Ansatzfunktionen werden für die Umgebung eines Phyxel angesetzt. Phyxel wirken aufeinander, wenn sich ihre Ansatzfunktionen überlappen. Müller et. al. demonstrieren die Einsetzbarkeit ihrer Methode zur Modellierung von elastischen und plastischen Deformationen und von schmelzenden Objekten.

### 2.2.4 Feder-Masse-Simulationen

*Feder-Masse* oder *Mass-Spring*-Modelle sind eine spezielle Art der *Partikelsimulationen*, die für die Simulation von deformierbarem Gewebe geeignet ist. Die Masse des simulierten Körpers ist in einzelnen Punkten ohne Ausdehnung konzentriert. Bei Partikelsystemen wirken die einzelnen Punkte mit Kräften wie der Gravitation oder elektrostatischer Anziehungen aufeinander. Im Fall von Mass-Spring werden Federkräfte modelliert. Die Massepunkte sind durch virtuelle Federn miteinander verbunden. Entspricht der Abstand zweier, durch eine Feder verbundene Massepunkte nicht der Nulllänge der Feder, wirkt die Feder mit einer Kraft auf ihre Endpunkte so ein, dass deren Abstand sich wieder an der Nulllänge der Feder nähert.

Die Kraft einer Partikelsimulation auf einen Massepunkt $\vec{F}_{i(t)}(\vec{x}, \vec{v})$ hat die Form

$$\vec{F}_i(t)(\vec{x}, \dot{\vec{x}}) \quad = S_i(t)(\vec{x}) \quad R_i(t)(\dot{\vec{x}}) + E_i(t) \tag{2.6}$$

wobei $R_i(t)(\dot{\vec{x}})$ die Reibungskraft, $S_i(t)(\vec{x})$ die Summe der Federkräfte auf Knoten $i$ und $E_i(t)$ die externe Kraft auf den Knoten ist.

Um von den Kräften auf die Veränderungen der Knotenpositionen zu kommen, wird meist eine explizite Integration verwendet. Feder-Masse-Simulationen mit expliziter Integration sind schnell. Der Rechenaufwand der Kraftberechnung steigt linear mit der Anzahl der Federn, der der expliziten Integration linear mit der Anzahl der Knoten. Topologische Änderungen benötigen nur die zur Veränderung des Simulationsgitters nötige Rechenzeit. Durch die explizite Integration sind Feder-Masse-Modelle allerdings nur bedingt stabil und die Konvergenzgeschwindigkeit ist begrenzt.

Die Federkonstanten eines Feder-Masse-Modells sind keine Materialeigenschaften wie beispielsweise das Elastizitätsmodul bei einem kontinuumsmechanischen Ansatz. Eine allgemein gültige Abbildung eines gegebenen Gewebeverhaltens auf eine Feder-Masse-Modell ist ein offenes Problem, auch

wenn bereits einige Methoden vorgeschlagen wurden: u. a. heuristische mathematische Formulierungen (Gelder [1998], Maciel, Boulic und Thalmann [2003]), Optimierungen mit *Neuronalen Netzen* (Nürnberger, Radetzky und Kruse [1998]) und *genetischer Programmierung* (Louchet, Provot und Crochemore [1995], Bianchi, Solenthaler, Szekely und Harders [2004]) . Trotzdem werden Feder-Masse-Modelle wie bei Mollemans, Schutyser, Cleynenbreugel und Suetens [2004] zur Operationsplanung vorgeschlagen.

Zur Erhöhung der Geschwindigkeit implementiert Mosegaard [2005] eine Mass-Spring Simulation, die auf einem Grafikprozessor (GPU) läuft. Durch das parallele Abarbeiten der einzelnen Federn und Knoten lässt sich die Geschwindigkeit damit um eine Größenordnung erhöhen.

### 2.2.5 Längenkorrektur

Provot [1995] schlägt als Erweiterung der Feder-Masse-Simulationen einer *Längenkorrektur* der Federn vor, um den Effekt der *Superelastizität* zu beseitigen. Unter Superelastizität versteht man, dass das Hooke'sche Gesetz auch für große Ausdehnungen seine Gültigkeit behält, was bei den meisten realen Materialien nicht der Fall ist. Eine mit der Dehnung zunehmende Federkonstante führt zu einem realistischeren Verhalten. Bei expliziten Integrationsverfahren bereiten steife Gewebeparameter allerdings numerische Probleme (siehe Abschnitt 2.1.3). Provot modelliert das Verhalten stark gedehnter Federn mit einem stabilen verschiebungsorientierten Ansatz.

Die Federn eines Simulationsgitters haben eine maximale Länge. In jedem Schritt der Längenkorrektur wird die Länge aller Federn kontrolliert. Überschreitet eine Feder ihre maximale Länge, wird sie unter Beibehaltung ihrer Richtung auf ihre maximale Länge gekürzt. Dabei werden drei Fälle unterschieden.

- Ist genau einer der Endpunkte der Federn fixiert, wird der nicht fixierte Endpunkt in Richtung des fixierten verschoben, bis die Feder auf ihre maximale Länge verkürzt ist.
- Sind beide Endpunkte nicht fixiert, werden beide Punkte um den gleichen Betrag aufeinander zu verschoben, bis die maximale Federlänge erreicht ist.
- Sind beide Endpunkte fixiert, findet keine Korrektur der Federlänge statt.

Der Ansatz von Provot kann auch ohne Feder-Masse-Simulation als rein verschiebungsorientiertes Modell verwendet werden. Kombiniert mit einer

Feder-Masse-Simulation nimmt die Längenkorrektur Energie aus dem Feder-Masse-System und erhöht so die numerische Stabilität.

Die Verschiebung eines Knotenpunktes verändert die Federlänge aller verbundenen Federn. Aus diesem Grund werden bereits korrigierte Federn bei der Korrektur angrenzender Federn häufig wieder über ihre maximale Länge ausgedehnt. In der Regel sind deshalb mehrere Durchgänge nötig, bis die Längenkorrektur zu einem statischen Zustand führt.

Wagner [2003] schlägt eine sortierte Längenkorrektur vor, um große lokale Störungen innerhalb eines Mass-Spring-Gitters schneller zu verteilen. Ähnlich wie die Knoten bei Brown et. al. [2001a] (siehe Abschnitt 2.1.3) werden die Federn nach ihrem Abstand von Interaktionspunkt abgearbeitet. Um die unerwünschte Verlängerung bereits korrigierter Federn zu verhindern, werden die Endpunkte bereits korrigierter Federn für den Rest der Korrektur fixiert.

Mosegaard [2004] kombiniert die Längenkorrektur von Provot mit dem Integrationsansatz von Brown et. al. [2001a]. Mosegaard nennt sein Modell *LR-Mass-Spring* für lokale Interaktion und Relaxation.

Feder-Masse Simulationen sind schnell und eignen sich um topologische Änderungen in Echtzeit zu modellieren. Steife Gewebeparameter führen zu hohen Kräften und numerischen Instabilitäten. Wie ein Vergleich der Algorithmen von Provot, Wagner und Mosegaard in Abschnitt 2.3.5 zeigt, erhöhen die verschiedenen Ansätze zur Längenkorrektur die mögliche Steifigkeit das Gewebeverhalten nur leicht.

### 2.2.6 Tensor-Mass-Modell

Das von Cotin, Delingette und Ayache [2000] vorgeschlagene *Tensor-Mass*-Modell kann man zwischen Feder-Masse und Finiter-Elemente-Simulation einordnen.

Wie bei einem Mass-Spring-Ansatz geht man davon aus, dass die Masse des simulierten Objektes in diskreten Punkten konzentriert ist.

Die Berechnung der elastischen Kräfte geschieht durch einen kontinuumsmechanischen Ansatz. Zur Simulation eines Objektes wird dessen Gebiet in Finite-Elemente aufgeteilt.

Die grundlegende Idee des Tensor-Mass-Modells ist die Aufteilung der Kräfte auf einen Knoten in zwei Teile: Die Kräfte, die durch die Verschiebung des Knoten selbst entstehen und die Kräfte, die durch die Verschiebungen der Nachbarn hervorgerufen werden. Die zugehörigen Einträge der Steifigkeitsmatrix werden an den Knoten selbst bzw. an Verbindungen zu den Nachbarknoten gespeichert. Die Speicherung der Einträge innerhalb des Gitters

ermöglicht, topologische Änderungen ohne großen Rechaufwand zu berücksichtigen.

In jedem Zeitschritt werden die inneren Knoten-Kräfte aus den Verschiebungen der Knoten und den Abständen benachbarter Knoten berechnet. Durch die diskrete Verteilung der Massen ist eine punktweise explizite Integration möglich.

Cotin et. al. gehen von der linearen Elastizitätstheorie (siehe Abschnitt A.3) aus, um ein Tetraederelement für ihren Ansatz zu entwickeln.

Picinbono, Lombardo, Delingette und Ayache [2000] entwickeln ein Tetraederelement mit der Fähigkeit *transversal isotrope* Materialien (siehe Abschnitt A.3) zu modellieren. Picinbono, Delingette und Ayache [2001] erweitern das Element anschließend auf den nichtlinearen *St. Vennant Kirchhoff* Elastizitätsansatz (siehe Abschnitt A.1).

Für eine umfangreiche Beschreibung des Tensor-Mass-Modells sei auf Delingette und Ayache [2004] verwiesen.

Gegenüber der Methode der Finiten-Elemente hat das Tensor-Mass-Modell den Vorteil, dass topologische Änderungen in Echtzeit berücksichtigt werden können. Hauptnachteil des Ansatzes ist die nur bedingte numerische Stabilität der expliziten Integration. Dadurch ist es schwierig steifes Gewebe in Echtzeit zu modellieren, vor allem wenn mögliche Rotationen der Elemente einen nichtlinearen Ansatz erforderlich machen.

### 2.2.7 Chain-Mail und Enhanced Chain-Mail

Gibson und Mirtich [1997] schlagen den deskriptiven *Chain-Mail* (engl. für Kettenhemd)-Algorithmus zur Modellierung von plastischen Verformungen von homogenen Volumenobjekten vor.

Jeder Knoten eines simulierten Objekts ist mit seinen Nachbarn in x- y- und z-Richtung durch virtuelle Kettenglieder (Chains) verbunden. Die Kettenglieder bieten einen gewissen Spielraum, in dem sich ein Knoten bewegen kann, ohne seine Nachbarn zu beeinflussen. Sind die Knotenverschiebungen größer, als es der Spielraum eines Kettengliedes erlaubt, wirken die Chains als starre Verbindungen und ziehen bzw. drücken den Nachbarknoten am andern Ende des Gliedes nach.

Der Algorithmus arbeitet ausgehend von einem Interaktionsknoten. Nach einer Verschiebung des Interaktionsknotens werden alle verbundenen Chains überprüft. Über- bzw. unterschreitet die x-Komponente der Länge einer Chain den für ihre x-Richtung festgelegten maximalen bzw. minimalen Wert, wird der Knoten am anderen Ende des Kettengliedes in x-Richtung verschoben, bis die x-Komponente der Länge der Chain sich wieder im erlaubten Bereich befindet. Danach werden die Kettenglieder des korrigierten

Knotens überprüft, bei Bedarf korrigiert und so weiter. Das geht so lange, bis der Algorithmus keine Chain mit ungültiger Länge in x-Richtung mehr findet. Für die y- und z-Koordinate läuft der Algorithmus analog ab. Durch die separaten Berechnungen in x-, y- und z-Richtung ist es möglich, Rotationen mit Chain-Mail zu modellieren.

Schill, Gibson, Bender und Männer [1998] erweiterten Chain-Mail um die Fähigkeit, inhomogene Strukturen zu simulieren. Ihr *ECM* (*Enhanced Chain-Mail*) arbeitet mit einer Liste, in die alle zu korrigierenden Chains eingetragen werden. Die Liste wird so sortiert, dass jeweils die Chain mit der größten Verletzung ihrer Länge als Erstes abgearbeitet wird. Dadurch wird garantiert, dass der Chain-Mail Algorithmus auch bei inhomogenen Geweben terminiert. Eine ausführliche Beschreibung von Chain-Mail und Enhanced Chain-Mail findet sich bei Schill [2001].

Chain-Mail ist numerisch stabil und gut geeignet um topologische Änderungen in Echtzeit, zu berücksichtigen. Hauptnachteil von Chain-Mail ist, dass es nicht möglich ist, Rotationen von Teilen des simulierten Objekts zu modellieren. Außerdem ist Chain-Mail ein Verfahren, das Gewebe rein plastisch modelliert. Es findet keine elastische Relaxation statt.

### 2.2.8 Zusammenfassung

Zusammenfassend lässt sich sagen, dass alle vorgestellten Simulationen Probleme haben steife Membrane mit topologischen Änderungen in Echtzeit zu modellieren.

Ein Problem bei Simulationen mit expliziter Integration (Tensor-Mass, nichtlineare FEM, die meisten Mass-Spring-Ansätze) ist die langsame Konvergenz bei großen lokalen Störungen (siehe Abschnitt 2.1.3). Pro Zeitschritt kann sich eine Störung nur um jeweils eine Knotenentfernung fortpflanzen. Zusätzlich können bei gegebenem Zeitschritt die Gewebeparameter aus Stabilitätsgründen nicht beliebig steif gewählt werden. Zur Modellierung eines instantan reagierenden, steifen Gewebeverhaltens sind deshalb viele Integrationsschritte pro Zeitschritt nötig.

Simulationen mit impliziter Integration, wie sie bei den meisten linearen FEM-Ansätzen verwendet werden, sind zwar numerisch stabil, dafür aber in der Regel langsamer und topologische Änderungen benötigen zusätzliche Rechenzeit.

Das verschiebungsorientierte Chain-Mail ist stabil und geeignet um topologische Änderungen in Echtzeit zu berücksichtigen, kann aber keine lokalen Rotationen modellieren und bietet keine elastische Relaxation.

## 2.3 Dragnet

Der *Dragnet-* (engl. für Schleppnetz) Algorithmus wurde entwickelt, um ein steifes Gewebeverhalten auch bei großen lokalen Störungen in einem Mass-Spring-Gitter zu modellieren. Bei dem Ansatz wurde darauf geachtet, dass topologische Änderungen der Gitterstruktur weiterhin in Echtzeit vorgenommen werden können.

### 2.3.1 Das Konzept von Dragnet

Dragnet wurde als verschiebungsorientierter Ansatz konzipiert um steifes Gewebe numerisch stabil modellieren, zu können. Der Algorithmus arbeitet direkt auf einem Mass-Spring-Gitter wodurch topologische Änderungen der Gitterstruktur automatisch berücksichtigt werden.

Die grundlegende Idee von Dragnet ist das Ziehen an einem Netz aus zusammengeknoteten Schnüren. Das Simulationsgitter besteht aus Knoten, die durch so genannte *Strings* miteinander verknüpft sind. Jeder String verbindet zwei Knoten miteinander. In der Regel hängt ein Knoten an mehreren Strings. Jeder String hat eine maximale Länge, bis zu der er gestreckt werden kann. Nach der Verschiebung eines Knoten wird die Länge aller mit ihm verbunden Strings überprüft. Überschreitet die Länge eines Strings seine Maximallänge, wird der String verkürzt. Dies geschieht, indem der Knoten der nicht bewegt wurde, soweit entlang des Strings verschoben wird, dass sich der String auf seine maximale Länge verkürzt. Bei dem jetzt verschobenen Knoten werden nachfolgend alle anliegenden Strings, die noch nicht korrigiert wurden, auf Verletzungen ihrer Maximallänge überprüft und gegebenenfalls korrigiert. Anschließend werden wieder die Strings der verschobenen Knoten ausgewertet, korrigiert usw. Dragnet terminiert, wenn der Algorithmus keinen String mit Verletzung seiner Maximallänge mehr findet, der noch nicht korrigiert wurde. Dies ist spätestens der Fall, wenn alle Strings einmal korrigiert wurden.

### 2.3.2 Der Dragnet-Algorithmus

Der Algorithmus arbeitet mit einer Liste $L$, in der die zu korrigierenden Strings $s_i$ gespeichert werden. Korrigiert werden müssen Strings, deren Länge (Länge($s_i$)) größer als ihre Maximallänge (maxLänge($s_i$)) ist. Die Strings in der Liste $L$ sind nach ihrer Längenverletzung (Länge($s_i$)-maxLänge($s_i$)) sortiert. Um zu verhindern, dass ein String doppelt abgearbeitet wird, erhält jeder String vor dem Einfügen in die Liste eine Markierung. Nur Strings, die keine aktuelle Markierung haben, werden der Liste hinzugefügt.

**Algorithmus 2.3** Dragnet($N_{ia}$)

1: **for all** $n_i \in N_{ia}$ **do**
2: **for all** $s_i \in S(n_i)$ **do**
3: **if** Länge($s_i$) > maxLänge($s_i$) **then**
4: Markiere $n_i$ als Interaktionsknoten von $s_i$
5: Markiere $s_i$ als bearbeitet.
6: Sortiere $s_i$ in die $L$ ein
7: **while** $L$ ist nicht leer **do**
8: $s_i$ = erster String in $L$
9: Entferne $s_i$ aus der $L$
10: Berechne die neue Position des $n_{ra}(s_i)$
$P_{ra} = P_{ia} + \frac{P_{ra}-P_{ia}}{|P_{ra}-P_{ia}|} \cdot$ maxLänge($s_i$)
11: **for all** $s_j \in S(n_{ra})$ **do**
12: **if** Länge($s_j$) > maxLänge($s_j$) **then**
13: **if** Markierung des $s_j$ ist nicht aktuell **then**
14: Markiere $n_{ra}$ als Interaktionsknoten von $s_j$
15: Markiere $s_j$ als bearbeitet.
16: Sortiere $s_j$ in die $L$ ein

Der Dragnet Algorithmus 2.3 bekommt beim Start eine Liste von Interaktionsknoten $N_{ia}$ übergeben. Zu Beginn eines Dragnet-Durchlaufs wird die Länge aller Strings geprüft, die mit den Interaktionsknoten verbunden sind (Zeile 1ff ). Strings mit einer Längenverletzung werden in die Liste einsortiert (Zeile 6). Dabei wird im String vermerkt, welcher seiner beiden Endknoten der Interaktionsknoten $n_{ia}(s_i)$ ist (Zeile 4). Der andere Knoten wird im Folgenden als *Reaktionsknoten* des Strings $n_{ra}(s_i)$ bezeichnet.

Das Abarbeiten der Liste geschieht folgendermaßen: In den Zeilen 8 und 9 wird der String mit der größten Längenverletzung wird aus der Liste genommen.

Die Länge des Strings wird korrigiert (Zeile 10), indem die Position des Reaktionsknoten $P_{ra}$ entlang der Stringrichtung soweit verschoben wird, dass der String auf seine Maximallänge (maxLänge($s_i$)) verkürzt ist. Der Reaktionsknoten des Strings wird jetzt zu einem Interaktionsknoten: In den Zeilen 12 und Zeilen 13 werden alle Strings, die mit dem neuen Interaktionsknoten verknüpft sind, werden überprüft. Für Strings mit einer Längenverletzung, die keine aktuelle Markierung aufweisen, wird der Interaktionsknoten $n_{ia}(s_i)$ vermerkt (Zeile 14) und der String wird in die Stringliste $L$ einsortiert (Zeile 16).

Die Abarbeitung der Liste wird wieder mit den nächsten String fortgesetzt (Zeile 8), der die größte Längenverletzung hat. Der Algorithmus terminiert, wenn die Liste vollständig geleert ist. Dies ist spätestens nach der einmaligen Korrektur aller Strings der Fall.

### 2.3.3 Modifikation von Dragnet

Der in den vorigen Abschnitten vorgestellte Dragnet-Algorithmus bewirkt die Fortpflanzung einer lokalen Störung innerhalb eines Zeitschrittes durch das gesamte Gitter. Um die Ausbreitungsgeschwindigkeit einer Störung zu kontrollieren, wird ein zusätzlicher Faktor $f_{dragnet}$ eingefügt, der die Längenkorrektur abschwächt.

