UNIVERSITÄT SIEGEN
Computergraphik und Multimediasysteme

Theorie
und Praxis
für Karrieren
von morgen

GPU-BASED VOLUME DEFORMATION USING MASS-SPRING-SYSTEMS

Diplomarbeit

Zur Erlangung des Grades eines Diplom-Informatikers im Studiengang
Angewandte Informatik mit Anwendungsfach Medienwissenschaften

vorgelegt von: Tobias Knoche

Erstgutachter: Dr. Christof Rezk-Salama
Fachgruppe für Computergrafik und Multimediasysteme

Zweitgutachter: Prof. Dr. Andreas Kolb
Fachgruppe für Computergrafik und Multimediasysteme

Siegen, im Februar 2008

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die vorliegende Diplomarbeit behandelt die direkte Deformation von dreidimensionalen Volumina auf der Grafikkarte in Echtzeit.

In diesem Rahmen wurde ein Softwaretool entwickelt, welches den Vergleich verschiedener Deformationssysteme untereinander ermöglichen soll. Mit dem physikalisch-basierten Masse-Feder-Modell sowie dem geometrie-basierten ChainMail-Modell wurden zwei Deformationssysteme implementiert, die in weiten Teilen unterschiedlichen Ansätze folgen.
Die visuelle Qualität des ChainMail-Modells ist nicht befriedigend. Daher wird nach der Deformation ein sogenannter Relaxation-Schritt durchgeführt. Dieser soll dazu dienen, die sichtbaren geometrischen Strukturen aufzulösen und somit die visuelle Qualität zu erhöhen.
Um einen Vergleich zwischen traditionellen Lösungen und hardware-unterstützten Lösungen zu ermöglichen wurde zudem jedes Deformationssystem in einer CPU-Lösung sowie einer GPU-Lösung implementiert. Bedingung für die Deformationsmodelle ist eine uniforme Verteilung einzelner Deformations-elemente in einem Würfel. Die Deformation der einzelnen Elemente wird in Form des Offsets zur Ursprungsdeformation in einer Textur gespeichert.
Der Offset wird in einem Pixelshader-Programm genutzt, um die Stelle der Abtastung des Volumens beim direkten Volume-Rendering zu beeinflussen. So kann eine Deformation des Volumens simuliert werden.

Die Hauptprobleme bei der Entwicklung dieses Verfahrens waren Realitätstreue der Materialien, Speicherkapazität der Grafikkarten sowie eine derzeit unausgereifte Shader-Technologie.
Einige Punkte konnten im Rahmen dieser Diplomarbeit wegen ihrer Komplexität nicht beleuchtet werden, wurden jedoch als Anknüpfpunkte erwähnt. Die Anbindung verschiedenster Deformationsmodelle ist theoretisch möglich. Der technologische Fortschritt der Grafikhardware wird in Zukunft vermutlich neue Möglichkeiten der Erweiterung aufkommen lassen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In unserer durch moderne Informations-Technologien geprägten Zeit nimmt die Repräsentation von Daten durch Zahlenwerte einen stetig wachsenden Stellenwert ein. Ab einer bestimmten Menge an Zahlen jedoch ist der Mensch damit überfordert, die tatsächliche Bedeutung dieser Zahlenwerte kognitiv zu erfassen. Es fällt schwer, den Überblick zu bewahren. Die Visualisierung als Disziplin der Computergrafik bietet für diese Problematik Lösungsansätze, indem sie die anschauliche grafische Aufbereitung komplexer Datensätze ermöglicht. Auf diese Art und Weise wird ein leichteres Verständnis komplizierter Sachverhalte bewirkt.

Visualisierungsverfahren werden in den verschiedensten Bereichen genutzt. In der Medizin zum Beispiel verwendet man die Visualisierung, um dreidimensionale *Magnet-Resonanz-Tomografie* (MRT)-Aufnahmen von Patienten darzustellen. Diese können u.a. dazu dienen, in der Mediziner-Ausbildung virtuelle Operationen durchzuführen. Unter Zuhilfenahme von Deformationsmodellen können komplexe Eingriffe erprobt werden. Der Vorteil ist offensichtlich: Indem man die dargestellten Daten für eine Simulation nutzbar macht, wird ein realer Patient für das Erlernen potentiell gefährlicher Eingriffe nicht mehr benötigt.

Die heutigen technischen Rahmenbedingungen eröffnen auf dem Gebiet der Volumendeformation neue Möglichkeiten. Veränderte PC-Hardware erlaubt außerdem komplexe Berechnungen unmittelbar auf der Grafikkarte.
Dieser Aspekt ist Gegenstand der vorliegenden Arbeit.

Die Umsetzung einer GPU-basierenden Volumendeformation ist das primäre Ziel, das mit dieser Arbeit verfolgt wird. Dabei soll sowohl die Berechnung der Deformation als auch die Visualisierung auf der Grafikkarte erfolgen. Die Visualisierung des deformierten Volumens soll – sofern möglich – ohne eine aufwändige Vorverarbeitung ausgeführt werden. Ferner sollen verschiedene Deformationsalgorithmen auf ihre GPU-Tauglichkeit hin untersucht und miteinander verglichen werden. Im Rahmen dieser Arbeit soll ein intuitiv zu bedienendes Softwaretool entwickelt werden, in dem die Lösungen der genannten Anforderungen umgesetzt werden.

1.1 Gliederung

Die Arbeit ist in einen einführenden Theorieteil, einen praktischen Teil sowie eine kritische Reflexion der Methoden und Ergebnisse gegliedert.
Im Theorieteil werden zunächst die erforderlichen Grundlagen aus dem Bereich der Grafik-Programmierung ausgeführt. Anschließend wird eine Einführung in die Volumenvisualisierung vorgenommen. Die Auswahl des Visualisierungs-Verfahrens für die zu entwickelnde Software soll anhand der theoretischen Ergebnisse getroffen werden.

In einem weiteren Kapitel werden zunächst die theoretischen Grundlagen für Deformationsmodelle erläutert. Darauf aufbauend werden dann verschiedene Deformationsmodelle vorgestellt und näher beleuchtet.
Der Theorieteil endet mit der Zusammenführung der theoretischen Ergebnisse, welche dann auf ihre Verwendbarkeit für die Volumendeformation analysiert werden. Auf dieser Grundlage wird dann beurteilt, welche Verfahrensweisen in der von mir entwickelten Software umgesetzt werden können.

Der sich anschließende Praxisteil soll die Entwicklung der Software dokumentieren, um anschließend eine differenzierte Analyse der Ergebnisse zu ermöglichen.
Es werden zunächst die Ziele für die zu entwickelnde Software definiert. Dann wird der strukturelle Aufbau des entwickelten Softwareproduktes vorgestellt. Daraufhin wird die Ausführung der Implementierung beschrieben.

Im letzten Kapitel werden die erarbeiteten Ergebnisse verglichen und ausgewertet. Zudem sollen Probleme, die sich bei der Entwicklung gezeigt haben, diskutiert werden.

In einem abschließenden Fazit wird eine Bilanz gezogen und ein Ausblick im Hinblick auf die zu erwartende Entwicklung formuliert.

2 Theorie

2.1 Einleitung

Die beiden für diese Arbeit zentralen Themen sind zum einen die Visualisierung von 3D-Volumina und zum anderen die Volumendeformation. Um zu begründeten Entscheidungen für meine praktische Arbeit zu gelangen, werden zunächst allgemeine Grundlagen sowie zwei konkurrierende Visualisierungsverfahren vorgestellt. Physikalisch-basierte und geometrie-basierte Deformationssysteme bilden einen weiteren Schwerpunkt des Theorieteils.

2.2 *Shader*

Primärziel dieser Diplomarbeit ist die Nutzung der leistungsstarken Grafik-Hardware für die Berechnung von Volumenvisualisierungs- und Deformationsverfahren. Bis kurz nach der Jahrtausendwende musste diese mangels Alternativen auf der CPU durchgeführt werden. In herkömmlicher Rasterisierungs-Hardware wurde für Farb- und Beleuchtungsberechnungen die *fixed-function-pipeline* eingesetzt. Pipeline bedeutet in diesem Zusammenhang, dass Daten als Input für die Pipeline dienen und am Ende ein fertig verarbeitetes Ergebnis z.B. auf einem Bildschirm sichtbar gemacht wird. Innerhalb der *fixed-function-pipeline* werden an den Polygon-Eckpunkten die jeweilige Farbe und die Beleuchtung ermittelt. Diese Werte werden dann innerhalb des Polygons interpoliert. Das Verfahren ist auch als *Gouraud-Shading* bekannt (Gouraud, 1971).

Dass man auf die Art und Weise, in welcher die Beleuchtungsberechnung durchgeführt wird, nur sehr eingeschränkt Einfluss nehmen kann, hat zur Folge, dass auch die Ergebnisse verhältnismäßig undifferenziert und statisch sind. Dies stellt eine Restriktion dar, die mittlerweile dank programmierbarer Grafik-Hardware nicht mehr existieren muss. Seit 2001 konkurrieren die Grafikkartenhersteller ATI und NVidia auf einer für diese Zeit neuen Niveaustufe von Grafikkarten. Seitdem unterstützen die Grafikkarten *Shader*. Nun kann man

auf Pixelbasis eigene, sogenannte Fragment-Programme (auch *Pixelshader*[1] genannt) entwickeln, welche die *fixed-function pipeline* in einigen Punkten ersetzen können. Man spricht im Zusammenhang mit *Shadern* von der *programmable pipeline*, da einzelne Komponenten aus der Pipeline nun programmierbar sind. Die Beleuchtungsberechnung auf Pixelbasis mit *Pixelshadern* ähnelt der beim *Phong-Shading* (Phong, 1975). Verwendet man diese Technik, so können aufwändige Berechnungen – bei weitem nicht nur die Beleuchtung betreffend – direkt auf der Grafikkarte durchgeführt werden.

Berechnungen in *Shader*-Programmen außerhalb des Grafikkontextes fasst man unter dem Begriff *GPGPU* (**G**eneral **P**urpose Computations on **G**raphic **P**rocessor **U**nits) zusammen. Die Leistungsfähigkeit von moderner Grafik-Hardware übertrifft heute die von CPUs schon um ein Vielfaches, wobei sie jedoch speziell auf die parallele Datenverarbeitung ausgelegt ist. Mittlerweile gehört programmierbare Grafik-Hardware zum Standard-Umfang von fast jedem neuen Home-PC.

Die Möglichkeiten, welche die *Shader*-Programmierung bietet, erlauben eine wesentlich gezieltere Einflussnahme auf die Berechnungen der Grafikkarte als zuvor. Dies ist der Grundstein für die Überlegungen, auf denen diese Arbeit beruht, da nach Möglichkeit die aufwändigen Berechnungen der Volumendeformation direkt auf der Grafikkarte ausgeführt werden sollen.

2.3 Volumenvisualisierung

Bei allen Formen der Visualisierung von Datensätzen ist es wichtig, dass der Betrachter eine Vorstellung von den Daten entwickeln kann, so dass ihm die Interpretation erleichtert wird. Man kann verschiedene Messwerte z.B. durch die Größe eines Objektes oder seine Farbigkeit darstellen. Die Möglichkeiten sind sehr differenziert und sollten sorgsam ausgewählt werden.

Zu den steigenden Anforderungen an computerunterstütztes Arbeiten zählt auch die anschauliche Darstellung von Volumina. Diese können z.B. im Rahmen von

[1] Neben dem *Pixelshader* müssen an dieser Stelle auch *Vertexshader* sowie *Geometry Shader* erwähnt werden. Diese werden vor dem *Pixelshader* ausgeführt und existieren ebenfalls erst in moderner Grafikhardware, haben jedoch keinen direkten Einfluss auf Beleuchtungsberechnungen.

medizinischen Untersuchungen entstandene dreidimensionale Volumendatensätze sein. Darstellungen von Flächen haben den Vorteil, dass ein Bildschirm ebenfalls zweidimensional ist. Eine geeignete Darstellungsweise ist somit naheliegend und verhältnismäßig einfach umsetzbar. Im Falle von räumlichen Darstellungen, denen dreidimensionale Daten zugrunde liegen, stellt eine adäquate Aufbereitung der komplexen Daten zunächst eine anspruchsvollere Aufgabe dar. Die Herausforderung liegt darin, in einem räumlichen Gebilde verteilte Daten in die Fläche zu übersetzen, also für ein 3D-Objekt eine analoge 2D-Form zu finden.

Problematisch für die Darstellung von dreidimensionalen Daten ist zudem die Tatsache, dass sie maschinell gemessen bzw. generiert werden. Das Ergebnis einer MRT eines Menschen enthält zum Beispiel Informationen über das Knochengewebe, über Haut, Körperflüssigkeiten, Muskeln, etc. Zudem werden auch an den Stellen Daten erfasst, wo man als Mensch intuitiv keine Messung vornehmen würde – so wird beispielsweise bei medizinischen Messungen irrelevanter Raum ebenfalls einbezogen. Daraus resultiert, dass häufig mehr Daten verfügbar gemacht werden, als überhaupt nötig sind.

Wird, wie oben beschrieben, eine so große Menge an Daten erzeugt, hat dies den Vorteil, dass viele Informationen automatisch verfügbar gemacht werden. Dennoch wird es im Einzelnen notwendig sein, *ausschließlich die gewünschten Daten* darzustellen. Ein Anwendungsbeispiel aus der medizinischen Visualisierung soll dies verdeutlichen: Will man einen Muskel darstellen, so müssen darüber liegende Gewebeschichten transparent gemacht oder ausgeblendet werden können. Dazu werden effiziente Verfahren benötigt, die anschauliche Ergebnisse hervorbringen. Einige solcher Verfahren werden im Folgenden vorgestellt und auf ihren Nutzen für meine Arbeit untersucht.