Die Längenkorrektur in Algorithmus 2.3 sieht dann folgendermaßen aus:

$$P_{ra} = P_{ia} + f_{dragnet} \frac{P_{ra} - P_{ia}}{|P_{ra} - P_{ia}|} \cdot \text{maxLänge}(s_i) \tag{2.7}$$

Dragnet liefert vernünftige Ergebnisse für $f_{dragnet} \in ]0; 1]$. Ist der Faktur $f_{dragnet} = 0$ findet keine Deformation statt. Für Werte kleiner 0 oder größer als 1 wird der Algorithmus instabil.

### 2.3.4 Ergebnisse

Der Dragnet-Algorithmus wurde wie oben beschrieben implementiert. Ein Beispiel für ein mit Dragnet simuliertes Objekt zeigt Abbildung 2.4.

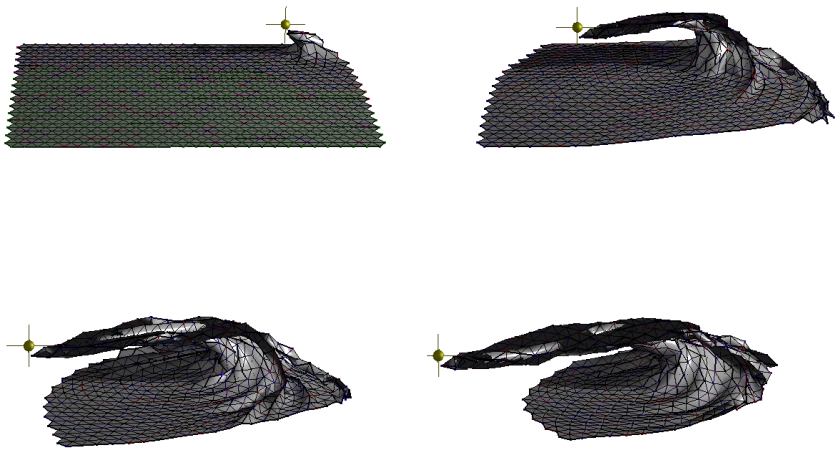

**Abbildung 2.4:** Beispiel eines mit Dragnet simulierten Gitters. Ein Randknoten wird gegriffen und von links hinten nach rechts vorne über das Gitter gezogen.

Nachfolgend wird der Dragnet Algorithmus experimentell untersucht. Alle Messungen beziehen sich auf ein flächiges Gitter in einem dreidimensionalen Raum.

Die Interaktion mit dem Simulationsgitter wird modelliert, indem der Knoten des Gitters, der dem Gitterschwerpunkt am nächsten ist, um einen bestimmten Vektor verschoben wird. Anschließend wird die Reaktion des Gitters auf die lokale Störung beobachtet.

Alle Messungen wurden auf einem Pentium 4 mit HyperThreading System mit 2.8 GHz unter Windows XP durchgeführt. Die Anwendung lief hierzu mit Echtzeit-Priorität an einen virtuellen Prozessor gebunden. Alle anderen Programme einschließlich der Benutzeroberfläche wurden geschlossen und alle unnötigen Dienste angehalten, um Störung der Messungen zu minimieren.

**Konvergenzgeschwindigkeit** Zunächst wird die Konvergenzgeschwindigkeit von Dragnet in Abhängigkeit vom Korrekturfaktor $f_{dragnet}$ in Gleichung 2.7 untersucht. Hierzu wird die Anzahl der Zeitschritte gemessen, die Dragnet benötigt, damit ein Simulationsgitter aus 630 Knoten und 1799 Strings nach der Auslenkung eines Knoten wieder einen statischen Zustand erreicht. Das Gitter wird als statisch definiert, wenn in einem Zeitschritt die größte Knotenverschiebung vier Größenordnungen unter der Gittergröße liegt. Als Referenz wird die Konvergenzgeschwindigkeit der Längenkorrektur aus Abschnitt 2.2.5 untersucht. Hierzu wird bei der Längenkorrektur ein analoger Korrekturfaktor eingeführt. Das verwendete Gitter besteht aus 630 Knoten, die mit 1799 Strings bzw. Federn und 1170 Dreiecken verbunden sind. Die Ergebnisse der Messungen sind in Tabelle D.1 wiedergegeben und in Diagramm 2.5 veranschaulicht.

Wie das Diagramm zeigt, konvergiert Dragnet deutlich schneller als die Längenkorrektur. Bereits ab einem Korrekturfaktor von 0.1 erreicht Dragnet eine Konvergenzgeschwindigkeit in der Größenordnung der Längenkorrektur von Provot [1995] (Faktor 1.0). Bei einem Korrekturfaktor von 1.0 benötigt Dragnet unter den Testbedingungen genau einen Zeitschritt um einen stabilen Zustand zu erreichen. Für Korrekturfaktoren über 1.0 ist der Dragnet-Algorithmus nicht stabil. Bei der Längenkorrektur lassen sich auch Überkorrekturen mit Korrekturfaktor größer 1.0 realisieren. Die Konvergenzgeschwindigkeit nimmt bei einer Erhöhung der Überkorrektur weiter zu, bis die Längenkorrektur bei einem Faktor von ca. 2.1 instabil wird.

**Abbruchbedingung und Laufzeitverhalten** Der Dragnet-Algorithmus 2.3 terminiert, wenn die Liste mit abzuarbeitenden Strings leer ist. Beim Abarbeiten eines Elements der Liste werden alle mit dem Reaktionsknoten verbunden Strings in die Liste hinzugefügt wenn 1) sie eine Längenverletzung aufweisen und 2) sie in diesem Zeitschritt noch nicht der Liste hinzugefügt wurden. Aus Bedingung 1) folgt, dass in einem Gitter, in dem kein

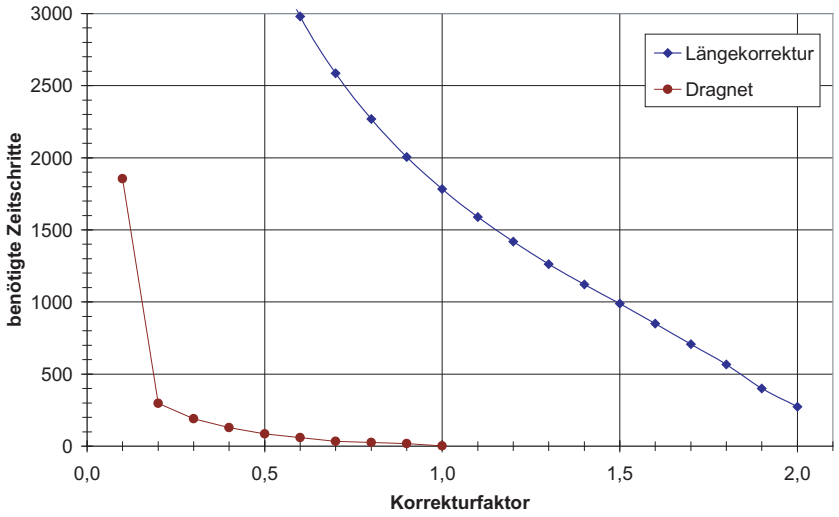


**Abbildung 2.5:** Grafische Auftragung der Daten aus Tabelle D.1. Bei Dragnet nimmt die Anzahl der benötigten Zeitschritte mit steigendem Korrekturfaktor anfangs stark ab. Später ist eine leichte Abnahme auf niedrigem Niveau zu verzeichnen. Die Längenkorrektur benötigt bei allen Korrekturfaktoren deutlich mehr Zeitschritte als Dragnet. Die zum Diagramm gehörenden Messwerte sind in Tabelle D.1 im Anhang abgedruckt.

String eine Längenverletzung aufweist, der Algorithmus sofort wieder terminiert. Er hat für den *best case* eine Laufzeit von $\Omega(1)$. Aus Bedingung 2) folgt, dass in einem Gitter, in dem jeder String eine Längenverletzung aufweist, jeder String pro Durchlauf genau einmal korrigiert wird. Das heißt, die *Worst Case* Laufzeit beträgt $O$(Anzahl der Strings)

Aus Bedingung 2) folgt eine Begrenzung der maximalen Laufzeit. Ohne diese Beschränkung würde der Algorithmus bei ungünstigen Konfigurationen nicht in endlicher Zeit terminieren. Bedingung 2) bewirkt auch, dass das Gitter sich nach einem Durchgang von Dragnet nicht unbedingt in einem Gleichgewicht befindet. Unter einem Gleichgewicht versteht man in diesem Fall, dass kein String eine Längenverletzung aufweist, also ein weiterer Dragnet-Durchgang keine Knotenposition verändert. Die Messwerte die in Diagramm 2.6 veranschaulicht sind, bestätigen die theoretischen Überlegungen zum Laufzeitverhalten des Dragnet Algorithmus. Die Messungen wurden ebenfalls unter den oben beschrieben Bedingungen durchgeführt.

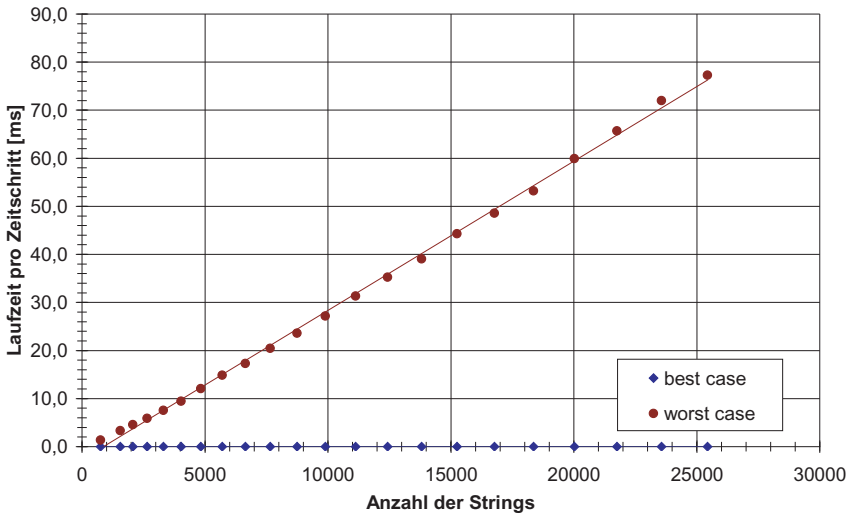


**Abbildung 2.6:** Die Laufzeit von Dragnet pro Zeitschritt. Im best case, wenn keine Strings eine Längenverletzung aufweisen ist die Laufzeit des Algorithmus konstant. Im Worst Case weisen alle Strings eine Längenverletzung auf. Die Laufzeit steigt linear mit der Anzahl der Strings. Die zum Diagramm gehörenden Messwerte sind in Tabelle D.2 im Anhang abgedruckt.

### 2.3.5 Vergleich verschiedener Verfahren und Kombinationen

Neben dem Dragnet-Algorithmus existieren auch noch andere Verfahren, um die Konvergenzgeschwindigkeit und die numerische Stabilität einer expliziten Euler-Integration zu verbessern. Im Folgenden wird Dragnet mit der Längenkorrektur von Provot [1995], der sortierten Längenkorrektur von Wagner [2003](beides Abschnitt 2.2.5) und der interaktionspunkt-basierten Integration von Brown et. al. [2001a] (Anschnitt 2.1.3) verglichen. Außerdem werden Unterschiede und Ähnlichkeiten zwischen Dragnet und Chain-Mail diskutiert.

Zum Vergleich wurde ein Gitter mit unterschiedlichen Algorithmen simuliert. Das Testprogramm arbeitet unter denselben Voraussetzungen wie bei obiger Untersuchung der Konvergenzgeschwindigkeit. Die Position eines Knotens wird verändert und die Veränderungen des Simulationsgitters werden beobachtet. Gemessen wird die Anzahl der Zeitschritte, bis eine starke Verlangsamung der Simulationsgeschwindigkeit zu beobachten ist. Die Zeit bis zum starken Verlangsamen der Simulationsgeschwindigkeit nimmt ein Benutzer bei einer Interaktion als Reaktionszeit des Gewebes wahr. Veränderungen, die anschließend mit niedriger Geschwindigkeit ablaufen, werden als Relaxation des Gewebes wahrgenommen. Steifes Gewebe reagiert sehr schnell und hat keinen merklichen Relaxationseffekt. Die Simulationsgeschwindigkeit wird in dem Versuch als langsam erachtet, wenn die Veränderung der Knotenpositionen drei Größenordnungen unter der Gittergröße liegt. Zusätzlich wird die durchschnittliche und maximale Rechzeit der Simulation pro Zeitschritt gemessen. Alle Berechnungen arbeiten auf einem Gitter mit 1403 Knoten, 4066 Federn, Chains bzw. Strings und 2664 Dreiecken. Bei allen Messungen wurden die freien Parameter so gewählt, dass unter Beibehaltung der numerischen Stabilität ein möglichst steifes Gewebeverhalten erzielt wurde. Die Messwerte sind in Tabelle 2.1 abzulesen. Abbildung 2.7 zeigt die Simulationen, nachdem sich ihre Simulationsgeschwindigkeit stark reduziert hat. Die Falschfarbendarstellung gibt Aufschluss über die Spannungsverteilung im simulierten Gewebe und ermöglicht so Rückschlüsse, wie weit die Simulation von einem Gleichgewichtszustand entfernt ist.

**Mass-Spring mit expliziter Euler-Integration** Betrachtet man Tabelle 2.1, so fällt auf, dass eine Mass-Spring-Simulation mit expliziter Euler-Integration sich nach wenigen Zeitschritten verlangsamt. Von einem Gleichgewicht ist die Simulation zum Zeitpunkt der Verlangsamung noch weit entfernt sind. In Abbildung 2.7 (a) wird ersichtlich, dass sich die Spannungen noch nicht innerhalb des Gitters verteilt haben. Das Gleichgewicht wird in erster Linie durch einen langsamen Relaxationsprozess erreicht, der erst

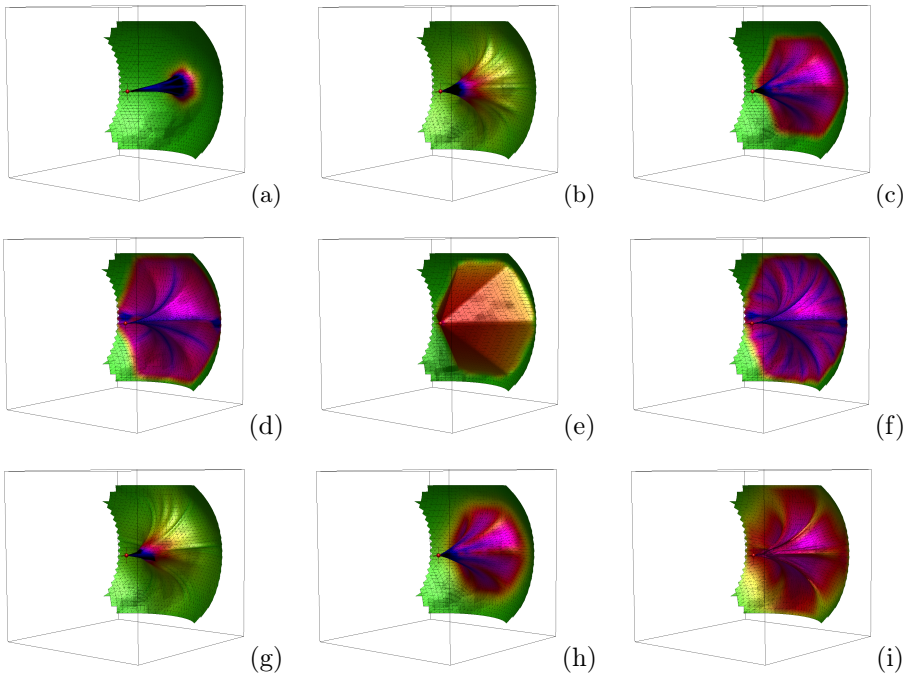


**Abbildung 2.7:** Verschiedene Simulationen, nachdem sich ihre Simulationsgeschwindigkeit stark reduziert hat. Von links oben nach rechts untern: Mass-Spring mit expliziter Euler-Integration (a), Mass-Spring mit interaktionspunkt-basierter Integration (b), Längenkorrektur (c), sortierte Längenkorrektur (d), Chain-Mail (e), Dragnet (f), LR-Mass-Spring Modell (g), Mass-Spring mit Längenkorrektur (h) und Mass-Spring kombiniert mit Dragnet (i). In allen Abbildungen sind die Knotenspannungen in Falschfarben dargestellt. Von Grün über Rot zu Blau nehmen die Knotenspannungen zu.

| Simulation | Anzahl Zeitschritte | Rechenzeit gesamt [ms] | Durchschnitt [ms] | Maximal [ms] |
|---|---|---|---|---|
| Mass-Spring | 86 | 188.5 | 2.2 | 2.3 |
| Interaktions-punkt basierte Integration | 149 | 565.0 | 3.8 | 4.2 |
| Längen-korrektur | 193 | 335.6 | 1.7 | 1.8 |
| sortierte Längen-korrektur | 32 | 139.6 | 4.4 | 4.5 |
| Enhanced Chain-Mail | 1 | 3.5 | 3.5 | 3.5 |
| Dragnet | 1 | 5.0 | 5.0 | 5.0 |
| LR-Mass-Spring | 192 | 1044.6 | 5.4 | 5.5 |
| Längen-korrektur Mass-Spring | 221 | 843.8 | 3.8 | 4.0 |
| Dragnet Mass-Spring | 75 | 184.1 | 2.5 | 9.4 |

**Tabelle 2.1:** Die Tabelle zeigt, wie viel Zeitschritte und Rechenzeit eine Simulation benötigt bis ihre Simulationsgeschwindigkeit sich stark verlangsamt hat.

nach Minuten beendet ist.

**Mass-Spring mit interaktionspunkt-basierter Integration** Bei der Integrationsmethode von Brown et. al. [2001a] verlangsamt sich die Simulation später als bei der normalen expliziten Euler-Integration (siehe Tabelle 2.1). Die lokale Störung hat sich weiter im Simulationsgitter ausgebreitet. Die Spannungsverteilung (Abbildung 2.7 (b)) zeigt, dass die Simulation trotzdem noch weit von einem Gleichgewichtszustand entfernt ist. Im Vergleich zur normalen expliziten Euler-Integration konvergiert die interaktionspunkt-basierte Integration schneller, benötigt allerdings zusätzliche Rechenzeit für das Hangeln durch das Simulationsgitter.

**Vergleich zwischen Dragnet und der Längenkorrektur** Die Längenkorrektur von Provot [1995] sorgt, wie in Abbildung 2.7 (c) ersichtlich für eine bessere Verteilung der lokalen Störung innerhalb des Simulationsgitters als der oben beschriebene Mass-Spring Ansatz. Aus Tabelle 2.1 kann man entnehmen, dass die Endkonfiguration erst nach relativ vielen Zeitschritten erreicht wird. Betrachtet man die Simulationsergebnisse in Abbildung 2.7 (h) fällt auf, dass die Kombination der Längenkorrektur mit einem Mass-Spring-Ansatz und expliziter Integration das Endstadium der Simulation kaum beeinflusst. Bei einer Kombination mit einer Mass-Spring-Simulation und interaktionspunkt-basierten Integration (Abbildung 2.7 (g)) wird das Ergebnis hauptsächlich von der Mass-Spring-Simulation bestimmt. Bei beiden Kombinationen bleibt die Anzahl der benötigten Zeitschritte im Bereich der unkombinierten Längenkorrektur.

Im Vergleich mit Dragnet benötigt die Längenkorrektur mehr Zeitschritte, bis sie ihren Endzustand erreicht (siehe Tabelle 2.1). Bei beiden Algorithmen bleiben Falten mit hoher Spannung, die sich durch das Mesh ziehen (Abbildung 2.7 (c) und (f)). Bei Dragnet bilden sich deutlich mehr und ausgeprägtere Falten heraus. Bei der Kombination der Algorithmen mit einer Mass-Spring-Simulation erhöht sich jeweils die Anzahl der benötigten Zeitschritte. Bei Dragnet wird durch die Kombination die Spannungsverteilung deutlich homogener (vergleiche Abbildung 2.7 (f) und (i)). Bei der Längekorrektur ändert sich die Spannungsverteilung hingegen nur gering (vergleiche Abbildung 2.7 (c) und (h)).