2.3.1 Indirekte Volumenvisualisierung

Bei der indirekten Volumenvisualisierung muss zunächst eine aufwändige Vorverarbeitung der Daten durchgeführt werden. Das bedeutet, dass innerhalb des gegebenen Skalarfeldes eine Oberflächenextraktion durchgeführt werden muss. Die daraus resultierende Oberfläche wird dann mittels auf Polygonen basierender Rendering-Verfahren dargestellt. Diese Oberfläche repräsentiert Werte eines Volumens mit gleicher Dichte. Sie wird als Isofläche bezeichnet. Die Verwendung von Isoflächen führt zu einem Informationsverlust, da die

notwendige Beschränkung auf Oberflächen eine gravierende Reduktion der Daten bedeutet. Soll eine andere Isofläche dargestellt werden, so muss erneut eine Vorverarbeitung stattfinden.

In dieser Arbeit soll ein Volumendatensatz möglichst ohne Vorverarbeitung direkt deformiert werden können. Dafür ist dieses Verfahren nicht geeignet, da es Flächen zurückliefert.

2.3.2 Direkte Volumenvisualisierung

Die direkte Volumenvisualisierung benötigt keine Vorverarbeitung. Die gesamte Information des 3D-Skalarfeldes kann daher genutzt werden. Dazu wird das gegebene Skalarfeld als transparentes Medium interpretiert. Jeder skalare Wert wird auf ein physikalisches Teilchen abgebildet, welches Emissions- und Absorptionseigenschaften aufweist (Hege, Höllerer, & Stalling, 1993). So kann das Volumen mit Hilfe eines *physikalischen Modells für Lichtausbreitung in einem transparenten Medium* dargestellt werden. Bezüglich der Darstellungsmethode wird zwischen Objektraum-Verfahren und Bildraum-Verfahren unterschieden. Es existieren auch Ansätze, die man als Hybridverfahren bezeichnet, da sie eine Mischform aus beiden zuvor genannten Verfahren sind.

Bei einem Objektraum-Verfahren werden einzelne Voxel[2] auf die Bildebene projiziert. Zunächst werden die Beiträge der einzelnen Voxel zu jedem Bildpunkt ermittelt. Diese Beiträge werden dann für jeden Bildpunkt aufsummiert.

In Bildraumverfahren werden die einzelnen Voxel von der Bildebene aus abgetastet. Das bekannteste Bildraumverfahren ist *Raycasting*. Beim *Raycasting* werden von den Bildpunkten des Zielbildes ausgehend Strahlen in die Szene geschickt. Auf diesen Strahlen werden dann die geschnittenen Voxel abgetastet und deren Beiträge zum Ergebnisbild integriert.

Ein Beispiel für ein Hybridverfahren ist der *Shear-Warp-Algorithmus* (Lacroute & Levoy, 1994). Hier wird zunächst das Volumen so transformiert, dass die Sehstrahlen senkrecht auf den einzelnen Schichten des Volumens stehen. Dieser Schritt entspricht dem eines Objektraumverfahrens. Das Ergebnis ist ein verzerrtes Zwischenbild, welches nun in einem 2D-Nachbearbeitungsschritt

[2] Voxel ist eine Abkürzung für *Volume Element* (analog zu Pixel für *Picture Element* steht)

entzerrt werden muss. Dieser Nachbearbeitungsschritt entspricht einem Schritt eines Bildraumverfahrens.

Die Art und Weise, in welcher der Integrationsvorgang durchgeführt wird, bezeichnet man auch als *Compositing*. Es existieren verschiedene *Compositing*-Verfahren, die sich zum Teil in der Geschwindigkeit sowie im dargestellten Ergebnis selbst deutlich unterscheiden. Im Folgenden stelle ich verschiedene *Compositing*-Verfahren kurz vor:

- **Integration von Emission und Absorption:** Entlang eines Strahls wird die Lichtenergie integriert. Dabei können Teilchen Licht emittieren oder absorbieren.

- **Integration von Emission und Absorption über eine Transferfunktion:** Über eine Transferfunktion können verschiedene Bereiche anhand der Skalarwerte im Volumen mit unterschiedlichen Farben oder auch transparent dargestellt werden.

- **Mittelwert:** Entlang eines Strahls wird der Mittelwert über alle Farbwerte gebildet.

- **Maximum Intensity Projection:** Entlang eines Strahls wird der maximale Wert ermittelt.

- **First Local Maximum:** Entlang eines Strahls wird der Wert des ersten lokalen Maximums ermittelt.

Die oben genannten Integrations-Verfahren ermöglichen im Gegensatz zur indirekten Volumenvisualisierung eine direkte und verlustfreie Darstellung von Volumina.
Bei der Integration entlang eines Strahls gibt es die Möglichkeit, die Werte von hinten nach vorne oder von vorne nach hinten zu integrieren. Im erstgenannten Fall spricht man von *back-to-front-compositing*, im anderen von *front-to-back-compositing*. Nutzt man *front-to-back-compositing*, so besteht die Möglichkeit einer Geschwindigkeitsoptimierung. Diese beruht darauf, dass halbtransparente Materialien ab einer gewissen Tiefe kein Licht mehr durchlassen. Für das Compositing-Verfahren bedeutet dies, dass die Integration abgebrochen werden

kann, wenn ein bestimmter Wert überschritten wurde. Man nennt dies *early-ray-termination* (Levoy, 1988).

Sowohl Interpolationen als auch die numerische Integration sind sehr rechenintensiv. Daher eignet sich die moderne programmierbare Grafik-Hardware sehr gut dazu, diese Prozesse darauf durchzuführen.

Ein für diese Arbeit relevanter Ansatz wird im Folgenden erläutert.

2.3.2.1 Texturbasiertes Rendering

Beim texturbasierten Ansatz der Volumenvisualisierung werden die Vorzüge der *Pixelshader*-Technologie genutzt. Aus diesem Grund ist die Realisierung von interaktiven bzw. echtzeitfähigen Algorithmen möglich, ohne dass dafür spezielle Hardware benötigt wird.

Die Idee beim texturbasierten Rendering ist, dass *Volume-Rendering* eine sehr hohe Anzahl an Interpolationen bedeutet. Daher nutzt man die beschleunigte Interpolation der Grafik-Hardware. Zunächst wird aus dem vorhandenen 3D-Volumendatensatz eine textur-basierte Repräsentation generiert. Dies kann auf verschiedene mehreren Wegen durchgeführt werden:

- 2D-Texturen: Bei der Nutzung von 2D-Texturen wird eine Zerlegung des Volumens in achsenparallele Schichten durchgeführt. Somit müssen drei Kopien des Datensatzes gespeichert werden – für jede Schicht eine. Beim Integrieren der Emission bzw. der Absorption entlang eines Sehstrahls müssen die entsprechenden Werte im Volumen per t*exture-lookup* bilinear interpoliert werden. Die Interpolation wird in Abhängigkeit von der Blickrichtung in den entsprechenden Schichten durchgeführt. Es muss immer der Schichtenstapel gewählt werden, in dem die Ausrichtung der einzelnen Schichten den größten Winkel zum Sehstrahl hat.

  Der Nachteil dieses Ansatzes ist, dass visuelle Artefakte auftreten, falls die Blickrichtung nicht senkrecht zum Schichtenstapel ist. Dies liegt daran, dass senkrecht zum Schichtenstapel nicht interpoliert wird. Daher wird der Abtast-Abstand entlang des Sehstrahls größer, je schräger die Blickrichtung zum jeweiligen Schichtenstapel ist. Für gute visuelle Ergebnisse muss der Abtast-Abstand jedoch unabhängig von der Blickrichtung konstant bleiben (Cabral, Cam, & Foran, 1994). Da eine

möglichst hohe visuelle Qualität der Volumendeformation erzielt werden soll, ist die Verwendung von 2D-Texturen für meine praktische Arbeit nicht geeignet.

- 3D-Texturen: Hier wird das Volumen auf eine 3D-Textur abgebildet. 3D-Texturen haben den Vorteil, dass die Aufteilung des Volumens nicht auf Achsenparallelität beschränkt ist. Die Schichten können parallel zur Bildebene erzeugt werden. Dadurch ist eine Abtastung in konstantem Abstand möglich. Eine Überabtastung des Volumens kann einfach durch eine Erhöhung der Abtastrate durchgeführt werden. Nachteilig ist hier, dass das gesamte Volumen als 3D-Textur in den Speicher passen muss, da es sonst aufgrund vieler zusätzlicher Datentransfers von Volumendaten in den Speicher der Grafikkarte zu erheblichen Leistungseinbußen kommen kann. Dem kann entgegengewirkt werden, indem das Volumen in kleinere Subvolumina unterteilt wird. Bei einer geeigneten Wahl der Unterteilung kann der Transfer der Volumina sowie das Rendering parallel ausgeführt werden (Wilson, VanGelder, & Wilhelms, 1994). Für die durchzuführenden Untersuchungen hinsichtlich der Volumendeformation ist bietet sich die Nutzung dieses Verfahren an.
- 2D-multi-Texturen: Bei diesem Ansatz sollen die Vorteile der schnellen 2D-Texturen genutzt und die Nachteile kompensiert werden. Die Ausgangskonfiguration ist hier dieselbe wie bei 2D-Texturen. Nun soll aber ermöglicht werden, dass an jeder beliebigen Position abgetastet werden kann, um visuelle Artefakte zu vermeiden. Dies erreicht man durch eine bilineare Interpolation und *blending* zweier benachbarter Texturschichten, was dann einer trilinearen Interpolation entspricht. Dadurch, dass nur noch die nötigen Informationen auf die Grafikkarte transferiert werden müssen, kann ein effektives *Load-Balancing* erreicht werden (Rezk-Salama, Engel, Bauer, Greiner, & Ertl, 2000). Auch dieses Verfahren ist potentiell für meine Aufgabenstellung geeignet.

Nachdem nun eine Repräsentation des Volumens in Texturform vorliegt, werden die einzelnen Schichten der Textur unter Verwendung eines *Compositing*-Verfahrens gerendert (siehe Kapitel 2.3.2). Eine Klassifikation des Volumens kann anhand einer Transferfunktion durchgeführt werden. Diese Transferfunktion

hat die Aufgabe, die Datenwerte aus dem 3D-Skalarfeld auf Farben bzw. die Opazität abzubilden.

Die texturbasierten Ansätze erfordern, dass das darzustellende Volumen in Form einer Textur in den Grafikspeicher geladen wird. Dies ist für mein Vorhaben insofern nützlich, dass die später durchgeführte Deformation ebenfalls in Form einer Textur mit dem Volumen im Grafikspeicher kombiniert werden soll. Auch die gute Performance dieses Verfahrens, begründet durch die beschleunigte Grafik-Hardware, spricht für den Einsatz in meiner Software.

2.4 Deformationssysteme

Deformation wird in verschiedenen Anwendungen wie z.B. Computerspielen, Kleidungsanimation, Muskelanimation, *Virtual Reality*-Anwendungen etc. bereits intensiv eingesetzt.

Die Deformation volumetrischer Objekte in der Realität ist ein so komplexer Vorgang, dass eine Computersimulation dies nicht adäquat repräsentieren kann. Die physikalische Tatsache etwa, dass jedes einzelne Atom sein Nachbaratom beeinflusst, zeigt, wie folgenreich eine jede Krafteinwirkung ist. Es müssten also zahllose Faktoren für eine entsprechende realitätsabbildende Simulation berücksichtigt werden, was nie vollständig zu leisten ist. Die Entwicklung neuer Modelle jedoch hat eine immer präzisere Darstellung zum Ziel.

Derzeit kann folglich nur mit Kompromisslösungen gearbeitet werden. Umso wichtiger ist es, im Vorfeld die Anforderungen an ein Modell genau zu bestimmen. Nur so kann ein geeignetes Modell ausgewählt werden, etwa in Abhängigkeit von der gewünschten Genauigkeit oder Effizienz.
Grundsätzlich sollten Rechenaufwand und Speicherbedarf als gewichtige Faktoren bei der Konzeption eines Deformationsmodells eingeplant werden.

Im folgenden Kapitel werden zunächst die Grundlagen für die Deformation erläutert. Darauf aufbauend werden dann das Masse-Feder-Modell und der ChainMail-Algorithmus vorgestellt und im Hinblick auf die Verwertbarkeit für meine Entwicklung beurteilt. Ich habe diese beiden Modelle ausgewählt, da nach ersten Untersuchungen eine Umsetzung als möglich eingestuft wurde. Auch die Tatsache, dass das Masse-Feder-Modell physikalisch-basiert und das ChainMail-Modell geometrie-basiert ist, legt einen Vergleich der beiden Ansätze nahe.

2.4.1 Grundlagen der Deformation

2.4.1.1 Hookesches Gesetz der Elastizität

Das *Hookesche Gesetz der Elastizität* beschreibt, dass eine auf einen Körper wirkende Kraft $\vec{F}$ eine proportionale Abhängigkeit zu der Verformung l des Körpers aufweist. Dies kann durch folgende Gleichung ausgedrückt werden:

$$\vec{F} = D \cdot \Delta l \tag{1}$$

Diese Gleichung beschreibt im Prinzip eine lineare Feder, wobei D eine Federkonstante darstellt.

Allgemeiner formuliert stellt das *Hookesche Gesetz* eine Spannung σ in eine lineare Beziehung zu einer Dehnung ε in einem elastischen Körper:

$$\sigma = E \cdot \varepsilon \tag{2}$$

E stellt in Gleichung (2) eine Proportionalitätskonstante dar und wird auch als Elastizitätsmodul bezeichnet. In der Realität existiert keine lineare Beziehung zwischen Spannung und Dehnung, wobei dieses Modell dennoch in Teilen auf die Realität übertragbar ist. Das bedeutet, dass das *Hookesche Gesetz* angewendet werden kann, solange die Dehnung einer Feder in einem bestimmten Rahmen bleibt. Wenn jedoch ein vom Material abhängiger Wert überschritten wird, so verliert es an Gültigkeit.

2.4.1.2 Zweites Newton'sches Gesetz

Newtons zweites Gesetz wird als das Aktionsprinzip bezeichnet. Es besagt, dass wenn eine Kraft $\vec{F}$ auf eine Masse m wirkt, diese Masse m in Richtung der Kraft $\vec{F}$ mit der Beschleunigung $\vec{a}$ beschleunigt wird. Dieser Zusammenhang wird in Gleichung (3) deutlich gemacht:

$$\vec{F} = m \cdot \vec{a} \tag{3}$$

Man kann dies aus der Definition, dass eine Kraft einer Impulsänderung über die Zeit ist, herleiten.

Ein Impuls $\vec{p}$ ist als mit einer Geschwindigkeit $\vec{v}$ bewegte Masse m definiert (siehe Gleichung (4)).

$$\vec{p} = m \cdot \vec{v} \tag{4}$$

Die Definition der Beschleunigung $\vec{a}$ ist eine Geschwindigkeitsänderung über die Zeit *t* (siehe Gleichung (5)).

$$\vec{a} = \frac{d\vec{v}}{dt} \tag{5}$$

Die Geschwindigkeit $\vec{v}$ ist wiederum als eine Änderung des Weges *s* in der Zeit *t* definiert (siehe Gleichung (6)).

$$\vec{v} = \frac{ds}{dt} \tag{6}$$

Somit ergibt sich folgende Gleichung:

$$\vec{F} = \frac{d\vec{p}}{dt} = \frac{m \cdot d\vec{v}}{dt} = m \cdot \vec{a} = m \cdot \ddot{s} \tag{7}$$

Aus den Zusammenhängen der obigen Gesetzmäßigkeiten in Gleichung (7) kann geschlussfolgert werden, dass durch die alleinige Kenntnis von Position und Masse die Wirkung von Kräften über die Zeit simuliert werden kann. Dies dient als Grundlage für die physikalisch-basierten Deformationsmodelle.

Durch die obige Ausführung ist klar, welche physikalischen Zusammenhänge betrachtet werden müssen. Ungeklärt ist bisher jedoch, auf welche Art und Weise genau die Berechnung eines physikalischen Teilchens erfolgen muss. Im folgenden Kapitel werden verschiedene Integrationsverfahren vorgestellt, welche diese Thematik betreffen:

2.4.1.3 Explizites Euler-Verfahren

Das explizite Euler-Verfahren ist ein Integrationsverfahren und wird genutzt, um z.B. die Bahn von Partikeln in physikalisch-basierten Systemen zu ermitteln. Es soll ausgehend von einer Ausgangsposition $\vec{x}_i$ die neue Position $\vec{x}_{i+1}$ in Abhängigkeit von einer momentanen Geschwindigkeit $\vec{v}(\vec{x}_i)$ berechnet werden (siehe Gleichung (8)).

$$\vec{x}_{i+1} \approx \vec{x}_i + \tau \cdot \vec{v}(\vec{x}_i) \tag{8}$$

Die Genauigkeit des Verfahrens hängt unmittelbar von der Schrittweite τ ab. Je kleiner τ gewählt wird, umso genauer ist das Ergebnis. Für eine große Schrittweite ist das explizite Euler-Verfahren numerisch instabil, da Änderungen der Geschwindigkeit nur näherungsweise berücksichtigt werden können.
Die Güte eines Verfahrens wird durch die Fehlerordnung ausgedrückt. Das explizite Euler-Verfahren hat die Fehlerordnung 1.

2.4.1.4 Implizites Euler-Verfahren

Beim impliziten Euler-Verfahren sucht man eine Position $\vec{x}_{i+1}$, mit deren Geschwindigkeitsvektor man von der Ausgangsposition $\vec{x}_i$ die Folgeposition $\vec{x}_{i+1}$ erreichen würde. Das implizite Euler-Verfahren ist zwar nicht genauer als das explizite, denn es hat ebenfalls die Fehlerordnung 1. Der Vorteil aber liegt darin, dass es numerisch stabil bleibt. Die Berechnung ist allerdings im Allgemeinen aufwändiger und benötigt iterative Verfahren.

2.4.1.5 Runge-Kutta-Verfahren

Das als Runge-Kutta-Verfahren zweiter Ordnung bekannte Heun-Verfahren (oder auch Prädiktor-Korrektor-Verfahren) kann als Kombination aus dem impliziten und dem expliziten Euler-Verfahren angesehen werden. Anhand des expliziten Euler-Verfahrens wird in einem Prädiktor-Schritt die Position ermittelt, die in einem zweiten Korrektor-Schritt durch Anwendung des impliziten Verfahrens korrigiert wird.

Hinter diesem Ansatz verbirgt sich eine ganze Klasse von Verfahren: die *Runge-Kutta-Verfahren.*
Beim klassischen Runge-Kutta-Verfahren vierter Ordnung werden die Diskretisierungsfehler durch gezielte Kombination verschiedener Differenzquotienten weitestgehend kompensiert. Es sind jedoch noch weitere Runge-Kutta-Verfahren bekannt, die eine noch bessere Fehlerordnung besitzen (Fehlberg, 1969).

2.4.2 Deformationsmodelle

In diesem Kapitel werden die theoretischen Grundlagen der Deformationsmodelle erläutert, die bei der praktischen Umsetzung in Software zum Tragen kommen.

2.4.2.1 Masse-Feder-Modell

Da das Masse-Feder-Modell auf Gesetzmäßigkeiten der Mechanik beruht, zählt es zur Kategorie der physikalisch-basierten[3] Deformationsmodelle.

Man nimmt an, dass Massepunkte im Raum verteilt sind (siehe Abbildung 1).

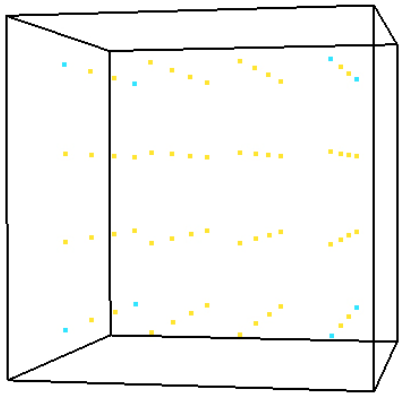


Abbildung 1

Diese Massen werden durch gedämpfte Federn verbunden, wobei die Federn einen Ruhezustand haben können, gestaucht oder gedehnt werden können. Zwei beliebige Massen können durch eine Feder miteinander verknüpft werden. Abbildung 2 zeigt ein Masse-Feder-System, dessen Federn als rote Linien dargestellt sind.

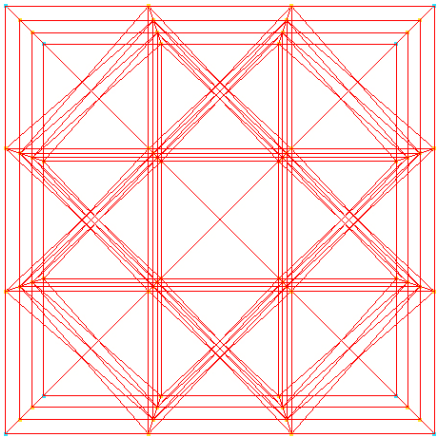


Abbildung 2

Wird eine Feder gestaucht oder gedehnt, so wirkt eine Kraft auf die beiden

[3] Im Gegensatz zu geometrie-basierten Deformationsmodellen (siehe Kapitel 2.4.2.2)

miteinander verbundenen Massen, da die Kraft linear abhängig von der Länge der Feder ist (siehe Kapitel 2.4.1). Die Kraft veranlasst eine Feder, möglichst immer wieder zurück in den Ruhezustand zu kehren. Auf diese Weise kann sich eine Deformation über die Federn im System fortpflanzen. In Abbildung 3 ist eine Deformation abgebildet. In diesem Beispiel wurden vier Elemente eines würfelförmigen Objektes nach oben bewegt, woraufhin mit dem Masse-Feder-Modell die dargestellte Deformation berechnet wurde.

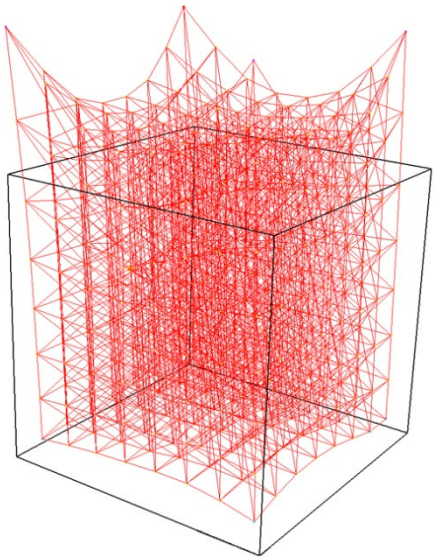


Abbildung 3

Da bei der Berechnung der Deformation die Kräfte in diskreten Zeitschritten aufsummiert werden, ist das Ergebnis unabhängig von der Auswahl des Integrationsverfahrens nur eine Näherungslösung.

2.4.2.2 ChainMail

Der 3D-ChainMail-Algorithmus ist geometrie-basiert. Er wurde dafür entwickelt, volumetrische Objekte in Echtzeit zu deformieren, um etwa in chirurgischen Simulationen seine Anwendung zu finden. Das Grundprinzip funktioniert folgendermaßen:

Elemente sind bei ChainMail miteinander verknüpft, so, wie es Kettenglieder bei einer Kette sind. Wird nun ein Element von seinem Nachbarelement weg bewegt, muss sein Nachbarelement sich genau dann in die gleiche Richtung bewegen, wenn ein maximaler Abstand zwischen beiden Elementen überschritten wurde.

Wenn das Element in Richtung seines Nachbarelements bewegt wird, so muss der Nachbar im Falle der Unterschreitung eines minimalen Abstandes bewegt werden. Der dritte Fall, in dem ein Nachbarelement bewegt werden muss, tritt dann ein, wenn eine maximale Scherung zu diesem überschritten wird. Eine Scherung bedeutet hier eine Bewegung in senkrechter Richtung zu der Position der Nachbarelemente.

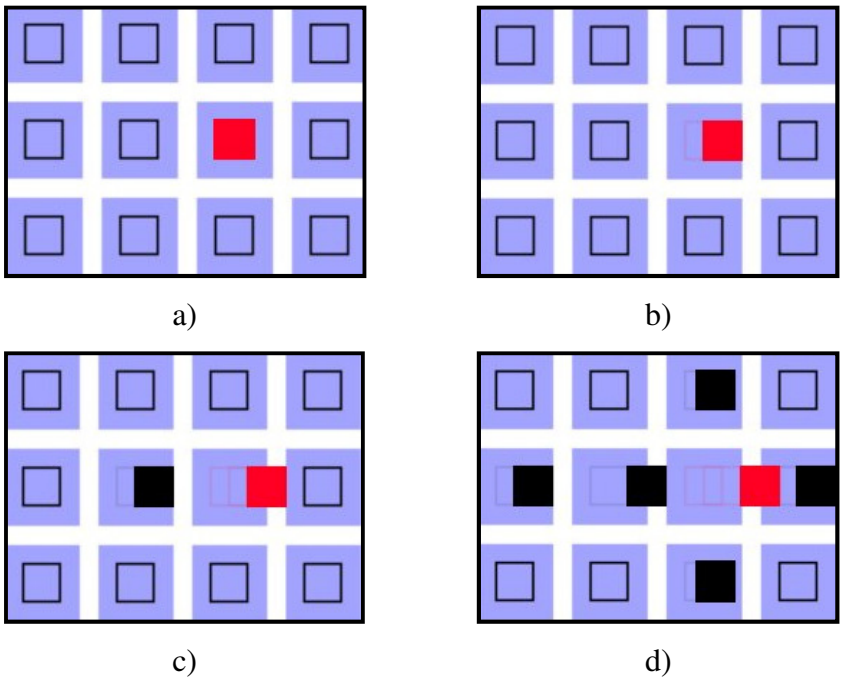


Abbildung 4

In Abbildung 4 wird der Ablauf einer sich fortpflanzenden Deformation für den zweidimensionalen Fall dargestellt. Das gewählte bzw. bewegte Element ist rot gefärbt. Die lokalen Abstandsbedingungen sind blau gefärbt. Das ist der Bereich, in dem sich ein Element bewegen kann, ohne seine Nachbarn zu beeinflussen. Nicht bewegte Elemente sind durch einen schwarzen Rahmen angedeutet, während bewegte Elemente durch schwarze Quadrate dargestellt werden. In Abbildung a) ist der Ursprungszustand abgebildet. Darstellung b) zeigt eine Bewegung des gewählten Elements, wobei noch kein Nachbarelement beeinflusst wird. Bei Bild c) wird ein Nachbarelement mitverschoben, da zu diesem der maximale Abstand überschritten wurde. In Abbildung d) werden weitere vier Elemente bewegt. Der rechte Nachbar wird verschoben, da der minimale Abstand unterschritten wurde. Für die Elemente darüber und darunter wurde die Scherungsbedingung verletzt.