**Vergleich zwischen Dragnet und der sortierten Längenkorrektur** Die Endkonfigurationen in Abbildung 2.7 bei der sortierten Längenkorrektur (d) und Dragnet (f) unterscheiden sich nur gering. Der Faltenwurf ist bei Dragnet ist etwas ausgeprägter. Der Dragnet-Algorithmus erreicht bei der Testsimulation nach dem ersten Schritt bereits einen statischen Zustand,

die sortierte Längenkorrektur benötigt etwas 30 Zeitschritte (siehe Tabelle 2.1).

Algorithmisch besteht der Hauptunterschied beider Ansätze in der Reihenfolge, in der die Feder bzw. Strings korrigiert werden. Während die sortierte Längekorrektur die Federn vom Interaktionspunkt ausgehend abarbeitet, werden die Strings bei Dragnet in Abhängigkeit der Überschreitung ihrer Maximallänge behandelt. Strings mit einer größeren Überschreitung der Maximallänge werden zuerst korrigiert. Dadurch konvergiert der Dragnet Algorithmus deutlich schneller.

**Vergleich zwischen Dragnet und Chain-Mail** Dragnet und der (Enhanced) Chain-Mail Algorithmus arbeiten beide mit Knotenverbindungen. Die verbundenen Knoten können sich in einem gewissen Spielraum bewegen, ohne die Nachbarknoten zu beeinflussen. Bei der Überschreitung des Bewegungsspielraums wirken die Verbindungen als starre Elemente. Der Hauptunterschied zwischen beiden Ansätzen liegt in dem Verhalten der Verbindungselemente. Während die Elemente bei Dragnet eine obere Schranke für den radialen Abstand der verbundenen Knoten garantieren, werden bei Chain-Mail die Abstände in x-, y- und z-Richtung separat beschränkt. Bei Chain-Mail gibt es für jede Richtung eine obere und eine untere Schranke. Die Separation der Richtungen führt dazu, dass mit Chain-Mail keine Rotationen modelliert werden können. Das Ziehen einer mit Chain-Mail modellierten Membran führt deshalb zu einer Pyramidenstruktur (siehe Abbildung 2.7 (e)) Dragnet hat den Nachteil, dass es nicht volumen- oder flächenerhaltend ist. Ein zweiter Unterscheidungspunkt ist die Abbruchbedingung. Der Chain-Mail Algorithmus bricht ab, wenn alle Elemente sich in einem gültigen Zustand befinden. Dies ist bei Beendigung eines Dragnet-Durchgangs nicht garantiert. Dragnet bricht spätestens ab, wenn alle Verbindungen einmal korrigiert wurden. Die Abbruchbedingung beschränkt die Laufzeit des Algorithmus im Worst Case.

## 2.4 Zusammenfassung

In diesem Kapitel wurden Algorithmen vorgestellt, mit denen sich Deformation von Objekten berechnen lassen. Gesucht war ein Ansatz, der Deformationen steifer Membrane unter Berücksichtigung topologische Änderungen in Echtzeit modelliert. In den Tabellen 2.2 und 2.3 sind verschiedene Simulationen mit ihren wichtigsten Eigenschaften zusammengefasst. Die Ansätze unterscheiden sich im Bezug auf das modellierte Gewebeverhalten, Geschwindigkeit und Stabilität.

Die Ansätze aus dem Stand der Technik (2.2) die mit expliziter Integration arbeiten sind numerisch nur bedingt stabil und haben Probleme ein

steifes Gewebeverhalten zu modellieren. Ansätze mit impliziter Integration benötigen zusätzliche Rechenzeit, wenn topologische Änderungen modelliert werden müssen. Der verschiebungsorientierte Chain-Mail-Ansatz hat das Problem, dass keine lokalen Rotationen modelliert werden können.

Im Rahmen dieser Arbeit wurde der Dragnet-Algorithmus entwickelt. In einem Dragnet-Gitter sind jeweils zwei Knoten durch ein Element verbunden. Die Elemente lassen Knotenverschiebungen bis zu einem gewissen maximalen Abstand zu. Bei Überschreitung der maximalen Distanz wird der Knotenabstand korrigiert. Die Korrektur erfolgt gerichtet von einem Interaktionsknoten ausgehen.

Die Stärken von Dragnet sind die Modellierung von großen lokalen Interaktionen mit steifen Membranen oder Materialien wie beispielsweise Stoffen. Dragnet beschleunigt die Konvergenz von Mass-Spring-Simulationen nach großen lokalen Interaktionen erheblich. Verglichen mit anderen Ansätzen, die die Konvergenzgeschwindigkeit eines Mass-Spring-Ansatzes erhöhen, zeigt Dragnet bei der Modellierung einer Membran den höchsten Geschwindigkeitszuwachs. Topologische Änderungen wie Reißen und Schneiden lassen sich ohne zusätzlichen Aufwand modellieren. Im Gegensatz zu Chain-Mail modelliert Dragnet auch lokale Rotationen.

Ein Nachteil von Dragnet ist der relativ hohe Rechenaufwand, der durch die interaktionspunkt-basierte Abarbeitung der Strings bedingt ist. Ein weiterer Schwachpunkt des Ansatzes ist, dass mit Dragnet keine elastischen Deformationen modelliert werden können. Aus diesem Grund wird Dragnet für die Anwendungen in Kapitel 5 mit einer Mass-Spring-Simulation gekoppelt. Zukünftig könnte man den Dragnet-Strings auch elastische Eigenschaften zuordnen.

Der Dragnet Algorithmus wurde unter Grimm et. al. [2004] veröffentlicht.

| Simulation | Modellierung | Deformationen | Zeitverhalten | Topologische Änderungen | Konvergenz-geschwindigkeit | Numerische Stabilität |
|---|---|---|---|---|---|---|
| Finite-Elemente-Methode | Kontinuums-mechanischer Ansatz, kraftbasiert, implizite oder explizite Integration | elastisch | abhängig von Elastizitäts- und Integrationsan-satz, insgesamt recht hoch | teuer bei impliziter Integration $O(\#Knoten^3)$ | abhängig von Integration | abhängig von verwendeter Integration, gering bei nichtlinearer Elastizität |
| Mass-Spring (mit expliziter Integration) | kraftbasiert, physikalisch Orientiert, deskriptiv | elastisch | $O(\#Federn)$ (Kraftberech-nung) + $O(\#Knoten)$ (Integration) | $O(1)$ | gering | gering |
| Tensor-Mass | kraftbasiert, physikalisch mit mass-lumping, explizite Integration | elastisch | $O(\#Tetraeder)$ (Kraftberech-nung) + $O(Knotenzahl)$ (Integration) | $O(1)$ | gering | nur bedingt stabil |
| Gitterlose Verfahren | physikalisch | elastisch, plastisch und Schmelzen möglich | hoch | im Modell berücksichtigt | abhängig von Integration | abhängig von verwendetem Integrationsan-satz |

**Tabelle 2.2:** Überblick über die verschiedenen kraftbasierte Simulationsansätze.

| Simulation | Modellierung | Deformationen | Zeitverhalten | Topologische Änderungen | Konvergenz-geschwindigkeit | Numerische Stabilität |
|---|---|---|---|---|---|---|
| Längenkorrektur | verschiebungs-orientiert, deskriptiv mit physikalischer Motivation | plastischer und elastischer Anteil | $O(\#Federn)$ | $O(1)$ | gering | stabil, erhöht die numische Stabilität einer Mass-Spring-Simulation |
| Chain-Mail | deskriptiv, verschiebungs-orientiert, x-, y- und z-Achse unabhängig, keine Rotationen möglich, von Interaktionknoten ausgehend | plastisch, volumenerhaltung | abhängig von der Anzahl der Knoten die verschoben werden | $O(1)$ | Stabiler Zustand nach jedem Durchgang erreicht | stabil |
| Dragnet | deskriptiv, verschiebungs-orientiert, von Interaktionknoten ausgehend | plastisch | $Omega(1)$ $O(\#Strings)$ | $O(1)$ | schnell, in der Regel wenige Zeitschritte | stabil, erhöht die numische Stabilität einer Mass-Spring-Simulation |

**Tabelle 2.3:** Überblick über die verschiedenen verschiebungsorientiert Simulationsansätze.

# 3

# Modellierung von Reißen

> „To know that we know what we know, and to know that we do not know what we do not know, that is true knowledge“
>
> —Nikolaus Kopernikus [1]

Kapitel 3 beschäftigt sich mit der Modellierung von reißendem Gewebe. Speziell ist ein Algorithmus gesucht, mit dem das Reißen von Membranen in Echtzeit modelliert werden kann.

Bei der Modellierung von Reißen sind zwei grundlegende Fragestellungen zu beantworten. Wie modelliert man Risse innerhalb eines Simulationsgitters und wie wird entscheiden, wann, wo und in welche Richtung gerissen wird? Demzufolge gliedert sich der Stand der Technik in Abschnitt 3.1 in zwei Teile (Abschnitt 3.1.2 und 3.1.1). Als Lösungsansatz wird zunächst eine einfache Methode diskutiert, die Reißen durch Löschen überdehnter Federn bei einer Mass-Spring-Simulation modelliert (Abschnitt 3.2). In Abschnitt 3.3 wird ein flächenerhaltender Reißalgorithmus vorgestellt, der im Rahmen dieser Arbeit entwickelt wurde. Der Algorithmus dient dazu das Reißverhalten bestimmter Membrantypen (siehe Abschnitt 3.3.1) abzubilden. Aus den zusammengetragenen Anforderungen (Abschnitt 3.3.2) wurde ein Algorithmus (Abschnitt 3.3.3) entwickelt. In Abschnitt 3.3.4 werden die Ergebnisse des Ansatzes diskutiert. Zunächst wird demonstriert, dass der Ansatz auch geeignet ist, um spröde Materialien zu modellieren (Abschnitt 3.3.4). Anschließend wird der Algorithmus mit dem Reißen überdehnter Federn in Bezug auf Geschwindigkeit und Reißverhalten verglichen (Abschnitt 3.3.4).

[1] Polnischer Astronom und Mathematiker (1473–1543)

## 3.1 Stand der Technik

In der wissenschaftlichen Literatur findet sich wenig über die Echtzeitsimulation von reißendem Gewebe. Zwei verwandte Themen mit zum Teil identischen Fragestellungen werden häufig diskutiert: die Animation von Brüchen und das Schneiden von biologischem Gewebe während einer virtuellen Operation.

### 3.1.1 Modellierung von Brüchen

Bei der Animation von Brüchen meist spröder Objekte steht die Frage nach dem Ort und Zeit der Bruchentstehung und der Bruchausbreitung im Vordergrund. Ähnliche Fragestellungen sind auch für die Modellierung von Rissentstehung und Rissausbreitung zu klären. Die im Folgenden diskutierten Ansätze kommen aus dem Bereich der Computergrafik, genauer der offline Animation von Spezialeffekten. Die Geschwindigkeit der Algorithmen spielt daher eine eher untergeordnete Rolle. Zeitaufwendige Veränderungen der Gitterstruktur werden in Kauf genommen.

**Bemerkung:** Unter spröden Materialien versteht man in diesem Zusammenhang Materialien, die sich unter Belastung nicht signifikant verformen sondern ab einem Schwellwert bersten. In spröden Materialien führen beispielsweise Unregelmäßigkeiten in der Gitterstruktur zu Spannungen. Die Energie der Spannungen ist groß genug, um nach der Entstehung eines Risses dessen autonome Rissausbreitung zu ermöglichen. Der Riss pflanzt sich dabei mit Schallgeschwindigkeit fort. Ein Beispiel für ein sprödes Objekt in diesem Sinne ist ein Luftballon. Die elastische Energie des aufgeblasenen Ballons reicht für eine Rissentstehung nicht aus. Sticht man mit einer Nadel in den Ballon und erzeugt somit einen Anfangsriss, sind die Spannungen groß genug, damit sich der Riss ausbreitet. Der Luftballon zerplatzt.

Pionierarbeit bei der Modellierung von unelastischen Deformationen wurde von Terzopoulos und Fleischer [1988] geleistet. Neben plastischen und viskoelastischen Verformungen beschreiben Terzopoulos und Fleischer auch das Problem des Brechens von Materialien. Sie beschreiben, dass Material ab einer bestimmten Auslenkung bricht. Grund hierfür sind lokale Singularitäten in der Spannung, die an den Ecken von Störstellen im Festkörper entstehen.

Mazarak, Martins und Amanatides [1999] beschreiben ein Modell zur Animation von Explosionen. Das Modell basiert auf einer Rigid-Body Simulation auf Volumenelement- *(Voxel-)* Ebene. Die Voxel eines Objektes sind starr miteinander verbunden. Eine mathematische Beschreibung der Druckwelle während einer Explosion liefert die Belastung der Verbindungselemente des

simulierten Objekts. Überschreitet die Belastung den Schwellwert einer Verbindung, wird diese getrennt. Mazarak et. al. modellieren ein explorierendes Gebäude mit dieser Technik.

Neff und Fiume [1999] entwickeln ein rein deskriptives Modell zur Beschreibung von Rissen. Anders als Mazarak et. al. unterscheiden Neff und Fiume zwischen der Entstehung, der Ausbreitung und der Verzweigung eines Risses. Ausgehend von einem Anfangsriss entsteht ein Rissbaum. Ein Riss verzweigt sich, wenn die frei werdende elastische Energie einen kritischen Wert überschreitet. Der Winkel, in dem sich der Riss verzweigt, kann variiert werden. Jeder Riss breitet sich solange aus, bis er auf einen anderen Riss oder das Ende des simulierten Objekts trifft. Als Beispiel wird das Zerspringen eines Fensters animiert.

Smith, Witkin und Baraff [2000] beschreiben ein Modell zur Modellierung von spröden Materialien, das auf Punktmassen basiert, die mit Linienelementen verbunden sind. Die Länge der Linienelemente unterliegt einer Zwangsbedingung. Die Kräfte, die zur Aufrechterhaltung der Zwangsbedingung nötig sind, werden ausgewertete, um zu entscheiden, wann und wo das Objekt zerbricht und um die Geschwindigkeit zu ermitteln, mit der die Einzelteile auseinander fliegen. Um das Wachstum der Risse zu beschleunigen, werden die Elemente in der Umgebung eines gerissenen Elementes im nächsten Zeitschritt geschwächt. Die Schwächung der Elemente ist vom Winkel zwischen dem Element und dem gerissenen Element abhängig. Smith et. al. modellieren mit ihrer Methode das Zerbrechen eines Weinglases, eines Tongefäßes und eines Glastischs.

Der Ansatz von O'Brien und Hodgins [1999] basiert auf einer kontinuumsmechanischen Formulierung des Problems. Ein linearer Finite-Elemente-Ansatz liefert den lokalen Spannungstensor, aus dem die internen Kräfte auf jeden Elementknoten berechnet werden. Wirken auf einen Knoten starke Kräfte in verschiedene Richtungen, wird der Knoten in zwei separate Knoten aufgespaltet. Eine Rissebene wird definiert. Die Finiten-Elemente, in denen der gerissene Knoten enthalten ist, werden verfeinert, um die Rissebene zu modellieren. O'Brien und Hodgins modellieren mit ihrem Ansatz die Zerstörung einer Wand durch eine Kanonenkugel und das Zerspringen einer Tasse, die auf den Boden fällt. In dem Reißmodell von O'Brien und Hodgins können sich Risse pro Zeitschritt nur um jeweils ein Element ausbreiten. Für die Modellierung von spröden Materialien sind somit viele Zeitschritte nötig, um die Ausbreitung eines Risses vollständig zu berechnen.

Müller et. al. [2001] präsentieren einen hybriden Ansatz zur Modellierung von Brüchen. Eine Rigid-Body-Simulation berechnet die Flugbahn einzelner Objektstücke. Die quasi-statische Finite-Element-Methode berechnet die Reaktion des simulierten Körpers, wenn Kollisionen erkannt werden. Der geometrische nicht-lineare Ansatz liefert Deformationen und die lokalen

Spannungstensoren. Überschreitet der größte positive Eigenwert des Spannungstensors einen materialabhängigen Schwellwert, bricht das Objekt. Der zu dem Eigenwert gehörende Eigenvektor bestimmt die Ebene des Bruchs. Die Ausdehnung des Bruchs wird aus der Größe des Eigenwertes bestimmt. Die Tetraeder in der Umgebung der Bruchstelle werden auf beide Seiten der Bruchebene aufgeteilt. Die Methode von Müller et. al. erlaubt Echtzeitsimulationen durch den einfachen Ansatz einer Rigid-Body Simulation. Der Rechenaufwand erhöht sich aber drastisch, wenn Kollisionen auftreten.

### 3.1.2 Modellierung von Schneiden

Bei der Modellierung von Schnitten im Gewebe bestimmt die Form des virtuellen Schneideinstruments, wo und wann ein Schnitt entsteht und wie er verläuft. Der Benutzer bewegt das virtuelle Instrument. Eine Kollisionserkennung bestimmt die Schnittlinien im Gewebe. Das Kernproblem bei der Modellierung von Schneiden liegt in der Modellierung der Schnittkanten und deren Berücksichtigung in Simulation und Visualisierung unter der Echtzeitrandbedingung. Ein ähnliches Problem muss auch bei der Modellierung von Rissen gelöst werden. Aus diesem Grund wird im Folgenden der Stand der Technik bei der Modellierung von Schnitten diskutiert.

Cotin et. al. [2000] modellieren das Schneiden in Volumenmodellen, die aus Tetraedern aufgebaut sind und mit dem Tensor-Mass-Modell (siehe 2.2.6) simuliert werden. Tetraederelemente, die in das virtuelle Schneideinstrument eindringen, werden aus der Gitterstruktur entfernen. Hauptnachteil bei der Modellierung von Schneiden durch Entfernung von Elementen ist Entstehung von Löchern im Gitter. Vor allem bei groben Diskretisierungen und geringer Deformation im Bereich der Schnittkante fallen Volumenverlust und der ausgefranste Kantenverlauf optisch störend auf.

Um den Effekt zu minimieren, verfeinern Forest, Delingette und Ayache [2002] das Gitter dynamisch in der Umgebung, in der die Interaktion stattfindet. Verfeinerungen der Gitterstruktur führen aber zu einem erhöhten Rechenaufwand und können bei der expliziten Integration des Tensor-Mass-Modells zu numerischen Instabilitäten führen (siehe Abschnitt 2.1.3).

Bielser, Maiwald und Gross [1999] ersetzen die geschnittenen Tetraeder durch mehrere kleine Tetraeder. Dabei wird die Aufteilung des Tetraeders an den Verlauf der Schnittlinie angepasst. Die neuen Tetraeder links der Schnittlinie werden anschließend von den Tetraedern auf der rechten Seite getrennt. Das Verfahren führt zu einer glatten Schnittlinie und erhält das Volumen des simulierten Körpers. Es entstehen weniger neue Elemente als bei dem Ansatz von Forest et. al. [2002], was das Problem des erhöhten Rechenbedarfs reduziert. Mor und Kanade [2000] verwenden einen ähnlichen Ansatz wie Bielser et. al., spalten aber die Tetraeder bereits, wenn sie

erst teilweise geschnitten sind. Diese als *progressive cut* bezeichnete Technik führt zu einer flüssigeren Ausbreitung der Schnittlinie.

Ganovelli, Cignoni, Montani und Scopigno [2000] entwickelt einen Schneidealgorithmus, der mit der Aufteilung von Tetraederelementen auf einer *Multiresolution* Gitterstruktur arbeitet, um die Simulation nach Bedarf auf verschiedenen Detailstufen (*Level-Of-Detail*) laufen zu lassen.

Nienhuys und van der Stappen [2001] passen das Simulationsgitter an die Schnittlinie an, ohne die Anzahl der Tetraeder zu erhöhen. Sie verschieben die Knoten innerhalb des Gitters in der Art, dass die Schnittlinie durch Gitterkanten näherungsweise beschrieben wird. Anschließend trennen sie die Elemente auf beiden Seiten der Schnittlinie. In [2002] verfeinern Nienhuys und van der Stappen ihre Technik. Eine auf der *Delaunay*-Triangulierung basierende *flip* Operation dreht Tetraeder, um bei der Anpassung der Gitterstruktur numerisch gutmütigere Elemente zu erhalten.

## 3.2 Reißen durch Löschen überdehnter Federn

Im Folgenden wird ein einfacher Ansatz beschrieben, das Reißen in einem System aus Federn, Massepunkten und Dreiecken zu modellieren. Der beschriebene Algorithmus findet beispielsweise Anwendung, um das Reißen der Epiretinalen Membranen im EYESI-*Vitreoretinal* Simulator zu modellieren. Der Ansatz wird als Referenz für den flächenerhaltenden Reißalgorithmus verwendet, der in Abschnitt 3.3 vorgestellt wird.

### 3.2.1 Grundlegende Idee

Ausgangspunkt des Modells ist die von Terzopoulos und Fleischer [1988] (siehe oben) beschriebene Annahme, dass ein Material reißt, sobald eine bestimmte materialspezifische Spannung lokal überschritten wird. Bei einer *Hooke'schen* Feder ist die Veränderung der Federlänge über die Federkonstante direkt mit der Spannungsänderung verknüpft. Eine Feder reißt, sobald ihre Länge einen Maximalwert überschreitet. Der Schwellwert ist in der Regel ortsabhängig, da in realen Objekten Gebiete mit verschieden starken Strukturen existieren.

Bei der Modellierung der topologischen Änderungen ist darauf zu achten, dass das Gitter sich immer in einem konsistenten Zustand befindet. Per Definition ist ein Gitter dann nur dann konsistent wenn:

- jeder Knoten mit mindestens einem Dreieck verbunden ist
- keine zwei Dreiecke mit denselben drei Knoten verbunden sind

- entlang jeder Dreieckskante genau eine Feder verläuft
- jede Feder entlang genau einer oder zweier Dreieckskanten verläuft

### 3.2.2 Der Algorithmus

Bei jedem Zeitschritt werden die Längen aller Federn überprüft. Überschreitet eine Feder ihre maximale Länge, wird sie gelöscht. Dreiecke mit einer Kante entlang der gelöschten Feder werden ebenfalls gelöscht, um die Konsistenz des Gitters zu erhalten. Entstehen durch das Löschen der Dreiecke isolierte Knoten, die mit keinem anderen Dreieck verbunden sind, werden die Knoten ebenfalls gelöscht. Algorithmus 3.1 beschreibt das Reißen durch Löschen von Federn. Dabei ist $S$ die Menge aller Federn, $t(n_i)$ und $t(s_i)$ sind die Mengen aller Dreiecke, die mit Knoten $n_i$ verbunden sind bzw. Feder $s_i$ als Kante haben.

**Algorithmus 3.1** ReißenDurchLöschen()

```
1: for all s_i ∈ S do
2:   if Länge(s_i) > maxLänge(s_i) then
3:     for all t_i ∈ T(s_i) do
4:       for all n_i ∈ N(t_i) do
5:         for all t'_i ∈ T(n_i) do
6:           if NOT t'_i ∈ T(s_i) then
7:             next n_i;
8:           delete n_i;
9:       delete t_i;
10:    delete s_i;
```

### 3.2.3 Ergebnisse

Der Hauptnachteil des vorgestellten Ansatzes liegt im Löschen der Dreiecke während des Reißens begründet. Das Reißen führen zu einer merklichen Reduktion der Oberfläche (siehe Abbildung 3.1). Bei grober Auflösung wird das Reißen als ein Verschwinden von Dreiecken wahrgenommen.

Spröde Objekte lassen sich mit dem Ansatz nicht modellieren. Da nicht zwischen Rissentstehung und Rissausbreitung unterschieden wird, entsteht ein Riss, sobald die inneren Spannungen anliegen, die zur Rissausbreitung nötig wären.

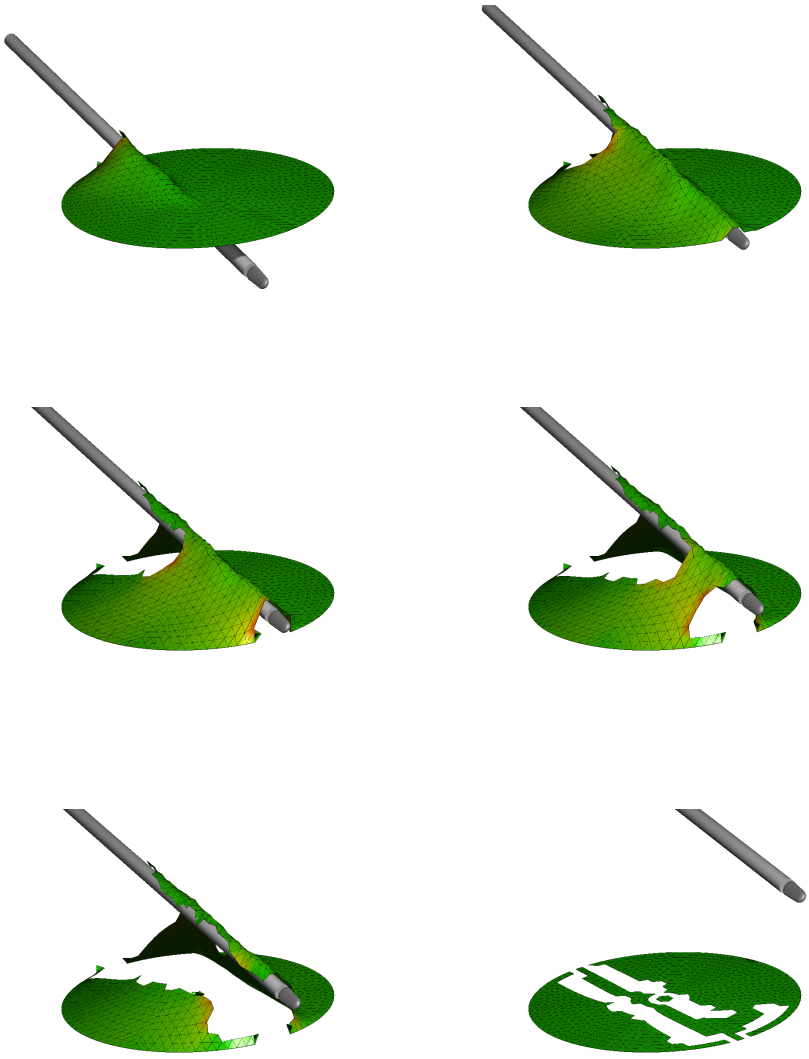

**Abbildung 3.1:** Ein Instrument dehnt eine virtuelle Membran, bis sie reißt. Der Reisalgorithmus löscht überdehnte Federn und die anliegenden Dreiecke. Dadurch kommt es zu einer Reduktion der Fläche der simulierten Membran im Bereich des Risses. Im letzten Bild wurden alle Knoten des Simulationsgitters auf ihre Anfangsposition zurückgesetzt. Dadurch wird ersichtlich, welche Dreiecke gelöscht wurden.

## 3.3 Flächenerhaltendes Reißmodell

Im Folgenden wird ein deskriptives Reißmodell beschrieben, das auf einem Mass-Spring-Gitter beruht.

Das Modell wurde entwickelt um das Reißen von steifen Membranen in Echtzeit, zu modellieren. Um unter der Echtzeitrandbedienung numerische Stabilität zu garantieren, darf die Auflösung eines Feder-Masse-Gitters nicht zu hoch gewählt werden (siehe Abschnitt 2.1.3). Dadurch ist der einfache Reißansatz aus Abschnitt 3.2.2 für steifes Gewebe nicht geeignet.

Die Konsistenzdefinition wird für das flächenerhaltende Reißen gegenüber dem Reißen durch Löschen von Federn um einen Punkt ergänzt. Dadurch wird sichergestellt, dass zwei Gitterteile immer flächig verbunden sind.

- Jeder Knoten ist mit mindestens einem Dreieck verbunden.
- Es existieren keine zwei Dreiecke mit denselben drei Knoten verbunden sind.
- Entlang jeder Dreieckskante verläuft genau eine Feder.
- Jede Feder verläuft entlang einer oder zwei Dreieckskanten.
- Alle Dreieckspaare, die einen gemeinsamen Knoten haben, haben auch eine gemeinsame Kante.

### 3.3.1 Beobachtungen

Grundlage des Modells, ist eine Reihe von Beobachtungen, die bei der Auswertung von Reißversuchen und Videoaufnahmen medizinischer Membranentfernungen gemacht wurden.

- Das Reißen wird durch das Dehnen einer Membran ausgelöst. Stauchungen führen nicht zu einem Reißen.
- Die Ausbreitung eines vorhandenen Risses ist deutlich wahrscheinlicher als die Entstehung eines neuen Risses.
- Neue Risse entstehen bevorzugt am Rand einer Membran. Die Bildung eines neuen Risses in der Mitte des Gewebes ist unwahrscheinlicher.
- Inhomogenitäten im Gewebe beeinflussen die Rissausbreitung.

### 3.3.2 Anforderungen

Neben der Modellierung der Beobachtungen aus Abschnitt 3.3.1 soll der entwickelte Algorithmus noch folgende Randbedingungen erfüllen.

- Echtzeitfähigkeit
- Keine Veränderung des Flächeninhaltes durch das Reißen.
- Der Reißalgorithmus soll auf einem Gitter aus Knoten, Federn und Dreiecken arbeiten, das die obigen Konsistenzbedingungen erfüllt.

### 3.3.3 Algorithmus

Im Folgenden werden einige Begriffe definiert, die bei der Beschreibung des Algorithmus benötigt werden. Die Definitionen gelten für ein Simulationsgitter aus Knoten, Federn und Dreiecken, das die oben beschriebenen Konsistenzbedingungen erfüllt.

**Definition 3.1** (Kantenfeder)**.** *Eine Feder ist genau dann eine Kantenfeder, wenn sie entlang der Kante von genau einem Dreieck verläuft.*

**Definition 3.2** (Kantenknoten)**.** *Ein Knoten ist genau dann ein Kantenknoten, wenn er mit mindestens einer Kantenfeder verbunden ist.*

**Definition 3.3** (innerer Knoten)**.** *Alle Knoten die keine Kantenknoten sind heißen innere Knoten.*

**Definition 3.4** (Rissknoten)**.** *Der Rissknoten ist der Knoten, von dem ausgehend sich ein neuer Riss bildet oder ein vorhandener Riss ausbreitet.*

**Definition 3.5** (Rissfeder)**.** *Die Rissfeder ist die Feder, in deren Richtung sich ein Riss vom Rissknoten ausgehend, ausbreitet.*

**Definition 3.6** (Endknoten des Risses)**.** *Der Endknoten des Risses ist der Knoten der Rissfeder, der nicht der Rissknoten ist.*

**Definition 3.7** (Kantennachbarn)**.** *Die Kantennachbarn eines Knoten $n_i$ sind die Knoten, die mit $n_i$ durch eine Kantenfeder verbunden sind.*

**Definition 3.8** (Risskeim)**.** *Ein Kantenknoten ist genau dann ein Risskeim, wenn einer seiner beiden Kantennachbarn durch Verdoppelung des anderen Kantennachbarn entstanden ist.*

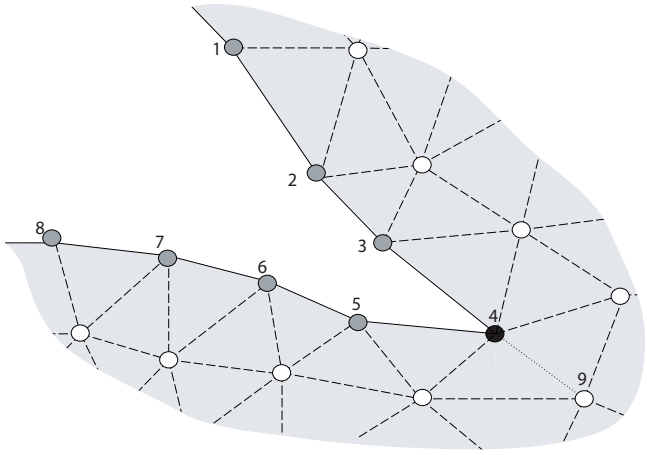


**Abbildung 3.2:** Beim Reißen werden drei Arten von Knoten unterschieden. Risskeime (schwarz) am Ende eines Risses, Kantenknoten (grau) und Knoten innerhalb des Gitters (weiß). Kantenfedern sind in der Abbildung als durchgezogen Linien dargestellt, andere Federn sind gestrichelt. Bei der Ausbreitung des Risses von Knoten 4 zu Knoten 9 wird 4 zum Rissknoten und die gepunktete Linie zwischen 4 und 9 markiert die Rissfeder. Die Dreiecksflächen des Simulationsgitters sind grau hinerlegt.

Abbildung 3.2 verdeutlicht die obigen Definitionen. Die inneren Knoten sind weiß, normale Kantenknoten sind grau und Risskeime sind schwarz eingefärbt. Kantenfedern sind durch durchgezogene Linien gekennzeichnet, während andere Federn gestrichelt sind. Die Kantennachbarn zu Knoten 6 in der Abbildung sind die Knoten 5 und 7. Die Kantennachbarn des Risskeims 4 sind die Knoten 3 und 5. Geht man davon aus, dass sich der Riss von Knoten 4 zu Knoten 9 ausbreitet, wäre Knoten 4 ein Rissknoten und die gepunktete Feder zwischen den Knoten 4 und 9 die Rissfeder. Die Dreiecke des Simulationsgitters sind grau eingefärbt.

Bei der Entwicklung eines Algorithmus, der das Reißen modelliert, sind zwei Punkte zu beachten. Zum einen muss festgelegt werden, wie Risse in der Gitterstruktur abgebildet werden. Zum anderen benötigt man Kriterien, die bestimmen wann, wo und in welche Richtung das Gitter reißt.

### Risskriterien

Die Knotenspannung $S(n_i)$ wird als Kriterium verwendet, um zu entscheiden, wann und an welchem Knoten ein Loch entsteht oder sich ein Riss ausbreitet oder verzweigt. Die Spannung an einem Knoten wird durch die Summe der halben Federspannungen $S(s_{ij})$ der mit dem Knoten verbundenen gedehnten Federn bestimmt. Um inhomogene Strukturen im Gitter

abbilden zu können, wird die Spannung noch mit einem knotenabhängigen Faktor $t_i$ gewichtet.

$$S(n_i) \quad = \quad t_i \quad \sum_j \quad \frac{1}{2} \quad S(s_{ij}) \tag{3.1}$$

Ein Knoten reißt, wenn seine Knotenspannung einen Schwellwert überschreitet. Ausgehend von den Beobachtungen aus Abschnitt 3.3.1 wird bei der Wahl des Schwellwerts zwischen drei Arten von Knoten unterschieden: Risskeime, Kantenknoten und innere Knoten (siehe die Definitionen 3.8, 3.2 und 3.3).

Risskeime sind für die Ausbreitung bestehender Risse verantwortlich. Da die Ausbreitung von Rissen wahrscheinlicher ist als die Entstehung von Löchern und die Verzweigung von Rissen, wird der Schwellwert der Risskeime in der Regel am kleinsten gewählt.

Ein bestehender Riss verzweigt sich, wenn der Rissknoten ein Kantenknoten aber kein Risskeim ist. Kantenknoten haben in der Regel einen mittleren Schwellwert für das Reißen. Die Verzweigung von Rissen kommt nicht so häufig vor wie die Ausbreitung eines bestehenden Risses.

Normalerweise wird der Schwellwerte für die inneren Knoten am höchsten gewählt, damit die Rissausbreitung und -verzweigung gegenüber der Lochbildung bevorzugt wird. Die Lochbildung ist in vielen Fällen lediglich für die Initiierung des Rissvorgangs nötig.

Zur Bestimmung des Rissknotens werden alle Knoten geprüft, die mit mindestens einer Feder verbunden sind, die keine Kantenfeder ist. Nach Gleichung 3.1 wird die Knotenspannungen bestimmt. Überschreitet mindestens ein Knoten $n_i$ den für seinen Knotentyp festgelegten Schwellwert $T(n_i)$, kommt es zu einem Reißvorgang. Wird der Schwellwert von mehreren Knoten überschritten, reißt nur der Knoten mit der größten relativen Überschreitung $r(n_i)$.

$$r(n_i) = \frac{S(n_i) - T(n_i)}{T(n_i)} \tag{3.2}$$

Zur Bestimmung der Rissfeder werden die Federn, die mit dem Rissknoten verbunden sind und keine Kantenfedern sind überprüft. Die Feder mit der geringsten relative Federlänge $l_r(sij)$ wird als Rissfeder festgelegt.

$$l_r(sij) = \frac{l(s_{ij})}{l_0(s_{ij})} \tag{3.3}$$

wobei $l(s_{ij})$ und $l_0(s_{ij})$ die aktuelle und die Nulllänge der Feder $s_{ij}$ ist.

**Bestimmung des Knotentyps** Zu obiger Beschreibung der Knotentypen: Risskeim, Kantenknoten und innerer Knoten wird im Folgenden betrachtet, wie der Knotentyp algorithmisch bestimmt wird.

**Algorithmus 3.2** IstKantenknoten($n_i$)

```
1: for all s_i ∈ S(n_i) do
2:    unsinged int dreiecksCounter(0);
3:    for all t_i ∈ T(n_i) do
4:       for all n_j ∈ N(t_i) do
5:          if n_j ∈ S(n_i) AND n_j ≠ n_i then
6:             dreiecksCounter++;
7:    if dreiecksCounter < 2 then
8:       return true;
9: return false;
```

Algorithmus 3.2 prüft, ob ein Knoten $n_i$ ein Kantenknoten ist. Dazu werden alle Federn, die mit dem Knoten verbunden sind, überprüft. Verläuft eine Feder entlang der Kante von nur einem Dreieck, ist die Feder eine Kantenfeder und der Knoten $n_i$ ein Kantenknoten.

Zur Beschleunigung des häufig nötigen Kantentests besitzt jeder Knoten ein Kantenflag. Zu Beginn der Simulation wird der Wert des Kantenflags für jeden Knoten mit dem Algorithmus 3.2 bestimmt. Im Verlauf der Simulation entstehen neue Gitterkanten. Das Flag wird für alle Knoten gesetzt, die durch das Reißen verdoppelt werden, neu entstehen oder bei denen ein Riss endet.

Algorithmus 3.3 bestimmt den Knotentyp eines Knotens $n_i$. Ist $n_i$ kein Kantenknoten, dann ist $n_i$ ein innerer Knoten. Wenn $n_i$ ein Kanteknoten ist, werden seine beiden Kantennachbarn gesucht. Haben beide Kantennachbarn die gleiche Anfangsposition, ist einer der Knoten durch Verdoppelung des anderen Knotens entstanden. Dann ist $n_i$ ein Risskeim.

### Änderungen im Gitter

Ausgehend von der Randbedingung der Flächenerhaltung, werden beim Reißen keine Dreiecke gelöscht. Stattdessen wird das Reißen durch eine Trennung der Verbindung zweier Dreiecke verwirklicht.

Bei den Änderungen der Gitterstruktur wird zwischen der Entstehung eines neuen Lochs, der Ausbreitung oder Entstehung eines Risses und dem Durchreißen des Gitters unterschieden.