Verletzt ein Element eine der Bedingungen, muss die Position des betroffenen Elementes so korrigiert werden, dass die Bedingung wieder erfüllt ist. Solange ein Element sich bewegt und keine Abstandsbedingung zu seinen Nachbarn verletzt, wird demnach auch kein anderes Element beeinflusst. Dies hat den Vorteil, dass nur diejenigen Elemente in die Berechnung miteinbezogen werden müssen, welche auch tatsächlich bewegt werden. Auf diesem Weg pflanzt sich eine etwaige Bewegung lokal in einem Objekt fort. ChainMail ist auf Grund der Lokalitätseigenschaft besser als das Masse-Feder-Modell für die Bearbeitung großer Datensätze geeignet. Dies gilt jedoch nur für kleine Bewegungen der Elemente, da bei größeren Bewegungen auch mehr Elemente angepasst werden müssen und somit der Lokalitätsvorteil gegenüber anderen Algorithmen verloren geht. Die Berechnung des ChainMail Algorithmus‘ ist Aufgrund der Komplexität von O(n) sehr schnell. Jedes Element wird maximal einmal überprüft bzw. verändert (Gibson, 1997).

Die Entscheidung, ob ein Element überprüft werden muss, wurde im originalen Algorithmus mit einer *first-in-first-out*-Liste je Richtung der Nachbarelemente getroffen. Somit stehen sechs Listen zur Verfügung, die nach dem Rotationsprinzip abgearbeitet werden. Damit wird eine gleichmäßige Deformation in alle Richtungen sichergestellt. Ein Element kommt genau dann in eine Liste, wenn es die Abstandsbedingungen zu einem Nachbarelement verletzt - also zu demjenigen Element, welches die Bewegung initiiert hat. Dieses Element wird auch Sponsor genannt. Ein Sponsor-Element wird bei einer Überprüfung auf Einhaltung der Abstandskriterien eines Elementes nicht miteinbezogen. Dadurch kann gewährleistet werden, dass jedes Element nur einmal verarbeitet wird.

Die Nutzung von sechs Listen hat sich als nicht verwendbar für heterogene Datensätze herausgestellt, da in diesem Fall die Elastizität eines Materials in unterschiedlichen Richtungen variiert. Folglich können auf diese Art und Weise keine heterogenen Materialien simuliert werden. Daher gibt es einen alternativen Denkansatz namens *Enhanced–ChainMail* (Schill, Gibson, Bender, & Reinhard, 1998). Der Grundgedanke dabei ist, dass die Grenzrestriktionen der Nachbarelemente dynamisch sein müssen. Das heißt, dass verschiedene Gewebetypen verschiedene Restriktionen bezüglich der Entfernung zu ihren Nachbarn haben müssen. In diesem Ansatz nutzt man anstatt der sechs Listen nur

noch eine *priority queue*, in der die Elemente sortiert eingefügt werden. Als Kriterium für die Sortierung dient hierbei das Ausmaß der Verletzung der Abstandsbedingungen zu den Nachbarelementen.

Die Simulation chirurgischer Eingriffe erfordert möglicherweise, Schnitte im Gewebe durchführen zu können. Das bedeutet, dass es wünschenswert ist, die Topologie verändern zu können. Dies ist bei ChainMail einfach zu erreichen, da sich die Verknüpfung einzelner Elemente aufheben lässt. Umgekehrt können Elemente verknüpft werden, um beispielsweise eine Verbindung von verschiedenen Geweben zu simulieren. Weiterhin bietet der ChainMail-Algorithmus die Möglichkeit, anisotrope Materialien wie z.B. Muskeln zu simulieren. Es können für jede Achse verschiedene Materialeigenschaften festgelegt werden.

Mit dem ursprünglichen ChainMail-Algorithmus ist nur die Verarbeitung von Daten möglich, die auf rectilinearen Gittern angeordnet sind. Für die Verwendung in dieser Arbeit ist dies ausreichend, da die gegebenen Volumendatensätze eben jene Anordnung haben. In anderen Anwendungsfällen sind gesteigerte Anforderungen denkbar. Li & Brodlie bieten hierfür eine Lösungsmöglichkeit an, die das Problem der topologischen Anordnung der Nachbarn kein Problem umgehen (Li & Brodlie, 2003).

Visuell sind die Ergebnisse einer Deformation mit Hilfe des ChainMail-Algorithmus‘ nicht optimal, da bei einer Deformation unnatürlich wirkende, geradlinige Strukturen von der Bewegungsachse ausgehend sichtbar sind. Um dieses Problem zu korrigieren, gibt es die Möglichkeit, so genannte *Relaxation*-Schritte durchzuführen. Dieses Prinzip wird im folgenden Kapitel vorgestellt.

2.4.2.2.1 *Relaxation*

Physikalisch gesehen erfordert die Bewegung von Objekten Energie. Für ein Deformationsmodell, das aus einzelnen Elementen besteht, bedeutet dies folgendes: Wenn von einem zu deformierenden Objekt ein zugehöriges Element bewegt wird, so wird letztlich dem Objekt Energie zugeführt. In Ruhelage hat das

Objekt nach dem *Modell für elastische Materialien* eine minimale Energiekonfiguration. Für plastische Materialien muss die minimale Energiekonfiguration nicht der ursprünglichen Ruhelage entsprechen, da Energie in Wärme umgewandelt werden kann. Natürliche Materialien sind in der Regel plastoelastisch. Im Falle einer Energieeinwirkung von außen tritt also ein Energieverlust auf. Das deformierte Objekt ist immer bestrebt, eine minimale Energiekonfiguration zu erreichen.

Eine Darstellung von Rundungen, wie reale deformierte Objekte sie häufig aufweisen, ist mit den Ergebnissen der ChainMail-Berechnung oftmals nicht möglich, so dass das Ergebnis der Deformation nicht realistisch wirkt. Dies wird besonders deutlich, wenn viele Elemente von der Deformation betroffen sind. Abbildung 5 zeigt eine Deformation eines Würfels, bei dem ein Element nach rechts verschoben wurde.

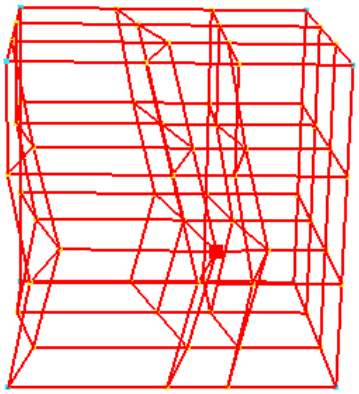


Abbildung 6

Schon bei einer geringen Anzahl betroffener Elemente, wie im dargestellten Fall, kann der beschriebene Sachverhalt beobachtet werden: Die bewegten Elemente verformen sich nach geometrischen Bedingungen, so dass der Betrachter keine Assoziation zu einem in der Natur existierenden Gewebe hat. Dies ist aber gerade das Ziel der meisten Anwendungen der Volumendeformation, da in der Regel die Verformung eines natürlichen Objekts simuliert werden soll.

Durch zusätzliche *Relaxation*-Schritte kann der visuelle Nachteil dieses geometrie-basierten Modells ausgeglichen werden. Bei einer physikalischen Betrachtung des Modells dienen diese Schritte dazu, die ungleichmäßig verteilte Energie innerhalb des deformierten Objektes auszugleichen. Das bedeutet, dass

die Position der Elemente anhand bestimmter Umgebungskriterien korrigiert wird. Setzt man beispielsweise die einzelnen Elemente auf den jeweiligen Mittelpunkt ihrer direkten Nachbarelemente, so resultiert daraus eine gleichmäßigere Verteilung der Elemente und das Ergebnis wird visuell deutlich stimmiger. Für einen *Relaxation*-Schritt muss jedes Element einmal angeglichen werden. Das kann bei einer großen Anzahl an Elementen durchaus viel zusätzlich benötigte Rechenzeit bedeuten. Ein Vorteil ist, dass nur sehr wenige *Relaxation*-Schritte benötigt werden, um eine erhebliche Verbesserung des Deformationsergebnisses zu erzielen.

Abbildung 7 zeigt auf der linken Seite ein mit dem ChainMail-Algorithmus deformiertes Objekt. Auf der rechten Seite wurde ein *Relaxation*-Schritt für die Elemente des Objektes durchgeführt. Es ist gut erkennbar, dass sich auf der rechten Seite das Objekt seiner minimalen Energiekonfiguration angenähert hat.

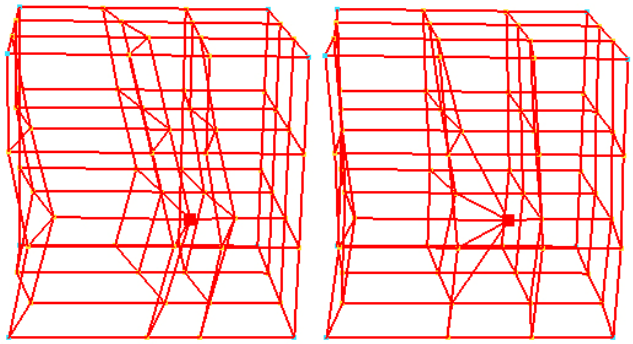


Abbildung 7

2.5 Volumendeformation

Den Kern dieser Arbeit bildet die GPU-basierte Volumendeformation. Volumendeformation steht hier für eine Kombination aus Deformation und Visualisierung. Ziel ist in erster Linie, einen Volumendatensatz in Echtzeit darzustellen und deformieren zu können. Das bedeutet in der Konsequenz, dass zuerst ein Deformationsmodell gewählt werden muss. Für das Ergebnis, das dieses Modell liefert, muss dann eine geeignete Methode gefunden werden, um die Deformation auf einen vorhandenen Volumendatensatz anwenden zu können. Die Zusammenführung von Deformation und Visualisierung des Volumens geschieht in einem *Shader*-Programm auf der Grafikkarte. Dieses Programm soll anhand der vorliegenden Deformationsergebnisse den Volumendatensatz darstellen. Man kann sich dies modellhaft so vorstellen, als ob ein Lichtstrahl, der

das Volumen passiert, entlang der errechneten Deformation umgelenkt wird. Im Ergebnisbild sieht dann das Volumen für den Betrachter so aus, als sei es deformiert.

Es ist aufgrund des oben genannten Vorhabens sinnvoll, eine passende Anordnung der Daten zu wählen, so dass die Daten der Deformation nicht mehr zusätzlich für die Visualisierung umgerechnet werden müssen. Da die Darstellung des Volumens mit dem texturbasierten Volume-Rendering Ansatz erfolgen soll, liegt das Volumen zum Zeitpunkt der Darstellung als 3D-Textur vor.

Das Deformationsergebnis muss demnach in einer analogen Form gespeichert werden. Für die Deformation muss also eine Textur erstellt werden – die so genannte Offset-Textur – aus der zur Laufzeit die Deformationsinformation ausgelesen werden kann.

Für jedes Element der Deformation muss gespeichert werden, welchen Abstand es zu seinem Ursprungspunkt eingenommen hat. Daher wird in jedem Texel der Offset-Textur der Offset eines Deformationselements zu einem bestimmten Zeitpunkt abgespeichert.

Die ungünstige Konstellation aus beschränkter Größe des Grafikspeichers und speicherintensiven Volumentexturen kann zu einem Engpass in der Berechnung führen. Volumendatensätze sind in der Regel sehr groß. Ein realer Datensatz hat eine Größe von mindestens 256x256x256 Voxel, *Computertomografie* (CT)-Datensätze haben oft 512x512x512 Voxel und mehr. Der Skalarwert eines Voxels wird durch 8- oder 16-Bit Integer-Werte repräsentiert. Daraus resultiert dann ein Speicherbedarf von 128 MB auf der Grafikkarte und mehr.

3 Entwicklung

3.1 Einleitung

In diesem Kapitel werden einführend die Ziele benannt, die mit der entwickelten Software umgesetzt werden sollen. Anschließend wird die grobe Struktur der Software dargestellt. Dann wird auf das eingesetzte Verfahren der Volumenvisualisierung näher eingegangen. Die Implementierung der einzelnen Deformationssysteme sowie die Umsetzung der Volumendeformation werden abschließend im Detail vorgestellt.

3.2 Ziele

Im Rahmen dieser Arbeit soll eine Software entwickelt werden, mit der eine effiziente Volumendeformation durchgeführt werden kann. Das Programm soll dem Nutzer Auswahlmöglichkeiten zwischen verschiedenen Deformations-systemen bieten. Unter Verwendung der Ergebnisse des Theorieteils soll das Masse-Feder-Modell sowie eine Implementierung des ChainMail-Algorithmus' auswählbar sein. Weiterhin soll die Möglichkeit bestehen, die konkrete Berechnung der Deformation auch auf der Grafikkarte durchzuführen, um die verschiedenen Möglichkeiten auf ihre Vorzüge und Nachteile zu überprüfen.

Die Interaktion mit dem Programm soll vorwiegend mit der Maus stattfinden. Der Fokus bei der Interaktion liegt darauf, dass das Deformations-Volumen bewegt, rotiert und skaliert werden kann. Einzelne Elemente des Deformationsmodells sollen mit der Maus ausgewählt und frei bewegt werden können. Dies beinhaltet, dass nach Möglichkeit nicht nur ein Element für die Deformation bewegt werden kann, sondern auch mehrere Elemente simultan. Die Bewegung soll nur in der zur Bildebene parallelen Ebene durchgeführt werden können, in der sich das zu bewegende Element befindet. Sind mehrere Elemente gewählt, sollen diese relativ zueinander die gleiche Position behalten. Dabei soll nur ein auswählbares Referenz-Element gesteuert werden können.

Verschiedene Features wie z.B. die Darstellung des Deformationsmodells, des Volumens, der Offset-Textur etc. sollen je nach Bedarf ein- bzw. ausgeblendet werden können. So soll gewährleistet sein, dass jedes Feature separat auf seine Funktionalität hin überprüft und genutzt werden kann.