**Entstehung eines Lochs** Die Position und Ausdehnung eines neuen Lochs wird durch den Rissknoten und zwei Rissfedern festgelegt. Die Mo-

**Algorithmus 3.3** BestimmeKnotenType($n_i$)

```
1: if IstKantenknoten(n_i) != true then
2:   return IST_INNERER_KNOTEN
3: kantenNachbar1(0);
4: kantenNachbar2(0);
5: for all s_i ∈ S(n_i) do
6:   for all t_i ∈ T(n_i) do
7:     for all n_j ∈ N(t_i) do
8:       if n_j ∈ S(n_i) AND n_j ≠ n_i then
9:         if kantenNachbar1 == 0 then
10:          kantenNachbar1 = n_j;
11:        else
12:          kantenNachbar2 = n_j;
13:          break;
14: if startPosition(kantenNachbar1) == startPosition(kantenNachbar2)
    then
15:   return IST_RISSKEIM;
16: else
17:   return IST_KANTENKNOTEN;
```

dellierung des Lochs erfolgt in den folgenden Schritten.

1. Der Rissknoten wird verdoppelt.
2. Die Rissfedern werden verdoppelt. Der Rissknoten wird in der neu erstellten Feder durch den neuen Knoten ersetzt.
3. Für alle Federn und Dreiecke, die mit dem Rissknoten verbunden sind, wird ermittelt, auf welcher Seite des Lochs sie liegen. Bei den Federn und Dreiecken, die auf einer gewählten Seite des Lochs liegen, wird ohne Beschränkung der Allgemeinheit, der Rissknoten durch den neu erstellten Konten ersetzt.

Die Modellierung des Lochs wird in Abbildung 3.3 veranschaulicht.

**Verzweigung und Ausbreitung eines Risses** Von dem Rissknoten ausgehend verläuft der Riss entlang der Rissfeder. Das Reißen wird in vier Schritten modelliert.

1. Der Rissknoten wird verdoppelt.
2. Die Rissfeder wird verdoppelt. Der Rissknoten wird in der neuen Feder durch den neuen Knoten ersetzt.

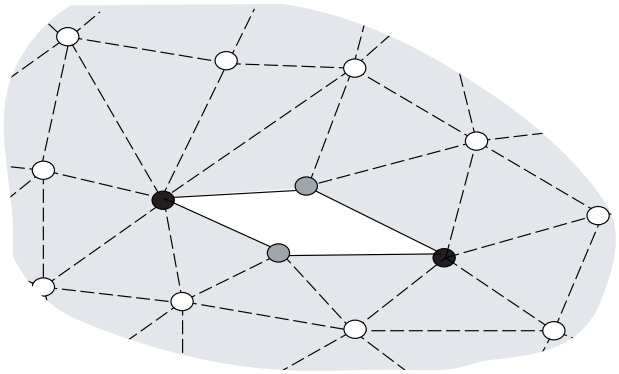

**Abbildung 3.3:** Die Entstehung eines Lochs im Gitter. Der Rissknoten (grau) und zwei anliegende Rissfedern werden verdoppelt.

3. Für alle Federn und Dreiecke, die mit dem Rissknoten verbunden sind, wird ermittelt, auf welcher Seite des Risses sie liegen. Bei den Federn und Dreiecken, die auf einer gewählten Seite des Risses liegen, wird ohne Beschränkung der Allgemeinheit der Rissknoten durch den neu erstellten Konten ersetzt.

4. Ist der Endknoten des Risses ein Kantenknoten, wird das Gitter an dieser Stelle getrennt.

Die ersten drei Schritte des Reißens sind in Abbildung 3.4 veranschaulicht. Der vierte Schritt wird im Folgenden erläutert.

**Durchreißen des Gitters** Erreicht ein Riss eine Kante, verliert das Gitter seinen konsistenten Zustand. Es existiert ein Dreieckspaar mit einem gemeinsamen Knoten, das keine gemeinsame Kante hat (siehe Abbildung 3.5). Um das Gitter wieder in einen konsistenten Zustand zu überführen, werden die Gitterteile an dieser Stelle getrennt.

1. Der Knoten, an dem der Riss eine Kante trifft, wird verdoppelt.

2. Federn und Dreiecke auf beiden Seiten des Risses werden zwischen dem neuen und dem alten Knoten aufgeteilt.

Der Vorgang ist in Abbildung 3.5 veranschaulicht.

**Aufteilung der Federn und Dreiecken** Alle oben beschriebenen Änderungen des Gitters basieren auf der Verdoppelung des Rissknotens. Anschließend müssen alle mit dem Rissknoten verbundenen Federn und Dreiecke zwischen dem Rissknoten und dem neu entstanden Knoten aufgeteilt

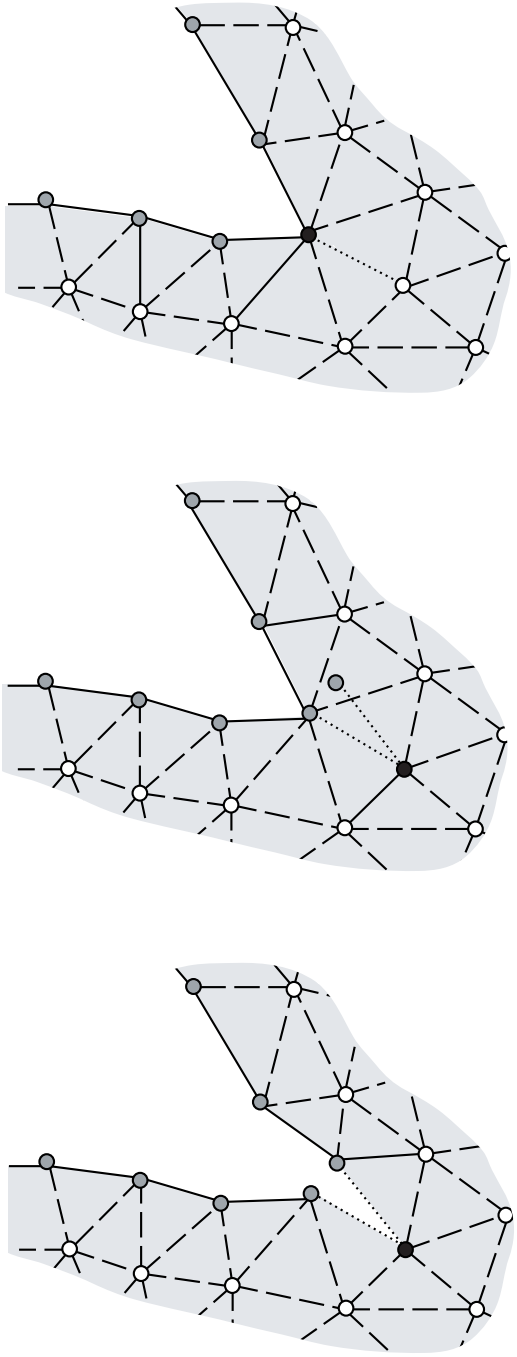

**Abbildung 3.4:** Die Schritte beim Reißen veranschaulicht von oben nach unten. Verdoppelung des Rissknotens, Verdoppelung der Rissfeder, Aufteilung der Federn und Dreiecken des Rissknotens zwischen Rissknoten und dem neu erstellten Knoten.

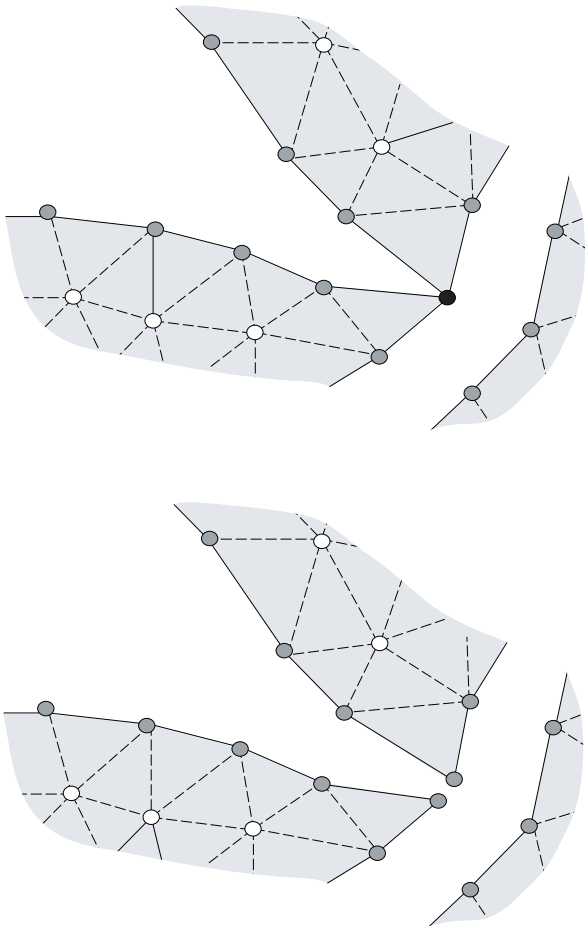

**Abbildung 3.5:** Erreicht der Riss eine Kante (obere Abbildung) wird das Gitter an der Stelle durch Knotenverdoppelung und Aufteilung der Federn und Dreiecke durchtrennt (untere Abbildung).

werden. Federn und Dreiecke auf einer gewählten Seite des Risses bleiben mit dem Risskeim verbunden. Bei den Federn und Dreiecke auf der anderen Seite des Risses werden Rissknoten und neu erstellter Knoten ausgetauscht.

Algorithmisch geschieht dies, indem alle mit dem Rissknoten verbundenen Knoten, die auf einer Seite des Risses liegen, in eine Liste geschrieben werden.

Einer der Kantennachbarn des Risskeims wird gewählt und ohne Beschränkung der Allgemeinheit als erster Knoten in die Liste eingetragen. Anschließend wird das Dreieck gesucht, das sowohl den Rissknoten als auch den Kantennachbarn enthält. Der dritte Knoten des Dreiecks wird an die Liste angehängt. Als Nächstes wird das Dreieck gesucht, das den Rissknoten und den letzten Koten der Liste enthält dessen dritter Knoten aber nicht der vorletzten Knoten der Liste ist. Wird ein entsprechendes Dreieck gefunden, wird der dritte Knoten des Dreiecks in die Liste eingetragen und nach dem nächsten Dreieck gesucht. Dies geschieht solange, bis kein entsprechendes Dreieck gefunden wird oder der dritte Knoten eines gefunden Dreiecks der Endknoten des Risses ist. Abbildung 3.6 veranschaulicht die Bestimmung aller Knoten, die auf einer Seite eines Risses liegen.

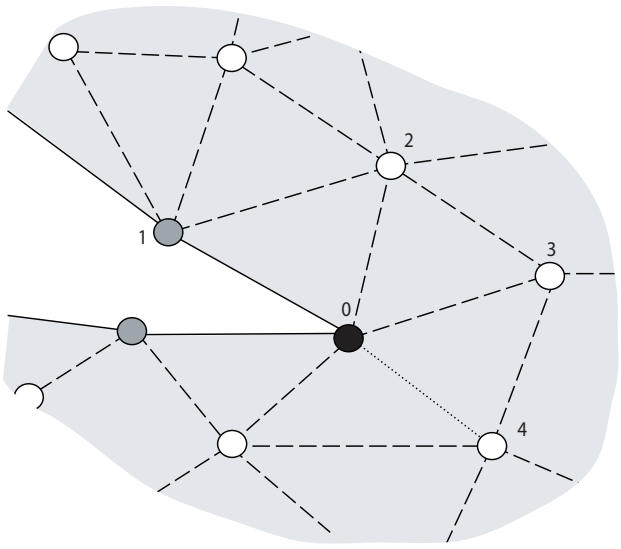


**Abbildung 3.6:** Die Bestimmung aller Knoten, die auf einer Seite des Risses liegen. Von Rissknoten 0 ausgehend wird zunächst einer der beiden Kantennachbarn gesucht (Knoten 1). Anschließend wird das Dreieck gesucht, das sowohl Knoten 0 als auch Knoten 1 enthält. Der dritte Dreiecksknoten ist Knoten 2. Knoten 3 befindet sich in dem Dreieck, das Knoten 0 und Knoten 2, aber nicht Knoten 1 enthält. Analog wird Knoten 4 identifiziert. Hier endet der Algorithmus, da Knoten 4 der Endknoten des Risses ist.

Anschließend wird in allen Dreiecken und Federn, die mit dem Rissknoten

und einem Knoten der erstellten Liste verbunden sind, der Rissknoten durch den neu erstellten Knoten ersetzt.

#### Überblick über den Algorithmus

Der Gesamtalgorithmus ist unter 3.4 veranschaulicht. Zunächst werden in Zeile 1 die Knotenspannungen berechnet. In den Zeilen 2 bis 7 wird nach dem Rissknoten gesucht. Falls ein Rissknoten gefunden wird, wird abhängig vom Knotentyp ein neues Loch erzeugt (Zeile 10) oder die Funktion zur Rissausbreitung (Zeile 13) aufgerufen. Ist der Endknoten des Risses ein Kantenknoten, wird zusätzlich das Gitter durchtrennt (Zeile 15).

**Algorithmus 3.4** FlächenerhaltendesReißen()

```
 1: BerechneKnotenspannungen();
 2: r_max= 0;
 3: n_riss= 0;
 4: for all n_i ∈ N do
 5:    if r(n_i) > r_max then
 6:       r_max = r(n_i);
 7:       n_riss = n_i;
 8: if n_riss then
 9:    if KnotenType(n_riss) == IST_INNERER_KNOTEN then
10:       ErzeugeLoch(n_riss);
11:    else
12:       n_end = BestimmeEndknoten(n_riss);
13:       Rissausbreitung(n_riss, n_end);
14:       if KnotenType(n_end) == IST_KANTEN_KNOTEN then
15:          GitterDurchreißen(n_end);
```

### 3.3.4 Ergebnisse

Der oben beschriebene Algorithmus wurde implementiert. Das Programm entspricht den Anforderungen, die an das Reißen in den Abschnitten 3.3.1 und 3.3.2 gestellt wurden. Abbildung 3.7 zeigt das Reißen einer Beispielmembran. Knotenspannungen sind in Falschfarben dargestellt. grün eingefärbte Knoten weisen keine Spannung auf. Rote Knoten haben eine Knotenspannung in der Nähe des zum Reißen nötigen Schwellwerts.

#### Beispielsimulation platzender Ballon

Der vorgestellte Ansatz wurde entwickelt, um das Reißen von biologischen Membranen zu modellieren. Im Folgenden wird er verwendet, um einen

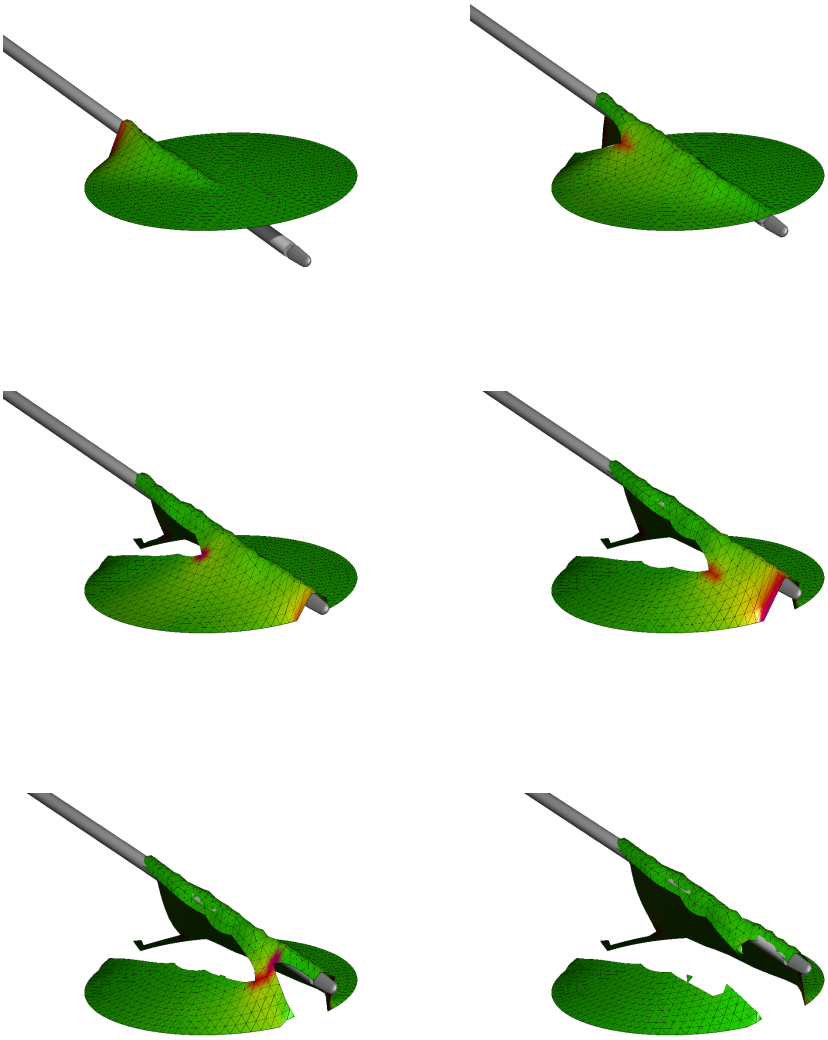

**Abbildung 3.7:** Beispiel für das Reißen eines Gitters mit dem in diesem Kapitel beschriebenen Algorithmus. Die Knotenspannung wird durch eine Einfärbung der Knoten veranschaulicht. Die Spannung der Knoten nimmt von Grün über Rot nach Blau zu.

platzenden Luftballon zu modellieren. Das Beispiel zeigt die Anwendung des Algorithmus auf das in der Literatur häufig beschriebene Zerbersten spröder Materialien.

Das Mass-Spring-Gitter des Ballons hat die Form einer Kugeloberfläche. Die innere Spannung eines spröden Körpers wird durch einen Druck innerhalb des Ballons modelliert. Der Druck vergrößert die Oberfläche des Ballons und sorgt für eine Vorspannung der Federn. Alle Knoten haben eine anfängliche Knotenspannung. Um einen spröden Körper zu simulieren, wird der Reißschwellwert für die inneren Knoten deutlich höher gewählt, als die anfängliche Knotenspannung. Dadurch wird verhindert, dass die Spannungen zu der spontanen Entstehung eines Risses führen. Die Reißschwellwerte für die Kantenknoten und Risskeime werden auf einen Wert deutlich unterhalb der anfänglichen Knotenspannung gesetzt. Dadurch wird sichergestellt, dass Ausbreitung und Verzweigung von Rissen nach der Entstehung eines anfänglichen Lochs ablaufen, ohne dass Energie von außen zugeführt werden muss. Das Verhältnis der Schwellwerte von Risskeimen und Kantenknoten bestimmt die Verzweigungswahrscheinlichkeit eines anfänglichen Risses. Abbildung 3.8 zeigt das Zerplatzen eines Ballons mit verschiedenen Verhältnissen der beiden Größen. Ist das Verhältnis des Schwellwertes eines Kantenknoten bezogen auf den Schwellwert des Risskeims größer, entstehen mehr Rissverzweigungen. Der Ballon zerplatzt unter Entstehung vieler kleiner Teile.

### Vergleich der Reißalgorithmen

Um die Leistungsfähigkeit des vorgestellten Ansatzes zu bewerten, wird er mit dem einfachen Reißansatz aus Abschnitt 3.2 verglichen.

### Laufzeitmessungen

Zunächst wird eine Analyse der Laufzeit der Algorithmen abhängig von der Auslösung des Gitters gemacht.

Alle Laufzeitmessungen wurden auf einem Pentium 4 System mit Hyper-Threading und 2.8 GHz unter Windows XP durchgeführt. Die Anwendung lief hierzu mit Echtzeit-Priorität und wurde an einen virtuellen Prozessor gebunden. Alle anderen Programme einschließlich der Benutzeroberfläche wurden geschlossen und alle unnötigen Dienste angehalten, um Störung der Messungen zu minimieren.

Um einheitliche Bedingungen herzustellen, wurden die Benutzerinteraktionen zunächst aufgezeichnet. Die Aufzeichnung diente beim Messdurchgang als Eingabe für die Simulation. Alle Messwerte sind Mittelwerte aus größenordungsmäßig tausend Messungen. Die Ergebnisse sind in Diagramm

**Abbildung 3.8:** Modellierung eines zerplatzenden Ballons. Die Momentaufnahmen zweier Simulationen sind chronologisch von oben nach unten angeordnet. Zu Veranschaulichung des Reißvorgangs werden die Rissknoten je nach Knotentyp eingefärbt. Innere Knoten werden Blau, Risskeime Rot und andere Kantenknoten Grün koloriert. Bei der Simulation in der linken Spalte ist der Schwellwert für die Verzweigung eines Risses deutlich größer gewählt als der Schwellwert für die Rissausbreitung. Auf der rechten Seite sind beide Werte von der gleichen Größenordnung. Über Anpassung der Schwellwerte lässt sich die Fragmentierung des Ballons regeln.

3.9 veranschaulicht. Es zeigt sich, dass das flächenerhaltende Reißen im Vergleich zum Reißen durch Löschen von Federn langsamer ist. Bei Gittergrößen von einigen tausend Federn benötigt das flächenerhaltende Reißen etwa doppelt so lange. Die Laufzeit beim Reißen durch Löschen steigt linear mit der Anzahl der Federn. Die Kurve beim flächenerhaltende Reißen ist leicht nach unten gekrümmt. Bei 3500 Federn benötigt der flächenerhaltende Algorithmus nur noch gut 50% mehr Rechenzeit.

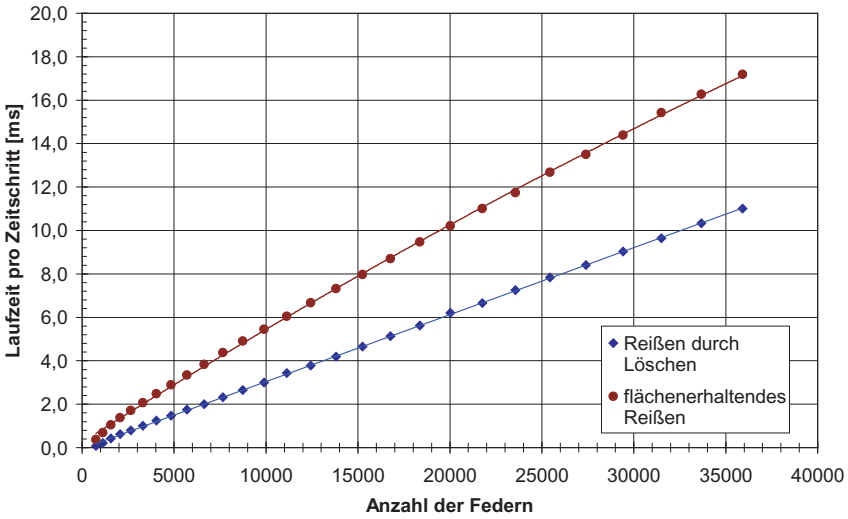


**Abbildung 3.9:** Die Laufzeiten der Reißalgorithmen in Abhängigkeit von der Anzahl der Feder. Reißen durch Löschen von Federn ist schneller als der flächenerhaltende Reißalgorithmus. Beim Reißen durch Löschen steigt die benötigte Rechenzeit linear mit der Anzahl der Feder. Die Kurve beim flächenerhaltenden Ansatz ist leicht gekrümmt und steigt etwas langsamer als linear an. Die zum Diagramm gehörenden Messwerte sind in Tabelle D.3 im Anhang abgedruckt.

### Visuelle Beurteilung

Als zweites Kriterium für die Bewertung des Reißens wurde die visuelle Erscheinung herangezogen. Die Bewegung eines Zylinders durch eine Membran wurde aufgezeichnet und anschließend mit beiden Reißalgorithmen getestet. Die Abbildungen 3.10 bis 3.12 zeigen exemplarisch die Unterschiede beider Ansätze.

Die Unterschiede beider Ansätze werden vor allem bei Gittern grober Auflösung augenscheinlich. Die Modellierung des Reißens durch Löschen führt in diesem Fall zu einer starken Reduktion der Oberfläche und dadurch zu einem nicht plausiblen Gewebeverhalten. Je feiner das Gitter diskretisiert ist desto weniger fallen die Unterschiede zwischen beiden Ansätzen auf.

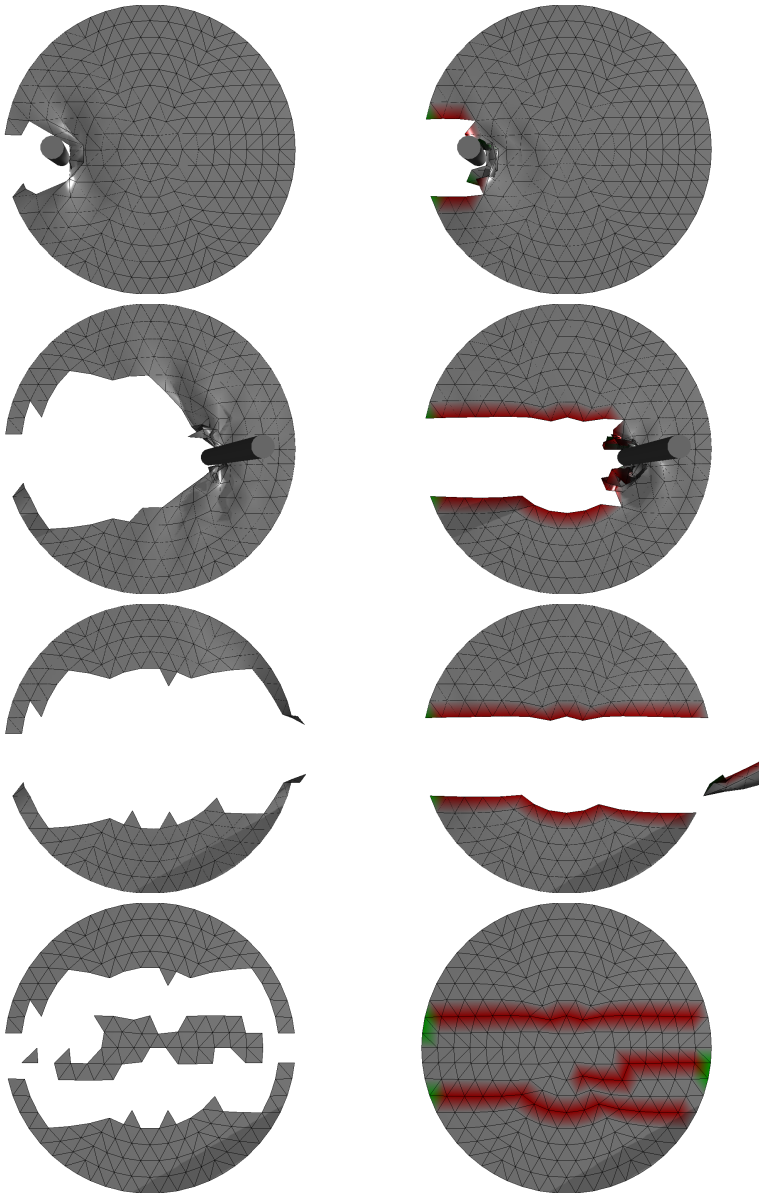

**Abbildung 3.10:** Vergleich beider Reißansätze auf einem groben Gitter (272 Knoten, 758 Federn und 486 Dreiecke). Reißen durch Löschen ist in der linken Spalte dargestellt. Die rechte Spalte zeigt den flächenerhaltenden Ansatz. Die Bilderserie zeigt von oben nach unten die Reaktion des Modells auf die Bewegung eines Zylinders durch das Gitter. Das unterste Bild zeigt jeweils das Gitter nach Beendigung der Interaktion, wobei alle Knoten auf ihre Ausgangsposition zurückgesetzt wurden. Die Verminderung der Anzahl der Dreiecke bei der Modellierung des Reißens durch Löschen führt zu einer starken Reduktion des Materials. Zur Veranschaulichung des flächenerhaltenden Reißvorgangs werden die Rissknoten je nach Knotentyp eingefärbt. Innere Knoten werden blau, Risskeime rot und andere Kantenknoten grün koloriert.

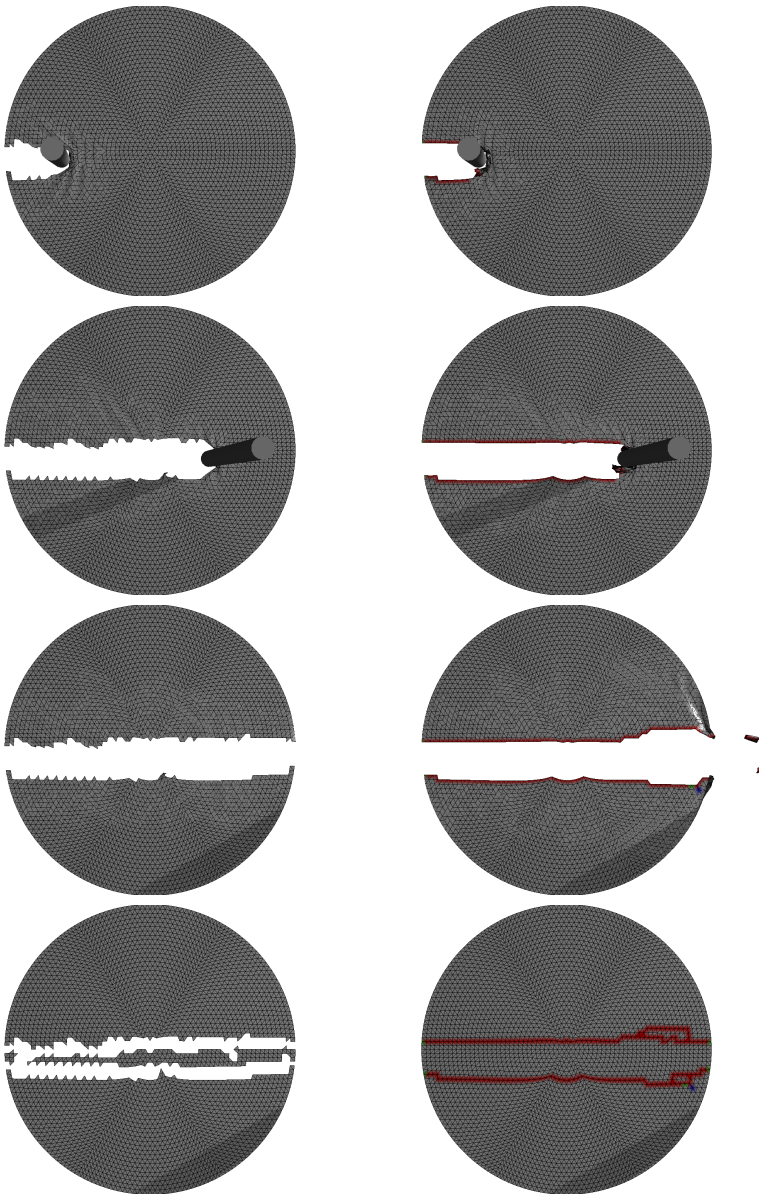

**Abbildung 3.11:** Die gleichen Bilderserien wie bei Abbildung 3.10 mit einem Gitter mittlerer Auflösung (3367 Knoten, 9900 Federn und 6534 Dreiecke). Die Verminderung des Materials ist deutlich sichtbar, aber nicht so gravierend wie im Fall des groben Gitters.

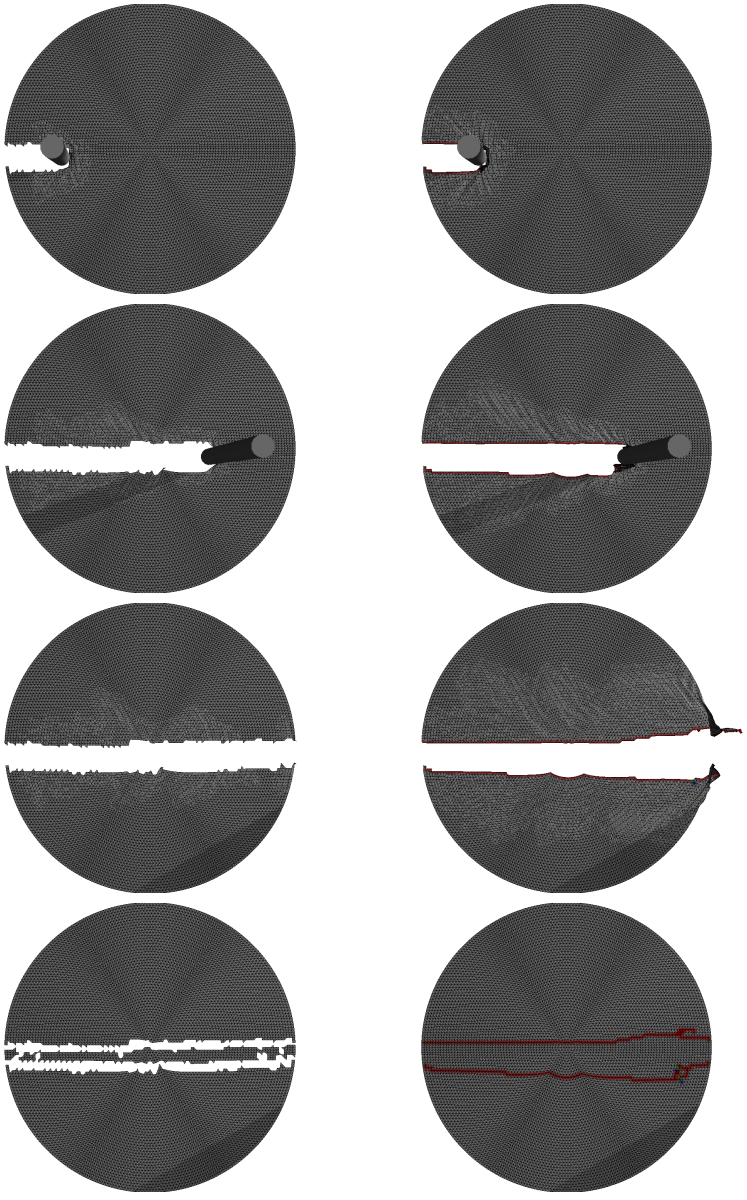

**Abbildung 3.12:** Die gleichen Bilderserien wie bei den vorherigen zwei Abbildungen auf einem sehr feinen Gitter (8587 Knoten, 25440 Federn und 16854 Dreiecke). Die Unterschiede zwischen beiden Ansätzen sind gering.

## 3.4 Zusammenfassung

In diesem Kapitel wurde ein Algorithmus vorgestellt, der das Reißen einer Membran simuliert. Der Algorithmus arbeitet auf einem Gitter aus Knoten, Dreiecken und Federn. Als Reiskriterium wird die Knotenspannung herangezogen. Verschiedene Schwellwerte je nach Lage des Knotens ermöglichen es, die Häufigkeit von Rissentstehung, Rissverzeigung und Rissausbreitung einzeln zu kontrollieren. Dadurch lässt sich das Reißverhalten an verschieden Materialen anpassen. Exemplarisch wird gezeigt, dass der Algorithmus auch zur Modellierung von spröden Materialien eingesetzt werden kann. Die topologischen Änderungen werden durch die Trennung der Verbindung zwischen zwei Dreiecken modelliert. Dadurch bleiben Anzahl der Dreiecke und somit die Membranfläche konstant. Der vorgestellte Algorithmus wird mit einem Ansatz verglichen, bei dem das Reißen durch Löschen überdehnter Federn modelliert wird.

Vor allem bei groben Gittern liefert der vorgestellte Ansatz ein deutlich realistischeres Gewebeverhalten, da beim Reißen die Anzahl de Dreiecke konstant bleibt. Mit dem Ansatz, der mit dem Löschen überdehnter Federn arbeitet, können außerdem keine spröden Materialen modelliert werden. Auf der anderen Seite ist der flächenerhaltende Ansatz langsamer. Bei der Modellierung von steifen Membranen spielt dies aber eine untergeordnete Rolle, da die Deformationsberechnungen den überwiegenden Teil der Rechenzeit benötigen.

Das flächenerhaltende Reißen wurde unter Grimm [2005b] veröffentlicht.

# 4

# Software-Architektur

„Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases."

—Norman R. Augustine [1]

Kapitel 4 beschreibt die Implementierung einer Simulationsbibliothek, die im Rahmen dieser Arbeit entwickelt wurde. Die Bibliothek gliedert sich in die so genannte *VRM-Architektur* ein. VRM steht für *Virtual Reality in Medicine*. Die Architektur wird in der Arbeitsgruppe *ViPA* (*Virtual Imageprocessing and Patient Analysis*) des Lehrstuhls für Informatik V entwickelt. Programmiert wind in C++ unter *Windows* und *Linux*. Das Framework bietet neben den hier behandelten Simulationen auch Module für Visualisierung, Tracking, Kollisionserkennung und andere Komponenten, die für VR-Anwendungen nötig sind. Wagner [2003] gibt einen ausführlichen Überblick über die VRM-Architektur.

Nach dem Stand der Technik in Abschnitt 4.1 werden die Anforderungen an die Simulationsbibliothek beschrieben (Abschnitt 4.2). Anschließend wird die Architektur der Simulationsbibliothek (Abschnitt 4.3) erklärt und auf die Repräsentation der Simulationsdaten eingegangen (Abschnitt 4.4). Dabei wird zwischen der Repräsentation der Daten in einer Graphstruktur (Abschnitt 4.4.1) und einer Array-Struktur (Abschnitt 4.4.1) unterschieden. Abschnitt 4.5 gibt einen Überblick über die implementierten Klassen und Simulationsmethoden. In Abschnitt 4.6 werden die Ergebnisse des gewählten Ansatzes diskutiert. Um den Vorteil der dualen Repräsentation der Daten in Array und Graphstruktur aufzuzeigen, wird die Integrationsgeschwindigkeit der beiden Ansätze verglichen.

[1] Amerikanischer Geschäftsmann und Autor, geb. 1935

## 4.1 Stand der Technik

Dem Autor sind zwei weitere VR-Bibliotheken bekannt die Algorithmen zur Simulation von biologischem Gewebe zu Verfügung stellen.

**Spring** Die von Montgomery, Bruyns, Brown, Sorkin, Mazzella, Thonier, Tellier, Lerman und Menon [2002] beschriebene *Spring* Bibliothek wurde am *National Biocomputation Center* in Stanford entwickelt. Als Simulationen stehen Rigid-Body und Feder-Masse-Modelle zur Verfügung. Implementierte Integrationsmethoden sind die explizite Euler-Integration, der quasistatische Euler-Ansatz und das Runge-Kutta-Verfahren 2. und 4. Ordnung. Eine ganze Reihe von VR-Simulatoren arbeitet mit dem Spring-Framework. Brown, Montgomery, Latombe und Stephanides [2001b] beschreiben eine auf Spring basierende VR-Anwendung, in der Blutgefäße genäht werden können. Bruyns, Montgomery und Wildermuth [2001] verwenden Spring als Grundlage einer virtuellen Sezierung von Ratten. Wildermuth, Bruyns, Montgomery, Beedu und Marincek [2001] entwickeln einen Koloskopie-Simulator auf Basis von Spring. Der Hysteroskopie-Simulator von Montgomery, Heinrichs, Bruyns, Wildermuth, Hasser, Ozenne und Bailey [2001] ist ebenfalls auf Spring aufgebaut.

Nach der Designphilosophie von SPRING berechnen zunächst alle Federn die von ihnen verursachten Knotenkräfte. Anschließend bestimmen alle Knoten ihre neuen Positionen. Der objektorientierte Ansatz führt zu einer guten Parallelisierbarkeit der Berechnungen. Durch die Auslagerung der Berechnungen auf die einzelnen Objekte ist SPRING hingegen kaum für implizite Integrationsverfahren oder Finiten-Elemente-Simulation geeignet, bei denen die Berechnungen in einer übergeordneten Klasse stattfinden.