Von primärer Bedeutung für dieses Projekt sind die Effizienz der Volumendeformation und deren Umsetzung auf der Grafikkarte. Die physikalische Korrektheit ist zwar erwünscht, spielt jedoch eher eine untergeordnete Rolle. Sowohl die Visualisierung als auch die Deformation erfordern ein hohes Maß an Interaktivität bei guter Qualität der Darstellung. Die Auflösung des Volumens sowie die Komplexität des Deformationsmodells sollen daher veränderbar sein, damit der User selbst entscheiden kann, welches Attribut für ihn von größerer Bedeutung ist.

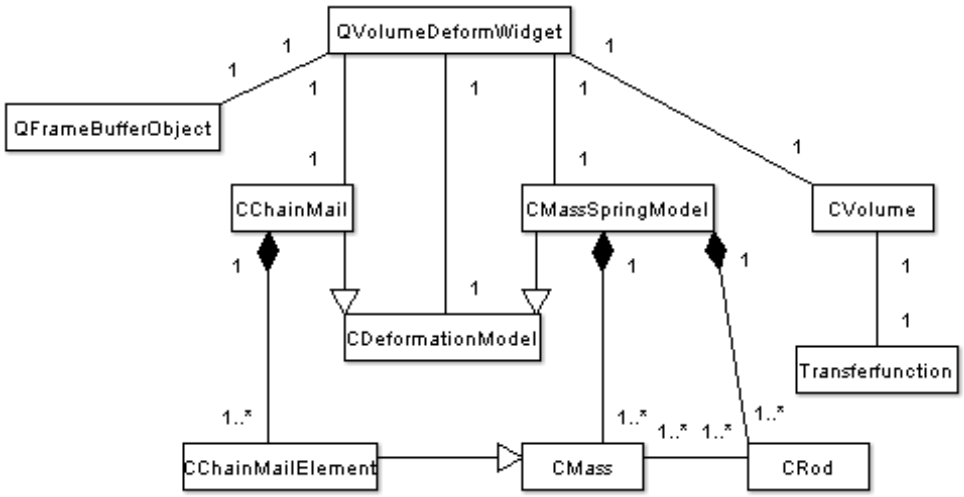
Für die Deformationsmodelle ist außerdem notwendig, dass unterschiedliche Materialeigenschaften für einzelne Elemente simuliert werden können. Besonders wichtig ist hier, dass Elemente fixiert werden können[5]. Dies ist erforderlich, damit eine Deformation das Volumen nicht nur im Ganzen verschiebt.

Die Umsetzung der Software soll einem modularen Aufbau folgen. Dies ist wichtig, um die verschiedenen Verfahrensweisen ohne aufwändige Umstrukturierungen innerhalb des Programms austauschen zu können. Es soll eine Schnittstelle erzeugt werden, an die auch weitere Deformationssysteme angebunden werden könnten. Das gewählte Verfahren der Volumenvisualisierung soll ebenfalls austauschbar bzw. erweiterbar sein.

Die Entwicklung der Software, auf welcher die Ergebnisse dieser Arbeit basieren, erfolgte zum größten Teil auf einem PC mit einer *NVidia GeForce 7950 GX2* Grafikkarte mit 512 MB Speicher unter Verwendung von Microsoft Visual Studio .NET 2005. Für die *Shader*-Programmierung wurde die an *C* angelehnte Programmiersprache *NVidia CG* verwendet. Der Kern der Software wurde in *C++* entwickelt. Die grafische Benutzeroberfläche wurde mit *Qt* erstellt. Heute sind zwei Grafik-APIs weit verbreitet – *DirectX* und *OpenGL*. Erstere ist speziell auf Windows-Betriebssysteme ausgelegt, während *OpenGL* den Anspruch erhebt, auf viele verschiedene Systeme portierbar zu sein. Daher habe ich mich dafür entschieden, die grafische Darstellung mit *OpenGL* durchzuführen. *OpenGL* ist eine frei verfügbare Grafikbibliothek.

[5] Dies entspricht einem theoretisch nicht deformierbaren Material.

3.3 Struktur der Software



Der grobe Ablauf der Software ist im Folgenden dargestellt:

1. Initialisierung von *OpenGL* und der *GUI*
2. Initialisierung des Volumens
3. Initialisierung des Deformationsmodells
    a. Initialiserung der Offset-Textur
    b. Initialisiserung der Deformationselemente
4. Darstellung von Volumen + Deformationsmodell
5. Userinteraktionen
6. Durchführung der Deformationsberechnungen
    a. Bestimmung der Deformation
    b. Update der Offset-Textur anhand Deformation
7. Wiederhole ab Schritt 4.

3.4 Volumenvisualisierung

Aufgrund der Anforderungen an diese Diplomarbeit sowie den Ergebnissen des Theorieteils habe ich mich für den textur-basierten Ansatz der Volumenvisualisierung entschieden.

Textur-basiertes *Volume-Rendering* soll die Basis für die Visualisierung der Volumendaten bilden. Das 3D-Volumen wird daher mit einem *Fragment-Shader* in Echtzeit gerendert. Die Eingabeparameter für das *Shader*-Programm bestehen im einfachsten Falle aus einer 3D-Textur, in der das Volumen gespeichert ist, und einer 1D-Textur, auf welche eine Transferfunktion abgebildet wird.

Im *Shader*-Programm wird die Transferfunktion auf das Volumen angewendet. Ein einfaches *Shader*-Programm ist in Codebeispiel 1 dargestellt. Dieses sampelt den Wert des Volumendatensatzes und klassifiziert ihn anhand einer eindimensionalen Transferfunktion.

3.5 Deformationssysteme

Die Implementierung der verschiedenen Deformationssysteme wurde jeweils für die CPU sowie für die GPU durchgeführt. Dadurch sollen sowohl die Vorteile als auch die Nachteile des jeweiligen Ansatzes herausgestellt werden.

Essentiell für die Deformation eines Volumendatensatzes mit Hilfe eines Deformationssystems auf der Grafikkarte ist die Art und Weise, in welcher die Deformation gespeichert wird. In der Software-Implementierung wird die Deformation des Volumens direkt in einer Textur gespeichert, damit diese dann in einem *Fragment-Shader* als Offset-Textur direkt auf das Volumen angewendet werden kann.

Im Folgenden wird zunächst der Ping-Pong-Ansatz vorgestellt, welcher notwendig ist, um Berechnungen auf der Grafikkarte durchzuführen. Anschließend werden die jeweiligen Implementierungen der Deformationssysteme vorgestellt und erläutert.

3.5.1 Ping-Pong-Ansatz

Die Notwendigkeit des Ping-Pong-Ansatzes liegt darin begründet, dass man Texturen in einem Schritt nur entweder auslesen oder beschreiben kann (GPGPU Programming Resources, 2004).

Zunächst erzeugt man ein *framebuffer object* (FBO). FBOs sind eine von der *Architectural Review Board (ARB) „superbuffers" Working Group* genehmigte Erweiterung zur *OpenGL API*. Sie repräsentieren eine Abstraktion des Grafik-Speichers, aus dem die GPU lesen und in die sie schreiben kann. Die Architektur von FBOs und ihren Komponenten wird in Abbildung 8 dargestellt.

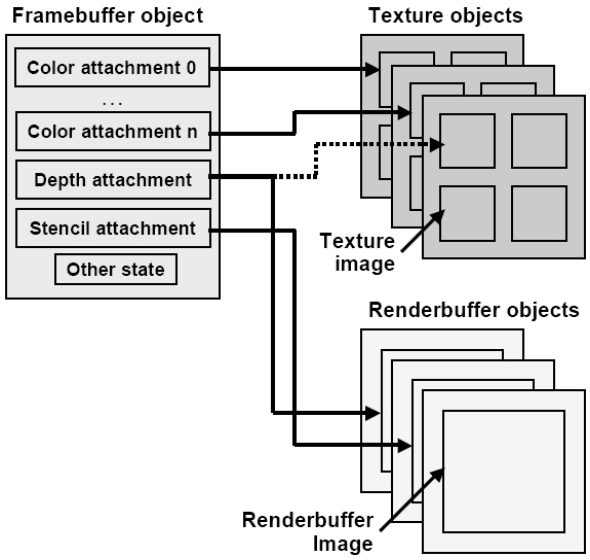


Abbildung 8 (Green, 2005)

Ein FBO kann bis zu 4 Texturen verwalten. Für den Ping-Pong-Ansatz werden nur zwei Texturen benötigt. Aus der einen Textur kann ausgelesen werden, in die andere kann geschrieben werden. Den Status der ersten Textur setzt man initial als *read-only* und den der zweiten als *write-only*. Die Quelldaten für die Berechnung werden in die erste Textur geladen. Es liegt nahe, die x-Koordinate eines Elementes auf die r-Komponente der Textur abzubilden, die y-Koordinate auf die g-Komponente und die z-Koordinate auf die b-Komponente. Nun wird die Berechnung anhand der Daten aus der ersten Textur in einem *Fragment-Shader*-Programm durchgeführt und das Ergebnis wird in die zweite Textur gerendert. Dieser Vorgang wird auch als *Render-To-Texture* bezeichnet. Im Anschluss muss der Status getauscht werden, woraufhin die nächste Berechnung durchgeführt werden kann. Auf gleiche Art und Weise kann eine beliebige Anzahl an Iterationsschritten erfolgen und die Textur jederzeit bei Bedarf ausgelesen werden.

Solange die Berechnungsphase auf der GPU andauert, muss die Textur nicht ausgelesen werden. Man kann die berechneten Daten folglich sofort darstellen. Daher müssen die Daten auch nicht von der Grafikkarte in den Hauptspeicher transferiert werden. Dies ist für mein Vorhaben ein wichtiger Schritt für eine effiziente Berechnung. Denn gerade der Datentransfer zwischen Grafikkarte und Hauptspeicher verursacht bei der Volumendeformation einen der Hauptengpässe, da permanent sehr viele Daten übertragen werden.

Der Ping-Pong-Ansatz ist die Grundlage der Berechnung eines Deformationsmodells auf der GPU. Im Folgenden werden die darauf basierenden Software-Implementierungen der Deformationsmodelle vorgestellt.

3.5.2 Masse-Feder-Modell

Die Implementierung von Masse-Feder-Systemen erfordert eine Repräsentation von Massen, Federn und globalen Eigenschaften in Quellcode.

Verschiedene Materialien können simuliert werden, indem für jede Feder bzw. für jede Masse eine entsprechende Federkonstante gewählt wird. Für die Federkonstanten wurde festgelegt, dass sie zwischen 0 und 1 liegen müssen. Der Wert 0 bedeutet hier, dass keine Dämpfung vorliegt und es sich damit um ein sehr elastisches Material handelt. Ein Wert von eins legt ein undeformierbares Material fest. Dadurch kann der Anforderung entsprochen werden, dass einzelne Elemente fixiert werden können, denn ihnen muss lediglich eine Federkonstante von 1 zugewiesen werden.

In der entwickelten Software wird wegen der hohen Effizienzansprüche das explizite Euler-Verfahren eingesetzt. Ein Nachteil des Verfahrens ist, dass bei einer ungünstigen Wahl der initialen Federeigenschaften bzw. der Schrittweite das gesamte System kollabieren kann. Die Kräfte können wegen des Fehlers bzw. wegen der numerischen Instabilität des Integrationsverfahrens unkontrollierbar wachsen.
Diesem Problem kann auf zwei Arten entgegenwirkt werden: Entweder durch eine genauere Berechnung mittels einer Verringerung der Schrittweite oder durch eine direkte Korrektur unerwünscht großer Werte.
Eine Verkleinerung der Schrittweite führt gleichzeitig dazu, dass mehr Rechenzeit benötigt wird. Dies kann zur Folge haben, dass die Interaktivität bzw.

Echtzeitfähigkeit der Anwendung leidet, da Rechenleistung nicht in beliebigem Maße zur Verfügung steht.
Eine Korrektur der Werte ist weniger präzise, führt aber zu visuell guten Ergebnissen und verursacht nur einen minimalen zusätzlichen Rechenaufwand. Eine mögliche Korrekturoption ist zum Beispiel das Setzen eines Maximums für den Betrag der Positionsänderung eines Massepunktes. Dadurch wird aber nicht automatisch ein stabiler Zustand des Masse-Feder-Systems erreicht. Diesen kann man künstlich herstellen, indem man die Berechnung abbricht, wenn ein zufriedenstellendes Ergebnis erzielt wurde.

Im Folgenden wird jeweils die CPU- und die GPU-Implementierung vorgestellt.

3.5.2.1 CPU

In der Software-Implementierung für diese Diplomarbeit wird für eine CPU-basierende Berechnung des Masse-Feder-Systems auf objektorientierte Strukturen zurückgegriffen. Die Entitäten *Masse* auf der einen Seite und *Feder* auf der anderen Seite werden auf Klassen abgebildet (*CMass* bzw. *CRod*), welche durch die Klasse *CMassSpringModel* verwaltet werden. So existiert für jede Masse ein *CMass*-Objekt und für jede Feder ein *CRod*-Objekt. Diese Objekte werden von der Objektinstanz der Klasse *CMassSpringModel* erzeugt.

Objekte vom Typ *CMass* enthalten Informationen über:

- Position der Masse
- Offset der Masse
- Kraft, die aktuell auf die Masse wirkt
- konstante Werte (Reibungskonstante)

Diese Attribute sind für die Berechnung ausreichend. Weitere Attribute wurden jedoch hinzugefügt, um Nutzerinteraktionen mit Massen zu ermöglichen. Jede Masse enthält beispielsweise Informationen darüber, ob sie fixiert, selektiert etc. ist.

Ein CRod-Objekt enthält folgende Informationen:

- Länge der Feder in Ruhelage
- je eine Referenz auf beide zugehörigen Massen
- Konstanten (Materialeigenschaften)

In einem Berechnungsdurchlauf stößt das *CMassSpringModel*-Objekt in einem ersten Schritt die Berechnung aller Federkräfte anhand der zugehörigen Massenpositionen an. In einem zweiten Schritt wird dann in Abhängigkeit von der Krafteinwirkung für jede Masse die Position aktualisiert.

Die wirkenden Federkräfte werden anhand der Positionen der dazugehörigen Massen ermittelt. Das bedeutet, dass die Kraft einer Feder direkt proportional zur Entfernung zwischen ihren Massen ist. Konkret bedeutet das, dass zu dem aktuellen Positionsoffset einer Masse im zweiten Schritt die Gesamtkraft addiert wird, wodurch die neue Position bestimmt ist.

Anfänglich wurden die Massen im Hinblick auf die Verwendung als Elemente einer Volumendeformation auf der Grafikkarte in einem Würfel uniform angeordnet, so dass jede Masse anteilsmäßig den gleichen Einfluss auf das Volumen hat.

3.5.2.2 GPU

Die numerische Integration bei Masse-Feder-Systemen ist unabhängig für jede Masse einzeln durchführbar. Daher eignen sich Masse-Feder-Systeme sehr gut zur parallelen Verarbeitung auf der Grafikkarte. Für die Berechnung der expliziten Euler-Schritte benötigt man lediglich die Position der jeweiligen Masse, sowie die Position der direkten Nachbarn. Aus diesem Grunde kann man auch große Masse-Feder-Systeme unter Verwendung des Ping-Pong-Ansatzes in Echtzeit berechnen. Dies erfordert, dass man die 3D-Darstellung des Deformationsmodells auf eine Textur abbildet.

Ein Nachteil bei einer Implementierung als *Shader*-Programm ist der große Speicherbedarf, den ein Masse-Feder-System benötigt. Außerdem ist die Implementierung auf der Grafikkarte komplizierter in der Handhabung als die auf der CPU. Für erstere ist der Befehlssatz der verwendeten *Shader*-Sprache sehr viel kleiner ist als für letztere. Folglich kann nicht die zuvor erläuterte Vorgehensweise gewählt werden.

Da in einem Rendering-Schritt implizit alle Elemente im *framebuffer* verarbeitet werden, braucht man keine Schleife mehr, in der man alle Massen bzw. Federn abarbeitet. Dies erledigt die Grafikkarte automatisch. Weil der Inhalt eines jeden Texels den Eigenschaften einer Masse entspricht, und die Topologie aufgrund der

Anordnung der Texel vollständig bekannt ist, kann man für jede Masse die auf sie wirkenden Federkräfte anhand der bekannten Nachbarmassen errechnen.

Zunächst müssen also alle Kräfte anhand der Nachbarpositionen ermittelt werden. Daraufhin wird der Mittelwert aller Kräfte als die resultierende Gesamtkraft errechnet. Die neue Position der Masse ergibt sich dann aus der aktuellen Position in Summe mit der Gesamtkraft.

Eine initiale Erzeugung von Federn ist hier also nicht nötig, da diese implizit gegeben sind. Dadurch lässt sich viel Speicher für die Verwaltung der Federstrukturen einsparen, da letztendlich nur eine Textur benötigt wird, in der alle Massen und damit auch Federn erfasst werden. Die Verwendung von impliziten Federn ist auch bei CPU-Implementierungen möglich, jedoch anders als hier, nicht erforderlich.

Laut der Anforderungen an diese Software soll die Möglichkeit bestehen, dass Massen verschiedene Attribute wie zum Beispiel *Unbeweglichkeit* haben können. In dem bisher noch nicht verwendeten Alpha-Kanal der Textur besteht die Möglichkeit, solche Attribute zu kodieren.

Ein Auszug des Quellcodes meiner *Shader*-Implementierung für das Masse-Feder-Modell ist in Codebeispiel 2 zu sehen.

3.5.3 ChainMail

Im Folgenden wird eine CPU- sowie eine GPU-Implementierung des ChainMail-Algorithmus' vorgestellt.

3.5.3.1 CPU

Der ChainMail-Algorithmus beruht auf der Idee, dass Elemente sich gegenseitig wie Kettenglieder beeinflussen. Diese anschauliche Vorstellung des Modells soll sich in der Implementierung wiederfinden. Daher gibt es eine Klasse *CChainMail*, welche die einzelnen Elemente vom Datentyp *CChainMailElement* verwaltet. Diese Elemente entsprechen den oben genannten Kettengliedern. Deren Erzeugung und Löschung wird durch die verwaltende Klasse sichergestellt. Außerdem stellt diese auch die Verknüpfung zwischen den einzelnen Elementen her.

Ein Objekt vom Typ *CChainMailElement* speichert Informationen über die Abstandsbedingungen zu seinen Nachbarn. Außerdem wird eine Referenz auf diese gesetzt. Konkret werden die Abstandsbedingungen in folgenden Variablen gespeichert:

- m_fMinDx und m_fMaxDx
- m_fMinDy und m_fMaxDy
- m_fMinDz und m_fMaxDz

} minimaler und maximaler Abstand pro Achse

- m_fShearDx, m_fShearDy und m_fShearDz } maximale Scherung

Zudem wird gespeichert, welches Element als Sponsor-Element im Fall einer Bewegung agiert, und ob das Element beweglich oder fixiert ist.

Bei der Berechnung des Algorithmus' müssen für jedes verarbeitete Element zwei Fälle unterschieden werden:

1. Das Element hat die Bewegung ausgelöst: In diesem Fall werden alle Nachbarelemente in eine Liste gehängt und das auslösende Element wird als Sponsor gesetzt. Diese Liste muss abgearbeitet werden, um sicherzustellen, dass kein Element die Abstandsbedingungen verletzt.

2. Das Element wurde von einem Nachbarn gesponsort: Nun muss für das Element geprüft werden, ob es die Abstandsbedingungen zu seinen Nachbarn einhält. Werden diese verletzt, so muss die Position dahingehend angeglichen werden, dass die Bedingungen wieder erfüllt sind. Außerdem müssen alle vorhandenen Nachbarelemente in die Liste gehängt werden und das nun auslösende Element als Sponsor vermerkt werden.

3.5.3.2 GPU

Bei der Implementierung des ChainMail-Algorithmus' auf der Grafikkarte treten Probleme auf, die bei einer CPU-basierenden Lösung so nicht vorkommen. Die dafür verantwortlichen Aspekte sind zum einen der im Vergleich mit dem Hauptspeicher kleinere Grafikspeicher und zum anderen die stark eingeschränkten Möglichkeiten der *Shader-S*prache.

Das Speicherproblem ist relevant, weil – wie auch beim Masse-Feder-Modell – zahlreiche Elemente gespeichert werden müssen. Im optimalen Fall müsste man auf jedes Voxel der Volumen-Textur ein Element des ChainMail-Modells abbilden, da so die größtmögliche Genauigkeit erzielt würde. Wie schon oben beschrieben, kann allein der Gebrauch des für die Volumenvisualisierung benötigten Speichers bereits zur kompletten Auslastung führen. Die Konsequenz für meine Implementierung ist, dass bewusst darauf geachtet wurde, keinen unnötigen Speicher zu nutzen. Da das Deformationsmodell zur Zeit der Berechnung auf der Grafikkarte vorliegen muss, werden alle Informationen in Form von einer Textur gespeichert. Die Offsets, die während einer Deformation für einzelne Elemente entstehen, sind auf die Farbkomponenten abgebildet. Die Attribute eines Elementes, wie z.B. *Fixierung* oder *Sponsor-Element*, werden im Alpha-Kanal kodiert hinterlegt. Dadurch wird nur eine Textur benötigt, in der alle relevanten Informationen enthalten sind. Auf die Verwendung weiterer Texturen wird komplett verzichtet, wodurch eine beachtliche Einsparung von Speicherplatz erzielt wird.

Nachbarschaftsbeziehungen müssen nicht zusätzlich gespeichert werden, da sie implizit gegeben sind. Der Grund dafür ist, wie oben beschrieben, dass die ChainMail-Elemente uniform auf einem Würfel verteilt angeordnet sind (siehe Abbildung 9).

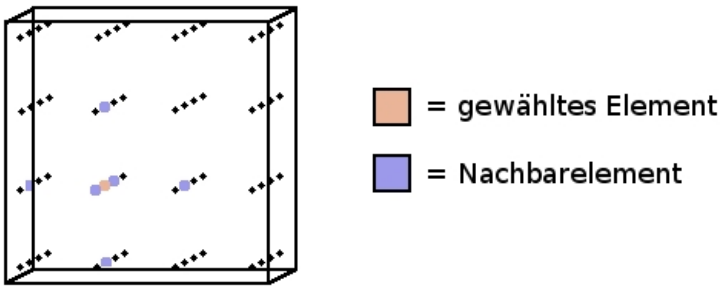


Abbildung 9

In Abbildung 10 wird veranschaulicht, wie die Anordnung der ChainMail-Elemente innerhalb einer Offset-Textur realisiert wird. Ein gewähltes Element ist rot markiert und dessen direkte Nachbarn blau. Seine Nachbarn in x- und y-Richtung sind, im Gegensatz zu denen in z-Richtung, in der Textur direkt neben dem gewählten Element zu finden. Wegen der Abbildung der einzelnen Schichten

mit unterschiedlichem z-Wert auf die 2D-Textur müssen im *Shader* die Nachbarn teilweise auf unterschiedliche Art und Weise ermittelt werden. Für ein Modell mit nxnxn Elementen muss bei den Nachbarelementen in y- bzw. z-Richtung jeweils im Abstand von $\frac{1}{n}$ gesampelt werden, um die entsprechenden Offsets auszulesen. In x-Richtung beträgt der Abstand $\frac{1}{n^2}$. Das Sampling ist nur möglich, da die Anzahl der Elemente je Richtung sowie die Topologie der Textur bekannt ist.

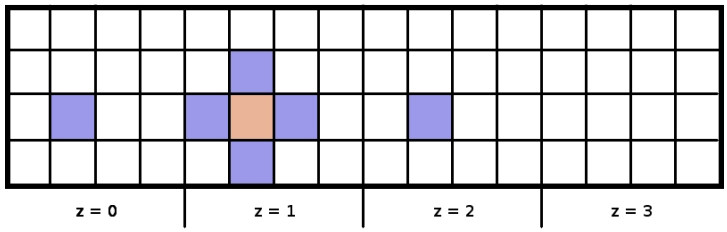


Abbildung 10

Die Klassenstruktur innerhalb der GPU-basierenden ChainMail-Implementierung ist dieselbe wie in der CPU-Implementierung (siehe Kapitel 3.5.3.1).

Ein Auszug der relevanten Teile des *Shaders* für die Berechnung des ChainMail-Algorithmus‘ ist in Codebeispiel 3 dargestellt.

3.6 Volumendeformation

Die Volumendeformation ist als Kombination von Deformation und Visualisierung zu verstehen. Sie bedingt die Fixierung einzelner Elemente für das Deformationssystem.

In meiner Software-Implementierung habe ich mich dafür entschieden, alle Eckpunkte des Deformationsmodells standardmäßig zu fixieren. Dadurch wird gewährleistet, dass es nur genau einen Zustand gibt – nämlich den Ursprungszustand - in dem keine Kräfte wirken. Würden diese Elemente nicht fixiert, so könnte sich das gesamte Volumen in Deformationsrichtung bewegen und schließlich außerhalb des sichtbaren Bereichs gelangen. Damit würde die eigentliche Deformation zu einer Translation werden, was nicht erwünscht ist. Es ist jedoch jederzeit für jedes Element eine Änderung des Fixierungs-Status möglich.

Das Ergebnis der Deformationsberechnung (sowohl per GPU als auch CPU) wird in einer Textur abgespeichert. Diese Textur dient als Grundlage, um in einem *Shader*-Programm die Deformation auf Volumendaten anwenden zu können und somit sichtbar zu machen. Details werden im folgenden Kapitel beschrieben.

3.6.1 Offset-Textur

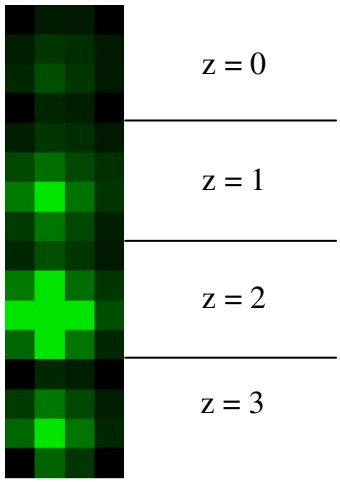


Abbildung 11

In der Offset-Textur wird die gesamte Information der Deformation zu einem bestimmten Zeitpunkt gespeichert. Die Textur wird im 16-Bit Floating-Point-Format erzeugt. Damit ist eine hinreichend genaue Berechnung gewährleistet.

Jedes Texel entspricht einem Element im Deformationssystem. Daher ist der Aufbau der Offset-Textur analog zu dem der Volumentexturen. Der Inhalt eines Texels gibt Auskunft über den Offset, also über die Verschiebung im Volumen an der jeweiligen Stelle. In der rot-Komponente wird der x-Offset, in der grün-Komponente wird der y-Offset und in der blau-Komponente der z-Offset abgespeichert. Der Alpha-Kanal enthält Informationen über die Dichte des Volumens an dieser Stelle. Eine Materialkonstante wird verwendet, um die Dichte abzubilden. Auf diese Art und Weise lassen sich verschiedene Materialien simulieren.