**KISMET** Am *Forschungszentrum Karlsruhe* wurde die Bibliothek *KISMET* für SGI und INTERGRAPH Workstations unter IRIX und Windows NT entwickelt (siehe Kühnapfel, Krumm, Kuhn, Hübner und Neisius [1995]). Als Simulationsverfahren sind Feder-Masse-Modelle und FEM implementiert. Çakmak und Kühnapfel [2000] beschreiben außerdem die Animation von Blutungen, Pulsschlag und Rauch auf der KISMET-Architektur. Auf Basis von KISMET wurden ein Endoskopie-Simulator (siehe Kühnapfel, Cakmak und Maass [2000]) entwickelt. Çakmak, Maasz, Strauss, Trantakis, Nowatius und Kühnapfel [2002] beschreibt außerdem Simulationsmodelle für die Gynäkologie, eine Cholezystektomie, eine Ventrikulo-Zisternomie sowie eine Felsenbein-OP.

Nachteil von KISMET ist die Beschränkung auf Workstations. Außerdem scheint die Software nicht mehr gepflegt zu werden. Die zum gegenwärtigen Zeitpunkt aktuelle Version 6.0.3 stammt vom 22.07.1999.

**SOFA** Die *Simulation Open Framework Architecture* kurz *SOFA* (siehe Cotin [2005]) ist ein Projekt der *ALCOVE* Gruppe von *INRIA* in Lille, Frankreich und der *CIMIT*-Gruppe am *Massachusetts Institute of Technology* in Boston. Ziel des Projektes ist die Erstellung einer plattformunabhängigen, erweiterbaren, Open-Source Architektur für medizinische Trainingsimulatoren. Zum Zeitpunkt der Fertigstellung dieser Arbeit befindet sich das SOFA-Projekt noch in einem sehr frühen Stadium. Einfache Mass-Spring Simulationen und lineare Finite-Elemente-Simulationen lassen sich mit dem Framework bereits durchführen. Geplant war die Integration einer Chain-Mail-, einer Tensor-Mass- und einer nichtlinearer Finiter-Elemente-Simulation in das Framework.

## 4.2 Anforderungen

Bei der Entwicklung einer Architektur zur Simulation von Gewebe sind zwei diametrale Zielsetzungen zu berücksichtigen: Geschwindigkeit und Flexibilität.

Große Datenmengen müssen schnell und effizient verarbeitet werden. Die Strukturen müssen flexibel sein, um universell einsetzbar und anpassbar zu sein.

Folgende Anforderungen stellen sich an die Entwicklung der Simulationsbibliothek.

- möglichst schnelle Verarbeitung großer Datenmengen während eines Simulationsschrittes
- topologische Änderungen des Simulationsgitters müssen während der Laufzeit effizient durchführbar sein
- Anpassbarkeit an verschiede Problemklassen: z.B. Echtzeitsimulation oder möglichst genaue Simulationen
- Verschiedene Simulationsalgorithmen sollen auf die gleichen Daten angewandt werden können.
- Kombination verschiedener Simulationsmethoden soll problemlos möglich sein.
- Bei Beschränkung auf einen oder wenige Simulationsmethoden soll es keinen Overhead im Zeitverhalten oder der Speicherbelegung geben
- Für Anwendungsprogrammierer soll es eine möglichst einfache Schnittstelle geben.
- Integration in die VRM-Architektur; Zusammenarbeit mit Bibliotheken für Visualisierung, Kollisionserkennung

## 4.3 Architektur

In der VRM-Architektur werden drei wesentliche Schichten unterschieden (siehe Abbildung 4.1), die sich auch im Simulationsmodul widerspiegeln. Der Aufbau lässt mit der Analogie eines Baukasten-Systems verdeutlichen.

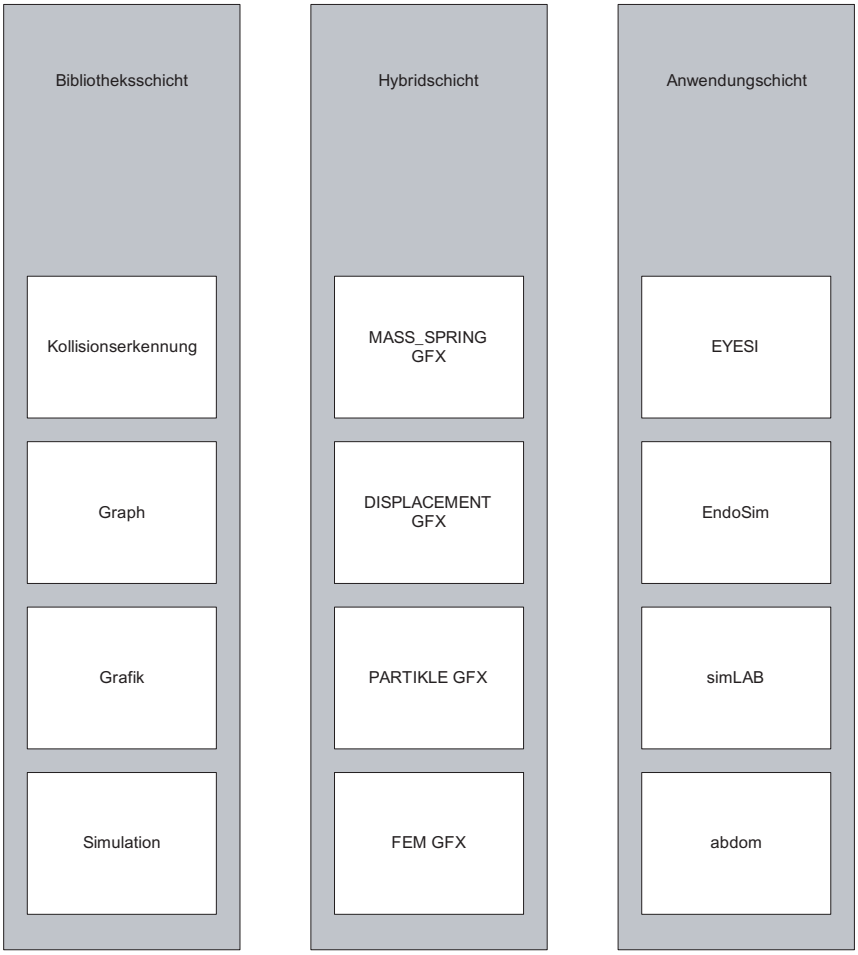


**Abbildung 4.1:** Das Konzept der VRM-Architektur unterscheidet zwischen drei Schichten. In der Bibliotheksschicht steckt die Funktionalität. In der Hybridschicht werden die für eine Aufgabe benötigte Teile der Bibliotheksschicht kombiniert und die Schnittstelle für den Anwendungsentwickler bereitgestellt. Die Anwendungsschicht füllt die Strukturen mit Daten und implementiert eine Benutzerschnittstelle. In der Grafik sind nur die für Simulationen wichtigen Bereiche der VRM-Architektur angedeutet.

**Die Bibliotheksschicht** Die unterste Ebene, die so genannte *Bibliotheksschicht* entspricht einer Sammlung vieler unterschiedlicher atomarer Bausteine, die nach ihrer Funktion in Kisten einsortiert sind. Konkret beinhaltet die Bibliotheksschicht die einzelnen Bibliotheken für Simulation, Visualisierung, Kollisionserkennung, usw. In den Bibliotheken sind die funktionalen Algorithmen und Daten definiert. Beispiele sind eine Runge-Kutta-Integration in Klasse *VsimBase* oder die Erstellung einer Objektsteifigkeitsmatrix in *VfemSimProp*. Die Algorithmen arbeiten mit generischen Objekttypen, die an dieser Ebene noch nicht deklariert sind. Von den Objekten sind auf der Bibliotheksebene die Eigenschaften so genannte *Props* (engl. properties) definiert. So sind in der Klasse *VsimNodeProp* die Knoteneigenschaften (Masse, Position, Geschwindigkeit, usw.) für einen allgemeinen Simulationsknoten definiert. Die Definitionen in den Props beschränken sich auf spezielle Simulationseigenschaften. Allgemeine Eigenschaften, die auch von anderen Bibliotheken benötigt werden, sind auf dieser Ebene nicht definiert. Beispielsweise werden die Eigenschaften und Methoden, die einen Knoten innerhalb einer Graphstruktur kennzeichnen, auf Ebene der Graph-Bibliothek in der Klasse *VgrMGE* definiert. Die Klassen in der Bibliotheksschicht sind einzelne Bausteine, aus denen eine Simulation zusammengesetzt wird. Sie können aber nicht direkt angewendet werden.

**Die Hybridschicht** Die anwendbaren Objekte werden in der mittleren *hybriden Schicht* zusammengestellt. Hier werden die benötigten Props aus Simulation und anderen Bibliotheken kombiniert. Ein Knoten in einer Finiten-Elemente-Simulation, der auch visualisiert werden soll (*VfemgfxNode*), ist beispielsweise von *VgrMGE* (Knoten in einer Graphstruktur), *VsimNodeProp* (Simulationseigenschaften), *VfemNodeProp* (zusätzliche Eigenschaften für FEM) und *VgfxNodeProp* (Eigenschaften für die Visualisierung) abgeleitet. Abbildung 4.2 zeigt, wie sich der *VfemgfxNode* zusammensetzt. Objekte, der Bibliotheksschicht, die mit Objekten der Hybridschicht arbeiten, erhalten die nötigen Typinformationen durch Template-Spezialisierung. So bekommt die *VsimBase* Klasse als Template-Parameter den Typ *VfemgfxNode* übergeben und arbeitet mit diesem Knotentyp. Im Baukastenmodell werden in der Hybridschicht aus den vielen verschiedenen Bausteinen komplexe Bauteile für bestimmte Aufgaben zusammengestellt.

**Die Anwendungsschicht** Die *Anwendungsschicht* kombiniert die Bauteile der hybriden Schicht zu Werkzeugen, die von einem Benutzer verwendet werden können. In der Anwendungsschicht werden hybride Objekte instanziiert, mit Daten gefüllt und eine Benutzerschnittstelle hinzugefügt. Typischerweise entstehen aus den Modulen in der Hybridschicht eine Graph- und eine Datenstruktur sowie eine Simulationsklasse. Die Graphstruktur beinhaltet eine Knotenliste und in der Regel mehrere Listen mit Verbin-

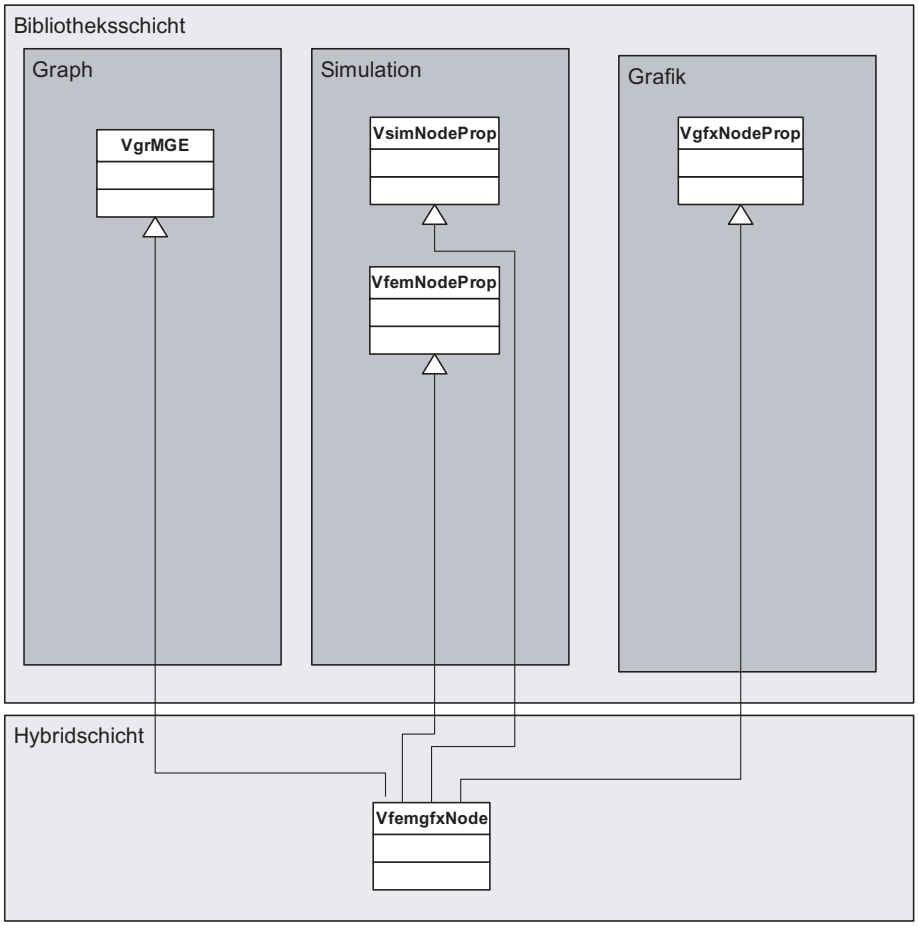


**Abbildung 4.2:** Die Abbildung zeigt vereinfacht, wie sich ein hybrider *Vfemgfx-Node* aus verschiedenen Modulen der Bibliotheksschicht aufbaut.

dungselementen wie Oberflächendreiecken, finiten Tetraederelementen oder kombinierten Feder-Dragnet-Verbindungen. Die Datenklasse beinhaltet alle nötigen Daten; in der Simulationsklasse sind die nötigen Algorithmen vereint. Mit diesen drei Klassen lassen sich dann die Anwendungen erstellen. Es entsteht eine Anwendung wie das *femLAB*[2], mit der Finiten-Elemente-Simulationen durchgeführt werden können. Abbildung 4.3 zeigt eine vereinfachte Klassenstruktur des femLAB.

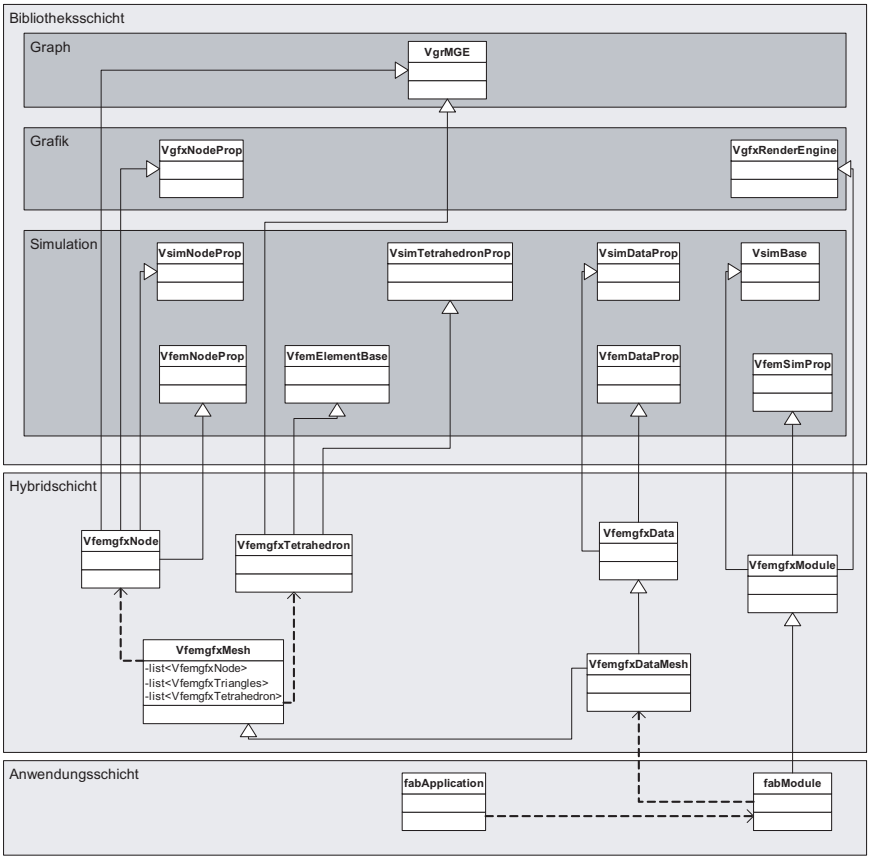


**Abbildung 4.3:** Vereinfachter Aufbau der femLAB-Anwendung. Das Basismodul der Simulation wird mit dem Finite-Elemente-Modul und Klassen aus anderen Bibliotheken kombiniert. In der Hybridschicht ergeben sich eine Graph- und Datenstruktur und eine Simulationsklasse. Auf Anwendungsebene wird mit den hybriden Klassen das fertige Programm erstellt.

Der Vorteil des Ansatzes liegt in der Flexibilität bei gleichzeitiger Scho-

[2]Der Name femLAB entstand durch Zufall und hat nichts mit dem kommerziellen Produkt FEMLAB zu tun.

nung von Ressourcen. Es entstehen schlanke Objekte, die nur den nötigten Speicherplatz belegen. Der Anwendungsentwickler sieht lediglich die für ihn wichtigen Eigenschaften und Methoden, wohingegen auf Hybridebene aus dem vollen Funktionsumfang der Bibliotheksschicht geschöpft werden kann.

## 4.4 Datenrepräsentation

Die meisten der in Kapitel 2 beschriebenen Simulationen basieren auf einem Simulationsgitter. Die Geometrie des simulierten Objekts wird durch eine Reihe von Knotenpunkten beschrieben. Die Knotenpunkte sind durch Verbindungen wie Federn oder Finite-Elemente miteinander verknüpft.

Überträgt man diese Struktur in einen objektorientierten Ansatz, werden Knoten und Verbindungen durch Klassen beschrieben, die Elemente einer Graphstruktur sind. Ein Verbindungselement (z.B. ein Dreieck) enthält Verweise auf seine Knoten. Die Knoten haben ihrerseits Verweise auf die Elemente, zu denen sie gehören. Die Verweise sind die Kanten eines Graphen. Die Graphstruktur macht es möglich Zugehörigkeiten und Nachbarschaftsbeziehungen sehr effizient auszuwerten. Veränderungen der Gitterstruktur lassen sich durch Änderung der Verweise ohne großen Aufwand abbilden.

Während einer Simulation werden die Positionen und andere Eigenschaften der Knoten häufig sequenziell abgearbeitet, um beispielsweise einen Integrationsschritt auszuführen. Um das Abarbeiten effizient zu gestalten, ist es sinnvoll, die Daten ebenfalls sequenziell in Arrays abzulegen.

Dadurch wird der Zugriff beschleunigt, da das Springen zur Position des nächsten Knotens nur der Inkrementierung eines Zeigers um eine Speicherstelle entspricht. In der Graphstruktur muss man zum nächsten Knoten springen und ihn nach einer Position fragen.

Neben der Minimierung der Anzahl von Instruktionen resultiert der Geschwindigkeitsvorteil bei einem Arrayzugriff vor allem auf dem *Caching* des Prozessors. Beim Caching wird ein Bereich des Arbeitsspeichers auf Verdacht in einen prozessornäheren und schnelleren Speicher geladen, um bei Bedarf schneller zugreifbar zu sein. Ohne auf das komplexe Caching moderner Prozessoren einzugehen, sei bemerkt, dass das Caching beim sequenziellen Auslesen von Daten effektiver funktioniert als bei zufälligen (*random access*) Zugriffen.

Die entwickelte Simulationsarchitektur verwendet einen hybriden Ansatz aus Graph und Array Repräsentation der Daten.

### 4.4.1 Die Graphstruktur

Die Graphstruktur wurde gewählt, um folgende Anforderungen zu erfüllen.

- Schneller Zugriff auf alle Verbindungselemente eines Knotens.
- Schneller Zugriff auf die Knoten eines Verbindungselements.
- Schnelle Bestimmung der Nachbarknoten eines Knotens.
- Schnelle Bestimmung der Nachbarverbindungen eines Verbindungselements.

**MGE-Struktur** Grundlage der Graphstruktur ist das in der ViPA Arbeitsgruppe entwickelte *Multigraph Element* (*MGE*), das Wagner in seiner Dissertation [2003] ausführlich vorstellt.

Das MGE besteht aus einer Reihe von Listen, in denen Verweise auf andere MGEs gespeichert werden. Die einzelnen Listen repräsentieren Verweisarten und werden über konstante Ganzzahlen (IDs) identifiziert.