Abbildung 11 zeigt eine 4x4x4-Offset-Textur, wobei je ein 4x4-Block eine Schicht mit konstantem z-Wert darstellt. Die Deformation der Elemente ist anhand der grünen Färbung zu erkennen. Dadurch kann man sehen, dass eine Deformation in y-Richtung erfolgt ist, weil in der grün-Komponente der Offset von Elementen in y-Richtung enthalten ist.

Für die Darstellung der Volumendeformation wurde der Visualisierungs-*Shader* erweitert. Ein Auszug des *Shaders* für die Volumendeformation ist in Codebeispiel 4 abgebildet. Inhaltlich ist der *Shader* wie folgt aufgebaut:

- Ermittlung des Offsets per Interpolation zwischen den jeweiligen Schichten der Offset-Textur in z-Richtung
- Subtraktion des ermittelten Offsets von den Texturkoordinaten
- Sampling der Volumentextur an der Stelle des Ergebnis-Offsets
- Klassifikation des Volumens anhand einer Transferfunktion

In x- und y-Richtung wird automatisch Interpoliert, da lineare Interpolation in *OpenGL* aktiviert wurde. Da das Deformationssystem aber in einer 2D Textur gespeichert wird, ist die Ermittlung des Offsets per Interpolation erforderlich, da sonst bei der Darstellung der Deformation visuelle Artefakte in z-Richtung sichtbar sind.
Wird ein Deformations-Element von einem Punkt A im Volumen zu einem Punkt B im Volumen verschoben, so muss am Punkt B der Inhalt des Punktes A angezeigt werden. Daher muss der Offset des Deformationselementes im *Shader* von der Texturkoordinate abgezogen werden, denn an der daraus resultierenden Stelle muss gesampelt werden.
Wie im ursprünglichen Volumenvisualisierungs-*Shader* wird dann der entsprechende Wert des Volumens anhand einer Transferfunktion klassifiziert.

4 Ergebnisse

4.1 Einleitung

In diesem Kapitel werden zunächst die Resultate der beiden in der Software implementierten Deformationsmodelle analysiert. Dabei stehen der Vergleich der Leistungsfähigkeit der jeweiligen CPU- bzw. GPU-Lösung und die Untersuchung der Unterschiede der Deformationsmodelle selbst im Fokus. Anschließend werden die Ergebnisse des Volumendeformations-Ansatzes vorgestellt. In einer Zusammenfassung werden die Auswirkungen der Ergebnisse auf die Volumendeformation erarbeitet und bewertet.

4.2

4.3 Masse-Feder-Modell

Das Masse-Feder-Modell liefert visuell ansprechende Ergebnisse. Bei einer Implementierung auf der CPU werden jedoch relativ schnell die Grenzen der Leistungsfähigkeit ausgereizt. Eine große Anzahl an Massen bewirkt, dass eine Echtzeit-Deformation nicht mehr möglich ist. Die Implementierung auf der Grafikkarte ist im Vergleich zu der CPU-Variante sehr viel effizienter.
Der Vorteil der auf Parallelverarbeitung ausgerichteten GPU kommt hier voll zum Tragen. Im Vergleich zur CPU-Implementierung ist die Berechnung von deutlich komplexeren Masse-Feder-Systemen mit interaktiven *Framerates* möglich.
Ein Nachteil dieser Lösung ist, dass eine Begrenzung durch den relativ hohen Bedarf an Speicherkapazitäten gegeben ist, da der Grafikspeicher im Vergleich zum Hauptspeicher noch relativ klein ist.

4.4 ChainMail

Der ChainMail-Algorithmus ist ein sehr effizienter Deformations-Algorithmus. Es werden ausschließlich Elemente in die Berechnung mit einbezogen, welche auch eine Änderung erfahren und jedes Element wird maximal einmal verwertet. Weil immer nur die Elemente betrachtet werden müssen, die die Anforderungen des Algorithmus‘ verletzen, ist bei lokalen Deformationen die Größe des Deformationssystems unerheblich. Dieser entscheidende Vorteil kann nicht bei der Berechnung auf der Grafikkarte ausgeschöpft werden, weil die Grafikkarte *per se* alle Elemente des Modells bearbeitet. Die GPU-Nutzung für das ChainMail-

Modell ist dennoch von Vorteil gegenüber der CPU-Lösung, da der Geschwindigkeitsvorteil der GPU auch hier entscheidend genutzt werden kann.

Visuell sind die erzielten Ergebnisse im Vergleich zu denen des Masse-Feder-Modells verhältnismäßig unbefriedigend. Dies liegt unter anderem daran, dass der ChainMail-Algorithmus geometrie-basiert arbeitet. Die dabei entstehenden geometrischen Strukturen sind oft sichtbar, was mitunter für den Menschen unnatürlich wirkt. Eine Lösung dieses Problems kann durch gezielte *Relaxation*-Schritte gefunden werden.

4.5 Volumendeformation

In den Überlegungen, die dieser Arbeit vorausgingen, war noch unklar, ob eine Umsetzung von direktem *Volume-Rendering* in Kombination mit einer Deformation überhaupt möglich ist. Meine Entwicklung beweist, dass dies unabhängig vom gewählten Deformationssystem gut funktioniert.

Die gewählte Lösung in der Software erfordert lediglich, dass das Ergebnis des Deformationsmodells in Form einer Offset-Textur zurückgegeben wird. Dieses Ergebnis kann dann in Echtzeit für die Volumenvisualisierung genutzt werden, wodurch eine aufwändige Vorverarbeitung vermieden werden kann.

4.6 Vergleich

Die Handhabung der Deformation mit dem ChainMail-Algorithmus gestaltet sich schwieriger als die mit dem Masse-Feder-Modell. Visuell werden mit ersterem zunächst unbefriedigende Ergebnisse erzielt. Die mit Masse-Feder-Systemen durchgeführten Volumendeformationen entsprechen in stärkerem Maße den menschlichen Sehgewohnheiten und weisen größere Analogien zu real auffindbaren Materialien vor.

Im Folgenden werden die Deformationen des Masse-Feder-Modells und des ChainMail-Modells anhand von Abbildungen miteinander verglichen. Die Simulation der Eigenschaften verschiedener Materialien konnte im Rahmen dieser Arbeit nicht genauer Untersucht werden. Aus diesem Grunde ist der Vergleich der Ergebnisse nur sehr eingeschränkt möglich.

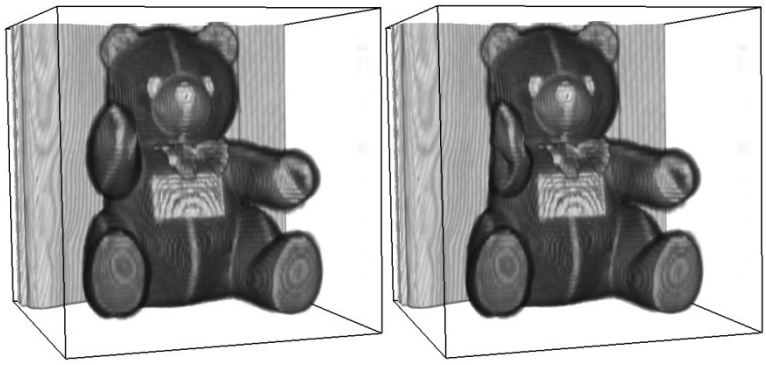


Abbildung 12

Abbildung 12 zeigt auf der linken Seite ein Volumen im Ursprungszustand. Auf der rechten Seite ist das Ergebnis der Deformation zu sehen, bei dem ein Element gewählt und nach rechts verschoben wurde. Die Deformation zeigt sich hauptsächlich im Bereich des Arms.

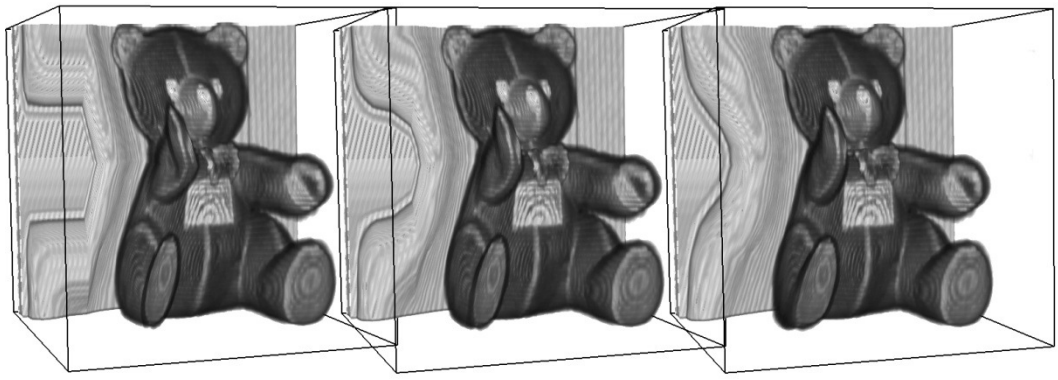


Abbildung 13

In Abbildung 13 ist links das Ergebnis einer Deformation mit dem ChainMail-Modell dargestellt. Es wurde das Element gewählt, welches der gewählten Masse in Abbildung 12 entspricht und dieses wurde dann in die gleiche Richtung verschoben. In den Auswirkungen der Deformation werden geradlinige Strukturen im Hintergrund sichtbar. In der Mitte sieht man die gleiche Deformation, in der aber noch ein zusätzlicher *Relaxation*-Schritt durchgeführt wurde. Auf der rechten Seite wurden insgesamt vier *Relaxation*-Schritte durchgeführt. Schon nach einem Schritt ist eine deutliche Änderung erkennbar. Offensichtlich werden durch die *Relaxation*-Schritte die geradlinigen Strukturen, welche die ChainMail-Deformation liefert, aufgelöst.

Der ChainMail-Algorithmus scheint auf den ersten Blick effizienter als Masse-Feder-Systeme zu sein, da er nur in der lokalen Umgebung der Deformation

Berechnungen durchführen muss. Daher sind selbst bei einer CPU-Implementierung interaktive *Framerates* kein Problem, so lange die Deformation in der lokalen Umgebung des gewählten Elements stattfindet. Dieser Vorteil kann jedoch in der GPU-Implementierung nicht genutzt werden, da die GPU ohnehin für jedes Element einzeln berechnet. Es wird also immer die maximale Anzahl an ChainMail-Berechnungen durchgeführt. Daher ist die Geschwindigkeit der Berechnung auf der Grafikkarte konstant. Die Vorzüge des Algorithmus können nicht – wie in der CPU-Implementiereung – effizient genutzt werden.

4.7 Probleme

Folgende Kernprobleme haben sich im Laufe der Softwareentwicklung für diese Arbeit ergeben:

- Realitätstreue
- Unausgereifte *Shader*
- Großer Ressourcenbedarf

Eine realitätsgetreue Simulation der Volumendeformation hängt nicht nur maßgeblich von der Wahl des Deformationsmodells ab. Es muss berücksichtigt werden, dass dafür eine möglichst naturgetreue Nachbildung von Materialien notwendig ist. In der entwickelten Software besteht zwar die Möglichkeit, verschiedene Materialien zu simulieren, nach welchen Kriterien die Materialabhängigen Werte aber gesetzt werden müssen, bleibt im einzelnen näher zu erforschen.

Bei der Implementierung von GPU-basierten Deformationsalgorithmen hat sich gezeigt, dass die *Shader*-Entwicklung noch fehleranfällig ist, da fast keine *Debugging*-Möglichkeiten bestehen. Die Programmierung auf *Shader*-Ebene hängt noch sehr stark von der jeweiligen Hardware ab. Dies wird zum Beispiel daran deutlich, dass Verzweigungen mit *if-Schleifen* bis vor kurzem in *Shader*-Programmen noch nicht möglich waren – in herkömmlicher Programmierung ist dies ein de-facto-Standard.
Des Weiteren ist die Lauffähigkeit der *Shader*-Programme noch stark an das jeweilige Grafikkartenmodell gebunden, wodurch eine Entwicklung auf verschiedenen Rechnern sich als sehr schwierig herausgestellt hat. Da *Shader-S*prachen im Vergleich zu objektorientierten Hochsprachen, „wie z.B. *C++*“ im

Funktionsumfang wesentlich eingeschränkter sind, mussten für die Umsetzung der GPU-basierten Verfahren alternative Verfahren und Lösungsansätze entwickelt werden.

Auf Grund des großen Speicherbedarfs von 3D-Volumina und Deformationsmodellen ist die Obergrenze des Grafikspeichers schnell erreicht. In der Konzeption der Entwicklung musste dieser Engpass stets Berücksichtigung finden.

5 Fazit

Die im Rahmen dieser Diplomarbeit entwickelte Software zeigt, dass eine Implementierung der Deformationsalgorithmen auf der Grafikkarte möglich ist. Ferner können die Ergebnisse der Deformation mit dem Visualisierungsverfahren verknüpft werden, wodurch eine Volumendeformation in Echtzeit berechnet werden kann. Da diese Methode auf direktem *Volume-Rendering* beruht, ist eine aufwändige Vorverarbeitung der Volumendaten nicht mehr nötig. Somit kann das Ergebnis einer Volumendeformation in einer verlustfreien Darstellung dem Anwender verfügbar gemacht werden.

Zwar hängt die Wahl eines geeigneten Deformationssystems von der jeweiligen Anwendung ab. Es hat sich aber gezeigt, dass bereits heute auch aufwändigere und realistischere Modelle mit diesem Ansatz umgesetzt werden können. Gerade für medizinische Simulationen, aber auch in anderen Bereichen spielt nicht nur die visuelle Qualität eine Rolle. Genauigkeit und Realitätsnähe sind häufig ein sehr wichtiges Kriterium bei Volumendeformation. Der in dieser Arbeit vorgestellte Ansatz demonstriert, dass man bereits heute den hohen und stetig steigenden Ansprüchen an Volumendeformation auch mit preiswerter Standard-Hardware gerecht werden kann.

5.1 Ausblick

Die Skalierbarkeit der vorgestellten Volumendeformations-Methode wird zum momentanen Stand der Technik maßgeblich durch die Kapazitäten des Grafikspeichers bestimmt. Die progressive Entwicklung von Grafikkarten lässt zukünftig eine noch bessere Anwendung der vorgestellten Verfahren erwarten, da neue Computer und im Besonderen deren Grafikkarten stets leistungsfähiger werden und deren Speicherkapazitäten wachsen. Vor diesem Hintergrund sollten einige Probleme, die im Verlaufe der Entwicklung dieser Arbeit aufkamen, von selbst an Relevanz verlieren.
Die hier beschriebenen Verfahren können aber im Vorfeld und auf Dauer die technische Entwicklung der Hardware durch eine gezielte Optimierung der Software ergänzen.

Ein Beispiel dafür ist eine intelligentere Speicherverwaltung, die z.B. sowohl das Volumen als auch die Deformationsmodelle in kleinere Einheiten aufteilt, um ein besseres *load-balancing* zu erreichen.

In weiterführenden Ansätzen können durch eine größere Anzahl von Texturen mehr Informationen in eine Deformationsberechnung einbezogen werden, um weitere Attribute oder Features in einer GPU-basierten Lösung zu nutzen.
Die stetige Weiterentwicklung der *Shader*-Technologien wird in Zukunft durch neue Funktionen die Programmierung weiter erleichtern und die Vielfalt der Möglichkeiten der GPU-basierten Verfahren erhöhen. Aus den Beobachtungen der letzten Entwicklungen lässt sich prognostizieren, dass auch die Kompatibilität der *Shader*-Sprachen mit verschiedener Grafik-Hardware verbessert wird.

In der von mir entwickelten Software wird ein deformiertes Volumen dargestellt, welches um Beleuchtungsberechnungen erweitert werden kann.
Bezüglich der Volumendeformation ist eine Erweiterung der Eingabe-möglichkeiten denkbar. Es könnten Virtual-Reality-Ansätze genutzt werden, um zum Beispiel virtuelle Operationen mit haptischen Werkzeugen simulieren zu können.
Aufgrund des Aufbaus der Software ist für die Art der Deformation jedes Deformationssystem potentiell geeignet, welches als Offset-Textur abgebildet werden kann.
Meine Erwartungen, dass dieser Ansatz auch für beliebige weitere Deformationssysteme genutzt werden kann, werden hier bestätigt.
Eine hybride Nutzung verschiedener Deformationssysteme ist ebenso denkbar. Für die Beispiele in dieser Arbeit muss beachtet werden, dass das Ergebnis einer Masse-Feder-Berechnung in der Regel nicht die geometrischen Kriterien des ChainMail-Modells erfüllt. Ein etwaiges Umschalten zwischen den Modellen muss also entsprechend behandelt werden. Umgekehrt kann von ChainMail zur Nutzung des Masse-Feder-Modells theoretisch ohne Probleme umgeschaltet werden, da das Masse-Feder-Modell keine Einhaltung geometrischer Restriktionen erfordert. In der Konzeption der Software für diese Arbeit wurde diese Möglichkeit insofern berücksichtigt, dass ChainMail-Elemente den gleichen Typ wie Massen haben und auch die Deformationssysteme von einem gemeinsamen Typ abstammen.

Eine – auch für die oben genannte hybride Nutzung – erforderliche Speicherung der Deformation kann unabhängig vom Volumen getätigt werden.

Die Ergebnisse dieser Arbeit zeigen, dass die visuellen Nachteile des ChainMail-Algorithmus durch gezielte *Relaxation*-Schritte weitestgehend kompensiert werden. Dies könnte durch die Wahl eines anderen *Relaxation*-Verfahrens verbessert werden. Denkbar ist zum Beispiel, dass nur die Elemente behandelt werden, die durch den ChainMail-Algorithmus eine Deformation erfahren. Dadurch würde eine größere Performanz erreicht.

Die vorangestellten Überlegungen demonstrieren, dass das Erweiterungspotential der direkten Volumendeformation als ausgelagerter Prozess auf der Grafikkarte eindeutig vorhanden ist. Die weitere Erforschung dieser Methodik bei gleichzeitiger Adaption auf die jeweils aktuelle Hardware lassen weitere Möglichkeiten beim Einsatz von Standard-Hardware erwarten.

6 Literaturverzeichnis

Cabral, B., Cam, N., & Foran, J. (1994). Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Symposium on Volume Visualization* , S. 91-98.

Fehlberg, E. (1969). Klassische Runge-Kutta-Formeln 5. und 7. Ordnung mit Schrittweitenkontrolle. *Computing* , S. 93-106.

Gibson, S. F. (1997). 3D ChainMail: a Fast Algorithm for Deforming Volumetric Objects. In *Symposium on Interactive 3D Graphics* (S. 149-154).

Gouraud, H. (Juni 1971). Continuous Shading of Curved Surfaces. (I. C. Society, Hrsg.) *IEEE Transactions on Computers* , S. 623-628.

GPGPU Programming Resources. (Juni 2004). Abgerufen am 22. Februar 2008 von Sourceforge.net: http://sourceforge.net/projects/gpgpu

Green, S. (2005). *The OpenGL Frambeuffer Object Extension.* Abgerufen am 23. Februar 2008 von NVidia.com: http://http.download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf

Hege, H., Höllerer, T., & Stalling, D. (1993). Volume Rendering - Mathematical Models and Algorithmic Aspects. *Technical Report TR 93-7.* Berlin: Konrad-Zuse-Zentrum für Informationstechnik.

Lacroute, P., & Levoy, M. (Juli 1994). Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Proc. SIGGRAPH '94* , S. 451-458.

Levoy, M. (Mai 1988). Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications* , S. 29-37.

Li, Y., & Brodlie, K. (2003). Soft Object Modelling with Generalised ChainMail. In *Extending the Boundaries of Web-based Graphics* (S. 717-728). Blackwell Publishing.

Phong, B. (Juni 1975). Illumination for Computer Generated Pictures. *Communications of the ACM* , S. 311-317.

Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., & Ertl, T. (2000). Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures und Multi-Stage Rasterization. *SIGGRAPH/Eurographics Workshop on Graphics Hardware* , S. 109-118.

Schill, M. A., Gibson, S. F., Bender, H.-J., & Reinhard, M. (1998). Biomechanical Simulation of the Vitreous Humor in the Eye Using an enhanced-ChainMail algorithm . *Proceedings of Medical Image Computation and Computer Integrated Surgery* , S. 679-687.

Wilson, O., VanGelder, A., & Wilhelms, J. (1994). Direct volume rendering via 3D textures. *Technical Report UCSC-CRL-94-19* .

7 Anhang

```
01      uniform sampler3D volume : TEXTURE0;
02
03      void main (         double3 str : TEXCOORD0,
04                          uniform sampler1D colortable,
05                          out double4 oColor : COLOR)
06      {
07              // volume-sampling
08              double4 value = tex3D(volume,str);
09              // calculate color by multiplying
10              oColor = tex1D(colortable,value.a);
11      }
```

Codebeispiel 1

```
01	half4 main(
02		float2 uv : TEXCOORD0,
03		float4 color : COLOR,
04		uniform sampler2D offset,
05		uniform float dim,
06		uniform float epsilon) : COLOR {
07
08		//current offset
09		offsetCenter = tex2D(offset,uv);
10		// check if masses are fixed
11		if (offsetCenter.w > 1.0-epsilon)
12			output = half4(offsetCenter.x,offsetCenter.y,offsetCenter.z,1.0);
13		// get offset of each neighbours
14		else {
15			// get left offset
16			if (uv.x > 1.0/dim) {
17				uvLeft = float2(uv.x - 1.0/dim, uv.y);
18				offsetLeft = tex2D(offset,uvLeft);
19			}
20			else
21				offsetLeft = float4(0.0,0.0,0.0,0.0);
22	/**
23	// …get offsets for right, bottom, top and rear
24	**/
25			// get front offset
26			if (uv.y < 1.0-1.0/dim) {
27				uvFront = float2(uv.x, uv.y + 1.0/dim);
28				offsetFront = tex2D(offset,uvFront);
29			}
30			else
31				offsetFront = float4(0.0,0.0,0.0,0.0);
32
33			// compute forces
34			if (offsetLeft.w < epsilon)
35				springLeft = (offsetLeft-offsetCenter)/2.0;
36			else
37				springLeft = offsetLeft-offsetCenter;
38	/**
39	//… compute the forces for right, bottom, top and front
40	**/
41			if (offsetRear.w < epsilon)
42				springRear = (offsetRear-offsetCenter)/2.0;
43			else
44				springRear = offsetRear-offsetCenter;
45
46			// Sum up forces
47			force = half4(springLeft.xyz*(offsetLeft.w+offsetCenter.w)/2.0 +
48				springRight.xyz*(offsetRight.w+offsetCenter.w)/2.0 +
49				springTop.xyz*(offsetTop.w+offsetCenter.w)/2.0 +
50				springBottom.xyz*(offsetBottom.w+offsetCenter.w)/2.0 +
51				springFront.xyz*(offsetFront.w+offsetCenter.w)/2.0 +
52				springRear.xyz*(offsetRear.w+offsetCenter.w)/2.0,0.0);
53			force*=(1.0-offsetCenter.w);
54			output = offsetCenter+3.0/dim * force;
55			output.w = offsetCenter.w;
56		}
57
58		return half4(output.x,output.y,output.z,output.w);
59	}
```

Codebeispiel 2

```
01	half4 main(
02		float2 uv : TEXCOORD0,
03		float4 color : COLOR,
04
05		uniform sampler2D offset,
06		uniform float dim,
07		uniform float epsilon) : COLOR {
08
09		//current offset
10		offsetCenter = tex2D(offset,uv);
11
12		// check if masses are fixed
13		if (offsetCenter.w > 1.0-epsilon)
14			output = half4(offsetCenter.x,offsetCenter.y,offsetCenter.z,1.0);
15
16		// get offset of each neighbours
17		else {
18			// get left offset
19			if (uv.x > 1.0/dim) {
20				uvLeft = float2(uv.x - 1.0/dim, uv.y);
21				offsetLeft = tex2D(offset,uvLeft);
22			}
23			else
24				offsetLeft = float4(0.0,0.0,0.0,0.0);
25	/**
26	// …get offsets for right, bottom, top and rear
27	**/
28			// get front offset
29			if (uv.y < 1.0-1.0/dim) {
30				uvFront = float2(uv.x, uv.y + 1.0/dim);
31				offsetFront = tex2D(offset,uvFront);
32			}
33			else
34				offsetFront = float4(0.0,0.0,0.0,0.0);
35
36			// compute forces
37			if (offsetLeft.w < epsilon)
38				springLeft = (offsetLeft-offsetCenter)/2.0;
39			else
40				springLeft = offsetLeft-offsetCenter;
41	/**
42	//… compute the forces for right, bottom, top and front
43	**/
44			if (offsetRear.w < epsilon)
45				springRear = (offsetRear-offsetCenter)/2.0;
46			else
47				springRear = offsetRear-offsetCenter;
48
49			// Sum up forces
50			force = half4(springLeft.xyz*(offsetLeft.w+offsetCenter.w)/2.0 +
51				springRight.xyz*(offsetRight.w+offsetCenter.w)/2.0 +
52				springTop.xyz*(offsetTop.w+offsetCenter.w)/2.0 +
53				springBottom.xyz*(offsetBottom.w+offsetCenter.w)/2.0 +
54				springFront.xyz*(offsetFront.w+offsetCenter.w)/2.0 +
55				springRear.xyz*(offsetRear.w+offsetCenter.w)/2.0,0.0);
56			force*=(1.0-offsetCenter.w);
57			output = offsetCenter+3.0/dim * force;
58			output.w = offsetCenter.w;
59		}
60
61		return half4(output.x,output.y,output.z,output.w);
62	}
```

Codebeispiel 3

```
01      float4 main(
02              float3 uvw : TEXCOORD0,
03
04              uniform sampler3D volume,
05              uniform sampler1D colortable,
06              uniform sampler2D Deformation,
07              uniform float dim   ) : COLOR {
08
09                      float Z    = -0.5 +  dim *float(uvw.z);
10
21                      float floorZ = max(0.0,floor(Z));
22                      float ceilZ  = min(ceil(Z),dim-1);
23                      float alpha = Z - floorZ;
24
25                      float fracY = modulo(max(0.0,min(dim*uvw.y, (dim))),dim)/dim;
26                      fracY = max( 1.0/(2.0*dim), min( (2.0*dim-1)/(2.0*dim), fracY));
27
28                      float floorV = max(0.0,min(1.0,(floorZ + fracY)/dim));
29                      float ceilV  = max(0.0,min(1.0,(ceilZ  + fracY)/dim));
30
31                      float2 uv1 = float2 (uvw.x,  floorV);
32                      float2 uv2 = float2 (uvw.x,  ceilV);
33
34                      // Offsets aus Deformationstextur holen
35                      float4 offset1 = tex2D(Deformation,uv1);
36                      float4 offset2 = tex2D(Deformation,uv2);
37                      float4 offset = lerp(offset1,offset2,alpha);
38
39                      // offset von Ursprungskoordinaten subtrahieren
40                      float3 sum = uvw - offset.xyz;
41
42                      // Volume abtasten
43                      float4 value = tex3D(volume,sum);
44
45                      // Transferfunktion anwenden
46                      float4 color = tex1D(colortable,value.a);
47
48                      return color;
49              }
```

Codebeispiel 4