Ist $M(M_1, I_1)$ die Menge aller MGEs, auf die MGE $M_1$ mit der Verweisart $I_1$ verweist dann gilt:

- $M_1 \in M(M_2, I_1) \Longleftrightarrow M_2 \in M(M_1,\text{ref}(I_1))$

Wobei ref($x$) mit $x \in Z$ zwei Eigenschaften erfüllt:

1. $\text{ref}(x) \neq \text{ref}(y) \quad \forall \quad x \neq y$
2. $\text{ref}(\text{ref}(x)) = x \quad \forall \quad x$

Zu jedem Verweis eines $M_1$ auf $M_2$ mit ID $I_1$ existiert also immer ein Verweis von $M_2$ auf $M_1$ mit ID $ref(I_1)$. Die implementierte MGE-Struktur stellt sicher, dass beim Hinzufügen und Entfernen von Verweisen und beim Löschen von MGEs der Graph immer in einem konsistenten Zustand bleibt.

**Spezialisierung der MGE-Struktur** In einem Simulationsgitter werden Knoten und Simulationselemente durch MGEs repräsentiert. Dadurch sind Zugriffe von den Knoten auf die Simulationselemente und umgekehrt möglich. Die so entstandene Symmetrie zwischen Knoten und Verbindungen wird auf der Ebene der Simulationsgitter wieder gebrochen. Die Anzahl der Simulationselemente, die mit einem Knoten verbunden sind, ist variabel und kann sich während einer Simulation ändern. Jedes Simulationselement ist hingegen immer mit einer festen Anzahl von Knoten verbunden. Eine Feder ist beispielsweise zwischen zwei Knoten aufgehängt und ein Tetraederelement hat genau vier Eckpunkte. Um einen konsistenten Zustand eines Simulationsgitters zu garantieren, zieht das Löschen eines Knotens immer

das Löschen aller seiner Simulationselemente mit sich. Das Löschen von Simulationselementen führt hingegen nicht zu einer Zerstörung seiner Knoten. Beim Anlegen neuer Simulationselemente müssen die verbundenen Knoten im Konstruktor angegeben werden. Dadurch ist sichergestellt, dass sich das Simulationsmesh immer in einem konsistenten Zustand befindet.

**Der Zeitstempel** Bei vielen Algorithmen, die auf einem Graph basieren, müssen abgearbeitete Elemente markiert werden. Sucht man beispielsweise die Nachbarn zu einem Knoten $n_i$, kann man alle Dreiecke des Knotens durchgehen und die Knoten aller Dreiecke in eine Liste schreiben. Dabei muss man beachten, dass Knoten nicht doppelt in die Liste aufgenommen werden.

Um solche und ähnliche Probleme zu beheben, wurde für das Simulationsmodul das Konzept eines *Zeitstempels* (engl. *timestamp*) in die MGE-Struktur eingefügt. Die *VgrMGE*-Klasse besitzt einen statischen und einen nicht-statischen Zeitstempel. Der statische Zeitstempel gilt global für alle MGEs und kann mit einer statischen Methode *VgrMGE::incTimestamp()* erhöht werden. Der nicht-statische Zeitstempel ist Attribut jeder MGE-Instanz. Die instanzspezifische Methode *setActualTimestamp()* setzen den nicht-statischen Zeitstempel einer MGE-Instanz auf den statischen Wert. Mit *isTimestampActual()* wird überprüft, ob der nicht-statische Zeitstempel eines MGEs den aktuellen Wert des statischen Zeitstempels hat.

Algorithmus 4.1 zeigt, wie der Zeitstempel eingesetzt werden kann, um die Nachbarn eines Knotens $n_i$ in einer MGE-Graphstruktur zu suchen.

**Algorithmus 4.1** FindNeighbors($n_i$)

```
1: std::list<VgrMGE>list
2: VgrMGE::incTimestamp()
3: for all t_i ∈ t(n_i) do
4:    for all n_j ∈ n(t_i) do
5:       if NOT n_j.isTimestampActual() then
6:          n_j.setActualTimestamp()
7:          list.add(n_j)
8: return list
```

Prinzipiell wäre es möglich, den Zeitstempel durch ein Flag zu ersetzen. Das Flag müsste nach jeder Verwendung in allen Knoten gelöscht werden, was zeitaufwendig und fehleranfällig ist. Mit dem Konzept des Zeitstempels wurden unter anderem Dragnet, Chain-Mail und die von Brown et. al. [2001a] vorgeschlagene interaktionspunkt-basierte Integration implementiert.

### 4.4.2 Die Arraystruktur

Für die Simulation wurde eine Arraystruktur implementiert, die folgende Anforderungen erfüllt.

- Der Zugriff auf die Knoteneigenschaften über die Knotenklasse soll weiterhin ungeändert möglich sein, damit Operationen auf dem Graph weiterhin durchgeführt werden können.
- Für den schnellen Zugriff auf die Knoteneigenschaften z.B. bei den Integrationsschritten sollen die Eigenschaften lückenlos nacheinander abgelegt werden.
- Entfernen und Hinzufügen von Knoten soll schnell und effizient möglich sein.

Um die Forderung nach einem gleich bleibenden Zugriff auf die Knoteneigenschaften zu erfüllen, kann ein Knoten auf zweierlei Arten speichern. Speichert ein Knoten seine Eigenschafen intern, verwaltet er selbst den benötigten Speicherplatz. Bei externer Speicherung bekommt der Knoten Zeiger auf die Speicherstellen seiner Eigenschaften übergeben. Der äußere Zugriff auf die Eigenschaften eines Knoten ändert sich dabei nicht.

Die Verwaltung des Arrays übernimmt eine übergeordnete Klasse, die die nachfolgenden Operationen unterstützt.

**Anlegen des Arrays** Beim Anlegen des Arrays werden zusammenhängende Speicherblöcke für die Eigenschaften der Simulationsknoten und gegebenenfalls einer gewissen Anzahl zusätzlicher Knoten angelegt. Der zusätzliche Speicherplatz wird für neu hinzukommende Knoten reserviert. Eine spätere Vergrößerung des angelegten Speicherplatzes ist nicht vorgesehen, um eine zeitintensive Verschiebung der Knoteneigenschaften zu vermeiden. Nach dem Reservieren des Speichers werden die Positionen in den Arrays von vorne beginnend den Knoten zugewiesen. Wurde Speicher für mehr Knoten reserviert als aktuell in der Simulation vorhanden sind, bleibt Speicher am Ende der Blöcke frei. Anschließend speichert jeder Knoten seine Eigenschaften in den Arrays ab und merkt sich die jeweiligen Positionen.

**Hinzufügen eines Knotens** Wird ein Knoten hinzugefügt, legt er seine Eigenschaften an der ersten freien Stelle der Speicherblöcke ab. Übersteigt die Anzahl der vorhandenen Knoten die Anzahl der Knoten, für die Speicherplatz reserviert wurde, tritt ein Fehler auf.

**Entfernen eines Knotens** Wird ein Knoten aus der Simulation entfernt, wird er angewiesen, seine Eigenschaften intern zu speichern. Um die entstehende Lücke innerhalb der Arrays zu schließen, werden die Eigenschaften des letzten Knotens an die frei werdende Stelle verschoben und die Zeiger auf seine Eigenschaften an die neuen Positionen angepasst. Die alte Speicherstelle des Knotens ist dann für neue Knoten verfügbar.

**Auflösen des Arrays** Beim Auflösen des Arrays werden alle Knoten angewiesen, ihre Eigenschaften intern zu speichern. Danach werden alle reservierten Speicherblöcke freigegeben.

## 4.5 Die implementierten Simulationsklassen

Nachfolgend wird ein Überblick über die Klassenstruktur innerhalb der Simulationsbibliothek gegeben. Die Auflistung der Klassen dient dazu, die Strukturen innerhalb der Simulationsbibliothek zu veranschaulichen. Sie ist keine vollständige Dokumentation, sondern eine Anlaufstelle für einen ersten Kontakt mit der Bibliothek. Die Liste erhebt keinen Anspruch auf Vollständigkeit.

Die Simulationsbibliothek gliedert sich in einen allgemeinen Basisteil und mehrere spezielle Module.

**Basisklassen der Simulation** Die *Vsim*-Klassen bilden die Basis für alle Simulationen. *VsimNodeProp* beinhaltet Knoteneigenschaften, die für alle Arten von Simulationen benötigt werden. *VsimLink* ist die Basis für Springs, Chains, Dragnet Strings und finite Linienelemente. Auf *VsimTriangle* und *VsimTetrahedron* basieren die jeweiligen finiten Elemente. Zusätzliche Simulationsparameter wie der verwendete Zeitschritt und die Arrays mit den Knoteneigenschaften werden in *VsimDataProp* gehalten. *VsimBase* stellt eine Reihe von Integrationsalgorithmen zur Verfügung. Momentan implementiert sind Euler, quasi-statischer Euler, Runge-Kutta 2. und 4. Ordnung und das implizite Houbolt-Verfahren.

**Displacement** Das *Displacement*-Modul beherbergt die Klassen für verschiebungsorientierte Simulationen. Die Verbindungselemente für Chain-Mail, die Längenkorrektur von Federn und Dragnet sind *VdisChainProp*, *VdisLinkProp* und *VdisStringProp*. Die zugehörigen Algorithmen werden in *VdisChainMailProp*, *VdisLinkSimProp* und *VdisDragnetProp* bereitgestellt.

**Rigid-Body** Die Eigenschaften und Methoden für Starre-Körper-Simulationen sind unter *Rigid-Body* implementiert. Die benötigten Daten werden in *VrbDataProp* gespeichert. Die Simulationsalgorithmen stehen in *VrbSimProp*, wobei die Klasse für die Integration auf *VsimBase* zurückgreift.

**Mass-Spring** Die Federeigenschaften bei einer Mass-Spring-Simulation sind in der Klasse *VmsSpringProp* definiert. In *VmsSimProp* sind die interaktionsorientierten Integrationen aus Abschnitt 2.1.3 implementiert. Diese können nicht auf *VsimBase*-Ebenen implementiert werden, da sie direkt auf einem Mass-Spring Gitter arbeiten.

**Tearing** Die Reißalgorithmen sind im Modul *Tearing* zusammengefasst. Das Membranreißen ist in der Klasse *VtearManager* implementiert. *VtearNodeProp* beinhaltet die knotenspezifischen Reißeigenschaften und -parameter.

**FEM** Finite-Elemente-Simulation werden mit dem *FEM*-Modul realisiert, globale Matrizen und andere benötigten Daten werden in *VfemDataProp* verwaltet. *VfemSimProp* beinhaltet die Simulationsalgorithmen. Die Finiten-Elemente *VfemLine*, *VfemTriangle* und *VfemTetrahedron* basieren auf *VfemElementBase*.

Tabelle 4.1 gibt einen Überblick über die Klassen der Simulationsbibliothek.

## 4.6 Ergebnisse

Um den Nutzen der hybriden Repräsentation der Daten als Graph und Array zu beurteilen, werden im Folgenden die Laufzeiten einer Euler-Integration auf der Graph- und auf der Arraystruktur gemessen und miteinander verglichen.

Die Laufzeitmessungen wurden auf einem Pentium 4 mit HyperThreading System mit 2.8 GHz unter Windows XP durchgeführt. Die Anwendung lief mit Echtzeit-Priorität an einen virtuellen Prozessor gebunden. Alle anderen Programme einschließlich der Benutzeroberfläche wurden geschlossen und alle unnötigen Dienste angehalten, um Beeinflussungen der Messungen zu minimieren.

Tabelle 4.2 listet die gemessenen Laufzeiten auf. Im Diagramm 4.4 wird die Anhängigkeit der Laufzeit von der Knotenzahl veranschaulicht.

Wie aus Tabelle 4.2 ersichtlich ist die Integration auf einem Array für kleine Simulationsgitter deutlich überlegen. Die Integration über der Graphstruktur benötigt bis zur achtfachen Zeit.

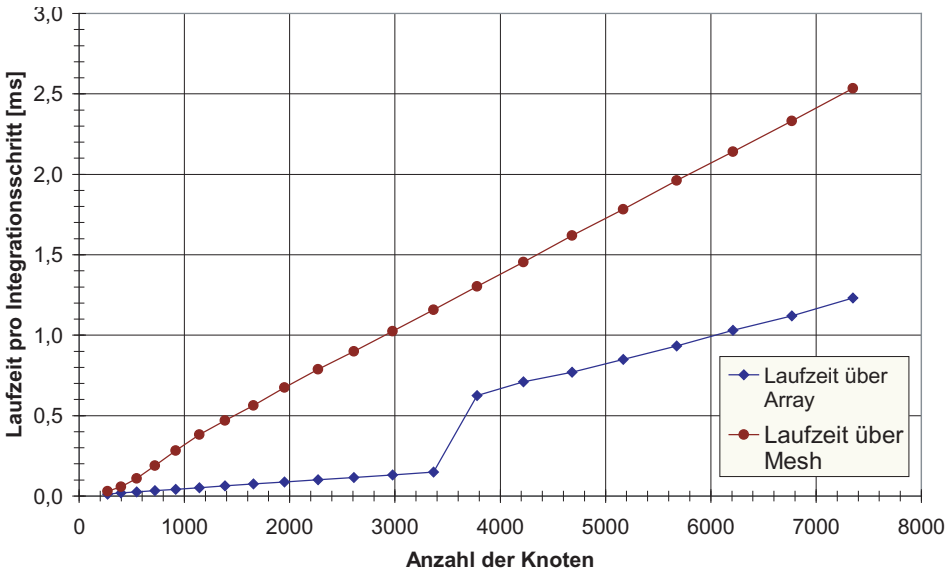


**Abbildung 4.4:** In dem Diagramm lässt sich die Ausnutzung des Prozessor Caches bei der Integration auf dem Array erkennen. Bei Gittergrößen bis ca. 3500 Knoten ist die Integration über der Arraystruktur deutlich schneller, als bei der Integration über der Graphstruktur. Die Laufzeit steigt mit zunehmender Gittergröße nur langsam an. Bei größeren Gittern bricht der Vorsprung sprunghaft ein, was darauf schließen lässt, dass bei diesen Datenmengen das Caching nicht mehr optimal funktioniert.

Diagramm 4.2 zeigt, dass bis zu ca. 3500 Knoten die benötigte Reichenzeit für die Integration über der Arraystruktur nur langsam mit der Gittergröße ansteigt. Die Laufzeit der Integration über der Graphstruktur steigt deutlich schneller an. Bei ca. 3500 Knoten macht die Laufzeit der Array-Integration einen Sprung. Ab dieser Gittergröße steigt die Laufzeit für die Integration über der Arraystruktur deutlich schneller an als bei kleineren Gittern. Sie bleibt aber immer noch unter der Laufzeit der Integration über der Graphstruktur und steigt auch langsamer an als diese. Die Ergebnisse deuten darauf hin, dass ab Gittern von ca. 3500 Knoten die Datenmenge, die abgearbeitet werden muss so groß wird, dass Engpässe im Datenzugriff entstehen die nicht mehr mit dem Caching des Prozessors ausgeglichen werden können.

## 4.7 Zusammenfassung

In diesem Kapitel wurde die Implementierung der Simulationsbibliothek beschrieben. Die Simulationsbibliothek ist Teil einer VR-Architektur, die entwickelt wurde, um medizinische Trainingsimulatoren zu entwickeln. Knoten und Verbindungselemente basieren auf einem Graph-Element. Die Graph-Struktur ist so implementiert, dass sich das Simulationsgitter immer in einem konsistenten Zustand befindet. Die für einen bestimmten Simulationsansatz nötigen Eigenschaften erben die Elemente und Modulklassen von so genannten Props. Dadurch ist eine flexible Kombination von verschiedenen Simulationsarten möglich. Die Eigenschaften der Simulationsknoten können auch in zusammenhängenden Arrays gespeichert werden.

Die Array-Struktur führt zu einem signifikanten Geschwindigkeitsgewinn, wodurch sich der numerische Fehler deutlich reduzieren lässt.

Die Bibliothek ist Grundlage der biomechanischen Modellierung in den VR-Simulatoren EYESI-*Vitreoretinal*, EYESI-*Cataract* und EndoSim.

Die Bibliothek wird ständig erweitert. Gegenwärtig wird darüber nachgedacht die Bibliothek so zu erweitern, dass Simulation auf *GPU* (*Graphics Processing Unit*) oder *FPGA*s *Field-Programmable Gate Array*) ausgelagert werden können. Die Möglichkeit der massiven parallelen Verarbeitung auf diesen Prozessoren soll die Algorithmen weiter beschleunigen. Außerdem sollen neue Ansätze wie die Gitterlosen Verfahren in die Bibliothek aufgenommen werden.

| Simulationsart | Knoten-Eigenschaften | Element-Eigenschaften | Simulationsklasse | Daten-Manager |
|---|---|---|---|---|
| allgemein | VsimNodeProp | VsimLinkProp, VsimTriangleProp, VsimTetrahedronProp | VsimSimBase | VsimDataProp |
| Rigid-Body | — | — | VrbSimProp | VrbDataProp |
| Particle | VpartNodeProp | — | — | — |
| Mass-Spring | VmsNodeProp | VmsSpringProp | VmsSimProp | — |
| Finite-Elemente-Methode | VfemNodeProp | VfemLineElement, VfemTriangleElement, VfemTerahedronElement | VfemSimProp | VfemDataProp |
| Chain-Mail | VdisNodeProp | VdisChainProp | VdisSimProp | — |
| Dragnet | VdisNodeProp | VdisStringProp | VdisSimProp | — |

**Tabelle 4.1:** Überblick über die wichtigsten implementierten Simulationsklassen.

| Anzahl der Knoten | Laufzeit über Array [ms] | Laufzeit über Graph [ms] | Graph durch Array-Laufzeit |
|---|---|---|---|
| 271 | 0,01 | 0,03 | 233% |
| 397 | 0,02 | 0,06 | 303% |
| 547 | 0,03 | 0,11 | 424% |
| 721 | 0,03 | 0,19 | 562% |
| 919 | 0,04 | 0,28 | 676% |
| 1141 | 0,05 | 0,38 | 725% |
| 1387 | 0,06 | 0,47 | 742% |
| 1657 | 0,08 | 0,56 | 742% |
| 1951 | 0,09 | 0,67 | 774% |
| 2269 | 0,10 | 0,79 | 783% |
| 2611 | 0,11 | 0,90 | 783% |
| 2977 | 0,13 | 1,03 | 778% |
| 3367 | 0,15 | 1,16 | 779% |
| 3781 | 0,62 | 1,30 | 209% |
| 4219 | 0,71 | 1,45 | 205% |
| 4681 | 0,77 | 1,62 | 210% |
| 5167 | 0,85 | 1,78 | 210% |
| 5677 | 0,93 | 1,96 | 210% |
| 6211 | 1,03 | 2,14 | 208% |
| 6769 | 1,12 | 2,33 | 208% |
| 7351 | 1,23 | 2,53 | 206% |
| 7957 | 1,32 | 2,75 | 208% |
| 8587 | 1,42 | 2,96 | 208% |
| 9241 | 1,53 | 3,18 | 207% |
| 9919 | 1,64 | 3,43 | 208% |
| 10621 | 2,26 | 3,67 | 163% |
| 11347 | 1,88 | 3,94 | 209% |
| 12097 | 2,02 | 4,15 | 206% |
| 12871 | 2,14 | 4,42 | 206% |
| 13669 | 2,27 | 4,69 | 207% |
| 14491 | 3,07 | 4,97 | 162% |
| 15337 | 2,54 | 5,25 | 206% |
| 16207 | 2,74 | 5,54 | 202% |
| 17101 | 2,83 | 5,84 | 206% |

**Tabelle 4.2:** Die Integration auf der Graphstruktur benötigt für kleine Simulationsgitter mit bis zu ca. 3500 Knoten bis fast achtmal so lange. Bei großen Gittern beträgt der Rückstand ca. 100%.